HamaWhite 原创，转载请注明出处！欢迎大家加入**Giraph 技术交流群**： 228591158

Giraph中Aggregator的基本用法请参考官方文档：<http://giraph.apache.org/aggregators.html> ，本文重点在解析Giraph如何实现**Aggregators**，**后文用图示的方法描述了Aggregator的执行过程。**

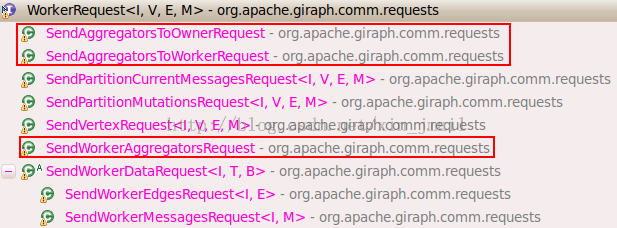
**基本原理**：在每个超级步中，每个Worker计算本地的聚集值。超级步计算完成后，把本地的聚集值发送给Master汇总。在MasterCompute()执行后，把全局的聚集值回发给所有的Workers。

**缺点**：当某个应用（或算法）使用了多个聚集器（Aggregators），Master要完成所有聚集器的计算。因为Master要接受、处理、发送大量的数据，无论是在计算方面还是网络通信层次，都会导致Master成为系统瓶颈。

**改进**：采用分片聚集 (sharded aggregators) . 在每个超级步的最后，每个聚集器被派发给一个Worker，该Worker接受和聚集其他Workers发送给该聚集器的值。然后Workers把自己的所有的聚集器发送给Master，这样Master就无需执行任何聚集，只是接收每个聚集器的最终值。在MasterCompute.compute执行后，Master不是直接把所有的聚集器发送给所有的Workers，而是发送给聚集器所属的Worker，然后每个Worker再把其上的聚集器发送给所有的Workers.

首先给出Master <-- > Worker间， Worker <--> Worker间通信协议，在每个类中的doRequest(ServerData serverData)方法中会解析并存储收到的消息。  
1).  org.apache.giraph.comm.requests.SendWorkerAggregatorsRequest 类 . Worker --> Worker Owner  
功能：每个worker把当前超步的局部 aggregated values 发送到该Aggregator的拥有者。  
2).  org.apache.giraph.comm.requests.SendAggregatorsToMasterRequest 类. Worker Owner--> Master  
功能：每个Worker把自己所拥有的Aggregator的最终 aggregated values 发送给 master。  
3).  org.apache.giraph.comm.requests.SendAggregatorsToOwnerRequest 类. Master --> Worker Owner.  
功能：master把最终的 aggregated values 或aggregators 发送给该Aggregator的拥有者。  
4).  org.apache.giraph.comm.requests.SendAggregatorsToWorkerRequest 类。 Worker Owner--> Worker  
功能： 发送最终的 aggregated values 到 其他workers。发送者为该Aggregator的拥有者，接受者为除发送者之外的所有workers。

http://img.blog.csdn.net/20140520172144734?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQveGluX2ptYWls/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center



**Aggregator分类和 注册**

**Giraph中把Aggregator分为两类：regular aggregators和persistent aggregators。**regular aggregators的值在每个超级步开始会被重置为初始值，然而persistent aggregators的值在整个应用（算法）中一直保持。举例来说，若LongSumAggregator在每个顶点的compute()方法中加1，如果使用regular aggregators，在每个超级步中就可以读取前一个超级步的参与计算的顶点总数；如果使用persistent aggregators，就可以获取前面所有超级步中参与计算的顶点总和。

    在使用aggregator之前，必须要在mastes上Registering aggregators。做法：继承org.apache.giraph.master.DefaultMasterCompute类，重写 void initalize() 方法。在该方法中注册aggregators，语法如下：

