

大数据分析技术 课程作业

Wikipedia Retrieval

TF-IDF Calculation under Hadoop

11300240100 徐程程

2013-12-13

目录

0.项目简介	1
1.任务需求与分析	1
2.开发环境搭建与开发工具	2
3.输入数据预处理	2
4.Mapper 和 Reducer 设计.....	5
4-1.Job1.....	5
4-2.Job2.....	6
4-3.Job3.....	6
4-4.Job4.....	6
4-5.Job5, Job6	6
5.Hbase 结构设计	7
6.利用 TF-IDF 结果的维基百科索引页面.....	7
查询 1——给出关键字检索相关页面.....	8
查询 2——给出文章标题，找出关键字.....	10
7.项目结构说明.....	11
8.实验结果与操作说明	11
9.心得与收获	13

0.项目简介

这个项目是大数据分析技术课程的期末作业之任务三。

项目目标：通过实现文档索引这一分布式计算经典应用，增加作者对 map reduce 及 hadoop 分布式集群的了解。

项目工作主要分为输入数据预处理与上传，MapReduce 算法设计与实现，参数调试与改进，Hbase 接口调试，查询页面服务器编写，代码与文档整理以及课堂展示六部分。

全部任务由作者一人完成。

完成时间：2013 年 11 月 24 日-2013 年 12 月 13 日。

1.任务需求与分析

任务需求：

设计 MapReduce 算法在 Hadoop 框架上处理 wikipedia 全部网页的内容，产生词语-文档检索信息并将索引结果信息存入 Hbase，并制作查询页面提供简单的检索服务。

通过简单分析，给出更具体的任务需求。

输入文件为 wikipedia 的 XML dump 压缩文件（解压缩前约 25G，解压缩后

约 44G)。输出为 wikipedia 中每篇文章与每个单词的 TF-IDF 检索信息，并在此基础上做一个检索页面。

其中 TF-IDF 定义与计算公式：

TF-IDF (term frequency-inverse document frequency) 是一种用于[资讯检索](#)与[文本挖掘](#)的常用加权技术。TF-IDF 是一种统计方法，用以评估一字词对于一个文件集或一个[语料库](#)中的其中一份[文件](#)的重要程度。字词的重要性随着它在文件中出现的次数成[正比](#)增加，但同时会随着它在语料库中出现的频率成反比下降。TF-IDF 加权的各种形式常被[搜索引擎](#)应用，作为文件与用户查询之间相关程度的度量或评级。除了 TF-IDF 以外，互联网上的搜寻引擎还会使用基于连结分析的评级方法，以确定文件在搜寻结果中出现的顺序。

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

以上式子中 $n_{i,j}$ 是该词在文件 d_j 中的出现次数，而分母则是在文件 d_j 中所有字词的出現次数之和。

$$idf_i = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$

其中

- $|D|$: 语料库中的文件总数
- $|\{j : t_i \in d_j\}|$: 包含词语 t_i 的文件数目 (即 $n_{i,j} \neq 0$ 的文件数目) 如果该词语不在语料库中，就会导致被除数为零，因此一般情况下使用 $1 + |\{j : t_i \in d_j\}|$

$$tfidf_{i,j} = tf_{i,j} \times idf_i$$

对每个单词和每个页面，我们提供输出对应 TF-IDF 值前 K 大的页面和单词。

2.开发环境搭建与开发工具

由于之前没有使用过 `hadoop`，以及刚开始写项目时，课程实验集群还没有搭建好。作者参考了 `yahoo!` 给出的 `hadoop` 教学手册 <http://developer.yahoo.com/hadoop/tutorial/index.html>。

作者在自己的笔记本的 `vmware` 虚拟机上安装了 `yahoo!` 给出的 `hadoop 0.18` 版本的虚拟机封装，并在自己的 `eclipse` 里安装了配套的 `eclipse` 插件。

经历了千辛万苦，作者阅读了教学手册的全部内容，掌握了 `map reduce` 的基本知识，学会了集群的基本使用，在环境搭建中理解了 `hadoop` 系统的与本机运行的通信关系，熟悉了 `hadoop` 的 API，跑通了简单的 `wordcount` 例子。

