Zionomicon

John De Goes and Adam Fraser

Contents

Chapter 1

Foreword by John A. De Goes

We live in a complex and demanding cloud-native. The applications that we develop are small parts of a much larger, globally distributed whole.

Modern applications must process a never-ending stream of new data and requests from all around the world, interacting with hundreds or even thousands of other services, remotely distributed across servers, racks, data centers and even cloud providers.

Modern applications must run 24x7, deal with transient failures from distributed services, and respond with ultra low latency as they process the data and requests from desktops, smartphones, watches, tablets, laptops, devices, sensors, APIs, and external services.

The complexity of satisfying these requirements gave birth to *reactive programming*, which is a style of designing applications so they are responsive, resilient, elastic, and event-driven.

This is the world that created *ZIO*, a new library that brings the power of functional programming to deliver a powerful new approach to building modern applications.

1.1 A Brief History of ZIO

On June 5th, 2017, I opened an issue in the *Scalaz* repository on Github, arguing that the next major version of this library needed a powerful and fast data type for asynchronous programming. Encouraged to contribute to Scalaz by my friend Vincent Marquez, I volunteered to build one.

This was not my first foray into the space of asynchronous computing. Previously, I had designed and built the first version of *Aff*, and assisted the talented Nathan Faubion with the second iteration.

The *Aff* library quickly became the number one solution for async and concurrent programming in the Purescript ecosystem, offering powerful features not available in Javascript Promises.

Prior to *Aff*, I had written a Future data type for Scala, which I abandoned after the Scala standard library incorporated a much better version. Before that, I wrote a Promise data type for the haXe programming language, and a Raincheck data type for Java.

In every case, I made mistakes, and subsequently learned... to make new and different mistakes!

That sunny afternoon in June 2017, I imagined spending a few months developing this new data type, at which point I would wrap it up in a tidy bow, and hand it over to the Scalaz organization.

What happened next, however, I never could have imagined.

1.2 The Birth of ZIO

As I spent free time working on the core of the new project, I increasingly felt like the goal of my work should not be to just create a pure functional effect wrapper for asynchronous side-effects.

That had been done before, and while performance could be improved over Scalaz 7, libraries built on pure functional programming cannot generally be as fast as bare-metal, abstraction-free, hand-optimized procedural code.

Now, for developers like me who are sold on functional programming, an effect-wrapper is enough, but if all it does it slap a *Certified Pure* label on imperative code, it's not going to encourage broader adoption. Although solving the async problem is still useful in a pre-Loom world, lots of other other data types already solved this problem, such as Scala's own Future.

Instead, I thought I needed to focus the library on concrete pains that are well-solved by functional programming, and one pain stood out among all others: concurrency including the unsolved problem of how to safely cancel executing code whose result is no longer needed, due to timeout or other failures.

Concurrency is a big space. Whole libraries and ecosystems have been formed to tame its wily ways. Indeed, some frameworks become popular precisely because they shield developers from concurrency, because it's complex, confusing, and error-prone.

Given the challenges, I thought I should directly support concurrency in this new data type, and give it features that would be impossible to replicate in a

procedural program without special language features.

This vision of a powerful library for safe concurrent programming drove my early development.

1.3 Contentious Concurrency

At the time I began working on the Scalaz 8 project, the prevailing dogma in the functional Scala ecosystem was that an effect type should have little or no support for concurrency.

Indeed, some argued that effect concurrency was inherently unsafe and must be left to streaming libraries, like FS2 (a popular library for doing concurrent streaming in Scala).

Nonetheless, having seen the amazing work coming out of Haskell and F#, I believed it was not only possible but very important for a modern effect type to solve four closely related concurrency concerns:

- Spawning a new independent 'thread' of computation
- Asynchronously waiting for a 'thread' to finish computing its return value
- Automatically canceling a running 'thread' when its return value is no longer needed
- Ensuring cancellation does not leak resources

In the course of time, I developed a small prototype of what became known as the *Scalaz 8 IO monad*, which solved these problems in a fast and purely functional package.

In this prototype, effects could be *forked* to yield a *fiber* (a cooperatively-yielding virtual thread), which could be joined or instantly interrupted, with a Haskell-inspired version of *try/finally* called *bracket* that provided resource safety, even in the presence of asynchronous or concurrent interactions.

I was very excited about this design, and I talked about it publicly before I released the code, resulting in some backlash from competitors who doubted resource safety or performance. But on November 16, 2017, I presented the first version at Scale by the Bay, opening a pull request with full source code, including rigorous tests and benchmarks, which allayed all concerns.

Despite initial skepticism and criticism, in time, all effect systems in Scala adopted this same model, including the ability to launch an effect to yield a fiber, which could be safely interrupted or joined, with support for finalizers to ensure resource safety.

This early prototype was not yet *ZIO* as we know it today, but the seeds of ZIO had been planted, and they grew quickly.

1.4 Typed Errors & Other Evolution

My initial design for the Scalaz 8 IO data type was inspired by the Haskell IO type and the Task type from Scalaz 7. In due time, however, I found myself reevaluating some decisions made by these data types.

For one, I was unhappy with the fact that most effect types have dynamically typed errors. The compiler can't help you reason about error behavior if you pretend that every computation can always fail in infinite ways.

As a statically-typed functional programmer, I want to use the compiler to help me write better code. I can do a better job if I know where I have and have not handled errors, and if I can use typed data models for business errors.

Of course, Haskell can sort of give you statically-typed errors with monad transformers, and maybe type classes. Unfortunately, this solution increases barriers to entry, reduces performance, and bolts on a second, often confusing error channel.

Since I was starting from a clean slate in a different programming language, I had a different idea: what if instead of rigidly fixing the error type to `Throwable`, like the Scala `Try` data type, I let the user choose the error type, exactly like `Either`?

Initial results of my experiment were remarkable: just looking at type signatures, I could understand exactly how code was dealing with errors (or not dealing with them). Effect operators precisely reflected error handling behavior in type signatures, and some laws that traditionally had to be checked using libraries like ScalaCheck were now checked statically, at compile time.

So on January 2, 2018, I committed what ended up being a radical departure from the status quo: introducing statically-typed errors into the Scalaz 8 effect type.

Over the months that followed, I worked on polish, optimization, bug fixes, tests, and documentation, and found growing demand to use the data type in production. When it became apparent that Scalaz 8 was a longer-term project, a few ambitious developers pulled the IO data type into a standalone library so they could begin using it in their projects.

I was excited about this early traction, and I didn't want any obstacles to using the data type for users of Scalaz 7.x or Cats, so on June 11, 2018, I decided to pull the project out into a new, standalone project with zero dependencies, completely separate from Scalaz 8.

I chose the name *ZIO*, combining the "Z" from "Scalaz", and the "IO" from "IO monad".

1.5 Contributor Growth

Around this time, the first significant wave of contributors started joining the project, including Regis Kuckaertz, Wiem Zine Elabidine, and Pierre Ricadat (among others)—many new to both open source and functional Scala, although some with deep backgrounds in both.

Through mentorship by me and other contributors, including in some cases weekly meetings and pull request reviews, a whole new generation of open source Scala contributors were born—highly talented functional Scala developers whose warmth positivity and can-do spirit started to shape the community.

ZIO accreted more polish and features to make it easier to build concurrent applications, such as an asynchronous, doubly-back-pressured queue, better error tracking and handling, rigorous finalization, and lower-level resource-safety than bracket.

Although the increased contributions led to an increasingly capable effect type, I personally found that using ZIO was not very pleasant, because of the library's poor type inference.

1.6 Improved Type-Inference

As many functional Scala developers at the time, I had absorbed the prevailing wisdom about how functional programming should be done in Scala, and this meant avoiding subtyping and declaration- site variance. (Indeed, the presence of subtyping in a language does negatively impact type inference, and using declaration-site variance has a couple drawbacks.)

However, because of this mindset, using ZIO required specifying type parameters when calling many methods, resulting in an unforgiving and joyless style of programming, particularly with typed errors. In private, I wrote a small prototype showing that using declaration-site variance could significantly improve type inference, which made me want to implement the feature in ZIO.

At the time, however, ZIO still resided in the Scalaz organization, in a separate repository. I was aware that such a departure from the status quo would be very controversial, so in a private fork, Wiem Zine Elabidine and I worked together on a massive refactoring in our first major collaboration.

On Friday July 20, 2018, we opened the pull request that embraced subtyping and covariance. The results spoke for themselves: nearly all explicit type annotations had been deleted, and although there was still some controversy, it was difficult to argue with the results. With this change, ZIO started becoming pleasant to use, and the extra error type parameter no longer negatively impacted usability, because it could always be inferred and widened seamlessly as necessary.

This experience emboldened me to start breaking other taboos: I started aggressively renaming methods and classes and removing jargon known only to pure

functional programmers. At each step, this created yet more controversy, but also further differentiated ZIO from some of the other options in the landscape, including those in Scalaz 7.x.

From all this turbulent evolution, a new take on functional Scala entered the ZIO community: a contrarian but principled take that emphasizes practical concerns, solving real problems in an accessible and joyful way, using all of Scala, including subtyping and declaration-site variance.

Finally, the project began to feel like the ZIO of today, shaped by a rapidly growing community of fresh faces eager to build a new future for functional programming in Scala.

1.7 Batteries Included

Toward the latter half of 2018, ZIO got compositional scheduling, with a powerful new data type that represents a schedule, equipped with rich compositional operators. Using this single data type, ZIO could either retry effects or repeat them according to near arbitrary schedules.

Artem Pyanykh implemented a blazing fast low-level ring-buffer, which, with the help of Pierre Ricadat, became the foundation of ZIO's asynchronous queue, demonstrating the ability of the ZIO ecosystem to create de novo high-performance JVM structures.

Itamar Ravid, a highly talented Scala developer, joined the ZIO project and added a *Managed* data type encapsulating resources. Inspired by Haskell, *Managed* provided compositional resource safety in a package that supported parallelism and safe interruption. With the help of Maxim Schuwalow, *Managed* has grown to become an extremely powerful data type.

Thanks to the efforts of Raas Ahsan, ZIO unexpectedly got an early version of what would later become `FiberRef`, a fiber-based version of `ThreadLocal`. Then Kai, a wizard-level Scala developer and type astronaut, labored to add compatibility with Cats Effect libraries, so that ZIO users could benefit from all the hard work put into libraries like Doobie, http4s, and FS2.

Thanks to the work of numerous contributors spread over more than a year, ZIO became a powerful solution to building concurrent applications—albeit, one without concurrent streams.

1.8 ZIO Stream

Although Akka Streams provides a powerful streaming solution for Scala developers, it's coupled to the Akka ecosystem and Scala's Future, and doesn't embrace the full compositional power of Scala.

In the functional Scala space, FS2 provides a streaming solution that works with ZIO but it's based on Cats Effect, whose type classes can't benefit from ZIO-specific features.

I knew that a ZIO-specific streaming solution would be more expressive and more type safe, with a lower barrier of entry for existing ZIO users. Given the importance of streaming to modern applications, I decided that ZIO needed its own streaming solution, one unconstrained by the limitations of Cats Effect.

Bringing a new competitive streaming library into existence would be a lot of work, and so when Itamar Ravid volunteered to help, I instantly said yes.

Together, in the third quarter of 2018, Itamar and I worked in secret on *ZIO Stream*, an asynchronous, back-pressured, resource-safe, and compositional stream. Inspired by the remarkable work of Eric Torreborre, as well as work in Haskell on iteratees, the initial release of ZIO Streams delivered a high-performance, composable concurrent streams and sinks, with strong guarantees of resource safety, even in the presence of arbitrary interruption.

We unveiled the design at Scale by the Bay 2018, and since then, thanks to Itamar and his army of capable contributors (including Regis Kuckaertz), ZIO Streams has become one of the highlights of the ZIO library—every bit as capable as other streaming libraries, but with much smoother integration with the ZIO effect type and capabilities.

Toward the end of 2018, I decided to focus on the complexity of testing code written using effect systems, which led to the last major revision of the ZIO effect type.

1.9 ZIO Environment

When exploring a contravariant reader data type to model dependencies, I discovered that using intersection types (emulated by the `with` keyword in Scala 2.x), one could achieve flawless type inference when composing effects with different dependencies, which provided a possible solution to simplifying testing of ZIO applications.

Excitedly, I wrote up a simple toy prototype and shared it with Wiem Zine Elabidine. *"Do you want to help work on this?"* I asked. She said yes, and together we quietly added the third and final type parameter to the ZIO effect type: the environment type parameter.

I unveiled the third type parameter at a now-infamous talk, *The Death of Finally Tagless*, humorously presented with a cartoonish Halloween theme. In this talk, I argued that testability was the primary benefit of the so-called "tagless-final" technique, and that it could be obtained much more simply and in a more teachable way by just "passing interfaces"—the very same solution that object-oriented programmers have used for decades.

As with tagless-final, and under the assumption of discipline, ZIO Environment provided a way to reason about dependencies statically. But unlike tagless-final, it's a joy to use because it fully infers, and doesn't require teaching type classes, category theory, higher-kinded types, and implicits.

Some ZIO users immediately started using ZIO Environment, appreciating the ability to describe dependencies using types without actually passing them. Constructing ZIO environments, however, proved to be problematic—impossible to do generically, and somewhat painful to do even when the structure of the environment was fully known.

A workable solution to these pains would not be identified until almost a year later.

Meanwhile, ZIO continued to benefit from numerous contributions, which added operators improved documentation, improved interop, and improved semantics for core data types.

The next major addition to ZIO was software transactional memory.

1.10 Software Transactional Memory

The first prototype of the Scalaz IO data type included `MVar`, a doubly-back-pressured queue with a maximum capacity of 1, inspired by Haskell's data type of the same name.

I really liked the fact that MVar was already "proven", and could be used to build many other concurrent data structures (such as queues, semaphores, and more).

Soon after that early prototype, however, the talented and eloquent Fabio Labella convinced me that two simpler primitives provided a more orthogonal basis for building concurrency structures:

- *Promise*, a variable data type that can be set exactly one time (but can be awaited on asynchronously and retrieved any number of times);
- *Ref*, a model of a mutable cell that can store any immutable value, with atomic operations for updating the cell.

This early refactoring allowed us to delete `MVar` and provided a much simpler foundation. However, after a year of using these structures, while I appreciated their power, it became apparent to me that they were the "assembly language" of concurrent data structures.

These structures could be used to build lots of other asynchronous concurrent data structures, such as semaphores, queues, and locks, but doing so was extremely tricky, and required hundreds of lines of fairly advanced code.

Most of the complexity stems from the requirement that operations on the data structures must be safely interruptible, without leaking resources or deadlocking.

Moreover, although you can build concurrent structures with `Promise` and `Ref`, you cannot make coordinated changes across two or more such concurrent structures.

The transactional guarantees of structures built with `Promise` and `Ref` are non-compositional: they apply only to isolated data structures, because they are built with `Ref`, which has non-compositional transactional semantics. Strictly speaking, their transactional power is equivalent to actors with mutable state: each actor can safely mutate its own state, but no transactional changes can be made across multiple actors.

Familiar with Haskell's software transactional memory, and how it provides an elegant, compositional solution to the problem of developing concurrent structures, I decided to implement a version for ZIO with the help of my partner-in-crime Wiem Zine Elabidine, which we presented at Scalar Conf in April 2019.

Soon after, Dejan Mijic, a fantastic and highly motivated developer with a keen interest in high-performance, concurrency, and distributed systems, joined the ZIO STM team. With my mentorship, Dejan helped make STM stack-safe for transactions of any size, added several new STM data structures, dramatically improved the performance of existing structures, and implemented retry-storm protection for supporting large transactions on hotly contested transactional references.

ZIO STM is the only STM in Scala with these features, and although the much older Scala STM is surely production-worthy, it doesn't integrate well with asynchronous and purely functional effect systems built using fiber-based concurrency.

The next major feature in ZIO would address a severe deficiency that had never been solved in the Scala ecosystem: the extreme difficulty of debugging async code, a problem present in Scala's Future for more than a decade.

1.11 Execution Traces

Previously in presenting ZIO to new non-pure functional programmers (the primary audience for ZIO), I had received the question: how do we debug ZIO code?

The difficulty stems from the worthless nature of stack traces in highly asynchronous programming. Stack traces only capture the call stack, but in Future and ZIO and other heavily async environments, the call stack mainly shows you the "guts" of the execution environment, which is not very useful for troubleshooting errors.

I had thought about the problem and had become convinced it would be possible to implement async execution traces using information reconstructed from the

call stack, so I began telling people we would soon implement something like this in ZIO.

I did not anticipate just how soon this would happen.

Kai came to me with an idea to do execution tracing in a radically different way than I imagined: by dynamically parsing the bytecode of class files. Although my recollection is a bit hazy, it seemed mere days before Kai had whipped up a prototype that seemed extremely promising, so I offered my assistance on hammering out the details of the full implementation, and we ended up doing a wonderful joint talk in Ireland to launch the feature.

Sometimes I have a tendency to focus on laws and abstractions, but seeing the phenomenally positive response to execution tracing was a good reminder to stay focused on the real world pains that developers have.

1.12 Summer 2019

Beginning in the summer of 2019, ZIO began seeing its first significant commercial adoption, which led to many feature requests and bug reports, and much feedback from users.

The summer saw many performance improvements, bug fixes, naming improvements, and other tweaks to the library, thanks to Regis Kuckaertz and countless other contributors.

Thanks to the work of the ever-patient Honza Strnad and others, `FiberRef` evolved into its present-day form, which is a much more powerful, fiber-aware version of ThreadLocal—but one which can undergo specified transformations on forks, and merges on joins.

I was very pleased with these additions. However, as ZIO grew, the automated tests for ZIO were growing too, and they became an increasing source of pain across Scala.js, JVM, and Dotty (the test runners at the time did not natively support Dotty).

So in the summer of 2019, I began work on a purely functional testing framework, with the goal of addressing these pains, the result of which was ZIO Test.

1.13 ZIO Test

Testing functional effects inside a traditional testing library is painful: there's no easy way to run effects, provide them with dependencies, or integrate with the host facilities of the functional effect system (using retries, repeats, and so forth).

I wanted to change that with a small, compositional library called ZIO Test, whose design I had been thinking about since even before ZIO existed.

Like the ground-breaking Specs2 before it, ZIO Test embraced a philosophy of tests as values, although ZIO Test retained a more traditional tree-like structure for specs, which allows nesting tests inside test suites, and suites inside other suites.

Early in the development of ZIO Test, the incredible and extremely helpful Adam Fraser joined the project as a core contributor. Instrumental to fleshing out, realizing, and greatly extending the vision for ZIO Test, Adam has since become the lead architect and maintainer for the project, and co-author of this book.

Piggybacking atop ZIO's powerful effect type, ZIO Test was implemented in comparatively few lines of code: concerns like retrying, repeating, composition, parallel execution, and so forth, were already implemented in a principled, performant, and type-safe way.

Indeed, ZIO Test also got a featherweight alternative to ScalaCheck based on ZIO Streams, since a generator of a value can be viewed as a stream. Unlike ScalaCheck, the ZIO Test generator has auto-shrinking baked in, inspired by the Haskell Hedgehog library; and it correctly handles filters on shrunk values and other edge case scenarios that ScalaCheck did not handle.

Toward the end of 2018, after nearly a year of real world usage, the ZIO community had been hard at work on solutions to the problem of making dynamic construction of ZIO environments easier.

This work directly led to the creation of *ZLayer*, the last major data type added to ZIO.

1.14 ZLayer

Two very talented Scala developers, Maxim Schuwalow and Piotr Gołębiewski, jointly worked on a ZIO Macros project, which, among other utilities, provided an easier way to construct larger ZIO environments from smaller pieces. This excellent work was independently replicated in Netflix's highly-acclaimed Polynote by Scala engineer Jeremy Smith, in response to the same pain.

At Functional Scala 2019, several speakers presented on the pain of constructing ZIO Environments, which convinced me to take a hard look at the problem. Taking inspiration from an earlier attempt by Piotr, I created two new data types, `Has` and `ZLayer`.

`Has` can be thought of as a type-indexed heterogeneous map, which is type safe, but requires access to compile-time type tag information. `ZLayer` can be thought of as a more powerful version of Java and Scala constructors, which can build multiple services in terms of their dependencies.

Unlike constructors, ZLayer dependency graphs are ordinary values, built from other values using composable operators, and ZLayer supports resources, asyn-

chronous creation and finalization, retrying, and other features not possible with constructors.

ZLayer provided a very clean solution to the problems developers were having with ZIO Environment—not perfect, mind you, and I don't think any solution prior to Scala 3 can be perfect (every solution in the design space has different tradeoffs). This solution became even better when the excellent consultancy Septimal Mind donated Izumi Reflect to the ZIO organization.

The introduction of ZLayer was the last major change to any core data type in ZIO. Since then, although streams has seen some evolution, the rest of ZIO was quite stable.

Yet despite the stability, until August 2020, there was still one major unresolved issue at the very heart of the ZIO runtime system: a full solution to the problem of structured concurrency.

1.15 Structured Concurrency

Structured concurrency is a paradigm that provides strong guarantees around the lifespans of operations performed concurrently. These guarantees make it easier to build applications that have stable, predictable resource utilization.

Since I have long been a fan of Haskell structured concurrency (via Async and related), ZIO was the first effect system to support structured concurrency in numerous operations:

- By default, interrupting a fiber does not return until the fiber has been interrupted and all its finalizers executed.
- By default, timing out an effect does not return until the effect being timed out has been interrupted and all its finalizers executed.
- By default, when executing effects in parallel, if one of them fails, the parallel operation will not continue until all sibling effects have been interrupted.
- Etc.

Some of these design decisions were highly contentious and have not been implemented in other effect systems until recently (if at all).

However, there was one notable area where ZIO did not provide default structured concurrency: whenever an effect was forked (launched concurrently to execute on a new fiber), the lifespan of the executing effect was unconstrained.

Solving this problem turned out to require major surgery to the ZIO internal runtime system (which is a part of ZIO that few contributors understand completely).

In the end, we solved the problem in a satisfactory way, making ZIO the only effect system to fully support structured concurrency. But it required learning

from real world feedback and prototyping no less than 5 completely different solutions to the problem.

So after three years of development, on August 3rd, 2020, ZIO 1.0 was released live in an online Zoom-hosted launch party that brought together and paid tribute to contributors and users across the ZIO ecosystem. We laughed, we chatted, I rambled for a while, and we toasted to users, contributors, and the past and future of ZIO.

1.16 Why ZIO

ZIO is a new library for concurrent programming. Using features of the Scala programming language, ZIO helps you build efficient, resilient, and concurrent applications that are easy to understand and test, and which don't leak resources, deadlock, or lose errors.

Used pervasively across an application, ZIO simplifies many of the challenges of building modern applications:

- **Concurrency**. Using an asynchronous fiber-based model of concurrency that never blocks threads or deadlocks, ZIO can run thousands or millions of virtual threads concurrently.
- **Efficiency**. ZIO automatically cancels running computations when the result of the computations are no longer necessary, providing global application efficiency for free.
- **Error Handling**. ZIO lets you track errors statically, so the compiler can tell you which code has handled its errors, and which code can fail, including how it can fail.
- **Resource-Safety**. ZIO automatically manages the lifetime of resources, safely acquiring them and releasing them even in the presence of concurrency and unexpected errors.
- **Streaming**. ZIO has powerful, efficient, and concurrent streaming that works with any source of data, whether structured or unstructured, and never leaks resources.
- **Troubleshooting**. ZIO captures all errors, including parallel and finalization errors, with detailed execution traces and suspension details that make troubleshooting applications easy.
- **Testability**. With *dependency inference*, ZIO makes it easy to code to interfaces, and ships with testable clocks, consoles, and other core system modules.

ZIO frees application developers to focus on business logic, and fully embraces the features of the Scala programming languages to improve productivity, testability, and resilience.

Since it's 1.0 release in August of 2020, ZIO has sparked a teeming ecosystem of ZIO-compatible libraries that provide support for GraphQL, persistence, REST

APIs, microservices, and much more.

1.17 ZIO Alternatives

ZIO is not the only choice for concurrent programming in Scala. In addition to libraries and frameworks in the Java ecosystem, Scala programmers have their choice of several competing solutions with first-class support for the Scala programming language:

- **Akka**. Akka is a toolkit for building reactive, concurrent, and distributed applications.
- **Monix**. Monix is a library for composing asynchronous, event-based programs.
- **Cats Effect**. Cats Effect is a purely functional runtime system for Scala.

Akka is older and more mature, with a richer ecosystem and more production usage, but ZIO provides compositional transactionality, resource-safety, full testability, better diagnostics, and greatly improved resilience via compile-time error analysis.

Monix is more focused on reactive programming, and less focused on concurrent programming, but to the extent it overlaps with ZIO, ZIO has significantly more expressive power.

Cats Effect is more focused on so-called *tagless-final* type classes than concurrent programming, and while it has some concurrent features, it is much less expressive than ZIO, without any support for compositional transactionality.

Beyond just technical merits, I think there are compelling reasons for Scala developers to take a serious look at ZIO:

- **Wonderful Community**. I'm amazed at all the talent, positivity, and mentorship seen in the ZIO community, as well as the universal attitude that we are all on the same team—there is no *your code*, or *my code*, just *our code*.

2. **History of Innovation**. ZIO has been the first effect type with proper thread pool locking, typed errors, environment, execution tracing, fine-grained interruption, structured concurrency, and so much more. Although inspired by Haskell, ZIO has charted its own course and become what many believe to be the leading effect system in Scala.
3. **A Bright Future**. ZIO took three years to get right, because I believed the foundations had to be extremely robust to support a compelling next-generation ecosystem. If early libraries like Caliban, ZIO Redis, ZIO Config, ZIO gRPC, and others are any indication, ZIO will continue to become a hotbed of exciting new developments for the Scala programming language.

Concurrent programming in Scala was never this much fun, correct-by-construction, or productive.

1.18 Zionomicon

In your hands, you have Zionomicon, a comprehensive book lovingly crafted by myself and Adam Fraser with one goal: to turn you into a wizard at building modern applications.

Through the course of this book, you will learn how to build cloud-ready applications that are responsive, resilient, elastic, and event-driven.

They will be low-latency and globally efficient.

They will not block threads, leak resources, or deadlock.

They will be checked statically at compile-time by the powerful Scala compiler.

They will be fully testable, straightforward to troubleshoot with extensive diagnostics.

They will deal with errors in a principled and robust fashion, surviving transient failures, and handling business errors according to requirements.

In short, our goal with this book is to help you become a programmer of extraordinary power, by leveraging both the Scala programming language and the power of functional composition.

Congratulations on taking your first step toward mastering the dark art of ZIO!

Chapter 2

Essentials: First Steps With ZIO

Congratulations on taking your first step in mastering ZIO!

ZIO will help you build modern applications that are concurrent, resilient, and efficient, as well as easy to understand and test. But learning ZIO requires thinking about software in a whole new way—a way that comes from *functional programming*.

This chapter will teach you the critical theory you need to understand and build ZIO applications.

We will start by introducing the core data type in ZIO, which is called a *functional effect type*, and define functional effects as *blueprints* for concurrent workflows. We will learn how to combine effects sequentially, and see how this allows us to refactor legacy code to ZIO.

We will discuss the meaning of each of the type parameters in ZIO's core data type, particularly the error type and the environment type, which are features unique to ZIO. We will compare `ZIO` to the `Future` data type in the Scala standard library, to clarify the concepts we introduce.

We will see how ZIO environment lets us leverage the testable services built into ZIO for interacting with time, the console, and system information (among others). Finally, we'll see how recursive ZIO effects allow us to loop and perform other control flow operations.

By the end of this chapter, you will be able to write basic programs using `ZIO`, including those that leverage environmental effects and custom control flow operators, and you will be able to refactor legacy code to `ZIO` by following some simple guidelines.

2.1 Functional Effects As Blueprints

The core data type in the ZIO library is `ZIO[R, E, A]`, and values of this type are called *functional effects*.

A functional effect is a kind of *blueprint for a concurrent workflow*, as illustrated in Figure 1. The blueprint is purely descriptive in nature, and must be executed in order to observe any side-effects, such as interaction with a database, logging, streaming data across the network, or accepting a request.

Figure 1

A functional effect of type `ZIO[R, E, A]` requires you to supply a value of type `R` if you want to execute the effect (this is called the *environment* of the effect), and when it is executed, it may either fail with a value of type `E` (the *error type*, or succeed with a value of type `A` (the *success type*).

We will talk more about each of these type parameters shortly. But first, we need to understand what it means for an effect to be a *blueprint*.

In traditional procedural programming, we are used to each line of our code directly interacting with the outside world. For example, consider the following snippet:

```
val goShoppingUnsafe: Unit = {
  println("Going to the grocery store")
}
```

As soon as Scala computes the unit value for the `goShoppingUnsafe` variable, the application will immediately print the text "Going to the grocery store" to the console.

This is an example of *direct execution*, because in constructing a value, our program directly interacts with the outside world.

This style of programming is called *procedural programming*, and is familiar to almost all programmers, since most programming languages are procedural in nature.

Procedural programming is convenient for simple programs. But when we write our programs in this style, *what* we want to do (going to the store) becomes tangled with *how* we want to do it (going to the store *now*).

This tangling can lead to lots of boilerplate code that is difficult to understand and test, painful to change, and fraught with subtle bugs that we won't discover until production.

For example, suppose we don't actually want to go to the grocery store now but in an hour from now. We might try to implement this new feature by using a `ScheduledExecutorService`:

```
import java.util.concurrent.{ Executors, ScheduledExecutorService }
import java.util.concurrent.TimeUnit._

val scheduler: ScheduledExecutorService =
  Executors.newScheduledThreadPool(1)

scheduler.schedule(
  new Runnable { def run: Unit = goShoppingUnsafe },
  1,
  HOURS
)
scheduler.shutdown()
```

In this program, we create an executor, schedule `goShoppingUnsafe` to be executed in one hour, and then shut down the scheduler when we are done. (Don't worry if you don't understand everything that is going on here. We will see that ZIO has much easier ways of doing the same thing!)

Not only does this solution involve boilerplate code that is difficult to understand and test, and painful to change, but it also has a subtle bug!

Because `goShoppingUnsafe` is directly executed, rather than being a blueprint for a workflow, "Going to the grocery store" will be printed to the console as soon as `goShoppingUnsafe` is loaded by the JVM. So we will be going to the grocery store now instead of an hour from now!

In fact, the only thing we have scheduled to be executed in an hour is returning the `Unit` value of `goShoppingUnsafe`, which doesn't do anything at all.

In this case, we can solve the problem by defining `goShoppingUnsafe` as a `def` instead of a `val` to defer its evaluation until later. But this approach is fragile and error prone, and forces us to think carefully about when each statement in our program will be evaluated, which is no longer the order of the statements.

We also have to be careful not to accidentally evaluate a statement too early. We might assign it to a value or put it into a data structure, which could cause premature evaluation.

It is as if we want to talk to our significant other about going shopping, but as soon as we mention the word "groceries", they are already at the door!

The solution to this problem (and most problems in concurrent programming) is to make the statements in our program *values* that *describe* what we want to do. This way, we can separate *what we want to do* from *how we want to do it.*

The following snippet shows what this looks like with ZIO:

```
import zio._

val goShopping =
  ZIO.effect(println("Going to the grocery store"))
```

Here we are using `effect` constructor to build the `goShopping` functional effect. The effect is a blueprint that *describes* going to the store, but doesn't actually do anything right now. (To prove this to yourself, try evaluating the code in the Scala REPL!)

In order to go to the store, we have to *execute* the effect, which is clearly and forcibly separated from defining the effect, allowing us to untangle these concerns and simplifying code tremendously.

With `goShopping` defined this way, we can now describe *how* independent from *what*, which allows us to solve complex problems compositionally, by using operations defined on ZIO effects.

Using the `delay` operator that is defined on all ZIO effects, we can take `goShopping`, and transform it into a new effect, which will go shopping an hour from now:

```
import zio.clock._
import zio.duration._

val goShoppingLater =
  goShopping.delay(1.hour)
```

Notice how easy it was for us to reuse the original effect, which specified *what*, to produce a new effect, which also specified *when*. We built a solution to a more complex problem by transforming a solution to a simpler problem.

Thanks to the power of describing workflows as ordinary immutable values, we never had to worry about how `goShopping` was defined or about evaluating it too early. Also, the value returned by the `delay` operator is just another description, so we can easily use it to build even more sophisticated programs in the same way.

In ZIO, every `ZIO` effect is just a description—a blueprint for a concurrent workflow. As we write our program, we create larger and more complex blueprints that come closer to solving our business problem. When we are done and have an effect that describes everything we need to do, we hand it off to the ZIO runtime, which executes the blueprint and produces the result of the program.

So how do we actually run a `ZIO` effect? The easiest way is to extend the `App` trait and implement the `run` method, as shown in the following snippet:

```
import zio._

object GroceryStore extends App {
  def run(args: List[String]) =
    goShopping.exitCode
}
```

The `run` function requires that we return an `ExitCode`, which describes the exit value to use when the process terminates. The `exitCode` method defined on

effects is a convenience method that translates all successes to `ExitCode(0)` and failures to an `ExitCode(1)`.

As you are experimenting with ZIO, extending `App` and implementing your own program logic in the `run` method is a great way to see the output of different programs.

2.2 Sequential Composition

As discussed above, ZIO effects are blueprints for describing concurrent workflows, and we build more sophisticated effects that come closer to solving our business problem by transforming and combining smaller, simpler effects.

We saw how the `delay` operator could be used to transform one effect into another effect whose execution is delayed into the future. In addition to `delay`, ZIO has dozens of other powerful operators that transform and combine effects to solve common problems in modern application development.

We will learn about most of these operators in subsequent chapters, but one of the most important operators that we need to introduce is called `flatMap`.

The `flatMap` method of ZIO effects represents sequential composition of two effects, allowing us to create a second effect based on the output of the first effect.

A simplified type signature for `flatMap` looks something like this:

```
trait ZIO[R, E, A] {
  ...
  def flatMap[B](andThen: A => ZIO[R, E, B]): ZIO[R, E, B] = ...
  ...
}
```

In effect, `flatMap` says, "run the first effect, then run a second effect that depends on the result of the first one". Using this sequential operator, we can describe a simple workflow that readers user input and then displays the input back to the user, as shown in the following snippet:

```
import scala.io.StdIn

val readLine =
  ZIO.effect(StdIn.readLine())

def printLine(line: String) =
  ZIO.effect(println(line))

val echo =
  readLine.flatMap(line => printLine(line))
```

Notice how what we print on the console depends on what we read from the console: so we are doing two things in sequence, and the second thing that we do depends on the value produced by the first thing we do.

The `flatMap` operator is fundamental because it captures the way statements are executed in a procedural program: later statements depend on results computed by previous statements, which is exactly the relationship that `flatMap` describes.

For reference, here is the above program written in a procedural style:

```
val line = Console.readLine
Console.println(line)
```

This relationship between procedural programming and the `flatMap` operator is so precise, we can actually translate any procedural program into ZIO by wrapping each statement in a constructor like `ZIO.effect` and then gluing the statements together using `flatMap`.

For example, let's say we have the procedural program shown in the following snippet:

```
val data = doQuery(query)
val response = generateResponse(data)
writeResponse(response)
```

We can translate this program into ZIO as follows:

```
ZIO.effect(doQuery(query)).flatMap(data =>
  ZIO.effect(generateResponse(data)).flatMap(response =>
    ZIO.effect(writeResponse(response))
  )
)
```

Although a straightforward transformation, once you exceed two or three `flatMap` operations in a row, the nesting of the code becomes somewhat hard to follow. Fortunately, Scala has a feature called *for comprehensions*, which allow us to express sequential composition in a way that looks like procedural programming.

In the next section, we'll explore *for comprehensions* at length.

2.2.1 For Comprehensions

Using *for comprehensions*, we can take the following Scala snippet:

```
readLine.flatMap(line => printLine(line))
```

and rewrite it into the following *for comprehension*:

```
import zio._

val echo =
  for {
```

```
    line <- readLine
    _    <- printLine(line)
  } yield ()
```

As you can see from this short snippet, there is no nesting, and each line in the comprehension looks similar to a statement in procedural programming.

For comprehensions have the following structure:

1. They are introduced by the keyword `for`, followed by a code block, and terminated by the keyword `yield`, which is followed by a single parameter, representing the success value of the effect.
2. Each line of the *for comprehension* is written using the format `result <- effect`, where `effect` returns an effect, and `result` is a variable that will hold the success value of the effect. If the result of the effect is not needed, then the underscore may be used as the variable name.

A *for comprehension* with `n` lines is translated by Scala into `n - 1` calls to `flatMap` methods on the effects, followed by a final call to a `map` method on the last effect.

So, for example, if we have the following *for comprehension*:

```
for {
  x <- doA
  y <- doB(x)
  z <- doC(x, y)
} yield x + y + z
```

Then Scala will translate it into the following code:

```
doA.flatMap(x =>
  doB(x).flatMap(y =>
    doC(x, y).map(z => x + y + z)))
```

Many Scala developers find that *for comprehensions* are easier to read than long chains of nested calls to `flatMap`. In this book, except for very short snippets, we will prefer *for comprehensions* over explicit calls to `flatMap`.

2.3 Other Sequential Operators

Sequential composition is so common when using functional effects, ZIO provides a variety of related operators for common needs.

The most basic of these is `zipWith`, which combines two effects sequentially, merging their two results with the specified user-defined function.

For example, if we have two effects that prompt for the user's first name and last name, then we can use `zipWith` to combine these effects together sequentially, merging their results into a single string:

```
val firstName =
  ZIO.effect(StdIn.readLine("What is your first name?"))

val lastName =
  ZIO.effect(StdIn.readLine("What is your last name"))

val fullName =
  firstName.zipWith(lastName)((first, last) => s"$first $last")
```

The `zipWith` operator is less powerful than `flatMap`, because it does not allow the second effect to depend on the first, even though the operator still describes sequential, left-to-right composition.

Other variations include `zip`, which sequentially combines the results of two effects into a tuple of their results; `zipLeft`, which sequentially combines two effects, returning the result of the first; and `zipRight`, which sequentially combines two effects returning the result of the second.

Occasionally, you will see `<*` used as an alias for `zipLeft`, and `*>` as an alias for `zipRight`. These operators are particularly useful to combine a number of effects sequentially when the result of one or more of the effects are not needed.

For example, in the following snippet, we sequentially combine two effects, returning the `Unit` success value of the right hand effect:

```
val helloWorld =
  ZIO.effect(print("Hello, ")) *> ZIO.effect(print("World!\n"))
```

This is useful because even while `Unit` is not a very useful success value, a tuple of unit values is even less useful!

Another useful set of sequential operators is `foreach` and `collectAll`.

The `foreach` operator returns a single effect that describes performing an effect for each element of a collection in sequence. It's similar to a *for loop* in procedural programming, which iterates over values, processes them in some fashion, and collects the results.

For example, we could create an effect that describes printing all integers between `1` and `100` like this:

```
val printNumbers =
  ZIO.foreach(1 to 100) { n =>
    printLine(n.toString)
  }
```

Similarly, `collectAll` returns a single effect that collects the results of a whole collection of effects. We could use this to collect the results of a number of printing effects, as shown in the following snippet:

```
val prints =
  List(
```

```
    printLine("The"),
    printLine("quick"),
    printLine("brown"),
    printLine("fox")
  )

val printWords =
  ZIO.collectAll(prints)
```

With just what you have learned so far, you can take any procedural program and translate it into a ZIO program by wrapping statements in effect constructors and combining them with `flatMap`.

If that is all you do, you won't be taking advantage of all the features that ZIO has to offer, but it's a place to start, and it can be a useful technique when migrating legacy code to ZIO.

2.4 ZIO Type Parameters

We said before that a value of type `ZIO[R, E, A]` is a functional effect that requires an environment `R` and may either fail with an `E` or succeed with an `A`.

Now that we understand what it means for a `ZIO` effect to be a blueprint for a concurrent workflow, and how to combine effects, let's talk more about each of the ZIO type parameters:

- `R` is the environment required for the effect to be executed. This could include any dependencies the effect has, for example access to a database or a logging service, or an effect might not require any environment, in which case, the type parameter will be `Any`.
- `E` is the type of value that the effect can fail with. This could be `Throwable` or `Exception`, but it could also be a domain-specific error type, or an effect might not be able to fail at all, in which case the type parameter will be `Nothing`.
- `A` is the type of value that the effect can succeed with. It can be thought of as the return value or output of the effect.

A helpful way to understand these type parameters is to imagine a `ZIO` effect as a function `R => Either[E, A]`. This is not actually the way `ZIO` is implemented (this definition wouldn't allow us to write concurrent, async or resource-safe operators, for example), but it is a useful mental model.

The following snippet of code defines this toy model of a ZIO effect:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])
```

As you can see from this definition, the `R` parameter is an *input* (in order to execute the effect, you must supply a value of type `R`), while the `E` and `A`

parameters are *outputs*. The input is declared to be *contravariant*, and the outputs are declared to be *covariant*.

For a more detailed discussion of variance, see the appendix. Otherwise, just know that Scala's variance annotations *improve type inference*, which is why ZIO uses them.

Let's see how we can use this mental model to implement some basic constructors and operators:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def map[B](f: A => B): ZIO[R, E, B] =
    ZIO(r => self.run(r).map(f))
  def flatMap[R1 <: R, E1 >: E, B](
    f: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ZIO(r => self.run(r).fold(ZIO.fail(_), f).run(r))
}

object ZIO {
  def effect[A](a: => A): ZIO[Any, Throwable, A] =
    ZIO(_ => try Right(a) catch { case t : Throwable => Left(t) })
  def fail[E](e: => E): ZIO[Any, E, Nothing] =
    ZIO(_ => Left(e))
}
```

The `ZIO.effect` method wraps a block of code in an effect, converting exceptions into `Left` values, and successes into `Right` values. Notice that the parameter to `ZIO.effect` is *by name* (using the `=> A` syntax), which prevents the code from being evaluated eagerly, allowing ZIO to create a value that *describes* execution.

We also implemented the `flatMap` operator previously discussed, which allows us to combine effects sequentially. The implementation of `flatMap` works as follows:

1. It first runs the original effect with the environment `R1` to produce an `Either[E, A]`.
2. If the original effect fails with a `Left(e)`, it immediately returns this failure as a `Left(e)`.
3. If the original effect succeeds with a `Right(a)`, it calls `f` on that `a` to produce a new effect. It then runs that new effect with the required environment `R1`.

As discussed above, ZIO effects aren't actually implemented like this, but the basic idea of executing one effect, obtaining its result, and then passing it to the next effect is an accurate mental model, and it will help you throughout your time working with ZIO.

We will learn more ways to operate on successful `ZIO` values soon, but for now lets focus on the error and environment types to build some intuition about

them since they may be less familiar.

2.4.1 The Error Type

The error type represents the potential ways that an effect can fail. The error type is helpful because it allows us to use operators (like `flatMap`) that work on the success type of the effect, while deferring error handling until higher-levels. This allows us to concentrate on the "happy path" of the program and handle errors at the right place.

For example, say we want to write a simple program that gets two numbers from the user and multiplies them:

```
import zio._

lazy val readInt: ZIO[Any, NumberFormatException, Int] =
  ???

lazy val readAndSumTwoInts: ZIO[Any, NumberFormatException, Int] =
  for {
    x <- readInt
    y <- readInt
  } yield x * y
```

Notice that `readInt` has a return type of `ZIO[Any, NumberFormatException, Int]`, indicating that it does not require any environment and may either succeed with an integer (if the user enters a response that can be parsed into a valid integer) or fail with a `NumberFormatException`.

The first benefit of the error type is that we know how this function can fail just from its signature. We don't know anything about the implementation of `readInt`, but just looking at the type signature we know that it can fail with a `NumberFormatException`, and can't fail with any other errors. This is very powerful because we know exactly what kind of errors we potentially have to deal with, and we never have to resort to "defensive programming" to handle unknown errors.

The second benefit is that we can operate on the results of effects assuming they are successful, deferring error handling until later. If either `readInt` call fails with a `NumberFormatException`, then `readAndSumTwoInts` will also fail with the exception, and abort the summation. This bookkeeping is handled for us automatically. We can multiply `x` and `y` directly and never have to deal explicitly with the possibility of failure. This defers error handling logic to the caller, which can retry, report, or defer handling even higher.

Being able to see how an effect can fail and to defer errors to a higher level of an application is useful, but at some point, we need to be able to handle some or all errors.

To handle errors with our toy model of ZIO, let's implement an operator called `foldM` that will let us perform one effect if the original effect fails, and another one if it succeeds:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def foldM[R1 <: R, E1, B](
    failure: E => ZIO[R1, E1, B],
    success: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ZIO(r => self.run(r).fold(failure, success).run(r))
}
```

The implementation is actually quite similar to the one we walked through above for `flatMap`. We are just using the `failure` function to return a new effect in the event of an error, and then running that effect.

One of the most useful features of the error type is being able to specify that an effect *cannot fail at all*, perhaps because its errors have already been caught and handled.

In ZIO, we do this by specifying `Nothing` as the error type. Since there are no values of type `Nothing`, we know that if we have an `Either[Nothing, A]` it must be a `Right`. We can use this to implement error handling operators that let us statically prove that an effect can't fail because we have handled all errors.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def fold[B](failure: E => B, success: A => B): ZIO[R, Nothing, B] =
    ZIO(r => Right(self.run(r).fold(failure, success)))
}
```

2.4.2 The Environment Type

Now that we have some intuition around the error type, let's focus on the environment type.

We can model effects that don't require any environment by using `Any` for the environment type. After all, if an effect requires a value of type `Any`, then you could run it with `()` (the unit value), `42`, or any other value. So an effect that can be run with a value of any type at all is actually an effect that doesn't need any specific kind of environment.

The two fundamental operational of working with the environment are accessing the environment (e.g. getting access to a database to do something with it) and providing the environment (providing a database service to an effect that needs one, so it doesn't need anything else).

We can implement this in our toy model of ZIO as shown in the following snippet:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def provide(r: R): ZIO[Any, E, A] =
```

```
    ZIO(_ => self.run(r))
}

object ZIO {
  def environment[R]: ZIO[R, Nothing, R] =
    ZIO(r => Right(r))
}
```

As you can see, the `provide` operator returns a new effect that doesn't require any environment. The `environment` constructor creates a new effect with a required environment type, and just passes through that environment as a success value. This allows us to access the environment and work with it using other operators like `map` and `flatMap`.

2.5 ZIO Type Aliases

With its three type parameters `ZIO` is extremely powerful. We can use the environment type parameter to propagate information downward in our program (databases, connection pools, configuration, and much more), and we can use the error and success type parameters to propagate information upward.

In the most general possible case, programs need to propagate dependencies down, and return results up, and ZIO provides all of this in a type-safe package.

But sometimes, we may not need all of this power. We may know that an application doesn't require an external environment, or that it can only fail with a certain type of errors, or that it can't fail at all.

To simplify these cases, ZIO comes with a number of useful type aliases. You never have to use these type aliases if you don't want to. You can always just use the full ZIO type signature. But these type aliases are used frequently in ZIO code bases so it is helpful to be familiar with them, and they can make your code more readable if you choose to use them.

The key type aliases are:

```
type IO[+E, +A]   = ZIO[Any, E, A]
  type Task[+A]     = ZIO[Any, Throwable, A]
  type RIO[-R, +A]  = ZIO[R, Throwable, A]
  type UIO[+A]      = ZIO[Any, Nothing, A]
  type URIO[-R, +A] = ZIO[R, Nothing, A]
```

Here is a brief description of each type alias to help you remember what they are for:

- `IO[E, A]` - An effect that does not require any environment, may fail with an `E`, or may succeed with an `A`
- `Task` - An effect that does not require any environment, may fail with a `Throwable`, or may succeed with an `A`

- `RIO` - An effect that requires an environment of type `R`, may fail with a `Throwable`, or may succeed with an `A`.
- `UIO` - An effect that does not require any environment, cannot fail, and succeeds with an `A`
- `URIO[R, A]` - An effect that requires an environment of type `R`, cannot fail, and may succeed with an `A`.

Several other data types in ZIO and other libraries in the ZIO ecosystem use similar type aliases, so if you are familiar with these you will be able to pick those up quickly, as well.

ZIO comes with companion objects for each of these type aliases, so you can call static methods with these type aliases the same way you would on `ZIO` itself.

For example, these two definitions are equivalent and are both valid syntax:

```
import zio._

val first: ZIO[Any, Nothing, Unit] =
  ZIO.effectTotal(println("Going to the grocery store"))
val second: UIO[Unit] =
  UIO.effectTotal(println("Going to the grocery store"))
```

ZIO strives for excellent type inference across the board, but in rare cases where type parameters have to be specified explicitly, the constructors on the companion object can require fewer type parameters (e.g. constructors on `UIO` have no error parameter) so they can improve type inference and ergonomics.

2.6 Comparison to Future

We can clarify what we have learned so far by comparing `ZIO` with `Future` from the Scala standard library.

We will discuss other differences between `ZIO` and `Future` later in this book when we discuss concurrency, but for now there are three primary differences to keep in mind.

2.6.1 A Future Is A Running effect

Unlike a functional effect like `ZIO`, a `Future` models a running effect. To go back to our example from the beginning of the chapter, consider this snippet:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

val goShoppingFuture: Future[Unit] =
  Future(println("Going to the grocery store"))
```

Just like our original example, as soon as `goShoppingFuture` is defined this effect will begin executing. `Future` does not suspend evaluation of code wrapped in it.

Because of this tangling between the *what* and the *how*, we don't have much power when using `Future`. For example, it would be nice to be able to define a `delay` operator on `Future`, just like we have for `ZIO`. But we can't do that because it would be a method on `Future`, and if we have a `Future`, then it is already running, so it's too late to delay it.

Similarly, we can't retry a `Future` in the event of failure, like we can for `ZIO`, because a `Future` isn't a blueprint for doing something—it's an executing computation. So if a `Future` fails, there is nothing else to do. We can only retrieve the failure.

In contrast, since a `ZIO` effect is a blueprint for a concurrent workflow, if we execute the effect once and it fails, we can always try executing it again, or executing it as many times as we would like.

One case in which this distinction is particularly obvious is the persistent requirement that you have an implicit `ExecutionContext` in scope whenever you call methods on `Future`.

For example, here is the signature of `Future#flatMap`, which like `flatMap` on `ZIO`, allows us to compose sequential effects:

```
import scala.concurrent.ExecutionContext

trait Future[+A] {
  def flatMap[B](f: A => Future[B])(
    implicit ec: ExecutionContext
  ): Future[B]
}
```

`Future#flatMap` requires an `ExecutionContext` because it represents a running effect, so we need to provide the `ExecutionContext` on which this subsequent code should be immediately run.

As discussed before, this conflates *what* should be done with *how* it should be done. In contrast, none of the code involving `ZIO` we have seen requires an `Executor` because it is just a blueprint.

ZIO blueprints can be run on any `Executor` we want, but we don't have to specify this until we actually run the effect (or, later we will see how you can "lock" an effect to run in a specific execution context, for those rare cases where you need to be explicit about this).

2.6.2 Future Has An Error Type Fixed To Throwable

`Future` has an error type fixed to `Throwable`. We can see this in the signature of the `Future#onComplete`:

```
import scala.util.Try

trait Future[+A] {
  def onComplete[B](f: Try[A] => B): Unit
}
```

The result of a `Future` can either be a `Success` with an `A` value, or a `Failure` with a `Throwable`. When working with legacy code that can fail for any `Throwable`, this can be convenient, but it has much less expressive power than a polymorphic error type.

First, we don't know by looking at the type signature how or even if an effect can fail. Consider the multiplication example we looked at when discussing the `ZIO` error type implemented with `Future`:

```
def parseInt: Future[Int] =
  ???
```

Notice how we had to define this as a `def` instead of a `val` because a `Future` is a running effect. So if we defined it as a `val`, we would immediately be reading and parsing an input from the user. Then when we used `parseInt`, we'd always get back the same value, instead of prompting the user for a new value and parsing that.

Putting this aside, we have no idea how this future can fail by looking at the type signature. Could it return a `NumberFormatException` from parsing? Could it return an `IOException`? Could it not fail at all because it handles its own errors, perhaps by retrying until the user entered a valid integer? We just don't know, not unless we dig into the code and study it at length.

This makes it much harder for developers who call this method, because they don't know what type of errors can occur, so to be safe they need to do "defensive programming" and handle any possible `Throwable`.

This problem is especially annoying when we handle all possible failure scenarios of a `Future`, but nothing changes about the type.

For example, we can handle `parseInt` errors by using the `Future` method `fallbackTo`:

```
import scala.concurrent.Future

def parseIntOrZero: Future[Int] =
  parseInt.fallBackTo(Future.successful(0))
```

Here `parseIntOrZero` cannot fail, because if `parseInt` fails, we will replace it

with a successful result of `0`. But the type signature doesn't tell us this. As far as the type signature is concerned, this method could fail in infinitely many ways, just like `parseInt`!

From the perspective of the compiler, `fallBackTo` hasn't changed anything about the fallibility of the `Future`. In contrast, in ZIO `parseInt` would have a type of `IO[NumberFormatException, Int]`, and `parseIntOrZero` would have a type of `UIO[Int]`, indicating precisely how `parseInt` can fail and that `parseIntOrZero` cannot fail.

2.6.3 Future Does Not Have A Way To Model The Dependencies Of An Effect

The final difference between `ZIO` and `Future` that we have seen so far is that `Future` does not have any way to model the dependencies of an effect. This requires other solutions to dependency injection, which are usually manual (they cannot be inferred) or they depend on third-party libraries.

We will spend much more time on this later in the book, but for now just note that `ZIO` has direct support for dependency injection, but `Future` does not. This means that in practice, most Future code in the real world is not very testable, because it requires too much plumbing and boilerplate.

2.7 More Effect Constructors

Earlier in this chapter, we saw how to use the `ZIO.effect` constructor to convert procedural code to ZIO effects.

The `ZIO.effect` constructor is a useful and common effect constructor, but it's not suitable for every scenario:

1. **Fallible**. The `ZIO.effect` constructor returns an effect that can fail with any kind of `Throwable` (`ZIO[Any, Throwable, A]`). This is is the right choice when you are converting legacy code into ZIO and don't know if it throws exceptions, but sometimes, we know that some code doesn't throw exceptions (like retrieving the system time).
2. **Synchronous**. The `ZIO.effect` constructor requires that our procedural code be synchronous, returning some value of the specified type from the captured block of code. But in an asynchronous API, we have to register a callback to be invoked when a value of type `A` is available. How do we convert asynchronous code to ZIO effects?
3. **Unwrapped**. The `ZIO.effect` constructor assumes the value we are computing is not wrapped in yet another data type, which has its own way of modeling failure. But some of the code that we interact with return an `Option[A]`, an `Either[E, A]`, a `Try[A]`, or even a `Future[A]`. How do we convert from these types into ZIO effects?

Fortunately, ZIO comes with robust constructors that handle custom failure scenarios, asynchronous code, and other common data types.

2.7.1 Pure Versus Impure Values

Before introducing other ZIO effect constructors, we need to first talk about *referential transparency*. An expression such as `2 + 2` is *referentially transparent* if we can always replace the computation with its result in any program while still preserving its runtime behavior.

For example, consider this expression:

```
val sum: Int = 2 + 2
```

We could replace the expression `2 + 2` with its result, `4` and the behavior of our program would not change.

In contrast, consider the following simple program that reads a line of input from the console, and then prints it out to the console:

```
import scala.io.StdIn

val echo: Unit = {
  val line = StdIn.readLine()
  println(line)
}
```

We can't replace the body of `echo` with its result and preserve the behavior of the program. The result value of `echo` is just the `Unit` value, so if we replace `echo` with its return value we would have:

```
val echo: Unit = ()
```

These two programs are definitely not the same. The first one reads input from the user and prints the input to the console, but the second program does nothing at all!

The reason we can't substitute the body of `echo` with its computed result is that it performs *side effects*. It does things *on the side* (reading from and writing to the console). This is in contrast to referentially transparent functions, which are free of side-effects, and which just compute values.

Expressions without side-effects are called *pure expressions*, while functions whose body are pure expressions are called *pure functions*.

The `ZIO.effect` constructor takes side-effecting code, and converts it into a pure value, which merely describes side-effects.

To see this in action, let's revisit the ZIO implementation for our `echo` program:

```
import zio._
```

```
val readLine =
  ZIO.effect(StdIn.readLine())

def printLine(line: String) =
  ZIO.effect(println(line))

val echo =
  for {
    line <- readLine
    _    <- printLine(line)
  } yield ()
```

This program is referentially transparent because it just builds up a blueprint (an immutable value that describes a workflow), without performing any side-effects. We can replace the code that builds up this blueprint with the resulting blueprint and we still just have a plan for this `echo` program.

So we can view referential transparency as another way of looking at the idea of functional effects as blueprints. Functional effects make side-effecting code referentially transparent by describing their side-effects, instead of performing them.

This separation between description and execution untangles the *what* from the *how*, and gives us enormous power to transform and compose effects, as we will see over the course of this book.

Referential transparency is an important concept when converting code to ZIO, because if a value or a function is referentially transparent, then we don't need to convert it into a ZIO effect. However, if it's impure, then we need to convert it into a ZIO effect by using the right effect constructor.

ZIO tries to do the right thing even if you accidentally treat side-effecting code as pure code. But mixing side-effecting code with ZIO code can be a source of bugs, so it is best to be careful about using the right effect constructor. As a side benefit, this will make your code easier to read and review for your colleagues.

2.7.2 Effect Constructors For Pure Computations

ZIO comes with a variety of effect constructors to convert pure values into ZIO effects. These constructors are useful primarily when combining other ZIO effects, which have been constructed from side-effecting code, with pure code.

In addition, even pure code can benefit from some features of ZIO, such as environment, typed errors, and stack safety.

The two most basic ways to convert pure values into ZIO effects are `succeed` and `fail`:

```
object ZIO {
  def fail[E](e: => E): ZIO[Any, E, Nothing] = ???
```

```
  def succeed[A](a: => A): ZIO[Any, Nothing, A] = ???
}
```

The `ZIO.succeed` constructor converts a value into an effect that succeeds with that value. For example, `ZIO.succeed(42)` constructs an effect that succeeds with the value 42. The failure type of the effect returned by `ZIO.succeed` is `Nothing`, because effects created with this constructor cannot fail.

The `ZIO.fail` constructor converts a value into an effect that fails with that value. For example, `ZIO.fail(new Exception)` construct an effect that fails with the specified exception. The success type of the effect returned by `ZIO.fail` is `Nothing`, because effects created with this constructor cannot succeed.

We will see that effects which cannot succeed, either because they fail or because they run forever, often use `Nothing` as the success type.

In addition to these two basic constructors, there are a variety of other constructors that can convert standard Scala data types into ZIO effects.

```
import scala.util.Try

object ZIO {
  def fromEither[E, A](eea: => Either[E, A]): IO[E, A] = ???
  def fromOption[A](oa: => Option[A]): IO[None.type, A] = ???
  def fromTry[A](a: => Try[A]): Task[A] = ???
}
```

These constructors translate the success and failure cases of the original data type to the ZIO success and error types.

The `ZIO.fromEither` constructor converts an `Either[E, A]` into an `IO[E, A]` effect. If the `Either` is a `Left`, then the resulting ZIO effect will fail with an `E`, but if it is a `Right`, then the resulting ZIO effect will succeed with an `A`.

The `ZIO.fromTry` constructor is similar, except the error type is fixed to `Throwable`, because a `Try` can only fail with `Throwable`.

The `ZIO.fromOption` constructor is more interesting and illustrates an idea that will come up often. Notice that the error type is `None.type`. This is because an `Option` only has one failure mode. Either an `Option[A]` is a `Some[A]` with a value or it is a `None`, with no other information.

So an `Option` can fail, but there is essentially only one way it could ever fail—with the value `None`. The type of this lone failure value is `None.type`.

These are not the only effect constructors for pure values. In the exercises at the end of this chapter, you will explore a few of the other constructors.

2.7.3 Effect Constructors for Side Effecting Computations

The most important effect constructors are those for side-effecting computations. These constructors convert procedural code into ZIO effects, so they become blueprints that separate the *what* from the *how*.

Earlier in this chapter, we introduced `ZIO.effect`. This constructor captures side-effecting code, and defers its evaluation until later, translating any exceptions thrown in the code into `ZIO.fail` values.

Sometimes, however, we want to convert side-effecting code into a ZIO effect, but we know the side-effecting code does not throw any exceptions. For example, checking the system time or generating a random variable are definitely side-effects, but they cannot throw exceptions.

For these cases, we can use the constructor `ZIO.effectTotal`, which converts procedural code into a ZIO effect that cannot fail:

```
object ZIO {
  def effectTotal[A](a: => A): ZIO[Any, Nothing, A]
}
```

2.7.3.1 Converting Async Callbacks

A lot of code in the JVM ecosystem is non-blocking. Non-blocking code doesn't synchronously compute and return a value. Instead, when you call an asynchronous function, you must provide a callback, and then later, when the value is available, your callback will be invoked with the value. (Sometimes this is hidden behind `Future` or some other asynchronous data type.)

For example, let's say we have the non-blocking query API shown in the following snippet:

```
def getUserByIdAsync(id: Int)(cb: Option[String] => Unit): Unit =
  ???
```

If we give this function an `id` that we are interested in, it will look up the user in the background, but return right away. Then later, when the user has been retrieved, it will invoke the callback function that we pass to the method.

The use of `Option` in the type signature indicates that there may not be a user with the `id` we requested.

In the following code snippet, we call `getUserByIdAsync`, and pass a callback that will simply print out the name of the user when it is received:

```
getUserByIdAsync(0) {
  case Some(name) => println(name)
  case None =>       println("User not found!")
}
```

Notice that the call to `getUserByIdAsync` will return almost immediately, even though it will be some time (maybe even seconds or minutes) before our callback is invoked, and the name of the user is actually printed to the console.

Callback based APIs can improve performance, because we can write more efficient code that doesn't waste threads. But working directly with callback-based asynchronous code can be quite painful, leading to highly nested code, making it difficult to propagate success and error information to the right place, and making it impossible to handle resources safely.

Fortunately, like Scala's `Future` before it, ZIO allows us to take asynchronous code, and convert it to ZIO functional effects.

The constructor we need to perform this conversion is `ZIO.effectAsync`, and its type signature is shown in the following snippet:

```
object ZIO {
  def effectAsync[R, E, A](
    cb: (ZIO[R, E, A] => Unit) => Any
  ): ZIO[R, E, A] =
    ???
}
```

The type signature of `ZIO.effectAsync` can be tricky to understand, so let's look at an example.

To convert the `getUserByIdAsync` procedural code into ZIO, we can use the `ZIO.effectAsync` constructor as follows:

```
def getUserById(id: Int): ZIO[Any, None.type, String] =
  ZIO.effectAsync { callback =>
    getUserByIdAsync(id) {
      case Some(name) => callback(ZIO.succeed(name))
      case None       => callback(ZIO.fail(None))
    }
  }
```

The callback provided by `effectAsync` expects a ZIO effect, so if the user exists in the database, we convert the username into a ZIO effect using `ZIO.succeed`, and then invoke the callback with this successful effect. On the other hand, if the user does not exist, we convert `None` into a ZIO effect using `ZIO.fail`, and we invoke the callback with this failed effect.

We had to work a little to convert this asynchronous code into a ZIO function, but now we never need to deal with callbacks when working with this query API. We can now treat `getUserById` like any other ZIO function, and compose its return value with methods like `flatMap`, all without ever blocking, and with all of the guarantees that ZIO provides us around resource safety.

As soon as the result of the `getUserById` computation is available, we will just continue with the other computations in the blueprint we have created.

Note here that in `effectAsync` the callback function may only be invoked once, so it's not appropriate for converting all asynchronous APIs. If the callback may be invoked more than once, you can use the `effectAsync` constructor on `ZStream`, discussed later in this book.

The final constructor we will cover in this chapter is `ZIO.fromFuture`, which converts a function that creates a `Future` into a ZIO effect.

The type signature of this constructor is as follows:

```
def fromFuture[A](make: ExecutionContext => Future[A]): Task[A] =
  ???
```

Because a `Future` is a running computation, we have to be quite careful in how we do this. The `fromFuture` constructor doesn't take a `Future`. Rather, the constructor takes a function `ExecutionContext => Future[A]`, which describes how to make a `Future` given an `ExecutionContext`.

Although you don't need to use the provided `ExecutionContext` when you convert a `Future` into a ZIO effect, if you do use the context, then ZIO can manage where the `Future` runs at higher-levels.

If possible, we want to make sure that our implementation of the `make` function creates a new `Future`, instead of returning a `Future` that is already running. The following code shows an example of doing just this:

```
def goShoppingFuture(
  implicit ec: ExecutionContext
): Future[Unit] =
  Future(println("Going to the grocery store"))

val goShoppingTask: Task[Unit] =
  Task.fromFuture(implicit ec => goShoppingFuture)
```

There are many other constructors to create ZIO effects from other data types such as `java.util.concurrent.Future`, and third-party packages to provide conversion from Monix, Cats Effect, and other data types.

2.8 Standard ZIO Services

Earlier in this chapter, we talked about the ZIO environment type, but we haven't used it to write any programs. We will cover the environment in depth later in this book, and show how the environment provides a comprehensive solution for the dependency injection problem.

For now, we will talk about the basic *services* that ZIO provides for every application and how we can use them. *Services* provide well-defined interfaces that can be implemented differently in testing environments and production environments.

ZIO provides four to five different default services for all applications, depending on the platform:

1. **Clock**. Provides functionality related to time and scheduling. If you are accessing the current time or scheduling a computation to occur at some point in the future you are using this.
2. **Console**. Provides functionality related to console input and output.
3. **System**. Provides functionality for getting system and environment variables.
4. **Random**. Provides functionality for generating random values.
5. **Blocking**. Provides functionality for running blocking tasks on a separate `Executor` optimized for these kinds of workloads. Because blocking is not supported on Scala.js, this service is only available on the JVM.

Because this essential system functionality is provided as services, you can trivially test any code that uses these services, without actually interacting with production implementations.

For example, the `Random` services allows us to generate random numbers. The `Live` implementation of that service just delegates to `scala.util.Random`. But that may not always be the implementation we want. `scala.util.Random` is non-deterministic, which can be useful in production but can make it harder for us to test our programs.

For testing the random service, we may want to use a purely functional pseudo-random number generator, which always generates the same values given the same initial seed. This way, if a test fails, we can reproduce the failure and debug it.

ZIO Test, a toolkit for testing ZIO applications that we will discuss in a later chapter, provides a `TestRandom` that does exactly this. In fact, ZIO Test provides a test implementation of each of the standard services, and you can imagine wanting to provide other implementations, as well.

ZIO provides a default implementation of an `Executor` optimized for blocking tasks, but there are a variety of other ways you could tune an `Executor` for specific usage patterns.

By defining functionality in terms of well-defined interfaces, we defer concrete implementations until later. As you will see, using these services with ZIO is very easy, but at the same time, power users have tremendous flexibility to provide custom implementations of these services (or those you define in your own application).

The second, smaller benefit of using services is that you can use the feature to document which capabilities are being used in an effect. If we see an effect with a type signature of `ZIO[Clock, Nothing, Unit]` we can conclude that this effect, which cannot fail, is using functionality related to time or scheduling (probably not functionality related to random numbers).

However, this benefit is smaller because it is true only insofar as your team is very disciplined about coding to interfaces, instead of implementations. Because effect constructors can wrap any side-effecting code into a ZIO value, there is nothing stopping us from writing code like the following:

```
val int: ZIO[Any, Nothing, Int] =
  ZIO.effectTotal(scala.util.Random.nextInt())
```

In this snippet, we are generating random numbers, but this is not reflected in the type signature because we are wrapping `scala.util.Random` directly, instead of using the methods on the `Random` service. Unfortunately, the compiler cannot check this for us, so ensuring developers code to interfaces must be enforced with code review.

As such, we consider the ability to "see" what capabilities a computation uses as a *secondary* and optional benefit of using services. The primary benefit is just being able to plug in different implementations in testing and production environments.

Now let's discuss each of the standard ZIO services in some detail.

2.8.1 Clock

The `Clock` service provides functionality related to time and scheduling. This includes several methods to obtain the current time in different ways (`currentTime` to return the current time in the specified `TimeUnit`, `currentDateTime` to return the current `OffsetDateTime`, and `nanoTime` to obtain the current time in nanoseconds).

In addition, the `Clock` service includes a `sleep` method, which can be used to sleep for a certain amount of time.

The signature of `nanoTime` and `sleep` are shown in the following snippet:

```
import zio.duration._

package object clock {
  def nanoTime: URIO[Clock, Long]
  def sleep(duration: => Duration): URIO[Clock, Unit]
}
```

The `sleep` method is particularly important. It does not complete execution until the specified duration has elapsed, and like all ZIO operations, it is non-blocking, so it doesn't actually consume any threads while it is waiting for the time to elapse.

We could use the `sleep` method to implement the `delay` operator that we saw earlier in this chapter:

```
import zio.clock._
import zio.duration._
```

```
def delay[R, E, A](zio: ZIO[R, E, A])(
  duration: Duration
): ZIO[R with Clock, E, A] =
  clock.sleep(duration) *> zio
```

The `Clock` service is the building block for all time and scheduling functionality in ZIO. Consequently, you will see the `Clock` service as a component of the environment whenever working with retrying, repetition, timing, or other features related to time and scheduling built into ZIO.

2.8.2 Console

The `Console` service provides functionality around reading from and writing to the console.

So far in this book, we have been interacting with the console by converting procedural code in the Scala library to ZIO effects, using the `ZIO.effect` constructor. This was useful to illustrate how to translate procedural to ZIO, and demonstrate there is no "magic" in ZIO's own console facilities.

However, wrapping console functionality directly is not ideal, because we cannot provide alternative implementations for testing environments. In addition, there are some tricky edge corner cases for console interaction that the `Console` services handles for us. (For example, reading from the console can fail only with an `IOException`.)

The key methods on the `Console` service are `getStrLn`, which is analogous to `readLine()` and `putStrLn`, which is the equivalent of `println`. There is also a `putStr` method if you do not want to add a newline after printing text to the console.

```
package object console {
  val getStrLn: ZIO[Console, IOException, String]
  def putStr(line: => String): URIO[Console, Unit]
  def putStrLn(line: => String): URIO[Console, Unit]
```

The `Console` service is commonly used in console applications, but is less common in generic code than `Clock` or `Random`.

In the rest of this book, we will illustrate examples involving console applications with these methods, rather than converting methods from the Scala standard library.

2.8.3 System

The `System` service provides functionality to get system and environment variables.

```
package object system {
  def env(variable: String): IO[SecurityException, Option[String]]
  def property(prop: String): IO[Throwable, Option[String]]
```

The two main methods on the `System` service are `env`, which accesses a specified environment variable, and `property`, which accesses a specified system property. There are also other variants for obtaining all environment variables or system properties, or specifying a backup value, if a specified environment variable or property does not exist.

Like the `Console` service, the `System` service tends to be used more in applications or certain libraries (e.g. those dealing with configuration) but is uncommon in generic code.

2.8.4 Random

The `Random` service provides functionality related to random number generation. The `Random` service exposes essentially the same interface as `scala.util.Random`, but all the methods return functional effects. So if you're familiar with code like `random.nextInt(6)` from the standard library, you should be very comfortable working with the `Random` service.

The `Random` service is sometimes used in generic code in scheduling, such as when adding a random delay between recurrences of some effect.

2.8.5 Blocking

The `Blocking` service supports running blocking effects on an `Executor` optimized for blocking tasks. By default, the ZIO runtime is optimized for asynchronous and computationally-bound tasks, with a small fixed number of threads that perform all work.

Although this choice optimizes throughput for async and CPU tasks, it means that you could potentially exhaust all of ZIO's default threads if you run blocking I/O operations on them. Therefore, it is critical that blocking tasks be run on a separate blocking thread pool, which is optimized for these workloads.

The `Blocking` service has several methods to support this use case.

The most fundamental is the `blocking` operator, which takes an effect and ensures it will be run on the blocking thread pool.

```
import zio.blocking._

object blocking {
  def blocking[R <: Blocking, E, A](
    zio: ZIO[R, E, A]
  ): ZIO[R, E, A] =
```

```
    ???
}
```

Let's say that the database query described above was not implemented in terms of a callback based API but instead synchronously waited until the result was available (thereby blocking the calling thread). In this case, we would want to make sure the effect is executed on the blocking thread pool by using the `blocking` operator, as shown in the following snippet:

```
import zio.blocking._

def getUserById(id: Int): IO[Unit, String] =
  ???

def getUserByIdBlocking(id: Int): ZIO[Blocking, Unit, String] =
  blocking(getUserById(id))
```

Notice that the `Blocking` service is now a dependency in the type signature, clarifying that the returned effect involves blocking IO, in a way its previous signature did not.

If you want to construct an effect from a blocking side-effect directly, you can use the `effectBlocking` constructor, which is equivalent to `blocking(ZIO.effect(...))`.

2.9 Recursion And ZIO

We talked earlier in this chapter about using `flatMap` and related operators to compose effects sequentially.

Ordinarily, if you call a recursive function, and it recurses deeply, the thread running the computation may run out of stack space, which will result in your program throwing a stack overflow exception.

One of the features of ZIO is that ZIO effects are stack-safe for arbitrarily recursive effects. So we can write ZIO functions that call themselves to implement any kind of recursive logic with ZIO.

For example, let's say we want to implement a simple console program that gets two integers from the user and multiplies them together.

We can start by implementing an operator to get a single integer from the user, as shown in the following snippet:

```
import zio.console._

val readInt: RIO[Console, Int] =
  for {
    line <- console.getStrLn
```

```
    int  <- ZIO.effect(line.toInt)
  } yield int
```

This effect can fail with an error type of `Throwable`, because the input from the user might not be a valid integer. If the user doesn't enter a valid integer, we want to print a helpful error message to the user, and then try again.

We can build this functionality atop our existing `readInt` effect by using recursion. We define a new effect `readIntOrRetry` that will first call `readInt`. If `readInt` is successful, we just return the result. If not, we prompt the user to enter a valid integer, and then recurse:

```
lazy val readIntOrRetry: URIO[Console, Int] =
  readInt
    .orElse(console.putStrLn("Please enter a valid integer")
    .zipRight(readIntOrRetry))
```

Using recursion, we can create our own sophisticated control flow constructs for our ZIO programs.

2.10 Conclusion

Functional effects are blueprints for concurrent workflows, immutable values that offer a variety of operators for transforming and combining effects to solve more complex problems.

The ZIO type parameters allow us to model effects that require context from an environment before they can be executed; they allow us to model failure modes (or a lack of failure modes); and they allow us to describe the final successful result that will be computed by an effect.

ZIO offers a variety of ways to create functional effects from synchronous code, asynchronous code, pure computations, and impure computations. In addition, ZIO effects can be created from other data types built into the Scala standard library.

ZIO uses the environment type parameter to make it easy to write testable code that interacts with interfaces, without the need to manually propagate those interfaces throughout the entire application. Using this type parameter, ZIO ships with standard services for interacting with the console, the system, random number generation, and a blocking thread pool.

With these tools, you should be able to write your own simple ZIO programs, convert existing code you have written into `ZIO` using effect constructors, and leverage the functionality built into ZIO.

2.11 Exercises

1. Implement a ZIO version of the function `readFile` by using the `ZIO.effect` constructor.

```
def readFile(file: String): String = {
  val source = scala.io.Source.fromFile(file)

  try source.getLines.mkString finally source.close()
}

def readFileZio(file: String) = ???
```

2. Implement a ZIO version of the function `readFile` by using the `ZIO.effect` constructor.

```
def writeFile(file: String, text: String): Unit = {
  import java.io._
  val pw = new PrintWriter(new File(file))
  try pw.write(text) finally pw.close
}

def writeFileZio(file: String, text: String) = ???
```

3. Using the `flatMap` method of ZIO effects, together with the `readFileZio` and `writeFileZio` functions that you wrote, implement a ZIO version of the function `copyFile`.

```
def copyFile(source: String, dest: String): Unit = {
  val contents = readFile(source)
  writeFile(dest, contents)
}

def copyFileZio(source: String, dest: String) = ???
```

4. Rewrite the following ZIO code that uses `flatMap` into a *for comprehension*.

```
def printLine(line: String) = ZIO.effect(println(line))
val readLine = ZIO.effect(scala.io.StdIn.readLine())

printLine("What is your name?").flatMap(_ =>
  readLine.flatMap(name =>
    printLine(s"Hello, ${name}!")))
```

5. Rewrite the following ZIO code that uses `flatMap` into a *for comprehension*.

```
val random = ZIO.effect(scala.util.Random.nextInt(3) + 1)
def printLine(line: String) = ZIO.effect(println(line))
val readLine = ZIO.effect(scala.io.StdIn.readLine())
```

```
random.flatMap(int =>
  printLine("Guess a number from 1 to 3:").flatMap(_ =>
    readLine.flatMap(num =>
      if (num == int.toString) printLine("You guessed right!")
      else printLine(s"You guessed wrong, the number was ${int}!")))
)
```

6. Implement the `zipWith` function in terms of the toy model of a ZIO effect. The function should return an effect that sequentially composes the specified effects, merging their results with the specified user-defined function.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def zipWith[R, E, A, B, C](
  self: ZIO[R, E, A],
  that: ZIO[R, E, B]
)(f: (A, B) => C): ZIO[R, E, C] =
    ???
```

7. Implement the `collectAll` function in terms of the toy model of a ZIO effect. The function should return an effect that sequentially collects the results of the specified collection of effects.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def collectAll[R, E, A](
  in: Iterable[ZIO[R, E, A]]
): ZIO[R, E, List[A]] =
  ???
```

8. Implement the `foreach` function in terms of the toy model of a ZIO effect. The function should return an effect that sequentially runs the specified function on every element of the

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def foreach[R, E, A, B](
  in: Iterable[A]
)(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
    ???
```

9. Implement the `orElse` function in terms of the toy model of a ZIO effect. The function should return an effect that tries the left hand side, but if that effect fails, it will fallback to the effect on the right hand side.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def orElse[R, E1, E2, A](
  self: ZIO[R, E1, A],
```

```
  that: ZIO[R, E2, A]
): ZIO[R, E2, A] =
  ???
```

10. Using the following code as a foundation, write a ZIO application that prints out the contents of whatever files are passed into the program as command-line arguments. You should use the functions `readFileZio` and `writeFileZio` that you developed in these exercises, as well as `ZIO.foreach`.

```
import zio.{ App => ZIOApp }

object Cat extends ZIOApp {
  def run(commandLineArguments: List[String]) = ???
}
```

11. Using `ZIO.fail` and `ZIO.succeed`, implement the following function, which converts an `Either` into a ZIO effect:

```
def eitherToZIO[E, A](either: Either[E, A]): ZIO[Any, E, A] = ???
```

12. Using `ZIO.fail` and `ZIO.succeed`, implement the following function, which converts a `List` into a ZIO effect, by looking at the head element in the list and ignoring the rest of the elements.

```
def listToZIO[A](list: List[A]): ZIO[Any, None.type, A] = ???
```

13. Using `ZIO.effectTotal`, convert the following procedural function into a ZIO function:

```
def currentTime(): Long = System.currentTimeMillis()

lazy val currentTimeZIO: ZIO[Any, Nothing, Long] = ???
```

14. Using `ZIO.effectAsync`, convert the following asynchronous, callback-based function into a ZIO function:

```
def getCacheValue(
  key: String,
  onSuccess: String => Unit,
  onFailure: Throwable => Unit
): Unit =
  ???

def getCacheValueZio(key: String): ZIO[Any, Throwable, String] = ???
```

15. Using `ZIO.effectAsync`, convert the following asynchronous, callback-based function into a ZIO function:

```
trait User

def saveUserRecord(
```

```
  user: User,
  onSuccess: () => Unit,
  onFailure: Throwable => Unit
): Unit =
  ???

def saveUserRecordZio(user: User): ZIO[Any, Throwable, Unit] = ???
```

16. Using `ZIO.fromFuture`, convert the following code to ZIO:

```
import scala.concurrent.{ ExecutionContext, Future }
trait Query
trait Result

def doQuery(query: Query)(
  implicit ec: ExecutionContext): Future[Result] =
    ???

def doQueryZio(query: Query): ZIO[Any, Throwable, Result] =
  ???
```

17. Using the `Console`, write a little program that asks the user what their name is, and then prints it out to them with a greeting.

```
import zio. { App => ZIOApp }

object HelloHuman extends ZIOApp {
  def run(args: List[String]) = ???
}
```

18. Using the `Console` and `Random` services in ZIO, write a little program that asks the user to guess a randomly chosen number between 1 and 3, and prints out if they were correct or not.

```
import zio._

object NumberGuessing extends ZIOApp {
  def run(args: List[String]) = ???
}
```

19. Using the `Console` service and recursion, write a function that will repeatedly read input from the console until the specified user-defined function evaluates to `true` on the input.

```
import java.io.IOException

import zio.console._

def readUntil(
  acceptInput: String => Boolean
```

```
): ZIO[Console, IOException, String] =
  ???
```

20. Using recursion, write a function that will continue evaluating the specified effect, until the specified user-defined function evaluates to `true` on the output of the effect.

```
def doWhile[R, E, A](
  body: ZIO[R, E, A]
)(condition: A => Boolean): ZIO[R, E, A] =
  ???
```

Chapter 3

Essentials: Testing ZIO Programs

In addition to writing programs to express our desired logic it is important to test those programs to ensure that their behavior conforms to our expectations.

This is particularly important for ZIO and libraries in its ecosystem because a major focus of ZIO is *composability*, meaning that we can assemble solutions to more complex problems from a small number of building blocks and operators for putting them together. It is only possible to do this if each of the building blocks and operators honors *guarantees* about its expected behavior so we can reason about what guarantees we can expect the solution to provide.

For example, consider this simple program.

```
import zio.ZIO

def safeDivision(x: Int, y: Int): ZIO[Any, Unit, Int] =
  ZIO.effect(x / y).catchAll(t => ZIO.fail(()))
```

This program just tries to divide two integers, returning the result if it is defined or failing with the `Unit` value if it is not, for example because we are dividing by zero.

This program seems very simple and its behavior may seem obvious to us if we are familiar with the operators from the last chapter. But the only reason we are able to reason about it in this way is that each of the constructors and each of the operators for combining them honor certain guarantees.

In the example above the `effect` constructor guarantees that it will catch any non-fatal exception thrown while evaluating its argument and return that exception as the failed result of a `ZIO` effect, or otherwise return the result of evaluating its argument as a successful `ZIO` effect.

The `effect` constructor has to catch any non-fatal exception, regardless of its type, and it has to catch the exception every time. If no exceptions are thrown it has to return the result of evaluating its argument and can't change that result in any way.

This is what allows us to reason that when we divide two numbers within the `ZIO.effect` constructor we will get back either a successful `ZIO` effect with the result of the division if it is defined, or a failed `ZIO` effect with the error if it is not defined.

Similarly, the `catchAll` operator takes an error handler and has to apply the error handler to the failed result of the original effect, returning the result of the error handler if the original effect failed or the successful result of the original effect unchanged if it succeeded.

In this case these guarantees were relatively obvious and we probably did not even need to think about them. But as we learn more in this book we will see that much of the power of ZIO comes from the guarantees it gives us that are less obvious, for example that if a resource is acquired it will always be released or that if our program is interrupted all parts of it will immediately be shut down as quickly as possible.

These powerful guarantees, along with understanding how these guarantees apply when we compose different programs together, is how we build complex programs that still have very strong safety and efficiency properties for ourselves and our users. So being able to verify through testing that the components we are creating really do honor the guarantees we think they do is critical.

Of course there are already a variety of other testing frameworks in Scala. For example, here is how we could test a simple assertion using the ScalaTest library, which you may have learned about in one of your introductory courses on Scala.

```
import org.scalatest._

class ExampleSpec extends FunSuite {
  test("addition works") {
    assert(1 + 1 === 2)
  }
}
```

This works but it runs into some problems when we want to make assertions about ZIO effects instead of simple values. For example, here is an initial attempt to test a simple assertion about a ZIO effect.

```
class ExampleSpec2 extends FunSuite {
  test("addition works") {
    assert(ZIO.succeed(1 + 1) === 2)
  }
}
```

This compiles but this test doesn't make any sense and will always fail because we are comparing two completely unrelated types. The left hand side is a `ZIO` effect that is a blueprint for a concurrent program that will eventually return an `Int` whereas the right hand side is just an `Int`.

What we really want is not to say that `ZIO.succeed(1 + 1)` is equal to `2` but rather that the *result* of *evaluating* `ZIO.succeed(1 + 1)` is equal to `2`.

We can express this by creating a ZIO `Runtime` and using its `unsafeRun` method to run the ZIO effect, transforming the ZIO blueprint of a concurrent program into the result of actually running that program. This is similar to what the ZIO `App` trait did for us automatically in the previous chapter.

```
import zio.Runtime

val runtime: Runtime[Any] = Runtime.default
// runtime: Runtime[Any] = zio.Runtime$$anon$3@7ed6bf7f

class ExampleSpec3 extends FunSuite {
  test("addition works") {
    assert(runtime.unsafeRun(ZIO.succeed(1 + 1)) === 2)
  }
}
```

This test now makes sense and will pass as written but there are still several problems with it.

First, this won't work at all on Scala.js. The `unsafeRun` method runs a `ZIO` effect to produce a value, which means it needs to block until the result is available, but we can't block on Scala.js!

We could use even more complicated methods where we run the `ZIO` effect to a `scala.concurrent.Future` and then interface with functionality in ScalaTest for making assertions about `Future` values but we are already introducing quite a bit of complexity here for what should be a simple test.

In addition, there is a more fundamental problem. ScalaTest doesn't know anything about ZIO, its environment type, its error type, or any of the operations that ZIO supports. So ScalaTest can't take advantage of any of the features of ZIO when implementing functionality related to testing.

For example, ScalaTest has functionality for timing out a test that is taking too long.

But since as we learned in the previous chapter `Future` is not interruptible this "time out" just fails the test after the specified duration. The test is still running in the background, potentially consuming system resources.

ZIO has support for interruption, but there is no way for ScalaTest's timeout to integrate with this interruption since ScalaTest does not know anything about

ZIO, short of us doing extremely manual plumbing that takes us away from the work we are trying to do of expressing our testing logic.

Fundamentally, the problem is that most testing libraries treat effects as *second class citizens*. They are only things to be run to produce "real" values like `Int`, `String`, or possibly `Future`, which are the things the test framework understands.

The result of this is that we end up discarding all the power of ZIO when we go to write our tests with testing frameworks like this, which is painful because we have just gotten used to the power and composability of ZIO in writing our production code.

3.1 Writing Simple Programs With ZIO Test

The solution to this is *ZIO Test*, a testing library that treats effects as *first class values* and leverages the full power of ZIO.

To get started with ZIO Test, first add it as a dependency.

```
libraryDependencies ++= Seq(
  "dev.zio" %% "zio-test"     % zioVerzion
  "dev.zio" %% "zio-test-sbt" % zioVersion,
)
```

From there we can write our first test by extending `DefaultRunnableSpec` and implementing its `spec` method.

```
import zio.test._
import zio.test.Assertion._

object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    test("addition works") {
      assert(1 + 1)(equalTo(2))
    }
  )
}
```

So far this doesn't look that different from other testing frameworks.

Each collection of tests is represented as a spec that can either be a test or a suite containing one or more other specs. In this way, a spec is a tree like data structure that can support arbitrary levels of nesting of suites and tests, allowing a great deal of flexibility in how you organize your tests.

