



分布式键值存储系统

翟锦洋、付涛

December 31, 2025



中国科学院大学



Table of Contents

1 项目概述

▶ 项目概述

▶ meta-server 架构

▶ kv-store 架构

▶ 总结



实现功能

1 项目概述

- 设计并实现键值数据的多机分布式数据存储，采用 RPC 通信协议。
- 支持数据的增加、修改和删除操作
- 支持记录的多副本存储
- 提供读写接口/检索界面访问数据存储
- 支持 raft 一致性协议模块，故障情况下实现保证可靠存储
- 实现元数据管理



Table of Contents

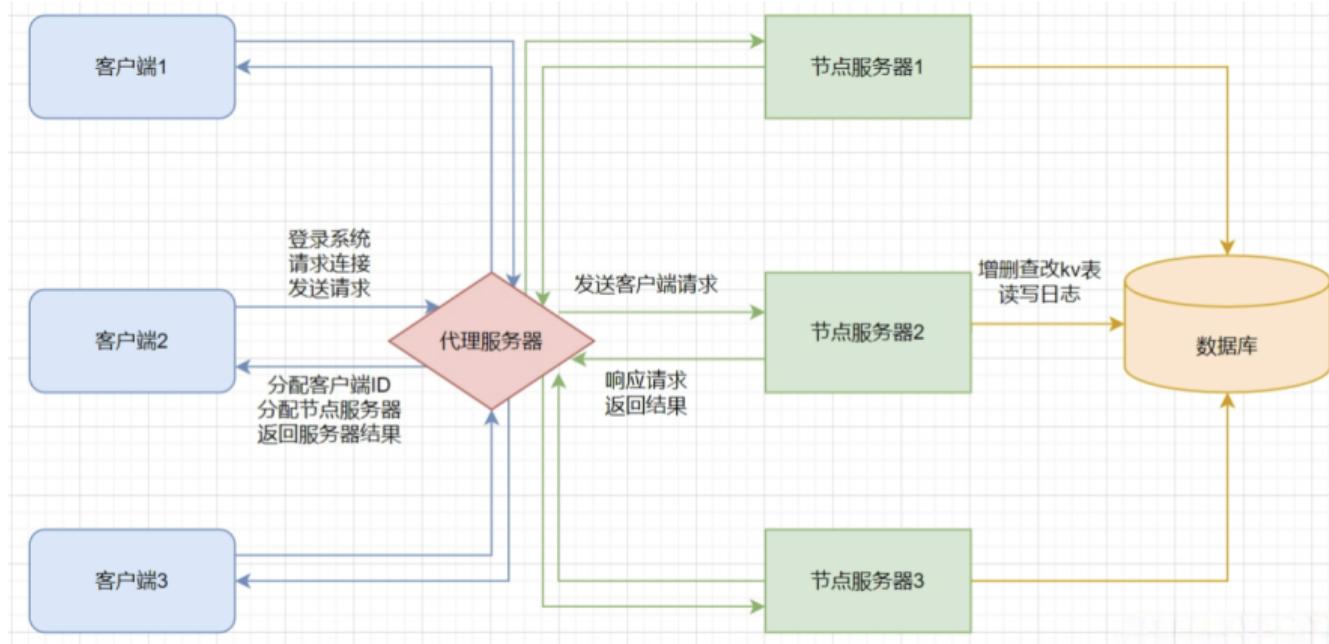
2 meta-server 架构

- ▶ 项目概述
- ▶ meta-server 架构
- ▶ kv-store 架构
- ▶ 总结



整体架构图

2 meta-server 架构





整体架构

2 meta-server 架构

应用架构

应用采用三层分布式架构，客户端通过代理服务器与节点服务器通信。代码组织上，使用 Python 语言实现 meta-server；使用 XML-RPC 实现客户端与服务器之间的通信；使用 Flask 实现 HTTP 测试服务器，使用 requests 库实现节点服务器与 Raft 集群的 HTTP 通信。

```
meta-server/
|-- client.py // 客户端实现
|-- proxy_server.py // 代理服务器
|-- node_server.py // 节点服务器
|-- test_flask.py // Flask测试服务器
|-- api.md // API文档
|-- requirements.txt // 依赖管理
```



系统架构概览

2 meta-server 架构

三层架构

- **客户端**: 用户交互界面
- **代理服务器**: 请求路由和负载分配
- **节点服务器**: 业务逻辑处理

通信方式

- 客户端 \leftrightarrow 代理服务器: XML-RPC
- 代理服务器 \leftrightarrow 节点服务器: XML-RPC
- 节点服务器 \leftrightarrow Raft 集群: HTTP



组件说明

2 meta-server 架构

客户端 (client.py)

