

哈尔滨工业大学

计算学部

2024 年秋季学期

《软件架构与中间件》课程

实验报告

Lab-3 : 数据层软件架构实验

姓名	学号	联系方式
余昊卿	2023120253	felixupri@gmail.com
马嘉良	2023120259	13596489104

目 录

1 实验概述	1
1.1 实验目的	1
1.2 实验要求	1
2 实验内容与过程	1
2.1 Mycat 数据库分库分表实验	1
2.2 Sharding-JDBC 数据库分库分表实验	5
2.3 Redis 数据缓存实验	10
3 结对开发过程记录	20
4 实验总结	20
5 教师评语	21

学号:	2023120253	姓名:	余昊卿
学号:	2023120259	姓名:	马嘉良

1 实验概述

1.1 实验目的

- 1) 学习使用 Mycat 和 Sharding-JDBC 实现数据分库分表
- 2) 学习使用 Redis 数据库实现数据缓存
- 3) 能够灵活应用 Mycat 或 Sharding-JDBC 实现分库分表架构到实际系统
- 4) 能够灵活应用 Redis 实现数据缓存架构到实际系统
- 5) 能够灵活应用计算层中间件到实际系统

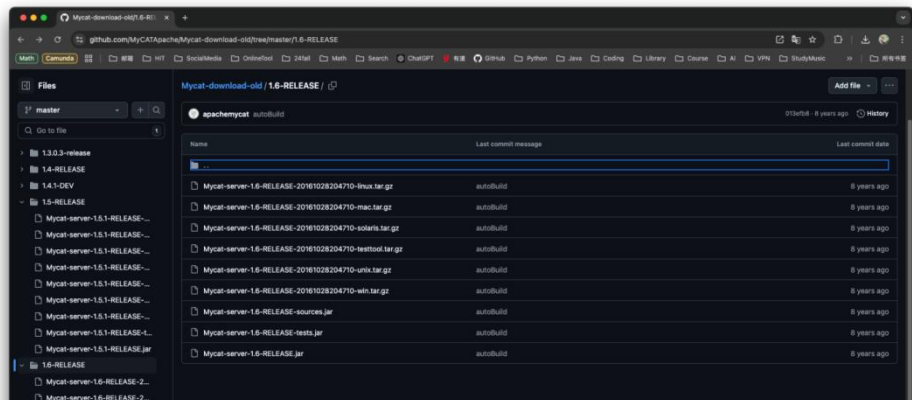
1.2 实验要求

- 1) 2 人结对成组
- 2) 结合《软件过程与工具》课程中进销存系统(或其他实际软件系统)进行计算层架构重构, 支持海量用户的在线高并发请求场景
- 3) 应给出关键过程的细节

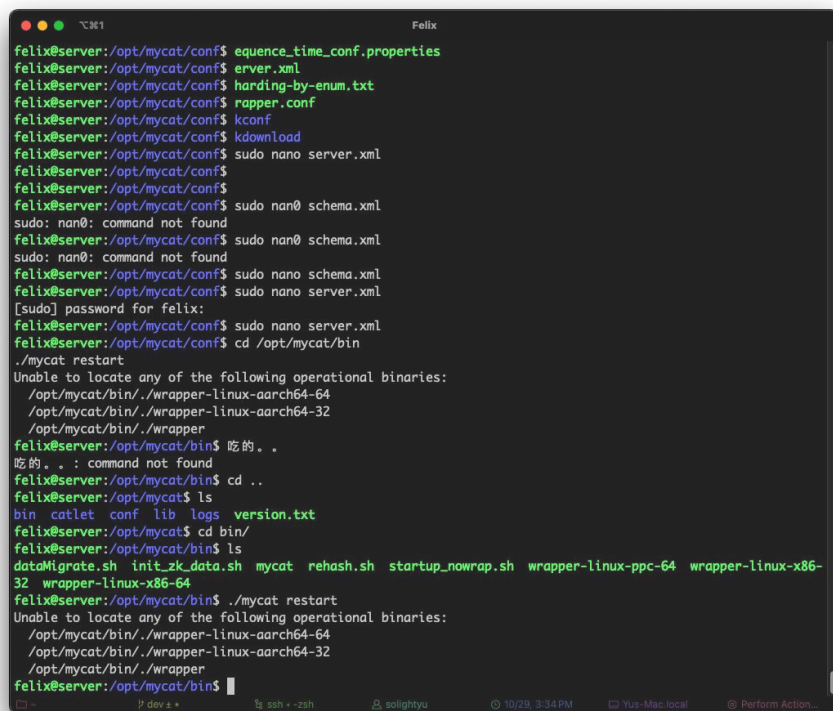
2 实验内容与过程

2.1 Mycat 数据库分库分表实验

- 1) 请给出 Mycat 配置安装过程中遇到的问题和解决方案。
Mycat 社区已经停止维护, 已经不能从官网下载。从 GitHub 仓库寻找资源下载。

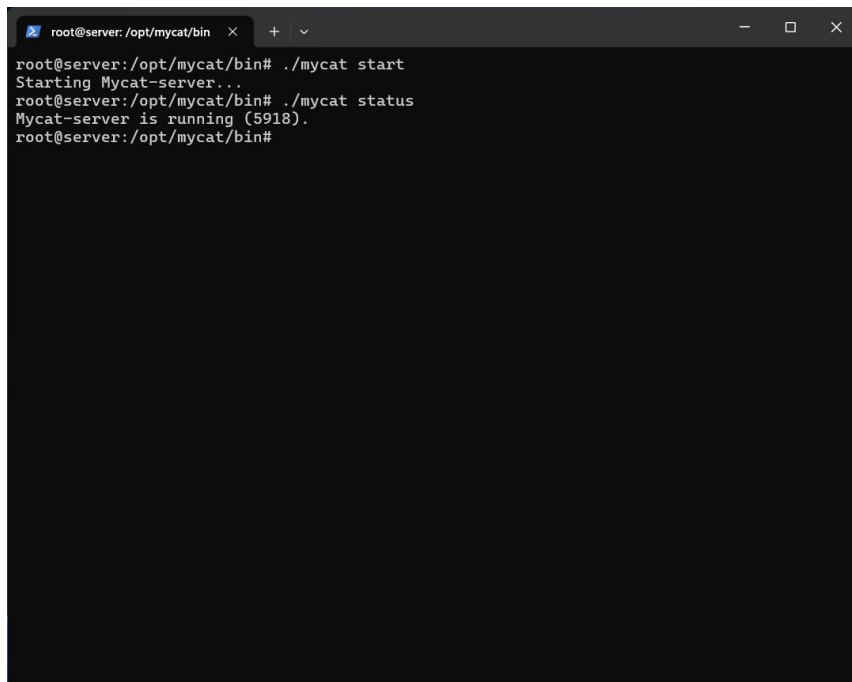


没有 ARM 的资源，只有 X86 资源，更换 x86 架构电脑实验。



```
felix@server:/opt/mycat/conf$ sequence_time_conf.properties
felix@server:/opt/mycat/conf$ server.xml
felix@server:/opt/mycat/conf$ harding-by-enum.txt
felix@server:/opt/mycat/conf$ rapper.conf
felix@server:/opt/mycat/conf$ kconf
felix@server:/opt/mycat/conf$ kdownload
felix@server:/opt/mycat/conf$ sudo nano server.xml
felix@server:/opt/mycat/conf$
felix@server:/opt/mycat/conf$ sudo nano schema.xml
sudo: nano: command not found
felix@server:/opt/mycat/conf$ sudo nano schema.xml
sudo: nano: command not found
felix@server:/opt/mycat/conf$ sudo nano schema.xml
felix@server:/opt/mycat/conf$ sudo nano server.xml
[sudo] password for felix:
felix@server:/opt/mycat/conf$ sudo nano server.xml
felix@server:/opt/mycat/conf$ cd /opt/mycat/bin
./mycat restart
Unable to locate any of the following operational binaries:
/opt/mycat/bin/./wrapper-linux-aarch64-64
/opt/mycat/bin/./wrapper-linux-aarch64-32
/opt/mycat/bin/./wrapper
felix@server:/opt/mycat/bin$ 吃的..
吃的..: command not found
felix@server:/opt/mycat/bin$ cd ..
felix@server:/opt/mycat$ ls
bin  catlet  conf  lib  logs  version.txt
felix@server:/opt/mycat$ cd bin/
felix@server:/opt/mycat/bin$ ls
dataMigrate.sh  init_zk_data.sh  mycat  rehash.sh  startup_nowrap.sh  wrapper-linux-ppc-64  wrapper-linux-x86-32  wrapper-linux-x86-64
felix@server:/opt/mycat/bin$ ./mycat restart
Unable to locate any of the following operational binaries:
/opt/mycat/bin/./wrapper-linux-aarch64-64
/opt/mycat/bin/./wrapper-linux-aarch64-32
/opt/mycat/bin/./wrapper
felix@server:/opt/mycat/bin$
```

