http4sでのHello World

公式ドキュメント: https://http4s.org/

Github: https://github.com/http4s/http4s

目的

最初から理解するのは難しいので、最初はこんな感じで書いてアプリケーション作っていくのねえ〜ぐらいの感覚で大丈夫です。

http4を使う上でCats Effectがどのように絡んでくるのかとかも見ていただければと思います。

http4sとは

> Typeful, functional, streaming HTTP for Scala

Scalaのためのタイプフル、関数的、ストリーミングHTTP

http4sはtypelevelエコシステムのシンプルなhttp(サーバー、クライアント)ライブラリです。
IO モナドなどのエフェクトライブラリと組み合わせて使います.

サーバー/クライアント

http4sのサーバー/クライアントは以下表にあるものから選んで構築を行うことができます。

Backend	Platform	Http Client	Http Server	Websocket Client	Websocket Server	Proxy support (Client)
Ember	JDK 8+ / Node.js 16+ / Native	✅	✅	❌	✅	❌
Blaze	JDK 8+	✅	✅	❌	✅	❌
Netty	JDK 8+	✅	✅	✅	✅	✅
JDK Http Client	JDK 11+	✅	❌	✅	❌	✅
Servlet	JDK 8+	❌	✅	❌	❌	❌
DOM	Browsers	✅	❌	✅	❌	❌
Feral	Serverless	❌	✅	❌	❌	❌

以前までBlazeがメインサーバーでしたが、現在はEmberというものがメインで開発を行っているサーバーになります。

※サンプルコードや記事ではBlazeServerを使うものが多いですが、最新はEmberServerなので使用する時は注意してください。

構成

http4はざっくり言うとパスごとに処理を書くサービスと、アプリケーションで実行するサービスをまとめたルーターと、ルーター(アプリケーション)を受け取りアプリケーションを実行するサーバーで構成されています。

サービス: HttpRoutes
ルーター: Router/HttpApp
サーバー: EmberServer (※ Emberを使用した場合)

依存関係の追加

依存関係に以下2つを追加します。

```
val http4sVersion = "0.23.16"
libraryDependencies ++= Seq(
  "org.http4s" %% "http4s-dsl" % http4sVersion,
  "org.http4s" %% "http4s-ember-server" % http4sVersion
)
```

サービス

http4sのサービスを構築するためのHttpRoutesは、Kleisliの単純なエイリアスです。

```
Kleisli[[T] =>> OptionT[F, T], Request[F], Response[F]]
```

KleisliはRequest[F] => F[Response[F]]の便利なラッパーに過ぎず、Fは効果的な操作です。

※Fは今回Cats EffectのIOを使用しますが、IOに関してはCats Effectの章で説明します。

※ =>> はScala3で追加されたType Lambdaというものです。

Type Lambdaとは

他の型をパラメーターとして受け取る型コンストラクターを型パラメーターとして使用したい場合は以下のように定義します。

```
trait TypeLambda[F[_]]
```

使用する時は以下のように定義します。

```
type F1 = TypeLambda[Option]
type F2 = TypeLambda[List]
```

ただMapのような2つの型パラメーターを必要とするものに関してはエラーとなってしまいます。

```
type F3 = TypeLambda[Map]
```

```
[error] -- Error: ...
[error] 14 |type F3 = TypeLambda[Map]
[error]    |                     ^^^
[error]    |        Type argument Map does not have the same kind as its bound [_$1]
[error] one error found
```

これは `F[_]` という形で型パラメータの数を一つとして定義しているため、型パラメータを二つ取る物を渡せないようになってしまっているからです。

つまり、この場合最低でも1つの型は決まっていないといけないということです。

```
type MapA[A] = Map[Int, A]

type F3 = TypeLambda[MapA]
```

けど、都度この型エイリアスを書くのもめんどくさいですよね。。。

2つのパラメーターを受け取るようなものを作れば、まあいけるのだが。。。
用途が限定されてしまいます。。。

```
trait TypeLambda1[F[_, _]]
```

この問題を解決するために、無名の型を定義してそれを渡し、足りない型を補う方法が開発されました。

それがType Lambdaと呼ばれるものです。
(Scala2だとkinder projectと呼ばれるライブラリ？がありました。`*`で書くやつ)

```
type F3 = TypeLambda[[A] =>> Map[Int, A]]
```

