Оглавление

Об этой книге

Перед вами второй том «Clojure на производстве». Это продолжение первой книги, которая вышла три года назад. Мы продолжим изучать Clojure — замечательный язык с акцентом на неизменяемость и асинхронность. Clojure называют современным Лиспом, потому что код на нем пишут S-выражениями — то есть со скобками.

По структуре и изложению книга не отличается от первой. Мы подробно рассмотрим несколько тем, чередуя теорию с практикой. Каждая мысль в тексте подтверждается кодом, и наоборот: сложный код разбит на части и подробно описан текстом.

Как и первый том, продолжение написано на русском языке. Автор много лет пишет на Clojure и знаком с индустрией и сообществом. Все примеры и задачи автор взял из реальных проектов; каждую строчку кода выполнил в REPL.

Коротко о том, что вас ждет. Первая глава расскажет о зиперах в Clojure. Это особый способ работы с коллекциями: непривычный, но крайне мощный. О зиперах мало информации даже на английском языке, и книга закрывает этот недостаток.

Вторая глава посвящена реляционным базам данных, в основном PostgreSQL. Мы рассмотрим основы SQL, подключение и работу с базой из Clojure. Автор учел все наболевшие темы: построение сложных запросов, шаблонизацию SQL, работу с выборкой и все то, о чем забывают другие руководства.

Третья глава охватывает сразу три смежные темы — REPL, Cider и Emacs. Читатель узнает, что такое REPL и как подключиться к нему из редактора. Мы поговорим о сетевом протоколе nREPL, о запуске проекта в Docker и на удаленной машине. Рассмотрим REPL на платформе Javascript и проведем массу экспериментов.

В тексте мы не раз ссылаемся на первую книгу, особенно когда речь идет об исключениях, системах или Clojure.spec. Это не помешает разобраться с темой, даже если вы не читали первый том. Все же автор советует ознакомиться с ним для лучшего понимания.

Книга рассчитана на продвинутую аудиторию. Желательно, чтобы у вас был опыт если не с Clojure, то хотя бы с одним из промышленных языков. Пожелаем читателю терпения, чтобы прочесть книгу до конца.

Код

Исходный код книги в виде файлов LaTeX находится в репозитории `igrishaev/clj-book2`[1]. Если вы нашли опечатку, создайте pull request или issue с описанием проблемы.


clj-book2

Код первой главы о зипперах доступен в репозитории `igrishaev/zipper-manual`[2]. Код второй и третьей — в репозитории `igrishaev/book-sessions`[3], пути `src/book/db.clj` и `repl-chapter` соответственно.


Zipper manual

Используйте код в любых целях, в том числе коммерческих.

Благодарности


Book sessions

Автор благодарен стартапу Clashapp[4] (ныне Huddles) и его коллективу за полученный опыт. Некоторые техники из этого проекта нашли место в книге.


Clashapp

Спасибо читателям блога за присланные опечатки и уточнения. С ними текст удалось улучшить до сдачи в печать.

Особая благодарность Андрею Листопадову за детальные отзывы к черновикам глав. Посетите его сайт: andreyor.st[5].


Andrey Orst

Благодарю коллектив издательства «ДМК Пресс» за то, что взяли рукопись в работу. Их усилиями вы читаете эту книгу сейчас.

Обратная связь

Присылайте ошибки и замечания на почту `ivan@grishaev.me`. Автор обновит макет, и, возможно, следующий читатель получит исправленную версию книги.

[1] github.com/igrishaev/clj-book2.
[2] github.com/igrishaev/zipper-manual.
[3] github.com/igrishaev/book-sessions.
[4] huddlesapp.co.
[5] andreyor.st.

Глава 1

Зипперы

В этой главе мы рассмотрим зипперы в языке Clojure. Это необычный способ работы с коллекциями. С помощью зиппера можно обойти произвольные данные, изменить их или выполнить поиск. Зиппер — мощный инструмент, но вложения в него окупаются со временем. Это сложная абстракция, которая требует подготовки.

1.1 Азы навигации

Объясним зиппер простыми словами. Это обёртка над данными с набором действий. Вот некоторые из них:

- перемещение по вертикали: вниз к потомкам или вверх к родителю;
- перемещение по горизонтали: влево или вправо среди потомков;
- обход всех элементов;
- добавление, редактирование и удаление узлов.

Это неполный список того, что умеют зипперы. Другие их свойства мы рассмотрим по ходу главы. Важно, что указанные действия относятся к любым данным, будь то комбинация векторов и словарей, дерево узлов или XML. Из-за этого зипперы становятся мощным инструментом. Разобраться с ними означает повысить свои навыки и открыть новые двери.

Gérard Huet

Термин «зиппер» ввел ученый Жерар Юэ[1] (Gérard Huet) в 1996 году. Юэ занимался деревьями и искал универсальный способ работы с ними. В знаменитой работе «Functional Pearl: The Zipper»[2] Юэ привел концепцию зиппера на языке OCaml. Документ привлек внимание простотой и ясностью: описание зиппера, включая код и комментарии, уместилось на четырёх страницах. Современные зипперы почти не отличаются от того изложения 1996 года.

zipper.pdf

Хотя Юэ отмечает, что зиппер можно создать на любом языке, лучше всего они прижились в функциональных: Haskell, OCaml, Clojure. Зипперы поощряют неизменяемые данные и чистые преобразования. Для указанных языков написаны библиотеки зипперов, в некоторых случаях больше одной. Наоборот, в императивной среде зипперы почти неизвестны.

Зипперы доступны в Clojure с первой версии. Их легко добавить в проект, не опасаясь проблем лицензии или новых зависимостей.

Зипперы в Clojure используют мощь неизменяемых коллекций. Технически зиппер — это коллекция, которая хранит данные и позицию в них. Всё вместе это называется локацией (location). Шаг в любую сторону вернёт новую локацию подобно тому, как функции `assoc` или `update` производят новые данные, оставляя прежние нетронутыми.

Из текущей локации можно получить *узел* (ноду) — данные, на которые ссылается указатель. На этом месте путаются новички, поэтому уточним различие. Локация — это исходные данные и положение в них. Передвижение по локации порождает локацию. Из локации можно извлечь узел — данные, которые встретились в локации.

