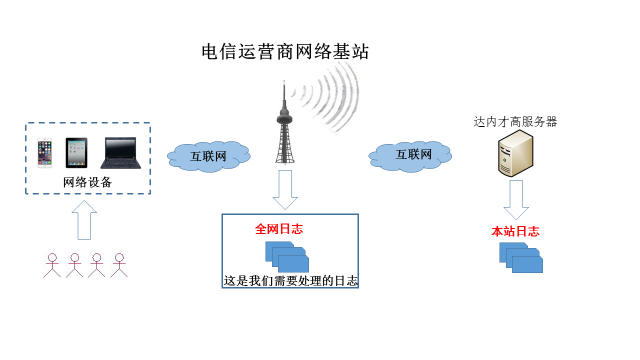
# zebra项目介绍与分析

## zebra概述

### zebra是什么

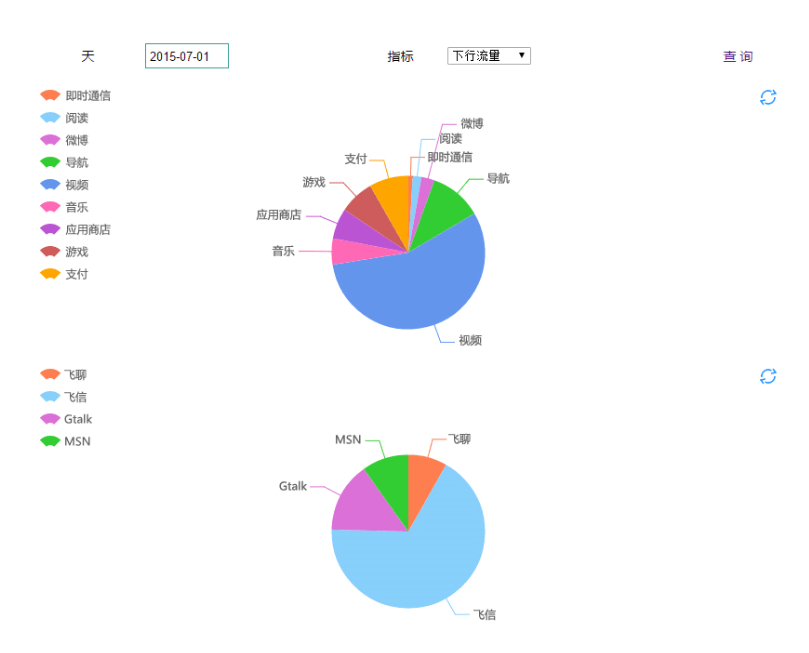


如图所示，电信运营商的用户通过连接到互联网中的各种网络设备访问一个网站时，其访问信息会通过基站在网络中传递，基站可以收集所有用户的访问日志数据

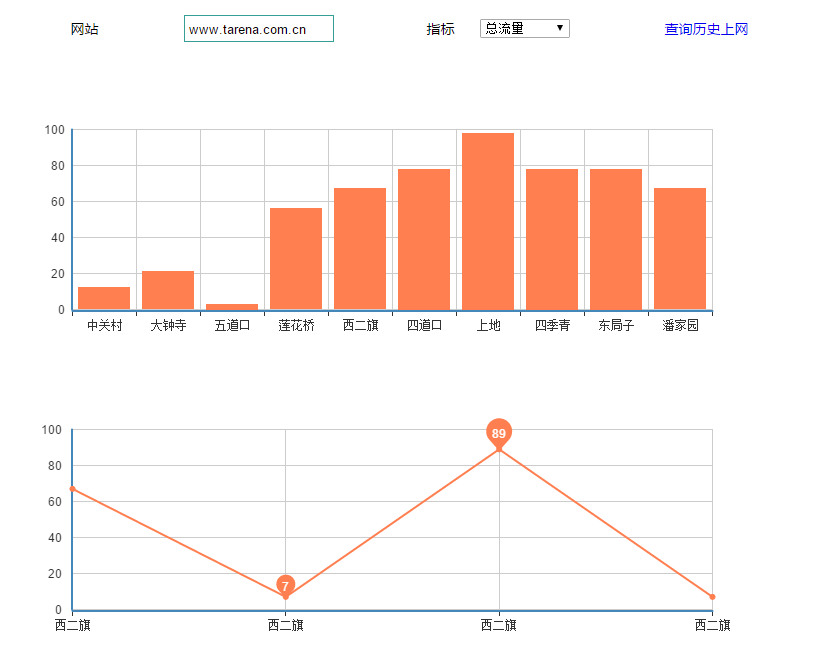
zebra是对电信运营商收集的用户上网数据进行分析的一个应用程序。通过分析得到的结果可以展现不同小区的上网详情。

**注：zebra本意为斑马，命名类似hadoop的方式，好记并无特殊含义！**

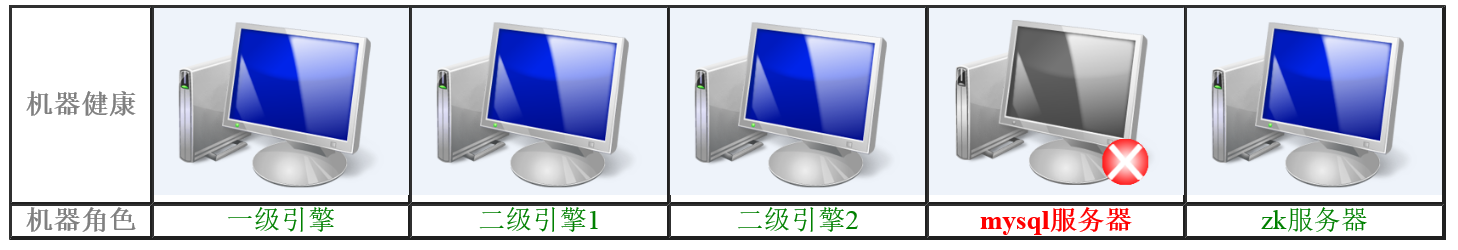
### 整体项目演示



如图所示，按天统计应用大类的某个指标，点击某个大类时能看到这个大类中的各个小类的详情。



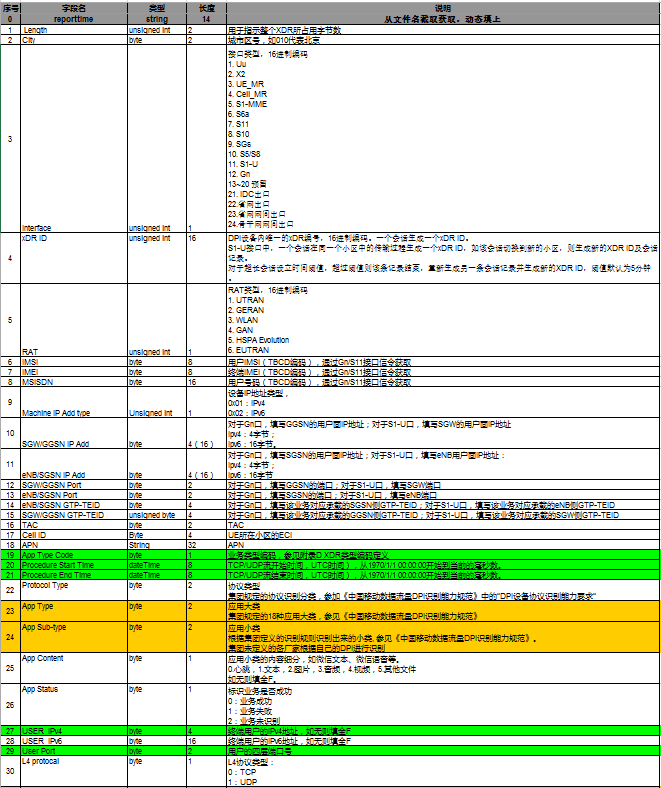
统计多个小区对同一个网站的访问情况，单击某个小区时可以查看某个小区的不同时间段内的详情

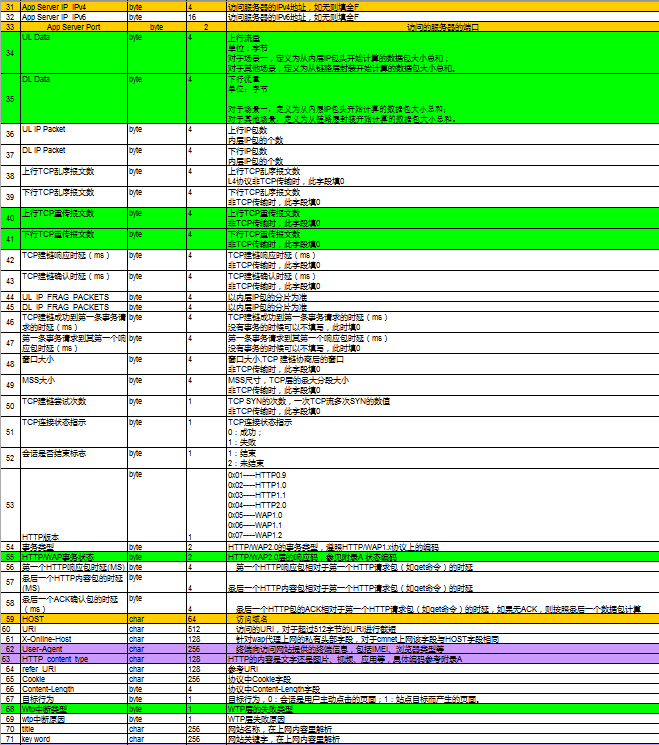


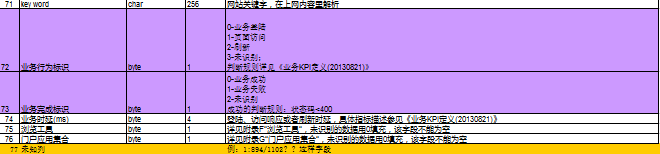
可以通过zookeeper监控整体集群的健康状况。

### zebra项目需求分析

* **来源数据格式**







**如图所示，运营商基站收集的用户上网日志中一行有很多字段，可能是用户某次短信发送，也可能是用户某次的通话，也可能是http访问或者app内通讯（比如：QQ好友之间聊天）等等。因为我们只需要分析所有的http访问，所以我们需要得到不同小区的上网详情只需要处理下列字段就可以了。**

