記事
Toshihiko Minamoto · 2020年11月3日 123m read

Caché における正規表現の使用について

1.本記事の内容

Caché パターンマッチングと同様に、Caché では正規表現を使ってテキストデータのパターンを特定することができますが、後者の場合はより高い表現力を利用できます。 本記事では正規表現を簡単に紹介し、Caché での活用方法について解説します。 本記事の情報は、主に Jeffrey Friedl 氏著作の「Mastering Regular Expressions (詳説 正規表現)」に加え、もちろん Caché のオンラインドキュメンテーションなど、様々なリソースを基に提供しています。
本記事は正規表現のあらゆる可能性や詳細について解説することを意図したものではありません。
更なる詳細にご興味のある方は、チャプター 5 に記載のソースを参照してください。
オフラインで読む場合は、PDF バージョンをダウンロードしていただけます。

パターンを使ったテキストの処理は複雑な作業な作業になることがあります。
正規表現を使用する場合、一般的には、パターンを探すテキスト、パターンそのもの (正規表現)、マッチ (パターンに一致するテキストの部分) など、異なる種類のエンティティを伴います。
こういったエンティティを簡単に区別できるよう、本ドキュメントでは以下のルールを使用しています。

テキストのサンプルは、モノスペース書体で個別に、追加の引用符を使わずに書かれています。

```
?? ????????? ??????? "??" ?????????
```

区別しにくい場合に限り、本文中にある正規表現はこの例にもあるように、灰色のバックグラウンドで表示されています: `\".*?\"`。

マッチした部分は必要に応じて異なる色で強調表示されます。

?? "???????" ??????? "??" ?????????

大きめのコードサンプルは、次の例のようにボックスに分けています。

```
set t="?? ""???????"" ??????? ""??"" ?????????"
set r="\"".*?\"""
w $locate(t,r,,,tMatch)
```

2.歴史 (とトリビア) の紹介

1940 当初、神経生理学者により人間の神経系がモデル化され、
その何年後かに、ある数学者が「正規セット」と呼ぶ代数学を用いてこのモデルを説明しました。
そして、この代数学の表記が「正規表現」と名付けられました。

1965 年になり、正規表現は初めてコンピューターの世界で言及されるようになり、 当時 UNIX オペレーティングシステムの一部であったエディター QED に正規表現が導入されるようになりました。 そのエディターの後のバージョンで、すべてのテキスト行で正規表現のマッチを検索し、その結果を出力するコマンドシーケンス g / regular expression / p (*global, regular expression, print*) が提供されました。
このコマンドシーケンスが、最終的にスタンドアロンの UNIX コマンドラインプログラム「grep」になりました。

今日、正規表現 (RegEx) は、数多くのプログラミング言語において、様々な形で実装されています (セクション 3.3 参照)。

3.正規表現 101

Caché パターンマッチングと同様に、正規表現を使ってテキストデータのパターンを特定することができますが、後者の場合はより高い表現力を利用できます。 以下のセクションでは、正規表現のコンポーネントとその評価をまとめ、使用可能なエンジンをいくつか紹介します。使用方法は、チャプター 4 で詳しく説明します。

3.1.正規表現のコンポーネント

3.1.1.正規表現のメタ文字

以下の文字は正規表現として使用された場合に特別な意味を持ちます。

```
. * + ? ( ) [ ] \ ^ $ |
```

これらをリテラルとして使用する場合は、バックスラッシュを使ってエスケープする必要があります。
リテラルシーケンスを明示的に指定する場合は、`\Q` `\E` を使用します。

3.1.2.リテラル

通常のテキストとエスケープされた文字はリテラルとして扱われます。以下はその一部です。

abc	abc
\f	改ページ
\n	改行
\r	行頭復帰
\v	(垂直) タブ
\0+ 3 桁の数字 (例: \0101)	8 進数。Caché (ICU) で使用される正規表現エンジンは、最大で \0377 (10 進法では 255) までの 8 進数に対応しています。他のエンジンから正規表現を移行させる場合は、8 進数がどう処理されるかを事前に確認しておいてください。
\x+ 2 桁の数字 (例: \x41)	16 進数。ICU ライブラリに 16 進数を処理する別のオプションが記載されています。ICU ドキュメンテーションを参照してください (リンクはセクション 5.8 にあります)

