ACL Redis API 文档

目录

基础类

redis_command

所有 Redis 命令类的基类。

构造函数

```
// 默认构造
redis_command();

// 单机模式
redis_command(redis_client* conn);

// 集群模式
redis_command(redis_client_cluster* cluster);

// Pipeline 模式
redis_command(redis_client_pipeline* pipeline);
```

连接管理

```
// 设置单机连接
void set_client(redis_client* conn);

// 设置集群连接
void set_cluster(redis_client_cluster* cluster);

// 设置 Pipeline 连接
void set_pipeline(redis_client_pipeline* pipeline);

// 获取当前连接
redis_client* get_client() const;
```

```
redis_client_cluster* get_cluster() const;
redis_client_pipeline* get_pipeline() const;
```

命令重置

```
// 重置命令对象以便复用
// save_slot: 是否保留哈希槽缓存（集群模式）
void clear(bool save_slot = false);
```

结果获取

```
// 获取结果类型
redis_result_t result_type() const;

// 获取状态信息（类型为 REDIS_RESULT_STATUS）
const char* result_status() const;

// 获取错误信息（类型为 REDIS_RESULT_ERROR）
const char* result_error() const;

// 获取整数值（类型为 REDIS_RESULT_INTEGER）
int result_number(bool* success = NULL) const;
long long int result_number64(bool* success = NULL) const;

// 获取字符串值（类型为 REDIS_RESULT_STRING）
const char* result_value(size_t i, size_t* len = NULL) const;

// 获取子结果（类型为 REDIS_RESULT_ARRAY）
const redis_result* result_child(size_t i) const;

// 获取结果对象
const redis_result* get_result() const;

// 获取结果大小
size_t result_size() const;
```

直接请求

```
// 直接发送 Redis 命令
const redis_result* request(size_t argc, const char* argv[],
                            const size_t lens[], size_t nchild = 0);

// 使用 vector 发送命令
const redis_result* request(const std::vector<string>& args,
                            size_t nchild = 0);
```

redis_client

单机 Redis 连接。

构造函数

```
redis_client(const char* addr,            // Redis 服务器地址 ip:port
             int conn_timeout = 60,       // 连接超时（秒）
             int rw_timeout = 30,         // 读写超时（秒）
             bool retry = true);          // 是否自动重连
```

配置方法

```
// 设置密码
void set_password(const char* pass);

// 设置数据库（0-15，仅单机模式）
void set_db(int dbnum);

// 设置 SSL 配置
void set_ssl_conf(sslbase_conf* ssl_conf);

// 设置是否分片请求
void set_slice_request(bool on);

// 设置是否分片响应
void set_slice_respond(bool on);
```

连接管理

```
// 检查连接是否关闭
bool eof() const;

// 关闭连接
void close();

// 获取底层 socket 流
socket_stream* get_stream(bool auto_connect = true);
```

redis_client_cluster

Redis 集群连接。

构造函数

```
redis_client_cluster(int max_slot = 16384);  // 最大哈希槽数
```

集群配置

```
// 自动发现所有节点和槽位
void set_all_slot(const char* addr,       // 任一集群节点地址
                  size_t max_conns,       // 每个连接池的最大连接数
                  int conn_timeout = 30,  // 连接超时
                  int rw_timeout = 30);   // 读写超时

// 手动设置槽位映射
void set_slot(int slot, const char* addr);

// 清除槽位映射
void clear_slot(int slot);

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

重定向控制

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

// 获取最大重定向次数
int get_redirect_max() const;

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

// 获取重定向等待时间
int get_redirect_sleep() const;
```

密码管理

```
// 设置密码
// addr: 节点地址，"default" 表示所有节点的默认密码
void set_password(const char* addr, const char* pass);

// 获取密码
const char* get_password(const char* addr) const;

// 获取所有密码配置
const std::map<string, string>& get_passwords() const;
```

redis_result

Redis 结果对象。

结果类型

```
typedef enum {
    REDIS_RESULT_UNKOWN,    // 未知
    REDIS_RESULT_NIL,       // 空值
    REDIS_RESULT_ERROR,     // 错误
    REDIS_RESULT_STATUS,    // 状态
    REDIS_RESULT_INTEGER,   // 整数
    REDIS_RESULT_STRING,    // 字符串
    REDIS_RESULT_ARRAY,     // 数组
} redis_result_t;
```

