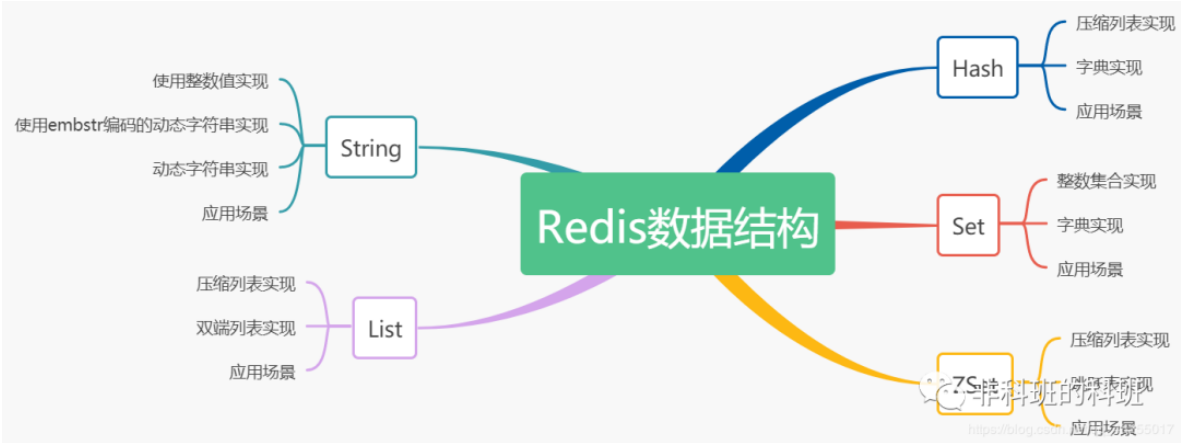
# 万字长文的Redis五种数据结构详解（理论+实战）

程序员大咖 *2020-08-15*

以下文章来源于非科班的科班，作者黎杜

## 本文脑图




## 前言

Redis是基于c语言编写的开源非关系型内存数据库，可以用作数据库、缓存、消息中间件，这么优秀的东西一定要一点一点的吃透它。

理论肯定是要用于实践的，因此最重要的还是实战部分，也就是这里还会讲解五种数据结构的应用场景。

话不多说，我们直接进入主题，很多人都知道Redis的五种数据结构包括以下五种：

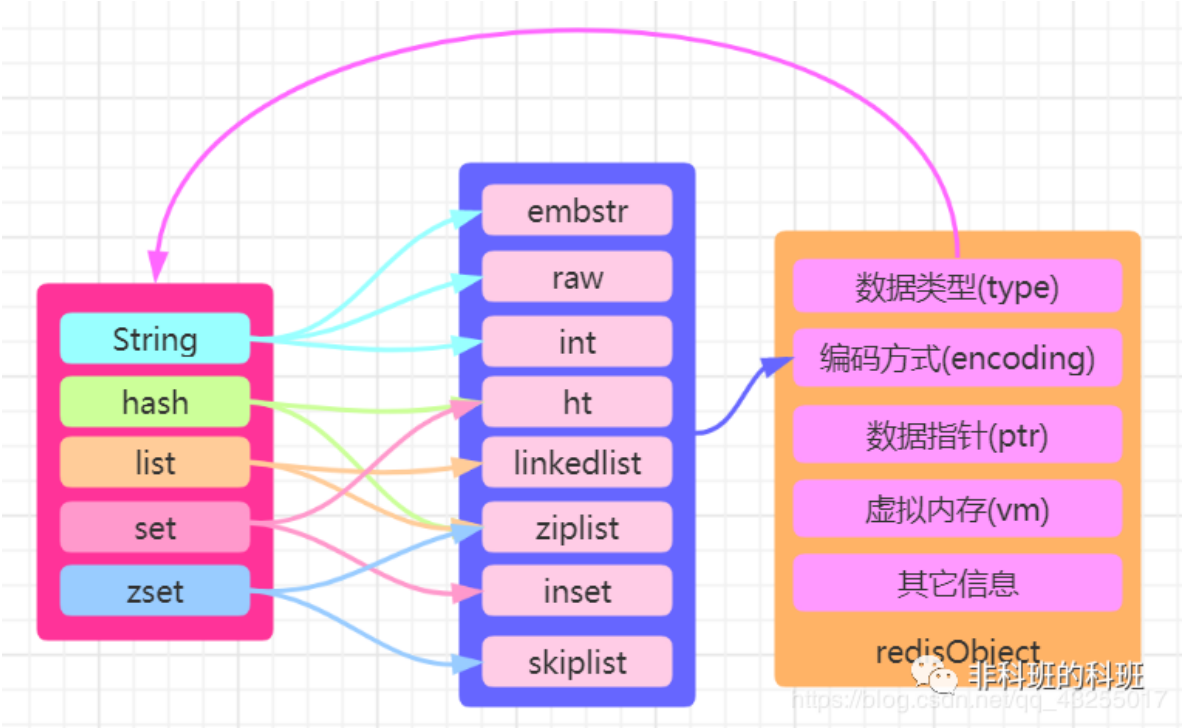
1. `String`：字符串类型
2. `List`：列表类型
3. `Set`：无序集合类型
4. `ZSet`：有序集合类型
5. `Hash`：哈希表类型

但是作为一名优秀的程序员可能不能只停留在只会用这五种类型进行crud工作，还是得深入了解这五种数据结构的底层原理。

## Redis核心对象

在Redis中有一个**「核心的对象」**叫做`redisObject`，是用来表示所有的key和value的，用redisObject结构体来表示`String、Hash、List、Set、ZSet`五种数据类型。

`redisObject`的源代码在`redis.h`中，使用c语言写的，感兴趣的可以自行查看，关于redisObject我这里画了一张图，表示redisObject的结构如下所示：