このType Lambdaを使えば以下のように型エイリアスや専用の型パラメーターを持ったものを用意することなく実装を行うことができるので、とても便利です。

```
type F1 = TypeLambda[Option]
type F2 = TypeLambda[List]
type F3 = TypeLambda[[A] =>> Map[Int, A]]
```

こういう書き方もできたりします。

```
type TypeLambdaMap = [A] =>> [B] =>> Map[A, B]

val typeLambda: TypeLambdaMap[Int][String] = Map(1 -> "")
```

Kleisliとは

Kleisliは `A => F[B]` という型の関数に対する特殊なラッパー

- Catsのドキュメント
- 圏論的に学びたい人はこちらを参照

まず関数には合成することができるという特性があります。

関数A => Bと関数B => Cがあれば、それらを組み合わせて新しい関数A => Cを作ることができる。

Catsドキュメントからの引用

```
val twice: Int => Int =
  x => x * 2

val countCats: Int => String =
  x => if (x == 1) "1 cat" else s"$x cats"

val twiceAsManyCats: Int => String =
  twice andThen countCats
```

```
twiceAsManyCats(1) // "2 cats"
// res0: String = "2 cats"
```

実装しているとモナド値を返すような関数を作ることはたくさんあると思います。
これを合成することはできるでしょうか？

```
val parse: String => Option[Int] =
  s => if (s.matches("-?[0-9]+")) Some(s.toInt) else None

val reciprocal: Int => Option[Double] =
  i => if (i != 0) Some(1.0 / i) else None
```

もし合成をしたいとなっても愚直にやる必要が出てきてしまいます。

純粋に `andThen` を使って合成することができず、何か処理が必要となってしまいます。

```
val parseReciprocal: String => Option[Double] =
  parse.andThen(_.flatMap(reciprocal))
```

Kleisliを使えばモナド値が含まれていても簡単に合成することができる。
実装自体もKleisliで囲むだけで良い

```
val parse: Kleisli[Option, String, Int] =
  Kleisli((s: String) => if s.matches("-?[0-9]+") then Some(s.toInt) else None)

val reciprocal: Kleisli[Option, Int, Double] =
  Kleisli((i: Int) => if i != 0 then Some(1.0 / i) else None)

val parseReciprocal: Kleisli[Option, String, Double] =
  parse andThen reciprocal
```

SGPのチャットボットはKleisliを使って、用途ごとに処理を分けて最後に合成を行って1つの関数にしています。

サービス

HttpRoutesを使用した最小サービス

これはメソッドがGETでパスが `/hello/takapi` の場合、200のレスポンスで `Hello, takapi.` を返すサービスを構築したことになる。

```
val helloWorldService = HttpRoutes.of[IO] {
  case GET -> Root / "hello" / name => Ok(s"Hello, $name.")
}
```

HttpRoutes.of[IO]

`of[F[_]: Monad]` メソッドは、部分関数を受け取りそれをHttpRoutesに昇格させるためのメソッドです。

```
def of[F[_]: Monad](pf: PartialFunction[Request[F], F[Response[F]]]): HttpRoutes[F] =
  Kleisli(req => OptionT(Applicative[F].unit >> pf.lift(req).sequence))
```

`Applicative[F].unit` は `pf.lift(req).sequence` の結果をFにliftしていると思ってください。

PartialFunctionとは

MapやOptionのcollectメソッドの内部などで使われているもの

Aに対してBを返すような関数ですが、必ずしもタイプAのすべての値を含むとは限らないことに注意

```
trait PartialFunction[-A, +B] extends (A) => B
```

つまり、簡単にいうと特定の引数のみ処理する関数。

普通の関数と同じように呼び出すが、パターンにマッチしない場合はMatchErrorになる。

```
val pf: PartialFunction[Int, String] = {
  case 1 => "first"
  ... // 複数定義可能
}

println(pf(1)) // => first
println(pf(2)) // => MatchError
```

PartialFunctionは合成することができる。

```
val pf1: PartialFunction[Int, String] = { case 1 => "first" }
val pf2: PartialFunction[Int, String] = { case 2 => "second" }

// pf1にマッチしない場合はpf2を適用するPartialFunctionを生成
val pf3 = pf1 orElse pf2

println(pf3(1)) // => first
println(pf3(2)) // => second
println(pf3(3)) // => MatchError
```

