

ACL Redis 示例代码

目录

- 基础示例
- 单机模式
- 集群模式
- Pipeline 模式
- 数据结构操作
- 高级特性

基础示例

Hello World

最简单的 Redis 操作示例。

```
#include "acl_cpp/lib_acl.hpp"

int main() {
    // 初始化 ACL 库 (Windows 必需)
    acl::acl_cpp_init();

    // 创建 Redis 客户端连接
    acl::redis_client client("127.0.0.1:6379", 10, 10);

    // 创建字符串命令对象
    acl::redis_string cmd(&client);

    // SET 命令
    if (!cmd.set("hello", "world")) {
        printf("SET 失败: %s\n", cmd.result_error());
        return 1;
    }
    printf("SET 成功\n");

    // GET 命令
    acl::string value;
    cmd.clear(); // 重置命令对象
    if (!cmd.get("hello", value)) {
        printf("GET 失败: %s\n", cmd.result_error());
        return 1;
    }
    printf("GET: %s\n", value.c_str());

    return 0;
}
```

使用统一接口

使用 `redis` 类访问所有命令。

```
#include "acl_cpp/lib_acl.hpp"

int main() {
    acl::acl_cpp_init();

    acl::redis_client client("127.0.0.1:6379", 10, 10);

    // 创建统一命令对象
    acl::redis cmd(&client);

    // 使用字符串命令
    cmd.set("name", "Alice");

    acl::string name;
    cmd.clear();
    cmd.get("name", name);
    printf("Name: %s\n", name.c_str());

    // 使用哈希命令
    std::map<acl::string, acl::string> user;
    user["name"] = "Bob";
    user["age"] = "25";
    user["city"] = "Beijing";

    cmd.clear();
    cmd.hmset("user:1001", user);

    // 使用列表命令
    cmd.clear();
    cmd.lpush("tasks", "task1");
    cmd.lpush("tasks", "task2");

    return 0;
}
```

单机模式

密码认证

```
#include "acl_cpp/lib_acl.hpp"

int main() {
    acl::acl_cpp_init();

    // 创建连接
    acl::redis_client client("127.0.0.1:6379", 10, 10);
```

```
// 设置密码
client.set_password("your_password");

// 选择数据库 (0-15)
client.set_db(1);

// 创建命令对象
acl::redis_string cmd(&client);

// 正常使用
cmd.set("key", "value");

return 0;
}
```

SSL/TLS 连接

```
#include "acl_cpp/lib_acl.hpp"
#include "acl_cpp/stream/polarssl_conf.hpp"

int main() {
    acl::acl_cpp_init();

    // 创建 SSL 配置
    acl::polarssl_conf ssl_conf;

    // 创建连接
    acl::redis_client client("127.0.0.1:6379", 10, 10);

    // 启用 SSL
    client.set_ssl_conf(&ssl_conf);

    // 正常使用
    acl::redis cmd(&client);
    cmd.set("key", "value");

    return 0;
}
```

连接池使用

```
#include "acl_cpp/lib_acl.hpp"

// 创建连接池
acl::redis_client_pool* create_pool(const char* addr) {
    acl::redis_client_pool* pool = new acl::redis_client_pool(
        addr,           // Redis 地址
        100,           // 最大连接数
        10,            // 连接超时时间 (毫秒)
        10,            // 重试次数
        1000);         // 重试间隔 (毫秒)
```

```
    10,           // 连接超时 (秒)
    10           // 读写超时 (秒)
);

// 可选: 设置密码
// pool->set_password("your_password");

return pool;
}

void worker_thread(acl::redis_client_pool* pool) {
    for (int i = 0; i < 1000; i++) {
        // 从池中获取连接
        acl::redis_client* client =
            (acl::redis_client*)pool->peek();

        if (client == NULL) {
            printf("获取连接失败\n");
            continue;
        }

        // 使用连接
        acl::redis_string cmd(client);
        acl::string key;
        key.format("key_%d", i);
        cmd.set(key.c_str(), "value");

        // 归还连接
        pool->put(client, true);
    }
}

int main() {
    acl::acl_cpp_init();

    // 创建连接池
    acl::redis_client_pool* pool = create_pool("127.0.0.1:6379");

    // 创建多个线程
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; i++) {
        threads.emplace_back(worker_thread, pool);
    }

    // 等待线程结束
    for (auto& t : threads) {
        t.join();
    }

    delete pool;
    return 0;
}
```

