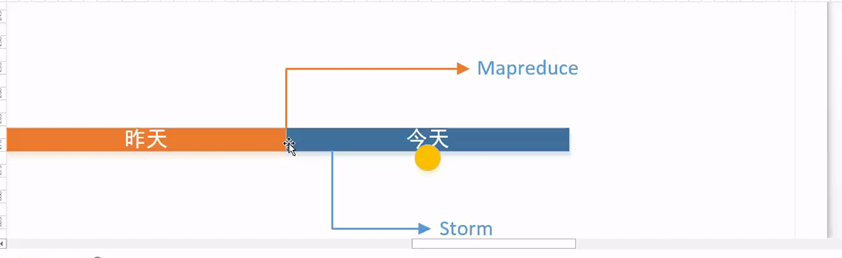
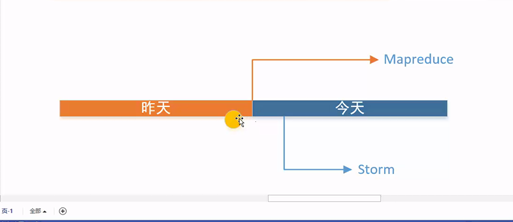
图1：



在讲Strom这一开始，忘记讲一个东西，先补一下

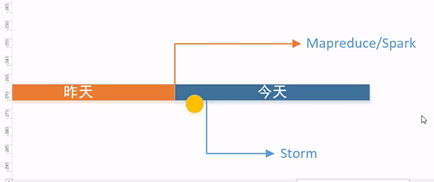
之前不是在讲过MapReduce，然后今天准备学习Storm是不是？那真正结合一个业务场景来说，这两个应该是怎么互相扶持的，因为各自有各自的优点，也有各自跟各自的缺点是不是？那互相取长补短，然后让整个系统就变得健全了是吧？那怎么变健全呢？有一个很好的例子（如上图1所示），当然了这个在公司里面把这个批量作业和增量作业结合到一块也都是这样的场景，所以理解起来很重要，那图1里面已经给大家画好了，橙色的部分代表的是咋天，蓝色的部分代表着今天，中间间隔的这部分你可以认为是半夜12点。

图2：



假设说如上图圆圈到中间左右的部分（如上图2），我们在咋天晚上的时候某一个时刻，因为当天后面时间还没走完，那这一部分的数据肯定是没有的，因为时间还没到呢，不可能预知未来嘛对不对，所以你想把咋天的全量数据从图中那个圆圈那里开始跑的话呢肯定是不可能的，你必须你经过了半夜12点之后才能做这样的事情（如下图3所示）

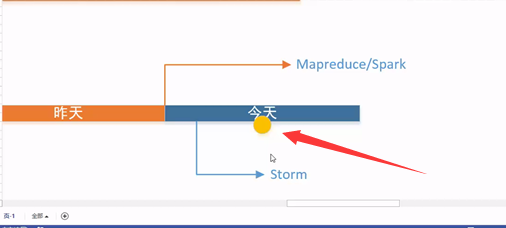
图3：



所以在今天的时候才能去把咋天的积累的数据能够批量的去跑批量的去运行，所以图3中划了一个黄线，当你这个半夜12点的时候你才可以做这样的一个黄色标志线的事情，但是做这样的事情怎么做呢？就用MapReduce去做，当然你除了去用MapReduce呢，还可以用Spark，Spark也可以跑一个批量的，它相当于就是下一代的MapReduce，所以你可以把它当作MapReduce去用没关系，把它定位成MapReduce跑批量作业没什么问题的。

还有一个就是说Storm，到了今天中午12点了（如下图4所示）。

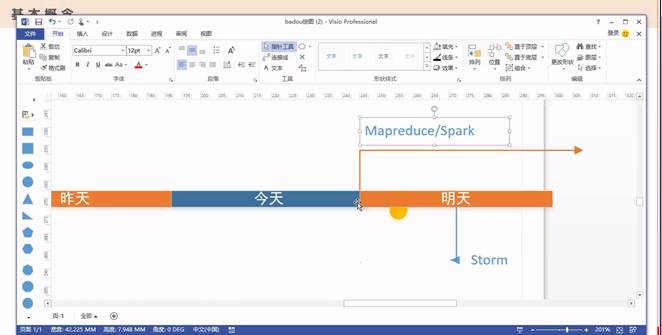
图4：



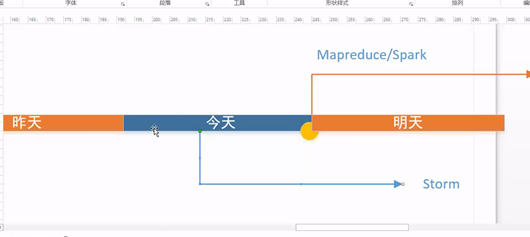
当这个太阳最大的时候，那这个时候我想把今天的数据跑一下，那只能跑这一小短这里的数据是不是？比如你这个时间跑了，你跑完了这个时候可能这个任务跑完之后你可能跑了一个小时，以12点开始跑的任务，等了一点才出结果，那好了这12点到1点之间，这个数据其实你是丢失的对吧？那你通过MapReduce的这种小步快跑的方式也是搞不定的，所以怎么办呢？所以通过流式计算通过Storm，你随着这个时间不断的往后推移，这个数据不断的去融入，然后你通过Storm这个框架就不紧不慢的给你去相当于是打太极一样，就是你来一个数据我就处理一个数据，就是非常柔和这种数据处理方式，然后不论你这个时间点是到了任何的一个阶段，你的积累的数据永远是随着时间变化而且是积累是最全的，那这个时候就是说MapReduces什么时候用，Storm什么时候用，这个大家一定要分清楚。

如果说好了我这个今天过完了，我又过了一个时间那就是明天（如下图5）

图5：



到了明天的时候，相当于是这个阶段跑Storm了，然后还有一个阶段跑MapReduce，把之前的数据可以批量的去跑了，它有的处理的数据，具体这个类型怎么说呢像MapReduce处理的数据，数据源可能是不一样，因为你这个MapReduce要批量处理的数据要求是你起码你的数据是要放在HDFS上面，那你这个Storm的数据，那你放HDFS上就不太合适了，你毕竟这个数据落地了嘛，那通常的Storm对接的方式都是从一个消息队列里面进来这是一方面，另外一方面就是可以从数据库里面去读，但是数据库里面你可以去从一个nosql或者是去hbase里面去读都可以，然后这MapReduce通常是从hbase里面或者是从HDFS都可以，所以呢这策略不一样，这有可能大家就有疑问了，当我这个时间到（如下图6圆圈那里）的时候

