

2020/12/16 ライブ翻訳チャレンジ

Method values

If the expression x has static type T and M is in the method set of type T , $x.M$ is called a method value. The method value $x.M$ is a function value that is callable with the same arguments as a method call of $x.M$. The expression x is evaluated and saved during the evaluation of the method value; the saved copy is then used as the receiver in any calls, which may be executed later.

メソッド値

式 x が静的型 T をもち、かつ M が型 T のメソッドセットに属する時、 $x.M$ をメソッド値と呼びます。メソッド値 $x.M$ は、メソッド呼び出し $x.M$ と同じ引数で呼び出すことができる関数値です。メソッド値 $x.M$ の評価の際に式 x も評価され保存されます。保存されたコピーは呼び出しの時にレシーバーとして用いられます。その呼び出しは後に行われる事もあります。

The type T may be an interface or non-interface type.

型 T はinterface型であってもinterface型でなくても構いません。

As in the discussion of method expressions above, consider a struct type T with two methods, M_v , whose receiver is of type T , and M_p , whose receiver is of type $*T$.

メソッド式についての以前の議論と同じように、型 T のレシーバを持つメソッド M_v と、型 $*T$ のレシーバを持つメソッド M_p を考えてみましょう。

```
type T struct {
    a int
}
func (tv T) Mv(a int) int    { return 0 } // value receiver
func (tp *T) Mp(f float32) float32 { return 1 } // pointer receiver

var t T
var pt *T
func makeT() T
```

The expression

$t.M_v$
yields a function value of type

$\text{func}(\text{int}) \text{int}$

式`t.Mv`は、型`func(int)int`を持つ関数値を産出します。

These two invocations are equivalent:

```
t.Mv(7)
f := t.Mv; f(7)
```

この2つの呼び出しは等価です。

Similarly, the expression

```
pt.Mp
yields a function value of type
```

```
func(float32) float32
```

同様に、式`pt.Mp`は、`func(float32) float32`型を持つ関数値を生み出します。

As with selectors, a reference to a non-interface method with a value receiver using a pointer will automatically dereference that pointer: `pt.Mv` is equivalent to `(*pt).Mv`.

セレクターと同じように、インタフェースメソッドでないバリューレシーバのメソッドにポインタを使うときは、自動的にdereferenceします。つまり、`pt.Mv`は、`(*pt).Mv`と等価です。

As with method calls, a reference to a non-interface method with a pointer receiver using an addressable value will automatically take the address of that value: `t.Mp` is equivalent to `(&t).Mp`.

メソッド呼び出しと同様に、インタフェースメソッドではないポインタレシーバのメソッドのレシーバとしてアドレス化可能な値を用いた場合には、自動的にその値のアドレスが取得されます。つまり、`t.Mp`は`(&t).Mp`と等価です。

```
f := t.Mv; f(7) // like t.Mv(7)
f := pt.Mp; f(7) // like pt.Mp(7)
f := pt.Mv; f(7) // like (*pt).Mv(7)
f := t.Mp; f(7) // like (&t).Mp(7)
f := makeT().Mp // invalid: result of makeT() is not addressable
```

Although the examples above use non-interface types, it is also legal to create a method value from a value of interface type.

上記の例は非インターフェース型を使っていますが、インターフェース型の値からもメソッド値を作ることができます。

```
var i interface { M(int) } = myVal
f := i.M; f(7) // like i.M(7)
Index expressions
A primary expression of the form
```

`a[x]`

denotes the element of the array, pointer to array, slice, string or map `a` indexed by `x`. The value `x` is called the index or map key, respectively. The following rules apply:

インデックス式

`a[x]`という形のプライマリ式は、`x`でインデックスされた配列、配列へのポインタ、スライス、文字列、もしくはマップである`a`の要素を表します。この`x`はインデックスまたはマップのキーと呼ばれます。これについては次のルールが適用されます。

If `a` is not a map:

the index `x` must be of integer type or an untyped constant

a constant index must be non-negative and representable by a value of type `int`

a constant index that is untyped is given type `int`

the index `x` is in range if $0 \leq x < \text{len}(a)$, otherwise it is out of range

mapではない場合:

- インデックス`x`は、整数型か、型付なし定数でなければなりません。
- 定数のインデックスは非負であり、かつ`int`型で表現可能でなければなりません。
- 型付けなしの定数のインデックスは、`int`型が与えられます
- インデックス`x`は、 $0 \leq x < \text{len}(a)$ の時は範囲内であり、そうでない時は範囲外です。

For `a` of array type `A`:

a constant index must be in range

if `x` is out of range at run time, a run-time panic occurs

`a[x]` is the array element at index `x` and the type of `a[x]` is the element type of `A`

配列型`A`の場合:

- 定数インデックスはin rangeでなければならない
- `x`が実行時に範囲外の場合、ランタイムパニックが発生する
- `a[x]`はインデックス`x`に位置する配列要素であり、`a[x]`の型は、`A`の要素型である

For `a` of pointer to array type:

`a[x]` is shorthand for `(*a)[x]`

配列へのポインタ型の場合:

- `a[x]`は、`(*a)[x]`の省略表現です。

For `a` of slice type `S`:

if `x` is out of range at run time, a run-time panic occurs

$a[x]$ is the slice element at index x and the type of $a[x]$ is the element type of S
For a of string type:

a constant index must be in range if the string a is also constant

if x is out of range at run time, a run-time panic occurs

$a[x]$ is the non-constant byte value at index x and the type of $a[x]$ is byte

$a[x]$ may not be assigned to

For a of map type M :

x 's type must be assignable to the key type of M

if the map contains an entry with key x , $a[x]$ is the map element with key x and the type of $a[x]$ is the element type of M

if the map is nil or does not contain such an entry, $a[x]$ is the zero value for the element type of M

Otherwise $a[x]$ is illegal.

An index expression on a map a of type $\text{map}[K]V$ used in an assignment or initialization of the special form

$v, ok = a[x]$

$v, ok := a[x]$

$\text{var } v, ok = a[x]$

yields an additional untyped boolean value. The value of ok is true if the key x is present in the map, and false otherwise.

Assigning to an element of a nil map causes a run-time panic.