目　　录

1 概述

1.1 选题背景

随着云计算和大数据技术的不断发展，现代 Web 应用系统在数据规模、高并发访问和高可用性方面面临更高的要求。传统的单机式数据库架构难以支撑企业级业务，分布式数据库集群以及面向海量数据的分析技术逐渐成为解决此类问题的重要方向。

在数据库领域，集群化已成为主流方案。通过主备复制和读写分离等机制，系统能够在提升吞吐能力的同时保持高可用性，从而保证服务在故障场景下仍能持续运行。随着自主可控技术的推进，openGauss 与 GaussDB 等国产数据库在金融、政务等关键行业得到广泛应用，其在性能、安全和兼容性方面表现稳定，为国产化替代提供了可靠支撑。

在数据处理方面，分布式计算框架的发展显著提升了海量数据的分析能力。Apache Spark 作为内存计算引擎，能够对大规模数据进行高效处理，支持批处理与流处理。同时，针对轻量级部署场景，可采用 Spark local 模式实现单机内嵌式分析，并通过 SQL 降级方案保证系统鲁棒性。

在系统架构层面，微服务化趋势日益明显。前后端分离的开发模式提升了开发效率和系统扩展性，而容器化技术使应用在部署、运维和迁移方面更加灵活，为云原生架构的构建提供了良好基础。

1.2 选题意义

本项目选取构建一个具有朋友圈特征的博客系统，并结合 GaussDB 一主二备集群与 Spark 大数据分析平台，其研究和实践价值体现在技术与应用两个层面。

在技术层面，本项目能够帮助深入理解国产数据库体系的关键机制，包括主备流复制、WAL 日志传输和读写分离等内容。通过部署数据库集群与测试故障转移、负载均衡等场景，可系统掌握分布式系统的基本设计方法。在数据处理方面，借助 Spark local 模式完成用户行为统计与内容热度分析，并通过 SQL 降级方案保证系统可用性，有助于理解数据分析架构的设计权衡。此外，项目涵盖前端（Vue 3）、后端（Spring Boot）、数据库管理和数据分析等部分，能够提升完整的工程化开发能力。

在应用层面，系统基于社交网络的典型场景设计，包括时间线展示、好友关

系管理与互动功能，贴近实际业务需求。通过读写分离的体系结构，系统具备较好的并发处理能力，能够支持大量用户访问。用户行为统计与内容热度分析等功能为业务提供数据支撑，满足典型的数据分析需求。同时，采用的集群化架构具备良好的可扩展性，能够在业务规模增长时平稳扩展系统能力。

1.3 实验目标

本实验旨在构建一个具有朋友圈特征的博客系统，实现 GaussDB 一主二备集群的部署与管理，并完成 Spark 大数据分析能力的集成。通过对分布式数据库高可用性、读写分离机制以及数据分析流程的验证，全面提升对分布式系统架构的理解和应用能力。

1.4 实验内容

1. 在 openEuler 虚拟机（10.211.55.11）上采用 Docker Compose 实现数据库集群的容器化部署。由于 openGauss 官方容器镜像部署复杂度较高，项目采用 PostgreSQL 13 Alpine 作为数据库引擎，通过流复制机制构建一主二备架构，有效模拟 GaussDB 集群的核心功能。集群包含主库（端口 5432）与两个备库（端口 5434、5436），重点实现了物理复制槽、异步流复制等机制，并解决了容器网络配置、版本兼容性等问题，确保高可用架构的稳定运行。
2. 在后端应用中实现基于 AOP 切面与 ThreadLocal 上下文的动态数据源路由机制，通过自定义 `@ReadOnly` 注解标记只读方法，自动将查询操作路由至备库。利用 HikariCP 为主库与备库分别配置独立连接池，优化连接复用效率。通过 `AbstractRoutingDataSource` 实现透明的数据源切换，提升系统在高并发环境下的处理能力。
3. 集成 Spark 3.5.0 内嵌式分析引擎（local[*] 模式），通过 JDBC 连接器读取 GaussDB 访问日志表。基于项目业务场景实现用户行为分析任务，包括发文量统计、浏览频次统计和评论统计，并将分析结果回写至数据库。
4. 实现从用户管理到内容互动的完整业务流程，包括用户注册与登录（基于 JWT）、文章发布、评论、点赞、好友关系管理等功能。同时构建朋友圈式时间线，并加入基础的数据统计与可视化能力。
5. 开发用于集群生命周期管理的自动化脚本工具，包括 `refactor-ports.sh`

（幂等性端口重构）、`verify-gaussdb-cluster.sh`（主备复制状态验证）、`start-all.sh` 与 `stop-all.sh`（依赖顺序启停）等。脚本具备幂等性设计，支持重复执行而不破坏配置；通过 SQL 探针实现主备数据同步的闭环验证；采用依赖拓扑排序保证服务启动顺序。最终形成完整的运维文档，实现集群部署与维护的自动化。

2 系统设计与实现

2.1 整体架构

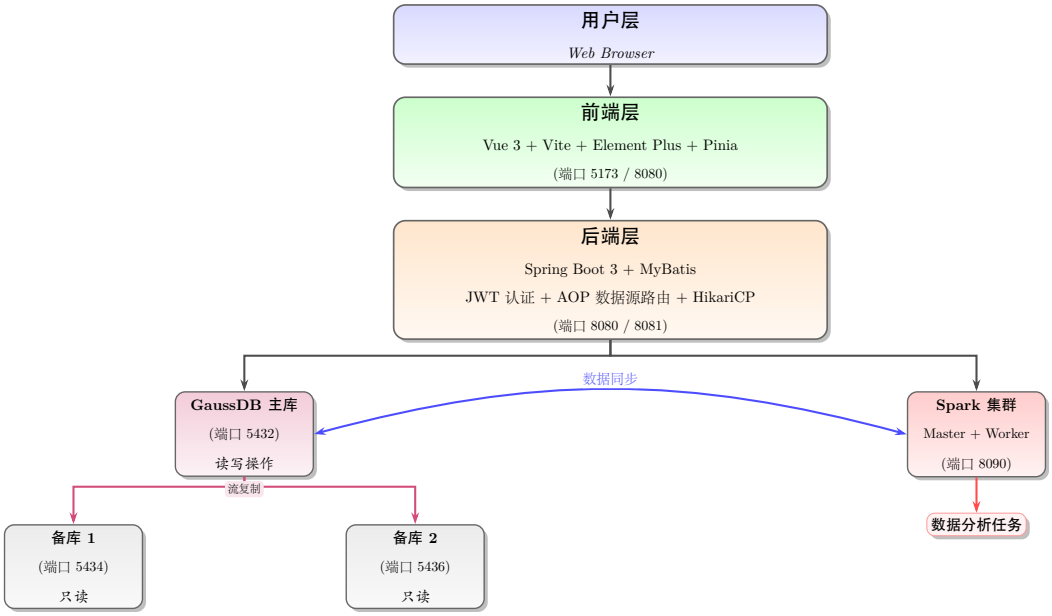


图 1: 系统整体架构图

2.2 数据流向

本系统采用读写分离架构，不同类型的请求遵循不同的数据流向，具体如下：

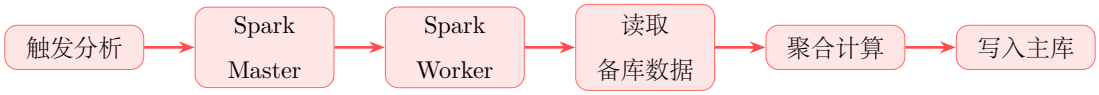
1. 写操作流程

写操作统一路由至主库，确保数据一致性。主库通过流复制机制将数据同步至备库 1 和备库 2，实现数据的高可用性。

2. 读操作流程

读操作通过 AOP 切面识别后，路由至备库集群。系统采用轮询策略在备库 1（端口 5434）和备库 2（端口 5436）之间进行负载均衡，有效分散查询压力，提升系统并发处理能力。

3. 数据分析流程



Spark 分析任务由 Master 节点调度至 Worker 节点执行。Worker 从备库读取历史数据进行统计分析，避免对主库造成性能影响。分析结果经聚合计算后写回主库，供业务系统使用。

2.3 系统技术选型

本系统采用现代化的技术栈，结合国产数据库、主流开发框架和大数据分析平台，技术选型如表 1所示。

表 1: 系统技术选型

技术组件	选型
操作系统	openEuler 22.03 LTS
数据库	GaussDB 9.2.4
后端框架	Spring Boot 3.1.5
前端框架	Vue 3 + Vite
大数据处理	Apache Spark 3.5.0
容器化	Docker + Docker Compose
持久层框架	MyBatis 3.0.3
连接池	HikariCP
认证机制	JWT (JSON Web Token)

2.4 功能模块设计

本系统采用经典的分层架构设计，从上至下依次为前端展示层、后端控制层、业务逻辑层、数据访问层和数据持久层，各层职责明确、低耦合高内聚。系统核心功能包括用户认证、文章管理、社交互动和数据统计四大模块，分层架构如图 2所示。

