Clojure

на производстве

(второе издание)

Иван Гришаев

Иван Гришаев

Clojure
на производстве

Второе издание

Версия 1774c7e

2021

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.

Оглавление

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.

Во втором издании исправлены опечатки, ошибки и смысловые неточности. В некоторых местах текст сокращён, в других, наобо-

рот, стал подробней. По просьбе читателей добавлены примеры к сложным разделам.

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. Досбол Жантолин внёс важные замечания к последней главе. Молодцы все, кто указал на ошибки в комментариях в блоге.

Алексей Шипилов адаптировал книгу под мобильные устройства и выполнил много рутинных задач по вёрстке.

Вместе с Евгением Бартовым мы перевели книгу на английский язык. Во время перевода Евгений нашёл неточности в русской версии, которые мы тоже исправили.

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. Ваши замечания попадут и в английскую версию книги.

Код

Исходный код книги в виде файлов LaTeX находится на GitHub в репозитории `igrishaev/clj-book`[1]. Если вы нашли опечатку,

Clojure book

[1] github.com/igrishaev/clj-book

откройте pull request или issue с описанием проблемы.


Book sessions

Все фрагменты кода из этой книги записаны в репозитории `igrishaev/book-sessions`[2]. Вы можете использовать код в любых целях, в том числе коммерческих.

Ресурсы

Следующие ресурсы помогут вам освоить язык и найти на нём работу.


Clojure

- Официальный сайт Clojure[3]. Его разделы «Getting Started», «Reference» и «Guides» подробно описывают язык и экосистему в целом. Прочтите их, даже если уверены в своих знаниях.


Slack Clojurians

- Сообщество в Slack под названием Clojurians[4]. Включает сотни каналов на разные темы, в том числе для отдельных библиотек и проектов. Каналы с кодами стран объединяют пользователей по языку. Есть канал `#ru` для русскоговорящих пользователей.


Telegram clojure_ru

- Чат в Телеграме[5] на русском языке. Основные темы: решение проблем, советы по оформлению кода, вакансии и поиск работы, анонсы мероприятий.


Ask Clojure

- Ask Clojure[6] — сервис вопросов и ответов по языку и его окружению, аналог StackOverflow.

[2] github.com/igrishaev/book-sessions
[3] clojure.org
[4] clojurians.slack.com
[5] t.me/clojure_ru
[6] ask.clojure.org

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] 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)
```

А это POST-запрос с JSON:

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

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

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

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."
}
```

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 равны `GET /about`, но такой страницы не существует. Сервер проверит путь по таблице маршрутов. Если его нет, получим ответ со статусом 404. Статус идёт раньше тела, что открывает путь для оптимизации. Логика клиента может быть такова, что, получив негативный статус, он пропустит ответ и потому справится быстрее. Подход выгоден и серверу, потому что ему не придётся пересылать тело.

Чтение и разбор всего содержимого занимает много времени. Современные фреймворки не делают этого зря. По заголовку `Content-Type` они определяют, стоит ли читать тело. Если приложение работает только с JSON, то для `text/xml` получим ошибку. 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`, что значит 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, потому что браузер сделает это сам.

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 сервер выключен и не отвечает на запросы.

Принято считать, что ответ со статусом, отличным от `2xx` означает ошибку. Многие HTTP-клиенты бросают исключение на ответ с негативным статусом. Это верно только на прикладном уровне, когда мы пишем код. 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. Обычно 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.

REST — не идеальный и не единственный подход к веб-разработке. Он конкурирует с JSON-RPC, gRPC и другими аналогами. В этой книге мы не будем задерживаться на конкретной парадигме. 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 в коде всё идеально.

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.

[4] restapitutorial.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 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. Ключевая особенность Spec в том, что она фундаментальна. Валидация данных — лишь малая часть её возможностей. Spec не только проверяет данные, но и преобразует их. На Spec легко писать парсеры.

Формально Spec — обычная библиотека. Но её абстракции настолько мощны, что Clojure переиспользует их. С версии 1.10

компилятор Clojure анализирует основные 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.

Ещё легче 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 представляет собой обычный валидатор. Он неявно сработает в выражении `new UnixPort(8080)`. Из-за неявности

возникает иллюзия, что мы создали тип. На самом 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` из разных библиотек. 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 — 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 строят 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 модулем `clojure.core`. Spec несёт макросы `s/and`, `s/or` и другие, у которых ничего общего с обычными `and` и `or`. Считается дурным тоном, когда 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 строит объект спеки из функции `string?` и помещает его в глобальный реестр с ключом `::string`.

