

커뮤니티 OS/AUTOMATE

최국현

2022/09

DAY 1

오픈소스

리눅스와 오픈소스 관계

오픈소스

오픈소스는 **리차드 스톨만**이 GNU도구 및 라이선스를 선언하였음.

리눅스 토발즈는 GNU도구 중 **리눅스 커널(Linux Kernel)**를 만들었다.

리차드 스톨만은 **Hurd**라는 커널을 만들었지만, 선진적인 디자인의 **마이크로 커널 디자인(Micro Kernel)**를 사용하여서, 당시 성능으로는 구현 및 사용하기가 어려운 부분이 많았음.

리눅스와 오픈소스

리눅스의 대다수 소프트웨어는 GNU기반의 소프트웨어를 사용하고 있다.

GNU와 Unix소프트웨어와 큰 차이점은 없지만, 라이선스 및 커뮤니티 기반의 생태계를 좀 더 강하게 가지고 있다.

오픈소스에서 많이 비교되는 대상인 BSD소프트웨어와 제일 큰 차이점은 BSD는 유닉스 코드 기반으로 구성되어 있어 있

리눅스 쉘 및 커널

user-space, kernel-space

Shell POSIX

POSIX에서 명시한 echo명령어는 불행하게도 UNIX System V, BSD 그리고 GNU기반에 따라서 각기 다르게 동작한다.

예를 들어서 GNU echo명령어는 표준은 UNIX System V 혹은 BSD와 다르게 동작한다. (완전 다르지 않는다. 조금씩 결과가 다르게 나온다)

그러한 이유로, echo명령어 대신 호환성을 높이기 위해서 printf명령어 사용하는 것을 권장한다.

BASH 무엇인가?

셸은 사용자가 커널 및 사용자 프로그램 사이에서 서로 대화 할 수 있도록 해주는 도구.

셸은 ssh같은 도구가 아님. 컴퓨터와 대화하기 위한 도구. 좀 더 정확하게는 시스템 호출.

BASH 무엇인가?

셸은 TTY가 아니다. 많은 사용자들이 오해하고 있는게 TTY == SHELL라는 오해를 가지고 있음.

PuTTY/iTerm/Xterm와 같은 터미널 도구가 아님.

Bash는 명령어 라인(CLI)도구가 아님. Bash는 대화형 모드를 가지고 있으며 이를 통해서 프로그램들을 사용함. 예를 들어서 ls같은 명령어.

BASH 무엇인가?

거의 대다수의 리눅스는 bash가 설치되어 있음.

다른 말로 **GNU/Linux**는 **bash가 표준**이라고 생각해도 됨.

예를 들어서 **FreeBSD/NetBSD/OpenBSD** 그 외 다른 유닉스 들에서는 bash가 아닌 csh, tsh를 사용하기도 한다. 하지만 bash사용이 가능하며, 기존에 사용하였던 csh,tsh에서 사용하던 스크립트는 여전히 bash에서 거의 수정 없이 사용이 가능.

프로토타입

프로토타입

bash기반으로 간단하게 스크립트 프로그래밍 가능하다. bash의 목표 중 하나가 “hack together”

대화형 관리 시스템

bash는 borne-style셸 스타일로 리눅스 및 유닉스 에서 손쉽게 명령어를 사용할 수 있도록 해준다. 양쪽에서 서로 사용이 가능하기 때문에 손쉽게 이식이 가능하다.

오토메이션

자주 사용하는 명령어를 스크립트 기반으로 작성하여 간단하게 실행 및 수행이 가능한 방법.

프로그램 상화작용

한 개 이상의 프로그램을 쉘 스크립트를 통해서 동시에 사용 할 수 있도록 한다.

파일이나 혹은 표준 입출력을 통해서 데이터를 서로 전달하여 프로세싱 할 수 있다.

이를 보통 **파이핑(Piping)**라고 한다.

BASH가 할 수 있는 것.

필터링 그리고 입력 변환: 어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다.

이러는 경우 bash는 다른 프로그램을 통해서 다시 데이터 재처리를 해준다.

예를 들어서 awk, sed같은 프로그램들을 사용하여 원하는 형태로 출력 내용을 다시 재가공하여 전달한다.

베이직 패턴 문자열 매칭

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다.

이를 통해서 파일이름, 경로 이름을 **파라미터(parameter)**로 확장 가능하다.

우리는 이 기능을 **와일드카드(wildcard)**라고 부른다.

BASH 이동성

bash는 다양한 유닉스 및 리눅스 시스템에 패키징이 되어있다.

이러한 이유로 POSIX를 셸 스크립트를 표준적으로 지원하며, 각기 다른 런타임 환경에서 사용하는 경우 bash는 강력한 도구가 될 수 있다.

마치, 자바처럼 셸 프로그램이 이식 되어 있으면 손쉽게 사용이 가능하다.

도움말

bash를 사용하다가 어려운 부분이 있으면 다음과 같은 방법으로 도움을 받을 수 있다.

내부 명령어를 통해서 사용법 확인이 필요한 경우 아래처럼 실행한다.

bash\$ help printf

외부 명령어를 통해서 사용법 확인이 필요한 경우 아래처럼 실행한다.

bash\$ man printf

리눅스 커널

리눅스 커널의 정확한 명칭은 GNU/Linux이다. 실제로 리눅스의 커널은 두 가지가 있다.

GNU/Linux

현재 제일 많이 사용하는 모놀리식 타입의 커널. 윈도우 서버와 마찬가지로 완전한 모놀리식 타입이 아닌 현재는 마이크로 커널의 타입도 어느 정도 도입이 되어서 유연하게 모듈 운영이 가능하다.

GNU/Hurd

허드 커널은 리처드 스톨만이 만들었으며, 여전히 개발이 되고 있다. 다만, 너무 빠른 게 디자인의 적용 및 릴리즈가 되어서 성능 문제 및 아키텍트 적용에 많은 문제가 있었다.

현재 허드 커널을 사용자가 원하는 경우, 데비안 리눅스에서 사용이 가능하나, 아직 많은 장치를 지원하지 않기 때문에 사용률이 낮다.

리눅스 커널

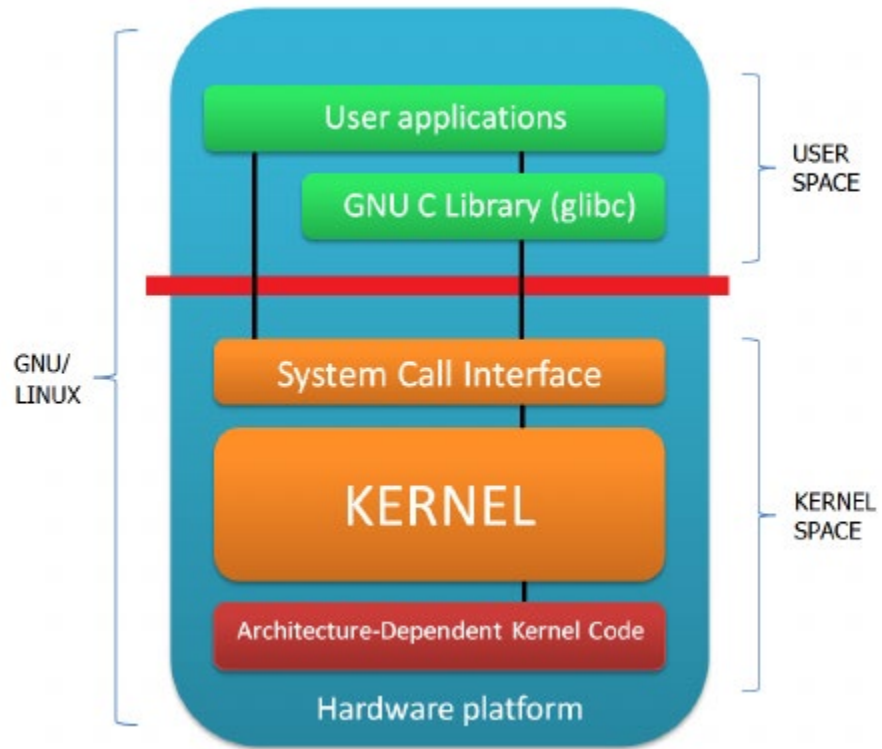
리눅스 커널의 정확한 명칭은 GNU/Linux이다. 실제로 리눅스의 커널은 두 가지가 있다.

GNU/Hurd

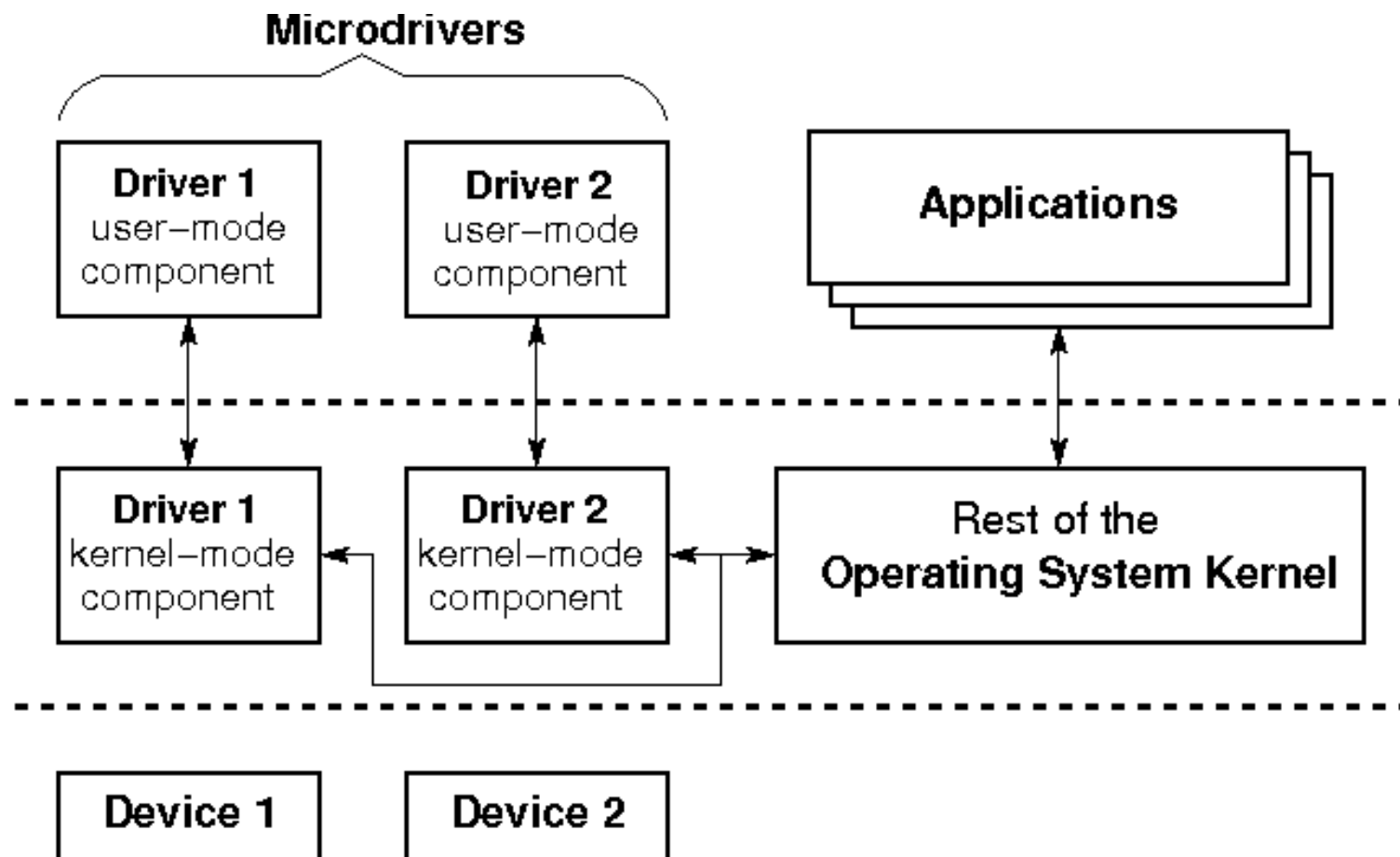
허드 커널은 리처드 스톨만이 만들었으며, 여전히 개발이 되고 있다. 다만, 너무 빠르게 디자인 적용 및 릴리즈가 되어서 성능 문제 및 아키텍트 적용에 많은 문제가 있었다.

현재 허드 커널은 사용자가 원하는 경우, 데비안 리눅스에서 사용이 가능하나, 아직 많은 장치를 지원하지 않기 때문에 사용율이 낮다.

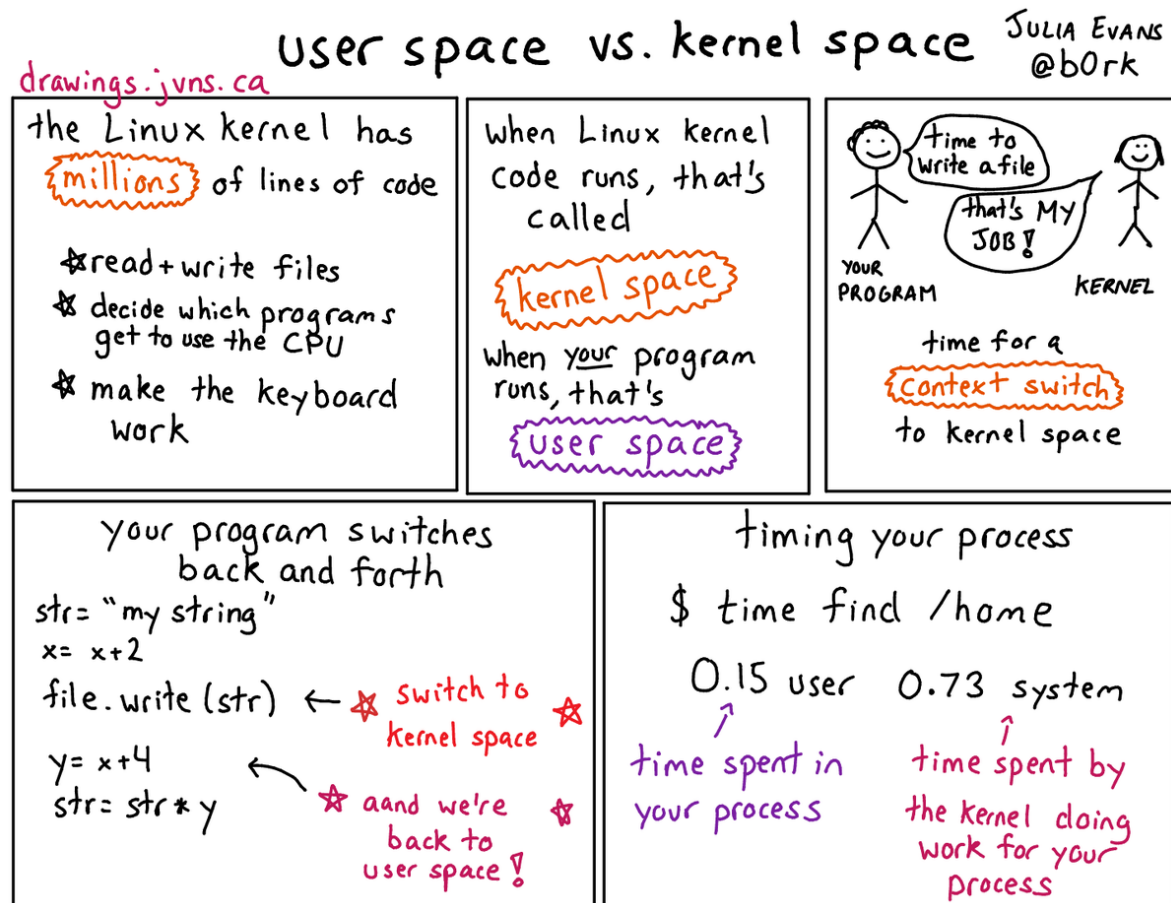
리눅스 커널



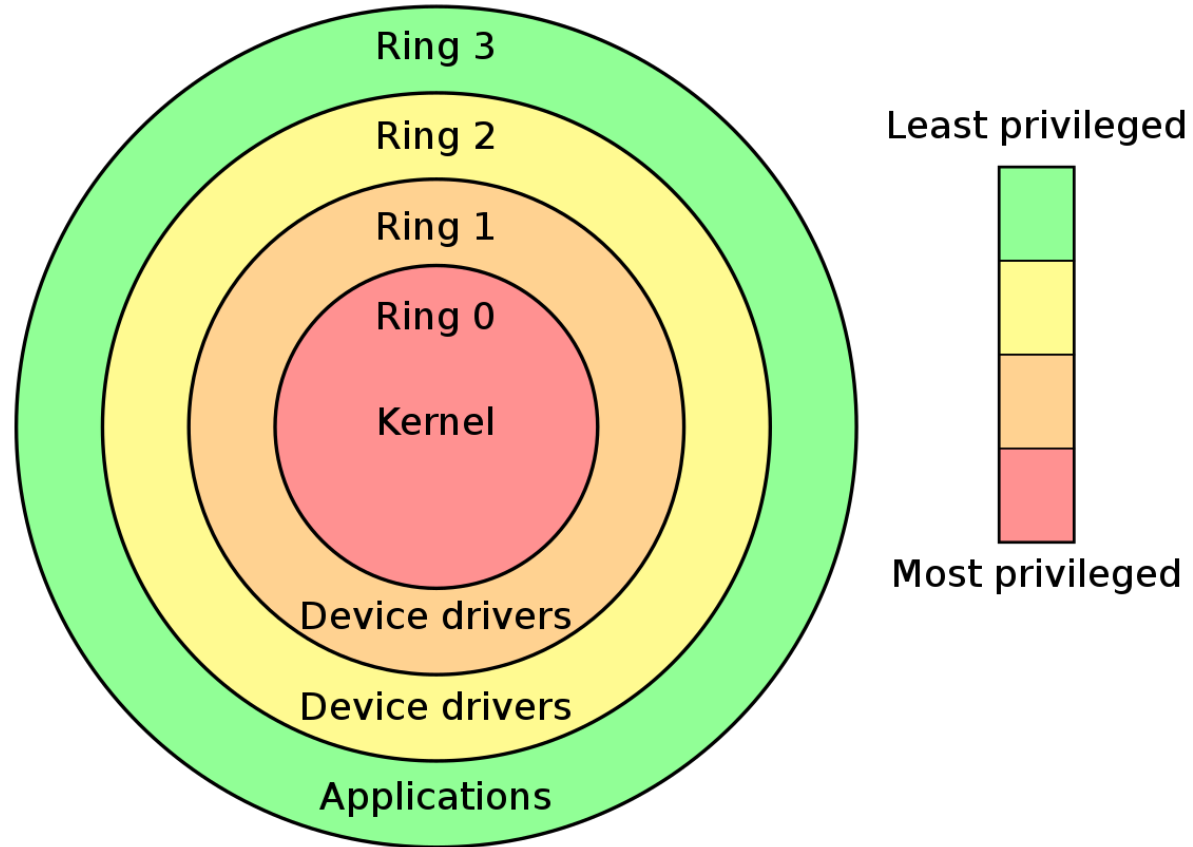
리눅스 커널



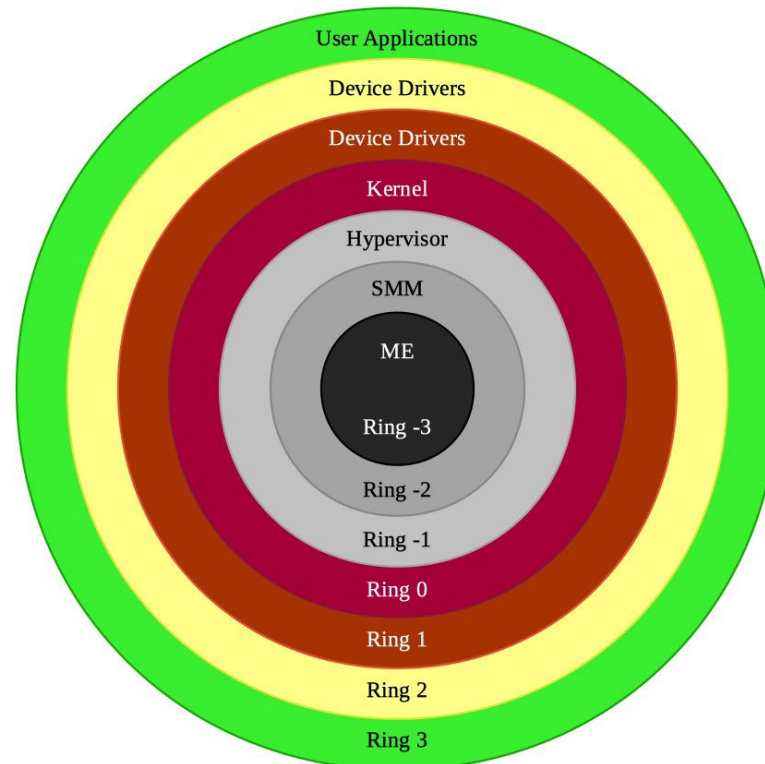
리눅스 커널



리눅스 커널



리눅스 커널



IA Negative Rings

자주 사용하는 리눅스 명령어

the useful GNU/Linux Commands

웹 기초

웹 기초 내용

KSH

KSH은 유닉스에서 많이 사용하는 쉘 중 하나.
보통 다음과 같은 유닉스 시스템에서 많이 사용한다.

BSD계열

Unix 계열(HP, AIX, Solaris)

KSH/BASH 차이점

ksh와 bash의 큰 차이점은 실제로는 거의 없다.

하지만, [Conflicts between ISO/IEC 9945 \(POSIX\) and the Linux Standard Base](#)의 내용에 충돌이 되기 때문에 서로 호환이 되지 않는 부분이 있다.

예를 들어서 LSB에서는 /usr/xpg4/bin를 사용하지 않으며 또한 솔라리스도 이러한 POSIX규격을 따르지 않는다.

