[1 辅助工具与技术 3](#_Toc204528865)

[1.1 Linux 3](#_Toc204528866)

[1.1.1 Linux基础介绍 3](#_Toc204528867)

[1.1.1.1 基本概念 3](#_Toc204528868)

[1.1.1.2 Linux环境 4](#_Toc204528869)

[1.1.1.3 Linux优势 4](#_Toc204528870)

[1.1.1.4 Linux的应用 5](#_Toc204528871)

[1.1.2 Linux标准Shell-Bash 5](#_Toc204528872)

[1.1.2.1 Bash基础介绍 5](#_Toc204528873)

[1.1.2.2 基础命令 6](#_Toc204528874)

[1.1.2.3 脚本编程 7](#_Toc204528875)

[1.1.3 Linux搭建测试环境 13](#_Toc204528876)

[1.1.4 高阶内容 13](#_Toc204528877)

[1.1.4.1 Linux 系统监控与调试： 13](#_Toc204528878)

[1.1.4.2 脚本调度与自动化： 13](#_Toc204528879)

[1.1.4.3 网络调试与测试： 13](#_Toc204528880)

[1.1.4.4 性能调优与负载测试： 13](#_Toc204528881)

[1.1.4.5 容器化与云测试环境： 13](#_Toc204528882)

[1.2 数据库 13](#_Toc204528883)

[1.2.1 SQL基本介绍 13](#_Toc204528884)

[1.2.1.1 SQL的基本作用 13](#_Toc204528885)

[1.2.1.2 2. SQL语法结构 13](#_Toc204528886)

[1.2.1.3 3. SQL语句分类 14](#_Toc204528887)

[1.2.1.4 4. SQL的特点 14](#_Toc204528888)

[1.2.2 MySQL基本概念 14](#_Toc204528889)

[1.2.2.1 MySQL的基本结构 14](#_Toc204528890)

[1.2.2.2 MySQL数据类型 14](#_Toc204528891)

[1.2.2.3 MySQL的约束 15](#_Toc204528892)

[1.2.3 MySQL表管理 15](#_Toc204528893)

[1.2.3.1 创建表（CREATE TABLE） 16](#_Toc204528894)

[1.2.3.2 查看表结构（DESCRIBE / SHOW） 16](#_Toc204528895)

[1.2.3.3 修改表（ALTER TABLE） 16](#_Toc204528896)

[1.2.3.4 删除表（DROP TABLE） 17](#_Toc204528897)

[1.2.3.5 其它表操作 17](#_Toc204528898)

[1.2.4 MySQL的查询 17](#_Toc204528899)

[1.2.4.1 基本语法 18](#_Toc204528900)

[1.2.4.2 WHERE 条件筛选 18](#_Toc204528901)

[1.2.4.3 DISTINCT 去重查询 18](#_Toc204528902)

[1.2.4.4 LIMIT 限制结果数量 18](#_Toc204528903)

[1.2.4.5 五、ORDER BY 排序 18](#_Toc204528904)

[1.2.4.6 六、别名（AS） 19](#_Toc204528905)

[1.2.5 MySQL的增删改 19](#_Toc204528906)

[1.2.5.1 增：插入数据：INSERT 19](#_Toc204528907)

[1.2.5.2 改：更新数据：UPDATE 20](#_Toc204528908)

[1.2.5.3 删：部分删除数据：DELETE 20](#_Toc204528909)

[1.2.5.4 删：清空表：TRUNCATE 20](#_Toc204528910)

[1.2.6 MySQL的聚合函数 20](#_Toc204528911)

[1.2.6.1 常见聚合函数汇总 21](#_Toc204528912)

[1.2.6.2 详细语法和示例 21](#_Toc204528913)

[1.2.6.3 结合GROUP BY 与HAVING 23](#_Toc204528914)

[1.2.7 MySQL的连接（JOIN） 24](#_Toc204528915)

[1.2.7.1 INNER JOIN（内连接） 24](#_Toc204528916)

[1.2.7.2 LEFT JOIN（左连接） 25](#_Toc204528917)

[1.2.7.3 RIGHT JOIN（右连接） 25](#_Toc204528918)

[1.2.7.4 注意事项 25](#_Toc204528919)

[1.2.8 MySQL的高级查询 26](#_Toc204528920)

[1.2.8.1 子查询（Subquery） 26](#_Toc204528921)

[1.2.8.2 联合查询（UNION / UNION ALL） 26](#_Toc204528922)

[1.2.8.3 分页查询（LIMIT） 26](#_Toc204528923)

[1.2.8.4 模糊查询（LIKE） 27](#_Toc204528924)

[1.2.8.5 条件判断（CASE WHEN） 27](#_Toc204528925)

[1.2.8.6 EXISTS 子句 27](#_Toc204528926)

[1.2.8.7 别名 AS 27](#_Toc204528927)

[1.2.8.8 高级 WHERE 组合逻辑 27](#_Toc204528928)

[1.2.9 MySQL速度性能优化 27](#_Toc204528929)

[1.2.9.1 SQL语句优化（基础、常用） 28](#_Toc204528930)

[1.2.9.2 索引优化（核心） 28](#_Toc204528931)

[1.2.9.3 表结构与数据库设计优化 29](#_Toc204528932)

[1.2.9.4 服务器/配置优化（了解） 29](#_Toc204528933)

[1.2.9.5 注意事项 29](#_Toc204528934)

[1.2.10 MySQL事务和事务实际应用 29](#_Toc204528935)

[1.2.10.1 事务的四大特性（ACID） 29](#_Toc204528936)

[1.2.10.2 MySQL 事务的基本操作 30](#_Toc204528937)

[1.2.10.3 事务的隔离级别 30](#_Toc204528938)

[1.2.11 Python使用数据库 31](#_Toc204528939)

[1.2.11.1 mysql-connector-python与PyMySql安装与对比 31](#_Toc204528940)

[1.2.11.2 数据库连接 32](#_Toc204528941)

[1.2.11.3 执行Sql语句 36](#_Toc204528942)

[1.2.11.4 事务控制 39](#_Toc204528943)

[1.2.11.5 数据库操作封装 40](#_Toc204528944)

[1.2.11.6 常见问题 43](#_Toc204528945)

[1.3 Redis 43](#_Toc204528946)

[1.3.1 redis基本概念与原理 43](#_Toc204528947)

[1.3.2 redis非关系数据库 44](#_Toc204528948)

[1.3.2.1 非关系型数据库（NoSQL）概念 44](#_Toc204528949)

[1.3.2.2 Redis对比关系数据库MySql 45](#_Toc204528950)

[1.3.3 redis数据类型 45](#_Toc204528951)

[1.3.4 redis基本命令 45](#_Toc204528952)

[1.3.4.1 通用 Key 操作 45](#_Toc204528953)

[1.3.4.2 主要数据类型命令 46](#_Toc204528954)

[1.3.5 Redis在后端的应用 46](#_Toc204528955)

[1.3.6 测试中的redis 47](#_Toc204528956)

[1.3.6.1 测试中的redis和数据库 47](#_Toc204528957)

[1.3.6.2 Redis在测试的使用 47](#_Toc204528958)

[1.3.6.3 测试时访问redis/数据库 48](#_Toc204528959)

[1.3.7 Python连接使用redis 48](#_Toc204528960)

[1.3.7.1 redis-py库 48](#_Toc204528961)

[1.3.7.2 连接redis 48](#_Toc204528962)

[1.3.7.3 Redis常用操作 49](#_Toc204528963)

[1.3.7.4 注意事项 53](#_Toc204528964)

[1.4 MOCK 53](#_Toc204528965)

[1.5 ADB 53](#_Toc204528966)

[1.5.1 ADB 54](#_Toc204528967)

[1.5.1.1 ADB环境配置和连接 54](#_Toc204528968)

[1.5.1.2 ADB常规操作和方法 54](#_Toc204528969)

[1.5.1.3 ADB shell命令 54](#_Toc204528970)

[1.6 Jenkins 54](#_Toc204528971)

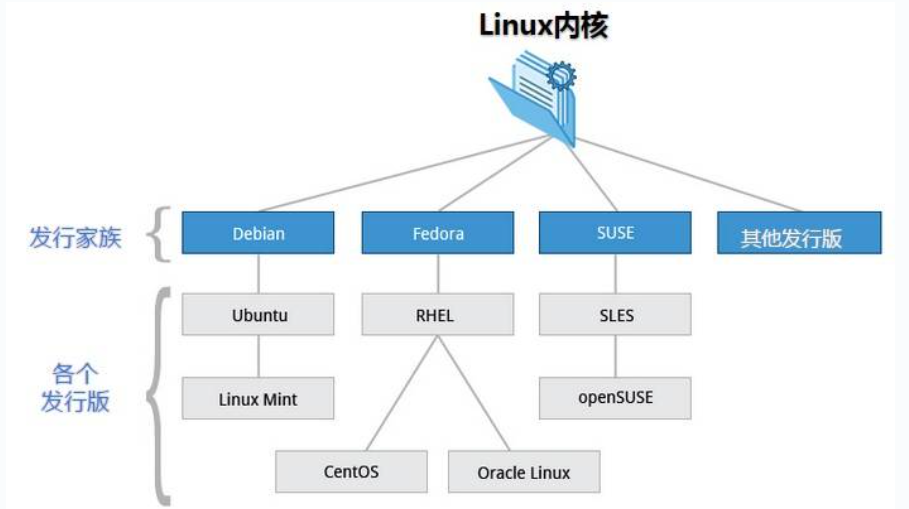
# 辅助工具与技术

## Linux

### Linux基础介绍

#### 基本概念

* **Linux**：是一种开源、类Unix的操作系统，最初由Linus Torvalds于1991年开发。它基于Unix操作系统，提供了高效、稳定、灵活的多用户、多任务处理环境，广泛用于服务器、嵌入式设备、个人电脑等领域。
* **内核（Kernel）**：Linux的内核是操作系统的核心部分，负责管理硬件资源（如CPU、内存、磁盘等），并提供用户与硬件之间的接口。
* **发行版（Distribution）**：
* Linux有很多不同的版本，称为发行版（如Ubuntu、CentOS、Debian等），这些发行版将Linux内核与各种应用软件和工具打包在一起，提供给用户安装使用。
* 目前市面上较知名的发行版有：Ubuntu、RedHat、CentOS、Debian、Fedora、SuSE、OpenSUSE、Arch Linux、SolusOS 等。



* **命令行接口（CLI）和图形用户界面（GUI）**：Linux的操作可以通过命令行（Shell）或图形界面进行。尽管图形界面在桌面环境中很常见，但许多Linux服务器通常只使用命令行。

#### Linux环境

* **文件系统：**Linux的文件系统是层级结构的，根目录是“/”，所有文件和目录从根目录开始组织。常见的目录包括：
* /bin：包含系统必要的可执行文件（如常用的命令）。
* /etc：包含系统配置文件。
* /home：用户的家目录，存储个人文件。
* /var：包含动态数据（如日志文件）。
* **用户与权限管理：**Linux是多用户操作系统，每个用户都有自己的账号和权限。系统管理员（root）拥有最高权限，普通用户拥有有限的权限。Linux通过**用户组**和**权限**（读、写、执行）管理文件和资源。
* **Shell**：Shell是用户与操作系统交互的命令行界面，常见的Shell有Bash（Bourne Again Shell）、Zsh等。用户可以通过Shell输入命令来操作系统。

#### Linux优势

* **开源与自由：**可以自由修改和分发，适用于各种需求。
* **高稳定性：**特别适用于服务器，长时间运行不容易出问题。
* **安全性：**强大的权限管理和多重安全措施减少系统攻击的风险。
* **高性能**：低资源占用，适合高效运行和高负载环境。
* **广泛的硬件支持**：支持多种硬件平台。
* **社区支持：**活跃的开源社区提供帮助和文档。