Приведём пример с вектором `[1 2 3]`. Чтобы переместиться на **двойку**, обернём данные в зиппер и выполним команды `zip/down` и `zip/right`. С первым шагом мы провалимся в вектор и окажемся на единице. Шаг вправо сдвинет нас на двойку.

Выразим это в коде: подключим модуль `clojure.zip` и переместимся по вектору:

[1] en.wikipedia.org/wiki/Gerard_Huet.

[2] www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf.

```
(require '[clojure.zip :as zip])

(-> [1 2 3]
    zip/vector-zip
    zip/down
    zip/right
    zip/node)
;; 2
```

Функция `zip/vector-zip` создает зиппер из вектора. Вызовы `zip/down` и `zip/right` передвинут указатель на двойку, как и ожидалось. Последний шаг `zip/node` вернёт значение (узел) текущей локации. Если убрать `zip/node`, получим локацию, которая соответствует двойке. Вот как она выглядит:

```
(-> [1 2 3]
    zip/vector-zip
    zip/down
    zip/right)

[2 {:l [1]
    :pnodes [[1 2 3]]
    :ppath nil
    :r (3)}]
```

Это пара, где первый элемент — значение, а второй — словарь направлений.

Наверняка у вас возникли вопросы: откуда мы знаем путь к двойке, ведь она могла быть в другом месте вектора? Что произойдет, если выйти за пределы коллекции? Мы ответим на эти вопросы ниже. Пока что, если вам что-то не понятно, не пугайтесь: мы не раз обсудим всё, что происходит.

Итак, зиппер предлагает перемещение по данным. Несмотря на всю мощь, он не знает, как это делается для конкретной коллекции, и нуждается в вашей помощи. Вот что нужно знать зиппер:

- является текущий элемент веткой или нет? Веткой называют элемент, из которого можно извлечь другие элементы;
- если это ветка, как именно получить её элементы?

Как только мы знаем ответы на эти вопросы, зиппер готов. Заметим, что для изменения зиппера нужен ответ на третий вопрос: как присоединить потомков к ветке. Однако сейчас мы рассматриваем только навигацию, и третий вопрос подождёт.

В техническом плане ответы на эти вопросы — функции. Первая принимает узел и возвращает истину или ложь. Если получили истину, зиппер вызовет вторую функцию с тем же узлом. От неё ожидают коллекцию дочерних узлов или `nil`, если их нет. В терминах зиппера функции называют `branch?` и `children` соответственно.

Чтобы получить зиппер, сообщите ему данные и эти две функции. Посколько мы только читаем зиппер, третья функция будет `nil`.

Зипперы находятся в модуле `clojure.zip`; подключите его с псевдонимом `zip`. В свободное время исследуйте код модуля: он занимает всего 280 строк[3]!

zip.clj

```
(ns my.project
  (:require [clojure.zip :as zip]))
```

Функция `zip/zipper` порождает зиппер из исходных данных и функций. Это центральная точка модуля, его строительный материал. Для особых случаев модуль содержит полуготовые зипперы, которые ожидают только данных. Примером служит функция `vector-zip`. Она работает с вектором, элементы которого могут быть другим вложенным вектором. Приведём её код в сокращении:

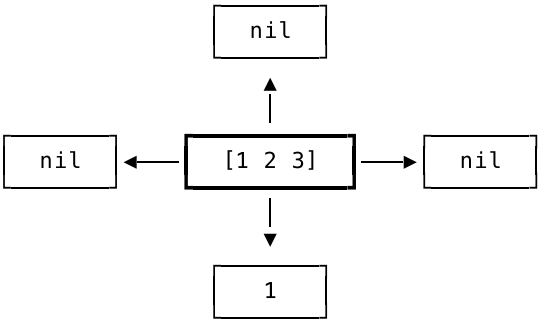
```
1 (defn vector-zip
2   [root]
3   (zipper vector?
4           seq
5           ...
6           root))
```

Третий параметр (строка 5) мы заменили на многоточие. Это функция, которая присоединяет к ветке дочерные узлы при изменении (пока что обходим вопрос стороной).

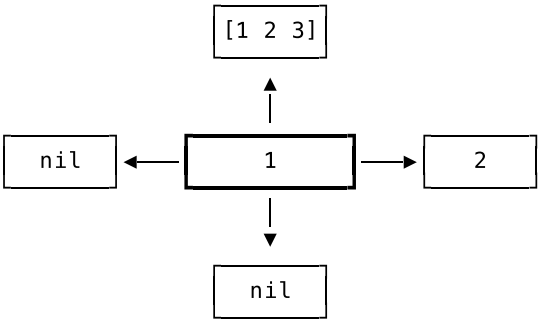
[3] github.com/clojure/clojure/blob/master/src/clj/clojure/zip.clj .

Если передать в `vector-zip` данные `[1 2 3]`, произойдёт следующее. Зиппер обернёт вектор и выставит на него указатель. Из начального положения можно следовать только вниз, потому что у вершины нет родителя (вверх) и соседей (влево и вправо). При смещении **вниз** зиппер сначала проверит, что текущий узел — ветка. Сработает выражение `(vector? [1 2 3])`, что вернёт истину. В этом случае зиппер выполнит `(seq [1 2 3])`, чтобы получить потомков. Ими станет последовательность `(1 2 3)`. Как только потомки найдены, зиппер установит указатель на крайний левый потомок — единицу.

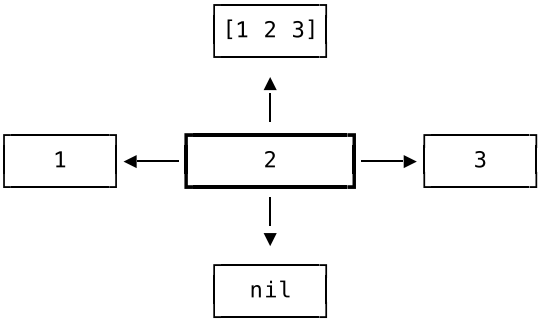
Покажем это на схеме. Начальная позиция, указатель на исходном векторе:



Шаг вниз, указатель на единице:



Шаг вправо, указатель на двойке:



Итак, мы находимся в двойке и можем двигаться дальше по горизонтали. Шаг вправо сдвинет нас на тройку, влево — на единицу. Вот как это выглядит в коде:

```
(def loc2
  (-> [1 2 3]
      zip/vector-zip
      zip/down
      zip/right))

(-> loc2 zip/node)           ;; 2

(-> loc2 zip/right zip/node) ;; 3

(-> loc2 zip/left zip/node)  ;; 1
```

При попытке сдвинуться вниз зиппер выполнит предикат `(vector? 2)`. Результат будет ложью, что означает, что текущий элемент не ветка и движение вниз запрещено.