* **需要分析的列**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 16 | TAC | byte | 2 | TAC |
| 17 | Cell ID | Byte | 4 | UE所在小区的ECI |
| 19 | App Type Code | byte | 1 | 业务类型编码，参见附录D XDR类型编码定义 |
| 20 | Procedure Start Time | dateTime | 8 | TCP/UDP流开始时间，UTC时间），从1970/1/1 00:00:00开始到当前的毫秒数。 |
| 21 | Procedure End Time | dateTime | 8 | TCP/UDP流结束时间，UTC时间），从1970/1/1 00:00:00开始到当前的毫秒数。 |
| 23 | App Type | byte | 2 | 应用大类 集团规定的18种应用大类，参见《中国移动数据流量DPI识别能力规范》 |
| 24 | App Sub-type | byte | 2 | 应用小类 根据集团定义的识别规则识别出来的小类, 参见《中国移动数据流量DPI识别能力规范》。 集团未定义的各厂家根据自己的DPI进行识别 |
| 27 | USER\_IPv4 | byte | 4 | 终端用户的IPv4地址，如无则填全F |
| 29 | User Port | byte | 2 | 用户的四层端口号 |
| 31 | App Server IP\_IPv4 | byte | 4 | 访问服务器的IPv4地址，如无则填全F |
| 33 | App Server Port | byte | 2 | 访问的服务器的端口 |
| 34 | UL Data | byte | 4 | 上行流量 单位：字节 对于场景一，定义为从内层IP包头开始计算的数据包大小总和； 对于其他场景，定义为从链路层封装开始计算的数据包大小总和。 |
| 35 | DL Data | byte | 4 | 下行流量 单位：字节  对于场景一，定义为从内层IP包头开始计算的数据包大小总和； 对于其他场景，定义为从链路层封装开始计算的数据包大小总和。 |
| 40 | 上行TCP重传报文数 | byte | 4 | 上行TCP重传报文数 非TCP传输时，此字段填0 |
| 41 | 下行TCP重传报文数 | byte | 4 | 下行TCP重传报文数 非TCP传输时，此字段填0 |
| 55 | HTTP/WAP事务状态 | byte | 2 | HTTP/WAP2.0层的响应码，参见附录A 状态编码 |
| 59 | HOST | char | 64 | 访问域名 |
| 62 | User-Agent | char | 256 | 终端向访问网站提供的终端信息，包括IMEI、浏览器类型等 |
| 63 | HTTP\_content\_type | char | 128 | HTTP的内容是文字还是图片、视频、应用等，具体编码参考附录A |
| 68 | Wtp中断类型 | byte | 1 | WTP层的失败类型 |
| 72 | 业务行为标识 | byte | 1 | 0-业务登陆 1-页面访问 2-刷新 3-未识别； 判断规则详见《业务KPI定义(20130821)》 |
| 73 | 业务完成标识 | byte | 1 | 0-业务成功 1-业务失败 2-未识别 成功的判断规则：状态码<400 |

**这是我们需要处理的每行数据中的字段，通过对这些数据的处理，我们可以得到不同小区的上网详情数据。具体来说，就是把一段时间内的同一个小区内访问同一网站、同一个ip的访问累计起来，就可以得到某小区内的某网站的访问详情。**

* **业务字典**

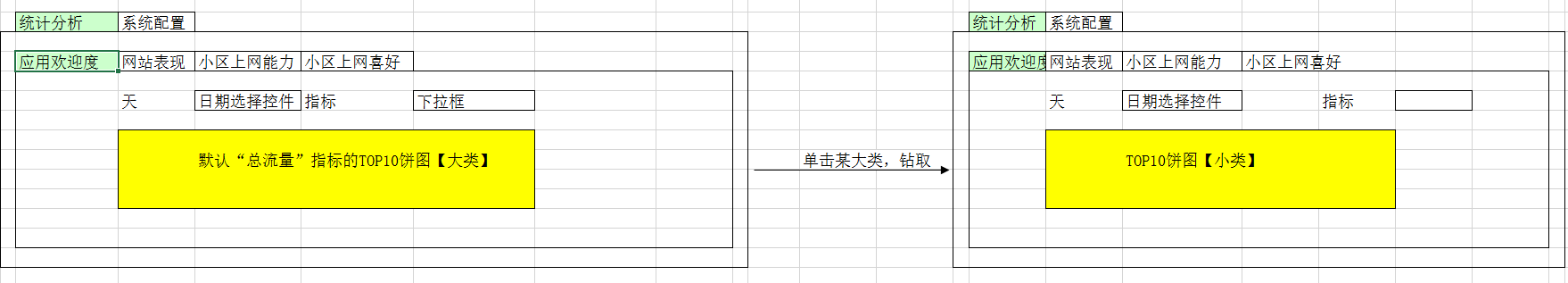
**数据流量业务大类分类**

|  |  |  |
| --- | --- | --- |
| **序号** | **业务类型** | **业务说明** |
| 1 | 即时通信 | 互联网消息即时收发业务,如：QQ、飞信等 |
| 2 | 阅读 | 向用户提供在线或离线阅读服务的业务，如：移动手机阅读、熊猫阅读等 |
| 3 | 微博 | 微博业务，如：移动微博、新浪微博等 |
| 4 | 导航 | 提供浏览、查询、导航等功能的电子地图类业务，如：谷歌地图、高德导航等 |
| 5 | 视频 | 向用户提供音视频内容的直播、分享和下载服务的网站和应用（不包括传统意义上基于P2P技术的视频业务），如：优酷、手机电视等 |
| 6 | 音乐 | 提供音乐在线欣赏和下载服务的网站和应用，如：咪咕音乐、QQ音乐等 |
| 7 | 应用商店 | 提供应用程序、音乐、图书等内容浏览、下载及购买服务的业务，如：Mobile Market、AppStore等 |
| 8 | 游戏 | 基于客户端或者网页的游戏业务：QQ游戏、开心农场等 |
| 9 | 支付 | 电子商务类业务，如：手机支付、支付宝、网银等 |
| 10 | 动漫 | 提供动漫在线欣赏和下载服务的网站和应用，如：手机动漫、爱看动漫等 |
| 11 | 邮箱 | 电子邮箱业务，如：139邮箱、QQ邮箱等 |
| 12 | P2P业务 | 基于P2P技术的资源共享业务，包括下载和视频两部分，前者如：迅雷、eMule等，后者如：迅雷看看、PPLive等 |
| 13 | VoIP业务 | 互联网语音通信业务，如：Skype、Uucall等 |
| 14 | 彩信 | 彩信业务 |
| 15 | 浏览下载 | 基于HTTP、WAP、FTP等的普通浏览和下载业务 |
| 16 | 财经 | 金融资讯、股票证劵类业务，如：手机商界、大智慧等 |
| 17 | 安全杀毒 | 提供网络安全服务的应用，如：360安全卫士、麦咖啡等；以及网络恶意流量，如：病毒、攻击等 |
| 18 | 其他业务 |  |

**DPI设备子业务识别能力要求（部分）**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 业务类型 | 子业务 | | | |
| 序号 | 子业务名称 | 优先级 | 备注 |
| 即时通信 | 1 | 飞聊 | 必选 | 自有业务 |
| 2 | 飞信 | 必选 |
| 3 | Gtalk | 必选 | 互联网业务 |
| 4 | MSN | 必选 |
| 5 | QQ | 必选 |
| 6 | TM | 必选 |
| 7 | 阿里旺旺 | 必选 |
| 8 | 米聊 | 必选 |
| 9 | 微信 | 必选 |
| 10 | 人人桌面 | 必选 |
| 11 | AOL AIM | 可选 |
| 12 | Gadu\_Gadu | 可选 |
| 13 | go聊 | 可选 |
| 14 | ICQ | 可选 |
| 15 | IMVU | 可选 |
| 16 | Lava-Lava | 可选 |
| 17 | NetChat | 可选 |
| 18 | Paltalk | 可选 |
| 19 | PowWow | 可选 |
| 20 | TeamSpeak | 可选 |
| 21 | Trillian | 可选 |
| 22 | VZOchat | 可选 |
| 23 | Xfire | 可选 |
| 24 | 百度Hi | 可选 |
| 25 | 都秀 | 可选 |
| 26 | 陌陌 | 可选 |
| 27 | 天翼Live | 可选 |
| 28 | 翼聊 | 可选 |
| 29 | 网易泡泡 | 可选 |
| 30 | 新浪UC | 可选 |
| 31 | 新浪UT | 可选 |
| 32 | 雅虎通 | 可选 |

