

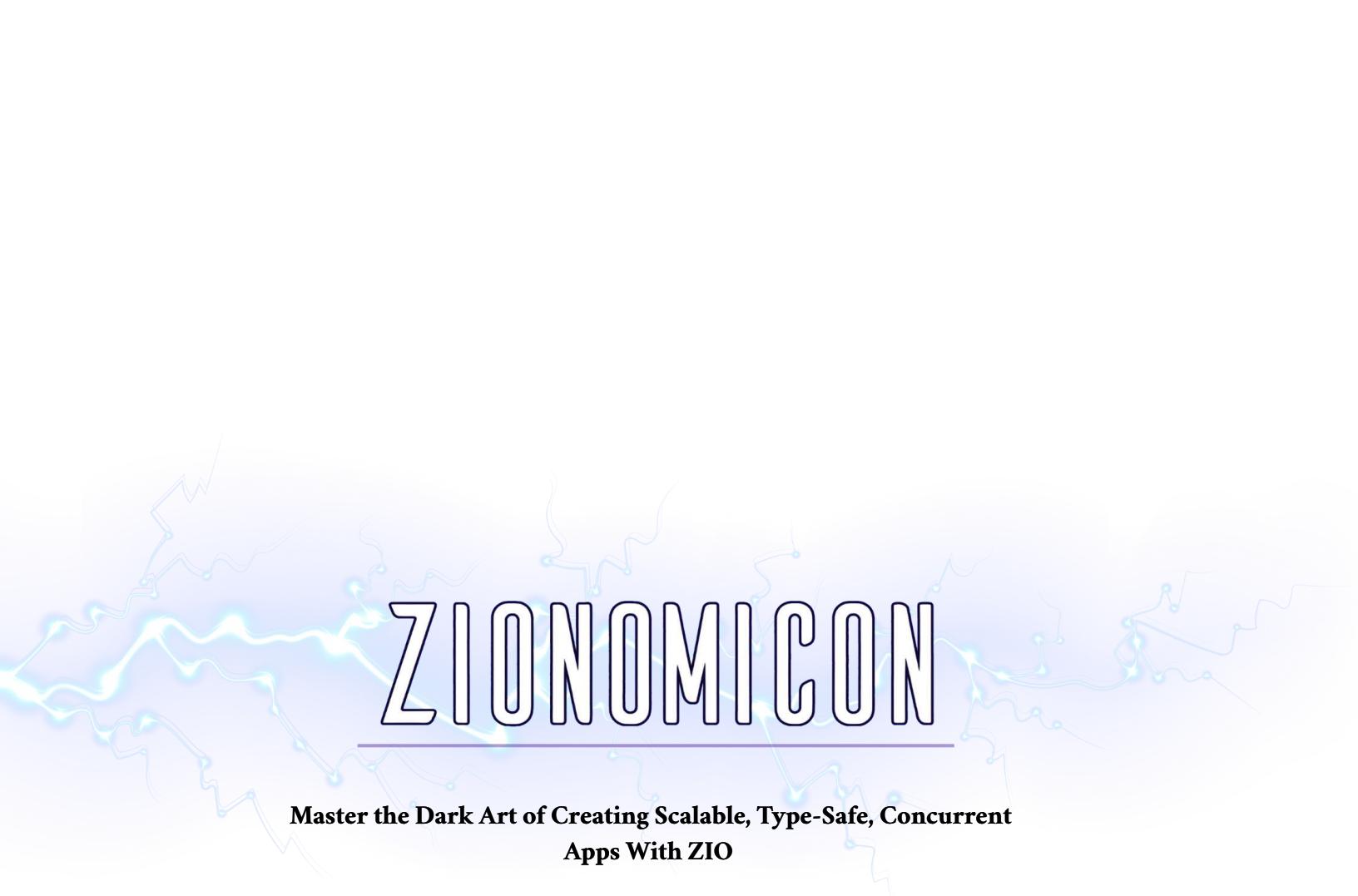
# ZIONOMICON

Master the dark art of creating scalable, type-safe,  
concurrent apps with ZIO



Updated for  
ZIO 2.1!

JOHN A. DE GOES,  
ADAM FRASER, MILAD KHAJAVI



# ZIONOMICON

---

**Master the Dark Art of Creating Scalable, Type-Safe, Concurrent  
Apps With ZIO**

John De Goes, Adam Fraser, and Milad Khajavi

January 28, 2025

# Zionomicon

**Master the Dark Art of Creating Scalable, Type-Safe, Concurrent Apps With ZIO**

By John De Goes, Adam Fraser, and Milad Khajavi

Copyright © 2024-2025 Ziverge, Inc. All rights reserved.

Published by Ziverge, Inc.

Revision History:

2025-01-22: Current Revision

2024-11-20: First Release

Although Ziverge and the authors have taken all reasonable care in preparing this book, we make no representation about the accuracy, completeness, or suitability of the information contained in this book. The content is provided "as is" without any warranties of any kind. Ziverge and the authors are not liable for any errors or omissions, or for any losses, damages, or other liabilities that may arise from using the information contained in this book. Readers should use the information at their own discretion and risk.

Any code examples provided in this book may be subject to various open source licenses or third-party intellectual property rights. It is the reader's responsibility to review and comply with all applicable licenses and rights before using or adapting any code from this book.



## Excellence powered by **ZIVERGE**

As a proud sponsor of **Zionomicon**, Ziverge leads the industry in ZIO and Functional Scala expertise. Our team of seasoned professionals delivers exceptional solutions through:

### **Enterprise Solutions**

- Custom ZIO applications and microservices
- Scala-based data processing pipelines
- Apache Spark optimizations and extensions
- Production-ready functional architectures

### **Training & Education**

- Hands-on ZIO workshops
- Functional Scala bootcamps
- Data engineering best practices
- Custom training programs

### **Professional Services**

- Architecture consulting and technical leadership
- Team augmentation and mentoring
- Performance optimization
- Legacy system modernization

### **Why Choose Ziverge?**

- Deep expertise in ZIO ecosystem
- Track record of enterprise implementations
- Industry-leading functional programming talent
- End-to-end project support

**Ready to elevate your ZIO journey?**

*Contact our team to discuss how we can help you achieve your technology goals.*

[Schedule a Consultation](#) | [Learn More About Our Services](#)

Ziverge - Your Partner in Functional Excellence



## Excellence powered by **ZIVERGE**

Our team would also like to take a moment to thank our patrons for making this version of Zionomicon possible.

**Here are just some of the names we'd love to share our gratitude with:**

A.J. Whitney  
Adam Maklér  
Adrian Filip  
Alan Artigao Carreño  
Alberto Maria Angelo Paro  
Alejandro Merchan  
Alejandro Triana  
Alessandro Candolini  
Alex Dupre  
Alex Pavlychev  
Alex Thom  
Alexander Ryblov  
Alexander Ushakov  
Alexandru Cuciureanu  
Americo Vargas  
Anargyros Akrivos  
André Uratsuka Manoel  
Andrea Passaglia  
Andrea Zubenko  
Andrew Johnson  
Andy Wang  
Angel M Vanegas IV  
Anjali Belani  
Anton Bossenbroek  
Antonio Davide Cali  
Aris Vlasakakis  
Arnold Lacko

Artem Andreev  
Ashelyn Bauman  
Atul S. Khot  
Avinder Bahra  
Babis Routis  
Ben Eyal  
Boris Potepun  
Brian Birke  
Brian Schlining  
Brieuc Kaisin  
Calvin Lee Fernandes  
Cameron Eades  
Chris Kimberley  
Clint Combs  
Curtis Stanford  
Dan Lagnöhed  
Dan Sickles  
Dan Todor  
Derrick Weis  
Dmitry Vostokov  
Dmytro Altsyvanovych  
Dominik Dorn  
Dr. Eric J. Smith  
Dr. Ivan Subotic, DaSCH

Duncan Folkes  
Eelke Boezeman  
Ematiq  
Erik Innocent  
Esteban Marin  
Eugene Losev  
Federico Piola  
Filip Dreger  
Frank Sauer  
Gabriel Ciuloaica  
Gabriel Raileanu  
George Fourikis  
Gerald Loeffler  
Giulio Cesare Solaroli  
Gregor Purdy  
Gregor Rayman  
Guilherme Ceschiatti  
Guodong Qu  
Hans Christian Wilhelm  
Herwig Habenbacher  
Ibrahim Moreno  
Igor Dorokhov  
Irina Ladygina  
Jack Wells



## Excellence powered by **ZIVERGE**

Our team would also like to take a moment to thank our patrons for making this version of Zonomicon possible.

**Here are just some of the names we'd love to share our gratitude with:**

Jakub Janeček	Larry Simon	Noe Perez
James King	Laurie-Anne Parant	Noel Gallagher
Jan Nosal	Leo Vergara	Odd Möller
Jeffrey Aguilera	Leonid Dubinsky	Olaf Maurer
Jeffrey Zhang	Lucas Manzanelli	Oleg Korolenko
Jernej Vasic	Ludvig Gislason	Oleksiy Rykhalskyy
Jeroen Zuijderhoudt	Luigi Mazzon	Olivier Nouguier
Jerome Moliere	Luis Miguel Serrano	Omar Diego Vera Ustariz
Jim Weinert	Mahamed Ali	Omar Reddam
Jisoo Park	Mariusz Nosiński	Panggi Libersa Jasri Akadol
Johan Segerlund	Mark Kegel	Pär Wenåker
Jon Strayer	Markus Feire	Patrizio Dazzi
Jorge Adriano Branco Aires	Martin Mauch	Peter Storm
Jorge Vásquez	Martin Tjon	Phil Derome
Jose Velascos	Martyn Gilmore	Pietro Loffredi
Josua Riha	Matan Keidar	Radek Tkaczyk
Juan Manuel Gimeno Illa	Máté Börcsök	Raffael Dzikowski
Justin Heyes-Jones	Matthew Jones	Rajkumar Natarajan
Kevin Boyette	Mauricio Fernandes de Castro	Renan Rossignatti de França
Kevin Esler	Maxim Duester	Reto Schöning
Kirill Pavkin	Michal Pociecha	Riccardo Cardin
Konstantinos Pouliasis	Nathan Knox	Riccardo Torsoli
Kristof Slechten	Nick Gravgaard	Richard B. Opsal
Kurt Fehlhauer	Niklas von Minckwitz	Richard Capraro



## Excellence powered by **ZIVERGE**

Our team would also like to take a moment to thank our patrons for making this version of Zionomicon possible.

**Here are just some of the names we'd love to share our gratitude with:**

Richard Searle	Tanju Erinmez
Roman Arkharov	Terrence Drozdowski
Roman Belous	Tim Pigden
Ronaldo Alves de Abreu	Tobia Loschiavo
Rudolf Markulin	Tomi Kasurinen
Russ White	TravelTime
Salva Alcántara	Vadym Vasiuk
Santhos Ramalingam	Venkat Gounder
Saqib Saleem	Victor Brenister
Sean Gao	Visar Zejnullahu
Sébastien Capron	Vishal Surana
Sergei Svitkov	Vishnu Raman
Shreedhan Shrestha	Vladimir Bystrov
Sriram Natarajan	William Harvey, Ph.D.
Stefan Wachter	Wouter Lammers
Steven Scott	Yoshiki Tobita
Tanju Erinmez	Yuehang Chen
Terrence Drozdowski	Zhe Zhang
Tim Pigden	
Tobia Loschiavo	
Tomi Kasurinen	
TravelTime	

# Contents

<b>Foreword by John A. De Goes</b>	<b>1</b>
0.1 A Brief History of ZIO . . . . .	1
0.2 The Birth of ZIO . . . . .	2
0.3 Contentious Concurrency . . . . .	2
0.4 Typed Errors & Other Evolution . . . . .	3
0.5 Contributor Growth . . . . .	4
0.6 Improved Type-Inference . . . . .	4
0.7 Batteries Included . . . . .	5
0.8 ZIO Stream . . . . .	6
0.9 ZIO Environment . . . . .	6
0.10 Software Transactional Memory . . . . .	7
0.11 Execution Traces . . . . .	8
0.12 Summer 2019 . . . . .	9
0.13 ZIO Test . . . . .	9
0.14 ZLayer . . . . .	10
0.15 Structured Concurrency . . . . .	11
0.16 Preparation for ZIO 2.0 . . . . .	11
0.17 Simplification of ZIO Environment and Dependency Injection . . . . .	12
0.18 ZIO Becomes Composable Resourceful Effect . . . . .	12
0.19 Service Pattern . . . . .	13
0.20 Smart Assertions . . . . .	13
0.21 More Concrete Types . . . . .	13
0.22 Unified Streaming . . . . .	14
0.23 Regional Settings and Contextual Scopes . . . . .	14
0.24 Observability . . . . .	14
0.25 Performance . . . . .	15
0.26 ZIO 2.0 . . . . .	16
0.27 What is Next? . . . . .	16
0.28 Why ZIO . . . . .	17
0.29 ZIO Alternatives . . . . .	17
0.30 Zionomicon . . . . .	18
<b>Preface</b>	<b>20</b>

0.31 Who Should Read This Book? . . . . .	20
0.32 How This Book Is Organized . . . . .	20
0.33 How to Use This Book . . . . .	21
0.34 Zionomicon is Updated with ZIO 2.1 . . . . .	21
0.35 Acknowledgments . . . . .	22
<b>1 Essentials: First Steps With ZIO</b>	<b>23</b>
1.1 Functional Effects as Blueprints . . . . .	23
1.2 Sequential Composition . . . . .	27
1.2.1 For Comprehensions . . . . .	28
1.3 Other Sequential Operators . . . . .	29
1.4 ZIO Type Parameters . . . . .	31
1.4.1 The Error Type . . . . .	32
1.4.2 The Environment Type . . . . .	34
1.5 ZIO Type Aliases . . . . .	35
1.6 Comparison to Future . . . . .	35
1.6.1 A Future is a Running Effect . . . . .	36
1.6.2 Future has an Error Type Fixed to Throwable . . . . .	37
1.6.3 Future Does not Have a Way to Model the Dependencies of an Effect	38
1.7 More Effect Constructors . . . . .	38
1.7.1 Pure Versus Impure Values . . . . .	39
1.7.2 Effect Constructors for Pure Computations . . . . .	40
1.7.3 Effect Constructors for Side Effecting Computations . . . . .	42
1.7.3.1 Converting Async Callbacks . . . . .	42
1.8 Default ZIO Services . . . . .	44
1.8.1 Clock . . . . .	45
1.8.2 Console . . . . .	46
1.8.3 System . . . . .	46
1.8.4 Random . . . . .	47
1.9 Recursion and ZIO . . . . .	47
1.10 Conclusion . . . . .	48
1.11 Exercises . . . . .	48
<b>2 Essentials: Testing ZIO Programs</b>	<b>54</b>
2.1 Writing Simple Programs with ZIO Test . . . . .	57
2.2 Using Assertions . . . . .	59
2.3 Test Implementations of Standard ZIO Services . . . . .	61
2.4 Common Test Aspects . . . . .	63
2.5 Basic Property-based Testing . . . . .	64
2.6 Conclusion . . . . .	66
2.7 Exercises . . . . .	66
<b>3 Essentials: The ZIO Error Model</b>	<b>68</b>
3.1 Exceptions Versus Defects . . . . .	68
3.2 Cause . . . . .	70
3.3 Exit . . . . .	70

3.4	Handling Defects . . . . .	71
3.5	Converting Errors to Defects . . . . .	72
3.6	Multiple Failures . . . . .	73
3.7	Other Useful Error Operators . . . . .	75
3.8	Combining Effects with Different Errors . . . . .	75
3.9	Execution Tracing . . . . .	77
3.10	Dealing With Stacked Errors . . . . .	78
3.11	Leveraging Typed Errors . . . . .	79
3.12	Conclusion . . . . .	80
3.13	Exercises . . . . .	80
<b>4</b>	<b>Essentials: Integrating with ZIO</b>	<b>83</b>
4.1	Integrating with Java . . . . .	86
4.2	Integrating with Javascript . . . . .	89
4.3	Integrating with Cats Effect . . . . .	90
4.4	Integrating with Specific Libraries . . . . .	92
4.4.1	Integrating with Doobie . . . . .	92
4.4.2	Integrating with http4s . . . . .	96
4.5	Conclusion . . . . .	98
4.6	Exercises . . . . .	99
<b>5</b>	<b>Parallelism and Concurrency: The Fiber Model</b>	<b>100</b>
5.1	Fibers Distinguished from Operating System Threads . . . . .	100
5.2	Forking Fibers . . . . .	101
5.3	Joining Fibers . . . . .	102
5.4	Interrupting Fibers . . . . .	104
5.5	Fiber Supervision . . . . .	106
5.6	Locking Effects . . . . .	108
5.7	Conclusion . . . . .	110
5.8	Exercises . . . . .	110
<b>6</b>	<b>Parallelism and Concurrency: Concurrency Operators</b>	<b>111</b>
6.1	The Importance of Concurrency Operators . . . . .	111
6.2	Race and ZipPar . . . . .	111
6.3	Variants of ZipPar . . . . .	112
6.4	Variants of Race . . . . .	114
6.5	Validation Errors . . . . .	115
6.6	Conclusion . . . . .	116
6.7	Exercises . . . . .	117
<b>7</b>	<b>Parallelism and Concurrency: Fiber Supervision in Depth</b>	<b>119</b>
7.1	Fork/Join Identity Law . . . . .	119
7.2	Structured Concurrency . . . . .	120
7.3	Custom Supervision Strategies . . . . .	123
7.3.1	Forking in the Global Scope . . . . .	123
7.3.2	Forking in the Current Local Scope . . . . .	123

7.3.3	Forking in a Specific Scope . . . . .	127
7.4	Fire-and-Forget . . . . .	128
7.5	Conclusion . . . . .	129
<b>8</b>	<b>Parallelism and Concurrency: Interruption in Depth</b>	<b>130</b>
8.1	Timing of Interruption . . . . .	130
8.1.1	Interruption Before an Effect Begins Execution . . . . .	131
8.1.2	Interruption of Side Effecting Code . . . . .	132
8.2	Interruptible and Uninterruptible Regions . . . . .	135
8.3	Composing Interruptibility . . . . .	137
8.4	Waiting for Interruption . . . . .	140
8.5	Conclusion . . . . .	143
8.6	Exercises . . . . .	143
<b>9</b>	<b>Concurrent Structures: Ref - Shared State</b>	<b>145</b>
9.1	Purely Functional Mutable State . . . . .	145
9.2	Purely Functional Equivalent of an Atomic Reference . . . . .	148
9.3	Operations are Atomic but do not Compose Atomically . . . . .	151
9.4	Ref.Synchronized for Evaluating Effects while Updating . . . . .	152
9.5	FiberRef for References Specific to each Fiber . . . . .	154
9.6	Conclusion . . . . .	157
9.7	Exercises . . . . .	158
<b>10</b>	<b>Concurrent Structures: Promise - Work Synchronization</b>	<b>160</b>
10.1	Various Ways of Completing Promises . . . . .	161
10.2	Waiting on a Promise . . . . .	163
10.3	Promises and Interruption . . . . .	164
10.4	Combining Ref and Promise for More Complicated Concurrency Scenarios	165
10.5	Conclusion . . . . .	167
10.6	Exercises . . . . .	168
<b>11</b>	<b>Concurrent Structures: Queue - Work Distribution</b>	<b>169</b>
11.1	Queues as Generalizations of Promises . . . . .	169
11.2	Offering and Taking Values from a Queue . . . . .	170
11.3	Varieties of Queues . . . . .	171
11.3.1	Back Pressure Strategy . . . . .	172
11.3.2	Sliding Strategy . . . . .	173
11.3.3	Dropping Strategy . . . . .	173
11.4	Other Combinators on Queues . . . . .	174
11.4.1	Variants of Offer and Take . . . . .	175
11.4.2	Metrics on Queues . . . . .	175
11.4.3	Shutting Down Queues . . . . .	176
11.5	Conclusion . . . . .	176
11.6	Exercises . . . . .	176
<b>12</b>	<b>Concurrent Structures: Hub - Broadcasting</b>	<b>178</b>

12.1	Hub: An Optimal Solution to the Broadcasting Problem . . . . .	179
12.2	Creating Hubs . . . . .	181
12.2.1	Bounded Hubs . . . . .	181
12.2.2	Sliding Hubs . . . . .	182
12.2.3	Unbounded Hubs . . . . .	183
12.2.4	Dropping Hubs . . . . .	184
12.3	Operators on Hubs . . . . .	185
12.4	Conclusion . . . . .	186
12.5	Exercises . . . . .	187
<b>13</b>	<b>Concurrent Structures: Semaphore - Work Limiting</b>	<b>189</b>
13.1	Semaphore Interface . . . . .	189
13.2	Using Semaphores to Limit Parallelism . . . . .	190
13.3	Using Semaphore to Implement Operators . . . . .	191
13.4	Making a Data Structure Safe for Concurrent Access . . . . .	192
13.5	Conclusion . . . . .	194
<b>14</b>	<b>Resource Handling: Acquire Release - Safe Resource Handling</b>	<b>195</b>
14.1	The Limitation of try-finally in Asynchronous Programming . . . . .	195
14.2	Acquire Release as a Generalization of Try and Finally . . . . .	197
14.3	The Ensuring Operator . . . . .	199
14.4	Conclusion . . . . .	200
14.5	Exercises . . . . .	200
<b>15</b>	<b>Resource Handling: Scope - Composable Resources</b>	<b>202</b>
15.1	Reification of Acquire Release . . . . .	203
15.2	Scope as a Dynamic Scope . . . . .	206
15.3	Constructing Scoped Resources . . . . .	207
15.3.1	Fundamental Constructors . . . . .	207
15.3.2	Convenience Constructors . . . . .	209
15.4	Transforming Scoped Resources . . . . .	211
15.5	Using Scoped Resources . . . . .	211
15.6	Varieties of Scoped Resources . . . . .	213
15.7	Conclusion . . . . .	214
15.8	Exercises . . . . .	215
<b>16</b>	<b>Resource Handling: Advanced Scopes</b>	<b>218</b>
16.1	Scopes Revisited . . . . .	218
16.2	From Scopes to Resources . . . . .	220
16.3	Using Resources . . . . .	222
16.4	Child Scopes . . . . .	224
16.5	Putting it All Together . . . . .	227
16.6	Conclusion . . . . .	229
<b>17</b>	<b>Dependency Injection: Essentials</b>	<b>231</b>
17.1	The Environment Type . . . . .	231

17.2	Fundamental Operators for Working with the Environment . . . . .	233
17.3	Typical Uses for the Environment . . . . .	235
17.4	The Onion Architecture . . . . .	236
17.5	Layers . . . . .	241
17.5.1	Constructing Layers . . . . .	241
17.5.2	Providing Layers . . . . .	244
17.6	Accessors . . . . .	246
17.7	Service Pattern . . . . .	248
17.8	Declaring Dependencies via ZIO Environment or Constructor Arguments? 249	249
17.9	Conclusion . . . . .	249
17.10	Exercises . . . . .	249
<b>18</b>	<b>Dependency Injection: Advanced Dependency Injection</b>	<b>251</b>
18.1	What is ZEnvironment? . . . . .	251
18.2	Providing Multiple Services of the Same Type . . . . .	253
18.3	Handling Errors in Layer Construction . . . . .	253
18.4	Memoization of Dependencies . . . . .	255
18.5	Automatic ZLayer Derivation . . . . .	258
18.6	Conclusion . . . . .	260
<b>19</b>	<b>Dependency Injection: Contextual Data Types</b>	<b>262</b>
19.1	Problem . . . . .	263
19.2	Regional Settings . . . . .	264
19.2.1	Using ZIO Environment . . . . .	264
19.2.2	Using FiberRef . . . . .	267
19.3	Transactional Effects Using ZIO Environment . . . . .	269
19.4	Context and ZIO API . . . . .	273
19.5	Conclusion . . . . .	274
19.6	Exercise . . . . .	275
<b>20</b>	<b>Configuring ZIO Applications</b>	<b>276</b>
20.1	Configuration Challenges . . . . .	276
20.2	Configuration Methods in Practice . . . . .	276
20.3	Describing Configurations . . . . .	282
20.4	Config Providers . . . . .	286
20.5	Conclusion . . . . .	289
20.6	Exercise . . . . .	290
<b>21</b>	<b>Software Transactional Memory: Composing Atomicity</b>	<b>291</b>
21.1	Inability to Compose Atomic Actions with other Concurrency Primitives	291
21.2	Conceptual Description of STM . . . . .	294
21.3	Using STM . . . . .	296
21.4	Retrying and Repeating Transactions . . . . .	301
21.5	Limitations of STM . . . . .	301
21.6	Conclusion . . . . .	304
21.7	Exercises . . . . .	305

<b>22 Software Transactional Memory: STM Data Structures</b>	<b>306</b>
22.1 Description of STM Data Structures . . . . .	307
22.1.1 TArray . . . . .	307
22.1.2 TMap . . . . .	309
22.1.3 TPriorityQueue . . . . .	311
22.1.4 TPromise . . . . .	312
22.1.5 TQueue . . . . .	313
22.1.6 TRentrantLock . . . . .	315
22.1.7 TSemaphore . . . . .	319
22.1.8 TSet . . . . .	320
22.2 Creating Your Own STM Data Structures . . . . .	320
22.3 Conclusion . . . . .	325
22.4 Exercises . . . . .	325
<b>23 Software Transactional Memory: Advanced STM</b>	<b>328</b>
23.1 How STM Works Under the Hood . . . . .	328
23.2 Troubleshooting and Debugging . . . . .	334
23.3 Optimization . . . . .	336
23.3.1 Narrowing the Transactional Boundaries . . . . .	336
23.3.2 Fine-grained Locking . . . . .	337
23.3.3 Diagnosing High-contention Critical Sections . . . . .	341
23.3.4 Prefer Delaying Write Operations . . . . .	341
23.3.5 Deferring Commits to Shared State with Local Buffers . . . . .	342
23.4 Transaction Control Flows . . . . .	342
23.5 Conclusion . . . . .	343
23.6 Exercises . . . . .	343
<b>24 Advanced Error Management: Retries</b>	<b>345</b>
24.1 Limitations of Traditional Retry Operators . . . . .	345
24.2 Retrying and Repeating with ZIO . . . . .	347
24.3 Common Schedules . . . . .	349
24.3.1 Schedules for Recurrences . . . . .	350
24.3.2 Schedules for Delays . . . . .	350
24.3.3 Schedules for Conditions . . . . .	354
24.3.4 Schedules for Outputs . . . . .	355
24.3.5 Schedules for Fixed Points in Time . . . . .	357
24.4 Transforming Schedules . . . . .	358
24.4.1 Transforming Inputs and Outputs . . . . .	358
24.4.2 Summarizing Schedule Outputs . . . . .	360
24.4.3 Side Effects . . . . .	361
24.4.4 Environment . . . . .	361
24.4.5 Modifying Schedule Delays . . . . .	362
24.4.6 Modifying Decisions . . . . .	365
24.4.7 Schedule Completion . . . . .	366
24.5 Composing Schedules . . . . .	367
24.5.1 Intersection and Union of Schedules . . . . .	367

24.5.2	Sequential Composition of Schedules . . . . .	371
24.5.3	Alternative Schedules . . . . .	373
24.5.4	Function Composition of Schedules . . . . .	374
24.6	Implementation of Schedule . . . . .	375
24.7	Conclusion . . . . .	377
24.8	Exercises . . . . .	377
<b>25</b>	<b>Advanced Error Management: Debugging</b>	<b>379</b>
25.1	Understanding Execution Flows . . . . .	379
25.2	Printing Debug Information . . . . .	380
25.3	Enabling Diagnostic and Debug Logging . . . . .	381
25.4	Reading Stack Traces . . . . .	382
25.5	Fiber Dumps . . . . .	383
25.6	Supervising Fibers . . . . .	385
25.7	Conclusion . . . . .	389
<b>26</b>	<b>Advanced Error Management: Best Practices</b>	<b>390</b>
26.1	Recoverable Errors . . . . .	390
26.2	Non-recoverable Errors . . . . .	391
26.2.1	Defects . . . . .	391
26.2.1.1	Log Defects for Further Investigation . . . . .	392
26.2.1.2	Don't Swallow Defects in the First Place . . . . .	393
26.2.1.3	Model Domain Errors Using Algebraic Data Types . . . . .	394
26.2.1.4	Centralize Logging Defects . . . . .	396
26.2.1.5	Sandboxing at the Edge . . . . .	397
26.2.2	Fatal Errors . . . . .	398
26.3	Conclusion . . . . .	401
26.4	Exercises . . . . .	401
<b>27</b>	<b>Streaming: First Steps with ZStream</b>	<b>403</b>
27.1	Streams as Effectful Iterators . . . . .	403
27.2	Streams as Collections . . . . .	405
27.2.1	Implicit Chunking . . . . .	405
27.2.2	Potentially Infinite Streams . . . . .	406
27.2.3	Common Collection Operators On Streams . . . . .	407
27.3	Constructing Basic Streams . . . . .	408
27.3.1	Constructing Streams from Existing Values . . . . .	408
27.3.2	Constructing Streams from Effects . . . . .	408
27.3.3	Constructing Streams from Repetition . . . . .	409
27.3.4	Constructing Streams from Unfolding . . . . .	410
27.4	Running Streams . . . . .	411
27.4.1	Running a Stream as Folding over Stream Values . . . . .	412
27.4.2	Running a Stream for its Effects . . . . .	413
27.4.3	Running a Stream for its Values . . . . .	415
27.5	Type Parameters . . . . .	416
27.5.1	The Environment Type . . . . .	416

27.5.2 The Error Type . . . . .	417
27.6 Conclusion . . . . .	417
<b>28 Streaming: Next Steps With ZStream</b>	<b>419</b>
28.1 Sinks . . . . .	423
28.2 Pipelines . . . . .	427
28.3 Conclusion . . . . .	429
<b>29 Streaming: Channels are the Foundation</b>	<b>430</b>
29.1 Channels . . . . .	430
29.2 Channel Constructors . . . . .	435
29.3 Channel Operators . . . . .	437
29.4 Channel Scopes . . . . .	443
29.5 Conclusion . . . . .	444
29.6 Exercises . . . . .	444
<b>30 Streaming: Transforming Streams</b>	<b>446</b>
30.1 Mapping . . . . .	446
30.1.1 Stateful Mapping . . . . .	447
30.1.2 Concurrent Mapping . . . . .	448
30.2 Transform and Combine . . . . .	449
30.3 Flattening . . . . .	450
30.4 Filtering Operators . . . . .	451
30.5 Collecting (Filter-map) . . . . .	451
30.6 Grouping . . . . .	452
30.6.1 Grouping into Stream of Chunks . . . . .	452
30.6.1.1 Fixed-size Grouping . . . . .	452
30.6.1.2 Time-limited Grouping . . . . .	453
30.6.1.3 Adjacent Grouping . . . . .	453
30.6.2 Grouping into Stream of Streams . . . . .	455
30.7 Partitioning . . . . .	457
30.8 Broadcasting and Distributing . . . . .	457
30.9 Flow Control and Rate Limiting . . . . .	459
30.9.1 Buffering . . . . .	460
30.9.2 Debouncing . . . . .	461
30.9.3 Throttling . . . . .	462
30.10 Conclusion . . . . .	463
30.11 Exercises . . . . .	463
<b>31 Streaming: Combining Streams</b>	<b>465</b>
31.1 Stream Concatenation . . . . .	465
31.2 Merging Streams . . . . .	466
31.3 Zipping Streams . . . . .	466
31.4 Cartesian Product . . . . .	469
31.5 Fallback Streams . . . . .	470
31.6 Stateful Stream Combination . . . . .	471

31.7 Interleaving . . . . .	474
31.8 Conclusion . . . . .	474
31.9 Exercises . . . . .	475
<b>32 Streaming: Pipelines</b>	<b>476</b>
32.1 Pipelines as Stream Transformations . . . . .	476
32.2 Using Pipelines . . . . .	478
32.3 Constructing Pipelines . . . . .	481
32.4 Conclusion . . . . .	488
32.5 Exercises . . . . .	488
<b>33 Streaming: Sinks</b>	<b>490</b>
33.1 Sinks as Composable Aggregation Strategies . . . . .	490
33.2 Using Sinks . . . . .	492
33.2.1 Running Streams into Sinks . . . . .	493
33.2.2 Transducing Streams with Sinks . . . . .	494
33.2.3 Asynchronous Aggregation with Sinks . . . . .	495
33.2.4 Tapping Streams with Sinks . . . . .	496
33.3 Combining Sinks . . . . .	498
33.3.1 Transforming Inputs and Outputs . . . . .	498
33.3.2 Sequential Composition Of Sinks . . . . .	500
33.3.3 Parallel and Concurrent Composition of Sinks . . . . .	501
33.4 Constructing Sinks . . . . .	502
33.5 Conclusion . . . . .	504
<b>34 Observability: Logging</b>	<b>506</b>
34.1 Understanding the Basics . . . . .	507
34.2 Writing Custom Loggers . . . . .	510
34.3 Log Annotations . . . . .	513
34.4 Log Spans . . . . .	516
34.5 Integrating with Existing Logging Frameworks . . . . .	517
34.5.1 Forwarding ZIO Logs to SLF4J . . . . .	519
34.5.2 Forwarding SLF4J Logs to ZIO . . . . .	522
34.6 Logging Errors while Ignoring Them . . . . .	524
34.7 Conclusion . . . . .	524
<b>35 Observability: Metrics</b>	<b>526</b>
35.1 Getting Started . . . . .	526
35.2 Metrics Types . . . . .	528
35.2.1 Counter . . . . .	528
35.2.2 Gauge . . . . .	529
35.2.3 Histogram . . . . .	529
35.2.4 Summary . . . . .	531
35.2.5 Frequency . . . . .	532
35.3 Writing Metric Client . . . . .	532
35.3.1 Prometheus Client . . . . .	534

35.3.2 StatsD Client . . . . .	537
35.4 Built-in JVM and ZIO Runtime Metrics . . . . .	537
35.5 Conclusion . . . . .	538
35.6 Exercises . . . . .	539
<b>36 Testing: Basic Testing</b>	<b>540</b>
36.1 Tests as Effects . . . . .	540
36.2 Specs as Recursively Nested Collections of Tests . . . . .	546
36.3 Conclusion . . . . .	546
<b>37 Testing: Assertions</b>	<b>548</b>
37.1 Assertions as Predicates . . . . .	548
37.2 Common ZIO Assertions . . . . .	554
37.3 Smart Assertions . . . . .	558
37.4 Conclusion . . . . .	562
<b>38 Testing: The Test Environment</b>	<b>563</b>
38.1 Test Implementation of Standard Services . . . . .	563
38.1.1 Test Implementation of Clock Service . . . . .	565
38.1.2 Test Implementation of Console Service . . . . .	569
38.1.3 Test Implementation of Random Service . . . . .	572
38.1.4 Test Implementation of System Service . . . . .	575
38.2 Accessing the Live Environment . . . . .	575
38.3 Creating Custom Test Implementations . . . . .	577
38.4 Conclusion . . . . .	582
<b>39 Testing: Test Aspects</b>	<b>583</b>
39.1 Test Aspects as Polymorphic Functions . . . . .	586
39.2 Ability to Constrain Types . . . . .	587
39.3 Common Test Aspects . . . . .	590
39.3.1 Running Effects Before and After Tests . . . . .	590
39.3.2 Flaky and NonFlaky Tests . . . . .	592
39.4 Repeating and Retrying Tests . . . . .	592
39.4.1 Ignoring Tests . . . . .	592
39.4.2 Diagnosing Deadlocks . . . . .	593
39.4.3 Handling Platform and Version-specific Issues . . . . .	593
39.4.4 Accessing Live Implementation of Test Services . . . . .	595
39.4.5 Controlling Parallelism . . . . .	595
39.4.6 Asserting That a Test Fails . . . . .	596
39.4.7 Timing Tests . . . . .	596
39.4.8 Annotation and Tagging . . . . .	597
39.4.9 Verifying Post-Conditions . . . . .	598
39.5 Conclusion . . . . .	599
<b>40 Testing: Using Resources in Tests</b>	<b>600</b>
40.1 Providing Resources to Tests . . . . .	601

40.2	Sharing Resources Between Test Iterations . . . . .	602
40.3	Providing Resources to Test Suites . . . . .	603
40.4	Sharing Resources Between Tests . . . . .	603
40.5	Conclusion . . . . .	605
<b>41</b>	<b>Testing: Property-Based Testing</b>	<b>606</b>
41.1	Generators as Streams of Samples . . . . .	610
41.2	Constructing Generators . . . . .	612
41.3	Operators on Generators . . . . .	614
41.3.1	Transforming Generators . . . . .	614
41.3.2	Combining Generators . . . . .	615
41.3.3	Choosing Generators . . . . .	618
41.3.4	Filtering Generators . . . . .	620
41.3.5	Running Generators . . . . .	620
41.4	Random and Deterministic Generators . . . . .	621
41.5	Samples and Shrinking . . . . .	625
41.6	Conclusion . . . . .	629
<b>42</b>	<b>Testing: Test Annotations</b>	<b>631</b>
42.1	Tagging Tests . . . . .	631
42.2	How Test Annotations Works . . . . .	632
42.3	Using Test Annotations in Tests . . . . .	634
42.4	Using Test Annotations in Test Aspects . . . . .	635
42.5	Implementing Test Annotation Reporter . . . . .	637
42.6	Conclusion . . . . .	638
<b>43</b>	<b>Testing: Reporting</b>	<b>639</b>
43.1	Gathering Data . . . . .	639
43.2	Analyzing Data . . . . .	641
43.3	Conclusion . . . . .	641
<b>44</b>	<b>Applications: Parallel Web Crawler</b>	<b>642</b>
44.1	Definition of a Parallel Web Crawler . . . . .	643
44.2	Interacting with Web Data . . . . .	645
44.3	First Sketch of a Parallel Web Crawler . . . . .	649
44.4	Making It Testable . . . . .	652
44.5	Scaling It Up . . . . .	654
44.6	Conclusion . . . . .	659
<b>Appendix 1: The Scala Type System</b>		<b>660</b>
44.7	Types And Values . . . . .	660
44.8	Subtyping . . . . .	661
44.9	Any and Nothing . . . . .	662
44.9.1	Any . . . . .	662
44.9.2	Nothing . . . . .	664
44.10	Product and Sum Types . . . . .	665

44.10.1 Product Types . . . . .	665
44.10.2 Sum Types . . . . .	667
44.10.3 Combining Product and Sum Types . . . . .	668
44.11 Intersection and Union Types . . . . .	668
44.11.1 Intersection Types . . . . .	668
44.11.2 Union Types . . . . .	669
44.12 Type Constructors . . . . .	671
44.13 Conclusion . . . . .	673
<b>Appendix 2: Mastering Variance</b>	<b>674</b>
44.14 Definition of Variance . . . . .	674
44.15 Covariance . . . . .	676
44.16 Contravariance . . . . .	680
44.17 Invariance . . . . .	684
44.18 Advanced Variance . . . . .	686
44.19 Conclusion . . . . .	690
<b>Index</b>	<b>691</b>

# Foreword by John A. De Goes

We live in a complex and demanding cloud-native world. 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 over 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 24/7, 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*, a style of designing applications that 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.

## 0.1 A Brief History of ZIO

On June 5th, 2017, I opened an issue in the *Scalaz*<sup>1</sup> repository on Github, arguing that the next major version of this library needed a powerful and fast data type for asynchronous programming. My friend Vincent Marquez encouraged me to contribute to Scalaz, and 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.

Before *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.

---

<sup>1</sup><https://github.com/scalaz/scalaz>

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.

## 0.2 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 is 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 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.

## 0.3 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 essential 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. However, on November 16th, 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.

## 0.4 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 2nd, 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 11th, 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".

## 0.5 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.

## 0.6 Improved Type-Inference

Like 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 20th, 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.

## 0.7 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.

## 0.8 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 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 Kuckaert), 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.

## 0.9 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*<sup>2</sup>, 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 the 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.

## 0.10 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.

---

<sup>2</sup><https://youtu.be/p98W4bUtbO8>

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.

## 0.11 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.

## 0.12 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.

## 0.13 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 main-

tainer 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.

## 0.14 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 typesafe 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, asynchronous 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<sup>3</sup> 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

---

<sup>3</sup><https://github.com/zio/izumi-reflect>

the very heart of the ZIO runtime system: a full solution to the problem of structured concurrency.

## 0.15 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.

## 0.16 Preparation for ZIO 2.0

Following the release of ZIO 1.0, the journey was far from over. While the core fundamentals were solid, I knew there were still opportunities to make ZIO even more powerful and accessible. Rather than rushing into immediate changes, we chose a methodical approach, releasing a series of milestone versions that allowed us to gather valuable feedback from the growing ZIO community.

Throughout our journey, we encountered many obstacles that influenced our development.

As we move forward, I will no longer detail every step of the evolution process. Instead, I will focus on the challenges we faced that shaped it into what it is today.

The direction for ZIO 2.0 centered on four fundamental principles that I thought would shape the library's next phase:

- First, we wanted to improve ergonomics and the developer experience dramatically. The library had proven itself capable, but we knew we could make it more intuitive and enjoyable to use.
- Second, although ZIO was already fast, we saw opportunities for even more aggressive performance optimization. In the world of high-performance distributed systems, every millisecond counts.
- Third, we recognized the need to enhance operational capabilities. Modern applications demand robust observability and diagnostics, and we wanted ZIO to excel in production environments.
- Finally, we set our sights on streaming. While ZIO Stream was already powerful, we saw the potential to make it even more expressive and performant.

## **0.17 Simplification of ZIO Environment and Dependency Injection**

While ZIO 1.0's Has and ZLayer provided a powerful foundation for dependency management, real-world feedback revealed a gap between power and simplicity. This realization led me to what might have seemed unthinkable months earlier—the complete removal of the Has data type.

So, we introduced a new type-level map called ZEnvironment, which was built into the ZIO itself instead of exposing Has on the surface. This led us to remove the Has data type from type signatures and simplify the API.

Talented Scala developer Kit Langton played a crucial role in what followed. Using Scala's compile-time capabilities, Kit Langton helped develop a sophisticated auto-wiring system to construct dependency graphs automatically. This enabled us to eliminate entire categories of boilerplate codes when injecting dependencies since the early days of ZIO.

## **0.18 ZIO Becomes Composable Resourceful Effect**

While the Managed data type in ZIO 1.0 was powerful for handling resource safety, it created an unnecessary burden. Developers had to constantly switch between two parallel worlds: ZIO for regular effects and Managed for resource management. Additionally, features were duplicated between both types—any enhancements made to ZIO often required a corresponding implementation in Managed.

As I had previously consolidated error handling, environments, and concurrency into the core ZIO effect type, I saw an opportunity to fold resource management directly into ZIO itself. This simplification would eliminate one other data type from the library while mak-

ing resource-safe programming more natural and maintainable.

So, instead of maintaining two parallel worlds, I introduced scopes as first-class values through collaboration with Adam Fraser. By introducing and adding the Scope data type to ZIO's environment, I enabled resource management directly within ZIO's effects. This wasn't just a minor improvement but a fundamental unification of the library's core abstractions.

Developers could now handle resources using familiar ZIO operators and patterns without switching back and forth between ZIO and ZManaged types. Whether working with concurrent effects or managing resources, everything stayed within the same powerful abstraction. This unification meant that any code accepting a ZIO value could seamlessly work with scoped resources.

## 0.19 Service Pattern

Despite the powerful capabilities we introduced to the ZIO Environment and ZLayer, we were still concerned about the significant amount of boilerplate needed to define and implement services. The ceremony of defining services felt at odds with ZIO's emphasis on developer ergonomics and joy.

The module pattern was a step in the right direction but still required more simplification. I borrowed three key elements from object-oriented design:

- Interfaces to define service contracts
- Classes to implement services in terms of other service interfaces
- Constructor-based dependency injection to wire services together

This realization led to the ZIO Service Pattern 2.0, merging functional and object-oriented principles that felt intuitive to developers. This pattern helped developers structure ZIO applications that naturally align with clean architecture principles, particularly onion architecture.

## 0.20 Smart Assertions

Continuing to make the API more ergonomic and simpler, Kit Langton started working on a macro-based approach to simplify writing tests with assertions. The result was smart assertions, which allowed developers to express expected behavior using ordinary Scala expressions that return Boolean values. These expressions were then translated behind the scenes into fundamental assertions, making test code more readable and easier to understand.

## 0.21 More Concrete Types

The journey toward simplification that began with ZIO's earliest days—when we chose concrete data types over tagless-final style—continued to guide our design decisions after

ZIO 1.0. We have seen that polymorphic data types like Ref and Queue can be powerful but can also introduce unnecessary complexity and developer friction. So, we took the bold step of eliminating these polymorphic variants, keeping only their concrete counterparts. This opinionated move further streamlined the library, making it more approachable for commercial teams.

## 0.22 Unified Streaming

Another area of our focus was ZIO Streams. I knew we needed to improve the ZIO Stream. During long discussions with ZIO contributors, I became convinced that we needed a more principled, lower-level abstraction that could unify all our streaming operations.

This led to the creation of ZChannel, a powerful abstraction inspired by Java NIO channels but reimagined in a purely functional way. Like its inspiration, a ZChannel supports both reading and writing operations but with the full power of ZIO's type system and functional programming principles.

With the help of the growing ZIO team, particularly the talented contributor Daniel Vigovszky, we rebuilt our entire streaming infrastructure on top of this new foundation. ZStream, ZPipeline, and ZSink—while maintaining their distinct APIs that users had come to rely on—were now all unified under the hood as specialized channels.

I was particularly excited about how this unification made it easier for users to work with different streaming abstractions. If a developer needed to build something more advanced, they could now directly use channels, enabling their solutions to integrate seamlessly with all existing streaming data types.

## 0.23 Regional Settings and Contextual Scopes

Embracing regional settings through structured programming principles was another goal concerning ergonomics and developer experience. I envisioned a system where settings can be only referenced and controlled on their block scope. With the help of Adam Fraser, we tackled this challenge by first enhancing FiberRef to support compositional updates. This enhancement became the foundation for a complete overhaul of ZIO's configuration system. Now, we could have nested scopes with different settings, each affecting the behavior of the ZIO application. Then, the same principles were applied to other aspects of ZIO, including concurrency, logging, and tracing.

## 0.24 Observability

I remember those days when the problem of useless stack traces had plagued effect systems for years, and ZIO was no exception. When an error occurred, developers would get stack traces full of framework internals rather than helpful information about where their code had failed.

A talented ZIO contributor, Rob Walsh, started working on a solution to this problem and

presented his work on ZIO World. With his contribution, ZIO's execution tracing took to the next level. ZIO became the first effect system to provide truly useful execution traces for asynchronous code, which pointed directly to the failing user code instead of getting lost in framework internals. This has a fantastic outcome: Any developer who could read a Java stack trace could now read ZIO's cost-free async stack traces.

Toward our goal to make ZIO the foundation for writing cloud-native applications, we also tried to have built-in solutions for logging, metrics, and integrated fiber dumps. Now, it was easier than ever to understand what was happening in ZIO applications with built-in and out-of-the-box observability solutions.

## 0.25 Performance

Performance had always been a key focus for ZIO, but I wanted to push the boundaries even further. Adam Fraser took on this challenge, creating a sophisticated fiber-aware scheduler inspired by Rust's Tokio. His implementation introduced work-stealing algorithms that could automatically balance workloads across operating system threads, maximizing fiber thread affinity while utilizing all available cores.

I also began a complete overhaul of the ZIO runtime to eliminate unnecessary work and bring you as close to bare metal as possible. This led to a significant performance improvement in ZIO applications and laid the foundation for post-Loom Java. This work led to what I consider the first "third generation" runtime in the effect ecosystem—one that minimizes JVM stack usage and delivers unprecedented performance among effect systems.

Another area of performance optimization we focused on was the challenge of developers handling blocking versus non-blocking operations.

In ZIO 1.0, we had implemented a reasonable solution at the time: developers had to use the blocking operator to signal blocking operations to the runtime. This allowed the runtime to shift heavy blocking work to a separate thread pool, protecting our core asynchronous operations from being blocked.

While I was proud that ZIO 1.0 pioneered the first tooling to manage this through its blocking service—an innovation that other runtimes later adopted—I wasn't fully satisfied. Developers needed to explicitly label their blocking code, which proved problematic in practice. When developers forgot to mark blocking operations or simply didn't realize they were importing blocking code, their applications could suffer serious performance degradation or even deadlock.

Adam Fraser and I began tackling this challenge head-on. We envisioned a runtime smart enough to manage this complexity automatically, removing the burden from developers entirely. The result was auto-blocking, where the runtime could intelligently detect blocking operations and automatically shift them to a dedicated blocking thread pool—without requiring any explicit labeling from developers. This dramatically simplifies ZIO applications, eliminating the need to distinguish between blocking and non-blocking operations manually. However, in later releases, this feature was disabled by default.

## 0.26 ZIO 2.0

Finally, we released ZIO 2.0 on June 24, 2022, after two years of continuous development and improvements based on community feedback. ZIO 2.0 was the solid foundation for the future of asynchronous and concurrent programming in Scala.

After the release of ZIO 2.0, we have had several maintenance releases—reaching version 2.1.13 at the time of writing. With these releases, I consciously decided to shift our focus. Rather than continuing to add new features, I wanted to ensure long-term stability through careful maintenance and optimization. So, new contributors attracted to this vision, including Kyriacos Petrou, an incredibly skilled developer who stepped forward to improve the performance of the ZIO runtime system. His work included a complete reimplementation of our Software Transactional Memory runtime, which significantly improved performance under high contention scenarios.

## 0.27 What is Next?

Looking ahead into 2025, I see ZIO evolving in response to changes in the Scala ecosystem. While ZIO has established itself as the “enterprise effect system” with growing adoption throughout 2024, we must adapt to a shifting landscape where we are in a situation that fewer companies are starting new Scala projects, and many are exploring alternative languages with stronger stability and tooling histories.

This evolution demands a thoughtful restructuring of the ZIO ecosystem. Rather than continuing to expand horizontally with new projects, we’re focusing on vertical integration and stability—bringing essential components closer to the core while ensuring long-term maintainability.

Some projects that have proven themselves indispensable to the ZIO experience—including ZIO Logging, ZIO Config, ZIO Metrics, and ZIO Profiling—will be promoted into ZIO core. This integration will provide a more cohesive experience while ensuring these critical components receive the same rigorous maintenance and guarantees as the core library.

We’re also investing heavily in mature projects that form the backbone of production ZIO applications. Libraries like ZIO HTTP, ZIO Schema, ZIO JSON, and Quill will continue to receive focused attention and improvements. In some areas, rather than maintaining our own implementations, we’re identifying and supporting ecosystem-neutral libraries that can provide excellent ZIO integration without the overhead of full maintenance.

While I have exciting ideas for ZIO 3—ideas that push beyond anything currently explored in the effect system ecosystem—our immediate priority is stabilization. Companies building on ZIO 2.1 should expect a very long life from the 2.x line, where backward compatibility and binary compatibility take precedence over chasing the latest trends. This stability-first approach means continued support for both Scala 2.13 and Scala 3.x, ensuring that enterprises can build with confidence on the ZIO platform.

The next chapter in ZIO’s story will focus less on breaking new ground and more on

strengthening the foundations we've built. By focusing our community's energy on actively maintained projects and core capabilities, we're ensuring that ZIO continues to be the top choice for building resilient, scalable cloud-native applications.

## 0.28 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 its 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.

## 0.29 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.

## 0.30 Zionomicon

In your hands, you have Zionomicon, a comprehensive book lovingly crafted by myself, Adam Fraser, and Milad Khajavi, 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!

# Preface

Welcome to Zonomicon, a comprehensive guide to building modern, resilient applications with ZIO. As we venture into the world of concurrent and distributed systems programming, this book aims to be your trusted companion in mastering one of Scala's most powerful libraries for building robust, scalable applications.

## 0.31 Who Should Read This Book?

This book is written for:

- Scala developers looking to build robust, concurrent applications
- Software engineers interested in functional programming approaches to system design
- Developers working on distributed systems and cloud-native applications
- Teams transitioning to ZIO from other concurrent programming frameworks
- Anyone interested in learning how to build resilient, production-grade applications

While prior Scala experience is helpful, we've structured the material to be accessible to developers who are relatively new to the language. That said, basic familiarity with functional programming concepts will help you get the most out of this book.

While ZIO builds on functional programming principles, you won't need to learn category theory or complex FP terminology to use it effectively. We deliberately avoid academic jargon and abstract mathematical concepts, focusing instead on practical solutions to real-world problems. Our goal is to make functional programming accessible to all developers, regardless of their theoretical background.

## 0.32 How This Book Is Organized

We've organized the material to build your understanding progressively:

- **Essentials** - Core concepts and fundamental building blocks of the ZIO ecosystem
- **Parallelism and Concurrency** - Understanding ZIO's fiber-based concurrency model and parallel execution capabilities
- **Concurrent Structures** - Built-in primitives for managing shared state and coor-

- inating concurrent operations
- **Resource Handling** - Patterns and tools for safely managing resources like connections, files, and memory
- **Dependency Injection** - ZIO's approach to managing dependencies and configuring application components
- **Software Transactional Memory** - Composable approach to handling atomic operations across multiple data structures
- **Advanced Error Management** - Sophisticated techniques for handling errors, debugging, and implementing resilient systems
- **Streaming** - Processing potentially infinite data streams with backpressure
- **Testing** - Comprehensive testing framework built into ZIO for unit, integration, and property-based testing

Each chapter includes practical examples and exercises to reinforce your learning. We suggest you work through these examples to deepen your understanding of the concepts presented and compare your solutions with the answers provided in the exercise repository. If your solution uses different approaches, we encourage you to share them with us by sending a pull request.

## 0.33 How to Use This Book

While the book is designed to be read sequentially, each chapter is relatively self-contained. Experienced developers may choose to jump to specific topics of interest. However, we recommend skimming the first chapters until chapter 7, as they establish essential foundational concepts.

The code examples are available in a GitHub repository<sup>4</sup>, and we encourage you to experiment with them as you read. Nothing reinforces learning like hands-on experience.

We welcome your feedback on any mistakes, typos, or areas for improvement in the Zionomicon Discord channel<sup>5</sup>. You can also use this channel for specific questions about the book's content. For general ZIO questions, please visit the zio-users Discord channel<sup>6</sup>.

## 0.34 Zionomicon is Updated with ZIO 2.1

This book is based on ZIO 2.1, which represents a stable, production-ready version of the library. While future versions may introduce new features or improvements, the core concepts and patterns covered in this book will remain relevant.

---

<sup>4</sup><https://github.com/zio/zionomicon>

<sup>5</sup><https://discord.com/channels/724305612166660202/770324663279943682>

<sup>6</sup><https://discord.com/channels/629491597070827530/630498701860929559>

## 0.35 Acknowledgments

This book would not have been possible without the vibrant ZIO community, whose questions, feedback, and real-world experiences have helped shape both the library and this book's content. Also, deep gratitude goes to the incredible core ZIO team and all contributors! Your dedication to creating and maintaining this outstanding toolkit for Scala developers is genuinely inspiring!

We're excited to guide you on your journey to mastering ZIO. Whether you're building your first concurrent application or architecting complex distributed systems, we hope this book helps you harness the full power of ZIO to create applications that are not just functional but truly exceptional.

Let's begin the journey into the world of ZIO.

# Chapter 1

## Essentials: First Steps With ZIO

ZIO will help you build modern applications that are concurrent, resilient, efficient, and 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 we can leverage the default 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.

### 1.1 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 1.1. 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.

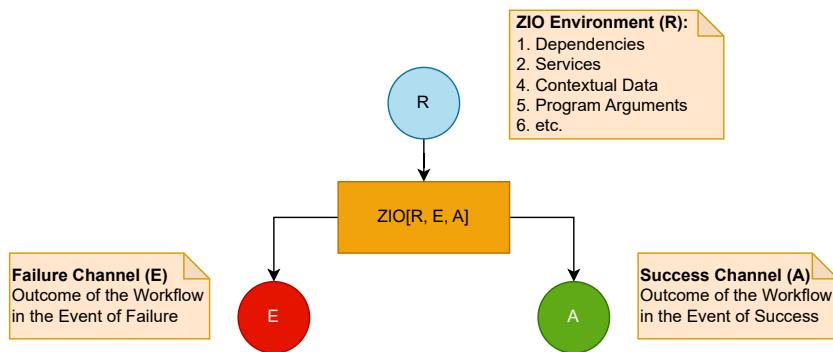


Figure 1.1: ZIO Workflow

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:

```

1 val goShoppingUnsafe: Unit = {
2     println("Going to the grocery store")
3 }

```

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`:

Listing 1.1: Going Shopping An Hour Later

```

1 import java.util.concurrent.{ Executors, ScheduledExecutorService
2   }
3 import java.util.concurrent.TimeUnit._
4
5 val scheduler: ScheduledExecutorService =
6   Executors.newScheduledThreadPool(1)
7
8 scheduler.schedule(
9   new Runnable { def run: Unit = goShoppingUnsafe },
10  1,
11  HOURS
12 )
13 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. However, 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:

```

1 import zio._
2
3 val goShopping =
4   ZIO.attempt(println("Going to the grocery store"))

```

Here, we are using the `attempt` 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:

```

1 val goShoppingLater =
2   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 `ZIOAppDefault` trait and implement the `run` method, as shown in the following snippet:

```

1 import zio._
2
3 object GroceryStore extends ZIOAppDefault {
4   val run = goShopping
5 }

```

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

## 1.2 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:

```

1 trait ZIO[R, E, A] {
2   ...
3   def flatMap[B](andThen: A => ZIO[R, E, B]): ZIO[R, E, B] = ...
4   ...
5 }
```

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 reads user input and then displays the input back to the user, as shown in the following snippet:

```

1 import scala.io.StdIn
2
3 val readLine =
4   ZIO.attempt(StdIn.readLine())
5
6 def printLine(line: String) =
7   ZIO.attempt(println(line))
8
9 val echo =
10  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:

```

1 val line = Console.readLine
2 Console.println(line)

```

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

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

```

1 val data = doQuery(query)
2 val response = generateResponse(data)
3 writeResponse(response)

```

We can translate this program into ZIO as follows:

```

1 ZIO.attempt(doQuery(query)).flatMap(data =>
2   ZIO.attempt(generateResponse(data)).flatMap(response =>
3     ZIO.attempt(writeResponse(response)))
4   )
5 )

```

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.

### 1.2.1 For Comprehensions

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

```

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

```

And rewrite it into the following *for comprehension*:

```

1 import zio._
2
3 val echo =
4   for {
5     line <- readLine
6     _    <- printLine(line)
7   } 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*:

```

1 for {
2   x <- doA
3   y <- doB(x)
4   z <- doC(x, y)
5 } yield x + y + z

```

Then Scala will translate it into the following code:

```

1 doA.flatMap(x =>
2   doB(x).flatMap(y =>
3     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`.

## 1.3 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:

```

1 val firstName =
2   ZIO.attempt(StdIn.readLine("What is your first name?"))
3
4 val lastName =
5   ZIO.attempt(StdIn.readLine("What is your last name?"))
6
7 val fullName =
8   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 for combining a number of effects sequentially when the result of one or more of the effects is not needed.

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

```
1 val helloWorld =
2   ZIO.attempt(print("Hello, ")) *> ZIO.attempt(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:

```
1 val printNumbers =
2   ZIO.foreach(1 to 100) { n =>
3     printLine(n.toString)
4 }
```

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:

```
1 val prints =
2   List(
3     printLine("The"),
4     printLine("quick"),
5     printLine("brown"),
6     printLine("fox")
7   )
8
9 val printWords =
10  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.

## 1.4 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 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:

```
1 | 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:

```
1 | final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =
  >
2 |   def map[B](f: A => B): ZIO[R, E, B] =
3 |     ZIO(r => self.run(r).map(f))
4 |   def flatMap[R1 <: R, E1 >: E, B](
5 |     f: A => ZIO[R1, E1, B]
```

```

6   ): ZIO[R1, E1, B] =
7     ZIO(r => self.run(r).fold(ZIO.fail(_), f).run(r))
8   }
9
10 object ZIO {
11   def attempt[A](a: => A): ZIO[Any, Throwable, A] =
12     ZIO(_ =>
13       try Right(a)
14       catch { case t: Throwable => Left(t) }
15     )
16   def fail[E](e: => E): ZIO[Any, E, Nothing] =
17     ZIO(_ => Left(e))
18 }
```

The `ZIO.attempt` method wraps a block of code in an effect, converting exceptions into `Left` values and successes into `Right` values. Notice that the parameter `a` to `ZIO.attempt` and `ZIO.fail` are *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, let's focus on the error and environment types to build some intuition about them since they may be less familiar.

### 1.4.1 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:

```

1 import zio._
2
```

```

3 lazy val readInt: ZIO[Any, NumberFormatException, Int] =
4   ???
5
6 lazy val readAndSumTwoInts: ZIO[Any, NumberFormatException, Int]
7   =
8   for {
9     x <- readInt
10    y <- readInt
11  } 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 `foldZIO` that will let us perform one effect if the original effect fails and another one if it succeeds:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =
2   >
3   def foldZIO[R1 <: R, E1, B](
4     failure: E => ZIO[R1, E1, B],
5     success: A => ZIO[R1, E1, B]
6   ): ZIO[R1, E1, B] =
7     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 run 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.

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =
2   >
3   def fold[B](
4     failure: E => B,
5     success: A => B
6   ): ZIO[R, Nothing, B] =
7     ZIO(r => Right(self.run(r).fold(failure, success)))

```

### 1.4.2 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 is actually an effect that doesn't need any specific kind of environment.

The two fundamental operations 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:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =
2   >
3   def provide(r: R): ZIO[Any, E, A] =
4     ZIO(_ => self.run(r))
5
6 object ZIO {
7   def environment[R]: ZIO[R, Nothing, R] =
8     ZIO(r => Right(r))
9 }

```

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`.

## 1.5 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. However, these type aliases are frequently used 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:

```

1 | type IO[+E, +A]    = ZIO[Any, E, A]
2 | type Task[+A]       = ZIO[Any, Throwable, A]
3 | type RIO[-R, +A]    = ZIO[R, Throwable, A]
4 | type UIO[+A]        = ZIO[Any, Nothing, A]
5 | 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[R, A]` - 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.

## 1.6 Comparison to Future

We can clarify what we have learned so far by comparing `ZIO{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.

### 1.6.1 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:

```

1 import scala.concurrent.Future
2 import scala.concurrent.ExecutionContext.Implicits.global
3
4 val goShoppingFuture: Future[Unit] =
5   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 is 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:

```

1 import scala.concurrent.ExecutionContext
2
3 trait Future[+A] {
4   def flatMap[B](f: A => Future[B])(implicit
5     ec: ExecutionContext
6   ): Future[B]
7 }

```

`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 codes involving ZIO we have seen require 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).

### 1.6.2 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`:

```
1 import scala.util.Try
2
3 trait Future[+A] {
4   def onComplete[B](f: Try[A] => B): Unit
5 }
```

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`:

```
1 def parseInt: Future[Int] =
2   ???
```

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 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 enters a valid integer? We just don't know 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`:

```
1 import scala.concurrent.Future
2
```

```

3 | def parseIntOrZero: Future[Int] =
4 |   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.

### 1.6.3 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 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.

## 1.7 More Effect Constructors

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

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

1. **Failable.** The `ZIO.attempt` constructor returns an effect that can fail with any kind of `Throwable` (`ZIO[Any, Throwable, A]`). This 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.attempt` 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.attempt` 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 these types into ZIO effects?

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

### 1.7.1 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:

```
1 | 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:

```
1 | import scala.io.StdIn
2
3 | val echo: Unit = {
4 |   val line = StdIn.readLine()
5 |   println(line)
6 | }
```

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:

```
1 | 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 bodies are pure expressions are called *pure functions*.

The `ZIO.attempt` 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:

```

1 import zio._
2
3 val readLine =
4   ZIO.attempt(StdIn.readLine())
5
6 def printLine(line: String) =
7   ZIO.attempt(println(line))
8
9 val echo =
10  for {
11    line <- readLine
12    _   <- printLine(line)
13  } 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. However, 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.

### 1.7.2 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`:

```

1 object ZIO {

```

```

2 |   def fail[E](e: => E): ZIO[Any, E, Nothing] = ???
3 |   def succeed[A](a: => A): ZIO[Any, Nothing, A] = ???
4 |

```

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 the 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)` constructs an effect that fails with the specified exception. The success type of the effect returned by `ZIO.fail` is `Nothing` because the effects created with this constructor cannot succeed.

We will see that effects that 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.

```

1 | import scala.util.Try
2 |
3 | object ZIO {
4 |   def fromEither[E, A](eea: => Either[E, A]): IO[E, A] = ???
5 |   def fromOption[A](oa: => Option[A]): IO[None.type, A] = ???
6 |   def fromTry[A](a: => Try[A]): Task[A] = ???
7 |

```

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.

### 1.7.3 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.attempt`. 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.succeed`, which converts procedural code into a ZIO effect that cannot fail:

```
1 object ZIO {
2   def succeed[A](a: => A): ZIO[Any, Nothing, A]
3 }
```

#### 1.7.3.1 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:

```
1 def getUserByIdAsync(id: Int)(cb: Option[String] => Unit): Unit =
2   ???
```

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:

```
1 getUserByIdAsync(0) {
2   case Some(name) => println(name)
3   case None =>      println("User not found!")
4 }
```

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.async`, and its type signature is shown in the following snippet:

```

1 object ZIO {
2     def async[R, E, A](
3         cb: (ZIO[R, E, A] => Unit) => Any
4     ): ZIO[R, E, A] =
5         ???
6 }
```

The type signature of `ZIO.async` 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.async` constructor as follows:

```

1 def getUserId(id: Int): ZIO[Any, None.type, String] =
2     ZIO.async { callback =>
3         getUserIdAsync(id) {
4             case Some(name) => callback(ZIO.succeed(name))
5             case None        => callback(ZIO.fail(None))
6         }
7     }
```

The callback provided by `async` 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 `getUserId` 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 `getUserId` computation is available, we will just continue with the other computations in the blueprint we have created.

Note here that in `async`, the callback function may only be invoked once, so it's not ap-

ropriate for converting all asynchronous APIs. If the callback may be invoked more than once, you can use the `async` 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:

```
1 def fromFuture[A](make: ExecutionContext => Future[A]): Task[A] =
2   ???
```

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:

```
1 def goShoppingFuture(
2   implicit ec: ExecutionContext
3 ): Future[Unit] =
4   Future(println("Going to the grocery store"))
5
6 val goShoppingZIO: Task[Unit] =
7   ZIO.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.

## 1.8 Default ZIO Services

`ZIO` provides four different default services for all applications:

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.

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` service 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.

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).

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

### 1.8.1 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:

```

1
2 object Clock {
3     def nanoTime: ZIO[Any, Nothing, Long]
4     def sleep(duration: => Duration): URIO[Any, Nothing, Unit]
5 }
```

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:

```

1 def delay[R, E, A](zio: ZIO[R, E, A])(
2     duration: Duration
3 ): ZIO[R, E, A] =
4     Clock.sleep(duration) *> zio
```

The `Clock` service is the building block for all time and scheduling functionality in ZIO.

### 1.8.2 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.attempt` constructor. This was useful in illustrating how to translate procedural to ZIO and demonstrating that 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 cases for console interaction that the `Console` service handles for us. (For example, reading from the console can fail only with an `IOException`.)

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

```
1 object Console {
2   val readLine: ZIO[Any, IOException, String]
3   def print(line: => String): ZIO[Any, Nothing, Unit]
4   def printLine(line: => String): ZIO[Any, Nothing, 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.

### 1.8.3 System

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

```
1 object System {
2   def env(variable: String): ZIO[Any, SecurityException, Option[
3     String]]
4   def property(prop: String): ZIO[Any, Throwable, Option[String]]
```

The two main methods on the `System` service are `System.env`, which accesses a specified environment variable, and `System.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.

### 1.8.4 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.

## 1.9 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:

```

1 import zio._
2
3 val readInt: ZIO[Any, Throwable, Int] =
4   for {
5     line <- Console.readLine
6     int  <- ZIO.attempt(line.toInt)
7   } 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:

```

1 import java.io.IOException
2
3 lazy val readIntOrRetry: ZIO[Any, IOException, Int] =

```

```

4   readInt
5     .orElse(Console.printLine("Please enter a valid integer"))
6     .zipRight(readIntOrRetry)
7   )

```

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

## 1.10 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. ZIO also ships with default services for interacting with the console, the system, random number generation.

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.

## 1.11 Exercises

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

```

1 def readFile(file: String): String = {
2   val source = scala.io.Source.fromFile(file)
3
4   try source.getLines().mkString
5   finally source.close()
6 }
7
8 def readFileZio(file: String) = ???

```

2. Implement a ZIO version of the function `writeFile` by using the `ZIO.attempt` constructor.

```

1 def writeFile(file: String, text: String): Unit = {
2   import java.io._
3   val pw = new PrintWriter(new File(file))
4   try pw.write(text)
5   finally pw.close
6 }
7
8 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`.

```

1 def copyFile(source: String, dest: String): Unit = {
2   val contents = readFile(source)
3   writeFile(dest, contents)
4 }
5
6 def copyFileZio(source: String, dest: String) = ???
```

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

```

1 def printLine(line: String) = ZIO.attempt(println(line))
2 val readLine                  = ZIO.attempt(scala.io.StdIn.
3   readLine())
4
5 printLine("What is your name?").flatMap(_ =>
6   readLine.flatMap(name => printLine(s"Hello, $name!"))
7 )
```

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

```

1 val random          = ZIO.attempt(scala.util.Random.
2   nextInt(3) + 1)
3 def printLine(line: String) = ZIO.attempt(println(line))
4 val readLine        = ZIO.attempt(scala.io.StdIn.
5   readLine())
6
7 random.flatMap { int =>
8   printLine("Guess a number from 1 to 3:").flatMap { _ =>
9     readLine.flatMap { num =>
10       if (num == int.toString) printLine("You guessed
11         right!")
12       else printLine(s"You guessed wrong, the number was
13         $int!")
```

```

10     }
11   }
12 }
```

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.

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A])
2
3 def zipWith[R, E, A, B, C](
4   self: ZIO[R, E, A],
5   that: ZIO[R, E, B]
6 )(f: (A, B) => C): ZIO[R, E, C] =
7   ???
```

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.

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A])
2
3 def collectAll[R, E, A](
4   in: Iterable[ZIO[R, E, A]]
5 ): ZIO[R, E, List[A]] =
6   ???
```

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 specified collection.

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A])
2
3 def foreach[R, E, A, B](
4   in: Iterable[A]
5 )(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
6   ???
```

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 fall back to the effect on the right-hand side.

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A])
2
3 def orElse[R, E1, E2, A](
4   self: ZIO[R, E1, A],
5   that: ZIO[R, E2, A]
6 ): ZIO[R, E2, A] =
7   ???
```

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 function `readFileZio` that you developed in these exercises, as well as `ZIO.foreach`.

```

1 object Cat extends ZIOAppDefault {
2     def run =
3         for {
4             args <- getArgs
5             //      -  <- ???
6         } yield ()
7 }
```

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

```

1 def eitherToZIO[E, A](either: Either[E, A]): ZIO[Any, E, A]
2     = ???
```

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.

```

1 def listToZIO[A](list: List[A]): ZIO[Any, None.type, A] =
2     ???
```

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

```

1 def currentTime(): Long = java.lang.System.currentTimeMillis
2
3 lazy val currentTimeZIO: ZIO[Any, Nothing, Long] = ???
```

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

```

1 def getCacheValue(
2     key: String,
3     onSuccess: String => Unit,
4     onFailure: Throwable => Unit
5 ): Unit =
6     ???
7
8 def getCacheValueZio(key: String): ZIO[Any, Throwable,
9     String] =
10    ???
```

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

```

1 trait User
2
3 def saveUserRecord(
4     user: User,
5     onSuccess: () => Unit,
6     onFailure: Throwable => Unit
7 ): Unit =
8     ???
9
10 def saveUserRecordZio(user: User): ZIO[Any, Throwable, Unit]
11     =
12     ???

```

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

```

1 import scala.concurrent.{ExecutionContext, Future}
2 trait Query
3 trait Result
4
5 def doQuery(query: Query)(implicit
6     ec: ExecutionContext
7 ): Future[Result] =
8     ???
9
10 def doQueryZio(query: Query): ZIO[Any, Throwable, Result] =
11     ???

```

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.

```

1 ``
2 object HelloHuman extends ZIOAppDefault {
3     val run = ???
4 }
5 ``

```

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 are correct or not.

```

1 import zio._
2
3 object NumberGuessing extends ZIOAppDefault {
4     val run = ???
5 }

```

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.

```
1 import java.io.IOException
2
3 def readUntil(
4     acceptInput: String => Boolean
5 ): ZIO[Any, IOException, String] =
6     ???
```

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.

```
1 def doWhile[R, E, A](
2     body: ZIO[R, E, A]
3 )(condition: A => Boolean): ZIO[R, E, A] =
4     ???
```

## Chapter 2

# Essentials: Testing ZIO Programs

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

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 building block and operator 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.

```
1 import zio._

2

3 def safeDivision(x: Int, y: Int): ZIO[Any, Unit, Int] =
4     ZIO.attempt(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. However, the only reason we can reason about it in this way is that each of the constructors and each of the operators combining them honor certain guarantees.

In the example above, the `ZIO.attempt` 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 `ZIO.attempt` 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.attempt 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 ZIO#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.

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

These powerful guarantees, along with an understanding of how they apply when composing different programs, are essential for building complex programs that maintain strong safety and efficiency properties for both ourselves and our users. So, it is critical to be able to verify through testing that the components we are creating really do honor the guarantees we think they do.

Of course, Scala already has a variety of other testing frameworks. 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:

```

1 import org.scalatest._

2

3 class ExampleSpec extends FunSuite {
4   test("addition works") {
5     assert(1 + 1 === 2)
6   }
7 }
```

This works but runs into problems when asserting ZIO effects instead of simple values. For example, here is an initial attempt to test a simple assertion about a ZIO effect:

```

1 class ExampleSpec2 extends FunSuite {
2   test("addition works") {
3     assert(ZIO.succeed(1 + 1) === 2)
4   }
5 }
```

This compiles, but this test doesn't make sense and will always fail because we compare 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:

```

1 import zio.{Runtime, Unsafe}
2
3 def unsafeRun[E, A](zio: ZIO[Any, E, A]): A =
4   Unsafe.unsafe { implicit unsafe =>
5     Runtime.default.unsafe.run(zio).getOrThrowFiberFailure()
6   }
7
8 class ExampleSpec3 extends FunSuite {
9   test("addition works") {
10     assert(unsafeRun(ZIO.succeed(1 + 1)) === 2)
11   }
12 }
```

This test now makes sense and will pass as written, but it still has several problems.

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 ZIO's features when implementing functionality related to testing.

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

However, as we learned in the previous chapter, `Future` is not interruptible, so 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 the 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 compositability of ZIO in writing our production code.

## 2.1 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 the ZIO Test, first add it as a dependency:

```
1 libraryDependencies ++= Seq(
2   "dev.zio" %% "zio-test"      % zioVersion,
3   "dev.zio" %% "zio-test-sbt" % zioVersion
4 )
```

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

```
1 import zio.test._
2 import zio.test.Assertion._
3
4 object ExampleSpec extends ZIOSpecDefault {
5
6   def spec = suite("ExampleSpec")(
7     test("addition works") {
8       assert(1 + 1)(equalTo(2))
9     }
10   )
11 }
```

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 we are making an assertion about and then an assertion 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.

```

1 object ExampleSpec extends ZIOSpecDefault {
2
3   def spec = suite("ExampleSpec")(
4     test("ZIO.succeed succeeds with specified value") {
5       assertZIO(ZIO.succeed(1 + 1))(equalTo(2))
6     }
7   )
8 }
```

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

The test framework knows that the test is returning a ZIO effect. It will automatically run the test along with all the other tests in the spec and report the results consistently across platforms.

Similarly, replacing `assert` with `assertZIO` indicates that the left-hand side of the assertion will have 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 `assertZIO` here. In fact, we can replace `assertZIO` with `assert` using `map` or a `for` comprehension.

```

1 object ExampleSpec extends ZIOSpecDefault {
2
3   def spec = suite("ExampleSpec")(
4     test("testing an effect using map operator") {
5       ZIO.succeed(1 + 1).map(n => assert(n)(equalTo(2)))
6     },
7     test("testing an effect using a for comprehension") {
8       for {
9         n <- ZIO.succeed(1 + 1)
10      } yield assert(n)(equalTo(2))
11    }
12  )
13 }
```

All three ways of writing this test are equivalent.

In general, we find that using `assertZIO` 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, disjunction, or `!` to negate an assertion:

```

1 object ExampleSpec extends ZIOSpecDefault {
2
3   def spec = suite("ExampleSpec")(
4     test("and") {
```

```

5   for {
6     x <- ZIO.succeed(1)
7     y <- ZIO.succeed(2)
8   } yield assert(x)(equalTo(1)) &&
9     assert(y)(equalTo(2))
10  }
11 }
12 }
```

## 2.2 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 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.

```

1 type Assertion[-A] = A => Boolean
2
3 def equalTo[A](expected: A): Assertion[A] =
4   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.

The `Assertion` companion object in the `zio.test` package contains various assertions. For now, we will provide a few examples to demonstrate their capabilities.

Assertions can be specialized for particular data types, so there are various 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.

```

1 object ExampleSpec extends ZIOSpecDefault {
2
3   def spec = suite("ExampleSpec")(
4     test("hasSameElement") {
5       assert(List(1, 1, 2, 3))(hasSameElements(List(3, 2, 1, 1)))
6     }
7   )
8 }
```

Another particularly useful assertion is the `fails` assertion, which allows us to assert that an effect fails with a particular value. We can use this by first calling `exit` 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.

```

1 object ExampleSpec extends ZIOSpecDefault {
2
3     def spec = suite("ExampleSpec")(
4         test("fails") {
5             for {
6                 exit <- ZIO.attempt(1 / 0).catchAll(_ => ZIO.fail(()))
7                 exit
8             } yield assert(exit)(fails(isUnit))
9         }
10    )

```

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 the 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:

```

1 val assertion: Assertion[Iterable[Int]] =
2   isNonEmpty && forall(nonNegative)

```

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:

```

1 val assertion: Assertion[Iterable[Any]] =
2   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:

```

1 val assertion: Assertion[Iterable[Any]] =

```

```
2 |     not(isDistinct)
```

## 2.3 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.

```
1 | val greet: ZIO[Any, Nothing, Unit] = 
2 |   for {
3 |     name <- Console.readLine.orDie
4 |     _      <- Console.printLine(s"Hello, $name!").orDie
5 |   } yield ()
```

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 ourselves 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 a 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? `readLine` will read an actual line from the console, and `printLine` will 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.

```
1 | object ExampleSpec extends ZIOSpecDefault {
2 |
3 |   def spec = suite("ExampleSpec")(
4 |     test("greet says hello to the user") {
5 |       for {
6 |         _      <- TestConsole.feedLines("Jane")
7 |         _      <- greet
```

```

8     value <- TestConsole.output
9   } yield assert(value)(equalTo(Vector("Hello, Jane!\n")))
10  }
11 }
12 }
```

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 and 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.

Using the `TestClock`, we could test a method that delays for a specified period of time:

```

1 val goShopping: ZIO[Any, Nothing, Unit] =
2   Console.printLine("Going shopping!").orDie.delay(1.hour)
3
4 object ExampleSpec extends ZIOSpecDefault {
5
6   def spec = suite("ExampleSpec")(
7     test("goShopping delays for one hour") {
8       for {
9         fiber <- goShopping.fork
10        -      <- TestClock.adjust(1.hour)
11        -      <- fiber.join
12       } yield assertCompletes
13     }
14   )
15 }
```

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 the 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.

## 2.4 Common Test Aspects

Another nice feature of ZIO Test to be aware of is *test aspects*. Test aspects modify some aspects 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 ensure it is not flaky.

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

```

1 import zio.test.TestAspect._

2

3 object ExampleSpec extends ZIOSpecDefault {

4

5   def spec = suite("ExampleSpec")(
6     test("this test will be repeated to ensure it is stable") {
7       assertZIO(ZIO.succeed(1 + 1))(equalTo(2))
8     } @@ nonFlaky
9   )
10 }
```

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, ensuring 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.

Many different test aspects can modify how tests are executed, such as running tests only on a certain platform or Scala version. So, if you need to modify how your tests are executed, it is worth checking whether there is already a test aspect for that.

## 2.5 Basic Property-based Testing

Another important feature to be aware of when you start writing tests is that the ZIO Test supports property-based testing out of the box.

In property-based testing, instead of manually generating inputs and verifying the expected outputs, the test framework generates a whole collection of inputs from a *distribution* of potential inputs you specify. It 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, including 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 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 testing 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, the generators may be infinite or finite and may be random or deterministic.

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

```
1 val intGen: Gen[Any, Int] =
2   Gen.int
```

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

```
1 object ExampleSpec extends ZIOSpecDefault {
2
3   def spec = suite("ExampleSpec")(
4     test("integer addition is associative") {
5       check(intGen, intGen, intGen) { (x, y, z) =>
6         val left  = (x + y) + z
7         val right = x + (y + z)
8         assert(left)(equalTo(right))
9       }
10    )
11  }
12}
```

Notice how similar we write property-based tests to normal tests.

We still use the `test` method to label a test.

Inside our test, we call the `check` operator, specifying each of the generators we want to use as arguments. 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 `assertZIO` 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 `checkZIO`, which is like `check` except that it allows us to perform effects within the property-based test. There are also `checkN` variants that specify 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:

```
1 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 the 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 signature type.

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 `asciiString` generator:

```
1 val genName: Gen[Random with Sized, String] =
2   Gen.asciiString
```

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:

```

1 val genAge: Gen[Random, Int] =
2   Gen.int(18, 120)

```

With these two generators implemented, all that is left is to combine them, conceptually sampling a name from `genName` and 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 do this quite easily:

```

1 val genUser: Gen[Random with Sized, User] =
2   for {
3     name <- genName
4     age  <- genAge
5   } yield User(name, age)

```

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!

## 2.6 Conclusion

This chapter has provided a brief overview of the 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 read 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.

## 2.7 Exercises

1. Write a ZIO program that simulates a countdown timer (e.g., prints numbers from 5 to 1, with a 1-second delay between each). Test this program using `TestClock`.
2. Create a simple cache that expires entries after a certain duration. Implement a program that adds items to the cache and tries to retrieve them. Write tests using `TestClock` to verify that items are available before expiration and unavailable after expiration.
3. Create a rate limiter that allows a maximum of N operations per minute. Implement a program that uses this rate limiter. Write tests using `TestClock` to verify that the rate limiter correctly allows or blocks operations based on the time window.

4. Implement a function that reverses a list, then write a property-based test to verify that reversing a list twice returns the original list.
5. Implement an AVL tree (self-balancing binary search tree) with insert and delete operations. Write property-based tests to verify that the tree remains balanced after each operation. A balanced tree is one where the height of every node's left and right subtrees differs by at most one.

## Chapter 3

# 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 a lack of memory, 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.

### 3.1 Exceptions Versus Defects

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

For example, take the following snippet of code:

```
1 import zio._  
2  
3 val divisionByZero: UIO[Int] =  
4   ZIO.succeed(1 / 0)
```

We used the `succeed` constructor here, which means the return type is `UIO`, indicating the effect cannot fail. But on the JVM, dividing by zero will throw an `ArithmaticException`. 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, we can do nothing except abort the program. These are also called **fiber failures**, **untyped failures**, or **unchecked failures**.

Treating 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`, so treating it as a defect could make sense. However, in an 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, such as retrying the connection or using a backup database. However, 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 that we could be late if we get stuck on a work call. But it wouldn't really make sense for us to tell our friend that we might be struck by lightning 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? However, always communicating every possible mode of failure is not helpful. What should our friend do because we might be struck by lightning?

Similarly, of course, any program can indeed 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.

## 3.2 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 with several subtypes that capture all possible failure scenarios for an effect.

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

```

1 sealed trait Cause[+E]
2
3 object Cause {
4   final case class Die(t: Throwable) extends Cause[Nothing]
5   final case class Fail[+E](e: E)     extends Cause[E]
6 }
```

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.

## 3.3 Exit

Another data type closely related to `Cause` is `Exit`. `Exit` is a 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]`:

```

1 sealed trait Exit[+E, +A]
2
3 object Exit {
4   final case class Success[+A](value: A) extends Exit[Nothing, A]
5   final case class Failure[+E](cause: Cause[E])
6     extends Exit[E, Nothing]
7 }
```

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 `Exit` 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 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.

## 3.4 Handling Defects

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

```

1 final case class ZIO[-R, +E, +A] (
2   run: R => Either[E, A]
3 ) { self =>
4   def foldZIO[R1 <: R, E1, B] (
5     failure: E => ZIO[R1, E1, B],
6     success: A => ZIO[R1, E1, B]
7   ): ZIO[R1, E1, B] =
8     ZIO(r => self.run(r).fold(failure, success).run(r))
9 }
```

This philosophy is an excellent default because, in most cases, handling defects doesn't make sense. Defects represent unanticipated or unrecoverable failures. So, most of the time, we want to avoid further complicating the signature of our error handling operators by specifying how to handle an unknown failure that we never expected and may be unable 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`. On the other hand, the report generation application 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.

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

```

1 final case class ZIO[-R, +E, +A] (
2   run: R => Either[Cause[E], A]
3 ) { self =>
4   def foldCauseZIO[R1 <: R, E1, B] (
5     failure: Cause[E] => ZIO[R1, E1, B],
```

```

6     success: A => ZIO[R1, E1, B]
7   ): ZIO[R1, E1, B] =
8     ZIO(r => self.run(r).fold(failure, success).run(r))
9 }
```

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

## 3.5 Converting Errors to Defects

As we move up from lower levels of our application to higher levels, 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 is 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 `ZIO#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:

```

1 sealed trait ZIO[-R, +E, +A] {
2   def orDie(implicit ev: E <:< Throwable): ZIO[R, Nothing, A] =
3     ???
4 }
```

We could use the method shown in the following example:

```

1 def readFile(file: String): ZIO[Any, IOException, String] =
2   ???
3
4 lazy val result: ZIO[Any, Nothing, String] =
5   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 only to 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 `ZIO#refineWith` method as shown in the following snippet:

```

1 def readFile(file: String): ZIO[Any, Throwable, String] =
2   ???
3
4 def readFile2(file: String): ZIO[Any, IOException, String] =
5   readFile(file).refineWith {
6     case e : IOException => e
7   }

```

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`:

```

1 def readFile(file: String): ZIO[Any, Throwable, String] =
2   ???
3
4 def readFile2(file: String): ZIO[Any, IOException, String] =
5   readFile(file).refineToOrDie[IOException]

```

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

## 3.6 Multiple Failures

So far, we have implicitly assumed 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 will not fail with both of the exceptions at the same time.

This is a reasonable assumption in simple procedural code. 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, if our program is concurrent, multiple parts of it may be executed simultaneously. We will discuss ZIO's support for concurrent programming soon, but a simple example will suffice for now.

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:

```

1 lazy val example =
2   try {
3     throw new Exception("Error using file!")
4   } finally {
5     throw new Exception("Couldn't close file!")
6 }
```

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 ensure a finalizer runs no matter how our code terminates, perhaps to close the file. But the finalizer throws an exception itself! In this case, which exception should be propagated to higher application levels?

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:

```

1 sealed trait Cause [+E]
2
3 object Cause {
4   final case class Die(t: Throwable) extends Cause[Nothing]
5   final case class Fail [+E] (e: E) extends Cause[E]
6   final case class Both [+E] (left: Cause[E], right: Cause[E])
7     extends Cause[E]
8   final case class Then [+E] (left: Cause[E], right: Cause[E])
9     extends Cause[E]
```

```
10 }
```

The `Both` data type represents two causes of failure, which occur concurrently. For example, if we were describing the query result 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.

## 3.7 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:

```
1 sealed trait ZIO[-R, +E, +A] {
2   def orElse[R1 <: R, E2, A1 >: A](
3     that: ZIO[R1, E2, A1]
4   ): ZIO[R1, E2, A1] =
5   ????
6 }
```

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 fall back 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.

## 3.8 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 of how errors combine.

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

```
1 final case class ApiError(message: String)
2   extends Exception(message)
3 final case class DbError(message: String)
4   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:

```

1 trait Result
2
3 lazy val callApi: ZIO[Any, ApiError, String] = ????
4 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`:

```

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

```

This default, called *supertype composition*, works well for error hierarchies that share a common structure. For example, most error 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 very useful for describing how an effect may fail.

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

```

1 final case class InsufficientPermission(
2   user: String,
3   operation: String
4 )
5
6 final case class FileIsLocked(file: String)
7
8 def shareDocument(
9   doc: String
10 ): ZIO[Any, InsufficientPermission, Unit] =
11   ???
12
13 def moveDocument(
14   doc: String,
15   folder: String
16 ): ZIO[Any, FileIsLocked, Unit] =
17   ???

```

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

```

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

```

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 with 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:

```

1 type DocumentError = Either[InsufficientPermission, FileIsLocked]
2
3 lazy val result2: ZIO[Any, DocumentError, Unit] =
4   shareDocument("347823")
5     .mapError(Left(_))
6     .zip(moveDocument("347823", "/temp/").mapError(Right(_)))
7     .unit

```

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.

## 3.9 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 address 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:

```
1 lazy val effect: ZIO[Any, IOException, String] = ???  
2  
3 effect.tapCause(cause => console.printLine(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.

## 3.10 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 profiles are optional in our database. In this case, our lookup function might have the following type signature:

```
1 trait DatabaseError  
2 trait UserProfile  
3  
4 def lookupProfile(  
5   userId: String  
6 ): ZIO[Any, DatabaseError, Option[UserProfile]] =  
7 ???
```

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:

```
1 def lookupProfile2(  
2   userId: String  
3 ): ZIO[Any, Option[DatabaseError], UserProfile] =  
4   lookupProfile.foldZIO(  
5     error => ZIO.fail(Some(error)),  
6     success => success match {  
7       case None          => ZIO.fail(None)  
8       case Some(profile) => ZIO.succeed(profile)
```

```

9     }
10    )

```

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 information as 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 `unsome` to shift it back to the success channel.

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

```

1 def lookupProfile3(
2   userId: String
3 ): ZIO[Any, Option[DatabaseError], UserProfile] =
4   lookupProfile(userId).some

```

You can easily handle wrapped error types by leveraging ZIO's typed error channel.

## 3.11 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 errors 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 a `MissingValue` error.

Rather than errors being something you react to when you debug an application, typed errors let you be proactive 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.

## 3.12 Conclusion

As we have seen in this chapter, ZIO’s error model is quite comprehensive, providing a full solution for both recoverable and unrecoverable errors.

ZIO provides a rich set of powerful operators for recovering from and transforming errors, and the polymorphic error channel removes the pain of 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.

## 3.13 Exercises

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.

```
1 def failWithMessage(string: String) =
2   ZIO.succeed(throw new Error(string))
```

2. Using the ZIO#foldCauseZIO 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.

```
1 def recoverFromSomeDefects[R, E, A](zio: ZIO[R, E, A])(
2   f: Throwable => Option[A]
3 ): ZIO[R, E, A] =
4   ???
```

3. Using the ZIO#foldCauseZIO 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.

```
1 def logFailures[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] =
2   ???
```

4. Using the ZIO#foldCauseZIO method, which “runs” an effect to an Exit value, implement the following function, which will execute the specified effect on any failure at all:

```
1 def onAnyFailure[R, E, A](
2   zio: ZIO[R, E, A],
3   handler: ZIO[R, E, Any]
4 ): ZIO[R, E, A] =
```

5 |    ???

5. Using the `ZIO#refineOrDie` method, implement the `ioException` function, which refines the error channel to only include the `IOException` error.

```
1 def ioException[R, A](
2   zio: ZIO[R, Throwable, A]
3 ): ZIO[R, java.io.IOException, A] =
4   ???
```

6. Using the `ZIO#refineToOrDie` method, narrow the error type of the following effect to just `NumberFormatException`.

```
1 val parseNumber: ZIO[Any, Throwable, Int] =
2   ZIO.attempt("foo".toInt)
```

7. Using the `ZIO#foldZIO` 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).

```
1 def left[R, E, A, B](
2   zio: ZIO[R, E, Either[A, B]]
3 ): ZIO[R, Either[E, B], A] =
4   ???
5
6 def unleft[R, E, A, B](
7   zio: ZIO[R, Either[E, B], A]
8 ): ZIO[R, E, Either[A, B]] =
9   ???
```

8. Using the `ZIO#foldZIO` 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).

```
1 def right[R, E, A, B](
2   zio: ZIO[R, E, Either[A, B]]
3 ): ZIO[R, Either[E, A], B] =
4   ???
5
6 def unright[R, E, A, B](
7   zio: ZIO[R, Either[E, A], B]
8 ): ZIO[R, E, Either[A, B]] =
9   ???
```

9. Using the `ZIO#sandbox` method, implement the following function.

```
1 def catchAllCause[R, E1, E2, A](
2   zio: ZIO[R, E1, A],
3   handler: Cause[E1] => ZIO[R, E2, A]
```

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

10. Using the `ZIO#foldCauseZIO` method, implement the following function.

```
1 | def catchAllCause[R, E1, E2, A] (
2 |   zio: ZIO[R, E1, A],
3 |   handler: Cause[E1] => ZIO[R, E2, A]
4 | ): ZIO[R, E2, A] = ???
```

## Chapter 4

# 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 when starting to write code with ZIO is integrating 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 these libraries often 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 integrating with specific other types of code, it is helpful to discuss a few principles for integrating ZIO with other kinds of code that apply more generally.

First, when starting to use ZIO or migrating a code base to ZIO, beginning with a small, relatively self-contained part of your application is often helpful.

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 `Runtime#unsafe.run` 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 `Runtime#unsafe.run` at the “edges” of the part of your application that has been migrated to ZIO.

Ideally, in a program written entirely using ZIO, your entire program would be one ZIO effect describing all of your program logic:

```

1 trait Example extends ZIOAppDefault {
2
3   val run = myProgramLogic
4
5   val myProgramLogic: ZIO[ZEnv, Any, Any] =
6     ???
7 }
```

You would only call `Runtime#unsafe.Run` once at the top of your entire application, often implementing the `App` trait. This is ideal, and `Runtime#unsafe.run` 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 `Runtime#unsafe.run` except at the very top of your application.

In contrast, with a mixed code base, you will have to call `Runtime#unsafe.run` at every edge between the “island” of code you migrated to ZIO and the rest of your code base, which is fine.

To facilitate this, creating a single `Runtime` at the top of your application that you can use to run all of your effects is often helpful:

```

1 import zio._
2
3 val runtime: Runtime[Any] =
4   Runtime.default
5
6 object MyZIOService {
7   def doSomething: ZIO[Any, Throwable, Int] =
8     ???
9 }
10
11 object MyLegacyService {
12
13   def doSomethingElse: Int = {
14     val something: Exit[Throwable, Int] = {
15       Unsafe.unsafe { implicit unsafe =>
```

```

16     runtime.unsafe.run(MyZIOService.doSomething)
17   }
18 }
19 something match {
20   case Exit.Success(value) => value
21   case Exit.Failure(cause) =>
22     throw new RuntimeException(
23       cause.prettyPrint,
24       cause.squash
25     )
26   }
27 }
28 }
```

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.unsafe.fromLayer` constructor to create a runtime from a `ZLayer` that produces the services you need.

Adopting this pattern allows you to 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 expand 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 `Runtime#unsafe.run` 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 `unsafe.run` 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 `unsafe.run` 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, such as allowing you to reason about your entire program in the same way and avoiding the 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 how that framework addresses your business problems or because it is just not worth migrating, which 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 discuss 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{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.

## 4.1 Integrating with Java

As Java has developed over various versions, the authors of the `java.concurrent` package have implemented several different abstractions to describe 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 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.

```
1 import zio._
2 import java.util.concurrent.Future
3
4 object ZIO {
5   def fromFutureJava[A](thunk: => Future[A]): Task[A] =
6     ???
7 }
```

There are a couple of things to note about this.

First, the `ZIO.fromFutureJava` constructor and 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 `ZIO.fromFutureJava` constructor returns an effect that runs on the blocking executor. 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. 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. If that is the situation you are in, you are already blocking to get the result of that computation, so the `fromFutureJava` constructor just makes sure it is run on the right thread pool.

To convert a `CompletionStage` or a `CompletableFuture` into a `ZIO` effect, we can use the `ZIO.fromCompletionStage` constructor:

```
1 import java.util.concurrent.CompletionStage
2
3 object ZIO {
4     def fromCompletionStage[A](
5         thunk: => CompletionStage[A]
6     ): Task[A] =
7     ???
8 }
```

This is very similar to the `ZIO.fromFutureJava` constructor, except it works with a `CompletionStage` instead of a `Future`. Unlike `ZIO.fromFutureJava`, this runs on `ZIO`'s default executor 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 `CompletableFuture` is a subtype of a `CompletionStage`, there is no separate `ZIO.fromCompletableFuture` constructor. Instead, you can just use `ZIO.fromCompletionStage` on any `CompletableFuture` 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, you can take a couple of approaches 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 succeed constructor:

```

1 import java.util.concurrent.atomic.AtomicReference
2
3 import zio._
4
5 def setAtomicReference[A] (
6   reference: AtomicReference[A]
7 )(a: A): UIO[Unit] =
8   ZIO.succeed(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 a mutable state. 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:

```

1 libraryDependencies ++= Seq(
2   "dev.zio" %% "zio-interop-guava" % "32.1.0"
3 )

```

With this dependency added, we can then convert a `ListenableFuture` into a ZIO value with the `ZIO.fromListenableFuture` operator:

```

1 import zio._
2 import java.util.concurrent.Executor
3
4 import com.google.common.util.concurrent.ListenableFuture

```

```

5  object guava {
6    def fromListenableFuture[A](
7      make: Executor => ListenableFuture[A]
8    ): Task[A] =
9      ???
10   }
11 }
```

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 be run on ZIO's own thread pool when evaluated.

## 4.2 Integrating with Javascript

Just like ZIO has tools to integrate with data types from the Java `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 `ZIO#toPromiseJS` and `ZIO#toPromiseJSWith` operators:

```

1 type JSPromise = scala.scalajs.js.Promise
2
3 trait ZIO[-R, +E, +A] {
4   def toPromiseJS(
5     implicit ev: E <:< Throwable
6   ): URIO[R, JSPromise[A]]
7   def toPromiseJSWith(f: E => Throwable): URIO[R, JSPromise[A]]
8 }
```

The `ZIO#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 `ZIO#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:

```

1 object ZIO {
2     def fromPromiseJS(promise: => JSPromise[A]): Task[A] =
3         ???
4 }
```

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 ZIO.fromPromiseJS or define it elsewhere as a `def` or `lazy val` to prevent it from being evaluated too early.

### 4.3 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:

```

1 import zio._
2 import scala.io.StdIn
3
4 val zioGreet: UIO[Unit] =
5     for {
6         name <- ZIO.succeed(StdIn.readLine("What's your name?"))
7         _      <- ZIO.succeed(println(s"Hello, $name!"))
8     } yield ()
```

In contrast, the same program written with the Cats Effect library would look like this:

```

1 import cats.effect._
2 import cats.syntax.all._
3
4 def catsGreet[F[_]: Sync]: F[Unit] =
5     for {
6         name <- Sync[F].delay(StdIn.readLine("What's your name?"))
7         _      <- Sync[F].delay(println(s"Hello, $name!"))
8     } 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.attempt and ZIO.succeed 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 classes of the appropriate type 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 regarding 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:

```
1 | val zioGreet: Task[Unit] =  
2 |   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 which 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. We need to use Throwable for the error type 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 the one in the example above works.

To get it, first, add a dependency on `zio-interop-cats` as follows:

```
1 libraryDependencies ++= Seq(
2   "dev.zio" %% "zio-interop-cats" % "23.1.0"
3 )
```

Then, importing `zio.interop.catz._` should bring all the required instances into scope:

```
1 import zio.interop.catz._
2
3 val zioGreet: Task[Unit] =
4   catsGreet[Task]
```

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.

## 4.4 Integrating with Specific Libraries

In addition to working with Cats Effect in general, ZIO users frequently need to work with a couple of specific libraries written in the Cats Effect style, which can pose special challenges.

The first of these is Doobie.

### 4.4.1 Integrating with 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 describes concurrent programs without actually running them.

For example, here is how we might describe a simple SQL query using Doobie:

```
1 import doobie._
2 import doobie.implicits._
3
4 val query: ConnectionIO[Int] =
5   sql"select 42".query[Int].unique
```

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 query description is independent of any particular database being queried and of 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, we do this 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 discussed when we learned about integrating with Cats Effect above:

```
1 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 `Runtime#unsafe.run` 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 `Transactor.fromDriverManager` constructor:

```
1 val transactor: Transactor[Task] =
2   Transactor.fromDriverManager[Task](
3     // configuration specific details go here
4   )
```

Notice that we typically have to specify the type of the functional effect we are using, in this case, `Task`, because `Transactor.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.

```
1 val effect: Task[Int] =
2   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 `Transactor.from.DriverManager` constructor Doobie takes care of providing reasonable default implementations of each of these thread pools for us, which is excellent for prototyping applications. However, in production, we often want more fine-grained control over how our queries are executed and use constructors that 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:

```

1 object HikariTransactor {
2   def newHikariTransactor[F[_]](
3     driverClassName: String,
4     url: URL,
5     user: String,
6     password: String,
7     connectionEC: ExecutionContext,
8     transactionEC: Blocker
9   ): Resource[F, HikariTransactor[F]] =
10   ???
11 }
```

Note that the returned type is not a `Transactor` but a `Resource[F, Transactor]`. A `Resource` describes a resource with some necessary *finalization* associated with it, in this case, shutting down the connection pool, and is similar to a scoped ZIO workflow that we will learn about later in this book.

The `zio-interop-cats` package provides a `toScopedZIO` operator on any `Resource` that we can use to convert the `Resource` to a scoped ZIO. From there, we can call the `flatMap` operator to get access to the `HikariTransactor` itself, with the guarantee that the connection pool will automatically close as soon as the scope is closed.

The final step is providing 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:

```

1 import cats.effect.Blocker
2 import doobie.hikari._
3
4 lazy val hikariTransactor: ZIO[
5   Scope,
6   Throwable,
```

```

7   HikariTransactor[Task]
8 ] =
9 for {
10   blocking <- ZIO.blockingExecutor.map(_.asExecutionContext)
11   runtime <- ZIO.runtime[Any]
12   transactor <- HikariTransactor.newHikariTransactor[Task](
13     ??,
14     ??,
15     ??,
16     ??,
17     runtime.runtimeConfig.executor.
18       asExecutionContext,
19       Blocker.liftExecutionContext(blocking)
20     ).toScopedZIO
21 } yield transactor
22
23 lazy val effect: Task[Int] =
24   ZIO.scoped {
25     hikariTransactor.flatMap(query.transact)
26   }

```

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 creating the `Transactor`, passing it to 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 `asExecutionContex` operator on `Executor` to be able to view a ZIO `Executor` as a `scala.concurrent.ExecutionContext` and the `toScopedZIO` operator to convert a Cats Effect `Resource` into a scoped ZIO.

The final result is that we can 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.

#### 4.4.2 Integrating with http4s

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 an HTTP route is actually:

```
1 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 an 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:

```
1 import org.http4s._
2 import org.http4s.dsl.request._
3
4 val helloRoute: HttpRoutes[Task] =
5   HttpRoutes.of[Task] {
6     case GET -> Root / "hello" / name =>
7       Response(Status.Ok)
8         .withBody(s"Hello, $name from ZIO on a server!")
9   }
```

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 various 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:

```
1 import org.http4s.implicits._
2 import org.http4s.server.Router
3
4 val httpApp =
5   Router( "/" -> helloRoute).orNotFound
```

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:

```

1 import org.http4s.server._
2 import org.http4s.server.blaze._
3 import zio.interop.catz.implicits._
4
5 val server: ZIO[Scope, Throwable, Server[Task]] =
6   ZIO.runtime[Any].flatMap { implicit runtime =>
7     val executionContext =
8       runtime.runtimeConfig.executor.asExecutionContext
9     BlazeServerBuilder[Task](executionContext)
10    .bindHttp(8080, "localhost")
11    .withHttpApp(httpApp)
12    .resource
13    .toScopedZIO
14 }
```

There are a couple of things to note here.

First, we had to do a couple of things to ensure that appropriate instances were in scope for all the capabilities that `Task` needs 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 builder of the Blaze server returns a Cats Effect Resource because the server has logic that needs to be executed when shutting it down, so we use the `toScopedZIO` operator to convert it to a scoped ZIO 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 launch our server.

The server we defined above is a scoped ZIO 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 it afterward.

But here, we don't want to do anything with the server other than running it indefinitely until we shut it down. We can use `ZIO.scoped` along with `ZIO.never` to do just that:

```
1 val useServer: Task[Nothing] =  
2   ZIO.scoped(server *> ZIO.never)
```

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!

## 4.5 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 interoperate 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.

## 4.6 Exercises

1. Create a ZIO program that uses Doobie to perform a database operation. Implement a function that inserts a user into a database and returns the number of affected rows. Use the following table structure:

```
1 CREATE TABLE users (
2   id SERIAL PRIMARY KEY,
3   name TEXT NOT NULL,
4   age INT NOT NULL
5 )
```

2. Write a ZIO program that uses lepus to connect to RabbitMQ server and publish arbitrary messages to a queue. Lepus is a purely functional scala client for RabbitMQ. You can find the library homepage here<sup>1</sup>.

---

<sup>1</sup><http://lepus.hnaderi.dev/>

## Chapter 5

# 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 differ from threads. We will learn the basic operations on fibers, including forking, joining, 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.

### 5.1 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 `Runtime#unsafe.run`, 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 ensures 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. Furthermore, since we create the runtime, we can build in logic for how we ensure fairness, how we handle interruption, and so on.

## 5.2 Forking Fibers

The first fundamental operation of fibers is forking them:

```

1 import zio._

2

3 trait ZIO[-R, +E, +A] {
4   def fork: URIO[R, Fiber[E, A]]
5 }
```

Forking creates a new fiber that executes the effect being forked concurrently with the current fiber. No start method is required; as soon as the fiber is created using `ZIO#fork`, it starts executing.

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:

```

1 lazy val doSomething: UIO[Unit]      = ???
2 lazy val doSomethingElse: UIO[Unit] = ???
```

```

3
4 lazy val example1 = for {
5   - <- doSomething
6   - <- doSomethingElse
7 } yield ()

```

On the other hand, if we `fork` `doSomething`, then the order of execution is no longer guaranteed:

```

1 lazy val example2 = for {
2   - <- doSomething.fork
3   - <- doSomethingElse
4 } 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. To order the execution, we can use `Fiber#join` to wait for the forked fiber to complete execution.

## 5.3 Joining Fibers

The second fundamental operation of fibers is joining fibers:

```

1 trait Fiber[+E, +A] {
2   def join: IO[E, A]
3 }

```

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:

```

1 object ForkJoinExample extends ZIOAppDefault {
2   lazy val doSomething: UIO[Unit] =
3     ZIO.debug("do something!").delay(10.seconds)
4
5   lazy val doSomethingElse: UIO[Unit] =
6     ZIO.debug("do something else!").delay(2.seconds)
7
8   override def run = for {
9     - <- ZIO.debug("Starting the program!")
10    fiber <- doSomething.fork
11    - <- doSomethingElse
12    - <- fiber.join // Wait for the fiber to complete
13    - <- ZIO.debug("The fiber has joined!")
14  } yield ()
15 }

```

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 execute other fibers and doesn't waste any resources waiting for the result to be available.

In the previous example, after the ZIO runtime reaches the `fiber.join` line, it semantically blocks the current fiber until the `fiber` completes execution. However, the underlying operating system thread is free to execute other fibers in the meantime.

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:

```

1 import zio._

2

3 object ForkJoinFailedFiberExample extends ZIOAppDefault {
4   lazy val doSomething: ZIO[Any, String, Nothing] =
5     ZIO.debug("do something!").delay(2.seconds) *> ZIO.fail("Boom
6   !")
7
8   override def run =
9     for {
10       _    <- ZIO.debug("Starting the program!")
11       fiber <- doSomething.fork
12       _    <- fiber.join // Fail with the error from the fiber
13       _    <- ZIO.debug("The fiber has joined!")
14     } yield ()
15 }
```

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`:

```

1 trait Fiber[+E, +A] {
2   def await: UIO[Exit[E, A]]
3 }
```

The `Fiber#await` method will return an `Exit` value representing the result of the fiber. This can be either a success with a value, a failure with an error, or an interruption. Using `Exit#foldZIO` we can handle the result of the fiber:

```

1 object ForkAwaitFailedFiberExample extends ZIOAppDefault {
2   lazy val doSomething: ZIO[Any, String, Nothing] =
3     ZIO.debug("do something!").delay(2.seconds) *> ZIO.fail("Boom
4   !")
5
6   override def run =
7     for {
```

```

7   _ <- ZIO.debug("Starting the program!")
8   fiber <- doSomething.fork
9   exit <- fiber.await
10  _ <- exit.foldZIO( // Handle the result of the fiber
11    e => ZIO.debug("The fiber has failed with: " + e),
12    s => ZIO.debug("The fiber has completed with: " + s)
13  )
14  _ <- ZIO.debug("The fiber has joined!")
15 } yield ()
16 }
```

A fiber is a data type representing a running computation whose result is not yet available or is already evaluated. We can also tentatively observe the state of the fiber without waiting for it using `poll`:

```

1 trait Fiber[+E, +A] {
2   def poll: UIO[Option[Exit[E, A]]]
3 }
```

This will return `Some` with an `Exit` value if the fiber has completed execution, or `None` otherwise.

## 5.4 Interrupting Fibers

The final fundamental operation on fibers is interrupting them:

```

1 trait Fiber[+E, +A] {
2   def interrupt: UIO[Exit[E, A]]
3 }
```

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`:

```

1 object InterruptFiberExample extends ZIOAppDefault {
2   lazy val doSomething: ZIO[Any, Nothing, Long] =
3     ZIO
4       .debug("some long running task!")
5       .repeat(Schedule.spaced(2.seconds))
6
7   override def run =
8     for {
9       _ <- ZIO.debug("Starting the program!")
10      fiber <- doSomething.fork
11      _ <- ZIO.sleep(5.seconds)
12      _ <- fiber.interrupt
13    }
```

```
13     _           <- ZIO.debug("The fiber has been interrupted!")
14   } yield ()
15
16 }
```

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 `ZIO.acquireRelease` 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 that 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.

## 5.5 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:

```

1 import zio._
2 import zio.Console._

3
4 object FiberSupervisionParentChildExample extends ZIOAppDefault {

5
6     val child: ZIO[Any, Nothing, Unit] =
7         printLine("Child fiber beginning execution...").orDie *>
8             ZIO.sleep(5.seconds) *>
9                 printLine("Hello from a child fiber!").orDie
10
11    val parent: ZIO[Any, Nothing, Unit] =
12        printLine("Parent fiber beginning execution...").orDie *>
13            child.fork *>
14            ZIO.sleep(3.seconds) *>
15                printLine("Hello from a parent fiber!").orDie
16
17    def run =
18        for {
19            fiber <- parent.fork
20            -      <- ZIO.sleep(1.second)
21            -      <- fiber.interrupt
22            -      <- ZIO.sleep(10.seconds)
23        } yield ()
24 }
```

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.

The issue is easy to observe in this simple example, 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`:

```
1 trait ZIO[-R, +E, +A] {
2   def forkDaemon: URIO[R, Fiber[E, A]]
3 }
```

Any fiber that is forked in the global scope does not have a parent fiber, and therefore, it can live as long as the application is running. It will be terminated when it ends naturally, or it may be interrupted when the application is shut down or when it is explicitly interrupted:

```
1 import zio._
2
3 object ForkDaemonExample extends ZIOAppDefault {
4   val healthChecker: ZIO[Any, Nothing, Long] =
5     ZIO
6       .debug("Checking the health of the system...")
7       .repeat(Schedule.spaced(1.second))
8       .onInterrupt(ZIO.debug("Health checker interrupted!"))
9
10  val parent: ZIO[Any, Nothing, Unit] = {
11    for {
12      - <- ZIO.debug("Parent fiber begins execution...")
13      - <- healthChecker.forkDaemon
14      - <- ZIO.sleep(5.seconds)
15      - <- ZIO.debug("Shutting down the parent fiber!")
16    }
```

```

16     } yield ()
17 }.onInterrupt(ZIO.debug("Parent fiber interrupted"))
18
19 def run =
20   for {
21     fiber <- parent.fork
22     _ <- ZIO.sleep(1.seconds)
23     _ <- fiber.interrupt
24     _ <- ZIO.sleep(10.seconds)
25   } yield ()
26 }
```

In this example, `healthChecker` is forked as a daemon fiber, so it will continue running even after `parent` is interrupted. This is a good pattern for background processes that should run for the application's lifetime. It is worth mentioning that the `bootstrap` layer is also a good place for putting long-running processes that need to be initialized once and shared across the application. By placing such processes in the bootstrap layer, you ensure they are properly started during the application startup and can be managed or referenced by other components as needed.

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 `ZIO#fork` with `ZIO#forkDaemon`.

Another good rule of thumb is to always `Fiber#join` or `Fiber#interrupt` {`Fiber!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 create and handle their acquisition and release.

## 5.6 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`. 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 `onCombinators`.

```

1 import scala.concurrent.ExecutionContext
2
3 trait ZIO[-R, +E, +A] {
4   def onExecutor(executor: => Executor): ZIO[R, E, A]
5 }
```

`onExecutor` 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 `onExecutor` 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:

```

1 lazy val doSomething: UIO[Unit]      = ???
2 lazy val doSomethingElse: UIO[Unit]   = ???

3
4 lazy val executor: Executor = ???

5
6 lazy val effect = for {
7   -<- doSomething.fork
8   -<- doSomethingElse
9 } yield ()

10
11 lazy val result = effect.onExecutor(executor)

```

Both `doSomething` and `doSomethingElse` are guaranteed to be executed on `Executor`

To illustrate the second rule, if we change the example to:

```

1 lazy val executor1: Executor = ???
2 lazy val executor2: Executor = ???

3
4 lazy val effect2 = for {
5   -<- doSomething.onExecutor(executor2).fork
6   -<- doSomethingElse
7 } yield ()

8
9 lazy val result2 = effect2.onExecutor(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.

## 5.7 Conclusion

In this chapter, we have gone into a lot of details 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 understand 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.

