

Contents

Collection Pipeline

Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next. (Common operations are filter, map, and reduce.) This pattern is common in functional programming, and also in object-oriented languages which have lambdas. This article describes the pattern with several examples of how to form pipelines, both to introduce the pattern to those unfamiliar with it, and to help people understand the core concepts so they can more easily take ideas from one language to another.

25 June 2015



Martin Fowler

◆ OBJECT COLLABORATION DESIGN

◆ API DESIGN

◆ RUBY

◆ LANGUAGE FEATURE

CONTENTS

First encounters

Defining Collection Pipeline

Exploring more pipelines and operations

Getting total word counts (map and reduce)

Getting the number of articles of each type (group-by)

Getting the number of articles for each tag

Alternatives

Using Loops

Using Comprehensions

Nested Operator Expressions

[Laziness](#)[Parallelism](#)[Immutability](#)[Debugging](#)[When to Use It](#)

SIDEBARS

[Smalltalk Syntax](#)[Operation Catalog](#)

The collection pipeline is one of the most common, and pleasing, patterns in software. It's something that's present on the unix command line, the better sorts of OO languages, and gets a lot of attention these days in functional languages. Different environments have slightly different forms, and common operations have different names, but once you get familiar with this pattern you don't want to be without it.

First encounters

I first came across the collection pipeline pattern when I started with Unix. For example, let's imagine I want to find all my bliki entries that mention "nosql" in the text. I can do this with grep:

```
grep -l 'nosql' bliki/entries
```

I might then want to find out how many words are in each entry

```
grep -l 'nosql' bliki/entries/* | xargs wc -w
```

and perhaps sort them by their word count

```
grep -l 'nosql' bliki/entries/* | xargs wc -w | sort -nr
```

and then just print the top 3 (removing the total)

```
grep -l 'nosql' bliki/entries/* | xargs wc -w | sort -nr | head -4 | tail -3
```

Compared with other command line environments I'd come accross before (or indeed later) this was extremely powerful.

At a later date I found the same pattern when I started using Smalltalk. Lets imagine I have a collection of article objects (in someArticles) each of which has a collection of tags and a word count. I can select only those articles that have the #nosql tag with

```
someArticles select: [ :each | each tags includes: #nosql]
```

The select method takes a single argument Lambda (defined by the square brackets, and called a "block" in smalltalk) which defines a boolean function which it applies every element in someArticles and returns a collection of only those articles for which the lambda resolves as true.

To sort the result of that code, I expand the code.

```
(someArticles
  select: [ :each | each tags includes: #nosql])
  sortBy: [:a :b | a words > b words]
```

Smalltalk Syntax

In Smalltalk, messages sent to objects are separated by whitespace, so while in most OO languages these days you'd write `anArticle.tags`, in Smalltalk you'd write `anArticle tags`. If a message takes an argument, you add a colon and the argument - hence `anArticle tags includes: #nosql`. The `#nosql` is a symbol. If you need more than one argument you add extra keywords `aList copyFrom: 1 to: 3`

To specify a lambda you use square brackets, separating the arguments with a vertical bar:

```
aList sortBy: [:a :b | a words > b words]
```

The `sortBy` method is another method that takes a lambda, this time the code used to sort the elements. Like `select` it returns a new collection so I can continue the pipeline

```
((someArticles
  select: [ :each | each tags includes: #nosql])
  sortBy: [:a :b | a words > b words])
  copyFrom: 1 to: 3
```

The core similarity to the unix pipeline is that each of the methods involved (`select`, `sortBy`, and `copyFrom`) operate on a collection of records and return a collection of records. In unix that collection is a stream with the records as lines in the stream, in Smalltalk the collection is of objects, but the basic notion is the same.

These days, I do much more programming in Ruby, where the syntax makes it nicer to set up a collection pipeline as I don't have to wrap the earlier stages of the pipeline in parentheses

```
some_articles
  .select{|a| a.tags.include?(:nosql)}
  .sort_by{|a| a.words}
  .take(3)
```

Forming a collection pipeline as a method chain is a natural approach when using an object-oriented programming language. But the same idea can be done by nested function invocations.

To go back to some basics, let's approach how you might set up a similar pipeline in common lisp. I can store each article in a structure called `articles`, which allows me to access the fields with functions named like `article-words` and `article-tags`. The function `some-articles` returns the ones I start with.

The first step is to select only the `nosql` articles.

```
(remove-if-not
  (lambda (x) (member 'nosql (article-tags x))))
  (some-articles))
```

As with the case with Smalltalk and Ruby, I use a function `remove-if-not` that takes both the list to operate on and a lambda to define the predicate. I can then expand the expression to sort them, again using a lambda

```
(sort
  (remove-if-not
    (lambda (x) (member 'nosql (article-tags x)))
    (some-articles))
  (lambda (a b) (> (article-words a) (article-words b)))))
```

I then select the top 3 with `subseq`.

```
(subseq
  (sort
    (remove-if-not
      (lambda (x) (member 'nosql (article-tags x)))
      (some-articles))
    (lambda (a b) (> (article-words a) (article-words b)))))
  0 3)
```

The pipeline is there, and you can see how it builds up pretty nicely as we go through it step by step. However it's questionable as to whether its pipeline nature is clear once you look at the final expression [\[1\]](#). The unix pipeline, and the smalltalk/ruby ones have a linear ordering of the functions that matches the order in which they execute. You can easily visualize the data starting at the top-left and working its way right and/or down through the various filters. Lisp uses nested functions, so you have to resolve the ordering by reading from deepest function up.