集群模式

基础使用

```
#include "acl_cpp/lib_acl.hpp"

int main() {
    acl::acl_cpp_init();

    // 创建集群对象
    acl::redis_client_cluster cluster;

    // 自动发现所有节点和槽位
    // 只需提供任意一个集群节点地址
    cluster.set_all_slot(
        "127.0.0.1:7000",   // 任一集群节点
        100,                // 每个连接池的最大连接数
        10,                 // 连接超时
        10                  // 读写超时
    );

    // 创建命令对象
    acl::redis cmd;
    cmd.set_cluster(&cluster);

    // 使用方式与单机模式完全相同
    cmd.set("user:1001", "Alice");

    acl::string name;
    cmd.clear();
    cmd.get("user:1001", name);
    printf("Name: %s\n", name.c_str());

    return 0;
}
```

集群密码认证

```
#include "acl_cpp/lib_acl.hpp"

int main() {
    acl::acl_cpp_init();

    acl::redis_client_cluster cluster;
    cluster.set_all_slot("127.0.0.1:7000", 100, 10, 10);

    // 设置所有节点的默认密码
    cluster.set_password("default", "your_password");

    // 或者为特定节点设置密码
```

```

cluster.set_password("127.0.0.1:7000", "password1");
cluster.set_password("127.0.0.1:7001", "password2");

acl::redis cmd;
cmd.set_cluster(&cluster);

// 正常使用
cmd.set("key", "value");

return 0;
}

```

多线程集群访问

```

#include "acl_cpp/lib_acl.hpp"
#include <thread>
#include <vector>

void worker_thread(acl::redis_client_cluster* cluster, int thread_id) {
    // 每个线程创建自己的命令对象
    acl::redis cmd;
    cmd.set_cluster(cluster);

    for (int i = 0; i < 1000; i++) {
        acl::string key, value;
        key.format("thread_%d_key_%d", thread_id, i);
        value.format("value_%d", i);

        // SET 操作
        if (!cmd.set(key.c_str(), value.c_str())) {
            printf("线程 %d SET 失败: %s\n",
                   thread_id, cmd.result_error());
            continue;
        }

        // GET 操作
        acl::string result;
        cmd.clear();
        if (!cmd.get(key.c_str(), result)) {
            printf("线程 %d GET 失败: %s\n",
                   thread_id, cmd.result_error());
            continue;
        }

        cmd.clear();
    }

    printf("线程 %d 完成\n", thread_id);
}

int main() {

```

```

acl::acl_cpp_init();

// 创建集群对象（线程安全）
acl::redis_client_cluster cluster;
cluster.set_all_slot("127.0.0.1:7000", 100, 10, 10);

// 配置重定向参数
cluster.set_redirect_max(15);      // 最大重定向次数
cluster.set_redirect_sleep(100);    // 重定向等待时间（微秒）

// 创建多个线程
std::vector<std::thread> threads;
for (int i = 0; i < 10; i++) {
    threads.emplace_back(worker_thread, &cluster, i);
}

// 等待所有线程完成
for (auto& t : threads) {
    t.join();
}

printf("所有线程完成\n");
return 0;
}

```

手动槽位管理

```

#include "acl_cpp/lib_acl.hpp"

int main() {
    acl::acl_cpp_init();

    acl::redis_client_cluster cluster;

    // 手动设置槽位映射（适用于已知集群拓扑的情况）
    // 槽位 0-5460 -> 127.0.0.1:7000
    for (int i = 0; i <= 5460; i++) {
        cluster.set_slot(i, "127.0.0.1:7000");
    }

    // 槽位 5461-10922 -> 127.0.0.1:7001
    for (int i = 5461; i <= 10922; i++) {
        cluster.set_slot(i, "127.0.0.1:7001");
    }

    // 槽位 10923-16383 -> 127.0.0.1:7002
    for (int i = 10923; i <= 16383; i++) {
        cluster.set_slot(i, "127.0.0.1:7002");
    }

    acl::redis cmd;