Mycat 启动:



```
root@server:/opt/mycat/bin$ ./mycat start
Starting Mycat-server...
root@server:/opt/mycat/bin$ ./mycat status
Mycat-server is running (5918).
root@server:/opt/mycat/bin$
```

2) 请详析 Mycat 的分库分表原理和操作方法。

Mycat 的分库分表是基于分布式数据库架构的设计。

1 逻辑库和物理库的映射:

Mycat 对用户提供了逻辑上的单一数据库视图 (逻辑库), 而实际数据则分布在多个物理数据库中。应用程序只需连接 Mycat 提供的逻辑库, 具体的分库、分表细节由 Mycat 透明处理。

2 分片策略 (Sharding Strategy) :

Mycat 通过分片规则将数据分散到不同的物理数据库和表中。常用的分片策略有:

按范围分片 (Range Sharding) : 根据字段的值范围决定数据落在哪个库或表。

按哈希分片 (Hash Sharding) : 根据哈希函数计算结果来分布数据。

按时间分片 (Time-based Sharding) : 根据时间戳按月、季度或年份拆分数据。

3 路由规则 (Routing) :

在应用程序发起查询时, Mycat 根据配置好的分片规则, 将 SQL 请求路由到正确的物理数据库和表。路由规则在 schema.xml 文件中配置, 包括表名、字段和分片策略的定义。

4 分布式事务支持:

Mycat 支持两阶段提交协议, 以确保跨多个数据库实例的事务一致性。但在高并发情况下, 为避免性能开销, 推荐将事务范围控制在单个分片内。

2) 请模拟具有复杂表结构和含有较大数据量的数据库表, 并基于此库表描述分库分表的结果, 且验证分库分表的效果。

3)

使用 `mysql -uroot -p123456 -h localhost -P8066 -D TESTDB`, 使用 mycat 进行代理, 将数据写入到 TESTDB 中, mycat 根据 rule 文件的分表策略自动进行分表。

```
mysql> INSERT INTO item(id, name, price) VALUES(1, 'Sample Item', 100);
ERROR 1062 (23000): Duplicate entry '1' for key 'item.PRIMARY'
mysql> INSERT INTO item(id, name, price) VALUES(512, 'Sample Item', 100);
Query OK, 1 row affected (0.01 sec)

mysql> select * from users;
+-----+-----+-----+-----+
| id | username | email | created_at |
+-----+-----+-----+-----+
| 1 | zhangsan | NULL | 2024-11-12 17:21:21 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from items;
ERROR 1146 (42S02): Table 'TESTDB.items' doesn't exist
mysql> select * from item;
+-----+-----+-----+-----+-----+
| id | name | description | price | created_at |
+-----+-----+-----+-----+-----+
| 1 | Sample Item | NULL | 100.00 | 2024-11-12 17:22:02 |
| 512 | Sample Item | NULL | 100.00 | 2024-11-12 17:22:18 |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| TESTDB |
| db1 |
| db2 |
| db3 |
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
8 rows in set (0.00 sec)

mysql>
```

Mycat 代理，将插入的数据按照分表策略写入到 db1、db2、db3 中，有效分散存储和查询压力。

```
mysql> use db1;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from users;
+-----+-----+-----+
| id | name | indate |
+-----+-----+-----+
| 1 | zhangsan | 2024-11-12 17:28:28 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> use db2;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from item;
+-----+-----+-----+
| id | value | indate |
+-----+-----+-----+
| 1 | 100 | 2024-11-12 17:29:58 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> use db3;
ERROR 1049 (42000): Unknown database 'db3'

mysql> use db3;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

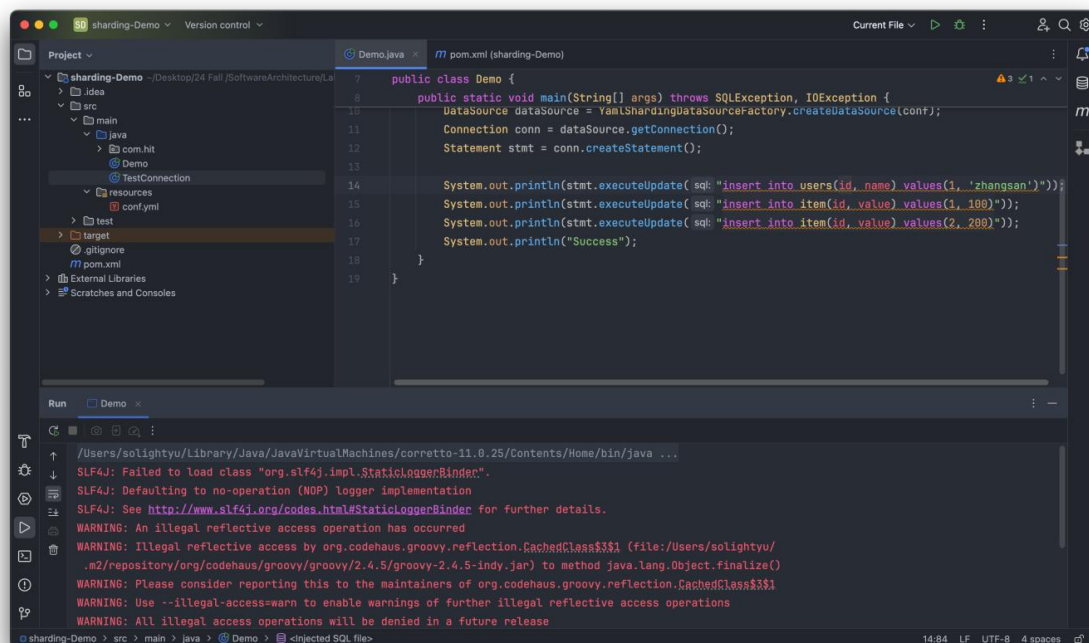
Database changed
mysql> select * from item;
+-----+-----+-----+
| id | value | indate |
+-----+-----+-----+
| 512 | 100 | 2024-11-12 17:30:24 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

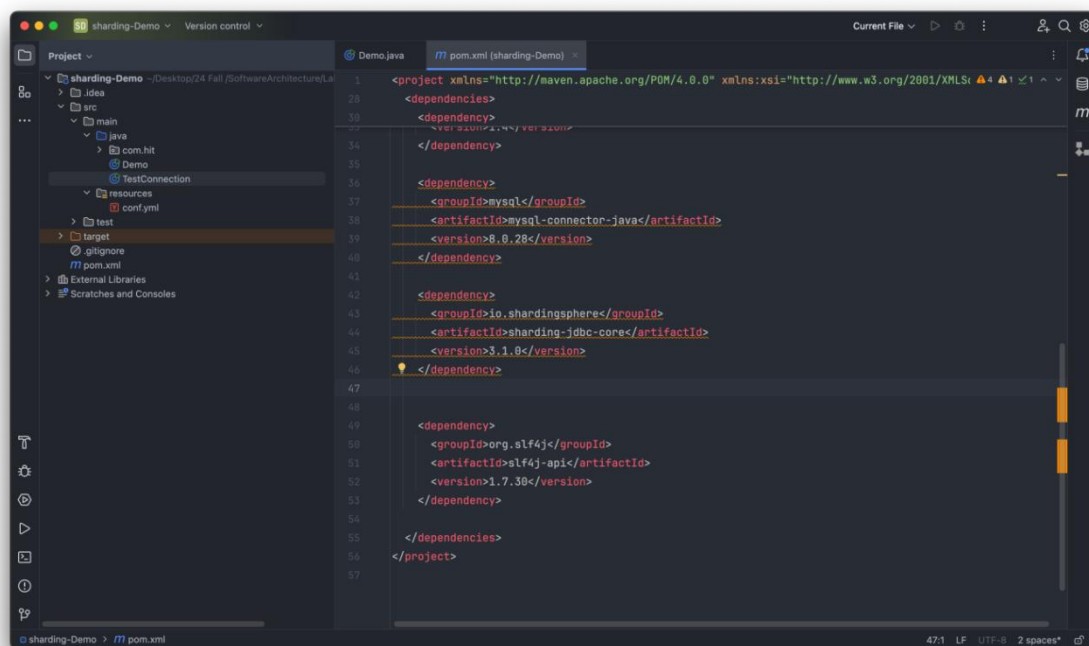
2.2 Sharding-JDBC 数据库分库分表实验

1) 请给出 Sharding-JDBC 配置安装过程中遇到的问题和解决方案。

遇到 SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".



解决方法: 在 maven 配置文件 pom.xml 中引入缺少的日志包 org.slf4j 即可解决



2) 请详析 Sharding-JDBC 的分库分表原理和操作方法。

Sharding-JDBC 属于客户端代理层，它直接嵌入在应用程序中，不需要独立部署服务。工作原理如下：

1. SQL 解析: 拦截应用程序的 SQL 请求, 根据预定义的分片规则解析 SQL 语句。
2. SQL 路由: 根据 SQL 中的查询条件, 将请求分发到具体的数据库和表。
3. 结果合并: 在数据库执行 SQL 后, 将结果集进行合并并返回给应用程序。
4. 事务管理: 支持跨库的事务管理 (分布式事务) 。

Sharding-JDBC 的分片规则 决定了如何将数据路由到不同的数据库和表。常见的分片策略包括：

1. 按范围分片: 根据某个字段的范围划分数据, 如时间区间。
2. 按取模分片: 根据某个字段的值对分片数量取模, 如 `user_id % 3`。
3. 按哈希分片: 对字段值进行哈希运算后分片。

操作方法:

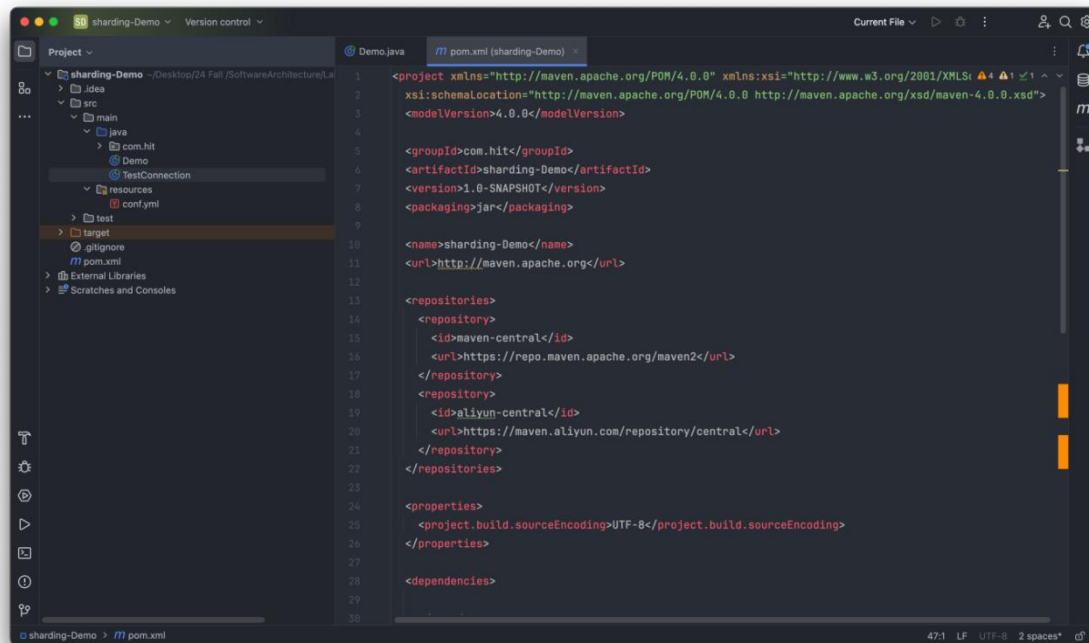
1. 登录数据库, 创建多个数据库表, 如 db1、db2、db3

