02

Exit status

O exit status

Uma outra característica que é importante do *shell* é que toda vez que executamos um comando, pode ser que ele funcione ou não.

As vezes queremos saber o que aconteceu com esse comando, será que ele funcionou? Será que ele não funcionou? Queremos saber qual é o *status* que o programa terminou.

Por exemplo, se rodarmos o comando `ls`, que lista o diretório atual, podemos nos perguntar, será que esse comando funcionou?

Vamos digitar o `ls` e dar um "Enter":

```
> ls
Desktop     examples.desktop        mostra_idade        Pictures        Videos
Dowloads        help        mostra_idade~Public     zip
Downloads       loja        Music       Templates
```

Existe uma variável no *shell* que mostra o resultado do último programa executado. Basta digitar `echo`, espaço, `$` e o nome da variável que representa o resultado do último comando executado. Teremos `echo $?`. O que estamos fazendo ao digitar o `echo $?`? Estamos perguntando qual é o último resultado que um comando, um programa teve. Se deu tudo "ok" o resultado é `0`. Vamos testar o `ls`:

```
> echo $?
0
```

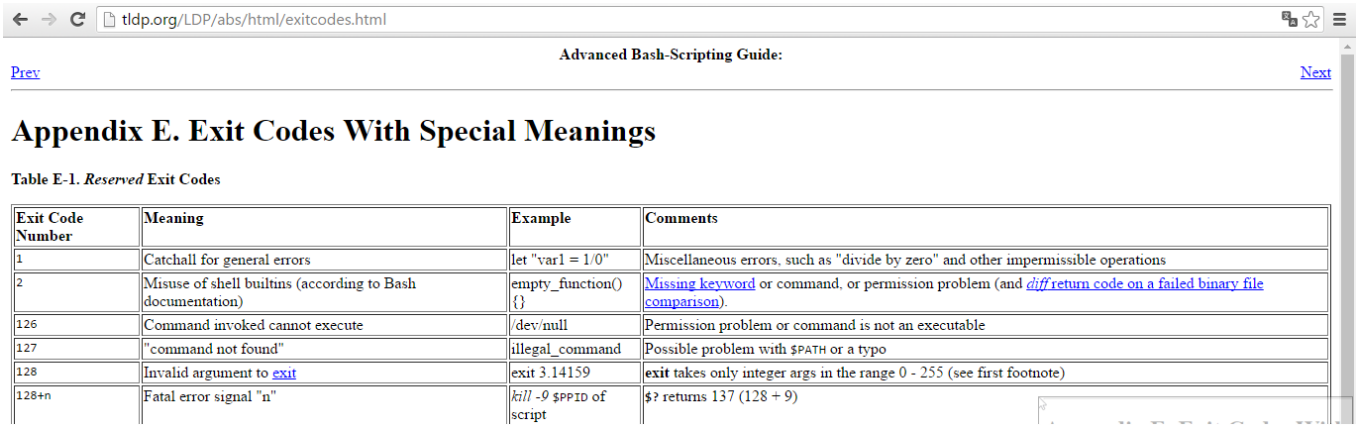
Vamos tentar executar um outro comando, por exemplo, vamos digitar algo qualquer, um `facaalgo`. Teremos a seguinte resposta:

```
> facaalgo
facaalgo: commmand not found
```

Esse comando não foi encontrado, pois, ele não existe. E se perguntarmos agora `echo $?` para saber a respeito de seu resultado? Ele nos responderá `127`. Observe:

```
> echo $?
127
```

O número `127` é um código de erro. E o que ele indica? O número `0` quer dizer que não aconteceu nada e todos os outros, por mais que existam certos padrões adotados em alguns casos, não existe uma regra geral. Inclusive, podemos buscar na Internet *bash error code 127*. Vamos dar uma olhada na seguinte página:



tldp.org/LDP/abs/html/exitcodes.html

Advanced Bash-Scripting Guide:

Prev Next

Appendix E. Exit Codes With Special Meanings

Table E-1. *Reserved* Exit Codes

Exit Code Number	Meaning	Example	Comments
1	Catchall for general errors	let "var1 = 1/0"	Miscellaneous errors, such as "divide by zero" and other impermissible operations
2	Misuse of shell builtins (according to Bash documentation)	empty_function() {}	Missing keyword or command, or permission problem (and *diff* return code on a failed binary file comparison).
126	Command invoked cannot execute	/dev/null	Permission problem or command is not an executable
127	"command not found"	illegal_command	Possible problem with $PATH or a typo
128	Invalid argument to exit	exit 3.14159	**exit** takes only integer args in the range 0 - 255 (see first footnote)
128+n	Fatal error signal "n"	*kill -9* $PPID of script	$? returns 137 (128 + 9)

Veja que essa página nos traz alguns comandos que são especiais. Diz que o "127" indica *commmand not found*. Entretanto, isso não indica que quem for utilizar o número "127" vai utilizar esse padrão, nós esperamos que sim, que as pessoas utilizem o padrão que afirma que o "127" indica um programa não encontrado.

