Иван Гришаев

# CLOJURE
## на производстве

Зипперы
Базы данных
REPL

Продолжение книги «Clojure на производстве» об одноименном языке. В этот раз мы рассмотрим зипперы, работу с базой данных и обширное понятие REPL. Материал нацелен на практику и отталкивается от реальных задач, которые ждут в повседневной работе.

Как и первый том, книга не содержит азов изучения Clojure. Ожидается, что читатель знаком с Clojure или другим промышленным языком. Для аудитории продвинутого уровня.

# 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 зипперах в Clojure. Это особый способ работы с коллекциями: непривычный, но крайне мощный. О зипперах 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

## 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» ввёл 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` создаёт 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 произойдёт, 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:

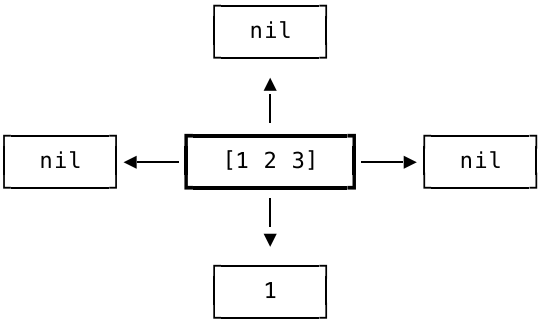
```
1 (defn vector-zip
2   [root]
3   (zipper vector?
4           seq
5           ...
6           root))
```

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

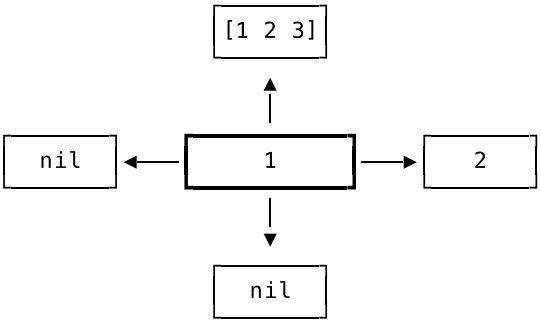
[3] github.com/clojure/clojure/blob/master/src/clj/clojure/zip.clj

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.

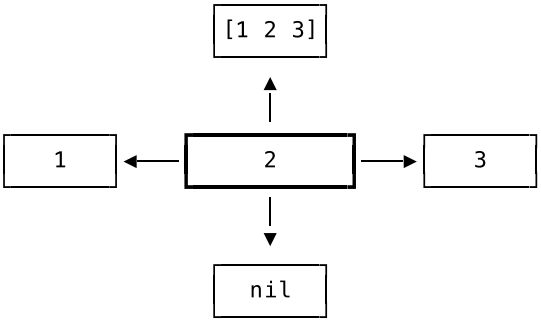
Покажем это на схеме. Начальная позиция, указатель на исходном векторе:




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.

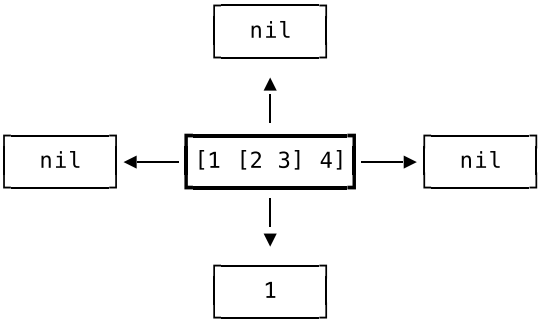
```
(-> [1 2 3]
    zip/vector-zip
    zip/down
    zip/left
    zip/down)

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

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

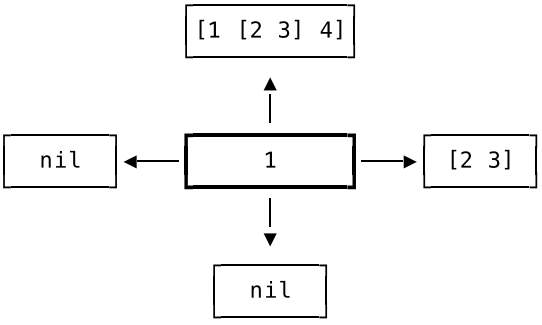
Рассмотрим случай, когда у вектора вложенные элементы: `[1 [2 3] 4]`. Чтобы переместиться на **тройку**, выполним шаги «вниз», «вправо», «вниз», «вправо». Сохраним локацию в переменную `loc3`:

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

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

(zip/node loc3)

;; 3
```

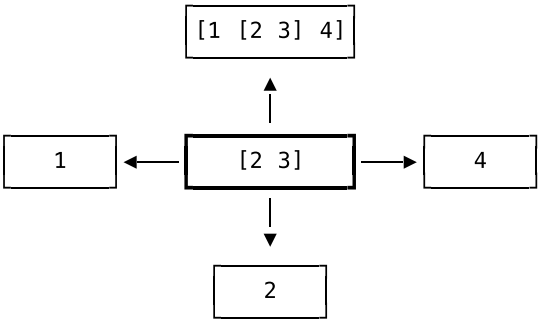
Рисунки ниже показывают, что происходит на каждом шаге. Исходная позиция:



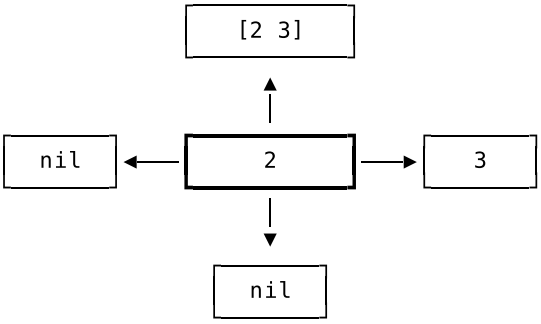

Шаг вниз:



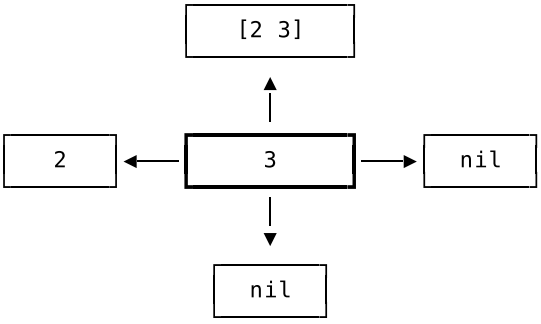
Вправо:




Вниз:

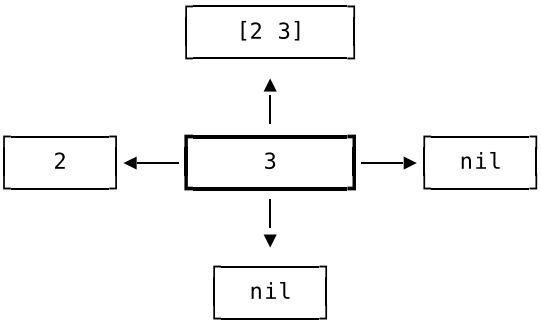



Вправо. Мы у цели:

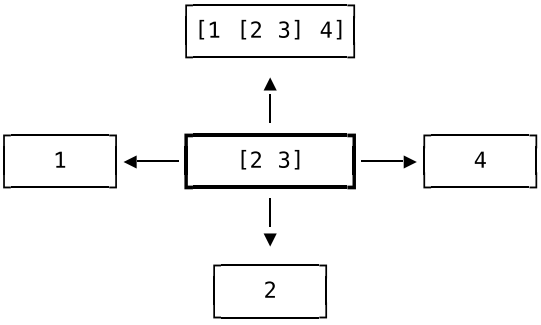



Чтобы перейти на **четвёрку** из текущей позиции, сначала поднимемся вверх. Указатель сдвинется на вектор `[2 3]`. Мы находимся среди потомков исходного вектора и можем перемещаться по горизонтали. Сделаем шаг вправо и окажемся на цифре 4.

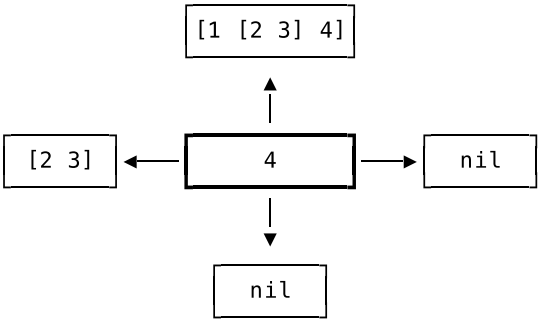
То же самое графически. Текущая локация (тройка):




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.

# Глава 2

# Реляционные базы данных

*В этой главе мы поговорим о реляционных базах данных в Clojure. Бóльшую часть описания займёт библиотека clojure.java.jdbc и дополнения к ней. Вы узнаете, какие проблемы встречаются в этой области и как их решают в Clojure.*

В разработке бэкенда базы данных занимают центральное место. Если говорить упрощённо, любая программа сводится к обработке данных. Конечно, они поступают не только из баз, но и из сети и файлов. Однако в целом за доступ к информации отвечают именно базы данных — специальные программы, сложные, но с богатыми возможностями.

Базы данных, или сокращенно БД, бывают разных видов. Они различаются в архитектуре и способе хранения информации. Некоторые базы работают только на клиенте, то есть в рамках одного компьютера. Другие хранят лишь текст и оставляют вывод типов на ваше усмотрение. Встречаются базы, где данные хранятся в оперативной памяти и пропадают после выключения.

Мы не ставим цель охватить как можно больше баз и способов работы с ними. Наоборот, сфокусируем внимание на том, что встретит вас в реальном проекте. Скорее всего, это будет классическая реляционная база данных вроде PostgreSQL или MySQL. О них мы и будем говорить.

Реляционная алгебра

Реляционные базы данных называют так из-за модели реляционной алгебры[1]. Это изящная математическая модель с набором операций: выборкой, проекцией, декартовым произведением и другими. Из модели следуют правила о том, как работает та или иная

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

Нормальные формы

операция. В свою очередь, из правил следуют нормальные формы[2] (первая, вторая и третья), доказать которые можно аналитически, а не на глаз.

Мы будем учить реляционную алгебру с самых азов. Обратитесь к статье в Википедии или книгам, где она описана без привязки к конкретной БД.

