# Redis 数据类型与 C++ 客户端实践指南（redis-plus-plus）（163 ~ 181）

Redis 是高性能的内存数据库，支持多种数据结构。`redis-plus-plus` 是 C++ 生态中成熟的客户端，它在 `hiredis` 基础上做了面向对象封装，提供了易用的接口，同时兼顾性能和 STL 原生适配。本文将系统介绍各数据类型的核心命令与 C++ 特性适配，以及客户端设计思想和最佳实践。

---

## 一、String 类型核心命令与 C++ 适配

### 1. 基础读写与批量操作

Redis 中字符串（String）是最基础的数据类型，用于存储短文本或序列化对象。C++ 中通过 `redis-plus-plus` 的接口直接封装：

代码块

```
#include <sw/redis++/redis++.h>

using namespace sw::redis;

Redis redis("tcp://127.0.0.1:6379");

// 基础写入
redis.set("key", "value");

// 基础读取
auto val = redis.get("key");
if(val.has_value()) {
    std::cout << "Value: " << val.value() << std::endl;
}
```

- **批量操作**：
  - `mset`：一次性设置多个键值对，避免循环调用多次 `set`。
  - `mget`：一次性获取多个键值，返回 `std::vector<Optional<std::string>>`。

代码块

```
redis.mset({{"k1", "v1"}, {"k2", "v2"}});
```

```
auto vals = redis.mget({"k1", "k2", "k3"});
for(auto &v : vals) {
    if(v) std::cout << *v << std::endl;
    else std::cout << "Key not found" << std::endl;
}
```

- **子串操作**：
  - `getrange(key, start, stop)`：获取字符串的部分区间。
  - `setrange(key, offset, val)`：从指定位置覆盖字符串内容，适合处理大字符串的局部更新。

代码块

```
redis.set("long_str", "HelloWorld");
auto part = redis.getrange("long_str", 0, 4); // "Hello"
redis.setrange("long_str", 5, "Redis"); // "HelloRedis"
```

### 2. 数值操作与类型转换

Redis 可以直接将字符串作为整数处理：

代码块

```
redis.set("counter", "10");
long long cnt = redis.incr("counter"); // 原子加 1
cnt = redis.decr("counter");           // 原子减 1
```

- `incr`、`decr` 返回 `long long`，避免手动字符串转换。
- `get` 返回 `Optional<std::string>`，如果需要数值类型，需要转换：
  - C++17 可用 `std::from_chars`
  - 传统方法：`std::stoll` / `std::stoi`
  - Boost 方法：`boost::lexical_cast<long long>(val.value())`

---

## 二、List 类型核心命令与特性

Redis List 是有序的双端队列，支持高效插入和弹出操作。

### 1. 双端操作与阻塞弹出

```
redis.lpush("list1", "a"); // 头部插入
redis.rpush("list1", {"b", "c"}); // 批量尾部插入
auto front = redis.lpop("list1");
auto back = redis.rpop("list1");
```

- 返回值类型为 `Optional<std::string>`，空列表返回空值。
- 阻塞式弹出命令：

代码块

```
auto res = redis.brpop({"list1", "list2"}, std::chrono::seconds(5));
if(res) {
    std::cout << "List: " << res->first << ", Value: " << res->second << std::endl;
} else {
    std::cout << "Timeout" << std::endl;
}
```

- 特点：
  - 阻塞操作等待新元素插入。
  - 返回包含列表名和元素值的 `std::pair`。
  - 适合消息队列、任务调度场景。

### 2. 范围查询与长度获取

代码块

```
auto items = redis.lrange("list1", 0, -1); // 获取全部元素
long long len = redis.llen("list1");       // 获取列表长度
```

- `lrange` 返回 `std::vector<std::string>`，可直接迭代或与 STL 算法结合。
- `llen` 为 O(1) 操作，快速获取长度。

---

## 三、Set 类型核心命令与迭代器

Redis Set 是无序集合，支持去重和集合运算。

### 1. 基础增删查与集合运算

代码块

```
redis.sadd("set1", {"a", "b", "c"});      // 批量添加元素
auto members = redis.smembers("set1");    // 获取所有元素
bool exists = redis.sismember("set1", "a"); // 判断存在性
long long size = redis.scard("set1");       // 集合大小

// 集合运算
redis.sinterstore("set_inter", {"set1", "set2"}); // 交集存储
```