리눅스 및 솔라리스 경우 2004년도 기준으로 양쪽 진영은 표준 사양을 변경하였다. 이를 통해서 리눅스 및 솔라리스 같은 유닉스와 스크립트 격차를 많이 줄였다.

Shell POSIX

POSIX에서 명시한 echo명령어는 불행하게도 UNIX System V, BSD 그리고 GNU기반에 따라서 각기 다르게 동작한다.

예를 들어서 GNU echo명령어는 표준은 UNIX System V 혹은 BSD와 다르게 동작한다. (완전 다르지 않는다. 조금씩 결과가 다르게 나온다)

그러한 이유로, echo명령어 대신 호환성을 높이기 위해서 printf명령어 사용하는 것을 권장한다.

BASH 무엇인가?

셸은 사용자가 커널 및 사용자 프로그램 사이에서 서로 대화 할 수 있도록 해주는 도구.

셸은 ssh같은 도구가 아님. 컴퓨터와 대화하기 위한 도구. 좀 더 정확하게는 시스템 호출.

BASH 무엇인가?

셸은 **TTY**가 아니다. 많은 사용자들이 오해하고 있는게 **TTY == SHELL**라는 오해를 가지고 있음.

PuTTY/iTerm/Xterm와 같은 터미널 도구가 아님.

Bash는 명령어 라인(CLI)도구가 아님. Bash는 대화형 모드를 가지고 있으며 이를 통해서 프로그램들을 사용함. 예를 들어서 ls같은 명령어.

BASH 무엇인가?

거의 대다수의 리눅스는 bash가 설치되어 있음.

다른 말로 **GNU/Linux**는 **bash가 표준**이라고 생각해도 됨.

예를 들어서 **FreeBSD/NetBSD/OpenBSD** 그 외 다른 유닉스 들에서는 bash가 아닌 **csh, tsh**를 사용하기도 한다. 하지만 bash사용이 가능하며, 기존에 사용하였던 **csh,tsh**에서 사용하던 스크립트는 여전히 bash에서 거의 수정 없이 사용이 가능.

셸 기초

Bash 버전 확인하는 방법은 다음과 같다.

```
bash$ declare --p BASH  
BASH="/usr/local/bin/bash"
```

chsh에서 bash로 변경하는 경우 아래와 같이 변경이 가능하다.

```
$ chsh -s /usr/local/bin/bash
```

셸 기초

리눅스 시스템에서 사용하는 셸 목록은 아래와 같이 확인이 가능하다.

```
$ chsh -l
```

설치가 안되어 있으면,

```
# yum install chsh -y
```

프로토타입

프로토타입

bash기반으로 간단하게 스크립트 프로그래밍 가능하다. bash의 목표 중 하나가 “hack together”

대화형 관리 시스템

bash는 borne-style셸 스타일로 리눅스 및 유닉스 에서 손쉽게 명령어를 사용할 수 있도록 해준다. 양쪽에서 서로 사용이 가능하기 때문에 손쉽게 이식이 가능하다.

오토메이션

자주 사용하는 명령어를 스크립트 기반으로 작성하여 간단하게 실행 및 수행이 가능한 방법.

프로그램 상화작용

한 개 이상의 프로그램을 쉘 스크립트를 통해서 동시에 사용 할 수 있도록 한다.

파일이나 혹은 표준 입출력을 통해서 데이터를 서로 전달하여 프로세싱 할 수 있다.

이를 보통 **파이핑(Piping)**라고 한다.

BASH가 할 수 있는 것.

필터링 그리고 입력 변환: 어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다.

이러는 경우 bash는 다른 프로그램을 통해서 다시 데이터 재처리를 해준다.

예를 들어서 awk, sed같은 프로그램들을 사용하여 원하는 형태로 출력 내용을 다시 재가공하여 전달한다.

베이직 패턴 문자열 매칭

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다.

이를 통해서 파일이름, 경로 이름을 **파라미터(parameter)**로 확장 가능하다.

우리는 이 기능을 **와일드카드(wildcard)**라고 부른다.

BASH 이동성

bash는 다양한 유닉스 및 리눅스 시스템에 패키징이 되어있다.

이러한 이유로 POSIX를 셸 스크립트를 표준적으로 지원하며, 각기 다른 런타임 환경에서 사용하는 경우 bash는 강력한 도구가 될 수 있다.

마치, 자바처럼 셸 프로그램이 이식 되어 있으면 손쉽게 사용이 가능하다.

도움말

bash를 사용하다가 어려운 부분이 있으면 다음과 같은 방법으로 도움을 받을 수 있다.

내부 명령어를 통해서 사용법 확인이 필요한 경우 아래처럼 실행한다.

bash\$ help printf

외부 명령어를 통해서 사용법 확인이 필요한 경우 아래처럼 실행한다.

bash\$ man printf

명령어 구조

bash

이미 알고 계시겠지만...

기본적인 내용이지만 좀 더 자세하게 기본 내용을 확인 하도록 한다.
만약 잘 아신다고 생각하시면...



간단한 명령어

bash에서 명령어 사용 방법은 다음과 같다.

```
$ mkdir -p testdir/subdir
```

매우 간단하다! :)")

간단한 명령어

하지만, 이러한 방식으로만 사용하지 않는다.

앞에서 이야기 하였던 쉘에서 사용하는 특수문자(metacharacters)가 있다. 예를 들어서 다음과 같다.

```
$ touch test file
```

```
$ ls -l
```

```
test
```

```
file
```

위와 같이 나온다.

여기에 좀 더 복잡하게 특수문자를 사용하면 다음처럼 결과가 나온다.

간단한 명령어

\$ touch report<SRV1>.txt

\$ touch report;Final.txt

위의 명령어를 실행을 하면 올바르게 실행이 되지 않는다.

왜?

bash 명령어 구조



설명해라 강사야!!

간단한 명령어

몇 가지 명령어를 더 테스트 해본다.

```
$touch $$$likeMoney.txt
```

```
$ ls
```

```
12234likeMoney.txt
```

역시 올바르게 실행이 되지 않는다. 왜 그럴까요?

간단한 명령어

이러한 문제를 해결 하기 위해서 **쿼트(Quoting/Quote)**을 사용해야 한다.

보통 다음과 같은 방법이 있다.

'<string>'

"<string>"

\<escape>\<escape>

특수문자 해결 방법

```
$ touch report\<SRV1\>.txt  
$ touch report\;Final.txt  
$ touch \$\$\$likeMoney.txt
```



그러면 만약 백 슬래시(\)를 출력하는 방법?

```
$ echo \\backslash\\  
\\backslash\\
```

그냥 두 번 입력하면 된다. 첫 번째는 **탈출문자(metacharacter)**, 두 번째는 문자로써 입력.

그러면 만약 백 슬래시(\)를 출력하는 방법?

```
$ echo backslash\
```

```
> hello
```

```
> ygl
```

```
backslashhelloygl
```

특수문자

\$ touch 'report document'

\$ touch 'report<SRV1>.txt'

\$ touch 'report;Final.txt'

\$ touch '\$\$\$report.txt'

싱글쿼트

싱글 쿼트 줄 내림(Single Quote)

```
$ echo 'quotes
```

```
> foo
```

```
> bar'
```

```
quotes
```

```
foo
```

```
bar
```

싱글쿼트

```
$ echo 'it\'s today'
```

```
$ echo 'it''s today'
```

```
$ echo 'it\'\'s today'
```

어떻게 출력이 될까요?

더블쿼트

```
$ echo "This is my shell: $SHELL"
```

```
$ echo 'This is my shell: $SHELL'
```

```
$ echo "The variable: \ $total"
```

```
$ echo "Back\\to\\back\\back\\slashes\\_world"
```

```
$ echo "hello $USER"!!'
```


백틱 (backticks)

```
$ echo "backticks: \`\`\`"
```

복합적 쿼트

```
$ echo Hello, \ "$USER"! Hell to the "$HOSTNAME"!"
```

Shell Exit Values

간단하게...

```
$ echo $?
```

```
$ true ; echo $?
```

```
$ false ; echo $?
```

1: False

0: True

Shell Exit Values

명령어가 없는 경우...

```
$ thereisnocommand ; echo $?
```

127

명령어 연속 실행 시 오류 중지

여러 명령어를 수행하는 방법은 두 가지가 있다.

```
$ cd ; rm -r /testdir ; ls
```

```
$ cd / && rm -r /testdir ; ls
```

첫 번째 방법은 오류가 발생하여도 다음명령어가 실행이 되며, 두 번째 방법은 첫 번째 명령어가 실행 중 문제가 발생하면 중지.

포그라운드/백그라운드

명령어 수행 시 포그라운드/백그라운드 실행은 다음과 같은 명령어로 실행이 가능하다.

\$ sleep 10

\$ sleep 10 &

포그라운드/백그라운드

백 그라운드로 실행이 된 명령어를 확인 하려면 아래와 같은 명령어로 확인이 가능하다.

\$ bg

기본 명령어

bash

bash 기본 명령어

셸에서 기본적으로 제공하는 **내부 명령어(internal command)**가 있다.

이 명령어를 통해서 좀 더 편하게 셸 환경을 꾸밀 수 있다.

빌트인 명령어

내부 명령어는 외부에 별도의 명령어 없이 실행이 가능한 내장형 명령어.

대표적인 명령어는 다음과 같다.

echo, type, source

런타임 명령어

런타임 명령어(runtime)는 기본적으로 쉘 스크립트(bash)명령어 기반으로 구성이 되어 있다.

이러한 도구들은 보통 `function()`, `alias`로 구성이 되어 있으며, 자체적으로 실행은 불가능하며, 쉘을 통한 조합 실행이 가능하다.

시스템 명령어

시스템 명령어는 파일 시스템에 독립적으로 구성이 되어 있는 프로그램이다.

이러면 명령어는 어떠한 패키지가 설치가 되어 있느냐에 따라서 다르다.

대표적인 명령어는 다음과 같다.

grep, ping, rm, ls, nmcli

셸 대표 명령어

type: 명령어 형식 확인. 셸에 alias로 구성이 되어 있는 경우 alias를 출력한다.

echo: 문자열 출력

printf: echo와 비슷하지만 포매팅 형식으로 출력

pwd: 셸의 현재 위치 출력

cd: 셸 위치 변경

set: 현재 셸에 설정된 값을 출력

declare: 변수나 함수(function)를 관리하는 명령어

test, [, [[: 정규식 구성, 대괄호로 구성이 된다.

대표 명령어

ls: 디렉터리 및 파일 출력

mv: 파일 이동 및 이름 변경

cp: 파일 복사

rm: 파일 및 디렉터리 제거, 다른 명령어로 rmdir이 있다.

grep: 문자열 검색하는 정규식 도구

cut: 행 단위로 자르는 명령어

wc: 줄 및 문자열 카운팅 도구

find: 파일이나 혹은 디렉터리를 특정 조건으로 찾는 도구

sort, uniq: 정렬 및 중복제거 하는 도구

type

bash에서 명령어 사용 방법이 궁금한 경우 아래와 같이 사용이 가능하다.

```
bash$ help type
```

type명령어를 사용하면 다음과 같이 출력한다.

```
bash $ type cp
```

```
cp is /usr/bin/cp
```

```
bash $ type ls
```

```
ls is aliased to `ls --color=auto`
```

```
bash $ help cp
```

```
-sh: help: no help topics match `cp`. Try `help help' or `man -k cp' or `info cp'.
```

```
bash $ type for
```

```
for is a shell keyword
```

type

function()를 type으로 확인하기.

```
bash $ myfunc() { : ; }
```

```
bash $ type myfunc
```

```
myfunc is a function
```

```
myfunc()
```

```
{
```

```
:
```

```
}
```

```
bash $ alias myalias=:
```

```
bash $ type myalias
```

```
myalias is aliased to `:`
```


type

```
bash $ type -a true  
true is a shell builtin  
true is /bin/true
```

echo

```
bash $ echo hello
```

```
hello
```

```
bash $ echo 'Hello, "$USER"\'!
```

```
Hello, bashuser!
```

기본 명령어

dommie95

잠깐 분명 명령어인데 왜 자꾸 빌트인이라고 하시나요?



gic.kr.com

printf

```
bash $ printf '%s\n' 'Hello!'  
Hello!
```

이 명령어는 C의 printf와 흡사한 포매팅 구조를 가지고 있다.

빌트인?

몇몇 명령어는 시스템에 존재하지만, 실제로는 쉘의 빌트인 명령어.
다음과 같은 명령어로 확인이 가능하다.

type echo

echo is a shell builtin

whereis alias

alias: /usr/bin/alias /usr/share/man/man1/alias.1.gz

file /usr/bin/alias

/usr/bin/alias: a /usr/bin/sh script, ASCII text executable

cat /usr/bin/alias

#!/usr/bin/sh

builtin alias "\$@"

printf

```
bash $ printf '%s\n' -n
```

```
-n
```

```
bash $ string=-n
```

```
bash $ printf '%s\n' "$string"
```

```
-n
```

printf

```
$ printf '%s\n' foo bar baz
```

foo

bar

baz

pwd

현재 작업하는 위치를 확인 시 사용하는 명령어.

\$pwd

/home/bash/working

\$ ls secret/report.txt

\$ ls report_review.txt

\$ ls nonexistent

tildes(~)

경로 관련된 ~(틸드, tild)특수 문자. 이 문자를 통해서 홈 디렉터리로 이동이 가능하다. 틸드 path이외 다른 특수 문자 '.', '..'등 있지만, 보통 홈 디렉터리 이동 시 틸드를 더 선호 한다.

```
$ echo ~
```

```
$ echo ~/secret
```

```
# echo ~
```

```
# echo ~root/.ssh
```

tildes

만약 존재하지 않는 경우 그냥 문자열로 출력이 된다.

```
$ echo ~nouser
```

```
~nouser
```

tildes

특수 문자로 출력하고 싶은 경우 아래처럼 실행한다.

```
$ echo ~bash  
/home/bash
```

```
$ echo \~bash '~bash' "~bash"  
~bash ~bash ~bash
```

cd

cd는 change directory의 약자, 이를 통해서 디렉터리 이동을 한다.
이와 관련된 쉘 변수는 \$PWD 혹은 pwd명령어가 있다.

```
$ pwd
```

```
/home/bash
```

```
$ cd /tmp
```

```
$ pwd
```

```
/tmp
```

cd

앞에서 언급하였던 .. 혹은 .를 통해서 디렉터리 이동이 가능함.

```
$ cd .
```

```
$ cd ..
```

```
$ pwd
```

pushd/popd

cd 명령어와 같이 pushd, popd 명령어로 다음처럼 사용이 가능하다.

```
$ sudo yum install httpd -y
```

```
$ pushd /var/www
```

```
$ pushd /opt/
```

```
$ push -n /usr/local/
```

pushd/popd

```
$ push -n /usr/local/
```

-n 옵션이 붙어 있으면 실제로 디렉터리 이동하지는 않음
내역을 확인하기 위해서는 다음과 같은 명령어로 확인 가능.

```
$ dirs -l -v
```

```
$ pushd +2
```

```
$ pushd -1. == $ cd -
```

pushd/popd

popd 명령어로 이전에 사용하였던 하나씩 이동하면서 스택(stack)에서 하나씩 제거한다.

```
$ pushd /usr/local
```

```
$ pushd /opt/
```

```
$ pushd /var/www
```

```
$ popd
```


set

set명령어는 빌트인 명령어.

set은 현재 사용중인 셸에 설정되어 있는 값을 화면에 변수 이름과 함께 출력한다.

기존에 사용하던 값을 set명령어를 통해서 설정 및 변경도 가능하다.

알파벳 순서대로 정렬해서 화면에 출력한다. 여기에 출력된 변수들은 전부 셸에서 명시된 값이다.

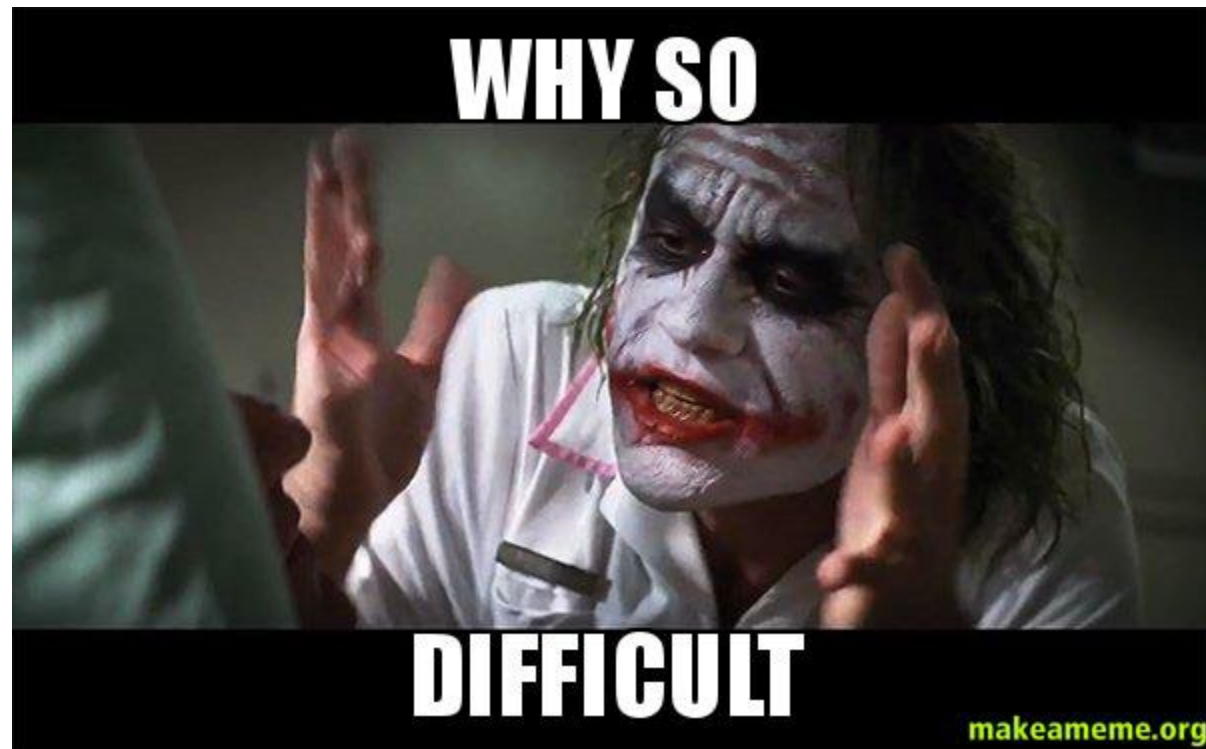
set

옵션	설명	참고 주소
declare	Declare variables and give them attributes. If no names are given, then display the values of variables instead.	https://www.gnu.org/software/bash/manual/bash.html#Bash-Builtins
set	This builtin is so complicated that it deserves its own section. set allows you to change the values of shell options and set the positional parameters, or to display the names and values of shell variables	https://www.gnu.org/software/bash/manual/bash.html#The-Set-Builtin
env	environment variable in Bash	https://www.gnu.org/software/bash/manual/bash.html#Bash-Variables
export	Mark each name to be passed to child processes in the environment	https://www.gnu.org/software/bash/manual/bash.html#Bourne-Shell-Builtins

set

옵션	설명	참고 주소
declare	Declare variables and give them attributes. If no names are given, then display the values of variables instead.	https://www.gnu.org/software/bash/manual/bash.html#Bash-Builtins
set	This builtin is so complicated that it deserves its own section. set allows you to change the values of shell options and set the positional parameters, or to display the names and values of shell variables	https://www.gnu.org/software/bash/manual/bash.html#The-Set-Builtin
env	environment variable in Bash	https://www.gnu.org/software/bash/manual/bash.html#Bash-Variables
export	Mark each name to be passed to child processes in the environment	https://www.gnu.org/software/bash/manual/bash.html#Bourne-Shell-Builtins

set



set

실제로는 set명령어보다 declare명령어 사용을 더 권장한다.

set명령어는 -v옵션을 통해서 설정된 값 확인이 가능하다.

확인 하는 방법은 **printenv, env, compgen** 같은 명령어 사용을 권장한다.

set

-e: 프로그램이 종료가 되면 0 상태 값을 반환한다.

하지만 이 방법을 스크립트에서 사용하는 경우 거의 대다수가
올바르지 않는 값을 반환한다.

그 이유는 -e 옵션은 결과값 예측이 어렵기 때문이다.

그래서 대다수 스크립트는 `set -e`를 사용하지 않고 자체적으로 오류
값 핸들링을 더 선호한다.

set

```
set1.sh
```

```
#!/bin/bash
```

```
set -e
```

```
function hello()
```

```
{
```

```
    echo "I hate the Java language!!"
```

```
}
```

```
hello
```

set

set2.sh

```
#!/bin/bash
```

```
function hello()
```

```
{
```

```
    set -e
```

```
    echo "I hate the Java language!!"
```

```
}
```

```
hello
```


set

set3.sh

#!/bin/fish

set -e

function Error()

{

echo "error occur at line \$1"

}

trap 'error \$LINENO' ERR

Eraaa

set

-n: 명령어를 읽어 오지만 실행하지 않는다.

```
#!/bin/bash
```

```
set -n
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set

set4.sh

-u: 변수가 선언이 되었지만, 값을 가지고 있지 않는 경우 경고 메시지 출력.

```
#!/bin/bash
```

```
set -u
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set

set5.sh

-o: -u하고 비슷하지만 조금 더 자세하게 출력이 된다.

```
#!/bin/bash
```

```
set -o nounset
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set

-f: 글러빙 끄기

```
$ ls *
```

```
$ set -f
```

```
$ ls *
```

set

-a: 변수를 외부에서도 사용할 수 있도록 한다.

