**Spark Core**

|  |  |
| --- | --- |
| **审核人** |  |
| **重要性** | 高 |
| **紧迫性** | 低 |
| **拟制人** | 张包峰 |
| **提交日期** |  |

**作者：张包峰**

(版权所有,翻版必究)

**修改记录**

|  |  |  |
| --- | --- | --- |
| **更新时间** | **变更内容** | **变更理由** |
| 2014-04-20 | 1. RDD 基础 2. 任务模型 3. 任务执行模型 | 阶段性产出 |
|  |  |  |
|  |  |  |
|  |  |  |

[**作者：张包峰** 1](#_Toc392841740)

[1 SparkContext 5](#_Toc392841741)

[1.1 主要方法 5](#_Toc392841742)

[1.1.1 制造RDD 5](#_Toc392841743)

[1.1.2 广播 5](#_Toc392841744)

[1.2 SparkEnv及其内关键部件 5](#_Toc392841745)

[1.2.1 Serializer 6](#_Toc392841746)

[1.2.2 CacheManager 6](#_Toc392841747)

[1.2.3 SecurityManager 6](#_Toc392841748)

[1.2.4 MapOutputTracker 6](#_Toc392841749)

[1.2.5 ShuffleFetcher 7](#_Toc392841750)

[1.2.6 BroadcastManager 7](#_Toc392841751)

[1.2.7 BlockManager 7](#_Toc392841752)

[1.3 其他部件 7](#_Toc392841753)

[1.3.1 ContextCleaner 7](#_Toc392841754)

[2 RDD 8](#_Toc392841755)

[2.1 Dependency 8](#_Toc392841756)

[2.1.1 RDD中的使用 9](#_Toc392841757)

[2.1.2 宽依赖和窄依赖 9](#_Toc392841758)

[2.2 Partition 10](#_Toc392841759)

[2.3 Partitioner 11](#_Toc392841760)

[2.3.1 默认Partitioner 11](#_Toc392841761)

[2.3.2 HashPartitioner 12](#_Toc392841762)

[2.3.3 RangePartitioner 12](#_Toc392841763)

[2.4 Persist 13](#_Toc392841764)

[2.5 Checkpoint 14](#_Toc392841765)

[2.6 API 15](#_Toc392841766)

[2.6.1 Transformations 15](#_Toc392841767)

[2.6.2 Actions 17](#_Toc392841768)

[2.7 Sub RDDs 18](#_Toc392841769)

[2.7.1 CheckpointRDD 18](#_Toc392841770)

[2.7.2 HadoopRDD 19](#_Toc392841771)

[2.7.3 NewHadoopRDD 19](#_Toc392841772)

[2.7.4 WholeTextFileRDD 20](#_Toc392841773)

[3 Storage 22](#_Toc392841774)

[3.1 BlockManager 22](#_Toc392841775)

[3.1.1 读取流程 22](#_Toc392841776)

[3.2 Tachyon支持 22](#_Toc392841777)

[4 Broadcast机制 22](#_Toc392841778)

[4.1 BroadcastManager 23](#_Toc392841779)

[4.2 Broadcast的两种实现 24](#_Toc392841780)

[4.2.1 HTTP 24](#_Toc392841781)

[4.2.2 Torrent 25](#_Toc392841782)

[5 Spark任务模型 26](#_Toc392841783)

[5.1 Task 26](#_Toc392841784)

[5.2 TaskContext 27](#_Toc392841785)

[5.3 Task两种实现 27](#_Toc392841786)

[5.3.1 ShuffleMapTask及Shuffle模型 27](#_Toc392841787)

[5.3.2 ResultTask 31](#_Toc392841788)

[5.4 进程模型 vs. 线程模型 32](#_Toc392841789)

[6 Spark任务执行模型 32](#_Toc392841790)

[6.1 DAGScheduler 32](#_Toc392841791)

[6.1.1 Event生命周期 34](#_Toc392841792)

[6.1.2 Event处理 34](#_Toc392841793)

[6.2 TaskSchedulerImpl 36](#_Toc392841794)

[6.2.1 维护Task集模型：SchedulableBuilder 37](#_Toc392841795)

[6.2.2 Task集维护类：TaskSetManager 37](#_Toc392841796)

[6.2.3 获取执行结果：TaskResultGetter 37](#_Toc392841797)

[7 SchedulerBackend 39](#_Toc392841798)

[7.1 Local模式 40](#_Toc392841799)

[7.2 粗粒度模式 40](#_Toc392841800)

[7.2.1 Standalone Deploy 40](#_Toc392841801)

[7.2.2 Mesos粗粒度 40](#_Toc392841802)

[7.3 Mesos细粒度模式 40](#_Toc392841803)

[7.4 YARN 41](#_Toc392841804)

[8 其他类 41](#_Toc392841805)

[8.1 SerializableWritable 41](#_Toc392841806)

# SparkContext

SparkContext下面的东西比较杂乱，按照主要方法和主要部件介绍下SparkContext提供的东西。

## 主要方法

### 制造RDD

### 广播

## SparkEnv及其内关键部件

**class** SparkEnv (

**val** executorId: String,

**val** actorSystem: ActorSystem,

**val** serializer: Serializer,

**val** closureSerializer: Serializer,

**val** cacheManager: CacheManager,

**val** mapOutputTracker: MapOutputTracker,

**val** shuffleFetcher: ShuffleFetcher,

**val** broadcastManager: BroadcastManager,

**val** blockManager: BlockManager,

**val** connectionManager: ConnectionManager,

**val** securityManager: SecurityManager,

**val** httpFileServer: HttpFileServer,

**val** sparkFilesDir: String,

**val** metricsSystem: MetricsSystem,

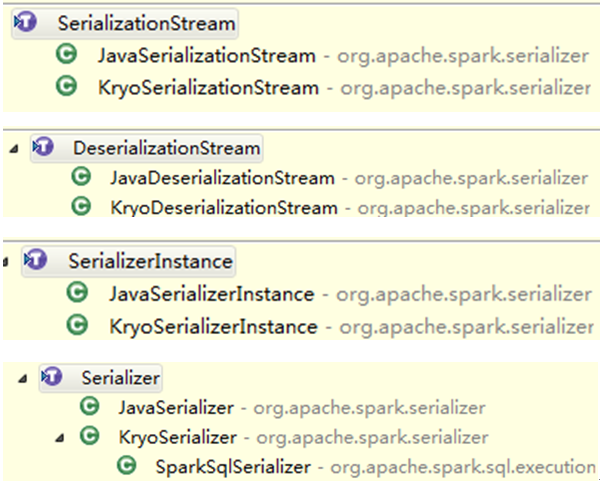
**val** conf: SparkConf) **extends** Logging {

### Serializer

Spark提供serializer trait，要求继承类通过newInstance()方法返回SerializerInstance实例。使用时要保证线程安全。

SerializerInstance提供了从ByteBuffer或Input/OutputStream里序列化、反序列化的方法。在处理Stream的时候，得到的是SerializationStream和DeserializationStream trait。前者的writeAll方式接收输入Iterator，将iterator的每一个next()进行writeObject()；后者的读数据方式是返回成一个Iterator，每当iterator.getNext()的时候就进行一次readObject操作。

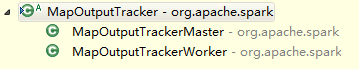
在实现方面，Spark Core内提供了Java序列化类和Kryo序列化类。各自实现了上面所说的四种trait。



