Table of Contents

イントロダクション 1.1

基本 1.2

- こんにちは世界 1.2.1
- 関数 1.2.2
- 関数についてもっと 1.2.3
- インポートとモジュール 1.2.4
- ユニオン型 1.2.5
- 型エイリアス 1.2.6
- ユニット型 1.2.7

Elmアーキテクチャ 1.3

- 導入 1.3.1
- 構造 1.3.2
- メッセージ 1.3.3
- アプリケーションの流れ 1.3.4
- ペイロードを含むメッセージ 1.3.5
- 合成 1.3.6
- 合成 - 親 1.3.7
- 合成 - フロー 1.3.8

サブスクリプション(購読)とコマンド 1.4

- サブスクリプション(購読) 1.4.1
- コマンド 1.4.2

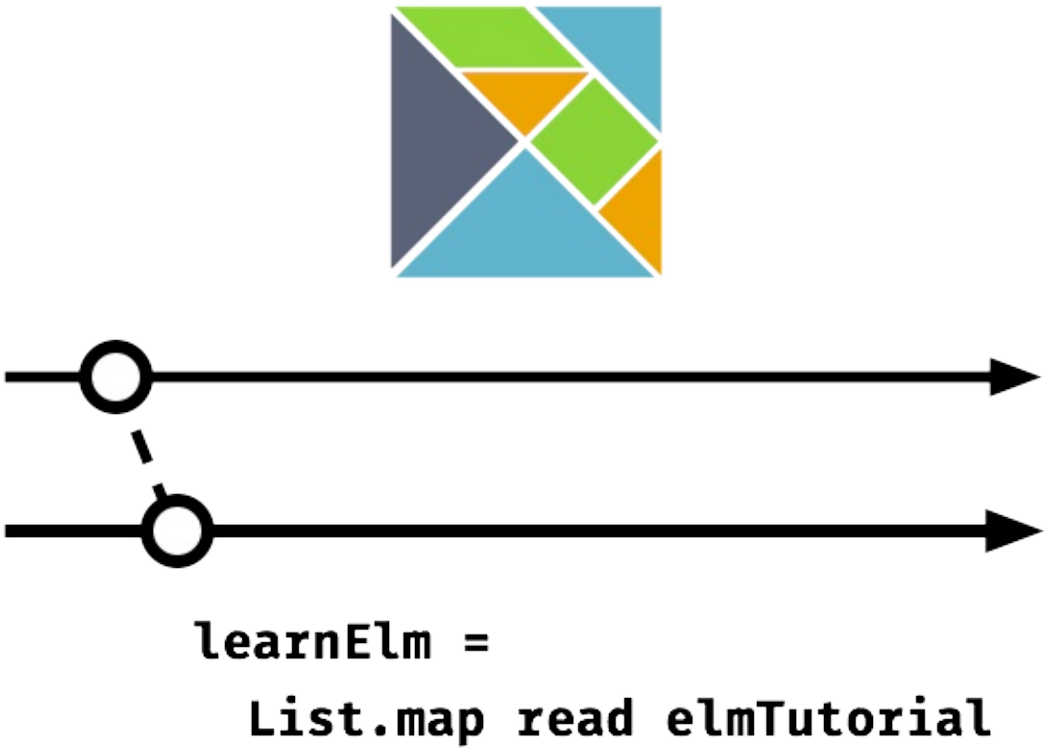
アプリケーションの開始 1.5

- 計画 1.5.1
- バックエンド 1.5.2
- webpackその1 1.5.3
- webpackその2 1.5.4
- webpackその3 1.5.5
- webpackその4 1.5.6
- 複数のモジュール 1.5.7

リソース 1.6

- 導入 1.6.1
- プレイヤー 1.6.2
- プレイヤーのリスト 1.6.3
- メインモジュール 1.6.4

メインビュー 1.6.5
メイン 1.6.6
フェッチ 1.7
計画 1.7.1
プレイヤーのメッセージ 1.7.2
プレイヤーの更新 1.7.3
プレイヤーのコマンド 1.7.4
メイン 1.7.5
試してみよう 1.7.6
ルーティング 1.8
導入 1.8.1
ルーティング 1.8.2
プレイヤー編集のビュー 1.8.3
メインモデル 1.8.4
メインメッセージ 1.8.5
メインアップデート 1.8.6
メインビュー 1.8.7
メイン 1.8.8
試してみよう 1.8.9
ナビゲーション 1.8.10
編集 1.9
計画 1.9.1
メッセージ 1.9.2
プレイヤー編集 1.9.3
コマンド 1.9.4
更新 1.9.5
おわりに 1.10
さらなる改善 1.10.1
ヒントとテクニック 1.11
コンテキスト 1.11.1
ポイントフリースタイル 1.11.2
トラブルシューティング 1.11.3



Elmチュートリアル

Elmによるシングルページアプリ開発のチュートリアル

このチュートリアルの内容は以下の通りです。

- elmの基本
- elmのコマンド(Cmd)とサブスクリプション(Sub)について
- elmのアーキテクチャを理解する
- サブコンポーネントとリソースにアプリケーションを分解する
- CSSの統合(訳注: 今はない)
- JSONの取得と解析
- ルーティング
- CRUD操作

こちらからオンラインで読めます。

オフライン版はこちら(PDF, ePub, Mobi)から。

コード

このチュートリアルの第2部で構築するサンプルアプリケーションのコードは、https://github.com/sporto/elm-tutorial-appにあります。

要件

このチュートリアルでは、次のものが必要になります。

- Elm version 0.18 (インストールについてはチュートリアルの後半で説明します)
- Node JS version 4 +

貢献するには

https://github.com/sporto/elm-tutorialにイシューを登録してPRを送ってください

Share on Twitter | Follow @sebasporto

基礎

この章の内容:

- 基本的なElmアプリケーションの実行
- Elmにおける関数と型の基礎

こんにちは世界

Elmのインストール

http://elm-lang.org/install にアクセスして、システムに適したインストーラをダウンロードしてください。

最初のElmアプリケーション

最初のElmアプリケーションを書いてみましょう。アプリケーション用のフォルダを作成します。このフォルダで、ターミナルで次のコマンドを実行します。

```
elm package install elm-lang/html
```

これで *html* モジュールがインストールされます。次に、以下のコードを含む `Hello.elm` ファイルを追加します：

```
module Hello exposing (..)

import Html exposing (text)


main =
    text "Hello"
```

ターミナルでこのフォルダに移動して、次のように入力します。

```
elm reactor
```

以下が表示されるはずです：

```
elm reactor 0.18.0
Listening on http://localhost:8000/
```

ブラウザーで `http://0.0.0.0:8000/` を開きます。`Hello.elm` をクリックしてください。ブラウザに `Hello` が表示されます。

`main` の型アノテーションがないことに関する警告が表示されることに注意してください。これは後で型アノテーションを入力するとして、今のところ無視しましょう。

ここで何が起こっているのかを見てみましょう：

モジュール宣言

```
module Hello exposing (..)
```

Elmのすべてのモジュールはモジュール宣言で始まらなければなりません。この場合、モジュール名は `Hello` です。ファイルとモジュールに同じ名前を付けるのが規約です。 `Hello.elm` には `module Hello` が含まれています。

宣言の `exposing(..)` 部分は、このモジュールをインポートする他のモジュールにどのような関数と型を公開するかを指定します。この場合、すべてを `(..)` で公開します。

インポート

```
import Html exposing (text)
```

Elmでは、明示的に使用したいモジュールをインポートする必要があります。この場合、**Html**モジュールを使用します。

このモジュールには、htmlを扱う多くの関数があります。 `.text` を使用するので、 `exposing` を使ってこの関数を現在の名前空間にインポートします。

メイン

```
main =
    text "Hello"
```

Elmのフロントエンドアプリケーションは `main` という関数から始まります。 `main` は、ページに描画する要素を返す関数です。この場合、 `text` を使って作成されたHtml要素を返します。

Elm reactor

Elm **reactor**は、オンザフライでコンパイルするサーバーを作成します。 **reactor**はビルドプロセスを種々に設定する必要なくアプリケーションを開発するのに便利です。しかし、**reactor**には制限があるので、将来的には適切なビルドプロセスに切り替える必要があるでしょう。

関数の基本

この章では、Elmに慣れるために重要な基本的な構文、関数、関数シグネチャ、部分適用、およびパイプ演算子について説明します。

機能

Elmは2種類の関数をサポートしています：

- 無名関数
- 名前付き関数

無名関数

無名関数は、その名前が示すように、名前なしで作成する関数です。

```
\x -> x + 1

\x y -> x + y
```

バックスラッシュと矢印の間には関数の引数を列挙し、矢印の右側にはそれらの引数で何をすべきかを示します。

名前付き関数

Elmの名前付き関数は次のようになります。

```
add1 : Int -> Int
add1 x =
  x + 1
```

- この例の最初の行は関数のシグネチャです。シグネチャはElmではオプションですが、関数の意図を明確にするために記述することが推奨されています。
- 残りは関数の実装です。実装は、その直上で定義したシグネチャに従わなければなりません。

この場合、シグネチャは「引数として整数(Int)をとり、別の整数(Int)を返す」ということを言っています。

この関数は、以下のように呼び出すことができます：

```
add1 3
```

Elmでは、関数適用を表すために(カッコの代わりに)空白を使用します。

以下は別の名前付き関数です：

```
add : Int -> Int -> Int
add x y =
  x + y
```

この関数は両方ともIntである2つの引数をとり、別のIntを返します。この関数は次のように呼び出すことができます。

```
add 2 3
```

引数なし

Elmでは定数は引数をとらない関数です。

```
name =
  "Sam"
```

どのように関数適用がなされるか

前述のように、2つの引数をとる関数は次のようになります。

```
divide : Float -> Float -> Float
divide x y =
    x / y
```

このシグネチャを、2つの浮動小数点数をとり、別の浮動小数点数を返す関数と考えることもできます。

```
divide 5 2 == 2.5
```

しかし、これはまったく真実ではありません。Elmでは、すべての関数はただ1つの引数をとり、1つの結果を返します。ここではその結果は別の関数になります。 上記の関数を使って説明しましょう。

```
-- 次のようにすると：

divide 5 2

-- 次のように評価されます：

((divide 5) 2)

-- 最初の「divide 5」が評価されます。
-- 引数 '5'は `divide`に適用され、中間的な関数になります。

divide 5 -- ->中間的な関数

-- この中間的な関数を `divide5`と呼びます。
-- この中間的な関数のシグネチャと本体を見ることができたなら、次のようになるでしょう：

divide5 : Float -> Float
divide5 y =
  5 / y

-- したがって、すでに `5`が適用された関数があります。

-- 次の引数、すなわち `2`が適用されます

divide5 2

-- これは最終結果を返します
```

括弧を書かないで済むのは、関数適用が左結合だからです。

カッコでグループ化する

別の関数呼び出しの結果で関数を呼び出す場合は、呼び出しをグループ化するために括弧を使用する必要があります。

```
add 1 (divide 12 3)
```

ここでは、「add」の第2引数として「divide 12 3」の結果が与えられます。

対照的に、他の多くの言語では、次のように書かれます。

```
add(1, divide(12, 3))
```

部分適用

上で説明したように、すべての関数は1つの引数しか取らず、別の関数または結果を返します。これが意味するのは、引数が1つだけでも上記の `add` のような関数を呼び出すことができるということです。たとえば `add 2` とすると、部分的に適用された関数が返ります。この結果の関数のシグネチャ

は、`Int -> Int`です。

`add 2`は最初引数が値'2'に束縛された別の関数を返します。返ってきた関数に2番目の引数を与えて呼び出すと、その結果は「2 + 2番目の引数」になります。

```
add2 = add 2
add2 3 -- 結果は5
```

部分適用は、コードを読みやすくしたり、アプリケーション内の関数間で状態を渡したりすることができ、Elmプログラミングにおいて非常に便利です。

パイプ演算子

上記のように、次のような関数を入れ子にすることができます：

```
add 1 (multiply 2 3)
```

これは簡単な例ですが、もっと複雑な例を考えてみましょう：

```
sum (filter (isOver 100) (map getCost records))
```

このコードが読み難いのは、内から外に解決していくからです。パイプ演算子を使用すると、そのような式をより読みやすく書くことができます。

```
3
    |> multiply 2
    |> add 1
```

このようにできるのは前述の部分適用のおかげです。この例では、値「3」が部分適用された関数`multiply 2`に渡されます。その結果は、部分適用されたまた別の関数`add 1`に渡されます。

パイプ演算子を使用すると、上記の複雑な例は次のようになります。

```
records
    |> map getCost
    |> filter (isOver 100)
    |> sum
```

関数についてもっと

型変数

以下のような型シグネチャを持つ関数を考えてみましょう：

```
indexOf : String -> List String -> Int
```

ここで仮定した関数は、文字列と文字列のリストを取り、指定された文字列がリスト内で見つかった場合はインデックスを、見つからない場合は-1を返します。 もちろん、文字列のリストではなく整数リストに対しては、この関数を使用することはできません。

しかし、特定の型の代わりに型変数またはスタンドインを使用することによって、この関数をジェネリックにすることができます。

```
indexOf : a -> List a -> Int
```

`String` を `a` で置き換えることにより、今や `indexOf` のシグネチャは「任意の型 `a` の値と同じ型 `a` のリストをとり、整数を返す」というものになります。型が一致する限り、コンパイラは満足します。 `indexOf` は今や、引数「 `String` と `String` のリスト」でも「 `Int` と `Int` のリスト」でも呼び出すことができ、期待するように動作します。

この方法で関数をよりジェネリックにできます。複数の型変数を持つこともできます：

```
switch : ( a, b ) -> ( b, a )
switch ( x, y ) =
  ( y, x )
```

この関数は `(a,b)` 型のタプルをとり、 `(b,a)` 型のタプルを返します。次はすべて有効な呼び出しです。

```
switch (1, 2)
switch ("A", 2)
switch (1, ["B"])
```

型変数には任意の小文字の識別子を使用でき、 `a` と `b` のように1文字にするのは慣習にすぎません。たとえば、次のシグネチャは完全に有効です。

```
indexOf : thing -> List thing -> Int
```

引数としての関数

次のようなシグネチャを考えてみましょう：

```
map : (Int -> String) -> List Int -> List String
```

この関数は：

- 引数として関数「`(Int -> String)`」と整数のリストをとり、
- そして文字列のリストを返します。

興味深いのは`(Int -> String)`の部分です。これは、引数の関数が`(Int -> String)`シグネチャに従わなければならないことを示しています。

例えば、coreにある`toString`はそのような関数です。したがって、この`map`関数を以下のように呼び出すことができます：

```
map toString [1, 2, 3]
```

しかし、`Int`と`String`は特殊すぎます。代りに型変数を使ったシグネチャを良く見るでしょう。

```
map : (a -> b) -> List a -> List b
```

この関数は`a`のリストを`b`のリストにマップします。`a`と`b`が実際にどのような型であるかは、最初の引数に与えられた関数が同じ型を使用している限り気にしません。