We write tests using the `assert` operator, which takes first a value that we are making an assertion about and then an assertion that we expect to hold for that value. Here we are using the simple `equalTo` assertion which just expects the

value to be equal to the argument to `equalTo`, but as we will see in the next section we can have a variety of other assertions that express more complicated expectations.

Where things really get interesting is when we want to start testing effects. Let's look at how we would test that `ZIO.succeed` succeeds with the expected value that we were struggling with before.

```
object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("ZIO.succeed succeeds with specified value") {
      assertM(ZIO.succeed(1 + 1))(equalTo(2))
    }
  )
}
```

Did you catch the difference? Beyond replacing `1 + 1` with `ZIO.succeed(1 + 1)` the only change we made is replacing `test` with `testM` and `assert` with `assertM`.

Replacing `test` with `testM` tells the test framework that the test will return a `ZIO` effect. The test framework will automatically take care of running the test along with all the other tests in the spec and reporting the results in a way that is consistent across platforms.

Similarly replacing `assert` with `assertM` indicates that the left hand side of the assertion will be a `ZIO` effect and the test framework should run the left hand side and compare its result to the expectation on the right hand side.

There is nothing magical about `assertM` here. In fact we can replace `assertM` with `assert` using `map` or a `for` comprehension.

```
object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("testing an effect using map operator") {
      ZIO.succeed(1 + 1).map(n => assert(n)(equalTo(2)))
    },
    testM("testing an effect using a for comprehension") {
      for {
        n <- ZIO.succeed(1 + 1)
      } yield assert(n)(equalTo(2))
    }
  )
}
```

All three ways of writing this test are equivalent.

In general, we find that using `assertM` is most readable when the entire test fits

on a single line and using a `for` comprehension is preferable otherwise, but you can pick the style that works for you.

You can also use `&&` and `||` to combine multiple `assert` statements using logical conjunction and disjunction or `!` to negate an assertion.

```
object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("and") {
      for {
        x <- ZIO.succeed(1)
        y <- ZIO.succeed(2)
      } yield assert(x)(equalTo(1)) &&
        assert(y)(equalTo(2))
    }
  )
}
```

3.2 Using Assertions

In the examples above we used the `equalTo` assertion, which is one of the most basic assertions. You can get quite far using just the `equalTo` assertion, but there are a variety of other assertions that come in handy in certain situations.

A useful way to think of an `Assertion[A]` is as a function that takes an `A` value and returns a `Boolean` indicating either `true` if the value satisfies the assertion or false if it does not.

```
type Assertion[-A] = A => Boolean

def equalTo[A](expected: A): Assertion[A] =
  actual => actual == expected
```

This is not exactly how `Assertion` is implemented because the data type returned by running an assertion on a value needs to contain some additional information to support reporting test results. However, this should give you a good mental model for an assertion similar to the toy ZIO implementation we worked through in the previous chapter.

There are a variety of assertions in the `Assertion` companion object in the `zio.test` package. For now we will just provide a few examples to show their capabilities.

Assertions can be specialized for particular data types so there are a variety of assertions that express more complex logic that may be harder for us to implement directly.

For example, when working with collections we may want to assert that two

collections have the same elements, even if they do not appear in identical order. We can easily do this using the `hasSameElements` assertion.

```
object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    test("hasSameElement") {
      assert(List(1, 1, 2, 3))(hasSameElements(List(3, 2, 1, 1)))
    }
  )
}
```

Another assertion that is particularly useful is the `fails` assertion, which allows us to assert that an effect fails with a particular value. We can use this by first calling `run` on our effect to obtain a `ZIO` effect that succeeds with an `Exit` value representing the result of the original effect and then using the `fails` assertion with that `Exit` value.

```
object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("fails") {
      for {
        exit <- ZIO.effect(1 / 0).catchAll(_ => ZIO.fail(())).run
      } yield assert(exit)(fails(isUnit))
    }
  )
}
```

One other thing you may notice here is that many assertions take other assertions as arguments. This allows you to express more specific assertions that "zero in" on part of a larger value.

In the example above, the `fails` assertion required that the result of the `ZIO` effect be a failure and then allowed us to provide another argument to make a more specific assertion about what that failure value must be. In this case we just used the `isUnit` assertion which is a shorthand for `equalTo(())` but we could have used whatever assertion we wanted.

If you ever get to a point where you don't care about the specific value, for example you just care that the effect failed and don't care about how it failed, you can use the `anything` assertion to express an assertion that is always true.

Another nice feature about assertions is that we can compose them using logical conjunction, disjunction, and negation.

For example, suppose we want to assert that a collection of integers has at least one value and that all of the values are greater than or equal to zero. We could do that like this:

```
val assertion: Assertion[Iterable[Int]]=
  isNonEmpty && forall(nonNegative)
// assertion: Assertion[Iterable[Int]] = (isNonEmpty() && forall(isGreaterThanEqualTo(0)))
```

Similarly, we can express alternatives. For example, we might want to express the expectation that a collection is either empty or contains exactly three elements.

```
val assertion: Assertion[Iterable[Any]] =
  isEmpty || hasSize(equalTo(3))
// assertion: Assertion[Iterable[Any]] = (isEmpty() || hasSize(equalTo(3)))
```

We can also negate assertions using the `not` assertion. For example, we could express an expectation that a collection contains at least one duplicate element like this:

```
val assertion: Assertion[Iterable[Any]] =
  not(isDistinct)
// assertion: Assertion[Iterable[Any]] = not(isDistinct())
```

3.3 Test Implementations Of Standard ZIO Services

One of the common issues we run into when testing ZIO programs that ZIO Test can help us with is testing effects that use ZIO's standard services.

For example, consider this simple console program.

```
import zio.console._

val greet: ZIO[Console, Nothing, Unit] =
  for {
    name <- getStrLn.orDie
    _    <- putStrLn(s"Hello, $name!")
  } yield ()
// greet: ZIO[Console, Nothing, Unit] = zio.ZIO$FlatMap@29b82f97
```

This is a very simple program so we might be relatively confident it is correct but how would we go about testing it?

We could run the program ourself and verify that we receive the expected console output but that is extremely manual and will likely result in very minimal test coverage of potential console inputs and lack of continuous integration as other parts of our code base change. So we don't want to do that.

But how else do we test it? `getStrLn` is going to read an actual line from the console and `putStrLn` is going to print an actual line to the console, so how do we supply the input and verify that the output is correct without actually doing it ourselves?

This is where the fact that `Console` is a service in the environment comes to the rescue. Because `Console` is a service we can provide an alternative implementation for testing, for example one that "reads" lines from an input buffer that we have filled with appropriate inputs and "writes" lines to an output buffer that we can examine.

And ZIO Test does just this, providing `TestConsole`, `TestClock`, `TestRandom`, and `TestSystem` implementations of all the standard ZIO services that are fully deterministic to facilitate testing.

ZIO Test will automatically provide a copy of these services to each of our tests, making this extremely easy. Generally all we need to do is call a couple of specific "test" methods to provide the desired input and verify the output.

To see this, let's look at how we could test the console program above.

```
import zio.test.environment._

object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("greet says hello to the user") {
      for {
        _     <- TestConsole.feedLines("Jane")
        _     <- greet
        value <- TestConsole.output
      } yield assert(value)(equalTo(Vector("Hello, Jane!\n")))
    }
  )
}
```

We have now gone from a program that was not testable at all to one that is completely testable. We could now provide a variety of different inputs, potentially even using ZIO Test's support for property based testing described below, and include this in our continuous integration process to obtain a very high level of test coverage here.

Note that a separate copy of each of these services is automatically provided to each of your tests, so you don't have to worry about interference between tests when working with these test services.

Another test service that is particularly useful for testing concurrent programs is the `TestClock`. As we saw in the last chapter, we often want to schedule events to occur after some specified duration, for example to `goShopping` in one hour, and we would like to verify that the events really do occur after the specified duration.

Again, we face a problem of testing. Do we have to wait an hour for `goShopping` to execute to verify that it is being scheduled correctly?

No! The `TestClock` allows us to deterministically test effects involving time without waiting for real time to pass.

Here is how we could test a method that delays for a specified period of time using the `TestClock`.

```
import zio.clock._
import zio.duration._

val goShopping: ZIO[Console with Clock, Nothing, Unit] =
  putStrLn("Going shopping!").delay(1.hour)
// goShopping: ZIO[Console with Clock, Nothing, Unit] = zio.ZIO$FlatMap@4a841724

object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("goShopping delays for one hour") {
      for {
        fiber <- goShopping.fork
        _     <- TestClock.adjust(1.hour)
        _     <- fiber.join
      } yield assertCompletes
    }
  )
}
```

We are introducing a couple of new concepts here with the `fork` and `join` operators that we will learn about more fully in a couple of chapters, but `fork` here is kicking off execution of `goShopping` as a separate logical process while the main program flow continues and `join` is waiting for that process to complete.

Since the `Clock` implementation being used is the `TestClock`, time only passes when adjusted by the user by calling operators such as `adjust`. Here `adjust(1.hour)` causes all effects scheduled to be run in one hour or less to immediately be run in order, causing `goShopping` to complete execution and allowing the program to terminate.

We use `assertCompletes` here, which is just an assertion that always is satisfied, to more clearly express our intent that what we are testing here is just that this program completes at all.

3.4 Common Test Aspects

Another nice feature of ZIO Test to be aware of is *test aspects*. Test aspects modify some aspect of how tests are executed. For example, a test aspect could time out a test after a specified duration or run a test a specified number of times to make sure it is not flaky.

We apply test aspects by using `spec @@ aspect` syntax like this:

```
import zio.test.TestAspect._

object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("this test will be repeated to ensure it is stable") {
      assertM(ZIO.succeed(1 + 1))(equalTo(2))
    } @@ nonFlaky
  )
}
```

In this case there probably isn't a need to use the `nonFlaky` aspect unless we have some reason to be particularly suspicious of `ZIO.succeed` but when we are testing concurrent programs that might be subject to subtle race conditions or deadlocks it can be extremely useful in turning rare bugs that we don't see until production into consistent test failures that we can diagnose and debug.

There are a variety of other test aspects we can use. For example, we can use `timeout` with a specified duration to time out a test that takes longer than the duration or we can use `failing` to specify that we expect a test to fail.

Since tests are themselves `ZIO` effects, timing out a test will actually interrupt the test, making sure that no unnecessary work is done and any resources acquired in connection with the test are appropriately released.

One feature of test aspects that is particularly nice is that you can apply them to either individual tests or entire suites, modifying all the tests in the suite. So if you want to apply a timeout to each test in a suite just call `timeout` on the suite.

There are many different test aspects to modify how tests are executed, for example only running tests on a certain platform or Scala version. So if you do need to modify something about how your tests are executed it is worth checking whether there is already a test aspect for that.

3.5 Basic Property Based Testing

Another important feature to be aware of as you start writing tests is that ZIO Test has support for property based testing out of the box.

In property based testing, instead of you manually generating inputs and verifying the expected outputs, the test framework generates a whole collection of inputs from a *distribution* of potential inputs that you specify and verifies that the expectation holds for all the inputs.

Property based testing can be very good for maximizing developer productivity in writing tests and catching bugs that might not otherwise be found until

production because it lets the test framework immediately generate a large number of test cases, includes ones the developer might not have considered initially.

However, care must be taken with property testing to ensure that the right distribution of generated values is used, including a sufficient number of "corner cases" (e.g. empty collections, integers with minimum and maximum values) and a sufficient space of generated values to cover the range of values that might be seen in production (e.g. long strings, strings in non-ASCII character sets).

ZIO Test supports property based through its `Gen` data type and the `check` family of operators.

A `Gen[R, A]` represents a *generator* of `A` values that requires an environment `R`. Depending on the implementation generators may either be infinite or finite and may either be random or deterministic.

ZIO Test contains generators for a variety of standard data types in the `Gen` companion object. For example, we could create a generator of integer values using the `anyInt` generator.

```
import zio.random._

val intGen: Gen[Random, Int] =
  Gen.anyInt
// intGen: Gen[Random, Int] = Gen(zio.stream.ZStream$$anon$1@4865c270)
```

Once we have a generator we create a test using that generator using the `check` operator. For example:

```
object ExampleSpec extends DefaultRunnableSpec {

  def spec = suite("ExampleSpec")(
    testM("integer addition is associative") {
      check(intGen, intGen, intGen) { (x, y, z) =>
        val left = (x + y) + z
        val right = x + (y + z)
        assert(left)(equalTo(right))
      }
    }
  )
}
```

Notice how similar the way we write property based tests is to the way we write normal tests.

We still use the `testM` method to label a test. Property based tests will always use `testM` instead of `test` because running a property based test itself involves effects.

Inside our test we call the `check` operator, specifying as arguments each of the generators we want to use. ZIO Test has overloaded variants of the `check` operators for different numbers of generators so you can use `check` with a single generator or with several different generators as in the example above.

We then provide a function that has access to each of the generated values and returns a test result using either the `assert` or `assert` operator, just like we used in the tests we wrote above. The test framework will then repeatedly sample combinations of values and test those samples until it either finds a failure or tests a "sufficient" number of samples without finding a failure.

There are several variants of the `check` operator. The most important is `checkM` which is like `check` except that it allows us to perform effects within the property based test. There are also `checkN` variants that allow specifying how many samples to test and `checkAll` variants for testing all samples from a finite generator.

Much of the work in writing property based tests tends to be in writing the generators themselves. If the values we want to generate are data types from ZIO or the Scala standard library and we don't need any special distribution then we can often just use an existing generator, as with the `intGen` we used above.

When we need to create generators for our own data type we can use the existing `Gen` constructors and the operators on `Gen` to create the generator we need. Many of these operators will already be familiar to us from what we have learned about ZIO so far.

As a motivating example, say we want to create a generator for a `User` data type we have defined.

```
final case class User(name: String, age: Int)
```

Since `User` is a data type we defined there is no existing generator for `User` values in ZIO Test. Furthermore, based on our understanding of the domain we know that `User` values must satisfy certain properties that are not captured in the type signature.

1. Names always consist of ASCII characters
2. Ages always fall into natural lifespans for human adults, say between `18` and `120`

We can implement a generator for names using the existing `anyASCIIString` generator.

```
val genName: Gen[Random with Sized, String] =
  Gen.anyASCIIString
// genName: Gen[Random with Sized, String] = Gen(
//   zio.stream.ZStream$$anon$1@97622a4
// )
```

This generator requires a service we have not seen before, `Sized`, which is a service specific to ZIO Test that allows controlling the "size" of generated values, for example how large a list we should generate, or in this case how large a `String`.

For the age generator, we can use the `int` constructor which generates integer values within the specified range.

```
val genAge: Gen[Random, Int] =
  Gen.int(18, 120)
// genAge: Gen[Random, Int] = Gen(zio.stream.ZStream$$anon$1@5338d64e)
```

With these two generators implemented, all that is left is to combine them, conceptually sampling a name from `genName` and an age from `genAge` and combining the two to generate a `User` value. The `Gen` data type supports many of the operators we are already familiar with including `map`, `flatMap`, and `zipWith` so we can actually do this quite easily.

```
val genUser: Gen[Random with Sized, User] =
  for {
    name <- genName
    age  <- genAge
  } yield User(name, age)
// genUser: Gen[Random with Sized, User] = Gen(
//   zio.stream.ZStream$$anon$1@7ded7e05
// )
```

We now have a generator of `User` values that we can use in any of the `check` variants to generate `User` values for our property based tests!

3.6 Conclusion

This chapter has provided a brief overview of ZIO Test. There is much more to learn about ZIO Test and handling more complicated scenarios but this should give you the tools you need to start writing your own tests for ZIO programs.

As you proceed through this book we encourage you to create tests for the code you write as well as the examples we show and the guarantees we claim that ZIO data types provides.

When we say that a data structure is safe for concurrent access try updating it from multiple fibers and make sure you get the correct result. When we say that a finalizer will always be run even if an effect is interrupted, try interrupting it and verify that the finalizer is run.

By doing this you will not only build your skill set in writing tests for `ZIO` effects but also deepen your understanding of the material you are learning, verifying that the guarantees you expect are honored and potentially finding

some situations where your intuitions about the guarantees that apply need to be refined.

3.7 Exercises

Chapter 4

Essentials: The ZIO Error Model

Complex applications can fail in countless ways. They can fail because of bugs in our code. They can fail because of bad input. They can fail because the external services they depend on fail. They can fail because of lack of memory, lack of stack space, or hardware failure.

In addition, some of these failures are local and others are global, some recoverable, and others non-recoverable. If we want to build robust and resilient applications that work according to specification, then our only hope is to leverage Scala's type system to help us tame the massive complexity of error management.

This chapter introduces you to the full power of the ZIO error model. We've already learned about the error type and how this allows us to express the ways an effect can fail. But in this chapter, we'll cover more advanced error handling operators, learn how ZIO deals with effects that fail in unexpected ways, and see how ZIO keeps track of concurrent failures.

4.1 Exceptions Versus Defects

The ZIO error type allows us to see just by looking at the type signature all the ways that an effect can fail. But sometimes, a failure can occur in a way that is not supposed to happen.

For example, take the following snippet of code:

```
import zio._

val divisionByZero: UIO[Int] =
```

```
  UIO.effectTotal(1 / 0)
// divisionByZero: UIO[Int] = zio.ZIO$EffectTotal@33f9056
```

We used the `effectTotal` constructor here, which means the return type is `UIO`, indicating the effect cannot fail. But on the JVM, dividing by zero will throw an `ArithmeticException`. How is this failure handled?

The answer is that ZIO draws a distinction between two types of failures:

- **Errors.** Errors are potential failures that are represented in the error type of the effect. They model failure scenarios that are anticipated and potentially recoverable. When parsing an integer from a string, a `NumberFormatException` is an example of an error. We know that parsing can fail because the string may not be a valid integer, and there are a variety of ways we could recover (e.g. using a default value, parsing another integer, propagating the failure to a higher level of the application, etc.). These are sometimes called **typed failures** or **checked failures**.
- **Defects**. Defects are potential failures not represented in the error type of the effect. They model failure scenarios that are unanticipated or unrecoverable. For example, in a console program an `IOException` in reading from the console might be a defect. Our console program is based on console interaction with the user, so if we cannot even read from the console, there's nothing we can do except abort the program. These are also called **fiber failures**, **untyped failures**, or **unchecked failures**.

Whether to treat a certain type of failure as an error or a defect can involve some judgment. The same type of failure could be an error in one application, but a defect in another. Indeed, the same type of error could be a failure at one level of an application, and a defect at a higher-level of the same application.

For an example of the former, in a console application, there may be no way to recover from an `IOException` and so it could make sense to treat it as a defect. But in application that allows the user to enter console input to customize the settings for report generation it could make sense to treat that as an error and handle it by generating a report with default settings.

For an example of the latter, a failure to connect to a database might be considered an error in low-level code, because perhaps we have some way to recover, by retrying the connection or using a backup database. But at a higher-level, an application that cannot connect to any database will ultimately abort.

Some other effect types, as well as async data types like Scala's own `Future`, keep the error type rigidly fixed to `Throwable` because any program could potentially fail with a `Throwable`. Not only does this reduce flexibility, because there's no way to describe effects that can't fail or those that can fail with some business error, but it obscures the distinction between failures and defects.

If we had plans to meet a friend for dinner tonight, we might tell our friend if we get stuck on a work call that could run late. But it wouldn't really make sense

for us to tell our friend that we might be struck by lightening, or that there might be an earthquake while we're driving to meet our friend.

Communicating information about anticipated failures that are potentially recoverable is helpful. Maybe our friend can wait to leave for the restaurant until we text that we are leaving work? But always communicating every possible mode of failure is not helpful. What is our friend supposed to do because we might be struck by lightning?

Similarly, of course, it is true that any program can fail with a catastrophic error. For example, if we spill soda on our laptop. But that information generally doesn't help our colleagues or users do anything different with our code. Failures indicate the anticipated ways our programs could fail that could potentially be addressed. Everything else is captured as a defect.

4.2 Cause

ZIO formalizes this distinction between failures and defects using a data type called `Cause`. So far, we have said that `ZIO[R, E, A]` is the type of effects that can potentially fail with an `E` or succeed with an `A`. Now we can be more precise, and say that an effect of type `ZIO[R, E, A]` can potentially fail with a `Cause[E]` or succeed with an `A`.

A `Cause[E]` is a sealed trait that has several subtypes that capture all possible failure scenarios for an effect.

For now, the most relevant subtypes are shown in the following snippet:

```
sealed trait Cause[+E]

object Cause {
  final case class Die(t: Throwable) extends Cause[Nothing]
  final case class Fail[+E](e: E) extends Cause[E]
}
```

A `Cause[E]` can either be a `Fail[E]`, containing an error of type `E`, or a `Die`, containing a `Throwable`. `Fail` describes errors and `Die` describes defects.

4.3 Exit

Another data type that is closely related to `Cause` is `Exit`. `Exit` is sealed trait that describes all the different ways that running effects can finish execution. In particular, effects of type `ZIO[R, E, A]` may either succeed with a value of type `A`, or fail with a `Cause[E]`.

```
sealed trait Exit[+E, +A]

object Exit {
```

```
  final case class Success[+A](value: A) extends Exit[Nothing, A]
  final case class Failure[+E](cause: Cause[E]) extends Exit[E, Nothing]
}
```

Once we understand `Cause`, `Exit` is a relatively simple data type. It is equivalent to `Either[Cause[E], A]`, which is the encoding we used in our mental model of `ZIO` in the first chapter with `E` replaced by `Cause[E]` in the `Left` case. Creating a separate data type for `Cause` just allows us to provide useful methods and clarifies what this data type represents in type signatures.

You will most commonly encounter `Exit` when working with some operators that allow you to do something with the result of an effect. We'll see more specific examples of this later, but for now, just be aware that this data type exists and understand that it represents all the ways a running ZIO effect can finish execution.

4.4 Handling Defects

Most error handling operators only deal with errors instead of defects. For example, if we go back to the signature of `foldM` from the previous chapter, we see that it has no case for handling a defect with a type of `Throwable`:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def foldM[R1 <: R, E1, B](
    failure: E => ZIO[R1, E1, B],
    success: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ZIO(r => self.run(r).fold(failure, success).run(r))
}
```

This philosophy is a great default, because in most cases it doesn't make sense to handle defects. Defects represent unanticipated or unrecoverable failures. So most of the time, we don't want to further complicate the signature of our error handling operators by specifying how to handle an unknown failure that we never expected and may not be able to recover from.

However, in some cases, we may want to handle defects in addition to failures. For example, if we are implementing logging for our application, we may want to log defects to preserve information, even though there is nothing we can do to recover from them.

Handling defects can be particularly important at the edges between different layers in our application. Let's consider the preceding example of a report generation application that uses a console application to allow users to customize report settings. At the level of the console application, there is no way to handle an `IOException`, so this would be treated as a defect, and the console application would die with an `IOException`. The report generation application, on the other

hand, can treat this as a failure and handle it by generating reports with the default settings.

ZIO provides a separate family of operators that give you this flexibility.

Conceptually, the most basic of these operators is `sandbox`, which has the following signature:

```
trait ZIO[-R, +E, +A] {
  def sandbox: ZIO[R, Cause[E], A]
}
```

Sandboxing an effect exposes the full cause of a failure in the error channel of our effect. This lets us use all of the normal error handling operators to handle the `Cause`. When we are done, if we want to submerge the full `Cause` into the effect, so we only have to deal with typed errors, we can use the `unsandbox` operation, which is the inverse of `sandbox`.

The type signature of the `unsandbox` operator is as follows:

```
trait ZIO[-R, +E, +A] {
  def unsandbox[E1](implicit ev: E <:< Cause[E1]): ZIO[R, E1, A]
}
```

Together, `sandbox` and `unsandbox` are powerful enough to implement any operators we might want for dealing with the full cause of a failure. However, ZIO also provides a variety of convenience methods, which are designed to make specific use cases easier.

One of the most general operators is `foldCauseM`. This is like `foldM`, which we have discussed previously, but now the error case includes the full cause of the failure:

```
final case class ZIO[-R, +E, +A](run: R => Either[Cause[E], A]) { self =>
  def foldCauseM[R1 <: R, E1, B](
    failure: Cause[E] => ZIO[R1, E1, B],
    success: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ZIO(r => self.run(r).fold(failure, success).run(r))
}
```

Just like `foldM` can be used to implement a variety of error handling operators, `foldCauseM` can be used to implement many more specific operators for dealing with the full cause of a failure.

4.5 Converting Errors to Defects

As we move up from lower-levels of our application to higher-levels of our application, we get closer to business logic, and it becomes clearer which types of failures are recoverable, and which are unrecoverable.

For example, a low-level utility function that reads a file into a string cannot know if failure to read the file is anticipated and can be recovered from.

However, at a higher-level in our application, we may know a file stores some reference data, which we require to enrich some event-oriented data that we are reading from Kafka. At this higher level, we know that if the file is not present, our application was deployed incorrectly, and the most we can do is fail with some descriptive error.

This means that at a low-level, our utility function that reads a file into a string would return something like `ZIO[Any, IOException, String]`. But at some point higher up, we would want to treat this `IOException` as being unrecoverable—a defect in the way our application was deployed.

To translate from errors to defects, we can use a few different functions, the simplest of which is the `orDie` method. The `orDie` method takes an effect that can fail with any subtype of `Throwable`, and returns an effect that will fail with a defect if the original effect fails with an error.

The type signature of `orDie` is shown in the following snippet:

```
sealed trait ZIO[-R, +E, +A] {
  def orDie(implicit ev: E <:< Throwable): ZIO[R, Nothing, A] = ???
}
```

We could use the method as shown in the following example:

```
def readFile(file: String): ZIO[Any, IOException, String] = ???

lazy val result: ZIO[Any, Nothing, String] = readFile("data.txt").orDie
```

In this example, the effect returned by `orDie` will fail with a defect whenever the original effect would fail with an error.

Sometimes, we don't want to convert every error into a defect. We may want to only treat certain classes of failures as defects. In these cases, the method `ZIO#refineWith` is useful, because it allows us to specify a partial function, which can "pick out" the errors that we wish to keep inside the typed error channel.

As an example, let's say our `readFile` utility function returns a `Throwable`, but we wish to treat all errors except `IOException` as defects (for example, we wish to treat a `SecurityException` as a defect, because there is no plausible way we can recover from such an error).

In this case, we could use the `refineWith` method as shown in the following snippet:

```
def readFile(file: String): ZIO[Any, Throwable, String] = ???

def readFile2(file: String): ZIO[Any, IOException, String] =
  readFile(file).refineWith {
```

```
    case e : IOException => e
  }
```

The `refineWith` method also allows us to change the error type, if we wish, although we do not have to do so. Note that when you use `refineWith`, any error type you do not explicitly match for will be converted to a defect.

Although there are other variants, the last variant we will look at in this section is the `refineToOrDie` method, which accepts a single type parameter: the type of error to keep inside the error channel (any other type of error will be converted to a defect).

Here's how we could implement `readFile2` more simply using `refineToOrDie`:

```
def readFile(file: String): ZIO[Any, Throwable, String] = ???

def readFile2(file: String): ZIO[Any, IOException, String] =
  readFile(file).refineToOrDie[IOException]
```

The `refineToOrDie` method is common when there is only a single error type (among potentially many) that we can recover from.

4.6 Multiple Failures

So far we have been implicitly assuming that a computation will only fail with a single failure value. For example, our application might fail with an `IOException` or a `NumberFormatException`, but it's not going to fail with both of the exceptions at the same time.

In simple procedural code, this is a reasonable assumption. We evaluate one statement at a time, and the first time a statement throws an exception, we stop evaluating further statements and propagate that exception up the call stack.

But there are a few ways this assumption breaks down.

First, there may be multiple parts of our program being executed at the same time if our program is concurrent. We will talk about ZIO's support for concurrent programming soon, but for now, a simple example will suffice.

Say that we need to create a table with customer data. Some of the data exists in our West Coast data center and the rest is stored in the East Coast data center. To reduce latency, we send requests to each data center in parallel. But unfortunately, both of our requests fail with errors `e1` and `e2`, respectively.

What error should we return? In the simple model, where computations can only return a single failure, we are forced to throw away either `e1` or `e2`, discarding information that would potentially be important to the caller.

Second, even in purely sequential programs, there are opportunities for multiple failures to arise.

Consider the following code snippet:

```
lazy val example = try {
  throw new Exception("Error using file!")
} finally {
  throw new Exception("Couldn't close file!")
}
```

In this snippet, we are executing code that throws an exception, perhaps working with a file. We are using Scala's `try` / `finally` syntax to make sure a finalizer runs no matter how our code terminates, perhaps to close the file. But the finalizer itself throws an exception itself! In this case, which exception should be propagated to higher levels of the application?

In addition to being a source of puzzlers, especially when there are multiple nested statements like this, we are again inevitably throwing away information.

To deal with situations like this, ZIO includes two other subtypes of `Cause` in addition to the ones we discussed above:

```
sealed trait Cause[+E]

object Cause {
  final case class Die(t: Throwable) extends Cause[Nothing]
  final case class Fail[+E](e: E) extends Cause[E]
  final case class Both[+E](left: Cause[E], right: Cause[E]) extends Cause[E]
  final case class Then[+E](left: Cause[E], right: Cause[E]) extends Cause[E]
}
```

The `Both` data type represents two causes of failure, which occur concurrently. For example, if we were describing the result of the query for customer data from the two data centers, we would use `Cause.Both(e1, e2)`.

The `Then` data type represents two causes of failure, which occur sequentially. For example, if we are working with a file, and failures occur both when using the file and in closing it, we would use `Cause.Then(useFailure, releaseFailure)`.

Notice that the causes within `Both` and `Then` can themselves contain multiple causes, so `Cause` allows us to not just represent *two failures*, but arbitrarily many failures, all while preserving information about the parallel and sequential structure of these failures.

4.7 Other Useful Error Operators

There are a few other useful error operators that you will see in ZIO applications and may find useful.

The first of these operators is the `orElse` operator, which also exists on data types like `Option`. The type signature of this method is shown below:

```
sealed trait ZIO[-R, +E, +A] {
  def orElse[R1 <: R, E2, A1 >: A](that: ZIO[R1, E2, A1]): ZIO[R1, E2, A1] = ???
}
```

The `orElse` operator is a kind of fallback operator: it returns an effect that will try the effect on the left-hand side, and if that fails, it will fallback to the effect on the right-hand side.

An effect `a.orElse(b)` can only fail if `b` can fail, and only in the way that `b` can fail, because of the fallback behavior.

4.8 Combining Effects with Different Errors

Not every effect can fail, and for those effects that can fail, not all of them fail in the same way. As we saw in the first chapter, effects can be combined with various operators like `zip` and `flatMap`, which raises the question, how do errors combine?

As a concrete example, let's say we have the following two error types:

```
final case class ApiError(message: String) extends Exception(message)
final case class DbError(message: String) extends Exception(message)
```

Now, let's say we have two effects, one which describes calling an API, and the other of which describes querying a database:

```
trait Result

lazy val callApi: ZIO[Any, ApiError, String] = ???
lazy val queryDb: ZIO[Any, DbError, Int] = ???
```

As we can see, these two effects fail in different ways. When we combine them, using an operator like `zip`, ZIO chooses the error type to be the most specific type that is a supertype of both `ApiError` and `DbError`:

```
lazy val combine: ZIO[Any, Exception, (String, Int)] = callApi.zip(queryDb)
```

This default, called *supertype composition*, works well for error hierarchies that share common structure. For example, most errors types on the JVM have `Exception` as a parent type, and many have more specific types of exceptions, such as `IOException`, as a parent type.

In some cases, however, our errors share no common structure. In this case, their common supertype will be `Any`, which is not that useful to describe how an effect may fail.

For example, let's take the following two error types and effectful functions:

```
final case class InsufficientPermission(user: String, operation: String)

final case class FileIsLocked(file: String)
```

```
def shareDocument(doc: String): ZIO[Any, InsufficientPermission, Unit] = ???

def moveDocument(doc: String, folder: String): ZIO[Any, FileIsLocked, Unit] = ???
```

If we combine the effects returned by `shareDocument` and `moveDocument` using `zip`, we end up with the following type:

```
lazy val result: ZIO[Any, Any, Unit] = shareDocument("347823").zip(moveDocument("347823", "/
```

When the error or success type is `Any`, it indicates we have no type information about the value of that channel. In this case, we have no information about the error type, because we lost it as the two different error types were composed using supertype composition. We cannot safely do anything with a value of type `Any`, so the most we can say about such an effect is that it can fail for some unknowable reason.

In these cases, we can explicitly change one error type into another error type using the `ZIO#mapError` method. Just like the ordinary `map` method lets us change an effect that succeeds with one type into an effect that succeeds with another type (by transforming from one type to the other), the `mapError` method lets us change an effect that fails with one type into an effect that fails with another type.

In the preceding example, before we zip together the two effects that have different error types, we can first call `mapError` on them to change them to have the same type, which is "big enough" to contain both error types.

In this case, we could map their errors into `Either[InsufficientPermission, FileIsLocked]`, as shown in the following code snippet:

```
scala mdoc:nest  lazy val result2: ZIO[Any, Either[InsufficientPermission,
FileIsLocked], Unit] =     shareDocument("347823").mapError(Left(_)).zip(moveDOcument("34782
"/temp/").mapError(Right(_)))
```

Although `Either` works in this simple example, it's not a very convenient data type if you have many unrelated errors. To deal with many unrelated errors, you'll either want to use Scala 3, which has a new way of composing types (*union types*), use a type-level set (a so-called `HSet`), or ensure your error types share common supertypes.

4.9 Designing Error Models

Because ZIO supports statically-typed errors, it gives you some flexibility in using the error channel. If you are interacting with Java code, you will probably use `Exception` as the error type for the integration code, because most Java code fails with `Exception`. On the other hand, if you are interacting with Scala code, you will probably use `Throwable`, because Scala libraries did not follow the same conventions as Java.

TODO

4.10 Execution Tracing

Diagnosing failures in `Future`-based code is notoriously difficult, which is because the stack traces generated by `Future` code are not very helpful. Rather than showing the surrounding context, they show implementation details deep inside the implementation of `Future`.

To deal with this problem, ZIO includes a feature called *execution tracing*, which provides extremely fine-grained details on the context surrounding failures. Execution tracing is turned on by default, and allows you to more quickly track down the root cause of problems in your code.

Because ZIO's error channel is polymorphic, you can use your own data types there, and ZIO has no way to attach execution tracing to unknown types. As a result, to retrieve the execution trace of an error, you need the full `Cause`. Once you have the cause, you can simply call `prettyPrint` to convert the full cause information, including execution tracing, into a human-readable string.

In the following code snippet, the full cause data is logged to the console in the event of a failure:

```
lazy val effect: ZIO[Any, IOException, String] = ???

effect.tapCause(cause => console.putStrLn(cause.prettyPrint))
```

Note the method `ZIO#tapCause` is used, which allows us to "tap into" the cause of a failure, effectively doing something on the side (like logging) without changing the success or failure of the effect.

4.10.1 Interpreting Error Tracing

TODO

4.11 Dealing With Stacked Errors

Because ZIO's error type is polymorphic, you have the ability to use the error channel to represent multiple levels of failures. This can simplify some otherwise complicated code.

An example is looking up a user profile from a database. Now, let's say the database query could fail, perhaps because the user is not contained inside the database. But let's also say that in our database, profiles are optional. In this case, our lookup function might have the following type signature:

```
trait DatabaseError
trait UserProfile
```

```
def lookupProfile(userId: String): ZIO[Any, DatabaseError, Option[UserProfile]] = ???
```

There's certainly nothing wrong with this type signature, and if we only have one persistence method returning results wrapped in an `Option`, then it may work just fine.

However, if we find ourselves dealing with a lot of success values wrapped in `Option`, then ZIO's error channel provides a better way: we can "unwrap" data types like `Option` (as well as other failure-oriented data types, including `Try`, `Either`, and so on) by shifting some of the failure cases over to the error channel.

In the following snippet, we introduce a new function that does just this:

```
def lookupProfile2(userId: String): ZIO[Any, Option[DatabaseError], UserProfile] =
  lookupProfile.foldM(
    error   => ZIO.fail(Some(error)),
    success => success match {
      case None          => ZIO.fail(None)
      case Some(profile) => ZIO.succeed(profile)
    }
  )
```

The effect returned by this function can fail with `Option[DatabaseError]`. This may seem like a strange error type, but it should make sense if you think about it for a second: if the original effect succeeded with `None`, then the new effect will fail with `None`. But if the original effect failed with some error `e`, then the new effect will fail with `Some(e)`.

The new effect, which shifts the failure case of `Option` over to the error channel, has the same amount of information to the original effect. But the new effect is easier to use, because if we call `flatMap` on it (or use the effect in a *for comprehension*), then we don't have to worry about the user profile not being there. Rather, if the user profile wasn't there, our code that uses it won't be executed, and we can handle the error at a higher level.

This technique works for other data types like `Try` and `Either`, and some of the exercises in this chapter ask you to write helpers for these cases.

For the case of `Option`, these helpers are already baked into ZIO. You can simply call `some` on a ZIO effect to shift the `None` case of an `Option` over into the error channel, and `optional` to shift it back over to the success channel.

For example, we can implement the `lookupProfile2` method using `some` as follows:

```
def lookupProfile3(userId: String): ZIO[Any, Option[DatabaseError], UserProfile] =
  lookupProfile(userId).some
```

By leveraging ZIO's typed error channel, you can handle wrapped error types with ease.

4.12 Leveraging Typed Errors

ZIO's typed error channel is useful primarily because it lets the Scala compiler check our error handling for us. If we need to ensure that some section of our code handles its own errors, we can change the type signature, and let Scala tell us where the errors are introduced. Then we can handle these error one-by-one until the Scala compiler tells us that we have handled all of them.

A secondary benefit to typed errors is that it embeds statically-checked documentation into the code on how effects can fail. So you can look at a function that reads a file, for example, and know that it fails with `IOException`. Or you can look at a function that validates some data, and know that it can fail with `MissingValue` error.

Rather than errors being something you react to when you debug an application, typed errors let you be pro-active in designing which parts of your application should not fail (and thus, must handle their own errors), and which parts of your application can fail (and how they should fail).

Error behavior doesn't have to be something you discover—it can be something you design.

4.13 Conclusion

As we have seen in this chapter, ZIO's error model is quite comprehensive, providing a full solution to both recoverable errors, as well as unrecoverable errors.

ZIO gives us a rich set of powerful operators to recover from and transform errors, and the polymorphic error channel lets us remove the pain from interacting with stacked error types.

All of these methods help us write applications that react and respond to failure appropriately. Meanwhile, when our applications do fail, ZIO's execution tracing and `Cause` structure give us deep insight into the nature of those failures, including multiple failures that arise from concurrency and finalizers.

4.14 Exercises

As an exercise, implement `catchAllCause`, which allows performing a new effect based on the failure of a computation:

1. Using the appropriate effect constructor, fix the following function so that it no longer fails with defects when executed. Make a note of how the inferred return type for the function changes. `scala mdoc:nest def failWithMessage(string: String) = ZIO.succeed(throw new Error(string))`

2. Using the `ZIO#foldCauseM` operator and the `Cause#defects` method, implement the following function. This function should take the effect, inspect defects, and if a suitable defect is found, it should recover from the error with the help of the specified function, which generates a new success value for such a defect. `scala mdoc:nest def recoverFromSomeDefects[R, E, A](zio: ZIO[R, E, A])(f: Throwable => Option[A]): ZIO[R, E, A] = ???`

3. Using the `ZIO#foldCauseM` operator and the `Cause#prettyPrint` method, implement an operator that takes an effect, and returns a new effect that logs any failures of the original effect (including errors and defects), without changing its failure or success value. `scala mdoc:nest def logFailures[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] = ???`

4. Using the `ZIO#foldCauseM` method, which "runs" an effect to an `Exit` value, implement the following function, which will execute the specified effect on any failure at all: `scala mdoc:nest def onAnyFailure[R, E, A](zio: ZIO[R, E, A], handler: ZIO[R, E, Any]): ZIO[R, E, A] = ???`

5. Using the `ZIO#refineOrDie` method, implement the `ioException` function, which refines the error channel to only include the `IOException` error. `scala mdoc:nest def ioException[R, A](zio: ZIO[R, Throwable, A]): ZIO[R, java.io.IOException, A] = ???`

6. Using the `ZIO#refineToOrDie` method, narrow the error type of the following effect to just `NumberFormatException`. `scala mdoc:nest val parseNumber: ZIO[Any, Throwable, Int] = ZIO.effect("foo".toInt)`

7. Using the `ZIO#foldM` method, implement the following two functions, which make working with `Either` values easier, by shifting the unexpected case into the error channel (and reversing this shifting). `scala mdoc:nest def left[R, E, A, B](zio: ZIO[R, E, Either[A, B]]): ZIO[R, Either[E, B], A] = ??? def unleft[R, E, A, B](zio: ZIO[R, Either[E, B], A]): ZIO[R, E, Either[A, B]] = ???`

8. Using the `ZIO#foldM` method, implement the following two functions, which make working with `Either` values easier, by shifting the unexpected case into the error channel (and reversing this shifting). `scala mdoc:nest def right[R, E, A, B](zio: ZIO[R, E, Either[A, B]]): ZIO[R, Either[E, A], B] = ??? def unright[R, E, A, B](zio: ZIO[R, Either[E, A], B]): ZIO[R, E, Either[A, B]] = ???`

9. Using the `ZIO#sandbox` and `ZIO#unsandbox` methods, implement the following function. `scala mdoc:nest def catchAllCause[R, E1, E2, A]( zio: ZIO[R, E1, A], handler: Cause[E1]`

```
=> ZIO[R, E2, A]     ): ZIO[R, E2, A] = ???
```

10. Using the `ZIO#foldCauseM` method, implement the following function. `scala mdoc:nest     def catchAllCause[R, E1, E2, A]( zio: ZIO[R, E1, A],       handler: Cause[E1] => ZIO[R, E2, A]     ): ZIO[R, E2, A] = ???`

Chapter 5

Essentials: Integrating With ZIO

With the materials in the previous chapters you should have the basic building blocks to begin writing your own code in ZIO, wrapping existing side effecting code in effect constructors to make it easier to reason about your code and see how effects can fail.

However, a very common problem that arises when starting to write code with ZIO is how to integrate it with existing code and libraries that don't "speak the language" of ZIO.

We already know how to wrap existing side effecting code, but often these libraries will return new data types or require additional parameters. For example:

1. How do we convert a `CompletionStage` from the `java.util.concurrent` package into a `ZIO` effect?
2. How do we provide instances of type classes for ZIO data types to interface with libraries based on Cats Effect?
3. How do we provide more specialized data types that certain libraries require, such as a `Transactor` for working with Doobie?

In this chapter we will answer these questions, highlighting ZIO's support for working with other frameworks through both the functionality in ZIO itself as well as through various *interop* packages designed to facilitate using ZIO with particular types of code.

Of course, if you can do everything with ZIO and libraries with native ZIO support your life will be easier. But there are not (yet!) ZIO libraries providing every piece of functionality you might need and no matter what you will have to work with mixed code bases as you are migrating to ZIO, so this support for interoperability is an excellent feature to have.

Before diving into how to integrate with specific other types of code, it is helpful to discuss a few principles for integrating ZIO with other types of code that apply more generally.

First, when starting to use ZIO or migrating a code base to ZIO it is often helpful to start with a small, relatively self-contained part of your application.

This part of your application will go from returning existing data types to returning `ZIO` effects. Other parts of your application that call into the part of your application that has been migrated will now need to call `unsafeRun` or one of its variants to convert those `ZIO` effects back into the original data type that the rest of your application is expecting.

Second, get comfortable using `unsafeRun` at the "edges" of the part of your application that has been migrated to ZIO.

In a program written entirely using ZIO, ideally your entire program would be one `ZIO` effect describing all of your program logic, and you would only call `unsafeRun` once at the top of your entire application, often implementing the `App` trait.

```
trait Example extends App {

  def run(args: List[String]): URIO[ZEnv, ExitCode] =
    myProgramLogic.exitCode

  val myProgramLogic: URIO[ZEnv, Unit] =
    ???
}
```

This is the ideal, and `unsafeRun` has the somewhat intimidating sounding name it does to emphasize that unlike everything else in ZIO it actually does something instead of describing doing something. So in a pure ZIO program you really do want to try to avoid calling `unsafeRun` except at the very top of your application.

In contrast, with a mixed code base you are going to have to call `unsafeRun` at every one of the edges between the "island" of code that you have migrated to ZIO and the rest of your code base, and this is fine.

To facilitate this, it is often helpful to create a single `Runtime` at the top of your application that you can use to run all of your effects.

```
import zio._

val runtime: Runtime[ZEnv] =
  Runtime.default
// runtime: Runtime[ZEnv] = zio.Runtime$$anon$3@7ed6bf7f

object MyZIOService {
  def doSomething: UIO[Int] =
```

```
    ???
}

object MyLegacyService {
  def doSomethingElse: UIO[Int] = {
    val something = runtime.unsafeRun(MyZIOService.doSomething)
    ???
  }
}
```

The `Runtime.default` constructor can be used to create a `Runtime` with the default environment that you can use to run effects throughout your program. If you need access to an additional environment to run the effects you are working with you can use the `Runtime.unsafeFromLayer` constructor to create a runtime from a `ZLayer` that produces the services you need.

By adopting this pattern, you can quickly migrate a small part of your code base to ZIO while continuing to have your whole program compile and run correctly. This ability to get everything to work with a mixed environment is incredibly helpful when migrating a large code base to ZIO.

Then over time you can push out the boundaries of your "island" of code that has been migrated to ZIO by picking one additional part of your code base that interfaces with the original part and migrating that to `ZIO`.

Each time you do this you will no longer need to call `unsafeRun` in the part of your application you migrated, because now that part of your code knows how to work with ZIO effects. But now any parts of your code base that call into the newly migrated code will need to call `unsafeRun` just as you did before.

By doing this each time you incrementally push farther and farther back the boundaries of when you need to call `unsafeRun`, until eventually when your entire application has been migrated you will only need to call it a single time at the very top of your application.

You may also never get to this point.

Migrating an application entirely to ZIO can have definite advantages in allowing you to reason about your entire program in the same away and avoiding costs of translating between different representations.

But you may be happy to maintain parts of your code base using another framework, either because you are happy with the way that framework addresses your business problems or because it is just not worth migrating, and that is fine too. In that case you can use the techniques discussed above and the tools in this chapter to continue integrating that code with the rest of your ZIO application indefinitely.

With that introduction out of the way, let's dive into ZIO's support for integrating with various specific types of code.

We will first talk about integrating with various data types in the `java.util.concurrent` package since these come bundled with every Scala project on the JVM and are used in many existing applications. In connection with this we will also address ZIO's support for interoperability with data types from Google's Guava library that are designed to provide richer versions of these abstractions in some cases.

Next we will discuss integrating with existing data types on Javascript such as Javascript's `Promise` data type. This material will only be relevant for applications targeting Scala.js but will be very helpful for developers working in that environment who want to interface with existing concurrency abstractions on that platform.

After this we will spend some time on Cats Effect, another older functional effect system in the Scala language, and see how ZIO's `cats-interop` package provides instances of necessary type classes for using `ZIO` as the concrete effect type with libraries based on Cats Effect. We will also see how the core data type of Cats Effect is not as rich as `ZIO` and how we can use ZIO type aliases to address this.

Finally, we will review some specific libraries in the Scala ecosystem that pose other challenges for interoperability and show how to address those. For example, we will see how to create a database query with the Doobie library from ZIO and how to define a web server using http4s with ZIO.

5.1 Integrating With Java

As Java has developed over various versions the authors of the `java.util.concurrent` package have implemented several different abstractions for describing potentially asynchronous computations. These include the original `Future` interface as well as the `CompletionStage` interface and its concrete implementation the `CompletableFuture`.

All of these represent some version of an asynchronous computation that can be completed by another logical process and allow either waiting for the result or registering a callback to be invoked upon the result becoming available. So ideally we would like to be able to convert each of these to a `ZIO` effect which describes an asynchronous computation, and in fact we can do that.

To convert a `Future` from the `java.util.concurrent` package to a `ZIO` effect we can use the `fromFutureJava` constructor on ZIO.

```
import java.util.concurrent.Future

import zio.blocking._

object ZIO {
  def fromFutureJava[A](thunk: => Future[A]): RIO[Blocking, A] =
```

```
    ???
}
```

There are a couple of things to note about this.

First, the `fromFutureJava` constructor as well as all the other variants we will learn about in this section take a *by name parameter*. This is because a `Future` represents a concurrent program that is already "in flight" whereas a `ZIO` effect represents a blueprint for a concurrent program.

Therefore, to avoid the `Future` being evaluated too soon we need to accept it as a by name parameter so that it is not evaluated until the `ZIO` effect is actually run and can potentially be evaluated multiple times if the effect is run multiple times. As a user, this means you should define the `Future` either inside the argument to the `fromFutureJava` constructor or in a `def` or `lazy val` to prevent it from being evaluated too eagerly.

Second, the `fromFutureJava` constructor returns an effect that depends on the `Blocking` service. This is because there is no way to do something with the result of a `Future` without blocking. Unlike other asynchronous interfaces, a `java.util.concurrent.Future` does not support registering a callback when the `Future` is completed so the only way to get the value out of the `Future` is to block and wait for it.

For this reason, it is not recommended to work with the `Future` interface at all and if you are working with a concrete data type that implements both the `Future` interface and another interface that does support registering a callback, such as `CompletionStage`, you should integrate with ZIO using the constructor designed for that more powerful interface.

However, if you are working purely in terms of the `Future` interface you are probably already aware of these limitations and if that is the situation you are in you are already blocking to get the result of that computation, so the `fromFutureJava` constructor just exposes that in the type signature and makes sure it is run on the right thread pool.

The convert a `CompletionStage` or a `CompletableFuture` into a `ZIO` effect we can use the `fromCompletionStage` constructor.

```
import java.util.concurrent.CompletionStage

object ZIO {
  def fromCompletionStage[A](thunk: => CompletionStage[A]): Task[A] =
    ???
}
```

This is very similar to the `fromFutureJava` constructor except it works with a `CompletionStage` instead of a `Future`. Unlike `fromFutureJava`, this returns a `Task[A]` instead of an `RIO[Blocking, A]` because we can register asynchronous

callbacks with a `CompletionStage` so we don't need to block to convert a `CompletionStage` into a `ZIO`.

Note that because a `CompleteableFuture` is a subtype of a `CompletionStage` there is no separate `fromCompletableFuture` constructor. Instead, you can just use `fromCompletionStage` on any `CompleteableFuture` to convert it to a `ZIO` effect.

There are many other concurrency primitives in the `java.util.concurrent` package but these are the ones that are most similar to `ZIO` in terms of describing asynchronous computations and have direct support for conversion into `ZIO` effects.

For other data types in the `java.util.concurrent` package there are a couple of approaches you can take for integrating.

First, many data types have ZIO equivalents that we will learn about later in this book. For example, ZIO's `Ref` data type is the equivalent of an `AtomicReference` and ZIO has its own `Promise`, `Semaphore`, and `Queue` data types.

Second, you can always continue working with existing data types and simply wrap operators on them in `ZIO` effect constructors.

For example, if for some reason we really wanted to work with an `AtomicReference` instead of ZIO's `Ref` data type we could do so by wrapping the existing operators on `AtomicReference` in the `effectTotal` constructor.

```
import java.util.concurrent.atomic.AtomicReference

import zio._

def setAtomicReference[A](reference: AtomicReference[A])(a: A): UIO[Unit] =
  ZIO.effectTotal(reference.set(a))
```

It is important to use an effect constructor here because the `AtomicReference` represents a *mutable* variable and most other concurrent data types from Java also contain mutable state, so we need to use an effect constructor so we can reason about working with them the same way we do with the rest of our code.

In addition to the `Future` and `CompleteableFuture` interfaces defined in the `java.util.concurrent` package, Google's Guava library contains a `ListenableFuture` abstraction that can also be converted to a `ZIO` value.

In this case, we need to add a new dependency on the `zio-interop-guava` library, since this functionality only makes sense for users who are already working with Guava. You can do that like this:

```
libraryDependencies ++= Seq(
  "dev.zio" %% "zio-interop-guava" % "30.0.0.1"
)
```

With this dependency added, we can then convert a `ListenableFuture` into a `ZIO` value with the `fromListenableFuture` operator.

```
import java.util.concurrent.Executor

import com.google.common.util.concurrent.ListenableFuture

object guava {
  def fromListenableFuture[A](make: Executor => ListenableFuture[A]): Task[A] =
    ???
}
```

This is similar to the other interop operators we have seen in this section in that instead of accepting a `ListenableFuture`, which represents a running computation, it accepts a function that produces a `ListenableFuture`. The difference is that the function also has access to a `java.util.concurrent.Executor` which can be used to create the `ListenableFuture`.

This ensures that the `ZIO` effect created from the `ListenableFuture` will, when evaluated, be run on ZIO's own thread pool.

5.2 Integrating With Javascript

Just like ZIO has tools to integrate with data types from the `java.util.concurrent` package, many of which are specific to the JVM, ZIO also has support for integrating with data types that are specific to Javascript.

These methods allow converting back and forth between a Javascript `Promise` and a `ZIO` effect.

To convert a `ZIO` effect into a `Promise` we can use the `toPromiseJS` and `toPromiseJSWith` operators.

```
type JSPromise = scala.scalajs.js.Promise

trait ZIO[-R, +E, +A] {
  def toPromiseJS(implicit ev: E <:< Throwable): URIO[R, JSPromise[A]]
  def toPromiseJSWith(f: E => Throwable): URIO[R, JSPromise[A]]
}
```

The `toPromiseJS` operator returns a new effect that succeeds with a Javascript `Promise` that will contain the result of the effect. Running the new effect will run the original effect and return a `Promise` that will contain the result of that effect when it is complete.

This operator requires the error type to be a subtype of `Throwable`. If the error type is not a subtype of `Throwable` then you can use the `toPromiseJSWith` variant and provide a function to map your error type to a `Throwable`.

In addition to being able to convert a `ZIO` effect to a Javascript `Promise`, we can also convert a Javascript `Promise` to a `ZIO` effect.

```
object ZIO {
  def fromPromiseJS(promise: => JSPromise[A]): Task[A] =
    ???
}
```

Just like with the constructors to create a `ZIO` effect from a Java `Future`, this constructor takes a Javascript `Promise` as a *by name* parameter because a Javascript `Promise` typically already has associated logic running to complete it. So once again you should either define your Javascript `Promise` within the argument to `fromPromiseJS` or define it elsewhere as a `def` or `lazy val` to prevent it from being evaluated too early.

5.3 Integrating With Cats Effect

The next important category of integration to consider is with other functional effect systems in Scala, in particular the older Cats Effect library. Since Cats Effect existed before ZIO was developed, there are a variety of libraries based on Cats Effect that ZIO users may want to take advantage of so it is important to be able to work with them.

While a full discussion of Cats Effect is outside the scope of this book, this library generally relies on a *tagless final* style of programming where instead of working with a concrete effect type such as `ZIO` directly users work with an abstract effect type that has certain *capabilities* represented by *type classes*.

For example, a simple console program written with `ZIO` might look like this:

```
import scala.io.StdIn

val zioGreet: UIO[Unit] =
  for {
    name <- UIO.effectTotal(StdIn.readLine("What's your name?"))
    _    <- UIO.effectTotal(println(s"Hello, $name!"))
  } yield ()
// zioGreet: UIO[Unit] = zio.ZIO$FlatMap@77b7f7b4
```

In contrast, the same program written with the Cats Effect library would look like this:

```
import cats.effect._
import cats.syntax.all._

def catsGreet[F[_]: Sync]: F[Unit] =
  for {
    name <- Sync[F].delay(StdIn.readLine("What's your name?"))
```

```
    _    <- Sync[F].delay(println(s"Hello, $name!"))
  } yield ()
```

Here the program is parameterized on some *effect type* `F` that has the capabilities described by an abstraction called `Sync` which represents the ability to suspend side effecting code similar to what `ZIO.effect` and `ZIO.effectTotal` can do and the ability to sequentially compose computations, similar to what `ZIO#flatMap` can do.

To actually run this program we need to fix `F` to a specific effect type and have instances in implicit scope for the appropriate type classes that describe the required capabilities. We then use those capabilities to suspend the side effects of reading from and writing to the console and to compose these computations using the `delay` operator on `Sync` and the `flatMap` operator on `F` provided through implicit syntax.

This style of programming has some issues with regard to accessibility. Just in the last few paragraphs we have had to introduce concepts of higher kinded types, type classes and their encoding in Scala, and a particular conceptualization of abstraction over the capabilities of effect types, for example.

It also has some issues with regard to expressive power. We cannot model an effect that cannot fail within the Cats Effect library, for example, since every effect can fail with any `Throwable`, nor can we model the required environment of an effect without introducing additional data types.

However, one advantage of Cats Effect is that it makes it relatively easy for ZIO users to work with libraries written in the Cats Effect style.

For example, say we want to take advantage of the logic in `catsGreet` in our `ZIO` program. Conceptually, we could do this as follows:

```
val zioGreet: Task[Unit] =
  catsGreet[Task]
// error: zioGreet is already defined as value zioGreet
// val zioGreet: Task[Unit] =
//     ^^^^^^^^
// error: Could not find an instance of Sync for zio.Task
//   catsGreet[Task]
//             ^
```

There are a couple of things to note here.

First, we have to use the `Task` type alias here instead of `ZIO` directly. This is because `ZIO` has the wrong number of "holes" in the type signature.

`ZIO` has three type parameters, `R` for the required environment of the effect, `E` for the potential ways the effect can fail, and `A` for the potential ways the effect can succeed. In contrast, the `F[_]` expected by Cats Effect only has one type parameter, indicated by the single underscore, corresponding to the `A` type of `ZIO`.

So to fit the type signature of Cats Effect we have to use one of the `ZIO` type aliases that is less polymorphic and only has a single type parameter.

For the environment type, that means we need to use `Any` to describe an effect that doesn't require any environment to be run. For the error type, that means we need to use `Throwable`, because an effect in Cats Effect can always fail with any `Throwable`.

Putting these requirements together, we get `ZIO[Any, Throwable, A]`, which if we remember our type aliases from earlier in this section is equivalent to `Task[A]`. So we use `Task` instead of `ZIO` when specifying the effect type `F`.

Second, we need to have an instance of `Sync` for ZIO in implicit scope for this program to compile.

This is where the `zio-interop-cats` library comes in. It provides instances of each of the appropriate Cats Effect type classes for ZIO data types, so that code like in the example above works.

To get it, first add a dependency on `zio-interop-cats` as follows:

```
libraryDependencies ++= Seq(
  "dev.zio" %% "zio-interop-cats" % "2.2.0.1"
)
```

Then importing `zio.interop.catz._` should bring all of the requires instances into scope.

```
import zio.interop.catz._

val zioGreet: Task[Unit] =
  catsGreet[Task]
// zioGreet: Task[Unit] = zio.ZIO$FlatMap@3a991c94
```

We have now converted this Cats Effect program into a ZIO program that we can compose with other ZIO effects using all the normal operators we know and run just like any other ZIO effect.

5.4 Integrating With Specific Libraries

In addition to working with Cats Effect in general, there are a couple of specific libraries written in the Cats Effect style that ZIO users frequently need to work with and that can pose special challenges.

The first of these is Doobie.

Doobie provides a purely functional interface for Java Database Connectivity. This allows users to describe interactions with a database separately from actually running them, similar to how ZIO allows users to describe concurrent programs without actually running them.

For example, here is how we might describe a simple SQL query using Doobie.

```
import doobie._
import doobie.implicits._

val query: ConnectionIO[Int] =
  sql"select 42".query[Int].unique
// query: ConnectionIO[Int] = Suspend(
//   BracketCase(
//     Suspend(PrepareStatement("select 42")),
//     doobie.hi.connection$$$Lambda$133638/0x0000000805235040@178420ba,
//     cats.effect.Bracket$$Lambda$133640/0x000000080599f840@18befe89
//   )
// )
```

Here `query` describes a simple SQL SELECT statement that just returns the number `42`.

The return type is `ConnectionIO[Int]`. `ConnectionIO` is a data type specific to Doobie and while its exact implementation is beyond the scope of this book, you can think about it as being similar to `ZIO` in that it *describes* a database transaction that will eventually return an `Int`.

Notice that at this point the description of the query is independent of any particular database being queried. It is also independent of even any particular dialect of SQL, for example MySQL versus PostgreSQL.

To actually run a `ConnectionIO` we need to provide it with a particular database to run the query on, similar to how in ZIO to run a program that required an environment of type `Database` we would need to provide it with a concrete `Database` implementation.

In Doobie, the way we do this is with a data type called a `Transactor`. A `Transactor` knows how to execute a query with respect to a particular database, handling all the bookkeeping of opening database connections, closing them, and so on.

A `Transactor` is parameterized on some concrete effect type `F[_]` similar to the one we talked about when we learned about integrating with Cats Effect above.

```
trait Transactor[F[_]]
```

A `Transactor` knows how to take a *description* of a query of type `ConnectionIO` and turn it into an `F` program that is ready to be run. For example, if `F` was ZIO's `Task` type alias, then a `Transactor[Task]` would be able to turn a `ConnectionIO` description of a query into a ZIO `Task` describing performing that query on a particular database that we could run with `unsafeRun` or combine with other ZIO effects to build a larger program.

The Doobie library will allow us to build a `Transactor` for any effect type as long as there is an instance of the `Async` type class from Cats Effect in implicit

scope for `F`. Fortunately, the `zio-interop-cats` library discussed above already provides just such an instance.

There are a variety of ways of creating a `Transactor` in Doobie and a full discussion of them is beyond the scope of this book, but one common way to create a `Transactor` is with the `fromDriverManager` constructor.

```
val transactor: Transactor[Task] =
  Transactor.fromDriverManager[Task](
     // configuration specific details go here
  )
```

Notice that we typically have to specify the type of the functional effect we are using, in this case `Task`, because `fromDriverManager` could potentially construct multiple `Transactor` instances that could each interpret queries into a different effect system (e.g. Monix versus ZIO) and needs us to tell it which one we want.

With the appropriate `Transactor`, we can then convert a query to an effect we can run using the `transact` operator on the query.

```
val effect: Task[Int] =
  query.transact(transactor)
```

One further complication that can arise is providing any necessary thread pools. Doobie's execution model assumes that two thread pools will be used.

One of these is a *bounded* thread pool that will be used for awaiting connections. The other is a *blocking* thread pool that will be used to perform the actual Java Database Connection operations.

In the simple example above with the `fromDriverManager` constructor Doobie takes care of providing reasonable default implementations of each of these thread pools for us and this is excellent for prototyping applications. However, in production we often want more fine grained control over how our queries are executed and use constructors which require us to provide our own thread pool implementations.

For example, Doobie includes a `doobie-hikari` module that supports a `HikariTransactor` backed by a Hikari connection pool. To create a `HikariTransactor` we need to provide it with an `ExecutionContext` for awaiting requests and a `Blocker`, which is similar to ZIO's `Blocking` service, for performing potentially blocking database transactions.

```
object HikariTransactor {
  def newHikariTransactor[F[_]](
    driverClassName: String,
    url: URL,
    user: String,
    password: String,
    connectionEC: ExecutionContext,
    transactionEC: Blocker
```

```
  ): Resource[F, HikariTransactor[F]] =
    ???
}
```

Note that the returned type is not a `Transactor` but a `Resource[F, Transactor]`. A `Resource` describes a resource that has some necessary *finalization* associated with it, in this case shutting down the connection pool, and is similar to ZIO's `ZManaged` data type that we will learn about later in this book.

The `zio-interop-cats` package provides a `toManagedZIO` operator on any `Resource` that we can use to convert the `Resource` to a `ZManaged`. From there we can call the `use` operator on `ZManaged` to get access to the `HikariTransactor` itself with the guarantee that the connection pool will automatically be closed as soon as the `use` effect finishes executing.

The final step is just to provide the appropriate non-blocking and blocking thread pools.

Conceptually we know how to do this using the operators we learned about earlier in this section to access ZIO's non-blocking and blocking thread pools. However, there is a lot going on here, especially when we're trying to focus on our business logic instead of interoperability, so let's see how all of this works.

```
import cats.effect.Blocker
import doobie.hikari._
import zio.blocking._

lazy val hikariTransactor: ZManaged[Blocking, Throwable, HikariTransactor[Task]] =
  for {
    blockingExecutor <- blockingExecutor.toManaged_
    runtime          <- ZIO.runtime[Any].toManaged_
    transactor       <- HikariTransactor.newHikariTransactor[Task](
                          ???,
                          ???,
                          ???,
                          ???,
                          runtime.platform.executor.asEC,
                          Blocker.liftExecutionContext(blockingExecutor.asEC)
                        ).toManagedZIO
  } yield transactor

lazy val effect: ZIO[Blocking, Throwable, Int] =
  hikariTransactor.use(transactor => query.transact(transactor))
```

As you can see, there is no magic here, but there are a decent number of things we need to get right, which is why we are spending time on this. Let's walk through this one step at a time.

1. On the first line, we are calling the `blockingExecutor` operator on the `Blocking` service to get access to ZIO's blocking `Executor`.
2. On the second line, we are using the `runtime` operator on the `ZIO` companion object to access ZIO's runtime, which will allow us to access the `Executor` for non-blocking tasks.
3. On the third line we are are creating the `Transactor`, passing it both the blocking and non-blocking executors
4. Finally, we `use` the transactor to run the query.

Along the way we are using a couple of helper methods to convert between data types, including the `asEC` operator on `Executor` to be able to view a ZIO `Executor` as a `scala.concurrent.ExecutionContext` and the `toManagedZIO` operator to convert a Cats Effect `Resource` into a `ZManaged`.

The end result is that we are able to use all the features of Doobie from within ZIO, specifying precisely how we want each thread pool to be used. There are many other operators to learn in Doobie but you can use this pattern to convert your database code written with Doobie to work with ZIO.

A second library written in the Cats Effect style that ZIO users sometimes need to work with is http4s. http4s allows defining HTTP clients and servers in a purely functional way so we are just building a description of the client or server until we actually go to run it.

Unfortunately, https4s uses a relatively high amount of advanced category theoretic concepts so it can not be the easiest library to get started with, especially when we also need to interoperate with another effect system. For example, the core data type of a HTTP route is actually:

```
type HttpRoutes[F] = Kleisli[OptionT[F, *], Request, Response]
```

Just like we did with Doobie, we will take things one step at a time.

Similar to a `ConnectionIO` in Doobie, a `HttpRoutes[F]` is a *description* of a HTTP route that can be combined with a concrete server backend, such as a Blaze server, to produce an `F` program that describes running the server. For example, if we have an `HttpRoutes[Task]` then we could run it with a server backend to produce a ZIO `Task` that we could run to launch the server or combine with the rest of our ZIO program.

We can define a simple HTTP route using the `HttpRoutes.of` constructor.

```
import org.http4s._
import org.http4s.dsl.request._

val helloRoute: HttpRoutes[Task] =
  HttpRoutes.of[Task] {
    case GET -> Root / "hello" / name =>
      Response(Status.Ok).withBody(s"Hello, $name from ZIO on a server!")
  }
```

```
// helloRoute: HttpRoutes[Task] = Kleisli(
//   org.http4s.HttpRoutes$$$Lambda$133641/0x0000000805839840@723eaf5c
// )
```

The pattern here is relatively similar to the one we used for working with Database queries.

`HttpRoutes` is expecting an `F[_]` with only a single type parameter so we use the less polymorphic ZIO type signature `Task` to fit the expected type. We need to specify the type `Task` explicitly because http4s can construct routes for a variety of effect types, such as ZIO and Monix, so we need to let the library know what specific effect type we want to use.

The next step is to convert our route into an HTTP application.

```
import org.http4s.implicits._
import org.http4s.server.Router

val httpApp =
  Router("/" -> helloRoute).orNotFound
// httpApp: cats.data.Kleisli[Task, Request[Task], Response[Task]] = Kleisli(
//   org.http4s.syntax.KleisliResponseOps$$Lambda$133647/0x0000000804912840@6a1b3f73
// )
```

This converts our HTTP route into an application that can handle requests that do not match the route as well, in this case just by returning a Not Found status code.

With this done, we are now ready to implement a simple actual server backend so that we can try out our application on our local machine.

```
import org.http4s.server._
import org.http4s.server.blaze._
import zio.interop.catz.implicits._

val server: ZManaged[Any, Throwable, Server[Task]] =
  ZIO.runtime[Any].toManaged_.flatMap { implicit runtime =>
    BlazeServerBuilder[Task](runtime.platform.executor.asEC)
      .bindHttp(8080, "localhost")
      .withHttpApp(httpApp)
      .resource
      .toManagedZIO
  }
// server: ZManaged[Any, Throwable, Server[Task]] = zio.ZManaged$$anon$2@66f33251
```

There are a couple of things to note here.

First, we had to do a couple of things to make sure that appropriate instances were in scope for all the capabilities that `Task` needs to have to be used as the

effect type with http4s. Specifically, we needed to have a Cats Effect `Timer` and a `ConcurrentEffect` instance in scope.

A `Timer` in Cats Effect is similar to the `Clock` service in ZIO and models the capability to access the current time as well as to schedule effects to run after a specified delay.

The `zio-interop-cats` package doesn't provide a `Timer` instance for us automatically when we do `import zio.interop.catz._` because the `Timer` instance is hard coded to using the live clock, which can create confusion if we want to provide an alternative implementation, for example for testing. However, to get the `Timer` instance all we have to do is add an additional import as shown above.

```
import zio.interop.catz.implicits._
```

So remember, if you ever get a warning about the compiler not being able to find an implicit `Timer` when interoperating with a Cats Effect library just add this implicit.

The `ConcurrentEffect` type class in Cats Effect is the second set of capabilities we need. It represents the ability to execute effects concurrently on multiple fibers, for example by racing two effects, returning the first to complete, and canceling the other.

The `zio-interop-cats` package can provide the necessary instance for us, but to do so we have to have an implicit ZIO `Runtime` in scope to actually do the work of forking these concurrent effects. The library can't just provide a `Runtime` to us automatically because different applications may be configured differently, for example with thread pool settings optimized for that application.

The easiest way to bring an implicit ZIO `Runtime` into scope is to use the `ZIO.runtime` operator to access the runtime and then use `flatMap` to make the runtime available to subsequent parts of the program, marking the runtime as `implicit`. You can see that pattern used in the example of building the Blaze server above.

With these two steps of importing the `Timer` instance and making the implicit `Runtime` available the http4s library now has all the necessary capabilities to use `Task` as the concrete effect type for the server.

The second thing to notice is that we are using many of the same tools as we used with Doobie to convert between ZIO and Cats Effect data types.

The Blaze server builder returns a Cats Effect `Resource` because the server has logic that needs to be executed when shutting it down, so we use the `toManagedZIO` operator to convert it to a ZIO `ZManaged` value. The server builder also expects to be provided with an `ExecutionContext` so we access the `Executor` in ZIO's runtime and treat it as an `ExecutionContext` to fulfill the necessary requirement.

Finally we are ready to actually launch our server.

The server we defined above is a `ZManaged` value, indicating that it is a resource that requires some finalization. In many cases we would want to acquire the resource, do something with it, and then release the resource afterwards.

But here we don't want to do anything with the server other than running it indefinitely until we shut it down. `ZManaged` provides a special `useForever` operator to do just that.

```
val useServer: Task[Nothing] =
  server.useForever
// useServer: Task[Nothing] = zio.ZIO$CheckInterrupt@4ace0e8d
```

The server will now run forever until it either terminates with an error or the program is interrupted. To see this for yourself try running `useServer` using either a ZIO `App` or `unsafeRun`, then navigate to `localhost:8080/hello/name` with your name and see the response you get!

5.5 Conclusion

This chapter has provided an overview of general principles for working with mixed code bases, which can occur either because you are in the process of migrating an application to ZIO or because you are using ZIO for some components of your application and another framework for other components.

In addition, we have looked in significant detail at how we can operate with several commonly used frameworks, including existing concurrency abstractions on the JVM and Javascript as well as the older Cats Effect functional library for Scala and several specific libraries written in terms of it.

While there are many other frameworks for dealing with particular domains, this should give you a good overview of the core concepts that you can apply to interoperating with any other framework you come across. In addition, we will be showing how to tackle several specific domains, for example using Kafka with ZIO Stream, later in this book in the appendices where we develop solutions to specific problem areas.

5.6 Exercises

Chapter 6

Parallelism And Concurrency: The Fiber Model

This chapter begins our discussion of ZIO's support for asynchronous, parallel, and concurrent programming. ZIO is based on a model of **fibers** so we will begin by learning what fibers are and how they are different than threads. We will learn the basic operations on fibers including forking them, joining them, and interrupting them. We will also discuss ZIO's fiber supervision model and how it makes it easier for us to write safe concurrent code.

6.1 Fibers Distinguished From Operating System Threads

Fibers are lightweight equivalents of operating system threads. Like a thread, a fiber models a running computation and instructions on a single fiber are executed sequentially. However, fibers are much less costly to create than operating system threads, so we can have hundreds of thousands of fibers in our program at any given time whereas maintaining this number of threads would have a severe performance impact. Unlike threads, fibers are also safely interruptible and can be joined without blocking.

To see how fibers work, we need to understand a little more about the ZIO runtime and how it executes our programs.

When we call `unsafeRun` the ZIO runtime creates a new fiber to execute our program. At this point our program hasn't started yet, but the `Fiber` represents a running program that we happen to not have started yet but will be running at

some point. You can think of it as a `Thread` we have created to run our program logic but have not started yet.

The ZIO runtime will then submit that `Fiber` for execution on some `Executor`. An `Executor` is typically backed by a thread pool in multithreaded environments. The `Executor` will then begin executing the `ZIO` program one instruction at a time on an underlying operating system thread, assuming one is available.

Because a `ZIO` effect is just a "blueprint" for doing something, executing the ZIO program just means executing each instruction of the ZIO program one at a time. However, the ZIO program will not necessarily run to completion. Instead after a certain number of instructions have been executed, the `maximumYieldOpCount`, the ZIO program will suspend execution and any remaining program logic will be submitted to the `Executor`.

This ensure fairness. Say we have five fibers. The first four fibers are performing long running computations with one million instructions each. The fifth fiber is performing a short computation with only one thousand instructions. If the `Executor` was backed by a fixed thread pool with four threads and we did not yield we would not get to start running the fifth fiber until all the other fibers had completed execution, resulting in a situation where the work of the fifth fiber was not really performed concurrently with the other fibers, leading to potentially unexpected and undesirable results.

If instead each fiber yields back to the runtime after every thousand instructions then the first four fibers do a small amount of work, then the fifth fiber gets to do its work, and then the other four fibers get to continue their work. This creates the result of all five fibers running concurrently even though only four are ever actually running at any given moment.

In essence, fibers represent units of "work" and the `Executor` switches between running each fiber for a brief period, just like the operating system switches between running each `Thread` for a small slice of time. The difference is that fibers are just data types we create that do not have the same operating system overhead as threads, and since we create the runtime we can build in logic for how we ensure fairness, how we handle interruption, and so on.

6.2 Forking Fibers

The first fundamental operation of fibers is forking them.

```
import zio._

trait ZIO[-R, +E, +A] {
  def fork: URIO[R, Fiber[E, A]]
}
```

Forking creates a new fiber that executes the effect being forked concurrently

with the current fiber.

For example, in the following code `doSomething` is guaranteed to complete execution before `doSomethingElse` begins execution because both effects are being performed on the same fiber:

```
lazy val doSomething: UIO[Unit] = ???
lazy val doSomethingElse: UIO[Unit] = ???

lazy val example1 = for {
  _ <- doSomething
  _ <- doSomethingElse
} yield ()
```

On the other hand, if we `fork` `doSomething` then the order of execution is no longer guaranteed:

```
lazy val example2 = for {
  _ <- doSomething.fork
  _ <- doSomethingElse
} yield ()
```

Here there is no guarantee about the order of execution of `doSomething` and `doSomethingElse`. `fork` does not even wait for the forked fiber to begin execution before returning, so `doSomething` and `doSomethingElse` could begin and complete execution in any order.

6.3 Joining Fibers

The second fundamental operation of fibers is joining fibers.

```
trait Fiber[+E, +A] {
  def join: IO[E, A]
}
```

While `fork` executes a computation concurrently with the current one, `join` waits for the result of a computation being performed concurrently and makes it available to the current computation.

An important characteristic of `join` is that it does not block any underlying operating system threads. When we `join` a fiber, execution in the current fiber can't continue until the joined fiber completes execution. But no actual thread will be blocked waiting for that to happen. Instead, internally the ZIO runtime registers a callback to be invoked when the forked fiber completes execution and then the current fiber suspends execution. That way the `Executor` can go on to executing other fibers and doesn't waste any resources waiting for the result to be available.

Joining a fiber translates the result of that fiber back to the current fiber, so

joining a fiber that has failed will result in a failure. If we instead want to wait for the fiber but be able to handle its result whether it is a success or a failure we can use `await`:

```
trait Fiber[+E, +A] {
  def await: UIO[Exit[E, A]]
}
```

We can also tentatively observe the state of the fiber without waiting for it using `poll`:

```
trait Fiber[+E, +A] {
  def poll: UIO[Option[Exit[E, A]]]
}
```

This will return `Some` with an `Exit` value if the fiber has completed execution, or `None` otherwise.

6.4 Interrupting Fibers

The final fundamental operation on fibers is interrupting them.

```
trait Fiber[+E, +A] {
  def interrupt: UIO[Exit[E, A]]
}
```

Interrupting a fiber says that we do not need this fiber to do its work anymore and it can immediately stop execution without returning a result. If the fiber has already completed execution by the time it is interrupted the returned value will be the result of the fiber. Otherwise it will be a failure with `Cause.Interrupt`.

Interruption is critical for safe resource usage because often we will start doing some work that may ultimately not be needed. For example, a user navigates to a web page and we launch a bunch of requests to gather information to display but then the user closes the browser. These kinds of scenarios can easily lead to a resource leak over time if we are repeatedly doing unnecessary work during the life of a long running application.

ZIO has several features that make interruption much more useful than it is in thread based models.

First, interruption is safe in ZIO whereas it is not in thread based concurrency models.

We can interrupt a `Thread` by calling `Thread.interrupt` but this is a very "hard" way to stop a thread's execution. When we interrupt a thread we have no guarantee that any finalizers associated with logic currently being executed by that thread will be run, so we could leave the system in an inconsistent state where we have opened a file but not closed it, or we have debited one bank account without crediting another.

In contrast, interrupting a fiber in ZIO causes any finalizers associated with that fiber to be run. We will talk more about finalizers in the section on `bracket` and resource usage, but ZIO provides a way to attach finalizers to an effect with a guarantee that if that effect begins execution the finalizers will always be run, whether the effect succeeds with a value, fails with an error, or is interrupted.

Second, interruption in ZIO is much more efficient than interruption in thread based models because fibers themselves are so much more lightweight.

A thread is quite expensive to create, relatively speaking, so even if we can interrupt a thread we have to be quite careful in doing so because we are destroying this valuable resource. In contrast fibers are very cheap to create, so we can create many fibers to do our work and interrupt them when they aren't needed anymore.

Interruption just tells the ZIO runtime that there is no more work to do on this fiber. The operating system threads backing the `Executor` continue running the other active fibers without any changes.

In addition to its support for interruption, one of the features ZIO provides is the ability to turn interruption on and off for certain sections of code. We will get into interruption much more soon. Conceptually, it is important to be able to mark certain sections of code as uninterruptible so once we start doing something we know we can finish doing that thing. Preventing interruption is also important for finalizers, since we want the finalizers to run if we are interrupted and don't want the execution of the finalizers to itself be interrupted.

Interruptibility can be controlled with the `interruptible` and `uninterruptible` combinators.

6.5 Fiber Supervision

So far we have been focused primarily on forking and joining single effects. But what happens if we have multiple layers of forked effects? For example, consider this program:

```
import zio._
import zio.clock._
import zio.console._
import zio.duration._

object ZIOExample {

  val child: ZIO[Clock with Console, Nothing, Unit] =
    ZIO.sleep(5.seconds) *> putStrLn("Hello from a child fiber!")

  val parent: ZIO[Clock with Console, Nothing, Unit] =
    child.fork *> ZIO.sleep(3.seconds) *> putStrLn("Hello from a parent fiber!")
```

```
  val example: ZIO[Clock with Console, Nothing, Unit] =
    for {
      fiber <- parent.fork
      _     <- ZIO.sleep(1.second)
      _     <- fiber.interrupt
      _     <- ZIO.sleep(10.seconds)
    } yield ()
}
```

In the main fiber we fork `parent` and `parent` in turn forks `child`. Then we interrupt `parent` when `child` is still doing work. What should happen to `child` here?

Based on everything we have discussed so far, `child` would keep running in the background. We have interrupted `parent`, not `child`, so `child` would continue execution.

But that seems like a less than ideal outcome.

`child` was forked as part of `parent` , presumably to do some computation that is part of `parent` returning its result. In this example the return type is just `Unit` because this is illustrative, but we could imagine `parent` computing the digits of pi and `child` doing part of that work. If we no longer need the digits of pi we no longer need to do the work to compute them.

It also seems to expose an internal implementation detail of `parent`. We would like to be able to refactor code to perform work in parallel by using fibers, for example replacing a sequential algorithm with a parallel one, and have that refactoring preserve the meaning of our code (other than hopefully improving the efficiency). But we see here that forking the work of `child`, rather than performing it in `parent` directly, has caused an observable change in the meaning of our code. If we had done the work directly interrupting `parent` would have interrupted everything, but because we forked `child` part of the work is still being done despite interruption.

In this simple example the issue is easy to observe, but if this were a more complex combinator, potentially from another library, it could be very difficult for us to know whether this effect was forking other fibers internally.

To address this, ZIO implements a **fiber supervision model**. You can think of this as somewhat akin to how actors might supervise their children in an actor model. The rules are as follows:

1. Every fiber has a **scope**
2. Every fiber is forked in a scope
3. Fibers are forked in the scope of the current fiber unless otherwise specified
4. The scope of a fiber is closed when the fiber terminates, either through success, failure, or interruption
5. When a scope is closed all fibers forked in that scope are interrupted

The implication of this is that by default fibers can't outlive the fiber that forked them. So in the example above `child` would be forked in the scope of `parent`. When `parent` was interrupted its scope would be closed and it would interrupt `child`.

This is generally a very good default. If you do need to create a fiber that outlives its parent (e.g. to create some background process) you can fork a fiber on the **global** scope using `forkDaemon`.

```
trait ZIO[-R, +E, +A] {
  def forkDaemon: URIO[R, Fiber[E, A]]
}
```

The global scope is never closed, so a fiber forked using `forkDaemon` will never be interrupted because its scope has been closed and will run until it terminates or is explicitly interrupted, potentially forever.

We will see more combinators to work with scopes and achieve more fine grained control over fiber supervision in a couple of chapters, but for now a good rule of thumb is that if you find your effects are being terminated too early, replace `fork` with `forkDaemon`

Another good rule of thumb is to always `join` or `interrupt` any fiber you `fork`. The fiber supervision model can be thought of as a set of rules for how to deal with fibers when the user doesn't explicitly handle them. The easiest thing you can do to not have to worry about this is to handle your fibers yourself. Think of fibers as a resource you are creating and handle their acquisition and release.

6.6 Locking Effects

Normally, fibers will be executed by ZIO's default `Executor`. However, sometimes we may want to execute some or all of an effect with a particular `Executor`. We already saw an example of this with our discussion of the `Blocking` service. Specifying the `Executor` can also be important if we are working with a library that requires work to be performed on a thread pool provided by that library or if we just want to run some workloads on an `Executor` specialized for those workloads as an optimization.

ZIO makes it easy to do this with the `lock` and `on` combinators.

```
import scala.concurrent.ExecutionContext

import zio.internal.Executor

trait ZIO[-R, +E, +A] {
  def lock(executor: Executor): ZIO[R, E, A]
  def on(executionContext: ExecutionContext): ZIO[R, E, A]
}
```

`lock` is in some sense the more fundamental combinator. It takes the effect that it is called on and guarantees that it will be run on the specified `Executor`. This guarantee will hold even if the effect involves asynchronous steps. If the effect forks other fibers, those fibers will also be locked to the specified `Executor` unless otherwise specified.

One of the important characteristics of `lock` that you will find with other concepts such as interruptibility is that it is **regional**. This means that:

1. When an effect is locked to an `Executor` all parts of that effect will be locked to that `Executor`
2. Inner scopes take precedence over outer scopes

To illustrate the first rule, if we have:

```
lazy val doSomething: UIO[Unit] = ???
lazy val doSomethingElse: UIO[Unit] = ???

lazy val executor: Executor = ???

lazy val effect = for {
  _ <- doSomething.fork
  _ <- doSomethingElse
} yield ()

lazy val result = effect2.lock(executor)
```

Both `doSomething` and `doSomethingElse` are guaranteed to be executed on `Executor`.

To illustrate the second rule, if we change the example to:

```
lazy val executor1: Executor = ???
lazy val executor2: Executor = ???

lazy val effect2 = for {
  _ <- doSomething.lock(executor2).fork
  _ <- doSomethingElse
} yield ()

lazy val result2 = effect2.lock(executor1)
```

Now `doSomething` is guaranteed to be executed on `executor2` and `doSomethingElse` is guaranteed to be executed on `executor1`.

Together, these rules make controlling where an effect is executed compositional and easy to reason about.

`on` is like `lock` except it takes an `ExecutionContext`. Since `Executor` is a ZIO data type whereas `ExecutionContext` is one from the Scala standard library, this variant can be more convenient when interfacing with external libraries

requiring that certain blocks of code be executed on a thread pool provided by that library.

6.7 Conclusion

In this chapter we have gone into a lot of detail on how ZIO actually implements concurrency. In the process we have seen how fibers work and touched on concepts like interruption and supervision that we will come back to in more detail in subsequent chapters.

While helpful to understand, especially if you want to implement your own concurrency combinators, ZIO is written so that in most cases you shouldn't have to deal with fibers. Fibers are the low level building blocks that are used to implement higher level concurrency combinators. In most cases you should just be able to use the higher level combinators, which will automatically "do the right thing" with regard to interruption, the lifetime of fibers, and so on. However, it is good to have some understanding of how these combinators work if you run into issues or need to do something yourself that isn't covered by the existing combinators.

The next chapter will go through ZIO's built in parallelism and concurrency combinators in detail.

6.8 Exercises

Chapter 7

Parallelism And Concurrency: Concurrency Operators

This chapter goes through the key parallelism and concurrency combinators defined on `ZIO`. Unlike the last chapter, which focused a bit more on the "how", this chapter is squarely focused on the "what". In this chapter you will learn the key concurrency combinators you can use to solve the vast majority of the problems you face day to day in writing parallel and concurrent code.

7.1 The Importance Of Concurrency Operators

You should use these combinators whenever possible because as much as we try to make it easy, concurrency is hard, and fibers are relatively low level. By using the combinators in this chapter you will be taking advantage of the work and experience of all the ZIO contributors and users before you and avoiding any potential pitfalls. Use fibers directly when you have to for implementing new combinators, but a good sign in higher level code is if you are not using fibers directly at all.

7.2 Race And ZipPar

The two most basic concurrency and parallelism combinators defined on `ZIO` are `raceEither` and `zipPar`:

```
trait ZIO[-R, +E, +A] { self =>
  def raceEither[R1 <: R, E1 >: E, B](that: ZIO[R1, E1, B]): ZIO[R1, E1, Either[A, B]]
```

```
  def zipPar[R1 <: R, E1 >: E, B](that: ZIO[R1, E1, B]): ZIO[R1, E1, (A, B)]
}
```

Both of these describe binary operations that combine two `ZIO` values. They also describe the two ways we can combine two `ZIO` with different return types using some degree of parallelism or concurrency.

First, we can combine the `A` and `B` values to return both of them as we do in `zipPar`. This describes running the `self` and `that` effects in parallel, waiting for both of them to complete, and returning their result.

Second, we can combine the `A` and `B` values to return either one or the other as we do in `raceEither`. This describes running the `self` and `that` effects concurrently, returning the first to complete.

Though they seem simple these two combinators actually allow us to solve a great many problems. `zipPar` lets us run two effects in parallel, but we can call `zipPar` multiple times to run arbitrarily many effects in parallel and so can implement a great many parallelism combinators with `zipPar`. Similarly, we can call `raceEither` multiple times to return the first of arbitrary many effects to complete, or race an effect against an effect that sleeps a specified duration to implement a timeout.

Let's look at each of these combinators in more detail.

`zipPar` takes two effects and returns a new effect that describes running the two effects in parallel and returning both of their results when available. What happens if one effect or the other fails? Because `zipPar` cannot return successfully unless both effects complete successfully, there is no point in running one effect once the other fails. So as soon as one effect fails `zipPar` will immediately interrupt the other computation. The error returned will include both causes if both effects failed.

`raceEither` runs two effects and returns a new effect that describes running the two effects concurrently and returning the first result. Since only one result is needed, as soon as one effect completes successfully the other will be interrupted. If the first effect to complete fails with an error `raceEither` will wait for the second effect to complete and return its value if it is a success, or else fail with a cause containing both errors.

7.3 Variants of ZipPar

`zipPar` can be used to implement a variety of other parallelism combinators. The mostly commonly used are the `foreachPar` and `collectAllPar` variants:

```
object ZIO {
  def collectAllPar[R, E, A](in: Iterable[ZIO[R, E, A]]): ZIO[R, E, List[A]] = ???
  def foreachPar[R, E, A, B](in: Iterable[A])(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] = ??
}
```

The signatures of these are the same as the `collectAll` and `foreach` variants we discussed previously except this time they will perform all the effects in parallel instead of sequentially.

`collectAllPar` and `foreachPar` have optimized implementations for performance, but conceptually they can both be implemented in terms of `zipPar` and this will give you an excellent understanding of the semantics of these combinators.

Just like `zipPar`, `collectAllPar` and `foreachPar` need to return a collection that is the same size as the original collection, which means they can only return successfully after every effect has completed successfully. Thus, if any effect fails, these combinators will immediately interrupt all the other effects and fail with a cause containing all the errors.

There are variants of each of these combinators with an `_` suffix for when the return value is not needed:

```
object ZIO {
  def collectAllPar_[R, E, A](in: Iterable[ZIO[R, E, A]]): ZIO[R, E, Unit] = ???
  def foreachPar_[R, E, A, B](in: Iterable[A])(f: A => ZIO[R, E, B]): ZIO[R, E, Unit] = ???
}
```

The implementation of these variants can be optimized because it does not need to create a data structure so use these variants if you don't need the return value but otherwise the semantics are the same.

In addition, there are variants that allow performing parallel computations with bounded parallelism.

