Julia 超新手教學 III part 2

by 杜岳華

Outline

- define methods
- method ambiguity
- parametric method
- empty generic function

Define methods

```
In [1]: f(x::Float64, y::Float64) = 2x + y

Out[1]: f (generic function with 1 method)

In [2]: f(2.0, 3.0)

Out[2]: 7.0
```

看起來不就跟function一樣嗎？

Function與method的差別

In [3]:

```
f(x::Number, y::Number) = 2x - y
f(2.0, 3)
```

Out[3]:

```
1.0
```

你會發現Julia幫f(Float64, Int64)這樣的組合挑選了f(Number, Number)這個method

In [4]:

```
methods(f)  # 你可以查詢目前這個函式名稱有多少種實作
```

Out[4]:

2 methods for generic function **f**:

- f(x::**Float64**, y::**Float64**) in Main at In[1]:1
- f(x::**Number**, y::**Number**) in Main at In[3]:1

在 Julia，function 指的是f，這個**介面**

Method 指的則是f(x::Number, y::Number) = 2x - y，這個**實作**

介面與實作

介面

```
fly
```

實作

```
fly(bird::Bird) = println("Bird flies.")

fly(airplane::Airplane) = println("Airplane flies.")
```

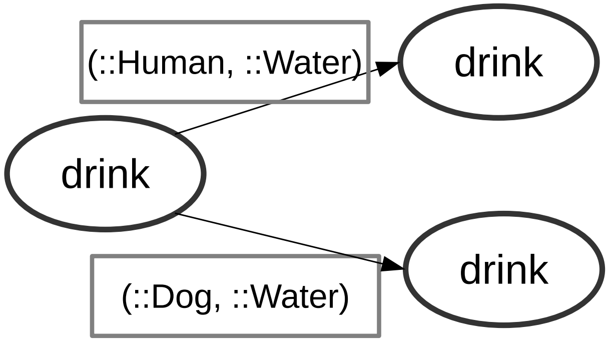
程式語言，如同人類的自然語言一樣，對應不同的**情境**，同一個詞有不同意思。

最多變的會是**行為**

為了對應不同的情境，可能有不同版本的實作

多重分派（Multiple dispatch）

對程式語言來說，他要如何決定要使用哪一個function的實作版本？



在Julia中，如果有很多個相同名字的methods的話，要決定用哪一個呢？

Julia會依據**參數的數量**跟**method中所有型別種類**決定要挑哪一個method出來執行

使用**method中所有型別種類**決定要執行哪一個method，這樣的方法稱為multiple dispatch

Method ambiguity

```
In [5]: g(x::Float64, y) = 2x + y
```

```
Out[5]: g (generic function with 1 method)
```

```
In [6]: g(x, y::Float64) = x + 2y
```

```
Out[6]: g (generic function with 2 methods)
```

```
In [7]: g(3.0, 4.0)
```

```
MethodError: g(::Float64, ::Float64) is ambiguous. Candidates:
  g(x, y::Float64) in Main at In[6]:1
  g(x::Float64, y) in Main at In[5]:1
Possible fix, define
  g(::Float64, ::Float64)

Stacktrace:
 [1] top-level scope at In[7]:1
```

這樣的定義會造成語意不清，g(Float64, Float64)要執行哪一條呢

由精確到廣義的定義順序是很棒的方式

```
In [8]:  g(x::Float64, y::Float64) = 2x + 2y
         g(x::Float64, y) = 2x + y
         g(x, y::Float64) = x + 2y

Out[8]:  g (generic function with 3 methods)

In [9]:  g(2.0, 3)

Out[9]:  7.0

In [10]: g(2, 3.0)

Out[10]: 8.0

In [11]: g(2.0, 3.0)

Out[11]: 10.0
```

Example: replace if-else by dispatching

In [12]:

```
xs = ["123", ["23", "345"], ["1234", "456", "789"], "567"]
collections = []
for x in xs
    if x isa String
        push!(collections, x)
    elseif x isa Vector
        append!(collections, x)
    end
end
```

In [13]:

```
collections
```

Out[13]:

```
7-element Array{Any,1}:
 "123"
 "23"
 "345"
 "1234"
 "456"
 "789"
 "567"
```

In [14]:

```
handle!(collections, x::String) = (push!(collections, x))
handle!(collections, x::Vector) = (append!(collections, x))
```

