面试基本信息

基础信息

- 面试岗位：运维工程师(k8s方向)、运维开发工程师、SRE工程师
- 工作经验：5年
- 薪资范围：年薪40万左右，上海
- 面试时间：2024年12月
- 面试公司:中大型公司

面试题汇总

> 注：☆表示多次出现过的高频面试题

Linux

Linux系统基础

- grep sed awk cut组合使用 ☆
- 常用的文本处理命令有哪些（sort、uniq、tr等）
- shell脚本中的特殊变量含义（$?、$#、$@等）☆
- http错误码和原因
- 长连接、短连接、WebSocket区别和使用场景

网络服务

- nginx性能优化有哪些方式 ☆
- lvs、nginx、haproxy区别和使用场景 ☆
- 什么是nginx的异步非阻塞
- nginx的worker_processes和worker_connections如何配置 ☆
- 如何实现nginx的动态upstream
- keepalived的工作原理和配置方式
- DNS解析过程详解
- TCP三次握手和四次挥手详解 ☆

进程管理

- 僵尸进程是什么
- 进程、线程、协程区别 ☆
- 什么是进程中断
- 什么是软中断、硬中断
- 什么是不可中断进程
- 进程调度算法有哪些
- 如何查看系统的运行级别
- systemd和init的区别 ☆

系统性能

- linux网络丢包怎么排查 ☆
- 常用的性能分析诊断命令 ☆
- 如何排查CPU使用率过高的问题 ☆
- 如何排查内存泄漏问题
- 什么是栈内存和堆内存
- top命令里面可以看到进程哪些状态 ☆
- Linux系统中/proc是做什么的
- load和cpu使用率区别
- 如何使用stress进行压力测试

网络基础

- MAC地址IP地址如何转换
- TCP/IP协议栈各层的作用 ☆
- 什么是TCP粘包，如何解决
- 如何使用tcpdump抓包分析 ☆
- iptables的四表五链详解
- 网络路由协议有哪些（OSPF、BGP等）

存储管理

- 常见的raid有哪些，使用场景是什么
- lvm怎么划分
- 文件系统格式有哪些（ext4、xfs等）☆
- 磁盘IO调度算法有哪些
- 如何进行磁盘分区和挂载
- 软链接和硬链接的区别 ☆

系统优化

- jvm内存如何查看
- 如何管理和优化内核参数
- 什么是进程最大数、最大线程数、进程打开的文件数，怎么调整 ☆
- du和df统计不一致原因 ☆
- buffers与cached的区别 ☆
- 如何优化系统启动时间
- 如何处理OOM问题 ☆

进程通信与管理

- lsof命令使用场景
- Linux中的进程间通信的方式及其使用场景
- Linux中的进程优先级与设置方法
- nice和renice的使用方法
- 如何查看系统调用（strace的使用）☆

内存管理

- 什么是内存分页和分段
- 什么是swap分区，何时使用
- 如何查看内存使用情况详情
- 什么是内存碎片，如何处理 ☆

系统服务

- 如何创建和管理自定义systemd服务
- Linux内核模块的加载与卸载过程
- 如何配置开机自启动服务
- journalctl日志管理使用方法 ☆

自动化运维

- ansible roles使用场景，现在有多台机器需要批量加入k8s集群，怎么实现 ☆
- 如何使用expect实现自动化交互
- 如何使用puppet/salt实现配置管理
- 常用的运维监控工具有哪些 ☆

安全管理

- SELinux的作用和配置方法
- sudo权限管理配置详解
- 如何防范常见的Linux安全攻击 ☆
- 如何进行系统安全加固

linux-答案

Linux系统基础

1. grep sed awk cut组合使用 ☆

各命令基本用法

1. **grep**: 文本搜索工具

```
grep [选项] 模式 文件名
常用选项:
-i: 忽略大小写
-v: 反向选择
-r: 递归搜索
-n: 显示行号
-E: 扩展正则表达式
```

2. **sed**: 流编辑器

```
sed [选项] '命令' 文件名
常用命令:
s/pattern/replacement/: 替换
d: 删除
i: 插入
a: 追加
常用选项:
-i: 直接修改文件
-n: 安静模式
```

3. **awk**: 文本处理工具

```
awk [选项] 'pattern {action}' 文件名
内置变量:
$0: 整行内容
$1,$2...: 第n个字段
NR: 行号
NF: 字段数
FS: 字段分隔符
```

4. **cut**: 列提取工具

```
cut [选项] 文件名
常用选项:
-d: 指定分隔符
-f: 指定字段
-c: 按字符位置提取
```


组合使用示例

1. 提取nginx访问日志中状态码为404的IP地址和访问路径:

```
cat access.log | grep "404" | awk '{print $1,$7}'
```

2. 统计某个文件中每个IP地址的访问次数，并按次数排序:

```
cat access.log | awk '{print $1}' | sort | uniq -c | sort -nr
```

3. 替换配置文件中的特定内容:

```
sed -i 's/old_version/new_version/g' config.conf
```

2. 常用的文本处理命令

sort

- 功能：排序
- 常用选项：
  - `-n`: 按数字排序
  - `-r`: 逆序
  - `-k`: 指定列
  - `-t`: 指定分隔符

```
sort -t: -k3,3n /etc/passwd  # 按UID排序
```

uniq

- 功能：去重
- 常用选项：
  - `-c`: 显示重复次数
  - `-d`: 只显示重复行
  - `-u`: 只显示唯一行

```
sort file | uniq -c  # 统计重复行次数
```

tr

- 功能：字符转换或删除
- 常用用法：
  - 大小写转换
  - 字符替换
  - 删除特定字符

```
echo "HELLO" | tr 'A-Z' 'a-z'  # 转小写
cat file | tr -d '\r'  # 删除回车符
```