Важно понимать, что `::string` — не спека, а псевдоним. Макросы Spec работают не с объектами спеки, а с ключам. На нижнем уровне они сами найдут спеку в реестре. Это удобно, потому что ключи глобальны: в любом месте можно написать `:my.project/string` без лишних импортов.

Вторым аргументом идёт предикат `string?`. Предикатом называют функцию, которая возвращает истину или ложь. Функция — это не спека, а строительный материал для неё. Спека оборачивает функцию в объект, который реализует внутренний протокол спеки. Технически на объект можно сослаться: функция `s/get-spec` вернёт его по ключу спеки. На практике объект спеки 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 её старую версию.

Spec не работает с ключами без пространства, например `:name` или `:email`. С ними легко случайно затереть одну спеку другой. Чтобы назначить ключу текущее пространство, поставьте два двоеточия: `::name`, `::email`. При запуске такая запись станет `:my.module/name` и `:my.module/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` происходит от «**n**on-**e**mpty string». Спека встречается часто, поэтому логично сэкономить на её имени.

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

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

Мы собрали 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.

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.

В своих программах новички следуют только положительному пути: пишут код так, словно исключений не может быть в принципе. Из-за этого 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 кода 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. С ним 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` (стандартный канал ошибок). Мы увидим его класс, текст и то, что называют «стектрейс». Это цепочка вызовов, которые прошло исключение с момента выброса до завершения программы.

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

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.

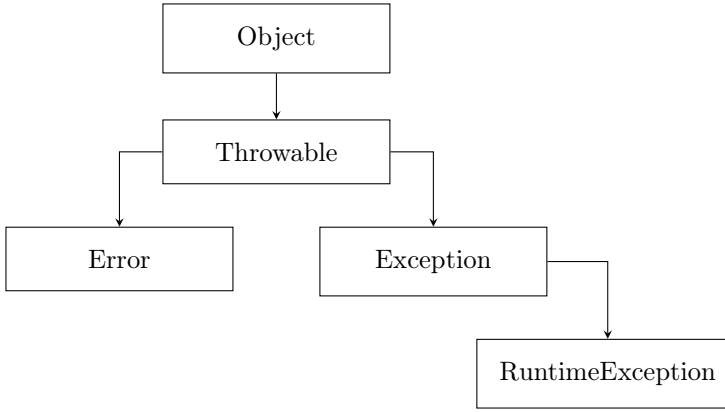

Рис. 3.1. Базовые классы исключений

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 и многие другие. Бросать `Throwable` считается дурным тоном: в этом классе 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). Как и любой ресурс, память ограничена, и неаккуратный код может её исчерпать. Перехватывать это исключение нет смысла, поскольку при нехватке памяти система ведёт себя нестабильно.

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 задан как `IOException`, мы поймаем все исключения, унаследованные от этого класса.

Чем выше класс в дереве наследования, тем больше исключений охватит `catch`. Однако 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 определить, какой именно файл не найден. 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 пароль. 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.

В конце управление перейдёт к методу, который знает, что делать. Логика зависит от типа исключения и бизнес-правил. Если файла нет, программа создаст его или выполнит поиск в другом месте. Если в HTTP-запросе произошёл сбой, подождём секунду и повторим его, а после третьей попытки завершим программу.

В системе должен быть последний рубеж, где перехватывают все исключения. Если ошибка дошла до этого уровня, значит, её не поймали правила ниже. Это говорит о нештатном поведении системы. Клиенту вернут текст о том, что запрос не прошёл. Чтобы расследовать причину, исключение пишут в лог и передают сборщику ошибок.

Глава 4

Изменяемость

В этой главе речь пойдёт об изменяемых типах. В этом плане Clojure занимает особую позицию. 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 нас постоянным коллекциям. Это правильный подход: чем труднее изменить данные, тем меньше шансов сделать это по ошибке. Но полностью избежать изменений невозможно, и вы должны знать, как управлять ими. Поэтому займём другую сторону: рассмотрим, как управлять состоянием в программах на Clojure.

4.1 Общие проблемы

Код на Clojure с трудом ложится на императивный стиль, который делает акцент на изменение данных. Скажем, чтобы получить список удвоенных чисел в классическом языке, мы должны:

- создать пустой список — будущий результат;
- пройти по элементам исходного списка;
- на каждом шаге вычислить новый элемент;

- добавить его к результату.

Выразим алгоритм в коде на Python:

```
numbers = [1, 2, 3]
doubles = []

for x in numbers:
    y = x * x
    doubles.append(y)

doubles # [1, 4, 9]
```

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

Автор 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.