```
object ZIO {
  def collectAllParN[R, E, A](n: Int)(in: Iterable[ZIO[R, E, A]]): ZIO[R, E, List[A]] = ???
  def foreachParN[R, E, A, B](n: Int)(in: Iterable[A])(f: A => ZIO[R, E, B]): ZIO[R, E, List
}
```

These variants have an `N` suffix and take an additional parameter for the maximum degree of parallelism. This is the maximum number of effects in the collection that will be performed in parallel. This can be useful when we want to perform an effect for each element in a large collection in parallel but don't necessarily want to perform all the effects at once. For example, if we had ten thousand requests we needed to perform we might overwhelm our backend if we submitted all of those at the same time. We could instead specify a lower level of parallelism so we could have, say, a dozen outstanding requests at any given time.

Conceptually you can think of this as using a `Semaphore` to limit the degree of parallelism though as with the other variants beyond the limited parallelism the other semantics are exactly the same. There are also `collectAllParN_` and `foreachParN_` variants of these if you do not need the return value.

7.4 Variants of Race

7.5 Other Variants

Another useful parallelism combinator in some situations is `validatePar`. As we discussed above, `foreachPar` can only succeed if the effectual function succeeds for each element in the collection and so immediately interrupts the other effects on the first failure. In general this is an excellent behavior because if we can't succeed there is no point in doing additional work.

However, sometimes we are interested in getting all of the failures. For example, a user might enter their first name, last name, and age into a web form and we execute three separate effects in parallel to validate each input against applicable business logic.

With `foreachPar` if the validation of the first name fails we would immediately interrupt the validation of the last name and the age, so we would likely fail with a cause that contains the first name validation error and `Cause.Interrupted` for the other two effects. However, we would really like to see if there are any validation errors for the last name or age so we can present all the validation errors to the user at once and provide a better customer experience versus the user having to repeatedly get different error messages.

We can do that through the `validatePar` combinator:

```
object ZIO {
  def validatePar[R, E, A, B](
    in: Iterable[A]
  )(f: A => ZIO[R, E, B]): ZIO[R, ::[E], List[B]] = ???
}
```

`validatePar` takes exactly the same arguments as `foreachPar` but returns an effect with a slightly different error type and has different semantics. Unlike `foreachPar`, `validatePar` will not interrupt the other effects on the first failure but will instead allow all the effects to complete execution. If all of the effects complete successfully it will return a collection of all of their results, just like `foreachPar`. But if any effect fails it will fail with a collection containing all the failures.

So in our example above, if validation failed for the first name, last name, and age, `validatePar` would fail with a collection containing all three errors, making it easy for us to see everything that went wrong and deal with it appropriately.

Note the use of the `::` type here. `::` is the `Cons` case of `List` from the Scala standard library.

```
trait List[+A]

final case class ::[+A](head: A, tail: List[A]) extends List[A]
```

```
case object Nil extends List[Nothing]
```

Because `::` is guaranteed not to be empty it allows us to express additional information in the type signature beyond what we could with `List`. For example, in `validatePar` we know that if a failure has occurred there must be at least one error, so the collection of errors can never be empty. This is similar to the `NonEmptyChunk` data type in ZIO or the `NonEmptyList` data type in ZIO Prelude but allows us to express this with no additional dependencies and in a way that interoperates seamlessly with the Scala standard library.

7.6 Conclusion

7.7 Exercises

Chapter 8

Parallelism And Concurrency: Fiber Supervision In Depth

We learned in our discussion of ZIO's fiber based concurrency model that ZIO supports a concept of *fiber supervision*. This means that when a fiber completes execution it will, by default, interrupt its children.

This approach prevents unnecessary work because when a fiber is interrupted it automatically interrupts any fibers it has forked as part of computing its result. It also promotes resource safety because it avoids fibers outliving the effects they were associated with, which can cause memory leaks.

8.1 Preventing Fibers From Being Interrupted Too Early

For the reasons above, by default ZIO interrupts fibers as soon as the fiber that forked them has completed execution. This is usually the right answer but sometimes it can result in fibers being interrupted too early.

For example, consider the following:

```
import zio._
val grandChild: UIO[Unit] =
  ZIO.effectTotal(println("Hello, World!"))
// grandChild: UIO[Unit] = zio.ZIO$EffectTotal@51896bff

val child: UIO[Fiber[Nothing, Unit]] =
  grandChild.fork
```

```
// child: UIO[Fiber[Nothing, Unit]] = zio.ZIO$Fork@2a1ea749

child.fork *> ZIO.never
// res0: ZIO[Any, Nothing, Nothing] = zio.ZIO$FlatMap@22a6543e
```

In this example, "Hello, World!" will not always be printed to the console.

The only thing the `child` fiber is doing is forking another effect, `grandChild`. So `child` will complete execution almost immediately, possibly before `grandChild` has even started execution.

As soon as `child` completes execution, it will by default interrupt all of its children, including `grandChild`. So if `child` completes execution and interrupts `grandChild` before `grandChild` begins execution, "Hello, World!" will never be printed to the console.

One way to solve this problem is to always either `join` or `interrupt` any fiber we `fork`.

```
val grandChild: UIO[Unit] =
  ZIO.effectTotal(println("Hello, World!"))
// grandChild: UIO[Unit] = zio.ZIO$EffectTotal@6218883

val child: UIO[Unit] =
  grandChild.fork.flatMap(_.join)
// child: UIO[Unit] = zio.ZIO$FlatMap@31789141

child.fork *> ZIO.never
// res1: ZIO[Any, Nothing, Nothing] = zio.ZIO$FlatMap@30865796
```

Now `child` will not complete execution until `grandChild` completes execution and `join` returns. So there is no risk that `child` will interrupt `grandChild` before it has printed "Hello, World" to the console.

For example, we may want to fork a background effect to send a heartbeat signal to some service to indicate that we are still interested in consuming values from it.

```
import zio.clock._
import zio.duration._

val effect: ZIO[Clock, Nothing, Int] =
  for {
    _ <- ZIO.effectTotal(println("heartbeat")).delay(1.second).forever.fork
    _ <- ZIO.effectTotal(println("Doing some expensive work"))
  } yield 42
// effect: ZIO[Clock, Nothing, Int] = zio.ZIO$FlatMap@6ab148a3

for {
  fiber <- effect.fork
```

```
  _      <- ZIO.effectTotal(println("Doing some other work")).delay(5.seconds)
  result <- fiber.join
} yield result
// res2: ZIO[Clock, Nothing, Int] = zio.ZIO$FlatMap@26f6847e
```

The `effect` fiber will complete execution almost immediately because all it is doing is forking the background heartbeat fiber and printing "Doing some expensive work". As soon as it finishes execution it will interrupt the background fiber, which is not what we want.

And we can't just interrupt or join the background fiber because we want to continue sending the heartbeat signal and we want it to continue occurring in the background as we proceed with the rest of our program.

The quick fix for solving these problems of fibers being interrupted too early we presented previously was replacing `fork` with `forkDaemon`.

```
val effect: ZIO[Clock, Nothing, Int] = for {
  _ <- ZIO.effectTotal(println("heartbeat")).delay(1.second).forever.forkDaemon
  _ <- ZIO.effectTotal(println("Doing some expensive work"))
} yield 42
// effect: ZIO[Clock, Nothing, Int] = zio.ZIO$FlatMap@346fc49d

for {
  fiber <- effect.fork
  _      <- ZIO.effectTotal("Doing some other work").delay(5.seconds)
  result <- fiber.join
} yield result
// res3: ZIO[Clock, Nothing, Int] = zio.ZIO$FlatMap@49c19265
```

With the `forkDaemon` operator, the background heartbeat fiber will now be forked in the global scope instead of the scope of the `effect` fiber, so it will not be interrupted when the `effect` fiber completes execution but instead will continue until it is explicitly interrupted, potentially forever.

This does solve the immediate problem, but in a way it is too blunt of a solution. Now the background fiber will continue sending the heartbeat signal forever, but what if the heartbeat signal only needs to be sent while a certain part of the application is processing, not for the entire application?

```
val effect: ZIO[Clock, Nothing, Int] = for {
  _ <- ZIO.effectTotal(println("heartbeat")).delay(1.second).forever.forkDaemon
  _ <- ZIO.effectTotal(println("Doing some expensive work"))
} yield 42
// effect: ZIO[Clock, Nothing, Int] = zio.ZIO$FlatMap@6b7e0ec4

val module = for {
  fiber <- effect.fork
  _      <- ZIO.effectTotal("Doing some other work").delay(5.seconds)
```

```
  result <- fiber.join
} yield result
// module: ZIO[Clock, Nothing, Int] = zio.ZIO$FlatMap@636f2bba

val program = for {
  fiber <- module.fork
  _     <- ZIO.effectTotal("Running another module entirely").delay(10.seconds)
  _     <- fiber.join
} yield ()
// program: ZIO[Clock, Nothing, Unit] = zio.ZIO$FlatMap@65b71d7c
```

We would like for the background heartbeat fiber to be automatically interrupted when the `module` fiber completes execution, but not before. How do we achieve this?

Recall from our rules for fiber supervision that every fiber is forked in a scope, by default the scope of the parent fiber. And when a fiber terminates it interrupts all fibers forked in its scope.

So what we are really asking is how do we fork a fiber in a different scope? We know how to fork a fiber in the default scope of the current fiber using `fork` and we know how to fork it in the global scope using `forkDaemon`, but how do we fork it in some other scope?

8.2 Scopes

The answer is the `forkIn` operator, which allows us to fork an effect in any scope we specify:

```
trait ZIO[-R, +E, +A] {
  def forkIn(scope: ZScope[Exit[Any, Any]]): URIO[R, Fiber.Runtime[E, A]]
}
```

This is the first time we have seen the `ZScope` data type. So far we have been able to talk about scopes conceptually, but there is actually nothing special about scopes, they are a data type like any other.

Let's take this opportunity to better understand the `ZScope` data type since it will help us with the rest of our discussion of fiber supervision.

The simplified interface of `ZScope` looks like this:

```
sealed abstract class ZScope[+A] {

  // Whether the scope has been closed
  def closed: UIO[Boolean]

  // Add a finalizer that will be run when the scope is closed
  def ensure(finalizer: A => UIO[Any]): UIO[Either[A, ZScope.Key]]
```

```
  // Delay running finalizers until that scope has been closed
  def extend(that: ZScope[Any]): UIO[Boolean]

  // Whether finalizers associated with this scope have been run
  def released: UIO[Boolean]
}

object ZScope {

  // Make a new scope
  def make[A]: UIO[Open[A]] =
    ???

  // A scope along with a way of closing the scope
  final case class Open[A](close: A => UIO[Boolean], scope: Local[A])

  // The global scope
  object global extends ZScope[Nothing] {
    def closed: UIO[Boolean] =
      ???
    def ensure(
      finalizer: Nothing => UIO[Any]
    ): UIO[Either[Nothing, ZScope.Key]] =
      ???
    def extend(that: ZScope[Any]): UIO[Boolean] =
      ???
    def released: UIO[Boolean] =
      ???
  }

  // A local scope
  final class Local[A] extends ZScope[A] {
    def closed: UIO[Boolean] = ???
    def ensure(finalizer: A => UIO[Any]): UIO[Either[A, ZScope.Key]] = ???
    def extend(that: ZScope[Any]): UIO[Boolean] = ???
    def released: UIO[Boolean] = ???
  }

  // A key associated with a finalizer
  final class Key
}
```

Conceptually a `ZScope[A]` represents a scope that will eventually be closed with a value of type `A`.

Finalizers can be added to the `ZScope` with the `ensuring` operator and they

will be run when the scope is closed.

If the scope is still open when `ensuring` is called a `Key` will be returned that can be used to look up the finalizer. If the scope was already closed `ensuring` will return the value the scope was closed with so the caller can decide what to do, for example running the finalizer immediately.

We can tell whether a scope is open or not with the `open` operator.

We can also `extend` a scope so that its finalizers are not run until another scope has been closed.

There is also a `released` operator that tells us whether finalizers associated with a scope have been run. Note that this could be different from whether the scope is open because the scope could have been extended so it is closed but its finalizers have not been run yet.

One thing you might notice is not part of the interface of `ZScope` itself is the ability to close a scope.

When a new scope is created with the `make` method the caller gets access to a `close` action that they can use to close the scope, but the scope itself does not come with the capability to close it. This gives the ability to pass around scopes without fear that one user of a scope will close the scope and "pull the rug out" from other users of the same scope.

Now that we understand more about what a scope is, how do we work with scopes in connection with effects and fibers?

The two most basic operators are `ZIO.scopeWith` and `ZIO#forkIn`.

The `scopeWith` operator allows us to access the scope of the current fiber and do something with it, such as adding a finalizer to it, extending it, or passing it to another effect. For example, here is how we could add another finalizer to a scope:

```
for {
  ref   <- Ref.make(false)
  fiber <- ZIO.scopeWith(scope => scope.ensure(_ => ref.set(true))).fork
  _     <- fiber.await
  value <- ref.get
} yield value
// res4: zio.ZIO[Any, Nothing, Boolean] = zio.ZIO$FlatMap@1835f50
```

The `scopeWith` operator accesses the current scope of the fiber and adds a new finalizer that will set the `Ref` to `true`. We then `await` the fiber, which will suspend until the fiber has completed execution and any finalization logic associated with it has completed.

Since the only thing the forked fiber is doing is adding the finalizer to the scope the forked fiber will complete execution immediately after that and then the

scope will be closed, causing the finalizer to be run and the `Ref` to be set to `true`.

Normally we don't add finalizers directly to scopes and let the ZIO runtime take care of that for us, preferring operators like `ensuring` on ZIO, but it is helpful to see that there is no magic there.

The other basic operator to work with scopes and effects is `forkIn`.

So far we have `fork`, which forks an effect in the scope of the current fiber, and `forkDaemon`, which forks an effect in the global scope. `forkIn` lets us fork an effect in any scope we want.

With `forkIn` we can even make our own scope and fork a fiber in that:

```
import zio._

for {
  ref     <- Ref.make(false)
  promise <- Promise.make[Nothing, Unit]
  open    <- ZScope.make[Exit[Any, Any]]
  effect   = (promise.succeed(()) *> ZIO.never).ensuring(ref.set(true))
  _       <- effect.forkIn(open.scope)
  _       <- promise.await
  _       <- open.close(Exit.unit)
  value   <- ref.get
} yield value
// res6: ZIO[Any, Nothing, Boolean] = zio.ZIO$FlatMap@2c06b1d6
```

Here we fork `effect` in the scope we create. Since we created the scope, we also have the ability to close it, and we see that when the scope that effect is forked in is closed `effect` itself is interrupted, causing its finalizer to run and set the `Ref` to `true`.

Again, working directly with scopes like this is not normally the way we want to program with ZIO but is helpful for getting familiar with the concept of scopes and how they interact with the ZIO runtime.

Using `scopeWith` and `forkIn`, we actually have the tools we need to solve the problem above regarding getting the right scope for the heartbeat fiber.

```
import zio.clock._
import zio.duration._

def effect(scope: ZScope[Exit[Any, Any]]): ZIO[Clock, Nothing, Int] = for {
  _ <- ZIO.effectTotal(println("heartbeat")).delay(1.second).forever.forkIn(scope)
  _ <- ZIO.effectTotal(println("Doing some expensive work"))
} yield 42

val module = for {
```

```
  fiber <- ZIO.forkScopeWith(scope => effect(scope).fork)
  _     <- ZIO.effectTotal("Doing some other work").delay(5.seconds)
  result <- fiber.join
} yield result
// module: ZIO[Clock, Nothing, Int] = zio.ZIO$FlatMap@60ed5dad

val program = for {
  fiber <- module.fork
  _     <- ZIO.effectTotal("Running another module entirely").delay(10.seconds)
  _     <- fiber.join
} yield ()
// program: ZIO[Clock, Nothing, Unit] = zio.ZIO$FlatMap@425686ad
```

The program is almost identical to the previous version except now in `module`, the scope that we want the heartbeat signal to persist in, we access the current scope using `forkScopeWith` and pass the scope to `effect`, which now takes the scope as a parameter and forks the heartbeat signal in the scope.

This does exactly what we want it to and for application code this can be exactly what we need.

However, there is something that is not entirely satisfactory about this. Our ability to do the right thing here depended on us understanding the implementation of `module`, and in fact being able to change the implementation to use `forkIn` and take a `ZScope` that we provided to it.

But what if we are writing an operator that takes an effect that the caller provides? In this case, we may not even know whether the effect the caller provides forks fibers at all and we definitely do not have a way to modify the implementation.

For example, consider this simple implementation of the `zipWithPar` operator:

```
def zipWithPar[R, E, A, B, C](
  left: ZIO[R, E, A], right: ZIO[R, E, B]
)(f: (A, B) => C): ZIO[R, E, C] =
  for {
    leftFiber  <- left.fork
    rightFiber <- right.fork
    a          <- leftFiber.join
    b          <- rightFiber.join
  } yield f(a, b)
```

This simple implementation gets some things right, for example if the effect is interrupted `leftFiber` and `rightFiber` will be interrupted due to fiber supervision. It also has some problems, in particular if `rightFiber` fails `leftFiber` will not be immediately interrupted.

But let's put that aside for the moment and focus on the problem of this implementation of `zipWithPar` behaves when `left` or `right` fork fibers that

they don't join or interrupt.

```
val heartBeat: ZIO[Clock, Nothing, Fiber[Nothing, Unit]] =
  ZIO.effectTotal(println("heartbeat")).delay(1.second).forever.fork
// heartBeat: ZIO[Clock, Nothing, Fiber[Nothing, Unit]] = zio.ZIO$Fork@4afa064

val somethingElse: UIO[Unit] =
  ZIO.effectTotal(println("doing something else"))
// somethingElse: UIO[Unit] = zio.ZIO$EffectTotal@19a6077b

val effect: ZIO[Clock, Nothing, Fiber[Nothing, Unit]] =
  zipWithPar(heartBeat, somethingElse)((fiber, _) => fiber)
// effect: ZIO[Clock, Nothing, Fiber[Nothing, Unit]] = zio.ZIO$FlatMap@6c60d08d
```

We are going to have the same problem as we saw at the beginning of the chapter. `zipWithPar` forks `heartBeat` into a new fiber as part of running effects in parallel. But the only thing that `heartbeat` does is itself fork a fiber to send the heartbeat message.

As soon as `heartbeat` completes execution it will interrupt the fibers that it has forked by default. But that isn't the behavior we want!

`heartBeat` would normally just return the fiber sending the heartbeat signal, allowing the signal to continue until explicitly terminated by interrupting the fiber. But now `heartBeat` will terminate almost immediately just because we composed it with another effect with this version of `zipWithPar`.

What solutions do we have as the implementors of `zipWithPar` to solve this problem with the tools we have discussed in this chapter so far?

We could replace `fork` in `zipWithPar` with `forkDaemon`. This would prevent the heartbeat fiber from being interrupted too early, but it would also mean that the `left` and `right` effects would no longer be interrupted if the `zipWithPar` effect was interrupted, so that does not seem like a very good solution.

We could try to use `scopeWith` and `forkIn` but we quickly run into a problem.

Conceptually we want to make sure the fiber that is actually sending the heartbeat signal is forked in the scope of the main `zipWithPar` fiber. But we don't have access to the code that forks that fiber from `zipWithPar`.

From the perspective of `zipWithPar`, `left` and `right` are just arbitrary effects that may or may not fork fibers. We have no way of "looking into" the internal implementation of the `left` and `right` effects and rewriting calls to `fork` to `forkIn` with a scope we specify.

We want to be able to say that any fibers forked by `leftFiber` should be forked in a certain scope, but we don't have the power to do that yet. Right now we can say that a certain fiber should be forked in a specified scope, but not that a fiber should fork all of its children in a specified scope.

8.3 Overriding The Scope A Fiber Forks New Fibers In

The `forkScopeOverride` is one additional piece of information that every fiber maintains. It is an *optional* field that, when set, indicates that children of a fiber should be forked in the specified scope instead of in the fibers own scope.

One important property of the `forkScope` is that it is *not* inherited by forked fibers.

For example, if we set fiber A's `forkScopeOverride` to scope `X` and then fiber A forks fiber B, fiber B will be forked in scope `X`. But if Fiber B then forks fiber C, fiber C will be forked in Fiber B's own scope.

This is important because when we set the `forkScopeOverride` we want to make sure a fiber under our control forks its children in a specified scope, but those children may fork their own fibers and we want to preserve the normal supervision relationship within that effect, which may be outside of our control.

Just like with the scope of a fiber, there are combinators for getting and setting the `overrideForkScope`:

```
trait ZIO[-R, +E, +A] {
  def overrideForkScope(scope: ZScope[Exit[Any, Any]]): ZIO[R, E, A] =
    ???
  def resetForkScope: ZIO[R, E, A] =
    ???
}

object ZIO {
  def forkScope: UIO[ZScope[Exit[Any, Any]]] =
    ???
}
```

`overrideForkScope` sets the `overrideForkScope` to the specified scope.

`resetForkScope` removes any `overrideForkScope` currently set for this fiber, so effects forked by this fiber will be forked in the fiber's own scope.

`forkScope` retrieves the scope that fibers will be forked in. If an `overrideForkScope` has been set that scope will be returned. Otherwise the fibers own scope will be returned.

Let's see how to use these operators to control what scope a fiber will fork effects in.

```
import zio._

for {
  promise <- Promise.make[Nothing, Unit]
```

```
  ref     <- Ref.make(false)
  open    <- ZScope.make[Exit[Any, Any]]
  effect1 = (promise.succeed(()) *> ZIO.never).ensuring(ref.set(true))
  effect2 <- effect1.overrideForkScope(open.scope)
  fiber   <- (promise.succeed(()) *> ZIO.never).ensuring(ref.set(true)).fork
  _       <- promise.await
  _       <- open.close(Exit.unit)
  value   <- ref.get
} yield value
// res8: ZIO[Any, Nothing, Boolean] = zio.ZIO$FlatMap@5d59f551
```

Notice how this time we just used `fork` instead of `forkIn`. Normally `fork` would just fork a new fiber in the scope of the current fiber. But because we used `forkScopeOverride`, `fiber` will instead be forked in the scope we specified there.

There are also a couple of other methods built on top of these operators. `forkScopeMask` is like `uninterruptibleMask` and `interruptibleMask` in that it sets the `forkScopeOverride` for an effect but also gives a `restore` function that can be used to restore the original `forkScopeOverride` for part of the effect.

But the most useful operator working with `forkScopeOverride` is `transplant`. The signature of `transplant` is:

```
def transplant[R, E, A](f: ZIO.Grafter => ZIO[R, E, A]): ZIO[R, E, A] =
  ???
```

The `transplant` operator first obtains the scope in which the current fiber forks effects. It then gives us this `grafter`, which lets us take an effect and return a new version with its `forkScopeOverride` set to the specified scope.

This allows us to "skip over" one or more levels of fibers when forking fibers and is the solution we need to easily solve problems like the `zipWithPar` one we saw above. Using `transplant` we can do something like:

```
def zipWithPar[R, E, A, B, C](
  left: ZIO[R, E, A], right: ZIO[R, E, B]
)(f: (A, B) => C): ZIO[R, E, C] =
  ZIO.transplant { graft =>
    for {
      leftFiber  <- graft(left).fork
      rightFiber <- graft(right).fork
      a          <- leftFiber.join
      b          <- rightFiber.join
    } yield f(a, b)
  }
```

Now when `left` and `right` fork effects those fibers will be forked in the scope of `zipWithPar` instead of in `leftFiber` or `rightFiber`, so they will not be

terminated early just because `leftFiber` or `rightFiber` complete execution.

Using `transplant` we "skipped over" `leftFiber` and `rightFiber` and made any effects forked by `left` or `right` supervised by the `zipWithPar` fiber. The fact that we are using `leftFiber` and `rightFiber` is really an implementation detail so this ensures that the details of how we implement parallelism in `zipWithPar` doesn't impact the semantics of user code.

8.4 Conclusion

This chapter has gone into detail on the mechanics of the fiber supervision model, some of the challenges that can arise when working directly with fibers, and the operators for dealing with them. Some of these are perhaps the most advanced operators in ZIO for working directly with fibers.

These are powerful tools when you need them, but in most circumstances you will not need to resort to them. Here is a suggested "order of operations" for working with fibers and dealing with problems of fibers being terminated "too early".

1. Use existing higher level operators

Fibers are the building blocks of ZIO's approach to concurrency, but fibers are very low level. As much as ZIO tries to simplify it concurrency can be hard and these operators are battle tested to do the right thing with regard to interruption, fiber supervision, and so on.

2. Join or interrupt the fibers you fork

In the vast majority of cases fibers should not outlive the effect they are forked as part of. A very useful pattern is to think fibers as resources that we allocate by forking as part of performing some effect and clean up by joining or interrupting when we are done with that effect.

For example, we might fork a number of fibers as part of computing the digits of pi to some precision. Once we have computed the result with the appropriate precision we don't need the fibers anymore and should make sure they are complete or interrupt them ourselves.

Many times this will happen automatically as part of the flow of our program or other ZIO operators, for example interrupting a queue will interrupt all fibers offering or awaiting the queue.

Fiber supervision will usually have our back even if we forget to do this and interrupt these fibers. But awaiting or interrupting fibers we fork ourselves is a good way of making sure we release these resources as quickly as possible and avoiding any uncertainty about the lifetime of these fibers.

3. If you really do want a fiber to outlive the life of an effect, consider just returning the fiber as part of the result type of the effect.

For example, if you want the fiber sending the heartbeat signal to continue after the effect that created it consider just return the fiber as the result of the effect. Then you can either interrupt that fiber at the appropriate point in your code when it is no longer needed as discussed above, or pass to it a higher level of the application, if necessary.

4. If you are using forking fibers internally that execute arbitrary effects, use `transplant` to "skip" your internal fibers in fiber supervision

This is the pattern we saw above with `zipWithPar`. Using `transplant` in your operator and then calling `graft` on the effects provided by the caller of your operator removes your internal fibers from the chain of supervision, ensuring that the semantics of user effects are not impacted by the implementation of your operator.

Make sure to take any appropriate actions based on the exit values of internal fibers you fork, since those results will no longer be propagated to the forked fibers through automatic supervision

With these guidelines you should be in a good position to use the tools of the fiber supervision framework when necessary so that your application always does the right thing.

8.5 Exercises

Chapter 9

Parallelism And Concurrency: Interruption In Depth

We learned about interruption before when we discussed the fiber model. But how exactly does interruption work?

When can an effect be interrupted? How do we achieve fine grained control of interruption so we can make certain portions of our code interruptible and prevent interruption in other parts?

What happens if interrupting an effect itself takes a long time to complete? How do we control whether we wait for interruption to complete or not?

9.1 Timing Of Interruption

To understand interruption more, we first need to understand a little more about how the ZIO runtime system works and how it executes effects.

We said before that `ZIO` effects are "blueprints" for concurrent programs.

This is true in the sense that `ZIO` effects *describe* rather than *do* things so we can run the same effect multiple times. But it is also true in the sense that a `ZIO` effect is literally a list of instructions.

For example, consider the following program:

```
import scala.io.StdIn

import zio._
```

```
val greet: UIO[Unit] =
  for {
    name <- ZIO.effectTotal(StdIn.readLine("What's your name?"))
    _    <- ZIO.effectTotal(println(s"Hello, $name!"))
  } yield ()
// greet: UIO[Unit] = zio.ZIO$FlatMap@6b01819b
```

This translates into a "blueprint" that looks like this:

1. Execute this arbitrary blob of Scala code that happens to read a line from the console.
2. Then take the result of that and use it to execute this other arbitrary blob of Scala code that happens to print a greeting to the console.

There are a couple of things to note here.

First, every ZIO effect translates into one or more statements in this plan. Each statement describes one effect and operators like `flatMap` connect each of these statements togethers.

Second, constructors that import arbitrary code into a `ZIO` execute that code as a single statement.

When we import Scala code into a `ZIO` using a constructor like `effectTotal` the ZIO runtime just sees a single block of Scala code, essentially a thunk `() => A`.

The ZIO runtime has no way to know whether that Scala code is performing a simple statement, like reading a single line from the console, or a much more complex program like reading 10,000 lines from a file. As far as the ZIO runtime is concerned it is a single function `() => A` and the ZIO runtime can either run it or do nothing with it.

This is important because the ZIO runtime checks for interruption before executing each instruction. So in the example above, the runtime would check for interruption once before even starting to execute the code and a second time after getting the user's name before printing it to the console.

9.1.1 Interruption Before An Effect Begins Execution

One important implication of this is that an effect can be "pre-interrupted" before it even begins execution.

For example, in the program below "Hello, World!" may never be printed to the console because the fiber may be interrupted before it begins execution.

```
for {
  fiber <- ZIO.effectTotal(println("Hello, World!")).fork
  _     <- fiber.interrupt
} yield ()
```

This is very logical in that if we know we will not need the result of an effect there is no point in even starting to run the effect.

However, we have to be careful of cases where we are inaccurately assuming that an effect will at least begin execution. For example, consider the following program:

```
for {
  ref   <- Ref.make(false)
  fiber <- ZIO.never.ensuring(ref.set(true)).fork
  _     <- fiber.interrupt
  value <- ref.get
} yield value
```

We might be tempted to think that this program will always return `true` because `ensuring` should guarantee that the `Ref` is set to `true` before we get it. However, this is not actually the case.

The guarantee of `ensuring` is that *if* an effect begins execution then the specified finalizer is guaranteed to be run regardless of how the effect completes execution, whether by success, failure, or interruption.

But we said before that interruption is checked before each effect is executed. So if `fiber.interrupt` executes before `ZIO.never` begins execution then the finalizer in `ensuring` will never be run and the `Ref` will never be set to `true`.

We can guarantee that the finalizer runs in this case by using something like a `Promise` to ensure that the effect has begun execution before it is interrupted.

```
for {
  ref     <- Ref.make(false)
  promise <- Promise.make[Nothing, Unit]
  fiber   <- (promise.succeed(()) *> ZIO.never).ensuring(ref.set(true)).fork
  _       <- promise.await
  _       <- fiber.interrupt
  value   <- ref.get
} yield value
```

Now we complete a `Promise` within the forked fiber and await that `Promise` before interrupting, so we have a guarantee that the forked effect has begun execution before it is interrupted and thus that the finalizer in `ensuring` will be run.

Being able to interrupt effects before they run is a capability that we want and normally if an effect doesn't even begin executing we don't want to run finalizers associated with it. But sometimes this possibility of "pre-interruption" is something you need to watch out for when writing more advanced concurrency operators so it is good to be aware of it.

9.1.2 Interruption Of Side Effecting Code

Another implication is that in the absence of special logic the ZIO runtime does not know how to interrupt single blocks of side effecting code imported into ZIO.

For example, the effect below prints every number between `1` and `100000` to the console.

```
val effect: UIO[Unit] =
  UIO.effectTotal {
    var i = 0
    while (i < 100000) {
      println(i)
      i += 1
    }
  }

for {
  fiber <- effect.fork
  _     <- fiber.interrupt
} yield ()
```

Here `effect` will not be interruptible during execution. Interruption happens only "between" statements in our program and here `effect` is a single statement because it is just wrapping one block of Scala code.

So it is possible in the example above that we will interrupt effect before it begins execution. But if we do not then it will always print all of the numbers between `0` and `100000`, never stopping part way through.

Normally this is not a problem because in most cases ZIO programs consist of large numbers of smaller statements glued together with operators like `flatMap`, so normally there are plenty of opportunities for interruption.

But it is another good watch out and a reminder that well written ZIO programs are typically structured as a large number of simple effects composed together to build a solution to a larger problem, rather than one "monolithic" effect. Otherwise in the extreme we could end up with our whole application being a single imperative program wrapped in a ZIO effect constructor, which would defeat much of the point of using ZIO in the first place!

Sometimes we do need to be able to interrupt an effect like this. For example, we may really want to perform an operation a hundred thousand times in a tight loop in a single effect for efficiency.

In this case we can use specialized constructors like `effectBlockingCancelable`, `effectBlockingInterrupt`, and `effectAsyncInterrupt` to provide our own logic for how ZIO should interrupt the code we are importing.

Let's see how we could use `effectBlockingCancelable` to support safe cance-

lation of a loop like the one above.

The signature of `effectBlockingCancelable` is:

```
def effectBlockingCancelable(effect: => A)(cancel: UIO[Unit]): RIO[Blocking, A] =
  ???
```

`effectBlockingCancelable` essentially says we are importing an effect into ZIO that may take a long time to complete execution and we know the ZIO runtime doesn't know how to interrupt arbitrary Scala code, so we are going to tell it how by providing a `cancel` action to run when we interrupt the fiber that is executing effect.

Here is what it looks like in action:

```
import java.util.concurrent.atomic.AtomicBoolean

import zio.blocking._

def effect(canceled: AtomicBoolean): RIO[Blocking, Unit] =
  blocking.effectBlockingCancelable(
    {
      var i = 0
      while (i < 100000 && !canceled.get()) {
        println(i)
        i += 1
      }
    },
    UIO.effectTotal(canceled.set(true))
  )

for {
  ref   <- ZIO.effectTotal(new AtomicBoolean(false))
  fiber <- effect(ref).fork
  _     <- fiber.interrupt
} yield ()
```

Now `effect` can be safely interrupted even while it is executing by setting the `AtomicBoolean` to true and breaking out of the `while` loop.

`effectBlockingInterrupt` is similar to this except it interrupts the effect by actually interrupting the underlying thread. Interrupting threads is a relatively heavy handed solution so this should only be used when necessary and only with a thread pool that will create new threads as needed to replace ones that are interrupted like ZIO's blocking thread pool.

`effectAsyncInterrupt` is like `effectBlockingInterrupt` but allows us to import an asynchronous effect with an effect that will cancel it. With this, for example, we can request data by providing a callback to be invoked when the data is available but also provide an effect that will signal to the data source that

we no longer need that information and it can cancel any ongoing computations and does not need to invoke the callback.

Again most ZIO programs are composed of small effects imported into ZIO composed together to build solutions to larger problems so in most cases interruption "just works" but this is the chapter on interruption in depth so we are going over more sophisticated use cases like this and how to handle them.

9.2 Interruptible and Uninterruptible Regions

Now that we understand when the ZIO runtime checks for interruption, we are in a better position to discuss how to control whether certain parts of our effect are interruptible or not.

Any part of a `ZIO` effect is either *interruptible* or *uninterruptible*. By default all ZIO effects are interruptible but some operators may make parts of a `ZIO` effect uninterruptible.

The basic operators for controlling interruptibility are `interruptible` and `uninterruptible`.

```
val uninterruptible: UIO[Unit] =
  UIO(println("Doing something really important")).uninterruptible

val interruptible: UIO[Unit] =
  UIO(println("Feel free to interrupt me if you want")).interruptible
```

The `interruptible` in the second statement is technically redundant because effects are interruptible by default but is shown for illustrative purposes.

You can think of interruptibility as a "flag" that is tracked by the ZIO runtime. As the runtime is executing statements before each statement the runtime checks whether the effect has been interrupted and also whether it is interruptible.

If before executing a statement the runtime finds that an effect has been interrupted and is interruptible it stops executing new instructions.

If the effect has been interrupted and is not interruptible it continues executing instructions as normal, checking each time whether the interruptibility status has changed. If it ever finds that the effect is interruptible it stops executing further instructions as described above.

We can think of `interruptible` and `uninterruptible` then as just special instructions that change the interruptibility status to `true` or `false`, respectively, for the duration of the effect they are called on.

One implication of this is that an effect can be interrupted before the interruptibility status is set to `uninterruptible`. For example, consider this program:

```
for {
  ref   <- Ref.make(false)
  fiber <- ref.set(true).uninterruptible.fork
  _     <- fiber.interrupt
  value <- ref.get
} yield value
```

We might think that `value` will always be `true` because we marked `ref.set` as uninterruptible.

But remember that `uninterruptible` is just another instruction. Translating the "blueprint" of the forked effect into a plan it looks roughly like this:

1. Set the interruptibility status to uninterruptible
2. Set the `Ref` to `true`
3. Set the interruptibility status back to its previous value

So applying the same logic we talked about above that interruption is checked before evaluating each instruction, the runtime will check for interruption before evaluating the first instruction and if the effect has been interrupted will stop executing further instructions at this point.

Keep in mind that this is an issue of what is guaranteed to occur. In the example above often the `Ref` will be set to `true` because the forked fiber will have begun executing and set the interruption status to uninterruptible before the interruption is executed.

But that is not guaranteed to happen and if we run this as a test using something like the `nonFlaky` combinator from ZIO Test we will see this does not always occur. Flakiness like this can be the source of subtle bugs so when working with interruption it is critical to write code in a way that it always works instead of works "most of the time".

So how do we ensure that the `Ref` is set to `true` in the example above?

The answer is to move the `uninterruptible` outside of the `fork` like this:

```
for {
  ref   <- Ref.make(false)
  fiber <- ref.set(true).fork.uninterruptible
  _     <- fiber.interrupt
  value <- ref.get
} yield value
```

This works because when a fiber is forked it inherits the interruptibility status of its parent at the time that it is forked. Since the parent fiber was uninterruptible at the time of the fork due to the `uninterruptible` operator, the fiber is forked with an initial status of being uninterruptible, so now the `Ref` is guaranteed to always be set to `true`.

This reflects an important concept that we saw before for `lock` when we discussed

the fiber model. Settings such as what `Executor` an effect is running on or whether an effect is interruptible are *regional*.

This means:

- Combinators that change these setting apply to the entire scope they are invoked on
- Inner scopes override outer scopes
- Forked fibers inherit the settings of their parent fiber at the time they are forked

These rules help us reason compositionally about settings like interruptibility and also help us apply the same mode of reasoning to different settings.

Let's see how each of these rules work.

The first rule says that combinators that change interruptibility apply to the entire scope they are called on.

For example, if we have `(zio1 *> zio2 *> zio3).uninterruptible`, all three effects will be uninterruptible.

The second rule says that inner scopes take precedence over outer scopes. For this, consider the following example:

```
(zio1 *> zio2.interruptible *> zio3).uninterruptible
```

Here `zio2` is marked as interruptible, but the three effects together are marked as uninterruptible. So which should take precedence?

The answer is that inner scopes take precedence over outer ones. So `zio2` would be interruptible.

This gives us tremendous ability to compose effects together with fine grained control of interruptibility. It also mirrors the way scoping works in general, with inner scopes taking precedence over outer scopes.

The final rule says that forked fibers inherit the settings of their parent fiber at the time of forking by default. So for example if we are in an uninterruptible region and fork a fiber that fiber will also be uninterruptible.

This reflects the concept of *fork join identity* discussed in the chapter on fiber supervision and helps us refactor our code without accidentally creating bugs.

For instance, say we are working with part of our program that needs to be performed without interruption. To optimize this part of our code, we decide to fork two parts and run them in parallel instead of performing them sequentially.

Without fibers inheriting the status of their parent this refactoring could result in the forked fibers now being interruptible, creating a serious bug in our program! By having fibers inherit the status of their parents we facilitate treating forking or not as more of an implementation detail, supporting fearless refactoring.

9.3 Composing Interruptibility

So far we know how to mark certain regions of our code as interruptible or uninterruptible and how to combine these to create any possible combination of interruptible and uninterruptible regions in our code.

For simple applications `interruptible` and `uninterruptible` may be all that we need. We know how each part of our code will be called and we know that it should either be interruptible or uninterruptible.

But things can get more complex when dealing with interruption in library code or as part of larger applications.

We may not know where our code will be called from, and in particular it could be called within a region that is either interruptible or uninterruptible. How do we make sure our code works the right way in both cases so that both our code does what we expect it to do and the caller's code does what the caller expects it to do?

We may also be writing operators for `ZIO` effects? How do we make sections of code that need interruptible or uninterruptible without changing the status of the user's effect, since they may be relying on it being interruptible or uninterruptible?

The first guideline for solving this problem is moving from `uninterruptible` and `interruptible` to `uninterruptibleMask` and `interruptibleMask`.

The signature of `uninterruptibleMask` is as follows:

```
def uninterruptibleMask[R, E, A](
  k: ZIO.InterruptStatusRestore => ZIO[R, E, A]
): ZIO[R, E, A]
```

The signature may look somewhat intimidating, but `InterruptStatusRestore` is just a function that takes any effect and returns a version of that effect with the interruptibility status set to the previous value.

For example, if we needed to do some work before and after executing an effect provided by the caller and wanted to make sure we were not interrupted during that work we could do:

```
def myOperator[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A]
  ZIO.uninterruptibleMask { restore =>
    UIO(println("Do some important work that shouldn't be interrupted")) *>
    restore(zio) *>
    UIO(println("Do some other important work that also shouldn't be interrupted"))
  }
```

The `restore` function here will restore the interruptibility status of `zio` to whatever it was at the point that `uninterruptibleMask` was called.

So if `uninterruptibleMask` was called in an interruptible region `restore` would make `zio` interruptible again. If it was called in an uninterruptible region then `zio` would still be uninterruptible.

The `uninterruptibleMask` operator is often preferable to `uninterruptible` because it is in some sense less intrusive. If there is some part of our code that needs to not be interrupted it lets us do that but doesn't change anything about the effect provided by the caller.

If we instead used `uninterruptible` on our portions of the effect and `interruptible` on the effect provided by the caller we might be inadvertently changing the effect provided by the user and turning an effect that is supposed to be uninterruptible into one that is interruptible.

The second guideline is to be very careful about using `interruptible` or `interruptibleMask`.

Generally `uninterruptibleMask` gives us the tools we need to solve the vast majority of problems we face involving interruption.

ZIO effects are interruptible by default, so typically we only need `interruptible` to "undo" making parts of our effects uninterruptible when we need to make sure we are not interrupted. But we already saw above that `uninterruptibleMask` is a better tool to do that.

The problem with `interruptible` is that it can "punch holes" in otherwise uninterruptible regions.

For example, consider the following `delay` combinator:

```
import zio._
import zio.duration._

def delay[R, E, A](zio: ZIO[R, E, A])(duration: Duration): ZIO[R with Clock, E, A] =
  clock.sleep(duration).interruptible *> zio

// Don't do this!
```

This simply delays for the specified duration before executing the effect. We use the `sleep` method on `clock` to do the delay and with the best of intentions we mark the `sleep` effect as interruptible because if we are interrupted there is no use in continuing to wait to wake up from the `sleep` call.

But what happens if we compose this effect into a region that is supposed to be uninterruptible?

```
for {
  ref       <- ref.make(false)
  promise   <- Promise.make[Nothing, Unit]
  effect    =  promise.succeed(()) *> ZIO.never
  finalizer =  ref.set(true).delay(1.second)
```

```
  fiber     <- effect.ensuring(finalizer).fork
  _         <- promise.await
  _         <- fiber.interrupt
  value     <- ref.get
} yield value
```

We might think that `value` should always be `true`. We are setting it to `true` in the finalizer action of `ensuring` as we learned from the examples above and used a `Promise` to make sure that the effect has begun execution before being interrupted so the finalizer is always guaranteed to be run.

However, this doesn't work. Interruptibility is a regional setting so the most specific scope takes precedence and `clock.sleep` is interruptible because we called `interruptible` on it.

As a result, even though `delay` is called in an uninterruptible region of the finalizer, `interruptible` creates a "pocket" within that region that is interruptible. So when the runtime is evaluating instructions and gets to this point it finds that the effect has both been interrupted and is interruptible and stops executing further instructions, never running the `ref.set(true)` effect.

This is extremely counterintuitive and in general highly unsafe. Conceptually using `interruptible` at a very low level of the application can, if we are not careful, make effects that would otherwise be uninterruptible subject to interruption at much higher levels of the application.

As such, you should be extremely careful about using `interruptible`. At the time of writing the `interruptible` combinator is only used four times in ZIO's own source code outside of tests, which should give you a sense of how infrequently it is actually needed to solve problems given that ZIO is a library for asynchronous and concurrent programming.

If you do use `interruptible`, you should only do it in situations where you "handle" the interruption yourself. For example, here is a safer implementation of the `delay` combinator from above.

```
def saferDelay[R, E, A](zio: ZIO[R, E, A])(
  duration: Duration
): ZIO[R with Clock, E, A] =
  clock.sleep(duration).interruptible.run *> zio
```

By calling `run` we "handle" the potential interruption, converting it to a `ZIO` effect that succeeds with an `Exit` value with a `Cause.Interrupted`. We then ignore that result and proceed to perform the specified effect regardless.

The semantics of this variant are now that if we are interrupted we will essentially "skip" the delay and then depending on the interruptibility of the surrounding region we will either abort entirely if we are in an interruptible region or otherwise continue performing the specified effect if we are in an uninterruptible region.

Now composing `delay` into an uninterruptible region like a finalizer will merely result in the delay being skipped but the effect and the rest of the finalizer will still be performed.

Of course, the best solution here is to not use `interruptible` at all since we don't absolutely have to here.

9.4 Waiting For Interruption

The final topic we need to cover in our discussion of interruption is waiting for interruption.

We normally interrupt an effect when it is taking too long and we don't need the result. But an interrupted effect may take significant time to wind down if we need to finish work or close resources.

How does ZIO handle this?

The basic rule is that interruption does not return until all logic associated with the interrupted effect has completed execution.

Take the following example:

```
for {
  promise   <- Promise.make[Nothing, Unit]
  effect     = promise.succeed(()) *> ZIO.never
  finalizer  = ZIO.effectTotal(println("Closing file")).delay(5.seconds)
  fiber     <- effect.ensuring(finalizer).fork
  _         <- promise.await
  _         <- fiber.interrupt
  _         <- ZIO.effectTotal(println("Done interrupting"))
} yield ()
```

When `fiber` is interrupted any finalizers associated with it will immediately begin execution. In this case that will involve delaying for five seconds and then printing "Closing file" to the console.

The `fiber.interrupt` effect will not complete execution until that finalization logic has completed execution. So when we run the program above we will always see "Closing file" printed to the console before "Done interrupting".

This is valuable because it provides a guarantee that all finalization logic has been completed when interruption returns. It is also very easy to obtain the opposite behavior with this as the default, whereas it would be hard to do it the other way around.

If we want to interrupt an effect without waiting for the interruption to complete we can simply `fork` it.

```
for {
  promise   <- Promise.make[Nothing, Unit]
  effect     = promise.succeed(()) *> ZIO.never
  finalizer  = ZIO.effectTotal(println("Closing file")).delay(5.seconds)
  fiber     <- effect.ensuring(finalizer).fork
  _         <- promise.await
  _         <- fiber.interrupt.fork
  _         <- ZIO.effectTotal(println("Done interrupting"))
} yield ()
```

Now the interruption will occur on a separate fiber so "Done interrupting" will be printed to the console immediately and "Closing file" will be printed five seconds later.

ZIO also provides tools for more fine grained control of whether we want to wait for interruption to complete through the `disconnect` operator.

Say we have a similar example to the one above but we now have two files, `A` and `B`, to close in the finalizer.

`A` is critical to close and we need to know that it is closed before proceeding with the rest of our program. But `B` just needs to be closed eventually, we can proceed with the rest of our program without waiting for it.

How can we wait for the closing of `A` to complete without the closing of `B` to complete when we are interrupting a single effect?

The answer is the `disconnect` operator, which we can call on any `ZIO` effect to return a version of the effect with the interruption "disconnected" from the main fiber. If we call `disconnect` on an effect and then interrupt it the interruption will return immediately instead of waiting while the interruption logic is completed on a background fiber.

With the `disconnect` operator we can solve our problem in a very straightforward way.

```
val a: URIO[Clock, Unit] =
  ZIO.never.ensuring(ZIO.effectTotal("Closed A").delay(3.seconds))

val b: URIO[Clock, Unit] =
  ZIO.never.ensuring(ZIO.effectTotal("Closed B").delay(5.seconds)).disconnect

for {
  fiber     <- (a <&> b).fork
  _         <- clock.sleep(1.second)
  _         <- fiber.interrupt
  _         <- ZIO.effectTotal(println("Done interrupting"))
} yield ()
```

Now we will see "Closed A" printed to the console first, followed by "Done

interrupting" and finally "Closed B". The interruption now returns after the interruption of `a` completes without waiting for the interruption of `b` to complete because `b` has been disconnected.

9.5 Conclusion

This chapter has covered a lot of the mechanics of how interruption works.

The first lesson should be that this is a complex topic. Concurrency is hard and as much as ZIO tries to make it easy there is some unavoidable complexity.

When possible, take advantage of the built in operators on `ZIO`. Interruption and controlling interruptibility are building blocks and ZIO contributors have already built most of the common solutions you will need so you don't have to build them yourself.

Operators and data types like `zipPar`, `race`, `bracket`, and `ZManaged`, discussed later, already do the right thing with respect to interruption.

That being said, when you do need to write your own operators or deal with a more complex situation yourself, with the material in this chapter you have the tools to handle it.

Remember to think about how interruption is checked before each instruction in a ZIO program is executed and use `uninterruptibleMask` as your go to tool for protecting certain blocks of code from the possibility of interruption. With these tools in hand you should be well on your way!

9.6 Exercises

Chapter 10

Concurrent Structures: Ref - Shared State

So far we have learned how to describe multiple concurrent processes using fibers. But how do we exchange information between different fibers? For this we need `Ref`.

In this chapter we will learn how `Ref` gives us the ability to model mutable state in a purely functional way. We will see how `Ref` is also the purely functional equivalent of an `AtomicReference` and how we can use it to share state between fibers. We will also learn about `RefM`, a version of `Ref` that allows us to perform effects while modifying the reference, and `FiberRef`, a version of `Ref` that is specific to each fiber.

Finally, we will learn how `Ref` and `RefM` are actually type aliases for more general types, `ZRef` and `ZRefM`, and how those types give us even more functionality when working with references.

10.1 Purely Functional Mutable State

A common question is how we can model mutable state in a purely functional context. Isn't changing mutable state what we are supposed to be avoiding?

The answer to this, like several problems we have seen before, is that we create a computation which **describes** allocating and modifying mutable state, rather than modifying the state directly. This way we can model programs that mutate state while preserving referential transparency and the ability to reason about our programs with the substitution model.

Here is a simple model of the interface for a purely functional reference:

```
import zio._

trait Ref[A] {
  def modify[B](f: A => (B, A)): UIO[B]
  def get: UIO[A] =
    modify(a => (a, a))
  def set(a: A): UIO[Unit] =
    modify(_ => ((), a))
  def update[B](f: A => A): UIO[Unit] =
    modify(a => ((), f(a)))
}

object Ref {
  def make[A](a: A): UIO[Ref[A]] =
    ???
}
```

Each of the combinators on `Ref` returns a `ZIO` that describes modifying the state of the `Ref`. We can then describe a program that manipulates state like this:

```
def increment(ref: Ref[Int]): UIO[Unit] =
  for {
    n   <- ref.get
    _   <- ref.set(n + 1)
  } yield ()
```

This method take a `Ref` as an argument and returns a `ZIO` effect that describes incrementing the `Ref`. No actual changes to the value of the reference occur when we call this method and this just creates a "blueprint" for modifying the reference that we can compose with other descriptions in building up our program.

Note that the only way to create a `Ref` is with `make`, which returns a `Ref` in the context of a `ZIO` effect. One common mistake is forgetting that allocating mutable state is also an effect that needs to be suspended in an effect constructor. To see this, imagine we exposed a method `unsafeMake` that allocated a `Ref` outside the context of an effect.

```
trait Ref[A] {
  def modify[B](f: A => (B, A)): UIO[B]
  def get: UIO[A] =
    modify(a => (a, a))
  def set(a: A): UIO[Unit] =
    modify(_ => ((), a))
  def update[B](f: A => A): UIO[Unit] =
    modify(a => ((), f(a)))
}
```

```
object Ref {
  def make[A](a: A): UIO[Ref[A]] =
    ???
  def unsafeMake[A](a: A): Ref[A] =
    ???
}
```

We could then write a simple program that creates and increments two references:

```
lazy val ref1: Ref[Int] = Ref.unsafeMake(0)
lazy val ref2: Ref[Int] = Ref.unsafeMake(0)

lazy val result = for {
  _ <- ref1.update(_ + 1)
  _ <- ref2.update(_ + 1)
  l <- ref1.get
  r <- ref2.get
} yield (l, r)
```

This program creates two references and increments each of them be one, returning `(1, 1)`.

Now let's say we refactor this program. `Ref.unsafeMake(0)` is the right hand side of both `ref1` and `ref2` so we should be able to extract it out:

```
lazy val makeRef: Ref[Int] = Ref.unsafeMake(0)

lazy val ref1: Ref[Int] = makeRef
lazy val ref2: Ref[Int] = makeRef

lazy val result = for {
  _ <- ref1.update(_ + 1)
  _ <- ref2.update(_ + 1)
  l <- ref1.get
  r <- ref2.get
} yield (l, r)
```

But this time we have only created a single reference and `ref1` and `ref2` refer to the same reference. So now we increment this single reference twice instead of incrementing two separate references once, resulting in a return value of `(2, 2)`. This inability to use the substitution model when allocating mutable state shows that even the creation of mutable state is an effect that must be suspended in an effect constructor. By only exposing `make` instead of `unsafeMake` we prevent this.

Here is the original program again without the unsafe method:

```
lazy val makeRef1: UIO[Ref[Int]] = Ref.make(0)
lazy val makeRef2: UIO[Ref[Int]] = Ref.make(0)
```

```
lazy val result = for {
  ref1 <- makeRef1
  ref2 <- makeRef2
  _    <- ref1.update(_ + 1)
  _    <- ref2.update(_ + 1)
  l    <- ref1.get
  r    <- ref2.get
} yield (l, r)
```

And here is the refactored program:

```
lazy val makeRef: UIO[Ref[Int]] = Ref.make(0)

lazy val makeRef1: UIO[Ref[Int]] = makeRef
lazy val makeRef2: UIO[Ref[Int]] = makeRef

lazy val result = for {
  ref1 <- makeRef1
  ref2 <- makeRef2
  _    <- ref1.update(_ + 1)
  _    <- ref2.update(_ + 1)
  l    <- ref1.get
  r    <- ref2.get
} yield (l, r)
```

Now `make` is only a description of the act of creating a mutable reference so we are free to extract it into its own variable.

The lesson is to always suspend the creation of mutable state in an effect constructor.

10.2 Ref As Purely Functional Equivalent Of An Atomic Reference

Not only does `Ref` allow us to describe mutable state in a purely functional way, it also allows us to do so in a way that is safe for concurrent access. So far we have described how to model mutable state in a purely functional way but we haven't needed anything special about `Ref` to do that. We could just wrap creating and modifying a mutable variable in an effect and do the same thing.

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

object Var {
```

```
  def make[A](a: A): UIO[Var[A]] =
    UIO.effectTotal {
      new Var[A] {
        var a0 = a
        def get: UIO[A] =
          UIO.effectTotal(a0)
        def set(a: A): UIO[Unit] =
          UIO.effectTotal {
            a0 = a
            ()
          }
        def update(f: A => A): UIO[Unit] =
          UIO.effectTotal(a0 = f(a0))
      }
    }
}
```

This allows us to describe state in a purely functional way. But what happens if we want to share that state across multiple fibers like in the program below?

```
for {
   variable <- Var.make(0)
   _        <- UIO.foreachPar_((1 to 10000).toList)(_ => variable.update(_ + 1))
   result   <- variable.get
} yield result
// res0: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@189eecba
```

The result of this program will be nondeterminate. A mutable variable is not safe for concurrent access. So we can have a situation where one fiber reads a value from the variable at the beginning of the `update` operation, say `2`, and another fiber reads the value before the first fiber writes the updates value. So the second fiber sees `2` as well and both fibers write `3`, resulting in a value of `3` instead of the expected value of `4` after both fibers have completed their updated.

This is basically the problem with using a mutable variable instead of an atomic one in side effecting code. The solution is the same here, replace `Int` with `AtomicInteger` or in general replace a `var` with a `AtomicReference`. We can do the same thing in implementation here to come quite close to the implementation of `Ref`.

```
import java.util.concurrent.atomic.AtomicReference

trait Ref[A] {
  def modify[B](f: A => (B, A)): UIO[B]
  def get: UIO[A] =
    modify(a => (a, a))
  def set(a: A): UIO[Unit] =
    modify(_ => ((), a))
```

```
  def update[B](f: A => A): UIO[Unit] =
    modify(a => ((), f(a)))
}

object Ref {
  def make[A](a: A): UIO[Ref[A]] =
    UIO.effectTotal {
      new Ref[A] {
        val atomic = new AtomicReference(a)
        def modify[B](f: A => (B, A)): UIO[B] =
          UIO.effectTotal {
            var loop = true
            var b: B = null.asInstanceOf[B]
            while (loop) {
              val current = atomic.get
              val tuple   = f(current)
              b = tuple._1
              loop = !atomic.compareAndSet(current, tuple._2)
            }
            b
          }
      }
    }
}
```

`Ref` is backed by an `AtomicReference` internally, which is a mutable data structure that is safe for concurrent access. The `AtomicReference` is not exposed directly so the only way it can be accessed is through the combinators we provide, which all return effects.

In the implementation of `modify` itself we use a compare-and-swap approach where we get the value of the `AtomicReference`, compute the new value using the specified function, and then only set the `AtomicReference` to the new value if the current value is still equal to the original value. That is, if no other fiber has modified the `AtomicReference` between us reading from and writing to it. If another fiber has modified the `AtomicReference` we just retry.

This is similar to the implementation of combinators on at `AtomicReference`, we are just wrapping the state modification in an effect constructor and implementing the `modify` function, which is very useful in implementing other combinators on `Ref`.

With this new implementation, we can safely write to a `Ref` from multiple fibers at the same time.

```
for {
   ref <- Ref.make(0)
   _        <- UIO.foreachPar_((1 to 10000).toList)(_ => ref.update(_ + 1))
```

```
   result   <- ref.get
} yield result
// res1: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@7e321bd1
```

This will now return `10000` every time it is run.

10.3 Operations Are Atomic But Do Not Compose Atomically

One important limitation of working with `Ref` to be aware of is that while each individual operation on `Ref` is performed atomically, combined operations are not performed atomically.

In the example above we could safely call `update` to increment the value of the `Ref` because the update function was performed atomically. From the perspective of the caller, `update` always reads the old value and writes the new value in a single "transaction" without other fibers being able to modify the value in the middle of this transaction.

But this guaranteed doesn't apply if we combine more than one combinator on `Ref`. For example, the following is not correct:

```
for {
   ref <- Ref.make(0)
   _         <- UIO.foreachPar_((1 to 10000).toList) { _ =>
    ref.get.flatMap(n => ref.set(n + 1))
   }
   result   <- ref.get
} yield result
// res2: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@696e354e
```

This is the same program as above expect we have replaced `ref.update(_ + 1)` with `ref.get.flatMap(n => ref.set(n + 1))`. These may appear to be the same because they both increment the value of the reference by one. But the first one increments the reference atomically, the second one does not. So the program above will be nondeterminate and will typically return a value less than `10000` because multiple fibers will read the same value.

In this example the issue was somewhat obvious, but in others it can be more subtle. A similar problem is having pieces of mutable state in two different references. Because operations on `Ref` do not compose atomically, we do not have a guarantee that the state of the two references will be consistent at all times.

Here are some best practices that will help you use `Ref` most effectively in cases where multiple fibers are interacting with the same `Ref` and avoid these problems:

1. Put all state that needs to be "consistent" with each other in a single `Ref`.

2. Always modify the `Ref` through a single operation

Although these guidelines create some limitations they are the same ones you would have working with an `AtomicReference` in imperative code, so they should feel relatively familiar if you have worked with atomic variables before in concurrent programming. And we can actually implement a very large number of concurrency combinators and data structures with just `Ref` and the `Promise` data type we will describe in the next chapter.

If you do need to be able to compose operations on a reference, you can use the Software Transactional Memory (STM) functionality in ZIO. See the section on STM for an in depth discussion of this.

10.4 RefM For Evaluating Effects While Updating

Another limitation of `Ref` is that we can't perform effects inside the `modify` operation. Going back to the interface for `Ref`, the signature of `modify` is:

```
trait Ref[A] {
  def modify[B](f: A => (B, A)): UIO[B]
}
```

The function `f` here must be a pure function. It can transform the old value to a new value, for example incrementing an integer or changing the state of a case class from `Open` to `Closed` but it can't perform effects such as logging a value to the new or allocating new mutable state.

This limitation is not arbitrary but is actually fundamental to the implementation of `Ref`. We saw above that `Ref` is implemented in terms of `compare-and-swap` operations where we get the old value, compute the new value, and then only set the new value if the current value is equal to the old value. Otherwise we retry.

This is safe to do because computing the new value has no side effects, so we can do it as many times as we want and it is exactly the same as if we did it a single time. We can increment an integer once or a hundred times and it will be exactly the same other than us having used slightly more computing power.

On the other hand, evaluating an effect many times is not the same as evaluating it a single time. Evaluating `console.putStrLn("Hello, World!")` a hundred times will print "Hello, World!" to the console a hundred times versus evaluating it once will only print it a single time. So if we perform side effects in `modify` they could be repeated an arbitrary number of times in a way that would be observable to the caller and would violate the guarantees of `Ref` that updates should be performed atomically and only a single time.

In most cases, this is not actually a significant limitation. In the majority of cases where we want to use effects within `modify` we can refactor our code to

return a value and then perform the effect with the value. For example, here is how we could add console logging to an update operation:

```
import zio.console._

def updateAndLog[A](ref: Ref[A])(f: A => A): URIO[Console, Unit] =
  ref.modify { oldValue =>
    val newValue = f(oldValue)
    ((oldValue, newValue), newValue)
  }.flatMap { case (oldValue, newValue) =>
    console.putStrLn(s"updated $oldValue to $newValue")
  }
```

For cases where we really do need to perform effects within the `modify` operation, ZIO provides another data type called `RefM`. The interface for `RefM` is the same as the one for `Ref` except we can now perform effects within the `modify` operation.

```
trait RefM[A] {
  def modify[R, E, B](f: A => ZIO[R, E, (B, A)]): ZIO[R, E, B]
}
```

The guarantees of `RefM` are similar to those of `Ref`. Each operation on `RefM` is guaranteed to be atomically. In this case that means that once one fiber begins modifying the reference, all other fibers attempting to modify the reference must suspend until the first fiber completes modifying the reference. Of course, as with all other operations on fibers suspending does not actually block any underlying operating system threads but just registers a callback to be invoked when the first fiber is finished.

Internally, `RefM` is implemented using a `Semaphore` to ensure that only one fiber can be interacting with the reference at the same time. We'll learn more about ZIO's implementation of a semaphore later but for now you can just think of it as a semaphore in other concurrent programming paradigms that guards access to some shared resource and only permits a certain number of concurrent processes to access it at the same time, in this case one.

One of the most common use cases of `RefM` is when we need to allocate some mutable state within the update operation. For example, say we want to implement a `RefCache`. This will be similar to a normal cache except each value in the cache will be a `Ref` so the values can themselves be shared between multiple fibers.

```
import zio._

trait RefCache[K, V] {
  def getOrElseCompute(k: K)(f: K => V): UIO[Ref[V]]
}

object RefCache {
```

```
  def make[K, V]: UIO[RefCache[K, V]] =
    RefM.make(Map.empty[K, Ref[V]]).map { refM =>
      new RefCache[K, V] {
        def getOrElseCompute(k: K)(f: K => V): UIO[Ref[V]] =
          refM.modify { map =>
            map.get(k) match {
              case Some(ref) => UIO.succeed((ref, map))
              case None      => Ref.make(f(k)).map(ref => (ref, map + (k -> ref)))
            }
          }
      }
    }
}
```

We want the `getOrElseUpdate` operation to be atomic but we potentially need to execute an effect within it to create a new `Ref` if one doesn't already exist in the cache, so we need to use `RefM` here.

`RefM` is never going to be as fast as using `Ref` because of the additional overhead required to make sure that only a single fiber is interacting with the reference at the same time. But sometimes we need that additional power. When we do use `RefM`, the key is to do as little work as possible within the `modify` operation of `RefM`. Effects occurring within the `modify` operation can only occur one at a time, whereas effects outside of it can potentially be done concurrently. So if possible do the minimum work possible within `modify` (e.g. creating a `Ref`, forking a fiber) and perform additional required effects based on the return value of the `modify` operation.

10.5 FiberRef For References Specific To Each Fiber

`Ref` represents some piece of mutable state that is shared between fibers, so it is extremely useful for communicating between fibers. But sometimes we need to maintain some state that is local to each fiber. This would be the equivalent of a `ThreadLocal` in Java. For example, we might want to support logging that is specific to each fiber.

To define a `FiberRef`, in addition to defining an initial value like with a `Ref`, we also define a `fork` operation that defines how, if at all, the value of the parent reference from the parent fiber will be modified to create a copy for the new fiber when the child is forked, and an operation `join` that specifies how the value of the parent fiber reference will be combined with the value of the child fiber reference when the fiber is joined back.

```
def make[A](
  initial: A,
```

```
  fork: A => A,
  join: (A, A) => A
): UIO[FiberRef[A]] =
  ???
```

To see how this works, let's look at how we could use `FiberRef` to create a logger that captures the structure of how computations were forked and joined.

To do this, we will first create a simple data structure that will capture the tree like structure of how a fiber can fork zero or more other fibers, which can themselves fork zero or more fibers recursively to an arbitrary depth.

```
final case class Tree[+A](head: A, tail: List[Tree[A]])
```

A tree has a `head` with a value of type `A`, which in our case will represent the log for the particular fiber. The `tail` will represent logging information for child fibers, so if the fiber did not fork any other fibers the list will be empty and if it forked five other fibers there will be five elements in the list.

We will set the type of `A` to `Chunk[String]` here, using ZIO's `Chunk` data type to support efficiently appending to the log for each fiber and allowing logging arbitrary strings. We could imagine using a more complex data type instead of `String` to support logging in a more structured format.

```
type Log = Tree[Chunk[String]]
```

We then need to specify implementations for the `initial`, `fork`, and `join` parameters to `make`.

```
val loggingRef: UIO[FiberRef[Log]] =
  FiberRef.make[Log](
    Tree(Chunk.empty, List.empty),
    _ => Tree(Chunk.empty, List.empty),
    (parent, child) => parent.copy(tail = child :: parent.tail)
  )
// loggingRef: UIO[FiberRef[Log]] = zio.ZIO$FiberRefNew@7fbf9418
```

When we initially create the `FiberRef` we set its value to an empty log. In this case we only want child fibers writing to the log so when we fork a new fiber we ignore the value of the parent's log and just give the child fiber a new empty log that it can write to.

Finally, the `join` logic is most important. In the `join` function the first `A` represents the parent fiber's `FiberRef` value and the second `A` value represents the child fiber's `FiberRef` value. So when we join a fiber back we want to take the child fiber's log and add it to the collection of child fiber logs in `tail`.

With this, we have all the logic necessary to do logging that reflects the structure of concurrent computations.

To see it in action let's create a `loggingRef` and do some logging in a simple application that uses multiple fibers. To make things easier we will also define a

helper method for writing to the log.

```
def log(ref: FiberRef[Log])(string: String): UIO[Unit] =
  ref.update(log => log.copy(head = log.head :+ string))

for {
  ref <- loggingRef
  left = for {
    a <- ZIO.succeed(1).tap(_ => log(ref)("Got 1"))
    b <- ZIO.succeed(2).tap(_ => log(ref)("Got 2"))
  } yield a + b
  right = for {
    c <- ZIO.succeed(1).tap(_ => log(ref)("Got 3"))
    d <- ZIO.succeed(2).tap(_ => log(ref)("Got 4"))
  } yield c + d
  fiber1 <- left.fork
  fiber2 <- right.fork
  _      <- fiber1.join
  _      <- fiber2.join
  log    <- ref.get
  _      <- console.putStrLn(log.toString)
} yield ()
// res4: ZIO[console.package.Console, Nothing, Unit] = zio.ZIO$FlatMap@2b3bdb3c
```

If you run this program you will get a tree with three nodes. The parent node will not have any logging information, because we did not log any information in the parent fiber.

The two child nodes will each have information in the logs, with one node showing `Got 1` and `Got 2` and the other node showing `Got 3` and `Got 4`. This let's us see what happened sequentially on each fiber as well as what happened concurrently on different fibers.

You could imagine implementing a renderer for this tree structure that would display this logging information in a visually accessible way that would allow seeing exactly what happened where, though this is left as an exercise for the reader.

One other operator on `FiberRef` that is particularly useful is `locally`:

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

This sets thhe `FiberRef` to the specified value, runs the `zio` effect, and then sets the value of the `FiberRef` back to its original value. The value is guaranteed to be restored to the original value immediately after the `zio` effect completes execution, regardless of whether it completes successfully, fails, or is interrupted.

Because the value of the `FiberRef` is specific to each fiber, the change to the

value of the `FiberRef` is guaranteed to only be visible to the current fiber, so as the name implies the change in the value of the `FiberRef` is locally scoped to only the specified `zio` effect.

This pattern can be particularly useful when the `FiberRef` contains some configuation information, for example the logging level. The `locally` operator could then be used to change the logging level for a specific effect without changing it for the rest of the application.

The `locally` operator also composes, so you could for example use `locally` in one part of your application to turn off logging but then inside that part of your application call `locally` again in one particular area to turn logging back on.

```
lazy val doSomething: ZIO[Any, Nothing, Unit] = ???
lazy val doSomethingElse: ZIO[Any, Nothing, Unit] = ???

for {
  logging <- FiberRef.make(true) // logging is enabled by default
  _       <- logging.locally(false) { // logging is disabled in this region
    for {
      _ <- doSomething
      _ <- logging.locally(true)(doSomethingElse) // logging is reenabled for this effect
    } yield ()
  }
} yield ()
// res5: ZIO[Any, Nothing, Unit] = zio.ZIO$FlatMap@494298fa
```

Using a `FiberRef` in the environment can be a particularly powerful pattern for supporting locally scoped changes to configuration parameters like this.

10.6 Conclusion

With the materials in this chapter you have powerful tools at your disposal for managing concurrent state between fibers.

The key thing to remember is to always use the `modify` and `update` operators to make changes to references versus separately using `get` and `set` because while individual operations on references are atomic, operations do not compose atomically.

Also try to use `Ref` versus `RefM` wherever possible.

`Ref` is significantly more performant because it is implemented directly in terms of compare and swap operations versus a `RefM` which has to force other fibers to semantically block while an effectual update is being performed. In most cases code that performs effects within the `modify` oepration can be refactored to to return an updated state that then allows the appropriate effect to be performed outside of the `modify` operation.

Finally, make sure that you are not using mutable data structure inside a `Ref` and that the update operations you do are relatively fast.

It is easy to think of a `Ref` as being updated by one fiber at a time because the atomicity of the updates guarantees us that it looks like that is the case. However, that is actually achieved by retrying if there is a conflicting update as described above, so if your update operation takes a very long time there is a risk that it will continually be forced to retry by conflicting updates, leading to poor performance.

If you use atomic operations on `Ref`, use immutable data structures inside it, and keep your update operations fast you will be well on your way to using `Ref` as one of the core building blocks to solving even the most advanced concurrency problems.

In the next chapter we will talk about `Promise`. While `Ref` allows sharing some piece of state between fibers, it doesn't provide any way waiting for some state, we can just get the current state and do something with it.

`Promise` allows us to wait for some state to be set, semantically blocking until it occurs, and so forms the key building block along with `Ref` for building more complicated concurrent data structures.

With that context, let's read on!

10.7 Exercises

Chapter 11

Concurrent Structures: Promise - Work Synchronization

In the last chapter we learned about `Ref`, one of the fundamental data structure for concurrent programming with ZIO. In this chapter, we will learn about `Promise`, the other basic building block for solving problems in concurrent programming.

`Ref` is excellent for sharing state between fibers, but it provides no way to **synchronize** between fibers. When a fiber interacts with a `Ref` it always immediately gets, sets, or updates the value of the reference and then continues. A fiber has no way to wait for another fiber to set a `Ref` to a particular value other than polling, which is extremely inefficient.

Of course, we can wait for the result of a fiber using `join` or `await`. But many times this functionality may not be sufficient because we want to wait for a value to be set without knowing which fiber will ultimately be responsible for setting it. For example, if we are implementing a mailbox we may want to wait for the first message in the mailbox but not know which fiber will deliver it. In fact there may be many fibers running concurrently delivering messages to the mailbox, so it may be impossible for us to know in advance which fiber will deliver the next message.

These are the use cases for which `Promise` shines. A simplified version of the interface of `Promise` is as follows:

```
import zio._

trait Promise[E, A] {
```

```
  def await: IO[E, A]
  def fail(e: E): UIO[Boolean]
  def succeed(a: A): UIO[Boolean]
}
```

A `Promise` can be thought of as a container that has one of two states: (1) empty or (2) full with either a failure `E` or a value `A`. Unlike a `Ref`, which always contains some value, a `Promise` begins its lifetime in an empty state. The promise can be completed **exactly once** using `succeed` to fill it with a value or `fail` to fill it with an error. Once a promise is full with a value or a failure it cannot be completed again and will always have that value.

We can get the value out of a `Promise` with the `await` combinator. `await` will semantically block until the promise is completed, returning a new computation that either succeeds or fails with the result of the promise. Thus, by creating a `Promise` and having a fiber `await` its completion, we can prevent that fiber from proceeding until one or more other fibers signal that it may do so by completing the `Promise`.

To see a simple example of this, let's fork two fibers and use a `Promise` to synchronize their work:

```
for {
  promise <- Promise.make[Nothing, Unit]
  left    <- (console.putStr("Hello, ") *> promise.succeed(())).fork
  right   <- (promise.await *> console.putStr(" World!")).fork
  _       <- left.join *> right.join
} yield ()
// res0: ZIO[console.package.Console, Nothing, Unit] = zio.ZIO$FlatMap@6a55f6a7
```

Without the `Promise`, this program would exhibit nondeterminism. "Hello," and " World!" are being printed to the console on different fibers so the output could be "World! Hello," instead of "Hello, World!".

The `Promise` allows us to coordinate the work of the different fibers. Even if the runtime begins execution of the `right` fiber first, the fiber will suspend on `promise.await` since the `Promise` has not been completed yet. The `left` fiber, on the other hand, will print its output to the console and then complete the `Promise`. Once the `Promise` is completed, the `right` fiber will resume and print its output to the console.

Using the `Promise` we have taken two parts of a program that would have operated concurrently and imposed a linear ordering between them. In this simple example, the resulting program is no different than if we had written the program entirely sequentially. But in more complex situations our goal is to use tools like `Promise` to impose the minimum amount of sequencing necessary for correctness, while otherwise allowing the program to be as concurrent as possible for performance.

11.1 Various Ways of Completing Promises

The description of a `Promise` above is actually incomplete. A completed promise actually contains an `IO[E, A]`, an effect that can either fail with an `E` or succeed with an `A`, rather than just a failure `E` or a value `A`.

Most of the time this distinction does not matter. The `succeed` and `fail` methods above just complete a `Promise` with a computation that succeeds with the specified value or fails with the specified error, respectively. We start to see this distinction with some of the other ways of completing promises:

```
trait Promise[E, A] {
  def die(t: Throwable): UIO[Boolean]
  def done(e: Exit[E, A]): UIO[Boolean]
  def halt(e: Cause[E]): UIO[Boolean]
}
```

These all allow completing promises with other types of effects. `die` allows completing a promise with an effect that dies with the specified `Throwable`. `done` allows completing a promise with an effect that returns the specified `Exit` value. `halt` allows completing a promise with an effect that dies with the specified `Cause`.

Having this variety of methods makes sense because calling `await` on a `Promise` returns the computation inside the `Promise`, when it is available. So calling `await` on a `Promise` that was completed with `die` will result in an effect that dies with the specified `Throwable`. This allows us to propagate a greater range of information, for example the full cause of an error, than we could if a `Promise` could only be completed with a failure `E` or a value `A`.

The distinction between completing a promise with the result of an effect versus with the effect itself becomes even more clear when comparing the behavior of the `complete` and `completeWith` combinators.

```
trait Promise[E, A] {
  def complete(io: IO[E, A]): UIO[Boolean]
  def completeWith(io: IO[E, A]): UIO[Boolean]
}
```

`complete` completes a `Promise` with the result of an effect. This means that the effect will be evaluated a single time when the `Promise` is completed and then that result will be provided to all callers of `await`. On the other hand, `completeWith` completes a promise with the effect itself. This means that the effect will not be evaluated when the promise is completed but once each time `await` is called on it.

To illustrate this, consider the following two programs:

```
import zio._
```

```scala
import scala.util.Random

val randomInt: UIO[Int] =
  UIO.effectTotal(Random.nextInt)
// randomInt: UIO[Int] = zio.ZIO$EffectTotal@3fb487ff

val complete: UIO[(Int, Int)] =
  for {
    p <- Promise.make[Nothing, Int]
    _ <- p.complete(randomInt)
    l <- p.await
    r <- p.await
  } yield (l, r)
// complete: UIO[(Int, Int)] = zio.ZIO$FlatMap@6502c9da

val completeWith: UIO[(Int, Int)] =
  for {
    p <- Promise.make[Nothing, Int]
    _ <- p.completeWith(randomInt)
    l <- p.await
    r <- p.await
  } yield (l, r)
// completeWith: UIO[(Int, Int)] = zio.ZIO$FlatMap@66611bc4
```

In `complete`, the `Promise` is completed with the result of the `randomInt` effect. When `complete` is called `randomInt` will be evaluated and a particular random number will be generated. `l` and `r` will then return effects that simply succeed with that random number. So in the first program `l` will always be equal to `r`.

In contrast, in `completeWith`, the `Promise` is completed with the `randomInt` effect itself. Calling `completeWith` does not evaluate the `randomInt` effect at all, but merely completes the `Promise` with that effect. `l` and `r` will then each evaluate the `randomInt` and return a different random number. So in the second program `l` will almost always be different from `r`.

The most common use case of promises is to complete them with values using combinators such as `succeed` and `fail`, so generally you will not have to worry about this distinction. But when you are completing promises with effects it is helpful to keep in mind. `completeWith` is more efficient than `complete` so if `await` is only going to be called on a `Promise` once you should prefer `completeWith`. If you need the behavior of "memoizing" the result of the computation to be shared between multiple fibers then you should use `complete`.

All of these methods return a `UIO[Boolean]`. The returned value will be `false` if the `Promise` has already been completed and `true` otherwise. In many situations this may not matter because we may not care which fiber completes the `Promise`, just that some fiber completes it. However, this can be important in more advanced use cases where we want to take some further action if the `Promise`

was already completed.

11.2 Waiting On A Promise

We have already seen `await`, which allows a fiber to wait for a `Promise` to be completed. In addition to this there are several other methods on `Promise` that can be useful to observe its state:

```
trait Promise[E, A] {
  def await: IO[E, A]
  def isDone: UIO[Boolean]
  def poll: UIO[Option[IO[E, A]]]
}
```

As described above, `await` suspends until the `Promise` has been completed and then returns the effect that the `Promise` was completed with. Thus, if the effect the `Promise` was completed with is a success, the effect returned by `await` will succeed with that value. If the `Promise` is completed with an effect that fails or dies, that error will be propagated to the effect returned by `await`.

It is also possible to poll for completion of the `Promise` using `poll`. This returns an `Option[IO[E, A]]`. The returned value will be `None` if the `Promise` has not been completed, and `Some` with an effect if the `Promise` has been completed. The return type of `IO[E, A]` versus `Either[E, A]` is another indication of the fact that a `Promise` is completed with an effect. While repeatedly polling is not recommended, `poll` can be useful if we want to take some action if the result of the `Promise` does happen to already be available and otherwise do something else.

Finally, `isDone` allows us to check whether the `Promise` has been completed when we only care about whether the `Promise` has been completed and not its value. This can easily be implemented in terms of `poll` as `poll.map(_.nonEmpty)` so this is more of a convenience method.

11.3 Promises And Interruption

Promises support ZIO's interruption model. This means that if a `Promise` is completed with an effect that is interrupted, all fibers waiting on that `Promise` will also be immediately interrupted.

In general, this isn't something you have to worry about but "just works". If a fiber is waiting on a `Promise` that has been interrupted then that `Promise` is never going to complete with a value so it doesn't make sense for the process to continue. With `Promise`, this happens automatically.

You can also interrupt a `Promise` yourself using `interrupt` and `interruptAs` combinators.

```
trait Promise[E, A] {
  def interrupt: UIO[Boolean]
  def interruptAs(fiberId: Fiber.Id): UIO[Boolean]
}
```

`interrupt` completes the `Promise` with an interruption, immediately interrupting any fibers waiting on the `Promise`. The identifier of the fiber doing the interruption will be the fiber that calls `interrupt`. If you want to specify a different fiber as the interruptor you can use `interruptAs` which allows you to specify the identifier of a different fiber as the one responsible for the interruption. `interruptAs` can be useful for library authors to provide more accurate diagnostics in traces and fiber dumps if one fiber is really doing the interruption "on behalf of" another but should not normally be necessary in user code.

11.4 Combining Ref And Promise For More Complicated Concurrency Scenarios

`Ref` and `Promise` are both very powerful data types themselves. When put together they provide the building blocks for solving many more complicated concurrency problems. In general if you need to share some state between different fibers think about using a `Ref`. If you need to synchronize fibers so that something doesn't happen on one fiber until something else happens on another fiber think about a `Promise`. If you need both of these as part of your solution, use a combination of `Promise` and `Ref`.

To see how we can use a combination of `Promise` and `Ref` to build an asynchronous cache. The cache will contain values of type `Key` and values of type `Value`. Conceptually the cache may be in one of three states:

1. The key does not exist in the cache
2. The key exists in the cache and there is a value associated with it
3. the key exists in the cache and there is some other fiber currently computing a value associated with it.

The possibility of this third state is what makes this an asynchronous cache.

We can model such a cache using the following interface:

```
trait Cache[-K, +E, +V] {
  def get(key: K): IO[E, V]
}

object Cache {
  def make[K, R, E, V](lookup: K => ZIO[R, E, V]): URIO[R, Cache[K, E, V]] =
    ???
}
```

A `Cache` is defined in terms of the `get` function. If the key exists in the cache and

there is a value associated with it `get` should return an effect that succeeds with that value. If the key exists in the cache and another fiber is already computing the value associated with it `get` should return the result of that computation when it is complete. If the key does not exist in the cache then `get` should insert the key into the cache, begin computing the value associated with it, and return the result of that computation when it is complete.

The `Cache` is created using a `make` method. The `make` method takes a `lookup` function that will be used to compute new values in the cache. It returns an effect that describes the creation of the effect, since we will need to allocate mutable memory and potentially perform other effects to create the cache.

An implementation could look like this:

```
import zio._

trait Cache[-K, +E, +V] {
  def get(key: K): IO[E, V]
}

object Cache {
  def make[K, R, E, V](lookup: K => ZIO[R, E, V]): URIO[R, Cache[K, E, V]] =
    for {
      r   <- ZIO.environment[R]
      ref <- Ref.make[Map[K, Promise[E, V]]](Map.empty)
    } yield new Cache[K, E, V] {
      def get(key: K): IO[E, V] =
        Promise.make[E, V].flatMap { promise =>
          ref.modify { map =>
            map.get(key) match {
              case Some(promise) => (Right(promise), map)
              case None => (Left(promise), map + (key -> promise))
            }
          }.flatMap {
            case Left(promise) => lookup(key).provide(r).to(promise) *> promise.await
            case Right(promise) => promise.await
          }
        }
    }
}
```

Here our implementation is backed by a `Ref[Map[K, Promise[V]]]`. The `Ref` handles the shared mutable state of which keys are in the cache. The `Promise` handles synchronization between the fibers, allowing one fiber to wait for a value in the cache associated with another fiber.

Let's walk through the code.

In the `make` function, we first access the environment using `ZIO.environment`

and create the data structure that is going to back the `Cache` using `Ref.make`. We need to access the environment because our `lookup` function returns an effect that requires an environment `R`, but from its signature our `Cache` does not have any environmental requirements. Rather, the environmental requirement is expressed at the level of the outer effect that returns the `Cache`, `URIO[R, Cache[K, E, V]]`. So we need to access the environment now and provide it to `lookup` function each time it is called.