图6：  


Storm已经把里面的数据都已经跑完了，那你这时候MapReduce你在今天和明天的这个中间时间段再去跑一遍是不是相当于把数据重复劳动了对不对？但是为什么建议大家还是要重复劳动一次呢？为什么这么做呢？其实还是有差异的，你想一想Storm它之所以能够保证实时数据的处理，那必然呢在他内部实现的时候，要避免一些耗时的操作,要求它的算法，数据处理比较简单，你不能太搞一些复杂的逻辑在里面。

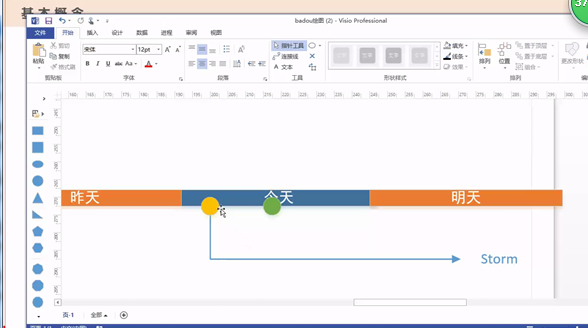
你像Mapreduce就不一样了，MapReduce跑去呗是吧？一个小时跑完我等你一个小时，你两个小时跑完我给你等是不是？那你可以跑的慢一点，但是起码你的数据跑的准一点，通常MapReduce跑得任务是算法比较复杂一点，然后你可以去叠加更多的策略，这个时候MapReduce跑输出数据是更加的准确，那Storm这一块它仅仅是为了快速然后呢尽量的能够把数据的结果能贴近于真实，但是分场景把，像一般复杂场景它这个只能是尽量的去把算法做的简单，然后尽量的逼近一些真实的结果。

然后假设说举个例子，就是说那个我现在的这个策略是比较简单的，我不管你用MapReduce还是Storm，我实现的策略是完全一样的，就是说我实现这个算法都简单，就是你用Storm去算的结果跟我MapReduce算的结果都是一样的，那这个时候没关系，这个时候你就可以用Storm了，从头跑到尾这都没有关系，这个时候就是你不需要你的MapReduce和Sprak这是可以的。这个Storm处理的数据肯定是最全的对不对，如果要是说你中间要有一些数据落地，比如说要以这种天为单位，以星期为单位，以月为单位的窗口去做定量的基准的话最好还是用MapReduce去做一个，你可以把它想象成备份也好，还是做一个保险也好，最好还是有一个MapReduce来每天做一个保证比较好一些，为什么呢？因为Storm这个东西毕竟从稳定性来讲肯定是没有MapReduce稳定的。

再举个例子，这些个例子在实际工作中都是一些极限，就是说你这个Storm之所以能够健康的运行的很重要的因素是完全依赖于你的数据源，你的数据源就是你这个开水龙头，这水龙头开的越大，你这个数据流就越大，数据流越大就对于Storm集群会有一种挑战对不对？你能不能扛得住这么大的压力？我一秒给你一万个并发和一秒给你一百个并发这个量级完全是不一样的，我一秒钟给你一百个并发往这个集群去处理，一秒一百个，这个是没什么压力，但是突然给你来了一个峰值，就相当于那个心跳一样，一开始是很平稳的是不是？突然某一个时刻直线上升上去了，就好像那个炒股一样，突然来个涨停，这时候这个Storm就突然就扛不住了，一旦一开始扛不住，那他这个数据量突然一增多了，那后面即使你这个数据源这个流量恢复正常那这个Storm也会存在一定的隐患，它需要处理前面这样的大量处理，只能去会导致后面的数据会有拥堵，因为你前面的数据你必须要按照顺序去处理对吧？

好了，这时候会有什么问题呢？举个例子（如下图7所示）

图7：



比如说Storm这个黄色这一圆圈突然就感觉这个心率不正常了，突然跳动加快，然后这个时候Storm集群就出问题了，出问题之后相当于这个时候从黄色这一圆圈往后的数据就处理的没那么快了没那么实时了对不对？这个时候怎办，那就是报警，报警完了呢相应的负责人只能把这个集群重启一下是不是？你重启一下其实没关系，因为这Storm本身就不存储数据的，但是它可以存数据，这个我们后面会讲，你可以认为暂时不存储，暂时不存储那我们重启一下呗，比如说你在这个绿色节点重启了，如果你在绿色这个圆圈重启了，相当于你重启完之后，你这个数据流或者是在你原来，就是在你重启之前你这队列里面不是有很多数据吗？但是数据一直在阻塞的状态下，那你重启了这个数据丢失了，那相当于这个时间段的数据就完全属于那种失忆的状态对不对？就相当于Storm你出了问题到你问题修复这段时间的数据有可能存在丢失，这个时候怎办呢？让这个数据进行回数。

假设说你Storm这个模块它的输入源是一个消息队列，假设说这个消息队列压根就，就是我把这个消息给你了，如果你在跟我要，我说没有，就是这我已经都给你了，相当于我不给你做缓存，你再找我要，我是没有这个数据交给你了，但是你没处理完那是你的事情是不是?

那这个时候Storm对于这种情况下数据丢了就丢了，它没有办法追回了明白了把？所以这是一个。

另外一个就是怎么解决这个问题呢？一般有两种方案。

第一种方案就是你起码这个消息队列能够帮我做个缓冲，就是我没处理完的消息，你这个消息就不要再发给我了，等我什么时候恢复的话我就再接着往下去读对不对？这个是一个方案

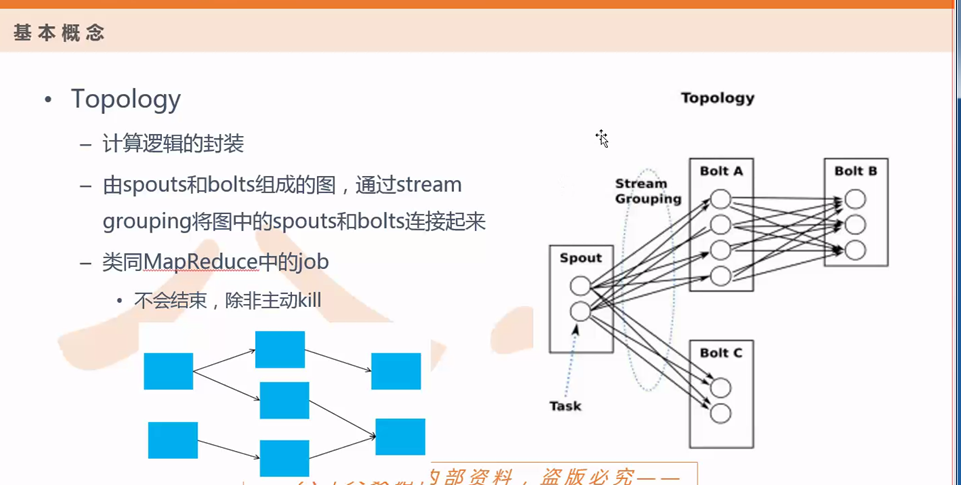
第二种方案就是借助一些存储，借助什么存储呢？有一个很好的存储，但是这个我们还没有讲，但是这个很重要，我们后面会讲，这个就是叫做Hbase，现在大家可能不太理解Hbase这个数据库，这个数据库有一个很好的特点，就是你去写它数据的时候，它自然而然会对每一条记录的后面追加一个时间戳，就是有一个版本号，其实版本号就是时间戳，相当于每条记录他都有一个时间戳，那既然Hbase存的是数据它是丢失不了的吗？你可以认为Hbase的数据就存在HDFS的，但是内存会分，这个后面就会说，但是你可以认为这个Hbase就是给你固化了，那好了这个时候你这个Storm完全是可以去这个Hbase数据里面去读取，那假设说Storm有一块地方断片了，没关系，Hbase数据仍然有，你可以把之前这个断片的这段时间你再回数一遍对吧？你这个时间段是有的，什么时候出问题你是知道的，什么时候恢复的你知道的，那我用这个时间戳用这个版本两边做一个条件判断，我中间有哪些个数据没有被处理那很快就读出来了是不是？就开始做一个数据回数工作，一直到回数跟到当前的这个时间的步骤节奏为止，然后就开始按部就班的往下接着进行，这个大家理解了吗？这个相当于避免数据丢失的一个方式。