The popular recent lisp, Clojure, avoids this problem allowing me to write it like this.

```
(->> (articles)
      (filter #(some #{:nosql} (:tags %)))
      (sort-by :words >)
      (take 3)))
```

The `"->>"` symbol is a threading macro, which uses lisp's powerful syntactic macro capability to thread the result of each expression into an argument of the next expression. Providing you follow conventions in your libraries (such as making the subject collection the last argument in each of the transformation functions) you can use this to turn a series of nested functions into a linear pipeline.

For many functional programmers, however, using a linear approach like this isn't something important. Such programmers handle the depth ordering of nested functions just fine, which is why it took such a long time for an operator like `"->>"` to make it into a popular lisp.

These days I often hear fans of functional programming extoll the virtues of collection pipelines, saying that they are a powerful feature of functional languages that OO languages lack. As an old Smalltalker I find this rather annoying as Smalltalkers used them widely. The reason people say that collection pipelines aren't a feature of OO programming is that the popular OO languages like C++, Java, and C# didn't adopt Smalltalk's use of lambdas, and thus didn't have the rich array of collection methods that underpin the collection pipeline pattern. As a result collection pipelines died out for most OO programmers. Smalltalkers like me cursed the lack of lambdas when Java became the big thing in town, but we had to live with it. There were various attempts to build collection pipelines using what we could in Java; after all, to an OOer, a function is merely a class with one method. But the resulting code was so messy that even those familiar with the technique tended to give up. Ruby's comfortable support for collection pipelines was one of the main reasons I started using Ruby heavily around 2000. I'd missed things like that a lot from my Smalltalk days.

These days lambdas have shaken off much of their reputation for being an advanced and little-usable language feature. In mainstream language C# has had them for several years, and even Java has finally joined in. [2] So now collection pipelines are viable in many languages.



Defining Collection Pipeline

I consider Collection Pipeline to be a pattern of how we can modularize and compose software. Like most patterns, it pops up in lots of places, but looks superficially different when it does so. However if you understand the underlying pattern, it makes it easy to figure out what you want to do in a new environment.

A collection pipeline lays out a sequence of operations that pass a collection of items between them. Each operation takes a collection as an input and emits another collection (except the last operation, which may be a terminal that emits a single value). The individual operations are simple, but you can create complex behavior by stringing together the various operations, much as you might plug pipes together in the physical world, hence the pipeline metaphor.

Collection Pipeline is a special case of the Pipes and Filters pattern. The filters in Pipes and Filters correspond to the operations in Collection Pipeline, I replace "filter" with operation because filter is a common name for one of the kinds of operations in a Collection Pipeline. From another perspective, the collection pipeline is a particular, but common, case of composing higher-order functions, where the functions all act on some form of sequence data structure. There isn't an established name for this pattern, so I felt it necessary to resort to a Neologism.

The operations and the collections that are passed between the operations take different forms in various contexts.

In Unix the collection is a text file whose items are the lines in the file. Each line contains various values, separated by whitespace. The meaning of each value is given by its ordering in the line. The operations are unix processes and collections are composed using the pipeline operator with the standard output of one process piped to the standard input of the next.

In an object-oriented program the collections are a collection class (list, array, set, etc). The items in the collection are the objects within the collection, these

objects may be collections themselves or contain more collections. The operations are the various methods defined on the collection class itself - usually on some high level superclass. The operations are composed through a method chain.

In a functional language the collections are collections in a similar way to that of an object-oriented language. However the items this time are generic collection types themselves - where an OO collection pipeline would use objects, a functional language would use a hashmap. The elements of the top level collection may be collections themselves and the hashmap's elements may be collections - so like the object-oriented case we can have an arbitrarily complex hierarchical structure. The operations are functions, they may be composed either by nesting or by an operator that's capable of forming a linear representation, such as Clojure's arrow operators.

The pattern pops up in other places too. When the relational model was first defined it came with a relational algebra which you can think of as a collection pipeline where the intermediate collections are constrained to be relations. But SQL doesn't use the pipeline approach, instead using an approach that's rather like comprehensions (which I'll discuss later.)

The notion of a series of transformations like this is a common approach to structuring programs - hence the harvesting of the Pipes and Filters architectural pattern. Compilers often work this way, transforming from source code, to syntax tree, through various optimizations, and then to output code. The distinguishing thing about a collection pipeline is that the common data structure between stages is a collection, which leads to a particular set of common pipeline operations.




Exploring more pipelines and operations

The one example pipeline I've used so far just uses a few of the operations that are common in collection pipelines. So now I'll explore more operations with a few examples. I'll stick with ruby, as I'm more familiar with that language these days, but the same pipelines can usually be formed in other languages that support this pattern.