#### Linux的应用

* **服务器**：广泛用于Web服务器（如Apache、Nginx）、数据库服务器等，因其高稳定性和低资源占用。
* **嵌入式系统：**如路由器、智能设备等，Linux的定制性使其适合嵌入式开发。
* **云计算与容器化**：Linux在云平台和容器技术（Docker、Kubernetes）中应用广泛，是云计算基础设施的主流操作系统。
* **开发环境**：支持多种编程语言和开发工具，是开发者首选的操作系统。
* **安全测试：**Kali Linux等专门的渗透测试工具基于Linux，成为安全领域的标准平台。

### Linux标准Shell-Bash

#### Bash基础介绍

* **Bash是语言与命令的融合**

Bash（Bourne Again Shell）是Linux系统中最常用的Shell，作为GNU项目的一部分，它是Bourne Shell（sh）的扩展与改进。它不仅提供了命令行界面（CLI）用于用户与操作系统交互，还支持脚本编程、文件操作、系统管理等功能。

* **核心特性**
* **命令解释与执行**：Bash解释并执行命令，包括内建命令和外部程序。
* **变量和环境管理**：支持用户定义变量和环境变量，方便存储配置信息、路径、系统参数等。
* **命令替换**：允许将命令的输出嵌入其他命令中。
* **历史管理**：支持命令历史功能，用户可以方便地重复执行之前的命令。
* **自定义功能**：用户可以通过定义别名、函数等方式自定义Shell的行为。
* **Bash与Windows命令行CMD对比**

|  |  |  |
| --- | --- | --- |
| **对比项** | **Bash（Linux）** | **CMD（Windows 命令行）** |
| **是否是脚本语言** | 是，功能完备的脚本语言（Shell） | 否，主要是命令调用器（非常弱的脚本支持） |
| **流程控制（if/for/while）** | 支持完整逻辑控制 | 极为有限 |
| **函数支持** | 支持自定义函数 | 不支持 |
| **变量/运算/字符串处理** | 强，适合做文本处理 | 有限、语法繁琐 |
| **用途** | 脚本自动化、部署、测试、运维 | 简单批处理、系统操作 |
| **替代方案** | bash 脚本语言本身就强 | 推荐用 PowerShell 替代 CMD |

* **Bash与Python/JS语言对比**

|  |  |  |
| --- | --- | --- |
| **项目** | **Bash** | **Python / JavaScript** |
| 编程能力 | 有一定逻辑编程能力 | 强大的编程能力 |
| 文件处理 | 非常擅长 | 也可以做但略麻烦 |
| 调用系统命令 | 非常方便 | 需要用模块（如 Python 的 subprocess） |
| 可读性 | 简洁但不规范 | 更易维护和扩展 |
| 典型用途 | 自动化脚本、系统任务、CI流程、运维脚本 | Web、数据分析、测试脚本等通用编程 |

#### 基础命令

* **文件/目录操作**

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **示例** |
| ls | 查看文件/目录 | ls -l /home |
| cd | 切换目录 | cd /etc |
| pwd | 显示当前目录路径 | pwd |
| mkdir | 创建目录 | mkdir new\_dir |
| rmdir | 删除空目录 | rmdir old\_dir |
| rm | 删除文件/目录 | rm -rf temp/ |
| cp | 复制文件/目录 | cp file1.txt /backup/ |
| mv | 移动/重命名文件 | mv a.txt b.txt |
| touch | 创建空文件 | touch test.txt |
| find | 查找文件 | find / -name "\*.log" |
| stat | 显示文件详细属性 | stat myfile.txt |

* **权限与用户组管理**

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **示例** |
| whoami | 显示当前用户 | whoami |
| id | 显示用户UID/GID | id |
| adduser | 添加用户（Debian系） | sudo adduser tom |
| passwd | 修改用户密码 | passwd tom |

* **进程管理**

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **示例** |
| ps | 查看当前进程 | ps aux | grep python |
| top | 动态查看进程状态 | top |
| htop | 更直观进程管理器（需安装） | htop |
| kill | 杀掉指定 PID 进程 | kill 1234 |
| killall | 按名称杀进程 | killall nginx |
| jobs | 查看后台作业 | jobs |
| fg / bg | 前台 / 后台运行作业 | fg %1 / bg %1 |
| nice/renice | 设置 / 修改进程优先级 | nice -n 10 ./script.sh |

* **网络管理**

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **示例** |
| ping | 检查网络连通 | ping baidu.com |
| ip | 查看/配置网络（代替 ifconfig） | ip a |
| ifconfig | 查看网络接口（老旧） | ifconfig |
| netstat | 查看网络连接（旧） | netstat -tuln |
| ss | 查看 Socket 状态（现代） | ss -tulnp |
| curl | 请求网页/接口（支持API测试） | curl https://api.xxx.com |
| wget | 下载文件 | wget https://example.com/file.zip |

* **磁盘与系统监控**

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **示例** |
| df | 查看磁盘使用情况 | df -h |
| du | 查看文件/目录大小 | du -sh /var/log |
| free | 查看内存使用情况 | free -m |
| uptime | 系统负载与运行时间 | uptime |
| uname | 查看内核信息 | uname -a |
| hostname | 查看/设置主机名 | hostname |
| reboot | 重启系统 | sudo reboot |
| shutdown | 关闭系统 | sudo shutdown now |

* **解压缩**

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **示例** |
| tar | 打包/解包文件 | tar -czf a.tar.gz folder/ |
| gzip/gunzip | 压缩 / 解压 gzip 文件 | gzip log.txt / gunzip log.txt.gz |
| zip/unzip | 压缩 / 解压 zip 文件 | zip a.zip a.txt / unzip a.zip |

* **文本查看与处理**

|  |  |  |
| --- | --- | --- |
| **命令** | **说明** | **示例** |
| cat | 查看文件内容 | cat test.txt |
| head/tail | 查看前/后几行 | head -n 10 log.txt |
| less/more | 分页查看内容 | less largefile.txt |
| grep | 按关键字搜索 | grep "error" log.txt |
| awk | 字段级文本处理 | awk '{print $1}' file.txt |
| sed | 文本查找替换 | sed 's/old/new/g' file.txt |

* **IO管道与重定向**

|  |  |  |
| --- | --- | --- |
| **符号/命令** | **说明** | **示例** |
| > / >> | 输出重定向（覆盖/追加） | echo hello > out.txt |
| < | 输入重定向 | sort < unsorted.txt |
| ` | ` | 管道连接多个命令 |
| tee | 同时输出到终端和文件 | ls | tee list.txt |

#### 脚本编程

Bash脚本基本语言构成如下

|  |  |  |
| --- | --- | --- |
| **分类** | **说明** | **示例** |
| **变量定义** | 变量名=值，无空格 | name="Tom" |
| **环境变量** | export 导出变量 | export PATH=$PATH:/my/bin |
| **读取输入** | 使用 read | read -p "Enter name:" name |
| **命令替换** | 使用 $(...) 或 `` | today=$(date) |
| **算术运算** | $((...)) 或 let | sum=$((a + b)) |
| **字符串操作** | 截取、拼接、长度等 | ${str:0:3}, ${#str} |
| **数组支持** | 支持一维数组 | arr=(a b c)，${arr[0]} |

##### Bash脚本基础

Bash 脚本是一种以 Bash 解释器执行的命令文件，本质上是将一系列 Linux 命令按顺序组织起来，添加变量、流程控制与函数，实现自动化任务。

* **结构组成**

一个基本脚本包含以下要素：

* 解释器声明：第一行 #!/bin/bash 指明由哪种 shell 执行（也可以是 /bin/sh 等）。
* 注释：使用 #，用于说明脚本用途和作者。
* 命令序列：每行可写一个命令，也可控制其执行顺序、条件和循环。
* **执行方式**
* 添加执行权限：chmod +x script.sh
* 运行方式：
* ./script.sh（作为可执行程序）
* bash script.sh（以 Bash 解释器执行，不需要 +x）

|  |  |
| --- | --- |
| **项目** | **内容** |
| **脚本定义** | Bash 脚本是包含一系列 Bash 命令的可执行文件，通常用于自动化任务 |
| **文件扩展名** | 通常为 .sh，但并非必须 |
| **第一行 shebang** | #!/bin/bash 指定脚本解释器，必须位于第一行 |
| **执行方式** | bash script.sh 或 chmod +x script.sh && ./script.sh |

##### 变量与数据类型

Bash 是弱类型语言，变量不需声明类型，但可以通过语法手段表示不同的“数据结构”。

###### 标量变量（字符串/数字）

* 数字变量在 (( )) 中可参与算术运算，否则仅为字符串。
* 字符串支持拼接、长度、截取等操作。

name="Tom"

age=20

echo "Name: $name, Age: $age"

* 引号使用：
* 双引号 "..."：变量会被解析
* 单引号 '...'：原样输出
* 字符串操作

str="HelloWorld"

echo ${#str}         # 长度

echo ${str:0:5}      # 截取 Hello

echo ${str/World/Bash}  # 替换 HelloBash

###### 数组（Indexed Array）

* 使用括号声明，空格分隔元素。
* 用 ${arr[@]} 获取所有元素。

arr=(apple banana cherry)

echo ${arr[1]}       # banana

echo ${arr[@]}       # 所有元素

echo ${#arr[@]}      # 元素个数

###### 关联数组（Associative Array）

* 使用 declare -A 声明。
* 下标是字符串，必须用双引号括起键。

仅 Bash 4+

declare -A user

user[name]="Tom"

user[age]=21

echo ${user[name]}    # Tom

###### 环境变量与导出

export 导出变量为子进程可见。

export PATH=$PATH:/my/bin

###### 只读变量

readonly 设置不可更改变量。

readonly VERSION="1.0.0"

###### 算术运算

Bash 本身不支持浮点数，仅支持整数计算。

a=5; b=3

sum=$((a + b))        # 推荐用法

let "diff=a-b"

##### 控制结构

###### 条件判断 if/else

用于根据条件选择执行路径，判断语法一般配合 [ 条件 ] 或 [[ 条件 ]]。

if [ $a -gt 10 ]; then

  echo "大于10"

elif [ $a -eq 10 ]; then

  echo "等于10"

else

  echo "小于10"

fi

###### 条件表达式

|  |  |
| --- | --- |
| **类型** | **表达式** |
| 整数比较 | -eq, -ne, -gt, -lt, -ge, -le |
| 字符串 | =, !=, -z, -n |
| 文件 | -f, -d, -e, -r, -w, -x |
| 逻辑 | -a, -o, !，或用 &&, ` |

###### case 模式匹配

用于简洁处理多种情况选择（类似 switch-case）。

case "$input" in

  start) echo "开始" ;;

  stop) echo "停止" ;;

  \*) echo "未知命令" ;;

esac

###### 循环结构

* **for 循环（列表）**

for item in a b c; do

  echo $item

done

* **for 循环（C风格）**

for ((i=0; i<5; i++)); do

  echo $i

done

* **while 循环**

i=1

while [ $i -le 3 ]; do

  echo $i

  ((i++))

done

* **until 循环**

until [ $i -gt 3 ]; do

  echo $i

  ((i++))

done

* **控制指令**
* break：跳出循环
* continue：跳过当前迭代
* exit：退出脚本

##### 函数与模块化

* **函数定义与调用**

函数封装命令逻辑，实现模块化、可复用性。

function say\_hello {

  echo "Hello, $1"

}

say\_hello "Tom"