- `smembers` 返回 `std::vector<std::string>`，便于 STL 遍历。
- `sadd` 返回新增元素数量，便于统计。

### 2. 迭代器与插入特性

- Set 是无序集合，不存在尾部插入，无法 `push_back`。
- 可使用 STL 插入迭代器批量操作：

代码块

```
std::set<std::string> cpp_set;
auto redis_set = redis.smembers("set1");
std::copy(redis_set.begin(), redis_set.end(), std::inserter(cpp_set,
cpp_set.begin()));
```

- 优势：直接将 Redis 数据导入 STL 容器，便于算法处理。

---

## 四、Hash 与 ZSet 类型核心命令

### 1. Hash 类型

Hash 是键值对集合，适合存储对象属性。

代码块

```
redis.hset("user:1", "name", "Alice");
redis.hset("user:1", {{"age", "20"}, {"gender", "F"}}); // 批量设置
auto name = redis.hget("user:1", "name");
auto fields = redis.hkeys("user:1");
auto values = redis.hvals("user:1");
auto all = redis.hgetall("user:1"); //
std::unordered_map<std::string,std::string>
long long count = redis.hlen("user:1");
```

- 批量操作减少网络请求。
- `hgetall` 返回 STL 容器，直接可用于业务逻辑。

### 2. ZSet 类型

ZSet 是有序集合，元素带分数，用于排行榜和排序。

代码块

```
redis.zadd("leaderboard", {{"Alice", 100}, {"Bob", 90}});
auto top = redis.zrange("leaderboard", 0, -1); // 默认升序
auto top_with_score = redis.zrange_with_scores("leaderboard", 0, -1);

auto rank = redis.zrank("leaderboard", "Alice");       // 升序排名
auto rev_rank = redis.zrevrank("leaderboard", "Alice"); // 降序排名
long long z_size = redis.zcard("leaderboard");
```

- `zrange_with_scores` 返回 `std::vector<std::pair<std::string, double>>`，便于 STL 遍历。
- `zrank/zrevrank` 返回 `Optional<long long>`，元素不存在时安全处理。

---

## 五、C++ 客户端设计特点与最佳实践

### 1. 接口一致性设计

- 所有 Redis 数据类型命令都返回 C++ 原生类型：
  - 单值：`Optional<std::string>` / `Optional<long long>`
  - 多值：`std::vector<std::string>` / `std::unordered_map<std::string,std::string>`
- 支持批量参数传递：

代码块

```
redis.sadd("set_key", {"a", "b", "c"}); // 支持 initializer_list
```

- 优势：
  - 降低学习成本。
  - 保证类型安全。
  - 接口风格统一，便于工程化开发。

### 2. STL 原生适配与迭代器

- 返回值可直接与 STL 算法结合：

```
auto keys = redis.keys("user:*");
std::sort(keys.begin(), keys.end());
```

- 批量插入到 STL 容器：

```
std::vector<std::string> v;
std::copy(redis.smembers("set1").begin(), redis.smembers("set1").end(),
          std::back_inserter(v));
```

### 3. 空值与错误处理

- 空值命令（如 `get`，`lpop`，`zrank`）使用 `Optional`：

```
if(val.has_value()) do_something(val.value());
else handle_missing();
```

- 系统异常（网络、协议错误）抛出 `Error` 异常：

```
try {
    redis.set("key", "value");
} catch(const Error &err) {
    std::cerr << err.what() << std::endl;
}
```

- 设计思想：
  - 空值 → Optional → 业务逻辑处理。
  - 系统异常 → Exception → 错误处理逻辑。

---

## 六、总结与实践价值

1. **性能与易用性平衡**：
   - StringView、Optional 避免不必要拷贝与空指针问题。
   - 面向对象接口易用，同时保持底层性能。
2. **STL 原生适配**：
   - 返回值与 STL 容器无缝结合，迭代器和算法可直接使用。
   - 避免手动容器转换，降低代码复杂度。
3. **空值与异常分离**：
   - 业务空值使用 Optional 安全访问。
   - 网络或协议异常抛出异常，增强健壮性。
4. **批量操作与迭代器结合**：
   - 支持 initializer_list、迭代器传入批量命令。
   - 可与 STL 算法配合，实现高效批量处理。
5. **工程化实践**：
   - 连接池、事务、管道和阻塞操作支持。
   - 高性能、高可维护性，适合企业级项目。

---