내용

컬렉션 파이프 라인

컬렉션 파이프 라인은 컬렉션을 한 작업의 출력으로 가져와 다음 작업에 공급하여 구성하는 일련의 작업으로 계산을 구성하는 프로그래밍 패턴입니다. (일반적인 작업은 필터, 매핑 및 축소입니다.)이 패턴은 함수형 프로그래밍과 람다가있는 객체 지향 언어에서도 일반적입니다. 이 기사에서는 파이프 라인을 형성하는 방법에 대한 몇 가지 예를 통해 패턴을 설명하고 익숙하지 않은 사람들에게 패턴을 소개하고 사람들이 한 언어에서 다른 언어로 아이디어를 더 쉽게 가져올 수 있도록 핵심 개념을 이해하도록 돕습니다.

2015 년 6 월 25 일

마틴 파울러

개체 협업 디자인

API 설계

루비

언어 기능

내용

수집 파이프 라인은 소프트웨어에서 가장 일반적이고 유쾌한 패턴 중 하나입니다. 그것은 유닉스 명령 줄에 존재하는 것, 더 나은 종류의 OO 언어이며, 요즘 기능 언어에서 많은 관심을 받고 있습니다. 환경마다 형태가 약간 다르고 일반적인 작업의 이름도 다르지만이 패턴에 익숙해지면 패턴 없이는 싶지 않습니다.

첫 만남

저는 Unix로 시작했을 때 처음으로 컬렉션 파이프 라인 패턴을 발견했습니다. 예를 들어, 텍스트에서 "nosql"을 언급하는 모든 bliki 항목을 찾고 싶다고 가정 해 보겠습니다. 나는 grep으로 이것을 할 수있다.

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

그런 다음 각 항목에 몇 개의 단어가 있는지 알고 싶을 수 있습니다.

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

단어 수를 기준으로 정렬 할 수 있습니다.

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

그런 다음 상단 3을 인쇄하십시오 (총계 제거)

```
grep -l 'nosql'bliki / entries / * | xargs wc -w | 정렬 -nr / 머리 -4 / 꼬리 -3
```

이전에 (또는 나중에) 건너온 다른 명령 줄 환경과 비교할 때 이것은 매우 강력했습니다.

나중에 Smalltalk를 사용하기 시작했을 때 동일한 패턴을 발견했습니다. someArticles각각 태그 모음과 단어 개수가있는 기사 객체 모음이 있다고 가정 해 보겠습니다 . #nosql태그 가있는 기사 만 선택할 수 있습니다.

```
someArticles 선택 : [: each | 각 태그에는 다음이 포함됩니다. #nosql]
```

이 select메서드는 someArticles의 모든 요소를 적용하고 람다가 true로 확인되는 아티클 만 컬렉션을 반환하는 부울 함수를 정의 하는 단일 인수 Lambda (대괄호로 정의되고 smalltalk에서 "블록"이라고 함)를 사용합니다. .

해당 코드의 결과를 정렬하기 위해 코드를 확장합니다.

```
( someArticles
      선택 : [: 각 | 각 태그에는 다음이 포함됩니다. #nosql] )
      sortBy : [: a : b | a 단어> b 단어]
```

스몰 토크 구문

스몰 토크에서 객체로 전송 된 메시지는 공백으로 구분되므로 요즘 대부분의 OO 언어 anArticle.tags에서는를 작성 하지만 스몰 토크에서는 anArticle tags. 메시지가 인수를 받으면 콜론과 인수를 추가합니다 anArticle tags includes: #nosql. 는 #nosql상징이다. 둘 이상의 인수가 필요한 경우 추가 키워드를 추가합니다.aList copyFrom: 1 to: 3

람다를 지정하려면 대괄호를 사용하여 세로 막대로 인수를 구분합니다. aList sortBy: [:a :b | a words > b words]

이 sortBy메서드는 람다를 사용하는 또 다른 메서드이며 이번에는 요소를 정렬하는 데 사용되는 코드입니다. select파이프 라인을 계속할 수 있도록 새 컬렉션을 반환하는 것처럼

```
( (일부 기사
      선택 : [: 각 | 각 태그에는 다음이 포함됩니다. #nosql])
      sortBy : [: a : b | a 단어> b 단어] )
      copyFrom : 1 ~ : 3
```

유닉스 파이프 라인 코어 유사성 방법들의 각각은 관련된 (즉 select, sortBy한 copyFrom레코드의 집합에서 동작하고 레코드들의 집합을 리턴). 유닉스에서 컬렉션은 스트림의 라인으로 레코드가있는 스트림이고, 스몰 토크에서 컬렉션은 객체이지만 기본 개념은 동일합니다.

요즘에는 Ruby로 훨씬 더 많은 프로그래밍을합니다. 구문을 사용하면 파이프 라인의 초기 단계를 괄호로 묶을 필요가 없기 때문에 컬렉션 파이프 라인을 설정하는 것이 더 좋습니다.

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

컬렉션 파이프 라인을 메서드 체인으로 형성하는 것은 객체 지향 프로그래밍 언어를 사용할 때 자연스러운 접근 방식입니다. 그러나 중첩 된 함수 호출로 동일한 아이디어를 수행 할 수 있습니다.

몇 가지 기본 사항으로 돌아가려면 common lisp에서 유사한 파이프 라인을 설정하는 방법에 접근 해 보겠습니다. 나는라는 구조에서 각 기사 저장할 수있는 articles 액세스와 같은 이름 지정된 함수와 필드 나를 수, article-words과 article-tags. 이 함수 some-articles 는 내가 시작한 것을 반환합니다.

첫 번째 단계는 nosql 아티클 만 선택하는 것입니다.

```
(아니면 제거
   (lambda (x) (멤버 'nosql (article-tags x)))
   (일부 기사))
```

스몰 토크와 루비의 경우와 마찬가지로, 나는 remove-if-not목록에서 작동하는 것과 술어를 정의하기 위해 람다를 모두 취하는 함수 를 사용합니다 . 그런 다음 다시 람다를 사용하여 표현식을 확장하여 정렬 할 수 있습니다.

```
(종류
   (아니면 제거
      (lambda (x) (멤버 'nosql (article-tags x)))
      (일부 기사))
   (lambda (ab) (> (article-words a) (article-words b)))))
```

그런 다음으로 상위 3 개를 선택합니다 subseq.

```
(subseq
   (종류
      (아니면 제거
         (lambda (x) (멤버 'nosql (article-tags x)))
         (일부 기사))
      (lambda (ab) (> (article-words a) (article-words b)))))
 0 3)
```

파이프 라인이 거기에 있으며 단계별로 진행하면서 얼마나 잘 구축되는지 볼 수 있습니다. 그러나 최종 표현식 [1] 을 살펴보면 파이프 라인의 성격이 분명한지는 의문입니다. 유닉스 파이프 라인과 스몰 토크 / 루비 파이프 라인은 실행 순서와 일치하는 함수의 선형 순서를 가지고 있습니다. 다양한 필터를 통해 왼쪽 상단에서 시작하여 오른쪽 및 / 또는 아래쪽으로 데이터를 쉽게 시각화 할 수 있습니다. Lisp는 중첩된 함수를 사용하므로 가장 깊은 함수에서 위로 읽어 순서를 해결해야합니다.