3. shell脚本特殊变量含义 ☆

位置参数

- `$0`: 脚本名称
- `$1` 到 `$9`: 第1到第9个参数
- `${10}`: 第10个参数（需要大括号）
- `$#`: 参数个数
- `$*`: 所有参数（作为一个整体）
- `$@`: 所有参数（分别对待）

状态变量

- $?: 上一个命令的退出状态（0表示成功）
- $$: 当前shell的进程ID
- $!: 最后一个后台进程的进程ID
- $-: 当前shell的选项标记

使用示例

```
#!/bin/bash
echo "脚本名称: $0"
echo "第一个参数: $1"
echo "参数个数: $#"
echo "所有参数: $@"
echo "上一命令退出状态: $?"
```

4. HTTP错误码和原因

1xx（信息性状态码）

- 100 Continue: 继续请求
- 101 Switching Protocols: 协议切换

2xx（成功状态码）

- 200 OK: 请求成功
- 201 Created: 已创建
- 204 No Content: 无内容返回

3xx（重定向状态码）

- 301 Moved Permanently: 永久重定向
- 302 Found: 临时重定向
- 304 Not Modified: 未修改

4xx（客户端错误）

- 400 Bad Request: 请求语法错误
- 401 Unauthorized: 未授权
- 403 Forbidden: 禁止访问
- 404 Not Found: 资源不存在
- 405 Method Not Allowed: 方法不允许

5xx（服务器错误）

- 500 Internal Server Error: 服务器内部错误
- 502 Bad Gateway: 网关错误
- 503 Service Unavailable: 服务不可用

- 504 Gateway Timeout: 网关超时

5. 长连接、短连接、WebSocket区别和使用场景

短连接

- 特点:
  - 每次请求都建立新的TCP连接
  - 请求完成后立即断开
- 优点:
  - 服务器资源占用少
  - 适合并发量大但通信频率低的场景
- 缺点:
  - 频繁建立连接开销大
- 使用场景:
  - 简单的HTTP请求
  - 静态资源加载

长连接（Keep-Alive）

- 特点:
  - 复用TCP连接
  - 设置超时时间
- 优点:
  - 减少连接建立开销
  - 提高响应速度
- 缺点:
  - 服务器需要保持连接状态
- 使用场景:
  - 频繁请求的Web应用
  - API调用

WebSocket

- 特点:
  - 全双工通信
  - 服务器可主动推送
  - 基于TCP的持久连接
- 优点:
  - 实时性好
  - 数据量小
  - 支持双向通信
- 缺点:
  - 部分老旧浏览器不支持
  - 连接保持成本高
- 使用场景:
  - 实时聊天
  - 在线游戏

- 实时数据监控
- 协同编辑

```
# 网络服务

## 1. nginx性能优化有哪些方式 ☆

### 系统层面优化
1. **系统参数调整**
```bash
# /etc/sysctl.conf
net.ipv4.tcp_max_tw_buckets = 6000
net.ipv4.ip_local_port_range = 1024 65000
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_syncookies = 1
```

2. **打开文件数优化**

```
# /etc/security/limits.conf
* soft nofile 65535
* hard nofile 65535
```

Nginx配置优化

1. **worker进程优化**

```
worker_processes auto;  # CPU核心数
worker_cpu_affinity auto;  # CPU亲和性
worker_rlimit_nofile 65535;  # 最大文件打开数
```

2. **事件处理优化**

```
events {
    use epoll;  # 使用epoll事件模型
    worker_connections 65535;  # 单个worker最大连接数
    multi_accept on;  # 尽可能多接受请求
}
```

3. **HTTP优化**
```

```
http {
    keepalive_timeout 65;  # 长连接超时时间
    client_header_buffer_size 32k;  # 请求头缓冲区
    large_client_header_buffers 4 32k;  # 大请求头缓冲区
    client_max_body_size 8m;  # 最大请求体大小

    # Gzip压缩
    gzip on;
    gzip_min_length 1k;
    gzip_buffers 4 16k;
    gzip_types text/plain text/css application/json;
}
```

4. **缓存优化**

```
# 开启缓存
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m max_size=10g
inactive=60m use_temp_path=off;

location / {
    proxy_cache my_cache;
    proxy_cache_use_stale error timeout http_500 http_502 http_503 http_504;
    proxy_cache_valid 200 304 12h;
}
```