3.1.3.アンカー

アンカーは、テキスト / 文字列の位置を一致させる場合に使用します。以下はその例です。

- \A 文字列の先頭
- \Z 文字列の末尾
- ^ テキストまたは行の先頭
- $ テキストまたは行の末尾
- \b 単語の境界
- \B 単語の境界ではない
- \< 単語の先頭
- > 単語の末尾

一部の RegEx エンジンは、単語を構成するものの正確な定義や、単語の区切り文字と見なされる文字の定義によって、動作が異なります。

3.1.4.量指定子

量指定子を使用すると、先行する要素がマッチとみなされるための出現回数を指定できます。

- {x} x 回の出現
- {x,y} x 回から y 回の出現
- * 0 回以上、{0,} と同じ
- + 1 回以上、{1,} と同じ
- ? 0 回または 1 回

最長一致

量指定子は「*最長一致*」、つまり、できるだけ多くの文字を一致させようとします。
以下のようなテキスト文字列があり、引用符内のテキストを見つける必要があるとしましょう。

```
This is "a text" with "four quotes".
```

セレクタは最長一致なため、正規表現 `\".*\"` を使うと、一致するテキストの数が多くなり過ぎてしまいます。

```
This is "a text" with "four quotes".
```

この例の正規表現 `.*` は、引用符のペアの間にある文字をできるだけ多く検出しようとします。
しかし、ドットセレクタ (`.`) により引用符も検出されてしまうため、期待している結果が得られません。

一部の正規表現エンジン (Caché で使用されるものを含む) では、クエスチョンマーク (?)
を追加することにより、量指定子の最長一致となることを抑制できます。 そうすることで、正規表現 `\".*?\"`
は、引用符で囲まれたテキストの 2 つの部分に一致し、期待通りの結果が得られます。

```
This is "a text" with "four quotes".
```

3.1.5.文字クラス (文字の範囲)

文字の範囲や文字のセットは、角括弧を使って `[a-zA-Z0-9]` または `[abcd]`
のように指定します。正規表現では、これを「文字クラス」と呼んでいます。 範囲の中で一致するのは 1
文字だけです。つまり、範囲定義内にある文字の順番は関係ありません。`[dbac]` と `[abcd]`
では同じ文字が一致します。

特定の文字の範囲を省く場合は、範囲定義の前 (角括弧内) に ^ を挿入します: `[^ abc]` と指定すると、a、b、c
以外のすべてが一致します。

一部の正規表現エンジンでは、事前定義された文字クラス (POSIX) を使用できます。以下はその一部です。

- `[:alnum:] [a-zA-z0-9]`
- `[:alpha:] [a-zA-Z]`
- `[:blank:] [ \t]`
- …

3.1.6.グループ

括弧を使用すると、正規表現の一部をグループ化することができます。
これは、セレクタのグループに量指定子を適用したり、同じ正規表現 (後方参照) および正規表現を呼び出す
Caché オブジェクトスクリプトコード (キャプチャバッファ) の両方からグループを参照する場合に便利です。

グループはネストできます。

下の正規表現は、順に 3 桁の数字、ダッシュ、大文字と数字の 3 つのペア、ダッシュ、先頭と同じ 3 桁の数字で構成される文字列に一致します。

```
([0-9]{3})-([A-Z][[0-9]){3}-\1
```

この例は、*後方参照* (以下を参照)
を使用して、構造だけでなく中身にも一致させる方法を示しています。後方参照 (紫) は、先頭の 3 桁の数字を末尾でも検索するようエンジンに指示しています (黄) 。 また、より複雑な構造 (緑色) に量指定子を適用する方法も示しています。

