Linux开发与工具面试题

> 基于简历：Git版本管理、CMake构建、性能分析工具（perf/valgrind）、Linux开发环境

一、Linux基础命令

Q1: 常用的文件操作命令?

查看文件：

```
ls -la          # 详细列表，包括隐藏文件
cat file.txt    # 查看文件内容
less file.txt   # 分页查看
head -n 10 file.txt  # 前10行
tail -n 10 file.txt  # 后10行
tail -f log.txt      # 实时监控日志
```

文件查找：

```
find /path -name "*.cpp"              # 按名称查找
find /path -type f -mtime -7          # 7天内修改的文件
find /path -size +100M                # 大于100M的文件
grep -r "keyword" /path               # 递归搜索内容
```

文件权限：

```
chmod 755 file.sh    # rwxr-xr-x
chmod +x file.sh     # 添加执行权限
chown user:group file  # 改变所有者
```

磁盘相关：

```
df -h           # 磁盘使用情况
du -sh dir      # 目录大小
du -h --max-depth=1  # 当前目录下各子目录大小
```

与你项目的联系： "在开发文件传输系统时，用`du`命令查看测试文件大小，用`tail -f`实时查看服务器日志。"

Q2: 进程管理命令?

查看进程：

```
ps aux           # 所有进程
ps -ef           # 标准格式
ps -T -p <pid>   # 查看进程的线程
top              # 实时监控
top -H -p <pid>  # 查看进程的线程详情
htop             # top的增强版
```

进程控制：

```
kill -9 <pid>       # 强制杀死进程
killall process     # 杀死所有同名进程
pkill -f "pattern"  # 按模式杀死进程

# 后台运行
./program &
nohup ./program &   # 不挂断运行

# 查看后台任务
jobs
fg %1               # 切换到前台
bg %1               # 继续后台运行
```

系统监控：

```
free -h          # 内存使用
vmstat 1         # 系统性能，每秒刷新
iostat           # I/O统计
netstat -tulnp   # 网络连接
ss -tulnp        # netstat的替代品（更快）
```

与你项目的联系： "压测文件传输系统时，用`top`查看CPU占用，用`free`查看内存，用`ss`查看连接数。"

Q3: 如何查看某个端口被哪个进程占用?

方法1：netstat

```
netstat -tulnp | grep 8080
# 输出：tcp  0  0  0.0.0.0:8080  0.0.0.0:*  LISTEN  12345/./server
# 进程ID是12345
```

方法2：lsof

```

```

```
lsof -i :8080
# 输出：
# COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
# server  12345 user    3u  IPv4  12345      0t0  TCP *:8080 (LISTEN)
```

方法3：ss

```
ss -tulnp | grep 8080
```

杀死占用端口的进程：

```
# 一行命令
kill -9 $(lsof -t -i :8080)
```

二、Git版本管理

Q4: Git基本工作流程?

本地操作：

```
# 初始化仓库
git init
git clone <url>

# 查看状态
git status
git diff        # 查看修改
git diff --cached  # 查看暂存区修改

# 添加和提交
git add file.cpp
git add .
git commit -m "Add feature X"

# 查看历史
git log
git log --oneline
git log --graph --all

# 撤销修改
git checkout -- file.cpp  # 撤销工作区修改
git reset HEAD file.cpp   # 取消暂存
git reset --hard HEAD^    # 回退到上一个版本
```

分支操作：

```
# 创建和切换分支
git branch feature-x
git checkout feature-x
git checkout -b feature-x  # 创建并切换

# 查看分支
git branch         # 本地分支
git branch -r      # 远程分支
git branch -a      # 所有分支

# 合并分支
git checkout main
git merge feature-x

# 删除分支
git branch -d feature-x   # 安全删除（已合并）
git branch -D feature-x   # 强制删除
```

远程操作：

```
# 查看远程仓库
git remote -v

# 拉取和推送
git fetch origin     # 获取远程更新（不合并）
git pull origin main  # 拉取并合并
git push origin main  # 推送

# 推送新分支
git push -u origin feature-x
```

Q5: Git冲突如何解决?