Getting total word counts (map and reduce)

```
- title: NoDBA                                5219
  words: 561
  tags: [nosql, people, orm]
  type: :bliko
- title: Infodeck
  words: 1145
  tags: [nosql, writing]
  type: :bliko
- title: OrmHate
  words: 1718
  tags: [nosql, orm]
  type: :bliko
- title: ruby
  words: 1313
  tags: [ruby]
  type: :article
- title: DDD_Aggregate
  words: 482
  tags: [nosql, ddd]
  type: :bliko
```



Two of the most important pipeline operations can be explained with a simple task - how to get a total word count for all the articles in the list. The first of these operations is map, this returns a collection whose members are the result of applying the given lambda to each element in the input collection.

```
[1, 2, 3].map{|i| i * i} # => [1, 4, 9]
```

So if we use this we can transform a list of articles into a list of the word counts for each article. At this point we can then apply one of more awkward

collection pipeline operations: reduce. This operation reduces an input collection into a single result. Often any function that does this is referred to as a reduction. Reductions often reduce to a single value, and then can only appear as the final step in a collection pipeline. The general reduce function in ruby takes a lambda which has two variables, the usual one for each element and another accumulator. At each step in the reduction it sets the value of the accumulator to result of evaluating the lambda with the new element. You can then sum a list of numbers like this

```
[1, 2, 3].reduce {|acc, each| acc + each} # => 6
```

With these two operations on the menu, calculating the total word count is a two-step pipeline.

```
some_articles  
  .map{|a| a.words}  
  .reduce {|acc, w| acc + w}
```

The first step is the map that transforms the list of articles into a list of word counts. The second step runs a reduction on the list of word counts to create a sum.

You can pass functions into pipeline operations either as lambdas or by the name of a defined function

At this point, it's worth mentioning that there are a couple of different ways that you can represent the functions that make up steps in a collection pipeline. So far, I've used a lambda for each step, but an alternative is to just use the name of the function. Writing this pipeline in clojure, the natural way to write it would be

```
(->> (articles)  
      (map :words)  
      (reduce +))
```

In this case, just the names of the relevant functions are enough. The function passed to `map` is run on each element of the input collection, and the `reduce` is run with each element and an accumulator. You can use the same style with ruby too, here `words` is a method that's defined on each object in the collection.

[3]

```
some_articles
  .map(&:words)
  .reduce(:+)
```

In general, using the name of a function is a bit shorter, but you're limited to a simple function call on each object. Using lambdas gives you a bit more flexibility, for a bit more syntax. When I program in Ruby I tend to prefer using a lambda most of the time, but if I were working in Clojure I'd be more inclined to use function names when I can. It doesn't matter greatly which way you go.

[4]

Getting the number of articles of each type (group-by)

<ul style="list-style-type: none"> - title: NoDBA words: 561 tags: [nosql, people, orm] type: :blik - title: Infodeck words: 1145 tags: [nosql, writing] type: :blik - title: OrmHate words: 1718 tags: [nosql, orm] type: :blik - title: ruby words: 1313 tags: [ruby] type: :article - title: DDD_Aggregate words: 482 		<pre>{blik: 4, article: 1}</pre>
---	---	----------------------------------

```
tags: [nosql, ddd]  
type: :blikl
```

For our next pipeline example, let's figure out how many articles there are by each type. Our output is a single hashmap whose keys are the types and values are the corresponding number of articles. [5]

To pull this off, we first need to group our list of articles by the article's type. The collection operator to work with here is a group-by operation. This operation puts the elements into a hash indexed by the result of executing its supplied code on that element. I can use this operation to divide the articles into groups based on how many tags they have.

```
some_articles  
  .group_by {|a| a.type}
```

All I need to do now is get a count of the articles in each group. On the face of it, this is a simple task for the map operation, just running a count on the number of articles. But the complication here is that I need to return two bits of data for each group: the name of the group and the count. A simpler, although connected problem is that the map example we saw earlier uses a list of values, but the output of the group-by operation is a hashmap.

*It's often useful to treat hashmaps as
lists of key-value pairs.*

This issue is a common one in collection pipelines once we've gone past the simple unix example. The collections that we may pass around are often lists, but can also be hashes. We need to easily convert between the two. The trick to doing so is to think of a hash as a list of pairs - where each pair is the key and corresponding value. Exactly how each element of a hash is represented varies from language to language, but a simple (and common) approach is to treat each hash element as a two-element array: [key, value].

Ruby does exactly this and also allows us to turn an array of pairs into a hash with the `to_h` method. So we can apply a map like this

```
some_articles
  .group_by {|a| a.type}
  .map {|pair| [pair[0], pair[1].size]}
  .to_h
```

This kind of bouncing between hashes and arrays is quite common with collection pipelines. Accessing the pair with array lookups is a bit awkward, so Ruby allows us to destructure the pair into two variables directly like this.

```
some_articles
  .group_by {|a| a.type}
  .map {|key, value| [key, value.size ]}
  .to_h
```

Destructuring is a technique that's common in functional programming languages, since they spend so much time passing around these list-of-hash data structures. Ruby's destructuring syntax is pretty minimal, but enough for this simple purpose.