也可省略 function 关键字：

say\_hello() {

  echo "Hello"

}

* **参数和返回值**
* $1, $2：位置参数
* $@：所有参数（每个保留为单独参数）
* return：返回整数状态码
* 需要字符串返回值时用 echo 配合 $(...) 捕获

greet() {

  echo "Hi, $1"

}

msg=$(greet "Alice")

echo $msg

* **模块化与函数库调用**

将常用函数写入单独 .sh 文件，主脚本使用 source 引入。

# tools.sh

log() {

  echo "[LOG] $1"

}

与

# main.sh

source ./tools.sh

log "任务开始"

##### 脚本示例

一个完整 Bash 脚本包含如下结构：

* 头部信息（元数据、检查依赖）
* 全局变量（目录、配置）
* 函数定义（封装子逻辑）
* 主逻辑 main()（结构清晰）
* 异常处理（退出码、清理）

#!/bin/bash

# 备份脚本示例

SOURCE="/home/user/data"

DEST="/backup"

LOG="/var/log/backup.log"

log() {

  echo "$(date) $1" >> $LOG

}

check\_space() {

  space=$(df "$DEST" | awk 'NR==2 {print $4}')

  [ "$space" -lt 100000 ] && return 1 || return 0

}

backup() {

  ts=$(date +%Y%m%d%H%M%S)

  tar -czf "$DEST/backup\_$ts.tar.gz" "$SOURCE"

  log "Backup complete"

}

main() {

  if check\_space; then

    backup

  else

    log "Insufficient space"

    exit 1

  fi

}

main

### Linux搭建测试环境

### 高阶内容

#### Linux 系统监控与调试：

#### 脚本调度与自动化：

#### 网络调试与测试：

#### 性能调优与负载测试：

#### 容器化与云测试环境：

## 数据库

### SQL基本介绍

SQL（Structured Query Language）是一种用于管理和操作关系型数据库的标准编程语言。SQL主要用于从数据库中查询数据、插入数据、更新数据以及删除数据。它被广泛应用于数据库管理系统（DBMS）中，尤其是关系型数据库中。

#### SQL的基本作用

SQL的核心作用是用于数据库的操作，主要包括以下几个方面（CURD）：

* **数据查询**：使用 SELECT 语句从数据库中提取数据。
* **数据插入：**使用 INSERT 语句向数据库表中插入新数据。
* **数据更新：**使用 UPDATE 语句修改表中已存在的数据。
* **数据删除：**使用 DELETE 语句删除数据库表中的数据。

#### 2. SQL语法结构

SQL语法结构通常由以下部分组成：

* **关键词**：如 SELECT、INSERT、UPDATE、DELETE、FROM、WHERE、GROUP BY、HAVING 等。
* **标识符**：表名、列名、数据库名等。
* **表达式：**如运算符、函数、算式等。
* **谓词：**如 =、<、>、BETWEEN、IN、LIKE、AND、OR 等，用来表达条件。
* **命令结束符**：SQL语句通常以分号（;）结尾。

#### 3. SQL语句分类

SQL可以分为以下几类语句：

* **DQL（数据查询语言）**：主要用于数据查询操作，SELECT 是最常用的命令。
* **DML（数据操作语言）**：包括数据的插入、更新和删除操作，分别是 INSERT、UPDATE、DELETE。
* **DDL（数据定义语言）**：用于数据库对象的定义、修改、删除等操作，如 CREATE、ALTER、DROP。
* **DCL（数据控制语言）**：用于数据库的访问控制，涉及权限管理，如 GRANT 和 REVOKE。
* **TCL（事务控制语言）**：用于数据库事务的管理，如 COMMIT、ROLLBACK 和 SAVEPOINT。

#### 4. SQL的特点

* **标准化语言**：SQL是一个标准化的查询语言，几乎所有关系型数据库都支持SQL。
* **易学易用**：SQL语法接近自然语言，易于学习和使用。
* **强大的数据查询能力**：SQL能够非常高效和灵活地进行复杂的查询，支持多种查询方式和高级操作。
* **面向集合的操作**：SQL通过操作数据集合来处理数据，能够一次性处理多个数据行，避免了编程中的循环控制。

### MySQL基本概念

MySQL 是一种开源的关系型数据库管理系统（RDBMS），它使用 SQL（结构化查询语言）进行数据的查询和操作，广泛应用于 Web 开发和企业级应用中。MySQL 的设计理念是高效、稳定、易用，同时它也支持高度的可扩展性。

#### MySQL的基本结构

MySQL 的数据库架构通常包含以下几个重要组件：

* **数据库：**MySQL 存储数据的容器，可以包含多个表。每个数据库拥有独立的权限和资源。
* **表（Table）：**数据库中的数据以表的形式存储。表由行（record）和列（field）组成，每一行表示一条记录，每一列表示数据的一个字段。
* **行（Row）：**每一行代表一个实体或记录。
* **列（Column）：**每一列代表实体的一个特征或属性，定义了数据的类型。
* **字段（Field）：**指单个数据单元，对应表的一个列。

#### MySQL数据类型

数据类型常用于建表是对数据进行约束

|  |  |  |
| --- | --- | --- |
| **数据类型** | **描述** | **示例** |
| **数值类型** |  |  |
| INT | 存储整数，常用于存储较小的整数值 | INT(11) |
| BIGINT | 存储大整数值 | BIGINT |
| DECIMAL | 精确的定点数，通常用于存储金额等 | DECIMAL(10, 2) |
| FLOAT | 存储浮动小数点数 | FLOAT |
| DOUBLE | 存储双精度浮动小数点数 | DOUBLE |
| **字符类型** |  |  |
| CHAR | 固定长度的字符串 | CHAR(10) |
| VARCHAR | 变长字符串，存储较长文本 | VARCHAR(255) |
| TEXT | 用于存储较长的文本数据 | TEXT |
| **日期时间类型** |  |  |
| DATE | 存储日期，格式为 YYYY-MM-DD | DATE |
| TIME | 存储时间，格式为 HH:MM:SS | TIME |
| DATETIME | 存储日期和时间，格式为 YYYY-MM-DD HH:MM:SS | DATETIME |
| TIMESTAMP | 存储时间戳，通常用于记录记录的创建时间 | TIMESTAMP |
| **二进制类型** |  |  |
| BLOB | 存储二进制数据，如图像、文件等 | BLOB |
| **其他类型** |  |  |
| ENUM | 枚举类型，存储从一组预定义的值中选择的一个 | ENUM('small', 'medium', 'large') |
| SET | 集合类型，存储多个选项的组合 | SET('a', 'b', 'c') |

#### MySQL的约束

在 MySQL 中，约束用于限制表中数据的类型和范围，以保证数据的完整性。常见的约束包括：

* **NOT NULL**：列的值不能为空。
* **UNIQUE**：确保列中每个值都是唯一的。
* **PRIMARY KEY**：指定列为主键，表中每个值唯一且不可为 NULL。
* **FOREIGN KEY**：定义外键，保证数据的一致性和完整性。
* **CHECK**：确保列中的数据满足指定的条件（在较新版本的 MySQL 中支持）。
* **DEFAULT**：为列指定默认值。

### MySQL表管理

MySQL 中的表是存储数据的核心结构。表由行（记录）和列（字段）组成，每张表都属于一个数据库。数据库的管理主要包括：创建表、查看表结构、修改表、删除表等操作。

#### 创建表（CREATE TABLE）

* 基本语法：

CREATE TABLE 表名 (

    列名1 数据类型 [约束],

    列名2 数据类型 [约束],

    ...

);

* 🔹 示例：

CREATE TABLE users (

    id INT PRIMARY KEY AUTO\_INCREMENT,

    username VARCHAR(50) NOT NULL,

    email VARCHAR(100),

    created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP

);

* 📌 说明：
* **PRIMARY KEY：**主键；
* **AUTO\_INCREMENT：**自增；
* **NOT NULL：**非空；
* **DEFAULT：**默认值。

#### 查看表结构（DESCRIBE / SHOW）

* 查看表结构：

DESCRIBE 表名;

或者：

SHOW COLUMNS FROM 表名;

* 查看当前数据库下的所有表：

SHOW TABLES;

#### 修改表（ALTER TABLE）

用于对已存在的表进行结构上的修改：

|  |  |
| --- | --- |
| **操作** | **示例语句** |
| 添加列 | ALTER TABLE 表名 ADD 列名 数据类型; |
| 修改列类型或属性 | ALTER TABLE 表名 MODIFY 列名 新类型; |
| 修改列名 | ALTER TABLE 表名 CHANGE 旧名 新名 数据类型; |
| 删除列 | ALTER TABLE 表名 DROP COLUMN 列名; |
| 添加主键 | ALTER TABLE 表名 ADD PRIMARY KEY (列名); |
| 删除主键 | ALTER TABLE 表名 DROP PRIMARY KEY; |
| 重命名表 | RENAME TABLE 原表名 TO 新表名; |

#### 删除表（DROP TABLE）

彻底删除整个表结构和数据：

DROP TABLE 表名;

⚠️ 注意：删除后无法恢复，请谨慎使用。

#### 其它表操作

##### 清空表数据（TRUNCATE TABLE）

只删除数据，不删除表结构：

TRUNCATE TABLE 表名;

📌 TRUNCATE 比 DELETE 更高效，但不能加 WHERE 条件。

##### 复制表结构（不含数据）

CREATE TABLE 新表名 LIKE 原表名;

##### 复制表结构 + 数据

CREATE TABLE 新表名 AS SELECT \* FROM 原表名;

如果只复制部分字段或加筛选条件：

CREATE TABLE 新表名 AS SELECT id, username FROM 原表名 WHERE id < 100;

##### 查看建表语句

可以查看某张表的完整建表 SQL：

SHOW CREATE TABLE 表名;

### MySQL的查询

MySQL 中最核心、最常用的语句就是 SELECT，用于从数据库表中查询数据。

**SELECT 查询语法**

**├── 查询字段：SELECT 字段1, 字段2**

**│ └── ALL：SELECT \***

**├── 数据来源：FROM 表名**

**├── 条件筛选：WHERE 条件**

**│ ├── 比较运算符**

**│ ├── 范围：BETWEEN、IN**

**│ ├── 模糊：LIKE**

**│ └── 空值：IS NULL**

**├── 去重：DISTINCT**

**├── 排序：ORDER BY 字段 [ASC|DESC]**

**├── 限制条数：LIMIT [offset], [count]**

**└── 别名：AS**

#### 基本语法

SELECT 字段1, 字段2, ... FROM 表名;

如果要查询所有字段，可以使用星号（\*）：

SELECT \* FROM 表名;

#### WHERE 条件筛选

用于添加筛选条件，只返回符合条件的数据。

SELECT \* FROM 表名 WHERE 条件;

🔹 示例：

SELECT \* FROM users WHERE id = 1;

SELECT \* FROM users WHERE username = 'Tom';

常见操作符：

|  |  |  |
| --- | --- | --- |
| **操作符** | **含义** | **示例** |
| = | 等于 | id = 5 |
| != 或 <> | 不等于 | username != 'admin' |
| > < >= <= | 比较大小 | age > 18 |
| BETWEEN A AND B | 在范围内 | age BETWEEN 18 AND 25 |
| IN (...) | 多值匹配 | id IN (1, 3, 5) |
| LIKE | 模糊匹配 | username LIKE 't%' |
| IS NULL / IS NOT NULL | 判断空值 | email IS NULL |

#### DISTINCT 去重查询

去除重复值，只返回唯一结果。

SELECT **DISTINCT** 字段 FROM 表名;

🔹 示例：

SELECT **DISTINCT** age FROM users;