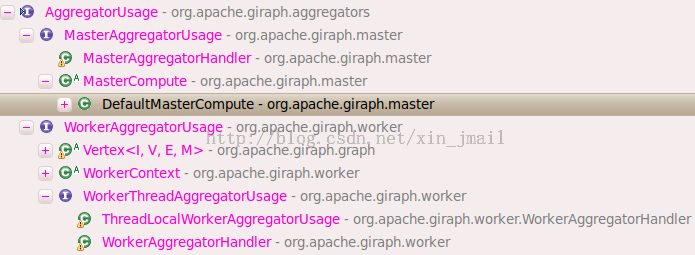
    registerAggregator(aggregatorName, aggregatorClass)  
    registerPersistentAggregator(aggregatorName, aggregatorClass)

   说明：MasterCompute.initalize()方法只在第 INPUT\_SUPERSTEP (-1) 超级步中执行一次，具体在 BSPServiceMaster.runMasterCompute(long superstep)方法中。在MasterCompute.compute()方法中，可以使用下述方法读取或修改聚集器的值。

     getAggregatedValue(aggregatorName) //获取前一个超级步的聚集器值  
     setAggregatedValue(aggregatorName, aggregatedValue) //修改聚集器的值

     MasterCompute.compute()总是在Vertex.compute()前执行。 由于第 INPUT\_SUPERSTEP （ -1）个超级步进行的是数据的加载和重分布过程，不计算Vertex.compute()。第0个超级步Vertex.compute()又是在MasterCompute.compute()方法后执行。**故对第 -1 、 0个超级步MasterCompute.compute()方法中获得的聚集器值均为其初始值**。从第1个超级步开始，MasterCompute.compute()方法才获得了所有Vertex.compute()在第0个超级步聚集的值。

1. 从第0个超级步开始，BspServiceMaster调用MasterAggregatorHandler类的finishSuperStep(MasterClient masterClient) 方法把聚集器派发给Worker，聚集器的value为上一个超级步的全局聚集值（final aggregated values），第一次为初始值。先给出MasterAggregatorHandler的类继承关系，如下：



finishSuperStep(MasterClient masterClient) 方法核心内容如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. /\*\*
2. \* Finalize aggregators for current superstep and share them with workers
3. \*/
4. **public** **void** finishSuperstep(MasterClient masterClient) {
5. **for** (AggregatorWrapper<Writable> aggregator : aggregatorMap.values()) {
6. **if** (aggregator.isChanged()) {
7. // if master compute changed the value, use the one he chose
8. aggregator.setPreviousAggregatedValue(
9. aggregator.getCurrentAggregatedValue());
10. // reset aggregator for the next superstep
11. aggregator.resetCurrentAggregator();
12. }
13. }
15. /\*\*
16. \* 把聚集器发送给所属的Worker。发送内容：
17. \* 1). Name of the aggregator
18. \* 2). Class of the aggregator
19. \* 3). Value of the aggretator
20. \*/
21. **try** {
22. **for** (Map.Entry<String, AggregatorWrapper<Writable>> entry :
23. aggregatorMap.entrySet()) {
24. masterClient.sendAggregator(entry.getKey(),
25. entry.getValue().getAggregatorClass(),
26. entry.getValue().getPreviousAggregatedValue());
27. }
28. masterClient.finishSendingAggregatedValues();
29. } **catch** (IOException e) {
30. **throw** **new** IllegalStateException("finishSuperstep: " +
31. "IOException occurred while sending aggregators", e);
32. }
33. }

问题1：如何确定aggregator的Worker Owner ？  
答：根据aggregator的Name来确定它所属的Worker，计算方法如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. /\*\*
2. \* 根据aggregatorName和所有的workers列表来计算aggregator所属的Worker
3. \* 参数aggregatorName：Name of the aggregator
4. \* 参数workers： Workers的list列表
5. \* 返回值：Worker which owns the aggregator
6. \*/
7. **public** **static** WorkerInfo getOwner(String aggregatorName,List<WorkerInfo> workers) {
8. //用aggregatorName的HashCode()值模以 Workers的总数目
9. **int** index = Math.abs(aggregatorName.hashCode() % workers.size());
10. **return** workers.get(index);  //返回aggregator所属的Worker
11. }

问题2：Worker 如何判断自身是否接收完自己所拥有的aggregators？  
答：Master给某个Worker发送aggregators时，同时发送到该Worker的aggregators数目。使用的 SendAggregatorsToOwnerRequest类对消息进行封装和解析。

