Лекція 9-10.

Програмування на Clojure

Что такое Clojure?

Clojure — Lisp'oобразный язык общего назначения, разработанный для Java Virtual Machine (JVM)[1]. Автором языка является Rich Hickey, который несколько лет разрабатывал язык в одиночку вплоть до выпуска первой публичной версии в 2007-м году. В настоящее время, стабильной версией является версия 1.2, выпущенная в августе 2010-го года, и наш рассказ будет вестись именно о ней.

В отличие от других реализаций Lisp'a и Scheme для виртуальной машины Java, таких как ABCL, Kawa и т.д., Clojure не совместим на 100 процентов ни с Common Lisp, ни с Scheme, но позаимствовал многие идеи из этих языков, добавив новые вещи, такие как неизменяемость данных, конкурентное выполнение кода и т.п. Более подробно о том, зачем был создан новый язык, можно прочитать на сайте проекта.

Несмотря на то, что Clojure — молодой язык программирования, достаточно много людей используют его в своих проектах, в том числе и коммерческих, например, FlightCaster, который использует Clojure при обработке большого количества данных, решая задачи Machine Learning в распределенной среде. Существуют и другие фирмы (например, Sonian, Runa, Emendio), использующие этот язык в своей работе — ссылки на них вы сможете найти на сайте языка.

Основные возможности языка

Clojure является функциональным языком программирования с поддержкой функций в качестве объектов первого класса (first class objects) и неизменяемыми (за исключением специальных случаев) данными, включая поддержку "ленивых" коллекций данных. От Lisp'a Clojure "унаследовал" макросы, мультиметоды и интерактивный стиль разработки, а JVM дает переносимость и доступ к большому набору библиотек, созданных для этой платформы.

Неизменность структур данных позволяет использовать их в разных потоках выполнения программы, что упрощает многопоточное программирование. Однако не все структуры являются неизменяемыми — в нужных случаях программист может явно использовать изменяемые структуры данных, используя Software Transactional Memory (STM), что обеспечивает надежную работу в многопоточной среде. (В качестве примера многопоточной программы, работающей с разделяемыми данными, можно привести программу "муравьи" (ants), которую достаточно сложно написать на Java из-за большого количества моделируемых сущностей, но которая достаточно просто выглядит на Clojure).

За счет того, что Clojure был спроектирован для работы на базе JVM, обеспечивается доступ к большому набору библиотек, существующих для данной платформы. Взаимодействие с Java реализуется в обе стороны — как вызов кода, написанного на Java, так и реализация классов, которые доступны как для вызова из Java, так и из других языков, существующих для JVM, например, Scala. Подробнее о взаимодействии с JVM написано далее.

Отличия от Lisp

Несмотря на схожесть синтаксиса, Clojure отличается и от Common Lisp, и от Scheme. Некоторые отличия обусловлены тем, что язык разработан для платформы JVM, что накладывает некоторые ограничения на реализацию. Например, JVM не поддерживает оптимизацию хвостовых вызовов (tail call optimization, TCO), поэтому в язык были введены явные операторы `loop` и `recur`. Также важными определяющими факторами JVM-платформы являются:

- boxed integers — нет поддержки полного набора типов чисел (numeric tower), которые есть в Scheme и Common Lisp;
- система исключений как в Java (в Common Lisp используется сигнальный протокол);
- используется соглашение о вызовах как в Java.

Полный список отличий можно найти на отдельной странице на сайте языка. Из явных отличий от Common Lisp можно отметить следующие:

- идентификаторы в Clojure регистрозависимы (case-sensitive);
- большая часть данных — неизменяемая;
- пользователь не может изменять синтаксис языка путем ввода собственных макросов в процедуре чтения кода (read macros);
- введен специальный синтаксис для литералов, векторов, отображений (maps), регулярных выражений, анонимных функций и т.д.;
- существует возможность связывания метаданных с переменными и функциями;
- можно реализовать функции с одним именем и разным набором аргументов;
- многие привычные вещи, такие как `let`, по синтаксису отличаются от их аналогов в Common Lisp и Scheme (при этом используется меньше скобок), например, `let` связывает данные последовательно, аналогично `let*` в Scheme;
- вместо функций `car` и `cdr` используются функции `first` и `rest`;
- `nil` не равен пустому списку или другому набору данных (коллекции) — он всего лишь означает отсутствующее значение (аналог `null` в Java);
- используется общее пространство имен, как в Scheme;

- сравнение на равенство производится одной функцией в отличие от Common Lisp и Scheme;
- поддержка "ленивых" коллекций.

Источники информации о языке

Основной источник информации по данному языку — сайт проекта и список рассылки. Помимо сайта проекта, хорошим источником информации является набор видеолекций на Blip.TV, а также видеолекции, в которых автор языка рассказывает о Clojure и об особенностях его использования. Кроме того, следует отметить набор скринкастов, созданных Sean Devlin, в которых он рассказывает о разных возможностях языка, включая новые, появившиеся в версии 1.1.

Из книг в настоящее время доступна книга Programming Clojure, выпущенная в серии Pragmatic Programmers, которая в принципе содержит всю необходимую информацию о языке, включая описание основных возможностей языка, вопросы взаимодействия с Java, основные функции, отличие языка от Common Lisp, и т.п. В мае 2010 года издательство Apress выпустило еще одну книгу по Clojure — Practical Clojure. The Definitive Guide, которая является кратким описанием современной версии языка, включая новшества, которые введены в версии 1.2. А на начало 2011 года в издательстве Manning запланирован выход книг Clojure in Action (введение в язык и примеры практического использования) и The Joy of Clojure. Thinking the Clojure Way (более "глубокое" описание языка, с разъяснением сложных понятий).

В свободном доступе можно найти книгу Clojure Programming, работа над которой ведется в рамках проекта WikiBooks. Также существует достаточно подробный практический учебник — Clojure Scripting. Кроме того, недавно был опубликован учебник Clojure Notes, который использовался в рамках курса обучения Clojure.

Хорошее описание того, как можно использовать макросы для построения абстракций, можно найти в известной книге On Lisp Пола Грэма (Paul Graham). Несмотря на то, что в ней используется Common Lisp, многие вещи будут применимы и для Clojure.[2]

Очень большое количество информации о языке, разрабатываемых библиотеках и проектах, использующих Clojure, публикуется в блогах. Для того, чтобы свести всю эту информацию воедино, существует проект Planet Clojure, на который вы можете подписаться, чтобы быть в курсе новостей о языке.

Установка и запуск

Установка Clojure достаточно проста — скачайте последнюю версию с сайта языка и распакуйте в нужный каталог. После этого вы можете запустить ее с помощью команды:

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

Эта команда приведет к запуску JVM и вы получите доступ к REPL ("read-eval-print loop" — цикл ввода выражений и выдачи результатов). Стандартный REPL имеет не очень хорошие возможности по редактированию кода, так что при работе с REPL лучше использовать библиотеку `jline`, как описано в разделе Getting Started официальной документации Clojure, или воспользоваться одной из сред разработки, описанных в разделе Среды разработки. Более подробные инструкции по развертыванию для разных сред разработки вы можете найти в описании проекта labrepl, целью которого является упрощение начала работы с Clojure. В составе данного проекта имеется набор учебных материалов, которые будут полезны начинающим работать с языком.

Работая в REPL вы можете получать информацию о функциях, макросах и других объектах языка. Для получения информации о каком-либо символе или специальной форме вы можете использовать макрос `doc`. Например, `(doc map)` напечатает справку по функции `map`, которая была задана при объявлении этой функции. А если вы не помните точное название символа, можно провести поиск по документации с помощью функции `find-doc`, которая принимает один аргумент — строку с регулярным выражением по которому будет проводиться поиск.

Из чего состоит язык Clojure

Синтаксис языка Clojure следует стандартному для Lisp'образных языков подходу "код как данные", когда данные и код имеют общий синтаксис. Как и в других диалектах Lisp'а, код записывается в виде списков, используя префиксную нотацию и представляя собой синтаксическое дерево. Однако по сравнению с другими языками, в Clojure введены дополнительные сущности: кроме стандартных для Lisp'а символов, базовых литералов (строки, числа и т.п.) и списков, в язык введен дополнительный синтаксис для векторов, отображений (maps) и множеств (sets), являющихся объектами первого класса (first class objects).

Кроме этого, процедура чтения кода (reader) распознает специфические для Clojure конструкции: `@` — для доступа к изменяемым данным и различные конструкции, начинающиеся с символа `#` — анонимные функции, метаданные (включая информацию о типах данных), регулярные выражения и т.д. Процедура чтения также рассматривает пробелы и запятые между элементами языка как один символ, разделяющий эти элементы.

Основные типы данных

Данные в Clojure можно разделить на две большие группы: базовые типы данных — числа, строки и т.д., и последовательности (коллекции), к которым относятся списки, векторы, отображения и множества. Пользователь может определять свои структуры данных с помощью `defstruct`, но они являются частным случаем отображений и введены для обеспечения более эффективной работы со сложными данными.

Все типы данных имеют общий набор характеристик: данные неизменяемы и реализуют операцию "равенство" (equality).

Базовые типы данных

К базовым типам данных Clojure относятся следующие:

логические значения
: в языке определено два объекта для представления логических значений: `true` — для истинного значения и `false` — для ложного. (Все остальные значения, кроме `false` и `nil`, рассматриваются как истинные);

числа

в языке могут использоваться числа разных типов. По умолчанию для представления целых чисел используются классы, унаследованные от `java.lang.Number` — `Integer`, `BigInteger`, `BigDecimal`, но в Clojure реализуется специальный подход, который позволяет представлять число наиболее эффективным способом, автоматически преобразуя числа в случае необходимости — например, при переполнении числа. Если вы хотите для целого числа явно указать тип `BigDecimal`, то вы можете добавить букву `M` после значения.

Для чисел с плавающей точкой используется стандартный класс `Double`.

Кроме этих видов чисел, в Clojure определен специальный тип `Ratio`, представляющий числа в виде рациональных дробей, что позволяет избегать ошибок округления — например, при делении.

строки

строки в Clojure являются экземплярами класса `java.lang.String` и к ним можно применять различные функции определенные в этом классе. Форма записи строк Clojure совпадает со стандартной записью строк в Java;

знаки (characters)

являются экземплярами класса `java.lang.Character` и записываются либо в форме `\N`, где `N` — соответствующая буква, либо как названия для неотображаемых букв — например, как `\tab` и `\space` для символа табуляции и пробела и т.д.;

символы (symbols)

используются для ссылки на что-то — параметры функций, имена классов, глобальные переменные и т.д. Для представления символа как отдельного объекта, а не как значения, для которого он используется в качестве имени, используется стандартная запись `'symbol` (или специальная форма `quote`);

keywords (ключевые символы)

это специальные символы, имеющие значение самих себя[3], аналогично символам (symbols) в Lisp и Ruby. Одним из важных их свойств является очень быстрая операция проверки на равенство, поскольку происходит проверка на равенство указателей. Это свойство делает их очень удобными для использования в качестве ключей в отображениях (maps) и тому подобных вещах. Для именованных аргументов существует специальная форма записи `:keyword`.

Стоит также отметить, что символы и keywords имеют некоторую общность — в рамках интерфейса `IFn` для них создается функция `invoke()` с одним аргументом, что позволяет использовать символы и keywords в качестве функции. Например, конструкция `(:mykey my-hash-map)` или `('mysym my-hash-map)` аналогичны вызову `(get my-hash-map :mykey)` или `(get my-hash-map 'mysym)`, который приведет к извлечению значения с нужным ключом из соответствующего отображения.

В языке Clojure имеется специальное значение `nil`, которое может использоваться как значение любого типа данных, и совпадающее с `null` в Java. `nil` может использоваться в условных конструкциях наравне со значением `false`. Однако стоит отметить, что, в отличие от Lisp, `nil` и пустой список — `()` не являются взаимозаменяемыми и использование пустого списка в условной конструкции будет рассматриваться как значение `true`;

Коллекции, последовательности и массивы

Кроме общих характеристик базовых типов перечисленных выше, все коллекции в Clojure имеют следующие характеристики:

- вся работа с коллекциями проводится через общий интерфейс;
- существует возможность связывания метаданных с коллекцией;
- для коллекций реализуются интерфейсы `java.lang.Iterable` и `java.util.Collection`, что позволяет работать с ними из Java;
- все коллекции рассматриваются как "последовательности" данных, вне зависимости от конкретного представления данных внутри них.

Неизменяемость коллекций означает, что результатом работы всех операций по модификации коллекций является другая, новая коллекция, в то время как исходная коллекция остается неизменной. В Clojure существует эффективный механизм, помогающий реализовывать неизменяемые коллекции. С его помощью операции, изменяющие коллекцию, могут эффективно создавать "измененную" версию данных, которая использует большую часть исходных данных, не создавая полной копии.