#### LIMIT 限制结果数量

控制返回结果的条数（常用于分页）：

SELECT \* FROM 表名 **LIMIT N;** -- 返回前 N 条

SELECT \* FROM 表名 **LIMIT 偏移量, 条数;** -- 分页用法

🔹 示例：

SELECT \* FROM users LIMIT 10; -- 前10条

SELECT \* FROM users LIMIT 20, 10; -- 跳过20条后取10条

#### 五、ORDER BY 排序

对结果进行排序，默认升序（ASC），降序为 DESC：

SELECT \* FROM 表名 ORDER BY 字段 **[ASC|DESC]**;

🔹 示例：

SELECT \* FROM users ORDER BY age **ASC**;

SELECT \* FROM users ORDER BY created\_at **DESC**;

也可以按多个字段排序：

SELECT \* FROM users ORDER BY age **DESC**, id **ASC**;

#### 六、别名（AS）

给查询出的字段或表起个别名，提升可读性：

SELECT 字段 AS 别名 FROM 表名 **AS** 表别名;

🔹 示例：

SELECT username AS 用户名 FROM users **AS** u;

### MySQL的增删改

相比于查询，增删改操作要相对简单

|  |  |  |
| --- | --- | --- |
| **操作** | **语句** | **说明** |
| 插入数据 | INSERT | 添加一条或多条记录 |
| 修改数据 | UPDATE | 修改表中已存在的记录 |
| 删除数据 | DELETE | 删除表中满足条件的记录 |
| 清空表 | TRUNCATE | 删除所有记录，无法加条件，更快 |

#### 增：插入数据：INSERT

##### 语法1：标准插入（所有字段）

**插入完整的一行**

INSERT INTO 表名 VALUES (值1, 值2, ...);

示例：

INSERT INTO users VALUES (NULL, 'Alice', 'alice@example.com', NOW());

* 主键为自增时可用 NULL 占位
* 注意字段顺序必须与表结构一致

##### 语法2：指定字段插入（推荐）

只插入部分内容，按需插入

INSERT INTO 表名 (字段1, 字段2, ...) VALUES (值1, 值2, ...);

示例：

INTO users (username, email) VALUES ('Bob', 'bob@example.com');

##### 插入多行数据

根据字段名对于插入的值

INSERT INTO 表名 (字段1, 字段2) VALUES

('Tom', 'tom@example.com'),

('Jerry', 'jerry@example.com');

##### 插入查询结果（从其他表）

嵌套，把查询的结果内容作为插入的数据值

INSERT INTO 新表 (字段1, 字段2)

SELECT 字段1, 字段2 FROM 原表 WHERE 条件;

#### 改：更新数据：UPDATE

用于修改表中已有的数据。

* **基本语法：**

UPDATE 表名 SET 字段1 = 新值1, 字段2 = 新值2 WHERE 条件;

示例：

UPDATE users SET email = 'new\_email@example.com' WHERE id = 3;

* **注意事项：**
* 必须加 WHERE，否则会更新整张表
* 支持更新多个字段：

UPDATE users SET username = 'Tommy', age = 20 WHERE id = 1;

#### 删：部分删除数据：DELETE

用于删除表中已有数据。

* **基本语法：**

DELETE FROM 表名 WHERE 条件;

示例：

DELETE FROM users WHERE id = 5;

* **注意事项：**
* 一定要加 WHERE 条件！否则删除全表数据
* 可与 LIMIT 搭配控制删除条数：

DELETE FROM users WHERE age < 18 LIMIT 5;

#### 删：清空表：TRUNCATE

用于删除整张表的所有数据，但不删除表结构，效率比 DELETE 更高。

TRUNCATE TABLE 表名;

### MySQL的聚合函数

聚合函数（Aggregate Functions）通常与 GROUP BY 子句联合使用，也可以直接用于整表或满足条件的数据集合。

**聚合函数**

**├── COUNT()：计数（\* / 字段）**

**├── SUM()：求和**

**├── AVG()：平均**

**├── MAX()/MIN()：最大/最小**

**└── 配合使用**

**├── GROUP BY：分组聚合**

**└── HAVING：聚合过滤**

#### 常见聚合函数汇总

|  |  |  |
| --- | --- | --- |
| **聚合函数** | **含义** | **示例用法** |
| COUNT() | 计数：返回记录的数量 | COUNT(\*) / COUNT(字段) |
| SUM() | 求和：返回某字段值的总和 | SUM(salary) |
| AVG() | 平均：返回某字段的平均值 | AVG(score) |
| MAX() | 最大：返回字段中的最大值 | MAX(age) |
| MIN() | 最小：返回字段中的最小值 | MIN(age) |

#### 详细语法和示例

##### 结合聚合函数综合语法

聚合函数一般写在 SELECT 子句中：

SELECT 聚合函数1, 聚合函数2, ...

FROM 表名

[WHERE 条件]

[GROUP BY 分组字段]

[HAVING 聚合过滤条件];

##### COUNT()：计数

* **基本语法：**

COUNT(字段名 | \*)

* **参数说明：**
* **字段名**：只统计该列中非 NULL 的值。
* **\***：统计总行数（包括 NULL）。
* **返回值：**

返回的是一个 INT 类型的整数，表示满足条件的记录条数。

* **注意：**
* COUNT(\*) 包括所有行（推荐用于总数统计）。
* COUNT(字段) 会忽略字段值为 NULL 的行
* **示例：**

SELECT COUNT(\*) FROM users;                -- 统计总行数

SELECT COUNT(email) FROM users;            -- 统计非 NULL 的 email 数

##### 2. SUM()：求和

* **基本语法：**

SUM(字段名)

* **参数说明：**
* 字段名：通常为数值型字段，如 price、salary。
* 非数值字段使用会报错。
* **返回值：**

返回总和，数据类型通常是与字段一致（如 DECIMAL、DOUBLE）。

* **注意：**

如果字段全部为 NULL，则返回 NULL。

* **示例**

SELECT SUM(salary) FROM employees;

常用于工资、价格等数值型字段的总计。

##### 3. AVG()：平均值

* **基本语法：**

AVG(字段名)

* **参数说明：**
* 字段需为数值类型。
* NULL 值不参与平均计算。
* **返回值：**

返回该列值的平均值，类型为 DECIMAL 或 DOUBLE。

* **示例**

SELECT AVG(score) FROM students WHERE class = '三年级';

##### 4. MAX() / MIN()：最大值 / 最小值

* **基本语法：**

MAX(字段名)

MIN(字段名)

* **参数说明：**
* 字段可为数值、字符串、日期类型等。
* NULL 值会被忽略。
* **返回值：**

返回该列中的最大/最小值，类型与字段一致。

* **注意：**
* 对字符串字段排序时遵循字典序。
* 日期类型会比较时间先后。
* **示例**

SELECT MAX(age) FROM users;

SELECT MIN(score) FROM students;

#### 结合GROUP BY 与HAVING

##### GROUP BY：分组聚合

GROUP BY 子句用于将查询结果**按一个或多个字段分组**，常与聚合函数（如 COUNT()、SUM()、AVG()）配合使用，对每个分组执行统计操作。

* **1.2 语法结构：**

SELECT 分组字段, 聚合函数

FROM 表名

[WHERE 条件]

GROUP BY 分组字段;

* **1.3 示例讲解：**

▶ 示例 1：按部门统计每个部门的员工数量

SELECT department, COUNT(\*) AS 员工数

FROM employees

GROUP BY department;

▶ 示例 2：按性别统计学生平均成绩

SELECT gender, AVG(score) AS 平均分

FROM students

GROUP BY gender;

* **1.4 注意事项：**
* GROUP BY 后的字段，**必须出现在 SELECT 语句**中，除非用于聚合函数。
* 如果在 SELECT 中的字段没有聚合函数修饰，就必须包含在 GROUP BY 里。
* 默认排序顺序为分组字段的升序（可加 ORDER BY 自定义排序）。

##### HAVING：聚合结果过滤

* HAVING 是用于过滤分组后的结果。
* 它的功能类似于 WHERE，但 WHERE 只能过滤原始数据，不能用于聚合函数。
* 而 HAVING 可以对聚合函数的结果进行过滤。
* **语法结构：**

SELECT 分组字段, 聚合函数

FROM 表名

[WHERE 原始过滤条件]

GROUP BY 分组字段

HAVING 聚合函数过滤条件;

* **示例讲解：**

▶ 示例 1：只查看员工数多于 5 的部门

SELECT department, COUNT(\*) AS 员工数

FROM employees

GROUP BY department

HAVING 员工数 > 5;

▶ 示例 2：统计每个城市的用户数，筛选用户数超过 10 的城市

SELECT city, COUNT(\*) AS 用户数

FROM users

GROUP BY city

HAVING 用户数 > 10;

* **注意事项：**

|  |  |  |
| --- | --- | --- |
| **比较点** | **WHERE** | **HAVING** |
| 作用阶段 | 分组前，原始数据过滤 | 分组后，聚合结果过滤 |
| 可否用聚合 | ❌ 不可以使用聚合函数 | ✅ 可以使用聚合函数 |
| 使用对象 | 原始字段 | 分组字段 / 聚合表达式 |

### MySQL的连接（JOIN）

**JOIN** 是 SQL 中用于将**两个或多个表**按特定字段**连接起来**查询的操作。一般来说语法就是JOIN 表x ON 条件

常见连接有：

|  |  |
| --- | --- |
| **类型** | **简介** |
| INNER JOIN | 内连接：只返回两个表中**匹配的行** |
| LEFT JOIN | 左连接：返回**左表全部** + 匹配行 |
| RIGHT JOIN | 右连接：返回**右表全部** + 匹配行 |
| FULL JOIN | 全连接（MySQL 不支持，可模拟） |

其中有如下对比：

|  |  |  |
| --- | --- | --- |
| **类型** | **匹配项** | **不匹配行为** |
| INNER JOIN | 两表都有 | 不返回 |
| LEFT JOIN | 左表为主 | 右表字段为 NULL |
| RIGHT JOIN | 右表为主 | 左表字段为 NULL |

#### INNER JOIN（内连接）

* **语法：**

SELECT 表1.列, 表2.列, ...

