일일 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2022.08.24

세부 사항

1. 업무 내역 요약 정리

목표 내역	Done	Plan
1. 쉘 스크립트 언어 이해 및 활용 2. 미로게임 만들기 3. 오라클 데이터베이스 설치과정 복기	오늘은 버추얼박스 위에 CentOs7 을 설치한 뒤 xhell에 연결하는 것을 완료하였습니다. 그리고 쉘 스크립트 언어에 대해 부족한 개념들을 더 공부하고 여러 예제를 통해 연습을 하였습니다. 하지만 예제 중 6. Ride file into bash array가 잘 이해가 가지 않아 더 공부를 한 뒤 나중에 다시 봐야할 것 같습니다. 사용을 하다 보니 스크립트 언어라 그런지 파이썬과 비슷하다는 느낌이 들었습니다. 그리고 아까 교육받은 내용인 프로세스에 대해 조금 더 알아보고 정리하였습니다.	내일부터는 오늘 푼 예제들을 다시 살펴본 뒤 쉘 스크립트 언어에 대해서 조금 더 자세하게 공부한 뒤 아까 팀장님과 얘기한 미로게임을 개발해 보겠습니다.

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

1. read : 사용자로부터 입력을 받기 위한 명령어

```
#!/bin/bash

echo "당 신 의  나 이 를  입 력 하 세 요 "
read age
echo "당 신 의  키 를  입 력 하 세 요 "
read height

echo "당 신 의  나 이 는  $age 키 는  $height 입 니 다 ."
```

```
당 신 의  나 이 를  입 력 하 세 요
20
당 신 의  키 를  입 력 하 세 요
188
당 신 의  나 이 는  20 키 는  188 입 니 다 .
```

2. 숫자를 입력 받아 별 출력하기

- 별의 개수가 왼쪽에서 오른쪽으로 늘어남

```
#!/bin/bash

echo -n "type number : "
read num

a=1

while [ $a -le $num ]
do
        b=1
        while [ $b -le $a ]
        do
                echo -n "*"
                b=` expr $b + 1 `
        done
        echo ""
a=` expr $a + 1 `

done
exit 0
```

변수 num을 정할 숫자를 입력 받는다.
a=1, a가 num보다 작거나 같다면 반복해라
b=1 이고 , b가 a보다 작거나 같다면 반복해라
*을 출력해라, b는 1 씩 증가한다
a도 1 씩 증가한다, a가 num보다 크다면 끝내라

```
[testuser@lms sh3]$ sh star.sh
type number : 5
*
**
***
****
*****
```

- 별의 개수가 오른쪽에서 왼쪽으로 늘어남

```
#!/bin/bash

echo -n "type number : "
read num

c=1
while [ $c -le $num ]
do
        b=1
        a=` expr $num - $c + 1 `

        while [ $b -le $num ]
        do
                if [ $b -lt $a ]
                then
                        echo -n " "
                else
                        echo -n "*"
                fi
        b=` expr $b + 1 `
        done
        echo ""

c=` expr $c + 1 `
done
exit 0
```

변수 num을 정할 숫자를 입력 받는다.

c=1, c가 num보다 작거나 같으면 반복해라.

b=1, a= num -c +b

b가 num보다 작거나 같으면 반복해라

만일 b가 a보다 작으면

공백을 출력하고

아니면 *을 출력해라

b는 1 씩 증가한다.

c는 1 씩 증가한다

c가 num보다 크다면 중단해라

```
[testuser@lms sh3]$ sh star2.sh
type number : 5
    *
   **
  ***
 ****
*****
```

echo -n : 마지막에 따라오는 개행 문자(new line)를 출력하지 않는다.

*개행문자 : 컴퓨터에서 줄바꿈을 나타내는 제어문자

3. Simple Backup Bash

```
#!/bin/bash

tar -czf star.tar.gz star.sh
```

```
[testuser@lms sh3]$ chmod 777 Backup.sh
[testuser@lms sh3]$ ./Backup.sh
[testuser@lms sh3]$ ls
Backup.sh  read.sh  star.sh  star.tar.gz  star2.sh
```

star2.sh를 star.tar.gz라는 파일명으로 압축하여 저장한 것이다.

