

Паттерны программирования

Информатика
10-11 классы

24 января 2012 г.

Паттерны программирования

- Одним из отличительных свойств хорошего программиста является умение решать стандартные задачи стандартными методами.
- Практически любая задача в своём итоге сводится к нескольким проблемам:
 - 1 Считать исходные данные.
 - 2 Придумать способ хранения данных внутри программы.
 - 3 Понять, как проще всего разбить на подзадачи.
 - 4 Проверить, нет ли готовых решений?
 - 5 Преобразовать получившийся результат к требуемому.

Паттерн 1

Запомнить нужное значение

Запомнить нужное значение

- Часто в задаче требуется найти какую-то величину.
- К примеру, задача: найти наибольший элемент массива.
- Конечно, можно и нужно воспользоваться методом *max*, однако попробуем решить эту задачу без читов.
- В данном случае нам нужно найти максимальный элемент.
- Если что-то надо найти, нужна переменная, куда мы будем записывать значение найденного.
- В данной задаче назовём её *max*.

Алгоритм для паттерна 1

- 1 Пройтись по всему массиву и рассмотрим последовательно каждый его элемент.
- 2 В переменную *max* будем записывать максимальный элемент **на текущий момент**.
- 3 Если вдруг очередной элемент массива больше текущего максимального *max*, запишем его значение в переменную *max*.
- 4 Единственно, возникает вопрос: а какое число изначально записать в *max*?
- 5 Если мы сделаем $max = 0$, то программа будет неверно работать для массивов, состоящих из отрицательных чисел.

Правило 1

Думайте, всегда ли программа будет работать правильно. Из-за “забытых” случаев падают космические корабли.

Алгоритм для паттерна 1

- **Решение:** очевидно, что максимальный элемент массива больше либо равен нулевого элемента (так как он — максимальный).
- Поэтому если массив не пуст, то в качестве *значения по умолчанию* можно взять именно его.

Listing 1: Паттерн 1

```
def max(array)
  return false if array.empty?
  max = array[0]
  for i in 0..array.size-1
    max = array[i] if (array[i] > max)
  end
  max
end
```

Улучшения паттерна 1

- Заметим, что при каждой итерации цикла нам приходится вычислять значение длины массива `array.size`.
- Конечно, современные языки умеют *кэшировать* такие операции, но лучше не полагаться на это.
- **Задача.** Как сделать так, чтобы не вычислять длину массива на каждом шаге?
- Вычислить её единожды! А результат записать в дополнительную переменную!
- Да, кстати: почему `array.size-1`. Откуда взялась -1 ?
- Массивы в `ruby` нумеруются с нуля. Поэтому для массива, состоящего из n элементов, ключ последнего будет равен $n - 1$.

Улучшения для паттерна 1

Listing 2: Паттерн 1

```
def max(array)
  return false if array.empty?
  max = array[0]
  size = array.size - 1
  for i in 0..size
    max = array[i] if (array[i] > max)
  end
  max
end
```

Паттерн 2

Ключ и значение максимального элемента.

Ключ и значение максимального элемента

- Усложним чуть-чуть задачу.
- **Задача:** найти ключ и значение наибольшего элемента массива.
- Здесь уже стандартный метод *max* не поможет, так как он находит только значение, а не ключ.
- Конечно, есть и другие стандартные методы, но мы опять сделаем вручную.
- Нам нужно найти уже два числа: ключ и значение.
- В данной задаче назовём их *max_key* и *max_value*.

Изменения по сравнению с паттерном 1

- Первое изменение: если мы нашли элемент, который больше текущего максимального, перезаписать нужно не только значение `array[i]`, но и соответствующий значению ключ `i`.
- Второе изменение: функция должна возвращать не только максимальное значение, но и ключ.
- Самый простой способ вернуть несколько значений — через массив.
- Можно, конечно, вернуть и единичный хэш “ключ–значение”, в зависимости от общего стиля программы.

Программа для паттерна 2

Listing 3: Паттерн 2

```
def max(array)
  return false if array.empty?
  max_key = 0
  max_value = array[0]
  size = array.size - 1
  for i in 0..size
    if (array[i] > max_value)
      max_key = i
      max_value = array[i]
    end
  end
  [max_key, max_value]
end
```

Паттерн 3

Когда нужны булевские переменные.