O "127" é usado para isso, entretanto, ele também pode ser utilizado erroneamente.

Agora, queremos executar um comando e devolver um resultado qualquer.

Um exemplo de comando que podemos executar é o `python`, executamos ele e damos um `Ctrl+D` para terminar o programa. E podemos estar nos perguntando como o programa terminou, para isso digitaremos `echo $?`

```
> python
Python 2.7.10 (default, Oct 14 2015, 16:09:02)
[GCC 5.2.1 20151010] on Linux
Type "help"," copyright" , "credits" or "license"
> echo $?
0
```

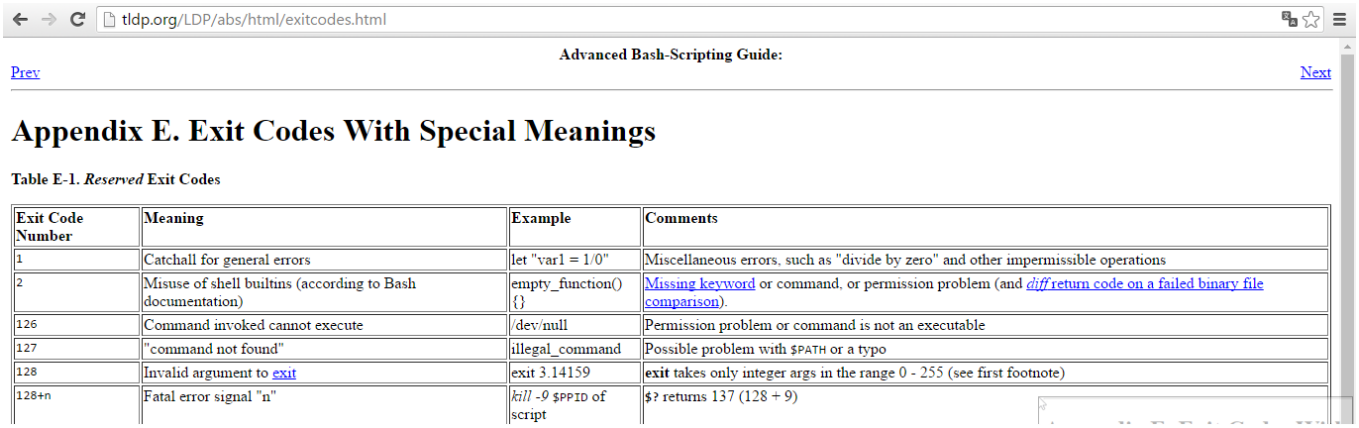
O `phyton` terminou "ok". Pois temos um `0`.

Vamos ver o `perl`, vamos executar ele e parar sua execução usando o "Ctrl C" e vamos observar o que temos se digitarmos o `echo $`:

```
> perl
C^
echo $?
130
```

Ele nos responde 130. Mas, por que ele devolve 130?

Vamos observar em nossa tabela o que significa o `130`.



tldp.org/LDP/abs/html/exitcodes.html

Advanced Bash-Scripting Guide:

Prev Next

Appendix E. Exit Codes With Special Meanings

Table E-1. *Reserved* Exit Codes

Exit Code Number	Meaning	Example	Comments
1	Catchall for general errors	let "var1 = 1/0"	Miscellaneous errors, such as "divide by zero" and other impermissible operations
2	Misuse of shell builtins (according to Bash documentation)	empty_function() {}	Missing keyword or command, or permission problem (and *diff* return code on a failed binary file comparison).
126	Command invoked cannot execute	/dev/null	Permission problem or command is not an executable
127	"command not found"	illegal_command	Possible problem with $PATH or a typo
128	Invalid argument to exit	exit 3.14159	**exit** takes only integer args in the range 0 - 255 (see first footnote)
128+n	Fatal error signal "n"	*kill -9* $PPID of script	$? returns 137 (128 + 9)

Ele nos fala que o `130` significa terminado com "Ctrl C". Repare que no `python` utilizamos um `Ctrl D` para interromper sua execução e quando pedimos `echo$?` recebemos um `0` isso ocorre porque nesse caso é tranquilo utilizar o "Ctrl D". Quando usamos o "Ctrl C" em algum programa, o código de erro vêm na forma de `130` .