Многие 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  def api_call(**params):
8      api_params = DEFAULT_PARAMS
9      api_params.update(params)
10     resp = requests.post(
11         "https://api.host.com", **api_params
12     )
13     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. Наше представление о коде не совпадает с тем, как он работает.

Другой пример: на собеседованиях часто показывают 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 состояние там, где оно не требуется:

```
(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)          (for [n [1 2 3 4 5]]
     [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. Прибегайте к нему только в крайних случаях. Если вы написали код с состоянием 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 весомый 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

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

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, но вместе с тем открыло новые парадигмы.

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 — 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, используем его и возвращаем. Для потребителя пул — примитивный объект, который выдаёт и забирает соединения.

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

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:

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 выполнить запрос.

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. Из-за конфликта портов вы не сможете запустить два сервиса одновременно.

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 верно, 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 делать пуш 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 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 проверяют до выпуска, но всегда остаётся риск, что в бою *что-то пойдёт не так*. В таком случае флаг меняют на ложь и перезапускают сервер. Это не только сэкономит время, но и сохранит репутацию фирмы.

5.3 Цикл конфигурации

Чем лучше устроено приложение, тем больше его частей опирается на параметры. Поэтому первое, что делает программа при запуске — ищет конфигурацию. Её обработка — не монолитная задача, а набор шагов. Перечислим наиболее важные из них.

На первом этапе программа **читает конфигурацию**. Чаще всего она находится в файле в формате JSON, YAML и других. Приложение содержит код, чтобы разобрать формат и получить данные. Мы рассмотрим плюсы и минусы известных форматов ниже.

Другой способ сообщить программе настройки — поместить их в переменные среды. Переменные — часть операционной системы, 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.

Возможна и обратная схема: приложение читает параметры из файла, а секретные данные — из переменных. Так поступают, чтобы другие программы, в том числе шпионские, не смогли прочитать из файлов пароли и ключи доступа.

Продвинутые конфигурации используют теги. Перед значением в файле ставят тег — короткую строку, которая означает, что значение нужно обработать особо. Например, в строке `:password #env DB_PASSWORD` тег `#env` говорит о том, что в `:password` окажется 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 и 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 с 1 января 1970 года (эпоха UNIX[1]). Специальный код приводит такую запись к типу даты, принятому в языке.


Unix time

Вывод типов применяют и для коллекций. Иногда словарей и массивов не хватает для комфортной работы. Если нужно указать допустимые значения чего-либо, логично сделать это множеством, потому что оно отсекает дубли и предлагает быструю проверку на вхождение.

Сложные типы удобнее описать скаляром (числом или строкой) и позже вывести их. Например, строка `"http://test.com"` станет экземпляром класса `java.net.URL`, а цепочка из 36 символов и дефисов — объектом `java.util.UUID`.

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.

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

После вывода типов приступают к **валидации данных**. В главе про Spec мы выяснили, что тип не обещает верное значение (с. 58). Проверка нужна, чтобы в конфигурации нельзя было указать порт 0 или -1.

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 любая проверка 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 отвечать 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 конфигурации, объясните, какое поле тому виной. В главе про Spec мы рассмотрели, как улучшить отчёт спеки (с. 80). 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`. Рано или поздно он попросит доработать сообщение 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 в данный момент программа не обращается к нему. Избегайте этого: ошибка появится в самый неподходящий момент.

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 в учётом зависимостей.

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 или 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. Своим 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 — 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 вызовет сбой. 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 исполнены в одном стиле. Если этому следовать, служебные компоненты уйдут в библиотеки, а в проекте останется только логика. Будет 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. 381).

6.2 Подготовка к обзору

В главе об изменяемых данных мы упоминали системы (с. 199). Тот способ работал 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 исследуем 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).

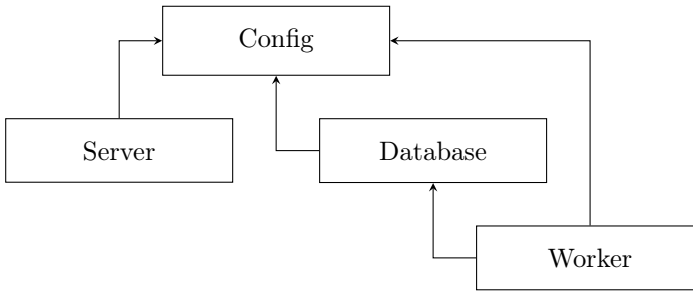

Рис. 6.1. Схема будущей системы с зависимостями

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.

6.2.1 База и воркер

Мы упоминали, что открывать соединение на каждый запрос не оптимально. В боевых проектах с базой работают через пул. Это сущность с состоянием, поэтому его тоже включают и выключают.

Базы вроде SQLite и H2 хранят данные в памяти. Это удобно для быстрого старта, но не отражает реалии производства — то, к чему мы стремимся в этой книге. Для in-memory баз не используют пулы соединений. В этом нет смысла: данные в памяти, а не в сети. Мы будем работать с реляционной БД PostgreSQL и пулом HikariCP.

Фоновый процесс (или воркер) дополняет записи из базы информацией, которую ищет в сети. Предположим, фирма ведёт аналитику посещений сайта. Когда кто-то открывает страницу, приложение сохраняет URL и IP-адрес клиента. Чтобы строить отчёты по странам и городам, нужно получить геоданные по IP из сторонних сервисов. Это долгая операция, поэтому записи ставят флаг «в обработке» и выносят логику в фон.

6.2.2 Docker

Если у вас установлен PostgreSQL, создайте новую базу и таблицу в ней. Если нет, самое время попробовать Docker. Это программа для запуска приложений из образов. Под образом понимают специальный файл, в котором упаковано приложение со всем необходимым для запуска. Запущенный образ называют контейнером.

У контейнеров несколько преимуществ. Приложение живёт в изолированной среде и поэтому отделено от основной системы. Кроме безопасности, это решает проблему чистоты — контейнер не оставляет следов работы, если это не задано специально.

Docker ищет образы в открытом репозитории. В нём публикуют программы разных версий и комплектации. Если нужен PostgreSQL версии строго 9.3, скачайте образ с нужным тегом. Без Докера поиск и установка этой версии займёт время и, вероятно, окончится конфликтом с уже работающей версией программы.

Некоторые образы можно настроить переменными среды или файлами. Образ PostgreSQL устроен так, что при старте он загружает все `*.sql` файлы из папки `/docker-entrypoint-initdb.d`. Если сопоставить ей локальный путь с миграциями, получим готовую базу. При этом мы не написали ни строчки кода, а только указали настройки.

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. Пусть мы написали 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)))))
```