获取结果

```
// 获取结果类型
redis_result_t get_type() const;

// 获取结果大小
size_t get_size() const;

// 获取整数值
int get_integer(bool* success = NULL) const;
long long int get_integer64(bool* success = NULL) const;

// 获取浮点数值
double get_double(bool* success = NULL) const;

// 获取状态字符串
const char* get_status() const;

// 获取错误信息
const char* get_error() const;

// 获取字符串（支持二进制数据）
const char* get(size_t i, size_t* len = NULL) const;

// 获取所有字符串数组
const char** get_argv() const;
const size_t* get_lens() const;

// 获取子结果（数组类型）
const redis_result* get_child(size_t i) const;
const redis_result** get_children(size_t* size) const;
```

```
// 组合字符串数据到连续内存
int argv_to_string(string& buf, bool clear_auto = true) const;
```

String 命令

redis_string

字符串操作命令类。

SET/GET

```
// SET key value
bool set(const char* key, const char* value);
bool set(const char* key, size_t key_len,
        const char* value, size_t value_len);

// SET key value [EX seconds] [PX milliseconds] [NX|XX]
// flag: SETFLAG_EX | SETFLAG_PX | SETFLAG_NX | SETFLAG_XX
bool set(const char* key, const char* value, int timeout, int flag);

// SETEX key seconds value
bool setex(const char* key, const char* value, int timeout);

// PSETEX key milliseconds value
bool psetex(const char* key, const char* value, int timeout);

// SETNX key value
// 返回值: -1 错误, 0 key已存在, 1 设置成功
int setnx(const char* key, const char* value);

// GET key
bool get(const char* key, string& buf);
bool get(const char* key, size_t len, string& buf);
const redis_result* get(const char* key);

// GETSET key value
bool getset(const char* key, const char* value, string& buf);

// APPEND key value
int append(const char* key, const char* value);
int append(const char* key, const char* value, size_t size);
```

批量操作

```
// MSET key1 value1 key2 value2 ...
bool mset(const std::map<string, string>& objs);
bool mset(const std::vector<string>& keys,
         const std::vector<string>& values);
```

```
bool mset(const char* keys[], const char* values[], size_t argc);

// MSETNX key1 value1 key2 value2 ...
// 返回值: -1 错误, 0 部分key已存在, 1 全部设置成功
int msetnx(const std::map<string, string>& objs);
int msetnx(const std::vector<string>& keys,
           const std::vector<string>& values);

// MGET key1 key2 key3 ...
bool mget(const std::vector<string>& keys,
          std::vector<string>* out = NULL);
bool mget(const std::vector<const char*>& keys,
          std::vector<string>* out = NULL);
bool mget(const char* keys[], size_t argc,
          std::vector<string>* out = NULL);
```

数值操作

```
// INCR key
bool incr(const char* key, long long int* result = NULL);

// INCRBY key increment
bool incrby(const char* key, long long int inc,
            long long int* result = NULL);

// INCRBYFLOAT key increment
bool incrbyfloat(const char* key, double inc, double* result = NULL);

// DECR key
bool decr(const char* key, long long int* result = NULL);

// DECRBY key decrement
bool decrby(const char* key, long long int dec,
            long long int* result = NULL);
```

字符串操作

```
// STRLEN key
int get_strlen(const char* key);

// SETRANGE key offset value
int setrange(const char* key, unsigned offset, const char* value);

// GETRANGE key start end
bool getrange(const char* key, int start, int end, string& buf);
```