たとえば、次のシグネチャを持つ関数があるとき、

```
convertStringToInt : String -> Int
convertIntToString : Int -> String
convertBoolToInt : Bool -> Int
```

ジェネリックなmapは次のように呼び出すことができます：

```
map convertStringToInt ["Hello", "1"]
map convertIntToString [1, 2]
map convertBoolToInt [True, False]
```

インポートとモジュール

Elmでは、`import`キーワードを使用してモジュールをインポートします。

```
import Html
```

これは`Html`モジュールをインポートします。すると完全修飾パスを使用して、このモジュールの関数と型を使用できます。

```
Html.div [] []
```

また、モジュールをインポートして、そこから特定の関数と型をexposeすることもできます。

```
import Html exposing (div)
```

`div`は現在のスコープに導入され、直接使うことができます：

```
div [] []
```

モジュール内のすべてをexposeすることさえできます：

```
import html exposing (..)
```

この場合、モジュール中のすべての関数と型を直接使用することができます。しかし、あいまいさやモジュール間の衝突の可能性があるため、これはほとんどの場合お勧めできません。

モジュール名と同じ名前の型

多くのモジュールは、モジュールと同じ名前の型をエクスポートします。例えば、`Html`モジュールは`Html`型を持ち、`Task`モジュールは`Task`型を持っています。

この関数は`Html`要素を返します：

```
import Html

myFunction : Html.Html
myFunction =
    ...
```

上記は以下と等価です：

```
import Html exposing (Html)

myFunction : Html
myFunction =
    ...
```

最初の例は `Html` モジュールのみをインポートし、完全修飾パス `Html.Html` を使用します。

2番目の例では、 `Html` モジュールを `Html` モジュールからexposeしています。また、 `Html` 型を直接使用します。

モジュールの宣言

elmでモジュールを作成するときは、最初に `module` 宣言を追加します：

```
module Main exposing (..)
```

`Main` はモジュールの名前です。 `exposing(..)` は、このモジュールのすべての関数と型を公開することを意味します。 elmは、このモジュールを**Main.elm**というファイル、つまりモジュールと同じ名前のファイルで見つけることを想定しています。

アプリケーションにより深いファイル構成を持たせることができます。たとえば、ファイル **Players/Utils.elm**には次のような宣言が必要です。

```
module Players.Utils exposing (..)
```

アプリケーションのどこからでもこのモジュールをインポートすることができます：

```
import Players.Utils
```

ユニオン型

Elmでは、ユニオン型は信じられないほど柔軟性があるため、多くのものに使用されています。ユニオン型は次のようになります。

```
type Answer = Yes | No
```

`Answer` は `Yes` または `No` のいずれかです。ユニオン型は、コードをより一般的なものにするのに便利です。たとえば、

```
respond : Answer -> String
respond answer =
    ...
```

ように宣言された関数respondは最初の引数として `Yes` または `No` を取ることができます。たとえば「respond Yes」は有効な呼び出しです。

Elmでは、ユニオン型はタグと呼ばれることもあります。

ペイロード

ユニオン型は、関連する情報を持つことができます。

```
type Answer = Yes | No | Other String
```

この場合、タグ「Other」には文字列が紐付けられ、次のように `respond` を呼び出すことができます：

```
respond (Other "Hello")
```

この場合括弧が必要です。さもなければElmはこれを2つの引数を渡して応答すると解釈してしまいます。

コンストラクタ関数として

ペイロードを「Other」に持たせる方法に注意してください。

```
Other "Hello"
```

これは `Other` が関数であるときの関数呼び出しと同じです。ユニオン型は関数と同様に動作します。たとえば、以下のような型があるとき、

```
type Answer = Message Int String
```

次のようにして `Message` タグを作成します：

```
Message 1 "Hello"
```

他の関数と同様に部分適用も可能です。これらは一般的に `コンストラクタ` と呼ばれ、これを使って完全な型を作ることができます。たとえば、関数 `Message` を使って `(Message 1 "Hello")` を構築することができます。

ユニオン型の入れ子

ユニオン型を他の型の中に入れ子にすることは非常に一般的です。

```
type OtherAnswer = DontKnow | Perhaps | Undecided

type Answer = Yes | No | Other OtherAnswer
```

これを例えば、引数に `Answer` 型を取る `respond` 関数に渡すことができます：

```
respond (Other Perhaps)
```

型変数

型変数(もしくはスタンドイン)を使うこともできます：

```
type Answer a = Yes | No | Other a
```

これは、たとえばIntやStringなどの異なる型でも使用できる `Answer` です。

これを使って関数 `respond` を次のように定義できます。

```
respond : Answer Int -> String
respond answer =
    ...
```

ここでは、 `a` スタンドインは、 `Answer Int` シグネチャによって「 `Int` 型でなければならない」と言っています。

なので、以下のように `respond` を呼ぶことができます：

```
respond (Other 123)
```

しかし、 `respond` は `a` として整数を期待するので、 `respond(Other "Hello")` はコンパイルエラーとなります。(訳注:「respond: Anwer a -> String」のようにジェネリックなままで関数を定義することもできます。)

一般的な使用方法

ユニオン型の典型的な使い方は、プログラム内の値を渡すことです。この値は、既知の可能な値のセットの1つになります。

例えば、典型的なウェブアプリケーションでの実行可能なアクションとして、ユーザーの追加、ユーザーの追加、ユーザーの削除などがあるとします。これらのアクションのいくつかはペイロードを持ちます。

このとき、ユニオン型を使用するのが一般的です。

```
type Action
    = LoadUsers
    | AddUser
    | EditUser UserId
    ...
```

ユニオン型については他にも多くの話題があります。興味があればこちら こちらをご覧ください。

型エイリアス

Elmの型エイリアスは、その名前のとおり型に対する別名です。例えば、Elmでは、coreに `Int` 型と `String` 型がありますが、これらに別名をつけることができます：

```
type alias PlayerId = Int

type alias PlayerName = String
```

他のcoreの型を単純に指し示す2つの型エイリアスを作成することで、以下のような関数を書く代わりに：

```
label: Int -> String
```

次のように書くことができます：

```
label: PlayerId -> PlayerName
```

こうすることで、関数の処理がはるかに明確になります。

レコード

Elmのレコード定義は次のようになります。

```
{ id : Int
, name : String
}
```

レコードを直接引数に取る関数の場合、次のようなシグネチャを書かなければならないでしょう：

```
label: { id : Int, name : String } -> String
```

これだとかなり冗長ですが、型エイリアスを使うことで以下のように書けます：

```
type alias Player =
    { id : Int
    , name : String
    }

label: Player -> String
```

ここでは、レコード定義を指す `Player` 型エイリアスを作成しています。次に、その型名を関数シグネチャに使用しています。

コンストラクタ

レコードに対する型エイリアス名はコンストラクタ関数として使用できます。つまり型エイリアスを関数として使用し、レコードの値を作成できます。

```
type alias Player =
    { id : Int
    , name : String
    }

Player 1 "Sam"
==> { id = 1, name = "Sam" }
```

ここでは `Player` 型エイリアスを作成し、次に2つのパラメータを持つ関数として `Player` を呼び出します。これにより、適切な属性を持つレコードが返されます。引数の順序によって、どの値がどの属性に割り当てられるかが決まることに注意してください。

ユニット型

空のタプル `()` はElmではユニット型と呼ばれます。一般的なので説明が必要でしょう。

`a` で表される型変数を持つ型エイリアスを考えてみましょう：

```
type alias Message a =
    { code : String
    , body : a
    }
```

このように `body` が `String` である `Message` を期待する関数を作ることができます：

```
readMessage : Message String -> String
readMessage message =
    ...
```

または `body` が整数のリストであるような `Message` を期待する関数も書けます：

```
readMessage : Message (List Int) -> String
readMessage message =
    ...
```

しかし、bodyに値を必要としない関数はどうでしょう？その場合、bodyが空であることを示すために、ユニット型を使用します。

```
readMessage : Message () -> String
readMessage message =
    ...
```

この関数は、空の**body**を持つ `Message` をとります。これは任意の値を持ち、それが空である、というのとは違います。

このように、ユニットタイプは通常、空の値のプレースホルダとして使用されます。

タスク

`Task` 型でのユニット型の使用例を見てみましょう。 `Task` を使う場合、ユニット型を頻繁に使用します。

典型的なタスクはエラーと結果を持ちます。

```
Task error result
```

- エラーを安全に無視できるタスクが必要な場合があります： `Task () result`
- あるいは、結果を無視する場合： `Task error ()`

- またはその両方を無視する場合： `Task () ()`

Elmアーキテクチャ

この章の内容:

- Elmアーキテクチャ概要
- Html.programの紹介
- メッセージ
- コマンド
- サブスクリプション(購読)

> This page covers Elm 0.18

はじめに

Elmのフロントエンドアプリケーションを構築する際には、Elmアーキテクチャと呼ばれるパターンを使用します。このパターンは、再利用、結合、構成など多様性を持った自己完結型コンポーネントを作成する方法を提供します。

Elmはこのための `Html.program` モジュールを提供しています。これを理解するために小さなアプリを構築してみましょう。

elm-htmlをインストールする：

```
elm package install elm-lang/html
```

App.elmというファイルを作成します。

```
module App exposing (..)

import Html exposing (Html, div, text, program)


-- MODEL


type alias Model =
    String


init : ( Model, Cmd Msg )
init =
    ( "Hello", Cmd.none )



-- MESSAGES


type Msg
    = NoOp



-- VIEW


view : Model -> Html Msg
view model =
    div []
        [ text model ]
```

```
-- UPDATE


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        NoOp ->
            ( model, Cmd.none )



-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none



-- MAIN


main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

以下のようにこのプログラムを実行することができます：

```
elm reactor
```

そして http://localhost:8000/App.elm を開きます。

これは "Hello"を表示するためのコードですが、非常に複雑になり得るElmアプリケーションの構造を理解するのに役立ちます。

> This page covers Elm 0.18

Html.programの構造

Import

```
import Html exposing (Html, div, text, program)
```

- `Html`モジュールの`Html`型と、`div`、`text`、`program`などいくつかの関数を使用します。

モデル

```
type alias Model =
    String


init : ( Model, Cmd Msg )
init =
    ( "Hello", Cmd.none )
```

まず、この種のアプリケーションのモデルを型エイリアスとして定義します。ここでは単に`String`です。 次に、`init`関数を定義します。この関数は、アプリケーションの初期入力値を提供します。

Html.programは`(model, command)`というタプルを期待しています。このタプルの最初の要素は、初期状態です(例えば"Hello")。2番目の要素は、実行する最初のコマンドです。これについては後で詳しく説明します。

elmアーキテクチャを使用する場合、すべてのコンポーネントのモデルを一つの状態ツリーとして構築します。これについては後で詳しく説明します。

メッセージ

```
type Msg
    = NoOp
```

メッセージは、コンポーネントが応答するアプリケーションで発生するものです。この場合、アプリケーションは何もしないので、`NoOp`というメッセージしかありません。

メッセージの他の例としては、ウィジェットの表示と非表示を切り替えるための「Expand」または「Collapse」があります。メッセージにはユニオン型を使用します：

```
type Msg
    = Expand
    | Collapse
```

ビュー

```
view : Model -> Html Msg
view model =
    div []
        [ text model ]
```

`view` 関数は、アプリケーションのモデルを使ってHtml要素をレンダリングします。型シグネチャは `Html Msg` であることに注意してください。これは、このHtml要素がMsgというタグが付けられたメッセージを生成することを意味します。今後、インタラクションを導入するときに必要になります。

更新

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        NoOp ->
            ( model, Cmd.none )
```

次に、 `update` 関数を定義します。この関数は、メッセージが受信されるたびにHtml.programによって呼び出されます。 `update` 関数は、モデルを更新するメッセージに応答し、必要に応じてコマンドを返します。

この例では、 `NoOp` にのみ応答し、変更されていないモデルと、実行するコマンドがないことを意味する `Cmd.none` を返します。

サブスクリプション(購読)

```
subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none
```

サブスクリプションを使用して、アプリケーションへの外部入力を待ち受けます。サブスクリプションのいくつかの例は次のとおりです。

- マウスの移動
- キーボードイベント
- ブラウザの閲覧ロケーションの変更

この場合には、外部入力には関心がないので、Sub.noneを使用しています。

メイン

```
main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

最後に、 `Html.program` はすべてを結びつけ、ページ中にレンダリングできるhtml要素を返します。 `program` 関数は引数に `init` 、 `view` 、 `update` と `subscriptions` を取ります。

> This page covers Elm 0.18

メッセージ

先のセクションでは、Html.programを使って静的なHtmlを使用するアプリケーションを作成しました。次に、メッセージを使ってユーザーとやりとりをするアプリケーションを作成しましょう。

```
module Main exposing (..)

import Html exposing (Html, button, div, text, program)
import Html.Events exposing (onClick)


-- モデル


type alias Model =
    Bool


init : ( Model, Cmd Msg )
init =
    ( False, Cmd.none )



-- メッセージ


type Msg
    = Expand
    | Collapse



-- VIEW


view : Model -> Html Msg
view model =
    if model then
        div []
            [ button [ onClick Collapse ] [ text "Collapse" ]
            , text "Widget"
            ]
    else
        div []
            [ button [ onClick Expand ] [ text "Expand" ] ]
```

```
-- 更新


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Expand ->
            ( True, Cmd.none )

        Collapse ->
            ( False, Cmd.none )



-- サブスクリプション(購読)


subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none



-- MAIN


main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

このプログラムは以前のプログラムと非常によく似ていますが、今は `Expand` と `Collapse` という2つのメッセージがあります。このプログラムをファイルにコピーし、Elm reactorを使用して開くことができます。

`view` と `update` 関数をもっと詳しく見てみましょう。

表示

```
view : Model -> Html Msg
view model =
    if model then
        div []
            [ button [ onClick Collapse ] [ text "Collapse" ]
            , text "Widget"
            ]
    else
        div []
            [ button [ onClick Expand ] [ text "Expand" ] ]
```

モデルの状態に応じて、折りたたまれたビューまたは拡大されたビューのいずれかが表示されます。

`onClick` 関数に注意してください。このビューは `Html Msg` 型であるため、`onClick` を使ってそのタイプのメッセージをトリガーすることができます。折りたたみと展開の両方がメッセージタイプです。

更新

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Expand ->
            ( True, Cmd.none )