* **需要展现的结果**

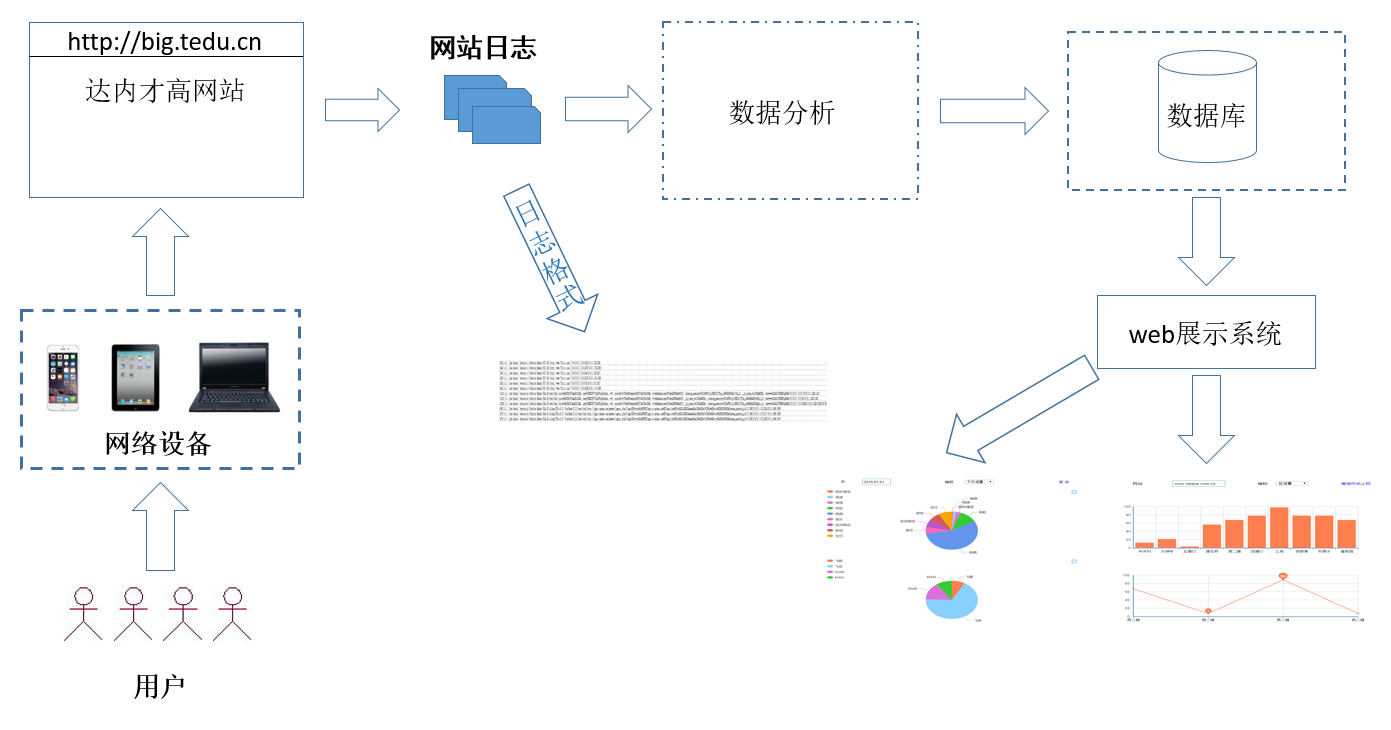


如图所示，需要根据计算，得到不同小区的上网能力，具体包括：应用的欢迎程度、网站表现、小区上网能力和小区上网喜好。通过点击某个大类下的小类的详情。

## zebra整体分析

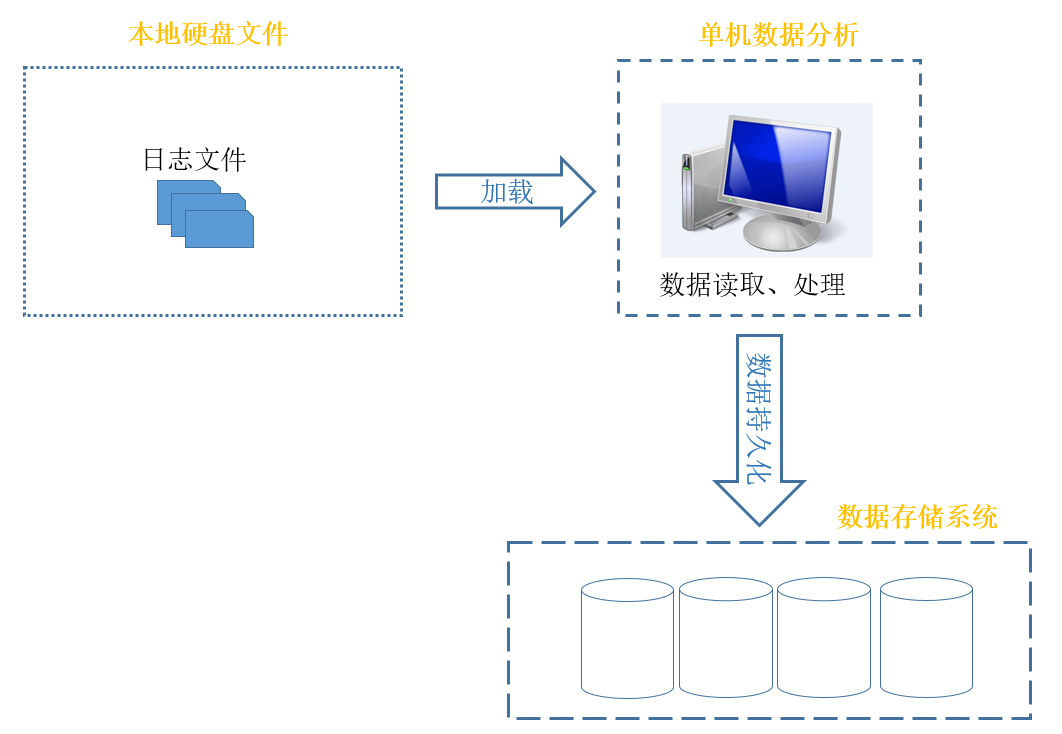
### zebra整体流程

如下图所示，用户通过网络设备访问达内才高网站，通过电信运营商的基站被记录成日志的形式，日志数据由运营商收集，并以单位时间形成一个文件的形式存放到本地硬盘上。通过zebra对数据进行分析处理，将处理后的结果放置到关系型数据库中，通过web图形化的方式展示不同小区的上网详情！



### zebra整体思路设计

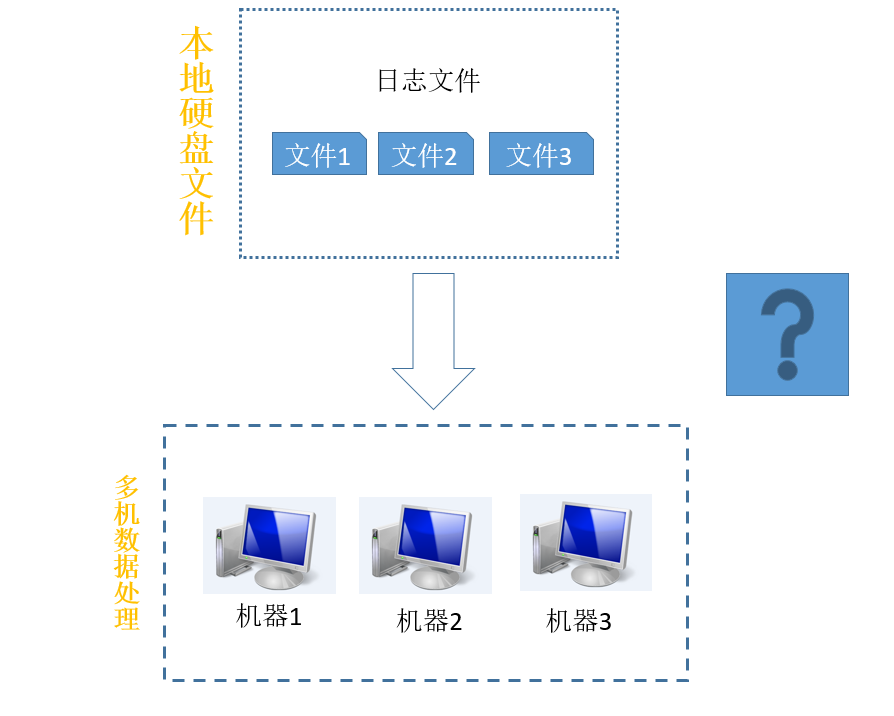
* **单机处理方式**



这种方式会将数据的读取（即从本机的硬盘读取到本机的内存的过程）和运算以及数据持久化都集中在一台机器上。这台机器要完成数据读取、数据运算和数据持久化工作，压力比较大。

