Функциональное программирование в примерах

Created By	Ⓓ Daniil A (Kiky Tokamuro)
Last Edited	@Nov 20, 2019
Tags	functional programming lisp python

Краткое объяснение:

Если верить Википедии, то функциональное программирование — это раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). То есть функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

Особенности функционального программирования:

- Чистые функции — это функции, которые не обладают побочными эффектами, а так же являются детерминированными.
- Анонимные функции / Лямбда функции — это функции, которые объявляются в месте использования, и не получают уникального идентификатора для доступа к ним. Чаще всего для определения анонимных функций используют лямбда выражения.
- Функции высшего порядка — это функции, которые могут принимать другие функции в качестве аргументов, или возвращать другие функции в качестве результата.
- Композиция функций — это применение одной функции к результату другой функции.
- Каррирование — это преобразование функции с многими аргументами, к набору функций от одного аргумента.
- Замыкание — это комбинация функции и окружения, в котором она была определена. То есть это функция, которая определяется в теле другой функции, и создается каждый раз во время ее выполнения, а так же она имеет ссылки на локальные переменные внешней функции.
- Неизменяемые данные — это данные, состояние которых не может быть изменено после создания.
- Сопоставление с образцом — это метод анализа и обработки данных, основанный на сопоставлении исследуемого значения с тем или иным образцом, в качестве которого может выступать число, строка, или иная конструкция поддерживаемая языком.
- Рекурсия — это вызов функции из нее же самой, или через другие функции.

Функции первого класса:

Функции могут передаваться в качестве аргументов другим функциям, возвращаться из функций, храниться в переменных и структурах данных и создаваться во время выполнения.

Примеры на языке Python:

```
# Определение функции которая прибавляет единицу к каждому элементу списка:
def inc(l):
    return [x + 1 for x in l]

# Функция, которая принимает функцию в качестве аргумента, и возвращает другую функцию как результат:
def changeList(f, l):
    return f(l)

# changeList(f, l) = f(l)
>>> changeList(inc, [1,2,3])
[2, 3, 4]


# Хранение функции в переменной
>>> c = changeList
>>> c(inc, [2, 3])
[3, 4]


# Хранение функций в структурах данных предоставляемых языком:
# В данном случае, хранение анонимных функций в словаре.
hashIncDec = {
    'inc': lambda x: x + 1,
    'dec': lambda x: x - 1
}

>>> hashIncDec['inc'](5)
6
>>> hashIncDec['dec'](5)
4
```

Примеры на языке Scheme (GNU Guile):

```
;; Определение функции которая прибавляет единицу к каждому элементу списка:
(define (inc-list l)
  (map 1+ l))

;; Функция, которая принимает функцию в качестве аргумента, и возвращает другую функцию как результат:
(define (change-list f l)
  (f l))

scheme@(guile-user)> (change-list inc-list '(1 2 3))
$1 = (2 3 4)


;; Хранение функции в переменной
(define c change-list)

scheme@(guile-user)> (c inc-list '(1 2 3))
$2 = (2 3 4)


;; Хранение функций в структурах данных предоставляемых языком:
;; В данном случае, хранение анонимных функций в хеш-таблице.
(define hash-inc-dec (make-hash-table))

(hash-set! hash-inc-dec 'inc (lambda (x) (+ x 1) ))
(hash-set! hash-inc-dec 'dec (lambda (x) (- x 1) ))

scheme@(guile-user)> ((hash-ref hash-inc-dec 'inc) 5)
$3 = 6
scheme@(guile-user)> ((hash-ref hash-inc-dec 'dec) 5)
$4 = 4
```

Чистые функции:

- В чистых функциях отсутствуют побочные эффекты, так же как и в математических функциях.

- В чистых функциях для одного и того же аргумента, всегда будет возвращаться один и тот же результат.
- Результат вызова функции определяется ее входными аргументами.
- Если функция выполняет ввод/вывод информации, она уже не является чистой (чтение из файла, запись в файл, печать строки, и тд).
- Чистые функции являются детерминированными.
- Чистые функции не имеют состояния.

Таким образом, чистые функции, делают работу кода более предсказуемым, а так же улучшают удобство его отладки, и дают возможность осуществлять композицию функций.

