第 5 章

クエリの絞り込み

本章では、Ecto.Query モジュールのマクロ where/3 を用いてクエリ構造体に絞り込み条件を設定する方法について学びます。

5.1 準備作業

実験スクリプト filter.exs の作成と実行

前巻の第 7 章でチェンジセットについて学ぶ際に、ディレクトリ priv の下にサブディレクトリ exp を作成しました。“exp”は“experimental”の略で、ここに「実験」のためのスクリプトを置きます。このディレクトリが存在しない場合は、次のコマンドで作成してください。

```
$ mkdir -p priv/exp
```

そして、priv/exp ディレクトリの下に新規ファイル filter.exs を作成し、次の内容を書き込んでください。

priv/exp/filter.exs (New)

```
1  import Ecto.Query
2  alias NanoPlanner.Repo
3  alias NanoPlanner.Schedule.PlanItem
4
```

```
5  items =
6    PlanItem
7    |> order_by(asc: :id)
8    |> Repo.all()
9    |> Enum.map(& &1.name)
10
11 IO.inspect(items)
```

そして、ターミナルで次のコマンドを実行します。

```
$ mix run priv/exp/filter.exs
```

すると、図 5.1 のような結果が出力されます。

```
[debug] QUERY OK source="plan_items" db=3.2ms decode=6.8ms
SELECT p0."id", p0."name", p0."description", p0."starts_at", p0."ends_at", p0."i
nserted_at", p0."updated_at" FROM "plan_items" AS p0 ORDER BY p0."id" []
["読書", "買い物", "帰省"]
```

図 5.1　filter.exs の最初の実行結果

1 行目の [debug] で始まる出力は、文字通り「デバッグ」のための情報です。クエリ対象のテーブル名およびクエリの実行に要した時間が書いてあります。2 行目から 3 行目に出力されているのは今回のクエリで発行された SQL 文です。

最終行には次のように出力されています。

```
["読書", "買い物", "帰省"]
```

これは、filter.exs の末尾にある関数 IO.inspect/2 によって出力されています。

> 次節以降では、実行結果を載せる際に原則としてデバッグのための情報と SQL 文を省略します。

実験スクリプト filter.exs の解説

9 行目では『初級②』第 9 章で学んだキャプチャ演算子 & を用いて無名関数を作り、関数 Enum.map/2 の第 2 引数に指定しています。キャプチャ演算子を使わずに書けば、次のようになります。

```
|> Enum.map(fn(i) -> i.name end)
```

5.2 マクロ where/3 の第 1 の使い方

クエリの結果をある条件に合致するものだけに絞り込みたいとき、マクロ where/3 を利用します。

このマクロを呼び出すには 2 通りの方法があります。第 1 の方法では、キーワードリストを用います。第 2 の方法では、後述する「クエリ・バインディングス（query bindings）」と用います。

まずは、単純でわかりやすい第 1 の方法を学びましょう。

キーワードリストによる絞り込み

filter.exs を次のように書き換えてください。

priv/exp/filter.exs

```
  :
5  items =
6    PlanItem
7 +  |> where(name: "読書")
8    |> order_by(asc: :id)
9    |> Repo.all()
10   |> Enum.map(&  &1.name)
11
12 IO.inspect(items)
```

再び、ターミナルで次のコマンドを実行します。

```
$ mix run priv/exp/filter.exs
```

すると、次のような結果が出力されます。

```
["読書"]
```

ここでは、マクロ where/3 にキーワードリスト [name: "読書"] を渡すことにより、plan_items テーブルから name 列の値が「読書」に等しいレコードだけを取り出すクエリ構造体を作っています。

キーワードリストの要素を増やす

キーワードリストの要素を増やすことによりレコードをさらに絞り込むことができます。例えば、7 行目を次のように書き換えてみましょう。

priv/exp/filter.exs

```
  :
- |> where(name: "読書")
+ |> where(name: "読書", description: "")
  :
```

これは「name 列の値が「読書」に等しく、かつ description 列の値が空文字に等しい」という条件を表します。

変数と式を使用するときの注意点

マクロ where/3 には、重要な利用上の注意点があります。それは、引数の中で変数や式を使用するときには、前にキャレット記号（^）を置かなければならないということです。

