

Функциональное программирование

Чаз Эмерик
Брайан Карпер
Кристоф Гранд

Программирование на Clojure

Практика применения
Lisp в мире Java

ОМК
ПРОС
ИЗДАТЕЛЬСТВО

УДК 004.432.42Clojure
ББК 32.973-018.1
Э54

- Эмерик Ч., Карпер Б., Гранд К.
- Э54 Программирование на Clojure: Пер. с англ. Киселева А. Н. – М.: ДМК Пресс, 2013. – 816 с.: ил.

ISBN 978-5-94074-925-7

Почему многие выбирают Clojure? Потому что это функциональный язык программирования, не только позволяющий использовать Java-библиотеки, службы и другие ресурсы JVM, но и соперничающий с другими динамическими языками, такими как Ruby и Python.

Эта книга продемонстрирует вам гибкость Clojure в решении типичных задач, таких как разработка веб-приложений и взаимодействие с базами данных. Вы быстро поймете, что этот язык помогает устранить ненужные сложности в своей практике и открывает новые пути решения сложных проблем, включая многопоточное программирование.

Издание предназначено для программистов, желающих освоить всю мощь и гибкость функционального программирования.

УДК 004.432.42Clojure
ББК 32.973-018.1

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-449-39470-7 (анг.)

ISBN 978-5-94074-925-7 (рус.)

Copyright © 2012 Chas Emerick,
Brian Carper, and Christophe Grand
© Оформление, перевод
ДМК Пресс, 2013



Содержание

Предисловие к русскому изданию	15
Благодарности	16
Предисловие	17
Глава 1. Вниз по кроличьей норе	26
Почему Clojure?	26
Как получить Clojure	29
Интерактивная оболочка REPL для Clojure	30
Вам не придется путаться в частоколе скобок	34
Выражения, операторы, синтаксис и очередность	35
Гомоиконность	38
Механизм чтения	41
Скалярные литералы	43
Строки	43
Логические значения	43
nil	43
Знаки (characters)	44
Ключевые слова (keywords)	44
Символы (symbols)	46
Числа	46
Регулярные выражения	48
Комментарии	49
Пробелы и запятые	51
Литералы коллекций	51
Прочий синтаксический сахар механизма чтения	52
Пространства имен	53
Интерпретация символов	56
Специальные формы	57
Подавление вычислений: quote	59
Блоки кода: do	60
Определение переменных: def	61

Связывание локальных значений: let	62
Деструктуризация (let, часть 2).....	64
Деструктуризация упорядоченных коллекций	65
Деструктуризация ассоциативных массивов	69
Создание функций: fn.....	74
Деструктуризация аргументов функций	77
Литералы функций	80
Выполнение по условию: if	82
Организация циклов: loop и recur	83
Ссылки на переменные: var	85
Взаимодействие с Java: . и new	86
Обработка исключений: try и throw	86
Специализированная операция set!	87
Примитивы блокировок: monitor-enter и monitor-exit	87
Все вместе	87
eval.....	88
Это лишь начало.....	90

Часть I. ФУНКЦИОНАЛЬНОЕ И КОНКУРЕНТНОЕ ПРОГРАММИРОВАНИЕ 91

Глава 2. Функциональное программирование 92

Что подразумевается под термином «Функциональное программирование»?	93
О важности значений.....	94
О значениях	95
Сравнение значений изменяемых объектов	96
Важность выбора	101
Функции, как сущности первого порядка, и функции высшего порядка	103
Частичное применение	111
Композиция функций	116
Создание функций высшего порядка.....	120
Создание простейшей системы журналирования с применением композиции функций высшего порядка	121
Чистые функции	126
В чем преимущество чистых функций?.....	129
Функциональное программирование в реальном мире	132

Глава 3. Коллекции и структуры данных 134

Главенство абстракций над реализациями	135
Коллекции	139
Последовательности	142

Последовательности не являются итераторами	145
Последовательности не являются списками	146
Создание последовательностей	147
Ленивые последовательности	148
Удержание мусора	156
Ассоциативные коллекции	157
Берегитесь значения nil	161
Индексирование	162
Стек	164
Множество	165
Сортированные коллекции	166
Определение порядка с помощью компараторов и предикатов	168
Упрощенный доступ к коллекциям	173
Идиоматические приемы использования	175
Коллекции, ключи и функции высшего порядка	176
Типы структур данных	177
Списки	178
Векторы	179
Векторы как кортежи	180
Множества	181
Ассоциативные массивы	182
Ассоциативные массивы как специализированные структуры	183
Другие применения ассоциативных массивов	185
Неизменяемость и сохранность	189
Сохранность и совместное использование	190
Визуализация сохранности: списки	191
Визуализация сохранности: ассоциативные массивы (векторы и множества)	193
Очевидные преимущества	196
Переходные структуры данных	198
Метаданные	205
Включаем коллекции Clojure в работу	207
Идентификаторы и циклы	208
Думайте иначе: от императивного к функциональному	210
Вспоминаем классику: игра «Жизнь»	211
Генерация лабиринтов	220
Навигация, изменение и зипперы (zippers)	228
Манипулирование зипперами	229
Собственные зипперы	231
Зиппер Ариадны	232
В заключение	236

