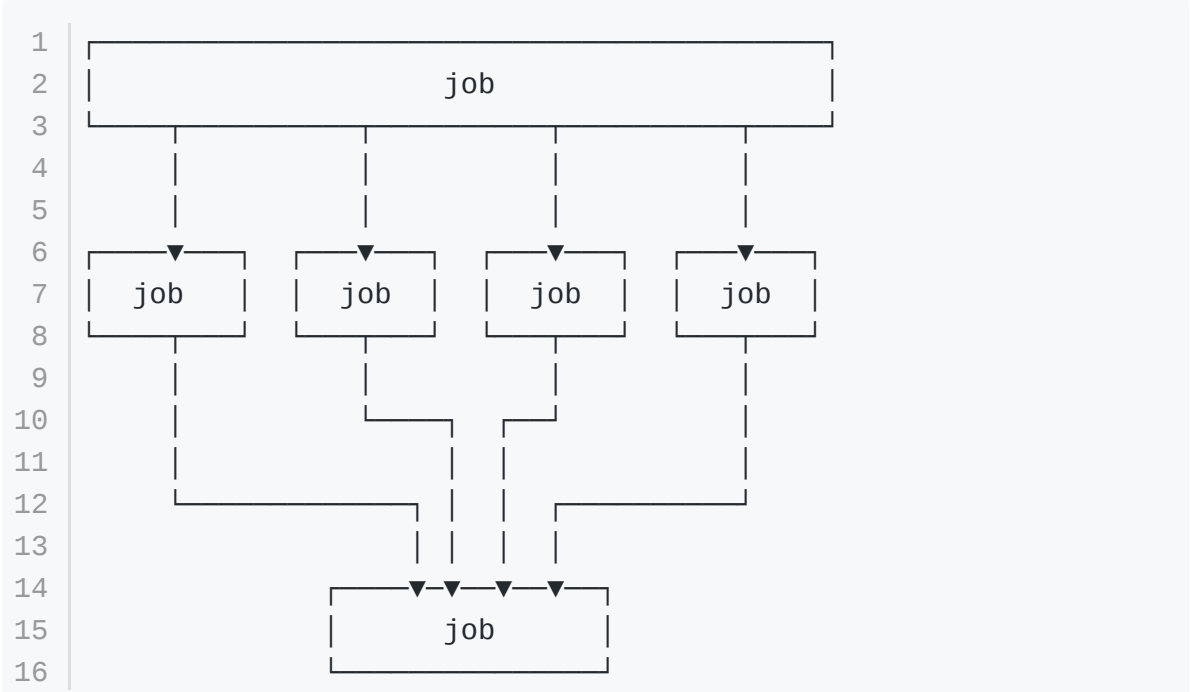
1. MapReduce是什么

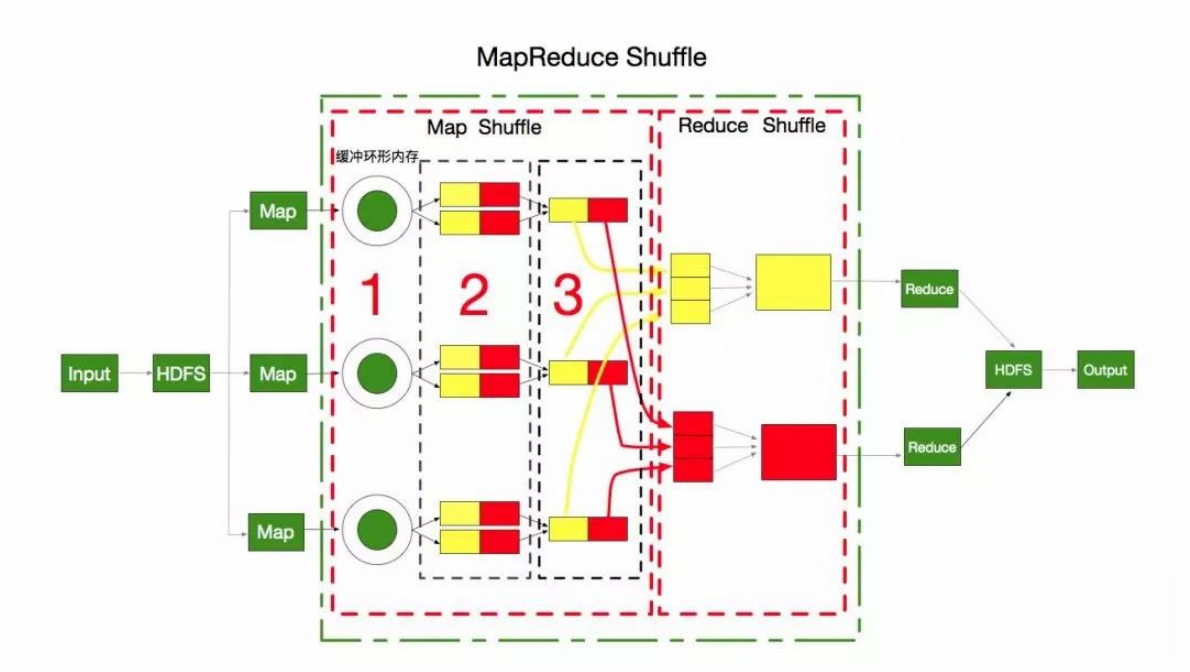
MapReduce是一个分布式计算平台。它与传统的并行计算框架有什么区别呢？传统的并行计算对于编程水平要求很高，程序员不仅要定义任务，更要控制好线程锁等问题，在操作系统课上大家会接触到并行编程的线程锁控制。这就导致它比较适合计算密集型应用。而MapReduce适合数据密集型应用。下面就会说明为什么它适合数据密集型应用。



MapReduce的核心理念是，将并行的计算过程抽象成两个简单的函数，Map和Reduce。Map是将任务切分成一个一个的片，分发给各个节点处理，比如图中输入文件通过hadoop文件系统切割成三个Map，交给三个节点进行处理。

Reduce则是负责将前面Map处理好的分散的任务进行合并处理。其实在Map和Reduce之间还有一个shuffle，用来处理中间文件，加速Reduce过程。