4. Global vs Local variables

```
#!/bin/bash

VAR="global variable"
function bash {
local VAR="local variable"
echo $VAR
}
echo $VAR
#call the function
bash
echo $VAR
```

위 코드는 지역변수와 전역변수의 차이를 알아보는 예제로 지역변수와 전역변수 둘 다 VAR이라는 같은 변수명으로 선언하고 출력했다. 여기서 지역변수는 함수 안에서만 작용한다. 따라서 함수 밖의 처음 echo에서는 전역변수 global이 출력되고 다음 bash함수를 호출했을 때는 함수 안에서만 작용하는 local이 출력되고 마지막 echo에서는 다시 전역변수가 출력됨을 알 수 있다.

```
[testuser@lms sh3]$ sh gv.sh
global variable
local variable
global variable
```

5. Declare simple bash array

```
#!/bin/bash

ARRAY=('Debian Linux' 'Redhat Linux' Ubuntu Linux)
ELEMENTS=${#ARRAY[@]}
for ((i=0; i<$ELEMENTS; i++));
do
        echo ${ARRAY[${i}]}
done
```

쉘 스크립트에서 배열은 C언어랑 다르게 쉼표로 값들을 구분하는 것이 아닌 공백으로 구분한다. 그리고 작은

따옴표는 변수를 포함하지 않을 때 사용하고 큰 따옴표는 변수를 포함할 때 사용한다.
ARRAY라는 배열에 4 개의 값이 할당되어 있다. 그리고 배열의 길이를 계산해주는 ${#배열명[@]}을 ELEMENTS에 저장하여 이를 반복문의 횟수로 줌으로서 배열에 저장되어 있는 값을 차례로 출력하는 코드이다.

```
[testuser@lms sh3]$ sh arr.sh
Debian Linux
Redhat Linux
Ubuntu
Linux
```

6. Read file into bash array

```
vi bash.txt
```

```
Bash
Scripting
Tutorial
Guide
```

```
vi bash_script.sh
```

```
#!/bin/bash

declare -a ARRAY
exec 10<&0
exec < $1
let count=0

while read LINE; do
        ARRAY[$count]=$LINE
        ((count++))
done
echo Number of elements: ${#ARRAY[@]}
echo ${ARRAY[@]}
exec 0<&10 10<&-
```

declare : 변수 값을 선언한다.
declare -a : 각 이름을 배열 변수로 선언한다.
exec : 주어진 명령어를 실행하는데 새로운 프로세스를 생성하지 않고, 쉘 프로세스를 대체한다.

위 예제는 파일 입출력을 다룬 예제로 bash_script.sh에서 bash.txt의 값을 읽어 오는 것이다.
파일에서 값을 읽어오려면 읽어올 파일이 존재해야 하므로 bash.txt라는 이름의 파일을 만들어준다.
다음으로 bash_script.sh라는 파일을 만들어서 LINE으로 입력 받았다.
그리고 반복문 안에서 count로 인덱스를 설정해서 ARRAY에 차례로 읽어온 값을 저장한다.
그리고 마지막으로 입력 받은 값들을 echo를 통해 인자의 개수와 bash.txt에서 한 줄씩 읽어온 값들을 공백으로 구분하여 아래와 같이 출력한다.

```
[testuser@lms sh3]$ sh bash_script.sh bash.txt
Number of elements: 4
Bash Scripting Tutorial Guide
```

7. Simple Bash if/else statement

```
#!/bin/bash

directory="./BashScripting"
if [ -d $directory ]; then
        echo "Directory exits"
else
        echo "Directory does not exists"
fi
```

위 코드는 if문을 이용하여 BashScripting 디렉터리가 존재하면 존재한다고 없으면 존재하지 않는다고 출력해주는 예제다. 여기서는 if문에 참 거짓 조건문으로 $directory를 주어 존재하는 것을 참인 조건으로 설정해서 fi로 끝낸 것이다.

-d 파일명 : 파일이 디렉토리면 참

```
[testuser@lms sh3]$ sh bashifel.sh
Directory does not exists
```

8. Nested if/else

```
#!/bin/bash

choice=4
echo "1. Bash"
echo "2. Scripting"
echo "3. Tutorial"
echo -n "Please choose a word [1, 2 or 3]?"
while [ $choice -eq 4 ]; do
read choice
if [ $choice -eq 1 ] ; then
        echo "You have chosen word : Bash"
else
        if [ $choice -eq 2 ] ; then
                echo "You have chosen word : Scripting"
        else
                if [ $choice -eq 3 ] ; then
                        echo "You have chosen word : Tutorial"
                else
                        echo "Please make a choice between 1-3 !"
                        echo "1. Bash"
                        echo "2. Scripting"
                        echo "3. Tutorial"
                        echo -n "Please choose a word [1, 2 or 3]?"
                        choice=4
                fi
        fi
fi
done
```

