셸 프로그래밍
Shell Programming

- 셸 프로그램 개요
- 입출력
- 셸 변수
- 함수
- 제어 구조
- 디버깅

셸 프로그램 개요

셸 프로그램

- 복잡한 명령어나 반복적인 명령어를 처리할 때는 셸 명령어들을 나열한 스크립트를 작성

Simple Example

- 첫 행은 항상 #!로 시작
  - 시스템에 해당 스크립트가 직접 실행 가능하다는 것을 알림
  - /bin/bash: Bash 셸로 명령어를 해석해야 함
- 나머지 행의 #는 주석(comments)
  - 주석은 실행에서 제외되며 스크립트를 설명하기 위해 사용

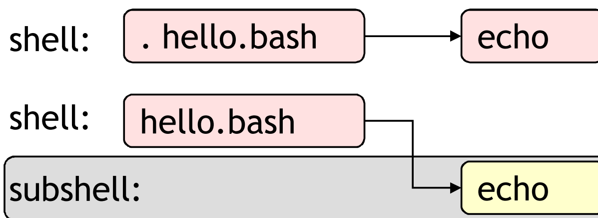
```
juyoon@localhost:~
[juyoon:~23]$ cat hello.bash
#!/bin/bash
#hello.bash
echo "Hello world!"
[juyoon:~24]$ hello.bash
-bash: ./hello.bash: 허가 거부됨
[juyoon:~25]$ bash hello.bash
Hello world!
[juyoon:~26]$ . hello.bash
Hello world!
[juyoon:~27]$ chmod +x hello.bash
[juyoon:~28]$ hello.bash
Hello world!
[juyoon:~29]$
```

셸 프로그램 개요

❑ 실행방법

- . 명령으로 실행
  - $. hello.bash
- 실행 권한을 부여한 후 직접 호출
  - $ chmod +x hello.bash
    $ hello.bash

❑ 직접 호출 시에는 하위 셸(subshell)을 생성하여 실행



전역환경변수를 부모로부터 복사하고 지역변수는 새로 초기화해서 사용

셸 프로그램 개요

❑ 셸 스크립트 기본 요소

- 변수 (variable)
- 함수 (function)
- 제어문 (control statement)
  - 조건문 (conditional branch)
  - 반복문 (iteration)
- 명령행 옵션 (command line option)
- 인터럽트 처리 (interrupt handling)

입출력

❑ 표준 출력

```
echo 문자열
```

❑ 표준 입력

```
read varname …
```

- 변수의 개수보다 입력 단어가 많으면 마지막 변수에 모두 할당
- 변수 이름 없이 사용하면 REPLY 변수에 입력 할당
- 기본적으로 행 단위 처리

```
juyoon@localhost:~/system/script
[juyoon:script/20]$ read a b
babo computer merong!
[juyoon:script/21]$ echo $a
babo
[juyoon:script/22]$ echo $b
computer merong!
[juyoon:script/23]$ read
babo computer merong!
[juyoon:script/24]$ echo $REPLY
babo computer merong!
[juyoon:script/25]$
```


입출력

❑ 파일에서 입력

- redirection을 활용한다.

```
juyoon@localhost:~/system/script
[juyoon:script/141]$ m readex
#!/bin/bash

cut -f1 -d: /etc/passwd > ids
cut -f5 -d: /etc/passwd > users
paste ids users > uu

egrep '^[axv]' uu > kusers
echo -e "ID₩tNAME"
while read id user; do
        echo -e "$id₩t$user"
done < kusers
[juyoon:script/142]$ readex
ID      NAME
adm     adm
vcsa    virtual console memory owner
apache  Apache
xfs     X Font Server
assistant
vision
```

셸 변수

❑ 셸 변수: bash 제공 내장 변수 + 사용자 지정 변수

❑ 변수명

- 예약어(reserved word)가 아니어야 한다.

```
if      then   else   elif   fi     case   esac
for     while  until  do     done   function
in      select        !      {      }      time
```

- 영문자로 시작하며 '_'와 영숫자로만 이루어진다.

❑ 변수값 설정

```
varname=value
```

- value는 문자열, 공백이 포함되면 '나 "로 싼다.

❑ 변수값 참조

- $varname 또는 ${varname}

셸 변수

❑ 변수 사용 범위

- 기본적으로 지역 변수, 즉 현재 실행 중인 셸 또는 스크립트에서만 효력을 가진다.
- export 명령으로 환경 변수가 된다. 즉, 하위 셸 또는 로그아웃 후 재실행하는 경우에도 사용할 수 있다.

