# MapReduce

这里来一张MapReduce整体运行过程的图。

## Task运行过程概述

每个MapTask处理输入数据的一片数据（InputSplit）。

MapTask：通过用户提供的InputFormat将对应的InputSplit解析成一系列K/V，交给用户编写的map()函数处理，接着按照指定的Partitioner对数据分片，以确定每个K/V交给哪个Reduce Task处理，之后将数据交给用户定义的Combiner进行一次本地规约（没有定义就直接跳过），最后将结果保存到磁盘上。

Reduce Task：通过HTTP请求从各个已经运行完成的Map Task上拷贝对应的数据分片，待所有数据拷贝完成后，再以key为关键字对所有数据进行排序，通过排序，key相同的记录聚集到一起形成若干分组，然后将每组数据交给用户编写的reduce()函数处理，并将结果直接写到HDFS上作为最终输出结果。

## 基础组成

因为用户可以指定InputFormat和OutputFormat，但不包括Map Output输出格式（Reduce Input输入格式）。为了尽可能减少数据量以避免不必要的磁盘和网络开销，Hadoop内部实现了支持行压缩的数据存储格式——IFile。

### IFile存储格式

支持行级压缩。有序。Map Output和Reduce Input的写、读。CRC32校验。

数据格式：<key-len, value-len, key, value>

IFile是Map过程Output的中间文件。对应一个IFile.Writer和IFile.Reader。

IFileInputStream，IFileOutputStream用来给IFile做checksum

在reduce包下还有一个InMemoryReader来读取内存里的map-outputs

### 排序

缓冲区数据排序使用的是优化过的快排。

IFile文件的合并使用了基于堆实现的优先队列。

Map Task里，处理的结果会先放到缓冲区里，使用率达到阈值时，会对缓冲区里的数据进行一次排序，并将这些有序数据以IFile文件形式写到磁盘上。数据处理完毕后，对磁盘上所有文件进行一次合并，合并成一个大的有序文件。

Reduce Task里，从每个Map Task那边远程拷贝相应的数据文件，如果文件大小超过一定阈值，则放到磁盘上，否则放在内存里。如果磁盘上的文件数目达到一定阈值，会合并成一个大文件；如果内存里文件大小或数目超过一定阈值，则进行一次合并后写到磁盘上。所有数据拷贝完后，统一对内存和磁盘上的所有数据进行一次合并。

#### 快排

QuickSort的代码在common项目的util包里。

优化点：

1. 枢轴选择
2. 子序列划分方法
3. 对相同元素的优化
4. 减少递归次数：小于13长度的时候直接插入排序

具体可以看QuickSort.java代码，加注释了。

#### 归并排序

MergeSort的代码在common项目的util包里。

长度较小(7)的时候也是直接使用插入排序。

#### 优先队列

由Merger类主导，Merger被Map和Task任务用来归并内存里的和磁盘上的segments。Segment是对磁盘和内存中的IFile格式文件的抽象。

Segment是Merger的一个内部类。

RawKeyValueIterator专门用于迭代读取sort/merge中间结果数据的KV。

Merger内部优先队列的实现类：

**private** **static** **class** MergeQueue<**K** **extends** Object, **V** **extends** Object>

**extends** PriorityQueue<Segment<**K**, **V**>> **implements** RawKeyValueIterator

抽象类PriorityQueue是hadoop-common项目，util包下hadoop自己实现的一个版本。PriorityQueue实现了heap的操作，留了一个比较函数lessThan(Object a, Object b)让子类实现。

PriorityQueue的分析：

push()：直接放入最末，进行一次upHeap()

insert()：看size大小，如果小，直接push；否则与top比较，通过的话取代top，然后进行downHeap()

pop()：把top的取出来后，把最后的元素放在first位置，然后进行一次downHeap()

top()：取第一个值

upHeap(), downHeap()是对Heap的遍历调整。上推和下推操作都很好理解，父和子的位置关系是两倍的关系，一步步比较就可以了。

Merger内部Segment与MergeQueue关系，MergeQueue的merge()操作分析：

MergeQueue里面排序的对象是Segment<K, V>，排序逻辑略，比较繁杂。

### Reporter

Map和Reduce进行中的任务进度汇报。Task周期性向TaskTracker汇报最新进度和计数器值。

Task类的内部类TaskReporter实现了Report接口。

在common-hadoop项目的util包里，有个Progressable接口。Reporter接口继承了Progressable。

#### 任务执行进度

对Map Task来说，当前已读取数据量占总数据量的比例作为任务当前执行进度值。

对于Reduce Task来说，Shuffle、Sort、Reduce，每个阶段占1/3，在shuffle阶段，有M个Map Task读取数据的话，相当于每个阶段占了shuffle阶段的1/M。整个Progress是一颗进度树。