闪瞎人的五颜六色图

在redisObject中**「type表示属于哪种数据类型，encoding表示该数据的存储方式」**，也就是底层的实现的该数据类型的数据结构。因此这篇文章具体介绍的也是encoding对应的部分。

那么encoding中的存储类型又分别表示什么意思呢？具体数据类型所表示的含义，如下图所示：

**表 8-4　不同类型和编码的对象**

| 类　型 | 编　码 | 对　象 |
|---|---|---|
| REDIS_STRING | REDIS_ENCODING_INT | 使用整数值实现的字符串对象 |
| REDIS_STRING | REDIS_ENCODING_EMBSTR | 使用 embstr 编码的简单动态字符串实现的字符串对象 |
| REDIS_STRING | REDIS_ENCODING_RAW | 使用简单动态字符串实现的字符串对象 |
| REDIS_LIST | REDIS_ENCODING_ZIPLIST | 使用压缩列表实现的列表对象 |
| REDIS_LIST | REDIS_ENCODING_LINKEDLIST | 使用双端链表实现的列表对象 |
| REDIS_HASH | REDIS_ENCODING_ZIPLIST | 使用压缩列表实现的哈希对象 |
| REDIS_HASH | REDIS_ENCODING_HT | 使用字典实现的哈希对象 |
| REDIS_SET | REDIS_ENCODING_INTSET | 使用整数集合实现的集合对象 |
| REDIS_SET | REDIS_ENCODING_HT | 使用字典实现的集合对象 |
| REDIS_ZSET | REDIS_ENCODING_ZIPLIST | 使用压缩列表实现的有序集合对象 |
| REDIS_ZSET | REDIS_ENCODING_SKIPLIST | 使用跳跃表和字典实现的有序集合对象 |



图片截图出自《Redis设计与实现第二版》

可能看完这图，还是觉得一脸懵。不慌，会进行五种数据结构的详细介绍，这张图只是让你找到每种中数据结构对应的储存类型有哪些，大概脑子里有个印象。

举一个简单的例子，你在Redis中设置一个字符串`key 234`，然后查看这个字符串的存储类型就会看到为int类型，非整数型的使用的是embstr储存类型，具体操作如下图所示：

```
127.0.0.1:6379> set key 234
OK
127.0.0.1:6379> object encoding key
"int"
127.0.0.1:6379> set k2 3.200
OK
127.0.0.1:6379> object encoding k2
"embstr"
```

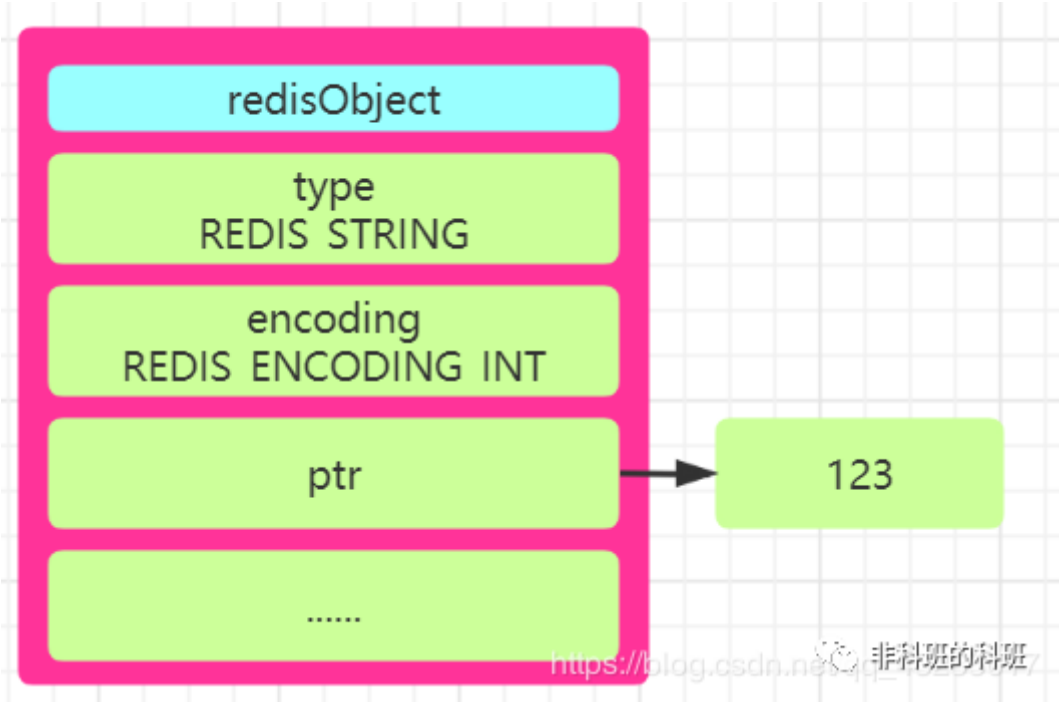
# String类型

String是Redis最基本的数据类型，上面的简介中也说到Redis是用c语言开发的。但是Redis中的字符串和c语言中的字符串类型却是有明显的区别。

String类型的数据结构存储方式有三种`int、raw、embstr`。那么这三种存储方式有什么区别呢?

