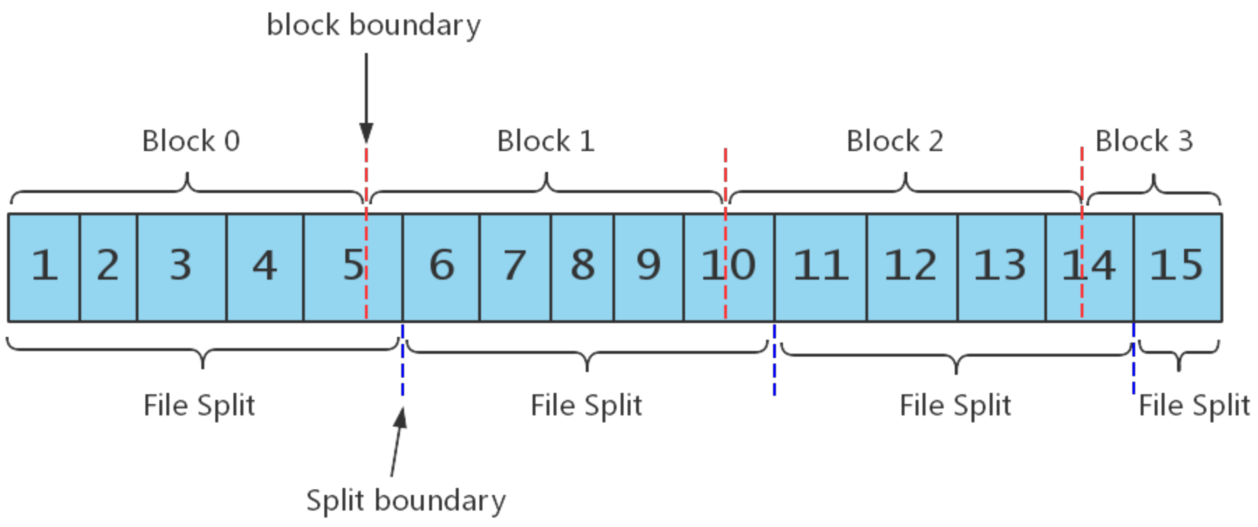
HDFS 块和 Input Splits 的区别与联系(源码版)

在 《HDFS 块和 Input Splits 的区别与联系》 文章中介绍了HDFS 块和 Input Splits 的区别与联系，其中并没有涉及到源码级别的描述。为了补充这部分，这篇文章将列出相关的源码进行说明。看源码可能会比直接看文字容易理解，毕竟代码说明一切。

为了简便起见，这里只描述 TextInputFormat 部分的读取逻辑，关于写 HDFS 块相关的代码请参见网上其他人编写的文章。在阅读下面源码之前，看下下图图中的内容，以便能够深入的理解代码的含义。



在初始化 LineRecordReader 的时候，如果 FileSplit 的起始位置 start 不等于0， 说明这个 Block 块不是第一个 Block，这时候一律丢掉这个 Block 的第一行数据。具体参见下面代码：

```
public void initialize(InputSplit genericSplit,
                       TaskAttemptContext context) throws IOException {
  FileSplit split = (FileSplit) genericSplit;
  Configuration job = context.getConfiguration();
  this.maxLineLength = job.getInt(MAX_LINE_LENGTH, Integer.MAX_VALUE);
  start = split.getStart();
  end = start + split.getLength();
  final Path file = split.getPath();

  // open the file and seek to the start of the split
  final FileSystem fs = file.getFileSystem(job);
  fileIn = fs.open(file);
  CompressionCodec codec = new CompressionCodecFactory(job).getCodec(file);
  if (null!=codec) {
    // 处理压缩文件，我们不考虑
  } else {
    fileIn.seek(start);
    if (null == this.recordDelimiterBytes){
      in = new LineReader(fileIn, job);
    } else {
      in = new LineReader(fileIn, job, this.recordDelimiterBytes);
    }

    filePosition = fileIn;
  }

  // 如果不是第一个 Split，直接丢掉第一行数据的内容。
  // 因为这一行的数据已经被上一个 split 读取了。
  if (start != 0) {
    start += in.readLine(new Text(), 0, maxBytesToConsume(start));
  }
  this.pos = start;
}
```

