Иван Гришаев

Clojure
на производстве

2020

Книга рассказывает о Clojure — современном диалекте Лиспа. Это функциональный язык с акцентом на неизменяемость и многопоточность. Он появился десять лет назад и постепенно набирает популярность в России. В семи главах мы рассмотрим, как работать с Clojure на производстве.

Эта книга не для тех, кто учит язык с нуля. Ожидается, что читатель знаком с Clojure или другим диалектом Лиспа. Чтобы лучше усвоить материал, желательно иметь опыт программирования. Для аудитории продвинутого уровня.

Вёрстка: Иван Гришаев
Свёрстано в LaTeX
Сборка: a2ddbf6 1592310405
Дата сборки: 16 июня 2020 г.

Оглавление

1 Веб-разработка **9**

1.1 Основы HTTP 10
1.2 HTTP в Clojure 13
1.3 Запросы и ответы 16
1.4 Маршруты 18
1.5 Middleware 25
1.6 Файлы и ресурсы 42
1.7 Стриминг и проксирование 44
1.8 Другие библиотеки 46
1.9 Заключение 46

2 Clojure.spec **49**

2.1 Типы и классы 50
2.2 Основы spec 51
2.3 Исключения 53
2.4 Спеки-коллекции 54
2.5 Вывод значений 57
2.6 Спеки-перечисления 59
2.7 Продвинутые техники 61
2.8 Логические пути 63
2.9 Обратное действие 64
2.10 Анализ ошибок 66
2.11 Понятные ошибки 68
2.12 Парсинг 80
2.13 Разбор кода (теория) 88
2.14 Спецификация функций 92
2.15 Повторное использование спек 95
2.16 Дополнения 96
2.17 Будущее спеки 97
2.18 Заключение 98

3 Исключения **99**
3.1 Основы исключений . . . 100
3.2 Цепочки и контекст . . . 102
3.3 Переходим к Clojure . . . 104
3.4 Подробнее о контексте . . . 107
3.5 Когда бросать исключения . . . 108
3.6 Подробнее о цепочках . . . 110
3.7 Печать исключений . . . 112
3.8 Логирование . . . 113
3.9 Сбор исключений . . . 117
3.10 Sentry и Ring . . . 120
3.11 Переходы по коду . . . 121
3.12 Finally и контекстный менеджер . . . 124
3.13 Исключения на предикатах . . . 126
3.14 Приёмы и функции . . . 129
3.15 Заключение . . . 132

4 Изменяемость **133**
4.1 Общие проблемы . . . 133
4.2 Атомы . . . 137
4.3 Volatile . . . 146
4.4 Переходные коллекции . . . 148
4.5 Переменные и alter-var-root . . . 153
4.6 Присваивание с set! . . . 160
4.7 Изменения в контексте . . . 163
4.8 Локальные переменные в контексте . . . 168
4.9 Глобальные изменения в контексте . . . 170
4.10 Заключение . . . 175

5 Конфигурация **177**
5.1 Постановка проблемы . . . 177
5.2 Семантика . . . 178
5.3 Цикл конфигурации . . . 179
5.4 Ошибки конфигурации . . . 180
5.5 Загрузчик конфигурации . . . 181
5.6 Подробнее о переменных среды . . . 186
5.7 Конфигурация в среде . . . 188
5.8 Недостатки среды . . . 190
5.9 Переменные среды в Clojure . . . 193
5.10 Простой менеджер конфигурации . . . 198
5.11 Чтение среды из конфигурации . . . 200
5.12 Короткий обзор форматов . . . 203

5.13 Промышленные решения 208
5.14 Заключение 214

6 Системы **217**
6.1 Подробнее о системе 217
6.2 Подготовка к обзору 219
6.3 Mount 222
6.4 Component 236
6.5 Integrant 262
6.6 Заключение 273

7 Тесты **275**
7.1 Основные понятия 275
7.2 Тесты в Clojure 283
7.3 Полезные практики 289
7.4 Фикстуры 296
7.5 Метки и селекторы 305
7.6 Проблема окружения 309
7.7 Тестирование веб-приложений 330
7.8 Тестирование систем 334
7.9 Интеграционные тесты 337
7.10 Другие решения 342
7.11 Заключение 347

Что дальше **349**

Предметный указатель **350**

Об этой книге

У вас в руках книга о языке программирования Clojure. Это современный диалект Лиспа на платформе JVM. От устаревших диалектов он отличается тем, что делает ставку на функциональный подход и неизменяемость данных. Язык устроен так, чтобы решать сложные задачи простым способом.

Эта книга — не перевод, она изначально написана на русском языке. Вы не найдёте тяжёлых предложений, в которых слышна английская речь. Вам не придётся читать «маркер» вместо «токен» и другую нелепицу. Термины написаны в том виде, чтобы быть понятными программисту.

В книге нет вводной части, где написано, что скачать и установить. Также мы не рассматриваем азы вроде чисел и строк. На тему введения в Clojure уже написаны статьи и посты в блогах. Будет нечестно предлагать материал, где половина повторяет сказанное ранее. Эта книга от начала и до конца — то, о чём ещё никто не писал.

Другое её достоинство — упор на практику. Примеры кода взяты из реальных проектов. Все техники и приёмы автор опробовал лично. В описании проблем мы отталкиваемся от того, что вас ждёт на производстве. Покажем, где теория расходится с практикой и что предпочесть в таком случае.

Коротко о том, что вас ждёт. Начнём с веб-разработки — вспомним протокол HTTP и как с ним работать в Clojure. Затем рассмотрим Clojure.spec — библиотеку для проверки данных. Третья глава расскажет про исключения, четвёртая — про изменяемые данные. Далее переходим к конфигурации. В шестой главе знакомимся с системами. В последней научимся писать тесты.

Даже если вы не любите Лисп и книга попала к вам случайно, не спешите её откладывать. Clojure — это новые правила и другой мир, а книга — шанс туда попасть. Может быть, Clojure изменит ваше мнение о программировании. Обнаружит вопросы там, где, казалось бы, всё решено.

Желаем читателю терпения, чтобы прочесть книгу до конца.

Благодарности

Спасибо стартапу Flyerbee, моей первой работе на Clojure. Именно там я закрепил скромные знания языка.

Я счастлив работать в компании Exoscale в окружении талантливых инженеров. Многие вещи, не только технические, я узнал в этом коллективе.

Спасибо Петру Маслову и Евгению Климову за крупные партии найденных опечаток. Алексей Шипилов внёс важное замечание к первой главе, Досбол Жантолин — к последней. Молодцы все, кто указал на ошибки в комментариях в блоге.

Обратная связь

Автор будет признателен за указанные опечатки и неточности. Присылайте их по адресу `ivan@grishaev.me`. Возможно, в промежутках между тиражами получится обновить макет, и следующий читатель не увидит ошибки, о которой вы сообщили. Автор учтёт все замечания при переводе на английский язык.

Глава 1

Веб-разработка

В первой главе мы рассмотрим, как писать веб-приложения на Clojure. Поговорим о передаче данных по протоколу HTTP. Какие абстракции над ним возводят и что предлагает Clojure. Чем хорош функциональный подход и почему разработка на нём удобнее.


Cognitect 2018

Каждый год компания Cognitect опрашивает[1] разработчиков на Clojure. Один из вопросов уточняет, в какой области вы работаете. В 2010 году под веб писала половина опрошенных. К 2018 году эта цифра выросла до 80%, что уже четыре человека из пяти. Похожую динамику показывают опросы StackOverflow[2]. Согласно им, всё больше инженеров переходят в веб из смежных областей.


