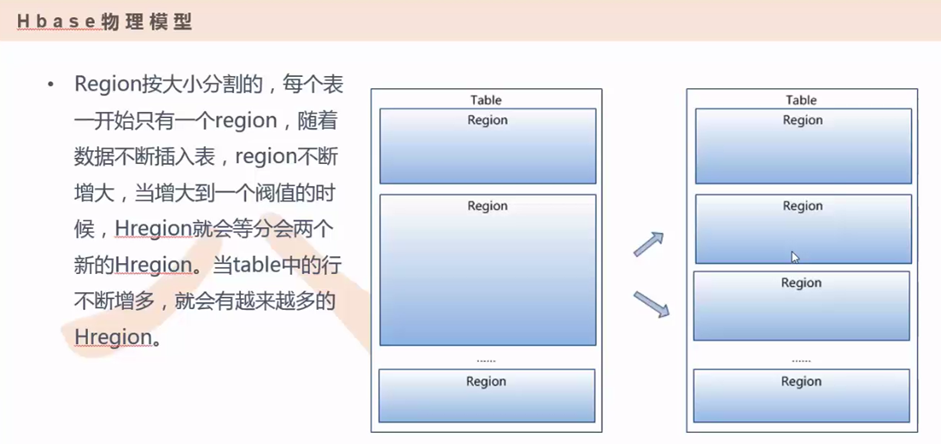
图1：



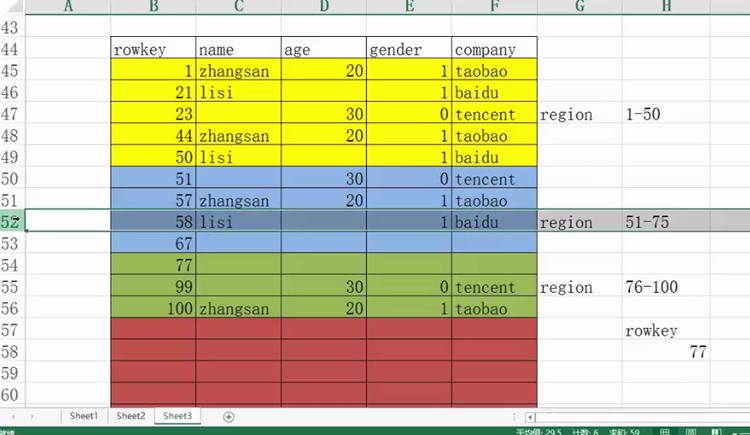
可能大家会有一个疑问，就是什么情况下这个region会自动分裂，就一般来说这个Region他有一个默认配置可以配，如果你不配的话他有一个默认，就是它这个region它是有一个大小的一个数量限制，默认是10G，一个region里面它所管辖的数据里面超过了10个G的话，它会自动的去切分，你这个用户的表越来越大，随着你这个数据的不断的写入，然后这个表不断的增大，那就会导致你这个分裂就越来越多，那越来越多会导致你这个region的个数也会相应的增加，那所以这个region你可以认为是在Hbase整个集群里面是数据分布的一个最小的单位。

一个table它包含了很多的region，但是这些region并不是在一个机器上来完成的，它在hbase里面的最基本单位是一个region，它只是一个数据分布的一个最小单位，那这每一个region它有可能是在就两个不同的region，它可能是在同一个机器上也有可能在不同的机器上就是这么一个关系，那如果随便看一个region的话，那这个region它仅只能是属于某一个机器的，那这个机器的名字叫做regionService，那从这个名字就可以看出这个regionService就是一个类似于像一个节点一样，就一个region它只属于一个regionService，但是不允许一个region它既属于regionService又属于另外一个regionService，这是不允许的，它只能属于一个regionService这个大家要记住。

然后我之前讲那个HDFS的时候，比如说有一个datanode直接挂掉了，那它的block就丢失了，那为了达到它三副本的这么一个要求达到那个平衡状态，怎么做的呢？把当前的这个挂掉的datanode这个节点上的数据会平均分配到其他的datanode上，那么在Hbase里面就是类似，如果一个regionService挂的话，它这里面可以管辖很多的region，那如果这个regionService又挂的话，这三个region就会被其他的regionService来接管。

那这个机器挂了，那么我这上面的数据肯定是读不到的对吧？那么这个数据不就丢了嘛，那怎么能够保证我的数据还能拷贝过去呢？因为我们要对HBase进行操作的时候，通常我们操作也是和我们传统的数据库一样，你比如说你就改其中的某一列或者是某几个列，不可能是把数据全部改变，你要改变可能也是改这么一个字段或者两个字段，那如果是想改某一个字段的话，比如说下图2这个58行的记录（如下图2所示）

图2：



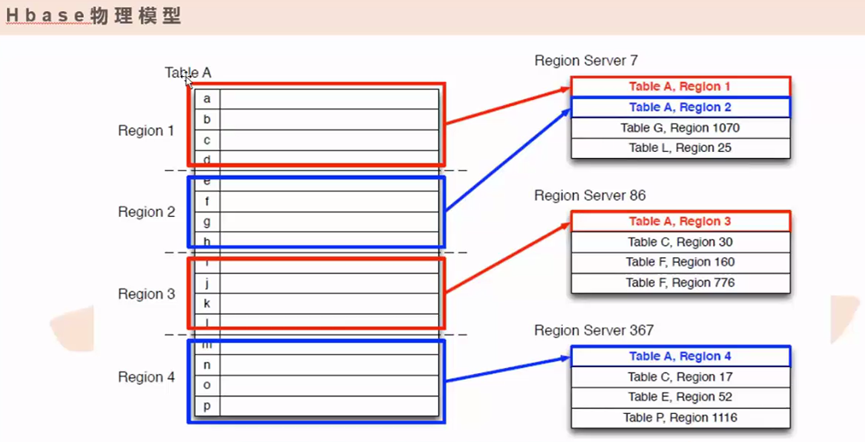
这58行这个记录，如果对于这种又有读又有写的这种情况我们当时是怎么处理来的？

我们必须要有一个锁对吧？必须有一个顺序是吧？在Hbase里面你想对某一行记录要去做修改的时候尽管你只修改里面其中一个字段，其他字段不修改，把整行记录加锁明白这意思把？所以Hbase它是按行锁定的，那么把这58行加锁了之后也就说明由我当时获得锁的这个进程是拥有绝对的主权的对吧？尽管我当时在读数据，我只读这个字段但是这三个字段我不会去碰，那么这个时候有其他进程让你写那也是不允许的明白这意思把？

所以它这个锁的粒度也不是特别的细，行锁列它的粒度就比较粗的，这个大家注意一下，就是对于一个数据表来说这个region是什么意思呢？这个region相当于是一个逻辑概念，然后这个region就是把这个表里面分了很多的区域，然后每一个区域代表一个region，这是一个逻辑概念。

然后每一个region会分配到不同的service上去，每一个service都会管理着一个或者多个region（如下图3所示）

图3：



TableA有四张Region，第一个的这个region可能就属于到这个Service7这个上面去，然后第三个Region就属于到这个Service86集群上去了，这不一样的，所以你这个从用户表格上来看，这个数据好像是前后之间相互连接到一块的，看上去是一整张一张表对不对？但是实际上物理来说其实这些数据都是分布到不同机器上，而且即使同一个机器上也是不同的Region上去的，是这么一个分配情况。

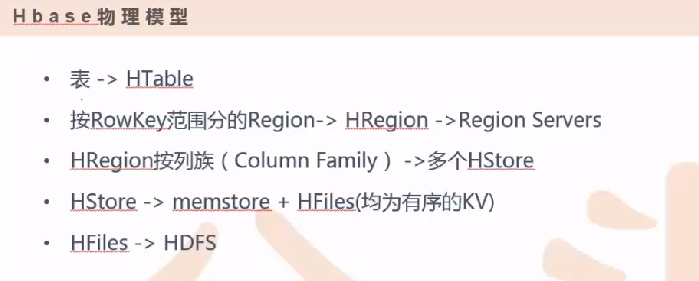
所以有一个Region只能属于一个RegionService，但一个RegionService是可以拥有多个不同的Region的。

那么接下来我们再看一下这个RegionService主要是干嘛的呢？它其实主要是负责用户的IO请求，什么是IO请求，就是读和写对不对？那用户读和写的目的是干嘛？不就是操作数据嘛对不对？那它的数据哪里来的呢？就是在这个HDFS上，这个RegionService就给你提供了这么一个IO的一个入口然后供你客户端来对HDFS进行一个交互，所以RegionService在这上面是一台机器，但是在这个机器上面它会运行着一个进程，这个进程就叫做HRegionServer，这是进程，它这个进程是在Hbase里面是属于一个非常非常核心的一个进程，一个HRegionServer内部管理了一系列的HRegion对象。

然后这个时候大家肯定又有疑问了，我这里面是写的是HRegion，但是这个上图3表里面是一个Region，一个带H一个不带H，这两个是一回事吗？Region是你对应Table的一个Region，然后HRegion相当于是对着Region一个封装，你可以简单理解就是这个Region就是逻辑概念，而HRegion就是一个物理概念。

所以我看如下图4这个表里面

图4：

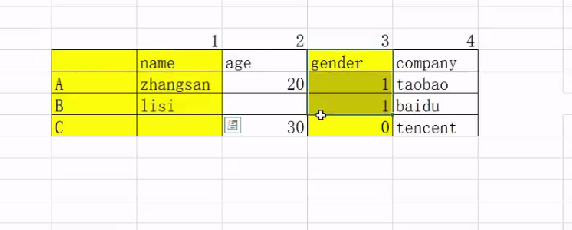


我们通常面对的一个数据表，那么在Hbase里面叫做HTable，那么我们从这个Roekey范围，因为这个region它管理着一个部分的一个Roekey它所对应的这个记录对不对？，那如果按照这个Roekey范围可以分出很多的region的话，其实这个region刚才说了是逻辑概念，然后每一个逻辑的region背后都是有HRegion来去封装起来的。

并且由多个HRegion组成了Region Servers这么一个概念，我们现在目前学的只是学到了HRegion这么个级别，就到了Region 这么一个级别，其实我们在从Region上往下进行细分，然后每一个Region 其实在内部的时候，它可以分多个HStore，这个是一个新的概念。