```

```

cmd.set_cluster(&cluster);

// 使用
cmd.set("key", "value");

return 0;
}

```

Pipeline 模式

基本 Pipeline

```

#include "acl_cpp/lib_acl.hpp"

int main() {
    acl::acl_cpp_init();

    // 创建 Pipeline 连接
    acl::redis_client_pipeline pipeline("127.0.0.1:6379");

    // 创建命令对象
    acl::redis cmd;
    cmd.set_pipeline(&pipeline);

    // 批量发送命令 (不立即获取结果)
    for (int i = 0; i < 100; i++) {
        acl::string key;
        key.format("key_%d", i);
        cmd.set(key.c_str(), "value");
        cmd.clear(true); // 保留 pipeline 状态
    }

    // 发送所有命令并接收所有响应
    pipeline.flush();

    // 获取所有结果
    const std::vector<acl::redis_result*>& results =
        pipeline.get_results();

    printf("总共 %zu 个结果\n", results.size());

    // 处理结果
    for (size_t i = 0; i < results.size(); i++) {
        const acl::redis_result* result = results[i];
        if (result->get_type() == acl::REDIS_RESULT_STATUS) {
            printf("命令 %zu: %s\n", i, result->get_status());
        } else if (result->get_type() == acl::REDIS_RESULT_ERROR) {
            printf("命令 %zu 错误: %s\n", i, result->get_error());
        }
    }
}

```

```
    return 0;  
}
```

Pipeline 批量读写

```
#include "acl_cpp/lib_acl.hpp"  
  
void pipeline_write(const char* addr, int count) {  
    acl::redis_client_pipeline pipeline(addr);  
    acl::redis_string cmd;  
    cmd.set_pipeline(&pipeline);  
  
    // 批量写入  
    for (int i = 0; i < count; i++) {  
        acl::string key, value;  
        key.format("key_%d", i);  
        value.format("value_%d", i);  
  
        cmd.set(key.c_str(), value.c_str());  
        cmd.clear(true);  
    }  
  
    pipeline.flush();  
  
    printf("写入 %d 条记录\n", count);  
}  
  
void pipeline_read(const char* addr, int count) {  
    acl::redis_client_pipeline pipeline(addr);  
    acl::redis_string cmd;  
    cmd.set_pipeline(&pipeline);  
  
    // 批量读取  
    for (int i = 0; i < count; i++) {  
        acl::string key;  
        key.format("key_%d", i);  
  
        cmd.get(key.c_str());  
        cmd.clear(true);  
    }  
  
    pipeline.flush();  
  
    // 处理结果  
    const std::vector<acl::redis_result*>& results =  
        pipeline.get_results();  
  
    int success = 0;  
    for (const auto* result : results) {  
        if (result->get_type() == acl::REDIS_RESULT_STRING) {  
            success++;  
        }  
    }  
}
```

```

        }
    }

    printf("成功读取 %d/%d 条记录\n", success, count);
}

int main() {
    acl::acl_cpp_init();

    const char* addr = "127.0.0.1:6379";
    int count = 10000;

    // 测量写入性能
    struct timeval start, end;
    gettimeofday(&start, NULL);
    pipeline_write(addr, count);
    gettimeofday(&end, NULL);

    double write_time = (end.tv_sec - start.tv_sec) * 1000.0 +
                       (end.tv_usec - start.tv_usec) / 1000.0;
    printf("写入耗时: %.2f ms, QPS: %.2f\n",
           write_time, count * 1000.0 / write_time);

    // 测量读取性能
    gettimeofday(&start, NULL);
    pipeline_read(addr, count);
    gettimeofday(&end, NULL);

    double read_time = (end.tv_sec - start.tv_sec) * 1000.0 +
                       (end.tv_usec - start.tv_usec) / 1000.0;
    printf("读取耗时: %.2f ms, QPS: %.2f\n",
           read_time, count * 1000.0 / read_time);

    return 0;
}