В текущей версии Clojure реализованы следующие основные виды коллекций:

списки (lists)
: записываются точно также как и в других реализациях Lisp. В Clojure списки напрямую реализуют интерфейс `ISeq`, что позволяет функциям работы с последовательностями эффективно работать с ними. (При использовании функции `conj` новые элементы списков добавляются в начало);

векторы (vectors)
: представляют собой последовательности, элементы которых индексируются целым числом (с последовательными значениями индекса в диапазоне `0..N`, где `N` — размер вектора). Для определения вектора необходимо заключить его элементы в квадратные скобки, например, `[1 2 3]`. Для преобразования других коллекций в вектор можно использовать функции `vector` или `vec`. Поскольку вектор индексируется целым числом, то операция доступа к произвольному элементу реализуется достаточно эффективно, что удобно при работе с некоторыми видами данных. (При использовании функции `conj` новые элементы векторов добавляются в конец.)

Кроме того, для вектора в Clojure создается функция одного аргумента (целого числа — индекса значения) с именем, совпадающим с именем символа, связанным с вектором. Это позволяет использовать имя вектора в качестве функции для доступа к нужному значению. Например, вызов `(v 3)` в данном коде:

```
user> (def v [1 2 3 4 5 "string"])
user> (v 3)
4
```

вернет значение четвертого элемента вектора.

отображения (maps)

это специальный вид последовательности, который отображает одни значения данных (ключ) в другие (значения). В Clojure существуют два вида отображений: `hash-map` и `sorted-map`, которые создаются с помощью соответствующих функций. `hash-map` обеспечивает более быстрый доступ к данным, а `sorted-map` хранит данные в отсортированном по ключу виде. Отображения записываются в виде набора значений (с четным количеством элементов), заключенных в фигурные скобки. Значения, стоящие на нечетных позициях рассматриваются как ключи, а на четных — как значения, связанные с данным ключом. В качестве ключа могут использоваться любые поддерживаемые Clojure типы данных, но очень часто в качестве ключей используют keywords, поскольку для них реализована очень быстрая проверка на равенство.

Также как и для векторов, для отображений создается функция одного аргумента (ключа), которая позволяет использовать имя символа, связанного с отображением, для доступа к элементам. Например,

```
user> (def m {:1 1 :abc 33 :2 "2" })
#'user/m
user> (m :abc)
33
```

множества (sets)

представляет собой набор уникальных значений. Также как и для отображений, существует два вида множеств — `hash-set` и `sorted-set`. Определение множества имеет следующий вид `#{elements...}`, а для создания множества из других коллекций может использоваться функция `set`, например, для получения множества уникальных значений вектора, можно использовать следующий код:

```
user> (set [1 2 3 2 1 2 3])
#{1 2 3}
```

В Clojure также определены дополнительные виды отображений, позволяющие в специальных случаях добиться большей производительности:

отображения-структуры (struct maps)
: могут использоваться для эмуляции записей (records), имеющихся в других языках программирования. В этом случае отображения имеют набор одинаковых ключей и Clojure реализует эффективное хранение информации о ключах, а также предоставляет быстрый доступ к элементам по ключу. В случае необходимости, имеется возможность генерации специализированной функции доступа с помощью функции `accessor`.

  Определение отображения-структуры производится с помощью макроса `defstruct` или функции `create-struct`. Новые экземпляры отображений создаются с помощью функции `struct-map` или `struct`, которые получают список элементов для заполнения данного отображения. При этом стоит отметить, что отображение-структура может иметь большее количество ключей, чем было определено в `defstruct` — в этом отношении, отображения-структуры ведут себя точно также, как и обычные отображения.

отображения-массивы (array maps)

это специальный вид отображений, в котором сохраняется порядок ключей. Такие отображения реализованы в виде обычного массива, содержащего ключи и значения. Поиск в отображении является линейной функцией от количества элементов, и поэтому, такие отображения должны использоваться только для хранения небольшого количества элементов. Новые отображения-массивы могут создаваться с помощью функции `array-map`.

Работа с коллекциями выполняется единообразно — для всех коллекций поддерживаются операции `count` для получения размера коллекции, `conj` для добавления элементов в коллекцию (реализуется по-разному, в зависимости от конкретного типа) и `seq` для представления коллекции в виде последовательности — это позволяет применять к ним функции работы с последовательностями: `cons`, `first`, `map` и т.д. Функцию `seq` также можно использовать для преобразования в последовательности и коллекций Java.

Большая часть функций для работы с последовательностями является "ленивой", обрабатывая данные по мере их надобности, что позволяет эффективно работать с данными большого размера, в том числе и с бесконечными последовательностями. Пользователь может создавать свои функции, которые возвращают "ленивые" последовательности, с помощью макроса `lazy-seq`. Также в версии 1.1 было введено понятие блоковых последовательностей (chunked sequence), которые позволяют создавать элементы блоками по N элементов, что в некоторых случаях позволяет улучшить производительность.

Из общего ряда выпадает работа с массивами Java, поскольку они не являются коллекциями в терминах Clojure. Для работы с массивами определен набор функций, которые позволяют определять массивы разных типов (`make-array`, `XXX-array`, где `XXX` — название типа), получения (`aget`) и установки (`aset`) значений в массиве, преобразования коллекций в массив (`into-array`) и т.д.

"Ленивые" структуры данных

Как уже упоминалось выше, большая часть структур данных в Clojure (и функций для работы с этими структурами данных) являются "ленивыми" (lazy). В языке имеется явная поддержка "ленивых" структур данных, позволяя программисту эффективно работать с ними. Одним из достоинств поддержки "ленивых" структур данных является то, что можно реализовывать очень большие или бесконечные последовательности используя конечное количество памяти. "Ленивые" последовательности и функции также могут использоваться для обхода ограничений, связанных с отсутствием оптимизации хвостовых вызовов в Clojure.

В Clojure программист может определить "ленивую" структуру данных воспользовавшись макросом `lazy-seq`. Данный макрос в качестве аргумента принимает набор выражений, которые возвращают структуру данных, реализующую интерфейс `ISeq`. Из этих выражений затем создается объект типа `Seqable`, который вызывается по мере необходимости, кешируя полученные результаты.

В качестве примера использования "ленивых" структур данных давайте рассмотрим создание бесконечной последовательности чисел Фибоначи:

```
(defn fibo
  ([] (concat [1 1] (fibo 1 1)))
  ([a b]
     (let [n (+ a b)]
       (lazy-seq (cons n (fibo b n))))))
```

В данном случае мы определяем функцию, которая при запуске без аргументов создает начальную последовательность из чисел 1 и 1 и затем вызывает сама себя, передавая эти числа в качестве аргументов. А функция двух аргументов обернута в вызов `lazy-seq`, который производит вычисление следующих чисел Фибоначи. При этом мы можем определить переменную, которая, например, будет содержать первые сто миллионов чисел Фибоначи:

```
user> (def many-fibs (take 100000000 (fibo)))
#'user/many-fibs
```

но поскольку мы работаем с "ленивыми" последовательностями, то значение будет создано мгновенно, без вычисления всех чисел Фибоначи. А само вычисление чисел будет происходить по мере надобности. Например, мы можем получить 55-е число Фибоначи с помощью следующего кода:

```
user> (nth many-fibs 55)
225851433717
```

Переходные структуры данных (transients)

При интенсивной работе с неизменяемыми коллекциями иногда возникает слишком много промежуточных объектов, что достаточно неэффективно. В версии 1.1 появилась возможность временно использовать изменяемые коллекции данных используя переходные (transient) структуры данных. Эта функциональность была специально введена в язык для оптимизации производительности.

Основная идея заключается в том, чтобы избежать ненужного копирования данных, что происходит когда вы работаете с неизменяемыми данными. Стоит отметить, что не все структуры данных поддерживают эту возможность — в версии 1.1.0 поддерживаются векторы, отображения и множества, а списки — нет, поскольку для них нет существенного выигрыша в производительности. В общем виде работа с переходными структурами данных происходит следующим образом:

- вы преобразуете стандартную структуру данных в переходную структуру (вектор, отображение и т.д.) с помощью функции `transient`, получающей один параметр — соответствующую структуру данных;
- выполняете изменение структуры по месту (inplace) с помощью специальных функций `assoc!`, `conj!` и т.п., которые аналогичны по действию соответствующим функциям, но без символа `!`, но применяются только к переходным структурам данных;
- после окончания обработки, превращаете переходную структуру данных в стандартную, неизменяемую структуру данных с помощью функции `persistent!`.

Рассмотрим простой пример использования стандартных и переходных структур данных[4]:

```
(defn vrange [n]
  (loop [i 0
         v []]
    (if (< i n)
      (recur (inc i) (conj v i))
      v)))

(defn vrange2 [n]
  (loop [i 0
         v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))))
```

Обе функции создают вектор заданного размера, состоящий из чисел в диапазоне `0..n`. В первой функции используются стандартные, неизменяемые структуры данных, а во второй — переходные структуры данных. Как видно из примера, во второй функции выполняются все требования к использованию переходных структур — сначала с помощью вызова `(transient [])` создается ссылка на вектор, который затем заполняется с помощью функции `conj!`, и в конце происходит возвращение неизменяемой структуры данных, созданной из переходной структуры с помощью вызова `(persistent! v)`. Если мы запустим обе функции с одинаковыми параметрами, то мы получим следующие результаты:

```
user> (time (def v (vrange 1000000)))
"Elapsed time: 439.037 msecs"
user> (time (def v2 (vrange2 1000000)))
"Elapsed time: 110.861 msecs"
```

Как видно из этого примера, использование переходных структур дает достаточно большой выигрыш в производительности — примерно в четыре раза. А на некоторых структурах данных выигрыш в производительности может быть больше. Копирование исходных данных и создание неизменяемой структуры — это операции со сложностью O(1), при этом происходит эффективное использование оригинальных данных. Также стоит отметить, что использование переходных структур данных приводит к принудительной изоляции потока выполнения — изменяемые данные становятся недоступными из других потоков выполнения.

Более подробно о переходных структурах данных вы можете прочитать на сайте языка.

Базовые конструкции языка

Синтаксис языка в общем виде (за исключением специальных форм для отображений, векторов и других элементов) совпадает с синтаксисом Common Lisp и Scheme — все описывается S-выражениями. Код записывается в виде списков, используя префиксную нотацию, когда первым элементом списка является функция, макрос или специальная форма[5], а остальные элементы — аргументы, которые будут переданы в это выражение. Кроме списков, S-выражения могут состоять из атомов: чисел, строк, логических констант и т.д.

Объявление и связывание символов

В Lisp'е существуют объекты типа символ, которые в отличие от других языков представляют собой не просто имена переменных, а отделенные сущности. То, что в других языках понимается как присваивание значения переменной (с определенным именем, которое по сути является просто меткой области памяти), в Lisp'е формулируется по другому — как связывание значения с символом, т.е. связывание двух объектов. Через эти же понятия формулируется и подход Lisp'а к типизации — значения имеют типы, а переменные — нет, поскольку переменная — это и есть пара символ-значение. Если вы используете имя символа без маскирования (специальная форма `quote`, или `'`), то вместо символа будет подставлено значение, связанное с этим символом. Если же вам нужно передать сам символ, то вы должны использовать запись `'имя-символа`.

Имеется два вида связывания символа со значением:

- *лексическое*, когда значение привязано к символу только внутри тела одной формы, например `let` или `fn`. Как раз тут и проявляется особенность Clojure — функциональные переменные: внутри тела `let` нельзя установить значение используя `set!`. Можно считать, что в этом случае `x` — это псевдоним для значения `2`.
- *динамическое*, когда значение привязывается к символу на время выполнения программы. Как правило, в этом случае символ делается доступным глобально (хотя в Common Lisp он может быть и локальным, но объявленным с помощью *declare special*). Суть слова *динамичность* в том, что в любой момент и в любом месте символ может быть заново связан с другим значением, и значение будет "видно" при всех обращениях к этому символу в любых формах. Такие динамические (в данном случае и глобальные в рамках пространства имен) символы определяются и связываются в Clojure формой `def`, на основе которой затем строятся макросы `defn`, `declare`, `defonce` и т.д. Таким образом, например, имя функции привязывается к самому объекту функции, что будет видно на примере раскрытия макроса `defn` ниже.

Таким образом, говорится о двух областях видимости: лексической и динамической. Нужно иметь в виду, что это не то же самое, что и локальные и глобальные переменные. Хотя динамически связанные символы, как правило, глобальны, а лексически связанные — локальны. А как быть в случае конфликта лексической и динамической привязки?

- В Common Lisp существует дуализм связывания, выраженный в специальной форме `let`, которая может связывать как лексические, так и специальные (динамические) символы. Преимущество этой формы в том, что при такой привязке динамический символ как бы приобретает свойство лексического: его новое значение видно только во время выполнения тела `let`, в то время как в других формах доступно его "глобальное" значение. Это очень мощная возможность, благодаря которой использование глобальных переменных в Lisp не приводит к тем разрушительным последствиям, которые присутствуют в других языках;

- В Clojure реализован тот же принцип за тем исключением, что разделены формы, выполняющие обычное лексическое связывание (для лексических символов, которые теперь смело можно назвать локальными) — `let`, и лексическое связывание для динамических символов — это форма `binding`. Подробней об этом можно прочитать в обсуждении "Let vs. Binding in Clojure" на StackOverflow.com.