위 코드는 쉘 스크립트에서 반복문과 조건문을 사용한 예제이다.
먼저 choice 변수를 4 로 선언해주고 read choice로 값을 입력 받아서
입력 받은 값이 1-3 사이이면 해당하는 값을 출력해주고 자연스럽게 choice=1-3 으로 choice가 4 인동안 반복하는 조건문을 나가서 프로세스를 종료하는데 입력 받은 값이 1-3 사이가 아니면 가장 안쪽의 else문에서 choice를 4 로 초기화 시켜 반복문을 계속 돌게 함을 안 수 있다. 따라서 아래와 같이 6 을 입력했을 땐 반복문을 다시 돌고 1-3 사이의 값을 입력하면 스크립트가 종료된다.

```
[testuser@lms sh3]$ sh nested.sh
1. Bash
2. Scripting
3. Tutorial
Please choose a word [1, 2 or 3]?6
Please make a choice between 1-3 !
1. Bash
2. Scripting
3. Tutorial
Please choose a word [1, 2 or 3]?2
You have chosen word : Scripting
```

9. Arithmetic Comparisons

```
#!/bin/bash

a=2
b=2
c=3
d=4
if [ $a -eq $b ]; then
        echo " a and b is equal "
else
        echo " a and b is not equal"
fi

if [ $c -eq $d ]; then
        echo " c and d is equal "
else
        echo " c and d is not equal"
fi
```

첫 번째 if문에서 a와 b가 같으면 같다고 다르면 다르다고 출력해주는데 a와 b는 같기 때문에 첫 출력은 같다고 나갈 것이다.
다음 if문도 마찬가지로 c와 d를 비교하여 같은지 다른지 알려주는데 c와 d는 다르므로 두번째 출력은 다르다고 나갈 것이다.

```
[testuser@lms sh3]$ sh statement.sh
 a and b is equal
 c and d is not equal
```

10. String Comparisons

```
#!/bin/bash

a="Bash"
b="scripting"
c="silc"
d="silc"
if [ $a = $b ]; then
        echo " a and b is equal "
else
        echo " a and b is not equal"
fi

if [ $c = $d ]; then
        echo " c and d is equal "
else
        echo " c and d is not equal"
fi
```

10 번은 9 번과 비슷하게 값이 같은지 다른지를 비교하는 예제인데 9 번은 숫자이고 10 번은 문자열을 비교한다. 문자열이기에 -eq가 아닌 =를 사용한 것 외에 차이점이 없다.

```
[testuser@lms sh3]$ sh ste.sh
 a and b is not equal
 c and d is equal
```

11. Bash for loop

```
#!/bin/bash

for f in $( ls /var/ ); do
        echo $f
done
```

위 코드는 쉘 스크립트의 반복문을 이용하였다. 그런데 특이한 점은 ls /var/ 명령을 사용해서 /var 디렉토리에 있는 값을 하나씩 출력하도록 반복문을 사용하였다. 따라서 위의 ls /var/ 명령을 사용했을 때 나오는 반복문으로 출력하는 코드이다.

```
[testuser@lms sh3]$ sh loop.sh
account
adm
cache
crash
db
empty
games
gopher
kerberos
lib
local
lock
log
mail
nis
opt
preserve
run
spool
target
tmp
yp
```

12. bash while loop

```
#!/bin/bash

COUNT=6
while [ $COUNT -gt 0 ]; do
        echo Value of count ls: $COUNT
        let COUNT=COUNT-1
done
```

값 1 -gt 값 2 : 값 1 이 값 2 보다 큼

위 while문에서는 조건식 -gt를 사용하여 count가 0 보다 클 때 까지 반복하는 반복문이다. 따라서 반복문 안에서 count값을 하나씩 줄여가며 6 번 출력한다.

```
[testuser@lms sh3]$ sh wloop.sh
Value of count ls: 6
Value of count ls: 5
Value of count ls: 4
Value of count ls: 3
Value of count ls: 2
Value of count ls: 1
```

13. bash until loop

```
#!/bin/bash

COUNT=0
until [ $COUNT -gt 5 ]; do
        echo Value of count is : $COUNT
        let COUNT=COUNT+1
done
```