試しに、filter.exs を次のように書き換えてみてください。

priv/exp/filter.exs

```
1 import Ecto.Query
2 alias NanoPlanner.Repo
3 alias NanoPlanner.Schedule.PlanItem
```

```
 4 +
 5 + name = "読書"
 6
 7   items =
 8     PlanItem
 7 -   |> where(name: "読書", description: "")
 9 +   |> where(name: name)
10     |> order_by(asc: :id)
11     |> Repo.all()
12     |> Enum.map(& &1.name)
13
14   IO.inspect(items)
```

そして、このスクリプトを実行すると次のようなエラーメッセージが出力されます。

```
** (Ecto.Query.CompileError) variable `name` is not a valid query >
expression. IF you wanted to inject a variable, you have to >
explicitly interpolate it with ^. If you wanted to refer to a field, >
use the source.field syntax
...
```

エラーメッセージを日本語に翻訳すると、次のようになります。

> 変数 name は有効なクエリ式ではありません。もし変数を挿入したかったのであれば、^ を用いて明示的に式展開する必要があります。もしあるフィールドを参照したかったのであれば、source.field 構文を用いてください。

初心者にはわかりにくいエラーメッセージであり、完全に意味を理解する必要はありません。要するに、変数 name の前にキャレット記号を書き忘れたためにエラーが発生しています。実際、filter.exs を次のように書き換えればスクリプトは正常に動作します。

priv/exp/filter.exs

```
  :
 9 -   |> where(name: name)
 9 +   |> where(name: ^name)
```

```
  :
```

マクロ where/3 への引数の中で式を使いたい場合、式全体を括弧で囲んだ上でキャレット記号を前に置きます。例えば、filter.exs を次のように書き換えてみましょう。

priv/exp/filter.exs

```
  :
  9 -   |> where(name: ^name)
  9 +   |> where(name: ^(name <> name))
  :
```

このスクリプトは正常に動作します。ただし、検索結果は空となります。式 name <> name は変数 name の参照する文字列を 2 度繰り返した文字列（「読書読書」）を返しますので、条件に合致するレコードは見つかりません。

では、変更を元に戻してから次の節に進みましょう。

priv/exp/filter.exs

```
  :
  9 -   |> where(name: ^(name <> name))
  9 +   |> where(name: ^name)
  :
```

5.3 マクロ where/3 の第 2 の使い方

クエリ・バインディングスとクエリ式

では、マクロ where/3 の第 2 の呼び出し方を学んでいきましょう。まずは、実際の使用例を見るために、filter.exs を次のように書き換えてください。スクリプトの結果は書き換えの前後で変化しません。

priv/exp/filter.exs

```
  :
  9 -   |> where(name: ^name)
```

```
  9 +   |> where([i], i.name == ^name)
  :
```

2 つの引数が与えられています。前者 [i] をクエリ・バインディングス（query bindings）と呼び、後者 i.name == ^name をクエリ式（query expression）と呼びます。

上記の例ではクエリ・バインディングスを構成する要素はただ 1 つです。この i は Elixir の変数のように見えますが、そうではありません。データベーステーブル plan_items を指す仮の「名前」です。なお、この「名前」は Elixir のコンパイラが変数とみなすものであれば、どんなものでも構いません。i の代わりに p や t0 を使えます。しかし、大文字の K、数字の 3、記号の$などはクエリ・バインディングの名前として適しません。

同様に、クエリ式の中に現れる i.name も Elixir のマップや構造体から値を取り出しているわけではなく、テーブル plan_items の name 列を指す「名前」に過ぎません。

さらに言えば、クエリ式で使われている演算子 == も、Elixir の比較演算子 == と同じ形をしていて意味も似ていますが、まったく別物です。SQL の比較演算子 = の代わりをしているだけです。

■コラム: ドメイン固有言語（DSL）

マクロ where/3 は、通常の Elixir とはまったく異なる考え方で与えられた引数を解釈します。このマクロの世界では、Elixir に似た別の「言語」が動いているのです。

限定された範囲でのみ使える「言語」のことを、ソフトウェア用語でドメイン固有言語（domain-specific languag, DSL）と呼びます。

クエリ式で使える演算子と関数

マクロ where/3 のクエリ式では、表 5.1 のような演算子と関数を用いることができます。また、演算子の優先順位を明示するため括弧を使うこともできます。

表 5.1　クエリ式で使える主な演算子・関数

クエリ式の演算子・関数	対応する SQL の演算子・関数	意味
==	=	等しい
!=	<>	等しくない
>	>	より大きい
<	<	より小さい
>=	>=	以上
<=	<=	以下
not	NOT	否定
and	AND	かつ
or	OR	または
is_nil	IS NULL	NULL である
in	IN	リストに含まれる
like	LIKE	パターンに合致する