```
$ set -a
```

```
$ number1=23
```

```
$ number2=34
```

set

```
$ cat <<EOF> testfile.sh  
#!/bin/bash  
echo $number1 number2  
EOF  
$ bash testfile.sh
```

set

-v: 읽어오는 값을 그대로 화면에 출력한다.

디버깅시 매우 유용하지만, -x와는 다른 정보를 출력한다.

set

-x: 이 옵션은 각각 인자 값이 실행하는 걸 출력한다.
이 옵션을 실행하면 bash스크립트 디버깅에 도움이 된다.
어떤 부분에서 bash프로그램이 입력을 받아서 확장이 되는지
확인 가능하다.

set

디버깅시 다음과 같은 방법으로 진행이 가능하다.

set -x 를 CLI에 같이 선언 혹은 매직-뱅(Magic-Bang)에 다음과 같이 선언한다.

```
#!/bin/bash -x
```

또한 set의 종료 문자는 "--"으로 사용이 가능하며, 이는 위치 값(positional parameter)로 사용이 가능하다.

set

set6.sh

```
#!/bin/bash
```

```
var = 5
```

```
while [ $var -gt 0 ] ; do
```

```
    var=$(( $var - 1 )
```

```
    echo $var
```

```
    sleep 2
```

```
done
```

set

```
set6-1.sh
```

```
#!/bin/bash
```

```
set -x
```

```
var = 5
```

```
while [ $var -gt 0 ] ; do
```

```
    var=$(( $var-1 )
```

```
    echo $var
```

```
    sleep 2
```

```
done
```

set

혹은 다음처럼 실행하여도 된다.

```
bash -x <SCRIPT>
```

declare

declare는 내장 명령어이다. 이를 통해서 변수, 배열 혹은 함수를 구현한다. 변수 구현을 위해서 자주 사용하지는 않지만, 셀의 현재 상태 정보 및 다른 정보들 set를 통해서 훑어보기(overview)가 가능하다.

declare

```
$ declare -p | head -5
```

```
declare -- BASH="/bin/sh"
```

```
$ declare -r
```

```
BASHOPTS="checkwinsize:cmdhist:complete_fullquote:expand_aliases:  
extglob:extquote:force_fignore:histappend:interactive_comments:login  
_shell:progcomp:promptvars:sourcepath"
```

declare

이 명령어를 사용할 때 -f옵션을 -p와 함께 사용하면 declare는 선언이 되어 있는 모든 함수에 대해서 출력한다.

test, [, [[

test 내부 명령어는 두 개의 값을 비교할 때 보통 사용한다.

일반적으로, 다음과 같은 방법으로 많이 사용한다.

&&기호는 test의 if조건문과 비슷하게 동작한다.

```
$ test -e /etc/passwd && echo 'Password file'
```

test, [, [[

다음 명령어로 “[“이다. 이 명령어는 보통 사용시 type명령어와 같이 사용한다.

```
$ type -a [[
```

```
is a shell building [ is /usr/bin/[[
```

test, [, [[

하지만 test, [의 제일 큰 차이점은 test는 단일 명령어로 사용하며, 대괄호 “[“는 쌍으로 사용을 해야 한다.

그래서 이전에 사용하였던 test를 다음처럼 변경이 가능하다.

```
test -e /etc/passwd -> [ -e /etc/passwd ]
```

test, [, [[

하지만, 이 부분도 버전이 변경이 되면서 다음처럼 변경이 되었다.
이유는 간단하게 안전성 및 파싱(parsing)강화를 위해서 변경.

하지만 두 가지 방법 다 사용이 가능하다.

[-> [[

\$ type [[

[[is a shell keyword

test, [, [[

bash 스크립트에서는 test를 많이 사용하지 않는다.
일반적으로 [혹은 [[키워드를 더 많이 사용한다고 생각하자.

편의상 test 및 [비슷한 기능을 하며 유일한 명령어라고 생각하면 편하다.

일반 시스템 명령어

\$ ls

adir bdir cdir

\$ ls -a

. .. .profile .bash_profile Downloads

일반 시스템 명령어(ls)

\$ ls -al

```
drwxrwxr-x. 4 tang tang 4096 May 24 12:19 kubectl-plugins
drwxrwxr-x. 10 tang tang 249 May 24 12:16 kubectx
drwxrwxr-x. 9 tang tang 251 May 24 12:15 kubelogin
drwxrwxr-x. 3 tang tang 80 May 24 12:26 kubergui
```

일반 시스템 명령어(ls)

만약 'ls' 명령어 없이 파일 이름 및 목록을 얻으려면 어떻게 해야 할까?

\$ ls -l

\$ grep pattern -- *

일반 시스템 명령어(ls)

이걸 좀 더 보기 좋게 표현하기 위해서 다음과 같이 시도한다.

```
$ find . -type f -exec grep <PATTERN> -- {} \;
```

```
$ for file in * ; do grep -F <PATTERN> -- "$file" ; done
```

일반 시스템 명령어(mv)

mv 명령어는 디렉터리 및 파일을 이동하거나 혹은 이름을 바꾸는 명령어.

```
$ mv file path/to/directory
```

```
$ mv file1 file2
```

```
$ mv file1 file2 dir4 path/to/directory
```

```
$ mv testdir/ goodir/
```

일반 시스템 명령어(mv)

대화형으로 사용을 원하는 경우 아래와 같은 명령어로 사용이 가능하다.

```
$ mv -i olduser/ newuser/
```

일반 시스템 명령어(mv)

mv1.sh

#!/bin/bash

if [[-e old_user]] ; then

printf 'the old user exists.\n'

mv old_user new_user

fi

mv new_user user

일반 시스템 명령어(cp)

cp 명령어는 파일이나 혹은 파일을 복사 시 사용한다. 이와 비슷한 명령어는 ln명령어가 있다. 하지만, ln명령어는 복사 하지는 않는다.

```
$ cp doc1 doc2 doc3 dir/
```

```
$ ls
```

```
dir
```

```
$ cp dir/ back_dir/
```

일반 시스템 명령어(cp)

```
$ cp -R
```

```
$ cp -r
```

```
$ cp -R back_dir/ rel_dir/
```

-r, -R의 차이점은 기본 옵션은 GNU에서는 -r를 사용한다. 하지만, 전통적인 유닉스에서는 -R옵션을 사용한다.

일반 시스템 명령어(rm/rmdir)

rm, rmdir은 비슷한 동작을 하지만 약간의 차이가 있다.

```
$ rm testfile
```

```
$ rm -rf testdir/
```

```
$ rmdir testdir/
```

```
Directory not empty
```

일반 시스템 명령어(grep)

grep 명령어는 특정 문자열을 표준 출력에서 검색 및 출력한다.

```
$ grep 'ssh' /etc/services
```

ssh	22/tcp	# The Secure Shell (SSH) Protocol
ssh	22/udp	# The Secure Shell (SSH) Protocol
x11-ssh-offset	6010/tcp	# SSH X11 forwarding offset
ssh	22/sctp	# SSH
sshell	614/tcp	# SSLshell
sshell	614/udp	# SSLshell

일반 시스템 명령어(grep)

```
$ grep '^ssh' /etc/services
```

ssh	22/tcp	# The Secure Shell (SSH) Protocol
ssh	22/udp	# The Secure Shell (SSH) Protocol
ssh	22/sctp	# SSH
sshell	614/tcp	# SSLshell
sshell	614/udp	# SSLshell
ssh-mgmt	17235/tcp	# SSH Tectia Manager
ssh-mgmt	17235/udp	# SSH Tectia Manager

일반 시스템 명령어(grep)

```
$ grep -i 'ssh$' /etc/services
```

ssh	22/sctp	# SSH
netconf-ssh	830/tcp	# NETCONF over SSH
netconf-ssh	830/udp	# NETCONF over SSH
sdo-ssh	3897/tcp	# Simple Distributed Objects over SSH
sdo-ssh	3897/udp	# Simple Distributed Objects over SSH
tl1-ssh	6252/tcp	# TL1 over SSH

일반 시스템 명령어(grep)

중복된 출력 결과를 다음과 같은 명령어로 정리가 가능하다.

```
$ uniq
```

```
$ sort
```

혹은 grep에서 -q옵션을 사용하여 조건에 맞는 문자열이 있는지 확인이 가능하다.

```
$ grep -q telnet /etc/service && echo 'Matched'
```

```
$ echo $?
```

```
$ grep -q foobar /etc/service && echo 'Matched'
```

```
$ echo $?
```

일반 시스템 명령어(grep)

혹은 몇 개의 라인이 일치하는지 확인 하려면 다음과 같은 방법이 있다.

```
$ grep -c telnet /etc/service
```

```
5
```

```
$ grep telnet /etc/service | wc -l
```

복합적으로 문자열을 확인이 필요한 경우는 다음과 같은 명령어로 확인이 가능하다.

```
$ grep -e ssh -e telnet /etc/services
```

일반 시스템 명령어(grep)

혹은 -F 옵션을 통해서 문자열 검색이 가능하다.

```
$ grep -F '$' ikea
```

만약 매치가 되지 않는 문자열만 출력하고 싶은 경우 -v 문자열을 사용한다. '^#' 옵션은 주석 문자인 #를 제외하고 화면에 출력.

```
$ grep -v '^#' /etc/services
```

일반 시스템 명령어(grep)

조금 더 복잡하게 특정 문자만 카운팅이 필요한 경우 다음과 같이 실행한다

```
$ grep -cFv -e bash -e ' ' /etc/shells
```

10

일반 시스템 명령어(cut)

cut명령어는 말 그대로 자르는 명령어.

특정 필드나 문자열을 자르고 싶은 경우 cut명령어를 통해서 사용이 가능하다.

자주 사용하는 옵션은 보통 -d, -f이며, -d 는 필드 분리 문자, -f는 필드이다.

일반 시스템 명령어(cut)

```
$ cut -d: -f1 /etc/passwd
```

root

bin

daemon

adm

lp

sync

shutdown

일반 시스템 명령어(cut)

만약, 맨 끝 필드를 출력하고 싶은 경우 아래처럼 실행한다.

```
$ cut -d: -f6- /etc/passwd
```

```
/root:/bin/bash
```

```
/bin:/sbin/nologin
```

```
/sbin:/sbin/nologin
```

일반 시스템 명령어(wc)

단어 혹은 줄 개수 그리고 바이트를 출력한다. 기본 값은 전부를 출력한다.

```
$ wc .bash_history
```

```
200 437 3373 .bash_history
```

위의 출력된 내용은 200줄, 437줄 그리고 총 3373바이트. 여기에서 특정 값만 출력을 원하는 경우 -c, -w, -l같은 옵션을 사용해서 출력이 가능하다. -l옵션은 엄연히 따지고 보면 newline개수만 출력한다.

일반 시스템 명령어(wc)

-c 옵션 경우에는 바이트를 출력한다.

```
$ wc -c .bash_history
```

```
3373 .bash_history
```

일반 시스템 명령어(wc)

-m 총 문자 개수를 카운팅 하기 위해서는 -m 옵션을 사용한다.

```
$ wc -m .bash_history
```

```
3373 .bash_history
```

일반 시스템 명령어(wc)

```
$ wc -m -c Korean
```

```
35 93 test
```

한국어를 UTF-8로 입력하면 영문과 다르게 바이트 및 문자가 카운트가 된다. 그 이유는 인코딩 및 2바이트 문자열로 인하여 개수와 크기가 다르게 출력이 된다.

일반 시스템 명령어(du)

wc명령어로 파일의 크기를 알 수 있다.

```
$ wc -c /bin/bash
```

하지만 전체적인 크기에 대해서는 확인이 불가능 하기 때문에 du명령어를 사용하요 확인한다.

```
$ du -k /bin/bash  
1124  /bin/bash
```

디렉터리 크기를 알고 싶은 경우 아래처럼 실행한다. 모든 결과는 POSIX 1kibibyte(1024)로 출력된다.

```
$ du -k /bin
```

일반 시스템 명령어(find)

find 명령어는 리눅스에서 많이 사용하는 명령어 중 하나이다.
find를 통해서 디렉터리 및 파일 검색이 가능하다. 기본적인 동작 방식은 다음과 같다.

```
$ find ~/
```

출력만 하고 싶은 경우 다음처럼 옵션을 지정한다.

```
$ find /usr/share -print
```

일반 시스템 명령어(find)

여러 위치에서 검색이 필요한 경우 다음처럼 사용이 가능하다.

```
$ find ~/docs ~/googles email.txt
```

출력된 내용을 정렬하기 위해서 다음과 같이 혼용해서 사용이 가능하다.

```
$ find ~/docs | sort
```


일반 시스템 명령어(find)

여러 위치에서 검색이 필요한 경우 다음처럼 사용이 가능하다.

```
$ find ~/docs ~/googles email.txt
```

출력된 내용을 정렬하기 위해서 다음과 같이 혼용해서 사용이 가능하다.

```
$ find ~/docs | sort
```

일반 시스템 명령어(find)

test라는 문자열을 가지고 있는 파일 및 디렉터리

```
$ find ~/docs -name '*test*' -print
```

fish라는 문자열 가지고 있는 파일만 출력

```
$ find ~/docs -name '*fish*' -type f -print
```

수정한지 3일이 지난 파일

```
$ find ~/docs -mtime +3 -print
```

5일전에 수정한 파일

```
$ find ~/docs -mtime -5 -print
```

일반 시스템 명령어(find)

이름에 beef라는 문자열이 없는 것만 검색

```
$ find ~/docs ! -name '*beef*'
```

만약 특정 문자열에 대해서 무시를 하고 싶은 경우, 아래처럼 명령어를 실행한다.

```
$ find . -name '.git*' -prune -o -print
```

일반 시스템 명령어(find)

find를 사용하면, 각각 결과마다 명령어를 같이 실행이 가능하다.

```
$ find vim -type f -name '*.vim' -exec grep -F search -- {} \;
```

-exec: 명령어 실행, 여기서는 'grep -F search --' 명령어

{ }: 이 부분에 find에 찾을 출력 결과물로 대체

\:: 명령어 종료 알림

일반 시스템 명령어(find)

위의 명령어를 구분으로 나누면 다음과 같이 두개 구조로 나누어진다.

```
$ find vim -type f -name '*.vim'
```

```
vim/map.vim
```

```
vim/test.vim
```

```
$ grep -F search -- vim/map.vim
```

```
$ grep -F search -- vim/test.vim
```

일반 시스템 명령어(find)

만약 find의 -exec에 더 많은 인자값이 필요한 경우 +기호를 사용하여 좀 더 많은 인자 값 옵션 사용이 가능하다.

```
$ find vim -type f -name '*.vim' -exec grep -F search -- {} +
```

```
$ grep -F search -- vim/map.vim vim/test.vim
```

일반 시스템 명령어(xargs)

find와 또한 많이 사용하는 명령어는 xargs이다. 이 명령어는 -exec와 비슷하지만 조금 다른 동작 구조를 가지고 있다.

```
$ find vim -type f -name '*.vim' | xargs grep -F search --
```

위의 명령어는 find에서 실행한 결과를 표준 출력으로 오른쪽 xargs명령어에 전달하여 grep 명령어가 작업을 수행한다.

일반 시스템 명령어(xargs)

만약, 출력되는 결과에 null-byte로 종료되는 경우, 올바르게 동작이 안될 수 있기 때문에 find나 xargs를 사용시에 다음처럼 사용을 권장한다.

find -print0 혹은 xargs -0

```
$ find vim -type f -name '*.vim' -print0 | xargs -0 grep -F search --
```


일반 시스템 명령어(sort, uniq)

sort, uniq도 쉘 프로그램 작성 시 많이 사용하는 명령어. sort는 출력된 결과를 알파벳 순서로 정렬 시 사용하는 문자이다.

```
$ sort /etc/shells
```

```
$ sort ~/file1 ~/file2
```

```
$ printf '%s\n' 'line 2' 'line 1' | sort
```

혹은 특정 필드 조합을 통해서 특정 열만 추출이 가능하다.

```
$ sort -t: -k6,6 /etc/passwd
```

일반 시스템 명령어(sort, uniq)

또한 같이 사용하는 옵션은 다음과 같다.(sort)

-n: 숫자가 알파벳 순서보다 먼저 출력이 된다.

-r: 역 순정렬, 맨 마지막 결과가 맨 처음에 출력

일반 시스템 명령어(sort, uniq)

```
$ cat ipaddress.txt
```

```
$ sort ipaddress.txt | uniq -c | sort -k1, 1nr
```

Input, Output, Redirection

```
$ printf "Hello World!\n"
```

```
Hello World!
```

```
$ printf "Hello World!\n" > myfile
```

```
$ cat myfile
```

Input, Output, Redirection

```
$ printf "" > empty.txt
```

```
$ wc -c empty.txt
```

```
0
```

디렉터리 위치로 재전달 혹은 출력하는 경우 다음처럼 한다.

```
$ cd
```

```
$ printf 'Hello World!\n' > ~/tmp/test.txt
```

```
$ printf 'Hello World!\n' > /tmp/test.txt
```

Input, Output, Redirection

```
$ printf 'Hello World!\n' > "$HOME"/text.txt
```

```
$ printf 'Hello, root! \n' > ~root/tmp/text.txt
```

덮어쓰기 금지

bash에서 덮어쓰기를 방지하고 싶은 경우 -c 옵션을 통해서 가능하다.

```
$ print 'First Command' > /tmp/test.txt
```

```
$ print 'Second Command' > /tmp/test.txt
```

```
$ cat test.txt
```

```
Second Command
```

덮어쓰기 금지

```
$ set -C
```

다음처럼 다시 시도한다.

```
$ printf 'Third Command\n' > /tmp/test.txt
```


덮어쓰기 금지

덮어쓰기 금지 상태에서 강제로 덮어쓰기를 하려면 다음처럼 명령어를 실행한다.

```
$ set -C
```

```
$ printf 'Thrid Command\n' >| /tmp/test.txt
```

```
$ echo $?
```

이어 붙이기

기존 텍스트 내용에 추가를 하거나 혹은 이어 붙이기 하는 경우
다음처럼 명령어를 실행한다.

```
$ printf 'First Command\n' > /tmp/test_doc.txt
```

```
$ printf 'Second Command\n' >> /tmp/test_doc.txt
```

```
$ cat test_doc.txt
```

First Command

Second Command

이어 붙이기

```
#!/bin/bash
printf 'Starting script\n' >> log
printf 'Creating test directory\n' >> log
mkdir test || exit
printf 'Changing into test directory\n' >> log
cd test || exit
printf 'Writing current date\n' >> log
date > date || exit
```

이어 붙이기

```
$ mkdir test || exit
```

“||” 만약 해당 디렉터리가 이미 생성이 되어 있는 경우
exit 명령어를 실행한다.

퍼미션

여기에서는 chmod, chown에 대해서는 자세하게 다루지 않는다.
다만, 쉘 스크립트에서 소유권 및 퍼미션을 어떻게 핸들링 하는지
학습 한다.

미리 필요한 선수 지식

- chown
- chmod
- umask

퍼미션

stat, stat 명령어를 통해서 0~7사이의 숫자 8진수를 통해서 퍼미션 확인이 가능하다.

```
$ stat /tmp/test.txt
```

stat 명령어를 통해서 퍼미션 확인을 원하는 경우 아래와 같은 명령어로 확인 가능하다.

```
$ stat -c %a /tmp/test.txt
```

일반 시스템 명령어(cut)

여러 개 필드를 동시에 처리를 하려면 아래와 같이 하면 된다.

```
$ cut -d: -f1,6 /etc/passwd
```

```
root:/root
```

```
bin:/bin
```

```
daemon:/sbin
```

```
adm:/var/adm
```

```
lp:/var/spool/lpd
```

```
sync:/sbin
```

입출력 전환

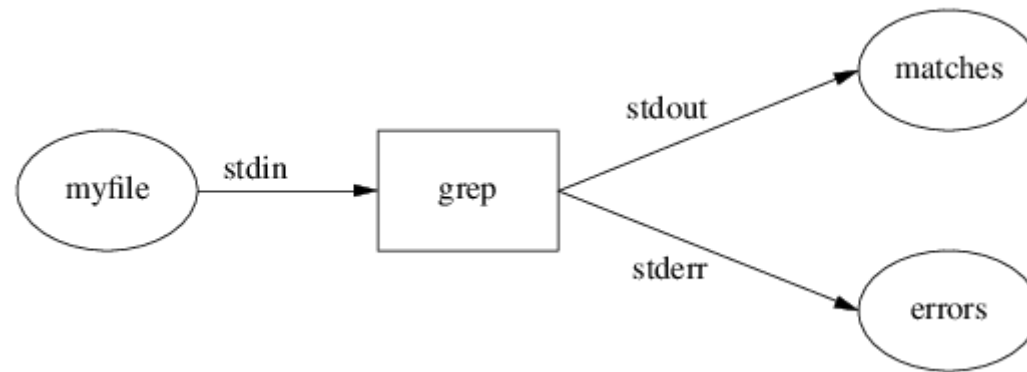
표준 입출력 오류는 각각 숫자를 가지고 있다.

bash에서는 stdout이라고 부르며 이것을 “표준출력”이라고 말한다. 오류는 stderr 그리고 표준 오류라고 부른다.

이 옵션은 매우 bash에서 스크립트에서 매우 일반적으로 사용한다.

다른 호칭으로 파일-디스크립터(File Descriptor)라고 부른다.
디스크립터는 총 3개를 가지고 있다.

입출력 전환



입출력 전환

0

표준 출력 및 오류에 대한 동시 출력, 번호로는 0번 사용.

1

표준출력, 오류가 정상 메시지를 화면에 출력한다.

2

표준오류, 오류 부분에 대해서만 화면에 출력한다.

입출력 전환

오류와 표준 출력을 같이 하는 경우 보통 다음처럼 사용한다.

