# 输入和输出

Graphhopper是通过输入一系列的gpx数据点，输出其匹配文件。

# 处理输入：运行python文件

初始数据为一个txt文件。

输入读取的批量txt所在的文件夹的路径，读取包括子文件夹在内的所有txt文件（因为服务器上有一个table\_export.txt，读取的话有点小问题，所以有一个对比字符串，不读取它）。

**对于每个txt文件：**

首先过滤超过地图界限的点。float(wd) <= 24.9 and float(wd) >= 24.4 and float(jd) >= 117.7 and float(jd) <= 118.4**（可能之后要重新生成一个滤过点后的txt文件，否则之后的路段号的txt与原txt匹配会出现问题）**

写入gpx文件时，(len(c)-2)/2-cal\_max(list1[count1]) > 4 and (len(c)-2)/2 > 6，就是过滤掉gpx点过少（点数小于6，我认为较少点的匹配是没有意义的）的文件同时检查出现次数最多的点的次数，如果所有点的个数-出现最多次数的点的次数 > 4，则认为是合理的数据，就写入一个out文件夹，不合理的数据写入wrong文件夹。（当然，这个数字的大小是可以修改的，判定条件也是可以修改的）

经过python文件的运行，将产生一系列的gpx文件，从而可以直接在jar中运行。

# 如何运行java文件

**mvn clean**

//将目录下生成的target文件移除

**mvn package –DskipTests**

//生成目录下生成的target文件

**java -jar matching-web/target/graphhopper-map-matching-web-0.11-SNAPSHOT.jar import map-data/xiamen.osm**

//解析osm地图文件，生成graph-cache文件夹，文件夹中包含了地图信息，之后就不再需要osm文件。

**java -jar matching-web/target/graphhopper-map-matching-web-0.11-SNAPSHOT.jar match matching-core/src/test/out/\*.gpx**

//输入一个文件夹中的所有gpx点，输出两个文件夹，result和street，其中result包含匹配后的gpx数据，street包含每个gpx数据对应的路段号，该路段号与原始gpx文件一一对应（如果要对应txt，不要忘了超出地图范围的已经在python中被删除的数据点）。

**java -jar matching-web/target/graphhopper-map-matching-web-0.11-SNAPSHOT.jar server config.yml**

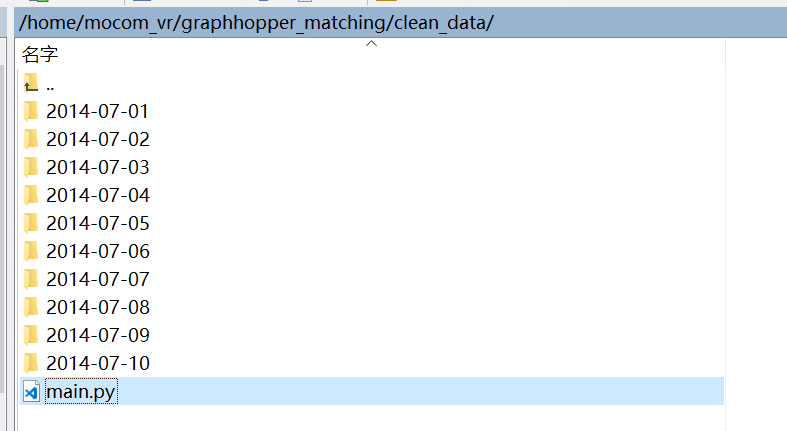
//运行出一个可视化界面：<http://localhost:8989>，选择路段gpx文件，在图中显示匹配信息。黑色为原始数据，绿色为匹配结果。