With these effects performed, we can now return a `Cache` and implement its `get` method. In `get` we begin by creating a `Promise` which we will use later. We then use `modify` to atomically modify the `Map` backing the `Cache`. It is important to modify the `Ref` atomically here because if two fibers are calling `get` on the `Cache` at the same time with the same key we want to make sure that the `lookup` function is only computed once.

Within `modify`, we check if the key already exists in the `Map`. If it does, we simply return the `Promise` wrapped in a `Right` and leave the `Map` unchanged. Here `Right` indicates that another fiber is already computing the value so this fiber can just await the result of the `Promise`. On the other hand, if the key does not already exist in the `Map` then we return the `Promise` we previously created wrapped in a `Left` and add a new binding with the key and promise to the `Map`. `Left` indicates that no other fiber is computing this value so the `lookup` function must be invoked to compute it.

Finally, we perform a further computation using the result of `modify` by calling `flatMap` on its result. If the result of `modify` was `Right` we can just await the result of the `Promise` since another fiber is already computing the value. If the result is `Left` then we must complete the `Promise` ourself, which we do by calling the `lookup` function with the key and providing the environment. We use the `to` combinator on `ZIO`, which is a shorthand for `Promise.complete(io)`.

We want to use `complete` versus `completeWith` because we want the `lookup` function to be computed once now and for its result to be available to all callers getting the associated key from the cache, rather than being recomputed each time.

There are a variety of areas for further work here, but we can see how with very few lines of code we were able to build a relatively sophisticated concurrent data structure. We used `Ref` to manage the shared mutable state of the cache. We used `Promise` to manage synchronization between fibers computing values and fibers accessing values currently being computed. These data structures fit together very well to build a solution to a more complex problem out of solutions to smaller problems. And by using these these data structures we got several important guarantees, like atomic updates and waiting without blocking "out of the box".

11.5 Conclusion

In this chapter we learned about `Promise`, the second fundamental concurrent data structure in ZIO. We saw how a `Promise` could be used to synchronize work between fibers. We also learned about various way to complete and wait on promises, as well as how promises interact with interruption. Finally, we discussed how `Ref` and `Promise` can be combined to build solutions to more sophisticated concurrency problems and walked through an example of building an asynchronous cache.

In the next chapter we will talk about queues, which can be thought of in some ways as generalizations of promises. Queues allow a producer to "offer" many values to a queue versus a promise which can only be completed with a single value. Queues also allow a consumer to "take" many values from a queue whereas a promise, once completed, will only ever have a single value.

11.6 Exercises

Chapter 12

Concurrent Structures: Queue - Work Distribution

Queues are the next concurrent data structure in ZIO.

Another excellent use case of queues is serving as buffers between push based and pull based interfaces. As we will see later, ZIO Stream is a pull based streaming solution, meaning that the downstream consumer "pulls" values from upstream producer as it is ready to consume them. In contrast, other interfaces are push based, meaning that the upstream producer "pushes" values to the downstream consumer as it produces them.

12.1 Queues As Generalizations Of Promises

Like a `Promise`, a `Queue` allows fibers to suspend waiting to offer a value to the `Queue` or taking a value from the `Queue`. But unlike a `Promise`, a `Queue` can potentially contain multiple values and values can be removed from the `Queue` in addition to being placed in the `Queue`.

While promises are particularly good for synchronizing work between fibers, queues are excellent for distributing work among fibers.

A promise is completed once with a single value and never changes, so it very good for allowing one fiber to wait on the result of some work and then doing something with it. However, a promise can only be completed with a single value so it does not support the notion of multiple "units" of work that may need to be distributing among different fibers. In addition, the value of a promise cannot be changed once it is completed so there is no way for a fiber to signal that it is "taking responsibility" for some unit of work and other fibers should not do that work.

In contrast, queues can contain multiple values and allow both offering and taking values, so it is easy do distribute work among multiple fibers by having them each take values from a queue that represent some work that needs to be done.

12.2 Offering And Taking Values From A Queue

The basic operations of a queue are `offer` and `take`:

```
import zio._

trait Queue[A] {
  def offer(a: A): UIO[Boolean]
  def take: UIO[A]
}
```

`offer` offers a value to the queue. `take` removes a value from the `Queue`, suspending until there is a value in the `Queue` to take.

Here is a simple example of using `offer` and `take` with a `Queue`:

```
for {
  queue <- Queue.unbounded[Int]
  _     <- ZIO.foreach(List(1, 2, 3))(queue.offer)
  value <- ZIO.collectAll(ZIO.replicate(3)(queue.take))
} yield value
// res0: ZIO[Any, Nothing, Iterable[Int]] = zio.ZIO$FlatMap@100ac953
```

This will simply offer the values `1`, `2`, and `3` to the queue and then take each of those values from the `Queue`. Queues always operate on a First In First Out ("FIFO") basis. So since `1` was the first value offered from the queue it will also be the first value taken from the queue and so on.

Notice that we used the `unbounded` constructor on `Queue` as opposed to the `make` constructor we saw before for `Ref` and `Promise`. Unlike references and promises, which are all "the same" other than their type parameters, different types of queues can be constructed, including unbounded queues, bounded queues with back pressure, sliding queues, and dropping queues. We will discuss the varieties of queues in the next section. For now we will focus on unbounded queues as we discuss the basic operations on queues.

`take` will semantically block until there are values in the `Queue` to take. This allows us to create workflows that repeatedly take values from the `Queue` without worry that we will be blocking threads or polling.

```
for {
  queue <- Queue.unbounded[Int]
  _     <- queue.take.flatMap(n => console.putStrLn(s"Got $n!")).forever.fork
  _     <- queue.offer(1)
```

```
  _     <- queue.offer(2)
} yield ()
// res1: ZIO[console.package.Console, Nothing, Unit] = zio.ZIO$FlatMap@70faee2a
```

Normally printing something forever would result in our console being filled with output. But here the only thing that will be printed to the console is `Got 1` and `Got 2` because those were the only values offered the queue. There will also not be any threads blocked or polling done here. Instead the fiber that is taking values from the queue will simply suspend until there is another value in the queue to take.

Another important property of queues, like all the data structures in ZIO, is that they are safe for concurrent access by multiple fibers. This is key to using queues for work distribution.

```
import zio.clock._
import zio.console._
import zio.duration._

def work(identifier: String)(n: Int): URIO[Clock with Console, Unit] =
  console.putStrLn(s"fiber $identifier starting some expensive work with input $n") *>
    ZIO.sleep(1.second) *>
    console.putStrLn(s"fiber $identifier finished with input $n")

for {
  queue <- Queue.unbounded[Int]
  _     <- queue.take.flatMap(work("left")).forever.fork
  _     <- queue.take.flatMap(work("right")).forever.fork
  _     <- ZIO.foreach_(1 to 10)(queue.offer)
} yield ()
// res2: ZIO[Clock with Console, Nothing, Unit] = zio.ZIO$FlatMap@eeedcd7
```

Here we fork two workers that will each take values from the `Queue` and perform expensive computations with those values. We then offer a collection of values to the `Queue`. The `Queue` is safe for concurrent access, so there is no risk that two workers will take the same value. Thus, we can use the `Queue` to distribute work among the two workers. Each fiber will take a value from the `Queue`, perform work on it, and then take another and work on it as long as there are values in the `Queue`.

In the example above we used the `Queue` to distribute work among only two workers but we could use the `Queue` to distribute work among an arbitrary number of workers, and in fact that is how several of the combinators in ZIO that use bounded parallelism are implemented.

12.3 Varieties Of Queues

Unlike `Ref` and `Promise`, there are several different varieties of queues.

The first distinction to make is whether a queue is **unbounded** or **bounded**

An unbounded queue has no maximum capacity. We can keep offering elements to the queue and even if no elements are being removed from the queue we can continue adding elements to the end of the queue. Unbounded queues are simpler, which is why we used them in the examples above. In cases where relatively few values will be offered to the queue or where it is the responsibility of the user to take values from the queue at an appropriate rate this can be acceptable.

However, for most production applications an unbounded `Queue` creates a risk of a memory leak. For example, if we are using a `Queue` as a buffer in a streaming application, if the upstream producer is running faster than the downstream consumer then values will continue accumulating in the `Queue` until it consumes all available memory, causing the system to crash. More generally this can happen in any situation where we are using a queue to distribute work and new units of work are being added to the queue faster than the workers can process them.

The solution to this problem is to use a bounded queue. A bounded queue has some definite maximum number of elements it can contain, called its **capacity**. A bounded queue only has a finite capacity so it cannot grow without limit like an unbounded queue, solving the memory leak problem. However, it raises the question of what to do when a caller attempts to `offer` a value to a `Queue` that is already at capacity. ZIO offers three strategies for this:

1. Back Pressure
2. Sliding
3. Dropping

12.3.1 Back Pressure Strategy

The first strategy for when a caller attempts to offer a value to a `Queue` that is already full is to apply **back pressure**. This means that `offer` will semantically block until there is capacity in the queue for that element.

Back pressure has the advantage that it ensures no information is lost. Assuming that values are eventually taken from the `Queue`, every value offered to the `Queue` will ultimately be taken. It also has a certain symmetry. Fibers taking values from the queue semantically block until there are elements in the queue and fibers offering values to the queue semantically block until there is capacity in the queue. This can prevent fibers offering values from the queue from doing further work to offer even more values to the `Queue` when it is already at capacity.

For these reasons, the `BackPressure` strategy is the default one when you create

a `Queue` using the `bounded` constructor.

```
for {
  queue <- Queue.bounded[String](2)
  _     <- queue.offer("ping").tap(_ => console.putStrLn("ping")).forever.fork
} yield ()
// res3: ZIO[Console, Nothing, Unit] = zio.ZIO$FlatMap@3d665ea3
```

We construct a bounded `Queue` using the `bounded` constructor and specifying the maximum capacity. We can see that even though the forked fiber offers new values to the queue continuously, `ping` is only printed to the console twice because the fiber suspends when it attempts to offer a value a third time and the `Queue` is full.

While an excellent default, the `BackPressure` strategy does have limitations. With the `BackPressure` strategy, the downstream consumers can't begin processing the most recent values until they have processed all the values previously offered to the `Queue`.

For example, consider a financial application where stock prices are being offered to the queue and workers are taking those prices and performing analytics based on them. If new prices are being offered faster than the workers can consume then the upstream process will suspend on offering new prices until there is capacity in the queue and the workers will continue doing analytics based on older stock prices, which may be stale. The other two strategies present different approaches for dealing with this.

12.3.2 Sliding Strategy

The second strategy for when a caller attempts to offer a value to a `Queue` that is already full is to immediately add that value to the `Queue` and to drop the first element in the `Queue`. This is called the **sliding** strategy because it essentially "slides" the queue, dropping the first element and adding a new element at the end.

The sliding strategy makes sense when we don't want to back pressure to avoid staleness and the newest values in the `Queue` are in some sense "more valuable" than the oldest values in the `Queue`. For example, in the financial application discussed above the newest stock prices are potentially more valuable than the oldest ones because they reflect more timely market data.

We can create a sliding queue using the `sliding` constructor:

```
for {
  queue <- Queue.sliding[Int](2)
  _     <- ZIO.foreach(List(1, 2, 3))(queue.offer)
  a     <- queue.take
  b     <- queue.take
```

```
} yield (a, b)
// res4: ZIO[Any, Nothing, (Int, Int)] = zio.ZIO$FlatMap@17c4013f
```

Here notice that despite us offering three values to a `Queue` that only had a capacity of two, this program did not suspend on offering values to the `Queue`. Instead, the first element, `1`, will be dropped, so when `take` is called the values `2` and `3` will be returned from the `Queue`.

12.3.3 Dropping Strategy

The final strategy for handling adding elements to a `Queue` that is already full is the `Dropping` strategy. The dropping strategy is like the `Sliding` strategy in that `offer` always immediately returns, but now if the `Queue` is already at capacity the value will simply be dropped and not added to the `Queue` at all. The dropping strategy can make sense for when we want to maintain some distribution of values.

For example, the values offered to the `Queue` might be weather readings. We would like to have a reasonable distribution of temperature readings throughout the day for long term analysis and don't care about having the most recent readings right now. In this case, a dropping `Queue` could be a reasonable solution. Since the `Queue` is already at capacity we are already generating weather readings faster than they can be consumed. If we drop this one, another weather reading will be available soon anyway and hopefully there will be capacity in the `Queue` then.

We can create a dropping `Queue` with the `dropping` constructor:

```
for {
  queue <- Queue.dropping[Int](2)
  _     <- ZIO.foreach(List(1, 2, 3))(queue.offer)
  a     <- queue.take
  b     <- queue.take
} yield (a, b)
// res5: ZIO[Any, Nothing, (Int, Int)] = zio.ZIO$FlatMap@2b949fa2
```

This is the same example as before except this time we created a dropping `Queue` instead of a sliding one. Once again `offer` will return immediately despite the `Queue` already being at capacity. But this time the value being offered will be dropped, so when we take two values from the `Queue` we get `1` and `2`.

12.4 Other Combinators On Queues

Now that we have seen different varieties of queues we are in a better position to review some of the other combinators defined on queues.

We already have seen `take` many times over the course of this chapter, and we have seen that it is one combinator whose semantics are the same regardless of the

variety of `Queue`. It always takes the first element from the `Queue`, semantically blocking until at least one element is available.

We have seen that `offer`, on the other hand, has different semantics depending on the variety of `Queue`. For unbounded queues it will return immediately and simply add the value to the end of the `Queue`. For bounded queues with back pressure it will suspend until there is capacity in the queue. For sliding queues it will return immediately, adding the value to the end of the `Queue` and dropping the first element in the `Queue`. For dropping queues it will return immediately but not add the value to the `Queue` at all if there is not capacity for it.

We are also now in a better position to understand the return type of `UIO[Boolean]` in the signature of `offer`. The `Boolean` indicates whether the value was successfully added to the `Queue`, similar to how the `Boolean` in the return type of the various ways of completing a `Promise` indicated whether the `Promise` was completed with that value. For unbounded, back pressure, and sliding queues `offer` will always return `true` because when `offer` completes execution the value will always have been successfully offered to the `Queue`. In contrast, with dropping queues `offer` could return `false` if there is not capacity in the `Queue`, allowing the caller to potentially take further action with the value.

These are the most basic combinators on queues, but there are a variety of other ones either for convenience or particular use cases.

12.4.1 Variants Of Offer And Take

```
trait Queue[A] {
  def offerAll(as: Iterable[A]): UIO[Boolean]
  def poll: UIO[Option[A]]
  def takeAll: UIO[List[A]]
  def takeUpTo(max: Int): UIO[List[A]]
  def takeBetween(min: Int, max: Int): UIO[List[A]]
}
```

First, there are several variants of `offer` and `take` for dealing with multiple values.

`offerAll` offers all of the values in the specified collection to the `Queue` in order. If there is not sufficient capacity in a bounded `Queue` with back pressure then this will immediately offer the values there is capacity for to the queue and then suspend until space is available in the queue, repeating this process until all values have been offered to the `Queue`.

`poll` is analogous to the `poll` method on `Promise` or `Fiber` and allows us to tentatively observe whether there is a value in the `Queue`, taking and returning it if one exists or returning `None` otherwise.

`takeAll` takes all of the values in the `Queue`. This method will return immediately,

potentially with an empty `List` if there are no values in the `Queue`. `takeUpTo` is like `takeAll` except that it allows specified a maximum number of values to take. Like `takeAll`, it will return immediately, potentially with an empty list if the `Queue` is empty. Finally `takeBetween` allows specifying a minimum and maximum. If the number of elements in the `Queue` is less than the minimum it will take the available values and then wait to take the remaining values, suspending until at least the minimum number of values have been taken.

These methods can be helpful in certain situations or to improve the ergonomics of offering or taking multiple values, but generally if you have a good understanding of `offer` and `take` and the different varieties of queues you should have a good understanding of these combinators.

12.4.2 Metrics On Queues

```
trait Queue[A] {
  def capacity: Int
  def size: UIO[Int]
}
```

A couple of methods are available to introspect on the status of the `Queue`.

`capacity` returns the maximum capacity of the `Queue`. For an unbounded `Queue`, this will return `Int.MaxValue` (technically unbounded queues are implementing as dropping queues with capacity equal to `Int.MaxValue`). Notice that `capacity` returns an `Int` that is not wrapped in an effect because the capacity of a `Queue` is a constant value that is specified when the `Queue` is created and never changes.

`size` returns the number of values currently in the `Queue`. This returns a `UIO[Int]` because the size of the `Queue` changes over time as values are offered and taken.

12.4.3 Shutting Down Queues

There are also several methods defined on `Queue` dealing with shutting down queues. Because there may be multiple fibers waiting to either offer values or take values, we want to have a way to interrupt those fibers if the `Queue` is no longer needed. `shutdown` give us that tool.

```
trait Queue[A] {
  def awaitShutDown: UIO[Unit]
  def isShutdown: UIO[Boolean]
  def shutdown: UIO[Unit]
}
```

`shutdown` shuts down a `Queue` and immediately interrupts all fibers that are suspended on offering or taking values. `isShutDown` returns whether the `Queue` has currently been shut down, and `awaitShutDown` returns an effect that suspends until the `Queue` is shutdown.

When creating a `Queue`, it is good practice to shut down the `Queue` when you are done with it. We will see in the next section on resource handling some easy ways to do that.

12.5 Polymorphic Queues

As with `Ref` and `ZRef`, `Queue` is actually a type alias for a significantly more powerful data type called `ZQueue`:

```
trait ZQueue[-RA, -RB, +EA, +EB, -A, +B] {
  def offer(a: A): ZIO[RA, EA, Boolean]
  def take: ZIO[RB, EB, B]
}

type Queue[A] = ZQueue[Any, Any, Nothing, Nothing, A, A]
```

A `ZQueue` is more general than a `Queue` in two important ways.

First, the input types and output types of the `ZQueue` can be different. For example, a `ZQueue` might allow us to `offer` values of type `String` and `take` values of type `WeatherData`. This indicates that the `ZQueue` is doing some kind of mapping internally, transforming values of type `String` into values of type `WeatherData` by parsing them. A `ZQueue` can even filter values, perhaps discarding some `String` values that are malformed and cannot be parsed into valid instances of `WeatherData`.

Second, a `ZQueue` allows performing effects as part of both the `offer` and `take` operations. For example, the `ZQueue` might automatically log all `String` values received to one database and log all `WeatherData` values taken to another database.

In most cases we could perform these effects separately by applying them to the inputs and the outputs of the `Queue` ourselves. But bundling these behaviors into a single data structure can be extremely powerful, for example describing a `Queue` that we can offer `String` data to and allows us to take validated `WeatherData`, automatically filtering out invalid data and logging inputs and outputs.

Despite their polymorphism, all `ZQueue` instances are either unbounded queues, bounded queues with back pressure, sliding queues, or dropping queues, so everything you learned above about queues still applies to `ZQueue`. In fact, the way we typically create more polymorphic queues is by first constructing a `Queue` and then transforming it.

Here are some operations we can use for working with `ZQueue`.

12.5.1 Transforming Outputs

The first set of operations for working with `ZQueue` allow us to transform the inputs and outputs of the queue.

```
trait ZQueue[-RA, -RB, +EA, +EB, -A, +B] {
  def map[C](f: B => C): ZQueue[RA, RB, EA, EB, A, C]
  def mapM[RC <: RB, EC >: EB, C](f: B => ZIO[RC, EC, C]): ZQueue[RA, RC, EA, EC, A, C]
}
```

These combinators allow transforming each value taken from the `ZQueue` with the specified function. The function can either be a pure function as in `map` or an effectual function that potentially requires an environment and fails with an error as in `mapM`.

As an example, let's see how we can create a simple polymorphic queue that accepts `String` input and outputs the length of each `String`:

```
for {
  queue <- Queue.unbounded[String].map(_.map(_.length))
  _     <- queue.offer("Hello, ZQueue!")
  n     <- queue.take
} yield n
// res6: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@867d8fb
```

We initially created a `Queue` of `String` values and then we used `map` on it to transform the output type, resulting in a `ZQueue` that accepted `String` inputs and generated `Int` outputs. We saw this in action by offering a `String` to the queue and getting back its length when we took a value from the queue.

12.5.2 Transforming Inputs

We can transform inputs in addition to outputs. We do this with the `contramap` and `contramapM` combinators.

```
trait ZQueue[-RA, -RB, +EA, +EB, -A, +B] {
  def contramap[A1](f: A1 => A): ZQueue[RA, RB, EA, EB, A1, B]
  def contramapM[RA1 <: RA, EA1 >: EA, A1](f: A1 => ZIO[RA1, RA1, A]): ZQueue[RA1, RB, EA1,
}
```

Here is an example:

```
for {
  queue <- Queue.unbounded[Int].map(_.contramap[List[String]](_.length))
  _     <- queue.offer(List("Hello", "ZQueue"))
  n     <- queue.take
} yield n
// res7: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@3e34ebaf
```

This time we created a `Queue` of `Int` values and then used `contramap` to change the input type to `List[String]`. Notice here that we had to provide a function of type `List[String] => Int`. Since our queue already knows how to accept inputs of type `Int`, we need to provide a function that transforms the new input type into the input type we already know how to handle. This reflects a typically

pattern of arrows going "the opposite way" when dealing with contravariant types as opposed to covariant ones.

If you want to transform the input and output types of a `ZQueue` at the same time you can always use the `dimap` and `dimapM` combinators, but conceptually these are just the same as using `map` and `contramap` or `mapM` and `contramapM`.

12.5.3 Filtering Inputs and Outputs

So far, all of the examples we have seen have transformed the inputs one for one into outputs. But that doesn't have to be the case. We can filter out some inputs entirely.

```
trait ZQueue[-RA, -RB, +EA, +EB, -A, +B] {
  def filterInput[A1 <: A](f: A1 => Boolean): ZQueue[RA, RB, EA, EB, A1, B]
  def filterInputM[RA1 <: RA, EA1 >: EA, A1 <: A](f: A1 => ZIO[RA1, EA1, Boolean]): ZQueue[R
}
```

These combinators allow us to apply some predicate to each value offered to the `ZQueue`. If the predicate is satisfied the value is placed in the queue, subject to all the semantics we discussed above regarding different varieties of queues. If the predicate is not satisfied, the element is discarded and is not included in the queue. This allows us to implement `ZQueue` instances that perform some validation on their inputs, like the queue for `WeatherData` discussed above that accepted raw `String` input and transformed it into validated `WeatherData`, discarding malformed strings.

Here is a simple example of this functionality in action:

```
for {
  queue <- Queue.unbounded[Int].map(_.filterInput[Int](_ % 2 == 0))
  _     <- queue.offerAll(List(1, 2, 3, 4, 5))
  evens <- queue.takeAll
} yield evens
// res8: ZIO[Any, Nothing, List[Int]] = zio.ZIO$FlatMap@5cfd3543
```

This queue filters all of the values offered to it, only keeping the even values. Thus, when we call `offerAll` with `List(1, 2, 3, 4, 5)` and then call `takeAll`, we get back `List(2, 4)`.

12.5.4 Combining Queues

The final thing we can do with `ZQueue` is create a queue that is the composite of two other queues using the `both`, `bothWith`, and `bothWithM` combinators. Offering a value to this composite queue will offer the value to both of the underlying queues. Taking a value from the composite queue will take a value from each of the underlying queues, either combining the results into a tuple with `both` or combining them using a user provided function in `bothWith` or `bothWithM`.

This can be used to implement "fan in" and "fan out" logic. For example, we could create a queue that broadcasts all values offered to two other queues that each perform their own logic.

```
import zio._
import zio.console._

final case class Name(first: String, last: String)

def leftWorker(queue: Queue[Name]): URIO[Console, Unit] =
  queue.take.flatMap { name =>
    console.putStrLn(s"Got first name ${name.first}")
  }.forever.fork.unit

def rightWorker(queue: Queue[Name]): URIO[Console, Unit] =
  queue.take.flatMap { name =>
    console.putStrLn(s"Got last name ${name.last}")
  }.forever.fork.unit

for {
  left  <- Queue.unbounded[Name]
  right <- Queue.unbounded[Name]
  both  =  left && right
  _     <- leftWorker(left)
  _     <- rightWorker(right)
  _     <- both.offer(Name("Jane", "Doe"))
} yield ()
// res10: ZIO[Console, Nothing, Unit] = zio.ZIO$FlatMap@759f5809
```

Each time we offer a value to the composite queue it will be broadcast to both the left and right queues. In this case they just printed debug statements, but each of those composite queues could have its own logic associated with it and could potentially have its own logic for filtering and transforming values.

A `ZQueue` can also be used to model "fan in" behavior where the values from multiple queues are combined into a single queue.

```
import zio.duration._
import zio.random._

final case class StockQuote(ticker: String, price: Double)

def offerStockPrices(ticker: String)(initial: Double)(queue: Queue[StockQuote]): URIO[Rando
  random.nextDouble.flatMap { n =>
    queue.offer(StockQuote(ticker, initial)) *>
      offerStockPrices(ticker)(initial + n - 0.5)(queue)
  }
```

```
def render(quotes: (StockQuote, StockQuote)): URIO[Console, Unit] =
  console.putStrLn(quotes.toString)

for {
  left  <- Queue.sliding[StockQuote](1)
  right <- Queue.sliding[StockQuote](1)
  both  =  left && right
  _     <- offerStockPrices("AAPL")(450.0)(left).delay(50.milliseconds).fork
  _     <- offerStockPrices("GOOG")(1500.0)(right).delay(75.milliseconds).fork
  _     <- both.take.flatMap(render).delay(100.milliseconds).forever
} yield ()
// res11: ZIO[Random with clock.package.Clock with Console with clock.package.Clock, Nothing
```

In this program, we create two sliding queues that both have a capacity of one. In essence they will always contain the most recent value offered to the queue and each time a new value is offered it will replace the current value in the queue. We then fork two processes that will simulate a push based interface that provides stock price data. Each time we pull from the combined queue we pull the most recent value from each queue and print them to the console.

If you are writing a lot of code like this you will probably be better off switching to ZIO Stream, which supports this functionality in a richer, more declarative style. But queues are an important part of streaming solutions in buffering between different parts of an application and even if you are not using streaming it is good to know you have this power under the hood.

12.6 Conclusion

In this chapter we took an in depth look at `Queue` and its more cousin `ZQueue`. In the process we went from the basics of offering and taking values to different varieties of queues to learning how we can construct queues that transform and filter values and even combine other queues for complex "fan out" and "fan in" behavior.

This chapter was a long one but if you focus on how queues can be used to distribute work between multiple fibers you should be in good shape. While this chapter was a long one befitting all the features that `ZQueue` brings to the table the next chapter on `Semaphore` will be a short one so you are almost done with your journey through ZIO's core concurrent data structures!

12.7 Exercises

Chapter 13

Concurrent Structures: Semaphore - Work Limiting

Most of the time we want to make our applications as concurrent as possible to maximize performance. But sometimes with ZIO's fiber based concurrency model we can be doing too many things at the same time! One of our internal services is receiving too many simultaneous requests now that we are doing as many things as possible in parallel and never blocking. How do we limit work when we need to so that we don't overwhelm other systems?

13.1 Interface Of A Semaphore

This is where `Semaphore` comes in. Despite its somewhat fancy sounding name, `Semaphore` is actually a very straightforward data structure. A `Semaphore` is created with a certain number of "permits" and is defined in terms of a single method, `withPermits`.

```
import zio._

trait Semaphore {
  def withPermits[R, E, A](n: Long)(task: ZIO[R, E, A]): ZIO[R, E, A]
}
```

The guarantee of `withPermits` is simple. Any fiber calling `withPermits` must first "acquire" the specified number of permits. If that number of permits or more are available, the number of available permits is decremented and the fiber can proceed with executing `task`. If that number of permits are not available, the fiber suspends until the specified number of permits are available. Acquired permits will be "released" and the number of available permits incremented as soon as the task finishes execution, whether it is successful, fails, or is interrupted.

A very common case is where fibers acquire a single permit and there is a common shorthand for this:

```
trait Semaphore {
  def withPermits[R, E, A](n: Long)(task: ZIO[R, E, A]): ZIO[R, E, A]
  def withPermit[R, E, A](task: ZIO[R, E, A]): ZIO[R, E, A] =
    withPermits(1)(task)
}
```

Conceptually you can think of a Semaphore as a parking lot operator, making sure there are only so many cars in the lot at the same time. The parking lot operator knows there are a certain number of spaces. Every time a car tries to enter the lot, the operator checks if there are available spaces. If so they let the car in. If not the car has to wait outside until another car exits the lot to make room. The guarantee that the parking lot operator provides is that there are never more cars in the lot than the available spaces.

Similarly, the guarantee that the `Semaphore` provides if each fiber takes one permit is that there are never more than a specified number of fibers executing a block of code protected by a `Semaphore`. This provides an extremely easy way for us to limit the degree of concurrency in part of our application without changing anything else about our code.

13.2 Using Semaphores To Limit Parallelism

Let's see an example of this in action.

```
import zio._
import zio.clock._
import zio.console._
import zio.duration._

def queryDatabase(connections: Ref[Int]): URIO[Console with Clock, Unit] =
  connections.updateAndGet(_ + 1).flatMap { n =>
    console.putStrLn(s"Aquiring connection, now $n simultaneous connections") *>
    ZIO.sleep(1.second) *>
    console.putStrLn(s"Closing connection, now ${n - 1} simultaneous connections")
  }

for {
  ref <- Ref.make(0)
  semaphore <- Semaphore.make(4)
  query = semaphore.withPermit(queryDatabase(ref))
  _ <- ZIO.foreachPar_(1 to 10)(_ => query)
} yield ()
// res1: ZIO[Console with Clock, Nothing, Unit] = zio.ZIO$FlatMap@4cd2ef5e
```

Without the `Semaphore`, `foreachPar_` would cause all ten queries to execute simultaneously. So we would see all ten connections acquired and then all ten connections released.

With the `Semaphore`, the first four queries will acquire one permit each and immediately begin executing, but the other six queries will find that no permits are available any longer and will suspend until permits become available. Once any of the first queries completes execution the permit it acquired will be returned to the `Semaphore` and another query will immediately begin execution. So we will see that there are never more that four simultaneous connections, and that there are always four active queries excluding brief periods between one query completing and the next query beginning execution.

Notice what we have done here. With essentially two lines of code we have limited the degree of parallelism in part of our application to an arbitrary maximum concurrency. We have done it without any threads ever blocking. We have done it without ever polling. We have done it in a way that is highly efficient where as soon as one query finishes another begins execution, rather than having to wait for the first four queries to finish before beginning the next four.

We have also done it in a way where we cannot ever leak permits as is possible in some other frameworks. In other frameworks we could acquire a permit and then either forget to return it or possibly not be able to return it because of an error or interruption. This could cause the `Semaphore` to "leak" permits and result in a deadlock if there are no more permits even though there are no more active workers. This can't ever happen in ZIO as long as we use `withPermits`.

13.3 Using Semaphore To Implement Operators

This pattern of using a `Semaphore` to limit the degree of parallelism with `foreachPar` is so common that there is a separate combinator for it. It is actually one we saw before in the very first chapter, `foreachParN`!

```
object ZIO {
  def foreachPar[R, E, A, B](as: Iterable[A])(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
    ???
  def foreachParN[R, E, A, B](n: Int)(as: Iterable[A])(f: A => ZIO[R, E, B]): ZIO[R, E, List
    for {
      semaphore <- Semaphore.make(n)
      bs        <- ZIO.foreachPar(as)(a => semaphore.withPermit(f(a)))
    } yield bs
}
```

In many cases you can just use `foreachParN`, but now you know how it works and how you can use `Semaphore` directly in more advanced use cases where you want to limit the degree of parallelism and aren't just using `foreachPar`.

13.4 Using Semaphore To Make A Data Structure Safe For Concurrent Access

Another common use case is a `Semaphore` with a single permit. In this case the `Semaphore` ensures that only a single fiber can be executing a block of code guarded by a `Semaphore`. In this way a `Semaphore` acts like a lock or a synchronized block in traditional concurrent programming but without ever blocking any underlying threads.

This can be useful when we need to expose a data structure that is not otherwise safe for concurrent access. For example, we talked about `RefM` in the chapter on references but were not ready at the time to discuss how `RefM` was implemented. Now we are in a position to do so.

Recall that `RefM` is a reference that allows performing effects in the `modify` function.

```
trait RefM[A] {
  def modify[R, E, B](f: A => ZIO[R, E, (B, A)]): ZIO[R, E, B]
}
```

We could try to naively implement `RefM` in terms of `Ref` like this:

```
object RefM {
  def make[A](a: A): UIO[RefM[A]] =
    Ref.make(a).map { ref =>
      new RefM[A] {
        def modify[R, E, B](f: A => ZIO[R, E, (B, A)]): ZIO[R, E, B] =
          for {
            a <- ref.get
            v <- f(a)
            _ <- ref.set(v._2)
          } yield v._1
      }
    }
}
```

But this won't work. When implemented this way the `modify` operation is not atomic because we `get` the `Ref`, then perform the effect, and finally `set` the `Ref`. So if two fibers are concurrently modifying the `RefM` they could both get the same value instead of one getting and setting and then the other getting the resulting value. This results in lost updates and an implementation that is not actually safe for concurrent access.

How do we get around this? `Ref#modify` expects a function that returns a modified value, not a function returning an effect producing a modified value, the entire problem we were trying to solve in the first place. We can't implement it in terms of the underlying `AtomicReference` either since that is also expecting a value instead of an effect. So what do we do? Are we stuck?

No! This is where `Semaphore` comes in. We can guard the access to the underlying `Ref` with a `Semaphore` so that only one fiber can be in the block of code getting and setting the `Ref` at the same time. This way, if two fibers attempt to modify the `RefM` concurrently the first will get the value, perform the effect, and set the value. The second fiber will suspend because there are no more available permits. Once the first fiber finishes setting the value it will return its permit, which will allow the second fiber to proceed with getting the updated value, performing its effect, and setting the value with its result. This is exactly the behavior we want.

To accomplish this all we have to do is create a `Semaphore` with a single permit in addition to the `Ref` and wrap the entire body of the `modify` method in `withPermit`.

```
object RefM {
  def make[A](a: A): UIO[RefM[A]] =
    for {
      ref       <- Ref.make(a)
      semaphore <- Semaphore.make(1)
    } yield new RefM[A] {
      def modify[R, E, B](f: A => ZIO[R, E, (B, A)]): ZIO[R, E, B] =
        semaphore.withPermit {
          for {
            a <- ref.get
            v <- f(a)
            _ <- ref.set(v._2)
          } yield v._1
        }
    }
}
```

Congratulations! You have implemented your own concurrent data structure.

`Semaphore` is a good example of a data structure that does one thing and does it exceedingly well. It doesn't solve the same variety of problems that some of the more polymorphic concurrent data structures like `Ref`, `Promise`, and `Queue` can. But when you need to deal with a problem regarding limiting how many fibers are doing some work at the same time it lets you do that in an extremely easy and safe way and should be your go to.

With this, we have finished our discussion of the core concurrent data structure in ZIO. In the next section we will discuss resource handling, an important concern in any long running application and one that is made more challenging with concurrency. We will see how ZIO provides a comprehensive solution to resource handling from ensuring that individual resources, once acquired, will be safely released, to composing multiple resources together and dealing with even more advanced resource management scenarios.

13.5 Conclusion

13.6 Exercises

Chapter 14

Resource Handling: Bracket - Safe Resource Handling

This chapter begins our discussion of ZIO's support for safe resource handling.

Safe resource handling is critical for any long running application. Whenever we acquire a resource, whether that is a file handle, a socket connection, or something else, we need to ensure that it is released when we are done with it. If we fail to do this, even a small fraction of the time, we will "leak" resources. In an application that is performing operations many times a second and running for any significant period of time such a leak will continue until it eventually depletes all available resources, resulting in a catastrophic system failure.

However, it can be easy to accidentally forget to release a resource or to create a situation in which a resource may not be released in every situation. This can be especially true when we are dealing with other challenging issues specific to our domain. It can also be true when dealing with asynchronous or concurrent code where it can be difficult to visualize all possible execution paths of a program. Unfortunately this describes most of the problems we are trying to solve!

So we need tools that provide strong guarantees that when a resource is acquired it will always be released and allow us to write code without worrying that we will accidentally leak resources. `bracket` and `ZManaged`, discussed in the next chapter, are ZIO's solutions to do this.

14.1 Inadequacy Of Try And Finally In The Face Of Asynchronous Code

The traditional solution to the problem of safe resource handling is the `try ... finally` construct.

```
trait Resource

lazy val acquire: Resource = ???
def use(resource: Resource): Unit = ???
def release(resource: Resource): Unit = ???

lazy val example = {
  val resource = acquire
  try {
    use(resource)
  } finally {
    release(resource)
  }
}
```

If the resource is successfully acquired in the first line then we immediately begin executing the `try ... finally` block. This guarantees that `close` will be called whether `use` completes successfully or returns an exception.

This is sufficient for code that is is fully synchronous. But it starts to break down when we introduce asynchrony and concurrency, as almost all modern applications do. Let's see what happens when we start to translate this to use `Future`. We will have to create our own variant of `try ... finally` that works in the `Future` context, let's call it `ensuring`.

```
import scala.concurrent.{ ExecutionContext, Future }
import scala.util.{ Failure, Success }

trait Resource

def acquire: Future[Resource] = ???
def use(resource: Resource): Future[Unit] = ???
def release(resource: Resource): Future[Unit] = ???

implicit final class FutureSyntax[+A](future: Future[A]) {
  def ensuring(
    finalizer: Future[Any]
  )(implicit ec: ExecutionContext): Future[A] =
    future.transformWith {
      case Success(a) => finalizer.flatMap(_ => Future.successful(a))
      case Failure(e) => finalizer.flatMap(_ => Future.failed(e))
    }
}

implicit val global = scala.concurrent.ExecutionContext.global
// global: concurrent.ExecutionContextExecutor = scala.concurrent.impl.ExecutionContextImpl
lazy val example = acquire.flatMap(resource => use(resource).ensuring(release(resource)))
```

We added a new method to `Future` called `ensuring`. `ensuring` waits for the original `Future` to complete, then runs the specified finalizer, and finally returns the result of the original `Future`, whether it is a `Success` or a `Failure`. Note that if the `finalizer` throws an exception the original exception will be lost. See the discussion of `Cause` in the chapter on error handling for further discussion of this.

This seems okay at first glance. The code is relatively terse and it does ensure `finalizer` will run whether `use` succeeds or fails.

The problem with this is around interruption. In general in writing asynchronous and concurrent code we have to assume that we could get interrupted at any point and in particular that we could get interrupted "between" effects that are composed together. What happens if we are interrupted after `acquire` completes execution but before `use` begins execution? This would be very common if `acquire` was not interruptible and we were interrupted while we were acquiring the resource, because then we would check for interruption immediately after that. In that case, we would never execute `use` because we were interrupted which also means that `finalizer` would never run. So we would have acquired the resource without releasing it, creating a leak.

For `Future` this isn't a problem because `Future` doesn't support interruption at all! But this creates a separate resource problem of its own. Since we have no way of interrupting a `Future` we have no way to stop doing work that no longer needs to be performed, for example if a user navigates to a web page, queries some information, and closes the browser. This can itself create a resource management issue.

14.2 Bracket As A Generalization Of Try And Finally

The solution is ZIO's `bracket` combinator. The problem with `try ... finally` and the `ensuring` solution we developed above is that they only operate on part of the resource lifecycle, the use and release of the resource. But in the face of interruption we can't operate on only part of the lifecycle because interruption might occur between the acquisition and the use / release of the resource. Instead, we need to look at all phases of the resource lifecycle together. `bracket` does just that.

```
import zio._

object ZIO {
  def bracket[R, E, A, B](
    acquire: ZIO[R, E, A],
    release: A => URIO[R, Any],
    use: A => ZIO[R, E, B]
```

```
  ): ZIO[R, E, B] = ???
}
```

Let's look at this type signature in more detail.

`acquire` is an effect that describes the acquisition of some resource `A`, for example a file handle. Acquiring the resource could fail, for example if the file does not exist. It could also require some environment, such as a file system service.

`use` is a function that describes using the resource to produce some result `B`. For example, we might read all of the lines from the file into memory as a collection of `String` values. Once again, the `use` action could fail and could require some environment.

`release` is a function that releases the resource. The type signature indicates that the release action cannot fail. This is because the error type of `bracket` represents the potential ways that the acquisition and use of the resource could fail. Generally finalizers should not fail and if they do that should be treated as a defect. The return type of the `release` action is `Any`, indicating that the release action is being performed purely for its effects (e.g. closing the file) rather than for its return value.

The `bracket` combinator offers the following guarantees:

1. The `acquire` action will be performed uninterruptibly
2. The `release` action will be performed uninterruptibly
3. If the `acquire` action successfully completes execution then the `release` action will be performed as soon as the `use` action completes execution, regardless of how `use` completes execution

These are exactly the guarantees we need for safe resource handling. We need the `acquire` action to be uninterruptible because otherwise we might be interrupted part way through acquiring the resource and be left in a state where we have acquired some part of the resource but do not release it, resulting in a leak. We need the `release` action to be uninterruptible because we need to release the resource no matter what, and in particular must release it even if we are being interrupted. The final guarantee is the one we need to solve the problem we saw before with `try ... finaly`. If the resource is successfully acquired then the release action will always be run. So even if we are interrupted before we begin using the resource it will still be released.

As long as we use `bracket` it is impossible for us to acquire a resource and finish using it without it being released.

In addition to this version of bracket, there is a curried version that can be more ergonomic and has better type inference. It looks like this:

```
ZIO.bracket(acquire)(release)(use)
```

The `use` action comes last here because there will typically be the most logic associated with using the resource, so this provides more convenient syntax.

```
ZIO.bracket(acquire)(release) { a =>
  ??? // do something with the resource here
}
```

There is also a version of `bracket` defined on the `ZIO` trait so you can also do this:

```
acquire.bracket(release)(use)
```

In addition to `bracket`, there is one more powerful variant to be aware of, `bracketExit`. `bracket` provides access to the resource in the `release` function, but it doesn't provide any indication of how the `use` effect completed. Most of the time this doesn't matter. Whether `use` completed successfully, failed, or was interrupted we still need to close the file handle or socket connection. But sometimes we may want to do something different depending on how the `use` action completes, either closing the resource a different way or performing some further actions.

```
object ZIO {
  def bracketExit[R, E, A, B](
    acquire: ZIO[R, E, A],
    release: (A, Exit[E, B]) => URIO[R, Any],
    use: A => ZIO[R, E, B]
  ): ZIO[R, E, B] =
    ???
}
```

The signature is the same except now `release` has access to both the resource `A` as well as an `Exit[E, B]` with the result of the `use` action. The caller can then pattern match on the `Exit` value to apply different release logic for success or failure or for different types of failures.

Just like with `bracket` there are curried versions of `bracketExit` and versions defined on both the `ZIO` trait and the `ZIO` companion object.

When using `bracket`, one best practice is to have `use` return the first value possible that does not require the resource being open. Remember that the `release` action runs when the `use` action is completed. So if you include the entire rest of your program logic in the body of `use` the resource will not be closed until the end of the application, which is longer than necessary. Releasing resources like this too late can itself result in a memory leak. Conversely, if you return an intermediate value that depends on the resource not being released, for example you return the resource itself from `bracket`, then the `release` action will execute immediately and you will likely get a runtime error because the resource has already been released.

For example, if you are opening a file to read its contents as a `String` and then do further analytics and visualization based on those contents, have the `use` action of `bracket` return a `String` with the file contents. That way the file can

be closed as soon as the contents have been read into memory and you can then proceed with the rest of your program logic.

14.3 The Ensuring Combinator

ZIO does also have an `ensuring` combinator like the one we sketched out at the beginning of the chapter for `Future`, but this one does handle interruption. The signature of `ensuring` is as follows:

```
trait ZIO[-R, +E, +A] {
  def ensuring[R1 <: R](finalizer: URIO[R1, Any]): ZIO[R1, E, A]
}
```

`ensuring` is like `bracket` without the resource. Its guarantee is that if the original effect begins execution, then the finalizer will always be run as soon as the original effect completes execution, either successfully, by failure, or by interruption. It allows us to add some finalization or cleanup action to any effect. For example, if we have an effect that is incrementing a timer perhaps we want to make sure to reset the timer to zero when the effect terminates, no matter what.

`ensuring` is very useful in a variety of situations where you want to add a finalizer to an effect, but one watch out is not to use `ensuring` when `bracket` is required. For example, the following code has a bug:

```
acquire.flatMap(resource => use(resource).ensuring(release))
```

Even with the ZIO version of `ensuring` this is not safe because execution could be interrupted between `acquire` completing execution and `use` beginning execution. The guarantee of `ensuring` is only that if the original effect begins execution, the finalizer will be run. If the original effect never begins execution then the finalizer will never run.

A good guideline is that if you are working with a resource, or anything that requires "allocation" in addition to "deallocation" use `bracket`. Otherwise use `ensuring`.

14.4 Conclusion

In this chapter we learned about `bracket`, the fundamental primitive in ZIO for safe resource usage. For situations where we need to use a single resource at a time, like opening a list of files to read their contents into memory, `bracket` is all we need and is a huge step forward in providing iron clad guarantees about safe resource usage. But what if we have to use multiple resources, for example to read from one file and write to another file. `ZManaged` takes resource handling to the next level by allowing us to describe a resource as its own data type and compose multiple resources together. It is what we are going to talk about next.

14.5 Exercises

Chapter 15

Resource Handling: Managed - Composable Resources

The `bracket` operator is the foundation of safe resource handling in ZIO. Ultimately, `bracket` is all we need, but composing multiple resources directly with `bracket` is not always the most ergonomic.

For example, say we have a file containing weather data.

We would like to use `bracket` to safely open the file containing weather data along with a second file that will contain results from our value added analysis. Then with both files open we would like to incrementally read data from the first file and write results to the second file, ensuring that both files are closed no matter what.

We could do that as follows:

```
import java.io.{ File, IOException }

import zio._

def openFile(name: String): IO[IOException, File] =
  ???

def closeFile(file: File): UIO[Unit] =
  ???

def withFile[A](name: String)(use: File => Task[A]): Task[A] =
  openFile(name).bracket(closeFile)(use)
```

```
def analyze(weatherData: File, results: File): Task[Unit] =
  ???

lazy val analyzeWeatherData: Task[Unit] =
  withFile("temperatures.txt") { weatherData =>
    withFile("results.txt") { results =>
      analyze(weatherData, results)
    }
  }
```

This will allow us to read from the weather data file and write to the results file at the same time while ensuring that both files will be closed no matter what.

But there are a couple issues with it.

First, it is not very composable.

In this case where we were only working with two resources it was fairly terse but what if we had a dozen different resources? Would we need to have that many layers of nesting?

Second, we have introduced a potential inefficiency.

The code above embodies a very specific ordering of handling these files. First we open the weather data file, then we open the results file, then we close the results file, and finally we close the weather data file.

But that specific ordering is not really necessary. For maximum efficiency we could open both the weather data and results files at the same time, transform the data, and then close both the weather data and results files at the same time.

In this case where we are just dealing with two local files acquiring and releasing the resources concurrently probably does not make a big difference, but in other cases it could make a huge difference.

More broadly, we are thinking about a lot of implementation details here. This code is very "imperative", saying to do this and do that, versus "declarative", letting us say what we want and not having to worry so much about the "how".

How do we recapture the simplicity we had in using `bracket` to deal with single resources in the case where we have multiple resources?

15.1 Managed As A Reification of Bracket

The first clue to recapturing the simplicity of `bracket` when working with multiple resources is the `withFile` operator we implemented above:

```
def withFile[A](name: String)(use: File => Task[A]): Task[A] =
  openFile(name).bracket(closeFile)(use)
```

This was a very natural operator for us to write in solving our problem and it reflects a logical separation of concerns.

One concern is the acquisition and release of the resource. What is required to acquire a particular resource and to safely release it is known by the *implementer* of the resource.

In this case the `withFile` operator and its author know what it means to open a file (as opposed to a web socket, a database connection, or any other kind of resource) and what it means to close the file. These concerns are very nicely wrapped up in the `withFile` operator so that the user of `withFile` does not have to know anything about opening or closing files but just what they want to do with the file.

The second concern is how to use the resource. Only the *caller* of the `withFile` operator knows what they want to do with the file.

The function `use` that describes the use of the resource could do almost anything, succeeding with some type `A`, failing with a `Throwable` in this case, and potentially performing arbitrary effects along the way. The implementer of the `withFile` operator has no idea how the resource will be used.

This reflects a very natural separation of concerns and so what we would conceptually like to do in the example above is take the resource described by `withFile("temperatures.txt")` and the resource described by `withFile("results.txt")` and combine them to create a new resource that describes the acquisition and release of both resources.

We can't do that as written because `withFile(name)` doesn't return a data type that can have its own methods. It is just a partially applied function.

But we can change that. We can "reify" the acquisition and release of a resource independently of how it is used into its own data type:

```
final case class ZManaged[-R, +E, A](
  acquire: ZIO[R, E, A],
  release: A => URIO[R, Any]
) {
  def use[R1 <: R, E1 >: E, B](f: A => ZIO[R1, E1, B]): ZIO[R1, E1, B] =
    acquire.bracket(release)(f)
}
```

`ZManaged` just wraps the `acquire` and `release` actions of `bracket` up into a data type that describes the concept of a resource independent of how the resource is used. This allows us to compose more complex resources out of simpler ones while deferring specifying exactly how we want to use the resource until we are ready to.

Note that this is not how `ZManaged` is actually implemented but it should give you a good intuition for the idea of capturing the acquisition and release of a resource as its own data type. In the next chapter on advanced uses of `ZManaged` we will learn more about the actual implementation of `ZManaged`.

Using our new `ZManaged` data type, we can describe a resource representing the safe acquisition and release of a file handle like this:

```
def file(name: String): ZManaged[Any, Throwable, File] =
  ZManaged(openFile(name), closeFile)
```

We can now use `file("temperatures.txt")` to describe the resource of the weather data file completely independently of how it is used. More importantly, we now have a data type for a resource that we can add methods to describing various ways we might want to compose resources.

Fortunately for us, ZIO contributors have added a wide variety of operators to describe different ways of composing and transforming resources covering both common use cases as well as much more advanced ones.

For example, just like `ZIO`, `ZManaged` has a `zipPar` operator that combines two resources to return a new resource that acquires and releases the original resources in parallel. Using this, we can solve the problem of reading from and writing to two files at the same time quite succinctly and correctly as:

```
import java.io.File

import zio._

def file(name: String): ZManaged[Any, Throwable, File] =
  ???

def analyze(weatherData: File, results: File): Task[Unit] =
  ???

lazy val weatherDataAndResults: ZManaged[Any, Throwable, (File, File)] =
  file("temperatures.txt").zipPar(file("results.txt"))

lazy val analyzeWeatherData: Task[Unit] =
  weatherDataAndResults.use { case (weatherData, results) =>
    analyze(weatherData, results)
  }
```

Now we have separated the description of the resources from how they are used and recovered the declarative style we have grown accustomed to when working with ZIO. We simply indicate that these are resources that can be acquired in parallel using `zipPar` and the implementation of `ZManaged` takes care of the rest.

15.2 Managed As A ZIO Effect With Additional Capabilities

Another helpful way to think about `ZManaged` is as a `ZIO` effect with additional structure. Just like a `ZIO` effect is a blueprint for a concurrent program that requires an environment `R` and may either fail with an `E` or succeed with an `A`, a `ZManaged` is a blueprint for a concurrent program that requires an environment `R`, may either fail with an `E` or succeed with an `A`, and requires requires some finalization.

As such, most of the operators defined on `ZIO` are also defined on `ZManaged`.

Perhaps the most important example of this is `flatMap`, which represents sequential composition of resources when one resource depends on another.

For example, we might need to open a network connection and then use the network connection to open a database connection on a remote server. The additional structure `ZManaged` adds is that in addition to the resources being acquired sequentially they will be released in reverse order of acquisition.

In the example above, the database connection must be closed before the network connection. Otherwise if the network connection is closed first we will have no ability to communicate with the remote server to close the database connection!

We can use `zip` and its variants to describe sequential acquisition and release of resources when those resources do not depend on each other. Similarly, we can describe sequentially acquiring and releasing many resources using the `foreach` and `collectAll` operators.

For example, say that we had several files with weather data for different years and we wanted to read all of their contents into memory. We could do this extremely simply with the `foreach` operator on `ZManaged`:

```
import scala.io.Source

lazy val names: List[String] = ???

lazy val files: ZManaged[Any, Throwable, List[File]] =
  ZManaged.foreach(names)(file)

lazy val weatherData: Task[List[String]] =
  files.use { files =>
    Task.foreach(files) { file =>
      Task(Source.fromFile(file).getLines.toList)
    }.map(_.flatten)
  }
```

We can also acquire and release resources in parallel using `zipPar`, `foreachPar`, and `collectAllPar`. We can even do this with bounded parallelism with

`foreachParN` and `collectAllN`.

So we could read all of the weather data in parallel making sure to never have more than four files open at the same time by just replacing `files` in the code above with the following definition:

```
lazy val files: ZManaged[Any, Throwable, List[File]] =
  ZManaged.foreachParN(4)(names)(file)
```

This will now read weather data from the files in parallel, opening up to four files at a time, ensuring that all files are closed no matter what, and immediately stopping the computation and closing any remaining files if there is any failure in reading from a file. As you can hopefully see, `ZManaged` provides tremendous power for composing resources in a declarative way.

15.3 Constructing Managed Resources

Now that we understand what `ZManaged` is and how powerful it can be, let's look at how we can use `ZManaged` in our own code.

In this section we will focus on the different ways we can create `ZManaged` values. Then in the coming sections we will learn about more ways to transform `ZManaged` values to build more complex resources and how we can ultimately use those resources.

15.3.1 Fundamental Constructors

The most basic constructor of `ZManaged` values is `make`:

```
def make[R, E, A](acquire: ZIO[R, E, A])(release: A => URIO[R, Any]): ZManaged[R, E, A] =
  ???
```

This signature is exactly the same as the constructor for the simplified representation of `ZManaged` we discussed earlier in this chapter except that the arguments have been curried for improved type inference.

One thing to note here is that with the `make` constructor both the `acquire` and `release` actions will be *uninterruptible*. This makes sense because we said before that `ZManaged` was a reification of `bracket` and when we call `bracket` both the `acquire` and `release` actions are uninterruptible.

More conceptually, we need the finalizer to be uninterruptible because the finalizer is supposed to be run after the resource is finished being used, regardless of how the use of the resource terminates. So if the use of the resource is interrupted we need to be sure that the finalizer itself won't be interrupted!

It is also generally important for the `acquire` action to be uninterruptible to make sure we do not accidentally find ourselves in an undefined state.

If the `acquire` effect is successful then the `release` effect is guaranteed to be run to close the resource. But what would happen if the `acquire` action were to be interrupted?

We couldn't run the `release` action to close the resource because we wouldn't have a resource to close. But the `acquire` action might still have done some work requiring finalization, for example opening a network connection in order to open a database connection on a remote server.

For this reason, the default is that the `acquire` effects of `bracket` and `ZManaged` are uninterruptible. If you do not want to make the `acquire` effect uninterruptible there is a `makeInterruptible` variant that supports this.

```
def makeInterruptible[R, E, A](acquire: ZIO[R, E, A])(release: A => URIO[R, Any]): ZManaged
  ???
```

This is exactly like `make` except the interruptibility of the `acquire` effect will be unchanged. The `release` action will still be uninterruptible.

There is also a more complex variant of these constructors called `makeReserve`:

```
def makeReserve[R, E, A](reservation: ZIO[R, E, Reservation[R, E, A]]): ZManaged[R, E, A] =
  ???
```

A `Reservation` is much like our simple representation of a `ZManaged`:

```
final case class Reservation[-R, +E, +A](
  acquire: ZIO[R, E, A],
  release: Exit[Any, Any] => URIO[R, Any]
)
```

A `Reservation` describes an `acquire` and `release` action but here the `release` action takes an `Exit` value, allowing the finalization logic to depend on how the `acquire` action completes, whether by success, failure, or interruption.

The key thing to notice is that the `makeReserve` constructor accepts as an argument a `ZIO` effect that *returns* a `Reservation`.

This supports a more complex two step model of resource acquisition and usage:

1. **Reservation** - The `reservation` effect that returns the `Reservation` is *uninterruptible*.
2. **Acquisition** - The `acquire` effect of the `Reservation` is *interruptible*.

While there is nothing fundamental about this pattern of two step resource acquisition, it turns out to be quite useful in practice in more advanced resource acquisition scenarios.

A common pattern, particularly when working with concurrent data structures, is that when we want to semantically block for some condition to be true we first want to do some internal bookkeeping to "register" ourselves as waiting for some state (e.g. for a permit to be available from a `Semaphore`). Once we have

done this we need to suspend until we receive a signal that the condition is true and we can proceed.

In this scenario we need the first "bookkeeping" step to be uninterruptible because we need to avoid corruption of the internal state of the data structure if we are interrupted after having updated one piece of internal state but before updating another. Making this uninterruptible is also not problematic because updating these different pieces of internal state is generally very fast.

In contrast, we need the second "waiting" step to be interruptible. We want to support interruption while we are waiting, for example for a permit to be available, because we could be stuck waiting a long time, potentially forever.

So the `makeReserve` constructor turns out to be quite useful for more advanced resource acquisition scenarios and the simpler `make` and `makeInterruptible` constructors can be implemented quite straightforwardly in terms of `makeReserve`. However, if you don't need the full power of `makeReserve` stick with `make` and the other constructors discussed in this section.

There is also a `makeExit` constructor that is like `make` but lets you specify different logic for how the resource should be finalized depending on the result of the acquisition. This can be thought of as an intermediate between `make` and `makeReserve` in that it allows specifying different logic depending on how the `acquire` effect terminates but still embodies a single step resource acquisition process.

```
def makeExit[R, E, A](acquire: ZIO[R, E, A])(release: Exit[Any, Any] => URIO[R, Any]): ZMana
  makeReserve(acquire.map(a => Reservation(ZIO.succeed(a), release)))
```

Notice that in the implementation above we performed the `acquire` effect and then constructed a `Reservation` with its result. This way the `acquire` effect will be uninterruptible, which is what we want here.

To a large extent `make` and its variants are the fundamental constructors for `ZManaged` values. Most of the other constructors are just convenience methods that make it easier to construct `ZManaged` values in specific cases.

15.3.2 Convenience Constructors

The first of these convenience constructors, `fromEffect`, just transforms a `ZIO` value into a `ZManaged` value without adding any finalization logic and without changing the interruptibility of the `ZIO` value. You can also do the same thing by calling `toManaged_` on any `ZIO` value.

```
def fromEffect[R, E, A](zio: ZIO[R, E, A]): ZManaged[R, E, A] =
  ZManaged.makeInterruptible(zio)(_ => ZIO.unit)
```

You might wonder why this is useful. If a `ZManaged` value is supposed to represent a resource that requires some finalization, what is the point in constructing a `ZManaged` value that has no finalization logic?

The answer is that it can be extremely useful because it allows us to compose all of our existing `ZIO` values with `ZManaged` values. For example, consider the following snippet:

```
lazy val weatherDataAndResults: ZManaged[Console, Throwable, (File, File)] =
  for {
    weatherData <- file("temperatures.txt")
    _           <- console.putStrLn("Opened weather data file.")
    results     <- file("results.txt")
    _           <- console.putStrLn("Opened results file.")
  } yield (weatherData, results)
// error: not found: type Console
// lazy val weatherDataAndResults: ZManaged[Console, Throwable, (File, File)] =
//                                          ^^^^^^^
// error: type mismatch;
//  found   : zio.ZIO[zio.console.Console,Nothing,(java.io.File, java.io.File)]
//     (which expands to)  zio.ZIO[zio.Has[zio.console.Console.Service],Nothing,(java.io.Fi
//  required: zio.ZManaged[?,?,?]
//     _           <- console.putStrLn("Opened results file.")
//     ^
// error: type mismatch;
//  found   : zio.ZManaged[Any,Throwable,Nothing]
//  required: zio.ZIO[?,?,?]
//     results     <- file("results.txt")
//     ^
// error: type mismatch;
//  found   : zio.ZIO[zio.console.Console,Nothing,Nothing]
//     (which expands to)  zio.ZIO[zio.Has[zio.console.Console.Service],Nothing,Nothing]
//  required: zio.ZManaged[?,Throwable,(java.io.File, java.io.File)]
//     _           <- console.putStrLn("Opened weather data file.")
//     ^
```

This seems quite straightforward. We just want to add some debugging logic to our definition of this resource that sequentially opens the weather data and results files, ensuring that they are safely closed when we are done using them.

But this code won't compile as written!

`file` returns a `ZManaged` value whereas `putStrLn` returns a `ZIO` value. The `flatMap` and `map` operators, which a *for comprehension* is translated into, require the different values being composed to be of the same type, either all `ZIO` values or all `ZManaged` values.

So what are we to do? Do we need to implement new versions of `putStrLn` and everything else to work in the context of a `ZManaged`?

No! To make this code compile, all we have to do is use `toManaged_` to transform our `ZIO` values into `ZManaged` values.

```
import zio.console._

lazy val weatherDataAndResults: ZManaged[Console, Throwable, (File, File)] =
  for {
    weatherData <- file("temperatures.txt")
    _           <- putStrLn("Opened weather data file.").toManaged_
    results     <- file("results.txt")
    _           <- putStrLn("Opened results file.").toManaged_
  } yield (weatherData, results)
```

Now our code compiles and we can add debugging statements, or whatever other logic we like, to the definition of our resources.

This illustrates a good general principle. Often when starting to work with other ZIO effect types such as `ZManaged` people can get into a situation where they have a mix of different effect types, like `ZIO` and `ZManaged`, resulting in code that doesn't compile or complex types with multiple nested data types.

A good rule of thumb when facing this problem is to lift all your effect types to the effect type with the "most" structure. For example, if you are working with a combination of `ZIO` and `ZManaged` values, promote all the `ZIO` values to `ZManaged` values with `toManaged_` and then everything will compose nicely together with operators like `flatMap`, `map`, `zip`, `foreach`, and `collectAll`.

This works because `ZManaged` has *more* structure than `ZIO` so we can always transform a `ZIO` value into a `ZManaged` value while preserving its meaning. We just add a finalizer that does nothing and so the fact that we have added it at all is completely unobservable.

In contrast, if we try to transform a `ZManaged` value into a `ZIO` value we lose information. If we `use` a `ZManaged` value to get a `ZIO` value we lose the ability to compose it with other resources to describe more complex resources, so we want to avoid doing that until we are sure we are done defining our resource.

There is also a `toManaged` variant on `ZIO` that makes the `ZIO` effect uninterruptible and adds a finalizer that will be run when the `ZIO` effect completes execution. This can be particularly useful where you have a `ZIO` effect that returns some resource requiring finalization.

```
trait ZIO[-R, +E, +A] { self =>
  def toManaged[R1 <: R](release: A => URIO[R1, Any]): ZManaged[R1, E, A] =
    ZManaged.make(self)(release)
}
```

For example, we saw previously that we can create a `Queue` using a constructor like `Queue.bounded` and that we can shut down the `Queue` by using the `shutdown` operator. Shutting down the `Queue` will cause any fiber attempting to offer a value to the `Queue` or take a value from the `Queue` to be immediately interrupted.

If we use one of these constructors directly we are responsible for shutting

down the queue ourselves, including dealing with the possibility of failure or interruption. If we instead transform the `ZIO` value describing the creation of the `Queue` into a `ZManaged` value this can be taken care of for us automatically.

```
val managedQueue: ZManaged[Any, Nothing, Queue[Int]] =
  Queue.bounded(16).toManaged(_.shutdown)
// managedQueue: ZManaged[Any, Nothing, Queue[Int]] = zio.ZManaged$$anon$2@6a16fdc1
```

We can then get access to the `Queue` by calling `use` with the guarantee that the `Queue` will automatically be shut down when the `use` action completes execution. We can also compose the `Queue` with other resources that require finalization.

In addition to these constructors for creating `ZManaged` values from `ZIO` effects there are also a variety of constructors for importing existing values or side effecting code into a `ZManaged` value without having to go through `ZIO`.

For example, to import side effecting code that doesn't throw an exception into a `ZManaged` we could do:

```
val helloWorldFromAManagedLong: ZManaged[Any, Nothing, Unit] =
  ZManaged.fromEffect(ZIO.effectTotal(println("Hello from a Managed!")))
// helloWorldFromAManagedLong: ZManaged[Any, Nothing, Unit] = zio.ZManaged$$anon$2@38b2d896
```

But we can do the same thing more succinctly as:

```
val helloWorldFromAManaged: ZManaged[Any, Nothing, Unit] =
  ZManaged.effectTotal(println("Hello, World!"))
// helloWorldFromAManaged: ZManaged[Any, Nothing, Unit] = zio.ZManaged$$anon$2@1d3849a7
```

For most common operators you are familiar with from `ZIO`, such as `succeed`, `effect`, and `effectTotal` there are variants on `ZManaged` that are exactly the same as the constructors you are familiar with from `ZIO` except that they return `ZManaged` instead of `ZIO` values. For constructors where there is not a specialized `ZManaged` variant, like `effectAsync`, you can just use the constructor on `ZIO` and then `fromEffect` or `toManaged` to transform it into a `ZManaged` value.

15.4 Transforming Managed Resources

The next thing for us to learn about are operators for transforming managed resources. Broadly, there are two categories of operators that we will talk about.

15.4.1 Operators With Direct Counterparts On ZIO

The first category is operators for composing `ZManaged` values that are analogous to existing operators for transforming `ZIO` values. These would include operators like `flatMap`, `map`, and `zipPar` that we have already talked about.

Going back to the idea that a `ZManaged` value is a `ZIO` effect with additional structure describing finalization logic, these operators transform `ZManaged` values

in a way analogous to how their counterparts transform `ZIO` values and contain further logic for preserving the additional structure of the `ZManaged` value.

For example, `ZManaged#flatMap` describes acquiring two resources sequentially just like `ZIO#flatMap` describes performing two concurrent programs sequentially and contains additional logic to describe how to combine the finalization logic of the two resources, in this case by releasing the resources in reverse order of acquisition. Similarly `ZManaged#zipPar` described parallel acquisition of resources just like `ZIO#zipPar` describes performing concurrent programs in parallel, this time adding the additional logic to also release the resources in parallel.

Most operators defined on `ZIO` are also defined on `ZManaged` with the exception of a few that deal directly with lower level concerns such as fibers. The only thing that may be different is the additional logic they add for dealing with finalizers and you can check the documentation for specific operators if you have any question about how they deal with finalizers.

15.4.2 Operators Without Direct Counterparts on ZIO

The second set of operators for composing `ZManaged` values take advantage of the additional structure a `ZManaged` value provides to keep track of the finalizers associated with a resource and so do not have a direct analogue with `ZIO`.

The first of these is the `onExitFirst` operator, which allows adding a new finalizer that will be run *before* the existing finalizers.

```
trait ZManaged[-R, +E, +A] {
  def onExitFirst(cleanup: Exit[E, A] => URIO[R, Any]): ZManaged[R, E, A]
}
```

Because `ZIO` doesn't have a concept of keeping track of the finalizers associated with an effect in the same way this operator isn't well defined on `ZIO` but it is on `ZManaged`.

There is also a simpler variant of this, `ensuringFirst`, that adds a finalizer when the finalizer does not depend on how the `ZManaged` completed execution.

```
trait ZManaged[-R, +E, +A] {
  def ensuringFirst(f: URIO[R, Any]): ZManaged[R, E, A]
}
```

The final set of operators in this category are `withEarlyRelease` and `withEarlyReleaseExit`.

```
trait ZManaged[-R, +E, +A] {
  def withEarlyReleaseExit: ZManaged[R, E, (UIO[Any], A)]
  def withEarlyReleaseExit(e: Exit[Any, Any]): ZManaged[R, E, (UIO[Any], A)]
}
```

These operators return a `ZManaged` effect that, in addition to producing a resource, also produces an effect that we can use to release the resource early.

15.5 Using Managed Resources

The final step in working with `ZManaged` is to `use` it. Ultimately we need to convert a `ZManaged` resource back into a `ZIO` blueprint of a concurrent effect so we can actually run it, and typically the way we will do that is by using it in some way.

The most basic operator for doing this is `use`, which we saw before.

```
trait ZManaged[-R, +E, +A] {
  def use[R1 <: R, E1 >: E, B](f: A => ZIO[R1, E1, B]): ZIO[R1, E1, B]
}
```

The `use` operator is the standard way of using a `ZManaged`. The `ZManaged` represents some resource and we use that resource with the specified function, which gets access to the resource and can do something with it, with the guarantee that the resource will be closed afterwards no matter what.

In addition to the basic variant of `use` there are several more specialized variants.

The first is `use_`, which lets us perform an effect between the resource being acquired and released but where the effect does not actually need access to the resource.

```
trait ZManaged[-R, +E, +A] {
  def use_[R1 <: R, E1 >: E, B](effect: ZIO[R1, E1, B]): ZIO[R1, E1, B]
}
```

Normally we think of a `ZManaged` as producing some concrete "resource" like a file handle that we use but that isn't necessarily the case.

For example, our application could require an instance of some external service to be online for our application to function. In that case the `acquire` action could just be starting that external service so that it is online when the main part of our application, the `effect` in the signature of `use_` above, starts running.

Another variant of `use` that can be useful is `useNow`.

```
trait ZManaged[-R, +E, +A] {
  val useNow: ZIO[R, E, A] =
    use(a => ZIO.succeed(a))
}
```

This is equivalent to just calling `use` and returning the result of the `ZManaged` instead of doing anything else with it. The important thing to keep in mind with this operator is that the `release` action of the `ZManaged` will be run as

soon as the `A` value is produced so the `A` value must be something that remains valid even after the `release` action has been run.

For example, this would not be a good idea:

```
lazy val effect: ZIO[Any, Throwable, File] =
  file("temperatures.txt").useNow
```

The `file` effect is supposed to return a `File`, but by the time `useNow` completes execution the file will have already been closed so anyone calling `file` and hoping to do something with it is going to be sorely disappointed.

On the other hand, this will work fine:

```
lazy val managed: ZManaged[Any, Throwable, Unit] =
  for {
    weatherData <- file("temperatures.txt")
    results     <- file("results.txt")
    _           <- analyze(weatherData, results).toManaged_
  } yield ()

lazy val effect: ZIO[Any, Throwable, Unit] =
  managed.useNow
```

We have encompassed all of the logic in our program in the `ZManaged` value `managed` so now when we call `useNow` both files will be opened, then the value added analysis will be performed and the results written to the results file, and finally both files will be closed. In this case the result type was just `Unit` but it would be fine for the `managed` to return some other data type such as a summary of the results as long as that data type was valid after the files were closed.

The one watch out here is that staying in a `ZManaged` makes it easier to compose with other managed effects, so generally it is best to avoid calling one of the `use` variants unless you are really sure you are done with the resource. But `useNow` can be very convenient for testing or debugging.

The final `use` variant is slightly more complicated and is `useForever`.

```
trait ZManaged[-R, +E, +A] {
  val useForever: ZIO[R, E, Nothing] =
    use(_ => ZIO.never)
}
```

This is another operator that may make us wonder at first how it could be useful. This `ZManaged` is supposed to represent a resource and now not only are we not doing anything with the resource but we are preventing our program from ever terminating.

The answer is that this can be very useful for a program like a server that is never supposed to return anything but instead keeps running and serving requests,

potentially forever or until we shut it down. To see this in action let's create a simple interface for a hypothetical server.

```
trait Server {
  def shutdown(): Unit
}

lazy val makeServer: ZIO[Any, Throwable, Server] =
  ???

lazy val server: ZManaged[Any, Throwable, Server] =
  ZManaged.make(makeServer)(server => ZIO.effectTotal(server.shutdown()))

lazy val myProgram: ZIO[Any, Throwable, Nothing] =
  server.useForever
```

The server has a `shutdown` method that we want to make sure is called when the server terminates.

The `makeServer` method creates a new server and starts it. The server will run indefinitely until the `shutdown` method is called.

We create a `ZManaged` value to describe starting and shutting down the server to make sure that if the server is started it will be closed. The only problem with this is that if we call `use` like we normally would with a `ZManaged` the server will be shut down as soon as the `use` action terminated.

To avoid this we call `useForever`, this way the `use` action never terminates. As a result, the only way this program will terminate is if the `server` method fails to create the server or if the program is externally interrupted, for example by us sending an interrupt signal. If such a signal is received the `use` action of `ZIO.never` will be interrupted and the finalizer will be run, cleanly shutting down the server.

So here even though the signature may have seemed a little odd initially `useForever` gets us exactly what we want for a relatively common use case.

15.6 Conclusion

If you are simply composing resources together sequentially or in parallel you should have the tools from this chapter to describe resources in an extremely safe and ergonomic way. For the vast majority of use cases this will give you what you need to solve the problems you are facing.

However, in some cases you may need to modify the normal lifecycle of a resource. For example, you may want to share the same resource between different parts of your application and prevent the resource from being released until those other parts of your application are also done using it, release a resource early, or even

switch back and forth between multiple managed resources.

In the next chapter we will learn more about the internal implementation of `ZManaged` as well as how we can use some of the more advanced operators defined on `ZManaged` to solve these and similar problems.

15.7 Exercises

Chapter 16

Resource Handling: Advanced Managed

In this chapter, we will go deeper in our understanding of ZIO's `ZManaged` data type. In doing so, we will focus on two key areas: (1) understanding the internal representation of `ZManaged` and (2) learning about operators that allow us to separate the acquisition and release effects of a `ZManaged`.

At the beginning of the last chapter we implemented a toy version of `ZManaged` that just wrapped up the `acquire` and `release` effects of `bracket` into a data type but we said that this was not how `ZManaged` is actually implemented. Now we will see what the actual implementation of `ZManaged` looks like.

This will position us to write our own advanced operators that need to be implemented directly in terms of the internal representation of `ZManaged` and cannot simply be implemented in terms of existing operators. It will also enable us to understand some of the more advanced operators on `ZManaged` implemented in ZIO itself that take advantage of the full power of this representation.

Later in the last chapter we looked at a variety of operators for working with `ZManaged` but generally all of these operators returned another `ZManaged` value that kept the `acquire` and `release` effects together and added some further logic to them. Using these operators is the safest thing to do because as long as we keep the `acquire` and `release` actions together then whenever we `use` a resource we are guaranteed that if the resource is acquired it will automatically be closed, no matter how the `use` effect terminates.

However, in some more advanced use cases we may need to separate the `acquire` and `release` effects before potentially recombining them. For example, we may want to support memoizing resources so that if multiple callers get the same resource the resource will only be acquired once, with subsequent callers receiving

a reference to the same resource and the resource automatically closed when all callers are done with it.

We will learn about a variety of advanced operators that have been implemented to make these kind of use cases easier, as well as some of the pitfalls to avoid, and will get practice solving these problems ourselves.

If all you want is to use `ZManaged` to safely describe and work with resources then the material in the last chapter should be all you need and you can feel free to skip this chapter or refer back to it as a reference if needed. However, if you are curious about how `ZManaged` is implemented or want to implement your own operators for working with managed resources then read on!

16.1 Internal Representation Of Managed

The actual implementation of `ZManaged` is:

```
import zio._

sealed abstract class ZManaged[-R, +E, +A] {
  def zio: ZIO[(R, ReleaseMap), E, (Finalizer, A)]
}

type Finalizer = Exit[Any, Any] => UIO[Any]

abstract class ReleaseMap {
  def add(finalizer: Finalizer): UIO[Finalizer]
  def releaseAll(exit: Exit[Any, Any], execStrategy: ExecutionStrategy): UIO[Any]
}

object ReleaseMap {
  def make: UIO[ReleaseMap] =
    ???
}
```

Let's spend some time unpacking this definition since it looks significantly different than the toy version of `ZManaged` we implemented in the last chapter to see how it actually has equivalent and even greater power.

As we can see, a `ZManaged[R, E, A]` is implemented in terms of one `ZIO` effect, which requires an environment including both `R` and a `ReleaseMap`, can fail with an `E`, and can succeed with both a value `A` and a `Finalizer`. What then is a `Finalizer` and a `ReleaseMap` and how do they help us turn an ordinary `ZIO` effect into a `ZManaged` value?

16.1.1 Finalizers

A `Finalizer` is just a type alias for a function `Exit[Any, Any] => UIO[Any]` and represents some finalization logic that can potentially depend on how an effect terminates.

The returned effect has a type of `UIO[Any]` because a finalizer does not return any useful value, it just performs some finalization logic for its effects. For example, a finalizer could close a file or a socket connection or remove a value from a mutable data structure.

In any of these cases the actual value returned by the finalizer doesn't matter, just the fact that it performs that finalization logic and closes the file or socket connection or removes the value from the data structure. Often, the effect returned by the finalizer will have a type of `UIO[Unit]`, but the type signature of `UIO[Any]` allows more flexibility in case the effect returns some additional value that we don't need, such as whether the value was in the data structure before we removed it.

The finalizer allows the finalization logic to depend on an `Exit` value, indicating how the effect that the resource was being used in terminated execution. For example, we could potentially perform different finalization logic for closing a file depending on whether the effect we were using the file with terminated successfully or failed with an error.

To see a simple example of a finalizer, we could define a finalizer for closing a file like this:

```
import java.io.File

def closeFile(file: File): UIO[Unit] =
  ???

def finalizer(file: File): Finalizer =
  _ => closeFile(file)
```

Notice that in this case the logic of how to close the file did not depend on how the effect we were performing with the file terminates. If we wanted the finalization logic to depend on how the effect using the file was terminated we could have pattern matched against the `Exit` value.

To summarize, a `Finalizer` just represents some finalization logic being performed for its effects that may depend on how the main effect completes execution.

16.1.2 The Release Map

In its simplest form, you can think of a `ReleaseMap` as a set of finalizers. Its two most fundamental operators are `add` and `releaseAll`.

The `add` operator adds a single `Finalizer` to the `ReleaseMap`, returning a

new `Finalizer` that will run the original `Finalizer` and remove it from the `ReleaseMap`. The `releaseAll` method takes an `Exit` value and runs all the finalizers in the `releaseMap`.

The `releaseAll` operator also takes as input an `ExecutionStrategy`. We haven't seen the `ExecutionStrategy` data type before but it is actually quite simple and just describes the different ways we can potentially try to do something sequentially or in parallel.

```
sealed trait ExecutionStrategy

object ExecutionStrategy {
  case object Sequential extends ExecutionStrategy
  case object Parallel extends ExecutionStrategy
  final case class ParallelN(n: Int) extends ExecutionStrategy
}
```

Conceptually there are three ways we can execute a group of effects: (1) sequentially like with the `foreach` operator on `ZIO`, (2) in parallel with unbounded parallelism like with the `foreachPar` operator on `ZIO`, and (3) in parallel with bounded parallelism like with the `foreachParN` operator on `ZIO`.

The `ExecutionStrategy` data type just allows us to describe these different possibilities as a data type. This allows us to simplify APIs in some cases.

For example, we could have had separate `releaseAll`, `releaseAllPar`, and `releaseAllParN` variants on `ReleaseMap`. But instead we can make the `ExecutionStrategy` another parameter and then we can implement a single variant of the `releaseAll` method that implements the appropriate logic for running finalizers either sequentially or in parallel based on the execution strategy we specify.

In addition to simply being a set of finalizers, the `ReleaseMap` handles some important logic regarding potential concurrent addition and release of finalizers. In particular, we need to make sure that we don't drop finalizers because a finalizer is added after `releaseAll` is called.

To prevent this, the guarantee is that if `add` is called after `releaseAll` has been called then the specified finalizer will immediately be run with the `Exit` value provided when `releaseAll` was called. We will see more about why this guarantee is important later as we see how `ReleaseMap` fits into the implementation of `ZManaged`.

16.1.3 Putting It Together

Now that we know what a `Finalizer` is and have a basic understanding of `ReleaseMap`, let's see how these fit together to allow us to implement `ZManaged`.

Recall that the definition of `ZManaged` is:

```
sealed abstract class ZManaged[-R, +E, +A] {
  def zio: ZIO[(R, ReleaseMap), E, (Finalizer, A)]
}
```

So a `ZManaged` is a `ZIO` effect that, if given a `ReleaseMap`, returns a resource along with a `Finalizer` that will release that resource.

Conceptually, then, creating a `ZManaged` value just requires creating a `ZIO` effect that returns the resource and adds a finalizer to the `ReleaseMap` to close that resource. We can see this for ourselves by implementing the `makeExit` constructor, one of the fundamental constructors for `ZManaged` values that we saw in the last chapter.

To start, let's implement a simple convenience constructor to make it easier to build `ZManaged` values without having to implement the `zio` method of `ZManaged` directly.

```
object ZManaged {
  def apply[R, E, A](run: ZIO[(R, ReleaseMap), E, (Finalizer, A)]): ZManaged[R, E, A] =
    new ZManaged[R, E, A] {
      def zio = run
    }
}
```

Now we can implement `makeExit` like this:

```
def makeExit[R, E, A](acquire: ZIO[R, E, A])(release: (A, Exit[Any, Any]) => URIO[R, Any]):
  ZManaged {
    ZIO.uninterruptible {
      for {
        r         <- ZIO.environment[(R, ReleaseMap)]
        a         <- acquire.provide(r._1)
        finalizer <- r._2.add(release(a, _).provide(r._1))
      } yield (finalizer, a)
    }
  }
```

The implementation is actually relatively straightforward, but let's walk through it. We are using the `apply` method on 'ZManaged we just implemented.

First, we wrap the entire effect in `ZIO.uninterruptible` since for `makeExit` the contract is that both the `acquire` and the `release` effects cannot be interrupted.

Second, we access the environment to get access to both the environment type of the `ZManaged`, `R`, as well as the `ReleaseMap`. Because `ZManaged` uses a tuple of `(R, ReleaseMap)` for the environment type whereas `ZIO` just uses `R`, a common pattern when implementing low level `ZManaged` operators is to have to use `environment` and `provide` to translate between these types.

Once we have access to the environment we `provide` it to the `acquire` effect. This eliminates the effects environmental dependency on `R` so now the only

environment type we depend on is `(R, ReleaseMap)`, the expected environment type for the `ZManaged` constructor.

The `acquire` effect returns the resource itself, `a`, and then the only thing we need to do is add the `release` effect to the `ReleaseMap`. The `release` effect in `makeExit` is expecting both an `A` value as well as an `Exit` value, since it needs access to the resource we acquired, for example the file, to close it, so we feed that to `release` and again provide the environment.

The `add` operator gives us back a new finalizer that will both run the original `release` action and remove it from the `ReleaseMap` so we don't accidentally try to run it a second time. We then just return back both the new finalizer and the original resource.

Hopefully you can see from the above that there actually isn't anything very fancy going on here. We had to do a little bit of work to get the environment types to line up right, but otherwise all we had to do was grab the `ReleaseMap` from the environment, acquire the resource, and then add the finalizer that will release the resource to the `ReleaseMap`, returning both the new finalizer and the resource.

One thing that made implementing this operator easier was that both the acquire and release actions were supposed to be uninterruptible so we could wrap the whole thing in a `ZIO.uninterruptible` block. In general one of the things we need to be careful of is making sure we are not interrupting between acquiring the resource and adding its finalizer to the release map, since if that happens we could leak the resource.

Now that we have an understanding of how we can create `ZManaged` values with this representation, let's look at how we can `use` a `ZManaged` value to get back to a normal `ZIO` effect.

Using a `ZManaged` value to get back to a `ZIO` value just requires creating an empty `ReleaseMap`, running the effect underlying the `ZManaged` value to produce the resource and doing something with it, and then running the finalizer at the end, using `bracket` to ensure that we always run the effect.

```
trait ZManaged[-R, +E, +A] {
  def zio: ZIO[(R, ReleaseMap), E, (Finalizer, A)]
  def use[R1 <: R, E1 >: E, B](f: A => ZIO[R1, E1, B]): ZIO[R1, E1, B] =
    ReleaseMap.make.bracketExit(
      (releaseMap, exit: Exit[E1, B]) => releaseMap.releaseAll(exit, ExecutionStrategy.Seque
      releaseMap => zio.provideSome[R]((_, releaseMap)).flatMap { case (_, a) => f(a) }
    )
}
```

Once again this may look somewhat complex but it is actually relatively straightforward. Let's follow the logic through.

The first thing we do is create an empty `ReleaseMap` and use `bracketExit` to

specify what we want to do with the `ReleaseMap` as well as some finalization logic that we want to run when we are done using the `ReleaseMap`, whether that is because of success, failure, or interruption.

What we want to do with the `ReleaseMap` is specified in the `use` action which is the second argument to `bracketExit`.

In this case we just use `provideSome` to provide the effect underlying the `ZManaged` with the `ReleaseMap` it requires, and then use `flatMap` to get access to the resource `A` and perform some effect with it. We can ignore the `Finalizer` that is returned by the effect underlying the `ZManaged` because the `release` action of `bracketExit`.

Moving on then to the release action of `bracketExit`, this ends up being quite simple too and just calls `releaseAll` on the `ReleaseMap` to actually run the finalizers.

16.2 Separating Acquire And Release Actions Of Managed

Now that we have a better understanding of the internal implementation of `ZManaged`, let's look at how we can use that to solve some more complicated problems involving `ZManaged`.

One of the helpful ways to distinguish "advanced" uses of `ZManaged` is whether we are having to work with the `acquire` and `release` actions of the `ZManaged` separately.

Ideally, we would never have to work with the `acquire` and `release` actions separately.

For example, in the last chapter we implemented a `file` constructor that returned a `ZManaged` describing the safe acquisition and release of a file handle. As long as we used that constructor and combined resources with standard operators like `map`, `flatMap`, `zipPar` and their variants we can use those resources with the guarantee that if a file handle is open it will always be closed, no matter what.

This is a very powerful guarantee, and essentially allows us to not have to worry about resource management when writing our programs anymore as long as we use these tools, letting us focus entirely on our business logic.

But sometimes there are cases where we need to actually separate the `acquire` and `release` action in order to combine them in some different way that is not described by existing operators. ZIO definitely provides the tools for us to do this, but we need to be more careful because when we work with the `acquire` and `release` actions separately we no longer have the same guarantees that `ZManaged` was previously providing, so we need to think about more of the things that were previously taken care of for us.

As a motivating example, let's look at how we could construct a `MemoMap`.

As defined here, a `MemoMap` is essentially a concurrent cache of resources.

A `MemoMap` is constructed by providing a `lookup` function, which given a key of type `K` returns a `ZManaged[R, E, V]` that describes some resource associated with the key.

Users interact with the `MemoMap` by calling `get` with a key `K` and getting back a `ZManaged[R, E, V]`. The returned `ZManaged` value will acquire the original resource if it has not already been acquired, but if the resource has already been acquired it will simply return a reference to the original resource.

This way if multiple fibers attempt to acquire the same resource the resource will only be acquired a single time and shared between the different fibers, instead of the resource being acquired multiple times. This is similar to a data structure that is used internally in the implementation of `ZLayer` to support memoizing services in ZIO's solution to the dependency injection problem, discussed in the next section.

The interface looks like this:

```
import zio._

trait MemoMap[-K, -R, +E, +V] {
  def get(k: K): ZManaged[R, E, V]
}

object MemoMap {
  def make[K, R, E, V](lookup: K => ZManaged[R, E, V]): UIO[MemoMap[K, R, E, V]] =
    ???
}
```

The tricky thing here is that we need to separate the `acquire` and `release` actions of the original resource. We want the `ZManaged` value returned by `get` to acquire the underlying resource the first time it is invoked, but we can't necessarily release the underlying resource when we are done using it because another caller may be using the resource at the same time.