### CacheManager

### SecurityManager

### MapOutputTracker



### ShuffleFetcher

### BroadcastManager

见Broadcast一节。

### BlockManager

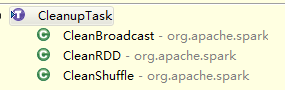
见Storage一节。

## 其他部件

### ContextCleaner

异步清理RDD，或shuffle，或broadcast state。

对应三种CleanupTask：



三种清理任务具体执行代码如下，可见其委托逻辑

/\*\* Perform RDD cleanup. \*/

**def** doCleanupRDD(rddId: Int, blocking: Boolean) {

**try** {

sc.unpersistRDD(rddId, blocking)

listeners.foreach(\_.rddCleaned(rddId))

} **catch** {

**case** e: Exception => logError("Error cleaning RDD " + rddId, e)

}

}

/\*\* Perform shuffle cleanup, asynchronously. \*/

**def** doCleanupShuffle(shuffleId: Int, blocking: Boolean) {

**try** {

mapOutputTrackerMaster.unregisterShuffle(shuffleId)

blockManagerMaster.removeShuffle(shuffleId, blocking)

listeners.foreach(\_.shuffleCleaned(shuffleId))

} **catch** {

**case** e: Exception => logError("Error cleaning shuffle " + shuffleId, e)

}

}

/\*\* Perform broadcast cleanup. \*/

**def** doCleanupBroadcast(broadcastId: Long, blocking: Boolean) {

**try** {

broadcastManager.unbroadcast(broadcastId, **true**, blocking)

listeners.foreach(\_.broadcastCleaned(broadcastId))

} **catch** {

**case** e: Exception => logError("Error cleaning broadcast " + broadcastId, e)

}

}

# RDD

RDD组成：上下文和一组依赖

**abstract** **class** RDD[T: ClassTag](

@transient **private** **var** sc: SparkContext,

@transient **private** **var** deps: Seq[Dependency[\_]]

) **extends** Serializable **with** Logging {

以下需要仔细理清：

A list of Partitions

Function to compute split (sub RDD impl)

A list of Dependencies

Partitioner for K-V RDDs (Optional)

Preferred locations to compute each split on (Optional)

## Dependency

Dependency代表了RDD之间的依赖关系，即血缘

### RDD中的使用

RDD给子类提供了getDependencies方法来制定如何依赖父类RDD

**protected** **def** getDependencies: Seq[Dependency[\_]] = deps

事实上，在获取first parent的时候，子类经常会使用下面这个方法

**protected**[spark] **def** firstParent[U: ClassTag] = {

dependencies.head.rdd.asInstanceOf[RDD[U]]

}

可以看到，Seq里的第一个dependency应该是直接的parent，从而从第一个dependency类里获得了rdd，这个rdd就是父RDD。

一般的RDD子类都会这么实现compute和getPartition方法，以SchemaRDD举例：

**override** **def** compute(split: Partition, context: TaskContext): Iterator[Row] =

firstParent[Row].compute(split, context).map(\_.copy())

**override** **def** getPartitions: Array[Partition] = firstParent[Row].partitions

compute()方法调用了第一个父类的compute，返回RDD copy结果

getPartitions返回的就是第一个父类的partitions

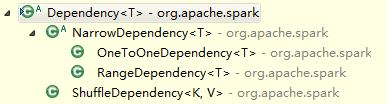
下面看一下Dependency类及其子类的实现。

### 宽依赖和窄依赖

**abstract** **class** Dependency[T](**val** rdd: RDD[T]) **extends** Serializable

Dependency里传入的rdd，就是父RDD本身。

继承结构如下：



NarrowDependency代表窄依赖，即父RDD的分区，最多被子RDD的一个分区使用。所以支持并行计算。

子类需要实现方法：

**def** getParents(partitionId: Int): Seq[Int]

其实现OneToOneDependency表示父RDD和子RDD的分区依赖是一对一的。

其实现RangeDependency表示在一个range范围内，依赖关系是一对一的，所以初始化的时候会有一个范围，范围外的partitionId，传进去之后返回的是Nil。

下面介绍宽依赖。

**class** ShuffleDependency[K, V](

@transient rdd: RDD[\_ <: Product2[K, V]],

**val** partitioner: Partitioner,

**val** serializer: Serializer = **null**)

**extends** Dependency(rdd.asInstanceOf[RDD[Product2[K, V]]]) {

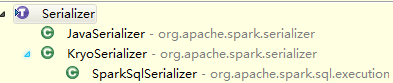
// 上下文增量定义的Id

**val** shuffleId: Int = rdd.context.newShuffleId()

rdd.sparkContext.cleaner.foreach(\_.registerShuffleForCleanup(**this**))

}

宽依赖针对的RDD是KV形式的，需要一个partitioner指定分区方式(下一节介绍)，需要一个序列化工具类，序列化工具目前的实现如下：

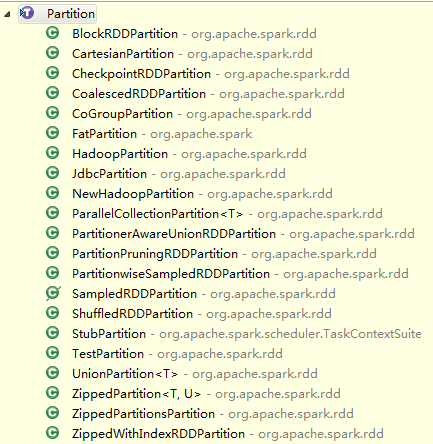


宽依赖和窄依赖对失败恢复时候的recompute有不同程度的影响，宽依赖可能是要全部计算的。

## Partition

Partition具体表示RDD每个数据分区。

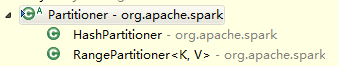
Partition提供trait类，内含一个index和hashCode()方法，具体子类实现与RDD子类有关，种类如下：



在分析每个RDD子类的时候再涉及。

## Partitioner

Partitioner决定KV形式的RDD如何根据key进行partition



**abstract** **class** Partitioner **extends** Serializable {

**def** numPartitions: Int

**def** getPartition(key: Any): Int

}

在ShuffleDependency里对应一个Partitioner，来完成宽依赖下，子RDD如何获取父RDD。

### 默认Partitioner

Partitioner的伴生对象提供defaultPartitioner方法，逻辑为：

传入的RDD（至少两个）中，遍历（顺序是partition数目从大到小）RDD，如果已经有Partitioner了，就使用。如果RDD们都没有Partitioner，则使用默认的HashPartitioner。而HashPartitioner的初始化partition数目，取决于是否设置了spark.default.parallelism，如果没有的话就取RDD中partition数目最大的值。

如果上面这段文字看起来费解，代码如下：

**def** defaultPartitioner(rdd: RDD[\_], others: RDD[\_]\*): Partitioner = {

**val** bySize = (Seq(rdd) ++ others).sortBy(\_.partitions.size).reverse

**for** (r <- bySize **if** r.partitioner.isDefined) {

**return** r.partitioner.get

}

**if** (rdd.context.conf.contains("spark.default.parallelism")) {

**new** HashPartitioner(rdd.context.defaultParallelism)

} **else** {

**new** HashPartitioner(bySize.head.partitions.size)

}

}

### HashPartitioner

HashPartitioner基于java的Object.hashCode。会有个问题是Java的Array有自己的hashCode，不基于Array里的内容，所以RDD[Array[\_]]或RDD[(Array[\_], \_)]使用HashPartitioner会有问题。