2. Worker接受Master发送的Aggregator，Worker把接收到的聚集体值发送给其他所有Workers，然后每个Workers就会得到上一个超级步的全局聚集值。  
由前文知道，每个Worker都有一个ServerData对象，ServerData类中关于Aggregator的两个成员变量如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

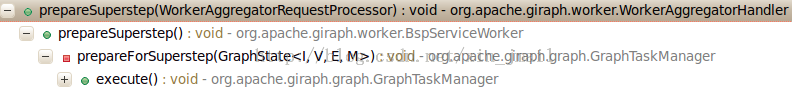
1. // 保存Worker在当前超步拥有的aggregators
2. **private** **final** OwnerAggregatorServerData ownerAggregator;
3. // 保存前一个超步的aggregators
4. **private** **final** AllAggregatorServerData allAggregatorData;

可以看到，ownerAggregatorData用来存储在当前超步Master发送给Worker的聚集器，allAggregatorData用来保存上一个超级步全局的聚集值。ownerAggregatorData和allAggregatorData值的初始化在SendAggregatorsToOwnerRequest 类中的doRequest(ServerData serverData)方法中，如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. **public** **void** doRequest(ServerData serverData) {
2. DataInput input = getDataInput();
3. AllAggregatorServerData aggregatorData = serverData.getAllAggregatorData();
4. **try** {
5. //收到的Aggregators数目。在CountingOutputStream类中有计数器counter，
6. //每向输出流中添加一个聚集器对象，计数加1. 发送时，在flush方法中把该值插入到输出流最前面。
7. **int** numAggregators = input.readInt();
8. **for** (**int** i = 0; i < numAggregators; i++) {
9. String aggregatorName = input.readUTF();
10. String aggregatorClassName = input.readUTF();
11. **if** (aggregatorName.equals(AggregatorUtils.SPECIAL\_COUNT\_AGGREGATOR)) {
12. LongWritable count = **new** LongWritable(0);
13. //Master发送给该Worker的requests总数目.
14. count.readFields(input);
15. aggregatorData.receivedRequestCountFromMaster(count.get(),
16. getSenderTaskId());
17. } **else** {
18. Class<Aggregator<Writable>> aggregatorClass =
19. AggregatorUtils.getAggregatorClass(aggregatorClassName);
20. aggregatorData.registerAggregatorClass(aggregatorName,
21. aggregatorClass);
22. Writable aggregatorValue =
23. aggregatorData.createAggregatorInitialValue(aggregatorName);
24. aggregatorValue.readFields(input);
25. //把收到的上一次全局聚集的值赋值给allAggregatorData
26. aggregatorData.setAggregatorValue(aggregatorName, aggregatorValue);
27. //ownerAggregatorData只接受聚集器
28. serverData.getOwnerAggregatorData().registerAggregator(
29. aggregatorName, aggregatorClass);
30. }
31. }
32. } **catch** (IOException e) {
33. **throw** **new** IllegalStateException("doRequest: " +
34. "IOException occurred while processing request", e);
35. }
36. //接受一个 request,计数减1，同时把收到的Data添加到allAggregatorServerData的List<byte[]> masterData中
37. aggregatorData.receivedRequestFromMaster(getData());
38. }

    每个Worker在开始计算前，会调用BspServiceWorker类的prepareSuperStep()方法来进行聚集器值的派发和接受其他Workers发送的聚集器值。调用关系如下：



    BspServiceWorker类的prepareSuperStep()方法如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. @Override
2. **public** **void** prepareSuperstep() {
3. **if** (getSuperstep() != INPUT\_SUPERSTEP) {
4. /\*
5. \* aggregatorHandler为WorkerAggregatorHandler类型,
6. \* 可参考上文中MasterAggregatorHandler的类继承关系.
7. \* workerAggregatorRequestProcessor声明为WorkerAggregatorRequestProcessor（接口）
8. \* 类型，实际为NettyWorkerAggregatorRequestProcessor的实例，
9. \* 用于Worker间发送聚集器的值。
10. \*/
11. aggregatorHandler.prepareSuperstep(workerAggregatorRequestProcessor);
12. }
13. }