To handle this, we will introduce a concept of "observers". Each time the `ZManaged` effect returned by `get` is used we will increment the number of observers by one when we start using the resource and decrement it by one when we finish using the resource.

This way when we finish using the resource we can check whether any other fibers are currently using the same resource. If they are then we won't release the resource and leave it to the other fibers to release the resource when they are done with it, otherwise it is safe for us to go ahead and release the resource ourselves.

Clearly this will require us substantially changing the finalization logic of the underlying managed resource, and potentially holding open the resource significantly longer than its normal life if multiple fibers are using it.

There is a lot going on here, so let's start from a high level. An initial sketch of an implementation might look like this:

```
def make[K, R, E, V](lookup: K => ZManaged[R, E, V]): UIO[MemoMap[K, R, E, V]] =
  Ref.make[Map[K, (ZIO[R, E, V], ZManaged.Finalizer)]](Map.empty).map { ref =>
    new MemoMap[K, R, E, V] {
      def get(k: K): ZManaged[R, E, V] =
        ZManaged {
          ref.modify { map =>
            map.get(k) match {
              case Some((acquire, release)) => ???
              case None => ???
            }
          }
        }
    }
  }
```

We create a `Ref` that will contain a `Map` from keys to `acquire` and `release` actions. When a fiber attempts to get the resource associated with a key from the `MemoMap` we use `Ref#modify` to atomically check whether the resource already exists in the map, returning a new resource that either acquires the underlying resource if it has not already been acquired or else just returns a reference to the previously acquired resource.

Let's start with implementing the case where the resource does not already exist in the map.

To avoid the resource being created twice if two fibers call `get` at the same time we know that in `modify` we will have to place a value in the `Map` immediately. However, the resource may take time to acquire so we will need to use a `Promise` to allow other fibers to wait on the resource being available while the first fiber acquires it.

We will also need to create a `Ref` to capture the number of observers that are currently using the resource. This will require us to convert our `Ref` containing the `Map` into a `RefM` to we can perform effects such as creating the `Promise` and `Ref`.

Here is what the implementation of the `None` case might look like

```
def make[K, R, E, V](lookup: K => ZManaged[R, E, V]): UIO[MemoMap[K, R, E, V]] =
  RefM.make[Map[K, (IO[Any, Any], ZManaged.Finalizer)]](Map.empty).map { ref =>
    new MemoMap[K, R, E, V] {
      def get(k: K): ZManaged[R, E, V] =
        ZManaged {
```

```
ref.modify { map =>
  map.get(k) match {
    case Some((acquire, release)) =>
      ???
    case None =>
      for {
        observers    <- Ref.make(0)
        promise      <- Promise.make[Any, Any]
        finalizerRef <- Ref.make[ZManaged.Finalizer](ZManaged.Finalizer.noop)

        resource = ZIO.uninterruptibleMask { restore =>
          for {
            env                  <- ZIO.environment[(R, ZManaged.ReleaseMap)]
            (r, outerReleaseMap)  = env
            innerReleaseMap      <- ZManaged.ReleaseMap.make
            result               <- restore(lookup(k).zio.provide((r, innerRelease
              case e @ Exit.Failure(cause) =>
                promise.halt(cause) *>
                  innerReleaseMap.releaseAll(e, ExecutionStrategy.Sequential) *>
                  ZIO.halt(cause)
              case Exit.Success((_, v)) =>
                for {
                  _ <- finalizerRef.set { (e: Exit[Any, Any]) =>
                    ZIO.whenM(observers.modify(n => (n == 1, n - 1)))(
                      innerReleaseMap.releaseAll(e, ExecutionStrategy.Sequential)
                    )
                  }
                  _              <- observers.update(_ + 1)
                  outerFinalizer <- outerReleaseMap.add(e => finalizerRef.get.flat
                  _              <- promise.succeed(v)
                } yield (outerFinalizer, v)
            }
          } yield result
        }

        memoized = (
          promise.await.onExit {
            case Exit.Failure(_) => ZIO.unit
            case Exit.Success(_) => observers.update(_ + 1)
          },
          (exit: Exit[Any, Any]) => finalizerRef.get.flatMap(_(exit))
        )

      } yield (resource, map + (k -> memoized))
  }
}.flatten
```

```
        }
    }
  }
```

This is one of the more complex examples we have seen, so let's walk through it in detail.

At a high level, we are doing two things here.

First, we are creating some pieces of internal state that we need to maintain. This includes the `Ref` with the number of observers, the `Promise` that will contain the result of acquiring the resource, and the `Ref` with the finalizer to run.

Second, we need to create two different managed resources.

`resource` is a `ZManaged` value that will, when used, acquire the underlying resource and complete the `Promise` to make the resource available to any other fibers getting the same resource. The first fiber to call `get` with a key will receive the `resource` effect and will be responsible for acquiring the underlying resource.

`memoized` is a `ZManaged` value that will, when used, await the `Promise` to be completed with the result of acquiring the underlying resource. All fibers calling `get` with a key other than the first one will receive `memoized` and will wait for the result of the first fiber acquiring the resource.

With that context, let's look in more detail at the implementation of `resource` and `memoized`, starting with the implementation of `resource`.

The first thing we do in our implementation of `resource` is call `ZIO#uninterruptibleMask`. This is important because we are going to interact with the underlying representaiton of `ZManaged` here.

Whenever we work directly with the `ReleaseMap` we need to make sure we are not interrupted between running the `zio` value underlying the `ZManaged` and adding the returned `Finalzier` to the `ReleaseMap`. Otherwise we could drop a finalizer.

We use `uninterruptibleMask` instead of `uninterruptible` because generally there are parts of our code that we don't want to make interruptible so that users can interrupt effects if necessary, so the `restore` function returned by `uninterruptibleMask` gives us a way to restore the original interruptibility status for certain portions of the code.

Now that we know we won't be interrupted, the next thing we do is access the `outerReleaseMap` in the environment. This is the one that would be part of the larger `ZManaged` value that this particular resource might be a part of, potentially comtaining other finalizers associated with other resources unrelated to the `MemoMap`.

We then create our own `innerReleaseMap`. This will allow us to keep track of finalizers associated with the resource we are creating without necessarily

modifying the finalizers associated with other resources that aren't related to the `MemoMap`.

Now that we have access to the `ReleaseMap` instances we're ready to actually acquire the resource. We do that by calling the lookup function with the key to get the underlying `ZManaged` value that we want to acquire, then calling `zio` to get access to the underlying effect and providing it with both its required environment and the inner `ReleaseMap`.

We want this to be `interruptible` so we use the `restore` function from `uninterruptibleMask` and we also want to handle any errors that occur, including interruption, ourselves so we use the `run` operator to return an effect that always succeeds with an `Exit` value and then handle that.

If the acquisition of the resource failed then we complete the `Promise` with the failure to propagate the error to any other fibers that are attempting to get the same resource, then perform any necessary finalization logic immediately by calling `releaseAll` on the `innerReleaseMap` and finally fail ourselves with the error.

On the other hand, if the acquisition of the resource succeeded, then we update the `finalizerRef` with new finalization logic that will, when we are done using the resource, decrement the number of observers by one and then if there are no other observers run `releaseAll` on the `innerReleaseMap`. We then increment the number of observers by one since we are currently using the resource.

We also have to add a new finalizer to the `outerReleaseMap` that runs the finalizer in the `finalizerRef` to make sure that if the larger `ZManaged` effect that we are part of is terminated then our finalization logic will be run as well.

Finally, we complete the `Promise` with the result of acquiring the resource to make it available to other fibers.

Fortunately, the implementation of the `memoized` managed resource that is returned to subsequent fibers calling `get` is much simpler.

When these fibers use the `memoized` resource all they do is await the completion of the `Promise`.

If the `Promise` is completed with a failure then that means the first fiber never successfully acquired the resource so we don't have to do anything other than returning the same failure. If the `Promise` is completed with a success then we need to increment the number of observers by one to indicate that we are using the underlying resource and prevent another fiber from releasing it while we are using it.

The `release` action for the `memoized` resource is just to run the same finalizer contained in the `finalizerRef`, which will decrement the number of observers by one and call `releaseAll` on the `innerReleaseMap` if this fiber was the only observer.

With all this done the only thing left is for us to return the `resource` effect, since we are in the branch where the resource doesn't currently exist in the map and we are responsible for acquiring it, and to update the map with a new binding from the key to the `memoized` effect so other fibers wait for the result of us acquiring the resource.

The implementation when a resource already exists in the map is also much simpler, since we already handled much of it as part of the finalization logic we created above.

Looking up the key in the map already returns an acquire and a release action, so essentially all we have to do is package those up into a `ZManaged` value. The only other thing we need to do is access the `ReleaseMap` of the larger `ZManaged` value that we are potentially composed into and add our own release action into that if the underlying resource was successfully acquired so that finalization logic gets run when the managed effect we are composed into terminates.

The final implementation looks like this:

```
def make[K, R, E, V](lookup: K => ZManaged[R, E, V]): UIO[MemoMap[K, R, E, V]] =
  RefM.make[Map[K, (IO[Any, Any], ZManaged.Finalizer)]](Map.empty).map { ref =>
    new MemoMap[K, R, E, V] {
      def get(k: K): ZManaged[R, E, V] =
        ZManaged {
          ref.modify { map =>
            map.get(k) match {
              case Some((acquire, release)) =>
                val cached = ZIO.accessM[(R, ZManaged.ReleaseMap)] { case (_, releaseMap) =>
                  acquire
                    .asInstanceOf[IO[E, V]]
                    .onExit {
                      case Exit.Success(_) => releaseMap.add(release)
                      case Exit.Failure(_) => ZIO.unit
                    }
                    .map((release, _))
                }

                UIO.succeed((cached, map))
              case None =>
                for {
                  observers    <- Ref.make(0)
                  promise      <- Promise.make[Any, Any]
                  finalizerRef <- Ref.make[ZManaged.Finalizer](ZManaged.Finalizer.noop)

                  resource = ZIO.uninterruptibleMask { restore =>
                    for {
                      env                  <- ZIO.environment[(R, ZManaged.ReleaseMap)]
                      (r, outerReleaseMap)  = env
```

```
                    innerReleaseMap      <- ZManaged.ReleaseMap.make
                    result               <- restore(lookup(k).zio.provide((r, innerRelease
                      case e @ Exit.Failure(cause) =>
                        promise.halt(cause) *>
                          innerReleaseMap.releaseAll(e, ExecutionStrategy.Sequential) *>
                          ZIO.halt(cause)
                      case Exit.Success((_, v)) =>
                        for {
                          _ <- finalizerRef.set { (e: Exit[Any, Any]) =>
                            ZIO.whenM(observers.modify(n => (n == 1, n - 1)))(
                              innerReleaseMap.releaseAll(e, ExecutionStrategy.Sequential)
                            )
                          }
                          _              <- observers.update(_ + 1)
                          outerFinalizer <- outerReleaseMap.add(e => finalizerRef.get.flat
                          _              <- promise.succeed(v)
                        } yield (outerFinalizer, v)
                    }
                  } yield result
                }

                memoized = (
                  promise.await.onExit {
                    case Exit.Failure(_) => ZIO.unit
                    case Exit.Success(_) => observers.update(_ + 1)
                  },
                  (exit: Exit[Any, Any]) => finalizerRef.get.flatMap(_(exit))
                )

              } yield (resource, map + (k -> memoized))
            }
          }.flatten
        }
    }
  }
```

There are a lot of details in this example, but here are a couple things to focus on:

- **Use `uninterruptibleMask` when separating the `acquire` and `release` actions of a `ZManaged`** - Whenever you work directly with the underlying representation of a `ZManaged` there is a risk that if you are interrupted between running the `zio` effect underlying the `ZManaged` and adding the finalizer to the `releaseMap` or otherwise adding finalization logic you will drop a finalizer. Use `uninterruptibleMask` to create a region where you are protected from interruption and use the `restore` function to allow `interruption` in specific areas where it is safe to do so.

- **Use the functionality provided by `releaseMap`** - The `ReleaseMap` data structure builds in a lot of useful logic, for example ensuring that a finalizer is immediately run if it is added to a `ReleaseMap` that has already been released, that prevents tricky race conditions. Take advantage of it.

16.3 Conclusion

The vast majority of the time using the common built in operators on `ZManaged` gives you everything you need to work with resources in a way that is safe and allows you to solve your domain specific problems without worrying about resource safety.

But sometimes we need more power and flexibility. Hopefully the materials in this chapter have given the tools such that if you do want to work with resources in more complex ways you have the tools to do so.

In the next section we will move on to look at ZIO's solution to the dependency injection problem, which builds on ZIO's support for safe resource usage to allow us to describe dependencies of our application that may require their own finalization logic, with the same guarantee that this finalization logic will be run no matter what that we have seen with `bracket` and `ZManaged`.

Chapter 17

Dependency Injection: Essentials

We said before that ZIO's environment type provides a comprehensive solution to the dependency injection problem. But so far we have made only limited use of the environment to access the default services provided by ZIO such as the clock, console, and random services.

In this chapter and the next we will show in detail how ZIO solves the dependency injection problem.

We will begin by describing the dependency injection problem, traditional approaches to dependency injection, and their limitations. Then we will discuss ZIO's approach to dependency injection, including how to access the environment, how to compose effects that require different environments, and how to provide an effect with its required environment.

By the end of this chapter you will be a fluent user of ZIO's environment type, knowing how to access services such as databases or logging from within your own programs as well as how to provide your programs with live or test implementations so you can run them.

In the next chapter we will discuss how you can implement your own services and how you can combine multiple services to build more complex services from simpler ones.

17.1 The Dependency Injection Problem

Modern applications almost always depend on one or more *services*, which provide functionality that is necessary for the application to run but is not implemented by the application itself.

For example, a simple application might depend on a database service that allows reading values from and writing them to a database as well as a logging service that allows logging information while the application is running. In reality, a modern application will typically depend on many more services than this.

To achieve *loose coupling* of modules, we do not want to hard code our application to a particular database or logging service. Instead, we would like to be able to specify that our application depends on having *some* database or logging service.

Conceptually we would like to be able to say:

```
import zio._

trait Database
trait Logging

trait Application {
  def run(database: Database, logging: Logging): UIO[Unit]
}
```

That is, `Database` and `Logging` represent *interfaces* that our application can interact with and to run our program we can provide any *implementation* that provides the functionality of that interface.

For instance, the `Logging` service might provide the functionality to log an arbitrary line.

```
trait Logging {
  def logLine(line: String): UIO[Unit]
}
```

We could then call the `logLine` method anywhere within our application code to log a line. And we could run our program by providing any implementation of the `Logging` service that implemented `logLine`.

This style of architecture has several advantages.

First, it promotes testability because we can provide test implementations for each of these services that produce known outputs for given inputs.

If we hard coded our application to print output to the live console there would be no way to programmatically verify the console output. But if we express the console service as a dependency then we can provide a test implementation that just writes all outputs to a data structure that we can check.

Second, it makes it easier for us to switch out one or more of the dependencies later.

If we hard coded the dependency on the database we would be likely to rely on variety of implementation specific details of whatever database we were using, requiring us to potentially refactor our entire application if we wanted to change the database we were using. On the other hand, if the database was abstracted

as a service then we should be able to use a different database service with no changes to our application code.

Third, it makes it easier for the developers of those services to independently evolve them.

In any domain such as logging there are a variety of details in exactly how a logging service should be implemented. The interface serves essentially as the public API of the service and since users are only relying on the interface, implementers of the service are free to change any other aspects of their code as long as they implement that interface.

So we understand that explicitly modeling the dependencies of our application is valuable. What then is the dependency injection problem?

The dependency injection problem is essentially a "plumbing" problem. How do we get the dependencies of our application from the top of our application to where we are actually using it?

In a large application, there might be ten levels of method calls between the top of our application and where the methods on the service are actually invoked. We could explicitly pass the service down through each of these method calls as another parameter, but this becomes boilerplate very quickly, especially when we might have a dozen or more different services that our application depends on.

We could of course expose each of these services globally in our application, but then we lose all ability to see which parts of our application depend on which services or to potentially provide some parts of our application with different or modified services.

17.2 Limitations Of Traditional Dependency Injection Approaches

There are several traditional approaches to dealing with this problem, but all of them have significant limitations.

17.2.1 Tagless Final

The first traditional approach for dealing with the dependency injection problem is the so called *tagless final* style of programming. In this style, dependencies are modeled as implicit capabilities.

```
trait Logging[F[_]] {
  def logLine(line: String): F[Unit]
}

trait Database[F[_]]
```

```
trait Application {
  def run[F[_]: Logging : Database]: F[Unit]
}
```

Here the `F:[_]: Logging : Database` syntax is a *context bound* and desugars to:

```
def run[F[_]](implicit logging: Logging[F], database: Database[F]): F[Unit] =
  ???
```

This essentially says that `F` is some effect type for which an implicit `Logging[F]` and `Database[F]` are available.

We can then pass services down to lower levels of our application just by including those services in the context bounds for method calls along the way.

```
def foo[F[_]: Logging : Database]: F[Unit] =
  ???

def run[F[_]: Logging : Database]: F[Unit] =
  foo
```

To actually run our program we need to provide instances of `Logging` and `Database` for some concrete type `F[_]`. If we specialize `F[_]` to `UIO`, for example, we get:

```
trait Logging[F[_]] {
  def logLine(line: String): F[Unit]
}

object Logging {

  implicit val UIOLogging: Logging[UIO] =
    new Logging[UIO] {
      def logLine(line: String): UIO[Unit] =
        ???
    }
}
```

We could then run our program by just calling `run[UIO]`.

Before we talk about the limitations of this approach, let's talk about what is good about it.

It does allow us to explicitly model the dependencies of each part of our application. Our entire application has a signature of:

```
def run[F[_]: Logging : Database]: F[Unit] =
  ???
```

This indicates that our entire application requires both a `Logging` service and a `Database` service.

And one part of our application might have a signature of:

```
def bar[F[_]: Logging]: F[Unit] =
  ???
```

This indicates that this part of the application only requires a `Logging` service and does not require a `Database` service.

This approach also allows us to express the idea that when we combine two parts of our program that require different services, the combined program requires *both* the services needed by the first part and the services needed by the second part.

```
def baz[F[_]: Database]: F[Unit] =
  ???

def combined[F[_]: Logging: Database]: F[Unit] = {
  bar
  baz
}
```

Here `bar` requires an implicit `Logging` service to be in scope and `baz` requires an implicit `Database` service to be in scope, so the Scala compiler will require that we propagate this dependency upward by making `combined` require both a `Logging` service and a `Database` service.

So the *tagless final* style definitely gets some things right. Where does it fall down?

First, this style of programming results in a style of code that is significantly more complex.

We have glossed over much of this complexity for the sake of brevity because we are trying to *describe* this style of coding rather than *teach* it. But just in the snippets above we have used a number of advanced concepts:

- `F[_]` is not a concrete type like a `List[Int]` but a type constructor that, when given another type, produces a concrete type.
- `Logging` and `Database` are modeled as implicit parameters, requiring knowledge of what implicit parameters are, how they can be represented as context bounds, how to represent implicit capabilities, and where to define implicits, among other things
- Since the `F` type is completely abstract other than the capabilities we require we will typically require additional capabilities to define basic concepts such as sequential composition. This would often be described by the `Monad` type class from functional programming in the tagless final style, for example. This obviously requires a much higher level of knowledge of functional programming concepts to understand this code at all.

Second, this style of programming actually doesn't fully solve our original problem of removing the boilerplate of explicitly passing dependencies through many levels of an application.

The tagless final style is certainly an improvement in allowing us to express the services required by a method as context bounds and it is relatively concise with only a couple of services. But we said before that a modern application might depend on a dozen or more services.

```
def run[F[_]: Logging : Database : Redis : Cassandra : Kafka : Persistence : Authentication
  ???
```

This is only eight services and already we're filling the entire line just writing out all of these services. Imagine that this entire bundle of services is required ten levels down in an application and repeating this type signature for every method declaration!

We would like to be able to abstract over this bundle of services by doing something like:

```
type Services[F[_]] = Logging[F] with Database[F] with Redis[F] with Cassandra[F] with Kafka
```

Then we could rewrite all of our method signatures to be:

```
def run[F[_]: Services]: F[Unit] =
  ???
```

This would allow us to just define the bundle of services that our application needs once and then use that definition everywhere just like we do when we factor out some other part of our program logic. But this does not work because Scala does not support abstracting over context bounds in this way.

Third, this style forces us to program our entire application in this `F` data type rather than a concrete type like `ZIO`.

This means that the only operators we can use are the ones available on the typeclasses that we require for `F`. Applications that use the tagless final style typically rely on functional programming libraries to provide abstractions that describe capabilities such as sequentially composing effects, but these are typically not as expressive as the operators available on a concrete data type like `ZIO` directly.

So overall the tagless final style is a step in the right direction but (1) is generally only feasible for senior teams or teams that are able to invest heavily in training and (2) does not solve the entire problem.

17.2.2 ReaderT

Another approach is to use *monad transformers*. Monad transformers allow layering some additional capability on top of an existing effect type.

For example, if we use `Task` as our base effect type to represent an effect that doesn't model having any environmental requirements, we could "layer on" the capability of working with an environment like this:

```
import zio._

final case class ReaderT[R, A](run: R => Task[A]) {
  def provide(r: R): Task[A] = run(r)
}

object ReaderT {
  def environment[R]: ReaderT[R, R] =
    ReaderT(r => Task.succeed(r))
}
```

We could write our entire application at the level of the `ReaderT` data type to get back to a similar place as we are with `ZIO`.

Again, this is a step in the right direction, as it is basically replicating the functionality that ZIO provides through its environment type. Recall that `provide` and `environment` were the fundamental methods of ZIO's environment type as well.

However, there are also a several areas where this falls short.

First, implementations that use `ReaderT` will not be as efficient as ones that use `ZIO` directly.

Using monad transformers like this is layering data types on top of data types so when our program actually goes to evaluate programs this way there are many additional steps of wrapping and unwrapping which both take time in of themselves and hamper the JVM's ability to do other optimizations.

ZIO is able to model accessing and providing the environment as well as all of the other capabilities we have seen as part of a single data type within a highly optimized runtime using a technique called *effect rotation* which results in much higher performance than using monad transformers.

Of course, sometimes this performance difference does not matter because we have other parts of our application, such as waiting on a network call, that take much longer. But generally we want each part of our stack to be as efficient as possible so that higher levels built on top of lower ones can also be as efficient as possible.

Second, there will typically not be nearly as many methods defined on the monad transformer data type as on a concrete effect type.

This is particularly true because monad transformers are generally written to be generic to the concrete effect type. So `ReaderT` does not even know that the type inside it is a `Task` but instead it is just an `F[_]` that potentially has some capabilities.

This makes it impossible to define operators on `ReaderT` that require specific capabilities of `F`, such as dealing with concurrency. So we often need to "dig into" the effect type inside `ReaderT` to perform operations, which adds another layer of boilerplate.

Third, type inference is typically significantly worse with monad transformers. The Scala compiler is generally less effective at inferring higher kinded types, such as `ReaderT` when `F[_]` is generic and this is especially true because as implemented monad transformers typically do not take full advantage of variance.

Fourth, monad transformers again significantly add to the complexity of code and the level of experience / training that developers need to be productive working with this type of code. We have tried to simplify as much as possible here but monad transformers generally involve higher kinded types and require type classes to describe the capabilities of the base effect type so require a much higher level of knowledge.

17.2.3 Implicits

Implicits are another approach that has been used to deal with the dependency injection problem. We saw some use of implicits before in our discussion of the tagless final style, but implicit parameters do not have to be used in connection with tagless final.

For example, one area where implicits are often used for dependency injection is with the `ExecutionContext` required for Scala's `Future`.

```
import scala.concurrent.{ ExecutionContext, Future }

def sayHello(name: String)(implicit ec: ExecutionContext): Future[Unit] =
  Future(println(s"Hello, $name!"))
```

By making the `ExecutionContext` implicit we then avoid the need to explicitly pass it around in method calls:

```
def run(implicit ec: ExecutionContext): Future[Unit] =
  for {
    _ <- sayHello("Alice")
    _ <- sayHello("Bob")
  } yield ()
```

We can see here that by making the `ExecutionContext` implicit we avoided the need to explicitly pass it to the invocations of `sayHello`.

But notice that this only reduced the boilerplate of the method *invocations*, not the boilerplate of the method *declarations*. We didn't have to specify the `ExecutionContext` when we called `sayHello` in the `run` method but we still had to list it as an implicit parameter in the signature of `sayHello`.

With a single implicit parameter like an `ExecutionContext` this isn't too bad,

although it does mean that often every method signature will require an implicit `ExecutionContext`!

But what happens if we have twelve different services that we need to pass around?

Just like above in our discussion of tagless final, Scala does not allow us to abstract over implicit parameters so we have to list all twelve of these services in each method declaration, which is a lot of boilerplate!

Using implicits this way also raises a risk that it can be harder to tell where our services are actually coming from.

Normally a best practice is that there should only be one implicit instance of a type. But there might be multiple `ExecutionContext` values in a given application (e.g. one for asynchronous tasks and another for blocking tasks).

This can create the risk that we accidentally run tasks on the wrong `ExecutionContext`, resulting in lower performance or even deadlocks. It also makes refactoring harder, because copying code to another file could potentially change the implicit that is in scope and the meaning of the code.

17.3 ZIO As A Reader Monad On Steroids

ZIO's approach to dependency injection begins by taking the idea of the *reader monad* and baking it directly into the `ZIO` data type. This means that the `access` and `environment` methods are available on `ZIO` directly and there is no additional data type needed.

```
trait ZIO[-R, +E, +A] {
  def provide(r: R): ZIO[Any, E, A]
}

object ZIO {
  def environment[R]: ZIO[R, Nothing, R] =
    ???
}
```

This has several benefits to the approaches described above.

Since ZIO directly provides these operators we never need to use any other data types and directly have access to all the operators on ZIO we have been learning about. Performance is also better because we aren't using additional data types.

Code complexity is greatly reduced because there are no higher kinded types, type classes, or implicits being used here.

Perhaps most importantly, we have the ability to *abstract over services* in a way that we didn't with any of the other approaches. If we have an effect that

depends on the `Clock`, `Console`, and `Random` services we can create a type alias for that:

```
import zio.clock._
import zio.console._
import zio.random._

type MyEnv = Clock with Console with Random
```

We can then use that type alias anywhere we were previously listing all three services.

```
def effect1: ZIO[Clock with Console with Random, Nothing, Unit] =
  ???

def effect2: ZIO[MyEnv, Nothing, Unit] =
  effect1
```

In this small example this is already a nice simplification but you can imagine how much boilerplate this could save us in a large application with many dependencies

```
type Env = Logging with Database with Redis with Cassandra with Kafka with Persistence with
```

With that introduction, let's get down to how we use the environment type in our own applications. Fundamentally, there are two things we need to do with the environment:

1. **Access** services in the environment to do something with them in our own code, for example access a logging service so we can log a line.
2. **Provide** an effect with the environment it needs, such as a live database implementation, so that we can actually run it.

In the next two sections we will discuss each of those in turn.

17.4 Accessing The Environment

We access the environment to do something with it using the `access`, `accessM`, and `environment` operators.

The `environment` operator is in a way the most fundamental. Its signature is:

```
object ZIO {
  def environment[R]: ZIO[R, Nothing, R] =
    ???
}
```

The `environment` operator gets a service from the environment so that we can do something with it. For example, we might access a logging service in the environment so that we can log something.

```
def logLine(line: String): ZIO[Logging, Nothing, Unit] =
  ZIO.environment[Logging].flatMap(service => service.logLine(line))
```

The `access` and `accessM` operators just bundle the *getting* the service from the environment and the *doing something with it* into a single operator.

```
def access[R, A](f: R => A): ZIO[R, Nothing, A] =
  ZIO.environment.map(f)
def accessM[R, E, A](f: R => ZIO[R, E, A]): ZIO[R, E, A] =
  ZIO.environment.flatMap(f)
```

One thing that sometimes trips people up when initially learning about the environment is that when we use the `environment`, `access`, or `accessM` operators the service we are using doesn't exist yet.

For example, consider:

```
def logLine(line: String): ZIO[Logging, Nothing, Unit] =
  ZIO.environment[Logging].flatMap(service => service.logLine(line))
```

We are using the `Logging` service in our method implementation but we haven't provided it yet. The `Logging` in the environment type of the effect indicates that we still "owe" the effect a `Logging` service.

The `environment`, `access`, and `accessM` operators are allowing us to write a "check" against a service, saying "let me do something with this service and I promise you I will provide an actual implementation of it later". Except this time the Scala compiler is going to make sure we pay our debts because it is not going to let us run this effect without providing all the services it needs.

This idea of deferring providing services is really at the core of the dependency injection problem. We want to be able to use these services in writing our code as if they exist but we want to defer actually providing them to a higher level of our application, and the environment type lets us do just that.

One other thing to note is that these are some of the very few operators in ZIO that will often require specifying the environment type.

The reason is that when we call `environment` or `access` we could theoretically be accessing any service in the environment. We could be accessing `Logging`, `Database`, or something else entirely. Sometimes the type of environment being accessed could be deduced by the methods we are calling on it, but the Scala compiler generally works "left to right".

So be prepared to specify the type of service you are accessing when you use these operators.

17.5 Composition Of Environment Types

The other important thing to understand when accessing the environment is how the environmental requirements of different effects compose.

For example, say we have one effect that requires a `Database` service and one effect that requires a `Logging` service. If we compose them together, what services will the composed effect require?

```
trait Database
trait Logging

def getName(userId: Long): ZIO[Database, Throwable, String] =
  ???

def logLine(line: String): ZIO[Logging, Nothing, Unit] =
  ???

def getAndLogName(userId: Long): ZIO[Database with Logging, Throwable, String] =
  for {
    name <- getName(userId)
    _    <- logLine(s"Got $name associated with $userId from database")
  } yield name
```

The answer is that the environment type will be `Database with Logging`.

The `A with B` syntax in Scala describes an intersection type, a type that provides all the functionality of `A` as well as all the functionality of `B`. So the type `Database with Logging` describes an environment supporting both all the methods on `Database`, such as reading and writing to the database, as well as all the methods on `Logging`, such as logging a line.

This makes sense because to run `getAndLogName` we are ultimately going to have to call `getName`, which requires database functionality, and `logLine`, which requires logging functionality. So to run this entire effect we are going to need to provide a service that implements both database and logging functionality.

One helpful way of thinking of the environment is as the set of services that an effect requires. This has several implications.

The first is that just like the elements of a set, the order of services in the environment doesn't matter so we can reorder services in the type signature of the environment type as often as we like.

```
import zio.clock._
import zio.console._
import zio.random._

lazy val clockConsoleRandom: ZIO[Clock with Console with Random, Nothing, Unit] =
  ???
```

```
lazy val randomConsoleClock: ZIO[Random with Console with Clock, Nothing, Unit] =
  clockConsoleRandom
```

The second is that since sets don't have duplicates, including the same service in the signature of the environment multiple times is irrelevant and can be added or discarded.

```
lazy val console1: ZIO[Console, Nothing, Unit] =
  ???

lazy val console2: ZIO[Console with Console, Nothing, Unit] =
  console1

lazy val console3: ZIO[Console, Nothing, Unit] =
  console2
```

The third is that since `R` is contravariant, `Any` is an identity element with respect to `with`. So you can add or remove `Any` from intersection types in type signatures in the environment as much as you want and it will always mean the same thing.

For example:

```
lazy val effect1: ZIO[Console, Nothing, Unit] =
  ???

lazy val effect2: ZIO[Console with Any, Nothing, Unit] =
  effect1

lazy val stillTheSame: ZIO[Console, Nothing, Unit] =
  effect1
```

17.6 Providing An Effect With Its Required Environment

So far we have described everything we need to know to access services in the environment and use them, like accessing a database to write something to it. But how do we actually provide our effects with the services they require so we can run them?

The simplest way is the `provide` operator, which provides an effect with its *entire* required environment.

```
trait ZIO[-R, +E, +A] {
  def flatMap[R1 <: R, E1 >: E, B](f: A => ZIO[R1, E1, B]): ZIO[R1, E1, B]
  def map[B](f: A => B): ZIO[R, E, B]
  def provide(r: R): ZIO[Any, E, A]
}
```

For example, we could provide an effect that needs a `Database` with a live implementation like this:

```
trait Database

lazy val database: Database =
  ???

lazy val effect1: ZIO[Database, Nothing, Unit] =
  ???

lazy val effect2: ZIO[Any, Nothing, Unit] =
  effect1.provide(database)
```

There is also a `provideSome` operator that provides an effect with *part* of its required environment.

```
trait ZIO[-R, +E, +A] {
  def provideSome[R1](f: R1 => R): ZIO[R1, E, A]
}
```

However, this operator requires us to manually build the full set of services our effect requires from the set of services that our new effect will still require. For example, if we have an effect that requires both `Database` and `Logging` services we could use `provideSome` to provide just the `Database` service like this:

```
trait Database
trait Logging

lazy val effect3: ZIO[Database with Logging, Nothing, Unit] =
  ???

lazy val effect4: ZIO[Logging, Nothing, Unit] =
  effect3.provideSome { logging =>
    ???
    // we need to provide a way to create a `Database with Logging`
    // service given a `Logging` service here
  }
```

We will see how to combine services together in the next chapter, but to avoid having to do this ourselves we typically use another set of operators, `provideLayer` and `provideSomeLayer`, instead of using `provide` and `provideSome` directly.

The slightly simplified signature of `provideLayer` is:

```
trait ZIO[-R, +E, +A] {
  final def provideLayer[E1 >: E, R0](layer: ZLayer[R0, E1, R]): ZIO[R0, E1, A] =
    ???
}
```

The `provideLayer` operator is the first operator we have looked at that uses the `ZLayer` data type, so let's look at that now to understand it and how it fits into solving the dependency injection problem.

We will introduce `ZLayer` now and understand how to work with it as a user of existing services here. In the next chapter we will spend much more time on `ZLayer` including its internal implementation and how we can use it to build and compose our own services.

A `ZLayer[-RIn, +E, +ROut]` is a *recipe* for building a set of services. It requires a set of input services `RIn` and can either fail with an error of type `E` or succeed with a set of output services of type `ROut`.

As you can see from the type signature above, layers can themselves have their own dependencies, so we could for example have a `ZLayer[Logging, Nothing, Logging with Database]`, which would be a recipe for building a set of `Logging` and `Database` services given a `Logging` service. But we typically eventually want to have a layer where the `RIn` type is any, indicating that this layer does not have any further dependencies.

`ZLayer` is similar to `ZManaged` in that it is a specialized `ZIO` effect that is designed to solve the problems of a particular domain. While `ZManaged` was designed to solve the problem of resource handling, `ZLayer` is designed to solve the problem of building an application's required dependencies.

Because of this, almost everything you have learned about `ZIO` and `ZManaged` carries over to `ZLayer`.

In particular, layers are just *blueprints* for constructing services so no services are actually constructed until we *build* the layer, typically by using the `provideLayer` operator.

Going back to our example of providing our effect with the database service it needs, using `provideLayer` it would look like this:

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

lazy val effect5: ZIO[Database, Nothing, Unit] =
  ???

lazy val effect6: ZIO[Any, Nothing, Unit] =
  effect5.provideLayer(databaseLayer)
```

This is typically the way we will work with services in the environment in practice.

The implementor of the service will, in addition to defining the service interface, create one or more layers that build implementations of the service. As a user of the service, we will then use `provideLayer` to build the services that our application needs and provide them to our application.

One advantage of this is that it allows us to describe services where the creation of the service potentially uses effects and requires resources that must be safely acquired and released.

For example, so far we have been talking about the `Database` service as if it was just something that was globally available that we could call methods on. But in reality creating a live database service probably requires performing effects like opening a database connection and finalization like closing that connection.

`ZLayer` lets us describe all of those things using the existing tools we have learned about for `ZIO` and `ZManaged`.

We will learn more about this in the next chapter, but for now you can think of `provideLayer` as similar to the `use` operator on `ZManaged`. When we call `provideLayer` on an effect, the services required for the effect will be acquired immediately before the effect is executed and released immediately after the effect is completed, no matter how it completes.

Another advantage of using layers is that they make it much easier for us to provide only part of the environment using the `provideSomeLayer` operator.

Let's go back to the situation we discussed above where we have an effect that needs both a database service and a logging service. We want to provide a concrete implementation of the database service now, but we want to defer providing the logging service until a higher level of our application.

Using `ZLayer`, this problems becomes extremely simple:

```
import zio._

type Database = Has[Database.Service]
object Database {
  trait Service
}
type Logging = Has[Logging.Service]
object Logging {
  trait Service
}

lazy val databaseLayer: ZLayer[Any, Nothing, Database] =
  ???

lazy val effect1: ZIO[Database with Logging, Nothing, Unit] =
  ???

lazy val effect2: ZIO[Logging, Nothing, Unit] =
  effect1.provideSomeLayer[Logging](databaseLayer)
```

Now we don't have to provide any special logic for how to combine `Database` service with a `Logging` service to get a `Database with Logging` service. `ZLayer`

knows how to do that for us automatically.

There is some work we need to do in how we define services to make this possible, as we will see in the next chapter. The reward is though now partially providing layers like this is extremely convenient.

One thing to note about `provideSomeLayer` is that it is one of the few operators in ZIO that requires manually specifying the type parameter. Unfortunately, the Scala compiler can't eliminate the services we are providing from the set of services that the effect requires so we have to do it ourselves, though the compiler is able to verify the soundness of the type we specify.

When using the `provideSomeLayer` operator, the type you specify should be the type you are "leaving behind", that is, the type you are not providing.

For instance, in the example above we did `provideSomeLayer[Logging](databaseLayer)`. The `Database` service is the one that we are providing so the `Logging` service is the dependency that is still outstanding and the type we want to specify.

If you find yourself "leaving behind" the same environment multiple times you can create a specialized operator that always specifies that as the environment that you are leaving outstanding.

For example, ZIO includes a `provideCustomLayer` operator that is defined as `provideSomeLayer[ZEnv]`. This requires the caller to specify a layer that provides any services an effect requires that are not part of the default services implemented by ZIO.

```
import zio.clock._
import zio.console._

lazy val effect1: ZIO[Database with Logging with Clock with Console, Nothing, Unit] =
  ???

// provides all services that are not part of standard ZIO environment
lazy val customLayer: ZLayer[Any, Nothing, Database with Logging] =
  ???

// effect now only requires default services
lazy val effect2: ZIO[ZEnv, Nothing, Unit] =
  effect1.provideCustomLayer(customLayer)
```

Similarly, ZIO Test implements a `provideCustomLayer` variant that requires the user to specify a layer that provides all services that are not part of the `TestEnvironment` that ZIO Test provides.

17.7 Conclusion

This chapter has focused on describing the dependency injection problem, traditional approaches and their limitations, and ZIO's approach to dependency injection.

The primary focus on this chapter has been from the perspective of a *user* of services. We learned how to access services in the environment, such as a database service to write to it, how to combine effects that require different services, and how to ultimately provide an effect with the services it requires so it is ready to be run.

In the next chapter we will shift to focus on being the implementer of services. We have seen how convenient it can be to use services in the environment but how do we actually build these services?

Read on to find out!

17.8 Exercises

Chapter 18

Dependency Injection: Advanced Dependency Injection

In the last chapter we focused on dependency injection primarily from the perspective of the user of services. We learned how to access services in the environment in our effects, how to combine effects that require different services, and how to provide an effect with the services it requires to run.

But in the last chapter we focused more on the *what* versus the *how*. We learned about services and the environment but how are services actually defined in ZIO and how does this help us build services?

In this chapter we will go into that in detail. We will begin by describing the problem of building services and introduce the `Has` data type which is ZIO's solution to addressing this.

Then we will learn about the *module pattern*, which is the idiomatic ZIO approach to defining services to provide an extremely fluent API for users like the one we saw in the last chapter.

Finally we will spend more time on `ZLayer` and how we can use some of the more advanced operators on `ZLayer` to build complex sets of services in the most efficient and safe way possible.

18.1 Composing Services

In the last chapter we described services as *interfaces* describing certain functionality. For example, we might have a database service and a logging service with simplified interfaces like this:

```
import zio._

trait Database {
  def getUserName(id: Long): UIO[String]
}

trait Logging {
  def logLine(line: String): UIO[Unit]
}
```

We also said that the result of combining two effects that require different services is a new effect that requires both services, where *both* here is represented as an intersection type.

```
lazy val databaseEffect: ZIO[Database, Nothing, String] =
  ???

lazy val loggingEffect: ZIO[Logging, Nothing, Unit] =
  ???

lazy val combinedEffect: ZIO[Database with Logging, Nothing, String] =
  databaseEffect <* loggingEffect
```

This is extremely logical and represents the concept that if part of our effect requires the functionality of a database service and part of our effect requires the functionality of a logging service then to run our effect we need something that provides the functionality of both a database service and a logging service. But how do we go about actually creating a `Database with Logging`?

Ideally, if we already had a `Database` service and a `Logging` service we would want to be able to combine them with a simple operator:

```
lazy val databaseService: Database =
  ???

lazy val loggingService: Logging =
  ???
```

Unfortunately this doesn't work. `Database` and `Logging` are just traits and there is no method like `++` to combine traits generically.

We would like to be able to implement a method like this:

```
scala: mdoc def mix[A, B](a: A, b: B): A with B =   ???
```

Conceptually, if we have a value of type `A` and a value of type `B` we should be able to implement a type `A with B` by taking any calls to methods defined on `A` and delegating them to the `A` value and taking any calls to methods defined on `B` and delegating them to the `B` value.

But there is not a way to express this in Scala without using macros.

As a result, if we define services simply as interfaces like this to implement combined services we need to extend each service and manually reimplement each of its methods:

```
val combinedService: Database with Logging =
  new Database with Logging {
    def getUserName(id: Long): UIO[String] =
      databaseService.getUserName(id)
    def logLine(line: String): UIO[Unit] =
      loggingService.logLine(line)
  }
// combinedService: Database with Logging = repl.MdocSession$App$$anon$1@63bda6df
```

We are just doing what we described conceptually above with the `mix` method but we can see that this is already starting to feel like boilerplate.

And this is when we only have two services that each implement a single method. Imagine how much boilerplate there would be if we had many services each of which implements many methods.

We can clean this up slightly by defining each *module* to contain a *service*.

```
trait Database {
  def database: Database.Service
}

object Database {
  trait Service {
    def getUserName(id: Long): UIO[String]
  }
}

trait Logging {
  def logging: Logging.Service
}

object Logging {
  trait Service {
    def logLine(line: String): UIO[Unit]
  }
}
```

Now we can compose *modules* by implementing the *services* of both module:

```
lazy val databaseModule: Database =
  ???

lazy val loggingModule: Logging =
  ???
```

```
lazy val combinedModule: Database with Logging =
  new Database with Logging {
    val database: Database.Service =
      databaseModule.database
    val logging: Logging.Service =
      loggingModule.logging
  }
```

This is better in that we only need to reimplement each of the services when we compose modules, rather than each of the methods of the service. But there is still a lot of boilerplate here, especially if we are working with a larger number of services.

And there is a more fundamental problem here.

This works if we want to combine two concrete services, like `Database` and `Logging`. But it doesn't work if one of the services is abstract.

For example, if we have a `Logging` service and some other set of services `R` we would like to be able to say that we can create a service of type `Logging with R`. But we can't do that:

```
def enrichWithLogging[R](r: R, logging: Logging): R with Logging =
  new R with Logging {
    ???
  }
// error: class type required but R found
//   new R with Logging {
//       ^
// error: illegal inheritance; supertype R
//  is not a subclass of the superclass Object
//  of the mixin trait Logging
//   new R with Logging {
//               ^^^^^^^
// error: type mismatch;
//  found   : R with Logging
//  required: R with Logging
//   new R with Logging {
//   ^^^
```

This doesn't work for a couple reasons.

First, since `R` is completely abstract the Scala compiler doesn't know if it can be extended at all. `R` might be a `sealed trait` or `final case class` for example.

Second, since `R` is completely abstract we don't know what methods are defined on it so we have no way of implementing those methods in `R with Logging`.

The inability to implement methods like this is a significant problem because

we would like to be able to *partially provide* the environment. For example, we would like to be able to take an effect that requires a logging service and some other set of services and just eliminate the dependency on the logging service.

```
def provideLogging[R, E, A](zio: ZIO[R with Logging, E, A])(logging: Logging): ZIO[R, E, A]
  ???
```

But we can't implement this because we would need to use `provideSome` and provide a function `R => R with Logging`. But we just said that it is not possible to implement such a function generically.

18.2 The Has Data Type

These challenges led to the introduction of `ZLayer` and a data type that supports it called `Has`.

You can think of `Has` as a `Map` from types of services to service implementations. For example, `Has[Console]` would be represented as something like:

```
Map(ConsoleServiceType -> ConsoleServiceImplementation)
// res1: Map[ConsoleServiceType.type, ConsoleServiceImplementation.type] = Map(
//   repl.MdocSession$App$ConsoleServiceType$1$@4fd70f29 -> repl.MdocSession$App$ConsoleSer
// )
```

`ConsoleServiceType` is a `Tag`, which is like a `ClassTag` or a `TypeTag` from the Scala standard library in that it captures type information at runtime. However, it presents a consistent cross-version and cross-platform interface and supports parameterized types, making it more convenient than working with `ClassTag` and `TypeTag` directly.

The `Has` data type lets us solve the problem of composing multiple services discussed above because while we do not know how to combine two types, we know how to compose two maps!

To see how this works, let's think about what it would take to combine `Has[Database]` with `Has[Logging]`.

The `Has[Database]` and `Has[Logging]` services look something like this:

```
Map(DatabaseServiceType -> DatabaseServiceImplementation)
// res2: Map[DatabaseServiceType.type, DatabaseServiceImplementation.type] = Map(
//   repl.MdocSession$App$DatabaseServiceType$1$@7fb6042b -> repl.MdocSession$App$DatabaseS
// )
Map(LoggingServiceType -> LoggingServiceImplementation)
// res3: Map[LoggingServiceType.type, LoggingServiceImplementation.type] = Map(
//   repl.MdocSession$App$LoggingServiceType$1$@30057f60 -> repl.MdocSession$App$LoggingSer
// )
```

Given that, implementing `Has[Database] with Has[Logging]` becomes relatively straightforward, at least conceptually.

A `Has[Database]` is something that, if we ask it for a database service, can provide us with an actual implementation of that service. And a `Has[Logging]` is something that, if we ask it for a logging service, can give us an actual implementation of a logging service.

So then `Has[Database] with Has[Logging]` just means something that we can ask for a database service and get back a database service and also ask for a logging service and get a logging service.

We can implement this simply by combining the two maps. So `Has[Database] with Has[Logging]` can be implemented like this:

```
Map(
  DatabaseServiceType -> DatabaseServiceImplementation,
  LoggingServiceType -> LoggingServiceImplementation
)
// res4: Map[Object, Object] = Map(
//   repl.MdocSession$App$DatabaseServiceType$1$@7fb6042b -> repl.MdocSession$App$DatabaseS
//   repl.MdocSession$App$LoggingServiceType$1$@30057f60 -> repl.MdocSession$App$LoggingSer
// )
```

We can create this map quite easily by combining the keys and values from the maps underlying the two original `Has` values.

One important thing to keep in mind with `Has` is that unlike many data types it is *invariant*. This is necessary for technical reasons so that the supertype of `Has[A]` and `Has[B]` is `Has[A] with Has[B]` instead of `Has[A with B]`.

Most of the time this doesn't make a difference but occasionally it can require a bit more work to make the types line up so it is something to be aware of.

18.3 Best Practices For Creating A Fluent API

So we have gotten an understanding of some of the issues with combining environment types and sketched out how `Has` lays the ground work for a solution. But how do we put this together to actually implement a service?

To implement services, we recommend following a standard pattern called the *module pattern*. Here is what a typical example of a service would look like using the module pattern:

```
import zio._

object logging {

  // Module definition
  type Logging = Has[Logging.Service]

  object Logging {
```

```
    // Service definition
    trait Service {
      def logLine(line: String): UIO[Unit]
    }

    // Module implementation
    val console: ZLayer[Any, Nothing, Logging] =
      ZLayer.succeed {
        new Service {
          def logLine(line: String): UIO[Unit] =
            UIO.effectTotal(println(line))
        }
      }
  }

  // Accessor methods
  def logLine(line: String): URIO[Logging, Unit] =
    ZIO.accessM(_.get.logLine(line))
}
```

There are several parts to this pattern, which we will discuss here.

First, we typically define all of the functionality related to a service in its own namespace. Here, that is `package object logging`.

Second, we define the module itself as a type alias for `Has[Module.Service]`. This is key because the `Has` data type is what gives us the power to compose services seamlessly.

If you create a module and do not define it as a type alias of `Has` you are likely to cause significant pain for your users because your module will not compose with other modules or work with ZIO operators the way that your users expect.

Third, we create a "companion object" for the type alias we just defined with the definition of `object Logging`.

Inside this object we define a `Service` trait that defines the actual interface of our service.

In the same object we also typically provide layers that implement actual instances of the module. For example, here `console` is a layer that builds a simple logging implementation that just logs everything to the console.

Finally, back in the overall namespace for our module, it is very nice for our users if we implement a set of *environmental accessor* methods.

These look like the `logLine` method at the bottom of the example above. They just access the module in the environment and call the corresponding method on the service.

Because environmental accessors just call corresponding methods on the service, these accessors do not actually implement new functionality. However, they can make for a much nicer experience for your users.

Consider these two snippets:

```
for {
  _ <- logging.logLine("Doing something")
  _ <- logging.logLine("Doing something else")
  _ <- logging.logLine("Doing a third thing")
} yield ()
// res6: ZIO[logging.Logging, Nothing, Unit] = zio.ZIO$FlatMap@4aaff24b

import logging.Logging

for {
  _ <- ZIO.accessM[Logging](_.get.logLine("Doing something"))
  _ <- ZIO.accessM[Logging](_.get.logLine("Doing something else"))
  _ <- ZIO.accessM[Logging](_.get.logLine("Doing a third thing"))
} yield ()
// res7: ZIO[Logging, Nothing, Unit] = zio.ZIO$FlatMap@581060ef
```

Most people would agree that the first is much nicer API than the second. For one thing, each line in the first example is just much more concise.

But beyond that, the first example is much more declarative, focusing on *what* we want done instead of *how* we want to do it.

In the first example, all the user has to be concerned about is that there is this `Logging` service in the environment and they can use it by calling `logging.logLine` to log something. Users don't need to know a lot other than that your service provides these methods they can use and where to find the implementation to ultimately provide at the top of their application.

In the second example, the user has to access the service themself using `accessM`. Then they have to call the `get` method on `Has` to retrieve the service they want to work with from the `Has` data type, potentially requiring understanding `Has` and its implementation as well.

We want to provide the best possible API for our users so we recommend using the module pattern and providing environmental accessors whenever possible.

This requires a bit more work from the implementors of services but provides a much better experience for users. If you want to avoid writing the environmental accessors yourself you can even use the `@accessible` annotation from ZIO Macros to do this for you.

18.4 Vertical And Horizontal Composition

Now we know how to implement individual services in an idiomatic way that allows them to be composed. How do we do that composition in practice, especially when we have multiple services that may have dependencies on each other?

To answer that question, it is helpful to distinguish two ways that different services can be related.

First, services may compose *vertically*. This means that a higher level service *depends on* a lower level service.

For example, our implementation of the `Database` service might describe a database that is located on a remote service. In this case we might need to have a `Connection` service to even be able to create the `Database` service.

We call this type of composition *vertical* because higher level layers in our application that describe more complex logic closer to our problem domain depend on lower level layers that are closer to implementation details.

Second, services may compose *horizontally*, meaning that both services are required by a higher level of the application. In the running example we have been working with involving the `Database` and `Logging` services, these services composed *horizontally* because we needed them both but neither one of them had a further dependency on the other.

Of course, sometimes service can compose both vertically and horizontally. For example, our application might require a `Database` and a `Logging` service and then the `Database` service itself might also require the `Logging` service.

This distinction between vertical and horizontal composition is helpful because it allows us to relatively mechanically transform a logical description of the relationship between a set of dependencies into code using `ZLayer`.

As a first step, we represent each service in our application as a `ZLayer`, explicitly modeling any services that service directly depends on as part of the `RIn` type of each `ZLayer`. We don't need to worry about services that a service indirectly depends on through another service because we will pick those dependencies up in our layer building process.

To illustrate this, let's look at the example we described above where our application depends on a `Database` service and a `Logging` service and the `Database` service itself depends on having access to the `Logging` service.

We begin by implementing layers that describe how to construct each of these services. Since the `Database` layer depends on the `Logging` layer we make that dependency explicit in the type signature.

```
lazy val databaseLayer: ZLayer[Logging, Nothing, Database] =
  ???
```

```
lazy val loggingLayer: ZLayer[Any, Nothing, Logging] =
  ???
```

Next, we "wire up" the layers using two simple rules

If we are composing two or more layers horizontally, we use the `++` operator. For example, since our application needs both the database and logging services we combine them using `++`.

This creates a new `ZLayer` that requires the inputs of both layers and produces the outputs of both layers.

```
lazy val horizontallyComposed: ZLayer[Logging, Nothing, Database with Logging] =
  databaseLayer ++ loggingLayer
```

If we are composing two layers vertically, we use the `>>>` operator with the "lower level" layer or the layer that provides what the other layer needs on the left hand side.

For example, since the `Database` layer depends on the `Logging` layer, we combine them like so:

```
lazy val verticallyComposed: ZLayer[Any, Nothing, Database] =
  loggingLayer >>> databaseLayer
```

This returns a new layer that requires all the inputs of the first layer and produces all the outputs of the second layer. The first layer must produce all the services that the second layer requires.

You can think of `>>>` as like function composition for layers. If we have a layer that requires an `A` and produces a `B` and glue it together with a layer that requires a `B` and produces a `C`, we get new layer that requires an `A` and produces a `C`.

To complete the process, we must model all of the vertically and horizontally composed layers using `++` and `>>>`.

It generally doesn't matter what order we do this in.

For example, if we look at the type signature of `horizontallyComposed` we can see that it outputs both a `Database` service and a `Logging` service, which is what we want. But it still requires a `Logging` service as input, which we don't want.

We can solve this problem quite simply by feeding in the logging layer to satisfy that need for a `Logging` service.

```
lazy val horizontalAndThenVertical: ZLayer[Any, Nothing, Database with Logging] =
  loggingLayer >>> (databaseLayer ++ loggingLayer)
```

We can also do these operations in the opposite order. If we look at the signature of `verticallyComposed` we have a layer that outputs a `Database` service and

doesn't require any additional inputs, which we want, but also doesn't yet output the `Logging` service.

We can fix this by horizontally composing what we have so far with the layer that outputs the logging service:

```
lazy val verticalAndThenHorizontal: ZLayer[Any, Nothing, Database with Logging] =
  (loggingLayer >>> databaseLayer) ++ loggingLayer
```

At this point you might be wondering, are the ways of building layers described above tremendously inefficient because we are building the `loggingLayer` twice in each implementation?

The answer is no, because by default layers are *memoized*.

This means that if a reference to a layer appears more than once in the description of a layer, that layer will still only be created a single time. So in the example above, regardless of if we compose the layers vertically and then horizontally or horizontally and then vertically, the `dasebaseLayer` and `loggingLayer` will each be created only once.

One important thing to note here is for purposes of memoization two layers will be considered to be equal based on *reference identity*.

While this is not ideal, since layers describe effects, there is not otherwise a sensible way of determining whether two layers are equal other than relying on arbitrary labels, which create the risk of runtime errors if the same label is inadvertently used in multiple parts of the dependency graph.

To take advantage of memoization, make sure you always define your layer as a `val`:

```
// do this
lazy val loggingLayer: ZLayer[Any, Nothing, Logging] =
  ???

// not this
def loggingLayer: ZLayer[Any, Nothing, Logging] =
  ???
```

Memoization is an extremely nice feature when building our dependency graph because it allows us to focus on the underlying logic of the relationship between dependencies without worrying about whether this representation is optimized or not. `ZLayer` will take care of that for us.

If you ever don't want memoization, you can use the `ZLayer#fresh` operator to ensure that a separate copy of the service will be built.

```
lazy val freshExample: ZLayer[Any, Nothing, Logging with Database] =
  loggingLayer ++ (loggingLayer.fresh >>> databaseLayer)
```

Here the `databaseLayer` will be provided with its own copy of the logging service that is distinct from the one provided to the main application.

Layers have several other properties that make using layers to build dependency graphs a joy.

One is that layers will acquire services in parallel whenever possible. Whenever we have layers that are horizontally composed we can acquire the two layers in parallel, assuming they do not have further dependencies.

The `++` operator does this for us automatically, so we never have to explicitly worry about parallelism when constructing layers. We describe the logical relationship between layers using the `++` and `>>>` and ZIO will automatically parallelize the process of constructing the layers to the maximum extent possible.

Another is that layers safely handle resources, even in the face of potential parallel acquisition of resources and memoization.

If any layer has a finalizer associated with it, for example because it was constructed using `ZLayer.fromAcquireRelease` or one of the `ZLayer.fromManaged` variants, that finalizer will be run no matter what.

This applies regardless of whether the effect that the layer is being provided to completes successfully, fails, is interrupted, or the process of building the rest of the dependency graph itself fails or is interrupted.

`ZLayer` will also be sure to run these finalizers as quickly as possible while ensuring correctness, running finalizers for resources acquired in parallel concurrently and running finalizers for resources acquired sequential in reverse order of acquisition. If a layer is being shared across different layers of the dependency graph due to memoization `ZLayer` will also correctly handle that, ensuring that resources aren't released too early.

All of this allows us to facilitate a mode of reasoning where if we describe the finalization of each individual service correctly, using constructors like `ZLayer.fromAcquireRelease` and `ZLayer.fromManaged`, then `ZLayer` will automatically ensure that the finalization of the entire dependency graph is handled correctly.

18.5 Local Modification And Elimination

We already saw how we can *partially eliminate* parts of the environment using the `provideSome` operator.

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

lazy val effect: ZIO[Database with Logging, Nothing, Unit] =
  ???

lazy val partiallyEliminated: ZIO[Logging, Nothing, Unit] =
  effect.provideSomeLayer[Logging](databaseLayer)
```

The `provideSomeLayer` operator allows us to provide some of the services that an effect needs while leaving other services to be provided later.

In addition to being able to partially eliminate the environment, another power that `ZLayer` gives us is the ability to locally modify parts of the environment.

For example, consider a `Config` service that contains default settings to be used for our application, such as level of parallelism that should be used.

```
type Config = Has[Config.Service]

object Config {
  trait Service {
    def defaultParallelism: Int
  }

  val default: ZLayer[Any, Nothing, Config] =
    ZLayer.succeed {
      new Service {
        val defaultParallelism = 4
      }
    }
}
```

We might want to update the default parallelism for our application. One way we can do this is with the `ZLayer#update` operator, which allows us to update one of the services output by a layer.

```
val modified: ZLayer[Any, Nothing, Config] =
  Config.default.update[Config.Service] { config =>
    new Config.Service {
      val defaultParallelism = config.defaultParallelism * 2
    }
  }
// modified: ZLayer[Any, Nothing, Config] = Fold(
//   Managed(zio.ZManaged$$anon$2@737f7cad),
//   Managed(zio.ZManaged$$anon$2@28b37d35),
//   Managed(zio.ZManaged$$anon$2@2e18cb3)
// )
```

We can also do this directly with an effect using the `ZIO#updateService` operator:

```
lazy val zio: ZIO[Config, Nothing, Unit] =
  ???

lazy val withUpdatedConfig: ZIO[Config, Nothing, Unit] =
  zio.updateService[Config.Service] { config =>
    new Config.Service {
```

```
      val defaultParallelism = config.defaultParallelism * 2
    }
  }
```

Note that when we call `update` on `ZLayer` we are taking an existing `Config` service and are modifying it. When we call `updateService` on an effect we are saying that whatever service we provide to this effect is then going to be modified before running the effect.

18.6 Conclusion

In this chapter we have gone from being users of services implemented by others to being able to implement our own services to describe capabilities that others can incorporate into their applications.

We have learned about `Has` and how it allows us to compose environment types much more efficiently and generically than we could before. We also learned more about `ZLayer` and how it allows us to build complex environments with dependencies between layers, letting us focus on the logical relationships and automatically handling optimizations.

With these tools you should be ready to incorporate environmental services into your own applications to make your code more modular and easier for others to include as part of their own applications.

18.7 Exercises

Chapter 19

Software Transactional Memory: Composing Atomicity

In this chapter we will begin our discussion of *software transactional memory*. Software transactional memory is a tool that allows us to compose individual operations together and have the entire operation performed *atomically* as a single *transaction*.

STM is a tool that gives us superpowers to solve challenging concurrency problems. We will see in this chapter and the next how problems that seem quite challenging with other paradigms become incredibly simple using STM.

But before we do that, we need to understand a little more about the problem software transactional memory addresses and how ZIO's STM data type solves it.

19.1 Inability To Compose Atomic Actions With Other Concurrency Primitives

In our discussion of `Ref` we learned that modifications to a `Ref` are performed atomically. This means that from the perspective of other fibers getting the old value and setting the new value take place as a single operation.

This is critical for the correctness of concurrent code because otherwise other fibers can observe an inconsistent state where the fiber updating the `Ref` has gotten the old value but not set the new value.

For example, even simply using multiple fibers to increment a `Ref` will not work

correctly if we do not perform the modification atomically.

```
import zio._

for {
  ref       <- Ref.make(0)
  increment = ref.get.flatMap(n => ref.set(n + 1))
  _         <- ZIO.collectAllPar(ZIO.replicate(100)(increment))
  value     <- ref.get
} yield value
// res0: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@7fe1f8bc
```

Here we might expect `value` to be `100` because we executed one hundred effects in parallel that each incremented the `Ref` once. However, if we run this program repeatedly we will often find that the actual result is less.

Why is that?

In the example above the updates are not performed atomically because we used `get` and `set` separately instead of using `update` or `modify`. Remember with `Ref` individual operations are atomic but they do not compose atomically.

This means that if the current value of the `Ref` is `4`, for example, one fiber could get the value `4`, then another fiber could get the value `4`, and then each fiber would set the value `5`, resulting in one increment being "missed".

We can fix this by using the `update` or `modify` operators.

```
for {
  ref       <- Ref.make(0)
  increment = ref.update(_ + 1)
  _         <- ZIO.collectAllPar(ZIO.replicate(100)(increment))
  value     <- ref.get
} yield value
// res1: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@5547e31d
```

Now the updates takes place as a single transaction, so if one fiber gets the value `4` as part of the update operation, other fibers will not be able to observe that old value and instead will observe `5` once the first fiber has completed the update. Now `value` will always be `100`.

In this simple example all we had to do is replace `get` and `set` with `update`, but in even slightly more complex scenarios the lack of ability to compose atomic updates can be much more problematic.

For example, consider the classic example of transferring funds between two bank accounts.

```
def transfer(from: Ref[Int], to: Ref[Int], amount: Int): Task[Unit] =
  for {
    senderBalance <- from.get
```

```
    _              <- if (amount > senderBalance)
                        ZIO.fail(new Throwable("insufficient funds"))
                      else
                        from.update(_ - amount) *> to.update(_ + amount)
  } yield ()
```

As implemented, this method has serious bugs!

We get the sender balance in `from.get` on the first line and use that to determine whether there are sufficient funds and we should initiate the transfer. But `transfer` could be called on a different fiber at the same time.

In that case, if the account had an initial balance of $100 and both transfers were for $100, we could conclude that both transfers were valid, since the amount is less than the balance each time we call `transfer`. But this allows us to do two transfers and end up in a situation where the senders account has a negative balance.

This is not good!

And unfortunately, there is not an easy way to fix it using just `Ref`.

We need to maintain two pieces of state, the balance of the sender's account and the balance of the receiver's account. And we can't compose updates to those two pieces of state atomically.

The typical solution, if all we have to work with is `Ref`, is to try to have a single `Ref` that maintains all the state we need for the problem we are trying to solve. For example:

```
final case class Balances(from: Int, to: Int)

def transfer(balances: Ref[Balances], amount: Int): Task[Unit] =
  balances.modify {
    case Balances(from, to) =>
      if (amount > from)
        (Left(new Throwable("insufficient funds")), Balances(from, to))
      else
        (Right(()), Balances(from - amount, to + amount))
  }.absolve
```

Now we have fixed the concurrency bug. We perform all of our work within a single `modify` operation, so we have a guarantee that it will be performed atomically.

If the transfer amount is greater than the send balance we return the original balances unchanged and return a `Left` from modify to signal failure. Otherwise we return the updated balances and use `Right` to signal success.

This approach of maintaining one `Ref` with all the state we need to track works on the small scale. For example if we have forked two fibers and need to keep

track of whether each fiber has completed its work a single `Ref` containing two `Boolean` values for whether each fiber is done would be an excellent solution.

But it doesn't scale to situations where we have a large number of different pieces of state we need to maintain.

In the example above, we were supposed to be implementing a `transfer` method that would allow us to transfer funds between any two accounts. But the `transfer` method we actually implemented only supported transferring funds between the two particular account balances we maintained as part of our combined `Ref`.

Of course, the logical conclusion of this is just to put more pieces of state in the `Ref`. At the extreme, we could put the balance of every account in our system in a single `Ref`.

But this has serious costs and is not a good solution. By doing this we are essentially creating a single gigantic piece of shared state for our entire application.

This will have serious negative performance implications. Recall that each update is performed atomically, which means that when one fiber is in the process of updating the `Ref` no other fiber can be updating that `Ref`.

We want the update to be atomic with respect to the accounts that we are transferring to and from for correctness. But if we put all of the account balances into a single `Ref` no transfer can take place while another one is being processed.

This means that if one fiber is executing a transfer from Alice to Bob, another fiber can't execute a transfer from John to Jane. There is no logical reason for this and we can imagine that if we had an actual banking application with millions of users this would be crippling to performance.

So how to do we get around this? Do we have to discard everything we have learned about ZIO and go back to using `synchronized` to solve these problems?

No! This is exactly the problem that software transactional memory and ZIO's STM data type are designed to solve, and they will make handling these types of problems extremely simple and elegant.

19.2 Conceptual Description Of STM

Recall that `Ref` and its nonfunctional equivalent `java.util.concurrent.atomic.AtomicReference` were both based on *compare and swap* operations.

In our discussion above we talked about the get and set portions of an update operation being performed atomically as part of a single transaction, so that no other fiber can observe the old value while an update is in progress.

But that isn't actually implemented by making any fiber actually suspend. For performance reasons we just accept the possibility that there could be conflicting

updates and *retry* if there is a conflicting update.

This creates the guarantee that no conflicting updates will occur, since if they would we just retry.

Let's go back and see a simple implementation of this for `Ref`.

```
import java.util.concurrent.atomic.AtomicReference

trait Ref[A] {
  def update(f: A => A): UIO[Unit]
}

object Ref {
  def make[A](a: A): UIO[Ref[A]] =
    UIO.effectTotal {
      new Ref[A] {
        val atomic = new AtomicReference(a)
        def update(f: A => A): UIO[Unit] =
          UIO.effectTotal {
            var loop = true
            while (loop) {
              val old = atomic.get
              val updated = f(old)
              loop = !atomic.compareAndSet(old, updated)
            }
          }
      }
    }
}
```

In the implementation of `update`, we first get the old value from the `AtomicReference` and use it to compute the updated value. We then use the `compareAndSet` method on the 'AtomicReference.

If the value of the `AtomicReference` is still equal to the old value, that is, there have not been any conflicting updates, then we just set the new value and are done. If there has been a conflicting update then we just loop and try the whole thing over, getting the value from the reference again and checking the same way.

Thus, by retrying whenever there has been a conflicting update we have taken an operation that was originally not atomic and make it atomic to all outside observers.

Conceptually then, it seems like we should be able to atomically perform an update of two variables, such as the sender and receiver account balances, using the same strategy. We could:

1. Get both account balances

2. Compute the updated balances
3. Check if the current balances still equal the ones we got in the first step
4. If the balances are equal, set the updated balances, otherwise retry

And we could do a version of that manually. But doing it manually requires a lot of work, is error prone, and doesn't scale well.

This amounts to having to implement logic for compare and swap and synchronization for every one of these "hard" concurrency problems we face, which is what makes them hard in the first place and what we are trying to avoid. It also requires mixing code that describes concurrency logic with code that describes our business logic, making both harder to understand and debug.

What we would really like is a way to *compose* atomic updates. That is, if we can update the sender balance atomically by retrying in the event of conflicting updates and we can update the receiver balance atomically by retrying in the event of conflicting updates, we should be able to update both balances atomically by retrying if there are conflicting updates to either balance.

Unfortunately, we can't do this with the `update` method on `Ref` because it is just an arbitrary `ZIO` effect.

We don't have any ability to "look inside" that effect to know that it contains a compare and swap loop or the structure of that loop. So we don't have any ability to take two of these update operations and combine their compare and swap loops into a single loop.

What we need is something that is a *blueprint* for an atomic update. Just like we saw that ZIO makes it easier to combine effects because it is just a description and we can combine descriptions, if we had a description of an atomic update with the appropriate structure then we could combine two descriptions to describe a single atomic update of both variables.

19.3 Using STM

The `ZSTM` data type is exactly that. A `ZSTM[R, E, A]` describes a *transaction* that requires an environment `R` and may either fail with an `E` or succeed with an `A`

We will commonly use the type alias `STM` to describe transactions that do not require an environment.

```
import zio.stm._

type STM[+E, +A] = ZSTM[Any, E, A]
```

We can convert the *blueprint* of a transaction into an actual effect that describes running that transaction with the `commit` operator on `ZSTM`. But once we commit a transaction we just have an ordinary `ZIO` value that we can no longer compose

atomically with other transactions, so we want to make sure we have described the entire transaction first.

Fundamental to ZIO's implementation of software transactional memory is the `TRef` data type. A `TRef` is like a `Ref` except all of its operators return STM effects that can be composed atomically with other STM effects.

```
trait TRef[A] {
  def get: STM[Nothing, A]
  def modify[B](f: A => (B, A)): STM[Nothing, B]
  def set(a: A): STM[Nothing, Unit]
  def update(f: A => A): STM[Nothing, Unit]
}
```

As you can see, the interface of `TRef` is almost exactly the same as the one for `Ref` other than the fact that the operators return `STM` transactions instead of `ZIO` effects.

`TRef` is the building block for `STM` transactions because `TRef` values, also called *transactional variables*, are the ones we check for conflicting updates when we execute a transaction. Essentially, when we execute an `STM` transaction, the following happens:

1. We tentatively perform the transaction as written, getting values from each transactional variable as necessary and recording the results
2. We check whether the values of all transactional variables have been changed since we began executing the transaction
3. If no transactional variables have been changed, we set the values of all transactional variables to the tentative values we computed in the first step and are done
4. If any transactional variables have been changed, discard the tentative results we computed and retry the entire transaction beginning at the first step above

Using `TRef` can be even easier than working with `Ref` because we don't have to worry about ensuring we do all of our updates as part of a single operator to ensure atomicity. For example, the following works fine:

```
import zio._
import zio.stm._

for {
  ref       <- TRef.make(0).commit
  increment = ref.get.flatMap(n => ref.set(n + 1)).commit
  _         <- ZIO.collectAllPar(ZIO.replicate(100)(increment))
  value     <- ref.get.commit
} yield value
// res3: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@41d84545
```

Because `STM` transactions compose into a single atomic transaction, we can now

`get` the `TRef` and then `set` the `TRef` and the entire thing will be performed atomically as long as it is part of a single `STM` transaction.

Of course, we didn't really need the power of `STM` in that case because we could just use `Ref#update`, but let's see how much simpler the implementation of the `transfer` method can become with `STM`.

```
def transfer(from: TRef[Int], to: TRef[Int], amount: Int): STM[Throwable, Unit] =
  for {
    senderBalance <- from.get
    _             <- if (amount > senderBalance)
                       STM.fail(new Throwable("insufficient funds"))
                     else
                       from.update(_ - amount) *> to.update(_ + amount)
  } yield ()
```

This now works correctly as is because if the `from` transactional variable is changed while we are performing the transaction the entire transaction will automatically be retried.

One nice thing about working with `STM` is that you can choose when to commit transactions and convert them back to `ZIO` effects.

Returning `STM` transactions can give users more flexibility because they can then compose those transactions into larger transactions and perform the entire transaction atomically.

For example, several investors might want to pool money for a new venture, but each investor only wants to contribute if the others contribute their share. With the `transfer` method defined to return a `STM` effect, implementing this is extremely simple.

```
def fund(senders: List[TRef[Int]], recipient: TRef[Int], amount: Int): STM[Throwable, Unit]
  ZSTM.foreach_(senders)(sender => transfer(sender, recipient, amount))
```

This will transfer funds from each sender to the recipient account. If any transfer fails because of insufficient funds the entire transaction will fail and all transactional variables will be rolled back to their original values.

And the entire transaction will be performed atomically so there is no risk that funds will be transferred from one user when another user has not funded or that a sender will spend their money somewhere else at the same time, even now this composed transaction is significantly more complicated than the original `transfer` transaction.

On the other hand, using `commit` and returning a `ZIO` effect can sometimes allow us to "hide" the fact that we are using `STM` in our implementation from our users and provide a simpler interface if our users are not likely to want to compose them into additional transactions or we do not want to support that. For example:

```
final class Balance private (private[Balance] val value: TRef[Int]) { self =>
  def transfer(that: Balance, amount: Int): Task[Unit] = {
    val transaction: STM[Throwable, Unit] = for {
      senderBalance <- value.get
      _             <- if (amount > senderBalance)
                         STM.fail(new Throwable("insufficient funds"))
                       else
                         self.value.update(_ - amount) *>
                           that.value.update(_ + amount)
    } yield ()
    transaction.commit
  }
}
```

Now we have almost entirely hidden the fact that we use `STM` in our implementation from the user. The disadvantage is that now transfers cannot be composed together atomically, so for example we could not describe funding a joint venture as part of a single transaction with this definition of `transfer`.

When in doubt, we would recommend returning STM transactions in your own operators to give users the flexibility to compose them together. But it can be helpful to be aware of the second pattern if you need it.

Working with `STM` is generally as easy as working with `ZIO` because most of the operators you are already familiar with on `ZIO` also exist on `STM`. You might have noticed above that just in the example so far we have used operators including `flatMap`, `map`, and `foreach` on `ZSTM` and constructors including `ZSTM.fail`.

In general, there are `ZSTM` equivalents of almost all operators on `ZIO` except those that deal with concurrency or arbitrary effects. Here are some common useful operators on `STM`:

- `flatMap` for sequentially composing STM transactions, providing the result of one STM transaction to the next
- `map` for transforming the result of STM transactions
- `foldM` for handling errors in STM transactions and potentially recovering from them
- `zipWith` for sequentially composing two STM transactions
- `foreach` for sequentially composing many STM transactions

We will see in the next section why there are not operators on STM dealing with concurrency or arbitrary effects.

In addition to this, there are some operators that are specific to `STM`. The most important of these is the `retry` operator, which causes an entire transaction to be retried.

The transaction won't be retried repeatedly in a "busy loop" but only when one of the underlying transactional variables have changed. This can be useful to suspend until a condition is met.

```
def autoDebit(account: TRef[Int], amount: Int): STM[Nothing, Unit] =
  account.get.flatMap { balance =>
    if (balance >= amount) account.update(_ - amount)
    else STM.retry
  }
```

Now if there are sufficient funds in the account we immediately withdraw the specified amount. But if there are insufficient funds right now we don't fail, we just retry.

As described above, this retrying won't happen in a busy loop where we are wasting system resources as we check the same balance over and over while nothing has changed. Rather, the STM implementation will only try this transaction again when one of the transactional variables, in this case the account balance, has changed.

Of course, we still might need to continue retrying if the new account balance is not sufficient, but we will only retry when there is a change to the account balance. When we commit a transaction like this, the resulting effect will not complete until the effect has either succeeded or failed.

So for example we could do:

```
for {
  ref   <- TRef.make(0).commit
  fiber <- autoDebit(ref, 100).commit.fork
  _     <- ref.update(_ + 100).commit
  _     <- fiber.await
} yield ()
// res4: ZIO[Any, Nothing, Unit] = zio.ZIO$FlatMap@295036a0
```

The first time we try to run `autoDebit` it is likely that there will be insufficient funds so the transaction will retry, waiting until there is a change in the `TRef`.

When the account balance is incremented by one hundred the `autoDebit` transaction will try again and this time find that there are sufficient funds. This time the transaction will complete successfully allowing the fiber running it to complete as well.

19.4 Limitations of STM

While STM is a fantastic tool for solving hard concurrency problems, there are some limitations to be aware of.

The first and most important is that STM does not support arbitrary effects inside STM transactions.

As we have seen above, to implement atomic transactions we have to retry a transaction, potentially multiple times, if there are updates to any of the

underlying transactional variables.

This is fine if the STM transaction is just doing pure computations, such as adding two numbers, or changing the value of transactional variables. If there is a conflicting change, we can just throw that work away and do it again and no one will ever be able to observe whether we were able to complete that transaction immediately or had to repeat it a hundred times.

But consider what would happen if we allowed an arbitrary effect, such as this inside an STM transaction:

```
val arbitraryEffect: UIO[Unit] =
  UIO.effectTotal(println("Running"))
// arbitraryEffect: UIO[Unit] = zio.ZIO$EffectTotal@694446d7
```

We would run this effect when we tried to execute the STM transaction, printing "Running" to the console. If we had to retry this transaction we then print "Running" again each time we executed the transaction.

This would create an observable difference between whether an effect was retried or not, which would violate the guarantee that to outside observers it should be as if the entire transaction was performed once as a single operation.

In this case just printing a line to the console is not the end of the world an in fact adding observable side effects like this can be useful for debugging STM transactions. But in general allowing arbitrary side effects to be performed within STM transaction would make it impossible to reason about transactions because we would not know how many times these transactions would be performed.

For the same reason, you will not see operators related to concurrency or parallelism defined on `ZSTM`.

Forking a fiber is an observable effect and if we forked a fiber within an STM transaction we would potentially fork multiple fibers if the transaction was retried. In addition to this, there is not a need for concurrency within an STM transaction because STM transactions are not performing arbitrary effects and there is typically no need to perform pure computations concurrently.

Note that while we can't perform concurrency *within* an STM transaction, we can easily perform multiple STM transactions concurrently.

For example:

```
for {
  alice     <- TRef.make(100).commit
  bob       <- TRef.make(100).commit
  carol     <- TRef.make(100).commit
  transfer1 = transfer(alice, bob, 30).commit
  transfer2 = transfer(bob, carol, 40).commit
  transfer3 = transfer(carol, alice, 50).commit
  _         <- ZIO.collectAllPar_(List(transfer1, transfer2, transfer3))
```

```
} yield ()
// res5: ZIO[Any, Throwable, Unit] = zio.ZIO$FlatMap@5697e709
```

Now the three fibers will separately execute each of the transactions. Each individual transaction will still be performed atomically, so we know that there is no possible of double spending and the account balances will always be accurate.

There are some categories of effects that could theoretically safely be performed safely within an STM transaction.

First, effects could be safely performed within an STM transaction if they were *idempotent*. This means that doing the effect once is the same as doing it many times.

For example, completing a `Promise` would be an idempotent operation. If we have a `Promise[Int]`, calling `promise.succeed(42).ignore` will have the same effect whether we perform it once or a hundred times.

A promise can only hold a single value and once completed trying to complete it again has no effect. So completing a promise with the same value multiple times has no effect other than using our CPU and in fact it would be impossible for a third party to observe whether there had been multiple attempts to complete a promise.

As a result, effects that are idempotent could theoretically be included in STM transactions. This feature is not currently supported but there has been some discussion about adding it.

Second, effects could be safely performed within an STM transaction if they were defined along with an *inverse* that reversed any otherwise observable effects of the transaction.

For example, we could imagine an effect describing inserting an entry into a database and an effect describing deleting the same entry. In this case it would be possible when retrying the transaction to run the inverse effect so that the transaction could safely be repeated.

ZIO's STM data type does not currently support this feature but if you are interested in learning more about this pattern you can check out the ZIO Saga library (https://github.com/VladKopanev/zio-saga).

The second main limitation of STM is that it retries whenever there are conflicting updates to any of the underlying transactional variables. So in situations of high contention it may need to retry a large number of times, causing a negative performance impact.

It is easy to think of STM as a "free lunch" with the way it allows us to easily solve many hard concurrency problems. But it is important to remember that it is still ultimately implemented in terms of retrying if there are changes to any of the transactional variables.

One way you can help avoid running into this is be conscious about the number of transactional variables you are including in an STM transaction. STM can support a large number of transactional variables, but if you have a very large number of transactional variables that are also being updated by other fibers you may have to retry many times since a transaction will retry if any of the transactional variables involved are updated.

For example, consider the following snippet:

```
def chargeFees(accounts: Iterable[TRef[Int]]): STM[Nothing, Unit] =
  STM.foreach_(accounts)(account => account.update(_ - 20))

for {
  accounts <- STM.collectAll(STM.replicate(100000)(TRef.make(100)))
  _        <- chargeFees(accounts)
} yield ()
// res6: ZSTM[Any, Nothing, Unit] = zio.stm.ZSTM@7e967ffd
```

Here `chargeFees` describes a single transaction that will deduct fees from all of the specified accounts. We then call this operator on a collection of 100,000 different accounts.

This is probably not the best idea.

Because we are describing this as a single transaction, if the balances of any of the 100,000 accounts are changed while we are executing `chargeFees` we will have to retry the entire transaction. If there are many other transactions being performed involving these same accounts the likelihood that we will have to retry many times is high.

In addition, in this case it does not appear that we actually need the transactional guarantee.

In the funding example, each investor needed to be sure that the other investors had funded to fund themselves. But here it seems like the fees are independent and we can go ahead and charge fees to one account even if we may not have charged fees to another account yet, for example because another transaction involving that account was ongoing.

If you expect to face circumstances involving very high contention despite this, you may want to benchmark STM versus other concurrency solutions.

STM is an *optimistic* concurrency solution in that it assumes there will not be conflicting updates and retries if necessary. If there are a very high number of conflicting updates then it may make sense to explore other concurrency solutions such as a `Queue` or using a `Semaphore` to guard access to impose sequencing more directly.

19.5 Conclusion

STM is an extremely powerful tool for solving hard concurrency problems. STM automatically retries if there are changes to any of the variables involved so it allows us to program "as if" our code was single threaded while still supporting concurrent access.

Code written with STM also never deadlocks, which can be a common problem when working with lower level primitives such as locks or synchronization.

19.6 Exercises

Chapter 20

Software Transaction Memory: STM Data Structures

In addition to the STM functionality described in the previous chapter, ZIO also comes with a variety of STM data structures, including:

- TArray
- TMap
- TPriorityQueue
- TPromise
- TQueue
- TReentrantLock
- TSemaphore
- TSet

It is also very easy to create your own STM data structures, as we will see.

There are a couple of things that are helpful to keep in mind as we learn about these data structures.

First, STM data structures represent versions of *mutable* data structures that can participate in STM transactions.

When we call operators on STM data structures we want to return versions of those data structures with those updates applied in place rather than new data structures. So the operators we define on STM data structures will look more like the operators on *mutable* collections rather than the standard *immutable* collections we deal with.

Second, all STM data structures are defined in terms of one or more `TRef` values.

As we said in the previous chapter, `TRef` is the basic building block of maintaining state within the context of an STM transaction so there is really nothing else.

Typically, the simplest implementation will just wrap an appropriate existing data structure in a single `TRef`. This will yield a correct implementation but may not yield the most performant one because it means we need to retry an update to a data structure whenever there is a conflicting update, even if the update is to a different part of the data structure.

For example, if we implemented a `TArray` as a single `TRef` wrapping an existing data structure supporting fast random access such as a `Chunk`, we would have to retry if the value at index `0` was updated while a the value at index `1` was in the process of being updated. But that is not really necessary for correctness because the values at the two indices are independent of each other.

As a result, we will see more complex implementations that use a collection of transactional variables to try to describe the parts of the data structure that can be updated independently as granularly as possible.

In some cases such as with a `TArray` where the value at each index is independent this will be quite straightforward. In other cases such as a `TMap` where values at different keys are theoretically independent but practically related through the underlying implementation of the map it may be much more complex.

In some of these cases the current representation may involve implementation specific details that are subject to change in the future, so we will try to focus on concepts rather than particular implementation details for some of these structures.

Existing implementations of STM data structures may also be useful for implementing new STM data structures. For example, several STM data structure are implemented in terms of other STM data structures as we will see.

20.1 Description Of STM Data Structures

With that introduction, let's dive into each of the STM data structures currently provided by ZIO.

For each data structure we will provide a high level description of the data structure, discuss what it should be used for, and review key operators.

When appropriate we will also review implementation so that you can see that there is nothing magic in implementing these data structures and in fact in most cases they are dramatically simpler than implementing new data structures from scratch. This will put us in a good position to demonstrate how we can implement our own STM data structure later in this chapter.

20.1.1 TArray

A `TArray` is the STM equivalent of a mutable array. A `TArray` has a fixed size at creation that can never be changed and the value at each index is represented by its own `TRef`.

Because the size of a `TArray` is fixed, the values at each index are fully independent from each other. This allows for an extremely efficient representation as an `Array[TRef[A]]`:

```
import zio.stm._

final class TArray[A] private (private val array: Array[TRef[A]])
```

Both the constructor and the underlying `Array` are `private` because the `Array` is a mutable structure that we do not want users accessing other than through the operators we provide.

Use a `TArray` any time you would normally use an `Array` but are in a transactional context. In particular, `TArray` is great for fast random access reads and writes.

If your use case can fit `TArray` it is an extremely good choice of STM data structure because the value at each index is represented as its own transactional variables so a transaction will only need to retry if there is a conflicting change to one of the indices being referenced in that transaction.

You can create a `TArray` using the `make` constructor, which accepts a variable arguments list of elements, or the `fromIterable` constructor, which accepts an `Iterable`.

The most basic methods on `TArray` are `apply`, which accesses the value at a specified index, and `update`, which updates the value at an index with the specified function. You can also determine the size of the array using the `size` operator.

One thing to note is that since the size of the `TArray` is statically known, accessing a value outside the bounds of the array is considered a defect and will result in a `Die` failure, so although the return signature of `apply` is `STM[Nothing, Unit]` you are still responsible for only attempting to access values within the bounds.

Let's see how we could use these methods to implement a `swap` operator on `TArray` that swaps the elements at two indices in a single transaction.

```
import zio.stm._

def swap[A](array: TArray[A], i: Int, j: Int): STM[Nothing, Unit] =
  for {
    a1 <- array(i)
    a2 <- array(j)
    _  <- array.update(i, _ => a2)
```

```
    _  <- array.update(j, _ => a1)
  } yield ()
```

We see again how simple it is to write these operators with STM since we don't have to worry about concurrent updates to the data structure while we are modifying it and can focus on just describing the transformation we want to make.

The other nice thing about this implementation is that it only references the `TRef` values at the specified index. So if we are swapping the values at indicates `0` and `1` for example another fiber could be swapping the values at indices `2` and `3` at the same time and neither transaction would have to retry.

In addition to these operators, `TArray` implements a wide variety of operators that reduce the values in the `TArray` to a summary value in some way. These include operators like `collectFirst`, `contains`, `count`, `exists`, `find`, `fold`, `forall`, `maxOption`, and `minOption`.

