

# ACL Redis 架构设计文档

## 目录

- [总体架构](#)
- [核心类设计](#)
- [连接管理](#)
- [命令处理流程](#)
- [集群支持](#)
- [内存管理](#)
- [线程安全](#)

## 总体架构

ACL Redis 采用分层的面向对象设计，主要分为以下几层：



## 核心类设计

### 1. redis\_command (基类)

`redis_command` 是所有 Redis 命令类的基类，提供核心功能：

```
class redis_command : public noncopyable {
public:
    // 构造函数
    redis_command();
    redis_command(redis_client* conn);
    redis_command(redis_client_cluster* cluster);
    redis_command(redis_client_pipeline* pipeline);

    // 连接管理
    void set_client(redis_client* conn);
    void set_cluster(redis_client_cluster* cluster);
    void set_pipeline(redis_client_pipeline* pipeline);

    // 命令重置
    void clear(bool save_slot = false);

    // 结果获取
    redis_result_t result_type() const;
    const char* result_status() const;
    const char* result_error() const;
    int result_number(bool* success = NULL) const;
    long long int result_number64(bool* success = NULL) const;

private:
    redis_client* conn_;           // 单机连接
    redis_client_cluster* cluster_; // 集群连接
    redis_client_pipeline* pipeline_; // Pipeline 连接
    dbuf_pool* dbuf_;             // 内存池
    int slot_;                     // 哈希槽
    string* request_buf_;          // 请求缓冲区
    const redis_result* result_;   // 结果对象
};
```

#### 核心职责：

- 管理 Redis 连接（单机/集群/Pipeline）
- 构建 Redis 协议格式的请求数据
- 解析 Redis 服务器响应
- 管理内存池和结果对象
- 计算和缓存哈希槽（集群模式）

## 2. redis\_result（结果对象）

封装 Redis 服务器返回的所有类型的结果：

```
typedef enum {
    REDIS_RESULT_UNKOWN, // 未知类型
    REDIS_RESULT_NIL,    // 空值
}
```

```

    REDIS_RESULT_ERROR,        // 错误
    REDIS_RESULT_STATUS,       // 状态 (如 OK)
    REDIS_RESULT_INTEGER,      // 整数
    REDIS_RESULT_STRING,       // 字符串
    REDIS_RESULT_ARRAY,        // 数组
} redis_result_t;

class redis_result : public noncopyable {
public:
    redis_result_t get_type() const;
    int get_integer(bool* success = NULL) const;
    long long int get_integer64(bool* success = NULL) const;
    const char* get_status() const;
    const char* get_error() const;
    const char* get(size_t i, size_t* len = NULL) const;
    const redis_result* get_child(size_t i) const;

private:
    redis_result_t result_type_;
    dbuf_pool* dbuf_;
    const char** argv_;          // 数据数组
    size_t* lens_;              // 长度数组
    const redis_result** children_; // 子结果数组
};

```

#### 设计特点:

- 使用内存池分配，无需手动释放
- 支持嵌套结构（数组套数组）
- 零拷贝设计，数据直接从网络缓冲区引用

### 3. 数据结构命令类

每种 Redis 数据结构都有专门的命令类:

```

// 字符串命令
class redis_string : virtual public redis_command {
public:
    bool set(const char* key, const char* value);
    bool get(const char* key, string& buf);
    bool incr(const char* key, long long int* result = NULL);
    // ... 更多字符串命令
};

// 哈希表命令
class redis_hash : virtual public redis_command {
public:
    bool hmset(const char* key, const std::map<string, string>& attrs);
    bool hmget(const char* key, const std::vector<string>& names,
               std::vector<string>* result = NULL);
    bool hgetall(const char* key, std::map<string, string>& result);
};

```

```

// ... 更多哈希表命令
};

// 其他数据结构命令类
class redis_list : virtual public redis_command { ... };
class redis_set : virtual public redis_command { ... };
class redis_zset : virtual public redis_command { ... };
class redis_key : virtual public redis_command { ... };
class redis_pubsub : virtual public redis_command { ... };
class redis_transaction : virtual public redis_command { ... };
class redis_script : virtual public redis_command { ... };
class redis_server : virtual public redis_command { ... };
class redis_geo : virtual public redis_command { ... };
class redis_stream : virtual public redis_command { ... };
class redis_hyperloglog : virtual public redis_command { ... };

