2019-06-16

A Brief History of Pipeline Operator

Pipeline operator is experimentally introduced to the bleeding edge of Ruby. For some reason, this seems to lead a lot of people to leave their comments on the ticket of the bug tracker.

I investigated the history of the pipeline operator. I'm not an expert on the operator, so let me know if this article contains any wrong description.

What is pipeline operator?

This is what pipeline operator is from users' perspective:

```
x |> f |> g |> h  # identical to h(g(f(x)))
```

In the function-application expression `h(g(f(x)))`, the order of the function call (`f`→`g`→`h`) is different from the literal order in the program (`h`→`g`→`f`). This problem becomes more serious when the function names are longer:

```
wonderful_process_h(
  marvelous_process_g(
    fantastic_process_f(
      astonishing_argument_x
    )
  )
)
```

It also suffers from indentation hell.

The pipeline operator `|>` solves both problems:

```
astonishing_argument_x
|> fantastic_process_f
|> marvelous_process_g
|> wonderful_process_h
```

There are neither "order" problem nor indent problem.

Isabelle/ML

According to The Early History of F#, the first language that introduced pipeline operator is Isabelle/ML. We can read the discussion at 1994.

According to the discussion, its purpose was to avoid the above two problems: programmers want to write function names as their execution order. For example, we write the following code in ML to make AAA, add BBB to it, add CCC to it, and then add DDD to it:

```
add DDD (add CCC (add BBB (make AAA ())))
```

By using `|>`, we can write the same program as follows:

```
make AAA
|> add BBB
|> add CCC
|> add DDD
```

The operator was called "also" in the original proposal. In fact, MLs let programmers define infix operators using even alphabets, and can write `make AAA add BBB add CCC add DDD` by defining `add` infix operator based on `add` function. So,

you actually don't need another feature like "`also`", but it would avoid messing the program with a bunch of `add_XXX` infix declarations.

Here are some commits involving `also` and `|>` from Isabelle/ML repository.

- `also` operator
- An example usage of `also`
- `also` was changed to `|>`

F

F# is a language for .NET Framework. It is based on OCaml.

Pipeline operator has been popular because it was introduced by F#. "Expert F#", a book written by Don Syme who is the designer of F#, says "The |> forward pipe operator is perhaps the most important operator in F# programming."

According to The Early History of F#, pipeline operator was added to the F# standard library in 2003. The reason why `|>` is particularly important is not only because the function order is good, but also because it is easy for type inference. The type-inference algorithm of F# is left-to-right, based on information available earlier in the program. Consider the following program:

```
let data = ["one"; "three"]
data |> List.map (fun s -> s.Length)
```

In the above case, the type inference knows that `data` is `string list`, so no type annotation is required. However, if we write it in a traditional style without `|>`,

```
List.map (fun (s: string) -> s.Length) data
```

An annotation `s: string` is required because `data` follows the anonymous function. (This example is attributed to "The Early History of F#".)

Pipeline operator in F# was introduced back to OCaml in 2013. F# is based on OCaml, but with regards to pipeline operator, F# was prior to OCaml.

Three "facts" about pipeline operator in ML

I digress from history. I'd like to summarize some facts of pipeline operator in ML family.

In ML, `|>` is not special. It is not a built-in operator but is merely a normal function. This is a cool hack that functional programming lovers like. There are three facts to make the hack possible.

1. Users can define their own infix operator.
2. Currying is built-in by default.
3. Most functions accept a "primary" argument as the final argument.

1 is cool, but 2 and 3 look particularly important.

By "primary", I mean the subjective argument of a function; for example, a list for list manipulation function, an array for array manipulation function, etc. In F#, most functions are carefully designed to accept their "primary" argument at the last. For example, `List.map` accepts a closure as a first argument, a list as a second argument, and then returns a new list.

```
List.map   : ('a -> 'b) -> 'a list -> 'b list
Array.map  : ('a -> 'b) -> 'a[] -> 'b[]
Option.map : ('a -> 'b) -> 'a option -> 'b option
Seq.map    : ('a -> 'b) -> #seq<'a> -> seq<'b>
```

Thanks to these facts, we can define `|>` just as follow. Great!

```
let (|>) x f = f x
```

Pipeline operator and method chain

As I said, `|>` in F# is used for chaining functions in the execution order: `x |> f |> g |> h`.

