Metaprogramming in Julia

Iblis Lin

2018/8/11

Introduction

粗淺的分類

- Text-based
  - e.g. macro in C
- Abstract Syntax Tree Level
  - Lisp
  - Julia

Introduction

Metaprogramming 把程式本身視為 data 的一種

Introduction

那麼有 data structure 跟 manipulations

Introduction

那麼有 data structure 跟 manipulations
在 Julia 中有 `Expr` 這個 type

Construct Expressions

julia 0.7/1.0:

```
julia> e = Meta.parse("42 + 1")
:(42 + 1)
```

julia 0.6:

```
julia> e = parse("42 + 1")
:(42 + 1)
```

Expressions

```
julia> typeof(e)
Expr
```

Fields of `Expr`:

- `head::Symbol`
- `args::Array{Any,1}`

Expressions

```
julia> e.head
:call

julia> e.args
3-element Array{Any,1}:
    :+
 42
  1
```

Expressions

程式已經用 `Expr` 來表示了，那麼怎麼執行？

Expressions

程式已經用 **Expr** 來表示了，那麼怎麼執行？

```
julia> e
:(42 + 1)

julia> eval(e)
43
```

Expressions is mutable

所以可以各種改。

```
e.args[1] = :-
e.args[3] = 50
```

其他生出 Expression 的方式

1. 直接 call constructor

```
Expr(:call, :+, 2, 3)
```

其他生出 Expression 的方式

1. 直接 call constructor

```
Expr(:call, :+, 2, 3)
```

2. Quoting

```
:(1 + 2)
```

```
quote
    1 + 2
    2 + 3
end
```

Printing Expr Instance

```
dump(e)
```

```
Meta.show_sexpr(e)
```

The Symbol Type

Symbol

Like symbol in Lisp or atom in Erlang.

```
julia> :foo
:foo

julia> typeof(:foo)
Symbol
```

Symbol

Like symbol in Lisp or atom in Erlang.

```
julia> :foo
:foo

julia> typeof(:foo)
Symbol
```

```
julia> Symbol("bar-1")
Symbol("bar-1")
```

Symbol

在 `Expr` 中的 identifier 會使用 symbol

- variable name
- function name
- ...

Symbol

在 Expr 中的 identifier 會使用 symbol

- variable name
- function name
- ...

```
julia> e = :(x = 1)
:(x = 1)

julia> e.args
2-element Array{Any,1}:
  :x
 1
```

Interpolation

動態的產生 Expr 的手段之一

Interpolation

用 $ 來 reference 外面的變數
長得很像 string interpolation

```
julia> x = 42;

julia> e = :($x + y)
:(42 + y)
```

Splatting Interpolation

可以展開整個 array

```
julia> A
3-element Array{Symbol,1}:
 :x
 :y
 :z

julia> e = :(f($(A...)))
:(f(x, y, z))
```

Eval and Scope

打開 REPL 後，我在哪裡？

Eval and Scope

打開 REPL 後，我在哪裡？

```
julia> @__MODULE__
Main
```

Eval and Scope

打開 REPL 後，我在哪裡？

```
julia> @__MODULE__
Main
```

```
foo = 1
e = :(foo += 42)
eval(e)
```

請問現在 foo 是多少？

Eval and Scope

`eval` 的影響範圍是 module 的 global scope
能夠對 module 的 global scope 有 side effect

Eval and Scope

eval 的影響範圍是 module 的 global scope
能夠對 module 的 global scope 有 side effect

```
function f()
  e = :(foo -= 100)
  eval(e)
end

foo = 1
f()
```

Eval and Scope

那麼這個呢？

```
e = :(bar + 1)
eval(e)
```

Example

那麼我們現在來實際解決點問題。

Example

那麼我們現在來實際解決點問題。
現在希望有個 helper function，這個 function 能夠對任意的新 `struct` 建立好看的 `show` function

Example

```
julia> struct Bar
         a::Int
         b::Bool
         magic::Any
       end

julia> bar = Bar(1, false, "good")
Bar(1, false, "good")

julia> make_show(Bar, :magic, :b)

julia> bar
magic -> good
b -> false
```

Example

```
function make_show(T::DataType, fields::Symbol...)
  fields = QuoteNode.(fields)
  print_exprs = [ :(println(io, "$($f) -> ", getfield(x, $f))) for f in fields]

  e = :(Base.show(io::IO, x::$T) = $(print_exprs...))
  eval(e)
end
```

Macro

Macro

Macro 能夠接受參數，而 return 一個 expression，並且立即執行這個 expression。

Macro

Macro 能夠接受參數，而 return 一個 expression，並且立即執行這個 expression。

```
julia> macro m()
           :(println("Hello, Macro"))
       end
@m (macro with 1 method)

julia> @m()
Hello, Macro
```

Macro

試試看參數

```
julia> macro m(name)
           :(println("Hello, ", $name))
       end
@m (macro with 2 methods)

julia> @m("Iblis")
Hello, Iblis
```

Macro – Debugging

可以用 `@macroexpand` 這個 macro 來看看你的 macro 回傳了啥

```
julia> @macroexpand(@m("Iblis"))
:((Main.println)("Hello, ", "Iblis"))
```

Macro – Syntax Sugar

在 call 一個 macro 的時候，可以偷懶不寫括號。

```
julia> @m "Iblis"
Hello, Iblis
```

而所有參數用空白切開。

Macro – Syntax Sugar

在 call 一個 macro 的時候，可以偷懶不寫括號。

```
julia> @m "Iblis"
Hello, Iblis
```

而所有參數用空白切開。

```
julia> @macroexpand @m "Iblis"
:((Main.println)("Hello, ", "Iblis"))
```

Macro – Example

```
macro make_show(T::Symbol, fields::Symbol...)
  fields = QuoteNode.(fields)
  print_exprs = [ :(println(io, "$($f) -> ", getfield(x, $f))) for f in fields]
  :(Base.show(io::IO, x::$T) = $(print_exprs...))
end
```

Macro – Parse time and Runtime

```
macro m(...)
  # parse time
  # ...

  return :(#= excuted at runtime =#)
end
```

Non-Standard String Literals

Non-Standard String Literals

這個東西就是透過 macro 完成的

e.g.

```
julia> VERSION
v"1.0.0"

julia> v"4.2"
v"4.2.0"

julia> typeof(v"4.2")
VersionNumber
```

Non-Standard String Literals

```
julia> using Sockets

julia> ip"1.1.1.1"
ip"1.1.1.1
```

Non-Standard String Literals

只要定義 `@x_str` 即可。

Non-Standard String Literals

只要定義 `@x_str` 即可。

```
macro foo_str(s)
  :(reverse($s))
end
```

Non-Standard String Literals

只要定義 `@x_str` 即可。

```
macro foo_str(s)
  :(reverse($s))
end
```

```
julia> foo"abc"
"cba"
```

Generated Functions

Generated Functions

當我們需要根據 Type 資訊，動態產生出不同的 function body，那麼就使用 generated functions。

Generated Functions

對於已經看過的 input type 組合，generated functions 的 function body 會 cache 起來，下次直接使用。

Generated Functions

```
@generated function bar(x)
  if x <: Integer
    :(x ^ 2)
  else
    :(x)
  end
end
```

Q & A