Stack-Overflow 2018

Если вы найдёте работу на Clojure, скорее всего это будет веб-приложение. Мы специально не говорим «сайт», потому что термин уходит в прошлое. Сегодня веб-приложение — это не только текст с картинками. В широком плане это сложный обмен данными по HTTP.

Протокол служит для передачи разметки HTML, но со временем подошёл и для данных. Его дизайн оказался настолько гибким, что не пришлось менять стандарт. Прежде чем перейти к Clojure, освежим в памяти устройство протокола: из каких частей он состоит и как с ним работает сервер. Это важно, потому что языки и фреймворки меняются, а протокол нет.

[1] blog.cognitect.com/blog/2017/1/31/clojure-2018-results
[2] insights.stackoverflow.com/survey/2018

1.1 Основы HTTP


RFC 2616

Протокол HTTP работает поверх стека TCP/IP. В широком смысле протоколы — это соглашения о том, как обмениваться данными. Они записаны в официальных документах. Документ HTTP называется RFC 2616[3]. С ним сверяются разработчики фреймворков и браузеров, чтобы код работал на разных языках и платформах.

HTTP удобен тем, что это текст. Не нужно парсить байты, чтобы понять, что происходит. Протокол работает и с бинарными данными, но главные его части остаются текстом. В HTTP различают запрос и ответ. Оба состоят из трёх частей: первая строка, заголовки и тело.

Первая (стартовая) строка несёт самую важную информацию. Её формат отличается для запроса и ответа. Для запроса это метод, путь и версия, для ответа — статус, сообщение и версия.

Заголовки — это пары ключей и значений. В коде их описывают словарём. Заголовки несут дополнительные сведения о запросе или ответе. Например, `Content-Type` сообщает, как читать тело. Был ли это XML- или JSON-документ? Программа сверяет заголовок и читает тело должным образом.

После заголовков следует тело. Им может быть что угодно — текст, пары полей и значений, JSON, картинка. Стандарт допускает смешанный тип, `multipart-encoding`. Тело такого запроса состоит из ячеек, в каждой из которых своё содержимое: текст, картинка, снова текст, архив.

Рассмотрим примеры трафика HTTP. Именно в таком виде его передают по сети. Ниже запрос к главной странице Google по слову clojure:

```
GET /search?q=clojure HTTP/1.1
Host: google.com
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

А это POST-запрос с JSON:

```
POST /api/users/ HTTP/1.1
Host: example.com
Content-Type: application/json

{"username": "John", "city": "NY"}
```

[3] tools.ietf.org/html/rfc2616

Обратите внимание на пустую строку выше: она отделяет тело от заголовков. Ответ на этот запрос:

```
HTTP/1.1 200 OK
Date: Tue, 19 Mar 2019 15:57:11 GMT
Server: Nginx
Connection: close
Content-Type: application/json

