Map! データ型

目次

1. 概要 1
2. リテラルシンタックス 1
3. 初期化シンタックス 1
4. valueの取得 2
5. keyとvalueの変更 3
6. keyの削除 5
7. リフレクション 6

1. 概要

mapはkeyとvalueのペアで関連付けされた配列です。mapは高速な読み取りアクセス（内部的にはハッシュテーブルを使用しています）と取り扱いやすいシンタックスを提供します。`hash!` データ型と異なり、mapはseriesではなく、したがってオフセットと位置の概念を持ちません。概念的には、 `map!` データ型は `hash!` と `object!` の中間に位置するデータ型です。

2. リテラルシンタックス

```
#(<key> <value>...)

<key>   : ハッシュのキー, 次のデータ型の値が使えます: scalar!,all-word!, any-string!
<value> : any-type!の値
```

3. 初期化シンタックス

```
make map! <spec>

<spec> : keyとvalueのペアまたはintegerの値
```

もし *spec* 引数がintegerの場合、指定された数のスロット分のメモリを確保した空のmap!が生成されます。（通常は、mapの拡大は動的に行われます。）

メモ:

- mapのボディ、つまりspecブロックということになりますが、この要素数は偶数である必要があります。奇数の場合は、エラーが発生します。
- 値は **reduce** されません。そのため、初期化シンタックスでは例えばlogic!のような具体的な値が必要になります。

例:

```
#(a: 1 b: 2)
== #(
    a: 1
    b: 2
)

make map! [a 1 'b 2 "c" 3]
== #(
    a: 1
    b: 2
    "c" 3
)
```

もしkeyが **any-word!** 型だった場合、mapにセットされたkeyは **set-word!** に変換されます。これはmapの中身がkeyとvalueの組み合わせに見えるようにするためです。しかし、取り扱いやすさの観点から、wordのkeyにアクセスする時には、set-wordでkeyを指定する必要はなく、シンプルにwordを指定できます（map生成時も、中身にアクセスする時も同様です）。同様に、 `keys-of` リフレクタ（後述のリフレクションのセクションで説明します）は、set-wordではなくwordを返します。これも、後続の処理を簡単にするためです（特にマッチング処理はset-wordよりもwordの方が簡単になります）。

NOTE:

- hash!やblock!のように、map!はデフォルトでは大文字・小文字を保持します が、参照時には区別しません。
- もし `none` が値として指定されている場合、keyは生成されません（「keyの削除」セクションを参照してください）。
- mapを生成する時には、全てのキーはディープコピーされます。
- series!型のvalueはmapの生成時にはコピーされません。選択はユーザーに任されます（これは一般的なユースケースのための最適化です）。

mapを生成する別の方法は、既存のmapに対して `copy` アクションを使用することです。

4. valueの取得

パス形式を使用する場合:

```
<map>/<key>
get '<map>/<key>

<map> : map!の値を参照しているword
<key> : wordのkey
```

selectアクションを使用する場合:

```
select <map> <key>

<map> : mapの値
<key> : 全ての有効な型のkey
```

これら全ての読み取りアクセスは大文字・小文字を区別しません。区別して値を識別したい場合、 `/case` リファインメントが存在する方法であれば可能です。

```
get/case '<map>/<key>
select/case <map> <key>
```

存在しないkeyにアクセスしようとした場合、 `none` が値として返されます。

例:

```
m: #(Ab: 2 aB: 5 ab: 10)
m/ab
== 2
select m 'aB
== 2
get/case 'm/aB
== 5
select/case m 'ab
== 10
```

5. keyとvalueの変更

パス形式を使用する場合:

```
<map>/<key>: <value>
set '<map>/<key> <value>

<map>   : map!の値を参照しているword
<key>   : map内のvalueを指定するwordのkey
<value> : 任意のvalue
```

変更用のアクションを使用する場合:

```
put <map> <key> <value>

<map> : mapの値
<key> : map内のvalueを指定する任意のkey値
```

一括変更するには:

```
extend <map> <spec>

<map>  : mapの値
<spec> : （1つ以上の）名前／値のペアのブロック
```

これらの全ての書き込みアクセスは大文字・小文字を区別しません。区別して値を識別したい場合、 `/case` リファインメントが存在する方法であれば可能です。

```
set/case '<map>/<key> <value>
put/case <map> <key> <value>
extend/case <map> <spec>
```

`extend` ネイティブは一度にたくさんのkeyを設定することができるため、一括変更時に便利です。

NOTE:

- mapに存在しないkeyを設定しようとした場合、単純に新しい**key**として追加されます。
- すでに存在するkeyを追加しようとした場合、keyに対応するvalueが変更されます。新しいものは追加されません（デフォルトでは大文字・小文字を区別しません）。

例:

```
m: #(Ab: 2 aB: 5 ab: 10)
m/ab: 3
m
== #(
    Ab: 3
    aB: 5
    ab: 10
)

put m 'aB "hello"
m
== #(
    Ab: "hello"
    aB: 5
    ab: 10
)

set/case 'm/aB 0
m
== #(
    Ab: "hello"
    aB: 0
    ab: 10
)
set/case 'm/ab 192.168.0.1
== #(
    Ab: "hello"
    aB: 0
    ab: 192.168.0.1
)

m: #(%cities.red 10)
extend m [%cities.red 99 %countries.red 7 %states.red 27]
m
== #(
    %cities.red 99
    %countries.red 7
    %states.red 27
)
```

6. keyの削除

mapからkey/valueのペアを削除したい場合、前述のいずれかの方法でkeyの値として `none` をセットしてください。

例:

```
m: #(a: 1 b 2 "c" 3 d: 99)
m
== #(
    a: 1
    b: 2
    "c" 3
    d: 99
)
m/b: none
put m "c" none
extend m [d #[none]]
m
== #(
    a: 1
)
```

NOTE 上記の例における初期化シンタックスは、 `none!` が `word!` として扱われてしまわないようにするためにこのような記述になっています。このやり方は必要なspec引数のblockを作る一例です。

また、 `clear` アクションを使用することで一度に全てのkeyを削除することもできます。

```
clear #(a 1 b 2 c 3)
== #()
```

7. リフレクション

- `find` はkeyがmapの中にあれば `true` 、なければ `none` を返します。

  ```
  find #(a 123 b 456) 'b
  == true
  ```

- `length?` はmapの中のkey/valueのペアの数を返します。

  ```
  length? #(a 123 b 456)
  == 2
  ```

- `keys-of` はmapの中のkeyのリストをblockで返します（set-wordはwordに変換されます）。

  ```
  keys-of #(a: 123 b: 456)
  == [a b]
  ```

- `values-of` はmapの中のvalueのリストをblockで返します。

```
values-of #(a: 123 b: 456)
== [123 456]
```

- `body-of` はmapの中の全てのkey/valueのペアをblockで返します。

```
body-of #(a: 123 b: 456)
== [a: 123 b: 456]
```