接下来，作者开始研究这次任务的算法。

3.输入数据预处理

输入数据是 `wikipedia` 官方给出的 XML dump 文件。`Wikipedia` 每周公开定时发布这个 dump 文件的更新版本，方便数据分析爱好者分析、实验。

XML 文件的结构性极大地方便了我们的预处理步骤。(省去了我们用爬虫获取 html 源数据的过程)但还是相当繁琐。

最原始 25G 的原文件解压后大小为 44G，为一整个 XML 结构文件，格式如下。

```
<mediawiki>
...
<page>
  <title>AccessibleComputing</title>
  <ns>0</ns>
  <id>10</id>
  <redirect title="Computer accessibility" />
  <revision>
    <id>381202555</id>
    <parentid>381200179</parentid>
    <timestamp>2010-08-26T22:38:36Z</timestamp>
    <contributor>
      <username>OlEnglish</username>
      <id>7181920</id>
    </contributor>
    <minor />
    <comment>[[Help:Reverting|Reverted]] edits by
[[Special:Contributions/76.28.186.133|76.28.186.133]] ([[User
talk:76.28.186.133|talk]]) to last version by Gurch</comment>
    <text xml:space="preserve">#REDIRECT [[Computer accessibility]]
{{R from CamelCase}}</text>
    <sha1>lo15ponaybcg2sf49sstw9gdjmdetnk</sha1>
    <model>wikitext</model>
    <format>text/x-wiki</format>
  </revision>
</page>
...
</mediawiki>
```

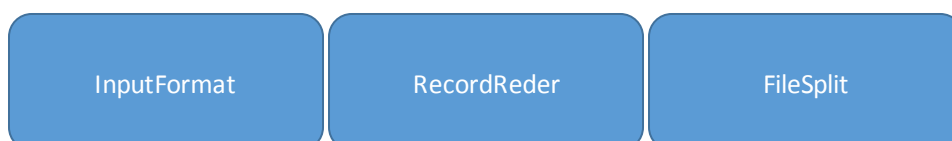
由于体积过大，这个文件用普通的编辑器无法打开，作者通过在 windows 下安装 cygwin 仿 linux 环境，用了里面的 less 功能才看到了文件真容！

观察发现 wikipedia 的每篇文章是包含在<mediawiki></mediawiki>中的一个<page>元素。其中对我们有用的关键元素是<title>和<text>分别是每个 wiki 页面的标题和文字内容。

Wordcount 无法适用，因为我们要建立的关系是每个标题和每个单词的关联度，wordcount 无法记录文章个体的信息。

仔细研究发现在 wordcount 这个例子中，没有特殊声明的类都用的是默认的输入输出格式类。

当 Mapreduce 用文件输入时，它会先调用一个 split 类把文件分成一个一个 split 的，对于每个 split 分配一个 mapper，这个 mapper 会把 split 分成一个一个 record，用一个 recordreader 类从中解析出 mapper 的 key 和 value，这之后再进行处理。



默认情况下，Job 会使用 FileInputSplit，这个类会按行来切文件，每行为一个 record，默认的 TextRecordReader 会把每条 record 拆成<内容，行号>这样的键值

对。

如果我们让 mapper 们直接处理这个 wikipedia.xml 的话，xml 本身的结构就乱掉了。据说 Mahout 类库提供了 xml split，但作者没有找到足够可靠的使用说明，而且这个 parser 理论上也不是很难写。于是作者就没有用 mahout 类库。可是自己写一个的话不仅要解析出<title>,<text>的内容，还要重写 split 使它正好切在<page>的分割处。

作者经过仔细思考，想到比较省力的处理办法。

作者先不用 hadoop，而是用 java 的 stream reader 流处理类库对 wikipedia 做一遍预处理。

这个过程共花费接近两小时。

Total time: 6974.232000 s.
