        Collapse ->
            ( False, Cmd.none )
```

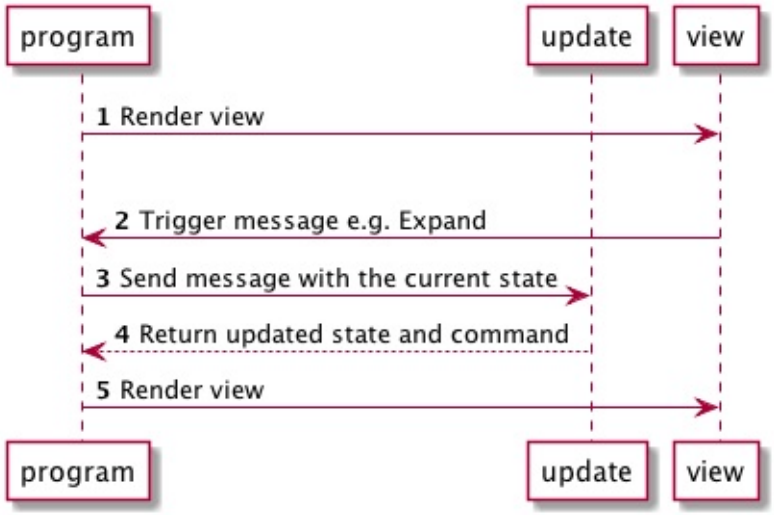
`update` は可能なメッセージに応答します。メッセージに応じて、目的の状態を返します。メッセージが `Expand` のとき、新しい状態は `True` (拡張)になります。

次に、**Html.program**がこれらの部分を一緒にオーケストレーションする方法を見てみましょう。

> This page covers Elm 0.18

アプリケーションフロー

次の図は、アプリケーションの部分がHtml.programとどのように対話するかを示しています。



1. `Html.program` は、初期モデルでビュー関数を呼び出してレンダリングします。
2. ユーザが「Expand」ボタンをクリックすると、ビューは「Expand」メッセージをトリガします。
3. `Html.program` は `Expand` メッセージを受け取り、 `Expand` メッセージと現在のアプリケーション状態を受け取る `update` を呼び出します。
4. `update` 関数は、更新された状態と実行するコマンド(または `Cmd.none`)を返すことによって、メッセージに反応します。
5. `Html.program` は更新された状態を受け取り、それを保存し、更新された状態を引数としてビューを呼び出します。

通常、 `Html.program` はElmアプリケーションが状態を保持する唯一の場所です。一つの大きな状態ツリーに集中しています。

> This page covers Elm 0.18

ペイロードを含むメッセージ

メッセージにペイロードを含めて送ることができます：

```
module Main exposing (..)

import Html exposing (Html, button, div, text, program)
import Html.Events exposing (onClick)


-- モデル


type alias Model =
    Int


init : ( Model, Cmd Msg )
init =
    ( 0, Cmd.none )


-- メッセージ


type Msg
    = Increment Int



-- ビュー


view : Model -> Html Msg
view model =
    div []
        [ button [ onClick (Increment 2) ] [ text "+" ]
        , text (toString model)
        ]


-- 更新


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Increment howMuch ->
            ( model + howMuch, Cmd.none )
```

```
-- サブスクリプション(購読)


subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none



-- MAIN


main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

`Increment` メッセージが整数を必要とすることに注意してください：

```
type Msg
    = Increment Int
```

次に、 `view` 内でペイロードでメッセージをトリガーします。

```
onClick (Increment 2)
```

そして最後にupdateでは、パターンマッチングを使用してペイロードを抽出します。

```
update msg model =
    case msg of
        Increment howMuch ->
            ( model + howMuch, Cmd.none )
```

> This page covers Elm 0.18

合成

Elmアーキテクチャを使用する大きな利点の1つは、コンポーネントを合成する方法です。これを理解するために例を示します。

- 親コンポーネント `App` があります
- その子コンポーネントに `Widget` があります

子コンポーネント

子コンポーネントから始めましょう。これは**Widget.elm**のコードです。

```
module Widget exposing (..)

import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)


-- モデル


type alias Model =
    { count : Int
    }


initialModel : Model
initialModel =
    { count = 0
    }



-- メッセージ


type Msg
    = Increase



-- VIEW


view : Model -> Html Msg
view model =
    div []
        [ div [] [ text (toString model.count) ]
        , button [ onClick Increase ] [ text "Click" ]
        ]



-- 更新


update : Msg -> Model -> ( Model, Cmd Msg )
update message model =
    case message of
        Increase ->
            ( { model | count = model.count + 1 }, Cmd.none )
```

このコンポーネントは、サブスクリプションとメインを除いて、前のセクションで作成したアプリケーションとほぼ同じです。このコンポーネントは：

- 独自のメッセージ(Msg)を定義します。
- 独自のモデルを定義します。
- 自身のメッセージ(`Increase` など)に応答する `update` 関数を提供します。

コンポーネントはここで宣言されたことだけを知っていることに注意してください。
`view` と `update` は両方とも、コンポーネント内で宣言された型(`Msg` と `Model`)のみを使用します。

次のセクションでは、親コンポーネントを作成します。

> This page covers Elm 0.18

合成

親コンポーネント

これは親コンポーネントのコードです。

```
module Main exposing (..)

import Html exposing (Html, program)
import Widget

-- モデル


type alias AppModel =
    { widgetModel : Widget.Model
    }


initialModel : AppModel
initialModel =
    { widgetModel = Widget.initialModel
    }


init : ( AppModel, Cmd Msg )
init =
    ( initialModel, Cmd.none )



-- メッセージ


type Msg
    = WidgetMsg Widget.Msg



-- VIEW


view : AppModel -> Html Msg
view model =
    Html.div []
        [ Html.map WidgetMsg (Widget.view model.widgetModel)
        ]
```

```
-- 更新


update : Msg -> AppModel -> ( AppModel, Cmd Msg )
update message model =
    case message of
        WidgetMsg subMsg ->
            let
                ( updatedWidgetModel, widgetCmd ) =
                    Widget.update subMsg model.widgetModel
            in
                ( { model | widgetModel = updatedWidgetModel }, Cmd.map WidgetMsg widge
tCmd )


-- サブスクリプション(購読)


subscriptions : AppModel -> Sub Msg
subscriptions model =
    Sub.none



-- APP


main : Program Never AppModel Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

このコードの重要なセクションを見てみましょう。

モデル

```
type alias AppModel =
    { widgetModel : Widget.Model ❶
    }
```

親コンポーネントには独自のモデルがあります。このモデルの属性の1つに `Widget.Model` ❶が含まれています。この親コンポーネントは `Widget.Model` の中身を知る必要はないことに注意してください。

```
initialModel : AppModel
initialModel =
    { widgetModel = Widget.initialModel ❷
    }
```

最初のアプリケーションモデルを作成するときは、単にここから `Widget.initialModel` ❷を呼び出します。

子コンポーネントが複数ある場合は、それぞれの子コンポーネントを同じように実行します。たとえば、次のようになります。

```
initialModel : AppModel
initialModel =
    { navModel = Nav.initialModel,
    , sidebarModel = Sidebar.initialModel,
    , widgetModel = Widget.initialModel
    }
```

あるいは、同じタイプの複数の子コンポーネントを持つこともできます。

```
initialModel : AppModel
initialModel =
    { widgetModels = [Widget.initialModel]
    }
```

メッセージ

```
type Msg
    = WidgetMsg Widget.Msg
```

メッセージがそのコンポーネントに属していることを示すために `Widget.Msg` をラップするユニオン型を使用します。これにより、アプリケーションが関連するコンポーネントにメッセージをルーティングできるようになります(これは、update関数を見ればより明確になります)。

複数の子コンポーネントを持つアプリケーションでは、次のようなものがあります。

```
type Msg
    = NavMsg Nav.Msg
    | SidebarMsg Sidebar.Msg
    | WidgetMsg Widget.Msg
```

表示

```
view : AppModel -> Html Msg
view model =
    Html.div []
        [ Html.map❶ WidgetMsg❷ (Widget.view❸ model.widgetModel❹)
        ]
```

メインアプリケーションの `view` は `Widget.view` をレンダリングします。しかし、`Widget.view` は `Widget.Msg` を送出するので、 `Main.Msg` を送出するこのビューと互換性がありません。

- `Html.map` ❶を使用して、放出されたメッセージをWidget.viewから期待されるタイプ(Msg)にマッピングします。 `Html.map` タグは `WidgetMsg` タグを使ってサブビューから来るメッセージにタグを付けます。
- 子コンポーネントが気にするモデルの部分、つまり `model.widgetModel` のみを渡します。

更新

```
update : Msg -> AppModel -> (AppModel, Cmd Msg)
update message model =
    case message of
        WidgetMsg❶ subMsg❷ ->
            let
                (updatedWidgetModel, widgetCmd)❹ =
                    Widget.update❸ subMsg model.widgetModel
            in
                ({ model | widgetModel = updatedWidgetModel }, Cmd.map❺ WidgetMsg widg
etCmd)
```

`WidgetMsg` ❶が `update` によって受け取られると、子コンポーネントに更新を委譲します。しかし子コンポーネントは自分が担当する `widgetModel` 属性だけを更新します。

パターンマッチングを使用して `WidgetMsg` から `subMsg` ❷を抽出します。この `subMsg` は `Widget.update` が予期する型になります。

この `subMsg` と `model.widgetModel` を使って、 `Widget.update` ❸を呼び出します。これは更新された `widgetModel` とコマンドを含むタプルを返します。

`Widget.update` からの応答を分解❹するためにパターンマッチングを使います。

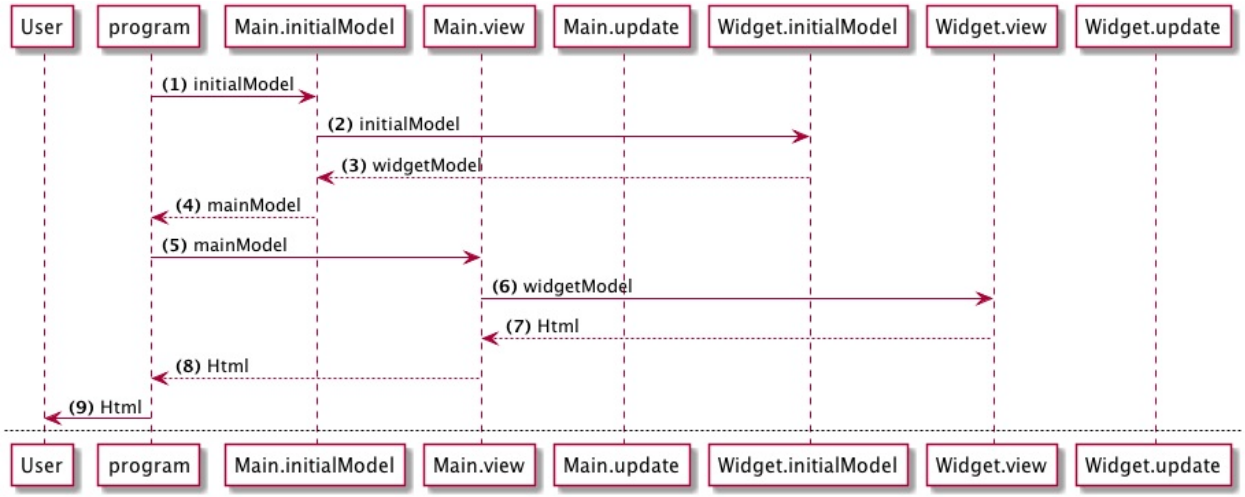
最後に、 `Widget.update` によって返されたコマンドを正しい型に対応付ける必要があります。このために `Cmd.map` ❺を使い、 `WidgetMsg` でコマンドにタグを付けます。これは、ビューで行ったのと同様です。

> This page covers Elm 0.18

合成

このアーキテクチャを説明する2つの図があります。

初期レンダリング



(1)**program**はアプリケーションの初期モデルを取得するために**Main.initialModel**を呼び出します

(2)**Main**は**Widget.initialModel**を呼び出します

(3)**Widget**は初期モデルを返します

(4)**Main**は、ウィジェットのモデルを含む合成されたメインのモデルを返します

(5)**program**は メインのモデルを渡して**Main.view**を呼び出します

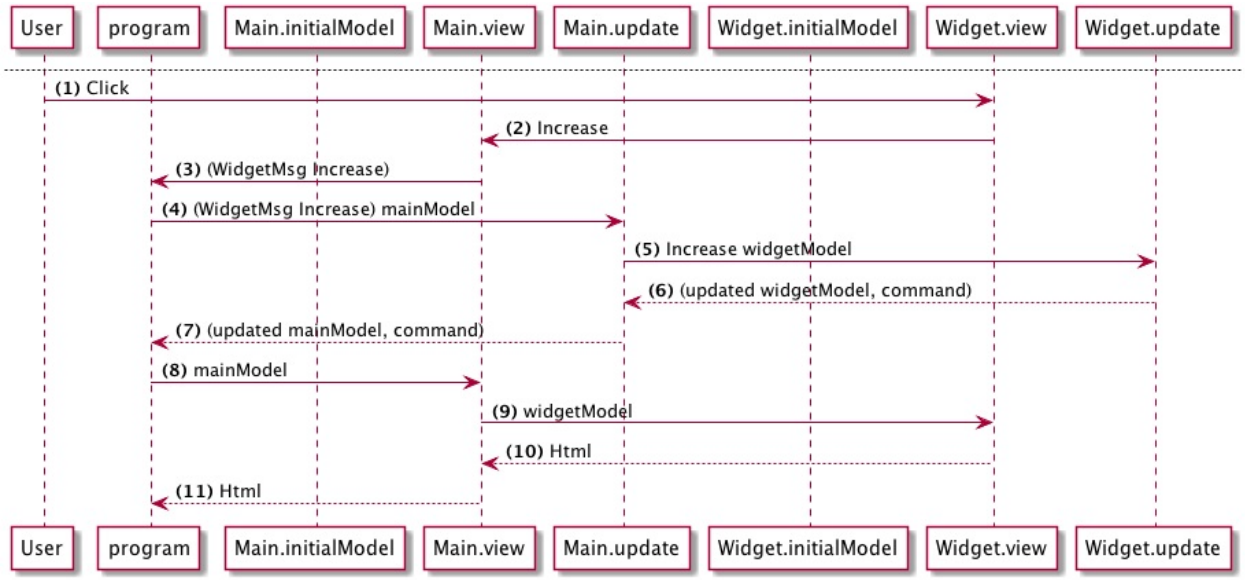
(6)**Main.view**はメインモデルから**widgetModel**を渡して**Widget.view**を呼び出します

(7)**Widget.view**はレンダリングされたHTMLを**Main**に返します

(8)**Main.view**はレンダリングされたHTMLを**program**に返します

(9)**program**はこれをブラウザにレンダリングします

ユーザーインタラクション



(1)ユーザが**増加**ボタンをクリックする

(2)**Widget.view**は、**Main.view**によって拾われる**Increase**メッセージを発行します。

(3)**Main.view**はこのメッセージにタグを付けて(WidgetMsg Increase)、**program**に送信します

(4)このメッセージとメインモデルで**program**が**Main.update**を呼び出す

(5)メッセージに**WidgetMsg**というタグが付いているので、**Main.update**は更新を**Widget.update**に委譲し、メインモデル一部である**widgetModel**部分を渡します

(6)**Widget.update**は、指定されたメッセージ(この場合は**Increase**)に従ってモデルを変更し、修正された**widgetModel**とコマンドを返します

(7)**Main.update**はメインモデルを更新し、それを**program**に返します

(8)**program**は、更新されたメインモデルを渡してビューを再びレンダリングします

キーポイント

- Elmアーキテクチャは、必要なだけ多くのレベルでコンポーネントを合成する(または入れ子にする)クリーンな方法を提供します。
- 子コンポーネントは、その親について何も知る必要はありません。子コンポーネントは、独自の型とメッセージを定義します。
- 子コンポーネントが特に何か(例えば、追加のモデル)を必要とする場合、関数シグネチャを使用して「必要である」と宣言します。親は、子供が必要とするものを提供する責任があります。
- 親は子どものモデルに何が含まれているのか、そのメッセージが何であるかを知る必要はありません。必要なのは、子供たちが求めているものを提供することだけです。

Subscriptions and Commands

外部入力を待ち受けて副作用を作成するためにサブスクリプション、コマンドについて学びます。

この章の内容:

- サブスクリプション(購読)
- コマンド

> This page covers Elm 0.18

サブスクリプション(購読)

Elmでは、サブスクリプションを使用すると、アプリケーションが外部入力を待ち受ける方法が決まります。いくつかの例があります：

- キーボードイベント
- マウスの動き
- ブラウザの閲覧ロケーションの変更
- Websocketイベント

これらを説明するために、キーボードイベントとマウスイベントの両方に応答するアプリケーションを作成しましょう。

まず必要なライブラリをインストールします。