{
  "code": "CREATED",
  "message": "User has been created successfully."
}
```

Видно, как изящно устроен протокол: данные идут по убыванию важности. Прочитав только первую строку, клиент и сервер готовы принять решение о том, что делать дальше.

Рассмотрим случай, когда метод и путь запроса `GET /about`, но такой страницы не существует. Сервер проверит путь по таблице маршрутов. Если его нет, получим ответ со статусом 404. Не придётся читать тело запроса, что ускорит работу сервера. Клиент получит статус 404 из первой строки. Логика клиента может быть такова, что для негативного статуса он не читает тело.

Чтение и разбор содержимого — это долгая операция. Современные фреймворки не делают этого зря. По заголовку `Content-Type` они определяют, стоит ли читать тело. Если приложение работает только с JSON, для `text/xml` вернётся ошибка. Аналогично поступают с заголовком `Content-Length`, где указана длина тела в байтах. Если значение больше лимита, сервер отклонит запрос до чтения.

Главные части запроса — это *метод* и *путь*. Путь указывает на определённый ресурс на сервере. Иногда он означает файл относительно заданной папки. Например, `/images/map.jpg` вернёт одноимённый файл из `/var/www/static`. Раздача файлов — это частный случай пути, и у него много других сценариев. В пути может быть номер сущности: `/users/9677/profile`. Сервер можно настроить так, что запросы с префиксом `/internal` и `/public` уходят на разные машины.

Метод запроса означает действие, которое мы намерены выполнить над ресурсом. Основные методы — это `GET`, `POST`, `PUT` и `DELETE` — прочитать, создать, обновить и удалить ресурс. Запрос `POST /users/` означает создать пользователя, а `GET /users/` — получить список пользователей.

Главный параметр ответа — это статус — целое положительное число. Статусы группируют по старшей цифре. Значения с 200 до 299

считают положительными. Они означают, что сервер обработал запрос без ошибки. Для краткости интервал обозначают `2xx`.

Значения из группы `3xx` связаны с направлением на другую страницу. В заголовке `Location` указан адрес, куда нужно отправить новый запрос. Современные браузеры и клиенты делают это автоматически. По адресу `http://yandex.ru` получим пустой документ с заголовком `Location: https://yandex.ru`. Разница в схеме протокола: сервер обязывает перейти на безопасное соединение. Мы даже не заметим этого: браузер сам сменит адрес.

Статусы `4xx` означают ошибку на стороне клиента. Чаще других встречается 404 — страница не найдена. Если прислать ошибочные данные, сервер ответит: 400 Bad request. Когда нет прав доступа, получим код 403.

Значения из группы `5xx` говорят о проблеме на стороне сервера. В основном это ошибки в коде: отказ базы данных, нехватка места на диске. Если сервер на техобслуживании, он вернёт код 503. В редких случаях он выключен и не отвечает на запросы.

Принято считать, что ответ со статусом вне группы `2xx` означает ошибку. Многие HTTP-клиенты бросают исключение на ответ с негативным статусом. Это верно только на абстрактном уровне. С точки зрения протокола ответ 404 такой же правильный, как и 200.

Когда действий с ресурсом много, применяют другие, более редкие методы. Например, HEAD — получить краткие сведения о сущности. Сервис Amazon S3 в ответ на HEAD вернёт только статус и заголовки с пустым телом. В них указаны тип файла и его размер, контрольная сумма, дата изменения. HEAD-запрос предпочтительней GET. Метаданные хранят отдельно от файла, поэтому доступ к ним быстрее, чем к диску.


REST

Подход «метод и ресурс» вырос в то, что сегодня называется REST[4]. Сторонники REST выделяют сущности и CRUD-операции над ними (**C**reate, **R**ead, **U**pdate, **D**elete). Считается верным подход, когда сущность задают через путь, например `/users/1`, а операцию — методом. Если это запрос на изменение, данные читают из тела с JSON. Мы не будем задерживаться на REST, потому что это лишь свод рекомендаций, не идеальный и не единственный.

Протокол не заставляет следовать REST и другим правилам. Работайте с HTTP так, как это удобно проекту. Идеальная архитектура не обещает успех, и наоборот: успех не значит, что под капотом всё идеально.

[4] restapitutorial.com

1.1.1 Фреймворк

Фреймворк — это абстракция над HTTP. Разработчик не читает запрос по байтам вручную — задачу берёт на себя чужой код. Взамен нам дают классы, чтобы описать логику приложения. Типичный проект на Python или Java состоит из следующих классов.

`Application` — это главная сущность проекта: она группирует классы рангом ниже. `Router` определяет, на какой обработчик подать входящий запрос — `Request`. Обработчик — это класс `Handler` с методами `.onGet`, `.onPost` и другими. Они вернут экземпляр класса `Response`. Так устроены промышленные фреймворки вроде Django и Rails. Имена и состав классов отличаются, но смысл прежний: приложение, роутер, обработчик, запрос и ответ.

Большие проекты делят на слои. Слой транспорта отвечает за обмен данными, слой логики исполняет код, ничего не зная об источнике данных. С таким подходом логика не зависит от транспорта, и последний можно сменить в любой момент. Например, направить долгий запрос в очередь задач. На практике это работает не всегда: по разным причинам, в том числе из-за спешки, слои перемешиваются.

Проекты на Clojure опираются на фреймворки. Принципы, о которых мы говорили выше, справедливы и для этого языка.

1.2 HTTP в Clojure


James Reeves

Разработчик Джеймс Ривз[5] (James Reeves) известен вкладом в экосистему Clojure. Нет проекта, который бы не использовал его библиотеки. Джеймс ввёл стандарт веб-разработки для Clojure на заре языка. Стандарт опирается на несколько простых идей.

Приложения бывают сколь угодно сложными: они полагаются на сторонние сервисы, машинное обучение, учитывают сотню фактов о клиенте. Но даже самое сложное приложение принимает запрос и возвращает ответ, и поэтому это функция. Скептики заметят, что мысль не нова. В Django обработчик тоже бывает не классом, а функцией. Разница в том, что обработчик — это ещё не приложение. Ему не хватает роутера, middleware и других абстракций. Функция-обработчик в других языках — это локальная возможность.

В Clojure приложение остаётся функцией на всех уровнях. Маршрут — это функция, которая принимает запрос, ищет обработчик и

[5] www.booleanknot.com/

Глава 2

Clojure.spec

В этой главе мы рассмотрим clojure.spec — библиотеку для проверки данных в Clojure. Это особенная библиотека: на ней пишут валидаторы и парсеры, с её помощью генерируют данные для тестов. Spec фундаментальна по своей природе, поэтому уделим ей пристальное внимание.

Название spec происходит от specification (с англ. — «спецификация, описание»). Это набор функций и макросов, чтобы схематично описать данные. Например, из каких ключей состоит словарь и типы его значений. Запись называют спецификацией данных или сокращённо спекой. Далее мы будем использовать короткий термин.

Специальные функции проверяют, подходят ли данные к спеке. Если нет, получим отчёт, в каком месте произошла ошибка и почему.

Spec входит в поставку Clojure начиная с версии 1.9. Полностью модуль называется `clojure.spec.alpha`. Не волнуйтесь о частичке alpha на конце имени: она осталась по историческим причинам.

Spec стала важной вехой в развитии Clojure. Ключевое свойство Spec в том, что она фундаментальна. Валидация данных — это малая часть её возможностей. Spec не только проверяет данные, но и преобразует их. На Spec легко писать парсеры.

Формально Spec — это обычная библиотека. Но её абстракции настолько мощны, что Clojure переиспользует их. С версии 1.10 компилятор Clojure анализирует главные макросы с помощью Spec. Так проекты дополняют друг друга.

Прежде чем браться за техническую часть, разберёмся с теорией. Вспомним, как связаны между собой классы, типы и валидация.

2.1 Типы и классы

Считается, что код на языке со статической типизацией безопаснее, чем с динамической. Компилятор не позволит сложить число и строку ещё до того, как мы запустим программу. Однако тип переменной — это лишь одно из многих ограничений. Редко случается так, что тип задаёт все допустимые значения. Чаще всего вместе с типом учитывают границы, длину, попадание в интервалы и перечисления. Иногда значения верны по отдельности, но не могут стоять в паре друг с другом.

Рассмотрим, как выразить в коде сетевой порт. В операционной системе это число от 0 до $2^{16} - 1$. Целые типы обычно описаны степенями двойки, поэтому найдётся условный `unsigned int`, который охватит именно этот диапазон. У нулевого порта особая семантика, и в прикладных программах его не используют. Вероятность, что в языке предусмотрен тип от 1 до $2^{16} - 1$, крайне мала.

Легче всего увидеть проблему на диапазоне дат. Единичная дата может быть сколь угодно разумной, но диапазон накладывает ограничение: начало строго меньше конца. Бизнес дополняет: разница не больше недели, обе даты в рамках текущего месяца.

В ООП знают об этой проблеме и решают её классами `UnixPort` и `DateRange`. Условный `UnixPort` — это класс с конструктором. Он принимает целое число и выполняет проверку на диапазон. Если число выходит за рамки $1 \ldots 2^{16} - 1$, конструктор бросит исключение. Программист уверен, что создал новый тип. Это неверно — классы и типы не тождественны.

Конструктор — это обычный валидатор. Он неявно сработает, когда мы напишем `new UnixPort(8080)`. Из-за неявности возникает иллюзия, что мы создали тип. На деле это валидация и синтаксический сахар.

В промышленных языках нельзя описать класс так, чтобы выражение `new UnixPort(-42)` привело к ошибке компиляции. Найти её могут только сторонние утилиты и плагины для IDE.

Конструктор нельзя использовать повторно. Представим классы `UnixPort` и `NetPort`. Первый класс проверяет порт на диапазон и бросает исключение. Выгодно пользоваться этим классом, поскольку он совмещён с валидацией. Однако сторонняя библиотека принимает `NetPort`. Возникает проблема конвертации: нужно извлечь «сырой» порт из `UnixPort` и передать в `NetPort`. Это лишний код и путаница с классами.

Признаки удобной валидации — это независимость и компоновка. Независимость означает, что данные не привязаны к валидации. Нет ничего зазорного в том, что порт — это целое число. Пусть библиотека принимает `integer`, а разработчик сам решит, как его проверить. Появится выбор, насколько строгой должна быть проверка.

Компоновка означает, что полезно иметь несколько простых проверок, чтобы составить из них сложные. Пусть заданы проверки «это» и «то» и теперь нужны комбинации «это *и* то», «это *или* то». В идеале компоновка занимает пару строк и считается тривиальной задачей.

Оба тезиса ложатся на функцию. На неё действует одна операция — вызов, что упрощает схему. Функция принимает значение и возвращает истину или ложь. Это ответ на вопрос, было ли значение правильным или нет. Функция — объект высшего порядка, поэтому другие функции порождают из них комбинации.

2.2 Основы spec

С багажом рассуждений мы подходим к Spec. Подключим модуль в текущее пространство:

```
(require '[clojure.spec.alpha :as s])
```

Синоним `s` нужен, чтобы избежать конфликтов имён с `clojure.core`. Модуль Spec несёт макросы `s/and`, `s/or` и другие, у которых ничего общего с обычными `and` и `or`. Считается дурным тоном, если имена одного модуля затеняют другие, поэтому обращаемся к Spec через синоним.

Главная операция в Spec — создать новую *спеку*:

```
(s/def ::string string?)
```

Макрос `s/def` принимает ключ и предикат. Он создал объект спеки из функции `string?` и поместил в глобальный реестр с ключом `::string`.

Важно понимать, что `::string` — это не спека, а псевдоним. Макросы Spec работают не с объектами спеки, а с ключам. Они сами найдут спеку в реестре. Это удобно, потому что ключи глобальны. В любом месте можно сослаться на `::string` без лишних импортов.

Вторым аргументом идёт предикат `string?`. Предикат — это функция, которая возвращает истину или ложь. Функция — это не спека, а строительный материал для неё. Спека оборачивает функцию в особый

объект. Технически на него можно сослаться: функция `s/get-spec` по ключу спеки вернёт её объект. На практике он не нужен, потому что везде указывают ключи.

