It's TINKOFF

О Дивный Контекстный Мир Scala

Иван Лягаев

It's TINKOFF

Иван Лягаев | Ведущий Scala разработчик

i.lyagaev@tinkoff.ru

FireFoxIL

Ссылка на резентацию

План

- Стандартный подход - Контекст, как переменная
  - `ThreadLocal`
  - `MDC`
  - `cats.effect.IO, IOLocal`

- Функциональный подход - Контекст, как аргумент функции
  - `cats.data.Reader`
  - `cats.data.ReaderT`

- Tagless Final и Абстракции для работы с контекстом
  - `tofu.WithContext`
  - `tofu.WithProvide`
  - `tofu.logging.Logging`

Мы хотим понимать, что происходит с запросом на наш backend сервис

Иначе говоря, мы хотим отвечать на вопрос - Что происходило во время запроса?

Помогает нам с этим *контекст* запроса

Будем называть *контекстом* запроса любую вспомогательная информацию о запросе, доступную на протяжении всего времени обработки

Контекст может содержать:

- Идентификатор запроса
- Начало выполнение запроса
- Инициирующую систему
- и т.д.

Контекст позволяет обеспечить телеметрию запроса:

- логирование
- подсчет метрик
- трейсинг

Самый простой способ научиться передавать контекст - использовать переменную

```scala
var context = Map.empty[String, String]

def doLogic(): Unit = {
  println(s"${context.mkString("[", ",","]")} Logic")
}

def handleRequest(req: Request): Response = {
  val requestId: String = getOrCreate(req)
  context = Map("requestId" -> requestId)
  doLogic()
  Response.Ok
}
```

```scala
var context = Map.empty[String, String]

def doLogic(): Unit = {
  println(s"${context.mkString("[", ",","]")} Logic")
}

def handleRequest(req: Request): Response = {
  val requestId: String = getOrCreate(req)
  context = Map("requestId" -> requestId)
  doLogic()
  Response.Ok
}
```

```
1  var context = Map.empty[String, String]
2
3  def doLogic(): Unit = {
4    println(s"${context.mkString("[", ",","]")} Logic")
5  }
6
7  def handleRequest(req: Request): Response = {
8    val requestId: String = getOrCreate(req)
9    context = Map("requestId" -> requestId)
10   doLogic()
11   Response.Ok
12 }
```

```scala
var context = Map.empty[String, String]

def doLogic(): Unit = {
  println(s"${context.mkString("[", ",","]")} Logic")
}

def handleRequest(req: Request): Response = {
  val requestId: String = getOrCreate(req)
  context = Map("requestId" -> requestId)
  doLogic()
  Response.Ok
}
```

```
1  var context = Map.empty[String, String]
2
3  def doLogic(): Unit = {
4    println(s"${context.mkString("[", ",","]")} Logic")
5  }
6
7  def handleRequest(req: Request): Response = {
8    val requestId: String = getOrCreate(req)
9    context = Map("requestId" -> requestId)
10   doLogic()
11   Response.Ok
12 }
```

```
[requestId -> 12345] Logic
```

Минусы:

- В многопоточной среде такой подход не будет работать
- Переменная должна быть проинициализирована значением до начала обработки запроса
- Происходит явная работа с контекстом на уровне логгирования

Для многопоточной среды в `java` принято использовать `ThreadLocal` - локальные переменные для каждого потока

На этом подходе выстроен инструмент для логирования - *Mapped Diagnostic Context* (MDC)

Он представляет из себя изменяемую структуру вида `Map[String, String]`, которая хранится в `ThreadLocal`

