Second PR progress

Description

Our issue(JarWriter.InputStreamEntryWriter consumes too much memory) is about Input Writer efficiency of `JarWriter`. The function is called `InputStreamEntryWriter()`.

```
private static final int BUFFER_SIZE = 32 * 1024;

private interface EntryWriter {

        void write(OutputStream outputStream) throws IOException;

    }

private static class InputStreamEntryWriter implements EntryWriter {

        private final InputStream inputStream;

        InputStreamEntryWriter(InputStream inputStream) {
            this.inputStream = inputStream;
        }

        @Override
        public void write(OutputStream outputStream) throws IOException {
            byte[] buffer = new byte[BUFFER_SIZE];
            int bytesRead;
            while ((bytesRead = this.inputStream.read(buffer)) != -1) {
                outputStream.write(buffer, 0, bytesRead);
            }
            outputStream.flush();
        }

    }
```

> ***Code Explanation:***
>
> This is part of the code about this issue. The asker said that this function takes too much memory. From the code, we could know that we write all the byte from the InputStream. When InputStream is large, this could lead to large memory consumption.

The asker didn't provide any codes or use circumstances, so we could only guess how this happens. Our assumption is that when `JarWriter` call this function and writer a large file to the Jar file, memory use would increase and due to the buffer size, it would takes a lot of time.

This assumption comes from this line. As you may see, we write from the `InputStream`. When we go to the definition of this, we find that `InputStream` is written by `entryWriter`.

```
...
ByteArrayOutputStream output = new ByteArrayOutputStream();
        entryWriter.write(output);
        entry.setComment("UNPACK:" +
unpackHandler.sha1Hash(entry.getName()));
        return new InputStreamEntryWriter(new
ByteArrayInputStream(output.toByteArray()));
...
```

> ***Code Explanation:***
>
> This part of code shows that `InputStreamEntryWriter` comes from `entryWriter.write(output)`. When this `output` becomes large, memory cost would be huge.

```
...
writeEntry(entry, (outputStream) -> {
                BufferedWriter writer = new BufferedWriter(
                        new OutputStreamWriter(outputStream,
StandardCharsets.UTF_8));
                for (String line : lines) {
                    writer.write(line);
                    writer.write("\n");
                }
                writer.flush();
...
```

> ***Code Explanation:***
>
> This part of code shows how the `output` is generated. Basically, the `BufferedWriter` takes all the files and put it into the `output`, and then this `output` is passed to our `InputStream`, and causes large memory consumption.

Solution

Our idea is to cut the file into small files and send them with multi threads, then compile them together. By this idea, we could send large with less memory, because we don't need to put the entire file to the `BufferedWriter`.

1. File Split.

```
public List<String> splitBySize(String fileName, int byteSize) throws
IOException {
        List<String> parts = new ArrayList<String>();
        File file = new File(fileName);
        int count = (int) Math.ceil(file.length() / (double) byteSize);
        int countLen = (count + "").length();
        ThreadPoolExecutor threadPool = new ThreadPoolExecutor(count,
                count * 3, 1, TimeUnit.SECONDS,
                new ArrayBlockingQueue<Runnable>(count * 2));

        for (int i = 0; i < count; i++) {
```

```
            String partFileName = file.getName() + "."
                    + leftPad((i + 1) + "", countLen, '0') + ".part";
            threadPool.execute(new SplitRunnable(byteSize, i *
byteSize,
                    partFileName, file));
            parts.add(partFileName);

        }
        return parts;

    }
```

Suppose we are sending a large file, we cut it to pieces, and each piece is smaller than 10 Mb. A main problem is that we need to be able to recover the larger file. So we need to set part number. After we get the start point and the file size, we could get the byte array of that part and send it to `FilePart`, and then execute the `inputstream` in multi threads.

2. Multi-thread

```
private class SplitRunnable implements Runnable {
        int byteSize;
        String partFileName;
        File originFile;
        int startPos;

        public SplitRunnable(int byteSize, int startPos, String
partFileName, File originFile) {
            this.startPos = startPos;
            this.byteSize = byteSize;
            this.partFileName = partFileName;
            this.originFile = originFile;
        }

        public void run() {
            RandomAccessFile rFile;
            OutputStream os;
            try {
                rFile = new RandomAccessFile(originFile, "r");
                byte[] b = new byte[byteSize];
                rFile.seek(startPos);
                int s = rFile.read(b);
                os = new FileOutputStream(partFileName);
                os.write(b, 0, s);
                os.flush();
                os.close();

            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

In this part, we use a class called `RandomAccessFile`. A main feature is that it supports reading and writing to a random access file. So we could combine the file with small memory consumption.

In thread pool, we send parts of files based on start point and file size. With this method, in each thread we write little file to the buffer reader and could save some time.

3. Combine

When we need to extract our files from jar, we could combine the data by the `combine()` function. We also need `RandomAccessFile` to visit the end of input stream in a short time.

```
public void combine(String dirPath, String partFileSuffix, int partFileSize, String mergeFileName) throws IOException {
        ArrayList<File> partFiles = FileUtil.getDirFiles(dirPath,
                partFileSuffix);
        Collections.sort(partFiles, new FileComparator());

        RandomAccessFile randomAccessFile = new RandomAccessFile(mergeFileName, "rw");
        randomAccessFile.setLength(partFileSize * (partFiles.size() - 1)
                + partFiles.get(partFiles.size() - 1).length());
        randomAccessFile.close();

        ThreadPoolExecutor threadPool = new ThreadPoolExecutor(
                partFiles.size(), partFiles.size() * 3, 1, TimeUnit.SECONDS,
                new ArrayBlockingQueue<Runnable>(partFiles.size() * 2));

        for (int i = 0; i < partFiles.size(); i++) {
            threadPool.execute(new MergeRunnable(i * partFileSize,
                    mergeFileName, partFiles.get(i)));
        }
    }
```

```
private class MergeRunnable implements Runnable {
        long startPos;
        String mergeFileName;
        File partFile;

        public MergeRunnable(long startPos, String mergeFileName, File partFile) {
            this.startPos = startPos;
            this.mergeFileName = mergeFileName;
            this.partFile = partFile;

        }

        public void run() {
            RandomAccessFile rFile;
            try {
                rFile = new RandomAccessFile(mergeFileName, "rw");
                rFile.seek(startPos);
```

```
                FileInputStream fs = new FileInputStream(partFile);
                byte[] b = new byte[fs.available()];
                fs.read(b);
                fs.close();
                rFile.write(b);
                rFile.close();

            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

A main problem is that we couldn't test our solution in a real application. I mean we could test if our split files and merge files work, and it is. But The asker did not provide their application and spring-boot didn't provide any application about this functionality either. What's more, this issue has been added to the milestone of the spring-boot and perhaps it could be addressed by official team in the future, so we decide to stay tuned for this issue and see if there is any update about this problem.