Lembrando, que o status de saída com valor `0` nos diz que tudo funcionou. O status de saída com valor `130` é reservado para "Ctrl C" e o `127` é reservado para comando não encontrado.

Temos, ainda, outros padrões de erro. Como o padrão "1", normalmente utilizado para erros comuns, erros em geral. Ele cita um exemplo que é criar uma divisão por zero e isso é um erro comum, ou seja, é um erro 1.

Gostaríamos de testar isso nós mesmos, vamos abrir um `gedit` e vamos abrir o *script* `mostra_idade` e simplesmente, vamos adicionar ao final um `exit` com um código 10, fazendo isso estamos saindo do *script* que está sendo executado. Teremos o seguinte:

```
echo ${FULL_NAME} voce tem {$AGE} anos.
job=Instrutor
export ADDRES='Rua Vergueiro, 3185'
exit 10
```

Vamos salvar isso e executar. Para isso digitamos `/home/guilherme/mostra_idade` e pedir qual é o `echo $?` . Teremos:

```
> /home/guilherme/mostra_idade
voce tem anos.
> echo $?
10
```

Temos que o código de saída do último comando foi 10.

Vamos alterar novamente e ver o que acontece? Ao em vez de depois de `exit` digitarmos o `10` , substituiremos isso por `0` .

```
echo ${FULL_NAME} voce tem {$AGE} anos.
job=Instrutor
```

```
export ADDRES=`Rua Vergueiro, 3185`
exit 0
```

Vamos ver o que acontece se executarmos novamente e digitarmos o `echo $?` ?

```
> /home/guilherme/mostra_idade
voce tem anos
> echo $
0
```

Por padrão, um script como este, se não escrevermos nada, a saída dele é "ok", é `0` . Então perceba que o processo padrão devolve o valor `0` quando é um *script* e o que fazemos quando executamos um programa? Estamos criando um processo novo executando esse processo inteiro e estamos devolvendo um *status* de saída. Esse `exit status` pode ser um número, `0` , e isso significa sucesso, outro número qualquer significa `erro` .

Existem alguns padrões que devem ser seguidos, claro que, nem todo o mundo segue, mas em geral é a regra é obedecida. E conseguimos saber se deu certo ou não deu certo usando a variável `echo $?` .

Agora, o que podemos fazer com a saída de um programa? Vamos no editor, e escreveremos um novo programa. No nosso caso, o nome do novo programa será "sucesso". O que fará esse novo programa? Ele mostrará uma mensagem falando `echo Estou no sucesso` e abaixo disso digitaremos `exit 0` , apenas para ficar explícito para nós que estamos saindo com um zero. Observe o que teremos:

```
echo Estou no sucesso
exit 0
```

Vamos criar um outro programa chamado `falha` e nele escreveremos `echo Estou no falha` e abaixo disso digitaremos `exit 1` para indicar que vai ocorrer um erro. Teremos no editor o seguinte:

```
echo Estou no falha.
exit 1
```

Vamos testar esses scripts?

Primeiro, temos que digitar uma permissão de execução no `sucesso` e no `falha` . Para isso digitaremos `chmod +x sucesso falha` e vamos executar o `/home/guilherme/sucesso` e o `/home/guilherme/falha` e digitaremos `echo $?` para os dois. Vamos observar o que acontece:

```
> /home/guilherme/sucesso
Estou no sucesso
echo $?
0
/home/guilherme/falha
Estou no falha
echo $?
1
```

Temos `0` , isto é, tudo certo para o `/home/guilherme/sucesso` e `1` , ou seja, deu erro para `/home/guilherme/falha` .

O que podemos fazer com isso? Acontece que as vezes queremos executar dois programas, um depois do outro. Queremos executar um programa e de acordo com o resultado desse programa queremos executar outro programa. Por exemplo, temos nossos scripts e se executarmos o `/home/guilherme/sucesso` ele vai executar esse programa e vai devolver uma saída. Mas, gostaria de executar um próximo comando, somente, se o comando anterior tiver sido um sucesso, isto é, tiver `0` . Para isso, podemos falar o seguinte, `se o comando foi um sucesso, então, execute o próximo comando` , para isso usaremos dois "E" comerciais, `&&` e falaremos `echo Win!` , que significa, execute o próximo comando.