Глава 4. Конкуренция и параллелизм	237
Сдвиг вычислений в пространстве и времени.....	238
delay.....	238
Механизм future	241
Механизм promise	243
Параллельная обработка по невысокой цене.....	246
Состояние и идентичность	250
Ссылочные типы.....	253
Классификация параллельных операций.....	255
Атомы.....	258
Уведомление и ограничение.....	261
Функции-наблюдатели	261
Функции-валидаторы	264
Ссылки.....	266
Программная транзакционная память	266
Механика изменения ссылок	268
Функция alter.....	271
Уменьшение конфликтов в транзакциях с помощью commute ...	273
Затирание состояния ссылки с помощью ref-set.....	279
Проверка локальной согласованности с помощью валидаторов.....	279
Острые углы программной транзакционной памяти	283
Функции с побочными эффектами строго запрещены	283
Минимизируйте продолжительность выполнения транзакций.....	284
Читающие транзакции могут повторяться	287
Искажение при записи	289
Переменные.....	291
Определение переменных.....	292
Приватные переменные	293
Строки документации.....	294
Константы	295
Динамическая область видимости	296
Переменные в языке Clojure не являются переменными в классическом понимании	303
Опережающие объявления.....	305
Агенты.....	307
Обработка ошибок в заданиях агентов	310
Режимы и обработчики ошибок в агентах	312
Ввод/вывод, транзакции и вложенная передача заданий.....	313
Сохранение состояний ссылок в журнале на основе агента....	315
Использование агентов для распределения нагрузки.....	318
Механизмы параллельного выполнения в Java	328
Блокировки	329
В заключение	330

Часть II. СОЗДАНИЕ АБСТРАКЦИЙ	331
Глава 5. Макросы	332
Что такое макрос?	333
Чем не являются макросы	335
Что могут макросы, чего не могут функции?	336
Сравнение макросов и механизма eval в Ruby	339
Пишем свой первый макрос	341
Отладка макросов	343
Функции развертывания макросов	344
Синтаксис	346
Сравнение quote и syntax-quote	348
unquote и unquote-splicing	349
Когда следует использовать макросы	351
Гигиена	353
Генераторы символов во спасение	355
Предоставление пользователю права выбора имен	359
Двукратное вычисление	360
Распространенные идиомы и шаблоны макросов	362
Неявные аргументы: &env и &form	364
&env	364
&form	367
Вывод сообщений об ошибках в макросах	367
Сохранение определений типов, сделанных пользователем	370
Тестирование контекстных макросов	373
Подробности: -> и ->>	375
В заключение	379
Глава 6. Типы данных и протоколы	380
Протоколы	381
Расширение существующих типов	383
Определение собственных типов	389
Записи	392
Конструкторы и фабричные функции	396
Когда использовать ассоциативные массивы, а когда записи ...	398
Типы	399
Реализация протоколов	402
Встроенная реализация	403
Встроенные реализации интерфейсов Java	405
Определение анонимных типов с помощью reify	407
Повторное использование реализаций	408
Интроспекция протоколов	413
Пограничные случаи использования протоколов	415

Поддержка абстракций коллекций.....	417
В заключение	427
Глава 7. Мультиметоды.....	428
Основы мультиметодов	428
Навстречу иерархиям	431
Иерархии	434
Независимые иерархии.....	437
Сделаем выбор по-настоящему множественным!	441
Кое что еще	443
Множественное наследование	443
Интроспекция мультиметодов	445
type и class; или месть ассоциативного массива	446
Функции выбора не имеют ограничений.....	447
В заключение	449
Часть III. ИНСТРУМЕНТЫ, ПЛАТФОРМЫ И ПРОЕКТЫ	450
Глава 8. Создание и организация проектов на Clojure	451
География проекта	451
Определение и использование пространств имен.....	452
Пространства имен и файлы	461
Знакомство с classpath	465
Местоположение, местоположение, местоположение	467
Организация программного кода по функциональным признакам ...	469
Основные принципы организации проектов	471
Сборка	472
Предварительная компиляция.....	473
Управление зависимостями	476
Модель Maven управления зависимостями	477
Артефакты и координаты.....	477
Репозитории	479
Зависимости.....	480
Инструменты сборки и шаблоны настройки.....	483
Maven.....	484
Leiningen	488
Настройка предварительной компиляции	491
Сборка гибридных проектов.....	493
В заключение	496
Глава 9. Java и взаимодействие с JVM	497
JVM – основа Clojure.....	498
Использование классов, методов и полей Java	499

