火龙果•整理
uml.org.cn

Hadoop C++ Extension:

A Framework for

Making MapReduce More Stable and Faster

总体设计

模块名称	Hadoop C++ Extension
所属系统	Baidu/sys/dpf/hadoop
模块负责人	杨栋 韩富昇 夏鹏
项目负责人	王守彦
文档提交日期	2009-12-8

修改记录

No	版本号	修改内容简介	修改日期	修改人
1	0.1	框架设计初稿	2009-12-9	王守彦，杨栋
2	1.0	完成总体设计	2010-1-8	杨栋，韩富昇，夏鹏

目 录

总体设计 - 1 -
1．背景简介 - 4 -
2．框架设计 - 5 -
2.1 总体架构 - 5 -
2.2 数据通路 - 5 -
2.3 消息序列 - 7 -
3．用户接口 - 8 -
4．模块设计 - 11 -
4.1 HceSubmitter - 11 -
4.2 HceMapRunner & HceReduceRunner - 11 -
4.3 Communication Protocol - 11 -
4.4 Status & Progress - 12 -
4.5 User-Class Registration - 12 -
4.6 MapOutputCollector & ReduceOutputCollector - 12 -
4.7 ReduceInputReader - 12 -
4.8 LineRecordReader & LineRecordWriter - 13 -
4.9 IFileReader & IFileWriter - 13 -
4.10 Combiner - 13 -
4.11 Committer - 13 -
4.12 Compression - 13 -
4.13 SequenceFileReader & SequenceFileWriter - 13 -
5．相关工作 - 14 -
6．参考资料 - 14 -

火龙果•整理
uml.org.cn

1. 背景简介

从实际生产得到的经验显示，执行大数据量的 MapReduce 计算，当 map 数达到数万个，Reduce 数达到数千个是，Hadoop 系统 JVM 对于节点内存的管理显得低效。Java 通过 GC 机制来管理内存，即当某个阈值到达之后才会回收内存，当多个一个节点上有多个 JVM 同时运行时，Java 的内存管理变得低效，影响计算的稳定性和效率。或许替换新的 GC 机制能够满足当前计算的需要，但是本着彻底解决问题的决心，我们计划将数据处理的相关部分全部迁移到 C++空间，希望能够在解决内存问题的基础上获得额外的效率优化。

在 Hadoop MapReduce 框架中，调度和数据处理是两个比较独立的模块，由调度系统决定任务的分配和故障处理。在任务分配之后，调度系统启动相应的处理进程对需要的数据进行处理。基于这个调度和处理非强耦合的 Hadoop MapReduce 框架，为了更有效、更方便地使用 C++方式进行 MapReduce 数据处理，Baidu/SYS/SOS Group 决定开启新的项目，代号 Hadoop C++ Extension（简称 HCE），旨在实现一套完整的 C++数据处理机制，允许用户使用默认或自行定制的功能对象来处理数据，同时在 C++空间更为直接地管理 TaskTracker 占用内存，并希望带来处理效率上的提升。

HCE 项目总的计划为两期:

第一期计划是实现并维护 HCE 第一个可用版本，包括框架部分和各个独立的功能模块，涉及 LineRecoredReader/LineRecoredWriter、IFileReader/IFileWriter、SequenceFileReader/SequenceFileWriter 、 HadoopOutputCommitter 、MapOutputCollector/CombineOutputCollector/ReduceOutputCollector 及支持 Lzo/Zlib 的流式压缩模块等等。其中，输入数据的切分仍由 Java 的 InputFormat 完成；Reduce 处理中，Shuffle 和 Merge Sort 仍由 Java 完成，其余的 MapReduce 数据处理通路将全部迁移到 C++实现。