FROM 表1

INNER JOIN 表2 ON 表1.字段 = 表2.字段;

* **特点：**
* 只返回两个表中完全匹配的记录。
* 匹配不到的行将被排除。
* **示例：**

SELECT students.name, classes.class\_name

FROM students

INNER JOIN classes ON students.class\_id = classes.id;

📌 只返回 class\_id 不为 NULL 且匹配成功的学生。

#### LEFT JOIN（左连接）

* **语法：**

SELECT 表1.列, 表2.列

FROM 表1

LEFT JOIN 表2 ON 表1.字段 = 表2.字段;

* **特点：**
* 返回左表的所有记录；
* 如果在右表中有匹配，则显示；否则右表字段为 NULL。
* **示例：**

SELECT students.name, classes.class\_name

FROM students

LEFT JOIN classes ON students.class\_id = classes.id;

📌 会返回所有学生，包括没有 class\_id 的（如 Charlie）。

#### RIGHT JOIN（右连接）

* **语法：**

SELECT 表1.列, 表2.列

FROM 表1

RIGHT JOIN 表2 ON 表1.字段 = 表2.字段;

* **特点：**
* 返回右表的所有记录；
* 若左表没有匹配，左表字段为 NULL。
* **示例：**

SELECT students.name, classes.class\_name

FROM students

RIGHT JOIN classes ON students.class\_id = classes.id;

📌 会返回所有班级，包括没有学生的班级（如 id=3）。

#### 注意事项

* ON 之后是连接条件，通常是主外键对应字段。
* 如果使用 WHERE 对某个方向的 NULL 再过滤，可能影响连接结果（比如 LEFT JOIN + WHERE 右表字段不为 NULL 会退化为内连接）。
* 多表连接时推荐使用表别名提高可读性。
* **多表连接：**可以依次连接多个表（如 A JOIN B JOIN C）
* **自连接：**一个表连接自己（如查找上下级关系）

### MySQL的高级查询

高级查询主要是指在常规 SELECT 基础上使用更复杂的结构或功能来实现数据提取、分析和处理。

|  |  |  |
| --- | --- | --- |
| **功能** | **关键词** | **简要说明** |
| 子查询 | SELECT (...) | 查询中嵌套查询 |
| 联合查询 | UNION | 合并结果集 |
| 排序 | ORDER BY | 按字段排序 |
| 分页 | LIMIT | 取部分记录 |
| 模糊查询 | LIKE, %, \_ | 模糊匹配 |
| 条件判断 | CASE WHEN | 条件分类映射 |
| 去重 | DISTINCT | 结果集去重 |
| 判断存在性 | EXISTS | 判断子查询是否返回结果 |

#### 子查询（Subquery）

* **含义：**

一个 SQL 查询语句中嵌套另一个查询。

* **用法分类：**

|  |  |
| --- | --- |
| **类型** | **示例** |
| **标量子查询** | SELECT name FROM student WHERE age = (SELECT MAX(age) FROM student); |
| **列子查询** | SELECT name FROM student WHERE class\_id IN (SELECT id FROM class); |
| **行子查询** | SELECT \* FROM orders WHERE (customer\_id, product\_id) IN (SELECT customer\_id, product\_id FROM blacklist); |
| **相关子查询** | 子查询依赖外层查询，如 WHERE EXISTS (SELECT ... WHERE t1.id = t2.ref\_id) |

#### 联合查询（UNION / UNION ALL）

* **用法：**

SELECT name FROM student

UNION

SELECT name FROM teacher;

* **区别：**
* UNION 合并结果并去重
* UNION ALL 合并结果不去重

#### 分页查询（LIMIT）

* **语法：**

SELECT \* FROM student LIMIT 10 OFFSET 20;

-- 等价于：

SELECT \* FROM student LIMIT 20, 10;  -- 从第21行开始取10条

常用于前端分页加载数据。

#### 模糊查询（LIKE）

SELECT \* FROM student WHERE name LIKE '张%';

* % 表示任意数量字符
* \_ 表示任意单个字符

#### 条件判断（CASE WHEN）

类似于 SQL 中的 if-else：

SELECT name,

       CASE

           WHEN score >= 90 THEN '优秀'

           WHEN score >= 60 THEN '及格'

           ELSE '不及格'

       END AS level

FROM student;

#### EXISTS 子句

用于判断某子查询是否返回结果，常用于存在性判断。

SELECT \* FROM student s

WHERE EXISTS (SELECT 1 FROM score sc WHERE sc.student\_id = s.id);

#### 别名 AS

SELECT name AS 姓名, score AS 分数 FROM student;

* 可用于字段名、表名。
* 提高可读性、避免歧义。

#### 高级 WHERE 组合逻辑

SELECT \* FROM student

WHERE (age > 20 AND class\_id = 1) OR name LIKE '李%';

可使用 AND、OR、括号组合多个条件。

### MySQL速度性能优化

MySQL 性能优化可以从三个层面进行：

|  |  |
| --- | --- |
| **层面** | **内容关键词** |
| SQL语句层 | 查询语句写法是否高效 |
| 索引层 | 是否建立了合理的索引 |
| 服务器层 | 参数配置、硬件资源、连接数等 |

#### SQL语句优化（基础、常用）

* **避免使用 SELECT \***

不要使用 SELECT \*，只选取需要的字段，减少网络 IO 与解析压力。

* **WHERE 子句避免函数处理字段**

-- ❌ 非推荐：

WHERE YEAR(create\_time) = 2024

-- ✅ 推荐：

WHERE create\_time BETWEEN '2024-01-01' AND '2024-12-31'

* **尽量避免使用 OR，改为 IN 或 UNION**

-- ❌ 效率低：

WHERE status = 1 OR status = 2

-- ✅ 推荐：

WHERE status IN (1, 2)

* **LIMIT 分页优化（深分页慢）**

-- ❌ 慢：

SELECT \* FROM orders LIMIT 100000, 20

-- ✅ 推荐：用主键或索引作为偏移

SELECT \* FROM orders WHERE id > 100000 LIMIT 20

#### 索引优化（核心）

* **给频繁查询的字段加索引**

特别是：WHERE、ORDER BY、GROUP BY、JOIN ON 中出现的字段。

* **合理使用联合索引**

比如 index(name, age)，只有查询中包含 name 才能触发索引（最左前缀原则）。

* **避免在索引列上做运算或函数处理**

会导致索引失效，回表查询变慢。

* **观察执行计划（EXPLAIN）**

EXPLAIN SELECT \* FROM student WHERE name = '张三';

重点看：

* type（是否为 ALL、index、ref、const）
* rows（预计扫描的行数）
* possible\_keys / key

#### 表结构与数据库设计优化

* **选择合适的数据类型**
* 尽量使用定长类型（如 CHAR）而不是变长类型（如 VARCHAR）用于长度固定的字段。
* 用 TINYINT / SMALLINT 替代 INT，节省存储。
* **主键建议使用整型**

自增主键更易索引，效率更高。

* **范式/反范式设计的权衡**

有时适度冗余可换取读性能（典型：冗余用户昵称等，减少 JOIN）

#### 服务器/配置优化（了解）

日常不常接触，了解即可。

* 合理配置连接池（max\_connections）
* 调整缓冲区大小（如 innodb\_buffer\_pool\_size）
* 分库分表、读写分离、缓存层（Redis）引入

#### 注意事项

* **常见误区**

|  |  |
| --- | --- |
| **错误做法** | **问题原因** |
| 盲目添加索引 | 写入变慢、占用空间 |
| 大量使用 SELECT \* | 数据量大时浪费带宽和 CPU |
| 使用函数包裹索引字段 | 索引失效 |
| 忽视执行计划 | 无法了解 SQL 的真实开销 |

* **口诀**

**“查精简、筛索引、查计划，结构轻，分页浅。”**

* 查精简：只查需要字段，不用 \*；
* 筛索引：WHERE、JOIN 字段加索引；
* 查计划：用 EXPLAIN 观察执行计划；
* 结构轻：字段小、数据类型合适；
* 分页浅：LIMIT 要优化偏移。

### MySQL事务和事务实际应用

事务（Transaction）是指一组SQL操作，这些操作被当作一个单独的工作单元来执行。事务中的操作要么全都成功，要么全都失败，保证了数据库的一致性。

#### 事务的四大特性（ACID）

1. **原子性（Atomicity）**

事务中的所有操作要么全部完成，要么全部不做。即使中途发生故障，也不会执行部分操作。

1. **一致性（Consistency）**

事务开始前和结束后，数据库都必须处于一致的状态。例如，资金转账操作后，账户余额的总和保持一致。

1. **隔离性（Isolation）**

一个事务的执行不应受到其他事务干扰，事务之间是相互独立的。不同的隔离级别决定了不同程度的干扰。

1. **持久性（Durability）**

一旦事务提交，其对数据库的更改将被永久保存，即使系统崩溃也不受影响。

#### MySQL 事务的基本操作

1. **开启事务：START TRANSACTION 或 BEGIN**

START TRANSACTION;

-- 或者

BEGIN;

1. **提交事务：COMMIT**

用于保存事务中的所有更改，使之成为永久有效。

COMMIT;

1. **回滚事务：ROLLBACK**

用于撤销事务中的所有更改，将数据库恢复到事务开始前的状态。

ROLLBACK;

#### 事务的隔离级别

事务隔离级别定义了一个事务与其他事务之间相互隔离的程度，有以下几种常见级别：

* **READ UNCOMMITTED（读未提交）**

允许读取尚未提交的事务数据，可能出现脏读。

* **READ COMMITTED（读已提交）**

只允许读取已提交的事务数据，防止脏读，但可能出现不可重复读。

* **REPEATABLE READ（可重复读）**

保证在同一个事务中多次读取同一数据的结果是一样的，防止脏读和不可重复读，但可能出现幻读。

* **SERIALIZABLE（串行化）**

严格的隔离级别，所有事务排队执行，防止脏读、不可重复读和幻读，但性能较差。

### Python使用数据库

#### mysql-connector-python与PyMySql安装与对比

在 Python 操作 MySQL 数据库时，最常用的两种驱动库是 **mysql-connector-python** 和 **PyMySQL**。前者是由 **MySQL 官方提供的驱动程序**，具有较高的稳定性和官方支持，适合构建对兼容性和功能要求较高的项目；而后者是 **社区维护的纯 Python 实现**，轻量、易用，尤其适合快速开发、教学或与主流 Web 框架（如 Django）集成。

##### mysql-connector-python

* **开发方：**由 MySQL 官方 提供和维护。
* **安装：**

pip install mysql-connector-python

如果使用的是 MySQL 8.0 及以上版本，建议优先使用此库。

* **功能特点：**
* 完全兼容 MySQL 特性，支持 Python 原生接口。
* 支持连接池、SSL连接、自动重连等功能。
* 内置驱动无需依赖第三方库。

##### PyMySQL

* **开发方：**社区驱动的第三方库。
* **安装：**

pip install pymysql

安装体积更小，安装速度也快

* **功能特点：**
* 使用纯 Python 实现，跨平台兼容性好。
* 更容易集成到一些第三方框架（如 Django 默认支持 PyMySQL）。
* 在轻量级应用中广泛使用，社区活跃。

##### 对比与使用建议

* **语法对比**

|  |  |  |
| --- | --- | --- |
| **对比项** | **mysql-connector-python** | **PyMySQL** |
| 连接方式 | mysql.connector.connect(...) | pymysql.connect(...) |
| 游标获取 | cursor = conn.cursor() | 同上 |
| 参数占位符 | %s（必须用 %s，不能用 ?） | %s（兼容性好） |
| 返回数据类型 | 返回 MySQLCursor 对象，支持 .fetchall() | 返回 Cursor 对象，行为更贴近原生 |
| 是否默认自动提交 | 否，需要手动调用 conn.commit() | 否，需要手动调用 conn.commit() |
| 是否支持连接池 | 支持 | 默认不支持（需手动封装） |
| 官方支持程度 | 高（MySQL 官方维护） | 第三方维护（社区较活跃） |

* **使用建议**

|  |  |  |
| --- | --- | --- |
| **场景** | **建议使用库** | **理由** |
| 对接企业级 MySQL 应用 | mysql-connector-python | 稳定、官方维护、功能丰富 |
| 快速开发、教学或轻量级项目 | PyMySQL | 安装简单、文档多、社区活跃、兼容主流框架 |
| 搭配 Django 框架 | PyMySQL（默认推荐） | 兼容性好，Django 可以自动识别 |

* mysql-connector-python 更适合构建**稳健框架**
* **敏捷开发**或**测试验证**中，PyMySQL 也完全够用。

#### 数据库连接

##### 基本介绍

mysql.connector.connect()是mysql-connector-python核心的连接函数如下：

mysql.connector.connect(

    host=None,

    user=None,

    password=None,

    database=None,

    port=3306,

    unix\_socket=None,

    charset='utf8',

    collation=None,

    auth\_plugin=None,

    pool\_name=None,

    pool\_size=5,

    pool\_reset\_session=True,

    raise\_on\_warnings=True,

    use\_pure=False,

    connection\_timeout=None,

    client\_flags=None,

    compress=None,

    ssl\_disabled=False,

    ssl\_options=None,

    ssl\_key=None,

    ssl\_cert=None,

    ssl\_ca=None,

    ssl\_verify\_cert=False,

    ssl\_verify\_identity=False

)