There are also operators to transform the `TArray` to other collection types, `toList` and `toChunk`. These operators are akin to taking a "snapshot" of the state of the `TArray` at a point in time and capturing that as a `Chunk` or a `List`.

Finally, there are various operations that work with particular indices of the `TArray`, such as `indexOf` and `indexWhere`.

Generally, any method on `Array` that reduces the array to a summary value, accesses an element of the array, or updates an element of the array should have an equivalent for `TArray`.

One question that people often have about `TArray` is where the `map` method is. This is an example of the point made at the beginning of the chapter that STM data structure generally represent STM equivalents of *mutable* data structure rather than *immutable* ones.

The `map` operator is one that takes each element of a collection and returns a *new* collection with the results of transforming each element of the original collection with the specified function.

That doesn't really apply to STM data structures because we don't want to return a new STM data structure, we want to update in place the values in the existing STM data structure. As a result, ZIO instead provides the `transform` operator for this use case.

```
val transaction = for {
  array <- TArray.make(1, 2, 3)
  _     <- array.transform(_ + 1)
  list  <- array.toList
} yield list
// transaction: ZSTM[Any, Nothing, List[Int]] = zio.stm.ZSTM@f9d849c
```

```
transaction.commit
// res1: zio.ZIO[Any, Nothing, List[Int]] = zio.ZIO$Read@146174c5
```

20.1.2 TMap

A `TMap` is a mutable map that can be used within a transactional context.

The internal implementation of `TMap` is relatively complicated because of the desire to separate out individual key and value pairs into separate transactional variables as much as possible but also the fact that all values are part of the same underlying map implementation. As a result, we will focus primarily on the operators on `TMap`, which are fortunately quite straightforward.

Use a `TMap` any time you would otherwise use a mutable map but are in a transactional context. In particular, `TMap` is good for when you want to be able to access values by a key other than their index and want the ability to dynamically change the size of the data structure by adding and removing elements.

The operators on `TMap` generally mirror those defined on a mutable `Map` from the Scala standard library except that they return their results in the context of STM effects.

You can create a new `TMap` with the `make` or `fromIterable` constructor, as well as the `empty` constructor that just creates a new empty `TMap`.

The primary operators specific to `TMap` are `delete`, `get`, and `put`.

```
trait TMap[K, V] {
  def delete(k: K): STM[Nothing, Unit]
  def get(k: K): STM[Nothing, Option[V]]
  def put(k: K, v: V): STM[Nothing, Unit]
}
```

Once again, one of the nice things about working with STM is that we can compose operations and never have to worry about creating concurrency issues by doing so. For example, here is how we can implement a `getOrElseUpdate` operator:

```
def getOrElseUpdate[K, V](map: TMap[K, V])(k: K, v: => V): STM[Nothing, V] =
  map.get(k).flatMap {
    case Some(v) => STM.succeed(v)
    case None    => STM.succeed(v).flatMap(v => map.put(k, v).as(v))
  }
```

With other data structures we might have to worry about composing `get` and `put` together this way and need a separate `getOrElseUpdate` operator to make sure we were doing this atomically. But since we are using STM this just works.

In addition to these fundamental operations `TMap` supports many of the other basic operations you would expect on a map, such as:

- `contains`. Check whether a key exists in the map.
- `isEmpty`. Check whether the map is empty.
- `fold`. Reduce the bindings in the map to a summary value.
- `foreach`. Perform a transaction for each binding in the map.
- `keys`. Returns the keys in the map.
- `size`. Returns the current size of the map.
- `toMap`. Take a "snapshot" of the keys and values in the map.
- `values`. Returns the values in the map.

20.1.3 TPriorityQueue

A `TPriorityQueue` is a mutable priority queue that can be used in a transactional context. A priority queue is like a normal queue except instead of values being taken in first in, first out order values are taken in the order of some `Ordering` defined on the values.

`TPriorityQueue` can be extremely useful when you want to represent a queue that needs to support concurrent offers and takes and where you always want to take the value that is "smallest" or "largest".

For example, you could use a `TPriorityQueue` to represent an event queue where each event contains a time associated with the event and an action to take to run the event, using time as the ordering for the `TPriorityQueue`.

Multiple producers could then offer events to the event queue while a single consumer would sequentially take events from the queue and run them. Since values are always taken from the `TPriorityQueue` in order, the consumer would always run the earliest event in the queue first, ensuring the correct behavior in a very straightforward way.

To construct a `TPriorityQueue` we use the `make`, `fromIterable`, or `empty` constructors we have seen before except this time we must also provide an implicit `Ordering` that will be used for the values.

```
import zio._

final case class Event(time: Int, action: UIO[Unit])

object Event {
  implicit val EventOrdering: Ordering[Event] =
    Ordering.by(_.time)
}

for {
  queue <- TPriorityQueue.empty[Event]
} yield queue
// res2: ZSTM[Any, Nothing, TPriorityQueue[Event]] = zio.stm.ZSTM@52a55992
```

A `TPriorityQueue` supports the typical operations you would expect of a queue.

You can offer one or more values to the queue using the `offer` and `offerAll` operators.

You can take values from the queue using the `take`, `takeAll`, or `takeUpTo` operators.

The `take` operator will take a single value from the queue, retrying until there is at least one value in the queue.

The `takeAll` operator will take whatever values are currently in the queue, returning immediately. The `takeUpTo` operator is like `takeAll` in that it will always return immediately, but it will only take up to the specified number of elements from the queue.

There is also a `takeOption` operator that will always return immediately, either taking the first value in the queue if it exists or otherwise returning `None`.

You can also observe the first value in the queue without removing it using the `peek` or `peekOption` operators.

The `peek` operator will return the first value in the queue without removing it, retrying until there at least one value in the queue. The `peekOption` operator is like `peek` but it will always return immediately, returning `None` if there are no values currently in the queue.

You can also check the size of the queue with the `size` operator and take a snapshot of the current contents of the queue with the `toList` and `toChunk` operators.

The `TPriorityQueue` also provides a couple of other operators to filter the contents of the queue such as `removeIf` and `retainIf` that remove all elements of the queue satisfying the specified predicate or not satisfying the specified predicate, respectively.

We will not spend more time on the implementation of `TPriorityQueue` here because we are actually going to implement it ourselves later in this chapter in the section on implementing our own STM data structures.

20.1.4 TPromise

A `TPromise` is the equivalent of the `Promise` data type in ZIO for the transactional context.

Use a `TPromise` whenever you would normally use a `Promise` to synchronize work between different fibers but are in a transactional context.

Remember that a single transaction can't involve concurrent effects like forking fiber but you can have multiple transactions involving the same transactional variables executed on multiple fibers. So you could await the completion of a `TPromise` in a transaction being executed on one fiber and complete the `TPromise` in a different transaction being executed on another fiber.

The implementation of `TPromise` is very simple and is another example of the power of STM.

```
final class TPromise[E, A] private (private val ref: TRef[Option[Either[E, A]]])
```

A `TPromise` is simply a `TRef` that is either empty, meaning the promise has not been completed, or completed with either a value of type `A` or an error of type `E`.

Let's see how simple it is to implement the basic interface of a promise using just this representation.

```
import zio.stm._

final class TPromise[E, A] private (private val ref: TRef[Option[Either[E, A]]]) {
  def await: STM[E, A] =
    ref.get.flatMap {
      case Some(value) => STM.fromEither(value)
      case None        => STM.retry
    }
  def done(value: Either[E, A]): STM[Nothing, Unit] =
    ref.get.flatMap {
      case None => ref.set(Some(value))
      case _    => STM.unit
    }
  def poll: STM[Nothing, Option[Either[E, A]]] =
    ref.get
}
```

We were able to implement a promise that supports waiting for a value to be set without ever blocking any fibers and without any polling in a couple lines of code.

One thing to note here is that when working with ZIO effects we talked about `Ref` and `Promise` as being two separate concurrency primitives that reflect separate concerns of shared state and work synchronization. But we see here that when working with STM only `TRef` is primitive and implementing waiting is trivial.

This is a reflection of the power of the `retry` operator on `STM`. The `retry` operator lets us retry a transaction until a condition is met without blocking and without any busy polling, retrying only when one of the underlying transactional variables changes.

With the `retry` operator it is extremely easy to implement operators that suspend until a specified condition is met, so we see that with `STM` implementing `Promise` was trivial, whereas with `Promise` an efficient implementation involves considerable work.

There is really not much more to say about `TPromise` than this.

We can create a new `TPromise` using the `make` constructor and there are `succeed` and `fail` convenience methods to complete a `TPromise` with a value or an error

instead of calling `done` directly. But there isn't much more to implementing `TPromise` with the power of STM.

20.1.5 TQueue

A `TQueue` is a mutable queue that can be used in a transactional context.

It is similar to the `TPriorityQueue` we discussed above except elements are always taken in first in first out order rather than based on some `Ordering` of the elements. It also supports bounded queues.

Once again, the implementation is quite simple.

```
import scala.collection.immutable.{ Queue => ScalaQueue }

final class TQueue[A] private (
  private val capacity: Int,
  private val ref: TRef[ScalaQueue[A]]
)
```

A `TQueue` is just a `scala.collection.immutable.Queue` wrapped in a `TRef`. We maintain the size as a separate value because it will never change so we don't need to store it in the `TRef`.

This is an example of the principle we talked about at the beginning of the chapter that we can almost always create a STM version of a data type by wrapping an appropriate immutable data type in a `TRef`.

If different parts of the data structure can be modified independently then we may be able to gain efficiency by splitting those parts into their own transactional variables. But we will be correct either way.

In the case of a `TQueue` the simple representation in terms of a single `TRef` makes sense because offering a value to the queue or taking a value from the queue changes the entire queue in terms of what value should be taken next and what values are remaining, so there are limited opportunities to modify different parts of the data structure separately.

Once again, lets see how simple it can be to define some of the basic queue operations using this representation.

```
import scala.collection.immutable.{ Queue => ScalaQueue }

final class TQueue[A] private (
  private val capacity: Int,
  private val ref: TRef[ScalaQueue[A]]
) {
  def offer(a: A): STM[Nothing, Unit] =
    ref.get.flatMap { queue =>
      if (queue.size == capacity) STM.retry
```

```
      else ref.set(queue.enqueue(a))
    }
  def peek: STM[Nothing, A] =
    ref.get.flatMap { queue =>
      if (queue.isEmpty) STM.retry
      else STM.succeed(queue.head)
    }
  def poll: STM[Nothing, Option[A]] =
    ref.get.flatMap { queue =>
      if (queue.isEmpty) STM.succeed(None)
      else queue.dequeue match {
        case (a, queue) => ref.set(queue).as(Some(a))
      }
    }
  def take: STM[Nothing, A] =
    ref.get.flatMap { queue =>
      if (queue.isEmpty) STM.retry
      else queue.dequeue match {
        case (a, queue) => ref.set(queue).as(a)
      }
    }
}
```

The implementations are all straightforward building on our ability to atomically compose `get` and `set` and the ability to retry. We use `ScalaQueue` as an alias for `scala.collection.immutable.Queue` to avoid ambiguity with the `TPriorityQueue` itself and the `Queue` data type in ZIO.

In `offer` we just get the current state of the `ScalaQueue` from the `TRef`. If the queue is already at capacity we retry, otherwise we call `enqueue` on the `ScalaQueue` to get a new `ScalaQueue` with the value added and set the `TRef` to that value.

In `peek` we again retry if the queue is empty and otherwise just return the first value in the queue. The `take` operator is similar except this time we use `dequeue` to actually remove the first element from the `ScalaQueue` instead of just accessing it.

The `poll` operator is even simpler. We don't have to retry at all because this operator should return immediately, so we just return `None` if the `ScalaQueue` is empty or else dequeue an element and set the `TRef` to the new `ScalaQueue` with the element removed.

The `TQueue` has other convenience methods for working with the queue, such as `isEmpty` and `isFull` to introspect on the state of the queue, and various versions of `offer` and `take`, but hopefully you can see from the implementation above how easy it is to add your own operators if you want to.

To construct a queue you use either the `bounded` or `unbounded` constructors,

which create a new empty `TQueue` with the specified capacity.

20.1.6 TReentrantLock

A `TReentrantLock` is the first STM data structure we have covered that is not a version of an existing data type in either ZIO or the Scala standard library. A `TReentrantLock` is like a `java.util.concurrent.locks.ReentrantLock` designed to work in the context of STM transactions.

A `TReentrantLock` supports four fundamental methods.

```
trait TReentrantLock {
  val acquireRead: STM[Nothing, Int]
  val acquireWrite: STM[Nothing, Int]
  val releaseRead: STM[Nothing, Int]
  val releaseWrite: STM[Nothing, Int]
}
```

The `Int` in the return type describes the number of read or write locks outstanding after the transaction. Typically we won't need this but it can be useful for some more advanced use cases.

A `TReentrantLock` is somewhat similar to a `Semaphore` in that it is typically used to guard access to some resource within concurrent code. However, a `TReentrantLock` maintains separate concepts of a access to *read from* the resource and access to *write to* the resource.

This distinction between readers and writers is important because it is safe for many different fibers to read from a resource at the same time because none of them are updating it so there is no possibility of conflicting updates.

In fact, it is even safe for multiple fibers to read from a resource at the same time as only one fiber is writing to it. Each of the fibers reading will access the current state as of a point in time between writes, but there is no risk of conflicting writes as long as only a single fiber has write access.

A `TReentrantLock` builds on this idea by allowing an unlimited number of fibers at a time to acquire read locks but only one fiber at a time to acquire a write lock.

One of the key things to keep in mind with `TReentrantLock`, as well as some of the other STM data structures such as `TSemaphore`, is that there is no need to use these data structures purely within the context of a single STM transaction, but they can be very useful within ZIO effects and the fact that they are transactional gives us more ability to compose them.

Let's break down a little more what this means.

The point of the `TReentrantLock` is to prevent multiple fibers from writing to the same resource. But we said in the last chapter when we discussed limitations

of STM that STM does not support concurrency within the context of a single STM transaction.

So for example, there is no point in doing this:

```
for {
  lock  <- TReentrantLock.make
  ref   <- TRef.make(0)
  _     <- lock.acquireWrite
  _     <- ref.update(_ + 1)
  _     <- lock.releaseWrite
  value <- ref.get
} yield value
// res4: ZSTM[Any, Nothing, Int] = zio.stm.ZSTM@1716e764
```

There is no point in using a lock within a single STM transaction like this because there is never more than one fiber executing a single STM transaction and if there were any conflicting changes made to one of the transactional variables by a transaction in another fiber the entire transaction would be retried.

Rather, where `TReentrantLock` can be useful is where we want to perform ZIO effects where we need to guard access to a resource. `TReentrantLock` actually has a couple helper methods to facilitate this:

```
import zio._

trait TReentrantLock {
  val acquireRead: STM[Nothing, Int]
  val acquireWrite: STM[Nothing, Int]
  val releaseRead: STM[Nothing, Int]
  val releaseWrite: STM[Nothing, Int]

  val readLock: Managed[Nothing, Int] =
    Managed.make(acquireRead.commit)(_ => releaseRead.commit)

  val writeLock: Managed[Nothing, Int] =
    Managed.make(acquireRead.commit)(_ => releaseRead.commit)
}
```

We can see that `readLock` and `writeLock` are much like the `withPermitManaged` in that they return a managed effect that describes acquiring and then releasing a lock. Both of these operators use `commit` to run the STM transactions to ordinary ZIO effects.

```
var bankAccount = 0
// bankAccount: Int = 0

for {
  lock <- TReentrantLock.make.commit
```

```
  zio1  = lock.writeLock.use(_ => ZIO.effectTotal(bankAccount += 100))
  zio2  = lock.writeLock.use(_ => ZIO.effectTotal(bankAccount += 50))
  zio3  = lock.writeLock.use(_ => ZIO.effectTotal(bankAccount += 25))
  _    <- ZIO.collectAllPar_(List(zio1, zio2, zio3))
} yield bankAccount
// res5: ZIO[Any, Nothing, Int] = zio.ZIO$FlatMap@67e01c43
```

Here we are using a mutable `var` for illustrative purposes which is *not* safe for concurrent access. However, this is still safe because access to the variable is guarded by the lock, so only one fiber can write to the variable at a time.

If other fibers wanted to read from the variable they could do that too and would not have to block for one of the writers to finish, just returning immediately with the current value of the variable.

We might ask, if this is the case why do we bother using `STM` for this data structure at all, except perhaps as an implementation detail? The answer is that returning `STM` effects allows us to compose locks together or compose them with other STM data structures.

Say we have two complex pieces of mutable state and we need to coordinate some state change between them. For example, perhaps these are two supervisors in an actor system and we need to describe transferring an actor from being the responsibility of one supervisor to the other.

```
trait Actor

trait Supervisor {
  def lock: TReentrantLock
  def supervise(actor: Actor): UIO[Unit]
  def unsupervise(actor: Actor): UIO[Unit]
}
```

To safely transfer an actor from one supervisor to another we need to lock on both supervisors. We can easily describe a lock on one supervisor using `TReentrantLock` as described above, but how do we describe locking on two different supervisors in a way that is safe and not subject to race conditions or deadlocks?

With STM, this is very easy.

```
def transfer(from: Supervisor, to: Supervisor, actor: Actor): UIO[Unit] = {
  ZIO.bracket {
    (from.lock.acquireWrite *> to.lock.acquireWrite).commit
  } { _ =>
    (from.lock.releaseWrite *> to.lock.releaseWrite).commit
  } { _ =>
    from.unsupervise(actor) *> to.supervise(actor)
  }
}
```

Now we have created a `transfer` method that lets us lock on two objects with mutable state in an extremely straightforward way that is not subject to deadlocks or race conditions.

By working directly with the `STM` operations we were able to compose larger transactions that described acquiring and releasing both locks. Then, we could run these transactions as ZIO effects to once again provide a simple interface for our users.

This pattern of both exposing STM functionality directly for power users who want to be able to compose transactions as well as ZIO functionality for users who want to directly work with a data type can be a good way to "provide the best of both worlds".

20.1.7 TSemaphore

A `TSemaphore` is a version of the `Semaphore` data type in ZIO that can be used in a transactional context. Like `TPromise` discussed above, the implementation of `TSemaphore` is extremely simple.

Because we do not fork fibers or perform concurrent effects *within* an STM transaction, we don't need a `TSemaphore` inside a single STM transaction.

However, we can use a `TSemaphore` when we have multiple effects being executed on different fibers and want to use a semaphore that we can compose with other transactional effects. For example, we might actually need to acquire permits from two separate semaphore, for example to read from one resource and write to another resource, and `TSemaphore` would let us compose acquiring those two permits into a single transaction.

```
final class TSemaphore private (private val permits: TRef[Long])
```

A `TSemaphore` is just a `TRef` that wraps a `Long` value indicating the number of permits available.

With this representation, we can implement the `acquire` and `release` methods to acquire a permit and release a permit.

```
final class TSemaphore private (private val permits: TRef[Long]) {
  val acquire: STM[Nothing, Unit] =
    permits.get.flatMap(n => if (n == 0) STM.retry else permits.set(n - 1))
  val release: STM[Nothing, Unit] =
    permits.update(_ + 1)
}
```

To acquire a permit, we simply check the current number of permits. If it is positive we decrement the number of permits by one and return immediately, otherwise we retry.

To release a permit, all we have to do is increment the number of permits.

In addition to the fact that we don't have to worry about conflicting updates, the implementation is considerably simplified here relative to a normal `Semaphore` because we don't have to worry about being interrupted within the context of an STM transaction.

Of course, this puts a bit more burden on the user of the `TSemaphore` to use the `acquire` and `release` operators with tools like `bracket` and `ZManaged` to ensure that permits are always released after an effect has completed. But this is an advanced use case, otherwise users can always fall back to working with a normal `Semaphore`.

20.1.8 TSet

The `TSet` rounds out our tour of STM data structures. A `TSet` is just the software transactional memory version of a mutable set.

It is much like a normal mutable set in that it can be thought of as a mutable map that doesn't have any values associated with the keys, and in fact that is the way `TSet` is represented internally.

```
final class TSet[A] private(private val map: TMap[A, Unit])
```

Use a `TSet` any time you would normally use a set, for example to maintain a collection regardless of ordering or the number of times an element appears, but are in a transactional context.

A `TSet` supports the usual operations of sets, such as `diff`, `intersect`, and `union`, as well as a variety of operators to fold the elements of the set to a summary value.

You can add a new element to the set with `put` or remove an element with `delete`, and check whether an element exists within `contains`.

Construct a new `TSet` using the usual `make`, `fromIterable`, or `empty` constructors.

20.2 Creating Your Own STM Data Structures

We have already seen a wide variety of different STM data structures and in many cases looked at their implementations as well, so you should already have a good sense of how these data structures are implemented. In this section we will take that understanding to the next level by walking through the implementation of an STM data structure from scratch.

In the process, we will try to walk through the thinking of *why* we are doing something in addition to *what* we are doing so you are in a better position to translate these learnings to implementing your own STM data structures.

The data structure we will focus on is the `TPriorityQueue`, which we described above and was one of the more recently added STM data structures in ZIO.

Recall that a `TPriorityQueue` is a queue where values are taken based on some `Ordering` defined on the element type instead of first in, first out, order.

The `TPriorityQueue` is a good example because it demonstrates the power of STM to solve a problem that would otherwise be quite difficult.

If we want a queue implementation that supports `take` semantically blocking until an element is in the queue we would normally use the `Queue` data type from ZIO. But all of the varieties of `Queue` currently implemented in ZIO are based on a first in, first out ordering rather than a priority ordering.

Furthermore, the internal implementation of `Queue` is highly optimized, using a specialized `RingBuffer` internally for maximum possible performance. This makes implementing a new variety of `Queue` a substantial undertaking that we would like to avoid if possible.

What are we supposed to do then if we need a queue that supports priority ordering, for example to maintain a queue of events where we can always take the earliest event? We will see that implementing this using STM is quite straightforward.

The first step in creating any new data structure is defining the interface we want this data structure to provide. Before we can start implementing it we need to say what we want this data structure to do.

In the case of `TPriorityQueue` we want to support two basic operations:

```
trait TPriorityQueue[A] {

  /**
   * Offers a value to the queue.
   */
  def offer(a: A): STM[Nothing, Unit]

  /**
   * Takes the first value in the queue based on an `Ordering` defined on `A`.
   */
  def take: STM[Nothing, A]
}
```

We will ultimately want to support some additional convenience methods typical of queues such as `poll` but if we can implement `offer` and `take` we should be able to implement those operators as well.

Now that we have defined the basic interface of the data structure, the next step is to determine a representation that will support those operations. For STM data structures, we know that the underlying representation will always be one or more `TRef` values.

In some cases it may be possible to use multiple `TRef` values to represent different parts of the data structure that can be modified independently as separate `TRef`

values for efficiency. But to get started it is helpful to just use a single `TRef` and then we can always optimize later.

So we know a `TPriorityQueue` is going to be represented as a `TRef` containing some underlying data structure.

```
final case class TPriorityQueue[A](tRef: TRef[_])
```

What data structure should go in the `TRef`?

A good rule of thumb is that if you are implementing an STM version of some data structure, wrapping the immutable version of the same data structure in a `TRef` is a good place to start. For example, we saw above that the underlying representation of `TQueue` was a `TRef[scala.collection.immutable.Queue]`.

So we might be tempted to think we could just implement our `TPriorityQueue` in terms of the `TRef` and `PriorityQueue` from the Scala standard library. Unfortunately, the Scala standard library only contains an implementation of a *mutable* priority queue.

Every value in a `TRef`, like every value in a `Ref`, should be an *immutable* value. While a `TRef` or a `Ref` represents a mutable value, the value inside must be immutable or otherwise safe for concurrent access because multiple fibers may be attempting to modify that value at the same time.

So unfortunately we will have to do a little more work here to find the right immutable data structure to support the functionality we want.

The next step is to ask what existing immutable data structures provide functionality that is most similar to what we need.

We know that the underlying data structure is going to have to support adding arbitrary values and maintaining those values based on some ordering, so a `SortedMap` or a `SortedSet` should come to mind.

We also know that our queue will also have to support multiple entries with the same value, for example we might call `offer` twice with the value `1` and we would expect be able to call `take` twice, getting back `1` each time. So it seems like `SortedSet` is not going to maintain enough information to support the functionality we need because it would allow us to represent that the queue contains the value `1` but not that it contains two entries with the value one.

Therefore, we get to the following for our representation of `TPriorityQueue`:

```
import scala.collection.immutable.SortedMap

final case class TPriorityQueue[A](ref: TRef[SortedMap[_, _]])
```

What should the key and value types of the `SortedMap` be?

The `SortedMap` maintains entries based on an `Ordering` defined on the keys, and we want the queue entries to be ordered based on an `Ordering` defined on the values in the queue, so it seems like the type of the keys should be `A`.

```
final case class TPriorityQueue[A](ref: TRef[SortedMap[A, _]])
```

What about the value type?

It is tempting to think we could get away with the value type being a an `Int` or a `Long` that represents how many times each value appears in the queue. However, this doesn't quite work.

To see why, consider what happens when the caller defines an `Ordering` that is not total.

```
final case class Event(time: Long, action: UIO[Unit])

implicit val eventOrdering: Ordering[Event] =
  Ordering.by(_.time)
```

This `Ordering` defines two events to have the same priority if they occur at the same time. But two events occurring at the same time may not be equal if they describe different actions.

If we use the representation where the value type in the `SortedMap` is `Int` or `Long`, two events that occur at the same time will have the same key and we will lose one of the two actions.

Of course, we could say that it is user error to define `eventOrdering` this way because normally we would like an `Ordering` to be a total ordering and to be consistent with the definition of equality for the type.

But this seems like a relatively common thing we can see users doing that could create bugs that are hard to diagnose, so it would be nice to avoid it if possible. In addition, in cases like this since `action` is an effect there is not really a way to define an ordering for actions.

To fix this, we can instead keep a collection of all the `A` values as the values in the `SortedMap`.

```
final case class TPriorityQueue[A](ref: TRef[SortedMap[A, List[A]]])
```

Now if two events are equal in the `Ordering` because they occur at the same time we simply keep copies of both of them in the map so that we can eventually return them both when values are taken from the queue. We will say that the order in which values with same priority in the `Ordering` are taken from the queue is not guaranteed and is up to the specific implementation.

We can refine this representation slightly by making one further observation: if an `A` value is in the `SortedMap` at all, the `List` contains at least one value. Otherwise that `A` value wouldn't appear in the `SortedMap` at all.

This allows us to refine our representation to:

```
final case class TPriorityQueue[A](ref: TRef[SortedMap[A, ::[A]]])
```

The use of the `::` type here may be unfamiliar. This is the `::` subtype of `List`, so it represents a `List` that contains at least one element.

Normally we do not expose the subtypes of algebraic data types like `List` separately in type signatures. However, the `::` type can be helpful in representing collections that can't be empty without having to introduce additional data types and with excellent compatibility with the Scala standard library.

We will see as we implement operators on `TPriorityQueue` how having this additional information lets the compiler prove for us that some of the things we are doing are correct.

So that's it! We now have our underlying representation for `TPriorityQueue`!

It may seem as if we spent a lot of time here on picking the right representation without really having worked with any STM operators yet. But as we will see once we have picked the right representation implementing the operators using ZIO's STM functionality is actually very easy.

Let's start by implementing the `offer` method.

```
final case class TPriorityQueue[A](ref: TRef[SortedMap[A, ::[A]]]) {
  def offer(a: A): STM[Nothing, Unit] =
    ref.update { map =>
      map.get(a) match {
        case Some(as) => map + (a -> ::(a, as))
        case None     => map + (a -> ::(a, Nil))
      }
    }
}
```

The `offer` method just needs to add the new value to the underlying `SortedMap`. So we call `update` to get access to the `SortedMap` and then insert the value.

To do the insertion, we check if the key already exists in the map. If it does we add the new value to the existing values associated with that key and otherwise we add a new key with the value.

The `take` requires us to use more STM functionality because we want to retry if the queue is empty.

```
final case class TPriorityQueue[A](ref: TRef[SortedMap[A, ::[A]]]) {
  def take(a: A): STM[Nothing, A] =
    ref.get.flatMap { map =>
      map.headOption match {
        case Some((a, as)) =>
          as.tail match {
            case h :: t => ref.set(map + (a -> ::(h, t))).as(as.head)
            case Nil    => ref.set(map - a).as(as.head)
          }
        case None => STM.retry
```

```
      }
    }
}
```

First we call `get` to access the `SortedMap` inside the `TRef`. Then we call `headOption` to either get the first key and value if they exist.

If there is no first key and value, that is the queue is empty, then we just retry.

If there is a key and value, we need to match on the value and determine if it contains only a single `A` value or multiple `A` values.

If it contains a single value, we remove the binding from the map entirely and return that value. Otherwise we update the binding to remove the value we just took, leaving the other values.

Either way we return the first value.

The use of `STM` functionality here is quite simple and just consists of using `get` to obtain the current state and then based on that either using `set` to update the state or `retry` to retry. The most complicated part of the logic was actually updating the underlying `SortedMap` implementation.

This is typical of STM data structures. The STM functionality itself is typically quite straightforward once we get the underlying representation right.

One thing to note here is how the use of `::` instead of `List` helped us.

We called both `head` and `tail` in the implementation above, which would normally not be safe because the list could be empty, and we would have to trust our own logic that these operators would never be called with an empty list. But by using the `::` we get the compiler to check that for us.

20.3 Conclusion

This chapter has described all the key STM data structures. In the process, you should have hopefully gained a better understanding of how you can implement your own STM data structures.

In addition, we saw several examples with data types like `TReentrantLock` of how STM data structures can be helpful even when we want to expose an interface that does not require users to work with STM transactions directly in most cases.

The next chapter will dive deeper into some of the more advanced topics including how STM is implemented in ZIO. If you want to go deeper in your learning about STM read on or otherwise feel free to skip ahead to the next section.

20.4 Exercises

Chapter 21

Software Transactional Memory: Advanced STM

Chapter 22

Advanced Error Management: Retries

This chapter begins our discussion of advanced error management in ZIO.

At this point you should understand ZIO's error type, including how to describe and recover from failures and the distinction between errors and defects.

In this chapter we will show how ZIO supports *composable retry strategies*. In the process, we will learn that the same strategies can also be used to describe repetition for effects that do not fail.

22.1 Limitations Of Traditional Retry Operators

It is very common for us to work with effects that can fail for one reason or another. For example, getting data from a web service could fail for a variety of reasons including:

- we do not have a network connection
- the service is unavailable
- we are being rate limited
- we do not have permission

In these cases we often want to *retry* the effect if it fails. Our *retry strategy* will typically include:

1. How many times and with what delay do we want to retry?
2. How should that change based on the type of failure that occurred?

We typically want some delay between retries because we want to allow time for the circumstances that caused the failure to potentially change before retrying.

If we do not have a network connection we are unlikely to have a connection if we retry immediately, but we may if we retry in one second, one minute, or one hour.

We also often want our retry strategy to depend on the error that occurred because different errors may require different retry strategies. If we are being rate limited then we need to wait to retry until our rate limit is reset, whereas if we don't have permission there is probably no point in retrying at all until we manually resolve the issue.

We can try to implement retry strategies using existing error handling operators and the ability to use recursion within the context of `ZIO`. For example, here is how we might implement an operator that retries an effect a specified number of times with a constant delay between each retry:

```
import zio._
import zio.clock._
import zio.duration._

def retryWithDelayN[R, E, A](n: Int)(zio: ZIO[R, E, A])(duration: Duration): ZIO[R with Cloc
  if (n <= 0) zio
  else zio.catchAll(_ => retryWithDelayN(n - 1)(zio)(duration).delay(duration))
```

If the number of remaining retries `n` is less than or equal to zero then we simply perform the effect. Otherwise we perform the effect and then if it fails call `retryWithDelayN` again with the specified delay and one fewer remaining retries.

At one level this code is relatively attractive. It uses the `catchAll` error handling operator and recursion to quite succinctly implement our retry strategy with strong guarantees that it will not block any threads and can be safely interrupted.

Compared to working with `Future` this is already a giant step forward because we can retry a `ZIO` effect, which is a description of a concurrent program, whereas we cannot retry a `Future` that is already "in flight". In addition, `Future` is not interruptible and does not have a built in way to schedule actions to be run after a specified duration.

However, this solution is not fully satisfactory. The problem is that it is not *composable* so we have to implement new retry strategies from scratch each time instead of being able to implement more complex retry strategies in terms of simpler ones.

To see this, consider what would happen if we wanted to retry a specified number of times with a constant delay between each retry, but only as long as the error satisfied a predicate.

Conceptually it seems like we should be able to implement this in terms of our existing implementation of retrying a specified number of times with a constant delay between each retry. After all, they are exactly the same except for the

additional condition, so we should be able to somehow "combine" the existing implementation with the new condition to implement this retry strategy.

Unfortunately, this is not possible.

The `retryWithDelayN` operator is just a method, so we have no way of "reaching inside" its implementation and only retrying certain errors. Furthermore, once we have applied the operator to a `ZIO` effect we get back another `ZIO` effect so we have no way to then modify the retry strategy that has already been applied.

Of course, we could just implement a new variant that also takes a predicate to determine whether to retry. But this is just restating the problem because then every time we want to implement a new retry strategy we need to implement it from scratch.

And we can imagine a wide variety of retry strategies. For example, perhaps the retries should proceed at a constant interval for a specified number of retries and then with an exponentially increasing delay after that up to some maximum, but only for certain types of errors.

What we need is a way of defining retry strategies where each retry strategy is a "building block" and we can glue simpler retry strategies together to make much more complex ones, just like we can build very complex ZIO programs from a relatively small number of simple operators.

With this perspective, we can see that even our original retry strategy implemented with `retryWithDelayN` actually mixes two concerns, how many times to retry and how long to delay between retries. Ideally we would like to be able to define this as the composition of two retry strategies that each deal exclusively with one of these concerns.

How do we go about actually defining retry strategies such that we can compose them in this way?

22.2 Retrying And Repeating With ZIO

The solution to this problem is ZIO's `Schedule` data type. A `Schedule` is a *description* of a *strategy* for retrying or repeating something.

```
trait Schedule[-Env, -In, +Out]
```

A `Schedule` requires an environment `Env`, consumes values of type `In` to decide whether to continue with some delay or be done, and produces a summary value of type `Out`.

The `Env` type parameter allows a schedule to potentially use some services in the environment to decide whether to continue with some delay or be done.

For example, a schedule that delays for a random duration within a specified interval would have an `Env` type of `Random` to reflect the schedule's dependency

on a source of randomness. If a schedule does not depend on any other services, as is often the case, the `Env` type will be `Any`.

The `In` type parameter represents the input that the schedule considers each time it decides whether to continue with some delay or be done. In the case of retrying, the `In` type would represent the types of errors that the retry strategy handles.

For example, a schedule that only continued for certain subtypes of `Throwable` and was done immediately for other subtypes of `Throwable` would have an `In` type of `Throwable`. If a schedule does not depend on the specific inputs, for example a schedule that always recurs a specified number of times, the `In` type would be `Any`.

The `Out` type parameter represents some summary value that describes the state of the schedule. The `Out` type parameter is not used directly when retrying with schedules, but is useful when composing schedules.

The exact type of the `Out` parameter will vary depending on the specific schedule but for example a schedule that recurs a specified number of times might output the number of recurrences so far. Some schedules may simply return their inputs unchanged, in which case the type of `In` would be the same as the type of `Out`.

With `Schedule` it is extremely simple to solve our original problem of implementing a retry strategy that retries a specified number of times with a constant delay between each retry.

```
import zio._
import zio.duration._

def delayN(n: Int)(delay: Duration): Schedule[Any, Any, (Long, Long)] =
   Schedule.recurs(n) && Schedule.spaced(delay)
```

Just like we wanted, we are able to build this schedule from two simpler schedules that separately describe the concepts of how many times we want to recur and how long we want to delay between recurrences.

The `recurs` schedule is exclusively concerned with the number of times to recur. It continues the specified number of times with no delay between each step.

The `spaced` schedule is exclusively concerned with the delay between recurrences. It continues forever with the specified delay between each step.

The `&&` operator combines two schedules to produce a new schedule that continues only as long as both schedules want to continue, using the maximum of the delays at each step.

So the combined schedule continues five times, because after that the `recurs` schedule does not want to continue and the composed schedule only continues as long as both schedules want to continue. And at each step it delays for the

specified duration `delay`, since `recurs` has zero delay and the composed schedule always uses the maximum delay.

This gives us the behavior we want in an extremely composable way to the point where we will often not even define specialized constructors but instead create our schedules directly using existing constructors and operators.

```
Schedule.recurs(5) && Schedule.spaced(1.second)
// res1: Schedule[Any, Any, (Long, Long)] = zio.Schedule$$anon$1@612a64ec
```

We will see many more schedule constructors and operators in the rest of this chapter, but this should give you an idea of the power as well as the elegance of working with `Schedule`.

Since `Schedule` just describes a strategy for retrying or repeating something, we need a way to take this description and apply it to retry or repeat a particular effect. The most common ways to do this are the `retry` and `repeat` operators on `ZIO`.

```
trait ZIO[-R, +E, +A] {
  def repeat[R1 <: R, B](schedule: Schedule[R1, A, B]): ZIO[R1, E, B]
  def retry[R1 <: R, S](schedule: Schedule[R1, E, S]): ZIO[R1, E, A]
}
```

The `retry` operator takes a `Schedule` that is able to handle all of the errors that the effect can potentially fail with and returns a new effect that either succeeds with the original value or fails with the original error.

One thing to note here is that while the `Schedule` produces an output of type `S` this result is ultimately discarded because we return a final effect that either fails with an `E` or succeeds with an `A`. If you want to perform further logic with the output of the `Schedule` there are `retryOrElse` and `retryOrElseEither` variants that allow specifying a fallback function that will be invoked if the effect still fails after all retries and has access to both the final error and output from the schedule.

The `repeat` operator is similar to the `retry` operator but allows using a `Schedule` to describe how to repeat an effect that succeeds. For example, we might repeat a `Schedule` to describe generating a report every Monday morning at a specified time.

While retrying an effect requires a `Schedule` that is able to handle all of the errors that the effect can potentially fail with, repeating an effect requires a schedule that is able to handle all the values that the effect can potentially succeed with. This allows the `Schedule` to decide whether to repeat based on the value produced by the effect, for example repeating until a predicate is satisfied.

The repeated effect succeeds with the value output by the `Schedule`. Often the `Schedule` will have identical `In` and `Out` types so the repeated effect will have the same result type as the original effect.

22.3 Common Schedules

Now that we have a basic understanding of `Schedule`, let's focus on different constructors of schedules.

These constructors form the basic "building blocks" of schedules. In the next two sections we will learn about different ways to transform and combine these building blocks, so by the end you will be in a good position to build complex schedules to address your own business problems.

22.3.1 Schedules For Recurrences

One simple family of `Schedule` constructors just describe recurring a specified number of times.

The most useful constructor here is `recurs`, which creates a schedule that recurs the specified number of times with no delay between recurrences.

```
object Schedule {
  def recurs(n: Long): Schedule[Any, Any, Long] =
    ???
}
```

Typically we will also want some delay in the final schedules we use to repeat or retry effects but by combining `recurs` with another constructor describing delaying we can create schedules that recur a finite number of times with a delay between each recurrence quite easily. There is also a `once` constructors that creates a schedule that continues a single time with no delay and a `stop` constructor that creates a schedule that is done immediately.

22.3.2 Schedules For Delays

The second family of `Schedule` constructors we will discuss describe patterns for delaying between recurrences.

Generally for modularity these schedules will describe a pattern for infinite recurrences with some delay between them. This allows us to compose them with other schedules that limit the number of recurrences as we saw above to precisely define the schedules we want.

The simplest of these constructors is the `spaced` constructor.

```
object Schedule {
  def spaced(duration: Duration): Schedule[Any, Any, Long] =
    ???
}
```

This schedule just delays the specified duration between each step and outputs the number of recurrences so far. It is the equivalent of calling the `delay` operator on `ZIO` on each repetition.

A slightly more complex variant of this is the `fixed` constructor.

```
object Schedule {
  def fixed(interval: Duration): Schedule[Any, Any, Long] =
    ???
}
```

The `fixed` constructor is similar to `spaced` in that it generates a schedule based on a constant delay and outputs the number of recurrences so far. The difference is how the time spent executing the effect being retried or repeated is factored into calculating the delay.

With the `spaced` schedule, the same delay will always be applied between each recurrence regardless of how long the effect being repeated or retried takes to execute.

For example, say we are repeating an effect that takes one second to execute with a schedule constructed using `spaced(10.seconds)`. In that case, the effect will run for one second, then there will be another ten second delay, then the effect will run for one second, then there will be another ten second delay, and so on.

This means there is always a constant delay between recurrences, but the actual frequency of recurrences may vary. For example, in the case above there is a ten second delay between each recurrence but since the effect takes one second to execute the effect is actually only executed once every eleven seconds.

In contrast, with the `fixed` schedule the time between the effect starting execution is fixed which means the actual delay can vary. If the effect takes one second to execute and we are retrying it with a schedule constructed using `fixed(10.seconds)` the effect will run for one second, then we will delay for nine seconds, then the effect will run for another second, then we will delay for nine seconds, and so on.

This means that effects being repeated or retried always recur at a fixed interval, but the actual delay between effects can vary based on how long the effects take to execute. In the case above there would now only be a nine second delay between each recurrence but the effect would be executed once every ten seconds.

The `fixed` schedule also has logic built in to prevent scheduled effects from "piling up". So if the effect being retried took twenty seconds to execute the next repetition would begin immediately but the effect would only be executed once on that recurrence instead of twice.

In general, if you want a constant *delay* between recurrences use `spaced`. If you want a constant *frequency* of recurrences use `fixed`.

One further variant of schedule constructors that use a constant delay is the `windowed` constructor.

```
object Schedule {
  def windowed(interval: Duration): Schedule[Any, Any, Long] =
    ???
}
```

This creates a schedule that delays until the nearest "window boundary" and returns the number of recurrences so far.

For example, if a schedule is constructed with the `windowed(10.seconds)` constructor and an effect takes one second to execute, the schedule will delay for nine seconds so the effect is evaluated again at the start of the next 10 second "window".

In this respect the `windowed` schedule is similar to the `fixed` schedule we just saw. Where the `windowed` schedule differs is with its behavior when the effect takes longer to execute than the window interval.

With the `fixed` schedule, we saw that if the effect takes longer than the fixed interval to execute then the next recurrence would immediately occur with no delay. For example, if the fixed interval is ten seconds and the effect takes eleven seconds to execute, the schedule would immediately recur and begin evaluating the effect again after eleven seconds.

In contrast, the `windowed` schedule will always wait until the beginning of the next window. So if the window was ten seconds long and the effect took eleven seconds to execute, the schedule would delay by nine seconds to begin evaluating the effect again at the beginning of the next ten second window.

The `windowed` operator is useful when you want to perform an effect at a specified interval and if the effect takes longer than the interval just wait until the next interval.

Moving on from here, we have various constructors that create schedules where the delays are not constant but vary in some way. Each of these constructors returns a schedule that outputs the most recent delay.

The first of these is the `linear` constructor.

```
object Schedule {
  def linear(base: Duration): Schedule[Any, Any, Duration] =
    ???
}
```

This constructs a schedule that delays between each recurrence like `spaced` but now the duration of the delays increase linearly starting with the base value. For example, `linear(1.second)` would construct a schedule with a one second delay, then a two second delay, a three second delay, and so on.

The `linear` schedule increases the delay between repetitions as the schedule continues so it can be useful when you initially want to retry an effect frequently but then want to retry less frequently if it still fails. For example, if a request

to a web service fails because there is no connection we might want to initially retry quickly in case it is a momentary issue but then retry more slowly if the connection is still unavailable.

Another variant on this theme is the `expontential` constructor.

```
object Schedule {
  def exponential(base: Duration, factor: Double = 2.0): Schedule[Any, Any, Duration] =
    ???
}
```

This creates a schedule that increases the delay between repetitions as the schedule continues like `linear`, but now increases the delay exponentially rather than linearly. So for example a schedule constructed with `exponential(1.second)` would initially delay for 1 second, then 2 seconds, then 4 seconds, and so on.

The `exponential` schedule models an exponential backoff strategy and increases the delay between subsequent recurrences much faster than the `linear` strategy. In the face of uncertainty about how long a condition will persist an exponential backoff strategy can be an excellent strategy because it increases the time between retries as our best estimate of how long the condition will persist increases.

If you want to change the exponential growth factor you can do that by specifying your own value for the `factor` parameter. For example, a schedule constructed using `exponential(1.second, 3.0)` would initially delay for one second, then three seconds, then nine seconds, and so on.

The `fibonacci` constructor describes another strategy for increasing the delay between subsequent recurrences, this time based on the Fibonacci series where each delay is the sum of the two preceding delays.

```
object Schedule {
  def fibonacci(one: Duration): Schedule[Any, Any, Duration] =
    ???
}
```

With this schedule, the first two delays will be for the specified duration and then each subsequent delay will be the sum of the preceding two delays. So for example, a schedule constructed using `fibonacci(1.second)` would delay for one second, one second, two seconds, three seconds, five seconds, eight seconds, and so on.

The remaining constructors for schedules with a delay allow creating finite schedules from a custom series, rather than infinite schedules from an existing mathematical series.

The simplest of these is the `fromDuration` constructor, which creates a schedule that recurs a single time with the specified delay.

```
object Schedule {
  def fromDuration(duration: Duration): Schedule[Any, Any, Duration] =
```

```
    ???
}
```

This is a very simple schedule so we will rarely want to use this schedule as our entire solution to a problem but it can be a helpful tool for modifying existing schedules to fit our business requirements.

The slightly more complex variant of this is the `fromDurations` constructor

```
object Schedule {
  def fromDurations(duration: Duration, durations: Duration*): Schedule[Any, Any, Duration]
    ???
}
```

This allows creating a schedule that recurs once for each of the specified durations, each time with the corresponding delay. The schedule outputs the duration of the last recurrence. This constructor can be useful when you have a specific series of delays that you want to use that does not fit neatly into one of the existing mathematical series that ZIO has support for out of the box.

22.3.3 Schedules For Conditions

The third family of schedule constructors we will talk about allow expressing concepts of conditionality.

These schedules examine the input to the schedule and decide to continue or not based on whether the input satisfies a predicate. There are a large number of variants of these constructors but conceptually they are all quite similar.

```
object Schedule {
  def recurWhile[A](f: A => Boolean): Schedule[Any, A, A] =
    ???
  def recurWhileM[Env, A](f: A => URIO[Env, Boolean]): Schedule[Env, A, A] =
    ???
  def recurWhileEquals[A](a: => A): Schedule[Any, A, A] =
    ???
  def recurUntil[A](f: A => Boolean): Schedule[Any, A, A] =
    ???
  def recurUntilM[Env, A](f: A => URIO[Env, Boolean]): Schedule[Env, A, A] =
    ???
  def recurUntilEquals[A](a: => A): Schedule[Any, A, A] =
    ???
}
```

The `recurWhile` constructor is representative of this family of schedule constructors. It constructs a schedule that recurs forever with no delay as long as the specified predicate is true, but is done immediately as soon as the predicate is false.

There is also a `recurWhileM` variant that allows performing an effect as part of determining whether the schedule should continue, as well as a simplified `recurWhileEquals` constructor that continues as long as the input is equal to the specified value.

Each of these constructors also has a corresponding `recurUntil` variant that continues forever as long as the specified predicate is false and is done immediately once the specified condition evaluates to true.

These constructors allow us to begin to express the idea that we want whether or not a schedule continues to depend on the input type. For example, we could use these constructors to express the idea that we should only retry network connection errors, or that we should repeat trying to find a solution until the solution satisfies certain parameters.

22.3.4 Schedules For Outputs

The next family of schedule constructors create schedules that output values that are useful in some way, either by converting existing inputs into outputs or by generating new outputs from the schedule itself such as how much time has passed since the first step in the schedule.

The most basic schedule constructor for converting inputs into outputs is `fromFunction`, which creates a schedule from a function.

```
object Schedule {
  def fromFunction[A, B](f: A => B): Schedule[Any, A, B] =
    ???
}
```

The schedule returned by `fromFunction` always recurs with no delay and simply transforms every `A` input it receives into a `B` output using the specified function `f`.

A variant of this is the `identity` constructor, which is like `fromFunction` but constructs a schedule that passes its inputs through unchanged instead of transforming them with a function.

```
object Schedule {
  def fromFunction[A, B](f: A => B): Schedule[Any, A, B] =
    ???
  def identity[A]: Schedule[Any, A, A] =
    fromFunction(a => a)
}
```

The `identity` constructor can be particularly useful when composed with other constructors to create more complex schedules that still return their original inputs unchanged. These schedules can then be used with the `retry` operator on `ZIO` to return effects with retry logic added that still succeed with the same type of value as the original effect.

One other variant of the `identity` constructor is `collectAll`.

```
object Schedule {
  def collectAll[A]: Schedule[Any, A, Chunk[A]] =
    ???
}
```

The `collectAll` constructor is like `identity` in terms of passing through the `A` values except this time instead of just outputting the most recent `A` value input each time the schedule maintains a data structure containing all the `A` values input so far and outputs all of those values each time.

For example, if we have a schedule constructed using `collectAll[Int]` and it receives inputs of `1`, `2`, and `3`, it will output `Chunk(1)`, `Chunk(1, 2)`, and `Chunk(1, 2, 3)`, each time outputting all the inputs received so far.

This can be useful for collecting all the intermediate outputs produced in evaluating a schedule. For example, if we are repeating an effect until it reaches a satisfactory solution to some problem, we could use `collectAll` to visualize all the intermediate solutions generated along the way to the final solution.

In addition to the basic `collectAll` constructor, there are `collectWhile` and `collectUntil` variants that continue as long as some predicate is satisfied. You can think of these as a composition of the `recurWhile` and `recurUntil` schedules we saw above with the `collectAll` schedule.

The constructors above are the primary ones for creating schedules that transform inputs into outputs in some way. In addition to these, there are a set of constructors that create schedules outputting certain values regardless of the inputs received.

One of the most general of these is the `unfold` constructor, which allows constructing a schedule by repeatedly applying a function to an initial value.

```
object Schedule {
  def unfold[A](a: => A)(f: A => A): Schedule[Any, Any, A] =
    ???
}
```

The schedule returned by `unfold` will initially output the specified `a` value and then, on each recurrence after that, will apply the function `f` to the last value to generate a new value.

For example, we can implement another constructor, `count`, that just outputs the number of recurrences so far, using `unfold`.

```
object Schedule {
  val count: Schedule[Any, Any, Long] =
    unfold(0L)(_ + 1L)
  def unfold[A](a: => A)(f: A => A): Schedule[Any, Any, A] =
```

```
    ???
}
```

Another even simpler constructor we can implement in terms of `unfold` is `succeed`, which just returns a schedule that always outputs the same constant value.

```
object Schedule {
  def succeed[A](a: => A): Schedule[Any, Any, A] =
    unfold(a)(a => a)
  def unfold[A](a: => A)(f: A => A): Schedule[Any, Any, A] =
    ???
}
```

One final variant that is particularly useful is the `elapsed` constructor, which returns a new schedule that always recurs with no delay but outputs the time since the first step.

```
object Schedule {
  val elapsed: Schedule[Any, Any, Duration] =
    ???
}
```

We can use this to see how much time has passed each time the schedule recurs or with other operators to stop once a certain amount of time has passed.

22.3.5 Schedules For Fixed Points In Time

The final important family of schedule constructors is those for schedules that recur at fixed points in time.

So far most of our discussion has focused on schedules that recur at some *relative* time intervals, for example with a one minute delay between recurrences.

But we can also use schedules to describe patterns of recurrence that occur at fixed *absolute* points in time, such as every Monday at 9 A.M. This allows us to use schedules to implement "cron job" like functionality in a very principled and composable way.

ZIO's support for this functionality begins with several basic constructors for describing recurrences at various absolute points in time. As we will see, each of these is quite specific and none of them individually is necessarily particularly useful, but they compose together in all the right ways to describe any possible series of recurrences at absolute points in time.

We will focus in this section on describing these constructors and we will see later how we can start to combine them to build schedules that solve much more complex "real world" problems.

The basic constructors for working with absolute points in time are:

```
object Schedule {
  def secondOfMinute(second: Int): Schedule[Any, Any, Long] =
    ???
  def minuteOfHour(minute: Int): Schedule[Any, Any, Long] =
    ???
  def hourOfDay(hour: Int): Schedule[Any, Any, Long] =
    ???
  def dayOfWeek(day: Int): Schedule[Any, Any, Long] =
    ???
}
```

Each of these constructors returns a schedule that always recurs and delays for each recurrence until the next absolute point in time satisfying the specified condition, returning the number of recurrences so far. For example, a schedule constructed with `secondOfMinute(42)` would recur at 12:00:42, 12:01:42, 12:02:42, and so on whereas a schedule constructed with `minuteOfHour(42)` would recur at 12:42:00, 1:42:00, 2:42:00, and so on.

Typically we don't just want to recur every hour or every day but at some combination of these, such as every Monday at 9 AM. We will see more about how this works later in this chapter, but just like we could combine schedules for relative delays we can also combine schedules for fixed points in time.

One of the most useful ways to do this is the `&&` operator, which describes the *intersection* of the recurrence intervals of two schedules. For example, here is how we could describe doing something every Monday at 9 AM.

```
val automatedReportSchedule =
  Schedule.dayOfWeek(1) && Schedule.hourOfDay(9)
// automatedReportSchedule: Schedule[Any, Any, (Long, Long)] = zio.Schedule$$anon$1@4b0c14c8
```

Again, we will see more about how to compose schedules like this later in the chapter but hopefully this gives you a sense of how these seemingly very simple constructors can be used to describe much more complex patterns for recurrence at fixed points in time.

22.4 Transforming Schedules

At this point we know all the basic constructors of schedules. The next step is to understand the operators on `Schedule` that allow building more complex schedules from simpler ones.

These operators fall into two main categories.

The first, which we will discuss in this section, are operators for transforming *one* schedule. These operators let us take an existing schedule and add logic to it in some way, for example modifying the delay produced by the original schedule or performing some effect like logging for each step in the schedule.

The second, which we will discuss in the next section, are operators for combining *two* schedules to create a new more complex schedule that somehow combines the logic of each of the original schedules. We will see in the next section that there are multiple ways to combine schedules that can be useful in different situations.

With that introduction, let's dive into the key operators for transforming schedules.

22.4.1 Transforming Inputs and Outputs

The first set of operators we will look at allow transforming the input or output of an existing schedule. We know that a `Schedule` accepts values of type `In` and produces values of type `Out` so it shouldn't be surprising that we can transform these input and output types with many of the same operators we have seen for other ZIO data types.

```
trait Schedule[-Env, -In, +Out] {
  def as[Out2](out2: => Out2): Schedule[Env, In, Out2]
  def dimap[In2, Out2](f: In2 => In, g: Out => Out2): Schedule[Env, In2, Out2]
  def map[Out2](f: Out => Out2): Schedule[Env, In, Out2]
  def contramap[Env1 <: Env, In2](f: In2 => In): Schedule[Env, In2, Out]
  def mapM[Env1 <: Env, Out2](f: Out => URIO[Env1, Out2]): Schedule[Env1, In, Out2]
}
```

The two most basic operators for transforming inputs and outputs are `map` and `contramap`.

The `map` operator allows us to transform the output type of a schedule using a specified function and conceptually just returns a new schedule that just calls the underlying schedule and then transforms each of its outputs with the function. This operator is useful when we need to transform the outputs of the schedule but don't want to change the underlying logic of the schedule with regard to whether to continue or how long to delay for.

The `contramap` operator is similar but just works for the input type, taking a function that transforms each input from the new input type we want to the original input type that the schedule knows how to handle. This can be useful if we want to adapt the inputs that the schedule can accept to match the effect we want to repeat or retry.

In addition to these basic operators there are a few other variants.

The `dimap` operator transforms both the inputs and outputs to the schedule using a pair of functions. You can think of it as just calling `map` with one function and `contramap` with the other so it can be more concise if you need to transform both the input and output types but there is nothing special going on here.

The `as` operator works the same way as the `as` operator on `ZIO` and is equivalent to mapping every output to a constant value. Finally, there is a `mapM` variant which allows performing effects while transforming the output of a schedule.

With these operators you should be able to transform the input and output types of any schedule to meet your needs while keeping the underlying logic of the schedule intact.

22.4.2 Summarizing outputs

The next set of operators focus on summarizing the outputs of a schedule. Whereas the operators above allowed transforming each output value individually, the operators in this section allow maintaining some state along the way so the new output value can depend on not just the old output value but also all previous output values.

```
trait Schedule[-Env, -In, +Out] {
  def collectAll: Schedule[Env, In, Chunk[Out]]
  def fold[Z](z: Z)(f: (Z, Out) => Z): Schedule[Env, In, Z]
  def foldM[Env1 <: Env, Z](z: Z)(f: (Z, Out) => URIO[Env1, Z]): Schedule[Env1, In, Z]
  def repetitions: Schedule[Env, In, Int]
}
```

The most general of these is the `foldM` operator, which allows *statefully* transforming the output values from the schedule. It works as follows:

1. For each input, provide the input to the original schedule to get an output value
2. If the schedule is done, return the initial summary value `z` immediately
3. Otherwise, use the function `f` to produce a new summary value and output that
4. Repeat the process above with the initial summary value replaced by the new one

The `foldM` operator allows performing an effect within each step of the fold, whereas the simpler `fold` operator just uses a normal function.

To get a sense for working with these operators we can use `fold` to implement the specialized `collectAll` and `repetitions` operators, which output all the values seen before or the number of repetitions so far, respectively.

```
trait Schedule[-Env, -In, +Out] {
  def fold[Z](z: Z)(f: (Z, Out) => Z): Schedule[Env, In, Z]
  def collectAll: Schedule[Env, In, Chunk[Out]] =
    fold[Chunk[Out]](Chunk.empty)(_ :+ _)
  def repetitions: Schedule[Env, In, Int] =
    fold(0)((n, _) => n + 1)
}
```

The implementation of `collectAll` just starts the fold with an empty `Chunk` and then adds each new output value to the current `Chunk`. Similarly, the implementation of `repetitions` just started with zero and adds one for each repetition.

Whenever you need to transform the output values of a schedule and you need the current output value to depend on the *previous* output values look to one of the fold operators.

22.4.3 Side Effects

These operators don't transform the input or output values of the `Schedule` but allow performing some effect based on those values, such as logging them.

```
trait Schedule[-Env, -In, +Out] {
  def tapInput[Env1 <: Env, In1 <: In](f: In1 => URIO[Env1, Any]): Schedule[Env1, In1, Out]
  def tapOutput[Env1 <: Env](f: Out => URIO[Env1, Any]): Schedule[Env1, In, Out]
  def onDecision[Env1 <: Env](f: Decision[Env, In, Out] => URIO[Env1, Any]): Schedule[Env1,
}
```

The more basic variants are `tapInput` and `tapOutput`, which allow performing some effect for each input value received by the schedule or output value produced. This can be useful when you don't want to transform the values but just want to do something else with them like printing them to the console for debugging, logging them, or updating a data structure like a `Ref` or a `Queue` based on them.

The slightly more complex variant is `onDecision`. This is somewhat similar to `tapOutput` in that it allows performing an effect for each output, except this time it has access to more information about the internal state of the schedule.

We will see more about the `Decision` data type at the end of the chapter when we discuss the internal implementation of `Schedule` but you can think of `Decision` as representing the schedule's decision of whether to continue and how long to delay for.

The `onDecision` operator can be useful when you want the effect you perform to depend not just on the value output but the decision of the schedule itself. For example, you could use `onDecision` to log whether the schedule decided to continue and how long it delayed for on each recurrence as part of debugging.

22.4.4 Environment

As we have seen, `Schedule` has an environment type, so there are also some of the traditional operators for working with the environment type including `provide` and `provideSome`.

```
trait Schedule[-Env, -In, +Out] {
  def provide(env: Env): Schedule[Any, In, Out]
  def provideSome[Env2](f: Env2 => Env): Schedule[Env2, In, Out]
}
```

Generally we will not need these operators in most cases.

First, most schedules do not have any environmental requirements. Remember that a `Schedule` is just a *description* of a strategy for retrying or repeating

so a schedule itself will typically not depend on a `Clock` even though actually *running* a schedule usually will.

Second, most of the time when a schedule does require some environment we just want to propagate that environmental dependency to the effect being retried or repeated. For example, if we have a schedule that applies a random delay and use it to retry an effect, the new effect would have a dependency on `Random` which would accurately represent its requirements and we could later provide at a higher level of our application.

However, in some cases it can be useful to provide a schedule with some or all of its environment. In particular, if we are trying to "hide" the use of a schedule or our particular implementation we might want to provide the dependency that the schedule requires to avoid exposing it to our users.

For example, we might retry part of some logic with a random delay internally as an implementation detail, but not want to expose that to our users. In that case we could `provide` the schedule with a live implementation of that `Random` service to eliminate the dependency.

Again, in most cases that should not be necessary but if you do want to provide some or all of the services that a schedule requires you can use the `provide` and `provideSome` operators to do that.

22.4.5 Modifying Schedule Delays

The next set of operators allow modifying the delays between each recurrence of the schedule without modifying the schedule's decision of whether to continue or not.

```
import zio.duration._
import zio.random._

trait Schedule[-Env, -In, +Out] {
  def addDelay(f: Out => Duration): Schedule[Env, In, Out]
  def addDelayM[Env1 <: Env](f: Out => URIO[Env1, Duration]): Schedule[Env1, In, Out]
  def delayed(f: Duration => Duration): Schedule[Env, In, Out]
  def delayedM[Env1 <: Env](f: Duration => URIO[Env1, Duration]): Schedule[Env1, In, Out]
  def jittered(min: Double = 0.0, max: Double = 1.0): Schedule[Env with Random, In, Out]
  def modifyDelay(f: (Out, Duration) => Duration): Schedule[Env, In, Out]
  def modifyDelayM[Env1 <: Env](f: (Out, Duration) => URIO[Env1, Duration]): Schedule[Env1,
}
```

As you can see, there are three primary variants for modifying schedule delays, `addDelay`, `delayed`, and `modifyDelay`, as well as one more specialized variant `jittered`. Each of these also has a variant with an `M` suffix that allows performing an effect as part of the modification function.

The most general operator is `modifyDelay`, which allows returning a new delay

for each recurrence of the schedule based on the original delay and the output value of the schedule. For example, we could create a version of a schedule that delays for twice as long as the original schedule.

```
def doubleDelay[Env, In, Out](schedule: Schedule[Env, In, Out]): Schedule[Env, In, Out] =
  schedule.modifyDelay((_, duration) => duration * 2)
```

In this case we just produced a new delay based on the original delay but we could have also used to the output from the schedule to determine the new delay to apply.

The `addDelay` operator is similar to `modifyDelay` except it only gives access to the output value of the schedule and *adds* the returned duration to the original delay of the schedule.

```
trait Schedule[-Env, -In, +Out] {
  def modifyDelay(f: (Out, Duration) => Duration): Schedule[Env, In, Out]
  def addDelay(f: Out => Duration): Schedule[Env, In, Out] =
    modifyDelay((out, duration) => f(out) + duration)
}
```

The `addDelay` operator can be particularly useful to transform a schedule that outputs values without doing delays into a schedule that actually delays between each recurrence.

For example, let's consider how we could implement the Fibonacci schedule, which delays between each recurrence for a duration specified by the Fibonacci series, where each delay is the sum of the previous two delays.

We can divide this problem into two steps.

First, we need to create a schedule that outputs values based on the Fibonacci series. Then we need to transform that schedule to actually delay between each recurrence based on those values.

We can accomplish the first step using the `unfold` constructor we saw before.

```
import zio._
import zio.duration._

def fibonacci(one: Duration): Schedule[Any, Any, Duration] =
  Schedule.unfold((one, one)) { case (a1, a2) =>
    (a2, a1 + a2)
  }.map(_._1)
```

So far this schedule outputs durations based on the Fibonacci series but doesn't actually delay between each recurrence, because the `unfold` constructor always creates schedules that recur without delay.

To add a delay to this we can use the `addDelay` operator to add a delay to each step based on the output value of the schedule.

```
def fibonacci(one: Duration): Schedule[Any, Any, Duration] =
  Schedule.unfold((one, one)) { case (a1, a2) =>
    (a2, a1 + a2)
  }.map(_._1).addDelay(out => out)
```

The `delayed` operator is another variant on this theme. It is like `modifyDelay` but only gives access to the original delay for each recurrence of the schedule instead of the output plus the original delay.

The final more specialized variant to be aware of is `jittered`. The `jittered` operator modifies each delay of the schedule to be a random value between a `min` and `max` factor of the original delay.

For example, a schedule constructed using `spaced(10.seconds).jittered(0.5, 1.5)` would delay for each recurrence for a random duration uniformly distributed between five seconds and fifteen seconds. There is a default version of the `jittered` operator modifies each delay to between zero and the original delay, equivalent to `jittered(0.0, 1.0)`.

The `jittered` operator can be very useful along with other schedules to add some randomness to an existing schedule so that many effects do not interact with an external service at exactly the same time. It can also be useful in some situations to present against adversarial behavior by preventing others interacting with our code from being able to predict exactly when recurrences will occur.

22.4.6 Modifying Decisions

The operators in this section are similar to the ones in the last section but allow modifying *whether* the schedule continues.

```
trait Schedule[-Env, -In, +Out] {
  def check[In1 <: In](test: (In1, Out) => Boolean): Schedule[Env, In1, Out]
  def checkM[Env1 <: Env, In1 <: In](test: (In1, Out) => URIO[Env1, Boolean]): Schedule[Env1
  def reconsider[Out2](f: Decision[Env, In, Out] => Either[Out2, (Out2, Interval)]): Schedul
  def reconsiderM[Env1 <: Env, In1 <: In, Out2](
    f: Decision[Env, In, Out] => URIO[Env1, Either[Out2, (Out2, Interval)]]
  ): Schedule[Env1, In1, Out2]
}
```

There are two key variants, `check` and `reconsider`, each of which also have effectual variants with the `M` suffix.

The `check` operator passes each input and output from the schedule to the specified function and continues only if both the original schedule wants to continue and the function `test` evaluates to true. So `check` adds an additional condition to the schedule continuing.

In this way `check` can be thought of as the converse to the operators we saw in the last section. While those operators modify the delay of the schedule without

changing whether or not it continues, `check` modifies whether the schedule continues without changing the delay if it does.

The `reconsider` operator is the most powerful variant and allows changing both whether the schedule continues as well as how long it delays for. It has access to the full original `Decision` of the schedule and the provided function returns an `Either` where the `Left` case indicates that the schedule should terminate with the specified output and the `Right` case indicates that the schedule should continue with the specified output and delay.

The `reconsider` operator also allows transforming the output type of the schedule, so it is one of the most powerful variants of operators to transform a schedule, allowing transforming the output, decision of whether to continue, and decision of how long to delay for in arbitrary ways.

22.4.7 Schedule Completion

The final set of operators for transforming a single schedule add additional logic to be run when a schedule is to complete or to reset the schedule to its original state.

```
trait Schedule[-Env, -In, +Out] {
  def ensuring(finalizer: UIO[Any]): Schedule[Env, In, Out]
  def forever: Schedule[Env, In, Out]
  def resetAfter(duration: Duration): Schedule[Env, In, Out]
  def resetWhen(f: Out => Boolean): Schedule[Env, In, Out]
}
```

The first of these operators is `ensuring`, which adds some finalizer that will be run when the schedule decides not to continue. One thing to note about the `ensuring` operator here is that it does not provide the same guarantees around interruption as the `ensuring` operator on ZIO.

If the schedule is run to completion and decides not to continue then the finalizer will be run. However, if the schedule terminates before deciding not to continue early, for example due to interruption, the finalizer will not be run.

So if you need guarantees around interruption it is best to use the `ensuring` or `bracket` operators on `ZIO` itself instead of the `ensuring` operator on `Schedule`.

The next operator, `forever`, makes sure the schedule never finishes continuing by just repeating the original schedule each time it would otherwise be done.

Each time the schedule would otherwise be done its state is reset back to its original state. So for example if we had a schedule that delayed for one second, two seconds, and three seconds before being done and called `forever`, we would get back a new schedule that would repeat the same pattern forever.

The `resetAfter` and `resetWhen` operators are variants on this idea of resetting the state of the schedule.

Many time a schedule will maintain some internal state. For example, the schedule created by `linear` needs to maintain the number of recurrences so far so that it can increase the delay between each recurrence.

Sometimes we may want to continue the schedule but reset its state, for example going back to recurring with a shorter delay between recurrences in the case of the `linear` schedule. The `resetAfter` and `resetWhen` operators allow us to do that by resetting the schedule back to its original state either after the specified duration has elapsed or when the specified predicate evaluates to true.

22.5 Composing Schedules

At this point we have learned both how to construct schedules as well as how to transform individual schedules to add additional logic to them. In this section we will expand that knowledge to include combining two different schedules to produce a third schedule that in some way reflects the logic of both of the first two schedules.

The `Schedule` data type actually supports several different ways to compose schedules.

22.5.1 Intersection And Union Of Schedules

The first way of composing schedules is intersection or union. In this case we have two schedules that both accept the same type of inputs and we feed the same input to both schedules, deciding whether the overall schedule should continue and how long it should delay for based on some combination of the decisions from the two schedules.

This corresponds to geometric intersection or union if we think of schedules as sequences of intervals during which each schedule wants to recur.

The two most general operators for describing this are `intersectWith` and `unionWith`.

```
trait Schedule[-Env, -In, +Out] { self =>
  def intersectWith[Env1 <: Env, In1 <: In, Out2](
    that: Schedule[Env1, In1, Out2]
  )(f: (Interval, Interval) => Interval): Schedule[Env1, In1, (Out, Out2)]
  def unionWith[Env1 <: Env, In1 <: In, Out2](
    that: Schedule[Env1, In1, Out2]
  )(f: (Interval, Interval) => Interval): Schedule[Env1, In1, (Out, Out2)]
}
```

The `intersectWith` and `unionWith` operators take two schedules that can handle the same input type and create a new combined schedule that works as follows:

1. For each input value, feed the input to both the `self` and `that` schedules

2. Get a decision from each schedule regarding whether it wants to continue and how long it wants to delay for
3. Apply some *continue logic* to decide whether or not to continue based on whether each schedule wants to continue
4. Apply some *delay logic* to decide how long to delay for based on how long each schedule wants to delay for
5. Apply some *output logic* to combine the output values from the two schedules into a single output value
6. Produce an overall decision of whether to continue and how long to delay for based on the two steps above

Stated this way, these operators describe very general but also very powerful logic for combining two schedules by feeding inputs to both schedules.

Within this framework, the `intersectWith` and `unionWith` operators differ in the *continue logic* they apply.

The `intersectWith` operator uses the logic "continue only as long as both schedules wants to continue". In contrast, the `unionWith` operator uses the logic "continue as long as either schedule wants to continue".

The schedules both defer providing any *delay logic*, allowing the caller to do that by specifying a function `f` to combine the delays from the two original schedules to produce a new delay. And they both also defer specifying the *output logic* to some extent, just returning a tuple of the output types from the two original schedules that the caller can then choose how to combine using `map` or a similar operator.

These operators are then further specialized in `&&` and `||`, which are implemented in terms of `intersectWith` and `unionWith` and "fill in" the logic for how to combine the delays produced by the two original schedules that was left open in the more general operators.

```
import zio.duration._

trait Schedule[-Env, -In, +Out] { self =>
  def intersectWith[Env1 <: Env, In1 <: In, Out2](
    that: Schedule[Env1, In1, Out2]
  )(f: (Interval, Interval) => Interval): Schedule[Env1, In1, (Out, Out2)]
  def unionWith[Env1 <: Env, In1 <: In, Out2](
    that: Schedule[Env1, In1, Out2]
  )(f: (Interval, Interval) => Interval): Schedule[Env1, In1, (Out, Out2)]
  def &&[Env1 <: Env, In1 <: In, Out2](that: Schedule[Env1, In1, Out2]): Schedule[Env1, In1,
    (self intersectWith that)(_ max _)
  def ||[Env1 <: Env, In1 <: In, Out2](that: Schedule[Env1, In1, Out2]): Schedule[Env1, In1,
    (self unionWith that)(_ min _)
}
```

The `&&` operator continues only as long as both schedules want to continue and

always delays for the *maximum* of the duration that each schedule that wants to delay for. You can think of `&&` as representing the geometric intersection of the intervals the two schedules want to recur.

The `||` operator on the other hand continues as long as either schedule wants to continue and always delays for the *minimum* of the duration that each schedule wants to delay for. You can think of `||` as representing the geometric union of the intervals the two schedules want to recur for.

These operators, especially the `&&` operator and its variants, are some of the most frequently used when working with schedules, so it is helpful to get some experience with them.

To start with, let's go back to the example from the beginning of this chapter now that we have a better understanding of `Schedule` and schedule constructors. We would like to implement a schedule that recurs five times with a delay of one second between each recurrence.

We already have the `spaced` constructor which constructs a schedule that recurs forever with specified delay between each recurrence and the `recurs` constructor which constructs a schedule that recurs the specified number of times with no delay. How do we combine them to build the schedule to solve our problem?

If we use the `&&` operator then our composed schedule will recur only as long as both schedules want to continue. The `spaced` schedule will always continue but the `recurs` schedule will only recur five times, so the resulting schedule will only recur five times, which is what we want.

The schedule composed with the `&&` operator will also delay for the *maximum* of the length of the delays from each of the original schedules. The `spaced` schedule will always delay for one second while the `recurs` schedule will not delay at all, so the resulting schedule will delay for one second between recurrences, again exactly what we want.

So using `spaced(1.second) && recurs(5)` will give us the behavior we are looking for.

What if we had instead used the `||` operator?

In this case the schedule would continue as long as either schedule wanted to continue. Since the `spaced` schedule always wants to continue this would mean the composed schedule would continue forever.

The scheduled composed with the `||` operator will also delay for the *minimum* of the length of the delays from each of the original schedules.

The `recurs` schedule recurs five times with no delay, so the first five recurrences would have no delay. After that only the `spaced` schedule continues and it delays for one second each time.

So the resulting schedule would recur five times with no delay and then forever with a delay of one second between each recurrence after that.

In this case the schedule composed using `&&` was definitely what we wanted, but hopefully working through the implications of using both operators gives you a better sense for the different ways to compose schedules.

You can use the `intersectWith` and `combineWith` operators to implement operators that have different combinations of logic for deciding whether to continue and how long to delay for. For example, it is possible to combine two schedules to produce a new schedule that continues as long as both schedules want to continue but using the *minimum* delay.

However, we have found that the most useful operators are the ones defined in terms of `&&` and `||` since they mirror the concept of geometric union and intersection when schedules are conceptualized as sequences of intervals when recurrence may occur.

The `zipWith` operator is implemented in terms of `&&` and is the basis of several more specialized operators that combine the output values of the two schedules in various ways.

```
trait Schedule[-Env, -In, +Out] { self =>
  def &&[Env1 <: Env, In1 <: In, Out2](that: Schedule[Env1, In1, Out2]): Schedule[Env1, In1,
  def map[Out2](f: Out => Out2): Schedule[Env, In, Out2]
  def zipWith[Env1 <: Env, In1 <: In, Out2, Out3](
    that: Schedule[Env1, In1, Out2]
  )(f: (Out, Out2) => Out3): Schedule[Env1, In1, Out3] =
    (self && that).map(f.tupled)
}
```

As you can see, `zipWith` just combines two schedules using intersection and then applies a function `f` to convert the output values of the two original schedules into a new schedule. All of the other `zip` variants that you are used to on other ZIO data types, such as `zip`, `zipLeft`, `zipRight`, `<*>`, `<*` and `*>` are also defined on `Schedule`.

Using these, we can clean up our implementation of combining the `spaced` and `recurs` schedules slightly. Recall that in our introduction to working with schedules our implementation of the `delayN` schedule was:

```
def delayN(n: Int)(delay: Duration): Schedule[Any, Any, (Long, Long)] =
   Schedule.recurs(n) && Schedule.spaced(delay)
```

One thing that is slightly awkward about this is the output type of `(Long, Long)`. Both `recurs` and `spaced` return schedules that output the number of repetitions so far, so we get a composed schedule that returns the number of times that each schedule has recurred.

But the two schedules will have recurred the same number of times for each step so we are not really getting additional information here and are unnecessarily complicating the type signature of our returned schedule constructor. We can clean this up by using the `*>` operator to discard the output of the first schedule.

```
def delayN(n: Int)(delay: Duration): Schedule[Any, Any, Long] =
   Schedule.recurs(n) *> Schedule.spaced(delay)
```

Note that we could have used `<*` as well since `&&` is commutative and both schedules output the same values in this case.

The `&&` operator is one of the most useful operators in composing more complex schedules from simpler ones so think about how to use it when composing your own schedules for describing retries in your business domain.

22.5.2 Sequential Composition Of Schedules

The second way of composing schedules is sequential composition of schedules. In this case we have two schedules and we feed inputs to the first schedule for as long as it wants to continue, ignoring the other schedule, and then switch over to feeding inputs to the second schedule once the first schedule is done.

This corresponds to thinking of schedules as sequences of intervals during which each schedule wants to recur and concatenating the sequences end to end.

The most general operator to describe this type of composition is the `andThenEither` operator.

```
trait Schedule[-Env, -In, +Out] { self =>
  def andThenEither[Env1 <: Env, In1 <: In, Out2](
    that: Schedule[Env1, In1, Out2]
  ): Schedule[Env1, In1, Either[Out, Out2]]
}
```

The `andThenEither` operator conceptually works as follows:

1. For each input value, feed the input to the `self` schedule
2. As long as the `self` schedule wants to continue, continue with the delay from the `self` schedule
3. If the `self` schedule is ever done, then switch over and feed each input value to the `that` schedule
4. As long as the `that` schedule wants to continue, continue with the delay from the `that` schedule
5. When the `that` schedule is done as well the schedule is done
6. Return each output value from the `self` schedule in a `Left` and each output value from the `that` schedule in a `Right`

The `andThenEither` operator represents running one schedule "and then" running the other schedule.

The most general version returns an output type of `Either[Out, Out2]` to capture the fact that the output values could come from the first or the second schedule. When the output types of the two schedules are the same the `andThen` operator can be used to automatically unify the output types.

```
trait Schedule[-Env, -In, +Out] {
  def andThen[Env1 <: Env, In1 <: In, Out2 >: Out](that: Schedule[Env1, In1, Out2]): Schedul
}
```

This can be very useful when combining multiple schedules to avoid creating nested `Either` data types when the schedules being composed have a common output type such as `Duration`. You can also use the `++` operator as an alias for `andThen`.

To get a feeling for how we can use sequential composition of schedules, let's try to describe a schedule that will recur five times with a constant delay of one second and then ten times with exponential backoff.

We can implement the first part of the schedule as:

```
val schedule1: Schedule[Any, Any, Long] =
  Schedule.spaced(1.second) *> Schedule.recurs(5)
// schedule1: Schedule[Any, Any, Long] = zio.Schedule$$anon$1@a000dab
```

Similarly we can implement the second part of the schedule as:

```
val schedule2: Schedule[Any, Any, Duration] =
  Schedule.exponential(1.second) <* Schedule.recurs(10)
// schedule2: Schedule[Any, Any, Duration] = zio.Schedule$$anon$1@75493461
```

We can use the `++` operator to describe doing the first schedule and then doing the second schedule. The only thing we need to do is unify the output types so we can use `andThen` instead of `andThenEither`.

The first schedule outputs a `Long` representing the number of recurrences while the second returns a `Duration` representing the length of the last delay. In this case we will unify by transforming the output of the first schedule to a constant value of one second so that both schedules output the duration of the most recent delay.

```
val schedule3: Schedule[Any, Any, Duration] =
  schedule1.as(1.second) ++ schedule2
// schedule3: Schedule[Any, Any, Duration] = zio.Schedule$$anon$1@1569b75d
```

And that's it! We have already defined a relatively complex custom schedule in an extremely concise, declarative way where we were able to specify *what* we wanted done without having to write any messy, error prone retry logic ourselves describing *how* we wanted it done.

We can also see that if we wanted to describe even more complex schedules we could do it in exactly the same way because each of these schedules is *composable* with the constructors and operators we have defined. So we never have to worry about running into an issue where our schedule is "too complex" to describe and we have to implement it from scratch ourselves.

22.5.3 Alternative Schedules

The third way of composing schedules is alternative composition of schedules. Here we have two schedules that each know how to handle different input types and we combine them into one schedule that knows how to handle both input types by feeding the first type of inputs to the first schedule and the second type of inputs to the second schedule, returning the decision of whichever schedule was used.

Conceptually this corresponds to thinking of schedules as able to handle some set of inputs and creating schedules that are able to handle" bigger sets of inputs from schedules that are able to handle smaller sets of inputs.

The key operator for this way of composing schedules is `+++`.

```
trait Schedule[-Env, -In, +Out] { self =>
  def +++[Env1 <: Env, In2, Out2](
    that: Schedule[Env1, In2, Out2]
  ): Schedule[Env1, Either[In, In2], Either[Out, Out2]]
}
```

The `+++` operator works as follows:

1. For each input, determine whether it is a `Left` with an input `In` or a `Right` with an input `In2`
2. If the input is a `Left`, send the input to the `self` schedule and get its result
3. If the input is a `Right`, send the input to the `that` schedule and get its result
4. Return the decision of whether to continue and how long to delay for from whichever schedule was run
5. Wrap the result in a `Left` with an `Out` if the first schedule was used or a `Right` with an `Out2` otherwise

The `+++` operator represents running "either" the first schedule or the second schedule each time depending on what type of input is received.

There is also a variant `|||` that can be used if the output types of the two schedules is the same and simply unifies them. This can be useful to avoid creating nested `Either` data types when the outputs of the two schedules can be unified to a common supertype.

```
trait Schedule[-Env, -In, +Out] { self =>
  def |||[Env1 <: Env, Out1 >: Out, In2](
    that: Schedule[Env1, In2, Out1]
  ): Schedule[Env1, Either[In, In2], Out1]
}
```

This way of composing schedules is particularly useful for combining schedules that can handle narrower classes of inputs to create schedules that can handle

broader classes of inputs.

For example, we might have one schedule that knows how to retry connection errors and another schedule that knows how to handle rate limiting errors. We could combine the two schedules using `|||` to create a new schedule that can retry either type of errors.

```
sealed trait WebServiceError

trait ConnectionError extends WebServiceError
trait RateLimitError extends WebServiceError

lazy val connectionErrorSchedule: Schedule[Any, ConnectionError, Duration] =
  ???

lazy val rateLimitErrorSchedule: Schedule[Any, RateLimitError, Duration] =
  ???

lazy val webServiceErrorSchedule: Schedule[Any, WebServiceError, Duration] =
  (connectionErrorSchedule ||| rateLimitErrorSchedule).contramap {
    case connectionError: ConnectionError => Left(connectionError)
    case rateLimitError: RateLimitError   => Right(rateLimitError)
  }
```

As this example shows, the `contramap` operator can be extremely useful along with the `|||` operator to decompose a sum type into either of the types that the schedule created with `|||` can handle.

22.5.4 Function Composition Of Schedules

The fourth way of composing schedules is function composition of schedules. In this case we have two schedules and we feed every input to the first schedule, get its output, and then feed that output as the input to the second schedule to get a final decision of whether to continue and how long to delay for.

This corresponds to thinking of schedules as like functions and feeding the output of one schedule as the input to another.

The fundamental operator describing this way of composing schedules is `>>>`.

```
trait Schedule[-Env, -In, +Out] {
  def >>>[Env1 <: Env, Out2](that: Schedule[Env1, Out, Out2]): Schedule[Env1, In, Out2]
}
```

Notice the difference between the type signature here and the ones for the other types of schedule composition.

For operators like `&&`, `||`, and `++` the two schedules being combined accepted the same input type because we were always taking the inputs to the composed

schedule and sending them to both of the original schedules, either in parallel in the case of `&&` and `||` or sequentially in the case of `++`.

In contrast, here the `In` type of the second schedule is the `Out` type of the first schedule. This reflects that we are sending every input to the first schedule and then sending the output of the first schedule to the second schedule.

For example, if the output type of the first schedule was a `Duration`, the second schedule could consume those durations and apply further logic to determine whether and how long to delay.

This type of composition of schedules tends to be less common and is not used in implementing most schedule operators, so while it is good to be aware that this mode of composition exists generally using the previous concepts of composition will be enough to solve almost all schedule problems.

22.6 Implementation Of Schedule

The last section of this chapter describes the internal implementation of `Schedule`.

The `Schedule` data type is designed so that you should not have to know about its implementation to use `Schedule` to solve your own problems in implementing composable strategies for retrying and repeating effects. We have seen throughout this chapter how to use schedules to describe complex patterns of repetition and we have said very little about the internal implementation of a schedule other than that at each step a schedule returns a decision of whether to continue for a delay or be done, along with a summary output value.

However, if you are interested in implementing your own new `Schedule` operators, or are just curious how `Schedule` works under the hood, this section is for you. Otherwise, feel free to skip on to the conclusion and exercises at the end of this chapter.

A very slightly simplified implementation of `Schedule` is as follows:

```
import java.time.OffsetDateTime

trait Schedule[-Env, -In, +Out] {
  def step: StepFunction[Env, In, Out]
}

type StepFunction[-Env, -In, +Out] =
  (OffsetDateTime, In) => ZIO[Env, Nothing, Decision[Env, In, Out]]

sealed trait Decision[-Env, -In, +Out]

final case class Done[+Out](out: Out)
```

```
final case class Continue[-Env, -In, +Out](
  out: Out,
  interval: OffsetDateTime,
  next: StepFunction[Env, In, Out]
)
```

A `Schedule` is defined in terms of a `StepFunction`, which is just a function that takes an input `In` and an `OffsetDateTime`, representing the current time, and returns a ZIO effect producing a `Decision`.

```
type StepFunction[-Env, -In, +Out] =
  (OffsetDateTime, In) => ZIO[Env, Nothing, Decision[Env, In, Out]]
```

This indicates that in addition to having access to the input in deciding whether to continue, a schedule also has access to the current time. This facilitates using schedules for use cases like cron jobs where actions have to be scheduled to occur at a specific absolute point in time rather than just a relative point in time as in our examples with fixed or exponential delays.

The result of a `StepFunction` is a `Decision`, which is just an algebraic data type with two cases, `Done` and `Continue`.

The `Done` case indicates that the schedule no longer wants to continue and just returns a final summary output value. For example, the schedule we saw earlier that recurs five times with a fixed delay would emit `Done` after the last repetition.

The `Continue` case indicates that the schedule wants to continue. It includes the current summary output value, an interval to delay before continuing, and a new `StepFunction` to run next time with the next input.

```
sealed trait Decision[-Env, -In, +Out]

final case class Done[+Out](out: Out)
final case class Continue[-Env, -In, +Out](
  out: Out,
  interval: OffsetDateTime,
  next: StepFunction[Env, In, Out]
)
```

One trick when using this implementation is how we actually thread *state* through a schedule. Conceptually we have the ability to maintain some internal state as part of the schedule because the `Continue` case returns a new `StepFunction`, which can contain some updated state that is different that the state contained in the original `StepFunction`.

How do we go about implementing operators in terms of this though, especially when we often want to define schedules that recur many times, potentially forever? The answer is that we will typically define an inner `loop` function which defines the updating process we want to use and then repeatedly call that in each recurrence of the `Schedule`.