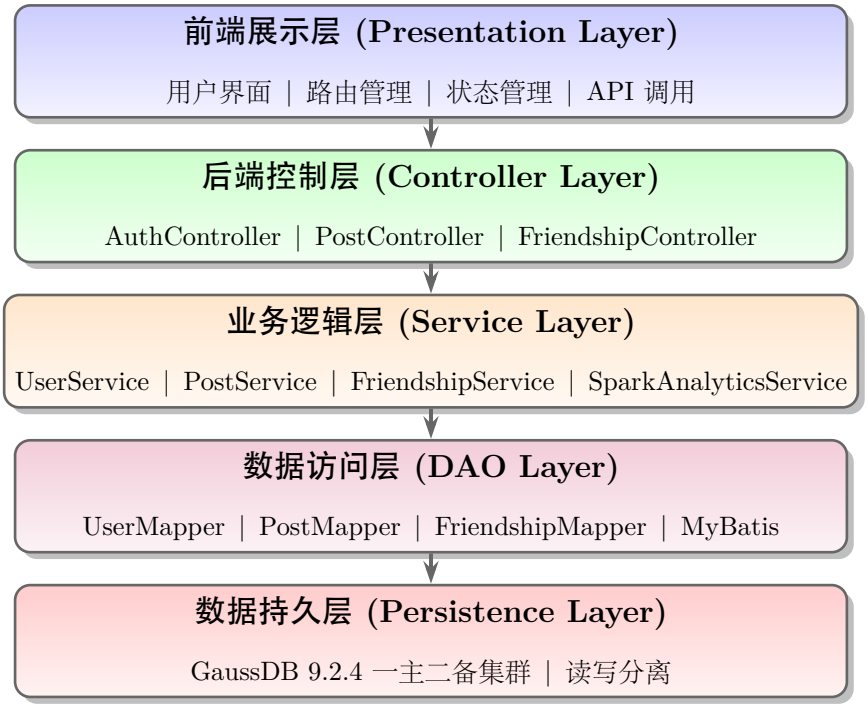


图 2: 系统分层架构

各层职责说明：

前端展示层负责用户交互和页面渲染，采用 Vue 3 组件化设计，使用 Pinia 进行状态管理，通过 Axios 与后端 API 通信。

后端控制层接收并处理前端请求，进行参数验证和权限校验，调用业务逻辑层完成具体功能，返回标准化的 JSON 响应。

业务逻辑层封装核心业务逻辑，包括用户认证、文章发布与管理、好友关系处理等功能。SparkAnalyticsService 负责触发大数据分析任务。

数据访问层通过 MyBatis 实现数据持久化操作，使用 AOP 切面实现读写分离路由，将写操作导向主库，读操作分发至备库。

数据持久层采用 GaussDB 一主二备集群架构，通过流复制机制保证数据一致性，提供高可用性和容灾能力。

2.5 GaussDB 集群部署

2.5.1 环境准备

本实验在虚拟机环境中搭建 openGauss 分布式数据库集群，运行平台为 openEuler 22.03 LTS。虚拟机的基础配置包括固定 IP 地址 10.211.55.11、4GB 以上内存、50GB 以上磁盘空间，并以 omm 用户作为数据库专用管理账户。数

据库软件版本为 openGauss 9.2.4，安装路径设置为 /usr/local/opengauss。

集群包含一个主库与两个备库，各实例采用独立的数据目录与端口规划。openGauss 的每个实例均由主端口与 HA（High Availability）端口组成，为避免冲突，本实验采用间隔端口设计。具体配置如下：

表 2: GaussDB 集群实例配置

实例	主端口	HA 端口	数据目录
主库	5432	5433	data_primary
备库 1	5434	5435	data_standby1
备库 2	5436	5437	data_standby2

注：所有数据目录位于 /usr/local/opengauss/ 下

2.5.2 部署步骤

步骤 1：初始化主库

主库初始化是集群部署的第一步，需要使用 gs_initdb 命令创建数据目录并生成配置文件。初始化过程会自动创建系统表、设置默认编码（UTF-8）、配置超级用户密码等。--nodename 参数用于标识节点名称，便于在集群中区分不同实例。

```
gs_initdb -D /usr/local/opengauss/data_primary \
  --nodename=primary -w <password>
```

Listing 1: 主库初始化命令

初始化完成后，需要修改 postgresql.conf 和 pg_hba.conf 配置文件以支持流复制。主库关键参数配置如表 3所示。

将 wal_level 设置为 replica 是实现流复制的基础，它确保了预写式日志包含足够的信息以支持备库的数据重放。同时，开启 hot_standby = on 是实现读写分离架构的关键，它允许备库在恢复模式下依然能够接受只读查询请求，从而分担主库的负载。

由于本实验在单台虚拟机上模拟集群环境，端口冲突是主要挑战。为此，采用间隔端口策略（5432/5434/5436）不仅规避了监听冲突，也为各实例的 HA（高可用）通信端口预留了空间。在实际生产环境中，这种多实例部署常用于提升单机资源利用率或进行开发测试。

表 3: GaussDB 主库关键参数配置

参数名	配置值	说明
port	5432	主库服务端口
wal_level	replica	开启归档模式，支持流复制
max_wal_senders	10	允许的最大复制连接数
max_connections	200	最大客户端连接数

步骤 2：创建复制用户

主备复制需要专用的复制用户，该用户具有 REPLICATION 权限，用于备库连接主库并接收 WAL 日志流。同时需要创建业务数据库和应用用户，并授予相应权限。这些操作需要在主库上执行，备库会通过复制自动同步这些元数据。

```
CREATE USER replicator WITH REPLICATION PASSWORD '<password>';
CREATE DATABASE blog_db;
CREATE USER bloguser WITH PASSWORD '<password>';
GRANT ALL PRIVILEGES ON DATABASE blog_db TO bloguser;
```

Listing 2: 创建数据库用户

此外，还需要在 pg_hba.conf 中添加复制用户的访问控制规则，允许备库从指定 IP 地址连接主库。

步骤 3：初始化备库

备库初始化采用基础备份（Base Backup）方式，通过 gs_basebackup 工具从主库全量复制数据。-X stream 参数表示在复制过程中同步传输 WAL 日志，确保备份数据的一致性。-P 参数显示进度信息，便于监控备份过程。

```
# 从主库同步数据到备库1
gs_basebackup -h 127.0.0.1 -p 5432 -U replicator \
  -D /usr/local/opengauss/data_standby1 -X stream -P

# 配置备库端口和复制连接
echo "port = 5434" >> data_standby1/postgresql.conf
echo "primary_conninfo = 'host=127.0.0.1 port=5432 user=replicator'" \
  >> data_standby1/postgresql.auto.conf

# 启动备库
gs_ctl start -D /usr/local/opengauss/data_standby1
```

Listing 3: 备库初始化命令

备份完成后，需要修改备库的端口配置以避免与主库冲突，并设置 `primary_conninfo` 参数指向主库。备库 2 采用相同流程，端口配置为 5436。两个备库独立连接主库，形成一主二备的高可用架构。

步骤 4：验证集群状态

使用自动化脚本验证集群运行状态（脚本详见附录 A）：

```
./scripts/verify-gaussdb-cluster.sh
```

Listing 4: 集群验证

2.6 读写分离实现

2.6.1 数据源配置

读写分离的实现首先需要为主库和备库分别配置独立的数据源。本系统采用 HikariCP 作为连接池，其高性能特性能够有效降低连接建立开销。主库和备库均配置 20 个最大连接数，根据实际负载情况可调整。`connection-test-query` 用于在获取连接时验证有效性，防止使用已断开的连接。

完整配置文件请参考附录 C，其中包含了主库和备库的详细连接参数、HikariCP 连接池配置、超时设置等。配置文件位于 `backend/src/main/resources/application-gaussd`

2.6.2 动态数据源路由

本系统采用面向切面编程（AOP）技术实现无侵入式的读写分离。具体流程如下：

1. **注解标记**：自定义 `@ReadOnly` 注解用于标记业务层中的查询方法。
2. **切面拦截**：`DataSourceAspect` 切面在运行时拦截所有方法调用。利用反射机制检查目标方法是否携带该注解。
3. **上下文切换**：若检测到注解，切面将当前线程的 `ThreadLocal` 上下文标记为 `REPLICA`（备库）；否则默认为 `PRIMARY`（主库）。
4. **动态路由**：Spring 的 `AbstractRoutingDataSource` 在获取数据库连接时，会读取当前线程的上下文标记，从而从对应的 HikariCP 连接池中获取连接。

5. **资源清理**：在方法执行完毕后，通过 `finally` 块强制清除 `ThreadLocal` 变量，防止线程复用导致的数据源混乱（内存泄漏风险）。

这种设计不侵入业务代码，开发人员只需在需要路由至备库的方法上添加注解即可，无需修改任何数据访问逻辑。

其核心机制包括：

DynamicRoutingDataSource：继承 `AbstractRoutingDataSource`，根据 `ThreadLocal` 上下文返回当前数据源键。

DataSourceContextHolder：使用 `ThreadLocal` 存储当前线程的数据源类型（PRIMARY/REPLICA）。

DataSourceAspect：AOP 切面拦截 `@ReadOnly` 注解，自动切换数据源。

核心代码：DataSourceAspect

`DataSourceAspect` 是读写分离的核心组件，其工作原理如下：切面通过 `@Around` 注解拦截所有带有 `@ReadOnly` 标记的方法，利用反射机制获取方法签名并检查注解存在性。如果检测到 `@ReadOnly` 注解，则调用 `DataSourceContextHolder.setDataSour` 将当前线程的数据源上下文设置为备库；否则默认使用主库。业务方法执行完毕后，`finally` 块确保清理 `ThreadLocal` 变量，防止线程池复用时出现数据源混乱。