Во время движения каждый шаг порождает новую локацию, не изменяя старую. Если вы сохранили локацию в переменную, дальнейшие вызовы `zip/right` или `zip/down` не изменят её. Выше мы объявили переменную `loc2`, которая указывает на двойку. Проследуем от неё к исходному вектору:

```
(-> loc2 zip/up zip/node)

;; [1 2 3]
```

При ручном перемещении велики шансы выйти за пределы данных. Шаг в никуда вернёт `nil` вместо локации:

```
(-> [1 2 3]
    zip/vector-zip
    zip/down
    zip/left)
nil
```

Это сигнал, что вы идёте по неверному пути. Из `nil` нельзя вернуться на прежнее место, потому что у `nil` нет сведений о позиции. Для `nil` функции `zip/up`, `zip/right` и другие тоже вернут `nil`. При ручном перемещении проверяйте результат на `nil` или пользуйтесь оператором `some->`:

```
(some-> [1 2 3]
        zip/vector-zip
        zip/down
        zip/left
        zip/left
        ...)
;; nil
```

К исключению относится функция `zip/down`: при попытке спуститься из `nil` вы получите `NullPointerException`. Это недочёт, который, возможно, когда-нибудь исправят.

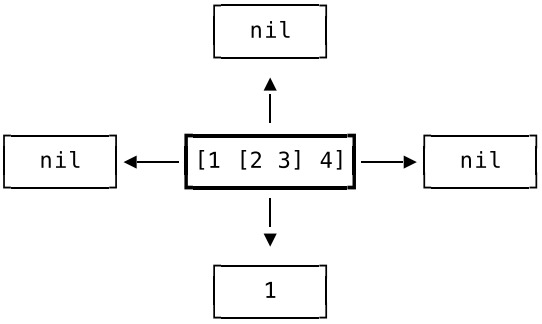
```
(-> [1 2 3]
    zip/vector-zip
    zip/down
    zip/left
    zip/down)

;; Execution error (NullPointerException)...
```

Как и в случае выше, от исключения вас убережёт макрос `some->`.

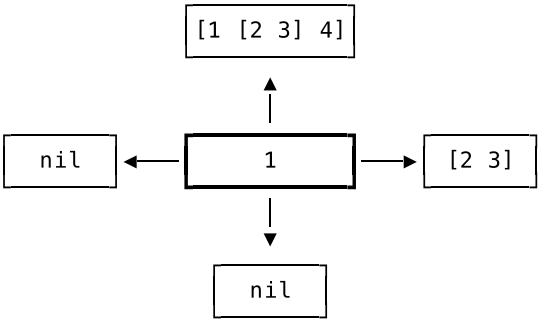
Рассмотрим случай, когда у вектора вложенные элементы: `[1 [2 3] 4]`. Чтобы переместиться на **тройку**, выполним шаги «вниз», «вправо», «вниз», «вправо». Сохраним локацию в переменную `loc3`:

```
(def data
  [1 [2 3] 4])

(def loc3
  (-> data
      zip/vector-zip
      zip/down
      zip/right
      zip/down
      zip/right))

(zip/node loc3)

;; 3
```

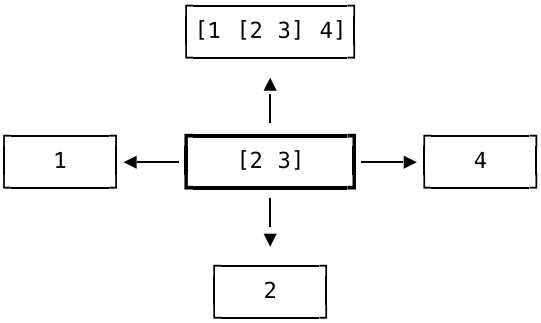
Рисунки ниже показывают, что происходит на каждом шаге. Исходная позиция:



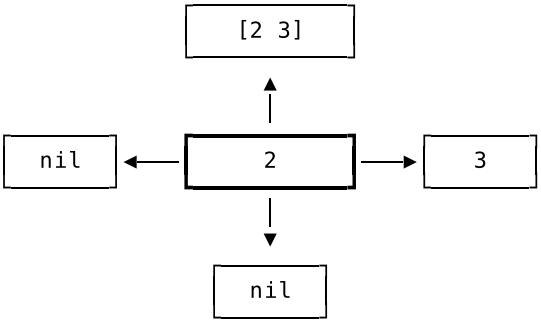
Шаг вниз:



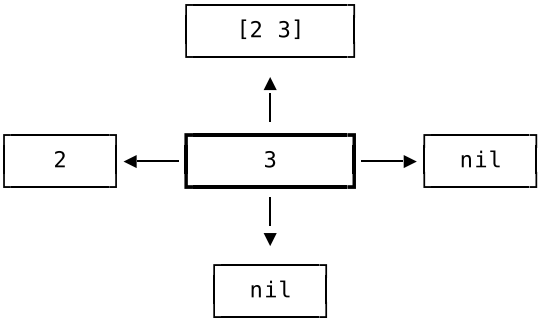
Вправо:



Вниз:

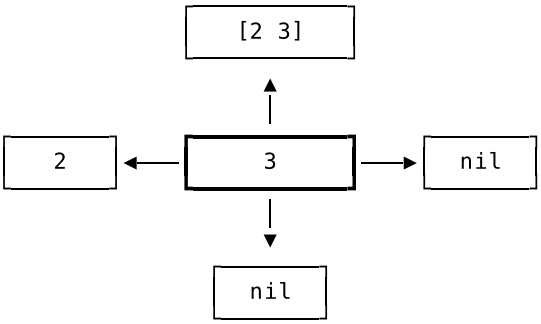


Вправо. Мы у цели:

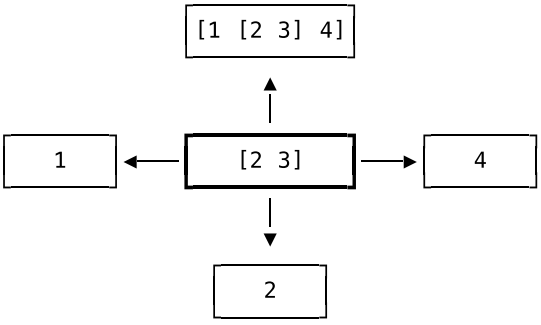


Чтобы перейти на **четвёрку** из текущей позиции, сначала поднимемся вверх. Указатель сдвинется на вектор `[2 3]`. Мы находимся среди потомков исходного вектора и можем перемещаться по горизонтали. Сделаем шаг вправо и окажемся на цифре 4.

