3단원. 쉘 스크립트 기본 문법

> 대상: 전공자 · 정보처리기사 취득자 | 목적: 인프라 직무(네트워크·클라우드·시스템·보안) 기술 면접 대비 중요도: ★★★ 최빈출 · ★★ 자주 출제 · ★ 기본 개념

> **Q1.** 쉬뱅(**shebang**)과 쉘 스크립트에서의 변수 선언 및 사용 방법을 설명하세요. ★★

답안 쉬뱅은 스크립트 파일 첫 줄에 `#!/bin/bash`처럼 작성해 이 스크립트를 어떤 인터프리터로 실행할지 커널에게 알려주는 지시자입니다. 쉬뱅이 없으면 현재 사용 중인 셸이 스크립트를 해석하려 시도하므로 이식성 문제가 생길 수 있어, `#!/usr/bin/env bash`처럼 환경에 따라 인터프리터 경로를 유연하게 찾는 방식도 널리 쓰입니다. 변수는 `NAME=value` 형태로 선언하며 등호 앞뒤에 공백이 있으면 안 되고, 값을 참조할 때는 `$NAME` 또는 `${NAME}`을 사용합니다. 셸 변수는 기본적으로 문자열로 취급되며, 다른 프로세스에도 값을 물려주려면 `export`로 환경변수로 승격시켜야 합니다. `readonly`로 상수를 선언하거나 `unset`으로 변수를 제거하는 것도 기본 문법에 포함됩니다.

관련 개념 #!/usr/bin/env, export, readonly, 환경변수 vs 셸 변수

> **Q2.** 쉘 스크립트의 조건문(**if, case**) 사용법을 설명하세요. ★★

답안 `if` 문은 `if [ 조건 ]; then ... elif [ 조건 ]; then ... else ... fi` 구조로 작성하며, 대괄호 `[ ]`는 사실 `test` 명령어의 다른 표현이므로 대괄호 안팎에 공백을 반드시 넣어야 합니다. 문자열 비교는 `=` / `!=`, 숫자 비교는 `-eq` / `-ne` / `-

`gt` / `-lt` 등을 사용하며, 파일 존재 여부는 `-f`(일반 파일), `-d`(디렉토리), `-e`(존재) 등으로 확인합니다. 최신 bash에서는 이중 대괄호 `[[ ]]`를 사용하면 패턴 매칭이나 논리 연산자(`&&`, `||`)를 더 안전하게 쓸 수 있습니다. **`case`** 문은 하나의 변수 값을 여러 패턴과 비교해 분기할 때 `if-elif`보다 가독성이 좋으며, `case $var in pattern1) ... ;; pattern2) ... ;; *) ... ;; esac` 형태로 작성하고 각 분기는 `;;`로 종료합니다.

관련 개념 [ ] vs [[ ]], test 명령어, 파일 테스트 연산자(-f/-d/-e), case의 ;; 종료

> **Q3.** 쉘 스크립트의 반복문(**for, while**)을 설명하세요. ★★

답안 **`for`** 문은 리스트나 범위를 순회할 때 사용하며 `for i in 1 2 3; do ... done`처럼 값 목록을 직접 순회하거나, `for file in *.log; do ... done`처럼 파일 글롭 패턴을, 또는 `for ((i=0; i<10; i++))`처럼 C 스타일 산술 반복도 지원합니다. **`while`** 문은 조건이 참인 동안 반복하는 구조로 `while [ 조건 ]; do ... done` 형태이며, 파일을 한 줄씩 읽어 처리할 때 `while read -r line; do ... done < file.txt`와 같은 패턴이 로그 처리 자동화에서 매우 자주 사용됩니다. 무한 루프는 `while true; do ... done`으로 만들고 `break`로 탈출, `continue`로 다음 반복으로 건너뛸 수 있습니다. `read -r`에서 `-r` 옵션은 백슬래시를 이스케이프 문자로 해석하지 않도록 막아주므로 파일을 읽을 때는 관례적으로 항상 붙이는 것이 좋습니다.

