# 详细探究Spark的shuffle实现

architecture (8) (/categories.html#architecture-ref)

spark 9 (/tags.html#spark-ref) shuffle 1 (/tags.html#shuffle-ref)

mapreduce 2 (/tags.html#mapreduce-ref) cloud 8 (/tags.html#cloud-ref)

**04 January 2014**

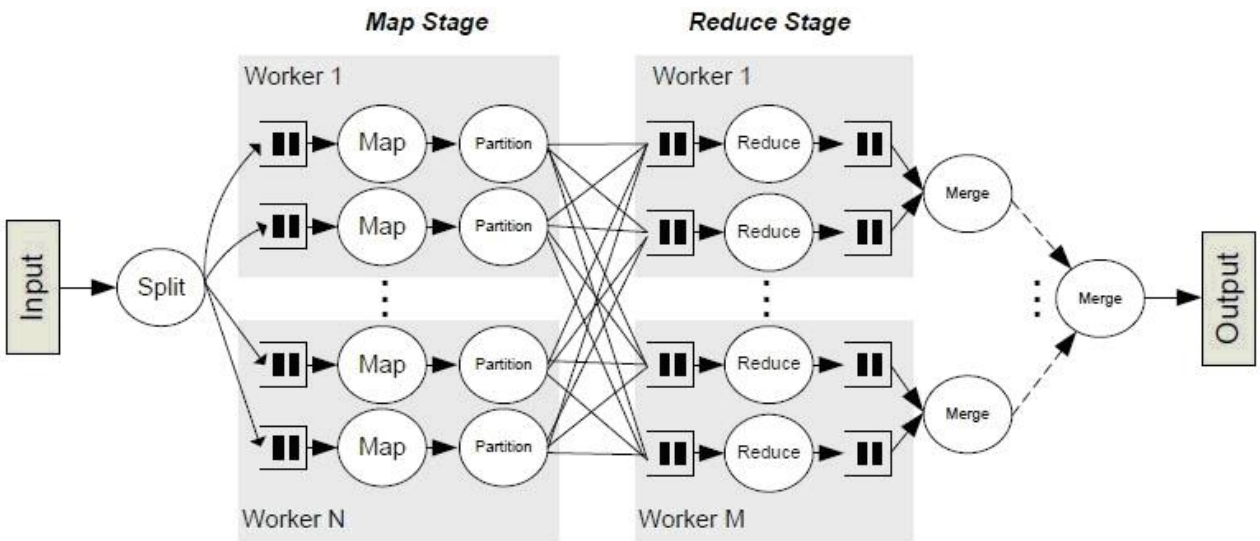
## Background

在MapReduce框架中，shuffle是连接Map和Reduce之间的桥梁，Map的输出要用到Reduce中必须经过shuffle这个环节，shuffle的性能高低直接影响了整个程序的性能和吞吐量。Spark作为MapReduce框架的一种实现，自然也实现了shuffle的逻辑，本文就深入研究Spark的shuffle是如何实现的，有什么优缺点，与Hadoop MapReduce的shuffle有什么不同。

# Shuffle

Shuffle是MapReduce框架中的一个特定的phase，介于Map phase和Reduce phase之间，当Map的输出结果要被Reduce使用时，输出结果需要按key哈希，并且分发到每一个Reducer上去，这个过程就是shuffle。由于shuffle涉及到了磁盘的读写和网络的传输，因此shuffle性能的高低直接影响到了整个程序的运行效率。