Doing this in clojure is pretty much the same: [\[6\]](#)

```
(->> (articles)
      (group-by :type)
      (map (fn [[k v]] [k (count v)]))
      (into {}))
```

Getting the number of articles for each tag

- title: NoDBA		:nosql:
words: 561		:articles: 4
tags: [nosql, people, orm]		:words: 3906
type: :blik		:people:
- title: Infodeck		:articles: 1
words: 1145		:words: 561
tags: [nosql, writing]		:orm:



```

    type: :bliko                :articles: 2
- title: OrmHate                :words: 2279
  words: 1718                   :writing:
  tags: [nosql, orm]           :articles: 1
  type: :bliko                :words: 1145
- title: ruby                   :ruby:
  words: 1313                   :articles: 1
  tags: [ruby]                 :words: 1313
  type: :article               :ddd:
- title: DDD_Aggregate          :articles: 1
  words: 482                    :words: 482
  tags: [nosql, ddd]
  type: :bliko

```

For the next pipeline, we'll produce article and word counts for each tag mentioned in the list. Doing this involves a considerable reorganization of the collection's data structure. At the moment our top level item is an article, which may contain many tags. To do this we need to unravel the data structure so our top level is a tag that contains multiple articles. One way of thinking about this is that we're inverting a many-to-many relationship, so that the tag is the aggregating element rather than the article.

This example inverts a many-to-many relationship

This reorganizing of the hierarchical structure of the collection that starts the pipeline makes for a more complicated pipeline, but is still well within the capabilities of this pattern. With something like this, it's important to break it down into small steps. Transformations like this are usually much easier to reason about when you break the full transformation down into little pieces and string them together - which is the whole point of the collection pipeline pattern.

The first step is to focus on the tags, exploding the data structure so that we have one record for each tag-article combination. I think of this is rather like how you represent a many-to-many relationship in a relational database by

using an association table. To do this I create a lambda that takes an article and emits a pair (two element array) for each tag and the article. I then map this lambda across all of the articles.

```
some_articles
  .map {|a| a.tags.map{|tag| [tag, a]}}
```

which yields a structure like this:

```
[
  [[:nosql, Article(NoDBA)]
   [:people, Article(NoDBA)]
   [:orm, Article(NoDBA)]]
  [[:nosql, Article(Infodeck)]
   [:writing, Article(Infodeck)]]
  # more rows of articles
]
```

The result of the map is a list of lists of pairs, with one nested list for each article. That nested list is in the way, so I flatten it out using the flatten operation.

```
some_articles
  .map {|a| a.tags.map{|tag| [tag, a]}}
  .flatten 1
```

yielding

```
[
  [:nosql, Article(NoDBA)]
  [:people, Article(NoDBA)]
  [:orm, Article(NoDBA)]
  [:nosql, Article(Infodeck)]
  [:writing, Article(Infodeck)]
  # more rows of articles
]
```

This task of generating a list with unnecessary level of nesting that needs to be flattened out is so common that most languages provide the flat-map operation to do this in a single step.

```
some_articles
  .flat_map {|a| a.tags.map{|tag| [tag, a]}}
```

Once we have a list of pairs like this, then it's a simple task to then group it by the tag

```
some_articles
  .flat_map {|a| a.tags.map{|tag| [tag, a]}}
  .group_by {|pair| pair.first}
```

yielding

```
{
  :people:
    [ [:people, Article(NoDBA)] ]
  :orm:
    [ [:orm, Article(NoDBA)]
      [:orm, Article(OrmHate)]
    ]
  :writing:
    [ [:writing, Article(Infodeck)] ]
  # more records
}
```

But like with our first step, this introduces an annoying extra level of nesting, because the value of each association is a list of key/article pairs rather than just a list of articles. I can trim this out by mapping a function to replace the list of pairs with a list of articles.

```
some_articles
  .flat_map {|a| a.tags.map{|tag| [tag, a]}}
  .group_by {|pair| pair.first}
  .map {|k,pairs| [k, pairs.map {|p| p.last}]}
```

this yields

```
{
  :people:    [ Article(NoDBA) ]
  :orm:       [ Article(NoDBA), Article(OrmHate) ]
  :writing:   [ Article(Infodeck) ]
  # more records
}
```

Now I've reorganized the basic data to articles for each tag, reversing the many-to-many relationship. To produce the required results all I need is a simple map to extract the exact data I need

```
some_articles
  .flat_map {|a| a.tags.map{|tag| [tag, a]}}
  .group_by {|pair| pair.first}
  .map {|k,pairs| [k, pairs.map {|p| p.last}]}
  .map {|k,v| [k, {articles: v.size, words: v.map(&:words).reduce(:+)}]}
  .to_h
```

This yields the final data structure which is a hash of hashes.

```
:nosql:
  :articles: 4
  :words: 3906
:people:
  :articles: 1
  :words: 561
:orm:
  :articles: 2
  :words: 2279
:writing:
  :articles: 1
  :words: 1145
:ruby:
  :articles: 1
  :words: 1313
:ddd:
  :articles: 1
  :words: 482
```

Doing the same task in Clojure takes the same form.

```
(->> (articles))
```

```

    (mapcat #(map (fn [tag] [tag %]) (:tags %)))
    (group-by first)
    (map (fn [[k v]] [k (map last v)]))
    (map (fn [[k v]] {k {:articles (count v), :words (reduce + (map :words v))}})
    (into {}))

```

Clojure's flat-map operation is called *mapcat*.