* **考虑使用多机的处理方式**

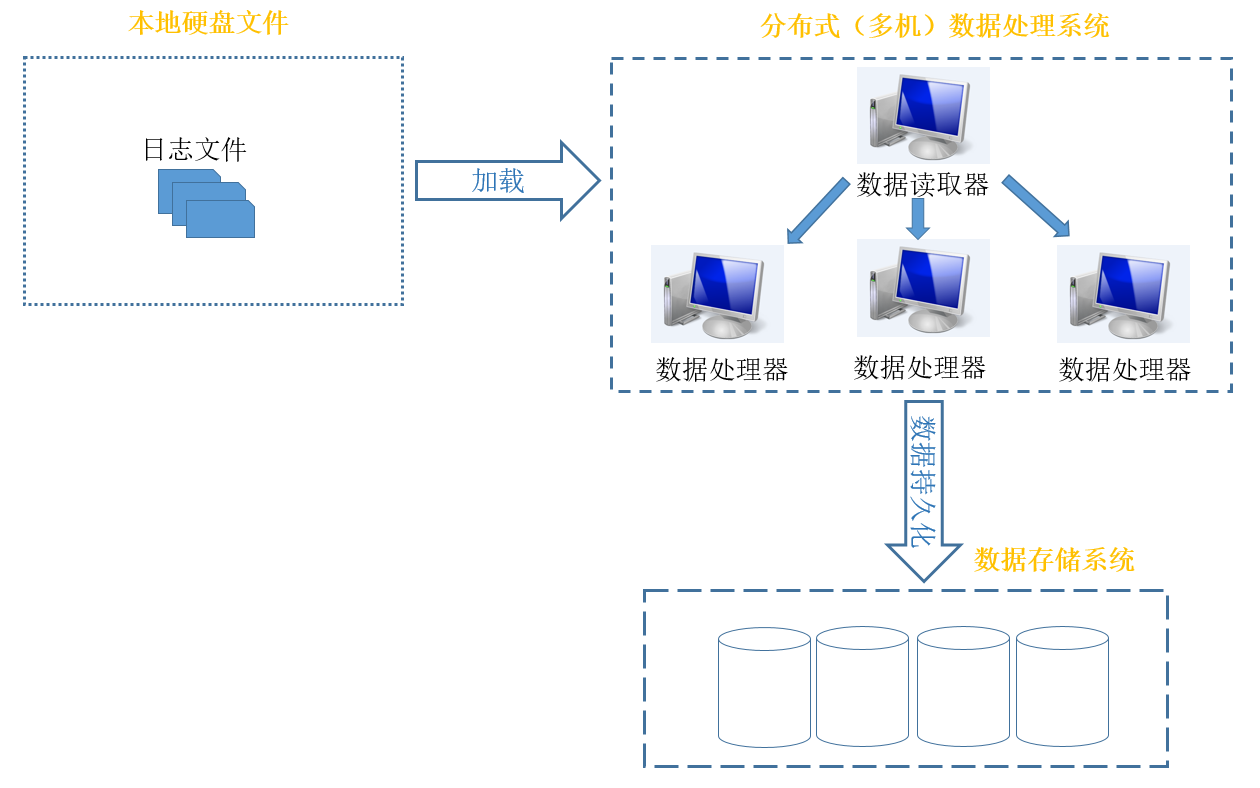
可以考虑利用人多力量大的思想，使用多台机器合作处理多个文件！



问题：多台机器之间如何协调和分工？哪些机器负责数据读取？哪些机器负责数据的运算？

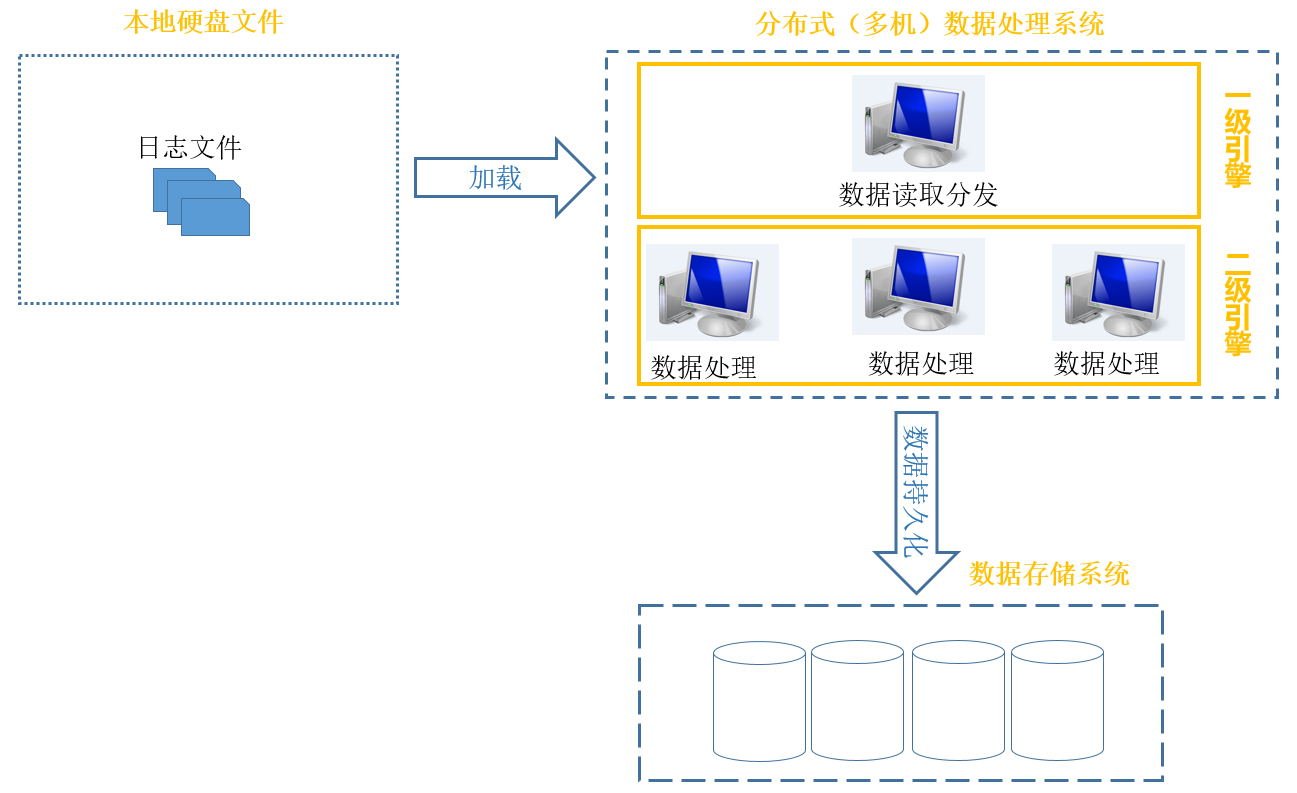
* **多机处理方式**

这种方式会将数据的读取和传输（传输，即从一台机器的内存到其他机器的内存的过程）交给一台机器，这台机器只负责数据的读取和传输，减轻自身压力！再由其他机器负责数据运算和持久化，提高处理能力。



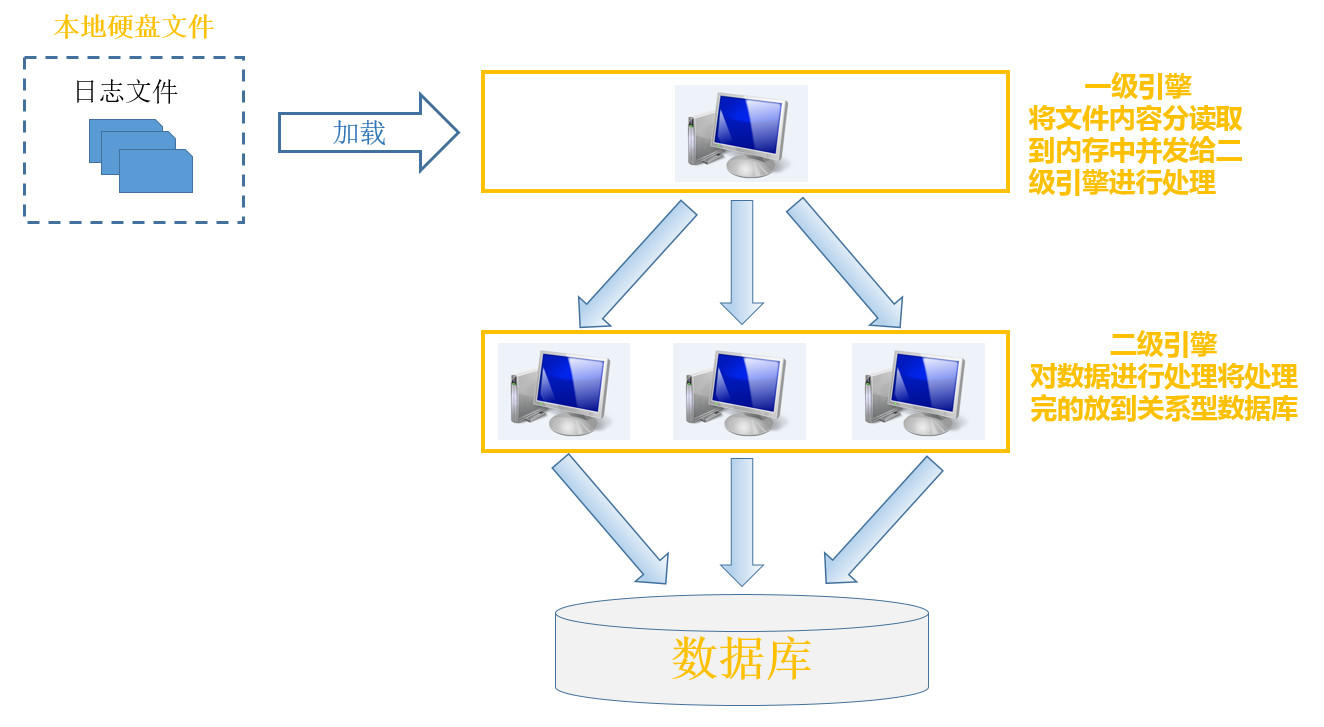
* **最终选取方式**

我们将集群中负责数据读取和分发的机器称之为一级引擎，其他负责运算和数据持久化的机器称之为二级引擎。



* **最终方式角色描述**

我们的方案最终的结构如下：一级引擎负责数据的读取和分发，二级引擎负责数据的计算和持久化。



# zebra项目技术实现

## 技术准备

因为整个项目中运用了java中nio相关的api和大量的java.util.concurrent包（并发包）中的类和接口，所以我们在正式实现整体项目之前先对nio和并发编程进行学习！

### 任务的概念

在程序和生活中都有任务的概念，任务本身是一个宽泛的概念，在生活中，一个任务可以是吃一顿饭也可以是一次旅行。在程序中也是如此，可以是读取一个文件，也可以是获取文件的绝对路径。总结来说，任务可大可小，一个大任务可以是通过一堆小任务堆积组成，即任务是可嵌套的。具体到程序中，一个最小任务就是一个执行单元，所以在不同上下文中所指的任务可能完全不同！所以，我们要在具体的场景下来说明基于这个场景中的任务是什么。

