从零实现Mini-Redis完整教程

更多C++Linux教程：[程序员老廖的个人空间-程序员老廖个人主页-哔哩哔哩视频]

概述

本教程将手把手带你从零实现一个高性能的Mini-Redis，涵盖RESP协议解析、事件驱动网络编程、数据结构实现、持久化和主从复制等核心技术。

项目目标

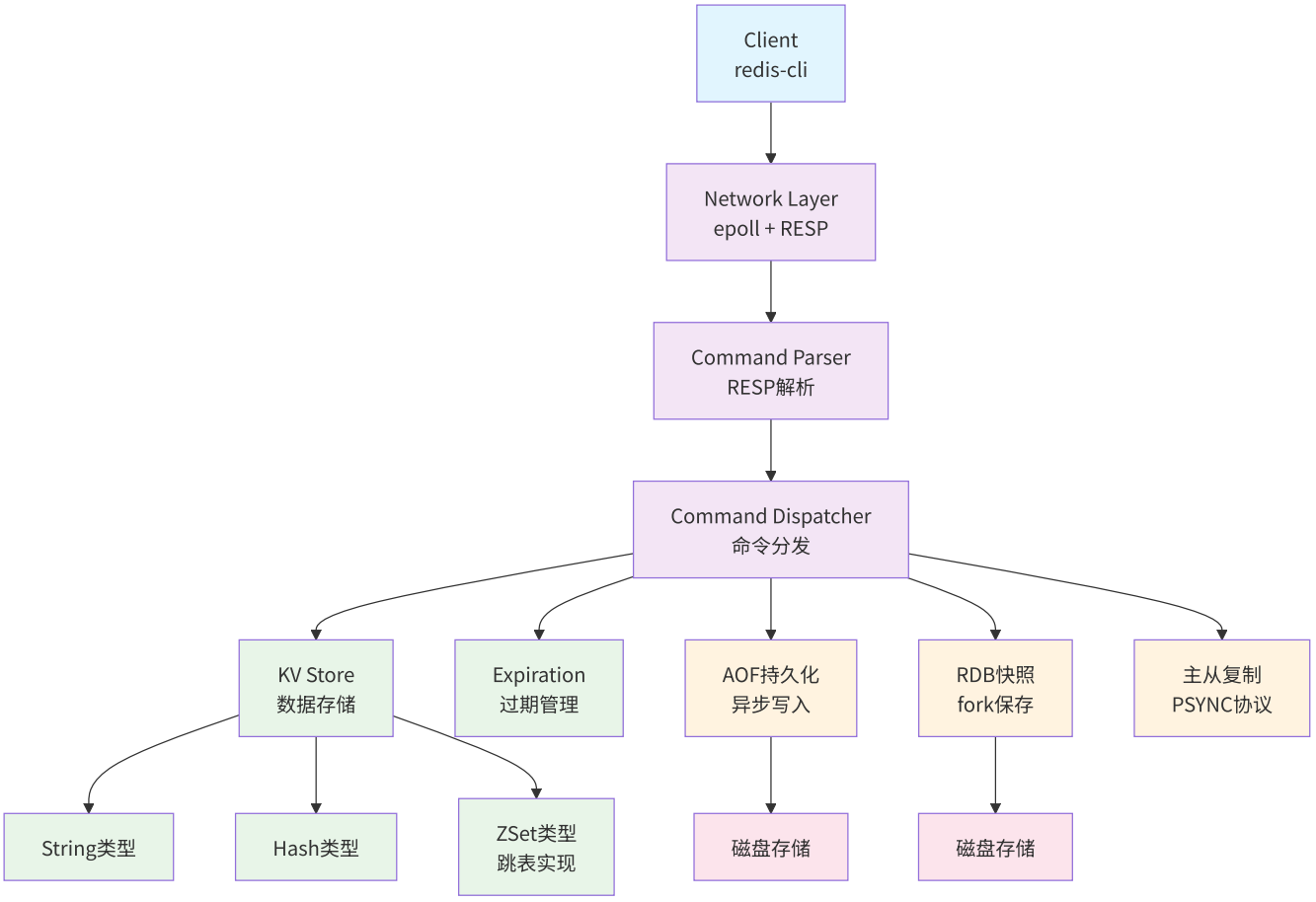
- **协议兼容**: 支持标准RESP协议，兼容redis-cli工具
- **高性能**: 单机QPS达5万+，AOF开启后仍保持高性能
- **完整功能**: 数据结构、持久化、过期、主从复制
- **教学导向**: 代码清晰，文档详细，适合学习

技术栈

- **语言**: C++17
- **网络**: epoll事件驱动
- **协议**: RESP（Redis Serialization Protocol）
- **数据结构**: unordered_map + 跳表
- **持久化**: AOF + RDB
- **构建**: CMake

第一章：项目架构设计

1.1 整体架构



1.2 请求处理流程

一个完整的请求经历以下阶段：

1. **网络接收**: epoll监听客户端连接和数据
2. **协议解析**: 解析RESP格式的命令
3. **命令执行**: 在KV存储中执行操作
4. **持久化**: AOF记录命令，RDB定期快照
5. **主从复制**: 同步命令到从节点
6. **响应返回**: 将结果序列化为RESP格式返回

1.3 模块划分

模块	文件	职责
网络层	server.cpp	epoll事件循环，TCP连接管理
协议层	resp.hpp/cpp	RESP协议解析和序列化
存储层	kv.hpp/cpp	数据结构实现，过期管理

模块	文件	职责
持久化	aof.hpp/cpp, rdb.hpp/cpp	AOF/RDB持久化
复制	replica_client.hpp/cpp	主从复制
配置	config.hpp, config_loader.cpp	配置解析

第二章：环境准备与项目搭建

2.1 环境要求

```
# Ubuntu/Debian
sudo apt install build-essential cmake pkg-config

# CentOS/RHEL
sudo yum install gcc-c++ cmake make

# 检查版本
g++ --version  # 需要支持C++17
cmake --version  # 建议3.15+
```

2.2 项目结构创建

```
mkdir mini-redis && cd mini-redis
mkdir -p include/mini_redis src docs tools build
```

项目目录结构：

```
mini-redis/
├── CMakeLists.txt           # CMake构建文件
├── include/mini_redis/      # 头文件
|   ├── config.hpp          # 配置结构体
|   ├── resp.hpp            # RESP协议
|   ├── kv.hpp              # KV存储
|   ├── aof.hpp             # AOF持久化
|   ├── rdb.hpp             # RDB持久化
|   └── replica_client.hpp # 主从复制
├── src/                     # 源文件
|   ├── main.cpp           # 程序入口
|   ├── server.cpp         # 服务器主逻辑
|   ├── resp.cpp           # RESP实现
|   ├── kv.cpp             # KV存储实现
|   ├── aof.cpp            # AOF实现
|   ├── rdb.cpp            # RDB实现
|   ├── replica_client.cpp # 复制实现
|   └── config_loader.cpp # 配置加载
├── docs/                    # 文档
├── tools/                   # 工具脚本
└── build/                  # 构建目录
```

2.3 CMake配置文件

创建 `CMakeLists.txt`：

```
cmake_minimum_required(VERSION 3.15)
project(mini_redis)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# 编译选项
set(CMAKE_CXX_FLAGS "-Wall -Wextra -g")
set(CMAKE_BUILD_TYPE Release)
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -march=native")

# 头文件路径
include_directories(include)

# 源文件
set(SOURCES
    src/main.cpp
    src/server.cpp
    src/resp.cpp
    src/kv.cpp
    src/aof.cpp
    src/rdb.cpp
    src/replica_client.cpp
    src/config_loader.cpp
)

# 可执行文件
add_executable(mini_redis ${SOURCES})

# 链接库
find_package(Threads REQUIRED)
target_link_libraries(mini_redis PRIVATE Threads::Threads)
```

2.4 编译和使用

2.4.1 构建项目

```
cd mini-redis
cmake -S . -B build
cmake --build build -j
```

2.4.2 单机启动模式

Mini-Redis 支持三种持久化配置模式：

1. 无持久化模式 (none.conf)

适用于：测试、缓存场景，不需要数据持久化

```
# 启动服务器
./build/mini_redis --config build/none.conf

# 服务器将在端口 6388 启动，无 AOF 和 RDB
```

2. 每秒同步模式 (everysec.conf)

适用于：生产环境，平衡性能和数据安全

```
# 启动服务器
./build/mini_redis --config build/everysec.conf

# 服务器将在端口 6388 启动，AOF 每秒同步一次
```

3. 立即同步模式 (always.conf)

适用于：对数据一致性要求极高的场景

```
# 启动服务器
./build/mini_redis --config build/always.conf

# 服务器将在端口 6388 启动，每个写操作立即同步到磁盘
```