顾名思义，getPartition方法实现如下

**def** getPartition(key: Any): Int = key **match** {

**case** **null** => 0

**case** \_ => Utils.nonNegativeMod(key.hashCode, numPartitions)

}

### RangePartitioner

RangePartitioner处理的KV RDD要求Key是可排序的，即满足Scala的Ordered[K]类型。所以它的构造如下：

**class** RangePartitioner[K : Ordering : ClassTag, V](

partitions: Int,

@transient rdd: RDD[\_ <: Product2[K,V]],

**private** **val** ascending: Boolean = **true**)

**extends** Partitioner {

内部会计算一个rangBounds(上界)，在getPartition的时候，如果rangBounds size小于1000，则逐个遍历获得；否则二分查找获得partitionId。

## Persist

默认cache()过程是将RDD persist在内存里，persist()操作可以为RDD重新指定StorageLevel，

**class** StorageLevel **private**(

**private** **var** useDisk\_ : Boolean,

**private** **var** useMemory\_ : Boolean,

**private** **var** useOffHeap\_ : Boolean,

**private** **var** deserialized\_ : Boolean,

**private** **var** replication\_ : Int = 1)

**extends** Externalizable {

**object** StorageLevel {

**val** NONE = **new** StorageLevel(**false**, **false**, **false**, **false**)

**val** DISK\_ONLY = **new** StorageLevel(**true**, **false**, **false**, **false**)

**val** DISK\_ONLY\_2 = **new** StorageLevel(**true**, **false**, **false**, **false**, 2)

**val** MEMORY\_ONLY = **new** StorageLevel(**false**, **true**, **false**, **true**)

**val** MEMORY\_ONLY\_2 = **new** StorageLevel(**false**, **true**, **false**, **true**, 2)

**val** MEMORY\_ONLY\_SER = **new** StorageLevel(**false**, **true**, **false**, **false**)

**val** MEMORY\_ONLY\_SER\_2 = **new** StorageLevel(**false**, **true**, **false**, **false**, 2)

**val** MEMORY\_AND\_DISK = **new** StorageLevel(**true**, **true**, **false**, **true**)

**val** MEMORY\_AND\_DISK\_2 = **new** StorageLevel(**true**, **true**, **false**, **true**, 2)

**val** MEMORY\_AND\_DISK\_SER = **new** StorageLevel(**true**, **true**, **false**, **false**)

**val** MEMORY\_AND\_DISK\_SER\_2 = **new** StorageLevel(**true**, **true**, **false**, **false**, 2)

**val** OFF\_HEAP = **new** StorageLevel(**false**, **false**, **true**, **false**)

RDD的persist()和unpersist()操作，都是由SparkContext执行的（SparkContext的persistRDD和unpersistRDD方法）。

Persist过程是把该RDD存在上下文的TimeStampedWeakValueHashMap里维护起来。也就是说，其实persist并不是action，并不会触发任何计算。

Unpersist过程如下，会交给SparkEnv里的BlockManager处理。

**private**[spark] **def** unpersistRDD(rddId: Int, blocking: Boolean = **true**) {

env.blockManager.master.removeRdd(rddId, blocking)

persistentRdds.remove(rddId)

listenerBus.post(SparkListenerUnpersistRDD(rddId))

}

## Checkpoint

RDD Actions api里提供了checkpoint()方法，会把本RDD save到SparkContext CheckpointDir

目录下。建议该RDD已经persist在内存中，否则需要recomputation。

如果该RDD没有被checkpoint过，则会生成新的RDDCheckpointData。RDDCheckpointData类与一个RDD关联，记录了该RDD的checkpoint信息，并且记录checkpoint RDD的一个状态，

[ Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed ]

**private**[spark] **object** CheckpointState **extends** Enumeration {

**type** CheckpointState = Value

**val** Initialized, MarkedForCheckpoint, CheckpointingInProgress, Checkpointed = Value

}

内部有一个doCheckpoint()方法（会被下面调用），执行真正的checkpoint操作，调用时机是在RDD第一次在job里被使用之后。

**doCheckpoint执行逻辑**

真正的checkpoint触发，在RDD私有方法doCheckpoint()里直接调用，调用代码如下。doCheckpoint()会被DAGScheduler调用，且是在此次job里使用这个RDD完毕之后，此时这个RDD就已经被计算或者物化过了。可以看到，会对RDD的父RDD进行递归。

**private**[spark] **def** doCheckpoint() {

**if** (!doCheckpointCalled) {

doCheckpointCalled = **true**

**if** (checkpointData.isDefined) {

checkpointData.get.doCheckpoint()

} **else** {

dependencies.foreach(\_.rdd.doCheckpoint())

}

}

}

RDDCheckpointData的doCheckpoint()方法关键代码如下：

// Create the output path for the checkpoint

**val** path = **new** Path(rdd.context.checkpointDir.get, "rdd-" + rdd.id)

**val** fs = path.getFileSystem(rdd.context.hadoopConfiguration)

**if** (!fs.mkdirs(path)) {

**throw** **new** SparkException(xx)

}

// Save to file, and reload it as an RDD

**val** broadcastedConf = rdd.context.broadcast(

**new** SerializableWritable(rdd.context.hadoopConfiguration))

rdd.context.runJob(rdd, CheckpointRDD.writeToFile[T](path.toString, broadcastedConf) \_)

**val** newRDD = **new** CheckpointRDD[T](rdd.context, path.toString)

**if** (newRDD.partitions.size != rdd.partitions.size) {

**throw** **new** SparkException(xx)

}

runJob最终调的是dagScheduler的runJob。做完后，生成一个CheckpointRDD。

具体CheckpointRDD相关内容可以参考其他章节。

## API

子类需要实现的方法

// 计算某个分区

def compute(split: Partition, context: TaskContext): Iterator[T]

protected def getPartitions: Array[Partition]