Каждый поток имеет свое значение MDC, которое он может независимо изменять

MDC присутствует во всех библиотеках для логирования, в частности - в `slf4j` фасаде

```scala
import org.slf4j.{Logger, LoggerFactory, MDC}
val logger: Logger = LoggerFactory.getLogger("Logger")

def doLogic(): Unit = {
  logger.info("Doing some logic")
}

def handleRequest(req: Request): Response = {
  val requestId: String = getOrCreate(req)
  MDC.put("requestId", requestId)
  doLogic()
  Response.Ok
}
```

```scala
import org.slf4j.{Logger, LoggerFactory, MDC}
val logger: Logger = LoggerFactory.getLogger("Logger")

def doLogic(): Unit = {
  logger.info("Doing some logic")
}

def handleRequest(req: Request): Response = {
  val requestId: String = getOrCreate(req)
  MDC.put("requestId", requestId)
  doLogic()
  Response.Ok
}
```

```scala
import org.slf4j.{Logger, LoggerFactory, MDC}
val logger: Logger = LoggerFactory.getLogger("Logger")

def doLogic(): Unit = {
  logger.info("Doing some logic")
}

def handleRequest(req: Request): Response = {
  val requestId: String = getOrCreate(req)
  MDC.put("requestId", requestId)
  doLogic()
  Response.Ok
}
```

```scala
import org.slf4j.{Logger, LoggerFactory, MDC}
val logger: Logger = LoggerFactory.getLogger("Logger")

def doLogic(): Unit = {
  logger.info("Doing some logic")
}

def handleRequest(req: Request): Response = {
  val requestId: String = getOrCreate(req)
  MDC.put("requestId", requestId)
  doLogic()
  Response.Ok
}
```

```
[requestId=12345] Doing some logic
```

`MDC` предоставляет удобное API:

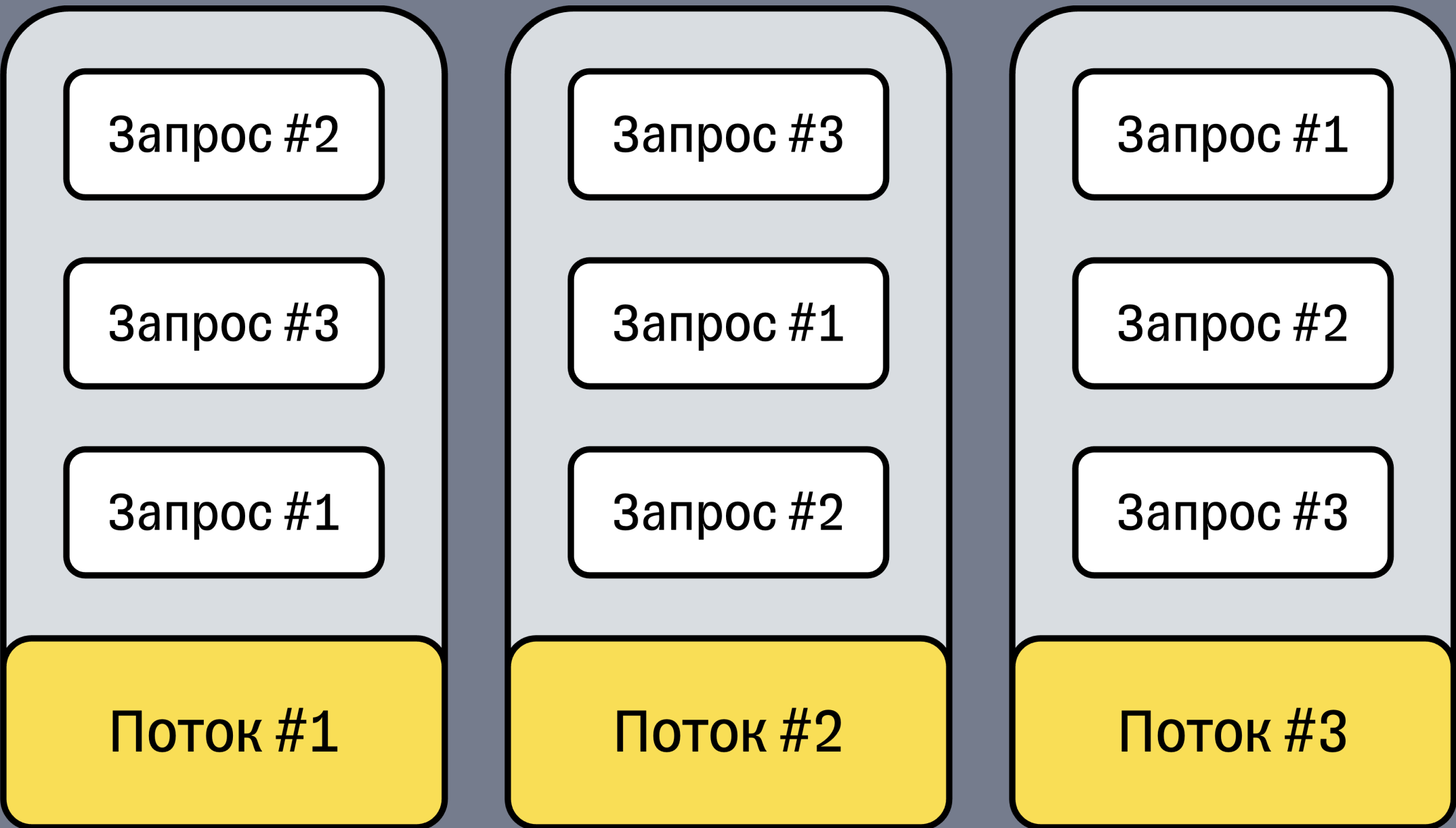
- Неявно прокидываем контекст запроса через `ThreadLocal`
- Задаем значения контекста в одном месте при старте обработки запроса
- Автоматически подхватываем контекста в процессе логгирования