クエリ式で使える演算子・関数についてさらに詳しく知りたい方は、https://hexdocs.pm/ecto/Ecto.Query.API.html を参照してください。

では、実際にクエリ式を書いてみましょう。filter.exs を次のように書き換えるとどのような結果になるでしょうか。

priv/exp/filter.exs

```
  :
  9 -   |> where([i], i.name == ^name)
  9 +   |> where([i], i.name != ^name)
  :
```

すると、このスクリプトの実行結果は次のように変化します。

```
["買い物", "帰省"]
```

これは簡単ですね。plan_items テーブルから列 name の値が「読書」でないという条件でレコードを絞り込んでいます。

さまざまな演算子・関数の使用例

では、表 5.1 にある演算子・関数を使いながら、クエリ式の書き方に慣れていきましょう。

演算子 in/2

filter.exs を次のように書き換えてください。

priv/exp/filter.exs

```
  :
- name = "読書"
+ names = ~W(読書 買い物)

items =
  PlanItem
- |> where([i], i.nane == ^name)
+ |> where([i], i.name in ^names)
  |> order_by(asc: :id)
  |> Repo.all()
  |> Enum.map(& &1.name)

IO.inspect(items)
```

すると、このスクリプトの実行結果は次のように変化します。

```
["読書", "買い物"]
```

演算子 in/2 は、左辺に指定された列の値が右辺に指定されたリストの中にある、という条件指定のための使います。

もし「リストの中にない」という条件指定をしたければ、否定演算子 not を用いて次のように書いてください。

```
|> where([i], i.name not in ^names)
```

関数 like/2

filter.exs を次のように書き換えてください。

priv/exp/filter.exs

```
  :
- names = ~W(読書 買い物)
+ pattern = "%買う%"

items =
  PlanItem
- |> where([i], i.name in ^names)
+ |> where([i], like(i.description, ^pattern))
  |> order_by(asc: :id)
  |> Repo.all()
  |> Enum.map(& &1.name)

IO.inspect(items)
```

すると、このスクリプトの実行結果は次のように変化します。

```
["買い物", "帰省"]
```

関数 like/2 は、第 1 引数に指定された列の値が第 2 引数に指定されたパターンに合致する、という条件指定のための使います。パターンの書き方は、SQL でのルールに従います。記号 % は長さ 0 個以上の任意の文字列に、記号 ? は任意の文字列 1 個にマッチします。

もし「パターンに合致しない」という条件指定をしたければ、否定演算子 not を用いて次のように書いてください。

```
|> where([i], not like(i.description, ^pattern))
```

SQL の LIKE 演算子が具体的にどのように働くかは、データベース管理システムにより若干の揺れがあります。例えば、PostgreSQL の LIKE 演算子はアルファベットの大文字と小文字を区別しますが、他の多くのデータベース管理システムは区別しません。PostgreSQL でアルファベットの大文字と小文字を区別しない条件指定を行いたければ、関数 like/2 の代わりに関数 ilike/2 を用いてください。

比較演算子 and、or

本章の締めくくりとして、比較演算子 and と or を組み合わせた少し複雑なクエリ式を書いてみましょう。

priv/exp/filter.exs

```
  :
- pattern = "%買う%"
+ name = "読書"
+ time0 = Timex.now("Asia/Tokyo") |> Timex.beginning_of_day()
+ time1 = time0 |> Timex.shift(days: 1)
+ time2 = time0 |> Timex.shift(days: 7)

items =
  PlanItem
- |> where([i], i.name != ^name)
+ |> where(
+   [i],
+   i.name != ^name or (i.starts_at > ^time1 and i.starts_at < ^time2)
+ )
  |> order_by(asc: :id)
  :
```

SQL では演算子 OR よりも AND の方が優先順位が高いので、括弧を使わなくてもこのクエリ式を書くことができます。しかし、括弧の使用例を示すため、そして読みやすさのために加えています。

シードデータを投入し直してからスクリプトを実行してください。

```
$ mix ecto.reset
$ mix run priv/exp/filter.exs
```

結果は次の通りです。

```
["読書", "買い物", "帰省"]
```

plan_items テーブルから次の 2 つの条件のどちらかが成立するレコードを取り出しています。

1. 列 name の値が「読書」ではない。
2. 列 starts_at の値が明日の午前 0 時以降、かつ 7 日後の午前 0 時より前である。