WorkerAggregatorHandler类的prepareSuperstep( WorkerAggregatorRequestProcessor requestProcessor)方法如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. **public** **void** prepareSuperstep(WorkerAggregatorRequestProcessor requestProcessor) {
2. AllAggregatorServerData allAggregatorData =
3. serviceWorker.getServerData().getAllAggregatorData();
4. /\*\*
5. \* 等待直到Master发送给该Worker的聚集器都已接受完，
6. \* 返回值为Master发送给该Worker的所有Data（聚集器）
7. \*/
8. Iterable<**byte**[]> dataToDistribute =
9. allAggregatorData.getDataFromMasterWhenReady(
10. serviceWorker.getMasterInfo());
12. // 把从Master收到的Data（聚集器）发送给其他所有Workers
13. requestProcessor.distributeAggregators(dataToDistribute);
15. // 等待直到接受完其他Workers发送给该Workers的聚集器
16. allAggregatorData.fillNextSuperstepMapsWhenReady(
17. getOtherWorkerIdsSet(), previousAggregatedValueMap,
18. currentAggregatorMap);
19. // 只是清空allAggregatorServerData的List<byte[]> masterData对象
20. // 为下一个超级步接受Master发送的聚集器做准备
21. allAggregatorData.reset();
22. }

下面详述Worker如何判定已接收完所有Master发送的所有Request ？ 主要目的在于描述分布式环境下线程间如何协作。在AllAggregatorServerData类中定义了TaskIdsPermitBarrier类型的变量masterBarrier，用来判断是否接收完Master发送的Request. TaskIdsPermitBarrier类中主要使用wait()、notifyAll()等方法来控制，当获得的aggregatorName等于AggregatorUtils.*SPECIAL\_COUNT\_AGGREGATOR*时，会调用requirePermits(long permits,int taskId)来增加接收的arrivedTaskIds和需要等待的request数目waitingOnPermits. 接受一个Request

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. /\*\*
2. \* Require more permits. This will increase the number of times permits
3. \* were required. Doesn't wait for permits to become available.
4. \*
5. \* @param permits Number of permits to require
6. \* @param taskId Task id which required permits
7. \*/
8. **public** **synchronized** **void** requirePermits(**long** permits, **int** taskId) {
9. arrivedTaskIds.add(taskId);
10. waitingOnPermits += permits;
11. notifyAll();
12. }

http://img.blog.csdn.net/20140521190740921?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQveGluX2ptYWls/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

接受一个Request后，会调用releaseOnePermit()方法把waitingOnPermits减1。

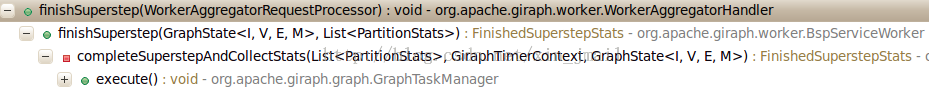


3. 在Vertex.compute()方法中，每个Worker聚集自身的值。计算完成后，调用WorkerAggregatorHandler类的finishSuperstep( WorkerAggregatorRequestProcessor requestProcessor)方法，把本地的聚集器的值给句聚集器的aggregatorName发送给该aggregator所属的Worker. Aggregator的属主Worker接受其他所有Workers发送的本地聚集值进行汇总，汇总完毕后发送给Master，供下一次超级步的MasterCompute.compute()方法使用。finishSuperstep方法如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. /\*\*
2. \* Send aggregators to their owners and in the end to the master
3. \*
4. \* @param requestProcessor Request processor for aggregators
5. \*/
6. **public** **void** finishSuperstep(
7. WorkerAggregatorRequestProcessor requestProcessor) {
8. OwnerAggregatorServerData ownerAggregatorData =
9. serviceWorker.getServerData().getOwnerAggregatorData();
10. // First send partial aggregated values to their owners and determine
11. // which aggregators belong to this worker
12. **for** (Map.Entry<String, Aggregator<Writable>> entry :
13. currentAggregatorMap.entrySet()) {
14. **boolean** sent = requestProcessor.sendAggregatedValue(entry.getKey(),
15. entry.getValue().getAggregatedValue());
16. **if** (!sent) {
17. // If it's my aggregator, add it directly
18. ownerAggregatorData.aggregate(entry.getKey(),
19. entry.getValue().getAggregatedValue());
20. }
21. }
22. // Flush
23. requestProcessor.flush();
24. // Wait to receive partial aggregated values from all other workers
25. Iterable<Map.Entry<String, Writable>> myAggregators =
26. ownerAggregatorData.getMyAggregatorValuesWhenReady(
27. getOtherWorkerIdsSet());
29. // Send final aggregated values to master
30. AggregatedValueOutputStream aggregatorOutput =
31. **new** AggregatedValueOutputStream();
32. **for** (Map.Entry<String, Writable> entry : myAggregators) {
33. **int** currentSize = aggregatorOutput.addAggregator(entry.getKey(),
34. entry.getValue());
35. **if** (currentSize > maxBytesPerAggregatorRequest) {
36. requestProcessor.sendAggregatedValuesToMaster(
37. aggregatorOutput.flush());
38. }
39. }
40. requestProcessor.sendAggregatedValuesToMaster(aggregatorOutput.flush());
41. // Wait for master to receive aggregated values before proceeding
42. serviceWorker.getWorkerClient().waitAllRequests();
43. ownerAggregatorData.reset();
44. }

