Redis 数据结构详解

目录

1. 五个基础数据结构

Redis 是一种高效的 内存数据库，它提供了五种不同的数据结构，分别是字符串（String ）、列表（List）、哈希（Hash）、集合（Set）和有序集合（Zset）。

> 推荐参考：Redis 常用命令

1.1 字符串（String）

Redis 的 String 数据结构用于存储字符串类型的数据。
其底层原理是基于 简单动态字符串 （Simple Dynamic String，SDS）的。
SDS 的优点包括：

- 空间预分配：当需要修改字符串时，Redis 会预先分配足够的空间，减少不必要的内存重分配次数。
- 惰性释放：当字符串缩短时，不会立即释放多余的内存，而是等待后续有需要时再进行释放。
- 二进制安全：可以存储任意类型的二进制数据。

例如，当向 Redis 中添加一个字符串 “hello” 时，Redis 会根据 SDS 原理进行存储和管理。
这种数据结构和底层原理的设计使得 Redis 在处理字符串类型的数据时具有高效性和灵活性。

应用场景：

- **缓存数据**：用户会话信息、页面缓存、API响应较慢缓存
- **网站访问计数**：使用INCR和DECR操作，记录网站的访问量。每当有用户访问网站时，就对相应的计数器进行增加操作，从而实时统计网站的访问量。

```
// 初始化一个键为"website_visits"，值为"0"的字符串
SET website_visits 0
// 每当有用户访问网站时，我们使用INCR命令对"website_visits"键的值进行递增操作。
INCR website_visits
// 最后，我们可以使用GET命令来获取当前的访问量。
GET website_visits
```

- **限流器**：使用INCRBY和过期时间，限制用户在一定时间内的操作次数。例如，可以限制用户每分钟只能登录5次，当登录次数达到限制时，可以拒绝用户的登录请求或提示用户稍后再试。

```
// 首先，我们使用SET命令（或INCR/INCRBY初始化）初始化一个键为"user:login_count:<user_id>"
//（其中<user_id>是用户的唯一标识符），值为"0"的字符串。同时，使用EXPIRE命令设置该键的过期时间为60秒。

SET user:login_count:<user_id> 0 EX 60

// 然后，每当用户尝试登录时，我们使用INCRBY命令对"user:login_count:<user_id>"键的值进行递增操作，增量为1。
INCRBY user:login_count:<user_id> 1
```

- **分布式锁**：可以使用Redis的String类型实现分布式锁。通过设置一个唯一的键和值，并设置过期时间，来确保在分布式环境中只有一个客户端能够获取到锁，从而避免并发问题。

```
SETNX lock_name <client_id> EX <lock_timeout>
```

1.2 列表（List）

> 推荐参考：Redis List 底层三种数据结构原理剖析

Redis 的 List 是一种线性的有序结构，可以按照元素被推入列表中的顺序来存储元素，能满足先进先出的需求，这些元素既可以是文字数据，又可以是二进制数据。其底层有 `linkedList`、`zipList` 和 `quickList` 这三种存储方式。

- `linkedList（双向链表）`：与 Java 中的 LinkedList 类似，Redis 中的 linkedList 是一个双向链表，由一个个节点组成。每个节点使用 adlist.h/listNode 结构来表示，其中包含 prev 和 next 指针，以及指向节点值的 value 指针。通过 prev 和 next 指针，程序可以高效地获取某个节点的前置节点和后继节点。同时，list 结构提供了 head 和 tail 指针，以及一些实现多态的特定函数，使得程序可以高效地获取链表的头节点和尾节点。
- `zipList（压缩列表）`：是一种内存紧凑的数据结构，占用一块连续的内存空间，提升内存使用率。当一个列表只有少量数据，并且每个列表项要么是小整数值，要么是长度比较短的字符串时，Redis 会使用 zipList 来做 List 的底层实现。
- `quickList（快速列表）`：由多个 ziplist 和一个双向循环链表组成。每个 ziplist 表示一个小的连续内存块，可以存储若干个元素。而双向循环链表用于连接多个 ziplist，形成一个大的、连续的内存空间。

应用场景：

- **消息队列**：Redis的List数据结构可以很方便地实现一个简单的消息队列。

生产者：使用LPUSH命令将消息插入到List的左侧。

```
Jedis jedis = new Jedis("localhost", 6379);
jedis.lpush("message_queue", "message1");
jedis.lpush("message_queue", "message2");
```