```
elm package install elm-lang/mouse
elm package install elm-lang/keyboard
```

そして以下のプログラムを作成します。

```
module Main exposing (..)

import Html exposing (Html, div, text, program)
import Mouse
import Keyboard



-- モデル


type alias Model =
    Int


init : ( Model, Cmd Msg )
init =
    ( 0, Cmd.none )



-- メッセージ


type Msg
    = MouseMsg Mouse.Position
    | KeyMsg Keyboard.KeyCode
```

```
-- VIEW


view : Model -> Html Msg
view model =
    div []
        [ text (toString model) ]



-- 更新


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        MouseMsg position ->
            ( model + 1, Cmd.none )

        KeyMsg code ->
            ( model + 2, Cmd.none )



-- サブスクリプション(購読)


subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.batch
        [ Mouse.clicks MouseMsg
        , Keyboard.downs KeyMsg
        ]


-- MAIN


main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

このプログラムをElm reactorで実行すると、マウスをクリックするたびにカウンターが1ずつ増えます。キーを押すたびにカウンターが2ずつ増加します。

このプログラムのサブスクリプションに関連する重要な部分を見てみましょう。

メッセージ

```
type Msg
    = MouseMsg Mouse.Position
    | KeyMsg Keyboard.KeyCode
```

`MouseMsg` と `KeyMsg` という2つのメッセージがあります。マウスまたはキーボードが押されたときに応じてトリガされます。

更新

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        MouseMsg position ->
            ( model + 1, Cmd.none )

        KeyMsg code ->
            ( model + 2, Cmd.none )
```

update関数は、それぞれのメッセージに異なる反応をします。すなわち、マウスを押すと1ずつ増やし、キーを押したときには2ずつ増やします。

サブスクリプション(購読)

```
subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.batch ❸
        [ Mouse.clicks MouseMsg ❶
        , Keyboard.downs KeyMsg ❷
        ]
```

ここでは、待ち受けたいことを宣言します。 `Mouse.clicks` ❶と `Keyboard.downs` ❷を待ち受けたいと思っています。これらの関数はどちらも引数にメッセージコンストラクタを取り、サブスクリプションを返します。

両方を待ち受けることができるように `Sub.batch` ❸を使います。バッチはサブスクリプションのリストを取り、そのすべてを含む1つのサブスクリプションを返します。

> This page covers Elm 0.18

コマンド

Elmでは、コマンド(Cmd)は、副作用を伴うものを実行するようにランタイムに指示する方法です。例えば：

- 乱数を生成する。
- httpリクエストを作成する。
- 何かをローカルストレージに保存する。

などです。`Cmd`は、行うべき事柄の1つ、または集合です。私たちはコマンドを使用して、実行する必要があるすべてのものを収集し、それらをランタイムに渡します。ランタイムはそれらを実行し、結果をアプリケーションにフィードバックします。

Elmなどの関数型言語では、すべての関数は値を返します。伝統的な意味での関数の副作用は言語設計によって禁止されており、Elmはそれらをモデリングするための代替アプローチを採用しています。本質的には、関数は、望む効果を表すコマンドの値を返します。 Elmアーキテクチャで使ってきたmainのHtml.programプログラムは、このコマンド値を最終的に受けとります。 Html.programプログラムのupdateメソッドには、名前付きコマンドを実行するロジックが含まれています。

コマンドを使ったサンプルアプリケーションを試してみましょう：

```
module Main exposing (..)

import Html exposing (Html, div, button, text, program)
import Html.Events exposing (onClick)
import Random


-- モデル


type alias Model =
    Int


init : ( Model, Cmd Msg )
init =
    ( 1, Cmd.none )


-- メッセージ


type Msg
    = Roll
    | OnResult Int
```

```
-- VIEW


view : Model -> Html Msg
view model =
    div []
        [ button [ onClick Roll ] [ text "Roll" ]
        , text (toString model)
        ]



-- 更新


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Roll ->
            ( model, Random.generate OnResult (Random.int 1 6) )

        OnResult res ->
            ( res, Cmd.none )


-- MAIN


main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = (always Sub.none)
        }
```

このアプリケーションを実行すると、クリックするたびに乱数を生成するボタンが表示されます。

関連する部分を見てみましょう：

メッセージ

```
type Msg
    = Roll
    | OnResult Int
```

アプリケーションには2つのメッセージがあります。次に新しい数字を表示するための `Roll` 。生成された数値を `Random` ライブラリから取得するための `OnResult` です。

更新

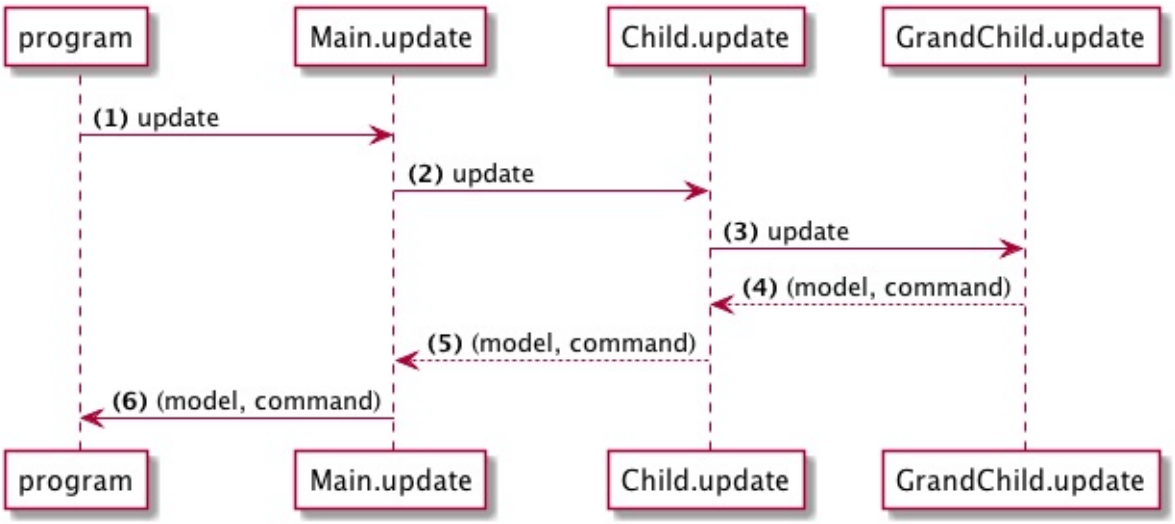
```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Roll ->
            ( model, Random.generate❶ OnResult (Random.int 1 6) )

        OnResult res ->
            ( res, Cmd.none )
```

❶ `Random.generate` は乱数を生成するコマンドを作成します。この関数は、最初の引数がアプリケーションにフィードバックされるメッセージのコンストラクタである必要があります。この場合、コンストラクタは `OnResult` です。

コマンドが実行されると、Elmは生成された数字で `OnResult` を呼び出し、例えば `OnResult 2` を生成します。**Html.program**はこのメッセージをアプリケーションに送り返します。

疑問に思うかもしれませんが、`OnResult res` はメッセージOnResultを示します。この場合、情報の追加ペイロードInteger 'res'を含みます。これは「パラメータ化された型」として知られているパターンです。

多くのネストされたコンポーネントを持つより大きなアプリケーションでは、一度に多くのコマンドを**Html.program**に送信する可能性があります。次の図を見てください：



ここでは、3つの異なるレベルからコマンドを収集します。最後に、これらのコマンドをすべて**Elm.program**に送信して実行します。

アプリケーションの開始

この章では、Elmアプリケーションの例を作成します。

このチュートリアルでは、アプリケーションの構築に関する次の側面について説明します。

- アプリケーションの構造
- リソースの取得
- ビュー
- ルーティング
- ユーザーの操作と変更の保存

アプリケーションの詳細については、次のページを参照してください。

> This page covers Elm 0.18

計画

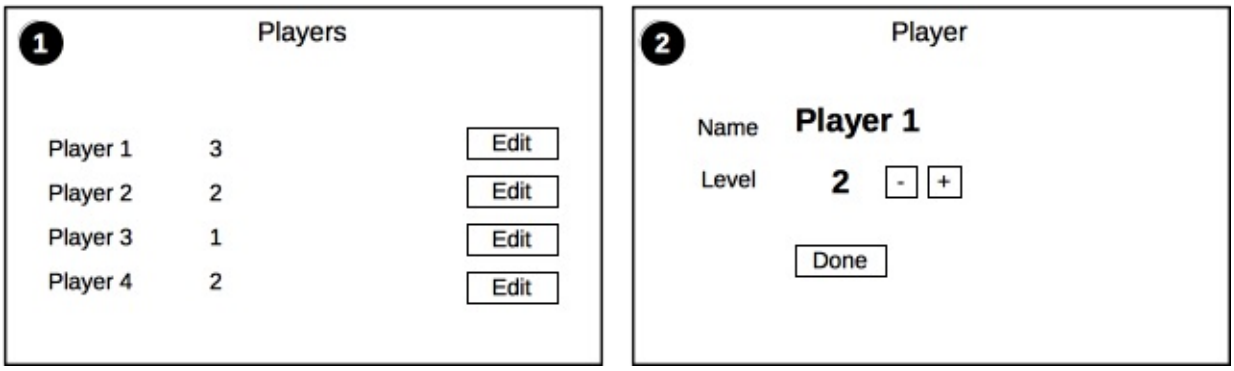
私たちは、架空のロールプレイングゲームを追跡するための基本的なアプリケーションを構築します。

リソース

このガイドの残りの部分では、アプリケーションの対象となるモデル、たとえば、このアプリケーションでのプレイヤーなどをリソースと呼びます。モデルという単語を使用すると、コンポーネント固有の状態もモデル(コンポーネントの展開/折りたたみ状態など)であるため、混乱する可能性があるためです。

ワイヤーフレーム

アプリケーションには2つのビューがあります：



画面❶

選手のリストを表示します。この画面では次のことができます。

- プレーヤーを編集するためにナビゲートする

画面❷

プレイヤーの編集ビューを表示します。この画面では次のことができます。

- レベルを変更する

これは非常に簡単なアプリケーションで、次のことをデモしています。

- 複数のビュー
- ネストされたコンポーネント
- アプリケーションをリソースに分割する
- ルーティング
- アプリケーション全体の共有状態
- レコードの操作の読み取りと編集
- Ajaxリクエスト

> This page covers Elm 0.18

バックエンド

アプリケーションのバックエンドが必要ですが、**json-server**を使用することができます。

json-serverは、偽のAPIを素早く作成できるnpmパッケージです。

新しいノードプロジェクトを開始します。

```
npm init
```

すべてのデフォルト値を受け入れます。

json-serverをインストールします：

```
npm i json-server@0.9 -S
```

プロジェクトのルートに**api.js**を作成します。

```
var jsonServer = require('json-server')

// Returns an Express server
var server = jsonServer.create()

// Set default middlewares (logger, static, cors and no-cache)
server.use(jsonServer.defaults())

var router = jsonServer.router('db.json')
server.use(router)

console.log('Listening at 4000')
server.listen(4000)
```

ルートに**db.json**を追加します：

```
{
  "players": [
    { "id": "1", "name": "Sally", "level": 2 },
    { "id": "2", "name": "Lance", "level": 1 },
    { "id": "3", "name": "Aki", "level": 3 },
    { "id": "4", "name": "Maria", "level": 4 },
    { "id": "5", "name": "Julian", "level": 1 },
    { "id": "6", "name": "Jaime", "level": 1 }
  ]
}
```

次のコマンドを実行してサーバーを起動します。

```
node api.js
```

参照することによってこの偽のAPIをテストしてください：

- http://localhost:4000/players

> This page covers Elm 0.18

Webpackその1

Elm reactorは単純なアプリケーションのプロトタイプ作成には最適ですが、より大きなアプリケーションでは不足です。今のところ、**reactor**は外部JavaScriptとの会話や外部CSSの読み込みをサポートしていません。これらの問題を解決するために、**Webpack**を使用してElmの代わりにElmコードをコンパイルします。

Webpackはコードを束ねるためのツール(バンドラー)です。それはあなたの依存関係の木を見て、インポートされたコードだけを束ねます。Webpackは、バンドル内のCSSやその他のアセットをインポートすることもできます。 Webpackの詳細はこちらにあります。

Webpackと同じように実現するために使用できる多くの選択肢があります。たとえば、次のようなものがあります。

- Browserify
- Gulp
- StealJS
- JSPM
- あるいは、RailsやPhoenixのようなフレームワークを使用している場合は、ElmコードとCSSをバンドルすることができます。

要件

これらのライブラリが期待通りに機能するには、Node JS version 4以上が必要です。

Webpackとローダのインストール

Webpackと関連パッケージをインストールする：

```
npm i webpack@1 webpack-dev-middleware@1 webpack-dev-server@1 elm-webpack-loader@4 file
-loader@0 style-loader@0 css-loader@0 url-loader@0 -S
```

このチュートリアルでは、**webpack**バージョン**1.13**と**elm-webpack-loader**バージョン**4.1**を使用しています。

ローダーは、Webpackが異なるフォーマットをロードできるようにする拡張機能です。例えば。 `css-loader` はwebpackに.cssファイルをロードさせます。

また、いくつかのライブラリを追加したいと思っています：

- CSS用のBasscss、 `ace-css` は、一般的なBasscssスタイルをバンドルするNpmパッケージです
- FontAwesomeのアイコン