Удобные утилиты взаимодействий	503
Исключения и обработка ошибок	506
Отказ от контролируемых исключений	509
with-open, прощай finally	510
Указание типов для производительности	512
Массивы	518
Определение классов и реализация интерфейсов	519
Экземпляры анонимных классов: proху	520
Определение именованных классов	523
gen-class	524
Аннотации	532
Создание аннотированных тестов для JUnit	533
Реализация конечных точек веб-службы JAX-RS	534
Использование Clojure из Java	537
Использование классов, созданных с помощью deftype и defrecord	541
Реализация интерфейсов протоколов	544
Сотрудничество	546

Глава 10. REPL-ориентированное программирование

Интерактивная разработка	547
Постоянное изменяющееся окружение	552
Инструменты	554
Оболочка REPL	555
Интроспекция пространств имен	557
Eclipse	560
Emacs	563
clojure-mode и paredit	564
inferior-lisp	565
SLIME	567
Отладка, мониторинг и исправление программ в REPL во время эксплуатации	570
Особые замечания по поводу «развертываемых» оболочек REPL	574
Ограничения при переопределении конструкций	576
В заключение	579

Часть IV. ПРАКТИКУМ

Глава 11. Числовые типы и арифметика

Числовые типы в Clojure	581
В Clojure предпочтение отдается 64-битным (или больше) представлениям	583
Clojure имеет смешанную модель числовых типов	583

Рациональные числа	586
Правила определения типа результата	587
Арифметика в Clojure	589
Ограниченная и произвольная точность	589
Неконтролируемые операции	593
Режимы масштабирования и округления в операциях с вещественными числами произвольной точности.....	595
Равенство и эквивалентность	597
Идентичность объектов (identical?).....	597
Равенство ссылок (=).....	598
Числовая эквивалентность (==)	600
Эквивалентность может защитить ваш рассудок.....	601
Оптимизация производительности операций с числами	603
Объявление функций, принимающих и возвращающих значения простых типов	604
Ошибки и предупреждения, вызванные несоответствием типов	608
Используйте простые массивы осмысленно.....	610
Механика массивов значений простых типов	613
Автоматизация указания типов в операциях с многомерными массивами	618
Визуализация множества Мандельброта в Clojure	620
Глава 12. Шаблоны проектирования	629
Внедрение зависимостей.....	631
Шаблон Стратегия (Strategy)	636
Цепочка обязанностей (Chain of Responsibility)	638
Аспектно-ориентированное программирование	642
В заключение	647
Глава 13. Тестирование.....	648
Неизменяемые значения и чистые функции	648
Создание фиктивных значений.....	649
clojure.test	651
Определение тестов	653
«Комплекты» тестов	656
Крепления (fixtures)	658
Расширение HTML DSL.....	662
Использование контрольных проверок.....	668
Предусловия и постусловия	670
Глава 14. Реляционные базы данных	673
clojure.java.jdbc	673
Подробнее о with-query-results	678

Транзакции.....	680
Пулы соединений	681
Korma.....	682
Вступление	683
Запросы	685
Зачем использовать предметно-ориентированный язык?	686
Hibernate	689
Настройка	690
Сохранение данных.....	694
Выполнение запросов.....	695
Избавление от шаблонного кода	695
В заключение	698
Глава 15. Нереляционные базы данных.....	699
Настройка CouchDB и Clutch	700
Простейшие CRUD-операции.....	701
Представления.....	703
Простое представление (на JavaScript)	704
Представления на языке Clojure	706
_changes: использование CouchDB в роли очереди сообщений	711
Очереди сообщений на заказ	713
В заключение	717
Глава 16. Clojure и Веб.....	718
«Стек Clojure»	718
Основа: Ring.....	720
Запросы и ответы.....	721
Адаптеры	724
Обработчики	725
Промежуточные функции	727
Маршрутизация запросов с помощью Compojure	729
Обработка шаблонов.....	743
Enlive: преобразование HTML с применением селекторов.....	745
Попробуем воду	746
Селекторы	748
Итерации и ветвление	750
Объединяем все вместе.....	752
В заключение	756
Глава 17. Развертывание веб-приложений на Clojure.....	758
Веб-архитектура Java и Clojure	758
Упаковка веб-приложения	762
Сборка .war-файлов с помощью Maven	764