什么是Hstore呢？之前我们说过了这个Hbase你是可以分多个Column Family的对吧，一个Column Family就是一个Hstore，你可以认为就是一系列的文件，你简单理解就是一个文件，一个列族就是一个文件，好了那这个时候一个HRegion它里面会包含多个HStore，然后你作为一个用户你创建表的时候你可以设置多个Column Family，然后每一个Column Family对应着一个HStore，为什么这么做呢？，这就回到了我们最开始说的列存储的概念了（如下图5所示）

图5：



我们争取能够把这些字段进行有效的隔离，然后就是相当于是尽量的把完全不同类型的字段进行隔离开，用不同的文件去维护起来，怎么通过文件隔离开呢？就是按照列族（Column Family）的方式，一个列族（Column Family）就代表一个文件。

这个Hstore有两部分来组成，有一个内存区（memstore）,还有一个就是(HFiles)真正落入磁盘的数据的一个数据，那这个内存区叫做一个memstore，真正落到磁盘上的叫HFiles。

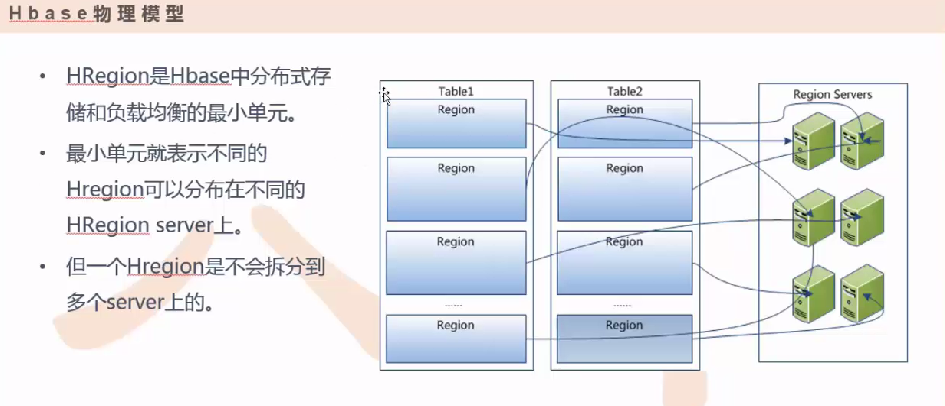
HFiles这个你应该很熟悉了，它其实就是HDFS上的数据，这个memstore是什么意思呢？这个特别像我们之前学mapreduce一样，我们mapreduce这个map读数据的时候，它首先把数据读到了你的内存区域，然后这个内存当它一达到了百分之80这个水位线的时候，它就会自动的锁住然后把这个百分之80的数据进行从内存溢写出来的，然后会生成很多很多的小文件对把？其实呢HStore也是这个逻辑，它首先把你的数据往memstore去写，当写到一定程度的时候，它就会把这个内存里面的数据做一个内存溢并写出多个Hfiles出来。什么叫有序KV？有序KV这个之前我们讲mapreduce的时候，那不是有一个排序嘛？按key进行排序这不就是一个有序kv嘛对吧？

然后这个HFiles相当于就是这个HDFS上的一个真正这个文件系统上存的普通的一个文件，大概是这么一个关系。

那么我们在说一下这个memstore，我们之前讲的那个memstore在mapreduce里面memstore它的默认的空间大小是多大来着？是100m是吧？然后它有一个百分之80的水位线，但是在这个Hbase里面这个memstore它是多大呢？就是128m，但是这个系统默认的128m可以改的，一旦达到了这个128m的时候它就开始往磁盘上存。

好了我们就继续往下（如下图6所示）

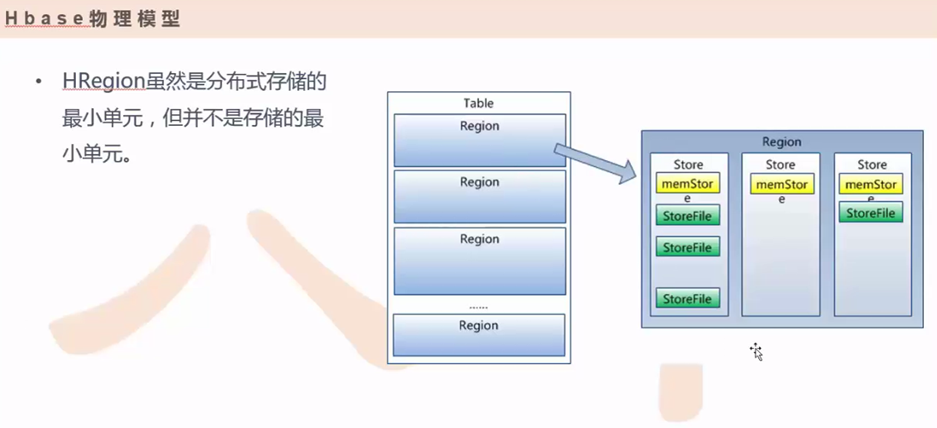
图6：



其实这块其实我们已经有些内容已经说过了，比如图6中有两张table表，这每一张表都是由这个不同的region来组成，然后每一个region就会存到了这个不同的regionService上面去然后相当于是一个类似于负载均衡,相当于是你这个HRegion是Hbase中分布式存储和负载均衡的最小单元，因为它相当于把这个整张表的所有的的数据都跟你分发到了不同的机器上去，那么这个时候你再去写或者读的时候，相当于是你可能需要面对着你来操作很多很多的机器是吧？

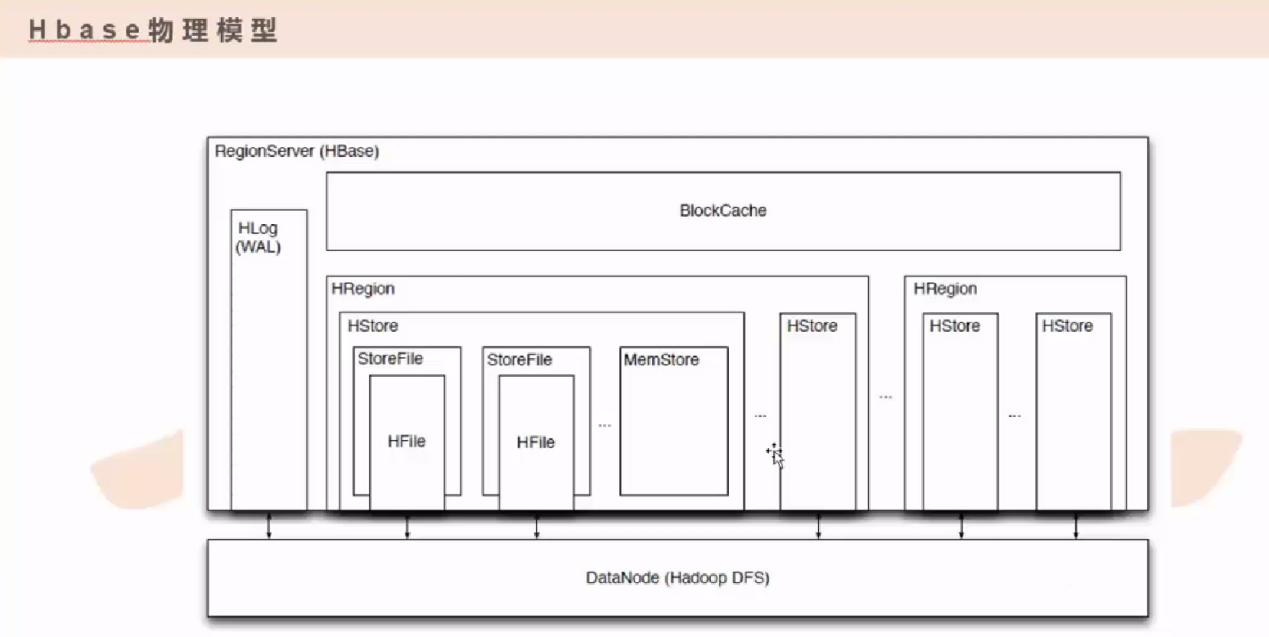
再看这个（如下图7所示）

图7：



然后一个region它包含了很多HStore，这个图7中的最右手边的Store就是HStore，然后HStore里面包含了两个部分，刚刚说了黄色的部分还有一个是绿色的部分，这个黄色的部分是叫menStor，然后绿色部分叫做Storefile，就是写数据的时候先往这个memSto去写，然后随着时间的推移，然后menSto的数据是不断的往外输出然后变成一个一个Storefile，这个Storefile就会越来越多，那么这个Storefile这个文件一多的话肯定是需要一个合并机制，需要定期的对它进行合并，好了那这个时候相当于我们每一个部分都已经拆解开来，那把最后把这几部分进行一个组装然后集中的看一下，是大概分了这么几个角色（如下图8所示）

