第 4 章

構造体

本章では、NanoPlanner の開発からいったん離れて「構造体」という Elixir の概念について学習します。『Elixir/Phoenix 初級①』第 14 章で学んだマップによく似ていますが、相違点もいくつかあります。本巻の重要なテーマであるデータベースへの問い合わせを学習するための基礎になりますので、しっかりと理解してください。

4.1 準備作業

この章で作成する Elixir プログラムを格納するディレクトリを作成してください。ディレクトリのパスは自由に決めて構いませんが、例として `~/elixir-primer/v02/ch04` を使うことにします。

```
$ mkdir -p ~/elixir-primer/v02/ch04
$ cd ~/elixir-primer/v02/ch04
```

以下、このディレクトリを「作業ディレクトリ」と呼びます。

この章で掲載するソースコードのパスは `~/elixir-primer/v02` からの相対パスで表示します。

続いて、ターミナルで次のコマンドを実行してください。

```
$ alias elixirc="elixirc --ignore-module-conflict"
```

このコマンドを実行しないと、同じ名前のモジュールを複数回コンパイルしたときに、次のような警告が出てしまいます。

```
warning: redefining module User (current version loaded from Elixir.User.beam)
  user.ex:1
```

上記の警告は「User モジュールを再定義しています」という意味です。プログラミングの学習をするときには、この警告が邪魔になります。

> alias コマンドの効果は、ターミナルを閉じるまで持続します。別のターミナルを開いたり、ターミナルを開き直したときは、上記のコマンドを改めて実行してください。

4.2 構造体の定義

構造体（struct）は、『Elixir/Phoenix 初級①』で学んだタプル、リスト、マップなどと同様に複数個の値を集合として扱うためのデータ型です。タプル、リストと異なり値に順序がなく、各値に割り当てられたユニークなキーで値が識別されます。構造体のキーは**フィールド**（field）とも呼ばれます。

このように、構造体はマップに類似しています。しかし、さまざまな点で構造体とマップは異なります。

具体例に基づいて構造体とマップを比べていきましょう。作業ディレクトリに、新規ファイル user.ex を次の内容で作成してください。

ch04/user.ex (New)

```
1  defmodule User do
2    defstruct [:name, :email]
3  end
```

これで、構造体 User が定義されました。マップと異なり、構造体には名前があります。その名前は、構造体を定義しているモジュールの名前と同じになります。

2 行目の defstruct は構造体を定義するためのマクロです。引数にはアトムのリストを取ります。これらのアトムが構造体のキー（フィールド）となります。

ターミナルで次のコマンドを実行してください。

```
$ elixirc user.ex
```

すると、作業ディレクトリに Elixir.User.beam というファイルが作られます。

マップと異なり構造体のキーは必ずアトムでなければなりません。したがって、defstruct マクロに文字列のリストを与えるとエラーになります。試しに、user.ex を次のように書き換えてください。

ch04/user.ex

```
1   defmodule User do
2 -   defstruct [:name, :email]
2 +   defstruct ["name", "email"]
3   end
```

このファイルをコンパイルすると、次のようなエラーが出ます。

```
== Compilation error on file user.ex ==
** (ArgumentError) struct field names must be atoms, got: "name"
...
```

user.ex を元に戻してから次に進んでください。

ch04/user.ex

```
1   defmodule User do
2 -   defstruct ["name", "email"]
2 +   defstruct [:name, :email]
3   end
```

4.3 構造体を作る

作業ディレクトリに、新規ファイル struct1.exs を次の内容で作成してください。

ch04/struct1.exs (New)

```
1  m = %{name: "foo", email: "foo@example.com"}
2  u = %User{name: "foo", email: "foo@example.com"}
3  IO.inspect m
4  IO.inspect u
```

これを Elixir スクリプトとして実行します。

```
$ elixir struct1.exs
```

すると、次のような結果が出力されます。

```
%{email: "foo@example.com", name: "foo"}
%User{email: "foo@example.com", name: "foo"}
```

ソースコードの 1〜2 行をご覧ください。

```
m = %{name: "foo", email: "foo@example.com"}
u = %User{name: "foo", email: "foo@example.com"}
```

1 行目でマップを作り、2 行目で構造体 User を作っています。表記法はよく似ています。構造体を作るときは、% 記号の直後に構造体の名前を挿入します。

マップではコロンの代わりに矢印記号（=>）を用いた記法も使えます。構造体ではどうでしょうか。struct1.exs を次のように書き換えてください。

ch04/struct1.exs

```
1 - m = %{name: "foo", email: "foo@example.com"}
1 + m = %{:name => "foo", :email => "foo@example.com"}
2 - u = %User{name: "foo", email: "foo@example.com"}
2 + u = %User{:name => "foo", :email => "foo@example.com"}
:
```