- **功能：** 用户交互界面
- **通信方式：** XML-RPC
- **主要特性：**
 - 用户认证（用户名/密码）
 - 客户端 ID 分配
 - 命令解析与执行



组件说明

2 meta-server 架构

代理服务器 (proxy_server.py)

- **功能：**请求路由和负载分配
- **端口：** 21000
- **主要特性：**
 - 用户认证管理
 - 客户端 ID 分配与管理
 - 命令转发到对应的节点服务器
 - 响应格式化与错误处理



组件说明

2 meta-server 架构

节点服务器 (node_server.py)

- **功能：**业务逻辑处理层
- **端口：** $20000 + \text{server_id}$
- **主要特性：**
 - 键值对操作 (PUT、GET、DEL、LIST)
 - 集群管理操作 (ADD-LEARNER、CHANGE-MEMBERSHIP、METRICS)
 - 操作日志记录
 - HTTP 请求封装



数据操作 API

2 meta-server 架构

支持的命令

- PUT: 添加或更新键值对
- GET: 读取单个键值
- DEL: 删除键值对
- LIST: 读取所有键值对



PUT 操作

2 meta-server 架构

功能特点

- **API 端点**: POST /write
- **请求格式**: {"Put": {"key": "key", "value": "value"}}
- **响应**: "Ok" 或 "Err"
- **特点**: 自动区分添加和更新操作

代码示例

```
json_data = {"Put": {"key": key, "value": value}}
response = self._http_request('/write', json_data=json_data)
```



GET 操作

2 meta-server 架构

功能特点

- **API 端点**: POST /read
- **请求格式**: 字符串 "key"
- **响应格式**: {"Ok": "value"} 或 {"Ok": ""}
- **特点**: 支持缓存机制（可选）

代码示例

```
json_data = key
response = self._http_request('/read', json_data=json_data, method='POST')
```



DEL 和 LIST 操作

2 meta-server 架构

DEL 操作

- **API 端点:** POST /write
- **请求格式:** {"Del": {"key": "key"}}
- **响应:** "Ok" 或 "Err"

LIST 操作

- **API 端点:** GET /read-all
- **响应格式:** {"Ok": [...], ...}
- **特点:** 返回完整数据库快照



集群管理 API

2 meta-server 架构

ADD-LEARNER

- **功能**: 添加 Raft 节点作为 learner
- **API 端点**: POST /add-learner
- **请求格式**: [node_id, "api_addr"]

CHANGE-MEMBERSHIP

- **功能**: 改变集群成员关系
- **API 端点**: POST /change-membership
- **请求格式**: [node_id1, node_id2, ...]



METRICS 操作

2 meta-server 架构

功能

查询集群状态，返回详细信息：

- 节点基本信息：ID、任期、状态、Leader
- 投票信息、日志信息、成员配置
- 心跳信息、复制信息

用途

集群监控和诊断



客户端连接流程

2 meta-server 架构

认证和连接

```
def connect(self, username, password):
    self.port = '21000'
    self.proxy = xmlrpclib.ServerProxy(
        'http://localhost:' + self.port)
    if self.proxy.authenticate(username, password):
        self.id = self.proxy.get_id()
        return self.id
    return None
```



命令处理流程

2 meta-server 架构

代理服务器命令路由

```
def function(self, client_id, clause):
    clause = clause.lower().strip().split()
    command = clause[0]
    if command in ['put', 'get', 'del', ...]:
        method_name = command
        server_function = getattr(self, method_name)
        return server_function(client_id, clause)
```



PUT 操作实现

2 meta-server 架构

节点服务器 PUT 方法

```
def put(self, key, value):
    existing_value = self.get(key)
    json_data = {"Put": {"key": key, "value": value}}
    response = self._http_request('/write', json_data=json_data)
    if response == "Ok":
        self.write_log(f"添加key: {key}, value: {value}")
        return True
    return False
```



Flask API 测试端点

2 meta-server 架构

HTTP 接口

- POST /write: 写入操作 (Put/Del)
- POST /read: 读取单个键值
- GET /read-all: 读取所有键值
- POST /add-learner: 添加 learner 节点
- POST /change-membership: 改变成员关系
- GET /metrics: 查询集群状态
- GET /health: 健康检查



Flask API 示例

2 meta-server 架构

写入操作实现

```
@app.route('/write', methods=['POST'])
def write():
    data = request.get_json()
    if 'Put' in data:
        put_data = data['Put']
        key = put_data.get('key')
        value = put_data.get('value')
        with db_lock:
            database[key] = value
    return "Ok", 200
...
```



