Работа с массивами в bash

<https://opensource.com/article/18/5/you-dont-know-bash-intro-bash-arrays>

* [Блог компании RUVDS.com](https://habr.com/company/ruvds/),
* [Разработка веб-сайтов](https://habr.com/hub/webdev/),
* [Разработка под Linux](https://habr.com/hub/linux_dev/)
* Перевод

Программисты регулярно пользуются bash для решения множества задач, сопутствующих разработке ПО. При этом bash-массивы нередко считаются одной из самых непонятных возможностей этой командной оболочки (вероятно, массивы уступают в этом плане лишь регулярным выражениям). Автор материала, перевод которого мы сегодня публикуем, приглашает всех желающих в удивительный мир bash-массивов, которые, если привыкнуть к их необычному синтаксису, могут принести немало пользы.

[[](https://habr.com/company/ruvds/blog/413725/)](https://habr.com/company/ruvds/blog/413725/)

Реальная задача, в которой пригодятся bash-массивы

Писать о bash — занятие неоднозначное. Дело в том, что статьи о bash нередко превращаются в руководства пользователя, которые посвящены рассказам о синтаксических особенностях рассматриваемых команд. Эта статья написана иначе, надеемся, вам она не покажется очередным «руководством пользователя».  
  
Учитывая вышесказанное, представим себе реальный сценарий использования массивов в bash. Предположим, перед вами стоит задача оценить и оптимизировать утилиту из нового внутреннего набора инструментов, используемого в вашей компании. На первом шаге этого исследования вам нужно испытать её с разными наборами параметров. Испытание направлено на изучение того, как новый набор инструментов ведёт себя при использовании им разного количества потоков. Для простоты изложения будем считать, что «набор инструментов» — это скомпилированный из C++-кода «чёрный ящик». При его использовании единственным параметром, на который мы можем влиять, является число потоков, зарезервированных для обработки данных. Вызов исследуемой системы из командной строки выглядит так:

./pipeline *--threads 4*

Основы

В первую очередь объявим массив, содержащий значения параметра --threads, с которыми мы хотим протестировать систему. Выглядит этот массив так:

allThreads=(1 2 4 8 16 32 64 128)

В этом примере все элементы являются числами, но, на самом деле, в bash-массивах можно хранить одновременно и числа, и строки. Например, вполне допустимо объявление такого массива:

myArray=(1 2 "three" 4 "five")

Как и в случае с другими переменными bash, обратите внимание на то, чтобы вокруг знака = не было бы пробелов. В противном случае bash сочтёт имя переменной именем программы, которую ему нужно выполнить, а = — её первым аргументом!  
  
Теперь, когда мы инициализировали массив, давайте извлечём из него несколько элементов. Тут можно заметить, например, что команда echo $allThreads выведет лишь первый элемент массива.  
  
Для того чтобы понять причины такого поведения, немного отвлечёмся от массивов и вспомним, как работать с переменными в bash. Рассмотрим следующий пример:

type="article"

echo "Found 42 $type"

Предположим, что имеется переменная $type, которая содержит строку, представляющую собой имя существительное. После этого слова надо добавить букву s. Однако нельзя просто добавить эту букву в конец имени переменной, так как это превратит команду обращения к переменной в $types, то есть, работать мы уже будем с совершенно другой переменной. В данной ситуации можно воспользоваться конструкцией вида echo "Found 42 "$type"s". Но лучше всего решить эту задачу с использованием фигурных скобок: echo "Found 42 ${type}s", что позволит нам сообщить bash о том, где начинается и заканчивается имя переменной (что интересно, тот же синтаксис используется в JavaScript ES6 для внедрения переменных в выражения в [шаблонных строках](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)).  
  
Теперь вернёмся к массивам. Оказывается, что, хотя фигурные скобки при работе с переменными обычно не нужны, они нужны для работы с массивами. Они позволяют задавать индексы для доступа к элементам массива. Например, команда вида echo ${allThreads[1]} выведет второй элемент массива. Если в вышеописанной конструкции забыть о фигурных скобках, bash будет воспринимать [1]как строку и соответствующим образом обработает то, что получится.  
  
Как видите, массивы в bash имеют странный синтаксис, но в них, по крайней мере, нумерация элементов начинается с нуля. Это роднит их с массивами из многих других языков программирования.

Способы обращения к элементам массивов

В вышеописанном примере мы использовали в массивах целочисленные индексы, задаваемые в явном виде. Теперь рассмотрим ещё два способа работы с массивами.  
  
Первый способ применим в том случае, если нам нужен $i-й элемент массива, где $i — это переменная, содержащая индекс нужного элемента массива. Извлечь этот элемент из массива можно с помощью конструкции вида echo ${allThreads[$i]}.  
  