那感觉这个流程你说简单也简单你说复杂也复杂是不是？所以说如果要是说你通过这种方式仍然不稳定那就建议大家就离线补，那什么叫离线补呢？就是刚才大家说的通过MapReduce你每隔一段时间去把数据补一遍对吧？离线补其实呢你可能就当天数据会受到一定的影响，但是第二天的数据肯定是帮你补回来的对不对？因为在正常的一个工作的逻辑里面，因为你这个Storm处理的这个数据它毕竟不是像MapReduce那么准，所以它仅仅是一个逼近的一个状态，所以这个就要看业务，有的业务需要你这个数据丢了就丢了，我可以允许你丢一些数据，但是有的业务是不允许的，所以跟你们这个业务的这种优先级或者是对数据渴望的程度，你的这个涉及的方案就有很多种了明白了把？这个就没有什么固定的一个方式了。

好了刚才讲了这么多关于图7的应用结合点大家都理解了吗？这个理解的话呢，那么接下来我们对项目的理解就更加的容易了。

我们回过头来再继续我们的内容（如下图8）

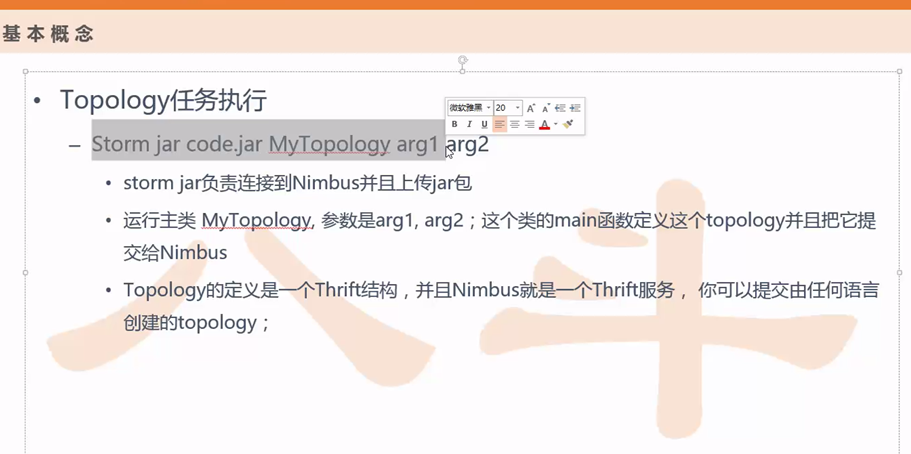
图8：



刚刚讲了一个Topology，这个Topology拓补就是一个网络结构是吧？那相当于你去提交任务的时候你就相当于把这个网络结构提交给你的Storm集群，相当于你提交的任务就是一个网络结构，那怎么提交呢？这个跟我们之前通过 streaming的方式去提交一个MapReduce任务很相似，当时我们提交了MapReduce前不是一个Hadoop jar 然后跟着一个streaming.jar后面就写了一堆东西，现在就改了一下，改了就是前面变成了个Storm jar，后面是code.jar是什么东西呢？如果你是用java开发的话呢，code.jar这个东西就是一个架包，这里面包含了一些class的一些实现明白了把？那MyTopology，因为code.jar里面有很多个类，那MyTopology是要具体执行哪一个类，然后最后那两个arg2 arg2你可以选择，这是是参数，像有一些参数你可以在类里面去写死，但是有一些参数你可以动态的去从外界进行传入，那通过外界传入的方式更麻烦一些，因为你要是发现这个参数要改的话呢，你比如在内部写死，你只能重新再产生一个架包然后再上传，这个整个过程还比较麻烦，相当于你只改了一个参数你把整个的流程都做了一遍这个是变成开发效率太低了，所以MyTopology更通用一点。

那这个Storm jar code.jar MyTopology arg1 arg2命令也是有了之前MapReduce开发经验，所以也比较容易了(如下图9所示)

图9：



Storm jar 负责连接Nimbus并且上传jar包，那Nimbus是什么意思呢？这里面有一个新的概念，Nimbus就可以认为是一个MapReduce里面JobTracker就是主节点，就是这个Storm是一个集群嘛，这个Storm集群里面也有主也有从，那Nimbus就是它的主，这个后面还会继续说的，刚刚接触这样的概念可能大家还不太熟悉。

然后Topology定义了一个Thrift结构，所以Nimbus这个主它也是一个Thrift服务，就是与这个Thrift进行一个交互，你可以用任何语言去开发，对语言来说是透明的。然后这个Topology怎么提交已经说过了，然后刚才图8整个的一个大的结构。