Floating point

Примечание: в тесте мы сравниваем результат `->fahr` с целым числом. Без обёртки в `(int ...)` функция вернёт число с плавающей запятой, которое не всегда равно[1] другому такому же числу. Для удобства значение приводят к целому.

Макрос `assert` бросит исключение, если тело вычисляется в ложь. Пока что `(test-fahr)` работает без ошибок, потому что расчёты верны. Если изменить формулу, получим исключение:

```
(defn ->fahr [cel]
  (+ (* cel 1.9) 32)) ;; 1.8 -> 1.9

(test-fahr)
;; Assert failed: (= (int (->fahr 20)) 68)
```

Функция `test-fahr` и есть тест. Она проверяет, что код `->fahr` не изменили так, что получится другой результат. В наших интересах вызвать `(test-fahr)` перед сборкой программы. Так мы не допустим, чтобы код с ошибкой попал в производство.

Тест не знает устройства функции, которую проверяет. Мы вправе менять алгоритм до тех пор, пока тест работает без ошибок. Предположим, мы тестируем функцию факториала. В первой версии мы линейно умножаем числа, что неэффективно. Но уже на этой стадии добавим тест, что $5! = 120$.

```
(defn fact [n]
  (reduce * (range 1 (inc n))))

(defn test-fact []
  (assert (= 120 (fact 5))))
```

Если заменить линейный алгоритм на дерево или таблицу значений, тест пройдёт без ошибок. В противном случае это значит, что в новом алгоритме ошибка.

[1] en.wikipedia.org/wiki/Floating-point_arithmetic

```
(defn fact [n]
  (case n
    1 1
    2 2
    3 6
    4 24
    5 120))

(test-fact) ;; no errors
```

7.1.1 Кейсы и покрытие

Близкие по семантике тесты объединяются в кейсы (англ. набор, вариант). Вспомним школьную задачу с квадратным уравнением: найти корни по заданным коэффициентам a, b, c. Особенность задачи в том, что её логика ветвится. В зависимости от параметров может быть два корня, один или ни одного.

```
(defn square-roots [a b c]
  (let [D (- (* b b) (* 4 a c))]
    (cond
      (pos? D) [(/ (+ (- b) (Math/sqrt D)) (* 2 a))
                (/ (- (- b) (Math/sqrt D)) (* 2 a))]
      (zero? D) [(/ (- b) (* 2 a))]
      (neg? D) nil)))
```

