Jeroen van Nieuwenhuizen
Snow B.V.
9 December 2017

Type less, do more

Effectief gebruik van de bash shell

Algemene intro

In dit artikel, gebaseerd op mijn T-Dose 2017 presentatie, neem ik de lezer mee op een (hopelijk) interessante tocht langs leuke bash features die je type werk besparen. Onderdelen die aanbod komen zijn brace expansion, tilde expansion, history expansion, wat handige trucs om slimmer door directories te navigeren en enkele shortcuts in de emacs mode. Zelf gebruik ik deze features ook te weinig, dus het maken van genoemde presentatie en dit artikel was voor mij weer een leuke opfrisser en voorziet mij weer van motivatie om nog slimmer te werken.

Brace expansion

De basis vorm van brace expansion komt tot uiting bij een commando zoals:

```
echo hallo{1,2,3}
```

wat als resultaat oplevert:

```
hallo1 hallo2 hallo3
```

Wat er dus in basis gebeurd is dat elk element uit de lijst, gescheiden door komma's, tussen { } afzonderlijk achter hallo wordt geplakt en deze combinaties als afzonderlijk woord op de output verschijnen. Tussen de { } kun je in principe elke lijst die je wilt opgeven.

Wat kan er nog meer met brace expansion? Als we kijken naar een standaard voorbeeld zoals:

```
for i in `seq 1 9`
do
        echo $i
done
```

Dan zijn daar een paar dingen op aan te merken. Ten eerste is de `` constructie deprecated en dien je eigenlijk gebruik te maken van de `$(..)` constructie. Wat zou resulteren in:

```
for i in $(seq 1 9)
do
        echo $i
done
```

Een van de voordelen van de `$(..)` constructie is bijvoorbeeld dat je gebruik kunt maken van nesting. Dus bijvoorbeeld:

```
echo $(echo $(echo $(echo $PWD)))
```

Levert uiteindelijk als resultaat je huidige working directory op. Maar terug naar brace expansion. Hiermee kun je bovenstaande for loops veel makkelijker doen, namelijk:

```
for i in {1..9}
do
        echo $i
done
```

Hierbij wordt het `{1..9}` dus expaned naar 1,2,3,4,5,6,7,8,9. En heeft het dus effectief het zelfde effect als het runnen van `seq 1 9`. Naast het expanden van nummers met de `{1..9}` constructie kan je het zelfde toepassen met letters. Dus

```
echo {a..z}
```

Levert je alle letters op van a t/m z.

Net als met de `$()` constructie kun je tenslotte brace expansion ook nog nesten.
Bijvoorbeeld:

```
echo hall{a{1..2},b{c,d,{e..g}}}
```

Naast het gebruik van brace expansion in for loops is er nog een andere praktische toepassing die ik met je wil delen. Stel je wilt /etc/ntp.conf editten en voor dat je dat doet /etc/ntp.conf kopiëren naar /etc/ntp.conf.bak dan kan dat met:

```
cp /etc/ntp.conf{,.bak}
```

Moving through directories

De meeste bekende manier om door directories heen te bewegen is natuurlijk het cd commando. De verschillende vormen zoals `cd <directory>`, `cd ..` en `cd /path/to/directory`, zullen bekend zijn. Met `cd` kan je echter nog meer. Zo is er `cd` zonder een argument om naar je eigen home directory te gaan. En `cd -` om terug te gaan naar de vorige directory.

Maar er is meer, bash beschikt namelijk ook over een zogenaamde dirstack. Met het commando `pushd` kun je hier directories op plaatsen en met het `popd` commando kan je ze er weer vanaf halen. Een voorbeeld: Stel je hebt de directory structuur a/b/c/d/e. Met `pushd a` kom je in de directory a terecht en je directory stack ziet er als volgt uit:

$ dirs -p
~/a
~

Na `pushd b` zit je in de ~/a/b directory en als we dan nog een `pushd c` commando uitvoeren zitten we in de ~/a/b/c directory. Vragen we wederom de directory stack op, dan ziet die er als volgt uit:

$ dirs -p
~/a/b/c/
~/a/b
~/a
~

Of te wel onze hele geschiedenis van directories, waar we door heen gegaan zijn met het `pushd` commando, staat op de stack. Willen we nu terug naar de ~/a/b directory, dan kan dat met het `popd` commando. Als we na het uitvoeren van `popd` een `echo $PWD` doen zien we dat we inderdaad in de ~/a/b directory zijn uitgekomen. Nogmaals een `popd` commando brengt ons dan weer in de ~/a directory en als we dan wederom een `popd` commando doen zijn we weer terug in de ~ directory.