2. lvs、nginx、haproxy区别和使用场景 ☆

LVS（Linux Virtual Server）

1. **特点**

- 工作在网络层（第4层）
- 性能最好，可支持百万并发
- 工作模式：NAT、DR、TUN
- 仅做转发，不处理具体数据

2. **使用场景**

- 需要超高并发的场景
- 需要四层负载均衡
- 大型网站的入口负载均衡

3. **优缺点**

- 优点：性能强大，稳定性好
- 缺点：配置相对复杂，功能单一

Nginx

1. **特点**

- 工作在应用层（第7层）
- 支持HTTP、HTTPS、SMTP等协议
- 具备反向代理、缓存等功能
- 配置灵活，功能丰富

2. **使用场景**

- Web应用负载均衡
- 静态资源服务器
- 反向代理服务器
- 小型到中型站点的主要入口

3. **优缺点**

- 优点：功能丰富，配置简单，运维方便
- 缺点：并发能力相对LVS差

HAProxy

1. **特点**

- 同时支持四层和七层负载均衡
- 支持会话保持
- 详细的统计信息
- 支持健康检查

2. **使用场景**

- 需要同时使用四层和七层负载均衡
- 对统计监控要求较高
- 对会话保持要求较高的场景

3. **优缺点**

- 优点：功能全面，性能优秀，监控强大
- 缺点：不支持缓存，规则配置相对复杂

3. nginx的异步非阻塞

工作原理

1. **事件驱动模型**

- 使用epoll事件模型
- 单个worker处理多个连接
- 非阻塞I/O处理请求

2. **进程模型**

```
master进程
  ├── worker进程1
  ├── worker进程2
  └── worker进程n
```

3. **处理流程**

- worker进程通过epoll监听事件
- 收到请求后立即处理
- 异步处理I/O操作
- 无需等待，继续处理其他请求

优势

1. **高并发处理能力**

- 单个worker可处理多个请求
- 充分利用多核CPU
- 减少进程/线程切换开销

2. **资源利用率高**

- 避免阻塞等待
- 减少内存占用
- 提高系统吞吐量

配置示例

```
events {
    use epoll;  # 使用epoll事件模型
    worker_connections 1024;  # 单个worker最大连接数
}

http {
    # 异步文件I/O
    aio on;
    directio 512;

    # 异步upstream
    upstream backend {
        server backend1.example.com max_fails=3 fail_timeout=30s;
        server backend2.example.com max_fails=3 fail_timeout=30s;
        keepalive 32;  # 保持连接数
    }
}
```

4. nginx的worker_processes和worker_connections配置

worker_processes

1. **含义**

- 工作进程数量
- 通常设置为CPU核心数
- 可以手动设置或自动检测

2. **配置方法**

```
# 自动设置
worker_processes auto;

# 手动设置
worker_processes 4;  # 具体数值根据CPU核心数设置
```

3. **最佳实践**

- 一般设置为CPU核心数
- 单核CPU设置为1
- IO密集型可以设置为CPU核心数*2

worker_connections

1. **含义**

- 单个worker进程的最大连接数
- 包括所有连接（客户端和上游服务器）
- 实际最大并发数 = worker_processes * worker_connections

2. **配置方法**

```
events {
    worker_connections 1024;
}
```

3. **计算公式**

- HTTP连接：max_clients = worker_processes * worker_connections
- HTTPS连接：max_clients = worker_processes * worker_connections/2
- 反向代理：max_clients = worker_processes * worker_connections/4

4. **影响因素**

- 系统可用文件描述符数量
- 内存大小
- 网络带宽

5. 如何实现nginx的动态upstream

实现方式

1. **nginx-upsync-module方案**

```
http {
    upstream backend {
        # 使用upsync模块从consul同步后端服务器列表
        upsync 127.0.0.1:8500/v1/kv/upstreams/backend/ upsync_timeout=6m
upsync_interval=500ms upsync_type=consul strong_dependency=off;
        upsync_dump_path /tmp/servers_backend.conf;

        # 默认服务器配置
        server 127.0.0.1:8080 backup;
    }
}
```

2. **nginx-plus商业版**

```
http {
    upstream backend {
        zone backend 64k;
        server backend1.example.com;
        server backend2.example.com;
    }
}

# 通过API动态管理
location /api {
    api write=on;
    allow 127.0.0.1;
    deny all;
}
```

3. **OpenResty + Lua实现**

```
http {
    lua_shared_dict upstream_list 10m;

    init_by_lua_block {
        local upstream_list = ngx.shared.upstream_list
        upstream_list:set("server1", "192.168.1.10:8080")
        upstream_list:set("server2", "192.168.1.11:8080")
    }

    upstream backend {
        balancer_by_lua_block {
```

```
            -- 动态选择后端服务器
            local balancer = require "ngx.balancer"
            local host = ngx.shared.upstream_list:get("server1")
            local port = 8080
            balancer.set_current_peer(host, port)
        }
    }
}
```

应用场景

- 服务器动态扩缩容
- 服务器故障自动摘除
- 灰度发布
- 动态调整权重

6. keepalived的工作原理和配置方式

工作原理

1. **VRRP协议**

- 虚拟路由冗余协议
- Master/Backup机制
- 通过优先级选举Master
- 共享虚拟IP(VIP)

2. **状态转换**

```
初始化(INIT) -> 备份(BACKUP) -> 主机(MASTER)
                    ^              |
                    |______________|
```

3. **健康检查**

- 服务级别检查
- 系统级别检查
- 自定义脚本检查

配置示例

1. **主节点配置**

```
vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 51
    priority 100
```

```
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.200.16/24
    }
}

# 服务检查
vrrp_script chk_nginx {
    script "pidof nginx"
    interval 2
    weight -20
}
```

2. **备节点配置**

```
vrrp_instance VI_1 {
    state BACKUP
    interface eth0
    virtual_router_id 51
    priority 90
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.200.16/24
    }
}
```


常见配置参数

- `state`: 初始状态
- `priority`: 优先级(0-255)
- `advert_int`: VRRP包发送间隔
- `nopreempt`: 非抢占模式
- `track_interface`: 监控网卡状态

7. DNS解析过程详解

解析流程

1. **本地解析**

```
1. 检查浏览器缓存
2. 检查操作系统缓存
```

```
3. 检查本地hosts文件
```

2. **递归查询**

```
本地DNS服务器 -> 根域名服务器 -> 顶级域名服务器 -> 权威域名服务器
```

3. **查询类型**

- 递归查询：客户端只发出一次请求
- 迭代查询：DNS服务器逐级查询

DNS记录类型

- A记录：域名到IPv4地址
- AAAA记录：域名到IPv6地址
- CNAME记录：域名别名
- MX记录：邮件服务器
- NS记录：域名服务器
- PTR记录：IP地址到域名
- TXT记录：文本信息

常用命令

```
# 查询DNS记录
dig example.com

# 指定记录类型
dig example.com MX

# 跟踪解析过程
dig +trace example.com

# 使用nslookup
nslookup example.com
```