```

数据结构操作

String 操作

```

#include "acl_cpp/lib_acl.hpp"

void string_operations() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_string cmd(&client);

    // SET/GET
    cmd.set("name", "Alice");
    acl::string name;
    cmd.clear();
    cmd.get("name", name);
}

```

```

// SETEX (带过期时间)
cmd.clear();
cmd.setex("session:123", "token_abc", 3600); // 1小时过期

// SETNX (仅当不存在时设置)
cmd.clear();
int ret = cmd.setnx("lock:resource", "locked");
if (ret == 1) {
    printf("获取锁成功\n");
} else {
    printf("锁已被占用\n");
}

// INCR/DECR
cmd.clear();
long long int counter;
cmd.incr("counter", &counter);
printf("计数器: %lld\n", counter);

// APPEND
cmd.clear();
cmd.append("log", "new log entry\n");

// MGET/MSET
std::map<acl::string, acl::string> kvs;
kvs["key1"] = "value1";
kvs["key2"] = "value2";
kvs["key3"] = "value3";

cmd.clear();
cmd.mset(kvs);

std::vector<acl::string> keys = {"key1", "key2", "key3"};
std::vector<acl::string> values;
cmd.clear();
cmd.mget(keys, &values);

for (size_t i = 0; i < values.size(); i++) {
    printf("%s = %s\n", keys[i].c_str(), values[i].c_str());
}
}

```

Hash 操作

```

#include "acl_cpp/lib_acl.hpp"

void hash_operations() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_hash cmd(&client);

```

```
// HMSET - 批量设置
std::map<acl::string, acl::string> user;
user["name"] = "Alice";
user["age"] = "25";
user["city"] = "Beijing";
user["email"] = "alice@example.com";

cmd.hmset("user:1001", user);

// HGET - 获取单个字段
acl::string name;
cmd.clear();
cmd.hget("user:1001", "name", name);
printf("姓名: %s\n", name.c_str());

// HMGET - 批量获取
std::vector<acl::string> fields = {"name", "age", "city"};
std::vector<acl::string> values;
cmd.clear();
cmd.hmget("user:1001", fields, &values);

for (size_t i = 0; i < fields.size(); i++) {
    printf("%s: %s\n", fields[i].c_str(), values[i].c_str());
}

// HGETALL - 获取所有字段
std::map<acl::string, acl::string> all_fields;
cmd.clear();
cmd.hgetall("user:1001", all_fields);

printf("所有字段:\n");
for (const auto& pair : all_fields) {
    printf(" %s: %s\n", pair.first.c_str(), pair.second.c_str());
}

// HINCRBY - 字段值增加
long long int new_age;
cmd.clear();
cmd.hincrby("user:1001", "age", 1, &new_age);
printf("新年龄: %lld\n", new_age);

// HDEL - 删除字段
cmd.clear();
cmd.hdel("user:1001", "email");

// HKEYS - 获取所有字段名
std::vector<acl::string> all_keys;
cmd.clear();
cmd.hkeys("user:1001", all_keys);

printf("所有字段名: ");
for (const auto& key : all_keys) {
    printf("%s ", key.c_str());
}
```

```

printf("\n");

// HLEN - 获取字段数量
cmd.clear();
int len = cmd.hlen("user:1001");
printf("字段数量: %d\n", len);
}

```

List 操作

```

#include "acl_cpp/lib_acl.hpp"

void list_operations() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_list cmd(&client);

    const char* list_key = "tasks";

    // LPUSH - 左侧插入
    cmd.lpush(list_key, "task1");
    cmd.lpush(list_key, "task2");
    cmd.lpush(list_key, "task3");

    // RPUSH - 右侧插入
    cmd.clear();
    std::vector<acl::string> tasks = {"task4", "task5"};
    cmd.rpush(list_key, tasks);

    // LLEN - 获取长度
    cmd.clear();
    int len = cmd.llen(list_key);
    printf("列表长度: %d\n", len);

    // LRANGE - 范围查询
    std::vector<acl::string> result;
    cmd.clear();
    cmd.lrange(list_key, 0, -1, result); // 获取所有元素

    printf("所有任务:\n");
    for (const auto& task : result) {
        printf(" %s\n", task.c_str());
    }

    // LPOP - 左侧弹出
    acl::string task;
    cmd.clear();
    if (cmd.lpop(list_key, task)) {
        printf("弹出任务: %s\n", task.c_str());
    }

    // RPOP - 右侧弹出
}