// 依赖的父RDD，默认就是返回整个dependency序列

protected def getDependencies: Seq[Dependency[\_]] = deps

protected def getPreferredLocations(split: Partition): Seq[String] = Nil

### Transformations

|  |  |
| --- | --- |
| map[U: ClassTag](f: T => U): RDD[U] | MappedRDD |
| flatMap[U: ClassTag](f: T =>  TraversableOnce[U]): RDD[U] | FlatMappedRDD |
| filter(f: T => Boolean): RDD[T] | FilteredRDD |
| distinct(numPartitions: Int): RDD[T] | map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(\_.\_1) |
| distinct(): RDD[T] | distinct(partitions.size) |
| repartition(numPartitions: Int): RDD[T] | coalesce(numPartitions, shuffle = true) |
| coalesce(numPartitions: Int, shuffle: Boolean = false): RDD[T] | CoalescedRDD |
| sample(withReplacement: Boolean, fraction: Double, seed: Int): RDD[T] | PartitionwiseSampledRDD |
| randomSplit(weights: Array[Double], seed: Long = System.nanoTime): Array[RDD[T]] | PartitionwiseSampledRDD |
| takeSample(withReplacement: Boolean, num: Int, seed: Int): Array[T] |  |
| union(other: RDD[T]): RDD[T] | UnionRDD |
| ++(other: RDD[T]): RDD[T] | 同上 |
| intersection(other: RDD[T]): RDD[T] | this.map().cogroup().filter |
| intersection(other: RDD[T], partitioner: Partitioner): RDD[T] | this.map().cogroup().filter |
| glom(): RDD[Array[T]] | GlommedRDD |
| cartesian[U: ClassTag](other: RDD[U]): RDD[(T, U)] | CartesianRDD |
| groupBy[K: ClassTag](f: T => K): RDD[(K, Iterable[T])] | groupBy[K](f, defaultPartitioner(this)) |
| groupBy[K: ClassTag](f: T => K,  numPartitions: Int): RDD[(K, Iterable[T])] | groupBy(f, new  HashPartitioner(numPartitions)) |
| groupBy[K: ClassTag](f: T => K, p:  Partitioner): RDD[(K, Iterable[T])] |  |
| pipe(command: String): RDD[String] | PipedRDD |
| pipe(command: String, env: Map[String, String]): RDD[String] | PipedRDD |
| mapPartitions[U: ClassTag](  f: Iterator[T] => Iterator[U],  preservesPartitioning: Boolean = false): RDD[U] | MapPartitionsRDD |
| mapPartitionsWithIndex[U: ClassTag](  f: (Int, Iterator[T]) => Iterator[U],  preservesPartitioning: Boolean = false): RDD[U] | MapPartitionsRDD |
| zip[U: ClassTag](other: RDD[U]): RDD[(T, U)] | ZippedRDD |
| zipPartitions[B: ClassTag, V: ClassTag]  (rdd2: RDD[B], preservesPartitioning: Boolean)  (f: (Iterator[T], Iterator[B]) => Iterator[V]): RDD[V] | ZippedPartitionsRDD2 |
|  |  |
|  |  |

### Actions

|  |  |
| --- | --- |
| foreach(f: T => Unit) |  |
| foreachPartition(f: Iterator[T] => Unit) |  |
| collect(): Array[T] |  |
| toLocalIterator: Iterator[T] |  |
| collect[U: ClassTag](f: PartialFunction[T, U]): RDD[U] |  |
| subtract(other: RDD[T]): RDD[T] |  |
| subtract(other: RDD[T], numPartitions: Int): RDD[T] |  |
| subtract(other: RDD[T], p: Partitioner): RDD[T] |  |
| reduce(f: (T, T) => T): T |  |
| fold(zeroValue: T)(op: (T, T) => T): T |  |
| aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U |  |
| count(): Long |  |
| countByValue(): Map[T, Long] |  |
| zipWithIndex(): RDD[(T, Long)] |  |
| zipWithUniqueId(): RDD[(T, Long)] |  |
| take(num: Int): Array[T] |  |
| first(): T |  |
| top(num: Int)(implicit ord: Ordering[T]): Array[T] |  |
| takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T] |  |
| max()(implicit ord: Ordering[T]):T |  |
| min()(implicit ord: Ordering[T]):T |  |
| saveAsTextFile(path: String) |  |
| saveAsTextFile(path: String, codec: Class[\_ <: CompressionCodec]) |  |
| saveAsObjectFile(path: String) |  |
| keyBy[K](f: T => K): RDD[(K, T)] |  |
| checkpoint() |  |
| getCheckpointFile: Option[String] |  |
|  |  |
|  |  |

## Sub RDDs

部分RDD子类的实现分析，包括以下几个部分：

1. 子类本身构造参数
2. 子类的特殊私有变量
3. 子类的Partitioner实现
4. 子类的父类函数实现

def compute(split: Partition, context: TaskContext): Iterator[T]

protected def getPartitions: Array[Partition]

protected def getDependencies: Seq[Dependency[\_]] = deps

protected def getPreferredLocations(split: Partition): Seq[String] = Nil

1. 子类的其他重要方法

### CheckpointRDD

**class** CheckpointRDD[T: ClassTag](sc: SparkContext, **val** checkpointPath: String) **extends** RDD[T](sc, Nil)

CheckpointRDDPartition继承自Partition，没有什么增加。

有一个被广播的hadoop conf变量，在compute方法里使用（readFromFile的时候用）

**val** broadcastedConf = sc.broadcast(**new** SerializableWritable(sc.hadoopConfiguration))

getPartitions: Array[Partition] 方法：

根据checkpointPath去查看Path下有多少个partitionFile，File个数为partition数目。getPartitions方法返回的Array[Partition]内容为New CheckpointRDDPartition(i)，i为[0, 1, …, partitionNum]

getPreferredLocations(split: Partition): Seq[String] 方法：

文件位置信息，借助hadoop core包，获得block location，把得到的结果按照host打散(flatMap)并过滤掉localhost，返回。

compute(split: Partition, context: TaskContext): Iterator[T] 方法：

调用CheckpointRDD.readFromFile(file, broadcastedConf, context)方法，其中file为hadoop file path，conf为广播过的hadoop conf。

**Hadoop文件读写及序列化**

伴生对象提供writeToFile方法和readFromFile方法，主要用于读写hadoop文件，并且利用env下的serializer进行序列化和反序列化工作。两个方法具体实现如下：

**def** writeToFile[T: ClassTag](

path: String,

broadcastedConf: Broadcast[SerializableWritable[Configuration]],

blockSize: Int = -1

)(ctx: TaskContext, iterator: Iterator[T]) {

创建hadoop文件的时候会若存在会抛异常。把hadoop的outputStream放入serializer的stream里，serializeStream.writeAll(iterator)写入。

writeToFile的调用在RDDCheckpointData类的doCheckpoint方法里，如下：

rdd.context.runJob(rdd,

CheckpointRDD.writeToFile(path.toString, broadcastedConf) \_)

**def** readFromFile[T](

path: Path,

broadcastedConf: Broadcast[SerializableWritable[Configuration]],

context: TaskContext

): Iterator[T] = {打开Hadoop的inutStream，读取的时候使用env下的serializer得到反序列化之后的流。返回的时候，DeserializationStream这个trait提供了asIterator方法，每次next操作可以进行一次readObject。

在返回之前，调用了TaskContext提供的addOnCompleteCallback回调，用于关闭hadoop的inputStream。

### HadoopRDD

参考HadoopRDD，已经实现了一个OrcfileRDD

### NewHadoopRDD

**class** NewHadoopRDD[K, V](

sc : SparkContext,

inputFormatClass: Class[\_ <: InputFormat[K, V]],

keyClass: Class[K],

valueClass: Class[V],

@transient conf: Configuration)

**extends** RDD[(K, V)](sc, Nil)

**with** SparkHadoopMapReduceUtil

**private**[spark] **class** NewHadoopPartition(

rddId: Int,

**val** index: Int,

@transient rawSplit: InputSplit **with** Writable)

**extends** Partition {

**val** serializableHadoopSplit = **new** SerializableWritable(rawSplit)

**override** **def** hashCode(): Int = 41 \* (41 + rddId) + index

}

getPartitions 操作:

根据inputFormatClass和conf，通过hadoop InputFormat实现类的getSplits(JobContext)方法得到InputSplits。

这样获得的split同RDD的partition直接对应。

compute 操作：

针对本次split(partition)，调用InputFormat的createRecordReader(split)方法，

得到RecordReader<K,V>。这个RecordReader包装在Iterator[(K,V)]类内，复写Iterator的next()和hasNext方法，让compute返回的InterruptibleIterator[(K,V)]能够被迭代获得RecordReader取到的数据。