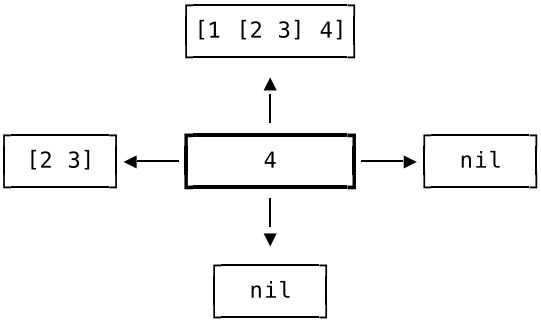
То же самое графически. Текущая локация (тройка):



Шаг вверх:



Шаг вправо:



Исходный вектор может быть любой вложенности. Ради интереса замените данные на `[5 [6 [7 [8] 9]]]` и проследуйте до девятки.

Что случится, если передать в `vector-zip` что-то отличное от вектора? Предположим, `nil`, строку или число. Перед тем как

двигаться, зиппер проверит, подходит ли узел на роль ветки. Сработает функция `vector?`, которая вернёт `nil` для всех отличных от вектора значений. В результате получим локацию, из которой нельзя никуда шагнуть: ни вниз, ни в стороны. Это тупиковый случай, и его нужно избегать.

```
(-> "test"
    zip/vector-zip
    zip/down)
nil
```

Модуль `clojure.zip` предлагает и другие встроенные зипперы. Особенно интересен `xml-zip` для навигации по XML-дереву. Мы обсудим его отдельно, когда читатель познакомится с другими свойствами зипперов.

1.2 Автонавигация

Мы разобрались с тем, как перемещаться по коллекции. Однако у читателя возникнет вопрос: как мы узнаем заранее, куда двигаться? Откуда приходит путь?

Ответ покажется странным, но всё же: ручная навигация по данным лишена всякого смысла. Если путь известен заранее, вам не нужен зиппер — это лишнее усложнение.

Clojure предлагает более простую работу с данными, структура которых известна. Например, если мы точно знаем, что на вход поступил вектор, второй элемент которого — вектор, и нужно взять его второй элемент, то воспользуемся `get-in`:

```
(def data
  [1 [2 3] 4])

(get-in data [1 1])
;; 3
```

То же самое касается других типов данных. Не важно, какую комбинацию образуют списки и словари: если структура известна заранее, до элемента легко добраться с помощью `get-in` или стрелочного оператора. В данном случае зипперы только усложнят код.

```
(def data
  {:users [{:name "Ivan"}]})

(-> data :users first :name)
;; "Ivan"
```

В чем же тогда преимущество зипперов? Свои сильные стороны они проявляют там, где `get-in` не работает. Речь о данных с **неизвестной** структурой. Представьте, что на вход поступил произвольный вектор и нужно найти в нём строку. Она может быть как на поверхности вектора, так и вложена на три уровня. В этом случае `get-in` не поможет, потому что мы не знаем путь.

Другой пример — XML-документ. Нужный тег может располагаться где угодно, и нужно как-то его найти. Таким образом, идеальный случай для зиппера — нечёткая структура данных, о которой у нас только предположения.

Функции `zip/up`, `zip/down` и другие образуют универсальную `zip/next`. Эта функция передвигает указатель так, что рано или поздно мы обойдём всю структуру. При обходе исключены повторы: мы побываем в каждом месте только раз. Пример с вектором:

```
(def vzip
  (zip/vector-zip [1 [2 3] 4]))

(-> vzip zip/node)
;; [1 [2 3] 4]

(-> vzip zip/next zip/node)
;; 1

(-> vzip zip/next zip/next zip/node)
;; [2 3]

(-> vzip zip/next zip/next zip/next zip/node)
;; 2
```

Очевидно, мы не знаем, сколько раз вызывать `zip/next`, поэтому пойдём на хитрость. Функция `iterate` принимает функцию `f` и значение `x`. Результатом станет последовательность, где первый

элемент `x`, а каждый следующий — `f(x)` от предыдущего. Для зиппера мы получим исходную локацию, затем `zip/next` от неё, потом `zip/next` от прошлого шага и так далее.

Переменная `loc-seq` ниже — это цепочка локаций исходного зиппера. Чтобы получить узлы, мы берём шесть первых элементов (число взяли случайно) и вызываем для каждого `zip/node`.

```
(def loc-seq
  (iterate zip/next vzip))

(->> loc-seq
     (take 6)
     (map zip/node))

;; ([1 [2 3] 4], 1, [2 3], 2, 3, 4)
```

`Iterate` порождает **ленивую** и **бесконечную** последовательность. Обе характеристики важны. Ленивость означает, что очередной сдвиг (вызов `zip/next`) не произойдёт до тех пор, пока вы не дойдёте до элемента в цепочке. Бесконечность означает, что `zip/next` вызывается неограниченное число раз. Понадобится признак, по которому мы остановим вызов `zip/next`, иначе локации никогда не кончатся.

