Beatles小记（三）－分布式数据流分析中Master的横向扩展

Author：放翁（文初）

Email：[fangweng@taobao.com](mailto:fangweng@taobao.com)

Mblog: weibo.com/fangweng

Blog: <http://blog.csdn.net/cenwenchu79/>

Beatles: <https://github.com/cenwenchu/beatles>

# 读前先看：

这篇文章主要讲述的是beatles流式数据分析框架中对于master的横向扩展真实的设计演进，是beatles框架介绍的第三篇，看第三部分的时候如果看过前两篇文章，这篇文章的递进应该比较容易理解（如果看过前面的代码，那么会更深入的理解其中的细节，文字图片描述一分钟，代码写写1个周）。如果没有看过前两篇，那前提你要理解hadoop等常用的分布式分析系统再看，否则最后可能交流起来就有些空对空了，因为真的写了和用了就会有体会细节的差别。

废话不多说，看完后如果不看代码其实也很难体会细节，因此建议看完后看看代码，跑一下测试用例子（MasterSlaveIntegrationTest\_SocketVersion是socket集成测试版本）。

下面文章中的mr表示MapReduce的意思。

# Master的横向扩展：

尽管Beatles这种松散模式下Slave可以随时随地的加入集群，但由于最终数据还要汇总到Master，Master本身承担着类似于Hadoop中Reduce的角色，所以Master在版本迭代的过程中不断的对本身做着各种优化：并行模式下多线程的数据合并，主干数据分析周期的磁盘换入换出，支持Slave多任务合并后返回，数据导出广度遍历column存储结构等等（详细的参看第二篇文章）。当数据base真的非常大的时候（例如：对于用户纬度的统计，统计后结果还是非常多，无法hold在内存），开放平台分析系统给数据需求方提供的解决方案是片段性输出，交由外部再次合并这些片段。（这是基于当时间片段足够小的时候，数据片内容可以被承受）但结果是让外部数据使用者利用数据库去对这些结果做二次归并。这不仅给数据结果使用者带来了问题，也使得Master随时会被最后一根稻草压倒（如果时间片数据依然无法被hold）。

下面看看老的结构图：

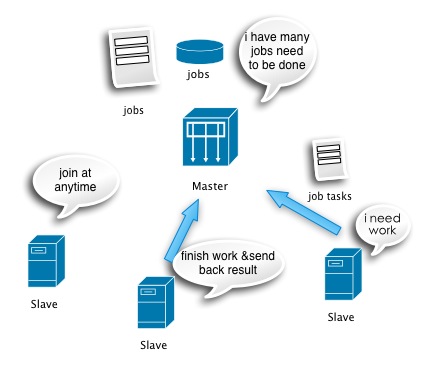


图1 Master横向扩展前结构图

1. master从不同的jobs来源构建需要处理的分析任务，并拆分为多个task是。
2. slave请求任务（一个或者多个）去做分析处理。
3. slave获得任务后从任务描述中获得数据来源，分析规则，输出，开始从数据来源增量获取数据进行分析。
4. slave根据任务定义判断多个任务结果是否可以合并，并将合并后的结果发回给master。至此slave的一轮业务生命周期结束，再从步骤3开始重复。
5. master收到各个slave的数据，开始多线程并行合并job结果，最终判断某一个job的task是否都已经完成，如果完成开始导出数据和临时文件，重置job开始下一轮的job执行。

思考：

是否增加一个角色：reducer来替代掉master的工作？

其实slave一次能够获取多个jobtask，然后自我合并，就是一种比较弱的reduce的做法。

否定了新增一个reducer的原因：新增角色增加了管理的复杂度和集群扩展性。（Hadoop就直接用slave作为reduce）

如果用slave来完全承担所有的reduce工作？

1. 破坏了原来master不管理slave集群的基本原则，slave动态扩展非常麻烦，同时需要增加心跳管理和各种调度算法，任务管理来完成结果的合并。（最后就是一个hadoop的设计）
2. 不考虑用文件作为数据交互的方式（因为流式分析的特点：片段性数据量不大，及时性要求高，所以最好直接是内存），因此hadoop最亮眼的hdfs没有用到，hadoop设计将会大打折扣。

如果用master继续做reduce，是否可以考虑横向扩展master？

Master的职责：

1. 从任务源（可能是本地配置文件或者db或者是http服务）获得jobs定义构建任务池。
2. 被动分配多个job的tasks。
3. 管理job状态以及jobtask的状态。
4. 根据jobtask状态合并slave返回结果到job各个主干上。
5. 根据job状态导出job每一轮的中间结果和统计结果。
6. 根据job状态判断是否重置job。