```
1.CREATE DATABASE db1;
2.CREATE DATABASE db2;
3.CREATE DATABASE db3;
4.
5.USE db1;
6.CREATE TABLE users (
7.    id INT NOT NULL AUTO_INCREMENT,
8.    name VARCHAR(50) NOT NULL,
9.    PRIMARY KEY (id)
10.);
11.
12.USE db2;
13.CREATE TABLE item (
14.    id INT NOT NULL AUTO_INCREMENT,
15.    value INT NOT NULL,
16.    PRIMARY KEY (id)
17.);
18.
19.USE db3;
20.CREATE TABLE item (
21.    id INT NOT NULL AUTO_INCREMENT,
```

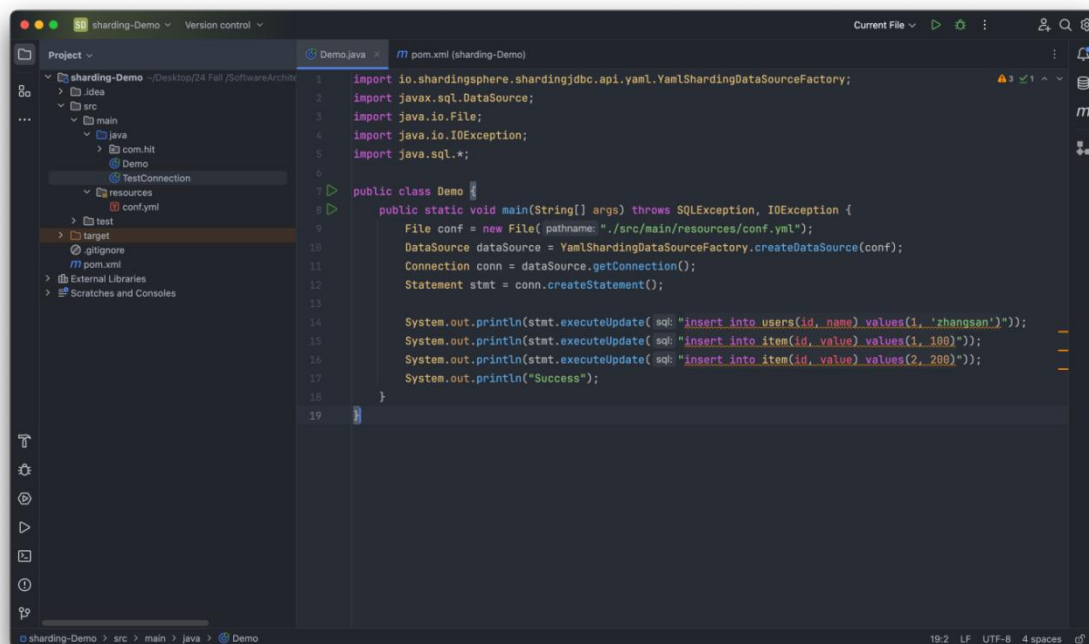