## int

Redis中规定假如存储的是**「整数型值」**，比如`set num 123`这样的类型，就会使用 int的存储方式进行存储，在redisObject的**「ptr属性」**中就会保存该值。



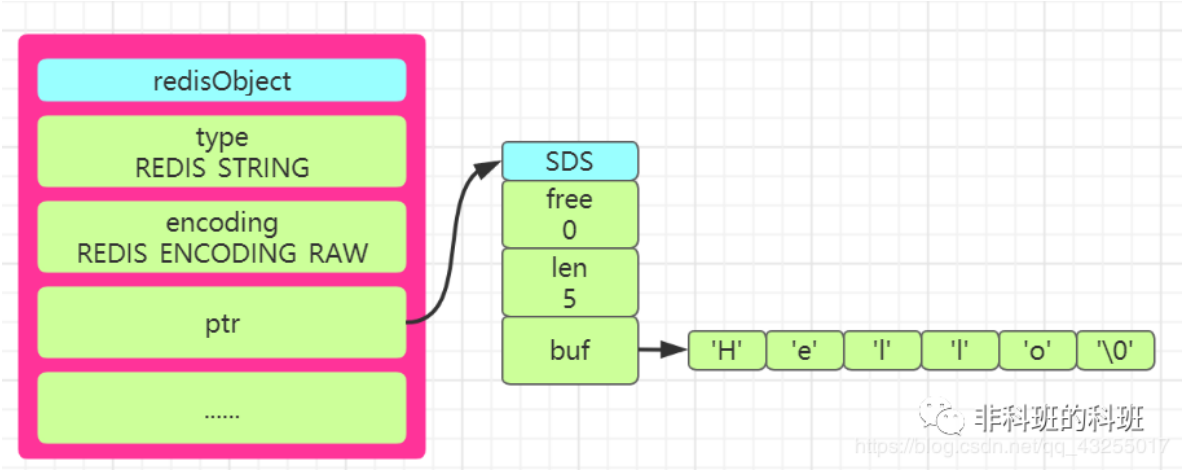

## SDS

假如存储的**「字符串是一个字符串值并且长度大于32个字节」**就会使用`SDS（simple dynamic string）`方式进行存储，并且encoding设置为raw；若是**「字符串长度小于等于32个字节」**就会将encoding改为embstr来保存字符串。

SDS称为**「简单动态字符串」**，对于SDS中的定义在Redis的源码中有的三个属性`int len、int free、char buf[]`。

len保存了字符串的长度，free表示buf数组中未使用的字节数量，buf数组则是保存字符串的每一个字符元素。

因此当你在Redsi中存储一个字符串Hello时，根据Redis的源代码的描述可以画出SDS的形式的redisObject结构图如下图所示:




## SDS与c语言字符串对比

Redis使用SDS作为存储字符串的类型肯定是有自己的优势，SDS与c语言的字符串相比，SDS对c语言的字符串做了自己的设计和优化，具体优势有以下几点：

（1）c语言中的字符串并不会记录自己的长度，因此**「每次获取字符串的长度都会遍历得到，时间的复杂度是O(n)」**，而Redis中获取字符串只要读取len的值就可，时间复杂度变为O(1)。

（2）**「c语言」**中两个字符串拼接，若是没有分配足够长度的内存空间就**「会出现缓冲区溢出的情况」**；而**「SDS」**会先根据len属性判断空间是否满足要求，若是空间不够，就会进行相应的空间扩展，所以**「不会出现缓冲区溢出的情况」**。

（3）SDS还提供**「空间预分配」**和**「惰性空间释放」**两种策略。在为字符串分配空间时，分配的空间比实际要多，这样就能**「减少连续的执行字符串增长带来内存重新分配的次数」**。

当字符串被缩短的时候，SDS也不会立即回收不适用的空间，而是通过`free`属性将不使用的空间记录下来，等后面使用的时候再释放。

具体的空间预分配原则是：**「当修改字符串后的长度len小于1MB，就会预分配和len一样长度的空间，即len=free；若是len大于1MB，free分配的空间大小就为1MB」**。

（4）SDS是二进制安全的，除了可以储存字符串以外还可以储存二进制文件（如图片、音频，视频等文件的二进制数据）；而c语言中的字符串是以空字符串作为结束符，一些图片中含有结束符，因此不是二进制安全的。

为了方便易懂，做了一个c语言的字符串和SDS进行对比的表格，如下所示：

| c语言字符串 | SDS |
| --- | --- |
| 获取长度的时间复杂度为O(n) | 获取长度的时间复杂度为O(1) |
| 不是二进制安全的 | 是二进制安全的 |
| 只能保存字符串 | 还可以保存二进制数据 |
| n次增长字符串必然会带来n次的内存分配 | n次增长字符串内存分配的次数<=n |

## String类型应用

说到这里我相信很多人可以说已经精通Redis的String类型了，但是纯理论的精通，理论还是得应用实践，上面说到String可以用来存储图片，现在就以图片存储作为案例实现。

（1）首先要把上传得图片进行编码，这里写了一个工具类把图片处理成了Base64得编码形式，具体得实现代码如下：

```
/**
     * 将图片内容处理成Base64编码格式
     * @param file
     * @return
     */
    public static String encodeImg(MultipartFile file) {
        byte[] imgBytes = null;
        try {
            imgBytes = file.getBytes();
        } catch (IOException e) {
            e.printStackTrace();
        }
        BASE64Encoder encoder = new BASE64Encoder();
        return imgBytes==null?null:encoder.encode(imgBytes );
    }
```

（2）第二步就是把处理后的图片字符串格式存储进Redis中，实现的代码如下所示：

```
    /**
     * Redis存储图片
     * @param file
     * @return
     */
    public void uploadImageServiceImpl(MultipartFile image) {
        String imgId = UUID.randomUUID().toString();
        String imgStr= ImageUtils.encodeImg(image);
        redisUtils.set(imgId , imgStr);
        // 后续操作可以把imgId存进数据库对应的字段，如果需要从redis中取出，只要获取到这个字段后从redis中取出即可。
    }
```

