Введение

Писать модульные приложения безусловно очень важно в области программирования. Умение разбивать проблему на более мелкие части и затем собирать их воедино для создания больших приложений — это базовое понятие. Это позволяет нам создавать программное обеспечение, независимо от степени сложности, которая в нем содержится. На самом деле, составляемость была одним из основных принципов ZIO с самого начала. Поэтому, чтобы хорошо понять, насколько ZIO великолепен для модульности, данная документация будет посвящена написанию приложения "Крестики-нолики" с использованием типа данных ZLayer.

Вот что вы узнаете:

- Какова структура сервиса, как предложено ZIO.
- Типы данных ZIO для создания модульных приложений: ZEnvironment и ZLayer.
- Псевдоним-типы ZLayer.
- Как организовать приложение ZIO вокруг ZLayers.
- Как создавать и комбинировать ZLayers
- Как организовать тестирование и моки вокруг ZLayers.
- Как уменьшить шаблонизацию при работе с ZLayers.
- Как автоматически генерировать диаграмму зависимости вашего приложения.

Дизайн игры "Крестики-нолики"

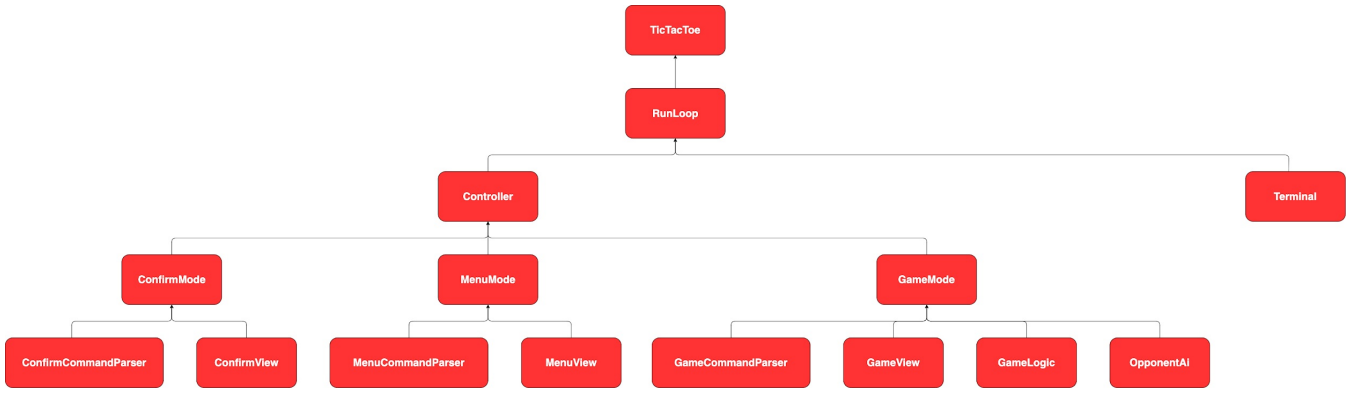
Прежде чем внедрять игру в Три в ряд, давайте рассмотрим аспекты дизайна, которые нам нужно учитывать:

- Это должно быть командная строка приложение, так что игра должна быть отрисована в консоль и пользователь должен взаимодействовать через текстовые команды.
- Приложение должно быть разделено на три режима, где режим определяется его состоянием и списком команд, доступных пользователю. Эти режимы должны быть: Режим подтверждения: Этот режим должен просто ждать подтверждения пользователя в виде команд да/нет.

- Режим меню: В этом режиме пользователь должен сможет начать, продолжить или выйти из игры.
- Режим игры: В этом режиме должны быть реализованы логика игры herself и возможность играть против искусственного интеллекта противника.

- Наша программа должна читать с Терминала, соответствующим образом изменять состояние и写入 на Терминал в Цикле.
- Мы также хотели бы очищать консоль перед каждым кадром.

Мы создадим отдельную службу для каждого из этих вопросов. Каждая служба будет зависеть от других служб, как показано на изображении ниже:



Глубокое исследование модульных приложений с ZIO

Как возможно, вы уже знаете, ZIO спроектирован вокруг трёх параметров типа:

ЗИО[-R, +E, +A]

Вы также можете помнить, что хорошая ментальная модель типа данных Zio является последовательной:

Zenvironment [r] => либо [e, a]

Это означает, что для выполнения эффекта ZIO需要一个类型为 ZEnvironment[R] среду (мы будем обсуждать это более подробно в следующем разделе о типе ZEnvironment),

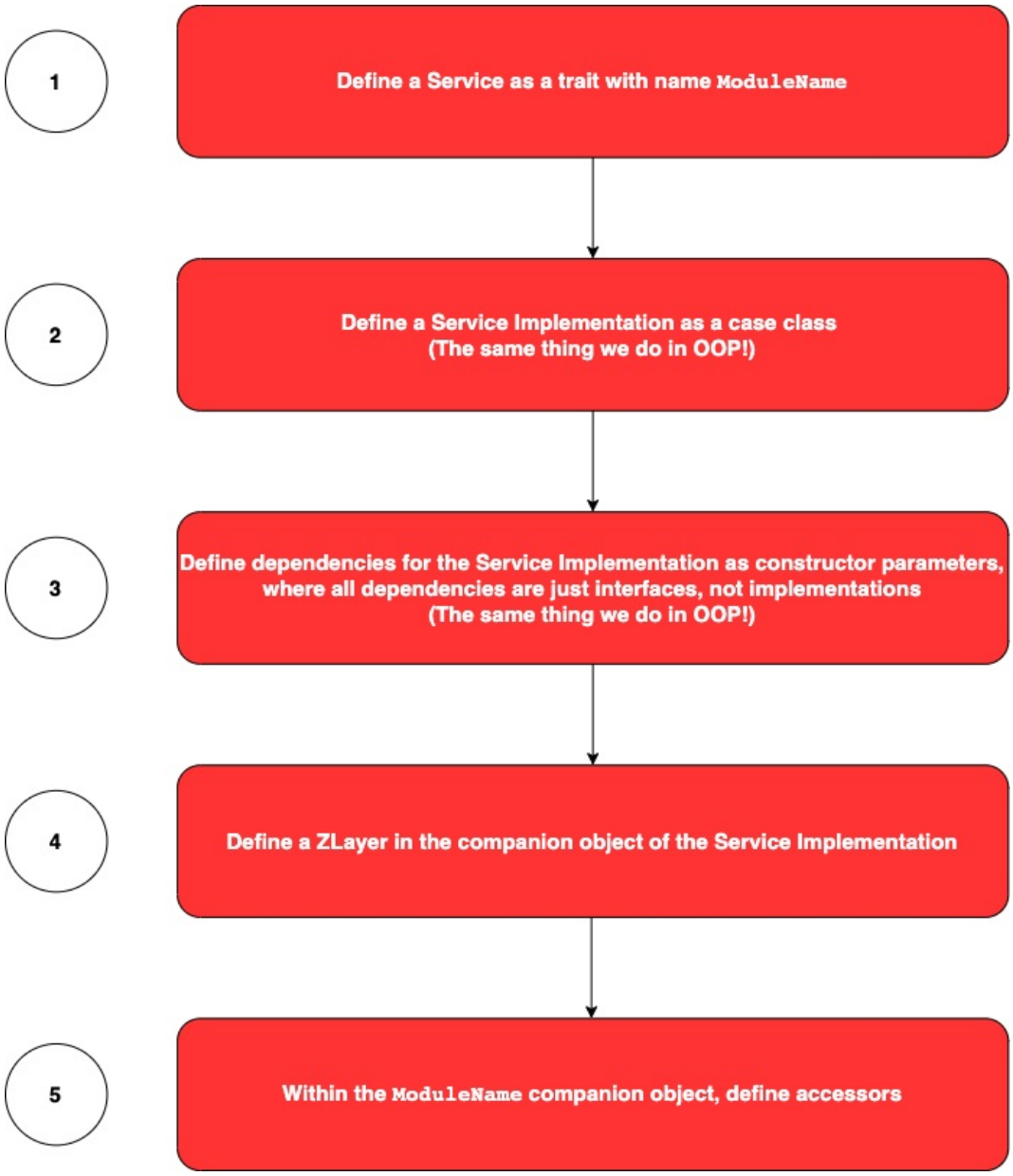
Таким образом, нам нужно удовлетворить это требование, чтобы сделать эффект теялватсдерп]R[tnemnorivnEZ пит тотэ ,онтеркнок ешьлоБ .уксупаз к لباق зависимость от сервиса или нескольких сервисов, необходимых для запуска эффекта. Поэтому давайте теперь обсудим, как в ZIO определяются сервисы (кстати, если вам нужна более глубокая информация о ZIO, вы можете ознакомиться с этой статьей в блоге Scalac)