预处理后的文件变成这个样子：

```
<title>AustriaLanguage</title> [[austrian german]]
<title>AcademicElitism</title> [[academic elitism]]
<title>AxiomOfChoice</title> [[axiom of choice]]
<title>AmericanFootball</title> [[american football]]
<title>America</title> [[america]]
<title>AnnaKournikova</title> [[anna kournikova]]
```

每个 page 保证写在一行上，且标题用<title>标签隔开。这样在 mapreduce 处理的时候，只需要重写 recordreader，把分隔符改成</title>，取出前后标题和内容两部分即可。

这步预处理做完，我得到了 mapper 可以直接处理的 input 文件。

共 2121 个文件，每个文件大小大约在 10~13MB，总共大约 25GB。

 wiki-input2107	10,666 KB	文件
 wiki-input2108	11,528 KB	文件
 wiki-input2109	11,983 KB	文件
 wiki-input2110	12,143 KB	文件
 wiki-input2111	12,216 KB	文件
 wiki-input2112	13,020 KB	文件
 wiki-input2113	12,670 KB	文件
 wiki-input2114	10,845 KB	文件
 wiki-input2115	11,376 KB	文件
 wiki-input2116	11,207 KB	文件
 wiki-input2117	12,013 KB	文件
 wiki-input2118	11,752 KB	文件
 wiki-input2119	11,921 KB	文件
 wiki-input2120	13,075 KB	文件
 wiki-input2121	2,933 KB	文件

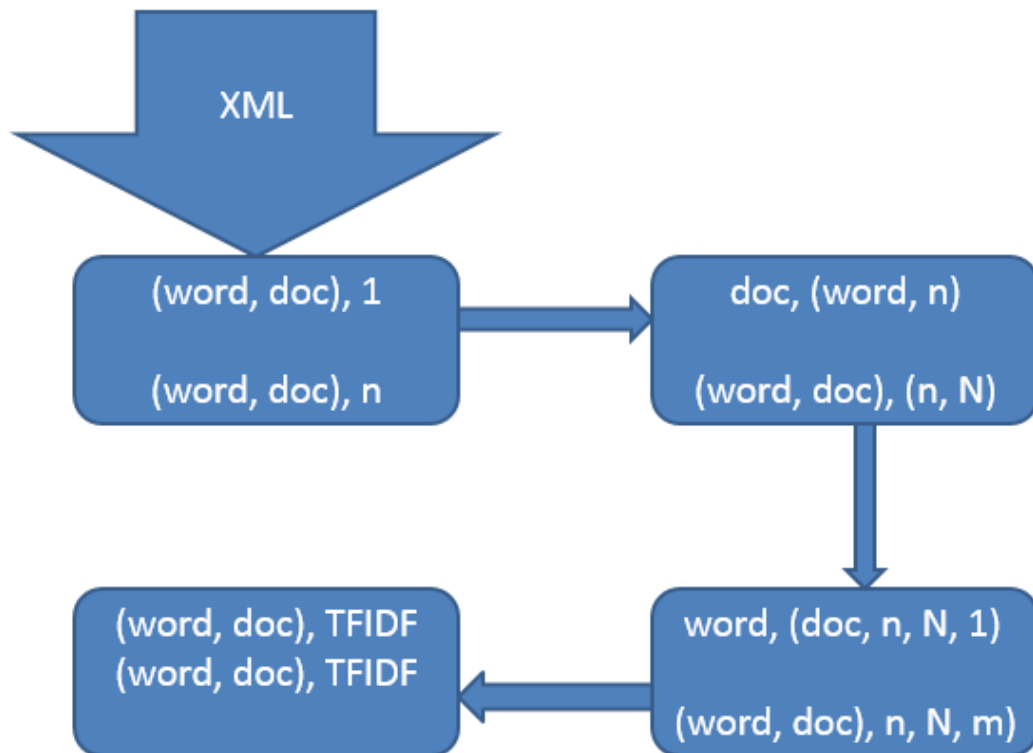
另外在 tfidf 的计算过程中，要用到一个全部文章总数的常数 D，我写了一个

同样是调用 stream reader 的程序，计算出整个 wiki 共 13872614 个页面。

这部分耗时 1516s.(截图打错了)

```
Total number of pages!13872614
Total time for counting: 1516.957000 ms.
```

4.Mapper 和 Reducer 设计



TF-IDF 计算公式已经列在上面了，具体算法十分简单，划分成 4 个 job 后如上图所示，不多解释。下面说明一下各个 job 遇到的问题。

4-1.Job1

Job1 的工作主要是对输入数据的进一步清理和 wordcount。

我在预处理这些文件时，发现<text>元素里，有很多并不是实际网页内容的信息，例如：

```
#REDIRECT
```

```
{{R from CamelCase}}
```