최근의 인기있는 lisp 인 Clojure는이 문제를 피하여 이렇게 작성할 수 있습니다.

```
(->> (기사)
     (필터 # (일부 # {: nosql} (: 태그 %)))
     (정렬 기준 : 단어>)
     (3))
```

"->>"기호는 lisp의 강력한 구문 매크로 기능을 사용하여 각 표현식의 결과를 다음 표현식의 인수로 스레딩하는 스레딩 매크로입니다. 라이브러리의 규칙 (예 : 주제 컬렉션을 각 변환 함수의 마지막 인수로 설정)을 따르도록 제공하면이를 사용하여 일련의 중첩 함수를 선형 파이프 라인으로 바꿀 수 있습니다.

그러나 많은 기능 프로그래머에게 이와 같은 선형 접근 방식을 사용하는 것은 중요하지 않습니다. 이러한 프로그래머는 중첩 함수의 깊이 순서를 잘 처리하므로 "->>"와 같은 연산자가 인기있는 lisp로 만드는 데 시간이 너무 오래 걸렸습니다.

요즘에는 함수형 프로그래밍 팬들이 컬렉션 파이프 라인의 장점을 OO 언어가 부족한 함수형 언어의 강력한 기능이라고 말하는 것을 자주 듣습니다. 오래된 스몰 토커로서 저는 스몰 토커가 널리 사용했기 때문에 이것이 다소 짜증나는 것을 발견했습니다. 사람들이 컬렉션 파이프 라인이 OO 프로그래밍의 기능이 아니라고 말하는 이유는 C ++, Java 및 C #과 같은 인기있는 OO 언어가 Smalltalk의 람다 사용을 채택하지 않았기 때문에 수집 방법이 풍부하지 않기 때문입니다. 수집 파이프 라인 패턴을 뒷받침합니다. 결과적으로 수집 파이프 라인은 대부분의 OO 프로그래머에게 없어졌습니다. 저와 같은 스몰 토커들은 자바가 마을에서 큰 인기를 얻었을 때 람다 부족을 저주했지만 우리는 그와 함께 살아야했습니다. Java에서 할 수있는 것을 사용하여 컬렉션 파이프 라인을 구축하려는 다양한 시도가있었습니다. 결국 OOer에게 함수는 하나의 메서드를 가진 클래스 일뿐입니다. 그러나 결과 코드는 너무 지저분하여 기술에 익숙한 사람조차도 포기하는 경향이 있습니다. 컬렉션 파이프 라인에 대한 Ruby의 편안한 지원은 제가 Ruby를 2000 년경에 많이 사용하기 시작한 주된 이유 중 하나였습니다. 저는 Smalltalk 시절에 그런 것들을 많이 놓쳤습니다.

요즘 람다는 고급 및 거의 사용할 수없는 언어 기능이라는 명성을 크게 흔들어 놓았습니다. 주류 언어에서 C #은 몇 년 동안 그것들을 가지고 있었고 심지어 Java도 마침내 합류했습니다. [2] 이제 컬렉션 파이프 라인은 여러 언어로 실행 가능합니다.

수집 파이프 라인 정의

Collection Pipeline은 소프트웨어를 모듈화하고 구성하는 방법의 패턴이라고 생각합니다. 대부분의 패턴과 마찬가지로 많은 곳에서 튀어 나오지만 그렇게되면 표면적으로 다르게 보입니다. 그러나 기본 패턴을 이해하면 새로운 환경에서 수행 할 작업을 쉽게 파악할 수 있습니다.

컬렉션 파이프 라인은 항목 컬렉션을 전달하는 일련의 작업을 배치합니다. 각 작업은 컬렉션을 입력으로 사용하고 다른 컬렉션을 내 보냅니다 (단일 값을 내보내는 터미널 일 수있는 마지막 작업 제외). 개별 작업은 간단하지만 다양한 작업을 함께 묶

어 복잡한 동작을 생성 할 수 있습니다. 마치 실제 세계에서 파이프를 연결하는 것처럼 파이프 라인 은유가됩니다.

컬렉션 파이프 라인은 파이프 및 필터 패턴 의 특수한 경우입니다 . 파이프 및 필터의 필터는 컬렉션 파이프 라인의 작업에 해당합니다. 필터는 컬렉션 파이프 라인의 작업 종류 중 하나에 대한 공통 이름이기 때문에 "필터"를 작업으로 바꿉니다. 다른 관점에서 볼 때, 컬렉션 파이프 라인은 함수가 모두 어떤 형태의 시퀀스 데이터 구조에서 작동하는 고차 함수를 구성하는 특별하지만 일반적인 경우입니다. 이 패턴에 대한 이름이 정해져 있지 않아 신조 론 에 의지해야한다고 느꼈습니다 .

작업간에 전달되는 작업 및 컬렉션은 다양한 컨텍스트에서 서로 다른 형식을 취합니다.

Unix에서 컬렉션은 항목이 파일의 행인 텍스트 파일입니다. 각 줄에는 공백으로 구분 된 다양한 값이 포함됩니다. 각 값의 의미는 행의 순서에 따라 제공됩니다. 작업은 유닉스 프로세스이며 컬렉션은 파이프 라인 연산자를 사용하여 구성되며 한 프로세스의 표준 출력은 다음 프로세스의 표준 입력으로 파이프됩니다.

객체 지향 프로그램에서 컬렉션은 컬렉션 클래스 (목록, 배열, 집합 등)입니다. 컬렉션의 항목은 컬렉션 내의 개체이며 이러한 개체는 컬렉션 자체이거나 더 많은 컬렉션을 포함 할 수 있습니다. 오퍼레이션은 콜렉션 클래스 자체에 정의 된 다양한 메소드입니다. 일반적으로 일부 상위 레벨의 수퍼 클래스에 있습니다. 작업은 메서드 체인을 통해 구성됩니다.

기능적 언어에서 컬렉션은 객체 지향 언어와 비슷한 방식으로 컬렉션입니다. 그러나 이번에 항목은 일반 컬렉션 유형 자체입니다. OO 컬렉션 파이프 라인은 객체를 사용하고 기능 언어는 해시 맵을 사용합니다. 최상위 컬렉션의 요소는 컬렉션 자체 일 수 있고 해시 맵의 요소는 컬렉션 일 수 있습니다. 따라서 객체 지향 사례와 마찬가지로 임의로 복잡한 계층 구조를 가질 수 있습니다. 연산은 함수이며, 중첩 또는 Clojure의 화살표 연산자와 같이 선형 표현을 형성 할 수있는 연산자에 의해 구성 될 수 있습니다.

패턴은 다른 곳에서도 나타납니다. 관계형 모델이 처음 정의되었을 때 중간 컬렉션이 관계로 제한되는 컬렉션 파이프 라인으로 생각할 수 있는 관계형 대수 와 함께 제공 되었습니다. 그러나 SQL은 파이프 라인 접근 방식을 사용하지 않고 대신 이해력과 비슷한 접근 방식을 사용합니다 (나중에 논의 할 것임).

