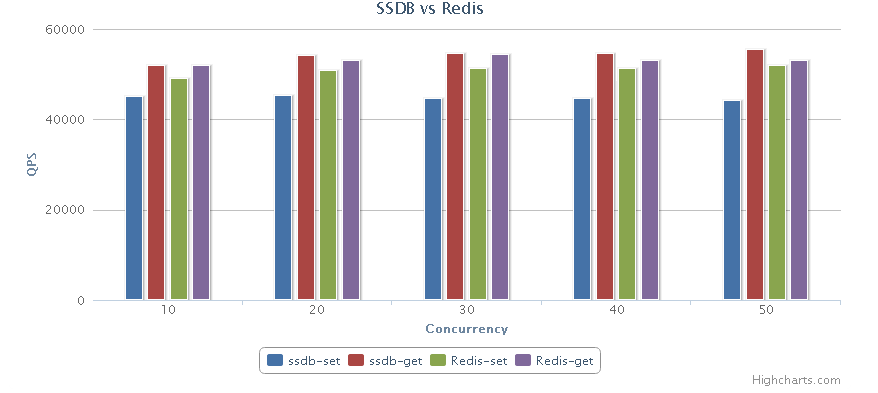
数据库介绍

1. 目前主流数据库
   1. 关系：Oracle，mysql，其他。
   2. 不仅仅关系：mongoDB，redis，ssdb，levelDB，OceanBase，Cassandra，HBase，rocksdb。
2. 游戏数据库特点
   1. 存储，查询，单连接（也有多连接的，激战2），排行榜。
3. 游戏数据库
   1. mysql：大部分公司都可能会用到。
      1. 天龙八部的ODBC+sharememory。
         1. 具体实现
      2. 剑网3的mysql+存储过程
      3. 骑士信条缓存mysql+拼凑sql语句。
      4. 上述的数据库都可以将逻辑不放在数据库。基本上都能达到全自动化的方式。
   2. redis和ssdb：协议是同步的，
      1. 快速开发的小公司。易上手。
      2. 据说新浪和小米也是用redis，官网上有github，twitter，Instagram，alibaba等。<http://techstacks.io/tech/redis>
      3. SSDB，360游戏（貌似就是SSDB作者ideawu），百度。游戏网，九游，穷游，酷狗等。



* 1. 自研：
     1. 完美的这种key，value数据库。
     2. 冰火的key，value数据库。

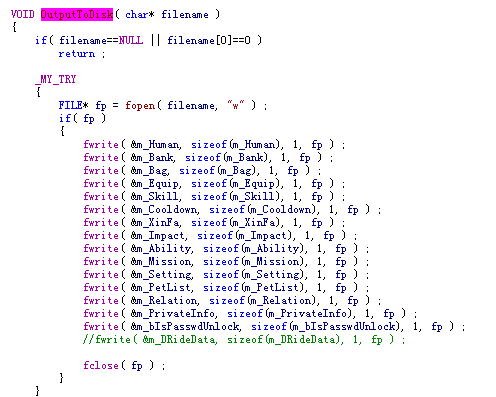
1. 天龙八部的ODBC+sharememory具体实现
   1. sharememory具体实现。

1.extern SMUPool<HumanSMU> g\_HumanSMUPool;（创建一个池子）

2.HumanSMU\* pSMU = g\_HumanSMUPool.NewObj();（为每一个对象保存数据）

3.SMULogicManager<HumanSMU>\* pHumanSMULogicMgr = static\_cast<SMULogicManager<HumanSMU>\*>(m\_SMULogicManager[i].m\_SMULogicManager);（这里定时向数据库写入，相当于调用后BOOL SMULogicManager<HumanSMU>::DoSaveAll()）

4.全部写入文件



* 1. ODBC具体实现

在sharememory中将mysql连接上，然后拿着这个ODBCInterface这个指针就可以进行保存了。

具体如下：



* 1. 线程模型，单线程。

1. redis的游戏实现方案一
   1. redis实现的大体方案是拼凑redis的sql语句。
   2. 实现流程如下：

大体就是通过DBThreadPool创建N个DBConnectThread线程。每个线程都拿着redis的一个连接。并且每个线程都从DBCommandPool中选择一个DBCommand来执行。

* 1. redis的Sql语句，如下：

\_snprintf(szCommand, MAX\_COMMAND\_LEN, "HMGET %d\_%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s",m\_nGuid, DBKEYWORD\_PLAYER\_GENERAL\_INFO……………..