消费者：使用RPOP命令从List的右侧获取消息，并按顺序读取。

```
String message1 = jedis.rpop("message_queue");
String message2 = jedis.rpop("message_queue");
System.out.println("Message 1: " + message1);
System.out.println("Message 2: " + message2);
```

阻塞读取：为了避免消费者在没有消息时不断轮询队列，Redis提供了BRPOP命令，该命令会阻塞直到有新消息到达。

```
List<String> result = jedis.brpop(0, "message_queue");
String message = result.get(1);
System.out.println("Received message: " + message);
```

- **保存最新的消息列表**：在某些应用中，需要保存最新的消息或日志，并限制列表的长度。Redis的List数据结构可以方便地实现这一需求。

插入消息：使用LPUSH命令将新消息插入到List的左侧。

```
jedis.lpush("latest_messages", "message1");
jedis.lpush("latest_messages", "message2");
```

保持列表长度：使用LTRIM命令设置List的范围，从而保持其长度不超过指定的大小。

```
int maxMessages = 10;
jedis.ltrim("latest_messages", 0, maxMessages - 1);
```

获取消息列表：使用LRANGE命令获取List中的所有元素。

```
List<String> latestMessages = jedis.lrange("latest_messages", 0, -1);
for (String message : latestMessages) {
    System.out.println("Message: " + message);
}
```

1.3 哈希（Hash）

> 推荐参考：Redis高可用系列——Hash类型介绍及底层原理详解

Hash 是 Redis 中一种键值对集合，类似于编程语言中的 map 对象。一个 hash 类型的键最多可以存储 2^32 -1 个字段。Hash 类型的底层实现有三种：ziplist、listpack 和 hashtable。

应用场景：

- **用户信息管理**：存储用户的基本信息，如用户名、密码、邮箱等。

```
# 为用户ID为1001的用户存储基本信息
HSET user:1001 username "Alice"
HSET user:1001 password "$2a$10$abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"  # 存储的是密码的哈希值
HSET user:1001 email "alice@example.com"

```

```
# 获取用户的基本信息
HGETALL user:1001
# 结果:
# 1) "username"
# 2) "Alice"
# 3) "password"
# 4) "$2a$10$abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
# 5) "email"
# 6) "alice@example.com"
```

- **商品详情管理**：存储商品的详细信息，如价格、库存、描述等。

```
# 为商品ID为2002的商品存储详细信息
HSET product:2002 name "Laptop"
HSET product:2002 description "A high-performance laptop with the latest technology."
HSET product:2002 image_url "http://example.com/laptop.jpg"

# 获取商品的详细信息
HGETALL product:2002
# 结果可能是:
# 1) "name"
# 2) "Laptop"
# 3) "description"
# 4) "A high-performance laptop with the latest technology."
# 5) "image_url"
# 6) "http://example.com/laptop.jpg"
```

1.4 集合（Set）

> 推荐参考：【大课堂】Redis中hash、set、zset的底层数据结构原理

Redis Set 数据结构是一个 无序的、自动去重的集合 数据类型。Set 底层用两种数据结构存储，一个是 `hashtable`，一个是 `intset`。
hashtable 的 key 为 set 中元素的值，而 value 为 null。inset 为可以理解为 数组 ，使用intset数据结构需要满足下述两个条件：

- 元素个数不少于默认值512；
- set-max-intset-entries的值为512。

应用场景：

- **去重操作**：在处理用户输入或数据时，经常需要去除重复的元素。Redis的Set数据结构由于其元素唯一性，非常适合用于此类去重操作。

假设有一个用户注册系统，需要存储用户的邮箱地址，并确保每个邮箱地址只被注册一次。

```
Jedis jedis = new Jedis("localhost", 6379);

// 添加用户邮箱地址到Set中，自动去重
jedis.sadd("user_emails", "user1@example.com");
jedis.sadd("user_emails", "user2@example.com");
jedis.sadd("user_emails", "user1@example.com"); // 重复添加，不会生效

// 获取所有用户邮箱地址
Set<String> emails = jedis.smembers("user_emails");
for (String email : emails) {
    System.out.println("Email: " + email);
}
```

- **集合运算**：Redis的Set数据结构支持 交集、并集、差集 等集合运算，这些运算在处理具有关联关系的数据时非常有用。其他比如 共同好友、共同关注的人 等。