관련 개념 while read -r, 파일 글롭, break/continue, C 스타일 for, IFS

> **Q4.** 쉘 스크립트에서 함수를 정의하고 인자를 처리하는 방법을 설명하세요. ★

답안 셸 함수는 `function_name() { 명령어들 }` 형태로 정의하며, `function` 키워드는 bash에서는 선택 사항입니다. 함수 내부에서 전달받은 인자는 `$1`, `$2`처럼 위치 매개변수로 접근하고, 전체 인자 개수는 `$#`, 모든 인자는 `$@`나 `$*`로 참조합니다. 다른 프로그래밍 언어와 달리 셸 함수는 값을 `return`으로 반환할 때 정수

(0~255) 형태의 종료 상태 코드만 반환할 수 있으므로, 문자열 결과값이 필요하면 `echo`로 출력한 뒤 호출부에서 `$(function_name)`처럼 커맨드 치환으로 받아야 합니다. 함수 내에서 선언한 변수를 함수 지역 변수로 한정하려면 `local` 키워드를 사용해야 하며, 이를 생략하면 기본적으로 전역 변수가 되어 스크립트 다른 부분과 충돌할 수 있습니다.

관련 개념 local 변수, $1/$#/$@, 커맨드 치환 $(), return 코드 제한

> **Q5.** 표준입출력 리다이렉션(>, >>, <, 2>, &>)을 설명하세요. ★★★

답안 리다이렉션은 프로세스의 표준입출력을 파일이나 다른 대상으로 연결을 바꾸는 기능입니다. `>`는 표준출력을 파일에 덮어쓰기로 저장하고, `>>`는 기존 내용 끝에 이어 붙입니다(append). `<`는 파일 내용을 표준입력으로 넘겨주며, `2>`는 표준에러(FD 2)만 파일로 리다이렉션합니다. 표준출력과 표준에러를 모두 하나의 파일로 합치려면 `command > file.log 2>&1`처럼 순서에 유의해서 작성해야 하며(먼저 stdout을 파일로 보낸 뒤 stderr를 stdout이 가리키는 곳으로 복사), 최신 bash에서는 `&>`로 한 번에 처리할 수도 있습니다. 에러 메시지를 무시하고 싶을 때는 `2>/dev/null`로 버리는 패턴이 로그 자동화 스크립트에서 매우 자주 사용됩니다.

관련 개념 2>&1의 순서, /dev/null, append vs overwrite, FD 재지정

> **Q6.** 파이프(|)의 동작 원리와 활용 예시를 설명하세요. ★★★

답안 파이프(`|`)는 한 명령어의 표준출력을 다른 명령어의 표준입력으로 바로 연결해 주는 IPC 메커니즘으로, 커널이 두 프로세스 사이에 커널 버퍼(파이프 버퍼)를 만들어 중간에 임시 파일을 거치지 않고 데이터를 흘려보냅니다. 이를 이용해 여러 명령어를 조합해 복잡한 텍스트 처리 파이프라인을 구성할 수 있는데, 예를 들어 `ps aux | grep nginx | awk '{print $2}'`는 nginx 관련 프로세스만 걸러내 PID만 추출합니다. 파이프로 연결된 각 명령어는 병렬적으로(스트리밍 방식으로) 실행되며, 앞 명령의 출력이 생성되는 대로 뒤 명령이 소비할 수 있어 대용량 로그도 메모리 효율적으로 처리할 수 있습니다. 파이프라인 전체의 종료 코드는 기본적으로 마

지막 명령의 종료 코드를 따르며, 중간 명령의 실패까지 확인하려면 `PIPESTATUS` 배열이나 `set -o pipefail`을 사용해야 합니다.

관련 개념 프로세스 간 통신(IPC), PIPESTATUS, pipefail, 스트리밍 처리

> **Q7.** 쉘 스크립트의 종료 코드($?)와 **exit** 사용법을 설명하세요. ★★

답안 리눅스의 모든 명령어와 스크립트는 종료될 때 0~255 사이의 정수 종료 코드 **(exit status)**를 반환하며, 관례상 0은 성공, 0이 아닌 값은 실패나 특정 오류 상황을 의미합니다. 방금 실행한 명령의 종료 코드는 특수 변수 `$?`로 즉시 확인할 수 있으며, 다음 명령을 실행하면 값이 갱신되므로 필요한 시점에 바로 확인해야 합니다. 스크립트 안에서 `exit N`을 호출하면 그 시점에서 스크립트 실행을 즉시 종료하고 N을 자신의 종료 코드로 반환하며, `exit`을 생략하면 스크립트의 마지막 명령의 종료 코드가 그대로 사용됩니다. 이 값은 `command1 && command2`(성공 시에만 다음 실행), `command1 || command2`(실패 시에만 다음 실행) 같은 논리 연산자와 결합해 자동화 스크립트의 흐름 제어에 핵심적으로 활용됩니다.