参数说明

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **类型** | **默认值** | **描述** | **示例** |
| **host** | str | None | MySQL 服务器的主机地址。可以是 **IP 地址、域名，或者 localhost。** | "localhost", "127.0.0.1", "192.168.1.1" |
| **user** | str | 必选 | 连接到 MySQL 的用户名。 | "root" |
| **password** | str | 必选 | 对应于 user 的密码，用于身份验证。 | "password123" |
| **database** | str | None | 要连接的数据库名称。如果不指定，连接后不会自动选择数据库。 | "your\_database" |
| **port** | int | 3306 | MySQL 服务器的端口号。默认为 3306。 | 3307 |
| **unix\_socket** | str | None | Unix socket 文件路径。通常用于通过 Unix socket 连接数据库，适用于本地连接。 | "/var/run/mysqld/mysqld.sock" |
| **charset** | str | 'utf8' | 连接使用的字符集，默认为 'utf8'。可以选择其他字符集如 'utf8mb4'。 | 'utf8mb4' |
| **collation** | str | None | 用于数据库操作的字符集排序规则。通常不需要设置，除非有特殊需求。 | 'utf8\_unicode\_ci' |
| **auth\_plugin** | str | None | 指定身份验证插件，如 mysql\_native\_password 或 caching\_sha2\_password。 | 'mysql\_native\_password' |
| **pool\_name** | str | None | 连接池的名称。用于管理和复用数据库连接。 | 'mypool' |
| **pool\_size** | int | 5 | 连接池的大小，即最大连接数。 | 10 |
| **pool\_reset\_session** | bool | True | 每次从连接池中获取连接时，是否重置会话的状态。 | True |
| **raise\_on\_warnings** | bool | True | 如果设置为 True，MySQL 服务器的警告将抛出异常；设置为 False 时，忽略警告。 | False |
| **use\_pure** | bool | False | 是否使用纯 Python 实现的 MySQL 客户端。设置为 True 时，使用纯 Python 客户端，适用于没有 C 语言扩展的环境。 | True |
| **connection\_timeout** | float | None | 设置连接超时（秒）。超过此时间则自动失败。 | 10 |
| **client\_flags** | list | None | 设置客户端标志，例如 CLIENT\_FOUND\_ROWS 或 CLIENT\_MULTI\_STATEMENTS。 | [mysql.connector.ClientFlag.MULTI\_STATEMENTS] |
| **compress** | bool | None | 是否启用压缩协议，用于减少客户端和服务器之间的带宽消耗。 | True |
| **ssl\_disabled** | bool | False | 设置为 True 时禁用 SSL 连接。 | True |
| **ssl\_options** | dict | None | 设置 SSL 选项，例如 ssl\_key、ssl\_cert 等。 | {'ssl\_key': '/path/to/key.pem', 'ssl\_cert': '/path/to/cert.pem'} |
| **ssl\_key** | str | None | 客户端密钥文件路径，用于 SSL 连接。 | '/path/to/client-key.pem' |
| **ssl\_cert** | str | None | 客户端证书文件路径，用于 SSL 连接。 | '/path/to/client-cert.pem' |
| **ssl\_ca** | str | None | CA 证书文件路径，用于验证 MySQL 服务器的 SSL 证书。 | '/path/to/ca-cert.pem' |
| **ssl\_verify\_cert** | bool | False | 设置为 True 时，会验证服务器 SSL 证书的有效性。 | True |
| **ssl\_verify\_identity** | bool | False | 设置为 True 时，验证服务器的身份，确保服务器的主机名与证书匹配。 | True |

##### 常见使用示例

下面是一个常见的 MySQL 数据库连接使用实例，展示了从连接数据库到执行 SQL 语句的完整流程。以 mysql-connector-python 库为例，涉及的操作包括建立连接、执行查询、获取结果和关闭连接。

import mysql.connector

# 连接到数据库

conn = mysql.connector.connect(

    host='localhost',           # 数据库地址，'localhost'表示本地数据库

    port=3306,                  # MySQL默认端口号

    user='root',                # 数据库用户名

    password='your\_password',   # 数据库密码

    database='test\_db',         # 连接的数据库

    charset='utf8mb4'           # 字符集

)

# 获取操作游标（用于执行SQL查询）

cursor = conn.cursor()

# 执行一个简单的查询

cursor.execute("SELECT \* FROM users")

# 获取查询结果

results = cursor.fetchall()  # fetchall() 返回所有结果

# 打印查询结果

for row in results:

    print(row)

# 执行完操作后，关闭游标

cursor.close()

# 关闭连接

conn.close()

##### 注意事项

* **连接参数配置：**
* **host**：默认为 localhost，如果数据库在远程服务器上，需要使用相应的 IP 地址或域名。
* **port**：默认端口为 3306，如果 MySQL 使用了自定义端口，必须传入正确的端口号。
* **user 和 password**：请确保提供的数据库用户名和密码正确。如果使用的是超级用户（如 root），确保密码安全。
* **字符集设置：**

为避免字符集不兼容导致的乱码问题，建议设置 charset='utf8mb4'，它支持更广泛的字符集（包括表情符号等特殊字符）。

* **使用连接池：**
* 在高并发环境下，频繁创建和销毁数据库连接是一个性能瓶颈。可以使用连接池来复用数据库连接，提高性能。
* 在 mysql-connector-python 中，使用连接池时可以通过 pool\_name 和 pool\_size 参数来配置。
* **事务管理：**
* 默认情况下，mysql-connector-python 会在数据库操作后自动提交事务，但如果你设置了 autocommit=False，则需要手动调用 conn.commit() 来提交事务，避免事务未提交的情况。
* 如果在事务中发生错误，可以通过 conn.rollback() 回滚事务。
* **异常处理：**
* 使用数据库时，应该捕获异常以避免程序崩溃。常见的异常包括连接失败、SQL 错误等。可以使用 try-except 块来捕获并处理这些异常。

try:

    conn = mysql.connector.connect(

        host='localhost',

        user='root',

        password='your\_password',

        database='test\_db'

    )

except mysql.connector.Error as err:

    print(f"Error: {err}")

* **连接超时设置：**

在高负载环境中，连接可能会超时。可以通过设置 connection\_timeout 来指定连接的最大等待时间（单位：秒）。例如：connection\_timeout=10 表示最大等待时间为 10 秒。

* **资源关闭：**

每次数据库操作完成后，务必关闭游标（cursor.close()）和连接（conn.close()），避免连接资源泄漏。尤其在使用连接池时，更要注意正确的连接归还和关闭。

* **多线程/多进程问题：**

默认情况下，mysql-connector-python 是线程安全的，但你仍然需要小心多线程/多进程环境下的数据库连接管理，尤其是在没有连接池的情况下。

#### 执行Sql语句

##### cusor

**ursor（游标）** 是数据库连接中的核心对象，负责执行 SQL 语句并获取执行结果。

* **获取游标对象**

cursor = conn.cursor()  # 默认返回元组(tuple)

或

cursor = conn.cursor(dictionary=True)  # 返回字典格式，更便于读取字段名

* **注意**
* 若需要结构化处理数据，推荐使用 dictionary=True
* 每次数据库操作都应使用 cursor，操作完成后需关闭

##### execute()详解

cursor.execute(sql, params=None)

* **sql：**要执行的 SQL 语句，支持占位符。
* **params：**可选的参数，用于传递 SQL 占位符对应的值。
* **占位符使用**

**mysql-connector-python 使用 %s 占位符（而不是 ? 或 :param）。**

sql = "INSERT INTO users (username, age) VALUES (%s, %s)"

values = ('Alice', 25)

cursor.execute(sql, values)

**注意：**即使插入整数，也要用 %s 占位，不用担心类型。

* **批量插入**

executemany() 会自动为每组参数执行一次语句，常用于批量插入。

sql = "INSERT INTO users (username, age) VALUES (%s, %s)"

values = [

    ('Tom', 22),

    ('Jerry', 24),

    ('Anna', 21)

]

cursor.executemany(sql, values)

conn.commit()

##### 查询结果获取

* 查询操作后需调用 fetch...() 系列函数获取数据。
* 对于大批量数据，推荐使用 fetchmany() + 分批处理，节省内存。

|  |  |
| --- | --- |
| **方法** | **说明** |
| **fetchone()** | 获取单行结果 |
| **fetchmany(size)** | 获取指定数量的结果 |
| **fetchall()** | 获取所有结果 |

* 示例

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall()

for row in rows:

    print(row)

字典模式：

cursor = conn.cursor(dictionary=True)

cursor.execute("SELECT name, age FROM users")

for user in cursor.fetchall():

    print(user["name"], user["age"])

##### DDL与DML执行

DDL与DML语句不会有结果集，只需执行即可，注意是否需要 commit()：

* **DDL（数据定义语言）**

创建/修改表等结构性操作：

cursor.execute("""

CREATE TABLE IF NOT EXISTS users (

    id INT AUTO\_INCREMENT PRIMARY KEY,

    name VARCHAR(50),

    age INT

)

""")

conn.commit()  # 注意仍需 commit

* **DML（数据操作语言）**

插入（INSERT）、更新（UPDATE）、删除（DELETE）

cursor.execute("INSERT INTO users (name, age) VALUES (%s, %s)", ("Lucy", 18))

cursor.execute("UPDATE users SET age = %s WHERE name = %s", (19, "Lucy"))

cursor.execute("DELETE FROM users WHERE name = %s", ("Lucy",))

conn.commit()

##### 注意事项

|  |  |
| --- | --- |
| **项目** | **说明** |
| 必须提交 | DML（增删改）操作需显式调用 conn.commit() |
| 防止注入 | 永远不要拼接 SQL 字符串，使用参数化 %s |
| 自动关闭 | 操作完成应 cursor.close() 与 conn.close()，可使用 with 封装 |
| 字符集问题 | 连接时设置 charset='utf8mb4'，防止中文乱码 |
| 多结果集 | mysql-connector-python 不支持一次执行多个语句（不支持分号连接的多条语句） |

##### 示例

from mysql.connector import connect, Error

try:

    conn = connect(

        host='localhost',

        user='root',

        password='your\_password',

        database='test\_db',

        charset='utf8mb4'

    )

    cursor = conn.cursor(dictionary=True)

    # 创建表

    cursor.execute("""

        CREATE TABLE IF NOT EXISTS users (

            id INT PRIMARY KEY AUTO\_INCREMENT,

            name VARCHAR(50),

            age INT

        )

    """)

    # 插入数据

    insert\_sql = "INSERT INTO users (name, age) VALUES (%s, %s)"

    cursor.execute(insert\_sql, ("John", 25))

    # 查询数据

    cursor.execute("SELECT \* FROM users")

    for row in cursor.fetchall():

        print(f"User {row['id']}: {row['name']} - {row['age']}")

    # 更新数据

    cursor.execute("UPDATE users SET age = %s WHERE name = %s", (30, "John"))

    # 删除数据

    cursor.execute("DELETE FROM users WHERE name = %s", ("John",))

    conn.commit()

except Error as e:

    print("数据库操作出错：", e)

    conn.rollback()

finally:

    if cursor:

        cursor.close()

    if conn:

        conn.close()

#### 事务控制

**事务（Transaction）** 是一组要么全部执行、要么全部不执行的操作，主要用于保证数据的一致性和完整性。mysql-connector-python进行事务控制的主要方法就是不自动commit，手动添加操作后再一次性执行commit