然后调用pDBConnector->ExecuteRedisCommand(szCommand);即可执行。同理保存也是一样。

上述能这样做的原因是redis自身支持。

redis> HMSET website google www.google.com yahoo www.yahoo.com

OK

redis> HGET website google

"www.google.com"

redis> HGET website yahoo

"www.yahoo.com"

* 1. 线程模型：

如上所示:多线程的情况下选择具体的sql语句来执行。这点类似笑傲的多线程。

另外如果一条DBCommand执行完后。需要逻辑层来处理一些逻辑。这个时候采用回调的方式来处理。这么做的目的是为了逻辑分离。不加锁。当你的逻辑线程再跑的时候将这个回调的消息取出来。

* 1. 特别的。对于DBCommand而言。这里采用的是工厂模式实现的。相当于天龙的消息的工厂。

1. 骑士信条的数据库实现（mysql）
   1. 大体还是拼凑sql。只不过是自动化的拼凑。数据库一次性写完后。就不需要管理了。
   2. 核心为recordset。即不管是查询还是写入都通过recordset进行中转。见下图。
   3. 
   4. 举个例子：
   5. 写入数据库

//保存一级属性

{

m\_objRecordPtr->SetDec(db::fields::Character::strength,attrs[STAT\_STRENGTH]);

m\_objRecordPtr->SetDec(db::fields::Character::intellect,attrs[STAT\_INTELLECT]);

m\_objRecordPtr->SetDec(db::fields::Character::vitality,attrs[STAT\_VITALITY]);

m\_objRecordPtr->SetI32(db::fields::Character::areaid, attrs[STAT\_INT\_AREAID]); }

1. 从数据库导出

// 从数据库中载入符文栏信息

{

size\_t size = m\_objRecordPtr->GetI32(db::fields::Character::rune\_size);

}

Table缓存表。其实就是将数据做一次缓存。只要数据插入，更新。缓存表都有一份。删除的时候也会将缓存表删除。

优点：

基本上算全自动化的数据库。插入，更新，删除，只需要传递对应的数据，就能生成sql语句。

更新数据已经有ORM的模型了。

缺点：

缓存表相当于所有数据都缓存。没有使用到缓存算法。

查询的时候返回的是所有数据。如果不需要返回所有数据，则需要定制RPC消息。

对于游戏的单链接。mysql的性能提升不了。

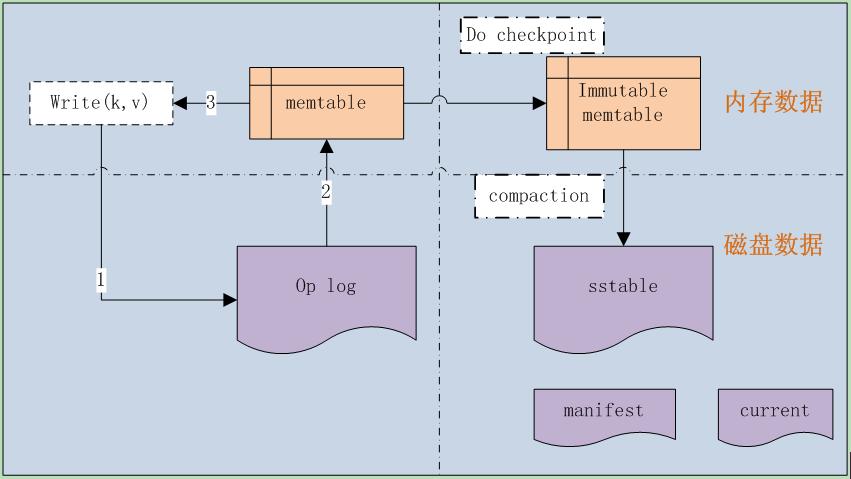
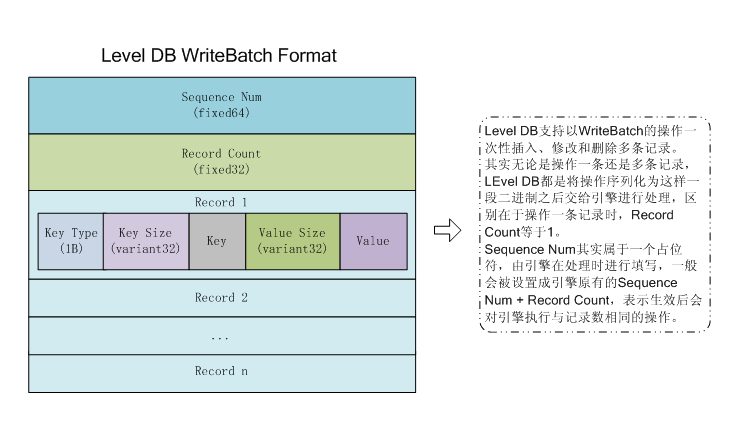
线程模型：