Будем считать, что именно такого API мы и хотим добиться для всех инструментов для работы с контекстом

MDC также отлично работает в многопоточной среде, когда мы выделяем на обработку запроса отдельный поток


Запрос #1
Запрос #2
Запрос #3
Поток #1
Поток #2
Поток #3

К сожалению, такой подход не может существовать с современными реактивными фреймворками, где каждый запрос может исполняться на нескольких потоках


Запрос #2
Запрос #3
Запрос #1
Поток #1
Запрос #3
Запрос #1
Запрос #2
Поток #2
Запрос #1
Запрос #2
Запрос #3
Поток #3

К счастью, в Scala есть множество библиотек для реализации реактивного подхода.

Примеры: `cats-effect`, `monix`, `zio`

И каждая из таких библиотек дает возможность работы с контекстом "из коробки"

Рассмотрим, как пример - библиотеку `cats-effect`

`cats-effect` реализует рантайм для приложений, используя концепцию файберов

Каждый запрос моделируется, как отдельное вычисление - файбер

Файберы могут порождать дочерние вычисления, которые могут исполняться конкурентно

Файбер - облегченный логический поток, из-за чего мы можем иметь в системе в несколько раз больше файберов, чем системных тредов

При этом каждый файбер может иметь свой контекст - произвольную изменяющуюся структуру данных, которая автоматически передается дочерним файберам

Задача - реализовать простой web-сервер для отправки сообщений между пользователями

Работу с контекстом будем разбирать на примере логгирования

```scala
trait IOUserRepository {
  def getUser(id: UserId): IO[User]
}

trait IOMessageSender {
  def send(from: User, to: User, msg: Message): IO[Unit]
}
```

`IO` имеет метод `flatMap`

```scala
trait IO[+A] {
  def flatMap[B](f: A => IO[B]): IO[B]
}
```

Он позволяет последовательно композировать разные IO вычисления

```scala
class IOMessageService(
  sender: IOMessageSender,
  users: IOUserRepository
 ) {
  def send(from: UserId, to: UserId, msg: Message): IO[Unit] = {
    users.getUser(from).flatMap { fromUser =>
      users.getUser(to).flatMap { toUser =>
        sender.send(fromUser, toUser, msg)
      }
    }
}
```