Перейдём к понятиям, более привычным программисту. Базы хранят содержимое в таблицах. Запись в таблице называется кортежем и состоит из отдельных полей. Поля могут быть разного типа. Состав полей и их порядок одинаковы в рамках таблицы. Не может быть так, что в первой записи два поля, а во второй три. Если нужно указать, что в поле нет значения, в него пишут специальное пустое, чаще всего NULL.

У записи есть особое поле, которое называют первичным ключом, Primary key. Ключ однозначно указывает на запись в таблице. Не может быть двух записей с одинаковым ключом. Чаще всего роль ключа играет число с автонумерацией, но иногда это строка, например электронная почта или артикул товара.

В редких случаях ключ может быть составным, то есть определяться парой полей, например тип платежной системы и номер платежа. В этом случае мы допускаем, что в таблице могут быть несколько платежей Stripe с разными номерами или платежи Stripe и AppStore, у которых платежи одинаковы, но не то и другое вместе.

```
 gateway   | trx_no |
-----------+--------
 stripe    | 1      | ;; ok
 stripe    | 2      | ;; ok
 stripe    | 3      | ;; ok
 appstore  | 1      | ;; ok
 appstore  | 2      | ;; ok
 appstore  | 2      | ;; error!
```

Внешним ключом называется поле, которые ссылается на первичный ключ другой таблицы. Для краткости его называют ссылкой (ref). Примером может быть поле `user_id` таблицы профилей, которое указывает на поле `id` таблицы пользователей (строка 7).

[2] en.wikipedia.org/wiki/Database_normalization

```
CREATE TABLE users(
    id   serial primary key,
    name text not null
);
CREATE TABLE profiles(
    id      serial primary key,
    user_id integer not null references users(id),
    avatar  text
);
```

Ссылка отличается от числа особыми свойствами: она гарантирует, что указанный пользователь действительно существует. Обычное число этой гарантии не дает. Говорят, что ссылки поддерживают целостность базы. Целостность означает: в базе нет ссылок на несуществующие записи.

Кроме целостности, внешний ключ поддерживает реакцию на удаление. По умолчанию нельзя удалить сущность, на которую кто-то ссылается. Однако можно задать правило, что связанная сущность тоже удаляется. Так, при удалении пользователя удалится и его профиль, ведь он не нужен в отрыве от пользователя. В другом случае ссылка на удалённую сущность станет NULL, если это разрешено в свойствах поля.

Из университета мы знаем, что связи бывают разных типов: один к одному или многим, многие ко многим. Профиль, который ссылается на пользователя, — это связь один к одному. Несколько заказов у пользователя — один ко многим. Тип связи легко задать ограничением на поле ссылки. Если в таблице профилей сделать поле `user_id` уникальным, не получится создать два профиля одному пользователю.

Связь «многие ко многим» строят через таблицу-мост со ссылками на другие таблицы. Чтобы сущности нельзя было соединить несколько раз, применяют составной первичный ключ из ссылок.

## 2.1 Запросы


SQL

База данных обращается с миром через SQL[3] (Structured Query Language) — структурированный язык запросов. Это текст, который описывает наши намерения — прочитать таблицу, добавить

[3] en.wikipedia.org/wiki/SQL

запись, обновить поле. Запросы имеют чёткую структуру, которая чаще всего зависит от главного оператора. К ним относятся `SELECT`, `INSERT`, `UPDATE`, `DELETE` и другие команды.

Существует несколько стандартов SQL, обозначенных годами, когда они были приняты: SQL'92, '99, '2003 и другие. Каждая база данных поддерживает стандарт определенного года, включая предыдущие. Кроме стандарта, базы предлагают расширения — возможности, которые не входят в него. При чтении документации обращайте внимание на то, относится ли конкретная возможность к стандарту или это частное решение.

В аббревиатуре SQL последняя буква означает language, язык. Однако это не язык программирования, потому что на нём нельзя выразить алгоритм задачи. Говоря точнее, SQL не полон по Тьюрингу: вы не сможете построить SQL-выражение на нём самом. Сложные запросы строят при помощи полноценных (полных по Тьюрингу) языков: Java, Python, Clojure.


SQL DO

Некоторые базы данных предлагают встроенные языки и операторы, чтобы это исправить. Так, в PostgreSQL доступен блок DO[4], где работают переменные, циклы и даже перехват исключений. Однако это частное решение, которое не входит в стандарт.

## 2.2 Доступ из Clojure

Теперь, когда мы освежили теорию, перейдём к практике. По умолчанию Clojure не предлагает доступа к базам данных. Чтобы работать с ними, подключают библиотеку clojure.java.jdbc — тонкую обертку над JDBC. Так называется встроенный в Java пакет для реляционных баз данных.


JDBC

JDBC[5] (Java Database Connectivity) отсчитывает свой возраст с 1997 года. Главная задача JDBC — предоставить общий API для разных баз. Для этого JDBC устроен из нескольких слоев. Потребители используют общий API, который, в зависимости от типа базы, вызывает разные драйверы.