那接下来我们深入到一些细节来看（如下图10所示）

图10：



那第一个很重要的一个细节就是说输入源数据源叫做Spout，叫消息来源，消息生产者，那消息分可靠性消息和不可靠消息两者。

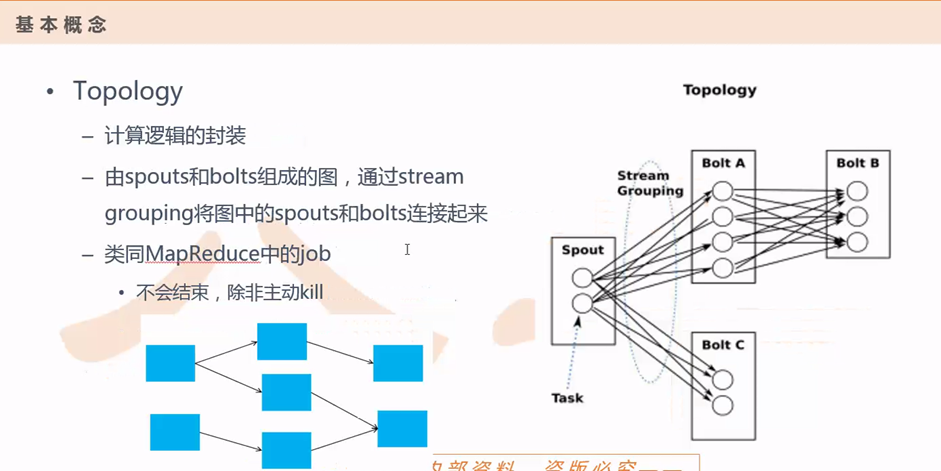
那什么是可靠消息呢？刚才我们已经举了个例子了是不是？就是老板让你干活，然后你及时的汇报这个工作的进展，这叫做可靠性消息。

那不可靠消息就是说老板让你做事情，不管你是做完不做完，就是你压根就不会反馈给你的老板，那这个就是不可靠消息。

那可靠消息即使你这个数据没有被成功的处理，那你也要反馈回来一个失败，然后这个时候你的主就是Nimbus是可以监控成功和失败的，如果要是监控到的是失败的话呢，那可以重新把刚才的消息重新发送一遍，当然这个重新发送这个过程需要大家开发明白了把？这个可靠性需要大家做一些个人为介入，但这个框架是帮你提供了是告诉你成功还是失败，但是成功了怎么做失败了以后怎么做，那这个是需要我们开发解决的，它不会说自动说这个消息失败了就是这个框架就自然的帮你托管重新发送，这不会的，这个需要你自己去开发这个大家注意一下。

同时Spout我们重新看一下下面这张图的过程(如下图11所示)

图11：

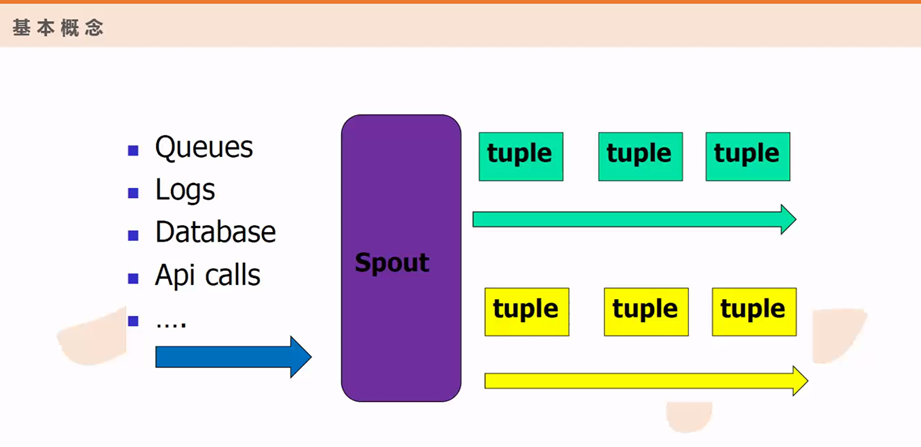


如上图11所示，这个Spout输出可以输出一条流处理，一条流就是一根线，输出到后面这个环节，那你可以生成出多根线出来，相当于是往两个不同角色的Boit集合里面去发送是不是？，那这个时候相当于这个Spout可以发送多个Stream流是不是？那具体怎么发送多个流，这个具体到时候开发代码的时候会说。

然后有一个代码叫nextTuple，这个是一个函数的名字，这个暂时记不住没关系，我们开发的时候自然而然就记住了，就是说你开发代码的时候就主要的工作就是围绕这个nextTuple去开发。

然后接下来就是我们看一下怎么用图去表达（如下图12所示）

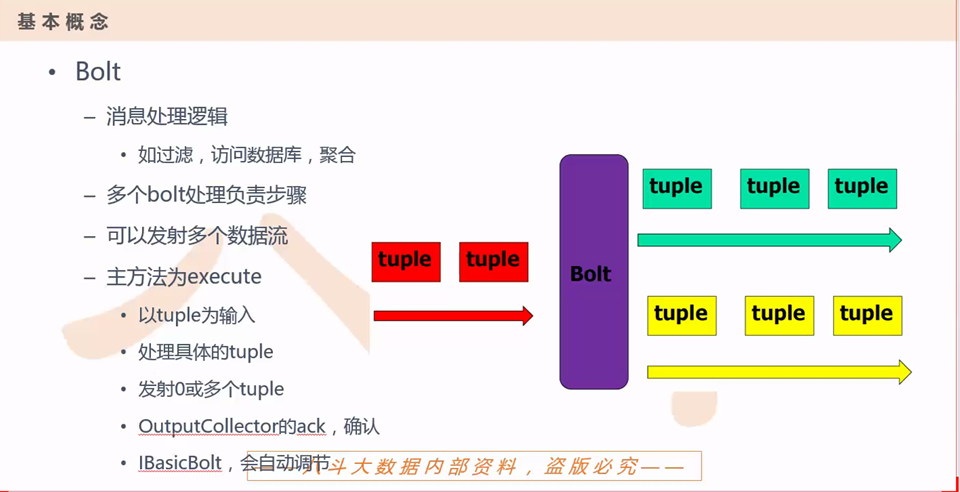
图12



这个Spout输入也有很多个类型，可以通过队列，也可以通过数据库，也可以通过日志，也可以通过API调用等等，然后就相当于传输到这个Spout里面来，那Spout可以输出单条流也可以输出多条流，那不管是单条流还是多条流，这个每条流上面的数据都用他里面封装好的。

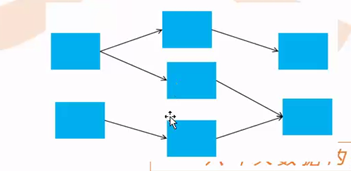
Bolt是中间或者是结束环节，它不可能是中间环节，开始环节永远是Spout，中间的环节和结束环节都是Bolt(如下图13)

图13：



然后这个Bolt输入的时候，你当然也可以是一条流或者多条流都可以输入，都可以进行一个网络，其实这个网络你是可以通过一些很简单的一些语句去把这个图描绘出来（如下图14所示）这是很简单的