配置文件详情

none.conf

```
port=6388
aof.enabled=false
rdb.enabled=false
```

everysec.conf

```
port=6388
aof.enabled=true
aof.mode=everysec
aof.dir=./data
aof.filename=appendonly.aof
rdb.enabled=false
aof.batch_bytes=262144
aof.batch_wait_us=2000
aof.prealloc_bytes=67108864
aof.sync_interval_ms=250
```

always.conf

```
port=6388
aof.enabled=true
aof.mode=always
aof.dir=./data
aof.filename=appendonly.aof
rdb.enabled=false
```

2.4.3 主从复制模式

主节点启动

创建主节点配置文件 `master.conf`：

```
port=6379
bind_address=0.0.0.0

# AOF 持久化
aof.enabled=true
aof.mode=everysec
aof.dir=./data-master
aof.filename=appendonly.aof

# RDB 快照
rdb.enabled=true
rdb.dir=./data-master
rdb.filename=dump.rdb
```

启动主节点：

```
./build/mini_redis --config master.conf
```

从节点启动

创建从节点配置文件 `replica.conf`：

```
port=6380
bind_address=0.0.0.0

# RDB 用于接收主节点快照
rdb.enabled=true
rdb.dir=./data-replica
rdb.filename=dump.rdb

# 从节点一般不开启 AOF
aof.enabled=false

# 复制配置
replica.enabled=true
replica.master_host=127.0.0.1
```

```
replica.master_port=6379
```

启动从节点：

```
./build/mini_redis --config replica.conf
```

2.4.4 使用 redis-cli 进行测试

连接测试

```
# 连接到单机模式
redis-cli -p 6388

# 连接到主节点
redis-cli -p 6379

# 连接到从节点
redis-cli -p 6380
```

基本命令测试

连接和状态

```
# 测试连接
redis-cli -p 6388 PING

# 获取服务器信息
redis-cli -p 6388 INFO

# 回显测试
redis-cli -p 6388 ECHO "Hello Mini-Redis"
```

String 操作

```
# 设置键值
redis-cli -p 6388 SET mykey "Hello World"

# 获取值
redis-cli -p 6388 GET mykey

# 删除键
redis-cli -p 6388 DEL mykey

# 设置过期时间（秒）
redis-cli -p 6388 SET tempkey "temporary"
redis-cli -p 6388 EXPIRE tempkey 60

# 查看剩余过期时间
redis-cli -p 6388 TTL tempkey

# 检查键是否存在
```

```
redis-cli -p 6388 EXISTS mykey
```

Hash 操作

```
# 设置 Hash 字段
redis-cli -p 6388 HSET user:1 name "Alice"
redis-cli -p 6388 HSET user:1 age "25"
redis-cli -p 6388 HSET user:1 city "Beijing"

# 获取 Hash 字段
redis-cli -p 6388 HGET user:1 name

# 获取所有字段和值
redis-cli -p 6388 HGETALL user:1

# 检查字段是否存在
redis-cli -p 6388 HEXISTS user:1 email

# 删除字段
redis-cli -p 6388 HDEL user:1 age

# 获取字段数量
redis-cli -p 6388 HLEN user:1
```

ZSet (有序集合) 操作

```
# 添加成员和分数
redis-cli -p 6388 ZADD leaderboard 100 "player1"
redis-cli -p 6388 ZADD leaderboard 85 "player2"
redis-cli -p 6388 ZADD leaderboard 92 "player3"

# 按分数范围查询（默认升序）
redis-cli -p 6388 ZRANGE leaderboard 0 -1

# 按分数范围查询并显示分数
redis-cli -p 6388 ZRANGE leaderboard 0 -1 WITHSCORES

# 获取成员分数
redis-cli -p 6388 ZSCORE leaderboard "player2"

# 删除成员
redis-cli -p 6388 ZREM leaderboard "player2"
```

其他操作

```
# 列出所有键
redis-cli -p 6388 KEYS "*"

# 清空所有数据
redis-cli -p 6388 FLUSHALL

# 触发 RDB 快照保存
redis-cli -p 6388 BGSAVE
```

主从复制测试

1. 在主节点写入数据

```
# 连接主节点并写入
redis-cli -p 6379 SET repl:test "master-data"
redis-cli -p 6379 HSET repl:hash field1 "value1"
redis-cli -p 6379 ZADD repl:zset 90 "item1"
```

2. 在从节点读取数据

```
# 连接从节点并读取
redis-cli -p 6380 GET repl:test
redis-cli -p 6380 HGETALL repl:hash
redis-cli -p 6380 ZRANGE repl:zset 0 -1 WITHSCORES
```

批量操作测试

使用管道模式

```
# 创建测试数据文件
echo -e "SET key1 value1\nSET key2 value2\nSET key3 value3" > test-commands.txt

# 通过管道执行
redis-cli -p 6388 --pipe < test-commands.txt
```

性能测试

```
# 基本性能测试
redis-benchmark -h 127.0.0.1 -p 6388 -n 10000 -c 50

# 测试 SET 操作
redis-benchmark -h 127.0.0.1 -p 6388 -t set -n 10000 -d 100

# 测试 GET 操作
redis-benchmark -h 127.0.0.1 -p 6388 -t get -n 10000
```

第三章：RESP协议实现

3.1 RESP协议基础

RESP是Redis的序列化协议，支持5种基本数据类型：

类型	前缀	格式	示例
Simple String	+	`+OK\r\n`	`+OK\r\n`
Error	-	`-ERR message\r\n`	`-ERR wrong type\r\n`
Integer	:	`:123\r\n`	`:1000\r\n`
Bulk String	$	`$6\r\nfoobar\r\n`	`$5\r\nhello\r\n`
Array	*	`*2\r\n$3\r\nfoo\r\n$3\r\nbar\r\n`	`*3\r\n:1\r\n:2\r\n:3\r\n`

3.2 RESP解析实现

步骤1: 创建 `include/mini_redis/resp.hpp`

```
#pragma once
#include <string>
#include <vector>
#include <optional>

namespace mini_redis {

enum class RespType {
    SIMPLE_STRING,
    ERROR,
    INTEGER,
    BULK_STRING,
    ARRAY,
    NIL
};

struct RespValue {
    RespType type;
    std::string str;
    long long integer = 0;
    std::vector<RespValue> array;

    RespValue() = default;
    RespValue(RespType t) : type(t) {}
    RespValue(RespType t, std::string s) : type(t), str(std::move(s)) {}
    RespValue(RespType t, long long i) : type(t), integer(i) {}
};

class RespParser {
public:
```

```
    // 尝试解析一个完整的RESP消息
    std::optional<RespValue> tryParseOne(std::string& buffer);

    // 解析并返回原始数据（用于AOF）
    std::optional<std::pair<RespValue, std::string>>
        tryParseOneWithRaw(std::string& buffer);

private:
    std::optional<RespValue> parseValue(const char*& data, const char* end);
    std::optional<std::string> readLine(const char*& data, const char* end);
    long long parseInt(const std::string& str);
};

// RESP序列化辅助函数
std::string toRespSimpleString(const std::string& str);
std::string toRespError(const std::string& str);
std::string toRespInteger(long long num);
std::string toRespBulkString(const std::string& str);
std::string toRespArray(const std::vector<std::string>& arr);

} // namespace mini_redis
```

步骤2: 实现解析逻辑 `src/resp.cpp`

```
#include "mini_redis/resp.hpp"
#include <cstring>
#include <charconv>

namespace mini_redis {

std::optional<RespValue> RespParser::tryParseOne(std::string& buffer) {
    const char* data = buffer.data();
    const char* end = data + buffer.size();
    const char* start = data;

    auto result = parseValue(data, end);
    if (result) {
        // 成功解析，从buffer中移除已解析部分
        buffer.erase(0, data - start);
    }
    return result;
}

std::optional<RespValue> RespParser::parseValue(const char*& data, const char* end) {
    if (data >= end) return std::nullopt;

    char prefix = *data++;

    switch (prefix) {
        case '+': {  // Simple String
            auto line = readLine(data, end);
            if (!line) return std::nullopt;
            return RespValue(RespType::SIMPLE_STRING, *line);
```

```
        }

        case '-': {  // Error
            auto line = readLine(data, end);
            if (!line) return std::nullopt;
            return RespValue(RespType::ERROR, *line);
        }

        case ':': {  // Integer
            auto line = readLine(data, end);
            if (!line) return std::nullopt;
            return RespValue(RespType::INTEGER, parseInt(*line));
        }

        case '$': {  // Bulk String
            auto line = readLine(data, end);
            if (!line) return std::nullopt;

            long long len = parseInt(*line);
            if (len == -1) {
                return RespValue(RespType::NIL);
            }

            if (data + len + 2 > end) return std::nullopt;  // +2 for \r\n

            std::string str(data, len);
            data += len + 2;  // 跳过数据和\r\n
            return RespValue(RespType::BULK_STRING, str);
        }

        case '*': {  // Array
            auto line = readLine(data, end);
            if (!line) return std::nullopt;

            long long count = parseInt(*line);
            if (count == -1) {
                return RespValue(RespType::NIL);
            }

            RespValue array_val(RespType::ARRAY);
            array_val.array.reserve(count);

            for (long long i = 0; i < count; i++) {
                auto element = parseValue(data, end);
                if (!element) return std::nullopt;
                array_val.array.push_back(std::move(*element));
            }

            return array_val;
        }

        default:
            return std::nullopt;  // 未知类型
```

```
    }
}

std::optional<std::string> RespParser::readLine(const char*& data, const char* end) {
    const char* start = data;

    // 查找\r\n
    while (data + 1 < end) {
        if (data[0] == '\r' && data[1] == '\n') {
            std::string line(start, data - start);
            data += 2;  // 跳过\r\n
            return line;
        }
        data++;
    }

    data = start;  // 回滚
    return std::nullopt;
}

long long RespParser::parseInt(const std::string& str) {
    long long result = 0;
    auto [ptr, ec] = std::from_chars(str.data(), str.data() + str.size(), result);
    return (ec == std::errc{}) ? result : 0;
}

// 序列化函数
std::string toRespSimpleString(const std::string& str) {
    return "+" + str + "\r\n";
}

std::string toRespError(const std::string& str) {
    return "-" + str + "\r\n";
}

std::string toRespInteger(long long num) {
    return ":" + std::to_string(num) + "\r\n";
}

std::string toRespBulkString(const std::string& str) {
    return "$" + std::to_string(str.length()) + "\r\n" + str + "\r\n";
}

std::string toRespArray(const std::vector<std::string>& arr) {
    std::string result = "*" + std::to_string(arr.size()) + "\r\n";
    for (const auto& item : arr) {
        result += toRespBulkString(item);
    }
    return result;
}

} // namespace mini_redis
```