# 服务器上的运行

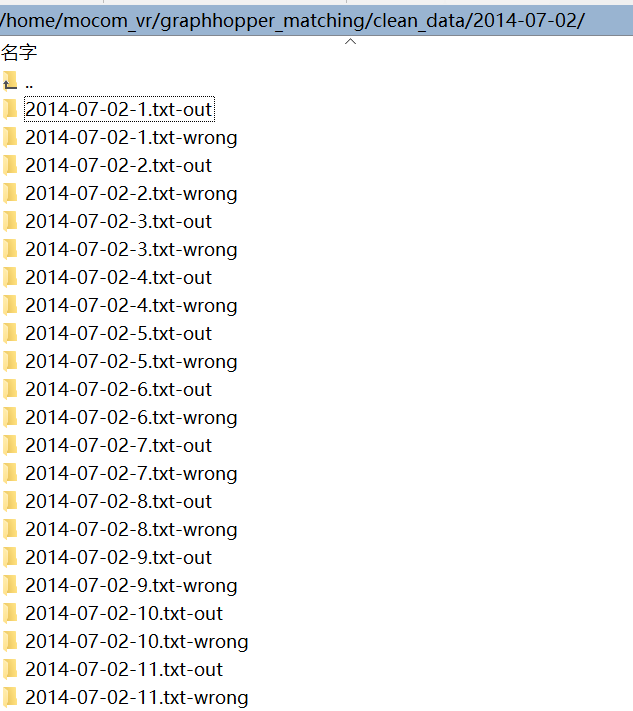
在服务器的/home/mocom\_vr/graphhopper\_matching/目录下，有两个文件夹，clean\_data和map-maching-master。

## Python文件

其中clean\_data中包含一个main.py文件，运行该文件：python main.py，处理输入数据。



得到输入的gpx数据的文件夹。

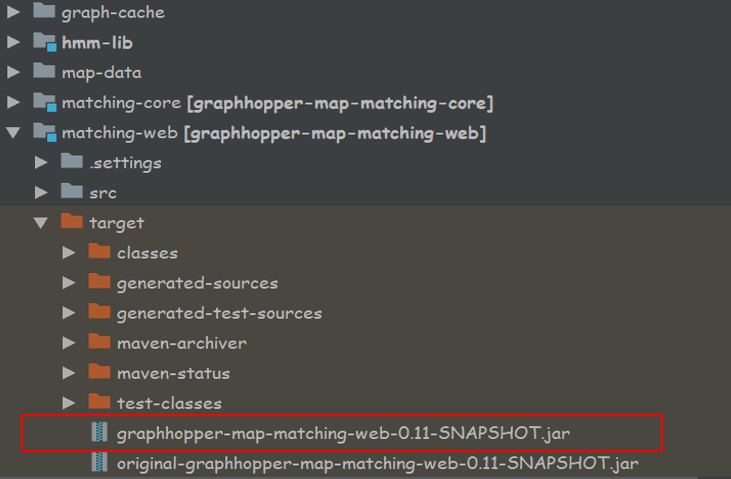


out为合理的数据，wrong为不合理数据。

## Java文件

/home/mocom\_vr/graphhopper\_matching/map-matching-master/目录下，有一个文件夹graph-cache，其中包含了地图信息。

除此之外，还有一个jar包。该jar包是在本地**mvn package –DskipTests**命令运行后，在target目录下的。



可以发送到服务器上运行该jar包。

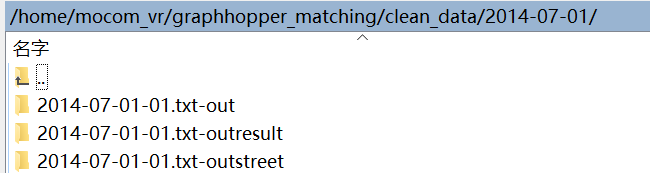
**java -jar graphhopper-map-matching-web-0.11-SNAPSHOT.jar match** /home/mocom\_vr/graphhopper\_matching/clean\_data/2014-07-01/2014-07-01-01.txt-out/\*.gpx

match为运行调用的类class MatchCommand extends Command（目录名：Matching-master\matching-web\src\main\java\com\graphhopper\matching\cli\MatchCommand.java）

该类的name属性为match，后接gpx文件夹的路径名。

目前只做了一个一个文件夹的匹配，理论上是可以做到包括子文件夹的全部匹配

运行后产生两个文件夹。



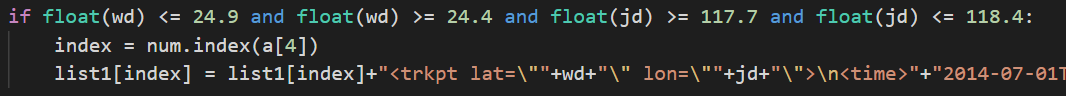
result和street，其中result包含匹配后的gpx数据，street包含每个gpx数据对应的路段号，该路段号与原始gpx文件一一对应（如果要对应txt，不要忘了超出地图范围的已经在python中被删除的数据点）。