Out[14]:

```
handle! (generic function with 2 methods)
```

In [15]:

```
collections = []
for x in xs
  handle!(collections, x)
end
```

In [16]:

```
collections
```

Out[16]:

```
7-element Array{Any,1}:
 "123"
 "23"
 "345"
 "1234"
 "456"
 "789"
 "567"
```

Parametric method

聰明的設計讓 multiple dispatch 替你"回答問題"

```
In [17]: same_type(x::T, y::T) where {T} = true
         same_type(x, y) = false
```

```
Out[17]: same_type (generic function with 2 methods)
```

```
In [18]: same_type(1, 2)  # 兩者型別相同
```

```
Out[18]: true
```

```
In [19]: same_type(1, 2.0)  # 兩者型別不同
```

```
Out[19]: false
```

Example

In [20]:
```
concat(v::Vector{T}, x::T) where {T} = [v..., x]
```

Out[20]:
```
concat (generic function with 1 method)
```

In [21]:
```
concat([1, 2, 3], 4)
```

Out[21]:
```
4-element Array{Int64,1}:
 1
 2
 3
 4
```

In [22]:
```
concat([1, 2, 3], 4.0)
```

```
MethodError: no method matching concat(::Array{Int64,1}, ::Float64)
Closest candidates are:
  concat(::Array{T,1}, !Matched::T) where T at In[20]:1

Stacktrace:
 [1] top-level scope at In[22]:1
```

在方法上 加上限制

In [23]:
```
foobar(a, b, x::T) where {T <: Integer} = (a, b, x)  # 限制參數型別
```

Out[23]:
```
foobar (generic function with 1 method)
```

In [24]:
```
foobar(1, 2, 3)
```

Out[24]:
```
(1, 2, 3)
```

In [25]:
```
foobar(1, 2.0, 3)
```

Out[25]:
```
(1, 2.0, 3)
```

In [26]:
```
foobar(1, 2.0, 3.0)
```

```
MethodError: no method matching foobar(::Int64, ::Float64, ::Float64)
Closest candidates are:
  foobar(::Any, ::Any, !Matched::T<:Integer) where T<:Integer at In[23]:1

Stacktrace:
 [1] top-level scope at In[26]:1
```

多型

多型，是在物件導向風格裏面很重要的特性，讓子型別可以繼承父型別的方法，方法會依據不同的子型別有不同的行為

多型擁有更廣泛的意思：**方法會依據不同的子型別有不同的行為**

也就是說，多型不只是單單放在物件導向的繼承上，只要符合 **同樣的函式會依據不同型別而有不同行為** 就算

若是依據維基百科的定義：

> *Polymorphism is the provision of a single interface to entities of different types.*
>
> *多型為不同型別的實體提供了單一介面*

說到底，多型就是為了要 ***在同樣的介面上提供不同的實作***。

參數多型

參數多型不考慮確切的型別，而是提供一種行為框架，直接定義一個函式，然後依據使用時傳入的型別做操作

Julia 本身就是採用這樣的方式。

泛型（generic programming），就是參數多型的一種表現方式

在其他語言中有這樣的 generic functions 就是參數多型的精隨了

Empty generic function

有的時候你需要定義method的介面，但不定義實作

這樣介面跟實作分離的使用情境時常用在增加程式碼的可讀性上

你可以這樣寫：

In [27]:

```
function generic  # 沒有參數，作為一個佔位符使用
end
```

Out[27]:

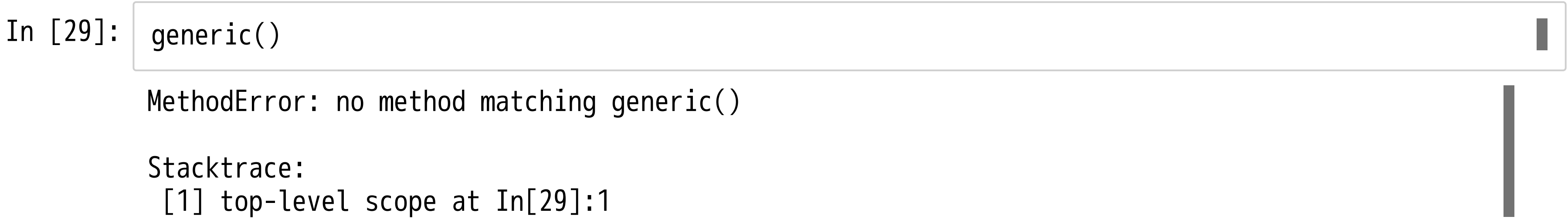
```
generic (generic function with 0 methods)
```

In [28]:

```
methods(generic)
```

Out[28]:

0 methods for generic function **generic**:



```
In [29]: generic()
```

```
MethodError: no method matching generic()

Stacktrace:
 [1] top-level scope at In[29]:1
```

Integrate with design pattern: observer pattern

In [30]:

```
abstract type Subscriber end

function update end

abstract type Publisher end

function register_subscriber! end
function remove_subscriber! end
function notify_subscriber end
```

Out[30]:

```
notify_subscriber (generic function with 0 methods)
```

In [31]:

```
struct Customer <: Subscriber
    name::String
end

update(c::Customer, x) = println("$(c.name) got $x.")

struct NewspaperPublisher <: Publisher
    subscribers::Vector{Subscriber}
    NewspaperPublisher() = new(Subscriber[])
end

function register_subscriber!(np::NewspaperPublisher, s::Subscriber)
    push!(np.subscribers, s)
end

function remove_subscriber!(np::NewspaperPublisher, s::Subscriber)
    pop!(np.subscribers, s)
end

function notify_subscriber(np::NewspaperPublisher)
    for s in np.subscribers
        update(s, "newspaper")
    end
end
```

Out[31]:

```
notify_subscriber (generic function with 1 method)
```

In [32]:

```
np = NewspaperPublisher()
```

Out[32]:

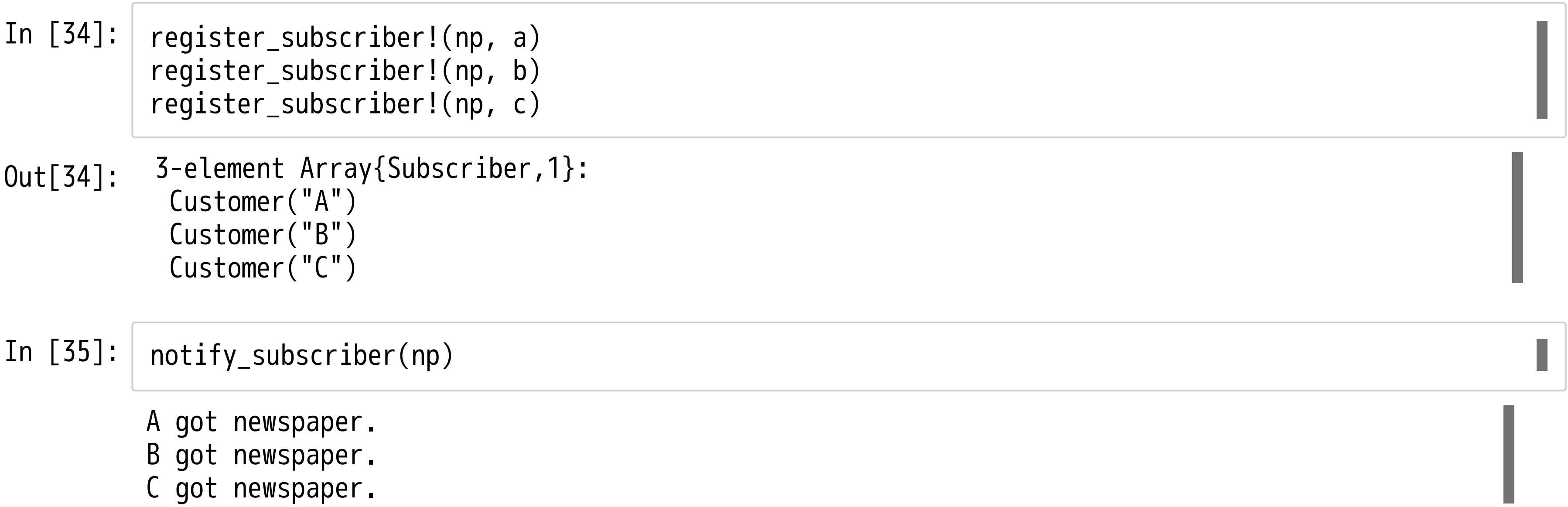
```
NewspaperPublisher(Subscriber[])
```

In [33]:

```
a = Customer("A")
b = Customer("B")
c = Customer("C")
```

Out[33]:

```
Customer("C")
```



In [34]:

```
register_subscriber!(np, a)
register_subscriber!(np, b)
register_subscriber!(np, c)
```

Out[34]:

```
3-element Array{Subscriber,1}:
 Customer("A")
 Customer("B")
 Customer("C")
```

In [35]:

```
notify_subscriber(np)
```

```
A got newspaper.
B got newspaper.
C got newspaper.
```

Q&A