会发现其实master归根到底就是对job的管理以及对job中数据结果的合并和导出，而最为消耗的就是类似与reducer的4,5两个步骤。

先介绍一下job和jobtask的概念，利于对后面的拆分有更好的理解：

job包含了一组jobtask，job本身定义了一组mr规则（类似有很多mr处理实现），定义了要处理的数据来源（其他信息参看代码）。

Jobtask是将job的多个数据来源拆分后得到的一个子任务，也就是每一个数据来源和同一组mr成为了一个任务，在slave获得一个或者多个task的时候，可以自己通过数据来源去获取数据，然后根据一组规则直接对流式数据做大量的mr（这也是和最早hadoop自己写mr的差别，其实数据的多次移动和读入才是计算集群的最大成本），但最终所有的jobtask都要合并到job的结果主干上，最后导出临时结果和报表数据。

1. 如果等价部署多个master，所有slave连接一组master，来做master横向扩展会如何？
   1. 任务管理就需要多节点之间做并发控制，当前可以看看master内的代码是一个进程内做并发控制。这种方式过于复杂，带来的消耗远大于收益
2. 如果等价部署多个master，所有slave连接一组master，但是将job或者jobtask按照业务（上面说过job就是定义了多个mr的实现，要拆分也只能将mr分组放到不同的master上来减轻mr产生的<k,v>对存储带来的压力）分摊到多个master上，即不用对任务管理做并发控制，又可以分担reduce的工作。
   1. slave主动请求任务，如何判断应该优先向谁请求任务？任务负载均衡如何做？最后可能还是单独部署多套集群来的靠谱。
   2. 将不同的mr放在多个master可行，但结果就和hadoop独立的写mr带来的结果一样，对同一份数据来源处理，却因为mr的分组数据被反复读入和移动。
3. 和Zookeeper类似，master建立group，但是只有一台负责1,2,3,6的工作，而4,5则可以扩展到多个master上。需要对原来系统的改造为：
   1. 多个master job池构建来源保持一致，构建完毕增加mr与master的对应关系（根据算法实现平均分配，后面介绍关于分配的算法，注意只有主的那台master接受任务分发请求，负责将mr与master建立对应关系在task中传递给slave）
   2. slave从一台master上获取任务，分析后将结果按照mr分组（执行的Task中带有，所有设计都是这样，slave不保留任何带有业务性或者集群管理性的配置，保证slave随时离场，随时加入），将分组后的mr结果并行的发送到多个master上。
   3. master在合并和输出结果的时候判断自己所负责的mr部分，按需合并和存储（虽然在slave中已经有做业务数据分组）。

会发现多个master好像多台流水线一样，保持着相同的动作和周期性，从同一个slave获取到了不同原始材料以后，开始制作零件，然后以匹配的速度来将不同零件丢到一个筐子里交给后续处理者。

当然你会考虑到还有容错，master集群动态扩展，速度不匹配等问题，后续细节慢慢介绍。

基于上面所描述的情况，结构演变成如下：

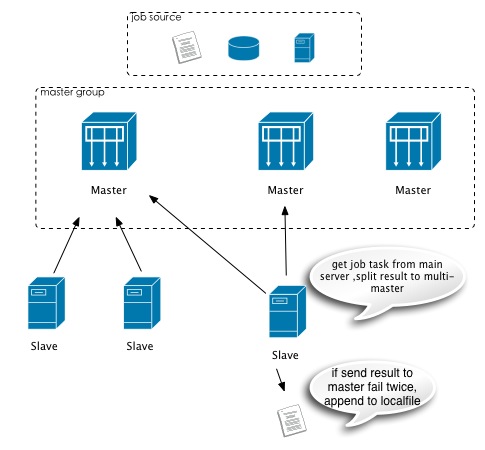


图2 横向扩展后的结构图

可以看到Slave现在获取任务还是集中在一台，但是返回任务结果会支持分散到多台master，解决master瓶颈最大的问题。同时master组的jobs来源保持一致，作为横向扩展的基础（主master分组mr简化master的协同，其他master没有获得数据就没有结果输出）

# 细节：

1. master group之间不需要通信。（主要是业务分拆信息，可以通过幂等算法，也可以通过单机分配，分析结果过滤投递的方式），当前主要是用分析结果过滤投递来保证。
2. 平均分配算法。