冲突产生：

```
git pull origin main
# 如果本地和远程修改了同一文件的同一位置，产生冲突
```

冲突标记：

```
<<<<<<< HEAD
int x = 10;  // 本地修改
=======
```

```
int x = 20;  // 远程修改
>>>>>>> origin/main
```

解决步骤：

```
# 1. 编辑文件，选择保留哪个版本
int x = 10;  # 或者 int x = 20; 或者其他合并方案

# 2. 标记已解决
git add file.cpp

# 3. 提交
git commit -m "Resolve conflict"
```

避免冲突：

- 提交前先pull
- 小步提交，频繁同步
- 不同功能在不同分支开发

Q6: Git常用高级操作?

1. 暂存工作区（stash）

```
# 暂存当前修改
git stash

# 查看暂存
git stash list

# 恢复暂存
git stash pop        # 恢复并删除stash
git stash apply      # 恢复但保留stash

# 应用场景：
# 正在开发功能A，突然要修bug，可以先stash，改完bug后再pop
```

2. 修改提交（amend）

```
# 修改最后一次提交
git add forgotten_file.cpp
git commit --amend

# 修改提交信息
git commit --amend -m "New message"
```

3. 交互式rebase

```
# 合并最近3次提交
git rebase -i HEAD~3

# 进入编辑器，选择操作：
# pick   = 保留提交
# squash = 合并到上一个提交
# edit   = 修改提交
```

4. cherry-pick

```
# 将其他分支的某个提交应用到当前分支
git cherry-pick <commit-id>
```

5. 查找bug（bisect）

```
git bisect start
git bisect bad HEAD      # 当前版本有bug
git bisect good v1.0     # v1.0版本正常
# Git会二分查找，逐个测试
git bisect good/bad      # 标记测试结果
git bisect reset         # 结束查找
```

三、CMake构建系统

Q7: CMake基本使用?

最简单的CMakeLists.txt：

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

set(CMAKE_CXX_STANDARD 14)

add_executable(myapp main.cpp)
```

构建过程：

```
mkdir build
cd build
cmake ..      # 生成Makefile
```

```
make            # 编译
./myapp         # 运行
```

添加库：

```
# 静态库
add_library(mylib STATIC lib.cpp)

# 动态库
add_library(mylib SHARED lib.cpp)

# 可执行文件链接库
add_executable(myapp main.cpp)
target_link_libraries(myapp mylib)
```

包含头文件：

```
# 添加头文件目录
include_directories(include)

# 或者（推荐）
target_include_directories(myapp PRIVATE include)
```

多目录项目：

```
project/
├── CMakeLists.txt
├── src/
│   ├── CMakeLists.txt
│   └── main.cpp
└── lib/
    ├── CMakeLists.txt
    └── mylib.cpp

# 根CMakeLists.txt
add_subdirectory(src)
add_subdirectory(lib)
```

与你项目的联系： "我的项目使用CMake构建，分为src/、include/、tests/三个目录，每个目录有独立的CMakeLists.txt。"

Q8: CMake常用变量和命令?

常用变量：

```
# 项目名称
${PROJECT_NAME}

# 目录
${CMAKE_SOURCE_DIR}       # 顶层CMakeLists.txt所在目录
${CMAKE_BINARY_DIR}       # build目录
${CMAKE_CURRENT_SOURCE_DIR}  # 当前CMakeLists.txt所在目录

# 编译选项
${CMAKE_CXX_STANDARD}     # C++标准(11/14/17)
${CMAKE_BUILD_TYPE}       # Debug/Release
${CMAKE_CXX_FLAGS}        # 编译选项
```

编译选项：

```
# Debug模式
set(CMAKE_BUILD_TYPE Debug)
set(CMAKE_CXX_FLAGS_DEBUG "-g -O0")

# Release模式
set(CMAKE_BUILD_TYPE Release)
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -DNDEBUG")

# 添加编译选项
add_compile_options(-Wall -Wextra)
target_compile_options(myapp PRIVATE -Wall)
```

查找依赖：

```
# 查找包
find_package(Boost REQUIRED)
include_directories(${Boost_INCLUDE_DIRS})
target_link_libraries(myapp ${Boost_LIBRARIES})