第二期计划是实现新的 Shuffle 和 Sort 方式，这部分仍在计划中。

火龙果•整理
uml.org.cn

2. 框架设计

2.1 总体架构

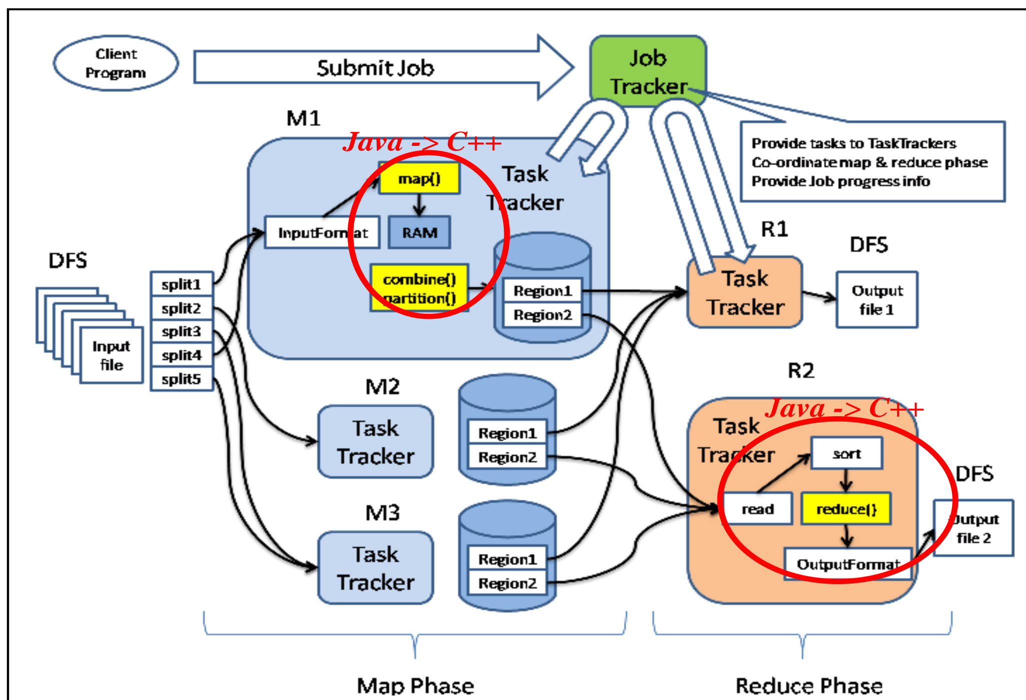

图 1 Hadoop Mapreduce 框架示意图

图 1 是 Hadoop Mapreduce 的框架示意图，JobTracker 负责调度，TaskTracker 启动 Child Java 执行 task，Task 相关的数据处理是由 Child Java 完成的；而在 HCE 框架设计中，Task 相关的数据处理将由独立的 C++子进程来完成，TaskTracker 的 Java 部分只负责驱动和监测。

2.2 数据通路

图 2 给出了 HCE 框架下用户作业执行的大致流程，下面描述一下具体的数据通路，相关 Java 模块和 C++模块的设计将在模块设计中介绍。

1. 用户在程序中自定义相关子类，例如 Mapper、Reducer 及 RecordReader 和 RecordWriter 等。

2. 用户通过命令行提交作业，提交时 HceSubmitter 配置 HCE 运行所需的参数，应用程序提交作业到 JobTracker，提交之前需注册 HceNoJavaInputFormat 类，其负责将输入数据切分为逻辑块 InputSplit，这些块的地址会传递给 C++子进程，由 C++进程读取数据。

3. 作业被 JobTracker 分解成为 Task，分发到集群中，在 TaskTracker 的控制下运行，Task 包括 MapTask 和 ReduceTask。

TaskRunner 启动 Child 子进程来处理每个 Task，在启动 Child 之前，已经完成可执行程序分发和工作目录创建，Task 对象也已经创建。Child 根据 Task 是 MapTask 还是 ReducTask 调用对应的 run 方法。