О сервисах в ZIO

Как упоминается на странице документации ZIO: «Сервис — это группа функций, которая занимается только одной задачей. Ограничение области применения каждого сервиса только одной задачей улучшает нашу способность понимать код, так как нам нужно сосредоточиться только на одной теме в один момент без балансирования слишком большого количества концепций в голове».

Идея заключается в том, что ZIO позволяет нам определять сервисы и использовать их для создания различных слоев приложений, которые зависят друг от друга. Это означает, что каждый слой зависит от слоев, которые расположены сразу beneath него, хотя он не знает ничего о деталях их реализации. Это очень мощное concepto, так как оно улучшает composable и testability (поскольку вы легко можете изменить реализацию каждого сервиса, не затрагивая другие слои).

Теперь, если вы думаете о том, как определить эти услуги, ZIO предоставляет нам отличный рецепт для следования при определении новой услуги. Этот рецепт должен быть знаком объектно-ориентированным программистам:



Не волнуйтесь, если все это кажется слишком абстрактным в данный момент, потому что мыlater будем применять этот рецепт для реализации приложения "Крестики-нолики". Единственное, что важно сейчас, это узнать ZLayer, очень важный тип данных, упомянутый в этом рецепте, и который связан с другим очень важным:

Зенвайронмент. Давайте обсудим их сейчас.

Тип данных ZEnvironment

Как упоминается на странице документации ZIO, ZEnvironment[R] — это встроенный уровень карты для типа данных ZIO, который отвечает за поддержание окружающей среды ZIO effect. Тип данных ZIO использует эту карту (можно думать о ней как о Map[ServiceType, ServiceImplementation]), чтобы поддерживать все окружающие сервисы и их реализации.

Важно отметить, что ZEnvironment заменяет старый тип данных Has из ZIO1.0, который не был очень удобным для пользователей. Также, ZEnvironment теперь включает в себя тип данных ZIO himself, что еще больше улучшает его удобство использования.

Немногое пример того, как используется ZEnvironment

Теперь давайте посмотрим на очень простой пример использования ZEnvironment. Давайте предположим, что у нас есть эффект getCurrentUser, который требует某些 служб (Logging и HttpClient) из окружения:

```
val getCurrentUser: URIO[Вход с помощью HttpClient, Пользователь] = ???
```

Наша услуга определена следующим образом:

```
атрибут Logging                                    // интерфейс службы
в конечном итоге класс LoggingLive() extends Logging // Реализация услуги

трэйт HttpClient
конец случая класс HttpClientLive() extends HttpClient
```

Таким образом, чтобы ZIO мог выполнить эффект getCurrentUser, нам нужно предоставить ему необходимые зависимости. Для этого сначала нужно создать ZEnvironment, содержащий все необходимые зависимости:

```
значение env: ZEnvironment[Ведение журнала с помощью HttpClient] =
  ZEnvironment(LoggingLive(), HttpClientLive())
```

Теперь мы можем предоставить эту среду getCurrentUser, вызвав ZIO#provideEnvironment:

```
значение getcurrentuserwithenv: uio [user] = getcurrentuser.provideenvironment (env)
```

И теперь у нас есть эффект ZIO, который не требует никакой среды, потому что мы предоставили все необходимые зависимости, и ZIO сможет его выполнить.

Теперь несколько слов о ZEnvironment, и это то, что обычно вам не нужно работать с ним напрямую. Потому что есть более мощный тип данных, который вы можете использовать вместо этого, чтобы предоставить необходимые зависимости для эффекта ZIO: ZLayer.

Тип данных Zlayer

Тип данных Zlayer является неизменным значением, которое содержит чистое описание, запрещающее ZenVironment [Rout], начиная с значения RIN, возможно, создавая страницу во время создания:

```
ЗЛейер[-RIn, +E, +ROut]
```

Если подумать, ZLayer немного напоминает конструктор класса. Однако, в то время как конструктор класса описывает, как вы строите объекты определенного класса, он не описывает процесс как ценность, но ZLayer делает это! Так что:

- Конструктор класса не является значением так же, как и команда не является значением
- Как ZIO эффекты превращают предложения в значения, ZLayers превращают конструкторы в значения
- ZLayers также могут описывать процесс разрушения услуги, а не только её строительство!

Поскольку ZLayers являются значениями, они имеют высокую составность и могут комбинироваться двумя основными способами:

- Вертикально: Для создания слоя, который имеет требования и предоставляет возможности обоих слоев, мы используем оператор ++.
- Вертикально: В этом случае, результат одного слоя используется в качестве ввода для следующего слоя, что приводит к слою, требующему ввода первого и вывода второго слоя. Мы используем оператор >>> для этого.

Опять же, не паникуйте, если это не слишком понятно вам в данный момент, потому что мы будем применять как горизонтальное, так и вертикальное составление при реализации приложения Крестики-нолики, и все станет яснее.

Почему ZLayer?

Теперь, возможно, вы думаете: действительно ли нам нужен ZLayer? Не够 ли ZEnvironment для предоставления зависимостей ZIO эффекту?

Ответ заключается в том, что в небольших приложениях с ограниченным количеством услуг ZEnvironment вполне достаточен. Однако на практике множество приложений состоят из тысяч или миллионов строк кода, содержат несколько различных сервисов и имеют различные тестовые и производственные реализации для каждого сервиса. Ручное подключение всех этих сервисов становится утомительным упражнением, и это шанс для людей повторять одни и те же ошибки снова и снова.

Что вы вместо этого хотите, так это某种 автоматический механизм инъекции зависимостей, который предоставляет вам структуру, много структуры, так что вы в основном делаете это очень простым для людей, чтобы добавлять новые сервисы, новые реализации и обеспечивать соблюдение лучших практик.

Это и есть суть ZLayer! Он помогает вам структурировать масштабные приложения так, чтобы это масштабировалось.

Кстати, beste practices, которые автоматически enforced by ZLayer (вам даже не нужно думать о них!), включают следующее:

- Когда вы подключаете граф зависимостей вашего приложения, вы должны пытаться сделать это параллельно, чтобы уменьшить время загрузки бутстрэп. Naturally, вы не можете подключить весь граф зависимостей параллельно, потому что иногда у вас есть последовательные части, и ZLayer точно знает, какие части можно построить параллельно, а какие последовательно.
- Также, когдаапу компонент вашей программы больше не используется, вы должны безопасно высвободить ресурсы, такие как открытые файловые дескрипторы или сетевые подключения.