从这里我们也就能看出来，Map高度并行，通常有很多节点，取决于切分的每一片的大小。而Reduce并行能力差，通常只有一个节点。

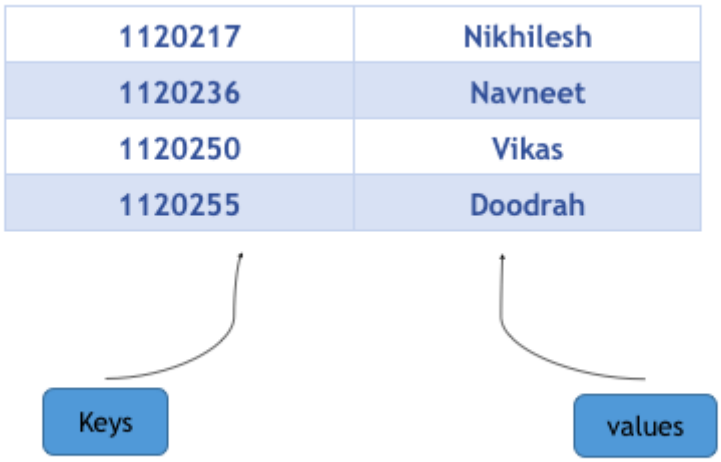


MapReduce的一个开源实现是用java语言的Hadoop。

2. 我们的三种实现

2.1 简单的串行版本

首先讲一下map，一种类似于verilog里面的向量。map提供的是一种键值对容器，里面的数据都是成对出现的。java里的map和C++中的map相似。



在串行版本中，我们使用的是<string, CountForWord>的map，记录字符串和它的出现次数。

```
Map<String, CountForWord> m = new HashMap<String,
CountForWord>();
```

首先是按行读入，然后分析

```
String line;
while ((line = br.readLine()) != null) {
    tokenizeAndSubmit(m, line);
}
```