Можно использовать специальный синтаксический сахар - for-comprehension

```
class IOMessageService(
  sender: IOMessageSender,
  users: IOUserRepository
 ) {
  def send(from: UserId, to: UserId, msg: Message): IO[Unit] =
    for {
      fromUser <- users.getUser(from)
      toUser   <- users.getUser(to)
      _        <- sender.send(fromUser, toUser, msg)
    } yield ()
}
```

Для простоты работы с контекстом и сайд-эффектами - проще объявить свой фасад для логгирования возвращающий IO

```scala
trait IOLogger {
  def info(input: String): IO[Unit]
}
```

```scala
import cats.effect.{IO, IOLocal}

class LoggerImpl(
  local: IOLocal[Map[String, String]]
) extends IOLogger {

  def info(input: String): IO[Unit] =
   for {
     ctx <- local.get
     _   <- IO.println(s"[INFO] ${ctx.mkString("[", ",", "]")} $input")
   } yield ()
}
```

```scala
import cats.effect.{IO, IOLocal}

class LoggerImpl(
  local: IOLocal[Map[String, String]]
) extends IOLogger {

  def info(input: String): IO[Unit] =
   for {
     ctx <- local.get
     _   <- IO.println(s"[INFO] ${ctx.mkString("[", ",", "]")} $input")
   } yield ()
}
```

```scala
import cats.effect.{IO, IOLocal}

class LoggerImpl(
  local: IOLocal[Map[String, String]]
) extends IOLogger {

  def info(input: String): IO[Unit] =
   for {
     ctx <- local.get
     _   <- IO.println(s"[INFO] ${ctx.mkString("[", ",", "]")} $input")
   } yield ()
}
```

```scala
class LoggingIOUserRepository(
  delegate: IOUserRepository,
  logger: IOLogger
 ) extends IOUserRepository {

  def getUser(id: UserId): IO[User] =
    for {
      _    <- logger.info(s"Getting user=$id")
      user <- delegate.getUser(id)
    } yield user
  }
}
```

```scala
class LoggingIOMessageSender(
  delegate: IOMessageSender,
  logger: IOLogger
 ) extends IOMessageSender {
  def send(from: User, to: User, msg: Message): IO[Unit] =
    for {
      _ <- logger.info(s"Sending msg=$msg")
      _ <- delegate.send(from, to, msg)
      _ <- logger.info(s"Sent msg=$msg")
    } yield ()
}
```

```scala
trait IOHttpHandler {
  def handle(req: Request): IO[Response]
}
```

```scala
class IOMessageHttpHandler(
  srv: IOMessageService,
  local: IOLocal[Map[String, String]]
 ) extends IOHttpHandler {

  def handle(req: Request): IO[Response] = {
    for {
      requestId <- getOrCreate(req)
      _ <- local.set(Map("requestId" -> requestId))
      (from, to, msg) <- parse(req)
      _ <- srv.send(from, to, msg)
    } yield Response.Ok
  }
}
```

Пример логов:

```
[INFO] [requestId -> "12345"] Getting user=1
[INFO] [requestId -> "12345"] Getting user=2
[INFO] [requestId -> "12345"] Sending msg=Message(Privet)
[INFO] [requestId -> "12345"] Sent msg=Message(Privet)
```

`IOLocal` позволяет работать с типизированным контекстом, который передается на все вычисление

Также, он позволяет неявно прокидывать контекст внутри одного файбера

Все вроде бы все хорошо, но остается пара проблем:

- Контекст должен быть чем-то проинициализирован. Отсюда нет гарантий, что его не забудут проставить
- Есть вероятность, что поведет себя не так,как мы ожидаем. (Issue #3100)

И главная проблема всех разобранных подходов - они неразрывно связаны с рантаймом

А рантаймы довольно часто меняются

Подход с `MDC` и `IOLocal` строился на подходе, что у нас есть неявная переменная, которую мы можем спокойно изменять и читать