8. TCP三次握手和四次挥手详解 ☆

三次握手过程

1. **建立连接流程**

```
客户端 -------- SYN=1,seq=x --------> 服务端  (第一次握手)
客户端 <--- SYN=1,ACK=1,seq=y,ack=x+1 --- 服务端  (第二次握手)
客户端 -------- ACK=1,seq=x+1,ack=y+1 --------> 服务端  (第三次握手)
```

2. **状态变化**

```
客户端：CLOSED -> SYN_SENT -> ESTABLISHED
服务端：CLOSED -> LISTEN -> SYN_RCVD -> ESTABLISHED
```

四次挥手过程

1. **断开连接流程**

```
客户端 -------- FIN=1,seq=u --------> 服务端 （第一次挥手）
客户端 <-------- ACK=1,ack=u+1 -------- 服务端 （第二次挥手）
客户端 <-------- FIN=1,seq=v -------- 服务端 （第三次挥手）
客户端 -------- ACK=1,ack=v+1 --------> 服务端 （第四次挥手）
```

2. **状态变化**

```
客户端：ESTABLISHED -> FIN_WAIT_1 -> FIN_WAIT_2 -> TIME_WAIT -> CLOSED
服务端：ESTABLISHED -> CLOSE_WAIT -> LAST_ACK -> CLOSED
```

重要概念

1. **TIME_WAIT状态**

- 持续时间：2MSL
- 作用：
  - 确保最后一个ACK能到达
  - 防止旧连接数据干扰新连接

2. **常见问题**

- SYN攻击：大量半连接
- TIME_WAIT过多：端口资源耗尽
- 连接建立失败：防火墙/网络问题

3. **优化配置**

```
# 调整内核参数
net.ipv4.tcp_syncookies = 1  # 防止SYN攻击
net.ipv4.tcp_tw_reuse = 1    # 允许TIME_WAIT重用
net.ipv4.tcp_max_tw_buckets = 5000  # TIME_WAIT最大数量
```

进程管理

1. 僵尸进程

定义

- 僵尸进程是已经终止但其父进程尚未回收其资源（如进程描述符）的进程
- 在进程表中状态显示为"Z"或"defunct"

产生原因

1. **父进程没有调用wait()/waitpid()**

- 子进程退出时，父进程未及时处理子进程退出信号
- 父进程没有调用wait相关函数回收子进程资源

2. **实际场景**

```
# 查看僵尸进程
ps aux | grep 'Z'
```

危害

- 占用进程号(PID)
- 占用系统进程表项
- 大量僵尸进程可能导致系统无法创建新进程

解决方法

1. **编程预防**

```
// 注册SIGCHLD信号处理
signal(SIGCHLD, SIG_IGN);
// 或
signal(SIGCHLD, handle_child);

void handle_child(int sig) {
    while(waitpid(-1, NULL, WNOHANG) > 0);
}
```

2. **已有僵尸进程处理**

- 找到并杀死父进程
- 僵尸进程会被init进程接管并回收

2. 进程、线程、协程区别 ☆

进程(Process)

1. **特点**

- 系统资源分配的基本单位
- 拥有独立的内存空间
- 拥有独立的文件描述符表
- 上下文切换开销大

2. **优缺点**

- 优点：独立性强，安全性高
- 缺点：资源开销大，切换成本高

线程(Thread)

1. **特点**

- CPU调度的基本单位
- 共享所属进程的内存空间
- 拥有独立的线程栈
- 切换开销比进程小

2. **优缺点**

- 优点：资源开销小，切换快
- 缺点：共享内存可能导致同步问题

协程(Coroutine)

1. **特点**

- 用户态的轻量级线程
- 由应用程序自己调度
- 共享所属线程的资源
- 切换开销最小

2. **优缺点**

- 优点：开销极小，切换成本低
- 缺点：不能利用多核，调试相对困难

对比表格

特性	进程	线程	协程
调度方式	系统调度	系统调度	用户调度
切换成本	高	中	低
资源占用	高	中	低
系统支持	完善	完善	部分支持
通信方式	IPC	共享内存	共享内存

3. 软中断、硬中断

硬中断(Hardware Interrupt)

1. **定义**

- 由硬件设备触发
- 优先级高
- 直接中断CPU

2. **特点**

- 异步执行
- 不可屏蔽中断(NMI)和可屏蔽中断(IRQ)
- 处理时间要求短

3. **常见来源**

- 键盘输入
- 网卡收包
- 硬盘IO完成
- 时钟中断

软中断(Software Interrupt)

1. **定义**

- 由软件触发
- 优先级低于硬中断
- 可以被硬中断打断

2. **特点**

- 同步执行
- 可以被延迟处理
- 在ksoftirqd内核线程中处理

3. **常见类型**

```
# 查看软中断统计
cat /proc/softirqs

# 主要类型
- NET_TX/NET_RX: 网络发送/接收
- TIMER: 定时器
- TASKLET: 小任务
- SCHED: 调度
```

区别对比

特性	硬中断	软中断

特性	硬中断	软中断
触发源	硬件	软件
优先级	高	低
执行时间	短	可较长
可延迟	否	是
处理函数	interrupt handler	softirq handler

4. 不可中断进程

定义

- 进程状态为D(TASK_UNINTERRUPTIBLE)
- 不响应异步信号，包括SIGKILL
- 通常在等待IO操作完成

特点

1. **产生原因**

- 系统调用等待IO
- 直接操作硬件
- 等待内核锁

2. **查看方法**

```
# 查看D状态进程
ps aux | grep D

# 使用top查看
top -b -n 1 | grep D
```


常见场景

1. **磁盘IO**

- 等待磁盘读写完成
- NFS挂载无响应

2. **设备操作**

- USB设备操作
- 打印机操作

3. **内核锁**

- 等待内核互斥锁

- 等待自旋锁

处理方法

1. **预防措施**

- 使用异步IO
- 避免长时间的IO等待
- 合理设置超时时间

2. **问题排查**

```
# 使用strace跟踪
strace -p PID

# 查看系统IO状态
iostat -x 1

# 查看进程堆栈
cat /proc/PID/stack
```