初始化完 LineRecordReader，程序就可以一行一行地读取 HDFS block
里面的内容了，这个实现主要调用 LineRecordReader 里面 nextKeyValue 的方法：

```
public boolean nextKeyValue() throws IOException {
  if (key == null) {
    key = new LongWritable();
  }
  key.set(pos);
  if (value == null) {
    value = new Text();
  }
  int newSize = 0;
  // 这里每次会多读取一行的数据，注意 getFilePosition() <= end，
  // 这里的 <= ，只有这样，才能保证每次会从下一个 Block 中多读取
  // 一行的数据。
  while (getFilePosition() <= end) {
    newSize = in.readLine(value, maxLineLength,
        Math.max(maxBytesToConsume(pos), maxLineLength));
    pos += newSize;
    if (newSize < maxLineLength) {
      break;
    }

    // line too long. try again
    LOG.info("Skipped line of size " + newSize + " at pos " +
             (pos - newSize));
  }
  if (newSize == 0) {
    key = null;
    value = null;
    return false;
  } else {
    return true;
  }
}
```

readLine 的实现如下：

```
public int readLine(Text str, int maxLineLength,
                    int maxBytesToConsume) throws IOException {
  if (this.recordDelimiterBytes != null) {
    return readCustomLine(str, maxLineLength, maxBytesToConsume);
  } else {
    return readDefaultLine(str, maxLineLength, maxBytesToConsume);
  }
}
```

如果用户指定了 textinputformat.record.delimiter
参数，说明用户自定义数据行分隔符，则程序会调用 readCustomLine 函数；否则调用
readDefaultLine，这时候行的分隔符是 ₩n。readCustomLine 和 readDefaultLine
的处理逻辑很类似，这里只列出 readDefaultLine 的实现：

```
private int readDefaultLine(Text str, int maxLineLength, int maxBytesToConsume)
throws IOException {
  /* We're reading data from in, but the head of the stream may be
   * already buffered in buffer, so we have several cases:
   * 1. No newline characters are in the buffer, so we need to copy
   *    everything and read another buffer from the stream.
   * 2. An unambiguously terminated line is in buffer, so we just
   *    copy to str.
```

```
 * 3. Ambiguously terminated line is in buffer, i.e. buffer ends
 *    in CR.  In this case we copy everything up to CR to str, but
 *    we also need to see what follows CR: if it's LF, then we
 *    need consume LF as well, so next call to readLine will read
 *    from after that.
 * We use a flag prevCharCR to signal if previous character was CR
 * and, if it happens to be at the end of the buffer, delay
 * consuming it until we have a chance to look at the char that
 * follows.
 */
str.clear();
int txtLength = 0; //tracks str.getLength(), as an optimization
int newlineLength = 0; //length of terminating newline
boolean prevCharCR = false; //true of prev char was CR
long bytesConsumed = 0;
do {
  int startPosn = bufferPosn; //starting from where we left off the last time
  if (bufferPosn >= bufferLength) {
    startPosn = bufferPosn = 0;
    if (prevCharCR) {
      ++bytesConsumed; //account for CR from previous read
    }
    bufferLength = in.read(buffer);
    if (bufferLength <= 0) {
      break; // EOF
    }
  }
  for (; bufferPosn < bufferLength; ++bufferPosn) { //search for newline
    if (buffer[bufferPosn] == LF) {
      newlineLength = (prevCharCR) ? 2 : 1;
      ++bufferPosn; // at next invocation proceed from following byte
      break;
    }
    if (prevCharCR) { //CR + notLF, we are at notLF
      newlineLength = 1;
      break;
    }
    prevCharCR = (buffer[bufferPosn] == CR);
  }
  int readLength = bufferPosn - startPosn;
  if (prevCharCR && newlineLength == 0) {
    --readLength; //CR at the end of the buffer
  }
  bytesConsumed += readLength;
  int appendLength = readLength - newlineLength;
  if (appendLength > maxLineLength - txtLength) {
      appendLength = maxLineLength - txtLength;
    }
    if (appendLength > 0) {
      str.append(buffer, startPosn, appendLength);
      txtLength += appendLength;
    }
  // 没有找到换行符，这时候需要再调用 in.read(buffer); 再从文件中读取部分数据。
  } while (newlineLength == 0 && bytesConsumed < maxBytesToConsume);


  if (bytesConsumed > (long)Integer.MAX_VALUE) {
    throw new IOException("Too many bytes before newline: " + bytesConsumed);
  }
  return (int)bytesConsumed;
}
```

当外部程序调用这个读取函数的时候，每次会读取大约 64kb 的数据，并存放在 buffer 数组中，缓存区的大小可通过 io.file.buffer.size 参数配置。读取每个 Block 的时候都会从下一个 Block 多读取一行的数据，也就是说 in.read(buffer); 操作会读取两个 block 的数据。