假设有两个用户集合，分别表示喜欢篮球的用户和喜欢足球的用户。现在需要找出同时喜欢篮球和足球的用户。

```
Jedis jedis = new Jedis("localhost", 6379);
// 添加用户到集合中
jedis.sadd("basketball_fans", "user1", "user2", "user3");
jedis.sadd("football_fans", "user2", "user3", "user4");
// 计算交集，找出同时喜欢篮球和足球的用户
Set<String> commonFans = jedis.sinter("basketball_fans", "football_fans");
for (String fan : commonFans) {
    System.out.println("Common Fan: " + fan);
}
```

- **抽奖活动**：假设有一个抽奖活动，用户通过提交自己的ID来参与抽奖。现在需要确保每个用户只能中奖一次。

```
Jedis jedis = new Jedis("localhost", 6379);

// 用户提交ID参与抽奖
String userId = "user123";

// 检查用户是否已经中奖（是否已经在Set中存在）
if (!jedis.sismember("winners", userId)) {
    // 用户未中奖，将其添加到中奖者集合中
    jedis.sadd("winners", userId);
    System.out.println("Congratulations! User " + userId + " has won.");
} else {
    System.out.println("Sorry, User " + userId + " has already won.");
}
```

1.5 有序集合（Zset）

> 推荐参考：【大课堂】Redis中hash、set、zset的底层数据结构原理

Redis 的 Zset（有序集合）是一种特殊的数据结构，它类似于集合（Set），但每个元素都关联了一个分数，用于进行有序排序。
Zset 的底层原理基于 跳表 （Skip List）实现。
以下是 Zset 的一些特点和原理：

- 元素有序：元素按照分数进行有序排列。
- 分数：每个元素都有一个分数，用于确定排序顺序。
- 快速插入和删除：通过跳表实现高效的插入和删除操作。
- 范围查询：支持按照分数范围查询元素。

例如，假设有一个 Zset 用于存储学生的成绩，成绩作为分数：

```
ZADD scores 85 "Alice"
ZADD scores 92 "Bob"
ZADD scores 78 "Charlie"
```

可以通过以下方式进行查询：

- 按照分数排序获取所有学生：ZRANGE scores 0 -1。
- 获取分数在 80 到 90 之间的学生：ZRANGEBYSCORE scores 80 90。

Zset 在实现排行榜、实时搜索等场景中非常有用。它提供了高效的插入、删除和查询操作，同时能够保证元素的有序性。

应用场景：

- **排行榜**：排行榜是Zset数据结构最典型的应用场景之一。在游戏、社交媒体、电商平台等，经常需要根据用户的积分、等级、销量等数据进行排名。

假设有一个游戏，需要记录玩家的积分并展示积分排行榜。

```
Jedis jedis = new Jedis("localhost", 6379);

// 添加玩家积分到Zset中
jedis.zadd("game_scores", 1000, "player1");
jedis.zadd("game_scores", 1500, "player2");
jedis.zadd("game_scores", 800, "player3");

// 获取积分最高的前3名玩家
Set<Tuple> topPlayers = jedis.zrevrangeWithScores("game_scores", 0, 2);
for (Tuple player : topPlayers) {
    System.out.println("Player: " + player.getElement() + ", Score: " + player.getScore());
}

// 更新玩家积分
jedis.zincrby("game_scores", 50, "player1");

// 获取指定积分范围内的玩家
Set<Tuple> rangePlayers = jedis.zrangeByScoreWithScores("game_scores", 900, 1400);
for (Tuple player : rangePlayers) {
    System.out.println("Player: " + player.getElement() + ", Score: " + player.getScore());
}
```

- **延时队列**：在任务调度系统中，经常需要实现延时执行的功能，例如发送延时消息、定时任务等。Zset数据结构可以通过设置元素的权重为执行时间的时间戳，来实现延时队列的功能。

假设有一个延时消息系统，需要发送消息并在指定时间后消费。

```
Jedis jedis = new Jedis("localhost", 6379);

// 发送延时消息，设置消息的执行时间为当前时间+5秒
long delayTime = System.currentTimeMillis() + 5000;
jedis.zadd("delay_queue", delayTime, "message1");

// 消费者线程不断轮询延时队列，获取到执行时间小于等于当前时间的消息并消费
while (true) {
    Set<Tuple> readyMessages = jedis.zrangeByScoreWithScores("delay_queue", 0, System.currentTimeMillis());
    for (Tuple message : readyMessages) {
        String messageId = message.getElement();
        // 消费消息
        System.out.println("Consuming message: " + messageId);
        // 消费后从队列中移除
        jedis.zrem("delay_queue", messageId);
    }
    // 为了避免频繁轮询，可以设置一个短暂的休眠时间
    Thread.sleep(1000);
}
```