```
(s/get-spec ::string)
;; #object[clojure.spec.alpha$reify 0x3e9dde1d]
```

Спеки хранятся в глобальном реестре под своими ключами. Макрос `s/def` не проверяет, была ли уже такая спека, перед тем как поместить её в реестр. Если была, мы потеряем старую версию.

Spec не работает с ключами без пространства, например `:name` или `:email`. Это повышает риск конфликта ключей. Чтобы назначить ключу текущее пространство, поставьте два двоеточия: `::name`, `::email`.

Самое простое, что можно сделать со спекой, — проверить, подходит ли ей значение. Функция `s/valid?` принимает ключ спеки, значение и возвращает `true` или `false`.

```
(s/valid? ::string 1)      ;; false
(s/valid? ::string "test") ;; true
```

Пустая строка пройдёт валидацию, но чаще всего в этом нет смысла. Пустые имя или заголовок означают ошибку. Объявим спеку, которая дополнительно проверит, что строка не пустая. Наивный способ это сделать — усложнить предикат:

```
(s/def ::ne-string
  (fn [val]
    (and (string? val)
         (not (empty? val)))))
```

Быстрая проверка:

```
(s/valid? ::ne-string "test") ;; true
(s/valid? ::ne-string "")     ;; false
```

Ключ `::ne-string` — это сокращение от «**n**on-**e**mpty string». Спека встречается часто, поэтому логично сэкономить на её имени.

Более изящный способ задать эту спеку — объединить предикаты через `every-pred`. Функция принимает предикаты и возвращает суперпредикат. Он вернёт истину только если истинны все предикаты.

```
(s/def ::ne-string
  (every-pred string? not-empty))
```

Мы собираем новую сущность из базовых, что короче и следует функциональному стилю. Но ещё лучше комбинировать не предикаты, а спеки. Макрос `s/and` объединяет несколько предикатов и спек в новую спеку:

```
(s/def ::ne-string
  (s/and string? not-empty))
```

Так в Clojure строят сложные спеки: объявляют примитивы и наращивают их комбинации.

2.3 Исключения

Во время проверки Spec не перехватывает исключения — о них заботится программист. Рассмотрим спеку для проверки URL. Проще всего это сделать регулярным выражением:

```
(s/def ::url
  (partial re-matches #"(?i)^http(s?)://.*"))

(s/valid? ::url "test")            ;; false
(s/valid? ::url "http://test.com") ;; true
```

Что-то отличное от строки вызовет ошибку:

```
(s/valid? ::url nil)
;; Execution error (NullPointerException)
;; at java.util.regex.Matcher...
```

Примечание: класс `NullPointerException` — частый гость в мире Java. Для краткости его называют `NPE`.

Причина в том, что `nil` попал в функцию `re-matches`. Функция трактует аргумент как строку, что приводит к `NPE`. Перепишите спеку так, чтобы она не бросала исключения. В примере с `::url` сначала убедимся, что это строка, и только потом проверим регулярным выражением.

```
(s/def ::url
  (s/and ::ne-string
         (partial re-matches #"(?i)^http(s?)://.*")))

(s/valid? ::url nil) ;; false
```

Макрос `s/and` устроен так, что на первой неудаче цепь оборвётся. Всё, что после `::ne-string`, не сработает, и исключения не будет.

По аналогии проверим возраст пользователя. Это предикаты на число и диапазон.

```
(s/def ::age
  (s/and int? #(<= 0 % 150)))

(s/valid? ::age nil) ;; false
(s/valid? ::age -1)  ;; false
(s/valid? ::age 42)  ;; true
```

2.4 Спеки-коллекции

Выше мы проверяли примитивные типы, или *скаляры*. Это удобно для примеров, но редко встречается на практике. В основном проверяют не скаляры, а коллекции. Spec предлагает макросы, чтобы задать спеки-коллекции из примитивов.

Макрос `s/coll-of` принимает предикат или ключ и возвращает спеку-коллекцию. Она проверяет, что каждый элемент проходит валидацию. Вот так мы определим список URL:

```
(s/def ::url-list (s/coll-of ::url))
```

Быстрая проверка:

```
(s/valid? ::url-list ["http://test.com" "http://ya.ru"])
;; true

(s/valid? ::url-list ["http://test.com" "dunno.com"])
;; false
```

Макрос `s/map-of` описывает словарь. Вспомним поле `:params` из главы про веб-разработку (с. 27). Его ключи — кейворды, а значения — строки. На языке спеки это выглядит так:

```
(s/def ::params
  (s/map-of keyword? string?))

(s/valid? ::params {:foo "test"})  ;; true
(s/valid? ::params {"foo" "test"}) ;; false
```

Глава 3

Исключения

В этой главе мы рассмотрим исключения в Clojure: как они устроены и чем отличаются от аналогов Java. Когда лучше бросать, а когда перехватывать исключения. Что и как писать в лог, чтобы расследовать инцидент было легко.

Возможно, читателю покажется странным, что исключениям отдана целая глава. Тему считают простой: исключения можно кинуть, поймать и записать в лог. В теории этого хватит, чтобы работать в проекте.

Исключения просты с технической стороны, но несут обширную *семантику*. Когда именно кидать исключения, а когда перехватывать? Какую полезную информацию они несут? Куда записывать исключения? Можно ли ловить их предикатами? На практике мы тонем во множестве частных случаев.

Новички следуют только положительному пути, поэтому в их коде тяжело расследовать ошибки. Почему сервер ответил с кодом 500? Возможны сотни причин, по которым запрос не удался. Но запись в логе слишком скудна, чтобы понять, что произошло.

Хороший программист внимателен к ошибкам. С опытом становится ясно: экономия на исключениях не даёт выигрыша. Да, мы быстрее закроем задачу, и кода получится меньше. Однако позже возникнут задачи на устранение ошибок и их детализацию.

Исключения в коде столь же равноправны, как и нормальное поведение. Избегайте мысли, что это недоразумение, которое не случится с вами. Если в проекте много задач на непойманные ошибки, это значит, что пора изучить тему.

3.1 Основы исключений

Прежде чем углубляться в детали, вспомним, что такое исключения и как они себя ведут.

Исключение — это объект, чаще всего экземпляр класса `Exception`. От других классов он отличается тем, что его можно *бросить*. В разных языках для этого служат операторы `throw`, `raise` и другие.

Брошенный объект прерывает исполнение и всплывает по стеку вызовов. Возможны два исхода: либо его поймали оператором `catch` на одном из уровней, либо перехват не состоялся.

В первом случае мы получим *объект* исключения. К нему обращаются как обычно: читают поля, вызывают методы, передают в функции. Дальнейшее поведение зависит от логики программы. Иногда исключение пишут в лог и завершают программу, в других случаях продолжают работу.

Когда исключение не поймали, программа завершится с кодом, отличным от нуля. Если не предусмотрено иное, перед выходом программа запишет исключение в `stderr` (канал ошибок). Мы увидим его класс, текст и то, что называют «стектрейс». Это цепочка вызовов, которые прошло исключение.

Отдельные платформы позволяют задать реакцию на непойманное исключение. Например, записать его в файл или завершить программу особым способом.

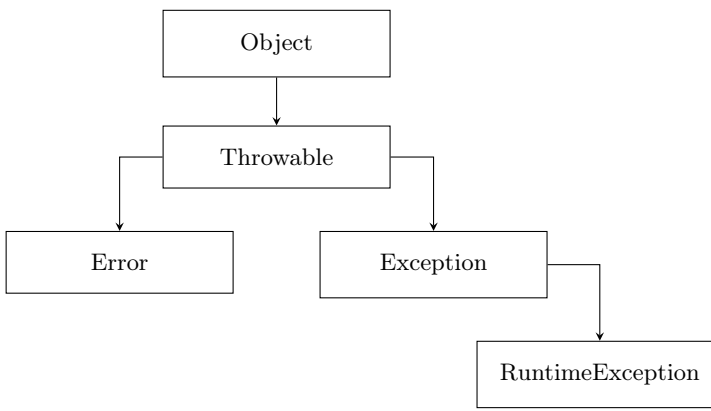

Рис. 3.1. Базовые классы исключений

Clojure — это гостевой язык (англ. hosted language). Он опирается на возможности, которые предлагает домашняя платформа, *хост*. Исключения — одна из областей, в которую Clojure не вмешивается. По умолчанию Clojure использует формы `try` и `catch`, аналогичные Java.

Рассмотрим исключения в Java (рис. 3.1). Платформа содержит базовый класс `Throwable` — предок всех исключений. Другие классы наследуют его и расширяют семантику. Наследники первого уровня — это классы `Error` и `Exception`. Класс `RuntimeException` унаследован от `Exception` и так далее.

Пакеты Java несут дополнительные исключения, унаследованные от описанных выше. Например, `java.io.IOException` для ошибок ввода-вывода, `java.net.ConnectException` для сетевых проблем — другие. Бросать `Throwable` считается дурным тоном. Этот класс несёт слишком мало информации о том, что случилось.

В дереве исключений каждый класс дополняет семантику предка. Рассмотрим исключение `FileNotFoundException`. Оно возникает, когда файла не оказалось на диске. Родословная класса выглядит так:

```
java.lang.Object
└─ java.lang.Throwable
   └─ java.lang.Exception
      └─ java.io.IOException
         └─ java.io.FileNotFoundException
```

Схему читают «объект → выбрасываемое → исключение → ошибка ввода-вывода → файл не найден». По имени `FileNotFoundException` легко догадаться, с чем связана проблема. Если же разработчик бросил `Throwable`, это усложнит поиск причины.

Различают *checked-* и *unchecked*-исключения, проверяемые и нет. Разница между ними в семантике. Разработчик должен предвидеть checked-исключения и обработать их в коде. При чтении файла справедливо ожидать, что его не окажется на диске, поэтому класс `FileNotFoundException` относится к категории checked.

Предсказать нехватку памяти трудно, поэтому `OutOfMemoryError` непроверяемое исключение (unchecked). Когда ресурсы ограничены, любое действие может исчерпать память. Перехватывать это исключение нет смысла, поскольку при нехватке памяти система нестабильна.

Классы, унаследованные от `Error` и `RuntimeException`, — это непроверяемые исключения (unchecked). Унаследованные от `Exception` — проверяемые (checked).

Чтобы бросить исключение, его экземпляр передают в оператор `throw`. Оператор `catch` перехватывает исключения. В Java и других языках он устроен на иерархии классов. Если искомый тип `IOException`, мы поймаем все исключения, унаследованные от этого класса.

Чем выше класс в дереве наследования, тем больше исключений охватит `catch`. В Java считается плохим тоном ловить ошибки классами `Throwable` или `Exception`. Современные IDE выдают предупреждение too wide catch expression, слишком широкий охват. Класс `Exception` заменяют на несколько более точных исключений. Например, отдельно ошибки ввода-вывода, сети и другие.

Одного класса недостаточно, чтобы понять причину исключения. У `FileNotFoundException` нет поля `file`, чтобы отследить, какой именно файл не удалось найти. Большинство исключений принимают строку с сообщением об ошибке. Сообщение должно быть понятно человеку. Из строки File C:/work/test.txt not found станет ясно, к какому файлу мы обращались.

Иногда текста не хватает, чтобы объяснить причину ошибки. Предположим, данные не прошли валидацию и нам хотелось бы исследовать их позже. Если записать данные в сообщение, текст получится слишком большим. Это небезопасно: в данных могут быть личные данные или ключи доступа. Такое сообщение нельзя писать в лог или показывать пользователю. Даже путь к файлу может выдать важную информацию.

Если нужно сохранить данные для расследования, создают новый класс исключения. У него отдельное поле для данных, из-за которых произошла ошибка. Поле заполняют в конструкторе исключения. Сообщение формируют так, чтобы оно не выдало приватную информацию.

3.2 Цепочки и контекст

Исключения строятся в цепочку. Каждый экземпляр принимает необязательный аргумент `cause` (англ. «причина»). Он хранит либо `Null`, либо ссылку на другое исключение.

Цепочки образуются, когда код перехватил исключение, но не знает, как с ним поступить. Это нормально, потому что на низком уровне код не видит полной картины. Предположим, метод пишет данные в файл. У него нет полномочий решить, что делать, если файла нет, поэтому метод бросит исключение. Его перехватит метод, который тоже не принимает решений. Остаётся бросить новое исключение со ссылкой на первое. Это и есть цепочка.

Глава 4

Изменяемость

В классических языках данные меняются, а стандартная библиотека предлагает ограничения: локи, атомарные действия, постоянные коллекции. В Clojure, наоборот, данные не меняются, а мутабельные типы задвинуты на второй план. Это сделано специально, потому что неизменяемость — центральная идея языка.

Руководства по Clojure учат постоянными коллекциям. Это правильный подход, но когда появляется состояние, новички испытывают трудности. В этой главе мы займём другую позицию: рассмотрим, как управлять состоянием в программах.

4.1 Общие проблемы

На Clojure трудно писать в императивном стиле, когда акцент сделан на изменении данных. Скажем, чтобы получить список удвоенных чисел, выполняют шаги:

- создать пустой список — будущий результат;
- пройти по элементам исходного списка;
- на каждом шаге вычислить новый элемент;
- добавить его к результату.

Базовые типы Clojure не меняются, и к ним нельзя применить алгоритм выше. Те, кто пришёл из императивных языков, поначалу не могут писать код с постоянными коллекциями. Привычка менять данные так сильно укрепилась в них, что иммутабельность кажется физическим ограничением.

Создатель Clojure полагает, что изменяемость — основная проблема в разработке ПО. Когда мы пишем код, то видим его начальное состояние, в котором он будет первый такт машинного времени. Затем программа инициирует классы, заполнит поля, и объекты изменятся.

Некоторые ошибки трудно расследовать, потому что код и состояние расходятся. Чтобы исправить ошибку, её повторяют в локальном окружении. Однако привести код в конкретное состояние не так просто. Неизменяемые данные отсекают целый пласт ошибок, от которых страдают императивные языки.

Рассмотрим примеры на Python. В модуле заданы параметры запроса по умолчанию. Функция `api_call` принимает дополнительные параметры, объединяет со стандартными и передаёт в HTTP-клиент:

```
1  DEFAULT_PARAMS = {
2      "allow_redirects": True,
3      "timeout": 5,
4      "headers": {"Content-Type": "application/json"},
5      "auth": ("username", "password"),
6  }
7
8  def api_call(**params):
9      api_params = DEFAULT_PARAMS
10     api_params.update(params)
11     resp = requests.post("https://api.host.com", **api_params)
12     return resp.json()
```

В теле `api_call` грубая ошибка: переменная `api_params` получает не копию глобальных параметров, а *ссылку* на них (строка 9). Изменяя `api_params`, мы на самом деле меняем `DEFAULT_PARAMS` (строка 10). На каждый вызов глобальные параметры меняются, что ведёт к странному поведению программы. Код и состояние «разъехались».

На собеседованиях часто задают следующий вопрос. Представьте функцию с сигнатурой ниже. Объясните, что в ней не так, и приведите пример ошибки.