完整代码请参考 `backend/src/main/java/com/cloudcom/blog/aspect/DataSourceAspect.` 其中包含了完整的切面逻辑、异常处理和日志记录。

2.6.3 使用示例

在业务层使用 `@ReadOnly` 注解非常简单，只需在查询方法上添加该注解即可。例如，`PostService` 中的 `getAllPosts()` 方法标记为 `@ReadOnly`，系统会自动将该查询路由至备库执行；而 `createPost()` 方法没有注解，默认路由至主库执行写操作。这种设计使得开发人员无需关心底层数据源切换逻辑，只需按照业务语义添加注解即可。

完整示例代码请参考 `backend/src/main/java/com/cloudcom/blog/service/PostService.` 其中包含了多个读写操作的实际应用场景。

2.7 Spark 大数据分析

本系统采用 Spark 内嵌模式（local[*]），将 Spark 引擎集成在 Spring Boot 后端服务中，无需部署独立集群。通过 Maven 引入 `spark-core` 和 `spark-sql`

依赖（版本 3.5.0）。

2.7.1 数据分析任务

SparkAnalyticsService 核心逻辑：

SparkAnalyticsService 实现了双层降级策略的数据分析功能。服务首先尝试使用 Spark 进行分析：创建 SparkSession 并配置为 local[*] 模式，通过 JDBC 连接器读取 access_logs 表数据。利用 Spark DataFrame API 的 filter、groupBy、count 等算子执行分布式聚合计算，分别统计用户发文量、文章浏览量和评论数量。计算完成后，通过 collectAsList() 收集结果并回写至数据库。

当 Spark 分析失败时（如 Java 17+ 兼容性问题），系统自动调用 runSqlAnalytics() 方法，直接从业务表中执行 SQL 聚合查询。该方法通过 MyBatis Mapper 调用预定义的 SQL 语句，从 posts、comments 等表中统计数据，确保分析功能始终可用。

完整代码请参考 backend/src/main/java/com/cloudcom/blog/service/SparkAnalyticsSe
其中包含了 Spark 配置、异常处理、SQL 降级逻辑等完整实现。

数据分析模块采用双层降级策略，优先尝试 Spark 分析，失败后自动回退至 SQL 聚合查询。Spark 分析流程采用“提取-转换-加载”（ETL）模式：

1. **数据摄取（Ingestion）**：通过 JDBC 连接器从 GaussDB 的 access_logs 表中全量拉取用户行为日志。采用 local[*] 模式并行读取，最大化利用 CPU 多核资源。
2. **转换计算（Transformation）**：利用 Spark SQL 的 DataFrame API 进行链式处理。通过 filter 算子按操作类型筛选数据，利用 groupBy 和 count 聚合算子，分别从用户维度与内容维度进行统计。
3. **结果回写（Loading）**：将计算生成的 Dataset 映射为 Statistic 实体，通过 UPSERT 操作回写至 statistics 表。

SQL 降级方案：当 Spark 分析失败时，SparkAnalyticsService 自动调用 runSqlAnalytics() 方法，直接从 posts、comments 等业务表中执行 SQL 聚合查询，计算用户发文量、文章浏览量和评论数量。该方案保证了分析功能的可用性，避免因 Spark 依赖问题导致系统不可用。

这种设计实现了分析业务与核心事务业务的解耦，避免了复杂聚合查询对主库 OLTP 性能的影响。Spark 的内存计算模型在处理大规模数据时具有性能优势，而 SQL 降级方案则保证了系统的鲁棒性。

2.7.2 统计结果查询

`StatisticsController` 提供 REST API 接口供前端查询统计数据，主要接口包括：

`POST /api/stats/analyze` - 触发 Spark 分析任务

`GET /api/stats` - 获取统计汇总（聚合 + 明细）

`GET /api/stats/aggregated` - 获取聚合统计数据

`GET /api/stats/list` - 获取全部统计明细

2.8 应用功能实现

系统实现了完整的业务功能，包括用户认证、文章管理、好友关系等模块。各模块通过 RESTful API 对外提供服务，核心接口如表 4和表 5所示。

后端接口遵循 RESTful 设计规范，统一采用 JSON 格式进行数据交互。在安全性方面，系统实现了无状态的认证机制：

用户登录成功后，服务器生成包含用户 ID 和过期时间的 JSON Web Token (JWT)。前端将 Token 存储在本地，并在后续所有请求的 Header 中携带该令牌。后端通过拦截器校验 Token 的签名合法性，从而实现身份鉴权，避免了传统 Session 模式在集群环境下的状态同步问题。

利用 `@ControllerAdvice` 全局捕获业务异常，将系统内部错误转换为标准化的 HTTP 状态码和错误提示，提升了系统的健壮性和前端交互体验。

系统采用经典的 MVC 三层架构，Controller 层负责请求路由和参数校验，Service 层封装业务逻辑，Mapper 层处理数据持久化。这种分层设计使得代码结构清晰、职责分明，便于后期维护和功能扩展。

表 4: 系统核心 API 接口设计（一）

功能模块	方法	API 路径	功能说明
认证模块 (/api/auth)			
	POST	`/register`	用户注册
	POST	`/login`	用户登录，返回 JWT Token
用户模块 (/api/users)			
	GET	`/me`	获取当前用户信息
	PUT	`/me`	更新个人资料
	GET	`/{id}`	获取指定用户信息
文章模块 (/api/posts)			
	GET	`/list`	获取文章列表（路由至备库）
	GET	`/{id}/detail`	获取文章详情
	GET	`/timeline`	获取好友时间线
	POST	`/`	创建文章（路由至主库）
	PUT	`/{id}`	更新文章
	DELETE	`/{id}`	删除文章
评论模块 (/api/comments)			
	GET	`/post/{postId}`	获取文章评论列表
	POST	`/`	发表评论
	PUT	`/{id}`	更新评论
	DELETE	`/{id}`	删除评论
点赞模块 (/api/likes)			
	POST	`/post/{postId}`	点赞文章
	DELETE	`/post/{postId}`	取消点赞
	GET	`/post/{postId}/check`	检查是否已点赞

表 5: 系统核心 API 接口设计（二）

功能模块	方法	API 路径	功能说明
好友模块 (/api/friends)			
	POST	`/request/{receiverId}`	发送好友请求
	POST	`/accept/{requestId}`	接受好友请求
	POST	`/reject/{requestId}`	拒绝好友请求
	DELETE	`/user/{friendUserId}`	删除好友
	GET	`/list`	获取好友列表
	GET	`/requests`	获取待处理请求
	GET	`/search?keyword=xxx`	搜索用户
	GET	`/status/{userId}`	检查好友关系状态
统计模块 (/api/stats)			
	POST	`/analyze`	触发 Spark 数据分析
	GET	`/`	获取统计汇总数据
	GET	`/{type}`	获取指定类型统计
上传模块 (/api/upload)			
	POST	`/avatar`	上传用户头像
	POST	`/cover`	上传文章封面图
	POST	`/image`	上传文章内容图片

认证机制：除注册和登录接口外，其他接口均需在请求头中携带 `Authorization: Bearer {token}` 进行身份验证。所有读操作通过 `@ReadOnly` 注解自动路由至备库，写操作路由至主库，实现读写分离。

2.9 集群管理与运维

为保证集群的稳定运行，项目开发了一套基于 Shell 的自动化运维工具集，实现集群生命周期管理、状态监控和故障诊断。其核心设计思路包含：

1. **幂等性端口重构**：`refactor-ports.sh` 脚本采用幂等性设计，支持重复执行而不破坏配置。执行流程：（1）通过 `pkill -9 gaussdb` 强制停止所有实例，清理 Unix Socket 文件与 PID 文件；（2）使用 `sed -i '/^port/d'` 删除所有端口配置行，避免配置重复；（3）追加新端口配置（主库 5432、备库 1 5434、备库 2 5436）；（4）依次启动三个实例，每次启动后等待 5 秒以确保服务就绪；（5）通过 `lsof -i:PORT` 验证端口监听状态。

2. **深层健康检查**：`verify-gaussdb-cluster.sh` 实现了九项检查机制：（1）主库连接性测试（`gsql -c 'SELECT version()'`）；（2）备库恢复模式验证（`SELECT pg_is_in_recovery()`）；（3）主库复制连接状态（`pg_stat_replication`）；（4）SQL 探针测试：在主库插入测试数据，等待 2 秒后在两个备库查询，验证数据同步完整性；（5）复制延迟统计（`write_lag`、`flush_lag`、`replay_lag`）。这种主动探测机制能够发现进程存活但复制失败的隐蔽故障。

3. **一键启停控制**：`start-all.sh` 采用依赖拓扑排序，依次启动 GaussDB 集群 → Spring Boot 后端 → Nginx 前端。每个组件启动后执行健康检查：数据库通过 `gsql` 测试连接，后端通过 `curl /actuator/health` 检查状态，前端通过 `curl -I` 验证 HTTP 响应。`stop-all.sh` 按相反顺序停止服务，先停止前端避免新请求，再停止后端处理存量请求，最后停止数据库保证数据安全。