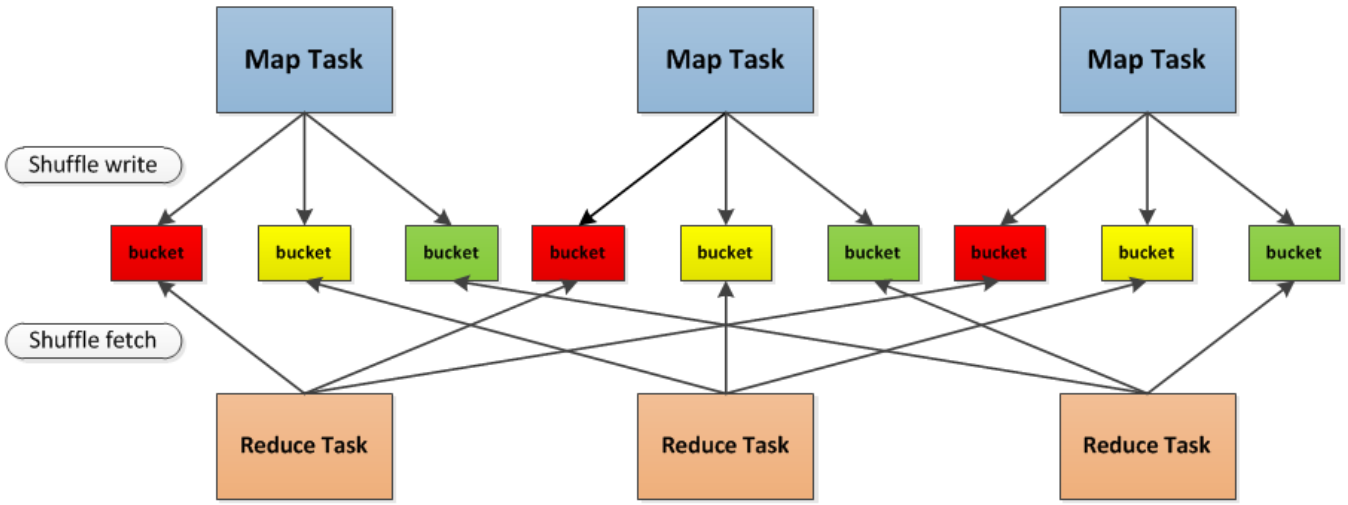
下面这幅图清晰地描述了MapReduce算法的整个流程，其中shuffle phase是介于Map phase和Reduce phase之间。



概念上shuffle就是一个沟通数据连接的桥梁，那么实际上shuffle这一部分是如何实现的的呢，下面我们就以Spark为例讲一下shuffle在Spark中的实现。

# Spark Shuffle进化史

先以图为例简单描述一下Spark中shuffle的整一个流程：




- 首先每一个Mapper会根据Reducer的数量创建出相应的bucket，bucket的数量是$M \times R$，其中$M$是Map的个数，$R$是Reduce的个数。
- 其次Mapper产生的结果会根据设置的partition算法填充到每个bucket中去。这里的partition算法是可以自定义的，当然默认的算法是根据key哈希到不同的bucket中去。
- 当Reducer启动时，它会根据自己task的id和所依赖的Mapper的id从远端或是本地的block manager中取得相应的bucket作为Reducer的输入进行处理。

这里的bucket是一个抽象概念，在实现中每个bucket可以对应一个文件，可以对应文件的一部分或是其他等。

接下来我们分别从**shuffle write**和**shuffle fetch**这两块来讲述一下Spark的shuffle进化史。

# Shuffle Write

在Spark 0.6和0.7的版本中，对于shuffle数据的存储是以文件的方式存储在block manager中，与`rdd.persist(StorageLevel.DISk_ONLY)`采取相同的策略，可以参看：

```
override def run(attemptId: Long): MapStatus = {
  val numOutputSplits = dep.partitioner.numPartitions

  ...
    // Partition the map output.
    val buckets = Array.fill(numOutputSplits)(new ArrayBuffer[(Any, Any)])
    for (elem <- rdd.iterator(split, taskContext)) {
      val pair = elem.asInstanceOf[(Any, Any)]
      val bucketId = dep.partitioner.getPartition(pair._1)
      buckets(bucketId) += pair
    }

    ...

    val blockManager = SparkEnv.get.blockManager
    for (i <- 0 until numOutputSplits) {
      val blockId = "shuffle_" + dep.shuffleId + "_" + partition + "_" + i
      // Get a Scala iterator from Java map
      val iter: Iterator[(Any, Any)] = buckets(i).iterator
      val size = blockManager.put(blockId, iter, StorageLevel.DISK_ONLY, false)
      totalBytes += size
    }
  ...
}
```

我已经将一些干扰代码删去。可以看到Spark在每一个Mapper中为每个Reducer创建一个bucket，并将RDD计算结果放进bucket中。需要注意的是每个bucket是一个`ArrayBuffer`，也就是说Map的输出结果是会先存储在内存。

接着Spark会将ArrayBuffer中的Map输出结果写入block manager所管理的磁盘中，这里文件的命名方式为：`shuffle_ + shuffle_id + "_" + map partition id + "_" + shuffle partition id`。