这样就是实现了图片得二进制存储，当然String类型得数据结构得应用也还有常规计数：**「统计微博数、统计粉丝数」**等。

# Hash类型

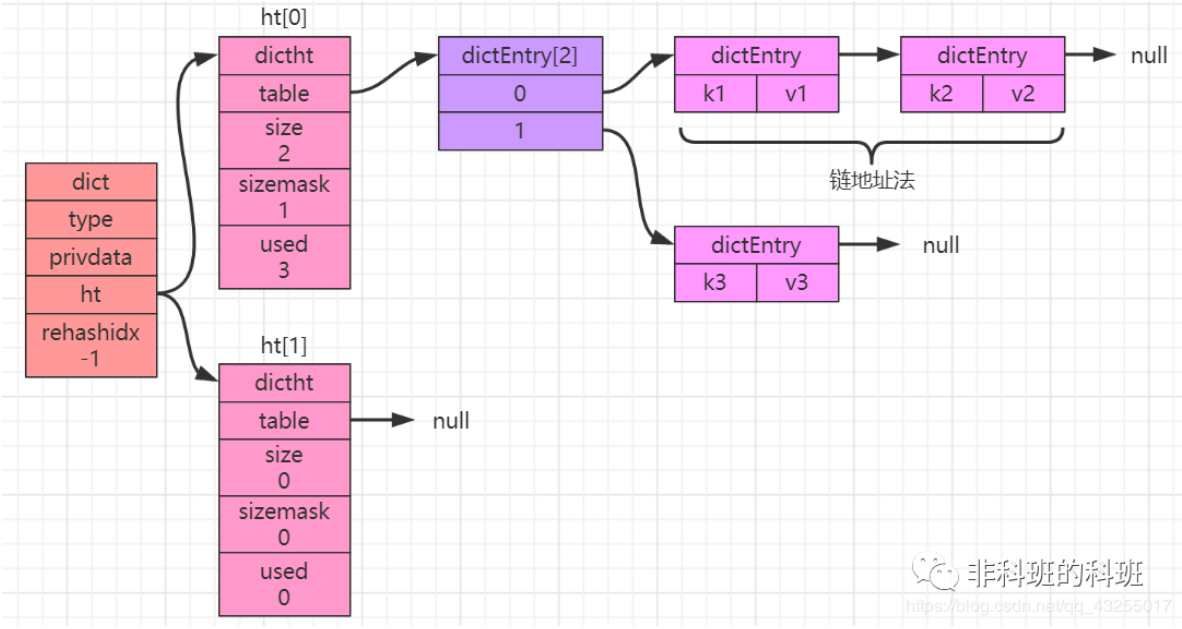
Hash对象的实现方式有两种分别是`ziplist、hashtable`，其中hashtable的存储方式key是String类型的，value也是以`key value`的形式进行存储。

字典类型的底层就是hashtable实现的，明白了字典的底层实现原理也就是明白了hashtable的实现原理，hashtable的实现原理可以与HashMap的是底层原理相类比。

## 字典

两者在新增时都会通过key计算出数组下标，不同的是计算法方式不同，HashMap中是以hash函数的方式，而hashtable中计算出hash值后，还要通过sizemask 属性和哈希值再次得到数组下标。

我们知道hash表最大的问题就是hash冲突，为了解决hash冲突，假如hashtable中不同的key通过计算得到同一个index，就会形成单向链表（**「链地址法」**），如下图所示：




## rehash

在字典的底层实现中，value对象以每一个dictEntry的对象进行存储，当hash表中的存放的键值对不断的增加或者减少时，需要对hash表进行一个扩展或者收缩。

这里就会和HashMap一样也会就进行rehash操作，进行重新散列排布。从上图中可以看到有`ht[0]`和`ht[1]`两个对象，先来看看对象中的属性是干嘛用的。

在hash表结构定义中有四个属性分别是`dictEntry **table、unsigned long size、unsigned long sizemask、unsigned long used`，分别表示的含义就是**「哈希表数组、hash表大小、用于计算索引值，总是等于size-1、hash表中已有的节点数」**。

ht[0]是用来最开始存储数据的，当要进行扩展或者收缩时，ht[0]的大小就决定了ht[1]的大小，ht[0]中的所有的键值对就会重新散列到ht[1]中。

扩展操作：ht[1]扩展的大小是比当前 ht[0].used 值的二倍大的第一个 2 的整数幂；收缩操作：ht[0].used 的第一个大于等于的 2 的整数幂。

当ht[0]上的所有的键值对都rehash到ht[1]中，会重新计算所有的数组下标值，当数据迁移完后ht[0]就会被释放，然后将ht[1]改为ht[0]，并新创建ht[1]，为下一次的扩展和收缩做准备。

## 渐进式rehash

假如在rehash的过程中数据量非常大，Redis不是一次性把全部数据rehash成功，这样会导致Redis对外服务停止，Redis内部为了处理这种情况采用**「渐进式的rehash」**。

Redis将所有的rehash的操作分成多步进行，直到都rehash完成，具体的实现与对象中的`rehashindex`属性相关，**「若是rehashindex 表示为-1表示没有rehash操作」**。

当rehash操作开始时会将该值改成0，在渐进式rehash的过程**「更新、删除、查询会在ht[0]和ht[1]中都进行」**，比如更新一个值先更新ht[0]，然后再更新ht[1]。

而新增操作直接就新增到ht[1]表中，ht[0]不会新增任何的数据，这样保证**「ht[0]只减不增，直到最后的某一个时刻变成空表」**，这样rehash操作完成。

上面就是字典的底层hashtable的实现原理，说完了hashtable的实现原理，我们再来看看Hash数据结构的两一种存储方式**「ziplist（压缩列表）」**