Если исследовать `loc-seq`, станет ясно, что в какой-то момент `zip/next` уже не сдвигает указатель. Возьмём наугад сотый и тысячный элементы итерации. Их узел будет исходным вектором:

```
(-> loc-seq (nth 100) zip/node)

;; [1 [2 3] 4]

(-> loc-seq (nth 1000) zip/node)

;; [1 [2 3] 4]
```

Причина кроется в устройстве зиппера. Функция `zip/next` работает по принципу кольца. Когда она достигает исходной локации, цикл завершается. При этом локация получит признак завершения, и дальнейший вызов `zip/next` вернёт её же. Проверить признак можно функцией `zip/end?`:

```
(def loc-end
  (-> [1 2 3]
      zip/vector-zip
      zip/next
      zip/next
      zip/next))

loc-end
;; [[1 2 3] :end]

(zip/end? loc-end)
;; true
```

Чтобы получить конечную цепь локаций, будем сдвигать указатель до тех пор, пока локация не последняя. Всё вместе даёт функцию `iter-zip`:

```
(defn iter-zip [zipper]
  (->> zipper
       (iterate zip/next)
       (take-while (complement zip/end?))))
```

Функция вернёт все локации от начальной до конечной, не включая её. Напомним, что локация хранит узел (элемент данных), который можно извлечь с помощью `zip/node`. Код ниже показывает, как превратить локации в данные:

```
(->> [1 [2 3] 4]
     zip/vector-zip
     iter-zip
     (map zip/node))

;; ([1 [2 3] 4], 1, [2 3], 2, 3, 4)
```

Теперь, когда мы получили цепочку локаций, напишем поиск. Предположим, нужно проверить, есть ли в векторе кейворд `:error`. Сначала напишем предикат для локации — равен ли её узел этому значению:

```
(defn loc-error? [loc]
  (-> loc zip/node (= :error)))
```

Проверим, если ли среди локаций та, что подходит нашему предикату. Для этого вызовем `some`:

```
(def data
  [1 [2 3 [:test [:foo :error]]] 4])

(some loc-error? (-> data
                     zip/vector-zip
                     iter-zip))
;; true
```

Из-за ленивости мы не сканируем вектор целиком. Если нужный узел нашелся на середине, `iter-zip` прекращает итерацию, и дальнейшие вызовы `zip/next` не сработают.

Полезно знать, что `zip/next` обходит дерево в глубину. При движении он стремится вниз и влево, а поднимается, лишь когда шаги в эти стороны невозможны. Как мы увидим далее, в некоторых случаях порядок обхода важен. Попадаются задачи, где мы должны двигаться вширь. По умолчанию в `clojure.zip` нет других вариантов обхода, но мы напишем собственный. Также мы рассмотрим задачу, где понадобится обход в ширину.

Встроенный зиппер `vector-zip` служит для вложенных векторов. Но гораздо чаще встречаются вложенные словари. Напишем зиппер для обхода данных, как в примере ниже:

```
(def map-data
  {:foo 1
   :bar 2
   :baz {:test "hello"
         :word {:nested true}}})
```

За основу возьмём знакомый нам `vector-zip`. Зипперы похожи, разница лишь в типе коллекции. Подумаем, как задать функции `branch?` и `children`. Сам по себе словарь — это ветка, чьи потомки — элементы `MapEntry`. Тип `MapEntry` содержит ключ и значение. Если значение — словарь, получим из него цепочку вложенных `MapEntry` и так далее.

Для разминки напишем проверку на тип `MapEntry`:

```
(def entry?
  (partial instance? clojure.lang.MapEntry))
```

Зиппер `map-zip` выглядит так:

```
1  (defn map-zip [mapping]
2    (zip/zipper
3     (some-fn entry? map?)
4     (fn [x]
5       (cond
6         (map? x)
7         (seq x)
8         (and (entry? x) (-> x val map?))
9         (-> x val seq)))
10    nil
11    mapping))
```

Поясним основные моменты. Композиция `some-fn` (строка 3) вернёт истину, если хотя бы один из предикатов сработает положительно. Иными словами, на роль ветки мы рассматриваем только словарь или его узел (пару ключ-значение).

Во второй функции, которая ищет потомков, приходится делать перебор. Для словаря (проверка `map?`) получим потомков функцией `seq` — она вернёт цепочку элементов `MapEntry`. Если текущий элемент — `MapEntry`, проверим, является ли его значение вложенным словарём (функция `val` вернёт второй элемент `MapEntry`). Если да, получим потомков той же функцией `seq`.

Обход зиппера вернёт все пары ключей и значений. Если значение — вложенный словарь, мы провалимся в него при обходе. Пример:

```
(->> {:foo 42
      :bar {:baz 11
            :user/name "Ivan"}}
     map-zip
     iter-zip
     rest
     (map zip/node))


([:foo 42]
 [:bar {:baz 11, :user/name "Ivan"}]
 [:baz 11]
 [:user/name "Ivan"])
```

Обратите внимание на функцию `rest` после `iter-zip` (строка 6). Мы отбросили первую локацию, в которой находятся исходные данные. Они известны, поэтому нет смысла печатать их.

С помощью нашего `map-zip` легко проверить, есть ли в словаре ключ `:error` со значением `:auth`. По отдельности эти кейворды могут быть где угодно — в ключах или значениях на любом уровне. Однако нас интересует их комбинация. Для этого напишем предикат:

```
(defn loc-err-auth? [loc]
  (-> loc zip/node (= [:error :auth])))
```

Убедимся, что в первом словаре нет пары, даже несмотря на то, что значения встречаются по отдельности:

```
(->> {:response {:error :expired
                 :auth :failed}}
     map-zip
     iter-zip
     (some loc-err-auth?))

;; nil
```

Но даже если пара вложена глубоко, мы найдём её:

```
(def data
  {:response {:info {:message "Auth error"
                     :error :auth
                     :code 1005}}})

(->> data
     map-zip
     iter-zip
     (some loc-err-auth?))

;; true
```

Предлагаем читателю несколько заданий для самостоятельной работы.

1. Зиппер `map-zip` не учитывает случай, когда ключ словаря — другой словарь, например:

   ```
   {{:alg "MD5" :salt "..."} "deprecated"
    {:alg "SHA2" :salt "..."} "deprecated"
    {:alg "HMAC-SHA256" :key "..."} "ok"}
   ```

   Такие коллекции хоть и редко, но встречаются в практике. Доработайте `map-zip`, чтобы он проверял не только значение `MapEntry`, но и ключ (вместо `val` используйте `key`).