For example, here is what the implementation of the `unfold` constructor we saw earlier in this chapter looks like:

```
def unfold[A](a: => A)(f: A => A): Schedule[Any, Any, A] = {
  def loop(a: A): StepFunction[Any, Any, A] =
    (now, _) => ZIO.succeed(Decision.Continue(a, now, loop(f(a))))

  Schedule((now, _) => ZIO.effectTotal(a).map(a => Decision.Continue(a, now, loop(f(a)))))
}
```

We first define a `loop` function which will return a new `StepFunction` based on an input, in this case the previous `A` value. The `loop` functions returns a `StepFunction` that always continues and calls `loop` recursively in the new `StepFunction` within continue.

We then define a new `Schedule` that always continues and cals this `loop` function. This way the schedule will continue with an updated state reflecting the next recursive call to `loop` each time.

Since recursion is stack safe within `ZIO`, we do not have to worry that doing this will cause a stack overflow error.

This use of an inner `loop` function is common in implementing `Schedule` operators and you will likely see it yourself if you look at the implementation of existing operators or end up implementing your own at some point.

Once we have implemented a `Schedule`, to actually run it we can then do the following:

1. Run the effect and if it succeeds return its result immediately
2. If the effect fails, provide the failure along with the current time to the `StepFunction` in the `Schedule` and get its result
3. If the decision of the `StepFunction` is done then return the original failure, we have no more recurrences left
4. If the decision of the `StepFunction` is to continue then delay for the specified time and then repeat at the first step, using the new `StepFunction` produced by the decision to continue instead of the current one

ZIO abstracts much of this logic in a concept called the `Driver` which knows how to do the bookkeeping associated with running a schedule and that is then used in the implementation of operators like `repeat` or `retry` operators on ZIO.

```
final case class Driver[-Env, -In, +Out](
  next: In => ZIO[Env, None.type, Out],
  last: IO[NoSuchElementException, Out],
  reset: UIO[Unit]
)
```

A `driver` is defined in terms of three operators.

The `next` operator accepts an input `In` and either returns an `Out` value or fails

with `None` if the `Schedule` is already done and thus is not able to accept any more inputs.

The `last` operator returns the last output from the `Schedule`, or fails with a `NoSuchElementException` if there has been no output from the schedule yet.

Finally, the `reset` operator resets the `Schedule` to its original state.

The `Driver` interface makes it quite easy to define new ways of running a `Schedule` and is significantly easier than working with the `Schedule` interface directly to run a `Schedule`.

You shouldn't need this in most cases because existing operators are already defined for working with schedules. But for example if you are implementing a way to retry or repeat your own custom effect type using `Schedule` then `Driver` is the tool you should look to.

22.7 Conclusion

We have covered a lot of ground in this chapter. `Schedule` represents one of the more unique data types in ZIO, modeling a powerful solution for the problem of describing schedules for repeating or retrying effects. Because of this, we spent more time than we have in many chapters both building the intuition for working with schedules as well as going through the many different operators for creating, transforming, and composing schedules.

With this knowledge in hand, you should be in a good position to handle any problem you face regarding repeating or retrying effects.

You won't have to implement your own retry logic anymore or redo your implementation when the business requirements change. Instead you now know how to use some simple schedule constructors and ways of combining them to handle describing virtually every type of schedule you can imagine.

In the future, we will see how the `Schedule` data type can be used outside of just the context of `ZIO` itself. For example, we can use `Schedule` to describe how we would like to emit stream elements in *ZIO Stream* or how we would like to repeat a test to make sure it is stable in *ZIO Test*.

One of the advantages of `Schedule` being a data type that *describes* a strategy for repeating or retrying an effect is that we can potentially apply that strategy to any data type or problem we want, for example our own custom effect type or our own problem that uses a series of recurrences for some other purpose.

For now though, let's continue our journey into advanced error management. In the next chapter we will learn more about ZIO's capabilities for debugging with execution and fiber dumps. These tools make it dramatically easier for us to find and fix bugs in our code than with traditional frameworks for asynchronous and concurrent programming.

22.8 Exercises

Chapter 23

Advanced Error Management: Debugging

Chapter 24

Advanced Error Management: Best Practices

Chapter 25

Streaming: First Steps With ZStream

Streaming is an incredibly important paradigm for modern data driven applications. A very common use case is that we are constantly receiving new information and have to produce new outputs based on that information.

This could be:

- Updating a user interface based on a stream of user interaction events
- Issuing fraud alerts based on a stream of credit card usage data
- Generating real time insights from a stream of social media interactions

ZIO provides support for streaming through ZIO Stream, a feature rich, composable, and resource safe streaming library built on the power of ZIO.

In the course of learning about ZIO's support for streaming we will see how streaming presents new challenges. At the same time, we will see how many of the features of ZIO, such as its support for interruptibility and resource safety, provide the foundation for solving these problems in a robust and principled way.

25.1 Streams As Effectual Iterators

The core data type of ZIO Stream is `ZStream[R, E, O]`, an *effectual stream* that requires an environment `R`, may fail with an error `E`, and may succeed with *zero or more* values of type `O`.

The key distinction between a `ZIO` and a `ZStream` is that a `ZIO` always succeeds with a *single value* whereas a `ZStream` succeeds with *zero or more values* values, potentially infinitely many. Another way to say this is that a `ZIO` produces its result *all at once* whereas a `ZStream` produces its result *incrementally*.

To see what this means, consider how we would represent a stream of tweets using only a `ZIO`.

```
import zio._

trait Tweet

lazy val getTweets: ZIO[Any, Nothing, Chunk[Tweet]] =
  ???
```

The single value returned by a `ZIO` can be a collection, as shown here. However, a `ZIO` is either not completed or completed exactly once with a single value.

So `getTweets` is either incomplete, in which case we have no tweets at all, or complete with a single `Chunk` of tweets that will never change. But we know that there are always more tweets being produced in the future!

Thus, `getTweets` cannot represent getting a stream of tweets where new tweets are always coming in the future but can at most represent getting a snapshot of tweets as of some point in time.

To model a stream of tweets, we conceptually need to be able to evaluate `getTweets` once to get the tweets as of that point in time and then be able to repeatedly evaluate `getTweets` in the future to get all the incremental tweets since the last time we evaluated it.

The `ZStream` data type does just that.

```
trait ZStream[-R, +E, +O] {
  def process: ZManaged[R, Option[E], Chunk[O]]
}
```

A `ZStream` is defined in terms of one operator, `process`, which can be evaluated repeatedly to pull more elements from the stream.

Each time `process` is evaluated it will either:

1. Succeed with a `Chunk` of new `O` values that have been pulled from the stream,
2. Fail with an error `E`, or
3. Fail with `None` indicating the end of the stream.

The `process` is a `ZManaged` rather than a `ZIO` so that resources can be safely acquired and released in the context of the stream.

With this signature, you can think of `ZStream` as an *effectual iterator*.

Recall that the signature of `Iterator` is:

```
trait Iterator[+A] {
  def next: A
  def hasNext: Boolean
}
```

An `Iterator` is a data type that allows us to repeatedly call `next` to get the next value as long as `hasNext` returns `true`. Similarly, `ZStream` allows us to repeatedly evaluate `process` as long as it succeeds to get the next `Chunk` of values.

An `Iterator` is the fundamental *imperative* representation of collections of zero or more and potentially infinitely many values. Similarly, `ZStream` is the fundamental *functional* representation of collections of one or more and potentially infinitely many *effectual* values.

25.2 Streams As Collections

In addition to thinking of a `ZStream` as an effectual iterator, we can also conceptualize it as a *collection* of potentially infinitely many elements. This allows us to use many of the operators we are already familiar with from Scala's collection library, such as `map` and `filter`, when working with streams.

```
trait ZStream[-R, +E, +O] {
  def filter(f: O => Boolean): ZStream[R, E, O]
  def map[O2](f: O => O2): ZStream[R, E, O2]
}
```

Treating streams as collections lets us write higher level, more declarative code than working with them as effectual iterators directly, just like we prefer to use collection operators rather than the interface of `Iterator`. We can then use the underlying representation of a `ZStream` as an effectual iterator primarily to implement new operators on streams and to understand the implementation of existing operators.

25.2.1 Implicit Chunking

One thing to notice here is that while the underlying effectual iterator returns a `Chunk[O]` each time values are pulled, operators like `filter` and `map` work on individual `O` values. This reflects a design philosophy of *implicit chunking*.

For efficiency we want to operate on *chunks* of stream values rather than *individual* values. For example, we want to get a whole chunk of tweets when we pull from the stream instead of getting tweets one by one.

At the same time, chunking represents an implementation detail.

If we have a stream of tweets we know that it is actually composed of chunks of tweets with sizes depending on how we are pulling from the Twitter service and how quickly new tweets are being created. But we want to be able to program at a higher level where we are able to just think of the stream as a collection of tweets and have the streaming library manage the chunking for us.

ZIO Stream does exactly this.

Under the hood values in a stream are processed in chunks for efficiency. But as the user you are able to operate at the level of individual stream values, with ZIO Stream automatically translating these operations to efficient operations on chunks of values.

In certain situations it may be helpful to manually change the chunk size as an optimization. For example if you have incoming data that is being received individually it may be helpful to chunk it before further processing.

And you can always operate at the level of chunks if you want to. But the philosophy of ZIO Stream is that the user should have to manually deal with chunking only as an optimization and that in most cases the framework should automatically "do the right thing" with regard to chunking.

To see how this works and to start to get a feel for how we can implement collection operators in terms of the representation of a stream as an effectual iterator, let's look at how we can implement the `map` operator on `ZStream`.

```
trait ZStream[-R, +E, +O] { self =>
  def process: ZManaged[R, Option[E], Chunk[O]]
  def map[O2](f: O => O2): ZStream[R, E, O2] =
    new ZStream[R, E, O2] {
      def process: ZManaged[R, Option[E], Chunk[O2]] =
        self.process.map(_.map(f))
    }
}
```

Since `ZStream` is defined in terms of `process`, when implementing new operators we generally want to create a new `ZStream` and implement its `process` method in terms of the `process` method on the original stream.

In this case the implementation of `map` is quite simple. We just return a new stream that pulls values from the original stream and then applies the specified function to each value in the `Chunk`.

The implementation here was quite easy because the operation preserved the original chunk structure of the stream. We will see later that in some other cases ZIO Stream needs to do more work to make sure that users get the right behavior operating on stream values without having to worry about chunking themselves.

25.2.2 Potentially Infinite Streams

The other important distinction to keep in mind when dealing with collection operators on streams is that streams are *potentially infinite*. As a result, some collection operators defined on finite collections do not make sense for streams.

For example, consider the `sorted` operator in the Scala collection library that sorts elements of a collection based on an `Ordering` defined on the elements of that collection.

```
val words: List[String] =
  List("the", "quick", "brown", "fox")
// words: List[String] = List("the", "quick", "brown", "fox")

val sorted: List[String] =
  words.sorted
// sorted: List[String] = List("brown", "fox", "quick", "the")
```

There is no corresponding `sorted` operator defined on `ZStream`. Why is that?

Sorting is an operation that requires examining the entire original collection before determining even the first element of the resulting collection.

The resulting collection must include the "smallest" element in the `Ordering` first, but it is always possible that the last element in the original collection is the "smallest". So we cannot determine even the first element in the resulting collection without examining every element in the original collection.

This is fine when the original collection is finite, such as a `List`, because we have all the collection elements and can simply examine them. But in a streaming application implementing a `sorted` operator would require not only waiting to emit any value until the original stream had terminated, which might never happen, but also buffering a potentially unlimited number of prior values, creating a potential memory leak.

For this reason, you will only see collection operators defined on `ZStream` that make sense for potentially infinite streams. Another way to think of this is that as we said above streams produce their results *incrementally* so it only makes sense to define operators that return streams if those operators can also produce their results *incrementally*.

25.2.3 Common Collection Operators On Streams

Here are some common collection operators that you can use to transform streams:

- `collect` - map and filter stream values at the same time
- `collectWhile` - transform stream values as long as the specified partial function is defined
- `concat` - pull from the specified stream after this stream is done
- `drop` - drop the first `n` values from the stream
- `dropUntil` - drop values from the stream until the specified predicate is true
- `dropWhile` - drop values from the stream while the specified predicate is true
- `filter` - retain only values satisfying the specified predicate
- `filterNot` - drop all values satisfying the specified predicate
- `flatMap` - create a new stream for each value and flatten them to a single stream

- `map` - transform stream values with the specified function
- `mapAccum` - transform stream values with the specified stateful function
- `scan` - fold over the stream values and emit each intermediate result in a new stream
- `take` - take the first `n` values from the stream
- `takeRight` - take the last `n` values from the stream
- `takeUntil` - take values from the stream until the specified predicate is true
- `takeWhile` - take values from the stream while the specified predicate is true
- `zip` - combine two streams point wise

Many of these operators also have effectual variants with the `M` suffix. For example, while the `map` operator can be used to transform each stream value with a function, the `mapM` operator can be used to transform each stream value with an effectual function.

When stream operators have effectual variants, they may also have variants that perform effects in parallel with the `Par` suffix.

For instance, there is also a `mapMPar` variant of the `mapM` operator. Whereas `mapM` performs the effectual function for each stream value before proceeding to process the next, the `mapMPar` operator processes up to a specified number of stream elements at a time in parallel.

25.3 Constructing Basic Streams

ZIO Stream comes with many different ways of constructing streams. We will see more specialized variants in the coming chapters, but here we will focus on basic constructors to get started working with streams.

25.3.1 Constructing Streams From Existing Values

One of the simplest stream constructors is `fromIterable`, which takes an `Iterable` and returns a stream containing the values in that `Iterable`.

```
object ZStream {
  def fromIterable[O](os: Iterable[O]): ZStream[Any, Nothing, O] =
    ???
}

lazy val stream: ZStream[Any, Nothing, Int] =
  ZStream.fromIterable(List(1, 2, 3, 4, 5))
```

The `apply` method on the `ZStream` companion object behaves similarly, constructing a stream from a variable length list of values. So the example above could also be written as `ZStream(1, 2, 3, 4, 5)`.

The streams created using these constructors are in some ways not the most interesting because they are always finite and do not involve performing effects to generate the stream values. However, these constructors can be very useful when you are getting started with ZIO Stream to create basic streams and see the results of transforming them in different ways.

In addition, even when you are an expert on streams these can be useful constructors to compose with other operators. For example, you could create a stream from a list of file names, then create a new stream describing reading incrementally from each of those files.

25.3.2 Constructing Streams From Effects

Another set of basic stream constructors are `fromEffect` and `managed`, which let us lift a `ZIO` or `ZManaged` effect into a single element stream.

```
object ZStream {
  def fromEffect[R, E, O](zio: ZIO[R, E, O]): ZStream[R, E, O] =
    ???
  def managed[R, E, O](managed: ZManaged[R, E, O]): ZStream[R, E, O] =
    ???
}
```

```
import zio._
import zio.console._
import zio.stream._

val helloStream: ZStream[Console, Nothing, Unit] =
  ZStream.fromEffect(console.putStrLn("hello")) ++
    ZStream.fromEffect(console.putStrLn("from")) ++
    ZStream.fromEffect(console.putStrLn("a")) ++
    ZStream.fromEffect(console.putStrLn("stream"))
// helloStream: ZStream[Console, Nothing, Unit] = zio.stream.ZStream$$anon$1@4441d86e
```

These constructors only create single element streams but now let us work with `ZIO` and `ZManaged` effects within the stream context. We can combine them with other operators to create much more complex streams, for example reading from one file and then creating a new stream that reads incrementally from each file listed in the original file.

25.3.3 Constructing Streams From Repetition

Building on these operators, the `repeat`, `repeatEffect`, and `repeatEffectOption` constructors and their `Chunk` variants get us even closer to constructing complex streams directly.

```
object ZStream {
```

```
  def repeat[O](o: => O): ZStream[Any, Nothing, O] =
    ???

  def repeatEffect[R, E, O](zio: ZIO[R, E, O]): ZStream[R, E, O] =
    ???

  def repeatEffectChunk[R, E, O](
    zio: ZIO[R, E, Chunk[O]]
  ): ZStream[R, E, O] =
    ???

  def repeatEffectOption[R, E, O](
    zio: ZIO[R, Option[E], O]
  ): ZStream[R, E, O] =
    ???

  def repeatEffectChunkOption[R, E, O](
    zio: ZIO[R, Option[E], Chunk[O]]
  ): ZStream[R, E, O] =
    ???
}

lazy val ones: ZStream[Any, Nothing, Int] =
  ZStream.repeat(1)

trait Tweet

lazy val getTweets: ZIO[Any, Nothing, Chunk[Tweet]] =
  ???

lazy val tweetStream: ZStream[Any, Nothing, Tweet] =
  ZStream.repeatEffectChunk(getTweets)
```

These constructors let us repeat either a single value or performing a single effect.

The basic variants create streams that repeat those effects forever, while the `Option` variants return streams that terminate when the provided effect fails with `None`.

We can also see that there are variants that take effects returns chunks as arguments with a `Chunk` suffix. These let us create streams from effects that return chunks of values, like all the tweets since the last time we ran the effect.

25.3.4 Constructing Streams From Unfolding

The final set of stream constructor variants we will look at here are the `unfold` variants.

```
object ZStream {

  def unfold[S, A](
    s: S
  )(f: S => Option[(A, S)]): ZStream[Any, Nothing, A] =
    ???

  def unfoldM[R, E, A, S](
    s: S)
  (f: S => ZIO[R, E, Option[(A, S)]]): ZStream[R, E, A] =
    ???

  def unfoldChunk[S, A](
    s: S
  )(f: S => Option[(Chunk[A], S)]): ZStream[Any, Nothing, A] =
    ???

  def unfoldChunkM[R, E, A, S](
    s: S
  )(f: S => ZIO[R, E, Option[(Chunk[A], S)]]): ZStream[R, E, A] =
    ???
}
```

These are very general constructors that allow constructing a stream from an initial state `S` and a state transformation function `f`. The way these constructors work is as follows:

1. The first time values are pulled from the stream the function `f` will be applied to the initial state `s`
2. If the result is `None` the stream will end
3. If the result is `Some` the stream will return `A` and the next time values are pulled the first step will be repeated with the new `S`

The `S` type parameter allows us to maintain some internal state in constructing the stream. For example, here is how we can use `unfold` to construct a `ZStream` from a `List`.

```
def fromList[A](list: List[A]): ZStream[Any, Nothing, A] =
  ZStream.unfold(list) {
    case h :: t => Some((h, t))
    case Nil    => None
  }
```

In this example, the state we maintain is the remaining list elements.

At each step, if there are remaining list elements we emit the first one and return a new state with the remaining elements. If there are no remaining list elements we return `None` to signal the end of the stream.

One important thing to note when working with these constructors is that each step of unfolding the stream is only evaluated as values are pulled from the stream, so `unfold` can be used to safely describe streams that continue forever. Of course, we can also describe terminating the stream early by returning `None` in the state transformation function as we saw above.

In addition to the basic variant of `unfold` there is also an effectual variant, `unfoldM`, which allows performing an effect in the state transformation function. This allows describing many types of streams, for example reading incrementally from a data source while maintaining some cursor that represents where we currently are in reading from the data source.

Finally, there are `Chunk` variants of both `unfold` and `unfoldM`. These variants allow returning a `Chunk` of values instead of a single value in each step and will be more efficient when stream values can naturally be produced in chunks.

25.4 Running Streams

Once we have described our streaming logic using `ZStream` we need to *run* our stream to actually perform the effects described by the stream.

Running a stream always proceeds in two steps:

1. Run the `ZStream` to a `ZIO` effect that describes running the stream using one of the operators discussed in this section
2. Run the `ZIO` effect using `unsafeRun` or one of its variants that we learned about earlier in this book

The first step takes the higher level `ZStream` program that describes *incrementally* producing *zero or more* values and "compiling it down" to a single `ZIO` effect that describes running the streaming program *all at once* and producing a *single value*. The second step actually runs that `ZIO` effect.

Because a `ZStream` produces *zero or more* and potentially infinitely many values, there are a variety of ways we could *run* a stream to produce a single value. For example, we could:

- Run the stream to completion, discarding its result and returning the `Unit` value
- Run the stream to completion, collecting all of its values into a `Chunk`
- Run the stream to return the first value without running the rest of the stream
- Run the stream to completion and return the last value
- Fold over the stream values to produce some summary value, consuming only as many values as are necessary to compute the summary

All of these strategies for running a stream could make sense depending on our application and ZIO Stream provides support for each of them. The important thing is picking the right one for your application.

A helpful first question to ask is whether your stream is *finite* or *infinite*. Several ways of running a stream, such as collecting all of its values into a `Chunk`, only make sense if the stream is finite.

If your stream is infinite then your strategy for running the stream should have some way of terminating without evaluating the entire stream, unless the point of your application is just to run the stream forever.

Beyond that, it is helpful to think about what kind of result you want from running the stream.

If your entire program is described as a stream then you may not need any result from the stream at all other than to run it. If you do need a result, do you need all of the stream values, the first stream value, the last stream value, or some more complex summary?

25.4.1 Running A Stream As Folding Over Stream Values

All of the ways of running a stream can be conceptualized as some way of folding over the potentially infinite stream of values to reduce it to a summary value.

The most general operator to do this is `foldWhile`, which allows reducing a stream to a summary value and also signaling early termination.

```
trait ZStream[-R, +E, +O] {
  def foldWhile[S](
    s: S
  )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
}
```

You can think of `foldWhile` on `ZStream` as like `foldLeft` in the Scala collection library with an additional parameter supporting early termination. Note that there is also an effectual version `foldWhileM` that allows performing effects within the fold as well as a `foldWhileManaged` variant that return a `ZManaged` instead of a `ZIO`.

Let's see how we can use `foldWhile` to implement some of the other more specialized operators for running streams.

25.4.2 Running A Stream For Its Effects

The first operator we will implement is `runDrain`, which runs a stream to completion purely for its effects.

```
trait ZStream[-R, +E, +O] {
  def foldWhile[S](
```

```
    s: S
  )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
  def runDrain: ZIO[R, E, Unit] =
    foldWhile(())(_ => true)((s, _) => s)
}
```

The `runDrain` operator always continues but ignores the values produced by the stream, simply returning an effect that succeeds with the `Unit` value.

This operator is useful if your entire logic for this part of your program is described as a stream. For example, we could have a stream that describes getting tweets, performing some filtering and transformation on them, and then writing the results to a database.

In this case running the stream would be nothing more than running each of the effects described by the stream. There would be no further meaningful result of this stream other than the effect of writing the results to the database.

The `runDrain` operator can be useful even if the stream is infinite if the application consists of evaluating the stream forever. For example, the same tweet analysis application described above could run, potentially forever, continuously getting new tweets, transforming them, and writing them to the database.

One variant of this that can be particularly useful is the `foreach` operator.

```
trait ZStream[-R, +E, +O] {
  def foldWhileM[R1 <: R, E1 >: E, S](
    s: S
  )(cont: S => Boolean)(f: (S, O) => ZIO[R1, E1, S]): ZIO[R1, E1, S]
  def foreach[R1 <: R, E1 >: E, Any](
    f: O => ZIO[R1, E1, Any]
  ): ZIO[R1, E1, Unit] =
    foldWhileM[R1, E1, Unit](())(_ => true)((_, o) => f(o).unit)
}
```

The operator is similar to `runDrain` in that it runs a stream to completion for its effects, but now it lets us specify an effect we want to perform for every stream element. For example, if we had a stream of tweets we could call `foreach` to print each tweet to the console or write each tweet to a database.

One feature of `foreach` that is particularly nice is the way it interacts with Scala's syntax for *for comprehensions*.

Recall that a *for comprehension* that ends with a `yield` is translated into a series of `flatMap` calls followed by a final `map` call.

```
for {
  x <- List(1, 2)
  y <- List(x, x * 3)
} yield (x, y)
// res1: collection.immutable.List[(Int, Int)] = List(
```

```
//   (1, 1),
//   (1, 3),
//   (2, 2),
//   (2, 6)
// )
```

When a *for comprehension* does not end in a `yield` it is translated into a series of `foreach` calls instead.

```
for {
  x <- List(1, 2)
  y <- List(x, x * 3)
} println((x, y))
// (1,1)
// (1,3)
// (2,2)
// (2,6)
```

Normally we avoid using *for comprehensions* in this way because the standard definition of `foreach` on a data type like `List` is:

```
trait List[+A] {
  def foreach[U](f: A => U): Unit
}
```

This method performs side effects that are not suspended in an effect type like ZIO that *describes* doing something rather than immediately doing that thing. As such, it interferes with our ability to reason about our programs and build more complex programs from simpler ones as discussed in the first chapter.

However, as we saw above the signature of the `foreach` method on `ZStream` returns a `ZIO` effect and so it is safe to use. Using this, we can write code like:

```
val effect: ZIO[Console, Nothing, Unit] =
  for {
    x <- ZStream(1, 2)
    y <- ZStream(x, x + 3)
  } console.putStrLn((x, y).toString)
// effect: ZIO[Console, Nothing, Unit] = zio.ZIO$CheckInterrupt@48c82eb6
```

This now just describes running these two streams and printing the values they produce to the console.

You don't have to use this syntax, but it can create a quite ready way of saying "run this stream by doing this effect for each element of the stream".

25.4.3 Running A Stream For Its Values

Another common way of running a stream is to return either the first or the last value of the stream.

```
trait ZStream[-R, +E, +O] {
  def foldWhile[S](
    s: S
  )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
  def runHead: ZIO[R, E, Option[O]] =
    foldWhile[Option[O]](None)(_.isEmpty)((_, o) => Some(o))
  def runLast: ZIO[R, E, Option[O]] =
    foldWhile[Option[O]](None)(_ => true)((_, o) => Some(o))
}
```

The `runHead` operator runs the stream to produce its first value and then terminates immediately, returning either `Some` with the first value if the stream succeeds with at least one value or `None` otherwise. The `runLast` operator runs the stream to produce its last value, returning `Some` with the last value if the stream succeeds with at least one value or `None` otherwise.

These operators can be particularly useful if the stream has already been filtered so that the first or last values satisfy desired properties.

For example, a stream could represent geolocation data with later values in the stream having more precision. If the stream is then filtered for values having the desired precision `runHead` would represent an efficient strategy for continuing to pull new geolocation data until the desired precision is reached.

We could also collect all of the values produced by the stream into a `Chunk`.

```
trait ZStream[-R, +E, +O] {
  def foldWhile[S](
    s: S
  )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
  def runCollect: ZIO[R, E, Chunk[O]] =
    foldWhile[Chunk[O]](Chunk.empty)(_ => true)((s, o) => s :+ o)
}
```

This runs the stream to completion, collecting all values produced into a `Chunk`.

This can be particularly useful if the stream is already known to be a finite size because we previously called an operator like `take`. For example, if we just want to view a sample of tweets about a topic for inspection we could just use `take(100)` to take the first hundred values and then `runCollect` to collect all of them into a collection that we could print to the console.

There are also a couple of more specialized ways of running streams.

```
import scala.math.Numeric

trait ZStream[-R, +E, +O] {
  def foldWhile[S](
    s: S
  )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
```

```
  def runCount: ZIO[R, E, Long] =
    foldWhile(0L)(_ => true)((s, _) => s + 1L)
  def runSum[O1 >: O](implicit ev: Numeric[O1]): ZIO[R, E, O1] =
    foldWhile(ev.zero)(_ => true)(ev.plus)
}
```

The `runCount` operator runs a stream to completion, returning the total number of values produced by the stream. The `runSum` operator is only defined for numeric streams, and simply returns the sum of all the stream values.

25.5 Type Parameters

Like `ZIO` and `ZManaged`, `ZStream` has three type parameters:

- `R` represents the set of services requires to run the stream
- `E` represents the type of errors the stream can potentially fail with
- `O` represents the type of values output by the stream

These type parameters are analogous to the ones we have already learned about with regarding to `ZIO` and `ZManaged` so we have been able to get this far with saying very little about them other than their basic definition.

However, it is helpful to highlight here some of the further similarities as well as a couple of differences.

25.5.1 The Environment Type

The environment type has exactly the same meaning with respect to `ZStream` as it does with respect to `ZIO`, describing the set of services the stream needs to run. It also has almost all of the same operators.

In particular, we can access the environment using `ZStream.environment`, which returns a single element stream that contains the environment.

We can also provide the environment using the same operators we are used to, including `provide`, `provideLayer`, `provideSomeLayer`, and `provideCustomLayer`.

The only significant difference when working with the environment type for `ZStream` is that there are a couple of additional variants of the `access` operators:

- `access` - Access the environment and return a value that depends on it
- `accessM` - Access the environment and return a `ZIO` that depends on it
- `accessManaged` - Access the environment and return a `ZManaged` that depends on it
- `accessStream` - Access the environment and return a `ZStream` that depends on it

All of these operators return a `ZStream` so they just provide slightly more convenient variants if you want to do something with the value in the environment that returns an effect, a managed effect, or a stream.

25.5.2 The Error Type

Just like with `ZIO`, the `E` type parameter represents the potential errors that the stream can fail with.

Many of the same operators defined to deal with errors on `ZIO` also exist on `ZStream`. For instance, you can use `catchAll` or `catchSome` to recover from errors in a stream and fall back to a new stream instead.

The main thing to be aware of with the error type is that if a stream fails with an error it is not well defined to pull from that stream again.

For example, if we have a stream that fails with an `E` we can use an operator like `catchAll` to ignore that error and instead execute the logic of some new stream instead. But it is not well defined to ignore that error and then pull from the original stream again.

25.6 Conclusion

With the material in this chapter you are already well on your way to mastering streaming with ZIO Stream.

At this point you know the underlying representation of a `ZStream` as an effectual iterator and how this represents the fundamental *functional* analogue to an `Iterator` to describe a collection of potentially infinitely many *effectual* values.

You also understand many of the basic operations on `ZStream` that treat a `ZStream` as a collection of elements, including key ideas such as implicit chunking and the fact that streams represent potentially infinite collections of values.

You also know how to create streams using basic constructors and how to run streams.

In the next chapter we will work on applying this knowledge to the concrete domain of working with files. In the process we will see how we can use additional specialized stream constructors to create a stream from a file as well as introduce the concept of *sinks*, which can be thought of as composable strategies for running streams for their effects.

From there we will step back and look at some of the more advanced operators for working with streams, dividing these operators between those that transform one stream to another and those that combine two or more streams into a single stream.

Finally, we will dive deep into each of the other major data types in ZIO Stream: sinks, transducers, and summaries. We will understand how they are

implemented and how they can help us solve even more problems in streaming. With that introduction, let's dive in!

Chapter 26

Streaming: Next Steps With ZStream

Chapter 27

Streaming: Creating Custom Streams

Chapter 28

Streaming: Transforming Streams

Chapter 29

Streaming: Combining Streams

Chapter 30

Streaming: Transducers

Chapter 31

Streaming: Sinks

Chapter 32

Streaming: Summaries

Chapter 33

Testing: Basic Testing

In this chapter, we will begin our detailed discussion of ZIO Test and testing ZIO applications.

We already learned how to write basic tests in the first section of this book and have gotten some practice throughout as we have written tests for various programs we wrote. Therefore, the focus of this and subsequent chapters will be more on how ZIO Test is implemented as well as how it can be used to solve various problems in testing that may come up in your day to day work.

As you read these chapters, in addition to focusing on how to solve specific problems in testing, pay attention to how ZIO Test is using the power of ZIO to solve these problems.

ZIO Test is ultimately a library trying to solve problems in the specific domain of testing. How does ZIO Test leverage the power of ZIO to solve problems in this domain?

This will give you a good opportunity to review many of the features that we have learned about earlier in this book to reinforce and deepen your understanding.

33.1 Tests As Effects

Building on the discussion above about leveraging the power of ZIO, the core idea of ZIO Test is to treat every test as an effect.

```
import zio._
import zio.test._

type ZTest[-R, +E] = ZIO[R, TestFailure[E], TestSuccess]
```

That is, a test is just a workflow that either succeeds with a `TestSuccess`, indicating that the test passed, or fails with a `TestFailure`, indicating that the

test failed.

The `TestSuccess`, in turn, may be either a `TestSuccess.Succeeded`, indicating that the test actually passed, or a `TestSuccess.Ignored`, indicating that the test only "passed" because we ignored it, for example because we wrote a unit test for a feature but have not implemented that feature yet. Capturing these as separate data types lets the test framework reported succeeded tests differently from ignored tests when we display test results.

A `TestFailure` is parameterized on an error type `E` and can also have two cases. It can either be a `TestFailure.Assertion`, indicating that an assertion was not satisfied or a `TestFailure.Runtime` indicating that some error `E` occurred while attempting to run the test.

Conceptualizing a test as an effect has several benefits.

First, it allows us to avoid having to call `unsafeRun` all the time when our tests involve effects.

In other test frameworks where a test is not a `ZIO` value or another functional effect, we have to unsafely run our effects to get an actual value that we can make an assertion about. For example, in ScalaTest if we wanted to test that `ZIO.succeed` constructs an effect that succeeds with the specified value, we would have to do it like this:

```
import org.scalatest._

class ScalaTestSpec extends FunSuite {
  test("addition works") {
    assert(Runtime.default.unsafeRun(ZIO.succeed(1)) === 2)
  }
}
```

In addition to being annoying to write all the time, having to call `unsafeRun` like this can be a source of subtle errors if we forget to call `unsafeRun` when we are supposed to or accidentally nest multiple invocations of `unsafeRun`. See https://github.com/typelevel/fs2/issues/1009 for an example of this problem.

Tests written this way are also not safe on a cross-platform basis because `unsafeRun` blocks for the result of the potentially asynchronous ZIO effect to be available, so this will crash on Scala.js where blocking is not supported. We can work around this problem by unsafely running each ZIO effect to a `Future`, but then we still have all the problems above.

```
import org.scalatest._

class ScalaTestSpec extends AsyncFunSuite {
  test("addition works") {
    assert(Runtime.default.unsafeRunToFuture(ZIO.succeed(1)).map(_ === 2))
```

```
  }
}
```

Second, it allows us to unify testing effectual and non-effectual code.

Sometimes we want to write tests involving effects, for example to verify that the value we get from a `Queue` is what we expect. Other times we want to write tests that don't involve effects, for example that a collection operator we have implemented returns the expected value.

Ideally we would like writing these two types of tests to be very similar so we could seamlessly switch between one and the other for the specific test we were writing.

But we saw above that if tests are not effects then we need to unsafely run each of our effects to a data type like a `Future` and then use `map` to make an assertion in the context of that `Future`, whereas for tests that don't involve effects we can write simple assertions. So now we need to be quite careful with which tests involve effects and which do not, making sure to unsafely run our effects in the right way.

In contrast, if our tests are effects then it is very easy to create a test that does not involve effects because we can always put values into effects with constructors like `ZIO.succeed`, `ZIO.fail`, and `ZIO.effectTotal`. Let's see what this would look like.

```
lazy val pure: Either[TestFailure[Nothing], TestSuccess] =
  ???

lazy val effectual: ZIO[Any, TestFailure[Nothing], TestSuccess] =
  ZIO.fromEither(pure)
```

The third benefit, and one that we will see throughout our discussion of ZIO Test, is that making tests effects lets us use all of the functionality of ZIO to help us solve problems in testing.

For example, one common piece of functionality we want to provide in a testing framework is the ability to time out tests. We may accidentally write code that doesn't terminate and we would like the test framework to time out that test at some point and report that instead of us having to kill the application and try to figure out what happened.

Other test frameworks support this but often to a very limited extent.

Recall from our discussion of `Future` in the first chapter that `Future` does not support interruption. So test frameworks like ScalaTest can potentially just stop waiting for the result of a `Future` if it is taking too long and report that the test timed out, but that test may still be running in the background consuming system resources and potentially causing our application to crash at some point.

In contrast, we know that `ZIO` supports safe interruption, and there is even a

built in operator `ZIO#timeout` to safely interrupt an effect if it takes more than a certain amount of time to execute. So just by using the `timeout` operator that already exists on ZIO we get the ability to interrupt an effect, preventing it from doing further work, and even running any finalization logic for resources used in the test.

At this point the built in functionality in ZIO in providing most of the functionality that we need to time out tests out of the box. All that is left for us as implementors of the test framework is some minor bookkeeping to capture the appropriate information about that failure for reporting.

Safe resource usage is another great example of the functionality that we get "for free" from ZIO. Often in our tests we will need to use some kind of resource, whether it is creating an `ActorSystem`, setting up and tearing down a database, or creating a Kafka client.

This is a very hard problem to solve in a safe way in other testing frameworks because safe resource usage for asynchronous code is built upon well-defined support for interruption and the `bracket` operator, and `Future` has neither of those. As a result other testing frameworks often need to introduce additional concepts such as fixtures and separate actions to be performed before and after tests that add complexity and are not safe in the presence of interruption.

By making tests effects we will see later in this section how we can use all the tools we are familiar with like `bracket`, `ZManaged` and `ZLayer` to make safe resource handling a breeze and support a variety of different scenarios of shared or per test resources.

With this introduction, let's look in more detail at how ZIO Test converts a test we write into the `ZTest` data type we described above.

To do this, we will introduce one other data type that ZIO Test uses internally, `BoolAlgebra`. While the name sounds somewhat intimidating, this is just a data type that allows us to capture the result of an assertion and combine multiple assertions together while retaining information about all of them.

It looks roughly like this:

```
sealed trait BoolAlgebra[+A]

final case class And[+A](left: BoolAlgebra[A], right: BoolAlgebra[A]) extends BoolAlgebra[A]
final case class Not[+A](result: BoolAlgebra[A]) extends BoolAlgebra[A]
final case class Or[+A](left: BoolAlgebra[A], right: BoolAlgebra[A]) extends BoolAlgebra[A]
final case class Success[+A](value: A) extends BoolAlgebra[A]
```

`BoolAlgebra` is paramaterized on some type `A` that typically contains details about the result of making an assertion about a value. By convention `Success` indicates that the assertion passed and `Not` indicates that the assertion failed.

This makes it easy for us to do things like require multiple assertions to be true, negate an assertion, or express that one assertion being true implies that another

must also be true. We will use this data type more when we do our deep dive on assertions, but for now this gives us enough of an understanding to complete our discussion of tests as effects.

How then does ZIO Test take the test we write in `test` or `testM` and convert it to the `ZTest` data type we saw above?

Every test we write in the body of `testM` returns a `ZIO[R, E, TestResult]`, where `TestResult` is a type alias for `BoolAlgebra[FailureDetails]` and `FailureDetails` contains details about the result of making a particular assertion. Here is how that gets converted into a `ZTest`:

```
import zio._
import zio.test._

object ZTest {
  def apply[R, E](assertion: => ZIO[R, E, TestResult]): ZIO[R, TestFailure[E], TestSuccess]
    ZIO.effectSuspendTotal(assertion).foldCauseM(
      cause => ZIO.fail(TestFailure.Runtime(cause)),
      result =>
        result.failures match {
          case None => ZIO.succeed(TestSuccess.Succeeded(BoolAlgebra.unit))
          case Some(failures) => ZIO.fail(TestFailure.Assertion(failures))
        }
    )
}
```

We will walk through this just to be sure we understand how the tests we write get converted into test results.

The first thing to notice is that we accept `assertion` as a by name parameter and immediately wrap it in `ZIO.effectSuspendTotal`. We do this because we want to make sure we capture any thrown exceptions in the creation of `Assertion` and convert them to test failures so they can be properly reported by the test framework.

We might think that we don't have to worry about exceptions being thrown because the type of `assertion` is `ZIO[R, E, TestResult]`, so any exceptions should be captured in the `ZIO` effect as either a `Cause.Fail` with an error `E` or a `Cause.Die` with some `Throwable`. And in fact if our users were always doing the right thing that would be the case.

But there is nothing to stop a user from doing something like this:

```
def bomb[R, E]: ZIO[R, E, TestResult] =
  throw new Exception("Uh, oh!")
```

Clearly, ideally our users would always properly manage their own exceptions, for example wrapping the exception above in a `ZIO` constructor. But users may not always do this and so at the edges of our application when users can provide

use with arbitrary effects that may do things we don't want it may make sense for us to take special measures to handle those effects ourselves.

In this case, by accepting `assertion` as a by name parameter and then immediately wrapping it in `effectSuspendTotal` we ensure that even if an exception is thrown by `assertion`, that assertion will still at least result in a `ZIO` effect that dies with a `Throwable`, rather than throwing an uncaught exception that must be handled by a higher level of the application. We then use `foldCauseM` to allow us to handle both the potential failure and the success.

If a failure occurred, we simply convert it into a `ZIO` effect that fails with a `TestFailure.Runtime` containing the exception. This is the situation where a runtime failure occurs during test evaluation.

If no failure occurs, then we get back a `TestResult`, which is a `BoolAlgebra[FailureDetails]`. We then call the `failures` operator on `BoolAlgebra`, which returns `None` if no assertion failures occurred or `Some` with a nonempty list of assertion failures if one or more expectations were not satisfied.

If no assertion failures occurred then all expectations were satisfied and the test passed! If one or more assertion failures occurred then the test failed and we package that up into a `TestFailure.Assertion` containing the failures that did occur so we can report them.

Going back to our earlier discussion, with this implementation it is also extremely easy for us to convert pure assertions into tests. The body of `test` just returns a `TestResult`, so all we have to do is wrap it in `ZIO.effectTotal` to get a `ZIO[Any, Nothing, TestResult]` and then we can feed it into the same `apply` method of `ZTest` we saw above.

At this point you should have a solid understanding of how the assertions you write in the body of `testM` and `test` get converted to `ZTest` values that are effects describing tests. The next step is to see how ZIO Test supports combining multiple `ZTest` values to create Specs.

33.2 Specs As Recursively Nested Collections Of Tests

In ZIO Test, a `ZSpec` is a tree like data structure that can be either a single test or a suite containing one or more other Specs.

```
sealed trait ZSpec[-R, +E]

final case class Suite[-R, +E](label: String, specs: ZManaged[R, TestFailure[E], Vector[ZSpe
final case class Test[-R, +E](label: String, test: ZIO[R, TestFailure[E], TestSuccess])
```

Note that the implementation here has been slightly simplified from the actual, but this should give you the idea. A `ZSpec` can either be a `Test` containing a

single test along with a label or a `Suite` containing a label and a collection of `ZSpec` values.

In this way a `ZSpec` can support arbitrary nesting, allowing users a great deal of flexibility to define and organize their tests the way they want.

For example, you can have a very "flat" spec where you have a single suite containing a large number of tests that are all at the same logical level. You can also have a more hierarchical spec, where you group tests into different subcategories that are relevant for you.

You can have as many layers of nesting as you want and you can have a large number of tests at the same level and then several suites at that level that contain nested tests, so there is a great deal of flexibility for you to organize your tests the way that makes sense for you.

One thing to notice here that we will come back to in our discussion of using resources in tests is the fact that in a suite each collection of specs is wrapped in a `ZManaged` value. This is important because it gives us the capability to describe resources that are shared across all tests in a suite.

For example, it may be expensive to create an `ActorSystem` so we may want to create it only once and then use it in multiple tests. Representing the collection of tests as a `ZManaged` value allows us to construt the `ActorSystem` at the beginning of running the suite of tests, with the guarantee that it will be properly shutdown as soon as the suite is done, no matter how that happens.

33.3 Conclusion

With the material in this chapter you should have a better understanding of the "how" of testing, including how ZIO Test converts the tests you write into effects that describe those tests and how ZIO Test supports grouping tests into specs.

In the next chapter we will learn more about ZIO Test's `Assertion` data type. We were introduced to assertions before but we will see how assertions are actually implemented and how testing an assertion produces the `TestResult` data type that we learned about in this chapter.

We will also go through the full variety of assertions. There are many assertions for testing specific expectations about many data types, which can be a blessing because there is often specific logic here that is already implemented for us but can also sometimes make it a challenge to find exactly the right assertion for our use case.

We will present a taxonomy of the different assertions that exist to make it as easy as possible for you to find the write assertion for the case at hand, and also go through how you can implement your own assertions to factor out expectations that you want to test for multiple times.

33.4 Exercises

Chapter 34

Testing: Assertions

Chapter 35

Testing: The Test Environment

Chapter 36

Testing: Test Aspects

Chapter 37

Testing: Using Resources In Tests

Chapter 38

Testing: Property Based Testing

One of the great features of ZIO Test is its out of the box support for property based testing.

Property based testing is an approach to testing where the framework generates test cases for us instead of having to come up with test cases ourselves.

For example, we might want to check that integer addition is associative, that is that `(x + y) + z` is equal to `x + (y + z)`. In a traditional testing approach we would test this by choosing particular values of `x`, `y` and `z` and verifying that the expectation is satisfied.

```
import zio.test._
import zio.test.Assertion._

test("integer addition is associative") {
  val (x, y, z) = (1, 2, 3)
  val left = (x + y) + z
  val right = x + (y + z)
  assert(left)(equalTo(right))
}
// res0: ZSpec[Any, Nothing] = Spec(
//   TestCase(
//     "integer addition is associative",
//     <function1>,
//     Map(zio.test.TestAnnotation@4cf27564 -> List(SourceLocation(38-property-based-testing
//   )
// )
```

This definitely gives us some reason to believe that integer addition is associative,

but there is also something somewhat unsatisfying about it.

The associativity of integer addition is supposed to be true for any three possible integers, but we only picked one specific set of integers. Is it possible that there is something special about this particular combination, for example that the third number is the sum of the first two, that makes the test pass in this case even though the property is not always true?

We could provide some additional evidence that this property is true by adding additional tests with some other sets of values, perhaps trying to pick values with no obvious relationship to each other to increase our confidence that this property really is true in general. However, we are still probably only going to write five or ten of these tests at most, which is not very many.

In addition, each of these tests takes developer time, both in writing the tests as well as thinking about what these specific values are that we want to test and any particular properties we want to make sure that the values do or do not satisfy. Taking our usual "lazy" approach as developers we might wonder whether there is a way to automate this.

Property based testing does just that. In property based testing the test framework generates a large number of values and tests each of them, identifying either a counterexample to the assertion or reporting that no counterexample was found.

Here is what a property based test for the same assertion would look like with ZIO Test.

```
testM("integer addition is associative") {
  check(Gen.anyInt, Gen.anyInt, Gen.anyInt) { (x, y, z) =>
    val left = (x + y) + z
    val right = x + (y + z)
    assert(left)(equalTo(right))
  }
}
// res1: ZSpec[TestConfig with zio.random.package.Random, Nothing] = Spec(
//   TestCase(
//     "integer addition is associative",
//     <function1>,
//     Map(zio.test.TestAnnotation@4cf27564 -> List(SourceLocation(38-property-based-testing
//   )
// )
```

This test will generate a large number of combinations of `x`, `y` and `z` values and test that the assertion is true for all of them.

While property based testing is a great tool to have in our toolbox it is not better than traditional testing in all situations and in fact a combination of traditional and property based testing can often provide the highest level of test coverage.

The obvious advantage of property based testing is that it allows us to quickly test a large number of test cases, potentially revealing counterexamples that might not have been obvious.

However, there are some issues to watch out for when using property based testing.

The first is that property based testing is often not good at identifying highly specific counterexamples.

Property based tests typically only generate one hundred to two hundred test cases. In contrast, even a single `Int` can take on more than a billion different values.

If we are generating more complex data types the number of possibilities increases exponentially. So in most real world applications of property based testing we are only testing a very small portion of the sample space.

This is fine as long as counterexamples are relatively common in the sample space. However, if counterexamples require multiple generated values to take on very specific values then we may not generate an appropriate counterexample even though such a counterexample does exist.

A solution to this is to complement property based testing with traditional tests for particular degenerate cases identified by developers. Bug reports can be another fruitful source for these "edges cases".

A second issue is that our property based tests are only as good as the samples we generate.

We introduced property based testing as a way to avoid having to come up with specific test cases but we often need to spend as much time thinking about what generator of values we want to use. Of course, the benefit is that once we do so we can leverage this to check a large number of test cases.

A good generator should generate test cases that are specific enough to satisfy the conditions of the property we are testing.

In the example above, the property of integer associativity is supposed to hold for all integers so we could just use `anyInt`, which generates random integers between `Int.MinValue` and `Int.MaxValue`. However, some properties may only hold for values in a narrower domain, for example positive integers or integers that are not equal to zero.

A good generator should also be general enough to generate test cases covering the full range of values over which we expect the property to hold.

For example, a common mistake would be to test a form that validates user input with a generator of ASCII characters. This is probably very natural for many of us to do, but what happens if the user input is in Mandarin?

A third issue is that we need to identify properties that we expect to hold for the objects and operations in our domain.

Sometimes this may be very obvious, such as when the properties are already defined in domains such as mathematics. But often within our business domains properties may be less immediately obvious.

One helpful way to identify properties is to ask yourself how you would know whether you would expect an assertion for a particular test case to hold or not.

For example, if you are testing a sorting algorithm a simple assertion would be that `List(2, 1).sorted == List(1, 2)`. But why do you know that this assertion should be true?

On reflection you might conclude that part of the reason is that the two lists contain the same elements. Sorting is supposed to rearrange the elements of a collection but not add or remove elements from the collection.

You've just got a property! When thinking about what properties you expect to hold it is often helpful to start by stating them conceptually and then you can deal with translating them into code later.

Thinking a bit more, you might observe that this is a necessary but not a sufficient condition for the assertion being true because the resulting list also has to be in order. That's a second property!

Another trick you can use to develop properties for testing is to think about identities that you expect to be true. For example, if we are testing ZIO's `Chunk` data type then a simple property would be that creating a `Chunk` from a `List` and then converting it back to a `List` should return the original `List` unchanged.

You can also think about whether there is another operator that you know is correct that should give the same result as the operator you are testing. For example, if we are testing the `filter` method on `Chunk`, we could say that filtering a list of elements with a predicate should give the same result as filtering a chunk of the same elements with that predicate.

None of this is meant to dissuade you from using property based testing but merely to highlight the advantages and disadvantages so you can decide what mix of property based and traditional tests are right for your use case. One of the great things about ZIO Test is it makes it easy to mix and match property based and traditional tests.

With this introduction, let's focus on the anatomy of a property based test. In ZIO Test, a property based test always has three parts:

1. A `check` operator - This tells ZIO Test that we are performing a property based test and controls parameters of how the property based test is executed such as how many samples to check, whether to check samples in parallel, and whether the assertion we are checking involves effects or not.

2. One of more `Gen` values - These tell ZIO Test what values we want to check. You can think of each `Gen` as representing a distribution of potential values and each time we run a property based test we sample values from that distribution.
3. An `assertion` - This is the actual assertion that we are testing and is just like any traditional test we write in ZIO Test except that it takes the generated values as an input. If you have a traditional test you can convert it to a property based test by copying the test into the body of a `check` operator and replacing the hard coded test values with the generated values.

Let's see what that looks like in the example from the beginning of the chapter.

```
testM("integer addition is associative") {
  check(Gen.anyInt, Gen.anyInt, Gen.anyInt) { (x, y, z) =>
    val left = (x + y) + z
    val right = x + (y + z)
    assert(left)(equalTo(right))
  }
}
// res2: ZSpec[TestConfig with zio.random.package.Random, Nothing] = Spec(
//   TestCase(
//     "integer addition is associative",
//     <function1>,
//     Map(zio.test.TestAnnotation@4cf27564 -> List(SourceLocation(38-property-based-testing
//   )
// )
```

On the first line, we construct a property based test using the `testM` operator we have seen before. Property based tests will always use `testM` as opposed to `test` because checking a property based test involves effects.

The other thing to notice here is that other than always using `testM` we write property based tests with exactly the same syntax as we use for traditional tests, which makes it easy to mix traditional with property based tests or to refactor from one to another.

In the second line, we call the `check` operator, which tells ZIO Test that we want to perform a property based test. There are different variants of the `check` operator to control various aspects of how a property based test is run that we will learn about later in this chapter, but for now if you use `check` for assertions that don't involve effects and `checkM` for assertions that involve effects you will have what you need for most cases.

The `check` operator takes two sets of arguments. The first is the generators we want to use and the second is the assertion we want to test.

In the first set of arguments, each generator describes one test value we want to generate. For instance, here we want to generate three integers so we use three

generators.

Finally, in the second argument list to the `check` operator we get access to the values from each of the generators and make an assertion, which looks just like any normal test we would write.

Notice that the three lines in the body of the `check` operator here are identical to the corresponding lines in the traditional test from the beginning of this chapter. The only difference is that instead of defining `x`, `y`, and `z` as hard coded values in the line above we are now getting them from the `check` operator.

One implication of this is that most of the time we spend learning how to use a property based testing framework is learning how to construct the appropriate generators.

The first part of writing a property based test, the `check` operator, only has a few variants and generally we just pick one of them.

The third part of writing a property based test, the assertion we want to check, is identical to writing an assertion for a normal test. There is some additional thought for us to do to describe our expectations as properties instead of individual test cases but this is largely independent of the test framework itself.

Therefore, most of the material we will cover in the rest of this chapter will be on generators. We want to generate samples of values that are of interest to us in our business domain, and we want to do this in a composable way where we can build up generators for these objects from simple generators and control various parameters to get the distribution of values we want.

So from here we will dive into generators, understanding what a generator is, different kinds of generators, and operators for working with generators. We will also look at shrinking, which is a very useful feature where once the test framework identifies a counterexample it attempts to "shrink" that counterexample to find a simpler one that is easier for us to debug.

By the end of these materials you should have the ability to construct your own generator for any values that are of interest to you in your domain. Finally, in the last section we will come back to the `check` operator and look at how we can control different parameters for how property based tests are run.

Together, these materials will give you a comprehensive understanding of how you can apply property based testing to any scenario where it is helpful to you.

38.1 Generators As Streams Of Samples

So what is a generator? We know that a generator has the ability to generate values of a given type, but how is a generator actually represented?

In ZIO Test, a generator is a stream of samples:

```
import zio.stream._

final case class Gen[-R, +A](sample: ZStream[R, Nothing, Sample[R, A]])
```

A sample is a value along with a tree of potential "shrinkings" of that value.

```
final case class Sample[-R, +A](value: A, shrinks: ZStream[R, Nothing, Sample[R, A]])
```

Don't worry too much about the `shrinks` for now, we will spend more time on that when we talk about shrinking. For now, you can just think of a `Sample` as containing a value of type `A`.

So you can think of a `Gen` as a stream of test values.

Conceptualizing a generator as a stream has several benefits.

First, it lets us take advantage of all of the existing power of ZIO when generating values.

For example, we often need to use random number generation when generating values, either because we are generating random numbers directly or because we are using them to construct more complex data types.

Because we have access to ZIO's environment type, it is easy to represent this dependency on the capability of generating random numbers using the `Random` service we have seen before. For example, here is the type signature of the `anyInt` generator we saw above.

```
import zio.stream._

val integers: Gen[Random, Int] =
  Gen.anyInt
// integers: Gen[Random, Int] = Gen(zio.stream.ZStream$$anon$1@4865c270)
```

This makes it clear just looking at the type signature that this generator uses randomness as part of its process of generating random values. It also makes it easy for us to provide a random number generator that is deterministic so that we can always replicate a test failure.

One of the advantages of this is that it also makes it easy to see when a generator does not depend on random number generation or another capability. Here is a generator that always generates deterministic values.

```
val deterministic: Gen[Any, Boolean] =
  Gen.fromIterable(List(true, false))
// deterministic: Gen[Any, Boolean] = Gen(zio.stream.ZStream$$anon$1@402d7355)
```

This generator will always generate the values `true` and `false`.

In addition to having access to the environment type, we have all the other capabilities of ZIO such as safe resource handling. For example, we could imagine a generator that generates values by opening a local file with test data, reading

the contents of that file into memory, and each time generating a value based on one of the lines in that file.

We could use the support for safe resource handling built into `ZStream` to ensure that the file was opened before we generated the first value and closed when we were done generating values. Of course we would get all the other benefits we would expect such as the file being read incrementally and being automatically closed if the test was taking too long and we interrupted it.

38.2 Constructing Generators

Now that we know what generators are, our main question from a practical perspective is how we build generators for our own data types.

We've seen generators for some very simple data types like `Gen.anyInt` but where do we find generators for all the data types we work with from ZIO and the Scala standard library? And how do we create new generators for data types from other libraries or data types we have created ourselves?

To create generators for a data type we will generally follow a two step process.

First, construct generators for each part of the data type. If the parts are data types from ZIO or the Scala standard library you can use an existing generator, otherwise apply this same process again to each part.

Second, combine these generators using operators on `Gen` such as `flatMap`, `map`, and `oneOf` to build a generator for your data type out of these simpler generators.

To see how this works, let's imagine we have a simple domain model like this:

```
final case class Stock(ticker: String, price: Double, currency: Currency)

sealed trait Currency

case object USD extends Currency
case object EUR extends Currency
case object JPY extends Currency
```

We would like to construct a generator for `Stock` values to test some logic in our trading applications.

To do so, the first step is to break the problem down into smaller pieces. Just like with `ZIO`, `ZManaged`, or `ZStream` we can use `flatMap` to combine multiple generators into a single value.

```
lazy val genTicker: Gen[Any, String] =
  ???

lazy val genPrice: Gen[Any, Double] =
  ???
```

```
lazy val genCurrency: Gen[Any, Currency] =
  ???

lazy val genStock: Gen[Any, Stock] =
  for {
    ticker   <- genTicker
    price    <- genPrice
    currency <- genCurrency
  } yield Stock(ticker, price, currency)
```

At this point we still don't know how to generate a ticker, a price, or a currency but if we did know how to generate them we could combine them using the `flatMap` and `map` operators on `Gen`. There are a variety of other familiar operators on `Gen` like `zipWith` but the pattern shown above using for comprehensions is a very simple and readable one and you can apply it to building generators for any more complex data type that contains multiple other data types.

Our next step is to construct generators for each of these simpler data types.

Prices and tickers are represented as `String` and `Double` values, respectively, which are standard data types, so we should expect that ZIO Test will provide us with appropriate generators to construct these values and we just need to use the correct one. Generally, generators are named based on the type of values that they generate, with the `any` prefix used for generators that generate the full range of values for that type.

For `Ticker`, let's assume that the tickers will be represented as ASCII strings. In that case we can use the `Gen.anyASCIIString` constructor.

```
val genTicker: Gen[Random with Sized, String] =
  Gen.anyASCIIString
// genTicker: Gen[Random with Sized, String] = Gen(
//   zio.stream.ZStream$$anon$1@1ee593d5
// )
```

Notice that the environment type of the generator is now `Random with Sized`, indicating that the generator will use both random number generation to generate the values as well as the size parameter to control the size of the generated values.

Similarly, we can generate `Double` values using the `Gen.double` constructor. This time, let's assume that the generated prices should only have two decimal points of precision to see how we can use the `map` operator to customize generated values to suit our needs.

```
val genPrice: Gen[Random, Double] =
  Gen.double(0.01, 100000).map(n => math.round(n * 100) / 100)
// genPrice: Gen[Random, Double] = Gen(zio.stream.ZStream$$anon$1@145fda74)
```

Finally, generating the `Currency` values lets us see how to deal with combining generators for values that can be of one or more types. We saw above how we could use `flatMap` to construct generators for data types that include multiple other data types. We can use the `oneOf` constructor to construct generators for data types that may be one of several alternatives.

```
lazy val genUSD: Gen[Any, Currency] =
  ???

lazy val genJPY: Gen[Any, Currency] =
  ???

lazy val genEUR: Gen[Any, Currency] =
  ???

lazy val genCurrency: Gen[Random, Currency] =
  Gen.oneOf(genUSD, genJPY, genEUR)
```

The `oneOf` constructor takes a variable arguments list of generators and samples from each of the generators with equal probability, so in this case we will get `USD`, `EUR`, and `JPY` values one third of the time each. If we wanted to sample from the generators with different probabilities we could use the `weighted` constructor, which would allow us to specify a probability associated with sampling from each generator.

Finally, generating `USD`, `JPY`, and `EUR` values is very simple because they are just case objects so there is only one of each of them. For this we can use the `const` constructor for "constant", which is similar to the `succeed` operator on `ZIO`, `ZManaged`, and `ZStream`.

```
lazy val genUSD: Gen[Any, Currency] =
  Gen.const(USD)

lazy val GenJPY: Gen[Any, Currency] =
  Gen.const(JPY)

lazy val genEUR: Gen[Any, Currency] =
  Gen.const(EUR)
```

And there we have it. We now have a generator of `Stock` values that we can use in testing our application, and each step of constructing it was composable so if we ever want to change any of the logic it will be easy for us to do so.

38.3 Operators On Generators

Being a stream of samples, `Gen` supports many of the same operators we are already familiar with from `ZIO` and `ZStream`, though it is often helpful to

conceptualize how they apply to the domain of generators.

38.3.1 Transforming Generators

One of the most basic operators on generators is `map`, which we saw above. The `map` operator says "take every value generated by this generator and transform it with the specified function".

We can use `map` to reshape existing generators to fit the shape of the data we want to generate. For example, if we have a generator of integers from `1` to `100` we can transform it into a generator of even integers in the same range like this:

```
val ints: Gen[Random, Int] =
  Gen.int(1, 100)
// ints: Gen[Random, Int] = Gen(zio.stream.ZStream$$anon$1@13d44cd5)

val evens: Gen[Random, Int] =
  ints.map(n => if (n % 2 == 0) n else n + 1)
// evens: Gen[Random, Int] = Gen(zio.stream.ZStream$$anon$1@225f899b)
```

Every value produced by the original generator will be passed through the specified function before being generated by the new generator, so every `Int` generated by `evens` will be even.

This illustrates a helpful principle for working with generators, which is to prefer *transforming* generators instead of *filtering* generators.

We will see below that we can also filter the values produced by generators, but this has a cost because we have to "throw away" all of the generated data that doesn't satisfy our predicate. In most cases we can instead transform data that doesn't meet our criteria into data that does like in the example above.

This can make a significant difference to test performance. It also avoids the risk of us accidentally filtering out all generated values!

In addition to the `map` operator there is a `mapM` variant that allows transforming the result of a generator with an effectual function.

```
trait Gen[-R, +A] {
  def mapM[R1 <: R, B](f: A => ZIO[R1, Nothing, B]): Gen[R1, B]
}
```

Generally this operator is not used as often since we don't need to use additional effects to generate our test data beyond the ones our generators already perform, but you could imagine using this to log every value produced by a generator, for example.

One thing to notice here is that the error type of the effect must be `Nothing`. Generators do not have an error type because it does not really make sense for generators to fail.

If data is invalid it should just not be produced by the generator or if an error really does occur in the process of generating test data it should be treated as a fiber failure and handled by the test framework. This also keeps the error channel of a test available exclusively for errors from the code we are testing rather than the generator of test data.

38.3.2 Combining Generators

The next set of operators on generators allow us to combine two or more generators.

One of the most powerful of these is the `flatMap` operator which we saw above. Conceptually this lets us say "generate a value from one generator and then based on that generator pick a generator to generate a value from".

```
trait Gen[-R, +A] {
  def flatMap[R1 <: R, B](f: A => Gen[R1, B]): Gen[R1, B]
}
```

For example, if we had a generator of positive integers and a generator of lists of a specified size we could use `flatMap` to create a new generator of lists with a size distribution based on the first generator.

```
def listOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, List[A]] =
  ???

val smallInts: Gen[Random, Int] =
  Gen.int(1, 10)
// smallInts: Gen[Random, Int] = Gen(zio.stream.ZStream$$anon$1@2765aa24)

def smallLists[R <: Random, A](gen: Gen[R, A]): Gen[R, List[A]] =
  smallInts.flatMap(n => listOfN(n)(gen))
```

The composed generator will first generate a value from the first generator and then use that value to generate a value from the second generator. So if the first value produced by the `smallInts` generator is `2` `smallLists` will produce a list with two elements, and if the next value produced by `smallInts` is `r` the next list produced by `smallLists` will have five elements.

The `flatMap` operator is very helpful for chaining generators together. For example if we wanted to generate a custom distribution of durations we might first want to generate a `Boolean` value to determine whether the duration we generate should be a "short" or a "long" one and then generate a value from the appropriate distribution based on that.

In addition to the `flatMap` operator there are also the `cross`, and `crossWith` operators and their symbolic alias `<*>`. These let us sample values from two generators and either combine them into a tuple or with a function.

Conceptually a generator as a stream of samples, these methods return the Cartesian product of all possible pairs of values from the two generators.

For example, if we wanted to generate a pair of integers we could do it like this:

```
val pairs: Gen[Random, (Int, Int)] =
  Gen.anyInt <*> Gen.anyInt
// pairs: Gen[Random, (Int, Int)] = Gen(zio.stream.ZStream$$anon$1@4114f018)
```

We can easily implement `crossWith` in terms of `map` and `flatMap` as in all other cases:

```
def crossWith[R, A, B, C](left: Gen[R, A], right: Gen[R, B])(f: (A, B) => C): Gen[R, C] =
  left.flatMap(a => right.map(b => f(a, b)))
```

And in fact as discussed above we will often use the convenient `for` comprehension syntax even when generators do not actually depend on each other.

```
val pairs2 = for {
  x <- Gen.anyInt
  y <- Gen.anyInt
} yield (x, y)
// pairs2: Gen[Random, (Int, Int)] = Gen(zio.stream.ZStream$$anon$1@7da6f2a3)
```

In addition to `cross` and `crossWith` there are also analogs to the `foreach` and `collectAll` operators we have seen from `ZIO` for combining collections of values. In the case of the `Gen` data type we are more concerned with constructing collections of particular types, typically from a single generator, so these have more specialized signatures.

```
def chunkOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, Chunk[A]] =
  ???

def listOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, List[A]] =
  ???

def mapOfN[R, A, B](n: Int)(key: Gen[R, A], value: Gen[R, B]): Gen[R, Map[A, B]] =
  ???

def setOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, Set[A]] =
  ???
```

There are also convenience methods without the `N` suffix such as `listOf` that generate collections with a size range determined by the testing framework and variants with the `1` suffix such as `listOf1` that generate non-empty collections with a size range determined by the testing framework.

As an exercise, try implementing the `listOfN` operator yourself in terms of the operators we have seen so far. Why might the `setOfN` and `mapOfN` operators present some particular challenges to implement correctly?

One potential inefficiency you may have noticed in some of the examples above is that the `flatMap` operator requires us to run our generators sequentially, because the second generator we use can depend on the value generated by the first generator. However, in many of the cases we have seen such as the generating the `ticker`, `price`, and `currency` for a `Stock` the generated values were actually independent of each other and could have been generated in parallel.

ZIO Test supports this through the `zipWith` and `zip` operators and their symbolic alias `<&>`. These will generate the two values in parallel and then combine them into a tuple or using the specified function.

Thinking again about generators as streams of samples, these operators "zip" to streams of samples together by repeatedly pulling from each stream pairwise.

Using these operators can improve the performance of generators in some cases, especially when generating large case classes, but in general it is fine to use a for comprehension or the sequential operators unless testing time for property based tests is a particular issue for you.

38.3.3 Choosing Generators

In the section above we looked at ways of combining generators that conceptually pulled values from two or more generators and combined them somehow to produce new generated values. This pattern of sampling from *both* generators is one of the fundamental ways of combining generators and corresponds to generating data for *sum types* that have values of two or more types inside them.

For example, the `Stock` data type above was a sum type that had values of three different types inside it, a `String`, a `Double`, and a `Ticker`. Using a `for` comprehension or one of the `zip` variants such as `zipN`, which allows combining more than two generates with a function, is a very natural solution for generating values for data types like that.

In contrast, the second fundamental way of combining generators is by pulling values from *either* one generator or another instead of *both* generators. If `zip` is the most basic operator for sampling from both generators, `either` is the most basic operator for sampling from either generator.

```
def either[R, A, B](left: Gen[R, A], right: Gen[R, B]): Gen[R, Either[A, B]] =
  ???
```

The `either` operator is helpful for when we want to generate data for sum types that can be one type or another, such as the `Currency` data type above.

For example, say we want to generate samples of `Try` values from the Scala standard library. A `Try` value may be either a `Success` with some value or a `Failure` with some `Throwable`. We can sample from both distributions using `either` and then use `map` to combine them into a common data type.

```
import scala.util.{Failure, Success, Try}

def genTry[R <: Random, A](gen: Gen[R, A]): Gen[R, Try[A]] =
  Gen.either(Gen.throwable, gen).map {
    case Left(e) => Failure(e)
    case Right(a) => Success(a)
  }
```

In addition to the `either` operator there are operators to construct other common sum types such as `option` and a couple of helpful convenience methods for combining more than two alternatives.

The first is the `oneOf` operator, which picks from one of the specified generators with equal probability. We saw this above in our implementation of the generator for `Currency` values.

```
def oneOf[R <: Random, A](gens: Gen[R, A]*): Gen[R, A] =
  ???
```

The second is the `elements` operator, which is like `oneOf` but just samples from one of a collection of concrete values instead of from one of a collection of generators. For example, we could simplify our implementation of the `Currency` generator using `elements` like this:

```
sealed trait Currency

case object USD extends Currency
case object EUR extends Currency
case object JPY extends Currency

val genCurrency: Gen[Random, Currency] =
  Gen.elements(JPY, USD, EUR)
// genCurrency: Gen[Random, Currency] = Gen(
//   zio.stream.ZStream$$anon$1@63efacd7
// )
```

One thing you may notice is that the implementation of `either` to a certain extent makes a choice for us in that it samples from each of the `left` and `right` generators with equal probability. This is a sensible default but not necessarily what we want.

If we want to sample from multiple generators with custom probabilities we can use the `weighted` operator, which allows us to specify a collection of generators and weights associated with them.

```
def weighted[R <: Random, A](gs: (Gen[R, A], Double)*): Gen[R, A] =
  ???
```

For example, we could create a generator that generates `true` values 90% of the time and `false` values 10% of the time like this:

```
val trueFalse: Gen[Random, Boolean] =
  Gen.weighted(Gen.const(true) -> 9, Gen.const(false) -> 1)
// trueFalse: Gen[Random, Boolean] = Gen(zio.stream.ZStream$$anon$1@730aae7f)
```

For more complex cases you can use the `flatMap` operator described above to create your own logic where you first generate a probability distribution of cases and then generate a value for each case.

As an exercise, try implementing the `genTryWeighted` constructor yourself without using `weighted` in terms of `flatMap` and existing constructors.

```
val genFailure: Gen[Random, Try[Nothing]] =
  Gen.throwable.map(e => Failure(e))
// genFailure: Gen[Random, Try[Nothing]] = Gen(
//   zio.stream.ZStream$$anon$1@5158c97b
// )

def genSuccess[R, A](gen: Gen[R, A]): Gen[R, Try[A]] =
  gen.map(a => Success(a))

def genTryWeighted[R <: Random, A](gen: Gen[R, A]): Gen[R, Try[A]] =
  Gen.weighted(genFailure -> 0.1, genSuccess(gen) -> 0.9)
```

38.3.4 Filtering Generators

In addition to combining and choosing between generators we can also filter generators.

```
trait Gen[-R, +A] {
  def collect[B](pf: PartialFunction[A, B]): Gen[R, B]
  def filter(f: A => Boolean): Gen[R, A]
  def filterNot(f: A => Boolean): Gen[R, A]
}
```

If we think of a generator as a bag of potential values that we pull a value out of each time we sample filtering corresponds to throwing away some values and picking another one until we find an acceptable value.

As mentioned above, we want to be careful about filtering our generators because filtering can negatively impact test performance. Every time we filter out a value the test framework has to generate another one to replace, potentially repeatedly if the newly generated value also does not satisfy our predicate.

If we are only filtering a small number of generated values then filtering can be fine and can provide a simple way to remove certain degenerate values while maintaining the same distribution of other values. However, if we are filtering out a large number of values, such as half of all values in the example above regarding even numbers, then testing time can be significantly impacted.

This is especially true if we are applying multiple filters to generated values. In addition, there is the risk that we may inadvertently filter out all values in a generator, resulting in test timeouts.

For these reasons, best practice is where possible to transform generated values by using operators like `map` to turn invalid values into valid values, as in the example with `evens` above, rather than filtering values.

38.3.5 Running Generators

The final set of operators on generators allow us to run generators ourselves.

Most of the time we don't need to worry about running generators ourselves because the test framework takes care of that for us. We just provide our generator to the `check` operator and the test framework takes care of using the generator to generate a large number of values, testing the assertion with those values, and reporting the results.

However, sometimes it can be useful for us to be able to generate values from a generator directly without using the test framework. The most common use case for this is during development when we want to get a sense of the values produced by a generator and make sure they conform to our expectations.

A generator is just a stream of samples, so we can always call the `sample` operator on a `Gen` to get access to the underlying `ZStream` and run the `ZStream` using any of the usual operators for running streams, but there are also several convenience methods on `Gen` to save us a few steps.

The most useful of these is `runCollectN`, which repeatedly runs the generator and collects the specified number of samples.

```
trait Gen[-R, +A] {
  def runCollectN(n: Int): ZIO[R, Nothing, List[A]]
}
```

Using this you can quickly get a sense of the distribution of values produced by a generator by doing something like:

```
import zio.console._

def debug[R <: Console, A](gen: Gen[R, A]): ZIO[R, Nothing, Unit] =
  gen.runCollectN(100).flatMap(as => ZIO.foreach_(as)(n => console.putStrLn(n.toString)))
```

38.4 Random And Deterministic Generators

In addition to being able to take advantage of the power of ZIO, one of the advantages of representing a `Gen` as a `ZStream` of values is that we can unify random and deterministic property based testing.

Traditionally in property based testing there has been a distinction between *random* and *deterministic* property based testing.

In random property based testing, values are generated using a pseudorandom number generator based on some initial seed. For example, the first three values we get from a random generator of small random integers might be `1`, `-4` and `8`.

The advantage of random property based testing is that it is relatively easy to generate values from the full distribution of the sample even if the sample space is very large. Even though there are more than a billion integers we can construct a generator of random integers that is equally likely to generate any of them, allowing us to generate a collection of test cases that include positive integers, negative integers, large integers, small integers, and so on.

For this reason, random property based testing is the most popular form of property based testing in Scala and is used by previous testing libraries such as ScalaCheck and its predecessor QuickCheck in Haskell.

The disadvantage of property based testing is that it is impossible for us to ever *prove* a property with random property based testing, we can merely fail to falsify it.

For example, say we want to test the property that logical conjunction is associative.

```
import zio.test.Assertion._

check(Gen.boolean, Gen.boolean, Gen.boolean) { (x, y, z) =>
  val left = (x && y) && z
  val right = x && (y && z)
  assert(left)(equalTo(right))
}
// res6: URIO[TestConfig with Random, TestResult] = zio.ZIO$FlatMap@2f427b2c
```

Using random property based testing we will generate a large number of combinations of values of `x`, `y`, and `z` and verify that the property holds for all of them. This is all well and good but we can actually make a stronger claim than this and don't have to test one hundred to two hundred combinations of values.

Each value can only be `true` or `false` so there are only eight combinations of possible values. We can check them all, which in this case will be more efficient than checking a hundred or more different values.

Furthermore, we can conclude that we have proved this property since we have verified it for every possible input rather than merely providing some evidence for believing it because we failed to falsify it for a sample of test cases.

Deterministic property based testing builds on this idea by generating test cases in a predefined order, typically based on some concept of "smallest" to "largest". For instance, the first three values from a deterministic generator of integers

would typically be `0`, `1`, and `-1`, conceptualizing `0` as the "smallest" value and larger values as moving away from it in both directions on the number line.

The advantage of deterministic property based testing is that it allows us to know that we have explored all possible test cases up to some "size", and possibly all test cases that could exist if the domain of possible values is small enough.

The disadvantage of deterministic property based testing is that for more complex data types the domain of possible values can grow so quickly that exhaustively testing starting with smaller samples fails to test many of the more complex cases we are interested in within a reasonable timeframe.

Nevertheless, there are libraries such as SmallCheck that support this kind of property based testing and for domains that are small enough it can be extremely useful both to avoid repeating the same test cases as well as to prove properties.

Historically these two types of generators have been represented differently, with random generators typically being represented as some "effect" type with the capability of random number generation and deterministic generators being represented as some lazy sequence of values. These have existed in separate libraries so users have typically had to select one approach or the other, with most users in Scala opting for random property based testing.

Notice that we said above that random generators were represented as an effect type whereas deterministic generators were represented as a lazy sequence of values. By representing a generator as a stream of values we can unify these two approaches because a `ZStream` can model both effects as well as zero or more values.

Thus, we can represent a random generator as:

```
import zio.random._
import zio.stream._

val randomGenerator: Gen[Random, Int] =
  Gen(ZStream.fromEffect(nextInt).map(Sample.noShrink))
// randomGenerator: Gen[Random, Int] = Gen(zio.stream.ZStream$$anon$1@6d3331e8)
```

In other words, a random generator is just a stream with a single value. That value is an effect that, each time it is evaluated, will produce a new random number.

Don't worry about the use of `Sample.noShrink` here. We will talk about shrinking in the next section but right now just think of it as constructing a sample with a value but no shrinking logic to satisfy the type signature required by `Gen`.

We can represent a deterministic generator as:

```
val deterministicGenerator: Gen[Any, Boolean] =
  Gen(ZStream(true, false).map(Sample.noShrink))
```

```
// deterministicGenerator: Gen[Any, Boolean] = Gen(
//   zio.stream.ZStream$$anon$1@724da362
// )
```

A deterministic generator is a stream with zero or more values where those values typically do not involve effects and represent the full domain of the generator. In this case the stream has two values representing the two possible `Boolean` values of `true` and `false`.

Notice that in the snippet above the type signature makes clear that this is a deterministic generator because it does not include `Random` in the environment type.

When the test framework runs a generator when evaluating the `check` operator, it internally calls `forever.take(n)` to get an appropriate number of samples from the generator. If your generator is deterministic, you can instead use the `checkAll` operator, which will just sample the full domain of the generator.

Make sure that the generator you are using is finite and is small enough for all of its values to be evaluated in a reasonable timeframe before calling the `checkAll` operator!

In ZIO Test, almost all of the constructors create random generators, as this is generally the best default outside of specific cases where the sample space is small.

The main exception and the starting point if we want to do deterministic property based testing with finite generators is the `fromIterable` constructor. The slightly simplified signature of `fromIterable` is:

```
def fromIterable[A](as: Iterable[A]): Gen[Any, A] =
  ???
```

The type signature here is similar to the `elements` constructor we saw above but there are important differences.

```
def elements[A](as: A*): Gen[Random, A] =
  ???
```

Both of these constructors take a collection of `A` values so initially they might seem quite similar. But notice that the environment type of `elements` is `Random` whereas the environment type of `fromIterable` is `Any`.

This indicates that `elements` constructs a random, infinite generator whereas `fromIterable` constructs a deterministic, finite generator.

Internally `elements` is represented as a single element effectual stream, where that single element is an effect that each time it is evaluated will randomly pick one of the elements in the initial collection. In contrast, the generator returns from `fromIterable` is a two element stream that just contains the values `true` and `false`.

Most of the time you don't have to worry about this and if you aren't thinking about it you are probably using random, infinite generators and the operators on `Gen` will just automatically do the right thing for you. But let's explore how we can use the tools that ZIO Test gives us to do deterministic property based testing with finite generators.

As an example, we will show how we can more efficiently test the property discussed above that logical conjunction is associative and actually *prove* the property instead of merely failing to falsify it.

To do so, we will start by constructing a finite, deterministic, generator of `Boolean` values using the `fromIterable` constructor.

```
val booleans: Gen[Any, Boolean] =
  Gen.fromIterable(List(true, false))
// booleans: Gen[Any, Boolean] = Gen(zio.stream.ZStream$$anon$1@74d98ff8)
```

Now we can test all possible combinations of values by replacing the `check` operator with the `checkAll` operator.

```
checkAll(booleans, booleans, booleans) { (x, y, z) =>
  val left = (x && y) && z
  val right = x && (y && z)
  assert(left)(equalTo(right))
}
// res7: URIO[TestConfig, TestResult] = zio.ZIO$FlatMap@76117b1
```

Now ZIO Test will generate all possible combinations of `x`, `y`, and `z` values, testing the assertion eight times in total, allowing us to both test the property more efficiently as well as know that we have proved it instead of merely failing to have falsified it.

This may seem somewhat magical. All we did was replace `Gen.booleans` with our deterministic `booleans` generator and used `checkAll` instead of `check` and we got to use an entirely different property based testing paradigm!

To unpack this a little more, let's look at how ZIO Test did this. How did it know that there were only eight possible values and generate samples for all of them.

To answer this, let's break down how ZIO Test generated these values.

When we supply more than one generator to a `check` method or one of its variants, ZIO Test combines them all with `cross`, which in turn is implemented in terms of `flatMap`, to generate the product of all possible combinations of these values. So the values that are generated above correspond to the following generator:

```
val triples: Gen[Any, (Boolean, Boolean, Boolean)] =
  for {
    x <- booleans
```

```
    y <- booleans
    z <- booleans
  } yield (x, y, z)
// triples: Gen[Any, (Boolean, Boolean, Boolean)] = Gen(
//   zio.stream.ZStream$$anon$1@70c0e883
// )
```

Thinking about generators again as streams of samples, each invocation of `booleans` corresponds to a stream with two elements. So just like if we had a `List` with two elements, using `flatMap` returns a new stream with all possible combinations of values from the original stream.

This is where the conceptualization of generators as streams becomes so powerful.

When we have a finite stream `flatMap` and operators derived from it return a new stream with all possible combinations of values from the original stream. On the other hand, when we have a single element effectual stream `flatMap` just returns a new single element effectual stream that runs both effects.

So we get the right behavior for both random, infinite generators as well as deterministic, finite generators!

When working with finite generators we do have to think a little more about the semantics of some operators and that conceptualization of generators as streams of samples.

For example, for infinite random generators we said the difference between `cross` and `zip` was that `cross` would run the left generator before the right generator whereas `zip` would run them in parallel, so which one we used was mostly just an optimization. In contrast, for finite, deterministic streams `cross` and `zip` have quite different semantics.

The `cross` operator corresponds to the Cartesian product of all possible combinations of two streams. So when we do `cross` with `booleans` and `booleans` the generated values will be `(true, true)`, `(true, false)`, `(false, true)`, and `(false, false)`.

In contrast, the `zip` operator corresponds to the pairwise product of two streams. So when we do `zip` with `booleans` and `booleans` the generated values will be `(true, true)` and `(false, false)`.

Both of these behaviors can be useful. In our example the `cross` behavior was what we wanted and that is the default way generators are combined, for example in `check` and the implementation of other operators.

But the `zip` behavior can be useful as well if we want to check for example that pairs of order data and corresponding customer data yield expected results when processed by our application.

The point is just that if you are doing deterministic property based testing with finite samples think a bit more about the operators you are using and make sure

they reflect your intended semantics.

38.5 Samples And Shrinking

The final major aspect of property based testing we will cover here is *shrinking*.

Typically when we run property based tests the values will be generated randomly and so when we find a counterexample to a property it will typically not be the "simplest" counterexample.

For example, if we are testing a queue implementation a property might be that offering a collection of values to the queue and then taking them yields the same value back. If there is a bug in our queue implementation the initial test failure may involve a relatively large test case, say a collection of ten numbers of varying sizes and one expected number is missing from the output.

In this situation it can be difficult for us to diagnose the problem.

Is there something about that number of elements that breaks our implementation? Or the fact that one element is a particular value?

To help us it is useful if the test framework tries to *shrink* failures to ones that are "simpler" in some sense and still violate the property. In the example above, if the test framework was able to shrink the counterexample to offering the single element `0` to the queue then that would be very helpful and could indicate that we have some kind of simple bug like an off by one error.

When we shrink we want to do two things.

First, we want to generate values that are "smaller" than the original value in some sense. This could mean closer to zero in terms of numbers or closer to zero size in terms of collections.

Second, we want to make sure that the "smaller" values we shrink to also satisfy the conditions of the original generator. It doesn't do any good to report a minimized counterexample that isn't actually a counterexample!

Shrinking is one of the strong points of ZIO Test because shrinking is *integrated* with value generation.

In some other frameworks shrinking is handled separately from value generation, so there is for example an implicit `Shrink[Int]` that knows how to shrink integers towards smaller values. The problem with this is that the `Shrink` instance knows how to shrink integers in general but not how to shrink the particular integer values that are being generated.

So if the values that are being generated have to obey particular constraints, they have to be even integers for example, the shrinker can shrink to values that no longer satisfy that constraint, resulting in useless counterexamples that aren't actually counterexamples.

Instead of doing this ZIO Test uses a technique called *integrated shrinking* where every generator already knows how to shrink itself and all operators on generators also appropriately combine the shrinking logic of the original generators. So a generator of even integers can't possibly shrink to anything other than an even integer because it is built that way.

To see how this works we have to look at ZIO Test's `Sample` data type.

We said before that a generator is a stream of samples.

```
final case class Gen[-R, +A](sample: ZStream[R, Nothing, Sample[R, A]])
```

But so far we haven't said anything about what a `Sample` is other than that conceptually it is a value along with a tree of potential shrinkings. We're now at a point where we can say more about what a sample is.

Here is what the `Sample` data type looks like:

```
final case class Sample[-R, +A](value: A, shrink: ZStream[R, Nothing, Sample[R, A]])
```

A `Sample` always contains a `value` of type `A`. That's the original value that we generate when we call the `check` method in our property based tests and if we didn't care about shrinking we wouldn't need anything but this value and could simplify the representation of `Gen` to `ZStream[R, Nothing, A]`.

In addition to a `value` a `Sample` also contains a "tree" of possible "shrinkings" of that value. It may not be obvious from the signature but `ZStream[R, Nothing, Sample[A]]` represents a tree.

The root of the tree is the original `value`. The next level of the tree consists of all the values for the samples in the `shrink` collection.

Each of these values in turn may have its own children, represented by its own shrink tree. And this can occur recursively, potentially forever.

The other thing to notice is that each level of the tree is represented as a stream. This is important because a stream can be lazily evaluated.

In general even for relatively simple generated values the entire shrink tree can be too large to fit in memory so it is important that we evaluate the shrink tree lazily, only exploring parts of it as we search for the optimal shrink.

The shrink tree must obey the following invariants.

First, within any given level, values to the "left", that is earlier in the stream, must be "smaller" than values that are later in the stream.

Second, all children of a value in the tree must be "smaller" than their parents.

These properties allow us to implement an efficient search strategy using the shrink tree.

We begin by generating the first `Sample`` in theshrinkstream and testing whether itsvalue‘ is also a counterexample to the property being tested.

If it is a valid counterexample, we recurse on that sample. If it is not, we repeat the process with the next `Sample` in the original `shrink` stream.

It is important for all branches of the shrink tree to continue to maintain the possibility to shrink to the smallest value, since when we combine generators a shrink value for one generator may not be a valid counterexample because another generated value has taken on a certain value, or there may still be a smaller counterexample even though this was not a valid counterexample.

For example, the default shrinking logic for integral values first tries to shrink to zero, then to half the distance between the value and zero, then to half that distance, and so on. At each level we repeat the same logic.

This is not guaranteed to generate the "optimal" shrink but in general results in quite good shrinkings in a reasonable time. Shrunk values may not be as close to optimal for very complex data types, such as large nested case class hierarchies, because the space to explore is so much larger and the test framework only explores a certain number of potential shrinks to avoid spending too much time on shrinking.

The other part of shrinking is maintaining the shrink tree when we compose generators. `Sample` itself has its own operators such as `map` and `flatMap` for combining `Sample` values.

The `map` operator conceptually transforms both the `value` of the sample as well as all of its potential shrinkings with the specified function. When we call `map` on a `Gen` value the implementation in turn calls `map` on the `Sample` value, which is why when we use `map` to transform a generator to generate only even integers we are guaranteed that the shrinkings will also contain only even integers.