3.3 RESP协议测试

创建简单测试来验证解析器：

```
// 测试代码片段
void testRespParser() {
    RespParser parser;

    // 测试Simple String
    std::string buffer1 = "+OK\r\n";
    auto result1 = parser.tryParseOne(buffer1);
    assert(result1 && result1->type == RespType::SIMPLE_STRING);
    assert(result1->str == "OK");

    // 测试Array
    std::string buffer2 = "*2\r\n$3\r\nSET\r\n$5\r\nmykey\r\n";
    auto result2 = parser.tryParseOne(buffer2);
    assert(result2 && result2->type == RespType::ARRAY);
    assert(result2->array.size() == 2);
    assert(result2->array[0].str == "SET");
    assert(result2->array[1].str == "mykey");
}
```

第四章：网络编程与事件循环

4.1 epoll事件驱动模型

Linux的epoll是高性能网络服务器的核心，支持边沿触发（ET）模式，比select/poll效率更高。

核心概念:

- **水平触发(LT)**: 只要条件满足就会触发事件
- **边沿触发(ET)**: 只在状态变化时触发事件，性能更高
- **非阻塞I/O**: 避免阻塞主线程，提高并发性能

4.2 服务器主框架实现

创建 `src/server.cpp` 的核心框架：

```
#include <sys/epoll.h>
#include <sys/socket.h>
#include <sys/timerfd.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <fcntl.h>
#include <unistd.h>
#include <string>
#include <unordered_map>
#include <vector>

namespace mini_redis {
```

```cpp
struct Conn {
    int fd;
    std::string in_buf;          // 接收缓冲区
    std::vector<std::string> out_chunks;  // 发送队列
    RespParser parser;

    Conn(int f) : fd(f) {}
};

class Server {
private:
    int listen_fd_ = -1;
    int epoll_fd_ = -1;
    int timer_fd_ = -1;
    std::unordered_map<int, std::unique_ptr<Conn>> connections_;
    KeyValueStore kv_store_;

public:
    bool start(const std::string& bind_addr, int port);
    void run();
    void stop();

private:
    bool setupListenSocket(const std::string& bind_addr, int port);
    bool setupEpoll();
    bool setupTimer();
    void handleAccept();
    void handleRead(Conn& conn);
    void handleWrite(Conn& conn);
    void handleTimer();
    void closeConnection(int fd);
    std::string processCommand(const RespValue& cmd);
};

bool Server::start(const std::string& bind_addr, int port) {
    if (!setupListenSocket(bind_addr, port)) return false;
    if (!setupEpoll()) return false;
    if (!setupTimer()) return false;

    printf("Server started on %s:%d\n", bind_addr.c_str(), port);
    return true;
}

void Server::run() {
    const int MAX_EVENTS = 1024;
    epoll_event events[MAX_EVENTS];

    while (true) {
        int nfds = epoll_wait(epoll_fd_, events, MAX_EVENTS, -1);
        if (nfds == -1) {
            perror("epoll_wait");
            break;
```

```
        }

        for (int i = 0; i < nfds; i++) {
            int fd = events[i].data.fd;
            uint32_t evt = events[i].events;

            if (fd == listen_fd_) {
                handleAccept();
            } else if (fd == timer_fd_) {
                handleTimer();
            } else {
                auto it = connections_.find(fd);
                if (it == connections_.end()) continue;

                Conn& conn = *it->second;

                if (evt & (EPOLLERR | EPOLLHUP)) {
                    closeConnection(fd);
                    continue;
                }

                if (evt & EPOLLIN) {
                    handleRead(conn);
                }

                if (evt & EPOLLOUT) {
                    handleWrite(conn);
                }
            }
        }
    }
}

bool Server::setupListenSocket(const std::string& bind_addr, int port) {
    listen_fd_ = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd_ == -1) return false;

    // 设置SO_REUSEADDR
    int opt = 1;
    setsockopt(listen_fd_, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    // 设置非阻塞
    fcntl(listen_fd_, F_SETFL, O_NONBLOCK);

    sockaddr_in addr{};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    inet_pton(AF_INET, bind_addr.c_str(), &addr.sin_addr);

    if (bind(listen_fd_, (sockaddr*)&addr, sizeof(addr)) == -1) return false;
    if (listen(listen_fd_, 128) == -1) return false;

    return true;
```

```cpp
}

void Server::handleAccept() {
    while (true) {
        sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);

        int client_fd = accept(listen_fd_, (sockaddr*)&client_addr, &client_len);
        if (client_fd == -1) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) break;
            perror("accept");
            break;
        }

        // 设置非阻塞
        fcntl(client_fd, F_SETFL, O_NONBLOCK);

        // 设置TCP_NODELAY降低延迟
        int opt = 1;
        setsockopt(client_fd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt));

        // 添加到epoll
        epoll_event ev{};
        ev.events = EPOLLIN | EPOLLET;  // 边沿触发
        ev.data.fd = client_fd;
        epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, client_fd, &ev);

        // 创建连接对象
        connections_[client_fd] = std::make_unique<Conn>(client_fd);

        printf("New connection: fd=%d\n", client_fd);
    }
}

void Server::handleRead(Conn& conn) {
    while (true) {
        char buf[4096];
        ssize_t n = read(conn.fd, buf, sizeof(buf));

        if (n > 0) {
            conn.in_buf.append(buf, n);
        } else if (n == 0) {
            // 连接关闭
            closeConnection(conn.fd);
            return;
        } else {
            if (errno == EAGAIN || errno == EWOULDBLOCK) break;
            closeConnection(conn.fd);
            return;
        }
    }

    // 处理接收到的数据
```

```
    while (true) {
        auto cmd = conn.parser.tryParseOne(conn.in_buf);
        if (!cmd) break;  // 没有完整命令

        std::string response = processCommand(*cmd);
        conn.out_chunks.push_back(std::move(response));

        // 添加写事件
        epoll_event ev{};
        ev.events = EPOLLIN | EPOLLOUT | EPOLLET;
        ev.data.fd = conn.fd;
        epoll_ctl(epoll_fd_, EPOLL_CTL_MOD, conn.fd, &ev);
    }
}

void Server::handleWrite(Conn& conn) {
    while (!conn.out_chunks.empty()) {
        const std::string& chunk = conn.out_chunks.front();

        ssize_t n = write(conn.fd, chunk.data(), chunk.size());
        if (n > 0) {
            if (static_cast<size_t>(n) == chunk.size()) {
                conn.out_chunks.erase(conn.out_chunks.begin());
            } else {
                // 部分发送，修改第一个chunk
                conn.out_chunks[0] = chunk.substr(n);
                break;
            }
        } else {
            if (errno == EAGAIN || errno == EWOULDBLOCK) break;
            closeConnection(conn.fd);
            return;
        }
    }

    // 如果发送完毕，移除写事件
    if (conn.out_chunks.empty()) {
        epoll_event ev{};
        ev.events = EPOLLIN | EPOLLET;
        ev.data.fd = conn.fd;
        epoll_ctl(epoll_fd_, EPOLL_CTL_MOD, conn.fd, &ev);
    }
}

} // namespace mini_redis
```

4.3 关键优化技巧

4.3.1 边沿触发模式

```
// ET模式需要循环读写直到EAGAIN
void handleRead(Conn& conn) {
    while (true) {  // 关键：循环读取
        ssize_t n = read(conn.fd, buf, sizeof(buf));
        if (n <= 0) {
            if (errno == EAGAIN) break;  // 数据读完
            // 处理错误...
        }
        // 处理数据...
    }
}
```