Второй способ позволяет вывести все элементы массива. Он заключается в замене числового индекса символом @ (его можно воспринимать как команду, указывающую на все элементы массива). Выглядит это так: echo ${allThreads[@]}.

Перебор элементов массивов в циклах

Вышеописанные принципы работы с элементами массивов пригодятся нам для решения задачи перебора элементов массива. В нашем случае это означает запуск исследуемой команды pipeline с каждым из значений, которое символизирует число потоков и хранится в массиве. Выглядит это так:

for t in ${allThreads[@]}; do

 ./pipeline --threads $t

done

Перебор индексов массивов в циклах

Рассмотрим теперь несколько иной подход к перебору массивов. Вместо того, чтобы перебирать элементы, мы можем перебирать индексы массива:

for i in ${!allThreads[@]}; do

 ./pipeline --threads ${allThreads[$i]}

done

Разберём то, что здесь происходит. Как мы уже видели, конструкция вида ${allThreads[@]}представляет собой все элементы массива. При добавлении сюда восклицательного знака мы превращаем эту конструкцию в ${!allThreads[@]}, что приводит к тому, что она возвращает индексы массива (от 0 до 7 в нашем случае).  
  
Другими словами, цикл for перебирает все индексы массива, представленные в виде переменной $i, а в теле цикла обращение к элементам массива, которые служат значениями параметра --thread, выполняется с помощью конструкции ${allThreads[$i]}.  
  
Читать этот код сложнее, чем тот, что приведён в предыдущем примере. Поэтому возникает вопрос о том, к чему все эти сложности. А нужно это нам из-за того, что в некоторых ситуациях, при обработке массивов в циклах, нужно знать и индексы и значения элементов. Скажем, если первый элемент массива нужно пропустить, перебор индексов избавит нас, например, от необходимости создания дополнительной переменной и от инкрементации её в цикле для работы с элементами массива.

Заполнение массивов

До сих пор мы исследовали систему, вызывая команду pipeline с передачей ей каждого интересующего нас значения параметра --threads. Теперь предположим, что эта команда выдаёт длительность выполнения некоего процесса в секундах. Нам хотелось бы перехватить возвращаемые ей на каждой итерации данные и сохранить в другом массиве. Это даст нам возможность работать с сохранёнными данными после того, как все испытания закончатся.

Полезные синтаксические конструкции

Прежде чем говорить о том, как добавлять данные в массивы, рассмотрим некоторые полезные синтаксические конструкции. Для начала нам нужен механизм получения данных, выводимых bash-командами. Для того чтобы захватить вывод команды, нужно использовать следующую конструкцию:

output=$( ./my\_script.sh )

После выполнения этой команды то, что выведет скрипт myscript.sh, будет сохранено в переменной $output.  
  
Вторая конструкция, которая нам очень скоро пригодится, позволяет присоединять к массивам новые данные. Выглядит это так:

myArray+=( "newElement1" "newElement2" )

Решение задачи

Теперь, если собрать вместе всё то, что мы только что изучили, можно будет создать скрипт для тестирования системы, который выполняет команду с каждым из значений параметра из массива и сохраняет в другом массиве то, что выводит эта команда.

allThreads=(1 2 4 8 16 32 64 128)

allRuntimes=()

for t in ${allThreads[@]}; do

 runtime=$(./pipeline --threads $t)

 allRuntimes+=( $runtime )

done

Что дальше?

Только что мы рассмотрели способ использования bash-массивов для перебора параметров, используемых при запуске некоей программы и для сохранения данных, которые возвращает эта программа. Однако этим сценарием варианты использования массивов не ограничиваются. Вот ещё пара примеров.

Оповещения о проблемах

В этом сценарии мы рассмотрим приложение, которое разбито на модули. У каждого из этих модулей имеется собственный лог-файл. Мы можем написать скрипт задания cron, который, при обнаружении проблем в соответствующем лог-файле, будет оповещать по электронной почте того, кто ответственен за каждый из модулей:

*# Списки лог-файлов и заинтересованных лиц*

logPaths=("api.log" "auth.log" "jenkins.log" "data.log")

logEmails=("jay@email" "emma@email" "jon@email" "sophia@email")

*# Проверяем логи на предмет наличия сообщений об ошибках*

for i in ${!logPaths[@]};

do

 log=${logPaths[$i]}

 stakeholder=${logEmails[$i]}

 numErrors=$( tail -n 100 "$log" | grep "ERROR" | wc -l )

*# Оповещаем заинтересованных лиц при обнаружении более 5 ошибок*

 if [[ "$numErrors" -gt 5 ]];

 then

   emailRecipient="$stakeholder"

   emailSubject="WARNING: ${log} showing unusual levels of errors"

   emailBody="${numErrors} errors found in log ${log}"

   echo "$emailBody" | mailx -s "$emailSubject" "$emailRecipient"

 fi

done

Запросы к API

Предположим, вы хотите собрать сведения о том, какие пользователи комментируют ваши публикации на Medium. Так как у нас нет прямого доступа к базе данных этой площадки, SQL-запросы обсуждать мы не будем. Однако, для доступа к данным такого рода можно использовать различные API.  
  