`of` メソッドの中で使われている `lift` はPartialFunctionのキーに該当してたらSomeに包んで値を返し、なければNoneを返すメソッド。

```
def of[F[_]: Monad](pf: PartialFunction[Request[F], F[Response[F]]]): HttpRoutes[F] =
  Kleisli(req => OptionT(Applicative[F].unit >> pf.lift(req).sequence))
```

HttpRoutes

つまりHttpRoutesの構築は、リクエストを受け取りそのリクエストに一致したものがあれば
その値(Response)を返す関数だということがわかります。

つまり

KleisleもPartialFunctionも合成可能。

つまり、HttpRoutesも合成ができるということ

```
  val helloWorldService: HttpRoutes[IO] = HttpRoutes.of[IO] {
    case GET -> Root / "hello" / name => Ok(s"Hello, $name.")
  }

  val crudService: HttpRoutes[IO] = HttpRoutes.of[IO] {
    case GET -> Root / "crud" => Ok("CRUD")
  }

  val composeService: HttpRoutes[IO] = helloWorldService <+> crudService
```

Requestなんてないやん

ここで最小の実装をもう1度見て見ましょう。

```
val helloWorldService = HttpRoutes.of[IO] {
  case GET -> Root / "hello" / name => Ok(s"Hello, $name.")
}
```

これを見てパッと見どれがRequestかわかる人は少ないと思います。

実装は以下のようになっているのですが、ではどれがRequestなのでしょうか？

```
final class Request[F[_]] private (
  val method: Method,
  val uri: Uri,
  val httpVersion: HttpVersion,
  val headers: Headers,
  val body: EntityBody[F],
  val attributes: Vault,
) ...
```

以下画像の赤枠で囲った部分が全てRequestになります。

```
val helloWorldService = HttpRoutes.of[IO] {
  case GET -> Root / "hello" / name => Ok(s"Hello, $name.")
}
```

これ全部がRequestになります。

どういうこと？

なんでこれがRequestになるの？と思ったかもしれません。

なので1つずつ分解していきましょう。

Method -> Path

まずは `->` の部分に関して見ていきます。
これは結論から言うと、RequestをMethodとPathに分解するunapplyメソッドを持った抽出子objectです。

実装は以下のようになっています。

```
object -> {
  def unapply[F[_]](req: Request[F]): Some[(Method, Path)] =
    Some((req.method, req.pathInfo))
}
```

※applyメソッドが引数を取りオブジェクトを作るコンストラクタであるように、unapplyは1つのオブジェクトを受け取り引数を返そうとするものです。

Path / Path

次は `/` の部分に関して見ていきます。

こちらはRequestのPathを細かく分解するunapplyメソッドを持った抽出子objectです。

```
object / {
  def unapply(path: Path): Option[(Path, String)] =
    if (path.endsWithSlash)
      Some(path.dropEndsWithSlash -> "")
    else
      path.segments match {
        case allButLast :+ last if allButLast.isEmpty =>
          if (path.absolute)
            Some(Root -> last.decoded())
          else
            Some(empty -> last.decoded())
        case allButLast :+ last =>
          Some(Path(allButLast, absolute = path.absolute) -> last.decoded())
        case _ => None
      }
}
```

`/` は `->` でRequestをMethodとPathに分解した後に、Pathを更に分解するものになります。

unapply x パターンマッチの恩恵を受けずに書くと？

最小サービスの実装をunapply x パターンマッチの恩恵を受けずに書くと以下のようになります。

愚直に書くとなんとなくやってることがわかったんじゃないでしょうか？

```
def requestToResponse(request: Request[IO]): IO[Response[IO]] =
  ->.unapply(request) match
    case Some((method, path1)) if method.name == "GET" => /.unapply(path1) match
      case Some((path2, str)) => /.unapply(path2) match
        case Some((_, str1)) if str1 == "hello" => Ok(s"Hello, $str")
        case _ => NotFound("")
      case None => NotFound("")
end requestToResponse

val helloWorldService = HttpRoutes.of[IO] {
  case request => requestToResponse(request)
}
```