位操作

```
// SETBIT key offset value
bool setbit_(const char* key, unsigned offset, bool bit);

// GETBIT key offset
bool getbit(const char* key, unsigned offset, int& bit);

// BITCOUNT key [start end]
int bitcount(const char* key);
int bitcount(const char* key, int start, int end);

// BITOP AND destkey key1 key2 ...
int bitop_and(const char* destkey, const std::vector<string>& keys);
int bitop_and(const char* destkey, const char* keys[], size_t size);

// BITOP OR destkey key1 key2 ...
int bitop_or(const char* destkey, const std::vector<string>& keys);

// BITOP XOR destkey key1 key2 ...
int bitop_xor(const char* destkey, const std::vector<string>& keys);
```

Hash 命令

redis_hash

哈希表操作命令类。

基本操作

```
// HSET key field value
// 返回值: -1 错误, 0 字段已存在, 1 新字段
int hset(const char* key, const char* name, const char* value);
int hset(const char* key, const char* name, size_t name_len,
        const char* value, size_t value_len);

// HSETNX key field value
// 返回值: -1 错误, 0 字段已存在, 1 设置成功
int hsetnx(const char* key, const char* name, const char* value);

// HGET key field
bool hget(const char* key, const char* name, string& value);
const redis_result* hget(const char* key, const char* name);

// HDEL key field [field ...]
// 返回值: 删除的字段数量, -1 表示错误
int hdel(const char* key, const char* name);
int hdel(const char* key, const std::vector<string>& names);
int hdel(const char* key, const char* names[], size_t argc);

// HEXISTS key field
bool hexists(const char* key, const char* name);
```

```
// HLEN key
int hlen(const char* key);
```

批量操作

```
// HMSET key field1 value1 field2 value2 ...
bool hmset(const char* key, const std::map<string, string>& attrs);
bool hmset(const char* key, const std::vector<string>& names,
          const std::vector<string>& values);
bool hmset(const char* key, const char* names[], const char* values[],
          size_t argc);

// HMGET key field1 field2 field3 ...
bool hmget(const char* key, const std::vector<string>& names,
          std::vector<string>* result = NULL);
bool hmget(const char* key, const char* names[], size_t argc,
          std::vector<string>* result = NULL);

// HGETALL key
bool hgetall(const char* key, std::map<string, string>& result);
bool hgetall(const char* key, std::vector<string>& names,
            std::vector<string>& values);

// HKEYS key
bool hkeys(const char* key, std::vector<string>& names);

// HVALS key
bool hvals(const char* key, std::vector<string>& values);
```

数值操作

```
// HINCRBY key field increment
bool hincrby(const char* key, const char* name, long long int inc,
            long long int* result = NULL);

// HINCRBYFLOAT key field increment
bool hincrbyfloat(const char* key, const char* name, double inc,
                 double* result = NULL);
```

扫描

```
// HSCAN key cursor [MATCH pattern] [COUNT count]
const redis_result* hscan(const char* key, int cursor,
                          const char* pattern = NULL,
                          const size_t* count = NULL);
```

List 命令

redis_list

列表操作命令类。

插入

```
// LPUSH key value [value ...]
int lpush(const char* key, const char* value);
int lpush(const char* key, const std::vector<string>& values);
int lpush(const char* key, const char* values[], size_t argc);

// LPUSHX key value
int lpushx(const char* key, const char* value);

// RPUSH key value [value ...]
int rpush(const char* key, const char* value);
int rpush(const char* key, const std::vector<string>& values);

// RPUSHX key value
int rpushx(const char* key, const char* value);

// LINSERT key BEFORE|AFTER pivot value
int linsert(const char* key, const char* pivot, const char* value,
            bool before = true);
```

弹出

```
// LPOP key
bool lpop(const char* key, string& buf);
const redis_result* lpop(const char* key);

// RPOP key
bool rpop(const char* key, string& buf);
const redis_result* rpop(const char* key);

// BLPOP key [key ...] timeout
// 返回值: NULL 超时, 否则返回弹出的元素
const redis_result* blpop(const std::vector<string>& keys, size_t
timeout);
const redis_result* blpop(const char* keys[], size_t argc, size_t
timeout);

// BRPOP key [key ...] timeout
const redis_result* brpop(const std::vector<string>& keys, size_t
timeout);
```

```
// RPOPLPUSH source destination
bool rpoplpush(const char* src, const char* dst, string& buf);

// BRPOPLPUSH source destination timeout
bool brpoplpush(const char* src, const char* dst, size_t timeout, string&
buf);
```