```

#### 设计特点:

- 使用虚继承 (**virtual public**) 避免菱形继承问题
- 每个类专注于一种数据结构的操作
- 提供类型安全的参数和返回值

#### 4. redis (统一接口类)

通过多重继承提供所有命令的统一访问:

```

class redis
: public redis_connection
, public redis_hash
, public redis_hyperloglog
, public redis_key
, public redis_list
, public redis_pubsub
, public redis_script
, public redis_server
, public redis_set
, public redis_string
, public redis_transaction
, public redis_zset
, public redis_cluster
, public redis_geo
, public redis_stream
{
public:
    redis(redis_client* conn = NULL);
    redis(redis_client_cluster* cluster);
    redis(redis_client_pipeline* pipeline);
};

```

#### 优势:

- 用户只需创建一个对象即可使用所有命令
- 简化代码，无需创建多个命令对象
- 保持向后兼容性

## 连接管理

### 1. redis\_client（单机连接）

```
class redis_client : public connect_client {
public:
    redis_client(const char* addr, int conn_timeout = 60,
                 int rw_timeout = 30, bool retry = true);

    void set_password(const char* pass);
    void set_db(int dbnum);
    void set_ssl_conf(sslbase_conf* ssl_conf);

    socket_stream* get_stream(bool auto_connect = true);
    const redis_result* run(dbuf_pool* pool, const string& req,
                           size_t nchildren, int* rw_timeout = NULL);

private:
    socket_stream conn_;
    char* addr_;
    char* pass_;
    int dbnum_;
    sslbase_conf* ssl_conf_;
};
```

#### 功能：

- 管理与单个 Redis 服务器的连接
- 支持密码认证和数据库选择
- 支持 SSL/TLS 加密连接
- 继承自 `connect_client`，可用于连接池

### 2. redis\_client\_cluster（集群连接）

```
class redis_client_cluster : public connect_manager {
public:
    redis_client_cluster(int max_slot = 16384);

    void set_all_slot(const char* addr, size_t max_conns,
                     int conn_timeout = 30, int rw_timeout = 30);
    void set_slot(int slot, const char* addr);
    void clear_slot(int slot);

    redis_client_pool* peek_slot(int slot);
    redis_client* redirect(const char* addr, size_t max_conns);
};
```

```

        const redis_result* run(redis_command& cmd, size_t nchild,
                                int* timeout = NULL);

private:
    int max_slot_;           // 最大哈希槽数
    const char** slot_addrs_; // 槽位到地址映射
    int redirect_max_;        // 最大重定向次数
    int redirect_sleep_;      // 重定向等待时间
};

```

#### 功能：

- 管理 Redis 集群的所有节点连接
- 维护哈希槽到节点的映射关系
- 自动处理 MOVE 和 ASK 重定向
- 内置连接池管理
- 线程安全

### 3. redis\_client\_pipeline (Pipeline 连接)

```

class redis_client_pipeline {
public:
    redis_client_pipeline(const char* addr);

    void flush();
    const std::vector<redis_result*>& get_results();

private:
    redis_client* client_;
    std::vector<redis_request*> requests_;
    std::vector<redis_result*> results_;
};

```

#### 功能：

- 批量发送多个命令
- 批量接收响应结果
- 减少网络往返次数

## 命令处理流程

### 1. 命令构建

```

用户调用命令方法
↓
构建 Redis 协议格式
↓

```

```
*<参数个数>\r\n
$<参数1长度>\r\n<参数1>\r\n
$<参数2长度>\r\n<参数2>\r\n
...
↓
存储到 request_buf_
```

示例：SET key value 构建为：

```
*3\r\n
$3\r\nSET\r\n
$3\r\nkey\r\n
$5\r\nvalue\r\n
```

## 2. 单机模式处理流程

1. 用户调用命令方法（如 `cmd.set("key", "value")`）  
↓
2. `redis_string::set()` 调用 `build()` 构建请求  
↓
3. 调用 `run()` 发送请求并接收响应  
↓
4. `redis_client::run()` 执行网络 I/O  
↓
5. 解析响应创建 `redis_result` 对象  
↓
6. 返回结果给用户

