20分でわかる！正規表現のキホン

Nextbeat Tech Bar： 楽しい正規表現

<script type="module"> import mermaid from 'https://cdn.jsdelivr.net/npm/mermaid@11.4.1/dist/mermaid.esm.min.mjs'; mermaid.initialize({ startOnLoad: true }); </script>

2025/07/18

@kmizu

自己紹介

- 名前: **kmizu**
- Twitter: @kmizu
- 株式会社ネクストビートでソフトウェアエンジニアをしています
- プログラミ言語、構文解析、正規表現などが好き

今日は、皆さんが普段使っている正規表現の「キホン」を、理論的な側面から楽しく解説します！

みなさん、正規表現つかってますか？

- エディタでの検索・置換
- `grep` コマンドでのログ検索
- フォームのバリデーション (メールアドレス、電話番号など)
- プログラムの字句解析

など、プログラミングの様々な場面で活躍する強力なツールですよね。

でも、その「理論」はご存知ですか？

「ググった正規表現をコピペして使っている」
「なんとなく動くけど、なぜ動くのかはよくわからない」
「複雑な正規表現は読み解けない…」

という方も多いのではないでしょうか？

大丈夫です！
今日の発表を聞けば、正規表現の基本的な仕組みがわかり、もっと正規表現と仲良くなれます。

本日のアジェンダ

1. **正規表現の「キホン」のキホン (基本3構造)**
    - 正規表現を構成する、たった3つの要素とは？
2. **いろんな記法を分解してみよう**
    - `?` や `+` などの記法は基本構造の組み合わせでできている
3. **正規表現とオートマトンの関係**
    - 正規表現がどうやって文字列にマッチするのか
4. **正規表現の限界**
    - 正規表現に「できない」こと

1. 正規表現の「キホン」のキホン

(基本3構造)

正規表現を支える3つの基本構造

実は、どんなに複雑に見える正規表現（※）も、たった3つの基本的な構造の組み合わせでできています。

1. **連接** (Concatenation)
2. **選択** (Alternation)
3. **反復** (Repetition / Kleene Star)

これらを順番に見ていきましょう。

※後方参照などの一部の高度な拡張機能を除きます。

1. 連接 (Concatenation)

文字を順番に並べる、最もシンプルな構造です。

例: `abc`

これは、 `"a"` の次に `"b"` が来て、その次に `"c"` が来る文字列 `"abc"` にマッチします。

- `abcde` -> マッチする (`abc`)
- `ab` -> マッチしない
- `acb` -> マッチしない

2. 選択 (Alternation)

`|` (パイプ) を使って「または」を表現します。

例: `cat|dog`

これは、 `"cat"` または `"dog"` という文字列にマッチします。

- `I have a cat.` -> マッチする (`cat`)
- `I have a dog.` -> マッチする (`dog`)
- `I have a bird.` -> マッチしない

3. 反復 (Repetition / Kleene Star)

`*` (アスタリスク) を使って「0回以上の繰り返し」を表現します。
「クリーネ閉包」とも呼ばれます。

例: `ab*c`

これは、`"a"` の次に `"b"` が0回以上繰り返され、最後に `"c"` が来る文字列にマッチします。

- `ac` -> マッチする (bが0回)
- `abc` -> マッチする (bが1回)
- `abbbbc` -> マッチする (bが4回)
- `axc` -> マッチしない

演算の優先順位とグルーピング `()`

`*` (反復) は `|` (選択) や連接よりも優先されます。
計算式と同じように、 `()` を使うことで優先順位を変えることができます。

- `ab|c` -> `"ab"` または `"c"`
- `a(b|c)` -> `"ab"` または `"ac"`

まとめ：基本3構造

- **連接** (`ab`)
- **選択** (`a|b`)
- **反復** (`a*`)

この3つの組み合わせとグルーピング `()` だけで、理論的な「正規表現」はすべて表現可能です。

2. いろんな記法を分解してみよう

よく見るあの記法は？

`?`, `+`, `.`, `[a-z]` など、便利な記法がたくさんありますよね。

これらは、先ほどの基本3構造を使って表現できる、いわば**糖衣構文 (Syntactic Sugar)** のようなものです。

いくつか例を見てみましょう。

`?` (0回か1回の繰り返し)

例: `colou?r`

`u` が0回または1回出現する `"color"` と `"colour"` の両方にマッチします。

これは **選択** を使って次のように書けます。

`colo(u|)r`
(`u` があるか、何もない(`|` の右側)かを選択)

`+` (1回以上の繰り返し)

例: `go+gle`

`o` が1回以上出現する `"gogle"`, `"google"`, `"gooogle"` などにマッチします。

これは **連接** と **反復** を使って次のように書けます。

`goog*le`
(最初の `o` は必須で、その後に `o` が0回以上続く)

`.` (任意の一文字)

改行文字を除く、任意の一文字にマッチします。

これは、対象となる文字セット**すべて**の **選択** と同じ意味です。

```
(a|b|c|...|z|A|...|Z|0|...|9|...|!)
```

(対象文字セットがASCIIの場合のイメージ)

`[abc]` (文字クラス)