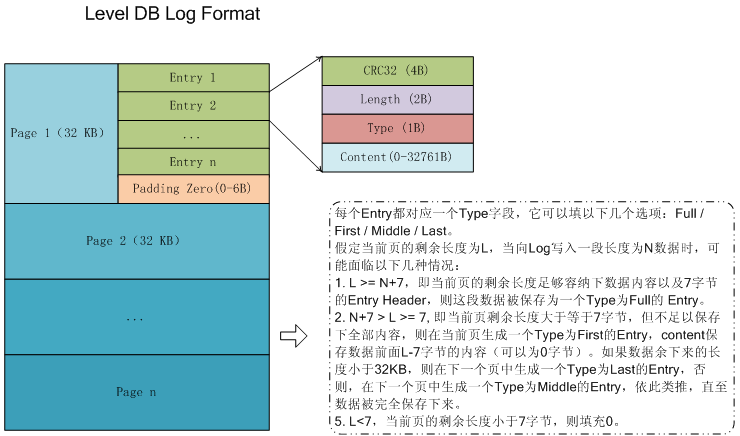
两条线程。一条用于类似ORM模型的通道。一条用于普通的SQL语句。

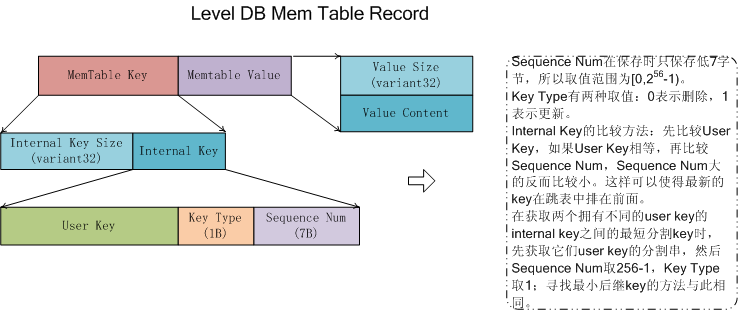
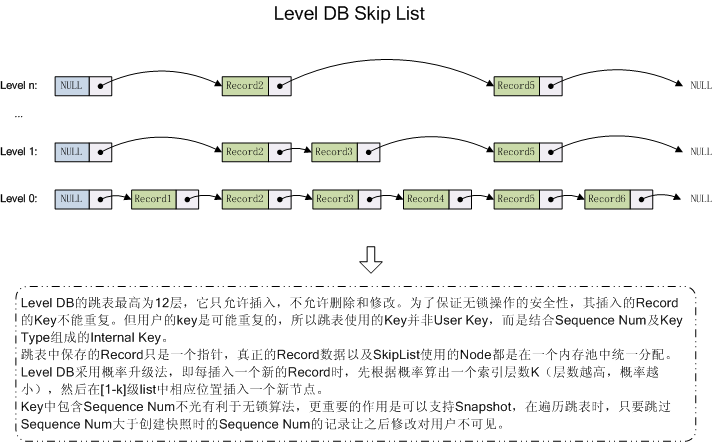
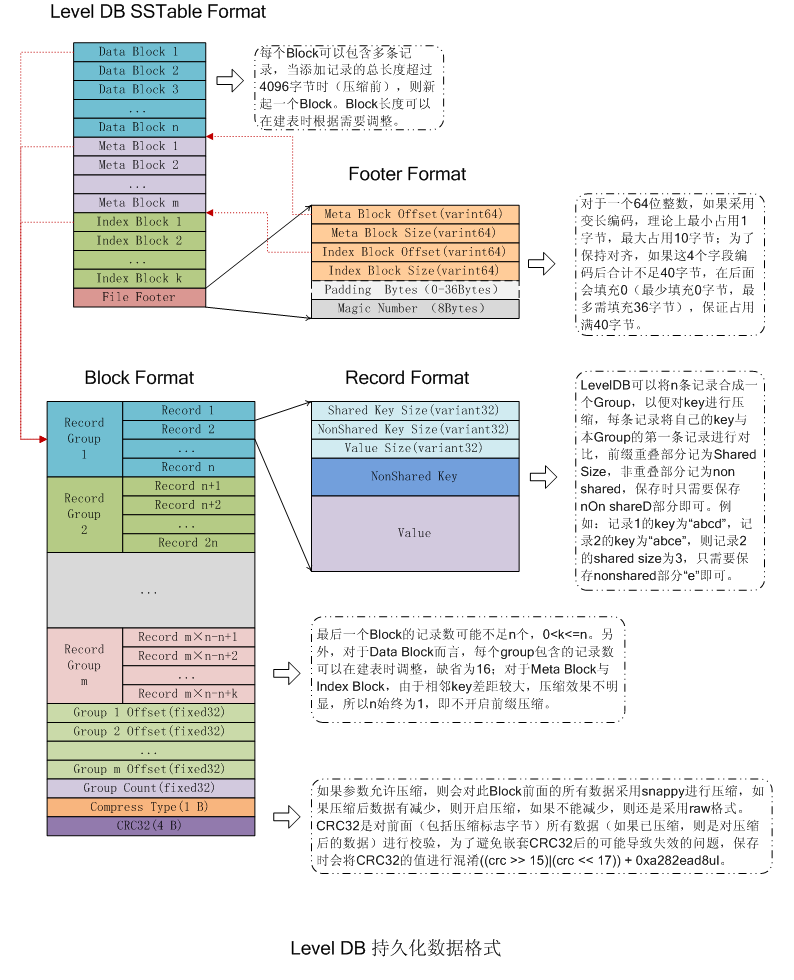
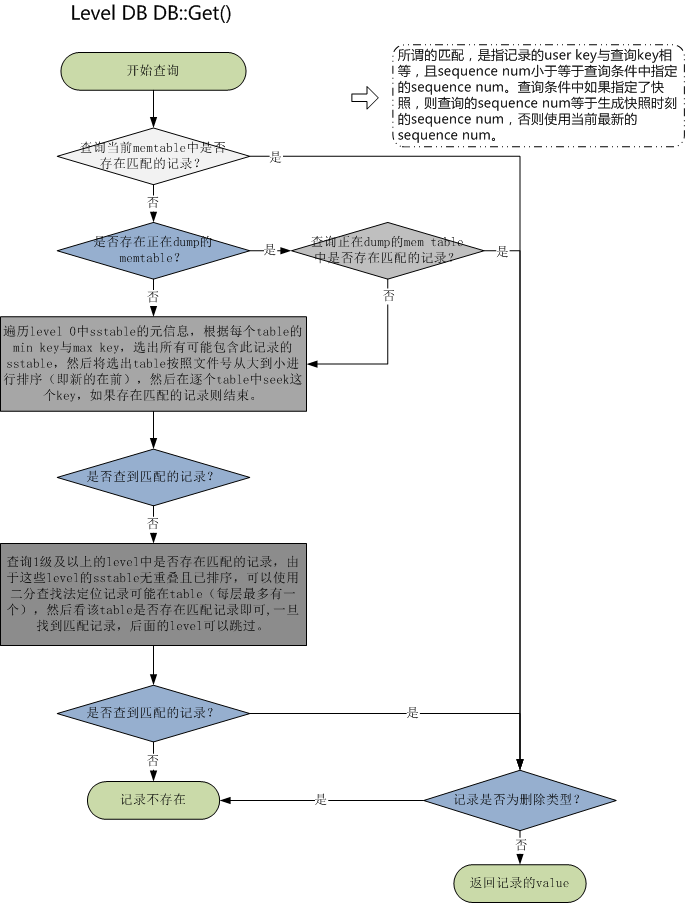
1. 剑网3的数据库（mysql）
   1. 这套数据库没源码。大概实现思路如下：
      1. 写mysql脚本，就是存储过程。在脚本里写好每一个table的插入，删除，更新，获取数据。
      2. 上层C++代码调用具体的存储过程就可以了。
      3. 这个黄磊写过，具体的就不班门弄斧了。
2. 冰火数据库
   1. 这套数据库实现底层采用levelDB作为存储引擎。
   2. 
   3. levelDB实现：
      1. 介绍

