부록 6. Shell Script

💡 5번째 부록에서 다양한 리눅스 명령어에 대해 알아보았습니다. 이제 리눅스와의 어색함이 조금 풀어지셨다면 쉘 스크립트로 넘어가서 복잡한 작업을 컴퓨터에게 시키는 법을 익혀보겠습니다.

> 출처
> - **LinuxCommand.org** : https://linuxcommand.org/index.php
> - **Linux Shell Commands** : https://docs.cs.cf.ac.uk/notes/linux-shell-commands/

0. Prologue

0.1 Shell Script

쉘 스크립트를 한마디로 정의하면 '**명령어의 모음**' 입니다. 컴퓨터가 상황별로 다양한 명령을 수행할 수 있도록 나름의 문법을 지켜서 스크립트를 작성해야 하는데, 이번 부록에서는 그 문법을 익힌 후 실제로 활용하는 법을 전달해드리겠습니다.

0.2 작성 방법

VScode 등을 활용하여 작성하는 방법도 있지만 리눅스 서버에 원격접속하여 작업하는 일이 많기 때문에 `vi`, `nano` 와 같은 에디터를 사용합니다.

- Vim
- GNU nano editor

`vi` 의 경우 익숙해지면 아주 편리하게 사용할 수 있지만 그렇지 느끼지 않는 분들도 많이 계시기 때문에 `nano` 에디터를 통해 실습을 진행하겠습니다.

1. Shell Script 실행 절차

1.1 Shell Script 작성

```
nano hello.sh
```

```
#!/bin/bash echo "Hello, Shell Script!"
```

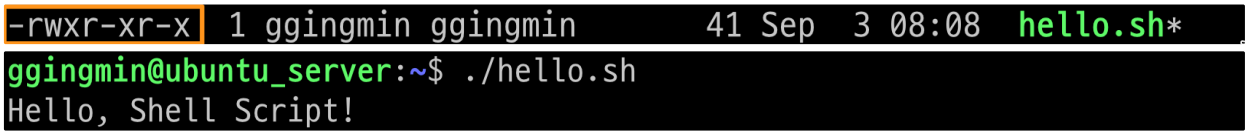
가장 간단한 형태의 쉘 스크립트를 작성해보았습니다. 개발을 배울 때 무조건 해보는 콘솔에 `"Hello, World"` 찍어보기 입니다.

쉘 파일의 확장자는 `*.sh` 입니다. 가장 중요한 것이 바로 첫번째 행인데요, 바로 쉘 파일을 해석할 프로그램을 지정해주는 역할을 합니다. 프로그램의 경로를 명시하기 위해서는 반드시 셔뱅(shebang)이라고 하는 `#!` 을 붙여주어야 합니다. 위에 작성한 소스를 해석하자면 " `/bin/bash` 로 쉘의 내용을 해석할 것이다". 가 됩니다.

1.2 실행 권한 부여

위와 같이 작성된 파일은 `ls -l` 로 조회하면 실행권한이 없는 것으로 나옵니다. 그렇기 때문에 파일명을 입력해서 실행하려고 하면 권한문제로 거부처리 됩니다. 부록 5에서 배운 `chmod` 명령어를 활용하면 이를 해결할 수 있습니다.

```
chmod 755 ./hello.sh
```



`chmod` 로 실행권한을 부여했기 때문에 파일 조회 시 `x` 권한이 추가 됐음을 확인할 수 있습니다. 파일을 실행해보면 거부처리 되지 않고 작성한 스크립트대로 문자열이 출력됩니다.

1.3 Shell 경로 설정

지금 `hello.sh` 을 실행하기 위해서는 파일의 위치를 절대경로, 혹은 상대경로 형태로 반드시 작성해주어야 실행이 가능합니다. 하지만 쉘에게 미리 경로를 알려주면 `hello.sh` 라는 파일을 바로 찾아 실행시킬 수 있습니다.

```
echo $PATH
```

`PATH` 를 조회해보면 현재 쉘에서 참조하고 있는 경로가 `:` 으로 구분되어 출력됩니다. 홈 디렉토리 하위에 `/bin` 디렉토리를 만들고 `hello.sh` 을 옮겨보겠습니다. 이어서 `export` 명령어를 통해 환경변수 `PATH` 에 새롭게 생성한 경로를 추가하여 대입합니다.

```
mkdir ./bin mv ./hello.sh ./bin/hello.sh export PATH=$PATH:/home/ggingmin/bin
echo $PATH
```

```
ggingmin@ubuntu_server:~$ echo $PATH
/home/ggingmin/.local/bin:/home/ggingmin/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loca
l/games:/snap/bin:/home/ggingmin/bin
```

`PATH` 값을 조회해보면 `export` 를 통해 세팅한 경로가 들어가있는 것을 알 수 있습니다. 이제 `hello.sh` 을 실행시켜 보겠습니다.

```
hello.sh
```

```
ggingmin@ubuntu_server:~$ hello.sh
Hello, Shell Script!
```

현재 위치에 `hello.sh` 이라는 파일이 존재하지 않음에도 불구하고 쉘이 등록된 경로에서 해당 파일을 찾아 실행합니다.