Типы алиасов ZLayer

В конце концов, стоит отметить, что ZIO предоставляет一些 типовые.alias для типа ZLayer, которые очень полезны при представлении некоторых общих случаев использования. Хорошая новость заключается в том, что логика для определения этих типовых alias почти такая же, как и для определения типовых alias ZIO. Вот полный список:

- TaskLayer[+ROut] = ZLayer[Any, Throwable, ROut]: Это означает, что TaskLayer[ROut] является ZLayer, который:
  - Не требует ввода (поэтому тип RIn заменяется на Any)
  - Может失败并抛出 Бросаемый
  - Может succeed с ROut
- ULayer[+ROut] = ZLayer[Any, Nothing, ROut]: Это означает, что ULayer[ROut] является ZLayer, который:

- Не требует ввода
- Нельзя промахнуться
- Можешь succeed с ROut

- RLayer[-RIn, +ROut] = ZLayer[RIn, Throwable, ROut]: Это означает, что RLayer[RIn, ROut] является ZLayer, который:
  - Требует входной RIn
  - Может失败，抛出异常
  - Может успешиться с ROut

Слой [+e, +round] = zlayer [any, e, rout]: это означает слой [e, rout] - это азлиер, что:

Не требует ввода

- Может失败与一个E
- Может succeed с an ROut

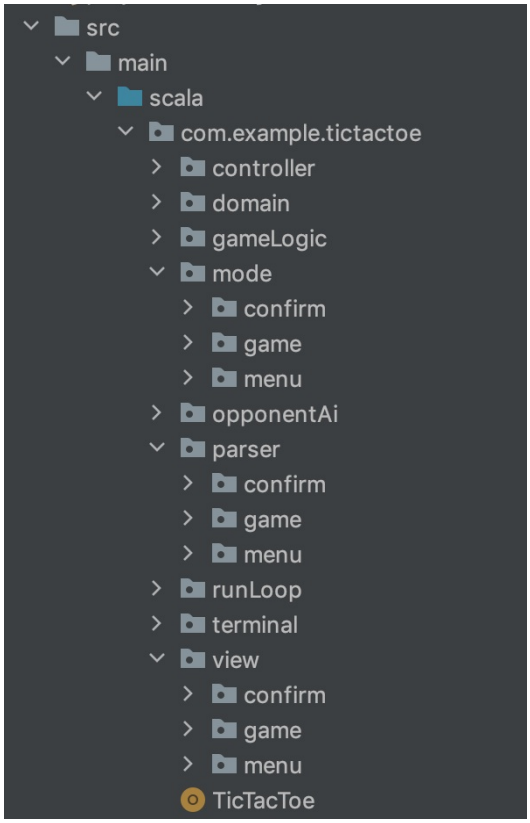
- URLayer[-RIn, +ROut] = ZLayer[RIn, Nothing, ROut]: Это означает, что URLayer[RIn, ROut] является ZLayer, который:
  - Требует входной RIn
  - Не может не succeed
  - Может succeed с ROut

Теперь, если вы wonder, как создать и использовать ZLayers, не уходите, так как мы скоро увидим, насколько легко это сделать в следующем разделе.

Реализация приложения Tic-Tac-Toe

Времяimplemented приложение Крестики-нолики using the ZIO Service Pattern with ZLayer! В следующих разделах мы будем анализировать код некоторых из служб (наиболеерepresentative ones). Вы можете увидеть completo код в репозитории jorge-vasquez-2301/zio-zlayer-tictactoe.

Кстати, это будет структура каталога проекта:



Итак, каждый сервис ZIO будет реализован в виде пакета, содержащего:

- Сервисный интерфейс как trait.
- Реализации сервисов как случаи классов.

Рассуждая об этом, эти услуги отражают начальный дизайн, представленный выше. У нас также есть пакет домена, содержащий объекты домена, и основной объект TicTacToe.

Кроме того, нам нужно добавить некоторые зависимости в наш build.sbt (atto используется для анализа команд):

```
скалавер val = "2.13.10"

val attroversion = "0,7,2" val
zioversion = "2,0,12" val
ziomockversion = "1.0.0-rc11"
```

```
ленивый val compileDependencies = Seq("dev.zio"
  %% "zio" % zioVersion,"dev.zio" %%
  "zio-macros" % zioVersion,"org.tpolecat"
  %% "atto-core" % attoVersion) map (_ %
Скомпилировать)

Ленивый Val TestDependencies = seq ("dev.zio" ""
  дядя-тест " % Zoversion," dev.zio "%%"
  дядя-тест-SBT " % Zoversion" dev.zio "%%"
  Дядя-мак " % Ziomockversion ) Карта (_ %
тест)

Настройки Lazy Val = seq (имя: =
  "Zio-Zlayer-tictactoe", версия:
  = "4.0.0",
  Скалаверсия: = Скайвер,
  scalacOptions += "-Ymacro-annotations", библиотеки +=
  compileDependencies ++ testDependencies, testFrameworks := Seq(new
  TestFramework("zio.test.sbt.ZTestFramework")))


lazy val root = (проект в файле(".")).
 настройки(settings)
```

Пожалуйста, обратите внимание, что мы работаем с Scala 2.13.10 и нам нужно будет включить компилятор flag -Ymacro-annotations, чтобы мы могли использовать некоторые макросы, предоставляемые zio-macros. Если вы хотите работать с Scala < 2.13, вам нужно добавить плагин компилятора macro paradise:

```
compilerPlugin (("org.scalamacros" % "paradise" % "2.1.1") кросс-версия.полная)
```

Реализация сервиса GameCommandParser

Здесь мы видим интерфейс сервиса GameCommandParser, в файле parser/game/GameCommandParser.scala:

```
特性 GameCommandParser {синтаксический анализ определения (ввод: строка):
  Ввод-вывод[AppError, GameCommand]}
```

Как вы можете видеть, интерфейс сервиса всего лишь простая характеристика, которая экспонирует некоторые возможности, такие как метод parse, который может失败 с AppError или успешным с GameCommand. Хотелось бы упомянуть что-то очень важное здесь: в общем, когда пишете интерфейс сервиса, вы никогда не должны иметь методы, возвращающие ZIO effects, требующие 环境. Причины для этого объяснены очень хорошо в этой статье о Трёх Законах Среды ZIO из документации ZIO.

Теперь, когда мы написали интерфейс сервиса, нам нужно определить возможные реализации. На данный момент у нас будет только одна реализация, которая будет case класс названный GameCommandParserlive, в a файл синтаксический анализатор/game/GameCommandParserLive.scala:

```
конечный класс GameCommandParserLive() расширяет GameCommandParser {def
  синтаксический анализ (ввод: строка): IO[AppError, GameCommand] =
    ZIO.from(команда.синтаксический анализ (ввод).готово.опция).orElseFail(ОшибкаПарсинга)

  частная команда lazy val: Синтаксический анализатор[GameCommand] =
    выбор (меню, поместить)

  private ленив val menu: Парадер[КомандаИгры] =
```

```
    (Строка ("Menu") <~ endofinput)> | GameCommand.menu

  частный ленивый ввод значения: Парсер[GameCommand]
    =(int <~ Конечный вывод). fl atMap { значение =>
      Поле
        .сделать(значение)
        .fold (err [GameCommand] (S "Недопустимое значение Fi ELD: $
          значение")) {поле => ok (поле) .карта (gamecommand.put)
        }
    }
}
```

Как было показано, способ записи реализации услуги именно такой же, как если бы мы занимались Объектно-ориентированным программированием (OOP)!! Просто создайте новый класс случая, который реализует определение услуги (в данном случае, GameCommandParserTrait). И, поскольку GameCommandParserLive не зависит от других служб, у него есть пустой конструктор.