Есть и другой подход - передавать контекст через аргументы функции

```scala
case class RequestContext(id: String)

trait ArgUserRepository {
  def getUser(id: UserId)(ctx: RequestContext): User
}

trait ArgMessageSender {
  def send(from: User, to: User, msg: Message)(ctx: RequestContext): Unit
}
```

```scala
class LoggingArgUserRepository(
  delegate: ArgUserRepository
) extends ArgUserRepository {

  def getUser(id: UserId)(ctx: RequestContext): User = {
    println(s"$ctx Getting user=$id")
    delegate.getUser(id)(ctx)
  }
}
```

```
1  class LoggingArgMessageSender(
2    delegate: ArgMessageSender
3  ) extends ArgMessageSender {
4
5    def send(from: User, to: User, msg: Message)
6            (ctx: RequestContext): Unit = {
7      println(s"$ctx Sending msg=$msg")
8      delegate.send(from, to, msg)(ctx)
9      println(s"$ctx Sent msg=$msg")
10   }
11 }
```

```scala
class ArgMessageService(
  sender: ArgMessageSender,
  users: ArgUserRepository
 ) {
  def send(from: UserId, to: UserId, msg: Message)(ctx: RequestContext): U
    val fromUser = users.getUser(from)(ctx)
    val toUser   = users.getUser(to)(ctx)

    sender.send(fromUser, toUser, msg)(ctx)
  }
}
```

```scala
trait HttpHandler {
  def handle(req: Request): Response
}

class ArgMessageHttpHandler(srv: ArgMessageService)
  extends HttpHandler {

  def handle(req: Request): Response = {
    val (from, to, msg, ctx) = parse(req)
    srv.send(from, to, msg)(ctx)
    Response.Ok
  }
}
```

Теперь давайте возвращать не значение, а функцию от контекста

```scala
trait ArgUserRepository {
  def getUser(id: UserId): RequestContext => User
}

trait ArgMessageSender {
  def send(from: User, to: User, msg: Message): RequestContext => Unit
}
```

```scala
class ArgMessageService(
  sender: ArgMessageSender,
  users: ArgUserRepository
 ) {
  def send(
    from: UserId,
    to: UserId,
    msg: Message
  ): RequestContext => Unit = { ctx =>
    val fromUser = users.getUser(from)(ctx)
    val toUser   = users.getUser(to)(ctx)

    sender.send(fromUser, toUser, msg)(ctx)
  }
}
```

```scala
class ArgMessageHttpHandler(srv: ArgMessageService)
  extends HttpHandler {

  def handle(req: Request): Response = {
    val (from, to, msg, ctx) = parse(req)
    srv.send(from, to, msg)(ctx)
    Response.Ok
  }
}
```

Если приглядеться к функции `E => A`, то можно увидеть Reader монаду

Она нам позволит уйти от явного проброса контекста

```
1 case class Reader[E, A](run: E => A)
```

Мы точно также можем объявить `flatMap` метод

```scala
case class Reader[E, A](run: E => A) {
  def flatMap[B](f: A => Reader[E, B]): Reader[E, B] =
    Reader(e => f(run(e)))
}
```

```scala
type Context[A] = Reader[RequestContext, A]

object Context {
  def ask: Context[RequestContext] =
    Reader(a => a)

  def lift[B](b: B): Context[B] =
    Reader(_ => b)
}
```

```scala
case class Reader[E, A](run: E => A)
type Context[A] = Reader[RequestContext, A]

trait ArgUserRepository {
  def getUser(id: UserId): Context[User]
}

trait ArgMessageSender {
  def send(from: User, to: User, msg: Message): Context[Unit]
}
```

```scala
class LoggingArgUserRepository(
  delegate: ArgUserRepository
) extends ArgUserRepository {
  def getUser(id: UserId): Context[User] =
    for {
      ctx  <- Context.ask
      _    <- Context.lift(
        println(s"$ctx Getting user=$id")
      )
      user <- delegate.getUser(id)
    } yield user
}
```

```scala
class ArgMessageService(
  sender: ArgMessageSender,
  users: ArgUserRepository
 ) {
  def send(from: UserId, to: UserId, msg: Message): Context[Unit] =
    for {
      fromUser <- users.getUser(from)
      toUser   <- users.getUser(to)
      _ <- sender.send(fromUser, toUser, msg)
    } yield ()
}
```