Level DB（<http://code.google.com/p/leveldb/>）是google开源的Key/Value存储系统，它的committer阵容相当强大，基本上是bigtable的原班人马，包括像jeff dean这样的大牛，它的代码合设计非常具有借鉴意义，是一种典型的LSM Tree的KV引擎的实现，从它的数据结构来看，基本就是sstable的开源实现，而且针对各种平台作了port，目前被用在chrome等项目中

Level DB是典型的Log-Structured-Merge Tree的实现，它通过延迟写入以及Write Log Ahead技术来加速数据的写入并保障数据的安全。LevelDB的每个数据文件(sstable)中的记录都是按照Key的顺序进行排序的，但是随机写入时，key的到来是无序的，因此难以将记录插入到其排序位置。于是需要它采取一种延迟写入的方式，批量攒集一定量的数据，将它们在内存中排好序，一次性写入到磁盘中。但是这期间一旦系统断电或其他异常，则可能导致数据丢失，因此需要将数据先写入到log的文件中，这样便将随机写转化为追加写入，对于磁盘性能会有很大提升，如果进程发生中断，重启后可以根据log恢复之前写入的数据。

* + 1. 大致框架
  1. 
  2. 写入数据库执行流程
  3. 外界不管是写入，删除，还是更新数据库都是通过Write（k，v）写入，调用这个接口后。writebatch数据格式为
  4. 
  5. write完毕后，会生成对应的Log信息log日志的格式如下



1. 当写完Log后，会将Log日志的信息转换为memtable。即即将开始写入数据库，memtable的格式如下，memtable是采用skiplist作为数据结构来存储数据。
2. 
3. 
4. 当发现memtable数据存储满了后，再讲这个memtable指针赋值给imm(Immutable Memtable)。imm再执行写入level 0.并且重新申请mmetable供下一个使用。
5. 写入memtable后需要将数据写入level 各个层级。写入是将skiplist中的数据dump到磁盘上，生成一个sst文件，sstable文件格式如下：
6. 
7. 这个相当于写入数据库完成了，查询数据库如下：
8. 

数据结构层：

command list: \n"

"\n"

"\t""list" "\n"

"\n"

"\t""create dbname" "\n"

"\t""select dbname" "\n"

"\t""delete dbname" "\n"

"\n"

"\t""hget name key" "\n"

"\t""hset name key value" "\n"

"\t""hdel name key" "\n"

"\t""hsetnx name key value" "\n"

"\t""hmultiget name key1 key2 ..." "\n"

"\t""hmultiset name key1 val1 key2 val2 ..." "\n"

"\t""hmultidel name key1 key2 ..." "\n"

"\t""hincrby name key int64value" "\n"

"\t""hincrbyfloat name key doublevalue" "\n"

"\t""hall name" "\n"

"\t""hvals name" "\n"

"\t""hclear name" "\n"

"\t""hsize name" "\n"

"\n"

"\t""zget name key" "\n"

"\t""zset name key score" "\n"

"\t""zdel name key" "\n"

"\t""ztop name min max" "\n"

"\t""zrtop name min max" "\n"

"\t""zclear name" "\n"

"\t""zsize name" "\n"

"\n"

"\t""dump" "\n"