```
def foo(bar=[]):
```

Ответ: параметры функции по умолчанию создаются однажды. В данном случае `bar` равен пустому списку. В Python список изменяется. Если в `bar` ничего не передали, получим исходный список. Добавим в него элемент, и в следующий раз `bar` будет уже не пустой:

```
def foo(bar=[]):
    bar.append(1)
    return bar
```

Вызов `foo` вернёт списки `[1]`, `[1, 1]` и так далее. Ещё хуже: если результат `foo` сохранить в переменную и позже добавить к нему элемент, на самом деле изменится злосчастный `bar`.

Современные IDE проверяют код на неявные ошибки. Про список в сигнатуре знают все анализаторы и линтеры. Но мы не можем целиком положиться на утилиты: если данные меняются постоянно, трудно понять, где ошибка, а где умысел.

Начинающих кложуристов выдаёт код:

```
(let [result (atom [])
      data [1 2 3 4 5]]
  (doseq [item data]
    (let [new-item (* 2 item)]
      (swap! result conj new-item)))
  @result)
```

Это привычка из императивного прошлого. Атом-аккумулятор лишний, достаточно `map` или `for`:

```
(map (partial * 2)
     [1 2 3 4 5])
```

```
(for [n [1 2 3 4 5]]
  (* n 2))
```

Оба выражения короче и понятней. Не нужно создавать вектор и добавлять в него элементы — это делают функции. Если обход коллекции завязан на атоме, скорее всего это слабое решение.

Авторы Clojure сделали всё, чтобы выделить состояние на общем фоне. К состоянию прибегают только в крайних случаях. Если вы написали код на атомах без уважительной причины, вам сделают замечание или не примут работу.

4.1.1 В защиту состояния

Мы говорили, что состояние несёт потенциальные ошибки. Это слишком линейное заявление: без состояния работают только небольшие программы. Например, скрипты, которые запускают раз в день. Писать промышленный код без состояния невозможно.

Постоянные данные избавляют нас от ошибок с перезаписью полей. Это значимый выигрыш, но кроме данных приложение полагается на *ресурсы*. Для них действует правило: дешевле работать с открытым ресурсом, чем постоянно открывать и закрывать его. Состояние повышает скорость программы.


CGI

Много лет назад веб-серверы работали по протоколу CGI, Common Gateway Interface[1]. На каждый запрос сервер запускал скрипт или бинарный файл. Скрипт получал данные запроса из переменных среды. Программа писала ответ в стандартный поток. Сервер перехватывал его и выводил пользователю.

Схема была простой и удобной. Приложение могло быть скриптом на Perl или программой на C++. У сервера не было состояния. В любой момент разработчик обновлял файл, и изменения вступали в силу немедленно.

За преимущества платили низкой скоростью. Каждый запрос к серверу порождал новый процесс. Даже если программа написана на Си, запуск процесса занимает время. Индустрия пришла к тому, что приложение должно работать постоянно, а не по запросу.

Приложение на FastCGI устроено как самостоятельный сервер. Его производительность на два порядка выше, чем у CGI. В нём появилось состояние — открытый порт и цикл ввода-вывода. Цикл читает запрос и делегирует отдельному потоку. Это усложнило разработку, что привело к новым парадигмам и фреймворкам.

Похоже устроены соединения с базой данных. Представим, что на каждый запрос мы открываем соединение, работаем с ним и закрываем. В машинном мире открыть TCP-соединение — это долгая операция. Так появились пулы соединений.

Пул — это объект, который держит несколько открытых соединений. Пул знает, какое из них занято или свободно. Чтобы работать с базой, мы занимаем одно из свободных соединений, работаем с ним и возвращаем. Для потребителя пул — это примитивный объект, который выдаёт и забирает соединения.

Логика пула довольно сложна. Если соединений не хватает, он увеличивает свою ёмкость, а при избытке сокращает. Для каждого соединения пул считает время работы и прочие метрики. Он же решает, когда закрыть соединение и заменить его новым. Пул работает в отдельном потоке, чтобы не блокировать основную программу.

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

Столь сложное устройство компенсирует скорость доступа. Каждый запрос протекает по заранее открытому соединению, что намного быстрее, чем открывать его каждый раз.

Сама архитектура машин поощряет изменять данные. В школе нам объясняют память компьютера как массив ячеек. Запись в ячейку по адресу дёшева. И в C++, и в Python одинаково легко обновить элемент массива:

```
items[i] = 5;
```

Постоянные структуры хуже ложатся на эту модель памяти. Поэтому они сложнее: неизменяемый список — это не цепочка, а дерево узлов с указателем. Постоянные коллекции умны и копируют данные не полностью, а частично. Всё же на больших объёмах выгоднее работать с изменяемыми структурами.

Мы не призываем всюду внедрять состояние. Древовидность и замедление — это цена, которую платят за меньший риск ошибки. Инженер должен знать, на что идёт, когда добавляет состояние или избавляется от него. По ходу главы мы изучим императивные возможности Clojure — как ими пользоваться и когда это действительно нужно.

4.2 Атомы

Clojure предлагает несколько способов менять данные. Самый простой из них — атом — объект, который прячет в себе другой объект. Атом получают одноимённой функцией с начальным значением:

```
(def store (atom 42))
```

Если напечатать атом, увидим следующее:

```
#<Atom@10ed2e87: 42>
```

Чтобы извлечь значение, применяют оператор `@`. Запись `@store` — это укороченный вариант `(deref store)`. Функция `deref` принимает атом и возвращает содержимое. Семантически это то же самое, что получить значение по указателю. В русской литературе операцию называют «разыменование». В разговорном языке про оператор `@` говорят «дереф», «дерефнуть».

```
@store ;; 42
```

Глава 5

Конфигурация

В этой главе мы рассмотрим, как сделать проект на Clojure удобным в настройке. Разберём основы конфигурации: форматы файлов, переменные среды, библиотеки, их достоинства и недостатки.

5.1 Постановка проблемы

В материалах по Clojure встречаются примеры:

```
(def server
  (jetty/run-jetty app {:port 8080}))