从行中提取一个个单词，直到当前行为空为止==。如果这个单词已经出现在map中就CountForWord加一，否则就新建一条记录：

```
private static void tokenizeAndSubmit(Map<String,
CountForWord> m, String line) {
    String trimmed = line.trim();
    if (!line.isEmpty()) {
        StringTokenizer tok = new StringTokenizer(line, " ");
        while (tok.hasMoreTokens()) {
            String word = tok.nextToken();
            CountForWord c = m.get(word);
            if (c != null) {
                c.count++;
            } else {
                m.put(word, new CountForWord(word));
            }
        }
    }
}
```

最后利用重载后的比较函数，进行一个序的排：

```
(在CountForWord类中)
    @Override
    public int compareTo(CountForWord t) {
        if (count < t.count) {
            return 1;
        } else if (count > t.count) {
            return -1;
        } else {
            return word.compareTo(t.word);
        }
    }
```

```
ArrayList<CountForWord> lst = new ArrayList<>(m.values());
//sort it
Collections.sort(lst);
```

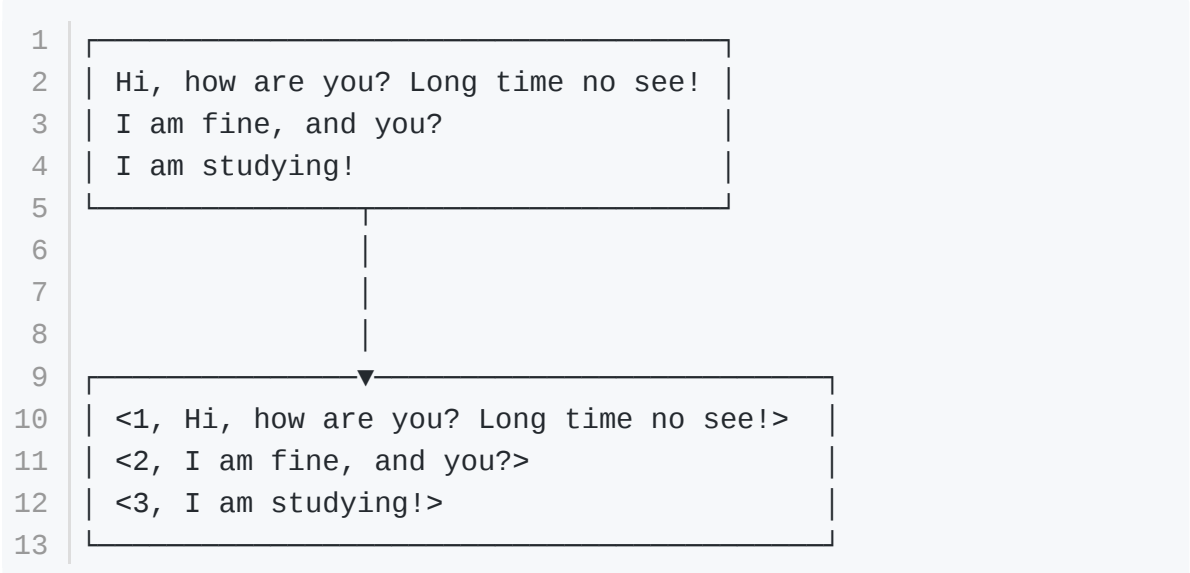
2.2 使用跳表的并行版本

首先我们看main函数：

```
//set the map, reduce class
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(Combiner.class);
job.setReducerClass(IntSumReducer.class);
```

Combiner接下来会解释。

MapReduce会自动把输入文件切分成<行号，行内容>的map，也就是键值对，如下图：



顺便解释一下为什么这个程序会自动切分，上个串行版本的程序我们手动切分。因为上个串行版本其实可以只调用java的标准库来实现，然后就像一个普通的.c文件一样在本地编译运行。但是这样会导致一个问题，我们用hadoop上跑一个程序，和本地跑一个程序进行比较，没有控制变量，所以即使是简单的串行版本，我们也是在hadoop上跑的。既然要在hadoop上跑，我们就要调用hadoop库的一些输入输出函数。不同的是，我们没有调用map和reduce这些函数。所以串行版本没有自动切分，而并行版本我们使用了mapreduce，mapreduce对输入文件进行了自动切分。