## 5.8 Exercises

1. Write a ZIO program that forks two effects, one that prints “Hello” after a two-second delay and another that prints “World” after a one-second delay. Ensure both effects run concurrently.
2. Modify the previous program to print “Done” only after both forked effects have completed.
3. Write a program that starts a long-running effect (e.g., printing numbers every second), then interrupts it after 5 seconds.
4. Create a program that forks an effect that might fail. Use `await` to handle both success and failure cases.
5. Create a program with an uninterruptible section that simulates a critical operation. Try to interrupt it and observe the behavior.
6. Write a program demonstrating fiber supervision where a parent fiber forks two child fibers. Interrupt the parent and observe what happens to the children.
7. Change one of the child fibers in the previous program to be a daemon fiber. Observe the difference in behavior when the parent is interrupted.

# Chapter 6

## 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.

### 6.1 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 implement new combinators, but a good sign in higher-level code is if you are not using fibers directly at all.

### 6.2 Race and ZipPar

The two most basic concurrency and parallelism combinators defined on ZIO are `raceEither` and `zipPar`:

```
1 trait ZIO[-R, +E, +A] { self =>
2   def raceEither[R1 <: R, E1 >: E, B](
3     that: ZIO[R1, E1, B]
4   ): ZIO[R1, E1, Either[A, B]]
5   def zipPar[R1 <: R, E1 >: E, B](
6     that: ZIO[R1, E1, B]
```

```

7 |   ): ZIO[R1, E1, (A, B)]
8 | }
```

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 `ZIO#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 we 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 are completed 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 fail.

`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`, it 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.

## 6.3 Variants of ZipPar

`ZIO#zipPar` can be used to implement a variety of other parallelism combinators. The most commonly used are the `ZIO.foreachPar` and `ZIO.collectAllPar` variants:

```

1 object ZIO {
2   def collectAllPar[R, E, A](
3     in: Iterable[ZIO[R, E, A]]
4   ): ZIO[R, E, List[A]] =
5     ???
6   def foreachPar[R, E, A, B](
7     in: Iterable[A]
```

```

8   )(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
9     ???
10 }

```

The signatures of these are the same as the `ZIO.collectAll` and `ZIO.foreach` variants we discussed previously, except this time, they will perform all the effects in parallel instead of sequentially.

`ZIO.collectAllPar` and `ZIO.foreachPar` have optimized implementations for performance, but conceptually they can both be implemented in terms of `ZIO#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 the `Discard` suffix for when the return value is not needed:

```

1 object ZIO {
2   def collectAllParDiscard[R, E, A](
3     in: Iterable[ZIO[R, E, A]]
4   ): ZIO[R, E, Unit] =
5     ???
6   def foreachParDiscard[R, E, A, B](
7     in: Iterable[A]
8   )(f: A => ZIO[R, E, B]): ZIO[R, E, Unit] =
9     ???
10 }

```

The implementation of these variants can be optimized because it does not need to create a data structure. Use these variants if you don't need the return value; otherwise, the semantics are the same.

In addition, some variants allow performing parallel computations with bounded parallelism:

```

1 object ZIO {
2   def collectAllParN[R, E, A](
3     n: Int
4   )(in: Iterable[ZIO[R, E, A]]): ZIO[R, E, List[A]] =
5     ???
6   def foreachParN[R, E, A, B](
7     n: Int
8   )(in: Iterable[A])(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
9     ???
10 }

```

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 simultaneously. 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 `collectAllParNDiscard` and `foreachParNDiscard` variants of these if you do not need the return value.

## 6.4 Variants of Race

Just as with `zipPar`, there are several variants of the `race` combinator that can be useful in different scenarios. Let's explore some of these variants:

- `ZIO#race` is similar to `ZIO#raceEither`, but it only returns the winning successful value. Whenever you have two effects with conforming types, you can use the `race` combinator instead of `raceEither`.
- `ZIO#raceAll` and `ZIO.raceAll` are similar to `ZIO#race`, but they allow you to race between multiple effects simultaneously, not just between two effects.
- `ZIO#raceFirst` and `ZIO.raceFirst` are similar to `ZIO#race`, but they return the first result to complete, regardless of whether it is a success or a failure.

The `timeout` operators (`timeout`, `timeoutFail`, and `timeoutTo`) can be implemented using the more primitive `race` operator. This demonstrates how more complex concurrent operations can be built from simpler ones. Let's look at how we might implement these:

```

1 | def timeout[R, E, A](
2 |   zio: ZIO[R, E, A],
3 |   duration: => Duration
4 | ): ZIO[R, E, Option[A]] =
5 |   zio.raceEither(ZIO.sleep(duration)).map {
6 |     case Left(value) => Some(value)
7 |     case Right(_)      => None
8 |   }

```

In the implementation of `timeout` above, we race the given effect against an effect that sleeps for the specified duration. If the original effect completes first, we return `Some(value)`; otherwise, we return `None`.

Please note that this implementation is for pedagogical purposes only, and the actual implementation of `timeout` differs from what we have shown here.

The actual interface of `timeout` is as follows:

```

1 | trait ZIO[-R, +E, +A] { self =>

```

```

2 |     def timeout(d: => Duration): ZIO[R, E, Option[A]] = ???
3 |

```

These are the variants of `timeout` that allow you to specify the behavior in the event of a timeout:

```

1 trait ZIO[-R, +E, +A] { self =>
2   def timeoutFail[E1 >: E](e: => E1)(d: => Duration): ZIO[R, E1, A]
3   def timeoutTo[B >: A](b: B)(f: A => B1)(duration: => Duration):
4     ZIO[R, E, B]
}

```

`timeoutFail` will fail with the specified error if a timeout occurs, whereas `timeoutTo` will return the specified value. For example, you can use it like `effect.timeoutTo(defaultValue)(identity)(duration)`.

Since the `timeout` operators are built on top of the `race` variants, we can be confident that the semantics of these operators are correct and that the underlying effect is properly interrupted when a timeout occurs.

## 6.5 Validation Errors

Another useful parallelism combinator in some situations is `ZIO.validatePar`. As discussed above, `ZIO.foreachPar` can only succeed if the effectual function succeeds for each element in the collection and 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 `ZIO.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 `ZIO.validatePar` combinator:

```

1 object ZIO {
2   def validatePar[R, E, A, B](
3     in: Iterable[A]
4     )(f: A => ZIO[R, E, B]): ZIO[R, ::[E], List[B]] = ???
5 }

```

`ZIO.validatePar` takes exactly the same arguments as `ZIO.foreachPar` but returns an effect with a slightly different error type and has different semantics. Unlike `ZIO.foreachPar`, `ZIO.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 `ZIO.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.

```

1 trait List[+A]
2
3 final case class ::[+A](head: A, tail: List[A]) extends List[A]
4 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 `ZIO.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.

## 6.6 Conclusion

In this chapter, we moved from the low-level fiber model to the high-level concurrency combinators that `ZIO` provides. These combinators allow you to express a wide range of parallel and concurrent computations in a safe and efficient way without the need to work directly with fibers.

At the heart of `ZIO`'s approach to concurrency lies a fundamental principle: the power of abstraction. By providing high-level operators like `race` and `zipPar` and their variants, `ZIO` abstracts away the complexities of low-level concurrency primitives, especially when working with fibers.

The `race` and `zip` families of operators encapsulate common concurrent patterns, allowing developers to express complex ideas succinctly. For instance, the simple act of racing two effects with `race` hides a sophisticated dance of fiber management, interruption handling, and resource cleanup. This abstraction allows developers to focus on the “what” rather than the “how”, leading to more readable and maintainable code.

`ZIO`'s concurrency operators strike a delicate balance between power and safety. On one hand, they provide the flexibility to handle a wide range of concurrent scenarios. For example, the interruption semantics of `zipPar` and `foreachPar` ensure that resources aren't

needlessly consumed when part of a parallel computation fails. The bounded parallelism variants (`collectAllParN`, `foreachParN`) provide a mechanism to prevent resource exhaustion. This balance allows developers to harness the power of concurrency without falling into its traps.

Finally, it's worth reflecting on the philosophical shift that ZIO's approach to concurrency represents. By providing these high-level operators, ZIO is not just offering tools; it's promoting a way of thinking about concurrent programming. It encourages developers to think in terms of composable, declarative descriptions of concurrent behavior rather than imperative management of threads and locks. By adopting this mindset, developers can create concurrent systems that are not only more efficient but also more predictable and easier to reason about.

In the next chapter, we will go back to low-level stuff, the fiber supervision model. This low-level exploration will unveil the rules governing the life cycles of ZIO fibers, from creation to termination.

## 6.7 Exercises

1. Implement the `collectAllPar` combinator using `foreachPar`.

```

1 def collectAllPar[R, E, A](
2   in: Iterable[ZIO[R, E, A]]
3 ): ZIO[R, E, List[A]] =
4   ???
```

2. Write a function that takes a collection of ZIO effects and collects all the successful and failed results as a tuple.

```

1 def collectAllParResults[R, E, A](
2   in: Iterable[ZIO[R, E, A]]
3 ): ZIO[R, Nothing, (List[A], List[E])] =
4   ???
```

Hint: Use `ZIO.validatePar` or `ZIO.partitionPar`.

3. Assume you have given the following `fetchUrl` function that fetches a URL and returns a ZIO effect:

```

1 def fetchUrl(url: URL): ZIO[Any, Throwable, String] = ???
```

And you have a list of URLs you want to fetch in parallel. Implement a function that fetches all the URLs in parallel and collects both the successful and failed results. Both successful and failed results should be paired with the URL they correspond to.

```

1 def fetchAllUrlsPar(
2   urls: List[String]
```

```
3 ) : ZIO[Any, Nothing, (List[(URL, Throwable)], List[(URL,
        String)])] =  
4     ???
```

## Chapter 7

# Parallelism and Concurrency: Fiber Supervision in Depth

In Chapter 6, we learned about the fiber model and fiber supervision basics. We found that every fiber has a scope, and that scope is responsible for managing the lifecycle of the fiber. We also learned that each fiber, by default, will be forked in the scope of its parent fiber unless specified otherwise. So, if a parent fiber terminates, all its children's fibers will be interrupted and terminated well. The process in which the runtime system manages the lifecycle of fibers is called fiber supervision. In this chapter, we will dive deeper into fiber supervision and learn how to supervise fibers more fine-grained.

Before we dive into fiber supervision customization, let's go over the fork/join identity law and structured concurrency, which are essential concepts to understand when working with concurrent programs.

### 7.1 Fork/Join Identity Law

The fork/join identity law states that if you take an effect, fork it into smaller effects, and then join them back together, the result should be the same as if we ran the original effect without forking:

```
:| effect.fork.flatMap(fiber => fiber.join) == effect
```

In simpler terms, this law states that introducing trivial concurrency should not change the program's semantics. This law is important because it allows us to reason about the behavior of our concurrent programs like we would with sequential programs.

As every concurrent operator in ZIO is based on fork/join, and ZIO respects the fork/join identity law, it turns out that all parallel variants of ZIO operators should give you the same result as the non-parallel version. For example, the `a.zip(b)` should have the same result as `a.zipPar(b)`, or `ZIO.foreach(events)(processEvent(_))` should give

you the same result as `ZIO.foreachPar(events)(processEvent(_))`.

In ZIO, the fork/join identity law is guaranteed by its support for structured concurrency and fiber supervision model. When you fork a fiber, the new fiber will be supervised by the parent fiber's scope. So we can be sure that all the fibers forked from a parent will not outlive the parent fiber. This preserves the fork/join identity law and makes it easier to reason about the behavior of our concurrent programs.

## 7.2 Structured Concurrency

In traditional concurrency models, tasks can be started and stopped at any time, with threads being created and managed in an ad-hoc manner. This often results in a tangled web of threads that makes visibility and error handling more challenging, potentially leading to unpredictable behavior, resource leaks, and difficulties in managing the lifecycle of concurrent operations.

Structured concurrency, on the other hand, enforces a scope for concurrent tasks, making it easier to manage their lifecycles and handle errors effectively. It organizes and manages the lifecycle of concurrent fibers in a well-defined manner, ensuring that all fibers are started, completed, or terminated predictably. This clarity enhances both the organization and reliability of concurrent programming.

Structured Concurrency is rooted in the principle that the lifecycle of concurrent tasks should be well-defined and predictable, similar to how variables are managed in structured programming through control structures and lexical scoping.

In structured programming, control structures—such as loops, if statements, and function calls—replace the less predictable `goto` statements, resulting in clearer and more maintainable code. This paradigm shift allows for:

1. Make the lifetime of any created variable clearly visible from the program's structure.
2. Dividing the program into smaller, more manageable sections, each with its own scope.

Lexical scoping, also known as static scoping, governs the visibility of variables within these nested structures, ensuring that variables are only accessible within their defined scope. This approach also enhances both the readability and reliability of code by providing a structured and predictable framework for organizing variables:

1. Every variable has a scope.
2. Variables are only accessible within their scope.
3. The scope of a variable is determined by its lexical context.
4. Variables are created and destroyed within their scopes.
5. Nested scopes can access variables from outer scopes but not vice versa.

Let's consider an example in Scala:

```
1 {  
2   val a = 42
```

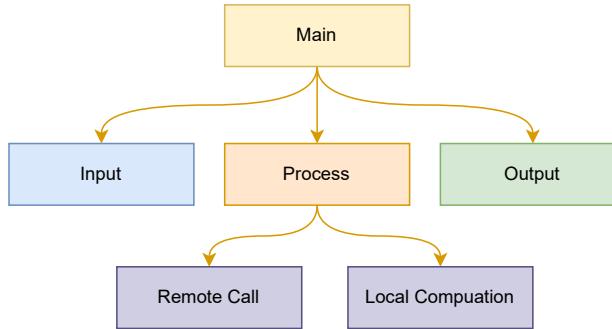


Figure 7.1: Structured Programming

```

3  {
4      val b = a + 1
5      println(b)
6  }
7  println(b) // Compilation error: b is not defined here
8 }
```

In the above example, the variable `b` is only accessible within its scope, and trying to access it outside that scope will result in a compilation error. The scope of `b` is statically visible and well-defined; after the scope ends, the variable `b` is no longer accessible.

In concurrent programming, we would like a similar structure for managing the lifecycle of concurrent tasks. This is where structured concurrency comes into play. We would like to have a clear structure for managing the lifecycle of concurrent tasks, ensuring that all tasks are started, completed, or terminated in a structured and predictable manner.

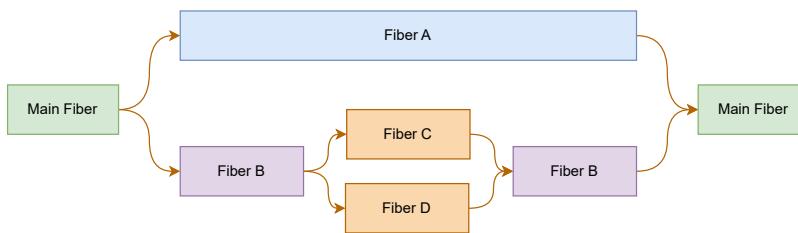


Figure 7.2: Structured Concurrency

In ZIO, structured concurrency is achieved through the fiber supervision model, which ensures all fibers are supervised by their parent fiber's scope. Let's recap the “fiber supervision model” rules we learned in Chapter 6:

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

Having these rules in place ensures that all fibers are managed in a structured and predictable manner. All fibers are guaranteed to terminate when their parent scope is closed, preventing any “dangling” fibers from running and eliminating the risk of resource leaks.

Let's consider an example in ZIO:

```

1 import zio._

2

3 object StructuredConcurrency extends ZIOAppDefault {
4   val parent: UIO[Unit] = {
5     for {
6       _ <- ZIO.debug("Parent fiber beginning execution...")
7       _ <- child("foo").fork
8       _ <- child("bar").fork
9       _ <- ZIO.debug("Parent fiber ending execution...")
10    } yield ()
11  }.onExit(_ => ZIO.debug("Parent fiber exited!"))

12

13  def child(name: String): UIO[Unit] =
14    ZIO(
15      .debug(s"Child fiber $name is running...")
16      .flatMap(_ => ZIO.sleep(5.seconds))
17      .onInterrupt(ZIO.debug(s"Child fiber $name is interrupted!"))
18    )
19
20  def run = for {
21    _ <- ZIO.debug("Main fiber beginning execution...")
22    _ <- parent.fork
23    _ <- ZIO.debug("Main fiber ending execution...")
24  } yield ()
}

```

In this example, we have three fibers: the main fiber, the parent fiber, and two child fibers. The parent fiber is forked in the main fiber's scope, and the child fibers are forked in the parent fiber's scope. When the parent fiber terminates, because of reaching the end of the effect, it will interrupt its child fiber, i.e., the parent fiber. Then, the parent fiber will interrupt the child's fibers. So, when the main fiber terminates, all the fibers forked in its scope, and their children will be interrupted. In this example, even the child fibers are expected to run for 5 seconds; they will be interrupted when the parent fiber is interrupted. This ensures that they will not outlive their parent fiber.

## 7.3 Custom Supervision Strategies

In ZIO, by default, all fibers are scoped to their parent fiber's scope, so they will be supervised by their parent fiber. However, there are cases where you might want to change the scope of a fiber. In this section, you will learn how to have fine-grained control over the supervision of fibers by changing their scope.

### 7.3.1 Forking in the Global Scope

Forking in the global scope allows a fiber to operate independently, without supervision from any parent fiber. This approach is beneficial for creating long-running fibers that must persist regardless of their parent fibers' termination. The `ZIO#forkDaemon` operator facilitates this functionality, as discussed in Chapter 6.

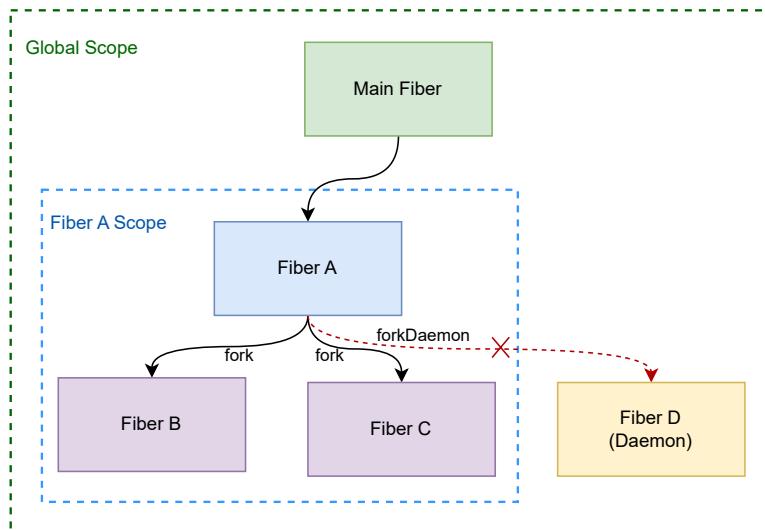


Figure 7.3: `forkDaemon`

In the diagram 7.3, fiber D is depicted as being forked in the global scope, indicating it is not under the supervision of any parent fiber. As a result, fiber D operates independently of the fiber it was forked from, enabling it to continue running for the entire duration of the application's lifecycle.

### 7.3.2 Forking in the Current Local Scope

Up to now, we have learned about two types of scopes: the default scope, where each forked fiber is supervised by its parent fiber, and the global scope, where no parent fiber supervises the forked fiber. These scopes do not have associated contextual data types, and they

are “implicit”. However, there are instances where you might need to fork a fiber within a specific scope. In such cases, ZIO employs a contextual data type called “Scope” to customize the fiber’s lifecycle, which makes the scope of the fiber “explicit”.

Please note that “Scope” is a contextual data type that allows you to manage the lifecycle of resources. We will explore this in detail in chapter 16, where we discuss resource management in ZIO. If you are not yet familiar with “Scope” and resource management in ZIO, you may skip the rest of this section and return to it later.

In ZIO, fibers are treated as resources that are acquired, used, and released. Thus, their lifecycle can be managed using the “Scope” data type. For instance, you can fork a fiber within the current local scope using the `ZIO#forkScoped` operator. This associates the fiber with an “explicit” scope, which is required as contextual data as ZIO environment for the effect:

```

1 trait ZIO[-R, +E, +A] {
2   def forkScoped: ZIO[R with Scope, Nothing, Fiber.Runtime[E, A]]
3     = ???
```

So, when you fork a fiber using the `ZIO#forkScoped`, it gives you an effect that requires the “Scope” as its contextual data. You can close the scope from a region of code by providing the “Scope” as contextual data to the effect, which is possible by using the `ZIO.scoped` operator. The `ZIO.scoped` operator scopes all resources used in the enclosed effect to the lifetime of the enclosing effect, ensuring that their finalizers are run as soon as the effect completes execution, whether by success, failure, or interruption:

```

1 object ZIO {
2   def scoped[R, E, A](zio: => ZIO[Scope with R, E, A]): ZIO[R, E,
3     A] = ???
```

This operator eliminates the “Scope” from the effect’s environment by providing the “Scope” itself and ensuring that all fibers attached to it are interrupted when the scope is eliminated from the environment (i.e., closed).

In the diagram 7.4, fiber D is forked in the current local scope from fiber A, so it no longer belongs to A’s scope. Instead, its scope becomes an explicit part of the ZIO environment. This allows developers to have more granular control over the fiber’s lifecycle. When a section of code is enclosed with the `ZIO.scoped` operator, all fibers forked within that scope are supervised by it. Consequently, they will be automatically interrupted when the scope is closed.

In the following example, fiber A is forked in the current local scope using `ZIO#forkScoped`, and when the scope is closed, fiber A is interrupted. As an exercise, you can change the `task("A").forkScoped` to `task("A").fork` and observe the difference in behavior:

```

1 import zio._
2 import java.time.LocalTime
```

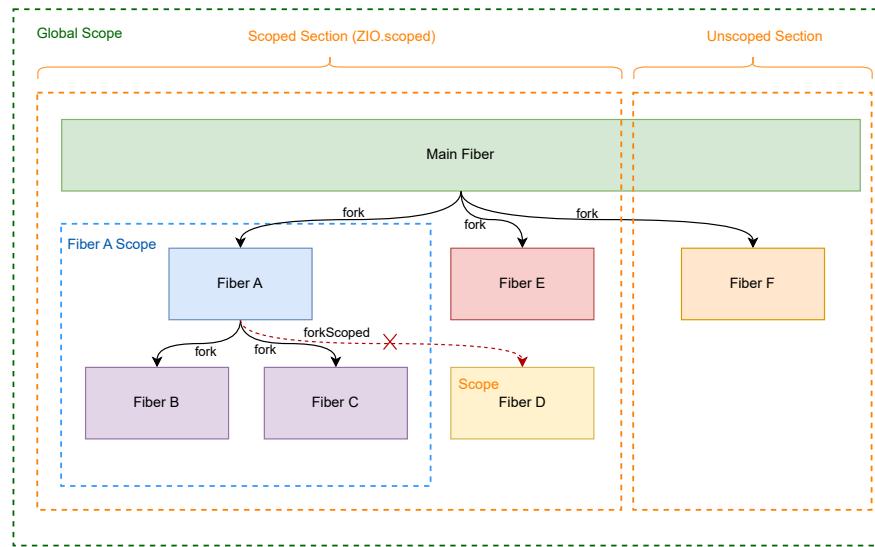


Figure 7.4: ZIO#forkScoped: Fiber D forked in a local scope

```

3 import java.time.format.DateTimeFormatter
4
5 object SimpleForkScopedExample extends ZIOAppDefault {
6   private def getTime =
7     LocalTime.now.format(DateTimeFormatter.ofPattern("HH:mm:ss"))
8
9   def debug(s: String) = ZIO.debug(s"$getTime -- $s")
10
11  def task(name: String): UIO[Unit] =
12    debug(s"Fiber $name is running.")
13    .schedule(Schedule.fixed(1.second))
14    .onInterrupt(debug(s"Fiber $name interrupted!"))
15    .ignore
16
17  def run =
18    for {
19      _ <- ZIO.scoped {
20        for {
21          _ <- task("A").forkScoped
22          _ <- debug("Fiber A forked!")
23          _ <- ZIO.sleep(5.seconds)
24        } yield ()
25      }
26      _ <- debug("Main fiber after closing the scope.")
}

```

```

27     _ <- debug("Sleeping for 3 seconds before exiting.")
28     _ <- ZIO.sleep(3.seconds)
29     _ <- debug("Exiting main fiber.")
30   } yield ()
31 }
```

An important note to remember is that the `ZIO#forkScoped` operator only scopes the fibers forked to a local scope. So, when the scope is closed using the `ZIO.scoped` operator, the fibers forked using the default supervision strategy (using `ZIO#fork`) or those forked in the global scope (using `ZIO#forkDaemon`) will not be interrupted. As a practice, to demonstrate this, let's try another example:

```

1 import zio._
2
3 object ForkScopedExample extends ZIOAppDefault {
4
5   def main =
6     for {
7       _ <- ZIO.scoped {
8         for {
9           fiberA <- taskA.fork
10          _ <- ZIO.log("Fiber A forked!")
11          fiberE <- task("E").fork
12          _ <- debug("Fiber E forked!")
13          _ <- debug("The scoped section on the main fiber
14 is closing after 9 seconds!")
15          _ <- ZIO.sleep(9.seconds)
16        } yield ()
17      }.onExit(e =>
18        debug(s"Scoped section exited with exit status: $e")
19      )
20      _ <- debug(s"Main fiber after closing the scope!")
21      _ <- task("F").fork.flatMap(_.interrupt.delay(11.seconds))
22    } yield ()
23
24   lazy val taskA: ZIO[Scope, Nothing, Unit] = {
25     for {
26       fiberB <- task("B").fork
27       _ <- debug(s"Task B forked")
28       _ <- debug(s"Waiting 3 seconds after forking task B")
29       _ <- ZIO.sleep(3.seconds)
30       fiberC <- task("C").fork
31       _ <- debug(s"Task C forked")
32       _ <- debug(s"Waiting 5 seconds after forking task C")
33       _ <- ZIO.sleep(5.seconds)
34       fiberD <- task("D").forkScoped
```

```

34     -      <- debug(s"Task D forkScoped")
35     -      <- debug("Waiting 7 seconds after forkingScoped task
36           D")
37     -      <- ZIO.sleep(7.seconds)
38   } yield ()
39 }.onExit(e => debug(s"Fiber A exited with exit status: $e"))
40
41 def run = main
42 }
```

In this example, we fork two fibers, A and E, from the main fiber. Fiber A subsequently forks three more fibers: B, C, and D, with fiber D being forked within a local scope. When running this example, it becomes evident that fiber D's lifetime is strictly managed by the local scope in which it was created. This means fiber D's lifespan differs from fibers B and C, which are forked within their parent fiber A's scope. Additionally, it's important to note that after the explicit local scope (the contextual scope) is closed, fiber E continues to run uninterrupted until its parent fiber, the main fiber, terminates.

### 7.3.3 Forking in a Specific Scope

The `ZIO#forkIn` is similar to `ZIO#forkScoped` but allows you to fork a fiber in a specific scope.

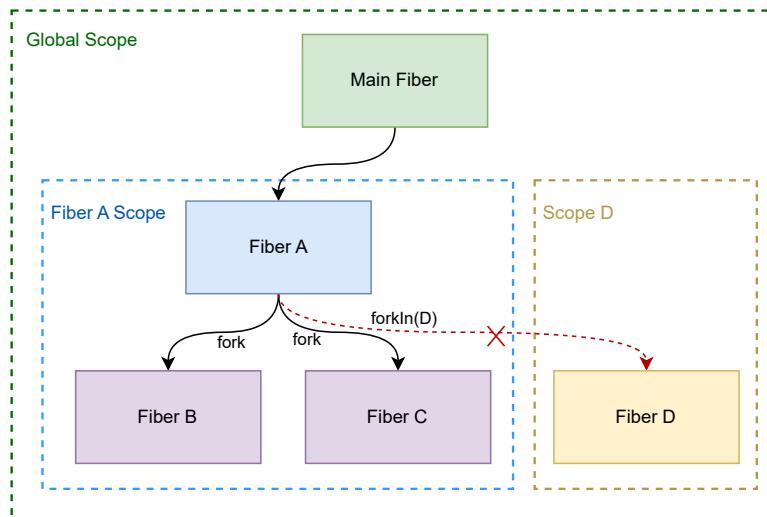


Figure 7.5: `forkIn`

In the example below, the fiber is forked in the outer scope, and when the outer scope is

closed, the fiber is interrupted:

```

1 import zio._

2
3 object ForkInExample extends ZIOAppDefault {
4     def run =
5         ZIO.scoped {
6             for {
7                 outerScope <- ZIO.scope
8                 _ <- ZIO.scoped {
9                     for {
10                         _ <- ZIO
11                             .debug("Task is still active...")
12                             .repeat(Schedule.fixed(1.second))
13                             .forkIn(outerScope)
14                             _ <- ZIO.sleep(3.seconds)
15                             _ <- ZIO.debug("Closing the innermost scope.")
16                     } yield ()
17                 }
18                 _ <- ZIO.sleep(5.seconds)
19                 _ <- ZIO.debug("Closing the outermost scope.")
20             } yield ()
21         }
22 }
```

The following output shows that after closing the inner scope, the fiber remains active, but after closing the outer scope, the fiber is interrupted:

```

1 Fiber is still active...
2 Fiber is still active...
3 Fiber is still active...
4 Closing the innermost scope.
5 Fiber is still active...
6 Fiber is still active...
7 Fiber is still active...
8 Fiber is still active...
9 Fiber is still active...
10 Fiber is still active...
11 Closing the outermost scope.
```

## 7.4 Fire-and-Forget

You might want to fork a fiber and forget its result in several scenarios. This is known as “fire-and-forget”. In ZIO, you can achieve this by forking a fiber and not joining it. This is useful when you want to run a long-lived fiber in the background without waiting for its result:

```

1 val businessLogic =
2   for {
3     - <- doSomething.fork
4     - <- doSomethingElse
5   } yield ()
6
7 val main =
8   for {
9     fiber <- businessLogic.fork
10    - <- fiber.join
11 } yield ()

```

In this example, we fired the `doSomething` and forgot its result. The question is, what happens to the `businessLogic` effect when the `doSomething` fiber is interrupted? In such a case, as we forgot the `doSomething` fiber, we also forgot about any errors it might have produced. This means that if the `doSomething` fiber is interrupted, the `businessLogic` effect will continue to run as if nothing happened.

On the other hand, if the `businessLogic` effect is interrupted due to some error in `doSomethingElse` or by interruption of outer fibers, the `doSomething` fiber will also be interrupted. This is because the `doSomething` fiber is a child of the `businessLogic` effect, and when the parent effect is interrupted, all its children will be interrupted as well.

Another interesting point to note is that if we fire-and-forget the `doSomething` fiber, and it takes longer to complete than the `businessLogic` fiber, the `doSomething` fiber will be interrupted once the `businessLogic` fiber completes. This interruption occurs because the `doSomething` fiber is a child of the `businessLogic` fiber, and the parent is unwilling to wait for the child to complete due to the fire-and-forget nature of the operation.

Please note that if we `join` the `doSomething` fiber inside the `businessLogic`, the propagation of interruptions will be similar to the propagation of errors in structured programming with single-threaded applications. If the child fiber is interrupted, the interruption will be propagated to its parent fiber, and if the parent fiber is interrupted, all of its children will be interrupted as well.

## 7.5 Conclusion

In this chapter, we learned about how to supervise fibers in a more fine-grained manner. We learned that due to ZIO respecting the fork/join identity law, we shouldn't worry about the changing semantics of our programs when we want to refactor our application to use concurrent ZIO operators. Also, we discussed structured concurrency, which has proven to be a safe, expressive, and understandable approach to concurrency. It makes our code more maintainable and reliable and reduces common pitfalls associated with uncontrolled spawning and fiber termination, which cause fiber leakages.

## Chapter 8

# 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 to 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?

### 8.1 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 that 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:

```
1 import scala.io.StdIn
2
3 import zio._
4
5 val greet: UIO[Unit] =
6   for {
```

```

7   name <- ZIO.succeed(StdIn.readLine("What's your name?"))
8     _      <- ZIO.succeed(println(s"Hello, $name!"))
9 } yield ()

```

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 together.

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 `succeed`, the ZIO runtime just sees a single block of Scala code, essentially a thunk `() => A`.

The ZIO runtime has no way of knowing whether 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.

### 8.1.1 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.

```

1 for {
2   fiber <- ZIO.succeed(println("Hello, World!")).fork
3   _      <- fiber.interrupt
4 } yield ()

```

This is very logical: 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 must be careful of cases where we inaccurately assume that an effect will at least begin execution. For example, consider the following program:

```

1 for {
2   ref    <- Ref.make(false)
3   fiber <- ZIO.never.ensuring(ref.set(true)).fork
4   _      <- fiber.interrupt
5   value <- ref.get
6 } 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:

```

1 for {
2   ref    <- Ref.make(false)
3   promise <- Promise.make[Nothing, Unit]
4   fiber   <- (promise.succeed(() *> ZIO.never)
5           .ensuring(ref.set(true)))
6           .fork
7   _      <- promise.await
8   _      <- fiber.interrupt
9   value  <- ref.get
10 } yield value

```

Now, we complete a `Promise` within the forked fiber and await that `Promise` before interrupting, so we 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 about more advanced concurrency operators, so it is good to be aware of it.

### 8.1.2 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.

```

1 val effect: UIO[Unit] =
2   ZIO.succeed {
3     var i = 0
4     while (i < 100000) {
5       println(i)
6       i += 1
7     }
8   }
9
10 for {
11   fiber <- effect.fork
12   _    <- fiber.interrupt
13 } 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 wraps one block of Scala code.

In the example above, it is possible to interrupt the `effect` before it begins execution. But if we do not, it will always print all of the numbers between 0 and 100000, never stopping partway through.

This is normally not a problem because, in most cases, ZIO programs consist of large numbers of smaller statements glued together with operators like `flatMap`, so 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 `ZIO.attemptBlockingCancelable`, `ZIO.attemptBlockingInterrupt`, and `ZIO.asyncInterrupt` to provide our own logic for how ZIO should interrupt the code we are importing.

Let’s see how we could use `ZIO.attemptBlockingCancelable` to support the safe cancellation of a loop like the one above.

The signature of `ZIO.attemptBlockingCancelable` is:

```

1 def attemptBlockingCancelable(
2   effect: => A
3 )(cancel: UIO[Unit]): Task[A] =
4   ???

```

`ZIO.attemptBlockingCancelable` essentially says we are importing an effect into ZIO that may take a long time to complete execution. 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 the effect.

Here is what it looks like in action:

```

1 import java.util.concurrent.atomic.AtomicBoolean
2
3 def effect(canceled: AtomicBoolean): Task[Unit] =
4   ZIO.attemptBlockingCancelable(
5     {
6       var i = 0
7       while (i < 100000 && !canceled.get()) {
8         println(i)
9         i += 1
10      }
11    }
12  )(ZIO.succeed(canceled.set(true)))
13
14 for {
15   ref   <- ZIO.succeed(new AtomicBoolean(false))
16   fiber <- effect(ref).fork
17   _     <- fiber.interrupt
18 } yield ()
```

Now, the `effect` can be safely interrupted even while it is being executed by setting the `AtomicBoolean` to true and breaking out of the `while` loop.

`ZIO.attemptBlockingInterrupt` 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.

`ZIO.asyncInterrupt` is like `ZIO.attemptBlockingCancelable` 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 and assembled to build solutions to larger problems, so in most cases, interruption "just works." However, this is an in-depth chapter on interruption, so we are going over more sophisticated use cases like this and how to handle them.

## 8.2 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*. All ZIO effects are interruptible by default, but some operators may make parts of a ZIO effect uninterruptible.

The basic operators for controlling interruptibility are `ZIO#interruptible` and `ZIO#uninterruptible`:

```

1 val uninterruptible: UIO[Unit] =
2   ZIO.debug("Doing something really important")
3     .uninterruptible
4
5 val interruptible: UIO[Unit] =
6   ZIO.debug("Feel free to interrupt me if you want")
7     .interruptible

```

The `interruptible` in the second statement is technically redundant because effects are interruptible by default, but it is shown for illustrative purposes.

You can think of interruptibility as a “flag” 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 usual, 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 `ZIO#interruptible` and `ZIO#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:

```

1 for {
2   ref  <- Ref.make(false)
3   fiber <- ref.set(true).uninterruptible.fork
4   _      <- fiber.interrupt
5   value <- ref.get
6 } yield value

```

We might think that `value` will always be `true` because we marked `ref.set` as uninterruptible.

But remember that `ZIO#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 discussed above, in which interruption is checked before evaluating each instruction, the runtime will check for interruption before evaluating the first instruction. If the effect has been interrupted, it 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, the `Ref` will often 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 the ZIO Test, we will see that 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 that always works instead of working “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:

```

1 for {
2   ref   <- Ref.make(false)
3   fiber <- ref.set(true).fork.uninterruptible
4   _     <- fiber.interrupt
5   value <- ref.get
6 } yield value

```

This works because when a fiber is forked, it inherits the interruptibility status of its parent at the time 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 we previously saw 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 settings 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 works.

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 a tremendous ability to compose effects together with fine-grained control of interruptibility. It also mirrors how scoping generally works, with inner scopes taking precedence over outer scopes.

The final rule says that forked fibers inherit the settings of their parent fiber by default at the time of forking. 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 on a 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 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.

## 8.3 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, `ZIO#interruptible` and `ZIO#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 interruptions 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 `ZIO#uninterruptible` and `ZIO#interruptible` to `ZIO.uninterruptibleMask` and `ZIO.interruptibleMask`

The signature of `ZIO.uninterruptibleMask` is as follows:

```
1 def uninterruptibleMask[R, E, A](
2   k: ZIO.InterruptStatusRestore => ZIO[R, E, A]
3 ): 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 the following:

```
1 def myOperator[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] =
2   ZIO.uninterruptibleMask { restore =>
3     ZIO.debug("Some work that shouldn't be interrupted") *>
4     restore(zio) <*
5     ZIO.debug("Some other work that shouldn't be interrupted")
6 }
```

The `restore` function here will restore the interruptibility status of `zio` to whatever it was at the point that `ZIO.uninterruptibleMask` was called.

So if `ZIO.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 `ZIO.uninterruptibleMask` operator is often preferable to `ZIO#uninterruptible` because it is, in some sense, less intrusive. If there is some part of our code that needs not to be interrupted, it lets us do that, but it doesn't change anything about the effect provided by the caller.

If we instead used `ZIO#uninterruptible` on our portions of the effect and `ZIO#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 `ZIO#interruptible` or `ZIO#`

`interruptibleMask`.

Generally, `ZIO.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 `ZIO#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 `ZIO.uninterruptibleMask` is a better tool to do that.

The problem with `ZIO#interruptible` is that it can “punch holes” in otherwise uninterruptible regions.

For example, consider the following `delay` combinator:

```

1 import zio._
2 import zio.Duration._

3
4 def delay[R, E, A](
5   zio: ZIO[R, E, A]
6 )(duration: Duration): ZIO[R, E, A] =
7   Clock.sleep(duration).interruptible *> zio
8
9 // 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?

```

1 for {
2   ref      <- Ref.make(false)
3   promise  <- Promise.make[Nothing, Unit]
4   effect    =  promise.succeed(() *> ZIO.never
5   finalizer =  delay(ref.set(true))(1.second)
6   fiber     <- effect.ensuring(finalizer).fork
7   -         <- promise.await
8   -         <- fiber.interrupt
9   value    <- ref.get
10 } 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 spe-

cific scope takes precedence, and `Clock.sleep` is interruptible because we called `ZIO#interruptible` on it.

As a result, even though `delay` is called in an uninterruptible region of the finalizer, `ZIO#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 `ZIO#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 when using `ZIO#interruptible`. At the time of writing, the `ZIO#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 use `ZIO#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:

```

1 | def saferDelay[R, E, A](zio: ZIO[R, E, A])(  

2 |   duration: Duration  

3 | ): ZIO[R, E, A] =  

4 |   Clock.sleep(duration).interruptible.exit *> zio

```

By calling `ZIO#exit`, 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 `saferDelay` 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 `ZIO#interruptible` at all since we don’t absolutely have to here.

## 8.4 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. However, 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:

```

1 for {
2     promise   <- Promise.make[Nothing, Unit]
3     effect    = promise.succeed(() *> ZIO.never
4     finalizer = ZIO.debug("Closing file")
5             .delay(5.seconds)
6     fiber     <- effect.ensuring(finalizer).fork
7     -         <- promise.await
8     -         <- fiber.interrupt
9     -         <- ZIO.debug("Done interrupting")
10 } 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 the “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 guarantees that all finalization logic has been completed when interruption returns. With this as the default, it is also very easy to obtain the opposite behavior, whereas it would be hard to do the other way around.

If we want to interrupt an effect without waiting for the interruption to complete, we can simply fork it.

```

1 for {
2     promise   <- Promise.make[Nothing, Unit]
3     effect    = promise.succeed(() *> ZIO.never
4     finalizer = ZIO.debug("Closing file")
5             .delay(5.seconds)
6     fiber     <- effect.ensuring(finalizer).fork
7     -         <- promise.await
8     -         <- fiber.interrupt.fork
9     -         <- ZIO.debug("Done interrupting")
10 } 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 `ZIO#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 `ZIO#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 `ZIO#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 `ZIO#disconnect` operator, we can solve our problem in a very straightforward way:

```

1 val a: UIO[Unit] =
2   ZIO.never
3     .ensuring(
4       ZIO.debug("Closed A").delay(3.seconds)
5     )
6
7 val b: UIO[Unit] =
8   ZIO.never
9     .ensuring(
10      ZIO.debug("Closed B").delay(5.seconds)
11    )
12     .disconnect
13
14 for {
15   fiber <- (a && b).fork
16   -<- Clock.sleep(1.second)
17   -<- fiber.interrupt
18   -<- ZIO.debug("Done interrupting")
19 } 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.

## 8.5 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 ZIO#zipPar, ZIO#race, ZIO.acquireRelease, and Scope, 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 ZIO.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!

## 8.6 Exercises

- Find the right location to insert ZIO.interruptible to make the test succeed:

```

1 import zio.test.{test, _}
2 import zio.test.TestAspect._

3
4 test("interruptible") {
5   for {
6     ref <- Ref.make(0)
7     latch <- Promise.make[Nothing, Unit]
8     fiber <- ZIO
9       .uninterruptible(latch.succeed(() *> ZIO.never)
10      .ensuring(ref.update(_ + 1))
11      .forkDaemon
12      _ <- Live.live(
13        latch.await *> fiber.interrupt.disconnect.timeout(1.
14        second)
15      )
16      value <- ref.get
17    } yield assertTrue(value == 1)
} @@ nonFlaky

```

- Find the right location to insert ZIO.uninterruptible to make the test succeed:

```

1 import zio.test._
2 import zio.test.TestAspect._


```

```
3
4 test("uninterruptible") {
5   for {
6     ref <- Ref.make(0)
7     latch <- Promise.make[Nothing, Unit]
8     fiber <- {
9       latch.succeed(() *>
10      Live.live(ZIO.sleep(10.millis)) *>
11      ref.update(_ + 1)
12    }.forkDaemon
13    _ <- latch.await *> fiber.interrupt
14    value <- ref.get
15  } yield assertTrue(value == 1)
16 } @@ nonFlaky
```

3. Implement `withFinalizer` without using `ZIO#ensuring`. If the given zio effect has not started, it can be interrupted. However, once it has started, the finalizer must be executed regardless of whether the zio effect succeeds, fails, or is interrupted:

```
1 def withFinalizer[R, E, A](zio: ZIO[R, E, A])(finalizer: UIO
2 [Any]): ZIO[R, E, A] = ???
```

Write a test that checks the proper execution of the `finalizer` in the case the given `zio` effect is interrupted.

Hint: use the `uninterruptibleMask` primitive to implement `withFinalizer`.

4. Implement the `ZIO#disconnect` with the stuff you have learned in this chapter, then compare your implementation with the one in ZIO.

# Chapter 9

## 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` allows us 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 `Ref.Synchronized`, a version of `Ref` that allows us to perform effects while modifying the reference, and `FiberRef`, a version of `Ref` specific to each fiber.

### 9.1 Purely Functional Mutable State

A common question is how we can model mutable states in a purely functional context. Isn't changing mutable states what we are supposed to avoid?

The answer to this, like several problems we have seen before, is that we create a computation that **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:

```
1 import zio._  
2  
3 trait Ref[A] {  
4   def modify[B](f: A => (B, A)): UIO[B]  
5   def get: UIO[A] =  
6     modify(a => (a, a))  
7   def set(a: A): UIO[Unit] =
```

```

8     modify(_ => ((), a))
9     def update(f: A => A): UIO[Unit] =
10    modify(a => ((), f(a)))
11  }
12
13 object Ref {
14   def make[A](a: A): UIO[Ref[A]] =
15   ???
16 }
```

Each combinator on `Ref` returns a ZIO that describes modifying the state of the `Ref`. We can then describe a program that manipulates a state like this:

```

1 def increment(ref: Ref[Int]): UIO[Unit] =
2   for {
3     n <- ref.get
4     _ <- ref.set(n + 1)
5   } yield ()
```

This method takes 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 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:

```

1 trait Ref[A] {
2   def modify[B](f: A => (B, A)): UIO[B]
3   def get: UIO[A] =
4     modify(a => (a, a))
5   def set(a: A): UIO[Unit] =
6     modify(_ => ((), a))
7   def update(f: A => A): UIO[Unit] =
8     modify(a => ((), f(a)))
9 }
10
11 object Ref {
12   def make[A](a: A): UIO[Ref[A]] =
13   ???
14   def unsafeMake[A](a: A): Ref[A] =
15   ???
16 }
```

We could then write a simple program that creates and increments two references:

```

1 lazy val ref1: Ref[Int] = Ref.unsafeMake(0)
2 lazy val ref2: Ref[Int] = Ref.unsafeMake(0)
3
4 lazy val result = for {
5   _ <- ref1.update(_ + 1)
6   _ <- ref2.update(_ + 1)
7   l <- ref1.get
8   r <- ref2.get
9 } yield (l, r)

```

This program creates two references and increments each of them by one, returning (1, 1).

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:

```

1 lazy val makeRef: Ref[Int] = Ref.unsafeMake(0)
2
3 lazy val ref1: Ref[Int] = makeRef
4 lazy val ref2: Ref[Int] = makeRef
5
6 lazy val result = for {
7   _ <- ref1.update(_ + 1)
8   _ <- ref2.update(_ + 1)
9   l <- ref1.get
10  r <- ref2.get
11 } 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 a mutable state shows that even the creation of a 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:

```

1 lazy val makeRef1: UIO[Ref[Int]] = Ref.make(0)
2 lazy val makeRef2: UIO[Ref[Int]] = Ref.make(0)
3
4 lazy val result = for {
5   ref1 <- makeRef1
6   ref2 <- makeRef2
7   _ <- ref1.update(_ + 1)
8   _ <- ref2.update(_ + 1)
9   l <- ref1.get
10  r <- ref2.get
11 } yield (l, r)

```

And here is the refactored program:

```

1 lazy val makeRef: UIO[Ref[Int]] = Ref.make(0)
2
3 lazy val makeRef1: UIO[Ref[Int]] = makeRef
4 lazy val makeRef2: UIO[Ref[Int]] = makeRef
5
6 lazy val result = for {
7   ref1 <- makeRef1
8   ref2 <- makeRef2
9   _    <- ref1.update(_ + 1)
10  _   <- ref2.update(_ + 1)
11  l   <- ref1.get
12  r   <- ref2.get
13 } yield (l, r)

```

Now, `make` only describes 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 a mutable state in an effect constructor.

## 9.2 Purely Functional Equivalent of an Atomic Reference

Not only does `Ref` allow us to describe mutable states in a purely functional way, but it also allows us to do so in a way that is safe for concurrent access. So far, we have described how to model a mutable state in a purely functional way, but we don't need 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:

```

1 trait Var[A] {
2   def get: UIO[A]
3   def set(a: A): UIO[Unit]
4   def update(f: A => A): UIO[Unit]
5 }
6
7 object Var {
8   def make[A](a: A): UIO[Var[A]] =
9     ZIO.succeed {
10    new Var[A] {
11      var a0 = a
12      def get: UIO[A] =
13        ZIO.succeed(a0)
14      def set(a: A): UIO[Unit] =
15        ZIO.succeed {
16          a0 = a

```

```

17         ()
18     }
19     def update(f: A => A): UIO[Unit] =
20       ZIO.succeed { a0 = f(a0) }
21   }
22 }
23 }
```

This allows us to describe the state in a purely functional way. But what happens if we want to share that state across multiple fibers, like in the program below?

```

1 for {
2   variable <- Var.make(0)
3   _ <- ZIO.foreachParDiscard((1 to 10000).toList) { _ =>
4     variable.update(_ + 1)
5   }
6   result <- variable.get
7 } yield result
```

The result of this program will be indeterminate. 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 updates.

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`.

```

1 import java.util.concurrent.atomic.AtomicReference
2
3 trait Ref[A] {
4   def modify[B](f: A => (B, A)): UIO[B]
5   def get: UIO[A] =
6     modify(a => (a, a))
7   def set(a: A): UIO[Unit] =
8     modify(_ => (((), a)))
9   def update(f: A => A): UIO[Unit] =
10    modify(a => (((), f(a))))
11 }
12
13 object Ref {
14   def make[A](a: A): UIO[Ref[A]] =
15     ZIO.succeed {
16       new Ref[A] {
17         val atomic = new AtomicReference(a)
```

```

18     def modify[B](f: A => (B, A)): UIO[B] =
19       ZIO.succeed {
20         var loop = true
21         var b: B = null.asInstanceOf[B]
22         while (loop) {
23           val current = atomic.get
24           val tuple   = f(current)
25           b = tuple._1
26           loop = !atomic.compareAndSet(current, tuple._2)
27         }
28         b
29       }
30     }
31   }
32 }
```

`Ref` is internally backed by an `AtomicReference`, a mutable data structure safe for concurrent access. The `AtomicReference` is not exposed directly, so the only way to access it is through the combinators we provide, which all return effects.

In the implementation of `Ref#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 `AtomicReference`, we are just wrapping the state modification in an effect constructor and implementing the `Ref#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 simultaneously.

```

1 for {
2   ref <- Ref.make(0)
3   _ <- ZIO.foreachParDiscard((1 to 10000).toList) { _ =>
4     ref.update(_ + 1)
5   }
6   result <- ref.get
7 } yield result
```

This will now return 10000 every time it is run.

## 9.3 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.

In the example above, we could safely call `Ref#update` to increment the value of the `Ref` because the update function was performed atomically. From the caller's perspective, `Ref#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 guarantee doesn't apply if we combine more than one combinator on `Ref`. For example, the following is not correct:

```

1 | for {
2 |   ref <- Ref.make(0)
3 |   - <- ZIO.foreachParDiscard((1 to 10000).toList) { _ =>
4 |     ref.get.flatMap(n => ref.set(n + 1))
5 |   }
6 |   result <- ref.get
7 | } yield result

```

This is the same program as above, except 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 does not. So, the program above will be indeterminate 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 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 pieces of state that need 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. We can implement many concurrency combinators and data structures with just `Ref` and the `Promise` data type, which we will describe in the next chapter.

If you need 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.

## 9.4 Ref.Synchronized for Evaluating Effects while Updating

Another limitation of Ref is that we can't perform effects inside the Ref#modify operation. Going back to the interface for Ref, the signature of Ref#modify is:

```
1 trait Ref[A] {
2   def modify[B](f: A => (B, A)): UIO[B]
3 }
```

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 console or allocating a 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 because computing the new value has no side effects. We can do it as many times as we want, and it will be exactly the same as if we had done it a single time. We can increment an integer once or a hundred times, and it will be exactly the same, except we have 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.printLine("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 Ref#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 Ref#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:

```
1 import zio.Console._
2
3 def updateAndLog[A](ref: Ref[A])(f: A => A): URIO[Any, Unit] =
4   ref.modify { oldValue =>
5     val newValue = f(oldValue)
6     ((oldValue, newValue), newValue)
7   }.flatMap { case (oldValue, newValue) =>
8     Console.printLine(s"updated $oldValue to $newValue").orDie
9   }
```

For cases where we really do need to perform effects within the modify operation, ZIO provides another data type called Ref.Synchronized. The interface for Ref.

Synchronized is the same as the one for Ref, except we can now perform effects within the Ref.Synchronized#modify operation.

```

1 trait Synchronized[A] {
2   def modifyZIO[R, E, B](f: A => ZIO[R, E, (B, A)]): ZIO[R, E, B]
3 }
```

The guarantees of Ref.Synchronized are similar to those of Ref. Each operation on Ref.Synchronized is guaranteed to be executed 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, Ref.Synchronized is implemented using a Semaphore to ensure that only one fiber can interact with the reference simultaneously. We'll learn more about ZIO's implementation of a semaphore later, but for now, you can 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 Ref.Synchronized is when we must 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.

```

1 import zio._
2
3 trait RefCache[K, V] {
4   def getOrElseCompute(k: K)(f: K => V): UIO[Ref[V]]
5 }
6
7 object RefCache {
8   def make[K, V]: UIO[RefCache[K, V]] =
9     Ref.Synchronized.make(Map.empty[K, Ref[V]]).map { ref =>
10       new RefCache[K, V] {
11         def getOrElseCompute(k: K)(f: K => V): UIO[Ref[V]] =
12           ref.modifyZIO { map =>
13             map.get(k) match {
14               case Some(ref) =>
15                 ZIO.succeed((ref, map))
16               case None =>
17                 Ref.make(f(k)).map(ref => (ref, map + (k -> ref)))
18             }
19           }
20       }
21     }
22 }
```

```

21     }
22 }
```

We want the `getOrElseCompute` 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 `Ref .Synchronized` here.

`Ref .Synchronized` is never going to be as fast as using `Ref` because of the additional overhead required to ensure that only a single fiber is interacting with the reference simultaneously. But sometimes, we need that additional power. When we do use `Ref .Synchronized`, the key is to do as little work as possible within the `modify` operation of `Ref .Synchronized`. 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.

## 9.5 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:

```

1 def make[A] (
2   initial: A,
3   fork: A => A,
4   join: (A, A) => A
5 ): UIO[FiberRef[A]] =
6   ???
```

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 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:

```
1 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 would be empty, and if it forked five other fibers, there would 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 arbitrary strings to be logged. We could imagine using a more complex data type instead of `String` to support logging in a more structured format:

```
1 type Log = Tree[Chunk[String]]
```

We then need to specify implementations for the `initial`, `fork`, and `join` parameters to make:

```
1 val loggingRef: ZIO[Scope, Nothing, FiberRef[Log]] =
2   FiberRef.make[Log](
3     Tree(Chunk.empty, List.empty),
4     _ => Tree(Chunk.empty, List.empty),
5     (parent, child) => parent.copy(tail = child :: parent.tail)
6   )
```

When we initially created 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:

```
1 def log(ref: FiberRef[Log])(string: String): UIO[Unit] =
2   ref.update(log => log.copy(head = log.head :+ string))
3
4 for {
5   ref <- loggingRef
6   left = for {
7     a <- ZIO.succeed(1).tap(_ => log(ref)("Got 1"))
8     b <- ZIO.succeed(2).tap(_ => log(ref)("Got 2"))
9   } yield a + b
10  right = for {
11    c <- ZIO.succeed(1).tap(_ => log(ref)("Got 3"))
12    d <- ZIO.succeed(2).tap(_ => log(ref)("Got 4"))
13  } yield c + d
14  fiber1 <- left.fork
```

```

15   fiber2 <- right.fork
16   -      <- fiber1.join
17   -      <- fiber2.join
18   log    <- ref.get
19   -      <- Console.printLine(log.toString)
20 } yield ()

```

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 lets us see what happened sequentially on each fiber as well as 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 `FiberRef#locally`:

```

1 trait FiberRef[A] {
2   def locally[R, E, A](value: A)(zio: ZIO[R, E, A]): ZIO[R, E, A]
3 }

```

This sets the `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 configuration information, such as the logging level. The `FiberRef#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 `FiberRef#locally` operator also composes, so you could, for example, use `FiberRef#locally` in one part of your application to turn off logging but then inside that part of your application call `FiberRef#locally` again in one particular area to turn logging back on:

```

1 lazy val doSomething: ZIO[Any, Nothing, Unit]      = ???
2 lazy val doSomethingElse: ZIO[Any, Nothing, Unit] = ???
3
4 for {
5   logging <- FiberRef.make(true)
6   -      <- logging.locally(false) {
7     for {

```

```
8      - <- doSomething
9      - <- logging.locally(true)(doSomethingElse)
10     } yield ()
11   }
12 } yield ()
```

Using a `FiberRef` in the environment can be a particularly powerful pattern for supporting locally scoped changes to configuration parameters like this.

## 9.6 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 `Ref.Synchronized` wherever possible.

`Ref` is significantly more performant because it is implemented directly in terms of compare and swap operations versus a `Ref.Synchronized` which has to force other fibers to semantically block while an effectual update is being performed. In most cases, the code that performs effects within the `modify` operation can be refactored to return an updated state that 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 that it looks like that. 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 discuss `Promise`. While `Ref` allows sharing some piece of state between fibers, it doesn't provide any way of 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!

## 9.7 Exercises

1. Write a simple Counter with the following interface that can be incremented and decremented concurrently:

```

1 trait Counter {
2   def increment: UIO[Long]
3   def decrement: UIO[Long]
4   def get: UIO[Long]
5   def reset: UIO[Unit]
6 }
```

2. Implement a bounded queue using Ref that has a maximum capacity that supports the following interface:

```

1 trait BoundedQueue[A] {
2   def enqueue(a: A): UIO[Boolean] // Returns false if queue
3   // is full
4   def dequeue: UIO[Option[A]] // Returns None if queue
5   // is empty
6   def size: UIO[Int]
7   def capacity: UIO[Int]
8 }
```

3. Write a CounterManager service that manages multiple counters with the following interface:

```

1 type CounterId = String
2
3 trait CounterManager {
4   def increment(id: CounterId): UIO[Int]
5   def decrement(id: CounterId): UIO[Int]
6   def get(id: CounterId): UIO[Int]
7   def reset(id: CounterId): UIO[Unit]
8   def remove(id: CounterId): UIO[Unit]
9 }
```

The corresponding counter should be created and initialized with zero if it does not exist.

Hint: Use `Ref.Synchronized` to be able to perform an effectful operation while updating the underlying mutable reference.

4. Implement a basic log renderer for the `FiberRef[Log]` we have defined through the chapter. It should show the hierarchical structure of fiber logs using indentation:
  - Each level of nesting should be indented by two spaces from the previous one.
  - The log entries for each fiber should be shown on separate lines
  - Child fiber logs should be shown under their parent fiber

```
1 trait Logger {  
2   def log(message: String): UIO[Unit]  
3 }  
4  
5 object Logger {  
6   def render(ref: Log): String = ???  
7 }
```

Example output:

```
1 Got foo  
2   Got 1  
3     Git 2  
4 Got bar  
5   Got 3  
6     Got 4
```

5. Change the log model and use a more detailed one instead of just a `String`, so that you can implement an advanced log renderer that adds timestamps and fiber IDs, like the following output:

```
1 [2024-01-01 10:00:01] [fiber-1] Child foo  
2   [2024-01-01 10:00:02] [fiber-2] Got 1  
3     [2024-01-01 10:00:03] [fiber-2] Got 2  
4 [2024-01-01 10:00:01] [fiber-1] Child bar  
5   [2024-01-01 10:00:02] [fiber-3] Got 3  
6     [2024-01-01 10:00:03] [fiber-3] Got 4
```

Hint: You can use the following model for the log entry:

```
1 case class LogEntry(  
2   timestamp: java.time.Instant,  
3   fiberId: String,  
4   message: String  
5 )
```

6. Create a more advanced logging system that supports different log levels. It also should support regional settings for log levels so that the user can change the log level for a specific region of the application:

```
1 trait Logger {  
2   def log(message: String): UIO[Unit]  
3   def withLogLevel[R, E, A](level: LogLevel)(zio: ZIO[R, E,  
4     A]): ZIO[R, E, A]  
5 }
```

## Chapter 10

# Concurrent Structures: Promise - Work Synchronization

In the last chapter, we learned about `Ref`, one of the fundamental data structures 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 them. 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`. However, this functionality may often be insufficient 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, many fibers run concurrently, delivering messages to the mailbox, so we can't know in advance which fiber will provide the next message.

These are the use cases for which `Promise` shines. A simplified version of the interface of `Promise` is as follows:

```
1 import zio._
2
3 trait Promise[E, A] {
4   def await: IO[E, A]
5   def fail(e: E): UIO[Boolean]
6   def succeed(a: A): UIO[Boolean]
7 }
```

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 `Promise#succeed` to fill it with a value or `Promise#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 `Promise#await` combinator. `Promise#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:

```

1 for {
2     promise <- Promise.make[Nothing, Unit]
3     left <-
4         (Console.print("Hello, ") *> promise.succeed()).fork
5     right <- (promise.await *> Console.print(" World!")).fork
6     _      <- left.join *> right.join
7 } yield ()
```

Without the `Promise`, this program would exhibit non-determinism. “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. On the other hand, the `left` fiber 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 took 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, we aim to use tools like `Promise` to impose the minimum amount of sequencing necessary for correctness, otherwise allowing the program to be as concurrent as possible for performance.

## 10.1 Various Ways of Completing Promises

The description of a `Promise` above is incomplete. A completed promise 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:

```

1 trait Promise[E, A] {
2   def die(t: Throwable): UIO[Boolean]
3   def done(e: Exit[E, A]): UIO[Boolean]
4   def failCause(e: Cause[E]): UIO[Boolean]
5 }
```

These all allow completing promises with other types of effects. `Promise#die` allows completing a promise with an effect that dies with the specified `Throwable`. `Promise#done` allows completing a promise with an effect that returns the specified `Exit` value. `Promise#failCause` allows completing a promise with an effect that dies with the specified `Cause`.

This variety of methods makes sense because calling `await` on a `Promise` returns the computation inside the `Promise` when 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 `Promise#complete` and `Promise#completeWith` combinators:

```

1 trait Promise[E, A] {
2   def complete(io: IO[E, A]): UIO[Boolean]
3   def completeWith(io: IO[E, A]): UIO[Boolean]
4 }
```

`Promise#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 `Promise#await`. On the other hand, `Promise#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:

```

1 import zio._
2
3 import scala.util.Random
4
5 val randomInt: UIO[Int] =
6   ZIO.succeed(Random.nextInt())
7
8 val complete: UIO[(Int, Int)] =
9   for {
10     p <- Promise.make[Nothing, Int]
11     _ <- p.complete(randomInt)
```

```

12     l <- p.await
13     r <- p.await
14 } yield (l, r)
15
16 val completeWith: UIO[(Int, Int)] =
17   for {
18     p <- Promise.make[Nothing, Int]
19     _ <- p.completeWith(randomInt)
20     l <- p.await
21     r <- p.await
22   } yield (l, r)

```

In `Promise#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 `Promise#completeWith`, the `Promise` is completed with the `randomInt` effect itself. Calling `Promise#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 differ from `r`.

The most common use case of promises is to complete them with values using combinators such as `Promise#succeed` and `Promise#fail`, so generally, you will not have to worry about this distinction. But when you are completing promises with effects, it is helpful to remember that `Promise#completeWith` is more efficient than `Promise#complete`. So if `Promise#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 `Promise#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. This may not matter in many situations 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` has already been completed.

## 10.2 Waiting on a Promise

We have already seen `Promise#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:

```

1 trait Promise[E, A] {
2   def await: IO[E, A]
3   def isDone: UIO[Boolean]
4   def poll: UIO[Option[IO[E, A]]]

```

5 }

As described above, `Promise#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 `Promise#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 `Promise#await`.

It is also possible to poll for completion of the `Promise` using `Promise#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, `Promise#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, `Promise#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 a more convenient method.

## 10.3 Promises and Interruption

Promises support ZIO's interruption model. This means that if a `Promise` is completed with an interrupted effect, all fibers waiting on that `Promise` will also be immediately interrupted.

In general, this isn't something you have to worry about, but it "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 `Promise#interrupt` and `Promise#interruptAs` combinators:

```
1 trait Promise[E, A] {
2   def interrupt: UIO[Boolean]
3   def interruptAs(fiberId: FiberId): UIO[Boolean]
4 }
```

`Promise#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 `Promise#interrupt`. If you want to specify a different fiber as the interrupter, you can use `Promise#interruptAs`, which allows you to specify the identifier of a different fiber as the one responsible for the interruption. `Promise#interruptAs` can be helpful 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 it should not normally be necessary in user code.

## 10.4 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, 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:

```

1 trait Cache[-K, +E, +V] {
2   def get(key: K): IO[E, V]
3 }
4
5 object Cache {
6   def make[K, R, E, V](
7     lookup: K => ZIO[R, E, V]
8   ): URIO[R, Cache[K, E, V]] =
9   ???
10 }
```

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 associated value, 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:

```

1 import zio._
2
```

```

3 trait Cache[-K, +E, +V] {
4     def get(key: K): IO[E, V]
5 }
6
7 object Cache {
8     def make[K, R, E, V](
9         lookup: K => ZIO[R, E, V]
10    ): URIO[R, Cache[K, E, V]] =
11     for {
12         r   <- ZIO.environment[R]
13         ref <- Ref.make[Map[K, Promise[E, V]]](Map.empty)
14     } yield new Cache[K, E, V] {
15         def get(key: K): IO[E, V] =
16             Promise.make[E, V].flatMap { promise =>
17                 ref.modify { map =>
18                     map.get(key) match {
19                         case Some(promise) => Right(promise), map
20                         case None           => Left(promise), map + (key ->
21                             promise)
22                     }
23                     }.flatMap {
24                         case Left(promise) =>
25                             lookup(key)
26                             .provideEnvironment(r)
27                             .intoPromise(promise) *> promise.await
28                         case Right(promise) =>
29                             promise.await
30                     }
31                 }
32             }
}

```

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 the `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 Ref#modify to atomically modify the Map backing the Cache. It is important to modify the Ref atomically here because if two fibers call get on the Cache simultaneously with the same key, we want to ensure 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 Ref#modify result 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 ourselves, which we do by calling the lookup function with the key and providing the environment. We use the ZIO#intoPromise combinator on ZIO, a shorthand for Promise.complete(io).

We want to use Promise#complete versus Promise#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 various 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 data structures, we got several important guarantees, like atomic updates and waiting without blocking “out of the box”.

## 10.5 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 ways to complete and wait on promises and 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 discuss 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.

## 10.6 Exercises

1. Implement a countdown latch using `Ref` and `Promise`. A countdown latch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. The latch is initialized with a given count, and the count is decremented each time an operation completes. When the count reaches zero, all waiting threads are released:

```

1 trait CountDownLatch {
2   def countDown: UIO[Unit]
3   def await: UIO[Unit]
4 }
5
6 object CountDownLatch {
7   def make(n: Int): UIO[CountDownLatch] = ????
8 }
```

2. Similar to the previous exercise, you can implement `CyclicBarrier`. A cyclic barrier is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. Once all threads have reached the barrier, they can proceed:

```

1 trait CyclicBarrier {
2   def await: UIO[Unit]
3   def reset: UIO[Unit]
4 }
5
6 object CyclicBarrier {
7   def make(parties: Int): UIO[CyclicBarrier] = ????
8 }
```

3. Implement a concurrent bounded queue using `Ref` and `Promise`. It should support enqueueing and dequeuing operations, blocking when the queue is full or empty:

```

1 trait Queue[A] {
2   def offer(a: A): UIO[Unit]
3   def take: UIO[A]
4 }
5
6 object Queue {
7   def make[A](capacity: Int): UIO[Queue[A]] = ????
8 }
```

# Chapter 11

## 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 the 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.

### 11.1 Queues as Generalizations of Promises

Like a `Promise`, a `Queue` allows fibers to suspend waiting to offer a value to the `Queue` or take 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 is perfect for allowing one fiber to wait on the result of some work and then do 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 distributed 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 to distribute work among multiple fibers by having them each take values from

a queue representing some work that needs to be done.

## 11.2 Offering and Taking Values from a Queue

The basic operations of a queue are `Queue#offer` and `take`:

```

1 import zio._
2
3 trait Queue[A] {
4   def offer(a: A): UIO[Boolean]
5   def take: UIO[A]
6 }
```

`offer` adds a value to the queue. `take` removes a value from the queue, suspending until there is a value to take.

Here is a simple example of using `offer` and `take` with a `Queue`:

```

1 for {
2   queue <- Queue.unbounded[Int]
3   -    <- ZIO.foreach(List(1, 2, 3))(queue.offer)
4   value <- ZIO.collectAll(ZIO.replicate(3)(queue.take))
5 } yield value
```

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 `Queue.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.

`Queue#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:

```

1 for {
2   queue <- Queue.unbounded[Int]
3   -    <- queue.take
4   -    .tap(n => Console.printLine(s"Got $n!"))
5   -    .forever
6   -    .fork
7   -    <- queue.offer(1)
8   -    <- queue.offer(2)
9 } yield ()
```

Typically, printing something forever would result in our console being filled with output. But here, the only things that will be printed to the console are Got 1 and Got 2 because those were the only values offered in 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:

```

1 import zio.Clock._
2 import zio.Console._

3
4 def work(id: String)(n: Int): URIO[Clock with Console, Unit] =
5   Console.printLine(s"fiber $id starting work $n").orDie *>
6     ZIO.sleep(1.second) *>
7     Console.printLine(s"fiber $id finished with work $n").orDie
8
9 for {
10   queue <- Queue.unbounded[Int]
11   -    <- queue.take.flatMap(work("left")).forever.fork
12   -    <- queue.take.flatMap(work("right")).forever.fork
13   -    <- ZIO.foreachDiscard(1 to 10)(queue.offer)
14 } yield ()
```

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 it 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.

## 11.3 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 removed from the queue, we can continue adding elements to the end of the queue. Unbounded queues are more straightforward, 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 use a Queue as a buffer in a streaming application, if the upstream producer runs faster than the downstream consumer, 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

### 11.3.1 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 `Queue#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 take values from the queue semantically block until there are elements in the queue, and fibers offer 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 when creating a Queue using the bounded constructor.

```

1 for {
2   queue <- Queue.bounded[String](2)
3   - <- queue
4     .offer("ping")
5     .tap(_ => Console.printLine("ping"))
6     .forever
7     .fork
8 } yield ()
```

We construct a bounded Queue using the `Queue.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 has limitations. With it, 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 in which stock prices are offered to the queue, and workers take those prices and perform analytics based on them. If new prices are being offered faster than the workers can consume, then the upstream process will suspend 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.

### 11.3.2 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:

```

1 for {
2   queue <- Queue.sliding[Int](2)
3   -    <- ZIO.foreach(List(1, 2, 3))(queue.offer)
4   a    <- queue.take
5   b    <- queue.take
6 } yield (a, b)
```

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 `Queue#take` is called, the values 2 and 3 will be returned from the Queue.

### 11.3.3 Dropping Strategy

The final strategy for adding elements to a Queue that is already full is the **Dropping** strategy. The dropping strategy is like the Sliding strategy in that `Queue#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 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:

```

1 for {
2   queue <- Queue.dropping[Int](2)
3   -    <- ZIO.foreach(List(1, 2, 3))(queue.offer)
4   a    <- queue.take
5   b    <- queue.take
6 } yield (a, b)

```

This is the same example as before, except we created a dropping Queue instead of a sliding one this time. Once again, `Queue#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.

## 11.4 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 `Queue#take` many times throughout 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.

On the other hand, we have seen that `Queue#offer` 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 no capacity for it.

We are also now in a better position to understand the return type of `UIO[Boolean]` in the signature of `Queue#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 `Queue#offer` completes execution, the value will always have been successfully offered to the Queue. In contrast, with dropping queues, `Queue#offer` could return `false` if there is no 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 others, either

for convenience or particular use cases.

#### 11.4.1 Variants of Offer and Take

```

1 trait Queue[A] {
2   def offerAll(as: Iterable[A]): UIO[Boolean]
3   def poll: UIO[Option[A]]
4   def takeAll: UIO[Chunk[A]]
5   def takeUpTo(max: Int): UIO[Chunk[A]]
6   def takeBetween(min: Int, max: Int): UIO[Chunk[A]]
7 }
```

First, several variants of `offer` and `take` exist to deal with multiple values.

`Queue#offerAll` offers all of the values in the specified collection to the Queue in order. If there is insufficient capacity in a bounded Queue with back pressure, then this will immediately offer the values there is a 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.

`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.

`Queue#takeAll` takes all of the values in the Queue. This method will return immediately, potentially with an empty Chunk if there are no values in the Queue. `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 chunk if the Queue is empty. Finally, `Queue#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.

#### 11.4.2 Metrics on Queues

A couple of methods are available to introspect on the status of the Queue:

```

1 trait Queue[A] {
2   def capacity: Int
3   def size: UIO[Int]
4 }
```

`Queue#capacity` returns the maximum capacity of the Queue. For an unbounded Queue, this will return `Int.MaxValue` (technically, unbounded queues are implemented

as dropping queues with a capacity equal to `Int.MaxValue`). Notice that `capacity` returns an `Int` that is not wrapped in an effect because a `Queue`'s capacity is a constant value specified when the `Queue` is created and never changes.

`Queue#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.

### 11.4.3 Shutting Down Queues

Several methods are defined on `Queue` dealing with shutting down queues. Because multiple fibers may be waiting to either offer or take values, we want to have a way to interrupt those fibers if the `Queue` is no longer needed. `Queue#shutdown` gives us that tool.

```

1 trait Queue[A] {
2   def awaitShutdown: UIO[Unit]
3   def isShutdown: UIO[Boolean]
4   def shutdown: UIO[Unit]
5 }
```

`Queue#shutdown` shuts down a `Queue` and immediately interrupts all fibers that are suspended on offering or taking values. `Queue#isShutdown` returns whether the `Queue` has currently been shut down, and `Queue#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. In the next section on resource handling, we will see some easy ways to do that.

## 11.5 Conclusion

In this chapter, we took an in-depth look at `Queue`. In the process, we went from the basics of offering and taking values to different varieties of queues to learning how to 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 `Queue` brings to the table, chapter 14, which focuses on `Semaphore`, will be a short one, so you are almost done with your journey through ZIO's core concurrent data structures!

## 11.6 Exercises

1. Implement load balancer that distributes work across multiple worker queues using a round-robin strategy:

```

1 trait LoadBalancer[A] {
2   def submit(work: A): Task[Unit]
```

```
3 |     def shutdown: Task[Unit]
4 | }
5 | object LoadBalancer {
6 |     def make[A](workerCount: Int, process: A => Task[A]) = ???
7 | }
```

2. Implement a rate limiter that limits the number of requests processed in a given time frame. It takes the time interval and the maximum number of calls that are allowed to be performed within the time interval:

```
1 trait RateLimiter {
2     def acquire: UIO[Unit]
3     def apply[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A]
4 }
5
6 object RateLimiter {
7     def make(max: Int, interval: Duration): UIO[RateLimiter] =
8         ???
9 }
```

3. Implement a circuit breaker that prevents calls to a service after a certain number of failures:

```
1 trait CircuitBreaker {
2     def protect[A](operation: => Task[A]): Task[A]
3 }
```

Hint: Use a sliding queue to store the results of the most recent operations and track the number of failures.

## Chapter 12

# Concurrent Structures: Hub - Broadcasting

One way to think about a concurrent data structure is as the optimal solution to a particular class of problems.

A queue represents the optimal solution to the problem of how to *distribute* work. This is an extremely important class of problems, and many other problems can be framed as problems of work distribution, which is why ZIO provides an extremely high-performance `Queue` data type.

However, a queue does not provide the optimal solution for another related class of problems: how to *broadcast* work.

To see the difference between these two classes of problems, consider what happens when a producer produces values A, B, C, and so on, and multiple consumers consume those values.

If we distribute these values, we want the first consumer to take A, the second consumer to take B, and the third consumer to take C. The same consumer may take multiple values over time, but we want to guarantee that each value will be consumed by *exactly one* consumer.

This would, for example, correspond to each value representing a unit of work and each consumer representing an identical worker. We want each unit of work to be done by some workers, but we never want the same unit of work to be done more than once.

In contrast, if we broadcast these values, we want each consumer to take A, B, and C. Specifically, we want the guarantee that each value will be consumed by *every* consumer.

This would correspond to each value representing a unit of work and each consumer representing a worker that does something different with that unit of work. Here, we want each unit of work to be done by all workers.

We can see that this reflects a fundamental duality. Distributing says “offer to this con-

sumer or that consumer” while broadcasting says “offer to this consumer and that consumer”.

Broadcasting work is a less common class of problem than distributing work. It can also be implemented, albeit inefficiently, as a subclass of the problem of distributing work.

Specifically, if we want to distribute work to  $N$  consumers, we can always model this by creating  $N$  queues and offering each value to every queue. Then, each consumer can have its own queue and take values from it, honoring the guarantee that each consumer will receive every value.

This approach is often taken to solve this class of problems, but if we think about it for a moment, there is a fundamental inefficiency here. We are creating  $N$  queues and offering each value to a queue  $N$  times when there is really only one of each value, and we should only have to offer it a single time.

ZIO’s Hub data type was created to address exactly this inefficiency.

## 12.1 Hub: An Optimal Solution to the Broadcasting Problem

The Hub reimagines the optimal solution to the broadcasting problem as being based on a single data structure, with each consumer maintaining their own reference to the position in the data structure they are currently consuming from. In this way, we only need to maintain one data structure and update it once when a producer produces a new value, as opposed to  $N$  data structures updated  $N$  times as in the naive solution.

The interface looks like this:

```

1 sealed abstract class Hub[A] {
2   def publish(a: A): UIO[Boolean] =
3     ???
4   def subscribe: ZIO[Scope, Nothing, Dequeue[A]] =
5     ???
6 }
7
8 object Hub {
9   def bounded[A](capacity: Int): UIO[Hub[A]] =
10    ???
11 }
```

A hub is defined in terms of two fundamental operators, `Hub#publish` and `Hub#subscribe`.

The `Hub#publish` operator publishes a value to the hub and is very similar to the `offer` operator on `Queue`.

The `Hub#subscribe` operator subscribes to receive values from the hub, returning a `Dequeue` that can be used to take values from the hub.

We will see more about the Scope data type later in this book, but for now, think of a scoped ZIO as being some resource that requires finalization. We can call `flatMap` on the resource to do something with it, with the guarantee that the resource will be released when the scope is closed, no matter whether the use of the resource succeeds, fails, or is interrupted.

Here, the “resource” described by the scoped ZIO is the subscription to the hub. The ZIO will subscribe to receive values from the hub and unsubscribe as part of its release action.

This concept of who is subscribed to the hub is important because consumers will only receive published values to the hub while they are subscribed. In addition, values will only be removed from the hub when all subscribers have taken them, so we need to know who is currently subscribed to the hub to know when it is safe to remove them.

The scoped ZIO takes care of this for us by handling subscribing before we start taking values and automatically unsubscribing when we are done.

```

1 trait Dequeue[+A] {
2   def take: ZIO[Any, Nothing, A]
3 }
4
5 trait Enqueue[-A] {
6   def offer(a: A): ZIO[Any, Nothing, Boolean]
7 }
```

For `Dequeue`, the `take` operator allows us to take values of type `A`, doesn’t require any environment, and can’t fail. So we can `take` values just like a normal `Queue`. `Dequeue` is a queue that can only be taken from (dequeued), so we can never actually offer a value to a `dequeue`.

Similar reasoning applies to `Enqueue` with the reversed treatment of the `offer` and `take` operators.

The `offer` operator on an `Enqueue` allows us to offer values of type `A`, doesn’t require any environment, and can’t fail. So we can `offer` values to an `Enqueue` just like a regular queue. The `Enqueue` is a queue that can only be offered to (enqueued), so we can never actually take a value from an `enqueue`.

A `Dequeue` is a queue that can only be taken from, and an `Enqueue` is a queue that can only be offered to.

At this point, we may wonder how either of these data types could ever be useful. What is the point of offering values to a queue if no one can ever take them or taking values when no one can ever offer them?

The answer is that even though we can’t offer or take values, someone else is producing or consuming those values. For example, in the case of the Hub, the values we take from the `Enqueue` are the ones published to the hub.

This supports a more abstract view of queues.

We normally think of queues as concrete data structures backed by an array or a linked list.

However, we can also think of queues more abstractly as “channels” that can be read from and written to. The `Dequeue` and `Enqueue` data types then give us the ability to interact with one side of this channel, reading or writing values as appropriate and performing basic operations such as observing whether the channel has been shut down.

In this conception, when we work with a queue, we aren’t necessarily saying anything about how this communication channel is implemented.

When we see a `Dequeue`, it could be backed by a concrete queue, a hub, or something else entirely. All we know is that it is a source of values that come from somewhere and supports this protocol.

Coming back to the Hub, the `Dequeue` returned by `subscribe` allows us to take values from the hub while we are subscribed. Since a `Dequeue` is a queue, we can use all the operators we are familiar with from queues to work with the subscription.

This is extremely powerful because it means we can use a subscription anywhere else; we need a queue that can only be written to. We will see later how this is used to provide extremely good integration with ZIO Stream.

## 12.2 Creating Hubs

Now that we have a basic understanding of a Hub, let’s discuss different ways to construct hubs.

Like queues, hubs can be *bounded* or *unbounded*. They can also use various *strategies* to describe what to do if a message is offered to the hub when it is at capacity.

As we go through this section and the materials below, you will see that the constructors and operators for hubs are very similar to those for queues. So, if you know how to use a queue, you should already be in a great position to use a hub.

### 12.2.1 Bounded Hubs

The most common variant of a hub is a bounded hub with a backpressure strategy. This is created by using the `Hub.bounded` operator and specifying a requested capacity:

```

1 object Hub {
2   def bounded[A](requestedCapacity: Int): UIO[Hub[A]] =
3     ???
4 }
```

A bounded hub is created with a specified capacity. For maximum efficiency, the capacity should be a power of two, though other capacities are supported.

Conceptually, a bounded hub is represented by an array containing the values in the hub as well as the number of subscribers who still have to take that value. Each subscriber maintains its own index in the array, and each time it takes a value, it decrements the number

of subscribers who still have to take a value from that index and removes the value if it is the last subscriber.

Because a bounded hub has a capacity, we need to answer the question of what we should do if a publisher publishes a message to a hub that is already at capacity. For a hub constructed with the bounded constructor, the answer is that the publisher will back pressure until there is space available in the hub.

A bounded hub with the back pressure strategy has several characteristics that make it a good default:

First, it guarantees that every subscriber will receive every value published to the hub while it is subscribed (i.e., values will never be dropped). This is an essential property for correctness in many applications, such as streaming.

Second, it prevents memory leaks. If the size of the hub is not bounded and messages are being published to the hub at a faster rate than they are being taken, the size of the hub can keep growing, consuming more memory until the program eventually crashes.

Third, it naturally propagates back pressure through a system. There is no point in publishers publishing additional values if the hub is already at capacity, and the back pressure strategy automatically handles this by causing publishers to suspend until capacity is available.

### 12.2.2 Sliding Hubs

The next variant of a hub is the sliding hub.

A sliding hub is constructed with the `Hub.sliding` constructor.

```

1 object Hub {
2     def sliding[A](requestedCapacity: Int): UIO[Hub[A]] =
3         ???
4 }
```

A hub constructed with the `Hub.sliding` constructor has a finite capacity, just like a hub constructed with the `Hub.bounded` constructor. The difference is the strategy that the sliding hub uses when a message is published to the hub and it is already at capacity.

Whereas the back pressure strategy causes publishers to suspend until the hub has capacity, the sliding strategy creates capacity by dropping the oldest message in the hub. This guarantees that publishing a value to the hub will always be completed immediately.

A sliding hub is particularly good for dealing with the slow consumer problem.

In the slow consumer problem, one subscriber takes messages from the hub much slower than the other consumers. This would cause the hub to quickly reach capacity because messages are only removed from the hub when all subscribers have taken them.

With the backpressure strategy, publishers would suspend until the slow subscriber took an additional value, freeing up additional space.

In many applications, this may be necessary for correctness. If every subscriber needs to receive every message, and we only have finite space to buffer messages, then there is really nothing we can do other than back pressure.

But in other cases, it can be less than ideal. By back-pressuring, we allow the slow consumer to determine the rate at which messages are received by other consumers since publishers can't publish new messages until the slow consumer takes the old one and makes space available.

This introduces interdependence between different consumers, whereas ideally, we would like them to be independent. If we really need every subscriber to receive every message and have limited space, then unfortunately, we can't do better than this.

However, if we are willing to relax these constraints, we can solve this problem by dropping messages to the slow subscriber. This is exactly what the sliding strategy does.

With the sliding strategy, when a publisher publishes a new message, the oldest message, which has not yet been taken by the slow subscriber, will simply be dropped. This will free up space in the hub, allowing the publisher to publish the new value and the other subscribers to immediately consume it.

In this way, the sliding strategy prevents any consumer from slowing down the rate at which messages are published to the hub. Thus, the rate at which a subscriber takes messages from the hub is driven entirely by itself and the publishing rate, achieving the independence between multiple subscribers we discussed above.

### 12.2.3 Unbounded Hubs

Unlike a hub constructed with the `Hub.bounded` or `Hub.sliding` constructors, a hub constructed with the `Hub.unbounded` constructor does not have a maximum capacity:

```

1 | object Hub {
2 |   def unbounded[A]: UIO[Hub[A]] =
3 |     ???
4 |

```

A bounded hub does not have to apply any strategy when a message is published to it and it is at capacity because the hub is never at capacity. If another message is published, we simply add it to the hub.

You can think of an unbounded hub as being backed by a linked list, where each subscriber maintains its own reference to its “head”. As subscribers take values, they decrease the number of subscribers that still need to take that node, and if they are the last subscriber, the node is removed from the linked list.

The main advantage of the unbounded hub is that it combines the properties that no value will ever be dropped, and publishing will always be complete immediately. However, it does this at the cost of potentially using unlimited memory.

An unbounded hub can make sense when we are confident that the hub's size will be relatively small or expect the caller to be responsible for consuming values.

However, unbounded hubs can result in memory leaks if messages are not being taken as quickly as they are being published. We recommend using bounded hubs with reasonable default or parameterizing operators that use hubs on the capacity if possible.

#### 12.2.4 Dropping Hubs

The final type of hub is a dropping hub:

```

1 object Hub {
2   def dropping[A](requestedCapacity: Int): UIO[Hub[A]] =
3     ???
4 }
```

A hub constructed with the `Hub.dropping` constructor has a finite size like hubs constructed with the `Hub.bounded` and `Hub.sliding` constructors but uses the dropping strategy if a publisher publishes a message and the hub is full.

The dropping strategy handles messages published to a full hub by simply dropping them. The `Hub#publish` operator will return `false` when this happens to signal to the publisher that the value was not successfully published and allow the publisher to take additional actions.

Dropping hubs tend to be less frequently used than sliding hubs. Since the new message is dropped, the dropping strategy still allows a slow subscriber to slow down the rate at which messages are received by other subscribers.

However, dropping hubs can still be useful as the basis for higher-level strategies to manage the possibility that the hub is full.

We can think of the other strategies as embodying very specific answers to the question of what to do when the hub is full. We either suspend the publisher or drop the oldest value.

These strategies have the advantage that they do a very specific thing. We often want to do one of those things, which is why these constructors exist.

But we can imagine wanting to handle the possibility that the hub is full differently in a particular application, for example, by retrying publishing according to a specified schedule and logging the failure.

If we use the backpressure or sliding strategies, we can't really do that because the possibility that the hub is full is already handled for us. Either the publisher suspends, or the oldest message is dropped.

The dropping strategy can be thought of as a strategy that does nothing itself. If the hub is full, it simply tells us that. We still have the original value we tried to publish, so we are free to do anything we want with it, like try to publish it again.

Thus, the dropping strategy is, in some ways, the most general since it allows us to implement whatever logic we want based on the hub being full. Of course, our logic doesn't have access to the internal implementation of the hub to implement that logic, whereas the bounded and sliding strategies can use that information.

## 12.3 Operators on Hubs

Now that we understand how to create hubs, let's look at some more operators on hubs.

The interface of Hub looks like this:

```

1 sealed trait Hub[A] {
2   def awaitShutdown: UIO[Unit]
3   def capacity: Int
4   def isShutdown: UIO[Boolean]
5   def publish(a: A): UIO[A]
6   def publishAll(as: Iterable[A]): UIO[Boolean]
7   def shutdown: UIO[Unit]
8   def size: UIO[Int]
9   def subscribe: ZIO[Scope, Nothing, Dequeue[A]]
10  def toQueue: Enqueue[A]
11 }

```

This should look quite familiar if you remember the interface of Queue. Other than the publish and subscribe operators, which take the place of offer and take, all operators are the same.

Just like a Queue, a Hub has a concept of being shut down. Once a hub is shut down, any fiber attempting to publish a value to the hub or take a value from a subscription to the hub will be immediately interrupted.

We can shut down a hub using the Hub#shutdown operator, check if it has been shut down using the Hub#isShutdown operator, or wait for shutdown to complete using the Hub#awaitshutdown operator.

We can also get the current number of messages in the hub with the Hub#size operator or the hub's capacity with the Hub#capacity operator. Note that Hub#capacity will be equal to Int.MaxValue in the case of an unbounded hub, but an unbounded hub will never be full.

Another important operator is Hub#toQueue. The Hub#toQueue operator allows us to view any Hub as a Queue that can only be written to.

Going back to the idea we discussed above about conceptualizing a Queue more abstractly as a “channel” that can be written to and read from, we can think of the Hub itself as a channel that can only be written to. Writing values to this channel corresponds to publishing them to the hub, which makes those values available to subscribers to the hub.

The hub is a channel that can only be written to because it does not make sense to take values from it. Rather, we subscribe to receive values from the hub to open a new channel and then take values from that channel.

In this way, we can view the hub itself as more of a “nexus” of communication. The Enqueue returned by Hub#toQueue represents an input channel, each subscription returned by Hub#subscribe represents an output channel and the Hub itself sits at the center of these channels and allows us to create new output channels.

The advantage of viewing a hub this way is that with the `Hub#toQueue` operator, we can use a Hub anywhere we need a Queue that can only be written to.

For example, we can imagine an effect that takes incoming values and offers them to a queue. We might then have a consumer that takes values from that queue and does something with them, say performing some business logic.

Now, we would like to add additional consumers that do something else with each value, perhaps persisting it to a database or logging it. We can do this easily with Hub because we can create a hub and view it as a queue using the `Hub#toQueue` operator.

Now, we can plug our hub into the operator that expects a queue to offer values to. All we need to do is have each of our consumers subscribe to receive values from the hub.

Since the subscriptions are themselves queues, the interface for both the producer and the existing consumer is almost entirely unchanged. But we have made quite a significant refactoring of the architecture of our application, changing from a linear data flow to a “fan out” data flow.

## 12.4 Conclusion

In this chapter, we learned about ZIO’s Hub data type, which supports extremely fast broadcast operations. We will see more about this data type when we talk about ZIO Stream and see how Hub is used to support streaming broadcast operations.

In the meantime, there are two key things to take away from this chapter.

First, you should be able to recognize problems involving broadcasting work as opposed to distributing work and use Hub to solve these problems efficiently. Problems involving broadcasting work are those involving distributing work, but when they exist, using Hub can be an order of magnitude more efficient than using queue-based solutions, so it is worth knowing how to spot these types of problems and use the best tool for the job.

Second, this chapter has hopefully gotten you thinking about separating interfaces from underlying representations of data.

Subscribing to a Hub returns a Dequeue from which we can take values, check the size, or shut down the queue just like any other. But as we saw, no actual queue data structure is created when we subscribe to a hub; the `ZQueue` interface is a description of a “channel” from which we can take values.

Similarly, we saw how we can view a Hub as an Enqueue and offer values to it just like any other queue, even though, again, there is no actual queue data structure but instead a hub with a completely different underlying implementation.

We took this idea even further in our discussion of polymorphic hubs and saw how we could describe broadcasting in terms of a set of publishers offering values of type A to an Enqueue and each subscriber taking values of type B from a Dequeue. At this point, no one on either side of the channel needs to know anything about the other side of the channel, including even the nature of the channel itself.

This provides a high level of decoupling of components, allowing us to change the producer or consumer side of a system without impacting the other side and even to substitute different implementations of the channel itself. At the same time, the actual implementation in terms of the Hub is blazing fast, so we don't sacrifice any performance for this abstraction.

It can be worth considering how you can apply similar ideas to your own code, providing a high-level, composable interface backed by a low-level, performant implementation.

But for now, we must move on to learn about other aspects of ZIO!

## 12.5 Exercises

1. Create a chatroom system using Hub where multiple users can join the chat so each message sent is broadcast to all users. Each message sent by a user is received by all other users. Users can leave the chat. There is also a process that logs all messages sent to the chat room in a file:

```

1 trait UserSession {
2   def sendMessage(message: String): UIO[Unit]
3   def receiveMessages: ZIO[Scope, Nothing, Dequeue[ChatEvent
4     ]]
5   def leave: UIO[Unit]
6 }
7
8 trait ChatRoom {
9   def join(username: String): ZIO[Scope, Nothing, Dequeue[
10     UserSession]]
11   def shutdown(username: String): UIO[Unit]
12 }
```

2. Write a real-time auction system that allows multiple bidders to practice in auctions simultaneously. The auction system should broadcast bid updates to all participants while maintaining the strict ordering of bids. Each participant should be able to place bids and receive updates on the current highest bid:

```

1 trait AuctionSystem {
2   def placeBid(
3     auctionId: String,
4     bidderId: String,
5     amount: BigDecimal
6   ): UIO[Boolean]
7
8   def createAuction(
9     id: String,
10    startPrice: BigDecimal,
11    duration: Duration
12  ): UIO[Unit]
```

```
13
14     def subscribe: ZIO[Scope, Nothing, Dequeue[AuctionEvent]]
15
16     def getAuction(id: String): UIO[Option[AuctionState]]
17 }
```

The core models for the auction system could be as follows:

```
1 case class Bid(
2     auctionId: String,
3     bidderId: String,
4     amount: BigDecimal,
5     timestamp: Long
6 )
7
8 case class AuctionState(
9     id: String,
10    currentPrice: BigDecimal,
11    currentWinner: Option[String],
12    endTime: Long,
13    isActive: Boolean
14 )
15
16 // Events we'll broadcast
17 sealed trait AuctionEvent
18 case class BidPlaced(bid: Bid) extends AuctionEvent
19 case class AuctionEnded(
20     auctionId: String,
21     finalPrice: BigDecimal,
22     winner: Option[String]
23 ) extends AuctionEvent
```

# 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, we can do too many things simultaneously with ZIO's fiber-based concurrency model! One of our internal services is receiving too many simultaneous requests now that we do as many things as possible in parallel and never block. How do we limit work when needed so we don't overwhelm other systems?

### 13.1 Semaphore Interface

This is where `Semaphore` comes in. Despite its somewhat fancy-sounding name, `Semaphore` is a very straightforward data structure. A `Semaphore` is created with a certain number of “permits” and is defined in terms of a single method, `Semaphore#withPermits`:

```
1 import zio._

2
3 trait Semaphore {
4   def withPermits[R, E, A](n: Long)(
5     task: ZIO[R, E, A]
6   ): ZIO[R, E, A]
7 }
```

The guarantee of `Semaphore#withPermits` is simple. Any fiber calling `Semaphore#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 the `task`. If that number of permits is unavailable, the fiber is suspended until the specified number of permits is available. Acquired permits will be “released”, and the number of available permits is 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:

```

1 trait Semaphore {
2   def withPermits[R, E, A](
3     n: Long
4   )(task: ZIO[R, E, A]): ZIO[R, E, A]
5   def withPermit[R, E, A](task: ZIO[R, E, A]): ZIO[R, E, A] =
6     withPermits(1)(task)
7 }
```

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 parking lot operator guarantees 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 a straightforward way 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:

```

1 import zio._
2 import zio.Clock._
3 import zio.Console._
4
5 def queryDatabase(
6   connections: Ref[Int]
7 ): URIO[Any, Unit] =
8   connections.updateAndGet(_ + 1).flatMap { n =>
9     Console.printLine(s"Aquiring, now $n connections").orDie *>
10       ZIO.sleep(1.second) *>
11       Console.printLine(s"Closing, now ${n - 1} connections").
12       orDie
13   }
14
15 for {
16   ref      <- Ref.make(0)
17   semaphore <- Semaphore.make(4)
18   query    = semaphore.withPermit(queryDatabase(ref))
```

```

18 |     _           <- ZIO.foreachParDiscard(1 to 10)(_ => query)
19 | } yield ()

```

Without the `Semaphore`, `ZIO.foreachParDiscard` would cause all ten queries to execute simultaneously. So, we would see all ten connections acquired and then 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 than 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 `Semaphore#withPermits`.

### 13.3 Using Semaphore to Implement Operators

This pattern of using a `Semaphore` to limit the degree of parallelism with `ZIO.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`:

```

1 object ZIO {
2     def foreachPar[R, E, A, B](
3         as: Iterable[A]
4     )(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
5         ???
6     def foreachParN[R, E, A, B](
7         n: Int
8     )(as: Iterable[A])(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
9         for {
10             semaphore <- Semaphore.make(n)
11             bs <- ZIO.foreachPar(as) { a =>
12                 semaphore.withPermit(f(a))
13             }

```

```

14     } yield bs
15 }
```

In many cases, you can just use `ZIO.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 `ZIO.foreachPar`.

## 13.4 Making 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 execute 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 unsafe for concurrent access. For example, we talked about `Ref.Synchronized` in the chapter on references but were not ready at the time to discuss how `Ref.Synchronized` was implemented. Now, we are in a position to do so:

Recall that `Ref.Synchronized` is a reference that allows performing effects in the `modify` function:

```

1 object Ref {
2   trait Synchronized[A] {
3     def modify[R, E, B](f: A => ZIO[R, E, (B, A)]): ZIO[R, E, B]
4   }
5 }
```

We could try to naively implement `Ref.Synchronized` in terms of `Ref` like this:

```

1 object Ref {
2   object Synchronized {
3     def make[A](a: A): UIO[Ref.Synchronized[A]] =
4       Ref.make(a).map { ref =>
5         new Ref.Synchronized[A] {
6           def modify[R, E, B](f: A => ZIO[R, E, (B, A)]): ZIO[R,
7             E, B] =
8             for {
9               a <- ref.get
10              v <- f(a)
11              _ <- ref.set(v._2)
12            } yield v._1
13         }
14       }
15 }
```

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 `Ref.Synchronized`, 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, which is 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 `Ref.Synchronized` concurrently, the first will get the value, perform the effect, and set the value. The second fiber will be suspended because there are no more available permits. Once the first fiber finishes setting the value, it will return its permit, allowing the second fiber to get the updated value, perform its effect, and set 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`:

```

1 object Ref {
2     object Synchronized {
3         def make[A](a: A): UIO[Ref.Synchronized[A]] =
4             for {
5                 ref      <- Ref.make(a)
6                 semaphore <- Semaphore.make(1)
7             } yield new Ref.Synchronized[A] {
8                 def modify[R, E, B](
9                     f: A => ZIO[R, E, (B, A)]
10                ): ZIO[R, E, B] =
11                     semaphore.withPermit {
12                         for {
13                             a <- ref.get
14                             v <- f(a)
15                             _ <- ref.set(v._2)
16                         } yield v._1
17                     }
18                 }
19             }
20 }
```

Congratulations! You have implemented your own concurrent data structure.

`Semaphore` is an excellent 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

Semaphores provide an elegant solution to one of the most common challenges in concurrent programming: controlling the degree of parallelism.

The beauty of ZIO's Semaphore implementation lies in its simplicity and safety. With just a single core operation (`withPermits`), it enables powerful concurrency control patterns while avoiding common pitfalls like permit leaks or thread blocking. Unlike traditional concurrent programming approaches that often require careful manual resource management, ZIO's Semaphore automatically handles permit release through its bracketing behavior, making it nearly impossible to write incorrect code.

We've seen how Semaphores can be applied in two main scenarios: 1. As a work limiter, ensuring that no more than N operations occur simultaneously (like limiting database connections) 2. As a concurrency guard, using a single permit to make non-thread-safe operations safe for concurrent access (as demonstrated in our `Ref . Synchronized` implementation)

These patterns are so fundamental that they appear throughout ZIO's ecosystem, from high-level operators like `foreachParN` to concurrent data structure implementations.

In the next chapter, we will discuss resource management using the `ZIO . acquireReleaseWith` operator, which allows us to safely acquire and release resources in a concurrent environment. After that, you will be well-equipped to understand the underlying implementation details of `Semaphore#withPermits` and other ZIO operators that manage resources safely and efficiently.

## Chapter 14

# Resource Handling: Acquire Release - 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 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 tiny 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 every time. This can be especially true when 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. `ZIO.acquireRelease` and `Scope`, discussed in the next chapter, are ZIO's solutions to do this.

### 14.1 The Limitation of `try-finally` in Asynchronous Programming

The traditional solution to the problem of safe resource handling is the `try ... finally` construct:

```
| trait Resource
```

```

2
3 lazy val acquire: Resource          = ???
4 def use(resource: Resource): Unit   = ???
5 def release(resource: Resource): Unit = ???
6
7 lazy val example = {
8     val resource = acquire
9     try {
10         use(resource)
11     } finally {
12         release(resource)
13     }
14 }
```

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 throws an exception.

This is sufficient for fully synchronous code. 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 translating 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`:

```

1 import scala.concurrent.{ExecutionContext, Future}
2 import scala.util.{Failure, Success}
3
4 trait Resource
5
6 def acquire: Future[Resource]          = ???
7 def use(resource: Resource): Future[Unit] = ???
8 def release(resource: Resource): Future[Unit] = ???
9
10 implicit final class FutureSyntax[+A](future: Future[A]) {
11     def ensuring(
12         finalizer: Future[Any]
13     )(implicit ec: ExecutionContext): Future[A] =
14         future.transformWith {
15             case Success(a) =>
16                 finalizer.flatMap(_ => Future.successful(a))
17             case Failure(e) =>
18                 finalizer.flatMap(_ => Future.failed(e))
19         }
20     }
21
22 implicit val global = scala.concurrent.ExecutionContext.global
23
24 lazy val example =
```

```

25   acquire.flatMap { resource =>
26     use(resource).ensuring(release(resource))
27 }
```

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 on this.

This seems okay at first glance. The code is relatively terse, ensuring that `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” workflows 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 is no longer needed, 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 Acquire Release as a Generalization of Try and Finally

The solution is ZIO’s `ZIO.acquireReleaseWith` operator. The problem with the `try ... finally` and the `ensuring` solution we developed above is that they only operate on the 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. `acquireReleaseWith` does just that.

```

1 import zio._
2
3 object ZIO {
4   def acquireReleaseWith[R, E, A, B](
5     acquire: ZIO[R, E, A]
6   )(
7     release: A => ZIO[R, Nothing, Any]
8     )(use: A => ZIO[R, E, B]): ZIO[R, E, B] =
```

```

9 |     ???
10| }
```

Let's look at this type signature in more detail.

`acquire` is a workflow that describes the acquisition of some resource A, for example, a file handle. Acquiring the resource could fail, for instance, 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 `acquireReleaseWith` 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 `acquireReleaseWith` operator 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, we 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 ... finally`. If the resource is successfully acquired, then the `release` action will always be run. So, it will still be released even if we are interrupted before we begin using the resource.

As long as we use `acquireReleaseWith`, it is impossible to acquire and finish using a resource without it being released.

The `use` action comes last here because using the resource typically involves the most logic, so this provides more convenient syntax:

```

1 ZIO.acquireReleaseWith(acquire)(release) { a =>
2   ??? // do something with the resource here
3 }
```

In addition to `ZIO.acquireReleaseWith`, there is one more powerful variant to be aware of, `ZIO.acquireReleaseExitWith`. `acquireReleaseWith` provides access to

the resource in the `release` function, but it doesn't provide any indication of how the use workflow completed. Most of the time, this doesn't matter. Whether use is completed successfully, failed, or interrupted, we must 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:

```

1 object ZIO {
2     def acquireReleaseExitWith[R, E, A, B](
3         acquire: ZIO[R, E, A]
4     )(
5         release: (A, Exit[E, B]) => ZIO[R, Nothing, Any]
6     )(use: A => ZIO[R, E, B]): ZIO[R, E, B] =
7     ???
8 }
```

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 various types of failures.

When using `ZIO.acquireReleaseWith`, a best practice is to have `use` return the first value possible that does not require the resource to remain 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 `ZIO.acquireRelease`, 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 open 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 `ZIO.acquireRelease` 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 Operator

`ZIO` does also have a `ZIO#ensuring` operator like the one we sketched out at the beginning of the chapter for `Future`, but this one does handle the interruption. The signature of `ensuring` is as follows:

```

1 trait ZIO[-R, +E, +A] {
2     def ensuring[R1 <: R](
3         finalizer: ZIO[R1, Nothing, Any]
4     ): ZIO[R1, E, A]
5 }
```

`ensuring` is like `ZIO.acquireReleaseWith` without the resource. It guarantees that if the original workflow begins execution, 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 workflow. For example, if we have a workflow incrementing a timer, we may want to reset the timer to zero when the workflow terminates, no matter what.

`ensuring` is very useful in various situations where you want to add a finalizer to a workflow, but one watch out is not to use `ensuring` when `acquireReleaseWith` is required. For example, the following code has a bug:

```
1 | acquire.flatMap(resource => use(resource).ensuring(release))
```

Even with the ZIO version of `ensuring`, this is unsafe because execution could be interrupted between `acquire` completing execution and `use` beginning execution. The guarantee of `ensuring` is only that if the original workflow begins execution, the finalizer will be run. If the original workflow 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 `acquireReleaseWith`, otherwise use `ensuring`.

## 14.4 Conclusion

In this chapter, we learned about `acquireReleaseWith`, 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, `acquireReleaseWith` is all we need and is a huge step forward in providing ironclad guarantees about safe resource usage. But what if we have to use multiple resources, such as reading from one file and writing to another? `Scope` 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

1. Rewrite the following `sendData` function in terms of the `ZIO.requireReleaseWith` operator:

```
1 | import zio._
2 | import scala.util.Try
3 | import java.net.Socket
4 |
5 | object LegacySendData {
6 |   def sendData(
7 |     host: String,
8 |     port: Int,
9 |     data: Array[Byte]
10 |   ): Try[Int] = {
```

```

11  var socket: Socket = null
12  try {
13    socket = new Socket(host, port)
14    val out = socket.getOutputStream
15    out.write(data)
16    Success(data.length)
17  } catch {
18    case e: Exception => Failure(e)
19  } finally {
20    if (socket != null) socket.close()
21  }
22 }
23 }
24
25 // rewrite the function using ZIO
26 def sendData(
27   host: String,
28   port: Int,
29   data: Array[Byte]
30 ): Task[Int] = ???
```

2. Implement `ZIO.acquireReleaseWith` using `ZIO.uninterruptibleMask`. Write a test to ensure that `ZIO.acquireReleaseWith` guarantees the three rules discussed in this chapter.
3. Implement a simple semaphore using `Ref` and `Promise` and using `ZIO.acquireReleaseWith` operator. A semaphore is a synchronization primitive controlling access to a common resource by multiple fibers. It is essentially a counter that tracks how many fibers can access a resource at a time:

```

1 trait Semaphore {
2   def withPermits[R, E, A](n: Long)(task: ZIO[R, E, A]): ZIO
3     [R, E, A]
4 }
5 object Semaphore {
6   def make(permits: => Long): UIO[Semaphore] = ???
7 }
```

## Chapter 15

# Resource Handling: Scope - Composable Resources

The ZIO.acquireReleaseWith operator is the foundation of safe resource handling in ZIO. Ultimately, ZIO.acquireReleaseWith is all we need, but composing multiple resources directly with ZIO.acquireReleaseWith is not always the most ergonomic.

For example, say we have a file containing weather data. We would like to use ZIO.acquireReleaseWith to safely open the file containing weather data and a second file containing 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:

```
1 import java.io.{File, IOException}
2
3 import zio._
4
5 def openFile(name: String): IO[IOException, File] =
6   ???
7
8 def closeFile(file: File): UIO[Unit] =
9   ???
10
11 def withFile[A](name: String)(use: File => Task[A]): Task[A] =
12   ZIO.acquireReleaseWith(openFile(name))(closeFile)(use)
13
14 def analyze(weatherData: File, results: File): Task[Unit] =
15   ???
```

```

17 lazy val analyzeWeatherData: Task[Unit] =
18   withFile("temperatures.txt") { weatherData =>
19     withFile("results.txt") { results =>
20       analyze(weatherData, results)
21     }
22   }

```

This will allow us to read from “temperatures.txt” and write to “results.txt” simultaneously while ensuring that both files are closed regardless of what happens.

But there are a couple of issues with it.

First, it is not very composable.

In this case, where we were only working with two resources, it was pretty 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 particular order for handling these files. First, we open the “temperatures.txt” file, then we open the “results.txt” file, then we close the “results.txt” file, and finally, we close the “temperatures.temperatures” file.

But that specific ordering is not really necessary. For maximum efficiency, we could open the weather data and results files simultaneously, transform the data, and then close them simultaneously.

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 it could make a huge difference in other cases.

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 `ZIO.acquireReleaseWith` to deal with single resources in the case where we have multiple resources?

## 15.1 Reification of Acquire Release

The first clue to recapturing the simplicity of `ZIO.acquireReleaseWith` when working with multiple resources is the `withFile` operator we implemented above:

```

1 def withFile[A](name: String)(use: File => Task[A]): Task[A] =
2   ZIO.acquireReleaseWith(openFile(name))(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 release it safely 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, 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. In this case, it succeeds with some type `A`, fails with a `Throwable`, and potentially performs arbitrary work 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. In the example above, we would conceptually like to combine the resource described by `withFile("temperatures.txt")` and the resource described by `withFile("results.txt")` 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:

```
1 def acquireRelease[R, E, A](
2   acquire: ZIO[R, E, A]
3 )(release: A => ZIO[R, Nothing, Any]): ZIO[R with Scope, E, A] =
4   ???
```

Here, a `Scope` represents something that finalizers can be added to and which can eventually be closed to run all the finalizers that have been added to it:

```
1 trait Scope {
2   def addFinalizer[R](
3     finalizer: ZIO[Any, Nothing, Any]
4   ): ZIO[Any, Nothing, Unit]
5   def close(exit: Exit[Any, Any]): ZIO[Any, Nothing, Unit]
6 }
```

When we acquire a resource constructed with `acquireRelease`, we add a finalizer to the `Scope`. The resulting workflow requires a `Scope` for the finalizer to be added to, which tells us that this workflow allocates some resources that require finalization.

We can work with a resource as much as all of the operators on `ZIO` we are familiar with, such as `flatMap`, to use the resource.

When we are done working with the resource, we can use the `ZIO.scoped` operator to create a new `Scope`, provide it to the workflow, and automatically close the `Scope` when the workflow completes execution, whether by success, failure, or interruption:

```

1 def scoped[R, E, A](zio: ZIO[R with Scope, E, A]): ZIO[R, E, A] =
2   ???

```

Let's look at how we can use `acquireRelease` to describe a file handle resource:

```

1 def file(name: String): ZIO[Scope, Throwable, File] =
2   ZIO.acquireRelease(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 can now use all the operators we are already familiar with from ZIO to compose resources.

For example, say we want to create a new resource that describes sequentially acquiring the "temperatures.txt" file and the "results.txt" file.

We know the operator on ZIO for running two workflows sequentially and keeping both of their results is `zip`. So we can just use the `ZIO#zip` operator here:

```

1 import java.io.File
2
3 import zio._
4
5 def file(name: String): ZIO[Scope, Throwable, File] =
6   ???
7
8 def analyze(weatherData: File, results: File): Task[Unit] =
9   ???
10
11 lazy val sequential: ZIO[Scope, Throwable, (File, File)] =
12   file("temperatures.txt").zip(file("results.txt"))
13
14 def analyzeWeatherData(
15   files: ZIO[Any, Nothing, (File, File)])
16 ): Task[Unit] =
17   ZIO.scoped {
18     files.flatMap { case (weatherData, results) =>
19       analyze(weatherData, results)
20     }
21   }

```

Let's say we want to acquire the two resources in parallel. We know the operator on ZIO for running two workflows in parallel is `zipPar`, so if we want to open the two files in parallel, we can just use that:

```

1 lazy val parallelAcquire: ZIO[Scope, Throwable, (File, File)] =
2   file("temperatures.txt").zipPar(file("results.txt"))

```

By default, when a Scope is closed, the finalizers associated with it are run in reverse to the order in which they were added. So, in the above example, the two files will still be released

sequentially, although the order in which they are released will be nondeterministic.

If we also want to close the files in parallel, we just need to use the `ZIO#parallelFinalizers` operator, which runs the finalizers associated with the Scope in parallel instead of sequentially:

```

1 | lazy val parallelRelease: ZIO[Scope, Throwable, (File, File)] =
2 |   file("temperatures.txt")
3 |   .zipPar(file("results.txt"))
4 |   .parallelFinalizers

```

By using Scope, we can work with resources as values and compose them using all the regular operators on ZIO. We can do all this with the guarantee that the resources will be released as soon as the Scope is closed.

## 15.2 Scope as a Dynamic Scope

Another helpful way to think about Scope is as a dynamic versus a static scope.

Let's go back to the `ZIO.acquireReleaseWith` operator that we learned about in the previous chapter:

```

1 | ZIO.acquireReleaseWith(acquire)(release) { resource =>
2 |   use(resource)
3 |

```

When we use `ZIO.acquireReleaseWith`, we already know exactly what we want to do with the resource.

Notice that the curly braces in the example above indicate precisely the lifetime of the resource. We know that anywhere between those open and closing curly braces, the resource will already have been acquired and will not have been closed yet, so we can safely work with it.

The curly braces represent the “scope” in which the resource is valid. In this case, the scope is very clearly indicated by the curly braces and corresponds to the lexical scope of that code block.

Most importantly, for our purposes, the scope is “fixed”. We know how long the resource will be valid just by looking at the code, and once we run `ZIO.acquireReleaseWith`, nothing can extend the life of that resource. It will permanently be closed as soon as that block ends.

In contrast, when we create a resource with `ZIO.acquireRelease`, we don't know what the lifetime of the resource will be yet:

```

1 | val resource = ZIO.acquireRelease(acquire)(release)

```

From the snippet above, we don't know what the life of the resource will be yet.

It might be very short if we close it immediately:

```
1 ZIO.scoped(resource)
```

It might be very long if we do many things with it before closing it:

```
1 ZIO.scoped {
2   resource
3     .flatMap(doSomething)
4     .flatMap(doSomethingElse)
5     .flatMap(doYetAnotherThing)
6 }
```

In the first snippet, the resource will be released as soon as it has been acquired. The resource will not be released in the second snippet until `doSomething`, `doSomethingElse`, and `doYetAnotherThing` have completed execution.

In this case, the “scope” of the resource is visually indicated by the curly braces associated with `ZIO.scoped`. `ZIO.scoped` determines the lifetime of the resource, and it is independent of the creation of the resource.

As a result, we say that Scope describes a “dynamic” scope that can be extended with further logic and then closed by some higher level of the program.

This can be helpful when considering whether you should use `ZIO.acquireReleaseWith` or `ZIO.acquireRelease`.

If you already know what you want to do with a resource, for example, if you just want to open a file and use its contents, you can generally just use `acquireReleaseWith` as discussed in the last chapter.

If you don’t know what you want to do with the resource yet and want to let the caller decide what they want to do with the resource, then use `ZIO.acquireRelease` and Scope.

## 15.3 Constructing Scoped Resources

Now that we understand Scope and its power let’s examine how we can construct scoped resources.

### 15.3.1 Fundamental Constructors

The most common way of constructing scoped resources is the `ZIO.acquireRelease` operator we saw above:

```
1 def acquireReleaseWith[R, E, A](
2   acquire: ZIO[R, E, A]
3 )(release: A => URIO[R, Any]): ZIO[R with Scope, E, A] =
4   ???
```

With `acquireRelease`, both the `acquire` and `release` actions will be *uninterruptible*.

We need the finalizer to be uninterruptible because it is supposed to be run after the resource is finished being used, regardless of how the resource's use terminates. So, if the resource's use 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 ensure 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 interrupted?

We couldn't run the `release` action to close the resource because we wouldn't have a resource to close. However, the `acquire` action might still have done some work requiring finalization, such as opening a network connection to a database connection on a remote server.

For this reason, the default is that the `acquire` workflow of `ZIO.acquireReleaseWith` is uninterruptible.

There is also a `ZIO.acquireReleaseExit` operator that is like `acquireRelease` but lets you specify different logic for how the resource should be finalized depending on the result of the acquisition.