```
22.     value INT NOT NULL,  
23.     PRIMARY KEY (id)  
24. );
```

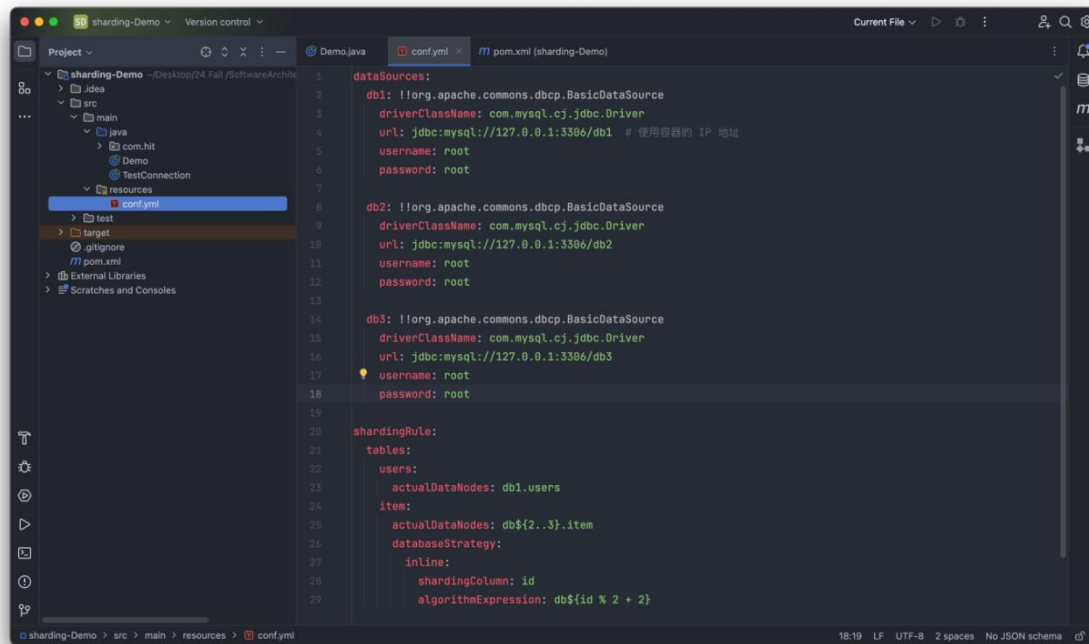
2、创建 maven 项目，并配置 pom 文件



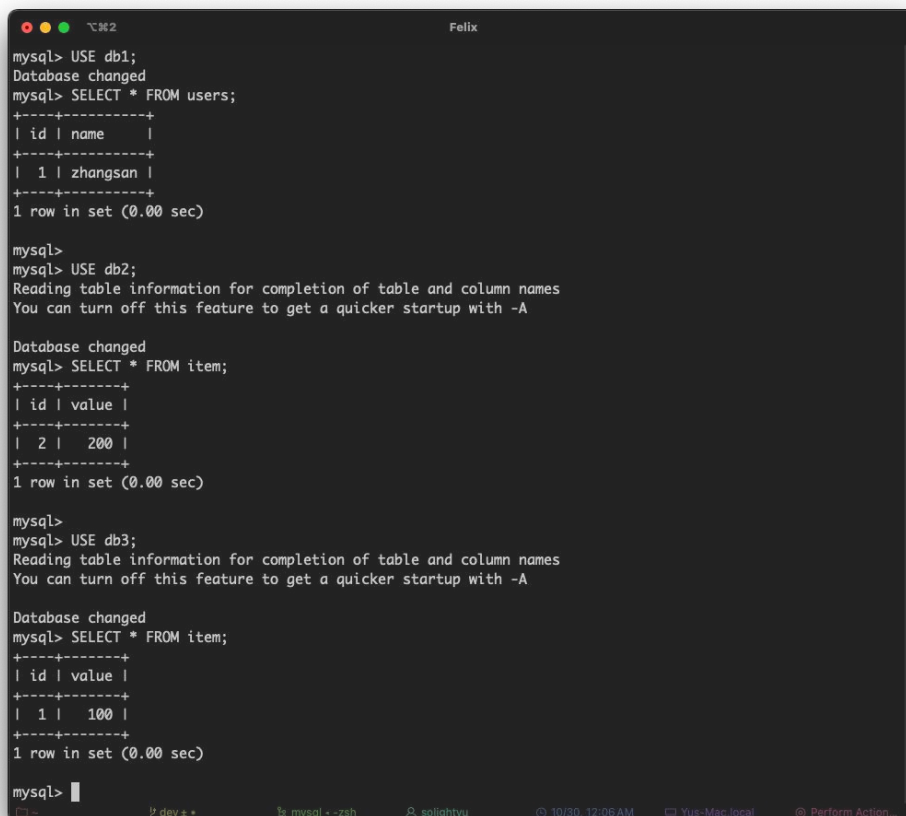
创建/src/main/java/Demo.java



编写文件/src/main/resources/conf.yml



运行 `Demo.mian()` 后再登入物理数据库检验是否插入成功



3) 请模拟具有复杂表结构和含有较大数据量的数据库表， 并基于此库表描述分库分表的结果， 且验证分库分表的效果。

电商订单管理系统表结构设计：

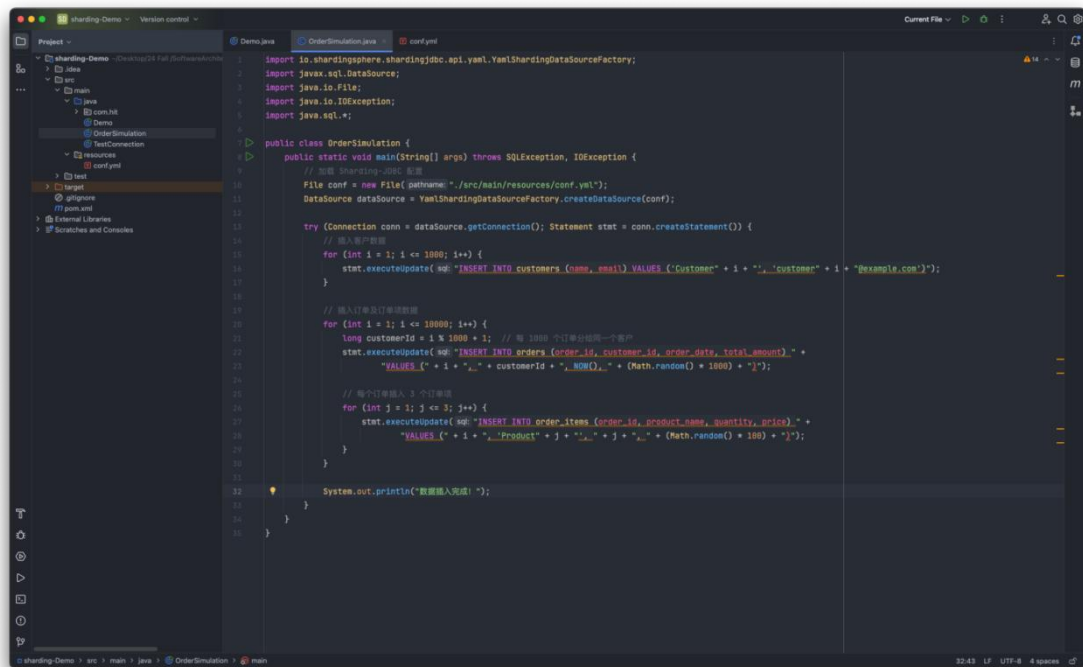
1. orders 表： 存储订单信息。
2. order_items 表： 存储订单项信息。
3. customers 表： 存储用户信息。

表之间的关系：

- 一个订单 (orders) 可能包含多个订单项 (order_items) 。
- 每个订单与一个用户 (customers) 相关联。

分库分表策略

- orders 表： 按 用户 ID (customer_id) 取模分片， 将数据分散到两个数据库 (db1 和 db2) 中。
- order_items 表： 按 订单 ID (order_id) 取模， 将订单项数据存储存在 db1 和 db2 的分片表中。
- customers 表： 集中存储在 db1 中， 因为用户表不会有大规模数据访问的场景。



```
import io.shardingsphere.shardingjdbc.api.yaml.YamlShardingDataSourceFactory;
import java.sql.DataSource;
import java.io.File;
import java.io.IOException;
import java.sql.*;

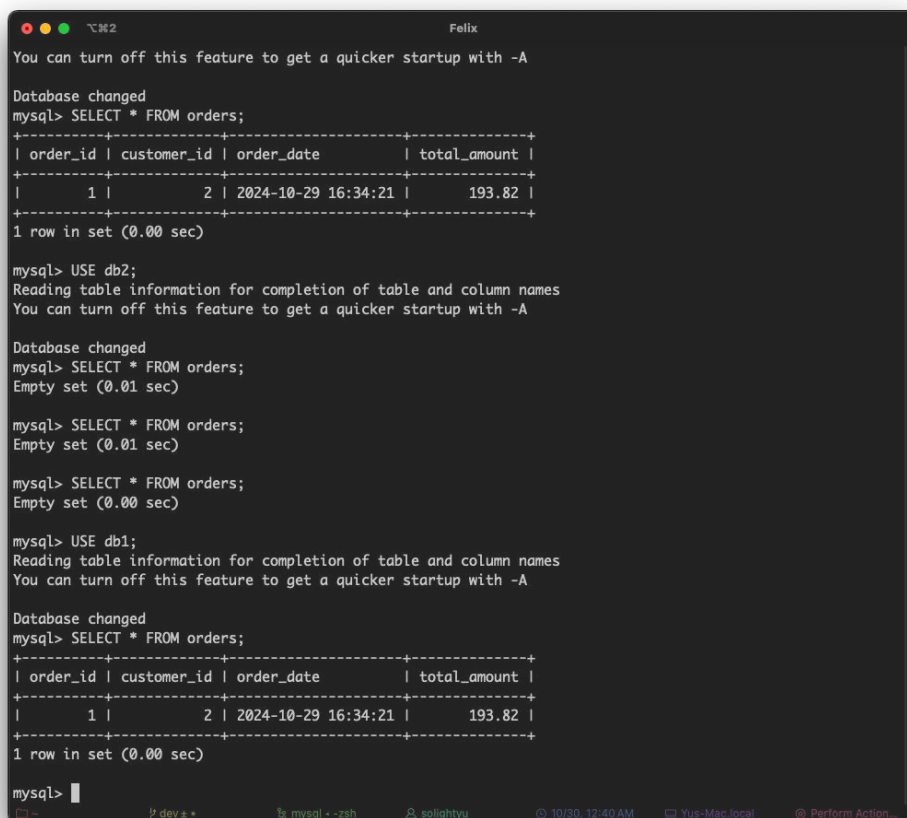
public class OrderSimulation {
    public static void main(String[] args) throws SQLException, IOException {
        // 加载 Sharding-JDBC 配置
        File conf = new File("src/main/resources/conf.yaml");
        DataSource dataSource = YamlShardingDataSourceFactory.createDataSource(conf);

