Домой Блог Контакты Печать Править

Программирование скриптов в Vim. Часть 7. Словари и объектно-ориентированное программирование

Предыдущие части

Часть 1. Запуск скриптов
Часть 2. Переменные
Часть 3. Работа со списками
Часть 4. Работа со строками
Часть 5. Операции ветвления и функции
Часть 6. Продвинутое использование функций

Оглавление

Введение

Нам осталось рассмотреть последний тип данных, существующий в Vim. Это словари, которые представляют собой хранилище данных, доступ к которым осуществляется по строковому ключу. Подобные типы данных в других языках программирования называют или картами (map), или ассоциативными массивами. Важное свойство словарей

Студентам

Фото

- Животные
- Черно-белые
- Пейзажи/Природа
- Город
- Закаты
- Панорамы
- Спорт
- Репортаж
- Разное

Контакты

заключается в том, что хранящиеся в нем данные не упорядочены, как в списке, поэтому доступ к элементам невозможен по порядковому номеру, а возможен исключительно по ключу.

Словари в Vim очень похожи на одноименный тип данных Python, основное отличие заключается в том, что в Vim в качестве ключа может выступать только строка, но в качестве хранимых данных могут выступать любые типы данных, в том числе и другие словари.

С помощью словарей в Vim можно организовать и объектно-ориентированный подход к программированию скриптов, о чем мы тоже поговорим.

Создание словарей

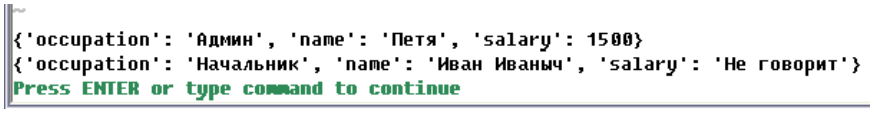
Для начала разберемся с тем как создаются словари, а создавать их можно двумя способами. Первый способ заключается в том, что при создании словаря сразу указывается некоторое количество пар ключ-значение (ключ-данные), которые сразу же окажутся в словаре. Синтаксис такого подхода очень напоминает Python. Давайте создадим несколько словарей, каждый из которых будет содержать различную информацию о работниках: имя (поле 'name'), должность (поле 'occupation') и зарплата (поле 'salary'):

```
let worker1 = {'name': 'Петя', 'occupation': 'Админ',
'salary': 1500}
let worker2 = {'name': 'Иван Иваныч', 'occupation':
'Начальник', 'salary': 'Не говорит'}

echo worker1
echo worker2
```

Исходник

Как Vim выводит словари вы можете увидеть на следующем скриншоте.



Такое применение словарей уже напоминает объектно-ориентированный подход, поэтому в дальнейших примерах мы будем придерживаться такой же структуры словарей. Как видите, в одном словаре в качестве данных могут

выступать разные типы данных (в данном случае - строки и целые числа).

Второй подход к созданию словарей состоит в том, чтобы сначала создать пустой словарь или словарь с начальными элементами, а потом уже его дополнить нужными данными.

```
let worker = {}
let worker['name'] = 'Петя'
let worker['occupation'] = 'Админ'
let worker['salary'] = 1500

echo worker
```

Исходник

Такая запись встречается во многих языках программирования, и думаю, что какие-то пояснения здесь не нужны. Но хочется обратить ваше внимание на порядок отображения этого словаря в Vim командой 'echo':

```
{'occupation': 'Админ', 'name': 'Петя', 'salary': 1500}
```

Здесь уже порядок вывода полей не совпадает с тем в какой последовательности мы добавляли их, что еще раз доказывает то, что на этот порядок нельзя рассчитывать - он может быть любым.