❑ 읽기 전용 변수: readonly

- 변수의 값을 변경하지 못하게 만드는 셸 내장 명령
- 인자가 없으면 현재 설정된 읽기 전용 변수 확인

```
readonly [-af] [name[=value] …]
```

- 새로운 셸 또는 하위 셸이 실행될 때 readonly 속성은 상속되지 않는다.

셸 변수

```
[juyoon:juyoon/24]$ readonly OS=linux
[juyoon:juyoon/25]$ OS=windows
-bash: OS: readonly variable
[juyoon:juyoon/26]$ readonly
declare -ar BASH_VERSINFO='([0]="2" [1]="05b" [2]="0" [3]="1" [4]="release" [5]=
"i386-redhat-linux-gnu")'
declare -ir EUID="726"
declare -r NAME="juyoon"
declare -r OS="linux"
declare -ir PPID="14759"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:ignoreeof:interactive
-comments:monitor:noclobber"
declare -ir UID="726"
```

셸 명령어 사용법을 잘 모르겠다? 게다가 man 해도 정보가 없다?
→ 셸 내장 명령어는 'help'로 찾아 본다!

셸 변수

❑ 위치 변수

셸 변수	의미
$$	명령을 실행하는 프로세스 번호
$#	명령어 줄에서의 변수의 수
$?	마지막으로 실행된 명령어의 exit 상태. 일반적으로 명령어 실행 후에는 0의 값을 반환한다.
$1, …, $9	명령어 줄 상에서의 위치에 따른 각 인수
$0	스크립트 이름
$*	명령어 줄 상에서의 모든 인수를 포함하는 하나의 문자열
$@	명령어 줄 상에서의 모든 인수를 나타내는 각 문자열, 즉 "$1" "$2" "$3" … (공백으로 구분)

셸 변수

❑ 위치 변수 예제

```
juyoon@localhost:~/system/script
[juyoon:script/53]$ m ex1
#!/bin/bash

echo "PID: $$"
echo "parameters:$@"
echo "$0: $1 $2 $3 $4"
echo "$# arguments"
echo $*
[juyoon:script/54]$ ex1
PID: 21054
parameters:
./ex1:
0 arguments

[juyoon:script/55]$ ex1 a b c
PID: 21055
parameters:a b c
./ex1: a b c
3 arguments
a b c
[juyoon:script/56]$
```


셸 변수

❑ 문자열 연산

연산자	의미	목적
${var:-word}	var이 존재하고 값이 널이 아니면 값을 반환, 아니면 word 반환	정의하지 않은 변수 사용 시 오류 대신 기본값 사용
${var:=word}	위와 같으나 위치변수와 특수변수 제외	
${var:?mesg}	var이 존재하고 값이 널이 아니면 값을 반환, 아니면 mesg 출력하고 현재 명령 무시	정의하지 않은 변수 사용 시 오류 메시지 출력
${var:+word}	var이 존재하고 값이 널이 아니면 word 반환, 아니면 널 반환	변수의 존재 유무 검사
${var:offset} ${var:offset:length}	var의 값(문자열)에서 offset부터 length까지 부분문자열 반환. offset은 0부터 센다.	문자열의 일부분 사용

셸 변수

❑ 문자열 연산 예제

juyoon@linda: ~/test/system/shell_scripts

```
juyoon@linda:~/test/system/shell_scripts$ m ex2
#!/bin/bash

# 명령행에 입력된 파일명을 이용해 새 파일 이름을 만든다.
# 이름이 입력되지 않았으면 오류 메시지를 출력하고 종료한다.

names=${1:?"명단 파일 이름이 입력되지 않았습니다.
Usage: ex2 namefile scorefile"}

# 점수 파일 이름이 입력되지 않았으면 score를 사용한다.
score=${2:-score}
newfile=$names.$score

# 학생 명단과 점수 파일을 붙여 새 파일을 만든다.

paste $names $score > $newfile

# 새로운 파일을 점수 순서대로 정렬한다.

sortfile="$newfile.sort"
sort -nr -k5 $newfile > $sortfile

juyoon@linda:~/test/system/shell_scripts$
```