위 코드에서 COUNT를 0 으로 선언하고 루프 안에서 COUNT를 1 씩 증가시킨다. 그런데 until문은 조건이 거짓일 때 동작하므로 COUNT가 5 보다 클 때 까지라는 조건이 틀릴 때 즉 COUNT가 5 일 때까지 동작하다가 COUNT=5 를 출력하고 COUNT가 6 이 되었을 때는 조건문이 참이 되므로 루프를 탈출한다.

```
[testuser@lms sh3]$ sh until.sh
Value of count is : 0
Value of count is : 1
Value of count is : 2
Value of count is : 3
Value of count is : 4
Value of count is : 5
```

14. Control bash loop with

```
#!/bin/bash

DIR="."
find $DIR -type f | while read file; do
if [[ "$file" = *[[:space:]]* ]]; then
mv "$file" `echo $file | tr ' ' '_'`
fi;
done
```

4 번째 라인에서 파이프를 이용하여 find 명령의 출력을 while문의 입력으로 주고 있는 것을 확인할 수 있다. find 명령에서는 현재 디렉토리에서 일반 파일을 찾고 이 결과를 while문으로 file에 저장하여 다음으로는 6 번째 라인에서 tr명령을 사용하여 공백을 언더바(_)로 대체 하였다.

```
[testuser@lms sh3]$ ls -lt
합 계  68
-rw-rw-r--. 1 testuser testuser   0  8월  24 14:34 a b c d
```

```
[testuser@lms sh3]$ sh 14.sh
[testuser@lms sh3]$ ls -lt
합 계  68
-rw-rw-r--. 1 testuser testuser   0  8월  24 14:34 a_b_c_d
```

15. Escaping Meta characters

```
#!/bin/bash

a="silc road"
echo $a
echo '$a'
echo '\'
```

위 코드는 따옴표의 역할을 알 수 있다. 일반적으로 echo $변수명 명령(큰 따옴표 생략가능)을 사용시 변수의 값을 출력해주지만 작은 따옴표로 감싸주면 글자 그대로 출력하기 때문에 첫 번째 줄에는 변수의 값이, 두 번째 줄에는 글자 그대로 세 번째 줄 또한 글자 그대로 출력된다.

```
[testuser@lms sh3]$ sh 15.sh
silc road
$a
\
```

16. Single quotes

```
#!/bin/bash

a="silc road"
echo $a
echo '$a "$a"'
```

위 코드 또한 15 번과 마찬가지로 작은 따옴표로 문자열을 감싸주면 $변수명을 사용하여도 그대로 출력함을 보여주는 예제이다. 따라서 처음 echo $BASH_VAR은 변수의 값을 출력하고 다음 출력문에 "$BASH_VAR"을 통해 큰 따옴표는 변수의 값을 출력하지만 제일 바깥의 따옴표가 작은 따옴표이면 큰 따옴표 또한 문자로 그대로 출력됨을 알 수 있다.

```
[testuser@lms sh3]$ sh 16.sh
silc road
$a "$a"
```

17. Double Quotes

```
#!/bin/bash

a="silc road"
echo $a
echo "It's $a and \"$a\" using backticks: `date`"
```

이 코드는 큰 따옴표안에 $변수명은 그대로 출력하지만 앞에 역슬래쉬를 붙여주면 문자의 값을 출력해주는 특수기호를 글자 그래도 출력해줌을 알 수 있는 예제이다. 따라서 Line4 에서는 변수의 값을 출력하고 Line5 에서 두번째 변수 출력문은 앞에 큰 따옴표를 붙여서 큰 따옴표 그대로를 출력하는 코드이다.

```
[testuser@lms sh3]$ sh 17.sh
silc road
It's silc road and "silc road" using backticks: 2022. 08. 24. (수 ) 14:53:30 KST
```

18. Bash quoting with ANSI-C style

```
#!/bin/bash

echo $'web : www.google.com\neamil:chocomin0211\x40gmail.com'
```

위 코드는 ₩n을 사용해서 줄을 바꿔서 출력하는 간단한 예제이다. ₩x40 은 @을 출력한다.

```
[testuser@lms sh3]$ sh 18.sh
web : www.google.com
eamil:chocomin0211@gmail.com
```

19. Bash Addition Calculator Example

```
#!/bin/bash

let RESULT1=$1+$2
echo $1+$2=$RESULT1 ' -> # let RESULT1=$1+$2 '
declare -i RESULT2
RESULT2=$1+$2
echo $1+$2=$RESULT2 ' -> # declare -i RESULT2; RESULT2=$1+$2 '
echo $1+$2=$(($1 + $2)) ' -> # $(($1 + $2)) '
```

