Визитка

ДЕНИС СИЛАКОВ, кандидат ф.-м. н., ЗАО «РОСА», член рабочей группы LSB, старший архитектор, занимается автоматизацией разработки ОС «РОСА», denis.silakov@rosalab.ru

Shebang – уроки истории

Все знают, что скрипты в UNIX принято начинать с символов #!, но не все догадываются, почему. Оказывается, это довольно поучительная история!

Известно, что учиться лучше всего на ошибках, причем на чужих. Но изучение удачных решений тоже очень полезно для повышения квалификации. Для разработчиков ИТ-систем хорошие примеры обоих видов предоставляет история shebang – такое имя получило сочетание символов решетки и восклицательного знака (#!), с которого начинаются скрипты в UNIX-подобных ОС.

Предыстория и постановка задачи

С точки зрения пользователя, одна из основных задач ОС – запускать и выполнять различные приложения. Однако что значит «запустить программу»? Для пользователей современных сред с графическим интерфейсом это означает кликнуть на иконку программы или выбрать соответствующий пункт в меню «Пуск» либо его аналогах. Во времена, когда возникла наша проблема, графический интерфейс пользователя оставался делом светлого будущего, а программы запускались из командной строки. При таком способе работы пользователь взаимодействует с интерактивной командной оболочкой – специальной программой, которая принимает вводимые с клавиатуры команды и выполняет их – либо самостоятельно, либо с помощью ядра ОС и других программ. Для запуска программы необходимо ввести имя запускаемого файла и нажать <Enter>.

Но все это лишь верхушка айсберга. Давайте посмотрим, что происходит «под капотом» (в оболочке, системных библиотеках и ядре ОС) после клика на иконку программы или ввода ее имени в командной строке.

В классическом случае двоичных скомпилированных программ процесс запуска подразумевает загрузку кода программы и ее данных в оперативную память и передачу управления в точку входа программы. Всем этим занимается ядро ОС, а для удобства взаимодействия с ядром часто предоставляются дополнительные системные библиотеки. Ядра и библиотеки большинства ОС (и нынешних, и сорокалетней давности) для запуска программ предоставляют набор системных вызовов exec(), которым в качестве аргумента передается имя файла программы. Эти функции уже много лет определяются стандартом POSIX.

Однако программы бывают не только бинарные. С первых версий UNIX оболочка Bourne Shell, с которой взаимодействовали операторы ЭВМ, предоставляла возможность создания скриптов на специальном языке. Язык этот так и назвали – язык Shell. Подобная функциональность стала популярна (и остается таковой по сей день) для решения повседневных рутинных задач как программистов, так и администраторов. Как оказалось, справиться со многими задачами можно скриптом в несколько строчек на языке Shell, причем скрипт не надо перекомпилировать при каждом изменении и держать для этого в системе компилятор.

Изначально для запуска скрипта с помощью интерпретатора необходимо было явно вызвать этот интерпретатор, передав ему имя скрипта в качестве аргумента. Например, в случае Shell, интерпретатор – это программа /bin/sh. Но можно указать просто sh, поскольку директория /bin традиционно включена в переменную PATH и просматривается при попытке определить местоположение запрошенной программы:

```
$ sh ./script.sh
```

Однако пользователи и администраторы – народ ленивый, а лень, как известно, – двигатель прогресса. И вполне естественно, что возникло желание запускать Shell-скрипты как обычные программы, то есть просто вводя их имя и нажимая <Enter>.

Первая реализация

Первая реализация такой возможности не заставила себя долго ждать и была достаточно примитивной – оболочка Bourne Shell в первую очередь пыталась «скормить» запускаемый файл ядру ОС. Если же ядро возвращало ошибку NOEXEC, означающую, что ему не удалось распознать формат файла, то оболочка пыталась передать файл интерпретатору sh, например, с помощью вызова execl(), который в случае успеха замещает исходный процесс новым:

```
execl(program, basename(program), (char *)0);
```

```
if (errno == ENOEXEC)
    execl ("/bin/sh", "sh", "-c", program, (char *)0);

perror(program);
return -1;
```

На первых порах такой реализации было достаточно. Но по мере развития UNIX и компонентов ОС выяснилось, что такой прямолинейный подход хотя и прост в реализации, но не отличается масштабируемостью и универсальностью.