火龙果•整理
uml.org.cn

4. Child Java 执行 HceMapRunner，启动 C++子进程，并与之建立 socket 通信。Java 端的 BinaryConnection 类负责与 C++端传递消息。

HCE C++子进程启动并建立连接后，等待 Task 到来。

5. Java 和 C++端建立连接后，HceMapRunner 先将 JobConf 传递给 C++子进程，随后会执行 RunMap 调用，即通知 C++子进程执行 Map 处理，而 Java 本身等待 task 执行完成或中断。

6. C++子进程收到 RunMap 命令，根据用户定制或默认实现创建 RecordReader、MapOutputCollector、IFileWriter、Partitioner 等对象，Mapper 调用 setup 操作，然后通过 RecordReader 通过 Java 端传来的 inputSplit 信息，从 DFS 上迭代读取 key/value 对，进行 map 操作，最后 cleanup。

7. Mapper 的结果，可能送到 Combiner 做合并；Mapper 最终处理的结果对<key, value>，是需要送到 Reducer 去合并的，合并的时候，相同 key 的键/值对会送到同一个 Reducer 那，哪个 key 到哪个 Reducer 的分配过程，是由 Partitioner 规定的。总之，Map 的结果是通过 MapOutputCollector 在本地进行 spill/IFileWriter 操作，而非提交给 Java 的 OutputCollector。

8. C++子进程 Mapper 执行完成或异常时，Java 端通过连接捕获异常，之后会释放所有资源。Child Java 收到 task 完成消息，选择返回或抛出异常。

9.执行 Reducer 的数据流程类似于 Mapper。Child Java 执行 HceReduceRunner，启动 C++子进程并与之建立通信；C++子进程收到 RunReduce 命令后，创建 RecordWriter、ReduceInputReader、ReduceOutputCollector 等对象。待 Reducer 执行完成后，会通知 child Java 线程。

这里需要强调的是，Shuffle 和 Merge Sort 过程仍然是由 Java 来完成的，Java 端将 shuffle 并排序以后的多个数据文件（mapX.out）的路径信息传递给 C++进程，框架内置的 C++ ReduceInputReader 读取这些已经拖到本地的数据文件，然后传递给 Reducer。Reduce 的结果，通过 ReduceOutputCollect 输出到文件中。

10. 当 Reduce 执行完毕之后，需要将写完的临时文件转正，即 move 到作业输出路径下，Commiter 完成这一步操作。框架的 Committer 可以由用户定制，这样在最终输出对象是 Hbase 或其他存储系统时，能够减少一次额外的拷贝。框架默认实现的是 C++的 HadoopOutputCommitter，由其完成输出结果文件的转正。

为了和 MapReduce 的 Java 数据通路完整契合起来，在 Java 端实现了 C++ Commiter 的代理类 HceOutputCommitter，它负责将 HadoopOutputCommitter 嵌入 Task Java 的处理逻辑。