```
$ grep pattern testfile.txt /thereisnowhere > matches 2> matches
```

아무튼, 위에 처럼 사용하는 경우 현재 오류 메시지만 matches파일에 기록이 남아 있다. 그래서 같이 출력을 하기 위해서는 &(ampersand)를 통해서 1에 대한 출력을 명시 해주어야 한다.

입출력 전환

```
$ grep pattern testfile.txt /thereisnowhere > matches 2>&1
```

위와 같이 실행하면 위의 실행 결과는 표준 출력과 표준 오류를 동시에 파일에 저장한다.

주의 할 부분은 동일한 작업을 수행하기 위해서는 >앞에다가 2>&1을 넣을 수 없다.

리디렉션은 왼쪽에서 오른쪽으로 항상 이루어 진다.

블록 오류 및 처리

가끔 발생한 모든 오류를 화면에 출력할 필요는 없다. 그럴 때 사용하는 유용한 장치가 있다.

`/dev/null`

위의 장치로 오류 내용을 내보내면 된다. 전달하는 방법은 매우 간단하다.

```
$ grep pattern testfile /thereisnowhere > matches 2 > /dev/null
```

블록 오류 및 처리

블록 오류를 처리하기 위해서 comm 명령어를 사용하여 테스트한다.

```
$ comm test1.txt test2.txt
```

```
$ comm test1.txt test2.txt 2> /dev/null
```

```
$ echo $?
```

여러 위치로 재 전달

여러 파일로 내용을 재 전달 하기 위해서는 tee를 사용하여 다음과 같이 처리한다.

```
$ printf 'Copy Copy rom rom\n' | tee my_copy_rom.txt
```

```
Copy Copy rom rom
```

여러 위치로 재 전달

기존 명령어에 다음처럼 파일 이름만 더 추가한다.

```
$ printf 'Copy Copy rom rom\n' | tee myfile1 myfile2 myfile3
```

```
Copy Copy rom rom
```

```
$ ls
```


입력 재 전달

리눅스 명령어 중에 입력 재전달의 대표적인 명령어가 있다.
그건 바로 tr이다. tr명령어는 입력 받은 내용을 대문자 혹은 소문자로 다시 출력한다.

```
$ tr a-z A-Z
```

```
Hello, world!
```

```
HELLO, WORLD!
```

입력 재 전달

위의 명령어를 가지고 재 전달 프로세스를 다루면 다음처럼 다를 수 있다.

```
$ cat mysample | tr a-z A-Z
```

혹은 미리 작성된 파일을 대소문자로 변경하는 경우에는 다음과 같은 명령어로 실행이 가능하다.

```
$ tr a-z A-Z < mysample
```

입력 재 전달

재 전달 오퍼레이터를 손쉽게 다음처럼 사용이 가능하다. 출력을 위한 cat같은 명령어를 사용할 필요가 없다.

```
$ < mysample tr a-z A-Z
```

혹은 아래처럼 복잡하게 입출력을 제어할 수 있다. 아래 명령어는 표준오류만 출력하는 예제이다.

```
$ tr a-z A-Z < mylines > mylines.capitalized 2> mylines.error
```

입력, 출력 그리고 재 전달

bash

Input, Output, Redirection

```
$ printf "Hello World!\n"
```

Hello World!

```
$ printf "Hello World!\n" > myfile
```

```
$ cat myfile
```

Input, Output, Redirection

```
$ printf '' > empty.txt
```

```
$ wc -c empty.txt
```

```
0
```

디렉터리 위치로 재 전달 혹은 출력하는 경우 다음처럼 한다.

```
$ cd
```

```
$ printf 'Hello World!\n' > ~/tmp/test.txt
```

```
$ printf 'Hello World!\n' > /tmp/test.txt
```

Input, Output, Redirection

```
$ printf 'Hello World!\n' > "$HOME"/text.txt
```

```
$ printf 'Hello, root! \n' > ~root/tmp/text.txt
```

덮어쓰기 금지

bash에서 덮어쓰기를 방지하고 싶은 경우 -c 옵션을 통해서 가능하다.

```
$ print 'First Command' > /tmp/test.txt
```

```
$ print 'Second Command' > /tmp/test.txt
```

```
$ cat test.txt
```

```
Second Command
```


덮어쓰기 금지

덮어쓰기 금지 상태에서 강제로 덮어쓰기를 하려면 다음처럼 명령어를 실행한다.

```
$ set -C
```

```
$ printf 'Thrid Command\n' >| /tmp/test.txt
```

```
$ echo $?
```

이어 붙이기

기존 텍스트 내용에 추가를 하거나 혹은 이어 붙이기 하는 경우
다음처럼 명령어를 실행한다.

```
$ printf 'First Command\n' > /tmp/test_doc.txt
```

```
$ printf 'Second Command\n' >> /tmp/test_doc.txt
```

```
$ cat test_doc.txt
```

First Command

Second Command

이어 붙이기

```
#!/bin/bash
printf 'Starting script\n' >> log
printf 'Creating test directory\n' >> log
mkdir test || exit
printf 'Changing into test directory\n' >> log
cd test || exit
printf 'Writing current date\n' >> log
date > date || exit
```

이어 붙이기

```
$ mkdir test || exit
```

"||" 만약 해당 디렉터리가 이미 생성이 되어 있는 경우
exit 명령어를 실행한다.

퍼미션

여기에서는 **chmod**, **chown**에 대해서는 자세하게 다루지 않는다.
다만, 쉘 스크립트에서 소유권 및 퍼미션을 어떻게 핸들링 하는지
학습 한다.

미리 필요한 선수 지식

- chown
- chmod
- umask

퍼미션

stat, stat 명령어를 통해서 0~7사이의 숫자 8진수를 통해서 퍼미션 확인이 가능하다.

```
$ stat /tmp/test.txt
```

stat 명령어를 통해서 퍼미션 확인을 원하는 경우 아래와 같은 명령어로 확인 가능하다.

```
$ stat -c %a /tmp/test.txt
```

일반 시스템 명령어(cut)

여러 개 필드를 동시에 처리를 하려면 아래와 같이 하면 된다.

```
$ cut -d: -f1,6 /etc/passwd
```

```
root:/root
```

```
bin:/bin
```

```
daemon:/sbin
```

```
adm:/var/adm
```

```
lp:/var/spool/lpd
```

```
sync:/sbin
```

입출력 전환

표준 입출력 오류는 각각 숫자를 가지고 있다.

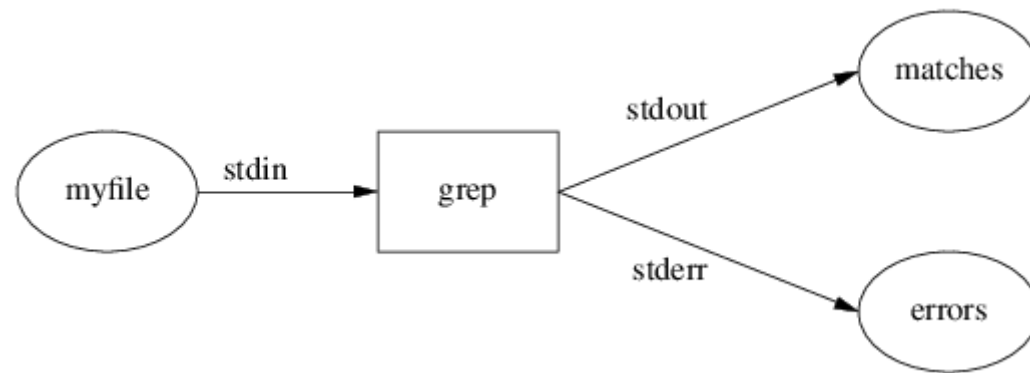
bash에서는 **stdout**이라고 부르며 이것을 "**표준출력**"이라고 말한다. 오류는 **stderr** 그리고 표준 오류라고 부른다.

이 옵션은 매우 bash에서 스크립트에서 매우 일반적으로 사용한다.

다른 호칭으로 **파일-디스크립터(File Descriptor)**라고 부른다.

디스크립터는 **총 3개 타입**을 가지고 있다.

입출력 전환



입출력 전환

0

표준 출력 및 오류에 대한 동시 출력, 번호로는 0번 사용.

1

표준출력, 오류가 정상 메시지를 화면에 출력한다.

2

표준오류, 오류 부분에 대해서만 화면에 출력한다.

입출력 전환

오류와 표준 출력을 같이 하는 경우 보통 다음처럼 사용한다.

```
$ grep pattern testfile.txt /thereisnowhere > matches 2> matches
```

위에 처럼 사용하는 경우 현재 오류 메시지만 matches파일에 기록이 남아 있다. 그래서 같이 출력을 하기 위해서는 "&(ampersand)"를 통해서 1에 대한 출력을 명시 해주어야 한다.

입출력 전환

```
$ grep pattern testfile.txt /thereisnowhere > matches 2>&1
```

위와 같이 실행하면 위의 실행 결과는 표준 출력과 표준 오류를 동시에 파일에 저장한다.

주의 할 부분은 동일한 작업을 수행하기 위해서는 ">"앞에다가 "2>&1"을 넣을 수 없다.

리디렉션은 왼쪽에서 오른쪽으로 항상 이루어 진다.

블록 오류 및 처리

가끔 발생한 모든 오류를 화면에 출력할 필요는 없다. 그럴 때 사용하는 유용한 장치가 있다.

`/dev/null`

위의 장치로 오류 내용을 내보내면 된다. 전달하는 방법은 매우 간단하다.

```
$ grep pattern testfile /thereisnowhere > matches 2 > /dev/null
```

블록 오류 및 처리

블록 오류를 처리하기 위해서 comm 명령어를 사용하여 테스트한다.

```
$ comm test1.txt test2.txt
```

```
$ comm test1.txt test2.txt 2> /dev/null
```

```
$ echo $?
```

여러 위치로 재 전달

여러 파일로 내용을 재 전달 하기 위해서는 tee를 사용하여 다음과 같이 처리한다.

```
$ printf 'Copy Copy rom rom\n' | tee my_copy_rom.txt
```

```
Copy Copy rom rom
```

여러 위치로 재 전달

기존 명령어에 다음처럼 파일 이름만 더 추가한다.

```
$ printf 'Copy Copy rom rom\n' | tee myfile1 myfile2 myfile3
```

```
Copy Copy rom rom
```

```
$ ls
```


입력 재 전달

```
$ export LC_ALL=C
```

```
$ export LANG=C
```

```
$ grep pattern myfile1 myfile2 myfile3
```

```
$ sort -k1,1 myfile1 myfile2 myfile3
```

입력 재 전달

리눅스 명령어 중에 입력 재전달의 대표적인 명령어가 있다.
그건 바로 tr이다. tr명령어는 입력 받은 내용을 대문자 혹은 소문자로 다시 출력한다.

```
$ tr a-z A-Z
```

```
Hello, world!
```

```
HELLO, WORLD!
```

입력 재 전달

위의 명령어를 가지고 재 전달 프로세스를 다루면 다음처럼 다를 수 있다.

```
$ cat mysample | tr a-z A-Z
```

혹은 미리 작성된 파일을 대소문자로 변경하는 경우에는 다음과 같은 명령어로 실행이 가능하다.

```
$ tr a-z A-Z < mysample
```

입력 재 전달

재 전달 오퍼레이터를 손쉽게 다음처럼 사용이 가능하다. 출력을 위한 cat같은 명령어를 사용할 필요가 없다.

```
$ < mysample tr a-z A-Z
```

혹은 아래처럼 복잡하게 입출력을 제어할 수 있다. 아래 명령어는 표준오류만 출력하는 예제이다.

```
$ tr a-z A-Z < mylines > mylines.capitalized 2> mylines.error
```

입력, 출력 그리고 재 전달

bash

긴 문자열 처리

```
#!/bin/bash
case $1 in
    -h | --help )
        printf '%s\n' foo command help'
    -h, --help: show this help
    -q, --quiet: run without diagnostics
    -v, --verbose: add extra diagnostics
        exit 0
    ;;
esac
```

긴 문자열 처리

```
#!/bin/bash
case $1 in
    -h | --help )
        cat << 'EOF'
        -h, --help: show this help
        -q, --quiet: run without diagnostics
        -v, --verbose: add extra diagnostics
        EOF
    exit 0
;;
esac
```

긴 문자열 처리

```
#!/bin/bash
```

```
cat << EOF
```

```
Hello, $USER; you are running this program on $(hostname -s).
```

```
Your home directory is $HOME
```

```
EOF
```


긴 문자열 처리

```
#!/bin/bash
```

```
cat <<- 'EOF'
```

```
    leading tabs
```

```
        will not be included in the output
```

```
EOF
```

파이프(Pipe)

```
$ tr A-Z a-z > words.lowercase
```

```
$ sort words.lowercase > words.sorted
```

```
$ uniq -c words.sorted > words.frequency
```

```
$ sort -k1,1nr words.frequency
```

긴 문자열 처리

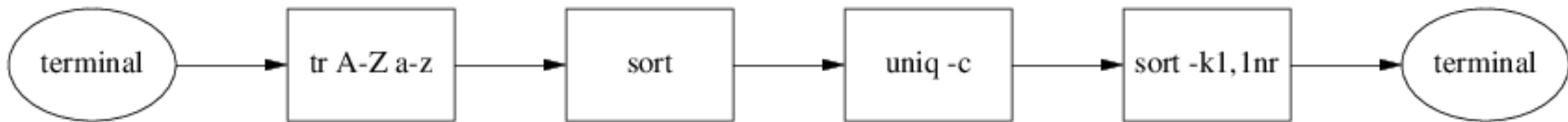
여러가지 명령어를 동시에 사용하기 위해서 쉘 스크립트 형식으로 명령어를 작성한다.

이를 동시에 처리하기 위해서 파이프를 사용한다.
예제로 tr명령어 기반으로 간단하게 구성을 하였다.

긴 문자열 처리

위와 같이 사용하는 경우 쉘 스크립트처럼 여러 명령어를 사용하기 때문에 다음처럼 파이프로 구성하여 동시에 처리 하도록 한다.

```
tr A-Z a-z | sort | uniq -c | sort -k1,1nr
```



긴 문자열 처리

```
$ tr A-Z a-z | sort | uniq -c | sort -k1,1nr
```

이것저것 입력 후 ctrl+d 누르면 아래와 같이 내용이 출력이 된다.

```
sdlkjskljieurwjldklskdj  
akljsnkcjnnjhdsiakjslia  
38473904889orjksdjm,snbns  
1 ($&$  
1 2948  
1 38473904889orjksdjm,snbns
```

긴 문자열 처리

이걸 쉘 스크립트로 변경하면 다음처럼 변경한다.

```
#!/bin/bash
```

```
tr A-Z a-z |
```

```
sort |
```

```
uniq -c |
```

```
sort -k1,1nr
```

Pipe

여러 개의 텍스트 파일을 하나로 통합할 때 다음처럼 스트리밍 처리한다.

```
$ cat testfile1 testfile2 testfile3 > testfiles.combined
```

```
$ date > output
```

```
$ hostname >> output
```

```
$ cat output
```

Pipe

그룹 명령어는 중괄호를 사용하여 처리한다. 아래처럼 실행하면 모든 출력이 대문자로 출력이 된다.

```
$ { date ; hostname ; } | tr a-z A-Z
```


Pipe

```
$ { date ; printf '%s\n' apple carrot banana ; } > testwords
```

```
$ { read date ; printf '%s\n' "$date" ; sort ; } < testwords
```

stream editor

```
$ cat /usr/share/doc/ls/LICENSE
```

```
$ sed '' /usr/share/doc/ls/LICENSE
```

```
$ sed '1,2d' /usr/share/doc/ls/LICENSE
```

```
$ sed '1,/CONFIG_SYSFS_DEPRECATED/d' /usr/share/doc/ls/LICENSE
```

```
$ sed 's/found/not_found/' /usr/share/doc/ls/LICENSE
```

AWK

```
$ cat /etc/passwd
```

```
$ awk '{ print $1 }' /etc/passwd
```

```
$ awk '{ print $2, $3 }' /etc/passwd
```

```
$ awk 'NR > 1 { print $2, $3 }' /etc/passwd
```

```
$ awk 'NR > 1 { printf "$%.2f\n", $2 * $3 }' /etc/costs.txt
```

변수

bash

변수

셸에서 변수는 다음과 같이 선언한다.

```
tangshell='This is mine shell'
```

tangshell를 변수 이름이 되며, 그 뒤에 홑 따옴표에 있는 내용은 셸 변수에서 사용하는 **자료(data)**가 된다.

이걸 **BASH**에서는 "**string content**"라고 한다.

변수

bash는 c 스타일을 사용하기 때문에 변수 이름에 절대 빈 공간이 있으면 안된다. 보통 다음과 같이 많이 선언한다.

```
tangshell = 'fish'
```

```
tangvar='GNU\'\'s Not Unix!'
```

```
tangprompt="$USER$HOST"
```

변수를 호출 할 때 c언어와 마찬가지로 "\$"붙여서 접근이 가능하며, **읽기전용(read only)**상태이다.

변수 목록

변수 목록 및 출력을 하기 위해서는 다음과 같은 방법을 사용해서 호출한다.

```
$ today="2022-06-26"
```

```
$ printf '%s\n' "$today"
```

```
2022-06-26
```

```
$ declare -p today
```

```
declare -- today='2022-06-26'
```

변수 목록

declare에 표시되는 "--"는 특별한 속성이 아니라는 표시이다.

declare -p명령어를 실행하면 다음과 같은 내용을 볼 수 있다.

```
declare -- BASH="/bin/bash"
```

```
declare -- PS1="\u@\h:\w\$"
```


변수 네이밍

변수 네이밍에 대한 규칙은 따로 없지만 일반적으로 다음과 같은 규칙을 가진다.

test_var

TestVar

testvar

Testvar

_testvar

testVar

TestVar_

변수 네이밍

하지만 다음과 같은 이름은 허용하지 않는다.

1testVar

test-var

test.var

test:var

변수 네이밍

또한 변수 설정 시 기존 설정된 이름과 충돌을 주의해야 한다. 매우 자주 있는 케이스는 보통 다음과 같다.

```
#!/bin/bash  
PATH=/home/tang/sbin:/home/tang/bin  
grep -- teststring "$PATH"
```

결과는 다음과 같이 출력이 된다.

```
grep: command not found
```

변수 네이밍

이유는 간단하다. \$PATH는 시스템 변수이며, 기존에 설정된 경로 값에 덮어쓰기가 되어버린 상태다. 그래서 path라는 이름을 사용한다면 다음처럼 선언을 보통 한다.

```
#!/bin/bash
```

```
path=/home/tang/bin:/home/tang/sbin
```

```
grep -- teststring "$path"
```

변수 네이밍

혹은 기존 **\$PATH**를 사용하면서 내용을 추가하는 경우, 다음처럼 처리가 가능하다.

```
PATH="$PATH:/home/tang/bin/:/home/tang/sbin"
```

```
MYHOME=/home/tang
```

```
export MYHOME
```

변수 취소

변수 초기화 하는 방법은 다음과 같다.

```
$ system_var =
```

그리고 초기화 된 내용을 확인 시 다음과 같은 방법으로 가능하다.

```
$ declare -p system  
declare -- system=""
```

변수 취소

POSIX 기준으로 변수에 값이 설정이 되어 있는지 확인 시, 다음과 같이 if문으로 확인한다.

```
#!/bin/bash
if [[ -n $test_system ]] ; then
    print 'the 'test_system' value is: %s\n' "$system"
fi
```

변수 취소

변수를 완전히 제거하려면 unset명령어를 통해서 할 수 있다.

```
$ unset -v test_system
```

```
$ declare -p system
```

```
bash: declare: system: not found
```


환경변수

셸에서 제공하는 환경변수가 있다. 일반적인 변수들은 현재 셸에서만 적용이 된다.

새로운 프로세서에서는 이전 셸에서 추가적용 하였던 변수는 적용되지 않는다.

이를 우리는 "**forked process**"라고 부른다.

이 환경변수를 쉽게 확인하는 방법은 다음과 같은 방법으로 확인이 가능하다.

환경변수

```
$ declare -p BASHPID
```

```
declare -ir BASHPID="327276"
```

```
$ shellusername='Tang'
```

```
$ declare -p shellusername
```

```
declare -- shellusername='Tang'
```

환경변수

```
$ bash
```

```
$ declare -p BASHPID
```

```
declare -ir BASHPID= "331219"
```

```
$ declare -p shellusername
```

```
declare: shellusername: not found
```

환경변수

다시 이전 환경으로 돌아가기 위해서는 **exit** 명령어 실행한다.

```
$ exit
```

```
$ declare -p BASHPID
```

```
$ declare -p shellusername
```

환경변수

만약 다른 환경에도 값을 전달하고 싶은 경우 **POSIX**명령어 **export**로 선언한다.

```
$ USERNAME="choi"
```

```
$ export USERNAME
```

```
$ declare -p USERNAME
```

```
declare -x USERNAME="choi"
```

환경변수

앞에서 이야기 하였지만, 변수 이름 설정 시 전부 대문자로 설정한다.
'**export**'로 설정하는 경우 '**declare**'명령어로 변수를 확인하면 다음처럼 변수 이름이 출력이 된다.

```
$ bash
```

```
$ declare -p USERNAME
```

```
declare -x USERNAME="choi"
```

```
$ exit
```

환경변수

환경변수는 **bash**에서만 사용하는 게 아니다.

모든 프로그램 및 운영체제 환경에서 적용이 되며, 다른 프로그램 언어 **Python, PHP, Awk**같은 명령어에서도 적용이 된다.

예를 들어서 '**sort**', '**date**' 혹은 '**ls**'같은 명령어는 몇몇 시스템 변수에 적용이 받는데 그중 하나가 **LANG, LC_ALL, LC_COLLATE**변수이다.

환경변수

```
$ printf '%s\n' 'helloworld' 'HelloTang' > words
```

```
$ LC_COLLATE=en_US.UTF-8
```

```
$ export LC_COLLATE
```

```
$ sort words
```

```
helloworld
```

```
HelloTang
```


환경변수

\$ LC_COLLATE=C

\$ sort words

HelloTang

helloworld

환경변수

\$ LC_COLLATE=C sort words

HelloTang

helloworld

변수 확장

셸에서 변수를 확장하기 위해서 **\$1**를 사용한다. 일반적으로 **\$0**은 자신이 사용하는 명령어, **\$1**부터는 확장 파라미터라고 한다.