By the way, a method chain in object-oriented programs is also used for chaining methods in the execution order: `x.f().g().h()`.

I'm not sure whether F# intentionally introduced `|>` being on aware of the above fact, or it was just a coincidence. Anyway, some people think that the two things are related.

- map - Method Chaining vs |> Pipe Operator - Stack Overflow
- FSharp: Using the Pipe Operartor to Chain Methods - ByATool
- Feature: Using dot operator instead of pipe operator · Issue #1638 · facebook/reason
- Fluent Feature in F# (slideshare, in Japanese)

Elixir

Returning to history.

Elixir introduced pipeline operator. According to José Valim, the author of Elixir, it came from F#. Its appearance is really similar to F#.

```
x |> f |> g |> h
```

However, it is quite different in terms of the language specifications. In Elixir, `|>` is not a normal operator, but a built-in language construction.

```
x |> f(args..) # identical to f(x, args...)
```

It first evaluates the left side (`x`), and passes it to the right call as the first (not last) argument.

If `|>` is a normal operator, the left and right expression are independent. Consider an addition expression `expr1 + expr2`. The two `expr`s are both self-contained expressions. In `expr1 |> expr2` of F#, `expr2` is independent. If `expr2` is a function that accepts two or more arguments, we can exploit currying and partial application. The "primary" argument that is typically passed in a pipeline is final, so the hack shines!

The right side of `|>` in Elixir is not an independent expression. It is an incomplete function call that lacks the first argument. So, Elixir's `|>` is not a normal operator; rather it is considered a language construction.

The rationale of the Elixir's design is, that the three facts are all unsatisfied in Elixir. It cannot define infix operator freely; there is no currying by default; most functions accept its "primary" argument at the first. Therefore, in my opinion, Elixir's `|>` is completely different from F#'s, though it looks similar.

When considering a new feature on a "practical" programming language, I think we have to care much about what users see on its appearance, rather than what some language maniacs say. That is, if a new operator in a language A has the same syntax as an operator in some other language B, and they work differently, it might not so important from the user of the language A.

Ruby

Finally, the development version of Ruby introduced pipeline operator only a few days ago.

```
x |> f |> g |> h # identical to x.f().g().h()
```

```
x |> f(args...) # identical to x.f(args...)
```

It first evaluates the left side. And then, it calls a method on that result. The method name and arguments are written on the right side.

Like Elixir, Ruby does not satisfy the three facts. So, Ruby introduced `|>` as a new language construction, not as a simple operator.

`|>` in Elixir passes the left expression as the first argument of the right function call. On the other hand, `|>` in Ruby passes it as a receiver of the right method call. In Ruby, the "primary" argument is typically a receiver. So this is a natural choice, in a sense.

What is this feature practically good for? It is arguable. But I'd say that it is useful to explicitly write a multi-line method chain. And we can omit parentheses.

```
(1..)
.take(10)
.each {|x| p x }

1..
|> take 10
|> each {|x| p x }
```

You might feel more comfortable with the later style. Mind you, either is fine for me.

Anyway, the feature has some problems.

- It is different from Elixir's.
- It is almost the same to the existing method call syntax (`.`).

My opinion for the first point: As far as I know, this is the first case to add pipeline operator to object-oriented language. Just saying "it is different from Elixir!" is not helpful at all because Ruby is not Elixir and Elixir is not object-oriented. We need to find a good design suitable for Ruby. Besides, most of the normal methods in Ruby accept the "primary" argument as a receiver. Function-style methods that accept the "primary" one in the first argument are relatively few. (Of course, there are some exceptions like `File.join`, `Math.log`, etc.) So, Elixir-style pipeline operator might not be very useful in Ruby.

My opinion for the second point: we don't have to hurry. If matz admits that there is no use case, he will remove the feature before it is released. The next release is planned in December.

Good news is that Matz also realizes those problems and is now considering another candidate of its name and symbol. Then, Ruby's new "pipeline" still has a chance to be a vaporware soon? I'm not sure. Keep your eyes on the development of Ruby.

Conclusion

I briefly explained the history of pipeline operator. I just investigated all on one night, so let me know if I was wrong. (I'm a non-native speaker, so don't get me wrong if the expression is not appropriate.)

Acknowledgment

Keiichiro Shikano kindly reviewed the draft of this article.

posted by Yusuke Endoh