Далее, нам нужно создать ZLayer, который описывает, как создавать экземпляр GameCommandParserLive. Для этого просто добавьте следующее в объект сопутствующего класса GameCommandParserLive:

```
объект GameCommandParserLive {
  значение слой: ULayer[GameCommandParser] = ZLayer.success(GameCommandParserLive())}
```

Теперь вы можете оценить, как легко создавать ZLayer! Нам нужно всего лишь вызвать метод ZLayer.succeed, предоставив экземпляр GameCommandParserLive. В данном случае, возвращается ZLayer[Any, Nothing, GameCommandParser], что эквивалентно ULayer[GameCommandParser]. Это означает, что возвращаемый ZLayer:

- Не имеет зависимостей.
- Не может失败 при создании.
- Возвращает GameCommandParser.

Мы почти закончили с нашей службой GameCommandParser, нам нужно только добавить несколько доступов, которые являются методами, помогающими нам создавать программы без забот о деталях реализации услуги. Мы поместили эти доступы в сопутствующий объект GameCommandParser:

```
объект GameCommandParser {
  синтаксический анализ определения (входные данные: строка): ZIO[GameCommandParser, AppError, GameCommand] = ZIO.
    serviceWithZIO[GameCommandParser](_.синтаксический анализ (ввод))
}
```

Командный парсер The GameCommandParser.parse использует ZIO.serviceWithZIO для создания эффекта, требующего GameCommandParser в качестве окружения, и просто вызывает метод parse на нем. В общем, написание аксессоров всегда будет следовать этому же шаблону.

Теперь хорошая новость заключается в том, что на самом деле нам не нужно писать эти доступы ourselves, мы можем использовать аннотацию @accessible (которая приходит из библиотеки zio-macros) на трейте GameCommandParser. Делая это, доступы будут автоматически генерироваться для нас:

```
импортировать zio.macros._

@доступно
Черта GameCommandParser {определение синтаксического анализа (ввода: строка): ВВОД-вывод
  [Ошибка, игровая команда]}
```

Реализация сервиса GameMode

Здесь мы имеем интерфейс обслуживания для услуги GameMode в файле mode/game/GameMode.scala:

```
@доступно
trait ИграМод {
  def обработка(вход: Строка, состояние: State.Game):
  UIO[State]def отрисовка(состояние: State.Game): UIO[Строка]
}
```

И у нас есть реализация сервиса в режиме/game/gamemodelive.scala:

```
конечный случай класс GameModeLive(
  gameCommandParser: GameCommandParser,
  gameView: GameView,
  противникAi: ПротивникAi,
  игроваяЛогика: GameLogic
) Расширяет GameMode {
 def обработка(вход: String, состояние: State.Game): UIO[State] =if (состояние.результат != GameResult.
  В процессе) ZIO.успех(State.Меню(None, MessageFooterEmpty))else if (isAiTurn(состояние))

    противникai
      . Случайное перемещение(state.board).
      плосКая карта(takeField(_, state))
  ещё
    игровойКомандныйПарсер.
      нарсинг (ввод)
      . fl на карте {
        case КомандаИгры.Меню => ZИО.успех(ГосударстваМеню(Некоторые(состояние),
        СообщениеПодНогой.Пустое))case КомандаИгры.Пут(поля) => takeField(поля, состояние)
      }
      .илиУспешноСкопироватьСтейт(footerMessage = GameFooterMessage.НеправильнаяКоманда)

частная защита - это поворот (состояние: Состояние.Игра): Логическое значение = (состояние.ход
  == Фигура.Крест && состояние.крест == Игрок.Ai) || (state.turn ==
    Часть.Ничто и& государство.ничто == Player.Ai)
```

```
  private def takeField(поля: Field, состояние: State.Game):
    UIO[Состояние] =(для {
      обновленныйBoard <- gameLogic.putPiece(state.board, поля, состояние.
      turn)обновленныйРезультат <- gameLogic.gameResult(обновленныйBoard)
      обновленныйTurn <- gameLogic.nextTurn(state.
    повернуть)} состояние выхода.копировать(
      доска = обновленныйBoard,
      result = обновленныйResult,
      turn = обновленныйTurn,
      SOOTERMESSAGE = GAMEFOOTERMESSAGE.EMPTY)). СОКРАЩЕННЫЙ (штат.Копировать
    (TOYERMESSAGE = GAMEFOOTERMESSAGE.Занято поле))))

  def render (состояние: state.game): uio [string] = {val player = if (state.
    поворот == фигура.пересечь) состояние.пересечь остальное состояние.noughtfor {

      заголовок <- GameView.header(состояние.результат, состояние.ход,
      игрок)содержимое <- GameView.content(состояние.доска, состояние.
      результат)подвал <- GameView.footer(состояние.footerMessage)
    }{ yield List(заголовок, содержимое, подвал).
  mkString("\n\n")}
}
```

Обратите внимание на то, как путь к делу этой службы GameMode очень похож на фонд GameCommandParser. Однако есть разница: GameCommandParserLive не имеет зависимостей, предоставленных через классовый constructor, но GameModeLive имеет четыре зависимости! Пожалуйста, обратите внимание, что мы используем интерфейсы для запроса этих зависимостей, а не реализации, поэтому мы следуем очень важному принципу ООП: Программируйте к интерфейсам, а не к реализациям!

Итак, OK, как мы теперь создаем ZLayer для GameModeLive? Мы можем сделать это так:

```
объект GameModeLive {
  val слой: URLayer[
    GameCommandParser с Просмотр игры, оппонент и логика игры, игровой режим

  ] = ZLayer.fromFunction(GameModeLive(_, _, _, _))
```

Нам нужно только вызвать метод ZLayer.fromFunction в конструкторе GameModeLive, чтобы поднять его до ZLayer. В этом случае, то, что возвращается, это ZLayer[GameCommandParser with GameView with OpponentAi with GameLogic, Nothing, GameCommandParser], что эквивалентно URLayer[GameCommandParser with GameView with OpponentAi with GameLogic, GameCommandParser]. Это означает, что возвращаемый ZLayer:

- Зависит от GameCommandParser, GameView, OpponentAi и GameLogic.
- Не может失败在创建时。
- Возвращает режим игры.

Создание более мощных ZLayers

В этой части я хочу упомянуть, что более мощным методом создания ZLayers является вызов ZLayer.fromZIO (эквивалент ZLayer.apply), что позволяет нам описывать более сложные процессы для создания Сервисов. Вот как, например, можно вывести сообщение в консоль при инстанцировании GameModeLive:

```
объект GameModeLive {
  val слой: URLayer[
    GameCommandParser с GameView с OpponentAi с GameLogic,GameMode

  ] =
    Zlayer {
      для {
        gameCommandParser <- ZIO.
        сервис[GameCommandParser]GameView <- ZIO.service[GameView]
        opponentAi <- ZIO.
        сервис[OpponentAi]GameLogic <- ZIO.
                          <- Console.printLine("Инсталлирование GameModeLive").Порядок} выход
        сервис[GameLogic]_
      GameModeLive(gameCommandParser, GameView, opponentAi, GameLogic)}
```

```
}
```

И мы можем создать ZLayers, которые могут выполнять еще более мощные действия, такие как открытие ресурсов (например, файлов), вызывая ZLayer.scoped, что позволяет нам создать ZLayer изScoped ZIO effect (я не буду объяснять все детали oScoped ZIO effects здесь, просто скажем, что они являются заменой старого типа данных ZManaged из ZIO 1.0, так что они, по сути, позволяют безопасно выделять и освобождать ресурсы. Если вы хотите больше详情, вы можете взглянуть на документацию ZIO).