5. 进程调度算法

Linux进程调度器

1. **CFS（Completely Fair Scheduler）**

- 完全公平调度器
- 默认调度算法
- 基于红黑树实现
- 使用虚拟运行时间进行调度

2. **实时调度算法**

- SCHED_FIFO：先进先出
- SCHED_RR：时间片轮转
- 优先级范围：0-99

调度策略

```
# 查看进程调度策略
chrt -p PID

# 设置调度策略
chrt -f -p [priority] PID  # SCHED_FIFO
chrt -r -p [priority] PID  # SCHED_RR
```

进程优先级

- nice值：-20到19，默认0
- 实时优先级：0到99

```
# 调整nice值
nice -n 10 command
renice -n 10 -p PID
```

6. 系统运行级别

传统init运行级别

- 0：关机
- 1：单用户模式
- 2：多用户模式（无网络）
- 3：多用户模式（有网络）
- 4：用户自定义
- 5：图形界面
- 6：重启

systemd目标（target）

```
# init运行级别对应的systemd目标
runlevel0.target -> poweroff.target
runlevel1.target -> rescue.target
runlevel2.target -> multi-user.target
runlevel3.target -> multi-user.target
runlevel4.target -> multi-user.target
runlevel5.target -> graphical.target
runlevel6.target -> reboot.target
```


查看和切换

```
# 查看当前运行级别
runlevel
systemctl get-default

# 切换运行级别
systemctl isolate multi-user.target
systemctl isolate graphical.target

# 设置默认运行级别
systemctl set-default multi-user.target
```

7. systemd和init的区别 ☆

init系统

1. **特点**

- 系统第一个进程（PID 1）
- 串行启动服务
- 使用运行级别
- 基于Shell脚本

2. **缺点**

- 启动慢
- 依赖关系复杂
- 无并行启动能力
- 服务管理不便

systemd系统

1. **特点**

- 替代init的现代初始化系统
- 并行启动服务
- 按需启动
- 自动处理依赖关系
- 统一的服务管理方式

2. **主要功能**

```
# 服务管理
systemctl start/stop/restart/status service

# 查看系统状态
systemctl status

# 查看启动耗时
systemd-analyze blame

# 服务开机启动
systemctl enable/disable service
```

3. **单元类型**

- .service：服务
- .socket：套接字
- .target：目标
- .mount：挂载点
- .timer：定时器

对比表格

特性	init	systemd
启动方式	串行	并行
配置文件	Shell脚本	单元文件
依赖处理	手动管理	自动处理
服务管理	service命令	systemctl命令
日志管理	syslog	journald
资源控制	无	cgroup集成

系统性能

linux网络丢包怎么排查

1. 使用netstat统计信息查看

```
netstat -s | grep -i "packet"
netstat -i   # 查看网卡丢包情况
```

2. 使用tcpdump抓包分析

```
tcpdump -i eth0 -n port 80
```

3. 查看系统日志

```
dmesg | grep -i "drop"
```

4. 使用sar命令监控网络接口

```
sar -n DEV 1
```

常用的性能分析诊断命令

1. top/htop - 实时监控系统进程、CPU、内存使用情况
2. vmstat - 监控系统内存、进程、CPU等信息
3. iostat - 监控系统IO和CPU使用情况
4. netstat/ss - 监控网络连接情况
5. sar - 收集、报告系统活动信息
6. dstat - 系统资源统计工具
7. perf - 性能分析工具

8. strace - 跟踪系统调用和信号

如何排查CPU使用率过高的问题

1. 使用top命令找出CPU使用率最高的进程
2. 通过ps命令查看具体进程信息

```
ps -ef | grep <pid>
```

3. 使用perf分析进程内部热点

```
perf top -p <pid>
```

4. 使用strace查看系统调用

```
strace -p <pid>
```

如何排查内存泄漏问题

1. 使用top/free命令监控内存使用趋势
2. 使用pmap查看进程内存映射

```
pmap -x <pid>
```

3. 使用valgrind工具检测内存泄漏
4. 查看系统日志中的OOM记录
5. 使用gdb调试程序

什么是栈内存和堆内存

- 栈内存(Stack)：
  - 由系统自动分配和释放
  - 存储局部变量、函数参数等
  - 空间较小，但访问速度快
  - 先进后出(FILO)的数据结构
- 堆内存(Heap)：
  - 由程序员手动申请和释放
  - 存储动态分配的数据
  - 空间较大，但访问速度相对较慢
  - 容易产生内存碎片

top命令里面可以看到进程哪些状态

- R (Running): 正在运行或在运行队列中等待
- S (Sleep): 可中断睡眠状态
- D (Disk Sleep): 不可中断睡眠状态
- Z (Zombie): 僵尸进程
- T (Stopped): 停止状态
- I (Idle): 空闲状态

Linux系统中/proc是做什么的

- /proc是一个虚拟文件系统
- 提供内核运行状态的接口
- 包含系统硬件和进程信息
- 重要文件：
  - /proc/cpuinfo: CPU信息
  - /proc/meminfo: 内存信息
  - /proc/loadavg: 系统负载
  - /proc/: 进程相关信息

load和cpu使用率区别

- CPU使用率：
  - 表示CPU执行非空闲进程的时间百分比
  - 反映CPU的繁忙程度
- 系统负载(Load)：
  - 表示系统中正在运行和等待运行的进程数
  - 包含CPU、IO等待的进程
  - Load Average通常显示1分钟、5分钟、15分钟的平均值

如何使用stress进行压力测试

1. 安装stress工具

```
apt-get install stress  # Debian/Ubuntu
yum install stress     # CentOS/RHEL
```