2. Мы рассмотрели зипперы для векторов и словарей по отдельности. На практике работают со смешанными данными, когда словари и векторы вложены друг в друга. Напишите универсальный зиппер, который учитывает обе коллекции при обходе.

1.3 XML-зипперы

Мощь зипперов раскрывается в полной мере при работе с XML. От других форматов он отличается тем, что задан рекурсивно. Например, в JSON объект и массив выглядят по-разному. То же самое относится с YAML, где у каждого типа свои синтаксис и структура. В XML, где бы мы ни находились, текущий узел состоит из трёх элементов: тега, атрибутов и содержимого.

Тег — это короткое имя узла, например `name` или `description`. Атрибуты — словарь свойств и значений. Наиболее интересно содержимое: это набор строк или других узлов. Вот как выглядит XML на псевдокоде:

```
XML = [Tag, Attrs, [String|XML]]
```

Чтобы убедиться в однородности XML, рассмотрим файл с товарами поставщиков. На вершине XML находится узел `catalog`. Это группировочный тег: он необходим, потому что на вершине не может быть несколько тегов. Потомки каталога — организации. В атрибуте `name` организации указано её имя. Под организацией идут товары — узлы с тегом `product` и атрибутом `type`. Вместо потомков товар содержит текст — его наименование. Ниже товара спуститься уже нельзя.

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <organization name="re-Store">
    <product type="iphone">iPhone 11 Pro</product>
    <product type="iphone">iPhone SE</product>
  </organization>
  <organization name="DNS">
    <product type="tablet">iPad 3</product>
    <product type="notebook">Macbook Pro</product>
  </organization>
</catalog>
```

Clojure предлагает XML-парсер, который вернёт структуру, похожую на схему `[Tag, Attrs, Content]` выше. Каждый узел станет словарем с ключами `:tag`, `:attrs` и `:content`. Последний хранит вектор, где элемент — либо строка, либо вложенный словарь.

Поместим XML с товарами в файл `resources/products.xml`. Напишем функцию `->xml-zipper`, чтобы считать файл в зиппер. Добавьте модули `xml` и `io`:

```
(:require
 [clojure.java.io :as io]
 [clojure.xml :as xml])
```

Оба входят в поставку Clojure и не требуют зависимостей. Чтобы получить зиппер, пропустим параметр `path` через серию функций:

```
(defn ->xml-zipper [path]
  (-> path
      io/resource
      io/file
      xml/parse
      zip/xml-zip))
```

Функция `xml/parse` вернёт структуру словарей с ключами `:tag`, `:attrs` и `:content`. Обратите внимание, что текстовое содержимое (например, название товара) — это тоже вектор с одной строкой. Тем самым достигается однородность узлов на всех уровнях.

Вот что получим после вызова `xml/parse`:

```
{:tag :catalog
 :attrs nil
 :content
 [{:tag :organization
   :attrs {:name "re-Store"}
   :content
   [{:tag :product
     :attrs {:type "iphone"}
     :content ["iPhone 11 Pro"]}
    {:tag :product
     :attrs {:type "iphone"}
     :content ["iPhone SE"]}]}
  {:tag :organization
   :attrs {:name "DNS"}
   :content
   [{:tag :product
     :attrs {:type "tablet"}
     :content ["iPad 3"]}
    {:tag :product
     :attrs {:type "notebook"}
     :content ["Macbook Pro"]}]}]}
```

Вызов `(->xml-zipper "products.xml")` вернёт первую локацию зиппера XML. Прежде чем работать с ним, заглянем в определение `zip/xml-zip`, чтобы понять, что происходит. Приведём его код в сокращении:

```
(defn xml-zip [root]
  (zipper (complement string?)
          (comp seq :content)
          ...
          root))
```

Очевидно, потомки узла — это его содержимое `:content`, дополнительно обёрнутое в `seq`. У строки не может быть потомков, поэтому `(complement string?)` означает: искать потомков в узлах, отличных от строки.

Рассмотрим, как бы мы нашли все товары из заданного XML. Для начала получим ленивую итерацию по зипперу. Напомним,

что на каждом шаге мы получим не словарь с полями `:tag` и другими, а локацию с указателем на него. Останется выбрать локации, чьи узлы содержат тег `product`. Для этого напишем предикат:

```
(defn loc-product?
  [loc]
  (-> loc zip/node :tag (= :product)))
```

Выборка с преобразованием:

```
(->> "products.xml"
     ->xml-zipper
     iter-zip
     (filter loc-product?)
     (map loc->product))

;; ("iPhone 11 Pro"
;;  "iPhone SE"
;;  "iPad 3"
;;  "Macbook Pro")
```

На первый взгляд здесь ничего особенного. Структура XML известна заранее, поэтому можно обойтись без зиппера. Для этого выберем потомков каталога и получим организации; из потомков организаций получим товары. Вместе получится простой код:

```
(def xml-data
  (-> "products.xml"
      io/resource
      io/file
      xml/parse))

(def orgs
  (:content xml-data))

(def products
  (mapcat :content orgs))

(def product-names
  (mapcat :content products))
```

Для краткости уберём переменные и сведём код к одной форме:

```
(->> "products.xml"
     io/resource
     io/file
     xml/parse
     :content
     (mapcat :content)
     (mapcat :content))

;; ("iPhone 11 Pro"
;;  "iPhone SE"
;;  "iPad 3"
;;  "Macbook Pro")
```

На практике структура XML неоднородна. Предположим, крупный поставщик разбивает товары по филиалам. В его случае XML выглядит так (фрагмент):

```
<organization name="DNS">
  <branch name="Office 1">
    <product type="tablet">iPad 3</product>
    <product type="notebook">Macbook Pro</product>
  </branch>
</organization>
```

Код выше, где мы слепо выбираем данные по уровню, сработает неверно. В списке товаров окажется узел:

```
("iPhone 11 Pro"

 "iPhone SE"

 {:tag :product
  :attrs {:type "tablet"}
  :content ...})
```

В то время как зиппер вернёт **только** товары, в том числе из филиала:

```
(->> "products-branch.xml"
     ->xml-zipper
     iter-zip
     (filter loc-product?)
     (map loc->product))