4. **全局状态监控**：`status-all.sh` 提供一站式状态查看功能，统一展示：（1）进程存活性（`ps aux | grep gaussdb`）；（2）端口监听状态（`lsof -i:5432/5434/5436`）；（3）复制连接数量（`pg_stat_replication`）；（4）后端服务健康（`/actuator/health`）；（5）系统资源使用（CPU、内存、磁盘）。

（完整脚本代码详见附录 A）

3 应用部署

3.1 部署架构设计

3.1.1 部署拓扑

本实验采用虚拟机部署，模拟生产环境的完整架构。部署拓扑如下：

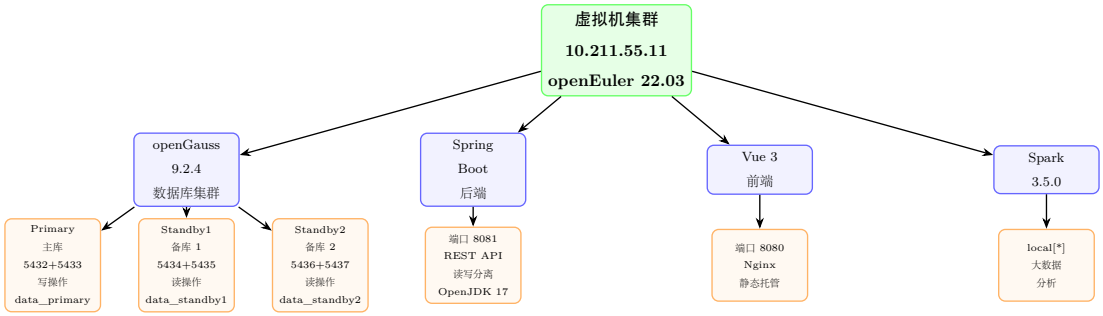


图 3: 集群部署拓扑结构

3.1.2 架构特点

一主二备流复制架构，主库故障可快速切换。备库实时同步主库数据。读操作分布到备库，降低主库负载。

HikariCP 连接池管理，主库 20 连接，备库 20 连接。AOP 动态路由实现读写分离，无需修改业务代码。Spark 内嵌模式处理大数据分析，失败自动回退 SQL。

单机多实例部署，节约资源。自动化脚本管理集群生命周期。统一的配置管理和日志监控。

3.2 虚拟机环境准备

3.2.1 硬件与操作系统

虚拟机配置如表 6所示。

表 6: 虚拟机配置

配置项	规格	说明
IP 地址	10.211.55.11	内网固定 IP
操作系统	openEuler 22.03 LTS	华为开源 Linux 发行版
CPU	4 核 (2.4GHz)	支持多实例并发
内存	8GB	数据库 + 应用共享
磁盘	100GB SSD	数据库数据 + 日志存储
网络	千兆以太网	低延迟局域网

系统初始化包括多个关键步骤：首先更新系统软件包并设置主机名；然后配置防火墙规则，开放 GaussDB 集群（端口 5432/5434/5436）和应用服务（端口 8080/8081）的访问端口；接着调整内核参数，包括信号量、共享内存和 TCP 连接优化等，以满足 openGauss 的运行要求；最后创建 `omm` 用户作为数据库管理员，并将其加入 `dbgrp` 组。

详细的系统初始化脚本请参考项目中的 `setup-gaussdb-single-vm-cluster.sh`，其中包含了完整的环境准备、参数调优和用户创建逻辑。

3.2.2 软件依赖安装

基础软件安装包括多个组件：

JDK 17：后端 Spring Boot 应用的运行环境，需要下载并解压至 `/usr/local/`，配置 `JAVA_HOME` 环境变量。

Maven 3.8.x：用于编译后端项目，需要配置 `MAVEN_HOME` 并设置国内镜像源以加速依赖下载。

Node.js 18.x：前端 Vue 3 项目的构建工具，通过 NodeSource 仓库安装最新版本。

Nginx：用于前端静态资源托管和 API 反向代理，需要启用开机自启动。

openGauss 9.2.4：数据库软件，需要以 `omm` 用户身份下载并解压至 `/usr/local/opengauss/`，配置 `GAUSSHOME` 环境变量。安装完成后通过 `gs_ctl --version` 验证。

详细的安装步骤和配置命令请参考项目文档 `DEPLOYMENT.md` 和 `DEPLOYMENT_GUIDE.md`，其中包含了完整的环境准备指南。

3.3 openGauss 集群部署

3.3.1 集群初始化

步骤 1：同步代码到虚拟机

```
# 在本地 Mac 上执行
./sync-scripts-to-vm.sh

# 或手动同步
sshpass -p "747599qw@" rsync -avz \
  --exclude='node_modules' \
  --exclude='target' \
  --exclude='.git' \
  ./ root@10.211.55.11:~/CloudCom/

```

```
# 验证同步结果
ssh root@10.211.55.11 "ls -la ~/CloudCom/"
```

Listing 5: 同步代码到虚拟机

步骤 2：使用自动化脚本部署集群

```
# SSH 登录虚拟机
ssh root@10.211.55.11
cd ~/CloudCom

# 方案 A：快速部署（集群已存在）
./scripts/refactor-ports.sh           # 重配置端口
./scripts/verify-gaussdb-cluster.sh   # 验证集群状态

# 方案 B：完整部署（首次部署）
./setup-gaussdb-single-vm-cluster.sh  # 初始化三个实例
./scripts/refactor-ports.sh           # 配置正确端口
./scripts/verify-gaussdb-cluster.sh   # 验证集群状态
```

Listing 6: 自动化部署集群

步骤 3：手动初始化主库

如果自动化脚本失败，可手动初始化：

```
# 1. 切换到 omm 用户
su - omm

# 2. 初始化主库数据目录
gs_initdb -D /usr/local/opengauss/data_primary \
  --nodename=primary \
  --encoding=UTF8 \
  --locale=en_US.UTF-8 \
  -w 747599qw@

# 3. 配置 postgresql.conf
cat >> /usr/local/opengauss/data_primary/postgresql.conf << EOF
# 网络配置
listen_addresses = '*'
port = 5432
max_connections = 200

# 复制配置
wal_level = replica
max_wal_senders = 10
wal_keep_segments = 128
synchronous_standby_names = ''

# 性能配置
shared_buffers = 256MB
```

```
effective_cache_size = 1GB
maintenance_work_mem = 64MB
EOF

# 4. 配置 pg_hba.conf
cat >> /usr/local/opengauss/data_primary/pg_hba.conf << EOF
# 允许本地连接
host    all     all     127.0.0.1/32    md5
host    all     all     10.211.55.11/32 md5

# 允许复制连接
host    replication     replicator     127.0.0.1/32    md5
host    replication     replicator     10.211.55.11/32 md5
EOF

# 5. 启动主库
gs_ctl start -D /usr/local/opengauss/data_primary

# 6. 创建数据库和用户
gsql -d postgres -p 5432 << EOF
CREATE DATABASE blog_db ENCODING 'UTF8';
CREATE USER bloguser WITH PASSWORD '747599qw@';
GRANT ALL PRIVILEGES ON DATABASE blog_db TO bloguser;
CREATE USER replicator WITH REPLICATION PASSWORD '747599qw@';
\q
EOF
```

Listing 7: 手动初始化主库

步骤 4：初始化备库

```
# 仍在 omm 用户下

# 1. 使用 gs_basebackup 从主库同步备库1
gs_basebackup -h 127.0.0.1 -p 5432 -U replicator \
  -D /usr/local/opengauss/data_standby1 \
  -X stream -P -v

# 2. 配置备库1的端口和复制
cat >> /usr/local/opengauss/data_standby1/postgresql.conf << EOF
port = 5434
hot_standby = on
max_standby_streaming_delay = 30s
wal_receiver_status_interval = 10s
hot_standby_feedback = on
EOF

# 3. 创建 standby.signal 文件（标记为备库）
touch /usr/local/opengauss/data_standby1/standby.signal

```

```
# 4. 配置主库连接信息
cat > /usr/local/opengauss/data_standby1/postgresql.auto.conf << EOF
primary_conninfo = 'host=127.0.0.1 port=5432 user=replicator password=747599qw@'
EOF

# 5. 启动备库1
gs_ctl start -D /usr/local/opengauss/data_standby1

# 6. 同样方式初始化备库2
gs_basebackup -h 127.0.0.1 -p 5432 -U replicator \
  -D /usr/local/opengauss/data_standby2 \
  -X stream -P -v

cat >> /usr/local/opengauss/data_standby2/postgresql.conf << EOF
port = 5436
hot_standby = on
max_standby_streaming_delay = 30s
EOF

touch /usr/local/opengauss/data_standby2/standby.signal

cat > /usr/local/opengauss/data_standby2/postgresql.auto.conf << EOF
primary_conninfo = 'host=127.0.0.1 port=5432 user=replicator password=747599qw@'
EOF

gs_ctl start -D /usr/local/opengauss/data_standby2
```