2. 常用压测命令

```
# CPU压力测试
stress --cpu 8 --timeout 60s

# 内存压力测试
stress --vm 2 --vm-bytes 1G --timeout 60s
```

```
# IO压力测试
stress --io 4 --timeout 60s

# 综合压力测试
stress --cpu 4 --io 3 --vm 2 --vm-bytes 128M --timeout 60s
```

网络基础

MAC地址和IP地址如何转换

- ARP (Address Resolution Protocol) 协议负责将IP地址转换为MAC地址
- 工作流程：
    1. 主机发送ARP广播请求，询问目标IP对应的MAC地址
    2. 拥有该IP的设备回复其MAC地址
    3. 发送主机将IP-MAC对应关系存入ARP缓存表
- 可以通过`arp -a`命令查看ARP缓存表

TCP/IP协议栈各层的作用

- 应用层：为应用程序提供服务，如HTTP、FTP、DNS等
- 传输层：提供端到端的通信，主要协议有TCP和UDP
    - TCP：面向连接、可靠传输、流量控制
    - UDP：无连接、不可靠传输、效率高
- 网络层：负责数据包的路由和转发，主要是IP协议
- 数据链路层：负责相邻节点之间的数据传输，处理MAC地址
- 物理层：定义物理传输媒介，如网线、光纤等的规范

TCP粘包问题及解决方案

- 粘包原因：
    1. TCP是面向流的协议，数据传输无边界
    2. 发送方Nagle算法的优化
    3. 接收方读取不及时
- 解决方案：
    1. 固定长度：每个数据包大小固定
    2. 分隔符：使用特殊字符分隔数据包
    3. 消息长度：在数据包头部添加长度字段
    4. 自定义协议：实现应用层协议来处理

tcpdump抓包分析

- 基本语法：`tcpdump [选项] [过滤表达式]`
- 常用选项：
    - `-i`：指定网络接口
    - `-n`：不解析主机名和端口
    - `-w`：将数据写入文件

    - `-r`：读取抓包文件
- 常用过滤表达式：
    - `host`：指定主机
    - `port`：指定端口
    - `tcp/udp`：指定协议
- 示例：

```
tcpdump -i eth0 port 80    # 抓取HTTP流量
tcpdump -i any tcp port 22 # 抓取SSH流量
```

iptables四表五链

- 四表（优先级从高到低）：
    1. raw表：连接跟踪机制
    2. mangle表：数据包修改
    3. nat表：网络地址转换
    4. filter表：包过滤
- 五链：
    1. PREROUTING：数据包进入路由表前
    2. INPUT：发往本机的数据包
    3. FORWARD：转发数据包
    4. OUTPUT：本机发出的数据包
    5. POSTROUTING：数据包离开路由表后

常见网络路由协议

- 内部网关协议(IGP)：
    - OSPF：开放最短路径优先
        - 链路状态协议
        - 适用于大型网络
        - 收敛速度快
    - RIP：路由信息协议
        - 距离矢量协议
        - 适用于小型网络
        - 有跳数限制
- 外部网关协议(EGP)：
    - BGP：边界网关协议
        - 路径矢量协议
        - 用于互联网核心路由
        - 可靠性高，复杂度大

存储管理

RAID类型及使用场景

- RAID 0（条带化）

    - 特点：数据分散存储，无冗余
    - 优点：读写性能最好
    - 缺点：无容错能力
    - 场景：追求性能，数据安全性要求低
- RAID 1（镜像）
    - 特点：数据完全复制
    - 优点：可靠性高，读性能好
    - 缺点：磁盘利用率50%
    - 场景：重要数据存储
- RAID 5
    - 特点：分布式奇偶校验
    - 优点：兼顾性能和可靠性
    - 缺点：写性能较差
    - 场景：常用企业存储方案
- RAID 10
    - 特点：RAID 1+0组合
    - 优点：高性能高可靠
    - 缺点：成本高
    - 场景：关键业务系统

LVM逻辑卷管理

- 基本概念：
    1. PV (Physical Volume)：物理卷
    2. VG (Volume Group)：卷组
    3. LV (Logical Volume)：逻辑卷
- 创建步骤：

```
pvcreate /dev/sdb      # 创建PV
vgcreate vg0 /dev/sdb  # 创建VG
lvcreate -L 10G -n lv0 vg0 # 创建LV
```

- 优势：
    - 动态调整卷大小
    - 在线迁移数据
    - 创建快照

文件系统格式比较

- ext4
    - 最大文件系统大小：1EB
    - 最大单文件大小：16TB
    - 特点：稳定性好，兼容性强
    - 适用：通用场景
- xfs
    - 最大文件系统大小：8EB

    - 最大单文件大小：8EB
    - 特点：高性能，适合大文件
    - 适用：大规模存储系统
- btrfs
    - 特点：支持快照，自带RAID
    - 适用：需要高级特性的场景
- zfs
    - 特点：数据完整性强，功能丰富
    - 适用：企业级存储

磁盘IO调度算法

- CFQ (Completely Fair Queuing)
    - 完全公平队列
    - 适用于桌面系统
- Deadline
    - 设置截止时间
    - 适用于数据库服务器
- NOOP
    - 简单FIFO队列
    - 适用于SSD
- BFQ (Budget Fair Queueing)
    - 基于预算的公平队列
    - 适用于交互性应用

磁盘分区和挂载

- 分区步骤：

```
fdisk /dev/sdb     # 交互式分区
mkfs.ext4 /dev/sdb1 # 格式化分区
```

- 挂载命令：

```
mount /dev/sdb1 /mnt/data  # 临时挂载
```

- 永久挂载：

```
# 在/etc/fstab中添加
/dev/sdb1 /mnt/data ext4 defaults 0 0
```

软链接vs硬链接

- 软链接（符号链接）
  - 类似Windows快捷方式
  - 可以跨文件系统
  - 可以指向目录
  - 源文件删除后链接失效
  - 创建：`ln -s 源文件 链接文件`
- 硬链接
  - 与源文件共享inode
  - 不能跨文件系统
  - 不能链接目录
  - 源文件删除后仍可访问
  - 创建：`ln 源文件 链接文件`