;; ("iPhone 11 Pro"
;;  "iPhone SE"
;;  "iPad 3"
;;  "Macbook Pro")
```

Очевидно, выгодно пользоваться кодом, который работает с обоими XML, а не поддерживать отдельную версию для крупного поставщика. В противном случае нужно где-то хранить признак и делать по нему `if/else`, что усложнит проект.

Однако этот пример не раскрывает всю мощь зипперов. Для обхода XML уже есть готовое средство — функция `xml-seq` из главного модуля Clojure. Она возвращает ленивую цепочку XML-узлов в том же виде (словари с ключами `:tag`, `:attr` и `:content`). `Xml-seq` — это частный случай более абстрактной функции `tree-seq`. Последняя похожа на зиппер тем, что принимает функции `branch?` и `children`, чтобы определить, подходит ли узел на роль ветки и как извлечь потомков. Определение `xml-seq` напоминает `xml-zip`:

```
(defn xml-seq
  "A tree seq on the xml elements as per xml/parse"
  {:added "1.0"
   :static true}
  [root]
  (tree-seq (complement string?)
            (comp seq :content)
            root))
```

Разница между зиппером и `tree-seq` в том, что при обходе зиппер возвращает локацию — элемент, в котором больше сведений. Кроме данных, он содержит ссылки на элементы по всем четырём направлениям. Наоборот, `tree-seq` итерирует данные без обёрток. Для обычного поиска `tree-seq` даже предпочтительней, поскольку не порождает лишних абстракций. Вот как выглядит сбор товаров с учётом филиалов:

```
(defn node-product? [node]
  (some-> node :tag (= :product)))

(->> "products-branch.xml"
     io/resource
     io/file
     xml/parse
     xml-seq
     (filter node-product?)
     (mapcat :content))

("iPhone 11 Pro" "iPhone SE" "iPad 3" "Macbook Pro")
```

Чтобы показать мощь зипперов, подберём такую задачу, где возможностей `tree-seq` не хватает. На эту роль подойдёт поиск с переходом между локациями.

1.4 Поиск в XML

Предположим, нам поставили задачу: выбрать из XML магазины, где продаются айфоны. Обратите внимание: мы впервые коснулись связи между узлами, и это важно. По отдельности выбрать данные легко. Магазины — это локации, у которых тег `organization`. Айфоны — локации, в которых узел с тегом `product` и атрибутом `type="iphone"`. Но как найти связь между ними?

В прошлый раз мы разложили XML в последовательность с помощью `xml-seq`. Проблема в том, что функция порождает коллекцию узлов без какой-либо связи, что не даёт нам решить задачу. Покажем это на примере. Для начала получим цепочку узлов:

```
(def xml-nodes
  (->> "products-branch.xml"
       io/resource
       io/file
       xml/parse
       xml-seq))
```

Вообразим, что в одном из элементов находится нужный товар. Например, мы встретили айфон в третьем (втором от нуля) узле:

```
(-> xml-nodes (nth 2))
;; {:tag :product
;;  :attrs {:type "iphone"}
;;  :content ["iPhone 11 Pro"]}
```

Однако не ясно, как найти магазин, которому он принадлежит. Можно предположить, что магазин находится слева от товара, потому что предшествует ему при обходе. Это станет ясно, если напечатать теги узлов:

```
(->> xml-nodes
     (mapv :tag)
     (remove nil?)
     (run! print))

;; :catalog :organization :product :product ...
```

Идея верная, но в целом слабая, потому что зависит от порядка обхода. Кроме того, задача усложняется: при обходе нужно не просто выбрать нужные товары, но и переместиться назад в поисках магазина. Затем продолжить вперед и при этом пропустить найденный товар, чтобы не попасть в вечный цикл.

Здесь и приходит на помощь зиппер. Локация, которую он возвращает на каждом шаге, помнит положение в структуре. От локации можно пройти в нужное место с помощью функций `zip/up`, `zip/right` и других, что мы рассмотрели в начале главы. Это тот случай, когда ручная навигация оправдана.

Вернёмся к XML со структурой «каталог — организация — товары». Освежим его в памяти:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <organization name="re-Store">
    <product type="iphone">iPhone 11 Pro</product>
    <product type="iphone">iPhone SE</product>
  </organization>
  <organization name="DNS">
    <product type="tablet">iPad 3</product>
    <product type="notebook">Macbook Pro</product>
  </organization>
</catalog>
```

Прежде всего найдём локации-айфоны. Напишем предикат на айфон:

```
(defn loc-iphone? [loc]
  (let [node (zip/node loc)]
    (and (-> node :tag (= :product))
         (-> node :attrs :type (= "iphone")))))
```

Получим локации с айфонами:

```
(def loc-iphones
  (->> "products.xml"
       ->xml-zipper
       iter-zip
       (filter loc-iphone?)))

(count loc-iphones)
2
```

Теперь, чтобы найти организацию по товару, поднимемся на уровень выше с помощью `zip/up`. Это верно, потому что организация — родитель товара:

```
(def loc-orgs
  (->> loc-iphones
       (map zip/up)
       (map (comp :attrs zip/node))))

({:name "re-Store"}
 {:name "re-Store"})
```

Для каждого айфона мы нашли организацию, где его продают. Получились дубли, потому что оба айфона продаются в re-Store. Чтобы избавиться от повторов, оберните результат в `set`.

```
(set loc-orgs)

#{{:name "re-Store"}}
```

Это и есть ответ на вопрос: айфоны можно купить в магазине re-Store. Если добавить айфон в организацию DNS, она тоже появится в `loc-orgs`.

Решим ту же задачу для XML с филиалами. Теперь мы не можем вызвать `zip/up` для товара, чтобы получить организацию, потому что в некоторых случаях получим филиал, и понадобится еще один шаг вверх. Чтобы не гадать, сколько раз подниматься, напишем функцию `loc->org`. Она вызывает `zip/up` до тех пор, пока не выйдет на нужный тег:

```
(defn loc-org? [loc]
  (-> loc zip/node :tag (= :organization)))

(defn loc->org [loc]
  (->> loc
       (iterate zip/up)
       (find-first loc-org?)))
```

Функция `find-first` находит первый элемент коллекции, который подошёл предикату. Она ещё не раз пригодится нам.

```
(defn find-first [pred coll]
  (some (fn [x]
          (when (pred x)
            x))
        coll))
```

Чтобы сократить код, не будем объявлять переменные `loc-iphones` и другие. Выразим поиск одной формой:

```
1  (->> "products-branch.xml"
2       ->xml-zipper
3       iter-zip
4       (filter loc-iphone?)
5       (map loc->org)
6       (map (comp :attrs zip/node))
7       (set))
8
9  ;; #{{:name "re-Store"}}
```

Новое решение отличается лишь тем, что мы заменили `zip/up` на функцию с более сложным восхождением (строка 5). В остальном ничего не изменилось.