Общая форма объявления "динамического" символа выглядит как `(def имя значение?)`, при этом значение может не указываться и тогда символ будет несвязанным со значением. Существуют разные макросы для объявления символов, которые выполняют определенные задачи, например, `defonce` позволяет объявить символ только один раз, и если он уже имеет значение, то новое значение будет проигнорировано. А с помощью макроса `declare` можно выполнить опережающее определение (forward declaration) символа — это часто удобно при объявлении взаимно рекурсивных функций. При объявлении символа можно указать метаданные, связанные с данным символом (см. раздел Метаданные).

Для объявления "лексических" символов используется форма `let`, которая выглядит следующим образом: `(let [имя1 знач1 имя2 знач2 ....] код)`. Например, выполнение следующего кода:

```
(let [x 1
      y 2]
   (+ x y))
```

выдаст значение `3`. В этом случае переменные `x` и `y` видны только внутри формы `(+ x y)`, и возможно маскируют значения переменных, объявленных на глобальном уровне, или выше по коду.

В некоторых случаях, программист может переопределить значение "динамического" символа для конкретного потока выполнения. Это делается с помощью макроса `binding`, который переопределяет значение указанного символа для данного потока выполнения и всех вызываемых из него функций. Например:

```
user> (def x 10)
user> (def y 20)
user> (defn test-fn []
            (+ x y))
user> (test-fn)
30
user> (binding [x 11
                y 22]
         (test-fn))
33
user> (let [x 11
            y 22]
         (test-fn))
30
```

В данном коде, если мы выполним `test-fn` на верхнем уровне кода, то получим значение `30`, равное сумме значений переменных `x` и `y`. А если эта функция будет вызвана из `binding`, то мы получим значение `33`, равное сумме переменных объявленных в `binding`. Данные значения изменяются только для текущего потока выполнения, и только для кода, который будет вызван из `binding`. После завершения выполнения кода в этой форме все предыдущие значения восстанавливаются. А при использовании `let` значения `x` и `y` не воздействуют на функцию `test-fn`, и в ней используются "глобальные" значения, давая в результате `30`. *Стоит быть осторожным при использовании функций работы с последовательностями внутри `binding`, поскольку они возвращают "ленивые" последовательности, данные в которых будут вычисляться уже вне `binding`.*

Деструктуризация параметров

Если вы передаете сложные структуры данных в функции (`defn` или `fn`), или связываете значения с символами в `let`, то вы можете использовать встроенную в Clojure деструктуризацию параметров, что позволяет связывать части структур данных с символами. Это значительно упрощает (и сокращает) код, поскольку вам не нужно писать явный код для получения значения значений из массивов или отображений.

Существует две формы деструктуризации параметров — деструктуризация отображений, и деструктуризация векторов, строк и массивов.

Деструктуризация векторов, строк и массивов

Деструктуризация векторов/массивов/строк имеет простую форму — вы помещаете список символов внутри вектора, и значения, стоящие на соответствующих местах в деструктурируемом объекте, будут связаны с этими символами. В том случае, если вы укажете меньше аргументов, чем указано символов, то отсутствующие значения будут рассматриваться как имеющие значение `nil`, если укажете больше, то они будут отброшены. Вы можете собрать "лишние" значения в отдельный список, используя запись `& символ`, и все "лишние" значения будут помещены в список, связанный с указанным символом.

Вот примеры использования деструктуризации векторов и строк:

```
user> (let [ [a b] [1 2]]
  (list a b))
(1 2)
user> (let [ [a b] [1]]
  (list a b))
(1 nil)
user> (let [ [a b] [1 2 3 4]]
  (list a b))
(1 2)
user> (let [ [a b & c] [1 2 3 4]]
  (list a b c))
(1 2 (3 4))
user> (let [ [a b & c] "abcde"]
  (list a b c))
(\a \b (\c \d \e))
user> ((fn [ [a b] ] (list a b)) [1 2])
(1 2)
user> (defn dfunc1 [ [a b] ]
  (list a b))
user> (dfunc1 [1 2])
(1 2)
```

Деструктуризация отображений

Деструктуризация отображений выглядит следующим образом — вы записываете отображение, в котором ключом является символ, с которым будет связано значение, а значением — является ключ в деструктурируемом отображении. Например,

```
user> (let [{a :k1 b :k2 c :k3} {:k1 1 :k2 2 :k3 3}]
  (list a b c))
(1 2 3)
user> (let [{a :k1 b :k2 c :k3} {:k1 1 :k3 3}]
  (list a b c))
(1 nil 3)
```

Также как и в случае с деструктуризацией векторов, если конкретный ключ не был указан, то символ получает значение `nil`, как это видно во втором примере.

Деструктуризация отображений может иметь более сложную форму, поскольку можно указывать еще и значения по умолчанию, которые будут присвоены соответствующим символам вместо `nil`, если нужный ключ не был указан в отображении. Это осуществляется путем указания ключевого символа `:or` и отображения, в котором перечисляются значения по умолчанию:

```
user> (let [{a :a b :b c :c :or {a 1 b 2 c 3}} {:a 4}]
  (list a b c))
(4 2 3)
```

Для того, чтобы не писать пары символ/ключ, имеется упрощенная форма записи, когда вы указываете ключевой символ `:keys` и вектор, содержащий список символов, которые будут превращены в ключевые слова с теми же самыми именами, и которые будут использоваться для поиска ключей в деструктурируемом отображении. Например, предыдущие примеры можно переписать в более компактной форме:

```
user> (let [{:keys [a b c]} {:a 1 :b 2 :c 3}]
  (list a b c))
(1 2 3)
user> (let [{:keys [a b c]} {:a 1 :c 3}]
  (list a b c))
(1 nil 3)
```

Кроме ключевых символов, в качестве ключей отображений вы можете использовать строки и символы. Для этого нужно использовать ключевое слово `:strs` — для строк и `:syms` — для символов, как это показано в следующих примерах:

```
user> (let [{:strs [a b c] :as m :or {a 2 b 3}} {"a" 5 "c" 6}]
  (list a b c m))
(5 3 6 {"a" 5, "c" 6})

user> (let [{:syms [a b c] :as m :or {a 2 b 3}} {'a 5 'b 6}]
  (list a b c m))
(5 3 6 {a 5, c 6})
```

Особенно полезна деструктуризация отображений для тех случаев, когда вы хотите иметь необязательные (и именованные) параметры в функциях. В этом случае, объявление функции будет иметь вид

```
(defn имя-функции [обязательные параметры
         & {:keys [необязательные параметры]
            :or {значения по умолчанию}}]
тело функции)
```

например,

```
user> (defn dfunc2 [a b & {:keys [c d] :or {c "c" d "d"}}]
  (list a b c d))
user> (dfunc2 1 2 :c 3)
(1 2 3 "d")
```

Здесь мы определяем функцию, принимающую два обязательных параметра, и два необязательных, значения которых можно указывать после соответствующих ключевых символов.

Получение оригинальных значений

Деструктуризация — это полезный инструмент, но иногда необходимо получить все переданные параметры в неизменном виде. Для этого можно использовать ключевой символ `:as символ`, которая связывает указанный символ с оригинальными параметрами. Например:

```
user> (let [[a b :as c] [1 2 3]]
  (list a b c))
(1 2 [1 2 3])
user> (let [{a :a b :b :as c} {:a 1 :b 2 :c 3}]
  (list a b c))
(1 2 {:a 1, :b 2, :c 3})
```

Деструктуризация вложенных структур

Также хочется отметить, что деструктуризация может применяться и к вложенным структурам, например, если у вас есть вектор векторов, или вложенные отображения. Например,

```
user> (let [ [a [b c]] [1 [2 3]]]
  (list a b c))
(1 2 3)
user> (let [ [a [b c]] [1 2]]
  (list a b c))
Error....
user> (let [{a :a {c :c d :d} :b} {:a 1 :b {:c 3 :d 4}}]
  (list a c d))
(1 3 4)
```

Но тут стоит быть осторожным, поскольку если вместо вектора вы передадите объект несовместимого типа, то вы получите ошибку, как во втором примере.

Дополнительные примеры вы можете найти на данной странице. Также хорошее описание деструктуризации параметров можно найти в книге "Clojure in Action".

Управляющие конструкции

Для управления потоком выполнения программы в Clojure имеется некоторое количество специальных форм, на основе которых затем строятся остальные примитивы языка, управляющие выполнением программы.

Для организации последовательного выполнения нескольких выражений существует специальная форма `do`, которой передаются выражения, которые вычисляются последовательно, и результат вычисления последнего выражения возвращается как результат `do`. Использование `do` похоже на использование `let` без объявления переменных. `do` часто используется в ветках условных выражений, когда необходимо выполнить несколько выражений вместо одного.

Условные операторы

Для обработки простых условий используется конструкция `if`, которая является специальной формой. На основе `if` затем строятся остальные конструкции — `when`, `when-not`, `cond` и т.д. `if` выглядит стандартно для Lisp'образных языков: `(if условие t-выражение f-выражение?)`, т.е. если `условие` вычисляется в истинное значение, то выполняется выражение `t-выражение`, иначе — выражение `f-выражение` (оно может не указываться, что используется в макросе `when`). Результаты вычисления одного из этих выражений возвращаются в качестве результата `if`.

Макрос `cond` позволяет проверить сразу несколько условий. По своему синтаксису он отличается от `cond` в Common Lisp и Scheme, и в общем виде записывается как `(cond условие1 выражение1 условие2 выражение2 ... :else выражение-по-умолчанию)`. Заметьте, что дополнительных скобок вокруг пары `условиеN выражениеN` не требуется.

Например:

```
(defn f1 [n]
  (cond
    (number? n) "number"
    (string? n) "string"
    :else "unknown"))
```

выражение `:else` не является ключевым словом языка и введено исключительно для удобства использования — вместо него можно использовать значение `true`.

Циклы и рекурсивные функции

Для организации циклов Clojure имеет несколько специальных форм, функций и макросов. Поскольку JVM имеет некоторые ограничения, не позволяющие реализовать оптимизацию хвостовых вызовов (Tail Call Optimization, TCO), то это накладывает ограничения на способ реализации некоторых алгоритмов, которые обычно реализуются через TCO в Scheme и других языках, поддерживающих эту оптимизацию.

Явные циклы организуются с помощью специальных форм `loop` и `recur`. Объявление `loop` похоже на `let`, но при этом имеется возможность повторного выполнения выражений, путем вызова `recur`[6] с тем же числом аргументов, которые были объявлены в списке переменных `loop` — обычно это новые значения цикла.

Вот простой пример — реализация функции вычисления факториала с помощью `loop/recur` (здесь нет проверки на отрицательный аргумент):

```
(defn factorial[n]
  (loop [cnt n
         acc 1]
    (if (zero? cnt)
      acc
      (recur (dec cnt) (* acc cnt)))))
```

В данном случае объявляется цикл с двумя переменными `cnt` и `acc`, которые получают начальные значения `n` и `1`. Цикл прекращается, когда `cnt` будет равен нулю — в этом случае возвращается накопленное значение, хранящееся в переменной `acc`. Если `cnt` больше нуля, то цикл начинается снова, уменьшая значение `cnt`, и увеличивая значение `acc`.

В большинстве случаев явный цикл по элементам последовательности можно заменить на вызов функций `reduce`, `map`, `filter` или макроса раскрытия списков (list comprehension) `for`. Функция `reduce` реализует операцию "свертка" (fold), и используется при реализации многих функций, таких как функции `+`, `-`, `import` и т.д. Для примера с факториалом, код становится значительно проще, чем в предыдущем примере:

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

Существует еще один метод оптимизации потребления стека при использовании взаимно рекурсивных функций — функция `trampoline`. Она получает в качестве аргумента функцию и аргументы для нее, и если переданная функция возвращает в качестве результата функцию, то возвращенная функция вызывается уже без аргументов. Вот пример определения четности числа, написанная для использования с `trampoline`:

```
(declare t-odd? t-even?)
(defn t-odd? [n]
  (if (= n 0)
    false
    #(t-even? (dec n))))
(defn t-even? [n]
  (if (= n 0)
    true
    #(t-odd? (dec n))))
```

Единственными отличиями от "стандартных" версий является то, что функции возвращают анонимные функции (строки 5 и 9). Если мы вызовем одну из этих функций с большим аргументом, например, вот так: `(trampoline t-even? 1000000)`, то вычисление произойдет без ошибки переполнения стека, в отличие от версии, которая не использует `trampoline`.

Стоит также отметить, что достаточно часто рекурсивные функции можно преобразовать в функции, производящие "ленивые" последовательности, как это было показано в разделе "Ленивые" структуры данных. Это положительно сказывается на производительности кода и потреблении памяти.