Примеры на языке Python:

```
# Чистая функция, которая ничего не изменяет, и при одном и том же аргументе, всегда возвращает один и тот же результат:
def inc(x):
  return x + 1

>>> inc(10)
11
>>> inc(10)
11
>>> inc(10)
11


# Грязная функция, которая при одном и том же аргументе, возвращает разные результаты:
# Грязные функции могут осуществлять ввод/вывод, так же они могут изменять состояние или глобальные переменные.
from random import randint

def randInc(x):
  return randint(0, 100) + x

>>> randInc(1)
94
>>> randInc(1)
6
>>> randInc(1)
33
```

Примеры на языке Scheme (GNU Guile):

```
;; Чистая функция, которая ничего не изменяет, и при одном и том же аргументе, всегда возвращает один и тот же результат:
(define (inc x)
  (+ x 1))

scheme@(guile-user)> (inc 10)
$1 = 11
scheme@(guile-user)> (inc 10)
$2 = 11
scheme@(guile-user)> (inc 10)
$3 = 11


;; Грязная функция, которая при одном и том же аргументе, возвращает разные результаты:
(define (rand-inc x)
  (+ (random 100) x))

scheme@(guile-user)> (rand-inc 1)
$1 = 62
scheme@(guile-user)> (rand-inc 1)
$2 = 52
scheme@(guile-user)> (rand-inc 1)
$3 = 37
```

Ссылочная прозрачность:

Ссылочная прозрачность подразумевает, что любое подвыражение может быть заменено его значением в выражении в любое время без изменения всего выражения. В математике все функции являются ссылочно прозрачными согласно определению математической функции. То есть к примеру `18 = square(4) + 2` может быть изменено на `18 = 16 + 2`. Поскольку прозрачность ссылок требует одинаковых результатов для любого заданного набора входных данных в любой момент времени, то ссылочно прозрачное выражение является детерминированным. Ссылочная прозрачность возможна только при отсутствии побочных эффектов.

Плюсы ссылочной прозрачности:

- Возможность трансформации программы.
- Каждое выражение может быть заменено его значением.
- Возможности для оптимизации компилятора.
- Программу с ссылочной прозрачностью легче объяснить, так как не приходиться волноваться о побочных эффектах.
- Возможность для рефакторинга программы.

Пример на языке Python:

```
# Ссылочная прозрачность
def square(x):
  return x ** 2

>>> square(4) + 2 == 16 + 2
True
```

Пример на языке Scheme (GNU Guile):

```
;; Ссылочная прозрачность
(define (square x)
  (* x x))

scheme@(guile-user)> (= (+ (square 4) 2) (+ 16 2))
$1 = #t
```

Замыкание:

Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

Пример на языке Python:

```
# Замыкание
def sum(x):
    def sumXY(y):
        return x + y
```

```
    return sumXY

>>> s = sum(10)
>>> result = s(5)
>>> result
15
```

Пример на языке Scheme (GNU Guile):

```
;; Замыкание
(define (sum x)
  (lambda (y)
    (+ x y)))

(define sum-x-y (sum 10))

scheme@(guile-user)> (sum-x-y 5)
$1 = 15
```

Каррирование:

Каррирование — это разложение функции с многими аргументами в цепочку функций с одним аргументом. Возможность такого преобразования впервые отмечена в трудах Готтлоба Фреге, систематически изучена Моисеем Шейнфинкелем в 1920-е годы, а наименование получило по имени Хаскелла Карри — разработчика комбинаторной логики, в которой сведение к функциям одного аргумента носит основополагающий характер. Каррированные функции могут принимать по одному аргументу за раз, а обычные функции должны принимать все аргументы сразу.