Давайте, например, предположим, что при инстанцировании GameModeLive сообщение, которое мы выводим на консоль, должно поступать из файла, а не быть硬编码ированным.

Мы можем это сделать так:

```
val слой: URLayer[
  Обработчик команд игры с видом игры, с искусственным интеллектом
  противника, с логикой игры, режимом игры
] = {
  импорт scala.io.Source

  val получитьSource: URIO[Scope, BufferedSource] =
   ЗIO.acquireRelease(ЗIO.attemptBlockingIO(Источник.fromFile("message.txt")).orDie)(источник
     => ЗIO.attemptBlockingIO(источник.close).Орди)
   )

 Zlayer.scoped {
   для {
     игровойКомандныйПарсинг <- ZIO.
     service[GameCommandParser]игровОглед <- ZIO.service[GameView]
     противникАИ <- ZIO.
     service[OpponentAi]игроваяЛогика <- ZIO.
     service[GameLogic]источник <- Получить источник
                        <- Консоль.printLine(source.mkString("\n")).Порядок} выход
   GameModeLive(gameCommandParser, GameView, opponentAi, GameLogic)}

}
```

Теперь у нас есть очень мощный ZLayer, который не только знает, как инстанцировать aGameModeLive, но и умеет безопасно освободить ресурсы (в этом случае, файл message.txt, который читается) при удалении экземпляра!

Осуществление основного объекта TicTacToe

Объект TicTacToe является точкой входа в наше приложение:

```
объект TicTacToe extends ZIOAppDefault {

  val программа: URIO[RunLoop, Unit] = {def
    loop(state: State): URIO[RunLoop, Unit] =
      RunLoop
        . шаг (state).
        что-то
        . плоский на карте
        (петля). пренебречь

    цикл(Состояние.начальное)


значение run = program.providelayer (Environmentlayer)

private ленивый val environmentLayer: ULayer[RunLoop] = {

  значение confirmModeDeps: ULayer[ConfirmCommandParser с ConfirmView] =
    ConfirmCommandParserLive.layer ++ ConfirmViewLive.layer
  значение menumodedeps: ulayer [menucommandparser с menuview] =
    menucommandparserlive.слой ++ menuviewlive.слой
  Val Gamemodedeps: Ulayer [GameCommandParser с GameView с GameLogic с OpponityAI] =
    GameCommandParserlive.слой ++ GameViewLive.Слой ++ GameLogiclive.слой ++ Opponitive.Слой.

  значение con fi rmmodenodeps: ulayer [confi rmmode] = con fi rmmodedeps >>> confi rmmodelive.layerval
  menumodenodeps: ulayer [menumode] = menumodedeps >>> menumodelive.layerval gamemodeps:
  ulayer [игровой режим] = gamemodeepseepselep слой

  Значение ControllerDeps: ulayer [confi -rmmode с Игровой режим с Menumode] = confi
    rmmodenodeps ++ Gamemodenodeps ++ menumodenodeps
```

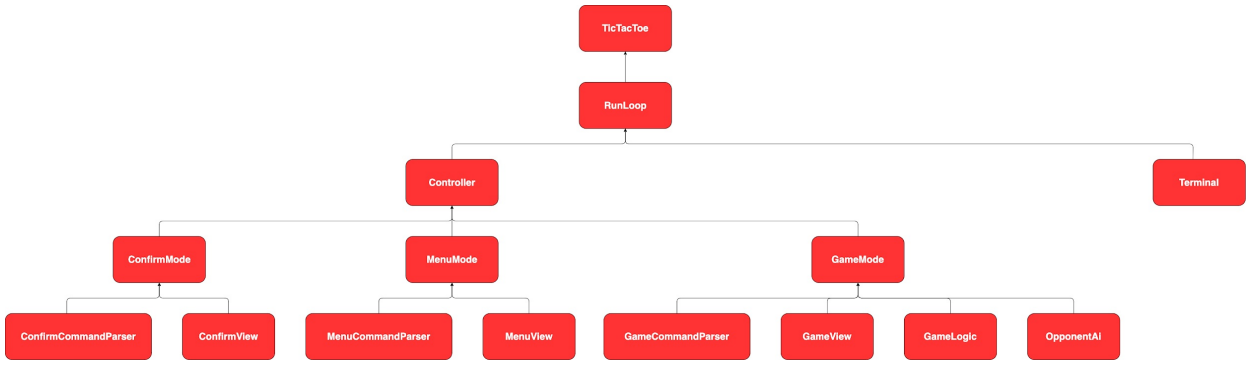
```
    Val Controlernodeps: Ulayer [Контроллер] = ControllerDeps >>> ControllerLive.Слой

    значение runloopnodeps = (controlernodeps ++ terminallive.слой) >>> runlooplive.слой

    runLoopNoDeps

}
```

Некоторые важные моменты для замечения в приведенном выше коде:

Tictactoe расширяет ZioAppdefault

- Значение программы определяет логику нашего приложения и зависит от RunLoop Service, который, в свою очередь, зависит от остальных сервисов нашего приложения.
- Метод run, который должен быть реализован в каждой ZIO-приложении, предоставляет подготовленную среду для выполнения нашей программы. Для этого он выполняет program.provideLayer, чтобы предоставить подготовленный ZLayer (определенный значением environmentLayer).

Так что теперь давайте шаг за шагом проанализируем реализацию prepareEnvironment. Для этого еще раз обратимся к нашему начальному设计方案:



Цель - предоставить реализацию слоя RunLoop для функции TicTacToe.run. Для этого мы будем следовать подходу снизу вверх.

Если мы посмотрим на нижнюю часть обновленного диаграммы, мы можем увидеть,

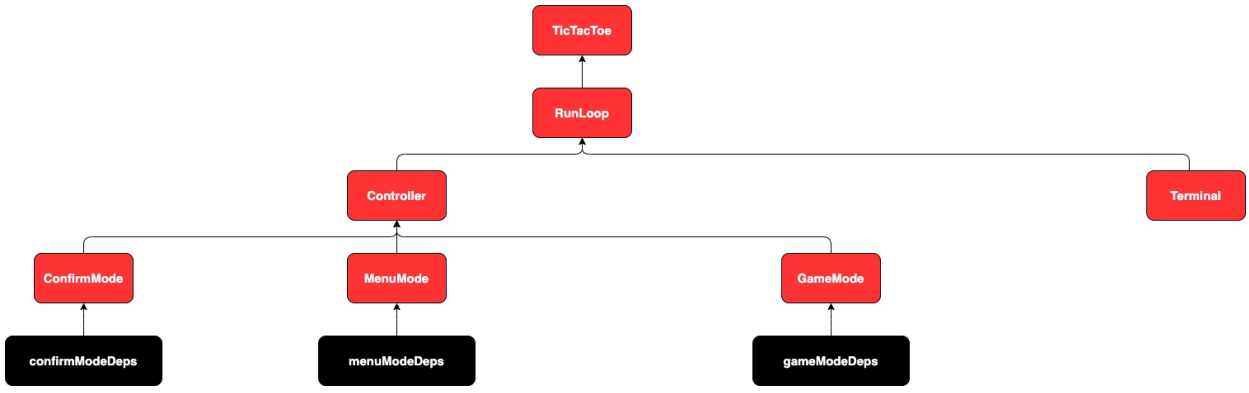
что есть некоторые возможности для выполнения горизонтальной композиции:

- Против rmCommandParser и Против rmView
- Против MenuCommandParser и Против MenuView
- Против GameCommandParser, GameView, GameLogic и Оппонент