早期的shuffle write有两个比较大的问题：

1. Map的输出必须先全部存储到内存中，然后写入磁盘。这对内存是一个非常大的开销，当内存不足以存储所有的Map output时就会出现OOM。
2. 每一个Mapper都会产生Reducer number个shuffle文件，如果Mapper个数是1k，Reducer个数也是1k，那么就会产生1M个shuffle文件，这对于文件系统是一个非常大的负担。同时在shuffle数据量不大而shuffle文件又非常多的情况下，随机写也会严重降低IO的性能。

在Spark 0.8版本中，shuffle write采用了与RDD block write不同的方式，同时也为shuffle write单独创建了`ShuffleBlockManager`，部分解决了0.6和0.7版本中遇到的问题。

首先我们来看一下Spark 0.8的具体实现：

```
override def run(attemptId: Long): MapStatus = {

  ...

  val blockManager = SparkEnv.get.blockManager
  var shuffle: ShuffleBlocks = null
  var buckets: ShuffleWriterGroup = null
```

```
 8.
 9.    try {
10.      // Obtain all the block writers for shuffle blocks.
11.      val ser = SparkEnv.get.serializerManager.get(dep.serializerClass)
12.      shuffle = blockManager.shuffleBlockManager.forShuffle(dep.shuffleId, numOutputSplits,
     ser)
13.      buckets = shuffle.acquireWriters(partition)
14.
15.      // Write the map output to its associated buckets.
16.      for (elem <- rdd.iterator(split, taskContext)) {
17.        val pair = elem.asInstanceOf[Product2[Any, Any]]
18.        val bucketId = dep.partitioner.getPartition(pair._1)
19.        buckets.writers(bucketId).write(pair)
20.      }
21.
22.      // Commit the writes. Get the size of each bucket block (total block size).
23.      var totalBytes = 0L
24.      val compressedSizes: Array[Byte] = buckets.writers.map { writer:   BlockObjectWriter
     =>
25.        writer.commit()
26.        writer.close()
27.        val size = writer.size()
28.        totalBytes += size
29.        MapOutputTracker.compressSize(size)
30.      }
31.
32.      ...
33.
34.    } catch { case e: Exception =>
35.      // If there is an exception from running the task, revert the partial writes
36.      // and throw the exception upstream to Spark.
37.      if (buckets != null) {
38.        buckets.writers.foreach(_.revertPartialWrites())
39.      }
40.      throw e
41.    } finally {
42.      // Release the writers back to the shuffle block manager.
43.      if (shuffle != null && buckets != null) {
44.        shuffle.releaseWriters(buckets)
45.      }
46.      // Execute the callbacks on task completion.
47.      taskContext.executeOnCompleteCallbacks()
48.      }
49.    }
50.  }
```

在这个版本中为shuffle write添加了一个新的类`ShuffleBlockManager`，由`ShuffleBlockManager`来分配和管理bucket。同时`ShuffleBlockManager`为每一个bucket分配一个`DiskObjectWriter`，每个write handler拥有默认100KB的缓存，使用这个write handler将Map output写入文件中。可以看到现在的写入方式变

为`buckets.writers(bucketId).write(pair)`，也就是说Map output的key-value pair是逐个写入到磁盘而不是预先把所有数据存储在内存中在整体flush到磁盘中去。

`ShuffleBlockManager`的代码如下所示：

```
1.  private[spark]
2.  class ShuffleBlockManager(blockManager: BlockManager) {
3.
4.    def forShuffle(shuffleId: Int, numBuckets: Int, serializer: Serializer): ShuffleBlocks
    = {
5.      new ShuffleBlocks {
6.        // Get a group of writers for a map task.
7.        override def acquireWriters(mapId: Int): ShuffleWriterGroup = {
8.          val bufferSize = System.getProperty("spark.shuffle.file.buffer.kb", "100").toInt
    * 1024
9.          val writers = Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
10.           val blockId = ShuffleBlockManager.blockId(shuffleId, bucketId, mapId)
11.           blockManager.getDiskBlockWriter(blockId, serializer, bufferSize)
12.         }
13.         new ShuffleWriterGroup(mapId, writers)
14.       }
15.
16.       override def releaseWriters(group: ShuffleWriterGroup) = {
17.         // Nothing really to release here.
18.       }
19.     }
20.   }
21. }
```