# 如何修改

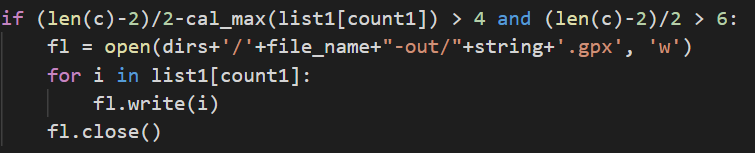
## Python文件

Python文件只是为了解析为gpx文件夹，可改的参数：

地图边界：该if语句



是否删除一条gpx数据的判断条件：（目前是根据重复数据点的个数与总点的个数判定）



Python文件较短，其他内容就不详述了。

## Java文件

### doWork函数

matching-core\src\main\java\com\graphhopper\matching\MapMatching.java中的doWork函数——>处理gpx数据的主函数。

public MatchResult doWork(List<GPXEntry> gpxList,File gpxfile)

1. 过滤过于相近的节点
2. 找到每个点的匹配边（A\*算法）
3. 创建一个图表，为这些边创建虚拟节点
4. 删除重复节点（虚拟节点不会被删掉）
5. 从所有GPX条目的查询结果中创建候选项
6. 计算最有可能的候选项
7. 传递返回值

该函数中原算法的第一步是filter，滤过部分相近的点，但是我们要输出每个点的对应的路段号，我的做法是去掉了这一步，其实还有一种做法可以尝试一下：仍然按照原算法滤过点，把滤过函数（filterGPXEntries）中滤过的点信息保留下来，在最后为其赋值——>与其相近的那个点的路段号。

第二步是找到相近的节点，利用A\*算法，找到匹配的边。

……这一个函数详细内容可以参考我之前报告时的一个ppt。

在这个函数的最后，做了一个输出，输出了以车辆命名的一个包含每个点的路段号的txt。

### run函数

在matching-web\src\main\java\com\graphhopper\matching\cli\MatchCommand.java中的run函数，即在命令行输入的match命令运行的函数。

首先加载地图数据，初始化一些变量。

循环每一个gpx文件。for (File gpxFile : args.<File>getList("gpx"))

调用doWork函数等函数得到匹配结果。

输出匹配后的gpx文件：new GPXFile(mr, il).doExport(outFile);

输出out.txt，包含未能成功处理的gpx数据的文件名。

**理论上可以完成但目前还没有做完的：**

目前是一次处理一个文件夹中的所有gpx数据（即一个小时的数据），可以改为一次处理一个文件夹包含子文件夹的数据（一天的数据）。（可以在MatchCommand.java中修改arg的for循环，读取子文件夹名中**以-out结尾的**子文件夹（注意：不要去读取结尾为xxxx-wrong的文件夹或者之前产生的xxxx-outresult文件夹中的gpx文件））

把doWork函数中输出的street文件夹中的txt与包含所有数据的txt（python处理之后要生成的一个新的txt）对应，理论上，可以根据street文件夹中的txt的文件名把其中的数据按顺序找到对应的该车辆的多条数据。（可以再写一个python文件做这个处理，为了这个处理简便，可以考虑把原始数据的txt按照车辆编号有序排列，再做这个对应也许会简单一点）。

Run函数中输出的out.txt，即包含未能处理的gpx文件的文件名的txt(可能需要改路径和文件名，现在这种文件名的会在下一次运行中被重写)

# 附录

项目我上传在了github上。

Python: <https://github.com/bepolygonum/Matching-master/blob/master/main.py>

Java: <https://github.com/bepolygonum/Matching-master>

其中还包含了之前报告时用的ppt(Graphhopper.pptx)与java项目阅读整理的一个word文档(map\_matching阅读整理.docx)。

Email：961646438@qq.com