```
npm i ace-css@1 font-awesome@4 -S
```

Webpackの設定

ルートに**webpack.config.js**を追加する必要があります：

```
var path = require("path");

module.exports = {
  entry: {
    app: [
      './src/index.js'
    ]
  },

  output: {
    path: path.resolve(__dirname + '/dist'),
    filename: '[name].js',
  },

  module: {
    loaders: [
      {
        test: /\.(css|scss)$/,
        loaders: [
          'style-loader',
          'css-loader',
        ]
      },
      {
        test:    /\.html$/,
        exclude: /node_modules/,
        loader:  'file?name=[name].[ext]',
      },
      {
        test:    /\.elm$/,
        exclude: [/elm-stuff/, /node_modules/],
        loader:  'elm-webpack?verbose=true&warn=true',
      },
      {
        test: /\.woff(2)?(\?v=[0-9]\.[0-9]\.[0-9])?$/,
        loader: 'url-loader?limit=10000&mimetype=application/font-woff',
      },
      {
        test: /\.(ttf|eot|svg)(\?v=[0-9]\.[0-9]\.[0-9])?$/,
        loader: 'file-loader',
      },
    ],

    noParse: /\.elm$/,
  },

  devServer: {
    inline: true,
```

```
    stats: { colors: true },
  },

};
```

留意事項：

- この設定は、Webpack devサーバを作成します。キー `devServer` を参照してください。 Elm reactorの代わりにこのサーバを開発用に使用します。
- アプリケーションのエントリーポイントは `./src/index.js` です、 `entry` キーを見てください。

> This page covers Elm 0.18

Webpackその2

index.html

Elm reactorを使用していないので、アプリケーションを格納するために独自のHTMLを作成する必要があります。 **src/index.html**を以下の内容で作成します：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Elm SPA example</title>
  </head>
  <body>
    <div id="main"></div>
    <script src="/app.js"></script>
  </body>
</html>
```

index.js

これは、Webpackがバンドルを作成するときに探すエントリポイントです。 **src/index.js**を追加：

```
'use strict';

require('ace-css/css/ace.css');
require('font-awesome/css/font-awesome.css');

// index.htmlがdistにコピーされるようにRequireする
require('./index.html');

var Elm = require('./Main.elm');
var mountNode = document.getElementById('main');

// .embed()はオプションの第二引数を取り、プログラム開始に必要なデータを与えられる。たとえばuserIDや何らかのトークンなど
var app = Elm.Main.embed(mountNode);
```

Elmパッケージをインストールする

以下を実行します：

```
elm-package install elm-lang/html
```

ソースディレクトリ

すべてのソースコードを `src` フォルダに追加するので、Elmに依存関係を検索する場所を指定する必要があります。 **elm-package.json**を以下のように変更します：

```
...
"source-directories": [
    "src"
],
...
```

これがなければ、Elmコンパイラはプロジェクトのルートにあるインポートを見つけようとします。

> This page covers Elm 0.18

Webpackその3

最初のElmアプリケーション

基本的なElmアプリケーションを作成します。 **src/Main.elm**では：

```
module Main exposing (..)

import Html exposing (Html, div, text, program)

-- モデル


type alias Model =
    String


init : ( Model, Cmd Msg )
init =
    ( "Hello", Cmd.none )



-- メッセージ


type Msg
    = NoOp



-- ビュー


view : Model -> Html Msg
view model =
    div []
        [ text model ]



-- 更新


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        NoOp ->
            ( model, Cmd.none )
```

```
-- サブスクリプション(購読)


subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none



-- MAIN


main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

> This page covers Elm 0.18

Webpackその4

package.json

最後に、npmスクリプトを追加して、サーバを簡単に実行できるようにします。 **package.json**で `scripts` を以下のように置き換えてください：

```
"scripts": {
    "api": "node api.js",
    "build": "webpack",
    "watch": "webpack --watch",
    "dev": "webpack-dev-server --port 3000"
},
```

- これで `npm run api` がフェイクのサーバーを実行します。
- `npm run build` はwebpackビルドを作成し、バンドルを `dist` に置きます。
- `npm run watch` はwebpackウォッチャーを実行し、ソースコードが変更された時にバンドルを `dist` に置きます。
- `npm run dev` はwebpack dev serverを実行します。

Node Foreman

ApiとFrontendの2つのサーバーがあり、アプリケーションをテストするために手動で両方を起動する必要があります。これでも問題ありませんが、より良い方法があります。

Node Foremanをインストールします:

```
npm install -g foreman
```

プロジェクトのルートに `Procfile` という名前のファイルを作成します：

```
api: npm run api
client: npm run dev
```

これは、同時に処理された両方を起動して終了させる `nf` というcliコマンドを与えます。

テストする

セットアップをテストしましょう。

ターミナルウィンドウで以下を実行します：

```
nf start
```

`http://localhost:3000/` を参照すると、アプリケーションが表示され、"Hello"が出力されます。サーバを終了するには `Ctrl-c` を使います。

アプリケーションコードはhttps://github.com/sporto/elm-tutorial-app/tree/02-webpackのようになります。

> This page covers Elm 0.18

複数のモジュール

アプリケーションはすぐに成長してしまいます。したがって、ファイルを1つのファイルに保存してしまうと、たちまち維持するのが難しくなるでしょう。

循環的な依存関係

直面するかもしれない別の問題は、循環依存です。たとえば、次のような場合があります。

- `Player` 型を定義する `Main` モジュール
- `Main` で宣言された `Player` 型をインポートする `View` モジュール
- ビューをレンダリングする `View` をインポートする `Main`

このとき、循環的な依存が発生しています。

```
Main --> View
View --> Main
```

回避方法は？

この場合、 `Player` 型を `Main` から `Main` と `View` の両方で読み込むことができるようにする必要があります。

Elmで循環的な依存関係を処理するには、アプリケーションをより小さなモジュールに分割するのが最も簡単です。この特定の例では、 `Main` と `View` の両方でインポートできる別のモジュールを作成できます。私たちは3つのモジュールを持つことになります：

- メイン
- ビュー
- モデル(`Player` 型を含む)

これで依存関係は次のようになります。

```
Main --> Models
View --> Models
```

これで循環依存はなくなりました。

messages、**models**、**commands**、**utilities**などのモジュール用に別々のモジュールを作成してみてください。モジュールは通常、多くのコンポーネントからインポートされます。

小さなモジュールでアプリケーションを分割しましょう：

src/Messages.elm

```
module Messages exposing (..)


type Msg
    = NoOp
```

src/Models.elm

```
module Models exposing (..)


type alias Model =
    String
```

src/Update.elm

```
module Update exposing (..)

import Messages exposing (Msg(..))
import Models exposing (Model)


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        NoOp ->
            ( model, Cmd.none )
```

src / View.elm

```
module View exposing (..)

import Html exposing (Html, div, text, program)
import Messages exposing (Msg)
import Models exposing (Model)
import View exposing (view)

view : Model -> Html Msg
view model =
    div []
        [ text model ]
```

src / Main.elm

```
module Main exposing (..)

import Html exposing (Html, div, text, program)
import Messages exposing (Msg)
import Models exposing (Model)
import Update exposing (update)
import View exposing (view)


init : ( Model, Cmd Msg )
init =
    ( "Hello", Cmd.none )


subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none



-- MAIN


main : Program Never Model Msg
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

コードはこちらにありますhttps://github.com/sporto/elm-tutorial-app/tree/018-03-multiple-modules

---

たくさんのモジュールを作成しましたが、簡単なアプリケーションには過剰でしょう。しかし、より大きなアプリケーションでは、分割すると作業がより簡単になります。

リソース

これまでのコードはhttps://github.com/sporto/elm-tutorial-app/tree/03-multiple-modulesのようになります。

この章では、最初のリソースPlayersをアプリケーションに追加します。

Playersリソース

アプリケーション構造を、アプリケーションのリソース名で整理します。 このアプリでは、1つのリソース(`Players`)しか持たないので、 `Players` ディレクトリしか存在しません。

`Players` ディレクトリには、メインレベルのモジュールと同じように、Elmアーキテクチャのコンポーネントごとに1つのモジュールがあります。

- Players/Messages.elm
- Players/Models.elm
- Players/Update.elm

しかし、我々はプレイヤーのための異なるビューを持つことになります。リストビューと編集ビューです。 各ビューには独自のElmモジュールがあります。

- Players/List.elm
- Players/Edit.elm

> This page covers Elm 0.18

Playersモジュール

Playersメッセージ

src/Players/Messages.elmを作成する。

```
module Players.Messages exposing (..)


type Msg
    = NoOp
```

ここでは、プレイヤーに関連するすべてのメッセージを記入します。

Playersモデル

src/Players/Models.elmを作成する。

```
module Players.Models exposing (..)


type alias PlayerId =
    String


type alias Player =
    { id : PlayerId
    , name : String
    , level : Int
    }


new : Player
new =
    { id = "0"
    , name = ""
    , level = 1
    }
```

ここでは、プレーヤーのレコードの外観を定義します。 ID、名前、レベルがあります。

`PlayerId` の定義にも注意してください。これは `String` の単なるエイリアスです。これは、複数のIDを引数に取る関数を後で導入することになった場合にわかりやすくなり便利です。例えば：

```
addPerkToPlayer : Int -> String -> Player
```

よりも、次のように書かれているとはるかに明確です。

```
addPerkToPlayer : PerkId -> PlayerId -> Player
```

プレーヤーの更新

src/Players/Update.elmを追加する

```
module Players.Update exposing (..)

import Players.Messages exposing (Msg(..))
import Players.Models exposing (Player)


update : Msg -> List Player -> ( List Player, Cmd Msg )
update message players =
    case message of
        NoOp ->
            ( players, Cmd.none )
```

この `update` は現時点では何もしません。

これらは、より大きなアプリケーションのすべてのリソースが従う基本パターンです。

```
Messages
Models
Update
Players
    Messages
    Models
    Update
Perks
    Messages
    Models
    Update
...
```

> This page covers Elm 0.18

Playersリスト

src/Players/List.elmを作成する。

```
module Players.List exposing (..)

import Html exposing (..)
import Html.Attributes exposing (class)
import Players.Messages exposing (..)
import Players.Models exposing (Player)


view : List Player -> Html Msg
view players =
    div []
        [ nav players
        , list players
        ]


nav : List Player -> Html Msg
nav players =
    div [ class "clearfix mb2 white bg-black" ]
        [ div [ class "left p2" ] [ text "Players" ] ]


list : List Player -> Html Msg
list players =
    div [ class "p2" ]
        [ table []
            [ thead []
                [ tr []
                    [ th [] [ text "Id" ]
                    , th [] [ text "Name" ]
                    , th [] [ text "Level" ]
                    , th [] [ text "Actions" ]
                    ]
                ]
            , tbody [] (List.map playerRow players)
            ]
        ]


playerRow : Player -> Html Msg
playerRow player =
    tr []
        [ td [] [ text player.id ]
        , td [] [ text player.name ]
        , td [] [ text (toString player.level) ]
        , td []
            []
        ]
```

このビューにはプレイヤーのリストが表示されます。

> This page covers Elm 0.18

メイン

メインレベルは私たちが作成したPlayersモジュールに接続する必要があります。

以下のように関連付ける必要があります：

```
メインメッセージ   ---> プレイヤーメッセージ
メインモデル       ---> プレイヤーモデル
メイン更新         ---> プレイヤー更新
```

メインメッセージ

src/Messages.elmを変更してプレイヤーメッセージを追加する：

```
module Messages exposing (..)

import Players.Messages


type Msg
    = PlayersMsg Players.Messages.Msg
```

主なモデル

プレイヤーを含めるように**src/Models.elm**を変更する：

```
module Models exposing (..)

import Players.Models exposing (Player)


type alias Model =
    { players : List Player
    }


initialModel : Model
initialModel =
    { players = [ Player "1" "Sam" 1 ]
    }
```

ここでは、1名のプレーヤーをハードコーディングしておきます。

メインの更新

src/Update.elmを次のように変更します。

```
module Update exposing (..)

import Messages exposing (Msg(..))
import Models exposing (Model)
import Players.Update


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        PlayersMsg subMsg ->
            let
                ( updatedPlayers, cmd ) =
                    Players.Update.update subMsg model.players
            in
                ( { model | players = updatedPlayers }, Cmd.map PlayersMsg cmd )
```

ここではElmアーキテクチャに従います：

- すべてのPlayersMsgは `Players.Update` にルーティングされます。
- パターンマッチングを使用して `Players.Update` の結果を抽出します
- 更新されたプレーヤーリストと実行する必要のあるコマンドを含むモデルを返します。

> This page covers Elm 0.18

メインビュー

src/View.elmを変更して、プレーヤーのリストを追加します：

```
module View exposing (..)

import Html exposing (Html, div, text)
import Messages exposing (Msg(..))
import Models exposing (Model)
import Players.List


view : Model -> Html Msg
view model =
    div []
        [ page model ]


page : Model -> Html Msg
page model =
    Html.map PlayersMsg (Players.List.view model.players)
```