调用关系如下：



4. 大同步后，Master调用MasterAggregatorHandler类的prepareSusperStep(masterClient)方法，收集聚集器的值。方法内容如下：

**[java]** [view plaincopyprint?](http://blog.csdn.net/xin_jmail/article/details/26381205)[在CODE上查看代码片](https://code.csdn.net/snippets/359108)

1. **public** **void** prepareSuperstep(MasterClient masterClient) {
3. // 收集上次超级步的聚集值，为master compute 做准备
4. **for** (AggregatorWrapper<Writable> aggregator : aggregatorMap.values()) {
5. // 如果是 Persistent Aggregator，则累加
6. **if** (aggregator.isPersistent()) {
7. aggregator.aggregateCurrent(aggregator.getPreviousAggregatedValue());
8. }
9. aggregator.setPreviousAggregatedValue(
10. aggregator.getCurrentAggregatedValue());
11. aggregator.resetCurrentAggregator();
12. progressable.progress();
13. }
14. }

然后调用MasterCompute.compute()方法（可能会修改聚集器的值），在该方法内若根据聚集器的值调用了MasterCompute类的haltCompute()方法来终止MaterCompute，则表明要结束整个Job。那么Master就会通知所有Workers要结束整个作业；在该方法内若没有调用MasterCompute类的haltCompute()方法，则回到步骤1继续进行迭代。

说明：Job迭代结束条件有三，满足其一就行：  
1) 达到最大迭代次数  
2) 没有活跃顶点且没有消息在传递  
3) 终止MasterCompute计算

总结：为解决在多个Aggregator条件下，Master成为系统瓶颈的问题。采取了把所有Aggregator派发给某一部分Workers，由这些Workers完成全局的聚集值的计算与发送，Master只需要与这些Workers进行简单数据通信即可，大大降低了Master的工作量。

**附加：**下面用图示方法说明上述执行过程。

实验条件：  
    1). 一个Master，四个Worker  
    2). 两个Aggregators，记为A1和A2。

1. Master把Aggregators发送给Workers，收到Aggregator的Worker就作为该Aggregator的Owner。下图中Master把A1发送给Worker1，A2发送给Worker3.那么Worker1就作为A1的Owner，Worker3就是A2的Owner。该步骤在MasterAggregatorHandler类的finishSuperStep(MasterClient masterClient) 方法中完成，使用的是SendAggregatorsToOwnerRequest 通信协议。注：每个Owner Worker 可能有多个聚集器。