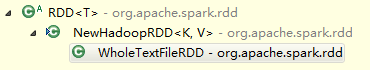
getPreferredLocations(split: Partition)操作：

theSplit.serializableHadoopSplit.value.getLocations.filter(\_ != "localhost")

在NewHadoopPartition里SerializableWritable将split序列化，然后调用InputSplit本身的getLocations接口，得到有数据分布节点的nodes name列表。

### WholeTextFileRDD

NewHadoopRDD的子类



**private**[spark] **class** WholeTextFileRDD(

sc : SparkContext,

inputFormatClass: Class[\_ <: WholeTextFileInputFormat],

keyClass: Class[String],

valueClass: Class[String],

@transient conf: Configuration,

minPartitions: Int)

**extends** NewHadoopRDD[String, String](sc, inputFormatClass, keyClass, valueClass, conf) {

复写了getPartitions方法：

NewHadoopRDD有自己的inputFormat实现类和recordReader实现类。在org.apache.spark.input package下专门写了这两个类的实现。是种参考。

#### InputFormat

WholeTextFileRDD在spark里实现了自己的inputFormat。读取的File以K，V的结构获取，K为path，V为整个file的content。

复写createRecordReader以使用WholeTextFileRecordReader

复写setMaxSplitSize方法，由于用户可以传入minSplits数目，计算平均大小(splits files总大小除以split数目)的时候就变了。

#### WholeTextFileRecordReader

复写nextKeyValue方法，会读出指定path下的file的内容，生成new Text()给value，结果是String。如果文件正在被别的进行打开着，会返回false。否则把file内容读进value里。

#### 使用场景

在SparkContext下提供wholeTextFile方法，

**def** wholeTextFiles(path: String, minPartitions: Int = defaultMinPartitions):

RDD[(String, String)] = {

**val** job = **new** NewHadoopJob(hadoopConfiguration)

NewFileInputFormat.addInputPath(job, **new** Path(path))

**val** updateConf = job.getConfiguration

**new** WholeTextFileRDD(

**this**,

classOf[WholeTextFileInputFormat],

classOf[String],

classOf[String],

updateConf,

minPartitions)

}

用于读取一个路径下的所有text文件，以K，V的形式返回，K为一个文件的path，V为文件内容。比较适合小文件。

# Storage

## BlockManager

BlockManagerMaster类在里面主要充当通信功能，内部主要是Actor，不同的调用对应不同的通信响应。

ShuffleBlockManager主要管理一些shuffle task的东西，之前shuffle模型里有写。

存储介质包括MemoryStore, DiskStore, TachyonStore

### 读取流程

BlockManager.doGetRemote(BlockId, xx)

BlockManagerWorker.syncGetBlock(GetBlock(), ConnectionManagerId(xx, xx))

ConnectionManager.sendMessageReliablySync(id, BlockMessageArray.xx)

## Tachyon支持

# Broadcast机制

类似DistributedCache，参考“Performance and Scalability of Broadcast in Spark”

SparkContext的broadcast方法

**def** broadcast[T: ClassTag](value: T): Broadcast[T] = {

**val** bc = env.broadcastManager.newBroadcast[T](value, isLocal)

cleaner.foreach(\_.registerBroadcastForCleanup(bc))

bc

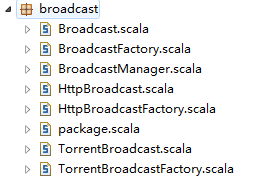
}

广播对象必须是Broadcast[T]类，比如在广播hadoop conf的时候，对象就是

Broadcast[SerializableWritable[Configuration]]

广播后的效果是每个slave上都保存一份，访问的时候使用broadcast.value的方式。

SparkContext借助BroadcastManager来实现广播



## BroadcastManager

BroadcastManager借助factory实现广播，分配一个递增的Id

**def** newBroadcast[T: ClassTag](value\_ : T, isLocal: Boolean) = {

broadcastFactory.newBroadcast[T](value\_, isLocal, nextBroadcastId.getAndIncrement())

}

BroadcastManager的初始化参数：

**private**[spark] **class** BroadcastManager(

**val** isDriver: Boolean,

conf: SparkConf,

securityManager: SecurityManager)

**extends** Logging {

**private** **var** initialized = **false**

**private** **var** broadcastFactory: BroadcastFactory = **null**

// 反射出广播工厂类

initialize()

**val** broadcastFactoryClass =

conf.get("spark.broadcast.factory","org.apache.spark.broadcast.HttpBroadcastFactory")

broadcastFactory = Class.forName(broadcastFactoryClass).newInstance.asInstanceOf[BroadcastFactory]

在初始化的时候，根据SparkConf里的”spark.broadcast.factory”参数获得工厂类，默认是HttpBroadcastFactory。

## Broadcast的两种实现



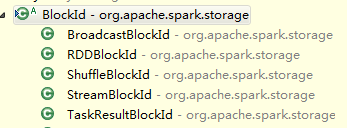
HttpBroadcastFactory的newBroadcast方法，实际上是生成一个HttpBroadcast[T](value, isLocal, id)类。

同样，TorrentBroadcastFactory的newBroadcast方法是生成一个TorrentBroadcast[T](value, isLocal, id)类

HttpBroadcast和TorrentBroadcast是Broadcast的子类，Broadcast提供了persist/unpersist, destroy方法。

### HTTP

广播的数据会存储在driver的BlockManager里，根据id，会生成一个BroadcastBlockId类。实际上，被BlockManager管理的Id，按类型分为以下几类(借此可以看到BlockMananger里储存的内容)



HttpBroadcast的数据会put进BlockManager里。putSingle是BlockManager里一个通用的存储单个Object的方法，最终会使用BlockManager的doPut()方法，具体在BlockManager一节里介绍。

HttpBroadcast.synchronized {

SparkEnv.get.blockManager.putSingle(

blockId, value\_, StorageLevel.MEMORY\_AND\_DISK, tellMaster = **false**)

}

本地的读写：

写的时候是往File里写，会压缩、会序列化。

broadcastDir是conf里的spark.local.dir目录。

读的时候根据id和server信息得到URI，得到流，反序列化出数据。

相应初始化的时候，会初始化这些元素:bufferSize, compress, server

**def** initialize(isDriver: Boolean, conf: SparkConf, securityMgr: SecurityManager) {

synchronized {

**if** (!initialized) {

bufferSize = conf.getInt("spark.buffer.size", 65536)

compress = conf.getBoolean("spark.broadcast.compress", **true**)

securityManager = securityMgr

**if** (isDriver) {

createServer(conf)

conf.set("spark.httpBroadcast.uri", serverUri)

}

serverUri = conf.get("spark.httpBroadcast.uri")

cleaner = **new** MetadataCleaner(MetadataCleanerType.HTTP\_BROADCAST, cleanup, conf)

compressionCodec = CompressionCodec.createCodec(conf)

initialized = **true**

}

}

}

### Torrent

类似BitTorrent的机制，driver把序列化好之后的广播数据切成小的chunks，存在BlockManager里。这些chunks的位置信息报告给BlockManagerMaster，这样所有的executors能知道chunks的位置。