查询

```
// LLEN key
int llen(const char* key);

// LINDEX key index
bool lindex(const char* key, int index, string& buf);
const redis_result* lindex(const char* key, int index);

// LRANGE key start stop
bool lrange(const char* key, int start, int stop,
            std::vector<string>& result);
const redis_result* lrange(const char* key, int start, int stop);
```

修改

```
// LSET key index value
bool lset(const char* key, int index, const char* value);

// LREM key count value
int lrem(const char* key, int count, const char* value);

// LTRIM key start stop
bool ltrim(const char* key, int start, int stop);
```

Set 命令

redis_set

集合操作命令类。

基本操作

```
// SADD key member [member ...]
int sadd(const char* key, const char* member);
int sadd(const char* key, const std::vector<string>& members);
int sadd(const char* key, const char* members[], size_t argc);
```

```
// SREM key member [member ...]
int srem(const char* key, const char* member);
int srem(const char* key, const std::vector<string>& members);

// SISMEMBER key member
bool sismember(const char* key, const char* member);

// SCARD key
int scard(const char* key);

// SMEMBERS key
bool smembers(const char* key, std::vector<string>& members);

// SPOP key
bool spop(const char* key, string& member);

// SRANDMEMBER key [count]
bool srandmember(const char* key, string& member);
bool srandmember(const char* key, int count, std::vector<string>&
members);
```

集合运算

```
// SINTER key [key ...]
bool sinter(const std::vector<string>& keys,
            std::vector<string>& members);

// SINTERSTORE destination key [key ...]
int sinterstore(const char* dst, const std::vector<string>& keys);

// SUNION key [key ...]
bool sunion(const std::vector<string>& keys,
            std::vector<string>& members);

// SUNIONSTORE destination key [key ...]
int sunionstore(const char* dst, const std::vector<string>& keys);

// SDIFF key [key ...]
bool sdiff(const std::vector<string>& keys,
           std::vector<string>& members);

// SDIFFSTORE destination key [key ...]
int sdiffstore(const char* dst, const std::vector<string>& keys);

// SMOVE source destination member
int smove(const char* src, const char* dst, const char* member);
```

扫描

```
// SSCAN key cursor [MATCH pattern] [COUNT count]
const redis_result* sscan(const char* key, int cursor,
                          const char* pattern = NULL,
                          const size_t* count = NULL);
```

Sorted Set 命令

redis_zset

有序集合操作命令类。

添加/删除

```
// ZADD key score member [score member ...]
int zadd(const char* key, const std::map<string, double>& members);
int zadd(const char* key, const std::vector<string>& members,
        const std::vector<double>& scores);
int zadd(const char* key, const char* members[], const double scores[],
        size_t argc);

// ZREM key member [member ...]
int zrem(const char* key, const char* member);
int zrem(const char* key, const std::vector<string>& members);

// ZREMRANGEBYRANK key start stop
int zremrangebyrank(const char* key, int start, int stop);

// ZREMRANGEBYSCORE key min max
int zremrangebyscore(const char* key, double min, double max);
int zremrangebyscore(const char* key, const char* min, const char* max);
```

查询

```
// ZCARD key
int zcard(const char* key);

// ZCOUNT key min max
int zcount(const char* key, double min, double max);
int zcount(const char* key, const char* min, const char* max);

// ZSCORE key member
bool zscore(const char* key, const char* member, double& result);

// ZRANK key member
int zrank(const char* key, const char* member);
```

```
// ZREVRANK key member
int zrevrank(const char* key, const char* member);
```