图8：

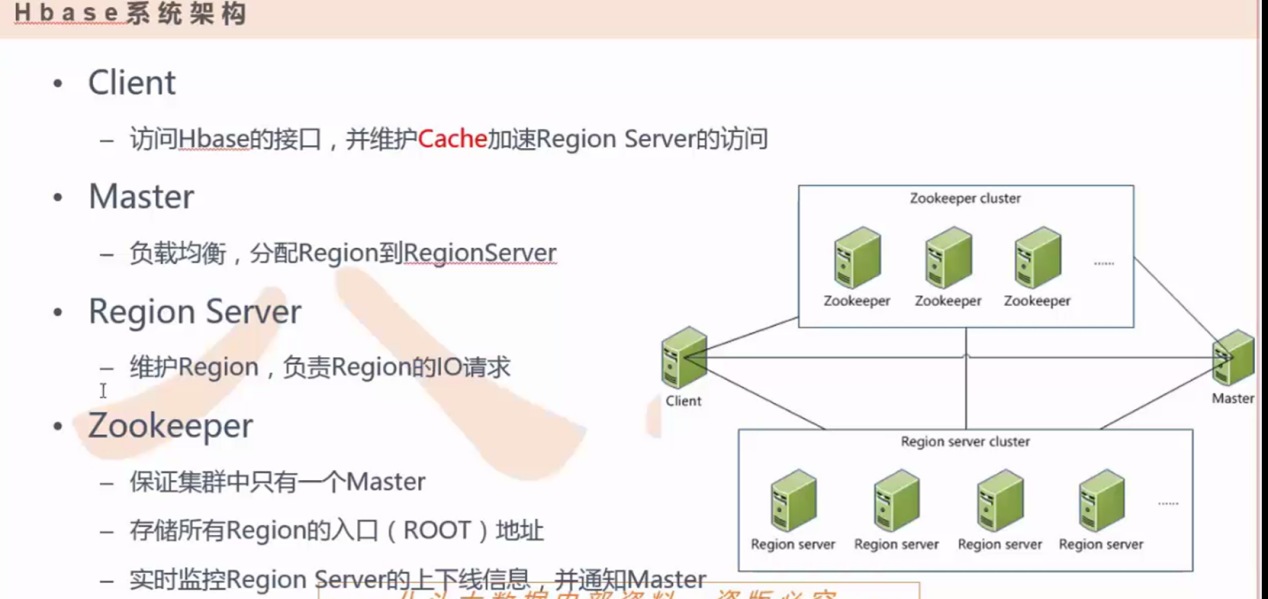


BlockCache是一个也是类似于像menStor一样是一块内存区域，它这个缓存区是用来读的，读缓存，我们刚才讲的那个memstor它是写缓存，当然这个memstor也可以读，memstor是在每一个HStore里面的，这上图8中每一个Hstore里面都有一个Mencache，但是整体来说它是有一个BlockCache，有一个整体的缓存区，为了加快它的读取速度。

所以BlockCache就是一个读缓存，一个RegionService它可以管理着很多个Region是吧？然后这每一个Region里面又包含了很多的Hstore，然后每一个Hstore里面它又包含了一个MemStore又包含着很多的StoreFile，其实StoreFile本质就是一个HFile，HFile就直接落地到了你的DataNode上面去了。

对于一个Hbase来说，或者是对于一个普通的一个集群来说，它主要包含了三个角色，这些我们已经说了很多次了（如下图9所示）

图9：



一个主和从和Client，那么自从有Zookeeper之后，我们暂且把它演练上来是吧？这时候所以Hbase整个系统架构就主要包含了四个部分是吧？那就是Client和Master和Region Server和Zookeeper。

Region Server你可以认为是整个HBase的一个从，就相当于是HDFS上datanode一样，如果把一个Region 当作blcok来看待的话，那你这个Region Server其实就是一个datanode，那我们看一下Client是什么呢？Client就是一个我们客户端，用来访问Hbase的一个接口，其实它又有一个Cache，我们客户端也有一个Cache，你要读的这个表，这个表具体要去哪个机器上去读，这个机器地址是可以在你的Client端这个Cache里面先缓存一份，这样的话就减少了一些网络交互并且加快了你的读取速度。

然后就是Master就是一个负载均衡,然后分配Region到RegionServer，比如说当你这个机器有一个Region Server挂掉了，那这个Region Server上面的这些Region 应该被哪些Region Server来接纳，这都是由Master来决定的。

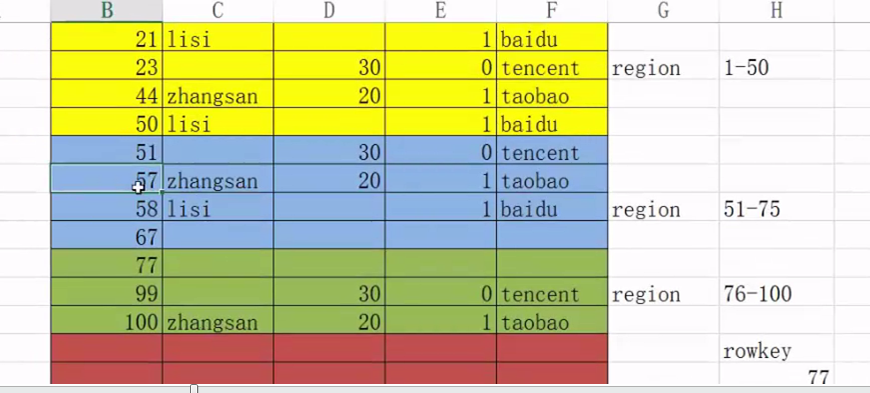
Region Server就是从，在这个Server上面可以维护很多个Region，然后来负责Region的IO，因为你对于客户端来说它要对你的数据进行做请求操作，不管写也好还是读也好，那这都是一些IO请求。

那Zookeeper这个更好理解了，那它就是一个简单的一个协调和服务于分布式应用程序的一个服务，其实在Hbase里面你这个主Master它是不存在一个单点故障的，其实在你这个HBase里面可以启动多个Master，但是真正服务的时候，只有一个Master来服务，那如果你这个Master挂了的话，另外一个master可以马上的来接管进行一个选主，那这里面可以和Storm有一个对应的一个地方，就是说当你的这个Master挂了，那你这个服务就失效了吗？也不是的，像我们之前讲Storm的时候，如果你的主挂了那么大不了就新任务提交不上来，但是你老的任务仍然可以运行，因为它数据都已经存在了Zookeeper上，所以在Hbase一样，它的数据其实都存在你的Zookeeper上，你这个主挂了的话，也不影响你的数据的去读，但是会影响你数据的写还会影响着你的Region的一个分裂，它就没有办法分裂了，如果你这个Region变得特别大的时候，那这个时候master就没有办法给它做一个负载均衡这么一个概念了。HDFS上的数据是真数据，而zookeeper存的数据是原数据。

然后另外一个就是这个Zookeeper它还存储着一个入口（ROOT）地址，但这个ROOT地址因为新的版本就已经把这个取消了，就没有这个Root，但是我们学的时候还是要去学，你可以暂时先认为这个Zookeeper会维护着一个入口，这个入口是干嘛用的呢？就假设说我们之前说过这个Client会维护一个缓存对吧？这个缓冲柱你之前你操作这个表，这个表所在的位置，你可以直接拿来用，这样的话就不用再去找你所操作那tablename它是具体是哪个Server上。

但是如果要是操作一个表，这个表你之前没有操作过，那你也不知道这个表应该去哪个服务器上去读，那么首先你得先干第一件事，要找到这个表所对应的那个服务器，就相当于之前我们做那个访问bolck，去访问namenode类似，你起码得知道这个Bolck存在哪个node上对吧？那么在这里面也一样，其实知道你这个要读的这个表，这个表它存在了哪几个region上，而且你读的这个key，比如你要去读某一个表，这个表里面读某一个key的话呢，你得知道这个key，就可以定位到你这条记录是存在了具体哪一个region上对吧？（比如说相当于下图10）

图10：

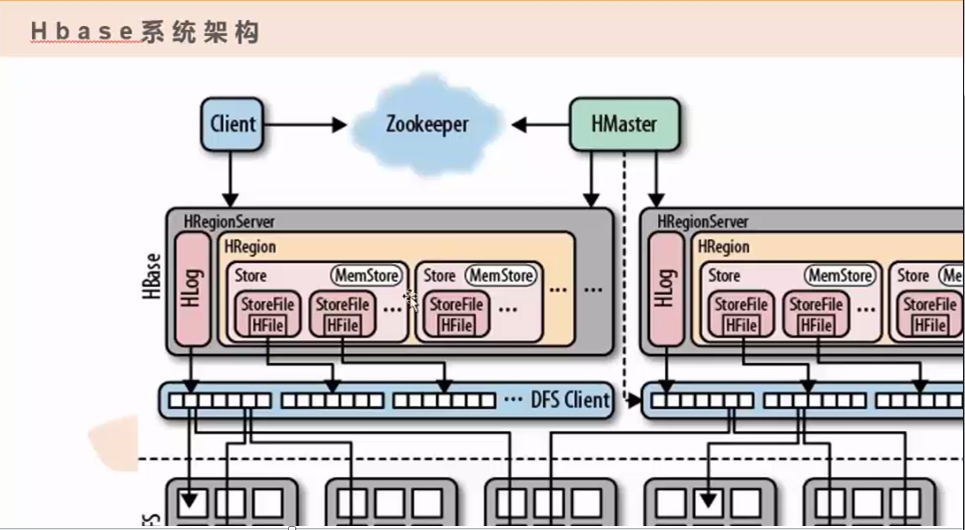


比如57这key一旦确定好，并且这个表格的名字也确定好了，这个region是不是就已经确定拉？这region确定之后，regionServer就已经确定了，所以你的目地就应该找到这个regionserver，找到这个Server之后你才能找到这条数据，所以现在的问题是什么呢?

如果你缓存里面你没有这个Cache这个表，你就要去想办法怎么找到这个regionServer，所以呢找regionServer的第一步就去找Zookeeper上的相应的一个地址，那么这个需要一个寻址，这个我们后面会说，Zookeeper是为了这个寻址，它要去存一个入口数据，所以Zookeeper里面还有这么一个功能，另外一个就是监控，这个监控跟之前都是一样的，通过心跳对吧？这个好理解。

好了然后这个下图11相当于是跟上图8差不多，但是下图11更好看一些更丰富一些

图11：



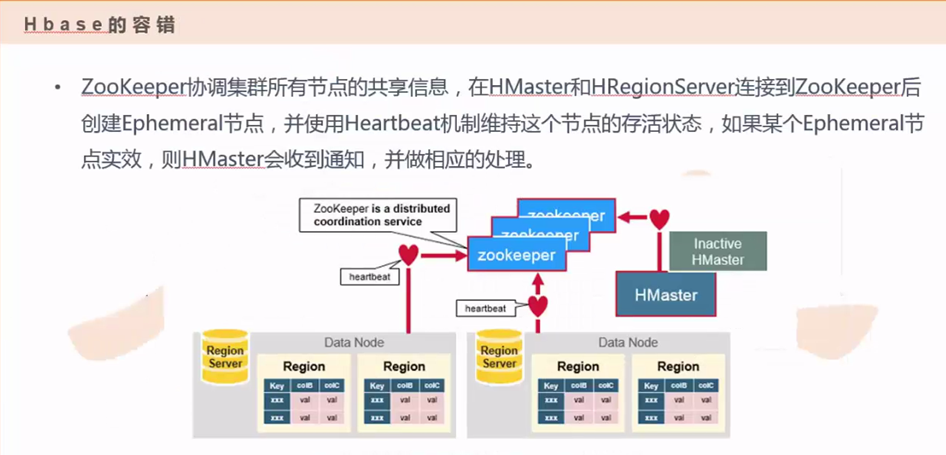
那么这图11里面也是一样有Client和Zookeeper和Hmaster，regionServer就是从。