Поиск отрицательного элемента

- Допустим, перед нами стоит задача узнать, есть ли в массиве отрицательный элемент.
- Конечно, мы бы могли воспользоваться “магическими” методами:
- Решим эту задачу без методов *find_all* и *elem*.
- В нашей задаче ответ бинарный: есть или нет. Если в задаче или подзадаче нужен такой ответ, значит, нужна булевская переменная.
- Её можно назвать *has_negative* или по старой традиции *flag* (аналог флажка, который либо опущен, либо поднят).

Listing 4: Паттерн 3

```
puts "Есть" if array.find_all{|elem| elem < 0}.any?
```

Алгоритм решения паттерна 3

- 1 Пройдёмся по всему массиву. Изначально *has_negative* присвоим ложь, так как ни одного отрицательного числа мы пока не нашли.
- 2 Проверим пробегаемый элемент, больше он или меньше нуля.
- 3 Если он меньше нуля, то “опустим флажок”, сделав *has_negative* равным истине и прервём цикл.
- 4 Итого, если в конце цикла *has_negative* имеет значение ИСТИНА, то как минимум один отрицательный элемент найден. Иначе — нет.
- 5 Заметим, что переменная *has_negative* как раз и будет отвечать на вопрос, есть ли в массиве отрицательный элемент. Поэтому функцией можно просто возвращать её значение.

Программа для паттерна 3

Listing 5: Паттерн 3

```
def array_has_negative(array)
  return false if array.empty?
  has_negative = false
  size = array.size - 1
  for i in 0..size
    if (array[i] < 0)
      has_negative = true
      break
    end
  end
  has_negative
end
```

Усложним задачу

- Усложним задачу. Допустим, у нас есть массив чисел, содержащий элементы от 1 до 100. Сколько — неизвестно. Повторы также возможны.
- Требуется вывести на экран те натуральные числа от 1 до 100, которые не встречаются в данном массиве.
- По сути, нам нужно ответить на 100 вопросов: есть ли в массиве число 1, есть ли в массиве число 2 и т.п.
- **Решение перебором в лоб:** пройтись циклом от 1 до 100 и проверить, есть ли в массиве пробегаемое число.
- **Недостаток:** ооочень долгое время работы. Нам придётся совершить $n \cdot 100$ (по 100 проходов для каждого из n чисел) итераций.
- Можно улучшить метод, заведя булевский массив длиной 100. И, пробегая всего лишь один раз по всему массиву

Усложним задачу

- Можно улучшить метод, заведя булевский массив длиной 100. Заполним его изначально ложью.
- Пройдём по начальному массиву.
- Будем помечать истиной элементы с такими ключами, которые встречаются в виде значений в массиве.
- Дубликаты нам не страшны, так как двойное присваивание ничего не поменяет.
- Кстати, от них можно избавиться с помощью метода *arr.uniq*.
- В конце просто пройдёмся по булевскому массиву и выведем на экран ключи тех элементов, которые равны лжи.

Программа для усложнённого паттерна 3

Listing 6: Усложнённый паттерн 3

```
def missen_numbers(array)
  return (1..100).to_a if array.empty?
  size = array.size-1
  has_numbers = []
  100.times { |i| has_numbers[i] = false }
  for i in 0..size
    has_numbers[array[i]-1] = true
  end
  numbers = []
  100.times { |i| numbers[] = i+1 unless has_numbers[i] }
  numbers
end
```

Правило 2

Никогда не забывайте, что элементы массива нумеруются с нуля.

Правило 3

Обязательно инициализируйте пустые массивы
нужной длины.

Необходимые пояснения

- Почему мы присваиваем истине `has_numbers[array[i]-1]`. Откуда минус единица?
- Известно, что значения массива — это числа от 1 до 100. У нас массив состоит из 100 элементов.
- То есть, ключи определены от 0 до 99. Вычитая единицу, мы “переводим” одно представление в другое.
- По аналогичным соображениям в последнем цикле `times` мы прибавляем к ключу единицу (обратная операция).
- По правилу 3: когда вы пишете конструкцию вида `arr[i] = arr[i] + 1`, предполагается, что i -ый элемент массива существует и определён. Если это не так, вы получите ошибку.

References

- Все презентации доступны на <http://school.smirik.ru!>
- Вопросы, предложения, д/з: smirik@gmail.com