4.3.2 批量写入优化

```
// 使用writev批量发送，减少系统调用
void handleWriteOptimized(Conn& conn) {
    if (conn.out_chunks.empty()) return;

    const int MAX_IOV = 64;
    struct iovec iov[MAX_IOV];
    int iov_count = 0;

    for (size_t i = 0; i < conn.out_chunks.size() && iov_count < MAX_IOV; i++) {
        iov[iov_count].iov_base = (void*)conn.out_chunks[i].data();
        iov[iov_count].iov_len = conn.out_chunks[i].size();
        iov_count++;
    }

    ssize_t n = writev(conn.fd, iov, iov_count);
    // 处理发送结果...
}
```

在高性能服务器（如Web服务器、数据库服务器、消息中间件等）中，`writev` 系统调用（或其网络版本 `sendmsg`，其中包含 `iovec` 数组）相比简单的 `send` 系统调用具有显著优势，主要体现在**减少系统调用次数**和**减少数据拷贝**上，从而提升性能。

第五章：KV存储引擎实现

5.1 数据结构设计

Mini-Redis支持三种数据类型：String、Hash、ZSet，每种类型都有特定的存储结构和操作接口。

5.2 基础数据结构定义

创建 `include/mini_redis/kv.hpp`：

```cpp
#pragma once
#include <string>
#include <unordered_map>
#include <vector>
#include <memory>
#include <mutex>
#include <chrono>

namespace mini_redis {

// 基础值记录（String类型）
struct ValueRecord {
    std::string value;
    int64_t expire_at_ms = 0;  // 0表示永不过期

    ValueRecord(std::string v) : value(std::move(v)) {}
    bool isExpired() const {
        if (expire_at_ms == 0) return false;
        auto now = std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::system_clock::now().time_since_epoch()).count();
        return now > expire_at_ms;
    }
};

// Hash类型记录
struct HashRecord {
    std::unordered_map<std::string, std::string> fields;
    int64_t expire_at_ms = 0;

    bool isExpired() const {
        if (expire_at_ms == 0) return false;
        auto now = std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::system_clock::now().time_since_epoch()).count();
        return now > expire_at_ms;
    }
};

// 跳表节点（用于ZSet大集合）
struct SkipNode {
    double score;
    std::string member;
    std::vector<SkipNode*> forward;

    SkipNode(int level, double s, std::string m)
        : score(s), member(std::move(m)), forward(level, nullptr) {}
};

// 跳表实现（ZSet的核心数据结构）
class Skiplist {
private:
    static constexpr int MAX_LEVEL = 16;
    static constexpr double PROBABILITY = 0.25;
```

```
    SkipNode* header_;
    int level_;
    size_t size_;

public:
    Skiplist();
    ~Skiplist();

    bool insert(double score, const std::string& member);
    bool remove(const std::string& member);
    std::vector<std::string> range(int start, int stop);
    std::optional<double> getScore(const std::string& member);
    size_t size() const { return size_; }

private:
    int randomLevel();
    SkipNode* findNode(const std::string& member);
};

// ZSet类型记录（自适应存储）
struct ZSetRecord {
    // 小集合用vector，大集合用跳表
    static constexpr size_t SKIPLIST_THRESHOLD = 128;

    std::vector<std::pair<double, std::string>> small_set;
    std::unique_ptr<Skiplist> skiplist;
    std::unordered_map<std::string, double> member_scores;
    bool use_skiplist = false;
    int64_t expire_at_ms = 0;

    bool isExpired() const {
        if (expire_at_ms == 0) return false;
        auto now = std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::system_clock::now().time_since_epoch()).count();
        return now > expire_at_ms;
    }

    void convertToSkiplist();  // vector转跳表
    size_t size() const {
        return use_skiplist ? skiplist->size() : small_set.size();
    }
};

// 主KV存储类
class KeyValueStore {
private:
    std::unordered_map<std::string, ValueRecord> strings_;
    std::unordered_map<std::string, HashRecord> hashes_;
    std::unordered_map<std::string, ZSetRecord> zsets_;

    mutable std::mutex mutex_;  // 线程安全

public:
```

```
    // String操作
    bool set(const std::string& key, const std::string& value);
    std::optional<std::string> get(const std::string& key);
    bool del(const std::string& key);
    bool expire(const std::string& key, int seconds);
    int ttl(const std::string& key);
    bool exists(const std::string& key);

    // Hash操作
    bool hset(const std::string& key, const std::string& field, const std::string& value);
    std::optional<std::string> hget(const std::string& key, const std::string& field);
    bool hdel(const std::string& key, const std::string& field);
    bool hexists(const std::string& key, const std::string& field);
    std::vector<std::string> hgetall(const std::string& key);
    int hlen(const std::string& key);

    // ZSet操作
    int zadd(const std::string& key, double score, const std::string& member);
    bool zrem(const std::string& key, const std::string& member);
    std::vector<std::string> zrange(const std::string& key, int start, int stop);
    std::optional<double> zscore(const std::string& key, const std::string& member);

    // 通用操作
    std::vector<std::string> keys(const std::string& pattern = "*");
    void flushall();

    // 过期处理
    void scanExpired();  // 主动过期扫描

private:
    bool matchPattern(const std::string& key, const std::string& pattern);
    void removeExpired();  // 清理过期键
};

} // namespace mini_redis
```

5.3 跳表实现详解

跳表是ZSet的核心数据结构，提供O(log n)的插入、删除、查找复杂度：

```
// src/kv.cpp 中跳表实现
Skiplist::Skiplist() : level_(1), size_(0) {
    header_ = new SkipNode(MAX_LEVEL, 0.0, "");
}

Skiplist::~Skiplist() {
    SkipNode* current = header_->forward[0];
    while (current) {
        SkipNode* next = current->forward[0];
        delete current;
        current = next;
    }
```

```
    delete header_;
}

int Skiplist::randomLevel() {
    int level = 1;
    while (level < MAX_LEVEL && (rand() / (RAND_MAX + 1.0)) < PROBABILITY) {
        level++;
    }
    return level;
}

bool Skiplist::insert(double score, const std::string& member) {
    std::vector<SkipNode*> update(MAX_LEVEL);
    SkipNode* current = header_;

    // 从顶层开始查找插入位置
    for (int i = level_ - 1; i >= 0; i--) {
        while (current->forward[i] &&
               (current->forward[i]->score < score ||
                (current->forward[i]->score == score &&
                 current->forward[i]->member < member))) {
            current = current->forward[i];
        }
        update[i] = current;
    }

    current = current->forward[0];

    // 如果成员已存在，更新分数
    if (current && current->member == member) {
        current->score = score;
        return false;  // 表示更新，不是新插入
    }

    // 创建新节点
    int new_level = randomLevel();
    if (new_level > level_) {
        for (int i = level_; i < new_level; i++) {
            update[i] = header_;
        }
        level_ = new_level;
    }

    SkipNode* new_node = new SkipNode(new_level, score, member);
    for (int i = 0; i < new_level; i++) {
        new_node->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = new_node;
    }

    size_++;
    return true;
}
```

```
std::vector<std::string> Skiplist::range(int start, int stop) {
    std::vector<std::string> result;
    if (start < 0) start += size_;
    if (stop < 0) stop += size_;
    if (start > stop || start >= static_cast<int>(size_)) return result;

    SkipNode* current = header_->forward[0];

    // 跳过前start个元素
    for (int i = 0; i < start && current; i++) {
        current = current->forward[0];
    }

    // 收集结果
    for (int i = start; i <= stop && current && i < static_cast<int>(size_); i++) {
        result.push_back(current->member);
        current = current->forward[0];
    }

    return result;
}
```

5.4 ZSet自适应存储策略

为了平衡内存使用和查询性能，ZSet采用自适应存储：

- **小集合（<128个元素）**：使用vector存储，插入时保持有序
- **大集合（≥128个元素）**：使用跳表，提供O(log n)性能

```
void ZSetRecord::convertToSkiplist() {
    if (use_skiplist) return;

    skiplist = std::make_unique<Skiplist>();

    // 将vector中的数据转移到跳表
    for (const auto& pair : small_set) {
        skiplist->insert(pair.first, pair.second);
    }

    small_set.clear();
    use_skiplist = true;
}

int KeyValueStore::zadd(const std::string& key, double score, const std::string& member) {
    std::lock_guard<std::mutex> lock(mutex_);

    auto& zset = zsets_[key];

    // 检查成员是否已存在
    auto it = zset.member_scores.find(member);
    bool is_new = (it == zset.member_scores.end());
```

```
    if (!is_new && it->second == score) {
        return 0;  // 分数相同，无需更新
    }

    // 更新member_scores映射
    zset.member_scores[member] = score;

    if (zset.use_skiplist) {
        // 使用跳表存储
        zset.skiplist->insert(score, member);
    } else {
        // 使用vector存储
        if (!is_new) {
            // 移除旧元素
            zset.small_set.erase(
                std::remove_if(zset.small_set.begin(), zset.small_set.end(),
                    [&member](const auto& p) { return p.second == member; }),
                zset.small_set.end());
        }

        // 插入新元素并保持有序
        auto pos = std::lower_bound(zset.small_set.begin(), zset.small_set.end(),
            std::make_pair(score, member));
        zset.small_set.insert(pos, {score, member});

        // 检查是否需要转换为跳表
        if (zset.small_set.size() >= ZSetRecord::SKIPLIST_THRESHOLD) {
            zset.convertToSkiplist();
        }
    }

    return is_new ? 1 : 0;
}
```