```

1 def acquireReleaseExit[R, E, A] (
2   acquire: ZIO[R, E, A]
3 )(
4   release: Exit[Any, Any] => URIO[R, Any]
5 ): ZIO[R with Scope, E, A] =
6   ???
```

If you need the `acquire` workflow to be interruptible there is a variant that supports this:

```

1 def acquireReleaseInterruptible[R, E, A] (
2   acquire: ZIO[R, E, A]
3 )(release: ZIO[R, Nothing, Any]): ZIO[R with Scope, E, A] =
4   ???
```

Here the interruptibility of the `acquire` workflow will not be changed. One thing to note here is that the `release` workflow does not have access to the result of `acquire`.

This is because the `acquire` workflow might be interrupted after the beginning execution but before completion. In this case, we would not have the resources, but we would still potentially need to do some finalization.

So, the `release` workflow here is responsible for doing any necessary clean-up without having access to the resource. For example, the `release` workflow might access some other in-memory state to determine what finalization was necessary, if any.

This operator is considerably less common than `ZIO.acquireRelease`. Like `ZIO.acquireReleaseExit`, there is an `acquireReleaseInterruptibleExit` variant that allows the `release` action to depend on how the acquisition and use of the resource

completed execution:

```

1 def acquireReleaseInterruptibleExit[R, E, A] (
2   acquire: ZIO[R, E, A]
3 )(
4   release: Exit[Any, Any] => ZIO[R, Nothing, Any]
5 ): ZIO[R with Scope, E, A] =
6   ???

```

Like `ZIO.acquireReleaseInterruptible`, the `release` workflow here does not have access to the resource because the `acquire` workflow might have been interrupted before completing execution.

### 15.3.2 Convenience Constructors

One useful operator for constructing scoped resources is `ZIO#withFinalizer`. This operator can be called on any `ZIO` workflow and treats that workflow as the acquisition of a resource, allowing us to specify our finalizer.

```

1 trait ZIO[-R, +E, +A] { self =>
2   def withFinalizer[R1](
3     f: A => ZIO[R1, Nothing, Any]
4   ): ZIO[R1 with Scope, Nothing, A] =
5     ZIO.acquireRelease(self)(f)
6 }

```

As we can see, `zio.withFinalizer(finalizer)` is implemented in terms of `acquireRelease`, so this does not add anything fundamental to our capabilities but can be more readable in some cases.

For example, when we create a `Queue`, we often want to shut it down when the current `Scope` is closed.

We can write that like this:

```
1 ZIO.acquireRelease(Queue.bounded(4096))(_.shutdown)
```

Alternatively, we can write this code with `withFinalizer` like this:

```
1 Queue.bounded(4096).withFinalizer(_.shutdown)
```

These two snippets are equivalent, but the second can be more readable, so it is helpful to be familiar with `ZIO#withFinalizer` so you can use it or understand it if someone else is using it in their code. There is also a `ZIO#withFinalizerExit` variant that provides access to the `Exit` value.

```

1 trait ZIO[-R, +E, +A] { self =>
2   def withFinalizerExit[R1](
3     f: (A, Exit[Any, Any]) => ZIO[R1, Nothing, Any]
4   ): ZIO[R1 with Scope, Nothing, A] =

```

```

5   ZIO.acquireReleaseExit(self)(f)
6 }
```

Another helpful family of operators is the `AutoCloseable` variants. Many data types implement the `AutoCloseable` trait introduced in Java 7.

```

1 package java.lang
2
3 trait AutoCloseable {
4   def close(): Unit
5 }
```

This allows data types to specify how they can be finalized, and `ZIO` has a couple of convenience operators to take advantage of this. Thus, you don't have to manually specify how these data types should be finalized.

The first of these is `ZIO.fromAutoCloseable`, which constructs a scoped resource from any workflow that produces a resource that implements the `AutoCloseable` interface:

```

1 def fromAutoCloseable[R, E, A <: AutoCloseable](
2   acquire: => ZIO[R, E, A]
3 ): ZIO[R with Scope, E, A] =
4   ZIO.acquireRelease(acquire)(a => ZIO.succeed(a.close()))
```

Notice here that we do not have to specify the `release` workflow because `ZIO` already knows that the way to close a resource that implements `AutoCloseable` is to call its `close` method. Both the `acquire` and `release` workflows will be executed uninterruptibly.

There is also a `ZIO#withFinalizerAuto` variant of this that works like `ZIO#withFinalizer` but doesn't require you to specify the finalizer.

As you can see, one of the advantages of `Scope` is that a `ZIO[R with Scope, E, A]` is a `ZIO`, so all of the operators we are familiar with for constructing ordinary `ZIO` values also work for constructing resources.

For example, the acquisition of the resource might be blocking, so we might want to give a hint of that to the `ZIO` runtime using the `blocking` operator. We can just do that with `ZIO.blocking`.

The resource acquisition might require calling a third-party API that uses callbacks to notify us when the resource is ready to be used. We can do that with the `ZIO.async` operator.

Since a resource is just a `ZIO` value, we can take advantage of all the operators we already know. We only need to learn a very small number of operators associated with resources that describe the minimal possible information about how that resource should be finalized.

## 15.4 Transforming Scoped Resources

Similarly, there are very few operators we need to learn to transform scoped resources.

Since a scoped resource is just a ZIO value, we can use all the operators we already know to transform ZIO values. These would include operators like ZIO#flatMap, ZIO#map, and ZIO#zipPar, which we have already discussed.

Only a few operators on ZIO are specific to scoped resources.

The first is one we saw before in our introduction to scoped resources, ZIO#parallelFinalizers, which allows us to modify a Scope so that when it is closed, the finalizers associated with that Scope will be run in parallel instead of sequentially:

```
1 trait ZIO[-R, +E, +A] {
2   def parallelFinalizers: ZIO[R with Scope, E, A]
3 }
```

Note that the principle of releasing resources in reverse order of acquisition typically makes sense because properly releasing a resource may depend on whether a previously acquired resource is still valid. For example, if we open a network connection and then open a file on the remote server, we need the network connection to still be open in order to close the remote file.

For this reason, you should be careful when using ZIO#parallelFinalizers to ensure that the finalization of one resource does not depend on another, typically because the resources were acquired independently. So, in our file example at the beginning of the chapter, closing the files in parallel would be perfectly fine.

There is also a ZIO#withEarlyRelease operator that gives us a handle to close resources associated with a particular workflow before the Scope is closed:

```
1 trait ZIO[-R, +E, +A] {
2   def withEarlyRelease: ZIO[R with Scope, E, (UIO[Any], A)]
3 }
```

Here, the UIO[Any] is a workflow that can be run to execute any finalizers associated with this resource even before the Scope is closed. There is also a ZIO#withEarlyReleaseExit version of this operator that allows specifying the Exit value that the finalizers should be run with.

## 15.5 Using Scoped Resources

The final step in working with Scope is to eliminate it.

The fundamental operator to do this is ZIO.scoped. This operator creates a new Scope immediately before running the workflow, provides it to the workflow, and then closes the Scope after the workflow completes, whether by success, failure, or interruption:

```
1 def scoped[R, E, A](zio: ZIO[R with Scope, E, A]): ZIO[R, E, A] =
```

```
2 | ???
```

Just like `ZIO.acquireRelease` is the fundamental operator for introducing a Scope, going from `ZIO[R, E, A]` to `ZIO[R with Scope, E, A]`, `scoped` is the fundamental operator for eliminating a Scope, going from `ZIO[R with Scope, E, A]` to `ZIO[R, E, A]`.

The `ZIO.scoped` operator also has a very fundamental interpretation as specifying the lifetime of the resource. All resources acquired in a workflow will be released at the end of the `scoped` block.

One of the advantages of the `ZIO.scoped` operator is that it is very flexible. As with other operators involving resources, we need only learn a small number of operators.

The most important principle when using `ZIO.scoped` is the same one that we discussed when we learned about `ZIO.acquireReleaseWith`. The value returned by `ZIO.scoped` should be the first one that is valid once the resources are closed.

For example, this would be a poor use of `ZIO.scoped`:

```
1 | ZIO.scoped {  
2 |   file("temperatures.txt") // don't do this!  
3 | }
```

Recall that `file` returns a `File`. But the finalizer associated with `file` closes the file handle, so trying to work with the file outside the scope will throw an exception.

Instead, we should do something with the file within `ZIO.scoped` to produce a value that is valid outside the Scope, for example, reading its contents into memory or doing something with them:

```
1 | ZIO.scoped {  
2 |   for {  
3 |     weatherData <- file("temperatures.txt")  
4 |     results      <- file("results.txt")  
5 |     _           <- analyze(weatherData, results)  
6 |   } yield ()  
7 | }
```

Now, the Scope won't be closed until `analyze` has completed execution and written our value-added analysis to the results file, so at this point, it is safe to close the file handles since we don't need them anymore. This is also the earliest point at which we could close the Scope since before that, `analyze` is still using them.

Notice also how clean this code is. It lets us specify very declaratively what we want to do: open these two files and do something with them.

We don't have to do anything ourselves except define `file` to add a finalizer to the Scope and use `scoped` to close the Scope.

Note that `ZIO` will provide an application-level Scope, and `ZIO Test` will provide a test-

level Scope if we don't provide one ourselves with `scoped`. However, at least in production code, it is better for us to use `scoped` ourselves so we do not keep resources open for the life of the application.

## 15.6 Varieties of Scoped Resources

One of the advantages of Scope is that it allows us to define resources as values in a way that is very "lightweight". We don't need any additional data type, and we can use all operators on ZIO.

As a result, resources can be used to describe many things beyond traditional resources like file handles and socket connections. Essentially, anything that has some finalization logic associated with it or has some concept of a lifetime can be thought of as a resource.

For example, consider the `ZIO#forkScoped` operator on ZIO:

```

1 trait ZIO[-R, +E, +A] {
2   def forkScoped: ZIO[R with Scope, Nothing, Fiber[E, A]] =
3     ???
4 }
```

A `fiber` is something that has a concept of a lifetime. It begins life when it is forked, and it ends life when it completes execution internally or is externally interrupted.

Usually, when we fork a fiber, we rely on ZIO's structured concurrency model to handle fiber lifetimes, but when we want to more explicitly manage the lifetime of fibers, it can be helpful to treat them as resources. For example, if we have a fiber that will continuously send a heartbeat signal that we want to continue as long as some part of our program is running, we can use `ZIO#forkScoped` to fork it in that Scope.

Here, the `acquire` action of the resource would be forking the fiber, and the `release` action would be interrupting it. By using `Scope`, we can model the lifetime of that fiber very explicitly and use all the other operators we are familiar with for working with resources.

This allows us to describe starting background processes like this as values that can easily be composed with the rest of our program. For example, we can describe a workflow that sends a heartbeat signal in the background like this:

```

1 val heartbeat: ZIO[Scope, Nothing, Fiber[Nothing, Unit]] =
2   Console.printLine(".").orDie.delay(1.second).forever.forkScoped
```

We can then compose it with the rest of our program like this:

```

1 lazy val myProgramLogic: ZIO[Any, Nothing, Unit] =
2   ???
```

```

1 ZIO.scoped {
2   for {
3     _ <- heartbeat
```

```
4     _ <- myProgramLogic
5   } yield ()
6 }
```

We were able to define `heartbeat` in a way that ensured the heartbeat signal would not continue being sent forever but would stop when the Scope was closed. At the same time, we didn't have to specify ourselves how long the heartbeat signal was but deferred that to the caller.

Similarly, if we wanted to change some configuration information, we could think of the configuration change as having a “lifetime” that describes the scope to which the change should be applied. Here, the `acquire` action for the resource would be setting the configuration to the new value, and the `release` action would be setting it back to the old value.

```
1 def withRandomScoped[A <: Random] (
2   random: Random
3 ): ZIO[Scope, Nothing, Unit] =
4 ???
```

The `withRandomScoped` operator lets us provide a custom implementation of the `Random` service, for example, if we want to use a cryptographically secure random number generator. We'll see later how we can use the `ZLayer.scoped` operator to convert any resource into a `ZLayer`, which can be an excellent way to describe these types of configuration changes as values and then apply them.

The point of these examples is not to say you need to know about these particular operators but to get you comfortable with thinking of resources as things beyond just traditional “resources” like file handles and network connections. Resources can be anything that has some concept of finalization or a lifetime.

As you can see above, this pattern is used extensively in ZIO and ZIO ecosystem libraries to describe fibers and configuration changes, among other things. So, it is helpful to understand this pattern when you see it.

Also, hopefully, this gives you ideas for things you can potentially conceptualize as resources in your code base. This pattern of treating anything that requires finalization or has a lifetime as a resource is a very powerful one that can help you take advantage of a lot of functionality in ZIO and clean up your own code.

## 15.7 Conclusion

With the materials in this chapter, you have powerful tools to define resources and work with them in a simple way that is guaranteed not to leak resources. You also actually have very little to learn to do so!

In most cases, this will be all you need, and the operators discussed in this chapter provide very strong guarantees, so if you use them, you will get resource safety without having to

think about it.

However, sometimes, you may need to do something more advanced with resources. In the next chapter, we will look more at the `Scope` data type and how we can use it to implement the operators we have learned about in this chapter. We will also look at some more advanced operators that give us more flexibility in working with resources but can also require us to be a little more careful.

If that sounds exciting, read on! Otherwise, feel free to skip ahead to the section on dependency injection.

## 15.8 Exercises

1. Assume we have written a worker as follows:

```

1 def worker(sem: Semaphore, id: Int): ZIO[Scope, Nothing,
2   Unit] =
3   for {
4     _ <- sem.withPermitsScoped(2)
5     _ <- Console.printLine(s"Request $id: Starting
6       processing").orDie
7     _ <- ZIO.sleep(5.seconds)
8     _ <- Console.printLine(s"Request $id: Completed
9       processing").orDie
10   } yield ()

```

Please explain how and why these two applications have different behavior:

Application 1:

```

1 object MainApp1 extends ZIOAppDefault {
2   def run =
3     for {
4       sem <- Semaphore.make(4)
5       _ <- ZIO.foreachParDiscard(1 to 10)(i => ZIO.scoped(
6         worker(sem, i)))
7     } yield ()
}

```

Application 2:

```

1 object MainApp2 extends ZIOAppDefault {
2   def run =
3     for {
4       sem <- Semaphore.make(4)
5       _ <- ZIO.scoped(ZIO.foreachParDiscard(1 to 10)(i =>
6         worker(sem, i)))
7     } yield ()
}

```

2. Continuing from implementing the `Semaphore` data type from the previous chapter, implement the `withPermits` operator, which takes the number of permits to acquire and release within the lifetime of the Scope:

```
1 trait Semaphore {
2   def withPermitsScoped(n: Long): ZIO[Scope, Nothing, Unit]
3 }
```

3. Write a cache service that caches values in memory. The cache service should have two methods: `put`, which puts a value in the cache with a time-to-live (TTL), and `get`, which gets a value from the cache if it is still valid.

```
1 trait Cache[K, V] {
2   def put(key: K, value: V, ttl: Duration): UIO[Unit]
3   def get(key: K): UIO[Option[V]]
4 }
5
6 object Cache {
7   def make[K, V](
8     invalidationInterval: Duration = 1.seconds
9   ): ZIO[Scope, Nothing, Cache[K, V]] = ????
10 }
```

The cache service should be up and running as long as the Scope is open:

```
1 // Example usage
2 object CacheServiceExample extends ZIOAppDefault {
3   def run =
4     ZIO.scoped {
5       for {
6         cache <- Cache.make[String, Int]()
7         _    <- cache.put("key1", 100, 2.seconds)
8         _    <- cache.put("key2", 200, 4.seconds)
9
10        v1 <- cache.get("key1")
11        _    <- ZIO.debug(s"Initial value for key1: $v1")
12
13        _    <- ZIO.sleep(3.seconds)
14        v2 <- cache.get("key1") // Should be None (expired)
15        v3 <- cache.get("key2") // Should still exist
16        _    <- ZIO.debug(s"After 3s - key1: $v2, key2: $v3")
17
18        _    <- ZIO.sleep(2.seconds)
19        v2 <- cache.get("key1") // Should be None (expired)
20        v3 <- cache.get("key2") // Should be None (expired)
21        _    <- ZIO.debug(s"After 2s - key1: $v2, key2: $v3")
22     } yield ()
23 }
```



# Chapter 16

## Resource Handling: Advanced Scopes

In this chapter, we will look more at the Scope data type and how we can use it to describe resources.

This will give us the foundation to understand some of the more advanced operators in ZIO for working with resources. These operators give us more flexibility for advanced use cases but also require us to be a little more careful.

This chapter discusses more advanced material, so it is excellent if you want to get a better understanding of scopes and resources. However, if you wish to be able to define and work with resources, you should feel free to skip this chapter, as the previous one has everything you need to know.

### 16.1 Scopes Revisited

Here is a slightly expanded definition of the interface of Scope from the previous chapter:

```
1 import zio._

2 trait Scope {
3   def addFinalizerExit(
4     finalizer: Exit[Any, Any] => UIO[Any]
5   ): UIO[Unit]
6   final def addFinalizer(finalizer: UIO[Any]): UIO[Unit] =
7     addFinalizerExit(_ => finalizer)
8 }
9
10 object Scope {
11   trait Closeable extends Scope {
```

```

13     def close(exit: Exit[Any, Any]): UIO[Unit]
14   }
15
16   def make: UIO[Scope.Closeable] =
17     ???
18 }
```

This is the same as the definition of `Scope` we examined in the previous chapter, with a couple of modifications.

First, we have generalized our definition of a finalizer from a `UIO[Any]` to a function `Exit[Any, Any] => UIO[Any]`. That is, a finalizer isn't just some workflow that we will run to release a resource but can potentially look at the `Exit` value the `Scope` was closed with to determine what finalization is required.

We can recapture our original notion of a finalizer that is just a workflow we run to release a resource simply by ignoring the `Exit` value, as we can see in the implementation of `Scope #addFinalizer`.

Second, we have separated the ability to add a finalizer to the `Scope` from the ability to close the `Scope` with the `Closeable` interface.

The `Scope.Closeable` interface allows us to add a finalizer to the `Scope` as well as to close the `Scope`. On the other hand, if we only have a `Scope`, we can add finalizers to it, but we can't close it.

This distinction is helpful because it allows us to pass a reference to a `Scope` to someone else with limited "access rights" where the user of the `Scope` can add a finalizer to the `Scope` but not close it. This prevents the user of the `Scope` from prematurely closing it when other users of the `Scope` are relying on it being open.

For example, let's look at the `ZIO.scopeWith` operator on `ZIO`. This allows us to access the current scope and do something with it:

```

1 import zio._
2
3 def scopeWith[R, E, A](f: Scope => ZIO[R, E, A]): ZIO[R, E, A] =
4   ???
```

The function `f` gets access to a `Scope`, so the implementation of `f` can do something like add a finalizer to it:

```

1 scopeWith { scope =>
2   scope.addFinalizer(ZIO.debug("Finalizing!"))
3 }
```

However, the user of the `Scope` does not have the ability to close it.

```

1 scopeWith { scope =>
2   scope.close(Exit.unit)
3 }
```

```
4 // does not compile
```

This makes sense because allowing the user of the Scope to close it would give them too much power. The user of the scope does not know what other parts of the program are potentially using resources associated with the scope, and prematurely closing the scope could violate the expectations of other parts of the program.

With these two changes, allowing finalizers to depend on the `Exit` value and separating the ability to add finalizers to a Scope from the ability to close it, our description of Scope now reflects the one in ZIO.

## 16.2 From Scopes to Resources

Now that we have a Scope, how do we go from it to describing resources?

We conceptually said that when we used `ZIO.acquireRelease` to construct a resource, we created a new workflow that required a Scope and added the finalizer associated with the resource to that Scope. But how does that actually work, and how do we maintain the guarantees we need for resource safety in the presence of complications like asynchronous interruption?

```
1 def acquireRelease[R, E, A] (
2   acquire: ZIO[R, E, A]
3 )(
4   release: A => ZIO[R, Nothing, Any]
5 ): ZIO[R with Scope, E, A] =
6   ???
```

Conceptually, we need to access a Scope in the environment, acquire the resource, and then add the finalizer associated with the resource to the Scope. Let's see how we do that.

We can access a Scope in the environment with the `ZIO.scope` operator:

```
1 def scope: ZIO[Scope, Nothing, Scope] =
2   ZIO.service[Scope]
```

This uses the `ZIO.service` operator, which we can use to access any service we need from the ZIO environment, to access a Scope. The `scope` operator is just a convenience method for `ZIO.service` when the service we are accessing is a Scope since that is a common use case.

With this, we are now in a position to implement an initial version of `ZIO.acquireRelease` ourselves:

```
1 def acquireRelease[R, E, A] (
2   acquire: ZIO[R, E, A]
3 )(
4   release: A => ZIO[R, Nothing, Any]
5 ): ZIO[R with Scope, E, A] =
```

```

6   for {
7     r      <- ZIO.environment[R]
8     scope <- ZIO.scope
9     a      <- acquire
10    _      <- scope.addFinalizer(release(a).provideEnvironment(r))
11  } yield a

```

Notice how closely the *for comprehension* syntax mirrors our description above. We access a Scope in the environment, acquire the resource, and add the finalizer associated with the resource to the Scope.

There is just one complication here, which is that the Scope doesn't know about any other services, so it needs finalizers added to it to already have all the dependencies they need to be run. The signature of `ZIO.acquireRelease` allows the finalizer to use services R, so we need to access those services ourselves and provide them to the finalizer.

Fortunately, this is easy to do, and we can access all the services described by R using the `ZIO.environment` operator and then provide them to the finalizer with the `ZIO#provideEnvironment` operator. If you don't remember these operators from our introduction to ZIO, don't worry; we will review them in the next section on dependency injection!

However, this implementation does not yet provide all the guarantees we want regarding resource safety.

To see this, consider what would happen if `ZIO.acquireRelease` was interrupted between the third and fourth lines of the *for comprehension*, after running `acquire` but before adding the finalizer to the Scope. In this case, the finalizer would never be run, and we would leak the resource.

To prevent that, we need to use the `ZIO.uninterruptible` operator to prevent us from being interrupted while we are acquiring the resource and adding the finalizer:

```

1 def acquireRelease[R, E, A](
2   acquire: ZIO[R, E, A]
3 )(
4   release: A => ZIO[R, Nothing, Any]
5 ): ZIO[R with Scope, E, A] =
6   ZIO.uninterruptible {
7     for {
8       r      <- ZIO.environment[R]
9       scope <- ZIO.scope
10      a      <- acquire
11      _      <- scope.addFinalizer(release(a).provideEnvironment(r))
12    }
13  } yield a
}

```

And that's it! We have now implemented the `ZIO.acquireRelease` operator, allowing

us to describe resources as values.

Notice how straightforward this was! While there were some things we had to be careful of, like using the `ZIO.uninterruptible` operator, overall, this was a quite straightforward implementation of functionality that could otherwise be quite complex and error-prone.

It also illustrates some of the things we will want to think about when dealing with other more complex resource acquisition scenarios. Typically, whenever we work with scopes, we will want to acquire some resource and then add the finalizer associated with that resource to a scope, and we will want to make sure we are not interrupted between those two steps.

## 16.3 Using Resources

Now that we have implemented resources with `Scope`, how do we actually use them?

We saw in the last chapter that the operator for using a resource was `ZIO.scoped`:

```
1 def scoped[R, E, A](zio: ZIO[R with Scope, E, A]): ZIO[R, E, A] =
2 ???
```

The `scoped` operator takes a workflow that requires a `Scope` and eliminates that requirement by creating a `Scope`, providing it to the workflow, and then closing the `Scope` when the workflow is done. How do we implement that?

The first part, creating the `Scope`, is covered by the `make` constructor we sketched out above:

```
1 def make: ZIO[Any, Nothing, Scope.Closeable] =
2 ???
```

We won't get into the internal implementation of `Scope` here. But conceptually, we can say that a `Scope` maintains an internal state that represents the set of finalizers associated with it, and creating a `Scope` involves allocating that internal state.

The second part, providing the `Scope` to the workflow, can be done with the normal operators on `ZIO` for providing a service required by a workflow. This functionality is wrapped up in the `extend` operator on `Scope`:

```
1 trait Scope { self =>
2   def extend[R, E, A](
3     zio: ZIO[R with Scope, E, A]
4   ): ZIO[R, E, A] =
5     zio.provideSomeEnvironment[R] { environment =>
6       environment.union[Scope](ZEnvironment(self))
7     }
8 }
```

This implementation requires the use of slightly complex operators for working with the ZIO environment that we will cover in more detail in the next section, but let's walk through it here.

We are starting with an original workflow, `zio`, that requires a set of services, `R`, and a `Scope`. We are then using the `ZIO#provideSomeEnvironment` operator, which allows us to provide some of the services required by the workflow while leaving the others as remaining dependencies.

We specify as a type parameter the services that we are leaving as remaining dependencies, in this case, `R`. We then have to provide a function `ZEnvironment[R] => ZEnvironment[R with Scope]` that describes how we can build the full set of services our workflow needs given the others that we are leaving as remaining dependencies.

Here, that implementation is relatively simple. Since we already have a `ZEnvironment[R]` and a `Scope`, we can just put the `Scope` inside a `ZEnvironment` with the `apply` constructor on `ZEnvironment` and then use the `union` operator on `ZEnvironment` to combine the set of services `R` with the `Scope`.

If this seemed a little complicated, that is okay! We'll cover this in more detail in the next section, and you shouldn't have to work with the ZIO environment at a low level like this because you can use other operators like `Scope#extend`.

Stepping back a bit, we had a workflow that needed a set of services, `R` and a `Scope`. We had a `Scope`, so we provided it, and now we have a workflow that only needs the set of services `R`.

It is important to note here what the `Scope#extend` operator does here and what it does not do. It takes a `Scope` and provides it to a workflow, eliminating that workflow's requirement for a `Scope`. However, it does not close the `Scope` when the workflow is done.

This is why this operator has the name `ZIO#extend`. It "extends" the life of any resources acquired in the workflow `zio` into the life of the `Scope`, but it does not close the scope, leaving that as the responsibility of whoever created the `Scope`.

Providing the scope and closing it when the workflow is done is the responsibility of another operator on `Scope` that builds on `Scope#extend` called `Scope.Closeable#use`:

```

1 object Scope {
2   trait Closeable extends Scope { self =>
3     def close(exit: Exit[Any, Any]): UIO[Unit]
4     def use[R, E, A](
5       zio: ZIO[R with Scope, E, A]
6     ): ZIO[R, E, A] =
7       self.extend[R, E, A](zio).onExit(self.close(_))
8   }
9
10  def make: UIO[Scope.Closeable] =
11    ???
12 }
```

The `use` operator provides a Scope to a workflow and then closes the Scope immediately after the workflow completes execution, providing the `Exit` value of the workflow to each of the finalizers in the Scope.

The `use` operator is defined on the `Closeable` interface because it closes the Scope, so it should only be accessible to a caller that has the right to close the Scope.

The implementation of `use`, like many of the other operators on `Scope`, is fairly simple. It just calls `extend` and then `onExit` to close the Scope with the `Exit` value of the workflow.

The implementation of `use` does not require any special logic around interruptibility because that should be handled by the resource itself.

That is, the resource should be responsible for ensuring that if resources requiring finalization are acquired, the finalizers associated with those resources are added to the Scope. As long as that is done, `use` can just call `onExit`, and if the workflow is interrupted, `onExit` will take care of making sure that the Scope is still closed.

Putting this all together, the implementation of `ZIO.scoped` is as simple as this:

```
1 def scoped[R, E, A](zio: ZIO[R with Scope, E, A]): ZIO[R, E, A] =  
2   Scope.make.flatMap(_.use[R, E, A](zio))
```

This is practically a one-to-one translation of our original description of what `scoped` needed to do. We create a `Scope` with the `make` constructor, provide it to the workflow, and close the workflow when the `Scope` is done with the `use` operator on `Scope`.

## 16.4 Child Scopes

Child scopes are another concept we need when implementing operators to work with resources. To motivate this, let's go back to the `ZIO#withEarlyRelease` operator from the previous chapter:

```
1 trait ZIO[-R, +E, +A] {  
2   def withEarlyRelease: ZIO[R with Scope, E, (UIO[Unit], A)]  
3 }
```

Recall that this operator's semantics are that it should return a new version of the workflow that produces both the resource acquired by the original workflow and a workflow that can be run to release the resource early. This creates the risk that we will release the resource when someone is still using it, but avoiding this is a responsibility the caller assumes when using a relatively low-level operator like this.

With our new understanding of scopes, we can start sketching out how to implement this operator and identify a couple of issues.

Conceptually, we could say we want to access a scope in the environment, and then we want to return both a workflow that will close the scope as well as the original result of the workflow:

```

1 trait ZIO[-R, +E, +A] { self =>
2   def withEarlyRelease: ZIO[R with Scope, E, (UIO[Unit], A)] =
3     ZIO.scope.flatMap(scope =>
4       scope.extend[R](self).map { a =>
5         (ZIO.fiberIdWith(id => scope.close(Exit.interrupt(id))), 
6          a)
7       }
8     )

```

However, this will not compile as written.

`ZIO.scope` just gives us access to a `Scope`, which, if we recall our discussion from the beginning of the chapter, allows us to add a finalizer to the `Scope` but not to close it. So, we aren't able to access the `close` operator on the `Scope` in the `release` workflow defined on the second line of the `for comprehension`.

Is this just the ZIO operators getting in our way?

No, it is actually telling us something important and helping us avoid making a mistake! The early release workflow should release any resources acquired by `self` but not any other resources that have been added to the same scope.

To see this, consider an example like the one below:

```

1 def resource(label: String): ZIO[Scope, Nothing, Unit] =
2   ZIO.acquireRelease {
3     ZIO.debug(s"Acquiring $label")
4   } { _ =>
5     ZIO.debug(s"Releasing $label")
6   }
7
8 ZIO.scoped {
9   for {
10     _           <- resource("A")
11     tuple       <- resource("B").withEarlyRelease
12     (release, _) = tuple
13     _           <- release
14     _           <- ZIO.debug("Using A")
15   } yield ()
16 }

```

In this program, our expectation is that we should see the following output:

```

1 Acquiring A
2 Acquiring B
3 Releasing B
4 Using A
5 Releasing A

```

Resource B was acquired as part of the workflow we called `ZIO#withEarlyRelease`, so when we run `release`, we should release that resource immediately. In the contract, A was not acquired as part of that workflow, so it should not be released until the Scope is closed after A is used.

However, if we were able to implement `ZIO#withEarlyRelease` as we sketched out above, we would actually see something like this output:

```

1 | Acquiring A
2 | Acquiring B
3 | Releasing B
4 | Releasing A
5 | Using A

```

Since the `release` workflow closes the Scope, it runs all the finalizers associated with that Scope, which causes it to prematurely release resource A. This illustrates why someone with the ability to add a finalizer to a Scope shouldn't necessarily be allowed to close it.

Now that we understand why ZIO doesn't allow us to close the Scope we access from `ZIO.scope` let's see how we can do this safely.

The answer is that we want to create our own Scope that we provide to the workflow that we call `ZIO#withEarlyRelease` on.

Since we are the creators of this Scope, we also have the ability to close it. This solves the problem of avoiding prematurely running other finalizers added to the original Scope.

However, we would also like to tie the lift of the new Scope we are creating to the life of the original Scope so that if the original Scope is closed, the new Scope will also be closed.

We can do this with the `fork` operator on Scope. The `Scope#fork` operator creates a new Scope that is a "child" of the original Scope. The child Scope will automatically be closed when the original Scope is closed.

With the `fork` operator, we can now implement `ZIO#withEarlyRelease` like this:

```

1 trait ZIO[-R, +E, +A] { self =>
2   def withEarlyRelease: ZIO[R with Scope, E, (UIO[Unit], A)] =
3     ZIO.scope.flatMap { parent =>
4       parent.fork.flatMap { child =>
5         child.extend[R](self).map { a =>
6           (ZIO.fiberIdWith(id => child.close(Exit.interrupt(id)))
7             , a)
8         }
9       }
10    }
}

```

This code will compile because we are calling `close` on the child Scope we created, which we do have the right to close. This now also has the expected semantics since we are just closing the child Scope and not the parent Scope.

## 16.5 Putting it All Together

For most use cases, working with `ZIO.acquireRelease` and its variants for constructing resources and `scoped` for using them gives us everything we need. That's why we focused on those operators in the previous chapter.

Both of these operators have the advantage of keeping the acquisition and release of resources closely tied together. This lets us write code with powerful guarantees about resource safety without having to think about it.

However, for some more advanced use cases, we may need to separate the acquisition and release of resources more. Now that we have the tools in this chapter, we have everything we need to tackle those use cases as well.

To motivate our discussion, we will work to implement a `memoized` operator that will return a memoized version of a function that acquires a resource:

```

1 | def memoize[R, E, A, B](
2 |   f: A => ZIO[R, E, B]
3 | ): ZIO[Scope, Nothing, A => ZIO[R with Scope, E, B]] =
4 |   ???

```

This operator should return a “memoized” version of the original function.

If the function has not been called with a given input before, it should call the function `f` to acquire the resource, store it, and return it. If the function has previously been called with a given input, it should immediately return the previously acquired resource.

Because we are going to acquire the resource once and potentially use it multiple times, we don't want to release it when we are done using it but instead when the outer Scope is closed. This introduces the complexity for resource management that will make this a helpful exercise to work through.

We can start by considering what work we want to do in the outer `ZIO` workflow. What state do we need to allocate to implement the semantics described above?

One piece of state it looks like we will need is a `Scope` to add finalizers associated with the resources to. Another piece of state we need is a `Map` to keep track of the resources we have previously acquired.

So, an initial implementation might look like this:

```

1 | def memoize[R, E, A, B](
2 |   f: A => ZIO[R, E, B]
3 | ): ZIO[Scope, Nothing, A => ZIO[R with Scope, E, B]] =
4 |   for {

```

```

5   scope <- ZIO.scope
6   ref   <- Ref.make(Map.empty[A, Promise[E, B]])
7 } yield ???

```

We are just requiring a Scope since we are having `memoize` return a resource itself so that our memoized resourceful function can potentially be composed with other resources.

We are initially using a `Ref` to contain the Map of previously acquired resources. The map contains a `Promise` that can be completed with a resource rather than the resource itself to handle the possibility that the function is called again with the same input while we are still acquiring the resource.

An implementation of the memoized resourceful function might then look like this:

```

1 def memoize[R, E, A, B](
2   f: A => ZIO[R, E, B]
3 ): ZIO[Scope, Nothing, A => ZIO[R with Scope, E, B]] =
4   for {
5     scope <- ZIO.scope
6     ref   <- Ref.make(Map.empty[A, Promise[E, B]])
7   } yield a =>
8     Promise.make[E, B].flatMap { promise =>
9       ref.modify { map =>
10         map.get(a) match {
11           case Some(promise) =>
12             (promise.await, map)
13           case None =>
14             (
15               scope.extend[R](f(a)).intoPromise(promise) *>
16               promise.await,
17               map + (a -> promise)
18             )
19           }
20         }.flatten
21     }

```

We are using the `extend` operator on `Scope` that we saw before to “extend” the scope of each acquired resource into the `Scope` of `memoize`. This way, all of the acquired resources will be released when the `Scope` of `memoize` is closed, but they won’t be closed before that.

This is safe because the resourceful function `f` is responsible for ensuring that finalizers associated with any acquired resources are added to the associated `Scope`. The implementation of `f` probably uses an internal operator like `acquireRelease` to do this.

Similarly, whoever is calling `memoize` needs to eliminate the requirement for a `Scope` by using an operator like `ZIO.scoped` to create a new `Scope`, provide it to the workflow, and close it when the workflow is done. This operator ensures that the `Scope` will be closed when the workflow completes execution, no matter what.

With these concerns handled for us, our implementation can actually be quite simple.

Implementing a `memoize` operator for a potentially asynchronous workflow is generally somewhat complex. We can see this in the use of the `Ref` and `Promise` operators, as well as the `modify` and `intoPromise` operators.

However, the actual complexity associated with the resourcefulness of the workflow was quite minimal.

We needed to “extend” the life of the resources from the life of one execution of the memoized resourceful function to the life of the `memoize` operator. We did this by accessing the `Scope` using the `ZIO.scope` operator and then using the `extend` operator to “extend” the life of each resource into that `Scope`.

You may encounter various other problems involving resources, but as this example illustrates, working with scopes makes them as straightforward as possible. If we need to modify the life of a resource, we can either extend it into a broader `Scope` as in this example or create a narrower scope for it like we did in our implementation of `ZIO#withEarlyRelease` with `fork`.

## 16.6 Conclusion

In this chapter, we have built up from the definition of `Scope` to implementing a variety of resource management operators ourselves.

Hopefully, you found these materials helpful in gaining a deeper understanding of how scopes are implemented and how they provide the foundation for composable resource management.

If you found the material in this chapter challenging, don’t worry about it. This chapter got more into the “how”, but the “what” in the previous chapter gives you everything you need to handle most resource management problems.

The key things to remember from both of these chapters areas follows.

Create your resources with the `ZIO.acquireRelease` constructor or, for more advanced scenarios, the `addFinalizer` operator. If you are using `addFinalizer`, make sure you are using the `uninterruptible` or `uninterruptibleMask` operators to ensure you aren’t interrupted between acquiring a resource and adding its finalizer to the `Scope`.

When you are done using the `ZIO.scoped` operator, you can eliminate your resources. This will create a `Scope` for you, provide it to your workflow, and close the `Scope` when it is done, ensuring the `Scope` is closed even if the workflow fails or is interrupted.

If you follow these simple rules, you can work with resources in a way that provides powerful guarantees about resource safety while allowing you to focus on the logic of the rest of your program instead of resource management.

If you’re comfortable with the materials in this section, we will move on to discuss how `ZIO` provides a solution to dependency management.

Dependency management builds on resource safety because many dependencies are resourceful. As we will see, dependency management is also a higher-level domain with its own problems, which is where ZIO’s `ZLayer` data type comes in.

# Chapter 17

## Dependency Injection: Essentials

ZIO's environment type is one of the most distinctive features of ZIO. In this chapter, we will gain a better understanding of ZIO's environment type and how it, along with the ZLayer data type, provides a comprehensive solution to dependency injection.

### 17.1 The Environment Type

As you may remember from the first chapter, a simple mental model of ZIO is as follows:

```
1 | type ZIO[-R, +E, +A] = ZEnvironment[R] => Either[Cause[E], A]
```

We will spend more time on ZEnvironment in the next chapter, but for now, you can think of the ZEnvironment as a map of types of services to implementations of those services. For example, a ZEnvironment[Int & String] might look like this:

```
1 | ZEnvironment(
2 |   Int -> 42,
3 |   String -> "Hello, world!"
4 | )
```

Note that we are using the `&` syntax from Scala 3 for intersection types. ZIO provides a type alias for `&` on Scala 2, so we can use this syntax on both Scala 2 and Scala 3.

A ZEnvironment[Int & String] has a value of type Int and a value of type String inside it.

Going back to the simple mental model of ZIO above, we can say that a ZIO[R, E, A] requires a set of services R to be run. We can also think of R as the *context* that R requires to be run.

For example, in the previous section, we saw that we could represent a resource as a ZIO [R `with` Scope, E, A].

A resource requires a Scope to add the finalizers associated with the resource to. We can also think of a resource as needing some context telling it what Scope the resource should be acquired in.

We can also have workflows that require more than one service or context. For instance, a workflow might be executed in the context of some Request that provides information on the current HTTP request that we are handling and might also use resources. We could represent that like this:

```

1 import zio._
2
3 trait Request
4
5 lazy val workflow: ZIO[Request & Scope, Throwable, Unit] =
6     ???
```

The interpretation of this signature is that this workflow requires *both* a Request and a Scope to be run.

When we combine workflows that require different services, we get a workflow that requires all the services required by each of those workflows:

```

1 lazy val workflow1: ZIO[Request, Throwable, Unit] = ???
2 lazy val workflow2: ZIO[Scope, Throwable, Unit] = ???
3
4 lazy val workflow3: ZIO[Request & Scope, Throwable, Unit] =
5     workflow1 *> workflow2
```

This makes sense because `workflow1` requires a Request, and `workflow2` requires a Scope. So, if we want to run `workflow1` and `workflow2`, we need both a Request and a Scope.

You should be aware of a couple of essential properties of the environment type.

First, an environment type of Any represents a workflow that does not require any services to run.

A ZEnvironment [Int & String] is a bundle of services that promises to contain both a value of type Int and a value of type String inside it. In contrast, a ZEnvironment [Any] is an empty bundle of services that does not contain any services inside of it.

So, a ZIO [Any, E, A] is a workflow that does not require any services or context and that is ready to be run.

Second, Any, representing an empty set of services, is an identity element with respect to & and can be freely introduced or eliminated in type signatures:

```

1 lazy val zio1: ZIO[Scope, Throwable, Unit]      = ???
2 lazy val zio2: ZIO[Scope & Any, Throwable, Unit] = zio1
```

```
3 | lazy val zio3: ZIO[Scope, Throwable, Unit]      = zio2
```

Since Any represents not requiring any services, “requires a Scope” is identical to “requires a Scope and no other services”. Sometimes IDEs insert unnecessary Any types into the environment in type signatures, so this tells us that it is always safe to remove those.

Third, & is associative and commutative, so the order of types in the environment type doesn’t matter. The environment type just describes the “bundle” of services we require, so needing a Scope and a Request is the same as needing a Request and a Scope:

```
1 | lazy val zio4: ZIO[Scope & Request, Throwable, Unit] = ???  
2 | lazy val zio5: ZIO[Request & Scope, Throwable, Unit] = zio4
```

Fourth, it doesn’t matter how often a service appears in the environment type. Requiring a Scope and a Scope is no different than just requiring a Scope, so we can always eliminate duplicate entries like this.

```
1 | lazy val zio6: ZIO[Scope, Throwable, Unit]      = ???  
2 | lazy val zio7: ZIO[Scope & Scope, Throwable, Unit] = zio6  
3 | lazy val zio8: ZIO[Scope, Throwable, Unit]      = zio7
```

Again, these properties may seem obvious if you think of the environment type as representing the set of services that our application requires, but it can be helpful to state them formally.

## 17.2 Fundamental Operators for Working with the Environment

With this understanding of the ZIO environment type as representing the services or context that a workflow requires, we can recognize that there are two fundamental operators for working with the environment.

The first is the `ZIO.environment` operator, which allows us to access a set of services in the environment and *introduces* a dependency on those services:

```
1 | object ZIO {  
2 |   def environment[R]: ZIO[R, Nothing, ZEnvironment[R]] =  
3 |     ???  
4 | }
```

There are a variety of convenient methods for accessing the environment. One of the most common is the `ZIO.service` constructor, which accesses a single service in the environment instead of a bundle of services:

```
1 | object ZIO {  
2 |   def service[R: Tag]: ZIO[R, Nothing, R] =  
3 |     ???  
4 | }
```

Once we access a service or set of services from the environment, we can do anything we want with them using other operators on ZIO such as `ZIO#map` and `ZIO#flatMap`:

```
1 | ZIO.service[Scope].flatMap(scope => scope.addFinalizer(????))
```

Here, we access a `Scope` in the environment and then use `flatMap` to do something with it, in this case, adding a finalizer to it. ZIO also has helpful shorthands for this with the `with` and `withZIO` suffixes for when we want to access a service and `map` it or `flatMap` it, so we could rewrite the above example as follows:

```
1 | ZIO.serviceWithZIO[Scope](scope => scope.addFinalizer(????))
```

As you can see, these operators can make your code slightly more concise, but if you just remember the `ZIO.environment` and `ZIO.service` operators, that is totally fine.

One thing that can be somewhat confusing for people when learning about the environment is wondering, “Where does the service come from?”

In the example above we are working with a `Scope` as if there is already a `Scope` by calling operators on it. But we don’t have a `Scope` yet!

So how does this work? The answer is that the workflow above is just a description that needs a `Scope` and says when it gets a `Scope`, here is what it will do with it.

Another way to think about it is that the ZIO environment type allows us to “borrow” a service we don’t have yet. As long as we “pay back the loan” by providing an implementation of the service, we can run our program.

This brings us, then, to the second fundamental operator for working with the environment, `ZIO#provideEnvironment`. Whereas `ZIO.environment` introduces a dependency on a bundle of services, `ZIO#provideEnvironment` eliminates that dependency by providing an actual implementation of those services:

```
1 | lazy val needsAnInt: ZIO[Int, Nothing, Unit] =
2 |   ZIO.serviceWithZIO[Int](ZIO.debug(_))
3 |
4 | lazy val readyToRun: ZIO[Any, Nothing, Unit] =
5 |   needsAnInt.provideEnvironment(ZEnvironment(1))
```

The `needsAnInt` workflow depends on an `Int` and prints it out for debugging purposes. By using the `ZIO#provideEnvironment` operator and giving it an `Int` we get a new workflow that does not need anything and is ready to be run.

The `ZIO#provideEnvironment` operator is relatively low-level, and you will not typically be using it yourself, but it is helpful to understand the fundamental operators for working with the environment before moving on to higher-level constructs.

## 17.3 Typical Uses for the Environment

The ZIO environment type reflects the fundamental reality of computation: workflows can require some context to be run. However, many developers are unfamiliar with such an ergonomic way of representing this, so a common question is when and how the environment should be used.

The most common situation in which the environment is used in ZIO libraries is to represent some context or capability that can be locally eliminated.

The use of the `Scope` data type described in the previous section is a prototypical example of this.

As we discussed, the `Scope` in the environment can be thought of as representing a workflow's requirement for some context, indicating what scope its resources should be acquired in. It can also be thought of as representing requiring the *capability* to add finalizers somewhere.

Data types like `Scope` that represent capabilities like this are typically accompanied by an operator that provides the required capability and eliminates the requirement. For example, in the case of `Scope`, the `ZIO.scoped` operator provides the capability to add resources to a `Scope` by creating the `Scope` and closing it after the workflow is done.

Another data type in ZIO that follows a similar pattern is the `ZState` data type, which represents some local state that can be accessed and updated during the execution of a workflow.

We can get or update the state using operators like `ZIO.getState` and `ZIO.updateState`, which introduce a dependency on some state. We can eliminate the requirement for this capability using the `ZIO.stateful` operator and providing an initial state, which will then be threaded through the computation.

Expect to find similar examples in other ZIO ecosystem libraries for context, like a `Transaction` that represents a database transaction. Such a requirement might be introduced by various operators and then eliminated by a `transactionally` operator, which would create a new transaction, run the workflow as part of that isolated transaction, and then commit the transaction.

These types of capabilities are distinguished by typically being locally introduced and eliminated.

For example, we don't typically have one `Scope` for our entire application but instead have various scopes representing the life of different resources. Similarly, we would not typically have one `Transaction` for the entire application but would do certain local sets of database operations transactionally.

Another situation in which the environment can be used is to represent some service that our workflow depends on. To understand this, it is helpful to think about the “onion architecture”, which ZIO highly encourages.

## 17.4 The Onion Architecture

To motivate our discussion of the onion architecture, let's think about a simple application that will read from and potentially write to certain GitHub repositories. For example, it might allow us to view all open issues in ZIO ecosystem libraries and comment that we are going to work on one of them.

If we are rapidly prototyping an initial version of our application, we might do everything in one place like this:

```

1 object Main extends ZIOAppDefault {
2
3   val run =
4     ??? // All my program logic here
5 }
```

Here, the implementation of the `run` method would call the GitHub API using some HTTP library such as `zio-http`. We would call the operators provided with `zio-http` with the appropriate parameters for the GitHub API.

We would sequence these calls to reflect our program logic. For example, we might first get all the open issues, render them to the user, get the user input for the issue they want to work on, and finally post a comment.

While this programming style could be useful for initial prototyping, like making sure we can call the API at all, writing code this way will quickly lead to a couple of problems.

First, we have created a bit of “spaghetti code” where our implementation of `run` mixes together several different layers of program logic. In this program, we have at least three of these layers of logic:

- **Business Logic** - The “business logic” of our application describes *what* we want to do in plain terms independent of *how* we want to do it. Here, our business logic might be something like “get all the open issues, present them to the user, get user input on the issue they want to work on, and add an appropriate comment”.
- **GitHub API Logic** - The GitHub API logic describes how those business concepts, like getting all the issues or posting a comment, should be translated into the domain of the GitHub API. It probably describes the URL we should call to perform a certain action, but still doesn't know anything about what to do with that URL.
- **HTTP Logic** - The HTTP logic knows how to actually make a request to that URL and return a response. It is likely implemented in terms of a particular library, such as `zio-http`, and makes a variety of implementation decisions.

By mixing all of these layers together in implementing the `run` method, we have made it harder to reason about any of them. Any time we want to modify our business logic, we also potentially have to modify the GitHub and HTTP logic, making it more difficult for us to refactor our code and increasing the risk of introducing bugs.

Another way to think of this is that we have introduced a “tight coupling” between these different layers. If we want to use a different HTTP library or a different version of GitHub's

API, it won't be easy to do that without changing the rest of our code.

This will make our code less testable because we have no way of substituting in a "test" version of one of these layers. For example, it would be helpful to be able to substitute in a "test" HTTP service that returns specified responses for certain requests, but we have no way to easily do that when all of our code is mixed together.

It can also make it more difficult to maintain a large code base with multiple teams. With many teams working on the same code, the risk of conflicts increases, or teams may step on each other's toes.

So how do we avoid this?

The pattern we recommend is called the "onion architecture", and the fundamental idea is to represent each of these different "layers" as its own service. In the "onion" analogy, the business logic is at the center of the onion, and each other layer "translates" that business logic into something closer to the outside world.

Here is what that might look like in code:

```
1 trait Issue
2 final case class Comment(text: String) extends Issue
3
4 trait BusinessLogic {
5   def run: ZIO[Any, Throwable, Unit]
6 }
7
8 trait GitHub {
9   def getIssues(
10     organization: String
11   ): ZIO[Any, Throwable, Chunk[Issue]]
12   def postComment(
13     issue: Issue,
14     comment: Comment
15   ): ZIO[Any, Throwable, Unit]
16 }
17
18 trait Http {
19   def get(
20     url: String
21   ): ZIO[Any, Throwable, Chunk[Byte]]
22   def post(
23     url: String,
24     body: Chunk[Byte]
25   ): ZIO[Any, Throwable, Chunk[Byte]]
26 }
```

Each "layer" of the onion is implemented exclusively in terms of the next outer layer. For example, the `BusinessLogic` service is implemented exclusively in terms of the `GitHub`

service.

Here is what a very simple implementation of our business logic might look like:

```

1 final case class BusinessLogicLive(github: GitHub)
2   extends BusinessLogic {
3
4   val run: ZIO[Any, Throwable, Unit] =
5     for {
6       issues <- github.getIssues("zio")
7       comment = Comment("I am working on this!")
8       - <- ZIO.getOrFail(issues.headOption).flatMap { issue =>
9         github.postComment(issue, comment)
10      }
11    } yield ()
12 }
```

This is an elementary implementation, but we can still see a couple of important things here.

The logic of this program is very declarative. We're saying we want to get all the issues, and then we want to take the first issue, but we're not worried at all about how we are doing these things.

We are able to do this because we delegate all of the "how" to the next layer of the onion, which we represent as constructor arguments to the `BusinessLogicLive` implementation of the `BusinessLogic` service. This makes the code at this level of the program straightforward to read and reason about.

We can similarly implement the GitHub service in terms of the `Http` service.

```

1 final case class GitHubLive(http: Http) extends GitHub {
2   def getIssues(
3     organization: String
4   ): ZIO[Any, Throwable, Chunk[Issue]] =
5     ???
6   def postComment(
7     issue: Issue,
8     comment: Comment
9   ): ZIO[Any, Throwable, Unit] =
10    ???
11 }
```

Implementing this service will require looking at the actual GitHub API, but we can see that if we already have an `Http` service, the implementation of these operators will be relatively straightforward as well.

To implement `getIssues`, we will call `get` with a URL that corresponds to getting all the repos in the organization, and then we will use an operator like `ZIO.foreach` to make

another `get` request corresponding to each repo for all the issues for that repo. We will then just combine all of those together.

Similarly, to implement `postComment`, we will just call `post` using the `Http` service, translating the `Issue` and `Comment` into the appropriate URL.

There is definitely some logic here in looking up the appropriate URL we should use for each of these, but our challenge is basically just converting data structures into a URL. We don't have to worry at all about how we are actually going to call that URL because the `Http` service will take care of that for us.

Finally, the `Http` service will take care of the actual logic of making HTTP requests:

```

1 final case class HttpLive() extends Http {
2   def get(
3     url: String
4   ): ZIO[Any, Throwable, Chunk[Byte]] =
5     ???
6   def post(
7     url: String,
8     body: Chunk[Byte]
9   ): ZIO[Any, Throwable, Chunk[Byte]] =
10    ???
11 }
```

There is obviously some complex logic that goes into actually implementing an HTTP client, but notice that at this point, the logic of the `Http` service is highly generic, so we can probably implement it directly in terms of an HTTP library like `zio-http`, possibly making various decisions about configuration.

In this case, the `HttpLive` service did not have any further dependencies, which we represented as a case class with an empty arguments list, but we could have more layers. For example, the `HttpLive` implementation might depend on an `HttpConfig` service that specified various configuration settings.

We also don't need to have each of our layers depend on just one other layer. For example, the `BusinessLogic` layer might also depend on a `UserInterface` layer that provided functionality for displaying information to the user and getting user input, which we would just reflect as another constructor argument.

This “onion architecture” approach has a couple of benefits.

First, it gives us a natural way to break down our problem into smaller problems.

This is not the most complex application, but even at this level, there is a decent amount going on between what we actually want to do, the GitHub API, and the HTTP client. Even with a relatively simple program like this, it can be challenging to keep in our heads.

Using the onion architecture pattern makes things a lot easier because each service only has to solve one specific problem. The business logic layer service describes what we want to do; the GitHub service just translates those things into the appropriate URL and calls

the HTTP service; the HTTP service just knows how to make generic HTTP requests.

This way, we can just focus on one part of our problem and move methodically from one to the other, knowing that when we put all those solutions together, they will fit together the right way. This is also extremely helpful with teams, as we can imagine on larger projects, different team members or teams might be responsible for implementing each of these services.

It also makes our code very testable. Since each of these services is just a “building block” that snaps together, we can substitute in a different implementation of one or more of these services. For example, we could use a test implementation of the `Http` service to test the rest of our application in a deterministic way.

Now that we have all of these services, how do we actually combine them and run our program? In this simple case, we could do it like this:

```
1 object Main extends ZIOAppDefault {  
2  
3     val http: Http                  = HttpLive()  
4     val github: GitHub                = GitHubLive(http)  
5     val businessLogic: BusinessLogic = BusinessLogicLive(github)  
6  
7     val run =  
8         businessLogic.run  
9 }
```

To actually run our application, we need to “wire it up” by creating each of our services, starting on the outside of the onion and working our way in.

This simple example helps us verify that we can actually combine these different layers to get a program we can run and substitute in different layers. But in real life, things are rarely as simple as this.

Constructing each of these layers may require ZIO workflows. For example, the `Http` service may need to load configuration information from a file.

Each of these layers may also require finalization. Starting the `Http` service may open a network connection that we need to close when we are done using the service.

The same service may also appear as a dependency of multiple other services. When this happens, we still only want to build each service once.

All of these combine to create a situation where, if we are not careful, our main method, where we “wire up” all of our dependencies, can itself become a mess of highly nested spaghetti code that is hard to reason about. This is a typical problem we have observed in large applications with a hundred or more dependencies before introducing ZIO.

## 17.5 Layers

To solve this problem, ZIO provides a data type called a `ZLayer`. A `ZLayer` is a “recipe” for building some service:

```
1 trait ZLayer[RIn, E, ROut]
```

A `ZLayer[RIn, E, ROut]` is a “recipe” for building a bundle of services `ROut` given a bundle of services `RIn` and potentially failing with an error `E`.

For example, extending our example above, a `ZLayer[Http, Nothing, GitHub]` is a recipe for building a `GitHub` service given an `Http` service.

Layers can be effectual and resourceful, so they can describe all the necessary logic of setting up a service like the `Http` service above and tearing it down when we are done with it. They also let us automatically wire together all of our application’s dependencies using ZIO’s support for automatic layer construction.

We’ll see that functionality in action shortly, but first, let’s look at how we can construct layers.

### 17.5.1 Constructing Layers

The simplest way to construct a layer is using the `ZLayer.fromFunction` constructor. This constructor is appropriate when the service requires no initialization and finalization.

For example, the `BusinessLogic` service would probably be a good candidate for using the `ZLayer.fromFunction` constructor because it does not appear to require any initialization or finalization. The same thing could be said of the `GitHub` service:

```
1 object BusinessLogicLive {
2   val layer: ZLayer[GitHub, Nothing, BusinessLogic] =
3     ZLayer.fromFunction(BusinessLogicLive(_))
4 }
5
6 object GitHubLive {
7   val layer: ZLayer[Http, Nothing, GitHub] =
8     ZLayer.fromFunction(GitHubLive(_))
9 }
```

The `ZLayer.fromFunction` constructor is based on the idea that the `apply` method of any `case class` can be viewed as a function from the constructor parameters of the case class to the case class itself. So we can view `BusinessLogic.apply` as a function `GitHub => BusinessLogic`.

The `ZLayer.fromFunction` just lifts that function into a `ZLayer` context. So now we have a `ZLayer` that says, “If you give me an implementation of the `GitHub` service, I can give you an implementation of the `BusinessLogic` service”.

The `ZLayer.fromFunction` constructor is very concise, so it is excellent to have layers

that implement some higher-level logic in terms of lower-level logic but don't do any initialization or finalization.

If we need to do initialization, we can use the `apply` method of `ZLayer` with a *for comprehension*. For example, let's say that the `HttpLive` service also has a `start` method that needs to be run before making `get` or `post` requests to establish the connection:

```

1 final case class HttpLive() extends Http {
2   def get(
3     url: String
4   ): ZIO[Any, Throwable, Chunk[Byte]] =
5     ???
6   def post(
7     url: String,
8     body: Chunk[Byte]
9   ): ZIO[Any, Throwable, Chunk[Byte]] =
10    ???
11   def start: ZIO[Any, Throwable, Unit] =
12     ???
13 }
```

We can then use the `ZLayer.apply` constructor to describe how to construct this service like this:

```

1 object HttpLive {
2   val layer: ZLayer[Any, Throwable, Http] =
3     ZLayer {
4       for {
5         http <- ZIO.succeed(HttpLive())
6         _      <- http.start
7       } yield http
8     }
9 }
```

This layer will ensure that any time we build the `Http` service, we start it before we do anything else with it.

If the service requires finalization, we can use the `ZLayer.scoped` constructor to ensure any finalizers associated with the service are run when we are done using it. Let's assume that the `HttpLive` service also had a `shutdown` method that needed to be called when we were done using it:

```

1 final case class HttpLive() extends Http {
2   def get(
3     url: String
4   ): ZIO[Any, Throwable, Chunk[Byte]] =
5     ???
6   def post(
7     url: String,
```

```

8     body: Chunk[Byte]
9   ): ZIO[Any, Throwable, Chunk[Byte]] =
10    ???
11  def start: ZIO[Any, Throwable, Unit] =
12    ???
13  def shutdown: ZIO[Any, Nothing, Unit] =
14    ???
15 }
```

We could make sure that the service was shutdown properly when we were done using it like this:

```

1 object HttpLive {
2   val layer: ZLayer[Any, Throwable, Http] =
3     ZLayer.scoped {
4       for {
5         http <- ZIO.succeed(HttpLive())
6         -> http.start
7         -> ZIO.addFinalizer(http.shutdown)
8       } yield http
9     }
10 }
```

If your service depends on other services, it is easy to do that too with this pattern. Just access each of these services in the *for comprehension* using the `ZIO.service` operator we discussed above:

```

1 trait HttpConfig
2
3 final case class HttpLive(config: HttpConfig) extends Http {
4   def get(
5     url: String
6   ): ZIO[Any, Throwable, Chunk[Byte]] =
7     ???
8   def post(
9     url: String,
10    body: Chunk[Byte]
11  ): ZIO[Any, Throwable, Chunk[Byte]] =
12    ???
13   def start: ZIO[Any, Throwable, Unit] =
14     ???
15   def shutdown: ZIO[Any, Nothing, Unit] =
16     ???
17 }
18
19 object HttpLive {
20   val layer: ZLayer[HttpConfig, Throwable, Http] =
```

```

21 ZLayer.scoped {
22   for {
23     config <- ZIO.service[HttpConfig]
24     http   <- ZIO.succeed(HttpLive(config))
25     -      <- http.start
26     -      <- ZIO.addFinalizer(http.shutdown)
27   } yield http
28 }
29 }
```

### 17.5.2 Providing Layers

Now that we know how to construct layers, how do we wire them up?

Let's see what happens if we try to access the `BusinessLogic` service and run it.

```

1 object Main extends ZIOAppDefault {
2
3   val run =
4     ZIO.serviceWithZIO[BusinessLogic](_.run)
5 }
```

We will get a helpful error message that tells us that we need to provide an implementation of the `BusinessLogic` service.

We do that with the `provide` operator, which lets us provide a layer to a `ZIO` workflow that needs the services constructed by that layer. The guarantee of `provide` is that the services will be acquired before the workflow is run and released immediately after the workflow completes execution, whether by success, failure, or interruption.

So what happens if we provide the `BusinessLogic` service like this?

```

1 object Main extends ZIOAppDefault {
2
3   val run =
4     ZIO
5       .serviceWithZIO[BusinessLogic](_.run)
6       .provide(BusinessLogicLive.layer)
7 }
```

We now get a new error message telling us that our application still requires a GitHub service, which is required by the `BusinessLogic` service.

We can follow the advice of the compiler and provide an implementation of the `BusinessLogic` service like this:

```

1 object Main extends ZIOAppDefault {
2
3   val run =
```

```

4   ZIO
5     .serviceWithZIO[BusinessLogic](_.run)
6     .provide(
7       BusinessLogicLive.layer,
8       GithubLive.layer
9     )
10 }

```

To provide additional services, we just call the `ZIO#provide` operator with all the services we want to provide. We don't need to worry about the order in which we provide these services since `ZIO` will automatically wire them together in the right way.

The `GitHubLive` layer requires an `Http` service, so now we will get a new compiler error telling us we need to provide one. We can do so in the same way:

```

1 object Main extends ZIOAppDefault {
2
3   val run =
4     ZIO
5       .serviceWithZIO[BusinessLogic](_.run)
6       .provide(
7         BusinessLogicLive.layer,
8         GitHubLive.layer,
9         HttpLive.layer
10      )
11 }

```

Now that we have provided our application with everything it needs, we can go ahead and run our program.

If we want, we can also compose layers independently of providing them to our application. For example, we might want to combine the `GitHub.live` layer and the `Http.live` layer to create a new layer that constructs the `GitHub` service and has no further dependencies.

We can do that with the `ZLayer.make` operator, which provides exactly the same automatic layer construction functionality we saw above. We need to specify to `make` the services we are trying to build since we aren't providing it to a specific workflow with known requirements that the compiler can use to infer it like with `ZIO#provide`.

```

1 val githubLayer: ZLayer[Any, Throwable, GitHub] =
2   ZLayer.make[GitHub](
3     GitHubLive.layer,
4     HttpLive.layer
5   )

```

This pattern can be very useful when we have a large number of services. It helps us build reasonably sized groups of services and then compose them together.

For instance, our business logic might, at a high level, depend on a `GitHub` service and a

UserInterface service, each of which has many dependencies of their own. Building these two bundles of services and then providing them or our business logic could be a very straightforward way for us to organize our application wiring.

If we aren't ready to provide all of our services yet, we can only use the ZIO#provideSome or ZIO.makeSome variants, which allow us to specify the services we are not providing yet.

We might use this to wire up our application with all of the services it needs except for the Http service, so we could then provide different test implementations later. We could do that like this:

```

1 | val testable: ZIO[Http, Throwable, Unit] =
2 |   ZIO(
3 |     .serviceWithZIO[BusinessLogic](_.run)
4 |     .provideSome[Http](
5 |       BusinessLogicLive.layer,
6 |       GitHubLive.layer
7 |     )

```

We have to specify the type of the services we are not providing yet in ZIO#provideSome so the framework knows that we are intentionally deferring providing these services and does not give us a compilation error for not providing them.

Layers also come with a variety of other useful features. We can add a ZLayer.Debug.tree layer to see a helpful console rendering of the entire dependency graph we are constructing, or use the ZLayer.Debug.mermaid layer to obtain a link to a graphical depiction that we can use in architecture diagrams.

As you can see, layers and the ZIO environment type give us the comprehensive tool kit we need to solve the full range of dependency injection problems. They help us keep our code nicely organized and avoid the mess of spaghetti code we can otherwise often find ourselves stuck with when we need to wire everything together.

Each layer can describe any necessary initialization and finalization logic associated with that service, which is impossible with traditional constructor-based dependency injection. As long as we have all the layers we need, we can just list them all when we call ZIO#provide, and ZIO will automatically wire them together.

ZIO also gives us some powerful guarantees when we do this. Layers will be acquired in parallel to the maximum extent possible, and no layer will be acquired more than once, even if it appears in multiple places in the dependency graph.

In this way, ZIO makes dependency injection a breeze.

## 17.6 Accessors

When we need to call an API from a service, we first need to obtain access to that service. Once we have access, we can call any method on the service:

```

1 val app: ZIO[BusinessLogic, Throwable, Unit] =
2   for {
3     business <- ZIO.service[BusinessLogic]
4     _           <- business.run
5   } yield ()

```

Alternatively, we can use the `with` suffix, such as `ZIO.serviceWithZIO`, to achieve the same result in a single step:

```

1 val app: ZIO[GitHub, Throwable, Unit] =
2   ZIO.serviceWithZIO[BusinessLogic](_.run)

```

The `ZIO.service` operator is used to access a specific service from the ZIO environment, and the `ZIO.serviceWith*` operators are utilized to access specific functionalities of the given service from the environment.

To make our code even more concise and ergonomic, we can define accessor methods in the companion object of the service trait, i.e `BusinessLogic`:

```

1 object BusinessLogic {
2   def run: ZIO[BusinessLogic, Throwable, Unit] =
3     ZIO.serviceWithZIO(_.run)
4 }

```

Now the main code can be simplified to:

```

1 object Main extends ZIOAppDefault {
2   val run =
3     BusinessLogic.run
4       .provide(
5         BusinessLogicLive.layer,
6         GitHubLive.layer,
7         HttpLive.layer
8       )
9 }

```

This pattern is common in ZIO ecosystem libraries and is an effective way to make your code more concise and readable. Whenever we use methods from a service, such as `BusinessLogic.run`, the compiler automatically infers the type of the required service. If we forget to provide any necessary services, the compiler will generate a helpful error message prompting us to include them.

We can do the same thing for all the services in our application. Here is an example for the GitHub service:

```

1 object GitHub {
2   def getIssues(organization: String): ZIO[GitHub, Throwable,
3             Chunk[Issue]] =
4     ZIO.serviceWithZIO(_.getIssues(organization))

```

```

4
5  def postComment(issue: Issue, comment: Comment): ZIO[GitHub,
6    Throwable, Unit] =
7    ZIO.serviceWithZIO(_.postComment(issue, comment))
}
```

## 17.7 Service Pattern

The pattern introduced in this chapter for writing modular and testable services is called “Service Pattern” in the ZIO ecosystem. This pattern comprises five key steps, which are summarized as follows:

1. Defining a service trait/interface that describes the functionality of the service.
2. Implementing the service in a concrete case class.
3. Introducing service dependencies via the class constructor.
4. Writing a service layer, and defining its initializer and finalizer logic.
5. Writing accessors for the service to make the DSL more ergonomic.

This pattern closely resembles object-oriented programming, where we define a service as a trait and extend it in a concrete class for implementation. The service pattern effectively decouples the service’s implementation from its usage, making the code easier to test and maintain.

In this chapter, we have seen an example of the service pattern in action. Let’s summarize the steps we followed:

1. We defined the `BusinessLogic`, `GitHub`, and `Http` services as traits, along with their corresponding data types, such as `Issue` and `Comment`.
2. Then, we implemented the services in concrete classes, e.g. `BusinessLogicLive`, `GitHubLive`, `HttpLive`.
3. We introduced service dependencies via the constructor of the concrete classes, where, for instance, `GitHubLive` depends on `Http`.
4. We created service layers within the companion objects of the concrete classes, e.g. `BusinessLogicLive.layer`, `GitHubLive.layer`, `HttpLive.layer`.
5. Finally, we implemented accessors for the services to streamline the DSL, including methods like, e.g. `BusinessLogic.run`, `GitHub.getIssues`, `GitHub.postComment`.

By following these steps, the service pattern provides a structured approach to develop robust and maintainable services like what we have seen in the object-oriented programming paradigm. This pattern ensures that program logic is clearly separated from implementation details, allowing the code to remain independent of concrete class implementations. As a result, testing and maintenance become significantly easier, as changes in the implementation do not affect the core logic of the application.

This flexibility enables developers to easily swap out implementations based on requirements, such as using a mock service for testing or a production service for deployment. Ultimately, the compiler seamlessly wires everything together, ensuring that all depen-

dencies are correctly resolved at compile time.

## 17.8 Declaring Dependencies via ZIO Environment or Constructor Arguments?

Step three of the service pattern is particularly important because it requires us to declare dependencies exclusively in the constructor of the service implementation. One might wonder why we don't simply use the ZIO environment to pass dependencies around? The answer lies in the distinction between defining service capabilities on a trait and the implementation itself.

When we define the service interface, we are specifying the behavior of the service without any concern for its dependencies; therefore, the service interface should not be aware of them. This is why, in most cases, the methods of a service in a trait have an environment type of Any, and we should not use the environment inside service definitions.

However, when we implement the service, we often find that certain dependencies are required. Consequently, if the service interface methods have an environment type of Any, the implementation must adhere to the same signature and cannot include dependencies in the environment type.

So, where should these dependencies be declared? Since the ZIO environment is not suitable for this purpose, we have another option: the class constructor. We should accept all the dependencies inside the constructor of the service implementation.

## 17.9 Conclusion

In this chapter we looked at the fundamentals of the ZIO environment type as well as how we can organize our applications into services to make them modular. We also saw how layers let us describe the logic for constructing each of these services, including initialization and finalization, and easily wire them together with automatic layer construction and helpful error messages.

In the next chapter we are going to look more at the implementation of the ZEnvironment data type as well as at some more advanced operators for working with the environment and layers. As with the chapter on advanced scopes, this material is great if you want to go deeper, but the materials in this chapter should give you all you need to know so also feel free to skip ahead to the next section on software transactional memory.

## 17.10 Exercises

1. What is the purpose of Tag[A], which is commonly used in the ZIO library, particularly in the ZEnvironment operators?
2. In this chapter, we introduced services that were monomorphic, meaning they operate on a single type. Now, let's create a key-value store service that can store and

retrieve values of any type using the service pattern. Ensure that the service is polymorphic over the types of keys and values by implementing the following trait:

```
1 trait KeyValueStore[K, V, E, F[_], _] {
2   def get(key: K): F[E, V]
3   def set(key: K, value: V): F[E, V]
4   def remove(key: K): F[E, Unit]
5 }
```

To implement this service, you will need a deep understanding of the Tag[A] and LightTypeTag types, which are essential for the ZEnvironment operators.

## Chapter 18

# Dependency Injection: Advanced Dependency Injection

In the previous chapter, we explored the basics of dependency injection and learned how to use the ZIO environment to access and provide dependencies within our application. In this chapter, we will dive deeper into the inner workings of the ZEnvironment. We will cover how to provide multiple services of the same type, handle errors in layer construction, implement memoization of dependencies, and utilize automatic ZLayer derivation.

### 18.1 What is ZEnvironment?

As we learned in the previous chapter, we can obtain and access various services from the ZIO environment using ZIO environment accessors while developing our application logic without worrying about how these services will be implemented or provided. The ZEnvironment is a key component of ZIO that is responsible for managing and providing these services to our application. It is a type-level map from a service type to the service implementation. When we provide (ZIO#provide) a service of type A to a ZIO effect, we are essentially adding a map entry from A type to the service implementation in the ZEnvironment. So whenever we need a service of type A, we can ask the ZEnvironment for it, and it will provide the corresponding service if the service is available in the environment.

```
1 import zio._
2
3 object ZEnvironmentExample extends App {
4   trait Foo {
5     def bar: Int
6   }
7
8   val zenv: ZEnvironment[String with Foo with Double] =
```

```

9   ZEnvironment.empty
10    .add("Hello!")
11    .add(new Foo { def bar = 42 })
12    .add(1.3)
13
14  println(s"String: ${zenv.get[String]}")
15  println(s"foo.bar: ${zenv.get[foo].bar}")
16  println(s"Double: ${zenv.get[Double]}")
17 }

```

When we incorporate the ZEnvironment into a ZIO application, we can access the services from the environment using the `ZIO#service*` methods and provide the services to the ZIO application using the `ZIO#provide*` methods:

```

1 import zio._
2
3 object ZEnvironmentWithZIOExample extends ZIOAppDefault {
4   trait Foo {
5     def bar: Int
6   }
7
8   def run = {
9     for {
10       s <- ZIO.service[String]
11       _ <- ZIO.debug(s"String: $s")
12       f <- ZIO.service[foo]
13       _ <- ZIO.debug(s"foo.bar: ${f.bar}")
14       d <- ZIO.service[Double]
15       _ <- ZIO.debug(s"Double: $d")
16     } yield ()
17   }.provideEnvironment(
18     ZEnvironment.empty
19       .add("Hello!")
20       .add(new Foo { def bar = 42 })
21       .add(1.3)
22   )
23 }

```

We usually don't use the ZEnvironment directly in our application code, so instead of `ZIO#provideEnvironment`, we can use the `ZIO#provide` method to provide the services to the ZIO application:

```

1 app.provide(
2   ZLayer.succeed("Hello!"),
3   ZLayer.succeed(new Foo { def bar = 42 }),
4   ZLayer.succeed(1.3)
5 )

```

## 18.2 Providing Multiple Services of the Same Type

You might be wondering what would happen if we had multiple services of the same type. Is there a way to distinguish between them? By default, ZIO only allows one service of a given type in the environment. But in some cases, we might need multiple of them, e.g., one for the production environment and one for the development environment. In such cases, we can use the `ZIO.serviceAt` method to access the service of a specific type with a particular key:

```

1 import zio._
2
3 val app: ZIO[Map[String, Foo], Option[Nothing], Unit] =
4   for {
5     foo1 <- ZIO.serviceAt[foo1]("foo1").flatMap(f => ZIO.
6       fromOption(f))
7     _      <- ZIO.debug(s"foo1.bar: ${foo1.bar}")
8     foo2 <- ZIO.serviceAt[foo2]("foo2").flatMap(f => ZIO.
9       fromOption(f))
10    _      <- ZIO.debug(s"foo2.bar: ${foo2.bar}")
11  } yield ()

```

To provide the services with keys, we can use the `ZIO.provideEnvironment` method:

```

1 app.provideEnvironment(
2   ZEnvironment.empty
3     .add(
4       Map(
5         "foo1" -> new Foo { def bar = 42 },
6         "foo2" -> new Foo { def bar = 5 }
7       )
8     )
9 )

```

In this example, we defined two services of the same type `Foo` with keys `foo1` and `foo2`. We then accessed these services using the `ZIO.serviceAt` method with the corresponding keys.

## 18.3 Handling Errors in Layer Construction

Like the `ZIO` effect, we can also handle failures in the `ZLayer`. Failure in a `ZLayer` can occur when we cannot initialize a service for some reason, e.g., the remote service is unavailable, there is a connection timeout, etc. `ZLayer` has various methods to handle failures like what we have in `ZIO` effects such as `catchAll`, `foldLayer`, `orElse`, `orDie`, and `mapError`.

If, for any reason, the service is not available, we can retry the service initialization by using the `ZLayer#retry` method. The `ZLayer#retry` method will retry the layer construc-

tion until it succeeds or the specified number of retries is reached. The `ZLayer#retry` method takes a `Schedule` that defines the retry strategy:

```
1 val defaultLayer =
2   RemoteDatabase.layer.retry(
3     Schedule.fibonacci(1.second) && Schedule.recur(5)
4   )
```

If the service is unavailable even after five retries, the `ZLayer#retry` method will fail with the last error that occurred during the retries. Now, we can specify fallback dependencies that will be used when the primary dependencies are not available. To achieve this, we can use the `ZLayer#orElse` method to provide a fallback layer:

```
1 val layer = defaultLayer.orElse(fallbackLayer)
```

In the following example, we have the `Database` service required by our application. We have two implementations of the `Database` service: `LocalDatabase` and `RemoteDatabase`. The `RemoteDatabase` is the primary implementation, which tries to connect to the remote database and creates a corresponding layer, but it times out after three seconds. So, it will fail with a timeout error:

```
1 import zio._
2
3 trait Database
4 trait RemoteDatabase extends Database
5
6 object RemoteDatabase {
7   val layer: ZLayer[Any, String, RemoteDatabase] =
8     ZLayer.fromZIO {
9       ZIO
10      .log("Connection to the remote database") *>
11      ZIO
12      .succeed(new RemoteDatabase {})
13      .delay(10.seconds)
14      .timeoutFail(
15        "Timeout: failed to connect to the remote database"
16      )(3.seconds)
17    }
18 }
```

The `Local` is the fallback implementation. If the `LocalDatabase` is not available, we want to use the `RemoteDatabase` as a fallback:

```
1 trait LocalDatabase extends Database
2
3 object LocalDatabase {
4   val layer: ULayer[LocalDatabase] =
5     ZLayer.fromZIO(
```

```

6     ZIO.succeed(new LocalDatabase {}) <* ZIO.log(
7       "Connected to the local database"
8     )
9   )
10 }

```

To do this, we can use the `ZLayer#orElse` method to provide a fallback layer:

```

1 object FallbackLayerExample extends ZIOAppDefault {
2   def run = {
3     for {
4       db <- ZIO.service[Database]
5       _ <- ZIO.log(s"Database service is obtained from the
6         environment $db")
7     } yield ()
8   }.provide(
9     RemoteDatabase.layer
10    .retry(
11      Schedule.fibonacci(1.second) && Schedule.recur(5)
12    )
13    .orElse(LocalDatabase.layer),
14  )
15 }

```

In this example, we provide the `RemoteDatabase` layer with a retry strategy that retries the layer construction five times with a Fibonacci backoff schedule. If the `RemoteDatabase` layer construction fails after five retries, the `LocalDatabase` layer will be used as a fallback.

## 18.4 Memoization of Dependencies

While layers are great for managing dependencies, constructing a layer can be expensive, especially if the construction involves I/O operations, complex resource allocation, significant computation, etc. Memoization can help improve performance by caching the result of the layer construction and reusing it when the same layer is requested again.

`ZIO`, by default, memoizes the layers. For example, if we have two layers, `B` and `C`, that require the same service, `A`:

```

1 trait A
2 trait B
3 trait C
4
5 val a: ZLayer[Any, Nothing, A] =
6   ZLayer.fromZIO(
7     ZIO.succeed(new A {}) <* ZIO.debug("initialized the A service
8     ")

```

```

8   )
9
10 val b: ZLayer[A, Nothing, B] =
11   ZLayer.fromFunction(_: A) => new B {}
12 val c: ZLayer[A, Nothing, C] =
13   ZLayer.fromFunction(_: A) => new C {}

```

By providing A to both layers, ZIO will memoize the A layer and provide the same instance of A to both B and C layers:

```

1 object LayerMemoization extends ZIOAppDefault {
2   val app: ZIO[C with B, Nothing, Unit] =
3     for {
4       b <- ZIO.service[B]
5       c <- ZIO.service[C]
6     } yield ()
7
8   def run = app.provide(a, b, c)
9 }

```

If we run the example, we will see that the A service is initialized only once.

Sometimes, this default behavior is not desirable, and we want to acquire a fresh layer each time it is requested to avoid memoization:

```

1 object LayerFresh extends ZIOAppDefault {
2   val app: ZIO[C with B, Nothing, Unit] =
3     for {
4       b <- ZIO.service[B]
5       c <- ZIO.service[C]
6     } yield ()
7
8   def run = app.provide(a.fresh, b, c)
9 }

```

Another consideration is the need to provide a different instance of a service to a specific part of the application. So, while the whole application is provided with a global instance of a service, by using the `ZIO#provide*` methods at the end of the world, we can provide a specific instance of a service to some local part of the application. When doing this, it will replace the global instance of that service for the entire application:

```

1 val foo = ZLayer.succeed(42)
2 val bar = ZLayer.succeed(5)
3
4 {
5   for {
6     i <- ZIO.service[Int]
7     _ <- ZIO.debug(s"$i resolved to $i in the global scope")
8   }
9 }

```

```

8   - <- {
9     for {
10       i <- ZIO.service[Int]
11       _ <- ZIO.debug(s"i resolved to $i in the inner scope")
12     } yield ()
13   }.provide(bar)
14   d <- ZIO.service[Int]
15   _ <- ZIO.debug(s"d resolved to $d in the global scope")
16 } yield ()
17 }.provide(foo)

```

In this example, we have a service `Int` that is provided by the `foo` layer. We provide the `bar` layer to the inner scope of the application. So, the `Int` service will be resolved to the value provided by the `bar` layer in the inner scope and the value provided by the `foo` layer in the outer scope.

Now that you understand how to provide dependencies locally, let's explore how to memoize a layer manually. For this purpose, the `ZLayer#memoize` method is the choice:

```

1 trait ZLayer[-RIn, +E, +ROut] {
2   def memoize: ZIO[Scope, Nothing, ZLayer[RIn, E, ROut]] = ????
3 }

```

It will return a new layer that memoizes the result of the layer construction. All the resources associated with this layer will be released when the scope is closed.

Assume we have a `foo` layer that takes three seconds to initialize:

```

1 val foo: ZLayer[Scope, Nothing, Int] =
2   ZLayer.fromZIO(
3     ZIO.acquireRelease(
4       ZIO.debug("acquiring") *> ZIO.sleep(3.seconds) *>
5         ZIO.succeed(42)
6       )(_ => ZIO.debug("releasing"))
7     )

```

The `foo.memoize` method will return an effect that contains the lazily computed result of this layer, so it will immediately return without waiting for the layer to be initialized. In this case, the layer will be initialized only when requested for the first time, `ZIO.service[Int]`. After that, any subsequent request for the layer will return the same instance:

```

1 for {
2   layer <- foo.memoize
3   _ <- ZIO.debug("foo layer memoized")
4   _ <- ZIO.scoped {
5     for {
6       _ <- ZIO.debug("Start of scoped block")
7       _ <- ZIO.debug("Acquiring service for the first time takes
8         3 seconds")

```

```

8   _ <- ZIO.service[Int].debug.provideSome[Scope](layer)
9   _ <- ZIO.debug("Acquiring service for the second time
10  should be instant")
11  _ <- ZIO.service[Int].debug.provideSome[Scope](layer)
12  _ <- ZIO.debug("End of scoped block")
13  _ <- ZIO.debug("After closing scope, the layer's finalizer
14  will be called")
15 } yield ()
}
} yield ()

```

An exciting aspect of this example is the lifetime of the layer. It will only be released when the scope is closed, allowing us to extend the layer's lifetime beyond the region in which it is provided. So, the layer will be finalized at the end of the scope, where we closed the scope with `ZIO.scoped`.

Another important point to note is that the `foo` layer requires a `Scope`. Therefore, when we attempt to provide the `foo` layer to the `ZIO.service[Int]` effect, the `ZIO#provide` method will raise a compile-time error indicating that the `Scope` is not provided. Since we don't want to close the scope at this point, we can use the `ZIO#provideSome` method, which allows us to provide only a portion of the environment to the effect. The `ZIO#provideSome` method requires the remaining type of environment that we are not providing. In this case, the only remaining part of the environment is the `Scope`, so we use `ZIO#provideSome[Scope](foo)` to supply the `foo` layer to the `ZIO.service[Int]` effect.

## 18.5 Automatic ZLayer Derivation

In the previous chapter, we learned how to construct a layer from a case class, where each field of the case class represents a dependency of the layer. ZIO provides macros to simplify this process, automatically detecting constructors, including ZIO data structures like `Ref`, `ScopedRef`, `FiberRef`, `Queue`, and others, to construct a layer for us.

For example, consider the following case class `FooService`:

```

1 import zio._
2
3 trait ServiceA
4 case class FooService(ref: Ref[Int], a: ServiceA, b: String)

```

We can create a layer from `FooService` manually as follows:

```

1 val manualLayer: ZLayer[Ref[Int] with ServiceA with String,
2   Nothing, FooService] =
3   ZLayer.fromZIO {
4     for {
      ref <- ZIO.service[Ref[Int]]

```

```

5     a    <- ZIO.service[ServiceA]
6     b    <- ZIO.service[String]
7   } yield FooService(ref, a, b)
8 }
```

Using the `ZLayer.derive` macro, we can simplify the process:

```

1 object FooService {
2   val layer: ZLayer[Ref[Int] with ServiceA with String, Nothing,
3                      FooService] =
4     ZLayer.derive[FooService]
```

To run the following example, we need to provide the `FooService` layer along with all its dependencies:

```

1 object AutomaticLayerConstruction extends ZIOAppDefault {
2   def run = {
3     for {
4       foo <- ZIO.service[FooService]
5         _    <- ZIO.debug(foo.b)
6     } yield ()
7   }.provide(
8     FooService.layer,
9     ZLayer.fromZIO(Ref.make(42)),
10    ZLayer.succeed(5),
11    ZLayer.succeed(new ServiceA {})
12  )
13 }
```

By adding an implicit object of type `ZLayer.Derive.Default`, we can define default values for any dependencies that may not be provided:

```

1 object FooService {
2   implicit val defaultRef =
3     ZLayer.Derive.Default.fromZIO(ZIO.service[Int].flatMap(Ref.
4                                   make(_)))
5
6   val layer =
7     ZLayer.derive[FooService]
```

Now, instead of providing the `Ref[Int]` layer, we can simply provide an `Int` layer:

```

1 object Main extends ZIOAppDefault {
2   def run = {
3     for {
4       foo <- ZIO.service[FooService]
```

```

5     _ <- foo.ref.get.debug
6   } yield ()
7 }.provide(
8   FooService.layer,
9   // ZLayer.fromZIO(Ref.make(42)),
10  ZLayer.succeed(42),
11  ZLayer.succeed("Hello!"),
12  ZLayer.succeed(new ServiceA {})
13 )
14 }

```

The `ZLayer.derive` macro is also capable of deriving resourceful layers. If we need to include initialization and finalization logic when acquiring and releasing the layer, we can use the `AcquireRelease` trait:

```

1 case class FooService(ref: Ref[Int], a: ServiceA, b: String)
2   extends ZLayer.Derive.AcquireRelease[Any, Nothing, FooService]
3   ]
4   override def acquire: ZIO[Any, Nothing, FooService] =
5     ZIO.debug("Acquiring FooService").as(new FooService(ref, a, b))
6
7   override def release(resource: FooService): ZIO[Any, Nothing,
8   Any] =
9     ZIO.debug("Releasing FooService")

```

Alternatively, we can define a scoped layer where the lifetime of the layer is managed by the `Scope`:

```

1 case class FooService(ref: Ref[Int], a: ServiceA, b: String)
2   extends ZLayer.Derive.Scoped[Any, Nothing] {
3   override def scoped(implicit trace: Trace): ZIO[Any & Scope,
4   Nothing, Any] =
5     ZIO.debug("Doing some background work")
6       .schedule(Schedule.fixed(1.seconds))
7       .forkScoped
8   }

```

## 18.6 Conclusion

In this chapter, we have delved into the advanced features of dependency injection using ZIO, focusing on the powerful capabilities of the `ZEnvironment` and `ZLayer` constructs. We explored how to manage multiple services of the same type, handle errors during layer construction, and implement memoization to optimize performance. Furthermore, the

automatic derivation of layers simplifies the creation of complex dependencies, reducing boilerplate code and enhancing maintainability.

As you continue to build upon these concepts, you will be better equipped to design scalable, maintainable applications that effectively leverage ZIO's capabilities. Embracing these advanced techniques will lead to cleaner code, better resource management, and, ultimately, a more robust application architecture.

In the next chapter, we want to explore a new topic about concurrent programming called software transactional memory (STM), which is a powerful building block for writing complex composable concurrent programs. Stay tuned!

## Chapter 19

# Dependency Injection: Contextual Data Types

Context is a piece of data or state that needs to be passed through different parts of an application during its execution. ZIO at its core is a functional effect system that allows you to model your application that carries context data through different parts of the application.

Here are some of the common types of context data that you might encounter in your application:

- **Regional Settings:** Data that is specific to a particular region or scope, such as parallelism settings, logging levels, and other configuration data that can vary between different parts of the application.
- **Request-scoped Context:** When we are developing a service that handles multiple requests concurrently, we often need to manage request-specific context data. This data is unique to each request and must be accessible throughout the request lifecycle. Examples of request-specific context data include user sessions, request IDs, security tokens, and other metadata that need to be propagated across different parts of the application.
- **Operational Context:** Contextual data that is required for cross-cutting operational concerns, such as logging, monitoring, and tracing.
- **Transactional Context:** Data or service that is required to be shared across multiple operations that are part of a single transaction, e.g., database connection.

Context propagation is the process of transferring this information across different parts of an application. While commonly used in request-scoped contexts, it has broader applications. Understanding how to properly manage and propagate context throughout your application is essential to have a well-structured and maintainable codebase.

This chapter explores essential patterns for managing context in ZIO applications. We'll examine some of the practical use cases that demonstrate key context management techniques, providing you with foundational principles; once you get the idea, you can apply

the same principles to anywhere you need to manage context data.

## 19.1 Problem

While context propagation is categorized into different categories, the context data share common characteristic: They need to be shared across different parts of the application. The simplest way to share context data is to pass it explicitly as a parameter to each function/component that needs it. However, this approach has several drawbacks:

- **Boilerplate Code:** Passing context data explicitly can lead to a lot of boilerplate code, especially in large applications with many components.
- **Inflexibility:** It makes the code less flexible and harder to maintain. If you need to add a new piece of context data, you have to update all the functions that use it.
- **Coupling:** It creates tight coupling between components, making it harder to test and refactor the code. So, it makes it harder to change the context data without affecting the entire application.
- **Lack of Maintainability:** It makes it harder to maintain the application, especially as it grows in size and complexity.

This problem becomes more challenging when the number of context data grows, so we need a more structured approach to manage context data in our application.

Let's begin with a fundamental operation in parallel processing. Consider this implementation of the `foreachParN` operator, which enables parallel execution with a maximum number of concurrent operations:

```

1 import zio._

2

3 def foreachParNDiscard[R, E, A, B](
4     items: Iterable[A],
5     n: Int
6 )(f: A => ZIO[R, E, B]): ZIO[R, E, Unit] = {
7     def worker(queue: Queue[A]): ZIO[R, E, Unit] =
8         queue.poll.flatMap {
9             case Some(a) => f(a) *> worker(queue)
10            case None      => Exit.unit
11        }
12
13    for {
14        -           <- ZIO.debug(s"parallelism factor ${n.min(items.size)}")
15        queue   <- Queue.bounded[A](items.size)
16        -           <- queue.offerAll(items)
17        fiber   <- ZIO.forkAllDiscard(
18            ZIO.replicate(n.min(items.size))(worker(queue))
19            )
20        -           <- fiber.join
21    } ZIO.succeed(())
22 }
```

```

21 } yield ()
22 }
```

The `foreachParNDiscard` operator accepts two parameters: a collection of items to process and a parallelism factor that determines the maximum number of concurrent operations. However, constantly specifying the parallelism factor with each call is cumbersome and violates the DRY (Don't Repeat Yourself) principle. Ideally, we want to configure this value once and reuse it across multiple operations. This scenario calls for a contextual data configuration that can be accessed by all parallel operations within our application.

In ZIO, we have two great approaches to managing context data: ZIO environment and `FiberRef`. Both methods are powerful and flexible, offering different trade-offs depending on your application's requirements. Let's start by exploring one of the most common use cases for managing context data: regional settings.

## 19.2 Regional Settings

Regional settings represent configuration data that can vary between different parts of your application. These settings often control operational aspects like parallelism levels, logging verbosity, timeouts, or feature flags that need to be accessible across multiple components but may change based on the execution context.

Managing regional settings effectively requires a system that supports:

1. Default values that can be used as fallbacks
2. The ability to override settings for specific regions or scopes of code
3. Automatic cleanup and restoration of previous values when leaving a scope

Without further ado, let's start by modeling regional settings using the ZIO environment and then explore how we can achieve the same using `FiberRef`.

### 19.2.1 Using ZIO Environment

Until now, we have seen the `R` parameter of `ZIO[R, E, A]` as a way to encode dependencies of a functional effect, for example, the `ZIO[DocService, Throwable, Doc]` is an effect that requires a `DocService` to run and return a `Doc` instance on success. However, we can interpret it from a different perspective: `R` is a way to express the context in which the effect is running. For example, we can think of `ZIO[Parallelism, Throwable, Unit]` as an effect running in a context that has a `Parallelism` configuration setting, or `ZIO[DbConnection, IOException, String]` as an effect running in

One way to model this contextual data is to use the ZIO environment. Instead of passing the parallelism factor as a parameter, we can store it in the environment and access it when needed. Let's write the `foreachParDiscard` operator to use the ZIO environment:

```

1 case class Parallelism(n: Int)
2
3 object Parallelism {
```

```

4 def live(n: Int): ULayer[Parallelism] =
5   ZLayer.succeed(Parallelism(n))
6 }
7
8 def foreachParDiscard[R, E, A, B](
9   items: Iterable[A]
10 )(f: A => ZIO[R, E, B]): ZIO[R with Parallelism, E, Unit] = {
11   for {
12     n <- ZIO.serviceWith[Parallelism](_.n)
13     _ <- foreachParNDiscard(items, n)(f)
14   } yield ()
15 }
```

The `foreachParDiscard` operator has been redesigned to obtain its configuration from the environment rather than requiring it as an explicit parameter. Looking at its updated signature, you'll notice it now requires an environment that includes the parallelism configuration.

This architectural change provides significant benefits. Instead of determining parallelism factors upfront in our code, we can focus on implementing the core logic first. The required configuration can be injected into the environment later at the end of the day. This separation of concerns follows dependency injection principles, effectively decoupling configuration management from application logic. As a result, we can modify performance characteristics without touching the business logic code, making our system more maintainable and flexible:

```

1 import java.io.IOException
2
3 def processInput(input: Int): IO[IOException, Unit] =
4   Console.printLine(s"Processing input: $input")
5
6 object MainApp extends ZIOAppDefault {
7   def run =
8     foreachParDiscard(1 to 10)(processInput)
9       .provide(Parallelism.live(4))
10 }
```

You can also use the `bootstrap` layer to provide the configuration at the entry point of your application:

```

1 abstract class EnvApp[R: EnvironmentTag] extends ZIOApp {
2   override type Environment = R
3   override def environmentTag = EnvironmentTag[R]
4 }
5
6 object BootstrappedConfig extends EnvApp[Parallelism] {
7   override val bootstrap =
8     ZLayer.succeed(Parallelism(4))
```

```

9
10   def run = foreachParDiscard(1 to 10)(processInput)
11 }
```

Now, let's try another interesting aspect of environment-based context: regional scoped contexts. Regional scoped contexts are contexts that are dynamic and can vary between different lexical scopes. To understand this, let's view `ZLayer` from a different perspective. We can think of `ZLayer` as eliminator of environmental effects. This means that you can think of `ZLayer[Any, Nothing, B]` as a function that takes an effect of type `ZIO[B with R, E, A]` and return another effect that the `B` is eliminated from its environment.

To make our DSL more expressive, let's define a `withParallelism` operator as an alias to `Parallelism.live` layer:

```

1 object AppConfig {
2   def live(p: Int): ULayer[Parallelism] =
3     ZLayer.succeed(Parallelism(p))
4
5   def withParallelism(n: Int): ULayer[Parallelism] =
6     live(n)
7 }
```

We are ready to configure a region of code with a specific parallelism factor:

```

1 for {
2   _ <- AppConfig.withParallelism(4) {
3     for {
4       _ <- foreachParDiscard(1 to 20)(processInput)
5       _ <- AppConfig.withParallelism(8) {
6         foreachParDiscard(1 to 10)(processInput)
7       }
8       _ <- foreachParDiscard(1 to 20)(processInput)
9     } yield ()
10   }
11 } yield ()
```

Effects within the `withParallelism` operator's scope (defined by its curly braces) can access the `Parallelism` value from the environment. This value is provided by the `withParallelism` operator itself. The scope is regional - once execution moves outside the `withParallelism` block, the environment layer and its values are no longer accessible.

This design follows key structured programming principles that govern how scopes interact with contextual values:

1. Inner scopes can access contextual values from their enclosing outer scopes.
2. Inner scopes have the ability to override contextual values inherited from outer scopes.

3. When an inner scope closes, the outer scope's contextual values are automatically restored.

This powerful and flexible approach allows you to define and manage contextual data at different levels of granularity, from global to regional settings. The design leverages the ZIO environment and layers to create a robust and maintainable approach to propagate context that adapts to your application's needs.

While this approach effectively manages contextually scoped data without requiring explicit parameter passing, it does have a limitation: the type of all contextual data must be declared through the environment channel. Though this isn't problematic in most cases and in some cases it is the best fit, it can become cumbersome when managing large amounts of contextual data. In such scenarios, you may prefer using an implicit context without type constraints - a topic we'll explore in the next section.

### 19.2.2 Using FiberRef

The `FiberRef` construct offers another powerful way to manage and propagate contextual data. This fiber-local reference allows you to store and retrieve data within a specific fiber's context, making it particularly valuable for managing fiber-specific contextual data.

Let's explore how `FiberRef` can help propagate contextual data by reimplementing our `AppConfig` as follows:

```

1 object AppConfig {
2   private val parallelism: FiberRef[Int] =
3     Unsafe.unsafe { implicit unsafe =>
4       FiberRef.unsafe.make(4)
5     }
6
7   // Temporarily changes the parallelism level for a specific ZIO
8   // effect
9   def withParallelism[R, E, A](n: => Int)(zio: ZIO[R, E, A]) =
10    parallelism.locally(n)(zio)
11
12  // Executes a function that takes the current parallelism level
13  // as input
14  def parallelismWith[R, E, A](f: Int => ZIO[R, E, A]): ZIO[R, E,
A] =
15    parallelism.getWith(f)
16 }
```

It has two primary methods: `withParallelism` and `parallelismWith`. These methods provide controlled access to modify and use the parallelism level:

- `withParallelism`: Temporarily changes the parallelism level for a specific given ZIO effect. It will execute the given effect with the specified parallelism level and finally restore the original parallelism after completion.

- **parallelismWith**: Executes a function that takes the current parallelism level as input. It gets the current parallelism level associated with the current fiber and uses it to execute the given function that takes the parallelism level as input.

Now that we have written our primary methods, we are ready to rewrite the `foreachParDiscard` operator to use the `AppConfig`:

```

1 def foreachParDiscard[R, E, A, B](
2   items: Iterable[A]
3 )(f: A => ZIO[R, E, B]): ZIO[R, E, Unit] =
4   AppConfig.parallelismWith(foreachParNDiscard(items, _)(f))

```

That's it! We have successfully rewritten the new API using the `FiberRef` construct. The `foreachParDiscard` has no explicit dependency on the `Parallelism` setting, but implicitly. Let's see how we can use this new API:

```

1 for {
2   _ <- foreachParDiscard(1 until 10)(processInput) // Running
3     with default value for parallelism
4   _ <- AppConfig.withParallelism(5) {
5     for {
6       _ <- foreachParDiscard(10 until 20)(processInput) // Running
7         with parallelism of 5
8       _ <- AppConfig.withParallelism(4) {
9         foreachParDiscard(20 until 30)(processInput) // Running
10        with parallelism of 4
11      }
12    } yield ()
13  } yield ()

```

Similar to the previous approach:

1. Inner scopes can access contextual values from their enclosing outer scopes.
2. Inner scopes have the ability to override contextual values inherited from outer scopes.
3. When an inner scope closes, the outer scope's contextual values are automatically restored.

Beside the fact that with the `FiberRef` construct we can manage the contextual data without the need to declare its type in the environment, there are two another difference between the `FiberRef` and the `ZLayer` construct: 1. With `FiberRef` we can have isolated configuration data per fiber, while with `ZLayer` we have a global configuration data that is shared across all fibers. This can be useful when you want to have a configuration data that is specific to a fiber. 2. With `FiberRef` we can have a default value for the contextual data, while with the `ZIO` environment we can't have a default value for contextual data.

## 19.3 Transactional Effects Using ZIO Environment

ZIO's environment offers an elegant approach to modeling transactional effects. So whenever we have some effect of type `ZIO[R with Transaction, E, A]` we can think of it as an effect that participates in a transaction. We can think of `Transaction` in environment of an effect as a marker that indicates that the effect is running in context of a transaction. We can compose multiple transactional effects together and finally commit them as a single atomic unit.

The beauty of this design lies in its simplicity. Both transactional and non-transactional effects can be modeled using the same ZIO type - their only distinction is the presence of `Transaction` in their environment type parameter. This unified model means you work with a single, consistent abstraction throughout your codebase.

Here's the key insight: The `Transaction` type in your environment marks the boundaries of your transactional context. When it's time to execute, we eliminate this marker from the effect through `run` method:

```

1 trait TransactionManager[Transaction] {
2   def run[R, E, A](zio: ZIO[R with Transaction, E, A]): ZIO[R, E,
3     A]
}
```

This is where the magic happens. The `run` method takes your transactional effect and executes everything atomically, transforming `ZIO[R with Transaction, E, A]` into `ZIO[R, E, A]`. So it is responsible for managing any errors that might occur during the transaction and rolling back the transaction if necessary.

Here's how you might use the `TransactionManager` in practice:

```

1 val trx1: ZIO[Transaction, Throwable, Unit] = ???
2 val trx2: ZIO[Transaction, Throwable, Int] = ???
3
4 // Compose transactional effects
5 val trx: ZIO[Transaction, Throwable, Int] =
6   trx1 *> trx2
7
8 val tm: TransactionManager[Transaction] = ???
9
10 // Execute the transaction
11 val result: ZIO[Any, Throwable, Int] =
12   tm.run(trx)
```

If in the middle of executing the transaction it encounters an error, the entire transaction will be rolled back. This ensures that your application remains in a consistent state, even in the face of failure.

An example use-case of this approach is modeling a JDBC transaction in a ZIO environment. In this case we use `java.sql.Connection` as the `Transaction` type parameter

to express that this effect is running in a transactional context of a JDBC connection:

```

1 import zio._
2 import java.sql.{SQLException, Connection}
3
4 trait JdbcTransactionManager extends TransactionManager[Connection]{
5   val connection: ZLayer[Any, SQLException, Connection]
6
7   def run[R, E, A](zio: ZIO[R with Connection, E, A]): ZIO[R, E, A] =
8     ZIO
9       .acquireReleaseWith(
10         ZIO.service[Connection]
11       ) { conn =>
12         ZIO.succeed(conn.close())
13       } { conn =>
14         zio
15           .tap(_ => ZIO.attempt(conn.commit()).orDie)
16           .tapError(_ => ZIO.attempt(conn.rollback()).orDie)
17       }
18       .provideSomeLayer[R](connection.orDie)
19 }
```

The run method uses the acquireReleaseWith operator to manage the lifecycle of the Connection. It acquires the Connection resource, executes the transaction, and finally releases the Connection resource. If the transactional effect succeeds, it commits the transaction; otherwise, it rolls back the transaction. The connection layer is an abstract member that should be implemented by the concrete implementation of the JdbcTransactionManager, e.g. SqliteTransactionManager:

```

1 import zio._
2 import java.sql.{DriverManager, SQLException, Connection}
3
4 case class SqliteTransactionManager(dbPath: String) extends JdbcTransactionManager {
5   val connection: ZLayer[Any, SQLException, Connection] =
6     ZLayer.fromZIO {
7       ZIO.attempt {
8         val conn = DriverManager.getConnection(s"jdbc:sqlite:$dbPath")
9         conn.setAutoCommit(false)
10        conn
11      }.refineToOrDie[SQLException]
12    }
13 }
```

Now, we are ready to define the `Ledger` service that interacts with the database to perform ledger operations, such as creating a new account, depositing, and withdrawing funds:

```

1 trait Ledger {
2   def balance(accountId: String): ZIO[Connection, SQLException,
3             BigDecimal]
4   def newAccount(accountId: String, balance: BigDecimal): ZIO[
5     Connection, SQLException, Unit]
6   def deposit(accountId: String, amount: BigDecimal): ZIO[
7     Connection, SQLException, Unit]
8   def withdraw(accountId: String, amount: BigDecimal): ZIO[
9     Connection, SQLException, Unit]
10 }
```

We have to write a JDBC version of such a ledger, like this:

```

1 class JdbcLedger extends Ledger {
2   private val selectBalanceSQL = "SELECT balance FROM ledger
3     WHERE id = ?"
4   private val updateBalanceSQL = "UPDATE ledger SET balance =
5     balance + ? WHERE id = ?"
6   private val insertAccountSQL = "INSERT INTO ledger (id, balance
7     ) VALUES (?, ?)"
8
9   override def newAccount(accountId: String, balance: BigDecimal):
10     ZIO[Connection, SQLException, Unit] = ???
11
12   override def balance(accountId: String): ZIO[Connection,
13     SQLException, BigDecimal] = ???
14
15   override def deposit(accountId: String, amount: BigDecimal):
16     ZIO[Connection, SQLException, Unit] =
17     ZIO.serviceWithZIO[Connection] { conn =>
18       ZIO.attempt {
19         val stmt = conn.prepareStatement(updateBalanceSQL)
20         stmt.setBigDecimal(1, amount.bigDecimal)
21         stmt.setString(2, accountId)
22         val updated = stmt.executeUpdate()
23         if (updated == 0) {
24           throw new SQLException(s"Account $accountId not found")
25         }
26       }.refineToOrDie[SQLException]
27     }
28
29   override def withdraw(accountId: String, amount: BigDecimal):
30     ZIO[Connection, SQLException, Unit] =
31     deposit(accountId, -amount)
32 }
```

25 | }

Now, it's time to put all the pieces together:

```

1 object MainApp extends ZIOAppDefault {
2   val tm = SqliteTransactionManager("ledger.db")
3
4   def run =
5     for {
6       - <- tm.run {
7         {
8           for {
9             ledger <- ZIO.service[Ledger]
10            - <- ledger.newAccount("sender", 100L)
11            - <- ledger.newAccount("receiver", 0L)
12          } yield ()
13        }.provideSomeLayer(JdbcLedger.live)
14      }
15      - <- tm.run {
16        {
17          for {
18            ledger <- ZIO.service[Ledger]
19            - <- ledger.withdraw("sender", 100L)
20            - <- ledger.deposit("receiver", 100L) *>
21              ZIO.fail(new SQLException("Simulated
22 error"))
23            } yield ()
24          }.provideSomeLayer(JdbcLedger.live)
25        }
26      } yield ()
}

```

In this example, we used `SqliteTransactionManager` that manages transactions using a SQLite database. We first created two accounts, `sender` and `receiver`, with initial balances of 100 and 0, respectively. After committing this transaction, we attempted to run another transaction that transfers 100 from the `sender` account to the `receiver` account. However, we simulated an error during the deposit operation by failing the effect with a `SQLException`. This caused the entire transaction to be rolled back, ensuring that the funds were restored to their original state.

This was a simple example of how you can model transactional effects using ZIO environment. But we are not limited to using JDBC transactions, we can write any implementation not only for databases but also for other distributed transactional systems. Consider a microservices architecture where we need to manage transactions across multiple services. We can define a `SagaTransaction` type that implements the `Transaction` interface to handle these distributed workflows.

In a saga pattern, we work with a sequence of service calls, each paired with a compen-

sating action that can undo its effects if needed. Each action-compensation pair can be modeled as a transactional effect using `ZIO[R with SagaTransaction, E, A]`. This type represents an effect that participates in a saga transaction, encapsulating both the primary action and its compensating action.

By composing these transactional effects, we can create a single, cohesive unit of work that maintains atomic properties across distributed services. The `SagaTransactionManager` orchestrates this process by:

- Executing the transaction steps in sequence
- Error handling and managing retries
- Coordinating rollbacks through compensating actions in case of failures

This approach provides a type-safe and composable way to handle distributed transactions while maintaining the reliability guarantees needed in microservice architectures.

## 19.4 Context and ZIO API

ZIO embraces the concept of context data and provides a rich set of APIs to manage and propagate context data across different application parts. I'm not going to cover all of them in detail, but it is good to have an overview of them:

1. The `Scope` data type is used to manage the lifecycle of resources that are acquired and released within a specific scope. It provides a way to acquire resources at the beginning of the scope and release them at the end of the scope. Similar to what we have seen in transactional effects, we have the `ZIO.scoped` method, which eliminates the `Scope` type from the environment of the effect, indicating that there are no longer any resources used by this effect that require a scope:

```

1 object ZIO {
2   def scoped[R, E, A](zio: ZIO[Scope with R, E, A]): ZIO[R,
3     E, A] = ???
```

2. ZIO provides operators to customize its default service implementations, including `Clock`, `Console`, `System`, `Random`, and `ConfigProvider`. These services can be overridden using their corresponding operators: `ZIO.withClock`, `ZIO.withConsole`, `ZIO.withSystem`, `ZIO.withRandom`, and `ZIO.withConfigProvider`.

ZIO maintains instances of these services using `FiberRef`. When you call one of these operators, the new service instance replaces the default one, but only within the scope of the executing fiber. This fiber-local customization ensures that service modifications remain isolated and don't affect other parts of your application.

3. Another use case of `FiberRef` in ZIO API is when we want to add some contextual data to the fiber running the effect, such as metrics and monitoring data, log annotations, log spans and so on. For example, using `ZIO.tagged` operator to tag a metric, or using `ZIO.logAnnotate` to add annotations to the logger, or using ZIO

`#logSpan` to adjust a new label for the current logging span. We will cover these in more detail in their respective chapters.

4. ZIO Runtime, uses `FiberRef` to configure runtime settings such as enabling/disabling runtime flags, setting the default and blocking executor, adding and removing logger and supervisor.

## 19.5 Conclusion

Understanding how to effectively manage and propagate contextual data is crucial for building well-structured and maintainable ZIO applications. Throughout this chapter, we explored two powerful approaches to handling context in ZIO: the ZIO environment and `FiberRef`.

The ZIO environment provides a type-safe way to manage context through its effect type system. By encoding contextual requirements in the R type parameter, we can clearly express and track the context requirements of our effects. This approach is particularly valuable when:

- You need strict type safety and compile-time verification of context requirements
- The context is part of your application's core domain model
- You want to leverage ZIO's dependency injection system

`FiberRef` is another flexible approach to managing fiber-local state without explicit type declarations. It excels in scenarios where:

- You want to avoid threading context types through your entire application
- You need to manage a context that varies between different fibers
- The context is primarily operational rather than domain-specific
- The context is not part of your core domain model, and it is a cross-cutting concern
- You need a default value for the contextual data

Both approaches support key structured programming principles:

1. Inner scopes can access context from outer scopes
2. Inner scopes can override inherited context
3. Context is automatically restored when leaving a scope

The choice between ZIO environment and `FiberRef` often depends on your specific needs: - Use the ZIO environment when type safety and explicit context requirements are paramount - Choose `FiberRef` when you need to access cross-cutting context that is not part of your core domain and don't want to clutter your ZIO environment. Also, choose it when it is important to you to have a default value for contextual data.

We also saw how these concepts are applied in real-world scenarios, from managing regional settings to implementing transactional effects. The ZIO API itself uses these patterns extensively, demonstrating their practical value in areas like resource management, service customization, and cross-cutting concerns like logging, diagnostic, and metrics.

By understanding and applying these patterns appropriately, you can create more mod-

ular, maintainable, and robust ZIO applications that effectively manage contextual data throughout their lifecycle.

## 19.6 Exercise

1. Implement a simple transactional key-value store that:
  - Supports basic operations (get, put, delete)
  - Uses ZIO environment to mark transactional effects
  - Provides atomic execution of multiple operations
  - Implements proper rollback on failure

Sample interface:

```
1 trait KVStore[k, V] {  
2     def get[V](key: K): ZIO[Transaction, Throwable, Option[V]]  
3     def put(key: K, value: V): ZIO[Transaction, Throwable,  
4         Unit]  
5     def delete(key: K): ZIO[Transaction, Throwable, Unit]  
6 }
```

2. Implement a distributed transaction manager using the Saga pattern discussed in the chapter by encoding the transaction into the ZIO environment. Each business action must have a corresponding compensating action that can roll back its effects. The transaction manager should compose these action-compensation pairs into a single atomic transaction, ensuring that either all actions are completed successfully or all completed actions are rolled back through their compensating actions.

# Chapter 20

# Configuring ZIO Applications

Configuration management is an essential element of cloud-native applications. As applications grow in complexity and move to cloud environments, the need for flexible and type-safe configuration becomes increasingly important.

This chapter delves into proven strategies for configuration management in ZIO applications, with a particular focus on leveraging ZIO's native configuration capabilities to build reliable and maintainable systems.

## 20.1 Configuration Challenges

Configuration management is a critical aspect of modern application development. Rather than hardcoding values directly into our applications, we need flexible ways to manage various settings that may change across environments or deployments. These configurations can range from basic infrastructure details like server hosts and ports to sensitive credentials such as database connection strings and API keys to runtime behavior controls like feature flags.

While the concept seems straightforward, implementing a robust configuration system presents several challenges. We must ensure type safety to catch errors early, maintain composability to combine multiple configuration sources, handle missing or invalid values appropriately, and keep the system simple enough for developers to use effectively. The configuration layer needs to work seamlessly whether settings come from environment variables, configuration files, remote configuration services, or command-line arguments.

## 20.2 Configuration Methods in Practice

While ZIO offers a robust built-in solution for configuration management, which is well-designed, understanding alternative approaches can be valuable before adopting diving into the ZIO's native approach.

One of the simplest methods for managing configurations is to pass them as function arguments. This approach is straightforward and works effectively for smaller applications:

```

1 import zio._

2

3 object DocumentService {
4     def make: ZIO[Any, Throwable, DocumentService] =
5         for {
6             databaseConfig <- DatabaseConfig.load()
7             database       <- Database.make(databaseConfig)
8
9             loggingConfig <- LoggingConfig.load()
10            logging        <- ZIO.service[Logging]
11
12            storageConfig <- StorageConfig.load()
13            storage        <- Storage.make(storageConfig)
14
15            config <- DocumentServiceConfig.load()
16        } yield new DocumentService(config, logging, database,
17            storage)
18    }
19
20 object MainApp extends ZIOAppDefault {
21     def run =
22         DocumentService.make
23             .flatMap(_.start)
24 }
```

However, passing configurations as function arguments has significant limitations.

First, as applications scale, manually threading configurations through multiple layers becomes tedious and error-prone. Each new service or component adds another configuration to manage, leading to increasingly complex function signatures and dependency chains.

Second, this approach enforces a rigid two-phase initialization pattern: first load all configurations, then construct services with those configurations. This creates repetitive boilerplate code and makes service implementations less flexible.

A more elegant solution leverages ZIO's environment to propagate configurations throughout the application. This approach offers better scalability and enables lazy configuration loading at the wiring stage. Here's how it works:

```

1 import zio._

2

3 object DocumentService {
4     val live: ZLayer[
5         DocumentServiceConfig with Logging with Database with Storage
6     ] =
```

```

6   Throwable,
7   DocumentService
8 ] =
9 ZLayer.fromZIO {
10   for {
11     logging <- ZIO.service[Logging]
12     database <- ZIO.service[Database]
13     storage <- ZIO.service[Storage]
14     config <- ZIO.service[DocumentServiceConfig]
15   } yield new DocumentService(config, logging, database,
16     storage)
17 }
18 }
19 DocumentService.make
20   .flatMap(_.start)
21   .provide(
22     Logging.live,
23     LoggingConfig.live,
24     Database.live,
25     DatabaseConfig.live,
26     Storage.live,
27     StorageConfig.live,
28     DocumentService.live,
29     DocumentServiceConfig.live
30 )

```

With dependency injection, the previous problems are mitigated to some extent, but it still has some drawbacks:

1. **Overuse of ZIO environment:** ZIO's environment pattern, while powerful, can lead to a bloated mix of service and configuration combinations within the environment. This reduces code readability and maintainability, especially as applications grow larger.
2. **Lack of Standardized Configuration Management:** ZIO's flexible nature allows users to implement configuration management in various ways. While this flexibility can be powerful, it often leads to divergent implementation patterns across teams and projects. New developers joining a project face a steeper learning curve as they encounter various custom configuration solutions rather than a single, well-defined pattern. Even though ZIO intentionally avoids being overly prescriptive, the lack of standardized configuration practices can hinder project maintainability.
3. **Limited Multi-Source Configuration Support:** Integrating multiple configuration sources requires significant manual implementation effort. The lack of built-in abstractions for handling diverse configuration sources (such as environment variables, files, and external services) means developers must write and maintain con-

siderable boilerplate code. This can be particularly challenging when dealing with complex configuration hierarchies or when switching between different environments.

Let's explore how ZIO's built-in configuration management can help address these challenges.

ZIO Config has two basic concepts:

1. Config Descriptor
2. Config Provider

Given these two inputs, you can load a configuration of type A with the following `loadConfig` function:

```

1 import zio._

2

3 def loadConfig[A](
4     configDescriptor: Config[A],
5     configProvider: ConfigProvider
6 ): ZIO[Any, Config.Error, A] =
7     ZIO.withConfigProvider(configProvider) {
8         ZIO.config[A](configDescriptor)
9     }

```

The `Config[A]` is a descriptor that describes the structure of the configuration of type A. It can be a simple descriptor of primitive types such as `Int`, `Long`, `Boolean`, etc., or a complex descriptor of product and sum types such as case classes, sealed traits, and enums.

The `ConfigProvider` is a provider that loads the configuration from a specific source, such as environment variables, system properties, files, or external services. ZIO Config provides built-in providers for common configuration sources, and you can also create custom providers for more specialized use cases.

Let's start by defining a simple configuration descriptor for the following typesafe HO-CON configuration file:

```

1 # application.conf
2 host = "localhost"
3 port = 8080

```

The first step is to model the config type for this configuration file. As the configuration is straightforward, we can define it using a case class that contains both `host` and `port` fields:

```

1 import zio._

2

3 case class AppConfig(host: String, port: Int)

4

5 object AppConfig {
6     // Define a configuration descriptor for AppConfig

```

```

7  implicit val config: Config[AppConfig] = {
8    Config.string("host") zip // Read a string value from the "host" key configuration
9      Config.int("port")     // Read an integer value from the "port" key configuration
10   }.map {
11     case (host, port) => AppConfig(host, port)
12   }
13 }
```

Inside the companion object, we defined a configuration descriptor for `AppConfig` using the `Config.string("host")` descriptor describes the `host` key with a value of type `String` from the configuration source, and the `Config.int("port")` descriptor describes the `port` key a value of type `Int`. The `zip` operator combines the two descriptors into a single descriptor of a tuple (`String`, `Int`), and the `map` function transforms the tuple (`host`, `port`) into an instance of `AppConfig`.

The next step is to define a config provider that loads the configuration from the typesafe HOCON file. ZIO Core has some built-in config providers for common configuration sources, such as environment variables, system properties, and console inputs. For more specialized out-of-the-box providers, you can use the ZIO Config<sup>1</sup> project that provides additional providers for typesafe HOCON, YAML, and XML files:

```

1 libraryDependencies += "dev.zio" %% "zio-config-typesafe" % "4.0.2"
```

With the `zio-config-typesafe` dependency added, we have all the necessary tools to load the configuration from the typesafe HOCON file:

```

1 import zio._
2 import zio.config.typesafe.TypesafeConfigProvider._
3
4 val config: ZIO[Any, Config.Error, AppConfig] =
5   loadConfig[AppConfig](
6     AppConfig.config,
7     fromHoconFilePath("config.conf")
8   )
```

Now we are ready to run the application with the loaded configuration:

```

1 object MainApp extends ZIOAppDefault {
2   def run =
3     config.flatMap { config =>
4       ZIO.debug(s"Server started with ${config.host}:${config.port}")
5     }
6 }
```

---

<sup>1</sup><https://zio.dev/zio-config>

In this example, we tried to do all things manually and explicitly to understand the underlying concepts. However, there are more refined ways to achieve the same result with less boilerplate code:

First, instead of manually writing the configuration descriptor, we can use automatic derivation with the `zio-config-magnolia` module:

```
1 libraryDependencies += "dev.zio" %% "zio-config-magnolia" % "4.0.2"
```

So then, we can use the `deriveConfig` function to automatically derive the configuration descriptor for the `AppConfig` case class:

```
1 import zio.config.magnolia._
2
3 case class AppConfig(host: String, port: Int)
4
5 object AppConfig {
6   implicit val config: Config[AppConfig] =
7     deriveConfig[AppConfig]
8 }
```

Second, instead of explicitly passing the configuration descriptor, we can use implicit parameters to automatically resolve the proper configuration descriptor:

```
1 def loadConfig[A](
2   configProvider: ConfigProvider
3 )(implicit config: Config[A]): ZIO[Any, Config.Error, A] =
4   ZIO.withConfigProvider(configProvider) {
5     ZIO.config[A]
6 }
```

Third, instead of passing the configuration provider for each configuration load, we update the default configuration provider for the entire application. ZIO by default uses the `ConfigProvider.defaultProvider` for loading configurations storing them in a `FiberRef` of default services, including the configuration provider. So we can update the default configuration provider using the `Runtime.setConfigProvider` to the `bootstrap` layer:

```
1 def loadConfig[A](implicit config: Config[A]): ZIO[Any, Config.Error, A] =
2   ZIO.config[A]
```

With these improvements, the code can become more concise and easier to maintain, and instead of `loadConfig` we can use the `ZIO.config` operator to load the configuration. Here is the final version of the application:

```
1 import zio._
2 import zio.config.magnolia._
```

```

3 import zio.config.typesafe.TypesafeConfigProvider._
4
5 case class AppConfig(host: String, port: Int)
6 object AppConfig {
7   // Automatically derive the configuration descriptor for
8   // AppConfig
9   implicit val config: Config[AppConfig] = deriveConfig[AppConfig]
10 }
11
12 object MainApp extends ZIOAppDefault {
13   // Update the default configuration provider globally
14   override val bootstrap =
15     Runtime.setConfigProvider(fromHoconFilePath("config.conf"))
16
17   def run =
18     for {
19       // Load the configuration using the ZIO.config operator
20       config <- ZIO.config[AppConfig]
21       _      <- ZIO.debug(s"Server started with ${config.host}:${config.port}")
22     } yield ()
23 }
```

Now that you have a basic understanding of how ZIO loads configurations, let's dive deeper into these two core concepts: config descriptors and config providers.

## 20.3 Describing Configurations

In the previous section, we learned how config descriptors are used to have type-safe configuration management. Config descriptors describe the various aspects of a configuration such as what keys are expected, what types their values are, how they are validated, and other essential configuration properties.

Let's start from the beginning and understand the basics of config descriptors.

First, for primitive types, you can use the built-in config descriptors provided by ZIO Config, such as `Config.string`, `Config.int`, `Config.long`, and `Config.boolean`. These descriptors are used to read primitive values from the configuration source. For example, if we have a `host = "localhost"` key-value pair in the configuration, you should read the value of the `host` key as a string using `Config.string`. But how do you define which key to read? This is where the nested configuration descriptors come into play.

Nested configuration descriptors are used to read values from a specific key or path in the configuration source. For example, with `Config.string.nested("host")`, you

can read the value of the host key from the configuration source. So if the configuration source is a typesafe HOCON file, this config can read the following configuration file:

```
1 host = "localhost"
```

Alternatively, you can easily use `Config.string("host")`. If it has more levels of nesting, you can use the `nested` operator to read values from nested keys. For example, to read the following configuration, you can use `Config.string("host").nested("Server")`:

```
1 Server {  
2   host = "localhost"  
3 }
```

You can do the same for more nested levels. For example, to read the following configuration, you can use `Config.string("host").nested("Server").nested("Application")`:

```
1 Application {  
2   Server {  
3     host = "localhost"  
4   }  
5 }
```

To read the optional key, we can call the `Config#optional` operator to make the key optional. For example, if the `cors_origins` key is optional, you can use the `Config.boolean("cors_origins").optional`:

```
1 host = "localhost"  
2 ; cors_origins = true
```

For multiple values, you can use `Config.chunkOf`, `Config.listOf`, `Config.vectorOf`, and `Config.setOf` constructors. For example, to read a set of blocked hosts, you can use `Config.listOf("blocked_hosts", Config.string)`:

```
1 blocked_hosts = ["example.com", "example.org", "example.net"]
```

Sometimes you'll want to provide default values for your configurations. When a configuration value is missing, you can fall back to a default using the `Config#withDefault` operator. For example, to set a default port value of 8080, use

```
1 Config.int("port").withDefault(8080)
```

This ensures your application continues running with sensible defaults even when configuration values are omitted.

You can also validate the configuration values using the `Config#validate` operator:

```
1 Config.int("port").validate("should be between 1 and 65535")(p =>  
  p >= 1 && p <= 65535)
```

This operator takes a message and a predicate function that validates the configuration value. If the predicate returns `false`, the configuration value is considered invalid, and an error message is returned.

This is your first encounter with explicit error messages in the configuration system. ZIO Config handles errors in a type-safe way. When you use `ZIO.config` to load the configuration, it returns a `ZIO[Any, Config.Error, A]` effect which can fail with a `Config.Error`. This sealed trait encompasses various error types that may occur during configuration loading, including validation errors, missing data errors, and unsupported source errors.

You can manually create errors using the `Config.fail` constructor, which takes an error message and returns a `Config.Error`. Additionally, when mapping config descriptors, you can use `mapOrFail` instead of `map` to handle error cases. Here's an example of using `mapOrFail` to convert a string config into a UUID config:

```

1 import java.util.UUID
2 import scala.util.control.NonFatal
3
4 def uuid: Config[UUID] =
5   Config.string.mapOrFail(text =>
6     try Right(UUID.fromString(text))
7     catch {
8       case NonFatal(_) =>
9         Left(Config.Error.InvalidData(Chunk.empty, s"Expected a
10           uuid, but found ${text}"))
11     }
12   )

```

The previous examples covered simple configuration values, but now let's explore more complex configurations. When you need to model product and sum types in your configurations, you can use the `zip` and `orElse` operators respectively.

For example, you can model a `MysqlConfig` as a product type containing `host`, `port`, `username`, and `password` fields. Here's how to create a case class and define its configuration descriptor using the `zip` and `map` operators:

```

1 import zio.Config.Secret
2
3 case class MysqlConfig(
4   host: String,
5   port: Int,
6   username: String,
7   password: Secret
8 )
9
10 object MysqlConfig {
11   implicit val config: Config[MysqlConfig] = {
12     Config.string("host") zip

```

```

13     Config.int("port") zip
14     Config.string("username") zip
15     Config.secret("password")
16   }.map {
17     case (host, port, username, password) =>
18       MysqlConfig(host, port, username, password)
19   }
20 }
```

Please note that we used the `Secret` data type to ensure memory-safe operations and prevent passwords from being exposed in logs unintentionally. The `Secret` class's `equals` method uses a constant-time comparison algorithm instead of standard string comparison and prevents timing attacks where an attacker could deduce the secret's content by measuring response times. The class is `final` to prevent inheritance and potential security bypasses. It also has a `wipe` method to explicitly zero out the memory containing the secret after it no longer needs to be retained in memory.

Let's extend our example to a case where a user has two choices, they can either use `MysqlConfig` or `SqliteConfig` to configure the database:

```

1 import zio._
2 import zio.Config.Secret
3
4 sealed trait DatabaseConfig
5 case class MysqlConfig(
6   host: String,
7   port: Int,
8   username: String,
9   password: Secret
10 ) extends DatabaseConfig
11 case class SqliteConfig(path: String)
12   extends DatabaseConfig
```

You can model this as a sum type using the `orElse` operator:

```

1 object MysqlConfig {
2   implicit val config: Config[MysqlConfig] =
3   {
4     Config.string("host") zip
5     Config.int("port") zip
6     Config.string("username") zip
7     Config.secret("password")
8   }.map { case (host, port, username, password) =>
9     MysqlConfig(host, port, username, password)
10   }
11   .nested("Mysql")
12 }
```

```

14 object SqliteConfig {
15   implicit val config: Config[SqliteConfig] =
16     Config.string("path").map(SqliteConfig(_)).nested("Sqlite")
17 }
18
19 object DatabaseConfig {
20   implicit val config: Config[DatabaseConfig] =
21     MysqlConfig.configorElse SqliteConfig.config
22 }

```

The `DatabaseConfig.config` descriptor can load either `MysqlConfig` or `SqliteConfig` objects from a configuration source. While this approach works, it has a limitation. In fact, the `orElse` operator is a fallback operator; so, it attempts to load `MysqlConfig` first, and only if that fails it tries loading `SqliteConfig`. This non-exclusive loading pattern means both configurations could potentially be valid simultaneously in a configuration source.

In most cases, we don't need stricter constraints to ensure the configuration source contains exactly one valid database configuration. However, we can take extra steps to enforce mutual exclusivity between the two configurations. As a practice, you can write a custom validation that ensures only one case of configuration is present in the configuration source and then try to load it.

## 20.4 Config Providers

Config providers are responsible for loading configurations from various sources such as environment variables, system properties, files, and external services. You can think of a config provider as the following trait:

```

1 trait ConfigProvider {
2   def load[A](config: Config[A]): IO[Config.Error, A]
3 }

```

It takes a config descriptor of type `A` and returns an effect with a configuration value of type `A`. If any error occurs during the parsing and loading of the configuration, it fails with `Config.Error`. You can write your own custom config providers for specialized use cases, but ZIO Core and ZIO Config<sup>2</sup> projects provide various config providers that cover the most common configuration sources.

Similar to the `Config` descriptor, the `ConfigProvider` has a fallback operator called `orElse` that allows you to chain multiple config providers together, so if the first provider fails, it tries the next one. This is useful when you have multiple configuration sources and want to try loading configurations from each source in order. The default config provider of ZIO is the chain of `envProvider` and `propsProvider`:

```

1 lazy val defaultProvider: ConfigProvider =

```

<sup>2</sup><https://zio.dev/zio-config>

```

2   ConfigProvider.envProvider.orElse(
3     ConfigProvider.propsProvider
4   )

```

This means that by default, ZIO will try to load configurations from environment variables first, and if it fails, it will try to load configurations from system properties.

We have two types of config providers in general, nested and flat:

A nested config provider that supports nested keys and multiple values for a single key natively. It is designed to work with complex configuration structures, such as Typesafe HOCON, YAML, JSON, and XML files. The `zio-config-typesafe`, `zio-config-yaml`, and `zio-config-xml` modules in ZIO Config<sup>3</sup> project provide nested providers for reading configurations from these sources.

A flat config provider is a simplified config provider that natively does not support nested keys or multiple values for a single key. It is designed to work with simple key-value pairs, such as environment variables, system properties, and console inputs. However, it can be used with nested keys and multiple values by using delimiters to represent new levels of nesting and to separate multiple values. All built-in providers in ZIO Core are flat providers:

- `ConfigProvider.envProvider`: Loads configurations from environment variables.
- `ConfigProvider.consoleProvider`: Loads configurations from console input.
- `ConfigProvider.propsProvider`: Loads configurations from system properties.
- `ConfigProvider.fromAppArgs`: Loads configurations from command-line arguments.
- `ConfigProvider.fromMap`: Loads configurations from a map of key-value pairs. It is useful for integration with external configuration libraries or for testing purposes.

Let's see how these two types of config providers work in practice.

Assume we have the following nested configuration classes:

```

1 case class DatabaseConfig(
2   url: String,
3   username: String,
4   password: Secret,
5   allowedSchemas: Set[String],
6   poolConfig: ConnectionPoolConfig = ConnectionPoolConfig()
7 )
8
9 object DatabaseConfig {
10   implicit val config: Config[DatabaseConfig] = {
11     {

```

---

<sup>3</sup><https://zio.dev/zio-config>

```

12     Config.string("url") zip
13     Config.string("username") zip
14     Config.secret("password") zip
15     Config
16         .setOf("allowed-schemas", Config.string)
17         .withDefault(Set("public")) zip
18     ConnectionPoolConfig.config
19         .withDefault(ConnectionPoolConfig())
20     }.map { case (url, username, password, allowedSchemas,
21       poolConfig) =>
22       DatabaseConfig(url, username, password, allowedSchemas,
23       poolConfig)
24     }
25   }.nested("Database")
26 }
27
28
29
30
31 )
32
33 object ConnectionPoolConfig {
34   implicit val config: Config[ConnectionPoolConfig] = {
35   {
36     Config.int("maxSize").withDefault(10) zip
37     Config.int("minSize").withDefault(1) zip
38     Config.long("idleTimeout").withDefault(300L) zip
39     Config.long("maxLifetime").withDefault(3600L)
40   }.map {
41     case (maxSize, minSize, idleTimeout, maxLifetime) =>
42       ConnectionPoolConfig(maxSize, minSize, idleTimeout,
43       maxLifetime)
44   }
45 }.nested("Pool")
46 }

```

The corresponding configuration file for Typesafe HOCON format would look like this:

```

1 Database {
2   url = "jdbc:postgresql://localhost:5432/mydb"
3   username = "db_user"
4   password = ${?DATABASE_PASSWORD}
5   allowed-schemas = ["public", "auth", "audit"]
6

```

```

7   Pool {
8     max-size = 10
9     min-size = 1
10    idle-timeout = 400
11    max-lifetime = 7200
12  }
13 }
```

This configuration file contains two levels of nesting: Database and Pool. It also supports multiple values for the allowed-schemas key, so its structure reflects the same structure as the case classes.

To support the same functionality with flat config providers, we have two challenges: first, how to encode nested keys in a flattened way, and second, how to encode multiple values with a single key. The solution is to use delimiters to indicate new levels of nesting and also for separating multiple values.

The path delimiter (pathDelim) is used to represent nested keys, and the seq delimiter (seqDelim) is used to represent multiple values. For example, assume we have the following environment config provider:

```
1 ConfigProvider.fromEnv(pathDelim = "_", seqDelim = ",")
```

It uses the underline character as the path delimiter and the comma character as the sequence delimiter. So the corresponding environment variables would look like this:

```

1 # .env file
2 DATABASE_URL=jdbc:postgresql://localhost:5432/mydb
3 DATABASE_USERNAME=db_user
4 DATABASE_PASSWORD=mypassword
5 DATABASE_ALLOWED_SCHEMAS=public, auth, audit
6 DATABASE_POOL_MAX_SIZE=10
7 DATABASE_POOL_MIN_SIZE=1
8 DATABASE_POOL_IDLE_TIMEOUT=400
9 DATABASE_POOL_MAX_LIFETIME=7200
```

You can load these variables into environment variables by storing them in a .env file and executing the source .env command in a shell. Please note how variables are named with the path delimiter. Also, note how the DATABASE\_ALLOWED\_SCHEMAS variable contains an array of strings each delimited by a comma.

## 20.5 Conclusion

Configuration management is a critical aspect of building robust, cloud-native applications, and ZIO provides a powerful, type-safe approach to handling this complexity. Throughout this chapter, we explored how ZIO's configuration system addresses common challenges through two core concepts: Config Descriptors and Config Providers.

Config Descriptors enable us to define type-safe configuration schemas, supporting everything from simple primitive values to complex nested structures. The ability to compose descriptors, provide validation rules, and handle defaults makes it possible to create precise, maintainable configuration definitions that catch errors when reading configurations.

Config Providers serve as the bridge between your application and its configuration sources, offering a flexible mechanism to load settings from various sources. By supporting environment variables, system properties, HOCON files, and custom implementations, Providers make it straightforward to handle different deployment scenarios—from local development to production environments.

As you build more complex applications with ZIO, the principles and patterns covered in this chapter will serve as a foundation for managing configuration in a type-safe, maintainable way. The system can scale well from simple applications to complex deployments requiring multiple configuration sources.

## 20.6 Exercise

1. Write a reloadable service that watches for changes in the configuration source and reloads the service with new configurations.
2. As your application becomes more complex, you may need to manage different configurations for different environments, each one a separate file, such as `Development.conf`, `Testing.conf`, and `Production.conf`. Write a config provider that loads configurations based on the environment variable `APP_ENV`.
3. Write a configuration descriptor for the `DatabaseConfig` which only accepts either `MysqlConfig` or `SqliteConfig` configurations.

# Chapter 21

## 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. In this chapter and the next, we will see how problems that seem quite challenging with other paradigms become incredibly simple using STM.

But before doing that, we need to understand more about the problem software transactional memory addresses and how ZIO's STM data type solves it.

### 21.1 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 takes 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:

```
1 import zio._  
2  
3 for {
```

```

4   ref      <- Ref.make(0)
5   increment = ref.get.flatMap(n => ref.set(n + 1))
6   _        <- ZIO.collectAllPar(ZIO.replicate(100)(increment))
7   value    <- ref.get
8 } yield value

```

Here, we might expect the `value` to be 100 because we executed one hundred effects in parallel, each incrementing 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 `Ref#get` and `Ref#set` separately instead of using `Ref#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 `Ref#update` or `Ref#modify` operators:

```

1 for {
2   ref      <- Ref.make(0)
3   increment = ref.update(_ + 1)
4   _        <- ZIO.collectAllPar(ZIO.replicate(100)(increment))
5   value    <- ref.get
6 } yield value

```

Now, the updates take 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 was replace `Ref#get` and `Ref#set` with `Ref#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:

```

1 def transfer(
2   from: Ref[Int],
3   to: Ref[Int],
4   amount: Int
5 ): Task[Unit] =
6   for {
7     senderBalance <- from.get
8     _ <- if (amount > senderBalance)
9       ZIO.fail(

```

```

10         new Throwable("insufficient funds")
11     )
12     else
13       from.update(_ - amount) *>
14         to.update(_ + amount)
15   } 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 to 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`. However, this allows us to do two transfers and end up with a negative balance on the sender's account.

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. 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:

```

1 final case class Balances(from: Int, to: Int)
2
3 def transfer(balances: Ref[Balances], amount: Int): Task[Unit] =
4   balances.modify { case Balances(from, to) =>
5     if (amount > from)
6       (
7         Left(new Throwable("insufficient funds")),
8         Balances(from, to)
9       )
10    else
11      (
12        Right(()),
13        Balances(from - amount, to + amount)
14      )
15  }.absolve

```

Now, we have fixed the concurrency bug. We perform all of our work within a single `Ref #modify` operation, so we have a guarantee that it will be performed atomically.

If the transfer amount exceeds the sender 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 having one Ref with all the states we need to maintain concurrently only works on a 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 need to maintain a large number of different pieces of state.

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 is to add more pieces of state to the Ref. At the extreme, we could add the balance of every account in our system to a single Ref, which has severe 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, which 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 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.

How 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.

## 21.2 Conceptual Description of STM

Recall that Ref and its nonfunctional equivalent AtomicReference were both based on *compare and swap* operations.

In our discussion above, we discussed 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 suspended. For performance reasons, we just accept the possibility that there could be conflicting updates and *retry* if there is a conflicting update.

This guarantees 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:

```

1 import java.util.concurrent.atomic.AtomicReference
2
3 trait Ref[A] {
4     def update(f: A => A): UIO[Unit]
5 }
6
7 object Ref {
8     def make[A](a: A): UIO[Ref[A]] =
9         ZIO.succeed {
10         new Ref[A] {
11             val atomic = new AtomicReference(a)
12             def update(f: A => A): UIO[Unit] =
13                 ZIO.succeed {
14                     var loop = true
15                     while (loop) {
16                         val old      = atomic.get
17                         val updated = f(old)
18                         loop = !atomic.compareAndSet(old, updated)
19                     }
20                 }
21             }
22         }
23     }

```

In the implementation of Ref#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 AtomicReference value 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 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 non-atomic and made it atomic to all outside observers.

Conceptually, it seems like we should be able to atomically update 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

We could also do a version of that manually. However, doing it manually requires a lot of work, is error-prone, and doesn't scale well.

This amounts to having to implement the 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 `Ref#update` method on `Ref` because it is just an arbitrary ZIO effect.

We don’t have the 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 the 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.

## 21.3 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.

```
1 import zio.stm._

2 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 have an ordinary ZIO value that we can no longer compose atomically with other transactions, so we want to ensure 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.

```
1 trait TRef[A] {
2   def get: STM[Nothing, A]
3   def modify[B](f: A => (B, A)): STM[Nothing, B]
4   def set(a: A): STM[Nothing, Unit]
5   def update(f: A => A): STM[Nothing, Unit]
6 }
```

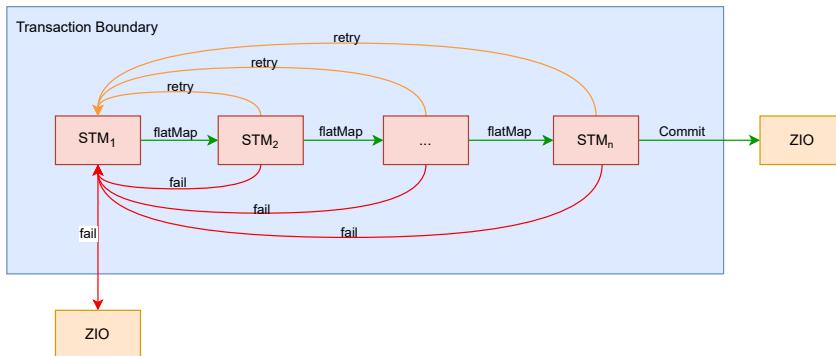


Figure 21.1: STM Transaction

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:

```

1 import zio._
2 import zio.stm._

3
4 for {
5   ref      <- TRef.make(0).commit
6   increment = ref.get.flatMap(n => ref.set(n + 1)).commit
7   -       <- ZIO.collectAllPar(ZIO.replicate(100)(increment))
8   value    <- ref.get.commit
9 } yield value

```

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 use `TRef#update`, but let's see how much simpler the implementation of the `transfer` method can become with STM:

```

1 def transfer(
2   from: TRef[Int],
3   to: TRef[Int],
4   amount: Int
5 ): STM[Throwable, Unit] =
6   for {
7     senderBalance <- from.get
8     _ <- if (amount > senderBalance)
9       STM.fail(new Throwable("insufficient funds"))
10    else
11      from.update(_ - amount) *>
12      to.update(_ + amount)
13 } yield ()

```

This now works correctly 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. They can 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.

```

1 def fund(
2   senders: List[TRef[Int]],
3   recipient: TRef[Int],
4   amount: Int
5 ): STM[Throwable, Unit] =
6   ZSTM.foreachDiscard(senders) { sender =>
7     transfer(sender, recipient, amount)
8   }

```

This will transfer funds from each sender to the recipient's 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.

The entire transaction will be performed atomically, so there is no risk that funds will be transferred from one user to another 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 ZSTM#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:

```

1 final class Balance private (
2     private[Balance] val value: TRef[Int]
3 ) { self =>
4     def transfer(that: Balance, amount: Int): Task[Unit] = {
5         val transaction: STM[Throwable, Unit] = for {
6             senderBalance <- value.get
7             _ <- if (amount > senderBalance)
8                 STM.fail(
9                     new Throwable("insufficient funds")
10                )
11             else
12                 self.value.update(_ - amount) *>
13                 that.value.update(_ + amount)
14         } yield ()
15         transaction.commit
16     }
17 }
```

Now, we have almost entirely hidden from the user the fact that we use STM in our implementation. The disadvantage is that 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 recommend returning STM transactions in your own operators to give users the flexibility to compose them together. However, 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 `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
- `foldSTM` 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

In the next section, we will see why there are no operators on STM for concurrency or

arbitrary effects.

In addition to this, there are some operators that are specific to STM. The most important of these is the `ZSTM#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 has changed. This can be useful to suspend until a condition is met.

```

1 def autoDebit(
2     account: TRef[Int],
3     amount: Int
4 ): STM[Nothing, Unit] =
5   account.get.flatMap { balance =>
6     if (balance >= amount) account.update(_ - amount)
7     else STM.retry
8 }
```

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 waste system resources by checking 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 might still need to continue retrying if the new account balance is insufficient, 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:

```

1 for {
2   ref    <- TRef.make(0).commit
3   fiber <- autoDebit(ref, 100).commit.fork
4   _      <- ref.update(_ + 100).commit
5   _      <- fiber.await
6 } yield ()
```

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.

## 21.4 Retryng and Repeating Transactions

Retrying (`STM.retry`) is completely different from repeating (`STM#repeatUntil` or `STM#repeatWhile`) a transaction. When a transaction encounters a conflict or fails validation, it is retried, meaning that the transaction is re-executed from the beginning only after the underlying transactional variables have changed. Also, it doesn't maintain the transaction state between retries:

```

1 def autoDebit(
2   account: TRef[Int],
3   amount: Int
4 ): STM[Nothing, Unit] =
5   account.get.flatMap { balance =>
6     if (balance >= amount)
7       account.update(_ - amount)
8     else
9       STM.retry
10 }
```

This retry mechanism is essential for ensuring consistency and correctness in concurrent programs. However, repeating a transaction has different behavior. When you repeat a transaction using `STM#repeatUntil` or `STM#repeatWhile`, it will use a busy loop and consume the underlying thread to repeatedly execute the transaction until the condition specified by the combinator is met. Unlike retrying, repeating doesn't wait for the transactional variables to change before repeating the transaction. This can lead to a busy-waiting scenario, where the transaction consumes CPU resources. Another difference is that the transaction state is maintained between repeats.

In the following example, taking a value from a queue is repeated until the value is equal to "bar":

```

1 val program =
2   for {
3     queue <- TQueue.unbounded[String].commit
4     _      <- queue.offerAll(List("foo", "bar", "baz", "qux"))
5     commit
6     _      <- queue.take.repeatUntil(_ == "bar").commit
7   } yield ()
```

Don't worry, you'll learn about `TQueue` in the next chapter.

## 21.5 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 performing pure computations, such as adding two numbers or changing the value of transactional variables. If a conflicting change occurs, we can 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:

```

1 | val arbitraryEffect: UIO[Unit] =
2 |   ZIO.succeed(println("Running"))

```

We would run this effect when we tried to execute the STM transaction by 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. In fact, adding observable side effects like this can be useful for debugging STM transactions. However, in general, allowing arbitrary side effects to be performed within STM transactions would make it impossible to reason about transactions because we would not know how often 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, there is no need for concurrency within an STM transaction because STM transactions do not perform 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:

```

1 | for {
2 |   allice    <- TRef.make(100).commit
3 |   bob       <- TRef.make(100).commit
4 |   carol    <- TRef.make(100).commit
5 |   transfer1 = transfer(allice, bob, 30).commit
6 |   transfer2 = transfer(bob, carol, 40).commit
7 |   transfer3 = transfer(carol, allice, 50).commit
8 |   transfers = List(transfer1, transfer2, transfer3)

```

```

9 |     _           <- ZIO.collectAllParDiscard(transfers)
10| } yield ()
```

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 possibility 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, idempotent effects 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<sup>1</sup> library].

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” because it allows us to solve many hard concurrency problems easily. However, it is important to remember that it is still ultimately implemented in terms of retrying if any of the transactional variables change.

One way you can help avoid this is to be conscious about the number of transactional variables you include 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:

---

<sup>1</sup><https://github.com/VladKopanev/zio-saga>

```

1 def chargeFees(
2   accounts: Iterable[TRef[Int]]
3 ): STM[Nothing, Unit] =
4   STM.foreachDiscard(accounts)(account => account.update(_ - 20))
5
6 for {
7   accounts <- STM.collectAll(STM.replicate(10000)(TRef.make(100)))
8   _           <- chargeFees(accounts)
9 } yield ()

```

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 10,000 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.

## 21.6 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.

## 21.7 Exercises

1. Create a concurrent counter using `TRef` and STM. Implement increment and decrement operations and ensure thread safety when multiple transactions modify the counter concurrently.
2. Implement a simple countdown latch using ZIO STM's `TRef`. A countdown latch starts with a specified count (`n`). It provides two primary operations:
  - `countDown`: Decrements the count by one but does nothing if it is already zero.
  - `await`: Suspends the calling fiber until the count reaches zero, allowing it to proceed only after all countdowns have been completed.

Note: This exercise is for educational purposes to help you understand the basics of STM. ZIO already provides a `CountDownLatch` implementation with more basic concurrency primitives.

3. Implement a read-writer lock using STM. A read-writer lock allows multiple readers to access a resource concurrently but requires exclusive access for writers. Implement the following operations:

```
1 trait ReadWriteLock {  
2     def readWith[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A]  
3     def writeWith[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A]  
4 }
```

## Chapter 22

# Software Transactional 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 straightforward 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 a state within the context of an STM transaction, so there is really nothing else.

Typically, the most straightforward implementation will 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 the value at index 1 was in the process of being updated. However, 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 granular 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 structures are implemented in terms of other STM data structures, as we will see.

## 22.1 Description of STM Data Structures

With that introduction, let's dive into each STM data structure currently provided by ZIO.

We will provide a high-level description of each data structure, discuss its use, 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. 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.

### 22.1.1 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 of each index is represented by its own TRef.

Because the size of a TArray is fixed, the values at each index are fully independent of each other. This allows for an extremely efficient representation as an `Array[TRef[A]]`:

```
:| import zio.stm._
```

```

2
3 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` when you would typically 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. Thus, 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 `TArray.make` constructor, which accepts a variable arguments list of elements, or the `TArray.fromIterable` constructor, which accepts an `Iterable`.

The most basic methods on `TArray` are `TArray#apply`, which accesses the value at a specified index, and `TArray#update`, which updates the value at an index with the specified function. You can also determine the array size using the `TArray#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 `TArray#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:

```

1 import zio.stm._

2
3 def swap[A](
4   array: TArray[A],
5   i: Int,
6   j: Int
7 ): STM[Nothing, Unit] =
8   for {
9     a1 <- array(i)
10    a2 <- array(j)
11    _ <- array.update(i, _ => a2)
12    _ <- array.update(j, _ => a1)
13  } 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 indices 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 represents STM equivalents of *mutable* data structures rather than *immutable* ones.

The `map` operator takes each element of a collection and returns a *new* collection, 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 `TArray#transform` operator for this use case.

```

1 val transaction = for {
2   array <- TArray.make(1, 2, 3)
3   _      <- array.transform(_ + 1)
4   list  <- array.toList
5 } yield list
6
7 transaction.commit
// List(2, 3, 4)
```

### 22.1.2 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` when you would otherwise use a mutable map but are in a transactional context. In particular, `TMap` is suitable for when you want 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 `TMap.make` or `TMap.fromIterable` constructor, as well as the `TMap.empty` constructor that just creates a new empty `TMap`.

The primary operators specific to `TMap` are `delete`, `get`, and `put`.

```

1 trait TMap[K, V] {
2   def delete(k: K): STM[Nothing, Unit]
3   def get(k: K): STM[Nothing, Option[V]]
4   def put(k: K, v: V): STM[Nothing, Unit]
5 }
```

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:

```

1 def getOrElseUpdate[K, V](
2   map: TMap[K, V]
3 )(k: K, v: => V): STM[Nothing, V] =
4   map.get(k).flatMap {
5     case Some(v) =>
6       STM.succeed(v)
7     case None =>
8       STM.succeed(v).flatMap(v => map.put(k, v).as(v))
9 }
```

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 are 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.

### 22.1.3 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 “smallest” or “largest” value.

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 TPriorityQueue.`make`, TPriorityQueue.`.fromIterable`, or TPriorityQueue.`.empty` constructors we have seen before, except this time we must also provide an implicit Ordering that will be used for the values:

```

1 import zio._
2
3 final case class Event(time: Int, action: UIO[Unit])
4
5 object Event {
6   implicit val EventOrdering: Ordering[Event] =
7     Ordering.by(_.time)
8 }
9
10 for {
11   queue <- TPriorityQueue.empty[Event]
12 } yield queue

```

A TPriorityQueue supports the typical operations you would expect of a queue.

You can offer one or more values to the queue using the TPriorityQueue`#offer` and TPriorityQueue`#offerAll` operators.

You can take values from the queue using the TPriorityQueue`#take`, TPriorityQueue`#takeAll`, or TPriorityQueue`#takeUpTo` operators.

The TPriorityQueue`#take` operator will take a single value from the queue, retrying until there is at least one value in the queue.

The TPriorityQueue`#takeAll` operator will take whatever values are currently in the queue and return them immediately. The TPriorityQueue`#takeUpTo` operator is like `takeAll` in that it will always return immediately but only take up to the specified number of elements from the queue.

A `TPriorityQueue#takeOption` operator 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 `TPriorityQueue#peek` or `peekOption` operators.

The `TPriorityQueue#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 will always return immediately, returning `None` if no values are currently in the queue.

You can also check the size of the queue with the `TPriorityQueue#size` operator and take a snapshot of the current contents of the queue with the `TPriorityQueue#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 implementing `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.

#### 22.1.4 TPromise

A `TPromise` is the equivalent of the `Promise` data type in ZIO for the transactional context.

Use a `TPromise` whenever you would typically 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.

```
1 final class TPromise[E, A] private (
2   private val ref: TRef[Option[Either[E, A]]]
3 )
```

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.

```
1 import zio.stm._
```

```

3  final class TPromise[E, A] private (
4    private val ref: TRef[Option[Either[E, A]]]
5  ) {
6    def await: STM[E, A] =
7      ref.get.flatMap {
8        case Some(value) => STM.fromEither(value)
9        case None         => STM.retry
10       }
11   def done(value: Either[E, A]): STM[Nothing, Unit] =
12     ref.get.flatMap {
13       case None => ref.set(Some(value))
14       case _     => STM.unit
15     }
16   def poll: STM[Nothing, Option[Either[E, A]]] =
17     ref.get
18 }
```

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 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 reflects 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 `TPromise.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.

### 22.1.5 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.

```

1 import scala.collection.immutable.{Queue => ScalaQueue}
2
3 final class TQueue[A] private (
4   private val capacity: Int,
5   private val ref: TRef[ScalaQueue[A]]
6 )

```

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 discussed at the beginning of the chapter: We can almost always create an 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, we may be able to gain efficiency by splitting them 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, let's see how simple it can be to define some of the basic queue operations using this representation:

```

1 import scala.collection.immutable.{Queue => ScalaQueue}
2
3 final class TQueue[A] private (
4   private val capacity: Int,
5   private val ref: TRef[ScalaQueue[A]]
6 ) {
7   def offer(a: A): STM[Nothing, Unit] =
8     ref.get.flatMap { queue =>
9       if (queue.size == capacity) STM.retry
10      else ref.set(queue.enqueue(a))
11    }
12   def peek: STM[Nothing, A] =
13     ref.get.flatMap { queue =>
14       if (queue.isEmpty) STM.retry
15       else STM.succeed(queue.head)
16     }
17   def poll: STM[Nothing, Option[A]] =
18     ref.get.flatMap { queue =>
19       if (queue.isEmpty) STM.succeed(None)
20       else
21         queue.dequeue match {

```

```

22         case (a, queue) => ref.set(queue).as(Some(a))
23     }
24   }
25   def take: STM[Nothing, A] =
26     ref.get.flatMap { queue =>
27       if (queue.isEmpty) STM.retry
28       else
29         queue.dequeue match {
30           case (a, queue) => ref.set(queue).as(a)
31         }
32     }
33 }
```

The implementations are all straightforward, building on our ability to compose `get` and `set` atomically 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 `TQueue#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 `TQueue#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 `TQueue#poll` operator is even simpler. We don't have to retry at all because this operator should return immediately, so we 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 `TQueue#isEmpty` and `TQueue#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 `TQueue.bounded` or `TQueue.unbounded` constructors, which create a new empty `TQueue` with the specified capacity.

### 22.1.6 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.

```

1 trait TReentrantLock {
2   val acquireRead: STM[Nothing, Int]
```

```

3   val acquireWrite: STM[Nothing, Int]
4   val releaseRead: STM[Nothing, Int]
5   val releaseWrite: STM[Nothing, Int]
6 }
```

The `Int` in the return type describes the number of read or write locks outstanding after the transaction. We won't typically 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 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 simultaneously 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 fiber reading will access the current state at 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. However, as we discussed in the last chapter when discussing STM's limitations, STM does not support concurrency within the context of a single STM transaction.

So, for example, there is no point in doing this:

```

1 for {
2   lock  <- TReentrantLock.make
3   ref   <- TRef.make(0)
4   -     <- lock.acquireWrite
5   -     <- ref.update(_ + 1)
6   -     <- lock.releaseWrite
7   value <- ref.get
8 } yield value
```

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 of helper methods to facilitate this:

```

1 import zio._

2
3 trait TReentrantLock {
4     val acquireRead: STM[Nothing, Int]
5     val acquireWrite: STM[Nothing, Int]
6     val releaseRead: STM[Nothing, Int]
7     val releaseWrite: STM[Nothing, Int]
8
9     val readLock: ZIO[Scope, Nothing, Int] =
10        ZIO.acquireRelease(acquireRead.commit)(_ => releaseRead.
11        commit)
12
13     val writeLock: ZIO[Scope, Nothing, Int] =
14        ZIO.acquireRelease(acquireRead.commit)(_ => releaseRead.
15        commit)
16 }
```

We can see that `readLock` and `writeLock` are much like the `withPermitScoped` in that they return a scoped effect that describes acquiring and then releasing a lock. Both of these operators use `commit` to run the STM transactions to ordinary ZIO effects:

```

1 var bankAccount = 0
2
3 for {
4     lock <- TReentrantLock.make.commit
5     zio1 = ZIO.scoped {
6         lock.writeLock.flatMap { _ =>
7             ZIO.succeed(bankAccount += 100)
8         }
9     }
10    zio2 = ZIO.scoped {
11        lock.writeLock.flatMap { _ =>
12            ZIO.succeed(bankAccount += 50)
13        }
14    }
15    zio3 = ZIO.scoped {
16        lock.writeLock.flatMap { _ =>
17            ZIO.succeed(bankAccount += 25)
18        }
19    }
20 } <- ZIO.collectAllParDiscard(List(zio1, zio2, zio3))
```

```
21 } yield bankAccount
```

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:

```
1 trait Actor
2
3 trait Supervisor {
4   def lock: TReentrantLock
5   def supervise(actor: Actor): UIO[Unit]
6   def unsupervise(actor: Actor): UIO[Unit]
7 }
```

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 safely that is not subject to race conditions or deadlocks?

With STM, this is very easy:

```
1 def transfer(
2   from: Supervisor,
3   to: Supervisor,
4   actor: Actor
5 ): UIO[Unit] =
6   ZIO.acquireReleaseWith {
7     (from.lock.acquireWrite *> to.lock.acquireWrite).commit
8   } { _ =>
9     (from.lock.releaseWrite *> to.lock.releaseWrite).commit
10 } { _ =>
11   from.unsupervise(actor) *> to.supervise(actor)
12 }
```

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 condi-

tions.

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 an excellent way to “provide the best of both worlds”.

### 22.1.7 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 semaphores, 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:

```
1 final class TSemaphore private (private val permits: TRef[Long])
```

`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:

```
1 final class TSemaphore private (private val permits: TRef[Long])
  {
2   val acquire: STM[Nothing, Unit] =
3     permits.get.flatMap { n =>
4       if (n == 0) STM.retry
5       else permits.set(n - 1)
6     }
7   val release: STM[Nothing, Unit] =
8     permits.update(_ + 1)
9 }
```

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 `acquireRelease` and `Scope` 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`.

### 22.1.8 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 how `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.

## 22.2 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 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.

We will focus on the `TPriorityQueue` data structure, 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 primary operations:

```

1 trait TPriorityQueue[A] {
2
3   /**
4    * Offers a value to the queue.
5    */
6   def offer(a: A): STM[Nothing, Unit]
7
8   /**
9    * Takes the first value in the queue based on an `Ordering`
10   * defined on `A`.
11   */
12   def take: STM[Nothing, A]
13 }
```

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 primary 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, which 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:

```

1 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 an excellent 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 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.

Unfortunately, we will have to do a little more work 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 to 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:

```
1 import scala.collection.immutable.SortedMap  
2  
3 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.

```
1 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 an Int or a Long representing 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:

```

1 final case class Event(time: Long, action: UIO[Unit])
2
3 implicit val eventOrdering: Ordering[Event] =
4   Ordering.by(_.time)

```

This `Ordering` defines two events to have the same priority if they occur at the same time. However, two events happening 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`:

```

1 final case class TPriorityQueue[A] (
2   ref: TRef[SortedMap[A, List[A]]]
3 )

```

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 the 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:

```

1 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.

As we implement operators on `TPriorityQueue`, we will see how this additional information lets the compiler prove 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 picking the right representation without really having worked with any STM operators yet. However, 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:

```

1 final case class TPriorityQueue[A] (
2   ref: TRef[SortedMap[A, ::[A]]]
3 ) {
4   def offer(a: A): STM[Nothing, Unit] =
5     ref.update { map =>
6       map.get(a) match {
7         case Some(as) => map + (a -> ::(a, as))
8         case None      => map + (a -> ::(a, Nil))
9       }
10    }
11 }
```

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 insert the key, we check if it already exists in the map. If it does, we add the new value to the existing values associated with that key; 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:

```

1 final case class TPriorityQueue[A] (
2   ref: TRef[SortedMap[A, ::[A]]]
3 ) {
4   def take(a: A): STM[Nothing, A] =
5     ref.get.flatMap { map =>
6       map.headOption match {
7         case Some((a, as)) =>
8           as.tail match {
9             case h :: t =>
10               ref.set(map + (a -> ::(h, t))).as(as.head)
11             case Nil =>
12               ref.set(map - a).as(as.head)
13           }
14         case None => STM.retry
15       }
16     }
17 }
```

First, we call `get` to access the `SortedMap` inside the `TRef`. Then, we call `headOption` to 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. It 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.

## 22.3 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.

## 22.4 Exercises

1. Write a shopping cart application using ZIO STM that each cart item has a key and quantity. The application should support the following operations in concurrent environment:
  - Add an item to the cart (`addItem`)

- Remove an item from the cart (remove)
- Update the quantity of an item in the cart (update)
- Calculate the total price of the items in the cart (checkout)

Hint: Use a TMap to store the items in the shopping cart, where the key is a String representing the item's key, and the value is an Item that includes the item's price and quantity. Here is an example:

```
1 case class Item(price: Double, quantity: Int)
2 case class ShoppingCart(items: TMap[String, Item])
```

2. Implement a red-black tree with ZIO STM that supports the following operations:

- Insert a new element
- Delete an element
- Search for an element
- Traverse the tree in-order
- Balance the tree after each insertion or deletion

Hint: Define a recursive RBNode data structure that represents the nodes of the Red-Black Tree, and store the entire tree encapsulated in a single TRef like this:

```
1 sealed trait Color
2 case object Red    extends Color
3 case object Black  extends Color
4
5 case class RBNode[K, V] (
6     color: Color,
7     key: K,
8     value: V,
9     left: Option[RBNode[K, V]],
10    right: Option[RBNode[K, V]])
11 )
12
13 class RedBlackTree[K, V](root: TRef[Option[RBNode[K, V]]])
```

3. Develop a real-time trading system that uses STM for placing orders and matching them on the order book. The system should support the following operations:

- Place a new order
- Cancel an order
- Update an order
- Match orders

Hint: Use TPriorityQueue to store the sell and buy orders and a TMap to store all the order books for each stock:

```
1 type Stock = String
2
3 case class Order(
```

```
4   id: Long,
5   stock: Stock,
6   price: Double,
7   quantity: Int,
8   isBuy: Boolean
9 )
10
11 case class OrderBook(
12   buyQueue: TPriorityQueue[Order],
13   sellQueue: TPriorityQueue[Order]
14 )
15
16 case class TradingSystem(orderBooks: TMap[Stock, OrderBook])
```

Please note that this is for pedagogical purposes only and is not intended for a real-world trading system. Maintaining the order book in real-world trading systems is much more complex and typically involves using more advanced architectural patterns and data structures.

## Chapter 23

# Software Transactional Memory: Advanced STM

Implementing Software Transactional Memory (STM) involves sophisticated mechanisms to manage concurrency, track changes, and ensure atomicity. In this chapter, we will explore the inner workings of STM in ZIO, including how STM transactions are executed, how changes are tracked, and how conflicts are detected and resolved. We will also discuss some advanced concepts and optimizations that can improve the performance of STM transactions.

### 23.1 How STM Works Under the Hood

When a transaction begins execution using `STM#commit` or `STM.atomically`, the ZIO STM runtime creates a virtual space for the transaction to operate. This space is isolated from other transactions, allowing the transaction to speculate the state of transactional variables without affecting the shared memory. This virtual space is called a transaction journal or transaction log.

Journal is a private data structure for each transaction that tracks the read and write operations, recording tentative changes to consistency before committing changes to shared memory. Each journal is a map of entries representing a transactional variable and its state during the transaction's execution. So, it is used to track changes, defer writes, detect conflicts, and roll back transactions in case of conflicts.

Each transaction passes through the following phases:

1. **Start:** When a transaction begins, a new journal is created. This journal will hold entries for all transactional variables the transaction interacts with.
2. **Tracking Reads and Writes:** As the transaction executes, the STM runtime intercepts all read and write operations. Reads and writes are not directly applied to shared variables (`TRefs`) but are instead recorded in the transaction journal:

- Reads: When a transaction reads from a transactional variable, an entry is added to the Journal noting the version of the transactional variable and its current value.
  - Writes: When a transaction writes to a transactional variable, the new value is recorded in the Journal but not immediately applied to the transactional variable.
3. **Validation Phase:** Before the transaction commits, the STM uses the Journal to check whether the versions of the transactional variables recorded in the journal still match the current versions in the shared memory. If any of the variables have changed (indicating another transaction has modified them), the transaction is considered invalid and will be retried.
  4. **Commit or Rollback:** If the transaction passes the validation phase, the STM system uses the journal to apply all the recorded writes to the actual transactional variables in an atomic operation. If validation fails, the transaction is rolled back, and the journal is discarded or reset, ensuring that no partial changes affect the shared state.

All these steps are heavily based on the transaction journal, which acts as a temporary storage for the transaction's changes. So let's take a closer look at how the journal works. Assume it is defined as below:

```
| type Journal = MutableMap[TRef[_], Entry]
```

Keys in a journal are transactional variables (TRefs), and values are Entry objects that track the state of the transactional variable during the transaction's execution.

Each entry contains the following fields and information:

1. **tref:** A reference to the corresponding transactional variable (TRef) in shared memory that the entry is tracking. A TRef holds the actual value of a transactional variable, which is shared across transactions.
2. **expected:** The expected version of the tref when the transaction attempts to commit. This value is used to check whether the tref has been modified by another transaction since the current transaction began, ensuring consistency.
3. **newValue:** A temporary or tentative value calculated and stored by the transaction during its execution. This value is not committed to the tref until the transaction is successfully completed.
4. **isChanged:** A flag indicating whether the transaction has modified the value of the tref. If the transaction has made changes to the tref during its execution, this flag is set to true.
5. **isValid:** A flag indicating whether the tref has not been modified by another transaction since the current transaction began. If the current version of the transactional variable (tref.versioned) matches the expected value, the entry is considered valid (tref.versioned eq expected).

For example, consider the following application that transfers money between two accounts using STM:

```
| import zio._
```

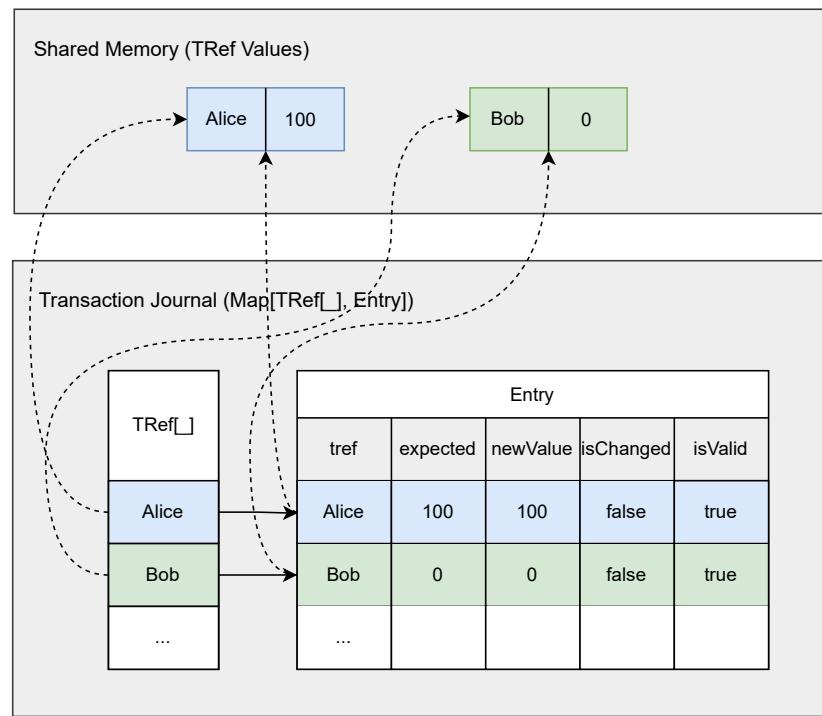


Figure 23.1: STM Journal

```
2 import zio.stm._
3
4 val app =
5   for {
6     alice <- TRef.make(100).commit.debug
7     bob   <- TRef.make(0).commit.debug
8     _     <- transfer(alice, bob, 50).commit
9     _     <- alice.get.commit.debug("from")
10    _     <- bob.get.commit.debug("to")
11  } yield ()
12
13 def withdraw(
14   from: TRef[Int],
15   amount: Int
16 ): STM[Throwable, Unit] =
17   for {
18     // 2. Read the balance
19     balance <- from.get
20     // 3. If the amount is greater than the balance fails the
21     //      transaction
22     _ <- if (amount > balance)
23       STM.fail(new Throwable("insufficient funds"))
24     else
25       // 4. Update the balance by subtracting the amount
26       from.update(_ - amount)
27   } yield ()
28
29 def transfer(
30   from: TRef[Int],
31   to: TRef[Int],
32   amount: Int
33 ): STM[Throwable, Unit] =
34   for {
35     // 1. Read the sender's balance
36     senderBalance <- from.get
37     // 5. If the amount is greater than the sender's balance
38     //      fails the transaction
39     _ <- if (amount > senderBalance)
40       STM.fail(new Throwable("insufficient funds"))
41     else
42       // 6. Update the sender's balance by subtracting the amount
43       from.update(_ - amount) *>
44       // 7. Update the receiver's balance by adding the amount
45       to.update(_ + amount)
46   } yield ()
```

The `transfer` transaction uses two transactional variables, `from` and `to`, which hold integer values. When the transaction starts, the STM runtime intercepts any read or write operations on these variables. Instead of immediately updating the shared memory, the runtime records these operations as tentative changes in the journal.

Let's walk through the execution of the `transfer` transaction to understand how the journal works. After reading Alice's balance, the runtime system creates a journal entry for the `alice` variable, reflecting its initial state:

```

1 // Shared memory
2 alice = TRef(id = 421882409, versioned.value = 100)
3 bob   = TRef(id = 538841410, versioned.value = 0)
4
5 // The journal for the transfer transaction
6 Map(
7   alice -> Entry(expected.value = 100, newValue = 100, tref =
8     from, isChanged = false)
9 )

```

Meanwhile, another fiber might begin executing a concurrent transaction to update the `alice` variable because Alice is requesting a withdrawal:

```

1 val app =
2   for {
3     alice <- TRef.make(100).commit.debug
4     bob   <- TRef.make(0).commit.debug
5     _      <- transfer(alice, bob, 50).commit && withdraw(alice,
6       70).commit.debug
7     _      <- alice.get.commit.debug("from")
8     _      <- bob.get.commit.debug("to")
9   } yield ()

```

The second transaction, namely the withdrawal transaction `withdraw(alice, 70)`. `commit`, will also have its own journal, which contains a single entry for the `alice` variable:

```

1 // Shared memory
2 alice = TRef(id = 421882409, versioned.value = 100)
3 bob   = TRef(id = 538841410, versioned.value = 0)
4
5 // Withdrawal transaction journal
6 Map(
7   alice -> Entry(expected.value = 100, newValue = 100, tref =
8     alice, isChanged = false)
9 )

```

Suppose the fiber running the second transaction proceeds before the first transaction completes. The withdrawal transaction checks whether `alice` has enough funds to with-

draw 70 and finds that the balance is sufficient. It then subtracts the amount from the balance and updates the journal entry to reflect the change:

```

1 // Shared memory
2 alice = TRef(id = 421882409, versioned.value = 100)
3 bob = TRef(id = 538841410, versioned.value = 0)
4
5 // Withdrawal transaction journal
6 Map(
7   alice -> Entry(expected.value = 100, newValue = 30, tref =
8     alice, isChanged = true)
9 )

```

This entry indicates that the transaction has modified the value of `alice` from 100 to 30, and the `isChanged` flag is set to `true` which means that the transaction has made changes to the transactional variable.

In the final stage, the runtime validates the journal associated with the withdrawal transaction. It confirms that the `alice` variable has not been modified by any other transaction, and its expected value matches the current version in Alice's `tref`. This validation ensures it is safe to commit the changes. The runtime then applies the new value of 30 to the `alice` variable, making the changes visible to other transactions. The shared memory is updated as follows:

```

1 // Shared memory
2 alice = TRef(id = 421882409, versioned.value = 30)
3 bob = TRef(id = 538841410, versioned.value = 0)

```

Now, suppose the first fiber, which was executing the transfer transaction, gets a chance to proceed. In step 5, the transfer transaction checks whether the transfer amount is greater than Alice's balance. Since Alice's balance was previously 100, it will continue to steps 6 and 7 to update the sender's and receiver's balances. The journal entries for the `alice` and `bob` will be updated as follows:

```

1 // Shared memory
2 alice = TRef(id = 421882409, versioned.value = 30)
3 bob = TRef(id = 538841410, versioned.value = 0)
4
5 // Transfer transaction journal
6 Map(
7   alice -> Entry(expected.value = 100, newValue = 50, tref =
8     alice, isChanged = true),
9   bob -> Entry(expected.value = 0, newValue = 50, tref = bob,
10     isChanged = true)
11 )

```

The journal entries indicate that the transaction has modified the values of both the `alice` and `bob` variables. The entry for `alice` states that it intends to change Alice's balance

from 100 to 50, but the current value of `alice` is 30, indicating that another transaction has already modified it. This conflict will cause the runtime to retry the entire transaction to ensure consistency.

On the next retry, the transfer transaction will start from the beginning, at step 1, and read the sender's balance again. This time, it will retrieve Alice's updated balance, which is now 30. Since the transfer amount is greater than the sender's balance, the transaction will fail, preventing any risk of a double-spending issue.

## 23.2 Troubleshooting and Debugging

ZIO STM is built around modular concurrency design principles, enabling developers to write concurrent programs that are both modular and composable. This approach significantly enhances the maintainability, readability, and testability of code while also reducing the complexity of concurrent programming. When it comes to STM transactions, the debugging process is further complicated by the fact that STM transactions can be rolled back and retried multiple times. This behavior requires developers to adopt a different mindset for debugging—focusing less on traditional step-by-step execution and more on understanding the broader transactional flow, including potential conflicts and retries, to effectively diagnose and resolve issues.

If you are facing issues with STM transactions, here are some tips to help you debug your code:

1. **Isolate the Problem:** When dealing with a complex transaction that isn't behaving as expected, it's essential to break it down into smaller, more manageable components. Large transactions can be prone to a variety of issues, such as conflicts, rollbacks, or performance bottlenecks, which can make diagnosing the problem challenging. By decomposing the transaction into smaller segments, you can isolate and test each segment independently, which simplifies the debugging process and helps you pinpoint the exact part that is causing issues.

Keep the boundaries of each transaction as small as possible. Smaller transactions are less likely to conflict with others, which reduces the likelihood of errors. This approach allows you to pinpoint the problematic segment more efficiently by focusing on isolated, smaller units before integrating them back together.

2. **Write Unit Tests:** Writing unit tests that cover a variety of scenarios, including those where multiple transactions are executed and interleaved, is crucial for identifying issues. It's a best practice to create test cases for all possible scenarios and edge cases. For example, you can simulate conditions where two transactions attempt to update the same transactional variable simultaneously by using synchronization primitives like mutexes and barriers. These tests allow you to observe how your transactions behave under different conditions within a controlled concurrent environment.

When testing for concurrency, it's vital to ensure that your tests are deterministic and reproducible, meaning they should yield the same results every time they run,

regardless of the environment or execution order. Non-deterministic tests can lead to flaky results, making it challenging to pinpoint the root cause of an issue. To mitigate this, consider applying the `TestAspect.nonFlaky` test aspect to your test cases, which ensures that your tests consistently produce reliable outcomes and prevent false positives in your test results.

3. **Use effectful logging:** Logging is a simple yet effective way to debug your program. You can add logging statements to see the state of the transactional variables and how the transaction flow is executed. In previous chapters, we have seen that due to the nature of STM transactions, as they can be retried multiple times, we cannot run side effects inside transactions. This is why the `ZSTM` data type doesn't provide any direct way to execute a side effect inside a transaction. However, you can debug transactions by adding side effects inside the transaction while getting/updating the transactional variables.

We know that if we run `ZSTM.succeed{ println("Hello, World!"); 42}`, this effect will be executed immediately, and the result will be evaluated. So, while running this transaction, the side effect will be executed, and you can see the "Hello, World!" message printed in the console, and the value 24 will be evaluated for the potential result of the transaction. Let's apply this technique to the `withdraw` transaction in the previous example:

```

1 def withdraw(
2     from: TRef[Int],
3     amount: Int
4 ): STM[String, Unit] =
5   for {
6     balance <- from.get
7     _      <- ZSTM.succeed { println(s"withdraw: start
8       withdrawing $amount from given account with balance
9       $balance") }
10    _ <- if (amount > balance)
11      STM.fail("insufficient funds")
12    else
13      from.update { b =>
14        println(s"withdraw: updating balance from $b to ${b
15        - amount}")
16        b - amount
17      }
18    } yield ()

```

These debug statements provide valuable insight into the transaction's flow and allow you to observe the state of transactional variables at various points during execution. This visibility can help pinpoint where conflicts arise and how often the transaction is retried.

However, be aware that adding logging statements in concurrent programs can sometimes alter their behavior, potentially leading to Heisenbugs. A Heisenbug is a

type of software bug that seems to disappear or change behavior when you attempt to investigate it. This can occur in concurrent programs because the introduction of side effects, such as logging, can influence the timing and order of execution, leading to different outcomes and making the original issue more difficult to reproduce.

## 23.3 Optimization

Now that we understand how ZIO STM works under the hood, we can discuss optimization techniques that improve the performance of STM transactions.

STM uses an optimistic concurrency control mechanism, which means it proceeds with the assumption that conflicts with other transactions are unlikely rather than immediately locking resources. However, this approach can lead to situations where a transaction has to be retried multiple times, resulting in wasted work and performance degradation. To optimize the performance of STM transactions, we need to minimize the number of conflicts and retries. In this section, we will discuss some strategies to achieve this.

### 23.3.1 Narrowing the Transactional Boundaries

A transactional boundary defines the scope within which a series of operations on shared memory are executed as a single atomic unit. These boundaries ensure that the operations inside them are performed in an atomic, consistent, and isolated manner, meaning that either all changes are applied successfully if the transaction commits or none are applied if the transaction aborts due to conflicts or errors.

In ZIO STM, a transactional boundary is defined by the `STM.atomically` combinator, which wraps a series of STM operations to be executed atomically:

```

1 | val transaction =
2 |   STM.atomically {
3 |     ???
4 |   }
```

The larger the transaction, the higher the likelihood of conflicts during validation. Remember that ZIO uses a lazy conflict detection mechanism, which means that conflicts are detected only when a transaction wants to commit its changes. The problem with large transactions is that they tend to read from and write to multiple transactional variables, leading to longer speculative execution and increasing the chances of conflicts.

Whenever you write a transaction, try to keep it as small as possible. If you see parts of the transaction that can be extracted and are not part of its core logic, consider moving them out of the transaction. This way, you can reduce the number of transactional variables accessed by the transaction, minimizing the window of time during which conflicts can occur.

### 23.3.2 Fine-grained Locking

Transactional values are shared across transactions, so the more fine-grained the transactional values are, the less likely they are to conflict with other transactions. This approach allows different parts of the data structure to be updated independently, reducing the likelihood of conflicts.

Let's consider a scenario where we want to create a concurrent sorted linked list using STM. The simplest approach would be to wrap the entire linked list in a single TRef:

```

1 private final case class Node[A](value: A, next: Option[Node[A]])
2
3 class SortedLinkedList[A] private (head: TRef[Option[Node[A]]]) {
4   def insert(value: A): ZIO[Any, Nothing, Unit] = ???
5 }
```

This approach works but can lead to contention and conflicts when multiple transactions attempt to update different parts of the linked list simultaneously in a highly concurrent environment. When we wrap the whole data structure in a single TRef, it's like we have a global lock on the entire data structure. So, if two transactions want to update two different nodes in the linked list, they will conflict with each other, and one of them will be retried.

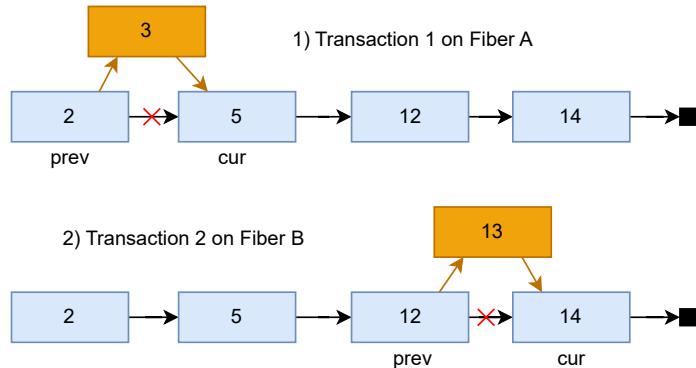


Figure 23.2: Sorted Linked List

To optimize this, you can use a fine-grained locking approach by wrapping each node in a separate TRef:

```

1 final case class Node[A] (
2   value: A,
3   next: TRef[Option[Node[A]]]
4 )
5
6 class SortedLinkedList[A] (
```

```

7   private val head: TRef[Option[Node[A]]]
8 )(implicit ord: Ordering[A]) {
9   def insert(value: A): ZIO[Any, Nothing, Unit] = ???  

10 }

```

With this approach, each node in the linked list is isolated in its own transactional variable. This isolation makes the insert operation more efficient. Even when multiple transactions are updating different nodes in the linked list, they are less likely to conflict with each other.

Let's see how the implementation of these two approaches differs in the `insert` method. Let's start with the coarse-grained locking implementation:

```

1 // Coarse-grained locking
2
3 import zio._
4 import zio.stm._
5
6 final case class Node[A](
7   value: A,
8   next: Option[Node[A]]
9 )
10
11 class SortedLinkedList[A](
12   private val head: TRef[Option[Node[A]]]
13 )(implicit ord: Ordering[A]) {
14
15   // Insert a new value into the sorted linked list
16   def insert(value: A): ZIO[Any, Nothing, Unit] = {
17     STM.atomically {
18       for {
19         currentHead <- head.get
20         sortedHead <- insertSorted(value, currentHead)
21         _           <- head.set(sortedHead)
22       } yield ()
23     }
24   }
25
26   // Helper method to insert a value in sorted order
27   private def insertSorted(
28     value: A,
29     currNodeOpt: Option[Node[A]]
30   ): STM[Nothing, Option[Node[A]]] = {
31     currNodeOpt match {
32       // If the list is empty, create a new node with the value
33       case None =>
34         STM.succeed(Some(Node(value, None)))

```

```

35     // If the value is less than or equal to the current node's
36     // value, insert the new node here
37     case Some(currentNode) if ord.lteq(value, currentNode.value)
38     ) =>
39         STM.succeed(Some(Node(value, currNodeOpt)))
40     // If the value is greater than the current node's value,
41     // recurse to the next node
42     case Some(currentNode) =>
43         insertSorted(value, currentNode.next)
44             .map(nextSorted => Some(currentNode.copy(next =
45                 nextSorted)))
46     }
47 }
```

The fine-grained locking implementation is quite similar:

```

1 // Fine-grained Locking
2
3 import zio._
4 import zio.stm._
5
6 // Define the Node class with next as a TRef
7 final case class Node[A](
8     value: A,
9     next: TRef[Option[Node[A]]]
10 )
11
12 class SortedLinkedList[A](
13     private val head: TRef[Option[Node[A]]]
14 )(implicit ord: Ordering[A]) {
15     // Insert a new value into the sorted linked list
16     def insert(value: A): ZIO[Any, Nothing, Unit] =
17
18         STM.atomically {
19             for {
20                 currentHead <- head.get
21                 newHead      <- insertSorted(value, currentHead)
22                 _           <- head.set(newHead)
23             } yield ()
24         }
25
26     // Helper method to find the correct insertion point and insert
27     // the node
28     private def insertSorted(
29         value: A,
30         currNodeOpt: Option[Node[A]]
```

```

30 ): STM[Nothing, Option[Node[A]]] =
31   currNodeOpt match {
32     // If the list is empty or we reach the end, create a new
33     // node
34     case None =>
35       TRef
36         .make(currNodeOpt)
37         .map(newNext => Some(Node(value, newNext)))
38     // If the new value should come before the current node
39     case Some(currNode) if ord.lteq(value, currNode.value) =>
40       TRef.make(currNodeOpt).map(newNext => Some(Node(value,
41         newNext)))
42     // Otherwise, continue to the next node
43     case Some(currNode) =>
44       for {
45         nextNode <- currNode.next.get
46         sortedNext <- insertSorted(value, nextNode)
47         _ <- currNode.next.set(sortedNext)
48       } yield currNodeOpt
49   }
50 }
```

The use-case scenario for both implementations is the same. Let's create a sorted linked list and insert some values into it:

```

1 import zio._
2
3 object Main extends ZIOAppDefault {
4   // The main program to test the SortedLinkedList
5   def run =
6     for {
7       // Create a new sorted linked list
8       list <- SortedLinkedList.empty[Int]
9
10      // Insert some values concurrently
11      _ <- ZIO.foreachParDiscard(List(5, 2, 7, 3)) { value =>
12        list.insert(value)
13      }
14      // Retrieve the current state of the list and print it
15      scalaList <- list.toList
16      _ <- ZIO.debug(s"Sorted List: ${scalaList.mkString(
17        ", ")}")
18    } yield ()
```

Fine-grained data structure enhances retry efficiency by enabling transactions to target more specific segments of the shared state. By leveraging STM.retryUntil, STM.

`retryWhile`, and `STM.check` combinators, we can control retries based on conditions applied to specific transactional variables. This approach minimizes unnecessary conflicts and contention, as it allows multiple transactions to operate concurrently on different segments of the shared state, improving overall system performance.

Please note that fine-grained locking can introduce additional complexity and overhead. Increasing the granularity of shared state can increase space complexity, as it requires more transactional variables to be monitored and managed by STM's journal. This can lead to higher memory consumption and potentially slower performance. Therefore, it's essential to strike a balance between granularity and performance based on your application's specific requirements. For example, fine-grained data structures may be unnecessary in scenarios with a low degree of contention and conflicts. In such cases, a more coarse-grained approach may suffice and make the code more readable and maintainable.

### 23.3.3 Diagnosing High-contention Critical Sections

Identifying transactional variables that are frequently accessed and modified by multiple transactions allows you to optimize these areas to reduce conflicts and retries. Symptoms of high-contention areas include high CPU usage, degraded throughput, and increased latency. If you notice these signs, consider diagnosing and profiling the problematic areas, then apply optimization techniques to alleviate the contention.

### 23.3.4 Prefer Delaying Write Operations

Delaying writes until the end of the transaction minimizes the window of time during which conflicts can occur, increasing the chances of successful execution without conflicts.

For example, instead of writing the following code for the transfer transaction:

```

1 def transfer(
2     amount: Int,
3     from: TRef[Int],
4     to: TRef[Int]
5 ): ZSTM[Any, String, Unit] =
6     for {
7         fromBalance <- from.get
8         _ <- if (fromBalance >= amount)
9             from.update(_ - amount) // Early write
10        else
11            STM.fail("Insufficient funds")
12         _ <- to.update(_ + amount) // Second write
13     } yield ()

```

Write the transaction as follows:

```

1 def transfer(
2     from: TRef[Int],
3     to: TRef[Int],

```

```

4   amount: Int
5 ): STM[String, Unit] =
6 {
7   senderBalance <- from.get
8   _ <- if (amount > senderBalance)
9     ZSTM.fail("insufficient funds")
10  else
11    // Delay the write operations until the end of the
12    transaction
13    from.update(_ - amount) *> to.update(_ + amount)
14 } yield ()

```

### 23.3.5 Deferring Commits to Shared State with Local Buffers

This technique is particularly effective in scenarios where frequent, minor updates to shared state create performance bottlenecks due to constant contention. If the intermediate state is not immediately required by others, these updates can be safely deferred without compromising correctness. By accumulating minor updates in a local buffer and committing them to the shared state only after reaching a certain threshold, you can significantly reduce the number of conflicts and retries, thereby improving overall system performance.

An excellent example of this approach is in real-time collaborative text editing applications, where multiple users may edit the same document simultaneously. These concurrent edits often lead to contention and conflicts. To optimize the editing process, each user's changes can be temporarily buffered locally and then committed to the shared document after a set interval or when specific conditions are met, all within a single transactional operation. This reduces the frequency of conflicts and retries, enhancing the application's responsiveness and performance.

## 23.4 Transaction Control Flows

We know that a transaction can be retried multiple times due to conflicts with other transactions or because of the transaction's own logic, such as the use of `STM.retry` or any of its variants, e.g., `STM#retryUntil`, `STM#retryWhile`, or `STM.check`. However, there are scenarios where you may not want to wait for a transaction to succeed in case of retries, or you may want to run another transaction in case of failure. You can introduce an alternative transaction using the `STM#orElse` combinator in such cases.

The `orElse` combinator allows you to specify an alternative transaction that will be executed if the original transaction fails or retries. This feature is particularly useful when providing fallback logic or handling specific error conditions without waiting for the original transaction to succeed. It allows the transaction to fail fast and avoid unnecessary retries.

Let's consider a scenario where both the first and alternative transactions retry. What

will happen to the whole transaction? The answer is that the entire transaction will only succeed if the underlying transaction variables of the alternative transaction are changed and a subsequent retry of the alternative transaction leads to a successful outcome. This means that after the original transaction retries, the STM runtime won't retry the original transaction again.

Another combinator called `STM#orTry` allows you to specify an alternative transaction that will be executed if the original transaction fails or retries. The difference between `STM#orElse` and `STM#orTry` is that `STM#orTry` will retry the whole transaction after both the original transaction and the alternative transaction encounter a retry. This combinator is useful when you want to provide fallback logic but still want the chance to retry the original transaction if the alternative transaction also encounters a retry.

## 23.5 Conclusion

In this chapter, we explored advanced concepts of Software Transactional Memory (STM) in ZIO, highlighting how STM manages concurrency, isolation, and atomicity. STM uses a transaction journal to track reads and writes, validate consistency, and commit changes atomically or roll back when conflicts occur.

Transactions move through key phases: initialization, read and write tracking, validation, and either commit or rollback. The transaction journal is central to this process, holding tentative changes and detecting conflicts.

Debugging STM transactions involves keeping transactions small and isolated, writing unit tests for concurrency, and using effectful logging judiciously. Performance can be optimized by narrowing transactional boundaries and applying fine-grained locking to minimize conflicts and retries.

Understanding these advanced STM concepts enables developers to leverage ZIO STM for writing concurrent, modular, and efficient applications, making concurrency more reliable and manageable.

## 23.6 Exercises

1. Improve the performance of the red-black tree you implemented in the previous chapter using fine-grained locking.

Hint: Instead of wrapping the entire tree in a single `TRef`, wrap each node in a separate `TRef` as well. You can minimize the chances of conflicts and retries by isolating each node in its own transactional variable.

2. The first naive implementation of a concurrent map (`TMap`) that comes to mind is to have a single `TRef` that holds a `Map` of keys and values:

```
1 case class TMap[K, V] private (private val map: TRef[Map[K,
  V]]) {
 2   def ut(key: K, value: V): STM[Nothing, Unit] =
```

```
3     map.update(_ + (key -> value))
4
5     def get(key: K): STM[Nothing, Option[V]] =
6         map.get.map(_.get(key))
7 }
```

However, this approach can lead to contention and conflicts when multiple transactions try to update different keys in the map simultaneously. Implement a more efficient version of TMap that updates to different keys in the map are isolated and do not conflict with each other in a concurrent environment.

## Chapter 24

# 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.

This chapter 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.

### 24.1 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 often and with what delay do we want to retry?
2. How should that change based on the type of failure?

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 rate-limited, 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:

```

1 import zio._
2 import zio.Clock._

3
4 def retryWithDelayN[R, E, A](
5   n: Int
6 )(
7   zio: ZIO[R, E, A]
8 )(duration: Duration): ZIO[R with Clock, E, A] =
9   if (n <= 0) zio
10  else
11    zio.catchAll { _ =>
12      retryWithDelayN(n - 1)(zio)(duration).delay(duration)
13    }

```

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. If this fails, we call `retryWithDelayN` again with the specified delay and one fewer remaining retry.

At one level, this code is relatively attractive. It uses the `ZIO#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 lacks 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 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”. 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, each dealing exclusively with one of these concerns.

How do we go about actually defining retry strategies such that we can compose them in this way?

## 24.2 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. 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:

```

1 import zio._
2
3 def delayN(
4   n: Int
5 )(delay: Duration): Schedule[Any, Any, (Long, Long)] =
6   Schedule.recurse(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 `recurve` 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.

```
1 Schedule.recurse(5) && Schedule.spaced(1.second)
```

So, the combined schedule continues five times because after that, the `recurve` schedule does not want to continue, and the composed schedule only continues as long as both schedules want to continue. At each step, it delays for the specified duration `delay` since `recurve` 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.

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:

```

1 trait ZIO[-R, +E, +A] {
2   def repeat[R1 <: R, B](
3     schedule: Schedule[R1, A, B]
4   ): ZIO[R1, E, B]
5
6   def retry[R1 <: R, S](
7     schedule: Schedule[R1, E, S]
8   ): ZIO[R1, E, A]
9 }
```

The `ZIO#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 `ZIO#retryOrElse` and `ZIO#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 `ZIO#repeat` operator is similar to the `ZIO#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.

## 24.3 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.

### 24.3.1 Schedules for Recurrences

One simple family of Schedule constructors describe recurring a specified number of times.

The most useful constructor here is `Schedule.recur`, which creates a schedule that recurs the specified number of times with no delay between recurrences:

```

1 object Schedule {
2     def recur(n: Long): Schedule[Any, Any, Long] =
3         ???
4 }
```

Typically, we will also want some delay in the final schedules we use to repeat or retry effects, but by combining `Schedule.recur` 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 `Schedule.once` constructor that creates a schedule that continues a single time with no delay and a `Schedule.stop` constructor that creates a schedule that is done immediately.

### 24.3.2 Schedules for Delays

For the second family of Schedule constructors, we will discuss 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 `Schedule.spaced` constructor.

```

1 object Schedule {
2     def spaced(duration: Duration): Schedule[Any, Any, Long] =
3         ???
4 }
```

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 `Schedule.fixed` constructor.

```

1 object Schedule {
2     def fixed(interval: Duration): Schedule[Any, Any, Long] =
3         ???
4 }
```

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 repeated or retried effects 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 `Schedule.spaced`. If you want a constant *frequency* of recurrences, use `Schedule.fixed`.

One further variant of schedule constructors that use a constant delay is the `Schedule.windowed` constructor:

```
1 object Schedule {
2     def windowed(interval: Duration): Schedule[Any, Any, Long] =
3         ???
4 }
```

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 `Schedule.linear` constructor:

```

1 object Schedule {
2   def linear(base: Duration): Schedule[Any, Any, Duration] =
3     ???
4 }
```

This constructs a schedule that delays between each recurrence like `Schedule.spaced`, but now the duration of the delays increases 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 of a momentary issue but then retry more slowly if the connection is still unavailable.

Another variant on this theme is the `Schedule.exponential` constructor:

```

1 object Schedule {
2   def exponential(
3     base: Duration,
4     factor: Double = 2.0
5   ): Schedule[Any, Any, Duration] =
6     ???
7 }
```

This creates a schedule that increases the delay between repetitions as the schedule continues like `Schedule.linear`, but now increases the delay exponentially rather

than linearly. So, for example, a schedule constructed with `Schedule.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 `Schedule.exponential(1.second, 3.0)` would initially delay for one second, then three seconds, then nine seconds, and so on.

The `Schedule.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:

```

1 object Schedule {
2     def fibonacci(one: Duration): Schedule[Any, Any, Duration] =
3         ???
4 }
```

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 `Schedule.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 the creation of finite schedules from a custom series rather than infinite schedules from an existing mathematical series.

The simplest of these is the `Schedule.fromDuration` constructor, which creates a schedule that recurs a single time with the specified delay:

```

1 object Schedule {
2     def fromDuration(
3         duration: Duration
4     ): Schedule[Any, Any, Duration] =
5         ???
6 }
```

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 `Schedule.fromDurations` constructor:

```

1 object Schedule {
2     def fromDurations(
```

```

3     duration: Duration,
4     durations: Duration*
5 ): Schedule[Any, Any, Duration] =
6     ???
7 }
```

This allows for the creation of 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 helpful 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.

### 24.3.3 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:

```

1 object Schedule {
2     def recurWhile[A](f: A => Boolean): Schedule[Any, A, A] =
3         ???
4     def recurWhileM[Env, A](
5         f: A => URIO[Env, Boolean]
6     ): Schedule[Env, A, A] =
7         ???
8     def recurWhileEquals[A](a: => A): Schedule[Any, A, A] =
9         ???
10    def recurUntil[A](f: A => Boolean): Schedule[Any, A, A] =
11        ???
12    def recurUntilM[Env, A](
13        f: A => URIO[Env, Boolean]
14    ): Schedule[Env, A, A] =
15        ???
16    def recurUntilEquals[A](a: => A): Schedule[Any, A, A] =
17        ???
18 }
```

The `Schedule.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 it is done immediately as soon as the predicate is false.

There is also a `Schedule.recurWhileZIO` variant that allows performing an effect as part of determining whether the schedule should continue, as well as a simplified `Schedule.recurWhileEquals` constructor that continues as long as the input is equal to the specified value.

Each of these constructors also has a corresponding `Schedule.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.

#### 24.3.4 Schedules for Outputs

The next family of schedule constructors create schedules that output values that are useful in some ways, 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 `Schedule.fromFunction`, which creates a schedule from a function:

```

1 object Schedule {
2   def fromFunction[A, B](f: A => B): Schedule[Any, A, B] =
3     ???
4 }
```

The schedule returned by `Schedule.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 `Schedule.identity` constructor, which is like `Schedule.fromFunction` but constructs a schedule that passes its inputs through unchanged instead of transforming them with a function:

```

1 object Schedule {
2   def fromFunction[A, B](f: A => B): Schedule[Any, A, B] =
3     ???
4   def identity[A]: Schedule[Any, A, A] =
5     fromFunction(a => a)
6 }
```

The `Schedule.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 succeeds with the same type of value as the original effect.

One other variant of the `Schedule.identity` constructor is `Schedule.collectAll`:

```

1 object Schedule {
2   def collectAll[A]: Schedule[Any, A, Chunk[A]] =
3     ???
4 }
```

The `Schedule.collectAll` constructor is like `Schedule.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 `Schedule.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 `Schedule.collectAll` to visualize all the intermediate solutions generated along the way to the final solution.

In addition to the basic `Schedule.collectAll` constructor, there are `Schedule.collectWhile` and `Schedule.collectUntil` variants that continue as long as some predicate is satisfied. You can think of these as a composition of the `Schedule.recurWhile` and `Schedule.recurUntil` schedules we saw above with the `Schedule.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 `Schedule.unfold` constructor, which allows constructing a schedule by repeatedly applying a function to an initial value:

```

1 object Schedule {
2   def unfold[A](a: => A)(f: A => A): Schedule[Any, Any, A] =
3     ???
4 }
```

The schedule returned by `Schedule.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, `Schedule.count`, that just outputs the number of recurrences so far, using `unfold`:

```

1 object Schedule {
2   val count: Schedule[Any, Any, Long] =
3     unfold(0L)(_ + 1L)
4
5   def unfold[A](a: => A)(f: A => A): Schedule[Any, Any, A] =
6     ???
7 }
```

Another even simpler constructor we can implement in terms of `Schedule.unfold` is `Schedule.succeed`, which just returns a schedule that always outputs the same constant value:

```

1 object Schedule {
2   def succeed[A](a: => A): Schedule[Any, Any, A] =
3     unfold(a)(a => a)
4
5   def unfold[A](a: => A)(f: A => A): Schedule[Any, Any, A] =
6     ???
7 }
```

One final variant that is particularly useful is the `Schedule.elapsed` constructor, which returns a new schedule that always recurs with no delay but outputs the time since the first step:

```

1 object Schedule {
2   val elapsed: Schedule[Any, Any, Duration] =
3     ???
4 }
```

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.

#### 24.3.5 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 AM. 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 to describe 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:

```

1 object Schedule {
2   def secondOfMinute(second: Int): Schedule[Any, Any, Long] = ????
3   def minuteOfHour(minute: Int) : Schedule[Any, Any, Long] = ????
4   def hourOfDay(hour: Int)      : Schedule[Any, Any, Long] = ????
5   def dayOfWeek(day: Int)       : Schedule[Any, Any, Long] = ????
6 }
```

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:

```
1 | val automatedReportSchedule =
2 |   Schedule.dayOfWeek(1) && Schedule.hourOfDay(9)
```

Again, we will see more about how to compose schedules like this later in the chapter. However, 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.

## 24.4 Transforming Schedules

At this point, we know all the primary 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.

### 24.4.1 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:

```

1 trait Schedule[-Env, -In, +Out] {
2   def as[Out2](out2: => Out2): Schedule[Env, In, Out2]
3
4   def dimap[In2, Out2](
5     f: In2 => In,
6     g: Out => Out2
7   ): Schedule[Env, In2, Out2]
8
9   def map[Out2](f: Out => Out2): Schedule[Env, In, Out2]
10
11  def contramap[Env1 <: Env, In2](
12    f: In2 => In
13  ): Schedule[Env, In2, Out]
14
15  def mapZIO[Env1 <: Env, Out2](
16    f: Out => URIO[Env1, Out2]
17  ): Schedule[Env1, In, Out2]
18 }

```

The two most basic operators for transforming inputs and outputs are `Schedule#map` and `Schedule#contramap`.

The `Schedule#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 `Schedule#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 `Schedule#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 `Schedule#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 `Schedule#mapZIO` 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.

### 24.4.2 Summarizing Schedule 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:

```

1 trait Schedule[-Env, -In, +Out] {
2   def collectAll: Schedule[Env, In, Chunk[Out]]
3
4   def fold[Z](z: Z)(f: (Z, Out) => Z): Schedule[Env, In, Z]
5
6   def foldZIO[Env1 <: Env, Z](
7     z: Z
8   )(f: (Z, Out) => URIO[Env1, Z]): Schedule[Env1, In, Z]
9
10  def repetitions: Schedule[Env, In, Int]
11 }
```

The most general of these is the `Schedule#foldZIO` 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 `Schedule#foldZIO` operator allows performing an effect within each step of the fold, whereas the simpler `Schedule#fold` operator just uses a normal function.

To get a sense of working with these operators, we can use `fold` to implement the specialized `Schedule#collectAll` and `Schedule#repetitions` operators, which output all the values seen before or the number of repetitions so far, respectively:

```

1 trait Schedule[-Env, -In, +Out] {
2   def fold[Z](z: Z)(f: (Z, Out) => Z): Schedule[Env, In, Z]
3
4   def collectAll: Schedule[Env, In, Chunk[Out]] =
5     fold[Chunk[Out]](Chunk.empty)(_ :+ _)
6
7   def repetitions: Schedule[Env, In, Int] =
8     fold(0)((n, _) => n + 1)
9 }
```

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 starts 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.

### 24.4.3 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:

```

1 trait Schedule[-Env, -In, +Out] {
2   def tapInput[Env1 <: Env, In1 <: In](
3     f: In1 => URIO[Env1, Any]
4   ): Schedule[Env1, In1, Out]
5
6   def tapOutput[Env1 <: Env](
7     f: Out => URIO[Env1, Any]
8   ): Schedule[Env1, In, Out]
9
10  def onDecision[Env1 <: Env](
11    f: Decision[Env, In, Out] => URIO[Env1, Any]
12  ): Schedule[Env1, In, Out]
13}
```

The more basic variants are `Schedule#tapInput` and `Schedule#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 `Schedule#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 `Schedule#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.

### 24.4.4 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 `Schedule#provide` and `Schedule#provideSome`:

```

1 trait Schedule[-Env, -In, +Out] {
2   def provide(env: Env): Schedule[Any, In, Out]
3
4   def provideSome[Env2](f: Env2 => Env): Schedule[Env2, In, Out]
5 }
```

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 an environment 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.

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.

Again, in most cases, that should not be necessary, but if you want to provide some or all of the services that a schedule requires, you can use the `Schedule#provide` and `Schedule#provideSome` operators to do that.

#### 24.4.5 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:

```

1 import zio.Random._

2
3 trait Schedule[-Env, -In, +Out] {
4   def addDelay(f: Out => Duration): Schedule[Env, In, Out]
5
6   def addDelayZIO[Env1 <: Env](
7     f: Out => URIO[Env1, Duration]
8   ): Schedule[Env1, In, Out]
9
10  def delayed(f: Duration => Duration): Schedule[Env, In, Out]
11
12  def delayedZIO[Env1 <: Env](
13    f: Duration => URIO[Env1, Duration]
14  ): Schedule[Env1, In, Out]
15
16  def jittered(
17    min: Double = 0.0,
18    max: Double = 1.0
19  ): Schedule[Env with Random, In, Out]
20
21  def modifyDelay(
22    f: (Out, Duration) => Duration
23  ): Schedule[Env, In, Out]
24
```

```

25 def modifyDelayZIO[Env1 <: Env](
26   f: (Out, Duration) => URIO[Env1, Duration]
27 ): Schedule[Env1, In, Out]
28 }
```

As you can see, there are three primary variants for modifying schedule delays, `Schedule#addDelay`, `Schedule#delayed`, and `Schedule#modifyDelay`, as well as one more specialized variant `Schedule#jittered`. Each of these also has a variant with a ZIO suffix that allows performing an effect as part of the modification function.

The most general operator is `modifyDelay`, which allows a new delay to be returned 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:

```

1 def doubleDelay[Env, In, Out](
2   schedule: Schedule[Env, In, Out]
3 ): Schedule[Env, In, Out] =
4   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 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:

```

1 trait Schedule[-Env, -In, +Out] {
2   def modifyDelay(
3     f: (Out, Duration) => Duration
4   ): Schedule[Env, In, Out]
5
6   def addDelay(f: Out => Duration): Schedule[Env, In, Out] =
7     modifyDelay((out, duration) => f(out) + duration)
8 }
```

The `Schedule#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 `Schedule.unfold` constructor we saw before.

```

1 import zio._
2
3 def fibonacci(one: Duration): Schedule[Any, Any, Duration] =
4   Schedule
5     .unfold((one, one)) { case (a1, a2) =>
6       (a2, a1 + a2)
7     }
8     .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:

```

1 def fibonacci(one: Duration): Schedule[Any, Any, Duration] =
2   Schedule
3     .unfold((one, one)) { case (a1, a2) =>
4       (a2, a1 + a2)
5     }
6     .map(_._1)
7     .addDelay(out => out)

```

The `Schedule#delayed` operator is another variant on this theme. It is like `Schedule#modifyDelay` but only gives access to the original delay for each schedule recurrence instead of the output plus the original delay.

The final, more specialized variant to be aware of is `jittered`. The `Schedule#jittered` operator modifies each schedule delay to be a random value between a `min` and `max` factor of the original delay.

For example, a schedule constructed using `Schedule.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 that modifies each delay to between zero and the original delay, equivalent to `Schedule#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.

#### 24.4.6 Modifying Decisions

The operators in this section are similar to the ones in the last section but allow modifying whether the schedule continues:

```

1 trait Schedule[-Env, -In, +Out] {
2   def check[In1 <: In](
3     test: (In1, Out) => Boolean
4   ): Schedule[Env, In1, Out]
5
6   def checkZIO[Env1 <: Env, In1 <: In](
7     test: (In1, Out) => URIO[Env1, Boolean]
8   ): Schedule[Env1, In1, Out]
9
10  def reconsider[Out2](
11    f: Decision[Env, In, Out] => Either[Out2, (Out2, Interval)]
12  ): Schedule[Env, In, Out2]
13
14  def reconsiderZIO[Env1 <: Env, In1 <: In, Out2](
15    f: Decision[Env, In, Out] => URIO[
16      Env1,
17      Either[Out2, (Out2, Interval)]
18    ]
19  ): Schedule[Env1, In1, Out2]
20 }
```

There are two key variants, `Schedule#check` and `Schedule#reconsider`, each of which also have effectful variants with the ZIO suffix.

The `Schedule#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 `Schedule#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.

#### 24.4.7 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:

```

1 trait Schedule[-Env, -In, +Out] {
2   def ensuring(finalizer: UIO[Any]) : Schedule[Env, In, Out]
3   def forever                  : Schedule[Env, In, Out]
4   def resetAfter(duration: Duration): Schedule[Env, In, Out]
5   def resetWhen(f: Out => Boolean) : Schedule[Env, In, Out]
6 }
```

The first of these operators is `Schedule#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 `acquireRelease` operators on ZIO itself instead of the `ensuring` operator on `Schedule`.

The next operator, `Schedule#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 was 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 `Schedule#resetAfter` and `Schedule#resetWhen` operators are variants on this idea of resetting the state of the schedule.

Many times, a schedule will maintain some internal state. For example, the schedule created by `Schedule.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.

## 24.5 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 ways reflects the logic of both of the first two schedules.

The `Schedule` data type actually supports several different ways to compose schedules.

### 24.5.1 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 `Schedule#intersectWith` and `Schedule#unionWith`:

```

1 trait Schedule[-Env, -In, +Out] { self =>
2   def intersectWith[Env1 <: Env, In1 <: In, Out2](
3     that: Schedule[Env1, In1, Out2]
4   )(
5     f: (Interval, Interval) => Interval
6   ): Schedule[Env1, In1, (Out, Out2)]
7
8   def unionWith[Env1 <: Env, In1 <: In, Out2](
9     that: Schedule[Env1, In1, Out2]
10    )(
11      f: (Interval, Interval) => Interval
12    ): Schedule[Env1, In1, (Out, Out2)]
13 }
```

The `Schedule#intersectWith` and `Schedule#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 `Schedule#intersectWith` and `Schedule#unionWith` operators differ in the *continue logic* they apply.

The `Schedule#intersectWith` operator uses the logic “continue only as long as both schedules want to continue”. In contrast, the `Schedule#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:

```

1 trait Schedule[-Env, -In, +Out] { self =>
2   def intersectWith[Env1 <: Env, In1 <: In, Out2](
3     that: Schedule[Env1, In1, Out2]
4   )( 
5     f: (Interval, Interval) => Interval
6   ): Schedule[Env1, In1, (Out, Out2)]
7 
8   def unionWith[Env1 <: Env, In1 <: In, Out2](
9     that: Schedule[Env1, In1, Out2]
10  )( 
11    f: (Interval, Interval) => Interval
12  ): Schedule[Env1, In1, (Out, Out2)]
13 
14  def &&[Env1 <: Env, In1 <: In, Out2](
15    that: Schedule[Env1, In1, Out2]
16  ): Schedule[Env1, In1, (Out, Out2)] =
17    (self intersectWith that)(_ max _)
18 
19  def ||[Env1 <: Env, In1 <: In, Out2](
20    that: Schedule[Env1, In1, Out2]
21  ): Schedule[Env1, In1, (Out, Out2)] =
22    (self unionWith that)(_ min _)
23 }
```

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.

Now that we have a better understanding of `Schedule` and schedule constructors, let's go back to the example from the beginning of this chapter. We would like to implement a schedule that recurs five times with a delay of one second between each recurrence.

We already have the `Schedule.spaced` constructor, which constructs a schedule that recurs forever with a specified delay between each recurrence, and the `Schedule.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 `Schedule.spaced` schedule will always continue, but the `Schedule.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 `Schedule.spaced` schedule will always delay for one second while the `Schedule.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 `Schedule.spaced` schedule always wants to continue, this would mean the composed schedule would continue forever.

The schedule composed with the `||` operator will also delay for the *minimum* of the length of the delays from each of the original schedules.

The `Schedule.recurs` schedule recurs five times with no delay, so the first five recurrences would have no delay. After that, only the `Schedule.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 of the different ways to compose schedules.