```
$ yourname='tang'
```

```
$ printf '%s\n' "Hello, $yourname."
```

```
Hello, Tang.
```

변수 확장

변수 이름을 좀 더 명확하게 보기 위해서 중괄호로 표시를 할 수 있다.

```
$ yourname='tang'
```

```
$ printf '%s\n' "__Hello, ${yourname}__."
```

```
__Hello, Tang__.
```

변수 확장

역시 변수에 값이 선언이 되어 있지 않는 경우에는 다음처럼 출력이 된다.

```
$ printf '%s\n' "Hello, $notest."
```

Hello, .

변수확장

만약에 선언이 되지 않는 변수에 대해서 오류 메시지를 출력하고 싶으면 다음처럼 선언한다.

```
$ set -u
```

```
$ printf '%s\n' "Hello, $notest."
```

```
bash: notest: unbound variable
```

변수확장

변수를 외부에서 받아 오기 혹은 읽어오기 할 때 내장함수인 **read**를 사용해서 사용한다.

이때 같이 사용하는 옵션은 **"-r"**를 같이 사용해서 변수에 할당한다.

```
$ read -r yourname
```

```
$ declare -p yourname
```

```
declare -- yourname='tang'
```

명령어 결과를 변수로

종종 사용하는 방법으로, 실행된 명령어의 결과를 변수에 저장한다. 이유는 여러가지가 있지만 일반적으로 재가공 혹은 출력 용도로 사용한다. 다음과 같은 예제를 본다.

```
$ whom=$(whoami)
```

```
$ printf '%s\n' "$whom"
```

```
tang
```


명령어 결과를 변수로

두개 이상의 명령어를 동시에 사용해야 되는 다음처럼 명령어를 구성한다.

```
$ yourpasswd="$(getent passwd $(whoami))"
```

기본값 설정

기본값 설정을 종종 필요하다.

예를 들어서 시스템에서 올바르게 기본값이 설정이 안되어 있으면 올바르게 동작이 안될 수 있다.

그래서 쉘에서는 `"${default_var:-default}"` 같은 형식을 제공한다.

```
$ cd -- "${MYDATA_DIR:-/usr/local/data}"
```

혹은 다음과 같이 재선언이 가능하다.

```
$ cd -- "${MYDATA_DIR:=/usr/local/data}"
```

문자열

bash

자르기

```
$ ta='type:json'
```

```
$ print 'Type: %s\n' ${ta#type:}'
```

```
Type: json
```

```
$ ta='type:json'
```

```
$ printf 'Field name: %s\n' "${ta%:*}'
```

```
Field name: type
```

자르기

```
$ text='*WARNING'
```

```
$ printf '%s\n' "${text#\*}"
```

```
$ printf '%s\n' "${text#'*}'
```

```
$ path=/usr/local/test/bin/test
```

```
$ printf 'Removed the path as look like a file : %s\n' "${path##*/}"
```

```
Removed the path as look like a file: test
```

문자열 자르기

문자열 처리 시, cut같은 명령어 사용이 가능하다.
조금 더 편하게 **`${var:<position>:<length>}`**사용이 가능하다.

```
$ title==' Password =='
```

```
$ printf '%s\n' "${title:3}"
```

```
Password ==
```

```
$ printf '%s\n' "${title:3:8}"
```

```
Password
```

```
$ alpha='abcdefghijklmn'
```

```
$ printf '%s\n' "${alpha: -3:2}"
```

```
ij
```

길이 측정

문자열 길이는 '**wc**'같은 명령어로 확인이 가능하다. 쉘 빌트인 사용하는 경우 좀 더 쉽게 가능하다.

```
$ alpha='abcdefghijklmn'
```

```
$ printf '%u\n' "${#alpha}"
```

```
11
```

문자열 바꾸기

```
$ notice='I\'\'I do it today.'
```

```
$ printf '%s\n' "${notice /today/tomorrow}"
```

```
$ notice='Yes, today. I\'\'I do it today.'
```

```
$ printf '%s\n' "${notice /today/tomorrow}"
```

Yes, tomorrow, I'll do it today.

```
$ printf '%s\n' "${notice //today/tomorrow}"
```

Yes, tomorrow, I'll do it tomorrow.

문자열 바꾸기

```
$ notice='Yes, today. I\'\'ll do it today.'
```

```
$ printf '%s\n' "${notice/today*/I\'\'ll do it soon.}"
```

```
Yes, I'll do it soon.
```

대소문자 변경

```
$ text='helloWorld'  
$ printf '%s\n' "${text^^}"  
HELLOWORLD
```

```
$ printf '%s\n' "${text^}"  
HelloWorld
```

```
$ printf '%s\n' "${text,}"  
helloWorld
```

외부 변수 합치기

```
#!/bin/bash
```

```
temp="${test:-foo}"
```

```
printf '%s\n' "${temp#.ext}"
```

글로벌 혹은 글로벌

bash

글로브(*) 사용하기

글로브, **별표** 혹은 **눈**이라고 말하는 문자이다.

셸에서는 글로브 즉 별표를 사용하면 기본적으로 **모든 것(all of match)**라고 표현한다.

매치가 되는 자원은 거의 모든 자원에 적용이 된다.

```
$ ls -a
```

```
.  ..  Documents  Pictures  Downloads
```

```
$ printf '%s\n' *
```

```
hello
```

```
world
```

```
Documents
```

```
Downloads
```

글로브(*) 사용하기

```
$ printf '%s\n' ???uments
```

Documents

```
$ printf '%s\n' *[ts]
```

test1.txt

test2.txt

글로브(*) 사용하기

```
$ printf '%s\n' ???uments
```

Documents

```
$ printf '%s\n' *[*ts]
```

jsues.txt

sjwue.txt

글로브(*) 사용하기

```
$ cat -- c*
```

```
cat: 'c*' : No such file or directory
```

```
$ shopt -s nullglob
```

```
$ printf '%s\n' c*
```

'shopt' 명령어는 bash의 빌트인 명령어이며, 내부 함수를 설정한다.
위의 명령어는 'nullglob'를 활성화 한다.

글로브(*) 사용하기

```
$ shopt -s nocaseglob
```

```
$ printf '%s\n' A*
```

```
apple
```

```
Aux
```

글로빙 확장하기

bash는 부족한 글로빙 기능을 확장하기 위해서 정규식(regular express)기반으로 고급기능(혹은 더 복잡하게) 사용이 가능하다.

?(pattern): 하나라도 패턴에 맞으면 출력

+(pattern): 최소 한 개로도 맞으면 출력

***(pattern):** 숫자 하나라도 맞으면 출력

@(pattern): 명시한 조건만 맞으면 출력

!(pattern): 조건에 맞지 않는 것 만 출력

글로빙 활용하기

예를 들어서 자주 사용하는 `"/var/log"`를 예를 들면 다음처럼 사용한다.

`/var/log`에서 `.gz`로 끝나지 않는것만 출력

```
$ ls -l /var/log/!(*.log)
```

`haw`혹은 `mess`단어가 맞으면 화면에 출력

```
$ ls -l /var/log/@(haw|mess)*
```

배열

bash

배열 사용하기

셸에서 배열을 사용하기 위해서는 **경계기호(delimiter)**를 사용해서 변수와 배열을 구별한다.

```
$ devices=('iphone', 'android', 'winphone')
```

위의 배열은 매우 기본적인 배열 선언이다.

배열 사용하기

```
$ printf '%s\n' "${devices[0]}"
```

iphone

```
$ printf '%s\n' "${divces[2]}"
```

winphone

배열 사용하기

```
$ printf '%s\n' "${devices[-1]}"
```

winphone

```
$ printf '%s\n' "${devices[@]}"
```

iphone

android

winphone

배열 사용하기

```
$ printf '%s\n' "${devices[@]^}"
```

IPHONE

ANDROID

WINPHONE

배열 사용하기

```
$ homefiles=("$HOME"/*)
$ printf '%s\n' "${homefiles[@]}"
/home/tang/1.txt
/home/tang/2.txt
/home/tang/3.txt
/home/tang/4.txt
```

배열 사용하기

bash4.0 이후에 새로 할당된 배열 할당이 있다.

파이썬의 사전(dictionary)처럼 기능을 제공한다.

```
$ declare -A colors
```

```
$ colors=([tomato]="red" [orange]="yellow" [carrot]="orange")
```

```
$ printf '%s\n' "${colors[orange]}"
```

```
yellow
```

네트워크 관리 명령어

IP utility and nftables

네트워크 관리 명령어

네트워크 관리 시 다음과 같은 명령어를 많이 사용한다.

ip(ifconfig)

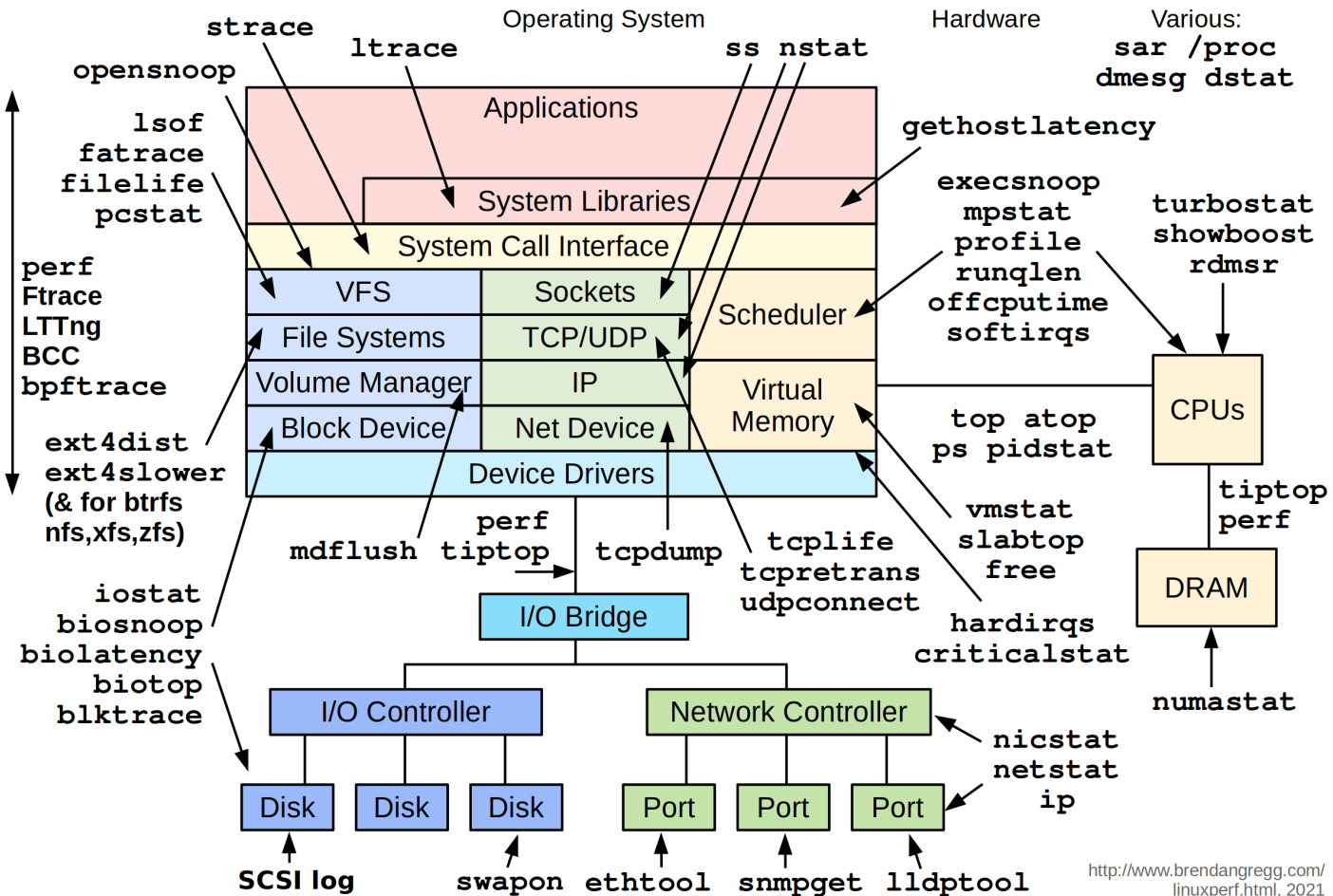
ss(netstat)

ip r(route)

nftables(iptables)

네트워크 관리 명령어

Linux Performance Observability Tools



네트워크 관리 명령어

레드햇 계열은 RHEL 7 혹은 CentOS 7으로 넘어가면서 기존에 사용하던 ifconfig, route 및 netstat 명령어를 더 이상 지원하지 않는다.

ipv6 및 namespace장치 지원을 위해서 명령어를 ip, ss로 교체하였다.

네트워크 관리 명령어

ip 명령어는 이전에 사용하던 ifconfig, route 명령어를 대체하는 명령어 이다.

이 명령어를 통해서 네트워크 카드에 설정된 아이피 정보 확인이 가능하고 수동으로 추가가 가능하다.

사용 방법은 일반적으로 다음과 같다.

네트워크 관리 명령어

ip address show

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 56:6f:08:c8:03:ca brd ff:ff:ff:ff:ff:ff
    inet 192.168.90.226/24 brd 192.168.90.255 scope global noprefixroute dynamic eth0
        valid_lft 86118sec preferred_lft 86118sec
    inet6 fe80::1b8e:665b:de5b:3457/64 scope link tentative noprefixroute dadfailed
        valid_lft forever preferred_lft forever
```


네트워크 관리 명령어

ip route

default via 192.168.90.250 dev eth0 proto dhcp metric 100

192.168.90.0/24 dev eth0 proto kernel scope link src 192.168.90.226
metric 100

네트워크 관리 명령어

ip monitor

192.168.90.91 dev eth0 lladdr 56:6f:08:c8:00:3d REACHABLE

192.168.90.91 dev eth0 lladdr 56:6f:08:c8:00:3d STALE

192.168.90.91 dev eth0 lladdr 56:6f:08:c8:00:3d REACHABLE

네트워크 관리 명령어

ss -antp

State	Recv-Q	Send-Q	Local Address:Port
LISTEN	0	128	*:22
.		users:(("sshd",pid=1210,fd=3))	
LISTEN	0	100	127.0.0.1:25
.		users:(("master",pid=1459,fd=13))	
ESTAB	0	0	192.168.90.226:22
192.168.90.91:39520		users:(("sshd",pid=1490,fd=3))	
LISTEN	0	128	:::22
:::*		users:(("sshd",pid=1210,fd=4))	
LISTEN	0	100	:::1:25
:::*			

네트워크 관리 명령어

```
# ip netns add test
```

```
# ip netns list
```

```
test
```

```
# ls -al /var/run/netns/
```

```
total 0
```

```
drwxr-xr-x.  2 root root  60 Sep 11 13:46 .
```

```
drwxr-xr-x. 24 root root 760 Sep 11 13:46 ..
```

```
-r--r--r--.  1 root root   0 Sep 11 13:46 test
```

네트워크 관리 명령어

```
# ip a add 192.168.1.200/255.255.255.0 dev eth1
```

```
# ip a s eth1
```

```
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc  
pfifo_fast state UP group default qlen 1000
```

```
    link/ether 56:6f:08:c8:03:cb brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.1.200/24 scope global eth1
```

```
        valid_lft forever preferred_lft forever
```

네트워크 관리 명령어

더미 장치 생성하는 방법

```
# nmcli connection add type dummy ifname dummy0 ipv4.method manual  
ipv4.addresses 192.0.2.1/24 ipv6.method manual ipv6.addresses  
2001:db8:2::1/64
```

```
# nmcli con sh
```

```
# ip link add dummy1 type dummy
```

```
# ip a add 192.168.1.200/255.255.255.0 dev dummy1
```

```
# ip link list
```

네트워크 관리 명령어

ip addr add brd 172.20.10.255 dev dummy1

ip addr add broadcast 172.20.10.255 dev dummy1

ip addr add broadcast 172.20.10.255 dev dummy1

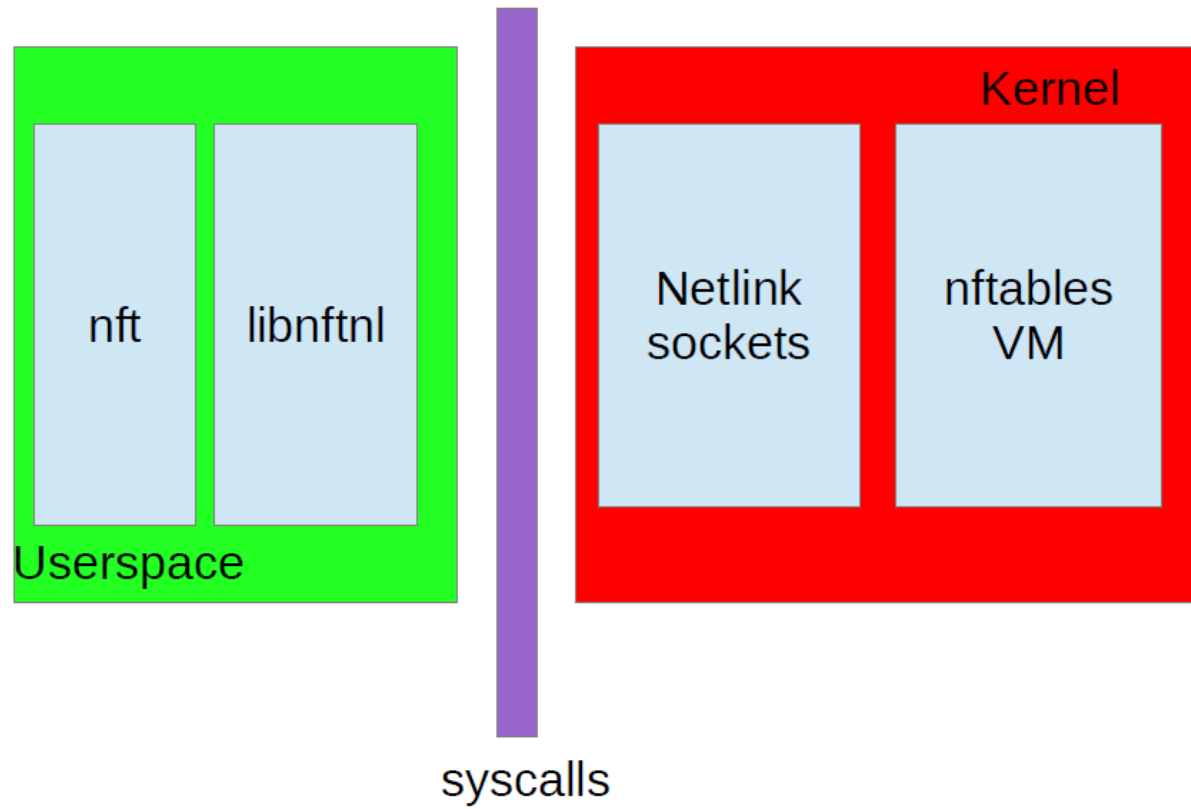
네트워크 관리 명령어

```
# ip netns add foo
# ip netns add bar
# ip netns set foo 12
# ip netns set bar 13
# ip -n foo netns set foo 22
# ip -n foo netns set bar 23
# ip -n bar netns set foo 32
# ip -n bar netns set bar 33
# ip netns list-id target-nsid 12
# ip netns list-id target-nsid 12 nsid 13
```