これを Elixir スクリプトとして実行すると、さきほどと同じ結果が出力されます。

さて、ここからはマップと構造体の異なる点を見ていきます。struct1.exs を次のように書き換えてください。

ch04/struct1.exs

```
1 - m = %{:name => "foo", :email => "foo@example.com"}
1 + m = %{:name => "foo", :email => "foo@example.com", :password => "xyz"}
2 - u = %User{:name => "foo", :email => "foo@example.com"}
2 + u = %User{:name => "foo", :email => "foo@example.com", :password => "xyz"}
:
```

これを Elixir スクリプトとして実行すると、次のようなエラーメッセージが出ます。

```
** (KeyError) key :password not found in: %User{email: "foo@example.com", >
name: "foo"}
    (stdlib) :maps.update(:password, "xyz", %User{email: "foo@example.com" >
, name: "foo"})
```

意味は「構造体 User には :password というキーが存在しない」というものです。マップと異なり、構造体の場合には限定されたキー（フィールド）しか使えません。user.ex の 2 行目をご覧ください。

```
  defstruct [:name, :email]
```

defstruct マクロで構造体を定義するときに指定したキーのリストに :password はありませんでした。だから、エラーが発生したのです。

続いて、struct1.exs を次のように書き換えてください。

ch04/struct1.exs

```
1 - m = %{:name => "foo", :email => "foo@example.com", :password => "xyz"}
1 + m = %{"name" => "foo", "email" => "foo@example.com"}
2 - u = %User{:name => "foo", :email => "foo@example.com", :password => "xyz"}
2 + u = %User{"name" => "foo", "email" => "foo@example.com"}
:
```

これを Elixir スクリプトとして実行すると、次のようなエラーメッセージが出ます。

```
** (KeyError) key "name" not found in: %User{email: nil, name: nil}
    (stdlib) :maps.update("name", "foo", %User{email: nil, name: nil})
```

アトム :name と文字列 "name" はキーとして区別されます。この点は、マップと同じです。

> 構造体ではそもそも文字列のキーは許されていないので、このエラーメッセージは少し奇妙な印象を与えます。実は、Elixir の構造体とマップはいずれも Erlang のマップの拡張として実装されており、さまざまな状況下で同じものとして扱われます。詳しくは、最終節「マップと構造体の関係」を参照してください。

4.4 構造体から値を取り出す

struct1.exs を次のように書き換えてください。

ch04/struct1.exs (New)

```
1 - m = %{"name" => "foo", "email" => "foo@example.com"}
1 + m = %{name: "foo", email: "foo@example.com"}
2 - u = %User{"name" => "foo", "email" => "foo@example.com"}
2 + u = %User{name: "foo", email: "foo@example.com"}
3 - IO.inspect m
3 + IO.inspect m.email
4 - IO.inspect u
4 + IO.inspect u.email
```

これを Elixir スクリプトとして実行すると、次のような結果が出力されます。

```
"foo@example.com"
"foo@example.com"
```

ソースコードの 3〜4 行をご覧ください。

```
IO.inspect m.email
IO.inspect u.email
```

マップ m と構造体 u からフィールド :email に対する値を取り出しています。

マップでも構造体でも、ドット記号とコロンなしのアトムを指定すれば値を取れます。

さて、マップの場合は角括弧（[ ]）の中にキーを指定する方法でも値を取得できました。構造体ではどうでしょうか。struct1.exs を次のように書き換えてください。

ch04/struct1.exs

```
  :
3 - IO.inspect m.email
3 + IO.inspect m[:email]
4 - IO.inspect u.email
4 + IO.inspect u[:email]
```

これを Elixir スクリプトとして実行すると、次のようなエラーメッセージが出ます。

```
"foo@example.com"
** (UndefinedFunctionError) function User.fetch/2 is undefined (User does >
not implement the Access behaviour)
    User.fetch(%User{email: "foo@example.com", name: "foo"}, :email)
...
```

3 行目の IO.inspect m[:email] は "foo@example.com" を出力しています。しかし、4 行目では「User.fetch/2 関数が存在しない」という意味のエラーメッセージが出ています。構造体では角括弧（[ ]）を用いて値を取得できません。マップと構造体の重要な相違点です。

> 角括弧（[ ]）はマクロであり、実行前に User.fetch/2 関数を用いた式に変換されます。そのため「User.fetch/2 関数が存在しない」というエラーメッセージが出ます。

では、さきほどの変更を元に戻してから、次に進みましょう。

ch04/struct1.exs

```
  :
3 - IO.inspect m[:email]
3 + IO.inspect m.email
```

```
4 - IO.inspect u[:email]
4 + IO.inspect u.email
```