Таким образом, у нас следующее в коде:

```
val con fi rmmodedeps: ulayer [con fi rmcommandparser с конфликтом] = confi
  rmcommandparserlive.уровень ++ confi rmviewlive.слой
значение параметров меню: ULayer[MenuCommandParser с MenuView] =
  MenuCommandParserLive.layer ++ MenuViewLive.layer
val gameModeDeps: уровень [GameCommandParser с GameView с GameLogic с OpponentAi] =
  GameCommandParserLive.layer ++ GameViewLive.layer ++ GameLogicLive.layer ++ OpponentAiLive.layer
```

И графически:



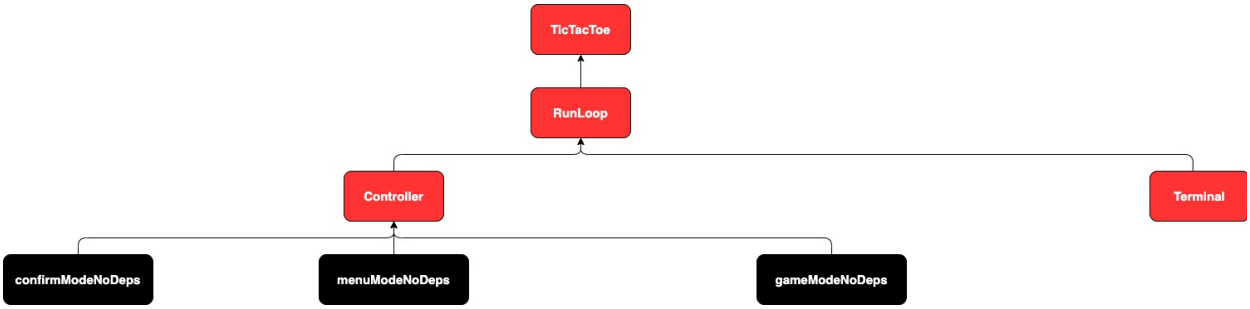
Ничего себе! Теперь мы можем сжать еще один уровень, применяя вертикальную композицию :

значение con fi rmmodenodeps: ulayer [confi rmmode] = con fi rmmodedeps >>> confi rmmodelive.layerval

menumodenodeps: ulayer [menumode] = menumodedeps >>> menumodelive.layerval gamemodeps:

ulayer [игровой режим] = gamemodeepseepselep слой

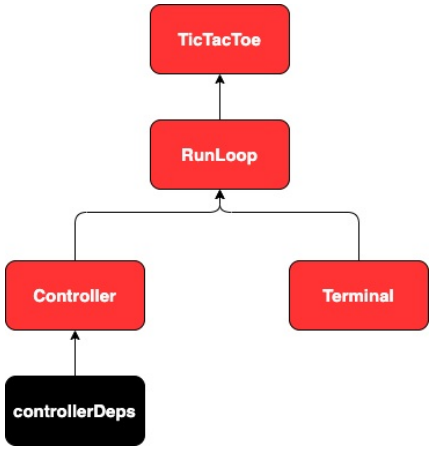
И теперь у нас есть:



Далее, мы можем снова применить горизонтальное composition:

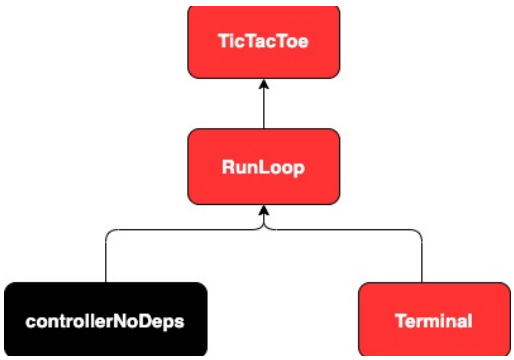
Значение ControllerDeps: ulayer [confi -rmmode c GameMode c Menumode] = confi

rmmodenodeps ++ Gamemodenodeps ++ menumodenodeps



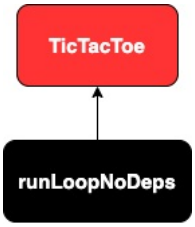
Следующим шагом будет (предупреждение: спойлер): Вертикальная компоновка!

Val Controlernodeps: Ulayer [Контроллер] = ControllerDeps >>> ControllerLive.Слой



И наконец, мы можем применить горизонтальное и вертикальное-compose в один шаг, и мы будем закончены:

```
значение runloopnodeps = (controlernodeps ++ terminallive.слой) >>> runlooplive.слой
```



Вот и всё! Теперь у нас есть подготовленная среда, которую мы можем предоставить нашему программному обеспечению для того, чтобы сделать его runnable, путём вызова ZIO#provideLayer:

```
значение run = program.providelayer (Environmentlayer)
```

Теперь, как упражнение для ума и для лучшего понимания关系的 между типами данных ZEnvironment и ZLayer, давайте посмотрим, как предоставить ZLayer эффекту ZIO, но используя ZIO#provideEnvironment:

```
значение run =
  для {
    Zenvironment <- EnvironmentLayer.build_
                  <- Программа.ProvideEnvironment (ЗенВирэнмент)
```

доходность ()

Здесь вы можете увидеть, по сути, что происходит под капотом, когда вы вызывается ZIO#provideLayer: Данный environmentLayer, который всего лишь является чистым описанием того, как конструировать зависимости нашего приложения, создается вызовом ZLayer#build, который возвращает ZIO эффект, успешный с ZEnvironment, который затем может быть предоставлен нашему программному обеспечению, вызывая ZIO#provideEnvironment.

Магически уменьшение шаблонного кода в объекте TicTacToe

В предыдущем разделе мы рассмотрели, как подготовить окружающую среду для нашего приложения, комбинируя ZLayers, используя горизонтальное и вертикальное составление, и нам нужно было это делать вручную.

Да, я знаю, о чём вы, возможно, сейчас думаете: я думал, что вся цель использования ZLayer вместо ZEnvironment для предоставления необходимых зависимостей ZIO effect заключалась в том, что мы не будем нуждаться в ручной настройке этих зависимостей, но整个过程 по-прежнему выглядит мне очень ручным!

Хорошая новость заключается в том, что в ZIO 2.0 действительно существует автоматический способ подключения всех ZLayers нашей программы, который я вам пока не показывал (в ZIO 1.0 этой функции нет, поэтому в этом случае вам нужно будет использовать внешнюю библиотеку, написанную KitLangton, под названием Магия ЗИО).

Хорошо, теперь посмотрим, как ZIO 2.0 может помочь нам уменьшить рутину при подготовке нашей среднеслой:

```
частный параметр environmentLayer: ULayer[RunLoop] =
  ZLayer.создать[RunLoop](
    ControllerLive.layer,
    GameLogicLive.layer,
    Настройка rmModeLive.layer,
    Игровая модель.слой,
```

```
    MenuModeLive.layer,
    OpponentAiLive.layer, Против
    rmCommandParserLive.layer, против
    GameCommandParserLive.layer, против
    MenuCommandParserLive.layer, против
    RunLoopLive.layer, против
    TerminalLive.layer, против
    rmViewLive.layer, против
    GameViewLive.layer, против
    MenuViewLive.layer)
```