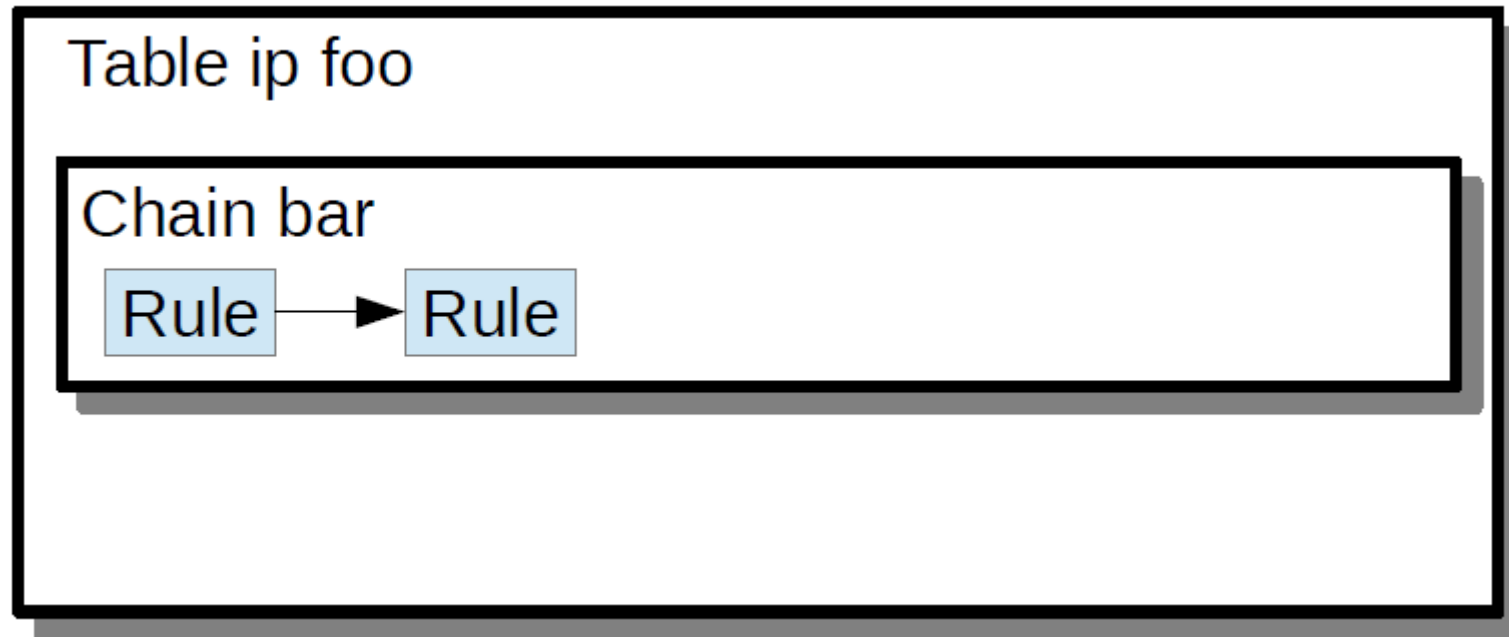

방화벽 및 네트워크 설정

nftables, firewalld, NetworkManager

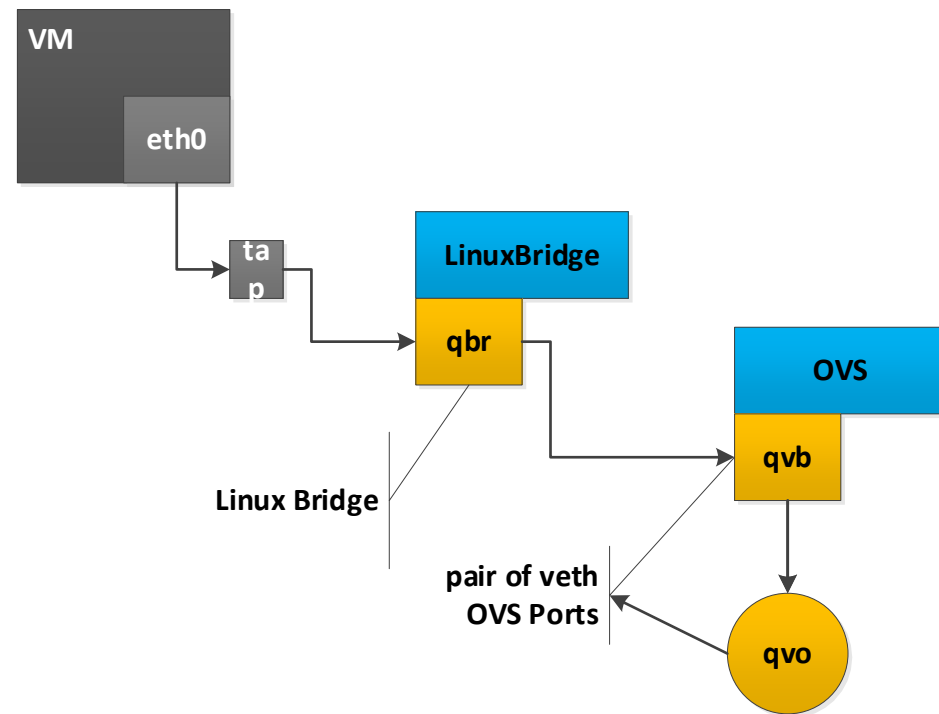
nftables



nftables



nftables



nftables

목록 출력

```
# nft list table filter
```

```
# nft list counters
```

특정 아이피 드롭

```
# nft add rule ip filter OUTPUT ip daddr 1.2.3.4 drop
```

nftables

패키지

```
# yum install nftables
```

```
# systemctl enable --now nftables.service
```

여전히 호환성 모드로 iptables 사용은 가능하나 가급적으면 nftables 기반으로 작업 권장. 혹은 firewalld로 작업.

nftables

특정 아이피 드롭, 하지만 카운팅 모듈 사용

```
# nft add rule ip filter OUTPUT ip daddr 1.2.3.4 counter drop
```

특정 아이피로 나가는 아이피 대역에 대한 카운팅

```
# nft add rule ip filter OUTPUT ip daddr 192.168.1.0/24 counter
```

특정 포트번호에 대한 패킷 드랍

```
# nft add rule ip filter INPUT tcp dport 80 drop
```

nftables

counter 모듈 설명

<https://lwn.net/Articles/631372/>

daddr: 도착 아이피

dport: 도착 포트

output/input: 입출력 패킷

nftables

```
# nft add rule filter INPUT icmp type-echo-request accept
```

```
# nft add rule ip filter OUTPUT ip protocol icmp ip daddr 1.2.3.4 counter drop
```

```
# nft -a list table filter
```

```
# nft delete rule filter OUTPUT handle <NUMBER>
```

```
# nft flush table filter
```

```
# nft insert rule filter INPUT tcp dport 80 counter accept
```

nftables

handle <NUMBER>

정책의 번호

nft add rule filter/ip <TYPE>

이전 iptables의 chain 이름. 레드햇 경우에는 대문자로 작성한다.

방화벽 및 네트워크

레드햇 계열은 네트워크 및 방화벽 설정은 사용자가 좀 더 사용하기 편하기 위해서 다음과 같은 도구를 사용한다.

NetworkManager

레드햇 계열 및 다른 리눅스 배포판에서 많이 도입해서 사용하고 있다. 최대 장점은 리눅스에서 사용하는 거의 모든 네트워크 타입 장치 구성이 가능하다.

firewalld

방화벽 및 네트워크

firewalld

기존에 사용하는 iptables, nftables위에 고급 계층을 올려서 사용자가 쉽게 사용할 수 있도록 한다.

firewall-cmd명령어 기반으로 **영역(zone)**, **서비스(service)**, **포트(port)**와 같은 elements를 xml기반으로 쉽게 구성이 가능하다.

firewalld

```
# systemctl start firewalld
```

```
# ls -l /lib/firewalld
```

```
# cd /lib/firewalld/services
```

```
...
```

```
http.xml
```

```
...
```

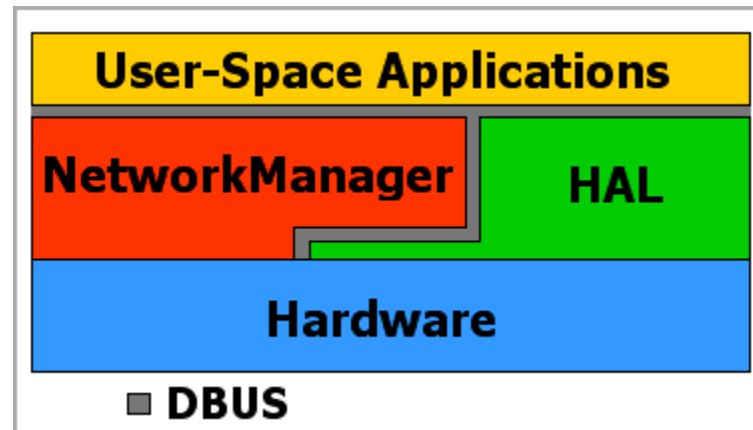
NetworkManager

systemctl start NetworkManager

nmcli

nmtui

nm-connection-editor



NetworkManager

/var/run/NetworkManager

devices initrd no-stub-resolv.conf resolv.conf system-connections

/var/lib/NetworkManager/

internal-590765f7-ea80-49b3-aca2-8a570d85952b-ens3.lease

NetworkManager-intern.conf secret_key seen-bssids timestamps

NetworkManager

/run/NetworkManager/

devices initrd no-stub-resolv.conf resolv.conf system-connections

/etc/NetworkManager/

conf.d dispatcher.d dnsmasq.d dnsmasq-shared.d

NetworkManager.conf system-connections

NetworkManager

/run/NetworkManager/

devices initrd no-stub-resolv.conf resolv.conf system-connections

/etc/NetworkManager/

conf.d dispatcher.d dnsmasq.d dnsmasq-shared.d

NetworkManager.conf system-connections

패키지 관리 방법

dnf, yum, rpm and modules

dnf

dnf는 기존에 사용하던 yum명령어를 대체하는 새로운 패키지 관리자. dnf에서 제일 큰 차이점은 바로 modules라는 기능이 새로 도입이 되었음.

이를 통해서 특정 프로그램 설치 시 의존성이 필요한 경우 dnf module를 통해서 패키지 제공이 되며, 또한 이 기능은 이전에 사용하였던 SCL(Software Collection Library)와 비슷한 기능을 제공하기도 함.

module

```
# dnf module list
```

CentOS Stream 8 - AppStream

Name	Stream	Profiles Summary
------	--------	------------------

container-tools	rhel8 [d][e]	common [Most recent (rolling) versions of podman, buildah, skopeo, runc, common, runc, common, CRIU, Udica, etc as well as dependencies such as container-selinux built and tested together, and updat
-----------------	--------------	---

d] ed as frequently as every 12 weeks.

container-tools	1.0	common [Stable versions of podman 1.0, buildah 1.5, skopeo 0.1, runc, common, CRIU, Udica, etc as well as dependencies such as container-selinux built and tested together, and supported for 24 month
-----------------	-----	---

d] s.

container-tools	2.0	common [Stable versions of podman 1.6, buildah 1.11, skopeo 0.1, runc, common, etc as well as dependencies such as container-selinux built and tested together, and supported as documented on the App
-----------------	-----	---

d] lication Stream lifecycle page.

container-tools	3.0	common [Stable versions of podman 3.0, buildah 1.19, skopeo 1.2, runc, common, etc as well as dependencies such as container-selinux built and tested together, and supported as documented on the App
-----------------	-----	---

d] lication Stream lifecycle page.

container-tools	4.0	common Stable versions of podman 4.0, buildah 1.24, skopeo 1.6, runc, common, CRIU, Udica, etc as well as dependencies such as container-selinux built and tested together, and supported as document
-----------------	-----	---

ed on the Application Stream lifecycle page.

rollback

yum 및 dnf는 rollback기능을 제공한다.

dnf history **rollback**

undo

redo

list

info

yum vs dnf-3

yum과 dnf의 제일 큰 차이점은 yum은 순수하게 파이썬으로 작성이 되어 있다.

하지만, dnf는 libdnf3라는 c언어로 래핑 되어 있으며, 이를 통해서 기존에 yum에서 불편했던 느린 반응 및 파이썬 라이브러리 문제 발생을 방지 할 수 있다.

yum vs dnf-3

rpm명령어는 기존에는 rpm이라는 하나의 명령어만 사용하였다. 하지만, 리눅스가 계속 발전이 되면서 rpm은 기능을 나누면서 몇몇 명령어가 기존 rpm에서 분리하여 추가 되었다.

- rpm
- rpm2archive
- rpm2cpio
- rpmdb
- rpmkeys
- rpmquery
- rpmverify

프로세스 관리를 위한 명령어

process utils

ps

- 1 UNIX options, which may be grouped and must be preceded by a dash.
- 2 BSD options, which may be grouped and must not be used with a dash.
- 3 GNU long options, which are preceded by two dashes.

ps

ps 명령어는 제일 기본적으로 사용하는 프로세스 확인 명령어.

제일 많이 사용하는 명령어는 다음과 같다.

ps -ef

위의 명령어는 AIX(Unix)에서 사용하던 유닉스 스타일 명령어

ps aux

위의 명령어는 GNU 스타일의 명령어

일반적으로 엔지니어들이 많이 사용하는 **스타일은 "dash(-)" 스타일**이다.

ps

ps 명령어는 어떤 옵션을 사용하느냐 따라서 시스템에 부하 수준이 다르다.

구동 중인 시스템에서는 'top' 명령어보다 'ps' 명령어 사용을 더 권장하고 있다.

ps aux or ps au

위의 명령어는 BSD스타일 명령어

ps

ps aux or ps au

a옵션은 BSD 스타일의 "only yourself" 옵션

u옵션은 EUID(effective user ID)

x옵션은 x BSD스타일의 "must have a tty" 옵션

```
[root@localhost ~]# ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.7 128144  6828 ?        Ss   Sep11   0:02 /usr/lib/systemd/systeme
root         2  0.0  0.0      0     0 ?        S    Sep11   0:00 [kthreadd]
root         4  0.0  0.0      0     0 ?        S<   Sep11   0:00 [kworker/0:0H]
root         5  0.0  0.0      0     0 ?        S    Sep11   0:00 [kworker/u32:0]
root         6  0.0  0.0      0     0 ?        S    Sep11   0:00 [ksoftirqd/0]
root         7  0.0  0.0      0     0 ?        S    Sep11   0:00 [migration/0]
root         8  0.0  0.0      0     0 ?        S    Sep11   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        R    Sep11   0:36 [rcu_sched]
root        10  0.0  0.0      0     0 ?        S<   Sep11   0:00 [lru-add-drain]
root        11  0.0  0.0      0     0 ?        S    Sep11   0:00 [watchdog/0]
```

ps

ps -ef

-f 옵션은 "full-formatting list"

-e 옵션은 "all processes"

위의 명령어는 Unix(AIX)스타일의 명령어

```
[root@localhost ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1         0  0 Sep11 ?        00:00:02 /usr/lib/systemd/systemd --switched-root --
root           2         0  0 Sep11 ?        00:00:00 [kthreadd]
root           4         2  0 Sep11 ?        00:00:00 [kworker/0:0H]
root           5         2  0 Sep11 ?        00:00:00 [kworker/u32:0]
root           6         2  0 Sep11 ?        00:00:00 [ksoftirqd/0]
root           7         2  0 Sep11 ?        00:00:00 [migration/0]
root           8         2  0 Sep11 ?        00:00:00 [rcu_bh]
root           9         2  0 Sep11 ?        00:00:36 [rcu_sched]
root          10         2  0 Sep11 ?        00:00:00 [lru-add-drain]
```

ps

ps -x

```
[tang@www ~]$ ps -x
```

PID	TTY	STAT	TIME	COMMAND
69492	?	Ss	0:00	/usr/lib/systemd/systemd --user
69496	?	S	0:00	(sd-pam)
79289	?	Ss	0:05	tmux
79290	pts/15	Ss	0:00	-sh
79320	pts/21	Ss	0:00	-sh
79350	pts/22	Ss	0:00	-sh
80825	pts/15	S+	0:00	ssh root@192.168.90.171
80828	pts/15	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
80861	pts/21	S+	0:00	ssh root@192.168.90.3
80864	pts/21	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3
80865	pts/22	S+	0:00	ssh root@192.168.90.168
80868	pts/22	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.168

ps

ps -fu or -fU <UID> <UNAME>

ps -fU tang

```
[tang@www ~]$ ps -fU tang
UID      PID      PPID  C  STIME TTY          TIME CMD
tang     69492      1    0 Sep06 ?        00:00:00 /usr/lib/systemd/systemd --user
tang     69496    69492    0 Sep06 ?        00:00:00 (sd-pam)
tang     79289      1    0 Sep06 ?        00:00:05 tmux
tang     79290    79289    0 Sep06 pts/15    00:00:00 -sh
tang     79320    79289    0 Sep06 pts/21    00:00:00 -sh
tang     79350    79289    0 Sep06 pts/22    00:00:00 -sh
tang     80825    79290    0 Sep07 pts/15    00:00:00 ssh root@192.168.90.171
tang     80828    80825    0 Sep07 pts/15    00:00:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
tang     80861    79320    0 Sep07 pts/21    00:00:00 ssh root@192.168.90.3
tang     80864    80861    0 Sep07 pts/21    00:00:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3
```

ps

ps -U root -u root

-u 영향 받는 사용자 아이디(RUID)

-U 실제 사용자 아이디(EUID)

```
[tang@www ~]$ ps -u tang -U tang
  PID TTY          TIME CMD
 69492 ?            00:00:00 systemd
 69496 ?            00:00:00 (sd-pam)
 79289 ?            00:00:05 tmux: server
 79290 pts/15      00:00:00 sh
 79320 pts/21      00:00:00 sh
 79350 pts/22      00:00:00 sh
 80825 pts/15      00:00:00 ssh
 80828 pts/15      00:00:00 sss_ssh_knownho
```


ps

ps -Fg tang or ps -fG 1000

```
[tang@www ~]$ ps -Fg tang
```

UID	PID	PPID	C	SZ	RSS	PSR	STIME	TTY
tang	69492	1	0	22388	9528	4	Sep06	?
tang	69496	69492	0	81902	3320	0	Sep06	?
tang	79289	1	0	7246	4528	5	Sep06	?
tang	79290	79289	0	6929	5140	3	Sep06	pts/15
tang	79320	79289	0	6929	5012	5	Sep06	pts/21
tang	79350	79289	0	6929	4988	3	Sep06	pts/22
tang	80825	79290	0	15448	6812	6	Sep07	pts/15
tang	80828	80825	0	24797	5588	6	Sep07	pts/15
tang	80861	79320	0	15448	6920	3	Sep07	pts/21

ps

ps -fp

```
[tang@www ~]$ sudo ps -fp 1
UID          PID    PPID  C STIME TTY          TIME CMD
root           1         0  0 Sep03 ?        00:01:30 /usr/li
```

ps

ps -f --ppid 1

```
[tang@www ~]$ sudo ps -f --ppid 1
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	751	1	0	Sep03	?	00:00:11	/usr/lib/systemd/systemd-journald
root	799	1	0	Sep03	?	00:00:06	/usr/lib/systemd/systemd-udevd
rpc	899	1	0	Sep03	?	00:00:03	/usr/bin/rpcbind -w -f
root	908	1	0	Sep03	?	00:00:04	/sbin/auditd

pstee

데모 및 설명

Isof

데모 및 설명

kill/killall/pkill

데모 및 설명

1, HUP, reload a process

9, KILL, kill a process without page down

15, TERM, gracefully stop a process

pskill/pgrep

pskill은 프로세스의 이름 혹은 다른 속성을 확인하여 시그널 전달

```
$ pskill <signal> <process_name>
```

```
$ pgrep -u root sshd
```

```
$ pgrep -u tang,root,adaemon
```

pgrep/pkill

```
$ pgrep -c ssh
```

```
10
```

```
$ pgrep -d "-" ssh
```

```
1363-30317-79439-79442-79459-79462-80023-80026-80047-80050-  
80051-80054-80825
```


pskill/pgrep

```
$ pgrep -u tang ssh
```

80825

80828

80861

80864

80865

80868

131706

pgrep/pkill

```
$ pgrep -u tang ssh -c  
12
```

```
$ pgrep -l ssh  
1363 sssd_ssh  
30317 sshd  
79439 ssh  
79442 sss_ssh_knownho  
79459 ssh  
79462 sss_ssh_knownho
```

pskill/pgrep

```
$ pgrep -a ssh
```

```
79439 ssh root@192.168.90.178
```

```
79442 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.178
```

```
79459 ssh root@192.168.90.187
```

```
79462 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.187
```

```
80023 ssh root@192.168.90.183
```

```
80026 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.183
```

```
131706 sshd: tang@pts/0
```

pskill/pgrep

\$ pgrep ssh -v ssh

\$ pgrep -x sshd

\$ pgrep -n ssh

최근에 생성된 ssh process만 출력

\$ pgrep -o ssh

이전에 생성된 ssh process만 출력

pskill/pgrep

```
$ pskill '^ssh$'
```

```
$ pskill -9 -f "ping 8.8.8.8"
```

```
$ pskill -u mark
```

```
$ pskill -u mark gnome -9
```

```
$ pskill -9 -n screen
```

사용자 관리를 위한 명령어

useradd, usermod, chmod, chown

coreutils

사용자 관리를 위한 명령어

`useradd/passwd`

이미 알고 있는 명령어, 그리고 많이 사용하는 명령어. 리눅스에는 두 가지 형식의 명령어가 존재한다.

`useradd`

GNU스타일 사용자 추가 명령어. 레드햇 및 대다수 리눅스 배포판은 `useradd` 명령어 기반으로 사용자 추가한다.

`adduser`

사용자 관리를 위한 명령어

adduser

이 명령어는 데비안 계열 및 BSD에서 많이 사용하는 명령어. 레드햇 계열 배포판은 useradd, adduser 둘 다 가지고 있지만 실제로는 useradd를 사용한다.

사용자 관리를 위한 명령어

이 부분에서는 일반적으로 사용하는 사용자 및 암호 설정 과정을 다른 방식으로 다루어 본다.

1. useradd with openssl