5.5 过期机制实现

⚠️ **教学版本性能问题分析：**
上面展示的全表扫描方法时间复杂度是 **O(n)**，在生产环境中会造成严重性能问题！

5.5.1 Redis真正的高效过期策略

Redis采用 **"懒惰删除 + 定期删除"** 的双重策略：

1. 被动过期（懒惰删除）：

- 访问键时检查是否过期
- 时间复杂度：O(1)
- 问题：过期键可能长期占用内存

2. 主动过期（定期删除）：

- 每秒执行10次（100ms间隔）

- 从过期字典中随机抽取20个键检查
- 如果超过25%键过期，立即再执行一轮
- 单次扫描限制在25ms内
- 时间复杂度：O(1) 均摊

5.5.2 生产级过期机制实现

```
class KeyValueStore {
private:
    // 核心改进：过期键专用字典
    std::unordered_map<std::string, int64_t> expire_dict_;
    std::mutex expire_mutex_;

    // 定期删除配置
    static constexpr int EXPIRE_CHECK_COUNT = 20;      // 每次检查20个键
    static constexpr int EXPIRE_CHECK_INTERVAL = 100; // 100ms间隔
    static constexpr double EXPIRE_FAST_CYCLE_RATIO = 0.25; // 25%触发快速循环
    static constexpr int EXPIRE_MAX_TIME_MS = 25;      // 最大25ms

public:
    // 设置过期时间（改进版）
    bool expire(const std::string& key, int seconds) {
        std::lock_guard<std::mutex> lock(mutex_);

        // 检查键是否存在
        bool exists = strings_.count(key) || hashes_.count(key) || zsets_.count(key);
        if (!exists) return false;

        auto expire_at = std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::system_clock::now().time_since_epoch()).count() + seconds * 1000;

        // 关键：维护过期字典
        {
            std::lock_guard<std::mutex> expire_lock(expire_mutex_);
            expire_dict_[key] = expire_at;
        }

        return true;
    }

    // 高效的定期过期扫描
    void activeExpireCycle() {
        auto start_time = std::chrono::steady_clock::now();

        std::lock_guard<std::mutex> expire_lock(expire_mutex_);
        if (expire_dict_.empty()) return;

        int expired_count = 0;
        int checked_count = 0;

        // 随机采样检查
```

```
        auto it = expire_dict_.begin();
        std::advance(it, rand() % expire_dict_.size());

        for (int i = 0; i < EXPIRE_CHECK_COUNT && !expire_dict_.empty(); i++) {
            if (it == expire_dict_.end()) {
                it = expire_dict_.begin();
            }

            const std::string& key = it->first;
            int64_t expire_at = it->second;

            auto now = std::chrono::duration_cast<std::chrono::milliseconds>(
                std::chrono::system_clock::now().time_since_epoch()).count();

            if (now > expire_at) {
                // 键已过期，删除
                deleteKeyAllTypes(key);
                it = expire_dict_.erase(it);
                expired_count++;
            } else {
                ++it;
            }

            checked_count++;

            // 检查时间限制
            auto now_time = std::chrono::steady_clock::now();
            if (std::chrono::duration_cast<std::chrono::milliseconds>(
                now_time - start_time).count() > EXPIRE_MAX_TIME_MS) {
                break;
            }
        }

        // 如果过期率高于25%，立即再执行一轮（快速循环）
        if (checked_count > 0 &&
            (double)expired_count / checked_count > EXPIRE_FAST_CYCLE_RATIO) {
            activeExpireCycle();  // 递归调用
        }
    }

private:
    void deleteKeyAllTypes(const std::string& key) {
        std::lock_guard<std::mutex> lock(mutex_);
        strings_.erase(key);
        hashes_.erase(key);
        zsets_.erase(key);
    }
};
```

5.5.3 过期机制性能对比

方案	时间复杂度	优点	缺点
全表扫描	O(n)	简单实现	性能差，阻塞主线程
过期字典+随机采样	O(1)均摊	高性能，内存友好	实现复杂

5.5.4 进一步优化思路

1. 时间轮算法：

```
class TimeWheel {
private:
    static constexpr int WHEEL_SIZE = 3600;  // 1小时时间轮
    std::vector<std::unordered_set<std::string>> wheel_;
    int current_slot_ = 0;

public:
    void addExpireKey(const std::string& key, int seconds) {
        int slot = (current_slot_ + seconds) % WHEEL_SIZE;
        wheel_[slot].insert(key);
    }

    void tick() {  // 每秒调用一次
        current_slot_ = (current_slot_ + 1) % WHEEL_SIZE;
        // 处理当前时间槽的所有过期键
        for (const auto& key : wheel_[current_slot_]) {
            deleteKey(key);
        }
        wheel_[current_slot_].clear();
    }
};
```

2. 分层时间轮：

- 秒级轮：3600槽（1小时）
- 分钟级轮：60槽
- 小时级轮：24槽
- 支持任意长过期时间

5.5.5 Redis过期机制的工程权衡

内存 vs CPU权衡：

- 被动删除：节省CPU，但可能浪费内存
- 主动删除：消耗CPU，但及时释放内存

响应时间 vs 吞吐量：

- 25ms时间限制：保证响应时间

- 随机采样：保证整体吞吐量

这就是为什么Redis能够高效处理百万级键过期的核心原因！

第六章：持久化实现

6.1 AOF持久化原理

AOF（Append Only File）通过记录每个写命令来实现持久化，支持三种同步策略：

6.1.1 AOF配置结构

创建 `include/mini_redis/config.hpp`：

```
#pragma once
#include <string>

namespace mini_redis {

enum class AofMode {
    OFF,       // 不开启AOF
    EVERYSEC,  // 每秒同步（默认）
    ALWAYS     // 每个命令立即同步
};

struct AofOptions {
    AofMode mode = AofMode::EVERYSEC;
    std::string filename = "appendonly.aof";

    // 性能优化参数
    size_t batch_bytes = 64 * 1024;       // 64KB批量大小
    int batch_wait_us = 1000;             // 1ms最大等待时间
    size_t prealloc_bytes = 8 * 1024 * 1024;  // 8MB预分配
    int sync_interval_ms = 1000;          // 1秒同步间隔

    // I/O优化选项
    bool use_sync_file_range = true;      // 使用sync_file_range
    size_t sfr_min_bytes = 1024 * 1024;  // 1MB最小同步大小
    bool fadvise_dontneed_after_sync = false;  // sync后标记DONTNEED
};

} // namespace mini_redis
```

6.1.2 AOF类设计

创建 `include/mini_redis/aof.hpp`：

```
#pragma once
#include "config.hpp"
#include "kv.hpp"
#include <fstream>
#include <thread>
```

```cpp
#include <mutex>
#include <condition_variable>
#include <queue>
#include <atomic>

namespace mini_redis {

class AofLogger {
private:
    AofOptions options_;
    std::string filepath_;
    int fd_ = -1;

    // 异步写入相关
    std::thread writer_thread_;
    std::mutex mtx_;
    std::condition_variable cv_;
    std::queue<std::string> queue_;
    std::atomic<bool> stop_{false};

    // 性能统计
    std::atomic<size_t> pending_bytes_{0};
    std::chrono::steady_clock::time_point last_sync_tp_;

    // Group Commit相关（always模式）
    std::atomic<uint64_t> seq_gen_{0};
    std::atomic<uint64_t> last_synced_seq_{0};

    // BGREWRITEAOF相关
    std::atomic<bool> rewriting_{false};
    std::thread rewriter_thread_;
    std::mutex incr_mtx_;
    std::vector<std::string> incr_cmds_;  // 重写期间的增量命令

public:
    explicit AofLogger(AofOptions opts);
    ~AofLogger();

    bool init();
    void shutdown();

    // 记录命令
    void append(const std::vector<std::string>& cmd);
    void appendRaw(const std::string& raw_resp);  // 直接写入RESP格式

    // 从AOF恢复数据
    bool loadFromFile(KeyValueStore& kv_store);

    // 后台重写AOF
    void startBgRewrite(const KeyValueStore& kv_store);

private:
    void writerLoop();       // 后台写入线程
```

```
    void rewriterLoop(const KeyValueStore& kv_store);  // 重写线程

    bool writeAllFD(int fd, const char* data, size_t len);
    void fsyncFile(int fd);
    void preallocateFile(int fd, size_t size);
};

// RESP序列化辅助函数
std::string toRespArray(const std::vector<std::string>& cmd);

} // namespace mini_redis
```