Is het nu wijs om dan maar gewoon `cd` te aliassen naar `pushd`? Nee, want bash kent geen manier om de totale lengte van de directory stack te beperken[1]. Je zou hier om heen kunnen werken door een PROMPT_COMMAND te maken die de oudste elementen uit je directory stack haalt door de oudste elementen te verwijderen. Dit implementeren valt echter buiten de scope van dit artikel.

[1] In tegenstelling tot zsh dat een DIRSTACKSIZE variable heeft om een limiet hier op te zetten.

Het is echter wel handig om te weten hoe je elementen uit je directory stack te verwijdert. Hoe gaat dat in zijn werk? Stel we hebben de volgende directory stack:

~/a/b/c/d/e/f
~/a/b/c/d/e
~/a/b/c/d
~/a/b/c
~/a/b
~/a
~

Stel we willen het element `~/a/b/c/d` verwijderen van de directory stack. Dan kunnen we beginnen met van boven te tellen en komen tot de conclusie dat het het 3de element in de lijst is. Verwijderen kan dan met `popd +2`. Merk op dat je dan niet van directory veranderd. Wil je van de onderkant tellen, dan zou het het 5de element van onderen geweest zijn en zou je hem hebben kunnen verwijderen met `popd -4`. Merk op dat tellen dus altijd met 0 begint.

tilde expansion

Bash kent ook zoiets als tilde expansion. De meeste bekende vorm, die we al voorbij hebben zien komen, is gewoon de `~`, wat staat voor de home directory van de user. Er zijn er echter nog een paar. `~+` staat voor de huidige working directory. In plaatst van

echo $PWD
Kun je in het vervolg dus altijd

echo ~+
gebruiken. Scheelt toch weer 2 karakters type werk. Ook is er nog `~-`, wat verwijst naar de vorige directory. Vooral handig als je iets moet kopiëren of editen in de vorige directory waarin je zat.

Waar dit echt nuttig wordt is als je dit gaat combineren met `pushd` en `popd`, aka wat er op de directory stack staat. Met `~N` en `~-N` (waarbij N dus een getal is) kun je verwijzen naar elementen op de directory stack. Het getal N volgt hierbij het zelfde stramien als dat bij het verwijderen van elementen uit de directory stack. Hiermee kun je dus handig verwijzen naar files in directories waar je eerder was (mits je dus `pushd` gebruikt hebt), zonder weer het hele pad in te moeten typen.

History expansion: events

Dat bash de geschiedenis van in getypte commando's bijhoudt zal ongetwijfeld ook wel bekend zijn. Deze history bekijken kan met het history commando. Een voorbeeld:

$ history
1 echo one
2 echo one two
3 echo one two three snow
4 echo one two three four

Als we nu het laatste commando (`echo one two three four`) nog eens willen uitvoeren, kan dat met `!!`. Dus:

$!!
echo one two three four
one two three four

De `!!` wordt natuurlijk het meest gebruikt als een commando dat je uitvoerde toch als root uitgevoerd moet worden, dus `sudo !!`

Gelukkig is dat niet alles wat kan met history expansion. Je kan ook verwijzen naar het nummer van een voorgaand commando. Dus als je `echo one two` nog eens uit zou willen voeren kan dat met `!2`.

$!2
echo one two
one two

Hierbij verwijst de 2 dus naar het regelnummer zoals het history commando dat in zijn output meegegeven heeft. Om de output one two three nogmaals te krijgen zouden we dus `!3` ingetypt hebben.