### 线程递减锁

* **场景**

现有一营销公司，其内部共有三个员工和经理、老板，员工有小王、小李、小张。现在项目经理分配了一个任务，公司有一个Excel文件，内含有1500条客户数据，每条数据都是一条包含客户电话号码的客户的资料，员工根据每条数据的资料给客户打电话进行推销，理分配给三个人每人500条，等这个Excel都处理完以后项目经理向老板申请给大家放假。在这个场景下我们就可以使用线程递减锁来控制整个工作的流程，保证所有人的任务都完成后再由项目经理告知老板工作做完了。

* **概念**

线程递减锁是jdk1.5在java.util.corrcuent并发包中提供的一个类CountDownLatch，提供了高级的类似线程的wati和notify机制。

* **示例代码**

详见zebrademo中关于线程递减锁的示例

* **总结**

一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待

### 阻塞式队列

* **场景**

我们有一个数据生产者（运营商基站）和一个数据消费者（处理基站数据的程序），为了使应用程序可以高效的运行，我们需要提供一种等待机制，即在生产者的生产能力较弱时，消费者可以在等到生产者生成数据之后再进行消费，程序不会中断。在这种情境下，生产者和消费者之间要有一个存放数据的容器，这个容器就可以用一个阻塞式队列来充当。阻塞式队列本身就提供了这种等待机制（我们更习惯称之为阻塞机制，都是一个意思）。

* **概念**

阻塞式队列也是jdk1.5在java.util.corrcuent并发包中提供的一中更加方便的添加了阻塞功能的队列。共有两种类型，一种是ArrayBlockingQueue,这种类型的队列在构造时就需要指定队列中能存放的数据的总条数。另一种是LinkedBlockingQueue,这种类型的队列在构造时可以不需要指定队列中的数据的总条数，可以无限扩容（当然需要在jvm可用的内存空间的基础上）。

* **原理**

其实阻塞式队列的本质就是在队列的基础上加上了阻塞机制，可以防止因队列中数据填满造成队列已满异常或队列中没有数据而造成空指针异常。其改进之后表现为，可以在队列中的数据填满后一直等待，直到队列中有空闲位置时才继续插入，或者在队列为空之后一直等待，直到队列中有数据之后再继续取出。

* **示例**

详见zebrademo中关于阻塞式队列的示例

### 线程池执行器

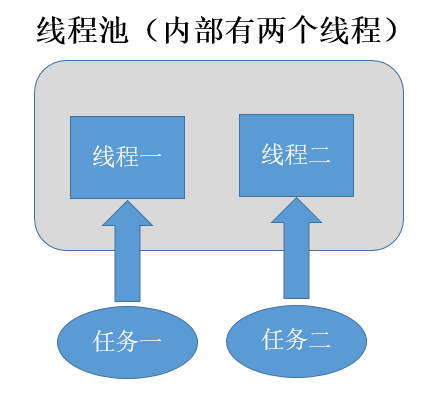
* **场景**

电信运营商（移动、联通、电信、铁通等）的基站手机用户的上网数据，使用多个处理线程来进行处理，需要一个任务管理和分配的一种机制，线程池管理器就可以实现这种机制。

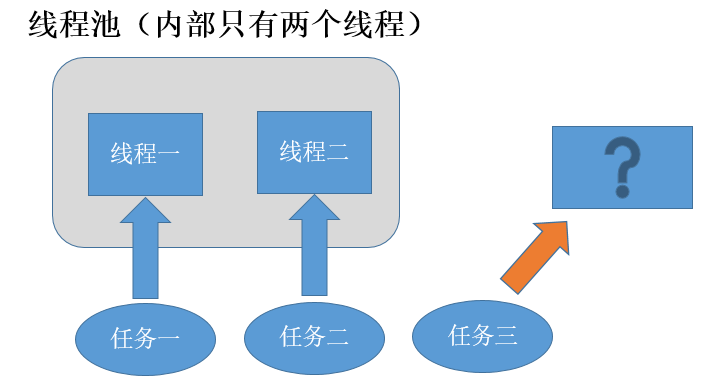
* **概念**

**场景：多线程程序设计**

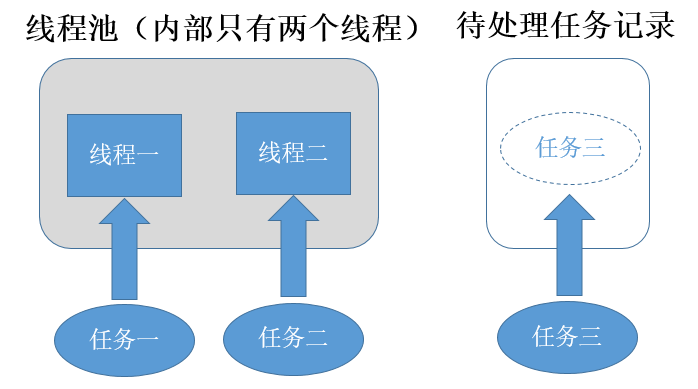
情况一:线程数据和任务数量正对应，这时候我们一个线程管理一个任务即可



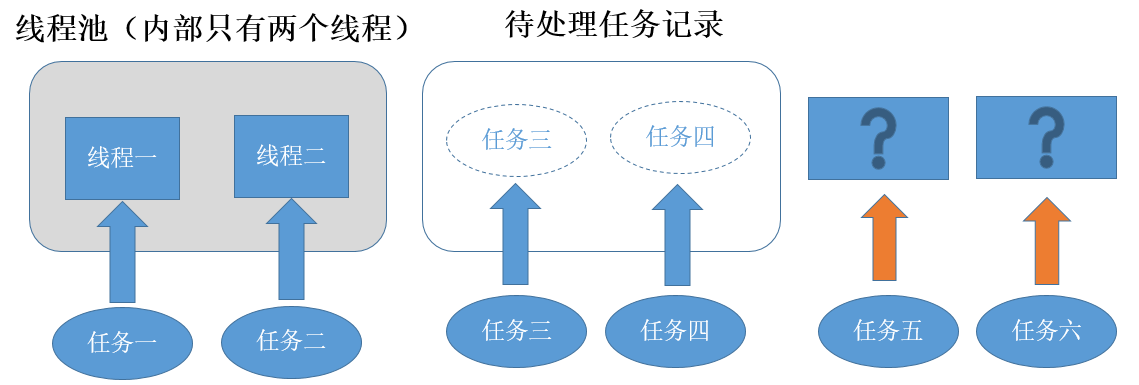
情况二：假如我们的机器的cpu是双核的，我开启两个线程是最高效的，我们想要以最高效的形式进行处理，这时候就会导致一个问题，有一个任务暂时无法处理，这时候我们怎么办？



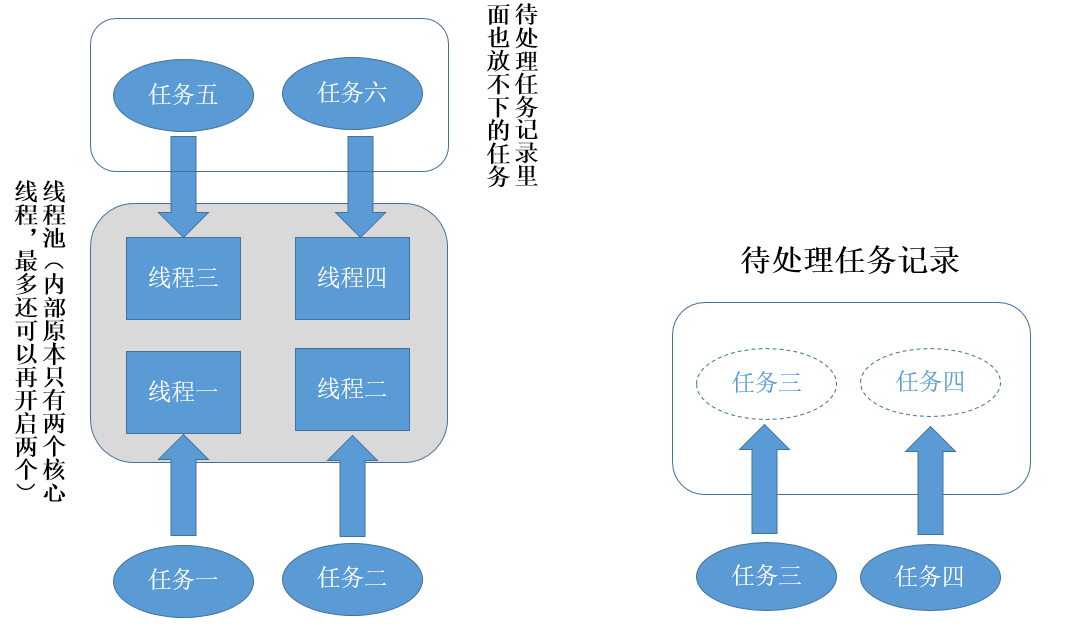
情况三：我们可以使用一种方式，将其记录下来，等前两个任务处理完成之后再接着处理第三个任务，这样可以一直高效的进行程序处理。