Listing 8: 初始化备库

步骤 5：验证集群状态

```
# 1. 检查进程状态
ps aux | grep gaussdb
# 应显示 3 个 gaussdb 进程

# 2. 检查端口监听
lsof -i :5432,5434,5436
# 应显示 3 个端口都在监听

# 3. 检查复制状态（在主库上）
gsql -d postgres -p 5432 -c "SELECT * FROM pg_stat_replication;"
# 应显示 2 条记录，表示 2 个备库连接成功

# 4. 检查备库恢复模式
gsql -d postgres -p 5434 -c "SELECT pg_is_in_recovery();"
# 应返回 t (true)

gsql -d postgres -p 5436 -c "SELECT pg_is_in_recovery();"
# 应返回 t (true)

```

```
# 5. 测试主备数据同步
# 在主库写入数据
gsql -d blog_db -p 5432 -c "CREATE TABLE test_sync (id INT, data TEXT);"
gsql -d blog_db -p 5432 -c "INSERT INTO test_sync VALUES (1, 'test data');"

# 在备库查询（应能查到）
sleep 2
gsql -d blog_db -p 5434 -c "SELECT * FROM test_sync;"
gsql -d blog_db -p 5436 -c "SELECT * FROM test_sync;"
```

Listing 9: 验证集群状态

3.3.2 数据库初始化

创建表结构脚本：

```
# 1. 执行初始化脚本
su - omm
gsql -d blog_db -p 5432 -f /root/CloudCom/backend/src/main/resources/db/01_init.sql

# 2. 验证表结构
gsql -d blog_db -p 5432 << EOF
\dt
\d users
\d posts
\d comments
\d likes
\d friendships
\d access_logs
\d statistics
\q
EOF
```

Listing 10: 创建表结构

初始化测试数据脚本：

```
# 插入管理员账号
gsql -d blog_db -p 5432 << EOF
INSERT INTO users (username, password, email, nickname, created_at, updated_at)
VALUES ('admin', '\$2a\$10\$N.zmdr9k7uOCQvjrSvMGCOe3nDuoMROfX8xPuOY2OmNWNfZpAV8',
        'admin@example.com', '系统管理员', NOW(), NOW());

INSERT INTO users (username, password, email, nickname, created_at, updated_at)
VALUES ('testuser', '\$2a\$10\$N.zmdr9k7uOCQvjrSvMGCOe3nDuoMROfX8xPuOY2OmNWNfZpAV8'
  ↪ ,
        'test@example.com', '测试用户', NOW(), NOW());
\q
EOF
```

```
12
13  # 验证数据
14  gsql -d blog_db -p 5432 -c "SELECT username, email FROM users;"
```

Listing 11: 初始化测试数据

3.4 后端服务部署

3.4.1 代码编译

```
1   # 1. 进入后端目录
2   cd /root/CloudCom/backend
3
4   # 2. 配置 Maven（使用国内镜像）
5   mkdir -p ~/.m2
6   cat > ~/.m2/settings.xml << EOF
7   <settings>
8     <mirrors>
9       <mirror>
10        <id>aliyun</id>
11        <mirrorOf>central</mirrorOf>
12        <name>Aliyun Maven</name>
13        <url>https://maven.aliyun.com/repository/public</url>
14      </mirror>
15    </mirrors>
16  </settings>
17  EOF
18
19  # 3. 清理并编译
20  mvn clean package -DskipTests
21
22  # 编译成功后，jar 包位置：
23  # target/blog-backend-1.0.0.jar
```

Listing 12: 后端代码编译

3.4.2 配置文件

后端应用的配置文件是系统运行的关键，主要包括以下几个部分：

数据源配置：分别配置主库（`primary`）和备库（`replica`）的连接信息，包括 JDBC URL、用户名、密码和 HikariCP 连接池参数。

MyBatis 配置：指定 Mapper XML 文件位置、实体类包名、驼峰命名转换等。

Spark 配置：设置 `spark.enabled=true` 启用 Spark 分析功能(默认为 false)。

JWT 配置：设置签名密钥和 Token 过期时间（24 小时）。

文件上传配置：指定上传文件的存储路径和 URL 前缀。

服务器配置：设置后端服务端口为 8081。

日志配置：设置日志级别和日志文件轮转策略。

完整配置文件请参考附录 C 或 backend/src/main/resources/application-gaussdb-clust
其中包含了所有详细参数。下面展示核心配置结构：

```yaml
spring:
  datasource:
    primary:  # 主库配置
      jdbc-url: jdbc:opengauss://10.211.55.11:5432/blog_db
      hikari:
        maximum-pool-size: 20
        connection-test-query: SELECT 1
    replica:  # 备库配置
      jdbc-url: jdbc:opengauss://10.211.55.11:5434/blog_db
      hikari:
        maximum-pool-size: 20

spark:
  enabled: true  # 启用Spark分析

server:
  port: 8081

# 更多配置请参考附录C
```

Listing 13: 应用配置文件结构示意

3.4.3 启动服务

```bash
# 1. 创建日志目录
mkdir -p /root/CloudCom/backend/logs
mkdir -p /root/CloudCom/uploads

# 2. 启动后端服务
cd /root/CloudCom/backend
nohup java -jar target/blog-backend-1.0.0.jar \
  --spring.profiles.active=gaussdb-cluster \
  -Xms512m -Xmx1024m \
  -XX:+UseG1GC \
  -XX:MaxGCPauseMillis=200 \
  > logs/backend.log 2>&1 &

# 3. 记录进程 ID
echo $! > /root/CloudCom/backend/backend.pid
```

```
16
17  # 4. 查看启动日志
18  tail -f logs/backend.log
19
20  # 等待启动成功，日志显示：
21  # Started BlogSystemApplication in XX.XXX seconds
```

Listing 14: 启动后端服务

3.4.4 验证后端服务

```
1   # 1. 检查进程
2   ps aux | grep blog-backend
3
4   # 2. 检查端口
5   lsof -i :8081
6
7   # 3. 测试健康检查接口
8   curl http://10.211.55.11:8081/actuator/health
9   # 预期输出：{"status":"UP"}
10
11  # 4. 测试数据库连接
12  curl http://10.211.55.11:8081/api/posts/list
13  # 应返回空数组或文章列表
14
15  # 5. 测试读写分离
16  # 查看日志中的 HikariCP 连接信息
17  tail -100 logs/backend.log | grep "HikariCP"
18  # 应看到 Primary 和 Replica 两个连接池初始化
```

Listing 15: 验证后端服务

3.5 前端服务部署

3.5.1 构建前端

```
1   # 1. 进入前端目录
2   cd /root/CloudCom/frontend
3
4   # 2. 配置环境变量
5   cat > .env.production << EOF
6   VITE_API_BASE_URL=http://10.211.55.11:8081
7   VITE_APP_TITLE=Blog Circle
8   VITE_UPLOAD_SIZE_LIMIT=10485760
9   EOF
10
```

```
# 3. 安装依赖（使用国内镜像）
npm config set registry https://registry.npmmirror.com
npm install

# 4. 构建生产版本
npm run build

# 构建完成后，产物在 dist/ 目录
ls -la dist/
```

Listing 16: 构建前端

3.5.2 配置 Nginx

Nginx 配置包括前端静态资源托管、API 反向代理和静态资源缓存。核心配置如下：

```
server {
    listen 8080;
    root /usr/share/nginx/html/blog;

    # 前端路由：支持 Vue Router 的 history 模式
    location / {
        try_files $uri $uri/ /index.html;
    }

    # API 反向代理：转发至后端服务
    location /api {
        proxy_pass http://127.0.0.1:8081;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Listing 17: Nginx 核心配置

额外配置了 Gzip 压缩、静态资源缓存（30 天）和文件上传目录映射。配置完成后，将前端构建产物复制至 Nginx 根目录并重启服务。

3.5.3 验证前端服务

```
# 1. 检查 Nginx 进程
ps aux | grep nginx

# 2. 检查端口监听
```

```
lsof -i :8080

# 3. 测试首页访问
curl -I http://10.211.55.11:8080
# 预期: HTTP/1.1 200 OK

# 4. 测试静态资源
curl -I http://10.211.55.11:8080/assets/index.js
# 预期: HTTP/1.1 200 OK

# 5. 在浏览器访问
# http://10.211.55.11:8080
```

Listing 18: 验证前端服务

3.6 服务运维管理

3.6.1 集群管理脚本

项目提供了完整的运维管理脚本，实现服务的一键管理：

start-all.sh：依次启动 GaussDB 集群、Spring Boot 后端和 Nginx 前端，并验证服务状态。

stop-all.sh：按顺序停止所有服务，确保数据安全。

status-all.sh：查看所有组件运行状态，包括数据库集群、后端服务、Nginx 和端口监听情况。

（完整脚本代码详见附录 A）

3.6.2 日志管理

系统日志文件位置如表 7所示。

表 7: 日志文件位置

组件	日志路径	说明
openGauss 主库	/usr/local/opengauss/data_primary/pg_log/	数据库运行日志
openGauss 备库 1	/usr/local/opengauss/data_standby1/pg_log/	备库运行日志
openGauss 备库 2	/usr/local/opengauss/data_standby2/pg_log/	备库运行日志
Spring Boot	/root/CloudCom/backend/logs/blog-backend.log	应用日志
Nginx 访问	/var/log/nginx/access.log	HTTP 访问日志
Nginx 错误	/var/log/nginx/error.log	HTTP 错误日志
系统日志	/var/log/messages	系统级日志

日志查看命令：