Building up a more complicated pipeline like this can be more of a struggle than the simple ones I've shown earlier. I find it's easiest to carefully do each step at a time, looking carefully at the output collection from each step to ensure it's in the right shape. Visualizing this shape usually requires some form of pretty-printing to display the collection's structure with indentation. It's also useful to do this in a rolling test-first style, writing the test initially with some simple assertion for the shape of the data (such as just the number of records for the first step) and evolving the test as I add extra stages to the pipeline.

The pipeline I have here makes sense of building it up from each stage, but the final pipeline doesn't reveal too clearly what's going on. The first stages are really all about indexing the list of articles by each tag, so I think it reads better by extracting that task into its own function.

```

(defn index-by [f, seq]
  (->> seq
    (mapcat #(map (fn [key] [key %]) (f %)))
    (group-by first)
    (map (fn [[k v]] [k (map last v)]))))
(defn total-words [articles]
  (reduce + (map :words articles)))

(->> (articles)
  (index-by :tags)
  (map (fn [[k v]] {k {:articles (count v), :words (total-words v)}}))
  (into {}))

```

I also felt it was worth factoring the word count into its own function. The factoring adds to the line-count, but I think I'm always happy to add some

structuring code if it makes it easier to understand. Terse, powerful code is nice - but terseness is only valuable in the service of clarity.

To do this same factoring in an object-oriented language like Ruby, I need to add the new `index_by` method to the collection class itself, since I can only use the collection's own methods in the pipeline. With Ruby I can monkey-patch `Array` to do this

```
class Array
  def invert_index_by &proc
    flat_map {|e| proc.call(e).map {|key| [key, e]}}
      .group_by {|pair| pair.first}
      .map {|k,pairs| [k, pairs.map {|p| p.last}]}
  end
end
```

I've changed the name here because the simple name "`index_by`" makes sense in the context of a local function, but doesn't make so much sense as a generic method on a collection class. Needing to put methods on the collection class can be a serious downside of the OO approach. Some platforms don't allow you to add methods to a library class at all, which rules out this kind of factoring. Others allow you to modify the class with monkey patching like this, but this causes a globally visible change to the class's API, so you have to think more carefully about it than a local function. The best option here is to use mechanisms like C#'s extension methods or Ruby's refinements that allow you to change an existing class, but only in the context of a smaller namespace. But even then there is a lot of ceremony to add the monkey-patch compared to the simple act of defining a function.

Once I have that method defined, I can factor the pipeline in a similar way to the clojure example.

```
total_words = -> (a) {a.map(&:words).reduce(:+)}
some_articles
  .invert_index_by {|a| a.tags}
  .map {|k,v| [k, {articles: v.size, words: total_words.call(v)}]}
  .to_h
```

Here I also factored out the word counting function like I did for the Clojure case, but I find the factoring less effective in ruby since I have to use an explicit method to call the function I created. It's not much, but it does add a bit of friction to the readability. I could make it a full method, of course, that would get rid of the call syntax. But I'm tempted to go a bit further here and add a class to contain the summary functions.

```
class ArticleSummary
  def initialize articles
    @articles = articles
  end
  def size
    @articles.size
  end
  def total_words
    @articles.map{|a| a.words}.reduce(:+)
  end
end
```

Using it like this

```
some_articles
  .invert_index_by {|a| a.tags}
  .map {|k,v| [k, ArticleSummary.new(v)]}
  .map {|k,a| [k, {articles: a.size, words: a.total_words}]}
  .to_h
```

Many people would feel it too heavyweight to introduce a whole new class just to factor out a couple of functions in a single usage. I have no trouble introducing a class for some localized work like this. In this particular case I wouldn't, since it's really only the total words function that needs extracting, but I'd only need a little bit more in the output to reach for that class.



Alternatives

The collection pipeline pattern isn't the only way to accomplish the kinds of things I've talked about so far. The most obvious alternative is what most people would have usually used in these cases: the simple loop.

Using Loops

I'll compare ruby versions of the top 3 NoSQL articles.

Collection Pipeline

```
some_articles
  .select{|a| a.tags.include?(:nosql)}
  .sort_by{|a| a.words}
  .take(3)
```

Loop

```
result = []
some_articles.each do |a|
  result << a if a.tags.include?(:nosql)
end
result.sort_by!(&:words)
return result[0..2]
```

The collection pipeline version is slightly shorter, and to my eyes clearer, primarily because the pipeline notion is one that's familiar and naturally clear to me. That said, the loop version isn't that much worse.

Here's the word count case.