2. 환경변수와 앨리어스

처음 Ubuntu 가상머신을 부팅하면 로그인 창이 뜹니다. 계정 로그인 이후에는 bash 프로그램이 바로 실행되어 시스템에 설정되어 있는 값들을 메모리에 로드합니다. 이렇게 로드된 정보들을 일컬어 **환경** 이라고 하고 환경을 구성하는 변수들을 **환경변수** 라고 합니다.

일반적으로 Ubuntu 에서의 환경 정보는 `~/.bashrc` 에 저장되어 있습니다. `nano` , `vi` 등으로 직접 수정할 수 있고 `cat` 으로 출력도 가능합니다. 맥의 경우에는 현재 최신의 OS 기본 쉘이 **zsh** 로 변경되었기 때문에 `~/.zshrc` 파일을 확인하시면 됩니다.

```
cat ~/.bashrc
```

```
ggingmin@ubuntu_server:~$ cat ~/.bashrc
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# If set, the pattern "**" used in a pathname expansion context will
# match all files and zero or more directories and subdirectories.
#shopt -s globstar

# make less more friendly for non-text input files, see lesspipe(1)
[ -x /usr/bin/lesspipe ] && eval "$(SHELL=/bin/sh lesspipe)"
```

```
# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Add an "alert" alias for long running commands.  Use like so:
#   sleep 10; alert
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*[0-9]\+
\s*//;s/[;&|]\s*alert$//'\'')"'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi


alias python=python3.9
alias pip=pip3

export PATH="$HOME/.local/bin:$PATH"
```

굉장히 많은 내용이 담겨 있습니다. 여기서 우리가 주목해야 할 부분은 가장 마지막 줄의 `export` 와 중간중간 보이는 `alias` 입니다.

1) **`export`**

쉘에게 `PATH` 라는 환경변수의 값을 사용할 수 있도록 알려주는 역할입니다.

2) **`alias`**

특정 명령에 별칭을 부여하여 명령어를 간소화 하거나 대체할 때 사용합니다. 기본적으로 사용하는 명령어 중 `ls -alF` 라는 명령어는 `ll` 로 앨리어싱 되어 있기 때문에 같은 명령을 수행합니다.

3. Shell Script 기본 구조

드디어 쉘 스크립트를 직접 작성해보는 시간입니다. 많이 다뤄보았던 `Dockerfile` 을 쉘을 통해 작성해보도록 하겠습니다.

```
nano Dockerfile.sh
```

```
#!/bin/bash cat << _EOF_ FROM nginx:alpine WORKDIR /usr/share/nginx/html RUN
rm -rf ./* COPY ./* ./ ENTRYPOINT ["nginx", "-g", "daemon off;"] _EOF_
```

Dockerfile 의 내용을 출력해주는 쉘을 작성해보았습니다. 처음보는 키워드 들을 하나나 살펴보겠습니다.

- `<<` : 표준 입력값을 전달하기 위한 기호입니다.
- `_EOF_` : 이는 사실 정해진 값은 아닙니다. bash에서 사용하는 예약어가 아니라면 무엇이든 사용할 수 있지만 관례로 `_EOF_` 를 많이 사용합니다. 넘겨줄 입력값의 시작과 끝을 나타내기 위해 사용되며 **token** 이라고 부릅니다.

4. 변수, 환경변수, 상수

```
#!/bin/bash filename="springboot-sample-app.jar" MAINTAINER=$USER PORT="8080"
VOLUME_DIR="/tmp" cat << _EOF_ FROM openjdk:8-jdk-alpine as builder LABEL
maintainer=$MAINTAINER COPY gradlew . COPY gradle gradle COPY build.gradle .
COPY settings.gradle . COPY src src RUN chmod +x ./gradlew RUN ./gradlew
bootjar FROM openjdk:8-jdk-alpine COPY --from=builder build/libs/*.jar
$filename VOLUME $VOLUME_DIR EXPOSE $PORT ENTRYPOINT ["java", "-jar",
"/$filename"] _EOF_
```

쉘 스크립트 내에서 반복되는 값이나 한 번에 수정해야 하는 값은 변수를 선언해서 활용할 수 있습니다. 파일명 같은 경우에는 여러 번 동일한 이름으로 사용되는 경우 오타 등으로 발생하는 불상사를 막을 수 있습니다.