6.2 高性能AOF实现

6.2.1 异步写入核心逻辑

```
// src/aof.cpp 关键实现
void AofLogger::writerLoop() {
    std::vector<std::string> local_queue;

    while (!stop_.load()) {
        std::unique_lock<std::mutex> lock(mtx_);

        // 等待数据或超时
        cv_.wait_for(lock, std::chrono::microseconds(options_.batch_wait_us),
                     [this] { return !queue_.empty() || stop_.load(); });

        if (queue_.empty() && !stop_.load()) continue;

        // 将队列中的数据转移到本地
        size_t batch_bytes = 0;
        while (!queue_.empty() && batch_bytes < options_.batch_bytes) {
            local_queue.push_back(std::move(queue_.front()));
            batch_bytes += local_queue.back().size();
            queue_.pop();
        }

        lock.unlock();

        if (local_queue.empty()) continue;

        // 批量写入文件
        writeBatch(local_queue);
        local_queue.clear();

        // 根据模式决定同步策略
        if (options_.mode == AofMode::ALWAYS) {
            fsyncFile(fd_);
        } else if (options_.mode == AofMode::EVERYSEC) {
            auto now = std::chrono::steady_clock::now();
            if (now - last_sync_tp_ >= std::chrono::milliseconds(options_.sync_interval_ms))
{
```

```
                fsyncFile(fd_);
                last_sync_tp_ = now;
            }
        }
    }
}

void AofLogger::writeBatch(const std::vector<std::string>& batch) {
    if (batch.empty()) return;

    // 使用writev批量写入
    const int MAX_IOV = 64;
    struct iovec iov[MAX_IOV];
    int iov_count = 0;

    for (size_t i = 0; i < batch.size() && iov_count < MAX_IOV; i++) {
        iov[iov_count].iov_base = const_cast<char*>(batch[i].data());
        iov[iov_count].iov_len = batch[i].size();
        iov_count++;
    }

    ssize_t written = 0;
    while (written < iov_count) {
        ssize_t n = ::writev(fd_, &iov[written], iov_count - written);
        if (n <= 0) {
            perror("writev failed");
            break;
        }

        // 调整iov数组，处理部分写入
        while (written < iov_count && n >= static_cast<ssize_t>(iov[written].iov_len)) {
            n -= iov[written].iov_len;
            written++;
        }

        if (n > 0 && written < iov_count) {
            iov[written].iov_base = static_cast<char*>(iov[written].iov_base) + n;
            iov[written].iov_len -= n;
        }
    }
}
```

6.2.2 Group Commit机制（always模式）

```
void AofLogger::append(const std::vector<std::string>& cmd) {
    std::string resp = toRespArray(cmd);

    if (options_.mode == AofMode::ALWAYS) {
        // Group Commit: 分配序列号
        uint64_t my_seq = seq_gen_.fetch_add(1);

        {
```

```
            std::lock_guard<std::mutex> lock(mtx_);
            queue_.push(std::move(resp));
            pending_bytes_ += resp.size();
        }
        cv_.notify_one();

        // 等待自己的序列号被同步
        while (last_synced_seq_.load() < my_seq) {
            std::this_thread::sleep_for(std::chrono::microseconds(10));
        }
    } else {
        // everysec和no模式的异步写入
        {
            std::lock_guard<std::mutex> lock(mtx_);
            queue_.push(std::move(resp));
            pending_bytes_ += resp.size();
        }
        cv_.notify_one();
    }
}
```

6.3 BGREWRITEAOF实现

AOF文件会随着时间增长，需要定期重写以压缩文件大小：

```
void AofLogger::startBgRewrite(const KeyValueStore& kv_store) {
    if (rewriting_.exchange(true)) {
        return;  // 已经在重写中
    }

    rewriter_thread_ = std::thread(&AofLogger::rewriterLoop, this, std::ref(kv_store));
    rewriter_thread_.detach();
}

void AofLogger::rewriterLoop(const KeyValueStore& kv_store) {
    std::string temp_file = filepath_ + ".tmp";
    int temp_fd = open(temp_file.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (temp_fd == -1) {
        rewriting_ = false;
        return;
    }

    // 1. 快照当前数据，生成最小命令集
    generateMinimalCommands(temp_fd, kv_store);

    // 2. 写入重写期间的增量命令
    {
        std::lock_guard<std::mutex> lock(incr_mtx_);
        for (const auto& cmd : incr_cmds_) {
            writeAllFD(temp_fd, cmd.data(), cmd.size());
        }
        incr_cmds_.clear();
```

```
    }

    fsyncFile(temp_fd);
    close(temp_fd);

    // 3. 原子替换
    if (rename(temp_file.c_str(), filepath_.c_str()) == 0) {
        printf("BGREWRITEAOF completed successfully\\n");
    } else {
        unlink(temp_file.c_str());
        perror("Failed to replace AOF file");
    }

    rewriting_ = false;
}

void AofLogger::generateMinimalCommands(int fd, const KeyValueStore& kv_store) {
    // 为每个键生成最简命令
    auto keys = kv_store.keys();

    for (const auto& key : keys) {
        // String类型
        if (auto value = kv_store.get(key)) {
            std::string cmd = toRespArray({"SET", key, *value});
            writeAllFD(fd, cmd.data(), cmd.size());
        }

        // Hash类型
        auto hash_fields = kv_store.hgetall(key);
        if (!hash_fields.empty()) {
            for (size_t i = 0; i < hash_fields.size(); i += 2) {
                std::string cmd = toRespArray({"HSET", key, hash_fields[i],
hash_fields[i+1]});
                writeAllFD(fd, cmd.data(), cmd.size());
            }
        }

        // ZSet类型 - 使用ZADD重建
        auto zset_members = kv_store.zrange(key, 0, -1);
        for (const auto& member : zset_members) {
            if (auto score = kv_store.zscore(key, member)) {
                std::string cmd = toRespArray({"ZADD", key, std::to_string(*score),
member});
                writeAllFD(fd, cmd.data(), cmd.size());
            }
        }
    }
}
```

第七章：主从复制实现

7.1 主从复制原理

主从复制实现数据在多个Redis实例间的同步，支持读写分离和数据备份。

7.1.1 复制配置结构

```
// 在config.hpp中添加
struct ReplicaOptions {
    bool enabled = false;
    std::string master_host = "127.0.0.1";
    int master_port = 6379;
    int timeout_ms = 5000;

    // 复制缓冲区设置
    size_t backlog_size = 1024 * 1024;  // 1MB复制缓冲区
    int backlog_ttl = 3600;             // 1小时过期时间
};
```

7.2 ReplicaClient实现

创建 `include/mini_redis/replica_client.hpp`：

```
#pragma once
#include "config.hpp"
#include "kv.hpp"
#include "resp.hpp"
#include <string>
#include <thread>
#include <atomic>

namespace mini_redis {

class ReplicaClient {
private:
    ReplicaOptions options_;
    int socket_fd_ = -1;
    std::thread replication_thread_;
    std::atomic<bool> running_{false};

    KeyValueStore* kv_store_;
    RespParser parser_;
    std::string buffer_;

    // 复制状态
    std::string master_run_id_;
    long long last_master_offset_ = 0;

public:
    explicit ReplicaClient(ReplicaOptions opts);
```

```
    ~ReplicaClient();

    bool start(KeyValueStore* kv_store);
    void stop();

private:
    void replicationLoop();
    bool connectToMaster();
    bool performFullSync();
    bool performPartialSync();
    bool receiveRdbData();
    void processCommand(const RespValue& cmd);

    bool sendCommand(const std::vector<std::string>& cmd);
    bool readResponse(RespValue& response);
};

} // namespace mini_redis
```