셸 변수

❑ 패턴 비교

연산자	의미
${var#pattern}	pattern과 var 값을 앞부터 비교하여 일치하는 가장 짧은 부분을 삭제하고 나머지 반환
${var##pattern}	pattern과 var 값을 앞부터 비교하여 일치하는 가장 긴 부분을 삭제하고 나머지 반환
${var%pattern}	pattern과 var 값을 끝부터 비교하여 일치하는 가장 짧은 부분을 삭제하고 나머지 반환
${var%%pattern}	pattern과 var 값을 끝부터 비교하여 일치하는 가장 긴 부분을 삭제하고 나머지 반환
${var/pattern/string}	pattern과 var 값을 비교하여 일치하는 가장 긴 부분을 string으로 대체. 처음 하나만 바꾼다.
${var//pattern/string}	pattern과 var 값을 비교하여 일치하는 가장 긴 부분을 string으로 대체. 모든 일치하는 부분을 바꾼다.

셸 변수

❑ 명령 대체 (command substitution)

- 명령의 표준 출력을 변수 값처럼 사용

```
$(command)
```

```
`  `
```

```
[juyoon:script/109]$ m subs
#!/bin/bash
# pwd 명령 수행 결과를 mypwd 변수에 저장
mypwd=$(pwd)
echo $mypwd
# newline 출력
echo -e -n ₩₩n
# pattern 연산 결과 출력
echo ${mypwd##/*/}
echo ${mypwd#/*/}
echo ${mypwd%/*/}
echo ${mypwd%%/*/}
echo ${mypwd/home/myhome}
# 또다른 명령 수행 방법과 결과
echo -e -n "₩n"
echo `ls $HOME`
[juyoon:script/110]$ subs
/home/juyoon/system/script

script
juyoon/system/script
/home/juyoon/system/script
/home/juyoon/system/script
/myhome/juyoon/system/script

5 cprogramming hello.bash man-solution sys
tem system.tar.gz
```


함수

❑ 함수 - script within a script

- 함수 이름을 명령어처럼 사용
  - 셸 메모리에 미리 저장 - 속도가 빠르다.
  - 내장 명령이나 스크립트보다 우선 순위가 높다.
- 함수 정의

```
function func_name {
    shell commands
}
```

```
func_name() {
    shell commands
}
```

  - 중괄호 '{' 다음엔 반드시 공백
- 함수 정의의 삭제

```
unset -f func_name
```

함수

❑ 변수의 참조 범위

- 지역 변수
  - 함수 내에서 정의된 변수
  - 함수 내에서만 의미가 있고 함수를 벗어나면 의미가 없다.
- 전역변수
  - 전체 스크립트에서 의미가 있는 변수

```
#!/bin/bash

function afunc {
        echo in function: $0 $1 $2
        var1="in function"
        echo var1: $var1
}

var1="outside function"
echo var1: $var1
echo $0: $1 $2
afunc funcarg1 funcarg2
echo var1: $var1
echo $0: $1 $2
```

지역 변수 ($0 $1 $2)

전역 변수 (var1)

위치 변수만 지역적!

함수

❑ 변수의 참조 범위

- 일반 변수를 지역 변수로 만들고 싶으면? → local

```
#!/bin/bash

function afunc {
        local var1
        echo in function: $0 $1 $2
        var1="in function"
        echo var1: $var1
}

var1="outside function"
echo var1: $var1
echo $0: $1 $2
afunc funcarg1 funcarg2
echo var1: $var1
echo $0: $1 $2
```

❑ 명령행에서 직접 함수를 사용하고 싶으면?

- .bashrc에 넣거나 명령행에서 함수 정의

함수

❑ 프로그램 실행 우선 순위

- 같은 이름의 명령, 함수 등이 있을 때
  1) alias
  2) function, if, for 등의 키워드
  3) 함수
  4) 내장 명령 (cd, type, pwd, …)
  5) 스크립트와 프로그램 (PATH 순서대로)

❑ 순서 바꾸기

- alias → unalias로 삭제
- command: 내장 명령과 일반 명령만 사용
- builtin: 내장 명령만 사용
- enable -n: 내장 명령 금지

함수

❑ 이름이 어떻게 정의되어 있는가? – type 명령

```
type [-all | -path | -type] name
```

juyoon@localhost:~/system/script

```
[juyoon:script/14]$ function ls {
> /bin/ls -la "$@"
> }
[juyoon:script/15]$ type -all ls
ls is aliased to `ls -sFC'
ls is a function
ls ()
{
    /bin/ls -la "$@"
}
ls is /bin/ls
[juyoon:script/16]$ type -path ls
[juyoon:script/17]$ type -type ls
alias
[juyoon:script/18]$ type -all -path ls
/bin/ls
[juyoon:script/19]$ type -all -type ls
alias
function
file
```