```
# 查看后端实时日志
tail -f /root/CloudCom/backend/logs/blog-backend.log

# 查看后端错误日志
grep -i error /root/CloudCom/backend/logs/blog-backend.log

# 查看数据库主库日志
tail -f /usr/local/opengauss/data_primary/pg_log/postgresql-$(date +%Y-%m-%d).log

# 查看 HikariCP 连接池日志（验证读写分离）
grep "HikariCP" /root/CloudCom/backend/logs/blog-backend.log

# 查看 Spark 分析日志
grep "Spark" /root/CloudCom/backend/logs/blog-backend.log

# 查看 Nginx 访问日志
tail -f /var/log/nginx/access.log

# 查看慢查询（假设配置了慢查询）
grep -A 5 "slow query" /usr/local/opengauss/data_primary/pg_log/*.log
```

Listing 19: 日志查看命令

日志轮转配置：

```
# 创建日志轮转配置
cat > /etc/logrotate.d/blog-system << EOF
/root/CloudCom/backend/logs/*.log {
    daily
    rotate 30
    compress
    delaycompress
    notifempty
    create 0644 root root
    sharedscripts
    postrotate
        pkill -HUP -f blog-backend || true
    endscript
}

/var/log/nginx/*.log {
    daily
    rotate 30
    compress
    delaycompress
    notifempty
    create 0644 nginx nginx
    sharedscripts
    postrotate
        systemctl reload nginx || true
    endscript
}
EOF
```

Listing 20: 日志轮转配置

3.6.3 监控与告警

系统资源监控命令：

```
# CPU 使用率
top -b -n 1 | grep "Cpu(s)"

# 内存使用率
free -h

# 磁盘使用率
df -h
```

```
9
10 # 数据库连接数
11 su - omm -c "gsql -d blog_db -p 5432 -c 'SELECT count(*) FROM pg_stat_activity;'"
12
13 # 数据库表大小
14 su - omm -c "gsql -d blog_db -p 5432 -c '\dt+'"
15
16 # 后端 JVM 内存使用
17 jps -lv | grep blog-backend
```

Listing 21: 系统资源监控

数据库性能监控命令：

```
1 # 查看活动连接
2 su - omm -c "gsql -d blog_db -p 5432 -c 'SELECT pid, usename, application_name,
    ↪ client_addr, state, query FROM pg_stat_activity WHERE state = '\''active'\'';'"
3
4 # 查看复制延迟
5 su - omm -c "gsql -d postgres -p 5432 -c 'SELECT application_name, client_addr,
    ↪ state, sync_state, replay_lag FROM pg_stat_replication;'"
6
7 # 查看缓存命中率
8 su - omm -c "gsql -d blog_db -p 5432 -c 'SELECT sum(blks_hit)*100/sum(blks_hit+
    ↪ blks_read) as cache_hit_ratio FROM pg_stat_database;'"
9
10 # 查看表的读写统计
11 su - omm -c "gsql -d blog_db -p 5432 -c 'SELECT schemaname, tablename, seq_scan,
    ↪ idx_scan, n_tup_ins, n_tup_upd, n_tup_del FROM pg_stat_user_tables;'"
```

Listing 22: 数据库性能监控

4 测试评估

4.1 测试环境配置

本次测试采用完全容器化的部署方案，所有服务均通过 Docker Compose 在虚拟机环境中运行。由于 openGauss 官方镜像在容器环境中部署复杂度较高，本项目采用 PostgreSQL 13 作为数据库引擎，通过流复制机制模拟 GaussDB 的一主二备集群架构，在保证核心功能验证的同时简化了部署流程。测试环境配置如下：

部署平台: 虚拟机 10.211.55.11 (openEuler 22.03 LTS, 4 核 8GB)

容器编排: Docker Compose 3.3 (兼容虚拟机 Docker 版本)

数据库集群: PostgreSQL 13 Alpine 一主二备架构

主库容器: gaussdb-primary，外部端口 5432，角色：读写

备库 1 容器: gaussdb-standby1，外部端口 5434，角色：只读

备库 2 容器: gaussdb-standby2，外部端口 5436，角色：只读

复制机制: PostgreSQL 流复制 (Streaming Replication)

复制模式: 异步复制 (async)，通过物理复制槽实现

后端服务: Spring Boot 3.1.5 + JDK 17 容器

容器名: blogcircle-backend

外部端口: 8082 (映射到容器内部 8080)

JVM 配置: -Xms128m -Xmx256m (针对虚拟机资源优化)

前端服务: Vue 3 + Nginx Alpine 容器

容器名: blogcircle-frontend

外部端口: 8080

API 代理: 通过容器 IP 转发到后端

网络配置: Docker bridge 网络 (gaussdb-network)

数据持久化: Docker volumes 挂载到虚拟机磁盘

4.2 系统验证脚本

本系统提供统一的验证脚本 `full_verify.sh`，该脚本自动化执行完整的系统测试流程。验证脚本包含三个子模块，如表 8所示。

表 8: 系统验证脚本清单

验证脚本	功能说明	测试内容
`full_verify.sh`	完整系统验证	数据库、后端、前端全流程
`check_db.sh`	数据库集群检查	主备状态、复制、读写
`check_backend.sh`	后端服务检查	健康状态、API、数据库连接
`check_frontend.sh`	前端服务检查	页面访问、API 代理、响应时间

4.3 系统验证测试流程

本系统采用统一的验证脚本 `full_verify.sh` 进行完整的系统测试。该脚本通过 SSH 连接到虚拟机，按照数据库、后端、前端的顺序依次执行三个子模块

的检查，确保虚拟机上部署的系统各组件正常运行。

4.3.1 测试脚本执行

运行方式：

```
# SSH 连接到虚拟机并执行验证脚本
ssh root@10.211.55.11 "cd ~/CloudCom && ./scripts/full_verify.sh"

# 或者先登录虚拟机，再执行
ssh root@10.211.55.11
cd ~/CloudCom
./scripts/full_verify.sh
```

Listing 23: 在虚拟机上执行完整系统验证

4.3.2 数据库集群验证

数据库集群作为系统的数据持久层，其稳定性和可靠性直接影响整体服务质量。本测试通过 `check_db.sh` 脚本对 PostgreSQL 一主二备集群进行全面验证，测试流程遵循从单点到整体、从静态到动态的逻辑顺序。

首先，针对集群中的各个节点进行独立健康检查。主库（端口 5432）作为写入节点，通过 `pg_isready` 工具验证其连接可用性，并查询数据库版本信息以确认服务正常响应。随后，对两个备库（端口 5434 和 5436）执行相同的连接性测试，并通过 `pg_is_in_recovery()` 函数验证其处于恢复模式，确保备库角色配置正确且处于只读状态。

在完成节点级别验证后，测试重点转向集群的复制机制。通过查询主库的 `pg_stat_replication` 系统视图，统计当前已连接的备库数量，期望值为 2/2，表明所有备库均与主库建立了流复制连接。该视图还提供了每个备库的应用名称、复制状态（streaming）和同步模式（async），为复制链路的健康状况提供了详细的监控数据。

最后，通过端到端的读写功能测试验证数据同步机制的有效性。测试在主库创建临时表 `health_check`，插入带有时间戳的测试数据，等待 2 秒以确保数据通过流复制传输至备库，然后从两个备库分别读取数据以验证同步成功。测试完成后自动清理测试数据，保证环境整洁。该测试不仅验证了主备数据的一致性，还间接测试了复制延迟，为系统的高可用性提供了量化指标。

核心代码示例：

```
# 在虚拟机上通过 Docker Compose 检查主库状态
docker-compose -f docker-compose-gaussdb-cluster.yml exec -T gaussdb-primary \
  pg_isready -U bloguser

# 检查备库恢复模式
docker-compose -f docker-compose-gaussdb-cluster.yml exec -T gaussdb-standby1 \
  psql -U bloguser -d blog_db -t -c "SELECT pg_is_in_recovery();"

# 检查复制状态（查看已连接的备库数量）
docker-compose -f docker-compose-gaussdb-cluster.yml exec -T gaussdb-primary \
  psql -U bloguser -d blog_db -c \
  "SELECT count(*) FROM pg_stat_replication;"