# 查找库
find_library(PTHREAD_LIB pthread)
target_link_libraries(myapp ${PTHREAD_LIB})
```

条件编译：

```
if (CMAKE_BUILD_TYPE MATCHES Debug)
    add_definitions(-DDEBUG_MODE)
endif()

if (UNIX)
```

```
    # Linux/Mac特定配置
endif()
```

四、性能分析工具

Q9: valgrind如何使用?

基本用法:

```
# 编译时加上调试信息
g++ -g -o myapp main.cpp

# 检测内存泄漏
valgrind --leak-check=full ./myapp

# 更详细的信息
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
./myapp
```

输出示例:

```
==12345== HEAP SUMMARY:
==12345==     in use at exit: 40 bytes in 1 blocks
==12345==   total heap usage: 2 allocs, 1 frees, 72,744 bytes allocated
==12345==
==12345== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12345==    at 0x4C2B0E0: operator new(unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==12345==    by 0x400724: main (main.cpp:10)
```

- `definitely lost`: 内存泄漏
- `possibly lost`: 可能泄漏
- `still reachable`: 可访问但未释放(通常不是问题)

常见内存错误:

1. **内存泄漏**: new后未delete
2. **非法访问**: 访问已释放的内存
3. **数组越界**: 访问数组边界外
4. **未初始化**: 使用未初始化的变量

与你项目的联系: "文件传输系统开发时,用valgrind发现了Buffer对象池的内存泄漏问题,通过修改为RAII模式解决。"

Q10: perf工具如何使用?

基本用法：

```
# 记录性能数据
perf record -g ./myapp

# 查看报告
perf report

# 实时监控
perf top

# 统计信息
perf stat ./myapp
```

perf stat输出：

```
Performance counter stats for './myapp':

    1000.45 msec task-clock          #    0.998 CPUs utilized
         12      context-switches     #    0.012 K/sec
          2      cpu-migrations       #    0.002 K/sec
        256      page-faults          #    0.256 K/sec
 3000000000      cycles               #    3.000 GHz
 2000000000      instructions         #    0.67  insn per cycle
  500000000      branches             #  500.000 M/sec
   10000000      branch-misses        #    2.00% of all branches
```

查找热点函数：

```
perf record -g ./myapp
perf report

# 输出示例：
# 35.00%  myapp  myapp  [.] hot_function
# 25.00%  myapp  myapp  [.] another_function
```

生成火焰图：

```
# 1. 记录数据
perf record -F 99 -g ./myapp

# 2. 生成火焰图（需要flamegraph工具）
perf script | stackcollapse-perf.pl | flamegraph.pl > flamegraph.svg
```

与你项目的联系： "用perf分析文件传输系统，发现80%时间在磁盘I/O，才想到用异步日志优化。"

Q11: gdb调试器基本使用?

启动gdb:

```
# 编译时加-g
g++ -g -o myapp main.cpp

# 启动调试
gdb ./myapp

# 或者attach到运行中的进程
gdb -p <pid>
```

常用命令:

```
# 运行
run                    # 运行程序
run arg1 arg2          # 带参数运行

# 断点
break main             # 在main函数打断点
break file.cpp:10      # 在第10行打断点
break func if x==5     # 条件断点
info breakpoints       # 查看断点
delete 1               # 删除断点1
disable 1              # 禁用断点1

# 执行
continue (c)           # 继续执行
next (n)               # 单步执行(不进入函数)
step (s)               # 单步执行(进入函数)
finish                 # 执行到函数返回
until                  # 执行到下一行

# 查看
print x                # 打印变量x
print *ptr             # 打印指针指向的内容
display x              # 每次停下来都打印x
info locals            # 查看局部变量
info args              # 查看函数参数