Теперь, давайте перейдем к асинхронной обработке

Для этого нам понадобится `ReaderT` и типы высшего порядка

```scala
case class ReaderT[F[_], E, B](run: E => F[A]) {
  def flatMap[B](f: A => ReaderT[F, E, B])(
    implicit monad: Monad[F]
  ): ReaderT[F, E, B] =
    ReaderT { e =>
      monad.flatMap(run(e))(f(_).run(e))
    }
}

trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

```scala
case class ReaderT[F[_], E, B](run: E => F[A]) {
  def flatMap[B](f: A => ReaderT[F, E, B])(
    implicit monad: Monad[F]
  ): ReaderT[F, E, B] =
    ReaderT { e =>
      monad.flatMap(run(e))(f(_).run(e))
    }
}

trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

```scala
case class ReaderT[F[_], E, B](run: E => F[A]) {
  def flatMap[B](f: A => ReaderT[F, E, B])(
    implicit monad: Monad[F]
  ): ReaderT[F, E, B] =
    ReaderT { e =>
      monad.flatMap(run(e))(f(_).run(e))
    }
}

trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

```scala
case class ReaderT[F[_], E, B](run: E => F[A]) {
  def flatMap[B](f: A => ReaderT[F, E, B])(
    implicit monad: Monad[F]
  ): ReaderT[F, E, B] =
    ReaderT { e =>
      monad.flatMap(run(e))(f(_).run(e))
    }
}

trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

```scala
type ContextIO[B] = ReaderT[IO, RequestContext, B]

object ContextIO {
  def ask: ContextIO[RequestContext] =
    ReaderT(a => IO.pure(a))

  def lift[B](io: IO[B]): ContextIO[B] =
    ReaderT(_ => io)
}
```

```scala
trait ContextIOUserRepository {
  def getUser(id: UserId): ContextIO[User]
}

trait ContextIOMessageSender {
  def send(from: User, to: User, msg: Message): ContextIO[Unit]
}
```

```scala
trait ContextIOLogger {
  def info(input: String): ContextIO[Unit]
}
```

```scala
class ContextIOLoggerImpl
  extends ContextIOLogger {

  def info(input: String): ContextIO[Unit] =
   for {
     ctx <- ContextIO.ask
     _   <- ContextIO.lift(
       IO.println(s"[INFO] [requestId -> ${ctx.id}] $input")
     )
   } yield ()
}
```

```scala
class LoggingContextIOUserRepository(
  delegate: ContextIOUserRepository,
  logger: ContextIOLogger
 ) extends ContextIOUserRepository {

  def getUser(id: UserId): ContextIO[User] =
    for {
      _    <- logger.info(s"Getting user=$id")
      user <- delegate.getUser(id)
    } yield user
  }
}
```

```scala
class LoggingContextIOMessageSender(
  delegate: ContextIOMessageSender,
  logger: ContextIOLogger
 ) extends ContextIOMessageSender {
  def send(from: User, to: User, msg: Message): ContextIO[Unit] =
    for {
      _ <- logger.info(s"Sending msg=$msg")
      _ <- delegate.send(from, to, msg)
      _ <- logger.info(s"Sent msg=$msg")
    } yield ()
}
```

```scala
trait IOHttpHandler {
  def handle(req: Request): IO[Response]
}

class IOMessageHttpHandler(
  srv: ContextIOMessageService
) extends IOHttpHandler {

  def handle(req: Request): IO[Response] = {
    for {
      (from, to, msg, ctx) <- parse(req)
      _ <- srv.send(from, to, msg).run(ctx)
    } yield Response.Ok
  }
}
```

`Reader` и `ReaderT` хорошие абстракции для работы с контекстом

- Они дают гарантию на передачу контекста, нужно явно "запустить" Reader
- Если контекст неизменяемый, то есть строгие гарантии поведения в многопоточной среде