Во-первых, со временем скрипты стало возможно писать не только на языке Shell. Постепенно появлялись как другие оболочки со своими языками (например, C Shell), так и другие интерпретируемые языки (например, AWK, созданный там же, где и UNIX, – в AT&T Bell Laboratories).

Во-вторых, с точки зрения ОС, интерпретируемые программы все-таки во многом отличались от программ бинарных. В частности, если вы хотели запустить скрипт из другой программы через вызов exec(), то в качестве имени файла ему необходимо было передавать имя интерпретатора, а имя скрипта уже передавалось в роли аргумента командной строки. Если же необходимо было с помощью команды ps посмотреть, какие программы работают, то получали множество экземпляров интерпретатора /bin/sh, а имена реально запущенных скриптов прятались в его аргументах.

Итак, возникла необходимость в более гибком и универсальном решении.

Продвинутая реализация

Наиболее логичный способ разрешить перечисленные выше проблемы был предложен Деннисом Ритчи (одним из создателей языка C, ОС UNIX и других замечательных программ) еще в 1980 году, в рамках подготовки UNIX Version 8 в Bell Labs. Идея Ритчи была достаточно проста: необходимо добавить поддержку интерпретируемых программ непосредственно в ядро ОС наравне с бинарными программами.

Традиционно ядро (да и многие другие программы) определяет тип файла по первым двум байтам, называемым «магическим числом». Изначально в случае бинарных программ первые два байта использовались ядром для определения формата файла, а точнее, для определения того, как именно этот файл запускать и можно ли вообще это сделать. Технические подробности первоначального использования «магического числа» в UNIX достаточно интересны, но выходят за рамки данной статьи; интересующиеся могут найти их в англоязычной Википедии. Ограничимся замечанием, что магическое число позволяет ядру не пытаться запустить файл, предназначенный для другой ОС или вовсе не являющийся программой (никто ведь не мешает любопытному пользователю дать права на исполнение какой-нибудь картинке и попробовать «запустить» ее).

Таким образом, ядро UNIX уже могло «умело» задействовать различные варианты запуска в зависимости от первых двух байтов файла, и можно было попробовать использовать этот механизм и для обработки интерпретируемых скриптов. Оставался вопрос: как именно с помощью двух байтов подсказать ядру, что надо запустить тот или иной интерпретатор? Понятно, что имя интерпретатора в два байта не уместить, но и предусматривать отдельное двухбайтовое число для каждого интерпретатора нерационально. Ведь новые интерпретаторы появляются постоянно, и, во-первых, со временем двух байт может и не хватить, а во-вторых, для поддержки нового интерпретатора необходимо будет изменять код ядра.

Авторами UNIX из Bell Labs было предложено другое решение: зарезервировать комбинацию символов #! (решетка и восклицательный знак) в качестве индикатора, что программу надо запускать с помощью интерпретатора, а имя интерпретатора (вместе с дополнительными параметрами, если это необходимо) указывать сразу за первыми двумя байтами файла. Окончанием строки для вызова интерпретатора считается символ новой строки (\n), то есть набор символов начиная с третьего байта и до первого встреченного символа новой строки, и есть имя интерпретатора вместе с дополнительными опциями. Фактически этот набор символов представляет собой командную строку, которая будет выполнена с запущенным файлом-скриптом в качестве аргумента.

Поскольку скрипты – это обычные текстовые файлы, то для обработки их ядром в такой концепции они должны иметь вид:

```
#! <интерпретатор>
<текст скрипта>
```

Например, реализация самой популярной программы на языке Shell будет выглядеть так:

```
#! /bin/sh
echo «Hello, world!»
```

Сочетание символов #! получило название «Shebang» (иногда можно встретить упоминания «sha-bang», «hash-bang» и схожие), получившееся в результате облагораживания конкатенации слов «SHArp» и «bang» – именно так традиционно назывались символы # и ! в UNIX.

Наблюдательные читатели наверняка заметят, что имя интерпретатора здесь на самом деле отделено от первых двух байтов пробелом; это допустимо, но необязательно (впрочем, о некоторых подобных особенностях shebang мы поговорим ниже).