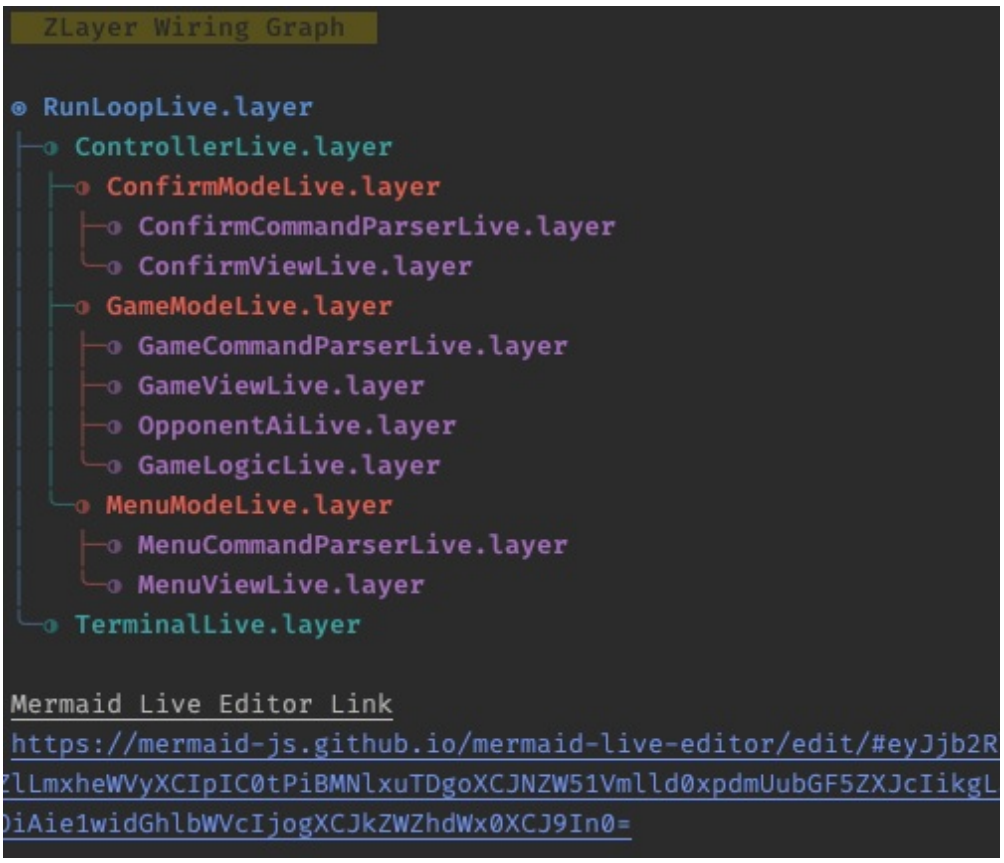
Ух! Это значительное улучшение, не так ли? В ZIO 2.0 вам нужно всего лишь вызвать ZLayer.make с параметром типа, указывающим тип ZLayer, который вы хотите создать (в данном случае ZLayer, который возвращает RunLoop), а затем просто предоставить все слои, которые необходимо соединить, в любом порядке, который вам подходит, и это все! Вам больше не нужно думать о горизонтальном и вертикальном составлении! ZIO 2.0 позаботится об этом за вас.

Кстати, приятной особенностью ZIO 2.0 является то, что если вы добавите слой ZLayer.Debug.mermaid в вызов ZLayer.make, как это:

```
частный параметр environmentLayer: ULayer[RunLoop] =
  ZLayer.создать[RunLoop](
    ControllerLive.слой,GameLogicLive.
    слой,Кон фи rmModeLive.слой,
    GameModeLive.слой,
    MenuModeLive.слой,
    OpponentAiLive.слой,Кон фи
    rmCommandParserLive.слой,
    GameCommandParserLive.слой,
    MenuCommandParserLive.слой,
```

```
RunLoopLive.layer,
TerminalLive.layer, Con
fi rmViewLive.layer,
GameViewLive.layer,
MenuViewLive.layer,
ZLayer.Debug.русалка)
```

Вы получите красивое представление дерева графа зависимостей во время компиляции, включая ссылку Mermaid.js с приятным диаграммой Mermaid, которую вы можете даже экспортировать в виде изображения!



И как бонус, оказалось, мы все еще можем уменьшить некоторые шаблонные фрагменты. Благодаря ZIO 2.0 нам больше не нужно поддерживать environmentLayer в виде отдельной переменной. Давайте

Посмотрите, как это будет работать, исходный метод выполнения, который использует environmentLayer, выглядит так:

```
значение run = program.providelayer (Environmentlayer)
```

Мы вызвали ZIO#provideLayer для предоставления подготовленной среды нашему программному обеспечению.

Что мы можем теперь сделать вместо этого, так это следующее:

```
значение run =
  программа
    .поставить(
      ControllerLive.layer,GameLogicLive.
      layer,Подтвердитьmodelive.layer,
      GameModeLive.layer,
      MenuModeLive.layer,
      Противникailive.layer,
      Подтвердитьcommandparserlive.
      layer,GameCommandParserLive.
      layer,MenuCommandParserLive.
      layer,RunLoopLive.layer,

      TerminalLive.layer,
      Подтвердитьviewlive.
      layer,GameViewLive.
      layer,MenuViewLive.
    слой)
```

Мы можем напрямую вызывать метод ZIO#provide в нашей программе для предоставления необходимых ZLayers в любом порядке.

Написание тестов

Поскольку мы успешно реализовали приложение TicTacToe с использованием ZLayers, давайте теперь напишем тесты для приложения. Мы рассмотрим здесь только некоторые из них, и, конечно же, вы можете посмотреть на полные тесты в репозитории jorge-vasquez-2301/zio-zlayer-tictactoe.

Писание GameCommandParserSpec

Вот тестовый набор для GameCommandParser:

```
объект GameCommandParserSpec расширяет ZIOSpecDefault
  {определение spec =
    -набор ("GameCommandParser")(набор("синтаксический анализ")(

        тест("menu возвращает команду
          Меню") {для {
            Результат <- GameCommandParser.parse ("Menu"). Los.
          right} допустить Asserttrue (result == GameCommand.menu)
        },
        тест ("номер в диапазоне 1-9 возвращает команду
          Put") {val Results = Zio.Foreach (от 1 до 9) {n =>
            для {
              result <- GameCommandParser.parse(s"$n").either.
              rightожидаемоеПоле <- ZIO.from(Поле.создать(n))
            } верю, что результат == GameCommand.Put(ожидаемоеПоle)}

          результаты. fl atMap(результаты => ZIO.from(результаты.
        reduceOption(_ && _)),
        test("недопустимая команда
          возвращает ошибку")
            {проверка (invalidCommandsGen) { ввод
          =>для {
              результат <- GameCommandParser.parse(input).either.
            left} выдать assertTrue(результат == ошибка синтаксического анализа)
          }
        }
      )
    ).provideLayer(Парсер команды игры live.слой)

private val validCommands = Список(1 до 9)private val invalidCommandsGen = Gen .
строка.фильтр(!validCommands.содержит(_))
```

Как вы видите, все тесты зависят от сервиса GameCommandParser, поэтому нам нужно предоставить его, чтобы zio-test мог запускать тесты. Теперь мы можем предоставить реализацию GameCommandParserLive всему набору тестов, используя Spec#provideLayer.