> This page covers Elm 0.18

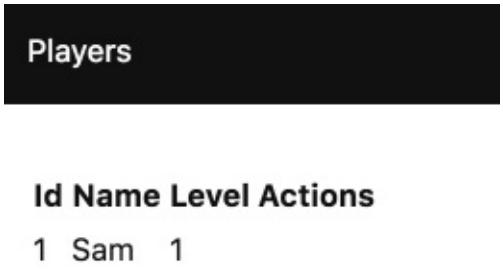
メイン

最後に `initialModel` を呼び出すように**src/Main.elm**を修正します：

```
module Main exposing (..)

import Html exposing (Html, div, text, program)
import Messages exposing (Msg)
import Models exposing (Model, initialModel)
import Update exposing (update)
import View exposing (view)

init : ( Model, Cmd Msg )
init =
    ( initialModel, Cmd.none )

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- MAIN

main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

ここでは、インポートに `initialModel` を追加し、 `init` を追加しました。

アプリケーションを実行すると、1人のユーザーのリストが表示されます。



アプリケーションはhttps://github.com/sporto/elm-tutorial-app/tree/04-resourcesのようになります。

リソースの取得

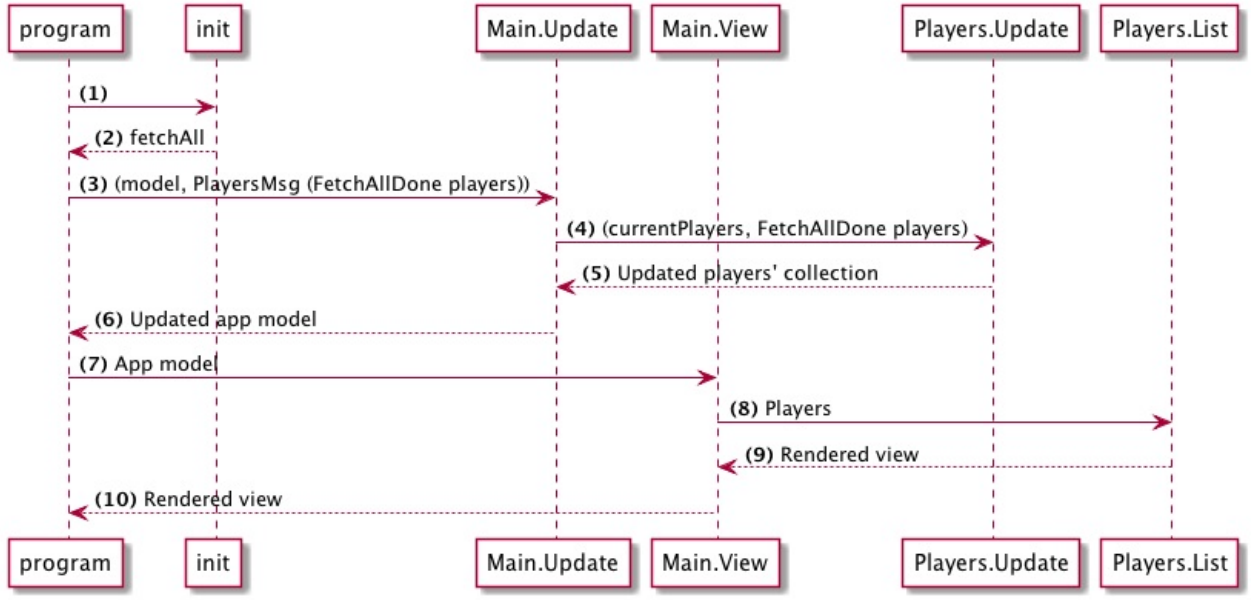
この章では、偽のAPIからプレーヤーのコレクションを取得する方法について説明します。

これまでのアプリケーションコードは、https://github.com/sporto/elm-tutorial-app/tree/04-resourcesのようになります。

計画

次のステップは、前に作成した偽のAPIからプレーヤのリストを取得することです。

これは計画です：



(1-2)アプリケーションがロードされると、Httpリクエストを開始してプレイヤーを取得するコマンドを起動します。 これはHtml.programの `init` で行われます。

(3-6)リクエストが完了すると、データとともに「FetchAllDone」をトリガーします。このメッセージはプレーヤーのコレクションを更新する `Players.Update` に流れます。

(7-10)その後、アプリケーションは更新されたプレーヤーのリストをレンダリングします。

依存関係

`http` モジュールが必要なのでインストールしておきます：

```
elm package install elm-lang/http
```

> This page covers Elm 0.18

Playersメッセージ

まず、プレイヤーを取得するために必要なメッセージを作成しましょう。**src/Players/Messages.elm**に新しいインポートとメッセージを追加します。

```
module Players.Messages exposing (..)

import Http
import Players.Models exposing (PlayerId, Player)


type Msg
    = OnFetchAll (Result Http.Error (List Player))
```

`OnFetchAll`はサーバからの応答を取得するときに呼び出されます。このメッセージは`Http.Error`かフェッチされたプレイヤーのリストのいずれかになる`Result`を保持します。

> This page covers Elm 0.18

プレーヤーの更新

プレーヤーのリクエストが完了すると、 `OnFetchAll` メッセージがトリガーされます。

src/Players/Update.elmは、この新しいメッセージに責任を持つ必要があります。 `update` を以下のように変更してください：

```
...
update : Msg -> List Player -> ( List Player, Cmd Msg )
update message players =
    case message of
        OnFetchAll (Ok newPlayers) ->
            ( newPlayers, Cmd.none )

        OnFetchAll (Err error) ->
            ( players, Cmd.none )
```

`OnFetchAll` というメッセージを得たとき、何を実行するかを決定するためにパターンマッチングを使用できます。

- `Ok` の場合、取得されたプレーヤーを返して、プレーヤーのコレクションを更新します。
- `Err` の場合、私たちは今までに持っていたものを返すだけです（もっと良いのはユーザーにエラーを表示することですが、チュートリアルを簡単にするために省略します)。

> This page covers Elm 0.18

Playersコマンド

サーバーからプレーヤーを取得するためのタスクとコマンドを作成する必要があります。
src/Players/Commands.elmを作成します。

```
module Players.Commands exposing (..)

import Http
import Json.Decode as Decode exposing (field)
import Task
import Players.Models exposing (PlayerId, Player)
import Players.Messages exposing (..)


fetchAll : Cmd Msg
fetchAll =
    Http.get fetchAllUrl collectionDecoder
        |> Http.send OnFetchAll


fetchAllUrl : String
fetchAllUrl =
    "http://localhost:4000/players"


collectionDecoder : Decode.Decoder (List Player)
collectionDecoder =
    Decode.list memberDecoder


memberDecoder : Decode.Decoder Player
memberDecoder =
    Decode.map3 Player
        (field "id" Decode.string)
        (field "name" Decode.string)
        (field "level" Decode.int)
```

このコードを見てみましょう。

```
fetchAll : Cmd Msg
fetchAll =
    Http.get fetchAllUrl collectionDecoder
        |> Http.send OnFetchAll
```

ここでは、アプリケーションを実行するためのコマンドを作成します。

- `Http.get` が `Request` を作成します
- 次に、このrequestを `Http.send` に送ります。このタスクはコマンドでラップします

```
collectionDecoder : Decode.Decoder (List Player)
collectionDecoder =
    Decode.list memberDecoder
```

このデコーダは、リストの各メンバーのデコードを `memberDecoder` に委譲します

```
memberDecoder : Decode.Decoder Player
memberDecoder =
    Decode.map3 Player
        (field "id" Decode.string)
        (field "name" Decode.string)
        (field "level" Decode.int)
```

`memberDecoder` は `Player` レコードを返すJSONデコーダを作成します。

デコーダの仕組みを理解するために、Elm replを使って遊んでみましょう。

ターミナルで `elm repl` を実行します。 Json.Decoderモジュールをインポートします。

```
> import Json.Decode exposing (..)
```

次に、Json文字列を定義します。

```
> json = "{\"id\":99, \"name\":\"Sam\"}"
```

また、 `id` を抽出するデコーダを定義します。

```
> idDecoder = (field "id" int)
```

これは、文字列が `id` キーを抽出してそれを整数としてパースするデコーダを作成します。

このデコーダを実行すると結果が表示されます。

```
> result = decodeString idDecoder json
OK 99 : Result.Result String Int
```

`Ok 99` はデコードが成功し、99が得られたことを意味しています。これは `(field "id" Decode.int)` がやったことであり、単一のキーのデコーダを作成します。

これは方程式の一部です。つぎに2番目の部分をやってみましょう。まず型を定義してください：

```
> type alias Player = { id: Int, name: String }
```

Elmでは、レコード型のエイリアス型名を関数として呼び出すことでレコードを作成することができます。たとえば、`Player 1 "Sam"`はプレーヤーレコードを作成します。パラメータの順序は他の関数と同様に重要であることに注意してください。

次を試してみてください：

```
> Player 1 "Sam"
{ id = 1, name = "Sam" } : Repl.Player
```

これらの2つのコンセプトで完全なデコーダを作成しましょう：

```
> nameDecoder = (field "name" string)
> playerDecoder = map2 Player idDecoder nameDecoder
```

`map2`は最初の引数(この場合Player)と2つのデコーダとしての関数をとります。次に、デコーダを実行し、その結果を関数の引数(Player)に渡します。

試してみましょう：

```
> result = decodeString playerDecoder json
Ok { id = 99, name = "Sam" } : Result.Result String Repl.Player
```

---

programにコマンドを送信するまで、実際には実行されません。

> This page covers Elm 0.18

メイン

メインのモデル

src/Models.elmのハードコーディングされたプレイヤーのリストを削除します。

```
initialModel : Model
initialModel =
    { players = []
    }
```

メイン

最後に、アプリケーションを起動するときに `fetchAll` を実行します。

src/Main.elmを以下のように修正します：

```
...
import Messages exposing (Msg(..))
...
import Players.Commands exposing (fetchAll)

init : ( Model, Cmd Msg )
init =
    ( initialModel, Cmd.map PlayersMsg fetchAll )
```

これで `init` はアプリケーションの起動時に実行するコマンドのリストを返します。

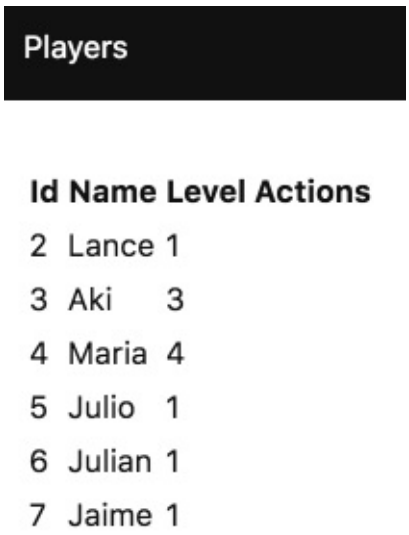
> This page covers Elm 0.18

試してみよう

さあ試してみましょう！ ある端末でアプリケーションを実行するには：

```
nf start
```

ブラウザをリフレッシュすると、アプリケーションはサーバーからプレーヤーのリストを取得するようになりました。 アプリは次のようになります：



アプリケーションコードはhttps://github.com/sporto/elm-tutorial-app/tree/018-05-fetchのようにこの段階を見なければなりません。

ルーティング

この章では、アプリケーションへのルーティングの追加について説明します。

これまでのアプリケーションコードは、https://github.com/sporto/elm-tutorial-app/tree/05-fetchのようになります。

> This page covers Elm 0.18

ルーティングの紹介

アプリケーションにルーティングを追加しましょう。 Elm Navigation packageとUrlParserを使用します。

- ナビゲーションはブラウザの場所を変更し、変更に対応する手段を提供します
- UrlParserはルートマッチャーを提供します

最初にパッケージをインストールします。

```
elm package install elm-lang/navigation 1.0.0
elm package install evancz/url-parser 1.0.0
```

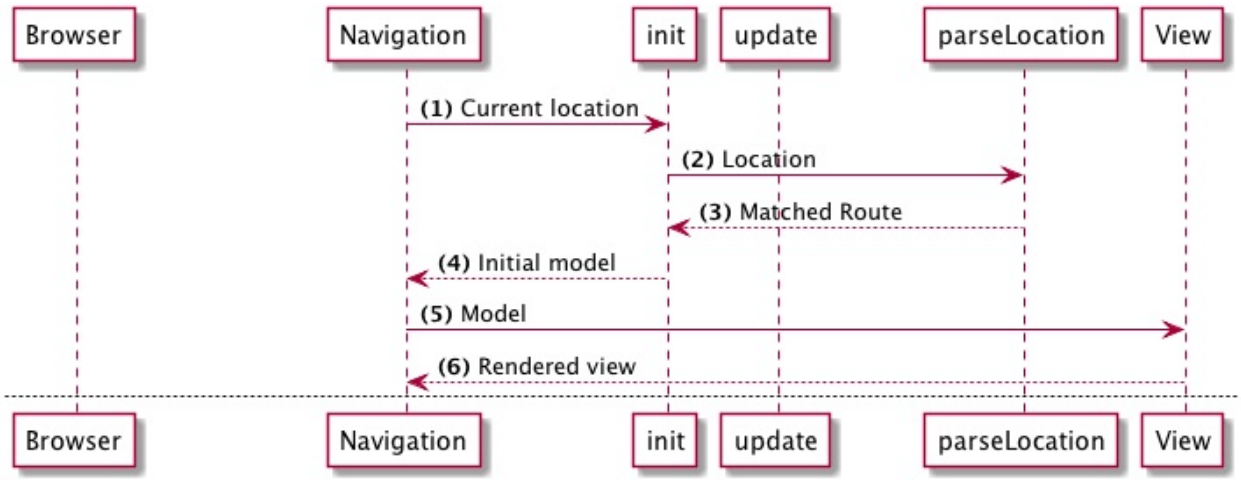
`Navigation` は `Html.program` をラップするライブラリです。 `Html.program` のすべての機能といくつかの余分な機能を備えています：

- ブラウザ上でのロケーションの変更を待ち受る
- 場所が変更されたときにメッセージをトリガする
- ブラウザの場所を変更する方法を提供する

フロー

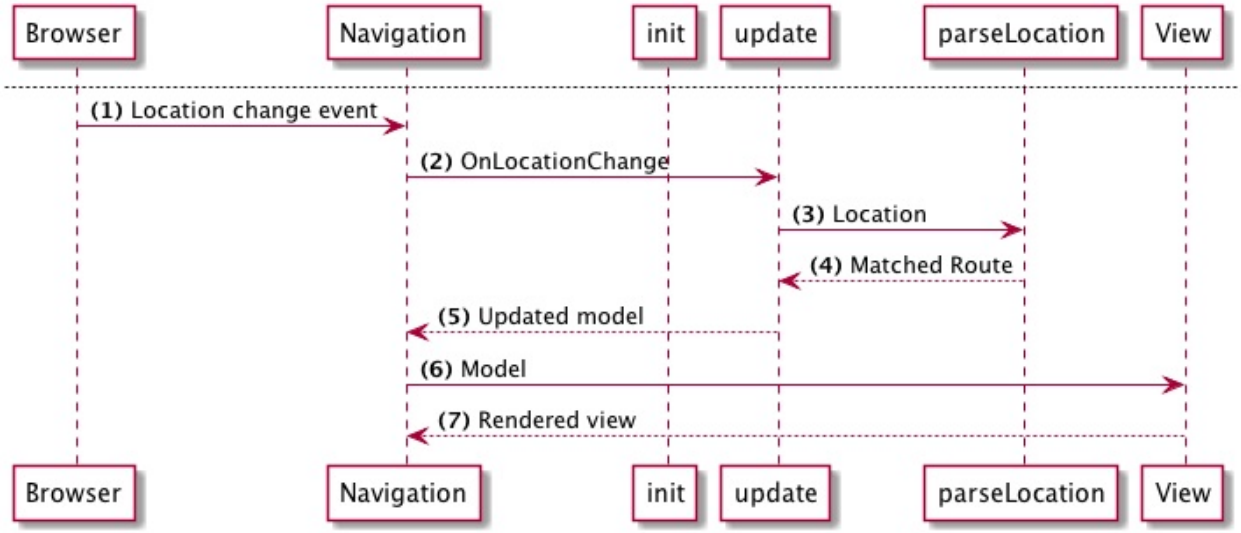
ルーティングの仕組みを理解するための図をいくつか示します。

初期レンダリング



(1) ページが最初に読み込まれるとき、 `Navigation` モジュールは現在の `Location` を取得し、それをアプリケーションの `init` 関数に送ります。 (2) `init` 関数中でこのlocationをパースし一致する `Route` を得ます。 (3,4) 一致した `Route` をアプリケーションの初期モデルに保存し、このモデルを `Navigation` に返します。 (5,6) この初期モデルを送ることで `Navigation` はビューをレンダリングします。

ロケーションが変更されたとき



(1) ブラウザの閲覧ロケーションが変更されると、ナビゲーション・ライブラリーはイベントを受け取ります (2) `OnLocationChange` メッセージが `update` 関数に送られます。このメッセージには新しい `Location` が含まれます。(3,4) `update` では、 `Location` を解析し、一致する `Route` を返します。(5) `update` からアップデート `Route` を含む更新されたモデルを返します。(6,7) `Navigation` は、アプリケーションを通常通りレンダリングします

> This page covers Elm 0.18

ルーティング

アプリケーションのルーティング設定を定義するモジュール**src/Routing.elm**を作成します。

このモジュールでは以下を定義します：

- アプリケーションのルート
- パスマッチャーを使用してブラウザパスをルートにマッチさせる方法
- ルーティングメッセージに反応する方法

src/Routing.elmでは：