Сборка .war-файлов с помощью Leiningen	767
Запуск веб-приложений на локальном компьютере	769
Развертывание веб-приложения	770
Развертывание приложений на Clojure с помощью	
Amazon Elastic Beanstalk	771
За пределами развертывания простых веб-приложений	775
Часть V. РАЗНОЕ	776
Глава 18. Выбор форм определения типов	777
Глава 19. Внедрение Clojure	780
Только факты.....	780
Подчеркните особую продуктивность	782
Подчеркните широту сообщества.....	784
Будьте благоразумны	786
Глава 20. Что дальше?	788
(dissoc Clojure 'JVM).....	788
ClojureCLR.....	788
ClojureScript	789
4Clojure	790
Overtone	790
core.logic	791
Pallet	792
Avout	793
Clojure на платформе Heroku	793
Об авторах.....	795
Иллюстрация на обложке	796
Предметный указатель	797



Глава 2. Функциональное программирование

Словосочетание «функциональное программирование» (ФП) является одним из тех аморфных понятий в разработке ПО, которые разные люди понимают по-разному. Несмотря на все полутона и оттенки, имеющиеся в понятии ФП, можно уверенно утверждать, что Clojure является языком функционального программирования, и эта его особенность является причиной его привлекательности.

Далее мы:

1. Расскажем, что такое «функциональное программирование».
2. Объясним, зачем оно необходимо.
3. Обсудим особенности Clojure, которые делают его языком функционального программирования.

Попутно мы надеемся показать, что ФП – и Clojure, как язык ФП в частности, – представляет не только академический интерес и способно ваши расширить возможности в проектировании и разработке ПО, также как структурное и объектно-ориентированное программирование.

Если вы уже знакомы с функциональным программированием (например, благодаря использованию языка Ruby или JavaScript, или даже таких функциональных языков, как Scala, F# или Haskell), большая часть из того, о чем будет рассказываться ниже, покажется вам хорошо знакомой, но эти сведения совершенно необходимы, чтобы вы могли понять функциональные черты Clojure.

Если вы вообще не знакомы с функциональным программированием или испытываете скепсис относительно его практической ценности, мы настоятельно рекомендуем продолжить чтение – это стоит ваших сил и времени¹. В главе 1 уже говорилось, что *язык Clojure*

¹ После знакомства со сведениями в этой главе, вы можете обнаружить, что страница в Википедии, посвященная функциональному программированию (http://ru.wikipedia.org/wiki/Функциональное_программирование) является удивительно хорошим трамплином для погружения в родственные темы.

требует вложить время и силы в его изучение, но все затраты окупятся сторицей. Так же, как вам, возможно, пришлось приложить силы для освоения объектно-ориентированного программирования, или шаблонов (generics) в языке Java, или языка Ruby, точно так же придется приложить некоторые усилия, чтобы освоить приемы ФП и, соответственно, язык Clojure. Однако взамен вы получите не только «новое мышление», но еще и множество инструментов и практических приемов, пригодных для использования в повседневной работе¹.

Что подразумевается под термином «Функциональное программирование»?

Функциональное программирование – это обобщающее понятие, охватывающее множество различных языковых конструкций и механизмов, которые в разных языках реализуются по-разному. В Clojure термин «функциональное программирование» означает:

- ❑ предпочтительное положение неизменяемых значений, в том числе:
 - использование неизменяемых структур данных, удовлетворяющих простым абстракциям, вместо свободно изменяемого состояния;
 - интерпретация функций как значений, что позволяет создавать *функции высшего порядка*;
- ❑ предпочтительное положение декларативной обработки данных перед императивными структурами ветвления и управления циклами;
- ❑ естественное пошаговое конструирование функций, функций высшего порядка и неизменяемых структур данных, решение сложных задач за счет использования высокоуровневых абстракций.

Все это является основой для множества улучшенных возможностей языка Clojure, о которых вы могли слышать. Особенно хо-

¹ Обратите внимание, что приемы функционального программирования *можно использовать* даже в таких языках, как Java, которые не поощряют (а иногда даже активно противодействуют) стилю ФП. Делать это намного проще, если в вашем распоряжении имеются неизменяемые структуры данных и реализации основных примитивов ФП, подобные тем, что предоставляются библиотеками из проектов Google Guava (<https://code.google.com/p/guava-libraries/>) и Functional Java (<http://functionaljava.org>).

телось бы отметить фантастическую поддержку многозадачности и параллельного программирования в языке Clojure, а также наличие семантики управления идентичностями (*identities*) и состояниями, более подробно описываемых в главе 4.