然后这里面再重复一下，这个Master有什么功能呢？主要有四个功能，第一个是负载均衡，这个主要是为了管理和分配你的Hregion用的，第二个是DDL：就是增删改的操作，就是你可以对table来说你可以去创建一个表，你可以把这个表删除，还有你可以对这个表cf做一个增加和删除都可以，还有一个就是namespace，就是对表的空间来增删改。第三个就是类似于namenode上管理一些元数据，这个元数据是指你的比如table的一些元数据，那么table的数据是什么元数据？就是你的table的结构对吧？那第四个就是ACL权限控制，就是比如说你在一个公司里面有很多的部门，然后每一个部门都可以共享一个Hbase的一个集群服务，但是部门与部门之间还是有一定的权限的一些要求，所以在Hbase里面还是有一些个权限控制这么一个手段在这里面，这是Hmaster这个主。

然后我们对应的我们看一下这个从HRegionServer，这个从有哪些特点呢？第一个很直观的特点就是管理和存放本地的HRegion，就是你一个HReginServer里面它包含了很多的Region对吧？，它是自己的Region它又管理起来对吧？还有一个就是HRegionServer它是提供了一个接口，可以读写HDFS提供IO操作，第三个就是本地化，什么是本地化呢？我们讲MapReduce的时候讲过这个问题，就是你的数据不移动，你的集成框架移动是吧？当时是这么讲的，那为什么集成框架移动呢？是因为尽可能的减少或者避免你的数据的远程的迁移对吧？尽量你的本地就开始把这个数据进行结算了，其实在HBase里面也是有类似这么一个本地化的概念。

就是说你的HRegion的数据尽量和数据所属的DataNode在一块，这样的话有什么好处呢？这样可以加快数据的读写性能，但是这个本地化不能够总是满足和实现，就是这个本地化不能总是满足这么一个本地的一种情况，为什么呢？因为你这个region它是不断的去移动的，为什么会不断的去移动呢？因为你的数据是不断的插入对吧？一旦不断的插入的话你这个Hregion就不断的去分裂，一旦分裂的话这个region就变了，然后这个时候你的本地化就不一定能够保证了对吧？那什么时候你的本地化可以从不能保证到保证的一个过渡呢？就需要合并了是吧？然后再把它重新的回归到你的本地化这么一个特点上来，所以主要是Hmaster和HregionServer这是两个重点，而Client和Zookeeper这个东西就不用多说了，所以接下来我们看一下容错方面（如下图12）

图12：



容错最大的工程其实就是个Zookeeper，这个怎么容错我们之前也已经说过了对不对？就是他可以协调集群中所有节点的一些信息，然后有一些心跳，你的主和从之间通过创建Ephemeral节点，现在再看这个节点的这个属性是什么意思应该知道了把？临时节点有哪些特点对吧？那就是监控和心跳，所以通过这个临时节点的方法进行一个监控容错。

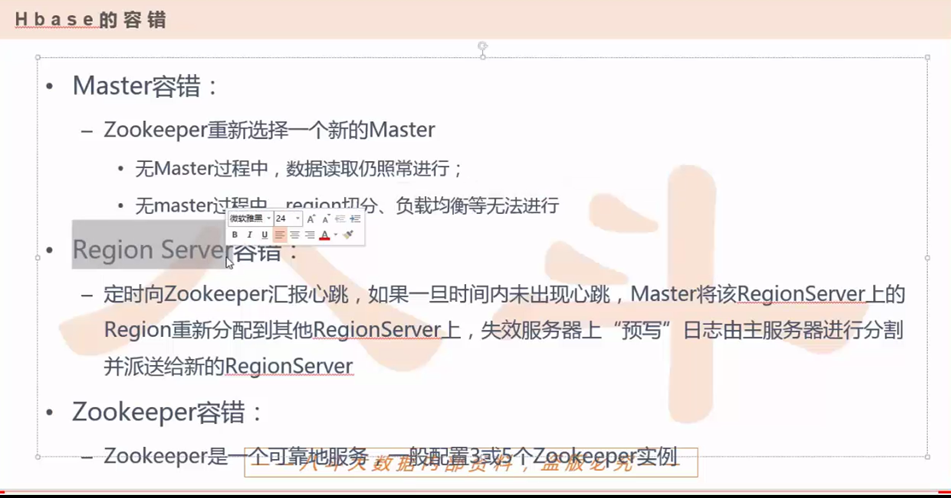
你的Hbase很多那些重要的数据存在了你的Zookeeper上，因为你的zookeeper我们之前也知道了它是类似于文件目录树的结构，然后这个文件目录数它维护了很多个节点，然后每个节点是可以存储一定的数据的，那这每一个节点分支它是一个绝对路径的方式，我们之前都说过了是吧？绝对路径，那么对于每一个绝对路径呢它里面都是存储相应的不同的数据对吧？？那这个是完全可以通过你的Zookeeper去通过那个我们讲的那个客户端的方式你去打开看一下，但是这个节点的数据都是加密的，你打开都是二进制（如下图13所示）

图13：



那这个的话，Master容错刚刚我们已经说过了，就是当你的Master挂了的话，那它不会影响你的数据读取，但是会影响你数据的写入并且会影响你region的分裂，如果你没有master的话你这个region切分，负载均衡就没有办法进行。（如下图14所示）

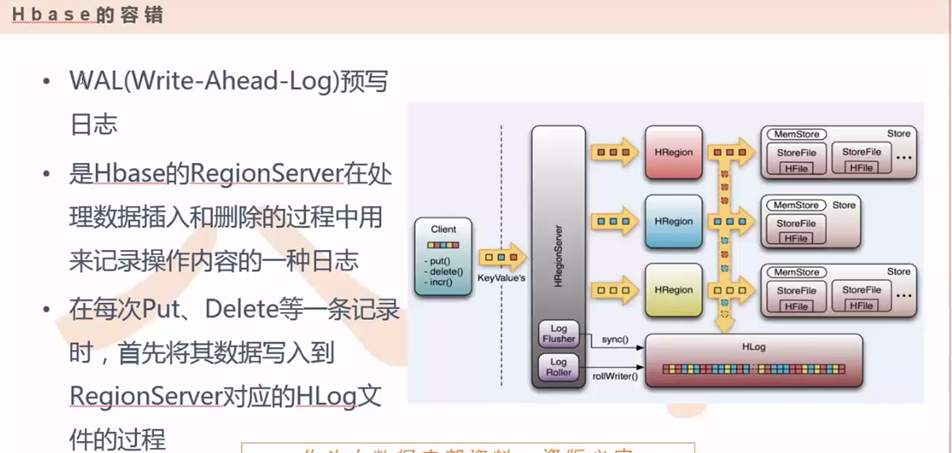
图14：



如果这个regionServer挂了咋办,因为一个健康的regionServer是可以向Zookeeper进行汇报心跳的，通过临时节点的方式进行一个汇报，那一旦超时那就认为你这个regionServer挂了，那这个时候把这个regionServer的任务可以重新分配到其他的regionServer上去，然后具体怎么重新分配，这个就是我们接下来要说的，那zookeeper容错这个东西就没有什么好说的，就认为是一个可靠的服务，一般三到五个实例。

既然我们看一下很重要的一个容错，也就是说预写功能，就是当你的一个RegionServer挂了的话，那怎么把这个已经挂的这个RegionServer这个Region怎么有效的迁移到其他的节点上去，那这个功能就完全需要依赖一个预写功能（如下图15所示）

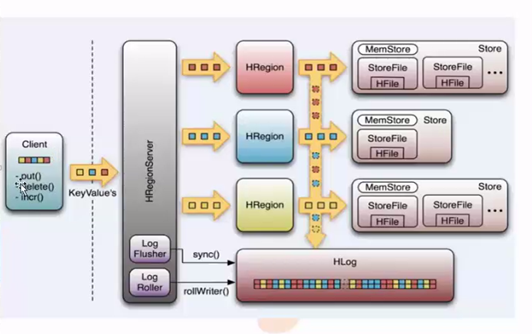
图15：



这个预写叫做WAL，我们作为客户端来说我们去写数据去删数据都是相对数据的改动对吧？那如果要是说我把这些数据的改动所有的不管是数据的本身也好还是数据记录也好，我都完全信任你这个节点用的，那好吧那所有的日志所有操作记录所有的数据都在某一个Region进行的话，这个事情是不是很危险？那如果万一这个Region挂了话，那相当于我之前所有的数据都丢失了都找不着了，那为了要提前有这么一个预警的话，那我应该我再去删除数据修改数据的第一步我要把这个数据写到一个日志里面去，这个日志不会属于任何一个regionserver，这个日志是大家共享的一个数据区域，这个数据在HDFS，就首先你要去对这个数据进行操作的时候，你首先要对这个HDFS做一个操作，你先把这个操作记录要记录到你HDFS上，然后才开始你真正的数据操作，比如在某一个RegionServer上做你真正的一个数据操作，这个时候如果你的RegionServer挂了的话，没关系，你的日志还在，因为你的日志还是在你的HDFS上，这块是不需要自己开发的，这个是完全是框架它已经统一支持的。

那这个WAL是在什么位置呢？（如下图16所示）

图16：



Client里面出现一个put()，什么是put()的？Put()的就是写数据，delete()就是删数据，incr()就是插入数据，当你写数据的时候首先你要通过这个Log的方式然后写到了一个HLog里面去，这个HLog是不属于任何一个hregion，但是每一个regionServer都可以去读这个HLog，都是共享的，每一个RegionServer都可以读这个Hloog。

好了这个时候我每一个RegionServer都可以维护多个HRegion，这上图16上的Hregien用不同的颜色来去表达出来，然后每一个HRegion都可以去读这个Hlog，然后如果你要是没有出现任何错误的情况下你是不需要与这个Hlog进行交互的，如果你正常的情况下你只需要对这个当前这个Region上面对流写数据的话，你就直接写这个HRegion上面的这个相应的HStore里面的MemStore，然后它自己去分一些HFile和HStoreFile之类的，这是正常的机制。