```
# useradd -p $(openssl passwd -1 <PASSWORD>) <USERNAME>
```

```
# useradd test
```

```
# echo "<USERNAME>:<PASSWORD>" | chpasswd
```

```
# echo <PASSWORD> | passwd --stdin <USERNAME>
```

사용자 관리를 위한 명령어

usermod

데모 및 설명

사용자 관리를 위한 명령어

chmod

데모 및 설명

사용자 관리를 위한 명령어

chown

데모 및 설명

DAY 2

디스크 관리를 위한 명령어

block device and disk recognized
management and naming of devices

디스크 관리를 위한 명령어

block device and disk recognized
management and naming of devices

디스크 파티셔닝

gdisk/fdisk/cfdisk/parted

parted, partprobe

디스크 파티셔닝

gdisk/fdisk/cfdisk/parted

parted, partprobe

Straits/VDO/LVM

Advanced of Disk management

Straits/VDO/LVM

Advanced of Disk management

다중 스왑 및 파일 스왑

SWAP within Partition and File

다중 스왑 및 파일 스왑

SWAP within Partition and File

네트워크 연결 스토리지

NFSv4

GlusterFS

네트워크 연결 스토리지

NFSv4

GlusterFS

kpatch

live kernel patch

가상머신 및 컨테이너 구성 및 구현

libvirt

podman

가상머신 및 컨테이너 구성 및 구현

DAY 3

가상서버 및 컨테이너 관리

libvirt

glusterfs

간단한 앤서블 문법

introduce of ansible

앤서블 기초

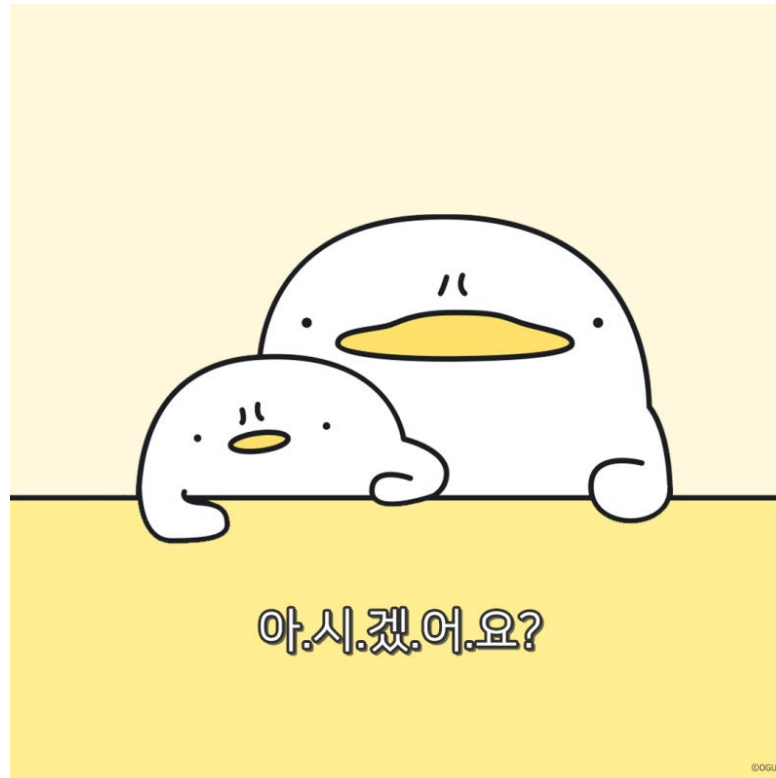
교육목적

이번 앤서블 교육은 다음 대상으로 제작 및 구성.

1. 앤서블 처음 사용하는 사용자
2. 실무까지는 아니어도 어떠한 방식으로 동작하는지 궁금하신 사용자
3. 간단하게 YAML 형태로 동작 학습
4. 전체적인 용어 및 기능 학습

앤서블 기초

교육목적



앤서블

앤서블 소개

앤서블은 본래 Ansible Community, Company에서 제작하였고 릴리즈 도중에 레드햇이 인수하였음.

앤서블은 총 두 가지 릴리즈를 유지하고 있음.

Ansible Core

Ansible Engine

앤서블

앤서블 코어

코어는 앤서블 앤서블 핵심 모듈로 구성되어 있으며, 그 이외 확장으로 posix, collection, community로 통해서 확장이 가능하다.

기본적으로 많이 사용하는 모듈은 core module/posix 그 이외 나머지 기능들은 collection, community로 확장.

조금 혼돈 스럽기는 하지만 앤서블 코어를 **엔진(engine)**이라고 표현하기도 한다. core, engine 둘 다 같은 기능이다.

앤서블

앤서블 코어

또한 앤서블 코어는 두 가지 릴리즈 방식이 있다.

ansible-core(another name is ansible-base)

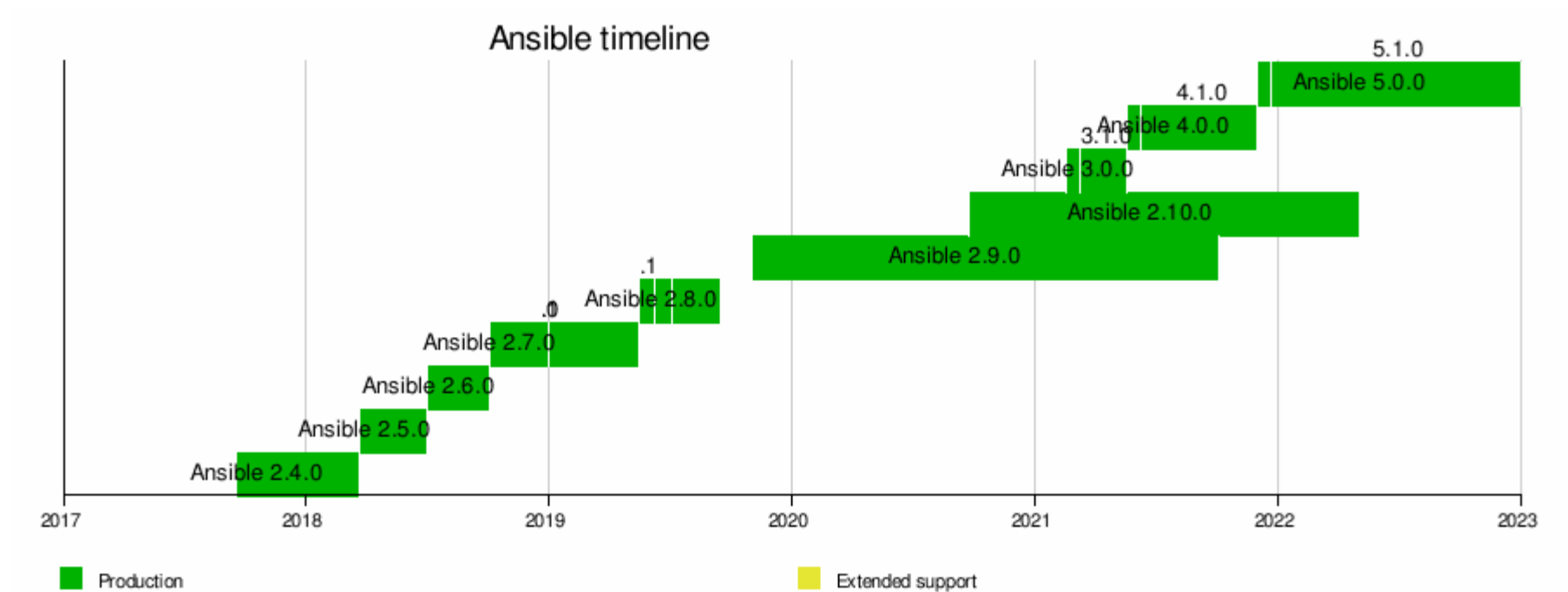
앤서블 코어는 앤서블 인터프리터 + 코어 모듈

ansible-project

앤서블 코어 + 추가적인 컬렉션 구성

앤서블

타임라인

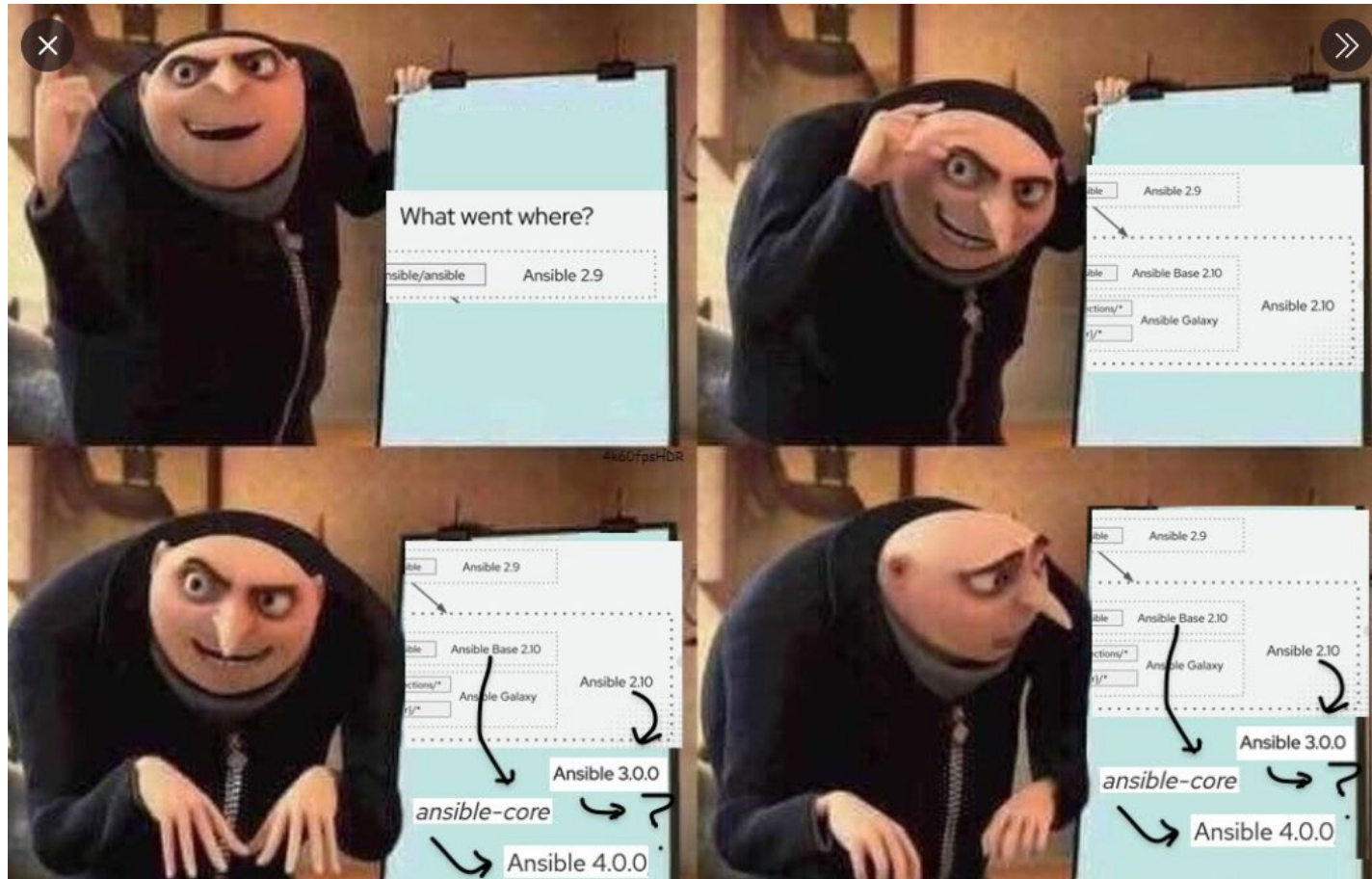


앤서블

릴리즈



앤서블



Andre Sencioles
@asenci

Is it just me struggling to keep up with Ansible changes? 😓

Red Hat Ansible @ansible · Feb 18

Ansible 3.0.0 has been released! This version of the #Ansible #community package marks the end of the restructuring of the Ansible ecosystem. To learn more about how and why we got to 3.0.0, take a look at "Announcing the Community Ansible 3.0.0 Package" [ansible.com/blog/announcing...](https://ansible.com/blog/announcing-the-community-ansible-3.0.0-package) pic.twitter.com/jdUiHlaKJR

[Show this thread](#)

5:48 PM · Feb 18, 2021 · Twitter Web App

9 Likes



앤서블

- ▶ The `ansible==4.0.0` package on PyPI will depend on `ansible-core>=2.11`
- ▶ `ansible==3.0.0` that was released today depends on `ansible-base>=2.10.5,<2.11`.
- ▶ `ansible-core` doesn't become 4.0.0, the next version will be 2.12.

앤서블

앤서블 소개

앤서블 타워

앤서블 타워는 두 가지 버전으로 구성이 되어있다.

첫 번째는 **앤서블 AWX 업 스트리밍 버전**이 있으며 레드햇에서 제고하는 서브스크립션 버전. 제일 큰 차이점은 버전 픽스가 되지 않는 롤링 업데이트 버전이다.

타워의 큰 특징은 웹 대시보드에서 API기반으로 **앤서블 호출, 구성 및 관리가 가능**하며 **사용자 단위로 작업 할당 및 추적**이 가능하다.

앤서블

앤서블 소개

앤서블 사용하기 전에 준비를 해야 될 부분은 다음과 같다.

YAML 작성 시 사용할 에디터(아무거나 좋다! 정말로!)

YAML 문법

ansible.cfg 및 directory work frame

ansible, ansible-playbook 명령어 사용 방법

앤서블

editor

리눅스 콘솔에서 작성 시 사용하는 대표적인 에디터는 **vi/vim** 에디터가 있다.

거의 대다수 리눅스는 기본적으로 vi는 설치가 되어 있으나, vim설치가 되어 있지 않는 경우가 있다.

이러한 이유로 vim설치 이외 몇 가지 기능을 추가적으로 구성 및 활성화 해야 한다.

앤서블

editor

다른 편집기를 원하는 경우, 다른 에디터를 사용해서 교육을 진행 하셔도 됩니다. 😊

앤서블

YAML

YAML은 HTML처럼 사양만 지정이 되어 있고, 여기에서 필요한 방법을 가져와서 자신이 사용하는 언어 기반으로 포매팅/파싱/프로세싱 하면 된다.

앤서블

<https://yaml.org/>

```
%YAML 1.2
---
YAML: YAML Ain't Markup Language™

What It Is:
  YAML is a human-friendly data serialization
  language for all programming languages.
|
YAML Resources:
  YAML Specifications:
    - YAML 1.2:
      - Revision 1.2.2      # Oct 1, 2021 *New*
      - Revision 1.2.1      # Oct 1, 2009
      - Revision 1.2.0      # Jul 21, 2009
    - YAML 1.1
    - YAML 1.0

  YAML Matrix Chat:  '#chat:yaml.io'      # Our New Group Chat Room!
  YAML IRC Channel:  'libera.chat#yaml'    # The old chat
  YAML News:         'twitter.com/yamlnews'
  YAML Mailing List: 'yaml-core'          # Obsolete, but historical

  YAML on GitHub:    # github.com/yaml/
    YAML Specs:       yaml-spec/
    YAML 1.2 Grammar: yaml-grammar/
    YAML Test Suite:  yaml-test-suite/
    YAML Issues:      issues/
```

앤서블 준비

SSH

앤서블은 기본적으로 두 가지 접근 방식을 제공한다.

1. SSH 비공개/공개키 접근 방법
2. 사용자 아이디 및 비밀번호 접근

이 둘 중 하나를 사용하면 된다. 일반적으로 첫번째 방식을 더 많이 사용한다.

앤서블 준비

SSH

```
$ ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
```

```
$ ssh-copy-id <ID>@<HOST>
```

앤서블 준비

SSH

- hosts: all

tasks:

- authorized_key:

user: "{{ ansible_user }}"

key: "{{ lookup('file', '/home/' + lookup('env', 'USER') +
'{{ k8s_public_rsa_locate }}') }}"

앤서블 준비

vim editor

```
$ touch .vimrc
```

```
$ sudo yum install vim -y
```

혹은 centos 8버전 이후를 사용한다면

```
$ sudo yum install vim-ansible
```


앤서블 준비

vim

```
$ vi ~/.vimrc
```

```
" Load local vimrc from project directory (set this in your main vimrc)
```

```
" set exrc
```

```
autocmd FileType yml,yaml setlocal ts=2 sts=2 sw=2 expandtab et ai cuc  
colorscheme torte
```

```
" Color column 80
```

```
highlight ColorColumn ctermbg=6 guibg=lightblue
```

```
set colorcolumn=80
```

앤서블 준비

.vimrc

" General

set cuc # Set cursorcolumn

set exrc " Load vimrc from project directory

set number " Show line numbers

set numberwidth=4

set showmatch " Highlight matching brace

set visualbell " Use visual bell (no beeping)

앤서블 준비

.vimrc

```
set hlsearch " Highlight all search results
set smartcase " Enable smart-case search
set ignorecase " Always case-insensitive
set incsearch " Searches for strings incrementally
```

```
set autoindent " Auto-indent new lines
set expandtab " Use spaces instead of tabs
set shiftwidth=2 " Number of auto-indent spaces
set smartindent " Enable smart-indent
set smarttab " Enable smart-tabs
set softtabstop=2 " Number of spaces per Tab
```

앤서블 준비

.vimrc

```
set autoindent " Auto-indent new lines
set expandtab  " Use spaces instead of tabs
set shiftwidth=2    " Number of auto-indent spaces
set smartindent " Enable smart-indent
set smarttab  " Enable smart-tabs
set softtabstop=2    " Number of spaces per Tab
```

앤서블 준비

.vimrc

```
set autoindent " Auto-indent new lines
set expandtab  " Use spaces instead of tabs
set shiftwidth=2    " Number of auto-indent spaces
set smartindent " Enable smart-indent
set smarttab  " Enable smart-tabs
set softtabstop=2    " Number of spaces per Tab
```

앤서블 준비

.vimrc

"" Advanced

set ruler " Show row and column ruler information

set undolevels=1000 " Number of undo levels

set backspace=indent,eol,start " Backspace behaviour

set nowrap

앤서블 준비

.vimrc

짧은 옵션은 다음과 같다.

```
filetype plugin indent on  
autocmd FileType yaml  
set sw=2 et ts=2 indent=yaml st=2
```

앤서블 준비

nano

```
$ touch ~/.nanorc
```

```
$ sudo yum install nano
```

```
$ nano ~/.nanorc
```


앤서블 준비

nano

```
# Supports `YAML` files  
syntax "YAML" "\.ya?ml$"  
header "^(---|===)" "%YAML"
```

Keys

```
color magenta "^\\s*[$A-Za-z0-9_-]+\\:"  
color brightmagenta "^\\s*@[$A-Za-z0-9_-]+\\:"
```

Values

```
color white "\\s.+\\$"
```

앤서블 준비

nano

Booleans

icolor brightcyan " (y|yes|n|no|true|false|on|off)\$"

Numbers

color brightred " [[[:digit:]]+(\.[:digit:]]+)?"

Arrays

color red "\" \"\]" ":\s+[|>]" "^\\s*- "

Reserved

color green "(^|)!!(binary|bool|float|int|map|null|omap|seq|set|str) "

앤서블 준비

nano

Comments

color brightwhite "#.*\$"

Errors

color ,red ":\w.+\$"

color ,red ":'\'+\$"

color ,red ":'\'+\$"

color ,red "\s+\$"

앤서블 준비

nano

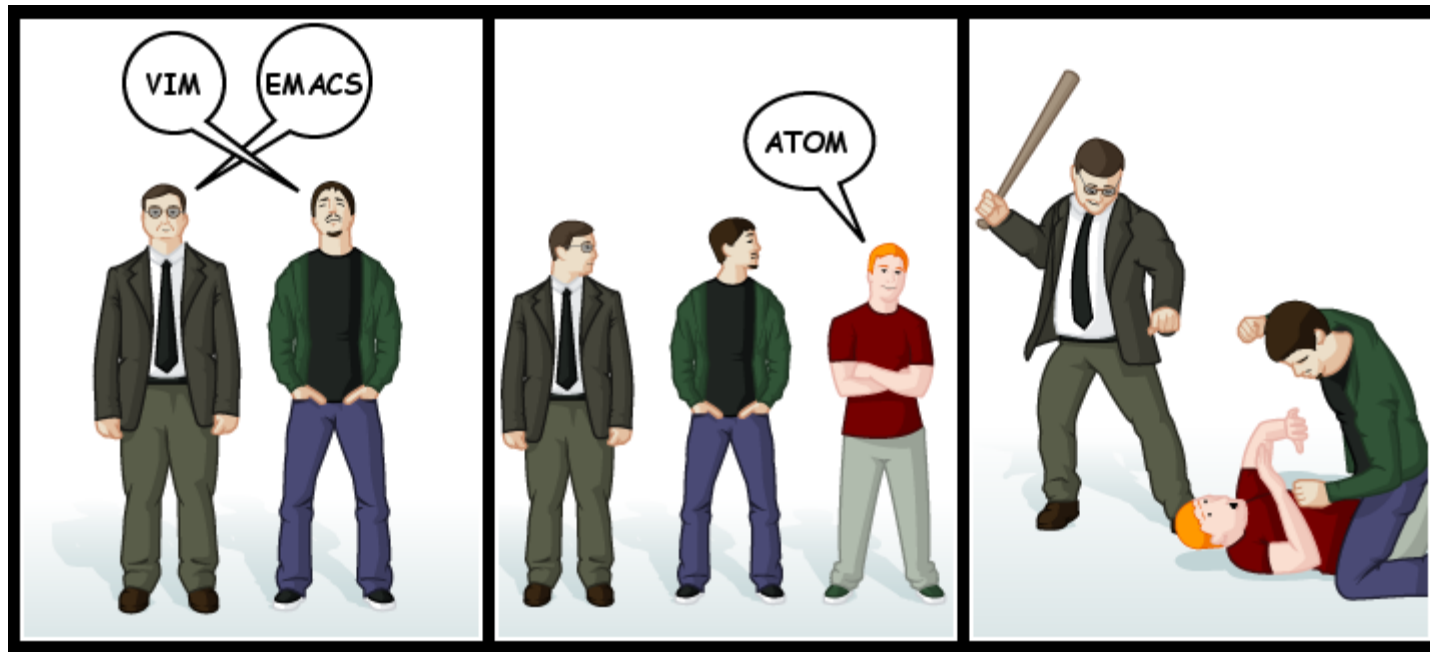
```
## Non closed quote  
color ,red "[\\"]^[\\"]]*$"
```

```
## Closed quotes  
color yellow "[\\"].*[\\"]"
```

```
## Equal sign  
color brightgreen ":(|$)"  
set tabsize 2  
set tabto spacs
```

앤서블 준비

Editor War



앤서블 준비

Editor War



앤서블 문법

YAML



앤서블 문법

YAML

앤서블에서 사용하는 문법을 작성하기 위해서는 다음과 같은 조건을 만족해야 한다.

최소 한 칸 이상의 띄어쓰기(권장은 2칸)

탭 사용시 반드시 빈 공간으로 전환

블록 구별은 -(대시)로 반드시 처리

기본 명령어 키워드

YAML

-name: simple playbook

hosts: all

become: true

tasks:

module:

args1:

args2:

기본 명령어 키워드

YAML

앤서블 블록 구별을 보통 "-"로 구별한다. 시작 블록은 보통 다음과 같은 형식으로 많이 사용한다.

-name:

<keyword>

<module>

-name:

<keyword>

<module>

기본 명령어 키워드

YAML

그래서 YAML상단에는 다음과 같은 형태로 키워드 명령어를 사용한다.