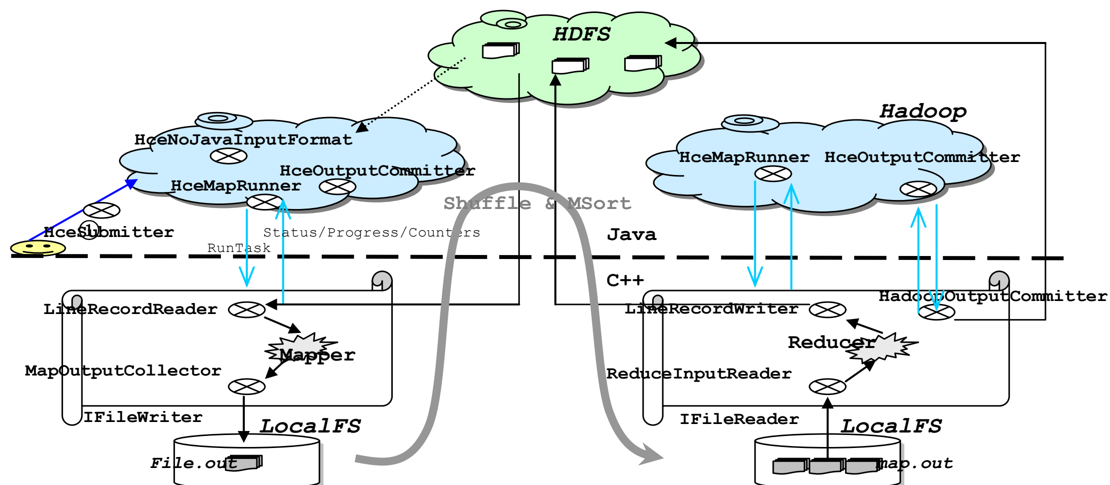

图 2 Hadoop C++ Extension 数据通路示意图

2.3 消息序列

图 3 描述了 HCE 框架的 Java 端和 C++端的消息交互时序，通过时序图能够知晓框架内部的驱动逻辑。

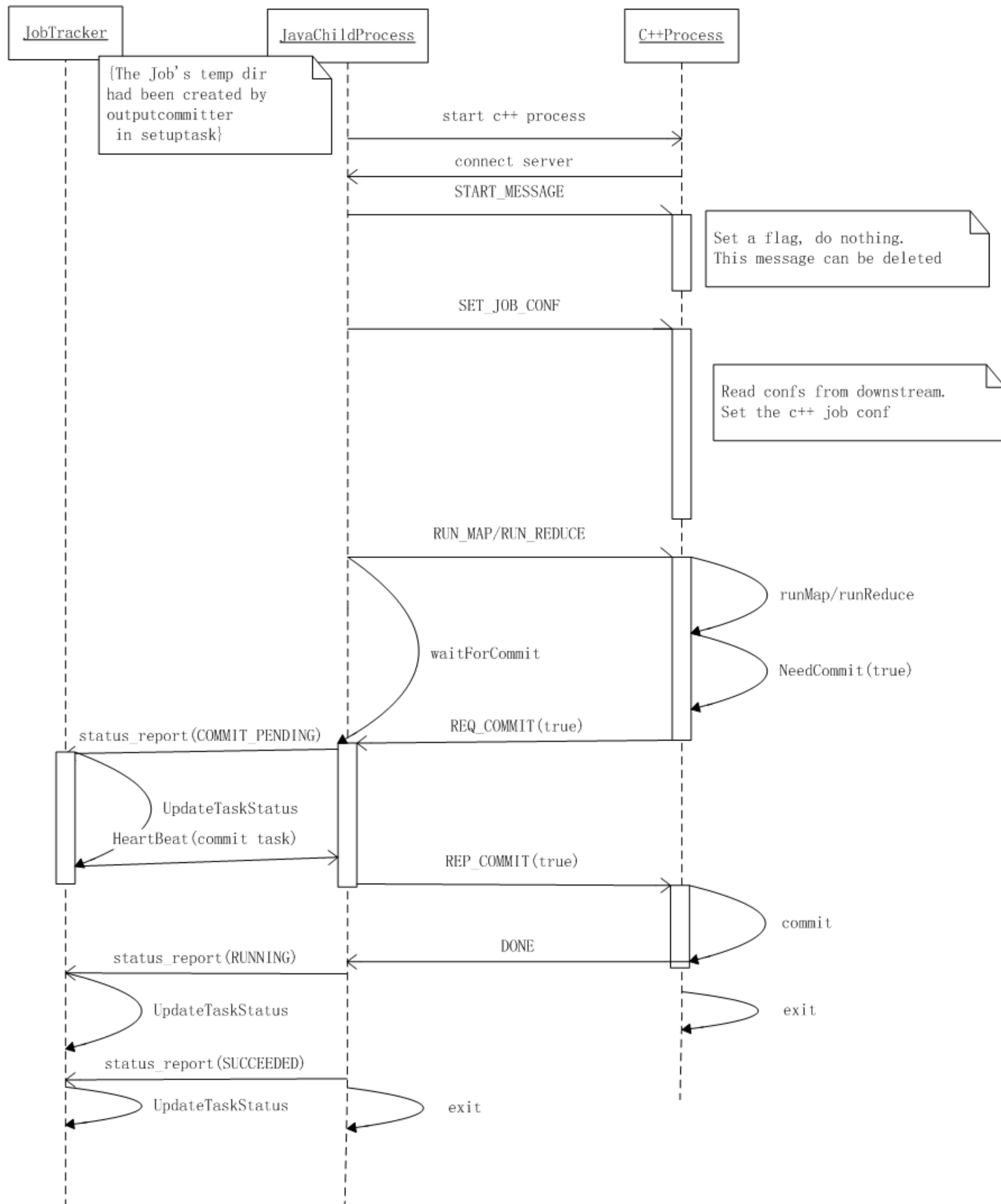

图 3 HCE 框架 Java-C++消息传输时序图

3. 用户接口

用户程序通过继承以下的接口类来定义自己的功能模块。

```
class Mapper {
public:
virtual int64_t setup() {return 0;}
virtual int64_t cleanup(bool isSuccessful) {return 0;}
virtual int64_t map(MapInput &input) = 0;

protected:
virtual void emit(const void* key, const int64_t keyLength,
const void* value, const int64_t valueLength) { getContext()->emit(key,
keyLength, value, valueLength); }
virtual TaskContext* getContext() { return context; }
};
```

```
class Reducer {
public:
virtual int64_t setup() {return 0;}
virtual int64_t cleanup(bool isSuccessful) {return 0;}
virtual int64_t reduce(ReduceInput &input) = 0;

protected:
virtual void emit(const void* key, const int64_t keyLength,
const void* value, const int64_t valueLength) { getContext()->emit(key,
keyLength, value, valueLength); } }
virtual TaskContext* getContext() { return context; }
};
```

```
class Combiner {
public:
virtual int64_t setup() {return 0;}
virtual int64_t cleanup(bool isSuccessful) {return 0;}
virtual int64_t combine(ReduceInput &input) = 0;

protected:
virtual void emit(const void* value, const int64_t valueLength)
{ getContext()->emit(getCombineKey(), getCombineKeyLength(), value,
valueLength); }
virtual TaskContext* getContext() { return context; } }
virtual const void* getCombineKey() { return combineKey; }