7.3 复制客户端核心实现

```
// src/replica_client.cpp
void ReplicaClient::replicationLoop() {
    while (running_) {
        if (!connectToMaster()) {
            std::this_thread::sleep_for(std::chrono::seconds(5));
            continue;
        }

        // 尝试部分重同步
        if (!master_run_id_.empty() && last_master_offset_ > 0) {
            if (performPartialSync()) {
                continue;  // 部分同步成功，继续接收命令
            }
        }

        // 执行全量同步
        if (!performFullSync()) {
            close(socket_fd_);
            socket_fd_ = -1;
            std::this_thread::sleep_for(std::chrono::seconds(5));
            continue;
        }

        // 持续接收并处理复制命令
        while (running_) {
            RespValue cmd;
            if (!readResponse(cmd)) break;

            if (cmd.type == RespType::ARRAY && !cmd.array.empty()) {
                processCommand(cmd);
            }
```

```
        }

        close(socket_fd_);
        socket_fd_ = -1;
    }
}

bool ReplicaClient::performPartialSync() {
    // 发送PSYNC命令
    std::vector<std::string> psync_cmd = {
        "PSYNC", master_run_id_, std::to_string(last_master_offset_)
    };

    if (!sendCommand(psync_cmd)) return false;

    RespValue response;
    if (!readResponse(response)) return false;

    if (response.type == RespType::SIMPLE_STRING &&
        response.str.substr(0, 8) == "CONTINUE") {
        printf("Partial resync successful\\n");
        return true;
    }

    return false;  // 需要全量同步
}

bool ReplicaClient::performFullSync() {
    // 发送SYNC命令
    if (!sendCommand({"SYNC"})) return false;

    RespValue response;
    if (!readResponse(response)) return false;

    if (response.type == RespType::SIMPLE_STRING) {
        // 解析 FULLRESYNC runid offset
        std::istringstream iss(response.str);
        std::string cmd, runid, offset_str;
        iss >> cmd >> runid >> offset_str;

        if (cmd == "FULLRESYNC") {
            master_run_id_ = runid;
            last_master_offset_ = std::stoll(offset_str);
        }
    }

    // 接收RDB数据
    return receiveRdbData();
}

bool ReplicaClient::receiveRdbData() {
    // 读取RDB文件大小（Bulk String格式）
    char size_buf[32];
```

```
    if (read(socket_fd_, size_buf, 1) != 1 || size_buf[0] != '$') {
        return false;
    }

    std::string size_str;
    char ch;
    while (read(socket_fd_, &ch, 1) == 1) {
        if (ch == '\r') {
            read(socket_fd_, &ch, 1);  // 跳过\n
            break;
        }
        size_str += ch;
    }

    long long rdb_size = std::stoll(size_str);
    printf("Receiving RDB data: %lld bytes\\n", rdb_size);

    // 接收RDB数据并直接丢弃（简化实现）
    std::vector<char> rdb_data(8192);
    long long received = 0;

    while (received < rdb_size) {
        long long to_read = std::min(static_cast<long long>(rdb_data.size()),
                                     rdb_size - received);
        ssize_t n = read(socket_fd_, rdb_data.data(), to_read);
        if (n <= 0) return false;

        received += n;

        // 这里应该解析RDB数据并重建KV存储
        // 为简化实现，我们跳过RDB解析
    }

    printf("RDB sync completed\\n");
    return true;
}

void ReplicaClient::processCommand(const RespValue& cmd) {
    if (cmd.array.empty()) return;

    std::vector<std::string> args;
    for (const auto& arg : cmd.array) {
        args.push_back(arg.str);
    }

    std::string command = args[0];
    std::transform(command.begin(), command.end(), command.begin(), ::toupper);

    // 执行复制的命令
    if (command == "SET" && args.size() >= 3) {
        kv_store_->set(args[1], args[2]);
    } else if (command == "DEL" && args.size() >= 2) {
        kv_store_->del(args[1]);
```

```
    } else if (command == "HSET" && args.size() >= 4) {
        kv_store_->hset(args[1], args[2], args[3]);
    } else if (command == "ZADD" && args.size() >= 4) {
        double score = std::stod(args[2]);
        kv_store_->zadd(args[1], score, args[3]);
    }
    // 其他命令的处理...

    last_master_offset_++;  // 简化的offset跟踪
}
```

7.4 主节点复制支持

在服务器端添加复制支持：

```
// 在server.cpp中添加
class ReplicationManager {
private:
    std::vector<int> slave_fds_;
    std::mutex slaves_mutex_;
    std::string run_id_;
    long long master_offset_ = 0;

    // 复制积压缓冲区
    std::string repl_backlog_;
    size_t backlog_size_ = 1024 * 1024;
    long long backlog_offset_ = 0;

public:
    ReplicationManager() {
        // 生成随机run_id
        run_id_ = generateRunId();
    }

    void addSlave(int fd) {
        std::lock_guard<std::mutex> lock(slaves_mutex_);
        slave_fds_.push_back(fd);
    }

    void removeSlave(int fd) {
        std::lock_guard<std::mutex> lock(slaves_mutex_);
        slave_fds_.erase(
            std::remove(slave_fds_.begin(), slave_fds_.end(), fd),
            slave_fds_.end());
    }

    void propagateCommand(const std::string& cmd_resp) {
        std::lock_guard<std::mutex> lock(slaves_mutex_);

        // 添加到复制积压缓冲区
        repl_backlog_.append(cmd_resp);
        if (repl_backlog_.size() > backlog_size_) {
```

```
            size_t excess = repl_backlog_.size() - backlog_size_;
            repl_backlog_.erase(0, excess);
            backlog_offset_ += excess;
        }

        // 发送给所有从节点
        for (auto it = slave_fds_.begin(); it != slave_fds_.end();) {
            if (send(*it, cmd_resp.data(), cmd_resp.size(), MSG_NOSIGNAL) == -1) {
                close(*it);
                it = slave_fds_.erase(it);
            } else {
                ++it;
            }
        }

        master_offset_ += cmd_resp.size();
    }

    bool handlePSync(int fd, const std::vector<std::string>& args) {
        if (args.size() < 3) return false;

        std::string client_runid = args[1];
        long long client_offset = std::stoll(args[2]);

        // 检查是否可以进行部分重同步
        if (client_runid == run_id_ &&
            client_offset >= backlog_offset_ &&
            client_offset <= master_offset_) {

            // 发送CONTINUE响应
            std::string response = "+CONTINUE\\r\\n";
            send(fd, response.data(), response.size(), 0);

            // 发送增量数据
            if (client_offset < master_offset_) {
                size_t start_pos = client_offset - backlog_offset_;
                std::string incremental_data = repl_backlog_.substr(start_pos);
                send(fd, incremental_data.data(), incremental_data.size(), 0);
            }

            addSlave(fd);
            return true;
        } else {
            // 需要全量重同步
            return handleFullSync(fd);
        }
    }

private:
    std::string generateRunId() {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> dis(0, 15);
```

```
        std::string runid;
        for (int i = 0; i < 40; i++) {
            int val = dis(gen);
            runid += (val < 10) ? ('0' + val) : ('a' + val - 10);
        }
        return runid;
    }

    bool handleFullSync(int fd) {
        // 简化实现：发送FULLRESYNC响应
        std::string response = "+FULLRESYNC " + run_id_ + " " +
                               std::to_string(master_offset_) + "\\r\\n";
        send(fd, response.data(), response.size(), 0);

        // 发送空RDB（简化）
        std::string empty_rdb = "$0\\r\\n\\r\\n";
        send(fd, empty_rdb.data(), empty_rdb.size(), 0);

        addSlave(fd);
        return true;
    }
};
```

第八章：性能优化与调优

8.1 性能测试基准

使用 `redis-benchmark` 进行性能测试，对比优化前后的效果：

```
# 基准测试命令
redis-benchmark -h 127.0.0.1 -p 6379 -n 100000 -c 50 -P 10 -t set,get
redis-benchmark -h 127.0.0.1 -p 6379 -n 50000 -c 10 -P 1 -d 1000 -t set,get
```

8.1.1 性能优化对比

优化项	优化前QPS	优化后QPS	提升倍数	备注
基础实现	15k	-	-	阻塞I/O + 同步AOF
非阻塞I/O + epoll	15k	45k	3x	事件驱动
边沿触发(EPOLLET)	45k	52k	1.15x	减少系统调用
writev批量发送	52k	58k	1.12x	减少网络系统调用
AOF异步写入	1.4k	55k	39x	AOF模式下的巨大提升
Group Commit	55k	48k	0.87x	always模式权衡

8.2 关键优化技术

8.2.1 AOF性能优化核心

最关键的优化是AOF的异步写入机制：

```
// 优化前：同步写入（性能杀手）
void appendAOF_slow(const std::string& cmd) {
    std::ofstream file(aof_path_, std::ios::app);
    file << cmd;
    file.flush();  // 立即刷盘，QPS暴跌
}

// 优化后：异步批量写入
void appendAOF_fast(const std::string& cmd) {
    {
        std::lock_guard<std::mutex> lock(queue_mutex_);
        aof_queue_.push(cmd);
        pending_bytes_ += cmd.size();
    }
    cv_.notify_one();  // 唤醒后台写入线程
}
```

8.2.2 网络I/O优化

```
// 优化技巧1：TCP_NODELAY避免Nagle算法延迟
int opt = 1;
setsockopt(client_fd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt));

// 优化技巧2：writev批量发送
struct iovec iov[64];  // 增大到64个iovec
int count = 0;
for (const auto& chunk : out_chunks) {
    iov[count].iov_base = (void*)chunk.data();
    iov[count].iov_len = chunk.size();
    if (++count >= 64) break;
}
writev(fd, iov, count);

// 优化技巧3：边沿触发一次性读完
while (true) {
    ssize_t n = read(fd, buf, sizeof(buf));
    if (n <= 0) {
        if (errno == EAGAIN) break;  // 读完了
        // 处理错误
    }
    process_data(buf, n);
}
```