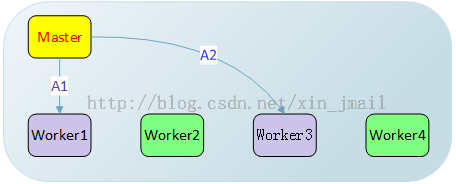
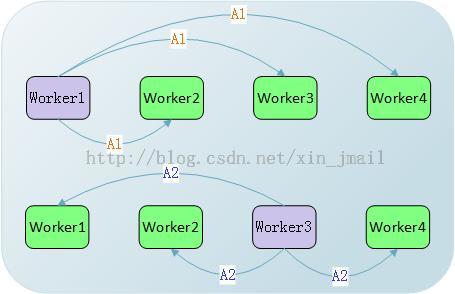
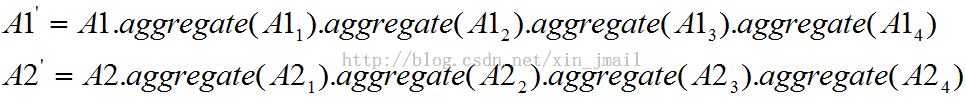


图1 Master分发Aggregator

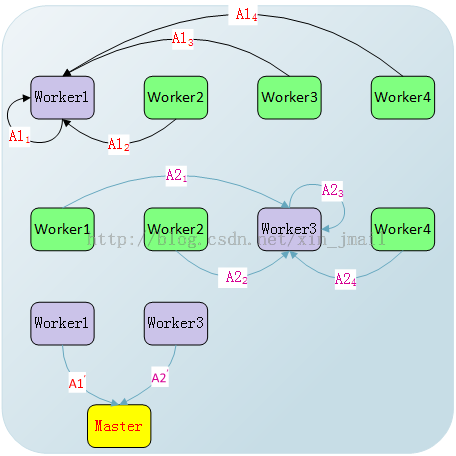
2. Workers接受Master发送的Aggregator，然后把Aggregator发送给其他Workers。Worker1要把A1分别发送给Worker2、Worker3和Worker4；Worker3要把A2分别发送给Worker1、Worker2和Worker4。该步骤在WorkerAggregatorHandler类的prepareSuperstep( WorkerAggregatorRequestProcessor requestProcessor)方法中完成，使用的是SendAggregatorsToMasterRequest 通信协议。此步骤完成后，每个Worker上都有了聚集器A1和A2（具体为上一个超步的全局最终聚集值）。



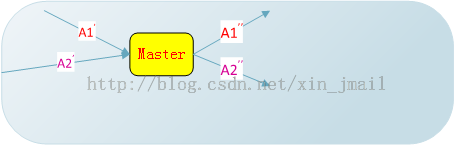
3. 每个Worker调用Vertex.compute()方法开始计算，收集本地的Aggregator聚集值。对聚集体A1来说，Worker1、Worker2、Worker3、Worker4的本地聚集值依次记为：A11、A12、 A13、A14；对聚集器A2来说，Worker1、Worker2、Worker3、Worker4的本地聚集值依次记为：A21、A22、 A23、A24。计算完成后，每个Worker就要把本地的聚集值发送给聚集器的Owner，聚集器的Owner在接收的时候会合并聚集。那么A11、A12、 A13、A14要发送给Worker1进行全局聚集得到A1’，A21 、A22、 A23、A24要发送给Worker3进行全局聚集得到A2’。计算公式如下：



此部分采用的是SendWorkerAggregatorsRequest通信协议。Worker1和Worker3要把汇总的A1和A2的新值：A1’ 和A2’发送给Master，供下一次超级步的MasterCompute.compute()方法使用采用的是SendAggregatorsToMasterRequest通信协议。此部分在WorkerAggregatorHandler类的finishSuperstep( WorkerAggregatorRequestProcessor requestProcessor)方法中完成。过程如下图所示：



4. Master收到Worker1发送的A1’ 和Woker3发送的A2’后，此步骤在MasterAggregatorHandler类的prepareSusperStep(masterClient)方法中完成。然后调用MasterCompute.compute()方法，此方法可能会修改聚集器的值，如得到A1’’和A2’’。在masterCompute.compute()方法内若根据聚集器的值调用了MasterCompute类的haltCompute()方法来终止MaterCompute，则表明要结束整个Job。那么Master就会通知所有Workers要结束整个作业；在该方法内若没有调用MasterCompute类的haltCompute()方法，则回到步骤1继续进行迭代，继续把A1’’发送给Worker1，A2’’发送给Worker3。



完！

本人原创，转载请注明出处！欢迎大家加入**Giraph 技术交流群**： 228591158

版权声明：本文为博主原创文章，未经博主允许不得转载。