4.5 構造体の値を置き換える

Elixir ではすべての値がイミュータブル（不変）です。構造体もそうです。しかし、構造体から別の構造体を作り出せば、実質的に構造体の値を置き換えることができます。struct1.exs を次のように変更してください。

ch04/struct1.exs

```
1   m = %{name: "foo", email: "foo@example.com"}
2 + m = %{m | email: "bar@example.com"}
3   u = %User{name: "foo", email: "foo@example.com"}
4 + u = %User{u | email: "bar@example.com"}
:
```

さらに、同ファイルを次のように変更してください。

ch04/struct1.exs

```
:
5 - IO.inspect m.email
5 + IO.inspect m
6 - IO.inspect u.email
6 + IO.inspect u
```

実行結果は次のとおりです。

```
%{email: "bar@example.com", name: "foo"}
%User{email: "bar@example.com", name: "foo"}
```

2 行目のようにパイプ記号（|）を用いてマップの値を置き換える方法については、『初級①』第 14 章で学習しました。構造体の値を置き換えるときにも、ほぼ同じような書き方ができるというわけです。

マップの場合には、関数 Map.merge/2 を用いて値を置き換えることもできました。構造体でもできるでしょうか。struct1.exs を次のように書き換えてください。

ch04/struct1.exs

```
1   m = %{name: "foo", email: "foo@example.com"}
2 - m = %{m | email: "bar@example.com"}
2 + m = Map.merge(m, %{email: "bar@example.com"})
3   u = %User{name: "foo", email: "foo@example.com"}
4 - u = %User{u | email: "bar@example.com"}
4 + u = Map.merge(u, %{email: "bar@example.com"})
:
```

さきほどと同じ実行結果になります。

```
%{email: "bar@example.com", name: "foo"}
%User{email: "bar@example.com", name: "foo"}
```

このように、構造体に対しても関数 Map.merge/2 が使えます。ただし、第 2 引数には構造体ではなくマップを指定します。

> 関数 Map.merge/2 の第 2 引数に構造体を指定してもエラーにはなりません。しかし、その結果は意外なものになります。詳しくは、最終節「マップと構造体の関係」をご覧ください。

4.6 デフォルト値

作業ディレクトリに、新規ファイル structs2.exs を次の内容で作成してください。

ch04/struct2.exs (New)

```
1   u = %User{email: "foo@example.com"}
2   IO.inspect u.name
3   IO.inspect u.email
```

これを Elixir スクリプトとして実行すると、次のような結果が出力されます。

```
nil
"foo@example.com"
```

ここで、user.ex を次のように書き換えます。

ch04/user.ex

```
1   defmodule User do
2 -   defstruct [:name, :email]
2 +   defstruct [{:name, "No name"}, :email]
3   end
```

そして、user.ex をコンパイルしてから、structs2.exs を Elixir スクリプトとして実行してください。すると、出力結果が次のように変化します。

```
"No name"
"foo@example.com"
```

user.ex の変更箇所（2 行目）をご覧ください。

```
  defstruct [{:name, "No name"}, :email]
```

変更前のコードでは、defstruct マクロの引数はアトムのリストでした。リストの 1 番目の要素をアトムからタプルに変更しました。そのタプルはアトム :name と文字列 "No name" により構成されています。このように書くことによって、フィールド :name に対して**デフォルト値**を指定できます。

続いて、user.ex を次のように書き換えてください。

ch04/user.ex

```
1   defmodule User do
2 -   defstruct [{:name, "No name"}, :email]
2 +   defstruct name: "No name", email: nil
3   end
```

user.ex をコンパイルしてから、structs2.exs を Elixir スクリプトとして実行すると、出力結果は前回から変化しません。このように defstruct マクロの引数としてキーワードリストを指定すれば、すべてのフィールドに対してデフォルト値を指定できます。デフォルト値として nil を指定することと、デフォルト値を指定しないことは同値です。

ちなみに、user.ex の 2 行目は次のようにも書けます。

```
defstruct [{:name, "No name"}, {:email, nil}]
```

『初級①』第 14 章で説明したように、キーワードリストは次の 3 条件をすべて満たす特別なリストです。

1. リストのすべての要素はタプルである。
2. それらのタプルの要素数はすべて 2 である。
3. それらのタプルの第 1 要素は常にアトムである。

一般的には、defstruct マクロの引数はリストであり、その要素はアトムまたはタプルです。要素がタプルである場合は、アトムとデフォルト値を組み合わせたものになります。そして、リストのすべての要素がタプルであるとき、それはキーワードリストになり、コロン記号（:）を用いた簡易表記が使えるようになります。