情况四：一下子来了六个任务，如果我们使用的记录方式只能记录有限个数的任务，当待处理的任务填满记录器之后，我们该如何处理？

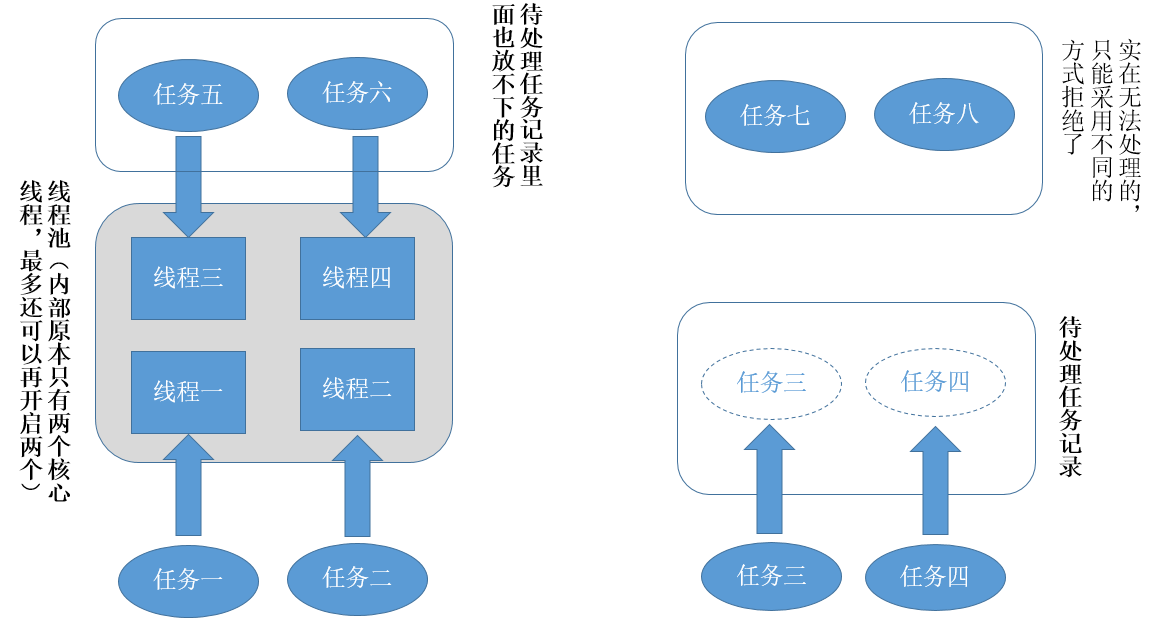


情况五：这个时候没有更好的办法，我们只能再多开几个线程将待处理任务记录器里面也放不下的任务直接处理了，但是这时候的整体处理效率会降低。



情况六：最复杂的情况，在现在的情境下一下子来了大于六个的任务，核心线程在处理的同时待处理任务记录器也记录满了，重新开了俩线程也都在执行任务，实在没有办法了，只能采取措施将不同的任务拒绝了。

思考：为什么不无限制的多开线程来处理多个任务呢？



* **原理**

线程池执行器，即是一个负责任务分配管理的线程。可针对不同的情况选用不同的应对策略。

* **示例**

详见zebrademo中关于任务管理器的示例

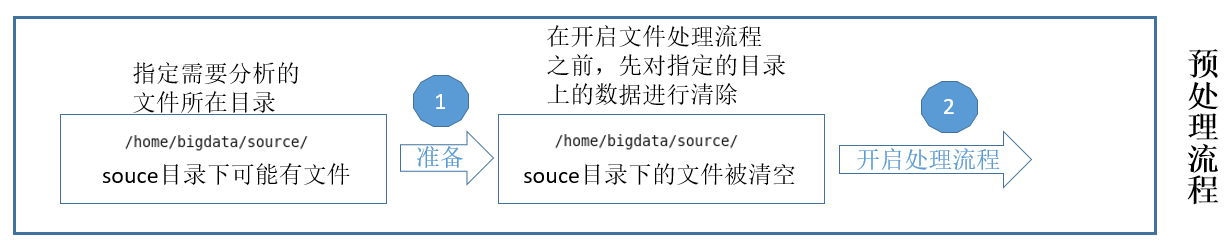
### NIO文件读写

详见zebrademo中关于NIO的示例

## 预处理（前置清理）

### 预处理概念

因为我们指定的存放待处理数据的的目录下可能存在一些以前存放的无用数据，可能会影响我们后续的一系列的处理流程，故我们先通过一个前置过程将其删除掉，这个过程我们就称之为预处理（也可以称之为前置清理）。如下图所示。



### 预处理流程图



需要注意的是，这里要使用递归删除，保证指定的目录下为空。

### 技术实现demo

package big.tedu.file;

import java.io.File;

import java.io.FileNotFoundException;

/\*\*

\* 目录处理工具类<br>

\* 提供一个递归删除某个目录下所有文件的方法

\*

\* @author tarena-mmy

\*

\*/

public class DirUtil {

/\*\*

\* 清空指定目录下的文件

\*

\* @param dir

\* 指定的目录

\* @throws FileNotFoundException

\*/

public static void ClearDir(String dir) throws Exception {

/\* 判断指定目录是否为空 \*/

if (null == dir || "".equals(dir)) {

throw new Exception("指定的目录为空");

}

// 根据指定地址创建文件对象

File file = new File(dir);

// 如果指定的目录存在就将其递归删除

if (file.exists()) {

// 调用递归删除一个目录下的方法将其删除

deleteFile(file);

if (file.list().length == 0) {

System.out.println("目录清空完毕！！！");

} else {

System.out.println("目录清空失败，可能文件正在被占用，请检查");

}

} else {

throw new FileNotFoundException("指定的目录不存在");

}

}

/\*\*

\* 递归删除一个指定file(可以是一个非空目录)

\*

\* @param file

\* 一个给定的目录

\* @return 返回空（即处理过程中没有出现异常）表示删除成功

\*/

private static void deleteFile(File file) {

File[] files = file.listFiles();

for (int i = 0; i < files.length; i++) {

if (files[i].isDirectory()) {

deleteFile(files[i]);

} else {

files[i].delete();

}

}

}

/\*\*

\* main测试方法

\*

\* @param args

\*/

public static void main(String[] args) {

try {

ClearDir("d:\\a");

} catch (Exception e) {

e.printStackTrace();

}

}

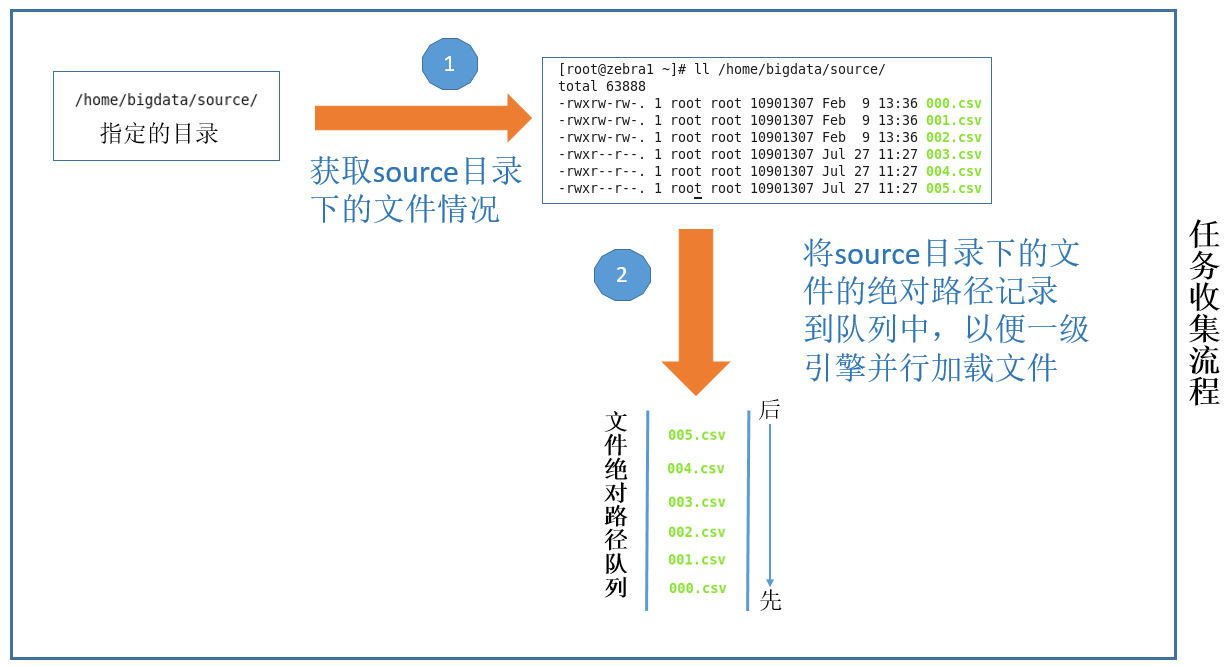
}

## 任务收集

### 概念

任务:在这里，我们称一个待处理的文件为一个待处理的任务。

我们想要通过一级引擎进行数据读取和分发的话，就需要提前告知一级引擎我们需要读取和分发的文件有哪些，我们通过一个线程来持续不断的将存放待处理数据的目录下的所有待处理文件的绝对路径收集起来，放置到一个阻塞式队列中，以供后续流程进行数据读取和分发，这个过程我们就称之为任务收集。如下图所示：



### 技术流程图



注意

1，我们这里有一个标识文件的概念，所谓的标识文件指的是基站将日志文件传递到我们能够处理的地方之后再传递过来一个同名的不同后缀的空文件，以用来标识日志文件已经传输完毕了。所以标识文件的作用就是一个传输完成的信号。

2，如果指定目录下没有数据，线程就休眠10秒，可以保证线程不死循环，减轻对指定目录的扫描压力。

### 技术实现demo

package big.tedu.task;

import java.io.File;

import java.util.concurrent.LinkedBlockingQueue;

/\*\*

\* 任务收集器<br/>

\* 本质为收集一个文件夹下的所有已经传输完成的数据文件的绝对路径，一个绝对路径(即一个待处理的文件)可以理解为一个待处理的任务<br>

\* 需要注意的是，在java中无论是一个目录还是一个文件，都是File类的一个对象

\*

\* @author tarena-mmy

\*

\*/

public class TaskCollector implements Runnable {

// 使用有锁（阻塞式）队列盛放未处理的文件

// 这里一个未处理的文件可以理解为一个任务

public LinkedBlockingQueue<String> pendingFiles = new LinkedBlockingQueue<String>();

// 需要收集文件信息的文件夹

public String path = "";

public void run() {

if (null == path || "".equals(path)) {

System.out.println("指定目录为空");

}

while (true) {

File pendingFile = new File(path);