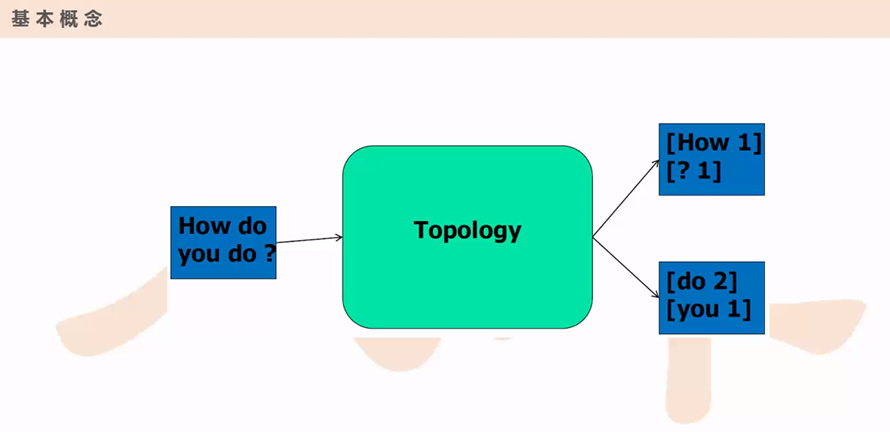
图14：



然后那个Bolt接入了流之后，然后内部做一些处理，你也可以输出单条流多条流（如图13所示），就是说在这个整个的网络的流转里面所有的数据都是tuple为单位进行流转的，这个要记住，然后对于Bolt的开发就集中在一个叫execute这么一个函数里面，这个函数开发的时候大家也会熟悉，有一个特点就是在execute的开发里面它是主要开发你的数据的处理，你的数据处理通常你需要去做一些个判断，你比如说你来的数据不是所有的数据都能够满足你的要求,比如说你前端传这个数据，这个数据可能是过长了过大了，有的数据可能缺少一些关键信息,有的数据可能格式不对等等有这样的一些原因，一旦有这样的原因的话你可能这个数据就是直接给它过滤掉，当然还有一些情况，比如说这个Bolt对外进行一些联系，它访问外部的一些接口，如果这个接口可能是不是那么的稳定，有可能这个接口有时候返回成功有时候返回失败是不是？一旦返回失败就是认为你这个数据就没有成功处理，一旦返回失败的时候这个时候的Bolt也肯定是要有一个失败的逻辑，那一旦走到了失败的逻辑的话呢，那你肯定要告诉Spout这个数据是失败了，那怎么失败呢？就需要你通过开发这个Bolt，然后进行主动的上报，但是这个怎么主动上报也是在execute这个函数里面去实现。

那我们接下来举一个很通俗易懂的例子（如下图15所示）来了解一下整个Storm是怎么去做数据处理的。

图15：

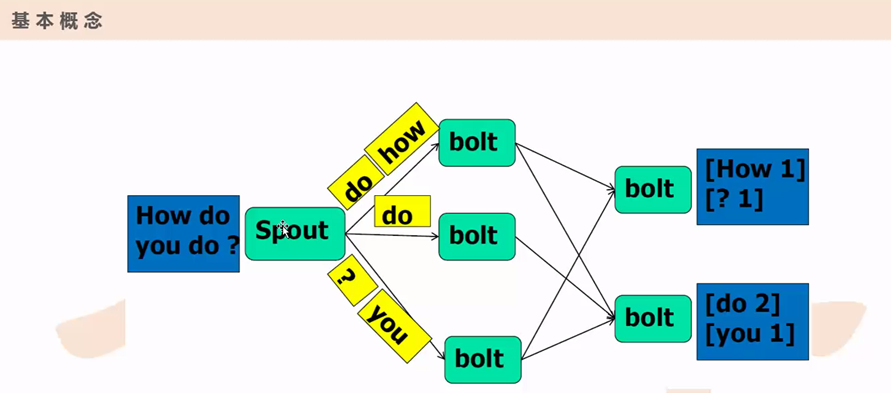


那我们之前讲MapReduce第一个代码案例是讲wordcount。那同样我们用Storm来实现一个wordcount，那怎么实现呢？

那它的输入How do you do是吧？那期望输出就是因为How出现了1次嘛对吧？所以是How 1 ？ 1 。那do出现了2次对吧？而you出现了一次，所以是do 2 you 1 ，期望得到的结果就是如上图15这样（如上图15所示）

输入就是How do you do这么一句话是不是？中间Topology就是数据的计算框架，这个计算框架你可以认为是一个黑盒，你不用关心里面是怎么处理的，你只需要关心你的输入和输出就可以，但是一旦涉及到我们开发，那我们相当于是必须是了解内部的实现嘛，你把这个框架你把这个黑盒打开，那你这个数据就由这个Spout进行接入（如下图16所示）

图16：

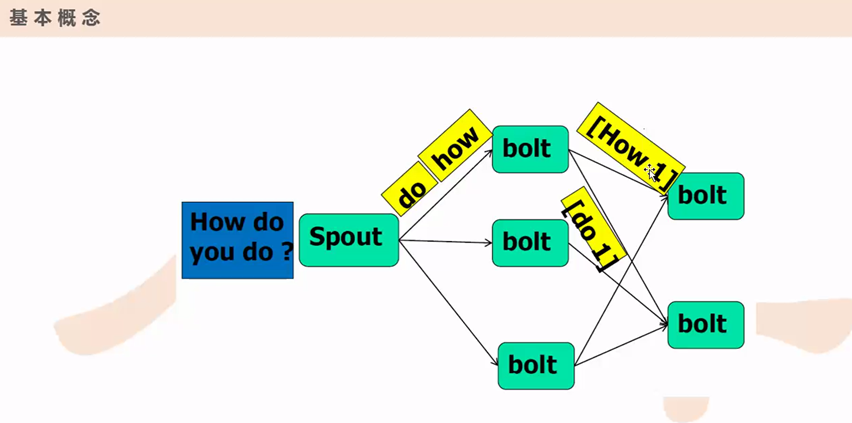


这个时候这个Spout相当于这个水管里它有水了是吧？你把这个水龙头一打开，这个水就可以跟着这个流转到后面不同的环节了对吧？

那第一环节呢，一个句子从Spout输出的时候，从上图16中就可以看出来是不是给How do you do这句做了一个分词了对吧？按照空格给它做了分词了是不是？那我想MapReduce做wordcount也是这样，你一开始来了每一条记录每一句话每一行你都会给它按照空格去做一个分词对不对？然后你对每一个词后面追加一个1是不是？所以这思想是借鉴的，所以这Spout给How do you do这个句子直接做一个分词然后发出去，但这里有一个细节你发现了没？图16中有两个do，但图中的两个do发送到了不同的线上去了是不是？感觉很奇怪，对于MapReduce的时候你要相同的单词是必须发送到同一个reduce上的，那你看图16中的do走了两根线，那从这一点就可以发现它的这种分发机制肯定不是按照之前哈希（hash）partition的策略走的对不对？它是按照随机，它随机的话就完全不关心你的key了，它随机分发，它的随机的目的是保证了你后面这个节点的压力的平均，因为你有时候做partition的时候很容易出现一些数据倾斜的问题，所以随机的这种partition有这种附带均衡的这个功能在里面。