---

map 函数，接受<行号，文本>，返回<单词, 1>，这里的1指的是单词出现次数：

```
<1, I think I am tired.>
            |
            ▼
      <I,       1>
      <think,   1>
      <I,       1>
      <am,      1>
      <tired,   1>
```

MapReduce会自动根据键来进行分类，传递给combiner的时候<I, 1> <I, 1>就变成了<I, <1, 1>>这样的键值对。

```
<I, 1> <I, 1> ——————▶ <I, <1, 1>>
```

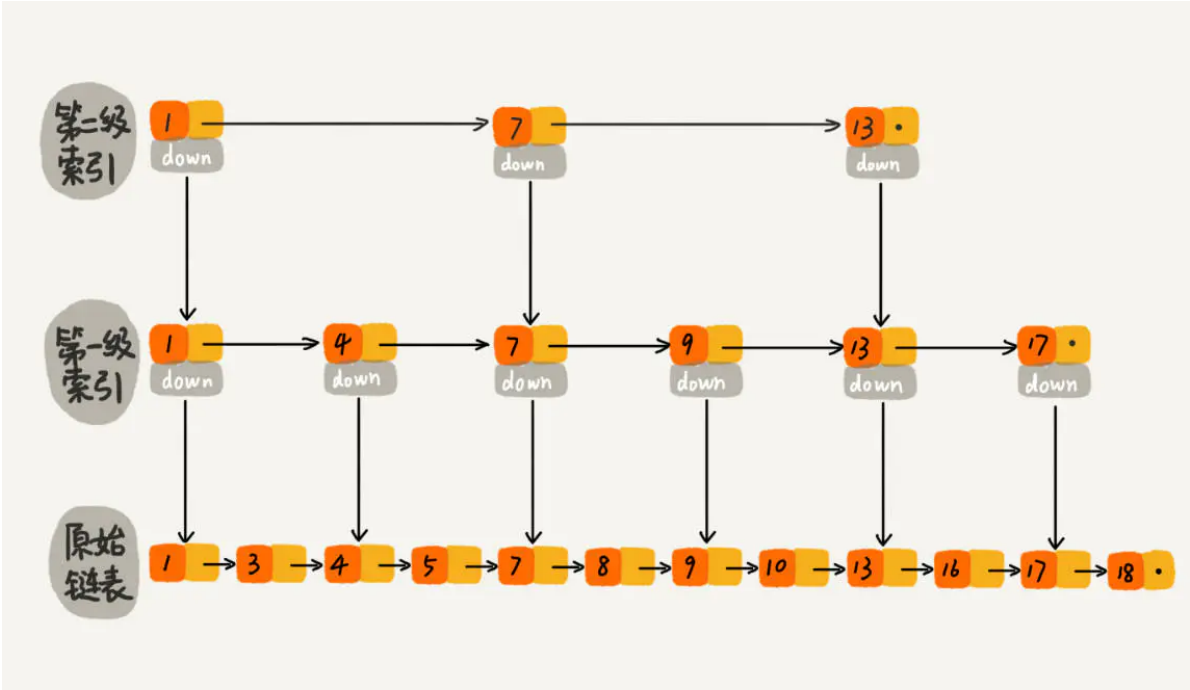
之前我们讲过Map和Reduce是什么，这里再介绍一下Combiner：Combine实质上是本地的reduce，为了节省网络带宽，所以会先在本地进行一次reduce，然后再传到reduce节点上上进行reduce。但是不论combine是否执行，都不应该影响reduce的执行。

为什么说是网络带宽呢？因为虽然hadoop是运行在本地的，但是是通过网络协议走localhost访问

combine函数，将相同的单词的键值对<单词，1>聚合成<单词，N>：

```
<I, <1, 1>> ——————▶ <I, 2>
```

最后通过reduce函数边整合边排序，我们采用java提供的跳表结构，将<单词，频次>的键值对改造成<频次,<单词们>>的键值对，边插入边排序。跳表在上一次展示中同学们已经讲的很清楚了，我们这里不再赘述：