이와 같은 일련의 변환 개념은 프로그램 구조화에 대한 일반적인 접근 방식이므로 파이프 및 필터 아키텍처 패턴을 수집합니다. 컴파일러는 종종 이러한 방식으로 작동하여 다양한 최적화를 통해 소스 코드에서 구문 트리로 변환 한 다음 코드를 출력합니다. 컬렉션 파이프 라인의 특징은 단계 간의 공통 데이터 구조가 컬렉션이라는 것입니다. 이는 특정 공통 파이프 라인 작업 집합으로 이어집니다.

더 많은 파이프 라인 및 운영 탐색

지금까지 사용한 파이프 라인의 한 가지 예는 컬렉션 파이프 라인에서 일반적으로 사용되는 몇 가지 작업 만 사용합니다. 이제 몇 가지 예를 통해 더 많은 작업을 살펴보겠습니다. 요즘은 그 언어에 더 익숙하기 때문에 루비를 고수 할 것이지만, 동일한 파이프 라인은 일반적으로이 패턴을 지원하는 다른 언어로 형성 될 수 있습니다.

총 단어 수 얻기 (맵핑 및 감소)

```
-제목 : NoDBA
  단어 : 561
  태그 : [nosql, people, orm]
  유형 : : bliki
-제목 : Infodeck
  단어 : 1145
  태그 : [nosql, 쓰기]
  유형 : : bliki
-제목 : OrmHate
  단어 : 1718
  태그 : [nosql, orm]
  유형 : : bliki
-제목 : 루비
  단어 : 1313
  태그 : [루비]
  유형 : : article
-제목 : DDD_Aggregate
  단어 : 482
```

```
5219
```

```
  태그 : [nosql, ddd]
  유형 : : bliki
```

가장 중요한 파이프 라인 작업 중 두 가지를 간단한 작업으로 설명 할 수 있습니다. 목록의 모든 기사에 대한 총 단어 수를 얻는 방법입니다. 이러한 작업 중 첫 번째는 map 이며, 이는 입력 컬렉션의 각 요소에 지정된 람다를 적용한 결과 인 멤버가있는 컬렉션을 반환합니다.

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

따라서 이것을 사용하면 기사 목록을 각 기사의 단어 수 목록으로 변환 할 수 있습니다. 이 시점에서 우리는 어색한 컬렉션 파이프 라인 작업 중 하나 인 reduce 를 적용 할 수 있습니다 . 이 작업은 입력 컬렉션을 단일 결과로 줄입니다. 종종이를 수행하는 모든 기능을 축소라고합니다. 감소는 종종 단일 값으로 감소한 다음 수집 파이프 라인의 마지막 단계로만 나타날 수 있습니다. 루비의 일반적인 reduce 함수는 두 개의 변수가있는 람다를 사용합니다. 각 요소에 대한 일반적인 변수와 다른 누산기입니다. 감소의 각 단계에서 새 요소로 람다를 평가 한 결과로 누산기의 값을 설정합니다. 그런 다음 다음과 같은 숫자 목록을 합산 할 수 있습니다.

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

메뉴에서이 두 가지 작업을 수행하면 총 단어 수를 계산하는 것은 2 단계 파이프 라인입니다.

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

첫 번째 단계는 기사 목록을 단어 수 목록으로 변환하는지도입니다. 두 번째 단계에서는 단어 수 목록에서 감소를 실행하여 합계를 만듭니다.

> 함수를 람다로 또는 정의 된 함수 이름으로 파이프 라인 작업에 전달할 수 있습니다.

이 시점에서 컬렉션 파이프 라인에서 단계를 구성하는 함수를 표현할 수있는 몇 가지 다른 방법이 있음을 언급 할 가치가 있습니다. 지금까지 각 단계에 람다를 사용했지만 대안은 함수 이름 만 사용하는 것입니다. 이 파이프 라인을 클로저로 작성하면 자연스럽게 작성하는 방법은 다음과 같습니다.

```
(->> (기사)
     (지도 : 단어)
     (줄이기 +))
```

이 경우 관련 기능의 이름만으로 충분합니다. map에 전달 된 함수는 입력 컬렉션의 각 요소에서 실행되고, reduce는 각 요소 및 누산기와 함께 실행됩니다. 루비에서도 동일한 스타일을 사용할 수 있습니다. 여기 words컬렉션의 각 객체에 정의 된 메서드가 있습니다. [삼]

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

일반적으로 함수 이름을 사용하는 것은 조금 더 짧지 만 각 개체에 대한 간단한 함수 호출로 제한됩니다. 람다를 사용하면 좀 더 많은 구문에 대해 좀 더 유연하게 사용할 수 있습니다. Ruby로 프로그래밍 할 때 대부분의 경우 람다를 사용하는 것을 선호하지만 Clojure에서 작업하는 경우 가능할 때 함수 이름을 사용하는 경향이 있습니다. 어느 방향으로 가는지는 크게 중요하지 않습니다. [4]

각 유형의 기사 수 가져 오기 (그룹 별)

```
-제목 : NoDBA
  단어 : 561
  태그 : [nosql, people, orm]
  유형 : : bliki
-제목 : Infodeck
  단어 : 1145
  태그 : [nosql, 쓰기]
  유형 : : bliki
-제목 : OrmHate
  단어 : 1718
  태그 : [nosql, orm]
  유형 : : bliki
```

```
{bliki : 4, 문서 : 1}
```

```
-제목 : 루비
  단어 : 1313
  태그 : [루비]
  유형 : : article
-제목 : DDD_Aggregate
  단어 : 482
  태그 : [nosql, ddd]
  유형 : : bliki
```

다음 파이프 라인 예제에서는 각 유형별로 얼마나 많은 기사가 있는지 알아 봅시다. 출력은 키가 유형이고 값이 해당 기사 수인 단일 해시 맵입니다. [5]

이를 위해서는 먼저 기사 유형별로 기사 목록을 그룹화해야합니다. 여기서 작업 할 컬렉션 연산자는 그룹 별 작업입니다. 이 작업은 해당 요소에서 제공된 코드를 실행한 결과에 의해 인덱싱 된 해시에 요소를 넣습니다. 이 작업을 사용하여 태그 수에 따라 기사를 그룹으로 나눌 수 있습니다.

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

지금해야 할 일은 각 그룹의 기사 수를 얻는 것입니다. 겉으로보기에 이것은지도 작업을위한 간단한 작업이며 기사 수를 계산하는 것뿐입니다. 그러나 여기서 복잡한 점은 각 그룹에 대해 2 비트의 데이터, 즉 그룹 이름과 개수를 반환해야한다는 것입니다. 더 간단하지만 연결된 문제는 앞서 본 맵 예제에서 값 목록을 사용하지만 그룹별 작업의 출력은 해시 맵이라는 것입니다.

> 해시 맵을 키-값 쌍의 목록으로 취급하는
> 것이 종종 유용합니다.