Исключения

Clojure поддерживает работу с исключениями (exceptions), которые часто используются в коде на Java. Специальная форма `throw` в качестве аргумента получает выражение, результат вычисления которого будет использован в качестве объекта-исключения[7].

Для выполнения выражений и перехвата исключений, которые могут возникнуть во время выполнения кода, используется специальная форма `try`. Форма `try` записывается следующим образом: `(try выражения* (catch имя-класса аргумент выражения*)* (finally выражения*)?)`. Блоки `catch` позволяют обрабатывать разные исключения в одном выражении `try`. А форма `finally` может использоваться для выражений, которые должны выполняться, и для случаев нормального завершения кода, и если произошел выброс исключения — например, для закрытия файлов и подобных этому задач.

Если мы введем следующий код:

```
(try
  (/ 1 0)
  (println "not executed")
  (catch ArithmeticException ex
    (println (str "exception caught... " ex)))
  (finally (println "finally is called")))
```

то на экран будет выведено следующее:

```
exception caught... java.lang.ArithmeticException: Divide by zero
finally is called
```

т.е. мы перехватили исключение и вывели его на экран, а затем выполнили выражение, указанное в блоке `finally`. При этом выражения, стоящие после строки приводящей к ошибке — `(println "not executed")`, не выполняются.

Функции

Функции в общем случае создаются с помощью макроса `fn`. Для объявления функций на верхнем ("глобальном") уровне пространства имен используются макросы `defn` или `defn-`, которые раскрываются в запись вида `(def имя-функции (fn ...))`. Второй макрос отличается от первого только тем, что функция будет видна только в текущем пространстве имен. Например, следующие объявления являются одинаковыми:

```
(def  func1 (fn [x y] (+ x y)))
(defn func2 [x y] (+ x y))
```

В общем виде объявление функции с помощью `fn` выглядит как `(fn имя? [аргументы*] условия? выражения+)`. Функция также может иметь разные наборы аргументов (разное число параметров) — тогда объявление будет выглядеть как `(fn имя? ([аргументы*] условия? выражения+)+)`.

Например, объявление функции:

```
(defn func3
  ([x] "one argument")
  ([x y] "two arguments")
  ([x y z] "three arguments"))
```

позволяет вызывать ее с одним, двумя или тремя аргументами[8]. Программист также может определить функцию, имеющую переменное число параметров, если укажет знак амперсанд (`&`) перед аргументом, в который будут помещены оставшиеся параметры.

Например, следующий код:

```
user> (defn func4 [x y & z] z)
user> (func4 1 2)
nil
user> (func4 1 2 3 4)
(3 4)
```

объявит функцию `func4`, имеющую два обязательных параметра — `x` и `y`, а остальные параметры будут помещены в список, который будет передан как аргумент `z`. См. раздел Деструктуризация параметров.

Для объявления небольших анонимных функций используется специальная запись `#(выражения+)`, а доступ к аргументам производится с помощью специальных переменных `%1`, `%2` и т.д., или просто `%`, если функция принимает один параметр. Например, следующие выражения эквивалентны:

```
(#(+ %1 %2) 10 20)
((fn [x y] (+ x y)) 10 20)
```

оба этих выражения возвращают одно и то же число. Специальная запись удобна, когда вам надо передать функцию в качестве аргумента, например, для функции `map` или `filter`.

Начиная с версии 1.1, при объявлении функции можно указывать пред- и постусловия, которые будут применяться к аргументам и результату. Эта функциональность реализует концепцию "контрактного программирования". Пред- и постусловия задаются как метаданные `:pre` и `:post`, которые указываются после списка аргументов. Каждое из условий состоит из вектора анонимных функций, которые должны вернуть `false` в случае ошибочных данных.

Например, рассмотрим следующую функцию:

```
(defn constrained-sqr [x]
  {:pre  [(pos? x)]
   :post [(> % 16), (< % 225)]}
  (* x x))
```

Данная функция принимает в качестве аргументов только положительные числа — условие `(pos? x)`, в диапазоне `5..14` — условие `(> % 16), (< % 225)`, иначе будет выдана ошибка проверки аргументов или результата.

В Clojure имеется набор функций, которые позволяют создавать новые функции на основе существующих. Функция `partial` используется для создания функций с меньшим количеством аргументов путем подстановки части параметров (каррирование), а функция `comp` создает новую функцию из нескольких функций (композиция функций):

```
user> (defn sum [x y] (+ x y))
user> (def add-5 (partial sum 5))
user> (add-5 10)
15
user> (def my-second (comp first rest))
user> (my-second [1 2 3 4 5])
2
```

В первом примере мы создаем функцию, которая будет прибавлять число `5` к переданному ей аргументу. А во втором примере, мы создаем функцию, эмулирующую функцию `second`, которая сначала применяет функцию `rest` к переданным ей аргументам, а затем применяет к результатам функцию `first`. Хороший пример использования функций `comp` и `partial` можно увидеть в скринкасте Point Free Clojure.

Макросы

Макросы — это мощное средство уменьшения сложности кода, позволяющие строить проблемно-ориентированную среду на основе базового языка. Макросы активно используются в Clojure, и множество конструкций, составляющих язык, определены как макросы на основе ограниченного количества специальных форм и функций, реализованных в ядре языка.

Макросы в Clojure смоделированы по образцу макросов в Common Lisp, и являются функциями, которые выполняются во время компиляции кода. В результате выполнения этих функций должен получиться код, который будет подставлен на место вызова макроса. Основная разница заключается в синтаксисе. В общем виде определение макроса выглядит следующим образом:

```
(defmacro name doc-string? attr-map? ([params*] body)+)
```

описание макроса (документация) — `doc-string?` и список атрибутов — `attr-map?` являются не обязательными, а список параметров и тело макроса могут указываться несколько раз, что позволяет определять макросы с переменным числом аргументов также, как и при объявлении функций (см. пример ниже).

Тело макроса должно представлять собой список выражений языка, результат выполнения которых будет подставлен на место использования макроса в виде списка Clojure, содержащего набор операций. Этот список может быть сформирован с помощью операций над списками — этот подход используется в простых случаях, так что вы можете сформировать тело макроса, используя операции `list`, `cons` и т.д. Хорошим примером этого подхода является макрос `when`, показанный ниже.

Другим подходом является маскирование (quote) всего выражения, с раскрытием только нужных частей кода. Для этого используется специальный синтаксис записи `` ` `` (обратная кавычка), внутри которого можно использовать `~` (тильда) для подстановки значений (аналогично операции `,` (запятая) в Common Lisp). Для подстановки списка не целиком, а поэлементно, используется синтаксис: `~@`. Хорошим примером второго подхода является макрос `and`, приведенный далее.

При работе с макросами очень полезными являются функции `macroexpand-1` и `macroexpand`, которые производят раскрытие заданного макроса, что позволяет программисту проверять корректность кода, используемого в макросах. Отличие между этими функциями заключается в том, что первая функция раскрывает макрос один раз, выполняя подстановки, но возможно возвращая код, который использует другие макросы. В то время как функция `macroexpand` — раскрывает макрос рекурсивно, раскрывая все использованные макросы.

Рассмотрим подстановки имен и значений более детально. Допустим, у нас есть две переменные — `x` со значением `2`, и `y`, представляющая собой список из трех элементов — `(4 5 6)`. Если мы попытаемся раскрыть разные выражения, то мы будем получать разные результаты:

```
user> (def x 2)
user> (def y '(4 5 6))
user> `(list 1 x 3)
(list 1 user/x 3)
user> `(list 1 ~x 3)
(list 1 2 3)
user> `(list 1 ~y 3)
(list 1 (4 5 6) 3)
user> `(list 1 ~@y 3)
(list 1 4 5 6 3)
```

В первом случае мы не выполняем никакой подстановки, поэтому `x` подставляется как символ. Во втором случае мы раскрываем выражение, подставляя значение символа и получая выражение `(list 1 2 3)`. В третьем случае у нас подставляется значение символа `y` в виде списка, в отличие от четвертого выражения, когда значение списка поэлементно подставляется (spliced) в выражение в раскрытом виде.

Примеры макросов

В качестве простого примера рассмотрим макрос `when`, определенный в базовой библиотеке:

```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an implicit do."
  [test & body]
  (list 'if test (cons 'do body)))
```

Данный макрос принимает один обязательный аргумент — условие `test`, а остальные аргументы рассматриваются как набор выражений, которые будут выполнены, если условие вернет истинное значение. Для того, чтобы можно было указать несколько выражений в качестве тела макроса, они обертываются в конструкцию `do`.

Если мы воспользуемся `macroexpand` для раскрытия макроса, то для конструкции вида:

```
(when (pos? a)
   (println "positive") (/ b a))
```

мы получим следующий код:

```
(if (pos? a)
  (do
    (println "positive")
    (/ b a)))
```

`when` — это достаточно простой макрос. Более сложные макросы могут создавать переменные, иметь разное количество аргументов, и т.д.

Например, макрос `and`, определенный следующим образом:

```
(defmacro and
  "Evaluates exprs one at a time, from left to right.
  If a form returns logical false (nil or false), and
  returns that value and doesn't evaluate any of the
  other expressions, otherwise it returns the value
  of the last expr. (and) returns true."
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

имеет три разных раскрытия для разного количества аргументов, при этом, если макросу передано больше одного аргумента, то он рекурсивно вызывает сам себя.

Мы можем увидеть это, раскрыв макрос для разных наборов аргументов:

```
user> (macroexpand '(and ))
true
user> (macroexpand '(and (= 1 2)))
(= 1 2)
user> (macroexpand '(and (= 1 2) (= 3 3)))
(let* [and__4457__auto__ (= 1 2)] (if and__4457__auto__ (clojure.core/and (=
3 3)) and__4457__auto__))
```

В этом примере вызов макроса без параметров приводит к подстановке значения `true`. При использовании одного параметра-выражения подставляется само выражение. А если параметров больше одного, то формируется форма `let`, в которой вычисляется первое выражение и связывается с переменной с уникальным именем (сгенерированным автоматически), а затем проверяется значение этой переменной. Если это значение истинное, то макрос вызывается еще раз, получая в качестве аргументов список параметров без первого элемента. А в том случае, если выражение не истинное, возвращается результат вычисления.

В макросе `and` для избежания конфликтов с кодом пользователя используется генерация уникальных имен переменных. Для этого используется специальный синтаксис `prefix#`, который создает уникальное имя, начинающееся с заданного префикса (в нашем случае имя начинается с `and`).

Шаблоны

В версии 1.1 была введена поддержка шаблонов (templates), которые могут использоваться с следующих случаях:

- когда нужны простые текстовые подстановки;
- когда подстановки нужны только в конкретном пространстве имен;
- когда нет возможности использовать функции высшего порядка для выполнения данной задачи.

Рассмотрим как это работает. Пространство имен `clojure.template` определяет два макроса: `apply-template` — предназначен для использования в других макросах, и `do-template` — для использования в обычном коде. Оба этих макроса имеют общую форму: `(do-template argv expr & values)`, где первым параметром указывается вектор параметров `argv`, которые будут подставляться в выражение `expr`, а в заключение идет список значений `values`, которые будут подставлены в выражение вместо соответствующих параметров. Необходимо помнить, что длина `values` должна быть кратной длине вектора `argv`, иначе остающиеся значения будут просто проигнорированы.

Рассмотрим пример с генерацией кода для тестов, взятый из скринкаста про работу с шаблонами.