Teremos o seguinte:

```
> /home/guilherme/sucecco
Estou no sucesso
> /home/guilherme/sucesso && echo Win!
```

Com isso estamos dizendo que se o comando anterior devolver `0` , o `shell` executará o comando que está depois dos dois `&&` .

Vamos dar um "Enter" e observar o que acontece. Ele executou o primeiro, que foi um sucesso, isto é, tem um `0` e, depois executou o segundo. Vamos observar:

```
> /home/guilherme/sucesso && echo Win!
Estou no sucesso
Win!
```

E se ao em vez disso pedíssemos para ele executar a `falha` ? Digitaríamos, `/home/guilherme/falha && echo Win!` . Vamos analisar o que acontece:

```
> /home/guilherme/falha && echo Win!
Estou no falha
```

Pararia na execução da `falha` . E por que isso acontece? Por que o `/home/guilherme/falha` devolve um erro de saída e o duplo "e" comercial, o `&&` fez com que parássemos no `falha` e não executássemos o próximo comando. O `&&` é usado para falar que o segundo comando será executado apenas se o *exit code* do primeiro comando for igual a zero, isto é, apenas se o primeiro for sucesso. Assim, se o primeiro não tivesse sido executado com sucesso, o segundo comando não seria executado.

Da mesma maneira que temos um "e" que é o `&&` também temos o "ou". Isto é, faça uma coisa ou faça outra. Para dizermos o "ou" utilizamos o *pipe* duas vezes `||` , o *pipe* é também chamado de cano. Então, quando digitamos `/home/guilhereme/sucesso` o que estamos dizendo é, execute o sucesso ou mostre a mensagem `Win!` . Vamos ver o que acontece?

```
> /home/guilherme/sucesso || echo Win!
Estou no sucesso
```

Quando executamos a primeira linha, ele nos mostra o `Estou no sucesso` e acaba por aí, isso acontece pois estamos falando faça isso ou aquilo e se o primeiro deu certo, não precisa executar o segundo.

Perceba que é totalmente diferente do comportamento que tínhamos no "e" que possui um sentido lógico distinto do "ou" .

O que o "ou" faz se o primeiro der verdadeiro? Não executa mais nada. E se o primeiro der falso? Ele vai tentar executar o segundo e ver o que acontece.

Retomando

O "e" executa o primeiro e só executa o segundo depois que o primeiro deu certo. O "ou" diz que é para executar o primeiro e só executar o segundo, ou seja, apenas, quando o primeiro der errado.

As vezes vamos querer utilizar um ou outro, o "e" ou o "ou". Por exemplo, se queremos compactar os arquivos e desejamos que uma mensagem seja mostrada apenas se deu certo, podemos escrever da seguinte maneira `zip.arquivos.zip mostra_idade sucesso falha` esses são os arquivos que desejamos compactar e se isso for um sucesso, temos `&&`, então, mostrará a mensagem compactada, o `echo Compactado`. Vamos observar como ficou?

```
> zip arquivos.zip mostra_idade succes falh && echo Compactado
    adding: mostra_idade(deflated 1%)
    adding sucesso (stored 0%
    adding: falha (stored 0%)
Compactado
```

Ele compactou os arquivos e também mostrou a mensagem compactada.

E se colocássemos um nome de arquivo que não existe e executássemos isso? Por exemplo, ao em vez de escrever `falha`, tivéssemos escrito `falh` e executássemos o `echo Compactado`. Vamos observar se isso tivesse acontecido:

```
> zip.arquivos.zip mostra_idade sucesso falh && echo Compactado
        zip warning:name not matched: falh
updating: mostra_idade (deflated 1%)
updating: sucesso (stored 0%)
Compactado
```

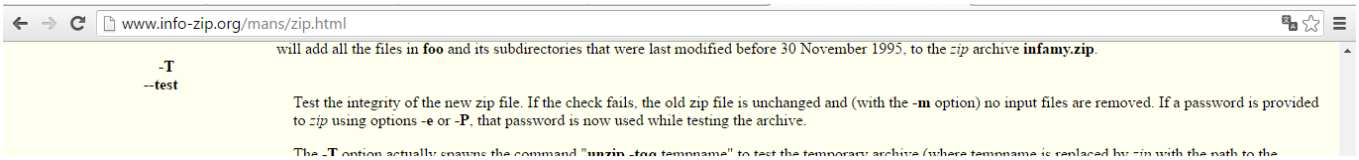
Aí ele executou, mesmo o `zip` não tendo encontrado esse arquivo, ele devolve o `0`. O *status* de saída vai depender de cada programa. Para o `zip`, não encontrar esse arquivo não é um erro de saída, é simplesmente que ele não encontrou o arquivo.