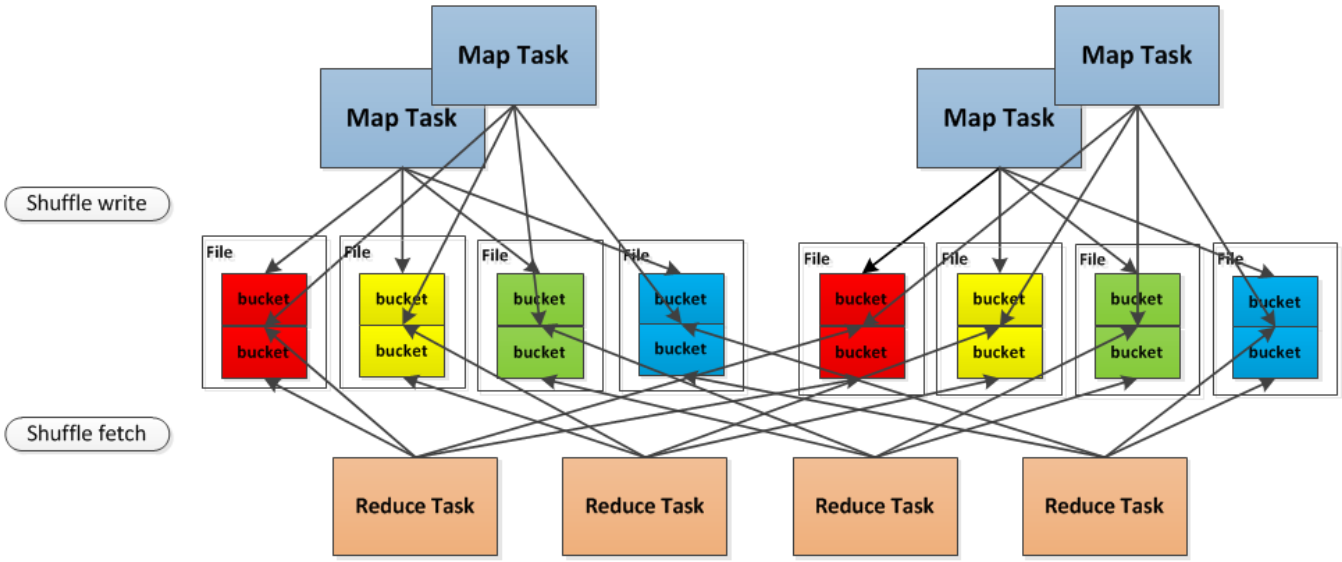
Spark 0.8显著减少了shuffle的内存压力，现在Map output不需要先全部存储在内存中，再flush到硬盘，而是record-by-record写入到磁盘中。同时对于shuffle文件的管理也独立出新的`ShuffleBlockManager`进行管理，而不是与rdd cache文件在一起了。

但是这一版Spark 0.8的shuffle write仍然有两个大的问题没有解决：

- 首先依旧是shuffle文件过多的问题，shuffle文件过多一是会造成文件系统的压力过大，二是会降低IO的吞吐量。
- 其次虽然Map output数据不再需要预先在内存中evaluate显著减少了内存压力，但是新引入的`DiskObjectWriter`所带来的buffer开销也是一个不容小视的内存开销。假定我们有1k个Mapper和1k个Reducer，那么就会有1M个bucket，于此同时就会有1M个write handler，而每一个write handler默认需要100KB内存，那么总共需要100GB的内存。这样的话仅仅是buffer就需要这么多的内存，内存的开销是惊人的。当然实际情况下这1k个Mapper是分时运行的话，所需的内存就只有`cores * reducer numbers * 100KB`大小了。但是reducer数量很多的话，这个buffer的内存开销也是蛮厉害的。

为了解决shuffle文件过多的情况，Spark 0.8.1引入了新的shuffle consolidation，以期显著减少shuffle文件的数量。

首先我们以图例来介绍一下shuffle consolidation的原理。




假定该job有4个Mapper和4个Reducer，有2个core，也就是能并行运行两个task。我们可以算出Spark的shuffle write共需要16个bucket，也就有了16个write handler。在之前的Spark版本中，每一个bucket对应的是一个文件，因此在这里会产生16个shuffle文件。

而在shuffle consolidation中每一个bucket并非对应一个文件，而是对应文件中的一个segment，同时shuffle consolidation所产生的shuffle文件数量与Spark core的个数也有关系。在上面的图例中，job的4个Mapper分为两批运行，在第一批2个Mapper运行时会申请8个bucket，产生8个shuffle文件；而在第二批Mapper运行时，申请的8个bucket并不会再产生8个新的文件，而是追加写到之前的8个文件后面，这样一共就只有8个shuffle文件，而在文件内部这有16个不同的segment。因此从理论上讲shuffle consolidation所产生的shuffle文件数量为$C \times R$，其中$C$是Spark集群的core number，$R$是Reducer的个数。