```
(do-template [input result]
             (is (= (first input) result))
             [:a :b :c] :a
             "abc" \a
             (quote (:a :b :c)) :a)
```

Это выражение будет раскрыто в следующий код (вы можете проверить это с помощью `macroexpand-1`):

```
(do
  (is (= (first [:a :b :c]) :a))
  (is (= (first "abc") \a))
  (is (= (first '(:a :b :c)) :a)))
```

В данном случае у нас имеются следующие объекты: `argv` имеет значение `[input result]`, `expr` равен `(is (= (first input) result)`, а `values` — `[:a :b :c] :a .....`

При раскрытии происходит следующее: берутся первые `N` значений из списка `values` (`N` — длина `args`), и подставляются на места соответствующих параметров в выражении `expr`, затем берутся следующие `N` значений, и т.д., до тех пор, пока список значений не будет исчерпан.

Мультиметоды

Так же как и Common Lisp, Clojure поддерживает использование мультиметодов, которые позволяют организовать диспетчеризацию вызовов функций в зависимости от аргументов. Синтаксис мультиметодов немного отличается, и вместо `defgeneric` используется макрос `defmulti`, а в остальном принцип работы схож с CLOS.

Объявление функции, которая будет вести себя по разному в зависимости от аргументов, производится с помощью макроса `defmulti`. Данный макрос получает в качестве аргументов имя объявляемой функции, функцию диспетчеризации (которая должна вернуть значение, служащее ключом диспетчеризации) и список опций, определяющих способ диспетчеризации.

После объявления функции, пользователь может добавлять реализации с помощью макроса `defmethod`, который получает в качестве аргументов имя функции, значение по которому будет производиться диспетчеризация (часто это имя класса), список аргументов и тело функции. Например, если мы объявим следующую функцию, которая выполняет диспетчеризацию по типу переданного значения (с помощью функции `class`):

```
(defmulti foo class)
(defmethod foo java.util.Collection [c] :a-collection)
(defmethod foo String [s] :a-string)
(defmethod foo Object [u] :a-unknown)
```

и попробуем применить ее к разным аргументам, то мы получим следующие результаты:

```
user> (foo [])
:a-collection
user> (foo #{:a 1})
:a-collection
user> (foo "str")
:a-string
user> (foo 1)
:a-unknown
```

Но этот пример является достаточно простым и похож на стандартную диспетчеризацию в ОО-языках. Clojure предоставляет возможность диспетчеризации вызова в зависимости от значения аргументов, а также других признаков. Например, мы можем определить мультиметод с собственной функцией диспетчеризации, которая вызывает разные функции в зависимости от переданного значения:

```
(defn my-bar-fn [n]
  (cond
   (not (number? n)) :not-number
   (= n 2) :number-2
   (>= n 5) :number-5-ge
   :else :number-5-lt))
(defmulti bar my-bar-fn)
(defmethod bar :not-number [n] "not a number")
(defmethod bar :number-2 [n] "number is 2")
(defmethod bar :number-5-ge [n] "number is 5 or greater")
(defmethod bar :number-5-lt [n] "number is less than 5")
```

И, вызывая этот мультиметод, мы получим соответствующие значения:

```
user> (bar 2)
"number is 2"
user> (bar 5)
"number is 5 or greater"
user> (bar -1)
"number is less than 5"
user> (bar "string")
"not a number"
```

В Clojure имеется набор функций для работы с иерархиями классов: получения информации об отношениях между классами — `parents`, `ancestors`, `descendants`; проверки принадлежности одного класса к иерархии классов — `isa?` и т.д. Программист также может создавать свои иерархии классов, используя функцию `make-hierarchy`, и определять отношения между классами с помощью функции `derive`. Например, следующий код:

```
(derive java.util.Map ::collection)
(derive java.util.Collection ::collection)
```

устанавливает `::collection` в качестве родителя классов `java.util.Map` и `java.util.Collection`, что позволяет изменять существующие иерархии классов[9].

В том случае, если имеется перекрытие аргументов, и Clojure не может выбрать соответствующую функцию, то программист может выбрать наиболее подходящий метод с помощью функции `prefer-method`. Другие примеры и дополнительную информацию о мультиметодах и иерархиях классов вы можете найти на сайте.

Протоколы и типы данных

Одно из самых больших изменений в Clojure версии 1.2 — введение в язык новых артефактов: протоколов (protocols) и типов данных (datatypes). Данные изменения позволяют улучшить производительность программ по сравнению с мультиметодами, что в будущем даст возможность написать Clojure на Clojure (в данный момент протоколы и типы данных уже активно используются при реализации Clojure).

Что это такое и зачем нужно?

Протоколы и типы данных — два связанных друг с другом понятия. Протоколы используются для определения полиморфных функций, которые затем могут быть реализованы для конкретных типов данных (в том числе и из других библиотек).

Существует несколько причин введения протоколов и типов данных в новую версию языка:

- Увеличить скорость работы полиморфных функций, при этом поддерживая большую часть функциональности мультиметодов, поскольку для протоколов диспатчеризация выполняется только по типу данных;
- Использовать лучшие стороны интерфейсов (только спецификация функций, без реализации, реализация нескольких интерфейсов одним типом), в тоже время избегая недостатков (список реализуемых интерфейсов задан во время реализации типа данных, создание иерархии типов вида `isa/instanceof`);
- Избежать Expression problem и дать возможность расширять набор операций над типами данных без изменениях определения типов данных (в том числе и чужих) и перекомпиляции исходного кода[10];
- Использовать высокоуровневые абстракции для типов данных и операций над ними[11], что упрощает проектирование программ.

Также как и интерфейсы, протоколы позволяют объединить объявление нескольких полиморфных функций (или одной функции) в один объект[12]. Отличием от интерфейсов является то, что вы не можете унаследовать новый протокол от существующего протокола.

В отличии от имеющегося в Clojure `gen-interface` (и соответствующих `proxy/gen-class`) определение протоколов и типов не требует AOT (ahead-of-time) компиляции исходного кода, что упрощает распространение программ на Clojure. Однако при определении протокола, Clojure автоматически создает соответствующий интерфейс, который будет доступен для кода, написанного на Java.

Типы данных, определенные с помощью `deftype` или `defrecord` позволяют программисту на Clojure определять свои структуры данных, вместо использования обычных отображений и структур, но об этом ниже.

Важно помнить, что протоколы и типы данных с одним и тем же именем могут быть определены в разных пространствах имен, так что стоит быть осторожным и не наделать ошибок при импорте определений и последующей реализации протоколов!

Определение протоколов

Протоколом называется именованный набор функций с определенными сигнатурами. Для определения используется макрос, применение которого выглядит следующим образом:

```
(defprotocol название "описание" & сигнатуры)
```

`название` — единственный обязательный параметр, хотя определение протокола без функций не имеет особого смысла. В описании вы можете описать ваш протокол, и это описание будет показываться при вызове функции `doc` для вашего протокола. Для протокола вы можете указать одну или несколько сигнатур функций, где каждая сигнатура выглядит следующим образом:

```
(имя [аргументы+]+ "описание")
```

Вы можете определять одну функцию, которая будет принимать различное количество параметров, но первым аргументом функции всегда является объект, на основании которого будет выполняться диспатчеризация, и к которому эта функция будет применяться. Вы можете рассматривать его как `this` в Java и C++. В дополнение к сигнатурам, вы можете описать вашу функцию, но это необязательно.

Давайте посмотрим на стандартный пример:

```
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] [a b] [a b c] "baz docs"))
```

Данный протокол определяет две функции: `bar` — с двумя параметрами, и `baz` — с одним, двумя или тремя параметрами.

`defprotocol` также создаст соответствующий интерфейс, с тем же самым именем что и протокол. Данный интерфейс будет иметь те же самые функции, что и протокол.

Реализация протоколов

Протокол сам по себе ни на что не влияет — чтобы использовать его, мы должны добавить его специализации для типов данных или классов JVM. Для этого может использоваться функция `extend`, использование которой выглядит следующим образом:

```
(extend тип-или-класс
  протокол-1
   {:метод-1 уже-определенная-функция
    :метод-2 (fn [a b] ...)
    :метод-3 (fn ([a]...) ([a b] ...)...)}
  протокол-2
    {...}
...)
```

Для этой функции вы указываете имя типа данных или класса (или `nil`), и передаете список состоящий из названий протоколов (`протокол-1` и т.д.) и отображений, которые связывают функции протокола (`метод-1` и т.д.) с их реализациями — анонимными или именованными функциями.

Стоит отметить, что функция `extend` является низкоуровневым инструментом реализации протоколов. Кроме этого, в состав языка введены макросы `extend-protocol` & `extend-type`, которые немного упрощают реализацию протоколов[13]. Протокол также может быть реализован непосредственно при объявлении типа данных.

Использование `extend-type` выглядит практически также как и использование `extend`, но пользователь записывает реализации в более удобном виде (`extend-type` раскрывается в соответствующий вызов `extend`):

```
(extend-type тип-или-класс
  протокол-1
    (метод-2 [a b] ...)
    (метод-3 ([a]...)
             ([a b] ...)...)
  протокол-2
    (....)
...)
```

Макрос `extend-protocol` использоваться в тех случаях, если вы хотите реализовать один протокол для нескольких типов данных или классов. В общем виде использование `extend-protocol` выглядит следующим образом:

```
(extend-protocol название-протокола
  Тип-или-Класс-1
   (метод-1 ...)
   (метод-2 ...)
  Тип-или-Класс-2
   (метод-1 ...)
   (метод-2 ...)
...)
```

При использовании, `extend-protocol` раскрывается в серию вызовов `extend-type` для каждого из используемых типов.

Давайте рассмотрим небольшой пример. Пусть мы объявим следующий простой протокол:

```
(defprotocol Hello "Test of protocol"
  (hello [this] "hello function"))
```

Мы можем использовать `extend`, `extend-protocol`, или `extend-type` для его специализации для класса `String`:

```
(extend String
  Hello
  {:hello (fn [this] (str "Hello " this "!"))})

(extend-protocol Hello String
                 (hello [this] (str "Hello " this "!")))

(extend-type String Hello
             (hello [this] (str "Hello " this "!")))
```

При использовании любой из этих реализаций для объекта класса `String` мы получим один и тот же ответ:

```
user> (hello "world")
"Hello world!"
```

Стоит отметить, что если вы не реализовали протокол для определенного типа данных, то при вызове функции будет сгенерировано исключение. В том случае, если вам необходима "реализация по умолчанию", то вы можете специализировать протокол для класса `Object`.

Определение типов данных

В Clojure 1.2 введены два метода определения новых именованных типов данных (`deftype` и `defrecord`), которые реализуют абстракции, определенные протоколами и/или интерфейсами (к типам данных относится также `reify`, который описан ниже).

`deftype` и `defrecord` динамически создают именованный класс, который имеет набор заданных полей и (необязательно) методов для одного или нескольких протоколов и/или интерфейсов. Поскольку они не требуют явной компиляции, то это дает возможность их использования в интерактивной разработке.

С точки зрения разработчика `deftype` и `defrecord` похожи на `defstruct`, но во многом они отличаются:

- они создают уникальный класс с соответствующими полями;
- созданный класс имеет конкретный тип;
- имеется конструктор;
- для полей можно указывать типы (это будет использоваться для оптимизации и ограничения типов в конструкторе).

`deftype` является "базовым" инструментом для определения типов данных — созданный тип имеет только конструктор, и ничего больше — все остальное должен реализовывать разработчик. Но при этом, `deftype` может иметь изменяемые поля, чего не имеет `defrecord`.

В отличии от `deftype`, `defrecord` более прост в использовании, поскольку создаваемый тип данных имеет большую функциональность (по большей части за счет реализации интерфейсов `IKeywordLookup`, `IPersistentMap`, `Serializable` и т.д.):

- автоматически генерируемые функции `hashCode` и `equals`;
- возможность указания мета-информации;
- доступ к полям с помощью ключевых символов;
- вы можете добавлять поля, не указанные в определении.

`deftype` и `defrecord` обычно имеют разные области применения: `deftype` в основном используется для "системных" вещей — коллекций, и т.п., тогда как `defrecord` в основном используется для хранения информации из "проблемной области" — данных о заказчиках, записях в БД и т.п. — то, для чего использовались отображения в версиях 1.0 и 1.1.

Давайте рассмотрим как использовать конкретные средства для создания типов данных.

deftype & defrecord

В общей форме использование макросов `deftype` и `defrecord` выглядит следующим образом:

```
(deftype имя [& поля] & спецификации)
(defrecord имя [& поля] & спецификации)
```

Для обоих макросов обязательным параметром является лишь имя, которое становится именем класса. Поля, которые станут членами класса, перечисляются в векторе, следующем за именем, и могут содержать объявления типов. После этого вектора, можно указать список реализуемых интерфейсов и протоколов, вместе с реализацией (это не обязательно, поскольку для этого вы позже можете использовать `extend-protocol` & `extend-type`).

Спецификации протоколов/интерфейсов выглядят следующим образом:

```
протокол/интерфейс
(название-метода [аргументы*] реализация)*
```

Вы можете указать любое количество протоколов/интерфейсов, которые будут реализованы данным типом данных. Давайте посмотрим на простейший тип данных, который реализует протокол `Hello`:

```
(deftype A []
  Hello
  (hello [this] (str "Hello A!")))
```

Мы можем вызвать функцию `hello` для нашего объекта, и получим следующий вывод:

```
user> (hello (A.))
"Hello A!"
```

Мы можем также создать тип с помощью `defrecord`:

```
(defrecord B [name]
  Hello
  (hello [this] (str "Hello " name "!")))
```

и вызвать метод `hello` для этого типа:

```
user> (hello (B. "world"))
"Hello world!"
```

Как уже отмечалось выше, создаваемые поля по умолчанию являются неизменяемыми, но если вы создаете тип с помощью `deftype`, то вы можете пометить некоторые поля как изменяемые, используя метаданные (с помощью ключевого символа `:volatile-mutable` или `:unsynchronized-mutable`). Для таких полей вы сможете использовать оператор `(set! afield aval)` для изменения данных.

Давайте посмотрим как это делается на примере — если мы создадим следующий протокол и тип данных:

```
(defprotocol Setter
  (set-name [this new-name]))
(deftype AM [^{:volatile-mutable true} mfield]
  Hello
  (hello [this] (str "Hello " mfield "!"))
  Setter
  (set-name [this new-name] (set! mfield new-name)))
```

то мы сможем изменять значение поля:

```
user> (def am (AM. "world"))
#'user/am
user> (hello am)
"Hello world!"
user> (set-name am "peace")
"peace"
user> (hello am)
"Hello peace!"
```

reify

`reify` используется тогда, когда вам нужно реализовать протокол или интерфейс только в одном месте — когда вы используете `reify` вы одновременно объявляете тип, и сразу создаете объект этого типа. Функция `reify` по своему использованию очень похожа на `proxy`, но с некоторыми исключениями:

- можно использовать только для интерфейсов и протоколов;
- реализуемые методы являются методами результирующего класса, и они вызываются напрямую, без поиска в отображении, но при этом не поддерживается подмена методов в отображении.

Эти отличия позволяют получить более высокую производительность по сравнению с `proxy`, и при создании и при выполнении.

Вот небольшой пример реализации протокола `Hello` для конкретного объекта:

```
(def int-reify (reify Hello
                 (hello [this] "Hello integer!")))
```

И при вызове `hello` для этого объекта, мы получим соответствующий результат:

```
user> (hello int-reify)
"Hello integer!"
```

Дополнительные функции и макросы для работы с протоколами

Для работы с протоколами и типами данных определено некоторое количество вспомогательных функций, которые могут вам понадобиться:

`extends?`
: возвращает `true` если данный тип данных (2-й аргумент) реализует интерфейс, заданный первым аргументом;

`extenders`
: возвращает коллекцию типов, реализующих заданный протокол;

`satisfies?`
: возвращает `true` если данный протокол (1-й аргумент) применим к данному объекту (2-й аргумент);

Пространства имен и библиотеки

Пространства имен (namespaces) используются в Clojure для организации кода и данных. По своему характеру, пространства имен аналогичны пакетам (packages) в Common Lisp, и наиболее часто они используются при создании библиотек кода. Пространства имен являются объектами первого класса (first class objects), и могут динамически изменяться — создаваться, удаляться, изменяться, их можно перечислять и т.д. Пользователь может управлять видимостью символов используя метаданные, или специальные макросы, такие как `defn-`, который определяет функцию, видимую только в текущем пространстве имен.

При работе в REPL, все символы определяемые пользователем помещаются в пространство имен `user`. Пользователь может переключиться в другое пространство имен с помощью функции `in-ns` и/или подключить символы из других пространств имен с помощью функций `use`, `require` и `import`. Имя текущего пространства имен можно всегда найти в специальной переменной `*ns*`, которая автоматически устанавливается макросом `ns` и функцией `in-ns`.

Наиболее часто используемыми функциями при работе с пространствами имен являются:

`use`
: помещает в текущее пространство имен символы (все, или только указанные) из другого пространства имен, в том числе и находящихся в других библиотеках, загружая их при необходимости;

`require`
: загружает заданные библиотеки, но не помещает символы, определенные в них, в текущее пространство имен;

`import`
: используется для библиотек JVM и импортирует заданные классы из указанного пакета.

Каждая из этих функций имеет разное количество параметров, описание которых можно найти в документации. В качестве пример давайте рассмотрим следующий код:

```
(use 'clojure.contrib.str-utils)
(require 'clojure.contrib.lazy-xml)
(require '[clojure.contrib.str-utils2 :as str2])
(import 'org.apache.commons.io.FileUtils)
(import '(java.io File InputStream))
```

Первая строка загружает библиотеку `clojure.contrib.str-utils` и помещает все определенные в ней символы в текущее пространство имен. Вторая строка загружает библиотеку `clojure.contrib.lazy-xml`, но для доступа к ее объектам, необходимо использовать полное имя символа, включающее название пространства имен. Третья строка также загружает библиотеку, но создает псевдоним для названия пространства имен, что позволяет использовать более короткое имя символа, например, `str2/butlast`. Четвертый пример импортирует один класс (`FileUtils`) из пакета `org.apache.commons.io`, а в пятой строке мы видим как можно импортировать несколько классов из одного пакета.

При написании кода, лучше всего определять пространство имен с помощью макроса `ns`, который выполняет всю работу по созданию пространства имен, а также позволяет указать список импортируемых классов (используя `import`), используемых пространств имен (используя `use`), и т.п. операции, включая генерацию новых классов, с помощью `get-class`. В общей форме, использование макроса `ns` выглядит следующим образом:

```
(ns name
  (:require [my.cool.lib :as mcl])
  (:use my.lib2)
  (:import (java-package Class))
  .... more options)
```

Данный код определяет пространство имен `name`, импортирует в него класс `Class` из пакета `java-package`, импортирует библиотеку `my.lib2` и определяет псевдоним `mcl` для библиотеки `my.cool.lib`. Опции, указываемые в макросе `ns`, совпадают с опциями соответствующих функций. Более подробное описание вы можете найти в документации.

Описание дополнительных операций, которые можно производить с пространствами имен, вы можете найти в официальной документации.

Метаданные

Одним из интересных свойств Clojure является возможность связывания произвольных метаданных с символами, определенными в коде. Некоторые функции и макросы позволяют указывать определенные метаданные для управления видимостью символов, указания типов данных, документации, аннотаций для Java и т.п. Стоит отметить, что наличие метаданных никак не влияет на значения, связанные с символом. Например, если мы имеем два отображения, с одинаковым содержимым, но разными метаданными, то эти отображения будут эквивалентны между собой.

Для указания метаданных используется специальный синтаксис, который распознается функцией чтения кода. Эта функция переходит в режим чтения метаданных, если она встречает строку `#^` (в версии 1.2 можно просто писать `^`). После этой строки может быть указано либо название типа, например, `#^Integer`, либо отображение, перечисляющее ключ метаданных и значение, связанное с данным ключом. Стоит отметить, что явное указание типов программистом помогает компилятору сгенерировать более компактный код, что в свою очередь ведет к увеличению производительности программ.

Некоторые специальные формы, такие как `def` и т.п., имеют определенный набор названий ключей метаданных, которые могут изменять поведение определяемого символа. Например, следующий код:

```
(defn
 #^{:doc "my function"
    :tag Integer
    :private true}
 my-func [#^Integer x] (+ x 10))
```

определяет функцию `my-func`, которая получает и возвращает целое число (форма `#^Integer` при указании аргументов функции, и атрибут `:tag` для возвращаемого значения), имеет строку описания `my function`, и видима только в текущем пространстве имен, поскольку атрибут `:private` имеет истинное значение.

Если мы прочитаем метаданные данной функции:

```
user> (meta #'my-func)
{:ns #<Namespace user>, :name my-func,
  :file "NO_SOURCE_FILE", :line 1,
  :arglists ([x]), :doc "my function",
  :tag java.lang.Integer, :private true}
```

то мы увидим, что интерпретатор добавил дополнительные данные, такие как `:ns`, `:file` и т.д. Это выполняется для всех символов

Разработчик имеет возможность считывания и изменения метаданных символов с помощью функций. Функция `meta` возвращает отображение, содержащее все имеющиеся метаданные. А с помощью функции `with-meta` можно добавить или изменить метаданные заданного символа.

В версии 1.2 появилась возможность указания аннотаций для типов и интерфейсов, что позволяет упростить работу с библиотеками, которые используют аннотации в своей работе (EJB, Spring и т.п.). Для указания аннотаций используются метаданные, которые могут быть связаны как с самими типами, так и с конкретными полями и методами. Кроме стандартных для Java аннотаций, таких как `Retention`, `Deprecated` и т.д., вы можете использовать и аннотации, специфичные для конкретных библиотек. Пример использования аннотаций в библиотеке вы можете найти в следующем постинге.

Конкурентное программирование

Помимо стандартных средств Java, предназначенных для выполнения кода в отдельных потоках выполнения, Clojure имеет в своем арсенале собственные средства конкурентного выполнения кода (`pmap` и `pcalls`), выполнения кода в отдельном потоке, используя механизм `future` и синхронизации между потоками с помощью `promise`.

`pmap` — это параллельный вариант функции `map`, который может использоваться в тех случаях, когда функция-параметр не имеет побочных эффектов, и требует достаточно больших затрат на вычисление. Функция `pcalls` позволяет вычислить результат нескольких функций в параллельном режиме, возвращая последовательность их результатов в качестве результата выполнения функции.

Future & promise

Достаточно часто при разработке приложений возникает необходимость выполнения долго работающего кода одновременно с выполнением других задач. Для более простой работы с таким кодом, в версии 1.1 было введено понятие `future`.

`future` позволяет программисту выделить некоторый код в отдельный поток выполнения, который выполняется параллельно с основным кодом. Результат выполнения `future` затем сохраняется, и может быть получен с помощью операции `deref` (`@`). Эта операция может заблокировать выполнение основного кода, если работа `future` еще не завершилась — в этом `future` похож на `promise`, который описан ниже. Значение, установленное при выполнении `future` сохраняется, и при последующих обращениях к нему, возвращается сразу, без вычисления.

Рассмотрим простой пример:

```
(def future-test
     (future
       (do
          (Thread/sleep 10000)
          :finished)))
```

Тут создается объект `future`, в котором выполняется задержка на 10 секунд, а затем устанавливается значение `:finished`. Если мы обратимся к объекту `future-test` до завершения операции, то мы будем ожидать завершения указанного блока кода.

Но в отличие от `promise`, `future` имеет больше возможностей — вы можете проверить, закончилось ли выполнение кода с помощью функции `future-done?`, что позволяет избежать блокирования в случае обращения к еще не закончившейся операции. Кроме того, вы можете отменить выполнение операции с помощью функции `future-cancel` и проверить, не была ли отменена операция, с помощью функции `future-cancelled?`.

Иногда возникают ситуации, когда один поток исполнения должен передать какие-то данные другому. Это может быть организовано с помощью `promise`, которые в некоторых вещах похожи на `future`. Общая схема работы следующая: в одном потоке выполнения вы создаете некоторый объект с помощью `promise`, выполняете работу и затем с помощью `deliver` устанавливаете значение объекта. Результат, сохраненный в объекте, может быть получен с помощью операции `deref` (краткая форма `@`) и не может быть изменен после установки с помощью `deliver`. Но если вы попробуете обратиться к значению, сохраненному в объекте, до того, как оно будет установлено, то ваш поток выполнения будет заблокирован, и возобновит работу только после установки значения. Однако после того как значение было установлено, его получение будет производиться уже без выполнения кода, использующегося для его вычисления.

Рассмотрим следующий пример:

```
(def p (promise))
(do (future
      (Thread/sleep 5000)
      (deliver p :fred))
 @p)
```

В первой строке мы создаем объект `p`, который затем используется для синхронизации в блоке `do`. Если мы выполним код в блоке `do`, то выполнение затормозится на 5 секунд, поскольку поток выполнения, созданный `future`, еще не установил значение. А после окончания ожидания и установки значения с помощью `deliver`, операция `@p` сможет получить установленное значение равное `:fred`. Если мы попробуем выполнить операцию `@p` еще раз, то мы сразу получим установленное значение.

Работа с изменяемыми данными

Хотя по умолчанию переменные в Clojure неизменяемые, язык предоставляет возможность работать с изменяемыми переменными в рамках четко определенных моделей взаимодействия — как синхронных, так и асинхронных[14]. Сочетание неизменяемых данных с механизмами обновления данных (через ссылки, атомы и агенты) создает очень удобную среду для многопотокового программирования, что становится все более актуальным, поскольку число ядер в современных процессорах продолжает расти.

Стоит отметить, что в Clojure различаются понятия *состояния* (state) и *"названия"* (identity). Состояние — это значение, связанное с названием в конкретный момент времени. Значение, существуюшее в данном состоянии никогда не изменяется, а то, что выглядит обновлением данных, является на самом деле обновлением identity, которое начинает указывать на новое значение (state). В тоже время, старое состояние может продолжать существовать и использоваться из других потоков выполнения. Более подробно об этом можно прочитать в следующей статье или в 6-й главе книги Practical Clojure.

Имеющиеся средства для работы с изменяемыми данными можно классифицировать по нескольким параметрам, как показано в следующей таблице:

Вид изменения	**Синхронное**	**Асинхронное**
Координированное	ref	-
Независимое	atom	agent
Изолированное	var	-

В Clojure имеется три механизма синхронного обновления данных и один — асинхронного. Наиболее часто в коде используются ссылки (`ref`), которые предоставляют возможность синхронного обновления данных в рамках транзакций, и агенты (`agent`), которые реализуют механизмы асинхронного обновления данных. Кроме этого, существуют еще атомы, рассмотренные ниже, и "переменные" (vars). "Переменные" имеют "глобальное" (root) значение, которое определено для всех потоков выполнения, но это значение можно переопределять для отдельных потоков выполнения, используя `binding` или `set!`.

Хорошим примером использования возможностей Clojure в части работы с изменяемыми данными в многопоточных программах является пример "муравьи" (ants) который Rich Hickey продемонстрировал в видеолекции "Clojure Concurrency в которой рассказывается о возможностях Clojure в части конкурентного программирования. Еще один хороший пример использования Clojure для таких задач можно найти в серии статей Tim Bray.

Ссылки (refs)

Синхронное изменение данных производится через ссылки на объекты данных. Изменение ссылок можно проводить только в рамках явно обозначенных транзакций. Изменение нескольких ссылок в рамках транзакции[15] является атомарной операцией, обеспечивающей целостность данных и выполняемой в изоляции (atomicity, consistency, isolation) — ACI (аналогично свойствам транзакций в базах данных, но без долговечности (durability)).

Изменение данных с помощью ссылок возможно благодаря использованию Software Transactional Memory, которая обеспечивает целостность данных при работе с ними из нескольких потоков выполнения. Описание принципов работы STM, вместе с подробным описанием ее реализации в Clojure, вы можете найти в статье Software Transactional Memory Марка Волкманна (R. Mark Volkmann).

Чтобы обновить какой-то объект, необходимо сначала объявить его с использованием функции `ref`, а изменение затем выполняется с помощью операций `alter`, `commute` или `ref-set`, которые находятся внутри блоков `dosync` или `io!`, запускающих новую транзакцию. Доступ к данным на чтение осуществляется с помощью оператора `deref` (или специального макроса процедуры чтения — `@`). При этом операции чтения не видят результатов еще не закончившихся транзакций. Необходимо помнить о том, что транзакция может быть запущена повторно (retried), и это надо учитывать в функциях, вызываемых из функций `alter` или `ref-set`.

Рассмотрим, например, код для управления набором счетчиков (например, для сбора статистики по каким-то действиям):