※本来はもっと条件分岐が必要になってきます。

愚直に書くととても実装が長くなってしまいますが、unapplyはパターンマッチで扱えるのでここにScalaの強力な型システムが組み合わさって、以下のようにとても少量のコードで同じような実装が実現できるのです。

```
val helloWorldService = HttpRoutes.of[IO] {
  case GET -> Root / "hello" / name => Ok(s"Hello, $name.")
}
```

つまり

http4sにはこのようにRequestに対しての抽出子オブジェクトが複数存在しています。

Requestに対しての抽出子オブジェクトを用意し、unapplyメソッドの1つのオブジェクトを受け取り引数を返そうとする特性を利用して、Requestのメソッドがなんなのかとパスがなんなのかをパターンマッチしているのです。

ルーター

http4sのルーターは`Router`というobjectを使用して構築します。

実装自体は以下のように行います。

```
val router: HttpRoutes[IO] = Router(
  "/"    -> helloWorldService,
  "/api" -> apiService
)
```

Play Frameworkのroutesファイルを思い出してください。

Play Frameworkはベースに `routes` ファイルを定義して、ある特定のパス配下のルーティングを別のファイルに切り出して実装を行うことができましたよね？

`routes` と `sub.api.routes` ファイルがある場合

```
# routesファイルに定義
-> /api sub.api.Routes
```

Routerの実装は、http4におけるこのPlay Frameworkと同じような実装だと思ってください。

Routerを構築したはずなのに、戻り値の型がサービスと同じになっているのに気づいたでしょうか？
なぜRouterを構築しているのに型は変わらないのか？

実装を見て確認して見ましょう。

```
def apply[F[_]: Monad](mappings: (String, HttpRoutes[F])*): HttpRoutes[F] =
  define(mappings: _*)(HttpRoutes.empty[F])
```

`apply`メソッドは`define`メソッドを呼んでいるのでそちらも見てみましょう。

単純に何をやっているかというと文字列で指定したパスの情報が空でなかった場合に、受け取ったRequestのパス情報の最初が指定されたパスの文字列と一致していれば後続の処理を行い、一致していない場合はdefault(ここではempty)の処理を行うようになっています。

```
def define[F[_]: Monad](
  mappings: (String, HttpRoutes[F])*
)(default: HttpRoutes[F]): HttpRoutes[F] =
  mappings.sortBy(_._1.length).foldLeft(default) { case (acc, (prefix, routes)) =>
    val prefixPath = Uri.Path.unsafeFromString(prefix)
    if (prefixPath.isEmpty) routes <+> acc
    else
      Kleisli { req =>
        if (req.pathInfo.startsWith(prefixPath))
          routes(translate(prefixPath)(req)).orElse(acc(req))
        else
          acc(req)
      }
  }
```

つまり

つまりhttp4sにおいてRouterの実装は、特定の用途ごとにHttpRoutesをマッピングする処理を行うということです。

サーバー

最初に挙げたように、http4sは様々なバックエンドをサポートしています。
今回はその中のEmberServerを使用してみます。

サーバーの構築は各種Builderを使用して実装を行います。

```
import org.http4s.ember.server.EmberServerBuilder
```

以下が最小のサーバー構築になります。

```
EmberServerBuilder
  .default[IO]
  .withHttpApp(router.orNotFound)
  .build
```

typelevel系のライブラリは`EmberServerBuilder.default[F]`のように型パラメータでエフェクトシステムを切り替えることができるようになっています。

また、`EmberServerBuilder.default[F].build`は`Resource`型を返すので開発者は明示的に`use`メソッドを呼ぶ必要があります。
`Resource`はリソースの生成・破棄を自動で行う機能です。

なぜ`default`なのか？
これはEmberServerBuilderのパラメーターが`private`になっており、`default`でインスタンスの生成をデフォルト引数で行っているためです。

```
final class EmberServerBuilder[F[_]: Async: Network] private (...)

...

def default[F[_]: Async: Network]: EmberServerBuilder[F] =
  new EmberServerBuilder[F](...)
```

EmberServerBuilderには`copy`メソッドが用意されており、各パラメーターの更新はこの`copy`メソッドを使用したメソッドを使用して更新を行っていきます。

今回はHttpAppの更新を行うために、`withHttpApp`メソッドを使用してパラメーターの更新を行いました。

```
def withHttpApp(httpApp: HttpApp[F]): EmberServerBuilder[F] = copy(httpApp = _ => httpApp)
```

HttpApp?