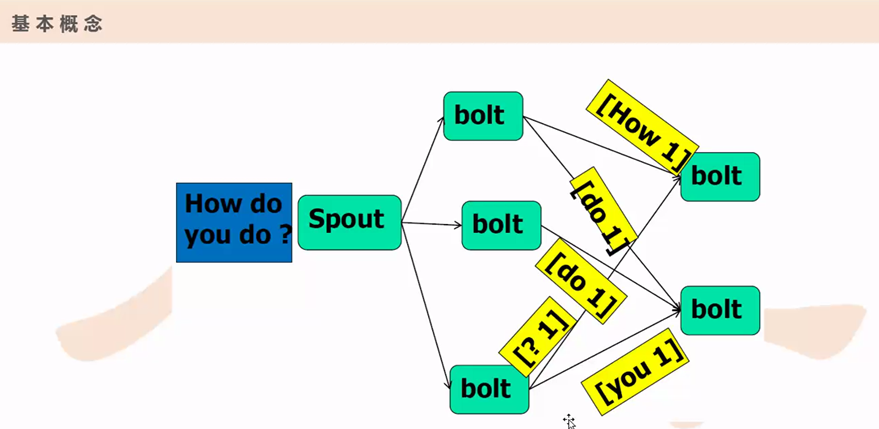
好了，第一步很简单，做分词。那接下来就是bolt了，这bolt要做什么呢？这个bolt做的话呢，也比较简单是吧？跟MapReduce一个思想，见到一个单词就一个打1，所以呢比如（下图17）do how这根线

图17：



见到how就打一个1，见到个do后面就打一个1，所有都这么做明白了把？这个时候你会发现一个问题，看如下图18，你只看第二列和第三列中间的这个分发策略是不是跟MapReduce基本上一样啊？

图18：



你看上图18，原来这个do在两个bolt不同的节点里面进行处理的，但是你看看它最后都归到同一个bolt上去了，所以这样的话这个bolt它最起码可以保证How do you do这句话里面它的一个全局有效性对不对？但是你用这个storm去做一个 wordcount，是不可能做到一个全局的那个wordcount，不像那个MapReduce一样，它是一个固定的一个输入源嘛，像Storm这东西没有一个固定的，所以它只能是保证你一小批一小批的这种wordcount明白了把？

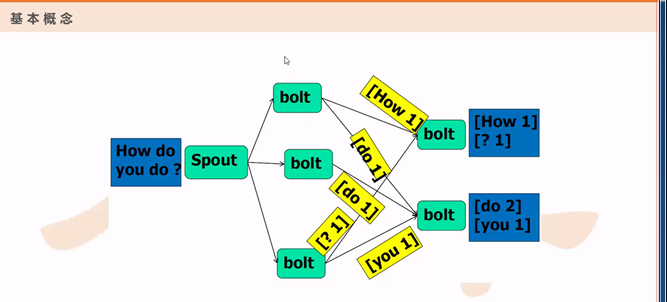
所以它的思想借鉴了那个MapReduce的partition是吧？

好了这个时候后面就简单了，结果一输出How1 do2 you 1 这是我们希望的结果。

然后我们再来看一下回顾一下中间整个流程就是一个Topology一个内部的一个计算框架的一个计算逻辑，那这个计算逻辑是需要我们开发的。所以刚才讲了这么一个很简单的wordcount的例子对吧？Storm大家能听懂了把？

就是如下图19这个例子里面，中间这一列的bolt他们的功能是一样的

图19：

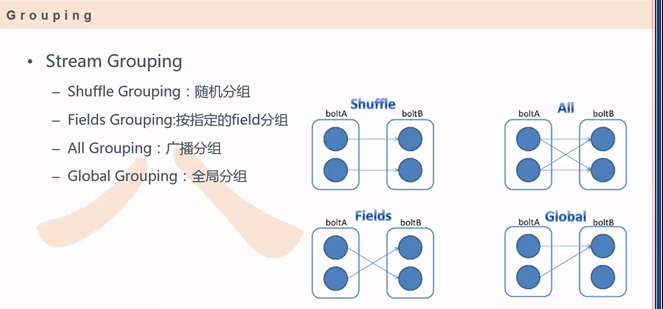


第三列的bolt的功能就是另外一个功能，是不一样的明白了把？

好了接下来我们再仔细看一下图19这个边是怎样做Grouping的。

这Grouping有很多种，下图20中列了四种。（如下图20）

图20：



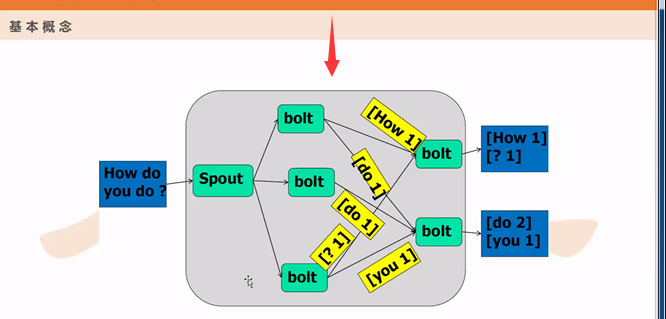
其实这四种通常常用的也就是前两种是最常用的，所以大家要开始学习，把图中的前两种要做好就可以了，真正做项目的时候也就是用前两种，基本上前两种就够用了（如下图21所示）

图21：



那第一种 Shuffle Grouping就是个随机分组，就跟下图22的中间部分是完全随机，刚刚说过了，随机好就负载均衡。

图22：



那Fields Grouping相当于是MapReduce里面的partition，这个partition里面肯定是有key的对吧？那在这个Storm这里面的key不叫key，叫field。

这个field也可以是一个也可以是多个，之前我们讲那个MapReduce里面的那个key，不也是有一个或者多个大家还记得把？你可以通过参数去配，我们可以通过参数你可以配置两个或者更多的key，但是在field的呢，这个是不是配置的，这个完完全全自己写的，你可以写哪些field做partition明白了把？

还有两个做一个扩展把好吧？

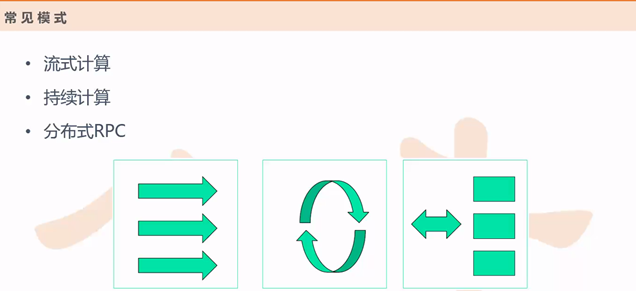
一个是All Grouping广播，什么是广播呢？广播也就是说我前面这个节点产生的消息，把这个消息就是往后面的这个节点每一份我都发一遍，广播就是让大家都知道,比如说后面有十个人，那我消息就是复制十份,每一人都收到一份这个广播。