You can use the `Schedule#intersectWith` and `Schedule#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 `Schedule#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:

```

1 trait Schedule[-Env, -In, +Out] { self =>
2   def &&[Env1 <: Env, In1 <: In, Out2](
3     that: Schedule[Env1, In1, Out2]
4   ): Schedule[Env1, In1, (Out, Out2)]
5
6   def map[Out2](f: Out => Out2): Schedule[Env, In, Out2]
7
8   def zipWith[Env1 <: Env, In1 <: In, Out2, Out3](
9     that: Schedule[Env1, In1, Out2]
10    )(f: (Out, Out2) => Out3): Schedule[Env1, In1, Out3] =
11     (self && that).map(f.tupled)
12 }
```

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:

```

1 def delayN(
2   n: Int
3 )(delay: Duration): Schedule[Any, Any, (Long, Long)] =
4   Schedule.recurse(n) && Schedule.spaced(delay)
```

One thing that is slightly awkward about this is the output type of `(Long, Long)`. Both `recurse` 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:

```
1 def delayN(n: Int)(delay: Duration): Schedule[Any, Any, Long] =  
2   Schedule.recurse(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.

### 24.5.2 Sequential Composition of Schedules

The second way of composing schedules is the 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 `Schedule#andThenEither` operator:

```
1 trait Schedule[-Env, -In, +Out] { self =>  
2   def andThenEither[Env1 <: Env, In1 <: In, Out2] (  
3     that: Schedule[Env1, In1, Out2]  
4   ): Schedule[Env1, In1, Either[Out, Out2]]  
5 }
```

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 `Schedule#andThen` operator can be used to automatically unify the output types:

```

1 trait Schedule[-Env, -In, +Out] {
2   def andThen[Env1 <: Env, In1 <: In, Out2 >: Out](
3     that: Schedule[Env1, In1, Out2]
4   ): Schedule[Env1, In1, Out2]
5 }
```

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 follows:

```

1 val schedule1: Schedule[Any, Any, Long] =
2   Schedule.spaced(1.second) *> Schedule.recur(5)
```

Similarly, we can implement the second part of the schedule as follows:

```

1 val schedule2: Schedule[Any, Any, Duration] =
2   Schedule.exponential(1.second) <*> Schedule.recur(10)
```

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:

```

1 val schedule3: Schedule[Any, Any, Duration] =
2   schedule1.as(1.second) ++ schedule2
```

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.

### 24.5.3 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 `+++`:

```

1 trait Schedule[-Env, -In, +Out] { self =>
2   def +++[Env1 <: Env, In2, Out2](
3     that: Schedule[Env1, In2, Out2]
4   ): Schedule[Env1, Either[In, In2], Either[Out, Out2]]
5 }
```

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 are the same, and they can be simply unified. 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:

```

1 trait Schedule[-Env, -In, +Out] { self =>
2   def |||[Env1 <: Env, Out1 >: Out, In2](
3     that: Schedule[Env1, In2, Out1]
4   ): Schedule[Env1, Either[In, In2], Out1]
5 }
```

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:

```

1 sealed trait WebServiceError
2
3 trait ConnectionError extends WebServiceError
4 trait RateLimitError extends WebServiceError
5
6 lazy val schedule1: Schedule[Any, ConnectionError, Duration] =
7   ???
8
9 lazy val schedule2: Schedule[Any, RateLimitError, Duration] =
10  ???
11
12 lazy val schedule3: Schedule[Any, WebServiceError, Duration] =
13   (schedule1 ||| schedule2).contramap {
14     case connectionError: ConnectionError => Left(connectionError)
15     case rateLimitError: RateLimitError    => Right(rateLimitError)
16   }

```

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.

#### 24.5.4 Function Composition of Schedules

The fourth way of composing schedules is the 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 `>>`:

```

1 trait Schedule[-Env, -In, +Out] {
2   def >>[Env1 <: Env, Out2] (
3     that: Schedule[Env1, Out, Out2]
4   ): Schedule[Env1, In, Out2]
5 }

```

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.

## 24.6 Implementation of Schedule

The last section of this chapter describes the internal implementation of `Schedule`.

The `Schedule` data type is designed so that you can use it to solve your own problems in implementing composable retry and repeat strategies for effects without needing to understand its implementation. 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 to the conclusion and exercises at the end of this chapter.

A very slightly simplified implementation of `Schedule` is as follows:

```
1 import java.time.OffsetDateTime
2
3 trait Schedule[-Env, -In, +Out] {
4     type State
5     val initial: State
6     def step(
7         now: OffsetDateTime,
8         in: In,
9         state: State
10    ): ZIO[Env, Nothing, (State, Out, Decision)]
11 }
12
13 sealed trait Decision
14
15 object Decision {
16     final case class Continue(interval: OffsetDateTime)
17         extends Decision
18     case object Done extends Decision
```

<sup>19</sup> }

A `Schedule` is defined in terms of `step`, which is just a function that takes an initial state `State`, an input `In` and an `OffsetDateTime`, representing the current time, and returns a `ZIO` effect producing a `Decision`.

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 `step` is an updated `State`, the current summary output value, and 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. 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 next interval during which the schedule is willing to recur.

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 `step` function in the `Schedule` and get its result
3. If the decision of the `step` function is done, then return the original failure, we have no more recurrences left
4. If the decision of the `step` function is to continue, then delay for the specified time and then repeat at the first step, using the new state produced by the decision to continue instead of the current one

`ZIO` abstracts much of this logic in a concept called the `Schedule.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`:

```

1 final case class Driver[-Env, -In, +Out] (
2   next: In => ZIO[Env, None.type, Out],
3   last: IO[NoSuchElementException, Out],
4   reset: UIO[Unit]
5 )

```

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 `Schedule.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.

## 24.7 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.

## 24.8 Exercises

1. Create a schedule that first attempts 3 quick retries with a delay of 500 milliseconds. If those fail, switch to exponential backoff for 5 more attempts, starting at 2 seconds and doubling each time.
2. Create a schedule that only retries during "business hours" (9 AM to 5 PM) on weekdays, with a one-hour delay between attempts.

3. Create a progressive jittered schedule that delays between each recurrence and increases the jitter percentage by 5 percent as the number of retries increases.
4. We want to call an API that has a rate limit of 100 requests per hour. Create a schedule that respects this rate limit, only recursing 100 times, and resets its retry count at the start of each hour.
5. We have an API that, when we flood it with requests, starts to return the following error:

```

1 case class RateLimitExceeded(
2   retryAfter: Duration,
3   remainingQuota: Int
4 )
5
6 def apiCall: IO[RateLimitExceeded, Unit] = ???
```

Write a schedule that retries the API call with respect to the `retryAfter` duration so it doesn't perform any requests until the `retryAfter` duration has elapsed.

6. Create a schedule for IoT devices that adjusts its polling frequency based on temperature changes:
  - Poll every 5 minutes when the temperature is stable (change < 3°C in the last 5 minutes)
  - Poll every 1 second when the temperature is unstable (change >= 3°C)
  - Return to stable polling once the temperature stabilizes again
7. Write a cron-like schedule that takes a set of seconds of the minute, minutes of the hour, hours of the day, and days of the week and returns a schedule that recurs at those times:

```

1 def cronSchedule[Env, In](
2   seconds0fMinute: Set[Int],
3   minutes0fHours: Set[Int],
4   hours0fDay: Set[Int],
5   days0fWeek: Set[Int]
6 ): Schedule[Env, Int, Long] = ???
```

# Chapter 25

## Advanced Error Management: Debugging

Debugging is a critical skill for any software developer, playing a key role in ensuring the reliability and correctness of applications. In the world of functional programming, especially when working with effectful systems like those built with ZIO, debugging can be more challenging due to the nature of the effects—particularly for newcomers.

Many beginners find it challenging to debug ZIO applications using breakpoints and stepping through the code, as the execution flow is not as straightforward as in traditional, synchronous programs. This chapter provides a comprehensive guide to effectively debugging ZIO applications, covering the tools, techniques, and best practices to help you diagnose and resolve issues in your ZIO code.

### 25.1 Understanding Execution Flows

When debugging ZIO applications, it is crucial to understand the flow of execution and how it differs from traditional synchronous code. In a synchronous program, the execution flow is linear, making it easy to follow by stepping through the code with breakpoints. However, in an asynchronous program, the flow is non-linear, and ZIO adds another layer of complexity: it is a functional effect system. This means that ZIO programs are descriptions of computations, not the computations themselves. Therefore, an additional component, the ZIO runtime system, is required to interpret and execute these descriptions.

The ZIO runtime system reads the program descriptions you have written and executes them in a non-blocking, asynchronous manner. This can cause the execution flow to jump between different fibers, yielding control to others, making it difficult to trace the execution path. As a result, traditional debugging techniques like setting breakpoints and stepping through code line-by-line in an IDE may not be as effective as they are in synchronous programming. It's common to see the execution flow move into the ZIO runtime system

and then back into your code, which can be confusing, especially for those new to ZIO.

Given these challenges, we do not recommend relying on breakpoints to debug ZIO applications. Instead, we suggest using tools and techniques that are better suited to asynchronous and effectful programs.

## 25.2 Printing Debug Information

One simple yet effective method for debugging ZIO applications is to print debug information to the console. This allows you to observe the flow of execution and the state of your program over time. For example, consider a simple ZIO program that computes the factorial of a number:

```
1 def factorial(n: Int): UIO[Int] =
2   if (n == 0) ZIO.succeed(1)
3   else factorial(n - 1).map(_ * n)
```

To debug this program, you can add print statements to print the value of `n` at each step of the computation:

```
1 def factorial(n: Int): UIO[Int] =
2 ZIO.succeed(println(s"Computing factorial($n)")) *> {
3   if (n == 0)
4     ZIO.succeed(1).flatMap { res =>
5       ZIO.succeed(println(s"Computed factorial(0): $res")).as(res)
6     }
7   else
8     factorial(n - 1)
9     .map(_ * n)
10    .flatMap(res => ZIO.succeed(println(s"Computed factorial($n): $res"))).as(res))
11 }
```

Let's run this program with a value of 5:

```
1 Computing factorial(5)
2 Computing factorial(4)
3 Computing factorial(3)
4 Computing factorial(2)
5 Computing factorial(1)
6 Computing factorial(0)
7 Computed factorial(0): 1
8 Computed factorial(1): 1
9 Computed factorial(2): 2
10 Computed factorial(3): 6
11 Computed factorial(4): 24
12 Computed factorial(5): 120
```

ZIO provides a useful method called `ZIO.debug`, which is a ZIO effect that takes a message and prints it to the console. Additionally, there is the `ZIO#debug` method, which taps into a ZIO effect and prints the value of the effect to the console without altering the effect's returned value:

```

1 def factorial(n: Int): UIO[Int] =
2   ZIO.debug(s"Computing factorial($n)") *> {
3     if (n == 0)
4       ZIO.succeed(1).debug("Computed factorial(0)")
5     else
6       factorial(n - 1)
7         .map(_ * n).debug(s"Computed factorial($n)")
8   }

```

## 25.3 Enabling Diagnostic and Debug Logging

Debugging with print statements is common in development environments. However, there are times when a bug is reported that we cannot reproduce in our local development environment. In such cases, we need to record detailed information about the application's execution in a production-like environment to diagnose the issue.

One effective method for debugging such issues is by adding debug logging to the application and deploying it in an environment that closely resembles production, where the bug was originally reported. This approach allows us to run the application over time, reproduce the bug, and analyze the logs to pinpoint the root cause.

Debug logs are more verbose than regular logs and are not typically suitable for production environments. Because debug logging can generate a significant amount of data, it is usually set at the lowest logging level and used selectively—either activated temporarily or filtered to specific areas of the application that require investigation.

ZIO allows you to log messages at various levels, including “DEBUG” and “TRACE,” which are especially useful for debugging. This can be done using the `ZIO.logDebug` and `ZIO.logTrace` methods, respectively. Below is an example of a factorial program with logging:

```

1 def factorial(n: Int): UIO[Int] =
2   ZIO.logDebug(s"Computing factorial($n)") *> {
3     if (n == 0)
4       ZIO.succeed(1).tap(res => ZIO.logDebug(s"Computed factorial
5           (0): $res"))
6     else
7       factorial(n - 1)
8         .map(_ * n)
9         .tap(res => ZIO.logError(s"Computed factorial($n): $res"))
10    )
11  }

```

ZIO provides a built-in logging facade that allows integration with various logging backends, such as log4j, slf4j, or logback. Using the ZIO Logging library lets you connect ZIO with your preferred logging backend. Once your ZIO application is ready for deployment to testing or staging environments, you can adjust the logger's log level to "DEBUG" or "TRACE" to generate more detailed logs. Monitoring these logs will help you gather clues and diagnose the root cause of the reported bug.

## 25.4 Reading Stack Traces

Stack traces are essential for debugging applications, offering a detailed view of a program's execution path when errors or exceptions occur. ZIO, a functional effect system with its own runtime, has a non-linear execution flow that can complicate stack traces. To address this, ZIO uses a specialized stack trace format to keep stack traces concise and relevant by filtering out internal operations.

When an error occurs in a ZIO program, the stack trace is captured and presented in a structured format that includes critical debugging information. This format details the failed ZIO effect, including the fiber ID, the cause of the failure, the associated stack trace, and the source code location of the calls to the effect, making it easier to identify and resolve issues.

To demonstrate this, let's consider a simple ZIO program that fails with an error:

```

1 import zio._

2
3 object StackTraceDemo extends ZIOAppDefault {
4     def foo(n: Int) =
5         bar(n) *> ZIO.debug("returned from bar")
6
7     def bar(n: Int) =
8         baz(n) *> ZIO.debug("returned from baz")
9
10    def baz(n: Int) = {
11        val _ = n
12        ZIO.fail("Boom!")
13    }
14
15    def run =
16        foo(5).ensuring {
17            ZIO.debug("Trying to recover from failure but ...") *>
18            ZIO.dieMessage("... it's too late! Boom!")
19        } *> ZIO.debug("done!")
20 }
```

ZIO will capture and print the stack trace as follows:

```

1 Trying to recover from failure but ...
```

```

1 timestamp=2024-09-03T13:05:43.673879Z level=ERROR thread=#zio-
2   fiber-68068046 message="" cause="Exception in thread \"zio-
3     fiber-1540396744\" java.lang.String: Boom!
4     at <empty>.StackTraceDemo.baz(StackTraceDemo.scala:12)
5     at <empty>.StackTraceDemo.bar(StackTraceDemo.scala:8)
6     at <empty>.StackTraceDemo.foo(StackTraceDemo.scala:5)
7     at <empty>.StackTraceDemo.run(StackTraceDemo.scala:16)
8     at <empty>.StackTraceDemo.run(StackTraceDemo.scala:19)
9     Suppressed: java.lang.RuntimeException: ... it's too late!
10    Boom!
11    at <empty>.StackTraceDemo.run(StackTraceDemo.scala:18)
12    at <empty>.StackTraceDemo.run(StackTraceDemo.scala:16)
13    at <empty>.StackTraceDemo.run(StackTraceDemo.scala:19)"

```

If you are a library author and want to avoid exposing the internal stack traces of your library, you should add an implicit trace parameter of type `Trace` to your methods. This approach helps prevent the internal details of your library from appearing in stack traces:

```

1 import zio._
2
3 object FooLibrary {
4   def foo(n: Int)(implicit trace: Trace) = ???
5 }

```

## 25.5 Fiber Dumps

Debugging concurrent programs can be particularly challenging due to the complex interplay of multiple fibers. When things go wrong—e.g., when your application experiences performance degradation or you encounter a deadlock—gaining insight into the state of your program becomes crucial.

For example, assume you have written a ZIO program that is stuck and won't progress further. One possible reason for this behavior is a deadlock, where two or more fibers are waiting indefinitely for resources held by each other. To demonstrate this, consider the following example:

```

1 import zio._
2
3 object DeadlockDemo extends ZIOAppDefault {
4   def run = for {
5     latchA <- Promise.make[Nothing, Unit]
6     latchB <- Promise.make[Nothing, Unit]
7     f1 <- (latchA.await *> latchB.succeed(())).fork
8     f2 <- (latchB.await *> latchA.succeed(())).fork
9     _ <- (f1 <*> f2).await
10   } yield ()
11 }

```

If we run the above program, it will hang indefinitely. In these scenarios, fiber dumps emerge as valuable tools for diagnosing issues. A fiber dump provides a snapshot of the current state of all active fibers within your application, including the stack, state, and execution context of each fiber. By examining a fiber dump, you can gain insights into the state of your application and identify issues like deadlocks or other concurrency-related bugs.

To inspect the state of fibers in a ZIO application, you can use the `Fiber.dumpAll` method, which prints the state of all active fibers to the console. This method is particularly useful when you want to inspect the state of fibers programmatically.

However, the ZIO runtime system also has built-in signal handlers that can be triggered by sending operating system signals, such as `SIGINT` in Windows or `SIGINFO` and `SIGUSR1` in Unix-like systems, to the running application:

```
1 > jps
2 2121942 sbt-launch.jar
3 4125602 DeadLockDemo
4 4176801 Launcher
5 3601481 Main
```

Assume you are running on a Unix-like system; you can send the `SIGUSR1` signal to the running application using the `kill` command:

```
1 kill -USR1 4125602
```

The signal handler will dump the state of all active fibers to the console:

```
1 "zio-fiber-23976402" (26s 457ms) waiting on #2037000743
2   Status: Suspended((Interruption, CooperativeYielding,
3     FiberRoots), <empty>.DeadlockDemo.run(DeadlockDemo.scala:9))
4     at <empty>.DeadlockDemo.run(DeadlockDemo.scala:9)

5 "zio-fiber-2037000743" (26s 547ms) waiting on #23976402
6   Status: Suspended((Interruption, CooperativeYielding,
7     FiberRoots), <empty>.DeadlockDemo.run(DeadlockDemo.scala:7))
8     at <empty>.DeadlockDemo.run(DeadlockDemo.scala:7)

9 "zio-fiber-926126749" (26s 547ms) waiting on #23976402
10   Status: Suspended((Interruption, CooperativeYielding,
11     FiberRoots), <empty>.DeadlockDemo.run(DeadlockDemo.scala:8))
12     at <empty>.DeadlockDemo.run(DeadlockDemo.scala:8)

13 "zio-fiber-46080767" (26s 743ms) waiting on #1226472550
14   Status: Suspended((CooperativeYielding, FiberRoots), <no
15     trace>)

16 "zio-fiber-1226472550" (26s 713ms)
```

```
18 Status: Suspended((Interruption, CooperativeYielding,
FiberRoots), <no trace>)
```

This fiber dump explains a deadlock situation caused by a circular dependency between two fibers: `zio-fiber-23976402` and `zio-fiber-2037000743`, where each fiber is waiting on the other to complete, creating a classic deadlock scenario.

A fiber can be in one of the following states: - **Running**: The fiber is actively executing. - **Suspended**: The fiber is waiting for another to complete. When we call `Fiber#join` on a fiber, the calling fiber will be suspended until the joined fiber completes. - **Done**: The fiber has completed its execution.

## 25.6 Supervising Fibers

Fiber dumps are particularly useful when you need a high-level overview of the state of all fibers in your application. But what if you want to monitor the state of a specific fiber or group of fibers during the lifecycle of their executions in more granular detail? This is where fiber supervision becomes essential.

Supervising fibers allows you to observe their state transitions, such as when they start, suspend, resume, or end. This gives you deeper insights into the lifecycle of fibers and enables you to track the progress of your application, generate performance metrics, and profile the execution of your ZIO application.

To achieve this, you can introduce a `Supervisor` to a ZIO effect using the `ZIO#supervised` method. All fibers forked within the supervised effect will be monitored by this supervisor.

Let's look at a simple example to demonstrate fiber supervision. Suppose you want to generate a metric that tracks the total number of active fibers in your application. To achieve this, you can define a supervisor of type `Int`, representing the count of active fibers:

```
1 import zio._

2

3 val fiberCounter: Supervisor[Int] = new Supervisor[Int] {
4   val fibers = new java.util.HashSet[Int]()
5
6   def value(implicit trace: Trace): UIO[Int] =
7     ZIO.succeed(fibers.synchronized(fibers.size()))
8
9   def onStart[R, E, A](
10     environment: ZEnvironment[R],
11     effect: ZIO[R, E, A],
12     parent: Option[Fiber.Runtime[Any, Any]],
13     fiber: Fiber.Runtime[E, A]
14   )(implicit unsafe: Unsafe): Unit =
15     fibers.synchronized(fibers.add(fiber.id.id)).asInstanceOf[Unit]
```

```

16
17 def onEnd[R, E, A](
18   value: Exit[E, A],
19   fiber: Fiber.Runtime[E, A]
20 )(implicit unsafe: Unsafe): Unit =
21   fibers.synchronized(fibers.remove(fiber.id.id)).asInstanceOf[
22     Unit]

```

In this example, the `onStart` and `onEnd` methods are invoked when a fiber starts and ends, respectively. The `value` method returns the current number of active fibers, using a `java.util.HashSet` to track the IDs of these fibers.

Now, consider a time-consuming computation that calculates the fibonacci number for a given input:

```

1 import zio._
2 import zio.Fiber._
3
4 def fib(n: Int): ZIO[Any, Nothing, Int] =
5   if (n <= 1) {
6     ZIO.succeed(1)
7   } else {
8     for {
9       // add some delay to simulate a long-running computation
10      _           <- ZIO.sleep(500.milliseconds)
11      fiber1 <- fib(n - 2).fork
12      fiber2 <- fib(n - 1).fork
13      v2     <- fiber2.join
14      v1     <- fiber1.join
15    } yield v1 + v2
16 }

```

After defining the supervisor, you can use it to supervise the fibers of this effect by applying the `ZIO#supervised` method and passing the supervisor:

```

1 def fiberStats(supervisor: Supervisor[Int]) = for {
2   length <- supervisor.value
3   _           <- ZIO.debug(s"total running fibers: $length")
4 } yield ()
5
6 for {
7   fiber <- fib(5).supervised(fiberCounter).fork
8   stats <- fiberStats(fiberCounter)
9   .repeat(
10     Schedule
11     .spaced(500.milliseconds)

```

```

12     .whileInputZIO[Any, Unit](_ => fiber.status.map(_ != Status.Done))
13   )
14   .fork
15   _ <- stats.join
16   result <- fiber.join
17   _ <- ZIO.debug(s"fibonacci result: $result")
18 } yield ()

```

The Supervisor also has `onSuspend`, `onResume`, and `onEffect` methods that can be used to monitor the suspension, resumption, and execution of effects within fibers, respectively.

The `onEffect` method is particularly useful for tracking the execution of underlying operations within a fiber, which is disabled by default. To enable this feature, you can use the `Runtime.enableOpSupervision` layer by overriding the `bootstrap` layer of the ZIO application. Please note that enabling operation supervision can negatively impact the performance of your application, so it should be used with caution and only when necessary in the debugging phase, such as for profiling the execution of specific ZIO effect:

```

1 import zio._
2
3 val operationSupervision = new Supervisor[Unit] {
4   def value(implicit trace: Trace): ZIO[Any, Nothing, Unit] =
5     ZIO.succeed(())
6
7   override def onEffect[E, A](
8     fiber: Fiber.Runtime[E, A],
9     effect: ZIO[_, _, _]
10 )(implicit unsafe: Unsafe): Unit =
11   println(s"fiber ${fiber.id.id} is executing effect: ${effect}")
12
13   def onStart[R, E, A](
14     environment: ZEnvironment[R],
15     effect: ZIO[R, E, A],
16     parent: Option[Fiber.Runtime[Any, Any]],
17     fiber: Fiber.Runtime[E, A]
18 )(implicit unsafe: Unsafe): Unit = ()
19
20   def onEnd[R, E, A](
21     value: Exit[E, A],
22     fiber: Fiber.Runtime[E, A]
23 )(implicit unsafe: Unsafe): Unit = ()
24 }

```

Now let's analyze the execution of a simple ZIO effect `ZIO.succeed(42).map(_ *`

42):

```

1 import zio._
2
3 object OperationSupervisionDemo extends ZIOAppDefault {
4   override val bootstrap =
5     Runtime.enableOpSupervision
6
7   def run =
8     ZIO
9     .succeed(42)
10    .map(_ * 2)
11    .supervised(operationSupervision)
12 }
```

This will print the following output:

```

1 fiber 1388727930 is executing effect: DynamicNoBox(<empty>).
  OperationSupervisionDemo.run(OperationSupervisionDemo.scala
  :10),1,zio.ZIO$$$Lambda$82/0x000000080017d040@5b751a18)
2 fiber 1388727930 is executing effect: FoldZIO(<empty>).
  OperationSupervisionDemo.run(OperationSupervisionDemo.scala
  :10),DynamicNoBox(<empty>.OperationSupervisionDemo.run(
  OperationSupervisionDemo.scala:10),4294967297,zio.
  ZIO$InterruptibilityRestorer$MakeInterruptible$$$Lambda$95/0
  x00000008001b9c40@282bf212),zio.ZIO$$$Lambda$110/0
  x00000008001c9040@b9c6a87,zio.ZIO$$$Lambda$108/0
  x00000008001c8040@4ef4457a)
3 fiber 1388727930 is executing effect: DynamicNoBox(<empty>).
  OperationSupervisionDemo.run(OperationSupervisionDemo.scala
  :10),4294967297,zio.
  ZIO$InterruptibilityRestorer$MakeInterruptible$$$Lambda$95/0
  x00000008001b9c40@282bf212)
4 fiber 1388727930 is executing effect: FlatMap(<empty>).
  OperationSupervisionDemo.run(OperationSupervisionDemo.scala
  :9),Sync(<empty>.OperationSupervisionDemo.run(
  OperationSupervisionDemo.scala:8),
  OperationSupervisionDemo$$$Lambda$348/0x000000080027c840@1
  ade437d),zio.ZIO$$$Lambda$74/0x000000080016a840@6d128956)
5 fiber 1388727930 is executing effect: Sync(<empty>).
  OperationSupervisionDemo.run(OperationSupervisionDemo.scala
  :8),OperationSupervisionDemo$$$Lambda$348/0x000000080027c840@1
  ade437d)
6 fiber 1388727930 is executing effect: Sync(<empty>).
  OperationSupervisionDemo.run(OperationSupervisionDemo.scala
  :9),zio.ZIO$$$Lambda$127/0x00000008001d7040@5320e21)
7 fiber 1388727930 is executing effect: Success(84)
```

Supervising only a specific fiber helps avoid overwhelming the application with excessive monitoring operations. This allows you to focus on the sections that require closer attention while leaving the rest of the application unaffected.

However, if you need to monitor all fibers in your application, including those created internally by ZIO, you can use the `Runtime.addSupervisor` method when bootstrapping the ZIO application:

```
1 object MainApp extends ZIOAppDefault {
2     val mySupervisor: Supervisor[Any] = ???
3
4     override val bootstrap =
5         Runtime.addSupervisor(mySupervisor)
6
7     def run = ???
8 }
```

This approach ensures that your supervisor is applied across all fibers, providing comprehensive monitoring for your application.

## 25.7 Conclusion

Debugging ZIO applications presents unique challenges due to the asynchronous execution and effectful nature of functional programming. Traditional methods like breakpoints and code stepping may not work as expected, given ZIO's complex execution flow managed by its runtime system. However, a solid grasp of ZIO's execution dynamics and proper debugging techniques can significantly aid in troubleshooting and fixing issues.

Effective debugging strategies include printing debug information, using diagnostic logging, analyzing ZIO-specific stack traces, and employing fiber dumps to examine the state of active fibers in concurrent programs. These methods offer clear insights into the application's behavior, helping developers to pinpoint and resolve problems more efficiently. By adopting these tools and practices, you can enhance the stability and performance of your ZIO applications.

## Chapter 26

# Advanced Error Management: Best Practices

This chapter will discuss some best practices for error management when developing ZIO applications.

In this chapter, we want to examine error management in ZIO applications more deeply. To better understand best practices, we need to understand the different types of errors that can occur in an application.

From the perspective of error recovery, errors can be categorized into two types: recoverable and non-recoverable. Recoverable errors, also known as expected errors, include cases like validation failures, incorrect inputs, or resources not found. On the other hand, non-recoverable errors are unexpected and usually stem from unforeseen situations that the developer did not anticipate.

Let's explore each of these categories in more detail.

### 26.1 Recoverable Errors

Recoverable errors are those that the application can handle, allowing it to continue running. For example, if a user enters an invalid email address, the application can prompt the user to provide a valid one. The application anticipates and manages these errors. Common examples of recoverable errors include:

- Invalid user input (e.g., incorrect email format).
- Resource not found (e.g., file not found).
- Insufficient permissions to access a resource.

These mistakes are commonly known as typed errors or expected errors. In ZIO, they are typically encoded in the error channel of an effect, such as `ZIO[Any, IncorrectEmailFormat, User]`. In domain-driven design, they are considered

“domain errors” and should generally be handled at the same layer in which they occur.

Handling recoverable errors involves various actions, such as providing clear feedback, guiding users to correct the input, implementing retry mechanisms for temporary issues like network connectivity problems, and offering alternative solutions to achieve the same outcome.

## 26.2 Non-recoverable Errors

Non-recoverable errors typically indicate unexpected problems outside the scope of domain errors. They can be divided into two categories: defects and fatal errors.

- **Defects** are errors that we have no way to handle, but we can still ignore them and safely continue the program execution.
- **Fatal errors**, on the other hand, are the most severe types of errors that cause the program to terminate immediately.

Let’s explore each of these categories in more detail.

### 26.2.1 Defects

Defects are errors that we do not expect to occur as they are not part of the domain errors. While we are not ready to handle them in the application, we can still continue the program execution safely.

Most of the time, stems from bugs in the application code, often due to incorrect assumptions. During development, we may discover that our domain model does not account for all domain errors, leading to unexpected errors. We can strengthen our domain model by fixing these bugs and reclassifying the defects as expected errors, making it capable of handling a more comprehensive range of domain-specific errors.

Another source of defects stems from improper integration with legacy services or libraries. For instance, consider a scenario where you import a legacy method into your ZIO application and expect it not to throw any exceptions, as no such behavior is documented in the API. For example:

```
1 object LegacyUserService {  
2     // This method throws a DatabaseConnectionException exception,  
3     // while it is not documented anywhere  
4     def registerLogic(username: String): Unit = ???  
5 }
```

So you use `ZIO.succeed(LegacyUserService.registerLogic(username))` to integrate it into your application:

```
1 import zio._  
2  
3 case class UserAlreadyExists(username: String) extends  
    Throwable
```

```

4 case class InvalidUsernameFormat(username: String) extends
5   Throwable
6
7 object UserService {
8   /**
9    * Registers a new user with the given username.
10   * Returns a `Task` that succeeds with a unit if the user is
11   * successfully registered.
12   * Otherwise, it fails with an `InvalidUsernameFormat` error if
13   * the username is invalid.
14   * or it fails with a `UserAlreadyExists` error if the user
15   * already exists.
16   */
17   def register(username: String): Task[Unit] =
18     if (validateUsername(username))
19       ZIO.fail(InvalidUsernameFormat(username))
20     else if (userExist(username))
21       ZIO.fail(UserAlreadyExists(username))
22     else
23       ZIO.succeed(LegacyUserService.registerLogic(username))
24 }
```

Now we can use the `UserService.register` method and use the `catchSome` operator to handle the expected errors and provide proper feedback to the user:

```

1 val app: ZIO[Any, Throwable, Unit] =
2   UserService.register("john").catchSome {
3     case UserAlreadyExists(username) =>
4       Console.printLine(s"User $username already exists, please
5         choose another username").orDie
6     case InvalidUsernameFormat(username) =>
7       Console.printLine(s"Invalid username format for $username")
8         .orDie
9   }
```

But what happens if the legacy method `registerLogic` throws an exception, e.g., `DatabaseConnectionException`? This exception is not documented in the API, and we are not prepared to handle it. As we imported the method using `ZIO.succeed`, the exception will convert to a defect and propagate up the call stack until causing the application to crash. To prevent the application from crashing, we can catch all defects by using `app.catchAllDefect(_ => ZIO.unit)`.

### 26.2.1.1 Log Defects for Further Investigation

A common mistake developers make is trying to catch all defects solely to prevent the application from crashing. This is not a good practice because defects often indicate deeper

issues within the application. By catching and ignoring defects, we only address the symptoms, not the root cause.

It is better to catch defects and log detailed information about the error and its context for later analysis and debugging. This helps us understand the root cause of the defect and fix it:

```

1 val app: ZIO[Any, Throwable, Unit] =
2   UserService
3     .register("john")
4     .catchSome {
5       case UserAlreadyExists(username) =>
6         Console.printLine(s"User $username already exists, please
7                           choose another username").orDie
8       case InvalidUsernameFormat(username) =>
9         Console.printLine(s"Invalid username format for $username
10                         ").orDie
11     }
12     .catchAllDefect { defect =>
13       ZIO.logErrorCause(
14         "An unexpected error occurred while registering the user
15           ",
16         Cause.die(defect)
17       )
18     }
19 }
```

As defects are unexpected errors, logging detailed information about the defect and its context using the `ZIO#logErrorCause` operator helps you to capture the errors and their context for later analysis and debugging. This allows you to understand the root cause of the defect and fix it.

However, this approach can lead to another common mistake: swallowing defects after logging them!

### 26.2.1.2 Don't Swallow Defects in the First Place

You shouldn't catch defects just for logging purposes, as higher application layers might handle them before reaching the system's edge. So, a better approach is to catch all defects, log the details, and allow the defects to propagate up the call stack to a higher-level layer until they are handled or reach the system's edge.

This can be done by rethrowing the defect by calling the `ZIO#refailCause`:

```

1 val app: ZIO[Any, Throwable, Unit] =
2   UserService
3     .register("john")
4     .catchSome {
5       case UserAlreadyExists(username) =>
```

```

6     Console.printLine(s"User $username already exists, please
7     choose another username").orDie
8     case InvalidUsernameFormat(username) =>
9       Console.printLine(s"Invalid username format for $username
10    ").orDie
11   }
12   .catchAllDefect { defect =>
13     ZIO.logErrorCause("An unexpected error occurred", Cause.die
14     (defect)) *>
15     ZIO.refailCause(Cause.die(defect)) // rethrow any
16     unexpected errors
17   }

```

Unfortunately, the `ZIO#catchAllDefect` operator does not provide us with contextual information about the defect and its cause. Instead, we can use the `ZIO#tapDefect` operator:

```

1 trait ZIO[R, E, A] {
2   def tapDefect[R1 <: R, E1 >: E](
3     f: Cause[Nothing] => ZIO[R1, E1, Any]
4   ): ZIO[R1, E1, A] = ????
5 }

```

It taps into the defects of the effect and allows you to do some side effects, such as logging the defect and its cause:

```

1 val app: ZIO[Any, Throwable, Unit] =
2   UserService
3     .register("john")
4     .catchSome {
5       case UserAlreadyExists(username) =>
6         Console.printLine(s"User $username already exists, please
7           choose another username").orDie
8         case InvalidUsernameFormat(username) =>
9           Console.printLine(s"Invalid username format for $username
10          ").orDie
11       }
12     .tapDefect{ cause =>
13       ZIO.logErrorCause("An unexpected error occurred", cause) *>
14       ZIO.refailCause(cause) // rethrow any unexpected errors
15     }

```

Now, let's talk about modeling domain errors, which makes our API design more robust.

### 26.2.1.3 Model Domain Errors Using Algebraic Data Types

Until now, we have used the `Throwable` error type, which helps us catch expected errors in a type-safe manner and give proper feedback to the user. But the `Throwable` won't

help us distinguish between defects and recoverable errors clearly. It is a superclass of all exceptions, and the problem with this error type is that it is not sealed. This means the compiler cannot detect what other types of errors remain to handle.

For example, what happens if the `UserService.register` method fails with a `RegistrationQuotaExceeded` exception that is not documented in the API? However, this is a failure, not a defect; we are not prepared to handle it, and it will cause the application to crash.

To avoid such situations, since we know that `app` is a fallible effect due to its error type, `Throwable`, on the error channel (`ZIO[Any, Throwable, Unit]`), we can use the `ZIO#catchAll` operator instead of `ZIO#catchSome` and add a default case to handle any remaining errors:

```

1 val app: ZIO[Any, Throwable, Unit] =
2   UserService
3     .register("john")
4     .catchAll {
5       case UserAlreadyExists(username) =>
6         Console.printLine(s"User $username already exists, please
choose another username")
7       case InvalidUsernameFormat(username) =>
8         Console.printLine(s"Invalid username format for $username
")
9       case other =>
10        ZIO.logErrorCause("An unexpected error occurred", Cause.
fail(other)) *>
11        ZIO.refailCause(Cause.fail(other))
12    }
13    .tapDefect{ cause =>
14      ZIO.logErrorCause("An unexpected error occurred", cause) *>
15      ZIO.refailCause(cause) // rethrow any unexpected errors
16  }

```

After catching and logging the remaining errors, we can detect unhandled errors and fix our domain errors to cover all edge cases. It is obvious that using `Throwable` didn't give us any significant advantage, and we still should take care of all remaining errors.

To prevent running into such situations, we can use a sealed trait to define all possible errors that can occur in our domain and use it as the error type of our effect. This narrows down the error type from `Throwable` to some specific types, which makes the compiler able to detect what other types of errors remain to handle through exhaustive type checking:

```

1 sealed trait UserServiceError
2   extends Product with Serializable
3   case class UserAlreadyExists(username: String)
4   extends UserServiceError

```

```

3 | case class InvalidUsernameFormat(username: String) extends
4 |   UserServiceError
5 | case class RegistrationQuotaExceeded(limit: Int)    extends
6 |   UserServiceError

```

Nothing has changed in the `UserService.register` method, but now it returns `ZIO[Any, UserServiceError, Unit]` instead of `Task[Unit]`:

```

1 | object UserService {
2 |   def register(username: String): ZIO[Any, UserServiceError, Unit]
3 |   ] =
4 |     if (validateUsername(username))
5 |       ZIO.fail(InvalidUsernameFormat(username))
6 |     else if (userExist(username))
7 |       ZIO.fail(UserAlreadyExists(username))
8 |     else
9 |       ZIO.succeed(LegacyUserService.registerLogic(username))
}

```

Now, after calling the `UserService.register` method, we can use the `ZIO#catchAll` operator. If we forget to catch one of the defined errors, the compiler detects it and gives us a compile-time error:

```

1 | val app: zio.ZIO[Any, Throwable, Unit] =
2 |   UserService
3 |   .register("john")
4 |   .catchAll {
5 |     case UserAlreadyExists(username) =>
6 |       Console.printLine(s"User $username already exists, please
7 | choose another username").orDie
8 |     case InvalidUsernameFormat(username) =>
9 |       Console.printLine(s"Invalid username format for $username
" ).orDie
}

```

If we forget to handle the `RegistrationQuotaExceeded` error, the compiler will give us a compile-time error:

```

1 | match may not be exhaustive.
2 | It would fail on the following input: RegistrationQuotaExceeded(_)
3 |           .catchAll {
}

```

#### 26.2.1.4 Centralize Logging Defects

As a client, when we call a method that can fail because of expected and unexpected errors, besides the fact that we would like to catch all expected errors and decide what to do with

them, we tend to log the unexpected errors in the first place. Why? Because we have more contextual information about the location where the defect is caught.

For example, we know that we caught the defect right after calling the `register` method with the input argument `john`. Therefore, there is a high chance that the defect is related to the `register` method or the input argument. As a good practice, we gather all contextual data and log it along with the defect. After logging the defect, we rethrow it.

Although catching the defect, logging it, and rethrowing it in the first place is a good practice, as it helps us to have more contextual information about the defect, it can be problematic when our application grows. After calling each method call, it is a repetitive task and can lead to code duplication. It also leads to duplicate logging; when you log a defect and rethrow it, the same defect might be logged multiple times as it propagates through the system. This can lead to log spam, making it harder to track down the root cause of the defect.

The best practice is to move the logging layer to the system's edge, where all defects can be logged in a centralized place, handled if possible, and rethrown if necessary. Whenever detailed information about a defect is needed, we can catch it right after the method call and add contextual information but defer the logging to a higher application layer.

Let's continue and discuss one last piece of advice when handling defects: sandboxing.

#### 26.2.1.5 Sandboxing at the Edge

When a defect occurs in a ZIO application, it propagates up the call stack until an error handler catches it. If the defect is not handled, it may cause the entire program to crash. This is especially undesirable when errors occur at the system's edges, such as when interacting with external services or handling user input.

Sandboxing is a technique for isolating specific portions of your code so that any errors within those portions do not affect the rest of your program, especially when interacting with external services or the end user. This is particularly useful for isolating the boundaries of your application.

A ZIO effect can fail due to three types of events: expected errors, unexpected errors, and interruptions. Under the hood, ZIO uses the `Cause` data type to represent the full cause of failures, including expected, unexpected, and interruptions.

To prevent an error from propagating up the call stack beyond the edge of the system, you can use the `ZIO#sandbox` operator, which has the following signature:

```
1 trait ZIO[-R, +E, +A] {
2   def sandbox: ZIO[R, Cause[E], A]
3 }
```

Sandboxing an effect exposes the full cause of a failure in the error channel. The `sandbox` operator converts all failures, interruptions, and defects into the error channel, changing its type from `E` to `Cause[E]`. This allows you to catch and handle all types of failures that occur within the effect:

```

1 effect.sandbox.catchAll {
2   case Cause.Die(t, _) =>
3     // Handle defects
4     ZIO.unit
5   case Cause.Fail(e, _) =>
6     // Handler failures
7     ZIO.unit
8   case Cause.Interrupt(fiberId, _) =>
9     // Handle interruptions
10    ZIO.unit
11   case _ =>
12     ZIO.debug("other cases")
13 }
```

This approach lets you use the ordinary error-handling operators to manage the `Cause`. When you're done, if you want to revert to dealing only with typed errors, you can use the `unsandbox` operation, which is the inverse of `sandbox`.

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

```

1 trait ZIO[-R, +E, +A] {
2   def unsandbox[E1](implicit ev: E <:< Cause[E1]): ZIO[R, E1, A]
3 }
```

A common use case for sandboxing is when developing an API service. It ensures that errors occurring while processing a request on one endpoint do not cause the entire service to crash, which could affect other endpoints.

Together, `sandbox` and `unsandbox` are powerful enough to implement any operators you might need to deal with the full cause of a failure. However, ZIO also provides a variety of convenience methods, such as `ZIO#catchAllCause`, `ZIO#mapErrorCause`, and `ZIO#foldCauseZIO`, which simplify handling specific use cases.

### 26.2.2 Fatal Errors

Fatal errors are the most severe category of non-recoverable errors. They represent critical failures that cannot be handled by any application layer and should result in an immediate shutdown of the application to prevent further damage or data corruption.

Examples of fatal errors include out-of-memory errors, stack overflow errors, critical hardware failures, and segmentation faults.

Catching fatal errors might seem like a way to make software more robust, but it is generally considered a bad practice. Fatal errors often indicate that the application is in an unpredictable or corrupted state. Catching these errors does not resolve the underlying issue and can leave the application running in a compromised state, which may lead to further errors, data corruption, or security vulnerabilities.

Let's assume that you can catch fatal errors and you catch one. What should you do next?

The best action is to log the error and immediately shut down the application to prevent further damage. This cannot be done safely by simply combining ZIO operators; it should be handled by the ZIO runtime itself.

ZIO runtime takes care of reporting fatal errors, by default, printing the call stack to the console and exiting the application with a proper exit code:

```

1 import zio._

2

3 object FatalStackoverflowErrorDemo extends ZIOAppDefault {
4   // A ZIO app that will throw a StackOverflowError
5   val myApp =
6     ZIO.attempt(
7       throw new StackOverflowError(
8         "The call stack pointer exceeds the stack bound."
9       )
10    )
11

12 def run =
13   myApp
14     .catchAll(_ => ZIO.unit)           // ignoring all expected
15     errors
16     .catchAllDefect(_ => ZIO.unit) // ignoring all defects
17     .ensuring(
18       ZIO.debug(
19         "This will never be printed because the program exits
20         ungracefully before it gets here."
21       )
22     )
23 }
```

Please note that fatal errors are platform-dependent. For instance, on the JVM, any `Throwable` that is a subclass of `java.lang.VirtualMachineError` is considered a fatal error. The `StackOverflowError`, a subclass of `java.lang.VirtualMachineError`, is treated as a fatal error by the ZIO runtime. When a `StackOverflowError` occurs, the ZIO runtime immediately shuts down the application and prints the call stack to the console. Even if all expected errors and defects are caught and ignored, fatal errors cannot be caught, resulting in an ungraceful application exit and preventing the finalizer from executing.

Additionally, you can introduce more fatal errors to the ZIO runtime system using the `Runtime.addFatal` method. For example, if you are developing a critical ledger system for a financial institution, you might want to treat an invalid state on the ledger, such as an `UnbalancedLedgerError`, as a fatal error. If your goal is to immediately shut down the application and prevent further damage to the ledger rather than exiting gracefully, you should add this error as a fatal error to the ZIO runtime system:

```

1 class UnbalancedFatalError(msg: String) extends Throwable(msg)
```

```

2
3 object CustomFatalErrorDemo extends ZIOAppDefault {
4   override val bootstrap: ZLayer[ZIOAppArgs, Any, Any] =
5     Runtime.addFatal(classOf[UnbalancedFatalError])
6
7   val myApp =
8     ZIO
9     .attempt(
10       throw new UnbalancedFatalError(
11         "Ledger is corrupted! The total balance is not zero."
12       )
13     )
14
15   def run =
16     myApp
17     .ensuring(
18       ZIO.debug("Clean up ledger and exit gracefully.")
19     )
20 }
```

If you run the `CustomFatalErrorDemo` application, you will see that the application immediately shuts down and prints the call stack to the console:

```

1 CustomFatalErrorDemo$UnbalancedFatalError: Ledger is corrupted!
2   The total balance is not zero.
3   at CustomFatalErrorDemo$.anonfun$myApp$1(
4     CustomFatalErrorDemo.scala:13)
5   at zio.ZIOCompanionVersionSpecific.$anonfun$attempt$1(
6     ZIOCompanionVersionSpecific.scala:100)
7   at zio.internal.FiberRuntime.runLoop(FiberRuntime.scala:999)
8   at zio.internal.FiberRuntime.runLoop(FiberRuntime.scala:1067)
9   at zio.internal.FiberRuntime.evaluateEffect(FiberRuntime.
10    scala:413)
11  at zio.internal.FiberRuntime.evaluateMessageWhileSuspended(
12    FiberRuntime.scala:489)
13  at zio.internal.FiberRuntime.drainQueueOnCurrentThread(
14    FiberRuntime.scala:250)
15  at zio.internal.FiberRuntime.run(FiberRuntime.scala:138)
16  at zio.internal.ZScheduler$$anon$3.run(ZScheduler.scala:380)
17 **** WARNING ****
18 Catastrophic error encountered. Application not safely
19 interrupted. Resources may be leaked. Check the logs for more
20 details and consider overriding `Runtime.reportFatal` to
capture context.
```

By default, the ZIO runtime prints the call stack to the console. However, if you need more details—such as printing fiber dumps to the console or saving them to a file before exiting

the application—you can customize this behavior by providing a custom fatal report handler using the `Runtime#setReportFatal` method:

```

1 override val bootstrap =
2   Runtime.addFatal(classOf[UnbalancedFatalError]) ++
3     Runtime.setReportFatal { (t: Throwable) =>
4       if (t.isInstanceOf[UnbalancedFatalError]) {
5         Unsafe.unsafe { implicit u =>
6           Runtime.default.unsafe.run(Fiber.dumpAll)
7         }
8       }
9       Runtime.defaultReportFatal(t)
10 }
```

## 26.3 Conclusion

In this chapter, you have learned the best practices for handling three types of errors that can occur in an application. Errors are categorized into recoverable errors, defects, and fatal errors. Recoverable errors are expected and can be managed by the application, whereas defects are unexpected and not part of the domain errors. Fatal errors are the most severe type and should not be handled by any layer of the application.

Let's recap the best practices we learned in this section:

1. Log the defects for further investigation and debugging analysis.
2. Don't swallow defects in the first place, and let them propagate up the call stack if you don't know how to handle them.
3. Model domain errors using algebraic data types to distinguish between domain errors and defects.
4. Don't log and rethrow the defects; enrich them if necessary, let them propagate up the call stack, and defer the logging to the centralized place at the system's edge.
5. Sandbox a long-running application at the edge to prevent defects from propagating up beyond the system's edge.
6. If you encounter fatal errors, log enough about the error and fail fast. Shut down the application immediately to prevent further damage or data corruption.

## 26.4 Exercises

1. When should you use an *abstract class* instead of a *sealed trait* to model domain errors? Provide an example using `UserServiceError`.
2. In Scala 3, how would you model `UserServiceError` using enums instead of a sealed trait?
3. You are developing a user registration form with the following criteria:
  - Username must be at least five characters long.

- Password must be at least eight characters long.
- Email must contain an "@" and a domain name.
- Age must be 18 or older.

How would you model and handle errors in the `register` method? The goal is to collect all errors and provide comprehensive feedback to the user without failing fast.

**Hint:** Consider using the `ZIO#validate` operator to validate input and collect all errors.

4. Utilize ZIO Prelude's `Validation` data type to accumulate errors from the previous exercise. Compare this approach with the previous method and discuss the pros and cons of each.

## Chapter 27

# Streaming: First Steps with ZStream

Streaming is an essential paradigm for modern data-driven applications. A common use case is that we constantly receive 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 ZIO's features, such as its support for interruptibility and resource safety, provide the foundation for solving these problems robustly and principled.

### 27.1 Streams as Effectful Iterators

The core data type of ZIO Stream is `ZStream[R, E, O]`, an *effectful 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*, 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 understand this, consider how we would represent a stream of tweets using only a ZIO.

```

1 import zio._

2 trait Tweet

4

5 lazy val getTweets: ZIO[Any, Nothing, Chunk[Tweet]] =
6     ???

```

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 more tweets will always be produced in the future!

Thus, `getTweets` cannot represent getting a stream of tweets where new tweets are constantly 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 repeatedly evaluate it in the future to get all the incremental tweets since the last time we've evaluated it.

The `ZStream` data type does just that:

```

1 trait ZStream[-R, +E, +O] {
2     def process: ZIO[R with Scope, Option[E], Chunk[O]]
3 }

```

A `ZStream` is defined in terms of one operator, `process`, which can be evaluated repeatedly to pull more elements from the stream.

Each time the `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 scoped ZIO so that resources can be safely acquired and released in the stream context.

With this signature, you can think of `ZStream` as an *effectful iterator*.

Recall that the signature of `Iterator` is:

```

1 trait Iterator[+A] {
2     def next: A
3     def hasNext: Boolean
4 }

```

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` will enable us to repeatedly evaluate `process` if it succeeds in getting 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 *effectful* values.

## 27.2 Streams as Collections

In addition to thinking of a `ZStream` as an effectful 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 `ZStream#map` and `ZStream#filter`, when working with streams:

```

1 trait ZStream[-R, +E, +O] {
2   def filter(f: O => Boolean): ZStream[R, E, O]
3   def map[O2](f: O => O2): ZStream[R, E, O2]
4 }
```

Treating streams as collections lets us write higher level, more declarative code than working with them as effectful 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 effectful iterator primarily to implement new operators on streams and to understand the implementation of existing operators.

### 27.2.1 Implicit Chunking

One thing to notice here is that while the underlying effectful 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, 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. However, as the user, you can 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, manually changing the chunk size may be helpful for optimization. For example, if you have incoming data that is being received individually, it may be helpful to chunk it before further processing.

You can always operate at the level of chunks if you want to. However, ZIO Stream's philosophy is that the user should have to deal manually with chunking only as an optimization and that, in most cases, the framework should automatically "do the right thing" regarding 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 effectful iterator, let's look at how we can implement the `map` operator on `ZStream`:

```

1 trait ZStream[-R, +E, +O] { self =>
2   def process: ZIO[R with Scope, Option[E], Chunk[O]] =
3
4   def map[O2](f: O => O2): ZStream[R, E, O2] = {
5     new ZStream[R, E, O2] {
6       def process: ZIO[R with Scope, Option[E], Chunk[O2]] =
7         self.process.map(_.map(f))
8     }
9   }

```

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 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 ensure that users get the right behavior operating on stream values without worrying about chunking themselves.

### 27.2.2 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:

```

1 val words: List[String] =
2   List("the", "quick", "brown", "fox")
3
4 val sorted: List[String] =
5   words.sorted

```

```
6 // 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 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*.

### 27.2.3 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 effectful variants with the `ZIO` suffix. For example, while the `map` operator can transform each stream value with a function, the `mapZIO` operator can transform each stream value with an effectful function.

When stream operators have effectful variants, they may also have variants that perform effects in parallel with the `Par` suffix.

For instance, there is also a `mapZIOPar` variant of the `mapZIO` operator. Whereas `mapZIO` performs the effectful function for each stream value before proceeding to process the next, the `mapZIOPar` operator processes up to a specified number of stream elements at a time in parallel.

## 27.3 Constructing Basic Streams

ZIO Stream offers many different ways of constructing streams. In the coming chapters, we will see more specialized variants, but here, we will focus on basic constructors to get started working with streams.

### 27.3.1 Constructing Streams from Existing Values

One of the simplest stream constructors is `ZStream.fromIterable`, which takes an `Iterable` and returns a stream containing the values in that `Iterable`:

```

1 object ZStream {
2   def fromIterable[0](os: Iterable[0]): ZStream[Any, Nothing, 0]
3     =
4     ???
5   }
6   lazy val stream: ZStream[Any, Nothing, Int] =
7     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 helpful constructors to compose with other operators. For example, you could create a stream from a list of file names and then create a new stream describing incremental reading from each of those files.

### 27.3.2 Constructing Streams from Effects

Another set of basic stream constructors are `ZStream.fromZIO` and `ZStream.scoped`, which let us lift a ZIO or a scoped ZIO effect into a single element stream:

```

1 object ZStream {
2   def fromZIO[R, E, O](zio: ZIO[R, E, O]): ZStream[R, E, O] =
3     ???  

4
5   def scoped[R, E, O](
6     zio: ZIO[R with Scope, E, O]
7   ): ZStream[R, E, O] =
8     ???  

9 }
10
11 import zio._  

12 import zio.stream._  

13
14 val helloStream: ZStream[Any, Nothing, Unit] =  

15   ZStream.fromZIO(Console.printLine("hello").orDie) ++
16   ZStream.fromZIO(Console.printLine("from").orDie) ++
17   ZStream.fromZIO(Console.printLine("a").orDie) ++
18   ZStream.fromZIO(Console.printLine("stream").orDie)

```

These constructors only create single-element streams, but now let us work with ZIO effects within the stream context. We can combine them with other operators to make 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.

### 27.3.3 Constructing Streams from Repetition

Building on these operators, the `ZStream.repeat`, `ZStream.repeatZIO`, and `ZStream.repeatZIOOption` constructors and their Chunk variants get us even closer to constructing complex streams directly:

```

1 object ZStream {
2   def repeat[O](
3     o: => O
4   ): ZStream[Any, Nothing, O] = ???  

5
6   def repeatZIO[R, E, O](
7     zio: ZIO[R, E, O]
8   ): ZStream[R, E, O] = ???  

9
10  def repeatZIOChunk[R, E, O](
11    zio: ZIO[R, E, Chunk[O]]
12  ): ZStream[R, E, O] = ???  

13
14  def repeatZIOOption[R, E, O](
15    zio: ZIO[R, Option[E], O]
16  ): ZStream[R, E, O] = ???
17

```

```

18 def repeatZIOChunkOption[R, E, O](
19   zio: ZIO[R, Option[E], Chunk[O]])
20 ): ZStream[R, E, O] = ???
21 }
22
23 lazy val ones: ZStream[Any, Nothing, Int] =
24   ZStream.repeat(1)
25
26 trait Tweet
27
28 lazy val getTweets: ZIO[Any, Nothing, Chunk[Tweet]] =
29   ???
30
31 lazy val tweetStream: ZStream[Any, Nothing, Tweet] =
32   ZStream.repeatZIOChunk(getTweets)

```

These constructors let us repeat either a single value or perform 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 return 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.

#### 27.3.4 Constructing Streams from Unfolding

The final set of stream constructor variants we will look at here are the `unfold` variants:

```

1 object ZStream {
2   def unfold[S, A](
3     s: S
4   )(f: S => Option[(A, S)]): ZStream[Any, Nothing, A] = ???
5
6   def unfoldZIO[R, E, A, S](s: S)(
7     f: S => ZIO[R, E, Option[(A, S)]]
8   ): ZStream[R, E, A] = ???
9
10  def unfoldChunk[S, A](
11    s: S
12  )(f: S => Option[(Chunk[A], S)]): ZStream[Any, Nothing, A] =
13    ???
14
15  def unfoldChunkZIO[R, E, A, S](
16    s: S
17  )(f: S => ZIO[R, E, Option[(Chunk[A], S)]]): ZStream[R, E, A] =
18    ???
19 }

```

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 when constructing the stream. For example, here is how we can use `ZStream#unfold` to construct a `ZStream` from a `List`:

```

1 def fromList[A](list: List[A]): ZStream[Any, Nothing, A] =
2   ZStream.unfold(list) {
3     case h :: t => Some((h, t))
4     case Nil      => None
5   }

```

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 describe streams that continue forever safely. 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 effectful variant, `unfoldZIO`, 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 `unfold` and `unfoldZIO`. These variants allow for the return of a `Chunk` of values instead of a single value in each step, and they will be more efficient when stream values can naturally be produced in chunks.

## 27.4 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 various 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, your strategy for running it should have some way of terminating without evaluating the entire stream unless the point of your application is just to run it forever.

Beyond that, thinking about what kind of result you want from running the stream is helpful.

If your entire program is described as a stream, you may not need any result from the stream 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?

### 27.4.1 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 `ZStream#foldWhile`, which allows reducing a stream to a summary value and also signaling early termination.

```

1 trait ZStream[-R, +E, +O] {
2   def foldWhile[S](
3     s: S
4     )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
5 }
```

You can think of `ZStream#foldWhile` as like `foldLeft` in the Scala collection library with an additional parameter supporting early termination. Note that there is also an effectful version `ZStream#foldWhileZIO` that allows performing effects within the fold as well as a `ZStream#foldWhileScoped` variant that return a scoped ZIO.

Let's see how we can use `ZStream#foldWhile` to implement some of the other more specialized operators for running streams.

### 27.4.2 Running a Stream for its Effects

The first operator we will implement is `ZStream#runDrain`, which runs a stream to completion purely for its effects:

```

1 trait ZStream[-R, +E, +O] {
2   def foldWhile[S](
3     s: S
4   )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S] = ???  

5
6   def runDrain: ZIO[R, E, Unit] =
7     foldWhile(()(_ => true)((s, _) => s)
8 }
```

The `ZStream#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 `ZStream#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 `ZStream#foreach` operator:

```

1 trait ZStream[-R, +E, +O] {
2   def foldWhileZIO[R1 <: R, E1 >: E, S](
3     s: S
4   )(  

5     cont: S => Boolean
6   )(f: (S, O) => ZIO[R1, E1, S]): ZIO[R1, E1, S] = ???  

7
8   def foreach[R1 <: R, E1 >: E, Any](  

9     f: O => ZIO[R1, E1, Any]
10   ): ZIO[R1, E1, Unit] =
```

```

11     foldWhileZIO[R1, E1, Unit](()(_ => true)((_, o) => f(o).unit
12     )

```

The operator is similar to `ZStream#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 `ZStream#foreach` to print each tweet to the console or write each tweet to a database.

One feature of `ZStream#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.

```

1 for {
2   x <- List(1, 2)
3   y <- List(x, x * 3)
4 } yield (x, y)

```

When a *for comprehension* does not end in a `yield`, it is translated into a series of `foreach` calls instead.

```

1 for {
2   x <- List(1, 2)
3   y <- List(x, x * 3)
4 } println((x, y))

```

Normally, we avoid using *for comprehensions* in this way because the standard definition of `foreach` on a data type like `List` is:

```

1 trait List[+A] {
2   def foreach[U](f: A => U): Unit
3 }

```

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 `ZStream#foreach` method returns a `ZIO` effect, and so it is safe to use. Using this, we can write code like:

```

1 val effect: ZIO[Any, Nothing, Unit] =
2   for {
3     x <- ZStream(1, 2)
4     y <- ZStream(x, x + 3)
5   } Console.printLine((x, y).toString).orDie

```

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".

### 27.4.3 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:

```

1 trait ZStream[-R, +E, +O] {
2   def foldWhile[S](
3     s: S
4   )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
5   def runHead: ZIO[R, E, Option[O]] =
6     foldWhile[Option[O]](None)(_.isEmpty)((_, o) => Some(o))
7   def runLast: ZIO[R, E, Option[O]] =
8     foldWhile[Option[O]](None)(_ => true)((_, o) => Some(o))
9 }
```

The `ZStream#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 `ZStream#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 handy 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 `ZStream#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`:

```

1 trait ZStream[-R, +E, +O] {
2   def foldWhile[S](
3     s: S
4   )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
5   def runCollect: ZIO[R, E, Chunk[O]] =
6     foldWhile[Chunk[O]](Chunk.empty)(_ => true)((s, o) => s :+ o)
7 }
```

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 `ZStream#take`. For example, if we just want to view a sample of tweets about a topic for inspection, we could just use `ZStream#take(100)` to

take the first hundred values and then `ZStream#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:

```

1 import scala.math.Numeric
2
3 trait ZStream[-R, +E, +O] {
4   def foldWhile[S](
5     s: S
6   )(cont: S => Boolean)(f: (S, O) => S): ZIO[R, E, S]
7   def runCount: ZIO[R, E, Long] =
8     foldWhile(0L)(_ => true)((s, _) => s + 1L)
9   def runSum[O1 >: O](implicit ev: Numeric[O1]): ZIO[R, E, O1] =
10    foldWhile(ev.zero)(_ => true)(ev.plus)
11 }
```

The `ZStream#runCount` operator runs a stream to completion, returning the total number of values produced by the stream. The `ZStream#runSum` operator is only defined for numeric streams and simply returns the sum of all the stream values.

## 27.5 Type Parameters

Like `ZIO`, `ZStream` has three type parameters:

- `R` represents the set of services required 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 regarding `ZIO`, so we have been able to get this far by saying very little about them other than their basic definition.

However, it is helpful to highlight some of the further similarities as well as a couple of differences here.

### 27.5.1 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`, `ZStream.provideSomeLayer`, and `ZStream#provideCustomLayer\index{ZStream\#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:

- `ZStream#environment` - Access the whole environment and return a value that depends on it
- `ZStream#environmentWith` - Access the given environment and return a stream that depends on it
- `ZStream#environmentWithZIO` - Access the given environment and return a `ZIO` that depends on it
- `ZStream#environmentWithStream` - Access the given 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 or a stream.

### 27.5.2 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 `ZStream#catchAll` or `ZStream#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 a stream fails with an `E`, we can use an operator like `ZStream#catchAll` to ignore that error and execute the logic of a new stream instead. However, it is not well-defined to ignore that error and then pull from the original stream again.

## 27.6 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 effectful iterator and how this represents the fundamental *functional* analog to an `Iterator` to describe a collection of potentially infinitely many *effectful* 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 apply 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 and introduce the concept of *sinks*, which can be thought of

as composable strategies for running streams for their effects.

From there, we will look back at some of the more advanced operators for working with streams, dividing these operators between those that transform one stream into 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: channels, sinks, and pipelines. 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 28

# Streaming: Next Steps With ZStream

The previous chapter introduced the concept of streams. We saw how we can create, transform, and consume streams.

In the example below, we are creating a stream using the `ZStream.apply` constructor, transforming it with the `map` operator, and then consuming it with the `ZStream#foreach` operator:

```
1 import zio._  
2 import zio.stream._  
3  
4 object Example1 extends ZIOAppDefault {  
5   val run =  
6     ZStream(1, 2, 3, 4, 5)  
7       .map(_ * 2)  
8       .foreach(Console.printLine(_))  
9 }
```

This style of “chaining” stream operators is a very common way of working with streams. Calling operators on a concrete data type is very straightforward, typically has excellent tooling support, and can be very readable if each stream operator is listed on a new line like the above.

However, there would be something unsatisfying if this was the only way we had to define ways to transform and consume streams. To see this, think about all the many ways we might want to consume a stream.

We may need to execute a ZIO workflow for each stream element, aggregate it into an in-memory data structure, write it to a file, or store it in a database, among many other possibilities.

We might define some of these common ways of consuming a stream as operators on a stream itself, just like the `foreach` operator:

```

1 trait ZStream[-R, +E, +A] {
2   def runToFile(name: String)(implicit
3     ev: A <:< String
4   ): ZIO[R, E, Unit]
5 }
```

This would be fine for very generic ways of consuming a stream that did not require any dependencies, but what happens if we want to define a way of writing a stream to S3? ZIO Stream can't depend on every possible library that might want to consume a stream, so we will no longer be able to define these operators on the stream itself:

```

1 object S3 {
2   def writeStream[R, E](
3     stream: ZStream[R, E, Byte]
4   ): ZIO[R, E, Unit] =
5   ???
6 }
```

This isn't the end of the world, but it does mean that calling this operator will be significantly less ergonomic since we will no longer be able to "chain" stream operators like before.

In addition, it isn't easy to modify these ways of consuming streams further if we need to. For instance, the `writeStream` method above accepts a stream of bytes, but we should be able to create a way of consuming a stream of strings that first converts the strings to bytes and then writes them to S3.

Conceptually, this requires transforming the inputs from strings to bytes before sending them to S3, but doing this is somewhat awkward with the above definition:

```

1 def writeStreamString[R, E](
2   stream: ZStream[R, E, String]
3 ): ZIO[R, E, Unit] =
4   S3.writeStream(
5     stream
6       .map(string => Chunk.fromArray(string.getBytes))
7       .flattenChunks
8   )
```

These problems are all symptoms of the fact that so far we have not been treating ways of transforming streams or consuming streams as *first-class values*. When something is just a method and not a data type, we can't define operators for working with it, so we have problems like the ones we saw above.

We solve this by making these ways of transforming and consumer streams their own data types. We call a stream transformer a `ZPipeline` and a stream consumer a `ZSink`:

```

1 trait ZStream[-R, +E, +A] {
2   def via[R1 <: R, E1 >: E, B](
3     pipeline: ZPipeline[R1, E1, A, B]
4   ): ZStream[R1, E1, B]
5
6   def run[R1 <: R, E1 >: E, B](
7     sink: ZSink[R1, E1, A, Any, B]
8   ): ZIO[R1, E1, B]
9 }
10
11 trait ZPipeline[-Env, +Err, -In, +Out]
12
13 object ZPipeline {
14   def map[In, Out](
15     f: In => Out
16   ): ZPipeline[Any, Nothing, In, Out] =
17     ????
18 }
19
20 trait ZSink[-Env, +Err, -In, +Leftover, +Summary]
21
22 object ZSink {
23   def fromFile(
24     name: String
25   ): ZSink[Any, Throwable, Byte, Nothing, Unit] =
26     ????
27 }
```

We can see that a `ZPipeline` is something that can transform a stream of values of type `In` to a stream of values of type `Out`. The pipeline can also use an environment of type `Env` to perform the transformation, and the transformation can potentially fail with an error of type `Err`.

A `ZSink` is something that can consume a stream of values of type `In` to produce a value of type `Summary`. The sink can also use an environment and potentially fail, and it may also choose not to consume some of the original stream elements of type `Leftover`, which we will discuss later.

Conceptually, we can rewrite any stream program written in the style of chaining operators to instead represent the corresponding stream transformation and consumption operators as pipelines. For instance, here is what the original example from the beginning of this chapter would look like with pipelines and sinks:

```

1 val stream: ZStream[Any, Nothing, Int] =
2   ZStream(1, 2, 3, 4, 5)
3
4 val pipeline: ZPipeline[Any, Nothing, Int, Int] =
```

```

5   ZPipeline.map(_ * 2)
6
7   val sink: ZSink[Any, Nothing, Int, Nothing, Unit] =
8     ZSink.foreach(Console.printLine(_).orDie)
9
10 stream.via(pipeline).run(sink)

```

We can also use the `>>>` symbolic operator instead of `via` and `run` so we could also write the last example like this:

```

1 stream >>> pipeline >>> sink

```

Your taste for symbolic operators may vary, but the arrows' direction nicely mirrors the direction of the data flow, with information going from the stream, being transformed by the pipeline, and finally being consumed by the sink.

You can also see here how pipelines and sinks help us scale our streaming applications.

In a very simple example like this, we probably don't need to make the transformation or consumption of the stream its own data type.

But imagine that the pipeline involved some very complex windowing that might be implemented by a separate team member or an entirely different team. The pipeline allows all of that logic to be wrapped up in this one `ZPipeline` data type, and then for you to just "plug in" that transformation to whatever part of your application it is needed in.

You could have multiple complex data transformation stages that all just come together in your application like this:

```

1 inputStream >>>
2   complexTransformation1 >>>
3   complexTransformation2 >>>
4   complexTransformation3 >>>
5   outputSink

```

In addition, because pipelines and sinks are now their own data types, we can now define operators on them. For example, let's go back to our problem above of how we can create a way of consuming a stream of strings and writing them to S3 from a way of consuming a stream of bytes and writing them to S3.

In the world of sinks and pipelines, that would just look like this:

```

1 lazy val s3Sink: ZSink[Any, Nothing, Byte, Nothing, Unit] =
2   ???
3
4 lazy val s3SinkBytes: ZSink[Any, Nothing, String, Nothing, Unit] =
5   s3Sink.contramapChunks[String](strings =>
6     strings.flatMap(string => Chunk.fromArray(string.getBytes))
7   )

```

Or using pipelines as well:

```

1 lazy val s3SinkBytes2: ZSink[Any, Nothing, String, Nothing, Unit]
2   =
3   ZPipeline.mapChunks[String, Byte](strings =>
4     strings.flatMap(string => Chunk.fromArray(string.getBytes))
5   ) >>> s3Sink

```

We can even use operators on sinks and pipelines to do much more complex operations. For example, if in our application, we wanted to write to two different data consumers, at the end of our pipeline, we can do it like this:

```

1 inputStream >>>
2   complexTransformation1 >>>
3   complexTransformation2 >>>
4   complexTransformation3 >>>
5   outputSink1.zipPar(outputSink2)

```

Here, the `ZSink#zipPar` operator combines two sinks into a new sink that will send all of its inputs to both of the original sinks:

```

1 trait ZSink[-R, In, E, L, Z] {
2   def zipPar[R1 <: R, In1 <: In, E1 >: E, L1 >: L <: In1, Z1](
3     that: => ZSink[R1, E1, In1, L1, Z1]
4   ): ZSink[R1, E1, In1, L1, (Z, Z1)] =
5   ????
6 }

```

Now that we have an understanding of what sinks and pipelines are, let's look at each of them in more detail and how we can use them.

## 28.1 Sinks

As discussed above, sinks describe ways of consuming elements of a stream. So whenever you need to “do something” with the elements of a stream, think about a `ZSink`.

And in fact, ZIO provides a variety of helpful sinks in the `ZSink` companion object for doing just this, such as `ZSink.collectAll` for collecting all the elements of a stream or `ZSink.fromFile` for writing the elements of a stream to a file.

Here is an example of how we could read the contents of a file into a stream and then write the contents of that stream to another file:

```

1 val stream: ZStream[Any, Throwable, Byte] =
2   ZStream.fromFileName("README.md")
3
4 val sink: ZSink[Any, Throwable, Byte, Byte, Long] =
5   ZSink.fromFileName("README2.md")

```

```

6
7 stream.run(sink)

```

To get a better understanding of sinks, let's go back to the type signature of a sink:

```

1 trait ZSink[-Env, +Err, -In, +Leftover, +Summary]

```

We can think of a sink as something that accepts inputs and, after being fed those inputs, either says it is “done” with a summary value and potentially some leftovers or that it “wants more”.

One way of representing this is as follows:

```

1 trait ZSink[-Env, +Err, -In, +Leftover, +Summary] {
2   def push
3     : ZIO[Any, Nothing, In => ZIO[Any, (Summary, Leftover), Unit
4   ]]
5 }

```

In this representation, we “start” the sink by evaluating the outer ZIO workflow. Then, we can “feed” the sink inputs of type In, and each time we either get a workflow that succeeds with Unit, indicating “need more input”, or a workflow that fails with a tuple of leftovers and a summary value, indicating that it is “done”.

We need to augment this representation slightly because we need a way to signal that the upstream is “done”. We can do this by changing the In to an Option[In] where None represents that the stream the sink is consuming from is done.

```

1 trait ZSink[-Env, +Err, -In, +Leftover, +Summary] {
2   def push: ZIO[Any, Nothing, Option[In] => ZIO[
3     Any,
4     (Summary, Leftover),
5     Unit
6   ]]
7 }

```

Finally, we need to build in the environment and error types, and we need the sink to be able to safely manage resources such as closing the file handle it is writing to using Scope. We also need to build in chunking, just like we did with streams.

So the signature becomes:

```

1 trait ZSink[-Env, +Err, -In, +Leftover, +Summary] {
2   def push: ZIO[Env with Scope, Nothing, Option[Chunk[In]] => ZIO
3     [
4       Env,
5       (Either[Err, Summary], Chunk[Leftover]),
6       Unit
7     ]]
}

```

This isn't how a sink is implemented, and there is no need for you to remember this type signature yourself, but it is a useful mental model for thinking about how a sink works.

It keeps accepting input until it produces either an error or a summary value, along with potentially some leftovers. It can use ZIO workflows in handling these inputs, for example, to write them to a file, and can ensure that resources are safely closed when the sink is done using Scope.

One question you may be having right now is what the significance of these "leftovers" is.

Leftovers arise because some sinks may not consume all of their inputs. Most of the sinks we have seen so far, such as `ZSink.foreach` or `ZSink.fromFile`, consume all of their inputs and so do not produce any leftovers.

But consider a simple sink such as `ZSink.collectAllN`, which collects the specified number of stream elements into a Chunk. What do we do if we write a `Chunk(1, 2, 3, 4, 5)` to the sink?

The sink clearly only needs the first three elements to produce its summary value, but what should it do with the remaining two elements? It could just drop them on the floor, and that would work for some simple cases:

```

1 val stream: ZStream[Any, Nothing, Int] =
2   ZStream(1, 2, 3, 4, 5)
3
4 val sink: ZSink[Any, Nothing, Int, Int, Chunk[Int]] =
5   ZSink.collectAllN(3)
6
7 stream.run(sink)
8 // Chunk(1, 2, 3)

```

However, consider another operator for running a stream with a sink called `ZStream#transduce`. The contract of `ZStream#transduce` is that it will repeatedly run the stream with the sink, emitting each summary value produced by the sink as a new stream.

So, for example, we would expect the following to work:

```

1 stream.transduce(sink).runCollect
2 // Chunk(Chunk(1, 2, 3), Chunk(4, 5))

```

We can see how `ZStream#transduce` would be a powerful operator, allowing us to apply a variety of aggregations that may not consume the whole stream. But for operators like `ZStream#transduce` to work, the first sink can't just drop the leftovers on the floor but needs to pass them on to the next sink.

That's exactly what the leftover type lets us do.

Sinks come with various constructors to create a variety of commonly useful sinks in the `ZSink` companion object. For example, you can create a sink that writes to a file using `ZSink.fromFile` or a sink that writes to a queue or hub using `ZSink.fromQueue` or `ZSink.fromHub`.

The `ZSink.foreach` constructor can be used to perform a ZIO workflow for each stream element, such as writing it to a file or printing it to the console. The `ZSink.unwrapScoped` operator can be used to construct a sink from a scoped ZIO workflow returning a sink, so you could, for example, open a database connection and then return a sink that writes to the database for each element.

There are also versions of many of these stream constructors that work on chunks with the `Chunk` suffix. These can be useful for performance-sensitive applications where you want to work on the level of chunks instead of individual elements.

There is also a low-level `ZSink.fromPush` constructor that lets you construct a sink directly from the low-level representation discussed above.

There are also a variety of operators for modifying sinks or composing them. Many of these should be very familiar to you by now from your work with ZIO.

We saw above that we could transform the input type of the sink using `ZSink#contramap`. There are also several other variants of that operator that let you transform inputs at the level of chunks or transform inputs with ZIO workflows.

You can also transform the output type of a sink using `ZSink#map` or similar operators such as `ZSink#mapZIO` or `ZSink#mapChunks`.

One operator on sinks that is particularly powerful is `ZSink#flatMap`, which lets you construct a new sink based on the result of the original sink. This allows you to read some input and then decide how you want to process further input based on that input.

This allows us to do “parsing” with sinks. Though we will see later that pipelines are the most high-performance way to do encoding and decoding, sinks can be used to do parsing in a way that is very flexible and user-friendly and so can be a good solution if there is not an off-the-shelf pipeline for your encoding and decoding needs, and you don’t want to spend a lot of time on an implementation.

Let’s look at how we can use sinks to parse some data with a simple variable-length encoding.

We will assume we are working with a stream of characters. The first character represents the length of the first element to be decoded, and so on:

```
1 | val encoded: ZStream[Any, Nothing, Char] = 
2 |   ZStream('3', 'a', 'b', 'c', '2', 'd', 'e', '1', 'f')
```

We can first implement a sink or a parser for the length:

```
1 | val length: ZSink[Any, Nothing, Char, Char, Option[Char]] = 
2 |   ZSink.head
```

This sink will read the first character from the input stream and return either `Some` with the character or `None` if the stream is empty.

Based on this, we would then like to read the specified number of characters from the stream. Let’s create a sink that does this:

```

1 def chars(n: Int): ZSink[Any, Nothing, Char, Char, String] =
2   ZSink.collectAllN(n).map(_.mkString)

```

We can then use `flatMap` to combine these two sinks into a sink that parses the encoded data.

```

1 def parser: ZSink[Any, Nothing, Char, Char, String] =
2   length.flatMap {
3     case Some(n) => chars(n.toInt)
4     case None      => ZSink.dieMessage("Unexpected end of stream")
5   }

```

If we want to parse all the input, we just use the `ZSink#transduce` operator we saw above:

```

1 val decoded: ZStream[Any, Nothing, String] =
2   encoded.transduce(parser)
3
4 decoded.runCollect
5 // Chunk("abc", "de", "f")

```

Other operators on sinks include the `ZSink#zipPar` operator for sending the elements of a stream to both sinks and error handling operators such as `ZSink#foldSink` that work like `ZSink#flatMap` but let you handle the failure value of a sink in addition to its success value.

In summary, sinks are extremely effective at describing ways to consume a stream, such as writing it to a file. Sinks can also be highly flexible parsers of streams, though lower-level implementations in terms of pipelines will typically be more efficient.

## 28.2 Pipelines

Pipelines are the other major streaming data type. Whereas a sink represents the “end” of a data processing flow, a pipeline represents the “middle” of one.

Conceptually, you could think of a pipeline as a stream transformation function, though this is not the way it is actually implemented:

```

1 trait ZPipeline[-Env, +Err, -In, +Out] {
2   def apply[Env1 <: Env, Err1 >: Err](
3     in: ZStream[Env1, Err1, In]
4   ): ZStream[Env1, Err1, Out]
5 }

```

It just takes a stream with elements of type `In` and transforms it into a stream with elements of type `Out`. Along the way, the pipeline may potentially use some services of type `Env` or fail with an error of type `Err`.

A pipeline is, in some ways, simpler than a sink in that it has no concept of leftovers. It also has no concept of a terminal value.

In fact, a pipeline can typically be thought of as a process that is willing to go on forever. As long as the stream it is transforming has more elements, the pipeline will transform them, and it will never be “done” on its own.

While any stream transformation can potentially be represented as a pipeline, one of the most useful applications of pipelines is for encoders and decoders. By its nature, an encoder or decoder takes inputs of one type and converts them to outputs of another type, potentially failing along the way, so it is a natural fit for a pipeline.

In fact, the `ZPipeline` companion object contains a variety of encoders and decoders for common data formats. So, if you ever need to go from one data format to another, this is the place to look.

Many ZIO ecosystem libraries also provide pipelines for encoding and decoding in formats that are relevant to them. So, for example, with `zio-json`, you can create pipelines for encoding or decoding JSON, or with `zio-schema`, you can create pipelines for encoding and decoding into a variety of formats.

Since pipelines are the “middle” of data process flows, you can compose them with either streams, as discussed above, or with sinks, using the `>>>` operator. By composing a pipeline with a sink, you get a new sink that takes the inputs of the pipeline, transforms them, and then sends them to the sink.

One question that commonly arises when working with pipelines is defining a stream transformation as an operator versus a pipeline. For example, if we want to transform stream elements from integers to strings, we could do it in two ways:

```

1 val ints: ZStream[Any, Nothing, Int] =
2   ZStream(1, 2, 3)
3
4 val toString: ZPipeline[Any, Nothing, Int, String] =
5   ZPipeline.map(_.toString)
6
7 val strings1: ZStream[Any, Nothing, String] =
8   ints.map(_.toString)
9
10 val strings2: ZStream[Any, Nothing, String] =
11   ints >>> toString

```

We can transform these stream elements either using the `map` operator on `ZStream` or by composing the stream with a pipeline. So which should we use?

The answer is that both are equivalent, so to a certain extent, it is a matter of team preference.

However, generally, a good approach is to use concrete operators on streams until a transformation becomes complex enough, or you want to use it in multiple places so that it

makes sense to “factor out” into its own pipeline. The style of composing different stream transformations can be less straightforward than just composing stream operators and can require additional type annotations in some places, so it is good to make sure that there is a “pay off” for this additional complexity.

If you do mainly use stream operators, it is still easy to take advantage of pipelines that provide specific functionality you want using the `ZStream#via` operator with the pipeline you want to take advantage of. In this pipelines just become a convenient “library” of common transformations like encoding and decoding that you can plug into your application.

## 28.3 Conclusion

In this chapter, we looked at sinks and pipelines. Along with streams, these are the three main data types in ZIO Stream, so if you understand how to work with these data types, you are in a great position to work with ZIO Stream.

Streams, pipelines, and sinks correspond to the “beginning”, “middle”, and “end” of any data processing flow and are the fundamental building blocks for creating complex data processing graphs.

Defining each of these data types as their own values lets us define a variety of streams, sinks, and pipelines that do exactly one thing and do it very well. `ZSink.fromFile` knows exactly how to write data from a file and nothing else, `ZStream.fromFile` knows exactly how to read data from a file and nothing else, and `ZPipeline.utfDecode` knows exactly how to decode bytes to strings and nothing else.

By having each of these building blocks and being able to glue them together, we can quickly create powerful solutions to various problems we may encounter.

## Chapter 29

# Streaming: Channels are the Foundation

In the previous two chapters, we learned that streams, sinks, and pipelines are the fundamental data types in ZIO Stream for describing data processing graphs, corresponding to the “beginning”, “middle”, and “end” of a data processing flow. We also learned about various for working with these data types and combining them to build solutions to various problems.

However, while we know how to use streams, sinks, and pipelines, we have been intentionally vague on what they are. We have described a variety of analogies to think about these data types but have not gotten into actual implementation, so you could quickly get started working with streams.

In this chapter, we will change that. We will look at the `ZChannel` data type, which unifies streams, sinks, and pipelines, and see how it serves as the foundation for everything in ZIO Stream.

Channels are low-level, so you will generally be able to use existing operators on streams, sinks, and pipelines. However, if you need to do something unusual, channels will let you do it in a way that will work with everything else in ZIO Stream.

Channels will also give you a deeper understanding of ZIO Stream and the nature of streaming data itself. However, if you want to get straight to additional streaming operators, feel free to skip ahead to the next chapter.

### 29.1 Channels

Thinking about streams, sinks, and pipelines conceptually, it feels that there is something “similar” between them.

A stream produces multiple elements, a sink consumes multiple elements, and a pipeline

both produces and consumes multiple elements. To a certain extent, streams and sinks seem to be two sides of the same coin, and pipelines are somewhere in between.

However, the existing implementations we have sketched out do not reflect that similarity.

Recall that this is a mental model we described for a stream as an effectful iterator:

```
1 trait ZStream[-R, +E, +O] {
2   def process: ZIO[R with Scope, Option[E], Chunk[O]]
3 }
```

This was the mental model we gave for a sink as something we could “push” elements into:

```
1 trait ZSink[-Env, +Err, -In, +Leftover, +Summary] {
2   def push: ZIO[Env with Scope, Nothing, Option[Chunk[In]] => ZIO
3     [
4       Env,
5       (Either[Err, Summary], Chunk[Leftover]),
6       Unit
7     ]]
}
```

The mental model for a pipeline was just a function from stream to stream:

```
1 trait ZPipeline[-R, +E, -I, +O] {
2   def apply[R1 <: R, E1 >: E, I1 <: I, O1 >: O](
3     stream: ZStream[R1, E1, I1]
4   ): ZStream[R1, E1, O1]
5 }
```

All of these look very different! They don’t explain how these different data types could fit together in certain ways that we know make sense, such as composing a pipeline with a sink.

To unify these different data types, we will have to tackle a couple of concepts. The first concept, which is critical to streaming and goes back to our discussion at the very beginning of this section, is the idea of incremental output.

If we recall the type signature of `ZIO[R, E, A]`, it is a workflow that requires a set of services `R`, and if it completes at all either succeeds with exactly one value of type `A` or fails with exactly one `Cause` of type `E`. Because of this, `ZIO` itself is fundamentally incompatible with representing streaming data types because streaming data types can produce multiple values before they complete execution.

Looking at streams, sinks, and pipelines, we can see that both streams and pipelines produce their results incrementally, whereas sinks produce a single summary value. So, what if we create a new data type with two type parameters, one of which represents the incremental output and the other of which represents the final done value?

```
1 trait ZChannel[-Env, +Err, +Elem, +Done]
```

This represents the blueprint for a program that requires a set of services `Env`, potentially produces zero or more values of type `Elem` while it is running, and completes with either a success value of type `Done` or a failure value of type `Err`.

The second concept we will have to tackle is input versus output. Streams only produce values, but sinks and pipelines both consume values, so our type signature will have to reflect the possibility of consuming as well as producing values.

One straightforward solution to this is that we should just create “duals” to each of our output types. This has the appeal of symmetry.

```

1 trait ZChannel[  
2   -Env,  
3   -InErr,  
4   -InElem,  
5   -InDone,  
6   +OutErr,  
7   +OutElem,  
8   +OutDone  
9 ]

```

Clearly, we need to put aside any concerns about too many type parameters for the moment! Aside from that, what does this type signature mean? What does this represent?

One way to think of this is that it is a description of a program that requires a set of services `Env` and accepts zero or more inputs of type `InElem` along with one final input of either type `InErr` or `InDone`. It produces zero or more outputs of type `OutElem` along with one final output of either type `OutErr` or `OutDone`.

Thought of this way, there are actually several natural analogs we can think of for this data type, admittedly without as many type parameters.

One would be low-level channels in libraries like `nio`. These libraries are imperative rather than functional, but channels have a very similar structure where we can write inputs to the channel and eventually “close” the channel, and we can likewise read outputs from the channel and eventually observe its final state.

Another would be the Reactive Streams protocol, where we see very similar concepts of elements and terminal values and elements of the protocol, such as having exactly one terminal value and not being able to write elements after a terminal value.

So, how does this concept of a channel map back to streams, sinks, and pipelines?

Let’s start with streams. A stream is a producer of zero or more values of type `Chunk[A]`. It doesn’t accept any inputs, and it doesn’t produce any terminal value.

We can start by filling in the type parameters we know:

```

1 type ZStream[-R, +E, +A] =  
2   ZChannel[R, ???, ???, ???, E, Chunk[A], ???]

```

We just said that a stream doesn't require any inputs, so we can replace each of the input types with `Any` just like we do for the environment type when we don't require any services. This is a program that doesn't require any input, so we can run it with any input at all, such as the `Unit` value or the number `42`:

```
1 | type ZStream[-R, +E, +A] =
2 |   ZChannel[R, Any, Any, Any, E, Chunk[A], ???]
```

A stream also doesn't produce any meaningful "done" value. Recall from our discussion of a stream as an effectful iterator that we just used `None` to signify that the stream was done, indicating that the stream did not have any meaningful terminal value other than the fact that it was done.

We can model this with the `Unit` value or with the `Any` type. This gets us to the final signature of `ZStream` as follows:

```
1 | type ZStream[-R, +E, +A] =
2 |   ZChannel[R, Any, Any, Any, E, Chunk[A], Any]
```

Let's apply the same logic to sinks. Sinks consume zero or more values of type `In` and produce a single summary value of type `Z`.

```
1 | type ZSink[-R, +E, -In, +L, +Z] =
2 |   ZChannel[R, ???, Chunk[In], ???, E, L, Z]
```

A sink uses an environment of type `R`, consumes elements of type `Chunk[In]`, can potentially emit leftovers of type `L`, and eventually produces either a summary value of type `Z` or fails with an error of type `E`.

How about the `InErr` and `InDone` types? A sink doesn't work with any more information about the done value of the upstream than the fact that it is done, so we can make that type `Any`.

For the `InErr` type, we need to think a little more.

The sink itself doesn't know how to handle any errors, so the error type has to be `Nothing`. If the upstream produces an error, it will have to deal with that itself rather than expect the sink to handle it.

```
1 | type ZSink[-R, +E, -In, +L, +Z] =
2 |   ZChannel[R, Nothing, Chunk[In], Any, E, L, Z]
```

Finally, a pipeline both accepts multiple inputs and produces multiple outputs. It also doesn't know how to handle errors and doesn't accept or produce a meaningful done value:

```
1 | type ZPipeline[-R, +E, -In, +Out] =
2 |   ZChannel[R, Nothing, Chunk[In], Any, E, Chunk[Out], Any]
```

And so we have:

```
1 | type ZStream[-R, +E, +A] =
```

```

2 ZChannel[R, Any, Any, Any, E, Chunk[A], Any]
3
4 type ZSink[-R, +E, -In, +L, +Z] =
5   ZChannel[R, Nothing, Chunk[In], Any, E, L, Z]
6
7 type ZPipeline[-R, +E, -In, +Out] =
8   ZChannel[R, Nothing, Chunk[In], Any, E, Chunk[Out], Any]

```

We now have a deep unification of these different streaming data types. Not only do we understand their similarities and differences, but we understand that they are all just variations of the same thing.

We can also see how this idea of a channel represents a very general notion of computation. We have already seen how ZIO reflects a fundamental idea of computation that requires some context and can fail or succeed.

By introducing the concept of incremental results and input versus output ZChannel takes this even further. So we can, for example, see ZIO itself as a particular type of channel:

```
1 type ZIO[-R, +E, +A] = ZChannel[R, Any, Any, Any, E, Nothing, A]
```

A ZIO is a workflow that requires a set of services R, doesn't require any inputs, doesn't produce any incremental outputs, and either produces a terminal done value A or fails with an error E.

To support providing operators on streams, sinks, and pipelines specific to each of these data types, ZIO does not actually define them as a type alias for a channel but instead defines them as a new data type that wraps a channel.

```

1 final case class ZStream[-R, +E, +A] (
2   channel: ZChannel[R, Any, Any, Any, E, Chunk[A], Any]
3 )

```

So, as a ZIO user, what do channels mean for you?

For the most part, other than hopefully finding this unification interesting, you can just enjoy its benefits.

First, having all streaming data types be channels means channels can be their own data type with a highly optimized execution model, just like ZIO. So stream operators based on channels should be faster.

Second and related to this, having all streaming data types be channels means that many tricky operators, like concurrent merge, only need to be implemented once, resulting in fewer bugs and more maintainable code.

Third, if you do need to do something bespoke that you can't with existing operators on streams, sinks, and channels, you can always drop down to this lower-level channel interface and implement your own operators. And then, since your data type will also be a channel, it will be able to work with all of the existing data types in ZIO Stream.

With that introduction, we'll begin diving into the constructors and operators for channels. This material will help you gain a deeper understanding of streams, but it is only necessary if you want to implement your own custom operators, so feel free to skip ahead to the next chapter if you are just interested in using streams, sinks, and pipelines.

## 29.2 Channel Constructors

Just like ZIO, ZChannel is implemented in terms of a small number of fundamental constructors for creating channels and operators for composing them together.

A rough way to think of the fundamental constructors is in terms of the different type parameters of ZChannel. Each of the output type parameters of ZChannel has a corresponding constructor that outputs a value in the appropriate channel:

```

1 object ZChannel {
2     def fail[E] (
3         cause: Cause[E]
4     ): ZChannel[Any, Any, Any, Any, E, Nothing, Nothing] =
5         ???
6
7     def succeed[Done] (
8         done: Done
9     ): ZChannel[Any, Any, Any, Any, Nothing, Nothing, Done] =
10        ???
11
12    def write[Out] (
13        out: Out
14    ): ZChannel[Any, Any, Any, Any, Nothing, Out, Any] =
15        ???
16 }
```

As you can see, the names of these constructors match the ones on ZIO with the exception of `write`, which writes an element to the output channel and does not have a corresponding method on ZIO. You can also see here that we use `Any` for the input channels and `Nothing` for the output channels if they are not being used.

The fundamental constructor with respect to the input channels is a single operator called `ZChannel.readWith` that reads an input and executes a new channel based on the result:

```

1 object ZChannel {
2     def readWith[
3         Env,
4         InErr,
5         InElem,
6         InDone,
7         OutErr,
8         OutElem,
```

```

9     OutDone
10    ](
11      in: InElem => ZChannel[
12        Env,
13        InErr,
14        InElem,
15        InDone,
16        OutErr,
17        OutElem,
18        OutDone
19      ],
20      error: InErr => ZChannel[
21        Env,
22        InErr,
23        InElem,
24        InDone,
25        OutErr,
26        OutElem,
27        OutDone
28    ],
29    done: InDone => ZChannel[
30      Env,
31      InErr,
32      InElem,
33      InDone,
34      OutErr,
35      OutElem,
36      OutDone
37    ]
38  ): ZChannel[
39    Env,
40    InErr,
41    InElem,
42    InDone,
43    OutErr,
44    OutElem,
45    OutDone
46  ] =
47  ???
48 }

```

Channels are pull-based, so when we read, we might get either an `InElem` or a terminal `InDone` or `InErr` value, so `readWith` requires us to handle all of those possibilities.

The final fundamental channel constructors allow us to import ZIO workflows into the `ZChannel` world so we can work with them:

```

1 object ZChannel {
2   def fromZIO[Env, Err, Done](
3     zio: ZIO[Env, Err, Done]
4   ): ZChannel[Env, Any, Any, Any, Err, Nothing, Done] =
5   ????
6 }
```

The `ZChannel.fromZIO` constructor just lets us convert a `ZIO` workflow into a `ZChannel`. As discussed above, a `ZIO` does not require any inputs and does not produce any incremental outputs, so we use `Any` for the input channels and `Nothing` for the output channels.

```

1 object ZChannel {
2   def scoped[Env, Err, Out](
3     zio: ZIO[Env with Scope, Err, Out]
4   ): ZChannel[Env, Any, Any, Any, Err, Out, Nothing] =
5   ????
6 }
```

The `ZChannel.scoped` constructor imports a scoped `ZIO` into the channel world, removing the `Scope` as a requirement and subsuming it in the `Scope` of the channel. Notice that the result of the scoped `ZIO` is written to the output channel, we will discuss scopes and channels in greater detail in the next section on channel operators.

These are really the fundamental constructors of channels. There are a variety of other constructors, typically corresponding to the ones on `ZIO`, allowing you to construct more specialized channels, but these will give you the tools to do everything you need to do if you are working with channels.

## 29.3 Channel Operators

Along with these fundamental constructors, there are also a set of fundamental operators for composing channels together into more complex channels.

The first one, and one that should be familiar to you, is `ZChannel#foldChannel`, which is like `foldZIO` and lets you run one channel and then run another channel after the first one is done based on its result.

```

1 trait ZChannel[
2   -Env,
3   -InErr,
4   -InElem,
5   -InDone,
6   +OutErr,
7   +OutElem,
8   +OutDone
9 ] {
```

```

10  final def foldChannel[
11    Env1 <: Env,
12    InErr1 <: InErr,
13    InElem1 <: InElem,
14    InDone1 <: InDone,
15    OutErr1,
16    OutElem1 >: OutElem,
17    OutDone1
18  ](
19    onErr: OutErr => ZChannel[
20      Env1,
21      InErr1,
22      InElem1,
23      InDone1,
24      OutErr1,
25      OutElem1,
26      OutDone1
27    ],
28    onSucc: OutDone => ZChannel[
29      Env1,
30      InErr1,
31      InElem1,
32      InDone1,
33      OutErr1,
34      OutElem1,
35      OutDone1
36    ]
37  ): ZChannel[
38    Env1,
39    InErr1,
40    InElem1,
41    InDone1,
42    OutErr1,
43    OutElem1,
44    OutDone1
45  ] =
46  ???
47 }

```

Notice that all of the other type parameters need to be unified. For example, both the original channel and the new channel can write elements, so the new channel will be able to write elements of either the output element type of the original channel or the output element type of the new channel.

Just like with ZIO, we can implement a variety of powerful operators in terms of ZChannel #foldChannel such as flatMap and \*>.

Let's see how we can put these together to implement a simple channel that maps elements while leaving errors and done values unchanged:

```

1 def map[Err, In, Out, Done](
2   f: In => Out
3 ): ZChannel[Any, Err, In, Done, Err, Out, Done] =
4   ZChannel.readWith[Any, Err, In, Done, Err, Out, Done](
5     in => ZChannel.write(f(in)) *> map[Err, In, Out, Done](f),
6     err => ZChannel.fail(err),
7     done => ZChannel.succeed(done)
8   )

```

This reflects a very common pattern where we read some input, process it, write some output, and recurse.

The next fundamental operator is related to the fact that channels have both input and output types and is called `ZChannel#pipeTo`. It essentially “pipes” all the outputs from one channel to the inputs of another channel:

```

1 trait ZChannel[
2   -Env,
3   -InErr,
4   -InElem,
5   -InDone,
6   +OutErr,
7   +OutElem,
8   +OutDone
9 ] {
10   final def pipeTo[Env1 <: Env, OutErr2, OutElem2, OutDone2](
11     that: => ZChannel[
12       Env1,
13       OutErr,
14       OutElem,
15       OutDone,
16       OutErr2,
17       OutElem2,
18       OutDone2
19     ]
20   ): ZChannel[
21     Env1,
22     InErr,
23     InElem,
24     InDone,
25     OutErr2,
26     OutElem2,
27     OutDone2
28   ] =
29   ????

```

30 }

You can think of this a bit like function composition for channels. The output types of the first channel have to line up with the input types of the second channel, and the first channel becomes the “upstream” while the second channel becomes the “downstream”.

The `ZChannel#pipeTo` operator has a symbolic alias `>>>` and it serves much the same purpose as the `>>>` we saw on streams, sinks, and pipelines before, sending all of the inputs through the given channel. In fact, all of the operators for combining streams, sinks, and pipelines are implemented in terms of `ZChannel#pipeTo`.

The next fundamental operator is `ZChannel#concatMap`, and you can think of this as somewhat like `flatMap` but for the `Out` channel instead of the `Done` channel. The signature looks like this:

```

1 trait ZChannel[
2   -Env,
3   -InErr,
4   -InElem,
5   -InDone,
6   +OutErr,
7   +OutElem,
8   +OutDone
9 ] {
10   final def concatMap[
11     Env1 <: Env,
12     InErr1 <: InErr,
13     InElem1 <: InElem,
14     InDone1 <: InDone,
15     OutErr1 >: OutErr,
16     OutElem2
17   ](
18     f: OutElem => ZChannel[
19       Env1,
20       InErr1,
21       InElem1,
22       InDone1,
23       OutErr1,
24       OutElem2,
25       Any
26     ]
27   )(implicit trace: Trace): ZChannel[
28     Env1,
29     InErr1,
30     InElem1,
31     InDone1,
32     OutErr1,
```

```

33     OutElem2,
34     Any
35   ] =
36   ????
37 }
```

Conceptually, `ZChannel#concatMap` takes an existing channel, maps each of the elements output by that channel to a new channel, and then sequentially runs those channels and emits their elements. `ZChannel#concatMap` is an important operator because `ZChannel#flatMap` is implemented in terms of `ZChannel#concatMap`.

The final fundamental operator for combining channels is `ZChannel#mergeWith`. This is the operator that introduces concurrency into the world of channels:

```

1 trait ZChannel[
2   -Env,
3   -InErr,
4   -InElem,
5   -InDone,
6   +OutErr,
7   +OutElem,
8   +OutDone
9 ] {
10   final def mergeWith[
11     Env1 <: Env,
12     InErr1 <: InErr,
13     InElem1 <: InElem,
14     InDone1 <: InDone,
15     OutErr2,
16     OutErr3,
17     OutElem1 >: OutElem,
18     OutDone2,
19     OutDone3
20   ](
21     that: ZChannel[
22       Env1,
23       InErr1,
24       InElem1,
25       InDone1,
26       OutErr2,
27       OutElem1,
28       OutDone2
29     ]
30   )(
31     leftDone: Exit[OutErr, OutDone] => ZChannel.MergeDecision[
32       Env1,
33       OutErr2,
```

```

34     OutDone2,
35     OutErr3,
36     OutDone3
37   ],
38   rightDone: Exit[OutErr2, OutDone2] => ZChannel.MergeDecision[
39     Env1,
40     OutErr,
41     OutDone,
42     OutErr3,
43     OutDone3
44   ]
45   ): ZChannel[
46     Env1,
47     InErr1,
48     InElem1,
49     InDone1,
50     OutErr3,
51     OutElem1,
52     OutDone3
53   ] =
54   ???
55 }
```

`ZChannel!mergeWith` runs two channels concurrently and merges their outputs. Both channels are run at the same time, and both have the ability to read from the upstream and write to the downstream.

Once one of the channels is done, what happens next is controlled by the `MergeDecision`, which is an algebraic data type that describes either terminating the slower channel or waiting for the slower channel to complete and doing something with its result. It is similar in this way to the `raceWith` operator on `ZIO`, which is the basis of most concurrent operators on `ZIO`.

In addition to these, one other fundamental operator is `ZChannel#ensuring`, which allows adding a finalizer that will be run after a channel is done:

```

1 trait ZChannel[
2   -Env,
3   -InErr,
4   -InElem,
5   -InDone,
6   +OutErr,
7   +OutElem,
8   +OutDone
9 ] {
10   final def ensuring[Env1 <: Env](
11     finalizer: => URIO[Env1, Any]
```

```

12   ): ZChannel[
13     Env1,
14     InErr,
15     InElem,
16     InDone,
17     OutErr,
18     OutElem,
19     OutDone
20   ] =
21   ???
22 }
```

We will be discussing finalizers and channel scopes more in the next section.

## 29.4 Channel Scopes

One topic that requires some discussion when working with channels is finalizers and channels.

A ZIO workflow never produces any incremental output, so when we use an operator like `ZIO#ensuring`, it is clear that the finalizer should be run immediately after ZIO completes execution and before any subsequent workflow is run. For example, consider the below code:

```

1 zio1.ensuring(finalizer).flatMap(_ => zio2)
```

It is clear that assuming `zio1` begins execution, `finalizer` will be run immediately after `zio1` completes execution and before `zio2` begins execution. ZIO represents a “static” scope in the sense that as soon as we write `zio1.ensuring(finalizer)`, no further operator can delay the `finalizer` from executing.

Now consider how this applies in the context of channels:

```

1 channel1.ensuring(finalizer).flatMap(_ => channel2)
2 channel1.ensuring(finalizer).pipeTo(channel2)
3 channel1.ensuring(finalizer).concatMap(_ => channel2)
4 channel1.ensuring(finalizer).mergeWith(channel2)
```

When should `finalizer` be run in each of these cases?

The basic answer is that `finalizer` should be run immediately after `channel1` completes execution, either by succeeding with a `Done` value or failing with an `Err`. This rule is consistent with ZIO, but we may need to think a bit about how it applies to these different operators.

In the first case, `ZChannel#flatMap` is not executed until `channel1` completes execution because `flatMap` needs the `Done` value of `channel1` to determine what to do next, so `finalizer` will run immediately after `channel1` completes execution and be-

fore `channel12` begins execution. This is why we sometimes say that `ZChannel1#flatMap` “closes” the scope of the channel.

In the second case, `ZChannel#pipeTo`, `channel1` is not done until `channel12` has finished reading from it, so while `channel12` is processing elements emitted by `channel1`, it can rely on ‘finalizer’ not being run yet. This reflects another important principle: we don’t want to run finalizers associated with an upstream channel while a downstream channel is still processing elements from that channel.

Similarly, in the third case, `ZChannel#concatMap`, `channel12` needs to be run before `finalizer` is run. Since `ZStream#flatMap` is implemented in terms of `ZChannel.concatMap`, streams have what we sometimes call a “dynamic scope” where in `stream1.ensuring.flatMap(_ => stream2)` the finalizer will not be run until `stream2` completes execution since the stream is not really “done” until the processing of its downstream is done.

Finally, in the fourth case, `ZChannel#mergeWith`, the finalizer will again be deferred to the scope of the merged stream because even though `channel1` is done emitting elements someone downstream may be processing those elements.

While this can require some thought, in general, it just does what you expect, and especially if you are working with higher-level data types such as streams, sinks, and pipelines, you shouldn’t have to worry about it. If you are directly working with channels, the most important thing to keep in mind is that `flatMap` and `foldChannel` do something “after” a channel has done, and so any resources associated with the original channel will be closed by the time the continuation is evaluated.

## 29.5 Conclusion

This chapter has introduced you to `ZChannel`, the underlying implementation of streams, sinks, and pipelines. We have seen how channels allow us to compose the different streaming types together in a rich way while supporting a highly performant implementation.

We also learned about some of the fundamental constructors and operators on `ZChannel`, so you know where to start if you need to implement your own custom channel operators.

However, for the most part, you should be able to work directly with streams, sinks, and pipelines. Armed with this knowledge, we’ll go back in the next chapter to look at more stream operators, focusing on different ways to transform and combine streams.

## 29.6 Exercises

1. Use `ZChannel` to implement a function that takes two lower and upper bounds of the range and returns a stream that emits all the numbers in that range:

```
1 | def range(start: Int, end: Int): ZStream[Any, Nothing, Int]
2 |   =
2 |   ???
```

2. Try to implement the `ZSink.collectAll` sink by yourself using `ZChannel`:

```
1 def collectAll[A]: ZSink[Any, Nothing, A, Nothing, Chunk[A]]  
2     =  
3     ???
```

3. Try to implement the `ZPipeline.dropWhile` pipeline using `ZChannel`:

```
1 def dropWhile[A](f: A => Boolean): ZPipeline[Any, Nothing, A  
2     , A] =  
3     ???
```

# Chapter 30

## Streaming: Transforming Streams

ZIO Streams, built on the robust foundation of the ZIO ecosystem, offers a rich set of transformation operations that empower developers to construct sophisticated data processing workflows. These transformations range from simple operations like mapping and filtering to more complex manipulations like stateful transformation, grouping, distributing, and broadcasting.

Whether you're building real-time analytics systems, reactive user interfaces, or data-intensive applications, the concepts and techniques covered in this chapter will equip you with the tools to solve complex streaming challenges using ZIO Streams elegantly.

Let's dive in!

### 30.1 Mapping

Mapping operations are fundamental to stream processing, allowing us to transform the elements of a stream as they flow through our system. In ZIO Streams, mapping operations can be broadly categorized into stateless and stateful mapping. Each type offers different capabilities and is suited to different scenarios in stream processing.

- **Simple Transformation** (e.g., `map`, `mapZIO`): These operations apply a function to each element of the stream, producing a new stream with transformed elements. The transformation can be either a pure or effectful operation.
- **Chunk-based Mapping** (e.g., `mapChunks`, `mapChunksZIO`): Instead of operating on individual elements, these functions work on chunks of data. This can be more efficient for operations that benefit from processing multiple elements at once.
- **Flattening Concatenation** (e.g., `mapConcat`, `mapConcatChunk`): These operations allow us to transform a single element into multiple elements, effectively flattening nested structures within the stream. Assume you have a function that returns

a list of elements for each input element. Without `mapConcat`, you will end up with a stream of lists, but with `mapConcat`, all the intermediate lists will be flattened into a single stream.

- **Splitting** (e.g., `split`, `splitOnChunk`): These operations split the elements of the stream based on a given predicate or a delimiter and return a stream of chunks.
- **Mapping Errors** (e.g., `mapError`, `mapErrorCause`): These operations allow us to map errors represented in the stream.
- **Stateful Mapping** (e.g., `mapAccum`, `sliding`, `scan`, `scanZIO`, `zipWithNext`, `zipWithPrevious`): The fundamental operator of these operations is `mapAccum`, which introduces the concept of maintaining and evolving state across the stream's lifecycle, enabling a wide range of use cases such as sliding window computations, scanning, running calculations, stateful parsing, data deduplication, session handling, creating histograms, sensitization, outlier detection, bloom filter, or even simple ones like filtering, dropping, or taking elements from the stream.
- **Concurrent Mapping** (e.g., `mapZIOPar`, `mapZIOParUnordered` and `mapZIOParByKey`): These operations allow for parallel processing of stream elements, introducing additional complexity in terms of ordering and resource management.

Many of these operators are easy to use and understand, but they can be combined in powerful ways to create complex stream processing pipelines. Among these operators, let's discuss two important ones in more detail: stateful and concurrent mapping.

### 30.1.1 Stateful Mapping

Stateful mapping operations, represented primarily by `mapAccum` and `mapAccumZIO`, introduce the concept of maintaining and evolving state across the stream's lifecycle. These operations are particularly powerful for scenarios where the transformation of an element depends not just on the element itself but on the history of the stream:

```

1 trait ZStream[-R, +E, +A] {
2   def mapAccum[S, A1](s: => S)(
3     f: (S, A) => (S, A1)
4   ): ZStream[R, E, A1] = ????
5 }
```

The `mapAccum` operation can be understood through the lens of automata theory, where the state of the automation is updated based on the initial state and a transition function that produces the next state and the output value. This allows us to accumulate state across the stream processing, enabling various use cases such as sliding window computations, running calculations (e.g., running average), stateful parsing, data deduplication, session handling, and more.

Let's look at an example of using `mapAccum` to calculate the running average of a stream of numbers:

```

1 import zio._
2 import zio.stream._

3
```

```

4 case class AverageState(sum: Double, count: Int)
5
6 val numbers: ZStream[Any, Nothing, Int] = ZStream(1, 2, 3, 4, 5)
7 val runningAverage = numbers.mapAccum(AverageState(0, 0)) { (
8     state, num) =>
9     val newSum = state.sum + num
10    val newCount = state.count + 1
11    val newAvg = newSum / newCount
12    (AverageState(newSum, newCount), newAvg)
13 }
14

```

### 30.1.2 Concurrent Mapping

ZIO Streams also provides concurrent mapping operations (e.g., `mapZIOPar`), allowing for parallel stream element processing. These operations introduce additional complexity in terms of ordering and resource management but can significantly improve performance for I/O-bound or CPU-intensive transformations:

```

1 import zio._
2 import zio.stream._
3 import java.io.IOException
4
5 case class File(name: String, size: Int)
6 object File {
7     def process(file: File): ZIO[Any, IOException, Unit] = ????
8 }
9
10 val files = List(
11     File("doc1.txt", 100),
12     File("image1.jpg", 500),
13     File("doc2.txt", 150),
14     File("video.mp4", 1000),
15     File("doc3.txt", 200)
16 )
17
18 ZStream
19     .fromIterable(files)
20     .mapZIOPar(3)(File.process) // Process up to 3 files
21         .concurrently
22         .runDrain

```

When using concurrent mapping, it's essential to consider the following implications:

- **Ordering Guarantees:** Some operations preserve the original order of elements, while others may reorder elements for improved performance:
  - `mapZIOPar` preserves the original element order, which is crucial for

- sequence-dependent operations.
- `mapZIOParUnordered` may reorder elements, potentially offering higher throughput at the cost of unpredictable output order.
  - `mapZIOParByKey` maintains order within partitions but may reorder elements across different partitions.
- **Concurrency Control:** Most methods allow specifying the level of concurrency, the `n` parameter, and the maximum number of elements to map in parallel, except for `mapZIOParByKey`, which is determined by the number of different key types.
  - **Chunking Structure:** The stream's chunking structure is destroyed when using concurrent mapping operations. Since you might lose some of the performance benefits of a well-chunked stream, it is recommended that you re-chunk the stream after concurrent mapping.
  - **Buffering and Backpressure:** ZIO Stream is pull-based, meaning that the downstream controls when elements are pulled from the stream. When using concurrent mapping, it is essential to ensure that the downstream can handle the increased rate of elements being produced. Excessive parallelism can lead to a bottleneck at the downstream, causing backpressure to propagate upstream. Adjusting the buffer size allows you to control the number of elements that can be buffered before backpressure is applied. By default, all concurrent map operations use a buffer of 16 elements, by default, but you can adjust this value using the `bufferSize` parameter.

## 30.2 Transform and Combine

The `flatMap` is a robust operation that allows you to transform each element of a stream into a new stream and then concatenate all these new streams in order. This operation is handy when you need to perform a transformation that results in multiple elements for each input element.

Assume we have a stream of user IDs and a function that retrieves the orders for a given user ID in the form of a stream. We want to process all the orders sequentially. In this case, we can use the `flatMap` operator:

```

1 val userIds: ZStream[Any, Nothing, Int] = ???
2 def userOrders(userId: Int): ZStream[Any, Nothing, Order] = ???
3 val allOrders = userIds.flatMap(userOrders)
4 val processOrder: Order => ZIO[Any, Nothing, Unit] = ???
5 val result = allOrders.foreach(processOrder)

```

In this example, `userIds` is a stream of user IDs, and `userOrders` is a function that retrieves the orders for a given user ID. By using `flatMap`, we transform each user ID into a stream of orders and concatenate all these streams into a single stream of orders. Finally, we process each order using the `processOrder` function.

Based on the nature of the transformation, sequential or concurrent, the `flatMap` operator has three variants:

1. **flatMap**: It transforms each element into a new stream and concatenates them in the original order. It processes all the streams in sequence, preserving the order of elements. Use this operator when the order of elements is essential, e.g., when processing time-series data, transaction processing in financial systems, event sourcing in event-driven architectures, etc.
2. **flatMapPar**: Runs inner streams concurrently up to the given level of parallelism; it doesn't proceed to the next element until one of the inner streams is completed. This operation doesn't preserve the order of the elements. Use this operator when you want to increase throughput, but the order of elements is not essential. For example, assume you have a stream of documents and want to process each document through multiple processing pipelines concurrently.
3. **flatMapParSwitch**: Executes inner streams concurrently up to a given level of parallelism. When a new element arrives, it cancels the oldest executing inner stream and starts a new one to process the incoming element. This functionality is proper when you want to prioritize processing newer elements over older ones, such as in real-time data processing scenarios where the latest information is most relevant. Like the `flatMapPar` operator, it doesn't preserve the order of elements. For example, you can use this operator to handle incoming sensor data, canceling the processing of older data when new readings arrive. This ensures that critical decisions (e.g., equipment shutdown) are based on the most current information.

### 30.3 Flattening

Flattening nested streams is a common operation in stream processing, transforming a stream of streams into a single, coherent stream of elements. This operation is closely related to the `flatMap` function. In ZIO Stream, we can implement `flatten` in terms of `flatMap`, e.g., `nestedStreams.flatMap(identity)`, which concatenates all the inner streams into a single stream:

```

1 val nestedStreams: ZStream[Any, Nothing, ZStream[Any, Nothing,
  Int]] =
2   ZStream(
3     ZStream(1, 2, 3),
4     ZStream(4, 5, 6),
5     ZStream(7, 8, 9)
6   )
7 val flattenedStream: ZStream[Any, Nothing, Int] =
8   nestedStreams.flatten

```

The same goes for the parallel variant of flattening, `flattenPar` and `flattenParUnbounded`, which are in terms of `flatMapPar`, e.g., `nestedStreams.flatMapPar(32)(identity)`. These parallel variants are helpful when you have a stream of streams and want to flatten them concurrently:

```

1 def getURLs: ZStream[Any, Nothing, URL] = ???
2 def fetchURL(url: URL): ZStream[Any, Nothing, Page] = ???

```

```
3 | getURLs.map(fetchURL).flattenPar(32)
```

Variants that are useful for flattening streams of Chunk, Exit, Iterable, and Take data types include flattenChunks, flattenExit, flattenExitOption, flattenIterable, and flattenTake:

For example, to flatten a stream of Chunk:

```
1 | val chunkStream: ZStream[Any, Nothing, Chunk[Int]] =
2 |   ZStream(Chunk(1, 2), Chunk(3, 4), Chunk(5, 6))
3 |
4 | val flattenedStream: ZStream[Any, Nothing, Int] =
5 |   chunkStream.flattenChunks
6 |
7 | flattenedStream.runCollect // Output: Chunk(1, 2, 3, 4, 5, 6)
```

## 30.4 Filtering Operators

Filtering is another fundamental operation in stream processing, allowing us to selectively include or exclude elements from a stream based on specific criteria. Let's explore the various filtering operations available in ZIO Streams:

- **Predicate-based Filtering** (e.g., filter, filterZIO, filterNot): These operations include or exclude elements based on a predicate function. The predicate can be either a pure function or an effectful function.
- **Find Operators** (e.g., find, findZIO): These operations allow us to find the first element that satisfies a given predicate. The predicate can be either a pure function or an effectful function.
- **Take and Drop Operators** (e.g., take, takeWhile, takeWhileZIO, takeUntilZIO, takeUntil, takeRight, drop, dropRight, dropWhile, dropUntil, dropWhileZIO): These operations allow us to include or exclude elements from the stream based on their position or a condition.
- **Changes Detection** (e.g., changes, changesZIO, and changesWith): These operations detect changes in the stream and only emit elements that differ from their predecessor.

## 30.5 Collecting (Filter-map)

Assume you have a stream of elements, and you want to filter even numbers and double them; you can combine filter and map operations to achieve this:

```
1 | // Approach 1: Separate filter and map
2 | val result1 = stream.filter(_ % 2 == 0).map(_ * 2)
```

However, ZIO Streams provides a more concise way to achieve this using the collect operator, which combines filtering and mapping into a single step:

```

1 // Approach 2: Using collect
2 val result2 = stream.collect { case n if n % 2 == 0 => n * 2 }

```

The collecting operators have two variants: non-early termination and early termination.

- **Non-early Termination** (e.g., `collect`): Processes all stream elements and transforms them into those that satisfy the predicate.
- **Early Termination** (e.g., `collectWhile`): This method stops processing the stream as soon as an element fails the predicate. It is useful when finding the first matching element or processing until a specific condition is met.

Each of these two variants has a corresponding operator for different types of elements. For collecting `Either` values, you can use `collectLeft`, `collectRight`, or `collectWhileLeft` and `collectWhileRight`. To collect `Option` values, you can use `collectSome` or `collectWhileSome`. To collect `Exit` values, you can use `collectSuccess` or `collectWhileSuccess`.

## 30.6 Grouping

We have two primary types of grouping operations:

- **Grouping into Stream of Chunks**: These operations collect elements into chunks, resulting in an output stream that emits chunks of elements instead of individual elements. Grouping can be beneficial for batch processing, windowing, or buffering elements for further processing.
- **Grouping into Stream of Streams**: These operations split the stream into multiple substreams based on a given criterion, taking a callback function that transforms each substream into a new stream.

Let's take a closer look at these two types of operations.

### 30.6.1 Grouping into Stream of Chunks

We have three primary grouping operations in ZIO Streams based on the criteria for grouping elements:

- Fixed-size Grouping, e.g. `grouped`
- Time-limited Grouping, e.g. `groupedWithin`
- Adjacent Grouping, e.g. `groupAdjacentBy`

Let's explore each one in more detail.

#### 30.6.1.1 Fixed-size Grouping

The `grouped` operator groups elements into fixed-size chunks. For example, if you want to group elements into chunks of size 3, you can use the `grouped(3)` operator:

```

1 val stream = ZStream(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
2 val groupedStream = stream.grouped(3)
3 // will emit: Chunk(1, 2, 3), Chunk(4, 5, 6), Chunk(7, 8, 9),
   Chunk(10)

```

### 30.6.1.2 Time-limited Grouping

The `groupedWithin` operator groups elements into chunks based on a fixed-size chunk. If the chunk is not filled within a specific time window, it emits the chunk. This is useful for cases where you want to process elements in batches but don't want to wait indefinitely for the batch to be filled.

Assume you have a stream of sensor data points, and you need to balance between efficient batch processing and timely data analysis; this operator helps you to achieve this:

```

1 import java.time.Instant
2
3 case class SensorReading(sensorId: String, temperature: Double,
4                           timestamp: Instant)
5
6 def sensorStream: ZStream[Any, Nothing, SensorReading] = ????
7
8 def processBatch(readings: Chunk[SensorReading]): String = {
9   val avgTemp = readings.map(_.temperature).sum / readings.size
10  f"Processed ${readings.size} readings. Avg Temp: $avgTemp%.2f°C
11 "
12 }
13
14 sensorStream
15  .groupedWithin(50, 5.seconds)
16  .map(processBatch)
17  .foreach(Console.printLine(_))

```

### 30.6.1.3 Adjacent Grouping

The `groupAdjacentBy` operator groups together adjacent elements based on a given predicate. This is useful when you need to group consecutive elements in a stream based on shared characteristics, e.g., detecting changes and grouping time-series data points that fall within the same window.

Assume you have a stream of logs with different log levels, and you want to group adjacent logs with the same log level:

```

1 import java.time.LocalDateTime
2
3 case class LogEntry(timestamp: LocalDateTime, level: String,
4                      message: String)
5
6 val logEntries = List(
7   ("2024-01-01T10:00:00", "INFO", "Application started"),
8   ("2024-01-01T10:01:00", "INFO", "User logged in"),
9   ("2024-01-01T10:02:00", "WARN", "High memory usage"),
10  ("2024-01-01T10:02:30", "WARN", "CPU threshold exceeded"),

```

```

10   ("2024-01-01T10:03:00", "ERROR", "Database connection failed"),
11   ("2024-01-01T10:03:10", "ERROR", "Retrying database connection"
12     ),
13   ("2024-01-01T10:04:00", "INFO", "Database connection restored")
14   ,
15   ("2024-01-01T10:05:00", "INFO", "Normal operation resumed")
16 )
17
18 val logStream =
19   ZStream.fromIterable(
20     logEntries.map { case (time, level, message) =>
21       LogEntry(LocalDateTime.parse(time), level, message)
22     }
23   )

```

You can use the `groupAdjacentBy` operator to group adjacent logs with the same log level:

```

1 import java.time.format.DateTimeFormatter
2
3 def generateReport(group: (String, NonEmptyChunk[LogEntry])):
4   String = {
5     val (level, entries) = group
6     val formatter = DateTimeFormatter.ofPattern("HH:mm:ss")
7     s"$level (${entries.size} entries):\n" +
8     entries.map(e =>
9       s"  ${e.timestamp.format(formatter)} - ${e.message}"
10      .mkString("\n"))
11  }
12
13 val groupedLogs =
14   logStream
15     .groupAdjacentBy(_.level)
16     .map(generateReport)
17     .foreach(Console.println(_)) *>
18       Console.println("Log processing completed.")

```

When you run this program, you'll see the following output:

```

1 INFO (2 entries):
2   10:00:00 - Application started
3   10:01:00 - User logged in
4 WARN (2 entries):
5   10:02:00 - High memory usage
6   10:02:30 - CPU threshold exceeded
7 ERROR (2 entries):
8   10:03:00 - Database connection failed

```

```

9   10:03:10 - Retrying database connection
10 INFO (2 entries):
11   10:04:00 - Database connection restored
12   10:05:00 - Normal operation resumed
13 Log processing completed.

```

### 30.6.2 Grouping into Stream of Streams

Let's start with the `groupByKey` operator, which groups elements based on a key extracted from each element:

```

1 trait ZStream[-R, +E, +A] {
2   def groupByKey[K](
3     f: A => K,
4     buffer: => Int = 16
5   ): ZStream.GroupBy[R, E, K, A] = ????
6 }

```

The `groupByKey` operator takes a function from elements to keys, `f: A => K`, and returns a `GroupBy` object. This allows you to filter over the grouped inner streams and transform them into a new stream concurrently by applying the transformation function to each group; finally, it will merge all the inner streams in a non-deterministic fashion into a single stream and return it.

Assume you have a stream of sales records:

```

1 case class SaleRecord(product: String, category: String, amount:
2   Double)
3
3 val salesStream: ZStream[Any, Nothing, SaleRecord] = ZStream(
4   SaleRecord("Laptop", "Electronics", 1200.0),
5   SaleRecord("T-shirt", "Clothing", 25.0),
6   SaleRecord("Phone", "Electronics", 800.0),
7   SaleRecord("Jeans", "Clothing", 50.0),
8   SaleRecord("Tablet", "Electronics", 300.0),
9   SaleRecord("Shoes", "Clothing", 80.0)
10 )

```

You want to create a report for each sales category in parallel. First, we need to write a stream transformation that maps a stream of sales records into a stream of group reports:

```

1 case class GroupReport(category: String, total: Double, count:
2   Int) {
3   def average: Double = if (count > 0) total / count else 0
4 }
4 object GroupReport {
5   def empty = GroupReport("", 0, 0)
6 }

```

```

7
8 def generateReport(
9     records: ZStream[Any, Nothing, SaleRecord]
10    ): ZStream[Any, Nothing, GroupReport] =
11     records.scan(GroupReport.empty) { (state, record) =>
12       GroupReport(
13         category = record.category,
14         total = state.total + record.amount,
15         count = state.count + 1
16       )
17     }
18     .takeRight(1) // Take the last state

```

As we only need the last state, we take the last state using the `takeRight(1)` operator. Now, we are ready to group the sales records by category and generate reports for each category:

```

1 for {
2   results <- salesStream
3     .groupByKey(_.category) {
4       case (_, groupStream) =>
5         generateReport(groupStream)
6     }
7     .runCollect
8   <- ZIO.debug("Sales Summary by Category:")
9   <- ZIO.foreach(results) { state =>
10     ZIO.debug(
11       f"${state.category}: Total Sales = ${state.total}%,.2f,
12       Average = ${state.average}%,.2f"
13     )
14   }
15 } yield ()

```

Please note that we used the same `generateReport` for all the groups in this example. However, you can use different transformation functions for each group.

The `groupBy` operator is more advanced than `groupByKey`, as it allows you to group elements based on an effectful function. This can be useful when the grouping operation itself is effectful, such as reading from a database or making an API call to determine the group key:

```

1 trait ZStream[-R, +E, +A] {
2   def groupBy[R1 <: R, E1 >: E, K, V](
3     f: A => ZIO[R1, E1, (K, V)],
4     buffer: => Int = 16
5   ): ZStream.GroupBy[R1, E1, K, V]
6 }

```

## 30.7 Partitioning

These operations split the stream into multiple sub-streams based on a given criterion. Partitioning is useful for routing elements to different processing pipelines, performing parallel processing, or distributing elements based on a key.

Assume you have a stream of financial transactions and a pipeline that flags potential fraudulent transactions. While processing these suspicious transactions may take longer than regular transactions, you don't want to block the processing of other transactions. You can partition the stream and process each partition in parallel:

```

1 | for {
2 |   partitioned <- transactionStream.partition(
3 |     isPotentiallyFraudulent, buffer = 50)
4 |     (suspiciousStream, normalStream) = partitioned
5 |
6 |     // Process streams concurrently
7 |     _ <- (suspiciousStream.mapZIO(processSuspiciousTransaction)
8 |       mergeSorted
9 |       normalStream.mapZIO(processNormalTransaction)).runDrain
10 |   } yield ()

```

The speed of `transactionStream` is controlled by the slowest partition, so introducing a proper buffer size is essential to avoid blocking the processing of the other partition.

## 30.8 Broadcasting and Distributing

In stream processing, **broadcasting** and **distributing** are two fundamental operations that enable efficient handling of data flows across multiple consumers or stages of a pipeline:

- **Broadcasting** allows you to replicate a stream into multiple independent streams, each of which can be processed separately. It is helpful for scenarios where you need to apply different transformations to the same data or feed the same data to multiple consumers.
- **Distributing**, on the other hand, refers to partitioning or splitting the stream among multiple consumers, where each consumer receives only a subset of the data. This often balances the load across several workers or nodes in a distributed system. Unlike broadcasting, where all consumers receive the same data, in distributing, each consumer handles only a portion of the total data stream, improving scalability and performance.

Assume you have a stream of stock tick data and want to broadcast it to three different consumers: a dashboard updater, a database storage, and a price alert checker. You can use the `broadcast` operator to achieve this:

```

1 | for {

```

```

2 broadcastStreams <- stockTickStream.broadcast(3, 16)
3 dashboardStream    = broadcastStreams(0)
4 databaseStream     = broadcastStreams(1)
5 alertStream        = broadcastStreams(2)
6 - <- dashboardStream.foreach(updateDashboard) zipPar
7         databaseStream.foreach(storeInDatabase) zipPar
8         alertStream.foreach(checkPriceAlerts)
9 } yield ()

```

The `broadcast` takes the number of consumers as the first parameter and the buffer size as the second. The buffer size determines how many elements can be buffered before backpressure is applied by a slow consumer. It means that if one of the consumers, e.g., `dashboardStream`, is slowed down for a while, the other consumers, e.g., `databaseStream` and `alertStream`, can continue processing without being blocked by filling up the buffer. As soon as the `dashboardStream` catches up, it will start processing the buffered elements. If the buffer size becomes full, the upstream will be backpressured.

The `broadcast` has two types of variants:

- **Dynamic-sized Consumers** (`broadcastDynamic`, `broadcastedQueuesDynamic`): Whenever the number of consumers is not known upfront, you can use these operators.
- **Fixed-sized Consumers** (`broadcast`, `broadcastedQueues`): They take the number of consumers as a parameter and create a fixed number of consumers.

Now, let's explore how to distribute a stream among multiple consumers. Assume you have a stream of sensor data and want to distribute it among three different processing pipelines: temperature monitoring, humidity monitoring, and air quality monitoring. You can use the `distributeWith` operator to achieve this:

```

1 trait ZStream[-R, +E, +A] {
2   def distributedWith[E1 >: E] (
3     n: => Int,
4     maximumLag: => Int,
5     decide: A => UIO[Int => Boolean]
6   ): ZIO[R with Scope, Nothing, List[Dequeue[Exit[Option[E1], A]]]] = ???
7 }

```

The fundamental part of the `distributeWith` operator is the `decide` function, which takes an element and returns a ZIO effect that determines which queues should receive the element. The `Int` in the inner function represents the index of the queue, and the `Boolean` indicates whether the element should be sent to that queue. The `distributeWith` operator returns a list of queues, where their index corresponds to the index identified by the `decide` function.

Let's say we have a stream of logs, and we want to distribute them among three different processing pipelines based on the log level, he

```

1 val distributionLogic = (entry: LogEntry) =>
2   ZIO.succeed { (queueId: Int) =>
3     queueId match {
4       case 0 => entry.level == "ERROR" | entry.level == "FATAL"
5       case 1 => entry.level == "WARN" | entry.level == "INFO"
6       case 2 => entry.level == "DEBUG" | entry.level == "TRACE"
7     }
8   }

```

In this example, we have three queues. The first queue receives all logs with the “ERROR” or “FATAL” level, the second queue receives logs with the “WARN” or “INFO” level, and the third queue receives logs with the “DEBUG” or “TRACE” level.