上記の正規表現は、以下の文字列に一致します。

`123-D1E2F3-123`

以下には一致しません。

`123-D1E2F3-456` (末尾の 3 桁の数字が先頭の 3 桁と異なる)

`123-1DE2F3-123` (中央部分が3 つの大文字、数字のペアではない)

`123-D1E2-123` (中央部分が 2 つの大文字、数字のペアしかない)

グループは、いわゆるキャプチャバッファの作成にも使用されます (セクション4.5.1を参照)。
これはとても強力な機能で、情報の一致と抽出を同時に実行できます！

3.1.7. 論理和指定子

論理和指定子を指定するには、`skyfall|done` のように縦線の文字を使います。 そうすることで、セクション 3.1.5 で説明した文字クラスを使う場合のように、より複雑な式を一致させることができます。

3.1.8.後方参照

後方参照を使用すると、以前に定義されたグループ (括弧内のセレクター) を参照できます。
下の正規表現の例は、同じ文字が 3 回繰り返す場合に一致します。

```
([a-zA-Z])\1\1
```

後方参照は「\x」で指定され、「x」は何番目の括弧で囲まれた式を参照するのかを意味します。

3.1.9.優先順位

1. () よりも [] を優先
2. シーケンスよりも , + および ? を優先。`ab` は `(ab)` ではなく、`a(b)` と同等である
3. 論理和指定子よりもシーケンスを優先。`ab|c` は、`a(b|c)` でなく、`(ab)|c` と同等である。

3.2.理論

通常、正規表現の評価は、下に紹介する 2 つの手段のどちらかで実施されます
(ここでは簡単に説明していますので、詳しい内容はチャプター 5 に記載の文献を参照してください)。

1. テキストを基にした判定 (DFA – *Deterministic Finite Automaton「決定性有限オートマトン」*)
   エンジンは入力されたテキスト文字を 1 文字ずつ確認し、それまで確認した文字を一致させようとする。
   入力されたテキストの末尾に到達すると、成功とします。

2. Regexを基にした判定(NFA – *Non-deterministic Finite Automaton「非決定性有限オートマトン」*)
   エンジンは、正規表現のトークンを 1 つずつ確認し、それをテキストに適用しようとします。
   最後のトークンに到達(かつ一致)すれば、成功とします。

手段 1 は決定的な方法です。実行時間は入力されるテキストの長さによります。
正規表現に使われるセレクタの順序が実行時間に影響を与えることはありません。

手段 2 は非決定的な方法です。エンジンは、一致するかエラーが発生するまで、正規表現に使われているセレクタのすべての組み合わせを確認します。 従ってこの方法は*一致しない*場合は特に時間がかかります(すべての可能な組み合わせを確認する必要があるため)。 セレクタの順番は、実行時間に影響を*与えます*。
ただし、この方法はバックトラックおよびキャプチャバッファを使用できます。

3.3.エンジン

正規表現エンジンは、プログラミング言語やオペレーティングシステムに組み込まれているものから、ほぼどこでも使用可能なライブラリまで、様々なものが存在します。
以下は評価手段別に分けた正規表現エンジンの例です。

- DFA: grep、awk、lex
- NFA: Perl、Tcl、Python、Emacs、sed、vi、ICU

下のテーブルは、様々なプログラミング言語やライブラリで使用可能な正規表現の機能を比較したものです。

詳細はこちらをお読みください: https://en.wikipedia.org/wiki/Comparisonofregularexpressionengines

4.RegEx と Caché

InterSystems Caché では、正規表現に ICU ライブラリが使用されています。その機能の多くは、Caché オンラインドキュメンテーションでご確認いただけます。 (後方参照などを含む) 完全な詳細は、ICU ライブラリのオンラインドキュメンテーションを参照してください。 – ICU へのリンクはセクション 5.8 をご覧ください。 以下のセクションで、その使用方法を簡単に紹介します。

4.4.$match() と $locate()