Naast natuurlijk direct de nummers opgeven kunnen we hier ook weer gebruik maken van relatieve nummers zoals we al zagen bij `popd`. Als we we weer even terug gaan naar ons history voorbeeld:

$ history
1 echo one
2 echo one two
3 echo one two three snow
4 echo one two three four

Als we nu wederom het commando met regelnummer 2 opnieuw zouden willen uitvoeren zouden we dat kunnen doen met:

$!-3
echo one two
one two

Immers regel 4 is relatief gezien -1, regel 3 is relatief gezien -2 en regel 2 is dus relatief gezien -3. Zelf vind ik over het algemeen relatieve nummers wat onhandiger en heb ik mijn bash prompt zo aangepast dat aan het begin van mijn prompt altijd het history nummer van het commando staat. Hierdoor is dan vrij gemakkelijk om absolute history nummers te gebruiken.

Er zijn nog 2 andere opties die handig zijn om te weten. Met `!<commando>` voer je nogmaals het laatste commando uit dat begon met <commando>. In ons history voorbeeld zouden we met `!echo` dus nogmaals `echo one two three four uitvoeren`. Wil je meer gericht zoeken op een specifieke string, dan kan dat met `!?<string>`. Zo zou bijvoorbeeld `!?snow` op basis van ons history voorbeeld commando 3 nog eens uitvoeren, dus `echo one two three snow`.

Tot nu toe hebben we gekeken naar terug in de tijd, maar kan je ook refereren naar de huidige regel? Ja, dat kan. Namelijk met `!#` kun je refereren naar de huidige regel die je tot nu toe hebt ingetypt.

Bijvoorbeeld

$ echo test !# test
echo test echo test test

Merk op de extra spatie naar de 2de test. Immers de spatie voor `!#` heb je ook al in getypt. Verder op in het artikel zal ik een voorbeeld geven waarin het nut van het gebruik van `!#` duidelijker zal worden. Maar voor nu is een leuke om zelf eens over na te denken is wat

$ echo test !# test !# test

op zou gaan leveren als output. De tweede `!#` include namelijk de expansie van de eerste `!#`.

Totzover het gebruik van ! constructies voor history expansie van hele regels. In het volgende deel van het artikel gaan we kijken naar hoe je delen van voorgaande commando's kan hergebruiken zonder ze helemaal opnieuw in te typen. Maar voordat we dat gaan doen wil ik nog een andere truc met jullie delen. Namelijk de ^ constructie. Wat kan je daarmee? Een praktijk voorbeeld:

$ systemctl start apache

en dan wil je daarna natuurlijk vaak wel eens iets doen zoals

$ systemctl enable apache.
Eigenlijk dus een herhaling van het voorgaande commando maar met start vervangen door enable. Hiervoor kun je dus de ^ constructie gebruiken. Dus om hetzelfde als de 2 voorgaande commandos te bereiken doe je:

$ systemctl start apache
$ ^start^enable

Of te wel vervang in het vorige commando het eerste voorkomen van start door enable en voer het uit.

History expansion: words

Je zult echter vaak zien dat je alleen geïnteresseerd bent in een deel van de vorige command line. Elke element/argument van een command line is een word. Of anders gezegd, een commandline bestaat uit 1 of meerdere words. Even een voorbeeld:

$ echo een twee drie vier vijf
Hierbij zijn er in totaal 6 words. Echo is word 0, een is word 1, twee is word 2 en zo voorts. Hoe kun je nu naar zo'n word refereren? Dat kan met `!:<wordnumber>`. Stel we hebben wederom getypt:

$ echo een twee drie vier vijf
En nu willen we `echo twee vier` uitvoeren. Dan kan dat als volgt:

$ echo !:1 !:4
echo een vier
een vier

Hierbij verwijzen we nog naar de vorige commandline. Je kunt dit echter ook weer combineren met wat we in het gedeelte over het expanden van events geleerd hebben. Laten we wederom ons bekende voorbeeld nemen:

$ history
1 echo one
2 echo one two
3 echo one two three snow
4 echo one two three four

Nu kan ik dus bijvoorbeeld doen:

$ echo !4:2 !3:4 !2:2
echo two snow two

!4:2 betekent namelijk, van regel 4 in onze history, neem het 2de word.

!3:4 betekent, van regel 3 in onze history, neem het 4de word.