# ziplist

压缩列表（ziplist）是一组连续内存块组成的顺序的数据结构，压缩列表能够节省空间，压缩列表中使用多个节点来存储数据。

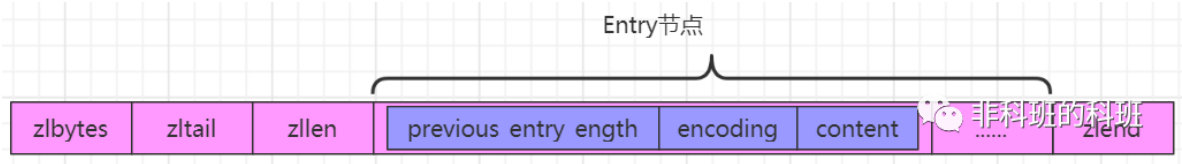
压缩列表是列表键和哈希键底层实现的原理之一，**「压缩列表并不是以某种压缩算法进行压缩存储数据，而是它表示一组连续的内存空间的使用，节省空间」**，压缩列表的内存结构图如下：


压缩列表中每一个节点表示的含义如下所示：

1. `zlbytes`：4个字节的大小，记录压缩列表占用内存的字节数。
2. `zltail`：4个字节大小，记录表尾节点距离起始地址的偏移量，用于快速定位到尾节点的地址。
3. `zllen`：2个字节的大小，记录压缩列表中的节点数。
4. `entry`：表示列表中的每一个节点。
5. `zlend`：表示压缩列表的特殊结束符号`'0xFF'`。

再压缩列表中每一个entry节点又有三部分组成，包括`previous_entry_ength`、`encoding`、`content`。

1. `previous_entry_ength`表示前一个节点entry的长度，可用于计算前一个节点的其实地址，因为他们的地址是连续的。
2. encoding：这里保存的是content的内容类型和长度。
3. content：content保存的是每一个节点的内容。




说到这里相信大家已经都hash这种数据结构已经非常了解，若是第一次接触Redis五种基本数据结构的底层实现的话，建议多看几遍，下面来说一说hash的应用场景。

# 应用场景

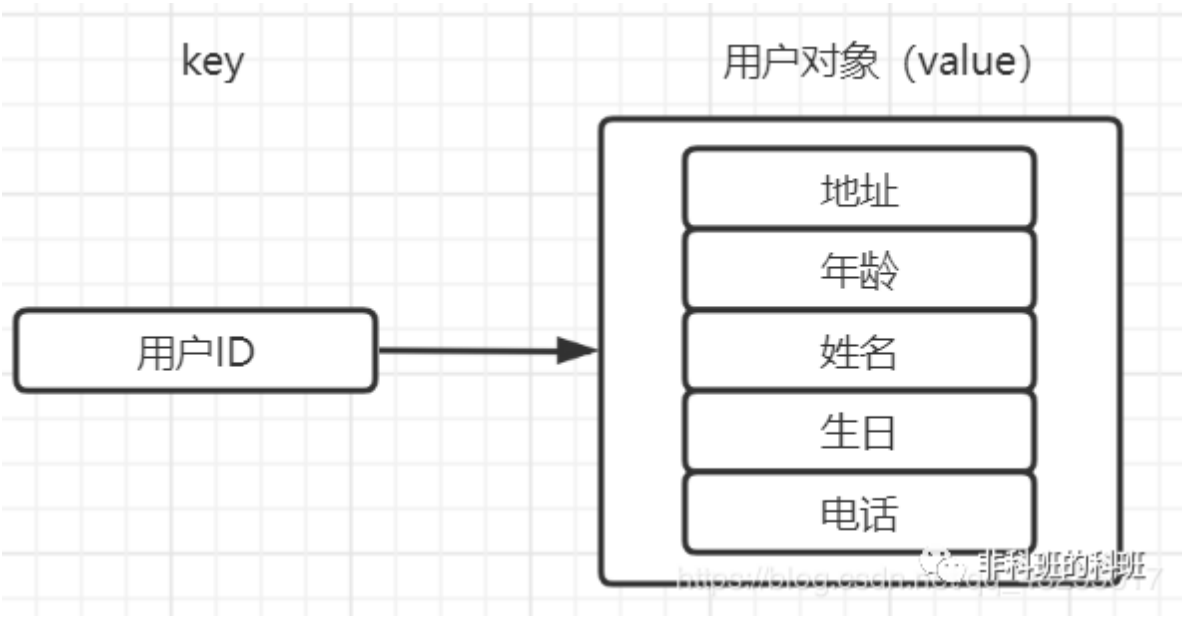
哈希表相对于String类型存储信息更加直观，存储更加方便，经常会用来做用户数据的管理，存储用户的信息。