角括弧内のいずれかの一文字にマッチします。

例: `gr[ae]y`

`"gray"` または `"grey"` にマッチします。

これも **選択** と同じです。

`gr(a|e)y`

まとめ：便利な記法は基本構造の組み合わせ

今見てきたように、普段私たちが便利に使っている記法の多くは、

- **連接**
- **選択**
- **反復**

の3つの基本構造に分解することができます。

正規表現のコアは、実はとてもシンプルなのです。

3. 正規表現とオートマトンの関係

どうやってマッチしてるの？

コンピュータは、正規表現という「文字列」をどのように解釈して、マッチングを行っているのでしょうか？

その答えが **オートマトン (Automaton)** です。

正規表現は、内部的に「オートマトン」と呼ばれる一種の仮想的な機械に変換されてから実行されます。

オートマトンとは

- **状態 (State)** と **遷移 (Transition)** を持つ計算モデルです。
- 入力文字列を1文字ずつ読み込み、現在の状態と読み込んだ文字に応じて、次の状態へ遷移します。
- 文字列をすべて読み終えたときに **受理状態 (Accept State)** と呼ばれる特別な状態にいれば、「マッチ成功」となります。

正規表現からNFAへ

正規表現は、まず **NFA (Non-deterministic Finite Automaton / 非決定性有限オートマトン)** と呼ばれるオートマトンに変換されます。

- **非決定性**: 1つの状態から、同じ入力文字で複数の状態に遷移できたり、入力文字なし（ε/イプシロン）で遷移できたりする。

基本3構造が、それぞれどのようにNFAに対応するか見てみましょう。

正規表現 → NFA (連接: `ab`)

`a` にマッチするオートマトンの次に `b` にマッチするオートマトンを繋げます。

```
graph LR
    S((Start)) -- a --> S1(( )) -- b --> F((Accept))
```

正規表現 → NFA (選択: `a|b`)

開始状態から、a のルートと b のルートに分岐します。

```
graph TD
    S((Start)) --> A
    S((Start)) --> B
    A -- a --> E((Accept))
    B -- b --> E((Accept))
```

正規表現 → NFA (反復: `a*`)

- `a` の遷移でループし、ループに入る前でも後でも受理状態に行けるようにします。
  （ε は空文字を表し、文字を消費せずに遷移します）

開始状態から、a のルートに戻る遷移を作ります。

```
graph LR
    S((Start)) -- ε --> M
    M -- a --> M
    S -- ε --> F((Accept))
    M -- ε --> F
```

NFAからDFAへ

NFAはそのまま実行することもできますが、DFA (Deterministic Finite Automaton / 決定性有限オートマトン) に変換されることもあります。

- 決定性: 遷移先が1つに決まる。曖昧さがない。
- 高速: DFAはNFAよりも高速にマッチングできます。

```
graph LR
    subgraph NFA for (a|b)*a
        style NFA fill:#f9f,stroke:#333,stroke-width:2px
        n0(( )) -- ε --> n1
        n1 -- a --> n2
        n2 -- ε --> n1
        n1 -- b --> n3
        n3 -- ε --> n1
        n1 -- a --> n4((Accept))
    end

    subgraph DFA for (a|b)*a
```

NFA VS. DFA

特徴	NFA	DFA
速度	遅いことがある（バックトラック）	常に高速（入力長に比例）
状態数	少ない	指数的に増加することがある
構築	正規表現から簡単	少し複雑（部分集合構成法）
備考	後方参照などの拡張機能はNFAベースの実装が多い	

多くの正規表現エンジンは、両方の長所を組み合わせたハイブリッドな方式を採用しています。

4. 正規表現の限界

正規表現は万能ではない

正規表現（有限オートマトン）は「有限」のメモリ（状態）しか持てません。
そのため、原理的にマッチングできないパターンが存在します。

言い換えると、正規表現は「数を数える」のが苦手です。

できないことの例：カッコの対応

例: `(((())))` のような、任意にネストしたカッコの対応

```
^\(.*\)$
```

これだと `())(()` のような不正な文字列にもマッチしてしまいます。

なぜできないのか？
-> 開きカッコ (の数を正確に記憶し、それと同じ数の閉じカッコ) があるかを確認する必要があるため。オートマトンの「有限の状態」では、無限の可能性のある数を記憶しきれません。
このようなパターンは、より強力な「プッシュダウン・オートマトン」やパーサが必要になります。

「非正規」な拡張機能に注意

最近の正規表現エンジン（PCRE, Ruby, Python, Java, JavaScript, .NETなど）は、本来の正規表現の能力を超える強力な拡張機能を備えています。

- 後方参照: `(\w+)\s+\1` -> `hello hello` にマッチ
- 再帰: `\(([^()]*|(?R))*\)` -> 入れ子になったカッコにマッチ (エンジンによる)

これらは非常に便利ですが「もはや"正規"表現ではない」ということを知っておくと、挙動に悩んだときの助けになります。

本日のまとめ

- 正規表現のキホンは**「連接」「選択」「反復」**の3つ。
- `?` や `+` などの便利な記法は、これら基本構造の組み合わせ。
- 正規表現は**オートマトン**という仮想機械に変換されて実行される。
- オートマトンには**NFA**と**DFA**があり、それぞれ特徴がある。
- 正規表現には理論的な**限界**があり、ネスト構造などを正しく扱うのは苦手。

正規表現の裏側にある理論を少し知ることで、より深く、そして楽しく正規表現を使いこなせるようになるはずです！

ご清聴ありがとうございました！

質疑応答