```

```

cmd.clear();
if (cmd.rpop(list_key, task)) {
    printf("弹出任务: %s\n", task.c_str());
}

// LINDEX - 获取指定位置元素
cmd.clear();
if (cmd.lindex(list_key, 0, task)) {
    printf("第一个任务: %s\n", task.c_str());
}

// LSET - 设置指定位置元素
cmd.clear();
cmd.lset(list_key, 0, "updated_task");

// LTRIM - 裁剪列表
cmd.clear();
cmd.ltrim(list_key, 0, 9); // 只保留前10个元素
}

```

Set 操作

```

#include "acl_cpp/lib_acl.hpp"

void set_operations() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_set cmd(&client);

    // SADD - 添加成员
    std::vector<acl::string> members1 = {"apple", "banana", "orange"};
    cmd.sadd("fruits:set1", members1);

    std::vector<acl::string> members2 = {"banana", "grape", "watermelon"};
    cmd.clear();
    cmd.sadd("fruits:set2", members2);

    // SISMEMBER - 检查成员是否存在
    cmd.clear();
    if (cmd.sismember("fruits:set1", "apple")) {
        printf("apple 在 set1 中\n");
    }

    // SCARD - 获取成员数量
    cmd.clear();
    int count = cmd.scard("fruits:set1");
    printf("set1 成员数量: %d\n", count);

    // SMEMBERS - 获取所有成员
    std::vector<acl::string> all_members;
    cmd.clear();
    cmd.smembers("fruits:set1", all_members);
}

```

```

printf("set1 所有成员:\n");
for (const auto& member : all_members) {
    printf(" %s\n", member.c_str());
}

// SINTER - 交集
std::vector<acl::string> keys = {"fruits:set1", "fruits:set2"};
std::vector<acl::string> intersection;
cmd.clear();
cmd.sinter(keys, intersection);

printf("交集:\n");
for (const auto& member : intersection) {
    printf(" %s\n", member.c_str());
}

// SUNION - 并集
std::vector<acl::string> union_result;
cmd.clear();
cmd.sunion(keys, union_result);

printf("并集:\n");
for (const auto& member : union_result) {
    printf(" %s\n", member.c_str());
}

// SDIFF - 差集
std::vector<acl::string> diff_result;
cmd.clear();
cmd.sdiff(keys, diff_result);

printf("差集 (set1 - set2):\n");
for (const auto& member : diff_result) {
    printf(" %s\n", member.c_str());
}

// SPOP - 随机弹出
acl::string popped;
cmd.clear();
if (cmd.spop("fruits:set1", popped)) {
    printf("随机弹出: %s\n", popped.c_str());
}

// SREM - 删除成员
cmd.clear();
cmd.srem("fruits:set1", "orange");
}