О важности значений

Понятие *состояние программы* имеет достаточно широкое толкование и длинную историю, но в общем случае под ним подразумеваются все скаляры и структуры данных, используемые для представления сущностей внутри приложения, а также всех связей приложения с внешним миром (таких как открытые файлы, сокет и прочее). Характер языка программирования в значительной степени определяется его отношением к обработке состояния: что позволяет, чему препятствует и что приветствует.

Большинство языков программирования, посредством идиом или явного синтаксиса, поощряет использование *изменяемого* состояния в виде объектов или структур данных. Языки функционального программирования, напротив, поощряют использование *неизменяемых* объектов — называемых *значениями* — для представления состояния программы. Язык Clojure не является исключением в этом отношении.

«Но, минутку!», — можете сказать вы: «Разговор об отказе от изменяемости данных не имеет смысла. Программа *должна взаимодействовать с окружающим миром*, поэтому изменение ее состояния неизбежно.» Вы определенно правы, что любая, мало-мальски полезная программа должна взаимодействовать с окружающим миром, чтобы получать

исходные данные и возвращать результаты, но это не препятствует использованию неизменяемых значений. Напротив, чем больше операций с неизменяемыми значениями будет выполняться в программе, тем проще будет анализировать ее поведение, в сравнении с программой, опирающейся на использование изменяемого состояния. На протяжении этой главы нам не раз представится возможность убедиться в правоте этих слов.

Переход от изменяемого состояния и объектов к неизменяемым значениям для мно-

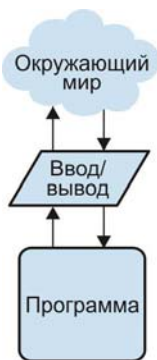


Рис. 2.1. Организация типичной программы

гих может оказаться непростым делом, но, возможно, вас подбодрит знание, что вы, как программист, уже пользуетесь неизменяемыми значениями в повседневной практике, и что они являются самыми надежными элементами в вашем приложении.

О значениях

Что такое «значения» и чем отличаются неизменяемые значения от изменяемых объектов? Ниже приводится несколько примеров значений:

```
true false 5 14.2 \T "hello" nil
```

Это – стандартные логические, числовые, символьные и строковые значения в JVM, которые также используются в Clojure. Все они являются неизменяемыми, все они являются значениями и всеми ими (или их аналогами в других языках) вы пользуетесь каждый день.

Ключевой особенностью значений является поддержка семантики равенства и сравнения. Например, следующие выражения *всегда* возвращают значение `true`:

```
(= 5 5)

(= 5 (+ 2 3))

(= "boot" (str "bo" "ot"))

(= nil nil)

(let [a 5]
  (do-something-with-a-number a)
  (= a 5))
```

Эквивалентные выражения на языках Java, Python и Ruby так же *всегда* возвращают `true`¹, и этот факт позволяет нам рассуждать об операциях, вовлекающих такие значения. Чтобы понять причину,

¹ Исключение составляет выражение `(= nil nil)`; многие языки имеют весьма ограниченную поддержку сравнения со значением `nil` или `null`, и обычно передача указателя `null` методу `equals` объекта вызывает неприятности. Однако в Clojure значение `nil` является самым обычным значением.

было бы полезно увидеть, что может произойти, если один из таких хорошо знакомых типов потеряет семантику значения.

Сравнение значений изменяемых объектов

Выбор между изменяемыми объектами и неизменяемыми значениями является важным решением, имеющим далеко идущие последствия, даже в самых простых случаях. Однако это решение часто принимается на основе сложившихся привычек, без учета возможных последствий. Поскольку состояние хранится в виде объектов, доступных для изменения, и может изменяться без вашего ведома, их использование при наличии неизменяемых альтернатив может быть опасным.

Это утверждение может показаться преувеличением, особенно для тех, кто использует изменяемые объекты в повседневной практике и не наблюдает особых проблем. Однако, давайте посмотрим, что может произойти, если сделать изменяемым привычное неизменяемое значение, такое как целое число:

Пример 2.1. Реализация изменяемого целого числа на языке Java

```
public class StatefulInteger extends Number {
    private int state;

    public StatefulInteger (int initialState) {
        this.state = initialState;
    }

    public void setInt (int newState) {
        this.state = newState;
    }

    public int intValue () {
        return state;
    }

    public int hashCode () {
        return state;
    }

    public boolean equals (Object obj) {
        return obj instanceof StatefulInteger &&
            state == ((StatefulInteger)obj).state;
    }

    // остальные методы xxxValue() класса java.lang.Number...
}
```