首先master有权重（有实体机器，也有虚拟机，处理能力不同），其次mr的权重也不同（有对app做简单统计的，有对用户做统计的，数据量相差非常大）。当前考虑一个mr一台虚拟机抗不论多少数据都能抗的住，如果扛不住后续可以再根据mr产生的结果分（这会增加分流数据的消耗），其实可以看作现在是分库，以后就是分表。分配算法其实就是在两边都有权重的情况下做任务分配，且任务不可切割。

采用了两个排序队列，然后按照简单的饥饿＋权重方式来分配，处理者根据当前获得的饥饿感排序，越饥饿的放在越前面（饥饿感＝已分配任务／自身权重），当已分配任务为0的时候饥饿感＝1／自身权重（保证能力最强的优先获得最大的任务）。任务按照权重排序，高权重的排在最前面。分配过程如下：

1. 获的当前任务队列最前面的任务（权重最高的任务）
2. 将任务分配给处理者队列中饥饿感最强的节点。
3. 节点吃了任务以后会再次按照处理者队列排序。
4. 循环到1再次分配任务，直到任务分配结束。

能看到的就是其实就是有一个评判饿的标准，按照资源权重高低来分配，最后平均分配资源。（也许各位会有更好的建议和算法）

这个算法在保证两个队列构建时始终一致性的情况下，能够变成等幂分配算法，其实也就是当两个队列中如果遇到评判标准相等的时候排序是否会有前后变化，如果没有，那么任何一个master都会产出相同的分配结果。（具体可以参看代码，在ReportUtil中）

1. Master的容灾。由于master其实不是等价集群的模式，因此master的不可靠会导致数据丢失。
   1. Slave如果发送错误，则会尝试再发送一次，如果两次错误，则将master和对应的job作为文件名保存这次隶属于这个job的tasks结果到文件中（append 进文件）。
   2. Master如果恢复的话，可以通过脚本将这些文件复制到master的目录下，master后台线程载入数据合并到主干。
   3. 过程中如果master恢复，后续的消息将会投递到master，因此不会再写这个文件，因此文件不会需要有一个增量拷贝的过程，同时master也可以在后台线程慢慢恢复合并，最后使得数据保持一致性。
   4. 当前还是处于半自动模式，后续可以考虑让slave后台线程将数据发送到恢复了的master上。
   5. 中间可能损失部分时间片数据。
2. mr的动态增加或者减少（随时随地可以针对流式数据产出新的统计分析结果）。原来的集群就支持mr的动态增加，因为都是配置文件修改，重新载入翻译一下即可（统计模型被抽象后mr就可以穷举了，具体参看前面的文档）。当前因为master是多个，同时开始的时候就做好了mr与master的对应关系，因此增加或者减少如果从新做mr与master的分配会产生数据迁移的需求，因此对于mr的增加只是将变化部分再对master group做一次分配（假设原来那些mr分配是均匀的，现在归零master再分配，大志结果也是均匀的），对于mr的减少，只是消除掉task中的定义，mr与master对应关系不消除，避免后面要恢复带来的数据迁移。
3. master的动态增加和减少。这个不可避免的要做数据迁移，当前做法是每天是所有job重置的周期，增加和减少master将在那个时候对整个集群做停机，然后启动集群做重新编译，从今天起点开始分析，追赶数据。以后可以考虑如何做到不停机扩容master。（主要就是数据迁移）
4. 后续考虑做master与mr的冗余分配，既可以保证数据可靠性（多份数据分析存储），又可以方便扩容（数据迁移成本可以间接降低）

# 一些感受

最后master的简单横向扩展模式，使得数据分片，一定程度上得到了数据安全性的保证，同时对于非常消耗cpu和memory的reduce被简单的拆分开来，为业务发展提供了突破，最重要的是系统依然保持最初的设计原则和目的。

任何系统都有自己的适用场景，不要因为要做一个大而全的系统，而丧失了自己设计的原则，我们很难做一个hadoop，但是我们要的也并不是一个通用的hadoop，找到业务场景的特点，才能够用最简单高效的设计方式来完成业务不断增长带来的技术挑战。

另，在设计过程中时不时有同学问我要不要引入zookeeper，过程中曾考虑过，但最后还是觉得要解决了根本问题以后再引入，因为它只是一个工具，便于管理的工具。就像我们要求代码能够做单元测试一样，如果没有zookeeper是否你的系统就无法run，小到模块，大到系统，接口化设计就是要屏蔽实现对系统设计后期可扩展，稳定性的影响，所有系统最后都能够退化为文件交换方式的处理模式，因此如果能够用文件交换可以实现的了的话，你的系统就是最ok的了。（你可以发现linux这么伟大的操作系统就是如此，一切文件化，一切接口化，简单就是美，这个美需要不断的坚持自己的原则）