表示重定向，还有许多{}表示链接地址等，于是我用

正则表达式过滤掉了这些脏数据。

接着是一个平凡的 wordcount，作者通过测试多遍，列出了看到的所有奇奇怪怪的字符，并把单词变成了小写。

```
StringTokenizer(line.toLowerCase(), "\\t=@#$%^&*\"\\./1234567890_+| ,()'.!<>:;[]{}");
```

这一步做完后，作者忽然发现少数页面中夹杂着 UTF-8 的火星文字符，理应

在预处理时进行转码。但是前面已经花费许多精力做了预处理和上传（内网上传速度极慢，而且时常断线，作者整整上传了一个周末）于是作者决定放弃这部分火星文，继续做任务。

4-2.Job2

这个 job 要特别注意 mapper reducer 输入时的 parse，要从(doc, word) 中分离出 word，我们要定位**最后一个**逗号的值，因为某些文章的标题会带逗号……

为了方便处理我又在 job1 中加了一步，把文章标题换成中的空格换成下划线。这样做还有一个好处：修改后的文章标题刚好是该页面的 url 后缀！这样我们写 query 的时候就方便了，直接把查询结果加上 wikipedia 的 url 前缀就可以获得目标页面的超链接。

4-3.Job3

Job2 和 Job3 差不多。

4-4.Job4

Job4 计算 TFIDF 结果并输出。这个 job 牵扯到输出问题比较麻烦。

作者首先实现了输出到文件的版本。但输出文件太大，无法支持查询。

为了实现最后的 query 功能，作者需要把输入结果存入 hbase 以供查询。

缺乏经验的作者又花费了千辛万苦搞通了 hbase 的 api 调用和环境配置。作者又用 tomcat 服务器写了 JSP 查询页面，从 hbase 里查询答案。

之后作者找了其中一个 13MB 左右的输入小样 wiki_input_1，依次运行了 job1, job2, job3.（之前都是用自造数据在测试）和 hbase 版的 job4。

作者十分激动地在这里第一次看到了一个检索的结果。

对于以单词为关键字的结果是正确合理的。但以文章标题为关键字查询关联度最高的单词时，出现的结果多为介词和 a, an, the 这种。

作者思考后发现，原来是常数 D 设大了的缘故！跑 13MB 的小样用了全部文件的 D 数字后就会造成 IDF 失去意义。后来作者发现即使在整个 wiki 上跑程序时也不能直接使用

13872514 这个数字，而应该设的小一些，因为 wiki 中很多页面都是重定向到其他页面的。这里作者也增加了 IDF 计算方法设计的巧妙之处。

最终版本的 job4 跑全部数据时，我设的 D 为 100000（这样那些在超过 100000 文章中出现的词的 TF-IDF 值就变成了负数，自动被过滤掉啦！）

4-5.Job5, Job6

原计划只有 4 个 job，将全部<word,doc>对的 tfidf 值算好后，存入 Hbase，每次查询中取出全部 tfidf 值，排序取前 k 大。后来发现这样做的 latency 太高，

按 word 查询大概需要十分钟，按 doc 查询也要几十秒。于是决定新加入两个 job 对查询结果预处理（前 20 大 TFIDF）后存入 Hbase。Job5 和 Job6 是分别以 word 和 doc 为关键字进行排序+插入 hbase 的处理。

5.Hbase 结构设计

RowKey	CF (Column)
[(doc, word)]	[TFIDF value]

RowKey	CF (Column)
(doc, word)	TFIDF value

RowKey	CF (Column)
(Ape, monkey)	0.201840493
(Ape, zoo)	0.051249872
(Apple, cellphone)	0.381234095
(Apple, device)	0.012291286
(Apple, usa)	0.109873412
(Banana, a)	0.000849823