## 3. 集群模式处理流程

1. 用户调用命令方法  
↓
2. 计算 key 的哈希槽：`CRC16(key) % 16384`  
↓
3. 根据哈希槽查找对应的 Redis 节点  
↓
4. 从连接池获取连接  
↓
5. 发送请求  
↓
6. 接收响应  
↓
7. 检查是否为 MOVE/ASK 重定向
  - ├ 是 → 更新槽位映射 → 重定向到新节点 → 重试
  - └ 否 → 返回结果

哈希槽计算：

```
void redis_command::hash_slot(const char* key, size_t len) {
    // 查找 {hashtag}
    const char* s = strchr(key, '{');
    if (s) {
        const char* e = strchr(s + 1, '}');
        if (e && e != s + 1) {
            key = s + 1;
            len = e - s - 1;
        }
    }

    // CRC16 计算
    slot_ = crc16(key, len) % 16384;
}
```

#### 4. Pipeline 模式处理流程

1. 批量构建多个命令请求  
↓
2. 一次性发送所有请求  
↓
3. 调用 flush() 发送数据  
↓
4. 批量接收所有响应  
↓
5. 解析所有结果  
↓
6. 用户遍历结果集

## 集群支持

### 1. 哈希槽映射

Redis 集群使用 16384 个哈希槽分配数据：

```
// 槽位到地址的映射
const char** slot_addrs_; // 16384 个元素的数组

// 设置槽位
void set_slot(int slot, const char* addr) {
    if (slot < 0 || slot >= max_slot_) return;
    slot_addrs_[slot] = strdup(addr);
}

// 获取槽位对应的连接池
redis_client_pool* peek_slot(int slot) {
```

```

    if (slot < 0 || slot >= max_slot_) return NULL;
    const char* addr = slot_addrs[slot];
    if (addr == NULL) return NULL;
    return (redis_client_pool*) get(addr);
}

```

## 2. 自动发现节点

```

void set_all_slot(const char* addr, size_t max_conns,
                 int conn_timeout, int rw_timeout) {
    // 连接任一节点
    redis_client client(addr, conn_timeout, rw_timeout);

    // 执行 CLUSTER SLOTS 命令
    redis_command cmd(&client);
    const redis_result* result = cmd.request(...);

    // 解析槽位分配信息
    // [[start_slot, end_slot, [ip, port], ...], ...]
    for (each slot_range in result) {
        int start = slot_range[0];
        int end = slot_range[1];
        const char* node_addr = slot_range[2];

        // 设置槽位映射
        for (int i = start; i <= end; i++) {
            set_slot(i, node_addr);
        }
    }
}

```

## 3. MOVE 重定向处理

当访问的 key 不在当前节点时，Redis 返回 MOVE 错误：

```
-MOVED 3999 127.0.0.1:7002
```

处理流程：

```

redis_client* move(redis_command& cmd, redis_client* conn,
                  const char* error_msg, int nretried) {
    // 解析 MOVE 信息
    // error_msg: "MOVED 3999 127.0.0.1:7002"
    int slot = parse_slot(error_msg);
    const char* addr = parse_addr(error_msg);

    // 更新槽位映射

```

```
    set_slot(slot, addr);

    // 重定向到新节点
    return redirect(addr, max_conns);
}
```

## 4. ASK 重定向处理

ASK 是临时重定向（槽位迁移中）：

```
-ASK 3999 127.0.0.1:7002
```

处理流程：

```
redis_client* ask(redis_command& cmd, redis_client* conn,
                  const char* error_msg, int nretried) {
    // 解析 ASK 信息
    const char* addr = parse_addr(error_msg);

    // 连接到新节点
    redis_client* new_conn = redirect(addr, max_conns);

    // 发送 ASKING 命令
    new_conn->run(..., "ASKING");

    // 重新发送原命令
    return new_conn;
}
```

## 5. 重定向控制

```
// 最大重定向次数（默认 15）
void set_redirect_max(int max);

// 重定向等待时间（默认 100 微秒）
void set_redirect_sleep(int n);

// 重定向重试间隔（默认 1 秒）
void set_retry_inter(int n);
```