hash也可以用作高并发场景下使用Redis生成唯一的id。下面我们就以这两种场景用作案例编码实现。

## 存储用户数据

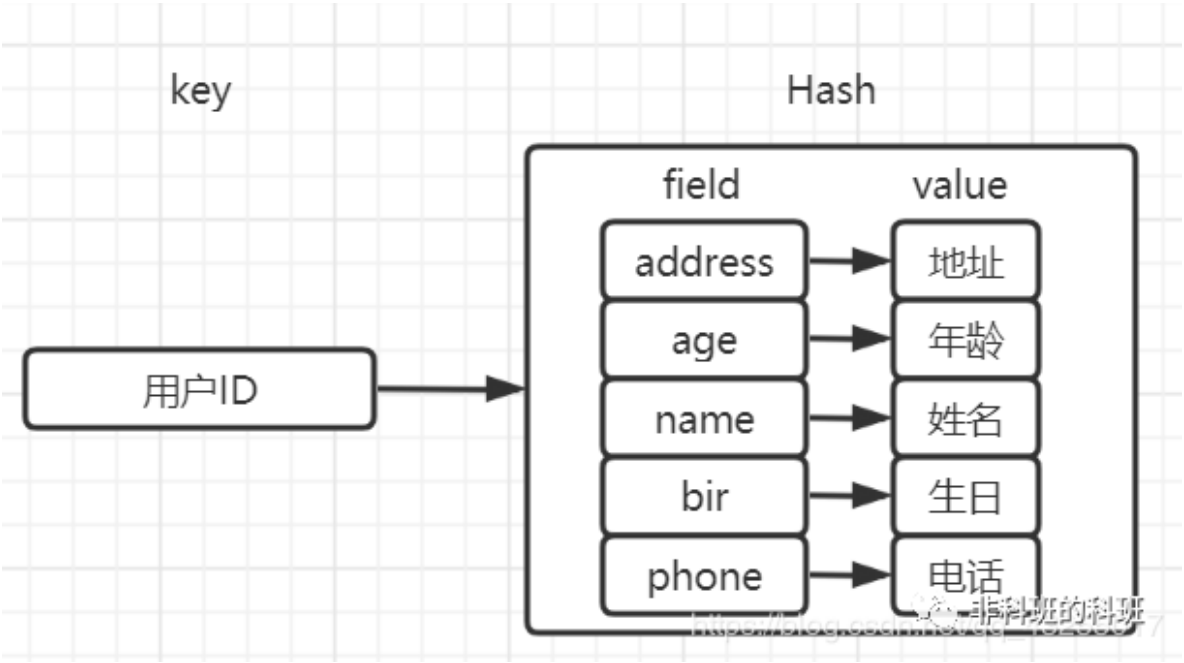
第一个场景比如我们要储存用户信息，一般使用用户的ID作为key值，保持唯一性，用户的其他信息（地址、年龄、生日、电话号码等）作为value值存储。

若是传统的实现就是将用户的信息封装成为一个对象，通过序列化存储数据，当需要获取用户信息的时候，就会通过反序列化得到用户信息。




但是这样必然会造成序列化和反序列化的性能的开销，并且若是只修改其中的一个属性值，就需要把整个对象序列化出来，操作的动作太大，造成不必要的性能开销。

若是使用Redis的hash来存储用户数据，就会将原来的value值又看成了一个k v形式的存储容器，这样就不会带来序列化的性能开销的问题。




## 分布式生成唯一ID

第二个场景就是生成分布式的唯一ID，这个场景下就是把redis封装成了一个工具类进行实现，实现的代码如下：

```
    // offset表示的是id的递增梯度值
    public Long getId(String key,String hashKey,Long offset) throws
BusinessException{
        try {
            if (null == offset) {
                offset=1L;
            }
            // 生成唯一id
            return redisUtil.increment(key, hashKey, offset);
        } catch (Exception e) {
```

```
            //若是出现异常就是用uuid来生成唯一的id值
            int randNo=UUID.randomUUID().toString().hashCode();
            if (randNo < 0) {
                randNo=-randNo;
            }
            return Long.valueOf(String.format("%16d", randNo));
        }
    }
```

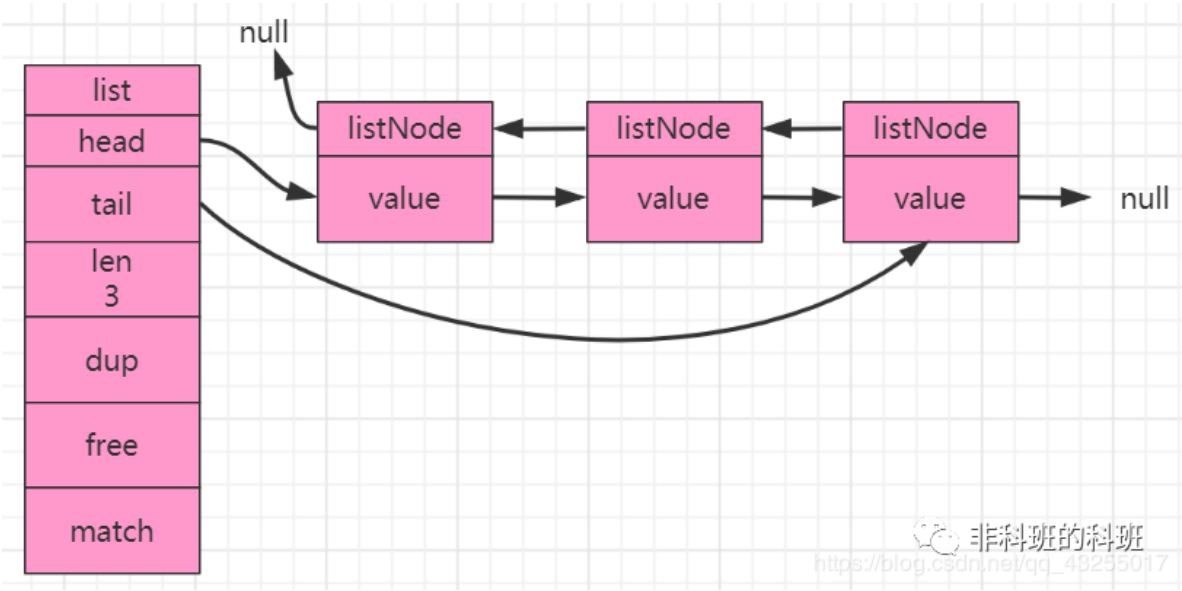
# List类型

Redis中的列表在3.2之前的版本是使用`ziplist`和`linkedlist`进行实现的。在3.2之后的版本就是引入了`quicklist`。

ziplist压缩列表上面已经讲过了，我们来看看linkedlist和quicklist的结构是怎么样的。

linkedlist是一个双向链表，他和普通的链表一样都是由指向前后节点的指针。插入、修改、更新的时间复杂度尾O(1)，但是查询的时间复杂度确实O(n)。

linkedlist和quicklist的底层实现是采用链表进行实现，在c语言中并没有内置的链表这种数据结构，Redis实现了自己的链表结构。




Redis中链表的特性：

1. 每一个节点都有指向前一个节点和后一个节点的指针。
2. 头节点和尾节点的prev和next指针指向为null，所以链表是无环的。
3. 链表有自己长度的信息，获取长度的时间复杂度为O(1)。