이 문제는 간단한 유닉스 예제를 지나친 후 수집 파이프 라인에서 흔히 발생하는 문제입니다. 우리가 전달할 수있는 컬렉션은 종종 목록이지만 해시 일 수도 있습니다. 우리는 둘 사이를 쉽게 변환해야합니다. 이렇게하는 비결은 해시를 쌍의 목록으로 생각하는 것입니다. 여기서 각 쌍은 키와 해당 값입니다. 해시의 각 요소가 정확히 어떻게 표현되는지는 언어마다 다르지만 간단하고 일반적인 접근 방식은 각 해시 요소를 요소를 2 개 가진 배열로 처리하는 것 `[key, value]`입니다.

Ruby는 정확히 이것을 수행하며 to_h메소드를 사용하여 쌍의 배열을 해시로 바꿀 수 있습니다 . 이렇게지도를 적용 할 수 있습니다.

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

이러한 종류의 해시와 배열 간의 바운싱은 컬렉션 파이프 라인에서 매우 일반적입니다. 배열 조회로 쌍에 액세스하는 것은 약간 어색하므로 Ruby를 사용하면 쌍을 이와 같이 직접 두 개의 변수로 분해 할 수 있습니다.

```
some_articles
  .group_by {| a | 유형}
  .map {| 키, 값 | [키, 값. 크기]}
  .to_h
```

Destructuring은 함수형 프로그래밍 언어에서 일반적으로 사용되는 기술입니다. 이러한 해시 목록 데이터 구조를 전달하는 데 너무 많은 시간을 소비하기 때문입니다. 루비의 구조화 구문은 매우 미미하지만이 간단한 목적에는 충분합니다.

클로저에서이 작업을 수행하는 것은 거의 동일합니다. [6]

```
(->> (기사)
     (그룹 별 : 유형)
     (맵 (fn [[kv]] [k (카운트 v)]))
     ({}로))
```

각 태그에 대한 기사 수 얻기

```
-제목 : NoDBA
  단어 : 561
  태그 : [nosql, people, orm]
  유형 : : bliki
-제목 : Infodeck
  단어 : 1145
  태그 : [nosql, 쓰기]
  유형 : : bliki
```

```
: nosql :
  : 기사 : 4
  : 단어 : 3906
:사람들:
  : 기사 : 1
  : 단어 : 561
: orm :
  : 기사 : 2
```

```
-제목 : OrmHate
  단어 : 1718
  태그 : [nosql, orm]
  유형 : : bliki
-제목 : 루비
  단어 : 1313
  태그 : [루비]
  유형 : : article
-제목 : DDD_Aggregate
  단어 : 482
  태그 : [nosql, ddd]
  유형 : : bliki
```

```
      : 단어 : 2279
    :쓰기:
      : 기사 : 1
      : 단어 : 1145
    :루비:
      : 기사 : 1
      : 단어 : 1313
    : ddd :
      : 기사 : 1
      : 단어 : 482
```

다음 파이프 라인을 위해 목록에 언급 된 각 태그에 대한 기사 및 단어 수를 생성합니다. 이를 위해서는 컬렉션의 데이터 구조를 상당히 재구성해야합니다. 현재 우리의 최상위 항목은 많은 태그를 포함 할 수있는 기사입니다. 이를 위해 데이터 구조를 풀어야 최상위 레벨이 여러 기사를 포함하는 태그가됩니다. 이에 대해 생각하는 한 가지 방법은 다 대다 관계를 반전하여 태그가 기사가 아닌 집계 요소가되도록하는 것입니다.

> 이 예는 다 대일 관계를 반전시킵니다.

파이프 라인을 시작하는 컬렉션의 계층 구조를 이렇게 재구성하면 더 복잡한 파이프 라인이 생성되지만 여전히이 패턴의 기능 내에 있습니다. 이와 같이 작은 단계로 나누는 것이 중요합니다. 이와 같은 변환은 일반적으로 전체 변환을 작은 조각으로 나누고 함께 묶을 때 추론하기가 훨씬 쉽습니다. 이것이 컬렉션 파이프 라인 패턴의 전체 요점입니다.

첫 번째 단계는 태그에 초점을 맞추고 데이터 구조를 확장하여 각 태그-아티클 조합에 대해 하나의 레코드를 갖도록하는 것입니다. 나는 이것이 연관 테이블을 사용하여 관계형 데이터베이스에서 다 대다 관계를 나타내는 방법과 비슷하다고 생각합니다. 이를 위해 기사를 가져와 각 태그와 기사에 대해 쌍 (두 요소 배열)을 내보내는 람다를 만듭니다. 그런 다음이 람다를 모든 기사에 매핑합니다.

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

다음과 같은 구조를 생성합니다.

```
[
  [[: nosql, 기사 (NoDBA)]
    [: 사람, 기사 (NoDBA)]
    [: orm, 기사 (NoDBA)]
  ]
  [[: nosql, 기사 (Infodeck)]
    [: 작성, 기사 (Infodeck)]
  ]
  기사 행 # 개 더
]
```

맵의 결과는 각 기사에 대해 하나의 중첩 된 목록이있는 쌍 목록입니다. 중첩 된 목록이 방해가되므로 병합 작업을 사용하여 병합 합니다.

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

굽힐 수 있는

```
[
  [: nosql, 기사 (NoDBA)]
  [: 사람, 기사 (NoDBA)]
  [: orm, 기사 (NoDBA)]
  [: nosql, 기사 (Infodeck)]
  [: 작성, 기사 (Infodeck)]
  기사 행 # 개 더
]
```

평면화해야하는 불필요한 수준의 중첩 목록을 생성하는이 작업은 매우 일반적이어서 대부분의 언어 에서 단일 단계로이를 수행 할 수있는 평면 맵 작업을 제공합니다 .

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

하나는 이와 같은 쌍 목록이 있고 태그별로 그룹화하는 간단한 작업입니다.

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

굽힐 수 있는

```
{
  :사람들:
    [[: 사람, 기사 (NoDBA)]]
  : orm :
    [[: orm, 기사 (NoDBA)]
      [: orm, Article (OrmHate)]
    ]
  :쓰기:
    [[: 작성, 기사 (Infodeck)]]
  레코드 # 개 더
}
```

그러나 첫 번째 단계와 마찬가지로 각 연결의 값이 단순히 기사 목록이 아닌 키 / 기사 쌍의 목록이기 때문에 이는 성가신 추가 수준의 중첩을 도입합니다. 쌍 목록을 기사 목록으로 대체하는 함수를 매핑하여이를 제거 할 수 있습니다.

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

이것은 산출

```
{
  : 사람들 : [기사 (NoDBA)]
  : orm : [Article (NoDBA), Article (OrmHate)]
  : writing : [기사 (Infodeck)]
  레코드 # 개 더
}
```

이제 기본 데이터를 각 태그에 대한 기사로 재구성하여 다 대다 관계를 역전 시켰습니다. 필요한 결과를 생성하려면 필요한 정확한 데이터를 추출하기위한 간단한지도만 있으면됩니다.