        try (Connection conn = dataSource.getConnection(); Statement stmt = conn.createStatement()) {
            // 插入用户数据
            for (int i = 1; i <= 1000; i++) {
                stmt.executeUpdate("INSERT INTO customers (name, email) VALUES ('Customer' + i + ', ' + 'example.com')");
            }

            // 插入订单及订单项数据
            for (int i = 1; i <= 10000; i++) {
                long customerId = i % 1000 + 1; // 每 1000 个订单分给同一个用户
                stmt.executeUpdate("INSERT INTO orders (order_id, customer_id, order_date, total_amount) " +
                    "VALUES (" + i + ", " + customerId + ", NOW(), " + (Math.random() * 1000) + ")");

                // 每个订单插入 3 个订单项
                for (int j = 1; j <= 3; j++) {
                    stmt.executeUpdate("INSERT INTO order_items (order_id, product_name, quantity, price) " +
                        "VALUES (" + i + ", 'Product' + j + ', ' + 'example.com' + (Math.random() * 100) + ")");
                }
            }

            System.out.println("数据插入完成!");
        }
    }
}
```



```
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM orders;
+-----+-----+-----+-----+
| order_id | customer_id | order_date       | total_amount |
+-----+-----+-----+-----+
| 1 | 2 | 2024-10-29 16:34:21 | 193.82 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> USE db2;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM orders;
Empty set (0.01 sec)

mysql> SELECT * FROM orders;
Empty set (0.01 sec)

mysql> SELECT * FROM orders;
Empty set (0.00 sec)

mysql> USE db1;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM orders;
+-----+-----+-----+-----+
| order_id | customer_id | order_date       | total_amount |
+-----+-----+-----+-----+
| 1 | 2 | 2024-10-29 16:34:21 | 193.82 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

通过实验，实现了一个复杂的分库分表案例，并使用 Sharding-JDBC 对数据进行了分片管理。分库分表提升了查询效率，并支持大规模数据扩展。但是需要设计合理的分片策略，避免跨库查询带来的性能开销。

2.3 Redis 数据缓存实验

1) 请给出 Redis 配置安装过程中遇到的问题和解决方案。
使用 Homebrew 安装 Redis:

```
(base) solightyu@Yus-Mac ~ % brew --version
Homebrew 4.4.2
(base) solightyu@Yus-Mac ~ % brew install redis
--> Downloading https://formulae.brew.sh/api/formula.jws.json
--> Downloading https://formulae.brew.sh/api/cask.jws.json
--> Downloading https://ghcr.io/v2/homebrew/core/redis/manifests/7.2.6
Already downloaded: /Users/solightyu/Library/Caches/Homebrew/downloads/b0e199bdf42bce7df0d32d7a4ec09709a932a3e7f0afb45235f84fe43108f809--redis-7.2.6.bottle_manifest.json
--> Fetching redis
--> Downloading https://ghcr.io/v2/homebrew/core/redis/blobs/sha256:d842dec721e3f9cfca9d86343d0ccc009f9580803b6ee7e31f695506a282b09af0d--redis--7.2.6.arm64_sequoia.bottle.tar.gz
--> Pouring redis--7.2.6.arm64_sequoia.bottle.tar.gz
--> Caveats
To start redis now and restart at login:
  brew services start redis
Or, if you don't want/need a background service you can just run:
  /opt/homebrew/opt/redis/bin/redis-server /opt/homebrew/etc/redis.conf
--> Summary
  /opt/homebrew/Cellar/redis/7.2.6: 15 files, 2.4MB
--> Running 'brew cleanup redis'...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see 'man brew').
(base) solightyu@Yus-Mac ~ %
```

启动 Redis，并验证

```
(base) solightyu@Yus-Mac ~ % redis-server
37467:C 27 Oct 2024 23:30:01.824 * o000o000o000o Redis is starting o000o000o000o
37467:C 27 Oct 2024 23:30:01.824 * Redis version=7.2.6, bits=64, commit=00000000, modified=0, pid=37467, just started
37467:C 27 Oct 2024 23:30:01.824 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
37467:M 27 Oct 2024 23:30:01.825 * Increased maximum number of open files to 10032 (it was originally set to 256).
37467:M 27 Oct 2024 23:30:01.825 * monotonic clock: POSIX clock_gettime

Redis 7.2.6 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 37467

https://redis.io

37467:M 27 Oct 2024 23:30:01.826 # WARNING: The TCP backlog setting of 511 cannot be enforced because kern.ipc.somaxconn is set to the lower value of 128.
37467:M 27 Oct 2024 23:30:01.826 * Server initialized
37467:M 27 Oct 2024 23:30:01.827 * Loading RDB produced by version 7.2.6
37467:M 27 Oct 2024 23:30:01.827 * RDB age 861 seconds
37467:M 27 Oct 2024 23:30:01.827 * RDB memory usage when created 1.05 Mb
37467:M 27 Oct 2024 23:30:01.827 * Done loading RDB, keys loaded: 0, keys expired: 0.
37467:M 27 Oct 2024 23:30:01.827 * DB loaded from disk: 0.001 seconds
37467:M 27 Oct 2024 23:30:01.827 * Ready to accept connections tcp
```