Caché ObjectScript (COS) では、2 つの関数 $match() と $locate() により、ICU ライブラリが提供する多くの Regex 機能を直接使用できます。$match(String, Regex) は、指定された Regex パターンを基に入力文字列を検索します。 マッチが見つかった場合は 1 を、それ以外の場合は 0 を返します。

例:

- w $match("baaacd",".*(a)\1\1.*") は 1 を返します
- w $match("bbaacd",".*(a)\1\1.*") は 0 を返します

$locate(String,Regex,Start,End,Value) は、$match() と同様に、指定された Regex パターンを基に入力文字列を検索します。 ですが、$locate() は扱いやすい上に、より多くの情報を返します。
Start により、入力文字列内でパターンの検索を開始する位置を $locate に指示します。 $locate() は、マッチを見つけると、その最初の文字の位置を返し、End をマッチの次の文字の位置に設定します。
マッチの内容は Value として返されます。

$locate() は、マッチが見つからないと 0 を返し、End と Value が指定されていも、それを変更することはありません。End と Value

は参照渡しで渡されるので、繰り返し使用する場合は注意が必要です (ループ内で使用する場合など)。

例:

- w $locate("abcdexyz",".d.",1,e,x) は 3 を返すと同時に、e は 6 に、x は "cde" に設定されます

$locate() は、パターンマッチングを実行すると同時に、最初のマッチの内容を返すことができます。
すべてのマッチの内容を抽出する必要がある場合は、ループの中で $locate()
を繰り返し呼び出すか、%Regex.Matcher のメソッドを実行します (次のセクションで解説)。

4.5.%Regex.Matcher

%Regex.Matcher を使用すると、$match() および $locate() と同様に、ICU
ライブラリの正規表現機能を使用できます。 しかし、%Regex.Matcher
を使用すると、高度な機能も利用でき、複雑なタスクの扱いがとてもシンプルになります。 次のセクションでは
、キャプチャバッファについてもう一度確認し、正規表現を使って文字列を置き換える方法やランタイムの動作を制御する方法について説明します。

4.5.1.キャプチャバッファ

グループや後方参照、$locate() のセクションで説明してきましたが、正規表現を使用すると、テキスト内のパターンを検索すると同時に、一致した内容を返すことができます。 これは、抽出したいパターンの部分を括弧 (グルーピング) に入れて行います。
マッチが成功すると、一致したすべてのグループの内容がキャプチャバッファに入ります。
これはマッチした内容を Value パラメーターで返す $locate() とは少し異なるので注意が必要です。$locate() はマッチ全体を返す一方で、キャプチャバッファは、マッチ (グループ) への*部分的*なアクセスを可能にします。

これを使用するには、%Regex.Matcher
クラスのオブジェクトを作成し、それに正規表現と入力文字列を渡します。 それから、%Regex.Matcher
が提供するメソッドを 1 つ呼び出せば、実際の作業を実行することができます。

例 1 (シンプルなグループ):

```
set m=##class(%Regex.Matcher).%New("(a|b).*(de)", "abcdeabcde")
w m.Locate() 1 ???
w m.Group(1) a ???
w m.Group(2) de ???
```

例 2 (ネストされたグループと後方参照):

```
set m=##class(%Regex.Matcher).%New("((a|b).*?(de))(\1)", "abcdeabcde")
w m.Match()                    1 ???
w m.GroupCount                 4 ???
w m.Group(1)                   abcde ???
w m.Group(2)                   a ???
w m.Group(3)                   de ???
w m.Group(4)                   abcde ???
```

(ネストされたグループの順番に注目してください。始め括弧がグループの始まりを意味するため、インデックス番号は外側のグループよりも内側のグループの方が高くなっています)

先ほども触れましたが、キャプチャバッファはパターンを一致させると同時に一致した内容を抽出できる、とても強力な機能です。 正規表現がないと、ステップ 1 として (パターンマッチオペレーターを使用するなどして) マッチを探し、ステップ 2 として何らかの条件を基に、マッチした内容を抽出 (または部分的に抽出)