* **控制方法**

conn.commit()       # 提交事务

conn.rollback()     # 回滚事务

conn.autocommit = True / False  # 设置是否自动提交或连接数据库时设置

* **示例**

try:

    conn = mysql.connector.connect(

        host='localhost',

        user='root',

        password='your\_password',

        database='test\_db'

    )

    cursor = conn.cursor()

    # 开启事务（默认已关闭自动提交）

    sql1 = "UPDATE accounts SET balance = balance - 100 WHERE id = 1"

    sql2 = "UPDATE accounts SET balance = balance + 100 WHERE id = 2"

    cursor.execute(sql1)

    cursor.execute(sql2)

    # 提交事务

    conn.commit()

except mysql.connector.Error as e:

    print("发生异常，执行回滚:", e)

    conn.rollback()

finally:

    cursor.close()

    conn.close()

#### 数据库操作封装

在**实际项目中，数据库操作“封装”是非常常见的实践**

##### 设计目标

|  |  |
| --- | --- |
| **需求** | **封装体现** |
| 快速插入/删除测试数据 | 支持 insert\_dict()、delete\_by\_column() 等常用方法 |
| 避免频繁写 SQL | 数据操作支持字典传参，拼接 SQL 自动处理 |
| 数据清理自动化 | 提供 TestDataContext 实现自动清理测试数据 |
| 易调试、易扩展 | 结构清晰、适合用于测试环境 |

##### 目录结构建议（用于测试项目）：

**project/**

**├── db/**

**│ ├── db\_config.py # 数据库配置**

**│ ├── db\_helper.py # 核心封装类**

**│ └── data\_context.py # 数据上下文（可选）**

**├── tests/**

**│ └── test\_case\_xxx.py # 调用封装进行数据准备与验证**

##### db\_config.py（配置文件）

配置文件配置好远程连接数据库的配置

# db/db\_config.py

DB\_CONFIG = {

    'host': '127.0.0.1',

    'port': 3306,

    'user': 'test\_user',

    'password': 'test\_pass',

    'database': 'test\_db',

    'autocommit': True,  # 开启自动提交

}

##### db\_helper.py（核心封装类）

就是把一些最常使用的数据库查询进行封装，例如某个网上商城系统可能需要频繁查询购物车列表，就可以将其封装好来方便调用

# db/db\_helper.py

import mysql.connector

from mysql.connector import Error

from db.db\_config import DB\_CONFIG

class DBHelper:

    def \_\_init\_\_(self, config=None):

        self.config = config or DB\_CONFIG

        self.conn = mysql.connector.connect(\*\*self.config)

        self.cursor = self.conn.cursor(dictionary=True)

    def insert\_dict(self, table, data: dict):

        """将字典数据插入表中，返回自增 ID"""

        keys = ", ".join(data.keys())

        values = ", ".join(["%s"] \* len(data))

        sql = f"INSERT INTO {table} ({keys}) VALUES ({values})"

        self.cursor.execute(sql, list(data.values()))

        self.conn.commit()

        return self.cursor.lastrowid

    def query\_by\_column(self, table, column, value):

        """根据某字段查询一行记录"""

        sql = f"SELECT \* FROM {table} WHERE {column} = %s LIMIT 1"

        self.cursor.execute(sql, (value,))

        return self.cursor.fetchone()

    def delete\_by\_column(self, table, column, value):

        """根据字段删除记录"""

        sql = f"DELETE FROM {table} WHERE {column} = %s"

        self.cursor.execute(sql, (value,))

        self.conn.commit()

        return self.cursor.rowcount

    def count\_rows(self, table, where\_clause=""):

        sql = f"SELECT COUNT(\*) as count FROM {table}"

        if where\_clause:

            sql += f" WHERE {where\_clause}"

        self.cursor.execute(sql)

        return self.cursor.fetchone()['count']

    def execute\_raw(self, sql, params=None):

        """执行原生 SQL"""

        self.cursor.execute(sql, params or ())

        return self.cursor.fetchall()

    def close(self):

        self.cursor.close()

        self.conn.close()

##### data\_context.py（可选：自动清理测试数据）

# db/data\_context.py

from db.db\_helper import DBHelper

class TestDataContext:

    def \_\_init\_\_(self, db: DBHelper, table, data\_list: list[dict]):

        self.db = db

        self.table = table

        self.data\_list = data\_list

        self.ids = []

    def \_\_enter\_\_(self):

        for row in self.data\_list:

            rid = self.db.insert\_dict(self.table, row)

            self.ids.append(rid)

        return self.ids

    def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

        for rid in self.ids:

            self.db.delete\_by\_column(self.table, "id", rid)

##### 示例：在测试中使用（test\_case\_user.py）

from db.db\_helper import DBHelper

from db.data\_context import TestDataContext

db = DBHelper()

def test\_query\_user():

    with TestDataContext(db, "users", [{"username": "临时用户A", "role": "tester"}]) as inserted\_ids:

        uid = inserted\_ids[0]

        user = db.query\_by\_column("users", "id", uid)

        assert user['username'] == "临时用户A"

        assert user['role'] == "tester"

#### 常见问题

## Redis

### redis基本概念与原理

Redis（Remote Dictionary Server） 是一个开源的、高性能的、基于内存的键值对（Key-Value）存储系统。它既可以作为数据库，也可以作为缓存系统和消息中间件使用。

* **核心作用**

粗略地说，Redis的目的就是「超快 + 高并发」

* **超快：**Redis 读写延迟通常在 0.1 ms 级别，体现为：
* 页面缓存命中后首屏响应显著加速
* 实时鉴权、限流、计数等操作几乎无感延时
* **高并发：**单实例可轻松承载数十万 QPS，集群可线性扩展至百万级，体现为：
* 大规模用户同时在线时，Session 校验、消息发布/订阅依然流畅
* 高峰期秒杀、抢票等场景下，缓存层稳住流量，后端库不崩溃
* **主要特征：**

1. 基于内存，读写速度极快，可以处理大量读写请求。
2. 支持多种数据结构，如字符串、哈希、列表、集合、有序集合等，具有丰富的数据表示能力。
3. 支持主从复制，提供数据冗余和故障恢复能力。
4. 支持持久化，可以将内存数据保存到磁盘中。
5. 支持事务，可以一次执行多个命令。
6. 丰富的功能，可用于缓存、消息队列等场景。

* **Redis原理:**

核心原理可概括为**内存优先架构+单线程事件驱动+高效数据结构+持久化与扩展机制：**

* **内存存储架构**​​

Redis**所有数据存储在内存中**，读写速度可达微秒级（读11万次/秒，写8.1万次/秒）。通过jemalloc优化内存分配，并支持动态数据结构编码（如小数据用压缩列表）以节省内存。内存管理采用LRU/LFU等淘汰策略防止溢出。

* **单线程模型​​**

采用**单线程处理命令，避免多线程锁竞争和上下文切换**。通过**I/O多路复用**（如epoll）监听数万并发连接，6.0版本后引入多线程仅处理网络I/O，核心命令执行仍为单线程以保证原子性。

* **高效数据结构​​**

支持**字符串、哈希、列表等7种数据结构**，底层采用**SDS动态字符串**（O(1)长度获取）、**跳表**（有序集合范围查询）、**压缩列表**（小数据紧凑存储）等优化结构。

* ​​**持久化机制​​**
* ​​**RDB快照**​​：定时全量备份，恢复快但可能丢失最新数据。
* ​​**AOF日志​​**：记录写操作命令，数据更安全但文件较大。
* ​​**混合模式**​​（Redis 6.0+）:结合RDB和AOF,启动时先加载快照再重放增量日志。
* ​​**高可用架构​​**
* ​​**主从复制**​​：数据从主节点同步到从节点，支持读写分离。
* ​​**哨兵模式**​​：监控节点状态，自动故障转移（如主节点宕机时提升从节点）。
* ​​**集群模式**​​：数据分片到16384个哈希槽，支持水平扩展和故障自愈。
* **主要应用场景：**

1. 缓存常见的使用场景，比如缓存查询结果、热点数据等，大大降低数据库负载。
2. 处理大量的读写请求，比如访问统计、消息队列等。
3. 排行榜、计数器等功能的实现。
4. pub/sub消息订阅。
5. QUE计划任务
6. 分布式锁等。

### redis非关系数据库

#### 非关系型数据库（NoSQL）概念

* 与传统的关系型数据库（如 MySQL）不同，NoSQL 不以表格的方式存储数据，没有固定的模式（schema）；与传统数据库相比，NoSQL以​**​灵活性+扩展性​**​为核心优势，但弱化复杂查询与事务支持
* Redis 属于 NoSQL（非关系型数据库）中最典型的 Key‑Value 存储，但其丰富的数据模型又超越了“简单 KV”范式：
* **无模式（Schema‑less）：**Key 和 Value 的结构可随时动态变化，无需事先定义表结构，也无需迁移脚本，极大提升迭代速度。
* **扁平访问：**所有操作以 Key 为入口，无 JOIN、无多表事务，读写路径简单、延迟可控。
* **分布式扩展：**天然支持主从复制、哨兵故障转移、Cluster 分片，轻松实现水平扩展与高可用。

#### Redis对比关系数据库MySql

|  |  |  |
| --- | --- | --- |
| **项目** | **Redis** | **MySQL（关系型数据库）** |
| 存储方式 | 内存 | 磁盘 |
| 数据模型 | Key-Value | 表结构（行与列） |
| 查询方式 | 键（K）访问 | SQL |
| 事务支持 | 简单事务 | 完整事务（ACID） |
| 性能 | 极高（内存操作） | 相对较慢（磁盘IO） |
| 扩展性 | 易于水平扩展（Cluster） | 水平扩展较难 |

Redis 以 ​​内存速度 + 灵活数据结构​​ 为核心，适合高并发实时场景；MySQL 以 ​​强一致性 + 复杂查询​​ 为优势，适合结构化数据存储。实际应用中，二者常结合使用：​​Redis 抗流量，MySQL 保数据​​，例如“缓存-数据库”分层架构

### redis数据类型

Redis 支持多种 Value 类型，每种都在内存中经过高度优化，满足不同场景需求

|  |  |  |
| --- | --- | --- |
| **类型** | **描述** | **场景示例** |
| **String** | 字符串或二进制安全的字节序列（最大 512 MB） | 缓存 JSON、计数器、Token |
| **Hash** | 字段–值映射（对象） | 用户 Profile、购物车 |
| **List** | 双端链表（可阻塞／非阻塞推拉） | 任务队列、时间线 |
| **Set** | 无序唯一集合 | 标签系统、交集运算 |
| **Sorted Set** | 带分数的有序集合（跳表＋哈希表） | 排行榜、延时队列 |
| **Bitmap** | 位图（按位操作） | 用户签到、状态标记 |
| **HyperLogLog** | 基数估计算法（≈12 KB） | 大规模去重计数（UV 统计） |
| **Stream** | 日志／消息流（带 ID、消费组） | 实时消息队列、事件流处理 |

### redis基本命令

#### 通用 Key 操作

|  |  |
| --- | --- |
| **命令** | **功能** |
| `SET key value [EX 秒] [PX 毫秒] [NX | XX]` |
| GET key | 读取字符串 |
| DEL key | 删除 Key |
| EXPIRE key 秒 | 设置过期时间（秒），TTL key 查看剩余寿命 |
| TYPE key | 查看 Key 的数据类型 |