系统优化

JVM内存查看

- 命令行工具：

```
jps          # 查看Java进程
jstat -gc PID  # 查看GC情况
jmap -heap PID # 查看堆内存使用
jstack PID     # 查看线程栈信息
```

- 可视化工具：
  - JConsole：JDK自带监控工具
  - VisualVM：功能强大的性能分析工具
  - JProfiler：商业级性能分析工具

内核参数管理和优化

- 查看和修改方式：

```
sysctl -a                        # 查看所有参数
sysctl -w 参数=值                # 临时修改
echo "参数=值" >> /etc/sysctl.conf  # 永久修改
```

- 常用优化参数：

```
# 网络优化
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn_backlog = 8192

# 内存优化
vm.swappiness = 10
```

```
vm.dirty_ratio = 40

# 文件系统
fs.file-max = 655350
```

进程、线程和文件数限制

- 进程最大数（max processes）
  - 查看：`ulimit -u`
  - 修改：`/etc/security/limits.conf`

```
* soft nproc 65535
* hard nproc 65535
```

- 最大线程数（max threads）
  - 查看：`cat /proc/sys/kernel/threads-max`
  - 修改：`sysctl -w kernel.threads-max=100000`
- 打开文件数（open files）
  - 查看：`ulimit -n`
  - 修改：`/etc/security/limits.conf`

```
* soft nofile 65535
* hard nofile 65535
```


du和df统计不一致原因

- 主要原因：
  1. 文件被删除但进程仍在使用
     - 进程打开的文件在删除后，空间不会立即释放
     - 需要重启进程或系统才能释放
  2. 文件系统损坏
     - 可能需要fsck修复
  3. 大量小文件
     - 文件系统块大小影响统计结果
- 解决方法：
  - 使用`lsof | grep deleted`查找已删除但占用的文件
  - 重启相关进程
  - 执行文件系统检查

buffers与cached的区别

- buffers（缓冲区）
  - 用于块设备I/O操作
  - 主要缓存文件系统元数据
  - 例如：目录、inode等信息
- cached（页面缓存）
  - 用于文件系统数据
  - 缓存实际文件内容
  - 提高文件读取性能
- 查看命令：

```
free -m     # 查看内存使用情况
vmstat      # 查看虚拟内存统计
```

系统启动时间优化

- 分析启动时间：

```
systemd-analyze blame     # 查看各服务启动时间
systemd-analyze critical-chain   # 查看启动链
```

- 优化方法：
  1. 禁用不必要的服务

```
systemctl disable 服务名
```

  2. 优化服务启动顺序
  3. 精简内核启动参数
  4. 使用并行启动

```
# /etc/systemd/system.conf
DefaultTimeoutStartSec=30s
```

OOM问题处理

- 原因分析：
  1. 内存泄漏
  2. 内存使用过大
  3. 系统内存不足
- 排查方法：

```
dmesg | grep -i "out of memory"  # 查看OOM日志
cat /var/log/messages            # 系统日志
top/htop                         # 监控进程内存使用
```

- 解决方案：
  1. 调整OOM优先级

```
echo -17 > /proc/PID/oom_adj  # 降低被OOM kill的概率
```

  2. 增加swap空间
  3. 限制进程内存使用

```
ulimit -v 限制值
```

  4. 优化应用程序内存使用
  5. 使用cgroup限制资源

进程通信与管理

lsof命令使用场景

- 基本用法：`lsof [选项] [参数]`
- 常见应用场景：
  1. 查看进程打开的文件

```
lsof -p PID
```

  2. 查看文件被谁占用

```
lsof /path/to/file
```

  3. 查看端口占用

```
lsof -i :80
```

  4. 查看用户打开的文件

```
lsof -u username
```

Linux进程间通信方式

1. 管道（PIPE）
    - 匿名管道：用于父子进程通信
    - 命名管道：可用于无关进程间通信
    - 场景：shell命令管道连接
2. 信号（Signal）
    - 系统中断机制
    - 场景：进程控制、异常处理
3. 共享内存（Shared Memory）
    - 最快的IPC方式
    - 场景：大数据量快速传输
4. 消息队列（Message Queue）
    - 可靠的消息传递机制
    - 场景：异步通信
5. 信号量（Semaphore）
    - 用于进程同步
    - 场景：资源访问控制
6. 套接字（Socket）
    - 可用于网络通信
    - 场景：网络服务开发

进程优先级管理

- 优先级范围：
    - nice值：-20到19（越小优先级越高）
    - 实时优先级：0到99（越大优先级越高）
- 查看优先级：

```
ps -el     # 查看进程优先级
top        # 实时查看进程状态
```

nice和renice使用

- nice：启动时设置优先级

```
nice -n 10 command     # 以优先级10启动命令
```

- renice：调整运行中进程优先级

```
renice -n 10 -p PID  # 修改进程优先级为10
```

strace使用方法

- 基本用法：

```
strace command        # 跟踪命令的系统调用
strace -p PID         # 跟踪运行中的进程
strace -c command     # 统计系统调用次数
```