En !2:2 betekent dan uiteraard, van regel 2 in onze history, neem het 2de word.

Dan zijn er ook nog shortcuts om te verwijzen naar hele delen van de voorgaande regels:

$ echo een twee drie vier vijf
$ echo !:2-
echo twee drie vier
twee drie vier

Of te wel de `!:<number>-` constructie levert alle words vanaf <number> *behalve* het laatste word.

$ echo een twee drie vier vijf
$ echo !:2*
echo twee drie vier vijf
twee drie vier vijf

Of te wel de `!:<number>*` constructie levert alle words vanaf <number> inclusief het laatste word.

$ echo een twee drie vier vijf
$ echo !:-2
echo echo een twee
echo een twee

De `!:-<number>` constructie levert dus alle words tot/met <number>, inclusief het 0de word.

$ echo een twee drie vier vijf
$ echo !:2-4
echo twee drie vier
twee drie vier

Verklaart dan zichzelf eigenlijk, alles vanaf word 2 t/m word 4.

Wat zijn er verder nog voor opties:

`!:^` verwijst naar word 1.

`!:$` is voor mensen die altijd het laatste woord willen hebben.

`!:*` is alles behalve word 0.

`!:-` is alles behalve het laatste word.

Zoals waarschijnlijk al duidelijk wordt kunnen commando regels best ingewikkeld worden als je meerdere van dit soort expansies gebruikt op een commando regel. Vaak zul je dan ook even willen checken of het commando ook inderdaad geworden is wat je graag wilt. Een handige shortcut hiervoor is `ctrl-meta-e`. Deze toetscombinatie voert de history expansie uit op de huidige commando regel, zonder deze al uit te voeren. (Voor mac gebruikers, de optie om alt alt te laten zijn zit ergens verstopt in je terminal settings.)

History expansion: modifiers

We hebben ondertussen geleerd hoe we kunnen verwijzen naar voorgaande commando regels en naar delen van voorgaande commando's. Daarnaast leerden we ook de `!#` constructie om te verwijzen naar de huidige regel die we tot nu toe getypt hadden. Maar uiteraard zijn er nog meer mogelijkheden. Je kunt namelijk ook de delen van voorgaande commando's waar je naar verwijst aanpassen. In onderstaande voorbeelden zal ik daarbij vooral gebruik maken van de `!#` constructie, maar het principe werkt het zelfde voor de andere constructies om naar (delen van) voorgaande commando's te verwijzen.

Als eerste hebben we de h modifier: remove de trailing file component. Wat eigenlijk neerkomt op de directory naam. Een voorbeeld:

$ echo /etc/ntp.conf !#:1:h
echo /etc/ntp.conf /etc
/etc/ntp.conf /etc

`!#:1:h` neemt van de huidige regel het eerste word. Dat is dus /etc/ntp.conf. Vervolgens verwijderen we de trailing file component (ntp.conf) en dus wordt `!#:1:h` /etc.

De t modifier verwijdert juist de directory naam en levert je de file naam op.

$ echo /etc/ntp.conf !#:1:t
echo /etc/ntp.conf ntp.conf
/etc/ntp.conf ntp.conf

De r modifier verwijdert de suffix. Dus van /etc/ntp.conf wordt .conf verwijderd.

$ echo /etc/ntp.conf !#:1:r
echo /etc/ntp.conf /etc/ntp
/etc/ntp.conf /etc/ntp

Mocht je alleen de suffix willen, dan is er uiteraard ook een optie voor. De e optie.

$ echo /etc/ntp.conf !#:1:e
echo /etc/ntp.conf .conf
/etc/ntp.conf .conf

Leuk allemaal natuurlijk, maar kan je ze ook combineren? Uiteraard kan dat:

$ echo /etc/ntp.conf !#:1:r:t
echo /etc/ntp.conf ntp
/etc/ntp.conf ntp
We verwijderen namelijk eerst de suffix (.conf en dat levert /etc/ntp op) en dan nemen we het file gedeelte van daarvan (ntp).