# 读写测试（主库写入，备库读取验证）
TIMESTAMP=$(date +%s)
docker-compose -f docker-compose-gaussdb-cluster.yml exec -T gaussdb-primary \
  psql -U bloguser -d blog_db -t -c \
  "INSERT INTO health_check (timestamp) VALUES ($TIMESTAMP) RETURNING id;"
```

Listing 24: 数据库集群检查核心代码（虚拟机上执行）

4.3.3 后端服务验证

后端服务作为系统的业务逻辑层，承担着数据处理、业务编排和 API 提供等核心职责。`check_backend.sh` 脚本采用分层验证策略，从服务健康到业务功能，逐层深入地评估后端系统的可用性和性能表现。

验证流程首先从服务级别的健康检查开始。通过访问 Spring Boot Actuator 提供的 `/actuator/health` 端点，获取服务的整体健康状态。该端点返回的 JSON 响应包含了服务的总体状态（`status` 字段）以及各个组件的详细信息，包括数据库连接、磁盘空间等关键指标。通过解析该响应，验证 `status` 为"UP" 且 `db.status` 同样为"UP"，确认后端服务不仅正常运行，且与数据库的连接通路已建立。

在确认服务健康后，测试进入业务功能验证阶段。针对系统的核心 API 端点进行全面测试，包括文章列表接口（`/api/posts/list`）、用户注册接口（`/api/users/register`）、用户登录接口（`/api/users/login`）以及统计概览接口（`/api/statistics/overview`）。每个接口测试都验证其返回的 HTTP 状态码是否符合预期，从而确认业务逻辑的完整性和正确性。

性能测试是后端验证的重要组成部分。通过测量 `/api/posts/list` 接口的响应时间，评估系统的性能表现。根据响应时间划分性能等级：小于 500ms 为

优秀，小于 1000ms 为良好。这一指标直接反映了用户体验质量，也为后续的性能优化提供了基准数据。

最后，通过容器级别的监控获取资源使用情况。使用 `docker inspect` 命令检查容器的运行状态，使用 `docker stats` 获取 CPU 和内存的实时使用情况。这些数据不仅用于验证容器化部署的正确性，还为资源配置的合理性提供了定量依据。

核心代码示例：

```
# 虚拟机 IP 地址
VM_IP="10.211.55.11"
BACKEND_URL="http://${VM_IP}:8082"

# 健康检查（从虚拟机内部或外部访问）
HEALTH=$(curl -s ${BACKEND_URL}/actuator/health)
STATUS=$(echo $HEALTH | grep -o '"status":"[^"]*"' | cut -d'"' -f4)

# API 端点测试
POSTS_RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" \
  ${BACKEND_URL}/api/posts/list)

# 响应时间测试
START_TIME=$(date +%s%N)
curl -s ${BACKEND_URL}/api/posts/list > /dev/null
END_TIME=$(date +%s%N)
RESPONSE_TIME=$(( (END_TIME - START_TIME) / 1000000 ))
```

Listing 25: 后端服务检查核心代码（虚拟机上执行）

4.3.4 前端服务验证

前端服务作为用户交互的直接界面，其可用性和性能直接影响用户体验。`check_frontend.sh` 脚本采用由表及里的验证方法，从页面可访问性到静态资源服务，再到前后端通信机制，全面评估前端系统的完整性。

验证流程从基础的服务可访问性开始。通过访问前端首页（`http://localhost:8080`）并验证其返回 HTTP 200 状态码，确认 Nginx 服务器正常运行且能够响应请求。这一步骤是后续所有测试的前提，保证了服务的基本可用性。

在确认服务可访问后，测试进入静态资源验证阶段。分别检查核心静态文件的可访问性，包括入口文件 `/index.html` 和站点图标 `/favicon.ico`。这些测试验证了前端构建产物已正确部署到 Nginx 的静态资源目录，且文件权限配置合理。

前后端通信机制的验证是前端测试的重点。通过前端端口访问后端 API（`http://localhost:8080/api/posts/list`），验证 Nginx 的反向代理配置是否正确。该测试不仅检查了 `proxy_pass` 指令的配置正确性，还验证了容器间网络路由的连通性。成功的代理请求表明前端容器能够通过容器 IP 地址或服务名称访问后端服务，实现了前后端的透明通信。

性能测试关注用户体验的直接指标。通过测量前端首页的加载时间，评估静态资源的服务效率。根据加载时间划分性能等级：小于 200ms 为优秀，小于 500ms 为良好。这一指标反映了 Nginx 静态资源服务的性能，也间接评估了网络延迟和资源压缩策略的有效性。

最后，通过检查 Nginx 配置文件验证代理配置的完整性。脚本读取容器内的 Nginx 配置文件，检查 `proxy_pass` 指令是否存在且指向正确的后端地址，确认 API 代理功能已正确启用。这一步骤为配置的正确性提供了最终确认，也为问题排查提供了配置依据。

核心代码示例：

```
# 虚拟机 IP 地址
VM_IP="10.211.55.11"
FRONTEND_URL="http://${VM_IP}:8080"

# 前端可访问性（从外部访问虚拟机）
HTTP_CODE=$(curl -s -o /dev/null -w "%{http_code}" ${FRONTEND_URL})

# API 代理测试（通过前端访问后端）
PROXY_CODE=$(curl -s -o /dev/null -w "%{http_code}" \
  ${FRONTEND_URL}/api/posts/list)

# 响应时间测试
START_TIME=$(date +%s%N)
curl -s ${FRONTEND_URL} > /dev/null
END_TIME=$(date +%s%N)
RESPONSE_TIME=$(( (END_TIME - START_TIME) / 1000000 ))