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

이것은 해시의 해시 인 최종 데이터 구조를 생성합니다.

```
: nosql :
  : 기사 : 4
  : 단어 : 3906
:사람들:
  : 기사 : 1
  : 단어 : 561
: orm :
  : 기사 : 2
  : 단어 : 2279
:쓰기:
  : 기사 : 1
  : 단어 : 1145
:루비:
  : 기사 : 1
  : 단어 : 1313
: ddd :
  : 기사 : 1
  : 단어 : 482
```

Clojure에서 동일한 작업을 수행하는 것은 동일한 형식을 취합니다.

```
(->> (기사)
     (mapcat # (map (fn [태그] [태그 %]) (: 태그 %)))
     (첫 번째 그룹)
     (맵 (fn [[kv]] [k (마지막 맵)]))
     (map (fn [[kv]] {k {: articles (count v), : words (reduce + (map : words v))}
     ({}로))
```

> *Clojure*의 평면지도 작업을 *mapcat*이라고합니다.

이와 같이 더 복잡한 파이프 라인을 구축하는 것은 앞서 보여 드린 단순한 파이프 라인보다 더 힘들 수 있습니다. 한 번에 각 단계를주의 깊게 수행하고 각 단계의 출력 컬렉션을주의 깊게 살펴보고 올바른 모양인지 확인하는 것이 가장 쉽습니다. 이 모양을 시각화하려면 일반적으로 들여 쓰기를 사용하여 컬렉션의 구조를 표시하기 위해 일종의 예쁜 인쇄가 필요합니다. 롤링 테스트 우선 스타일로이 작업을 수행하는 것도 유용합니다. 처음에는 데이터의 형태에 대한 간단한 단언 (예 : 첫 번째 단계의 레코드 수)을 사용하여 테스트를 작성하고 추가 단계를 추가 할 때 테스트를 발전시킵니다. 파이프 라인에.

여기에있는 파이프 라인은 각 단계에서 빌드하는 것이 의미가 있지만 최종 파이프 라인은 진행 상황을 너무 명확하게 드러내지 않습니다. 첫 번째 단계는 실제로 각 태그별로 기사 목록을 색인화하는 것이므로 해당 작업을 자체 기능으로 추출하여 더 잘 읽는다고 생각합니다.

```
(Defn index-by [f, seq]
    (->> seq
         (mapcat # (map (fn [key] [key %]) (f %)))
         (첫 번째 그룹)
         (맵 (fn [[kv]] [k (맵 마지막 v)]))))
(전체 단어 정의 [기사]
    (줄이기 + (지도 : 단어 기사)))

(->> (기사)
     (색인 기준 : 태그)
     (map (fn [[kv]] {k {: articles (count v), : words (total-words v)}}))
     ({}로))
```

또한 단어 수를 자체 기능으로 고려할 가치가 있다고 느꼈습니다. 팩토링은 줄 수를 추가하지만 이해하기 쉽도록 구조화 코드를 추가하는 것이 항상 행복하다고 생각합니다. 간결하고 강력한 코드는 좋지만 간결함은 명확성을 제공 할 때만 가치가 있습니다.

Ruby와 같은 객체 지향 언어에서 이와 동일한 팩토링을 수행하려면 index_by파이프 라인에서 컬렉션의 자체 메서드 만 사용할 수 있기 때문에 컬렉션 클래스 자체에 새 메서드를 추가해야합니다 . Ruby를 사용하면이 작업을 수행하기 위해 원숭이 패치 배열

```
class Array
  def invert_index_by & proc
```

```
    flat_map {| e | proc.call (e) .map {| key | [키, e]}}
      .group_by {| pair | pair.first}
      .map {| k, pairs | [k, pairs.map {| p | p.last}]}
  종료
종료
```

간단한 이름 "index_by"는 로컬 함수의 컨텍스트에서 의미가 있지만 컬렉션 클래스의 일반 메서드만큼 의미가 없기 때문에 여기에서 이름을 변경했습니다. 컬렉션 클래스에 메서드를 넣어야하는 것은 OO 접근 방식의 심각한 단점이 될 수 있습니다. 일부 플랫폼에서는 이러한 종류의 인수 분해를 배제하는 라이브러리 클래스에 메서드를 전혀 추가 할 수 없습니다. 다른 것들은 이와 같이 원숭이 패치를 사용하여 클래스를 수정할 수 있도록 허용하지만 이로 인해 클래스의 API가 전역 적으로 눈에 띄게 변경되므로 로컬 함수보다 더 신중하게 생각해야합니다. 여기서 가장 좋은 옵션은 C #의 확장 메서드 또는 기존 클래스를 변경할 수있는 Ruby의 개선과 같은 메커니즘을 사용하는 것입니다. 그러나 더 작은 네임 스페이스의 컨텍스트에서만 가능합니다.

이 메서드를 정의하면 clojure 예제와 비슷한 방식으로 파이프 라인을 인수 분해 할 수 있습니다.

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

여기서도 Clojure 케이스에서했던 것처럼 단어 계산 함수를 인수 분해했지만, 제가 만든 함수를 호출하기 위해 명시적인 메서드를 사용해야하므로 루비에서는 인수 분해가 덜 효과적이라는 것을 알았습니다. 많지는 않지만 가독성에 약간의 마찰을 추가합니다. 물론 호출 구문을 제거하는 완전한 방법으로 만들 수 있습니다. 하지만 여기서 조금 더 나아가 요약 함수를 포함하는 클래스를 추가하고 싶습니다.

```
ArticleSummary 클래스
  def 초기화 기사
    @articles = 기사
  종료
  데프 크기
    @ articles.size
```

```
  종료
  def total_words
    @ articles.map {| a | a.words} .reduce (: +)
  종료
종료
```

이렇게 사용하면

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

많은 사람들은 한 번의 사용으로 몇 가지 기능을 제외하기 위해 완전히 새로운 클래스를 도입하는 것이 너무 무겁다 고 느낄 것입니다. 이와 같은 현지화 작업에 대한 수업을 소개하는 데 문제가 없습니다. 이 특별한 경우에는 추출이 필요한 총 단어 함수이기 때문에 그렇게하지 않을 것입니다.하지만 그 클래스에 도달하기 위해서는 출력에 조금만 더 필요합니다.

대안

컬렉션 파이프 라인 패턴은 내가 지금까지 이야기 한 것들을 성취하는 유일한 방법이 아닙니다. 가장 명백한 대안은 대부분의 사람들이 이러한 경우에 일반적으로 사용했던 간단한 루프입니다.

루프 사용

상위 3 개의 NoSQL 기사의 루비 버전을 비교해 보겠습니다.

컬렉션 파이프 라인 고리

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

```
결과 = []
some_articles.each do | a |
  결과 << a if a.tags.include? (
종료
result.sort_by! (& : 단어)
반환 결과 [0..2]
```

컬렉션 파이프 라인 버전은 약간 짧고 제 눈에는 더 명확합니다. 주로 파이프 라인 개념이 저에게 익숙하고 자연스럽게 명확하기 때문입니다. 즉, 루프 버전은 그다지 나쁘지 않습니다.

여기에 단어 수 사례가 있습니다.