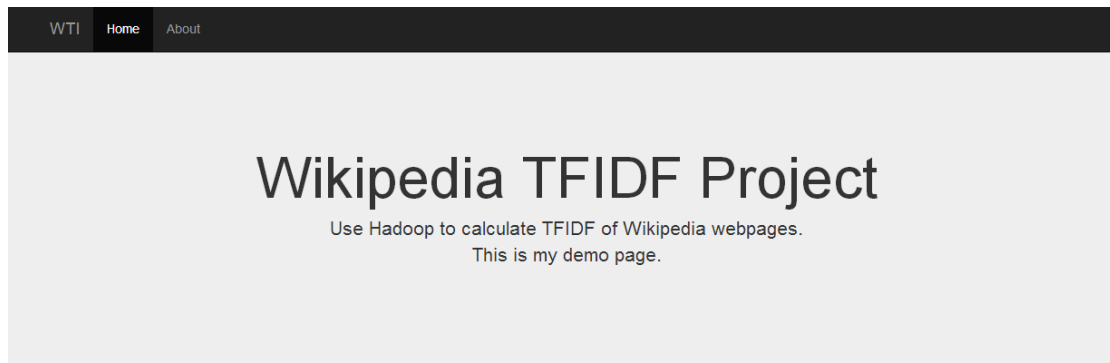
```
table.getScanner(new Scan("Apple", ".getBytes()", "Apple-".getBytes()));
```

如图所示，按照两种 query 需求，建立两张表，查询时利用了 hbase 本身对 RowKey 的字典排序和范围查询功能。

6.利用 TF-IDF 结果的维基百科索引页面

搜索关键字会给出相关度最高的页面链接：

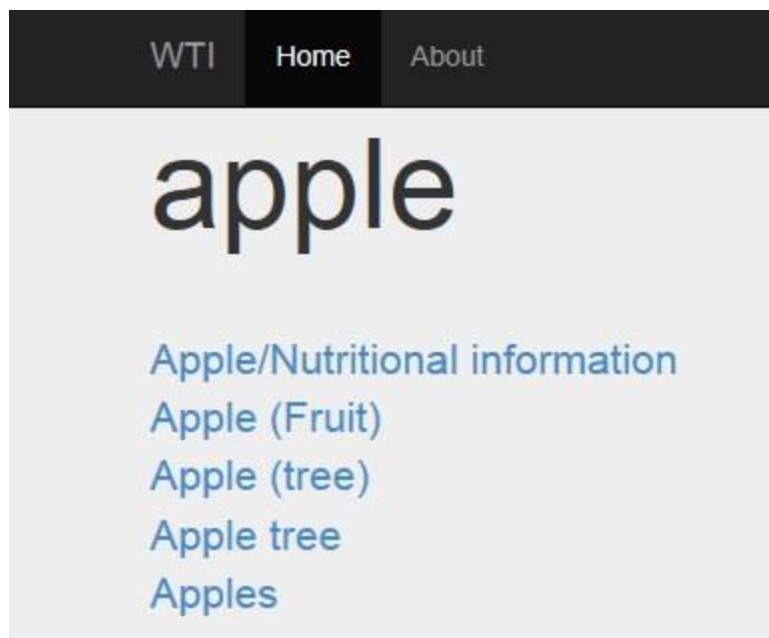
查询 1——给出关键字检索相关页面



Query top-k documents.

Query top-k words.

Query top-k documents.



fudan

[Fudan](#)
[Fudan University](#)
[Shanghai Medical University](#)
[Fudan University](#)
[Zhao Jingshen](#)
[Li Denghui](#)
[Research Science Institute](#)
[Su Buqing](#)
[Tong Dizhou](#)
[Li Lanqing](#)
[Ying Chen \(writer\)](#)
[Second Military Medical University](#)
[Yang Fujia](#)
[Auto-ID Labs](#)

obama

[Barak Obama](#)
[Obama \(disambiguation\)](#)
[BHO](#)
[Wikipedia:Today's featured article/August 18, 2004](#)
[Obama, Fukui](#)
[Pacific States](#)
[Barack Obama](#)
[Toot](#)
[Obama, Nagasaki](#)
[Aino, Nagasaki](#)
[Azuma, Nagasaki](#)
[Minamikushiyama, Nagasaki](#)
[Mizuho, Nagasaki](#)
[Kunimi, Nagasaki](#)
[Iwashiro, Fukushima](#)

查询 2——给出文章标题，找出关键字

Shanghai

_China,shanghai

shanghai

huangpu

shanghainese

pudong

china

yangtze

q???