Collection Pipeline

```
some_articles
  .map{|a| a.words}
  .reduce {|acc, w| acc + w}
```

Loop

```
result = 0
some_articles.each do |a|
  result += a.words
end
return result
```

The Group case

Collection Pipeline

Loop

Collection Pipeline**Loop**

```
some_articles
  .group_by {|a| a.type}
  .map {|pair| [pair[0], pair[1].size]}
  .to_h
```

```
result = Hash.new(0)
some_articles.each do |a|
  result[a.type] += 1
end
return result
```

The article count by tag

Collection Pipeline

```
some_articles
  .flat_map {|a| a.tags.map{|tag| [tag, a]}}
  .group_by {|pair| pair.first}
  .map {|k,pairs| [k, pairs.map {|p| p.last}]}
  .map {|k,v| [k, {articles: v.size, words: v.map(&:words).reduce(:+)}]}
  .to_h
```

res
som
a

e
end
res
w
v

e
r
end
ret

In this case the collection pipeline version is much shorter, although it's tricky to compare since in either case I'd refactor to bring out the intention in either case.

Using Comprehensions

Some languages have a construct for comprehensions, usually called list comprehensions, which mirror simple collection pipelines. Consider retrieving the titles of all articles that are longer than a thousand words. I'll illustrate this

with coffeescript which has a comprehension syntax, but can also use Javascript's own ability to create collection pipelines.

Pipeline

```
some_articles
  .filter (a) ->
    a.words > 1000
  .map (a) ->
    a.title
```

Comprehension

```
(a.title for a in some_articles where
```

The exact capabilities of a comprehension differ from language to language, but you can think of them as particular sequence of operations that can be expressed in a single statement. This way of thinking of them illuminates the first part of the decision of when to use them. Comprehensions can only be used for certain combinations of pipeline operations, so they are fundamentally less flexible. That said, comprehensions are defined for the most common cases, so they are still an option in many cases.

Comprehensions can usually be placed in a pipeline themselves - essentially acting as a single operation. So to get the total word count of all articles over 1000 words I could use:

Pipeline

```
some_articles
  .filter (a) ->
    a.words > 1000
  .map (a) ->
    a.words
  .reduce (x, y) -> x + y
```

Comprehension in a pipeline

```
(a.words for a in some_articles where
  .reduce (x, y) -> x + y
```

The question then is whether comprehensions are better than pipelines for the cases that they work in. Fans of comprehensions say they are, others might say that pipelines are just as easy to understand and more general. (I fall into the latter group.)



Nested Operator Expressions

One of useful things you can do with collections is manipulate them with set operations. So lets assume I'm looking at a hotel with functions to return rooms that are red, blue, at the front of the hotel, or occupied. I can then use an expression to find the unoccupied red or blue rooms at the front of the hotel.

ruby...

```
(front & (red | blue)) - occupied
```

clojure...

```
(difference  
  (intersection  
    (union reds blues)  
    fronts)  
  occ)
```

| Clojure defines set operations on its set datatype, so all the symbols here are sets.

I can formulate these expressions as collection pipelines

ruby...

```
red  
  .union(blue)  
  .intersect(front)  
  .diff(occupied)
```

| I monkey-patched Array to add the set operations as regular methods

clojure...

```
(->> reds  
  (union blues)  
  (intersection fronts)  
  (remove occ))
```

I need clojure's 'remove' method here in order to get the arguments in the right order for threading.

But I prefer the nested operator expression forms, particularly when I can use infix operators. And more complicated expressions could get really tangled as pipes.

That said, it's often useful to throw a set operation in the middle of a pipeline. Let's imagine the case where the color and location of a room are attributes of the room record, but the list of occupied rooms is in a separate collection.

ruby...

```
rooms
  .select{|r| [:red, :blue].include? r.color}
  .select{|r| :front == r.location}
  .diff(occupied)
```

clojure...

```
(->> (rooms)
      (filter #( #{:blue :red} (:color %)))
      (filter #( #{:front} (:location %)))
      (remove (set (occupied)))))
```

Here I'm showing (set (occupied)) to show how we'd use a set wrapped over a collection as a predicate for the set membership in clojure.

While infix operators are good for nested operator expressions, they don't work well with pipelines, forcing some annoying parentheses.

ruby...

```
((rooms
  .select{|r| [:red, :blue].include? r.color}
  .select{|r| :front == r.location}
) - occupied)
.map(&:num)
.sort
```

Another point to bear in mind with set operations is that collections are usually lists which are ordered and allow duplicates. You have to look at the

particulars of your library to see what means for set operations. Clojure forces you to turn your lists into sets before using set operations on them. Ruby will accept any array into its set operators but removes duplicates on its output while preserving the input ordering.

Laziness

The concept of laziness came from the functional programming world. The motivation may be some code like this:

```
large_list
  .map{|e| slow_complex_method (e)}
  .take(5)
```

With such code, you would spend a lot of time evaluating `slow_complex_method` on lots of elements and then throw away all the results except the top 5.

Laziness allows the underlying platform to determine that you only need the top five results, and then only to perform `slow_complex_method` on the ones that are needed.

Indeed this goes further into the runtime usage, let's imagine the result of `slow_complex_method` is piped into a scrolling list on a UI. A lazy pipeline would only invoke the pipeline on elements as the final results scroll into view.