范围查询

```
// ZRANGE key start stop [WITHSCORES]
bool zrange(const char* key, int start, int stop,
           std::vector<string>& members);
bool zrange(const char* key, int start, int stop,
           std::vector<string>& members, std::vector<double>* scores);

// ZREVRANGE key start stop [WITHSCORES]
bool zrevrange(const char* key, int start, int stop,
              std::vector<string>& members);

// ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]
bool zrangebyscore(const char* key, double min, double max,
                  std::vector<string>& members,
                  const int* offset = NULL, const int* count = NULL);

// ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]
bool zrevrangebyscore(const char* key, double max, double min,
                     std::vector<string>& members);
```

数值操作

```
// ZINCRBY key increment member
bool zincrby(const char* key, double inc, const char* member,
            double* result = NULL);
```

集合运算

```
// ZINTERSTORE destination numkeys key [key ...]
int zinterstore(const char* dst, const std::vector<string>& keys,
                const std::vector<double>* weights = NULL,
                const char* aggregate = NULL);

// ZUNIONSTORE destination numkeys key [key ...]
int zunionstore(const char* dst, const std::vector<string>& keys,
                const std::vector<double>* weights = NULL,
                const char* aggregate = NULL);
```

Key 命令

redis_key

键操作命令类。

基本操作

```
// DEL key [key ...]
int del(const char* key);
int del(const std::vector<string>& keys);
int del(const char* keys[], size_t argc);

// EXISTS key
bool exists(const char* key);

// TYPE key
redis_key_t type(const char* key);

// RENAME key newkey
bool rename(const char* key, const char* newkey);

// RENAMENX key newkey
int renamenx(const char* key, const char* newkey);
```

过期时间

```
// EXPIRE key seconds
int expire(const char* key, int ttl);

// EXPIREAT key timestamp
int expireat(const char* key, time_t stamp);

// PEXPIRE key milliseconds
int pexpire(const char* key, int ttl);

// PEXPIREAT key milliseconds-timestamp
int pexpireat(const char* key, long long stamp);

// TTL key
int ttl(const char* key);

// PTTL key
long long int pttl(const char* key);

// PERSIST key
int persist(const char* key);
```

扫描

```
// KEYS pattern
int keys_pattern(const char* pattern, std::vector<string>& keys);

// SCAN cursor [MATCH pattern] [COUNT count]
const redis_result* scan(int cursor, const char* pattern = NULL,
                         const size_t* count = NULL);
```

移动

```
// MOVE key db
int move(const char* key, int dbnum);

// RANDOMKEY
bool randomkey(string& key);
```

其他命令

redis_pubsub

发布订阅命令。

```
// PUBLISH channel message
int publish(const char* channel, const char* msg);

// SUBSCRIBE channel [channel ...]
int subscribe(const char* channel, ...);
int subscribe(const std::vector<string>& channels);

// UNSUBSCRIBE [channel [channel ...]]
int unsubscribe(const char* channel, ...);
int unsubscribe(const std::vector<string>& channels);

// PSUBSCRIBE pattern [pattern ...]
int psubscribe(const char* pattern, ...);

// PUNSUBSCRIBE [pattern [pattern ...]]
int punsubscribe(const char* pattern, ...);

// 获取订阅消息
bool get_message(string& channel, string& msg);
```

redis_transaction

事务命令。

```
```

```
// MULTI
bool multi();

// EXEC
const redis_result* exec();

// DISCARD
bool discard();

// WATCH key [key ...]
bool watch(const char* key);
bool watch(const std::vector<string>& keys);

// UNWATCH
bool unwatch();
```

redis_script

Lua 脚本命令。

```
// EVAL script numkeys key [key ...] arg [arg ...]
const redis_result* eval(const char* script,
                         const std::vector<string>& keys,
                         const std::vector<string>& args);

// EVALSHA sha1 numkeys key [key ...] arg [arg ...]
const redis_result* evalsha(const char* sha1,
                            const std::vector<string>& keys,
                            const std::vector<string>& args);

// SCRIPT LOAD script
bool script_load(const char* script, string& sha1);

// SCRIPT EXISTS sha1 [sha1 ...]
bool script_exists(const std::vector<string>& sha1s,
                   std::vector<int>& results);

// SCRIPT FLUSH
bool script_flush();

// SCRIPT KILL
bool script_kill();
```