(def db {:dbtype   "postgres"
         :dbname   "test"
         :user     "ivan"
         :password "test"})
```

Это сервер на порту 8080 и параметры подключения к базе. Польза примеров в том, что их можно скопировать в REPL и оценить результат: открыть страницу в браузере или прочитать таблицу.

На практике код пишут так, чтобы в нём не было конкретных чисел и строк. С точки зрения проекта плохо, что серверу явно задали порт. Это подойдёт для документации и примеров, но не для боевого запуска.

Порт 8080 и другие комбинации нулей и восьмёрок популярны у программистов. Велики шансы, что порт занят другим сервером. Это случается, когда запускают не отдельный сервис, а их связку на время разработки или тестов.

Код, написанный программистом, проходит несколько стадий. В разных фирмах набор отличается, но в целом это разработка, тестирование, предварительный и боевой запуск.

На каждой стадии приложение запускают бок о бок с другими проектами. Предположение, что порт 8080 свободен в любой момент, — это утопия. На жаргоне разработчиков ситуацию называют «хардкод» (англ. hardcode) или «прибито гвоздями». Когда в коде «прибитые» значения, это вносит проблемы в его цикл. Вы не сможете одновременно работать над проектом и тестировать его.

Приложение не должно знать порт сервера — информация об этом приходит извне. В простом случае это файл настроек. Программа читает из него порт и запускает сервер именно так, как это нужно на конкретной машине.

В более сложных сценариях файл составляет не человек, а специальная программа — менеджер конфигураций. Менеджер хранит информацию о топологии сети, адреса машин, параметры доступа к базам. По запросу он выдаёт файл для определённой машины или сегмента сети.

Процесс, когда приложению сообщают параметры, а оно принимает их, называется конфигурацией. Это интересный и важный этап в разработке программ. Когда он устроен удачно, проект легко проходит по всем стадиям производства.

5.2 Семантика

Цель конфигурации в том, чтобы управлять программой без изменений в коде. К ней приходят с ростом кодовой базы и инфраструктуры. Если у вас мелкий скрипт на Python, нет ничего зазорного в том, чтобы открыть его в блокноте и поменять константу. На предприятиях такие скрипты работают годами.

Чем сложнее инфраструктура фирмы, тем больше в ней ограничений. Современный подход сводит на нет спонтанные изменения в проекте. Нельзя сделать `git push` напрямую в мастер; запрещён `merge`, пока вашу работу не одобрят двое коллег; приложение не попадёт на сервер, пока не пройдут тесты.

Это приводит к тому, что малейшее изменение в коде займёт час, чтобы попасть в бой. Правка в конфигурации дешевле, чем выпуск новой версии продукта. Из этого следует правило: если можно вынести что-то в конфигурацию, сделайте это сейчас.

В крупных фирмах практикуют то, что называют feature flag. Это логическое поле, которое включает целый пласт логики в приложении. Например, новый интерфейс, систему обработки заявок, улучшенный чат. Обновления тестируют внутри фирмы, но всегда остаётся риск, что в бою *что-то пойдёт не так*. В этом случае флаг меняют на ложь и перезапускают сервер. Это не только сэкономит время, но и сохранит репутацию фирмы.

5.3 Цикл конфигурации

При запуске приложение ищет конфигурацию. Чем лучше устроено приложение, тем больше его частей опирается на параметры. Обработка конфигурации — это не монолитная задача, а набор шагов. Перечислим наиболее важные из них.

На первом этапе программа **читает конфигурацию**. Чаще всего это файл или переменные среды. Данные в файлах хранят в форматах JSON, YAML и других. Приложение содержит код, чтобы разобрать формат и получить данные. Мы рассмотрим плюсы и минусы известных форматов ниже.

Переменные среды — это часть операционной системы. Представьте их как глобальный словарь в памяти. Каждое приложение наследует его при запуске. Языки и фреймворки предлагают функции, чтобы считать переменные в строки и словари.

Файлы и переменные среды дополняют друг друга. Например, приложение читает данные из файла, но путь к нему ищет в переменных среды. Иногда в файле опускают критические данные: пароли, API-ключи. Так поступают, чтобы их не увидели другие программы, в том числе шпионские. Приложение читает параметры из файла, а секретные данные — из переменных.

Продвинутые конфигурации используют теги. В файле перед значением ставят тег: `:password #env DB_PASSWORD`. Это значит, что в поле `password` не строка `DB_PASSWORD`, а значение одноимённой переменной.

Первый этап завершается тем, что мы получили данные. Неважно, был ли это файл, переменные среды или что-то другое. Приложение переходит ко второму этапу — **выводу типов**.

JSON или YAML выделяют базовые типы: строки, числа, булево и `null`. Легко заметить, что среди них нет даты. С помощью дат задают промоакции или события, связанные с календарём. В файлах даты указывают либо строкой в формате ISO, либо числом секунд


Unix time

с 1 января 1970 года (эпоха UNIX[1]). Специальный код пробегает про данным и приводит даты к типу, принятому в языке.

Вывод типов применяют и для коллекций. Иногда словарей и массивов не хватает для комфортной работы. Типы события хранят в виде множества, потому что оно отсекает дубли и предлагает быструю проверку на вхождение. Скаляры тоже оборачивают в классы, например `UUID` для идентификаторов.

Переменные среды не настолько гибки, как современные форматы. Если JSON выделяет скаляры и коллекции, то переменные несут только текст. Вывод типов для них не просто желателен, а необходим. Нельзя передать порт в виде строки туда, где ожидают число.

После вывода типов приступают к **валидации данных**. В главе про Spec мы выяснили, что тип не обещает верное значение (с. 50). Проверка нужна, чтобы в конфигурации нельзя было указать порт 0, -1 или 80.

Из той же главы мы помним, что иногда значения верны по отдельности, но не могут быть в паре. Пусть в конфигурации задан период акции. Это массив из двух дат, начало и завершение. Легко перепутать даты местами, и проверка любой даты на интервал вернёт ложь.

После валидации переходят к последней стадии. Приложение решает, где **хранить конфигурацию**. Это может быть глобальная переменная или компонент системы. Другие части программы читают параметры уже оттуда, а не из файла.

5.4 Ошибки конфигурации

На каждом этапе может возникнуть ошибка: не найден файл, нарушения в синтаксисе, неверное поле. В этом случае программа выводит сообщение и завершается. Текст должен чётко ответить на вопрос, что случилось. Часто программисты держат в голове только положительный путь и забывают об ошибках. При запуске их программ виден стек-трейс, который трудно понять.

Если ошибка случилась на этапе проверки, объясните, какое поле тому виной. В главе про Spec мы рассмотрели, как улучшить отчёт спеки (с. 68). Это требует усилий, но окупается со временем.

В IT-индустрии одни сотрудники пишут код, а другие управляют им. Ваш коллега-DevOps не знает Clojure и не поймёт сырой `explain`.

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

Рано или поздно он попросит доработать сообщения конфигурации. Сделайте это заранее из уважения к коллегам.

Если с конфигурацией что-то не так, программа не должна работать в надежде, что всё обойдется. Бывает, один из параметров задан неверно, но программа к нему не обращается. Избегайте этого: ошибка появится в неподходящий момент.

Когда один из шагов конфигурации не сработал, программа завершается с кодом, отличным от нуля. Сообщение пишут в канал `stderr`, чтобы подчеркнуть внештатную ситуацию. Продвинутые терминалы печатают текст из `stderr` красным цветом.

5.5 Загрузчик конфигурации

Чтобы закрепить теорию, напишем систему конфигурации. Это отдельный модуль примерно на сто строк. Прежде чем садиться за редактор, обдумаем основные положения.

Будем хранить конфигурацию в JSON-файле. Считаем, что фирма недавно перешла на Clojure и у DevOps уже написаны скрипты на Python для управления настройками. Формат EDN усложнит работу коллегам.

Путь к файлу задают в переменной среды `CONFIG_PATH`. От файла мы ожидаем порт сервера, параметры базы данных и диапазон дат для промоакции. Даты должны стать объектами `java.util.Date`. Дата начала строго меньше конца.

Готовый словарь запишем в глобальную переменную `CONFIG`. Если на одном из шагов случилась ошибка, покажем сообщение и завершим программу.

Начнём со вспомогательной функции `exit`. Она принимает код завершения, текст и параметры форматирования. Если код равен нулю, пишем сообщение в `stdout`, иначе — в `stderr`.