1. 主要数据结构如下：
   1. hset ， hget ， hdel
   2. hmultiset ， hmultiget ， hmultidel
   3. zset ， zget ， zdel ， ztop ， zrtop
   4. 其他比如，自查询命令。
2. hashtableset实现。
   1. 首先hashtable是key , value的方式实现。对于游戏而言的存储，如何将表和字段变换为对应的 key和value在命令层讲解。
   2. 所以set就是简单的key+value。key和value是根据上层传递过来的值。然后将key和value封装为writebatch。写入数据库。
      1. 传递过来的key只是表名 + id，还需要对其编码为数据库能唯一识别的。比如说。user表 +　id ，能唯一确定一个key。一般表示方式为h!user\_1,
      2. 确定了key以后，将key和value先放入一个结果集（OperatorResult）中，放进去的目的是为了给slave发送操作日志。这样如果失败了，就不给slave发，成功了就发。
      3. 在使用Set命令时，会再额外的保存一份这个表的数量。不过会损失一部分性能
   3. 另外会有setNX， SetOW，这两个的区别是NX是不存在才写，存在就不写了，OW是只写，不用Size。
   4. multiSet就相当于同时处理多个Set命令。
   5. Get实现
      1. 将传递过来的Key，encode为数据库能识别的key后。直接查询就可以了。
3. zset实现。
   1. 原理是因为数据库存储是有序的。所以zset作为 有序的方式来存储的时候特别方便。具体实现如下：
      1. 首先还是将传递过来的name和score以及key，encode为一个新的key并保存下来。因为是顺序保存的，所以查询的时候只需要将数据取出来就是顺序的了。
      2. 逆序也需要存一份。
      3. 有时需要获取name和key对应的score，也需要将key和name，encode为一个key，然后保存。保存的目的是为了给新的数据排序的时候要对老数据进行删除。
      4. 然后同步给其他的slave。
      5. zdel以及zget类似。
      6. Ztop和ZRtop，相当于根据传递进来的name + key + 最低score 就可以查询到所有的数据。

命令层：

命令层通过宏定义对应到具体执行的命令：这个很容易实现。

类似如下：

g\_pCommandHandler[CMD\_HSET] = &ServerConnection::HandleHSet;

g\_pCommandHandler[CMD\_HGET] = &ServerConnection::HandleHGet;

g\_pCommandHandler[CMD\_HSETNX] = &ServerConnection::HandleHSetNX;

g\_pCommandHandler[CMD\_HSETOW] = &ServerConnection::HandleHSetOW;

g\_pCommandHandler[CMD\_HDEL] = &ServerConnection::HandleHDel;

g\_pCommandHandler[CMD\_HMULTISET] = &ServerConnection::HandleHMultiSet;

g\_pCommandHandler[CMD\_HMULTIGET] = &ServerConnection::HandleHMultiGet;

g\_pCommandHandler[CMD\_HMULTIDEL] = &ServerConnection::HandleHMultiDel;

g\_pCommandHandler[CMD\_HINCR] = &ServerConnection::HandleHIncr;

g\_pCommandHandler[CMD\_HINCR\_FLOAT] = &ServerConnection::HandleHIncrFloat;

g\_pCommandHandler[CMD\_HKEYS] = &ServerConnection::HandleHKeys;

g\_pCommandHandler[CMD\_HVALS] = &ServerConnection::HandleHVals;

g\_pCommandHandler[CMD\_HKEYVALS] = &ServerConnection::HandleHKeyVals;

g\_pCommandHandler[CMD\_HSCAN] = &ServerConnection::HandleHScan;

g\_pCommandHandler[CMD\_HSIZE] = &ServerConnection::HandleHSize;

g\_pCommandHandler[CMD\_HDROP] = &ServerConnection::HandleHDrop;

g\_pCommandHandler[CMD\_HLIST] = &ServerConnection::HandleHList;

通过一个宏定义对应一个函数。

网络层：

采用多路分离网络



有一个基类reactor他的作用就是用来收集各种网络处理类。并将数据转交给具体的处理类负责来处理具体收到的消息。

ORM：

ORM（Object Relational Mapping），是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换,就是将数据转化为一种数据结构。

说白了，就是将面向对象的数据拼成数据库能识别的sql语句。

ORM设计：



确切的说每一个表都是从OrmBase继承的。操作这些表通过OrmHelper就可以了。而每个表具有的操作函数就是转换为Buffer。然后通过Commit将这个buffer通过网络发送给db。db将Buffer解析出来后保存。

主从数据库

从数据库的作用是作为备份。也就是当主数据库在写数据的时候，顺便也需要向从数据库发送一份。所以实现方式为，从数据库在启动的时候从主数据库拷贝一份数据库的内容，然后再写入数据库的时候也让从数据库执行一遍写操作。（因为是单线程的，所以不用考虑备份的时候又在写的问题。但是必须注意是需要主从都启动成功后，才算成功。）