しなくてはならなくなります。

パターンを部分的にグループ化する必要がある (例: その部分に量指定子を適用するため)
が、マッチした部分の内容をキャプチャバッファに取り入れたくない場合は、以下の例 3
で示すように、グループの前に疑問符とコロンを順に挿入し、そのグループを "非キャプチャリング (non-capturing)" または "内気 (shy)" なグループとして定義することができます。

例 3 (内気 "shy" なグループ):

```
set m=##class(%Regex.Matcher).%New("((a|b).*?(?:de))(\1)","abcdeabcde")
w m.Match()                         1 ???
w m.Group(1)                        abcde ???
w m.Group(2)                        a ???
w m.Group(3)                        abcde ???
w m.Group(4)                        <REGULAR ???EXPRESSION>zGroupGet+3^%Regex.Matcher.
1
```

4.5.2.置換

%Regex.Matcher は、マッチした内容を即座に置き換えることができる、ReplaceAll() と ReplaceFirst()
というメソッドも提供します。

```
set m=##class(%Regex.Matcher).%New(".c.","abcdeabcde")
w m.ReplaceAll("xxxx")         a**xxxx**ea**xxxx**e ???
w m.ReplaceFirst("xxxx")      a**xxxx**eabcde ???
```

置換文字列でグループを参照することもできます。 前の例のパターンにグループを追加した場合、置換文字列に
$1 を含めることでその内容を参照することができます。

```
set m=##class(%Regex.Matcher).%New("<span style="background-color:#D3D3D3;">.</span><
strong><span style="background-color:#D3D3D3;">(</span></strong><span style="backgrou
nd-color:#D3D3D3;">c</span><strong><span style="background-color:#D3D3D3;">)</span></
strong><span style="background-color:#D3D3D3;">.</span>","abcdeabcde")
w m.ReplaceFirst("xx$1xx")   axxcxxeabcde ???
```

マッチした完全な内容を置換文字列に含めるには、$0 を使用します。

```
w m.ReplaceFirst("xx$0xx")    axxbcdxxeabcde ???
```

4.5.3.OperationLimit

セクション 3.2 で、正規表現を評価する 2 つの方法 (DFA と NFA) について解説しました。 Caché
で使用される正規表現エンジンは、非決定的有限オートマトン (NFA) です。
したがい、特定の入力文字列において様々な正規表現を評価するのにかかる時間は異なる場合があります。 [1]

%Regex.Matcher オブジェクトのプロパティ OperationLimit を使えば、(*クラスタ*と呼ばれる)
実行単位の数を制限することができます。 クラスターの実行にかかる正確な時間は、環境によって異なります。
通常、クラスターの実行はわずか数ミリ秒で完了します。 しかし、OperationLimitは 0 (制限なし)
にデフォルト設定されています。

4.6.実例: Perl から Caché への移行

このセクションでは、Perl から Caché への移行において、正規表現が関連する部分について説明します。 Perl
スクリプトは、文字のマッチと抽出の両方に使用される数十個の多少複雑な正規表現で構成されていました。

もし、Caché で正規表現の機能を使用できなかったとしたら、Caché
への移行プロジェクトは大掛かりな作業となったことでしょう。 幸い、Caché
では正規表現の機能を使用できる上に、Perl スクリプトの正規表現は、ほぼ何の変更も加えずに Caché
で使用することができました。

以下のリンクから Perl スクリプトを一部ご覧いただけます。

正規表現を Perl から Caché に移行する上で、唯一必要となった変更は (正規表現に大文字と小文字を区別させる) 修飾子 /i に関するもので、正規表現の末尾から先頭に移動させる必要がありました。

Perl では、キャプチャバッファの中身は特別な変数にコピーされます (上の Perl コードでいう $1 と $2)。 Perl プロジェクトのほぼ全ての正規表現で、このメカニズムが使用されていました。
これに似た作業を行えるよう、Caché Object Script ではシンプルなラッパーメソッドが作成されました。
%Regex.Matcher
を使ってテキスト文字列に対し正規表現を評価し、キャプチャバッファの中身をリストとして返すというものです ($lb())。