```
-name: <작업이 수행 시 출력되는 이름>  
  hosts: all          ## 대상서버 이름.  
  become: true        ## 앤서블 내장 키워드
```

기본 명령어 키워드

YAML

hosts키워드는 다음과 같은 미리 예약된 옵션이 있다.

localhost: 127.0.0.1와 같은 자기 자신 루프 백(loopback)

all : 인벤토리에(inventory)등록된 모든 호스트

[group]: 특정 그룹에만 적용하는 명령어 키워드

inventory, group 이런 부분은 너무 깊게 들어가지 말기!

앤서블 문법

YAML

인벤토리 구조는
sample-inventory 파일 참고

기본 명령어 키워드

전역 키워드 선언

예제

become:

remote_user:

매 상단에 있는 키워드 및 옵션은 보통 전역 키워드(global keyword)라고 생각하면 된다.

여기에서 적용된 옵션 및 명령어는 모든 플레이북에 적용이 되며 기존에 적용이 되어 있는 `ansible.cfg`의 내용을 오버라이드(override)가 된다.

기본 명령어 키워드

tasks:

모든 작업이 시작되는 구간.

tasks 구간에는 여러 **모듈(module)**이 모여서 하나의 작업 워크플로우(workflow)를 구성한다.

여러 개의 워크 플로우가 구성이 되면 이것을 플레이북 혹은 **플레이북 작업(playbook tasking)**이라고 부른다.

앤서블 문법

- name

그래서 권장하는 방법은 작성시 각각 모듈에 **name:**키워드를 사용하여 작성 및 구성을 권장한다.

- name: this is the first module task
ping:

위와 같은 방법으로 명시한 모듈에 어떻게 사용할 것인지 명시한다.

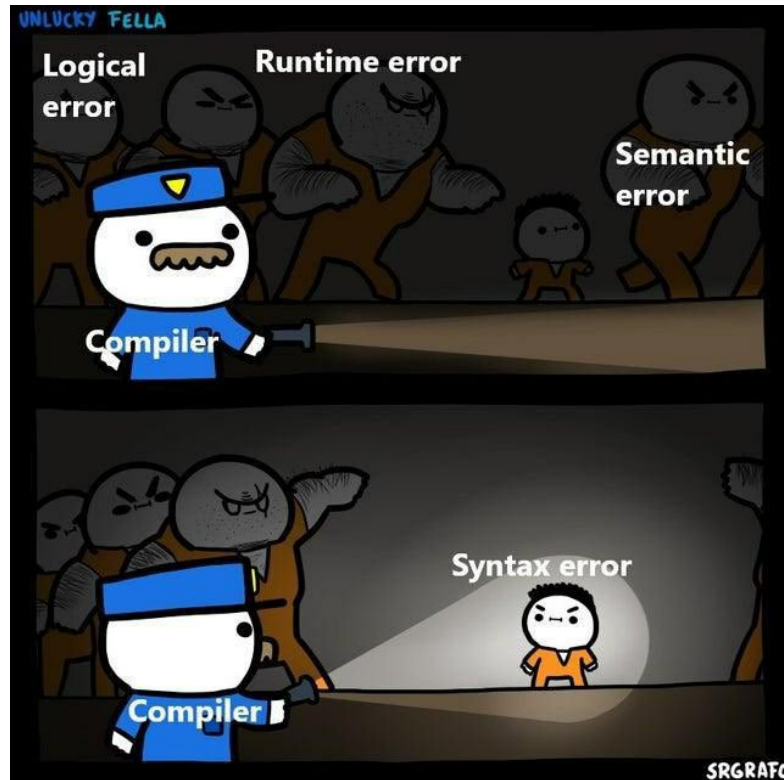
앤서블 문법

YAML

sample-yaml.yaml 파일 참고

앤서블 문법

YAML



좀 더 고급 자동화를 위한 앤서블 구성

intermedia of ansible

앤서블 ad-hoc 구성 및 linux-system-role 기반으로 설정 구현

ansible system role and roles

앤서블 모듈 및 패키지

MODULE

앤서블 모듈은 확장이 가능하며 확장 기능을 제공하는 프로그램.
현재 앤서블 모듈은 두 가지로 나누어서 기능 제공 하고 있음

ansible.core.*

ansible.posix.*

앤서블에서 제공하는 핵심 기능. 일반적인 핵심 기능은 posix모듈에서 제공한다. 여기에 copy, file, fetch와 같은 자주 사용하는 기능이 포함되어 있다.

앤서블 모듈 및 패키지

CORE MODULE

앤서블에서 제공하는 핵심 기능.

일반적인 핵심 기능은 CORE모듈에서 제공한다.

여기에 copy, file, fetch와 같은 자주 사용하는 기능이 포함되어 있다.

앤서블 모듈 및 패키지

ansible.posix

앤서블에서 제공하는 핵심 기능.

많이 사용하는 리눅스 배포판에서 사용이 가능함. 이를 통해서 배포판 상관없이 공통적인 작업 수행이 가능하다.

앤서블 모듈 및 패키지

ansible.collection

앤서블 확장 기능. 코어 기능에서 확장된 기능이며, 이걸 보통 **컬렉션(collection)**이라고 부른다. 컬렉션에는 벤더사에서 제공하는 모듈 기능도 포함이 되어 있다.

예를 들어서 대표적으로 많이 사용하는 **Network Manager, Linux System Role**도 컬렉션에 포함 되어 있다.

컬렉션은 ansible-galaxy통해서 설치가 가능하다.

앤서블 모듈 및 패키지

ansible.community

커뮤니티는 검증이 되지 않는 사용자 혹은 특정 그룹에서 만들어서 배포하는 **Role** 혹은 **Module**이다.

커뮤니티에 보면 보통 자신들이 만들어서 사용하는 플레이북 혹은 사용자화 된 Roles를 업로드 많이 한다.

커뮤니티도 컬렉션처럼 갤럭시를 통해서 설치가 가능하다.

앤서블 모듈 및 패키지

Modules

`/lib/python3.8/site-packages/ansible_collections`

앤서블 모듈 및 패키지

MODULE CASE

제일 많이 사용하는 모듈 **copy**기반으로 기능을 구현하면 다음과 같다.

- name: copy an issue file to remote server

copy:

src: /tmp/issue

dest: /etc/issue

앤서블 모듈 및 패키지

MODULE DOCUMENT

모듈에 대한 자세한 옵션을 보기 위해서는 다음과 같은 명령어로 실행한다.

```
$ ansible-doc <MODULE NAME>
```

사용 가능한 모듈 목록을 확인하기 위해서는 아래 명령어로 목록 확인이 가능하다.

```
$ ansible-doc -l
```

ad-hoc

앤서블 문법

ad-hoc

앤서블은 YAML형태 말고 ad-hoc방식이 있다.

이 방식은 마치 쉘 스크립트 실행하는 방식과 비슷하게 **모듈+인자** **형태**로 구성이 되어 있다.

아래는 간단한 ad-hoc사용 방식이다.

```
$ ansible <host>, -m <module> -a "arg1=<value> arg2=<value>:"
```

앤서블 문법

ad-hoc

애드훅은 쉘 스크립트에서 같이 사용하거나 혹은 몇몇 쉘 스크립트 기능을 표준화 모듈 기반으로 사용하기 위해서 사용한다.

자주 사용하지는 않지만 애드훅 기반으로 구성하는 경우 아래와 같은 방식으로 구성을 한다.

앤서블 문법

ad-hoc

```
$ vi adhoc_replace_shell_functions.sh
```

```
#!/bin/bash
```

```
if $(ansible localhost, -m ping)
```

```
then
```

```
    ansible all, -m copy -a "dest=/var/www/html/index.html, content='This is World'"
```

```
else
```

```
    ansible all, -m copy -a "dest=/root/README.md, content='This file is wrong'"
```

```
fi
```


앤서블 문법

ADHOC 연습문제

ad-hoc기반으로 다음과 같은 작업을 처리한다.
파일명은 `adhoc-exam.sh`으로 정한다.

1. "Hello SKT"라는 `default.html`파일을 `/var/www/html/`에 생성.
2. 웹 서버 패키지가 설치가 안되어 있으면 `yum`모듈을 사용해서 설치
3. 방화벽에 `http`, `https`서비스가 등록이 안되어 있으면 `http`, `https service`를 등록. 모듈은 `firewalld`를 사용.
4. 문제가 없으면 `uri`모듈을 통해서 웹 페이지 접근 및 접속이 잘 되는지 확인
5. 동작이 안되는 부분이 있으면 올바르게 동작하도록 수정

앤서블 문법

ADHOC 연습문제

materials/ adhoc-exam.sh

앤서블 인벤토리

앤서블 인벤토리

INVENTORY

앤서블 인벤토리는 다음과 같은 형식을 가지고 있다.

[인벤토리 이름]

<호스트>

[인벤토리 이름:children]

<그룹이름>

앤서블 인벤토리

INVENTORY

인벤토리는 위의 내용을 기준으로 다음과 같은 내용을 가지고 있다.

호스트 이름

아이피 주소

호스트에서 사용하는 변수

앤서블 인벤토리

INVENTORY

[test]

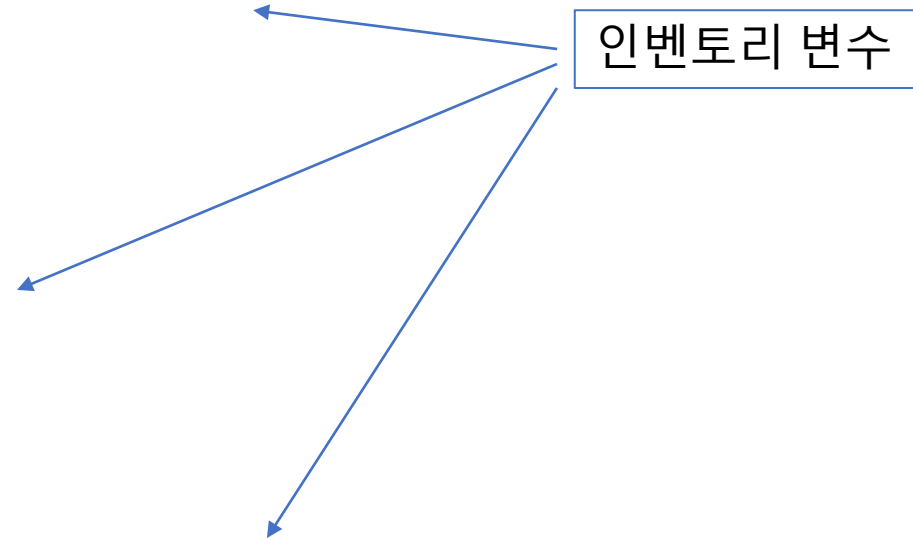
node1.example.com package=httpd

[db]

10.10.5.2 package=mariadb

[was]

was1.example.com package=wildfly




앤서블 인벤토리

INVENTORY

인벤토리 파일 기반으로 실행 시 다음과 같이 실행한다.

```
$ ansible-playbook -i hosts <PLAYBOOK>
```



인벤토리 옵션

앤서블 인벤토리

set to the inventory configuration in ansible.cfg

인벤토리 파일은 일반적으로 inventory라는 이름으로 구성함. 다른 이름으로 변경을 원하는 경우 ansible.cfg에서 변경이 가능함.

혹은 "-i"으로 임의적으로 선택 가능.

앤서블 인벤토리

Example of the ansible.cfg

```
$ cat ansible.cfg
```

```
[defaults]
```

```
inventory = <INVENTORY_FILE>
```

```
host_key_checking = False
```

```
remote_user = <SSH_USER>
```

```
ansible_ssh_port = <SSH_PORT>
```

```
ansible_ssh_user = <SSH_USER>
```

```
ansible_ssh_pass = <SSH_PASS>
```

```
forks = <PROCESSOR_FORK_NUMBER>
```

앤서블 인벤토리

예제

위의 내용은 아래 파일 참고

materials/sample-inventory
materials/sample-inventory-2
materials/sample-ansible.cfg

앤서블 인벤토리/설정파일

연습문제

ansible.cfg 및 인벤토리를 다음처럼 구성한다.

1. ansible.cfg에는 다음과 같은 내용이 구성이 되어야 한다.
2. "inventory_cloud"라는 파일을 기본 인벤토리로 불러와야 한다.
3. ssh 접근 포트는 8273번을 사용한다.
4. 접근 시 사용하는 사용자는 ansible, 사용자의 비밀번호는 ansible이다.
5. inventory_cloud라는 설정 파일에는 다음과 같은 서버를 등록한다.
10.10.10.1
10.20.30.5
192.168.90.*

앤서블 인벤토리/설정파일

연습문제

```
# vi ansible.cfg
```

```
[defaults]
```

```
inventory = inventory_cloud
```

```
ansible_ssh_port = 8273
```

```
ansible_ssh_user = ansible
```

```
ansible_ssh_pass = ansible
```

```
# vi inventory_cloud
```

```
10.10.10.1
```

```
10.20.30.5
```

```
192.168.90.*
```

앤서블 인벤토리/설정파일

연습문제

inventory_cloud에는 다음과 같이 설정 및 구성한다.

1. web, db라는 두개의 그룹이 존재하고 각각 그룹에 서버 하나씩 할당한다.
2. **web**에는 web.lab.example.com, **db**에는 db.lab.example.com 할당
3. 할당된 노드에는 nodename이라는 이름의 변수에 web.example.com, db.example.com라는 호스트 이름을 할당
4. together라는 그룹에 web, db가 동시에 사용이 가능하도록 한다

앤서블 인벤토리/설정파일

연습문제

```
# vi inventory_cloud  
[web]  
web.lab.example.com
```

```
[db]  
db.lab.example.com  
[together:children]  
web  
db
```

앤서블 인벤토리/설정파일

연습문제

```
# vi inventory_cloud  
[web]  
web.lab.example.com
```

```
[db]  
db.lab.example.com
```

```
[together:children]  
web  
db
```

```
# ansible -i inventory_cloud --list-hosts
```

셸 스크립트 활용하기

shell script

셸 스크립트 활용하기

앤서블 모듈은 파이썬 언어 기반으로 작성 권장.
하지만, 대 다수가 많이 사용하는 셸 언어 기반으로도 사용이 가능.

이 목차에서는 짧지만 어떻게 셸 스크립트 기반으로 작성하는지
그리고 집적 만들어서 테스트 해본다.

셸 스크립트 활용하기

여러 언어들을 사용하여 모듈을 만들 수 있지만, 간단한 지식을 통해서 빠르게 만들 수 있는 언어는 아마 셸 스크립트가 유일하다.

Perl, Python, Go Lang, PHP같은 언어로 만들려고 하는 경우 다시 언어에 대해서 기본적인 학습이 필요하기 때문에 그러한 과정을 최대한 줄이고 간결하게 사용할 수 있도록 리눅스에서 많이 사용하는 bash script기반으로 작성한다.

웹 스크립트 활용하기

앤서블 모듈은 다음과 같은 인자 값 핸들링 한다.

bashmodule:

dest: /tmp/hello

state: present

위와 같은 구조를 가지고 있다.

셸 스크립트 활용하기

이 변수를 셸에서는 다음과 같은 방식으로 핸들링 한다.

```
$ source $1
```

```
$ echo $1 $2
```

웹 스크립트 활용하기

앤서블 모듈은 다음과 규칙이 적용이 된다.

`changed` 성공적으로 작업이 완료가 된 경우 출력되는 메시지. 올바르게 수행이 되면 `true`, 수행이 되지 않으면 `false`으로 반환한다.

`failed`: 작업 수행이 실패한 경우, 해당 작업에 대해서 실패하면 `true`를 반환한다.

`msg`: 성공/오류/변경 같은 작업이 문제없이 수행이 되는 경우 메시지를 반환 합니다.

웹 스크립트 활용하기

위의 기능은 JSON기반으로 동작하면 메시지는 다음과 같은 형식으로 전달 합니다.

```
echo '{"changed": true, "msg": "Updated some thing"}'
```

반드시 모든 데이터는 JSON형태, 그리고 중괄호 안에서 홑 따옴표 및 쌍 따옴표로 전달이 되어야 한다. 다만, 안쪽에 있는 데이터 형식은 디렉터리 형태로 반드시 쌍 따옴표로 구별이 되어 있어야 한다.

셸 스크립트 활용하기

하지만, 셸 스크립트에서는 특수문자 핸들링 하기 위해서는 다음과 같이 두 가지 해결 방법이 있다.

```
echo "{\"changed\": true, \"msg\": \"Updated some thing\"}"
```

혹은 다음과 같이 bash에서 printf를 사용하여 출력이 가능하다.

```
printf '{"changed": true, "msg": "%s"}' "$msg"
```

셸에서 json형태로 출력 시 위의 같은 형태로 구현만 해주면 된다.

셸 스크립트 활용하기

하지만, bash또한 JSON형태로 출력이 불가능하기 때문에 파이썬 같은 도구를 사용해서 출력 도움을 받아야 한다.

```
msg="{\"changed\": true, \"msg\": \"$msg\"}"  
echo $(echo "$msg" | python -c 'import json,sys; print  
json.dumps(sys.stdin.read())')
```

셸에서 실행하면 다음과 같은 결과가 출력이 된다.

셸 스크립트 활용하기

간단하게 정리하면, echo, printf 둘 중 아무거나 사용하여도 됨.
올바르게 json 형태로 출력하면 된다.

셸 스크립트 활용하기

앤서블 모듈 작성시 꼭 필요한 기본 함수
명시된 함수는 없으나 모듈에서 사용하는 옵션 그대로 함수를 생성
해주어야 함.

예를 들어서 다음과 같은 옵션이 있으면 다음처럼 함수를 만들어야 함.

backup:

src: /usr/share/docs

dest: /tmp/docs.tar

label: 20220202_backup_ansible

셸 스크립트 활용하기

case:문법을 통해서 예외 처리가 필요함. 아래는 모듈에서 사용하는 옵션 핸들링 하는 부분.

```
case $state in
    present)
        create_file
        ;;
    absent)
        delete_file
        ;;
    upper)
        convert_to_upper
        ;;
```

셸 스크립트 활용하기

case:문법을 통해서 예외 처리가 필요함. 아래는 모듈에서 사용하는 옵션 핸들링 하는 부분.

```
lower)
    convert_to_lower
    ;;
*)
    printf '{"failed": true, "msg": "invalid state: %s"}' "$state"
    exit 1
    ;;
esac
printf '{"changed": %s, "msg": "%s", "contents": %s}' "$changed" "$msg" "$contents"
exit 0
```

앤서블 모듈

```
#!/bin/bash
```

```
function create_file
{
  if [ -f "$dest" ]; then
    changed="false"
    msg="file already exists"
  else
    echo 'Hello, "world!"' >> $dest
    changed="true"
    msg="file created"
  fi
  contents=$(cat "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
}
```

셸 스크립트 활용하기

```
function delete_file
{
  if [ -f "$dest" ]; then
    changed="true"
    msg="file deleted"
    contents=$(cat "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    output=$(rm -f $dest 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    if [ $? -ne 0 ]; then
      printf '{"failed": true, "msg": "error deleting file", "output": %s}' "$output"
      exit 1
    fi
  else
    changed="false"
    msg="file not present"
    contents=""
  fi
}
```

셸 스크립트 활용하기

```
function convert_to_upper
{
  if [ ! -f "$dest" ]; then
    create_file
    msg="$msg, "
  fi
  current=$(cat $dest)
  new=$(echo "$current" | tr '[:lower:]' '[:upper:]')
  if [ "$current" = "$new" ]; then
    changed="false"
    msg="{msg}file not changed"
    contents=$(printf "$current" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
  else
    changed="true"
    msg="{msg}file converted to upper case"
    contents=$(printf "$new" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
  fi
}
```

셸 스크립트 활용하기

```
function convert_to_lower
{
    if [ ! -f "$dest" ]; then
        create_file
        msg="$msg, "
    fi
    contents=$(ls -l "$dest" 2>&1 | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    current=$(cat $dest)
    new=$(echo "$current" | tr '[:upper:]' '[:lower:]')
    if [ "$current" = "$new" ]; then
        changed="false"
        msg="${msg}file not changed"
        contents=$(printf "$current" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    else
        echo "$new" > $dest
        changed="true"
        msg="${msg}file converted to lower case"
        contents=$(printf "$new" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
    fi
}
```


셸 스크립트 활용하기

```
source $1
```

```
if [ -z "$dest" ]; then  
    printf '{"failed": true, "msg": "missing required arguments: dest"}'  
    exit 1  
fi  
if [ -z "$state" ]; then  
    printf '{"failed": true, "msg": "missing required arguments: state"}'  
    exit 1  
fi
```

```
changed="false"  
msg=""  
contents=""
```

셸 스크립트 활용하기

```
case $state in
  present)
    create_file
    ;;
  absent)
    delete_file
    ;;
  upper)
    convert_to_upper
    ;;
  lower)
    convert_to_lower
    ;;
  *)
    printf '{"failed": true, "msg": "invalid state: %s"}' "$state"
    exit 1
    ;;
esac
printf '{"changed": %s, "msg": "%s", "contents": %s}' "$changed" "$msg" "$contents"
exit 0
```

셸 스크립트 활용하기

테스트 모듈에서 사용할 데이터를 임의로 작성한다.

```
$ cat <<EOF> test.txt
```

```
hello world tang and choi
```

```
EOF
```

셸 스크립트 활용하기

모듈 테스트 하기 위해서는 다음과 같은 명령어로 확인이 가능하다.

```
$ ansible localhost, -m bashmod -a 'dest=hello.txt state=upper'
```

셸 스크립트 활용하기

추가 기능인 제거를 실행한다.

```
$ ansible localhost, -m bashmod -a 'dest=hello.txt state=absent'
```