HttpAppとは`Kleisli[F, Request[G], Response[G]]`の型エイリアスです。

HttpRoutesと似ていますね。では違いはなんでしょうか？
2つを見比べてみると、HttpRoutesはOptionTになっています。

```
// HttpRoutes
Kleisli[[T] =>> OptionT[F, T], Request[F], Response[F]]

// HttpApp
Kleisli[F, Request[G], Response[G]]
```

OptionTだと存在しないものがあった場合にNoneになってしまいます。
もしサーバーを起動していて、どのRequestにも一致しないリクエストが来た場合どうなるでしょうか？

おそらくNo Responseになってしまうので、一致するパスが存在しないのか、サーバーがそもそも起動していないのかわかりにくいですよね。

サーバーを起動した以上何かしらのResponseを返さないといけないので、サーバーで起動するHttpAppはOptionTでは無くなっているのです。

HttpRoutes => HttpApp

HttpRoutesからHttpAppへの変換は以下のように行います。

```
router.orNotFound
```

これは受け取ったRequestに該当するものがない場合、NotFoundのレスポンスを返すというシンプルなものです。

実装自体もシンプルなので、もし特別な処理が必要な場合は自身でカスタマイズして実装することもできます。

```
final class KleisliResponseOps[F[_]: Functor, A](self: Kleisli[OptionT[F, *], A, Response[F]]) {
  def orNotFound: Kleisli[F, A, Response[F]] =
    Kleisli(a => self.run(a).getOrElse(Response.notFound))
}
```

サーバーの起動

```
import cats.effect.{ IO, Resource }
import cats.effect.unsafe.implicits.global

import org.http4s.*
import org.http4s.dsl.io.*
import org.http4s.server.{ Router, Server }
import org.http4s.ember.server.EmberServerBuilder

object HelloWorld:

  val helloWorldService: HttpRoutes[IO] = HttpRoutes.of[IO] {
    case GET -> Root / "hello" / name => Ok(s"Hello, $name.")
  }

  val router: HttpRoutes[IO] = Router(
    "/" -> helloWorldService,
  )

  val server: Resource[IO, Server] = EmberServerBuilder
    .default[IO]
    .withHttpApp(router.orNotFound)
    .build

  def main(args: Array[String]): Unit =
    server.use(_ => IO.never).unsafeRunSync()
```

サーバーの起動は、以下のような処理で行います。

serverは構築した時点では`Resource`になっているため、`use`を使用しています。`IO.never`を使用しているのは、サーバーは起動したら停止するまで起動し続けてもらう必要があるため設定しています。(これがないと`run`コマンドで実行しても即時停止してしまいます)

最後にIOになったものを`unsafeRunSync`で実行しています。

```
def main(args: Array[String]): Unit =
  server.use(_ => IO.never).unsafeRunSync()
```

※IOに関してはIOの章で説明するので、ここではそういうものかぐらいで大丈夫です。

サーバーを起動した後、設定したパスにアクセスを行いレスポンスが正常に帰って来てるか確認してみましょう。

```
curl http://localhost:8080/hello/takapi
```

IOAppでの実行

先ほど起動したサーバーをIOAppを使用して起動して見ましょう。

IOAppはプロセスを実行して、SIGTERMを受信したときに無限プロセスを中断し、サーバーを優雅にシャットダウンするためにJVMシャットダウンフックを追加してくれるものです。

IOを実行する時に必要な、Runtimeも内部で生成してくれます。

※ こちらもIOの章で説明します。

実行コード

```
import cats.effect.*

import org.http4s.*
import org.http4s.dsl.io.*
import org.http4s.server.{ Router, Server }
import org.http4s.ember.server.EmberServerBuilder

object HelloWorldWithIOApp extends IOApp:

  val helloWorldService: HttpRoutes[IO] = HttpRoutes.of[IO] {
    case GET -> Root / "hello" / name => Ok(s"Hello, $name.")
  }

  val router: HttpRoutes[IO] = Router(
    "/" -> helloWorldService,
  )

  val server: Resource[IO, Server] = EmberServerBuilder
    .default[IO]
    .withHttpApp(router.orNotFound)
    .build

  def run(args: List[String]): IO[ExitCode] =
    server
      .use(_ => IO.never)
      .as(ExitCode.Success)
```