需要注意的是当$M = C$时shuffle consolidation所产生的文件数和之前的实现是一样的。

Shuffle consolidation显著减少了shuffle文件的数量，解决了之前版本一个比较严重的问题，但是writer handler的buffer开销过大依然没有减少，若要减少writer handler的buffer开销，我们只能减少Reducer的数量，但是这又会引入新的问题，下文将会有详细介绍。

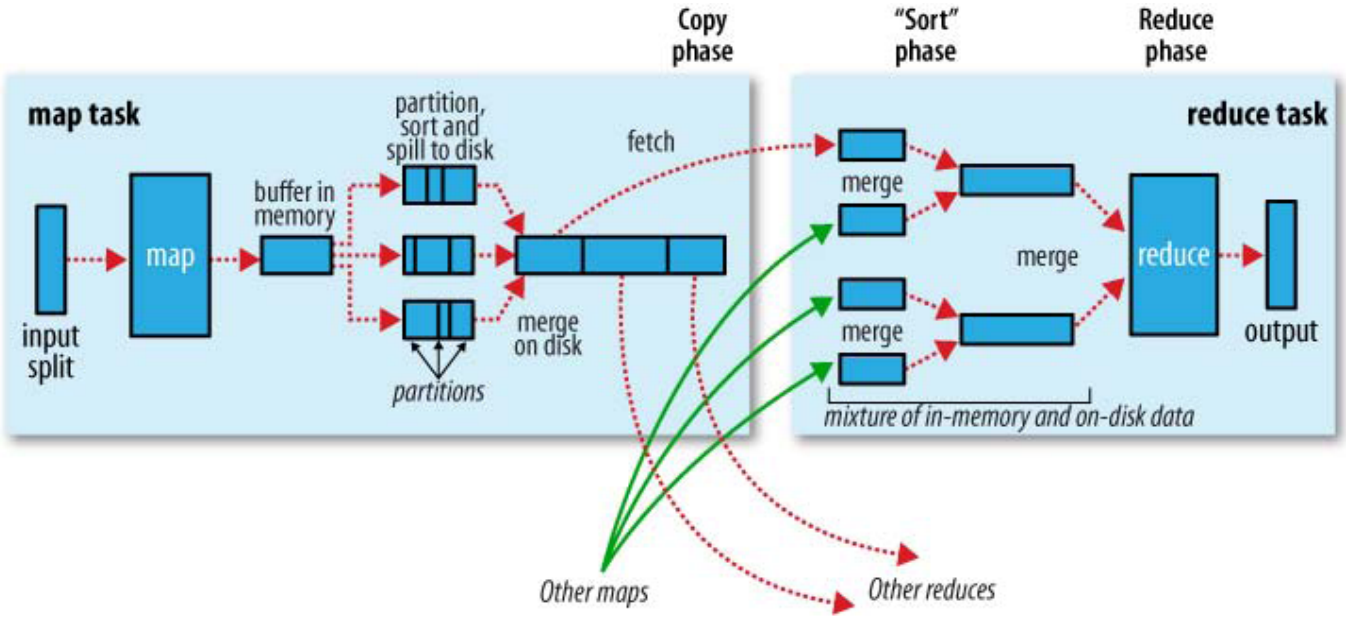
讲完了shuffle write的进化史，接下来要讲一下shuffle fetch了，同时还要讲一下Spark的aggregator，这一块对于Spark实际应用的性能至关重要。

# Shuffle Fetch and Aggregator

Shuffle write写出去的数据要被Reducer使用，就需要shuffle fetcher将所需的数据fetch过来，这里的fetch包括本地和远端，因为shuffle数据有可能一部分是存储在本地的。Spark对shuffle fetcher实现了两套不同的框架：NIO通过socket连接去fetch数据；OIO通过netty server去fetch数据。分别对应的类是`BasicBlockFetcherIterator`和`NettyBlockFetcherIterator`。