Выше говорилось, что в качестве ключа может выступать только строка, это верно, но при написании скриптов мы можем использовать в качестве ключа и целые числа, но они будут автоматически преобразованы в строки. Для того, чтобы в этом убедиться, немного изменим предыдущий пример:

```
let worker = {}
let worker[10] = 'Петя'
let worker[20] = 'Админ'
let worker[30] = 1500

echo worker
```

Исходник

В результате на экране мы увидим следующую строку:

```
{'10': 'Петя', 'occupation': '20', '30': 1500}
```

Таким образом словари можно использовать в качестве разреженных массивов, когда у элементов массива используются большие индексы, но самих элементов не так

много, и большая часть массива остается пустой. В этом случае не надо заводить действительно большой массив (список), а достаточно использовать целые числа в качестве индексов, в результате будет казаться, что мы работаем с огромным массивом.

У второго способа создания (заполнения) словарей есть еще сокращенная форма, которая по записи еще ближе нас подводит к объектно-ориентированному программированию. Эта запись выглядит следующим образом:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

echo worker
```

Исходник

Эти два примера равносильны.

А теперь вспомните, что у нас в арсенале есть такой замечательный тип данных, как указатели на функции, которые мы тоже можем хранить в словаре, таким образом мы получим и инкапсуляцию (включение) членов, и наличие методов. У таких функций будут дополнительные возможности для работы со словарями, куда они входят, но об этом речь пойдет чуть позже, в последующих разделах.

Доступ к элементам словарей

Итак, создавать словари мы уже умеем, теперь рассмотрим операции, которые мы можем с ними проделывать. Начнем с извлечения данных по ключу, это одна из самых часто используемых операций, ведь мы же не просто так сохраняем данные, надо научиться их и читать. Извлечение данных по синтаксису ничем не отличается от заполнения. Сразу рассмотрим пример:

```
let worker = {}
let worker['name'] = 'Петя'
let worker['occupation'] = 'Админ'
let worker['salary'] = 1500

echo worker['name']
echo worker['occupation']
echo worker['salary']
```

Исходник

Или то же самое в объектно-ориентированном стиле:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

echo worker.name
echo worker.occupation
echo worker.salary
```

Исходник

При работе со словарями иногда необходимо перебирать все хранящиеся в нем элементы, чтобы их как-то обработать. Для подобных операций в Vim существует несколько функций.

- *keys()* - возвращает список всех ключей в словаре.
- *values()* - возвращает список всех хранящихся в словаре значений (данные).
- *items()* - возвращает список списков. Каждый элемент этого списка, содержит список из двух элементов: ключ и соответствующее значение из словаря.

Давайте для начала посмотрим что вернут эти функции для нашего последнего примера.

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

echo keys (worker)
echo values (worker)
echo items (worker)
```

Исходник

Результат работы этого скрипта показан на следующем скриншоте.

```
['occupation', 'name', 'salary']
['Админ', 'Петя', 1500]
[['occupation', 'Админ'], ['name', 'Петя'], ['salary', 1500]]
Press ENTER or type command to continue
```

Рассмотрим наиболее типичные применения этих функций:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

for key in keys(worker)
    echo key "-" worker[key]
endfor
```

Исходник

Здесь мы получаем список ключей и, перебирая их в цикле *for*, получаем и выводим также соответствующее ключу значение.

```
~
occupation - Админ
name - Петя
salary - 1500
Press ENTER or type command to continue
```

Чтобы на каждой итерации цикла не надо было бы вручную получать соответствующие значения, можем воспользоваться функцией *items()* и получать поочередно пары ключ-значение одновременно:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

for [key, value] in items(worker)
    echo key "-" value
endfor
```

Исходник

Также есть полезная функция *has_key()*, которая принимает словарь и строковое значение и возвращает 1 в случае, если в словаре есть элемент под соответствующем ключом:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

echo has_key (worker, 'salary')
echo has_key (worker, 'age')
```

Исходник

В результате мы увидим:

```
~
1
0
Press ENTER or type command to continue
```

Удаление элементов из словарей

Теперь мы уже умеем создавать, заполнять словарь содержимым, а также извлекать из него данные. Теперь осталось научиться удалять из словаря элементы, ставшие ненужными. Для этого есть два подхода. Первый - использовать команду *unlet*, уже используемую нами при

удалении переменных. Второй способ - это воспользоваться функцией *remove()*.

Рассмотрим эти подходы по очереди.