那如果有一个RegionServer挂了的话，那你这些HRegion就该迁移，迁移的话这些MemStore的数据就消失了，但是HLog还存在，那这个时候在其他RegionServer在恢复的时候是用这个HLog来恢复的，这块有点像那个MYSQL是不是？，Mysql是有一个远程备份，这个远程备份并不是把你的数据进行拷贝过去就拉倒了,它相当于是把你的这个Mysql里面操作的每一条sql语句重新执行一遍，相当于这个就是数据恢复起来了。

你像那个mysql里面那个log，它那个log我记得是定期的会优化，比如说你第一步你创建这个表，第二步你把这个表删了，第三步你又把这个表创建了，你觉得第一步和第二步还用做吗？不用做了是吧？它里面肯定会做一些优化的。

好了，那么这块在做数据迁移的时候，它的这种容错是依赖于你HLog这个去保证的，然后当你写数据的时候，你第一步是要写你的日志（如下图17所示）

图17：



那如果是日志写失败的话，它就会立马会告诉你失败，如果这个日志写成功之后，客户端马上会告诉你是数据已经提交成功，然后写失败的话也会告诉你客户端是提交失败，这个时候可能大家会有一个疑问，为什么说只有当我的日志写成功之后，我的客户端会被告诉提交数据成功，这个感觉不太合理对不对？就是说我们在去操作数据的时候，我们只写完一个Log，那就告诉我这个数据成功了，其实这个数据成功了吗？没成功对吧？因为这个数据还没动呢！他先把Log写完了就算成功了，为什么可以这样呢？因为这个数据它在后台可以通过异步的方式去更新，这个有点像我们之前学的那个HDFS的写流程方式，比如三份，只要写一份的话，那就认为是成功了，然后另外两份通过异步的方式去同步。

然后一个RegionServer上所有的Region都共享一个HLog，一次数据的提交先写预写日志，写入成功之后再写memstore，当memstore的值达到一定的阈值，这个阈值就是128m是吧？就会形成一个个StoreFile，就可以理解成一个Hfile的封装，本质上就是一个HFile。

所以对于Hbase的基本操作的话是跟普通数据库操作一样，有写（PUT），有读(GET)，有删除（DELETE），但是这里面写和读都是对某一条记录来说的，那基本操作的时候你要去通过PUT和GET的方式，它只能对一条记录来操作的，如果你想读多条记录，那这只能是通过SCAN命令来操作（如下图18）

图18：

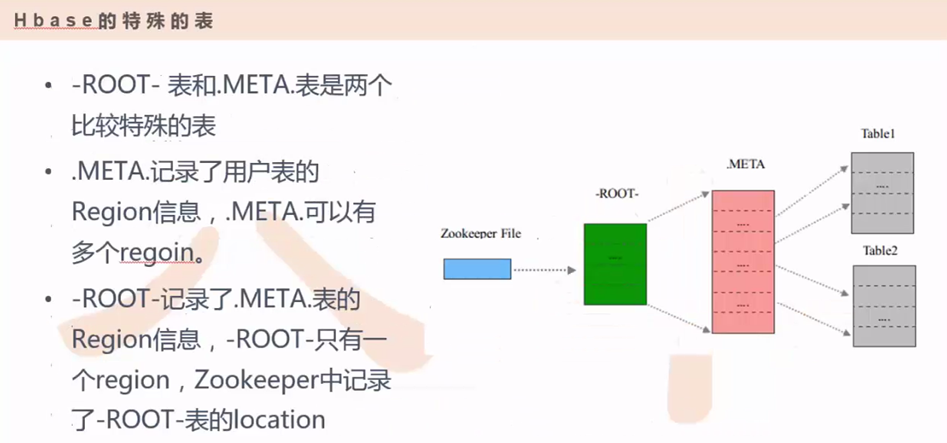


而这个SCAN命令你可以通过，因为它可以支持一些过滤（Filter）条件，然后可以加些时间戳等等很多的一些条件，你可以去检索符合你需求的那些数据，类似于我们像写Sql语句里面where 后面加条件 ，但SCAN命令也可以加些条件做一些个过滤。

好了，那它这个HBase刚才我们说过这个行加锁，它具备了一定的强一致性，那为什么说是强一致性呢？就是说虽然我这个粒度虽然有点粗但是我能保证同一时间内，某一时刻它这个数据不会出现说你写一下我写一下，就是不会出现那种很混乱的那种状态明白了把？

然后这里面会有一些特殊的表（如下图19）

图19：



那这个特殊表这里面主要是包含了有ROOT表和META表，然后在这个上图19中你会发现它要是去做一个映射的话，那它需要分三步，第一步相当于是从Zookeeper里面去拿到这个ROOT表这个地址，这个ROOT表和META表相当于就是一个Region，类似于像Hbase里面一个普通表一样，那既然它是一个普通表那他就是一个Region，那这个Region是存在上图19中的哪个Server上呢？在Zookeeper里面你可以去找到，这Zookeeper有一个节点，节点的数据会存你的ROOT的地址，然后找到这个地址之后，然后这个Root上是存什么东西呢？是存了一些你META这个表的地址，然后再通过ROOT表再去找到META表，这个META表里面存什么东西呢？就是存了你的每一个用户级别的那些表，就是你创建那些，你真正作为一个开发者或者是一个应用层，然后你创建了一些那些表，比如说我们刚才说的这个表（如下图20所示）

图20：



它的一些元数据就存在了META里面来，那什么元数据呢？就是说你要去这个表，这个表它具体是在哪个RegionServer上，那你可以从这个META里面直接查，然后直接查到定位到这个表所在的RegionServer，再找这个RegionServer去读，就是这么一个过程，但是刚才我已经说了在新的版本里面你会发现这个ROOT表感觉有一些存在的没有太多的存在的意义，所以在HBase0.96之后的就把ROOT表去掉了（如下图21所示）

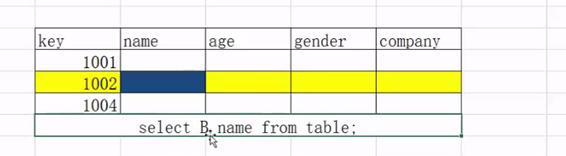
图21：



那么流程就那上图21下面的三个部分，第一个部分从Zookeeper里面直接找到你Meta表的地址，就没有上上图20中的绿色ROOT部分了，直接从Zookeeper直接跨到你这个粉色META的部分，从Zookeeper直接找到你的Meta的存储地址，然后这个Zookeeper这个地址因为这个META表这个数据表它不会做分裂的，META是一张很特殊的表，这个表不会做分裂，它里面只存了用户的元数据，所以这个Zookeeper里面这个数据就把这个META表这个位置缓存了，然后可以通过Zookeeper找到这个META这个地址，这是第一步找到这个META表。

那么第二步就是找到这个META之后你在这个META表上去找你所要访问的那个用户表的元数据，什么是用户表元数据？就等于刚才我们说的这个SELECT这块（如下图22所示）

图22：

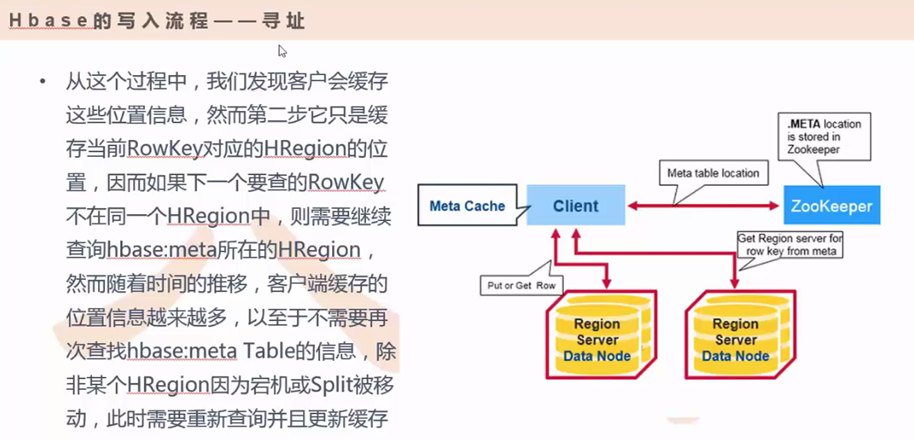


SELECT B name from table 这个table就是你的用户表，但这个表就是你们应用层真正去做项目中你去真正去读的那个表，那个叫应用表，这个表的元数据是在META里面。

那么这个时候你通过第二步查这个META你就可以查到这个应用表里面的数据它具体是在哪个Server上，这个时候你通过第二步映射就可以找到你具体的一个HRegionServer，然后第三步你已经找到这个HRegionServer，接下来你可以根据你这个ROWkey然后去直接定位到你相应的数据，大概是这么一个流程。

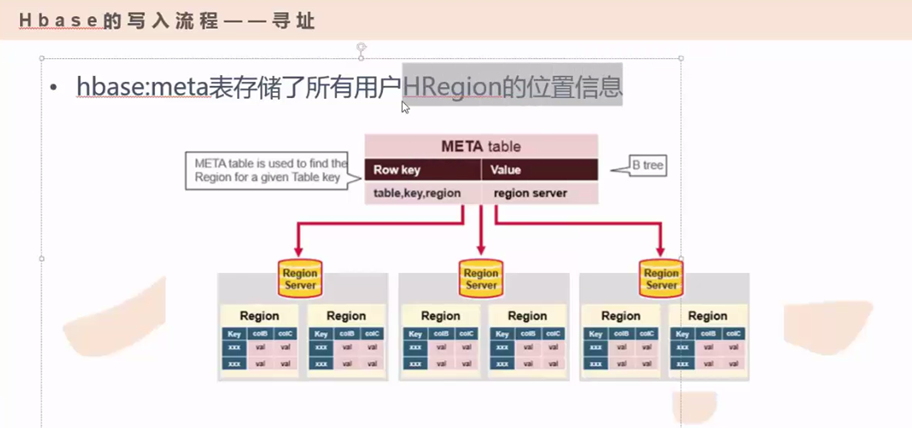
所以刚刚这个流程是什么流程呢？就是一个寻址流程（如下图23所示）

图23：



这个寻址流程过程中，这个META表很重要，META表里面存了很多的这个HRegion，然后随着时间推移，这个客户端缓存的数据就越来越多，所以当你的客户端一旦缓存数据越来越多的话，甚至很多的情况下就不需要再次寻找你的META表这个地址信息了（如下图24所示）