在reduce的cleanup收尾阶段，我们遍历数据结构，就可以按照想要的顺序依次输出<单词，频次>。

2.3 使用两次MapReduce的并行版本

为什么我们会考虑到使用两次MapReduce，这个到测试结果分析阶段再和大家说明。MapReduce会对键值对（<key, value>对）根据key进行排序（上个程序中，我们的key是单词），例如：

```
┌──────────────────┐
|   <sort1, 1   >  |
|   <sort2, 3   >  |     ┌──────────────────────────┐
|   <sort2, 88  >  |     |   <sort1, <1,2>       >  |
|   <sort2, 54  >  ├──►  |   <sort2, <3,54,88>   >  |
|   <sort1, 2   >  |     |   <sort6, <22,58,888> >  |
|   <sort6, 22  >  |     └──────────────────────────┘
|   <sort6, 888 >  |
|   <sort6, 58  >  |
└──────────────────┘
```

在第一次MapReduce中，我们的Map还是做这样一个工作，接受<行号，文本>，返回<单词, 1>。

```
<1, I think I am tired.>
            |
            ▼
      <I,       1>
      <think,   1>
      <I,       1>
      <am,      1>
      <tired,   1>
```

第一次MapReduce的Combine和Reduce工作相同，都是上一个版本的Combine所做的工作，将相同的单词的键值对<单词，1>聚合成<单词，N>：

```
<I, 1> <I, 1> ——————▶ <I, <1, 1>> ——————▶ <I, 2>
```

第二次MapReduce的Map阶段，接受上一次MapReduce的结果，根据<行号，1>构造<<1, string>, string>，自定义排序规则根据组合键<1, string>的FirstKey进行排序

```
<行号，1> ——————▶ <<1, string>, string>
```

第二次MapReduce的Reduce阶段，就直接将排好序的结果整合到一个文件中。

时空复杂度分析：

N代表文件的行数，K代表每行平均的单词数，M代表每次单词平均出现的次数。

- 串行版本
  - 一共有NK个单词，一共有有NK/M个不同的单词，所以map中一共有NK/M个键值对，空间复杂度为O(NK/M)
  - 读入N行时间复杂度是O(N)，每行的处理过程需要进行K次取单词操作，处理时间复杂度为O(NK)。排序要对NK/M项进行排序，时间复杂度约为O(NK/M log(NK/M))。综上，时间复杂度为O(NK)

MapReduce是一个编程框架，而不是算法。我们对其中的自动排序、归并等等实现的复杂度并不清楚。而且具体效率也以来于机器和配置，因此我们主要对试验结果进行分析。

实验结果

首先讲一下我们时间的统计方法：

程序开始时

```
long startTime=System.currentTimeMillis();
```

程序结束时

```
long endTime=System.currentTimeMillis();
System.out.println("程序运行时间：  "+(endTime-startTime)+"ms");
```

为什么用这种时间统计？这种统计出来的是real time，就是实时时间t。hadoop自己会统计时间，但是它统计的时间是CPU time，也就是对CPU来说执行了多长时间。

以例如linux中的time命令为例，time统计出来的三个时间，real, user, sys。user是命令在用户模式执行的时间，sys是命令在系统模式执行的时间，如果CPU是单核，那么real time = user time + sys time。如果是多线程执行，那么real time < user time + sys time。至于小多少就看并行的性能了。

```
qiu@qiu-hp:~$ time tar xjf 下载/anki-2.1.44-linux.tar.bz2 anki-2.1.44-linux/

real    0m18.190s
user    0m18.076s
sys     0m1.113s
```