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

unlet worker['salary']

echo worker
```

Исходник

В результате на экран будет выведена надпись:

```
{'occupation': 'Админ', 'name': 'Петя'}
```

Ту же операцию мы можем переписать в объектно-ориентированном стиле:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

unlet worker.salary

echo worker
```

Исходник

Теперь воспользуемся функцией *remove()*, она принимает в качестве параметров словарь и строку, являющуюся ключом к элементу, который должен быть удален. Сама функция *remove()* возвращает значение, которое хранилось по указанному ключу. Это продемонстрировано в следующем скрипте.

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

echo remove(worker, 'salary')

echo worker
```

Исходник

Результат работы этого скрипта показан на следующем скриншоте.

```
1500
{'occupation': 'Админ', 'name': 'Петя'}
Press ENTER or type command to continue
```

Также может быть полезна функция *empty()*, которая принимает в качестве параметра словарь (или список) и возвращает 1, если словарь (или список) пуст, и 0 в противном случае. Как и для списков, чтобы узнать количество элементов в словаре, можно воспользоваться функцией *len()*.

Глубокие копии

Мы уже обсуждали вопросы копирования в третьей части, когда разбирались со списками. Все сказанное там в полной мере относится и к словарям, поэтому лишь кратко вспомним в чем состоит проблема. Дело в том, что при присвоении одной переменной значения другой переменной, которая является словарем, мы, по сути, присваиваем только указатель на словарь. При этом сам словарь остается в единственном экземпляре, и его могут изменить обе переменные.

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

let worker2 = worker

echo worker
echo worker2

let worker2.salary = 2000

echo worker
echo worker2
```

Исходник

В результате мы увидим:

```
{'occupation': 'Админ', 'name': 'Петя', 'salary': 1500}
{'occupation': 'Админ', 'name': 'Петя', 'salary': 1500}
{'occupation': 'Админ', 'name': 'Петя', 'salary': 2000}
{'occupation': 'Админ', 'name': 'Петя', 'salary': 2000}
Press ENTER or type command to continue
```

Чтобы создать обычную (не глубокую) копию словаря можно воспользоваться функцией *copy()*:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

let worker2 = copy (worker)

echo worker
echo worker2

let worker2.salary = 2000

echo worker
echo worker2
```

Исходник

В данном случае это помогло:

```
{'occupation': 'Админ', 'name': 'Петя', 'salary': 1500}
{'occupation': 'Админ', 'name': 'Петя', 'salary': 1500}
{'occupation': 'Админ', 'name': 'Петя', 'salary': 1500}
{'occupation': 'Админ', 'name': 'Петя', 'salary': 2000}
Press ENTER or type command to continue
```

Однако, если в словаре будет храниться другой словарь или список, то в копию словаря попадет не заново скопированный словарь/список, а лишь указатель на него, в результате внутри обоих словарей будет храниться указатель на один и тот же внутренний словарь/список:

```
let foo = {'key1': 'val1', 'key2': {'bar': 'val_bar'} }

echo foo

let spam = copy (foo)
let spam.key2.bar = 'new_val'

echo foo
echo spam
```

Исходник

```
{'key1': 'val1', 'key2': {'bar': 'val_bar'}}
{'key1': 'val1', 'key2': {'bar': 'new_val'}}
{'key1': 'val1', 'key2': {'bar': 'new_val'}}
Press ENTER or type command to continue
```

Чтобы указать интерпретатору, что нужно копировать и внутренние списки/словари, нужно воспользоваться функцией *deepcopy()*:

```
let foo = {'key1': 'val1', 'key2': {'bar': 'val_bar'} }

echo foo

let spam = deepcopy (foo)
let spam.key2.bar = 'new_val'
```

```
echo foo
echo spam
```

Исходник

В результате копирования создается так называемая "глубокая копия", в которой все внутренние структуры скопированы рекурсивно (максимум до 100 уровня вложенности):