Их имплементации доступны в библиотеке `cats` (не путать с `cats-effect`)

Давайте чуть подробнее поглядим на все реализации наших интерфейсов

```scala
trait IOUserRepository {
  def getUser(id: UserId): IO[User]
}

trait ArgUserRepository {
  def getUser(id: UserId): Context[User]
}

trait ContextIOUserRepository {
  def getUser(id: UserId): ContextIO[User]
}
```

Все они отличаются одним параметром - контейнером возвращаемого значения

Давайте выведем его в параметр интерфейса

```scala
trait UserRepository[F[_]] {
  def getUser(id: UserId): F[User]
}

type IOUserRepository  = UserRepository[IO]
type ArgUserRepository = UserRepository[Context]
```

```scala
trait MessageSender[F[_]] {
  def send(from: User, to: User, msg: Message): F[Unit]
}

trait HttpHandler[F[_]] {
  def handle(req: Request): F[Response]
}
```

```scala
import cats.Monad
import cats.syntax.flatMap._
import cats.syntax.functor._

class MessageService[F[_]: Monad](
  sender: MessageSender[F],
  users: UserRepository[F]
) {
  def send(from: UserId, to: UserId, msg: Message): F[Unit] =
    for {
      fromUser <- users.getUser(from)
      toUser   <- users.getUser(to)
      _        <- sender.send(fromUser, toUser, msg)
    } yield ()
}
```

Такая техника называется - *Tagless Final*

```scala
trait LoggerF[F[_]] {
  def info(input: String): F[Unit]
}
```

```scala
class LoggingUserRepository[F[_]: Monad](
  delegate: UserRepository[F],
  logger: LoggerF[F]
) {
  def getUser(id: UserId): F[User] =
    for {
      _    <- logger.info(s"Getting user=$user")
      user <- delegate.getUser(id)
    } yield user
}
```

```scala
import cats.Monad
import cats.syntax.flatMap._
import cats.syntax.functor._

class LoggingMessageSender[F[_]: Monad](
  delegate: MessageSender[F],
  logger: LoggerF[F]
) {
  def send(from: User, to: User, msg: Message): F[Unit] =
    for {
      _ <- logger.info(s"Sending msg=$msg")
      _ <- delegate.send(from, to, msg)
      _ <- logger.info(s"Sent msg=$msg")
    } yield ())
}
```

```scala
trait RequestParser[F[_]] {
  def parse(req: Request): F[(UserId, UserId, Message)]
}

class MessageHttpHandler[F[_]: Monad](
  parser: RequestParser[F]
  srv: MessageService[F]
) {
  def handle(req: Request): F[Response] =
    for {
      (from, to, msg) <- parser.parse(req)
      _ <- srv.send(from, to, msg)
    } yield Response.Ok
}
```

```scala
type Id[A] = A

class SyncLogger extends LoggerF[Id] {
  def info(input: String): Id[Unit] =
    println(input)
}

class IOLogger extends LoggerF[IO] {
  def info(input: String): IO[Unit] =
    IO.println(input)
}
```

```scala
val users: UserRepository[IO] = ???
val sender: MessageSender[IO] = ???
val parser: RequestParser[IO] = ???

val logger: LoggerF[IO] = new IOLogger

val handler: HttpHandler[IO] =
  new MessageHttpHandler[IO](
    parser,
    new MessageService[IO](
      new LoggingUserRepository(users, logger),
      new LoggingMessageSender(sender, logger)
    )
  )
```

Пример логов:

```
Getting user=1
Getting user=2
Sending msg=Message(Privet)
Sent msg=Message(Privet)
```

Тайпкласс для получения контекста

```scala
trait WithContext[F[_], Context] {
  def ask: F[Context]
}
```

```scala
class ContextLoggerF[F[_]: Monad](
  logger: LoggerF[F]
)(implicit context: WithContext[F, RequestContext]) {

  def info(input: String): F[Unit] =
    for {
      ctx <- context.ask
      _   <- logger
        .info(s"[requestId -> ${ctx.id}] $input")
    } yield ()
}
```