For a collection pipeline to be lazy, the collection pipeline functions have to be built with laziness in mind. Some languages, commonly functional languages like Clojure and Haskell, do this right from the start. In other cases laziness can be built into a special group of collection classes - Java and Ruby have some lazy collection implementations.

Some pipeline operations cannot work with laziness and have to evaluate the whole list. Sorting is one example, without the entire list you cannot determine even a single top value. Platforms that take laziness seriously will usually document operations that are unable to preserve laziness.

Parallelism

Many of the pipeline operations naturally work well with parallel invocation. If I use map the results of using it for one element don't depend on any of the other elements in the collection. So if I'm running on a platform with multiple cores, I can take advantage of that by distributing the map evaluations across multiple threads.

Many platforms include the ability to distribute evaluations in parallel like this. If you're running a complex function over a large set, this can result in a significant performance boost by taking advantage of multicore processors.

Parallelizing, however, doesn't always boost performance. Sometimes it takes more time to set up the parallel distribution than it you gain the from the parallelism. As a result most platforms offer alternative operations that explicitly use parallelism, such as how Clojure's `pmap` function is a parallel version of `map`. As with any performance optimization, you should use performance tests to verify whether using a parallelizing operation actually provides any performance improvement.

Immutability

Collection-pipelines naturally lend themselves to immutable data structures. When building a pipeline it's natural to consider each operation as generating a new collection for its output. Done naively this involves a lot of copying, which can lead to problems with large amounts of data. However, most of time, it's not a problem in practice. Usually it's rather smaller sets of pointers that are copied rather than large hunks of data.

When it does become a problem then you can retain immutability with data structures that are designed to be transformed in this way. Functional programming languages tend to use data structures that can efficiently be manipulated in this style.

If necessary you can sacrifice mutability by using operations that update a collection rather than replacing it. Libraries in non-functional languages often offer destructive versions of the collection pipeline operators. I would strongly advise that you only use these as part of a disciplined performance tuning exercise. Start working with the non-mutating operations and only use something else when you have a known performance bottleneck in the pipeline.



Debugging

I've had a couple of questions about the difficulty of debugging a collection pipeline. Consider a pipeline that looks like this one in ruby:

```
def get_readers_of_books1(readers, books, date)
  data = @data_service.get_books_read_on(date)
  return data
```

```

      .select{|k,v| readers.include?(k)}
      .select{|k,v| !(books & v).empty?}
      .keys
end

```

Modern IDEs can do a lot to help debugging but let's imagine I'm doing this in ruby, with just emacs and I scoff at debuggers.

I might want to extract an intermediate variable half-way through the pipe

```

def get_readers_of_books2(readers, books, date)
  data = @data_service.get_books_read_on(date)
  temp = data
  .select{|k,v| readers.include?(k)}
  .select{|k,v| !(books & v).empty?}
  pp temp
  return temp
  .keys
end

```

Another, somewhat tricky, thing to do is to smuggle a print statement into a map in the pipeline.

```

def get_readers_of_books(readers, books, date)
  data = @data_service.get_books_read_on(date)
  return data
  .select{|k,v| readers.include?(k)}
  .select{|k,v| !(books & v).empty?}
  .map {|e| pp e; e}.to_h
  .keys
end

```

| In this case I need to convert back to a hash after the map operation

This does require some side-effects inside the pipeline, which should get you told off if it escaped to a code review, but it can help visualize what the code is doing.

With a proper debugger, you can usually evaluate expressions in the debugger. This way you set a breakpoint in the pipe and then evaluate expressions based

on parts of the pipe to see what's going on.

When to Use It

I see Collection Pipeline as a pattern, and with any pattern there are times you should use it, and times when you should take another route. I always get suspicious if I can't think of reasons not to use a pattern I like.

The first indication to avoid it is when the language support isn't there. When I started with Java, I missed being able to use collection pipelines a lot, so like many others I experimented with making objects that could form the pattern. You can form pipeline operations by making classes and using things like anonymous inner classes to get close to lambdas. But the problem is the that syntax is just too messy, overwhelming the clarity that makes collection pipelines so effective. So I gave up and used loops instead. Since then various functional-style libraries have appeared in java, many using annotations which weren't in the language in the early days. But my feeling remains that without good language support for clean lambda expressions, this pattern usually ends up being more trouble than is worth it.

Another argument against is when you have comprehensions. In that case comprehensions are often easier to work with for simple expressions, but you still need pipelines for their greater flexibility. Personally I find simple pipelines as easy to understand as comprehensions, but that's the kind of thing a team has to decide in its coding style.

*extract a method whenever there's a
difference between **what** a block of code
does and **how** it does it*

Even in languages that are suitable, you can run into a different limit - the size and complexity of pipelines. The ones I've shown in this article are small and linear. My general habit is to write small functions, I get twitchy if they go over half-a-dozen lines, and similar rules are there for pipelines. Larger pipelines need to be factored into separate methods, following my usual rule: extract a method whenever there a difference between what a block of code does and how it does it.

Pipelines work best when they are linear, each step has a single collection input and single output. It is possible to fork to separate inputs and outputs, although I've not put any such examples together in this article. Again, however, beware of this - factoring into separate functions is usually the key to keeping any longer behavior under control.