```
{'key1': 'val1', 'key2': {'bar': 'val_bar'}}
{'key1': 'val1', 'key2': {'bar': 'val_bar'}}
{'key1': 'val1', 'key2': {'bar': 'new_val'}}
Press ENTER or type command to continue
```

Объектно-ориентированное программирование в Vim

Итак, для доступа к элементам словаря в Vim можно использовать запись, напоминающую объектно-ориентированное программирование (ООП), что мы уже делали. Но ведь словари могут хранить и указатели на функции, что еще ближе нас подводит к ООП. Мы можем написать следующий код:

```
function! s:hello()
    echo 'Привет!'
endfunction

let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

let worker.hello = function ('s:hello')

call worker.hello()
```

Исходник

В результате на экране мы увидим приветствие:

```
Привет!
```

Но такой вызов функции хоть и может полезен тем, что функция и объект-словарь становятся связанными, но от ООП мы обычно ожидаем большего. В данном примере основная проблема состоит в том, что связь функции и объекта односторонняя: объект имеет доступ к функции, а функция к своему владельцу - нет, поэтому функция не может обращаться к членам объекта. А нам, допустим, хотелось бы, чтобы эта функция не просто писала

приветствие, но и обращалась к объекту по имени через член *name*.

Мы можем в явном виде передать в функцию наш словарь:

```
function! s:hello(self)
    echo printf ('Привет, %s!', a:self.name)
endfunction

let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

let worker.hello = function ('s:hello')

call worker.hello(worker)
```

Исходник

Теперь скрипт выведет строку:

```
Привет, Петя!
```

Но согласитесь, что такое решение не особо красивое, мы вынуждены дважды указывать объект *worker* при вызове функции. Было бы здорово, если бы параметр *self* передавался неявно, как это сделано с одноименным параметром в Python или *this* в C++. И разработчики Vim дали такую возможность, для этого достаточно после объявления функции добавить ключевое слово *dict*, тогда в функцию неявно будет передаваться указатель на объект-словарь, из которого функция была вызвана. Имя параметра так и останется *self*, с той лишь разницей, что перед именем этой переменной не нужно ставить префикс *a:*, показывающий, что это аргумент функции.

Теперь предыдущий пример у нас будет выглядеть следующим образом:

```
function! s:hello() dict
    echo printf ('Привет, %s!', self.name)
endfunction

let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

let worker.hello = function ('s:hello')

call worker.hello()
```

Исходник

Кроме того, существует возможность еще немного сократить запись, указав имя объекта при объявлении функции:

```
let worker = {}
let worker.name = 'Петя'
let worker.occupation = 'Админ'
let worker.salary = 1500

function! worker.hello() dict
    echo printf ('Привет, %s!', self.name)
endfunction

call worker.hello()
```

Исходник

Здесь мы использовали запись *function! worker.hello() dict*, что позволило избавиться от строки *let worker.hello = function ('s:hello')*.

Теперь мы можем создавать полноценные объекты с членами и методами.

На этот раз не будет какого-либо жизненного примера, поскольку мне не хватило фантазии придумать какой-нибудь короткий пример, где бы использование словарей не было бы притянуто за уши, а использование объектно-ориентированный подход чаще всего оправдывает себя в сравнительно крупных скриптах. Но из-за этого расстраиваться не стоит, теперь мы разобрали все типы данных, существующие в Vim, поэтому следующая часть статьи будет посвящена полностью написанию плагинов.

Часть 8. Более подробно о плагинах

Вы можете подписаться на новости сайта через RSS, Группу Вконтакте или Канал в Telegram.

★★★★★

Рейтинг 4.8/5. Всего 17 голос(а, ов)

○Плохо ○Так себе ○Неплохо ○Хорошо ○Отлично

Голосовать

Алексей 29.01.2012 - 17:26

Спасибо за статью

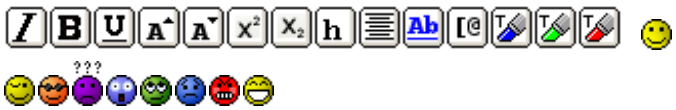
Спасибо, возьму на заметку

Подписаться на комментарии

Автор:

Тема:

Ваш комментарий



Введите код
983

Послать