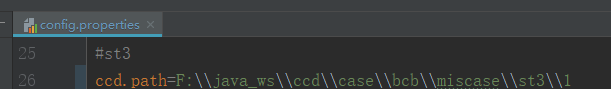
工具检测部分

**Java 端**

方法代码=====🡺key,(token序列+规则结点序列)=====🡺redis数据库

配置config.properties文件:



这里要改成被检测文件的全路径.

配置好后直接执行Main类的主方法即可.

**MethodVisitor做的事情:**

将每个java文件对应的AST进行遍历

由于工具的检测粒度为整个方法代码,

因此遍历AST时, 收集以方法为根的子树的结点序列.

收集过程分为两个部分:

**第一部分**是收集规则结点(Rule Node), 实际收集的是rule node的Index, 相当于ID .

比如下面两行代码, 会得到两个node序列, 序列之间按分号;分隔.

,

=>

123,124,125,126**;** 123,124,125,126;

代码中将用**statement**来收集整个方法的node序列.

**第二部分**是收集token序列

按照代码行,收集每个方法的token 序列. 实际收集的是token的index,相当于ID.

比如下面两行代码, 会得到两个token序列,中间用;分隔



=>

50,202, 202, 202**;** 50,203, 203;

代码中将用**line**变量来收集整个方法的token序列.

Line和Statement拼接在一起, 中间用**ccdMethodSeparate**分隔.

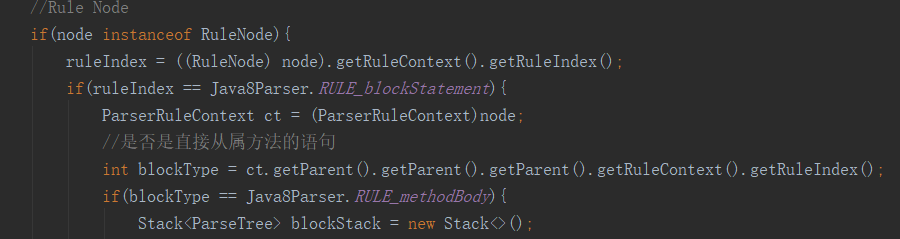
最后得到的**value**序列将代表一个方法. 同时作为**redis**的一条数据的**value**值传入.

**key**为方法所在的文件路径以及文件中开始行和结束行.(比如:”F://A.java(20-50)”)

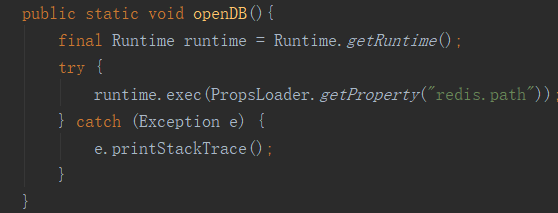
当被检测文件中所有方法转换为redis数据时, 这部分工作结束.

部分代码解释:

**Rule Node:**



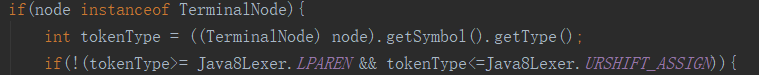
因为目前的做法是两层sent2vec, 第一层是方法体第一层级的code, 然后第二层是整个方法.举例:



上面方法,总共有两个BlockStatment(只考虑第一层级, 直接从属于代码体.), 第一个是final开头的变量声明,

第二个是整个try代码块.

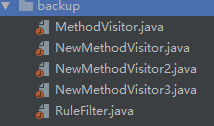
**Token**



遍历的第二部分, 考虑叶子结点(TerminalNode), 同时token的ID应该在一个范围内, 因为要除去各种符号的影响,比如{},[],+-\*/等等.

**Backup文件包:**

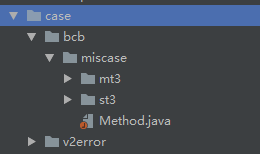
主要是存放以前版本的遍历模式.



**Case文件包:**

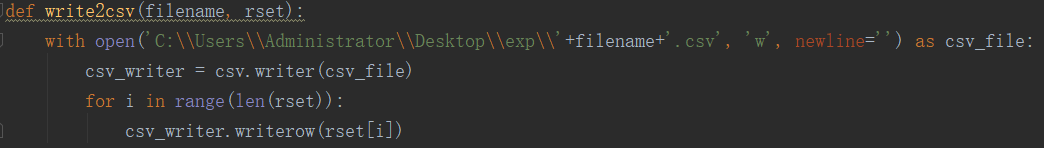
代码中放有miss掉的case, 这里面主要是当时跑bcb的结果, mt3:中三型,st3:强三型

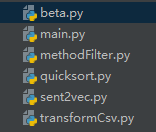
V2error是跑出来的FP:



**Python端**

修改路径csv文件输出的路径





Beta主要是求两个方法token相似度, 做成一个beta参数, 来优化结果.

Main主要是获取redis中的数据,然后调用两层sent2vec模型, 最后求方法之间的距离判断克隆.

MethodFilter主要用来过滤两个方法行相差超过3倍的情况,以减少比较次数,提升效率

Quicksort是在比较距离之前先对方法按照代码行从小到低进行快排,目的是为了减少比较次数,提升效率

Sent2vec句向量模型.

TransformCsv把检测结果转换为path1,file1,start1,end1, path2,file2,start2,end2输出到csv文件中.