관련 개념 $?, exit N, && / ||, 관례적 종료 코드 의미

> **Q8.** 쉘 스크립트의 특수 변수($0, $#, $@, $$, $!)를 설명하세요. ★

답안 `$0`은 실행 중인 스크립트 자신의 이름(경로)을 가리키고, `$1, $2...`는 스크립트에 전달된 위치 인자를 의미합니다. `$#`은 전달된 인자의 총 개수, `$@`는 모든 인자를 개별 항목으로, `$*`는 모든 인자를 하나의 문자열로 결합해 나타냅니다(`"$@"` 처럼 큰따옴표로 감쌌을 때 이 둘의 동작 차이가 뚜렷해집니다). `$$`는 현재 실행 중인 스크립트(셸)의 PID이고, `$!`는 가장 최근에 백그라운드로 실행한 프로세스의 PID를 담고 있어 백그라운드 작업을 나중에 `wait $!`로 기다리거나 종료시킬 때 활용됩니다. 이 변수들은 로그 파일명에 PID를 붙이거나, 인자 개수를 검증해 사용법 오류를 처리하는 스크립트에서 자주 사용됩니다.

관련 개념 "$@" vs "$*", 위치 매개변수, wait $!, PID 기반 로그 파일명

> **Q9.** 쉘 스크립트에서 배열을 선언하고 활용하는 방법을 설명하세요. ★

답안 bash에서 배열은 `arr=(a b c)`처럼 괄호로 값을 나열해 선언하며, 개별 요소는 `${arr[0]}`처럼 인덱스로 접근합니다(인덱스는 0부터 시작). 배열 전체 요소는 **`${arr[@]}`**로, 요소 개수는 **`${#arr[@]}`**로 확인할 수 있으며, `for item in "${arr[@]}"; do ... done`처럼 순회할 때는 요소에 공백이 포함될 수 있으므로 반드시 큰따옴표로 감싸는 것이 안전합니다. 배열에 요소를 추가할 때는 `arr+=("new")`를 사용하고, 연관 배열(키-값 쌍)을 쓰려면 `declare -A`로 선언해야 합니다. 배열은 여러 서버 목록을 순회하며 동일 작업을 반복하는 자동화 스크립트에서 자주 활용됩니다.

관련 개념 ${arr[@]}, declare -A, "${arr[@]}"의 안전한 인용, 연관 배열

> **Q10.** 쉘 스크립트에서 작은따옴표와 큰따옴표의 차이를 설명하세요. ★★

답안 작은따옴표(`'...'`)로 감싼 문자열은 내부의 모든 문자를 있는 그대로 취급해 변수 치환($)이나 커맨드 치환이 전혀 일어나지 않습니다. 큰따옴표(`"..."`)로 감싼 문자열은 변수 치환(`$VAR`)과 커맨드 치환(`$(...)`)은 그대로 수행하지만, 파일명 글롭 확장이나 단어 분리(word splitting)는 막아줍니다. 예를 들어 공백이 포함된 파일 경로를 다룰 때 `$FILE`처럼 인용 없이 쓰면 단어 분리로 인해 여러 인자로 쪼개져 스크립트 오류의 흔한 원인이 되므로, 변수를 참조할 때는 습관적으로 `"$FILE"`처럼 큰따옴표로 감싸는 것이 안전한 스크립팅의 기본 원칙입니다. 반대로 정규식이나 리터럴 문자열을 그대로 전달해야 할 때는 작은따옴표가 더 적합합니다.

관련 개념 단어 분리(word splitting), 글롭 확장 방지, 커맨드 치환, 변수 인용 습관