Запись TerminalSpec

Давайте посмотрим на спецификацию:

```
объект Спецификация терминала расширяет ZIOSpecDefault
  {спецификация определения =
    Suite ("терминал") (
      Test ("GetUserInput делегаты в консоли")
        {Проверка (Общаястрока) {ввод =>
          для {
            _ <- testconsole.feedlines (input)
            Результат <- terminal.getUserInput}
          assertTrue (результат == входные данные)}

      },
      тест("вывод делегатов на консоль")
        {проверка(Общая строка) { frame =>
          для {
            результат <- Terminal.
          отображение(рамка)} выход
        Утверждение истинности(результат == ())}
      }
    ).предоставитьСлой(TerminalLive.layer) @@ TestAspect.
тихий}
```

Некоторые важные вещи, которые стоит отметить:

- Каждому тесту需要一个TerminalLive環境来运行，并且TerminalLive использует стандартную службу Консоли от ZIO, чтобы быть способным выводить тексты в консоль.
- Отличие zio-test в том, что он не использует живые реализации стандартных служб ZIO, таких как Console, а тестовые реализации вместо этого. Например, TestConsole не только выводит тексты в консоль при вызове Console.printLine, но и хранит их в вектор TestConsole.output, который вы можете использовать для membuat aserciations. Также, вы можете имитировать ввод пользователя, вызывая TestConsole.feedLines.
- Также очень приятно то, что вы можете настроить поведение TestConsole так, чтобы тексты не выводились на консоль, а просто хранились в векторе. Для этого вы можете применить аспект к вашему набору тестов, более конкретно TestAspect.silent

Пишу GameModeSpec

В этом случае, давайте сосредоточимся на одном тесте вместо всего набора:

```
тест("возвращает состояние с добавленной фигурой и переходом хода к следующему игроку,
 если поле не занято") {val gameCommandParserMock: ULayer[GameCommandParser] =
   GameCommandParserMock.Parse(Утверждение.equalTo("положить 6"), Ожидание.value(Команда.положить(Поля.
Восток)))val gameLogicMock: ULayer[Логика игры] =
   Игровой логиксомock.putiece (
     Декларация.равно((gameState.board, Field.Восток, Шт.Cross)),
     Ожидание.значение(pieceAddedEastState.board)
   ) ++
     LogicИграmock
       .РезультатИгры(Утверждение.равенство(доскаДобавленногоВосточногоСостояния), Ожидание.значение(РезультатИгры.
     Проходящее)) ++ LogicИграMock.СледующийТур(Утверждение.равенство(Крест), Ожидание.значение(Шахматы.Ноль))
для {
  результат <- Игровой режим.
              process("положить 6",
              gameState).обеспечить(
                игровойКомандныйПарсерМок,
                GameViewMok.пустой,
                ПротивникAiMok.пустой,
                игроваяЛогикаМок,
                GameModeLive.слой)

}) выдать assertTrue(результат == pieceAddedEastState)
```

Тест, приведенный выше, предназначен для GameMode.process, и GameMode зависит от нескольких служб: GameCommandParser, GameView, OpponentAi и GameLogic. Поэтому, чтобы запустить тест, мы можем предоставить моки для этих служб, используя библиотеку zio-mock, и именно это происходит в приведенных выше строках. Сначала мы пишем мок для GameCommandParser:

```
импортировать zio.mock._

значение gameCommandParserMock: ULayer[GameCommandParser] = GameCommandParserMock.Parse(Утверждение.
 equalTo("поставить 6"), Математическое ожидание.значение(GameCommand.Поставить(Поле.Восток)))
```

Как вы, возможно, заметили, эта строка зависит от объекта GameCommandParserMock, и мы заявляем, что когда мы вызываем GameCommandParser.parse с входным значением, равным "put 6", он должен вернуть значение GameCommand.Put(Field.Восток). Кстати, GameCommandParserMock определен в файле mocks.scala:

```
импорт zio.mock._

@mockable[GameCommandParser]объект
GameCommandParserMock
```

Как показано выше, теперь мы используем аннотацию @mockable, входящую в библиотеку zio-mock. Эта аннотация - действительно nice макрос, который автоматически генерирует много шаблонного кода для нас, в противном случае нам пришлось бы написать его我们自己.

Кстати, есть еще что-то интересное: если мы более пристально рассмотрим это выражение:

```
GameCommandParserMock.Parse(Утверждение.equalTo("положить 6"), Ожидание.значение(GameCommand.Put(Поля.Восток)))
```

Оно возвращает значение типа Ожидание[GameCommandParser], но мы храним его как ULayer[GameCommandParser], и нет компиляционных ошибок… Причина в

то, что ZIO предоставляет неявную функцию Expectation#toLayer, которая преобразует Expectation[R] в ULayer[R]. Это означает, что, так как моки можно определить как ZLayers, мы можем легко предоставлять их для эффектов ZIO!

Я не буду углубляться в подробности о том, как работают моки ZIO. Однако, если вы хотите узнать больше о этом, вы можете ознакомиться с страницей документации ZIO.

Затем нам нужно написать мок для GameLogic:

```
val gamelogiccomock: ulayer [gamelogic] =
 gamelogicmock.putiece (
   Утверждение.equalto ((gamestate.board, field.east, piece.cross)),
   ожидание.
) ++
   ЛогикаИгрыМок
     .Результат игры (Assertion.Равенство(pieceaddedeststate.board), Ожидаемое значение(результат игры.Продолжается)) ++
   GameLogicMock.nextTurn(Утверждение.Равенство (Фигура.Крест), Ожидание.значение (Фигура.Ноль))
```

Идея здесь примерно та же, что и то, как мы определили gameCommandParserMock:

- Мок Defines as a ZLayer.
- Нам нужно определить объект GameLogicMock, так же, как мы это сделали выше для GameCommandParserMock.
- Для последовательного комбинирования ожиданий мы используем оператор ++ (который является всего лишь alias для метода Expectation#andThen).

Далее, нам нужно определить моки для GameView и OpponentAi. Однако, есть разница. Причина в том, что эти сервисы не вызываются на самом деле GameMode. process (функция, которую мы тестируем), поэтому эти моки должны заявлять, что мы ожидаем, что их не вызовут. К счастью, в текущей версии zio-mock есть легкий способ этого заявить. В основном, всего, что нам нужно сделать, это определить объекты GameViewMock и OpponentAiMock, как выше (используя аннотацию @mockable), и затем мы можем вызвать GameViewMock.empty и OpponentAiMock. empty, чтобы сгенерировать моки, которые нам нужны.

Затем нам нужно предоставить эти моки (не забывайте, что их можно рассматривать как обычные ZLayers) для выполнения теста:

```
для {
  Результат <- gamemode.process
                ("Поместить 6", Gamestate) .
                provide (
                  игроваяКомандаParserMock,
                  GameViewMock.пусто,
                  OpponentAiMock.пусто,
                  gameLogicMock,
                  GameModeLive.слой)

yield assertTrue(результат == pieceAddedEastState)
```

Сводка

В этом документе вы узнали, как написать приложение для игры в крестики-нолики с использованием ZLayers. Надеюсь, вы смогли оценить великолепную силу, которую ZLayer предоставляет для создания модульных и composable приложений в более доступном и понятном виде. В то же время, мы написали некоторые тесты и увидели, как легко использовать тестовые реализации стандартных служб ZIO (например, Console) или определять моковые окружения в виде ZLayers, которые могут предоставляться для тестов, чтобы сделать их выполняемыми.

Вы также узнали, как ZIO 2.0 помогает нам уменьшить множество шаблонных операций при подготовке окружения для ваших приложений. Благодаря этому, вам больше не нужно беспокоиться о горизонтальной и вертикальной композиции ZLayers: по умолчанию у вас автоматически происходит прокладка зависимости вашего приложения!

Я надеюсь, что концепции, связанные с типами данных ZEnvironment и ZLayer, теперь более понятны вам (если они weren't before), и что вы начнете использовать это знание в своих приложениях, чтобы сделать их extremely modular и composable!

Ссылки

- Репозиторий GitHub для этого документа
- Введение в программирование с ZIO функциональными эффектами, автор Jorge Vásquez
- Как написать командную строковую приложение с ZIO, автор Piotr Gołębiewski
- Как написать (абсолютно безблоковый) параллельный LRU Cache с ZIO STM, автор Jorge Vásquez
- Страница документации ZIO
- страница документации atto