예제

❑ pushd, popd

- 셸 내장명령
- 작업 디렉터리를 스택에 저장해 두었다가 되돌아갈 때 사용
- dirs: 디렉터리 스택을 보여 준다.
- pushd *dir*
  - 스택에 현재 디렉터리를 push
    → *dir*로 이동하고 *dir*을 스택에 push
  - 연속 실행 시에는 *dir*만 push
- popd: 스택 top에 있는 디렉터리를 pop해서 삭제하고 다음 디렉터리가 top이 되며 그 곳으로 이동
- 관련 내장 변수: DIRSTACK

예제

❑ 실행 예시

명령	스택의 내용	결과 디렉터리
pushd system	~/system ~	~/system
pushd /etc	/etc ~/system ~	/etc
popd	~/system ~	~/system
popd	~	~
popd	<empty>	(error)

❑ 옵션

- +N, -N: N개 만큼 rotate

예제

❑ pushd, popd를 직접 구현해 보자!

- 두 함수에 공통인 스택을 사용해야 하므로 환경변수를 설정한다.

```
$ DIR_STACK=""
$ export DIR_STACK
```

DIR_STACK은 디렉터리 이름들을 저장할 문자열

```
pushd() {
        dirname=$1
        DIR_STACK="$dirname ${DIR_STACK:-$PWD' '}"
        cd ${dirname:?"missing directory name"}
        echo "$DIR_STACK"
}

popd() {
        DIR_STACK=${DIR_STACK#* }
        cd ${DIR_STACK%% *}
        echo "$PWD"
}
```

- 기존 DIR_STACK에 현재 입력 디렉터리를 덧붙인다.
- DIR_STACK 맨 앞 단어(공백 구분)를 잘라낸다.
- DIR_STACK 맨 앞 단어로 이동한다.

- 어떤 문제가 있는가?

제어 구조

❑ Bash에서 사용되는 제어구조

- if/then/else
- for
- while
- until
- case
- select
- break/continue/exit

제어 구조

❑ if/then/else

- if 다음의 조건식이 참(true)이면 then 뒤의 명령어들을 실행하라는 의미
- 조건이 비교될 때 결과값이 0이면 참으로 인식
  - 명령어가 오류 없이 실행되면 0을 반환 (exit code)

```
if condition
then
    statements
[elif condition
  then statements ...]
[else
    statements]
fi
```

<사용 예>

```
pushd() {
        dirname=$1
        if cd ${dirname:?"missing directory name"}
        then
                DIR_STACK="$dirname ${DIR_STACK:-$PWD' '}"
                echo $DIR_STACK
        else
                echo still in $PWD
        fi
}
```

제어 구조

❑ 조건 검사

- [conditions] 또는 test 문 사용

❑ 문자열 비교

연산자	참인 경우
str1 = str2	str1과 str2가 같다.
str1 != str2	str1과 str2가 같지 않다.
str1 < str2	str1이 str2보다 작다.
str1 > str2	str1이 str2보다 크다.
-n str1	str1이 널이 아니다.
-z str1	str1이 널이다. (길이가 0)

- 사용 예

```
popd() {
        if [ -n "$DIR_STACK" ]; then
                DIR_STACK=${DIR__STACK#* }
                cd ${DIR_STACK%% *}
                echo "$PWD"
        else
                echo "stack empty, still in $PWD"
        fi
}
```

제어 구조

❑ 파일 속성 검사

연산자	참인 경우
-d file	file이 존재하고 디렉터리인 경우
-e file	file이 존재하는 경우
-f file	file이 존재하고 일반 파일인 경우
-r file	file이 읽기 권한이 있는 경우
-s file	file이 존재하고 비어 있지 않은 경우
-w file	file의 쓰기 권한이 있는 경우
-x file	file의 실행 권한이 있는 경우 또는 디렉터리 검색 권한이 있는 경우
-O file	file의 소유자인 경우
-G file	file의 그룹 ID가 현재 사용자 그룹과 일치하는 경우
file1 -nt file2	file1이 file2보다 나중에 작성된 경우
file1 -ot file2	file1이 file2보다 먼저 작성된 경우