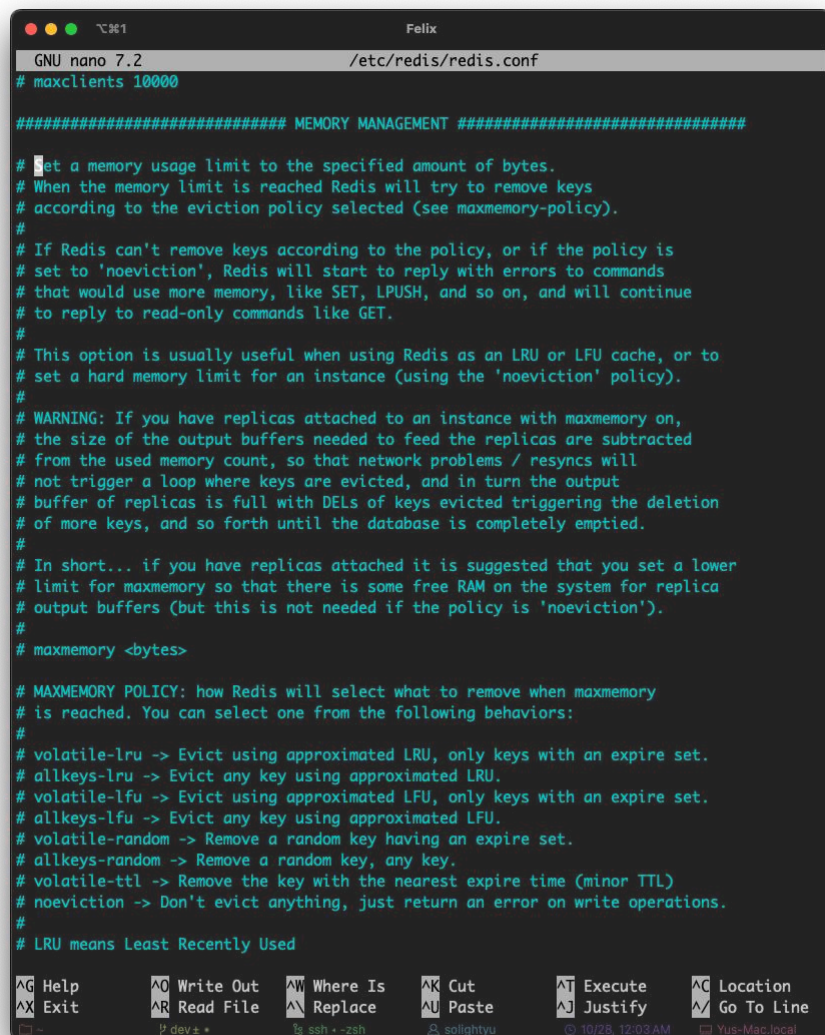
配置 Redis 文件:


```
GNU nano 7.2 /etc/redis/redis.conf
# Redis configuration file example.
#
# Note that in order to read the configuration file, Redis must be
# started with the file path as first argument:
#
# ./redis-server /path/to/redis.conf
#
# Note on units: when memory size is needed, it is possible to specify
# it in the usual form of 1k 5GB 4M and so forth:
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.
##### INCLUDES #####
# Include one or more other config files here. This is useful if you
# have a standard template that goes to all Redis servers but also need
# to customize a few per-server settings. Include files can include
# other files, so use this wisely.
#
# Note that option "include" won't be rewritten by command "CONFIG REWRITE"
# from admin or Redis Sentinel. Since Redis always uses the last processed
# line as value of a configuration directive, you'd better put includes
# at the beginning of this file to avoid overwriting config change at runtime.
#
# If instead you are interested in using includes to override configuration
# options, it is better to use include as the last line.
#
# Included paths may contain wildcards. All files matching the wildcards will
# be included in alphabetical order.
# Note that if an include path contains a wildcard but no files match it when
# the server is started, the include statement will be ignored and no error will
# be emitted. It is safe, therefore, to include wildcard files from empty
# directories.
#
# include /path/to/local.conf
# include /path/to/other.conf
# include /path/to/fragments/*.conf
#
##### MODULES #####
# Load modules at startup. If the server is not able to load modules
# it will abort. It is possible to use multiple loadmodule directives.
#
# loadmodule /path/to/my_module.so

^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location
^X Exit      ^R Read File  ^N Replace    ^U Paste      ^J Justify    ^_ Go To Line
-           P dev ± *    ssh - -zsh   solightyu     10/27, 11:57 PM  Yus-Mac.local
```

2) 请详析 Redis 的缓存清洗策略, 数据迁移及扩容策略, 面向缓存雪崩、穿透等问题的策略。

缓存清洗策略:



```
GNU nano 7.2 /etc/redis/redis.conf
# maxclients 10000

##### MEMORY MANAGEMENT #####

# Set a memory usage limit to the specified amount of bytes.
# When the memory limit is reached Redis will try to remove keys
# according to the eviction policy selected (see maxmemory-policy).
#
# If Redis can't remove keys according to the policy, or if the policy is
# set to 'noeviction', Redis will start to reply with errors to commands
# that would use more memory, like SET, LPUSH, and so on, and will continue
# to reply to read-only commands like GET.
#
# This option is usually useful when using Redis as an LRU or LFU cache, or to
# set a hard memory limit for an instance (using the 'noeviction' policy).
#
# WARNING: If you have replicas attached to an instance with maxmemory on,
# the size of the output buffers needed to feed the replicas are subtracted
# from the used memory count, so that network problems / resyncs will
# not trigger a loop where keys are evicted, and in turn the output
# buffer of replicas is full with DELs of keys evicted triggering the deletion
# of more keys, and so forth until the database is completely emptied.
#
# In short... if you have replicas attached it is suggested that you set a lower
# limit for maxmemory so that there is some free RAM on the system for replica
# output buffers (but this is not needed if the policy is 'noeviction').
#
# maxmemory <bytes>

# MAXMEMORY POLICY: how Redis will select what to remove when maxmemory
# is reached. You can select one from the following behaviors:
#
# volatile-lru -> Evict using approximated LRU, only keys with an expire set.
# allkeys-lru -> Evict any key using approximated LRU.
# volatile-lfu -> Evict using approximated LFU, only keys with an expire set.
# allkeys-lfu -> Evict any key using approximated LFU.
# volatile-random -> Remove a random key having an expire set.
# allkeys-random -> Remove a random key, any key.
# volatile-ttl -> Remove the key with the nearest expire time (minor TTL)
# noeviction -> Don't evict anything, just return an error on write operations.
#
# LRU means Least Recently Used

^G Help      ^O Write Out ^W Where Is  ^K Cut       ^J Execute  ^C Location
^X Exit      ^R Read File ^N Replace   ^U Paste     ^_ Justify  ^V Go To Line
~            ^P dev ± *  ^S ssh - -zsh  ^I solightlyu  ^T 10/28, 12:03 AM  ^Y Yus-Mac.local
```

Redis 提供两大类淘汰策略:

基于键是否设置过期时间的策略:

1. volatile-*: 仅对设置了过期时间的键进行淘汰。
2. allkeys-*: 对所有键 (无论是否有过期时间) 进行淘汰。

按算法划分的策略:

1. LRU (Least Recently Used) : 最近最少使用的键最先被淘汰。
2. LFU (Least Frequently Used) : 最少访问次数的键最先被淘汰。

3. TTL (Time to Live) : 优先淘汰即将过期的键。

4. Random: 随机淘汰一些键。

各策略的适用场景与分析

1. LRU 策略: volatile-lru / allkeys-lru

- 场景: 适合需要保存最新活跃数据, 且数据访问遵循“局部性原则”的场景 (如用户会话、热点数据)。
- 优点: 保留近期访问频繁的数据, 淘汰不常使用的数据。
- 缺点: 当缓存数据非常多时, LRU 的准确性可能受限, 需要对大量数据进行比较。

2. LFU 策略: volatile-lfu / allkeys-lfu

- 场景: 适用于一些数据长期频繁访问的情况 (如推荐系统中热门商品)。
- 优点: 避免了某些短时间被访问过的键被误判为高频访问的情况。
- 缺点: 相较于 LRU, LFU 实现和计算较为复杂, 会增加 Redis 的开销。

3. TTL 策略: volatile-ttl

- 场景: 适用于需要定期失效的缓存数据 (如秒杀活动缓存、会话数据)。
- 优点: 优先清理即将过期的数据, 确保缓存的有效性。
- 缺点: 可能会误删某些即将再次使用的数据。

4. 随机淘汰策略: volatile-random / allkeys-random

- 场景: 适用于对缓存数据不敏感的场景, 或系统性能优先级更高时使用。
- 优点: 实现简单, 不需要维护访问次数或时间的记录。
- 缺点: 淘汰策略无明确依据, 可能误删高价值数据。

5. Noeviction 策略: noeviction

- 场景: 适用于缓存数据不可丢失的情况, 如 Redis 用作数据库时。
- 优点: 确保数据不会被淘汰。
- 缺点: 当内存耗尽时, 会拒绝写入, 可能导致系统功能失效。

数据迁移及扩容策略:

Redis 的数据迁移主要分为手动迁移和工具迁移, 具体方式根据业务需求和集群架构来决定。

1. 手动迁移

手动迁移适用于数据量较小或需要单次迁移的情况。

2. 使用工具进行迁移

对于大规模数据迁移, 可以使用 Redis 官方和社区提供的工具。

Redis 的扩容分为垂直扩容和水平扩容。

1. 垂直扩容

垂直扩容是通过提升单个 Redis 实例的性能（如增加内存和 CPU）来应对更大的流量。

适用场景：数据量增长不大，但需要提升系统性能。业务架构简单，且数据一致性要求高。

缺点：

垂直扩容存在硬件限制，当单台机器性能无法再提升时，必须选择水平扩展。

2. 水平扩容

水平扩容是通过增加 Redis 实例，将数据分片存储在多个节点上，适用于大规模业务。

扩容方式：Redis Cluster（集群模式）Redis Cluster 是官方提供的分布式解决方案，将数据按照哈希分片存储在不同节点上。

3. 读写分离扩容：主从复制模式

主从复制是一种常见的扩展方案，主节点负责写操作，从节点负责读操作。

优点：

提升系统的读性能，适用于读多写少的场景。

缺点：

写操作只能由主节点处理，当写压力大时需要进一步扩展。

Redis 应对缓存雪崩的策略：

1. 设置缓存过期时间的随机化

如果所有缓存的过期时间设置为相同，那么它们会在同一时间同时失效。解决方案是对每个缓存设置一个随机的过期时间，避免缓存同时失效。

2. 数据预热

缓存预热是指在系统上线或高峰期前，将常用的数据提前加载到缓存中，避免用户请求直接打到数据库。

3. 请求限流与熔断

在高并发情况下，通过限流和熔断机制控制流量，避免瞬时请求量过大，造成数据库崩溃。

4. 缓存重建时加锁（互斥锁）

当缓存失效后，多个请求可能会同时访问数据库，从而造成缓存击穿。通过加锁机制，可以保证只有一个请求去数据库查询，其他请求等待缓存更新。

5. 双层或多层缓存

双缓存策略是指在主缓存（一级缓存）失效时，查询备用缓存（如二级缓存）以减轻数据库的压力。在复杂系统中，可以采用多层缓存策略。

Redis 应对缓存穿透的策略：

1. 缓存空结果

当数据库查询不到结果时，将空结果缓存，避免重复查询数据库。

2. 使用布隆过滤器（Bloom Filter）

布隆过滤器是一种基于概率的哈希结构，用来快速判断某个元素是否存在。Redis 在缓存层之前加上布隆过滤器，拦截数据库中一定不存在的请求。

3. 参数校验与限流

参数校验：在请求进入缓存系统之前，进行严格的参数校验，过滤掉不符合规范的请求。例如，用户 ID 必须为正整数，可以提前拦截非法请求。

限流：对于某些高频次的请求，采用限流策略，避免过多无效请求直达数据库。常用的限流算法包括令牌桶算法和漏桶算法。

4. 设置合适的过期时间

对于可能会被大量无效查询攻击的键，设置较短的过期时间。即使布隆过滤器或空结果缓存出现误判，短时间后缓存会自动失效，减少对内存的占用。

5. 采用全局唯一 ID 生成策略

在高并发系统中，如果请求参数是用户生成的 ID（如订单号、商品编号），可以采用全局唯一 ID 生成策略，确保数据一致性，避免缓存穿透。

3) 请模拟一个简单场景，实现缓存读写操作，缓存更新操作，给出缓存的效果，分析 2 问题中相关策略的效果。

假设有一个商品管理系统，商品的信息可能经常被读取，但偶尔被更新。为了提高

读取速度, 使用 Redis 作为缓存层。

缓存读取操作: 当有数据请求时, 先从缓存中读取, 如果缓存中不存在, 再从数据库中读取。

缓存写入操作: 将从数据库读取的数据写入缓存, 以便下次请求时可以直接从缓存获取。

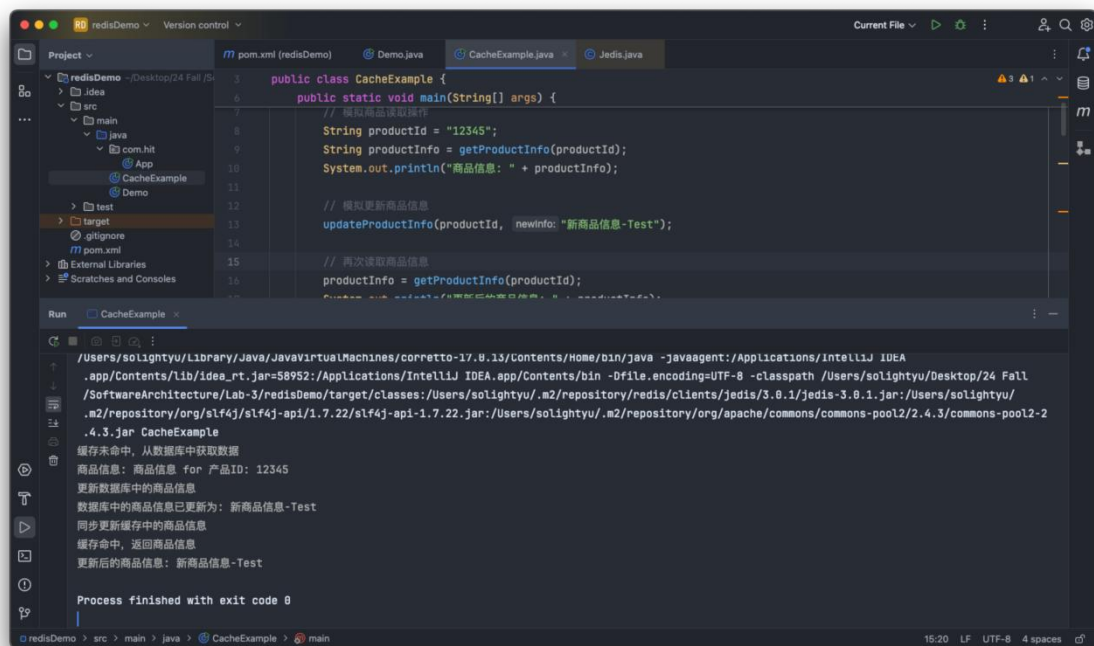
缓存更新操作: 当数据更新时, 确保缓存中的数据也随之更新。

Java 实现代码:

```
1.import redis.clients.jedis.Jedis;
2.
3.public class CacheExample {
4.    private static Jedis jedis = new Jedis("localhost", 6379, 100000);
5.
6.    public static void main(String[] args) {
7.        // 模拟商品读取操作
8.        String productId = "12345";
9.        String productInfo = getProductInfo(productId);
10.        System.out.println("商品信息: " + productInfo);
11.
12.        // 模拟更新商品信息
13.        updateProductInfo(productId, "新商品信息");
14.
15.        // 再次读取商品信息
16.        productInfo = getProductInfo(productId);
17.        System.out.println("更新后的商品信息: " + productInfo);
18.    }
19.
20.    // 从缓存或数据库中读取商品信息
21.    private static String getProductInfo(String productId) {
22.        // 尝试从缓存中获取商品信息
23.        String cacheKey = "product:" + productId;
24.        String productInfo = jedis.get(cacheKey);
25.
26.        if (productInfo != null) {
27.            // 缓存命中, 返回缓存中的数据
28.            System.out.println("缓存命中, 返回商品信息");
29.            return productInfo;
30.        } else {
```

```
31.         // 缓存未命中, 从数据库中获取数据 (此处用模拟)
32.         System.out.println("缓存未命中, 从数据库中获取数据");
33.         productInfo = queryProductInfoFromDB(productId);
34.
35.         // 将数据写入缓存, 设置过期时间为 60 秒
36.         jedis.setex(cacheKey, 60, productInfo);
37.         return productInfo;
38.     }
39. }
40.
41. // 更新商品信息并更新缓存
42. private static void updateProductInfo(String productId, String newInfo) {
43.     // 更新数据库中的商品信息 (此处用模拟)
44.     System.out.println("更新数据库中的商品信息");
45.     updateProductInfoInDB(productId, newInfo);
46.
47.     // 同步更新缓存中的商品信息
48.     String cacheKey = "product:" + productId;
49.     jedis.setex(cacheKey, 60, newInfo);
50.     System.out.println("同步更新缓存中的商品信息");
51. }
52.
53. // 模拟从数据库中查询商品信息
54. private static String queryProductInfoFromDB(String productId) {
55.     // 在真实场景中, 这里会执行数据库查询
56.     return "商品信息 for 产品 ID: " + productId;
57. }
58.
59. // 模拟更新数据库中的商品信息
60. private static void updateProductInfoInDB(String productId, String newInfo) {
61.     // 在真实场景中, 这里会执行数据库更新操作
62.     System.out.println("数据库中的商品信息已更新为: " + newInfo);
63. }
64. }
```