redis_server

服务器命令。

```
```

```
// INFO [section]
bool info(const char* section, string& result);
bool info(string& result);

// CONFIG GET parameter
bool config_get(const char* param, std::map<string, string>& result);

// CONFIG SET parameter value
bool config_set(const char* param, const char* value);

// CONFIG RESETSTAT
bool config_resetstat();

// DBSIZE
int dbsize();

// FLUSHDB
bool flushdb();

// FLUSHALL
bool flushall();

// SAVE
bool save();

// BGSAVE
bool bgsave();

// LASTSAVE
time_t lastsave();

// SHUTDOWN [NOSAVE|SAVE]
bool shutdown(bool save = false);

// TIME
bool time(long long& sec, long long& usec);

// CLIENT LIST
bool client_list(std::vector<string>& clients);

// CLIENT SETNAME connection-name
bool client_setname(const char* name);

// CLIENT GETNAME
bool client_getname(string& name);

// CLIENT KILL ip:port
bool client_kill(const char* addr);
```

redis_geo

地理位置命令（Redis 3.2+）。

```
// GEOADD key longitude latitude member [longitude latitude member ...]
int geoadd(const char* key, const std::map<string,
          std::pair<double, double>>& members);

// GEOPOS key member [member ...]
bool geopos(const char* key, const std::vector<string>& members,
           std::vector<std::pair<double, double>>& positions);

// GEODIST key member1 member2 [unit]
bool geodist(const char* key, const char* member1, const char* member2,
            double& result, const char* unit = "m");

// GEORADIUS key longitude latitude radius unit [WITHCOORD] [WITHDIST]
[...]
const redis_result* georadius(const char* key, double lng, double lat,
                              double radius, const char* unit,
                              const int* count = NULL,
                              const char* order = NULL,
                              bool withcoord = false,
                              bool withdist = false,
                              bool withhash = false);

// GEORADIUSBYMEMBER key member radius unit [...]
const redis_result* georadiusbymember(const char* key, const char* member,
                                      double radius, const char* unit,
                                      const int* count = NULL,
                                      const char* order = NULL);
```

redis_stream

流命令（Redis 5.0+）。

```
// XADD key [MAXLEN [~] count] * field value [field value ...]
bool xadd(const char* key, const std::map<string, string>& fields,
         string& id, const int* maxlen = NULL);

// XLEN key
int xlen(const char* key);

// XRANGE key start end [COUNT count]
const redis_result* xrange(const char* key, const char* start,
                           const char* end, const int* count = NULL);

// XREVRANGE key end start [COUNT count]
const redis_result* xrevrange(const char* key, const char* end,
                              const char* start, const int* count = NULL);

// XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID
...]
const redis_result* xread(const std::vector<string>& keys,
```

```
                            const std::vector<string>& ids,
                            const int* count = NULL,
                            const int* block = NULL);

// XDEL key ID [ID ...]
int xdel(const char* key, const std::vector<string>& ids);

// XTRIM key MAXLEN [~] count
int xtrim(const char* key, int maxlen, bool approx = false);
```

redis_hyperloglog

HyperLogLog 命令。

```
// PFADD key element [element ...]
int pfadd(const char* key, const std::vector<string>& elements);
int pfadd(const char* key, const char* elements[], size_t argc);

// PFCOUNT key [key ...]
int pfcount(const char* key);
int pfcount(const std::vector<string>& keys);

// PFMERGE destkey sourcekey [sourcekey ...]
bool pfmerge(const char* dst, const std::vector<string>& src);
```

注意事项：

1. 所有返回 `bool` 的方法，`true` 表示成功，`false` 表示失败
2. 返回 `int` 的方法，通常 `-1` 表示错误，其他值表示具体含义
3. 使用 `result_error()` 获取详细错误信息
4. 使用 `clear()` 方法重置命令对象以便复用
5. 字符串参数支持二进制安全（可指定长度）
6. 集群模式下，同一个命令的所有 key 必须在同一个哈希槽

更多详细信息请参考头文件注释和示例代码。