제어 구조

❑ 논리 연산

연산자	의미
!	부정 (NOT)
&& (-a)	AND
|| (-o)	OR

- **사용 예**

```
pushd () {
        dirname=$1
        if [ -n "$dirname" ] && [ ₩( -d "$dirname" ₩) -a ₩
                ₩( -x "$dirname" ₩) ]; then
                DIR_STACK="$dirname ${DIR_STACK:-$PWD' '}"
                cd $dirname
                echo "$DIR_STACK"
        else
                echo "still in $PWD"
        fi
}
```

제어 구조

❑ 파일 및 논리 연산 예제

```
if [ ! -e "$1" ]; then
        echo "file $1 does not exist."
        exit 1
fi

if [ -d "$1" ]; then
        echo -n "$1 is a directory that you may "
        if [ ! -x "$1" ]; then
                echo -n "not "
        fi
        echo "search."
elif [ -f "$1" ]; then
        echo "$1 is a regular file."
else
        echo "$1 is a special type of file."
fi

if [ -O "$1" ]; then
        echo 'you own the file.'
else
        echo 'you do not own the file.'
fi

if [ -r "$1" ]; then
        echo 'you have read permission on the file.'
fi

if [ -w "$1" ]; then
        echo 'uyou have write permission on the file.'
fi

if [ -x "$1" -a ! -d "$1" ]; then
        echo 'you have execute permission on the file.'
fi
```


제어 구조

❑ 산술 연산

- 변수의 속성(type) 정의 : declare

옵션	의미
-a	배열
-f	함수명만 사용
-F	정의하지 않은 함수명 출력
-I	정수
-r	읽기 전용
-x	export할 변수 표시

  - declare를 사용한 변수는 모두 함수 내 지역 변수
- 산술 연산자 : +, -, *, /, %, <<, >>, &, |, ~, !, ^
- 관계 연산자 : <, >, <=, >=, ==, !=, &&, ||

제어 구조

❑ 산술 연산

- 계산식을 변수에 할당: let *intvar=expression*
- expression은 ' '로 묶는 것이 좋다.

❑ 검사(test)용 관계 연산자

- 숫자를 사용할 때는 \(… \)로 묶거나, ((…)) 구문을 사용한다.

연산자	의미
-lt	작다.
-le	작거나 같다.
-eq	같다.
-ge	크거나 같다.
-gt	크다.
-ne	같지 않다.

제어 구조

❑ For

- 횟수가 아니라 문자열 리스트에 의해 반복

```
for name [ in list ]
do
   statements that can use $name ...
done
```

- 사용 예

```
#!/bin/bash

for dir in ${*:-.}; do
        if [ -e $dir ]; then
                result=$(du -s $dir | cut -f 1)
                let total=$result*1024
                echo -n "Total for $dir = $total bytes"
                if [ $total -ge 1048576 ]; then
                        echo " ($((total/1048576)) Mb)"
                elif [ $total -ge 1024 ]; then
                        echo " ($((total/1024)) Kb)"
                fi
        fi
done
```

제어 구조

- 순환 (recursion) 예제

```
#!/bin/bash

recdir() {
        tab=$tab$singletab
        for file in "$@"; do
                echo -e $tab$file
                thisfile=$thisfile/$file

                if [ -d "$thisfile" ]; then
                        recdir $(command ls $thisfile)
                fi
                thisfile=${thisfile%/*}
        done
        tab=${tab%₩₩t}
}

recls() {
        singletab="₩₩t"

        for tryfile in "$@"; do
                echo $tryfile
                if [ -d "$tryfile" ]; then
                        thisfile=$tryfile
                        recdir $(command ls $tryfile)
                fi
        done
        unset dir singletab tab
}

recls "$@"
```


제어 구조

- While/Until

```
while condition
do
    statements …
done
```

```
until condition
do
    statements …
done
```

- while: 조건이 참인 동안 statements 실행
- until: 조건이 거짓인 동안 statements 실행
- 예제

```
while ! cp $1 $2; do
    echo 'Attempt to copy failed. waiting...'
    sleep 5
done
```

```
until cp $1 $2; do
    echo 'Attempt to copy failed. waiting...'
    sleep 5
done
```

제어 구조

❑ Case

- 단일 문자열의 값에 근거한 다중 선택의 분기를 지원

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
    …
esac
```

- 패턴 검색에는 와일드 카드(?, *) 사용 가능

제어 구조

❑ Case 예제

- 명령행 인자의 수에 따라 cd 동작을 다르게 한다.
  - 0개, 1개면 내장 명령 cd 그대로 실행
  - 2개면 (cd old new), old 이름을 찾아 new로 바꾼 다음에 이동