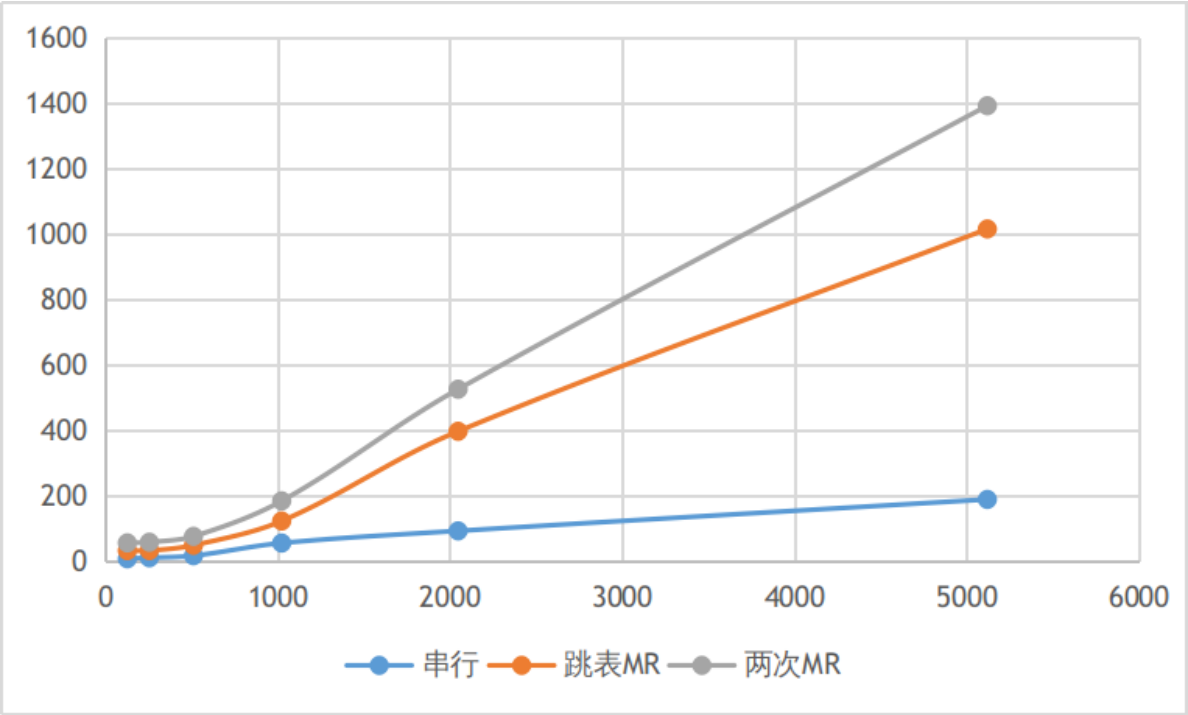
（我电脑是双核，所以这里的数据看上去非常拉胯）

然后讲一下我们测试数据的生成方法，1G一下的数据是用一个python程序随机生成的，1G以上的数据因为太大了会卡死，所以是将1G以下的数据通过重复几次

```
cat test1024.txt >> testxxxx.txt
```

来得到的，因为在记事本里复制粘贴会卡死所以只能用命令行操作了。

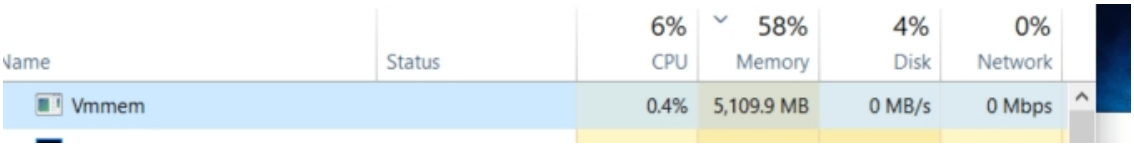
下面就是我们的测试结果：横坐标是测试数据大小，以兆为单位。纵坐标是real time，以秒为单位。



每个数据都是测试两次取平均值。

下面是我们的原因分析：

1. 首先是只有一个真正在跑hadoop的节点，节点数过少，无法完整发挥map reduce的优点。map reduce中的很多map节点都是虚拟出来的，整体算力有限，是伪分布式
2. map reduce本身就会消耗很多计算量，能证明我们这一点的是