Se passarmos três arquivos para o `zip` e ele não encontrar, ele fica de boa. Mas, e se o *zip* não encontrar nenhum arquivo, por exemplo, vamos digitar `mostra_ie` ao em vez de "mostra_idade", `scesso` ao em vez de "sucesso", e `flha` ao em vez de "falha". Vamos observar o que acontece se executarmos isso:

```
> zip arquivos.zip mostra_ie scesso falh && echo Compactado
        zip warning: name not matched: mostra_ie
        zip warning: name not matched: falh
```

Aqui o `zip` fica confuso, quer dizer, para definição do programa `zip` ele só devolve uma mensagem de erro se não encontrar nenhum arquivo para colocar dentro do `zip`. Aí ele devolve um valor diferente de zero, mas se ele encontrar pelo menos um arquivo, ele não devolverá diferente de zero. Claro, existem inúmeras possibilidades de errar o comando `zip`. Mas, se você quiser entender se ele vai devolver um erro ou um não erro. Teremos que olhar sobre esse programa.

Vamos pesquisar *linux zip error code*, vamos acessar . Podemos ver os possíveis erros que podem acontecer e ele cita quando esses erros podem acontecer. Por exemplo, deu erro de memória? É `zip -T`.



Assim, cada comando tem o seu comportamento para devolver zero ou outro valor. Nós temos que conhecer ele.

No nosso caso, podemos fazer ele `zipar` e caso consiga `zipar` deve mostrar a mensagem de compactado. E para isso usamos o `&&` .

O `&&` para dizer para executar o primeiro e se este for um sucesso, pode executar o segundo ou usa-se o "ou" quando ele não consegue executar o primeiro executa o segundo, por exemplo, mostrando uma mensagem de erro mais adequada para o nosso usuário final, poderia ser um caso bonitinho.

Perceba que o "e", `&&` , ou o "ou", `||` são usados no dia a dia para fazer esse tipo de trabalho. por exemplo, execute um comando e execute um segundo, ou execute um comando ou execute um segundo.

Para fazer testes é legal executar um *bash* dentro do *bash* atual. Para abrir um `bash` basta digitar `bash`

```
> bash
```

É como se tivéssemos um *shell* e entrássemos dentro dele. É como se tivéssemos um *shell* principal e tivéssemos criado um processo novo e dentro dele está rodando um novo *shell*. No *shell* novo vamos digitar `exit 15` para sair desse *shell* novo e voltamos para o *shell* principal:

```
> bash
> exit 15
```

Qual é o resultado do comando? Para saber isso basta digitar `echo $?` e teremos a resposta `15` .

```
> echo $?
15
```

Vamos entrar, novamente, no `bash` e dar `exit 0` e perguntar `echo$?` . Teremos o seguinte:

```
> bash
> exit 0
> echo $?
0
```

O resultado do comando é `0` . Podemos utilizar o *bash* para fazer os testes que queremos, podemos, inclusive, escrever em uma linha apenas, `bash -c "exit 0" && echo win!` . Observe que se for "sucesso", mostrará a mensagem `win!` . Vamos executar isso para ver o que acontece:

```
> bash -c "exit 0" && echo win!
> bash -c "exit 10" &&
 echo win!
```

Vamos testar com `exit 10` ? Isto é, `bash -c "exit 10" && echo win` . Não vai acontecer nada. Ele não vai se pronunciar.

Então, podemos utilizar o `bash -c` para executar o comando e fazer os testes que queremos. Seja esse tipo de comando, com o `&` comercial ou com o `|` . Lembrando que nessas situações digitamos o dois pipes, `||` e dois "e" comerciais, `&&` para fazer o papel do `ou` ou do `e` , respectivamente.

Repare que o `exit code` é extremamente importante para entendermos se o último processo rodado foi um sucesso ou não, para, a partir daí tomarmos a decisão se executamos ou não um próximo programa.