Драйверы служат для связи JDBC с бэкендом. Каждый из них реализует бинарный протокол, по которому работает база. На сегодняшний день JDBC поддерживает все известные реляционные

[4] www.postgresql.org/docs/current/sql-do.html
[5] docs.oracle.com/javase/tutorial/jdbc/basics/index.html

СУБД: PostgreSQL, MySQL, Oracle, SQLite и другие. Существуют драйверы для файловых хранилищ, например Excel, DBF, CSV.

Для работы с какой-либо базой вам понадобится драйвер к ней. Драйверы поставляются отдельно и должны быть объявлены в зависимостях проекта. JDBC автоматически находит и подключает драйвер; программисту не нужно заботиться об этом.

Перечислим основные сущности JDBC:

- `DriverManager` — класс для управления драйвером конкретной БД. Получает соединение с базой, по которому в дальнейшем идёт обмен данными;
- `PreparedStatement` — подготовленное выражение. Так называется запрос, который прошел стадию подготовки и теперь может быть вызван с разными параметрами;
- `ResultSet` — источник, из которого читают результат запроса. Обычно приходит из метода `executeQuery` подготовленного выражения.

На следующей странице приведен код на Java, который выбирает записи из таблицы с помощью этих классов (листинг 2.1). Из-за многословности Java мы уменьшили шрифт и сократили некоторые конструкции, например заменили `System.out.println` на `S.o.println`.

В идеале при смене базы код останется прежним, а изменится только синтаксис SQL. И хотя это верно лишь отчасти, JDBC решает основную задачу — предлагает единый доступ к разным бэкендам.

Похожий стандарт существует в других языках, например Python. Там он называется DB API[6]. Его задача — обозначить минимальный набор правил, которым должен следовать драйвер. С этим требованием легче писать код для разных баз.


DB API

## 2.3 Знакомство с clojure.java.jdbc

Из примера на Java видно, что даже простой запрос требует многих усилий. В Clojure принято упрощать рутину, и библиотека

[6] www.python.org/dev/peps/pep-0249/

```
import java.sql.*;

public class JDBCExample {

  static String DB_URL = "jdbc:postgresql://127.0.0.1/test";
  static String USER = "book";
  static String PASS = "book";
  static String QUERY = "SELECT * FROM users";

  public static void main(String[] args) {
    try {
      Connection conn =
        DriverManager.getConnection(DB_URL, USER, PASS);
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(QUERY);
      while (rs.next()) {
        S.o.println("ID: " + rs.getInt("id"));
        S.o.println("First name: " + rs.getString("fname"));
        S.o.println("Last name: " + rs.getString("lname"));
        S.o.println("Email: " + rs.getString("email"));
        S.o.println("Age: " + rs.getInt("age"));
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Листинг 2.1. Запрос и печать результата в Java


java.jdbc

clojure.java.jdbc[7] создана именно для этого. Она создаёт тонкую обертку вокруг классов `DriverManager`, `PreparedStatement` и других, чтобы получить результат буквально парой строк. Добавьте библиотеку и драйвер PostgreSQL в проект:

```
;; project.clj
[org.clojure/java.jdbc "0.7.8"]
[org.postgresql/postgresql "42.1.3"]
```

Предположим, сервер PostgreSQL развернут локально. Вот как выполнить к нему запрос:

[7] github.com/clojure/java.jdbc

```
1 (def db {:dbtype "postgresql"
2          :dbname "test"
3          :host "127.0.0.1"
4          :user "book"
5          :password "book"})
6
7 (jdbc/query db "select 1 as value")
8
9 ;; ({:value 1})
```

Мы объявили параметры подключения словарём и передали его в функцию `jdbc/query`. Запрос возвращает постоянное число, поэтому он сработает, даже если в базе нет таблиц.

Пока мы не ушли вперёд, разберём, что произошло. Переменная `db` называется JDBC-спекой (строка 1). Это словарь с различными полями: хостом, портом, пользователем и другими. Из него библиотека получает соединение с базой. По завершении запроса соединение закрывается. JDBC-спека принимает разные формы, и позже мы рассмотрим их.

Зависимость `[org.postgresql/postgresql "42.1.3"]` нужна для работы с PostgreSQL. Её драйвер находится в репозитории Maven. Тот факт, что мы используем именно PostgreSQL, JDBC поймёт из поля `dbtype` со значением «postgresql». Он автоматически загрузит драйвер, так что импортировать его в Clojure не нужно.

Чтобы не распылять внимание на разные базы, остановим выбор на PostgreSQL до конца главы. В пользу PostgreSQL говорят зрелость и мощь проекта, его открытый код и бесплатный доступ. В России действует официальный вендор PostgresPro[8]. Их силами созданы полезные расширения, переведена документация на русский язык.


Postgres Pro

Чтобы запрос выполнился, должен работать локальный сервер PostgreSQL. Наиболее быстрый способ запустить его с нужными настройками — вызывать одноимённый образ Docker. Следующая команда:

```
> docker run --rm -it \
    -e POSTGRES_PASSWORD=pass \
    postgres
```

[8] postgrespro.ru

запустит сервер на локальном порту 5432. Имя базы, пользователь и пароль определяются переменными окружения; в примере выше пароль к базе будет «pass». Запуск образа мы обсуждали в прошлой книге о Clojure[9], а именно в четвёртой и седьмой главах. В случае трудностей изучите примеры оттуда.


I том

Если всё настроено правильно и вы увидели результат запроса, примите поздравления. Первый этап пройден, и вы готовы двигаться дальше.

## 2.4 Основы clojure.java.jdbc

Библиотека предлагает несколько функций для работы с базой. Рассмотрим наиболее важные из них.

### 2.4.1 Чтение

Чаще всего в коде используют `query` — функцию, которая выполняет запрос и возвращает результат. Она принимает jdbc-спеку и запрос. Последний может быть либо строкой:

```
(jdbc/query db "SELECT * FROM users")
```

либо вектором, где первый элемент — запрос с подстановками, а остальные элементы — параметры. Подобную запись называют SQL-вектором.

```
(jdbc/query db
            ["SELECT * FROM users WHERE id = ?" 1])
```

Знаки вопроса означают подстановку; позже параметры становятся на их место. Ниже мы узнаем, в чем разница между значением в запросе и параметром. Кроме запроса, `query` принимает необязательный параметр, который указывает, как обработать результат. Эти тонкости мы тоже обсудим позже.

Функция `get-by-id` предлагает доступ к записи по первичному ключу. Она ожидает имя таблицы и значение ключа. Если в таблице `users` первичный ключ называется `id`, то, чтобы получить первого пользователя, выполним:

[9] grishaev.me/clojure-in-prod/

```
(jdbc/get-by-id db :users 1)

{:id 1
 :fname "John"
 :lname "Smith"
 :email "test@test.com"
 :age 25}
```

Функция вернёт либо одну запись, либо `nil`. Это важное отличие от `query`, результат которой — всегда коллекция, в том числе для одной записи.

По умолчанию `get-by-id` считает, что первичный ключ называется `id`. Если это не так, укажите его имя после значения:

```
(jdbc/get-by-id db :users 1 "account_id")
```

Функция `find-by-keys` выполняет отбор по нескольким полям. Предположим, мы хотим выбрать пользователей с именем John и возрастом 25 лет. Для этого укажем в `find-by-keys` имя таблицы и словарь поле → значение:

```
(jdbc/find-by-keys db :users {:fname "John" :age 25})
```

База выполнит запрос:

```
SELECT * FROM users WHERE fname = $1 AND age = $2
parameters: $1 = 'John', $2 = '25'
```

Функция принимает любое число полей в словаре. Они соединяются оператором `AND`, потому что это наиболее частый случай. Для более тонкого отбора, например с помощью `OR`, функция не подойдёт. Позже мы рассмотрим, как строить такие запросы.

Заметим, что `find-by-keys` вернёт список, даже если нашлась только одна запись. В некоторых случаях мы намеренно ищем только первую. Заведём функцию `find-first` с теми же аргументами, которая оборачивает результат в `first`.

```
(defn find-first [db table filters]
  (first (jdbc/find-by-keys db table filters)))
```

В этом случае результат будет либо первой записью, либо `nil`:

```
(find-first db :users {:fname "John" :age 25})

{:id 1 :fname "John" :lname "Smith"
 :email "test@test.com" :age 25}
```

Это не оптимальное решение, потому что запрос к базе не содержит оператора `limit 1`. Без него база вернёт все записи, лишние из которых отбрасываются на стороне Clojure. Измените `find-first` так, чтобы база возвращала одну запись силами SQL.

### 2.4.2 Вставка

Функция `insert!` добавляет запись в таблицу. Восклицательный знак на конце означает, что вызов влечёт побочный эффект. `Insert!` принимает таблицу, в которую происходит запись. Значения могут быть переданы в разном виде. Наиболее частый сценарий — словарь поле → значение, например:

```
(jdbc/insert! db :users
              {:fname "Ivan"
               :lname "Petrov"
               :email "ivan@test.com"
               :age 87})
```

Во втором варианте функция принимает списки полей и значений по отдельности. Их длины должны быть равны:

```
(jdbc/insert! db :users
              [:fname :lname :email :age]
              ["Andy" "Stone" "andy@test.com" 33])
```

Оба вызова сводятся к подобному запросу:

```
INSERT INTO users ( fname, lname, email, age )
VALUES ( $1, $2, $3, $4 )
parameters:
  $1 = 'Andy', $2 = 'Stone',
  $3 = 'andy@test.com', $4 = '33'
```

# Глава 3

# REPL, Cider, Emacs

*Эта глава расскажет о REPL — главной особенности Clojure. Так называют интерактивную работу с языком, когда код наращивают постепенно. Мы рассмотрим, что такое REPL-driven development и почему, однажды узнав, от него тяжело отказаться.*

Сочетание REPL происходит от четырех слов: Read, Eval, Print и Loop. Дословно они означают прочитать, выполнить, напечатать и повторить. REPL — устойчивый термин, под которым понимают интерактивный режим программы.

REPL доступен не только в Clojure, но и в других языках. Как правило, он запускается, если вызвать интерпретатор без параметров. Команды `python` или `node` запустят интерактивные сеансы Python и Node.js. В Ruby для этого служит утилита `irb` (где i означает interactive). REPL поддерживают не только интерпретаторы, но и языки, которые компилируются в байт-код (Java, Scala) или машинный код (Haskell, SBCL).

Несмотря на разнообразие, именно в Лиспе REPL имеет решающее значение. Если в Python или Node.js он считается дополнением, то в Лиспе он необходим. Разработка на любом Лиспе зависит от того, насколько хорошо вы взаимодействуете с REPL. На REPL опираются все инструменты и практики, документация, уроки и так далее.

В мире Лиспа принято понятие REPL-driven development. Это стиль разработки, когда код пишут малыми порциями и запускают в REPL. С таким подходом сразу видно поведение программы. Легче проверить неочевидные случаи, например вызвать функцию с `nil` или обратиться к ресурсу, которого не существует.

REPL полезен в работе с данными по сети, например извлечь что-то из источника и исследовать результат. Эта задача идеально ложится на интерактивный режим. Как правило, обращение к источнику предполагает несколько этапов: подготовку запроса, отправку, чтение ответа и поиск нужных полей. Эти шаги проходят интерактивно методом проб и ошибок. Позже мы рассмотрим пример HTTP-запроса в сеансе REPL.

## 3.1 Исторический экскурс


Lisp machine

REPL отсчитывает историю от первых Лисп-машин[1]. Это были мейнфреймы с запущенным на них интерпретатором Лиспа. Подобные машины использовали в Xerox для печати, обработки изображений, управления оборудованием, решения задач на оптимизацию и машинного обучения. Разработку Лисп-машин поддерживал отдел DARPA[2], в том числе потому, что их применяли в военной отрасли.


DART

Золотой век Лисп-машин пришёлся на период с 1977 по 1985 год, после чего их популярность пошла на спад. Из-за особенностей архитектуры они не могли конкурировать с процессором x86 и компилируемыми языками — в плане как цены, так и быстродействия. В итоге Лисп-машины ушли с рынка, но подход REPL, придуманный полвека назад, навсегда остался в индустрии.

Подход и вправду был инновационным. До него программу набирали в редакторе, компилировали и только потом запускали (для краткости опустим перфокарты и прочую рутину). Процесс был долгим и дорогим. Наоборот, Лисп-машина принимала код и выполняла его мгновенно. Для своего времени Лисп был очень высокоуровневым языком. На нём было легко выразить сложную логику, не отвлекаясь на низкоуровневые проблемы. Именно в Лиспе появился автоматический контроль за памятью и сборщик мусора. Всё это делало Лисп-машину идеальной площадкой для экспериментов.

Интерпретатор Лисп-машины был не просто программой по запуску кода. Фактически он был её операционной системой, потому что имел доступ к регистрам процессора, оперативной памяти и устройствам ввода-вывода. В REPL можно было просмотреть

[1] en.wikipedia.org/wiki/Lisp_machine
[2] en.wikipedia.org/wiki/Dynamic_Analysis_and_Replanning_Tool

все переменные и функции, переопределить и удалить их. Это свойство — полный контроль системой — тоже стало неотъемлемой частью REPL.

Clojure, как и другие диалекты Лиспа, активно поддерживает REPL и всё связанное с ним. Без знания REPL работа с Clojure неэффективна. Классический подход, когда сначала вы пишете программу, а потом запускаете, здесь не работает. Цель этой главы — показать практическую, REPL-ориентированную разработку, принятую в Clojure.

## 3.2 Пробуем REPL

Чтобы познакомиться с REPL, запустим его. Это можно сделать несколькими способами.


Leiningen

**Первый** — установить утилиту Leiningen[3] для управления проектами на Clojure. Инструкции по установке вы найдёте на официальном сайте. Когда утилита установлена, выполните в терминале:

```
> lein repl
```


Clojure CLI

**Второй** способ — установить набор утилит Clojure CLI[4]. На официальном сайте Clojure описана установка для Linux и MacOS. В системе появятся команды `clojure` и `clj`. Если вызвать любую из них, запустится REPL:

```
> clj
> clojure
```

**Третий** и устаревший способ — запустить архив jar командой `java`. Старые версии Clojure (до 1.8 включительно) состояли из одного файла, путь к которому передаётся в параметре `-jar`:

```
> java -jar clojure-1.8.0.jar
```

Вариант этой же команды, когда архив находится в classpath и явно указан класс `clojure.main`:

[3] leiningen.org/

[4] clojure.org/guides/install_clojure

```
> java -cp clojure-1.8.0.jar clojure.main
```

С версии 1.9 Clojure состоит из нескольких jar-файлов. Библиотека Clojure.spec, которую мы рассмотрели в первой книге, поставляется отдельно. Скачайте jar-файлы из репозитория Maven[5] в разделах `org.clojure/clojure` и `org.clojure/spec.alpha`. Далее выполните в терминале:


Maven

```
> java -cp clojure-1.11.1.jar:spec.alpha-0.3.218.jar \
       clojure.main
```

Запустив REPL любым из способов, вы увидите приглашение:

```
user=>
```

Слово `user` означает текущее пространство имен. Если не задано иное, REPL открывается в пространстве `user`. Позже мы узнаем, как задать другое пространство или переключить его.

Введите любое выражение на Clojure: число, строку в двойных кавычках или кейворд. Эти значения вычисляются сами в себя:

```
user=> 1
1

user=> :test
:test

user=> "Hello REPL!"
"Hello REPL!"
```

Задайте глобальную переменную:

```
user=> (def amount 3)
#'user/amount
```

и сошлитесь на неё в выражении:

```
user=> (+ amount 4)
7
```

[5] mvnrepository.com

Более сложный пример. Определите функцию `add`, которая складывает два числа. Введите её в одну строку:

```
user=> (defn add [a b] (+ a b))
#'user/add
```

и проверьте вызов:

```
user=> (add 2 3)
5
```

REPL поддерживает ввод нескольких строк за раз. Предположим, мы хотим задать функцию с переносом после параметров, чтобы код выглядел аккуратнее:

```
(defn add [a b]
  (+ a b))
```

Если напечатать `(defn add [a b]` и нажать ввод, по незакрытой скобке REPL определит, что выражение неполное. Ошибки не произойдёт, и следующая строка дополнит исходную. Как только скобки станут сбалансированы, REPL выполнит форму.

```
user=> (defn add [a b]
  #_=> (+ a b))
#'user/add
```

Подключите любой из модулей Clojure, например встроенный `clojure.string` для работы со строками:

```
user=> (require '[clojure.string :as str])
nil
```

С его помощью разбейте строку или выполните автозамену:

```
user=> (str/split "one two three" #"\s+")
["one" "two" "three"]
```

```
user=> (str/replace
          "Two minutes, Turkish!",
           #"Two" "Five")

;; "Five minutes, Turkish!"
```

Модуль `clojure.inspector` предлагает примитивный графический отладчик. Его функция `inspect-tree` принимает данные и выводит окно Swing с деревом папок. Значок папки означает коллекцию; если его раскрыть, появятся дочерние элементы с иконками файлов. Чтобы изучить переменные среды, выполните:

```
(require '[clojure.inspector :as insp])

(insp/inspect-tree (System/getenv))
```

Содержимое окна будет примерно таким:

```
{}
├─ JAVA_MAIN_CLASS_68934=clojure.main
├─ LC_TERMINAL=iTerm2
├─ COLORTERM=truecolor
├─ LOGNAME=ivan
├─ TERM_PROGRAM_VERSION=3.3.12
├─ PWD=/Users/ivan/work/book-sessions
└─ SHELL=/bin/zsh
```

Опробуйте случай с ошибкой: поделите число на ноль или сложите число с `nil`. REPL не завершится, но выведет исключение на экран:

```
user=> (/ 1 0)
Execution error (ArithmeticException) at ...
Divide by zero
```

Это правильное поведение: в разработке ошибки случаются часто, и нам бы не хотелось завершать JVM. Однако это справедливо только для сеанса REPL. В боевом запуске программы на Clojure ведут себя как обычно: если исключение не поймано, программа завершится с нулевым кодом.

По умолчанию REPL выводит краткое сообщение об ошибке. Последнее исключение остаётся в переменной `*e`. Исследуем её:

```
user=> *e

#error {
 :cause "Divide by zero"
 :via
 [{:type java.lang.ArithmeticException
   :message "Divide by zero"
   :at [c.l.Numbers divide "Numbers.java" 188]}]
 :trace
 [[clojure.lang.Numbers divide "Numbers.java" 188]
  [clojure.lang.Numbers divide "Numbers.java" 3901]
  ...
  [clojure.lang.AFn run "AFn.java" 22]
  [java.lang.Thread run "Thread.java" 829]]}
```

В первой книге мы рассмотрели, что можно сделать с исключением: напечатать в удобном виде, записать в лог, отправить в систему сборки ошибок.

Если исключения не было, результат остаётся в переменной `*1`. С ней легко избежать повторных вычислений. Это особенно полезно, когда выражение даёт объёмный результат:

```
(into {} (System/getenv))

{"HOME" "/Users/ivan"
 "LC_TERMINAL_VERSION" "3.3.12"
 "USER" "ivan"
 ...}
```

Чтобы не вычислять его повторно, введите:

```
(get *1 "USER")
;; "ivan"
```

Переменная `*1` полезна для записи в файл. Предположим, мы хотим сохранить переменные среды, чтобы исследовать позже. Для этого введите:

```
(into {} (System/getenv))

(spit "dump.edn" (pr-str *1))
```

На диске появится файл `dump.edn` с данными. Позже мы прочтем его комбинацией `slurp` и `read-string`:

```
user=> (read-string (slurp "dump.edn"))

{"HOME" "/Users/ivan"
 "LC_TERMINAL_VERSION" "3.3.12"
 "USER" "ivan"
 ...}
```

В REPL доступны три переменные результата: `*1`, `*2` и `*3`. С каждым вычислением результаты смещаются: последний будет в `*1`, предпоследний в `*2` и так далее. Покажем это на примере:

```
user=> 1
1

user=> 2
2

user=> 3
3

user=> (println *1 *2 *3)
3 2 1
```

Чтобы загрузить несколько определений, используйте функцию `load-file`. Она принимает один аргумент — путь к файлу с кодом на Clojure:

```
(load-file "my_functions.clj")
```

Эффект аналогичен тому, как если бы вы скопировали содержимое и вставили в REPL. В боевом коде `load-file` не используют, потому что такая загрузка делает код неочевидным: не ясно, откуда взялось то или иное определение. Но для экспериментов `load-file` подходит идеально.

REPL предлагает макросы для интроинспеции. Выражение `(doc ...)` напечатает справку указанной функции, например:

```
(doc assoc)
-------------------------
clojure.core/assoc
([map key val] [map key val & kvs])
  assoc[iate]. When applied to a map, returns a new map
    of the same (hashed/sorted) type, that contains the
    mapping of ...
```

А форма `(source ...)` — её исходный код (приведем в сокращении):

```
(source assoc)

(def
 ^{:arglists '([map key val] [map key val & kvs])
   :doc "..."
   :added "1.0"
   :static true}
 assoc
 (fn ^:static assoc
   ([map key val] (clojure.lang.RT/assoc map key val))
   ([map key val & kvs]
    (...)))) ;; truncated
```

REPL предлагает и другие возможности, о которых мы поговорим позже. Пока что завершите сеанс нажатием `Ctrl+D` или выполните `(quit)` либо `(exit)`.

## 3.3 Более сложный сценарий

REPL удобен не только для быстрых экспериментов; опытные разработчики проводят в нём часы и дни. Одна из причин в том, что REPL — лучший способ разведать ситуацию, когда вы не знаете точно, какие данные ожидать от внешних источников.

Предположим, мы пишем бота для Telegram, чтобы публиковать шутки для программистов. Понадобится сервис, который бы выступил в роли источника шуток. Быстрый поиск даёт нам сервис Joke API[6] с удобным API по протоколу HTTP.

Joke API

[6] jokeapi.dev/

Прежде чем писать бота, убедимся в работе сервиса. Для этого понадобятся HTTP-клиент и парсер JSON. Если вы запускаете REPL при помощи `lein`, создайте файл `project.clj` с содержимым:

```
(defproject repl-chapter "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.10.0"]
                 [clj-http "3.9.1"]
                 [cheshire "5.8.1"]])
```

Для утилит Clojure CLI файл `deps.edn` выглядит так:

```
{:deps
 {clj-http/clj-http {:mvn/version "3.9.1"}
  cheshire/cheshire {:mvn/version "5.8.1"}}}
```

Запустите REPL. Обе библиотеки, если ещё не были установлены локально, скачаются на ваш компьютер. Подключите их в сеансе:

```
(require '[clj-http.client :as client])
(require 'cheshire.core)
```

Подготовим словарь запроса. В нём мы указываем параметр `type` со значением `twopart`, чтобы шутка состояла из двух частей (подробнее об этом — ниже):

```
(def request
  {:url "https://v2.jokeapi.dev/joke/Programming"
   :method :get
   :query-params {:type "twopart"}
   :as :json})
```

Получим для него ответ:

```
(def response
  (client/request request))
```

Каждый шаг мы связываем с переменной при помощи `def`, чтобы позже сослаться на него. Такой стиль не подходит для промышленного кода, но приемлем в REPL. Поместим тело в отдельную переменную и напечатаем его:

# Послесловие

С момента выхода первой книги прошло три года, и в мире Clojure многое изменилось. Язык всё реже воспринимают как экзотику. Больше фирм замечены в поиске специалистов по Clojure, причем не только на Западе, но и в России. Clojure используют в крупных банках (NuBank, RBI). Растёт число пользователей в группе Clojurians[67] и Телеграм-канале clojure_ru[68].


clojurians.slack.com

Набирает популярность утилита Babashka для запуска скриптов на Clojure. Вокруг неё растёт экосистема со своими проектами и библиотеками. Babashka и GraalVM открыли для Clojure новую область применения: системное программирование и запуск в AWS Lambda. Автор мечтает развить эту тему в будущей книге.


@clojure_ru

Материал, что вы прошли в трёх главах, по праву считается сложным. Если вы разобрались в предмете, запустили код и выполнили хотя бы часть задач, ваш уровень действительно высок. Закрепите знания практикой: внедрите Clojure на текущей работе или перепишите личный проект.

Желаю читателю успеха во всех начинаниях.

*Иван Гришаев,*
*Россия,*
*2020–2023*

[67] clojurians.slack.com/
[68] t.me/clojure_ru

# Предметный указатель

## *Clojure на производстве*
## *Зипперы, базы данных, REPL*

Продолжение книги, изданной три года назад. Мы продолжим изучать Clojure — замечательный язык с акцентом на неизменяемость и асинхронность.

По структуре и изложению книга похожа на первый том. Мы подробно рассмотрим несколько тем, чередуя теорию с практикой. Вас ждут зипперы, базы данных и обширное понятие REPL.

Материал рассчитан на продвинутую аудиторию. Желательно, чтобы у вас был опыт работы хотя бы с одним из промышленных языков.

## *Об авторе*

Иван Гришаев — программист, последние восемь лет пишет на Clojure. Работает в стартапах. Любит языки семейства Лисп, редактор Emacs и систему вёрстки LaTeX. Ведёт блог о программировании grishaev.me.