```
(def counters (ref {}))

(defn add-counter [key val]
  (dosync (alter counters assoc key val)))

(defn get-counter [key]
  (@counters key 0))

(defn increment-counter [key]
  (dosync
   (alter counters assoc key (inc (@counters key 0)))))

(defn rm-counter [key]
  (let [value (@counters key)]
    (if value
      (do (dosync (alter counters dissoc key))
          value)
      0)))
```

Загрузим этот код, выполним несколько функций и посмотрим на состояние переменной `counters` после выполнения каждой из функций:

```
user> @counters
{}
user> (dosync (add-counter :a 1) (add-counter :b 2))
user> @counters
{:b 2, :a 1}
user> (dosync (increment-counter :a) (increment-counter :b))
user> @counters
{:a 2, :b 3}
```

Это простой пример, показывающий координированное изменение данных разных счетчиков, но эти функции можно использовать в разных потоках выполнения без страха потерять или получить неправильные данные.

Для обеспечения корректности данных, сохраняемых по ссылке, программист может установить функцию-валидатор. Это выполняется с помощью функции `set-validator!` (или сразу, при создании ссылки), которая получает два аргумента — ссылку и функцию-валидатор для данной ссылки. В том случае, если программист устанавливает некорректное значение, функция-валидатор должна вернуть ложное значение или выбросить исключение. Например, чтобы запретить отрицательные значения счетчиков, мы можем использовать следующую функцию-валидатор:

```
(set-validator! counters
  (fn [st] (or
             (empty? st)
             (not-any? #(neg? (second %)) st))))
```

и если пользователь попробует установить отрицательное значение счетчика, то Clojure выдаст ошибку.

Кроме этого, при работе с ссылками вы можете использовать так называемые функции-наблюдатели, которые позволяют получать информацию об изменениях состояния. Для добавления функции-наблюдателя вы можете воспользоваться функцией `add-watch`, которая принимает в качестве аргумента функцию, которая будет вызвана при изменении состояния, и ей будут переданы предыдущее и новое значение ссылки.

Более подробную информацию о работе с ссылками вы можете найти на сайте языка.

Агенты (agents)

Агенты позволяют осуществлять асинхронное обновление данных. Работа с агентами похожа на работу со ссылками (только вы должны использовать `agent` вместо `ref`), но обновление данных может произойти в любой момент (и программист не может на это влиять) в зависимости от количества заданий. Эти задания выполняются в отдельном пуле потоков выполнения, размер которого ограничен. В отличие от ссылок, вам нет необходимости явно создавать транзакцию с помощью функции `dosync` — вы просто посылаете "сообщение", состоящее из функции, которая установит новое значение агента, и аргументов для этой функции.

Пример со счетчиками, переписанный на использование агентов, будет выглядеть следующим образом:

```
(def acounters (agent {}))

(defn add-acounter [key val]
  (send acounters assoc key val))

(defn increment-counter [key]
  (send acounters update-in [key] (fnil inc 0)))

(defn get-acounter [key]
  (@acounters key 0))

(defn rm-acounter [key]
  (let [value (@acounters key)]
    (send acounters dissoc key)
    value))
```

Функции `send` и `send-off` получают в качестве аргументов имя агента, функцию, которую надо выполнить, и дополнительные параметры, которые будут переданы вызываемой функции. Вызываемая функция получает в качестве аргумента текущее состояние агента, и должна вернуть новое значение, которое получит агент[16]. Во время своего выполнения функция "видит" актуальное значение агента.

Разница между `send` и `send-off` заключается в том, что они используют разные по размеру пулы нитей выполнения. `send` рекомендуется применять для действий, которые ограничены по времени выполнения, такие как `conj` и т.д. А `send-off` лучше использовать для длительно выполняемых задач и задач, которые могут зависеть от ввода/вывода и других блокируемых операций.

В некоторых случаях вам может понадобиться, чтобы задания, посланные агенту, были завершены. Для этого в язык введены две функции, которые позволяют остановить выполнение текущего потока выполнения до завершения задач переданных агенту (или агентам). Функция `await` блокирует выполнение текущего кода до завершения всех задач, а функция `await-for` блокирует выполнение на заданное количество миллисекунд и возвращает контроль текущему потоку, даже если выполнение всех задач не было завершено.

Так же как и при использовании ссылок, при работе с агентами вы можете использовать функции-валидаторы и функции-наблюдатели. Остальную информацию об агентах вы можете найти на отдельной странице сайта языка.

Атомы (atoms)

Атомы предоставляют возможность синхронного изменения независимых данных, для которых не требуется синхронизация в рамках транзакции. Работа с атомами похожа на работу со ссылками, только производится без координации: вы объявляете переменную с помощью функции `atom`, можете получить доступ к значению используя `deref` (или `@`) и установить новое значение с помощью функции `swap!` (или низкоуровневой функции `compare-and-set!`).

Изменения осуществляются с помощью функции `swap!`, которая в качестве аргументов принимает функцию и аргументы для этой функции (если необходимо). Переданная функция применяется к текущему значению атома для получения нового значения, и затем делается попытка атомарного изменения с помощью `compare-and-set!`. В том случае, если другой поток выполнения уже изменил значение атома, то вызов пользовательской функции повторяется для вычисления нового значения, и опять делается попытка изменения значения атома и т.д., пока попытка изменения не будет успешной[17].

Вот простой пример кода, который использует атомы:

```
(def atom-counter (atom 0))
(defn increase-counter []
  (swap! atom-counter inc))
```

При использовании данного кода мы можем быть уверены, что значение счетчика будет увеличиваться всегда, независимо от того, сколько потоков выполнения вызывают эту функцию. Подробней об атомах вы можете прочитать на сайте языка.

Взаимодействие с Java

Clojure реализует двухстороннее взаимодействие с библиотеками, работающими на базе JVM — код на Clojure может использовать существующие библиотеки и вызываться из других библиотек, реализовывать классы и т.п. Отдельно стоит отметить поддержку работы с массивами объектов Java — поскольку они не являются коллекциями, то Clojure имеет отдельные операции для работы с массивами: создание, работа с индивидуальными элементами, конвертация из коллекций в массивы и т.д. Подробную информацию о взаимодействии с Java вы можете найти на сайте языка.

Вы также можете встроить Clojure в ваш проект на Java и использовать его в качестве языка расширения. Дополнительную информацию об этом вы можете найти в учебнике о Clojure.

Работа с библиотеками Java

Код, написанный на Clojure, может без особых проблем использовать библиотеки, написанные для JVM. По умолчанию в текущее пространство имен импортируются классы из пакета `java.lang`, что дает доступ к основным типам данных и их методам. А остальные пакеты и классы должны импортироваться явно, как это описано в разделе Пространства имен, иначе вам необходимо будет использовать полные имена классов с указанием названий пакетов.

Создание экземпляра класса производится с помощью специальной формы `new`, которая принимает в качестве аргументов имя нужного класса (записываемое с заглавной буквы) и аргументы, которые будут переданы конструктору класса. Кроме этого, определен макрос, который позволяет записывать создание новых экземпляров класса в более компактной форме. Для этого необходимо добавить знак `.` (точка) после имени класса и указать нужные аргументы для конструктора класса.

Например, следующие виды записи эквивалентны:

```
(new String "Hello")
(String. "Hello")
```

Для обращения к членам классов существует несколько форм:

- для доступа к не статическим членам класса используется форма `(.имяЧленаКласса объект аргументы*)` или `(.имяЧленаКласса ИмяКласса аргументы*)`. Например, `(.toUpperCase "Hello")` в результате вернет `"HELLO"`;
- для доступа к статическим членам класса используется запись вида `(ИмяКласса/имяМетода аргументы*)` — для вызова методов, или `ИмяКласса/имяПеременной` — для переменных. Например, `(Math/sin 1)` или `Math/PI`.

Данные формы являются макросами, которые раскрываются в вызов специальной формы `.` (точка). В общем виде эта форма выглядит следующим образом: `(. объект имяЧленаКласса аргументы*)` или `(. ИмяКласса имяЧлена аргументы*)`. Такчто вызов `(Math/sin 1)` раскрывается в `(. Math sin 1)`, вызов `(.toUpperCase "Hello")` в `(. "Hello" toUpperCase)` и т.д.

Существует еще один макрос, который позволяет организовывать связанные вызовы вида `System.getProperties().get("os.name")`, которые очень часто встречаются в коде на Java. Этот макрос называется `..` (две точки) и записывается в виде `(.. объектИлиИмяКласса выражение+)`. Например, код на Java, приведенный выше, в Clojure будет выглядеть следующим образом:

```
(.. System (getProperties) (get "os.name"))
```

В том случае, если нет необходимости передавать аргументы, можно использовать запись выражения без скобок:

```
(.. System getProperties (get "os.name"))
```

Есть еще одна форма, которая позволяет выполнить вызовы нескольких методов, примененных к одному объекту — это макрос `doto`, который в качестве аргументов получает объект и выражения, и в качестве результата возвращает объект. Например, следующий код:

```
(doto
  (new java.util.HashMap)
  (.put "a" 1) (.put "b" 2))
```

создаст новое отображение и поместит в него два объекта.

Поскольку объекты Java в отличие от объектов Clojure изменяемы, то программист имеет возможность установки значений полей класса. Это выполняется с помощью специальной формы `set!`, которая имеет следующий вид: `(set! (. объектИлиИмяКласса имяЧленаКласса) выражение)`. Однако помните, что вы можете применять эту форму только к классам Java.

Вы также можете использовать методы классов в качестве функций первого порядка. Для этого определен макрос `memfn`, который принимает имя метода и список аргументов этой функции, и создает соответствующую функцию Clojure. Например, код:

```
(map (memfn toUpperCase) ["aa" "bb" "cc"])
```

применит метод `toUpperCase` из класса `String` к каждой из строк вектора. В простых случаях этот код можно заменить на анонимную функцию вида:

```
(map #(.toUpperCase %) ["aa" "bb" "cc"])
```

но в некоторых случаях `memfn` просто удобнее.

Вызов кода на Clojure из Java

Существует несколько причин, по которым вам может понадобится вызвать код, написанный на Clojure из Java. Первая причина — вам необходимо реализовать так называемые обратные вызовы (callbacks), которые будут реализовывать обработку каких-то событий, например, при обработке XML файла или при реализации GUI. Вторая причина — вы хотите реализовать некоторую функциональность на Clojure, и позволить классам Java пользоваться этой функциональностью.

В Clojure существуют разные способы выполнения этих задач — вы можете создавать анонимные классы, полезные при реализации callbacks, с помощью макроса `proxy`, или создавать именованные классы с помощью макроса `gen-class`. Обе эти возможности описываются более подробно в следующих разделах.

Реализация обратных вызовов (callback) с помощью proxy

Макрос `proxy` используется для создания анонимных классов, которые реализуют указанные интерфейсы и/или расширяют существующие классы. В общем виде вызов этого макроса выглядит следующим образом: `(proxy [списокКлассовИлиИнтерфейсов] [аргументыКонструктораКласса] РеализуемыеМетоды+).`

Например, если вы хотите обрабатывать XML с помощью парсера SAX, то вы можете создать свой класс, который будет обрабатывать определенные события:

```
(import '(org.xml.sax InputSource)
        '(org.xml.sax.helpers DefaultHandler)
        '(java.io StringReader)
        '(javax.xml.parsers SAXParserFactory))

(def print-element-handler
  (proxy [DefaultHandler] []
    (startElement
      [uri local qname atts]
      (println (format "Saw element: %s" qname)))))