Тайпкласс для предоставления контекста

```scala
trait WithProvide[F[_], G[_], Context] {
  def runContext[A](fa: F[A])(ctx: Context): G[A]
}
```

```scala
implicit def derive[F[_], C]: WithProvide[ReaderT[F, C, *], F, C] =
  new WithProvide[ReaderT[F, C, *], F, C] {
    def runContext[A](fa: ReaderT[F, C, A])(ctx: C): F[A] =
      fa.run(ctx)
}
```

```scala
trait ContextParser[G[_]] {
  def parseContext(req: Request): G[RequestContext]
}
```

```scala
class ContextHttpHandler[F[_], G[_]: Monad](
  contextParser: ContextParser[G]
  delegate: HttpHandler[F]
)(implicit provide: WithProvide[F, G, RequestContext])
  extends HttpHandler[G] {

  def handler(req: Request): G[Response] =
    for {
      ctx <- parser.parseContext(req)
      r   <- provide.runContext(delegate.handle(req))(ctx)
    } yield r
}
```

```scala
type ContextIO[B] = ReaderT[IO, RequestContext, B]

class LiftLogger(ioLogger: LoggerF[IO]) extends
  LoggerF[ContextIO] = {

  def info(input: String): ContextIO[Unit] =
    ContextIO.lift(ioLogger.info(input))
}
```

```scala
class LiftUserRepository(
  users: UserRepository[IO]
) extends UserRepository[ContextIO] {
  def getUser(id: UserId): ContextIO[User] =
    ContextIO.lift(users.getUser(id))
}
```

```scala
val cUsers: UserRepository[ContextIO] =
  new LiftUserRepository(users)

val cSender: MessageSender[ContextIO] =
  new LiftMessageSender(sender)

val cParser: RequestParser[ContextIO] =
  new LiftRequestParser(parsers)

val cLogger: LoggerF[ContextIO] =
  new ContextLogger[ContextIO](new LiftLogger(logger)))
```

```scala
val contextParser: ContextParser[IO] = ???

val ctxHandler = new ContextHttpHandler[IO, ContextIO](
  contextParser,
  new MessageHandler[ContextIO](
    cParser,
    new MessageService[ContextIO](
      new LoggingUserRepository(cUsers, cLogger),
      new LoggingMessageSender(cSender, cLogger)
    )
  )
)
```

Пример логов:

```
[requestId -> 12345] Getting user=1
[requestId -> 12345] Getting user=2
[requestId -> 12345] Sending msg=Message(Privet)
[requestId -> 12345] Sent msg=Message(Privet)
```

На первый взгляд - куча бойлер плейта

Но, при использовании *Tagless Final* мы смогли добавить проброс контекст в приложение, не изменяя ни одной строчки уже существующего кода

И операции с поднятием (`Lift`) интерфейсов на самом деле хорошо покрываются макросами и другими абстракциями

Рассмотренные абстракции уже поставляются, как библиотеки и они не привязаны к конкретному рантайму.

- **`cats`** - предоставляет `Monad`
- **`tofu-kernel`** - предоставляет `WithContext`, `WithProvide`
- **`tofu-logging`** - предоставляет `Logging`

Выводы

Выбор инструмента упирается в ваши потребностей

- Если вас не пугает привязка к рантайму
- Если у вас есть потребность обогащать контекст во время выполнения запроса

То стоит использовать библиотеки и их встроенные механизмы по работе с контекстом: `cats-effect`, `zio` и `monix`

- Если вас пугает привязка к рантайму
- Если вам не нужно обогащать контекст

То стоит использовать `Reader` монаду из библиотеки `cats`

- Если вам снится в кошмарах привязка к рантайму,
- Если вы хотите получить максимальную гибкость от кода

То стоит использовать `Tagless Final` подход и тайпклассы из библиотек `cats` и `tofu`

Вопросы?