컬렉션 파이프 라인

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

고리

```
결과 = 0
some_articles.each do | a |
  결과 + = a. 단어

종료
반환 결과
```

그룹 사례

컬렉션 파이프 라인

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

고리

```
결과 = Hash.new (0)
some_articles.each do | a |
  결과 [a.type] + = 1
종료
반환 결과
```

태그 별 기사 수

컬렉션 파이프 라인

```
some_articles
```

```
    .flat_map {| a | a.tags.map {| tag | [태그, 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
```

이 경우 컬렉션 파이프 라인 버전은 훨씬 더 짧지 만 두 경우 모두 의도를 가져 오기 위해 리팩터링하기 때문에 비교하기가 까다 롭습니다.

이해력 사용

일부 언어에는 단순 컬렉션 파이프 라인을 반영하는 이해 구조 (일반적으로 목록 이해 구조라고 함)가 있습니다. 천 단어가 넘는 모든 기사의 제목을 검색해보십시오. 이해 구문이있는 coffeescript를 사용하여 설명하지만 Javascript의 자체 기능을 사용하여 컬렉션 파이프 라인을 만들 수도 있습니다.

관로

```
some_articles
  .filter (a)->
    a. 단어> 1000
  .map (a)->
    제목
```

이해

```
(a.words> 1000 인 경우 some_article
```

이해력의 정확한 기능은 언어마다 다르지만 단일 명령문으로 표현할 수있는 특정 작업 순서로 생각할 수 있습니다. 이러한 생각은 언제 사용할지 결정하는 첫 번째 부분을 조명합니다. 이해는 파이프 라인 작업의 특정 조합에만 사용할 수 있으므로 근본

적으로 유연성이 떨어집니다. 즉, 이해력은 가장 일반적인 경우에 대해 정의되므로 여전히 많은 경우에 옵션입니다.

이해는 일반적으로 파이프 라인 자체에 배치 될 수 있으며 본질적으로 단일 작업으로 작동합니다. 따라서 1000 단어가 넘는 모든 기사의 총 단어 수를 얻으려면 다음을 사용할 수 있습니다.

관로 파이프 라인에 대한

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

```
(a.words> 1000 인 경우 some_article
.reduce (x, y)-> x + y
```

그렇다면 문제는 그들이 작업하는 사례에서 이해력이 파이프 라인보다 나은지 여부입니다. 이해력을 좋아하는 사람들은 파이프 라인이 이해하기 쉽고 더 일반적이라고 말할 수 있습니다. (나는 후자 그룹에 속합니다.)

중첩 된 연산자 식

컬렉션으로 할 수있는 유용한 작업 중 하나는 집합 연산으로 컬렉션을 조작하는 것입니다. 따라서 빨간색, 파란색, 호텔 앞 또는 점유중인 객실을 반환하는 기능이있는 호텔을보고 있다고 가정 해 보겠습니다. 그런 다음 표현식을 사용하여 호텔 앞의 비어있는 빨간색 또는 파란색 방을 찾을 수 있습니다.

루비...

```
(전면 & (빨간색 | 파란색))-점유
```

클로저...

```
(차
 (교차로
  (유니온 레드 블루스)
  전선)
 occ)
```

> *Clojure*는 집합 데이터 유형에 대해 집합 연산을 정의하므로 여기에있는 모든 기호는 집합입니다.

이러한 식을 컬렉션 파이프 라인으로 공식화 할 수 있습니다.

루비...

```
빨간
  .union (파란색)
  .intersect (전면)
  .diff (점유)
```

> 나는 원숭이 패치 배열을 일반 메소드로 설정 작업을 추가합니다.

클로저...

```
(->> 빨강
     (유니온 블루스)
     (교차점)
     (OCC 제거))
```

> 스레딩을위한 올바른 순서로 인수를 얻으려면 여기에 *clojure*의 '*remove*'메서드가 필요합니다.

그러나 특히 중위 연산자를 사용할 수있는 경우 중첩 된 연산자 식 형식을 선호합니다 . 그리고 더 복잡한 표현은 정말 파이프처럼 얽힐 수 있습니다.

즉, 파이프 라인 중간에 집합 작업을 던지는 것이 유용한 경우가 많습니다. 방의 색상과 위치가 방 기록의 속성이지만 점유 된 방의 목록이 별도의 컬렉션에있는 경우를 상상해 봅시다.

루비...

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

클로저...

```
(->> (방)
     (필터 # (# {: blue : red} (: color %)))
     (필터 # (# {: front} (: location %)))
     (제거 (설정 (점유))))
```

> 여기 *(set (occupied))*에서는 클로저의 집합 멤버쉽에 대한 조건 자로 컬렉션에 래핑 된 집합을 사용하는 방법을 보여줍니다.

중위 연산자는 중첩 연산자 표현식에 적합하지만 파이프 라인에서는 잘 작동하지 않아 성가신 괄호가 필요합니다.

루비...

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

집합 연산에서 명심해야 할 또 다른 점은 컬렉션이 일반적으로 순서가 지정되고 중복을 허용하는 목록이라는 것입니다. 집합 작업의 의미를 확인하려면 라이브러리의 세부 사항을 살펴 봐야합니다. Clojure는 집합 연산을 사용하기 전에 목록을 집합으로 바꾸도록 강요합니다. Ruby는 모든 배열을 집합 연산자로 받아들이지 만 입력 순서를 유지하면서 출력에서 중복을 제거합니다.

게으름

게으름의 개념은 함수형 프로그래밍 세계에서 나왔습니다. 동기는 다음과 같은 코드일 수 있습니다.

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

이러한 코드를 사용하면 slow_complex_method많은 요소를 평가 하는 데 많은 시간을 소비 한 다음 상위 5 개를 제외한 모든 결과를 버릴 수 있습니다. 게으름을 사용하면 기본 플랫폼이 상위 5 개 결과 만 필요하다고 판단한 다음 작업 만 수행 할 수 있습니다 slow_complex_method. 필요한 것.

실제로 이것은 런타임 사용으로 더 진행됩니다. 결과 slow_complex_method가 UI의 스크롤 목록으로 파이프된다고 가정 해 봅시다 . 지연 파이프 라인은 최종 결과가보기로 스크롤 될 때 요소에 대해서만 파이프 라인을 호출합니다.

수집 파이프 라인이 게으 르기 위해서는 수집 파이프 라인 함수가 게으름을 염두에 두고 빌드되어야합니다. Clojure 및 Haskell과 같은 일반적으로 기능적인 언어 인 일부 언어는 처음부터이 작업을 수행합니다. 다른 경우에 게으름은 특별한 컬렉션 클래스 그룹에 빌드 될 수 있습니다. 자바와 루비에는 게으른 컬렉션 구현이 있습니다.

일부 파이프 라인 작업은 게으름으로 작동 할 수 없으며 전체 목록을 평가해야합니다. 정렬은 한 가지 예이며 전체 목록이 없으면 단일 상위 값도 결정할 수 없습니다. 게으름을 심각하게 받아들이는 플랫폼은 일반적으로 게으름을 유지할 수없는 작업을 문서화합니다.

병 행

많은 파이프 라인 작업은 자연스럽게 병렬 호출에서 잘 작동합니다. 내가 사용하는 경우 지도 컬렉션에있는 다른 요소의에 의존하지 않는 하나 개의 요소를 사용하여 결과를. 따라서 여러 코어가있는 플랫폼에서 실행중인 경우 맵 평가를 여러 스레드에 분산하여이를 활용할 수 있습니다.