songjiang

hongqiao

chinese

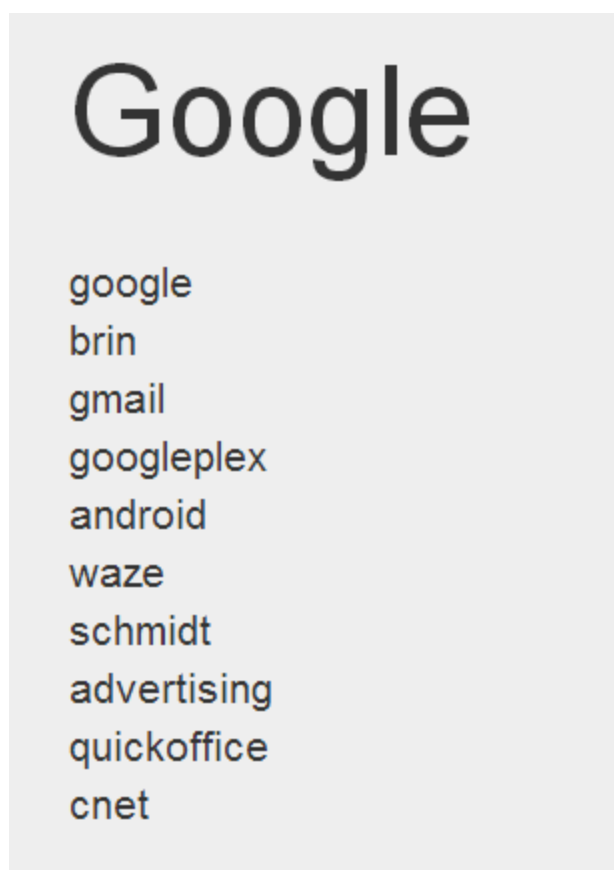
danielson

shikumen

bund

jiangsu

nanjing



7.项目结构说明

见附件。附件中包含预处理、统计、MR 和查询服务器的源代码及 MR 部分的 java 字节码。另外还有 presentation 的文件。

8.实验结果与操作说明

作者完成了预期的任务需求，实现了对这个巨大的 wikipedia 的检索。通过作者多次的补充和改进，最终的检索结果合理有效。但 latency 有点高，单词查询在 10 秒以上。

程序	运行时间
预处理-split	6974s
预处理-统计页面个数	1517s
MapReduce-Job1	4h,43min

MapReduce-Job2	1h,26min
MapReduce-Job3	8min
MapReduce-Job4	1h,29min
MapReduce-total	7h,46min

加入 Job5 Job6 后变化如下：

程序	运行时间
预处理-split	6974s
预处理-统计页面个数	1517s
MapReduce-Job1	4h,43min
MapReduce-Job2	1h,26min
MapReduce-Job3	8min
MapReduce-Job4	7min13sec
MapReduce-Job5 (word 为键)	11min59sec
MapReduce-Job6 (doc 为键)	10min19sec
MapReduce-total	6h,46min
查询 latency	2 秒之内

在跑大数据之前，我跑的 13MB 小样数据 4 个 job 用时约为 7min。
 $7 \times 2121 = 14846\text{min}$ ，而直接处理大数据仅花费了 $7\text{h}, 46\text{min} = 464\text{min}$ ，可见其中
 mapreduce 分布式计算对效率的贡献！

若想重现实验，可找作者索要预处理后的输入数据，上传至 HDFS 后，运行一次
 运行 MR jar 包中的 mapreduce job, WordCount1, WordCount2, WordCount3,
 WordCount4NoHbase, WordCount5, WordCount6.，或者运行 wikiretrieval 的组合
 job，但组合 job 时间过长，切从 log 中只能看到最后一个 job 的历史信息，所以
 不建议。

9.心得与收获

本次实验花费了作者很长时间，估算工作时间超过 60 小时。所谓纸上得来终觉浅，绝知此事要躬行。作者除了收获了实际经验，增加了对许多知识的理解之外，还有许许多多心得和收获，长话短说如下：

1. 配环境时要耐心，心里要有预期的目标，遇到困难一方面要保持毅力，攻克问题，另一方面也要及时转化问题，灵活变通
2. 实现算法之前要考虑清楚，并提前预估尽可能多的细节处理
3. 遇到问题时要先理解出错程序的运行原理
4. 真正实践了从原始数据预处理到输出结果查询的大数据处理的完整流程，增长了自信心，增长了驾驭一个大型数据的能力
5. 心里时常保持任务全局进度状况，这样才不会被此要的细节堵住，按时完成任务！