以下がその Caché Object Script コードです。

```
if ..RegexMatch(
        tVCSFullName,
        "(?i)[\\/]([^\\^\/]+)[\\/]ProjectDB[\\/](.+)[\\/]archives[\\/]",
        .tCaptureBufferList)
        {
              set tDomainPrefix=$zcvt($lg(tCaptureBufferList,1), "U")
              set tDomain=$zcvt($lg(tCaptureBufferList,2), "U")
        }
…

Classmethod RegexMatch(pString as %String, pRegex as %String, Output pCaptureBuffer="
") {

        #Dim tRetVal as %Boolean=0
        set m=##class(%Regex.Matcher).%New(pRegex,pString)
        while m.Locate() {
              set tRetVal=1
              for i=1:1:m.GroupCount {
                    set pCaptureBuffer=pCaptureBuffer_$lb(m.Group(i))
              }
        }
        quit tRetVal
}
```

5.リファレンス情報

5.7.一般情報

一般情報およびチュートリアル

- http://www.regular-expressions.info/engine.html

チュートリアルおよび例

- http://www.sitepoint.com/demystifying-regex-with-practical-examples/

正規表現エンジンの比較

- https://en.wikipedia.org/wiki/Comparisonofregularexpressionengines

クイックガイド

- https://www.cheatography.com/davechild/cheat-sheets/regular-expressions/pdf/

書籍

- Jeffrey E. F. Friedl (著): “ Mastering Regular Expressions (詳説 正規表現) ”http://regex.info/book.html 参照)

5.8.Caché オンラインドキュメンテーション

- Caché で正規表現を使う方法の概要:
  http://docs.intersystems.com/latestj/csp/docbook/DocBook.UI.Page.cls?KEY=GCOSregexp
- $match() に関するドキュメンテーション:
  http://docs.intersystems.com/latestj/csp/docbook/DocBook.UI.Page.cls?KEY=RCOSfmatch
- $locate() に関するドキュメンテーション:
  http://docs.intersystems.com/latestj/csp/docbook/DocBook.UI.Page.cls?KEY=RCOSflocate
- %Regex.Matcher のクラスリファレンス:
  http://docs.intersystems.com/latestj/csp/documatic/%25CSP.Documatic.cls?APP=1&LIBRARY=%25SYS&CLASSNAME=%25Regex.Matcher

5.9.ICU

上述のとおり、InterSystems Caché は ICU エンジンを使用します。
包括的なドキュメンテーションはオンラインでご利用いただけます。

- http://userguide.icu-project.org/strings/regexp
- http://userguide.icu-project.org/strings/regexp#TOC-Regular-Expression-Metacharacters
- http://userguide.icu-project.org/strings/regexp#TOC-Regular-Expression-Operators
- http://userguide.icu-project.org/strings/regexp#TOC-Replacement-Text
- http://userguide.icu-project.org/strings/regexp#TOC-Flag-Options

5.10.ツール

正規表現を作成するにあたり、開発者をサポートするツールは、無料で使用できるものから、商用ライセンスが付属するものまで数多く存在します。 私の個人的なお気に入りは RegexBuddy (http://www.regexbuddy.com/) です。インタラクティブで視覚的な機能を使用できるので、正規表現の作成とテストを様々な方法で行えます。

#ObjectScript #チュートリアル #Caché #InterSystems IRIS

ソースURL:
https://jp.community.intersystems.com/post/cach%C3%A9-%E3%81%AB%E3%81%8A%E3%81%91%E3%82%8B%E6%AD%A3%E8%A6%8F%E8%A1%A8%E7%8F%BE%E3%81%AE%E4%BD%BF%E7%94%A8%E3%81%AB%E3%81%A4%E3%81%84%E3%81%A6