```

Sorted Set 操作

```
#include "acl_cpp/lib_acl.hpp"

void zset_operations() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_zset cmd(&client);

    const char* zset_key = "leaderboard";

    // ZADD - 添加成员
    std::map<acl::string, double> players;
    players["Alice"] = 100.0;
    players["Bob"] = 85.0;
    players["Charlie"] = 95.0;
    players["David"] = 90.0;
    players["Eve"] = 88.0;

    cmd.zadd(zset_key, players);

    // ZCARD - 获取成员数量
    cmd.clear();
    int count = cmd.zcard(zset_key);
    printf("玩家数量: %d\n", count);

    // ZSCORE - 获取成员分数
    double score;
    cmd.clear();
    if (cmd.zscore(zset_key, "Alice", score)) {
        printf("Alice 的分数: %.2f\n", score);
    }

    // ZRANK - 获取排名 (从小到大)
    cmd.clear();
    int rank = cmd.zrank(zset_key, "Alice");
    printf("Alice 排名 (升序) : %d\n", rank);

    // ZREVRANK - 获取排名 (从大到小)
    cmd.clear();
    rank = cmd.zrevrank(zset_key, "Alice");
    printf("Alice 排名 (降序) : %d\n", rank);

    // ZRANGE - 范围查询 (升序)
    std::vector<acl::string> members;
    std::vector<double> scores;
    cmd.clear();
    cmd.zrange(zset_key, 0, -1, members, &scores);

    printf("排行榜 (升序) :\n");
    for (size_t i = 0; i < members.size(); i++) {
        printf(" %zu. %s: %.2f\n", i+1,
               members[i].c_str(), scores[i]);
    }

    // ZREVRANGE - 范围查询 (降序)
```

```

members.clear();
scores.clear();
cmd.clear();
cmd.zrevrange(zset_key, 0, -1, members, &scores);

printf("排行榜 (降序) :\n");
for (size_t i = 0; i < members.size(); i++) {
    printf(" %zu. %s: %.2f\n", i+1,
           members[i].c_str(), scores[i]);
}

// ZRANGEBYSCORE - 按分数范围查询
members.clear();
cmd.clear();
cmd.zrangebyscore(zset_key, 85.0, 95.0, members);

printf("分数在 85-95 之间的玩家:\n");
for (const auto& member : members) {
    printf(" %s\n", member.c_str());
}

// ZINCRBY - 增加分数
double new_score;
cmd.clear();
cmd.zincrby(zset_key, 10.0, "Bob", &new_score);
printf("Bob 新分数: %.2f\n", new_score);

// ZREM - 删除成员
cmd.clear();
cmd.zrem(zset_key, "Eve");

// ZCOUNT - 统计分数范围内的成员数
cmd.clear();
count = cmd.zcount(zset_key, 80.0, 100.0);
printf("分数在 80-100 之间的玩家数: %d\n", count);
}

```

高级特性

事务

```

#include "acl_cpp/lib_acl.hpp"

void transaction_example() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_transaction cmd(&client);

    // 开始事务
    if (!cmd.multi()) {
        printf("MULTI 失败\n");
        return;
    }
}

```

```

}

// 在事务中执行多个命令
acl::redis_string str_cmd(&client);
str_cmd.set("key1", "value1");
str_cmd.set("key2", "value2");
str_cmd.incr("counter");

// 执行事务
const acl::redis_result* result = cmd.exec();
if (result == NULL) {
    printf("EXEC 失败: %s\n", cmd.result_error());
    return;
}

// 检查结果
if (result->get_type() == acl::REDIS_RESULT_ARRAY) {
    size_t size = result->get_size();
    printf("事务执行了 %zu 个命令\n", size);

    for (size_t i = 0; i < size; i++) {
        const acl::redis_result* child = result->get_child(i);
        if (child->get_type() == acl::REDIS_RESULT_STATUS) {
            printf("命令 %zu: %s\n", i, child->get_status());
        } else if (child->get_type() == acl::REDIS_RESULT_INTEGER) {
            printf("命令 %zu: %lld\n", i, child->get_integer64());
        }
    }
}
}

void watch_example() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_transaction trans(&client);
    acl::redis_string str_cmd(&client);

    // 监视键
    std::vector<acl::string> keys = {"balance"};
    if (!trans.watch(keys)) {
        printf("WATCH 失败\n");
        return;
    }

    // 读取当前余额
    acl::string balance_str;
    if (!str_cmd.get("balance", balance_str)) {
        printf("GET 失败\n");
        trans.unwatch();
        return;
    }

    int balance = atoi(balance_str.c_str());
    printf("当前余额: %d\n", balance);
}

```

```

// 开始事务
if (!trans.multi()) {
    printf("MULTI 失败\n");
    return;
}

// 修改余额
balance += 100;
acl::string new_balance;
new_balance.format("%d", balance);
str_cmd.set("balance", new_balance.c_str());

// 执行事务
const acl::redis_result* result = trans.exec();
if (result == NULL) {
    printf("事务失败 (可能被其他客户端修改) \n");
} else {
    printf("事务成功, 新余额: %d\n", balance);
}
}