That said, collection pipelines are a great pattern, one that all programmers should be aware of, particularly in languages like Ruby and Clojure that support them so well. They can clearly capture what otherwise requires long and gnarly loops, and can help make code more readable and thus cheaper and easier to enhance.



Operation Catalog

Here is a catalog of the operations that you often find in collection pipelines. Every language makes different choices on what operations are available and what they are called, but I've tried to look at them through their common capabilities.

collect

Alternative name for **map**, from Smalltalk. Java 8 uses "collect" for a completely different purpose: a terminal that collects elements from a stream into a collection.

see map

concat



Concatenates collections into a single collection

more...

difference



Remove the contents of the supplied list from the pipeline

more...

distinct



Removes duplicate elements

more...

drop

A form of **slice** that returns all but the first *n* elements

see slice

filter



Runs a boolean function on each element and only puts those that pass into the output.

[more...](#)

flat-map



Map a function over a collection and flatten the result by one-level

[more...](#)

flatten



Removes nesting from a collection

[more...](#)

fold

*Alternative name for **reduce** Sometimes seen as foldl (fold-left) and foldr (fold-right).*

see [reduce](#)

group-by



Runs a function on each element and groups the elements by the result.

more...

inject

Alternative name for **reduce**, from Smalltalk's *inject:into:* selector.

see reduce

intersection



Retains elements that are also in the supplied collection

more...

map



Applies given function to each element of input and puts result in output

more...

mapcat

Alternative name for **flat-map**

see [flat-map](#)

reduce



Uses the supplied function to combine the input elements, often to a single output value

[more...](#)

reject

Inverse of **filter**, returning elements that do not match the predicate.

see [filter](#)

select

Alternative name for **filter**.

see [filter](#)

slice



Return a sub-sequence of the list between the given first and last positions.

[more...](#)

sort



Output is sorted copy of input based on supplied comparator

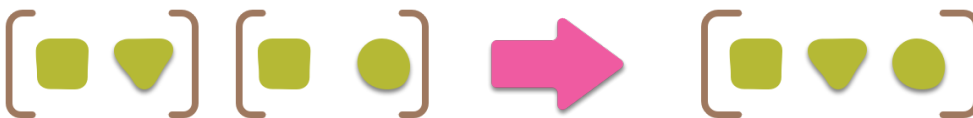
more...

take

*A form of **slice** that returns the first n elements*

see slice

union



returns elements in this or the supplied collection, removing duplicates

more...



Footnotes

1: A more idiomatic lisp pipeline

One issue here is that the lisp example isn't that idiomatic, since it's common to use named functions (easily referenced using the #'some-function syntax) - creating small functions for particular cases as you need them. This might be a better factoring of that example.

```
(defun nosqlp (article)
  (member 'nosql (article-tags article)))

(subseq
 (sort
  (remove-if-not #'nosqlp (some-articles))
  #'< :key #'article-words)
 0 3)
```

2: Java Pipeline

Here's the initial pipeline in Java

```
articles.stream()
  .filter(a -> a.getTags().contains("nosql"))
  .sorted(Comparator.comparing(Article::getWords).reversed())
  .limit(3)
  .collect(toList());
```

As you might expect, Java manages to be extra verbose in several respects. A particular feature of collection pipelines in Java is that the pipeline functions aren't defined on a collection class, but on the Stream class (which is different to IO streams). So you have to convert the articles collection into a stream at the beginning and back to a list at the end.

3: Rather than passing a block to a ruby method, you can pass a named function by preceding its name (which is a symbol) with "&" - hence &:words. With reduce, however there is an exception, you can just pass it a function's name, so you don't need the "&". I'm more likely to use a function name with reduce, so I appreciate the inconsistency.

4: Using a lambda or a function name

There's an interesting language history in the choice between using lambdas and function names, at least from my dabblers perspective. Smalltalk extended its minimal syntax for lambdas, making them easy to write, while calling a literal method was more awkward. Lisp, however, made it easy to call a named function, but required extra syntax to use a lambda - often leading to a macro to massage that syntax away.

Modern languages try to make both easy - both Ruby and Clojure make either calling a function or using a lambda pretty simple.

5: There is a polyseme here as "map" may refer to the operation map or to the data structure. For this article I'm going to use "hashmap" or "dictionary" for the data structure and only use "map" for the function. But in general conversation you'll often hear hashmaps referred to as maps.

6: Using Juxt

One option in clojure is to run multiple functions inside a map using juxt:

```
(->> (articles)
      (group-by :type)
      (map (juxt first (comp count second)))
      (into {}))
```

I find the version using a lambda to be clearer, but then I'm only a dabbler in Clojure (or functional programming in general).

Acknowledgements

My thanks to my colleagues who commented on an early draft of this article: Sriram Narayanan, David Johnston, Badrinath Janakiraman, John Pradeep, Peter Gillard-Moss, Ola Bini, Manoj Mahalingam, Jim Arnold, Hugo Corbucci, Jonathan Reyes, Max Lincoln, Piyush Srivastava, and Rebecca Parsons.

► Significant Revisions