virtual int64_t getCombineKeyLength() { return combineKeyLength; }
};
```

```
class Partitioner {
public:
virtual int64_t setup() {return 0;}
virtual int64_t cleanup() {return 0;}
virtual int partition(const void* key, const int64_t keyLength, int
numOfReduces) = 0;
};
```

```
class Committer {
public:
  virtual int commit() = 0;
  virtual bool needsTaskCommit() = 0;
  virtual ~Committer() {}
};
```

```
class RecordReader {
public:
  virtual int next(void*& key, int64_t& keyLength,
           void*& value, int64_t& valueLength) = 0;
  virtual float getProgress() = 0;
  virtual int64_t open(){return 0;}
  virtual int64_t close(){return 0;}
  virtual ~RecordReader(){};
};
```

火龙果•整理
uml.org.cn

```
class RecordWriter {
public:
  virtual void emit(int nStream, const void* key, const uint64_t keyLength,
                    const void* value, const uint64_t valueLength) {}
  virtual void emit(const void* key, const uint64_t keyLength,
                    const void* value, const uint64_t valueLength) = 0;
  virtual int64_t open(){return 0;}
  virtual int64_t close(){return 0;}
};
```

火龙果•整理
uml.org.cn

4. 模块设计

4.1 HceSubmitter

当用户执行如下命令行时，基于 Hce 框架的 MapReduce 计算将被启动。

```
./hadoop hce -Dmapred.job.name=text_1.4G -reduces 2 -input /user/test/data/text_1.4G -output /user/test/hce/output/20100221-1625 -f /user/test/hce/bin/wordcount
```

mapred.org.apache.hadoop.mapred.hce.Submitter.java 基于命令行参数提交一个 hce 作业，Submitter 会指定一些 conf 参数，例如：

