Функциональное программирование

Чаз Эмерик
Брайан Карпер
Кристоф Гранд

# Программирование на Clojure

## Практика применения Lisp в мире Java

ДМК Пресс
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**



ISBN 978-1-449-39470-7 (анг.)

ISBN 978-5-94074-925-7 (рус.)

# Содержание

# 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 времени[1]. 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 дл использования в повседневной работе[1].

## 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, сокеты 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.

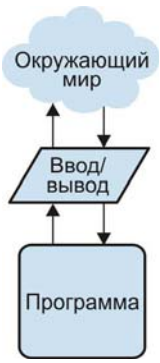


**Рис. 2.1.** Организация типичной программы

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

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 число[1]:

```
(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 неудачу[1].

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, что[1]:

- 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 давно[2] 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 состоянии[3].

---

[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 другие[1].

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 советов[2], 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 времени[3]. 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] Гейзенбагов, 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 Ruby[1] 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.