前面的executor照样是通过BlockManager来得到广播数据，这些广播数据chunks合并后，会继续广播，然后BlockManangerMaster就得知了chunks的多处存放位置，之后的executor在获取的时候就可以做到分布式的获取。优点就是让driver分发广播数据的时候不再成为瓶颈。

与Http不同的是，在put进BlockManager时候，分两部分：meta-info和具体blocks。在put的时候，需要通知BlockManagerMaster。Meta-info记录了一次广播的总块数和总大小，每块都单独put进BlockManager。

**def** sendBroadcast() {

**val** tInfo = TorrentBroadcast.blockifyObject(value\_)

totalBlocks = tInfo.totalBlocks

totalBytes = tInfo.totalBytes

hasBlocks = tInfo.totalBlocks

// Store meta-info

**val** metaId = BroadcastBlockId(id, "meta")

**val** metaInfo = TorrentInfo(**null**, totalBlocks, totalBytes)

TorrentBroadcast.synchronized {

SparkEnv.get.blockManager.putSingle(

metaId, metaInfo, StorageLevel.MEMORY\_AND\_DISK, tellMaster = **true**)

}

// Store individual pieces

**for** (i <- 0 until totalBlocks) {

**val** pieceId = BroadcastBlockId(id, "piece" + i)

TorrentBroadcast.synchronized {

SparkEnv.get.blockManager.putSingle(

pieceId, tInfo.arrayOfBlocks(i), StorageLevel.MEMORY\_AND\_DISK, tellMaster = **true**)

}

}

}

这应该算是对广播和共享的一种优化，比较适合大的数据？

# Spark任务模型

一个Spark的Job分为多个stage，最后一个stage会包括一个或多个ResultTask，前面的stages会包括一个或多个ShuffleMapTasks。

ResultTask执行并将结果返回给driver application。

ShuffleMapTask将task的output根据task的partition分离到多个buckets里。一个ShuffleMapTask对应一个ShuffleDependency的partition，而总partition数同并行度、reduce数目是一致的。

## Task

Task的代码在scheduler package下。

抽象类Task构造参数如下：

**private**[spark] **abstract** **class** Task[T](**val** stageId: Int, **var** partitionId: Int) **extends** Serializable {

Task对应一个stageId和partitionId。

提供runTask()接口、kill()接口等。

提供killed变量、TaskMetrics变量、TaskContext变量等。

除了上述基本接口和变量，Task的伴生对象提供了序列化和反序列化应用依赖的jar包的方法。原因是Task需要保证工作节点具备本次Task需要的其他依赖，注册到SparkContext下，所以提供了把依赖转成流写入写出的方法。

## TaskContext

## Task两种实现

### ShuffleMapTask及Shuffle模型

ShuffleMapTask构造参数如下，

**private**[spark] **class** ShuffleMapTask(

stageId: Int,

**var** rdd: RDD[\_],

**var** dep: ShuffleDependency[\_,\_],

\_partitionId: Int,

@transient **private** **var** locs: Seq[TaskLocation])

**extends** Task[MapStatus](stageId, \_partitionId)

RDD partitioner对应的是ShuffleDependency。

ShuffleMapTask复写了MapStatus向外读写的方法，因为向外读写的内容包括：stageId，rdd，dep，partitionId，epoch和split(某个partition)。对于其中的stageId，rdd，dep有统一的序列化和反序列化操作并会cache在内存里，再放到ObjectOutput里写出去。序列化操作使用的是Gzip，序列化信息会维护在serializedInfoCache = **new** HashMap[Int, Array[Byte]]。这部分需要序列化并保存的原因是：stageId，rdd，dep真正代表了本次Shuffle Task的信息，为了减轻master节点负担，把这部分序列化结果cache了起来。

#### runTask逻辑

主要步骤如下：

**val** ser = Serializer.getSerializer(dep.serializer)

shuffle = shuffleBlockManager.forMapTask(dep.shuffleId, partitionId, numOutputSplits, ser)

这一步是初始化一个ShuffleWriterGroup，Group里面是一个BlockObjectWriter数组。

**for** (elem <- rdd.iterator(split, context)) {

**val** pair = elem.asInstanceOf[Product2[Any, Any]]

**val** bucketId = dep.partitioner.getPartition(pair.\_1)

shuffle.writers(bucketId).write(pair)

}

然后每个Writer对应一个bucket，调用每个BlockObjectWriter的write()方法写数据

**var** totalBytes = 0L

**var** totalTime = 0L

**val** compressedSizes: Array[Byte] = shuffle.writers.map { writer: BlockObjectWriter =>

writer.commit()

writer.close()

**val** size = writer.fileSegment().length

totalBytes += size

totalTime += writer.timeWriting()

MapOutputTracker.compressSize(size)

}

这一步是执行writer.commit()，并得到结果file segment大小，对总大小压缩

**val** shuffleMetrics = **new** ShuffleWriteMetrics

shuffleMetrics.shuffleBytesWritten = totalBytes

shuffleMetrics.shuffleWriteTime = totalTime

metrics.get.shuffleWriteMetrics = Some(shuffleMetrics)

success = **true**

**new** MapStatus(blockManager.blockManagerId, compressedSizes)

这一步是记录metrcis信息，最后返回一个MapStatus类，里面是本地ShuffleMapTask结果的相关信息。

最后会release writers，让对应的shuffle文件得到记录和重用(ShuffleBlockManager管理这些file，这些file是Shuffle Task中一组Writer写的对象)。

关于Shuffle Write和Shuffle Fetch模型，下面这篇文章介绍地比较好

<http://jerryshao.me/architecture/2014/01/04/spark-shuffle-detail-investigation/>

#### Shuffle Write

下图展示的是Shuffle Write模型：

目前的Shuffle write实现是多组ShuffleTask共写一组中间文件，而且是边读边往Disk上的文件写的（而不是先在内存写好，一下flush到磁盘上）。Bucket数目等于并行度，等于reducer数目。



#### Shuffle Fetch

如何通过BlockManager从远端拉取?

#### 重要外部类

介绍涉及到的重要外部类，帮助理解。

##### ShuffleBlockManager\*

整体梳理：

ShuffleState维护了两个ShuffleFileGroup的ConcurrentLinkedQueue，以记录目前shuffle的state。

ShuffleState记录了一次shuffle操作的文件组状态，在ShuffleBlockManager内用Map为每个shuffleId维护了一个ShuffleState。

每个shuffleId通过forMapTask()方法得到一组writer，即ShuflleWriterGroup。这组里的writers共享一个shuffleId和mapId，但是每个对应不同的bucketId和file。在为writer分配FileGroup的时候，会从shuffleId对应的shuffle state里先取unusedFileGroup，如果不存在，则在HDFS上新建File。

对于HDFS上的目标file，writer是可以append写的。在新建file的时候，是根据shuffleId和bucket number和一个递增的fileId来创建新的文件的。

ShuffleFileGroup的重用files和记录mapId，index，offset这块似懂非懂。

重要方法：

**def** forMapTask(shuffleId: Int, mapId: Int, numBuckets: Int, serializer: Serializer) = { **new** ShuffleWriterGroup {}}

该方法被一个ShuffleMapTask调用，传入了这次shuffle操作的id，mapId是partitionId。Buckects数目等于分区数目。该方法返回的ShuffleWriterGroup里面是一组DiskBlockObjectWriter，每一个writer都属于这一次shuffle操作，所以他们有共同的shuffleId，mapId，但是他们对应了不同的bucket，并且各自对应一个file。