mapred.task.hce.enable 参数为 true；

MapRunner 为 HceMapRunner，ReduceRunner 为 HceReduceRunner；

mapoutput 和 reduceoutput 的 k/v 类型为 Text；

InputFormat 类为 HceNonJavaInputFormat，OutputFormat 为空；

HADOOP_LIB_DIR 和 LD_LIBRARY_PATH 的默认路径。

随后 Submitter 会作为 Jobclient 提交作业给 JobTracker。

4.2 HceMapRunner & HceReduceRunner

当TaskTracker启动JVM处理任务时，实际调用继承MapRunner/ReduceRunner的HceMapRunner/HceReduceRunner类（通过Submitter制定）的run方法。该方法会创建C++子进程，其作为客户端和java端通过本地socket建立连接，当然创建进程时Submitter指定的环境参数会一并传递过去。

连接创建之后，jvm向C++进程传递JobConf，C++收到后会构建相应的Map保存。另外，C++进程启动后会通过rlimit限制该进程的资源使用。

之后，HceMapRunner/HceReduceRunner会通过协议发命令RunMap/RunReduce给C++子进程，由其完成数据处理。

MapRunner在原有Hadoop实现中存在，其run方法是“run(RecordReader<K1, V1> input, OutputCollector<K2, V2> output, Reporter reporter)”；而ReduceRunner在原有实现中不存在，Reduce接口是Iterator方式而非RecordReader方式。因此，我们在保持原有逻辑的前提下，在ReduceTask.java中添加HceEnable路径，使Reduce任务执行时采用类似MapRunner的ReduceRunner方式。

ReduceRunner工作的前提是shuffle&sort之后的数据已经全部Dump到磁盘形成一系列文件（mapX.out）。这些文件列表将作为jobconf的配置传递给C++子进程，由其读取相应文件作为Reduce的输入。

4.3 Communication Protocol

Java 和 C++端都有两个 BinaryConnection 类来实现 socket 通信机制。

从 Java 到 C++的消息类型包括 SET_JOB_CONF、RUN_MAP、RUN_REDUCE、CLOSE、

ABORT、REP_COMMIT 等；从 C++到 Java 的消息包括 STATUS、PROGRESS、REGISTER_COUNTER、INCREMENT_COUNTER、REQ_COMMIT、DONE 等。

JVM 故障时，C++端通过 ping 连接知晓，即每隔 5s 通过连接操作，从而异常退出；C++进程故障退出时，Child Java 作为 socketServer 捕获异常 socket 消息（while 循环），进而抛出作业失败异常；C++进程执行过慢时，由 Hadoop Java 调度框架 kill task 重新调度。

4.4 Status & Progress

为了防止 C++子进程在处理过程中陷入困境而不能及时汇报 Status/Progress 给 JVM，C++进程在启动后又创建了一个子线程 pingThread，pingThread 负责每隔 5s 将 Status/Progress/Counters 传递给 JVM，再由 Java 传给 Reporter，触发 Reporter 的相关操作如 getCounter 等，以便用户从 web 界面或终端上查询。

Status 和 Progress 通过默认/用户定制的 RecordReader 类的 getProgress 方法获取，Counters 由 Hce 框架或用户程序注册，并在 Hce 框架内维护计数。

Java 端 Reporter 在获取进度时是通过 RecordReader 的 getProgress 方法，这个 RecordReader 在 InputFormat 中定义，即 HceNonJavaInputFormat 类中定义了 HceDummyRecordReader，当 Java 收到 C++传来的 progress 消息后，将收到的进度数值赋值给 HceDummyRecordReader 的相关变量，然后由其 getProgress 返回给 Reporter 显示。

pingThread 和主进程使用同一个 Connection，因为该 Connection 除了 Java 到 C++的 JobConf 和命令传递外，其余功能就是 C++到 Java 的状态/计数传递，在汇报状态/进度/计数是，连接是空闲的。同时，每隔 5s 的消息传递也会检测连接或 JVM 是否正常，如果消息传递重复多次失败，那么 pingThread 将会促使 C++进程异常退出。