// 如果待处理的目录存在就将其加入到任务队列中

if (pendingFile.exists()) {

taskCollect(pendingFile);

} else {

System.out.println("指定的目录不存在");

}

/\* 一次任务收集完成之后线程休眠5秒 \*/

try {

System.out.println("一次任务收集完成，线程休眠5秒~");

Thread.sleep(5000l);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

/\*\*

\* 任务收集<br>

\* 本质：将一个待处理的文件的绝对路径或者一个目录下的所有文件的绝对路径放入到任务队列中

\*

\* 此文件需保证已经是一个传输完毕的文件 确认方式：在其所在目录内有一个同名的.ok后缀的确认文件，该确认文件为空

\*

\* @param file

\* 一个待处理的文件或目录

\*/

private void taskCollect(File file) {

/\* 按照指定地址创建的File是一个目录 \*/

if (file.isDirectory()) {

// 获取一个file（其实在这里是一个目录）下的所有数据文件

File[] files = file.listFiles();

/\* 循环将所有文件的绝对路径添加到队列中 \*/

for (int i = 0; i < files.length; i++) {

File temp = files[i];

// 如果指定的file是一个目录就继续递归执行

if (temp.isDirectory()) {

taskCollect(temp);

} else {

dealFile(file);

}

}

} else {/\* 按照指定地址创建的File是一个文件 \*/

dealFile(file);

}

}

/\*\*

\* 处理一个文件<br>

\* 如果文件是一个数据文件，且数据传输完毕的确认文件存在就将其添加到队列中

\*

\* @param file

\* 这里的file确定是一个文件而不是一个目录

\*/

private void dealFile(File file) {

if (file.getName().matches("^.+\\.data$")) {

String filePath = file.getAbsolutePath();

String checkFileName = filePath.substring(0,

filePath.lastIndexOf("."))

+ ".ok";

File checkFile = new File(checkFileName);

if (checkFile.exists()) {

pendingFiles.add(filePath);

checkFile.delete();

} else {

System.out.println("数据文件存在，但是确认文件不存在。\t数据文件绝对路径为：" + filePath);

}

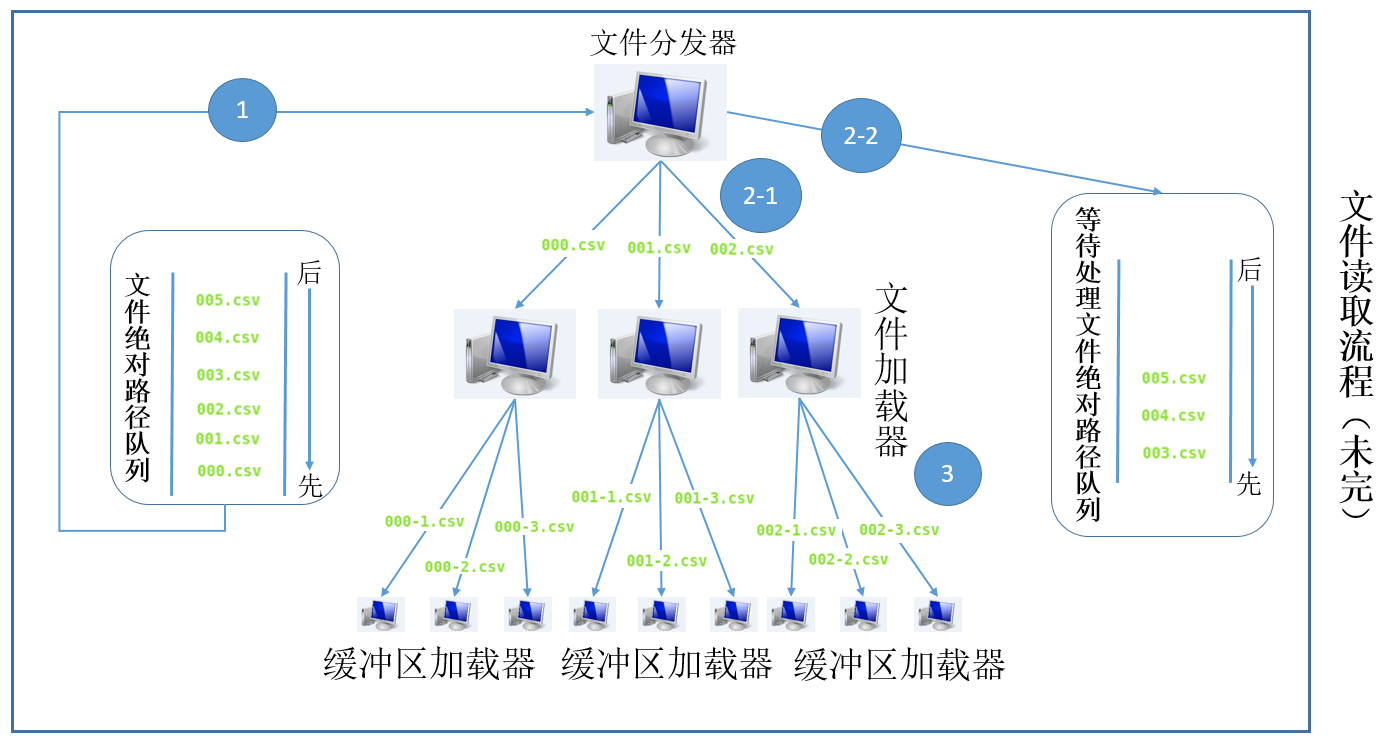
}

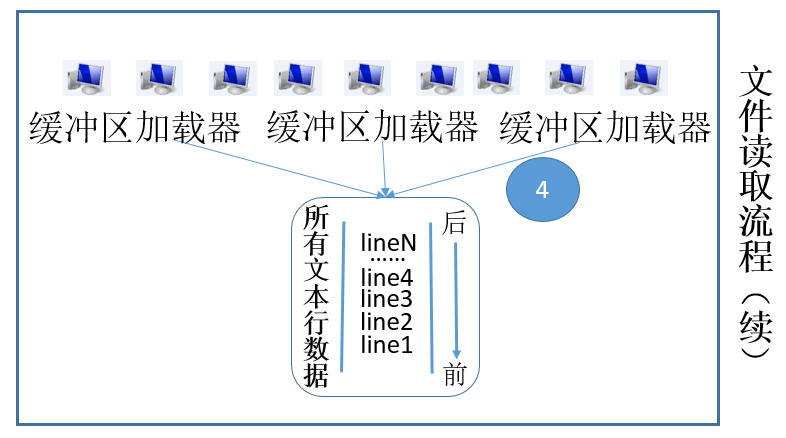
}

}

## 文件内容读取

### 整体概念





如图所示，在我们的文件内容读取流程中，一共有这么几个角色：文件分发器、文件加载器、缓冲区加载器。文件分发器从收集到的任务队列中的待处理文件分配给不同的文件加载器进行加载，一个文件加载器在同一时间只负责一个待处理的文件。然后文件加载器将自己负责的那个待处理文件按照合适的缓冲区的大小切分成多块，再将每个小块指定给一个缓冲区加载器。缓冲区加载器将每个小块中的字节数据切割成行文本数据存放到一个队列中，等待后续处理。

### 文件分发器概念

本质是一个内置ThreadPoolExecutor的线程。它负责将收集到的待处理文件的绝对路径从队列中取出分发给文件加载器进行处理。每个文件加载器在同一时间处理一个文件。

### 文件分发器流程图

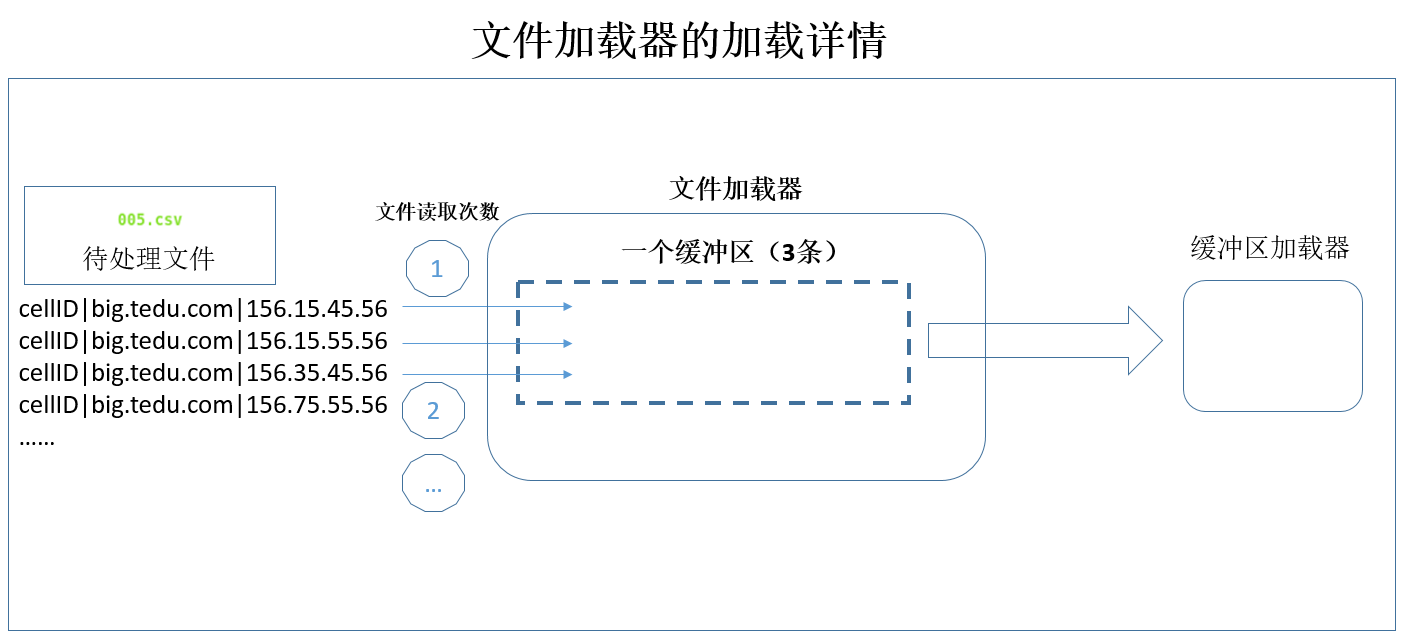


在这里需要注意的是，存放待处理文件的数据容器为阻塞式队列，可以保证在队列中没有待处理文件的情况下线程阻塞指定时间，这样可以保证线程不一直从队列中获取数据，减轻对队列的压力。