#### 主要数据类型命令

* **String**

INCR key # 自增

APPEND key val # 追加

GETRANGE key s e # 子串

* **Hash**

HSET key f v # 字段设值

HGET key f # 字段取值

HGETALL key # 全部字段

HINCRBY key f i # 字段自增

* **List**

LPUSH key v # 左入

RPUSH key v # 右入

LPOP key # 左出

BRPOP key timeout # 阻塞右出

LRANGE key 0 -1 # 全部元素

* **Set**

SADD key m… # 添加成员

SREM key m # 删除成员

SMEMBERS key # 列表所有成员

SISMEMBER key m # 成员检查

* **Sorted Set**

ZADD key score member # 添加/更新

ZRANGE key 0 -1 WITHSCORES # 按分升序

ZREVRANGE key 0 -1 WITHSCORES # 按分降序

ZSCORE key member # 查看分数

### Redis在后端的应用

在典型前后端分离架构中，Redis 常被用作后端的「高速缓存层」「状态管理层」和「轻量消息中间件」：

* **Cache‑Aside 缓存**
* 读：先查 Redis，Miss 再读库并回写；写：更新库后删失效缓存。
* 降低主库压力，提升响应速度。
* **Session／Token 存储**
* 将用户会话或 JWT 存入 Redis Hash，支持多实例共享登录态。
* **消息队列／事件流**
* 基于 List 或 Stream 实现异步任务队列、实时日志处理。
* **分布式锁**
* 利用 SET key val NX PX ttl 原子命令或 RedLock 算法，保护并发关键区块。
* **限流与计数**
* 通过 INCR+EXPIRE 或 Lua 脚本实现滑窗、漏桶限流，防止洪峰冲击。
* **排行榜与实时统计**
* 使用 Sorted Set 做游戏／活动排行榜；Bitmap、HyperLogLog 做用户行为统计。

后端使用redis可以类比操作系统使用内存：

* 后端程序先查询redis有没有需要的内容，如果有（hit）直接取；如果没有（miss）则去磁盘的关系数据库取，同时带上合理TTL（time to live，生存时间）更新redis；
* 写数据时，一般直接操作磁盘mysql数据库，然后更新redis或删除原本内容
* 需要注意的是，具体如何编写如何取如何查磁盘数据库，以及写内容的更新都是需要自行编码的，redis只是提供这个功能。

### 测试中的redis

#### 测试中的redis和数据库

在测试中，redis和持久数据库都有各自的作用

* **Redis：**用于缓存类或临时数据（如验证码、token、session、限流标志等），短期状态验证主要依赖 Redis。
* **数据库：**用于业务核心数据的持久化（如用户信息、订单数据、状态变更等），长期结果验证依赖数据库。

测试中通常需要同时验证接口响应 + Redis 状态 + 数据库记录，确保前后端行为和数据一致

|  |  |  |
| --- | --- | --- |
| **类型** | **数据来源** | **主要用途** |
| Redis | 内存缓存数据库 | 用于存储验证码、token、session、限流标记、临时状态等“短期敏感数据” |
| 数据库 | 关系型或非关系型数据库（MySQL、Mongo等） | 用于存储用户、订单、日志等“持久化业务数据” |

#### Redis在测试的使用

|  |  |  |
| --- | --- | --- |
| **场景类型** | **Redis 具体作用** | **测试中怎么用** |
| 接口测试 | 存储验证码、token、临时ID | 从 Redis 读取验证码、token 来构造测试请求 |
| 功能验证 | 缓存/会话状态校验 | 比对 Redis 中数据是否正确写入/清除（如退出登录后 token 是否清除） |
| 登录验证 | 登录态存入 Redis | 查看 Redis 中是否生成 session/token |
| 退出验证 | 清除 session/token | 校验 Redis 中登录信息是否被删除 |
| 并发/幂等测试 | Redis 实现请求限流/幂等 | 观察 Redis 中是否正确设置标记值；验证接口是否重复处理 |
| 性能测试 | 缓存命中率、限流规则 | 检查 Redis 中缓存是否生效，限流计数是否正确 |
| 自动化测试准备 | 初始化或清除测试状态 | 在 setup/teardown 时写入或清除 Redis 中的 key，确保测试环境一致 |
| 问题排查 | 检查状态异常是否与缓存有关 | 排查登录失败、缓存不同步等问题时需查看 Redis |

#### 测试时访问redis/数据库

一般来说测试代码与项目代码是分离的，所以需要远程访问redis/数据库才能进行数据验证。一般来说要么后端提供测试验证接口，要么直接访问redis/数据库。

|  |  |
| --- | --- |
| **方式** | **说明** |
| 直连测试环境的 Redis/数据库（白盒） | 需有访问权限和账号（如 Dev、Staging 环境） |
| 后端开放专用测试接口（灰盒） | 提供如“获取验证码”、“查询缓存状态”接口 |
| 测试自动化脚本中封装连接代码 | 如使用 redis-py、pymysql，在 CI/CD 流程或本地执行时访问 |
| 配合运维/后端日志排查 | 若权限受限，可与开发沟通，通过日志或运维支持侧查数据状态 |

### Python连接使用redis

类似于SQL，python也可以连接redis进行数据的获取、检查。

#### redis-py库

redis-py库是Redis 官方推荐的 Python 客户端库（原为第三方，现由 Redis Labs 维护），支持绝大多数 Redis 命令、连接池、事务、管道、发布订阅、集群等

**安装：**

pip install redis

或支持 Redis Cluster的

pip install redis[cluster]

#### 连接redis

通过redis.Redis()函数进行连接，得到redis对象进行操作：

* **函数与参数说明：**

r = redis.Redis(

    host='127.0.0.1',         # Redis 服务地址

    port=6379,                # Redis 端口

    db=0,                     # Redis 数据库编号（0~15）

    password='yourpass',      # 连接密码（如果有设置）

    decode\_responses=True,    # 是否自动解码为字符串（推荐开启）

    socket\_timeout=5          # 连接超时（秒）

)

* **示例：**

import redis

# 本地连接（默认端口6379，无密码）

r = redis.Redis(host='localhost', port=6379, db=0)

# 远程连接示例

r = redis.Redis(

    host='your.redis.host',   # 远程IP或域名

    port=6379,                # Redis端口

    password='yourpassword', # Redis密码（如设置了）

    db=0,                     # 默认使用第0个数据库

    decode\_responses=True     # 自动解码为字符串

)

# 设置键值对

r.set('username', 'chatgpt')

# 获取键值

print(r.get('username'))  # 输出: b'chatgpt'

#### Redis常用操作

##### 字符串（String）操作

# 设置键值对

r.set('name', 'Alice')

# 获取键值

r.get('name')  # 返回 b'Alice'，若设置 decode\_responses=True 则是 'Alice'

# 设置过期时间（单位：秒）

r.set('temp', '123', ex=10)

# 批量设置/获取

r.mset({'k1': 'v1', 'k2': 'v2'})

r.mget('k1', 'k2')  # 返回 [b'v1', b'v2']

#递增/递减（值必须是整数）

r.incr('counter')     # 自增 1

r.incrby('counter', 5)

r.decr('counter')

##### 哈希（Hash）操作（类似 Python 字典）

# 设置单个字段

r.hset('user:1', 'name', 'Tom')

# 获取字段值

r.hget('user:1', 'name')

# 设置多个字段

r.hmset('user:1', {'age': 18, 'gender': 'male'})  # 兼容用法，官方建议用 hset + dict

# 获取多个字段值

r.hmget('user:1', 'name', 'age')

# 获取所有字段名和值

r.hgetall('user:1')  # 返回 dict

# 判断字段是否存在

r.hexists('user:1', 'gender')

# 删除字段

r.hdel('user:1', 'gender')

##### 列表（List）操作（类似双端队列）

# 从左边插入元素（先进后出）

r.lpush('mylist', 'a', 'b', 'c')

# 从右边插入（先进先出）

r.rpush('mylist', '1', '2', '3')

# 获取指定区间元素

r.lrange('mylist', 0, -1)

# 弹出元素

r.lpop('mylist')   # 左弹出

r.rpop('mylist')   # 右弹出

# 查看长度

r.llen('mylist')

# 设置指定索引元素

r.lset('mylist', 1, 'new')

# 删除列表中指定值（count 表示删除几个）

r.lrem('mylist', count=1, value='a')

##### 集合（Set）操作（无序、去重）

# 添加成员

r.sadd('myset', 'a', 'b', 'c')

# 删除成员

r.srem('myset', 'a')

# 判断是否存在

r.sismember('myset', 'b')

# 获取所有成员

r.smembers('myset')  # 返回 set 类型

# 获取集合大小

r.scard('myset')

# 随机弹出一个成员

r.spop('myset')

# 求交集/并集/差集

r.sinter('set1', 'set2')

r.sunion('set1', 'set2')

r.sdiff('set1', 'set2')

##### 有序集合（Sorted Set）操作（按分数排序）

# 添加元素（带分数）

r.zadd('scoreboard', {'Tom': 100, 'Jerry': 80})

# 获取成员分数

r.zscore('scoreboard', 'Tom')

# 获取成员排名（默认从小到大）

r.zrank('scoreboard', 'Tom')

# 倒序排名

r.zrevrank('scoreboard', 'Tom')

# 按范围获取成员

r.zrange('scoreboard', 0, -1, withscores=True)

r.zrevrange('scoreboard', 0, -1, withscores=True)

# 删除成员

r.zrem('scoreboard', 'Jerry')

##### 键管理（Key 操作）

# 判断键是否存在

r.exists('key1')

# 删除键

r.delete('key1')

# 设置过期时间（秒）

r.expire('key1', 10)

# 获取剩余过期时间（秒）

r.ttl('key1')

# 重命名键

r.rename('old\_key', 'new\_key')

# 获取所有键（不推荐在生产环境使用）

r.keys('\*')

##### 事务（事务块）

pipe = r.pipeline()

pipe.set('k1', 'v1')

pipe.incr('counter')

pipe.execute()  # 原子执行多个操作

##### 管道（pipeline，节省通信开销）

pipe = r.pipeline(transaction=False)

pipe.set('a', 1)

pipe.set('b', 2)

pipe.get('a')

result = pipe.execute()  # 批量执行提高效率

##### 发布订阅（Pub/Sub）

# 发布

r.publish('news', 'Python 3.12 released!')

# 订阅

pubsub = r.pubsub()

pubsub.subscribe('news')

for msg in pubsub.listen():

    print(msg)  # msg['type'], msg['data']

##### 其他技巧与常用封装

* 自动解码返回值（推荐开发中加上）

r = redis.Redis(decode\_responses=True)

* 设置键值并带过期时间（常见于验证码、缓存）

r.set('verify\_code', '123456', ex=300)  # 5分钟过期

#### 注意事项

* **使用专用数据库**

Redis 默认有 16 个数据库（db=0~15），建议用 db=15 做测试，防止污染正式数据：

r = redis.Redis(host='localhost', port=6379, db=15)

* **每次测试清空数据**

避免数据残留，测试前后使用：r.flushdb()清空数据，可封装到 pytest fixture 中做自动管理。

* **键名加统一前缀**

避免冲突，便于清理：

r.set("test:user:001", "Tom")

* **连接池推荐使用**

提高效率，适用于多线程测试：

pool = redis.ConnectionPool(...)

r = redis.Redis(connection\_pool=pool)

* **避免阻塞命令**

如 blpop 应加 timeout，防止卡死：

r.blpop('q', timeout=2)

* **存复杂数据需序列化**

使用 JSON 安全可靠：

r.set('user', json.dumps({'name': 'Tom'}))

* **常见测试场景示例**

Redis 在软件测试中常用于：

* 登录态或验证码的缓存验证
* 接口限流功能的测试
* 后台任务标志状态的存取
* 消息队列或订阅发布机制模拟

## MOCK

## ADB

### ADB

#### ADB环境配置和连接

#### ADB常规操作和方法

#### ADB shell命令

## Jenkins