在shuffle run里的调用和参数传入：

**val** ser = Serializer.getSerializer(dep.serializer)

shuffle = shuffleBlockManager.forMapTask(dep.shuffleId, partitionId, numOutputSplits, ser)

shuffleId是由ShuffleDependency获得的全局唯一id，代表本次shuffle任务id

mapId等于partitionId

Bucket数目等于分区数目

产生writers：

Writer类型是DiskBlockObjectWriter，数目等于buckets数目。bufferSize的设置：

**private** **val** bufferSize = conf.getInt("spark.shuffle.file.buffer.kb", 100) \* 1024

blockId产生自：

**val** blockId = ShuffleBlockId(shuffleId, mapId, bucketId)

在生成writer的时候调用的是BlockManager的getDiskWriter方法，ShuffleBlockManager初始化的时候绑定BlockManager。

**private**[spark] **class** DiskBlockObjectWriter(

blockId: BlockId,

file: File,

serializer: Serializer,

bufferSize: Int,

compressStream: OutputStream => OutputStream,

syncWrites: Boolean)

**extends** BlockObjectWriter(blockId)

ShuffleFileGroup：私有内部类，对应了一组shuffle files，每个file对应一个reducer。一个Mapper会分到一个ShuffleFileGroup，把mapper的结果写到这组File里去。

##### MapStatus

注意到ShuffleMapTask的类型是MapStatus类。MapStatus类是ShuffleMapTask要返回给scheduler的执行结果，包括两个东西：

**private**[spark] **class** MapStatus(**var** location: BlockManagerId, **var** compressedSizes: Array[Byte])

前者是run这次task的block manager地址（BlockManagerId是一个类，保存了executorId, host, port, nettyPort），后者是output大小，该值会传给接下来的reduce任务。该size是被MapOutputTracker压缩过的。

MapStatus类提供了两个方法如下，ShuffleMapTask进行了复写。

**def** writeExternal(out: ObjectOutput) {

location.writeExternal(out)

out.writeInt(compressedSizes.length)

out.write(compressedSizes)

}

**def** readExternal(in: ObjectInput) {

location = BlockManagerId(in)

compressedSizes = **new** Array[Byte](in.readInt())

in.readFully(compressedSizes)

}

##### BlockManagerId

BlockManagerId类构造依赖executorId, host, port, nettyPort这些信息。伴生对象维护了一个blockManagerIdCache ，实现为ConcurrentHashMap[BlockManagerId, BlockManagerId]() 。

比如MapStatus的readExternal方法把ObjectInput传入BlockManagerId构造函数的时候，BlockManagerId的apply()方法就会根据ObjectInput取出executorId, host, port, nettyPort信息，把这个BlockManagerId obj维护到blockManagerIdCache内

### ResultTask

构造参数

**private**[spark] **class** ResultTask[T, U](

stageId: Int,

**var** rdd: RDD[T],

**var** func: (TaskContext, Iterator[T]) => U,

\_partitionId: Int,

@transient locs: Seq[TaskLocation],

**var** outputId: Int)

**extends** Task[U](stageId, \_partitionId) **with** Externalizable {

ResultTask比较简单，runTask方法调用的是rdd的迭代器：

**override** **def** runTask(context: TaskContext): U = {

metrics = Some(context.taskMetrics)

**try** {

func(context, rdd.iterator(split, context))

} **finally** {

context.executeOnCompleteCallbacks()

}

}

## 进程模型 vs. 线程模型

Spark同节点上的任务以多线程的方式运行在一个JVM进程中。

优点：

启动任务快

共享内存，适合内存密集型任务

Executor所占资源可重复利用

缺点：

同节点上的所有任务运行在一个进程中，会出现严重的资源争用，难以细粒度控制每个任务的占用资源。MapReduce为Map Task和Reduce Task设置不同资源，细粒度控制任务占用资源量。

MapReduce的每个Task都是一个JVM进程，都要经历：资源申请->运行任务->释放资源的过程

每个节点可以有一个或多个Executor，Executor配有一定数量slots，Executor内可以跑多个Result Task和ShuffleMap Task。

在共享内存方面，broadcast的变量会在每个executor里存一份，这个executor内的任务可以共享。

# Spark任务执行模型

## DAGScheduler

面向stage的调度层，为job生成以stage组成的DAG，提交TaskSet给TaskScheduler执行。

每一个Stage内，都是独立的tasks，他们共同执行同一个compute function，享有相同的shuffle dependencies。DAG在切分stage的时候是依照出现shuffle为界限的。

DAGScheduler负责逻辑上切分job的stages，并且提交Tasks给TaskScheduler。对于local可以执行的任务，会进行local的run task。

**class** DAGScheduler(

**private**[scheduler] **val** sc: SparkContext,

**private**[scheduler] **val** taskScheduler: TaskScheduler,

listenerBus: LiveListenerBus,

mapOutputTracker: MapOutputTrackerMaster,

blockManagerMaster: BlockManagerMaster,

env: SparkEnv)

**extends** Logging {

内部维护的内容:

**private**[scheduler] **val** nextJobId = **new** AtomicInteger(0)

**private**[scheduler] **def** numTotalJobs: Int = nextJobId.get()

**private** **val** nextStageId = **new** AtomicInteger(0)

**private**[scheduler] **val** jobIdToStageIds = **new** HashMap[Int, HashSet[Int]]

**private**[scheduler] **val** stageIdToJobIds = **new** HashMap[Int, HashSet[Int]]

**private**[scheduler] **val** stageIdToStage = **new** HashMap[Int, Stage]

**private**[scheduler] **val** shuffleToMapStage = **new** HashMap[Int, Stage]

**private**[scheduler] **val** jobIdToActiveJob = **new** HashMap[Int, ActiveJob]

**private**[scheduler] **val** resultStageToJob = **new** HashMap[Stage, ActiveJob]

**private**[scheduler] **val** stageToInfos = **new** HashMap[Stage, StageInfo]

// Stages we need to run whose parents aren't done

**private**[scheduler] **val** waitingStages = **new** HashSet[Stage]

// Stages we are running right now

**private**[scheduler] **val** runningStages = **new** HashSet[Stage]

// Stages that must be resubmitted due to fetch failures

**private**[scheduler] **val** failedStages = **new** HashSet[Stage]

// Missing tasks from each stage

**private**[scheduler] **val** pendingTasks = **new** HashMap[Stage, HashSet[Task[\_]]]

**private**[scheduler] **val** activeJobs = **new** HashSet[ActiveJob]

// Contains the locations that each RDD's partitions are cached on

**private** **val** cacheLocs = **new** HashMap[Int, Array[Seq[TaskLocation]]]

在start()方法中会初始化Actor，然后接收DAGSchedulerEvent处理。Scheduler会在SparkContext里start起来。

有一些提供给TaskScheduler调用的方法

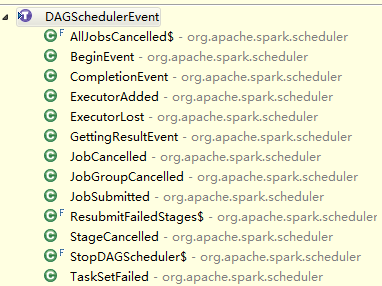
### Event生命周期

Event生命周期图：

### Event处理

源码的阅读入口：可以根据processEvent(event: DAGSchedulerEvent)方法展开。

处理的事件包括这么一些：



#### JobSubmitted

JobSubmitted事件：



提交任务的事件传入参数如下

JobSubmitted(jobId, rdd, func2, partitions.toArray, allowLocal, callSite, waiter, properties)