Similarly, the `flatMap` operator on `Sample` allows constructing a new `Sample` based on the value from the original one. It corresponds to taking the value at each node in the shrink tree and generating a new shrink tree rooted at that node.

You can see how with composed generators the entire shrink tree can quickly become quite large, which is why it is important to traverse it lazily.

Hopefully this gives you a good overview of how shrinking works in ZIO Test, but as a user you don't have to know much about shrinking to enjoy its benefits.

All constructors of generators in ZIO Test already build in shrinking logic and all operators preserve that shrinking logic, so unless you are implementing your own primitive generators you shouldn't have to think about shrinking much other than just enjoying its benefits.

The main area as a user where you may want to think about shrinking is there are a couple of operators that allow you to control the shrinking logic.

The most useful of these is the `noShrink` operator.

```
trait Gen[-R, +A] {
  def noShrink: Gen[R, A]
}
```

The `noShrink` operator just removes all shrinking logic from a generator.

Why would you want to do this, you might ask, if we just said that shrinking can be so useful?

Shrinking does take additional time, and sometimes the shrunk counterexample may not be particularly useful, or you may have gleaned what you can from it. If you are debugging you may just care at first whether the test passes and not want to wait for the test framework to shrink it, in which case you can use the `noShrink` operator.

Another operator that can sometimes be useful if you are implementing your own generator is `reshrink`.

```
trait Gen[-R, +A] {
  def reshrink[R1 <: R, B](f: A => Sample[R1, B]): Gen[R1, B]
}
```

The `reshrink` operator allows you to throw away the existing shrinking logic associated with a generator and replace it with new shrinking logic by mapping each value to a new `Sample` with its own shrinking logic.

This can be useful when the process to shrink a value is much simpler than the process to generate the value.

For example, you might create a generator that produces values between `0.0` and `1.0` using a complicated formula based on multiple other generated values. By default, ZIO Test's integrated shrinking would try to shrink that value by shrinking each of the inputs to that formula.

However, if the generator can produce any value between `0.0` and `1.0` then we don't need to do that. We can just shrink straight toward zero.

The `Sample` companion object contains several useful shrinking strategies that make it easy for you to do this.

```
def noShrink[A](a: A): Sample[Any, A] =
  ???

def shrinkFractional[A](smallest: A)(a: A)(implicit F: Fractional[A]): Sample[Any, A] =
  ???

def shrinkIntegral[A](smallest: A)(a: A)(implicit I: Integral[A]): Sample[Any, A] =
  ???
```

The `noShrink` operator applies no shrinking logic at all and so produces a `Sample` with an empty shrink tree. The `noShrink` operator on `Gen` we saw above can be implemented simply as `reshink(Sample.noShrink)`.

The `shrinkFractional` and `shrinkIntegral` shrink numeric values towards the specified "smallest" value and are the same ones used internally in ZIO Test that ultimately power most shrinking logic because most generators are ultimately derived from random number generators.

So in the example above you could shrink your generated values toward `0.0` using the following:

```
lazy val myGeneratorOfComplexDistributions: Gen[Random, Double] =
  ???

lazy val myGeneratorThatShrinksTowardZero: Gen[Random, Double] =
  myGeneratorOfComplexDistributions.reshrink(Sample.shrinkFractional(0.0))
```

In this way ZIO Test tries to make it very easy for you to manipulate shrinking logic when you need to without having to get into the nitty gritty of how shrinking works.

38.6 Conclusion

Hopefully the material in this chapter has given you a thorough understanding of property based testing and how you can use it to take your testing to the next level.

If used correctly property based testing is a great way to increase the quality of your code and catch bugs. And with ZIO Test it is easy to integrate property based testing with your existing testing style, even if that is just adding a single property based test.

38.7 Exercises

Chapter 39

Testing: Test Annotations

Chapter 40

Testing: Reporting

Chapter 41

Applications: Parallel Web Crawler

In this chapter and the remaining chapters of this book we will look at how we can use ZIO and libraries in the ZIO ecosystem to solve specific problems.

This has several benefits.

First, it will give you practice in integrating the material you have learned so far.

In previous sections of this book we focused on one feature at a time. While some of these features built on each other, we were necessarily focused on the new functionality introduced in each chapter.

However, solving most real world problems requires multiple tools. We don't just need a `Ref`, for example, but may also have to use multiple other concurrent data structures such as `Promise` and `Queue`, along with understanding ZIO's fiber based concurrency model, safe resource usage, and dependency injection.

Working through these applications will give you concrete experience in pulling together everything you have learned so you are prepared to do the same thing at your job or in your own personal projects.

Second and related to this, it will help you develop your ability to identify the right tool to solve a particular problem.

So far it has been fairly obvious which tool we should employ to solve a particular problem. We are reading the chapter on `STM` so clearly the solution is going to involve software transactional memory, and often we explicitly introduced a piece of functionality as a solution to a certain class of motiving problems.

In contrast, in our day to day programming work it is often much less obvious what the correct tool is for the job.

Do we need software transactional memory or can we use regular concurrent data structures? Do we need to work directly with fibers or can we utilize existing operators?

We have tried to provide some guidelines for answering specific questions like this throughout the book but it can be more challenging when we have to determine the appropriate tools for the job from scratch.

As we work through these applications we will try to walk through our thought process of determining the right tools to use for each problem so you can build this skill yourself. In the process we will often work iteratively towards a solution so you will see how often there is not one "obvious" right answer but an ongoing process of refining a solution.

Finally, these applications can service as blueprints for you in tackling certain types of problems.

We will walk through solutions to problems in a variety of domains, from a parallel web crawler to file processing, a command line interface, Kafka, GRPC microservices, a REST API, GraphQL, and working with Spark.

While your problem probably won't look exactly like one of these examples, in many cases it will involve one or more of these elements. In those cases you can use the content in these chapters to get a head start on issues particular to your domain and potentially even use the code as inspiration for your own solution.

With that introduction, let's dive into our first application, building a parallel web crawler.

41.1 Definition Of A Parallel Web Crawler

Our application for this chapter is going to be building a parallel web crawler. A web crawler is used by search engines to build a view of all the web pages in a particular domain or potentially even on the entire internet, facilitating rapidly serving these pages or performing further analysis such as Google's PageRank algorithm.

Conceptually, a web crawler proceeds by starting with a set of seed URLs, downloading the content for each of those pages and extracting any links on those pages, and then repeating the process with each of the links found. As the web crawler does this it builds a web of sites that are conceptually farther and farther from the initial seeds, so this process is also called *spidering*.

In the course of doing this, there are several things we need to be careful of.

First, we need to be sure that we do not repeatedly crawl a website that we have already visited. In longer running web crawlers we might want to visit a site again after a certain duration, but for our purposes we will say that we do not want to crawl a website that we have already visited.

Second, we want to allow some way to determine whether we should crawl a particular site.

We might only want to explore a certain domain or set of domains, for example creating an index of websites hosted by a particular educational institution. Or we might not want to crawl certain pages, for example pages indicating that they should not be indexed by search engines.

Finally, we have to answer the question of what we actually want to do with each site we crawl. Should we write it to a database, store it in memory, or send it somewhere else as part of some larger data processing pipeline we are developing?

In light of these issues, we will work towards the following interface for a web crawler:

```
import zio._

import java.net.URL

type Web = ???

def crawl[R, E](
  seeds: Set[URL],
  router: URL => Boolean,
  processor: (URL, String) => ZIO[R, E, Unit]
): ZIO[R with Web, Nothing, List[E]] =
  ???
```

In this conceptualization, `crawl` takes three parameters:

- `seeds` - The initial set of sites to start the process from. For example, we might start with the New York Times home page, available at https://www.nytimes.com, and recursively explore all sites linked to from there, all sites linked to from those sites, and so on.
- `processor` - A function from a `URL` and a `String` representing the HTML content located at that `URL` to a `ZIO[R, E, Unit]`. Since the return type of the `ZIO` effect is `Unit`, we know that for the processor to do something it must perform some observable effect other than its return value, for example writing the `URL` and its contents to a database.

There are a couple of other design choices that are implied by this type signature. In particular, the return type of `crawl` is `ZIO[R with Web, Nothing, List[E]]`.

We will come back to `Web` shortly but first notice the error and value types.

The error type of the effect returned by `crawl` is `Nothing`, which indicates that even if the `processor` effect fails, the overall effect returned by `crawl` must still succeed and somehow handle that error internally.

We get a better sense of how `crawl` will do that when we look at the value type, which is `List[E]`. So if there are any failures in the `processor` effect, `crawl` will still continue executing and simply accumulate those errors, returning a list of all the errors that occured, if any, when `crawl` completes.

The other implication of this is that the return value of `crawl` will not actually include the results of the web crawler. This means that the only way the results will be used is in the `processor` function.

This is fine because we could always have `processor` update a data structure like a `Ref` with the results of the web crawl, and this avoids us needing to retain the content of each page crawled in memory if we don't need to. We might just write the contents of each web page to a database or offer it to a `Queue` that will be consumed by some downstream process, for example.

One other constraint we will impose is that the web crawler must be *concurrent*, that is, it must support multiple fibers exploring different sites at the same time. There is a significant delay between requesting a page and receiving its contents relative to CPU bound operations, so for efficiency we would like to be able to fetch multiple pages at the same time and this will require us to take advantage of more of the features that ZIO has to offer.

41.2 Interacting With Web Data

Let's come back to the `Web` service in the signature of `crawl` above. So far we haven't defined what this is, but we can reason about it as follows.

Clearly, the web crawler is going to need some way to actually access the web and retrieve the content located at different URLs.

However, we would also like to be able to support some test version of this functionality where accessing certain URLs retrieves prepopulated content that we create and is available locally. This way we can test the logic of our web crawler in a deterministic way that doesn't depend on our internet connection or the contents of a particular website.

The logic of retrieving a web page is also relatively independent of the logic of the web crawler itself. The web crawler needs some way to say "give me the HTML content associated with this URL" but the web crawler doesn't really care how this happens as long as it gets the HTML back.

These two conditions are a great sign that we should make this functionality part of a service in the environment:

1. Some functionality is necessary for our business logic but we don't need to know how that functionality is implemented
2. We want to provide alternative implementations

So just from reasoning about it we are able to say that we want some `Web` service

that conceptually allows us to say "give me the HTML associated with this URL" and gives us that HTML back.

Let's formalize this in code as:

```
def getURL(url: URL): ZIO[Any, Throwable, String] =
  ???
```

The `getURL` method takes a `URL` that we want to get as an argument and returns a `ZIO` effect that either succeeds with a `String` containing the HTML associated with the `URL` or fails with a `Throwable`.

We know we need to return a `ZIO` effect here because `getURL` will potentially have to do real network I/O, so we need `ZIO` to help us manage that. And we use `Throwable` as the error type because we know these types of I/O operations can potentially fail with a `Throwable` but exactly which type of `Throwable` may depend on the implementation.

We will also take this opportunity to define our own `URL` data type that wraps a `java.net.URL`. This isn't particularly related to ZIO but just gives us some smart constructors and convenience methods for working with URLs so it is included here in case you want to follow along at home.

```
final class URL private (private val parsed: java.net.URL) { self =>
  override def equals(that: Any): Boolean =
    that match {
      case that: URL => this.parsed == that.parsed
      case _         => false
    }
  override def hashCode: Int =
    parsed.hashCode
  def relative(page: String): Option[URL] =
    try {
      Some(new URL(new java.net.URL(parsed, page)))
    } catch {
      case t: VirtualMachineError => throw t
      case _: Throwable           => None
    }
  override def toString: String =
    url
  def url: String =
    parsed.toString
}

object URL {
  def make(url: String): Option[URL] =
    try {
      Some(new URL(new java.net.URL(url)))
    } catch {
```

```
      case t: VirtualMachineError => throw t
      case _: Throwable           => None
    }
}

def extractURLs(root: URL, html: String): Set[URL] = {
  val pattern = "href=[\"\']([^\"\']+)[\"\']".r

  scala.util
    .Try({
      val matches = (for (m <- pattern.findAllMatchIn(html)) yield m.group(1)).toSet

      for {
        m   <- matches
        url <- URL.make(m) ++ root.relative(m)
      } yield url
    })
    .getOrElse(Set.empty)
}
```

The next step is for us to put this signature into the format of the module pattern so it is easy to compose with other services in the ZIO environment.

```
object web {

  type Web = Has[Web.Service]

  object Web {
    trait Service {
      def getURL(url: URL): ZIO[Any, Throwable, String]
    }
  }

  def getURL(url: URL): ZIO[Web, Throwable, String] =
    ZIO.accessM(_.get.getURL(url))
}
```

Recall that by convention we create an `object` or `package object` with the name of the service in lowercase to create a namespace for all of the functionality related to this service.

Then we define a type alias `type Web = Has[Web.Service]` to allow us to take advantage of the power of the `Has` data type to allow us to compose services without having to write `Has` everywhere. In the object `Web` we then define `Service` which actually describes the interface we sketched out above.

Finally we define an environmental accessor to make it easy for us to access a `Web` module in the environment and call the `getURL` method on it. With this

we can take advantage of the functionality of the `Web` service just by writing `web.getURL(url)`.

So far we have implemented the interface of the `Web` module so it will be easy for us to call `getURL` within our implementation of the `crawl` method. But we do not yet have any actual implementations of the `Web` module so we have no way to actually satisfy the dependency of `crawl` on the `Web` module and run our web crawler, even if we did implement the crawl method!

Let's fix that by implementing a live version of the `Web` module that will return HTML for a specified URL by actually retrieving the URL.

Following the module pattern, we know that we want to define each implementation of our module as a `ZLayer`. This will allow us to have our module potentially depend on other modules and use effects and finalization logic if necessary, and will also allow users of our module to provide it in the same way as other modules they are working with.

```
lazy val live: ZLayer[Any, Nothing, Web] =
  ???
```

How do we go about actually implementing this service? There are a variety of frameworks we could use to retrieve the HTML associated with a URL but for simplicity we will use `scala.io.Source` from the Scala standard library.

An initial implementation might look like this:

```
val live: ZLayer[Any, Nothing, Web] =
  ZLayer.succeed {
    new Web.Service {
      def getURL(url: URL): ZIO[Any, Throwable, String] =
        ZIO.effect(scala.io.Source.fromURL(url.url).getLines.mkString)
    }
  }
// live: ZLayer[Any, Nothing, Web] = Managed(zio.ZManaged$$anon$2@1809c847)
```

Here we are simply using the `fromURL` method on `Source` to construct a `Source` from a `URL`, then calling `getLines` to get all of the lines of the HTML document and calling `mkString` to combine them all into a single string of HTML. We are using the `ZIO.effect` constructor because `Source.fromURL` can throw exceptions, for example if the URL cannot be found, so we use the `effect` constructor to signal that and allow the ZIO runtime to manage those errors for us.

However, there is still something wrong with this. In addition to potentially throwing exceptions, this effect is also *blocking*. Retrieving content over the network takes a very long time relative to CPU bound operations and during this time the thread running this effect will block until the results are available.

We need to be very careful to avoid running blocking effects on ZIO's main asynchronous thread pool because by default ZIO's runtime works with a small number of threads that execute many fibers. If those threads are stuck performing

blocking effects they are not available to perform other effects in our program, potentially resulting in performance degradation or even thread starvation.

To avoid that, we want to use the `Blocking` service to run potentially blocking effects on a separate thread pool that is optimized for blocking workloads and is able to spin up additional threads as necessary without taking up the threads in ZIO's core asynchronous thread pool.

This is where describing our implementation as a `ZLayer` pays off for us, because with `ZLayer` it is easy to describe one service that depends on another service. We now see that to implement the `live` version of our `Web` service we actually need a `Blocking` service.

So we update the signature of `live` like this:

```
import zio.blocking._

lazy val live: ZLayer[Blocking, Throwable, Web] =
  ???
```

Notice that we describe the dependency on `Blocking` as part of our `live` implementation and not as part of the `Web` service itself. Other implementations of the `Web` service, such as a test one that uses in-memory test data, might not do blocking at all, so `Blocking` belongs as a requirement of this particular implementation rather than the `Web` service in general.

To create a service that depends on another service we can use the `ZLayer.fromService` constructor, so our `live` implementation now looks like this:

```
val live: ZLayer[Blocking, Nothing, Web] =
  ZLayer.fromService { blocking =>
    new Web.Service {
      def getURL(url: URL): ZIO[Any, Throwable, String] =
        blocking.effectBlockingIO {
          scala.io.Source.fromURL(url.url).getLines.mkString
        }
    }
  }
// live: ZLayer[Blocking, Nothing, Web] = Managed(
//   zio.ZManaged$$anon$2@445c3fd5
// )
```

Now all requests to get content for web pages will be run on a separate blocking thread pool.

We can do more work to implement a test version of the `Web` module, but let's switch gears and work on the implementation of the web crawler itself now that we have the `Web` interface defined so we actually have something to test!

41.3 First Sketch Of A Parallel Web Crawler

Now that we have the `Web` interface defined, let's think about how we would actually implement the web crawler. At a high level, the process is:

1. For each seed, get the HTML associated with that seed.
2. Extract all the links from that HTML string
3. Repeat the process for each link

We also know that we want to do this in parallel.

A good initial approach is to try to translate our high level logic into code using existing operators. This may not always be sufficient, but it is usually a helpful starting place and then we can adjust as necessary.

Here, the description of "for each seed, get the HTML associated with that seed, in parallel" translates quite nicely to the `foreachPar` operator defined by ZIO, so our first version might look something like this:

```
def crawl[R, E](
  seeds: Set[URL],
  router: URL => Boolean,
  processor: (URL, String) => ZIO[R, E, Unit]
): ZIO[R with Web, Nothing, List[E]] = {
  ZIO.foreachPar_(seeds) { url =>
    web.getURL(url).flatMap { html =>
      val urls = extractURLs(url, html)
      processor(url, html).catchAll(e => ???) *>
        crawl(urls.filter(router), router, processor)
    }
  }
  ???
}
```

There is a lot good here. For each of the initial seeds we are, in parallel, getting the HTML associated with that URL. We are then extracting all the links from that HTML using the `extractURLs` helper function, sending the URL and HTML to the `processor`, and recursively calling `crawl` with the new links.

However, writing this out in code has also revealed a couple of problems that we need to address.

First, it is not clear what we are supposed to do with the errors here. We know that `crawl` is supposed to succeed even if `processor` fails so we need to do something in `catchAll` to handle the potential error but we don't have anything to do with it right now other than just ignore it.

This same problem shows up in the return value. We are supposed to return a `List` of all the errors that occured, but right now we don't have anywhere to get

that list from, other than just always returning an empty list, which is clearly not right.

Second, right now we are not doing anything to keep track of which sites we have already visited, if we have two sites that directly or indirectly link to each other, which is very common, right now we will continue forever, repeatedly exploring each site. This is clearly not what we want.

Both of these problems are indicators that we need to be maintaining some *state* in the process performing our algorithm. Let's introduce a data type to capture the state that we want to maintain.

```
final case class CrawlState[+E](errors: List[E], visited: Set[URL]) {
  def visit(url: URL): CrawlState[E] =
    copy(visited = visited + url)
  def visitAll(urls: Iterable[URL]): CrawlState[E] =
    copy(visited = visited ++ urls)
  def logError[E1 >: E](error: E1): CrawlState[E1] =
    copy(errors = error :: errors)
}

object CrawlState {
  val empty: CrawlState[Nothing] =
    CrawlState(List.empty, Set.empty)
}
```

We could try to pass the `CrawlState` around in an recursive loop, but the other thing we know is that we want the web crawler to be parallel, which means we could have multiple fibers updating the `CrawlState` at the same time to log errors or get or update the set of visited sites.

Whenever you have some piece of state that multiple fibers need to update, think about using a `Ref`. Of course there are other more powerful solutions like using a `TRef`, but always start with a `Ref` and go from there if needed.

With `CrawlState` and using a `Ref`, we could try to refactor our implementation of `crawl` like this:

```
def crawl[R, E](
  seeds: Set[URL],
  router: URL => Boolean,
  processor: (URL, String) => ZIO[R, E, Unit]
): ZIO[R with Web, Nothing, List[E]] =
  Ref.make[CrawlState[E]](CrawlState.empty).flatMap { ref =>
    def loop(seeds: Set[URL]): ZIO[R with Web, Nothing, Unit] =
      ZIO.foreachPar_(seeds.filter(router)) { url =>
        ref.modify { crawlState =>
          if (crawlState.visited.contains(url))
            (ZIO.unit, crawlState)
```

```
          else
            (
              getURL(url).flatMap { html =>
                processor(url, html).catchAll(e => ref.update(_.logError(e))) *>
                loop(extractURLs(url, html))
              }.ignore,
              crawlState.visit(url)
            )
        }.flatten
    }
    loop(seeds) *> ref.get.map(_.errors)
}
```

Let's walk through this implementation.

We start by creating a `Ref` that contains the `CrawlState`, starting with an empty state where we have not visited any sites and not logged any errors.

We then call the new inner `loop` function we defined, which contains much of the logic that was in our previous implementation of `crawl`. Once again we call `foreachPar` to perform an effect in parallel for each seed, except now our logic is slightly more complex.

We need to make sure that we are not crawling a site we have already visited, so we call `Ref#modify` to allow us to access the current `CrawlState`, update it, and return some other value based on that. Within `modify` we have two possibilities to consider.

First, the URL may already have been visited. In this case we can just return the previous `CrawlState` unchanged and return `ZIO.unit`, an effect that does nothing, since there is no more work to do if the site has already been visited.

Second, the URL may not have been visited yet. In that case we need to update the `CrawlState` to include the new `URL` so that no other fiber crawls that `URL`. Then we need to get the HTML string associated with that `URL`, process it, and recursively call `loop` with the new URLs we extracted from the HTML string.

Now that we have defined `CrawlState` we also have a meaningful way to handle errors that occur during `processor` since we can update the `CrawlState` to log those errors.

Finally, when `loop` is done we just need to access the `CrawlState` and get the list of all the errors that occurred to return it.

This now seems not obviously wrong, but does it work? To answer that we will need a way to test our web crawler, which will require going back to do more work with the `Web` service.

41.4 Making It Testable

To test our implementation of the `crawl` method we are going to need a test version of the `Web` service that returns deterministic results for specified URLs. Fortunately, ZIO's solution for dependency injection makes it easy for us to do that without having to refactor any of our other code.

To do it, we just need to define a `test` implementation of our `Web` service.

```
lazy val test: ZLayer[Any, Nothing, Web] =
  ???
```

What would a test implementation of the `Web` service look like? Well a simple version would be backed by a set of prepopulated test data for a small number of sites. Then we could verify that crawling those sites produced the expected result.

To do that, let's start by defining some test data. We will use some sample data for the ZIO homepage, but you can test with whatever data you would like.

```
val Home          = URL.make("http://zio.dev").get
// Home: URL = http://zio.dev
val Index         = URL.make("http://zio.dev/index.html").get
// Index: URL = http://zio.dev/index.html
val ScaladocIndex = URL.make("http://zio.dev/scaladoc/index.html").get
// ScaladocIndex: URL = http://zio.dev/scaladoc/index.html
val About         = URL.make("http://zio.dev/about").get
// About: URL = http://zio.dev/about

val SiteIndex =
  Map(
    Home          -> """<html><body><a href="index.html">Home</a><a href="/scaladoc/index.ht
    Index         -> """<html><body><a href="index.html">Home</a><a href="/scaladoc/index.ht
    ScaladocIndex -> """<html><body><a href="index.html">Home</a><a href="/about">About</a><
    About         -> """<html><body><a href="home.html">Home</a><a href="http://google.com"
  )
// SiteIndex: Map[URL, String] = Map(
//   http://zio.dev -> "<html><body><a href=\"index.html\">Home</a><a href=\"/scaladoc/index
//   http://zio.dev/index.html -> "<html><body><a href=\"index.html\">Home</a><a href=\"/sca
//   http://zio.dev/scaladoc/index.html -> "<html><body><a href=\"index.html\">Home</a><a ht
//   http://zio.dev/about -> "<html><body><a href=\"home.html\">Home</a><a href=\"http://goo
// )
```

With this data, we can define a test version of the `Web` service like this:

```
val testLayer: ZLayer[Any, Nothing, Web] =
  ZLayer.succeed {
    new Web.Service {
      def getURL(url: URL): ZIO[Any, Throwable, String] =
```

```
        SiteIndex.get(url) match {
          case Some(html) =>
            ZIO.succeed(html)
          case None =>
            ZIO.fail(new java.io.FileNotFoundException(url.toString))
        }
    }
  }
// testLayer: ZLayer[Any, Nothing, Web] = Managed(
//   zio.ZManaged$$anon$2@770fe1e9
// )
```

The implementation is quite simple. When we receive a request to get the HTML content associated with a URL we simply check whether it is in the test data. If so we return the corresponding HTML and otherwise we fail with a `FileNotFoundException`.

With a test version of the `Web` service we are most of the way towards testing our web crawler. The only things that are left are implementing test versions of the `router` and `processor` and writing the actual test.

The `router` is quite straightforward, it is just a function `URL => Boolean` indicating whether a `URL` should be crawled. Let's implement a simple router that only crawls the `zio.dev` domain to verify that our web crawler doesn't crawl Google's home page, which is linked to from the About page.

```
val testRouter: URL => Boolean =
  _.url.contains("zio.dev")
// testRouter: URL => Boolean = <function1>
```

The implementation of the `processor` is only slightly more complex.

Recall that the only observable result of crawling each page is what the `processor` does with the page. If we want to verify all the pages processed an easy solution is to update a `Ref` with each of them.

```
def testProcessor(ref: Ref[Map[URL, String]]): (URL, String) => ZIO[Any, Nothing, Unit] =
  (url, html) => ref.update(_ + (url -> html))
```

We will create the `Ref` in our test and then pass it to the `testProcessor` function so that the web crawler updates the `Ref` with each page crawled. Then we can check that the `Ref` contains the expected results.

With all the pieces in place, the test itself is actually quite simple.

```
import zio.test._
import zio.test.Assertion._

testM("test site") {
  for {
    ref     <- Ref.make[Map[URL, String]](Map.empty)
```

```
    _       <- crawl(Set(Home), testRouter, testProcessor(ref))
    crawled <- ref.get
  } yield assert(crawled)(equalTo(SiteIndex))
}.provideCustomLayer(testLayer)
// res1: Spec[environment.package.TestEnvironment, TestFailure[Nothing], TestSuccess] = Spe
//   TestCase(
//     "test site",
//     zio.ZIO$CheckInterrupt@5b04130f,
//     Map(zio.test.TestAnnotation@4cf27564 -> List(SourceLocation(41-applications-parallel
//   )
// )
```

If you try running this test you will see that our parallel web crawler is indeed working with our test data!

41.5 Scaling It Up

So are we done? Well not quite.

To see why, now that we have tested our parallel web crawler let's try running it for real.

To do so, we will need to select a set of `seeds` and a `router` and `processor`.

Let's use the New York Times homepage, located at https://www.nytimes.com/, as a seed and let's use a router that only crawls pages on this domain. This will prevent our web crawler from trying to crawl the entire web but will also try to crawl a reasonably large number of pages so we can see how it scales up.

```
val seeds: Set[URL] =
  Set(URL.make("https://www.nytimes.com").get)
// seeds: Set[URL] = Set(https://www.nytimes.com)

val router: URL => Boolean =
  _.url.contains("https://www.nytimes.com")
// router: URL => Boolean = <function1>
```

For our `processor` we will use a simple one that just prints each URL to the console. This will avoid us being overwhelmed by the entire HTML string from each page while allowing us to visually get a sense of what the web crawler is doing.

```
import zio.console._

val processor: (URL, String) => ZIO[Console, Nothing, Unit] =
  (url, _) => console.putStrLn(url.url)
// processor: (URL, String) => ZIO[Console, Nothing, Unit] = <function2>
```

You can then try running the web crawler like this:

```
import zio.{App => ZIOApp}

object Example extends ZIOApp {
  def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] =
    for {
      fiber <- crawl(seeds, router, processor).provideCustomLayer(live).fork
      _     <- console.getStrLn.orDie
      _     <- fiber.interrupt
    } yield ExitCode.success
}
```

If you try running this program you will see that the web crawler does initially crawl a large number of pages from the New York Times domain. However, before too long the application will slow down and eventually crash with an out of heap space error.

What is going on here?

The answer is that our current recursive implementation, while having a certain elegance, is not very resource safe.

As conceptualized here our parallel web crawler is inherently not entirely resource safe since it maintains the set of visited websites in memory and this set could potentially grow without bound. But if we add a debug statement to show the size of the visited set over time we see that the set is not that large when we run into problems.

The issue, rather, is with the way we are forking fibers with `foreachPar`. Specifically, there are two related issues.

First, we are just forking a very large number of fibers. If the initial seed has ten links and each of those pages has another ten links and so on we will very quickly create an extremely large number of fibers.

Fibers are much cheaper than operating system threads but they are not free and here we are creating a very large number of fibers that probably exceeds our ability to efficiently run them.

We could potentially address this by using `foreachParN` to limit the degree of parallelism, but that doesn't completely solve our problem because `foreachParN` is called recursively. So even if we limit the original invocation to using, say, 100 fibers, each of those fibers would call `foreachParN` and potentially fork another 100 fibers.

Another solution would be to use a single `Semaphore` we created in `crawl` and require each effect forked in `loop` to acquire a `permit`. This would indeed limit the total parallelism in `crawl` to the specified level.

However, there is another problem. To be resource safe we want each fiber to be done when it is done with its work so that it can be garbage collected. But with any of the implementations we have discussed with `foreachPar` parent fibers

can't terminate until all of their children terminate, which means they can't be garbage collected.

To see this, think about the very first fiber that crawls the New York Times home page and forks ten other fibers. This fiber won't be done until all of the fibers it forked in `foreachPar` have completed and returned their results.

But each of these forked fibers is crawling one of the linked pages and forking more fibers to explore all of the linked pages, so those fibers also can't be done until all of their children are done.

The result is that none of the fibers "higher" in the fiber graph can terminate or be garbage collected until all of the fibers below them have terminated.

The combination of a large number of fibers being forked and the inability to garbage collect fibers results in more and more heap space being used until eventually we run out.

So how can we do better?

Doing so requires reconceputualizing our algorithm. The recursive solution is elegant, but as we saw above it creates these trees of fibers that are useful in general but we don't need here and have overhead that we don't want to pay for.

Another way to do the same thing would be a more "imperative" solution.

We already said above we wanted to limit the degree of parallelism to a specified number of fibers. So let's just create that number of fibers.

We will keep the URLs we have extracted that still need to be processed in a `Queue`, and each fiber will repeatedly take a `URL` from the queue, process it, offer any extracted links back to the queue, and repeat that process until there are no more URLs in the queue.

This adds a certain amount of additional complexity but is much more efficient because now there are only ever a fixed number of fibers and there are no additional resources that are not cleaned up after each URL is processed other than the set of visited sites, which is unavoidable short of moving that to a database.

Here is what this implementation might look like:

```
import web._

def crawl[R, E](
  seeds: Set[URL],
  router: URL => Boolean,
  processor: (URL, String) => ZIO[R, E, Unit]
): ZIO[R with Web, Nothing, List[E]] =
  Ref.make[CrawlState[E]](CrawlState.empty).flatMap { crawlState =>
    Ref.make(0).flatMap { ref =>
      Promise.make[Nothing, Unit].flatMap { promise =>
```

```
        ZIO.bracket(Queue.unbounded[URL])(_.shutdown) { queue =>
          val onDone: ZIO[Any, Nothing, Unit] =
            ref.modify { n =>
              if (n == 1) (queue.shutdown <* promise.succeed(()), 0)
              else (ZIO.unit, n - 1)
            }.flatten
          val worker: ZIO[R with Web, Nothing, Unit] =
            queue.take.flatMap { url =>
              web.getURL(url).flatMap { html =>
                val urls = extractURLs(url, html).filter(router)
                for {
                  urls <- crawlState.modify(state => (urls -- state.visited, state.visitAll(
                  _    <- processor(url, html).catchAll(e => crawlState.update(_.logError(e)
                  _    <- queue.offerAll(urls)
                  _    <- ref.update(_ + urls.size)
                } yield ()
              }.ignore <* onDone
            }
          for {
            _     <- crawlState.update(_.visitAll(seeds))
            _     <- ref.update(_ + seeds.size)
            _     <- queue.offerAll(seeds)
            _     <- ZIO.collectAll(ZIO.replicate(100)(worker.forever.fork))
            _     <- promise.await
            state <- crawlState.get
          } yield state.errors
        }
      }
    }
  }
```

The tricky issue here is how to know when we are done.

It would be tempting to say that if the queue of URLs that have been extracted but not processed is empty then we are done. However, that is not necessarily true because another fiber could have taken a URL from the queue, leaving it empty, but be about to offer new URLs it had extracted back to the queue.

The problem is that a URL is removed from the queue as soon as a fiber starts processing it but we don't really want to decrease the number of pending URLs observed by other fibers until the fiber is done processing it and has already offered any extracted URLs back to the queue.

To handle this, we add an additional piece of state captured in a `Ref[Int]` representing the number of extracted but not processed URLs. Fibers increment this by the number of newly extracted URLs after offering them to the queue and decrement it when the fiber completes processing a URL.

This way we know that when a fiber finishes processing an item and there are no other extracted but unprocessed URLs it is safe to terminate the crawl. We then complete a `Promise` that we wait on and shut down the `Queue` so that all the other fibers are interrupted when they attempt to offer or take values from it.

Notice that we also use `bracket` when we make the `Queue` with `shutdown` as the `release` action so that if the crawl is interrupted the `Queue` will be shut down and all of the fibers will be interrupted.

If you try running this version you will see that it continues to process new pages.

You can interrupt it at any time by pressing any key. Note that it may take a minute for the program to terminate as fibers that are currently processing a URL will not be interrupted until they complete the URL they are processing.

41.6 Conclusion

In this chapter we worked through the first of our applications, using ZIO to build a parallel web crawler. In the process we reinforced what we have learned about importing blocking effects, ZIO's fiber based concurrency model, concurrent data structures, and dependency injection.

Stepping back, one of the lessons of this chapter is the value of working iteratively. We started with an implementation of the parallel web crawler that was quite simple but not as efficient. At a small scale this would be a perfectly workable solution to the problem but at a larger scale we ran into issues.

We then showed a more complex but also more efficient implementation of the parallel web crawler. This works well at a medium scale but at a large scale this would also not work as we store the set of visited sites in memory and it potentially grows without bound as we visit more and more sites.

If you are interested in spending more time on this, a further project could be to build in a database to improve this.

In the next chapter we will work through an application involving file processing which will give us the chance to work with managed resources, streams, and importing effects and is a good example of how you can use ZIO to solve some very practical problems that you may encounter on a day to day basis. # Applications: File Processing

Chapter 42

Applications: Command Line Interface

Chapter 43

Applications: Kafka Stream Processor

Chapter 44

Applications: gRPC Microservices

Chapter 45

Applications: REST API

Chapter 46

Applications: GraphQL API

Chapter 47

Applications: Spark

Chapter 48

Appendix: The Scala Type System

This chapter will provide an overview of Scala's type system for working with ZIO and libraries in the ZIO ecosystem. There is a lot that could be said about this topic so this chapter will focus on what you need to know on a day to day basis to take advantage of Scala's type system and make the compiler work for you.

48.1 Types And Values

At the highest level, everything in Scala is either a *type* or a *value*.

A type is a kind of thing, for example fruits, animals, or integers.

A value is a specific instantiation of a type, for example one particular apple, your dog Fido, or the number one.

You can also think of a type as the set of all values belonging to that type. For example you can think of the type `Int` as the set of all Java integers.

We create new types in Scala every time we define a `class` or `trait`. We typically define certain operators on these types that allow us to use them to solve problems.

```
trait Animal {
  def name: String
}
```

Being a statically typed language, Scala prevents us from calling operators on values that are not of the appropriate type.

```
val baxter: Animal =
  new Animal {
    def name: String =
      "baxter"
  }

println(baxter.name)
// okay

println(1.name)
// does not compile
```

This helps prevent a wide variety of bugs.

48.2 Subtyping

Scala's type system also supports subtyping, so in addition to just having completely unrelated types like animals and fruits we can express that one type is a subtype of another.

For example, we could define a type `Cat` that is a subtype of `Animal`. This indicates that every `Cat` is an animal, but every `Animal` is not necessarily a `Cat`.

Graphically, if we drew a Venn diagram `Animal` would be a large circle and `Cat` would be a smaller circle completely within that first circle.

We define subtyping relationships in Scala using the `extends` keyword.

```
trait Cat extends Animal
```

The guarantee of a subtyping relationship is that if `A` is a subtype of `B` then in any program we should be able to safely replace an instance of `B` with an instance of `A`.

For example, if I have a program that prints the name of a specified animal to the console I should be able to provide it with a `Cat` instead of an `Animal`.

```
def printName(animal: Animal): Unit =
  println(animal.name)

val baxterCat: Cat =
  new Cat {
    def name: String =
      "baxter"
  }

printName(baxterCat)
// okay
```

In fact we can always treat an instance of a type as an instance of one of its supertypes.

```
val baxterAnimal: Animal =
  baxterCat
```

This reflects the fact that a `Cat` *is* an animal, or using the set view that Baxter is a member of both the set of all cats and the set of all animals.

Subtyping allows us to more accurately model domains we are working with and to share functionality across related data types.

For example, using subtyping we could express the concept of a `PaymentMethod` that has certain functionality and then various subtypes of `PaymentMethod` such as `CreditCard`, `DebitCard`, and `Check` that provide additional functionality.

We can check whether one type is a subtype of another using Scala's `<:<` operator, which requires implicit evidence that `A` is a subtype of `B` that only exists if `A` actually is a subtype of `B`. Here is an example of using it:

```
implicitly[Cat <:< Animal]
// okay

implicitly[Animal <:< Cat]
// does not compile
```

Note that if `A` is a subtype of `B` and `B` is a subtype of `A` then `A` is the same type as `B`. Again we can see that easily with the view of types as sets where if every element of set `A` is an element of set `B` and every element of set `B` is an element of set `A` then the two sets are the same.

48.3 Any And Nothing

There are two values that have special significance within the Scala type system, `Any` and `Nothing`.

48.3.1 Any

`Any` is a supertype of every type.

```
implicitly[Animal <:< Any]
// okay


implicitly[Int <:< Any]
// okay
```

Because it is a supertype of every other type, it is also sometimes referred to as at the "top" of Scala's type system.

The fact that `Any` is a supertype of every other type places severe restrictions on what we can do with a value of type `Any`.

Any functionality that `Any` provided would have to be implemented by every type we could possibly implement. But we saw above that we could create a new `class` or `trait` without implementing any operators.

So `Any` models a set that includes every value but essentially does not have any capabilities at all. This is not strictly true because Scala defines some basic methods such as `hashCode` and `toString` on every value, but we generally do not rely on that when reasoning about types.

One important implication of this is that we can always create a value of type `Any` by providing any value at all.

```
val any: Any = 42
```

Customarily we often use the `Unit` value `()` as an instance of `Any`, since `Unit` is a type that only contains a single value and so does not model any information.

If `Any` represents the set of all possible values and has no capabilities, how can it be useful to us? Because sometimes we don't care about a value.

One of the most common places this comes up in ZIO is in working with the environment type. Recall that a `ZIO` value models an effect that requires an environment `R` and may either fail with an `E` or succeed with an `A`.

In simplified form:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])
```

But how do we model effects that don't require any environment at all? Do we need a separate data type for them?

No! We can use `Any` as the environment type to model effects that do not require any environment at all.

```
type IO[+E, +A] = ZIO[Any, E, A]
```

And because we can always produce a value of `Any` by providing any arbitrary value, we can run a `ZIO` effect that requires an environment of type `Any` simply by providing the `Unit` value.

```
def run[E, A](zio: IO[E, A]): Either[E, A] =
  zio.run(())
```

If a `ZIO` effect does not use the environment at all then it doesn't matter what environment we provide. So `Any` provides an excellent way for us to model that.

The environment type is an example of using `Any` to model an input we don't care about. Sometimes we also don't care about the output.

Consider the signature of the `ensuring` operator on `ZIO`:

```
trait ZIO[-R, +E, +A] {
  def ensuring[R1 <: R](finalizer: URIO[R1, Any]): ZIO[R1, E, A]
}
```

The `ensuring` operator allows the user to attach a finalizer to an effect that will be run when the effect terminates, regardless of whether that is due to success, failure, or interruption. The finalizer could log information to the console, close a file, or do nothing at all.

The use of the `Any` signature in `f` reflects this nicely. We aren't going to do anything with the result of the finalizer because we are still going to return the result of the original effect rather than the finalizer.

So from our perspective the return type of the finalizer can really be anything at all. We will happily run the finalizer when the effect completes but we don't care what it returns.

We could use the `Unit` value for this, but that could require the user to transform the result of the finalizer if it would have otherwise returned some other type. There is no need for that when we aren't going to do anything with the value anyway.

So we have seen how despite `Any` represents the type of any possible value and having no functionality, it can be useful to model situations where we don't care about the value.

48.3.2 Nothing

`Nothing` is a type for which there are no values. Thinking about types as sets of values, `Nothing` represents the empty set.

One of the interesting implications of `Nothing` representing the empty set is that `Nothing` is a subtype of every other type.

```
implicitly[Nothing <:< Cat]
// okay

implicitly[Nothing <:< Int]
// okay
```

Thinking in terms of sets, the empty set is a subset of every other set.

Thinking in terms of capabilities, it is safe to treat `Nothing` as any other type because we could never actually have a value of type `Nothing` to call any operators on.

Because `Nothing` is a subtype of every other type, it is often referred to as being at the "bottom" of Scala's top system.

As with `Any`, we might be wondering how `Nothing` can ever be useful if there can never be a value of type `Nothing`. The answer is that it can be used to express with the type system that a state cannot exist.

For example, consider the error type of a `ZIO` effect. A `ZIO[R, E, A]` is an effect that requires an environment `R` and may either fail with an `E` or succeed

with an `A`.

How then do we model effects that can't fail at all? By using `Nothing` for the error type.

```
type URIO[-R, +A] = ZIO[R, Nothing, A]
```

We know there can never be a value of type `Nothing` so an effect of type `URIO` must either succeed with an `A` or continue forever.

Similarly if we have an effect where the value type is `Nothing` we know the effect can never succeed but will either run forever or fail.

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

The fact that `Nothing` is a subtype of every other type can also help us use the compiler to "prove" that certain transformations are valid.

For example, say we are using `Either[E, A]` to represent a result that is either a failure of type `E` or a success of type `A`. We could then model a result that cannot be a failure as `Either[Nothing, A]`.

If there can be no values of type `Nothing` then this must always be a `Right` with an `A` in it so it should be safe to extract out the `A` value.

Can we use the compiler to help us prove that is valid? Yes we can!

```
def get[A](either: Either[Nothing, A]): A =
  either.fold(e => e, a => a)
```

Because `Nothing` is a subtype of every other type, `Nothing` is also a subtype of `A` so we can fold over the `Either` with the `identity` function to get back the `A` value.

`Nothing` can also be used with input types to model a computation that can never be run. For example, a `ZIO[Nothing, E, A]` is an effect that can never be run because we could never provide it with a value of type `Nothing`.

However, this is typically less useful as there is no way to run the computation at all, unless that is the point. For example, the `readOnly` method on `ZRef` returns a `ZRef` where the type of value that can be set is `Nothing`, indicating that the user can never write a value to the `ZRef` but only read the current value.

Note that in Scala a thrown exception also has a type of `Nothing` so we can technically throw an exception as a value of type `Nothing` but we try to avoid doing this whenever possible.

48.4 Product And Sum Types

We can build more complex types out of simpler ones using *product* and *sum* types.

48.4.1 Product Types

Product types are types that are made up of values of one type and values of another type. For example, a `User` might consist of a `String` name and an `Int` age.

In Scala product types are modeled using case classes or tuples.

```
final case class User(name: String, age: Int)

val point: (Int, Int) = (0, 0)
```

Product types are extremely useful for modeling data that contains multiple parts. The `User` example above is a simple one but we could imagine capturing a variety of other information about a user.

Product types can also be nested, for example:

```
final case class Name(first: String, last: String)

final case class User(name: Name, age: Int)
```

These are called *product types* because the number of possible values of a product type is the *product* of the number of possible values for each of the types it is composed from.

Thinking about types in terms of sets, if we have a set of `A` values and a set of `B` values then the set of `A` and `B` values is the Cartesian product of the sets of `A` values and `B` values.

For example, say we have a `Status` data type to capture the status of a fiber. The fiber can be either interrupted or not and can either be in the process of interrupting itself or not.

```
final case class Status(
  interrupted: Boolean,
  interrupting: Boolean
)
```

Values of type `Boolean` can have two possible values, `true` and `false`. So for the product type `Status` there are two times two equals four possible states, corresponding to `(true, true)`, `(true, false)`, `(false, true)`, and `(false, false)`.

In this simple example this may seem obvious but in more complex cases it can provide a useful way to think about all the possible states we must consider.

If all states in the product type do not correspond to possible states of the domain then it may indicate a need to refactor.

The logic above may also indicate that some values are not possible at all. For example consider the following `Annotated` type, representing a value of type `A` along with some metadata.

```
final case class Annotated[+A](value: A, annotations: List[String])
```

Just by inspection, we can observe that there can never be a value of type `Annotated[Nothing]`.

The number of possible values of type `Annotated[Nothing]` is equal to the number of possible values of `value` multiplied by the number of possible values of `annotations`. We know there are no values of type `Nothing` and zero times any number is still zero, so we know there cannot be any values of type `Annotated[Nothing]`.

This also makes sense if we think about product types as types that are made up of two or more other types. An `Annotated[A]` has to contain a value of type `A`, so if we can never construct a value of type `A` then we can also never construct a value of type `Annotated[A]`.

48.4.2 Sum Types

Sum types are types that are made up of values that are either values of one type or values of another type. For example, an `Exit` representing the result of a `ZIO` effect could be either a `Success` or a `Failure`.

Sum types are represented in Scala as a `sealed trait` or `sealed abstract class` with subtypes representing each of the possibilities. `Either` represents a generic sum type much like `Tuple` represents a generic product type.

```
sealed trait Exit[+E, +A]

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

val eitherStringInt: Either[String, Int] = Right(1)
```

Sum types model data that is either one type or another but not both. For example, a user is `LoggedIn` or `LoggedOut`.

These are called `sum` types because the total number of possible values of a sum type is the *sum* of the number of possible values for each alternative.

In the conception of types as sets of values, the set of either `A` or `B` values is the union of the set of `A` values and the set of `B` values.

For example, say the result of looking up a value in a cache can either be a failure of type `E`, a success but with no value in the cache, or a success with a value in the cache of type `A`. We could model this as

```
type LookupResult[+E, +A] = Either[E, Option[A]]
```

How many possible states are there? To answer this we need to recognize that `Option` is itself a sum type with subtypes of `None`, which only has a single value, or `Some`, which has as many possible values as `A` values. So the possible states are `None`, plus the number of possible states of `E` plus the number of possible states of `A`.

Again, this may seem obvious in a simple example like this but even here thinking this way can pay dividends.

Often there are multiple ways to represent the same possible states. For example, we could instead model this as:

```
type LookupResult[+E, +A] = Either[Option[E], A]
```

Our rules about the number of possible states tell us that these encode the same information. But this encoding could be more performant if we are doing further computations on the `A` value to avoid multiple layers of wrapping and unwrapping.

This type of reasoning can be particularly valuable with the `Any` and `Nothing` types.

`Nothing` is a type for which there are no values, so any time we see a sum type for which one of the possibilities is impossible we can eliminate it. For example, if we have `Either[Nothing, A]` we can simplify that to just `A`.

48.4.3 Combining Product And Sum Types

Product and sum types can be combined to create rich domain models. For example a `User` may be a product type that is composed of a variety of fields, some of which are themselves sum types representing alternative states such as being logged in or logged out.

48.5 Intersection And Union Types

48.5.1 Intersection Types

Product types are types that contain values of two or more types. For example a `User` *has* both a `name` and an `age`.

Intersection types are types that *are* values of two or more types. For example, a `Cat` is an `Animal` but it may also be a `Greeter`.

This is modeled in Scala by using the `with` keyword with multiple traits.

```
trait Animal {
  def name: String
}

trait Greeter {
  def greet: Unit
}

final case class Cat(name: String) extends Animal with Greeter {
  def greet: Unit =
    println("meow")
}
```

Notice that intersection types model capabilities, not just data. A `Cat` *is* an `Animal` and *is* a `Greeter` and has all of the capabilities described by these two types.

Intersection types are important in ZIO because when the Scala compiler unifies two contravariant types it tries to find the intersection of the two types.

The next appendix provides additional detail regarding variance, but for now we can focus on the environment type `R` of `ZIO`, which represents the required environment for an effect.

When we compose multiple effects that each require some environment, the Scala compiler needs to unify them to a common environment type that is required by the composed effect.

```
import zio._

val effect = for {
  time <- clock.nanoTime
  _    <- console.putStrLn(time.toString)
} yield ()
```

What should be the environment type for `effect`? The answer is the intersection type of the environment required by each composed effect. In this case since `nanoTime` requires `Clock` and `putStrLn` requires `Console` the required environment type will be `Clock with Console`.

We have seen throughout this book that `ZLayer` typically provides a more ergonomic way to build environments than creating intersection types directly. But it is helpful to understand that there is nothing magical about the `with` keyword here.

The `Clock with Console` in the type signature indicates that this effect needs something that provides both the functionality of the `Clock` service and the functionality of the `Console` service. This makes sense because we need to use both capabilities for different parts of this effect.

One of the nice properties of intersection types is that they are generally associative and commutative, that is `Console with Clock` is the same as `Clock with Console` and the order we combine them does not matter. This is not necessarily true if we use more object oriented patterns where traits call implementations in other traits in supertypes, but we generally avoid that to maintain these attractive compositional properties.

48.5.2 Union Types

Union types are the conceptual analogue of intersection types. Whereas an intersection type represents a type that is *both* one type and another, a union type represents a type that is *either* one type or another.

For example, we might want to define a type `Pet` that is either a `Cat` or a `Dog`. Unfortunately union types are not currently supported in Scala 2.

This creates some issues in working with the error type in ZIO because with covariant types like ZIO's error type the Scala compiler will try to unify them but is not able to unify them to a union type.

There isn't a problem when we define our error type as a sum type that precisely models the types of possible errors that can occur. For example, we can define a domain specific error type for getting a password from a user.

```
sealed trait PasswordError

sealed trait IOError extends PasswordError
sealed trait InvalidPassword extends PasswordError
```

Here `IOError` indicates there was an error in obtaining the password from the user. `InvalidPassword` indicates that we successfully obtained the password from the user but it does not match the expected value.

If we combine an effect that can fail with an `IOError` with an effect that can fail with an `InvalidPassword` the Scala compiler will infer the error type of the combined effect to be `PasswordError`.

```
import zio._
import zio.console._

def getInput: ZIO[Console, IOError, String] =
  ???

def validateInput(
  password: String
): ZIO[Any, InvalidPassword, Unit] =
  ???
```

```
def getAndValidateInput: ZIO[Console, PasswordError, Unit] =
  getInput.flatMap(validateInput)
```

This makes sense because if getting the input could fail with an `IOError` and validating the input could fail with an `InvalidPassword` then getting and validating the input could fail with either an `IOError` or an `InvalidPassword`, which is precisely what `PasswordError` represents.

But what happens if we add an additional failure state to password error?

```
sealed trait PasswordError

case object IOError extends PasswordError
case object InvalidPassword extends PasswordError
case object SecurityError extends PasswordError
```

`SecurityError` is an error indicating that even if the password is correct it is still not safe to proceed because of a security issue. For example, the user has entered too many incorrect passwords in a short period of time, indicating a potential hacking attempt.

The return type of the `getAndValidateInput` method will still be `ZIO[Console, PasswordError, Unit]`. But notice that we have thrown information away here!

`PasswordError` indicates that our effect can fail with either an `IOError`, an `InvalidPassword`, or a `SecurityError`. But in fact our effect can only fail with an `IOError` or an `InvalidPassword`.

By widening the error type to the most specific supertype `PasswordError` we have lost type information that this effect can never fail with a `SecurityError`.

Ideally we would like the error type of the combined effect in this case to be `IOError | InvalidPassword`, where `|` is pseudocode for the union of two types. But this doesn't exist in the type system of Scala 2, so the compiler has to go further up the type hierarchy to `PasswordError`, forgetting some information along the way.

We can recover more specific information about the error type by refactoring our model of the possible error types as follows:

```
sealed trait PasswordError

sealed trait NormalError extends PasswordError
case object SecurityError extends PasswordError

case object IOError extends NormalError
case object InvalidPassword extends NormalError
```

Now the return type of `getAndValidateInput` will be `NormalError`, indicating that this operator cannot fail for a `SecurityError`.

Thus, when creating our own domain specific error hierarchies it can be useful to use multiple sum types to model the different categories and subcategories of error that can occur.

While a dramatic improvement from other approaches to error handling, it would still be nicer if we could have a bit more flexibility in our domain modeling here.

The lack of union types particularly shows when dealing with error types outside of our control.

For example, if we have one effect that can fail with a `IOException` and another that can fail with a `NoSuchElementException`, the compiler will widen the error type to the common supertype of `Throwable`, indicating that this effect could fail for any possible throwable rather than just these two specific types of throwable.

Scala 3 with its support for union types should allow libraries like ZIO to improve this situation, making it even easier to work with the error type and combine effects that can fail in different ways in the future.

48.6 Type Constructors

So far we have only been talking about specific types, like oranges, animals, or integers. But Scala's type system is also powerful enough to support type constructors.

A type constructor is not itself a type but if given one or more other types it will produce a new type.

In Scala a type constructor is any `class`, `trait`, or type alias that has one or more type parameters. For example:

```
sealed trait List[+A]
// List is a type constructor

type IntList = List[Int]
```

Here `List` is a type constructor because it takes one or more type parameters, in this case `A`. We can create types by "feeding" different types into a type constructor.

For example we can replace `A` with `Int` to create a list of integers or with `String` to create a list of strings.

A good way to tell whether something is a type or a type constructor is whether we have enough information to create values of that type.

For example, we can readily create a `List[Int]`, such as `List(1, 2, 3)` because `List[Int]` is a type. But how do we create a `List`?

The answer is we can't, because `List` is a type constructor. Without knowing what the type of the elements of the list are we don't have enough information

to construct list values.

Type constructors may seem complicated but in fact you likely already work with them all the time. For example, we just saw that `List` is a type constructor as are other collection types like `Vector`, `Set`, and `Map`.

Type constructors can also take more than one type parameter. For example, `ZIO[R, E, A]` is a type constructor that takes three type parameters, `R`, `E`, and `A` to create a new type.

You can think of type constructors as "blueprints" for creating types. A type constructor knows how to create specific types given the appropriate input types.

For example, we could substitute `Any` for `R`, `Nothing` for `E`, and `Int` for `A` to obtain a `ZIO[Any, Nothing, Int]` which is now a type that we can create values of, for example `ZIO.succeed(0)`.

Some classes or methods may actually expect inputs that are type constructors rather than types.

For instance, we might define a `ChunkLike` trait to describe collection types that are similar to the `Chunk` data type from ZIO but have different underlying representations.

```
import zio._

trait ChunkLike[Collection[+_]] {
  def toChunk[A](collection: Collection[A]): Chunk[A]
  def fromChunk[A](chunk: Chunk[A]): Collection[A]
}
```

Here `ChunkLike` describes some collection `Collection` that can be converted to and from a `Chunk`. For example, we can convert a `List[A]` into a `Chunk[A]` and back.

```
object ListIsChunkLike extends ChunkLike[List] {
  def toChunk[A](list: List[A]): Chunk[A] =
    Chunk.fromIterable(list)
  def fromChunk[A](chunk: Chunk[A]): List[A] =
    chunk.toList
}
```

The important thing to notice here is that `ChunkLike` is parameterized on a type constructor, in this case `List`, versus a type like `List[A]`. We express this in Scala by putting brackets after the name of the type constructor with an `_` for each type argument the type constructor takes.

It is important to parameterize `ChunkLike` on a type constructor here instead of a type because the property of `ChunkLike` should apply to collections with any element type, not collections with some specific element type. If `List` is

`ChunkLike` then we should be able to convert a `List[Int]` to a `Chunk[Int]` and a `List[String]` to a `Chunk[String]`.

So being `ChunkLike` is not really a property of a `List[Int]` or a `List[String]` but of the `List` type constructor itself, which Scala lets us express by parameterizing `ChunkLike` on a type constructor instead of a type.

These are typically more advanced use cases and tend to be used as little as possible in ZIO because the use of higher kinded types like this can negatively impact user ergonomics and type inference.

However, it is good to have some sense of what these are if you run across them and how parameterized types fit into the rest of Scala's type system.

48.7 Conclusion

With the materials in this chapter you should have a solid understanding of the basics of Scala's type system, in particular the concept of subtyping, `Any` and `Nothing` as "top" and "bottom" types, and sum and product types.

These come up extensively in ZIO, for example in using `Nothing` to indicate that an effect cannot fail, using `Any` to indicate that an effect does not require any environment, or reasoning about the possible states represented by a sum type to refactor `ZIO[R, E, Option[A]]` to `ZIO[R, Option[E], A]` to improve performance.

With the content in this chapter you should have the tools to understand these patterns in ZIO and more and start to use them in your own code base.

Chapter 49

Appendix: Mastering Variance

One of the keys to ZIO's ergonomics and excellent type inference is its pervasive use of *declaration site variance*. Scala's support for declaration site variance distinguishes it from many other programming languages and allows us to write code that is easier to use and "just does the right thing" most of the time.

But using declaration site variance also requires understanding some new concepts and using some additional notation. Otherwise we can be faced with error messages that can be difficult to understand, potentially leading us to give up on variance entirely!

Understanding variance is useful if you spend more time working with ZIO and libraries in the ZIO ecosystem because you will understand why types have the variance they do and why methods have certain type bounds. Developers who understand variance also often find it is helpful in writing their own code, even in areas that don't involve functional effects.

This appendix will give you the tools you need to master variance. We will talk about what variance is, covariant / contravariant / invariant types, and how and when to use them, taking a practical and accessible approach throughout.

49.1 Definition of Variance

In practical terms, we can think of *variance* as describing how the subtyping relationship for a parameterized type relates to the subtyping relationship for the type on which it is parameterized.

For example, let's define traits for a cat and an animal.

```
trait Animal {
  val name: String
}

final case class Cat(name: String) extends Animal
```

`Cat` extends `Animal`, indicating that `Cat` is a subtype of `Animal`. Everything that is a `Cat` is also an `Animal`, but not everything that is an `Animal` is a `Cat` (it might be a `Dog` for example).

We can verify that `Cat` is a subtype of `Animal` using the Scala compiler with the `<:<` operator.

```
implicitly[Cat <:< Animal]
```

So we know that `Cat` is a subtype of `Animal`. But what if we now have a collection of cats?

```
final case class Collection[A](elements: List[A])
```

Is a `Collection[Cat]` a subtype of a `Collection[Animal]`? Variance answers this question for us!

By default, parameterized types like `Collection` are invariant. This means that the subtyping relationship for the type on which it is parameterized has nothing to do with the subtyping relationship for the type itself.

We can verify this with the Scala compiler using the same trick as above:

```
implicitly[Collection[Cat] <:< Collection[Animal]]
// does not compile

implicitly[Collection[Animal] <:< Collection[Cat]]
// does not compile
```

As far as the Scala compiler is concerned, a collection of animals and a collection of cats are two completely unrelated things, just like an `Int` and a `String` are unrelated.

While invariance is the default, most of the time it does not reflect the domain we are trying to model. And it can have significant costs to user ergonomics and type inference.

For example, let's say we want to define a method to combine two collections, the same way we might concatenate two lists. Without declaration site variance it would look like this:

```
def combine[A](
  left: Collection[A],
  right: Collection[A]
): Collection[A] =
  Collection(left.elements ::: right.elements)
```

Now let's say we want to combine a collection of cats with a collection of dogs.

The resulting collection will contain both cats and dogs so the only thing we will know about the resulting collection is that it contains animals, but that is still a perfectly sensible thing to do. In particular we will still know that every element of the resulting collection has a name, since `name` is a field that is defined on `Animal`.

But what happens when we go do this?

```
final case class Dog(name: String) extends Animal

val cats: Collection[Cat] =
  Collection(List(Cat("spots"), Cat("mittens")))

val dogs: Collection[Dog] =
  Collection(List(Dog("fido"), Dog("rover")))

combine(cats, dogs)
// does not compile
```

This code doesn't compile! `combine` is a method that is supposed to take two collections of the same type but here the collections are of different types.

And since a `Collection[Animal]` is completely unrelated to a `Collection[Cat]` or `Collection[Dog]` the compiler can't unify them.

To get around this you will often see libraries that do not use declaration site variance implementing their own helper methods.

```
def widen[A, B >: A](collection: Collection[A]): Collection[B] =
  Collection(collection.elements)

combine(widen[Cat, Animal](cats), widen[Dog, Animal](dogs))
```

But this is a recipe for user pain and frustration.

So what went wrong here? Fundamentally, we knew something about our domain that we didn't express in the code we wrote.

In reality, a collection of cats *is* a collection of animals. If I have a collection of cats I can do anything with it that I could do with a collection of animals, like printing all of their names.

But we didn't tell the compiler this, making lives harder for our users and depriving ourselves of some fundamental information about these types.

So how can we use declaration site variance to change this?

49.2 Covariance

The first kind of variance we can express is *covariance*. This means that the subtyping relationship for the parameterized type is *the same* as the subtyping relationship for the type it is parameterized on.

In Scala, we express covariance by using a `+` before the type parameter in the declaration of a class or trait. For example:

```
final case class Collection[+A](elements: List[A])
```

This indicates that if `A` is a subtype of `B`, then `Collection[A]` is a subtype of `Collection[B]`. We can again verify this with the Scala compiler:

```
implicitly[Collection[Cat] <:< Collection[Animal]]
```

A good way to remember the meaning of covariance is from its prefix "co", indicating that the variance of the parameterized type and the type it is parameterized on move in the same direction.

Conceptually, parameterized types that are covariant in a type parameter `A` are types that *contain* or *produce* values of type `A`. Because they generate `A` values, we could always take the `A` values they generate and then widen those `A` values to some supertype `B` to generate `B` values, so it is always safe to widen a covariant type parameter.

Let's look at a few examples from ZIO and the Scala standard library to get a feel for covariant data types.

Most of the immutable data types in Scala's collection library are covariant just like the toy `Collection` data type we created above. For example, list is declared as covariant `List[+A]`.

This reflects the same natural relationship we discussed above where a `List[Cat]` is a `List[Animal]`. We can prove to ourselves that this is sound by using the `map` function, which lets us transform each element of a `List` with a function.

```
lazy val cats: List[Cat] =
  ???

lazy val animals: List[Animal] =
  cats.map(identity)
```

The `map` function says we can transform any `List[A]` into a `List[B]` by providing a function `A => B` to transform `A` values into `B` values. And since a `Cat` *is* an animal, we can transform a `Cat` into an `Animal` by simply mapping with the identity function, which returns the same value unchanged.

So using the `map` function we can always transform any `List[A]` into a `List[B]` when `A` is a subtype of `B`. And so we can safely treat any `List[A]` as a `List[B]`.

Types like `List`, `Vector`, `Option`, and `Either` from the Scala standard library are all examples of data types that contain `A` values. There are also similar examples from ZIO such as `Chunk`, `Exit`, and `Cause`.

The other important type of covariant data types are one that may not *contain* an `A` value but have the ability to *produce* `A` values. Examples of this would include `ZIO` itself with respect to its value and error types, `ZStream` with respect to its value and error types, and `Gen` from ZIO Test.

Although these data types may not contain an `A` value right now, and may never produce an `A` value, if they do produce one or more `A` values we could always treat those values as `B` values if `B` is a subtype of `A`. So again we can always safely widen these covariant types.

For example, if we have a generator of `Left` values we could always treat this as a generator of `Either` values, since a `Left` is an `Either`, so every value generated would be a valid result.

To summarize, data types that are covariant with respect to a type parameter `A` are *producers* of `A` values in some sense, either because they contain existing `A` values or have the ability to generate `A` values.

We have seen how using covariance gives us additional power to model our domains. But it also carries with it some restrictions.

A data type that is covariant with respect to a type parameter `A` can only *produce* `A` values and can never *consume* them. This is necessary to preserve the soundness of being able to widen covariant types.

For example, let's go back to the generators from ZIO Test we discussed above.

If I have a `Gen[Random, Left[String, Nothing]]` it is sound to treat it as a `Gen[Random, Either[String, Int]]` because every value which it generates will be a `Left[String, Nothing]` which is a subtype of `Either[String, Int]`.

But let's say that we defined `Gen` to have an additional method that told us whether a given value was in the domain of values that generator could sample. We might try to define it like this:

```
import zio._

trait Gen[-R, +A] {
  def sample: ZIO[R, Nothing, A]
  def contains(a: A): ZIO[R, Nothing, Boolean]
}
// Does not compile
```

The compiler will complain that covariant type parameter `A` appears in contravariant position in `contains`. What does that mean other than that we should give up on this variance thing entirely and go home?

Well go back to that idea of being able to safely widen a covariant type. The `A` being covariant in `Gen` means we should always be able to treat a `Gen[R, A]` as a `Gen[R, B]` if `A` is a subtype of `B`.

But is that still true with our new signature?

A `Gen[Random, Left[Int, Nothing]]` has a `contains` method with the following signature:

```
import zio._
import zio.random.Random

def contains(
  a: Left[Int, Nothing]
): ZIO[Random, Nothing, Boolean] =
  ???
```

In other words, it knows how to take a `Left[Int, Nothing]` and tell us whether that value exists in the domain it is sampling.

But a `Gen[Random, Either[Int, String]]` has to have a `contains` method with this signature:

```
def contains(
  a: Either[Int, String]
): ZIO[Random, Nothing, Boolean] =
  ???
```

This `contains` method has to be able to take any `Either` value, whether it is a `Left` or a `Right` and tell us whether it is in the domain.

If we treated a `Gen[Random, Left[Int, Nothing]]` as a `Gen[Random, Either[Int, String]]` we could create a runtime exception because we could then try to pass in a `Right` to `contains` when the actual `contains` method we have is only defined on `Left` values.

The Scala compiler is smart enough to stop us from doing this so it will prevent us from declaring a type parameter as covariant if it appears as an *input* to any of the methods on the data type.

So how do we get around this? Being able to test whether a value is within the domain of a generator seems like something we at least conceptually might want to be able to do.

The answer is by using type bounds. For covariant types `A` can't appear as an input, but `A1 >: A` can. So we could rewrite the interface for our `Gen` example as:

```
trait Gen[-R, +A] {
  def sample: ZIO[R, Nothing, A]
  def contains[A1 >: A](a: A1): ZIO[R, Nothing, Boolean]
}
```

This now says our `contains` method has to be able to handle any value that is a supertype of `A`, which could potentially be any value at all. So we have to have some way of handling arbitrary values, in this case potentially by returning `false` indicating that they don't appear in the domain.

We can use the same technique to solve the problem that originally motivated our discussion of variance involving combining two collections. Let's go back to the covariant implementation of our collection type.

```
final case class Collection[+A](elements: List[A]) { self =>
  def ++[A1 >: A](that: Collection[A1]): Collection[A1] =
    Collection(self.elements ::: that.elements)
}
```

Again, we need to use `A1 >: A` here because `A` is covariant and so can only appear as an *output* and not an *input*. But this also reflects something very logical that is actually the solution to our problem from earlier.

This signature is saying that when we have a collection of `A` values, we don't have to just combine them with other `A` values. We can also combine them with values of some more general type, but then we will get back a collection of values of the more general type.

So in our example above if we start with a collection of cats we don't have to just combine them with other cats. We can also combine them with other animals, but if we do we will get back a collection of animals instead of a collection of cats.

And the compiler will always give us the most specific type here. If we do combine a collection of cats with another collection of cats we will get back a collection of cats and this will happen automatically without us having to do any type annotations.

In summary, covariance lets us more accurately model our domain when we have a data type that contains or produces values of some other type. This leads to improved type inference and ergonomics for our users.

Over time it can also help us to reason about our data types. Methods that have or produce values of type `A` typically have a `map` method for transformating those values, for example, and potentially a `zipWith` combinator for combining different values or a `flatMap` method for chaining them.

As you get more experience working with variance you will notice these patterns with your data types and it can give you ideas for useful combinators.

But covariance is only one of the fundamental types of variance.

We talked before about how covariant types *produce* `A` values and never *consume* them. What would types that *consume* `A` values and never *produce* them look like?

49.3 Contravariance

The second type of variance is *contravariance*. Contravariance means that the subtyping relationship for a parameterized type is the *opposite* of the subtyping relationship for the type it is parameterized on.

We express that a type parameter is contravariant in Scala by using `-` before the declaration of the type parameter. For example:

```
trait Fruit

trait Apple extends Fruit
trait Orange extends Fruit

trait Drink

trait FoodProcessor[-Ingredient] {
  def process(ingredient: Ingredient): Drink
}
```

We often have a reasonably good intuition for covariance because we are familiar with types like collections. In addition, covariance can seem "natural" because the subtyping relationship is the same for the paramaterized type and the type it is parameterized on.

What does a contravariant type mean?

A good intuition is that a data type that is contravariant in a type parameter `A` knows how to *consume* `A` values to produce some value or effect.

For example, the `FoodProcesser` above knows how to consume ingredients and produce delicious drinks. Let's see how this would look:

```
trait Smoothie extends Drink
trait OrangeJuice extends Drink

val juicer: FoodProcessor[Orange] =
  new FoodProcessor[Orange] {
    def process(orange: Orange): Drink =
      new OrangeJuice {}
  }

val blender: FoodProcessor[Fruit] =
  new FoodProcessor[Fruit] {
    def process(fruit: Fruit): Drink =
      new Smoothie {}
  }
```

Here we have two different food processors.

The `juicer` lets us squeeze juice out of oranges to make orange juice. But it doesn't work on any other types of fruit.

The `blender` just blends whatever fruit we give it to make a smoothie, so it can work on any type of fruit.

We said before that `Orange` was a subtype of fruit, which we can verify by doing:

```
implicitly[Orange <:< Fruit]
```

But a `FoodProcessor[Fruit]` is actually a subtype of `FoodProcessor[Orange]`:

```
implicitly[FoodProcessor[Fruit] <:< FoodProcessor[Orange]]
```

What does this mean?

Recall that `A` being a subtype of `B` means that we should be able to use an `A` any time that we need a `B`. For example, if we need to eat a fruit to comply with a doctor's recommended diet, an orange will certainly qualify as a kind of fruit.

So we should be able to use a `FoodProcessor[Fruit]` any time we need a `FoodProcessor[Orange]`.

And in fact that is right. Any time we need to process some oranges to make a drink, we can always use a `FoodProcessor[Fruit]` like the blender, because that can make a drink out of any kind of fruit including an orange.

Stepping back, we said before that contravariant types *consume* `A` values. If `B` is a subtype of `A` then we can always treat a `B` value as an `A` value, since a `B` *is* an `A`.

So if we have a contravariant type that can *consume* `A` values it can also *consume* `B` values, and so it is safe to treat trait a `Consumer[A]` as a `Consumer[B]`.

Here are some other examples of contravariant types from the Scala standard library and ZIO:

- A function `A => B` is contravariant in the type `A` because it consumes `A` values to produce `B` values.
- A `ZIO` effect is contravariant with respect to its environment type `R` because it needs an `R` input and produces either an `E` or an `A`.
- A `ZSink` is contravariant with respect to its input type `I` because it consumes `I` values to produce some output.

Just like with covariance, using contravariance dramatically improves ergonomics and type inference.

We can see this in action with ZIO's environment type.

Recall that the signature of `ZIO` is `ZIO[-R, +E, +A]`, indicating that it is an effect that *requires* an environment of type `R` and *produces* either a failure of type `E` or a value of type `A`.

When working with ZIO we often want to combine effects that require different environments.

```
import zio.clock._
import zio.console._

val nanoTime: URIO[Clock, Long] =
  clock.nanoTime

def printLine(line: String): URIO[Console, Unit] =
  console.putStrLn(line)

val printTime: URIO[Clock with Console, Unit] =
  nanoTime.map(_.toString).flatMap(printLine)
```

Because of contravariance these effects compose extremely naturally. `nanoTime` requires a `Clock` service and `printLine` requires a `Console` service so if we want to call both `nanoTime` and `printLine` we need both a `Clock` and a `Console` service.

But this wouldn't have worked at all without contravariance. Without contravariance the Scala compiler would see `nanoTime` and `printLine` as completely unrelated types, just like the invariant collections we discussed earlier, and would not allow us to combine them.

Libraries that do not use contravariance often need to define methods like `narrow` analogous to the `widen` method we saw above to make code like this compile, but once again that is a recipe for user pain and frustration.

Just like with covariance, contravariance has significant benefits for ergonomics and type inference but also places some restrictions on how we do things to make sure the code we write is sound.

Specifically, a contravariant type parameter must always be an input and never be an output.

Just like this covariant types above, this is to prevent us from doing things that could ultimately violate Scala's type system.

For example, suppose that our food processor discussed above also produced some leftover fruit in addition to a drink:

```
trait FoodProcessor[-Ingredient] {
  def process(ingredient: Ingredient): (Drink, Ingredient)
}
// does not compile
```

Now `Ingredient` appears not just as an input but as an output. What happens now if we try to treat a `FoodProcessor[Fruit]` as a `FoodProcessor[Orange]`?

We said above that we should be able to use a `FoodProcessor[Fruit]` anywhere we need a `FoodProcessor[Orange]`.

But if we use a `FoodProcessor[Orange]` we're guaranteed to get an orange back. If we use a `FoodProcessor[Fruit]` we're no longer guaranteed to get an orange back.

We might have put apples and oranges into the blender and the leftovers might have both apples and oranges, or maybe only apples. So it is actually not safe for us to use a `FoodProcessor[Fruit]` anywhere we would have used a `FoodProcessor[Orange]` anymore!

Once again, the Scala compiler will stop us from getting to this point by warning us when we try to define `FoodProcessor` that the contravariant type `Ingredient` appears in covariant position in `process` because it appears as an output.

To get around this we can use the same trick as we learned for covariant types but with the arrow in the other direction.

So to go back to ZIO, when we define `flatMap` it has the following signature:

```
trait ZIO[-R, +E, +A] {
  def flatMap[R1 <: R, E1 >: E, B](
    f: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ???
}
```

We can't use `R` directly here because it would be appearing in covariant position. But we can define a new type `R1 <: R` and use that.

This also has a very natural interpretation.

If we think about `R` as the requirements of an effect, `R1 <: R` means that `R1` has more requirements than `R` does. For example, `R1` might be `Clock with Console` whereas `R` is only `Clock`.

So what this signature is saying is that if we have one effect that requires some services and another effect that requires additional services, the combined effect will require those additional services. This is the same thing we saw above.

49.4 Invariance

The final type of variance is invariance, which we identified above as the default when we do not declare the variance of a type parameter.

Invariant type parameters generally have worse ergonomics and type inference than type parameters that use declaration site variance. So they should be avoided when possible.

However, sometimes types we are working with have to be invariant because a type parameter appears as both an input and an output.

For example, consider the `Ref` data type from ZIO.

```
trait Ref[A] {
  def get: UIO[A]
  def set(a: A): UIO[Unit]
}
```

Just from the definition of `get` and `set` we can tell that `A` will have to be invariant.

`A` appears as an output in `get`, indicating that we can always get an `A` value out of the `Ref`. `A` appears as an input in `set`, indicating that we can also always put a new `A` value into the `Ref`.

Since `A` appears in covariant position in `get` and contravariant position in `set`, we will not be able to declare `A` as either covariant or contravariant and will have to default to invariance.

This isn't some arbitrary rule of the Scala compiler either but reflects something fundamental about this data type and the way we have defined it.

Say we have defined `Cat` as a subtype of `Animal` and have a `Ref[Cat]`. It is not safe for us to treat a `Ref[Cat]` as a `Ref[Animal]`.

If we had a `Ref[Animal]` we could set the value to be any `Animal`, but if the actual `Ref` we are working with is a `Ref[Cat]` that could result in a `ClassCastException` because we are trying to set a `Dog` in a value that is expecting a `Cat`.

Likewise, if we have a `Ref[Animal]` it is not safe to treat it as a `Ref[Cat]`. A `Ref[Cat]` guarantees us that whatever value we get out with `get` will be a `Cat`. But if the actual `Ref` is a `Ref[Animal]` the value we get could be any animal at all, resulting in a `ClassCastException` when we go to use that value in a method expecting a `Cat`.

So the invariance of `Ref` is not some arbitrary thing but is actually telling us something fairly fundamental about this data type, that it both accepts `A` values as inputs and provides them as outputs.

Other examples of invariant types in ZIO are queues and promises. We can both offer values and take values from a queue, and we can both complete and await a promise.

The lesson here isn't that you should never use invariant types. However, you should think about the variance of your data types and only declare them to be invariant if that accurately reflects the domain you are modeling.

One trick we can sometimes use to recover variance for invariant data types is creating separate type parameters for the covariant and contravariant positions

in the data type.

For example, the reason that `Ref` had to be invariant is that `A` appeared in covariant position in `get` but contravariant position in `set`. But what if we broke these out into two separate type parameters?

```
trait ZRef[-A, +B] {
  def get: UIO[B]
  def set(a: A): UIO[Unit]
}
```

Now `A` represents the type of values that can be set in the `ZRef` and appears only in contravariant position. `B` represents the type of values that can be gotten out of the `ZRef` and appears only in covariant position.

Conceptually, a `ZRef` has some function "inside" it that allows it to transform `A` values into `B` values so that what we get is related to what we set. We can imagine a `Ref` that allows use to `set` a first name and last name and `get` the first name back.

We can recover the original invariant version with a simple type alias.

```
type Ref[A] = ZRef[A, A]
```

Splitting out covariant and contravariant types typically allows for defining more operators on a data type.

For example we can define a `map` operator on `ZRef` that allows us to transform the output type and a `contramap` operator that allows us to transform the input type. Neither of these operators could be defined for the monomorphic `Ref`.

However, splitting out type parameters in this way does add some additional complication so you should think about whether the additional operators will be worthwhile.

In ZIO itself many invariant types are aliases for more polymorphic types with separate covariant and contravariant type parameters. This includes `Ref` and `ZRef`, `RefM` and `ZRefM`,`Queue` and `ZQueue`, and `TRef` and `ZTRef`.

You can see here that this is a convention within the ZIO ecosystem of using the `Z` prefix for the more polymorphic versions of data types!

This can be a helpful pattern for preserving polymorphism for users who want additional functionality while preserving simple type signatures for other users without code duplication.

For example, many users just want to work with a `Ref` that allows them to get, set, and update values of the same type to manage concurrent state. By defining `Ref` as a type alias for `ZRef`, those users don't have to know anything about polymorphic references even though a `Ref` *is* a `ZRef`.

ZIO also includes some data types, such as `Promise`, that are invariant and are not type aliases for more polymorphic versions because the additional polymorphism

was not judged worth the complexity. So do what makes sense for you regarding breaking out separate type parameters for invariant data types.

49.5 Advanced Variance

One other topic that can come up when working with variance is analyzing the variance of more complex nested data types.

We understand the basic principle that covariant types must appear only as outputs and contravariant types must appear only as inputs.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])
```

Here `R` appears as only an input in the definition of `run` and `E` and `A` appear only as outputs.

But what about more complicated situations? For example, we ran into this issue earlier when looking at the signature of the `flatMap` operator on ZIO.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {
  def flatMap[B](f: A => ZIO[R, E, B]): ZIO[R, E, B] =
    ???
}
```

Is this code okay as written? Or do we need to add additional type parameters like `R1 <: R` and `E1 >: E` to avoid variance issues?

To answer this question, other than by letting the Scala compiler tell us, we need determine whether `R`, `E`, and `A` appear in covariant or contravariant position in the definition of `flatMap`.

It may not seem entirely clear. `A` appears in the signature of `f` which is an input, but it isn't actually `A` that is an input but this function that itself takes `A` as an input.

Likewise `R` and `E` appear in the input `f` but now they are outputs of this function and themselves appear as parameters of a `ZIO` effect.

There are a couple of simple rules we can apply to analyze these situations:

- Think of a type parameter appearing in covariant position as a `+1`, a type parameter appearing in contravariant position as a `-1`, and a type parameter appearing in invariant postion as `0`.
- Multiply the values for each layer of a nested type together to determine what position each type parameter is in

Let's work through determining whether each of the type parameters appears in covariant position, contravariant position, or both.

We will start with the `R` parameter.

1. `R` appears in the function `A => ZIO[R, E, B]` which is an input to this data type, so we will mark `-1` for that for contravariant position.
2. `R` appears on the right hand side of `A => ZIO[R, E, B]`. A function `In => Out` is itself a type signature for `Function[-In, +Out]`, so since `R` appears on the right hand side it is in covariant position there and we mark `+1`.
3. `R` appears in the data type `ZIO[R, E, A]`. The signature of `ZIO` is `ZIO[-R, +E, +A]` so `R` appears in contravariant position there and we mark another `-1`.
4. Putting these all together we have `-1`, `+1`, and `-1`, which multiplied together is `+1`. So we conclude that `R` appears in covariant position here.

Another way to think about this is that when a type appears in covariant position we keep the same variance and when it appears in contravariant position we "flip" the variance. When it appears in invariant position it is always invariant.

Since `R` is defined as contravariant in `ZIO` but is appearing in covariant position in the argument `f` in `flatMap`, we are going to get a compilation error if we don't define a new type `R1` that is a subtype of `R`. So far we have this:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {
  def flatMap[R1 <: R, B](f: A => ZIO[R1, E, B]): ZIO[R, E, B] =
    ???
}
```

`R` also appears in the return type of `flatMap` and here it is in ontravariant position because the output is covariant and `R` appears in contravariant position in the environment type of `ZIO`, so combining covariant and contravariant gives us contravariant.

Since `R` is defined as being contravariant and appears in contravariant position in the return type of `flatMap` we don't have to make any changes here, at least from a variance perspective.

Let's apply the same logic to the other type parameters.

The `E` type parameter appears in two places, as part of the function `f` and as part of the return type of the function. We will need to analyze both of them.

We will evaluate the variance of the `E` in the `A => ZIO[R, E, B]` first:

1. `E` appears in the function `A => ZIO[R, E, B]` which is an input to this data type, so we mark `-1` for contravariance
2. `E` appears in the output of the function `A => ZIO[R, E, B]` so we mark `+1` for covariant position there
3. `E` appears in the error type of the `ZIO[R, E, A]`, which is covariant, so we mark another `+1` there.
4. Multiplying each of those values together we have `-1` times `+1` times `+1` which equals `-1`, so we conclude that `E` appears in contravariant position.

Since `E` is defined as being covariant but is appearing in contravariant position in the argument `f` to `flatMap`, we again need to introduce a new type `E1` that is

a supertype of `E`. So we now have:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {
  def flatMap[R1 <: R, E1 >: E, B](
    f: A => ZIO[R1, E1, B]
  ): ZIO[R, E, B] =
    ???
}
```

`E` also appears in the return type of `flatMap` and here it is in covariant position because the output is covariant and `E` appears in covariant position in the error type of `ZIO`, so combining contravariant and covariant gives us contravariant.

Since `E` is defined as being covariant and is in covariant position in the return type of `flatMap` we don't have to make any changes here.

Finally we can analyze `A`:

1. `A` appears in the input function `A => ZIO[R, E, B]` which is an input to `flatMap` so we mark `-1` for contravariant position.
2. `A` appears in the input of the function `A => ZIO[R, E, B]` which is also contravariant, so we mark another `-1`
3. Combining those we have `-1` times `-1` which is `+1` so `A` is in covariant position here.

Putting this all together `A` only appears in covariant position so we don't need to do anything special with it. So based purely on variance we have the following signature:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {
  def flatMap[R1 <: R, E1 >: E, B](
    f: A => ZIO[R1, E1, B]
  ): ZIO[R, E, B] =
    ???
}
```

This is indeed correct purely from a variance perspective and the snippet above will compile as written. However, when we try to go to actually implement `flatMap` we run into a problem.

Here is how we could implement `flatMap` in terms of this toy `ZIO` data type:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def flatMap[R1 <: R, E1 >: E, B](
    f: A => ZIO[R1, E1, B]
  ): ZIO[R, E, B] =
    ZIO(r => self.run(r).flatMap(a => f(a).run(r)))
}
```

When we do this, we get a compilation error. The compilation error tells us that we returned a value of type `ZIO[R1, E1, B]` but our method expected a

`ZIO[R, E, B]`.

Conceptually, the effect returned by `f` may require an environment with more capabilities than the environment required by the initial effect. And similarly, the effect returned by `f` may fail in more ways than the original effect could fail.

So if `flatMap` represents performing this effect and then performing another effect based on its result, performing both effects is going to require all the capabilities required by both of them and be able to fail in any the ways that either of them could fail.

For this reason, to make this code compile our final type signature needs to be:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def flatMap[R1 <: R, E1 >: E, B](
    f: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ZIO(r => self.run(r).flatMap(a => f(a).run(r)))
}
```

Again, there was nothing about variance that purely from the method signature required this. But to actually implement the method we needed to propagate these type parameters to the output type as well.

This reflects a very common pattern where in working with variance we will often need to introduce some new type parameters to satisfy variance, and then have to update some of the other types in our method signatures to reflect the way data actually flows in our program.

This also explains why we define these new type parameters as subtypes or supertypes of the existing type parameters.

Purely from the perspective of variance we don't need to do this. For example, we could define the `flatMap` method like so:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def flatMap[R1, E1, B](
    f: A => ZIO[R with R1, Any, B]
  ) =
    ZIO(r => self.run(r).flatMap(a => f(a).run(r)))
}
```

Notice that in the above `R1` and `E1` are completely unrelated to `R` and `E`, respectively.

This method signature satisfies variance, because no covariant type parameters appear in contravariant position and no contravariant type parameters appear in covariant position. But look at the return type!

For the environment type the Scala compiler can use an intersection type to describe the return type as `R with R1`, which accurately models that the resulting effect needs an environment with both the capabilities of `R` and `R1`. However,

for the error type Scala does not have union types, at least in Scala 2, so the only type it can infer for the error here is `Any`.

We clearly don't want to lose all type information about how this effect can fail like we do here, so we add constraints that `R1` is a subtype of `R` and `E1` is a supertype of `E` to allow the Scala compiler to `unify` the environment and error types of the two effects to a common type.

You don't have to understand variance at this level of detail to be extremely productive with ZIO and libraries in the ZIO ecosystem, but hopefully the material in this section has given you a better understanding of the basis for the more concrete guidance for working with covariant, contravariant, and invariant types from earlier in this chapter.

49.6 Conclusion

After having completed this chapter you should understand what variance is and how it lets us write code that better reflects the domains we are modeling. You should understand the difference between contravariance, covariance, and invariance as well as how to read variance annotations, understand the natural variance of a data type, and use declaration site variance in your own data types.

Variance can be a new concept so it may take a bit of time to be comfortable using it. However, if you do you will find you can create APIs that are much more ergonomic for your users and infer flawlessly in most cases.

As you become more comfortable with variance you will also see that it gives you a new way to see at a glance the "flow" of information within your program based on what types appear as inputs and outputs to each of your data types. This can let you very easily see how different data types can be combined together, for example feeding the output from one data type to the input of another or combining the outputs of two data types.