Итак, в качестве интерпретатора стало можно указывать не только классический /bin/sh или другую командную оболочку (например, C Shell – /bin/csh), но и вообще любую программу. В частности, стало можно написать очень короткую реализацию программы, печатающей свой исходный код, – буквально одна строка:

```
#!/bin/cat
```

(напомню, что программа cat выводит на экран содержимое файла, переданного ей в качестве аргумента).

Следует помнить, что файл со скриптом передается интерпретатору «как есть», то есть вместе с первой строчкой, содержащей shebang. Для успешной работы интерпретатор должен просто проигнорировать эту строку, ведь если он воспримет ее как инструкцию, то результат работы скрипта может оказаться неожиданным. Самый простой способ реализовать это – считать строки, начинающиеся с решетки, комментариями. Именно этот подход используется в большинстве современных интерпретаторов – всевоз-

можных командных оболочках (Bash, Ksh и другие), языках программирования (Perl, Python, Ruby) и так далее.

Как мы уже отметили, в предложенном Ритчи подходе можно указывать не просто имя интерпретатора, но и дополнительные опции, которые ему надо передать при запуске. Например, можно запустить интерпретатор Bash в режиме ограниченной функциональности (запрещающем смену рабочего каталога, изменение ряда глобальных переменных и так далее):

```
#! /bin/bash -r
```

Изначально и по сей день в качестве имени интерпретатора используется только абсолютный путь. Если же написать просто sh вместо /bin/sh, то ядро попробует вызвать sh из текущей директории. Причина этого проста – имя интерпретатора обрабатывается ядром, которое не занимается поиском программ в системе; обработка переменной PATH и прочие действия по поиску необходимого исполнимого файла – это удел оболочки. Впрочем, в Linux и многих других потомках UNIX это ограничение легко обходится с помощью команды env, которая способна определить местоположение программы и запустить ее. Так что, если вы не уверены, в каком месте у пользователя будет находиться интерпретатор python, вы можете просто написать:

```
#! /usr/bin/env python
```

Расположение самой программы env также может быть различно в разных ОС, но по крайней мере в рамках одного семейства (например, в дистрибутивах Linux) оно едино.

Альтернативный подход

Итак, красивое и удобное решение вопроса с запуском скриптов было предложено в AT&T Bell Laboratories. Однако в это время Bell Labs уже отходила от дел, связанных с UNIX, и разработки лаборатории с новой функциональностью так никогда и не дошли до широкой общественности.

Тем не менее идею подхватили и реализовали в Berkley, так что первой распространенной системой с поддержкой #! на уровне ядра стал BSD UNIX – в виде экспериментальной опции эта функциональность была добавлена в 2,8 BSD, а по умолчанию активирована в 4,2 BSD, вышедшем в 1983 году. Коммерческой же версией UNIX от AT&T занималась отдельная UNIX Support Group (USG), разрабатывавшая знаменитую линейку UNIX System V. Поддержка #! появилась в System V Release 4 (SVR4), релиз которой состоялся в 1988 году.

Однако пользователи и System V, и BSD ощутили потребность в нововведении гораздо раньше, и причиной стала свобода выбора, присущая открытым ОС. Дело в том, что пользователи UNIX хотели чего-то более функционального, чем стандартная оболочка Shell, отличавшаяся очень скудным набором возможностей. Многие из них нашли достойную замену в лице оболочки C Shell (csh). Однако системные скрипты во всех UNIX были написаны для старого доброго Bourne Shell, а языки Bourne Shell и C Shell для скриптов не очень-то совместимы. Поэтому просто заменить одну оболочку на другую в системе было нельзя, и приходилось ставить их параллельно.