Этот класс в значительной степени идентичен классу `java.lang.Integer`, за исключением отсутствия различных статических методов. Самое важное изменение в этом классе – единственное поле в нем является изменяемым (то есть, оно не объявлено финальным (`final`)) и для него предусмотрен метод записи, `setInt(int)`. Посмотрим, как можно использовать это изменяемое число¹:

```
(def five (StatefulInteger. 5))      ❶
;= #'user/five
(def six (StatefulInteger. 6))
;= #'user/six
(.intValue five)                    ❷
;= 5
(= five six)                        ❸
;= false
(.setInt five 6)                    ❹
;= nil
(= five six)                        ❺
;= true
```

- ❶ Создается пара экземпляров класса `StatefulInteger`, один со значением 5, а второй – со значением 6.
- ❷ Проверяется, что объект `five` действительно содержит значение «5»...
- ❸ ...и 5 не равно 6.
- ❹ Изменим значение объекта `five` на 6, и...
- ❺ теперь объект `five` стал равен 6.

Такое положение вещей должно вызывать сильное беспокойство. Мы привыкли к тому, что числа *действительно являются* значениями, причем, не только в техническом смысле. Число 5 всегда должно быть равно 5, и не должно изменяться по прихоти программиста.

Заключительный пример, иллюстрирующий последствия применения изменяемых объектов в вызовах функций или методов:

```
(defn print-number                    ❶
  [n]
  (println (.intValue n))
  (.setInt n 42))
```

¹ Мы будем работать с этим Java-классом из программного кода на языке Clojure, который обладает богатыми возможностями взаимодействия с Java и JVM. Подробности см. в главе 9.

```
;= #'user/print-number
(print-number six)           ❷
; 6
;= nil
(= five six)                 ❸
;= false
(= five (StatefulInteger. 42))
;= true
```

- ❶ Простейшая функция `print-number`, которая якобы должна просто вывести значение заданного числа в `stdout`. Однако, без нашего ведома она изменяет аргумент `StatefulInteger`¹.
- ❷ Вызов функции `print-number` и передача ей экземпляра `six` класса `StatefulInteger`. Когда функция вернет управление, объект окажется изменен.
- ❸ Теперь, в противоположность предыдущему примеру, объект `six` более не равен объекту `five`.

В общем случае функция, получающая изменяемые аргументы, как в примере выше, не должна быть вредоносной и необязательно написана некомпетентным программистом. Изменение аргументов внутри функций и методов является распространенным явлением и вам, возможно, уже приходилось сталкиваться (и даже вносить самому!) с ошибками, вызванными изменением внутри метода объекта, переданного в виде аргумента. То же относится и к изменяемым объектам, доступным глобально. Соответственно, никакая документация (которая читается еще реже, чем пишется!) не способна защитить от подобных ситуаций, и такие ловушки, вызванные изменяемостью, являются серьезным основанием для организации глубокого копирования объектов и создания конструкторов копирования.

Если бы числа в вашем языке программирования действовали аналогичным образом, вы могли бы завтра же бросить свою работу и заняться более простым делом. К сожалению, практически все объекты *действуют* именно таким способом.

В Ruby изменяемыми являются даже строки, обычно неизменяемые в других языках. Эта особенность может стать источником самых разнообразных проблем:

¹ Мы могли бы реализовать эту функцию как Java-метод. Однако для нашего обсуждения гораздо удобнее использовать функцию на Clojure.

```
>> s = "hello"
=> "hello"
>> s << "*"
=> "hello*"
>> s == "hello"
false
```

Коллекции в языке Ruby, такие как хеши и множества, как бы «замораживают» строки после их добавления в коллекцию. Однако, во всех классах и структурах, создаваемых вами и содержащих строки, необходимо соблюдать меры предосторожности, чтобы предотвратить появление серьезных проблем, относящихся к той же разновидности ошибок, возникающих при использовании изменяемых объектов в качестве ключей в хешах:

```
>> h = {[1, 2] => 3}      ❶
=> {[1, 2]=>3}
>> h[[1,2]]              ❷
=> 3
>> h.keys[0] << 3        ❸
=> [1, 2, 3]
>> h[[1,2]]              ❹
=> nil
```

- ❶ Создается хеш (отображение, или ассоциативный массив), отображающий массив в число...
- ❷ ...который, как предполагается, возвращает число, соответствующее искомому массиву.
- ❸ Если какой-либо ключ хеша изменится...
- ❹ ...попытки отыскать массив, который прежде обнаруживался без проблем, будут терпеть неудачу¹.

Проблемы, подобные этой, присутствуют во всех языках программирования, где имеются изменяемые объекты, но их влияние особенно пагубно в языках, где неизменяемые значения используются редко: многие программисты вынуждены изучать сложные в применении

¹ Да, мы могли бы воспользоваться методом хеша `rehash`, что «решило» бы проблему. Такой способ пригоден, когда имеется возможность перестроить ассоциативный массив перед каждой попыткой поиска, но он может снижать эффективность, блокируя доступ к ассоциативному массиву на период перестройки и поиска в хеше, в многопоточной среде выполнения.

