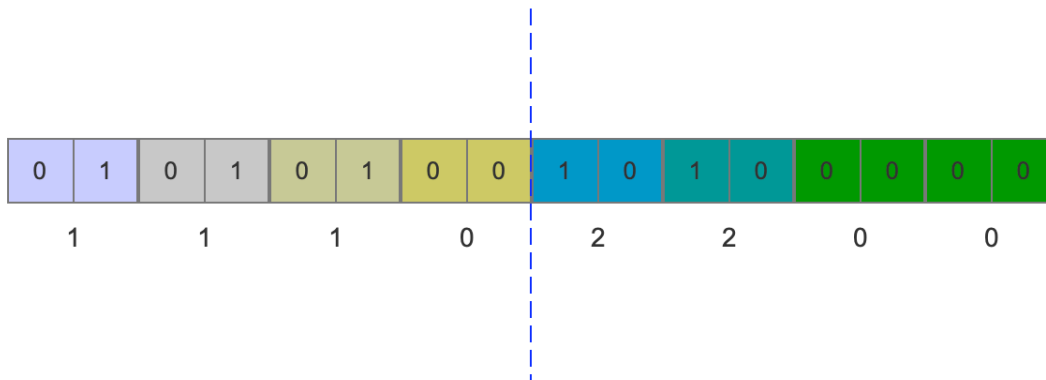


BulkOperationPacked

BulkOperation类的子类BulkOperationPacked，提供了很多对整数(integers)的压缩存储方法，其压缩存储过程其实就是对数据进行编码，将每一个整数（long或者int）编码为固定大小进行存储，大小取决于最大的那个值所需要的bit位个数。优点是减少了存储空间，并且对编码后的数据能够提供随机访问的功能。

1 例如有以下的数据{1, 1, 1, 0, 2, 2, 0, 0}，二进制的表示为{01, 01, 01, 0, 10, 10, 0, 0}，存储需要32个字节大小的空间。数据中最大的值是2，需要2个bit位即可表示，所以其他数据统一用2个bit位固定大小来表示，编码后需要的空间如下图：

图1：



如上图所示，编码后只需要2个字节的空间大小。

encode 源码解析

<https://github.com/luxugang/Lucene-7.5.0/blob/master/solr-7.5.0/lucene/core/src/java/org/apache/lucene/util/packed/BulkOperationPacked.java> 中给出了详细的注释，根据不同的 bitsPerValue(最大值需要的bit位个数)，BulkOperationPacked有几十个子类，但是编码操作都是调用了父类BulkOperationPacked的encode方法，仅仅是部分BulkOperationPacked子类的decode方法不同一共有四种encode方法：

- long[]数组编码至byte[]数组
- int[]数组编码至byte[]数组
- long[]数组编码至long[]数组
- int[]数组编码至long[]数组

在下面的代码中，挑选了其中一种encode方法，即将 long[]数组编码至byte[]数组，出于仅仅对encode逻辑的介绍，所以简化了部分代码，比如iterations，byteValueCount变量。这些变量不影响对encode过程的理解。下面的代码大家可以直接运行测试。

```
1 public class Encode {
```

```

2   public static byte[] encode(long[] values, int bitsPerValue) {
3       byte[] blocks = new byte[2];
4       int blocksOffset = 0;
5       int nextBlock = 0;
6       int bitsLeft = 8;
7       int valuesOffset = 0;
8       for (int i = 0; i < values.length; ++i) {
9           final long v = values[valuesOffset++];
10          // bitsPerValue指的是每一个元素需要占用的bit位数, 这个值取决于数据中最大元素需
          要的bit位
11          // 也就是每一个元素不管大小, 占用的bit位数都是一致的
12          if (bitsPerValue < bitsLeft) {
13              // 将当前处理的数值写到nextBlock中
14              nextBlock |= v << (bitsLeft - bitsPerValue);
15              bitsLeft -= bitsPerValue;
16          } else {
17              // flush as many blocks as possible
18              int bits = bitsPerValue - bitsLeft;
19              // nextBlock | (v >>> bits)的操作将v值存储到nextBlock中
20              // 然后将nextBlock的值存储到blocks[]数组中, 完成一个字节的压缩
21              blocks[blocksOffset++] = (byte) (nextBlock | (v >>> bits));
22              while (bits >= 8) {
23                  bits -= 8;
24                  // 将一个数组分成多块(按照一个8个bit大小划分)存储
25                  blocks[blocksOffset++] = (byte) (v >>> bits);
26              }
27              // then buffer
28              bitsLeft = 8 - bits;
29              // 把v的值的剩余部分存放到下一个nextBlock中, 也就是当前的v值的部分值会跟下一
              个v值的数据(可能是部分数据)混合存储到同一个字节中
30              // 这里说明了数据是连续存储, 可能分布在不同的block中
31              nextBlock = (int) ((v & ((1L << bits) - 1)) << bitsLeft);
32          }
33      }
34      return blocks;
35  }
36
37  public static void main(String[] args) {
38      long[] array = {1, 1, 1, 0, 2, 2, 0, 0};
39      byte[] result = Encode.encode(array, 2);
40      for (int a: result
41          ) {
42          System.out.println(a);
43      }
44  }
45  }