Redis中List的实现比较简单，下面我们就来看看它的应用场景。

# 应用场景

Redis中的列表可以实现**「阻塞队列」**，结合lpush和brpop命令就可以实现。生产者使用lupsh从列表的左侧插入元素，消费者使用brpop命令从队列的右侧获取元素进行消费。

（1）首先配置redis的配置，为了方便我就直接放在`application.yml`配置文件中，实际中可以把redis的配置文件放在一个`redis.properties`文件单独放置，具体配置如下：

```
spring
redis:
host: 127.0.0.1
port: 6379
password: user
timeout: 0
database: 2
pool:
max-active: 100
max-idle: 10
min-idle: 0
max-wait: 100000
```

（2）第二步创建redis的配置类，叫做`RedisConfig`，并标注上`@Configuration`注解，表明他是一个配置类。

```
@Configuration
public class RedisConfiguration {

@Value("${spring.redis.host}")
private String host;
@Value("${spring.redis.port}")
private int port;
@Value("${spring.redis.password}")
private String password;
@Value("${spring.redis.pool.max-active}")
private int maxActive;
@Value("${spring.redis.pool.max-idle}")
private int maxIdle;
@Value("${spring.redis.pool.min-idle}")
private int minIdle;
@Value("${spring.redis.pool.max-wait}")
private int maxWait;
@Value("${spring.redis.database}")
private int database;
@Value("${spring.redis.timeout}")
private int timeout;

@Bean
public JedisPoolConfig getRedisConfiguration(){
JedisPoolConfig jedisPoolConfig= new JedisPoolConfig();
jedisPoolConfig.setMaxTotal(maxActive);
jedisPoolConfig.setMaxIdle(maxIdle);
jedisPoolConfig.setMinIdle(minIdle);
jedisPoolConfig.setMaxWaitMillis(maxWait);
return jedisPoolConfig;
}

@Bean
public JedisConnectionFactory getConnectionFactory() {
JedisConnectionFactory factory = new JedisConnectionFactory();
factory.setHostName(host);
factory.setPort(port);
factory.setPassword(password);
factory.setDatabase(database);
JedisPoolConfig jedisPoolConfig= getRedisConfiguration();
```

```
factory.setPoolConfig(jedisPoolConfig);
return factory;
}

@Bean
public RedisTemplate<?, ?> getRedisTemplate() {
JedisConnectionFactory factory = getConnectionFactory();
RedisTemplate<?, ?> redisTemplate = new StringRedisTemplate(factory);
return redisTemplate;
}
}
```

（3）第三步就是创建Redis的工具类RedisUtil，自从学了面向对象后，就喜欢把一些通用的东西拆成工具类，好像一个一个零件，需要的时候，就把它组装起来。

```
@Component
public class RedisUtil {

@Autowired
private RedisTemplate<String, Object> redisTemplate;
/**
* 存消息到消息队列中
* @param key 键
* @param value 值
* @return
*/
public boolean lPushMessage(String key, Object value) {
try {
redisTemplate.opsForList().leftPush(key, value);
return true;
} catch (Exception e) {
e.printStackTrace();
return false;
}
}

/**
* 从消息队列中弹出消息
* @param key 键
* @return
*/
public Object rPopMessage(String key) {
try {
return redisTemplate.opsForList().rightPop(key);
} catch (Exception e) {
e.printStackTrace();
return null;
}
}

/**
* 查看消息
* @param key 键
* @param start 开始
* @param end 结束 0 到 -1代表所有值
* @return
*/
```

```
public List<Object> getMessage(String key, long start, long end) {
try {
return redisTemplate.opsForList().range(key, start, end);
} catch (Exception e) {
e.printStackTrace();
return null;
}
}
```

这样就完成了Redis消息队列工具类的创建，在后面的代码中就可以直接使用。

# Set集合

Redis中列表和集合都可以用来存储字符串，但是**「Set是不可重复的集合，而List列表可以存储相同的字符串」**，Set集合是无序的这个和后面讲的ZSet有序集合相对。

Set的底层实现是**「ht和intset」**，ht（哈希表）前面已经详细了解过，下面我们来看看inset类型的存储结构。

inset也叫做整数集合，用于保存整数值的数据结构类型，它可以保存`int16_t`、`int32_t`或者`int64_t`的整数值。

在整数集合中，有三个属性值`encoding、length、contents[]`，分别表示编码方式、整数集合的长度、以及元素内容，length就是记录contents里面的大小。

在整数集合新增元素的时候，若是超出了原集合的长度大小，就会对集合进行升级，具体的升级过程如下：

1. 首先扩展底层数组的大小，并且数组的类型为新元素的类型。
2. 然后将原来的数组中的元素转为新元素的类型，并放到扩展后数组对应的位置。
3. 整数集合升级后就不会再降级，编码会一直保持升级后的状态。

## 应用场景

Set集合的应用场景可以用来**「去重、抽奖、共同好友、二度好友」**等业务类型。接下来模拟一个添加好友的案例实现：

```
@RequestMapping(value = "/addFriend", method = RequestMethod.POST)
public Long addFriend(User user, String friend) {
    String currentKey = null;
    // 判断是否是当前用户的好友
    if (AppContext.getCurrentUser().getId().equals(user.getId)) {
        currentKey = user.getId.toString();
    }
    //若是返回0则表示不是该用户好友
    return currentKey==null?0l:setOperations.add(currentKey, friend);
}
```