```

1 for {
2   queues <- logStream.distributedWith(
3     n = 16,
4     maximumLag = 32,
5     decide = distributionLogic
6   )
7
8   - <- ZIO.foreachParDiscard {
9     queues.map(ZStream.fromQueue(_).flattenExitOption).
10      zipWithIndex
11    } { case (stream, streamId) =>
12      streamId match {
13        case 0 => stream.foreach(processFatalAndErrors)
14        case 1 => stream.foreach(processWarnsAndInfos)
15        case 3 => stream.foreach(processDebugAndTraces)
16      }
17    }
18 } yield ()

```

You may want a more advanced distribution method where the number of consumers is not known upfront, or you may want to introduce a callback function that gets called when the distribution is done. In such cases, you can use the `distributeWithDynamic` operator.

## 30.9 Flow Control and Rate Limiting

Controlling data flow is crucial for system stability and resource management in data streaming applications. ZIO Streams provides several operators to manage data flow and processing rates.

Various techniques and transformation operations control the flow of data. In this section, we will focus on buffering, debouncing, and throttling.

### 30.9.1 Buffering

In the ZIO Stream, when the consumer is slower than the producer, the consumer slows down the producer by applying back pressure. However, in some cases, you should buffer elements to prevent backpressure from propagating upstream. Buffering allows accumulating elements in memory until the consumer is ready to process them. This can help smooth out bursts of data and optimize performance in high-throughput scenarios.

When you need to buffer elements, you can use the `buffer` operator, which allows you to specify the maximum number of elements to buffer:

```
1 trait ZStream[-R, +E, +A] {
2   def buffer(capacity: Int): ZStream[R, E, A] = ???  
3 }
```

For example, if you have a stream of sensor data and want to buffer up to 10000 elements before processing them, you can use the `buffer` operator:

```
1 case class SensorData()  
2 def processLog(log: SensorData): ZIO[Any, Nothing, Unit] = ???  
  
1 sensorData  
2   .buffer(10000) // Buffer up to 10000 sensor data elements  
3   .mapZIOParUnordered(10)(processLog) // Process up to 10 element  
  concurrently
```

The `buffer` operator maintains a queue of elements of the given capacity. When the queue becomes full, it applies backpressure to the upstream, preventing it from producing more elements until the queue has space to accommodate new elements.

In some scenarios, you may want to use other buffering strategies:

- **Fixed-size Queue** (e.g., `buffer`): This is the default buffering strategy, where the buffer has a fixed capacity. When it is full, backpressure is applied upstream.
- **Unbounded Queue** (e.g., `bufferUnbounded`): This one does not apply backpressure when the buffer is full. Instead, it allows the buffer to grow indefinitely. Be cautious when using this strategy, as it can lead to excessive memory consumption if the consumer is slower than the producer.
- **Dropping Queue** (e.g., `bufferDropping`): When the buffer is full, this strategy discards the new elements.
- **Sliding Queue** (e.g., `bufferSliding`): This strategy removes the oldest elements from the buffer when the buffer is full, allowing new elements to be added.

Now, let's discuss the implications of buffering on the chunking structure of the stream. Buffering operators destroy the chunking structure of the stream, so to not lose the performance benefits of a well-chunked stream, it is recommended to re-chunk the stream after buffering:

```
1 sensorData  
2   .buffer(10000) // Buffer up to 10000 sensor data elements
```

```

3   .rechunk(4096) // Re-chunk the stream to improve performance
4   .mapZIOParUnordered(10)(processLog) // Process up to 10 element
      concurrently

```

In this example, we re-chunked the stream back to a chunk size of 4096 elements, the default chunk size in ZIO Streams.

In ZIO Streams, we have two types of buffering based on the granularity of buffering:

- **Chunk-wise Buffers** (e.g., `bufferChunks`, `bufferChunksDropping`, `bufferChunksSliding`, `bufferUnbounded`): These operations buffer underlying chunks of data instead of individual elements, hence preserving the stream's chunking structure.
- **Element-wise Buffers** (e.g., `buffer`, `bufferDropping`, `bufferSliding`, `bufferUnbounded`): These operations buffer elements individually, and hence they can't preserve the chunking structure of the stream.

### 30.9.2 Debouncing

Debouncing is a technique for controlling the rate at which a stream emits elements. The `debounce` operator discards incoming data until a certain time has passed. It's particularly useful for handling high-frequency events or inputs, ensuring that only the most recent or relevant data is processed after a specified delay.

With `debounce`, you can emit only the last element received within a specific time window. Assume you have a stream of user inputs and want to perform a search operation for them. It is not wise to perform a search operation for each keystroke, which can overwhelm the server. Instead, you can debounce the stream:

```

1 import zio._
2 import zio.stream._

3
4 val userInputs: ZStream[Any, Nothing, String] =
5   ZStream(
6     "h", "he", "hel", "hell", "hello", "hello ", "hello w", "
      hello wo",
7     "hello wor", "hello worl", "hello world"
8   ).schedule(Schedule.spaced(100.milliseconds))

9
10 def query(query: String): ZIO[Any, Nothing, String] = ???

11
12 val debouncedSearch =
13   userInputs
14     .debounce(300.milliseconds)
15     .mapZIO(query)

```

This prevents exceeding API limits and reduces server load by ensuring the search operation is triggered only after a certain period.

In general, debouncing is helpful in many event-handling scenarios, such as managing UI events, network requests, sensor data, GPS location requests, and more.

### 30.9.3 Throttling

Throttling is similar to debouncing, but instead of emitting only the last element, it limits the rate at which elements are emitted within the given time window:

```

1 trait ZStream[-R, +E, +A] {
2   def throttleEnforce[In](
3     units: Long,
4     duration: => Duration,
5     burst: => Long = 0
6   )
7 }
```

The `throttleEnforce` uses a token bucket algorithm. Assume you have a bucket that can hold a certain number of tokens; each token represents the permission to emit elements. Within a given time window, if an element wants to be emitted, it needs to consume some tokens (the exact amount will be calculated using `costFn`). For every incoming element, some tokens are consumed until we run out of tokens. From that point, every new incoming element will be dropped. The bucket is refilled with new tokens at a constant rate in the next time window.

By adjusting the `units` and `duration` parameters, you can control the number of tokens added per time interval and the duration of the time window. This helps to manage the rate of data flow and prevent overloading downstream systems. The `costFn` parameter allows you to define a cost function that calculates the cost of emitting each chunk in terms of the number of tokens.

Another interesting parameter is the `burst`, which allows you to define extra capacity for the bucket, temporarily increasing the number of permitted elements to be emitted within a time window. The maximum number of tokens that can be consumed in a single time window is the maximum size of the bucket. If many tokens are in the bucket (e.g., after a period of low traffic), the `throttleEnforce` allows a burst of traffic to be emitted; this can be at maximum bucket tokens (`units + burst`). After the burst, tokens are depleted rapidly. If further traffic arrives, it will be dropped until the bucket is refilled. The `units` tokens are refilled constantly, but the `burst` tokens are refilled based on the elapsed time and time window duration.

```

1 val throttledStream =
2   userInputs
3     .throttleEnforce(50, 300.milliseconds, burst = 30) {
4       (chunk: Chunk[Int]) => chunk.size.toLong
5     }
6     .mapZIO(query)
```

Another variant of throttling, `throttleShape`, allows you to throttle the data stream

without dropping any elements. Instead, it slows down the rate of data flow by delaying the emission of elements. This can be useful when you want to ensure that the downstream system can handle all incoming data without being overwhelmed.

## 30.10 Conclusion

ZIO Streams provides a robust toolkit for stream processing, offering a wide range of transformation operations. These tools enable developers to build sophisticated, efficient, and scalable data processing pipelines.

In this chapter, we explored various transformation operations such as mapping, filtering, grouping, partitioning, broadcasting, distributing, flow control, and rate limiting. We also discussed the importance of considering factors like stateful vs. stateless transformations, ordering guarantees, chunking structures, and buffering strategies when working with these operations.

By mastering these transformation operations, you can build complex stream processing applications that handle high-throughput data streams, manage data flow, and optimize resource utilization.

## 30.11 Exercises

1. Create an infinite stream of the Fibonacci sequence using `ZStream.unfold`.
2. Create a stream transformation that computes the running average of all integer elements seen so far using `ZStream.mapAccum`.
3. Create a stream transformation that computes the moving average of the last N elements using `ZStream.scan`.
4. Implement a stream transformation that deduplicates elements within a sliding time window while preserving order.
5. Create a stream that paginates through GitHub's REST API to fetch all repositories from the ZIO organization. Hint: Use the `ZStream.paginateZIO` operator to fetch all pages by passing the "page" path parameter to the "<https://api.github.com/orgs/zio/repos?page=>" endpoint.
6. Assume you have given a stream of `UserEvent`; write a stream transformation that counts the occurrence of each event type received until now:

```
1 sealed trait UserEvent
2 case object Click    extends UserEvent
3 case object View     extends UserEvent
4 case object Purchase extends UserEvent
```

7. Create a simple program that broadcasts a stream of integers to three consumers:
  - One consumer that prints only even numbers

- One consumer that prints only odd numbers
- One consumer that prints all numbers multiplied by 10

# Chapter 31

## Streaming: Combining Streams

After learning how to transform streams, it's time to learn how to combine them. In this chapter, we will focus on the various ways to combine streams in ZIO, allowing you to build complex data processing pipelines with ease.

Combining streams enables you to perform a wide range of operations when you have more than one stream of data and need to process them together, such as:

- Merging, joining, and aggregating data from multiple sources
- Coordinating and synchronizing parallel data flows
- Handling errors and implementing primary and backup stream failover strategies
- Detecting correlations and patterns between different data streams, such as anomaly detection and trend analysis

The way you combine streams will determine the operators you need to use. For instance, do you want to combine two streams sequentially, concurrently, and pairwise? Is the combination stateful or stateless? How should the system switch to another stream if the current one fails?

ZIO Stream offers a comprehensive set of operators to address all these cases. By mastering these combination techniques, you'll be prepared to tackle various stream processing challenges in your ZIO applications.

Let's dive into each of these methods and explore how they can be used to solve real-world problems with ZIO Stream.

### 31.1 Stream Concatenation

Stream concatenation, meaning joining two streams end-to-end. It is like appending the elements of the second stream to the first stream.

You can use the `concat` or its alias `++` operator to concatenate two streams. So it will first emit all elements of the first stream and then all elements of the second one:

```

1 val stream1 = ZStream(1, 2, 3)
2 val stream2 = ZStream(4, 5, 6)
3 val concatenated = stream1 ++ stream2

```

Unlike the `merge` operator, which combines streams in a concurrent or interleaved fashion, when you use `++`, you are essentially saying, “run this stream to completion and then run the next one”, which maintains a strict order of processing.

If you have a chunk of streams, you can use `ZStream.concatAll` to concatenate them all:

```

1 val concatenated =
2   ZStream.concatAll(
3     Chunk(ZStream(1, 2), ZStream(3, 4), ZStream(5, 6))
4   )

```

## 31.2 Merging Streams

Unlike `concat`, the `merge` operator processes both input streams in parallel, resulting in a non-deterministic order of elements in the output stream. It processes elements from both input streams as they become available, and the order of elements in the resulting stream is not guaranteed:

```

1 val stream1 = ZStream(1, 2, 3)
2 val stream2 = ZStream(4, 5, 6)
3 val merged = stream1.merge(stream2)

```

By default, the `merge` operator will terminate the resulting stream as soon as both input streams terminate. If you want to change this behavior, you can provide a termination strategy or use one of the specialized merge operators like `mergeHaltLeft`, `mergeHaltRight`, or `mergeHaltEither`.

If you want to have a sorted output, you can use the `mergeSorted` operator. It takes an implicit `Ordering` instance for the elements of the stream and will emit elements in sorted order.

Another interesting operator, `mergeWith`, allows you to merge two streams of different types by providing a function that unifies the two types into a common type.

## 31.3 Zipping Streams

ZIO Stream provides a rich set of zip operators to zip multiple streams in various ways. They combine elements from two or more streams into a single stream, emitting a tuple of elements from the input streams.

Let's start with the simplest one, `zip`:

```

1 val a = ZStream(1, 2, 3)
2 val b = ZStream("a", "b", "c")
3 val zipped = a.zip(b)

```

The `zip` operator, or its alias `<&>`, will terminate the resulting stream once one of the input streams terminates. For example, if the first stream has more elements than the second one, the resulting stream will terminate as soon as the second stream terminates:

```

1 val a = ZStream(1, 2, 3, 4)
2 val b = ZStream("a", "b")
3 val zipped = a <&> b
4 // Result: ZStream((1, "a"), (2, "b"))

```

If you only want to zip the elements but need the elements from the first stream, you can use `zipLeft` or `<&`; otherwise, you can use `zipRight` or `&>` to get the elements from the second stream.

```

1 val zippedLeft = a <& b // ZStream(1, 2) // ZStream(1, 2)
2 val zippedRight = a &> b // ZStream("a", "b")

```

If you need to map the pairs to a different value/type, you can use the `zipWith` operator, which takes a function from the tuple of elements to the desired value:

```

1 val zipped = a.zipWith(b)((num, str) => s"$num-$str")

```

The `zip` operator and its variants, such as `zipLeft`, `zipRight`, and `zipWith`, have early termination semantics. If one of the input streams terminates, the resulting stream will terminate. If you want to keep the resulting stream alive until all input streams terminate, you can use the `zipAll` operator.

The `zipAll` operator takes two default values for the case when one of the input streams terminates before the other, so it will keep emitting elements using the default value until the other stream terminates:

```

1 val zippedAll = a.zipAll(b)(0, "Z")

```

It also has left and right variants, `zipAllLeft` and `zipAllRight`, which take a default value for the left or right stream, respectively. To provide a custom function to combine the elements, you can use the `zipAllWith` operator.

The `zipLatest` (and also its `zipLatestWith` variant) is similar to `zip`, but if one of the input streams emits an element, it won't wait for the other stream to emit; it uses the latest element from the other stream. This operator is particularly useful in scenarios where you have two streams of data that update at different rates, and you always want to combine the most recent data from both streams. For example, assume you have two streams, one coming from a temperature sensor every second and the other from a humidity sensor every 5 seconds. You can use `zipLatest` to combine the latest temperature and humidity values:

```

1 val weatherData =
2   temperatureStream
3     .zipLatestWith(humidityStream) {
4       case (temp, humidity) => WeatherData(temp, humidity)
5     }

```

The `zipWithIndex` operator zips the elements of the stream with their index. It's useful when you need to keep track of the index of the elements in the stream:

```

1 val fruitStream: ZStream[Any, Nothing, String] =
2   ZStream("apple", "date", "banana", "cherry")
3 fruitStream.zipWithIndex.foreach {
4   case (fruit, index) => ZIO.debug(s"$index: $fruit")
5 }

```

There are also operators to zip elements with their previous, next, or both previous and next elements:

```

1 trait ZStream[-R, +E, +A] {
2   def zipWithPrevious: ZStream[R, E, (Option[A], A)]
3   def zipWithNext    : ZStream[R, E, (A, Option[A])]
4   def zipWithPreviousAndNext: ZStream[R, E, (Option[A], A, Option[A])]
5 }

```

These operators are useful when you need to compare elements with their neighbors:

```

1 case class StockPrice(date: String, price: Double)
2
3 val stockPrices = ZStream(
4   StockPrice("2024-01-01", 100.0),
5   StockPrice("2024-01-02", 101.5),
6   StockPrice("2024-01-03", 103.0),
7   StockPrice("2024-01-04", 102.5),
8   StockPrice("2024-01-05", 104.0)
9 )
10
11 stockPrices
12   .zipWithPrevious
13   .foreach { case (prev, curr) =>
14     val trend = prev match {
15       case Some(p) if curr.price > p.price => " Up"
16       case Some(p) if curr.price < p.price => " Down"
17       case Some(_) => " No change"
18       case None => "Initial price"
19     }
20     val change = prev.map(p => f"${curr.price - p.price}%.2f").
21    getOrElse("N/A")

```

```

21   Console.printLine(s"${curr.date}: $$$${curr.price} | Trend:
22     $trend | Change: $$$${change}")
}

```

Here is the output of the above code:

```

1 2023-01-01: $100.0 | Trend: Initial price | Change: $N/A
2 2023-01-02: $101.5 | Trend: Up | Change: $1.50
3 2023-01-03: $103.0 | Trend: Up | Change: $1.50
4 2023-01-04: $102.5 | Trend: Down | Change: $-0.50
5 2023-01-05: $104.0 | Trend: Up | Change: $1.50

```

ZIO Stream provides a set of powerful operators for combining sorted streams based on a common key. These operators are handy when working with related data streams with a common identifier. For example, consider a scenario where you have two streams of data; one stream contains user data, and the other contains user orders. You can combine these two streams based on the user ID to get a comprehensive view of users and their orders:

```

1 val users = ZStream(
2   (1, "Alice"),
3   (2, "Bob"),
4   (4, "David")
5 )
6
7 val orders = ZStream(
8   (1, "Book"),
9   (2, "Pen"),
10  (3, "Notebook"),
11  (4, "Laptop")
12 )
13
14 users.zipAllSortedByKey(orders)("Unknown User", "Unknown Order")
15   .map(_._2)
16   .foreach(ZIO.debug(_))

```

There are also left and right variants of this operator, `zipAllSortedByKeyLeft` and `zipAllSortedByKeyRight`, which allow you to keep only the elements from the left or right stream, respectively.

## 31.4 Cartesian Product

The cartesian product of two streams is a stream of all possible ordered pairs of elements from the two streams. The `cross` operator, or its alias `<*>`, allows you to compute the cartesian product of two streams:

```

1 val a = ZStream(1, 2)
2 val b = ZStream("a", "b", "c")

```

```
3| val crossed = a <*> b
```

Like the `zip` operation, it has two left and right variants, `crossLeft` or `<*` and `crossRight` or `*>`, which only keep the first or second element of the tuple, respectively.

## 31.5 Fallback Streams

Sometimes, you may want to fall back to another stream if the current stream fails. The `orElse` operator allows you to do that. It takes an alternative stream and emits elements from the first stream, but if the first stream fails, it will switch to the alternative stream:

```
1| val a = ZStream(1, 2, 3) ++ ZStream.fail("Oops!")
2| val b = ZStream(4, 5, 6)
3| val result = a orElse b
4| // Result: ZStream(1, 2, 3, 4, 5, 6)
```

It is like catching all errors from the first stream, discarding them, and switching to the alternative stream:

```
| val result = a.catchAll(_ => b)
```

With `catchAll`, we have more control over the error handling; we can decide to switch to which alternative stream based on the error:

```
1| val c = ZStream(7, 8, 9)
2|
3| val result = a.catchAll {
4|   case "Oops!" => b
5|   case "Ouch!" => c
6|   case _           => ZStream.empty
7| }
```

Like the `ZIO` data type, `ZIO Stream` has various operators to recover from errors and defects, such as `catchAll`, `catchSome`, `catchAllCause`, and `catchSomeCause`.

In some scenarios, a stream may stall without producing errors, failing to emit elements for an extended period. To address this, you can use the `timeoutTo` operation. This method allows you to specify a duration and an alternative stream. If the primary stream remains idle beyond the specified timeout, it automatically switches to the alternative stream:

```
1| val stream =
2|   ZStream(1, 2, 3) ++
3|     ZStream.fromZIO(ZIO.sleep(5.seconds)) ++
4|     ZStream(4, 5, 6)
5| val fallback = ZStream(10, 20, 30)
6| val resultStream = stream.timeoutTo(3.seconds)(fallback)
```

## 31.6 Stateful Stream Combination

Until now, we have seen how to combine streams with specialized operators; each one has its own semantics. Sometimes, you may need to combine streams with a more complex logic, with fine-grained control over how the elements are combined. In such cases, you can use the `combine` operator:

```

1 trait ZStream[-R, +E, +A] {
2   def combine[R1 <: R, E1 >: E, S, A2, A3](
3     that: => ZStream[R1, E1, A2]
4     )(s: => S)(
5       f: (S, ZIO[R, Option[E], A], ZIO[R1, Option[E1], A2]) =>
6         ZIO[R1, Nothing, Exit[Option[E1], (A3, S)]]
7     ): ZStream[R1, E1, A3]
8 }
```

The `combine` operator allows you to combine two streams statefully. It takes the `f`, a combiner function, with the following signature:

```

1 def combiner[S, A, A2, A3](
2   state: S,
3   pullLeft : ZIO[R, Option[E], A],
4   pullRight: ZIO[R1, Option[E1], A2]
5   ): ZIO[R1, Nothing, Exit[Option[E1], (A3, S)]]
```

It takes a state `S` and two effects. By calling each, you can pull elements from the left or right stream. The combiner function returns a tuple of the combined element with the updated state. The combiner function can also return an `Exit` value, which allows you to signal the termination of the resulting stream.

Let's take a closer look at the error channel of the pull functions. They have optional error types, `Option[E]` and `Option[E]`, meaning that the error can be either `None` or `Some` of an arbitrary error type. The `None` error type indicates that the corresponding stream (left or right stream) has terminated successfully, so there are no more elements to pull. The `Some` error type indicates that an error has occurred during the pulling and processing of the elements. Based on each error type, you can decide how to handle the error and whether to continue or terminate the resulting stream.

Now let's see the return type of the `combiner` function; its error channel is `Nothing`, meaning that the combiner function cannot fail; instead, we should encode both success and failure cases in the success channel using the `Exit` type. The success channel has an `Exit` type, which can be either `Exit.Success` or `Exit.Failure`:

- If you want to continue pulling elements from the input streams, you should return `Exit.Success` with the tuple of combined elements and the updated state.
- To terminate the resulting stream, you should return `Exit.Failure`. If the error is `None`, the combiner function should terminate the resulting stream successfully. If the error is `Some`, the combiner function should terminate the resulting stream with the specified error.

As the pull functions are effectful, you can use the full power of ZIO to handle errors, retry, or recover from failures. For example, you can `race` the two pull effects to pull elements concurrently or run each pull effect as many times as needed to get the desired element.

Let's try a simple example just for demonstration purposes. It's only going to zip two streams:

```

1 val numbers =
2   ZStream.fromIterable(1 to 10)
3     .schedule(Schedule.fixed(500.millis))
4
5 val characters =
6   ZStream.fromIterable(1 to 10)
7     .map(e => ('a' + e).toChar)
8     .schedule(Schedule.fixed(250.millis))
9
10 // Combine function
11 def combiner(
12   state: Int,
13   pullNumbers: ZIO[Any, Option[Nothing], Int],
14   pullCharacters: ZIO[Any, Option[Nothing], Char]
15 ): URIO[Any, Exit[Option[Nothing], ((Int, Char), Int)]] = 
16   pullNumbers
17     .zipWithPar(pullCharacters)((_, _))
18     .map((_, state))
19     .option
20     .map {
21       case Some(a) => Exit.succeed(a)
22       case None     => Exit.fail(None)
23     }
24
25 // Combine the two streams
26 val combinedStream = numbers.combine(characters)(0)(combiner)
```

The `combiner` function pulls elements from the two input streams concurrently, zips them, and returns the tuple of the zipped elements. The `combine` operator will keep calling the `combiner` function until one of the input streams terminates.

In this example, we didn't utilize the state parameter, but in more advanced use cases, you can use it to keep track of the combiner function's progress or to store intermediate results.

Let's explore a practical example. Suppose you have two streams: one for stock prices and one for trading signals:

```

1 case class StockPrice(timestamp: Long, price: Double)
2
3 sealed trait TradingSignal
4 case object Buy extends TradingSignal
5 case object Sell extends TradingSignal
```

```

6 case object Hold extends TradingSignal
7
8 val priceStream : ZStream[Any, Nothing, StockPrice] = ????
9 val signalStream: ZStream[Any, Nothing, TradingSignal] = ???

```

Let's define trade actions, which are the output of the trading algorithm:

```

1 sealed trait TradeAction
2 case class ExecuteBuy(price: Double) extends TradeAction
3 case class ExecuteSell(price: Double, profit: Double) extends TradeAction
4 case object NoAction extends TradeAction

```

And also to keep track of last buy price and total profit:

```

1 case class TradingState(
2   lastBuyPrice: Option[Double],
3   totalProfit: Double
4 )

```

Now, we can write a trading algorithm that combines these two streams to generate trading actions based on stock prices, trading signals, and the last buy price:

```

1 def decisionLogic(
2   state: TradingState,
3   priceIO: ZIO[Any, Option[Nothing], StockPrice],
4   signalIO: ZIO[Any, Option[Nothing], TradingSignal]
5 ): ZIO[Any, Nothing, Exit[Option[Nothing], (TradeAction,
6   TradingState)]] = {
7   def processTradeSignal(
8     price: Double,
9     signal: TradingSignal,
10    state: TradingState
11  ) = (signal, state.lastBuyPrice) match {
12    case (Buy, None) =>
13      (ExecuteBuy(price), state.copy(lastBuyPrice = Some(price))
14    ))
15    case (Sell, Some(buyPrice)) if price > buyPrice =>
16      val profit = price - buyPrice
17      (ExecuteSell(price, profit), TradingState(None, state.
18        totalProfit + profit))
19    case _ =>
20      (NoAction, state)
21  }
22
23  for {
24    signal <- signalIO
25    price <- priceIO
26  } processTradeSignal(price, signal)
27
28  state
29}

```

```

21   priceOpt <- priceIO.option
22   signalOpt <- signalIO.option
23   result <- (priceOpt, signalOpt) match {
24     case (Some(StockPrice(_, price)), Some(signal)) =>
25       val (action, newState) =
26         processTradeSignal(price, signal, state)
27       ZIO.succeed(Exit.succeed((action, newState)))
28     case _ =>
29       ZIO.succeed(Exit.fail(None))
30   }
31 } yield result
32 }
```

In this example, the `decisionLogic` function serves as the combiner, taking the current trading state, stock price, and trading signal to generate a trading action. The `TradingState` case class maintains the last buy price and total profit, and it gets updated based on the actions taken. This updated state is then used in the next iteration of `decisionLogic` to determine the next trading action.

## 31.7 Interleaving

The `interleave` operator allows you to interleave elements from two streams deterministically:

```

1 val a = ZStream(1, 2, 3)
2 val b = ZStream('a', 'b', 'c')
3 val interleaved = a.interleave(b)
4 // Result: ZStream(1, 'a', 2, 'b', 3, 'c')
```

If you want to have more control over the interleaving process, you can use the `interleaveWith` operator. It takes a stream of booleans to decide which stream to pull from:

```

1 val selector = ZStream(true, false, true, false, true, false)
2 val interleaved = a.interleaveWith(b)(selector)
```

Please note that the `interleave` element doesn't respect the chunking structure of the input streams, meaning that the resulting stream will have a chunk size of 1, even if the input streams have larger chunks. So, you may need to `rechunk` the resulting stream to ensure efficient processing.

## 31.8 Conclusion

In this chapter, you've expanded your toolkit for building sophisticated, reactive systems by mastering stream combination techniques. You are now able to handle complex scenarios such as:

- Combining data from multiple APIs or databases
- Implementing advanced error handling and recovery strategies
- Creating custom stream processing algorithms
- Building responsive user interfaces that react to multiple data sources

As you continue to work with ZIO Stream, you'll likely discover even more creative ways to leverage these combination operators. With practice, you'll become proficient at orchestrating complex data flows, enabling you to build robust and reactive applications.

## 31.9 Exercises

1. Implement a stream transformation that correlates events across multiple streams using flexible matching criteria:

```

1 case class Event[A] (
2   id: String,
3   timestamp: Long,
4   data: A
5 )
6
7 def correlateEvents[R, E, A, B, C](
8   stream1: ZStream[R, E, Event[A]],
9   stream2: ZStream[R, E, Event[B]],
10  correlationWindow: Duration,
11  matcher: (Event[A], Event[B]) => Boolean,
12  combiner: (Event[A], Event[B]) => C
13 ): ZStream[R, E, C] = ???
```

2. Combines two streams where the priority stream takes precedence. When elements are available from both streams, elements from the priority stream should be processed first:

```

1 def priorityMerge[A] (
2   priority: ZStream[Any, Nothing, A],
3   regular: ZStream[Any, Nothing, A]
4 ): ZStream[Any, Nothing, A] = ???
```

3. Implement a stream combinator that dynamically adjusts sampling rates based on a control stream, useful for monitoring systems that need to adapt to the system load:

```

1 case class SamplingConfig(samplesPerMinutes: Int)
2
3 def adaptiveSampling[R, E, A](
4   dataStream: ZStream[R, E, A],
5   controlStream: ZStream[Any, Nothing, SamplingConfig]
6 ): ZStream[R, E, A] = ???
```

# Chapter 32

## Streaming: Pipelines

As discussed in previous chapters, pipelines represent the middle part of data processing flows, sitting between the sources that produce data and the sinks that consume it.

Think of pipelines as assembly lines in a factory—they take raw materials (input data), process them through various transformation steps, and produce refined outputs that can be used downstream but in composable, reusable ways.

In this chapter, we'll explore:

- What pipelines are and how they fit into ZIO Stream's data processing model
- How to use existing pipelines to transform streams
- When to use pipelines versus stream transformation operators
- How to construct custom pipelines

Let's dive in and see how pipelines can help us build powerful streaming data transformations.

### 32.1 Pipelines as Stream Transformations

Pipelines are the final core data type in ZIO Stream. While streams represent the beginning of a data flow process and sinks represent the end of a data flow process, pipelines represent the middle of it.

For example, a relatively simple linear data flow process might look like this:

```
stream >>> pipeline1 >>> pipeline2 >>> pipeline3 >>> sink
```

Each of these pipelines represents some transformation step. This could be transforming the values from one type to another, aggregating them, filtering them, or anything else.

Conceptually, a relatively simple mental model to use to think about pipelines is just as a stream transformer:

```

1 trait ZPipeline[-Env, +Err, -In, +Out] {
2   def apply[Env1 <: Env, Err1 >: Err](stream: ZStream[Env1, Err1,
3                                         In] => ZStream[Env1, Err1, Out]

```

We can see from this signature that the definition of a pipeline is extremely broad.

It takes as input a stream with one element type and returns a new stream with a different element type. From this signature, it could transform the stream elements one for one, filter out stream elements, aggregate stream elements, append new stream elements, or anything else.

Just about the only thing we can say a pipeline cannot do from this signature is provide the stream with its required environment or handle stream errors.

We can see this because the new stream always requires at least the services of the original stream and can always fail with the errors of the original stream. In other words, just like a sink, a pipeline is a strategy for describing transforming values, not handling errors.

While this definition of a pipeline is a helpful mental model for you to get your head around pipelines and what they can do, it is not actually the way pipelines are implemented in ZIO Stream. The reason is that if a pipeline was just a function, we would have a very limited ability to introspect on it or compose it with other streaming data types.

For example, if we had a sink that aggregated A values and a pipeline that transformed B values into A values, we would like to be able to “hook them up” to get a new sink that aggregated B values. This new sink would take the B values, transform them into A values with the logic of the pipeline, and then aggregate those A values with the logic of the sink.

However, there is no way we could implement this if a pipeline was just a function from a stream to a stream and a sink was a completely different data type.

This is why pipelines, like all the other core data types in ZIO Stream, are represented as channels. As discussed in the chapter on channels, the actual implementation of a pipeline is just:

```

1 final case class ZPipeline[-Env, +Err, -In, +Out](
2   channel: ZChannel[Env, ZNothing, Chunk[In], Any, Err, Chunk[Out],
3                           Any]

```

Let’s unpack this definition a little more, just like we have for the other streaming data types.

A pipeline accepts two types of inputs. It can receive zero or more elements of type `Chunk[In]` and eventually a done value of type `Any` that does not contain any useful information beyond the fact that there will be no more input.

The pipeline, like the sink, has `ZNothing` for its input error type. Since there are no values of type `ZNothing`, this indicates that the pipeline can never receive errors, which is consistent with our discussion above: pipelines are strategies for transforming stream elements

and not strategies for handling stream errors.

The pipeline can emit zero or more elements of type `Chunk[Out]` and eventually be done, if it ever completes, with either a done value that does not contain meaningful information or an error of type `Err`.

## 32.2 Using Pipelines

Pipelines are quite simple in a way in that as transformers of elements, we essentially always use them one way by “hooking them up” to other streams, pipelines, or sinks.

Let’s first look at how we can hook a stream up with a pipeline:

```

1 import zio._
2 import zio.stream._

3
4 val stream: ZStream[Any, Nothing, Int] =
5   ZStream(1, 2, 3, 4, 5)

6
7 val doubler: ZPipeline[Any, Nothing, Int, Int] =
8   ZPipeline.map[Int, Int](_ * 2)

9
10 val doubled: ZStream[Any, Nothing, Int] =
11   stream_via(doubler)

```

To send a stream’s outputs through a pipeline, we use the `ZStream#via` operator or its symbolic alias `>>>`. This will take each element of the stream, send it through the pipeline, and emit the elements of the pipeline.

Like all data types in ZIO Stream, pipelines are pull-based, and so while it can be natural to talk about “sending” the elements of the stream to the pipeline, the actual process is driven by demand from the resulting stream.

When a value is read from the `doubled` channel, the pipeline will be read from. The pipeline may already have a value to write or may read from the original stream one or more times until it does have a value to write or terminates, and that process will be repeated over and over.

This is just the `pipeTo` operator that we saw on channels, and in particular the `pipeToOrFail` variant that fails with the error of the original stream if the stream fails rather than sending it to the pipeline since as we discussed previously, the pipeline cannot handle failures.

In addition to transforming the outputs of a stream with a pipeline, we can also transform the inputs of a sink with a pipeline using the `>>>` operator. For example, if we have a sink that aggregates numeric values, we could combine it with a pipeline that parses strings into numeric values to get a sink that aggregates strings:

```

1 val sink: ZSink[Any, Nothing, Int, Int, Int] =

```

```

2   ZSink.sum[Int]
3
4 val pipeline: ZPipeline[Any, NumberFormatException, String, Int] =
5   ZPipeline.mapZIO {
6     string => ZIO.attempt(string.toInt).refineToOrDie[
7       NumberFormatException]
8   }
9 pipeline >>> sink

```

Combining a pipeline with a sink results in a new sink, just as combining a pipeline with a stream results in a new stream.

The final way we can use pipelines is by combining them with each other. Going back to the analogy of pipelines as stream transformation functions, we can think of being able to compose two pipelines where the output type of one pipeline lines up with the input type of the other.

For example, if we have a pipeline that parses strings into integers and another pipeline that converts integers into floating point values, we can combine them to create a new pipeline that parses strings into floating point values:

```

1 val stringToInt: ZPipeline[Any, NumberFormatException, String,
2   Int] =
3   ZPipeline.mapZIO {
4     string => ZIO.attempt(string.toInt).refineToOrDie[
5       NumberFormatException]
6   }
7
8 val inttoDouble: ZPipeline[Any, Nothing, Int, Double] =
9   ZPipeline.map(_.toDouble)
10
11 val stringtoDouble: ZPipeline[Any, NumberFormatException, String,
12   Double] =
13   stringToInt >>> inttoDouble

```

The types will normally guide you in making sure that you compose pipelines in ways that make sense.

However, if you have multiple pipelines that have the same input and output types, you may need to think more about which ways of composing them reflect your intended logic. For example, mapping and then filtering elements may not be the same as filtering and mapping elements.

One of the most common questions that arises when working with pipelines is when to use pipelines versus when to use stream operators.

As the discussion above indicates, in principle almost any stream operator can be described

as a pipeline. The only exceptions would be operators that handle errors or operators that provide a stream with its required environment, which are definitely a relatively small number of the many stream operators.

So then there is this question of whether to transform streams by calling operators defined on the `ZStream` data type or by using `>>>` and constructors defined on `ZPipeline`. For example, the following are equivalent:

```
1 val stream1: ZStream[Any, Nothing, Int] =
2   ZStream(1, 2, 3).map(_ * 2)
3
4 val stream2: ZStream[Any, Nothing, Int] =
5   ZStream(1, 2, 3) >>> ZPipeline.map[Int, Int](_ * 2)
```

The first answer is do what works for you! Teams have different styles, and the extent to which to use pipelines can be a matter of personal preference.

In particular, we have found that users coming from Akka typically tend to use pipelines more frequently since this style of separate data types is more common in that framework with their concepts of sources, flows, and sinks that correspond to streams, pipelines, and sinks in ZIO Stream. In contrast, users coming from other functional programming based streaming libraries such as FS2 tend to use stream operators more frequently since those libraries do not support all of the parts of a data processing pipeline as first-class values.

While this flexibility to use either pipelines or stream operators can be liberating, it can also create differences in styles within a team, and some people prefer more prescriptive guidance, so here are some things to think about:

Stream operators tend to have better type inference and better tooling support via auto-completion, so as a baseline, we would tend to recommend using stream operators unless you have a specific reason to use pipelines as discussed below.

Defining pipelines will often require specifying type parameters if the pipeline is generic since otherwise, the Scala compiler cannot know what the pipeline's input type is.

For example, above, the `Int` type of the `map` pipeline had to be specified. Of course, this is not a problem if the pipeline is specialized to a particular type, such as a pipeline that decodes characters using a particular format.

Using pipelines will also often have less support in code editors because dot completion does not work as well. In writing `ZStream(1, 2, 3)`, the editor will immediately suggest relevant operators on streams based on the type of the stream, whereas when using pipelines, the full set of pipeline constructors will typically be suggested because the input type of the pipeline is not known at that point.

This can definitely be somewhat subjective, but our experience has been that developers are generally used to calling concrete operators on concrete data types, and sticking with this in the absence of a compelling reason to the contrary can be a good strategy.

Favor pipelines when you want to treat the transformation as a value and use it in multiple places. Encoding and decoding are great examples of this.

Transforming data between one format or another can be tricky, with all sorts of details of the formats that must be handled correctly. Once you've done this, you want to never do it again and use the same transformation in every place it is required.

This logic should also be completely independent of any particular stream. How to encode or decode data for a particular format should be completely independent of the logic of a particular stream as long as it is of the appropriate format.

So encoders and decoders are a great use case for pipelines, and in fact, if you look at ZIO itself, you will see that the `ZPipeline` companion object contains a variety of encoders and decoders for common data formats. Many other ZIO ecosystem libraries also expose similar functionality as pipelines, for example, ZIO JSON for streaming JSON encoding and decoding and ZIO Schema for streaming encoding and decoding for supported data formats.

Beyond this, think about using pipelines versus stream operators as another case of extracting a value versus defining it "inline".

If you're transforming stream values simply or in a way that is highly particular to a particular stream, there may be limited value in extracting that transformation into its own value. In contrast, if the transformation embodies complex logic that you want to use in more than one place, then extracting it out into a pipeline could make a lot of sense.

There is definitely a grey area between these two extremes, but hopefully, this discussion helps you make choices in how to use pipelines that make sense for you and your team and see how these two ways of writing code with streams are really two sides of the same coin, so in close cases, you are in good shape whichever way you choose to go.

### 32.3 Constructing Pipelines

The `ZPipeline` companion object contains various common pipelines corresponding to common stream transformations, such as the `map` pipeline we saw above. It also has more specialized pipelines for various types of encoding and decoding.

You will also find other pipelines in various ZIO ecosystem libraries. For example, ZIO JSON allows you to construct a pipeline that will encode to JSON or decode to JSON your data type as like as an encoder and decoder exists for that data type.

If they exist, you should definitely take advantage of these constructors because they build in a lot of logic for you. But what if you want to implement your own custom pipeline?

The answer is that this is relatively easy because a pipeline is just a channel, so we can implement a pipeline with any logic we want by creating a channel with the appropriate types.

Let's see what this looks like by implementing the `map` channel ourselves:

```
1 | def map[In, Out](f: In => Out): ZPipeline[Any, Nothing, In, Out]
2 | =  
3 | ???
```

Since a pipeline is just a channel, a good first step is to create a placeholder for the channel we need to create:

```

1 def map[In, Out](f: In => Out): ZPipeline[Any, Nothing, In, Out]
2   =
3
4   val channel: ZChannel[Any, ZNothing, Chunk[In], Any, ZNothing,
5     Chunk[Out], Any] =
6     ???
7
8   ZPipeline.fromChannel(channel)
9 }
```

There are a lot of type parameters here, which is one reason it is good to use existing pipeline constructors if possible! However, we can keep them straight if we just substitute our environment, error, input, and output types in the definition of a pipeline.

Now, we're ready to implement our channel. We can use any channel constructors and operators we want, but one pattern that we have found particularly helpful is using the `ZChannel.readWithCause` constructor.

The `ZChannel.readWithCause` constructor, as we may remember from our discussion of channels, will read one input from upstream and allows us to specify functions for handling the element, error, and done cases. By reading, writing, and then recursing, possibly maintaining some internal state, we can implement various stream transformations.

Let's see how this works with the `map` operator. In our implementation of `channel`, we will read from the upstream, which would be the stream we are transforming from:

```

1 def map[In, Out](f: In => Out): ZPipeline[Any, Nothing, In, Out]
2   =
3
4   val channel: ZChannel[Any, ZNothing, Chunk[In], Any, ZNothing,
5     Chunk[Out], Any] =
6     ZChannel.readWithCause(
7       elem => ???,
8       err  => ???,
9       done  => ???
10      )
11
12   ZPipeline.fromChannel(channel)
13 }
```

The error and done cases are typically the easiest, so we can handle them first. If the upstream has completed with an error or done value, then we want to propagate this value to the downstream:

```

1 def map[In, Out](f: In => Out): ZPipeline[Any, Nothing, In, Out]
2   =
3
```

```

2
3   val channel: ZChannel[Any, ZNothing, Chunk[In], Any, ZNothing,
4     Chunk[Out], Any] =
5     ZChannel.readWithCause(
6       elem => ???,
7       err  => ZChannel.failCause(err),
8       done => ZChannel.succeed(done)
9     )
10
11   ZPipeline.fromChannel(channel)
}

```

The element case requires slightly more thought. If we read a chunk of elements from the upstream, we want to transform those elements with the specified function, write them to the downstream, and then do the whole process over again:

```

1 def map[In, Out](f: In => Out): ZPipeline[Any, Nothing, In, Out]
2   = {
3
4   lazy val channel: ZChannel[Any, ZNothing, Chunk[In], Any,
5     ZNothing, Chunk[Out], Any] =
6     ZChannel.readWithCause(
7       elem => ZChannel.write(elem.map(f)) *> channel,
8       err  => ZChannel.failCause(err),
9       done => ZChannel.succeed(done)
10
11   ZPipeline.fromChannel(channel)
}

```

Notice that we had to change `channel` from a `val` to a `lazy val` because we are calling it recursively.

Notice also how this now nicely captures our logic of what it means to transform the elements of this stream. We keep reading elements from the upstream, transforming them with the specified function, and writing them to the downstream, repeating this until we get an error or done value and then terminating with that value.

This example was quite simple because it involved a one-for-one transformation of stream elements, and we didn't have to maintain any internal state, but let's see how we can adapt this for some slightly more complicated examples.

For our next example, we will implement a `filter` pipeline that filters stream elements based on some predicate. This will require us to implement a stream transformation that is not one-for-one because we will omit some elements entirely from the new stream we produce:

```

1 def filter[Elem](f: Elem => Boolean): ZPipeline[Any, Nothing,
2   Elem, Elem] = {

```

```

2
3   lazy val channel: ZChannel[Any, ZNothing, Chunk[Elem], Any,
4     ZNothing, Chunk[Elem], Any] =
5     ZChannel.readWithCause(
6       elem => ???,
7       err  => ZChannel.failCause(err),
8       done => ZChannel.succeed(done)
9     )
10
11   ZPipeline.fromChannel(channel)
12 }
```

We'll start the same way as before, implementing a channel in terms of `readWithCause` and propagating the error and done values.

The only thing we need to change is our logic for what we do when we read new elements. This time, we need to filter the elements according to the predicate, then write to the downstream any elements that satisfy the predicate and repeat.

```

1 def filter[Elem](f: Elem => Boolean): ZPipeline[Any, Nothing,
2   Elem, Elem] = {
3
4   lazy val channel: ZChannel[Any, ZNothing, Chunk[Elem], Any,
5     ZNothing, Chunk[Elem], Any] =
6     ZChannel.readWithCause(
7       elem => {
8         val filtered = elem.filter(f)
9         if (filtered.nonEmpty) ZChannel.write(filtered) *>
10        channel else channel
11       },
12       err  => ZChannel.failCause(err),
13       done => ZChannel.succeed(done)
14     )
15
16   ZPipeline.fromChannel(channel)
17 }
```

Again, we can express this quite clearly in code.

After reading new elements, we filter them to check which ones satisfy the predicate. If any elements satisfy the predicate, we write them and repeat; if none do, we just repeat.

Let's do one more example that requires us to maintain some internal state. Let's say we want to rechunk the stream, so we want to go from a stream that has chunks of arbitrary sizes to a stream where every chunk has the specified size:

```

1 def rechunk[Elem](n: Int): ZPipeline[Any, Nothing, Elem, Elem] =
2   {
```

```

3   lazy val channel: ZChannel[Any, ZNothing, Chunk[Elem], Any,
4     ZNothing, Chunk[Elem], Any] =
5     ZChannel.readWithCause(
6       elem => ???,
7       err  => ZChannel.failCause(err),
8       done => ZChannel.succeed(done)
9     )
10
11   ZPipeline.fromChannel(channel)
}

```

This will require slightly more work.

One thing we notice is that we will have to maintain some internal state, because we might read some elements from the upstream but not have enough elements yet to have a large enough chunk to write to the downstream. So, we will have to maintain some state with the pending elements that have not been written yet.

Let's update the type signature of `channel` to reflect that:

```

1 def rechunk[Elem](n: Int): ZPipeline[Any, Nothing, Elem, Elem] =
2   {
3     def channel(leftovers: Chunk[Elem]): ZChannel[Any, ZNothing,
4       Chunk[Elem], Any, ZNothing, Chunk[Elem], Any] =
5       ZChannel.readWithCause(
6         elem => ???,
7         err  => ZChannel.failCause(err),
8         done => ZChannel.succeed(done)
9       )
10
11   ZPipeline.fromChannel(channel(Chunk.empty))
}

```

Whenever we need to maintain some internal state when we are using this pattern, we make this state a parameter to the channel and create the channel with the initial value. Then, every time the channel calls itself, it will pass in the updated state.

With this updated signature, we can also update our handling of the done and error cases accordingly.

If we read an error or done value from the upstream, we should write the remaining leftovers and then propagate the done value or error. For example, if the original stream is `ZStream(1, 2, 3, 4, 5)` and we are doing `rechunk(2)`, then when we read the done value, we want to write 5 even though the final chunk will be smaller than the target chunk size:

```

1 def rechunk[Elem](n: Int): ZPipeline[Any, Nothing, Elem, Elem] =
2   {

```

```

2
3   def channel(leftovers: Chunk[Elem]): ZChannel[Any, ZNothing,
4     Chunk[Elem], Any, ZNothing, Chunk[Elem], Any] =
5     ZChannel.readWithCause(
6       elem => ???,
7       err  => ZChannel.write(leftovers) *> ZChannel.failCause(err
8     ),
9       done => ZChannel.write(leftovers) *> ZChannel.succeed(done)
10      )
11
12   ZPipeline.fromChannel(channel(Chunk.empty))
13 }
```

Whenever we are maintaining some internal state, we should think about how we want to handle that state when we read an error or a done value.

Now, all that is left is to implement the logic for handling reading a new element.

Conceptually, this should take in the new elements plus any leftovers and return zero or more chunks to write, plus some new leftovers. Then, if there are any chunks to write, we should write those and repeat with the new leftovers.

Often, when logic like this is not trivial, it can be helpful to extract it into its own method:

```

1 def rechunk[Elem](n: Int): ZPipeline[Any, Nothing, Elem, Elem] =
2   {
3
4     def modify(in: Chunk[Elem]): (Chunk[Chunk[Elem]], Chunk[Elem])
5       =
6       ???
7
8     def channel(leftovers: Chunk[Elem]): ZChannel[Any, ZNothing,
9       Chunk[Elem], Any, ZNothing, Chunk[Elem], Any] =
10      ZChannel.readWithCause(
11        elem => {
12          val (toWrite, newLeftovers) = modify(leftovers ++ elem)
13          if (toWrite.nonEmpty) ZChannel.writeChunk(toWrite) *>
14            channel(newLeftovers)
15          else channel(newLeftovers)
16        },
17        err  => ZChannel.write(leftovers) *> ZChannel.failCause(err
18      ),
19        done => ZChannel.write(leftovers) *> ZChannel.succeed(done)
20      )
21
22   ZPipeline.fromChannel(channel(Chunk.empty))
23 }
```

We still have to actually implement the `modify` method here, but now our `modify` method doesn't need to know anything about streams or channels; it just describes how to split a chunk into some new chunks and some leftovers. Similarly, our handler for the element case in `ZChannel.readWithCause` is also quite simple now and basically encapsulates the same logic as in our implementation of `filter` above of writing something if there is something to write and then repeating with the new state.

The final step is to actually implement our `modify` method, which might look like this:

```

1 def rechunk[Elem](n: Int): ZPipeline[Any, Nothing, Elem, Elem] =
2   {
3     def modify(in: Chunk[Elem]): (Chunk[Chunk[Elem]], Chunk[Elem]) =
4       {
5         val grouped = in.grouped(n)
6         val hasLeftovers = grouped.lastOption.fold(false)(_.length == n)
7         if (hasLeftovers) (grouped.dropRight(1), grouped.last) else (
8           grouped, Chunk.empty)
9       }
10
11     def channel(leftovers: Chunk[Elem]): ZChannel[Any, ZNothing,
12               Chunk[Elem], Any, ZNothing, Chunk[Elem], Any] =
13       ZChannel.readWithCause(
14         elem => {
15           val (toWrite, newLeftovers) = modify(leftovers ++ elem)
16           if (toWrite.nonEmpty) ZChannel.writeChunk(toWrite) *>
17             channel(newLeftovers)
18           else channel(newLeftovers)
19         },
20         err => ZChannel.write(leftovers) *> ZChannel.failCause(err),
21         done => ZChannel.write(leftovers) *> ZChannel.succeed(done)
22       )
23
24     ZPipeline.fromChannel(channel(Chunk.empty))
25   }

```

This idea of reading, writing, and repeating, possibly maintaining some internal state along the way, should give you a good sense of how you can implement your own custom pipelines if you need to. But again, there are a large variety of built-in pipeline constructors for you, so you should definitely take a look at those first!

## 32.4 Conclusion

At this point, you know what you need to know to be extremely productive working with pipelines.

We have seen that pipelines are another kind of channel that transforms one stream of elements to another.

We can compose pipelines with each other much like we do functions to get new pipelines that apply the transformation of the first pipeline and then the transformations of the second pipeline. We can also compose pipelines with streams to transform the stream's output or compose them with sinks to transform the input to the sink.

We saw some of the existing pipeline constructors we can use as well as how we can implement our own pipelines by repeatedly reading from the upstream and writing to the downstream.

Finally, we discussed how many stream transformations are both pipelines and operators on streams and some of the advantages and disadvantages of using one approach or the other. They are ultimately the same, so the real answer is do whatever works for you and your team!

## 32.5 Exercises

1. Create a pipeline that groups consecutive elements into pairs:

```
1 def pair[A]: ZPipeline[Any, Nothing, A, (A, A)] =  
2   ???
```

2. Design a pipeline that outputs the minimum and maximum values from a continuous data stream within a fixed time window (e.g., every minute).
3. Assume you have streams of timestamped `UserEvent` and write a stream sessionization pipeline that groups events into sessions based on an inactivity gap between events. A session is considered ended if there is no event for a given time window.

```
1 import java.time.Instant  
2  
3 case class UserEvent(  
4   userId: String,  
5   timestamp: Instant,  
6 )  
7  
8 case class Session(  
9   sessionId: String,  
10  userId: String,  
11  startTime: Instant,  
12  endTime: Instant,  
13  duration: Duration,
```

```
14     events: List[UserEvent]
15 )
16
17 def sessionize(
18     gapThreshold: Duration
19 ): ZPipeline[Any, Nothing, UserEvent, Session] =
20     ???
```

# Chapter 33

## Streaming: Sinks

As discussed previously, a sink is a composable aggregation strategy. In this chapter, we will look more at what this means and how we can use sinks, compose them, and construct them.

### 33.1 Sinks as Composable Aggregation Strategies

A sink represents a strategy for aggregating zero or more elements into a summary value. We can see this from the definition of a sink:

```
1 final case class ZSink[-R, +E, -In, +L, +Z] (
2   channel: ZChannel[R, ZNothing, Chunk[In], Any, E, Chunk[L], Z]
3 )
```

Let's unpack this a little more to see what it means.

A sink can receive two types of inputs.

First, it can receive zero or more element values of type `Chunk[In]`. These represent the elements that the sink is aggregating.

For efficiency, the sink receives chunks of elements instead of single elements. This is not fundamental, but it is something we will have to deal with sometimes when working with sinks.

Second, it can receive exactly one done value of type `Any`. This indicates that the done value does not contain any meaningful information other than that there will be more input elements.

We need the done value because the input elements might be produced asynchronously. So, in the absence of the done value, the sink would have no way of knowing that there would not be more elements in the future.

With these two types of inputs, we can represent zero elements as simply a done value, any finite number of elements as one or more chunks of elements followed by a done value, and an infinite number of inputs as one chunk of elements followed by another forever.

Notice that a sink cannot receive any input for its error type. The input error type of a sink is `ZNothing` which is just a subtype of `Nothing` with better type inference:

```
1 | type ZNothing <: Nothing
```

Since there are no values of type `Nothing`, there are also no values of type `ZNothing`, and so a sink can never receive an input error value.

This reflects the fact that sinks are composable strategies for aggregating *elements*. Sinks are not strategies for handling errors.

There could be some other data type built on channels that represent a strategy for handling errors, but sinks are designed to be strategies for aggregating elements and are very good at that.

Moving on to the output types, we see that a sink will eventually terminate, if at all, with either a summary value of type `Z` or an error of type `E`.

The summary value type `Z` represents the result of the aggregation. For example, if we are counting the number of elements, then it might be a `Long`, or if we are summing them, it might be a `Double`.

The error type `E` represents the potential errors that can occur during the aggregation. This would be 'Nothing' in simple cases like counting or summing because no errors are possible. However, if the aggregation was writing the contents to a file, the error type might be `IOException` to reflect that writing to the file could fail.

In addition to either producing a summary value or failing with an error, a sink may emit zero or more leftover values of type `Chunk[L]`. These represent inputs that were received by the sink but were not included in the aggregation.

Leftovers can arise for a couple of reasons.

Chunking can lead to leftovers when a sink does not need to all of the elements in the chunk to produce a summary value.

For example, consider the sink constructed using `ZSink.take(3)`:

```
1 | import zio._
2 | import zio.stream._
3 |
4 | val sink: ZSink[Any, Nothing, Int, Int, Chunk[Int]] = 
5 |   ZSink.take(3)
```

This sink wants to aggregate three values into a chunk. What happens if the first input element is `Chunk(1, 2)` and the second is `Chunk(3, 4)`?

Clearly, the first input element is not sufficient to produce the aggregated value since, at this point, the sink has only seen two values and does not have enough inputs to produce

a chunk of three values.

The sink will consume the next input element, `Chunk(3, 4)`. At this point, the sink has enough values to produce the aggregate `Chunk(1, 2, 3)`.

But what about the value 4? The sink received this element but did not include it in its aggregate, so what are we supposed to do with it?

Leftovers can also arise even without chunking. For example, consider the sink constructed using `ZSink.collectAllWhile(_ == "a")`:

```

1 val sink: ZSink[Any, Nothing, String, String, Chunk[String]] =
2   ZSink.collectAllWhile(_ == "a")
3 // sink: ZSink[Any, Nothing, String, String, Chunk[String]] = zio
  .stream.ZSink@b5c4f70b

```

If the inputs to this sink are `Chunk("a")` followed by `Chunk("b")`, the sink will include the first element in the aggregation. However, the sink is not done yet because the next value is potentially also `"a"`. So, the sink consumes the next input element, which is a `Chunk("b")`.

At this point, the sink knows that it is done because it has seen a value that is not `"a"` so it produces the aggregate `Chunk("a")`. But what should it do with the value `"b"` that it received but did not include in the aggregate?

One alternative would be to simply drop it.

In some cases, this might be acceptable. For example, if we are just doing `stream.run(ZSink.take(3))`, then we are only interested in the first three elements from the stream and can simply ignore the rest.

However, we will find in other cases that we want to preserve the leftovers for further processing. To support this, we need to include the leftovers in the output of the sink so that the user of the sink can decide what to do with them.

We do this by having the sink return any leftovers in its output element channel as leftovers of type `Chunk[L]`.

The leftover type `L` will typically be the same as the input type `In` if there are any leftovers. However, having separate type parameters for the inputs and the leftovers, improves type inference because it avoids making the element type invariant, which it would have to be if it appeared as both an input and an output.

It also allows us to be more precise by specifying `Nothing` for the leftover type if there can be no leftovers. For example, a sink that counts all inputs should never produce leftovers.

## 33.2 Using Sinks

Now that we have a better understanding of what a sink is, let's look at some different ways we can use sinks.

### 33.2.1 Running Streams into Sinks

The most common way we can use a sink is by running a stream into it using the `ZStream #run` operator:

```

1 val stream: ZStream[Any, Nothing, Int] =
2   ZStream.fromChunks(Chunk(1, 2), Chunk(3, 4))
3
4 val sink: ZSink[Any, Nothing, Int, Int, Chunk[Int]] =
5   ZSink.take(3)
6
7 stream.run(sink)

```

Now that we better understand sinks, we can be more precise about what the `run` operator is doing.

Recall that a stream is just a channel that emits zero or more elements of type `Chunk[A]` and eventually terminates with either a done value of type `Any` or an error of type `E`:

```

1 final case class ZStream[-R, +E, +A](channel: ZChannel[R, Any,
                                         Any, E, Chunk[A], Any])

```

So we can think of the stream above as a program that just emits `Chunk(1, 2)` followed by `Chunk(3, 4)` and then terminates with a done value.

Streams are pull-based so when we run this stream into the sink above, the sink first reads an element from the stream, which is `Chunk(1, 2)`. This is not enough input for the sink to produce a summary value, so the sink reads another element from the stream, which is `Chunk(3, 4)`.

At this point, the sink has enough inputs to produce the summary value `Chunk(1, 2, 3)`. The sink terminates with this value without reading from the stream again and returns the aggregation as the result of the `run` operator, discarding any leftovers.

This pattern of the sink repeatedly reading from the stream is just the `pipeTo` operator from channels we saw earlier that connects the output of one channel to the input of another. All the `run` operator is doing is piping the stream's output to the input of the sink and then running that channel to produce its done value!

The only complication is that the stream potentially fails with an error of type `E`, but the sink does not handle errors. To deal with this, we just use a variant of `pipeTo` called `pipeToOrFail` that fails with the error of the first channel if it fails without passing it through to the second channel.

So the implementation of `run` is just:

```

1 trait ZStream[-R, +E, +A] {
2   def run[R1 <: R, E1 >: E, Z](sink: => ZSink[R1, E1, A, Any, Z])
3     : ZIO[R1, E1, Z] =
4     stream.channel.pipeToOrFail(sink.channel).runDrain
}

```

This reflects exactly what we described above. Running a stream into a sink, pipes all the outputs of the stream to the inputs of the sink and then runs the resulting channel to produce the done value.

Building on this idea of piping the stream into the sink, we can also use the symbolic operator `>>>` to connect a stream to a sink instead of using the `run` operator. So instead of `stream.run(sink)`, we could also do `stream >>> sink`, which, depending on our appetite for symbolic operators, could more visually represent the concept of sending elements from the stream to the sink.

### 33.2.2 Transducing Streams with Sinks

Running a stream into a sink is the simplest way to use a sink, but it is not the only way. One of the benefits of defining aggregation strategies as first-class values with sinks is that we can use them in multiple places.

Another common way of using sinks is aggregating stream values multiple times using the `ZStream#transduce` operator:

```

1 val stream: ZStream[Any, Nothing, Int] =
2   ZStream.fromChunks(Chunk(1, 2), Chunk(3, 4))
3
4 val sink: ZSink[Any, Nothing, Int, Int, Chunk[Int]] =
5   ZSink.take(3)
6
7 val transduced: ZStream[Any, Nothing, Chunk[Int]] =
8   stream.transduce(sink)

```

The `ZStream#transduce` operator will feed each of the elements of the stream to the sink, just like `run` did. However, when the sink is done, `ZStream#transduce` will not terminate but instead feed any leftovers and the remaining elements to the sink again, repeating the process until the stream is done.

Each of the aggregations will be emitted incrementally as a new stream. So here, the result of transducing will be a stream that emits the first aggregate `Chunk(1, 2, 3)`, then emits the aggregate of the remaining values `Chunk(4)`, and finally emits a done value.

In this case, only two aggregates were emitted because the stream was done while the sink was running a second time, but the stream could potentially be infinite. Each time we read an element from the transduced stream, the sink would run, reading elements from the original stream until it could produce an aggregate and then writing that aggregate.

This idea of repeatedly running the sink illustrates another concept that we have discussed before but bears repeating. Sinks are descriptions or “blueprints” for aggregating values.

We can run a sink repeatedly, each time aggregating new elements with that strategy. That is why we can run the same sink multiple times in `ZStream#transduce`.

The `ZStream#transduce` operator is an example of where we get the payoff of the leftover type of sinks. If sinks did not emit their leftovers, then in the example above, we would

drop the value 4, which would make sinks much less useful for aggregating multiple stream elements and would also make the results of the aggregation depend on chunking, which is supposed to be an implementation detail.

The `transduce` operator is very powerful in allowing us to aggregate stream elements in different ways.

For example, transducing with `ZSink.take` allows us to group stream elements into chunks of a certain size. This could be helpful if we had a stream of records that we wanted to write somewhere in batches of a specified size.

As we will see below, when we look at ways of combining sinks, it also allows us to use sinks as highly composable incremental parsers.

For example, if we had a stream of characters representing run-length encoded DNA bases, we could have a sink that consumes enough elements to decode one base:

```

1 sealed trait Base
2
3 object Base {
4   case object A extends Base
5   case object C extends Base
6   case object G extends Base
7   case object T extends Base
8 }
9
10 val decoder: ZSink[Any, Nothing, Char, Char, Base] =
11   ???

```

With this, we could then decode a stream of arbitrary length simply by transducing the stream with the decoder:

```

1 val encoded: ZStream[Any, Nothing, Char] =
2   ???
3
4 val decoded: ZStream[Any, Nothing, Base] =
5   encoded.transduce(decoder)

```

We will look more in the next section on combining sinks at how we could actually implement this decoding sink.

### 33.2.3 Asynchronous Aggregation with Sinks

The `ZStream#transduce` operator is quite helpful when the aggregation we want to do depends solely on the elements we are aggregating. For example, we saw above that we can quite easily group a stream into chunks of a specified size using `ZStream#transduce` and `ZSink.take`.

However, frequently there is an element of time involved in determining the aggregation.

If we are consuming records from a Kafka topic, we may want to aggregate them into batches of a specified size for efficiency. But what happens if there are not enough records in some time interval to reach the specified size?

With `ZStream#transduce`, we will just wait until we have the specified number of records before emitting the batch, potentially indefinitely. This is likely unacceptable for many use cases because it trades off potentially unlimited latency in exchange for always achieving the specified batch size for maximum throughput.

Instead, we would like to specify both the number of records to aggregate and the maximum duration we will wait for aggregation. If we get enough records before the duration elapses, we immediately emit them; otherwise, we emit the records we have aggregated after the duration has elapsed.

We can do this using the `ZStream#aggregateAsyncWithin` operator, which allows us to specify both a `ZSink` describing an aggregation strategy and a `Schedule` describing a “timeout” for that aggregation strategy:

```

1 trait ZStream[-R, +E, +A] {
2   def aggregateAsyncWithin[R1 <: R, E1 >: E, A1 >: A, B] (
3     sink: => ZSink[R1, E1, A1, A1, B],
4     schedule: => Schedule[R1, Option[B], Any]
5   )(implicit trace: Trace): ZStream[R1, E1, B]
6 }
```

With `ZStream#aggregateAsyncWithin`, we divide the stream into two asynchronous regions, “upstream” and “downstream” of the aggregation.

The sink will read elements from the upstream until either the sink is done or the schedule recurs.

If the sink is done first, then the aggregated value will be emitted downstream, the previous schedule timeout will be canceled, and the process will be repeated with the sink being run again and the next recurrence of the schedule. This would correspond to receiving the specified number of records, emitting them, and repeating the process.

If the schedule is done first, we write a done value to the sink, causing it to write its aggregated value to the downstream immediately. Then, we run the sink again and the next recurrence of the schedule. This would correspond to the timeout elapsing, emitting the records we have aggregated so far, and repeating the process.

The `ZStream#aggregateAsyncWithin` operator is quite flexible. As you can see from the type signature above, it allows for complex patterns, such as varying timeout durations based on the `Schedule` and the timeout durations depending on the aggregated value. However, for most cases, a simple schedule such as `Schedule.spaced` will do what you want.

### 33.2.4 Tapping Streams with Sinks

The final common way to use sinks is to tap a stream with a sink:

```

1 trait ZStream[-R, +E, +A] {
2   def tapSink[R1 <: R, E1 >: E](
3     sink: => ZSink[R1, E1, A, Any, Any]
4   ): ZStream[R1, E1, A]
5 }
```

Tapping a stream with a sink is similar to tapping a stream with a ZIO workflow. We send each element of the stream to the sink, but instead of returning the summary value or leftovers of the sink, we just do this “on the side” and emit the original values of the stream.

For example, if we wanted to render each element of a stream to the console for debugging purposes, we could do it using the `ZStream#tap` operator on streams like this:

```

1 val stream: ZStream[Any, Nothing, Int] =
2   ZStream(1, 2, 3)
3
4 val tapped: ZStream[Any, Nothing, Int] =
5   stream.tap(n => Console.printLine(n))
```

This will send each element of the stream to the effectful function `Console.printLine` but also emit each of those elements unchanged.

We can do the same thing with `ZStream#tapSink` like this:

```

1 val stream: ZStream[Any, Nothing, Int] =
2   ZStream(1, 2, 3)
3
4 val sink: ZSink[Any, Nothing, Any, Nothing, Unit] =
5   ZSink.foreach(Console.printLine(_))
6
7 val tapped: ZStream[Any, Nothing, Int] =
8   stream.tapSink(sink)
```

Like in the example above, each stream element will be sent to the sink, but the elements will also be emitted unchanged.

The `ZStream#tapSink` operator is helpful when we want to do something “on the side” with the elements of a stream and we want to take advantage of the power of a sink.

For example, perhaps we want to write all the elements from the middle of our data processing pipeline to a file. If we have a sink that describes writing to that file we could use `ZStream#tapSink` to accomplish this quickly and easily, with the guarantee that the file would be opened once and closed no matter what.

We would not be able to do this with the same efficiency with `ZStream#tap` since each element would require evaluating the workflow. In contrast, `ZStream#tapSink` allows us to use the sink to maintain the aggregation state and handle any resources associated with it across handling different stream elements.

### 33.3 Combining Sinks

We have seen that one advantage of defining aggregation strategies as first-class values with sinks is that we can use them in different ways, like the `ZStream#run`, `ZStream#transduce`, `ZStream#aggregateAsyncWithin`, and `ZStream#tapSink` operators above. Another advantage is that we can define operators on sinks to transform and compose them, allowing us to build sinks with more complex aggregation strategies than simpler ones.

So far, we have been using simple sink constructors such as `ZSink.take` for illustrative purposes. However, sinks also come with various operators for composing them, which we will look at here.

Many of these operators may look familiar, but some have additional features because sinks both read inputs and write summary values. We will go through the main ways of transforming and composing sinks in turn.

#### 33.3.1 Transforming Inputs and Outputs

Some of the simplest operators on sinks just allow us to transform their inputs and outputs:

```

1 trait ZSink[-R, +E, -In, +L, +Z] {
2   def map[Z2](f: Z => Z2): ZSink[R, E, In, L, Z2]
3   def mapLeftover[L2](f: L => L2): ZSink[R, E, In, L2, Z]
4   def mapZIO[R1 <: R, E1 >: E, Z1](
5     f: Z => ZIO[R1, E1, Z1]
6   ): ZSink[R1, E1, In, L, Z1]
7   def contramap[In1](f: In1 => In): ZSink[R, E, In1, L, Z]
8   def contramapZIO[R1 <: R, E1 >: E, In1](
9     f: In1 => ZIO[R1, E1, In]
10  ): ZSink[R1, E1, In1, L, Z]
11 }
```

These let us adapt the inputs of a sink and the outputs of a sink to match the type of the stream or to do further processing on the aggregated value.

For example, if we have a stream of temperature readings, we might want to group them into chunks of a hundred readings and then calculate the average for each chunk. We could easily do this using the `take` constructor we have seen before and `map` to transform the outputs:

```

1 val averageTemperatureSink: ZSink[Any, Nothing, Double, Double,
2   Option[Double]] =
3   ZSink.take[Double](100).map { chunk =>
4     if (chunk.isEmpty) None
5     else Some(chunk.sum / chunk.size)
6 }
```

We could then transduce a stream of temperature readings with this sink to get a stream of average temperatures:

```

1 val temperatures: ZStream[Any, Nothing, Double] =
2   ???
3
4 val averageTemperatures: ZStream[Any, Nothing, Double] =
5   temperatures.transduce(averageTemperatureSink).collectSome

```

Similarly, if, for some reason, the temperature readings were in the form of `String` values, and we wanted the sink to be responsible for parsing them, we could use `ZSink#contramap` to transform the inputs of the sink. Because the parsing could fail, we will use `ZSink#contramapZIO` to model the possibility of this failure:

```

1 val decodingAverageTemperatureSink: ZSink[Any,
2   NumberFormatException, String, Double, Option[Double]] =
3   averageTemperatureSink.contramapZIO { string =>
4     ZIO.attempt(string.toDouble).refineToOrDie[
5       NumberFormatException]
6   }

```

Note that the type signature is somewhat awkward here because we have transformed the input type but not the leftover type. So, depending on how we use the sink, it could be an issue that the input and leftover types do not line up. We can fix this by using the `ZSink#mapLeftover` operator to transform the leftover type as well:

```

1 val decodingAverageTemperatureSink: ZSink[Any,
2   NumberFormatException, String, String, Option[Double]] =
3   averageTemperatureSink
4   .contramapZIO { string =>
5     ZIO.attempt(string.toDouble).refineToOrDie[
6       NumberFormatException]
7   }
8   .mapLeftover(_.toString)

```

Now, we can easily use the sink to parse and aggregate a stream of raw temperature readings:

```

1 val encodedTemperatures: ZStream[Any, Nothing, String] =
2   ???
3
4 val averageTemperatures: ZStream[Any, NumberFormatException,
5   Double] =
6   encodedTemperatures.transduce(decodingAverageTemperatureSink).
7   collectSome

```

### 33.3.2 Sequential Composition Of Sinks

In addition to transforming the inputs and outputs of sinks, we can compose sinks sequentially using the `ZSink#flatMap` operator. As with other data types we have seen, `ZSink#flatMap` corresponds to “run this sink, then use its summary value to construct another sink and run that sink”.

The `ZSink#flatMap` operator can be useful when describing more complex aggregation strategies where the aggregation strategy we should use depends on the results of a previous aggregation step.

For example, let’s go back to our run-length encoding of DNA example. We have four bases, “A”, “C”, “G”, and “T”, and each of them will be followed by an integer indicating how many times the base is repeated.

So, an encoded sequence of bases might look like this:

```
1 val encoded: Chunk[String] =
2   Chunk("A", "3", "C", "2", "G", "1", "T", "1")
```

And the desired output would be:

```
1 sealed trait Base
2
3 object Base {
4   case object A extends Base
5   case object C extends Base
6   case object G extends Base
7   case object T extends Base
8 }
9
10 val decoded: Chunk[Base] =
11   Chunk(A, A, A, C, C, G, T)
```

Conceptually, we need to run a sink that parses a sink base first, then use that sink to construct another sink that parses an integer and writes out the base that many times. Let’s start by constructing the two individual sinks, and then we can combine them with `ZSink#flatMap`.

The sink that parses a single base pair is relatively simple. We will use the `ZSink.head` constructor to construct a sink that reads a single element of input and then use `ZSink#mapZIO` to either transform that input into a decoded `Base` or fail with a `DecodingError`:

```
1 case object DecodingError
2
3 val baseDecoder: ZSink[Any, DecodingError, String, String, Option[Base]] =
4   ZSink.head[String].mapZIO {
5     case Some("A") => ZIO.succeed(Some(Base.A))
6     case Some("C") => ZIO.succeed(Some(Base.C))
```

```

7   case Some("G") => ZIO.succeed(Some(Base.G))
8   case Some("T") => ZIO.succeed(Some(Base.T))
9   case None      => None
10  case _          => ZIO.fail(DecodingError)
11 }

```

The sink that parses an integer is also fairly straightforward:

```

1 val lengthDecoder: ZSink[Any, DecodingError, String, String,
  Option[Int]] =
2   ZSink.head[String].mapZIO {
3     case Some(string) => ZIO.attempt(Some(string.toInt)).asError(
4       DecodingError)
5     case None => ZIO.succeed(None)
6   }

```

Now we can combine these two sinks using `ZSink#flatMap` to construct a new sink that reads a base, then reads an integer, and then succeeds with a summary value with that base repeated that many times:

```

1 val decoder: ZSink[Any, DecodingError, String, String, Option[
  Chunk[Base]]] =
2   baseDecoder.flatMap {
3     case Some(base) =>
4       lengthDecoder.mapZIO {
5         case Some(length) => ZIO.succeed(Some(Chunk.fill(length)(
6           base)))
6         case None          => ZIO.fail(DecodingError)
7       }
8     case None =>
9       ZSink.succeed(None)

```

Let's try to use it to decode our input:

```

1 val encoded =
2   ZStream("A", "3", "C", "2", "G", "1", "T", "1")
3
4 val decoded =
5   encoded.transduce(decoder).collectSome.flattenChunks

```

And we do indeed get our expected output!

### 33.3.3 Parallel and Concurrent Composition of Sinks

We can also compose sinks in parallel or even race them using the `zipPar` and `race` operators on sinks.

When we use these operators, unlike the sequential composition of sinks discussed above, elements will be sent to both sinks. In this way, these operators are somewhat similar to the

Hub data type we in that stream elements are “broadcast” to multiple sinks and processed by each of them.

This can be useful when we have multiple ways to aggregate stream values and want to run all of them.

For example, perhaps we want to send all the results of our data flow to a persistence service, a logging service, and an analytics service. We can represent sending elements to each of those services as a sink and combine them all together with `zipPar` to create a new sink that describes sending them to all three services.

We can also use `race` to aggregate stream elements in two different ways and terminate as soon as one of the aggregations has completed, safely interrupting the other. For example, perhaps there are two nodes that data can be sent to, but as long as the data is sent to one node successfully, it will automatically be replicated to the other.

## 33.4 Constructing Sinks

The final step in learning about sinks is looking at how to construct sinks.

The `ZSink` companion object contains various sinks that are helpful for common use cases or as building blocks for building more complex aggregations.

Many of these constructors correspond to ways of reducing ordinary Scala collections to a summary value. This builds on the analogy that a stream is like a potentially infinite collection, though as we have seen, a stream is also much more than this.

For example, there are constructors such as `ZSink.take` to take a specified number of values, `ZSink.head` to take the first value if it exists, `ZSink.count` to count the number of values, `ZSink.sum` to sum numeric values, and so on. Many of these are implemented in terms of more basic operators such as `ZSink#foldLeft`, just as many collection operators can be.

There are also some more complex constructors that reflect the incremental nature of streams.

For instance, there is a `ZSink.foldWeighted` constructor that allows creating a sink that aggregates value until a certain total “size” is reached, where a function can be provided to compute the size of each element. This would be similar to our example above, where we wanted to aggregate a certain number of records, but the records themselves might have very different sizes.

While these basic building block constructors are helpful, some of the most powerful sink constructors are specialized to specific domains and wrap up all the logic for sending values to destinations in those domains.

Some of these that are relatively generic are included in the ZIO Stream itself.

For example, a `ZSink.fromFile` constructor will create a sink that will write input to a file, automatically opening the file before writing and closing the file after. There are

also `ZSink.fromHub` and `ZSink.fromQueue` constructors that will send all elements to a specified Hub or Queue.

Some of the most powerful sinks are included in other ZIO libraries. ZIO Connect, in particular, is developing a variety of streams and sinks to describe ways to read data from and write data to a variety of other sources and destinations, much as Alpakka does for the Akka ecosystem.

With these tools, the vision is that it is easy for you to get data from wherever it is and send it to wherever it needs to go so you can focus on implementing your logic of transforming, enriching, and aggregating that data.

Most of the time, using these off-the-shelf sinks and the ways for transforming and combining them, discussed previously in this chapter, will be the way to go. However, if you are implementing your own sinks, you may want to use the low-level representation of a sink as a channel to do so.

In this final section of this chapter, we will look at an example of how to do that. Feel free to skip over this section and go directly to the next chapter if you are just interested in using sinks and are not interested in implementing your own custom sinks right now.

For this section, we will use a relatively simple example of implementing a sink that computes an average of numeric values.

We could implement this easily ourselves in terms of `ZSink.count`, `ZSink.sum`, and the `ZSink#zipWithPar` operator on sinks:

```

1 val average: ZSink[Any, Nothing, Double, Double, Option[Double]] =
2   =
3   ZSink.sum[Double].zipWithPar(ZSink.count[Double]) { (sum, count
4     ) =>
5     if (count == 0) None
6     else Some(sum / count)
7 }
```

We could also implement it in terms of existing sink constructors such as `ZSink.foldLeft`:

```

1 val average: ZSink[Any, Nothing, Double, Double, Option[Double]] =
2   =
3   ZSink.foldLeft[Double, (Double, Int)]((0.0, 0)) { case ((sum,
4     count), value) =>
5     (sum + value, count + 1)
6   }.map { case (sum, count) =>
7     if (count == 0) None
8     else Some(sum / count)
9 }
```

But let's assume that we didn't want to do either of those and see how we could implement it ourselves in terms of channels. We will use the pattern we have seen before in implement-

ing channel operators by creating a channel that reads input, writes output, and recursively calls itself when necessary:

```

1 val average: ZSink[Any, Nothing, Double, Double, Option[Double]] =
2   {
3     def channel(sum: Double, count: Long): ZChannel[Any, ZNothing,
4       Double, Any, Nothing, Option[Double]] = =
5       ZChannel.readWithCause(
6         in => channel(sum + in.sum, count + in.length),
7         err => ZChannel.failCause(err),
8         done =>
9           if (count == 0) ZChannel.succeed(None)
10          else ZChannel.succeed(Some(sum / count))
11        )
12      ZSink.fromChannel(channel(0.0, 0L))
13    }

```

This gives us another example of what it means to be a sink from the beginning of a channel. A sink reads zero or chunks of input and eventually either fails with an error or succeeds with a summary value, as well as potentially some leftovers.

Here, the sink cannot fail, and there are no leftovers, so the implementation is relatively simple.

We read to get more input, and if there is more input, we update our internal state with that input and read again. If there is no more input, then we use our internal state to compute the summary value and succeed with it.

## 33.5 Conclusion

In this chapter, we've seen how sinks provide powerful ways to describe aggregation strategies as values. We can define potentially complex aggregation strategies and then use them in multiple places in our program.

We can also use these aggregation strategies in different ways. The simplest way is to run a stream using a sink, essentially aggregating all of the values produced by the stream. But we can also use `ZStream#transduce` to repeatedly run a sink with the elements of a stream and produce a new stream of all the aggregates, or `ZStream#aggregateAsyncWithin` to aggregate all the values within a given time period.

Because sinks are first-class data types, they also have their own operators. We can combine them with each other to produce new sinks with more complex logic that use the result of one aggregation to determine the next aggregation strategy. We can even run sinks in parallel or race them with each other.

Hopefully, this chapter has given you a better sense of the power of sinks and how you can

use them in your code.

## Chapter 34

# Observability: Logging

Observability is the practice of designing and monitoring systems to gain deep insights into their internal states and behaviors. It is a critical aspect of modern software development, especially in distributed systems, microservices, and cloud-native applications.

When it comes to observability, there are three main pillars to consider: 1. **Logging**: Capturing, storing, and analyzing textual informational events generated by the system as the execution flow of the program progresses. Logs can help identify errors, exceptions, and issues in the system by analyzing events and actions taken by the system. 2. **Metrics**: Collecting and analyzing numerical values that typically quantify the system's state periodically. Metrics can help identify trends, patterns, and anomalies in the system's behavior by tracking key performance indicators (KPIs). 3. **Tracing**: Capturing and analyzing information that provides a detailed view of the execution flow and interactions between components in a concurrent and distributed system. Tracing can help identify bottlenecks, latency issues, and performance problems in a system by tracking the flow of requests across different services.

In this chapter, we will focus on logging, which is a fundamental aspect of observability.

In modern distributed systems, logging is crucial for understanding application behavior, debugging issues, and monitoring system health. ZIO Logging stands out by providing a comprehensive set of features that address the challenges of logging in concurrent and distributed applications:

- **Pluggable Backends** offer the flexibility to integrate with various logging implementations. ZIO provides a flexible built-in logging system that can work both independently and integrate seamlessly with existing logging frameworks. This means you can start with ZIO's native logging capabilities for simpler applications and easily transition to more robust logging frameworks like SLF4J, Log4j, or other solutions as your application grows. This flexibility ensures that you can maintain your existing logging infrastructure while taking advantage of ZIO's powerful logging features.

- **Log Annotations** enable you to enrich your logs with additional context. Rather than just recording what happened, annotations help you understand the full context of each log entry. You can tag logs with request IDs, user information, or any other metadata that helps trace and debug your application.
- **Log Spans** allow you to group related log entries together. ZIO Logging automatically calculates the execution duration of each span and includes this timing data in the logs with its corresponding span label. This automatic timing becomes invaluable when tracking operations that span multiple components or services, making it easier to understand complex workflows, diagnose performance issues, and identify bottlenecks in your system.
- **Structured Logging** moves beyond simple text messages to provide machine-readable log entries. By structuring your logs in a consistent format, you can easily parse, query, and analyze them using modern log management tools. This becomes increasingly important as applications scale and the volume of logs grows.
- **Logger Context**, implemented using `FiberRef`, provides a powerful way to maintain logging context across asynchronous operations. This feature, which includes support for Mapped Diagnostic Context (MDC), ensures that contextual information follows the logical flow of your application, even across different fibers and thread boundaries.

These features combine to create a logging system that's not just about recording events but about providing deep insights into your application's behavior. In the following sections, we'll explore each of these features in detail, showing you how to leverage them effectively in your ZIO applications.

## 34.1 Understanding the Basics

Logging events in ZIO is straightforward. The `ZIO.log` operator takes a message and logs it with the current log level:

```
1 import zio._

2

3 object LoggingExample extends ZIOAppDefault {
4     def run =
5         for {
6             _    <- ZIO.log("Application started!")
7             name <- Console.readLine("Enter your name: ")
8             _    <- ZIO.log(s"User entered name: $name")
9             _    <- Console.printLine(s"Hello, $name!")
10            _    <- ZIO.log("Application finished!")
11        } yield ()
12 }
```

When we run this program, we'll see the following output:

```

1 timestamp=2024-12-26T16:19:34.781889347Z level=INFO thread=#zio-
   fiber-1488561411 message="Application started!" location=<
     empty>.LoggingExample.run file=LoggingExample.scala line=6
2 Enter your name: John
3 timestamp=2024-12-26T16:19:40.951080795Z level=INFO thread=#zio-
   fiber-1488561411 message="User entered name: John" location=<
     empty>.LoggingExample.run file=LoggingExample.scala line=8
4 Hello, John!
5 timestamp=2024-12-26T16:19:40.953444216Z level=INFO thread=#zio-
   fiber-1488561411 message="Application finished!" location=<
     empty>.LoggingExample.run file=LoggingExample.scala line=10

```

Each log entry includes a timestamp, log level, fiber identifier, message, source location, file, and line number. These informations are useful for analyze logs when using log management tools.

While ZIO.log uses INFO as its default log level, you can modify this using the ZIO.logLevel operator:

```

1 import zio._
2
3 ZIO.logLevel(LogLevel.Trace) {
4   for {
5     _    <- ZIO.log("Application started!")
6     name <- Console.readLine("Enter your name: ")
7     _    <- ZIO.log(s"User entered name: $name")
8     _    <- Console.printLine(s"Hello, $name!")
9     _    <- ZIO.log("Application finished!")
10 } yield ()
11 }

```

Log levels are contextual, following the same principles discussed in chapter 19, “Contextual Data Types”. This allows us to override the log level for specific code regions. When exiting a region, the log level reverts to its previous value:

```

1 import zio._
2
3 ZIO.logLevel(LogLevel.Trace) {
4   for {
5     _    <- ZIO.log("Application started!") // logging at TRACE
6     level
7     _    <- ZIO.logLevel(LogLevel.Info) {
8       for {
9         name <- Console.readLine("Enter your name: ")
10        _    <- ZIO.log(s"User entered name: $name") // logging at
11        INFO level
12        _    <- Console.printLine(s"Hello, $name!")
13     } yield ()
14   }
15 }

```

```

12     }
13     _ <- ZIO.log("Application finished!") // logging at TRACE
14     level
15   } yield ()
}

```

In this example, we set the log level to TRACE for the outer block but override it to INFO for the inner block. Consequently, log entries inside the inner block use the INFO level, while the rest use the TRACE level.

ZIO also provides dedicated operators for specific log levels such as `ZIO.logTrace`, `ZIO.logDebug`, `ZIO.logInfo`, `ZIO.logWarning`, `ZIO.logError`, and `ZIO.logFatal` operators. Let's try an example:

```

1 import zio._
2
3 def processContent(content: String): ZIO[Any, Throwable, Unit] =
4   ???
5
6 ZIO.readFile("file.txt").tapBoth(
7   e => ZIO.logError(s"Failed to read file: $e"),
8   content => processContent(content)
9 )

```

Each of these operators has a variant that allows logging messages with the cause of the event, such as `ZIO.logErrorCause`:

```

1 ZIO.readFile("file1.txt")
2   .tapErrorCause(cause => ZIO.logErrorCause("Failed to read file"
3     , cause))
4   .orElse(
5     ZIO.readFile("file2.txt")
6   )

```

Here's a sample output when `file1.txt` is not found:

```

1 timestamp=2024-12-29T07:16:34.300184498Z level=ERROR thread=#zio-
2   fiber-1231259327 message="Failed to read file" cause=
3     Exception in thread "zio-fiber-1231259327" java.io.
4     FileNotFoundException: file1.txt (No such file or directory)
5       at java.base/java.io.FileInputStream.open0(Native Method)
6       at java.base/java.io.FileInputStream.open(FileInputStream.
7         java:219)
8       at java.base/java.io.FileInputStream.<init>(FileInputStream.
9         java:159)
10      at scala.io.Source$.fromFile(Source.scala:94)
11      at scala.io.Source$.fromFile(Source.scala:79)
12      at scala.io.Source$.fromFile(Source.scala:57)

```

```

8   at zio.ZIOCompanionPlatformSpecific.$anonfun$readFile$3(
9     ZIOPlatformSpecific.scala:134)
10  at zio.ZIOCompanionVersionSpecific.$anonfun$attempt$1(
11    ZIOCompanionVersionSpecific.scala:100)
12  at <empty>.ReadFileExample.run(ReadFileExample.scala:5)
13  at <empty>.ReadFileExample.run(ReadFileExample.scala:6)
14  at <empty>.ReadFileExample.run(ReadFileExample.scala:7)"
15  location=<empty>.ReadFileExample.run file=ReadFileExample.
16    scala line=6

```

The cause field in the log entry provides a stack trace, offering additional context about the error for easier diagnosis and troubleshooting. However, this log format can be difficult to parse reliably. A simple solution is to format the log entries in a structured format like JSON or XML.

## 34.2 Writing Custom Loggers

To be able to log in another format, you need to know how to create custom loggers in ZIO. ZIO maintains a set of default loggers that are used to log messages. By default, ZIO contains a single logger that logs messages to the console. To add and remove loggers, you can use the `Runtime.addLogger` and `Runtime.removeDefaultLoggers` operators:

```

1 object Runtime {
2   def addLogger(logger: ZLogger[String, Any]): ZLayer[Any,
3     Nothing, Unit] = ???
4   val removeDefaultLoggers: ZLayer[Any,
5     Nothing, Unit] = ???
6 }

```

A logger of type `ZLogger[Message, Output]` is a logger that converts a message of type `Message` to an output of type `Output` and may perform side effects, such as writing to the console, a file, or a socket. The `ZLogger` trait is defined as follows:

```

1 trait ZLogger[-Message, +Output] {
2   def apply(
3     trace: Trace,
4     fiberId: FiberId,
5     logLevel: LogLevel,
6     message: () => Message,
7     cause: Cause[Any],
8     context: FiberRefs,
9     spans: List[LogSpan],
10    annotations: Map[String, String]
11  ): Output
12 }

```

Here is an example program that demonstrates changing the default logger to a custom logger of our choice:

```

1 import zio._
2 import zio.json._

3
4 object JsonLogger {
5   def apply(): ZLogger[String, String] = (
6     trace: Trace,
7     fiberId: FiberId,
8     logLevel: LogLevel,
9     message: () => String,
10    cause: Cause[Any],
11    context: FiberRefs,
12    spans: List[LogSpan],
13    annotations: Map[String, String]
14  ) => {
15   Map(
16     "timestamp" -> java.time.Instant.now().toString,
17     "level" -> logLevel.toString,
18     "thread" -> fiberId.threadName,
19     "message" -> message()
20   ).toJsonPretty
21 }
22 }
```

In this example, we used the ZIO JSON<sup>1</sup> library to convert the log entry to a JSON string. We only included the base fields in the log entry; the remaining fields are left as an exercise for the reader. To use this custom logger, we need to add it to the runtime as shown below:

```

1 import zio._

2
3 object CustomLoggerExample extends ZIOAppDefault {
4   override val bootstrap =
5     Runtime.removeDefaultLoggers >>>
6     Runtime.addLogger(JsonLogger().map(println(_)))
7
8   def run =
9     ZIO.readFile("file1.txt")
10       .tapErrorCause(cause => ZIO.logErrorCause("Failed to read
11         file", cause))
12       .orElse(
13         ZIO.readFile("file2.txt")
14       )
15 }
```

---

<sup>1</sup><https://zio.dev/zio-json>

Now let's run the program and see the output.

```

1  {
2    "location" : "<empty>.CustomLoggerExample.run",
3    "timestamp" : "2024-12-31T19:05:07.784841660Z",
4    "line" : "12",
5    "thread" : "zio-fiber-802963435",
6    "cause" : "Exception in thread \"zio-fiber-802963435\" java.io.
7      FileNotFoundException: file1.txt (No such file or directory)\\
8      n\tat java.base/java.io.FileInputStream.open0(Native Method)\\
9      n\tat java.base/java.io.FileInputStream.open(FileInputStream.
10     java:219)\\n\tat java.base/java.io.FileInputStream.<init>(
11     FileInputStream.java:159)\\n\tat scala.io.Source$.fromFile(
12     Source.scala:94)\\n\tat scala.io.Source$.fromFile(Source.scala
13     :79)\\n\tat scala.io.Source$.fromFile(Source.scala:57)\\n\tat
14     zio.ZIOCompanionPlatformSpecific.$anonfun$readFile$3(
15     ZIOPPlatformSpecific.scala:134)\\n\tat zio.
16     ZIOCompanionVersionSpecific.$anonfun$attempt$1(
17     ZIOCompanionVersionSpecific.scala:100)\\n\tat <empty>.
18     CustomLoggerExample.run(CustomLoggerExample.scala:11)\\n\tat <
19     empty>.CustomLoggerExample.run(CustomLoggerExample.scala:12)\\
20     n\tat <empty>.CustomLoggerExample.run(CustomLoggerExample.
21     scala:13)",
22    "level" : "ERROR",
23    "message" : "Failed to read file",
24    "file" : "CustomLoggerExample.scala"
25  }

```

Three important things to note here:

First, ZIO maintains a set of loggers, so when you add a new logger, it will be added to the current set of loggers. You can have multiple loggers in your application, for example, one for logging to the console and another for logging to a file. If you only want to replace the default logger with a new one, you can use the `Runtime.removeDefaultLoggers` layer along with the `Runtime.addLogger` layer.

Second, the JSON logger we have created does not have any side effects! It only converts the log entry to a JSON string. To actually log the message to a specific target, we have to use the `ZLogger#map` operator with a function that performs the side effect. In this example, we mapped the JSON logger to the `println` function, which will print the JSON string to the console.

Third, the JSON debugger we have written here is just for educational purposes. The ZIO Logging<sup>2</sup> project provides a rich set of logger implementations, including JSON logger, which you can use out of the box.

As we have seen, ZIO has all the underlying facilities to support logging in a structured

---

<sup>2</sup><https://zio.dev/zio-logging>

format. In structured logging, each log entry is treated as an event object containing key-value pairs rather than a simple text string. These events typically include standard fields like timestamp, log level, and message, along with contextual information, log annotations such as user ID, request ID, logging spans, and any other metadata that can help in diagnosing issues.

### 34.3 Log Annotations

Log annotations are key-value pairs that provide additional context to log entries. They are used to correlate log entries with contextual information, such as user ID, request ID, session ID, and any other metadata that can help us understand and track the log entries better.

Let's start with a simple example:

```

1 import zio.http._

2 Method.GET -> handler { request: Request =>
3   processRequest(request)
4 }
5 }
```

This is a simple example of defining a route with ZIO HTTP<sup>3</sup> which processes incoming requests. Inside `processRequest`, we have complex logic that involves multiple operations, such as reading from a database, calling external services, and processing request data. When the request flows through these operations, we may have log entries that are suitable for diagnostic purposes later. As a result, we need log entries to be correlated with the request that triggered them. This correlation allows you to trace the entire journey of the request through the system and helps tremendously with debugging and performance analysis. For example, you can filter log entries based on the request ID by adding a unique request ID to each log entry as shown below:

```

1 def processRequest(request: Request) =
2   for {
3     userId <- ZIO.succeed(extractUserId(request))
4
5     readHistory <- ReadingHistoryService.getUserHistory(userId)
6       <- ZIO.logInfo(s"Fetched ${readHistory.length}")
7     reading history for user $userId"
8
9     queryHistory <- QueryHistoryService.getUserHistory(userId)
10      <- ZIO.logInfo(s"Fetched ${queryHistory.length}")
11     query history for user $userId"
12
13     recommendations <- RecommendationEngine.recommend(readHistory,
14       , queryHistory)
```

---

<sup>3</sup><https://zio.dev/zio-http>

```

12     _ <- ZIO.logInfo(s"Generated ${recommendations.
13       length} recommendations for user $userId")
14
15   response <- ZIO.succeed(
16     Response
17       .json(recommendations.toJson)
18       .addHeader(Header.ContentType(MediaType.application.json))
19     ),
20   )
21 } yield response

```

But this approach has a few drawbacks:

First, we have to manually add the user ID to each log entry, which is repetitive and cumbersome.

Second, if any of the services we are calling logs a message and we want to correlate those log entries with the request, we have to pass the user ID to those service calls as well.

Third, adding such information to the log messages makes them harder to organize and filter when analyzing logs.

To address these issues, ZIO Logging maintains a set of log annotations using `FiberRef` that allows you to add contextual information to log entries:

```

1 Method.GET -> handler { request: Request =>
2   ZIO.logAnnotate("user-id", extractUserId(request)) {
3     processRequest(request)
4   }
5 }

```

Or you can use the annotated ZIO aspect to add log annotations to the entire ZIO effect:

```

1 Method.GET -> handler { request: Request =>
2   processRequest(request) @@ ZIOAspect.annotated("user-id",
3     extractUserId(request))
4 }

```

Now, instead of manually adding the user ID to each log entry, ZIO will automatically add the user ID to all log entries within the scope of the `ZIO.logAnnotate` operator.

Now when a request comes in, the log entry will include the user ID as an annotation. For example, with the default logger, the log entry will look like this, which includes the user ID field:

```

1 timestamp=2025-01-02T17:23:51.410341805Z level=INFO thread=#zio-
2   fiber-1144938168 message="Fetched 5 reading history" location
3    =<empty>.RecommendationService.processRequest file=
4       Recommendation.scala line=84 user-id=37411ee1-364b-40a0-8386-
5         f714cddb40da

```

Please note that, the current log annotations are maintained using the `FiberRef` which means that they are not only for the current fiber, but also they are automatically propagated across asynchronous child fibers. This means that the log annotations will be maintained even if the execution flow crosses different fibers as they are passed to the child fibers. So in the previous example, if we change the `processRequest` into a version which calls `ReadingHistoryService` and `QueryHistoryService` in parallel, the log entries inside those services will also include the user ID as an annotation.

When exiting the scope of the `ZIO.logAnnotate` operator, the log annotations will be removed from the annotations associated to the current fiber.

The value of log annotations is when your endpoint is called concurrently by multiple users. Without log annotations, it would be challenging to correlate log entries with the requests (users) that triggered them.

The scope of log annotations in `ZIO.logAnnotate` is bound to its containing ZIO effect. For finer-grained control over annotation scope, use `ZIO.logAnnotateScoped`, which adds log annotations to the current Scope of the ZIO effect.

```

1 Method.GET -> handler { request: Request =>
2   ZIO.scoped {
3     for {
4       _ <- ZIO.logAnnotateScoped("user-id", extractUserId(request))
5     }
6     response <- processRequest(request)
7     _ <- ZIO.logInfo("Request processed")
8   } yield response
9 }
```

In this example, the log annotations added using `ZIO.logAnnotateScoped` will only be available within the scope of the `ZIO.scoped` operator. This is useful when you want to have more granular control over the scope of log annotations using type-level `Scope` data type.

Until now, we have learned how to update log annotations of a local scope. But what if we want to update the log annotations of the entire application? Assume we have to scale the recommendation service horizontally, and we want to distinguish the log entries of each instance when analyzing the logs. To modify log annotations globally, we have to use `bootstrap` layer to update the log annotations. This will change the log annotations before the application starts. This way, all log entries including internal log entries that come from ZIO will include the instance ID as an annotation:

```

1 import zio._
2 import zio.http._
3
4 object RecommendationService extends ZIOAppDefault {
5   override val bootstrap =
6     ZLayer.scoped {
```

```

7   for {
8     uuid      <- Random.randomUUID
9     instanceId <- System.envOrElse("INSTANCE_ID", uuid.
10    toString)
11    _           <- FiberRef.currentLogAnnotations.
12    locallyScopedWith(_ + ("instance-id" -> instanceId))
13    } yield ()
14  }
15
16 private val app: Routes[Any, Response] = ???
17
18 override val run = Server.serve(app).provideLayer(???
19 }
```

In this example, we used `FiberRef.currentLogAnnotations` to access the current log annotations of the fiber then using `FiberRef#locallyScopedWith` operator to update the log annotations of the current fiber, which will be available for the entire application.

## 34.4 Log Spans

Similar to log annotations, you can also label log entries with span labels to group them together and calculate the duration from the start of the span to the point where the log entry is made:

```

1 Method.GET -> handler { request: Request =>
2   ZIO.logSpan("recommendation") {
3     ZIO.logAnnotate("user-id", extractUserId(request)) {
4       processRequest(request)
5     }
6   }
7 }
```

Or we can do the same with the `spanned` ZIO aspect:

```

1 Method.GET -> handler { request: Request =>
2   ZIO.logSpan("recommendation") {
3     processRequest(request) @@ ZIOAspect.annotated("user-id",
4                   extractUserId(request))
5   }
6 }
```

It will create a new span with the label `recommendation`, every log entry within the scope of this span will be tagged with this label with the duration of as the time taken from the start of the span to the point where the log entry is made inside the span:

```

1 timestamp=2025-01-02T18:18:26.746803327Z level=INFO thread=#zio-
  fiber-1863607651 message="Fetched 10 reading history"
```

```

recommendation=2ms location=<empty>.RecommendationService.
processRequest file=Recommendation.scala line=84 user-id=
dc8f00b5-176b-4bf5-b294-494556de1659
2 timestamp=2025-01-02T18:18:26.747848879Z level=INFO thread=#zio-
fiber-1863607651 message="Fetched 16 queries" recommendation=
119ms location=<empty>.RecommendationService.processRequest
file=Recommendation.scala line=87 user-id=dc8f00b5-176b-4bf5-
b294-494556de1659
3 timestamp=2025-01-02T18:18:27.762932174Z level=INFO thread=#zio-
fiber-1863607651 message="Generated 3 recommendations"
recommendation=213ms location=<empty>.RecommendationService.
processRequest file=Recommendation.scala line=91 user-id=
dc8f00b5-176b-4bf5-b294-494556de1659

```

In the example above, we have a span with the label `recommendation` that includes three log entries. Each log entry includes the `recommendation` label and the duration from the start of the span to the point where the log entry is made.

Like the log annotations, the current log spans are maintained using `FiberRef`, so they are automatically propagated across asynchronous operations. This means that the span information will be maintained even if the execution flow crosses different fibers as they are passed to the child fibers.

Also like the log annotations, it has the scoped variant `ZIO.logSpanScoped` which allows you to add spans to the current `Scope` of the `ZIO` effect and manage the scope of log spans using type-level `Scope` data type.

## 34.5 Integrating with Existing Logging Frameworks

The built-in logging system is simple yet powerful enough for many use cases such as small to medium-sized applications like command-line tools, as it has zero dependency on any logging library.

However, when your application grows, you may want to integrate your `ZIO` application with existing logging frameworks like `SLF4J` and `Log4j`. `ZIO` provides a pluggable backend system that allows you to integrate with various logging frameworks seamlessly. The logging front-end in `ZIO` is designed to be independent of the underlying logging framework, which means you can switch between different logging frameworks without changing any line of your application code.

Assume you are using a third-party library that logs messages using `SLF4J`. If we don't integrate `ZIO` logging with `SLF4J`, you'll end up with two separate logging systems in your application. This can lead to several problems:

- Inconsistent log formats and destinations - Your `ZIO` logs might go to one place with one format, while `SLF4J` logs go elsewhere with a different format
- Difficulty in correlating logs - Without unified logging, it becomes harder to trace the flow of operations across your application and third-party libraries

- Separate configuration needed for each logging system - You have to manage two separate configurations, one for ZIO logging and another for SLF4J
- Different log levels and filtering mechanisms that need to be managed separately

Here is an example of such a scenario:

```

1 import zio._
2
3 object MainApp extends ZIOAppDefault {
4   def run =
5     ZIO.logSpan("my-span") {
6       ZIO.logAnnotate("user_id", "john_123") {
7         for {
8           _ <- ZIO.logInfo("App started!")
9           _ <- ZIO.attempt(my.lib.foo.doSomething())
10          _ <- ZIO.logInfo("App finished!")
11        } yield ()
12      }
13    }
14 }
```

In this example, we have a simple ZIO application that logs messages using ZIO logging. The `doSomething` function is imported from a third-party library by importing the `lib.foo` package. If we run this ZIO application, we might see something like this:

```

1 timestamp=2025-01-06T15:20:21.501984998Z level=INFO thread=#zio-
  fiber-898693704 message="App started!" my-span=8ms location=<
  empty>.MainApp.run file=Main.scala line=8 user_id=john_123
2 SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
3 SLF4J: Defaulting to no-operation (NOP) logger implementation
4 SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for
  further details.
5 timestamp=2025-01-06T15:20:21.519240169Z level=INFO thread=#zio-
  fiber-898693704 message="App finished!" my-span=22ms location
  =<empty>.MainApp.run file=Main.scala line=10 user_id=john_123
```

In this example, the ZIO logs are printed to the console, but the third-party library logs are not printed because the `foo` library uses the SLF4J API for logging, and we haven't provided an SLF4J provider in the classpath.

SLF4J has many providers. Let's put the SimpleLogger provider in the classpath:

```

1 libraryDependencies += "org.slf4j" % "slf4j-simple" % <
  SLF4J_VERSION>
```

Now, if we run the application, we will see the following output:

```

1 timestamp=2025-01-04T10:35:33.494689958Z level=INFO thread=#zio-
  fiber-420233556 message="App started!" location=<empty>.
  MainApp.run file=SLF4JLoggerExample.scala line=7
```

```

2 [ZScheduler-Worker-4] INFO SLF4J-LOGGER - Doing something!
3 timestamp=2025-01-04T10:35:33.563527029Z level=INFO thread=#zio-
   fiber-420233556 message="App finished!" location=<empty>.
   MainApp.run file=SLF4JLoggerExample.scala line=9

```

As you can see, the output is now mixed with ZIO logs and SLF4J logs with different formats; that is the problem we mentioned earlier. We do not want to have two different logging frameworks, each with different configurations and logging formats. To solve this problem, we have two options:

1. **Forwarding ZIO Logs to SLF4J** - This approach means routing all ZIO logging through the SLF4J API, making SLF4J the primary logging system. In this setup, all log entries (both from ZIO and third-party libraries) flow through SLF4J. You can use any SLF4J provider (Logback, Log4j, etc.) as your final logging implementation.
2. **Forwarding SLF4J Logs to ZIO** - This approach routes all SLF4J logging through the ZIO's logging system, making ZIO the primary logging system. In this setup, all log entries (both from ZIO and third-party libraries using SLF4J) flow through ZIO's logging system. You maintain full control over logs using logging capabilities of ZIO.

The choice between these approaches often depends on your specific needs. For example, if you are already using SLF4J in your other services, you might want to make SLF4J the primary logging system, so when developing new services with ZIO, you tend not to change the logging system. But the downside of this approach is that you lose some of the powerful features of ZIO logging. For example, you will lose contextual data, such as log annotations and spans, when importing third-party libraries that use SLF4J in your ZIO application.

On the other hand, if you are building a new application or set of services from scratch, you might want to use ZIO logging as the primary logging system, so you might decide to forward SLF4J logs to ZIO. This way, you can take advantage of all the powerful features of ZIO logging, such as log annotations and spans, even when importing third-party libraries that use SLF4J in your ZIO application.

### 34.5.1 Forwarding ZIO Logs to SLF4J

In this approach, we will forward ZIO log entries to the SLF4J API. This means all log entries from ZIO and third-party libraries will be forwarded to the SLF4J API, and then SLF4J will handle them:

```

1 for {
2   _ <- ZIO.logInfo("App started!") // ZIO Logging -> SLF4J API
   -> SLF4J Providers (Logback, Log4j, etc.)
3   _ <- ZIO.attempt(my.lib.foo.doSomething()) // SLF4J API ->
   -> SLF4J Providers (Logback, Log4j, etc.)
4   _ <- ZIO.logInfo("App finished!") // ZIO Logging -> SLF4J API
   -> SLF4J Providers (Logback, Log4j, etc.)
5 } yield ()

```

To do this we need to write a custom ZIO logger (ZLogger) that forwards ZIO log entries to the SLF4J API. Fortunately, the ZIO Logging<sup>4</sup> project has already implemented this for us.

First, add the following dependencies to your build.sbt file:

```

1 libraryDependencies += "dev.zio" %% "zio-logging"      % <
2   ZIO_LOGGING_VERSION>,
3 libraryDependencies += "dev.zio" %% "zio-logging-slf4j" % <
4   ZIO_LOGGING_VERSION>,
5 libraryDependencies += "org.slf4j" % "slf4j-simple"      % <
6   SLF4J_VERSION>
```

The “zio-logging-slf4j” module contains the custom ZIO logger to forward ZIO logs to the SLF4J API. The “slf4j-simple” module is a simple SLF4J binding that logs messages to the console.

Add the SLF4J.slf4j layer to append a logger that forwards log entries to SLF4J:

```

1 import zio._
2 import zio.logging.backend.SLF4J
3
4 object MainApp extends ZIOAppDefault {
5   override val bootstrap =
6     Runtime.removeDefaultLoggers ++ SLF4J.slf4j
7
8   override def run =
9     ZIO.logSpan("my-span") {
10       ZIO.logAnnotate("user_id", "john_123") {
11         for {
12           - <- ZIO.logInfo("App started!")
13           - <- ZIO.attempt(my.lib.foo.doSomething())
14           - <- ZIO.logInfo("App finished!")
15         } yield ()
16       }
17     }
18 }
```

Now if you run the application, you will see all log entries are logged through the SimpleLogger provided by “slf4j-simple”:

```

1 [ZScheduler-Worker-9] INFO <empty>.MainApp - App started!
2 [ZScheduler-Worker-9] INFO my.lib.foo - Doing something!
3 [ZScheduler-Worker-9] INFO <empty>.MainApp - App finished!
```

You probably noticed that the log entries printed to the console have less information than the ZIO log entries. This is the default log format for the SLF4J logger.

---

<sup>4</sup><https://zio.dev/zio-logging>

To customize the logging format, there are two basic data types: `LogFormat` and `LogAppender`. `LogFormat` is a DSL for describing format of a log message, and `LogAppender` is an interface for appending log messages to logging backends. Let's try to add more information to the log format:

```

1 import zio.logging._
2 import zio.logging.LogFormat.label
3 import zio.logging.backend.SLF4J
4
5 val location: LogFormat =
6   LogFormat.make { (builder, trace, _, _, _, _, _, _, _) =>
7     trace match {
8       case Trace(location, file, line) =>
9         builder.appendKeyValue("location", location)
10        builder.appendKeyValue("file", file)
11        builder.appendKeyValue("key", line.toString)
12        case _ => ()
13     }
14   }
15
16 private val slf4jLogger =
17   SLF4J.slf4j(format =
18     LogFormat.allAnnotations +
19     LogFormat.spans +
20     label("cause", LogFormat.cause).filter(LogFilter.
21       causeNonEmpty) +
22     label("fiber_id", LogFormat.fiberId) |-|
23     label("message", LogFormat.line) |-|
24     location
25   )

```

Now we can use `slf4jLogger` layer in the bootstrap layer:

```

1 object MainApp extends ZIOAppDefault {
2   override val bootstrap =
3     Runtime.removeDefaultLoggers ++ slf4jLogger
4
5   override def run = ???
6 }

```

To ensure that the log entries are printed with relevant timestamps, add the following configuration to the `src/main/resources/simplelogger.properties` file:

```

1 org.slf4j.simpleLogger.showDateTime=true
2 org.slf4j.simpleLogger.dateTimeFormat=yyyy-MM-dd HH:mm:ss:SSS

```

If we run the application with this custom logger, we will see the following output:

```

1 2025-01-06 19:20:56:840 [ZScheduler-Worker-6] INFO <empty>.
   MainApp - user_id=john_123 my-span=12ms location=<empty>.
   MainApp.run file=MainApp.scala line=42 fiber_id=zio-fiber
   -478802423 message=App started!
2 2025-01-06 19:20:56:843 [ZScheduler-Worker-6] INFO my.lib.foo -
   Doing something!
3 2025-01-06 19:20:56:844 [ZScheduler-Worker-6] INFO <empty>.
   MainApp - user_id=john_123 my-span=21ms location=<empty>.
   MainApp.run file=MainApp.scala line=44 fiber_id=zio-fiber
   -478802423 message=App finished!

```

That's it! We successfully included custom fields in log entries forwarded to the SLF4J API. However, this contextual data is only available for the log entries that are forwarded to the SLF4J API and won't be propagated to entries logged through third-party libraries. As you see, the second line of output lacks our added context.

Please note that you can do the same with other providers such as Logback and Log4j by adding the corresponding SLF4J provider to the classpath.

### 34.5.2 Forwarding SLF4J Logs to ZIO

Writing a bridge that forwards SLF4J log entries to ZIO is the opposite of the previous approach. This means all log entries from SLF4J and third-party libraries that use the SLF4J API will be forwarded to the ZIO logging system. This ensures that all logs are captured and handled by the ZIO logging system:

```

1 for {
2   _ <- ZIO.logInfo("App started!") // ZIO Logging
3   _ <- ZIO.attempt(my.lib.foo.doSomething()) // SLF4J API -> ZIO
   Logging
4   _ <- ZIO.logInfo("App finished!")// ZIO Logging
5 } yield ()

```

This can be achieved by writing a custom service provider that implements the `org.slf4j.spi.SLF4JServiceProvider` interface. When SLF4J tries to load the logger implementation using the `org.slf4j.LoggerFactory.getLogger` API, it will use our custom service provider. This custom service provider is responsible for forwarding the log entries coming from the SLF4J API to the ZIO logging system.

Fortunately, we do not have to write this from scratch. ZIO Logging<sup>5</sup> provides an SLF4J bridge out of the box.

Let's see how to use it:

First, we need to add one of the following dependencies to our `build.sbt` file based on the SLF4J version we want to forward logs from:

---

<sup>5</sup><https://zio.dev/zio-logging>

```

1 libraryDependencies += "dev.zio" %% "zio-logging-slf4j-bridge" %
2   <ZIO_LOGGING_VERSION>
2 libraryDependencies += "dev.zio" %% "zio-logging-slf4j2-bridge" %
3   <ZIO_LOGGING_VERSION>

```

Then, we have to add the `Slf4jBridge.init` layer to create a logger that forwards log entries to SLF4J:

```

1 import zio._
2 import zio.logging.slf4j.bridge.Slf4jBridge
3
4 object MainApp extends ZIOAppDefault {
5   override val bootstrap = Slf4jBridge.init()
6
7   override def run =
8     ZIO.logSpan("my-span") {
9       ZIO.logAnnotate("user_id", "john_123") {
10         for {
11           _ <- ZIO.logInfo("App started!")
12           _ <- ZIO.attempt(my.lib.foo.doSomething())
13           _ <- ZIO.logInfo("App finished!")
14         } yield ()
15       }
16     }
17 }

```

If we run this application, we will see that all log entries are forwarded to the ZIO logging system:

```

1 timestamp=2025-01-07T08:36:22.586030545Z level=INFO thread=#zio-
  fiber-9778569 message="App started!" my-span=4ms location=<
  empty>.MainApp.run file=SLF4JLoggerExample.scala line=11
  user_id=john_123
2 timestamp=2025-01-07T08:36:22.604818554Z level=INFO thread=#zio-
  fiber-1819028394 message="Doing something!" lib.foo=0ms my-
  span=22ms user_id=john_123 logger_name=lib.foo
3 timestamp=2025-01-07T08:36:22.605771456Z level=INFO thread=#zio-
  fiber-9778569 message="App finished!" my-span=23ms location=<
  empty>.MainApp.run file=SLF4JLoggerExample.scala line=13
  user_id=john_123

```

This solution enables capturing logs from non-ZIO libraries using SLF4J, while preserving all contextual data - including log annotations and spans - as they flow into the ZIO logging system. For example, both ZIO and third-party library logs will maintain consistent formatting and include contextual fields like `user_id` and `my-span` from annotations and spans.

As a final note, the integration between ZIO and logging libraries requires careful consideration. You must choose exclusively between `zio-logging-[slf4j]` or `zio-logging-[slf4j]-bridge` - attempting to use both simultaneously will create a circular dependency in the logging chain, resulting in stack overflow errors. Each library serves a distinct purpose:

- `zio-logging-slf4j` sends ZIO logs to SLF4J
- `zio-logging-slf4j-bridge` routes SLF4J logs into ZIO's logging system

Select the appropriate library based on your application's logging flow requirements.

## 34.6 Logging Errors while Ignoring Them

When dealing with effects that may fail, we sometimes want to continue program execution while maintaining visibility into any failures through logging. An example is metrics reporting during request handling - if the reporting operation fails, we want to log the error but still process the request successfully. For these scenarios, `ZIO#ignoreLogged` provides an elegant solution:

```

1 Method.GET -> handler { request: Request =>
2   for {
3     response <- processRequest(request)
4     metrics  <- calculateMetrics
5     // Fire-and-forget metrics reporting
6     _ <- reportMetrics(metrics).ignoreLogged.fork
7   } yield response
8 }
```

This operator logs any failures at the DEBUG level, capturing both the error message and root cause, while allowing the effect to proceed without failing.

## 34.7 Conclusion

ZIO offers a powerful yet flexible solution for application logging needs. Through its core features—structured logging, log annotations, and log spans—it provides deep insights into application behavior while maintaining type safety and ease of use. Its use of `FiberRef` ensures proper context propagation across asynchronous operations, making it particularly well-suited for concurrent applications.

For teams adopting ZIO, the logging framework provides a natural path from simple console logging to more sophisticated setups involving structured logging and integration with external log aggregation systems.

The pluggable backend system allows seamless integration with existing logging frameworks like SLF4J, giving teams the freedom to choose the most suitable logging infrastructure for their needs. Whether starting a new project or integrating with existing systems,

ZIO Logging provides the tools necessary for effective system observability, from basic debugging to complex production monitoring requirements.

# Chapter 35

## Observability: Metrics

Modern applications need to be observable. We need to understand what's happening inside them during runtime. Metrics are quantifiable measurements that give us insights into our application's behavior and performance. Whether you want to track the number of incoming requests, measure response times, or monitor resource usage, metrics provide the data you need to understand your system's health and performance.

ZIO offers a comprehensive metrics system that makes it easy to instrument your code and operations with type-safe metrics. The system integrates seamlessly with popular monitoring tools like Prometheus and StatsD, while providing powerful features like metric tagging and various metric types (counters, gauges, histograms, summaries, and frequencies).

In this chapter, we'll explore how to use ZIO's metrics capabilities effectively. Through practical examples, you'll learn how to instrument your applications to gather the insights you need for monitoring, debugging, and optimization.

Let's start by learning the basics.

### 35.1 Getting Started

Let's start by creating a simple counter metric, and then see how we can use it to measure how many times an effect has been executed:

```
1 import zio.metrics._  
2  
3 val counter = Metric.counter("total_execution").fromConst(1L)
```

In this example, we create a counter metric with the name `total_execution`. We can then use this counter to measure how many times an effect has been executed (we will explain the `fromConst` method later).

To apply the counter metric to an effect, we can use its `apply` method:

```
1 counter {
2   processRequest(request)
3 }
```

All metrics are `ZIOAspects` that can be applied to effects using the `@@` operator:

```
1 processRequest(request) @@ counter
```

After capturing metrics, ZIO automatically tracks the number of times an effect has been executed. To check the value of a metric, use the `Metric#value` method:

```
1 Metric.counter("total_execution").value.map(_.count)
```

Let's combine all of this into an example:

```
1 import zio._
2 import zio.metrics._

3
4 object MainApp extends ZIOAppDefault {
5   val effect = ZIO.debug("Effect executed!")
6
7   val mainApp =
8     Metric
9       .counter("total_execution")
10      .fromConst(1L)(effect)
11      .repeat(Schedule.exponential(100.milliseconds))
12
13   val monitorApp =
14     Metric
15       .counter("total_execution")
16       .fromConst(1L)
17       .value.map(_.count)
18       .debug("Counter Value: ")
19       .repeat(Schedule.fixed(1.second))
20
21   def run = mainApp <&> monitorApp
22 }
```

This example tracks the number of times the `effect` is executed. The `mainApp` effect repeats the `effect` with an exponential schedule, while concurrently the `monitorApp` retrieves the value of the `total_execution` counter and prints it every second.

Each metric we create has a unique key associated with it. The key is based on a combination of the metric type, name, description, and tags. This ensures that different metrics cannot have conflicting keys. The following code snippet shows how different metrics can be differentiated from each other:

```

1 for {
2   -<- ZIO.unit
3   -<- Metric.counter("foo")(ZIO.succeed(1L))
4   -<- Metric.counter("foo", "the foo key")(ZIO.succeed(2L))
5   -<- Metric.counter("foo").tagged("a", "b")(ZIO.succeed(3L))
6   -<- Metric.counter("foo").value.debug("first")
7   -<- Metric.counter("foo", "the foo key").value.debug("second")
8   -<- Metric.counter("foo").tagged("a", "b").value.debug("third")
9   -<- ZIO.debug("-----")
10  -<- ZIO.metrics.flatMap(_.prettyPrint).debug
11 } yield ()

```

This example demonstrates how `MetricKey` differentiates between metrics. The first metric has only a name, the second adds a description, and the third includes both a name and tag.

The `ZIO.metrics` operator provides a convenient way to capture a snapshot of all metrics during development, offering a `prettyPrint` method for human-readable output. For production environments, we recommend using the built-in `MetricsClient` from ZIO Metrics Connectors, which we'll cover later in this chapter.

## 35.2 Metrics Types

ZIO supports five metric types: Counter, Gauge, Histogram, Summary, and Frequency. Each has unique characteristics and use cases:

- **Counter:** A cumulative metric that only increases. Perfect for tracking total occurrences like request counts or completed tasks.
- **Gauge:** A metric that represents a single numerical value that can arbitrarily go up and down. Ideal for measuring current values like memory usage or active connections.
- **Histogram:** Tracks the distribution and frequency of a metric's values over time. Useful for analyzing data like request latencies or response sizes.
- **Summary:** Similar to histogram but calculates streaming  $\varphi$ -quantiles over time windows. Commonly used for tracking request duration percentiles or service level indicators.
- **Frequency:** Counts occurrences of specific events or values. Excellent for monitoring event distributions like HTTP status codes or error types.

### 35.2.1 Counter

Think of a Counter like a store's door clicker - it only goes up, never down. Every time someone enters, click! The number increases by one. At the end of the day, it shows exactly how many people visited the store.

A Counter's strength lies in its simplicity. It answers straightforward questions like "How

many customers visited today?" or "How many orders did we receive this week?" without complex calculations or rules.

Each counter metric has its own input type, which defines the value type the metric will receive. For example, the input type of `Metric.counter` is `Long`. To apply a counter metric to an effect, the effect must return a `Long` value:

```
1 ZIO.succeed(42L) @@ Metric.counter("total_execution")
```

This increases the counter metric by 42 each time the effect executes. To modify the input type of a metric, you can use combinators like `Metric#contramap` and `Metric#fromConst`. For example, to make the counter increment by 1 each time the effect executes, use the `fromConst` operator:

```
1 val totalRequest =
2   Metric.counter("total_requests", "Total number of requests")
3   .fromConst(1L)
4   .tagged("method", "GET")
5   .tagged("path", "/user/profile")
```

If we provide any other type, the compiler will raise a type error. This ensures that the metric is used with the correct type of values.

### 35.2.2 Gauge

Think of a gauge like your car's gas tank meter - it shows you exactly how much gas you have right now, and that number can go up when you fill up or down as you drive. It's always telling you the "current state" of things.