- **范围查找与筛选**：在数据处理和分析中，经常需要根据某个范围来查找和筛选数据。Zset数据结构可以通过设置元素的权重为需要筛选的字段值，来实现范围查找和筛选的功能。

假设有一个电商平台，需要根据商品的价格范围来筛选商品。

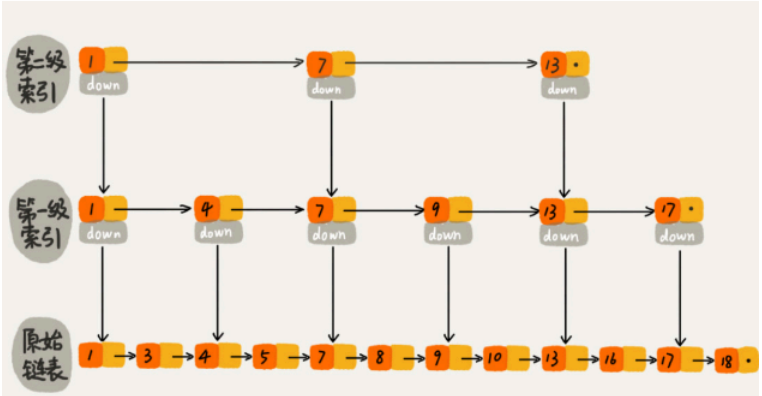
```
Jedis jedis = new Jedis("localhost", 6379);

// 添加商品到Zset中，设置商品的价格为权重
jedis.zadd("products", 100, "product1");
jedis.zadd("products", 200, "product2");
jedis.zadd("products", 150, "product3");

// 获取价格在100到200之间的商品
Set<Tuple> rangeProducts = jedis.zrangeByScoreWithScores("products", 100, 200);
for (Tuple product : rangeProducts) {
    String productId = product.getElement();
    double price = product.getScore();
    // 展示商品信息
    System.out.println("Product: " + productId + ", Price: " + price);
}
```

跳表

跳表全称为跳跃列表，它允许快速查询，插入和删除一个有序连续元素的数据链表。跳跃列表的平均查找和插入 时间复杂度 都是 O(logn)。快速查询是通过维护一个多层次的链表，且每一层链表中的元素是前一层链表元素的子集。所以它的查询速度媲美平衡二叉树，而且它的数据结构比平衡二叉树简单，结构示意图如下:



对于单链表来说，即使数据是已经排好序的，想要查询其中的一个数据，只能从头开始遍历链表，这样效率很低，时间复杂度很高，是 O(n)。跳表在 Redis 的 zset，leveldb 都有应用， 是替代平衡树的方案，其思想的复杂度较低 ，容易理解和接触。

2. 三个特殊数据结构

> 推荐参考：Redis三种特殊数据类型：HyperLogLog、BigMap、Geo

2.1 位图（bitmap）

- 特征：使用位数组来表示一系列布尔值，每个布尔值对应位数组中的一个位。
- 底层原理：位图通过位运算来高效地存储和查询大量布尔值。每个位可以独立地设置为0或1，代表对应元素的某种状态（例如，是否访问过）。
- 使用场景：适合用于统计和分析大规模数据，例如用户的活跃情况、网站的访问情况、商品的销售情况等。

2.2 基数统计（HyperLogLog）

- 特征：一种用于基数统计的算法，只需要使用很少的内存就能估计集合中不同元素的数量。
- 底层原理：基于概率计数原理，通过对每个元素进行哈希，并记录哈希值的最高位非零位的位置，从而估计集合的大小。随着元素的增加，算法会逐渐收敛到真实基数的近似值。
- 使用场景：适合用于统计网站的 UV、独立 IP 数、用户访问量等场景。

2.3 地理位置（Geo）

- 特征：使用有序集合（Sorted Set）来实现地理空间索引，有序集合中的每个成员都与一个经度和纬度相关联，成员按照分数（在地理空间中即距离）排序。
- 底层原理：Redis 使用 GeoHash 算法对地理位置进行编码，并将编码后的值作为有序集合的成员，距离作为分数。当执行地理空间相关的操作时（如查询附近地点），Redis 会根据GeoHash 值和给定的半径范围来检索符合条件的成员。
- 使用场景：适合存储和查询具有地理位置信息的数据，如用户位置、附近的商家、地理围栏等。