많은 플랫폼에는 이와 같이 병렬로 평가를 배포하는 기능이 포함되어 있습니다. 대규모 세트에서 복잡한 기능을 실행하는 경우 멀티 코어 프로세서를 활용하여 성능을 크게 향상시킬 수 있습니다.

그러나 병렬화가 항상 성능을 향상시키는 것은 아닙니다. 때로는 병렬 처리에서 얻는 것보다 병렬 배포를 설정하는 데 더 많은 시간이 걸립니다. 결과적으로 대부분의 플랫폼은 Clojure의 pmap기능이지도의 병렬 버전 인 방식과 같이 명시 적으로 병렬 처리를 사용하는 대체 작업을 제공합니다 . 성능 최적화와 마찬가지로 성능 테스트를 사용하여 병렬 작업을 사용하는 것이 실제로 성능 향상을 제공하는지 확인해야합니다.

불변성

컬렉션 파이프 라인은 자연스럽게 불변 데이터 구조에 적합합니다. 파이프 라인을 구축 할 때 각 작업을 출력에 대한 새 컬렉션을 생성하는 것으로 간주하는 것은 당연합니다. 순진하게 수행하면 많은 복사가 필요하므로 많은 양의 데이터에 문제가 발생할 수 있습니다. 그러나 대부분의 경우 실제로는 문제가되지 않습니다. 일반적으로 큰 데이터 덩어리가 아니라 복사되는 포인터 세트가 다소 작습니다.

문제가 될 때 이러한 방식으로 변환되도록 설계된 데이터 구조로 불변성을 유지할 수 있습니다. 함수형 프로그래밍 언어는이 스타일로 효율적으로 조작 할 수있는 데이터 구조를 사용하는 경향이 있습니다.

필요한 경우 컬렉션을 바꾸지 않고 업데이트하는 작업을 사용하여 변경 가능성을 희생 할 수 있습니다. 비 기능적 언어의 라이브러리는 종종 컬렉션 파이프 라인 연산자의 파괴적인 버전을 제공합니다. 나는 당신이 훈련 된 성능 튜닝 연습의 일부로 만 사용하는 것이 좋습니다 . 변경되지 않는 작업으로 작업을 시작하고 파이프 라인에 알려진 성능 병목 현상이있는 경우에만 다른 작업을 사용하십시오.

디버깅

컬렉션 파이프 라인 디버깅의 어려움에 대해 몇 가지 질문이있었습니다. Ruby에서 다음과 같은 파이프 라인을 고려하십시오.

```
def get_readers_of_books1 (독자, 책, 날짜)
  data = @ data_service.get_books_read_on (날짜)
  반환 데이터
    .select {| k, v | readers.include? (k)}
    .select {| k, v | ! (books & v) .empty?}
    .keys
종료
```

최신 IDE는 디버깅을 돕기 위해 많은 일을 할 수 있지만, 내가 이것을 루비로하고 emacs로하고 디버거를 비웃는다고 상상해 봅시다.

파이프를 통해 중간 변수를 추출하고 싶을 수도 있습니다.

```
def get_readers_of_books2 (독자, 책, 날짜)
  data = @ data_service.get_books_read_on (날짜)
  임시 = 데이터
    .select {| k, v | readers.include? (k)}
    .select {| k, v | ! (books & v) .empty?}
  pp 온도
  반환 온도
    .keys
종료
```

다소 까다로운 또 다른 일은 파이프 라인의지도에 print 문을 밀수입하는 것입니다.

```
def get_readers_of_books (독자, 책, 날짜)
  data = @ data_service.get_books_read_on (날짜)
  반환 데이터
    .select {| k, v | readers.include? (k)}
    .select {| k, v | ! (books & v) .empty?}
    .map {| e | pp e; e} .to_h
    .keys
종료
```

> 이 경우지도 작업 후 다시 해시로 변환해야합니다.

이는 파이프 라인 내부에 몇 가지 부작용이 필요하므로 코드 검토로 이스케이프 된 경우 알려야하지만 코드가 수행하는 작업을 시각화하는 데 도움이 될 수 있습니다.

적절한 디버거를 사용하면 일반적으로 디버거에서 식을 평가할 수 있습니다. 이렇게 하면 파이프에 중단 점을 설정 한 다음 파이프의 일부를 기반으로 표현식을 평가하여 무슨 일이 일어나고 있는지 확인합니다.

사용시기

컬렉션 파이프 라인을 패턴으로보고 어떤 패턴을 사용하든 사용해야 할 때가 있고 다른 경로를 취해야 할 때가 있습니다. 내가 좋아하는 패턴을 사용하지 않는 이유가 생각 나지 않으면 항상 의심을 받는다.

이를 피하는 첫 번째 징후는 언어 지원이 없을 때입니다. Java로 시작했을 때 컬렉션 파이프 라인을 많이 사용할 수 없다는 점을 놓 쳤기 때문에 다른 많은 사람들과 마찬가지로 패턴을 형성 할 수있는 객체를 만드는 실험을했습니다. 클래스를 만들고 익명 내부 클래스와 같은 것을 사용하여 람다에 가까워짐으로써 파이프 라인 작업을 구성 할 수 있습니다. 그러나 문제는 구문이 너무 복잡해서 컬렉션 파이프 라인을 매우 효과적으로 만드는 명확성을 압도한다는 것입니다. 그래서 포기하고 대신 루프를 사용했습니다. 그 이후로 다양한 기능 스타일 라이브러리가 자바에 등장했으며, 많은 사람들이 초기에는 언어에 없었던 주석을 사용했습니다. 하지만 내 느낌은 깨끗한 람다 표현에 대한 좋은 언어 지원 없이는

반대하는 또 다른 주장은 이해력이있을 때입니다. 이 경우 이해는 종종 간단한 표현으로 작업하기가 더 쉽지만 더 큰 유연성을 위해 여전히 파이프 라인이 필요합니다. 개인적으로 저는 이해력만큼이나 이해하기 쉬운 간단한 파이프 라인을 찾았지만, 팀이 코딩 스타일에서 결정해야하는 종류입니다.

> 차이있을 때마다하는 방법을 추출 어떤 코드 블록을 수행하고 어떻게 그것을 수행이

적합한 언어에서도 파이프 라인의 크기와 복잡성과 같은 다른 한계에 부딪 힐 수 있습니다. 이 기사에서 보여준 것은 작고 선형 적입니다. 내 일반적인 습관은 작은 함수를 작성하는 것이고, 6 줄을 넘으면 짜증이납니다. 파이프 라인에도 비슷한 규칙이 있습니다. 더 큰 파이프 라인은 일반적인 규칙에 따라 별도의 메서드로 고려되어야 합니다. 코드 블록이 수행하는 작업과 수행하는 방법간에 차이가있을 때마다 메서드를 추출합니다.

파이프 라인은 선형 일 때 가장 잘 작동하며 각 단계에는 단일 컬렉션 입력과 단일 출력이 있습니다. 이 기사에서는 이러한 예제를 함께 넣지 않았지만 입력과 출력을 분리하는 것이 가능합니다. 그러나 다시 한 번주의하십시오. 별도의 기능을 고려하는 것은 일반적으로 더 이상 동작을 제어하는 데 핵심입니다.