还有一个是Global Grouping 全局分组，全局分组呢相当于每一次的消息，这个不同前面发出来的消息你最后都要归拢到后面一个bolt上去，相当于有点像合并这样的一个概念，所以你通过这几个图你能够看的出来这个Grouping的一个方式。

Shuffle就是比较均匀，Fields就是你可以按照这个key去做partition，Global 刚才讲过了，比较好理解。

好了接下来我们看一下这个Storm能干什么事情（如下图23所示）

图23：



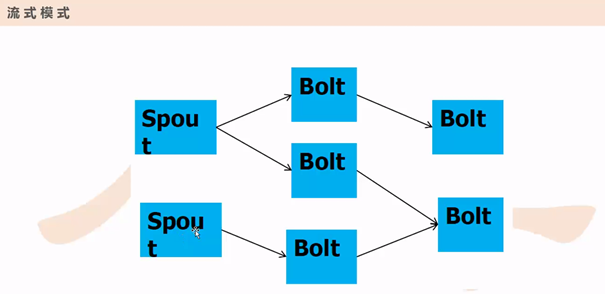
能干三个事，一个是传统的流式计算，也就是我们最最重要的关注的一个使用方法。

第二个是持续计算，持续计算这个事就跟我们刚才讲的一个反馈系统，是一回事。

第三个是分布式RPC是一种web服务，类似于一个云端服务这么一个情况，那我们先看一下流式计算。

流式计算，如下图24所示，这个图就相当于从前面盗过来的是吧？那你要想做一个流式计算，那你必须要分配好每一个节点都是干啥的，每一个节点都分配什么样的角色是不是？

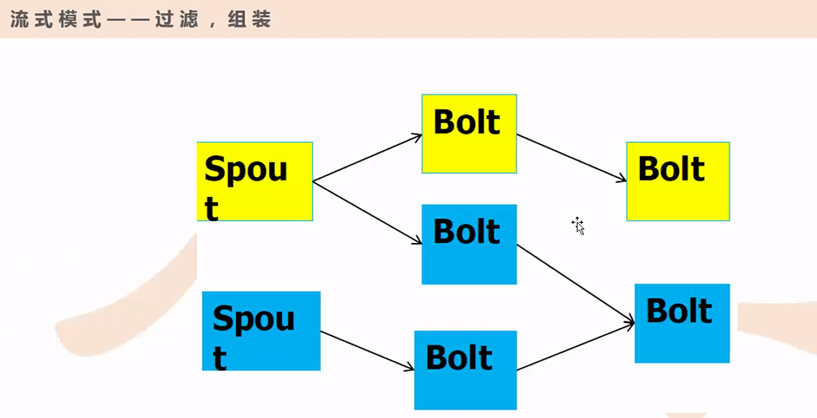
图24：



你即使都是Bolt角色，但是都是你Bolt与Bolt之间你不同角色的Bolt之间的处理功能也是不一样的，这个一定要区分好。

好了，那我们看一下过滤和组装（如下图25所示）

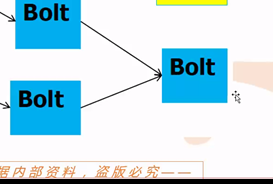
图25：



过滤是什么意思呢？就是这个Spout是一个水龙头是不是？这个水龙头这个流出的水它毕竟是有一些榨汁的对不对？那相当于前面这个输入的这个数据一样，你需要做一些个数据过滤，像有一些格式不匹配的或者是这个缺少字段的，或者字段长度不复符合预期的这些，你就要把这些个数据丢掉是吧？或者一些反作弊的一些垃圾数据你就要识别出来做过滤，做过滤这个比较简单，你就通过一个Bolt做过滤就可以了，中间数据流过来流出去，中间你相当于做一个污水的处理站就可以了是不是？做个清洗把，清洗这个就比较简单了，就做个if判断然后哪些数据过，哪些数据不过，就是一个作用过滤，很简单。

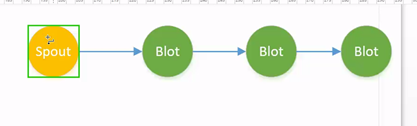
组装是就是这块，两个完全不同的Bolt，把数据合并到了同一个Bolt上（如下图26所示）

图26：



那这种方式就叫组装，但是组装和Join，你会发现这两个很相似，组装和Join都是由多到1的这么一个过程是不是？这个东西就要跟具体的一个业务需求有关了，但是这种情况不是特别的多，组装还稍微多一点。给大家举一个组装的例子，什么情况下会用到组装。

比如说如下图27所示：



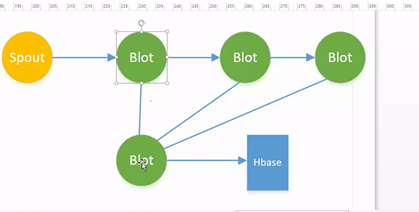
这个大家能看懂吗？我这个Spout这个数据源输出，然后中间经过了三个Blot，而且这三个Blot它是属于整个生产线的不同的角色是不同的环节对不对？这环节和环节之间前后是有顺序的和依赖关系的对不对？，所以他们的工作是不一样的，这个大家应该没什么问题。

那好了，之前刚才说过了这个Storm它是这种没有存储的是不是？没有持久化层的是吧？

没有持久化层有好处也有坏处，好处就是让你数据处理的更快速，坏处就是当你这个系统出了问题的时候我没有数据来指导我，你这个是因为什么问题，你出了问题我不好追踪对不对？所以怎么做呢？就希望应该有一个起码把你这个每一个环节成功和失败，或者失败的一些关键信息给我保留下来。

所以又增加一个Blot（如下图28所示）然后后面跟着一个Hbase

图28：



就是这三个Blot把它的信息都往下面的Blot去发，相当于把所有的信息都汇总到一个Blot上去，由一个Blot或者是同一角色的很多的Blot来同时记录Blot的一个功能明白了吧？所以这个流程这个情况就属于这种组装，就是你来了一个数据就写。

这个时候大家肯定会有疑问，就是记录数据可能不及时，其实不管你及时不及时，你想一想怎么能够达到及时的状态呢？这个都是跟你的数据没关系，也就是说当你这个Blot一旦我想写某一个记录的时候，你要把这个记录告诉这个下面的Blot对不对？

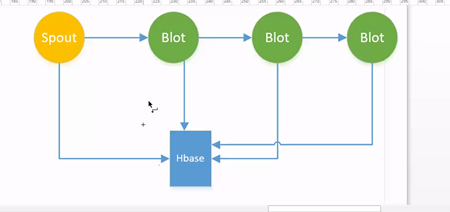
你应该写我的一个状态对不对？然后这个时间戳你必须得自己去打，这个时间戳是由上面的那上面的Blot来通知你，如果你这个时间戳完全依赖于下一个时间戳那就惨了，你这个想追问题根本就是追不出来明白了把？