```

Lua 脚本

```

#include "acl_cpp/lib_acl.hpp"

void lua_script_example() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_script cmd(&client);

    // Lua 脚本: 原子性增加并返回
    const char* script = R"
        local current = redis.call('GET', KEYS[1])
        if current == false then
            current = 0
        end
        current = tonumber(current) + tonumber(ARGV[1])
        redis.call('SET', KEYS[1], current)
        return current
    ";

    std::vector<acl::string> keys = {"counter"};
    std::vector<acl::string> args = {"10"};

    // 执行脚本
    const acl::redis_result* result = cmd.eval(script, keys, args);
    if (result == NULL) {
        printf("EVAL 失败: %s\n", cmd.result_error());
        return;
    }

    if (result->get_type() == acl::REDIS_RESULT_INTEGER) {

```

```

        printf("新计数器值: %lld\n", result->get_integer64());
    }

    // 加载脚本获取 SHA1
    acl::string sha1;
    cmd.clear();
    if (cmd.script_load(script, sha1)) {
        printf("脚本 SHA1: %s\n", sha1.c_str());

        // 使用 SHA1 执行脚本 (更高效)
        cmd.clear();
        result = cmd.evalsha(sha1.c_str(), keys, args);
        if (result && result->get_type() == acl::REDIS_RESULT_INTEGER) {
            printf("新计数器值: %lld\n", result->get_integer64());
        }
    }
}

```

发布订阅

```

#include "acl_cpp/lib_acl.hpp"
#include <thread>

void subscriber_thread() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_pubsub cmd(&client);

    // 订阅频道
    std::vector<acl::string> channels = {"news", "updates"};
    if (cmd.subscribe(channels) <= 0) {
        printf("订阅失败\n");
        return;
    }

    printf("已订阅频道, 等待消息...\n");

    // 接收消息
    acl::string channel, message;
    while (true) {
        if (!cmd.get_message(channel, message)) {
            printf("接收消息失败\n");
            break;
        }

        printf("收到消息 [%s]: %s\n",
               channel.c_str(), message.c_str());

        cmd.clear(true); // 保留订阅状态
    }
}

```

```

void publisher_thread() {
    sleep(1); // 等待订阅者准备好

    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_pubsub cmd(&client);

    // 发布消息
    for (int i = 0; i < 10; i++) {
        acl::string msg;
        msg.format("消息 %d", i);

        int subscribers = cmd.publish("news", msg.c_str());
        printf("发布到 %d 个订阅者\n", subscribers);

        cmd.clear();
        sleep(1);
    }
}

void pubsub_example() {
    std::thread sub(subscriber_thread);
    std::thread pub(publisher_thread);

    pub.join();
    sub.detach();
}

```

扫描大键

```

#include "acl_cpp/lib_acl.hpp"

void scan_large_dataset() {
    acl::redis_client client("127.0.0.1:6379", 10, 10);
    acl::redis_key cmd(&client);

    int cursor = 0;
    int total = 0;
    size_t count = 100; // 每次扫描100个键

    do {
        // SCAN cursor [MATCH pattern] [COUNT count]
        const acl::redis_result* result =
            cmd.scan(cursor, "user:*", &count);

        if (result == NULL) {
            printf("SCAN 失败: %s\n", cmd.result_error());
            break;
        }

        // 结果格式: [newcursor, [key1, key2, ...]]
        const acl::redis_result* cursor_result = result->get_child(0);

```

```
const acl::redis_result* keys_result = result->get_child(1);

if (cursor_result) {
    acl::string cursor_str;
    cursor_str = cursor_result->get(0);
    cursor = atoi(cursor_str.c_str());
}

if (keys_result) {
    size_t size = keys_result->get_size();
    for (size_t i = 0; i < size; i++) {
        const char* key = keys_result->get(i);
        if (key) {
            printf("找到键: %s\n", key);
            total++;
        }
    }
}

cmd.clear();

} while (cursor != 0);

printf("总共找到 %d 个键\n", total);
}
```

更多示例请参考：

- [lib_acl_cpp/samples/redis/ 目录](#)
- [API 文档](#)
- [最佳实践](#)