즉, 컬렉션 파이프 라인은 모든 프로그래머가 알아야 할 훌륭한 패턴입니다. 특히이를 잘 지원하는 Ruby 및 Clojure와 같은 언어에서 그렇습니다. 길고 거친 루프가 필요한 것을 명확하게 캡처 할 수 있으며 코드를 더 읽기 쉽고 저렴하고 쉽게 향상시킬 수 있습니다.

작업 카탈로그

다음은 컬렉션 파이프 라인에서 자주 볼 수있는 작업의 카탈로그입니다. 모든 언어는 사용 가능한 작업과 호출되는 작업에 대해 서로 다른 선택을하지만 공통 기능을 통해 살펴 보려고했습니다.

수집

스몰 토크의 지도 대체 이름 . *Java 8*은 완전히 다른 목적으로 "*collect*"를 사용합니다. 즉, 스트림에서 컬렉션으로 요소를 수집하는 터미널입니다.

지도 참조

연결

컬렉션을 단일 컬렉션으로 연결합니다.

더...

차

파이프 라인에서 제공된 목록의 내용을 제거합니다.

더...

뚜렷한

중복 요소를 제거합니다.

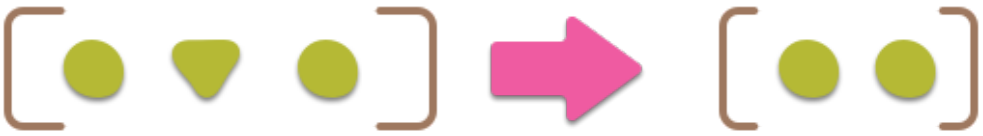
더...

하락

처음 n 개 요소를 제외한 모든 요소를 반환 하는 슬라이스 형식

슬라이스 참조

필터



각 요소에 대해 부울 함수를 실행하고 전달되는 요소 만 출력에 넣습니다.

더...

평면지도

컬렉션에 대한 함수를 매핑하고 결과를 한 수준으로 평면화

더...

단조롭게 하다

컬렉션에서 중첩을 제거합니다.

더...

겹

축소의 대체 이름 때때로 *foldl* (왼쪽 접기) 및 *foldr* (오른쪽 접기)로 표시됩니다.

감소 참조

그룹 별

각 요소에 대해 함수를 실행하고 결과에 따라 요소를 그룹화합니다.

더...

주입

*smalltalk*의 선택기 에서 ***reduce***의 대체 이름입니다 `inject:into:`.

감소 참조

교차로

제공된 컬렉션에도있는 요소를 유지합니다.

더...

지도

주어진 함수를 입력의 각 요소에 적용하고 결과를 출력에 넣습니다.

더...

맵캣

평면지도의 대체 이름

참조 평면지도

줄이다

제공된 함수를 사용하여 입력 요소를 종종 단일 출력 값으로 결합합니다.

더...

받지 않다

filter의 역으로 술어와 일치하지 않는 요소를 반환합니다.

필터 참조

고르다

필터의 대체 이름입니다.

필터 참조

일부분

주어진 첫 번째 위치와 마지막 위치 사이의 목록의 하위 시퀀스를 반환합니다.

더...

종류

출력은 제공된 비교기를 기반으로 정렬 된 입력 사본입니다.

더...

취하다

처음 n 개 요소를 반환하는 슬라이스 형식

슬라이스 참조

노동 조합

이 컬렉션 또는 제공된 컬렉션의 요소를 반환하여 중복을 제거합니다.

더...

각주

1 : 좀 더 관용적 인 lisp 파이프 라인

여기서 한 가지 문제는 lisp 예제가 그다지 관용적이지 않다는 것입니다. 왜냐하면 명명 된 함수 (`#'some-function`구문을 사용하여 쉽게 참조 됨)를 사용하는 것이 일반적이기 때문 입니다. 필요한 경우 특정 경우에 작은 함수를 생성합니다. 이것은 그 예를 더 잘 고려할 수 있습니다.

```
(defun nosqlp (문서)
  (멤버 'nosql (article-tags article)))

(subseq
 (종류
  (# 'nosqlp가 아닌 경우 제거 (일부 문서))
  # '<: key #'article-words)
 0 3)
```

2 : 자바 파이프 라인

다음은 Java의 초기 파이프 라인입니다.

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

예상 할 수 있듯이 Java는 여러 측면에서 매우 장황합니다. Java에서 컬렉션 파이프 라인의 특정 기능은 파이프 라인 함수가 컬렉션 클래스가 아니라 Stream 클래스 (IO 스트림과 다름)에 정의되어 있다는 것입니다. 따라서 처음에는 기사 모음을 스트림으로 변환하고 마지막에는 목록으로 다시 변환해야합니다.

3 : 루비 메소드에 블록을 전달하는 대신 이름 (기호) 앞에 "&"를 붙여 이름이 지정된 함수를 전달할 수 있습니다 `&:words`. 따라서 . 로 감소 하면 "&"필요가 없습니다, 그러나 예외가있다, 당신은 단지, 그것을 함수의 이름을 전달할 수 있습니다. 나는 reduce와 함께 함수 이름을 사용할 가능성이 더 높으므로 불일치에 감사드립니다.

4 : 람다 또는 함수 이름 사용

람다와 함수 이름을 사용하는 것 사이에서 적어도 내 관점에서 보면 흥미로운 언어 역사가 있습니다. 스몰 토크는 람다에 대한 최소한의 구문을 확장하여 작성하기 쉬우 며, 리터럴 메서드를 호출하는 것은 더 어색했습니다. 그러나 Lisp는 명명 된 함수를 쉽게 호출 할 수 있도록 만들었지 만 람다를 사용하려면 추가 구문이 필요했습니다. 종종 해당 구문을 제거하는 매크로로 이어졌습니다.

현대 언어는 둘 다 쉽게 만들려고합니다. Ruby와 Clojure는 함수를 호출하거나 람다를 매우 간단하게 사용합니다.

5 : "map"은 작업 맵 또는 데이터 구조를 나타낼 수 있으므로 여기에 다원 체가 있습니다 . 이 기사에서는 데이터 구조에 "hashmap"또는 "dictionary"를 사용하고 함수에는 "map"만 사용할 것입니다. 그러나 일반적인 대화에서는 종종 맵이라고하는 해시 맵을 듣게 될 것입니다.

6 : Juxt 사용

clojure의 한 가지 옵션은 juxt를 사용하여 맵 내에서 여러 기능을 실행하는 것입니다 .

```
(->> (기사)
     (그룹 별 : 유형)
     (맵 (처음에 juxt (컴포지션 카운트 두 번째)))
     ({}로))
```

나는 람다를 사용하는 버전이 더 명확하다는 것을 알지만 Clojure (또는 일반적으로 함수형 프로그래밍)에 능숙합니다.

감사의 말

이 기사의 초기 초안에 대해 언급 한 동료들에게 감사합니다 : 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 및 Rebecca Parsons.

▶ 중요한 수정