Пример:

```
Обычная функция:
f(x, y) = x * x + y
f(5, 10) = 5 * 5 + 10 = 25 + 10 = 35
f(5, 10) = 35

Эта же функция, преобразованная с помощью каррирования:
g(x) = (x -> y -> x * x + y)
g(5)(10) = (5 -> 10 -> 5 * 5 + 10) = 35

(x -> y -> x * x + y) 5 10
  = (x -> (y -> x * x + y)) 5 10
  = ((x -> (y -> x * x + y)) 5) 10
  = (y -> 5 * 5 + y) 10
  = 5 * 5 + 10
  = 35
```

Пример на языке Python:

```
def f(x):
  return lambda y: x * x + y

>>> f(5)(10)
35
```

Пример на языке Scheme (GNU Guile):

```
(define (f x)
  (lambda (y) (+ (* x x) y)))
```

```
scheme@(guile-user)> ((f 5) 10)
$1 = 35
```

Частичное применение функций:

Возможность зафиксировать часть аргументов многоместной функции и создать другую функцию, меньшей арности. Когда функция вызывается с меньшим количеством аргументов, чем она ожидает, и она возвращает функцию, принимающую остальные аргументы. Это называется частичным применением функций. Частично примененные функции не следует путать с каррированием.

Примеры на языке Python:

```
from functools import partial

# Обычная функция принимающая 3 аргумента:
def f(x, y, z):
    return (x + y + z) * 100

>>> f(2, 3, 4)
900


# Частичное применение этой же функции:
funcYZ = partial(f, 2) # x = 2

>>> funcYZ(3, 4)
900

>>> partial(f, 2)(3, 4)
900
```

Примеры на языке Scheme (GNU Guile):

```
(define (partial fun . args)
  (lambda x (apply fun (append args x))))


;; Обычная функция принимающая 3 аргумента:
(define (f x y z)
  (* (+ x y z) 100))


;; Частичное применение этой же функции:
(define (funcYZ y z)
  ((partial f 2) y z)) ;; x = 2


scheme@(guile-user)> (f 2 3 4)
$1 = 900
scheme@(guile-user)> (funcYZ 3 4)
$2 = 900
scheme@(guile-user)> ((partial f 2) 3 4)
$3 = 900
```

Хвостовая рекурсия и ее оптимизация:

Частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции. Подобный вид рекурсии примечателен тем, что может быть легко заменён на итерацию путём формальной и гарантированно корректной перестройки кода функции.

Чтобы понять хвостовую рекурсию нужно понимать, что такое хвостовой вызов. Хвостовой вызов — это вызов функции, который является последним действием в выполняемой функции.

Типовой механизм реализации вызова функции основан на сохранении адреса возврата, параметров и локальных переменных функции в стеке и выглядит следующим образом:

1. В точке вызова в стек помещаются параметры, передаваемые функции, и адрес возврата.
2. Вызываемая функция в ходе работы размещает в стеке собственные локальные переменные.
3. По завершении вычислений функция очищает стек от своих локальных переменных, записывает результат (обычно — в один из регистров процессора).
4. Команда возврата из функции считывает из стека адрес возврата и выполняет переход по этому адресу. Либо непосредственно перед, либо сразу после возврата из функции стек очищается от параметров.

Таким образом, при каждом рекурсивном вызове функции создаётся новый набор её параметров и локальных переменных, который вместе с адресом возврата размещается в стеке, что ограничивает максимальную глубину рекурсии объёмом стека. В чисто функциональных или декларативных (типа Пролога) языках, где рекурсия является единственным возможным способом организации повторяющихся вычислений, это ограничение становится крайне существенным, поскольку, фактически, превращается в ограничение на число итераций в любых циклических вычислениях, при превышении, которого будет происходить переполнение стека. Переполнение стека — компьютерная ошибка, когда в стеке вызовов храниться больше информации, чем он может вместить.

Нетрудно увидеть, что необходимость расширения стека при рекурсивных вызовах диктуется требованием восстановления состояния вызывающего экземпляра функции (то есть её параметров, локальных данных и адреса возврата) после возврата из рекурсивного вызова. Но если рекурсивный вызов является последней операцией перед выходом из вызывающей функции и результатом вызывающей функции должен стать результат, который вернёт рекурсивный вызов, сохранение контекста уже не имеет значения — ни параметры, ни локальные переменные уже использоваться не будут, а адрес возврата уже находится в стеке. Поэтому в такой ситуации вместо полноценного рекурсивного вызова функции можно просто заменить значения параметров в стеке и передать управление на точку входа. До тех пор, пока исполнение будет идти по этой рекурсивной ветви, будет, фактически, выполняться обычный цикл. Когда рекурсия завершится (то есть исполнение пройдёт по терминальной ветви и достигнет команды возврата из функции) возврат произойдёт сразу в исходную точку, откуда произошёл вызов рекурсивной функции. Таким образом, при любой глубине рекурсии стек переполнен не будет.