```
1 val storageUsage =
2   Metric.gauge("storage_usage", "Current storage usage")
```

Computer systems use gauges the same way to track things that go up and down. For instance, if you're running a restaurant delivery app: - The number of delivery drivers currently on the road - How many tables are currently occupied in the restaurant - The number of people waiting in the online queue right now

Unlike a counter (which only goes up, like counting total sales), a gauge is like taking a snapshot of what's happening at this exact moment - just like how your gas gauge tells you exactly how much gas is in your tank right now, not how much gas you've used in total.

### 35.2.3 Histogram

A histogram helps you understand the distribution of your data by breaking it into buckets. The histogram implementation in ZIO is cumulative, which means it keeps track of the total count of items up to a certain bucket.

Let me explain cumulative histograms using a simple pizza delivery example. Imagine you own a pizza shop and want to track delivery times to make customers happy. You decide to

measure how long each delivery takes and group the times into ranges:

- Up to 20 minutes
- Up to 30 minutes
- Up to 45 minutes
- All deliveries

The corresponding histogram metric is:

```

1 val deliveryTimeHistogram =
2   Metric.histogram(
3     name = "delivery_time",
4     boundaries = MetricKeyType.Histogram.Boundaries.fromChunk(
5       Chunk(20, 30, 45))
6     ).contramap[Int](_.toDouble)

```

Now let's say on a busy Friday night, you made 10 deliveries: 1. Mike: 25 minutes 2. Bob: 42 minutes 3. Sarah: 18 minutes 4. David: 29 minutes 5. Lisa: 35 minutes 6. Tommy: 15 minutes 7. John: 38 minutes 8. Alice: 50 minutes 9. Mary: 40 minutes 10. Jenny: 28 minutes

Here is the code to record the delivery times and print the snapshot of the metric registry:

```

1 for {
2   observations <- ZIO.succeed(Chunk(25, 42, 18, 29, 35, 15, 38,
3     50, 40, 28))
4   -      <- ZIO.foreachDiscard(observations)(ZIO.succeed(_)
5     -      00 deliveryTimeHistogram)
6   -      <- ZIO.metrics.flatMap(_.prettyPrint).debug
7 } yield ()

```

The cumulative histogram would show:

- Up to 20 minutes: 2 deliveries (Tommy and Sarah)
- Up to 30 minutes: 5 deliveries (Tommy, Sarah, Mike, Jenny, and David)
- Up to 45 minutes: 9 deliveries (everyone except Alice)
- All deliveries: 10 deliveries
- Minimum delivery time: 15 minutes
- Maximum delivery time: 50 minutes
- Total delivery time: 310 minutes

This analysis helps you answer critical questions about pizza delivery performance, revealing that 50% of pizzas (5 out of 10) are delivered within the 30-minute window and, more importantly, the company is successfully meeting its 45-minute delivery promise, with 90% of deliveries (9 out of 10) being completed within that timeframe.

With `Metric.histogram`, you can create evenly spaced or exponentially growing buckets. The `linear` boundary is useful when you want to create buckets that grow by a fixed amount, while the `exponential` method is useful when you want to create buckets that grow exponentially.

```

1 val linearDeliveryHistogram = Metric.histogram(
2   name = "delivery_time_linear",
3   boundaries = MetricKeyType.Histogram.Boundaries.linear(
4     start = 10.0, // Start at 10 minutes
5     width = 10.0, // 10-minute intervals
6     count = 6     // Creates boundaries at: 10, 20, 30, 40, 50,
7     60 minutes and beyond

```

```

7   )
8 ).contramap[Int](_.toDouble)
9
10 val responseTimeHistogram = Metric.histogram(
11   name = "response_time_exponential",
12   boundaries = MetricKeyType.Histogram.Boundaries.exponential(
13     start = 5.0,    // Start at 5 minutes
14     factor = 2.0,   // Double each time
15     count = 5       // Creates boundaries at: 5, 10, 20, 40, 80
16     minutes and beyond
17   )
18 ).contramap[Int](_.toDouble)

```

The linear method creates boundaries that increase by a constant width, where each subsequent boundary is calculated by adding the width to the previous value, starting from the initial value. In contrast, the exponential method generates boundaries that grow by a multiplicative factor, where each subsequent boundary is calculated by multiplying the previous value by the factor - this creates a sequence where the gaps between boundaries grow larger as the values increase, which is particularly useful for data that spans several orders of magnitude, such as latency measurements or performance metrics.

#### 35.2.4 Summary

Imagine you run a pizza delivery service and have the following delivery times: 18, 23, 14, 41, 24, 15, 26, 21, 30, and 22 minutes. You want to know how long it typically takes for a pizza to arrive for 50% of your customers and how long it takes for 90% of your customers. This is where a summary metric comes in handy. You'll need to create a summary metric that tracks the 50th and 90th percentiles of the delivery times:

```

1 val deliveryTimeSummary = Metric.summary(
2   name = "delivery_time",
3   maxAge = 1.minutes,
4   maxSize = 100,
5   error = 0.5,
6   quantiles = Chunk(0.5, 0.9)
7 ).contramap[Int](_.toDouble)

```

The `maxAge` parameter specifies the maximum age of the samples in the summary, while the `maxSize` parameter sets the maximum number of samples that can be stored. The `error` parameter controls the error allowed in the quantile estimation, and the `quantiles` parameter specifies the quantiles to track. In this example, we are tracking the 50th and 90th percentiles of the delivery times.

Let's apply the summary metric to the delivery times and print the snapshot of the metric registry:

```

1 for {

```

```

2   observations <- ZIO.succeed(Chunk(18, 23, 14, 41, 24, 15, 26,
3     21, 30, 22))
4   -      <- ZIO.foreach(observations)(ZIO.succeed(_) @@
5     -      deliveryTimeSummary)
6     -      <- ZIO.metrics.flatMap(_.prettyPrint).debug
7   } yield ()

```

Looking at the key delivery time metrics, we can see that the median (50th percentile) is 22 minutes, which means half of all deliveries reach customers in less than this time, effectively representing your typical delivery performance. Looking at the 90th percentile, which stands at 30 minutes, this indicates that 90% of your pizzas are delivered within this timeframe, though it's important to note that the remaining 10% of deliveries take longer than 30 minutes, representing your slower delivery cases.

These metrics together give us a comprehensive picture of your delivery service's performance and help identify both your standard service level and potential areas for improvement in those longer delivery times.

### 35.2.5 Frequency

The text provides a clear example of frequency metrics using HTTP response codes, and the bullet points are well-structured.

Let me explain the frequency metric using a real-world web application example. Think of a frequency metric as tracking HTTP response codes for your web API. Instead of just knowing you handled 10,000 requests today, it shows you exactly what types of responses occurred:

- 200 (Success): 8,500 responses
- 400 (Bad Request): 800 responses
- 401 (Unauthorized): 400 responses
- 404 (Not Found): 200 responses
- 500 (Server Error): 100 responses

Defining a frequency metric is simple:

```

1 val responseFrequency =
2   Metric.frequency("response_code", "HTTP response codes")

```

The output of the frequency metric is a map with string keys and long values, where each key represents a specific response code and the value represents the number of occurrences. It's particularly useful for alerting – for example, you might want to be notified if the frequency of 500 errors exceeds 1% of total requests.

## 35.3 Writing Metric Client

Production systems typically need to track multiple metrics simultaneously. To efficiently manage this complexity, ZIO provides the `ConcurrentMetricRegistry` class – a thread-safe registry that maintains all metrics and their values in a concurrent hash map. Values can be accessed through a snapshot API, and the registry supports listener registration for metric update notifications. Note that since this class is private to the `zio` package, it cannot be accessed directly:

```

1 private[zio] class ConcurrentMetricRegistry {
2   private val listenersRef: AtomicReference[Array[MetricListener]] = ???
3   private val map: ConcurrentHashMap[MetricKey[MetricKeyType], MetricHook.Root] = ???
4
5   // Poll-based API
6   def snapshot(): Set[MetricPair] = ???
7   def get(key: MetricKey[Type]): MetricHook = ???
8
9   // Push-based API
10  final def addListener(listener: MetricListener): Unit = ???
11  final def removeListener(listener: MetricListener): Unit = ???
12  private[zio] def notifyListeners[T](
13    key: MetricKey[MetricKeyType.WithIn[T]],
14    value: T,
15    eventType: MetricEventType
16  ): Unit = ???
17 }

```

It has two sets of methods: 1. **Poll-based API** - These methods are used to get a snapshot of all metrics or retrieve a specific metric by its key. They are useful when you want to periodically check metrics. 2. **Push-based API** - These methods are used to add or remove listeners that will be notified when a metric is updated. This is useful when you want real-time metric updates.

The `ConcurrentHashMap` has two internal states: a concurrent hash map to store the metrics and their values, and an atomic reference to store the listeners.

Now that we have learned how metrics work internally, let's see how we can use them in real-world applications. In production environments, we typically want to track metrics throughout an application's lifecycle by continuously collecting them and storing them in a time-series database, then visualizing them using a dashboard.

Using ZIO Metrics Connectors<sup>1</sup>, we can export ZIO metrics to popular collectors like Prometheus and StatsD. Let's explore the integration — one for pull-based monitoring and the other for push-based monitoring. Other setups follow a similar pattern.

For integration with pull-based monitoring tools like Prometheus, we need to: 1. Periodically capture application metrics from the metric registry 2. Encode the metrics into a format that the collector understands (in this case, the plaintext Prometheus format) 3. Create an HTTP endpoint in our application to expose the metrics, as pull-based monitoring systems require servers to periodically scrape metrics from exporters (targets)

For integration with push-based monitoring tools like StatsD, we need to: 1. Capture any changes in application metrics 2. Encode the metrics into a format that the collector understands (in this case, the StatsD format) 3. Send the metrics to the StatsD server when they

---

<sup>1</sup><https://zio.dev/zio-metrics-connectors>

change The ZIO Metrics Connectors has factored out the first step into a generic module called `MetricClient` that can be used to capture application metrics periodically as a background service. First, we need to add the corresponding dependency to our `build.sbt` file:

```
libraryDependencies += "dev.zio" %% "zio-metrics-connectors" % <ZIO_METRICS_VERSION>
```

The `MetricClient` takes a handler of type `Iterable[MetricEvent] => UIO[Unit]` and creates a background service that will periodically (at configurable intervals) take a snapshot of the metrics, generate a set of metric events, and pass them to the handler:

```
object MetricsClient {
  def make(
    handler: Iterable[MetricEvent] => UIO[Unit]
  ): ZIO[MetricsConfig, Nothing, Unit] = ???
```

Each `MetricEvent` can be `New`, `Updated`, or `Unchanged`. All these events share core properties: a metric key that identifies the metric, the current state value, and a timestamp marking when the event occurred. The `Updated` variant additionally tracks the metric's previous state through its `oldState` field, enabling change detection and historical analysis.

The metric client requires `MetricsConfig` from the ZIO environment, which configures the interval for polling the metric registry.

This interface allows you to write any custom metric client, though ZIO Metrics Connectors already provides several common metric clients out of the box. In this chapter, we'll cover the Prometheus and StatsD clients.

### 35.3.1 Prometheus Client

To create a client for Prometheus, we accept incoming events, encode them into the proper format, and expose them through an HTTP endpoint. Since the return type of the handler is `UIO[Unit]`, we cannot perform long-running effectful operations like serving an HTTP endpoint that exposes metrics. Because the HTTP server would run indefinitely and never finish, its return type would be `UIO[Nothing]`, not `UIO[Unit]`.

So how do we overcome this limitation? A good solution is to create shared memory that stores the final result to be exposed by the HTTP server. We can then run the HTTP server as a separate effect that reads from this shared memory and serves the metrics.

All these steps are already implemented in the `zio-metrics-connectors-prometheus` module. Let's add the corresponding dependency to our `build.sbt` file:

```
libraryDependencies += "dev.zio" %% "zio-metrics-connectors-prometheus" % <ZIO_METRICS_VERSION>
```

The `prometheusLayer` runs the Prometheus client as a background service that periodically captures metrics from the metric registry. After encoding the metrics, it stores them in the `PrometheusPublisher` service. This service is essentially a shared memory where the client's final result is stored. We can then write an HTTP server that reads the metrics from the `PrometheusPublisher` service and serves them:

```

1 import zio.metrics.connectors.prometheus._
2
3 import java.nio.charset.StandardCharsets
4
5 def noCors(response: Response): Response =
6   response.updateHeaders(_.combine(Headers(("Access-Control-Allow-
7   -Origin", "*"))))
8
8 val metricsRoute: Route[PrometheusPublisher, Nothing] =
9   Method.GET / "metrics" -> handler {
10     ZIO.serviceWithZIO[PrometheusPublisher](_.get).map { response
11       =>
12         noCors(
13           Response(
14             status = Status.Ok,
15             headers = Headers(Header.Custom(Header.ContentType.name
16             , "text/plain; version=0.0.4")),
17             body = Body.fromString(response, StandardCharsets.UTF_8
18           )
19           )
20         )
21       )
22     }
23   }
24 }
```

In this example, we'll use ZIO HTTP to create a metrics endpoint that serves data in the Prometheus format. While we'll explore ZIO HTTP in detail later in this book, for now let's focus on setting up this simple endpoint. First, we need to add the following dependency to our `build.sbt` file:

```

1 libraryDependencies += "dev.zio" %% "zio-http" % <
2   ZIO_HTTP_VERSION>
```

To serve metrics, we can use the `Server.install` method:

```

1 val metricsServer: ZIO[PrometheusPublisher with Server, Nothing,
2   Unit] =
3   for {
4     _ <- Server.install(Routes(metricsRoute))
5     _ <- ZIO.logInfo("Metric endpoint started!")
6     _ <- ZIO.never.onInterrupt(_ => ZIO.logInfo("Metric endpoint
7       stopped!"))
8   } yield ()
```

We have written the most important parts, and now we have to run the metrics server as a background service. Since the lifecycle of the metrics server should be the same as the main application's, we need to make it a daemon that is forked in the global scope:

```

1 import zio.metrics.connectors._
2 import zio.metrics.connectors.prometheus._

3
4 object PrometheusClientApp extends ZIOAppDefault {
5   def run = {
6     for {
7       _ <- metricsServer.forkDaemon
8       _ <- mainApp
9     } yield ()
10  }.provide(
11    Server.default,
12    ZLayer.succeed(MetricsConfig(5.seconds)),
13    publisherLayer,
14    prometheusLayer,
15  )
16 }
```

The `mainApp` is the main application that we want to monitor. It contains the business logic of the application and the metrics that we want to track. For example, it can be a simple effect like the one we used in the earlier example of this chapter:

```

1 import zio.metrics._

2
3 val effect =
4   ZIO.debug("Effect executed!")

5
6 val mainApp =
7   Metric.counter("total_execution")
8     .fromConst(1L)(effect)
9     .repeat(Schedule.exponential(100.milliseconds))
```

The `metricsServer` is forked to the global scope and runs as a daemon for as long as the main application is running. It serves metrics at the `localhost:8080/metrics` endpoint in Prometheus format:

```

1 > curl http://localhost:8080/metrics
2 # TYPE total_execution counter
3 # HELP total_execution
4 total_execution 9.0 1736886929913
```

We customized the client to update the metrics every five seconds. You can change this using the `MetricsConfig` layer.

### 35.3.2 StatsD Client

Similar to the Prometheus client, we first have to add the corresponding dependency to our `build.sbt` file:

```
1 libraryDependencies += "dev.zio" %% "zio-metrics-connectors-
  statsd" % <ZIO_METRICS_VERSION>
```

The `statsdLayer` runs the StatsD client as a background service, periodically capturing metrics from the metric registry and publishing them to the specified StatsD server:

```
1 import zio.metrics.connectors.-
2 import zio.metrics.connectors.statsd.-
3
4 object StatsDClientApp extends ZIOAppDefault {
5   def run =
6     mainApp.provide(
7       ZLayer.succeed(MetricsConfig(5.seconds)),
8       ZLayer.succeed(StatsdConfig("127.0.0.1", 8125)),
9       statsdLayer
10    )
11 }
```

In this example, it sends metrics to a local StatsD server listening on port 8125.

## 35.4 Built-in JVM and ZIO Runtime Metrics

In addition to custom metrics you are tracking, ZIO can collect and track JVM and ZIO Runtime metrics:

- **JVM Metrics:** These metrics provide insights into the JVM's performance, including memory usage, garbage collection, and thread management. They are essential for monitoring the health and performance of your application in production environments.
- **ZIO Runtime Metrics:** These metrics track the performance of the ZIO runtime, including fiber counts and execution times. They help you understand how your application is utilizing ZIO's concurrency model and identify potential bottlenecks.

These metrics can be collected either within a ZIO application or as a sidecar service, offering flexibility in implementation.

To collect metrics within a ZIO application, you can add two layers to the application's environment: `DefaultJvmMetrics.live` to collect JVM metrics and `Runtime.enableRuntimeMetrics` to collect ZIO runtime metrics.

```
1 import zio.-
2 import zio.http.-
3 import zio.metrics.-
4 import zio.metrics.jvm.-
```

```

5 import zio.metrics.connectors._
6
7 object JvmMetricAppExample extends ZIOAppDefault {
8   override def run = Server
9     .serve(metricsRoute)
10    .provide(
11      // ZIO HTTP default server layer, default port: 8080
12      Server.default,
13
14      // The prometheus reporting layers
15      prometheus.prometheusLayer,
16      prometheus.publisherLayer,
17
18      // Interval for polling metrics
19      ZLayer.succeed(MetricsConfig(5.seconds)),
20
21      // Default JVM Metrics
22      DefaultJvmMetrics.live.unit,
23
24      // Enable ZIO Runtime Metrics
25      Runtime.enableRuntimeMetrics,
26    )
27 }

```

By default, JVM metrics are collected every 10 seconds. Dynamic metrics, such as buffer pool metrics, need to be restarted to collect new data. By default, dynamic metrics are restarted every minute.

## 35.5 Conclusion

In this chapter, we've explored ZIO's metrics system, which provides powerful tools for monitoring and understanding your application's behavior in production environments.

We learned about the 5 metric types that ZIO supports and their specific use cases:

- Counters for tracking cumulative values
- Gauges for monitoring fluctuating values
- Histograms for analyzing value distributions
- Summaries for calculating quantiles over time windows
- Frequencies for counting occurrences of discrete events

We also covered the integration of metrics with popular monitoring systems through ZIO Metrics Connectors. We explored both pull-based (e.g., Prometheus) and push-based (e.g., StatsD) monitoring approaches, demonstrating how to set up metric clients for each.

Additionally, we examined ZIO's built-in JVM metrics capabilities, which offer essential insights into your application's runtime behavior without requiring additional instrumen-

tation code. This feature is particularly valuable for monitoring production systems and diagnosing performance issues.

As you build complex applications with ZIO, remember that effective monitoring through metrics is not just an add-on feature—it's a crucial aspect of maintaining reliable, performant systems. The tools and patterns we've explored in this chapter provide the foundation for implementing robust observability in your ZIO applications.

## 35.6 Exercises

1. In this chapter, we wrote two metric clients for Prometheus and StatsD. To make your client-related code more reusable, you can extract it and put it in a separate ZIO application. Then you can compose that application with your main application to run them together.

Hint: Use the `ZIOAppDefault#<>` operator to compose two ZIO applications.

# Chapter 36

## 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 and 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 the 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 testing domain. How does the 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.

### 36.1 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:

```
1 import zio._  
2 import zio.test._  
3  
4 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 report 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:

```

1 import org.scalatest._

2

3 def unsafeRun[E, A](zio: ZIO[Any, E, A]): A =
4   Unsafe.unsafe { implicit unsafe =>
5     Runtime.default.unsafe.run(zio).getOrThrowFiberFailure()
6   }

7

8 class ScalaTestSpec extends FunSuite {
9   test("addition works") {
10     assert(
11       unsafeRun(ZIO.succeed(1)) === 2
12     )
13   }
14 }
```

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 this page<sup>1</sup> 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:

```

1 import org.scalatest._
```

---

<sup>1</sup><https://github.com/typelevel/fs2/issues/1009>

```

2
3 class ScalaTestSpec extends AsyncFunSuite {
4   test("addition works") {
5     assert(Runtime.default.unsafeRunToFuture(ZIO.succeed(1)).map(
6       _ === 2))
7   }
}

```

Second, it allows us to unify testing effectful and non-effectful 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.succeed`. Let's see what this would look like:

```

1 lazy val pure: Either[TestFailure[Nothing], TestSuccess] =
2   ???
3
4 lazy val effectual: ZIO[Any, TestFailure[Nothing], TestSuccess] =
5   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 `ZIO#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 timeout 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 `ZIO#acquireRelease` 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 as effects we will see later in this section how we can use all the tools we are familiar with like `ZIO#acquireRelease`, `Scope` 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:

```

1 sealed trait BoolAlgebra[+A]
2
3 final case class And[+A](
4   left: BoolAlgebra[A],
5   right: BoolAlgebra[A]
6 ) extends BoolAlgebra[A]
7
8 final case class Not[+A](
9   result: BoolAlgebra[A]
10) extends BoolAlgebra[A]
11
12 final case class Or[+A](
13   left: BoolAlgebra[A],
14   right: BoolAlgebra[A]
```

```

15 ) extends BoolAlgebra[A]
16
17 final case class Success[+A](value: A)
18   extends BoolAlgebra[A]

```

`BoolAlgebra` is parameterized on some type `A` that typically contains details about the result of asserting a value. By convention, `Success` indicates that the assertion passed, and `Not` indicates that it 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 understanding to complete our discussion of tests as effects.

How then does ZIO Test take the test we write in `test` or `test` and convert it to the `ZTest` data type we saw above?

Every test we write in the body of `test` 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`:

```

1 import zio._
2 import zio.test._
3
4 object ZTest {
5   def apply[R, E](
6     assertion: => ZIO[R, E, TestResult]
7   ): ZIO[R, TestFailure[E], TestSuccess] =
8   ZIO(
9     .suspendSucceed(assertion)
10    .foldCauseZIO(
11      cause => ZIO.fail(TestFailure.Runtime(cause)),
12      assert =>
13        if (assert.isFailure)
14          ZIO.fail(TestFailure.Assertion(assert))
15        else
16          ZIO.succeed(TestSuccess.Succeeded())
17    )
18  }

```

We will walk through this to ensure 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.suspendSucceed`. We do this because we want to ensure we capture any thrown exceptions in the creation of `Assertion` and convert them to test failures so the test framework can properly report them.

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:

```
1 | def bomb[R, E]: ZIO[R, E, TestResult] =
2 |   throw new Exception("Uh, oh!")
```

Ideally, our users would always properly manage their own exceptions, such as, wrapping the exception above in a `ZIO` constructor. But users may not always do this, so at the edges of our application, when users can provide us 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 `ZIO.suspendSucceed`, 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 a higher level of the application must handle. We then use `ZIO#foldCauseZIO` to allow us to handle both the potential failure and the success.

If a failure occurs, 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` of `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.succeed` 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 `test` 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.

## 36.2 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:

```

1 sealed trait ZSpec[-R, +E]
2
3 final case class Suite[-R, +E](
4   label: String,
5   specs: ZIO[R with Scope, TestFailure[E], Vector[ZSpec[R, E]]]
6 ) extends ZSpec[R, E]
7
8 final case class Test[-R, +E](
9   label: String,
10  test: ZIO[R, TestFailure[E], TestSuccess]
11 )

```

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.

This allows a ZSpec to support arbitrary nesting, allowing users great flexibility in defining and organizing their tests.

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 to 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 in a 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 scoped ZIO value. This is important because it allows us to describe resources 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 scoped ZIO value allows us to construct 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.

## 36.3 Conclusion

With the material in this chapter, you should have a better understanding of the “how” of testing, including how the ZIO Test converts the tests you write into effects that describe those tests and how the ZIO Test supports grouping tests into specs.

In the next chapter, we will learn more about ZIO Test’s `Assertion` data type. Although

we were introduced to assertions before, we will see how they 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. We will also go through the full variety of assertions. There are many assertions for testing specific expectations about many data types. This can be a blessing because there is often specific logic already implemented for us. However, it can sometimes be a challenge to find exactly the right assertion for our use case.

We will present a taxonomy of the different assertions that exist. This will make it as easy as possible for you to find the right assertion for the case at hand. We will also go through how you can implement your own assertions. This will help you factor out expectations that you want to test for multiple times.

# Chapter 37

## Testing: Assertions

In the previous chapter, we discussed the basics of testing and how to write tests using the ZIO Test. Now, you are ready to dive deeper into the world of assertions, the core building blocks of tests.

An assertion is a predicate (a boolean-valued function) that is expected to always be true when the test reaches a certain point in your program, usually at the end of the test. It serves as an executable check for a property or invariant that you believe should consistently hold true. If an assertion is evaluated as false, it indicates a discrepancy between the actual and expected results, which causes the test to fail.

### 37.1 Assertions as Predicates

An assertion can be modeled as a predicate that takes a value of type A and returns a boolean, indicating whether the assertion holds true or false. With this definition, functions like `isPositive`, `lessThan(n)`, and `isSome` can be implemented as assertions:

```
1 def isPositive[A](value: A)(  
2   implicit num: Numeric[A]  
3 ): Boolean = num.gt(value, num.zero)  
4  
5 def isLessThan[A](n: A)(value: A)(  
6   implicit ord: Ordering[A]  
7 ): Boolean = ord.lt(value, n)  
8  
9 def isSome[A](value: Option[A]): Boolean =  
10  value.nonEmpty
```

Now, you can assert that a value is positive or that an option is non-empty:

```
1 val sut: Option[Int] = Some(42)
```

```

2 assert(
3   isSome(sut) &&
4     isPositive(sut.get) &&
5     isLessThan(100)(sut.get)
6 )

```

However, this approach has a limitation: the `isSome`, `isPositive`, and `lessThan` functions are not composable. This makes it hard to build complex assertions by combining simpler ones. What we need is a way to express assertions compositionally, like this:

```

1 val sut: Option[Int] = Some(42)
2 assert(isSome(isPositive[Int] && isLessThan(100)).test(sut))

```

To address this, let's redefine `Assertion` and introduce logical operators that allow us to compose assertions:

```

1 trait Assertion[-A] {
2   def test(value: A): Boolean
3
4   def &&[A1 <: A](that: Assertion[A1]): Assertion[A1] =
5     (value: A1) => test(value) && that.test(value)
6
7   def ||[A1 <: A](that: Assertion[A1]): Assertion[A1] =
8     (value: A1) => test(value) || that.test(value)
9
10  def unary_! : Assertion[A] =
11    (value: A) => !test(value)
12 }

```

To simplify the creation of assertions, let's define `isPositive`, `lessThan`, and `isSome` as methods within the `Assertion` companion object:

```

1 object Assertion {
2   def isPositive[A](implicit num: Numeric[A]): Assertion[A] =
3     (value: A) => num.gt(value, num.zero)
4
5   def isLessThan[A](n: A)(implicit ord: Ordering[A]): Assertion[A] =
6     (value: A) => ord.lt(value, n)
7
8   def isSome[A](assertion: Assertion[A]): Assertion[Option[A]] =
9     {
10       case Some(a) => assertion.test(a)
11       case None     => false
12     }
13 }

```

We can now refactor the previous example using the `Assertion` type:

```

1 import Assertion._
2 val sut: Option[Int] = Some(42)
3 assert(
4   isSome(isPositive[Int] && isLessThan(100)).test(sut)
5 )

```

This approach allows for a much more composable way to express assertions. Now, we can easily build complex assertions from simpler ones using the `&&`, `||`, and `unary_!` operators.

However, if we run the above test with `sut` set to `Some(-42)`, the assertion will fail because the value is not positive. The problem is, the developer won't know exactly why the test failed. The test has three assertions, but which one caused the failure?

We should aim to accumulate as much information as possible about failures rather than stopping at the first one. For example, in the test above, if the `isSome` assertion fails, there's no need to evaluate the other two assertions. But if `isSome` passes, and the `isPositive` assertion fails, we should still evaluate `isLessThan` to gather more failure information. This helps us provide a more complete picture of what went wrong, and we don't have to be lazy in evaluating assertions.

To address these issues, we need to reify the chain of assertions into a data structure that retains information about the sequence and combination of assertions used to form a complex assertion. To achieve this, we can use the concept of an arrow. An arrow is a generalization of functions, where functions become first-class citizens, and their composition can be represented as data structures.

Let's define a data type called `TestArrow`, which can be modeled as a sealed trait with different case classes, each representing a specific kind of assertion:

```

1 sealed trait TestArrow[-A, +B]
2
3 case class TestArrowF[-A, +B] (
4   f: A => Result[B]
5 ) extends TestArrow[A, B]
6
7 case class AndThen[A, B, C] (
8   f: TestArrow[A, B],
9   g: TestArrow[B, C]
10 ) extends TestArrow[A, C]
11
12 case class And[A] (
13   left: TestArrow[A, Boolean],
14   right: TestArrow[A, Boolean]
15 ) extends TestArrow[A, Boolean]
16
17 case class Or[A] (
18   left: TestArrow[A, Boolean],

```

```

19   right: TestArrow[A, Boolean]
20 ) extends TestArrow[A, Boolean]
21
22 case class Not[A](
23   arrow: TestArrow[A, Boolean]
24 ) extends TestArrow[A, Boolean]
25
26 case class Suspend[A, B](
27   f: A => TestArrow[Any, B]
28 ) extends TestArrow[A, B]

```

The `Result` data type, which represents the outcome of an assertion, can be defined as follows:

```

1 sealed trait Result[+A]
2 object Result {
3   def succeed[A](value: A) : Result[A]      = Succeed(value)
4   def fail(message: String): Result[Nothing] = Fail(message)
5
6   case class Fail(message: String) extends Result[Nothing]
7   case class Succeed[+A](value: A) extends Result[A]
8 }

```

In the above definition, the `TestArrowF` case class represents a single assertion that can be tested against a value of type `A`. The `AndThen` case class defines the sequential composition of two assertions. The `And` and `Or` case classes represent the logical conjunction and disjunction of assertions, respectively. The `Not` case class represents the logical negation of an assertion. Finally, the `Suspend` case class represents a lazy assertion that can be evaluated against a type `A` value.

With these definitions in place, we can now define operators for composing assertions:

```

1 sealed trait TestArrow[-A, +B] { self =>
2   def >>>[C](that: TestArrow[B, C]): TestArrow[A, C] =
3     AndThen[A, B, C](self, that)
4
5   def &&[A1 <: A](that: TestArrow[A1, Boolean]): TestArrow[A1,
6     Boolean] =
7     And(self.asInstanceOf[TestArrow[A1, Boolean]], that)
8
9   def ||[A1 <: A](that: TestArrow[A1, Boolean]): TestArrow[A1,
10    Boolean] =
11     Or(self.asInstanceOf[TestArrow[A1, Boolean]], that)
12
13   def unary_![A1 <: A](implicit ev: B <:< Boolean): TestArrow[A1,
14     Boolean] =
15     Not(self.asInstanceOf[TestArrow[A1, Boolean]])
16 }

```

Using arrows, we can express `isSome(isPositive[Int] && isLessThan(100))` as something like this:

```

1 def isSomeArrow          : TestArrow[Option[Int], Int] = ???
2 def isPositiveArrow      : TestArrow[Int, Boolean]     = ???
3 def isLessThanArrow(n: Int): TestArrow[Int, Boolean]   = ???
4
5 val isSomePositiveAndLessThan100 =
6   isSomeArrow >>> (isPositiveArrow && isLessThanArrow(100))

```

This enables us to defer evaluating the composed assertions until the test is run. The test runner can then evaluate them one by one and accumulate the results.

Now, let's redefine `Assertion` using the `TestArrow`:

```

1 case class Assertion[-A](arrow: TestArrow[A, Boolean]) {
2   def test(value: A): Boolean = ???
3
4   def run(value: => A): TestResult = ???
5
6   def &&[A1 <: A](that: Assertion[A1]): Assertion[A1] =
7     Assertion(arrow && that.arrow)
8
9   def ||[A1 <: A](that: Assertion[A1]): Assertion[A1] =
10    Assertion(arrow || that.arrow)
11
12  def unary_! : Assertion[A] =
13    Assertion(!arrow)
14}

```

The `Assertion` data type provides two key methods for evaluating assertions: `test` and `run`. The `test` method checks the assertion against a value of type `A` and returns a boolean indicating whether the assertion passed. The `run` method, on the other hand, evaluates the assertion against a value of type `A` and returns a `TestResult`, which contains detailed information, including accumulated error messages.

The `run` method can recursively traverse all assertion nodes, applying the assertion to the given value and gathering the results:

```

1 def run(value: => A): TestResult = {
2   def loop(arrow: TestArrow[A, Boolean], value: A): TestResult =
3     arrow match {
4       case TestArrowF(f) => ???
5       case AndThen(f, g) => ???
6       case And(f, g) => ???
7       case Or(f, g) => ???
8       case Not(f) => ???
9       case Suspend(f) => ???
10    }

```

```

11     loop(arrow, value)
12   }
13 }
```

Now, we can define arrows for the `isPositive`, `isLessThan`, and `isSome` assertions inside the companion object of `Assertion`. For instance, the `isPositive` assertion can be defined as follows:

```

1 def isPositive[A](implicit num: Numeric[A]): Assertion[A] =
2   Assertion(
3     TestArrowF(value =>
4       if (num.gt(value, num.zero))
5         Result.succeed(true)
6       else
7         Result.fail(s"$value was not positive")
8     )
9   )
```

The `isSome` assertion uses the `andThen` or `>>>` operator:

```

1 def isSome[A](assertion: Assertion[A]): Assertion[Option[A]] =
2   Assertion(
3     TestArrowF[Option[A], A] {
4       case Some(value) => Result.succeed(value)
5       case None => Result.fail("Option was none")
6     } >>> assertion.arrow
7   )
```

The actual implementation of the `Assertion` and `TestArrow` data types differs slightly from what we have presented here, but the primary idea is the same. Hence, let's move on to write a simple test using the ZIO Test to check if the given value is an optional integer that is positive and less than 100:

```

1 import zio.test._
2 import Assertion._
3
4 object ExampleTest extends ZIOSpecDefault {
5   def spec = suite("my suite")(
6     test("An optional number should be positive and less than 100
7       ") {
8       val sut = Option(-42)
9       assert(sut)(isSome(isPositive[Int] && isLessThan(100)))
10    }
11  }
```

If we run the test above, it will fail with the following output:

```

1 - An optional number should be positive and less than 100
2   -42 was not positive
3   sut did not satisfy isSome(isPositive[Int] && isLessThan(100))
4   isSome = -42
5   sut = Some(-42)
6   at ExampleTest.scala

```

This output provides the exact reason for the test failure. Now, let's run another test that fails due to multiple assertions:

```

1 - An optional number should be positive and greater than 100
2   -42 was not positive
3   sut did not satisfy isSome(isPositive[Int] && isGreaterThan
4     (100))
5   isSome = -42
6   sut = Some(-42)
7   at ExampleTest.scala
8   -42 was not greater than 100
9   sut did not satisfy isSome(isPositive[Int] && isGreaterThan
10    (100))
11  isSome = -42
12  sut = Some(-42)
13  at ExampleTest.scala

```

This output shows that the test failed because the value was not positive and was not greater than 100. This is a better way to report test failures than just returning a boolean value.

In this section, we also learned that we can nest assertions inside each other to build complex assertions from simpler ones. The `Assertion` data type internally uses the `TestArrow#>>>` (`andThen`) operator to compose such assertions. This enables us to write a wide range of assertions by composing simpler assertions. Now that we are equipped with this powerful tool, we are ready to take an overview of the most common assertions in the ZIO Test.

## 37.2 Common ZIO Assertions

Basic assertions form the foundation of the ZIO Test, providing simple validations such as checking for truth, falsehood, null, and unit values. Examples include `isTrue`, `isFalse`, `isNull`, `isUnit`, `anything`, and `nothing`. The last two assertions are specifically used to pass or fail a test without performing any validation:

```

1 test("Basic assertions") {
2   assert(true)(isTrue) &&
3   assert(false)(isFalse) &&
4   assert(null)(isNull) &&
5   assert(())(isUnit) &&
6   assert(false)(anything) && //always pass

```

```

7 |     assert(true)(nothing) //always fail
8 |

```

Equality assertions can verify both exact matches and approximate equality, such as `equalTo` and `approximatelyEquals`. Numeric assertions encompass a variety of mathematical comparisons, ranging from simple zero checks to more complex range validations, including `isZero`, `isNegative`, `nonNegative`, `isPositive`, `nonPositive`, `isLessThan`, `isLessThanOrEqualTo`, `isGreater Than`, `isGreaterThanOrEqualTo`, and `isWithin`:

```

1 | test("Equality and numeric assertions") {
2 |     assert(42)(equalTo(42)) &&
3 |     assert(42.0001)(approximatelyEquals(42.0, 0.001)) &&
4 |     assert(0)(isZero) &&
5 |     assert(-42)(isNegative) &&
6 |     assert(50)(isGreater Than(42))
7 |

```

String assertions are specifically designed for text-based validations. They enable developers to verify various aspects of string contents, including case sensitivity, length, and pattern matching. This is crucial for testing applications that handle textual data or user inputs. Examples of string assertions include `containsString`, `endsWithString`, `equalsIgnoreCase`, `hasSizeString`, `isEmptyString`, `isNonEmptyString`, `matchesRegex`, and `startsWithString`.

```

1 | test("String assertions") {
2 |     assert("example.com")(endsWithString(".com")) &&
3 |     assert("Hello, World!")(containsString("Hello")) &&
4 |     assert("TEST")(equalsIgnoreCase("test")) &&
5 |     assert("ZIO")(hasSizeString(equalTo(3))) &&
6 |     assert("email@example.com")(matchesRegex("^[^\\s@]+@[^\\s@]+\\.[^\\s@]+$"))
7 |

```

Collection assertions provide powerful tools for validating lists, sequences, and other iterable structures. These assertions can check for the presence of elements, verify orders, and validate collection properties like size and emptiness, such as `contains`, `endsWith`, `exists`, `forall`, `hasAt`, `hasFirst`, `hasLast`, `hasSize`, `isEmpty`, `isNonEmpty`, `startsWith`, `isOneOf`:

```

1 | test("Collection assertions") {
2 |     assert(List(1, 2, 3, 4, 5))(
3 |         contains(3) &&
4 |         hasSize(equalTo(5)) &&
5 |         forall(isLessThan(6)) &&
6 |         exists(isGreater Than(4))
7 |     )
8 |

```

Set and map assertions offer specialized checks for these data structures. Examples of set assertions include `hasSameElements`, `hasSameElementsDistinct`, `hasSubset`, `hasIntersection`, `hasAtLeastOneOf`, `hasAtMostOneOf`, `hasNoneOf`, `hasOneOf`, and `isDistinct`. Examples of map assertions include `hasKey`, `hasKeys`, and `hasValues`.

```

1 test("Set and Map assertions") {
2     assert(Set(1, 2, 3))(hasSubset(Set(1, 2))) &&
3         assert(Map("a" -> 1, "b" -> 2))(
4             hasKey[String, Int]("a") &&
5                 hasValues(hasSameElements(List(1)))
6         )
7 }
```

Structural assertions focus on type-based checks, enabling developers to verify the structure of their data beyond simple value comparisons. These assertions are particularly useful in systems with complex type hierarchies or when working with case classes, sealed traits, and other advanced data structures. Examples include `isCase`, `isSubType`, and `hasField`:

```

1 case class Person(name: String, age: Int)
2 val person: Person = Person("John", 42)
3
4 test("isCase assertion") {
5     assert(person)(
6         isCase[Person, Int](
7             termName = "Person",
8             term = p => Some(p.age),
9             assertion = equalTo(42)
10        )
11    )
12 }
```

Using the `isCase` assertion, we can focus on a specific part of a larger data structure and create nested assertions to validate the structure of a nested data type. This is particularly useful when we want to assert only on a specific nested part of a larger data structure. Similarly, the `hasField` assertion allows us to assert conditions on a specific field within a data structure:

```

1 test("hasField assertion") {
2     assert(person)(
3         hasField("age", _.age, equalTo(42))
4     )
5 }
```

The final structural assertion, `isSubType`, allows you to verify that a value is a subtype of a given type:

```

1 import java.io.IOException
2
3 test("isSubType assertion") {
4   val sut = new IOException("Permission denied")
5   assert(sut)(isSubtype[Throwable](anything))
6 }

```

There are also sum-type specific assertions designed for testing values that represent Option, Either, and Try constructs. These assertions offer specialized checks for handling optional values and error scenarios in tests, including isSome, isNone, isLeft, isRight, isSuccess, and isFailure:

```

1 test("Asserting Optional, Either, and Try values") {
2   assert(Some(5))(isSome(isGreaterThan(3))) &&
3   assert(Right("success"))(isRight(anything)) &&
4   assert(scala.util.Success(42))(isSuccess(equalTo(42)))
5 }

```

ZIO Test also offers various assertions for working with Exit, Cause, and Throwable data types. These assertions focus on error handling and program terminations, enabling developers to verify that specific operations throw expected exceptions or that ZIO effects terminate in particular ways. Examples include throws, throwsA, succeeds, fails, failsCause, dies, isInterrupted, and hasThrowableCause.

Let's try some of these assertions in a test:

```

1 import zio.{ZIO, Cause, FiberId}
2 import zio.Cause.Interrupt
3
4 test("Exit and Cause assertions") {
5   for {
6     success    <- ZIO.succeed(42).exit
7     failure    <- ZIO.fail("error").exit
8     fiberId    <- ZIO.fiberId
9     interrupted <- ZIO.never.fork.flatMap(_.interrupt)
10 } yield {
11   assert(success)(succeeds(equalTo(42))) &&
12   assert(failure)(fails(equalTo("error"))) &&
13   assert(interrupted)(
14     failsCause {
15       isCase[Cause[Any], FiberId](
16         termName = "Cause",
17         term = c => Some(c.asInstanceOf[Interrupt].fiberId),
18         assertion = equalTo(fiberId)
19       )
20     }
21   )
22 }

```

```
23 }
```

Exception assertions focus on error-handling scenarios:

```
1 test("Exception assertions") {
2     assert(throw new IllegalArgumentException("Invalid input"))(
3         throwsA[IllegalArgumentException] &&
4             throws[IllegalArgumentException](
5                 hasMessage(equalTo("Invalid input")))
6             )
7         )
8 }
```

Lastly, the `isSorted` assertion provides a way to verify the order of elements in collections. This is crucial for testing algorithms and data processing tasks where the sequence of elements matters:

```
1 import Assertion._
2 test("Sorting assertions") {
3     assert(List(1, 2, 3, 4, 5))(isSorted)
4 }
```

By using these simple yet powerful assertions, developers can create more advanced assertions. However, there may be cases where you need to create a custom assertion from scratch. This can be done using the `Assertion.assert` constructor, which takes a name and a function that accepts a value of type A and returns a Boolean value.

### 37.3 Smart Assertions

The `Assertion` and its underlying `TestArrow` data types serve as the building blocks for smart assertions. Smart assertions provide a simpler way to write assertions. Instead of using `assert(expr)(assertion)` or `assertZIO(effect)(assertion)`, you can adopt a more readable and concise syntax: `assertTrue(expr)`. With smart assertions, we only need to check whether the given expression is true, and the assertion is automatically inferred from the provided expression. The `assertTrue` macro converts the expression into a series of test arrows and composes them together to form an assertion. Additionally, it eliminates the need for two variants of assertions—one for pure expressions and another for effectful expressions—since the `assertTrue` macro can handle both cases seamlessly.

Instead of writing `assert(true)(isTrue)`, you can simply write `assertTrue(true)`. Likewise, instead of writing `assertZIO(ZIO.succeed(true))(isTrue)`, you can use the following syntax for improved readability and conciseness:

```
1 for {
2     result <- ZIO.succeed(true)
3 } yield assertTrue(result)
```

So, assume you have the following assertions:

```

1 suite("ordinary assertions")(
2   test("assert: pure expression")(
3     assert(1 + 2)(equalTo(3))
4   ),
5   test("assertZIO: pure expression") {
6     assertZIO(
7       for {
8         a <- ZIO.succeed(1)
9         b <- ZIO.succeed(2)
10      } yield a + b
11    )(equalTo(3))
12  }
13 )

```

They can be written as follows:

```

1 suite("smart assertions")(
2   test("assertTrue: pure expression") {
3     assertTrue((1 + 2) == 3)
4   },
5   test("assertTrue: pure expression") {
6     for {
7       a <- ZIO.succeed(1)
8       b <- ZIO.succeed(2)
9     } yield assertTrue((a + b) == 3)
10  }
11 )

```

With smart assertions, pure and effectful assertions are unified into a single syntax, making writing and reading tests easier. This enables us to leverage the full power of the Scala standard library, including Scala collections, allowing us to write assertions more concisely and understandably.

Let's rewrite some previous tests using smart assertions to see how they look. We will begin with the `Exit` and `Cause` assertions:

```

1 test("Exit and Cause assertions") {
2   for {
3     success      <- ZIO.succeed(42)
4     failure      <- ZIO.fail("error!").flip
5     fiberId      <- ZIO.fiberId
6     interrupted  <- ZIO.never.fork.flatMap(_.interrupt)
7     id          <- interrupted.catchAllCause(cause => ZIO.
8     fromOption(cause.interruptOption))
9   } yield assertTrue(success == 42) &&
  assertTrue(failure == "error!") &&

```

```

10     assertTrue(id == fiberId)
11 }
```

In this example, we wrote our expressions using ZIO's built-in operators. This approach makes testing more comfortable for developers, as there is no need to learn a new API; we can work directly with assertions.

Let's try another example that includes testing collections and optional values:

```

1 test("Collection assertions") {
2   val list = List(1, 2, 3, 4, 5)
3   assertTrue(
4     list.contains(3) &&
5       list.length == 5 &&
6       list.forall(_ < 6) &&
7       list.exists(_ > 4)
8   )
9 }
```

If any assertions fail, the test report will display the exact reason for the failure. Let's update the list to `List(1, 2, 3, 4, 5, 6)` and rerun the test. The test will fail with the following output:

```

1 - Collection assertions
2   6 was not equal to 5
3   list.contains(3) &&
4     list.length == 5 &&
5     list.forall(_ < 6) &&
6     list.exists(_ > 4)
7     .length = 6
8   list = List(1, 2, 3, 4, 5, 6)
9   at ExampleTest.scala
10  1 element failed the predicate
11  list.contains(3) &&
12    list.length == 5 &&
13    list.forall(_ < 6) &&
14    list.exists(_ > 4)
15    6 was not less than 6
16  list.contains(3) &&
17  list.length == 5 &&
18  list.forall(_ < 6) &&
19  list.exists(_ > 4)
20  _ = 6
21  at ExampleTest.scala
22  list = List(1, 2, 3, 4, 5, 6)
23  at ExampleTest.scala
```

This output indicates that the test failed because the list length was not equal to 5, and element 6 was not less than 6. The mismatched sections in the actual output are highlighted in yellow and red, which helps developers quickly understand the reason for the failure.

When working with algebraic sum data types, such as `Option`, `Either`, and `Try`, we can assert them using smart assertions like this:

```

1 test("asserting optional value") {
2   val sut = Some(5)
3   assertTrue(
4     sut.exists(_ > 3) // or `sut.map(_ > 3) == Some(3)` or `sut.
5       map(_ > 3).contains(true)`
6   )
}
```

The same goes for `Either` and `Try` data types. This syntax is a bit obscured and needs some improvement. ZIO Test has an extension method for such data types, called `is`, which peeks into the nested data type:

```

1 test("asserting optional value using test lens") {
2   val sut = Some(5)
3   assertTrue(sut.is(_.some) > 3)
4 }
```

This is helpful when we have a deeply nested data type like `Either[String, Option[Int]]`:

```

1 test("asserting either value using test lens") {
2   val sut: Either[String, Some[Int]] = Right(Some(5))
3   assertTrue(sut.is(_.right.some) > 3)
4 }
```

The `is` extension method utilizes `TestLens` to help you focus on the specific part of the data structure you want to assert. For inspecting an optional type, it provides `some`; for `Either`, it includes `left` and `right`; for `Try` values, it offers `success` and `failure`; for `Exit`, it has `die`, `failure`, `success`, and `interrupted`; and finally, for `Cause`, it includes `die`, `failure`, and `interrupted`.

If you have a custom data structure, you can write your custom assertion using `CustomAssertion.make`. For example, assume you have the following custom `Shape` data type:

```

1 sealed trait Shape
2 case class Circle(radius: Double) extends Shape
3 case class Rectangle(width: Double, height: Double) extends Shape
4 case class Triangle(base: Double, height: Double) extends Shape
```

You can write a custom assertion for the `Shape` data type like this:

```
1 test("custom assertion") {
2   val shape: Shape = Circle(1.2)
3   assertTrue{
4     shape.is(_.custom(
5       CustomAssertion.make[Shape] {
6         case Circle(radius) =>
7           Right(radius)
8         case _ =>
9           Left("Expected a circle")
10      }
11    )) == 1.2
12  }
13 }
```

This assertion checks whether the given shape is a circle; if it is, it returns its radius; otherwise, it returns an error message indicating that a circle was expected.

## 37.4 Conclusion

This chapter explored assertions in the ZIO Test, covering their fundamental concepts, implementation, and practical applications. We saw how assertions evolved from simple boolean predicates to more sophisticated and composable structures. This gives you a solid understanding of how assertions work behind the scenes.

The introduction of smart assertions further simplified the testing process by providing a unified, concise syntax for pure and effectful assertions without sacrificing the clarity of test reports.

By leveraging ZIO Test's advanced assertion techniques, developers can ensure the correctness and reliability of their systems, leading to more robust and maintainable Scala applications.

Now we are ready to move on to the next chapter, where we will explore the concept of test services and how we can use them when interacting with environment during the testing process.

## Chapter 38

# Testing: The Test Environment

When we studied ZIO's environment type, we said that one of the advantages of modeling our application's dependencies as services is that it allowed us to provide *test implementations* of those services.

One of the convenient features of ZIO Test is the test implementation of all the standard services in the ZIO environment, including the Clock, Console, Random, and System services.

In this chapter, we will see what each of these test services looks like and learn about the operators they provide to facilitate testing our programs. We will also learn about the tools that ZIO Test provides to modify the functionality of the test services. We will cover how to access the live environment from our tests when needed. Additionally, we will learn how to create test implementations of our own services.

### 38.1 Test Implementation of Standard Services

By default, ZIO Test provides each test with its own version of the `TestEnvironment`. The `TestEnvironment` is like the `ZEnv` provided by default to ZIO applications constructed by extending the `App` trait, except that it provides test versions of each of the standard services.

This has a couple of implications.

First, unless we explicitly access the `Live` environment as discussed below, any operators we invoke within our tests that rely on one or more standard services in the ZIO environment will use the test implementation of that service instead of the live one.

For example, consider this test:

```
1 import zio.{test => _, _}
2 import zio.test._
3
```

```

4 test("tests use test implementations of standard services") {
5   for {
6     _ <- ZIO.sleep(1.second)
7   } yield assertCompletes
8 }
```

We might naively expect that this test will complete after one second because that is the way the `sleep` operator would normally work if we were using the live `Clock`. However, in our tests, `sleep` is implemented in terms of the `TestClock`, which allows us to control the passage of time but requires us to actually do so.

This approach is perfect because, otherwise, we would have no way to test effects involving the passage of time without actually waiting for time to pass.

However, it can be a little counterintuitive at first. Fortunately, the ZIO Test will print a helpful warning for us if we accidentally write a test like this where we delay an effect without actually adjusting the time.

We will highlight throughout our discussion how each of the test implementations of the standard services are different from the live implementations, as well as where there are similarities. Just knowing that you are working with the test implementation of each of these services should allow you to avoid this being an issue for you.

Second, each test gets its own version of each of the standard services in the ZIO environment. Let's look at another example to illustrate this point:

```

1 import zio.test.Assertion._
2
3 suite("system suite")(
4   test("set a system variable") {
5     for {
6       _ <- TestSystem.putEnv("key", "value")
7     } yield assertCompletes
8   },
9   test("get a system variable") {
10    for {
11      value <- System.env("key")
12    } yield assert(value)(isSome(equalTo("value")))
13  }
14)
```

Here, we are working with the test implementation of the `System` service, which doesn't actually allow us to read system or environment variables from the underlying platform but instead reads values from an in-memory map. We can set the values in the map using operators on `TestSystem`, such as the `TestSystem#putEnv` operator used above.

We might naively expect that since we set the environment variable in the first test, we should be able to get it in the second test, so this suite should pass.

However, this suite will actually fail. Each test is provided with its own version of each of

the standard services, so the change to the environment in the first test is not visible in the second test.

Again, this is an excellent thing. If it wasn't true, we could have *interference* between one test and another.

If we had a shared state like this, we would have to add additional logic at the end of each of our tests to restore the state to its original value, or the results of our tests would be dependent on the order of test execution. It would also likely not be possible for us to run tests in parallel, which we want to do for efficiency.

By providing each test with its own version of the standard services, we avoid this problem entirely. Of course, there are situations where we need a shared state for efficiency reasons. For example, creating a Kafka service once that can be used by multiple tests, and we will see tools for dealing with that in the coming chapters, but it is better to avoid that if we can.

With that introduction out of the way, let's learn more about the test implementations of each of ZIO's standard services.

### 38.1.1 Test Implementation of Clock Service

The `TestClock` provides a testable implementation of the `Clock` service. The `TestClock` is quite a challenging task because it must allow us to test effects involving the passage of time that can be quite complex, for example, multiple fibers or many delays.

To accomplish this, the `TestClock` maintains an internal queue of pending tasks. Each time `sleep` is invoked on the `TestClock`, instead of actually scheduling the effect to be run after the specified delay, it is just added to the internal queue of pending tasks.

When we adjust the time, the `TestClock` just goes through the queue of pending tasks in order and runs them one at a time until all tasks scheduled for on or before the new time have been run.

The full interface of the `TestClock` looks like this:

```
1 import java.time._
2
3 trait TestClock {
4   def adjust(duration: Duration): UIO[Unit]
5   def setDateDateTime(dateTime: OffsetDateTime): UIO[Unit]
6   def setTime(duration: Duration): UIO[Unit]
7   def setTimeZone(zone: ZoneId): UIO[Unit]
8   def sleeps: UIO[List[Duration]]
9   def timeZone: UIO[ZoneId]
10 }
```

Three of these methods, `TestClock.adjust`, `TestClock.setDateDateTime`, and `TestClock.setTime`, change the current time maintained by the `TestClock` and cause effects scheduled for on or before that time to be run.

The `TestClock.timeZone` and `TestClock.getTimeZone` operators support getting and setting the time zone maintained by the `TestClock`. However, the time that the `TestClock` uses to determine whether to run effects is based on milliseconds since the epoch, so these operators tend to be less used and are provided primarily for completeness.

The `TestClock.sleeps` operator allows obtaining a “snapshot” of the durations corresponding to each effect that is currently in the task queue. Generally, we won’t use this because we will test effects involving the passage of time directly by advancing the time and directly verifying that an expected effect was performed.

However, sometimes, we merely want to test that something was scheduled for a particular time. For example, we could schedule two effects to be performed with some delay and then verify with `sleeps` that the returned collection has two elements with the appropriate values.

When working with the `TestClock`, generally, the pattern we want to follow when writing our tests is as follows:

1. Fork the effect involving the passage of time that we want to test
2. Adjust the `TestClock` to the time we expect the effect to be completed by
3. Verify that the expected results from the effect being tested have occurred

Let’s see how this pattern works by testing the `sleep` operator on ZIO.

```

1 test("we can test effects involving time") {
2   for {
3     ref    <- Ref.make(false)
4     _      <- ref.set(true).delay(1.hour).fork
5     _      <- TestClock.adjust(1.hour)
6     value <- ref.get
7   } yield assert(value)(isTrue)
8 }
```

This test will pass deterministically with no delay. Notice how we followed the pattern here:

1. On the second line of the `for` comprehension, we forked the effect we wanted to test, which, in this case, sets the `Ref` to `true` after a one-hour delay.
2. On the third line of the `for` comprehension, we adjusted the clock by one hour
3. On the fourth line of the `for` comprehension, we got the value of the `Ref` to verify that it had indeed been set to `true`.

This order is quite important, and if we don’t follow it, we can tell the test framework something we don’t intend, leading to unexpected results.

One mistake we want to avoid is adjusting the `TestClock` after the effect we are testing without forking it.

For example, consider the below test. Don’t do this!

```

1 test("don't adjust the clock after waiting for the effect") {
2   for {
3     ref    <- Ref.make(false)
4     -      <- ref.set(true).delay(1.hour)
5     -      <- TestClock.adjust(1.hour)
6     value <- ref.get
7   } yield assert(value)(isTrue)
8 }
```

This test is identical to the one above except that we did not `fork` the effect on the second line of the `for` comprehension. However, this test will be suspended indefinitely.

When we think about it, the reason is simple.

A `for` comprehension like this reflects a linear ordering of effects. So, we don't adjust the clock on the third line of the `for` comprehension until the `Ref` has already been set on the second line.

However, the second line scheduled an effect for future execution that won't execute until the third line is evaluated. So, this test will be suspended indefinitely.

A related problem can occur if we adjust the time before performing the effect being tested. Consider this test:

```

1 test("don't adjust the clock before starting the effect") {
2   for {
3     ref    <- Ref.make(false)
4     -      <- TestClock.adjust(1.hour)
5     -      <- ref.set(true).delay(1.hour).fork
6     value <- ref.get
7   } yield assert(value)(isTrue)
8 }
```

This test is identical to the first one, except we adjusted the clock before forking the effect we wanted to test. But this one will fail.

The reason is that operators like `delay` have to be interpreted as delaying for the specified duration from the *current time*.

Since we already adjusted the clock by one hour on the second line, the third line schedules the effect to occur one hour after that or two hours from the initial time. Since we never adjust the time by another hour, the delayed effect is never executed, and the `Ref` is never set to the new value.

You can avoid all this by following the simple pattern above. First, fork the test, then adjust the time, and finally, verify the expected results.

A note on implementation is in order here because you may wonder how the `TestClock` guarantees the expected results here in the face of concurrency.

In the recommended pattern, the effect being tested is occurring at the same time as adjust-

ing the clock. So, how do we make sure that the adjustment of the current time does not occur before the effect being tested is scheduled? If it does, the effect could be scheduled even farther in the future and never run, as in our last example.

The answer is that the `TestClock` interfaces with the ZIO Runtime to monitor the status of all fibers forked within the scope of the test and waits for them all to enter a suspended state before proceeding with an adjustment of the time. Thus, when we fork the effect being tested, the `TestClock` knows to allow that effect to complete scheduling its delay before adjusting the time.

This turns out to be quite important for producing correct, deterministic results in more complex situations and allows us to test a wide variety of effects involving the passage of time that would be almost impossible to test otherwise.

For example, say we want to test a schedule we have created with ZIO's `Schedule` data type. We can do that with `TestClock` exactly the same way as we did in the simple example above:

```

1 test("testing a schedule") {
2   for {
3     latch <- Promise.make[Nothing, Unit]
4     ref   <- Ref.make(3)
5     countdown = ref
6       .updateAndGet(_ - 1)
7       .flatMap(n => latch.succeed(()).when(n == 0))
8     _ <- countdown
9       .repeat(Schedule.fixed(2.seconds))
10      .delay(1.second)
11      .fork
12     _ <- TestClock.adjust(5.seconds)
13     _ <- latch.await
14   } yield assertCompletes
15 }
```

Here, we want to test the behavior of an effect that is repeated every two seconds after an initial one-second delay. Specifically, we want to assert that after five seconds, the effect has executed three times.

We start by doing some setup, defining a `Ref` that will be decremented each time the effect runs, a `Promise` that will be completed after the effect has run for the third time, and the effect itself.

Then, we follow the exact same pattern we saw above. We fork the effect being tested, adjust the time, and verify the expected results.

We can even apply this pattern to higher-level data types that use the `Clock` service, such as streams.

Here is another example testing the behavior of the `ZStream#zipWithLatest` operator, which combines two streams into a new stream that will emit a pair of the most recent

elements emitted by each stream each time one of the streams emits a new element:

```

1 test("testing a stream operator involving the passage of time") {
2   val s1 = Stream.iterate(0)(_ + 1).fixed(100.milliseconds)
3   val s2 = Stream.iterate(0)(_ + 1).fixed(70.milliseconds)
4   val s3 = s1.zipWithLatest(s2)((_, _))
5
6   for {
7     q      <- Queue.unbounded[(Int, Int)]
8     _      <- s3.foreach(q.offer).fork
9     fiber  <- ZIO.collectAll(ZIO.replicate(4)(q.take)).fork
10    _       <- TestClock.adjust(1.second)
11    result <- fiber.join
12  } yield assert(result)(
13    equalTo(List(0 -> 0, 0 -> 1, 1 -> 1, 1 -> 2))
14  )
15 }
```

Each of the initial streams is emitting elements with a fixed delay using the `fixed` operator, so we expect that initially we will emit (0, 0), then at 70 milliseconds the second stream will emit 1 and the combined stream will emit (0, 1), at 100 milliseconds the first stream will emit 1 and the combined stream will emit (1, 1), and so on.

Again, we follow the same pattern. We start by doing some setup. In this case, we define the two original streams and our combined stream. We also create a queue that will receive values emitted by the combined stream. Finally, we send values from the combined stream to the queue.

Then, we do our usual three steps: Fork the effect being tested, in this case, running the stream, adjust the clock, and then verify the expected results.

As you can see, the `TestClock` makes it easy to test even very complex effects involving the passage of time. It allows us to set the clock while ensuring that our effects that depend on time behave exactly the same as if we had waited for real time to pass.

### 38.1.2 Test Implementation of Console Service

Fortunately, the task of the `TestConsole` is significantly easier than that of the `TestClock`. The `TestConsole` needs to support testable implementations of the operators on the `Console` service, which fall into two categories: (1) operators that read from the console like `readLine` and (2) operators that write to the console like `printLine`.

The `TestConsole` does this by conceptualizing the console input and output as two buffers.

`readLine` just reads the next value from the input buffer. Similarly, `printLine` just writes the specified value to the output buffer.

Of course, that means we need to be able to add values to the input buffer so that `readLine`

has something to read and, similarly, to get values from the output buffer so we can verify if the values printed with `printLine` are what we expect. And this is just the interface that the `TestConsole` provides:

```

1 trait TestConsole extends Restorable {
2   def clearInput: UIO[Unit]
3   def clearOutput: UIO[Unit]
4   def feedLines(lines: String*): UIO[Unit]
5   def output: UIO[Vector[String]]
6   def outputErr: UIO[Vector[String]]
7 }
```

The two most fundamental operators are `TestConsole.feedLines` and `TestConsole.output`.

The `TestConsole.feedLines` operator adds the specified lines to the input buffer from which `readLine` reads. The first `String` in the list will also be the first one read by `readLine`.

The `TestConsole.output` operator returns the output buffer's contents so that we can verify they have the expected value. The first value printed will be the first value to appear in the output buffer.

There is also a `TestConsole.outputErr` operator that works just like `TestConsole.output` but returns the strings written to standard error versus standard output using `Console.printLineErr`. The `TestConsole` actually keeps track of these separately.

Finally, there are the `TestConsole.clearInput` and `TestConsole.clearOutput` operators, which just delete the current contents of the input and output buffers if we want to reset them and start over.

Let's see how we can use these operators to deterministically test a simple console program:

```

1 import zio.Console._
2
3 import java.io.IOException
4
5 val myConsoleProgram: ZIO[Any, IOException, Unit] =
6   for {
7     _    <- printLine("What's your name?")
8     name <- readLine
9     _    <- printLine(s"Hi $name! Welcome to ZIO!")
10 } yield ()
11
12 test("testing a console program deterministically") {
13   for {
14     _    <- TestConsole.feedLines("Jane")
15     _    <- myConsoleProgram
16     output <- TestConsole.output
17   } yield assert(output)(
```

```
18     equalTo(
19         Vector(
20             "What's your name",
21             "Hi Jane! Welcome to ZIO!")
22         )
23     )
24 }
25 }
```

The actual console program we are testing here prompts the user for their name and prints a greeting. But this program would be impossible to test without the ability to provide an implementation for the console service that does something other than actually writing to the console.

We can see that the implementation of the test was quite easy and again follows a very simple pattern:

1. Feed the `TestConsole` with the test data you would like to use for console input using `TestConsole.feedLines`
2. Run your console program
3. Obtain the output using `TestConsole.output` or `TestConsole.outputErr` and assert that it meets your expectations

The trickiest thing when using the `TestConsole` is probably remembering that `printLine` will insert a new line character after the line that is being printed, so in the example above, our expected output was “Hi Jane! Welcome to ZIO!\n” instead of “Hi Jane! Welcome to ZIO!”. But that isn’t really that tricky at all.

The one other thing to note about the `TestConsole` operator is that by default, it will print values to the standard output in addition to writing them to the output buffer. This isn’t necessary for any test functionality, but we found it to be helpful for debugging purposes. If, for example, you add `printLine("Entering function")` to your code as part of debugging a failing test, you actually see that printed to the console.

If you want to turn this functionality off or on, the easiest way to do it is with the `TestAspect.silent` and `TestAspect.debug` test aspects.

The `silent` test aspect will silence the `TestConsole` so that any values printed to the console with the `TestConsole` service will be added to the output buffer but will not actually be displayed on the console. This can be useful if the console output is “noisy” and clutters up the reporting of test results.

On the other hand, the `debug` test aspect will cause all values written to the output buffer of the `TestConsole` to also be displayed on the actual console. As its name implies, this can be extremely useful for debugging failing tests so that any debugging statements you add to your test or source code are displayed on the console.

By default, the `TestConsole` is set in debug mode, but you can control this using test aspects at the level of a test, a suite, or even all of your specs. So, for example, you can silence a particular test with noisy console output while leaving debug mode enabled for

the rest of your tests.

If you need even more fine-grained control, the `TestConsole` service has two additional operators that allow controlling the mode within the scope of a single effect:

```

1 trait TestConsole {
2   def debug[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A]
3   def silent[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A]
4 }
```

The `TestConsole.debug` operator will set the `TestConsole` to debug mode for the scope of the specified effect, returning it to whatever its previous value was immediately afterward. The `silent` operator will do the same thing, except it will set the `TestConsole` to be silent for the duration of an effect and then restore its previous status afterward.

Typically, you shouldn't have to worry about this at all, or the `silent` and `debug` test aspects will give you more than enough flexibility to control the output of the `TestConsole`. However, if you need extremely fine-grained control of what output gets printed to the console, these operators are here for you.

### 38.1.3 Test Implementation of Random Service

The `TestRandom` service provides a testable implementation of the `Random` service.

It works in two modes.

In the first mode, it serves as a purely functional random number generator. We can set the seed and generate a value based on that seed.

The implementation takes care of passing the updated seed through to the next call of the random number generator, so we don't have to deal with it ourselves. In this role, the `TestRandom` service acts much like the normal `Random` service, providing a solid foundation for other functionality in ZIO Test such as property-based testing that depends on random number generation.

In the second mode, the `TestRandom` service can be used similarly to the `TestConsole` service where we can "feed" it values of a particular type, and then subsequent calls to generate values of that type will return the data we fed to it.

If no data has been fed to it, the `TestRandom` service will generate data using the random seed, functioning in the first mode. If data of a particular type has been fed to it, it will use that data and function in the second mode until it runs out of fed data, at which point it will revert to the first mode.

The interface looks like this:

```

1 trait TestRandom {
2   def clearBooleans: UIO[Unit]
3   def clearBytes: UIO[Unit]
4   def clearChars: UIO[Unit]
5   def clearDoubles: UIO[Unit]
```

```

6  def clearFloats: UIO[Unit]
7  def clearInts: UIO[Unit]
8  def clearLongs: UIO[Unit]
9  def clearStrings: UIO[Unit]
10 def feedBooleans(boolean*: Boolean*): UIO[Unit]
11 def feedBytes(bytes: Chunk[Byte]*): UIO[Unit]
12 def feedChars(chars: Char*): UIO[Unit]
13 def feedDoubles(doubles: Double*): UIO[Unit]
14 def feedFloats(floats: Float*): UIO[Unit]
15 def feedInts(ints: Int*): UIO[Unit]
16 def feedLongs(longs: Long*): UIO[Unit]
17 def feedStrings(strings: String*): UIO[Unit]
18 def getSeed: UIO[Long]
19 def setSeed(seed: Long): UIO[Unit]
20 }

```

There are quite a few methods, but most of them are just variants for different data types, so there are really only four fundamental things going on here.

First, we have the `TestRandom.getSeed` and `TestRandom.setSeed` operators, which support the `TestRandom` service serving as a purely functional random number generator.

We can set the seed to the specified value with the `TestRandom.setSeed` operator and get the current seed with the `TestRandom.getSeed` operator. Given a specified seed, the `TestRandom` service is guaranteed to always generate the same sequence of random values.

For example, we can test this like so:

```

1 for {
2   _      <- TestRandom.setSeed(42L)
3   first <- Random.nextLong
4   _      <- TestRandom.setSeed(42L)
5   second <- Random.nextLong
6 } yield assert(first)isEqualTo(second))

```

This test will always pass because we set the random seed to the same value before generating the first and second numbers.

If we didn't set the random seed after generating the first number, the second number would almost certainly be different. It would be based on an updated random seed created when the first random number was generated. A random number generator that always generates the same number would not be very useful!

We can also get the seed using the `TestRandom.getSeed` operator. This can be useful if we have a test failure to display the seed of the failing test so that we can deterministically replicate it in the future by setting the seed to that value.

The rest of the operators support using the `TestRandom` service where we "feed" it values and there are two types of operators.

Operators like `feedInts` add the specified data to the internal buffer of the `TestRandom` instance, so the next time we call `nextInt`, the first value from the buffer will be returned. There are variants for each of the data types that the `Random` service generates, so, for example, feeding `Int` values will not impact the generation of `Double` values.

The operators like `clearInts` just remove the values currently in the buffer, so we can start over and feed new values or return to generating pseudorandom values based on the random seed if we do not feed any more values.

Here is a simple example of how the `TestRandom` service works in this mode.

```
1 for {
2     _ <- TestRandom.feedInts(1, 2, 3)
3     x <- Random.nextInt
4     y <- Random.nextInt
5     z <- Random.nextInt
6 } yield assert((x, y, z))(equalTo((1, 2, 3)))
```

We see that we fed some values to the `TestRandom` service, and it gave us back exactly what we fed it.

One thing to note about the `TestRandom` service when we explicitly feed it values of a type is that it will always return those values to us when we generate a value of that type, regardless of other invariants.

For example, consider this test:

```
1 for {
2     _ <- TestRandom.feedInts(42)
3     n <- Random.nextIntBounded(10)
4 } yield assert(n)(equalTo(42))
```

This test will pass, even though `nextInt(10)` would normally only return values bounded between 0 and 9. The reason is that since we have explicitly fed the `TestRandom` service an integer when we ask for an integer, it is just going to give it back to us.

Essentially, when we use the `TestRandom` service in this mode, where we explicitly provide it with values that we want it to return, we take responsibility for making sure those values make sense.

Generally, using the `TestRandom` service in the first mode as a purely functional random number generator is the way to go because it doesn't require you to know anything about which or how many random numbers you are going to need to generate. It also always produces values that are consistent with those that the live implementation of the `Random` service would produce.

However, in some cases, feeding specific values to the `TestRandom` service can make sense where you want to have fine-grained control over exactly which values are returned.

### 38.1.4 Test Implementation of System Service

The final test implementation of a service in the default ZIO environment is the `TestSystem` service. This service provides a testable version of functionality for reading system and environment variables, and it works essentially like an in-memory map.

We can “set” system properties and environment variables with the `TestSystem.putEnv` and `TestSystem.setProperty` operators. These don’t actually set any system properties or environment variables but just update an in-memory map maintained internally by the `TestSystem` service.

Subsequent calls to `TestSystem.env`, `TestSystem.property`, or the other variants of these operators on the `System` service that read system properties or environment variables will then simply return data from that in a memory map instead of reading any actual system properties or environment variables.

Here is what the interface looks like.

```

1 trait TestSystem {
2   def putEnv(name: String, value: String): UIO[Unit]
3   def putProperty(name: String, value: String): UIO[Unit]
4   def setLineSeparator(lineSep: String): UIO[Unit]
5   def clearEnv(variable: String): UIO[Unit]
6   def clearProperty(prop: String): UIO[Unit]
7 }
```

We can see that the `TestSystem.putEnv` and `TestSystem.setProperty` operators are just like adding keys and values to a map. The `TestSystem.clearEnv` and `TestSystem.clearProperty` operators just remove a value corresponding to a key from the map.

There is also an operator that allows us to set the line separator that is returned by calls to the `System` service.

Working with the `TestSystem` service is quite straightforward. We just set the environment variables or system properties to whatever we want the effect we are testing to “see”.

## 38.2 Accessing the Live Environment

Normally, the test implementations of the standard services are all we need for testing our programs, as they are explicitly designed to make it easy to test effects involving these services. However, sometimes we may need to access the “live” environment as opposed to the test environment.

For example, we may need to access some environment variable from the actual system we are running on in order to configure our test resources, or we may want to wait for some actual time to pass in one of our tests.

To support these use cases, ZIO Test also provides the `Live` service, which provides access to the “live” versions of all the standard services in the ZIO environment. These live

implementations are the same ones you use every day when you run your ZIO application outside of the ZIO Test.

The interface of the `Live` service looks like this:

```

1 trait Live {
2   def provide[E, A](zio: ZIO[Any, E, A]): IO[E, A]
3 }
4
5 object Live {
6
7   def live[E, A](zio: ZIO[Any, E, A]): ZIO[Live, E, A] =
8     ???
9
10  def withLive[R <: Live, E, E1, A, B](
11    zio: ZIO[R, E, A]
12  )(f: IO[E, A] => ZIO[Any, E1, B]): ZIO[R, E1, B] =
13    ???
14 }
```

The `live` service has a single operator that takes an effect that requires one or more of the standard services in the ZIO environment and returns a new effect that has those dependencies eliminated. It does that by simply providing the effect with the live versions of each of the default ZIO services, which the `Live` service has access to internally.

The best way to use the live service is with the `Live.live` and `Live.withLive` operators defined on the `Live` companion object. The `live` operator is the simplest and just lets us perform a single effect that requires one or more of the standard ZIO services with the live version of that service.

For example, here is how we could access an environment variable from the actual system we are running on from within a test:

```

1 for {
2   port <- Live.live(System.env("port"))
3 } yield assert(port)(isSome(equalTo("42")))
```

The `Live.withLive` operator is slightly more complicated and lets us perform just one operator on an effect with the live environment. For example, say we had a service that depended on our own custom environment, and we wanted to time it out after one second of real-time:

```

1 trait MyCustomEnvironment
2
3 lazy val effect: ZIO[MyCustomEnvironment, Nothing, Boolean] =
4   ???
```

We would like to do something like this:

```

1 Live.live(effect.timeout(1.second))
```

However, we can't do this. The `Live.live` operator requires that the effect have no dependencies other than the standard ZIO services since otherwise, the `Live` service would have no way to provide them.

Furthermore, we don't necessarily want to do this. We really just want to do the timeout with the live `Clock`; we still want the effect to use the test implementation of the standard services.

This is where `withLive` comes to the rescue. We can use it like this:

```
1 | Live.withLive(effect)(_.timeout(1.second))
```

Now the `ZIO#timeout` operator will be run with the live `Clock`, which is what we want, but `effect` will continue to be run with the test implementations of each of the standard ZIO services.

Again, in the vast majority of cases, the test services should provide you with everything you need, and you shouldn't have to worry about this at all. But if you ever do need to access the live versions of any of the standard ZIO services, they are right at hand with the `Live` service.

### 38.3 Creating Custom Test Implementations

The final topic we want to cover in this chapter is implementing testable versions of your own services. ZIO Test provides testable versions of all the standard ZIO services, but you will likely implement many of your own services and want to test logic involving those services.

Of course, the actual services we want to implement will vary widely, but we will try to hit on some common patterns that you can apply to create test implementations of your own services. In the process, we will see parallels with the way the ZIO Test approached similar problems in implementing the test services we learned about above.

In general, we can think of a service as a collection of functions that take some inputs and return some outputs, possibly maintaining some internal state.

To test such as a collection of functions, we would like to be able to:

1. Return appropriate outputs and update the state given a set of inputs
2. Access the state if it is not already observable
3. Set an appropriate initial state
4. Possibly update the state if it is not already supported by the service

To see these requirements in action and how we can fulfill them, we will work through implementing our own version of the `Console` service. For purposes of this exercise, we will use a simplified implementation that looks like this:

```
1 trait Console {  
2     def printLine(line: String): IO[IOException, Unit]  
3     def readLine: IO[IOException, String]
```

```

4 }
5
6 object Console {
7   def printLine(
8     line: String
9   ): ZIO[Console, IOException, Unit] =
10   ZIO.serviceWithZIO(_.printLine(line))
11
12 val readLine: ZIO[Console, IOException, String] =
13   ZIO.serviceWithZIO(_.readLine)
14 }
```

Just by looking at this interface, we can start sketching out a test implementation based on the principles above.

Looking first at the `printLine` operator, we can see by the `Unit` in the type signature that it returns no meaningful output, so it will have to update some internal state. We can also see that there are no operators defined on the `Console` interface that would return that updated state, so we will have to implement one ourselves.

With the `readLine` operator, we can see that it returns a `String` but takes no input, so that `String` value will have to come from the internal state. Again, the `Console` interface does not provide any way for us to set that state, so we will have to implement that ourselves.

At this point, it is helpful to think about what kind of internal state we need to maintain to support our test implementation. We can always refine it later, so we don't need to worry too much about getting the representation exactly right.

Here, it seems like we need two pieces of state to represent the lines we have printed and the lines we will read since there is no logical relationship between them. Since we aren't sure exactly how we will work with them at this point, we will represent them using ZIO's `Chunk` data type, which supports fast appends, prepends, and concatenation.

We can package up the two pieces of state into a single case class like this:

```

1 final case class State(input: Chunk[String], output: Chunk[String])
2
3 object State {
4   val empty: State =
5     State(Chunk.empty, Chunk.empty)
6 }
```

Now that we have defined our state, let's take a first stab at implementing the interface of the `Console` service. Since we want to change the state, we will wrap it in ZIO's `Ref` data type to support modifying the state:

```

1 import java.io.EOFException
2
3 final case class TestConsole(ref: Ref[State]) extends Console {
```

```

4  def printLine(line: String): IO[IOException, Unit] =
5    ref.update(state => state.copy(output = state.output :+ line)
6    )
7  def readLine: IO[IOException, String] =
8    ref.modify { state =>
9      if (state.input.isEmpty) {
10        val io = ZIO.fail(
11          new EOFException("There is no more input left to read")
12        )
13        (io, state)
14      } else {
15        val io = ZIO.succeed(state.input.head)
16        (io, state.copy(input = state.input.tail))
17      }
18    }.flatten
}

```

The implementation of `printLine` is relatively simple. We just update the output buffer to include the new line.

The implementation of `readLine` is slightly more complicated.

We use the `modify` operator on `Ref` because we want to both get the next value from the input buffer and remove that value from the input buffer so the next time we call `readLine` we read the next line from the input buffer. We also have to handle cases where there is no more input in the input buffer; in such cases, we should fail with an `EOFException`.

Notice that we are returning an effect from the `modify` operator and then calling `flatten` to run that effect. As discussed in the chapter on `Ref`, this is a common pattern to construct an effect that depends on the current value of the `Ref` and then run that effect immediately after completing the update operation.

We have made some good progress here, but a couple of things are still missing.

First, we have no way of actually constructing a `TestConsole` value and giving it an initial state.

Second, we have no way of accessing the state right now to verify the expected results. In particular, we have no way to access the output buffer to verify that `printLine` was actually called with the expected values.

Third, we have no way of modifying the internal state, and in particular, we have no way of setting values in the input buffer. We can imagine just doing that as part of constructing the `TestConsole` instance, but it might be nice to be able to do that later.

Let's work through each of these issues in turn.

To construct the `TestConsole`, we can just construct a `Ref` with an empty input and output buffer and call the `TestConsole` constructor with that. Note that since the state is empty, we are committing to provide some interface for updating the state, which we

will do momentarily:

```

1 object TestConsole {
2     val test: ZLayer[Any, Nothing, TestConsole] =
3         ZLayer {
4             for {
5                 ref <- Ref.make(State.empty)
6             } yield new TestConsole(ref)
7         }
8 }
```

We now have a way to construct our `TestConsole` service, but we don't have a way to read from the output buffer to verify that a value was printed to the console, and we also don't have a way to set values in the input buffer.

To remedy this, we need to enrich the interface of `TestConsole` with additional operators beyond those defined on the `Console` interface itself. We could do that like this:

```

1 final case class TestConsole(ref: Ref[State]) extends Console {
2     def feedLine(line: String): UIO[Unit] =
3         ref.update(state => state.copy(input = state.input :+ line))
4     val output: UIO[Chunk[String]] =
5         ref.get.map(_.output)
6     def printLine(line: String): IO[IOException, Unit] =
7         ref.update(state => state.copy(output = state.output :+ line))
8     def readLine: IO[IOException, String] =
9         ref.modify { state =>
10             if (state.input.isEmpty) {
11                 val io = ZIO.fail(
12                     new EOFException("There is no more input left to read"))
13                 (io, state)
14             } else {
15                 val io = ZIO.succeed(state.input.head)
16                 (io, state.copy(input = state.input.tail))
17             }
18         }.flatten
19 }
```

We have added two additional operators to the `TestConsole` interface to address these issues.

The `feedLine` operator just adds the specified line to the input buffer by modifying the `Ref`. We could imagine having other variants that allowed feeding multiple lines, but this will be sufficient for our purposes.

The `output` operator just gets the output buffer from the `Ref` and returns it. This will allow us to verify our expected results.

Since we could implement both of these operators in terms of `Ref` the logic for constructing our `TestConsole` service is largely unchanged.

Here is what our final implementation might look like:

```

1  final case class TestConsole(ref: Ref[State]) extends Console {
2    def feedLine(line: String): UIO[Unit] =
3      ref.update(state => state.copy(input = state.input :+ line))
4    val output: UIO[Chunk[String]] =
5      ref.get.map(_.output)
6    def printLine(line: String): IO[IOException, Unit] =
7      ref.update(state => state.copy(output = state.output :+ line))
8    def readLine: IO[IOException, String] =
9      ref.modify { state =>
10        if (state.input.isEmpty) {
11          val io = ZIO.fail(
12            new EOFException("There is no more input left to read"))
13          (io, state)
14        } else {
15          val io = ZIO.succeed(state.input.head)
16          (io, state.copy(input = state.input.tail))
17        }
18      }.flatten
19  }
20
21
22 object TestConsole {
23   val test: ZLayer[Any, Nothing, TestConsole] =
24     ZLayer {
25       for {
26         ref <- Ref.make(State.empty)
27       } yield new TestConsole(ref)
28     }
29
30   def feedLine(line: String): ZIO[TestConsole, Nothing, Unit] =
31     ZIO.serviceWithZIO(_.feedLine(line))
32
33   val output: ZIO[TestConsole, Nothing, Chunk[String]] =
34     ZIO.serviceWithZIO(_.output)
35 }
```

And we could use it like this:

```

1  val welcomeToZIO: ZIO[Console, IOException, Unit] =
2    for {
3      _    <- Console.printLine("What's your name?")
4      name <- Console.readLine
5    }
```

```
5      _      <- Console.printLine(s"Welcome to ZIO, $name!")
6    } yield ()
7
8 test("testing our console service") {
9   for {
10     _      <- TestConsole.feedLine("Jane Doe")
11     _      <- welcomeToZIO
12     output <- TestConsole.output
13   } yield assert(output)(
14     equalTo(
15       Chunk(
16         "What's your name?",
17         "Welcome to ZIO, Jane Doe!"
18       )
19     )
20   )
21 }
```

## 38.4 Conclusion

In this chapter, we learned about the test implementations provided by ZIO Test for each of the standard services in the ZIO environment. We saw how we could use these test implementations to test complex logic involving effects like console interaction and time efficiently and deterministically.

We also learned how we can create test implementations of our own services. Creating these test implementations can initially require more work than simply using mocks. However, they tend to pay for themselves very quickly as you maintain a code base, so this is a powerful tool to have at your disposal.

With the materials in this chapter, you should be ready to test effects involving any of the standard services in the ZIO environment. Additionally, you will be able to develop your own test implementations. This will make it just as easy to test your own services.

# Chapter 39

## Testing: Test Aspects

We have seen test aspects before. We can use `TestAspect.nonFlaky` to run a test a large number of times to make sure it is stable or `TestAspect.ignore` to ignore a test entirely.

But what are test aspects, and how do they work? How do we know what kinds of things should be test aspects as opposed to ordinary operators?

In this chapter, we will answer these questions.

A helpful starting point is to consider the idea that in any domain, there are concerns about *what* we want to do and concerns about *how* we want to do it. There isn't a hard and fast distinction between these two categories, but typically, “we know it when we see it”.

We can make this idea more specific in the testing domain by distinguishing between *what we want to test* and *how we want to test it*. To make this idea more concrete, consider the following test:

```
1 import zio.{test => _, _}
2 import zio.test._
3 import zio.test.Assertion._
4
5 test("foreachPar preserves ordering") {
6   val zio = ZIO
7     .foreach(1 to 100) { _ =>
8       ZIO.foreachPar(1 to 100)(ZIO.succeed(_)).map(_ == (1 to
9         100))
10      }
11      .map(_.forall(identity))
12      assertZIO(zio)(isTrue)
13 }
```

This test checks the property that the `foreachPar` operator on `ZIO` should preserve the ordering of results. That is, while we want to perform each of the effects in parallel, we

want the returned collection to contain the corresponding results in the order that they originally appeared.

This test involves concurrency, so we would like to run it multiple times to ensure that, at least up to the limit of our testing, the property is always true rather than merely sometimes true. To do this, we use the outer `foreach` to perform the inner effect a hundred times.

Then, inside the inner effect, we actually call `foreachPar` on a collection, performing an effect that just returns the original result unchanged and verify that we get the original collection back. Finally, we gather up all the results and assert that all of them are true.

This test is not very clear. We need an explanation like the one above to understand what is supposed to be happening here instead of what this test is supposed to be doing, which is evident from the test itself.

This is never a good sign.

If we step back and consider what went wrong here, we can see that we are mixing concerns about what we want to test with how we want to test it.

The inner effect describes what we want to test. If we take a collection and call `foreachPar` on it with `succeed`, we should get back the original collection unchanged.

How we want to test it is described by the outer `foreach`. In this case, the “how” is we want to run the test a hundred times and verify that the test passes each time.

When we say it this way, it seems quite clear. But our code is not so clear.

We can fix this by using test aspects. Here is the same test written using test aspects to separate the “what” from the “how”:

```

1 import zio.test.TestAspect._

2
3 test("foreachPar preserves ordering") {
4     for {
5         values <- ZIO.foreachPar(1 to 100)(ZIO.succeed(_))
6     } yield assert(values)(equalTo(1 to 100))
7 } @@ nonFlaky(100)

```

Now, this test is much clearer.

All of the logic about how we want to run the test is bundled up in the `nonFlaky` operator, which handles running the test the specified number of times and verifying that all the results are successes. The body of the test now says exactly what we want to test and is an almost exact translation of our textual description of the property we wanted to check above.

This test is also much more modular.

By separating what we are testing from how we want to test it, we can see that the logic for how we want to test something typically doesn’t care about what it is we are testing.

In the example above, the `nonFlaky` aspect repeats a test a specified number of times and

verifies that all the results pass. As long as it can run the test once and get some result that is a success or a failure, it can run the test many times and verify that all the results are successes.

Similarly, the logic for what we are testing is independent of how we are running the tests. The test above describes how we expect the `foreachPar` operator to behave independently of how many times we run that test.

This modularity is powerful because it allows us to define logic for how we want to run tests separately from defining the tests themselves. This allows us to factor out this functionality into aspects like `nonFlaky` and apply it to many different tests instead of having to reimplement it ourselves.

It also allows us to define different logic for how we want to run tests in a modular way.

In the example above, we only wanted to modify how our test was run in one way by repeating it. But often, we want to modify how our test is run in multiple ways, for example, by repeating it, only running it on a certain platform, and applying a timeout to it.

If we had to do that in each test, not only would we end up repeating ourselves, but the logic of the different ways we wanted to modify how we ran each test would become entangled. The description of how many times we wanted to run the test, when we wanted to run the test, and how long we wanted to wait for the test to run would all be in the same place and mixed up with each other.

In contrast, using test aspects this is easy:

```

1 test("foreachPar preserves ordering") {
2     assertZIO(ZIO.foreachPar(1 to 100)(ZIO.succeed(_))(
3         equalTo(1 to 100)
4     )
5 ) @@ nonFlaky(100) @@ jvmOnly @@ timeout(60.seconds)
```

Notice that we didn't have to change the body of our `test` method at all when we made these changes. This reflects that we now have an appropriate separation of concerns between what we are testing and how we are testing it.

Just as importantly, each of the ways we wanted to modify how the test was run was described by separate test aspects. This enforces modularity between different ways we want to modify our tests because the implementations `nonFlaky`, `jvmOnly`, and `timeout` can't depend on each other.

It also allows us to mix and match these test aspects in different ways to address our specific use case. How many times we want to repeat a test when we want to run it, and how long we want to wait for it are independent questions, and we can answer these questions in different ways simply by applying different test aspects.

Aspects are powerful ideas, and we will see more of them, but for now, let's get more concrete by looking at how aspects are actually represented in the ZIO Test.

## 39.1 Test Aspects as Polymorphic Functions

To think about how we can represent test aspects, we can start by considering what functionality we want tests to provide.

If we look at the examples above, when we apply an aspect to a test with the `@@` operator, we get a new test back. We can prove that to ourselves by adding or removing test aspects and noting that we still always get a test.

We can start by thinking of a test aspect as a function from a test to a test.

We can also apply test aspects to entire suites:

```

1 suite("new feature")(
2   test("some test")(??),
3   test("some other test")(??)
4 ) @@ ignore

```

Here, the `ignore` test aspect causes the test runner to ignore the entire suite. This makes sense because tests and suites are both subtypes of specs, which allows us to write tests and suites together with arbitrary levels of nesting.

So now we can refine our thinking and say that conceptually, a test aspect is a function from a spec to a spec.

We haven't had to worry too much about the internal representation of a spec so far because it is used primarily by ZIO Test, and we just get to write our tests. But here is what the signature looks like:

```

1 trait ZSpec[-R, +E]

```

A `ZSpec` can require some environment `R` and fail with an error of type `E`.

The `R` parameter in `ZSpec` allows a spec to depend on some set of services that are necessary to run the tests. For example, we may need a Kafka service to run a certain suite of tests.

The `E` parameter describes the type of errors that a test can fail with. Most of the time, we don't worry so much about that because if an error occurs, we will just fail the test and report the error, but sometimes, it can be useful to implement error handling logic that depends on the failure that can occur.

Given that, we can sketch out an initial implementation for a `TestAspect` as:

```

1 trait ZSpec[-R, +E]
2
3 trait TestAspect {
4   def apply[R, E](spec: ZSpec[R, E]): ZSpec[R, E]
5 }

```

This definition already says quite a lot. A test aspect can take any spec and return back a new spec of the same type.

In more abstract terms, a test aspect represents a *universally quantified* function from ZSpec to ZSpec. That means that for any spec, a test aspect has to be able to modify that spec and return a new spec with the same type.

This is reflected in the signature of the test aspect, where the R and E type parameters appear in the apply method rather than the signature of TestAspect itself. This means that at the time we implement the test aspect, we don't know the R and E types and have to be able to handle specs with any possible environment and error types.

It turns out we can still implement quite a few useful test aspects with this signature. For example, we can implement the nonFlaky operator because for any environment and error type, if we can run the test once, we can run it multiple times and collect the results.

Also, test aspects implemented this way are quite useful because we can apply them to any spec. Since applying them to a spec always returns a new spec, we can always chain them together.

So, as a first cut, we will say that the test aspect is a universally quantified function that goes from specs to specs.

## 39.2 Ability to Constrain Types

The definition above does not give us quite enough power to implement all the test aspects we want. To see why, consider the timeout test aspect we saw above:

```

1 trait TestAspect {
2   def apply[R, E](spec: Spec[R, E]): Spec[R, E]
3 }
4
5 val timeout: TestAspect =
6   new TestAspect {
7     def apply[R, E](spec: Spec[R, E]): Spec[R, E] =
8       ???
9 }
```

We will get into more detail regarding the structure of the ZSpec data type and implementing our own test aspects later in the chapter, but just thinking about this signature reveals a problem.

To timeout the test, we are going to need to access some functionality related to time, which in ZIO we would do with the Clock service. We probably even want to implement the timeout test aspect in terms of the timeout operator on ZIO, building on our idea that each test is an effect, and we want to leverage the power of the ZIO data type.

So if we are going to use the Clock service, then the ZSpec returned by the apply method on timeout will need to have a type of R `with` Clock. But we can't do that with this type signature because we have to accept any R type and return exactly the same R type back.

For example, if R is Any, then we have to be able to take a ZSpec with an environment type

of Any and return a ZSpec with an environment type of Any. But clearly, we can't do that if we need to depend on the Clock service.

So, our existing signature of TestAspect is not powerful enough to model aspects that depend on an environment. This is a significant limitation because there are many aspects we would like to write that depend on the environment in some way.

We may want to use a service in the environment to implement some aspects, such as using the Clock to timeout a test, as discussed above. We may also want to implement an aspect that modifies some service in the environment, such as changing the default size for property-based tests.

A similar problem arises with respect to the error type.

Some aspects may introduce their own error types. For example, implementing an aspect that opens a database connection before a group of tests and closes it after may fail with an IOException.

We can also have aspects that can only handle certain error types. For instance, we would like to implement a more generalized version of nonFlaky that retries a failing test according to some Schedule, but the schedule may only be able to handle certain types of errors.

What we are saying here is that the signature of TestAspect given above is too general. We would like to be able to keep the idea that a TestAspect is a polymorphic function from spec to spec but constrain the types in some way.

We can do this by introducing upper and lower bounds on the environment and error types of the spec that the test aspect can be applied to like this:

```

1 trait TestAspect[-LowerR, +UpperR, -LowerE, +UpperE] {
2   def apply[R >: LowerR <: UpperR, E >: LowerE <: UpperE](
3     spec: ZSpec[R, E]
4   ): ZSpec[R, E]
5 }
```

Now, we are saying that a test aspect returns a spec with the same environment and error types but allows constraints on the environment and error types to which it can be applied.

Let's see how we can use this power to solve our problem above of implementing the timeout test aspect:

```

1 trait TestAspect[-LowerR, +UpperR, -LowerE, +UpperE] {
2   def apply[R >: LowerR <: UpperR, E >: LowerE <: UpperE](
3     spec: ZSpec[R, E]
4   ): ZSpec[R, E]
5 }
6
7 val timeout: TestAspect[Nothing, Clock, Nothing, Any] =
8   new TestAspect {
9     def apply[R <: Clock, E](spec: ZSpec[R, E]): ZSpec[R, E] =
```

```

10     ???
11 }
```

Now, when we implement the test aspect, we know that R can depend on the `Clock` service, so we can use operators on ZIO like `timeout` in our implementation.

Note that because of variance, the test doesn't have to already require the `Clock` service. The Scala compiler can always add additional environmental requirements to satisfy the type signature.

So, for example, this works even though the original test does not require the `Clock`:

```

1 val spec: Spec[Any, Nothing] =
2   test("label")(ZIO.succeed(assertCompletes))
3
4 val specWithEnvironment: Spec[Live, Nothing] =
5   spec @@ timeout(60.seconds)
```

Even though the original test had no environmental requirements, the Scala compiler automatically added a dependency on the `Clock` service to satisfy the type signature of the `timeout` test aspect, which is exactly what we wanted. When we time out a test, the new test with the timeout logic applied depends on the `Clock` service even if the original one didn't.

The same thing applies to the error type:

```

1 import zio._
2
3 import java.io.IOException
4
5 val specWithFailure: Spec[Any, IOException] =
6   spec @@ after(Console.printLine("after"))
```

The `after` test aspect allows us to perform an arbitrary effect after a test completes execution. For example, it can be used to perform some cleanup. Because that effect can fail, the `after` aspect adds a lower bound for the error type. This reflects the way that the `after` effect can fail:

```

1 def after[R, E](
2   effect: ZIO[R, E, Any]
3 ): TestAspect[Nothing, R, E, Any] =
4   ???
```

Notice here that the test aspect has an upper bound on the environment type of R because whatever other services the test needs, it will now also need the services required by the `after` effect. It also introduces a lower bound on the error type of E, indicating that whatever other ways the original test could fail, it can now also fail with the error type of the `after` effect.

Just like in the example with `timeout` above, we can apply the `after` test aspect to `spec` even though `spec` doesn't require any services and can't fail at all. The Scala compiler can always *narrow* a contravariant type like `R` or *widen* a covariant type like `E`.

The most common type parameter you will see used for test aspects is an upper bound on the environment type, like in the `timeout` example above. This is because many test aspects will make use of one or more services in the environment to implement their functionality, and this allows test aspects to express that.

The next most common type parameter you will see used for test aspects is a lower bound on the error type. Any time a test aspect introduces a new failure mode, for example, by calling an effect that can fail in `TestAspect.after` that will be expressed as a lower bound on the error type.

The final type parameter you may sometimes see is an upper bound on the error type. This occurs with error handling operators like `TestAspect.retry`, which allows retrying a test with a schedule, and since the schedule may only be able to handle certain kinds of errors, the test aspect can only be applied to tests that fail with those types of errors.

In most cases, you shouldn't have to worry about the type parameters of test aspects much because they will "just work", but it can be helpful to understand how they are implemented. This is especially true since aspects are used in other libraries in the ZIO ecosystem, such as Caliban and ZIO Query. Understanding the concept of an aspect will get you one step ahead in working with similar functionality in those libraries.

## 39.3 Common Test Aspects

Now that we understand what a test aspect is, we will discuss some common test aspects. There are a wide variety of test aspects, so we won't be able to cover all of them, but this should give you a good idea of "go to" test aspects for common problems in testing as well as some sense of the power of test aspects.

### 39.3.1 Running Effects Before and After Tests

One common use case for test aspects is running effects before or after each test or a group of tests. This could be useful, for example, to do some setup before a test or cleanup after.

Note that test aspects do not allow the test to access any value returned by the effect, so test aspects are best suited for when you want to do something purely for its effects. If you want to make the value produced by an effect available within your test, make your before and after effects a `ZLayer` and provide it to your tests, as discussed in the next chapter.

There are several variants of these test aspects to cover different use cases:

```

1 def after[R0, E0](
2   effect: ZIO[R0, E0, Any]
3 ): TestAspect[Nothing, R0, E0, Any] =
4   ???
5

```

```

6  def afterAll[R0](
7    effect: ZIO[R0, Nothing, Any]
8  ): TestAspect[Nothing, R0, Nothing, Any] = ????
9
10 def aroundWith[R0, EO, AO](
11   before: ZIO[R0, EO, AO]
12 )(
13   after: AO => ZIO[R0, Nothing, Any]
14 ): TestAspect[Nothing, R0, EO, Any] = ????
15
16 def around[R0, EO](
17   before: ZIO[R0, EO, Any],
18   after: ZIO[R0, Nothing, Any]
19 ): TestAspect[Nothing, R0, EO, Any] = ????
20
21 def aroundAllWith[R0, EO, AO](
22   before: ZIO[R0, EO, AO]
23 )(
24   after: AO => ZIO[R0, Nothing, Any]
25 ): TestAspect[Nothing, R0, EO, Any] = ????
26
27 def aroundAll[R0, EO](
28   before: ZIO[R0, EO, Any],
29   after: ZIO[R0, Nothing, Any]
30 ): TestAspect[Nothing, R0, EO, Any] = ????
31
32 def before[R0, EO](
33   effect: ZIO[R0, Nothing, Any]
34 ): TestAspect[Nothing, R0, EO, Any] = ????
35
36 def beforeAll[R0, EO](
37   effect: ZIO[R0, EO, Any]
38 ): TestAspect[Nothing, R0, EO, Any] = ???

```

There are a good number of variants here, but they all fall into a couple of basic categories.

First, we can do an effect before, after, or “around” a test. Around here, we do one effect before the test starts and a second effect after the test is done, much like we are “bracketing” the test.

Second, we can do that either for each individual test or once for a group of tests. The aspects with the `All` suffix perform the before or after effect once for an entire suite, while the other operators do it for each test.

Finally, the `around` aspects have variants with a `With` suffix where the after effect needs access to the result of the before effect.

Together, these aspects provide a composable solution to problems of needing to do some-

thing before or after a test or a group of tests.

### 39.3.2 Flaky and NonFlaky Tests

Another very common use case for test aspects is dealing with “flaky” tests, which are tests that pass most of the time but exhibit occasional failures.

The first tool that ZIO Test provides is the `nonFlaky` test aspect, which we have seen before. This runs a test many times to make sure it is stable and can be very useful when we test tricky code involving issues like concurrency to ensure a test always passes and didn’t just happen to pass for us when we ran it.

By default, `nonFlaky` runs a test one hundred times, but you can also specify an argument for it if you want to run the test a different number of times. Using a large number of repetitions and a long timeout can be a good tool for testing for concurrency bugs, at least before going to specialized tools such as JCStress.

Clearly, our ideal solution would be to fix the tests or the underlying code so that the tests consistently pass. However, sometimes we may not be in a position to do that yet, and the ZIO Test provides a tool for us there as well in the form of the `flaky` test aspect.

The `flaky` test aspect can be considered the inverse of the `nonFlaky` aspect. It runs a test up to a certain number of times and verifies that the test passes at least one of those times.

This can be useful to verify that while a test is flaky, it is passing at least some of the time until we can identify and address the underlying source of flakiness.

## 39.4 Repeating and Retrying Tests

We have learned that the `nonFlaky` test aspect repeats a test a default number of times or can take a specific number as input. If all the results are successful, the test passes; however, it stops at the first failure and marks the test as failed. This test aspect helps ensure that a test is stable and not just passing by chance.

There are other similar test aspects with different behaviors. For instance, the `repeat` test aspect repeats a test according to a schedule. The schedule can be based on a fixed number of repetitions, a fixed amount of time with a set interval, or any other scheduling strategy. In addition to repeating the test, we can also specify the interval at which the repetition should occur. The test aspect will pass if all results are successful; otherwise, it will fail. Like `nonFlaky`, it stops at the first failure and marks the test as failed.

The `retry` test aspect behaves slightly differently. It retries a failing test according to the given schedule. It stops at the first success and marks the test as passed. If it doesn’t matter how many times the test fails, and you only want to check if it eventually passes, you can use the `eventually` test aspect.

### 39.4.1 Ignoring Tests

Another common thing we may want to do is ignore some tests.

For example, if we follow a test-driven development approach, we might start by writing tests for several features we have not yet implemented. We then implement the first feature and would like to test it.

All of the other tests we have written will obviously fail because we have not implemented those features yet, but that output is not very helpful to us as we are debugging our implementation of the first feature. We can temporarily ignore those other tests simply by using the `ignore` test aspect like this:

```

1 suite("new features")(
2   test("test for first feature")(??),
3   test("test for second feature")(??) @@ ignore
4 )

```

The test for the second feature will not be run and will be highlighted in yellow in the test output so we can focus on what we are currently testing. However, don't forget that we are currently ignoring some tests. You can ignore individual tests or entire suites of tests.

### 39.4.2 Diagnosing Deadlocks

Have you ever had an issue where a test did not terminate, and you didn't know why? This is the situation that the `diagnose` test aspect was designed to address:

```

1 object TestAspect {
2   def diagnose(
3     duration: Duration
4   ): TestAspect[Nothing, Live with Annotations, Nothing, Any] =
5     ???
6 }

```

The `diagnose` runs each test on a separate fiber and prints a fiber dump if the test has not been completed within the specified duration. You can use the fiber dump to see the status of each fiber in a test as well as what suspended fibers are waiting for.

So, if you are facing a test that is deadlocking, this can be an extremely useful tool if the cause of the deadlock is not apparent from the initial inspection.

### 39.4.3 Handling Platform and Version-specific Issues

Our testing logic may sometimes need to depend on the platform we are running on or the current Scala version.

For example, testing some functionality regarding thread pools may only make sense on the JVM. Or some features we want to test may not be available on Scala 3 yet.

Of course, we could just handle this by creating platform or version-specific files. However, this is a pain and makes tests less maintainable, so we should avoid it if possible.

ZIO Test provides an easy solution for these sorts of platform and version-specific issues. It offers a set of test aspects. These aspects allow you to run a test only on a certain platform

or version. Alternatively, you can run the test on all platforms or versions except for a specified one.

Here are the aspects for running tests only on a certain platform or on all platforms except a certain platform:

- `TestAspect exceptJS`
- `TestAspect exceptJVM`
- `TestAspect exceptNative`
- `TestAspect jsOnly`
- `TestAspect jvmOnly`
- `TestAspect nativeOnly`

Here are the aspects for running tests only on a certain Scala version or on all Scala versions except a certain version:

- `TestAspect exceptDotty`
- `TestAspect exceptScala2`
- `TestAspect exceptScala211`
- `TestAspect exceptScala212`
- `TestAspect exceptScala213`
- `TestAspect dottyOnly`
- `TestAspect scala20Only`
- `TestAspect scala211Only`
- `TestAspect scala212Only`
- `TestAspect scala213Only`

These aspects can, of course, be combined. For example, if we wanted to run a test only on Scala 2 on the JVM, we could do it like this:

```
:| test("label")(???) @@ scala20Only @@ jvmOnly
```

There are also variants of each of these operators that can be applied to another test aspect. These variants cause the test aspect to be used only on the specified platform:

- `TestAspect dotty`
- `TestAspect js`
- `TestAspect jvm`
- `TestAspect native`
- `TestAspect scala2`
- `TestAspect scala211`
- `TestAspect scala212`
- `TestAspect scala213`

These can be useful, for example, if you are using `nonFlaky` to repeat a test but only want to do it on the JVM:

```
:| test("label")(???) @@ jvm(nonFlaky)
```

To run a test only on a specific operating system, you can use the `os` test aspect, which takes an operating system as an argument. Alternatively, you can use any of the following

OS-specific test aspects:

- TestAspect.mac
- TestAspect.unix
- TestAspect.windows

#### 39.4.4 Accessing Live Implementation of Test Services

As mentioned earlier, all tests are executed with the default test environment by default. When a test uses any of the built-in services like Random, System, Clock, or Console, the test runner automatically uses their test-specific implementations. If you want to run a test that uses any of these services with their live versions instead, you can apply the following test aspects, which replace the test-specific implementations with the live ones:

- TestAspect.withLiveClock
- TestAspect.withLiveConsole
- TestAspect.withLiveRandom
- TestAspect.withLiveSystem

In the following test, we will verify that sleeping for one second does not take significantly longer than 1000 ms:

```

1 import java.util.concurrent.TimeUnit
2
3 test("Sleeping for one second 'shouldnt take significantly longer
      than 1000 ms.") {
4   val unit = TimeUnit.MILLISECONDS
5   for {
6     a <- Clock.currentTimeMillis(unit)
7     _ <- Clock.sleep(1.seconds)
8     b <- Clock.currentTimeMillis(unit)
9   } yield assert(b - a)(approximatelyEquals(1000L, 10))
10 } @@ withLiveClock

```

If you need the live versions of all these services, you can use the `withLiveEnvironment` test aspect.

#### 39.4.5 Controlling Parallelism

ZIO Test allows you to control the parallelism of your tests. By default, ZIO Test will run all of your tests in parallel. However, sometimes, you may want to run your tests sequentially. You can use the `sequential` test aspect to run your tests sequentially:

```

1 suite("Sequential Test Suite")(
2   test("first test") {
3     for (_ <- ZIO.sleep(1.seconds))
4       yield assertCompletes
5   } @@ withLiveClock,
6   test("second test")(assertCompletes)

```

7 ) `@@ sequential`

If we run the above test, you can see that the test runner waits until the first test is completed before running the second test:

```
1 + Sequential Test Suite
2 + first test
3   + second test
4 2 tests passed. 0 tests failed. 0 tests ignored.
```

Respectively, there is a `parallel` test aspect to make your tests run in parallel. This is the default behavior of the ZIO Test, so you don't need to use this aspect explicitly. Otherwise, you want to change the behavior of a suite that has been marked as sequential.

If you want to control the parallelism factor of your tests, you can use the `parallelN` test aspect, which takes the number of the number of concurrent tests as an argument.

#### 39.4.6 Asserting That a Test Fails

Sometimes, instead of expecting a test to pass, you may want to expect a test to fail. You can use the `failing` test aspect to assert that a test fails:

```
1 test("contains") {
2   val list = Some(List(1, 8, 132, 83))
3   assertTrue(list.get.contains(78))
4 } @@ failing
```

At first glance, it may be useless to assert a test that fails, but it helps us to have a more readable test. Also, when practicing regression testing, if you have a known bug that causes a failure, you can write a test and mark it `failing` to document it as a bug. Once the bug is fixed, the test will start failing, and you can remove the `failing` test aspect.

#### 39.4.7 Timing Tests

Using the live clock in a test has some consequences. For example, if you test a function that should return a result within a certain time, what happens if it is stuck forever? We need to time out such methods to ensure that the test is running in a reasonable time. The `timeout` aspect takes a duration as an argument, and if the test doesn't complete within the specified time, it will fail with a timeout error:

```
1 test("Adjusting the test clock shouldn't interfere with the live
      clock") {
2   for {
3     f1 <- ZIO.sleep(60.seconds).fork
4     f2 <- Live.live(ZIO.sleep(20.seconds)).fork
5     - <- TestClock.adjust(60.seconds)
6     - <- f1.join zipPar f2.join
7   } yield assertCompletes
8 } @@ timeout(5.seconds) @@ failing
```

In this example, run two fibers in parallel, one sleeping for 60 seconds using the test clock and the other one sleeping for 20 seconds using the live clock. By adjusting the test clock to 60 seconds, we expect the passage of the test clock to not interfere with the live clock in fiber one. After adjusting the test clock, the first fiber finishes its work, but the second fiber should wait for 20 seconds. As the whole test has a timeout of 5 seconds, and we have another test aspect that marks the test as a failing test, this test will pass after 5 seconds.

Another useful test aspect is the `timed` aspect. This aspect will measure the time it takes to run a test and print the time in the test report. If we apply this aspect to the above test, we can see the following output after the test is completed:

```
1 + Adjusting the test clock shouldn't interfere with the live
   clock - 5 s 24 ms
```

The `nonTermination` test aspect is the opposite of the `timeout` test aspect. It fails the test if it is completed within the specified time. So, it ensures that an operation continues running for at least the specified duration. Let's rewrite the previous test using `nonTermination`:

```
1 test("Adjusting test clock shouldn't interfere with live clock")
2   {
3     for {
4       f1 <- ZIO.sleep(60.seconds).fork
5       f2 <- Live.live(ZIO.sleep(20.seconds)).fork
6       _ <- TestClock.adjust(60.seconds)
7       _ <- f1.join zipPar f2.join
8     } yield assertCompletes
9   } @@ nonTermination(20.seconds)
```

This test aspect ensures that the test doesn't complete within 20 seconds, which is the time it takes for the second fiber using the live clock to complete its work.

### 39.4.8 Annotation and Tagging

Sometimes, you may need to categorize tests to run them selectively. For instance, you might want to execute only the tests related to a specific feature or limit the run to tests with specific tags. The `tagged` test aspect allows you to add string tags to your tests. You can then filter the specs to execute only the tests with specific tags:

```
1 suite("suite of tagged tests")(
2   test("foo")(assertCompletes) @@ tag("critical"),
3   test("bar")(assertCompletes) @@ tag("important"),
4   test("baz")(assertCompletes) @@ tag("critical", "staging")
5 ).filterTags(_ == "critical").getOrElse(Spec.empty)
```

In this example, we have three tests, each tagged appropriately. We then use the `ZSpec#filterTags` method to run only the tests tagged as `critical`.

If you need to attach additional information to a test, you might want to use `TestAnnotation`. Test annotations are more general than tags; they can be considered key-value pairs attached to a test. ZIO Test uses them internally to track certain information during test execution. Here are some common annotations:

- `TestAnnotation.ignored`: Counts the number of ignored tests.
- `TestAnnotation.repeated`: Tracks how many times a test has been repeated.
- `TestAnnotation.retried`: Tracks how many times a test has been retried.
- `TestAnnotation.timing`: Records the duration of test execution.

When generating a report, the test runner uses these annotations to display the results of each test.

For instance, the `nonFlaky` test aspect annotates a test with the `repeated` annotation, indicating the number of repetitions, while the `flaky` test aspect annotates it with the `retried` annotation, indicating the number of retries.

If you want to manually annotate a test, you can use the `annotate` test aspect. Similar to tags, you can filter your tests based on annotations using `ZSpec#filterAnnotations`.

### 39.4.9 Verifying Post-Conditions

You may want to verify some post-conditions after a test is completed. Assume you have shared a resource between all tests in a suite. You want to ensure that the state of your resource is not polluted after each test. You can use the `verify` test aspect to verify some post-conditions after a test is completed:

```

1 object ResourceManagerSpec extends ZIOSpecDefault {
2     def spec = {
3         suite("a suite of tests with a shared resource manager")(
4             test("allocate and release a resource") {
5                 for {
6                     manager <- ZIO.service[ResourceManager]
7                     _      <- manager.allocate("resource1")
8                     _      <- manager.release("resource1")
9                 } yield assertCompletes
10            },
11            test("simulate a test that fails to release all resources") {
12                for {
13                    manager <- ZIO.service[ResourceManager]
14                    _      <- manager.allocate("resource1")
15                    _      <- manager.allocate("resource2")
16                    // Intentionally not releasing resource2
17                    // _      <- manager.release("resource1")
18                    _      <- manager.release("resource2")
19                } yield assertCompletes
20            }
21        )
22    }
23 }
```

```
21 ) @@ verify(
22   for {
23     count <- ZIO.serviceWithZIO[ResourceManager](_.getAllocatedResourceCount)
24   } yield {
25     assert(count)(equalTo(0)).label("All resources should be
26       released after each test")
27   }
28   ) @@ sequential
29 }.provide(ZLayer.succeed(ResourceManager()))}
```

## 39.5 Conclusion

Test aspects are a powerful feature of ZIO Test that allows us to encapsulate cross-cutting testing logic in a reusable fashion. By separating the “what” from the “how” of testing, aspects promote cleaner test code that focuses on the core logic being tested.

Test aspects are polymorphic functions that transform specs into specs, enabling us to modify how tests are executed in a composable way. ZIO Test provides a rich set of built-in test aspects that cover a wide range of common testing scenarios. By combining these aspects in different ways, we can create test suites that meet the needs of our specific testing requirements.

As you work with the ZIO Test, look for opportunities to factor out common testing patterns into reusable aspects. This will help keep your test code clean and maintainable.

## Chapter 40

# Testing: Using Resources in Tests

When writing tests, you often need to set up resources such as database connections, network sockets, or files and use them in your tests. These resources must be initialized before the test runs and cleaned up afterward. Some resources are expensive to create, or you may want to maintain their state between tests. In such cases, sharing these resources across multiple tests is important.

Like the ZIO Core, the ZIO Test uses the same design principles to provide and manage resources. It uses the layers to provide resources to tests and scopes to manage the lifecycle of resources if needed. Similar to how we access the environment in ZIO workflows using the `ZIO.service*` methods and providing their implementations through the `ZIO#provide*` methods, we can do the same in tests.

In this chapter, you will learn how to provide resources to tests and manage their lifecycle based on your testing requirements.

To simplify the examples, we will use the `Counter` service as a resource. The `Counter` service is a simple service that provides two methods: `inc` to increment the counter and `get` to get the current value of the counter:

```
1 case class Counter(value: Ref[Int]) {  
2     def inc: UIO[Unit] = value.update(_ + 1)  
3     def get: UIO[Int] = value.get  
4 }
```

And we have written a layer for it inside its companion object:

```
1 object Counter {  
2     def inc = ZIO.serviceWithZIO[Counter](_.inc)  
3     def get = ZIO.serviceWithZIO[Counter](_.get)  
4 }
```

```

5  val layer: ZLayer[Any, Nothing, Counter] =
6    ZLayer.scoped(
7      ZIO.acquireRelease(
8        Ref.make(0).map(Counter(_)) <* ZIO.debug("Counter
9          initialized!")
10         )(c => c.get.debug("Number of tests executed"))
11       )
12 }
```

This layer encapsulates the initialization and finalization of the `Counter` service and prints debug messages for pedagogical purposes.

## 40.1 Providing Resources to Tests

Let's start with a simple test that requires the `Counter` service:

```

1 test("counter") {
2   for {
3     _ <- Counter.inc
4   } yield assertCompletes
5 }
```

If we try to run this test, we will get a compilation error because the `Counter` service is not available for the test. We can provide the `Counter` service to the test using the `ZIO#provideLayer` method before running the test, like this:

```

1 test("counter") {
2   {
3     for {
4       _ <- Counter.inc
5     } yield assertCompletes
6   }.provideLayer(Counter.layer)
7 }
```

At this point, you can compile and run the test without any issues. However, this isn't considered idiomatic. Typically, we provide layers to the entire test or test suite rather than providing them directly to the containing `ZIO` effect. Like the `ZIO` effect, all specs have `provide*` methods that allow you to provide layers to the entire test or test suite.

So let's improve the above test accordingly:

```

1 test("counter") {
2   for {
3     _ <- Counter.inc
4   } yield assertCompletes
5 }.provideLayer(Counter.layer)
```

## 40.2 Sharing Resources Between Test Iterations

Sometimes, we may want to repeat (or retry) a test multiple times, such as using the `nonFlaky` test aspect. In these cases, if we provide the required layers before applying the test aspect, the resources will be initialized and finalized for each test repetition.

To illustrate this, let's apply the `nonFlaky` test aspect and configure it to retry the test two more times:

```

1 test("counter") {
2     for {
3         _ <- Counter.inc
4     } yield assertCompletes
5 }.provideLayer(Counter.layer) @@ nonFlaky @@ repeats(2)

```

The output of running this test will look like this:

```

1 Counter initialized!
2 Number of tests executed: 1
3 Counter initialized!
4 Number of tests executed: 1
5 Counter initialized!
6 Number of tests executed: 1
7 + counter - repeated: 2
8 1 tests passed. 0 tests failed. 0 tests ignored.

```

We can observe that the `Counter` service is initialized and finalized for each repetition or retry of the test. This is the expected behavior in most cases. However, there are times when you want to preserve the state of the resources between test iterations.

If you want to preserve the state of the `Counter` service between test iterations, you need to provide the `Counter` to the entire test, including the applied test aspects:

```

1 {
2     test("counter") {
3         for {
4             _ <- Counter.inc
5         } yield assertCompletes
6     } @@ nonFlaky @@ repeats(2)
7 }.provideLayer(Counter.layer)

```

If you run the above test, you will see that the `Counter` service is initialized only once, and the state is maintained between each repetition of the test:

```

1 Counter initialized!
2 Number of tests executed: 3
3 + counter - repeated: 2
4 1 tests passed. 0 tests failed. 0 tests ignored.

```

## 40.3 Providing Resources to Test Suites

Until now, you have learned how to provide resources for individual tests. Let's try the same thing for a suite of tests. We know that both the `test` and `suite` methods return an instance of `Spec`. So it doesn't matter if you provide the resources for a test or a suite of tests; we can do the same for suites:

```

1 suite("suite of tests")(
2   test("foo") {
3     for {
4       _ <- Counter.inc
5     } yield assertCompletes
6   } @@ nonFlaky @@ repeats(5),
7   test("bar") {
8     for {
9       _ <- Counter.inc
10    } yield assertCompletes
11  } @@ nonFlaky @@ repeats(2)
12 ).provideLayer(Counter.layer)

```

If you run the above suite of tests, you will see that the `Counter` service is initialized and finalized per test of the suite:

```

1 + suite of tests
2 Counter initialized!
3 Counter initialized!
4 Number of tests executed: 6
5 Number of tests executed: 3
6 + foo - repeated: 5
7 + bar - repeated: 2
8 2 tests passed. 0 tests failed. 0 tests ignored.

```

Using this approach, you only need to provide resources once rather than for each individual test. ZIO Test takes responsibility for initializing and finalizing the resources before and after executing each test.

## 40.4 Sharing Resources Between Tests

Sometimes, you may want to share resources among all tests in the suite. For instance, how can we maintain the state of the `Counter` service across all tests? What approach can we take to achieve this?

There are two ways to achieve this.

The first solution is to use the `ZSpec#provideLayerShared` method. This method shares the provided layer among all tests in the suite:

```

1 suite("suite of tests")(

```

```

2 test("foo") {
3     for {
4         _ <- Counter.inc
5     } yield assertCompletes
6 } @@ nonFlaky @@ repeats(5),
7 test("bar") {
8     for {
9         _ <- Counter.inc
10    } yield assertCompletes
11 } @@ nonFlaky @@ repeats(2)
12 ).provideLayerShared(Counter.layer)

```

The output of running the test suite above will look like this:

```

1 Counter initialized!
2 + suite of tests
3   + foo - repeated: 5
4   + bar - repeated: 2
5 Number of tests executed: 9
6 2 tests passed. 0 tests failed. 0 tests ignored.

```

Another solution is to run the tests using the `ZIOSpec` trait instead of the default `ZIOSpecDefault` trait. Using the `ZIOSpec` trait, we can override the `bootstrap` method to extend the test environment with the `Counter` service:

```

1 object SharedResourceExampleTests
2 extends ZIOSpec[TestEnvironment with Counter] {
3   override def bootstrap: ULayer[TestEnvironment with Counter] =
4     testEnvironment ++ Counter.layer
5
6   def spec =
7     suite("suite of tests")(
8       test("foo") {
9         for {
10            _ <- Counter.inc
11        } yield assertCompletes
12      } @@ nonFlaky @@ repeats(5),
13      test("bar") {
14        for {
15            _ <- Counter.inc
16        } yield assertCompletes
17      } @@ nonFlaky @@ repeats(2)
18    )
19 }

```

Sharing resources between tests is a common pattern, especially when the resources are expensive to create or when you want to maintain their state across multiple tests.

## 40.5 Conclusion

In this chapter, we've explored various approaches to managing resources in ZIO tests, highlighting the framework's flexibility and power in handling test dependencies. Let's recap the key points:

1. **Providing Resources to Tests:** We learned how to use `provide*` methods to supply resources to individual tests, ensuring that each test has access to the necessary dependencies.
2. **Sharing Resources Between Test Iterations:** We discovered how to maintain resource state across multiple iterations of a single test, which is particularly useful when using test aspects like `nonFlaky` or property-based checking.
3. **Providing Resources to Test Suites:** We saw that providing resources at the suite level can simplify our test setup, allowing ZIO Test to handle resource initialization and cleanup for each test automatically.
4. **Sharing Resources Between Tests:** We explored two methods for sharing resources across multiple tests in a suite:
  - Using `provide*Shared` to share a layer among all tests in a suite.
  - Extending the `ZIOSpec` trait and overriding the `bootstrap` method.

These techniques demonstrate ZIO Test's alignment with ZIO's core principles, offering a consistent and powerful approach to resource management in testing scenarios. By leveraging ZIO's layer system and scopes, we can efficiently manage complex test setups, ensure proper resource cleanup, and create more maintainable and reliable test suites.

By mastering these techniques, you'll be well-equipped to integrate your tests with external resources and manage their lifecycles effectively. You can easily incorporate services like Cassandra, MySQL, PostgreSQL, and Kafka into your tests using `Testcontainers` and an exciting ZIO community library called "testcontainers-for-zio".

---

<sup>1</sup><https://github.com/scottweaver/testcontainers-for-zio>

# Chapter 41

## 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 in which the framework generates test cases for us instead of requiring us to create them 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.

```
1 import zio.test._  
2 import zio.test.Assertion._  
3  
4 test("integer addition is associative") {  
5   val (x, y, z) = (1, 2, 3)  
6   val left      = (x + y) + z  
7   val right     = x + (y + z)  
8   assert(left)(equalTo(right))  
9 }
```

This gives us some reason to believe that integer addition is associative, but there is also something 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, 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 additional evidence that this property is true by adding more tests with different sets of values. It might be helpful to choose values that have no obvious relationship to each other. This approach could increase our confidence that the property holds true in general. However, we will probably only write five or ten of these tests at most,

---

which is not very many.

In addition, each of these tests takes developer time. This includes both writing the tests and thinking about the specific values to test. Developers also need to consider any particular properties that these values should or should 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:

```

1 test("integer addition is associative") {
2     check(Gen.int, Gen.int, Gen.int) { (x, y, z) =>
3         val left  = (x + y) + z
4         val right = x + (y + z)
5         assert(left)(equalTo(right))
6     }
7 }
```

This test will generate a large number of combinations of  $x$ ,  $y$ , and  $z$  values and test whether the assertion is true for all of them.

While property-based testing is a great tool in our toolbox, it is not better than traditional testing in all situations. 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.

First, 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 generate 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, we may not generate an appropriate counterexample even though such a counterexample exists.

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 “edge cases”.

A second issue is that our property-based tests are only as good as the samples we generate.

We introduced property-based testing 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 `int`, 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. However, 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 `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 to be true because the resulting list also has to be in order. That's a second property!

Another trick for developing 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, 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 is right for your use case. One of the great things about the 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 the 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 or more Gen values** - These tell the 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 the 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.

```

1 test("integer addition is associative") {
2   check(Gen.int, Gen.int, Gen.int) { (x, y, z) =>
3     val left = (x + y) + z
4     val right = x + (y + z)
5     assert(left)(equalTo(right))
6   }
7 }
```

On the first line, we construct a property-based test using the `test` operator we have seen before. 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 the 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 `checkZIO` 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 that 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, has only a few variants, and we generally 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, the 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 be able to construct your own generator for any values that are of interest to you in your domain. Finally, in the last section, we will return to the `check` operator and examine how we can control different parameters for how property-based tests are run.

Together, these materials will give you a comprehensive understanding of applying property-based testing to any scenario that is helpful.

## 41.1 Generators as Streams of Samples

So, what is a generator? We know that it can generate values of a given type, but how is a generator actually represented?

In the ZIO Test, a generator is a stream of samples:

```
1 import zio.stream._  
2
```

```

3 final case class Gen[-R, +A](
4   sample: ZStream[R, Nothing, Sample[R, A]]
5 )

```

A `Sample` is a value along with a tree of potential “shrinkings” of that value:

```

1 final case class Sample[-R, +A](
2   value: A,
3   shrinks: ZStream[R, Nothing, Sample[R, A]]
4 )

```

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 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 `int` generator we saw above:

```

1 import zio.stream._
2
3 val integers: Gen[Random, Int] =
4   Gen.int

```

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 advantage 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.

```

1 val deterministic: Gen[Any, Boolean] =
2   Gen.fromIterable(List(true, false))

```

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 generating a value based on one of the lines in that file each time.

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.

## 41.2 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.int`, 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 `Gen.flatMap`, `Gen.map`, and `Gen.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:

```

1 final case class Stock(
2   ticker: String,
3   price: Double,
4   currency: Currency
5 )
6
7 sealed trait Currency
8
9 case object USD extends Currency
10 case object EUR extends Currency
11 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` or `ZStream`, we can use `flatMap` to combine multiple generators into a single value.

```

1 lazy val genTicker: Gen[Any, String] =
2   ???
3

```

```

4 lazy val genPrice: Gen[Any, Double] =
5     ???
6
7 lazy val genCurrency: Gen[Any, Currency] =
8     ???
9
10 lazy val genStock: Gen[Any, Stock] =
11     for {
12         ticker   <- genTicker
13         price    <- genPrice
14         currency <- genCurrency
15     } 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 used 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. We should expect that the ZIO Test will provide us with appropriate generators to construct these values; 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.asciiString` constructor:

```

1 val genTicker: Gen[Random with Sized, String] =
2     Gen.asciiString

```

Notice that the generator's environment type is now `Random with Sized`, indicating that the generator will use both random number generation to generate the values and 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:

```

1 val genPrice: Gen[Random, Double] =
2     Gen.double(0.01, 100000).map(n => math.round(n * 100d) / 100d)

```

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 `Gen.oneOf` constructor to construct generators for data types that may be one of several

alternatives:

```

1 lazy val genUSD: Gen[Any, Currency] =
2   ???
3
4 lazy val genJPY: Gen[Any, Currency] =
5   ???
6
7 lazy val genEUR: Gen[Any, Currency] =
8   ???
9
10 lazy val genCurrency: Gen[Random, Currency] =
11   Gen.oneOf(genUSD, genJPY, genEUR)

```

The `Gen.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 `Gen.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 `Gen.const` constructor for “constant”, which is similar to the `succeed` operator on `ZIO`, and `ZStream`:

```

1 lazy val genUSD: Gen[Any, Currency] =
2   Gen.const(USD)
3
4 lazy val genJPY: Gen[Any, Currency] =
5   Gen.const(JPY)
6
7 lazy val genEUR: Gen[Any, Currency] =
8   Gen.const(EUR)

```

And there we have it. We now have a generator of `Stock` values that we can use in testing our application. 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.

## 41.3 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.

### 41.3.1 Transforming Generators

One of the most basic operators on generators is `map`, which we saw above. The `Gen#map` operator says, “take every value generated by this generator and transform it with the

specified function”.

We can use `Gen#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:

```

1 val ints: Gen[Random, Int] =
2   Gen.int(1, 100)
3
4 val evens: Gen[Random, Int] =
5   ints.map(n => if (n % 2 == 0) n else n + 1)

```

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, as 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 `Gen#map` operator, there is a `Gen#mapZIO` variant that allows transforming the result of a generator with an effectful function:

```

1 trait Gen[-R, +A] {
2   def mapZIO[R1 <: R, B](f: A => ZIO[R1, Nothing, B]): Gen[R1, B]
3 }

```

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 effect’s error type must be `Nothing`. Generators do not have an error type because it does not make sense for them to fail.

If data is invalid, it should not be produced by the generator. If an error occurs in the process of generating test data, it should be treated as a fiber failure. This error should then be 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.

### 41.3.2 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 `Gen#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.”:

```

1 trait Gen[-R, +A] {
2   def flatMap[R1 <: R, B](f: A => Gen[R1, B]): Gen[R1, B]
3 }
```

For example, if we had a generator of positive integers and a generator of lists of a specified size, we could use `Gen#flatMap` to create a new generator of lists with a size distribution based on the first generator:

```

1 def listOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, List[A]] =
2   ???
3
4 val smallInts: Gen[Random, Int] =
5   Gen.int(1, 10)
6
7 def smallLists[R <: Random, A](gen: Gen[R, A]): Gen[R, List[A]] =
8   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 5, the next list produced by `smallLists`, will have five elements.

The `Gen#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 `Gen#flatMap` operator, there are also the `Gen#cross`, and `Gen#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.

For example, if we wanted to generate a pair of integers, we could do it like this:

```

1 val pairs: Gen[Random, (Int, Int)] =
2   Gen.int <*> Gen.int
```

We can easily implement `crossWith` in terms of `map` and `flatMap` as in all other cases:

```

1 def crossWith[R, A, B, C](
2   left: Gen[R, A],
3   right: Gen[R, B]
4 )(f: (A, B) => C): Gen[R, C] =
5   left.flatMap(a => right.map(b => f(a, b)))
```

This generates the cartesian product of all possible pairs of values from the two generators.

In fact, as discussed above, we will often use the convenient `for` comprehension syntax even when generators do not actually depend on each other:

```

1 val pairs2 = for {
2   x <- Gen.int
3   y <- Gen.int
4 } yield (x, y)

```

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:

```

1 def chunkOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, Chunk[A]] =
2   ???
3
4 def listOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, List[A]] =
5   ???
6
7 def mapOfN[R, A, B](
8   n: Int
9 )(key: Gen[R, A], value: Gen[R, B]): Gen[R, Map[A, B]] =
10  ???
11
12 def setOfN[R, A](n: Int)(gen: Gen[R, A]): Gen[R, Set[A]] =
13  ???

```

There are also convenience methods without the N suffix, such as `Gen.listOf` that generate collections with a size range determined by the testing framework and variants with the 1 suffix, such as `Gen!listOf1` that generate non-empty collections with a size range determined by the testing framework.

As an exercise, try implementing the `listOfN` operator yourself based on the operators we have seen so far. Why might the `Gen.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 `Gen#zipWith` and `Gen#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 the testing time for property-based tests is a particular issue for you.

### 41.3.3 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 newly 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 product 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 `Gen#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, `Gen.either` is the most basic operator for sampling from either generator:

```

1 def either[R, A, B](
2   left: Gen[R, A],
3   right: Gen[R, B]
4 ): Gen[R, Either[A, B]] =
5   ???
```

The `either` operator is helpful for generating 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 `Gen.either` and then use `Gen#map` to combine them into a common data type:

```

1 import scala.util.{Failure, Success, Try}
2
3 def genTry[R <: Random, A](gen: Gen[R, A]): Gen[R, Try[A]] =
4   Gen.either(Gen.throwable, gen).map {
5     case Left(e) => Failure(e)
6     case Right(a) => Success(a)
7 }
```

In addition to the `Gen.either` operator, there are operators to construct other common sum types such as `Gen.option` and a couple of helpful convenience methods for combin-

ing more than two alternatives.

The first is the `Gen.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:

```
1 def oneOf[R <: Random, A](gens: Gen[R, A]*): Gen[R, A] =
2     ???
```

The second is the `Gen.elements` operator, which is like `Gen.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 `Gen.elements` like this:

```
1 sealed trait Currency
2
3 case object USD extends Currency
4 case object EUR extends Currency
5 case object JPY extends Currency
6
7 val genCurrency: Gen[Random, Currency] =
8     Gen.elements(JPY, USD, EUR)
```

You may notice that the implementation of `Gen.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 `Gen.weighted` operator, which allows us to specify a collection of generators and weights associated with them.

```
1 def weighted[R <: Random, A](gs: (Gen[R, A], Double)*): Gen[R, A]
2     =
3     ???
```

For example, we could create a generator that generates `true` values 90% of the time and `false` values 10% of the time like this:

```
1 val trueFalse: Gen[Random, Boolean] =
2     Gen.weighted(Gen.const(true) -> 9, Gen.const(false) -> 1)
```

For more complex cases, you can use the `Gen#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:

```
1 val genFailure: Gen[Random, Try[Nothing]] =
2     Gen.throwable.map(e => Failure(e))
```

```

3 | def genSuccess[R, A](gen: Gen[R, A]): Gen[R, Try[A]] =
4 |   gen.map(a => Success(a))
5 |
6 |
7 | def genTryWeighted[R <: Random, A] (
8 |   gen: Gen[R, A]
9 | ): Gen[R, Try[A]] =
10 |   Gen.weighted(genFailure -> 0.1, genSuccess(gen) -> 0.9)

```

#### 41.3.4 Filtering Generators

In addition to combining and choosing between generators, we can also filter generators:

```

1 trait Gen[-R, +A] {
2   def collect[B](pf: PartialFunction[A, B]): Gen[R, B]
3   def filter(f: A => Boolean): Gen[R, A]
4   def filterNot(f: A => Boolean): Gen[R, A]
5 }

```

If we think of a generator as a bag of potential values that we pull a value out of, each time we sample a 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 only filter a small number of generated values, then filtering can be fine and provide a simple way to remove certain degenerate values while maintaining the same distribution of other values. However, if we filter 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, the best practice is, where possible, to transform generated values by using operators like `Gen#map` to turn invalid values into valid values, as in the example with `evens` above, rather than filtering values.

#### 41.3.5 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 uses the generator to generate a large number of values, tests the assertion with those values, and reports the results.

However, sometimes, it can be useful 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 `Gen#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 `Gen#runCollectN`, which repeatedly runs the generator and collects the specified number of samples:

```
1 trait Gen[-R, +A] {
2   def runCollectN(n: Int): ZIO[R, Nothing, List[A]]
3 }
```

Using this, you can quickly get a sense of the distribution of values produced by a generator by doing something like:

```
1 def debug[R <: Console, A](gen: Gen[R, A]): ZIO[R, Nothing, Unit]
2   =
3   gen.runCollectN(100).flatMap { as =>
4     ZIO.foreachDiscard(as)(n =>
5       Console.printLine(n.toString).orDie
6     )
7 }
```

## 41.4 Random and Deterministic Generators

In addition to taking advantage of the power of `ZIO`, one advantage 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 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:

```

1 import zio.test.Assertion._

2

3 check[Gen.boolean, Gen.boolean, Gen.boolean] { (x, y, z) =>
4   val left  = (x && y) && z
5   val right = x && (y && z)
6   assert(left)(equalTo(right))
7 }
```

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 only eight combinations of possible values exist. 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 pre-defined 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, libraries such as SmallCheck 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 and to prove properties.

Historically, these two types of generators have been represented differently, with random generators typically being represented as some “effect” type capable 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.

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.

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`.

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.

Ensure that the generator you are using is finite and small enough for all of its values to be evaluated in a reasonable timeframe before calling the `checkAll` operator!

In the 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 `Gen.fromIterable` constructor. The slightly simplified signature of `fromIterable` is:

```
1 | def fromIterable[A](as: Iterable[A]): Gen[Any, A] =
2 | ???
```

The type signature here is similar to the `Gen.elements` constructor we saw above, but there are important differences:

```
1 | def elements[A](as: A*): Gen[Random, A] =
2 | ???
```

Both of these constructors take a collection of `A` values, so initially, they might seem quite similar. But notice that the environment type of `Gen.elements` is `Random` whereas the environment type of `Gen.fromIterable` is `Any`.

This indicates that `Gen.elements` constructs a random, infinite generator whereas `Gen.fromIterable` constructs a deterministic, finite generator.

Internally, `Gen.elements` is represented as a single-element effectful 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

`Gen.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 the 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:

```
1 val booleans: Gen[Any, Boolean] =
2   Gen.fromIterable(List(true, false))
```

Now, we can test all possible combinations of values by replacing the `check` operator with the `checkAll` operator:

```
1 checkAll(booleans, booleans, booleans) { (x, y, z) =>
2   val left  = (x && y) && z
3   val right = x && (y && z)
4   assert(left)(equalTo(right))
5 }
```

Now, the ZIO Test will generate all possible combinations of `x`, `y`, and `z` values, testing the assertion eight times in total. This allows us to test the property more efficiently and 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 the 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 `Gen#zip\index{Gen!zip}`, which in turn is implemented in terms of `Gen#flatMap`, to generate the product of all possible combinations of these values. So, the values that are generated above correspond to the following generator:

```
1 val triples: Gen[Any, (Boolean, Boolean, Boolean)] =
2   for {
3     x <- booleans
4     y <- booleans
5     z <- booleans
6   } yield (x, y, z)
```

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, `Gen#flatMap`, 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 effectful stream, `Gen#flatMap` just returns a new single-element effectful stream that runs both effects.

So, we get the right behavior for both random infinite generators as well as deterministic finite generators!

The `Gen#zip` operator corresponds to the cartesian product of all possible combinations of two streams. So when we do `Gen#zip` with `booleans` and `booleans`, the generated values will be `(true, true)`, `(true, false)`, `(false, true)`, and `(false, false)`.

## 41.5 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, 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 sizes 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 the 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 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, the 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:

```
1 import zio.stream._

2

3 final case class Gen[-R, +A](
4   sample: ZStream[R, Nothing, Sample[R, A]]
5 )
```

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:

```
1 final case class Sample[-R, +A](
2   value: A,
3   shrink: ZStream[R, Nothing, Sample[R, A]]
4 )
```

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 may, in turn, have its own children, represented by its own shrink tree. 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”, which are earlier in the stream, must be “smaller” than values 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 the `shrink` stream and testing whether its `value` 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, it 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 `Sample#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.

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 the 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 where you may want to think about shrinking as a user is that there are a couple of operators that allow you to control the shrinking logic.

The most useful of these is the `Gen#noShrink` operator:

```

1 trait Gen[-R, +A] {
2   def noShrink: Gen[R, A]
3 }
```

The `Gen#noShrink` operator just removes all shrinking logic from a generator.

You might ask why you would want to do this 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 `Gen#reshrink`:

```

1 trait Gen[-R, +A] {
2   def reshink[R1 <: R, B](f: A => Sample[R1, B]): Gen[R1, B]
3 }
```

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 of shrinking a value is much simpler than the process of generating 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:

```

1 def noShrink[A](a: A): Sample[Any, A] =
2   ???
3
4 def shrinkFractional[A](
5   smallest: A
6 )(a: A)(implicit F: Fractional[A]): Sample[Any, A] =
7   ???
8
9 def shrinkIntegral[A](
10  smallest: A
11 )(a: A)(implicit I: Integral[A]): Sample[Any, A] =
12   ???

```

The `Sample.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 `Sample.shrinkFractional` and `Sample.shrinkIntegral` shrink numeric values towards the specified "smallest" value and are the same ones used internally in the 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:

```

1 lazy val myGeneratorOfComplexDistributions: Gen[Random, Double] =
2   ???
3
4 lazy val myGeneratorThatShrinksTowardZero: Gen[Random, Double] =
5   myGeneratorOfComplexDistributions.reshink(
6     Sample.shrinkFractional(0.0)
7   )

```

In this way, the 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.

## 41.6 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. With the 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.

# Chapter 42

## Testing: Test Annotations

This chapter explores how ZIO Test manages annotations, their underlying implementation, and how to leverage them effectively in test suites.

Annotations are used to record metadata and additional information about tests; they can be used for:

- Tagging tests with labels to categorize them for filtering and selective execution.
- Integrating tests with external systems, such as issue tracker tickets or CI/CD pipelines.
- Assigning priority levels to tests.
- Adding runtime execution metadata, such as duration or memory usage.
- Gathering performance metrics to generate detailed test reports.

Let's begin by learning how to tag tests, which is the most straightforward use of annotations.

### 42.1 Tagging Tests

You can tag and categorize tests based on various aspects, such as their importance, priority, or feature area, using labels like “critical”, “performance”, “dashboard” and so on:

```
1 import zio._  
2 import zio.test.{test, _}  
3 import zio.test.TestAspect._  
4  
5 val mySuite =  
6   suite("suite of tests")(  
7     test("test 1") {  
8       for {  
9         _ <- ZIO.unit  
10      } yield assertCompletes
```

```

11 } @@ tag("critical", "dashboard"),
12 test("test 2") {
13   for {
14     _ <- ZIO.unit
15   } yield assertCompletes
16 } @@ tag("dashboard")
17 )

```

The ZIO test runner will render the tags in the test output like this:

```

1 + suite of tests
2   + test 2 - tagged: "dashboard"
3   + test 1 - tagged: "critical", "dashboard"
4 2 tests passed. 0 tests failed. 0 tests ignored.

```

Each test is tagged with its corresponding labels. You can filter tests based on tags and only run tests with specific tags:

```

1 object FilterTagsExample extends ZIOSpecDefault {
2   def spec =
3     mySuite
4       .filterTags(_ == "critical")
5       .getOrElse(Spec.empty)
6 }

```

This will only run tests tagged with the “critical” label:

```

1 + suite of tests
2   + test 1 - tagged: "critical", "dashboard"
3 1 tests passed. 0 tests failed. 0 tests ignored.

```

Tagging is a specialized form of annotation.

More generally, annotations are key-value pairs that can store any metadata about tests, ranging from simple strings to complex data structures.

Understanding the test annotation requires some knowledge of how it is implemented. So, let’s take a look at the implementation details first.

## 42.2 How Test Annotations Works

Test annotations are a map of key-value pairs that store metadata associated with tests. An empty test annotation map can be created using the `TestAnnotationMap.empty` constructor:

```

1 import zio.test._
2
3 val annotations = TestAnnotationMap.empty

```

We can now add annotations to this map using the `TestAnnotationMap#annotate` method:

```

1 trait TestAnnotationMap {
2   def annotate[V](
3     key: TestAnnotation[V],
4     value: V
5   ): TestAnnotationMap = ???  

6 }
```

This method takes a key of type `TestAnnotation[V]` and its corresponding value of type `V`. The `TestAnnotationMap` is an immutable map responsible for storing annotations. When we use the `TestAnnotationMap#annotate` method, it returns a new map with the updated annotations.

This method takes a typed key containing the combination logic to keep track of multiple annotations of different types. For example, to introduce a new annotation counting how many times the test is repeated, we need a test annotation of type `Int`, such as:

```

1 object TestAnnotation {
2   val repeated: TestAnnotation[Int] =
3     TestAnnotation(
4       identifier = "repeated",
5       initial = 0,
6       combine = _ + _
7     )
8 }
```

Now we can update our annotation map with the `repeated` annotation:

```

1 import zio.test.TestAnnotation._  

2  

3 val updatedAnnotations =
4   annotations.annotate(repeated, 1)
```

This attempts to retrieve the value of `repeated` from the `annotations`. In this case, it doesn't have any corresponding value because the map is empty, so it uses the `initial` value and updates it with the `combine` logic and the provided value. Finally, it returns a new `TestAnnotationMap` with the updated value.

Since it returns a new `TestAnnotationMap`, we can chain multiple annotations together:

```

1 import zio.test.TestAnnotation._  

2  

3 val updatedAnnotations =
4   annotations
5     .annotate(repeated, 1)
6     .annotate(repeated, 1)
```

Whenever we need to retrieve a value from the annotation map, we can use the `TestAnnotationMap#get` method:

```
1 val repetitions: Int =
2   updatedAnnotations.get(repeated)
```

This will return the corresponding value of the `repeated` annotation, which is 2 in this case.

This is how the underlying annotation map works. To simplify usage, ZIO Test provides a more user-friendly API for working with annotations and handles the maintenance of the annotation map internally.

## 42.3 Using Test Annotations in Tests

In the first section of this chapter, we discussed how to tag tests. Now that we've explored the underlying implementation details, let's see how we can achieve this directly using the test annotations API.

The simplest way to annotate a test is to use the `ZSpec#annotate` method. Let's see how we can annotate a test with the `tagged` annotation:

```
1 test("my test") {
2   for {
3     _ <- ZIO.unit
4   } yield assertCompletes
5 }.annotate(tagged, Set("critical", "dashboard"))
```

In this example, we used the `TestAnnotation.tagged` annotation key, which is a built-in annotation for tagging tests:

```
1 object TestAnnotation {
2   val tagged: TestAnnotation[Set[String]] =
3     TestAnnotation("tagged", Set.empty, _ union _)
4 }
```

It defines the `tagged` annotation with an initial value of an empty set and a `combine` function that merges two sets. ZIO Test uses this definition to update the internal annotation map with the provided value.

Also, you can use the `@@` operator to apply annotations to tests using the `annotate` test aspect:

```
1 test("my test") {
2   for {
3     _ <- ZIO.unit
4   } yield assertCompletes
5 } @@ annotate(tagged, Set("critical", "dashboard"))
```

Like the ZSpec#filterTags, you can filter tests based on annotations using the ZSpec #filterAnnotations method:

```

1 import zio.test.TestAnnotation._

2

3 object FilterTagsExample extends ZIOSpecDefault {
4   def spec =
5     mySuite
6       .filterAnnotations(tagged)(_.contains("critical"))
7       .getOrElse(Spec.empty)
8 }
```

There are also two effectful operators for working with annotations: Annotations.annotate and Annotations.get:

```

1 object Annotations {
2   def annotate[V](
3     key: TestAnnotation[V],
4     value: V
5   ): UIO[Unit] = ???
6   def get[V](
7     key: TestAnnotation[V]
8   ): UIO[V] = ???
9 }
```

The Annotations.annotate operator takes a key of type TestAnnotation[V] and a corresponding value of type V, updating the internal annotation map. Accordingly, you can use Annotations.get to retrieve the annotation's value.

These operators are a good fit for updating annotations when implementing test aspects.

## 42.4 Using Test Annotations in Test Aspects

We sometimes need to add metadata to each test when writing test aspects. Thus, it is common to use test annotations when designing test aspects. Hence, other than the tagged annotation we discussed earlier, ZIO Test provides several built-in annotations within the TestAnnotation object:

- **TestAnnotation.ignored**: Counts the number of ignored tests.
- **TestAnnotation.repeated**: Tracks how many times a test has been repeated.
- **TestAnnotation.retried**: Counts the number of times a test has been retried after failure.
- **TestAnnotation.timing**: Records the execution time of a test.

Built-in test aspects also use these annotations to generate detailed test reports. For example, the nonFlaky test aspect repeats the test and updates the corresponding repeated annotation for each repetition, tracking the total number of repetitions. Likewise, the

timed test aspect measures the execution time of the test and updates the timing annotation accordingly.

Let's take a closer look at the non-flaky testing aspect. The core idea of the nonFlaky test aspect is to repeat the test multiple times. To achieve this, we can use the ZIO#repeatN operator:

```

1 val nonFlaky: TestAspectPoly = {
2   val nonFlaky = new PerTest.Poly {
3     def perTest[R, E](
4       test: ZIO[R, TestFailure[E], TestSuccess]
5     )(implicit trace: Trace): ZIO[R, TestFailure[E], TestSuccess]
6     =
7       for {
8         _ <- test
9         repeats <- TestConfig.repeats
10        result <- test.repeatN(repeats - 1)
11      } yield result
12    }
13  restoreTestEnvironment >>> nonFlaky
}

```

We read the number of repetitions from the TestConfig and repeat the test n times. However, we must also annotate the test with the number of repetitions. Let's change the nonFlaky test aspect to update the repeated annotation for each repetition:

```

1 import zio._
2 import zio.test._
3 import zio.test.TestAspect._

4
5 val nonFlaky: TestAspectPoly = {
6   val nonFlaky = new PerTest.Poly {
7     def perTest[R, E](
8       test: ZIO[R, TestFailure[E], TestSuccess]
9     )(implicit trace: Trace): ZIO[R, TestFailure[E], TestSuccess]
10    =
11      for {
12        _ <- test
13        repeats <- TestConfig.repeats
14        result <- test.tap { _ =>
15          Annotations.annotate(
16            key = TestAnnotation.repeated,
17            value = 1
18          )
19        }.repeatN(repeats - 1)
20      } yield result
21    }
22  restoreTestEnvironment >>> nonFlaky
}

```

```
22 }
```

Now, we can apply the `nonFlaky` test aspect to a test:

```
1 test("my test") {
2   for {
3     _ <- ZIO.unit
4   } yield assertCompletes
5 } @@ nonFlaky @@ repeats(42)
```

Since the ZIO Test runner has a built-in `TestAnnotationRenderer` for the repeated annotation, it displays the number of repetitions in the test output like this:

```
1 + my test - repeated: 42
2 1 tests passed. 0 tests failed. 0 tests ignored.
```

## 42.5 Implementing Test Annotation Reporter

But what if we want to create a custom annotation, and how will the test runner display it? At the time of writing, the ZIO Test doesn't have this feature to introduce a new annotation renderer. Still, you can create a custom reporter and use the `after` test aspect to display the custom annotation in the test output. Let's see how we can do this:

```
1 test("repeated test"){
2   for {
3     _ <- ZIO.unit
4   } yield assertCompletes
5 } @@ nonFlaky @@ repeats(3) @@ after(
6   Annotations
7     .get(TestAnnotation.repeated)
8     .debug("number of repetitions")
9 )
```

In this example, besides the default test report, we used the `after` test aspect to get the value of the `repeated` annotation and print it. We can use `Annotations.annotate` anywhere in the test body or by applying it as a test aspect using the `@@` operator. In this example, we used the `after` test aspect.

Here is the console output:

```
1 number of repetitions: 3
2 + repeated test - repeated: 3
3 1 tests passed. 0 tests failed. 0 tests ignored.
```

The `Annotations.annotate` operator takes a key of type `TestAnnotation[V]` and its corresponding value of type `V` and updates the internal annotation map. Accordingly, you can use the `Annotations.get` to retrieve the annotation's value.

## 42.6 Conclusion

Test annotations are powerful tools for augmenting tests with meaningful metadata. In this chapter, we explored how they are implemented and how to use them.

We also talked about writing custom annotations, a flexible way to define new metadata for your tests. The test runner knows how to render some built-in annotations, such as `tagged`, `repeated`, and `timing`, but if you have created a custom annotation, you have to implement a custom reporter to display it.

We also discussed the process of creating custom annotations, a flexible approach to define and incorporate new metadata tailored to your specific testing needs. You learned how to create custom test reporters for custom annotations.

By mastering test annotations, you can develop maintainable test suites where tests are enriched with valuable metadata. This leads to comprehensive and detailed test reports, ultimately improving the quality and maintainability of your testing process.

# Chapter 43

## Testing: Reporting

This is the final chapter of the testing series. In this chapter, we will discuss how to use test reports to analyze the test results.

Test reporting is a crucial aspect of the software development lifecycle, especially as projects grow in size and complexity. As a project grows, it becomes more challenging to analyze test results effectively. Investigating manually through test logs and test outputs can be tedious, especially when there are many test cases and various modules. This is where test reports become invaluable.

Test reports are a structured representation of test results that can be analyzed to gain insights into the codebase's stability, performance, and quality. Analyzing test reports helps us identify patterns of failures, performance bottlenecks, and instability in the codebase.

In this chapter, we will explore how to collect data from tests and generate test reports.

### 43.1 Gathering Data

Gathering data is the first step in generating test reports. The default format in which the console renderer renders the test result is not suitable for analysis. We need to store the test results in a structured format.

ZIO has a built-in JSON serializer that automatically serializes the test results into the JSON format inside the `target/test-reports-zio/output.json` file. It includes the test name, status (success or failure), duration, test annotations, and more.

For example, assume you have written the following tests:

```
1 import zio.test._  
2 import zio.test.TestAspect._  
3  
4 object ExampleSpec extends ZIOSpecDefault {  
5   def spec =
```

```

6   suite("suite of tests")(
7     test("foo") {
8       assertTrue(true)
9     } @@ tag("critical", "dashboard") @@ nonFlaky,
10    test("bar") {
11      assertTrue(true)
12    } @@ tag("dashboard") @@ timed
13  )
14}

```

The ZIO test runner will generate something like the following JSON output inside the target/test-reports-zio/output.json file:

```

1 {
2   "results": [
3     {
4       "name" : "test_case_1761515416/suite of tests/bar",
5       "status" : "Success",
6       "durationMillis" : "107",
7       "annotations" : "tagged: \"dashboard\" : 17 ms",
8       "fullyQualifiedClassName" : "test_case_1761515416",
9       "labels" : ["suite of tests", "bar"]
10    },
11    {
12       "name" : "test_case_1761515416/suite of tests/foo",
13       "status" : "Success",
14       "durationMillis" : "186",
15       "annotations" : "repeated: 100 : tagged: \"critical\", \""
16         dashboard\",
17       "fullyQualifiedClassName" : "test_case_1761515416",
18       "labels" : ["suite of tests", "foo"]
19    }
20  ]
}

```

You can also generate JUnit XML reports by integrating your project with SBT. To do this, add the following line to your build.sbt file:

```

1 libraryDependencies += "dev.zio" %% "zio-test-sbt" % zioVersion

```

Then, run sbt test to execute the tests, and JUnit XML reports will be generated in the target/test-reports directory.

You can gather the output from all your modules during the CI/CD pipeline and store it in a database or a file for further analysis. You can use a simple file-based database like SQLite or a full-fledged time-series database to prepare the data for future queries.

## 43.2 Analyzing Data

Once you've gathered and stored the test results, you can begin analyzing the data. Depending on your needs, you can define the specific metrics you want to track and evaluate. Below are some metrics you can calculate from the test results:

- **Test Pass/Failure Rate:** You can calculate the percentage of tests that pass or fail. Tracking this metric over time helps you understand the stability of the codebase.
- **Test Duration Analysis:** You can calculate the average test duration over time, which helps you understand the test suite's performance. Also, you can identify the top N slowest test and keep track of them over time. You can also spot performance regressions by tracking changes in test duration over time.
- **Test Suite Coverage:** You can calculate the percentage of test coverage for each suite. This can help you identify the areas that need more testing.
- **Flakiness Detection:** You can identify which tests occasionally fail. This helps you find unstable sections of code that need more attention.
- **Spotting Regressions:** Assume you are writing a module that requires a high level of stability and performance. You can track the module over time and spot any sudden drops in pass rates or spikes in test durations that could signal the introduction of bugs or performance issues.
- **Correlating with Project Milestones:** Annotate your trend charts with significant project events like major releases, framework upgrades, or architectural changes. This will help you understand how these milestones impact your test metrics.

Please note that these metrics are just examples. You need to specify your goals before defining and choosing metrics. You can also use visualization tools like Grafana, Kibana, or Tableau to create dashboards and track these metrics over time.

## 43.3 Conclusion

Throughout this chapter, we've emphasized the importance of generating, gathering, and analyzing test reports to gain valuable insights into our codebase and testing processes. We utilized ZIO Test's built-in functionality to generate test reports in JSON format, which helps us store and analyze test results over time.

We also discussed various metrics that can be derived from test results, such as pass/fail rates, test durations, and the detection of flaky tests.

By leveraging the power of test reports, you can transform raw test data into actionable insights, enabling your team to make data-driven decisions. This approach ultimately leads to more efficient development processes and higher-quality software products.

## Chapter 44

# 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 motivating 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 serve 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.

## 44.1 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:

```

1 import zio._
2
3 import java.net.URL
4
5 type Web = ????
6
7 def crawl[R, E](
8   seeds: Set[URL],
9   router: URL => Boolean,
10  processor: (URL, String) => ZIO[R, E, Unit]
11 ): ZIO[R with Web, Nothing, List[E]] =
12   ???

```

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 occurred, 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.

## 44.2 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:

```
1 | def getURL(url: URL): ZIO[Any, Throwable, String] =
2 | ???
```

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:

```
1 final class URL private (
2     private val parsed: java.net.URL
3 ) { self =>
4     override def equals(that: Any): Boolean =
5         that match {
6             case that: URL => this.parsed == that.parsed
7             case _              => false
8         }
9     override def hashCode: Int =
10        parsed.hashCode
11     def relative(page: String): Option[URL] =
12         try {
13             Some(new URL(new java.net.URL(parsed, page)))
14         } catch {
15             case t: VirtualMachineError => throw t
16             case _: Throwable          => None
17         }
18     override def toString: String =
19         url
20     def url: String =
21         parsed.toString
22 }
23
24 object URL {
25     def make(url: String): Option[URL] =
26         try {
27             Some(new URL(new java.net.URL(url)))
28         } catch {
29             case t: VirtualMachineError => throw t
30             case _: Throwable          => None
31         }
32 }
```

```

34 def extractURLs(root: URL, html: String): Set[URL] = {
35   val pattern = "href=[\"\\\"]([^\"]]+)[\"\\\"]".r
36
37   scala.util
38     .Try({
39       val matches =
40         (for {
41           m <- pattern.findAllMatchIn(html)
42         } yield m.group(1)).toSet
43
44       for {
45         m   <- matches
46         url <- URL.make(m) ++ root.relative(m)
47       } yield url
48     })
49     .getOrElse(Set.empty)
50 }

```

The next step is for us to put this signature into the format of the service pattern, so it is easy to compose with other services in the ZIO environment:

```

1 object web {
2
3   type Web = Web.Service
4
5   object Web {
6     trait Service {
7       def getURL(url: URL): ZIO[Any, Throwable, String]
8     }
9   }
10
11  def getURL(url: URL): ZIO[Web, Throwable, String] =
12    ZIO.serviceWithZIO(_.getURL(url))
13 }

```

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.

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` service 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` service, 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 service, so we have no way to actually satisfy the dependency of `crawl` on the Web service 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 service that will return HTML for a specified URL by actually retrieving the URL.

Following the service pattern, we know that we want to define each implementation of our service as a `ZLayer`. This will allow us to have our service potentially depend on other services and use effects and finalization logic if necessary, and will also allow users of our service to provide it in the same way as other services they are working with:

```
1 | lazy val liveLayer: ZLayer[Any, Nothing, Web] =
2 |     ???
```

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:

```
1 | val liveLayer: ZLayer[Any, Nothing, Web] =
2 |   ZLayer.succeed {
3 |     new Web.Service {
4 |       def getURL(url: URL): ZIO[Any, Throwable, String] =
5 |         ZIO.attempt {
6 |           scala.io.Source.fromURL(url.url).getLines.mkString
7 |         }
8 |       }
9 |     }
```

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.attempt` constructor because `Source.fromURL` can throw exceptions, for example, if the URL cannot be found, so we use the `attempt` 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 `ZIO.attemptBlockingIO` constructor to run poten-

tially 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 `liveLayer` version of our `Web` service we actually need a `Blocking` service.

So we update the signature of `liveLayer` like this:

```
1 | lazy val liveLayer: ZLayer[Any, Throwable, Web] =
2 | ???
```

To create a service that depends on another service, we can use the `ZLayer.succeed` constructor, so our `liveLayer` implementation now looks like this:

```
1 | val liveLayer: ZLayer[Any, Nothing, Web] =
2 |   ZLayer.succeed {
3 |     new Web.Service {
4 |       def getURL(url: URL): ZIO[Any, Throwable, String] =
5 |         ZIO.attemptBlockingIO {
6 |           scala.io.Source.fromURL(url.url).getLines.mkString
7 |         }
8 |     }
9 | }
```

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` service, 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!

### 44.3 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 `ZIO.foreachPar` operator defined by `ZIO`, so our first version might look something like this:

```

1 def crawl[R, E] (
2   seeds: Set[URL],
3   router: URL => Boolean,
4   processor: (URL, String) => ZIO[R, E, Unit]
5 ): ZIO[R with Web, Nothing, List[E]] = {
6   ZIO.foreachParDiscard(seeds) { url =>
7     web.getURL(url).flatMap { html =>
8       val urls = extractURLs(url, html)
9       processor(url, html).catchAll(e => ???) *>
10      crawl(urls.filter(router), router, processor)
11    }
12  }
13  ???
14 }
```

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 occurred, 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 of performing our algorithm. Let’s introduce a data type to capture the state that we want to maintain:

```

1 final case class CrawlState[+E] (
2   errors: List[E],
3   visited: Set[URL]
4 ) {
5   def visit(url: URL): CrawlState[E] =
```

```

6     copy(visited = visited + url)
7   def visitAll(urls: Iterable[URL]): CrawlState[E] =
8     copy(visited = visited ++ urls)
9   def logError[E1 >: E](error: E1): CrawlState[E1] =
10    copy(errors = error :: errors)
11  }
12
13 object CrawlState {
14   val empty: CrawlState[Nothing] =
15     CrawlState(List.empty, Set.empty)
16 }
```

We could try to pass the `CrawlState` around in a 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:

```

1 def crawl[R, E](
2   seeds: Set[URL],
3   router: URL => Boolean,
4   processor: (URL, String) => ZIO[R, E, Unit]
5 ): ZIO[R with Web, Nothing, List[E]] =
6   Ref.make[CrawlState[E]](CrawlState.empty).flatMap { ref =>
7     def loop(seeds: Set[URL]): ZIO[R with Web, Nothing, Unit] =
8       ZIO.foreachParDiscard(seeds.filter(router)) { url =>
9         ref.modify { crawlState =>
10           if (crawlState.visited.contains(url))
11             (ZIO.unit, crawlState)
12           else
13             (
14               getURL(url).flatMap { html =>
15                 processor(url, html).catchAll { e =>
16                   ref.update(_.logError(e))
17                   } *> loop(extractURLs(url, html))
18                 }.ignore,
19                 crawlState.visit(url)
20               )
21             }.flatten
22       }
23     loop(seeds) *> ref.get.map(_.errors)
}
```

24 }

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 ZIO.foreachParDiscard 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.

## 44.4 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.

```
1 lazy val testLayer: ZLayer[Any, Nothing, Web] =
2   ???
```

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.

```

1 val Home  = URL.make("http://zio.dev").get
2 val Index = URL.make("http://zio.dev/index.html").get
3 val ScaladocIndex =
4   URL.make("http://zio.dev/scaladoc/index.html").get
5 val About = URL.make("http://zio.dev/about").get
6
7 val SiteIndex =
8   Map(
9     Home          -> """<html><body><a href="index.html">Home</a>
10    <a href="/scaladoc/index.html">Scaladocs</a></body></html>
11    """,
12    Index         -> """<html><body><a href="index.html">Home</a>
13    <a href="/scaladoc/index.html">Scaladocs</a></body></html>
14    """,
15    ScaladocIndex -> """<html><body><a href="index.html">Home</a>
16    <a href="/about">About</a></body></html>""",
17    About          -> """<html><body><a href="home.html">Home</a><
18    a href="http://google.com">Google</a></body></html>"""
19  )

```

With this data, we can define a test version of the Web service like this:

```

1 val testLayer: ZLayer[Any, Nothing, Web] =
2   ZLayer.succeed {
3     new Web.Service {
4       def getURL(url: URL): ZIO[Any, Throwable, String] =
5         SiteIndex.get(url) match {
6           case Some(html) =>
7             ZIO.succeed(html)
8           case None =>
9             ZIO.fail(
10               new java.io.FileNotFoundException(url.toString)
11             )
12         }
13     }
14   }

```

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:

```
1 val testRouter: URL => Boolean =
2   _.url.contains("zio.dev")
```

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:

```
1 def testProcessor(
2   ref: Ref[Map[URL, String]]
3 ): (URL, String) => ZIO[Any, Nothing, Unit] =
4   (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:

```
1 import zio.test._
2 import zio.test.Assertion._
3
4 test("test site") {
5   for {
6     ref    <- Ref.make[Map[URL, String]](Map.empty)
7     _      <- crawl(Set(Home), testRouter, testProcessor(ref))
8     crawled <- ref.get
9   } yield assert(crawled)(equalTo(SiteIndex))
10 }.provideCustomLayer(testLayer)
```

If you try running this test, you will see that our parallel web crawler is indeed working with our test data!

## 44.5 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.

```

1 val seeds: Set[URL] =
2   Set(URL.make("https://www.nytimes.com").get)
3 // seeds: Set[URL] = Set(https://www.nytimes.com)
4
5 val router: URL => Boolean =
6   _.url.contains("https://www.nytimes.com")
7 // 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.

```

1 val processor: (URL, String) => ZIO[Any, Nothing, Unit] =
2   (url, _) => Console.readLine(url.url).orDie

```

You can then try running the web crawler like this:

```

1 object Example extends ZIOAppDefault {
2   val run =
3     for {
4       fiber <- crawl(seeds, router, processor)
5         .provideLayer(liveLayer)
6         .fork
7       _ <- Console.readLine.orDie
8       _ <- fiber.interrupt
9     } yield ExitCode.success
10 }

```

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 `ZIO.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 `ZIO.foreachParN` to limit the degree of parallelism, but that doesn't completely solve our problem because `ZIO.foreachParN` is called recursively. So even if we limit the original invocation to using, say, 100 fibers, each of those fibers would call `ZIO.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 `ZIO.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 reconceptualizing 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:

```
1 import web._
2
3 def crawl[R, E](
4   seeds: Set[URL],
5   router: URL => Boolean,
6   processor: (URL, String) => ZIO[R, E, Unit]
7 ): ZIO[R with Web, Nothing, List[E]] =
8   Ref.make[CrawlState[E]](CrawlState.empty).flatMap { crawlState
9     =>
10    Ref.make(0).flatMap { ref =>
11      Promise.make[Nothing, Unit].flatMap { promise =>
12        ZIO.acquireReleaseWith(Queue.unbounded[URL])(_.shutdown)
13        {
14          queue =>
15            val onDone: ZIO[Any, Nothing, Unit] =
16              ref.modify { n =>
17                if (n == 1)
18                  (
19                    queue.shutdown <* promise.succeed(()),
20                    0
21                  )
22                else (ZIO.unit, n - 1)
23              }.flatten
24            val worker: ZIO[R with Web, Nothing, Unit] =
25              queue.take.flatMap { url =>
26                web
27                  .getURL(url)
28                  .flatMap { html =>
29                    val urls =
30                      extractURLs(url, html).filter(router)
31                    for {
32                      urls <- crawlState.modify { state =>
33                        (
34                          urls -- state.visited,
35                          state.visitAll(urls)
36                        )
37                      }
38                    } -> processor(url, html).catchAll { e =>
39                      crawlState.update(_.logError(e))
40                    }
41                    -> queue.offerAll(urls)
42                    -> ref.update(_ + urls.size)
43                  } yield ()
44                }
45              .ignore <* onDone
46            }
47          }
48        }
49      }
50    }
51  }
```

```

45     for {
46       _ <- crawlState.update(_.visitAll(seeds))
47       _ <- ref.update(_ + seeds.size)
48       _ <- queue.offerAll(seeds)
49       _ <- ZIO.collectAll {
50         ZIO.replicate(100)(worker.forever.fork)
51       }
52       _ <- promise.await
53       state <- crawlState.get
54     } yield state.errors
55   }
56 }
57 }
58 }
```

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 `ZIO.acquireRelease` 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.

## 44.6 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 a 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 scoped 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.

# Appendix 1: 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. 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.

## 44.7 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.

```
1 trait Animal {  
2     def name: String  
3 }
```

Being a statically typed language, Scala prevents us from calling operators on values that are not of the appropriate type.

```
1 val baxter: Animal =  
2     new Animal {  
3         def name: String =  
4             "baxter"  
5     }  
  
1 println(baxter.name)
```

```

2 // okay
1 println(1.name)
2 // does not compile

```

This helps prevent a wide variety of bugs.

## 44.8 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.

```

1 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`:

```

1 def printName(animal: Animal): Unit =
2   println(animal.name)
3
4 val baxterCat: Cat =
5   new Cat {
6     def name: String =
7       "baxter"
8   }
9
10 printName(baxterCat)
11 // okay

```

In fact, we can always treat an instance of a type as an instance of one of its supertypes:

```

1 val baxterAnimal: Animal =
2   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:

```

1 | implicitly[Cat <:< Animal]
2 | // okay

1 | implicitly[Animal <:< Cat]
2 | // 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.

## 44.9 Any and Nothing

There are two values that have special significance within the Scala type system, `Any` and `Nothing`.

### 44.9.1 Any

\index{Any}

`Any` is a supertype of every type.

```

1 | implicitly[Animal <:< Any]
2 | // okay
3 |
4 | implicitly[Int <:< Any]
5 | // 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.

```
1 | 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:

```
1 | 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.

```
1 | 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:

```
1 | def run[E, A](zio: IO[E, A]): Either[E, A] =
2 |   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:

```
1 | trait ZIO[-R, +E, +A] {
2 |   def ensuring[R1 <: R](finalizer: URIO[R1, Any]): ZIO[R1, E, A]
3 | }
```

The `ZIO#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.

#### 44.9.2 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.

```

1 implicitly[Nothing <:< Cat]
2 // okay
3
4 implicitly[Nothing <:< Int]
5 // 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:

```

1 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:

```

1 trait ZIO[-R, +E, +A] {
2   def forever: ZIO[R, E, Nothing]
3 }
```

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!

```

1 def get[A](either: Either[Nothing, A]): A =
2   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.

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.

## 44.10 Product and Sum Types

We can build more complex types out of simpler ones using *product* and *sum* types.

### 44.10.1 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:

```

1 final case class User(name: String, age: Int)
2
3 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:

```

1 final case class Name(first: String, last: String)
2
3 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.

```

1 final case class Status(
2   interrupted: Boolean,
3   interrupting: Boolean
4 )

```

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:

```

1 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]`.

### 44.10.2 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`:

```

1 sealed trait Exit[+E, +A]
2
3 object Exit {
4     final case class Failure[+E](e: E) extends Exit[E, Nothing]
5     final case class Success[+A](a: A) extends Exit[Nothing, A]
6 }
7
8 val eitherStringInt: Either[String, Int] = Right(1)

```

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:

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

```

1 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:

```

1 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`.

### 44.10.3 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.

## 44.11 Intersection and Union Types

### 44.11.1 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:

```

1 trait Animal {
2   def name: String
3 }
4
5 trait Greeter {
6   def greet: Unit
7 }
8
9 final case class Cat(name: String) extends Animal with Greeter {
10   def greet: Unit =
11     println("meow")
12 }
```

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:

```

1 import zio._
2
3 val effect = for {
4   time <- Clock.nanoTime
5   _    <- Console.printLine(time.toString)
6 } 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 `printLine` 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.

### 44.11.2 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:

```

1 sealed trait PasswordError
2
3 sealed trait IOError          extends PasswordError
4 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`.

```

1 import zio._
2 import zio.Console._

3
4 def getInput: ZIO[Any, IOError, String] =
5   ???

6
7 def validateInput(
8   password: String
9 ): ZIO[Any, InvalidPassword, Unit] =
10  ???

11
12 def getAndValidateInput: ZIO[Any, PasswordError, Unit] =
13  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?

```

1 sealed trait PasswordError
2
3 case object IOError          extends PasswordError
4 case object InvalidPassword extends PasswordError
5 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:

```

1 sealed trait PasswordError
2
3 sealed trait NormalError extends PasswordError
4 case object SecurityError extends PasswordError
5
6 case object IOError extends NormalError
7 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.

## 44.12 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:

```

1 sealed trait List[+A]
2 // List is a type constructor

```

```

3
4 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:

```

1 import zio._
2
3 trait ChunkLike[Collection[_]] {
4   def toChunk[A](collection: Collection[A]): Chunk[A]
5   def fromChunk[A](chunk: Chunk[A]): Collection[A]
6 }

```

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:

```

1 object ListIsChunkLike extends ChunkLike[List] {
2   def toChunk[A](list: List[A]): Chunk[A] =

```

```
3     Chunk.fromIterable(list)
4 def fromChunk[A](chunk: Chunk[A]): List[A] =
5     chunk.toList
6 }
```

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.

## 44.13 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.

# Appendix 2: 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.

## 44.14 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:

```
1 trait Animal {  
2     val name: String  
3 }  
4  
5 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.

```
1 | implicitly[Cat <:< Animal]
```

So we know that `Cat` is a subtype of `Animal`. But what if we now have a collection of cats?

```
1 | 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:

```
1 | implicitly[Collection[Cat] <:< Collection[Animal]]  
2 | // does not compile
```

```
1 | implicitly[Collection[Animal] <:< Collection[Cat]]  
2 | // 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:

```
1 | def combine[A] (  
2 |   left: Collection[A],  
3 |   right: Collection[A]  
4 | ): Collection[A] =  
5 |   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?

```
1 | final case class Dog(name: String) extends Animal  
2 |  
3 | val cats: Collection[Cat] =
```

```

4   Collection(List(Cat("spots"), Cat("mittens")))
5
6 val dogs: Collection[Dog] =
7   Collection(List(Dog("fido"), Dog("rover")))
8
9 combine(cats, dogs)
10 // 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:

```

1 def widen[A, B >: A](collection: Collection[A]): Collection[B] =
2   Collection(collection.elements)
3
4 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?

## 44.15 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, a 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.

```

1 | lazy val cats: List[Cat] =
2 |   ???
3 |
4 | lazy val animals: List[Animal] =
5 |   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 A is a subtype of B. So again we can always safely widen these covariant types.

For example, if we have a generator (`Gen`) 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:

```

1 import zio._
2
3 trait Gen[-R, +A] {
4   def sample: ZIO[R, Nothing, A]
5   def contains(a: A): ZIO[R, Nothing, Boolean]
6 }
7 // 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:

```

1 import zio._
2
3 def contains(
4   a: Left[Int, Nothing]
5 ): ZIO[Random, Nothing, Boolean] =
6   ???

```

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:

```

1 def contains(
2   a: Either[Int, String]
3 ): ZIO[Random, Nothing, Boolean] =
4   ???

```

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:

```

1 trait Gen[-R, +A] {
2   def sample: ZIO[R, Nothing, A]
3   def contains[A1 >: A](a: A1): ZIO[R, Nothing, Boolean]
4 }

```

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.

```

1 final case class Collection[+A](elements: List[A]) { self =>
2   def ++[A1 >: A](that: Collection[A1]): Collection[A1] =
3     Collection(self.elements :: that.elements)
4 }

```

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 transforming 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?

## 44.16 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:

```

1 trait Fruit
2
3 trait Apple extends Fruit
4 trait Orange extends Fruit
5
6 trait Drink
7
8 trait FoodProcessor[-Ingredient] {
9   def process(ingredient: Ingredient): Drink
10 }
```

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 parameterized 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 FoodProcessor above knows how to consume ingredients and produce delicious drinks. Let's see how this would look:

```

1 trait Smoothie    extends Drink
2 trait OrangeJuice extends Drink
3
4 val juicer: FoodProcessor[Orange] = 
5   new FoodProcessor[Orange] {
6     def process(orange: Orange): Drink =
7       new OrangeJuice {}
8   }
9
10 val blender: FoodProcessor[Fruit] = 
11   new FoodProcessor[Fruit] {
12     def process(fruit: Fruit): Drink =
13       new Smoothie {}
14 }
```

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:

```
1 implicitly[Orange <:< Fruit]
```

But a `FoodProcessor[Fruit]` is actually a subtype of `FoodProcessor[Orange]`:

```
1 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.

```

1 val nanoTime: URIO[Clock, Long] =
2   Clock.nanoTime
3
4 def printLine(line: String): URIO[Console, Unit] =
5   Console.printLine(line).orDie
6
7 val printTime: URIO[Clock with Console, Unit] =
8   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:

```

1 trait FoodProcessor[-Ingredient] {
2   def process(ingredient: Ingredient): (Drink, Ingredient)
3 }
4 // 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:

```

1 trait ZIO[-R, +E, +A] {
2   def flatMap[R1 <: R, E1 >: E, B] (
3     f: A => ZIO[R1, E1, B]
4   ): ZIO[R1, E1, B] =
5     ???
6 }

```

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.

## 44.17 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.

```

1 trait Ref[A] {
2   def get: UIO[A]
3   def set(a: A): UIO[Unit]
4 }
```

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?

```

1 trait ZRef[-A, +B] {
2   def get: UIO[B]
3   def set(a: A): UIO[Unit]
4 }
```

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.

```

1 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.

## 44.18 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.

```
1 | 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`.

```
1 | final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {  
2 |   def flatMap[B](f: A => ZIO[R, E, B]): ZIO[R, E, B] =  
3 |     ???  
4 | }
```

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 position 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  $A \Rightarrow ZIO[R, E, B]$  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  $A \Rightarrow ZIO[R, E, B]$ . A function  $In \Rightarrow Out$  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:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {
2   def flatMap[R1 <: R, B](f: A => ZIO[R1, E, B]): ZIO[R, E, B] =
3     ???
4 }
```

R also appears in the return type of `flatMap` and here it is in contravariant 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  $A \Rightarrow ZIO[R, E, B]$  first:

1. E appears in the function  $A \Rightarrow ZIO[R, E, B]$  which is an input to this data type, so we mark -1 for contravariance
2. E appears in the output of the function  $A \Rightarrow ZIO[R, E, B]$  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:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {
2   def flatMap[R1 <: R, E1 >: E, B](
3     f: A => ZIO[R1, E1, B]
4   ): ZIO[R, E, B] =
5   ???
6 }
```

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 covariant and covariant gives us covariant.

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:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) {
2   def flatMap[R1 <: R, E1 >: E, B](
3     f: A => ZIO[R1, E1, B]
4   ): ZIO[R, E, B] =
5   ???
6 }
```

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:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =
2   >
3   def flatMap[R1 <: R, E1 >: E, B](
4     f: A => ZIO[R1, E1, B]
5   ): ZIO[R, E, B] =
6     ZIO(r => self.run(r).flatMap(a => f(a).run(r)))
7 }
```

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:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =
2   >
3   def flatMap[R1 <: R, E1 >: E, B](
4     f: A => ZIO[R1, E1, B]
5   ): ZIO[R1, E1, B] =
6     ZIO(r => self.run(r).flatMap(a => f(a).run(r)))
7 }
```

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:

```

1 final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =
2   >
3   def flatMap[R1, E1, B](
4     f: A => ZIO[R with R1, Any, B]
5   ) =
6     ZIO(r => self.run(r).flatMap(a => f(a).run(r)))
7 }
```

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.

## 44.19 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.

# Index

Acquire Release, 195  
Actor, 8  
Aff, 1  
Akka, 17  
    ActorSystem, 546  
    Akka Streams, 480  
    Alpakka, 503  
Akka Streams, 6  
Algebraic Data Types, 394  
Annotations  
    annotate, 635, 637  
    get, 635, 637  
Assertion  
    anything, 60, 554  
    approximatelyEquals, 555  
    contains, 555  
    containsString, 555  
    dies, 557  
    endsWith, 555  
    endsWithString, 555  
    equalsIgnoreCase, 555  
    equalTo, 57, 59, 555  
    exists, 555  
    fails, 60, 557  
    failsCause, 557  
    forall, 555  
    hasAt, 555  
    hasAtLeastOneOf, 556  
    hasAtMostOneOf, 556  
    hasField, 556  
    hasFirst, 555  
    hasIntersection, 556  
    hasKey, 556  
    hasKeys, 556  
        hasLast, 555  
        hasNoneOf, 556  
        hasOneOf, 556  
        hasSameElements, 59, 556  
        hasSameElementsDistinct, 556  
        hasSize, 555  
        hasSizeString, 555  
        hasSubset, 556  
        hasThrowableCause, 557  
        hasValues, 556  
        isCase, 556  
        isDistinct, 556  
        isEmpty, 555  
        isEmptyString, 555  
        isFailure, 557  
        isFalse, 554  
        isGreaterThan, 555  
        isGreaterThanOrEqualTo, 555  
        isInterrupted, 557  
        isLeft, 557  
        isLessThan, 555  
        isLessThanOrEqualTo, 555  
        isNegative, 555  
        isNone, 557  
        isNonEmpty, 555  
        isNonEmptyString, 555  
        isNull, 554  
        isOneOf, 555  
        isPositive, 555  
        isRight, 557  
        isSome, 557  
        isSorted, 558  
        isSubType, 556  
         isSuccess, 557

isTrue, 554  
    isUnit, 60, 554  
    isWithin, 555  
    isZero, 555  
    matchesRegex, 555  
    nonNegative, 555  
    nonPositive, 555  
    not, 60  
    nothing, 554  
    startsWith, 555  
    startsWithString, 555  
    succeeds, 557  
    throws, 557  
    throwsA, 557  
Asynchronous Callback, 87  
Asynchronous Computation, 88  
Asynchronous Execution Tracing, 8  
Asynchronous Queue, 5  
Asynchronous Side-effect, 2  
Asynchronous Thread Pool, 648  
Atomic Operation, 291  
AtomicInteger, 149  
atomicity, 328  
AtomicReference, 88, 145, 149, 193, 294,  
    295  
    compareAndSet, 295  
Auction System, 187  
AutoCloseable, 210  
Automatic Layer Construction, 241, 245  
Automatic ZLayer Derivation, 258  
  
    Back Pressure Strategy, 172  
Background Process, 107  
Blaze Server, 96, 98  
Blocking Executor, 87, 95  
Blocking Operation, 648  
Blocking Thread Pool, 94  
Bounded Hub, 181  
Bounded Queue, 170, 172  
Bounded Thread Pool, 94  
Bracket, 3  
Broadcasting, 178, 179  
Business Errors, 19  
Business Logic, 236  
By-name Parameter, 32, 87, 90, 544, 545  
    Caliban, 18  
Callback-based Asynchronous Code, 43  
Cartesian Product, 625  
Category Theory, 7, 96  
Cats Effect, 6, 18, 44, 83, 86, 90, 91, 98  
    cats-interop, 86  
    ConcurrentEffect, 97  
    Resource, 94, 95, 98  
    toScopedZIO, 94, 95, 98  
    Timer, 97  
    zio-interop-cats, 92, 94, 97  
Cause, 70, 394  
    Both, 75  
    Cause#defects, 80  
    Cause#prettyPrint, 77, 80  
Die, 70  
Fail, 70  
Interrupt, 104  
Interrupted, 140  
Then, 75  
Chatroom System, 187  
Checked Failure, 69  
Child Scope, 224  
Chunk, 307, 426  
Circuit Breaker, 177  
Circular Dependency, 385  
Clock, 44, 45, 565, 587  
    nanoTime, 45  
    sleep, 45  
Compare-and-swap (CAS), 150, 152,  
    294  
CompletableFuture, 86, 87  
CompletionStage, 83, 86, 87  
Composability, 54  
Composable Incremental Parser, 495  
Composable Resources, 202  
Composable Retry Strategies, 345  
Composing Automatic Updates, 296  
Concurrency, 17, 592  
Concurrent Data Structure, 7, 178  
Concurrent Streaming, 2  
Console, 44, 46  
    printLine, 46, 569  
    printLineErr, 570  
    readLine, 569  
    readline, 46

Constructor-based Dependency Injection, 246  
Contextual Data Type, 124  
Contravariance, 31  
Contravariant Reader, 6  
Contravariant Type, 590  
Cooperatively-yielding Virtual Thread, 3  
CountDownLatch, 305  
Counter Metric, 526  
Covariance, 5, 31, 676  
Covariant Type, 590  
Cron Job, 357, 376  
  
Daemon, 107  
Dangling Fiber, 122  
Data-driven Application, 403  
Database Transaction, 235  
Deadlock, 17, 191, 304  
deadlock, 383  
Debug Logging, 381  
Debugging, 334, 379  
Decision, 376  
Declaration-site Variance, 4, 674  
Declarative Programming, 203, 238  
Default Services, 23  
DefaultJvmMetrics, 537  
DefaultRunnableSpec, 57  
Defect, 69, 391  
Defect Handling, 71  
Defensive Programming, 33, 37  
Deferred Evaluation, 25, 42  
Deferring Error Handling, 32  
Degree of Concurrency, 190  
Degree of Parallelism, 114, 191, 656  
Dependency Graph, 10, 246  
Dependency Injection, 38, 231, 251, 652  
Dequeue, 179, 180  
Deterministic Random Number Generator, 611  
Domain Error, 391  
Domain-driven Design, 390  
Doobie, 6, 86, 92, 95, 98  
    ConnectionIO, 92, 93  
    doobie-hikari, 94  
    Transactor, 83, 93  
    fromDriverManager, 93  
Dotty, 9  
Double Spending, 303  
Doubly Back-pressured Queue, 4, 7  
Dropping Hub, 184  
Dropping Queue, 170  
Dropping Strategy, 173  
Dynamic Scope, 206, 207  
Dynamically-typed Error, 3  
  
Eager Evaluation, 87  
Efficiency, 17  
Elasticity, 1  
Enqueue, 180, 185  
ensuring, 196, 197  
Environment Type, 23  
Environment Variable, 46  
Error Channel, 3, 24, 32  
Error Handling, 4, 17, 32  
Error Management, 390  
Error Tracking, 4  
Error Type, 23  
Event-driven Programming, 1  
Execution Flows, 379  
Execution Tracing, 17, 77  
ExecutionContext, 36, 94, 98  
Executor, 98, 100, 105, 108  
Exit, 70, 80, 140, 199, 209, 211  
Expected Error, 69, 391  
Explicit Fiber Lifetime, 213  
Exponential Backoff Strategy, 353  
  
Fallback Operator, 75  
Fatal Error, 391, 398  
Fiber, 3, 100, 119  
    await, 103  
    Fiber Cancellation, 17  
    Fiber Dump, 593  
    Fiber Failure, 69  
    Fiber Interruption, 11, 56, 104, 130  
    Fiber Lifetime, 213  
    Fiber Model, 100  
    Fiber Supervision, 106  
    Fiber Yielding, 101  
    Fiber-based Concurrency, 8  
    interrupt, 104

join, 102, 108  
Locking Effect, 108  
poll, 104  
Fiber Dump, 383  
Fiber Supervision, 119, 385  
FiberRef, 6, 9, 145, 154  
    locally, 156  
FIFO, 170  
Finalization, 200  
Finalizer, 105, 198  
finalizer, 197  
Fine-grained Interruption, 18  
Fine-grained Locking, 337  
Fire-and-forget, 128  
First-class Values, 57  
Fixed Scope, 206  
For Comprehensions, 28  
Fork/Join Identity Law, 119, 137  
FS2, 2, 6, 480  
Functional Composition, 19  
Functional Effect, 23, 36  
Future, 23, 35, 44, 86, 87, 197, 346, 543  
    CompleteableFuture, 88  
    fallbackTo, 37  
    ListenableFuture, 88  
    onComplete, 37  
  
Gen, 64  
    asciiString, 65, 613  
    chunkOf, 617  
    const, 614  
    cross, 616  
    crossWith, 616  
    double, 613  
    either, 618, 619  
    elements, 619, 623  
    flatMap, 612, 615, 624  
    fromIterable, 623  
    int, 65, 612  
    listOf, 617  
    listOf1, 617  
    map, 612, 614  
    mapOf, 617  
    mapOfN, 617  
    mapZIO, 615  
    noShrink, 628, 629  
oneOf, 612–614, 619  
option, 618  
reshrink, 628  
runCollectN, 621  
sample, 621  
setOf, 617  
setOfN, 617  
weighted, 614, 619  
zip, 617  
zipWith, 617  
Global Scope, 107, 123  
Goto Statement, 120  
GraphQL, 17  
Guava, 86, 88  
    zio-interop-guava, 88  
  
Happy Path Programming, 32  
Has, 10  
Haskell Hedgehog, 10  
Haskell IO, 3  
Haskell iteratees, 6  
haXe, 1  
Heisenbug, 335  
High Contention, 303, 304  
High-contention Sections, 341  
Higher-kinded Types, 7, 91, 673  
Hikari Connection Pool, 94  
HikariTransactor, 94  
http4s, 6, 86, 96  
    HttpRoutes, 96  
Hub, 178, 502, 503  
    awaitshutdown, 185  
    bounded, 181  
    capacity, 185  
    dropping, 184  
    isShutdown, 185  
    publish, 179, 184  
    shutdown, 185  
    size, 185  
    sliding, 182  
    subscribe, 179, 185  
    toQueue, 185  
    unbounded, 183  
  
Idempotent Operation, 303  
Imperative Programming, 203

Implicit Evidence, 662  
Implicit Syntax, 91  
Implicits, 7  
Interoperability, 86  
Interruptible Acquire, 208  
Interruptible Combinator, 105  
Interruptible Region, 135  
InterruptStatusRestore, 138  
Intersection Types, 231, 668  
Invariant Type, 675  
Iterable, 408  
Izumi Reflect, 10  
  
Java Database Connectivity (JDBC), 92  
Javascript Integration, 89  
JCStress, 592  
JVM Metrics, 537  
  
Kafka, 496  
  
lazy conflict detection, 336  
Lazy Evaluation, 32, 87, 90  
Legacy Code, 38  
Lexical Scope, 120, 206  
Live, 45  
    live, 576, 577  
    withLive, 576  
Load Balancer, 176  
Local Scope, 123  
Local State, 235  
Locally Scoped Changes, 157  
Logging Defects, 392, 396  
Long-running Application, 195  
Loose Coupling, 236  
  
Memoization, 255  
Memoized Resource, 227  
Memoizing, 163  
Memory Leakage, 172, 184, 199  
Metric, 526  
    contramap, 529  
    Counter, 528  
    counter, 526  
    Frequency, 532  
    fromConst, 529  
    Gauge, 529  
  
Histogram, 529  
Summary, 531  
value, 527  
Metric Registry, 532  
MetricKey, 528  
MetricsClient, 528  
Microservice, 17  
Modeling Capabilities, 668  
Modeling Domain Errors, 394  
Modular Concurrency, 334  
Monad Transformer, 3  
Monix, 18, 44  
Mutable Array, 307  
Mutable Map, 309  
Mutable Priority Queue, 311  
Mutable Set, 320  
Mutable State, 8, 145  
MVar, 7  
  
Nexus of Communication, 185  
Non-blocking Executor, 95  
Non-deterministic Random Generator, 45  
Non-fatal Exception, 54  
Non-recoverable Error, 391, 398  
NonEmptyChunk, 116  
NonEmptyList, 116  
Nothing, 664  
  
Observability, 526  
OffsetDateTime, 376  
Onion Architecture, 236, 237, 239  
Operation Lifespan, 11  
Optimistic Concurrency, 304, 336  
  
Parallelism, 5  
Parsing, 426  
Persistence, 17  
Polymorphic Error Type, 37  
Pre-interrupted Effect, 131  
Premature Evaluation, 25  
PriorityQueue, 322  
Procedural Code, 46  
Procedural Programming, 24  
Product Types, 618, 665  
Profiling, 387

Programs as Values, 25  
Project Loom, 2  
Prometheus, 526  
Prometheus Client, 534  
Promise, 1, 7, 88–90, 160, 312, 658  
    await, 161, 163  
    complete, 162, 167  
    completeWith, 162, 167  
    die, 162  
    done, 162  
    fail, 161  
    failCause, 162  
    interrupt, 164  
    interruptAs, 164  
    isDone, 164  
    Javascript Promise, 86, 89  
    poll, 164  
    succeed, 161  
Property-based Testing, 64, 572, 609  
Pseudo Random Number Generator, 45,  
    621  
Pure Code, 40  
Pure Expressions, 39  
Pure Functions, 39  
Purescript, 1  
  
Queue, 7, 88, 169, 304, 503, 658  
    awaitShutdown, 176  
    bounded, 172  
    capacity, 175  
    isShutdown, 176  
    offer, 170, 172–174  
    offerAll, 175  
    poll, 175  
    shutdown, 176  
    size, 176  
    take, 170, 173, 174  
    takeAll, 175  
    takeBetween, 175  
    takeUpTo, 175  
    unbounded, 170  
QuickCheck, 622  
  
Raincheck, 1  
Random, 44, 47, 572  
Random Number Generator, 572  
Rate Limiter, 177  
Rate Limiting, 345  
Reactive Programming, 1, 18  
Recoverable Error, 390  
Recursion, 47, 346  
red-black tree, 326  
Ref, 7, 8, 88, 145, 148, 151, 157, 291, 294,  
    651  
    get, 292  
    make, 146, 166  
    modify, 150, 152, 167, 292, 293,  
        579, 652  
    set, 292  
    Synchronized, 145, 152, 153, 157  
        update, 151, 292, 295, 296  
RefCashe, 153  
Referential Transparency, 39, 40, 145  
Regional Locking, 109  
Reification, 204  
Resiliency, 1  
Resource Acquisition, 198  
Resource Finalization, 4  
Resource Leakage, 105, 195  
Resource Lifecycle, 197  
Resource Lifetime, 206, 207, 212  
Resource Management, 124  
Resource Safety, 4, 5, 17, 105, 655  
Responsive Design, 1  
REST API, 17  
Retries, 345  
Retry Strategy, 345, 347  
Ring Buffer, 5  
RingBuffer, 321  
Running Computation, 100  
Running Effect, 36  
Runtime, 56, 379  
    addFatal, 399  
    addSupervisor, 389  
    default, 85  
    enableOpSupervision, 387  
    enableRuntimeMetrics, 537  
    maximumYieldOpCount, 101  
    setReportFatal, 401  
    unsafe  
        fromLayer, 85  
    unsafe.run, 84, 93, 100

Safe Interruption, 5  
Safe Resource Handling, 195, 198, 543  
Saga Pattern, 303  
Sample  
    flatMap, 628  
    map, 627  
    noShrink, 629  
    shrinkFractional, 629  
    shrinkIntegral, 629  
Sandboxing, 397  
Scala 3, 77, 231, 593, 671  
Scala JS, 9  
Scala.js, 86, 541  
ScalaCheck, 4, 10, 622  
ScalaTest, 55, 56, 541, 542  
Scalaz, 1, 3, 4  
    IO, 7  
    Task, 3  
Schedule, 5, 254, 347, 375, 568  
    \*>, 371  
    ++, 372  
    <\*, 371  
        369  
        &&, 368  
        addDelay, 363  
        andThen, 372  
        andThenEither, 371  
        as, 359  
        check, 365  
        collectAll, 355, 360  
        collectUntil, 356  
        collectWhile, 356  
        Compositional Scheduling, 5  
        Continue Logic, 368  
        contramap, 359  
        count, 356  
        dayOfWeek, 357  
        Decision, 361  
        delayed, 363, 364  
        dimap, 359  
        Driver, 376  
        elapsed, 357  
        ensuring, 366  
        exponential, 352  
        fibonacci, 353  
    fixed, 350  
    fold, 360  
    foldZIO, 360  
    forever, 366  
    fromDuration, 353  
    fromDurations, 353  
    fromFunction, 355  
    Function Composition, 374  
    Geometric Intersection, 369  
    Geometric Union, 369  
    hourOfDay, 357  
    identity, 355  
    Internal State, 366  
    intersectWith, 367, 368  
    jittered, 363, 364  
    linear, 352, 366  
    map, 359  
    mapZIO, 359  
    minuteOfHour, 357  
    modifyDelay, 363  
    once, 350  
    onDecision, 361  
    provide, 361  
    provideSome, 361  
    reconsider, 365  
    recurs, 348, 350  
    recurUntil, 356  
    recurUntil\*, 355  
    recurWhile, 356  
    recurWhile\*, 354  
    repeat, 5  
    repetitions, 360  
    resetAfter, 366  
    resetWhen, 366  
    retry, 5  
    Schedule State, 348  
    secondOfMinute, 357  
    spaced, 348, 350, 496  
    stop, 350  
    succeed, 356  
    Summary Value, 348  
    tapInput, 361  
    tapOutput, 361  
    unfold, 356  
    unionWith, 367, 368  
    windowed, 351

zipWith, 370  
Scope, 95, 98, 124, 180, 195, 202, 204,  
    218, 232, 235, 437, 543  
    addFinalizer, 219  
    Closeable#use, 223  
    Dynamic Scope, 444  
    extend, 223  
    Static Scope, 443  
Scoped Resource, 207  
Semantic Blocking, 157, 172, 321  
Semaphore, 7, 88, 114, 153, 189, 304,  
    319, 656  
    withPermits, 189  
Sequential Composition, 27  
Service Pattern, 648  
Shared State, 145  
Side Effect, 24, 39, 40  
Sink, 423, 490  
Sized Service, 65  
Sliding Hub, 182  
Sliding Queue, 170  
Sliding Strategy, 173  
Slow Consumer Problem, 182  
SmallCheck, 622  
Software Transactional Memory, 328  
Software Transactional Memory (STM),  
    8, 151, 291, 306  
SortedMap, 322  
Space complexity, 341  
Specs2, 9  
Stack Overflow, 47  
Stack safety, 8  
Stack Trace, 8, 382  
Stack-safety, 47  
Stacked Errors, 78  
Standard Services, 563  
Static Scope, 120, 206  
Statically-typed Error, 3  
StatsD, 526  
StatsD Client, 537  
STM, 328  
    atomically, 328  
    check, 342  
    commit, 328  
    orElse, 342  
    orTry, 343  
repeatUntil, 301  
retry, 294, 301, 313, 342  
retryUntil, 342  
retryWhile, 342  
TMap, 343  
STM Optimization, 336  
Streaming Broadcast Operation, 186  
Structured Concurrency, 11, 120, 213  
Structured Programming, 120  
Subtyping, 4  
Success Channel, 24  
Sum Types, 618, 667  
Summary Value, 490  
Supertype Composition, 76  
Supervision Strategy, 123  
Supervisor, 385  
synchronized, 294  
Synchronized Block, 192  
System, 44, 46, 564, 575  
    env, 46  
    property, 46  
System Property, 46  
Tagless Final, 7, 18, 90  
TArray, 306, 307  
    apply, 308  
    fromIterable, 308  
    make, 308  
    size, 308  
    transform, 309  
    update, 308  
Task, 91  
TestAnnotation  
    ignored, 598, 635  
    repeated, 598, 633, 635, 637  
    retried, 598, 635  
    tagged, 634  
    timing, 598, 635  
TestAnnotationMap  
    annotate, 633  
    get, 634  
TestArrow  
    >>, 554  
    andThen, 554  
TestAspect, 63  
    after, 589, 590, 637

afterAll, 590  
annotate, 598  
around, 590  
aroundAll, 590  
aroundAllWith, 590  
aroundWith, 590  
before, 590  
beforeAll, 590  
debug, 571  
diagnose, 593  
dotty, 594  
dottyOnly, 594  
eventually, 592  
exceptDotty, 594  
exceptJS, 594  
exceptJVM, 594  
exceptNative, 594  
exceptScala2, 594  
exceptScala211, 594  
exceptScala212, 594  
exceptScala213, 594  
failing, 63, 596  
flaky, 592  
ignore, 583, 586, 592  
js, 594  
jsOnly, 594  
jvm, 594  
jvmOnly, 585, 594  
native, 594  
nativeOnly, 594  
nonFlaky, 63, 583–585, 588, 592,  
    598, 637  
nonTermination, 597  
parallel, 596  
repeat, 592  
retry, 590, 592  
scala2, 594  
scala211, 594  
scala211Only, 594  
scala212, 594  
scala212Only, 594  
scala213, 594  
scala213Only, 594  
scala2Only, 594  
sequential, 595  
silent, 571  
tagged, 597  
timed, 597  
timeout, 63, 585, 587, 596, 597  
withLiveEnvironment, 595  
TestAspect.withLiveClock, 595  
TestAspect.withLiveConsole, 595  
TestAspect.withLiveRandom, 595  
TestAspect.withLiveSystem, 595  
TestClock, 61, 568  
    adjust, 63, 565  
    getTimeZone, 566  
    setDateTime, 565  
    setTime, 565  
    sleeps, 566  
    timeZone, 566  
TestConsole, 61  
    clearInput, 570  
    clearOutput, 570  
    debug, 572  
    feedLines, 570  
    outputErr, 570  
TestRandom, 45, 61  
    getSeed, 573  
    setSeed, 573  
TestSystem, 61  
    clearEnv, 575  
    clearProperty, 575  
    env, 575  
    property, 575  
    putEnv, 564, 575  
    putProperty, 575  
Thread, 100, 105  
    interrupt, 105  
Thread Pool, 100  
Thread Starvation, 648  
ThreadLocal, 6, 9, 154  
Tight Coupling, 236  
Timing-out, 11  
TMap, 306, 307, 309  
    delete, 310  
    empty, 310  
    fromIterable, 310  
    get, 310  
    make, 310  
    put, 310  
TPriorityQueue, 306, 311, 313, 322

empty, 311  
fromIterable, 311  
make, 311  
offer, 311  
offerAll, 311  
peek, 312  
peekOption, 312  
removelf, 312  
retainIf, 312  
size, 312  
take, 311  
takeAll, 311  
takeOption, 312  
takeUpTo, 311  
toChunk, 312  
toList, 312  
TPromise, 306, 312  
    make, 313  
TQueue, 306, 313  
    bounded, 315  
    isEmpty, 315  
    isFull, 315  
    offer, 315  
    peek, 315  
    poll, 315  
    unbounded, 315  
Transactional Boundary, 336  
Transactional Journal, 328  
Transactional Memory, 7  
Transactional Variable, 297  
Transactor, 95  
Transient Failures, 19  
TReentrantLock, 306, 315  
TRef, 296, 306, 328  
Troubleshooting, 17  
Try, 3  
try/finally, 3, 197, 198  
TSemaphore, 306, 316, 319  
TSet, 306, 320  
Type Class, 3, 7, 18, 90, 91  
Type Constructor, 671  
Type Inference, 4, 31  
Type-indexed Heterogeneous Map, 10  
Type-level Set (HSet), 77  
Typed Errors, 79  
Typed Failure, 69  
Typelevel Cats, 4  
Unanticipated Failure, 71  
Unbounded Hub, 183  
Unbounded Queue, 170  
Unbounded Queues, 171  
Unchecked Failure, 69  
Unexpected Error, 69, 391  
Uninterruptible, 198  
Uninterruptible Acquire, 208  
Uninterruptible Combinator, 105  
Uninterruptible Region, 135, 207  
Union Types, 77, 668, 669  
Unknown Failure, 71  
Unrecoverable Failure, 71  
unsafeRun, 56  
Untyped Failure, 69  
Validation Errors, 115  
Variable-length Encoding, 426  
Variance, 674  
Virtual Threads, 17  
Web Crawler, 643  
widen, 676  
Work Distribution, 169, 178  
Work Limiting, 189  
Work Synchronization, 160  
ZChannel, 430, 481  
    concatMap, 440, 441, 444  
    ensuring, 442, 443  
    flatMap, 441, 443  
    foldChannel, 437  
    fromZIO, 437  
    MergeDecision, 442  
    mergeWith, 441, 444  
    pipeTo, 439, 444, 478, 493  
    pipeToOrFail, 478, 493  
    readWith, 435, 439  
    readWithCause, 482, 484, 487  
    scoped, 437  
ZEnv, 563  
ZEnvironment, 231, 251  
ZIO  
    \*>, 30

.flatMap, 34  
.map, 34  
<\*, 30  
acquireRelease, 105, 195, 206, 207,  
    212, 220, 543, 658  
acquireReleaseExit, 208  
acquireReleaseExitWith, 198, 199  
acquireReleaseInterruptible, 208  
acquireReleaseInterruptibleExit,  
    208  
acquireReleaseWith, 197, 202, 203,  
    206  
async, 43, 210  
asyncInterrupt, 133, 134  
attempt, 26, 32, 38, 42, 46, 54, 55,  
    91  
attemptBlockingCancelable, 133  
attemptBlockingInterrupt, 133,  
    134  
blocking, 210  
blockingExecutor, 95  
catchAll, 55, 346, 395, 396  
catchAllCause, 398  
catchAllDefect, 392, 394  
catchSome, 395  
collectAll, 30, 113  
collectAllPar, 112, 113  
collectAllParDiscard, 113  
collectAllParN, 113  
collectAllParNDiscard, 114  
debug, 381  
delay, 26  
disconnect, 142  
ensuring, 199, 366, 443, 663  
environment, 34, 166, 233, 234  
Error Channel, 31  
fail, 41  
flatMap, 27, 32, 91  
foldCauseZIO, 545  
foldZIO, 33, 71, 81  
foreach, 30, 113, 238  
foreachPar, 112, 113, 115, 191, 650,  
    655  
foreachParDiscard, 113, 191  
foreachParN, 113, 192, 656  
foreachParNDiscard, 114  
fork, 101, 108  
forkDaemon, 107, 108  
forkDeamon, 123  
forkIn, 127  
forkScoped, 123, 124, 213  
fromAutoCloseable, 210  
fromCompletableFuture, 87  
fromCompletionStage, 87  
fromEither, 41  
fromFuture, 44  
fromFutureJava, 86, 87  
fromListenableFuture, 88  
fromOption, 41  
fromPromiseJS, 90  
fromTry, 41  
getState, 235  
interruptible, 135, 137, 139  
interruptibleMask, 138  
IO, 35  
lock, 108  
logDebug, 381  
logErrorCause, 393  
logTrace, 381  
makeSome, 246  
mapError, 77  
mapErrorCause, 398  
metrics, 528  
never, 98  
on, 108  
orDie, 72  
orElse, 75  
parallelFinalizers, 206, 211  
provide, 34, 245, 251, 252  
provideEnvironment, 221, 234,  
    252, 253  
provideSome, 246, 258  
provideSomeEnvironment, 223  
raceEither, 111, 112  
raceWith, 442  
refailCause, 393  
refineOrDie, 81  
refineToOrDie, 73, 81  
refineWith, 73  
repeat, 349  
retry, 253, 349

retryOrElse, 349  
retryOrElseEither, 349  
RIO, 35  
runtime, 95, 97  
sandbox, 81, 397, 398  
scoped, 98, 124, 204, 207, 211, 235,  
    258  
scopeWith, 219  
service, 220, 233, 252  
serviceAt, 253  
serviceWith, 234  
serviceWithZIO, 234  
stateful, 235  
succeed, 41, 42, 57, 91  
Success Channel, 31  
supervised, 385, 386  
suspendSucceed, 544, 545  
tapCause, 78  
tapDefect, 394  
Task, 35  
timeout, 543, 577  
toPromiseJS, 89  
toPromiseJSWith, 89  
Type Aliases, 35  
UIO, 35  
uninterruptible, 135–137, 221, 222  
uninterruptibleMask, 138, 139  
unsandbox, 398  
updateState, 235  
URIO, 35  
validatePar, 115  
withEarlyRelease, 224  
withEarlyReleaseExit, 211  
withFinalizer, 209  
withFinalizerAuto, 210  
ZIO Environment, 6, 10, 31, 34,  
    223, 231, 563  
ZIOAppDefault, 26  
zip, 30, 205  
zipLeft, 30  
zipPar, 111–113, 205  
zipRight, 30  
zipWith, 29  
ZIO Config, 18  
ZIO Connect, 503  
ZIO Environment, 251  
ZIO gRPC, 18  
ZIO HTTP, 236, 239  
ZIO Interoperability, 83  
ZIO JSON, 428, 481  
ZIO Logging, 382  
ZIO Prelude, 116  
ZIO Redis, 18  
ZIO Runtime, 11, 26, 84, 97, 100, 105,  
    130, 399  
ZIO Runtime Metrics, 537  
ZIO Saga, 303  
ZIO Schema, 428, 481  
ZIO Services, 45  
ZIO STM, 8  
ZIO Stream, 6, 377, 403, 476  
    Aggregation Strategy, 496, 500  
    Asynchronous Aggregation, 495  
    Broadcasting, 502  
    Channel Scopes, 443  
    Concurrent Composition, 501  
    Concurrent Merge, 434  
    Decoder, 481  
    Early Termination, 412  
    Effectful Iterator, 405  
    Effectful Stream, 403  
    Encoder, 481  
    Finite Stream, 412  
    First-class Value, 420  
    Implicit Chunking, 405  
    Incremental Output, 431  
    Incremental Results, 434  
    Infinite Stream, 412  
    Leftover, 491  
    Parallel Composition, 501  
    Pipeline, 427, 476, 478  
    Potentially Infinite Stream, 407  
    Pull-based Streaming, 478, 493  
    Reactive Streams, 432  
    Stream Transformation, 427, 476  
    Stream Transformer, 476  
    Summary Value, 433, 493  
    Tapping Streams, 496  
    Terminal Value, 432  
    Transducing Streams, 494  
ZPipeline, 420, 427, 430, 433, 476  
ZSink, 420, 423, 424, 430, 433

ZStream, 403, 430, 432  
ZIO Test, 9, 45, 57, 377, 540, 575  
    Annotation, 597, 631  
    assert, 57, 58, 65, 558  
    assertCompletes, 63  
    Assertion, 59, 544, 548, 558  
    assertion, 544, 545  
    assertTrue, 558  
    assertZIO, 58, 65, 558  
    Auto-shrinking, 10  
    BoolAlgebra, 543  
    check, 64, 65, 609  
    checkAll, 65, 623  
    checkN, 65  
    checkZIO, 65, 609  
    Counterexample, 625  
    Deadlock, 593  
    Deterministic Finite Generator,  
        623  
    Deterministic Generator, 621  
    Deterministic Property-based Test-  
        ing, 621  
    Environment Variables, 575  
    FailureDetails, 544  
    Flaky Test, 592  
    Gen, 609, 611  
    Integrated Shrinking, 626  
    JUnit XML Report, 640  
    Lens, 561  
    Live Environment, 563, 575  
    Live Implementation, 564  
    Live Service, 575  
    Nested Tests, 546  
    NonFlaky Test, 592  
    Passage of Time, 565  
    Property-based Test, 588  
    Property-based Testing, 606  
    Random Generator, 621  
    Random Infinite Generator, 623  
    Random Number Generator, 611  
    Random Property-based Testing,  
        621  
    Reporting, 639  
    Resource Management, 600  
    Runtime Failure, 545  
    Safe Interruption, 543  
    Safe Resource Usage, 543  
    Sample, 611, 626  
    Scope, 546  
    Shared State, 565  
    Shrinking, 625  
    Smart Assertion, 558  
    System Properties, 575  
    Tagging, 597, 631  
    Test, 546  
    test, 65, 544  
    Test Arrow, 550  
    Test as Effect, 540  
    Test Case Generator, 608, 612  
    Test Case Shrinking, 611  
    Test Environment, 563, 575  
    Test Fixtures, 543  
    Test Generator, 65  
    Test Implementation, 563, 564  
    Test Runner, 640  
    Test Spec, 57  
    Test Suite, 57, 546  
    Testability, 17  
    TestAnnotation, 598, 633  
    TestAnnotationMap, 633  
    TestAnnotationRenderer, 637  
    TestArrow, 558  
    TestAspect, 583, 586, 588  
    TestClock, 564, 565, 567, 568  
    TestConfig, 636  
    TestConsole, 569, 572  
    TestEnvironment, 563  
    TestFailure, 540  
    TestFailure.Assertion, 545  
    TestFailure.Runtime, 545  
    TestLens, 561  
    TestRandom, 572, 574  
    TestResult, 544, 552  
    TestSuccess, 540  
    TestSystem, 564  
    ZIOSpec, 604  
    ZIOSpecDefault, 604  
    ZSPec, 546  
    ZSpec, 546, 586, 587  
    ZTest, 543–545  
    ZIO TestResult  
        Assertion, 552

ZIO.acquireRelease, 366  
ZIOAspect, 527  
ZLayer, 10, 241, 543, 648  
  apply, 242  
  Debug.mermaid, 246  
  Debug.tree, 246  
  derive, 259  
  fresh, 256  
  fromFunction, 241  
  make, 245  
  memoize, 257  
  orElse, 254, 255  
  scoped, 242  
ZManaged, 5  
ZPipeline  
  «», 478  
  filter, 483  
  map, 481  
  mapChunks, 423  
  rechunk, 486  
  utfDecode, 429  
ZSink, 490  
  collectAll, 423  
  collectAllN, 425  
  collectAllWhile, 492  
  contramap, 426, 498, 499  
  contramapChunks, 422  
  contramapZIO, 498, 499  
  count, 502, 503  
  flatMap, 426, 427, 500, 501  
  foldLeft, 502, 503  
  foldSink, 427  
  foldWeighted, 502  
  foreach, 426  
  fromFile, 423, 425, 502  
  fromFileName, 423  
  fromHub, 425, 503  
  fromPush, 426  
  fromQueue, 425, 503  
  head, 500, 502  
  Leftover, 421, 425  
  map, 426, 498  
  mapChunks, 426  
  mapLeftover, 498, 499  
  mapZIO, 426, 498, 500  
  race, 501  
    sum, 502, 503  
    take, 493, 495, 498, 502  
    transduce, 427  
    unwrapScoped, 426  
    zipPar, 423, 427, 501  
    zipWithPar, 503  
ZSpec  
  annotate, 634  
  filterAnnotations, 598, 635  
  filterTags, 597, 635  
  provideLayerShared, 603  
ZState, 235  
ZSTM, 296  
  commit, 296, 299  
  flatMap, 299  
  foldSTM, 299  
  foreach, 299  
  map, 299  
  retry, 300  
  zipWith, 299  
ZStream  
  «», 465  
  aggregateAsyncWithin, 496, 498, 504  
  apply, 419  
  async, 44  
  broadcast, 457, 458  
  broadcastDynamic, 458  
  broadcastedQueues, 458  
  broadcastedQueuesDynamic, 458  
  buffer, 460, 461  
  bufferChunks, 461  
  bufferChunksDropping, 461  
  bufferChunksSliding, 461  
  bufferDropping, 460, 461  
  bufferSliding, 460, 461  
  bufferUnbounded, 460, 461  
  catchAll, 417, 470  
  catchAllCause, 470  
  catchSome, 417, 470  
  catchSomeCause, 470  
  changes, 451  
  changesWith, 451  
  changesZIO, 451  
  collect, 407, 451, 452  
  collectLeft, 452

collectRight, 452  
collectSome, 452  
collectSuccess, 452  
collectWhile, 407, 452  
collectWhileLeft, 452  
collectWhileRight, 452  
collectWhileSome, 452  
collectWhileSuccess, 452  
combine, 471  
concat, 407, 465  
concatAll, 466  
cross, 469  
crossLeft, 470  
crossRight, 470  
debounce, 461  
distributeWith, 458  
distributeWithDynamic, 459  
drop, 407, 451  
dropRight, 451  
dropUntil, 407, 451  
dropWhile, 407, 451  
dropWhileZIO, 451  
environment, 416, 417  
environmentWith, 417  
environmentWithStream, 417  
environmentWithZIO, 417  
filter, 405, 407, 451  
filterNot, 407, 451  
filterZIO, 451  
find, 451  
findZIO, 451  
flatMap, 407, 449, 450  
flatMapPar, 450  
flatMapParSwitch, 450  
flatten, 450  
flattenChunks, 451  
flattenExit, 451  
flattenExitOption, 451  
flattenIterable, 451  
flattenPar, 450  
flattenParUnbounded, 450  
flattenTake, 451  
foldWhile, 412  
foldWhileScoped, 413  
foldWhileZIO, 413  
foreach, 413  
fromFileName, 423  
fromIterable, 408  
fromZIO, 408  
groupAdjacentBy, 452, 453  
groupBy, 456  
groupByKey, 455, 456  
grouped, 452  
groupedWithin, 452, 453  
interleave, 474  
interleaveWith, 474  
map, 405, 407, 446, 451  
mapAccum, 407, 447, 463  
mapAccumZIO, 447  
mapChunks, 446  
mapChunksZIO, 446  
mapConcat, 446, 447  
mapConcatChunk, 446  
mapError, 447  
mapErrorCause, 447  
mapZIO, 446  
mapZIOPar, 447, 448  
mapZIOParByKey, 447, 449  
mapZIOParUnordered, 447, 449  
merge, 466  
mergeHaltEither, 466  
mergeHaltLeft, 466  
mergeHaltRight, 466  
mergeSorted, 457, 466  
mergeWith, 466  
orElse, 470  
paginateZIO, 463  
partition, 457  
provideSomeLayer, 416  
repeat, 409  
repeatZIOOption, 409  
run, 493, 498  
runCount, 416  
runDrain, 413  
runHead, 415  
runLast, 415  
runSum, 416  
scan, 407, 447, 463  
scanZIO, 447  
scoped, 408  
sliding, 447  
split, 447

splitOnChunk, 447  
take, 407, 451  
takeRight, 407, 451, 456  
takeUntil, 407, 451  
takeUntilZIO, 451  
takeWhile, 407, 451  
takeWhileZIO, 451  
tap, 497  
tapSink, 496–498  
throttleEnforce, 462  
throttleShape, 462  
timeoutTo, 470  
transduce, 425, 494–496, 498, 504  
unfold, 411, 463  
unfold\*, 410  
via, 429  
zip, 407, 466, 470  
zipAll, 467  
zipAllLeft, 467  
zipAllRight, 467  
zipAllSortedByKeyLeft, 469  
zipAllSortedByKeyRight, 469  
zipAllWith, 467  
zipLatest, 467  
zipLatestWith, 467  
zipLeft, 467  
zipRight, 467  
zipWith, 467  
zipWithIndex, 468  
zipWithLatest, 568  
zipWithNext, 447  
zipWithPrevious, 447  
ZStream.mapZIOParUnordered, 460