Но пользователям было недостаточно общаться с C Shell в интерактивном режиме, им хотелось писать на нем и скрипты. И эти скрипты хотелось запускать, как и скрипты на обычном Shell, без явного вызова интерпретатора. Однако вспомним, что до появления поддержки shebang на уровне ядра запуск скрипта без явного указания интерпретатора был реализован на уровне самой оболочки. И, как мы видели в начале статьи, особым интеллектом стандартная оболочка не отличалась – если ядро отказывалось запустить файл, то он передавался на интерпретацию /bin/sh. Объяснить оболочке, что некоторые скрипты надо передавать /bin/csh, возможности не было. Заботящиеся о нуждах пользователей разработчики ряда вариантов UNIX нашли решение, которое иначе как «хаком» не назовешь (собственно, оно и вошло в историю под именем csh-hack). А именно они модифицировали C Shell, научив его смотреть на первый символ скрипта. Если этот символ был равен #, то скрипт запускался через /bin/csh, иначе – через /bin/sh.

Такой подход позволил решить одну частную проблему, однако стал источником многих других. Ведь подобная реализация не отличалась масштабируемостью: что делать, если в системе три оболочки? Предусматривать отдельный символ для каждой из них и модифицировать исходный код каждой оболочки для поддержки этих символов? Явно не лучшее решение. Да и с механизмом поддержки shebang эта функциональность плохо сочетается – если ядро поддерживает shebang, а оболочка – csh-hack, то без вникания в код оболочки нельзя понять, что произойдет со скриптом, начинающимся с #! /bin/sh. А такая ситуация была вполне вероятна, поскольку некоторые производители систем на основе SVR 3.2 переносили поддержку shebang в свои ядра.

Итак, пользователи одних систем создавали скрипты для C Shell, начинающиеся с одиночного символа #, пользователи других – скрипты для стандартной оболочки, начинающиеся с #! /bin/sh, а также программы для других интерпретаторов, также начинающиеся с #!. Нетрудно догадаться, что перенос скриптов между системами стал затруднителен, и, если вы сомневались, как именно будет запущен конкретный скрипт на вашей системе, вам для надежности следовало вызвать интерпретатор, а ведь именно от этого все и хотели избавиться! Таким образом, csh-hack можно считать хорошим примером неудачного решения. Возможно, оно рассматривалось авторами как временный ход (главное – побыстрее удовлетворить нужды пользователей!), но они вряд ли просчитали все последствия такого подхода, а поискать альтернативные варианты решения проблемы не пожелали.

Современные реализации и разночтения

Недальновидность подхода csh-hack обусловила его недолгую жизнь, а все UNIX-подобные системы получили поддержку shebang. Отметим, что при этом поддерживается обратная совместимость с самыми первыми оболочками, запускавшими скрипты без shebang: если вы запустите скрипт, не начинающийся с #!, то он будет передан на интерпретацию /bin/sh. Впрочем, в современных ОС /bin/sh это уже не старый Bourne Shell, а просто символьная ссылка на другую оболочку (в большинстве Linux используется BASH – Bourne Again Shell, но возможны и другие варианты).Однако и в рамках концепции shebang также нашлось

немало причин для разногласий и расхождений между различными реализациями. Например:

- какова максимально допустимая длина строки с shebang (впрочем, в большинстве случаев хватает и 32 символов, предусмотренных первыми реализациями);
- что передавать в качестве argv[0] интерпретатору – имя скрипта или имя интерпретатора, и надо ли там указывать полный путь или только само имя;
- как передавать интерпретатору остальные аргументы – нужно ли передавать все, что следует за его именем, как единый аргумент либо разбивать на массив аргументов;
- можно ли указывать в качестве интерпретатора программу, которая сама является скриптом и требует для своего запуска другого интерпретатора;
- следует ли автоматически удалять все пробелы в конце строки с shebang;
- как следует обрабатывать символ возврата каретки (\r) в конце строки, начинающейся с shebang, – этот вопрос стал актуален по мере появления скриптов, созданных в DOS и Windows, где в качестве перевода строки используется сочетание \r\n вместо традиционного для UNIX одиночного символа \n;
- …и так далее.

Детальное исследование особенностей реализации shebang в различных системах и познавательную сводную таблицу можно найти в статье [1].

Различия между вариациями UNIX были признаны достаточно серьезными, чтобы не включать shebang в стандарты POSIX и Single UNIX Specification. Первым стандартом, в котором появилась спецификация поведения shenabg, стал Linux Standard Base версии 4.0, выпущенный в 2008 году. Актуальную версию спецификации shebang в LSB можно найти по ссылке [2], но следует помнить, что это описание относится к различным разновидностям Linux, а другие UNIX-подобные системы могут ему не соответствовать.