4.7 マップと構造体の関係

構造体は __struct__ という名前の特別なフィールドを持っており、このフィールドに構造体を定義したモジュールを保持しています。ターミナルで次のコマンドを実行してください。

```
$ elixir -e "u = %User{}; IO.inspect u.__struct__"
```

ターミナルには User という結果が出力されます。"User" のように引用符で囲まれていないので、__struct__ フィールドの値は文字列ではなく、User モジュールを指すアトムであることが分かります。

> モジュールを指すアトム（先頭がアルファベットの大文字で始まる）の場合、先頭にコロン記号（:）が不要です。『初級①』第 4 章を参照してください。

また、ある値がマップであるかどうかを調べる関数 Kernel.is_map/1 は、引数に構造体が指定されたときにも true を返します。ターミナルで次のコマンドを実行してください。ターミナルには true という結果が出力されるはずです。

```
$ elixir -e "u = %User{}; IO.inspect is_map(u)"
```

実は、Elixir のマップと構造体は、いずれも Erlang のマップの拡張として実装されています。つまり、どちらも内部的には Erlang のマップであり、Elixir はフィールド `__struct__` の有無で両者を区別しているのです。

では、`__struct__` という名前のキーを持つマップを作ると、それは構造体になるでしょうか。作業ディレクトリに、新規ファイル `struct3.exs` を次のような内容で作成してください。

ch04/struct3.exs

```
1  u = %{__struct__: User, name: "foo", email: "foo@example.com"}
2  IO.inspect u
```

これを Elixir スクリプトとして実行すると、次のような結果が出力されます。

```
%User{email: "foo@example.com", name: "foo"}
```

構造体として認識されていますね。では、`struct3.exs` を次のように書き換えてください。

ch04/struct3.exs

```
1    u = %{__struct__: User, name: "foo", email: "foo@example.com"}
2 -  IO.inspect u
2 +  IO.inspect u[:name]
```

これを Elixir スクリプトとして実行すると、次のようなエラーメッセージが出ます。

```
** (UndefinedFunctionError) function User.fetch/2 is undefined (User does >
not implement the Access behaviour)
```

つまり、`__struct__` という名前のキーを持つマップは構造体となり、普通のマップとしての性質の一部（角括弧記号で値を取得できるなど）を失うというわけです。

> Elixir の公式ウェブサイト（http://elixir-lang.org/getting-started/structs.html）では、構造体のことを「裸のマップ（bare maps）」と呼んでいます。ここで言う「マップ」は、Erlang のマップを指しています。Elixir がマップのために付加した性質を持たないという意味で「裸の（bare）」という表現を用いています。

さらに構造体への理解を深めるため、作業ディレクトリに新規ファイル struct4.exs を次のような内容で作成してください。

ch04/struct4.exs

```
1  u = %User{name: "foo", email: "foo@example.com"}
2  u = Map.merge(u, %{name: "bar"})
3  IO.inspect u
```

これを Elixir スクリプトとして実行すると、次のような結果が出力されます。

```
%User{email: "foo@example.com", name: "bar"}
```

予想通りの結果と言えます。では、struct4.exs を次のように変更するとどうなるでしょうか。

ch04/struct4.exs

```
1  u = %User{name: "foo", email: "foo@example.com"}
2 - u = Map.merge(u, %{name: "bar"})
2 + u = Map.merge(u, %User{name: "bar"})
3  IO.inspect u
```

出力結果は次のように変化します。

```
%User{email: nil, name: "bar"}
```

意外な結果です。なぜ email フィールドの値が nil になってしまうのでしょうか。実は、%User{name: "bar"} という式は次の式と同値です。

```
%{__struct__: User, name: "bar", email: nil}
```

構造体 u とこのマップをマージするので、email フィールドの値が上書きされ

てしまうのです。

さらに、struct4.exs を次のように書き換えてください。

ch04/struct4.exs

```
1 - u = %User{name: "foo", email: "foo@example.com"}
1 + u = %{name: "foo", email: "foo@example.com"}
2   u = Map.merge(u, %User{name: "bar"})
3   IO.inspect u
```

出力結果は変化しません。

```
%User{email: nil, name: "bar"}
```

1 行目で作った変数 u には普通のマップが格納されています。そして u と構造体 %User{name: "bar"} をマージすると、__struct__ を含むすべてのキーについて値が上書きされるので、2 行目と 3 行目で使われている変数 u には構造体がセットされることになるのです。

以上のように、マップと構造体、あるいは構造体と構造体をマージすることは可能ですが、あまり実用的ではありません。関数 Map.merge/2 の第 2 引数には普通のマップを指定する、と覚えてください。