サーバーの起動

サーバーを起動した後、設定したパスにアクセスを行い同じようにレスポンスが正常に帰って来てるか確認してみましょう。

```
curl http://localhost:8080/hello/takapi
```

まとめ

今まで触って来たPlay Frameworkと比べてみてどう感じましたか？

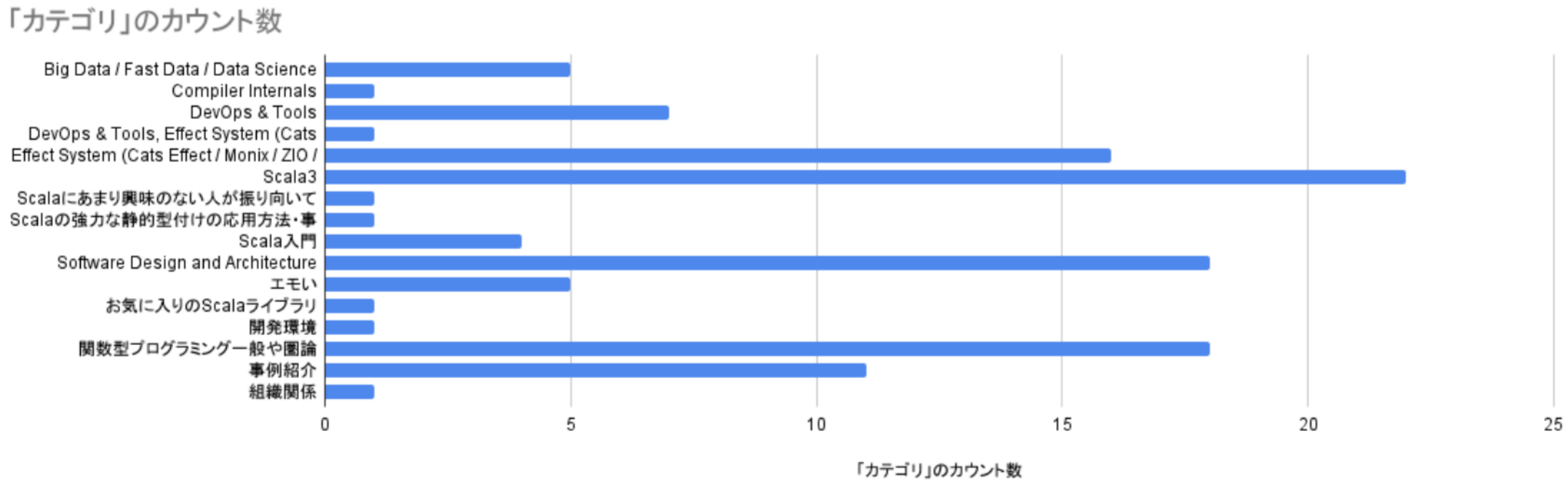
http4sは今まで触って来たものとは違い、かなり関数型だったかと思います。

今回触った技術に関して、
ScalaMatsuri2023用のアンケートの結果を見てみると、以下のように今回触ったものが上位を占めている状態です。

1. Scala3
2. Software Design and Architecture
3. 関数型プログラミング一般や圏論
4. Effect System (Cats Effect / Monix / ZIO / eff etc.)

興味はあるけど〜みたいな人が多い？

アンケート結果 (母数...)


「カテゴリ」のカウント数
Big Data / Fast Data / Data Science
Compiler Internals
DevOps & Tools
DevOps & Tools, Effect System (Cats
Effect System (Cats Effect / Monix / ZIO /
Scala3
Scalaにあまり興味のない人が振り向いて
Scalaの強力な静的型付けの応用方法・事
Scala入門
Software Design and Architecture
エモい
お気に入りのScalaライブラリ
開発環境
関数型プログラミング一般や圏論
事例紹介
組織関係
0
5
10
15
20
25
「カテゴリ」のカウント数

今回はhttp4sを軽く紹介しましたが、今回話した内容以外にもMiddlewareや、クライアント、テストなどいろいろなものがあります。

こちらに関しても、色々話せればと思いますが、
次はCats EffectのEffect Systemを勉強していく予定です。

この夕学を通して、少しでもScalaを使ったEffect Systemや関数型プログラミングに興味を持っていただければなと思います。

ありがとうございました。