Мультипарадигмене програмування

Лекції №7,8
Функціональна мова програмування
Clojure

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`.

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;
- Использовать высокоуровневые абстракции для типов данных и операций над ними, 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.

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.

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 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.

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.

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.

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.

Заключение

Количество проектов (в том числе и коммерческих) на Clojure постоянно увеличивается, и, может быть, вы также сможете использовать данный язык для написания программ, которые будут работать на платформе JVM.

1. Также существует версия для платформы .Net, но в ней пока отсутствуют некоторые возможности, реализованные в Clojure для JVM

2. В интернете можно найти примеры кода из книг On Lisp и Practical Common Lisp, переписанные на Clojure

3. Фактически, своего адреса в памяти

4. Примеры взяты из описания на сайте языка. Конструкции `loop` и `recur`, используемые в них, применяются для организации циклов и описаны далее

5. Специальные формы — это отдельные элементы языка, для которых не выполняются стандартные правила вычисления. Про специальные формы в Lisp вы можете прочитать в отдельной статье

6. Форма `recur` также может использоваться отдельно, без `loop` — тогда он выполнит переход к началу функции, в которой он используется

7. Результат должен быть объектом, унаследованным от `Throwable`

8. Внутри Clojure функции представляются как классы, реализующие интерфейс `IFn`, с функцией `invoke`, получающей нужное количество параметров

9. Стоит отметить, что можно добавлять только родительские объекты, а создание потомков возможно только через стандартные механизмы создания классов с помощью `gen-class`

10. Стоит однако отметить, что протоколы не реализуют monkey patching и внедрение методов (injection) в существующие типы данных.

11. Возможность реализации абстракций на Clojure и высокая скорость работы протоколов позволит в будущем написать Clojure на самой Clojure, без использования исходного кода на Java.

12. Люди знакомые с Haskell могут рассматривать протоколы как некоторое подобие типов классов (typeclasses) в этом языке, правда при этом нельзя определять реализации по умолчанию для методов.

13. Но `extend` может использоваться в тех случаях, когда вы хотите использовать одни и те же реализации для разных типов данных — в этом случае, вы можете создать отображение с нужными функциями, и использовать его для разных типов, например, как описано в следующем блог-постинге.

14. Стоит отметить, что изменяются не данные, а ссылки на данные. В статье мы будем говорить об "изменяемых данных", понимая под этим использование соответствующих механизмов изменения

15. Транзакции нужны не только для изменения данных, но и для координированного чтения данных из нескольких ссылок

16. Т.е. новое значение агента равно результату выполнения `(apply функция состояние-агента аргументы)`

17. В текущей версии Clojure количество попыток изменения ограничено значением 10000

18. Очень часто `gen-class` используется в объявлении пространства имен с помощью макроса `ns`