Чтобы проверить алгоритм, тест вызывает функцию `square-roots` минимум три раза. Подберём параметры так, чтобы сработала каждая ветка. Ещё лучше, если на каждую ветку приходится отдельный тест, чтобы расширить его в будущем. Тесты ниже образуют кейс, который проверяет алгоритм целиком.

```
(defn test-square-roots-two-roots []
  (let [[x1 x2] (square-roots 1 -5 6)]
    (assert (= [3 2] [(int x1) (int x2)]))))
```

```
(defn test-square-roots-one-root []
  (let [[x1 x2] (square-roots 1 6 9)]
    (assert (= [-3] [(int x1)]))
    (assert (nil? x2))))

(defn test-square-roots-no-roots []
  (assert (nil? (square-roots 2 4 7))))
```

В объектно-ориентированных языках кейсы — это классы, а тесты — их методы. В Clojure более простой подход: тест — это функция, а роль кейса играет пространство.

В тестах часто упоминают *покрытие*. Так называют долю кода, которая сработала в тесте. Фреймворк считает общее число строк и сколько из них выполнились. Покрытие — это отношение второй величины к первой, число от нуля до единицы.

Предположим, мы написали тест только для двух корней. В этом случае ветки `(zero? D)` и `(neg? D)` не сработают, и доля покрытия уменьшится.

Некоторые библиотеки строят отчёт, где выполненные строки отмечены зелёным (плюсом), а пропущенные — красным (минусом). Подсчёт покрытия — трудная задача, поэтому её выносят в расширение фреймворка. Для нашего случая мы получили бы вывод:

```
+(defn square-roots [a b c]
+  (let [D (- (* b b) (* 4 a c))]
+    (cond
+      (pos? D) [(/ (+ (- b) (Math/sqrt D)) (* 2 a))
+                (/ (- (- b) (Math/sqrt D)) (* 2 a))]
-      (zero? D) [(/ (- b) (* 2 a))]
-      (neg? D) nil)))
```

В функции `square-roots` семь строк. При запуске теста сработали пять из них. Покрытие составит $5/7$, что приблизительно 71%. Считается, что покрытия 90% достаточно, чтобы код работал надёжно. Значения ниже говорят о малом покрытии. Это значит, в проекте встречается код без защиты от спонтанных изменений. При попытке приблизиться к 100% тесты становятся затратными

в поддержке. В каждой команде следуют той величине, которая удобна процессу.

Величина в процентах не должна затмевать здравый смысл. Цифра 71%кажется большой только на первый взгляд. Из покрытия видно, что мы проверяем лишь одну ветку алгоритма из трёх. Если в двух других ошибки, мы не узнаем о них. При оценке покрытия смотрят не на проценты, а на развилки алгоритма. Надёжный тест гарантирует, что сработала каждая ветка.

7.1.2 Не только числа

У начинающих складывается мнение, что тесты пишут только для математических расчётов. На самом деле тесты подходят для всех действий, которые хотят зафиксировать. Это может быть обход коллекции, криптография, поиск в тексте.

Рассмотрим подпись HTTP-запроса. Функция `sign-params` принимает словарь параметров и секретный ключ. Алгоритм подписи следующий:

- отсортировать параметры по ключам;
- составить строку `param1=value1&param2=value2...`;
- экранировать пробел, процент и другие служебные символы;
- получить сигнатуру строки по алгоритму `HMAC-SHA256` и секретному ключу;
- вернуть словарь параметров с полем `:signature`.

Так работает авторизация в популярных сервисах. Напишем тест для функции подписи. Проверим, что входные параметры дополнены сигнатурой, которую посчитали заранее.

Что дальше

Дорогой читатель! Если вы дошли сюда от начала книги, примите искренние поздравления. Это был трудный путь, но с наградой в конце. Теперь вы знаете больше, сто́ите больше и можете сделать больше.

Чтобы удержать знания, поскорей закрепите их практикой. Устройтесь в проект на Clojure и отточите навыки в работе. Если компания не использует этот язык, предложите руководству эксперимент. Заинтересуйте коллег, проведите мастер-класс и сделайте проект на Clojure. Презентуйте его руководству и опишите преимущества: неизменяемость, скорость разработки, простота.

Русское сообщество языка развивается. Если возникли трудности, обращайтесь в Телеграм-канал `clojure_ru`. На канале `clojure_jobs` публикуют резюме и вакансии, связанные с Clojure. Автор отвечает на письма по адресу `ivan@grishaev.me` и в блоге `grishaev.me`.

Желаю читателю успеха во всех начинаниях.

Иван Гришаев,
Россия — Швейцария,
2019–2021

Предметный указатель

K

Л

М

Н