En dan 1 van mijn favorieten. We hadden natuurlijk al de ^^ truc, maar daarmee kan je niet verder terug in history en dat zul je soms wel willen. Even concreet een voorbeeld:

$ systemctl start apache
$ ps aux | grep httpd

Als ik nu systemctl enable apache wil uitvoeren. Dan wil ik natuurlijk ook zoiets als de ^^ doen, maar ja, te ver weg, want ^^ werkt alleen op het laatst uitgevoerde commando. Dit valt op te lossen door gebruik te maken van de s modifier, die ongeveer net zo werkt als bij sed. Om systemctl enable apache uit te voeren kan ik dus doen:

$!-2:s/start/enable/
systemctl enable apache

`!-2` om het start commando op nieuw uit te voeren kenden we al. En met de s modifier vervangen we start door enable wat ons oplevert wat we willen. Ook hierbij wordt alleen het eerste voorkomen van start vervangen door enable. Mocht je global replace willen dan kun je dat doen met de `!:gs/start/enable/` constructie. Dus in plaats van s gebruik je gs als modifier.

Hierbij komen we aan het eind van history expansie en ik had nog beloofd een nuttig voorbeeld te geven van het nut van `!#`. Stel je hebt een file in de files/etc/systemd/network/eth0.network staan en je wilt die verplaatsen naar de templates/etc/systemd/network/ directory en gelijk voor zien van de .erb extensie.

Dan kan je dat dus doen met:

$ mv files/etc/systemd/network/eth0.network !#:1:s/files/templates/.erb

fc

Het `fc` commando binnen bash geeft je de mogelijkheid om history entries te editen in je favoriete editor. Als je gewoon kiest voor `fc` edit je het laatst uitgevoerde commando in een editor. Na het opslaan en afsluiten van de editor wordt je commando dan uitgevoerd.

Door een history nummer op te geven aan `fc` kun je er voor kiezen om een ander commando dan de laatste te editen in je editor. Als we weer kijken naar ons standaard voorbeeld:

$ history
1 echo one
2 echo one two
3 echo one two three snow
4 echo one two three four

Zo zou `fc 2` dus er voor zorgen dat je het commando `echo one two` gaat editen. Wil je bijvoorbeeld de commandos `echo one two` en `echo one two three snow` in 1 keer aanpassen en ze allebei uitvoeren, dan kan dat met `fc 2 3`.

Tenslote biedt `fc` ook nog de mogelijkheid om ook weer een search en replace te doen. Bijvoorbeeld:

$ echo start start start
start start start
$ fc -s start=enable
echo enable enable enable
enable enable enable
Merk hierbij op dat `fc` dus altijd een global replace doet en niet alleen het eerste voorkomen van het woord vervangt.

Emacs mode

In dit gedeelte ga ik in op wat toetscombinaties die er zijn in Emacs mode en ik zelf grappig en/of nuttig vindt. Het is slechts een samenvatting van de vele mogelijke toetscombinaties die er zijn. Een veel grotere aantal valt terug te vinden in de bash manpage.

Moving around

Om je op de command line eenvoudig te kunnen verplaatsten zijn `ctrl-b` en `ctrl-f` handig om te kennen. Hiermee kun je respectievelijk een karakter terug op de commandline of in het geval van `ctrl-f` een karakter vooruit op de commandline. In praktijk is er geen verschil met het gebruik van het pijltje naar links of naar rechts.

Interessanter wordt het als we kijken naar `meta-b` en `meta-f`. Ze doen eigenlijk het zelfde als hun tegenhanger `ctrl-b` en `crtl-f`, maar in plaats van een verplaatsing van 1 karakter, ga je respectievelijk een woord naar links of naar rechts.

Als je juist helemaal aan het begin van de regel wil zijn kan dat met `crtl-a`[2] en wil je naar het eind van de regel, dan kan dat met `ctrl-e.`

Searching

Als je kiest voor `ctrl-r` en vervolgens begint te typen, kun je daarna door nog eens te kiezen voor `ctrl-r` terug scrollen door je history in de items die matchen wat je getypt hebt.