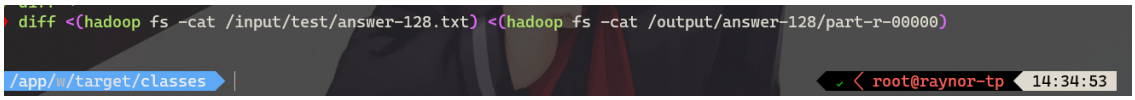
这是什么都没有跑的情况下map reduce占用的内存

3. 并行计算整体CPU时间大于串行，也就是他的工作量只会比串行更大，不会更小，这也是课堂上证明过的
4. hadoop的IO是通过网络进行的，尽管是localhost也会慢很多。IO较多的时候，例如各个节点的信息传来传去，会消耗很多时间，
5. hadoop并没有完整的利用CPU资源，由container去管理其资源使用，尽管设置了上限为8核16G内存，但是在实际运行的时候发现基本维持在2核-4核的CPU使用上，CPU利用率低。而实验观测到串行版本的CPU利用率反而非常高。
6. 数据量相比而言还是太小，hadoop2.x版本map阶段将输入文件分成大小为128M的片，即使是我们最大的5G测试数据，也只是有40个map节点，40个数据的归并对于分治优越性是不足以体现的。
7. 内存大小不够，hadoop实验中内存的占用率经常是99%, 100%，大量时间消耗在了SWAP上。正是因为这个原因，我们考虑要减少内存消耗，于是将跳表算法改成了两次MR算法，但是两次MR算法的IO操作又过多，所以最后的效果也不理想

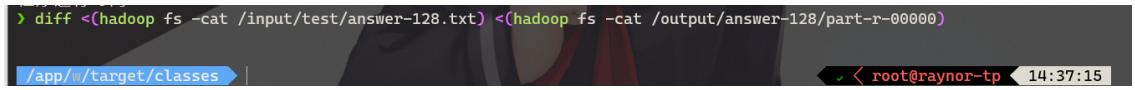
最后是正确性测试：我们通过自己的输出与hadoop自带的wordcount demo的输出用diff指令进行比较。

128MB

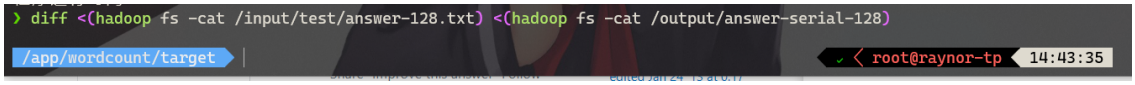
- 跳表+并行MR pass



- 两次MR pass



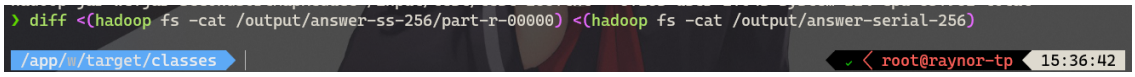
- 串行 pass



256MB

- 串行 diff 跳表并行 pass

- 串行 diff 两次MR pass



正确性测试到此为止

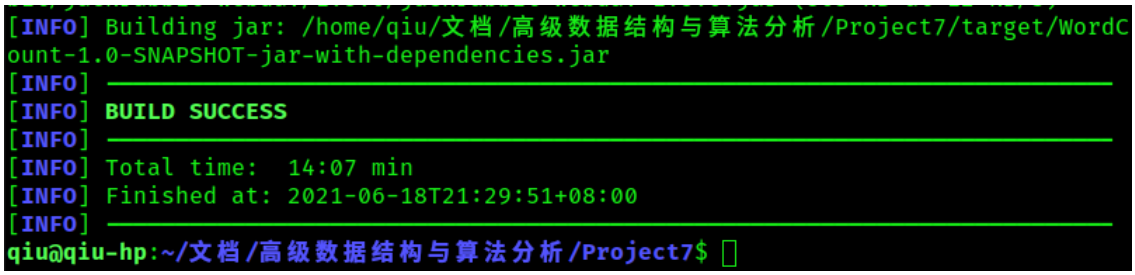
（如果还有多余时间，可以现场演示一下）

现场演示步骤：

1. 配置JAVA环境，演示指令 `java -version`
2. 配置hadoop环境，演示指令：

```
sudo -i
ssh localhost
cd /usr/hadoop/hadoop-2.7.2/sbin
./start-all.sh
jps
```

3. jar打包



```
mvn package
```

4. 现场运行

```
hadoop jar ... 主类
```