4.5 User-Class Registration

HCE 框架通过 factory 方式注册用户定制类。用户定义功能类之后，在 main 函数中通过执行如下方式来注册自定义类：

```
HCE::runTask(HCE::TemplateFactory<testMapper, testReducer, testPartitioner,
testCombiner, testCommitter, testRecordReader, testRecordWriter>());
```

4.6 MapOutputCollector & ReduceOutputCollector

// TODO

4.7 ReduceInputReader

// TODO

4.8 LineRecordReader & LineRecordWriter

// TODO

4.9 IFileReader & IFileWriter

// TODO

4.10 Combiner

// TODO

4.11 Committer

在 Hadoop MapReduce 原有实现中，Map 结果通过 OutputFormat 机制输出，它依赖两个辅助接口:RecordWriter 和 OutputCommitter。RecordWriter 提供了 write 方法，用于输出<key, value>和 close 方法，用于关闭对应的输出。OutputCommitter 提供了一系列方法，可以定制 Task 的准备或清理等。

在 Hadoop Java 中，FileOutputCommitter 是默认的 OutputCommitter 实现，MapTask/ReduceTask 在 Map/Reduce 执行完成后判断 task 是否需要 Commit，考虑到预测执行的需要，如果两个 Task 同时执行，那么有一个是不需要 commit 的。

HCE 框架在 Java 端提供了 HceOutputCommitter，它是 Proxy Committer，嵌入原有 Hadoop 实现逻辑中；实际工作的 Committer 是 C++空间默认实现的 HadoopOutputCommitter，它是 FileOutputCommitter 的 C++实现。

HadoopOutputCommitter 的 needCommit 方法会向 Java 发送附带是否需要 commit 信息的 reqCommit 请求，当 Task 需要 commit 并且 TaskTracker 和 JobTracker 通信之后，HceOutputCommitter 的 commit 方法向 C++发送 repCommit 应答，由 HadoopOutputCommitter 执行真正的 commit。

4.12 Compression

// TODO

4.13 SequenceFileReader & SequenceFileWriter

// TODO

5. 相关工作

Hadoop中作业调度和数据处理部分代码都是使用java编写的,默认接口语言为java语言，尽管如此，MapReduce应用程序可以使用其他语言编写，比如C++。

Hadoop 通过常用的 Streaming 接口可将应用程序嵌入数据处理模块，Hadoop Streaming帮助我们用非Java的编程语言使用MapReduce,Streaming用STDIN（标准输入)和STDOUT (标准输出)来和我们编写的Map和Reduce进行数据的交换数据。任何能够使用STDIN和STDOUT都可以用来编写MapReduce程序，比如我们用Python的sys.stdin和sys.stdout，或者是C中的stdin和stdout。

另一个框架Pipes也允许C++代码介入MapReduce框架，主要方法是将C++代码放到独立的进程，该进程处理定制的应用代码，其与JVM通过socket进行通信。作业可以包括 Java 或 C++实现的功能对象的任意组合，这些功能对象包括 RecordReaders、Mappers、Paritioner、Combiner、Reducer及RecordWriter等。

Hce也通过本地的socket实现JVM和C++子进程的通信，同样支持用户定义各种功能对象。与Pipes不同,Hce完成了MapOutputCollector、ReduceOutputCollector等功能对象的C++实现，另外也支持OutputCommitter的用户定义方式。这种方式使得Hadoop MapReduce框架各功能模块的耦合性更低，数据处理可以完全处于C++空间，从而实现了数据处理部分和作业调度部分的弱耦合。Hce 框架不仅在语言方面利用了 C++处理数据相较Java的优势，而且通过更低的耦合度保证了数据传输更加有效。

第一个版本的Hce将仍然使用Hadoop Java实现的数据shuffle和sort，另外，输入数据的切割也还是由Java完成。在下期工作中，Hce将实现更加高效的切分和拖取数据方式，从而使得整个MapReduce数据通路和Java调度框架完全弱耦合。

6. 参考资料

[1]OReilly.Hadoop.The.Definitive.Guide.June.2009.RETAiL.eBOOk-sUppLeX.pdf