在Spark 0.7和更早的版本中，只支持`BasicBlockFetcherIterator`，而`BasicBlockFetcherIterator`在shuffle数据量比较大的情况下performance始终不是很好，无法充分利用网络带宽，为了解决这个问题，添加了新的shuffle fetcher来试图取得更好的性能。对于早期shuffle性能的评测可以参看Spark usergroup (https://groups.google.com/forum/#!msg/shark-users/lHOb2u5HXSk/huTWyosl1n4J)。当然现在`BasicBlockFetcherIterator`的性能也已经好了很多，使用的时候可以对这两种实现都进行测试比较。

接下来说一下aggregator。我们都知道在Hadoop MapReduce的shuffle过程中，shuffle fetch过来的数据会进行merge sort，使得相同key下的不同value按序归并到一起供Reducer使用，这个过程可以参看下图：




所有的merge sort都是在磁盘上进行的，有效地控制了内存的使用，但是代价是更多的磁盘IO。

那么Spark是否也有merge sort呢，还是以别的方式实现，下面我们就细细说明。

首先虽然Spark属于MapReduce体系，但是对传统的MapReduce算法进行了一定的改变。Spark假定在大多数用户的case中，shuffle数据的sort不是必须的，比如word count，强制地进行排序只会使性能变差，因此Spark并不在Reducer端做merge sort。既然没有merge sort那Spark是如何进行reduce的呢？这就要说到aggregator了。

aggregator本质上是一个hashmap，它是以map output的key为key，以任意所要combine的类型为value的hashmap。当我们在做word count reduce计算count值的时候，它会将shuffle fetch到的每一个key-value pair更新或是插入到hashmap中(若在hashmap中没有查找到，则插入其中；若查找到则更新value值)。这样就不需要预先把所有的key-value进行merge sort，而是来一个处理一个，省下了外部排序这一步骤。但同时需要注意的是reducer的内存必须足以存放这个partition的所有key和count值，因此对内存有一定的要求。

在上面word count的例子中，因为value会不断地更新，而不需要将其全部记录在内存中，因此内存的使用还是比较少的。考虑一下如果是group by key这样的操作，Reducer需要得到key对应的所有value。在Hadoop MapReduce中，由于有了merge sort，因此给予Reducer的数据已经是group by key了，而Spark没有这一步，因此需要将key和对应的value全部存放在hashmap中，并将value合并成一个array。可以想象为了能够存放所有数据，用户必须确保每一个partition足够小到内存能够容纳，这对于内存是一个非常严峻的考验。因此Spark文档中建议用户涉及到这类操作的时候尽量增加partition，也就是增加Mapper和Reducer的数量。

增加Mapper和Reducer的数量固然可以减小partition的大小，使得内存可以容纳这个partition。但是我们在shuffle write中提到，bucket和对应于bucket的write handler是由Mapper和Reducer的数量决定的，task越多，bucket就会增加的更多，由此带来write handler所需的buffer也会更多。在一方面我们为了减少内存的使用采取了增加task数量的策略，另一方面task数量增多又会带来buffer开销更大的问题，因此陷入了内存使用的两难境地。

为了减少内存的使用，只能将aggregator的操作从内存移到磁盘上进行，Spark社区也意识到了Spark在处理数据规模远远大于内存大小时所带来的问题。因此PR303 (https://github.com/apache/incubator-spark/pull/303)提供了外部排序的实现方案，相信在Spark 0.9 release的时候，这个patch应该能merge进去，到时候内存的使用量可以显著地减少。

# End

本文详细地介绍了Spark的shuffle实现是如何进化的，以及遇到问题解决问题的过程。shuffle作为Spark程序中很重要的一个环节，直接影响了Spark程序的性能，现如今的Spark版本虽然shuffle实现还存在着种种问题，但是相比于早期版本，已经有了很大的进步。开源代码就是如此不停地迭代推进，随着Spark的普及程度

越来越高，贡献的人越来越多，相信后续的版本会有更大的提升。

← Previous (/algorithm/2014/01/02/simrank-mapreduce-survey) | Archive (/archive.html) | Next →




**CATEGORIES**

test (1) (/categories.html#test-ref)

architecture (8) (/categories.html#architecture-ref)

functional programming (1) (/categories.html#functional programming-ref)

algorithm (1) (/categories.html#algorithm-ref)

**LINKS**

阮一峰的网络日志 (http://www.ruanyifeng.com/blog/)

刘未鹏 (http://mindhacks.cn/)

酷壳 (http://coolshell.cn/)

BeiYuu.com (http://beiyuu.com/)

**MY FAVORITES**