TaskReporter线程汇报触发条件：1.任务执行进度变化；2.任务某个计数器值变化。

若某个时间间隔内两者都未变化，Task会通过RPC函数ping探测TaskTracker是否活着。

用户可以每隔一段时间调用一次TaskReporter.progress()来告诉TaskTracker自己仍活着。

#### 任务计数器

MapReduce内置计数器

用户自定义计数器

## Map Task

Map Task具体可以拆为五个过程：

Read -> Map -> Collect -> Spill -> Combine

MapTask继承Task，有mapPhase和sortPhase两个Progress(common包里的类，和report有关)。

内部有TrackedRecordReader<K, V>和SkippingRecordReader<K, V>

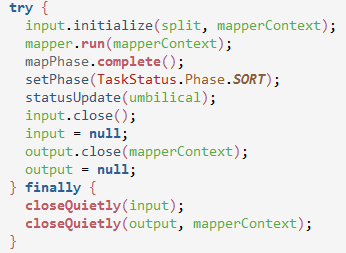
run逻辑：

如果没有reduce task，那么只有map阶段；否则有map阶段和sort阶段，并且整个progress是67%的map阶段，33%的sort阶段。

正式跑map task前，要先进行jobCleanupTask，jobSetupTask和taskCleanupTask。

执行map task逻辑：

先根据任务信息，new出TaskAttemptContext、Mapper、InputFormat，还有一个outputCollector等。然后是执行，



MapOutputCollector在collect()方法里把kv数据写到RecordWriter<K, V>里

MapOutputBuffer很复杂的类，里面有map里spillThread的设置，combine阶段等，主要是配合MapOutputCollector处理map出来的结果的后续处理逻辑。

如果没有reduce task，那么使用的是directOutputCollector实现，否则使用outputCollector实现

### Collect

MapOutputBuffer

双缓冲区和循环缓冲区

### Spill

溢写

IndexSorter的实现是QuickSort。

### Combine

## Reduce Task

ReduceTask具体可以拆为三个过程：

Shuffle -> Sort-> Reduce

Reduce有三个progress：copy阶段，sort阶段和reduce阶段。

run逻辑：

跑reduce task前，同样要先进行jobCleanupTask，jobSetupTask和taskCleanupTask。

在copy阶段和sort阶段，确定framework是local还是某个job tracker，如果不是local，需要new出CombineOutputCollector，ShuffleConsumerPlugin以获得mapper端的数据；如果是local，则没有copy阶段，直接对本地文件进行merge操作。

接下来进入reduce。

Reduce逻辑：

new出TaskAttemptContext和Reducer，reduce.run(context)

# Common

Hadoop Common里的主要内容。

## Hadoop Configuration

对java.util.Properties类的认识。

Properties是基于Hashtable的。

其他能力更丰富的第三方工具集：Apache Jakarta Commons的Configuration

Hadoop自己的org.apache.hadoop.conf.Configuration类

XML形式；new Configuration之后可以add多个Resources；配置之间字段可覆盖；

Configurable接口

## 序列化

三种用途：

作为一种持久化格式；

作为一种通信数据格式；

作为一种拷贝、克隆机制；

### Java序列化

只要实现Serializable接口

在某种OutputStream对象的基础上创建一个对象流ObjectOutputStream，调用writeObject()。写入到OutputStream里，

Java内建序列化的问题是他会额外写入更多的信息：这个对象的类、类签名、类的所有非暂态和非静态成员的值，以及他所有的父类都要被写入

### Hadoop序列化机制

在OutputStream对象基础上创建DataOutputStream，然后由对象自己使用write()写进流里，最后调用DataOutputStream的close()。

在Java的序列化机制中，反序列化过程会不断地创建新的对象，但在Hadoop序列化机制的反序列化过程中，用户可以复用对象。

#### Writable体系

Writable接口，提供两个基础方法

**void** write(DataOutput out) **throws** IOException;

**void** readFields(DataInput in) **throws** IOException;

第一个是写（序列化）方法，第二个是读（反序列化）方法，操作对象是DataOutput/DataInput

DataInput和DataOutput是java的类，主要用于操作java的基础类型与二进制流之间的读写转化

WritableComparable继承Writable和Comparable，WritableComparable体系大都是基础类型

基础类型的读写可以利用DataInput/DataOutput完成。

WritableComparator类继承自RawComparator，用于WritableComparable的比较。RawComparator需要子类实现如下方法，

**public** **int** compare(**byte**[] b1, **int** s1, **int** l1, **byte**[] b2, **int** s2, **int** l2);

s1和s2是object在byte数组里的start index，l1和l2是长度。

WritableComparable子类会注册到WritableComparator类里

ObjectWritable

WritableFactories可注册WritableFactory接口的实现类

#### Serializer框架

提供简单的序列化框架API，集成各种序列化实现， org.apache.hadoop.io.serializer包

Serialization<T>接口

**boolean** accept(Class<?> c);

Serializer<**T**> getSerializer(Class<**T**> c);

Deserializer<**T**> getDeserializer(Class<**T**> c);