```
(defn exit
  [code template & args]
  (let [out (if (zero? code) *out* *err*)]
    (binding [*out* out]
      (println (apply format template args))))
  (System/exit code))
```

Переходим к загрузчику. Это набор шагов, каждый из которых принимает результат предыдущего. Логику каждого легко понять

Глава 6

Системы

В этой главе мы поговорим о системах. Так называют набор компонентов со связями между ними. Рассмотрим, как большие проекты складываются из малых частей; как победить сложность и заставить части работать как одно целое.

Понятие системы связано с конфигурацией, которую мы только что обсудили. Отличие в том, что конфигурация отвечает на вопрос, как *получить* параметры, а система знает, как ими *распорядиться*.

Системы появились, когда возник спрос на долгоиграющие приложения. Для скриптов и утилит вопрос не стоял остро: их время работы коротко, и состояние живет недолго. При завершении ресурсы освобождаются, поэтому нет смысла в контроле за ними.

С серверными приложениями всё по-другому: они работают постоянно и поэтому устроены иначе, чем скрипты. Приложение состоит из компонентов, которые работают в фоне. Каждый компонент выполняет узкую задачу. При запуске приложение включает компоненты в правильном порядке и строит между ними связи.

6.1 Подробнее о системе

Компонент — это объект, который несёт состояние. На него действуют операции «включить» и «выключить». Как правило, включить компонент означает открыть ресурс, а выключить — закрыть его.

Типичные компоненты приложения — это сервер, база данных, кэш. Чтобы не открывать соединение на каждый запрос к базе, понадобится пул соединений. Не хотелось бы создавать его вручную и передавать

в функции JDBC. Должен быть компонент, который при включении открывает пул и хранит его. Потребителям компонент предлагает методы для работы с базой. Внутри они используют открытый пул.

На первый взгляд, схема напоминает ООП и инкапсуляцию. Не торопитесь с выводами: компоненты в Clojure работают иначе. Ниже мы рассмотрим разницу между объектами и компонентами.

6.1.1 Зависимости

Главная точка системы — это зависимости компонентов. Сервер, база и кэш не зависят друг от друга. Это базовые компоненты системы, на которые опираются другие, уровнем выше. Предположим, фоновый поток читает базу и отправляет письма. Будет неправильно, если компонент откроет новые подключения к базе и почте. Вместо этого он принимает включённые компоненты и работает с ними как с чёрным ящиком.

Система запускает и останавливает компоненты в верном порядке. Если компонент A зависит от B и C, то к моменту запуска A последние два должны быть включены. При завершении компоненты B и C нельзя выключить до тех пор, пока работает A, потому что это нарушит его работу. Система строит граф зависимостей между компонентами. Граф обходят так, чтобы удовлетворить всех участников.

В систему должно быть легко добавить новый компонент. В идеальном случае система — это комбинация словарей и списков. Код загрузки пробегает по ним и включает компоненты. Расширить систему означает добавить новый узел в дерево.

Когда система знает о зависимостях, можно включить её подмножество. Представим, нужно отладить обработчик почты, который зависит от базы и SMTP-сервера. Веб-сервер и кэш в данном случае не нужны, и запуск всей системы избыточен. Продвинутые системы предлагают функцию с семантикой «запусти этот компонент и его зависимости».

6.1.2 Преимущества

На первый взгляд кажется, что система — лишнее усложнение. Это новая библиотека, соглашения в команде и рефакторинг. Однако первичные неудобства окупаются со временем.

Система приводит проект в порядок. С ростом кодовой базы становится важно, чтобы части проекта были в одном стиле. Если этому

следовать, служебные компоненты уйдут в библиотеки, а в проекте останется только логика. Проще начать проект, когда под рукой база внутренних компонентов, испытанных в бою.

Системы полезны на всех стадиях производства, особенно тестировании. В тестах запускают систему, где некоторые компоненты работают по-другому. Например, отправитель СМС пишет сообщения в файл или атом. Компонент авторизации читает код подтверждения из этих источников. Подход не гарантирует полной надёжности, но выполнит тесты изолировано, без обращения к сторонним сервисам. Проблему изоляции мы рассмотрим в главе про тесты (с. 309).

6.2 Подготовка к обзору

В главе об изменяемых данных мы упоминали системы (с. 156). Способ работает на `alter-var-root` и глобальных переменных. Идея в том, чтобы вынести компонент в модуль и снабдить функциями `start!` и `stop!`, которые переключают состояние модуля. Запуск системы сводится к их вызову в верном порядке.

Это любительское решение, потому что система не знает о зависимостях между компонентами. Она хрупкая, работает в ручном режиме, и каждое изменение требует проверки.

Clojure предлагает несколько библиотек для систем. Мы рассмотрим Mount, Component и Integrant. Библиотеки различаются подходом: они по-разному описывают компоненты и зависимости. Так мы рассмотрим проблему с разных сторон.

Библиотеки нарочно следуют в таком порядке. Mount устроен проще, поэтому начнём с него в качестве знакомства с темой. Component стал промышленным стандартом. Уделим ему больше внимания и поэтому ставим в середину. Integrant замыкает обзор: его рассматривают как альтернативу Component, с которым читатель должен быть знаком.

Наша система похожа на то, с чем вы столкнётесь на практике. Она состоит из веб-сервера, базы данных и воркера — фоновой задачи, которая обновляет записи в базе. Мы добавили его специально, чтобы научиться работать с зависимостями. Чтобы лучше понять систему, нарисуем её топологию (рис. 6.1).

Стрелки означают отношения между компонентами. Выражение A → B означает «A зависит от B». В нашей схеме все компоненты зависят от конфигурации. Дополнительно фоновый обработчик нуждается в базе данных. Над этой системой мы будем работать до конца главы.

Глава 7

Тесты

В последней главе мы поговорим о тестировании кода. Читатель узнает, что такое тесты и каких видов они бывают. Обойдёмся без лишней сложности: опустим термины вроде TDD и BDT. Покажем, что в Clojure легко писать и поддерживать тесты.

7.1 Основные понятия

На тему тестов написано много книг и статей, придуманы методологии. Их сторонники отстаивают позиции в долгих спорах. Начинающих сбивает с толку фрагментация терминов и мнений. Чтобы погрузиться в тему, расскажем о тестах простыми словами.

Тест — это код, который проверяет другой код. Напишем функцию для перевода температуры из Цельсия в Фаренгейта:

```
(defn ->fahr [cel]
  (+ (* cel 1.8) 32))
```

Мы вызвали её несколько раз и убедились, что результаты такие же, как в справочнике. Зафиксируем расчёты в функции проверки. Она сравнивает вызов `->fahr` с каноничными значениями. Их посчитали заранее и расценивают как эталон.

```
(defn test-fahr []
  (assert (=  68 (int (->fahr 20))))
  (assert (= 212 (int (->fahr 100)))))
```

Предметный указатель

C

CIDER *233, 285, 286, 293, 301*

clojure.core

— *assert* *159*

— *out* *163*

— *print-length* *158, 163*

— *print-level* *163*

— *print-meta* *159, 251, 260*

— *warn-on-reflection* *159*

— alter-var-root *151, 152, 154, 161, 183, 217, 247*

— assert *159*

— assoc *36, 196*

— assoc! *147*

— assoc-in *196*

— binding *161, 168, 298, 327*

— case *57*

— comp *28*

— compare *90, 182*

— complement *81*

— conj! *148*

— constantly *168*

— contains? *59*

— defonce *248*

— defrecord *235, 258*

— deref *135*

— derive *74, 268*

— dissoc *36*

— dissoc! *147*

— doall *82*

— every-pred *51*

— filter *81*

— find *142*

— format *104*

— future *140, 145, 168, 169*

— get-in *36, 139*

— identity *64*

— keep *70*

— let *166*

— loop *109, 128, 149*

— memoize *142*

— ns-name *305*

— persistent! *147*

— re-matches *51, 68*

— realized? *225*

— reduce *149, 196*

— reset! *136*

— resolve *230*

— select-keys *193*

— set! *158, 251*

— set-validator! *141*

— slurp *40*

— some-fn *82*

— string? *49*

— swap! *136*

— Throwable→map *117*

— time *93*

— transient *146*

— update-in *139*

— var-get *166*