图24：



就是这个META表就是存了你的这个HRegion的一些位置信息，然后这个HRegion位置信息就是你要访问的具体的那个应用表的那个地址。

然后写的时候我们刚才已经说过了，如果作为一个客户端写的时候第一步，先写这个预写日志，然后再写Memstore，Memstore写完之后，那由Memstore往HDFS上去写的时候，这个就完全是它自己去分裂的（如下图25所示）

图25

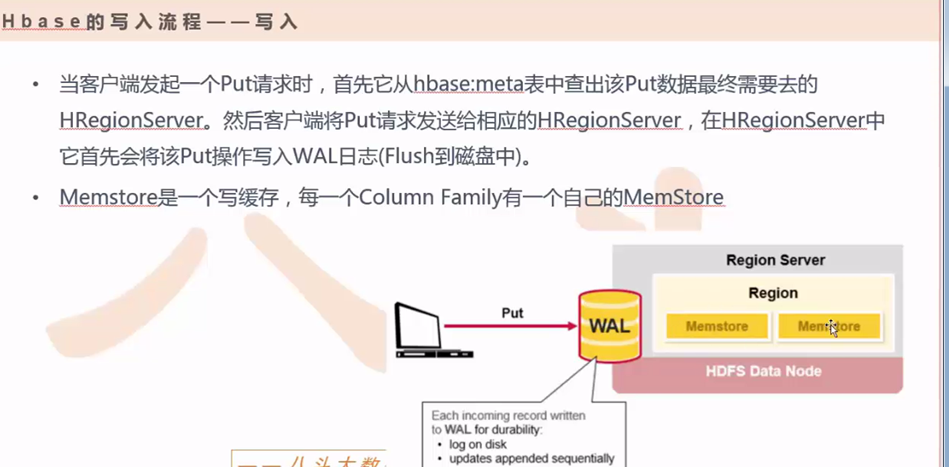
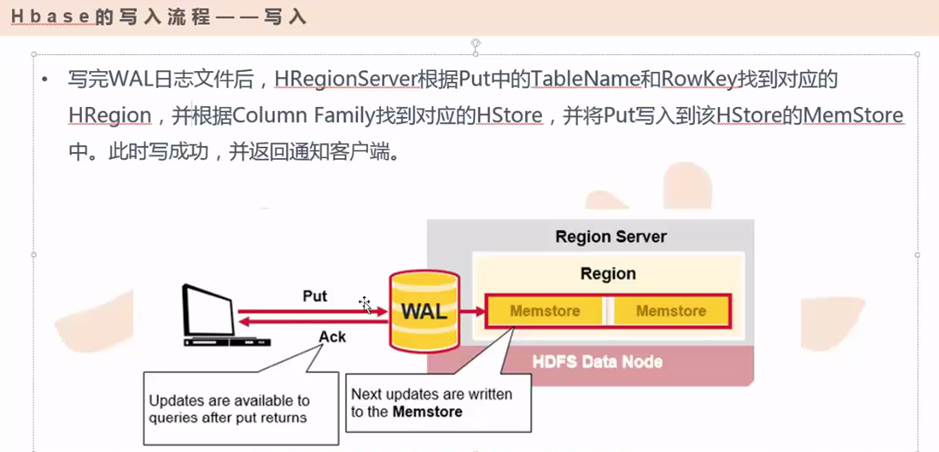
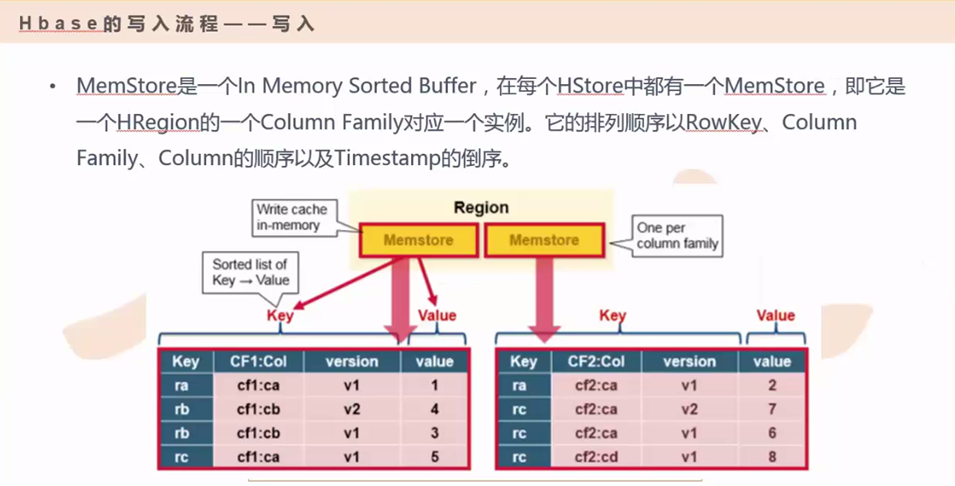


图26



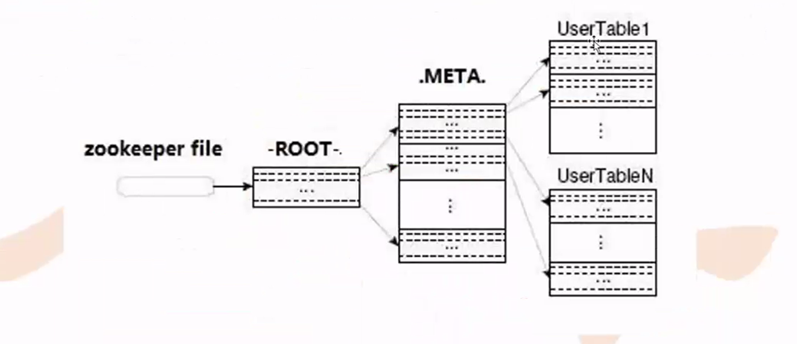
上图26这块的话刚才已经讲过了，当你写的时候一旦写日志写成功之后，然后就立即返回一个成功，但是后面的这个Memstore这块就是异步的过程了，这个过程相当于是刚才一个知识的一个总结（如下图27）

图27：



然后Memstore是一个缓存，是一个Buffer，每一个Hstor都有一个这么一个Buffer ，然后Hstor是什么呢？Hstor是一个Column Family的对应的一个实例，它排序就按我们之前说的那个三位有序的方式（Rowkey，然后列，以及时间戳的）倒序的方式然后排序，这个也是我们上节课说过的，所以（如下图28所示）的这个流程大概是这个样子

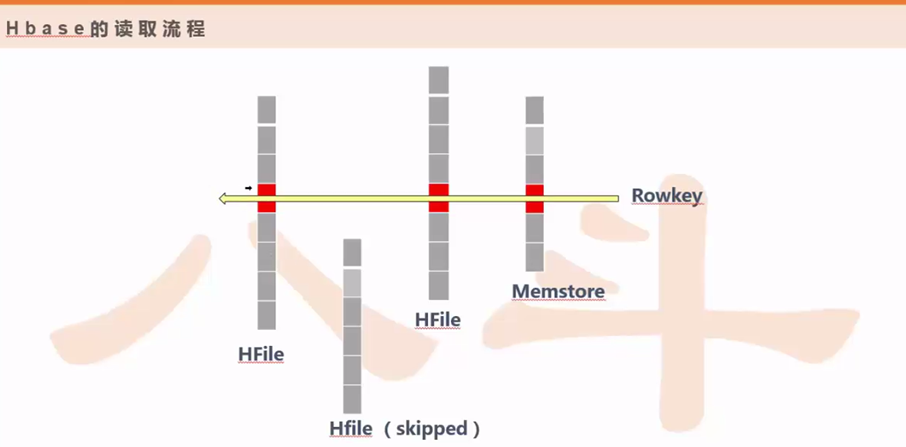
图28：



那么现在把这个ROOT去掉的话，那就相当于是Zookeeper直接到META然后到Usertable这么一个过程，这个应该没有什么太大的问题了把？

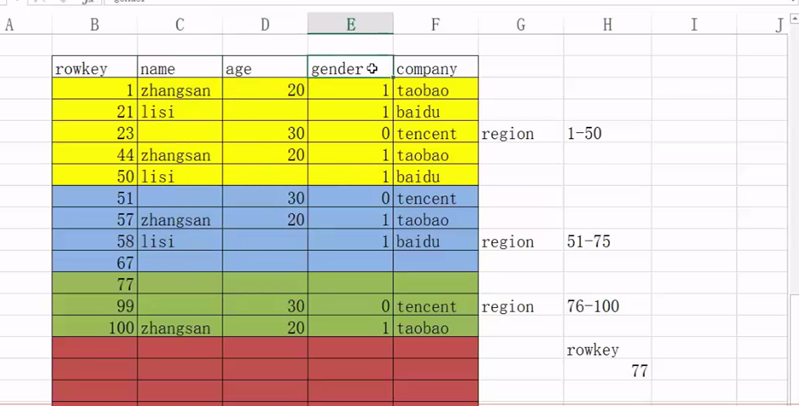
那么（如下图29）这个表是什么意思？

图29：



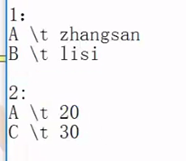
就是说我们在去读一张表的时候，比如说一张表里面分了很多个Column Family，假如说分了四个列（如下图30所示）

图30：



这四个列（从name到company）就代表了四个Column Family，然后每一个Column Family代表了一个文件，那每一个文件代表什么呢？就代表了一个Hfile，然后每一个Hfile里面它都是按照我们之前我们上个笔记有这么一个（如下图31所示）