Serializer<T>接口

**void** open(OutputStream out) **throws** IOException;

**void** serialize(**T** t) **throws** IOException;

**void** close() **throws** IOException;

Deserializer<T>接口

**void** open(InputStream in) **throws** IOException;

**T** deserialize(**T** t) **throws** IOException;

**void** close() **throws** IOException;

实现有：

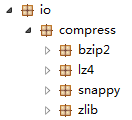
Java实现（Java Serialization类）

Avro实现（org.apache.hadoop.io.serializer.avro包）

## 压缩

org.apache.hadoop.io.compress包下。

2.2.0里面带的实现：



主要内容涉及：编码解码器、压缩器解压缩器、压缩流解压缩流

CompressionCodec接口使用抽象工厂模式，通过它可以得到某个种类的Compressor和CompressorStream的相关类。所以Snappy, Gzip等都实现了自己的Codec类。

CompressionCodecFactory类使用工厂模式，用来为不同的文件名产生不同种类的codec

CodecPool里有对google common里的LoadingCache的使用方式

### CompressorCodec

### Compressor

调用使用顺序

### CompressorStream

### Java本地方法调用

Java内置gzip压缩效率低，使用native声明本地方法，load某些(如Snappy).so文件，通过JNI的方式调用c语言的压缩实现。

具体细节可以看书里面的分析。

### Snappy压缩功能的实现

分析几个关键函数的配合和使用

## IO

### sequencefile

Writer类，两种实现

Writer.append

sync是什么操作

每个block和block之间都会有一个同步点(sync)，

<http://yaoyinjie.blog.51cto.com/3189782/922368>

## RPC：远程过程调用

### RPC基础

#### RPC原理

#### RPC机制的实现

Stub/Skeleton

#### Java RMI

介绍了RMI的实现，实现中涉及到的类体系，还包括底层涉及到的jdk里的工具

具体实例。

### Java动态代理

Java动态代理是Hadoop RPC实现的基础。

动态代理相关内容在java.lang.reflect包下，主要包括Proxy和InvocationHandler

Proxy用来创建代理接口

InvocationHandler用来完成调用转发

具体实例。

### Java NIO简介

Java NIO也是Hadoop RPC实现的基础。

NIO在Java基本套接字的基础上演变来，非阻塞是重要功能之一。

引入选择器（Selector）和通道（Channel）。

#### Buffer

java.nio.buffer缓冲区Buffer提供了一个比流抽象的、更高效和可预测的I/O。Channl通过Buffer传递数据，流通过字节传递。缓冲区和流还有一个区别是，流是单向的，缓冲区里可以往里put，也可以从里面get

对Buffer的内部状态做了清晰的介绍，用以下不定式概括：

0 <= mark <= position <= limit <= capacity

Position和Limit之间可以通过Buffer.remaining()来获得该缓冲区可以读取/写入的元素数

关于Buffer还有一些更详细的内容

#### Channel

对比Channel和Socket

#### Selector

总的来说，NIO这部分结合我以前的使用，为我更好的做了梳理，补充了以前的疑点和盲点。

### Hadoop IPC（Inter-Process Communication）

在org.apache.hadoop.ipc包下面。是Hadoop实现的一个简单RPC。

对比RMI，用Hadoop IPC实现一个例子。

接下来更多代码实现角度的分析先略。

#### 代码结构

待续

## Hadoop文件系统

### 文件系统

#### 文件系统实现

块管理：

1. 连续分配；
2. 链接表；
3. 索引链式表

目录管理

存储媒体上的文件系统管理

存储空间管理：  
1. 空闲文件目录；

2. 空闲块链；

3. 位示图

#### 保护控制

读写执行权限(RWX)

存取控制矩阵 -> 存取控制表和权限表

### Linux文件系统

#### Linux本地文件系统

i-node的元信息存储，间接块技术使得实现支持大文件的同时，保证了小文件的访问效率。

文件和目录的信息都存在i-node里

#### 虚拟文件系统、保护机制、文件系统API

VFS是由面向对象的思想发展起来的，提供了一个抽象基类，派生的子类支持具体的文件系统。

### 分布式文件系统

几个关键字：透明、分布、负载、扩充、副本等等。具体有一堆XX透明性来定义其特性。

遵循传统的客户机-服务器体系结构构造：NFS

基于集群的分布式文件系统：HDFS，GFS

基于P2P（完全分散式的解决方案）：麻省理工学院的Ivy，构建在Chord DHT（Distributed Hash Table）基础上

#### NFS

最初SUN公司开发

基本思想：

每个文件服务器都提供其本地文件系统的一个标准化视图。

换句话说，无论本地文件系统如何实现，每个NFS服务器通过一个通信协议，支持相同的模型，该协议允许客户以完全相同的方法访问存储在一个服务器上的文件，从而允许客户进程去共享公用的文件系统。

### Java文件系统

Java的java.io.File包封装了对用户机器文件系统操作的功能。

Java流操作功能和类继承体系

### Hadoop抽象文件系统

#### 文件系统API

对比java.io.FileSystem，Hadoop实现的org.apache.hadoop.fs.FileSystem

介绍其接口和相关内容，其一是处理文件和目录的相关事务，其二是读写文件数据

#### 输入/输出流

### Hadoop具体文件系统

# HDFS