위 코드는 문자열 연산의 예제로 line4 에서 RESULT2 를 정수로 declare해 주고 첫 번째 인자와 두 번째 인자를

뜻하는 $1, $2 를 더해주는 코드이다. 즉 입력된 두 숫자를 더해준다.

```
[testuser@lms sh3]$ sh 19.sh 40 60
40+60=100  -> # let RESULT1=$1+$2
40+60=100  -> # declare -i RESULT2; RESULT2=$1+$2
40+60=100  -> # $(($1 + $2))
```

20. Bash Arithmetics

```
#!/bin/bash

echo '### let ###'
let addition=3+5
echo "3 + 5 =" $addition
let sub=7-8
echo "7 - 8 =" $sub
let mul=5*8
echo "5 * 8 =" $mul
let div=4/2
echo "4 / 2 =" $div
let mod=9%4
echo "9 % 4 =" $mod
let pow=2**2
echo "2 ^ 2 =" $pow

echo '### Bash Arithmetic Expansion ###'
echo 4 + 5 = $((4+5))
echo 7 - 7 = $[ 7-7 ]
echo 4 x 6 = $((4*6))
echo 6 / 3 = $((6/3))
echo 8 % 7 = $((8%7))
echo 2 ^ 8 = $[2**8]

echo '### Declare ###'
echo -e "please enter two numbers \c"
read num1 num2
declare -i result
result=$num1+$num2
echo "Result is:$result"
result=2#10001
echo $result
result=8#16
echo $result
result=16#E6A
echo $result
```

Let은 새 변수를 만들어 연산하는 것이고 expr은 만들지 않고 연산하는 것이다. 이중 20 번에서는 let명령을 사용하여 연산하였다. 위 코드는 크게 세 덩어리로 나눌 수 있다. 첫 번째 덩어리는 간단하게 let 명령을 사용하여 지정한 문자를 연산하는 작업을 수행한다. 두 번째 덩어리는 산술확장$(())를 이용하여 지정한 숫자를 연산하고 마지막 덩어리에서는 숫자 두 개를 입력 받고 더해준 뒤 각각을 2 진수 8 진수 16 진수로 출력해준다.

```
Result is:
17
14
3690
[testuser@lms sh3]$ sh 20.sh
### let ###
3 + 5 = 8
7 - 8 = -1
5 * 8 = 40
4 / 2 = 2
9 % 4 = 1
2 ^ 2 = 4
### Bash Arithmetic Expansion ###
4 + 5 = 9
7 - 7 = 0
4 x 6 = 24
6 / 3 = 2
8 % 7 = 1
2 ^ 8 = 256
### Declare ###
please enter two numbers 20 10
Result is:30
17
14
3690
```

프로세스

- 프로세스 : 컴퓨터에서 연속적으로 실행되고 있는 컴퓨터 프로그램
- 멀티 프로세싱 : 여러 개의 프로세서를 사용하는 것
- 멀티태스킹 : 같은 시간에 여러 개의 프로그램의 띄우는 시분할 방식
- 프로그램 vs 프로세스 : 프로그램은 일반적으로 하드 디스크 등에 저장되어 있는 실행코드를 뜻하고, 프로세스는 프로그램을 구동하여 프로그램 자체와 프로그램의 상태가 메모리상에서 실행되는 작업 단위를 지칭한다.
- ps 명령어 : 현재 실행중인 프로세스 목록과 상태를 보여줌

Ps	Pid, tty, time, cmd만 보여줌
Ps -l	상세내역을 보여줌
Ps -e	모든 프로세스를 보여줌
Ps -ef	모든 프로세스의 모든 정보를 출력(uid, pid, ppid, c, stime, tty, time, cmd)
Ps -efc	명령이름까지 보여줌

uid : 유저 id

pid : 프로세스의 아이디

ppid : 프로세스의 부모 pid

c : cpu 사용량

stime : 프로세스 시작 시간

tty : 프로세스를 제어하는 수단 (콘솔 접속시 tty숫자 형태로 표시되며, 원격이나 에뮬레이터 접속시 pts/숫자 형태로 표시)

time : 프로세스에 사용된 cpu 시간
cmd : 프로세스 실행 명령어

- 프로세스를 죽이는 법 : kill 명령어 사용 kill – 옵션 pid
  -9 : 강제 종료, -15 : 작업 종료

  *$user : 사용자명을 담은 리눅스 환경변수
  *ps -ef | gerp $user : 내가 로그인 되어있는 user이 사용하는 프로세스를 보여줌