# 堆栈
backtrace (bt)         # 查看调用栈
frame 1                # 切换到栈帧1
up / down              # 上移/下移栈帧

# 线程
info threads           # 查看所有线程
thread 2               # 切换到线程2
```

```
# 内存
x/10x addr            # 查看内存（16进制）
x/10s addr            # 查看字符串
```

调试core dump：

```
# 开启core dump
ulimit -c unlimited

# 程序崩溃后
gdb ./myapp core

# 查看崩溃位置
backtrace
```

与你项目的联系： "并发bug很难复现，用gdb attach到运行中的进程，打条件断点，最终定位到竞态条件。"

五、Shell脚本

Q12: 常用的Shell脚本技巧?

基本语法：

```
#!/bin/bash

# 变量
name="Alice"
echo "Hello, $name"

# 命令替换
now=$(date +%Y-%m-%d)
files=$(ls *.cpp)

# 条件判断
if [ $? -eq 0 ]; then
    echo "Success"
else
    echo "Failed"
fi

# 循环
for file in *.cpp; do
    echo "Processing $file"
    gcc -c $file
done

# 函数
```

```
function build() {
    mkdir -p build
    cd build
    cmake ..
    make
}

build
```

文件测试：

```
if [ -f file.txt ]; then    # 文件存在
if [ -d dir ]; then         # 目录存在
if [ -x file.sh ]; then     # 文件可执行
if [ -z "$var" ]; then      # 字符串为空
if [ "$a" -eq "$b" ]; then  # 数字相等
if [ "$a" == "$b" ]; then   # 字符串相等
```

实用脚本示例：

```
# 批量重命名
for file in *.txt; do
    mv "$file" "${file%.txt}.bak"
done

# 查找并删除大文件
find /path -type f -size +100M -exec rm {} \;

# 批量编译
for dir in $(find . -name "src" -type d); do
    cd $dir
    make
    cd -
done

# 监控进程
while true; do
    if ! pgrep myapp > /dev/null; then
        echo "Process died, restarting..."
        ./myapp &
    fi
    sleep 10
done
```

六、快速复习清单

Linux命令

- [ ] 文件操作（find/grep）
- [ ] 进程管理（ps/top/kill）
- [ ] 系统监控（free/iostat/netstat）
- [ ] 查看端口占用（lsof/netstat）

Git

- [ ] 基本流程（add/commit/push/pull）
- [ ] 分支操作（branch/merge）
- [ ] 冲突解决
- [ ] 高级操作（stash/rebase/cherry-pick）

CMake

- [ ] 基本语法（add_executable/add_library）
- [ ] 常用变量（CMAKE_SOURCE_DIR）
- [ ] 编译选项（CMAKE_CXX_FLAGS）
- [ ] 查找依赖（find_package）

性能分析

- [ ] valgrind检测内存泄漏
- [ ] perf性能分析
- [ ] gdb调试

Shell脚本

- [ ] 基本语法（变量/循环/条件）
- [ ] 常用技巧

面试技巧

结合项目回答

示例： "在文件传输系统开发中：

- 用`valgrind`发现内存泄漏
- 用`perf`定位性能瓶颈
- 用`gdb`调试并发bug
- 用`CMake`管理多目录项目
- 用`Git`进行版本控制"

展示工具熟练度

不要只说"会用"，要说：

- 什么场景用
- 解决了什么问题
- 用了哪些高级功能

示例： "valgrind不仅能检测内存泄漏，还能：

- 检测非法内存访问
- 检测未初始化变量
- 分析缓存命中率（cachegrind）
- 分析调用次数（callgrind）"

常见追问

Q: "valgrind检测出内存泄漏后，你怎么定位具体代码？"

A: "valgrind会显示调用栈，指出在哪个文件哪一行分配的内存未释放。比如：

```
40 bytes in 1 blocks are definitely lost
at 0x4C2B0E0: operator new (vgpreload_memcheck)
by 0x400724: main (main.cpp:10)
```

说明在main.cpp第10行new的内存没有delete。然后用gdb在那一行打断点，追踪对象生命周期，找到应该delete的位置。"

记住：工具 + 场景 + 问题解决 = 高分回答