Arguments

Als we weer even naar een voorbeeld gaan:

$ echo een twee drie vier snow
een twee drie vier snow

Stel nu dat we het commando `echo snow twee` uitwillen voeren. Dan kunnen we dat doen door eerst `echo` te typen. Vervolgens te kiezen voor de toetscombinatie `meta-.` We krijgen dan op onze commandline `echo snow` te zien. Of te wel `meta-.` voegt het laatste argument van de vorige command line in. Om twee het 2de argument van de vorige commandline in te voegen moeten we iets meer werk doen. We selecteren namelijk eerst dat we het 2de argument willen met `meta-2`. Wil je een ander nummer dan toets je uiteraard dat nummer in. Vervolgens kun je met de toetscombinatie `crtl-meta-y` het geselecteerde argument invoegen op de huidige commandline. Wat in ons geval dus resulteert in:

$ echo snow twee
snow twee

Editing

Ook om te editen op de huidige commandline zijn er handige toetscombinaties in emacs mode. Een van mijn persoonlijke favorieten is `crtl-x ctrl-e`. Hiermee start je een editor op de regel die je tot nu toe getypt hebt. Een beetje als `fc` dus, maar dan voor de huidige regel.

Wil je vanaf het huidige punt op de regel wissen tot aan het einde van de regel, dan is `ctrl-k` je vriend. Wil je juist vanaf het begin van de regel wissen tot het huidige punt, dan kun je kiezen voor `ctrl-u`.

Een andere leuke optie is de transpose optie. Stel je hebt getypt:

$ git diff acceptance production

---

[2] Als je screen gebruikt dan werkt dit niet omdat ctrl-a afgevangen wordt door screen.

En je bedenkt je dat je helemaal niet wilt weten hoe je acceptatie gelijk krijgt aan productie, maar juist hoe je productie gelijk krijgt aan acceptatie. Met `meta-t` kun je de laatste 2 woorden voor de cursor omwisselen. Dus het typen van `meta-t` levert in ons voorbeeld op:

$ git diff production acceptance

Tot slot wil ik wat betreft editing nog afsluiten met 2 handige toetscombinaties. Als eerste is er `meta-r`. Dit brengt je commandline weer terug naar de status zoals hij was voordat je begon te editen. Wil je juist stap voor stap terug dan is `crtl-_` de toetscombinatie die je zoekt.

Macros

En om het emacs mode deel af te sluiten is er nog de macro feature. Ik heb er zelf nog geen nuttige toepassing voor gevonden, maar cool is het wel natuurlijk. Met de toets combinatie `crtl-x (` begin je met het opnemen van een macro. Ben je klaar met het opnemen dan toets je `crtl-x )` om het opnemen te stoppen. Wat je dan nog nodig hebt is de wetenschap dat je met `crtl-x e` de opgenomen macro weer kan afspelen.

Disown

Ik wil afronden met een handige optie voor als je een langlopende taak gestart hebt op een remote server vanaf je laptop, maar niet in iets als tmux of screen. Als je dan weg moet en je zou disconnecten dan kan het zijn dat je taak er mee ophoudt[3]. Hier is ook een oplossing voor indien je bash gebruikt en je niet geïnteresseerd bent in de stdout en stderr van de taak. Om weer in het standaard stramien van voorbeelden te blijven met een simpele lang lopende taak.

$ sleep 3600

Je kan nu met de toetscombinatie `ctrl-z` de job pauzeren. Om nu te zorgen dat je veilig kan disconnecten van de server kun je de job naar de achtergrond brengen (`bg`), vervolgens even kijken welk job nummer je job gekregen heeft (`jobs`) en vervolgens de job disconnecten van elke controlling terminal met het `disown` commando. In ons voorbeeld:

$ bg
[1]+ sleep 3600 &
$ jobs
[1]+ Running sleep 3600 &
$ disown %1

[3] Afhankelijk van of je systeem systemd gebruikt en/of de configuratie van systemd kan het zijn dat dit niet gebeurd.

Hierbij is het nummer in het `disown %1` commando dus het nummer dat onze job gekregen heeft bij het naar de achtergrond brengen. In ons geval dus 1.

Tot slot

Hopelijk heeft of gaat dit artikel je helpen om meer uit de bash shell te halen. Mocht je nog vragen of opmerkingen hebben naar aanleiding van dit artikel, mail dan gerust naar jeroen.van.nieuwenhuizen@snow.nl.