```

输出的结果是 84, -96。

decode 源码解析

全解码

不同的BulkOperationPacked子类有着不同的decode方法，解码的方法不尽相同，在Lucene7.5.0版本中，共有14种解码方法，尽管有这么多，但是逻辑都是大同小异。由于在上文中我们设置的bitPerValue的值为2，所以我们就介绍对应的decode方法。

```
1 public class Decode {
2     public static void decode(byte[] blocks, int blocksOffset, long[]
values, int valuesOffset, int iterations) {
3         for (int j = 0; j < iterations; ++j) {
4             final byte block = blocks[blocksOffset++];
5             // 因为bitPerValues的值为2，所以每次取2个bit位的数据即可
6             values[valuesOffset++] = (block >>> 6) & 3;
7             values[valuesOffset++] = (block >>> 4) & 3;
8             values[valuesOffset++] = (block >>> 2) & 3;
9             values[valuesOffset++] = block & 3;
10            // 至此，我们取出了1个字节的的所有数据，这些数据包含了4个编码前的数据
11        }
12    }
13
14    public static void main(String[] args) {
15        long[] array = {1, 1, 1, 0, 2, 2, 0, 0};
16        byte[] result = Encode.encode(array, 2);
17        System.out.println("Encode");
18        for (long a: result
19        ) {
20            System.out.println(a);
21        }
22        long [] arrayDecode = new long[8];
23        // 每次解码都是对一个字节操作，而编码后的byte[]数组有2个字节，所以iterations参数
为2
24        Decode.decode(result, 0, arrayDecode, 0, 2);
25        System.out.println("Decode");
26        for (long a : arrayDecode
27        ) {
28            System.out.println(a);
29        }
30    }
31 }
```

随机解码(随机访问)

上面的decode()方法对所有的数据，按照在byte[]数组中的顺序依次解码，下面介绍的就是这种编码带有的随机访问(随机解码)的功能。

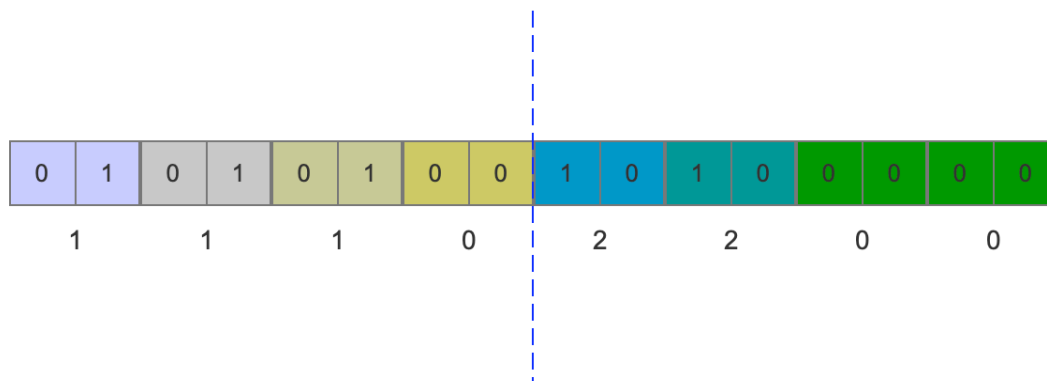
下面的get()方法在DirectReader类中实现。根据bitsPerValue的值(encode阶段, 固定大小的bit位数), 本篇博客中仅列出bitsPerValue值为2的解码方法, 对应上文的encode方法。更具体的注释请查看GitHub: <https://github.com/luxugang/Lucene-7.5.0/blob/master/solr-7.5.0/lucene/core/src/java/org/apache/lucene/util/packed/DirectReader.java>。

```
1 1. public long get(long index) {
2 2.     try {
3 3.         // 在encode阶段, 每一个值用2个bit位进行存储, 所以每一个block(一个字节大小)
           中起始位置只会有4种 可能, 即第1位, 第3位, 第5位, 第7位(计数从0开始)
4 4.         // 即shift的值只可能是 0, 2, 4, 6
5 5.         int shift = (3 - (int)(index & 3)) << 1;
6 6.         // 每一个数组元素都用2个bit位表示, 所以跟 0x3执行与操作
7 7.         return (in.readByte(offset + (index >>> 2)) >>> shift) & 0x3;
8 8.     } catch (IOException e) {
9 9.         throw new RuntimeException(e);
10 10.    }
11 11. }
```