Обратите внимание, насколько удобен XML в плане поиска и навигации. Если хранить данные в JSON, их структура и обход отличаются кардинально. Это будут различные комбинации векторов и объектов, которые трудно подчинить единым правилам. Для работы с ними нужен разный код. В случае с XML его структура однородна: добавление филиала меняет лишь глубину товаров, но не правила обхода.

Усложним задачу: среди обычных товаров встречаются наборы (bundle). Товар из набора нельзя купить отдельно. Например, тряпочка для экрана продаётся чаще всего с устройством (строка 10). Нас просят найти магазин, где тряпочку можно купить отдельно (строка 4). Пример XML, в котором мы будем искать:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <catalog>
3    <organization name="re-Store">
4      <product type="fiber">VIP Fiber Plus</product>
5      <product type="iphone">iPhone 11 Pro</product>
6    </organization>
7    <organization name="DNS">
8      <branch name="Office 2">
9        <bundle>
10         <product type="fiber">Premium iFiber</product>
11         <product type="iphone">iPhone 11 Pro</product>
12       </bundle>
13     </branch>
14   </organization>
15 </catalog>
```

Сначала найдём все тряпочки. В них окажутся как отдельные товары, так и из набора:

```
(defn loc-fiber? [loc]
  (some-> loc
          zip/node
          (get-in [:attrs :type])
          (= "fiber")))
```

```
(->> "products-bundle.xml"
     ->xml-zipper
     iter-zip
     (filter loc-fiber?)
     (map (comp first :content zip/node)))

;; ("VIP Fiber Plus" "Premium iFiber")
```

Теперь решим задачу. Из найденных тряпочек отсекаем те, что входят в набор. С точки зрения зиппера это значит, что у родителя этой локации тег *не равен* `:bundle`. От оставшихся тряпочек переходим к магазинам.

Введём предикат `loc-in-bundle?` — входит локация в набор или нет:

```
(defn loc-in-bundle?
  [loc]
  (some-> loc
          zip/up
          zip/node
          :tag
          (= :bundle)))
```

Решение:

```
(->> "products-bundle.xml"
     ->xml-zipper
     iter-zip
     (filter loc-fiber?)
     (remove loc-in-bundle?)
     (map loc->org)
     (map (comp :attrs zip/node))
     (set))

#{{:name "re-Store"}}
```

Магазин DNS не попал в результат, потому что в нём тряпочка продаётся в наборе.

Новое усложнение: мы хотим купить айфон, *но только в наборе* с тряпочкой. В какой магазин направить покупателя?

Решение: сначала ищем все айфоны. Оставляем те, что входят в набор. Среди соседей айфона ищем тряпочку. Если нашли, переходим от айфона или тряпочки к магазину. Основные функции уже готовы: это предикаты на проверку набора, тип товара и другие мелочи. Но мы не рассмотрели, как получить соседей локации.

Функции `zip/lefts` и `zip/rights` вернут элементы по левую и правую стороны от текущей локации. Совместим их через `concat`, чтобы получить всех соседей:

```
(defn node-neighbors [loc]
  (concat (zip/lefts loc)
          (zip/rights loc)))
```

Уточним, что это будут именно элементы, а не локации. Покажем это на примере вектора и локации с числом 2. Её соседями будут числа 1 и 3:

```
(-> [1 2 3]
    zip/vector-zip
    zip/down
    zip/right
    node-neighbors)

;; (1 3)
```

Зиппер устроен так, что получить правые и левые элементы проще, чем передвигать локацию влево или вправо. Поэтому при поиске среди соседей выгодно работать с элементами, а не локациями.

Добавим функции, чтобы проверить, есть ли в соседях локации тряпочка:

```
(defn node-fiber?
  [node]
  (some-> node :attrs :type (= "fiber")))

(defn with-fiber?
  [loc]
  (let [nodes (node-neighbors loc)]
    (find-first node-fiber? nodes)))
```

Готовая функция:

```
(defn shops-by-iphone-with-fiber [filepath]
  (->> filepath
       ->xml-zipper
       iter-zip
       (filter loc-iphone?)
       (filter loc-in-bundle?)
       (filter with-fiber?)
       (map loc->org)
       (map (comp :name :attrs zip/node))
       (set)))

(shops-by-iphone-with-fiber "products-bundle.xml")

;; #{"DNS"}
```

В результате получим магазин DNS, потому что именно в нём айфон продается в комплекте с тряпочкой. Оба товара продаются в re-Store по отдельности, что не подходит условию задачи.

Можно добавить больше ограничений. Например, из найденных магазинов выбрать те, что расположены в радиусе 300 метров от покупателя. Для этого понадобятся расположение магазинов на карте и функция попадания точки в окружность. Можно выбрать только открытые магазины или те, что предлагают доставку. Запишем эти признаки в атрибуты организаций и добавим функции отбора.

Легко увидеть, что XML-зиппер стал настоящей базой данных. Он даёт ответы на сложные запросы, при этом код растёт медленнее, чем их смысловая нагрузка. Из-за рекурсивной структуры XML хорошо поддается обходу, и зипперы усиливают это преимущество. Обратите внимание, как удобно работают переходы и связи между узлами. Представьте, каких усилий стоило бы разбить данные на таблицы и писать SQL с оператором JOIN.

Конечно, по сравнению с настоящей базой данных у XML недостаток — в нём нет индексов, и поиск работает линейным перебором, а не как в бинарных деревьях. Кроме того, наш подход требует, чтобы данные находились в памяти целиком. Он не сработает для очень больших документов с миллионами записей, но пока что не будем волноваться об этом.