Для того чтобы избежать долгих разговоров об аутентификации и токенах, будем, в качестве конечной точки, использовать общедоступное API сервиса [JSONPlaceholder](https://github.com/typicode/jsonplaceholder), ориентированного на тестирование. Получив от сервиса публикацию и вытащив из её кода данные по электронным адресам комментаторов, мы можем поместить эти данные в массив:

endpoint="https://jsonplaceholder.typicode.com/comments"

allEmails=()

*# Запрашиваем первые 10 публикаций*

for postId in {1..10};

do

*# Выполняем обращение к API для получения электронных адресов комментаторов публикации*

 response=$(curl "${endpoint}?postId=${postId}")

*# Используем jq для парсинга JSON и записываем в массив адреса комментаторов*

 allEmails+=( $( jq '.[].email' <<< "$response" ) )

done

Обратите внимание на то, что здесь использовано средство [jq](https://stedolan.github.io/jq/), которое позволяет парсить JSON в командной строке. В подробности работы с jq мы тут вдаваться не будем, если вам этот инструмент интересен — посмотрите документацию по нему.

Bash или Python?

Массивы — возможность полезная и доступна она не только в bash. У того, кто пишет скрипты для командной строки, может возникнуть закономерный вопрос о том, в каких ситуациях стоит использовать bash, а в каких, например, Python.  
  
На мой взгляд, ответ на этот вопрос кроется в том, насколько программист зависит от той или иной технологии. Скажем, если задачу можно решить прямо в командной строке, тогда ничто не препятствует использованию bash. Однако в том случае, если, например, интересующий вас скрипт является частью некоего проекта, написанного на Python, вы вполне можете воспользоваться Python.  
  
Например, для решения рассмотренной здесь задачи можно воспользоваться и скриптом, написанным на Python, однако, это сведётся к написанию на Python обёртки для bash:

import subprocess

all\_threads = [1, 2, 4, 8, 16, 32, 64, 128]

all\_runtimes = []

# Запускаем программу с передачей ей различного числа потоков

for t in all\_threads:

 cmd = './pipeline --threads {}'.format(t)

 # Используем модуль subprocess для получения того, что возвращает программа

 p = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)

 output = p.communicate()[0]

 all\_runtimes.append(output)

Пожалуй, решение этой задачи с помощью bash, без привлечения других технологий, получается короче и понятнее и здесь вполне можно обойтись без Python.

Итоги

В этом материале мы разобрали немало конструкций, использующихся для работы с массивами. Вот таблица, в которой вы найдёте то, что мы рассмотрели, и кое-что новое.

|  |  |
| --- | --- |
| Синтаксическая конструкция | Описание |
| arr=() | Создание пустого массива |
| arr=(1 2 3) | Инициализация массива |
| ${arr[2]} | Получение третьего элемента массива |
| ${arr[@]} | Получение всех элементов массива |
| ${!arr[@]} | Получение индексов массива |
| ${#arr[@]} | Вычисление размера массива |
| arr[0]=3 | Перезапись первого элемента массива |
| arr+=(4) | Присоединение к массиву значения |
| str=$(ls) | Сохранение вывода команды ls в виде строки |
| arr=( $(ls) ) | Сохранение вывода команды ls в виде массива имён файлов |
| ${arr[@]:s:n} | Получение элементов массива начиная с элемента с индексом s до элемента с индексом s+(n-1) |

На первый взгляд bash-массивы могут показаться довольно странными, но те возможности, которые они дают, стоят того, чтобы с этими странностями разобраться. Полагаем, что освоив bash-массивы, вы будете пользоваться ими довольно часто. Несложно представить себе бесчисленное множество сценариев, в которых эти массивы могут пригодиться.  
  
**Уважаемые читатели!** Если у вас есть интересные примеры применения массивов в bash-скриптах — просим ими поделиться.  
  
[https://habrastorage.org/files/1ba/550/d25/1ba550d25e8846ce8805de564da6aa63.png](https://ruvds.com/ru-rub/#order)

**Теги:**

* [Bash](https://habr.com/search/?q=%5BBash%5D&target_type=posts)
* [разработка](https://habr.com/search/?q=%5B%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0%5D&target_type=posts)