```
module Routing exposing (..)

import Navigation exposing (Location)
import Players.Models exposing (PlayerId)
import UrlParser exposing (..)


type Route
    = PlayersRoute
    | PlayerRoute PlayerId
    | NotFoundRoute


matchers : Parser (Route -> a) a
matchers =
    oneOf
        [ map PlayersRoute top
        , map PlayerRoute (s "players" </> string)
        , map PlayersRoute (s "players")
        ]


parseLocation : Location -> Route
parseLocation location =
    case (parseHash matchers location) of
        Just route ->
            route

        Nothing ->
            NotFoundRoute
```

このモジュールを見てみましょう。

ルート

```
type Route
    = PlayersRoute
    | PlayerRoute PlayerId
    | NotFoundRoute
```

これらはアプリケーションで利用可能なルートです。`NotFoundRoute`は、ブラウザパスと一致するルートがない場合に使用されます。

マッチャー

```
matchers : Parser (Route -> a) a
matchers =
    oneOf
        [ map PlayersRoute top
        , map PlayerRoute (s "players" </> string)
        , map PlayersRoute (s "players")
        ]
```

ここではルートマッチャーを定義します。これらのパーサは、url-parserライブラリによって提供されます。

3つのマッチャーが必要です：

- `PlayersRoute`に解決されるトップのルートのもの
- `PlayersRoute`にも解決される`/players`のためのもの
- `PlayerRoute id`に解決される`/players/id`のためのもの

順序は重要であることに注意してください。

このライブラリの詳細はこちらをご覧くださいhttp://package.elm-lang.org/packages/evancz/url-parser。

パースロケーション

```
parseLocation : Location -> Route
parseLocation location =
    case (parseHash matchers location) of
        Just route ->
            route

        Nothing ->
            NotFoundRoute
```

ブラウザ閲覧ロケーションが変わるたびに、ナビゲーション・ライブラリーは`Navigation.Location`レコードを含むメッセージをトリガーします。mainの`update`からこのレコードを引数として`parseLocation`を呼び出します。

`parseLocation`はこの`Location`レコードを解析し、可能ならば`Route`を返す関数です。すべてのマッチャーが失敗した場合は、`NotFoundRoute`を返します。

この場合、ハッシュを使用してルーティングするので、`UrlParser.parseHash`を実行します。代わりに`UrlParser.parsePath`を使ってパスを使ってルーティングすることもできます。

> This page covers Elm 0.18

プレイヤー編集ビュー

`/players/3` を打ったときに表示する新しいビューが必要です。

src/Players/Edit.elmを作成します：

```
module Players.Edit exposing (..)

import Html exposing (..)
import Html.Attributes exposing (class, value, href)
import Players.Messages exposing (..)
import Players.Models exposing (..)


view : Player -> Html Msg
view model =
    div []
        [ nav model
        , form model
        ]


nav : Player -> Html Msg
nav model =
    div [ class "clearfix mb2 white bg-black p1" ]
        []


form : Player -> Html Msg
form player =
    div [ class "m3" ]
        [ h1 [] [ text player.name ]
        , formLevel player
        ]


formLevel : Player -> Html Msg
formLevel player =
    div
        [ class "clearfix py1"
        ]
        [ div [ class "col col-5" ] [ text "Level" ]
        , div [ class "col col-7" ]
            [ span [ class "h2 bold" ] [ text (toString player.level) ]
            , btnLevelDecrease player
            , btnLevelIncrease player
            ]
        ]
```

```
btnLevelDecrease : Player -> Html Msg
btnLevelDecrease player =
    a [ class "btn ml1 h1" ]
        [ i [ class "fa fa-minus-circle" ] [] ]


btnLevelIncrease : Player -> Html Msg
btnLevelIncrease player =
    a [ class "btn ml1 h1" ]
        [ i [ class "fa fa-plus-circle" ] [] ]
```

このビューには、プレーヤーのレベルのフォームが表示されます。今のところ、実装を後まわしにした `btnLevelIncrease` などのいくつかのダミーボタンがあります。

> This page covers Elm 0.18

メインモデル

私たちのメインのアプリケーションモデルでは、現在のルートを保存します。 **src/Models.elm**を次のように変更します。

```
module Models exposing (..)

import Players.Models exposing (Player)
import Routing


type alias Model =
    { players : List Player
    , route : Routing.Route
    }


initialModel : Routing.Route -> Model
initialModel route =
    { players = []
    , route = route
    }
```

ここで我々は：

- モデルに `route` を追加しました
- `initialModel` を、引数 `route` を取るように変更しました

> This page covers Elm 0.18

メインアップデート

新規の `OnLocationChange` メッセージを処理するMainの `update` 関数です。

src/Update.elmに新しい分岐の枝を追加します:

```
...
import Routing exposing (parseLocation)

...

update msg model =
    case msg of
        ...
        OnLocationChange location ->
            let
                newRoute =
                    parseLocation location
            in
                ( { model | route = newRoute }, Cmd.none )
```

ここで、 `OnLocationChange` メッセージを受け取ると、このlocationをパースし、一致したルートをモデルに格納します。

> This page covers Elm 0.18

メインビュー

主なアプリケーションビューでは、ブラウザの場所を変更するときに別のページを表示する必要があります。

src/View.elmを次のように変更します。

```
module View exposing (..)

import Html exposing (Html, div, text)
import Messages exposing (Msg(..))
import Models exposing (Model)
import Players.Edit
import Players.List
import Players.Models exposing (PlayerId)
import Routing exposing (Route(..))


view : Model -> Html Msg
view model =
    div []
        [ page model ]


page : Model -> Html Msg
page model =
    case model.route of
        PlayersRoute ->
            Html.map PlayersMsg (Players.List.view model.players)

        PlayerRoute id ->
            playerEditPage model id

        NotFoundRoute ->
            notFoundView


playerEditPage : Model -> PlayerId -> Html Msg
playerEditPage model playerId =
    let
        maybePlayer =
            model.players
                |> List.filter (\player -> player.id == playerId)
                |> List.head
    in
        case maybePlayer of
            Just player ->
                Html.map PlayersMsg (Players.Edit.view player)

            Nothing ->
                notFoundView


notFoundView : Html msg
notFoundView =
    div []
        [ text "Not found"
        ]
```

正しいビューを表示

```
page : Model -> Html Msg
page model =
    case model.route of
        PlayersRoute ->
            Html.map PlayersMsg (Players.List.view model.players)

        PlayerRoute id ->
            playerEditPage model id

        NotFoundRoute ->
            notFoundView
```

これで、 `model.route` に応じた正しいビューを表示するためのcase式を持つ `page` 関数が定義できました。

プレーヤーの編集ルートが一致すると(たとえば `#players/2`)、ルートからプレーヤーID、つまり「PlayerRoute playerId」を取得できます。

プレーヤーを探す

```
playerEditPage : Model -> PlayerId -> Html Msg
playerEditPage model playerId =
    let
        maybePlayer =
            model.players
                |> List.filter (\player -> player.id == playerId)
                |> List.head
    in
        case maybePlayer of
            Just player ->
                Html.map PlayersMsg (Players.Edit.view player)

            Nothing ->
                notFoundView
```

`playerId` を取得できましたが、そのidに対応する実際のプレイヤーレコードを持っていないかもしれません。なのでプレーヤーのリストをIDでフィルタリングし、プレーヤーが見つかったかどうかに応じて正しいビューを表示するcase式を定義します。

notFoundView

`notFoundView` は経路が一致しないときに表示されます。 `Html Msg` ではなく `Html msg` の型に注目してください。これは、このビューはメッセージを生成しないため、特定の型 `Msg` の代わりにジェネリック型変数 `msg` を使用できるからです。

> This page covers Elm 0.18

メイン

最後に、メインモジュールのすべてを配線する必要があります。

src/Main.elmを次のように変更します。

```
module Main exposing (..)

import Navigation
import Messages exposing (Msg(..))
import Models exposing (Model, initialModel)
import View exposing (view)
import Update exposing (update)
import Players.Commands exposing (fetchAll)
import Routing exposing (Route)


init : Result String Route -> ( Model, Cmd Msg )
init result =
    let
        currentRoute =
            Routing.routeFromResult result
    in
        ( initialModel currentRoute, Cmd.map PlayersMsg fetchAll )


subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none


urlUpdate : Result String Route -> Model -> ( Model, Cmd Msg )
urlUpdate result model =
    let
        currentRoute =
            Routing.routeFromResult result
    in
        ( { model | route = currentRoute }, Cmd.none )


main : Program Never
main =
    Navigation.program Routing.parser
        { init = init
        , view = view
        , update = update
        , urlUpdate = urlUpdate
        , subscriptions = subscriptions
        }
```

新しいImport

`Navigation` と `Routing` のためのImportを追加しました。

初期化

```
init : Result String Route -> ( Model, Cmd Msg )
init result =
    let
        currentRoute =
            Routing.routeFromResult result
    in
        ( initialModel currentRoute, Cmd.map PlayersMsg fetchAll )
```

init関数は `Routing` に追加した `parser` から初期的な出力を受け取ります。パーサーの出力は `Result` です。 **Navigation**モジュールは初期ブラウザ閲覧ロケーションを解析し、その結果を `init` に渡します。この**route**初期値をモデルに格納します。

urlUpdate

`urlUpdate` は、ブラウザの場所が変更されるたびに**Navigation**パッケージによって呼び出されます。 `init` のように、ここではパーサの結果を得ます。ここで行うのは、新しい**route**をアプリケーションモデルに格納することだけです。

main

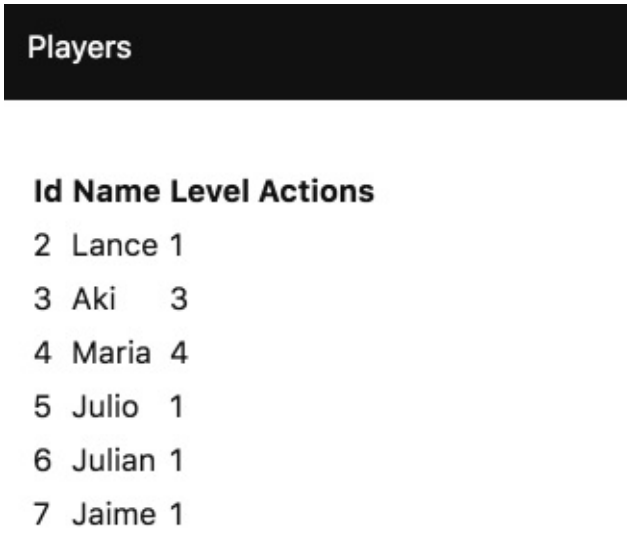
`main` は `Html.program` の代わりに `Navigation.program` を使います。 `Navigation.program` はHtml.programをラップしますが、ブラウザの場所が変更されたときには `urlUpdate` コールバックを追加します。

試してみよう

これまでのことを試してみましょう。 次のようにしてアプリケーションを実行します。

```
nf start
```

ブラウザで `http://localhost:3000` に行きます。 ユーザーの一覧が表示されます。



`http://localhost:3000/#players/2` に行くと、1人のユーザーが表示されます。

次に、ナビゲーションを追加します。

この時点までに、アプリケーションコードはhttps://github.com/sporto/elm-tutorial-app/tree/018-06-routingになります。

> This page covers Elm 0.18

ナビゲーション

次に、ビュー間を移動するためのボタンを追加しましょう。

EditPlayerメッセージ

src/Players/Messages.elmに2つの新しいアクションを追加するように変更してください：

```
...

type Msg
    = OnFetchAll (Result Http.Error (List Player))
    | ShowPlayers
    | ShowPlayer PlayerId
```

`ShowPlayers`と`ShowPlayer`を追加しました。

プレーヤーリスト

プレイヤーのリストは、各プレイヤーが`ShowPlayer`メッセージをトリガーするためのボタンを表示する必要があります。

まず、**src/Players/List.elm**に`Html.Events`を追加してください：

```
import Html.Events exposing (onClick)
```

このボタンの最後に新しい機能を追加します：

```
editBtn : Player -> Html Msg
editBtn player =
    button
        [ class "btn regular"
        , onClick (ShowPlayer player.id)
        ]
        [ i [ class "fa fa-pencil mr1" ] [], text "Edit" ]
```

ここでは、編集したいプレイヤーのIDで`ShowPlayer`をトリガーします。

そして、このボタンを含むように`playerRow`を変更してください：

```
playerRow : Player -> Html Msg
playerRow player =
    tr []
        [ td [] [ text (toString player.id) ]
        , td [] [ text player.name ]
        , td [] [ text (toString player.level) ]
        , td []
            [ editBtn player ]
        ]
```

プレイヤー編集

編集ビューにナビゲーションボタンを追加しましょう。**/src/Players/Edit.elm**で：

1つ以上のインポートを追加します：

```
import Html.Events exposing (onClick)
```

リストボタンの最後に新しい関数を追加します：

```
listBtn : Html Msg
listBtn =
    button
        [ class "btn regular"
        , onClick ShowPlayers
        ]
        [ i [ class "fa fa-chevron-left mr1" ] [], text "List" ]
```

ここでは、ボタンをクリックすると `ShowPlayers` を送ります。

そしてこのボタンをリストに追加し、 `nav` 関数を以下のように変更します：

```
nav : Player -> Html Msg
nav model =
    div [ class "clearfix mb2 white bg-black p1" ]
        [ listBtn ]
```

プレーヤーの更新

最後に、**src/Players/Update.elm**は新しいメッセージに応答する必要があります。

```
import Navigation
```

そして、2つの新しい枝をcase式に追加します。

```
update : Msg -> List Player -> ( List Player, Cmd Msg )
update message players =
    case message of
        ...