Table of Contents

3 kv-store 架构

- ▶ 项目概述
- ▶ meta-server 架构
- ▶ kv-store 架构
- ▶ 总结



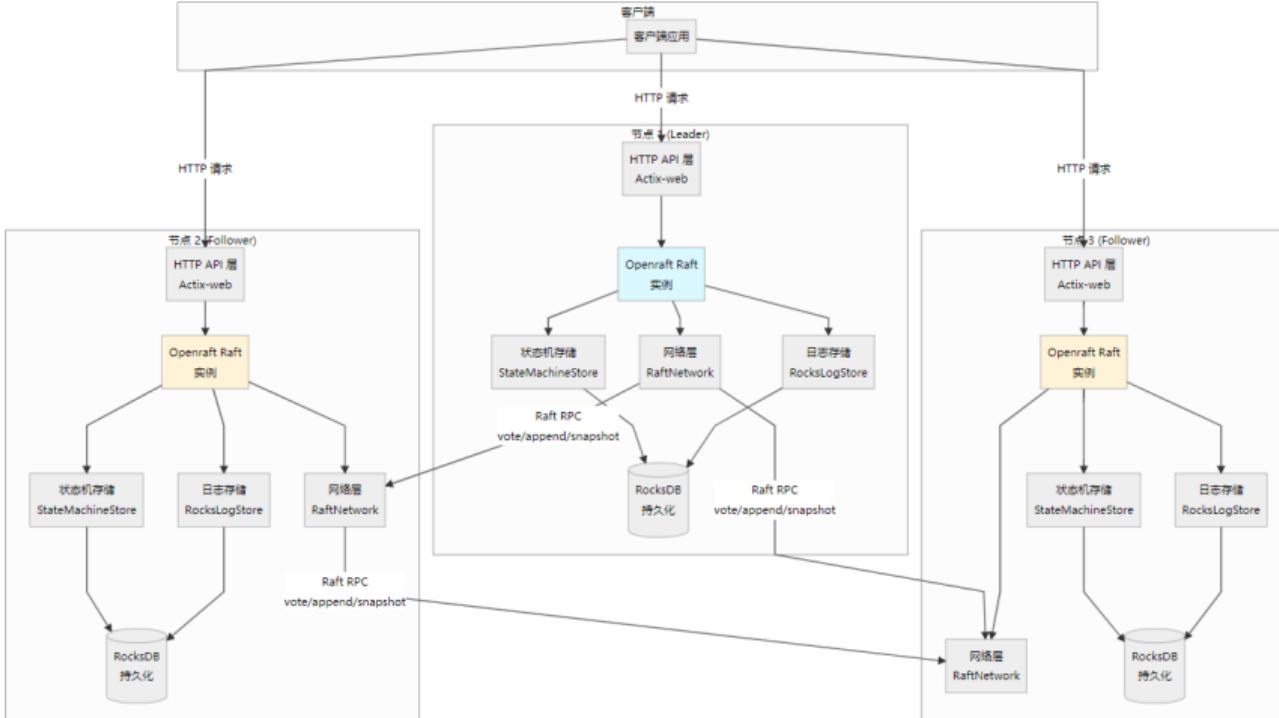
整体架构

3 kv-store 架构

应用架构

应用采用多节点分布式架构，每个节点运行独立的 Raft 实例。代码组织上，使用 rust 语言实现 kv-store；使用 openraft 实现 raft 协议提供一致性保证；使用 actix-web 实现 http 服务器，使用 rust bTreeMap 和 RocksDB 实现存储层。

```
kv-store/
|-- src/
|   |-- lib.rs // 集群实现
|   |-- app.rs // 集群声明
|   |-- bin/main.rs // 主入口
|   |-- store/ // 存储层
|       |-- mod.rs // 存储层实现
|           -- log_store.rs // 日志存储
|   |-- network // 网络层
|       |-- mod.rs // 网络层实现
|       |-- api.rs // 业务 API
|       |-- management.rs // 管理 API
|           -- raft.rs // raft协议实现
|   |-- openraft_network // openraft网络层
|-- test.sh // 测试脚本
-- start.sh // 启动脚本
```





核心组件 3 kv-store 架构

1. Raft 节点 (src/lib.rs)

- 每个节点包含一个 `openraft::Raft<TypeConfig>` 实例
- 节点通过 HTTP 服务 (Actix-web) 暴露 RPC 和业务 API

2. 存储层 (src/store/)

- **日志存储** (RocksLogStore): 基于 RocksDB 持久化 Raft 日志条目
- **状态机存储** (StateMachineStore): 管理 KV 数据状态, 实现日志应用到状态机

3. 网络层 (src/openraft_network/)

- 实现 `RaftNetwork` 和 `RaftNetworkFactory`, 处理节点间 RPC 通信
- 通过 HTTP 传输 Raft 协议消息 (`vote`、`append_entries`、`install_snapshot`)



存储层实现

3 kv-store 架构

日志存储 (RocksLogStore)

RocksLogStore 实现 RaftLogStorage<TypeConfig>:

- **持久化:** 使用 RocksDB logs column family 存储日志条目
- **元数据:** 使用 meta column family 存储 vote 和 last_purged_log_id
- **方法:**
 - append(): 追加日志条目并刷新 WAL
 - truncate(): 截断冲突日志
 - purge(): 清理已应用日志
 - get_log_state(): 获取日志状态

状态机存储 (StateMachineStore)

StateMachineStore 实现

RaftStateMachine<TypeConfig>:

- **状态管理:** BTreeMap<String, String> 存储 KV 数据
- **日志应用 (apply()):**
 - 处理 Put/Del 请求，更新状态机
 - 处理 Membership 变更
 - 更新 last_applied_log_id
- **快照:**
 - build_snapshot(): 序列化状态机数据
 - install_snapshot(): 从快照恢复状态机



网络层实现

3 kv-store 架构

Network 实现 RaftNetwork<TypeConfig>

- **RPC 方法:**
 - `vote()`: 选举投票
 - `append_entries()`: 日志复制
 - `install_snapshot()`: 快照传输
- **通信方式:** HTTP POST 请求, JSON 序列化

Raft RPC 接口

- POST /vote: 选举投票
- POST /append: 日志复制
- POST /snapshot: 快照传输



API 层设计 (src/network/)

3 kv-store 架构

管理 API

- POST /init: 初始化集群
- POST /add-learner: 添加 learner 节点
- POST /change-membership: 改变成员关系
- GET /metrics: 查询集群状态

业务 API

- POST /write: KV 写入操作 (Put/Del)
- POST /read: KV 读取操作 (单个键值)
- GET /read-all: 读取所有键值对



写入流程 (Put/Delete)

3 kv-store 架构

流程说明

/write API 接受 Request 类型的请求，支持两种操作：Put 用于插入或更新键值对，Del 用于删除指定键。write 的语义是修改状态机，即添加一条 log 日志。

执行步骤

1. 客户端调用 /write API
2. Raft.client_write() 将请求追加到日志
3. Leader 复制日志到多数节点
4. 日志提交后，apply() 应用到状态机
5. 返回响应给客户端



写入流程实现

3 kv-store 架构

src/mod.rs fn apply()

```
EntryPayload::Normal(req) => match req {  
    Request::Put { key, value } => {  
        let mut st = self.data.kvs.write().await;  
        st.insert(key, value.clone());  
        Response { value: Some(value) }  
    }  
    Request::Del { key } => {  
        let mut st = self.data.kvs.write().await;  
        st.remove(&key);  
        Response { value: Some("Ok".to_string()) }  
    }  
}
```



读取流程

3 kv-store 架构

流程说明

读取操作直接从状态机读取，无需经过 Raft 协议。

执行步骤

1. 客户端调用 /read 或 /read-all API
2. 直接从 app.key_values（状态机数据）读取
3. 返回查询结果



读取流程实现

3 kv-store 架构

直接从状态机读取 (src/network/api.rs)

```
#[post("/read")]
pub async fn read(app: Data<App>, req: Json<String>)
    -> actix_web::Result<impl Responder> {
    let key = req.0;
    let kvs = app.key_values.read().await;
    let value = kvs.get(&key);

    let res: Result<String, Infallible> =
        Ok(value.cloned().unwrap_or_default());
    Ok(Json(res))
}
```



测试场景

3 kv-store 架构

test.sh

- **集群初始化**: 启动 3 个未初始化的节点, 初始化节点 1 为单节点集群 (Leader)
- **成员管理**: 添加节点 2 和节点 3 为 learner, 将集群从单节点 [1] 扩展为三节点 [1, 2, 3]
- **数据写入**: 在 Leader 节点写入键值对, 验证数据已复制到所有节点
- **数据读取**: 从所有节点 (Leader 和 Followers) 读取数据, 验证一致性
- **数据删除**: 测试删除键值对, 验证删除操作在所有节点生效
- **模拟故障**: 杀死 Leader 节点 (节点 1), 验证剩余节点选举出新 Leader, 在新 Leader 上继续写入和读取
- **节点恢复**: 重启节点 1, 验证节点 1 能够从快照或日志恢复状态, 验证恢复后数据一致性



Table of Contents

4 总结

- ▶ 项目概述
- ▶ meta-server 架构
- ▶ kv-store 架构
- ▶ 总结



项目总结

4 总结

meta-server

- 实现客户端登录与节点分配
- 支持完整数据操作（增删改查）接口
- 实现了 raft 节点管理、集群状态监控

kv-store

- 基于 Openraft 保证据一致性，实现了日志与状态机同步
- 使用 RocksDB 进行日志、状态机和数据快照的持久化存储
- 测试了节点故障和恢复，具有容错性



分布式键值存储系统

*Thank you for listening!
Any questions?*