Примеры на языке Python:

```
# Функция с хвостовым вызовом:
def f(x):
  return tailCall(x * x) # Хвостовой вызов


# Функция вычисляющая факториал, используя хвостовую рекурсию:
# Промежуточные вычисления храняться в переменной `acc`.
def factorialTail(n, acc=1):
  if n == 0: return acc
  else: return factorialTail(n - 1, acc * n) # Хвостовой вызов

>>> factorialTail(5)
120
>>> factorialTail(10)
3628800


# Функция вычисляющая факториал, без хвостовой рекурсии:
def factorial(n):
```

```
  if n == 0: return 1
  else: return factorial(n-1) * n # Не считается хвостовым вызовом, так как еще есть операция умножения

>>> factorial(5)
120
>>> factorial(10)
3628800
```

Примеры на языке Scheme (GNU Guile):

```
;; Функция вычисляющая факториал, используя хвостовую рекурсию:
;; Промежуточные вычисления храняться в `acc`.
(define (factorial-tail n)
  (define (factorial-acc n acc)
    (if (zero? n)
  acc
  (factorial-acc (- n 1) (* acc n))))
  (factorial-acc n 1))

scheme@(guile-user)> (factorial-tail 5)
$1 = 120
scheme@(guile-user)> (factorial-tail 10)
$2 = 3628800


;; Функция вычисляющая факториал, без хвостовой рекурсии:
(define (factorial n)
  (if (= n 1) 1 (* (factorial(- n 1)) n))) ;; Не считается хвостовым вызовом, так как еще есть операция умножения

scheme@(guile-user)> (factorial 5)
$3 = 120
scheme@(guile-user)> (factorial 10)
$4 = 3628800


;; Трассировка выполнения вышепредставленных функций:
;; Можно видеть, что функция без хвостовой рекурсии занимает 10 вызовов в стеке.
scheme@(guile-user)> ,trace (factorial 10)
trace: (factorial 10)
trace: |  (factorial 9)
trace: |  |  (factorial 8)
trace: |  |  |  (factorial 7)
trace: |  |  |  |  (factorial 6)
trace: |  |  |  |  |  (factorial 5)
trace: |  |  |  |  |  |  (factorial 4)
trace: |  |  |  |  |  |  |  (factorial 3)
trace: |  |  |  |  |  |  |  |  (factorial 2)
trace: |  |  |  |  |  |  |  |  |  (factorial 1)
trace: |  |  |  |  |  |  |  |  |  1
trace: |  |  |  |  |  |  |  |  2
trace: |  |  |  |  |  |  |  6
trace: |  |  |  |  |  |  24
trace: |  |  |  |  |  120
trace: |  |  |  |  720
trace: |  |  |  5040
trace: |  |  40320
trace: |  362880
trace: 3628800
;; А функция с хвостовой рекурсией выполняется как один вызов.
scheme@(guile-user)> ,trace (factorial-tail 10)
trace: (factorial-tail 10)
trace: 3628800
```

Оптимизация хвостовых вызовов — это оптимизация, которая заменяет вызовы в хвостовых позициях переходами, что гарантирует выполнение циклов, реализованных с использованием рекурсии, в пространстве постоянного стека.

Это все замечательно, но в примере выше есть проблема, а именно то, что python не поддерживает оптимизацию хвостовых вызовов. Так как python скорее построен на идее итераций, чем на рекурсии. Но оптимизацию все же можно реализовать используя декораторы. Если язык программирования не

поддерживает оптимизацию хвостовых вызовов, то выполнить безопасно рекурсию невозможно. Так как большое количество вызовов приведет к переполнению стека, и программа завершит свою работу с ошибкой.

Языки программирования, которые поддерживают оптимизацию хвостовых вызовов:

- Scheme
- Haskell
- Ocaml
- Common Lisp
- Erlang
- Scala
- F#
- ...

Заключение:

Функциональное программирование — очень мощный и интересный подход к разработке программного обеспечения, который позволяет размять мозги и посмотреть по-новому на программирование в целом.

Больше примеров и заметок на эту тему, в моем репозитории на гитхабе.