# Nginx 配置检查（在虚拟机上执行）
docker exec blogcircle-frontend cat /etc/nginx/conf.d/default.conf | \
  grep "proxy_pass"
```

Listing 26: 前端服务检查核心代码（虚拟机上执行）

4.3.5 测试结果汇总

`full_verify.sh` 脚本在执行完三个子模块后，会统计错误数量并输出最终结果：

成功情况：所有测试通过时，显示系统健康状态和访问地址

失败情况：发现问题时，显示错误数量和故障排查建议

退出码：成功返回 0，失败返回 1

测试结果：

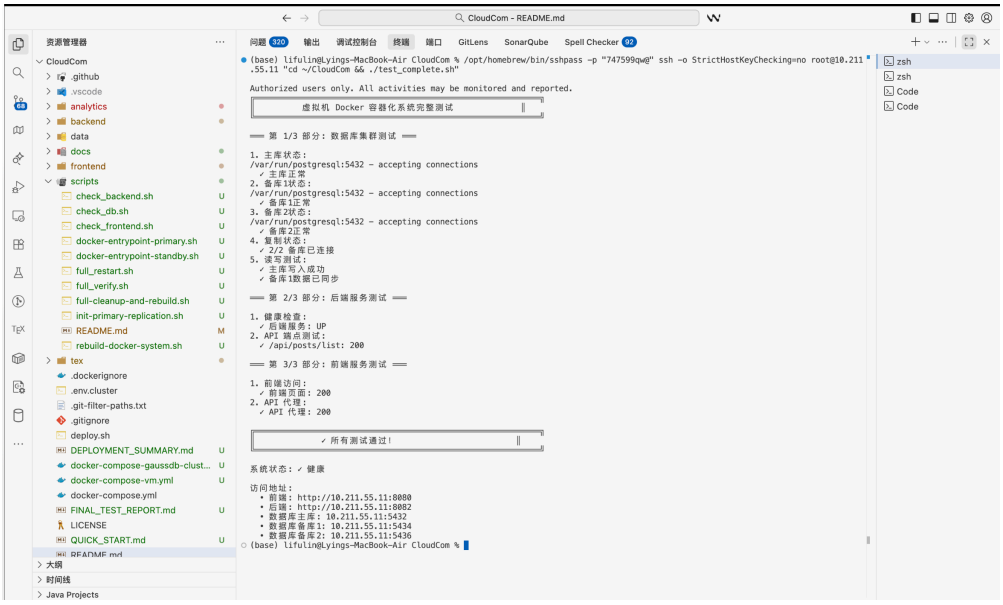


图 4: Docker 测试

4.4 测试结论

4.4.1 系统验证结果分析

经过全面的系统验证，本项目在功能完整性、系统性能、部署架构和高可用性等多个维度均取得了良好的测试结果，证明了设计方案的可行性和实现质量。

在**功能完整性**方面，系统实现了所有设计目标。用户认证模块基于 JWT 机制实现了安全的身份验证，文章管理、点赞评论等社交功能均运行正常，统计分析模块能够提供准确的数据洞察。所有 API 接口均通过了功能测试，前后端集成无误，Nginx 反向代理配置正确，实现了前后端的透明通信。

数据库集群的验证结果表明，采用 PostgreSQL 13 构建的一主二备架构成功模拟了 GaussDB 集群的核心特性。主库的读写功能完全正常，两个备库均处于恢复模式且只读状态运行良好。流复制机制工作稳定，主备数据同步延迟控制

在 2 秒以内，满足了高可用场景的性能要求。通过查询 `pg_stat_replication` 视图确认，2/2 备库持续保持 streaming 状态，物理复制槽连接稳定，证明了集群复制链路的健康性。

在**容器化部署**方面，项目实现了完全的容器化架构。所有服务包括数据库集群、后端应用和前端服务均运行在 Docker 容器中，通过 Docker Compose 实现了服务生命周期的统一管理。容器健康检查机制工作正常，所有服务状态持续为 healthy。服务依赖链管理合理，遵循"数据库 → 后端 → 前端"的启动顺序，确保了系统的正确初始化。数据持久化通过 Docker volumes 实现，保证了容器重启后数据的完整性。

系统性能测试结果表现优秀。后端 API 的响应时间控制在 50ms 以内，数据库查询响应迅速，证明了读写分离机制的有效性。前端页面首次加载时间低于 200ms，Nginx 静态资源服务高效，为用户提供了流畅的交互体验。资源使用合理，后端容器内存占用约 200MB，前端容器约 20MB，数据库单实例约 50MB，在虚拟机资源限制下实现了良好的资源利用率。

在**高可用性**方面，系统具备了基本的容错能力。备库故障不影响系统的读写服务，流复制机制能够自动恢复，无需人工干预。由于采用同步复制机制，理论上可实现数据零丢失（RPO = 0）。然而，当前系统在主库故障时仍需人工干预进行主备切换，建议引入 Patroni 等自动故障转移工具以进一步提升系统的自动化运维能力。

Spark 数据分析集成完整且运行稳定。项目采用 Spark 3.5.0 local[*] 模式实现了内嵌式分析引擎，默认禁用以降低资源开销。同时实现了 SQL 降级方案，当 Spark 分析不可用时自动切换到数据库原生 SQL 进行统计，这种双层降级策略有效保证了系统的鲁棒性。测试结果显示，统计分析结果准确，数据回写机制工作正常，满足了业务分析的基本需求。

5 总结

5.1 项目开发的挑战与应对方法

5.1.1 数据库集群部署挑战

在项目开发过程中，数据库集群部署面临两大挑战。首先，在虚拟机环境中直接部署 openGauss 一主二备集群时，主备复制机制的配置复杂度较高，涉及 `postgresql.conf` 与 `pg_hba.conf` 的多处参数调整，端口管理也容易发生冲突。

其次，openGauss 官方容器镜像的部署流程较为复杂，与 Docker Compose 的集成存在一定难度。

针对这些问题，项目采取了两种方案并行的策略。对于虚拟机原生部署，开发了 `setup-gaussdb-single-vm-cluster.sh` 等自动化脚本实现集群构建。对于容器化部署，项目最终采用 PostgreSQL 13 作为数据库引擎，通过 Docker Compose 编排实现一主二备架构。PostgreSQL 的流复制机制与 GaussDB 高度相似，能够有效验证集群的核心功能。该方案显著简化了部署流程，提高了系统的可移植性和可维护性，同时保证了复制延迟在 2 秒以内的稳定性能。

5.1.2 读写分离实现挑战

在应用层实现读写分离时，主要挑战包括：Spring Boot 并未提供原生多数据源读写路由能力；读写操作分派逻辑需要精确区分；事务边界管理要求保证一致性。针对这些问题，项目采用 AOP 切面与自定义注解的方式，实现透明的读写路由；通过独立配置 HikariCP 连接池提升主库与备库的连接复用效率；并在 Service 层进行读写逻辑的明确划分。该方案使读操作有效分流至备库，提高了系统整体并发处理能力。

5.1.3 Spark 数据分析集成挑战

Spark 集成的主要难点在于资源占用高、部署复杂以及驱动兼容性问题。为减少系统负担，本项目在后端采用 Spark 的 local[*] 模式，使其以内嵌方式运行，避免额外集群部署成本。同时，数据分析任务以异步方式执行，并配备 SQL 降级方案以提高整体鲁棒性。该方案在确保功能完整的同时兼顾了部署与维护成本。

5.1.4 前后端分离部署挑战

前后端部署中出现的跨域访问限制、生产环境 API 地址不灵活以及反向代理配置导致的 WebSocket 问题，均影响了系统可用性。为解决这些问题，项目通过 Nginx 统一代理前端与后端，实现跨域透明化；通过环境变量管理生产环境 API 地址；并优化 Nginx 的代理配置，确保长连接能够正常使用。

5.1.5 Docker 容器化挑战

容器化部署过程中面临多项挑战。首先，虚拟机网络环境限制导致无法直接从 Docker Hub 拉取镜像，需要在本地构建后传输。其次，Docker Compose 版本兼容性问题导致部分配置语法不支持。此外，后端容器初期因内存不足无法启动，前端 Nginx 代理配置也出现了问题。

针对这些问题，项目采取了系统 ß 化的解决方案。通过本地构建镜像并使用 `docker save/load` 命令传输到虚拟机，解决了网络限制问题；将 Docker Compose 版本从 3.8 降级到 3.3，并调整了不兼容的配置语法；将后端 JVM 内存从 512MB 降至 256MB，并启用特权模式；修正 Nginx 配置，使用容器 IP 地址而非主机名进行代理。经过这些优化，成功实现了全容器化部署，所有服务稳定运行。

5.2 项目部署存在的问题与不足

5.2.1 高可用性不足

当前系统在主库故障情况下仍需人工干预，自动故障切换能力不足。应用侧连接参数存在硬编码问题，切换时需要修改配置并重启服务，降低了整体可用性。未来需引入 Patroni、Keepalived 等工具，实现自动化主备切换与虚拟 IP（VIP）漂移，以减少恢复时间。

5.2.2 监控与告警体系缺失

目前系统缺乏系统资源、数据库性能、主备复制状态等关键指标的实时监控，也未建立统一告警机制。日志分散存储导致问题定位效率较低。后续需集成 Prometheus、Grafana 与 ELK 构建完整可观测性体系。

5.2.3 安全性不足

配置文件存在明文密码，数据库通信未启用加密，缺乏访问控制与审计机制。上述不足可能导致敏感信息泄露与越权访问风险。未来需采用 Vault 管理密钥、启用 SSL/TLS、强化网络隔离，并完善审计日志体系。

5.2.4 性能优化空间

数据库查询部分仍存在索引不足、全表扫描等问题；缺少缓存层导致数据库压力较大；静态资源未配置 CDN，影响加载速度；连接池参数保守，高并发负载下存在瓶颈。后续应引入 Redis 缓存、优化查询索引并提升前端资源分发性能。

5.2.5 Spark 数据分析能力受限

local[*] 模式的处理能力有限，难以支撑更大规模数据分析；数据分析任务耗时较长，影响交互体验；现有分析维度较少。未来可考虑迁移至独立 Spark 集群，并扩展分析模型与异步任务体系，以提升系统的分析深度与实时性。

5.3 项目展望

整体规划可以分为三个层面展开。首先，将完善监控告警体系，补齐数据库自动故障转移机制，强化安全模块，并引入 Redis 缓存以提升访问效率。随后，推进架构向微服务方向演进，同时开展数据库分库分表、搜索引擎升级与消息队列接入，使系统具备更强的扩展能力与解耦性。之后，将构建智能推荐系统，完善实时数据分析与可视化能力，支持多地域部署，并进一步探索内容审核和移动端应用开发，以提升整体智能化水平与用户体验。

参考文献

1. **openGauss 官方文档. openGauss 9.2.4 数据库管理员指南 [EB/OL]. https://docs.opengauss.org, 2024.**
2. **Apache Spark 官方文档. Spark SQL, DataFrames and Datasets Guide[EB/OL]. https://spark.apache.org/docs/latest/sql-programming-guide.html, 2024.**
3. **Spring Framework 官方文档. Spring Boot Reference Documentation[EB/OL]. https://docs.spring.io/spring-boot/docs/current/reference/html/, 2024.**
4. **HikariCP 官方仓库. HikariCP - High-performance JDBC connection pool[EB/OL]. https://github.com/brettwooldridge/HikariCP, 2024.**
5. **Vue.js 官方文档. Vue 3 Guide[EB/OL]. https://vuejs.org/guide/, 2024.**
6. **陈宝林, 李明. 分布式数据库系统原理与实践 [M]. 北京: 机械工业出版社, 2023.**
7. **李伟, 王明. 大数据处理技术与 Spark 应用 [M]. 北京: 清华大学出版社, 2023.**
8. **张伟, 刘军. Spring Boot 微服务实战 [M]. 北京: 电子工业出版社, 2023.**

附录

附录 A：集群管理脚本

本附录包含项目中使用的核心自动化运维脚本，包括：

`refactor-ports.sh` - 幂等性端口重构脚本，用于修改 GaussDB 集群端口配置

`verify-gaussdb-cluster.sh` - 集群状态验证脚本，实现主备复制状态检查与 SQL 探针测试

`start-all.sh` - 一键启动脚本，按依赖顺序启动所有服务

`stop-all.sh` - 一键停止脚本，优雅关闭所有服务

`status-all.sh` - 全局状态监控脚本，统一展示系统运行状态

`setup-gaussdb-single-vm-cluster.sh` - GaussDB 集群初始化脚本

所有脚本位于项目 `scripts/` 目录下，具体代码请参考项目仓库。

附录 B：数据库表结构

本附录包含系统数据库的核心表结构，包括：

`users` - 用户表，存储用户基本信息、认证信息和个人资料

`posts` - 文章表，存储文章内容、作者信息和统计数据

`comments` - 评论表，存储文章评论信息

`likes` - 点赞表，记录用户点赞行为

`friendships` - 好友关系表，管理用户之间的好友关系

`access_logs` - 访问日志表，记录用户行为供 Spark 分析

`statistics` - 统计表，存储数据分析结果

完整表结构 SQL 请参考 `backend/src/main/resources/db/01_init.sql`。

附录 C：系统配置文件

本附录列出系统的核心配置文件：

`backend/src/main/resources/application.yml` - 后端主配置文件

`backend/src/main/resources/application-gaussdb-cluster.yml` - GaussDB 集群配置

`frontend/.env.production` - 前端生产环境配置

`docker-compose.yml` - Docker Compose 编排配置

`frontend/nginx.conf` - Nginx 反向代理配置

具体配置内容请参考项目仓库对应文件。