所以每一个Blot都得有一个角色的一个标记，不仅有标号，因为你每一个角色都有一个标记，这个时候在Hbase去看记录的时候才知道，这条记录是来源于哪一个角色的Blot产生的明白了把？

然后这个时间也是由这个原Blot带过来的对不对？那同时写一个Blot会不会拥堵呢？这个是有这样的可能的，所以有两个点考虑，就是你写这个日志的时候，这个下面的Blot你可以并发度设置的多一点，可以多设置几个，但是也不用设置太多，另外一个这个下面的Blot去写Hbase这个操作其实不是一个特别重量级的操作，通常来说它不会造成整个集群的一个拥堵，所以这种模式相当于是一种通用的模式。

这就是我们说的那种组装的那种方式，这是一种，当然这个Spout也可以通知，但是总觉得有些学生这个下面那个Blot感觉没有必要是吧？那就删掉，删掉了咋办呢？那就每一个节点又不是干不了活是不是？每一个节点都有手有脚的是吧？那就自己去写自己的（如下图29所示）

图29：

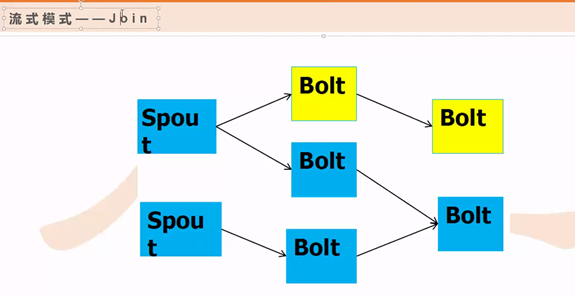


这种方式也可以，直接连Hbase也可以，两种方式大家可以去选择。

所以刚才那种方式就是来表达了这种组装的一个概念这个大家理解了把？

对这个集群时间是要同步的，如果这个集群的时间有不同步的情况下这个集群会报错。

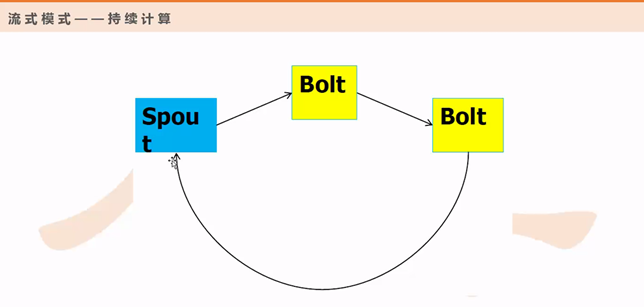
图30



好我们回过头来看上图30所示，这个join这个东西就有点坑了，怎么坑呢？比如说在讲MapReduce的时候相当于是两张表根据同一列的key去做一个数据的合并，这才是做一个join，这是比较理解的，但是这个流式计算你会发现这个数据有时候多有时候少根本就没有办法去做一个全局join是不是？所以这个Join在某一个时间段内然后呢两个Bolt做一个合并的join把，这个肯定是不能做全局的join是不是？所以呢这种join也要看你具体的场景能不能用得上，但是这种场景用的不是那么多。

但是可以做join，那怎么做join呢？这个join还比较麻烦，比如说两个Bolt，第一个Bolt相对比第二个Bolt我先到达了，这个数据你先在第三个Bolt里面进行缓存一下，我等一下下面这个Bolt，如果是下面的这个Bolt也到达的时候，然后再内存做一个join，但是总会有延迟的对吧？你第一个Bolt先到达了，我一直在等你的Bolt，然后这个数据源源不断的灌输进，这个内存会一直在去缓存，所以这个不能太惯着这个下面的这个Bolt了对不对？所以它必须要控制一个内存的一个量，不能让这个内存长的太厉害了是吧？所以这个join只能在一定的范围内它是有这种很强的局限的，所以这个用的不是很多，这个看具体的应用场景有关。

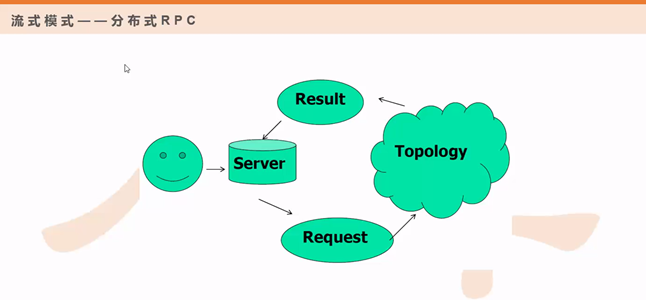
图31：



接下来就是持续计算（如上图31所示），刚才说过了，你的输出直接反馈到你的输入，由你的输出直到你的输入更好的传输，整个的进行系统不断的迭代，这相当于一个通常来说机器学习里面的一个模型一个迭代的一个过程，了解一下就可以。

还有一个就是分布式RPC（如下图32所示）

图32：



分布式RPC也了解一下就可以，就是对外就是一个小笑脸就是一个用户是不是？用户请求是请求了你的服务器，你的服务器呢能不能提供服务完全是需要你依赖服务器背后的一系列的功能和模块是不是？那这个功能呢？这里面都是一个Topology是不是？比如说你这个用户来向你的服务器提交了一个请求，当然你会发现你想用Topology去做这样的一个完全去做一个webServer这是不可能的，因为他是延迟太厉害了是吧？基本上都是几十毫秒的这种级别的，而且肯定没有webServer之前的这种就是纯webServer去读数据库的方式靠谱，那通常是用户上传了一个请求，但是我不着急马上要结果，这就相当于是一个异步过程，异步过程就是说我提交了这个请求，但是我不着急要结果，等着我忙完其他的事情之后我想起来来了我再看看结果有没有产生，这时候Server服务器接到了用户请求，这请求就传到了整个Topology里面，就是传到了整个你的Storm计算逻辑里面，在这个计算逻辑里面，在这个拓补里面不断做一个数据处理，数据处理之后把结果返回，然后传给Server，但是这里面可能会需要介入一个中间的一个存储，或者在Server上有一个Request或者有其他数据库，或者是存在磁盘等等，起码数据的缓存一样把是吧？因为毕竟你这个用户不会理解的要你的结果是吧？

那这个时候如果用户来了，然后这个结果也有了，那直接把这个数据直接读出是不是？所以刚才描述整个的这个过程就是一个分布式RPC，就是用多个机器用一个集群来为一个用户的请求做一个响应大概是这么一个思路。

所以我们主要的学习都集中在第一个流式计算里面。

好了此时已经下课，我们接下来进入Storm笔记进阶（Storm进阶笔记很快做好，敬请期待！！）