图31：



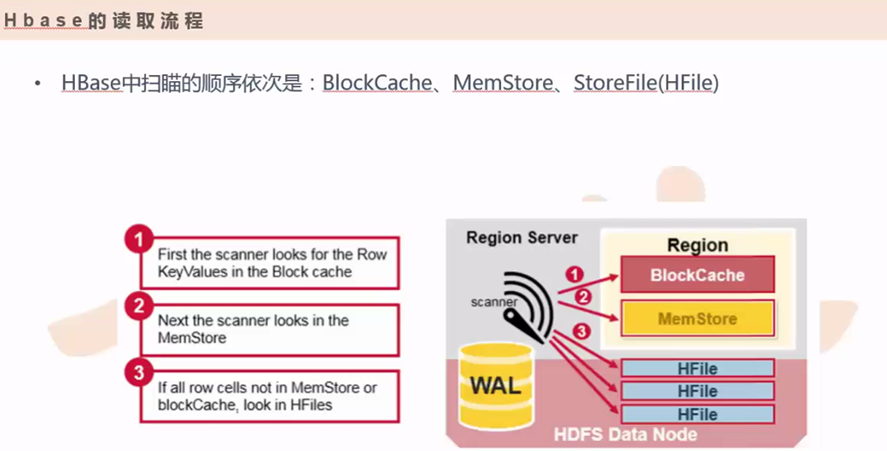
每一个Hfile就像上图31所示的形式去表达的，你第一个Column Family它对应的那个文件你这个key还是你的rowkey，你后面的那个值是代表着你的列的具体的value，第二个仍然是你rowkey，value是代表你这个第二个Column Family 去做应该存的那个value，这样的一个形式。

所以如果要想对某一个rowkey去做一个比如整行的一条读取的话，那对于一个整行读取的话它肯定会动用了多个hfile对不对？它肯定会读多个hfile，然后每一个hfile它的key的分布是不一样，所以上图29红色的区域是代表你的key是可以命中的，然后像整条都没有红色区域的话是不命中的这些区域，因为说明是这个Column Family根本就不存在这个rowkey。

现在把这两个hfile和这个memstore做一个连接，为什么memstore还有数据？因为这个memstore的数据是你存的最新的数据，那你hfile的数据是你落地的数据，那落地的数据这里面可能有一些数据它是还没来得及更新，所以你有些数据还是要去读这个memstore，然后hfile都是要相当于Column Family都是要去读一遍的，大概是这么一个情况。

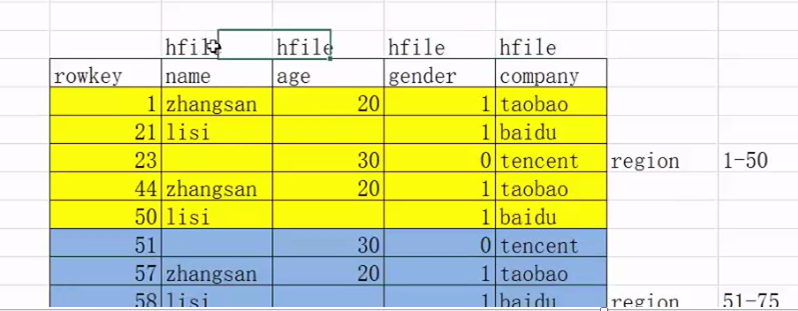
那么读取流程（如下图32所示）

图32：



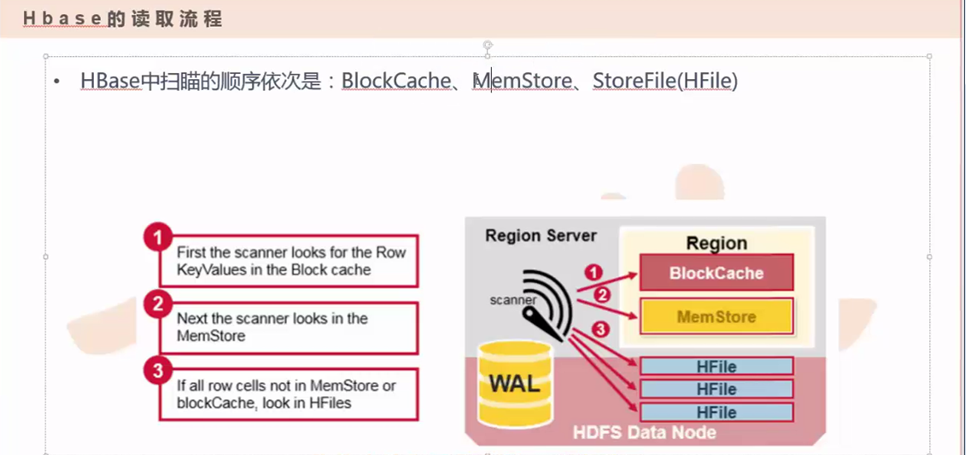
刚才我们说的这个表（如下图33所示）

图33：



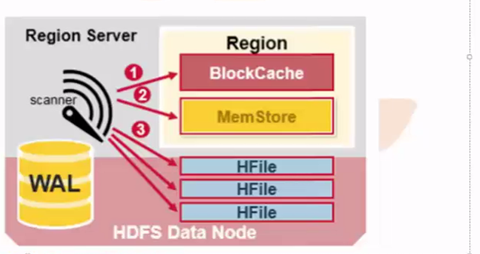
就是我们要对整行去读的话，我们会动用了多个hfile，然后通过对其相应的rowkey然后从多个数据源Hfile去把相应的记录读出来，把相应的values读出来，内存里面也要读，然后读的时候按照一定的顺序，首先要读你的BlockCache，这个BlockCache它是有一个lau这么一个算法，就是说你经常去访问的那些数据频繁去做操作，频繁被你读取那些数据，会被提前的缓存住，如果长时间不用这个数据源，它自然会被淘汰掉（如下图34）

图34：



然后接下来就是MenStore，前面是一个读缓存，一个是写缓存，最后就是实在这个缓存里面实在找不到这个数据，那就去找相应Hfile，这么一个过程，所以下图35里面有个1，2，3，这么三个步骤

图35：



读取的扫描的时候分成三个步骤。

好了那接下来就是一个合并和分裂，那为什么合并为什么分裂？这个我们其实已经讲过了是吧？为什么分裂呢？因为你这个region它不断的写不断的写，那总有一些region它就存的数据比较大，但是相对来说有一些region数据比较小，那一旦你想一下，假设说有一个region它已经规模已经比较大的话，那是不是它读的时候被别人读的时候那它承载的这个IO量是不是也大？那相当于这个请求量就不够均匀了对吧？相当于这个server比如说没有分裂的话，那你这个server上这个hregion，比如一个区它存了将近绝大部分的数据全都在这个region里面，这个region是隶属于一个regionServer，说明这个regionServer这个节点会被访问的频次是特别特别高的，那为了保证你这个负载均衡，所以一定要把这个hregion进行分裂，然后进行一个流量的平坦，这个分裂的话肯定会有很多的region是吧，会分漏出很多的子的region，然后这个时候它还会有一个自动合并的一些机制，就比如说你这个两个比较小的region，你完成可以合并起来，那这个是可以通过手动的方式去完成的。

但是还有另外一层意义上的合并，这层意义上的合并是在你的hstore里面去完成的，就你的hstore里面刚才说过了，就是说你这个storefile是越来越多的，那很有必要把这些storefile进行合并，如果不合并的话会带来什么风险？因为你一个storefile是代表了一个数据，类似于小的文件，那在hdfs上有了那么多的小文件，是不是给你的namenode带来一定的压力对不对？所以hdfs是不适合存小文件的，所以有必要把这些小的storefile就合并，合并成更大的一个storefile，那么目的就是把这些小文件合并成大的，但是有这么一种情况，就是说你前后有两个storefile，那比如说一开始前面那个storefile开始对一条记录做一定的修改。

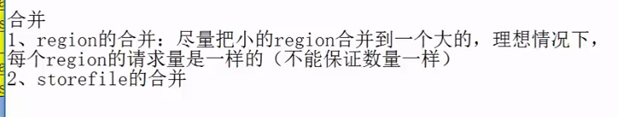
（如下图36所示）

图36：



合并主要有两个，第一个就是region的合并，第二个就是storefile的合并（如下图37所示）

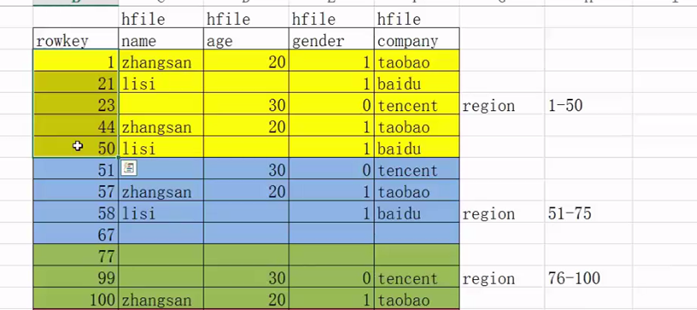
图37：



为什么要region合并呢？因为你总有一些region它存储的这个数据量比较小，合并到一块是不是也会带来一定的优化呢？就相当于尽量的把小的region合并到一个大的，理想情况下，每个region的请求量是一样的，如果你保证你的请求量一样的话，你就不能保证数据量一样。

刚刚不是讲了吗？你这个rowkey，比如从1到50，那可能1到50里面就这几个数，那50到75可能100个数，76到100可能里面有一千个数，每一个表大小都不一样的，要对外进行请求，怎么请求呢？就是被别人读对吧？就是你这个每一个rowkey的范围，你实际的情况你每一个rowkey的范围肯定是不一样的对吧？有的这个region它的rowkey范围可能会比较窄，有的rowkey它的范围可能比较宽明白了把？（如下图38所示）

图38：



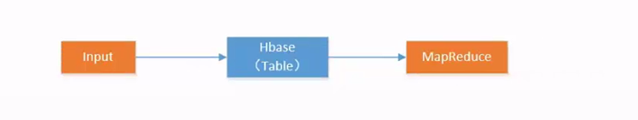
但是不一定说你这个比较宽的这个范围它被别人请求，就一定能够被别人请求，也不能说你这个比较窄的这个范围一定不能被别人请求权，这个大家理解吗？有可能你看上图38中的rowkey这个1到50的范围可能数字可能会比较多，但是没人请求没用，如果你没有人没有被访问过或者根本没有人去访问，那么这个没有任何意义是吧？那假设说你这个77到100比较窄，我就这三条数据，但是大部分人都去访问这三条数据，那它请求量是很大的。