(defn demo-sax-parse [source handler]
  (.. SAXParserFactory newInstance newSAXParser
      (parse (InputSource. (StringReader. source))
             handler)))
(demo-sax-parse "<foo><bar>body</bar></foo>" print-element-handler)
```

и после выполнения этого кода на стандартный вывод будут выданы названия элементов, составляющих данный XML документ.

Создание классов с помощью gen-class

Как отмечалось выше, макрос `gen-class` используется для создания именованных классов, которые будут доступны для кода на Java, только если вы откомпилируете исходный код в байт-код. В отличие от `proxy`, `gen-class`[18] имеет значительно больше опций, которые управляют его поведением, но при этом он предоставляет и большую функциональность.

В общем виде вызов макроса выглядит следующим образом: `(gen-class опции+)`. Полный список опций можно найти в официальной документации или в записи в блоге Meikel Brandmeyer, а здесь мы приведем небольшой пример реализации класса и рассмотрим, из чего он состоит:

```
(ns myclass
  (:import
    (org.apache.tika.parser Parser
                            AutoDetectParser
                            ParseContext)))

(gen-class
 :name MyClass
 :implements [org.apache.tika.parser.Parser])

(defn -parse [this stream handler metadata context]
  '())
```

В данном примере мы создаем класс `MyClass`, который реализует интерфейс `org.apache.tika.parser.Parser` и определяет метод `parse`, принимающий четыре аргумента. После компиляции этого кода, мы можем использовать его из кода на Java как самый обычный класс.

Отметьте, что методы не указываются в объявлении класса, а реализуются в текущем пространстве имен. Но это относится только к тем методы, которые уже объявлены в родительском классе или интерфейсе. Также заметьте, что имя реализуемого метода начинается со знака - (минус) — это префикс, который используется по умолчанию, чтобы отличать методы-члены класса от обычных функций. Разработчик может выбрать другой префикс с помощью опции `:prefix`.

В том случае, если вы хотите расширить существующий класс, вам необходимо использовать опцию `:extends`, вместо или наравне с опцией `:implements`, которая приведена в нашем примере.

Для инициализации класса вы можете использовать функцию, которая указывается в опции `:init`, и которой будут переданы аргументы конструктора класса. Кроме этого, существует опция `:constructors`, которая может использоваться, если вы хотите создать конструкторы класса, не совпадающие по количеству аргументов с конструкторами родительского класса. А новые методы могут быть добавлены к классу с помощью опции `:methods`.

По умолчанию, сгенерированный класс не имеет доступа к защищенным переменным родительского класса. Однако, к ним можно получить доступ, если использовать опцию `:exposes`. А с помощью опции `:exposes-methods` можно указать псевдонимы для методов родительского класса, если вам необходимо вызывать их из вашего класса.

Еще одной полезной опцией является опция `:state`, которая указывает имя переменной, в которой будет храниться внутреннее состояние вашего класса. Обычно в качестве значения используется `ref` или `atom`, которые могут быть изменены в процессе выполнения методов класса. Стоит отметить, что данное состояние должно быть установлено функцией, указанной в опции `:init`.

Поддержка языка

Эффективное использование языка невозможно без наличия инфраструктуры для работы с ним — редакторов кода, средств сборки, библиотек и т.п. вещей. Для Clojure имеется достаточное количество таких средств — как адаптированных утилит (Maven, Eclipse, Netbeans и т.п.), так и разработанных специально для этого языка — например, системы сборки кода Leiningen. Отладку приложений, написанных на Clojure, поддерживают почти все среды разработки, перечисленные ниже, а для профилирования можно использовать существующие средства для Java.

Число библиотек для Clojure постоянно увеличивается. Некоторые из них — лишь обертки для библиотек написанных на Java, а некоторые — специально разработанные для Clojure. Вместе с Clojure часто используют набор библиотек clojure-contrib, который содержит различные полезные библиотеки, не вошедшие в состав стандартной библиотеки языка: функции для работы со строками и потоками ввода/вывода, дополнительные функции для работы с коллекциями, монады и т.д. Среди других библиотек можно отметить `Compojure` — для создания веб-сервисов; `ClojureQL` — для работы с базами данных; `Incanter` — для статистической обработки данных; `crane`, `cascading-clojure` и `clojure-hadoop` — для распределенной обработки данных. Это лишь малая часть существующих библиотек, многие из которых перечислены на сайте языка.

Среды разработки

В настоящее время для работы с Clojure разработано достаточно много средств — поддержка Clojure имеется в следующих редакторах и IDE:

Emacs

Подсветка синтаксиса и расстановка отступов выполняются с помощью пакета `clojure-mode`. Для выполнения кода можно использовать `inferior-lisp-mode`, но лучше воспользоваться пакетом SLIME, для которого существует адаптер для Clojure — swank-clojure. SLIME разработан для работы с разными реализациями Lisp и предоставляет возможности интерактивного выполнения и отладки кода, анализа ошибок, просмотра документации и т.д. Судя по последнему опросу среди программистов на Clojure, Emacs и SLIME являются самым популярным средством разработки.
Установка обоих пакетов может быть выполнена (и это рекомендуется авторами пакетов) через Emacs Lisp Package Archive. Небольшое описание того, как установить и настроить `clojure-mode` и SLIME, вы можете найти в записи в блоге Романа Захарова.

Если вы используете Windows, то вы можете воспользоваться Clojure Box — пакетом, в котором поставляется уже настроенный Emacs, SLIME, Clojure и библиотека `clojure-contrib`. Использование этого пакета позволяет немного упростить процесс освоения языка.

Vim

Поддержка Clojure в Vim реализуется с помощью модуля VimClojure, который реализует следующую функциональность:

- подсветку синтаксиса языка;
- правильную расстановку отступов;
- выполнение кода;
- раскрытие макросов;
- дополнение символов (omni completion);
- поиск в документации, как для самого кода на Clojure, так и в документации Java (javadoc).

На домашней странице проекта вы можете найти необходимую информацию по установке плагина, а также скринкаст, демонстрирующий возможности VimClojure.

Eclipse

Для Eclipse существует плагин Counterclockwise, который обеспечивает выполнение следующих задач:

- подсветка, расстановка отступов и форматирование исходного кода;
- навигация по исходному коду;
- базовая функциональность по дополнению имен функций и переменных, включая функции библиотек написанных на Java;
- выполнение кода в REPL;
- отладка на уровне исходного кода.

Информацию по установке вы можете найти на странице проекта.

Netbeans

В Netbeans поддержка Clojure осуществляется плагином Enclojure со следующей функциональностью:

- подсветка и расстановка отступов в исходном коде, а также работа с S-выражениями;
- выполнение кода в REPL, включая работу с REPL на удаленных серверах, историю команд, тесную интеграцию с редактором кода;
- навигация по исходному коду, включая навигацию для мультиметодов;
- дополнение имен для функций Clojure и Java;
- отладка на уровне исходного кода, с установкой точек останова, показом значений переменных и пошаговым выполнением кода.

IntelliJ IDEA

Для этой IDE создан плагин La Clojure, реализующий следующие функции:

- подсветка и форматирование исходного кода с возможностью настройки пользователем;
- навигация по исходному коду;
- свертывание определений функций и переменных;
- дополнение имен для функций, переменных и пространств имен Clojure, а также поддержка дополнений для имен классов и функций в библиотеках написанных на Java;
- выполнение кода в REPL;
- отладка кода, в том числе и для кода в REPL;
- рефакторинг кода на Clojure;
- компиляция исходного кода в Java classes.

Описание того, как установить эти средства можно найти на сайте разработчиков языка. Кроме того, процесс установки некоторых из этих средств можно найти в наборе скринкастов, созданных Sean Devlin.

Компиляция и сборка кода на Clojure

Сборку кода, написанного на Clojure, можно осуществлять разными способами — начиная с компиляции, используя Clojure в командной строке, и заканчивая использованием высокоуровневых утилит для сборки кода, таких как Maven и Leiningen.

В принципе, компиляция кода — необязательный этап, поскольку Clojure автоматически откомпилирует загружаемый код, и многие проекты пользуются этим, распространяясь в виде исходных кодов. Однако предварительная компиляция (ahead-of-time, AOT) позволяет ускорить загрузку вашего кода, сгенерировать код, который будет использоваться из Java, а также позволяет не предоставлять исходный код, что важно для коммерческих проектов.

Компиляция кода на Clojure осуществляется в соответствии со следующими принципами:

- единицей компиляции является пространство имен;
- для каждого файла, функции и `gen-class` создаются отдельные файлы `.class`;
- также для каждого файла создается класс-загрузчик, вида `имя-файла__init.class`;
- файл, содержащий пространство имен, использующий имя со знаком - (минус), должен иметь имя, в котором - заменены на знак _ (подчеркивание).

Компиляция кода с помощью Clojure

Для компиляции из REPL имеется функция `compile`, которая в качестве аргумента получает символ, определяющий пространство имен, например:

```
(compile 'my-class)
```

что приведет к компиляции файла `my_class.clj`. Стоит отметить, что `CLASSPATH` также должен содержать в себе каталог `class`, находящийся в том же каталоге, что и исходный файл. В этот каталог будут помещены сгенерированные файлы `.class`.

Провести компиляцию исходного текста можно и не запуская REPL, для этого можно воспользоваться следующей командой:

```
java -cp clojure.jar:`pwd`/class -Dclojure.compile.path=class
clojure.lang.Compile my-class
```

которая выполняет компиляцию пространства имен `my-class`, находящегося в файле `my_class.clj`. Заметьте, что в `CLASSPATH` явно добавлен подкаталог `class`, указанный с помощью свойства `clojure.compile.path`. Команды такого вида можно использовать в других системах сборки, таких как Ant.

Ant

Чтобы не изобретать код для компиляции файлов Clojure для каждого нового проекта сборки, был создан проект clojure-ant-tasks, который определяет стандартные задачи (tasks) для компиляции и тестирования кода, написанного на Clojure. Подробное описание использования пакета задач вы можете найти на странице проекта.

Использование Maven с Clojure

Система сборки кода Maven достаточно популярна среди разработчиков на Java, поскольку она позволяет декларативно описывать процесс сборки, тестирования и деплоймента, а выполнение конкретных задач ложится на плечи конкретных модулей (plugins).

Для Maven написан модуль clojure-maven-plugin, который позволяет компилировать и тестировать код, написанный на Clojure. Этот модуль позволяет прозрачно интегрировать Clojure в существующую систему сборки на основе Maven. Кроме компиляции и тестирования, данный модуль определяет дополнительные задачи, такие как запуск собранного пакета, запуск REPL с загрузкой собранного пакета, а также запуск серверов SWANK или Nailgun, что позволяет использовать SLIME и VimClojure для интерактивной работы с собранным пакетом.

Подробное описание того, как использовать этот модуль вместе с Maven, вы можете найти в следующей статье.

Leiningen

Для Clojure также существует своя собственная система сборки — Leiningen, описывающая проекты и процесс сборки, используя язык Clojure. В последнее время эта система становится все более популярной — она имеет возможности расширения с помощью дополнительных модулей, например, для компиляции кода на Java и т.п.

Из коробки Leiningen позволяет выполнять базовые задачи — компиляцию кода, тестирование, упаковку кода в jar-архив, сборку jar-архива со всеми зависимостями и т.д. Кроме того, имеется базовая поддержка работы с Maven, что позволяет использовать собранный код в других проектах.

Установка Leiningen достаточно проста и описана на странице проекта. После установки вы можете начать его использовать в своем проекте, добавив файл `project.clj`, содержащий что-то вроде следующего кода:

```
(defproject test-project "1.0-SNAPSHOT"
  :description "A test project."
  :url "http://my-cool-project.com"
  :dependencies [[org.clojure/clojure "1.2.0"]
                 [org.clojure/clojure-contrib "1.2.0"]]
  :dev-dependencies [[org.clojure/swank-clojure "1.0"]])
```

который определяет новый проект `test-project` с зависимостями от Clojure и набора библиотек `clojure-contrib`, а также зависимостью, которая используется в процессе разработки — `swank-clojure`. `defproject` — это макрос Clojure, который раскрывается в набор инструкций по сборке, и является единственной обязательной конструкцией, которая должна быть указана в файле `project.clj`. Кроме этого, `project.clj` может содержать и произвольный код на Clojure, выполняемый в процессе сборки. Более подробную информацию о Leiningen можно найти на странице проекта или в следующей статье.

Репозитории кода

Некоторые системы сборки, такие как Maven и Leiningen, поддерживают автоматическую загрузку зависимостей из центральных репозиториев кода. Для Clojure также имеются отдельные репозитории, совместимые с этими системами.

В первую очередь это build.clojure.org, который содержит сборки как самой Clojure, так и набора библиотек `clojure-contrib`. Например, для Maven вы можете добавить Clojure в зависимости с помощью следующего кода, добавленного в файл проекта `pom.xml`:

```
<repositories>
  <repository>
    <id>clojure-releases</id>
    <url>http://build.clojure.org/releases</url>
  </repository>
</repositories>
```

Кроме того, для распространения библиотек написанных на Clojure, был создан проект clojars.org, который поддерживает работу с Maven и Leiningen, и на котором можно найти достаточно большое количество полезных библиотек.