```
cd()
{
    case "$#" in
        0 | 1)  builtin cd $1 ;;
        2    )  newdir=$(echo $PWD | sed -e "s:$1:$2:g")
                case "$newdir" in
                    $PWD)   echo "bash: cd: bad substitution" >&2 ; return 1 ;;
                    *   )   builtin cd "$newdir" ;;
                esac ;;
        *    )  echo "bash: cd: wrong arg count" 1>&2 ; return 1 ;;
    esac
}
```

제어 구조

❑ Select

- Korn shell과 bash에만 있음.

```
select name [in list]
do
   statements that can use $name ...
done
```

- list에 있는 각 항목으로 된 메뉴 생성
  - list가 없으면 "$@”가 default
  - 메뉴 형식은 번호
- name 변수에 선택한 내용을 저장하고 선택한 번호는 REPLY 변수에 저장
- statements를 실행
- 무한 루프 가능

제어 구조

❑ Select 예제

```
#!/bin/bash

OPTIONS="Hello Quit"
select opt in $OPTIONS; do
if [ "$opt" = "Quit" ]; then
        echo done
        exit
elif [ "$opt" = "Hello" ]; then
        echo Hello $USER
else
        clear
        echo bad option
fi
done
```

디버깅

❑ 스크립트 작성 시간의 많은 부분을 오류 찾는데 소모

- 가장 단순한 방법은 echo 명령어를 이용하는 것
- 셸은 자체적으로 디버그 모드를 포함
  - 문법적인 오류만을 체크

❑ 디버깅 옵션 사용

set -o	명령행	동작
noexec	-n	명령을 실행하지 않고 문법 오류만 검사
verbose	-v	명령을 실행하기 전 화면에 출력
xtrace	-x	명령행을 처리한 다음 화면에 출력

디버깅

❑ -x 옵션 사용 예

```
juyoon@localhost:~/system/script/examples/chapter5
[juyoon:chapter5/144]$ bash -x fileinfo fileinfo
+ '[' '!' -e fileinfo ']'
+ '[' -d fileinfo ']'
+ '[' -f fileinfo ']'
+ echo 'fileinfo is a regular file.'
fileinfo is a regular file.
+ '[' -O fileinfo ']'
+ echo 'you own the file.'
you own the file.
+ '[' -r fileinfo ']'
+ echo 'you have read permission on the file.'
you have read permission on the file.
+ '[' -w fileinfo ']'
+ echo 'you have write permission on the file.'
you have write permission on the file.
+ '[' -x fileinfo -a '!' -d fileinfo ']'
+ echo 'you have execute permission on the file.'
you have execute permission on the file.
[juyoon:chapter5/145]$
```

- '+' 프롬프트는 PS4 변수의 값
  - LINENO 변수를 활용하면 좋다. (실행 중인 행 번호 출력)
    export PS4='$LINENO: '

Uncovered Features

❑ 다루지 않은 것들

- trap
  - 발생 신호(signal)에 따라 인터럽트 처리
- Array
  - 변수명에 [ ] 붙여 배열로 사용
- 실수(floating point number) 연산
- 그 외 세밀한 여러 요소

❑ 더 공부하고 싶은 사람은…

- 셸 스크립트 관련 서적 참고
- http://kldp.org/HOWTO/html/Adv-Bash-Scr-HOWTO (원본: http://tldp.org/LDP/abs/html)

파일 시스템 작업

❑ 명령어: chmod

- 파일의 모드 바꾸기 (change mode)

```
chmod [options] mode[,mode] …  file …
```

```
chmod [options] octal-mode file …
```

- Mode 표현
  - 8진수(octal-mode): r, w, x를 각 하나의 비트로 표현

```
755 ← 111101101 ← rwxr-xr-x
```

  - 기호 표현 : 대상 [+-=][권한]
    - 대상: u(owner), g(group), o(other), a(all)
    - +-= : 권한 추가(+), 권한 삭제(-), 지정한 것만 남기고 삭제(=)
    - 권한: r(read), w(write), x(execute), X(디렉터리거나 권한이 이미 있을 때만 권한 유효), s(set UID 또는 GID), t(sticky bit)
- 주요 옵션
  - -R : Recursive. Subdirectory의 파일들도 모두 변경