- **변수 : 소문자로 명명하여 값 대입**
- **환경변수 : bash 가 실행될 때 시스템 내부적으로 메모리에 로드되어 있는 값**
- **상수 : 스크립트 내에서 불변해야 하는 값 대입**

5. 분기

```
nano ./test.sh
```

```
#!/bin/bash if [ -f .bashrc ]; then echo ".bashrc 파일이 존재합니다." elif [ -f
.bash_profile ]; then echo ".bash_profile 파일이 존재합니다." else exit fi # 기본 형
태 if [조건]; then (실행 블록) else (실행 블록) fi
```

```
ggingmin@ubuntu_server:~$ ./test.sh
.bashrc 파일이 존재합니다.
```

- `-d` : 디렉토리 여부
- `-f` : 파일 존재 여부
- `-r` : 읽기 가능 여부
- `-w` : 쓰기 가능 여부
- `-x` : 실행 가능 여부

쉘 스크립트 조건문에는 위와 같은 옵션을 통해 간편하게 조건을 설정할 수 있습니다.

6. 종료 및 종료상태

```
nano ./test.sh
```

```
#!/bin/bash if [ -f .bashrc ]; then echo ".bashrc 파일이 존재합니다." exit 0 elif
[ -f .bash_profile ]; then echo ".bash_profile 파일이 존재합니다." exit 0 else
exit 1 fi
```

종료 시그널을 적절히 넣어주는 것은 쉘 스크립트를 작성할 때 아주 중요한 작업입니다. 값은 두 가지로 나뉩니다.

- `0` : 스크립트 실행이 성공
- `1` : 스크립트 실행이 실패

위의 값은 스크립트가 종료되면서 쉘에 반환되기 때문에 연결된 프로세스 실행에 큰 영향을 미치게 됩니다. 성공한 경우와 실패한 경우 다른 프로세스를 따르게 해야하기 때문이죠.

```
ggingmin@ubuntu_server:~$ ./test.sh
.bashrc 파일이 존재합니다.
ggingmin@ubuntu_server:~$ echo $?
0
```

7. 반복문

```
nano ./test2.sh chmod 755 ./test2.sh ./test.sh
```

```
#!/bin/bash count=0 if [ -f .bashrc ]; then echo ".bashrc 파일이 존재합니다." for i in $(cat ~/.bashrc); do count=$((count + 1)) echo "$count 번째 단어 $i 는 $(echo -n $i | wc -c) 글자입니다." done exit 0 else exit 1 fi # 기본 형태 for [요소] in [반복 가능 오브젝트]; do [실행블록] done
```

```
ggingmin@ubuntu_server:~$ ./test2.sh
.bashrc 파일이 존재합니다.
1 번째 단어 # 는 1 글자입니다.
2 번째 단어 ~/.bashrc: 는 10 글자입니다.
3 번째 단어 executed 는 8 글자입니다.
4 번째 단어 by 는 2 글자입니다.
5 번째 단어 bash(1) 는 7 글자입니다.
6 번째 단어 for 는 3 글자입니다.
7 번째 단어 non-login 는 9 글자입니다.
8 번째 단어 shells. 는 7 글자입니다.
9 번째 단어 # 는 1 글자입니다.
10 번째 단어 see 는 3 글자입니다.
11 번째 단어 /usr/share/doc/bash/examples/startup-files 는 42 글자입니다.
12 번째 단어 (in 는 3 글자입니다.
13 번째 단어 the 는 3 글자입니다.
14 번째 단어 package 는 7 글자입니다.
15 번째 단어 bash-doc) 는 9 글자입니다.
16 번째 단어 # 는 1 글자입니다.
17 번째 단어 for 는 3 글자입니다.
18 번째 단어 examples 는 8 글자입니다.
19 번째 단어 # 는 1 글자입니다.
20 번째 단어 If 는 2 글자입니다.
21 번째 단어 not 는 3 글자입니다.
22 번째 단어 running 는 7 글자입니다.
23 번째 단어 interactively, 는 14 글자입니다.
24 번째 단어 don't 는 5 글자입니다.
```

쉘 스크립트에서 반복문을 사용하는 방법은 `if` 와 마찬가지로 보통의 프로그래밍 언어와 크게 다르지 않습니다.

`echo -n $i | wc -c` 이 코드는 두 부분으로 나눌 수 있는데, 첫번째 코드 실행의 결과를 받아 바로 두번째 코드를 실행하기 위해 `|` 파이프라인을 사용하였습니다. `echo -n $i` 의 결과로 줄바꿈 문자가 제거된 단어가 `wc -c` 명령을 통해 최종적으로 글자수를 반환하게 된 것입니다.