所以它的数据量大小和你的请求量是完全没关系的，但是我们目的是为了保证你的数据量还是保证请求量，如果你是对外进行提供服务的话，肯定是请求量，你跑mapreduce的时候如果hbase当作mapreduce的输入源的话，那就是一个map对应一个region，就相当于我们之前说的我们普通去读那个叫hdfs上的数据，我们一个map对应一个record是吧？竟似的等于一个block对不对？但是如果我们把habse当作一个输入源的话，那一个map就对应了一个region

那好了那一个region代表了其中的一个区域，另外一个region代表另外一个区域，那这时候map是个进程，对应hbase就是一个请求对不对？我的请求量保证均匀的话，那是不是对于我的mapreduce优化是个极大的好处？如果你要是保证你的数据均匀的话，那你这个没有办法保证你的mapreduce是优化的，这个类似于一个控制map个数的方法，就比如说你真要是把这个hbase当作一些mapreduce或者一些个任务的一些调度用的话，那建议大家这个region可能就要设计的更均匀一些。

那怎么能够保证均匀呢？你完全均匀是不太可能的，但是我们可不可以保证一个镜子均匀？怎么保证一个镜子的均匀？你把这个rowkey从小到大你排一遍序，你这个rowkey的数据记录请求了多少，比方说这个1这条数据，这个每条数据背后代表一个音乐，请求量看历史，比如说过去一天或者过去一个星期，比如1这音乐被观看了或者被听了多少次，比如说是100次，那么21这音乐被听了一次，这个23这个音乐被听了33次，因为你每一个rowkey背后不是有一个请求量吗？然后这个时候你把所有的请求量加一块求个和，比如说你求完和之后是100万，这100万你想分5个region，那每一个region的话是分了20万，好了你从头开始数，数到20万是一个region，再数到20万是一个region，你具体是从哪个阶段开始分这个是可以控制，那这个时候你从rowkey从前面开始数，把这个数字加起来，你到哪一个rowkey你加好的话，那相当于你在这块就可以做一个切分了，然后这个时候你就可以按照请求量开始按照均分5份了，每天每周充份一次，这个东西就看业务了，这个切分的区域完全是你人工维护的，怎么人工维护呢？最彻底的办法把表删了重新建一个，建的时候你要明确出每一个这个界限是多少，然后这个时候相当于你充完这个表之后紧接着后面跟一个mapreduce，这个时候它的任务跑起来是非常非常快的，把表删除数据怎么恢复呢？Hbase某一个table（如下图39所示）

图39：



就是我们现在一直围绕hbase这个table进行优化对不对？那刚才我们说已经按照请求量去怎么把这个region切分，这个大家理解了吗？这个时候大家可能会有疑问，如果是你这个hbase删了话，那你这个数据怎办？其实这个业务是这样的，就是说我们这个数据其实根本就不是存在hbase里面的，我们的数据完全是从上游的一个比如说mapreduce或者是一个Spark，你是可以把这个数据进行回溯的，然后这个数据是前面还会有一个log，这个log是在hdfs上的，是在另外一份文件上，不是在你hbase这个文件上，这个Log文件是只读的，然后相当于我要对这个hbase这个表做一个优化怎办呢？我相当于是用mapreduce或者是spark做一个模拟分析，它相当于是跑了mapreduce，做一个模拟分析，分析出我这个表应该怎么建，然后我分析出来之后我对它进行建表，我建完表之后相当于上面这个流程就已经不在存在了（如下图40和41所示产生的变化）

图40：

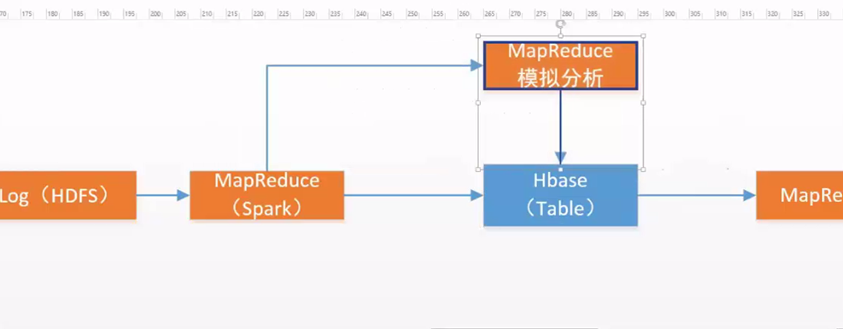
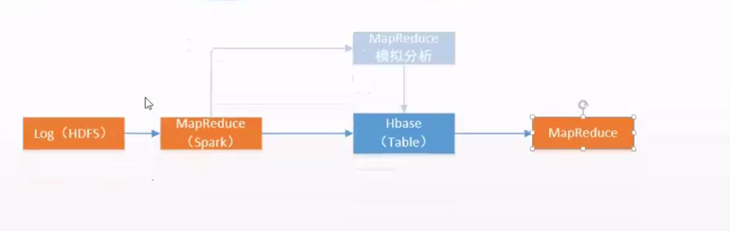


图41：



然后开始真正流程开始走下面这套，然后当你这个hbase表一产生，你前面这个数据一关闭，接下来这个最后的mapreduce就可以激活，你刚才做了这么多的工作的目的就是让你下面这个最后的mapreduce能够运行的快一点，性能好一点，明白了吗？而且你这个库不仅仅是用作mapreduce，你可以后面还有很多其他的一些依赖源，都有可能依赖于这个hbase里面的数据去做分析。

好了我们再回过头来，我们说这个合并有两种一种是region合并，一个是storfile合并，那region合并是为了把小的region合并在一块，理想情况下让每一个region请求量是一样的，但不能保证数据量一样，这个是没有问题的。

接下来要讲的是storefile的合并，storefile合并是针对于一个Hstorfile来说的，好了那刚才说了这个随着你的写不断的增多，然后fluch次数不断的增多，那你这个hfille文件就越来越多，文件越来越多的话需要对这些文件进行合并，那合并有两种一个叫Minor Compaction和一个叫Major Compaction，这两个是什么区别呢?第一个Minor Compaction相当于是比较轻量级一点，然后第二个Major Compaction相当于是比较重量级一点。

轻量级Minor Compaction是怎么样的呢？轻量级Minor Compaction是为了保证你的服务不中断，但是合并不彻底

重量级Major Compaction是为了合并的更彻底，但是服务存在中断风险。

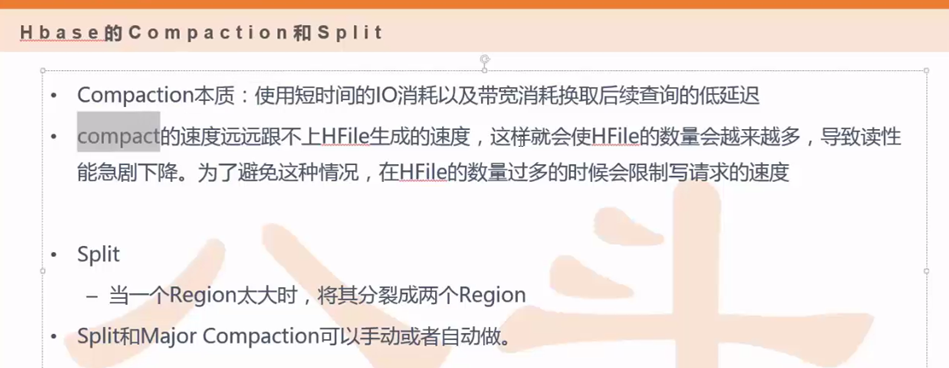
那我们先说不彻底的这个，就是什么时候不彻底呢？不彻底就是说直接把一些小的文件合并成一个大的文件，简单理解，但是这个肯定不这么做，简单理解就相当于是一个 cat a b > c明白了把？把这a和b两个文件直接合并到一块，这个操作是非常快的，但是有一个问题这里面有冗余数据，你比如说a这个记录里面你之前已经操作过一次了，你b里面相当于这个数据你又被操作过一次，那你觉得这个a还有用吗？还有必要保留吗？相当于是这个如果把a和b你合并到一块的话，中间存在了很多的冗余信息明白了把？不会删除那些已经冗余的部分或者不会处理那些已经失效的被删除的那些个数据明白了把？

但是它好处就是说一次这样的合并Minor Compaction直接能把小文件变成大文件，非常直接但是问题是它是不够彻底的，中间存了一些冗余，但是它不会影响你的服务，你的服务仍然正常运转不会影响你的服务。

那么Major Compaction这个是什么意思呢？Major Compaction的话就是它比较彻底，它会把一些已经删除的数据过期的数据，版本号以及超期的数据也会删除，这样的话它合并的时候，这个文件就会彻底一些，但是你会发现做这些操作还是比较麻烦的，然后比较耗时，持续时间比较长，消耗资源很大，所以对于一些上层业务比较有影响的这些比较重要的服务的时候，建议把这个Major Compaction功能关闭掉，就是如果要是说你每天你都要做这么一次Major Compaction合并的话，那相当于每一天到这个时间点，它后台自动合并了，那这个时候你每天到这个时候服务都变成异常了，就是你这个请求的时候都会变得非常非常的慢，建议的是把Major Compaction这个功能直接关闭了，但是Minor Compaction这个的话建议还是自动执行明白了把？

好了我们再看一下这个（如下图42所示）

图42：



合并的本质就是利用短时间的一些io消耗以及带宽消耗换取后面的低延迟，相当于就是一个之前我们经常开发的时候会遇到一个什么以空间换时间对不对？但这里面也不是说空间换时间，但是相当于是用一个短时间内的一个服务的一个就相当于的是我可以允许一个短时间内这个服务可能是不太很好的能够保证，但是我可以换取高峰期就是绝大部分时间的一个正常的访问这么一个情况，有点像那个长痛不如短痛，然后还有一个点就是当你合并的速度远远跟不上你的hfile上的速度，什么意思呢？就是你的写越来越多，但是你合并的速度相对来说就慢了对不对？那怎么办？那这个时候你写你来不及合对吧？怎办呢？为了避免这样的情况就是在你hfile的数据写的时候就限制它一些写的一些速度，就是控制它写的速度，能够保证你的合并能够有效的完成。

还有一个就是分裂，就是当一个region太大的时候就要分裂成两个region，这个split和这个Major Compaction合并可以手动可以自动做。