- 常用选项：
  - `-f`：跟踪子进程
  - `-e trace=网络/文件/进程`：跟踪特定类型系统调用
  - `-o file`：输出到文件

内存管理

内存分页和分段

- 分页（Paging）
  - 固定大小的内存块
  - 物理地址空间连续性不要求
  - 减少内存碎片
  - 支持虚拟内存
- 分段（Segmentation）
  - 不同大小的内存块
  - 按程序逻辑划分
  - 便于共享和保护
  - 可能产生外部碎片

swap分区

- 作用：
  - 虚拟内存的延伸
  - 内存不足时的临时存储
- 使用场景：
  - 物理内存不足

    - 休眠/挂起系统

- 管理命令：

```
swapon -s           # 查看swap使用情况
free -m             # 查看内存和swap使用
swapoff/swapon      # 关闭/启用swap
```

内存使用情况查看

- 命令工具：

```
free -h             # 查看内存概况
vmstat              # 虚拟内存统计
top/htop            # 进程内存使用
/proc/meminfo       # 详细内存信息
```

- 重要指标：
    - 总内存
    - 已用内存
    - 可用内存
    - 缓存/缓冲使用

内存碎片处理

- 内存碎片类型：

    1. 内部碎片：分配单元内部未使用空间
    2. 外部碎片：空闲空间不连续

- 处理方法：

    1. 内存规整（Compaction）

    ```
    echo 1 > /proc/sys/vm/compact_memory
    ```

    2. 使用大页内存

    ```
    # 配置透明大页
    echo always > /sys/kernel/mm/transparent_hugepage/enabled
    ```

    3. 定期重启服务

    4. 内存碎片监控

```
cat /proc/buddyinfo     # 查看内存碎片情况
```

系统服务

创建和管理systemd服务

- 服务单元文件位置：`/etc/systemd/system/`
- 创建服务示例：

```
[Unit]
Description=My Custom Service
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/myapp
Restart=always

[Install]
WantedBy=multi-user.target
```

- 管理命令：

```
systemctl daemon-reload    # 重载配置
systemctl start/stop 服务名
systemctl enable/disable 服务名
```


Linux内核模块管理

- 查看模块：

```
lsmod                      # 列出已加载模块
modinfo 模块名             # 查看模块信息
```

- 加载模块：

```
modprobe 模块名            # 智能加载模块
insmod 模块路径            # 直接加载模块
```

- 卸载模块：

```
modprobe -r 模块名        # 智能卸载模块
rmmod 模块名              # 直接卸载模块
```

配置开机自启动

- systemd方式：

```
systemctl enable 服务名
```

- rc.local方式：

```
# /etc/rc.local
#!/bin/bash
/path/to/script
```

journalctl日志管理

- 基本查询：

```
journalctl              # 查看所有日志
journalctl -u 服务名     # 查看特定服务日志
journalctl -f           # 实时查看日志
```

- 时间过滤：

```
journalctl --since "2024-01-01"
journalctl --until "2024-01-31"
```

- 日志维护：

```
journalctl --vacuum-size=1G  # 限制日志大小
journalctl --vacuum-time=1w  # 限制日志时间
```

自动化运维

Ansible Roles实现K8S集群扩展

- roles结构：

```
roles/
  k8s-join/
    tasks/
    templates/
    vars/
    handlers/
```

- 主要步骤：
  1. 预检查任务

```
- name: 系统检查
  include_tasks: precheck.yml
```

  2. 安装必要软件

```
- name: 安装kubernetes组件
  yum:
    name:
      - kubelet
      - kubeadm
    state: present
```

  3. 获取join命令

```
- name: 获取join命令
  shell: kubeadm token create --print-join-command
  register: join_command
```

  4. 执行加入集群

```
- name: 加入集群
  shell: "{{ join_command.stdout }}"
```

Expect自动化交互

- 基本语法：

```
#!/usr/bin/expect
spawn ssh user@host
expect "password:"
```

```
send "password\r"
expect eof
```

- 常用命令：
  - spawn：启动进程
  - expect：等待匹配
  - send：发送响应
  - interact：切换交互模式

配置管理工具

- Puppet：
  - 声明式配置
  - 主从架构
  - Ruby DSL
- Salt：
  - 事件驱动架构
  - YAML配置
  - 高性能通信

常用运维监控工具

- 系统监控：
  - Zabbix：企业级监控系统
  - Prometheus：云原生监控
  - Nagios：传统监控系统
- 日志监控：
  - ELK Stack：日志收集分析
  - Graylog：日志管理平台
- 性能监控：
  - Grafana：数据可视化
  - Node Exporter：主机监控

安全管理

SELinux配置

- 运行模式：

```
getenforce              # 查看当前模式
setenforce 1            # 设置强制模式
```

- 配置文件：`/etc/selinux/config`
- 常用操作：

```
semanage port -a -t http_port_t -p tcp 8080  # 添加端口
restorecon -R /path    # 恢复安全上下文
```

sudo权限管理

- 配置文件：`/etc/sudoers`
- 基本语法：

```
用户名 主机=(以谁身份) 命令列表
```

- 示例配置：

```
# 允许user1执行所有命令
user1 ALL=(ALL) ALL

# 允许user2免密执行特定命令
user2 ALL=(ALL) NOPASSWD: /bin/ls, /bin/cat
```

Linux安全攻击防范

- SSH安全：

```
# /etc/ssh/sshd_config
PermitRootLogin no
PasswordAuthentication no
```

- 防火墙配置：

```
firewall-cmd --permanent --add-service=https
firewall-cmd --reload
```

- 账户安全：
  - 密码策略
  - 登录失败限制
  - 定期审计

系统安全加固

1. 基础加固：
   - 最小化安装

- 及时更新补丁
- 禁用不必要服务

2. 访问控制：

- 文件权限设置
- ACL配置
- SELinux策略

3. 网络安全：

```
# /etc/sysctl.conf
net.ipv4.tcp_syncookies = 1
net.ipv4.icmp_echo_ignore_broadcasts = 1
```

4. 日志审计：

```
# 配置auditd
auditctl -w /etc/passwd -p wa -k passwd_changes
```