首次运行，缓存未命中，从数据库取数据。



```
Project: redisDemo
pom.xml (redisDemo)
Demo.java
CacheExample.java
Jedis.java

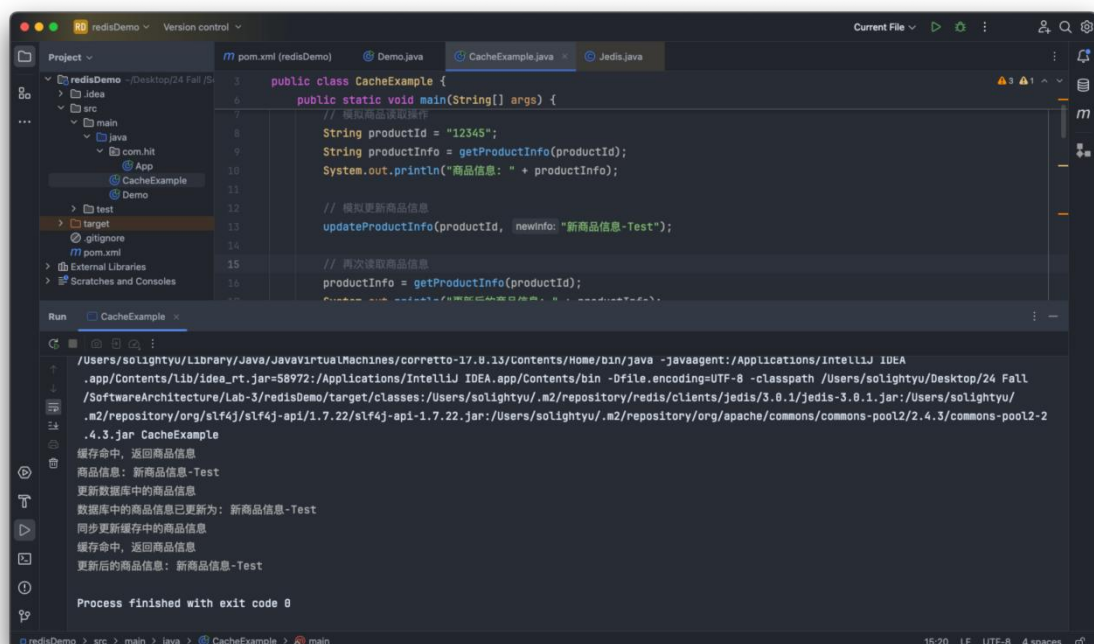
public class CacheExample {
    public static void main(String[] args) {
        // 模拟商品读取操作
        String productId = "12345";
        String productInfo = getProductInfo(productId);
        System.out.println("商品信息: " + productInfo);

        // 模拟更新商品信息
        updateProductInfo(productId, newInfo: "新商品信息-Test");

        // 再次读取商品信息
        productInfo = getProductInfo(productId);
        System.out.println("商品信息: " + productInfo);
    }
}

Run: CacheExample
/Users/solightyu/Library/Java/JavaVirtualMachines/corretto-17.0.13/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA
.app/Contents/lib/idea_rt.jar-58952:/Applications/IntelliJ IDEA.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /Users/solightyu/Desktop/24 Fall
/SoftwareArchitecture/Lab-3/redisDemo/target/classes:/Users/solightyu/.m2/repository/redis/clients/jedis/3.8.1/jedis-3.8.1.jar:/Users/solightyu/
.m2/repository/org/slf4j/slf4j-api/1.7.22/slf4j-api-1.7.22.jar:/Users/solightyu/.m2/repository/org/apache/commons/commons-pool2/2.4.3/commons-pool2-2
.4.3.jar CacheExample
缓存未命中，从数据库中获取数据
商品信息: 商品信息 for 产品ID: 12345
更新数据库中的商品信息
数据库中的商品信息已更新为: 新商品信息-Test
同步更新缓存中的商品信息
缓存命中，返回商品信息
更新后的商品信息: 新商品信息-Test
Process finished with exit code 0
```

再次运行，缓存命中，直接从缓存取出数据。



```
Project: redisDemo
pom.xml (redisDemo)
Demo.java
CacheExample.java
Jedis.java

public class CacheExample {
    public static void main(String[] args) {
        // 模拟商品读取操作
        String productId = "12345";
        String productInfo = getProductInfo(productId);
        System.out.println("商品信息: " + productInfo);

        // 模拟更新商品信息
        updateProductInfo(productId, newInfo: "新商品信息-Test");

        // 再次读取商品信息
        productInfo = getProductInfo(productId);
        System.out.println("商品信息: " + productInfo);
    }
}

Run: CacheExample
/Users/solightyu/Library/Java/JavaVirtualMachines/corretto-17.0.13/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA
.app/Contents/lib/idea_rt.jar-58972:/Applications/IntelliJ IDEA.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /Users/solightyu/Desktop/24 Fall
/SoftwareArchitecture/Lab-3/redisDemo/target/classes:/Users/solightyu/.m2/repository/redis/clients/jedis/3.8.1/jedis-3.8.1.jar:/Users/solightyu/
.m2/repository/org/slf4j/slf4j-api/1.7.22/slf4j-api-1.7.22.jar:/Users/solightyu/.m2/repository/org/apache/commons/commons-pool2/2.4.3/commons-pool2-2
.4.3.jar CacheExample
缓存命中，返回商品信息
商品信息: 新商品信息-Test
更新数据库中的商品信息
数据库中的商品信息已更新为: 新商品信息-Test
同步更新缓存中的商品信息
缓存命中，返回商品信息
更新后的商品信息: 新商品信息-Test
Process finished with exit code 0
```

3 结对开发过程记录

(1) 角色切换与任务分工

表 1-1 结对开发角色与任务分工

日期	时间(HH:MM - HH:MM)	驾驶员角色	领航员角色	本段时间的任务
10.22	16: 10-20: 30	余昊卿	马嘉良	Mycat 实验
10.23	10: 20-15: 20	马嘉良	余昊卿	Redis 实验
10.30	12: 20-16: 20	马嘉良	余昊卿	Sharding-JDBC 实验

(2) 工作日志

由领航员负责记录，记录结对开发期间的遇到的问题、两人如何通过交流合作解决每个问题的。

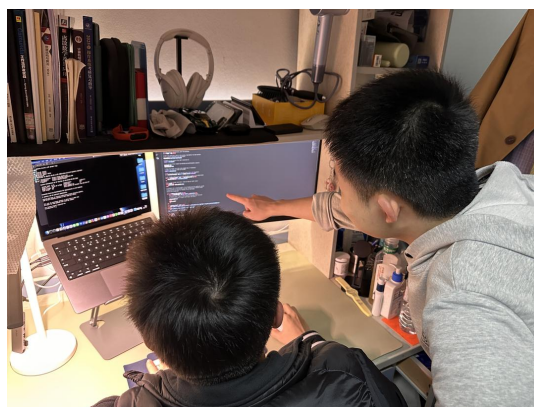
表 1-2 结对开发工作日志

日期/时间	问题描述	最终解决方法	交流过程
10.22	软件版本架构不匹配	更换 arm 版本	逐项排查错误
10.30	jdk 版本不匹配	更换 jdk	逐项排查错误

(3) 结对开发工作现场照片、或视频及文件沟通截图



结对开发现场照片 1



结对开发现场照片 2

4 实验总结

Mycat 是一个基于 Java 的开源数据库中间件，主要用于实现分库分表、读写分离等数据库分布式管理功能。它充当应用程序和数据库之间的代理层，将数据按规则分散到多个数据库节点上，帮助解决单库数据量过大和性能瓶颈问题。通过 Mycat，可以在不改动应用代码的情况下实现自动路由和查询优化，使得分布式存储和水平扩展更为便捷。此外，Mycat 支持多种数据库（如 MySQL、MariaDB）和高可用集群架构，适用于高并发、大数据量的场景，如电商和金融系统。

Sharding-JDBC 是一款轻量级的 Java 数据库分库分表框架，作为 JDBC 驱动层的增强组件，主要用于实现分布式数据库的分片、读写分离和事务管理。不同于代理层的数据库中间件，Sharding-JDBC 直接嵌入应用程序中，提供透明的分片逻辑，使应用无需修改代码即可访问多个数据库节点。它支持多种数据库（如 MySQL、PostgreSQL）和复杂的分片策略，同时提供分布式事务支持，适合微服务架构下的数据水平扩展需求。相比之下，Sharding-JDBC 更适合嵌入式和微服务场景，而 Mycat 适用于需要独立数据层的集中管理。

Redis 是一个功能强大且高效的数据存储工具，它不仅在缓存领域表现优异，还在数据共享、分布式系统等多个场景中提供了可靠的解决方案。通过学习 Redis，了解了 Redis 的缓存清洗策略，数据迁移及扩容策略，缓存雪崩、穿透等问题的策略的相关知识，对数据库中间件有了进一步的认识。我深刻体会到合理使用缓存的重要性，以及如何通过 Redis 提升系统的整体性能与响应速度。

5 教师评语	