приемы решения подобных проблем. Поэтому, даже когда для решения некоторой задачи наиболее эффективным, к примеру, является использование ассоциативного массива с ключами-коллекциями, прошлый опыт мешает им использовать простые решения и вынуждает использовать более сложные и в общем случае менее эффективные подходы. Что лучше? Создать новый класс, содержащий ассоциативный массив и имеющий ограниченный набор операций с ним? Использовать простые неизменяемые коллекции, обеспечивающие адекватную семантику, но требующие выполнять полное копирование, чтобы создать измененную версию? Как оказывается, выбор не прост.

В противоположность этому, в языке Clojure можно безопасно использовать коллекции в ассоциативных массивах (а также в множествах, векторах или записях), не заботясь о значениях ключей или об особенностях использования в многопоточной среде, потому что структуры данных в Clojure являются неизменяемыми и эффективными:

```
(def h {[1 2] 3})      ❶
;= #'user/h
(h [1 2])
;= 3
(conj (first (keys h)) 3)  ❷
;= [1 2 3]
(h [1 2])                ❸
;= 3
h                          ❹
;= {[1 2] 3}
```

- ❶ Создается ассоциативный массив Clojure, отображающий ключ-вектор в числовое значение и обеспечивающий ожидаемую семантику поиска.
- ❷ мы можем «добавить» значение в ключ-вектор и получить в результате «измененный» вектор. Но...
- ❸ ...это никак не повлияет на сам ассоциативный массив, на вектор, используемый в качестве ключа, и на успешные в прошлом попытки выполнить поиск. Все это обусловлено тем, что...
- ❹ ...ключ-вектор не изменился и в действительности *никогда не изменится*. При добавлении нового значения в действительности был создан совершенно новый вектор.

Но это лишь вершина айсберга. Понимание и умение использовать абстракции и реализации структур данных в языке Clojure является залогом эффективного владения языком. Пока что, в этой главе,

мы будем использовать структуры данных Clojure неформально, но всю следующую главу мы посвятим их детальному исследованию.

Важность выбора

В общем случае использование объектов, не ограничивающих возможность их изменения, означает, что¹:

- ❑ изменяемые объекты не могут безопасно передаваться методам;
- ❑ изменяемые объекты не обеспечивают должной надежности при использовании в качестве ключей в ассоциативных массивах, элементов в множествах, и так далее, потому что их семантика равенства может изменяться с течением времени;
- ❑ изменяемые объекты не обеспечивают должной надежности при кешировании;
- ❑ изменяемые объекты не могут безопасно использоваться в многопоточной среде выполнения, потому что требуют синхронизации выполнения операций с ними в разных потоках.

Целые классы ошибок, возникающие при использовании изменяемых объектов, становятся просто невозможными при переходе к использованию неизменяемых значений. Отрицательные последствия изменчивости в действительности изучены достаточно давно² и проявляются во всех языках, где отсутствуют простые неизменяемые альтернативы. Эти проблемы настолько общие, что в объектно-ориентированном мире был разработан целый комплекс механизмов копирования для решения проблем, связанных с неограниченной изменяемостью состояния, включая.

Конструкторы копий и методы глубокого копирования, гарантирующие безопасный доступ к объекту в определенном состоянии³.

¹ Брайен Гетц (Brian Goetz) подробно рассматривает все эти проблемы в статье по адресу <http://www.ibm.com/developerworks/java/library/j-jtp02183.html>.

² Например, Джошуа Блох (Joshua Bloch), один из главных идеологов стандартной библиотеки Java, рекомендовал «минимизировать изменчивость» (<http://www.artima.com/intv/bloch11.html>) и в своей книге «Effective Java» (Д. Блох, «Java. Эффективное программирование», Лори, 2002, ISBN: 5-85582-169-2. – Прим. перев.), в вышедшей еще в 2001 году, говорил, что: «Классы должны оставаться неизменяемыми, если нет веской причины делать их изменяемыми».

³ В некоторых ситуациях дело доходит даже до использования сериализации в качестве механизма глубокого копирования, когда объектная модель не предусматривает наличия надежного конструктора копий или методов дублирования.

Множество шаблонов проектирования, направленных на отслеживание изменений и управления ими, включая шаблоны «Наблюдатель» («Observer»), «Реактор» («Reactor») и другие¹.

Массу утилит, создающих относительно тонкие неизменяемые обертки вокруг изменяемых структур данных, таких как `java.util.Collections`. Такие обертки практически всегда делегируют выполнение операций изменяемым коллекциям, лежащим в их основе. То есть, если сама коллекция изменяется, «неизменяемая» обертка вокруг этой коллекции тоже изменяется.