## 内存管理

### 1. 内存池（dbuf\_pool）

ACL Redis 使用内存池管理所有动态分配的内存：

```

class dbuf_pool {
public:
    void* dbuf_alloc(size_t len);
    void* dbuf_calloc(size_t len);
    char* dbuf_strdup(const char* s);
    void dbuf_free(); // 释放整个池
};

```

#### 优势：

- 批量分配，减少系统调用
- 统一释放，避免内存泄漏
- 减少内存碎片
- 提高分配效率

## 2. 对象分配

```

// redis_result 使用内存池分配
void* redis_result::operator new(size_t size, dbuf_pool* pool) {
    return pool->dbuf_alloc(size);
}

// 字符串数据也在内存池中分配
const char* data = (const char*) dbuf->dbuf_alloc(len + 1);
memcpy((char*)data, buf, len);
((char*)data)[len] = '\0';

```

## 3. 生命周期管理

```

// 命令对象创建内存池
redis_command::redis_command() {
    dbuf_ = dbuf_create();
}

// 命令重置时清空内存池
void redis_command::clear(bool save_slot) {
    if (dbuf_) {
        dbuf_->dbuf_free();
    }
    result_ = NULL;
}

// 命令对象销毁时释放内存池
redis_command::~~redis_command() {
    if (dbuf_) {
        delete dbuf_;
    }
}

```

```
}  
}
```

## 线程安全

### 1. 单机模式

`redis_client` 本身不是线程安全的，但可以通过连接池实现多线程：

```
// 每个线程使用独立的命令对象  
void thread_func(redis_client_pool* pool) {  
    // 从池中获取连接  
    redis_client* conn = (redis_client*)pool->peek();  
  
    // 创建命令对象  
    redis_string cmd(conn);  
  
    // 执行操作  
    cmd.set("key", "value");  
  
    // 归还连接  
    pool->put(conn);  
}
```

### 2. 集群模式

`redis_client_cluster` 是线程安全的：

```
class redis_client_cluster : public connect_manager {  
private:  
    // 继承自 connect_manager 的锁  
    acl::thread_mutex lock_;  
  
    // 槽位更新时加锁  
    void set_slot(int slot, const char* addr) {  
        lock_.lock();  
        slot_addrs[slot] = ...;  
        lock_.unlock();  
    }  
};
```

多线程使用：

```
redis_client_cluster cluster;  
cluster.set_all_slot("127.0.0.1:7000", 100);  
  
// 多个线程共享集群对象
```

```

void thread_func(redis_client_cluster* cluster) {
    redis cmd;
    cmd.set_cluster(cluster);

    // 安全地执行操作
    cmd.set("key", "value");
}

```

### 3. Pipeline 模式

Pipeline 对象不是线程安全的，应该每个线程独立创建。

## 扩展性设计

### 1. 添加新命令

只需在相应的命令类中添加方法：

```

class redis_string : virtual public redis_command {
public:
    // 添加新命令
    bool set_with_get(const char* key, const char* value, string& old_val)
    {
        // Redis 6.2+ 支持 SET ... GET 选项
        build("SET", key, value, "GET");
        const redis_result* result = run();
        if (result && result->get_type() == REDIS_RESULT_STRING) {
            old_val = result->get(0);
            return true;
        }
        return false;
    }
};

```

### 2. 自定义命令

使用 `request()` 方法发送任意命令：

```

redis_command cmd(&client);

// 构建命令参数
std::vector<string> args;
args.push_back("CUSTOM");
args.push_back("arg1");
args.push_back("arg2");

// 发送并获取结果
const redis_result* result = cmd.request(args);

```

## 总结

ACL Redis 的架构设计具有以下特点：

1. **分层清晰**：网络层、连接层、命令层、接口层职责明确
2. **高性能**：内存池、连接池、Pipeline 等优化手段
3. **易用性**：统一接口、类型安全、自动内存管理
4. **可扩展**：虚继承、模板方法模式、开闭原则
5. **企业级**：集群支持、故障切换、线程安全

这使得 ACL Redis 成为一个生产级的 C++ Redis 客户端库。