GetNextStep的一些优化

- GetNextStep 可以很好的解决在server端的At-most-once与All-or-nothing问题，但是小车端的个人认为还存在一定的问题
- 我觉得维护的计数器不光存在数据库服务器（redis），还应该在小车端维护部分状态，保证小车也能做到对于指令的At-most-once与All-or-nothing

设计

- 也是一样维护这counter数值（n1），对于server返回给小车的下一步步骤也会存在一个counter（n2），比较此数值
- 当n1>=n2,说明此时server返回给小车的步骤已经过期，直接忽略
- n1=n2-1，正确的返回，执行，并且n1++
- n1< n2-1,系统出现比较严重的错误（需要对系统进行纠错）

对于timeout的一些设计（暂时不考虑系统crash，只考虑网络因素）

发生情况

- （1）小车对于下一步请求的timeout
- （2）server向redis请求的timeout
- （3）server对于小车回应的timeout
- （4）redis对于server回应的timeout

设计

- 针对（1），简单的使用“记时”机制，当超过一定时间（1s）没有得到回应，重发，与GetNextStep设计相互配合，避免重复执行
- 针对（2），因为考虑到redis数据库的特殊性，对于写入这一操作redis会先验证是否存在，可以简单的使用重发这一步来完成
- 针对（3），同理，因为server始终保证无状态，那么和（1）情况暂时没有本质区别，重发可以很好的解决问题
- 针对（4），同（2）同理，在server端重发即可，不需要考虑server的问题（因为暂时不考虑redis的crash）

对于crash的设计

发生情况

- （1）小车端的crash
- （2）server的crash
- （3）redis的crash

设计与目前的难题

- 针对（1），暂时考虑引入“心跳”机制，通过定时的小车端和server端之间心跳，来判断小车是否存活，但是现在的问题在于知晓小车crash后的操作应该如何执行
- 针对（2），因为本身就存在多个server，并且存在重发机制和负载均衡的设计，因此单一的server crash并不可怕，不回影响系统允许，但是难题在于如何找到那一台server crash以及如何恢复
- 针对（3），这是我们系统设计中最致命的地方，对于redis的恢复，我们考虑使用一直维护在小车端的数据来进行恢复（server反向向小车端进行请求），但是难题在于如何断定redis已经崩溃