### 文件加载器概念

本质是一个内置了CacheThreadPool类型的ExecutorService的线程，根据文件分发器分发过来的文件的绝对路劲创建File，先将文件读取到内存中，注意：到这里为止，进入到内存中的内容为字节，我们无法对字节直接进行运算，所以我们将一个缓冲区内的字节通过缓冲区加载器转换成行文本，以便于能够进行后续运算处理。

思考：我们按照缓冲区大小进行切割可以保证每个文件块的大小，但是无法保证每个文件块的最后一行肯定是一行完整的数据，我们需要怎么解决这种问题？



### 文件加载器流程图



需要注意的是，用来管理缓冲区加载器的线程管理器类型为CacheThreadPool，相比较ThreadPoolExecutor来说，我们可认为ThreadPoolExecutor是一种标准的线程池执行器，我们可以自定义核心池大小最大池大小等等所有参数。而CacheThreadPool类型的执行器就是java自身提供的写好的一个确定的ThreadPoolExecutor，主要用在小碎的多任务场景下，能够提升系统的处理效率。

### 缓冲区加载器概念

缓冲区加载器的本质是一个线程，将缓冲区内的数据按照行文本的形式添加到全局的存放行文本的队列中。

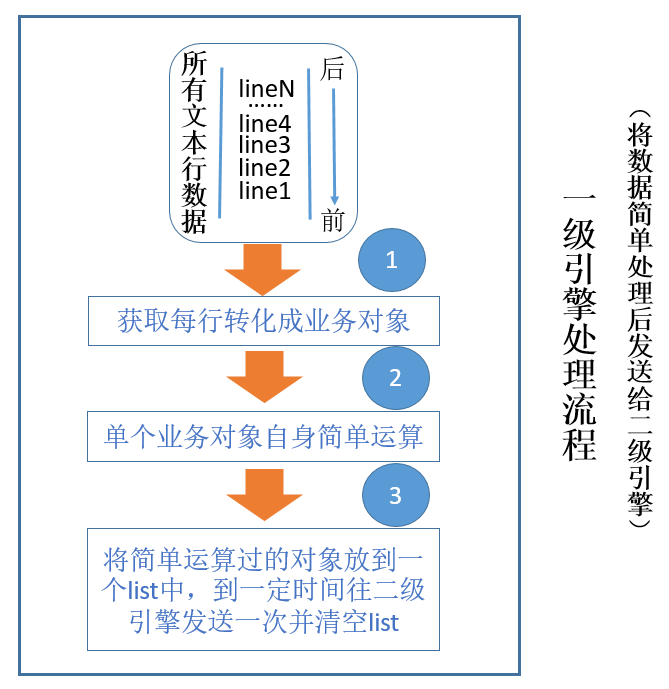
### 缓冲区加载器流程图



## 一级引擎

### 一级引擎概念

一级引擎实质也是一个线程。其将前面所读取到内存队列中的数据经过转换和简单运算后交由二级引擎进行处理。



### 一级引擎流程图



需要注意的是，这里使用了时间缓冲的概念。意思就是将多个从队列中获取的简单计算过的对象添加到一个无限大小的集合中，添加一段时间之后再往二级引擎发送。这样可以避免获取一条就发送一条，减少网络间数据发送所带来的开销。相应的，我们也可以使用空间缓存，即集合的大小是限定的，比如说是1000条，等到这1000条放满的时候我们再将其发送出去。时间缓冲和空间缓冲各有利弊，详见下表：

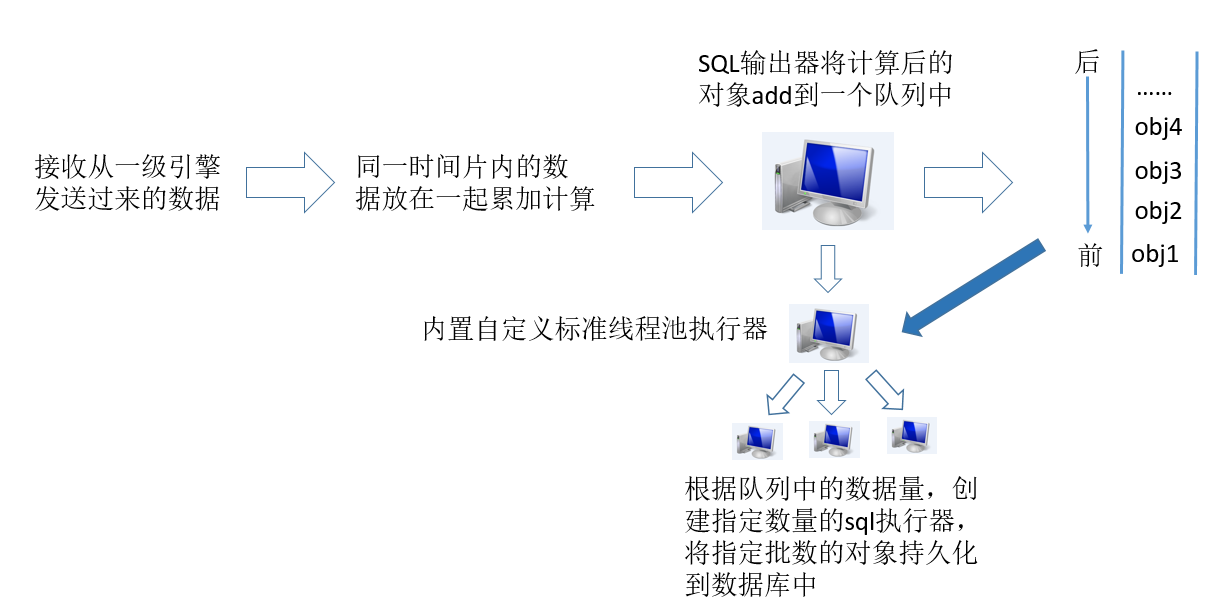
|  |  |  |
| --- | --- | --- |
| **缓冲类型** | **优点** | **缺点** |
| 时间缓冲 | 减少网络传送开销 | 可能会造成内存溢出 |
| 空间缓冲 | 减少网络传送开销 | 可能会造成时间过长 |

总结：在实际的应用程序中，应该结合时间缓冲和空间缓冲一起使用，保证缓冲的安全、高效。

注意：我们经常听到另一个词汇——缓存，缓存和缓冲完全是两个概念。缓存的常见场景在web应用程序中，将一些热门数据存放在内存中，以供快速读取，减轻数据库的压力。

## 二级引擎

### 二级引擎整体图解



二级引擎部分稍微简单一些，接收从一级引擎发送过来的数据，缓冲到一定条数后，批量交由多个sql执行器来进行执行。

### sql输出器

本质是一个内置标准线程池执行器的线程，接收从一级引擎发送过来的对象，按照时间片将其内部的数据按照某些相同的字段进行累加，并放置到一个map中。到达一定时间后将map中的数据分批交给sql执行器持久化到数据库中，并清空盛放待持久化数据的map容器。然后继续整体循环。

### sql执行器

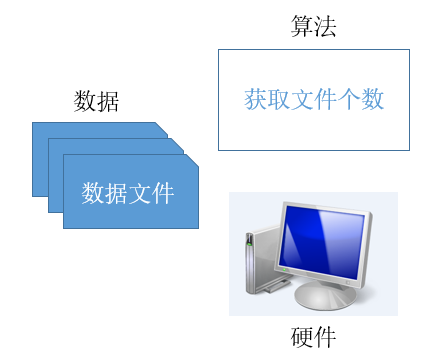
根据拿到的一批对象，遍历将其转换成持久化语句（将对象上的属性封装到持久化sql语句上），然后批量化执行持久化sql语句。

# zebra总结

## 分布式应用程序分析

### 分布式应用程序三要素

如图所示，分布式应用程序三要素如下：



* **数据**

即需要运算的数据，比如：zebra项目中的电信日志就是我们需要处理的数据

这里的数据的形式可以是文件、数据流等

* **算法**

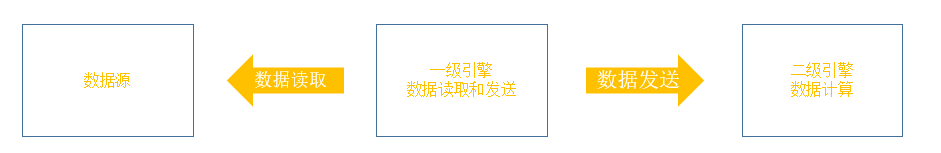
就是对数据的运算，比如：zebra项目中获取所有文件、将行转换成对象等等

算法是一个很宽泛的概念，我们程序中所有的处理都可以看成是算法

* **物理硬件**

即运行程序的机器，我们所有程序的运行都需要依赖具体的物理硬件，有cpu、内存、硬盘等

## zebra总结



整体流程为：

一级引擎从数据源读取数据，并将数据发送给二级引擎，二级引擎对数据计算后再进行其他操作（如：数据持久化到关系型数据库）。

需要注意的是：

二级引擎中被计算的数据在计算完成后就丢失了，只是结果数据被保存。