例子 (随机访问)

编码后的值如下图所示:

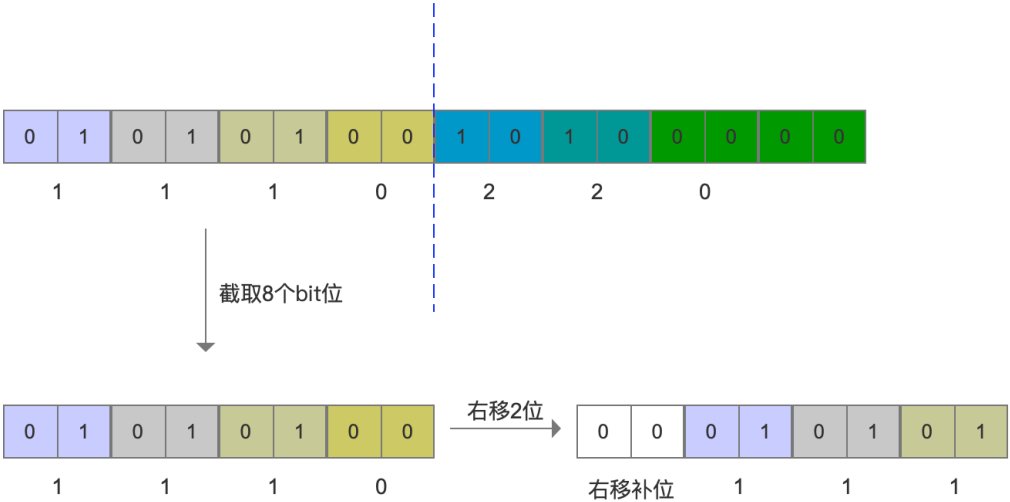
图1:



取出原数组中下标值(index)为2的数组元素

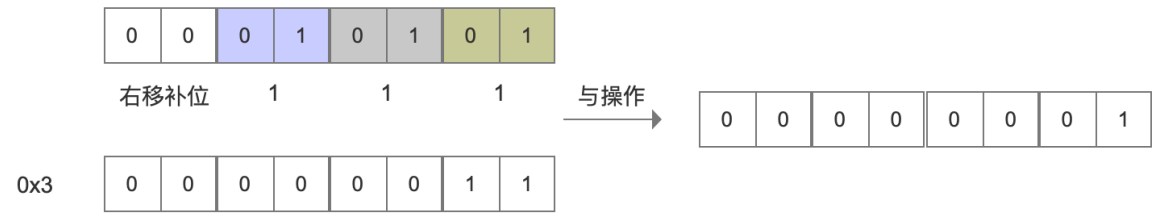
根据decode中第5行代码, 我们先求出shift的值为 2。截取上图中部分字节数据, 大小为 index >> 2, 即截取8个bit位, 然后根据shift的值执行无符号右移操作(第7行代码), 如下图:

图2:



接着根据第7行代码与0x3执行与操作，如下图：

图3:



从上面的解码过程可以看出，这种编码方式可以实现随机访问功能。

在Lucene中的应用

在DocValues中，有着广泛的应用，例如在SortDocValue中，用来存放ordMap[]数组的元素值，ordMap[]的概念会在后面介绍SortedDocValuesWriter类时候介绍。

结语

本篇博客介绍了使用BulkOperationPacked类实现对整数(long或者int)进行压缩存储，与去重编码相比，优点在于在解码时性能更高，并且能实现随机访问，在去重编码中，由于使用了差值存储，所以做不到随机访问。缺点在于当数据中出现较大的值时，压缩比就不如去重编码了。

[点击下载](#)Markdown文件