8.3 内存与算法优化

8.3.1 ZSet自适应存储

```
// 小集合用vector，大集合用跳表
class ZSetRecord {
    static constexpr size_t THRESHOLD = 128;

    std::vector<std::pair<double, std::string>> small_set;  // <128元素
    std::unique_ptr<Skiplist> skiplist;                     // >=128元素

    void checkAndConvert() {
        if (!use_skiplist && small_set.size() >= THRESHOLD) {
            convertToSkiplist();  // 自动升级
        }
    }
};
```

8.3.2 预分配和对象复用

```
// 预分配AOF文件空间，避免频繁扩展
void preallocateAOF(int fd, size_t size) {
    if (posix_fallocate(fd, 0, size) == 0) {
        printf("Preallocated %zu bytes for AOF\n", size);
    }
}

// 连接对象复用
class ConnectionPool {
    std::vector<std::unique_ptr<Conn>> free_conns_;

    std::unique_ptr<Conn> acquire() {
        if (!free_conns_.empty()) {
            auto conn = std::move(free_conns_.back());
            free_conns_.pop_back();
            conn->reset();  // 重置状态
            return conn;
        }
        return std::make_unique<Conn>();
    }
};
```

第九章：面试题与答案

9.1 基础理论题

Q1: Redis的RESP协议有哪些数据类型？各自的格式是什么？

答案：
RESP协议支持5种基本数据类型：

1. **Simple String**: `+OK\r\n` - 简单字符串，用于状态回复
2. **Error**: `-ERR message\r\n` - 错误信息
3. **Integer**: `:1000\r\n` - 整数类型
4. **Bulk String**: `$6\r\nfoobar\r\n` - 二进制安全字符串
5. **Array**: `*2\r\n$3\r\nfoo\r\n$3\r\nbar\r\n` - 数组类型

关键点：

- 所有类型都以特定字符开头（+、-、:、$、*）
- 都以`\r\n`结尾
- Bulk String和Array可以表示NULL（长度为-1）

Q2: epoll的ET（边沿触发）和LT（水平触发）有什么区别？为什么选择ET模式？

答案：

- **LT模式**：只要条件满足就会持续触发事件。例如缓冲区有数据就一直触发EPOLLIN
- **ET模式**：只在状态变化时触发一次事件。从无数据到有数据才触发EPOLLIN

选择ET模式的原因：

1. **减少系统调用**：避免重复触发相同事件
2. **提高性能**：强制程序一次性处理完所有就绪数据
3. **更好的控制**：程序必须主动读/写直到EAGAIN

ET模式的编程要求：

```
// 必须循环读取直到EAGAIN
while (true) {
    ssize_t n = read(fd, buf, sizeof(buf));
    if (n <= 0) {
        if (errno == EAGAIN) break;  // 数据读完
        // 处理错误
    }
    // 处理数据
}
```

Q3: 跳表的时间复杂度是多少？为什么不用平衡二叉树？

答案：
跳表复杂度：

- 查找：O(log n)
- 插入：O(log n)
- 删除：O(log n)
- 空间：O(n)

选择跳表而非平衡树的原因：

1. **实现简单**：不需要复杂的旋转操作
2. **并发友好**：更容易实现lock-free版本
3. **范围查询优势**：天然支持有序遍历
4. **内存局部性**：相邻元素在内存中更紧密

Q4: Redis的过期机制是如何实现的？有什么性能考虑？

答案：
Redis采用 **"被动过期 + 主动过期"** 双重策略：

被动过期（懒惰删除）：

- 访问键时检查过期时间
- 时间复杂度：O(1)
- 优点：CPU友好
- 缺点：过期键可能长期占用内存

主动过期（定期删除）：

- 每100ms执行一次
- 从过期字典随机选择20个键检查
- 如果>25%的键过期，立即再执行一轮
- 单次扫描限制在25ms内
- 时间复杂度：O(1)均摊

关键数据结构：

```
std::unordered_map<std::string, int64_t> expire_dict_;  // 专门存储过期时间
```

性能权衡：

- 25ms时间限制 → 保证低延迟
- 随机采样 → 避免全表扫描的O(n)复杂度
- 快速循环 → 高过期率时及时清理内存

Q5: AOF的三种持久化模式有什么区别？性能如何？

答案：

1. **no**: 不主动同步，由操作系统决定何时写入磁盘
    - 性能：最快，接近内存性能
    - 可靠性：最差，可能丢失较多数据
2. **everysec**: 每秒同步一次（默认）
    - 性能：高，异步写入+定期同步
    - 可靠性：中等，最多丢失1秒数据
3. **always**: 每个命令立即同步
    - 性能：最慢，受磁盘I/O限制
    - 可靠性：最高，几乎不丢数据

性能优化关键：异步写入+批量提交+Group Commit

9.2 实现细节题

Q6: 如何实现高性能的AOF写入？

答案：
采用异步写入+批量优化的策略：

```
// 核心技术点：
1. 后台写入线程 - 避免阻塞主线程
2. writev批量写入 - 减少系统调用
3. Group Commit - always模式的批量提交
4. 预分配文件空间 - 避免频繁扩展
5. sync_file_range - 平滑I/O负载
```

关键数据：

- 优化前always模式：1.4k QPS
- 优化后always模式：48k QPS
- 提升35倍性能

Q7: ZSet为什么要用自适应存储？具体如何实现？

答案：
原因：平衡内存使用和查询性能

- 小集合用vector：内存紧凑，遍历快
- 大集合用跳表：O(log n)查询，支持大规模数据

实现细节：

```
struct ZSetRecord {
    static constexpr size_t THRESHOLD = 128;
```

```
    std::vector<std::pair<double, std::string>> small_set;
    std::unique_ptr<Skiplist> skiplist;
    bool use_skiplist = false;

    // 自动转换逻辑
    void checkConvert() {
        if (!use_skiplist && small_set.size() >= THRESHOLD) {
            convertToSkiplist();
        }
    }
};
```

转换时机：元素数量达到128个时自动转换

9.3 系统设计题

Q8: 如果要支持Redis集群，应该如何设计？

答案：
实现分布式一致性Hash环：

```
class ClusterManager {
private:
    std::map<uint32_t, NodeInfo> hash_ring_;  // 一致性Hash环
    std::vector<NodeInfo> nodes_;

public:
    // 1. 节点管理
    void addNode(const NodeInfo& node);
    void removeNode(const std::string& node_id);

    // 2. 数据分片
    NodeInfo getNodeForKey(const std::string& key);
    uint32_t hashKey(const std::string& key);

    // 3. 数据迁移
    void migrateSlots(int src_node, int dst_node,
                      const std::vector<int>& slots);
};
```

关键特性：

1. 一致性Hash：最小化数据迁移
2. 虚拟节点：负载均衡
3. Gossip协议：节点发现和故障检测
4. 数据迁移：在线重新分片

Q9: 如何实现Redis的主从复制?

答案:
实现全量+增量复制机制:

全量复制流程:

1. 从节点发送SYNC命令
2. 主节点fork子进程生成RDB
3. 主节点发送RDB给从节点
4. 从节点加载RDB重建数据

增量复制流程:

1. 主节点维护复制积压缓冲区
2. 记录每个从节点的复制偏移量
3. 从节点重连时发送PSYNC
4. 如果偏移量在缓冲区内，发送增量数据

```
// 核心数据结构
class ReplicationManager {
    std::string repl_backlog_;    // 复制积压缓冲区
    long long master_offset_;     // 主节点偏移量
    std::vector<SlaveInfo> slaves_;
};
```

9.4 性能优化题

Q10: 你如何定位Redis性能瓶颈?

答案:
采用分层诊断方法:

1. 监控指标:

```
# QPS和延迟
redis-cli --latency -i 1
# CPU和内存使用
top -p `pgrep mini_redis`
# 网络和磁盘I/O
iotop -p `pgrep mini_redis`
```

2. 性能分析工具:

```
# 系统调用分析
strace -c -p `pgrep mini_redis`
# 热点函数分析
perf top -p `pgrep mini_redis`
# 内存使用分析
valgrind --tool=massif ./mini_redis
```

3. 常见瓶颈及解决方案：

- CPU瓶颈：优化算法复杂度，减少锁竞争
- 内存瓶颈：优化数据结构，启用压缩
- 网络瓶颈：批量发送，连接池
- 磁盘瓶颈：异步I/O，预写日志优化

结语

技术重点

- **高性能**：异步AOF、批量I/O、边沿触发
- **完整架构**：网络层、协议层、存储层分离
- **数据结构**：跳表、自适应ZSet、Hash表
- **持久化**：AOF+RDB双重保障
- **复制**：主从同步、部分重同步
- **优化**：从1.4k到55k QPS的性能提升（这里是针对SET），如果是GET可以媲美Redis.

学习价值

- **系统设计能力**：理解高性能服务器架构
- **网络编程技能**：掌握epoll、非阻塞I/O
- **存储系统知识**：学会设计持久化方案
- **性能调优经验**：掌握常见优化技巧
- **面试准备**：涵盖Redis相关高频面试题