假如两个用户A和B都是用上上面的这个接口添加了很多的自己的好友，那么有一个需求就是要实现获取A和B的共同好友，那么可以进行如下操作：

```
public Set intersectFriend(User userA, User userB) {
    return setOperations.intersect(userA.getId.toString(),
userB.getId.toString());
}
```

举一反三，还可以实现A用户自己的好友，或者B用户自己的好友等，都可以进行实现。

# ZSet集合

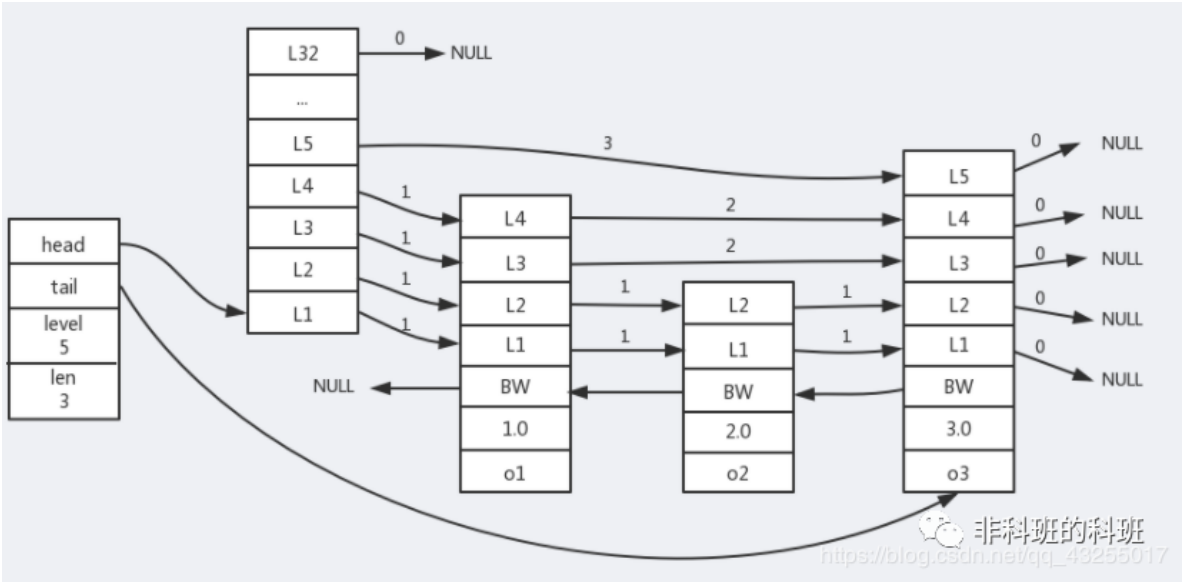
ZSet是有序集合，从上面的图中可以看到ZSet的底层实现是`ziplist`和`skiplist`实现的，ziplist上面已经详细讲过，这里来讲解skiplist的结构实现。

`skiplist`也叫做**「跳跃表」**，跳跃表是一种有序的数据结构，它通过每一个节点维持多个指向其它节点的指针，从而达到快速访问的目的。

skiplist有如下几个特点：

1. 有很多层组成，由上到下节点数逐渐密集，最上层的节点最稀疏，跨度也最大。
2. 每一层都是一个有序链表，至少包含两个节点，头节点和尾节点。
3. 每一层的每一个每一个节点都含有指向同一层下一个节点和下一层同一个位置节点的指针。
4. 如果一个节点在某一层出现，那么该以下的所有链表同一个位置都会出现该节点。

具体实现的结构图如下所示：




在跳跃表的结构中有head和tail表示指向头节点和尾节点的指针，能快速的实现定位。level表示层数，len表示跳跃表的长度，BW表示后退指针，在从尾向前遍历的时候使用。

BW下面还有两个值分别表示分值（score）和成员对象（各个节点保存的成员对象）。

跳跃表的实现中，除了最底层的一层保存的是原始链表的完整数据，上层的节点数会越来越少，并且跨度会越来越大。

跳跃表的上面层就相当于索引层，都是为了找到最后的数据而服务的，数据量越大，条表所体现的查询的效率就越高，和平衡树的查询效率相差无几。

# 应用场景

因为ZSet是有序的集合，因此ZSet在实现排序类型的业务是比较常见的，比如在首页推荐10个最热门的帖子，也就是阅读量由高到低，排行榜的实现等业务。

下面就选用获取排行榜前前10名的选手作为案例实现，实现的代码如下所示：

```
@Autowired
private RedisTemplate redisTemplate;
/**
 * 获取前10排名
 * @return
```

```
 */
    public static List<levelVO > getZset(String key, long baseNum, LevelService
levelService){
        ZSetOperations<Serializable, Object> operations =
redisTemplate.opsForZSet();
        // 根据score分数值获取前10名的数据
        Set<ZSetOperations.TypedTuple<Object>> set =
operations.reverseRangeWithScores(key,0,9);
        List<LevelVO> list= new ArrayList<LevelVO>();
        int i=1;
        for (ZSetOperations.TypedTuple<Object> o:set){
            int uid = (int) o.getValue();
            LevelCache levelCache = levelService.getLevelCache(uid);
            LevelVO levelVO = levelCache.getLevelVO();
            long score = (o.getScore().longValue() - baseNum + levelVO
.getCtime())/CommonUtil.multiplier;
            levelVO .setScore(score);
            levelVO .setRank(i);
            list.add( levelVO );
            i++;
        }
        return list;
    }
```

以上的代码实现大致逻辑就是根据score分数值获取前10名的数据，然后封装成lawyerVO对象的列表进行返回。

到这里我们已经精通Redis的五种基本数据类型了，又可以去和面试官扯皮了，扯不过就跑路吧，或者这篇文章多看几遍，相信对你总是有好处的。

【文章参考】

[1]《Redis设计与实现》