В общем, разработчикам надо было быть аккуратными при создании скриптов, использующих что-то более хитрое, чем вызов интерпретатора типа /bin/sh или /usr/bin/python – переносимость таких скриптов между различными системами может оказаться под вопросом.

Не обошлось и без ложных домыслов, вносивших сумятицу и запутывавших пользователей и разработчиков. Например, есть ошибочное мнение, что некоторые системы требуют отделять сочетание символов #! от имени интерпретатора пробелом. Более того, можно встретить утверждение, что ядра некоторых ОС анализируют первые четыре байта файла и эти байты должны составлять строку #! /. Один из возможных источников этого заблуждения – описание системных вызовов exec() в одной из предварительных версий BSD 4.1. Однако если разработчики BSD и подумывали о чем-то подобном, то до реализации дело не дошло. А, возможно, составлявший документацию человек просто сделал неверные выводы из анализа существующих скриптов, ведь по рассмотренным выше причинам после shebang почти всегда используется абсолютный путь, начинающийся в UNIX со слеша, а слеш часто отделяется от восклицательного знака пробелом для читаемости. Документация BSD была оперативно исправлена, и в последующих редакциях упоминаний об обязательности пробела после shebang не было, однако что написано пером – не вырубишь топором. Въедливые исследователи, создававшие руководство GNU Autoconf (инструментария для создания скриптов, выполняющих конфигурирование перед сборкой программы из исходных текстов – например, определяющих доступность и местоположение в системе необходимых библиотек и заголовочных файлов), посвятили целый раздел разработке переносимых Shell-скриптов. И в этом разделе они отметили, что в некоторых системах пробел после shebang может являться обязательным. Со временем этот раздел документации Autoconf стал ориентиром для многих разработчиков, создающих скрипты для Shell, даже если эти скрипты не имели к Autoconf никакого отношения. В результате некорректное утверждение, изначально промелькнувшее в небольшом количестве документов, «пошло в массы».

Можно было бы упрекнуть разработчиков Autoconf в том, что они не удосужились проверить все свои советы на практике. Однако следует помнить, что Autoconf рассчитан на использование во всевозможных вариантах UNIX. А большинство из этих вариантов отнюдь не бесплатно, и позволить себе иметь «зоопарк» из десятков различных версий всевозможных UNIX создатели документации Autoconf вряд ли могли. Так что, возможно, они просто решили подстраховаться – в конце концов даже если злосчастный пробел и не обязателен, его наличие ничего не испортит.

В итоге совет ставить пробел просуществовал в документации Autoconf достаточно долго и был убран только в июле 2009 года. Этот пример является хорошей демонстрацией того, что аккуратность важна не только при проектировании и разработке ПО, но и при составлении документации. К сожалению, для многих разработчиков является откровением тот факт, что пользователи принимают на веру все, что написано в документации. Неверные сведения имеют обыкновение надолго оставаться в памяти пользователей, и впоследствии можно потратить немало сил на их переучивание.

***

История shebang наглядно демонстрирует, что поговорка «семь раз отмерь – один раз отрежь» вполне актуальна и для индустрии разработки ПО, занимающейся производством нематериальных сущностей. К сожалению, ошибки, подобные сделанным при решении проблемы удобного запуска скриптов в UNIX, возникают и по сей день. Shebang – всего лишь один из примеров разногласий при реализации одной и той же функциональности в разных вариантах этой ОС. В совокупности такие разногласия привели к раздробленности мира UNIX и в итоге стали одним из факторов, приведших к катастрофическому снижению доли этой ОС на рынке. Поэтому имеет смысл сделать верные выводы из истории shenabg и UNIX вообще, чтобы ваши творения не постигла такая же участь. EOF

1. The #! magic, details about the shebang/hash-bang mechanism on various Unix flavours – http://www.in-ulm.de/~mascheck/various/shebang.
2. Linux Standard Base Core Specification 4.1, Chapter 18. Additional Behaviors – http://refspecs.linuxbase.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/executable-scripts.html.

Ключевые слова: shebang, скрипт, компилятор.