Кипы документации и вредных советов², накопившиеся за долгие годы и описывающие, как адекватно управлять одновременным доступом и изменением совместно используемого состояния посредством блокировок. В результате взаимоблокировки и состояния гонки за ресурсами стали одним из основных источников проблем с качеством во всей индустрии.

Наконец, значительная часть проблем в программировании сопряжена с идентификацией и обеспечением *неизменности* повсюду, где только возможно – в алгоритмах, приложениях или бизнес-правилах, которые не должны изменяться. Чем большей степени неизменности удастся добиться, тем плотнее вы сможете сосредоточиться на локальных эффектах конкретного фрагмента программного кода, и тем большую долю риска вы сможете исключить из системы, созданием которой занимаетесь. Использование неизменяемых значений открывает совершенно новые перспективы – вы получаете абсолютную уверенность, что передача коллекции в функцию не повлечет за собой изменение этой коллекции, что многочисленные потоки выполнения смогут использовать значение без всякого риска нарушить его целостность, причем без потери эффективности за счет использования сложных стратегий блокирования доступа, и что зависящие от времени изменения не повлекут за собой зависимость поведения программного кода от времени³. Все это уже гарантиру-

¹ В главе 12 будут представлены примеры, демонстрирующие, как возможности языка Clojure делают ненужными многие объектно-ориентированные шаблоны проектирования.

² Возможно, вы помните неразбериху и неопределенность, вокруг *блокировок с двойной проверкой* (double-checked locking) несколько лет тому назад, которые в конечном итоге были устранены за счет некоторого усложнения и введения новой модели управления памятью в JVM: <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>.

³ Гейзенбаргов, <https://ru.wikipedia.org/wiki/Heisenbug>.

ется при работе с такими значениями, как числа, и нет причин не потребовать то же самое от структур данных, примером чему могут служить неизменяемые коллекции и записи в языке Clojure.

Функции, как сущности первого порядка, и функции высшего порядка

Несмотря на различия толкования термина «функциональное программирование» в разных языках, одно из требований остается неизменным: функции сами должны быть значениями, чтобы их можно было интерпретировать как обычные данные, принимать в виде аргументов и возвращать в виде результатов из других функций.

Возможность интерпретировать функции как данные позволяет создавать абстракции, отсутствующие в других языках. В качестве простого примера представьте, что требуется написать функцию, которая дважды вызывает некоторую другую функцию. При этом создаваемая функция `call_twice` должна быть достаточно универсальна, чтобы вызывать не какую-то конкретную, а *любую* функцию с *любым* аргументом.

Эта задача легко решается в языках, где функции являются сущностями первого порядка. В Ruby¹ и Python:

```
# Ruby
def call_twice(x, &f)
  f.call(x)
  f.call(x)
end

call_twice(123) {|x| puts x}

# Python
def call_twice(f, x):
    f(x)
    f(x)

call_twice(print, 123)
```

¹ Блоки в языке Ruby, объекты, созданные посредством `lambda` или `Proc.new`, и даже методы классов `SomeClass.method(:foo)` — все это разновидности функций, как сущностей первого порядка. С другой стороны, все эти роли в языке Clojure играют функции.

В языке Clojure реализация функции выглядит ничуть не сложнее:

```
(defn call-twice [f x]
  (f x)
  (f x))

(call-twice println 123)
; 123
; 123
```

Однако в Java написать даже такую простую функцию будет значительно сложнее. По иронии, основной язык виртуальной машины JVM не позволяет интерпретировать функции как сущности первого порядка. В языке Java программный код может существовать только в методах, связанных с классами, и на Java-методы нельзя сослаться, не прибегая к механизму рефлексии.

Классы, содержащие только статические вспомогательные методы, такие как `java.lang.Math`, можно с натяжкой считать неким подобием пространств имен, компенсирующим отсутствие поддержки функций, как сущностей первого порядка. Другие вспомогательные методы – подобно строковым операциям, реализованным в классе `java.lang.String` – не являются статическими и потому должны вызываться относительно конкретного экземпляра.

```
Math.max(a, b);

someString.toLowerCase();
```

Такая архитектура может показаться разумной – что может быть проще, чем группировать родственные методы в выделенных классах или привязывать операции со значениями определенного типа к экземплярам этого типа?

В действительности, не делать ни те ни другие значительно проще и во многих отношениях эффективнее.

Например, представьте, что вам потребовалось на языке Java реализовать поиск наибольшего числа в массиве или преобразовать список строк в аналогичный список, но содержащий только знаки нижнего регистра.