        ShowPlayers ->
            ( players, Navigation.newUrl "#players" )

        ShowPlayer id ->
            ( players, Navigation.newUrl ("#players/" ++ id) )
```

`Navigation.newUrl` はコマンドを返します。Elmがこのコマンドを実行すると、ブラウザの閲覧ロケーションが変わります。

テストする

`http://localhost:3000/#players` のリストに行きます。編集ボタンが表示されます。クリックすると、その場所が編集ビューに変わります。

これまでのアプリケーションコードはhttps://github.com/sporto/elm-tutorial-app/tree/018-07-navigationになります。

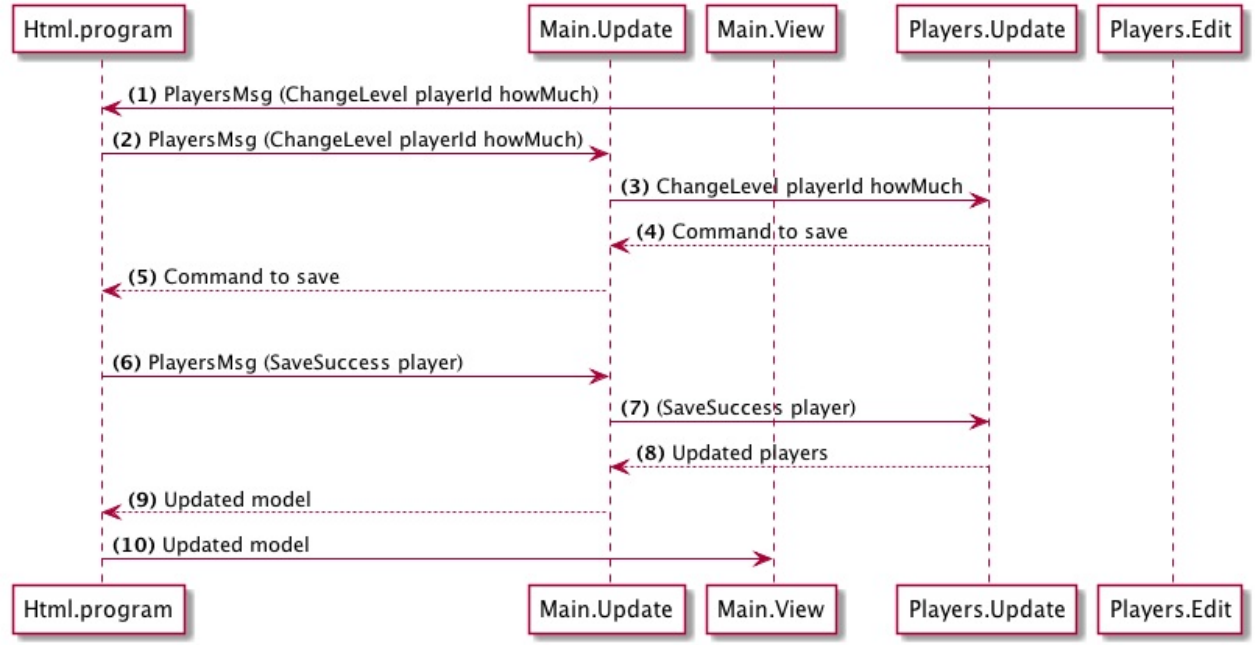
編集

このチュートリアルの最後の章では、プレイヤーのレベルを編集してバックエンドに保存します。

この時点までのアプリケーションコードはhttps://github.com/sporto/elm-tutorial-app/tree/07-navigationのようになります。

計画

プレーヤーのレベルを変更する計画は次のとおりです。



(1) ユーザーが増加、減少のボタンをクリックすると、 `playerId` と `howMuch` をペイロードとして、 `ChangeLevel` メッセージをトリガーします。

(2) **Html.program**(ナビゲーションがラップしている)は、このメッセージを `Main.Update` に送り返し、 `Players.Update` (3)に送ります。

(4) `Players.Update` はプレーヤーを保存するコマンドを返します。このコマンドは**Html.program**(5)に流れます。

(6) Elmランタイムはコマンドを実行し(API呼び出しをトリガする)、保存成功または失敗のいずれかの結果を返します。 成功の場合、更新されたプレーヤーをペイロードとして「SaveSuccess」メッセージをトリガーします。

(7) `Main.Update` は `SaveSuccess` メッセージを `Players.Update` にルーティングします。

(8) `Players.Update` では `players` モデルを更新して返します。 これはHtml.program(9)に戻ります。

(10) その後、Html.programは更新されたモデルでアプリケーションをレンダリングします。

メッセージ

まず、必要なメッセージを追加してみましょう。

src/Players/Messages.elmに以下を追加：

```
type Msg
    ...
    | ChangeLevel PlayerId Int
    | OnSave (Result Http.Error Player)
```

- ユーザーがレベルを変更したいときに `ChangeLevel` がトリガーされます。第2のパラメータは、レベルをどれだけ変化させるかを示す整数です。減少する場合は-1、増加する場合は1になります。
- その後、プレーヤーをAPIに更新するようリクエストします。 `OnSave` はAPIからの成功した応答の後にトリガされます。
- `OnSave` は成功時には更新されたプレイヤーを運び、失敗時にはHttpエラーを運びます。

> This page covers Elm 0.18

プレイヤー編集ビュー

私たちは `ChangeLevel` メッセージを作成しました。このメッセージをプレイヤーの編集ビューからトリガーしましょう。

src/Players/Edit.elmで `btnLevelDecrease` と `btnLevelIncrease` を変更してください：

```
...
btnLevelDecrease : Player -> Html Msg
btnLevelDecrease player =
    a [ class "btn ml1 h1", onClick (ChangeLevel player.id -1) ]
        [ i [ class "fa fa-minus-circle" ] [] ]


btnLevelIncrease : Player -> Html Msg
btnLevelIncrease player =
    a [ class "btn ml1 h1", onClick (ChangeLevel player.id 1) ]
        [ i [ class "fa fa-plus-circle" ] [] ]
```

これらの2つのボタンに `onClick(ChangeLevel player.id howMuch)` を追加しました。ここで `howMuch` はレベルを下げる場合は `-1` 、レベルを下げ場合には `1` になります。

> This page covers Elm 0.18

Playersコマンド

次に、プレーヤーをAPIで更新するコマンドを作成しましょう。

src/Players/Commands.elmに追加します：

```
import Json.Encode as Encode

...

saveUrl : PlayerId -> String
saveUrl playerId =
    "http://localhost:4000/players/" ++ playerId


saveRequest : Player -> Http.Request Player
saveRequest player =
    Http.request
        { body = memberEncoded player |> Http.jsonBody
        , expect = Http.expectJson memberDecoder
        , headers = []
        , method = "PATCH"
        , timeout = Nothing
        , url = saveUrl player.id
        , withCredentials = False
        }


save : Player -> Cmd Msg
save player =
    saveRequest player
        |> Http.send OnSave


memberEncoded : Player -> Encode.Value
memberEncoded player =
    let
        list =
            [ ( "id", Encode.string player.id )
            , ( "name", Encode.string player.name )
            , ( "level", Encode.int player.level )
            ]
    in
        list
            |> Encode.object
```

リクエストを保存する

```
saveRequest : Player -> Http.Request Player
saveRequest player =
    Http.request
        { body = memberEncoded player |> Http.jsonBody ❶
        , expect = Http.expectJson memberDecoder ❷
        , headers = []
        , method = "PATCH" ❸
        , timeout = Nothing
        , url = saveUrl player.id
        , withCredentials = False
        }
```

❶ここで、指定されたプレーヤーをエンコードし、エンコードされた値をJSON文字列に変換します ❷ここでは、レスポンスをパースする方法を指定します。この場合、返されたJSONをパースし、Elmの値に戻す必要があります。 ❸ `PATCH` はAPIがレコードを更新するときに期待するhttpメソッドです。

保存

```
save : Player -> Cmd Msg
save player =
    saveRequest player ❶
        |> Http.send OnSave ❷
```

ここでは、保存要求❶を作成し、 `Http.send` を使用して要求を送信するコマンドを生成します❷。 `Http.send` はメッセージコンストラクタ（この場合は `OnSave` ）をとります。リクエストが完了すると、Elmはリクエストに対するレスポンスとともに `OnSave` メッセージをトリガします。

> This page covers Elm 0.18

プレーヤーの更新

最後に、**src/Players/Update.elm**で `update` 関数で新しいメッセージを考慮する必要があります：

新しいimportを追加します：

```
import Players.Models exposing (Player, PlayerId)
import Players.Commands exposing (save)
```

更新コマンドを作成する

プレイヤーをAPIに保存するためのコマンドを作成するためのヘルパー関数を追加します。

```
changeLevelCommands : PlayerId -> Int -> List Player -> List (Cmd Msg)
changeLevelCommands playerId howMuch players =
    let
        cmdForPlayer existingPlayer =
            if existingPlayer.id == playerId then
                save { existingPlayer | level = existingPlayer.level + howMuch }
            else
                Cmd.none
    in
        List.map cmdForPlayer players
```

この関数は、 `ChangeLevel` メッセージを受け取ったときに呼び出されます。この関数は：

- プレーヤーIDとレベル差を受け取り、増減する
- 既存のプレイヤーのリストを受け取る
- リスト上の各プレイヤーのIDと、変更したいプレイヤーのIDを比較する
- idが私たちが望むものなら、そのプレイヤーのレベルを変更するコマンドを返す
- そして最後に実行するコマンドのリストを返す

プレイヤーを更新する

サーバーからの応答を受け取ったときにプレーヤーを更新するための別のヘルパー関数を追加する：

```
updatePlayer : Player -> List Player -> List Player
updatePlayer updatedPlayer players =
    let
        select existingPlayer =
            if existingPlayer.id == updatedPlayer.id then
                updatedPlayer
            else
                existingPlayer
    in
        List.map select players
```

この関数は、`SaveSuccess`を介してAPIから更新されたプレーヤーを受け取ったときに使用されます。この関数は：

- 更新されたプレーヤーと既存のプレイヤーのリストを取得します。
- 各プレイヤーのIDと更新されたプレーヤーとの比較
- IDが同じ場合は更新されたプレーヤーを返し、そうでない場合は既存のプレーヤーを返します

更新するための**case**式の枝を追加する

`update`関数に新しいcase式の枝を追加する：

```
update message players =
    case message of
        ...

        ChangeLevel id howMuch ->
            ( players, changeLevelCommands id howMuch players |> Cmd.batch )

        OnSave (Ok updatedPlayer) ->
            ( updatePlayer updatedPlayer players, Cmd.none )

        OnSave (Err error) ->
            ( players, Cmd.none )
```

- `ChangeLevel`では、上で定義したヘルパー関数`changeLevelCommands`を呼び出します。この関数は実行するコマンドのリストを返すので、`Cmd.batch`を使用してそれらを1つのコマンドにバッチ実行します。
- `OnSave (Ok updatedPlayer)`ではヘルパー関数`updatePlayer`を呼び出し、関連するプレイヤーをリストから更新します。

試してみよう

上記が、プレイヤーのレベルを変更するために必要なすべての設定です。試してみましょう。編集ビューに行き、-ボタンと+ボタンをクリックするとレベルが変更されます。ブラウザを更新しても変更はサーバー上に保存されています。

これまでのアプリケーションコードはhttps://github.com/sporto/elm-tutorial-app/tree/018-08-editになります。

おわりに

以上でこのチュートリアルは終了しますが、改善や機能に関するアイデアをお読みください。

私はこのチュートリアルを改善するためにあなたのフィードバックをお聞きしたいと思います。
https://github.com/sporto/elm-tutorialでissueを開いてください。

ありがとう！

さらなる改善

このアプリを、以下のように改善していくことができます。

プレーヤーの作成と削除

チュートリアルを短くするために実施していませんが、間違いなく重要な機能です。

プレーヤーの名前を変更する

Httpリクエストが失敗したときにエラーメッセージを表示する

プレイヤーの獲得や保存が失敗した場合、私たちは何もしません。ユーザーにエラーメッセージを表示するとよいでしょう。

さらに良いエラーメッセージ

エラーメッセージを表示するだけでなく、

- エラーや情報などの、さまざまな種類のフラッシュメッセージを表示する。
- 複数のフラッシュメッセージを同時に表示する
- メッセージを却下する能力がある
- 数秒後に自動的にメッセージを削除する

楽観的な更新

現時点では、すべての更新機能は悲観的です。つまり、サーバーからの応答が成功するまでモデルを変更しないということです。大きな改善案としては、楽観的な作成・更新・削除を追加することです。しかし、これを可能とするためには、より良いエラー処理が必要になります。

バリデーション

名前のないプレイヤーは避けるべきでしょう。プレーヤーの名前を検証し、空にできないようにするのが良いでしょう。

アイテムとボーナスを追加

プレーヤーがアイテムのリストを持てるようにすることができます。これらのアイテムは、機器、衣類、巻物、アクセサリー、たとえば「鋼鉄の剣」などです。それから、プレイヤーと特典の間には関連を持たせます。

それぞれの特典にはボーナスが付いています。プレイヤーは計算された強さに、レベルとボーナスを加えた合計を計算します。

このアプリケーションのより機能的なバージョンについては、https://github.com/sporto/elm-tutorial-app のマスターブランチを参照してください。

コンテキスト

典型的な `update` や `view` 関数は次のようになります：

```
view : Model -> Html Msg
view model =
  ...
```

あるいは

```
update : Msg -> Model -> (Model, Cmd Msg)
update message model =
  ...
```

このコンポーネントの `Model` だけを渡すということで非常にシンプルな話です。しかし、余分な情報が必要なとき、viewに追加的な情報を渡すのも問題ありません。例えば：

```
type alias Context =
  { model : Model
  , time : Time
  }

view : Context -> Html Msg
view context =
  ...
```

この関数は、親モデルで定義されているコンポーネントモデルと `time` を必要としています。

ポイントフリースタイル

ポイントフリーとは、体の中の1つ以上の引数を省略した関数を書くスタイルです。これを理解するために例を見てみましょう。

以下は、数値に10を加える関数です：

```
add10 : Int -> Int
add10 x =
    10 + x
```

接頭辞記法を使って `+` を使ってこれを書き直すことができます：

```
add10 : Int -> Int
add10 x =
    (+) 10 x
```

この場合の引数 `x` は厳密には必要ではないので、次のように書き直すことができます：

```
add10 : Int -> Int
add10 =
    (+) 10
```

`x'が` add10 `への入力引数と` + `への引数の両方としてどのように除去されたかに注意してください。` add10`は結果を計算するために整数と整数を必要とする関数のままです。このように引数を省略することを、ポイントフリースタイルと呼びます。

いくつかの例

```
select : Int -> List Int -> List Int
select num list =
    List.filter (\x -> x < num) list

select 4 [1, 2, 3, 4, 5] == [1, 2, 3]
```

は以下と同じです:

```
select : Int -> List Int -> List Int
select num =
    List.filter (\x -> x < num)

select 4 [1, 2, 3, 4, 5] == [1, 2, 3]
```

```
process : List Int -> List Int
process list =
    reverse list |> drop 3
```

は以下と同じです:

```
process : List Int -> List Int
process =
    reverse >> drop 3
```

トラブルシューティング

開発中に奇妙なコンパイラの動作が見つかった場合、`elm-stuff/build-artifacts`を削除してもう一度コンパイルすると、問題が修正される場合があります。