处理过程可以拆成三步看，每一步里面涉及的具体逻辑在下面补充展开

finalStage = newStage(finalRDD, partitions.size, None, jobId, Some(callSite))

本次newStage()操作可以对应新的result stage或者shuffle stage。返回Stage类(里面记录一些信息)。Stage类会传入Option[ShuffleDependency[\_,\_]]参数，内部有一个isShuffleMap变量，以标识该Stage是shuffle or result。

**val** job = **new** ActiveJob(jobId, finalStage, func, partitions, callSite, listener, properties)

ActiveJob类也是记录一些信息的类，可以当作是一个VO类

**if** (allowLocal && finalStage.parents.size == 0 && partitions.length == 1) {

// Compute very short actions like first() or take() with no parent stages locally.

listenerBus.post(SparkListenerJobStart(job.jobId, Array[Int](), properties))

runLocally(job)

} **else** {

jobIdToActiveJob(jobId) = job

activeJobs += job

resultStageToJob(finalStage) = job

listenerBus.post(SparkListenerJobStart(job.jobId, jobIdToStageIds(jobId).toArray, properties))

submitStage(finalStage)

}

首先判断stage没有父亲依赖，且partition为1的话，就执行本地任务。否则，submitStage。

submitStage的逻辑为，首先寻找本次stage的parents。如果没有missing的parent stage，那么就submitMissingTask,即提交本次stage的tasks了。如果有，会对parent stage进行递归submitStage，而且getMissingParentStages得到的结果集是按id降序排的，也就是说递归submitStage的时候会按parent stage的id顺序进行。

submitMissingTask处理的是stage的parent已经available的stage。主要逻辑如下：

第一步：通过stage.isShuffleMap来决定生成ShuffleMapTask还是ResultTask，生成的ShuffleMapTask数目和partition数目相等。

第二步：把生成的tasks组建成一个TaskSet，提交给TaskScheduler的submitTasks方法。

## TaskSchedulerImpl

DAGScheduler以stage为单位，提tasks给TaskScheduer，实现类为TaskSchedulerImpl。

TaskSchedulerImpl几个内部部件：

SchedulerBackend

DAGScheduler

MapOutputTracker (没有看到在里面用？)

SchedulableBuilder和一个rootPool

TaskSetManager

TaskResultGetter

Tasks信息（taskIdToTaskSetId，taskIdToExecutorId，activeExecutorIds）

别的信息（SchedulerMode）

TaskScheduler做接收task、接收分到的资源和executor、维护信息、与backend打交道、把任务分配好等事情。

start()，stop()的时候，backend的start()，stop()

submitTasks(TaskSet)逻辑：

为这批Task生成新的TaskSetManager，把TaskSetManager加到SchedulerBuilder里，然后向backend进行一次reviveOffer()操作。

### 维护Task集模型：SchedulableBuilder

SchedulableBuilder有FIFO和Fair两种实现， addTaskSetManager会把TaskSetManager加到pool里。FIFO的话只有一个pool。Fair有多个pool，Pool也分FIFO和Fair两种模式。

SchedulableBuilder的rootPool里面可以新增pool或者TaskSetManager，两者都是Scheduable的继承类，所以SchedulableBuilder用于维护rootPool这棵Scheduable 树结构。Pool是树上的非叶子节点，而TaskSetManager就是叶子节点。

在TaskScheduler初始化的时候会buildDafaultPool。

### Task集维护类：TaskSetManager

TaskSetManager负责这批Tasks的启动，失败重试，感知本地化等事情。每次reourseOffer方法会寻找合适(符合条件execId, host, locality)的Task并启动它。

reourseOffer方法，

**def** resourceOffer(

execId: String,

host: String,

maxLocality: TaskLocality.TaskLocality)

: Option[TaskDescription]

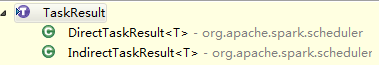
寻找符合execId, host和locality的task，找到的话就启动这个Task。启动的时候，把task加到runningTask的HashSet里，然后调DAGScheduler的taskStarted方法，taskStarted方法向eventProcessorActor发出BeginEvent的DAGSchedulerEvent。

### 获取执行结果：TaskResultGetter

维护一个线程池，用来反序列化和从远端获取task结果。

**def** enqueueSuccessfulTask(taskSetManager: TaskSetManager, tid: Long, serializedData: ByteBuffer) {

把序列化的数据反序列化解析出来之后，有两种情况：直接可读的result和间接task result。



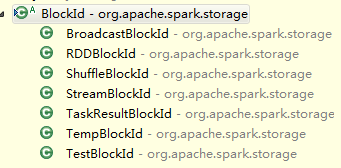
前者是DirectTaskResult[T]类：

**class** DirectTaskResult[T](**var** valueBytes: ByteBuffer, **var** accumUpdates: Map[Long, Any], **var** metrics: TaskMetrics)

后者是IndirectTaskResult[T]类：

**case** **class** IndirectTaskResult[T](blockId: BlockId) **extends** TaskResult[T] **with** Serializable

在反解析出IndirectTaskResult后，可以得到BlockId这个类，他的实现有这么些：



在TaskResultGetter里，会通过blockManager的getRemoteBytes(BlockId)方法来获得序列化的task result，对这个task result进行反解析后得到DirectTaskResult类，从而获得反序列化后的真正结果数据。

这是大致的一个过程，具体还有一些细节在之中，比如会向scheduler发送不同的event、blockManager会调用BlockManagerMaster把该Block remove掉。

BlockId类有这么些关键变量：

**sealed** **abstract** **class** BlockId {

/\*\* A globally unique identifier for this Block. Can be used for ser/de. \*/

**def** name: String

// convenience methods

**def** asRDDId = **if** (isRDD) Some(asInstanceOf[RDDBlockId]) **else** None

**def** isRDD = isInstanceOf[RDDBlockId]

**def** isShuffle = isInstanceOf[ShuffleBlockId]

**def** isBroadcast = isInstanceOf[BroadcastBlockId]

下面看BlockManager如何通过BlockId获得数据：

调用的是BlockManager的内部方法

**private** **def** doGetRemote(blockId: BlockId, asValues: Boolean): Option[Any] = {

require(blockId != **null**, "BlockId is null")

**val** locations = Random.shuffle(master.getLocations(blockId))

**for** (loc <- locations) {

logDebug("Getting remote block " + blockId + " from " + loc)

**val** data = BlockManagerWorker.syncGetBlock(

GetBlock(blockId), ConnectionManagerId(loc.host, loc.port))

**if** (data != **null**) {

**if** (asValues) {

**return** Some(dataDeserialize(blockId, data))

} **else** {

**return** Some(data)

}

}

logDebug("The value of block " + blockId + " is null")

}

logDebug("Block " + blockId + " not found")

None

}

思路是通过BlockManagerMaster来获得block的位置信息，得到的集合打乱后，遍历位置信息，通过BlockManagerWorker去获得数据，只要得到了，就反序列化之后返回。

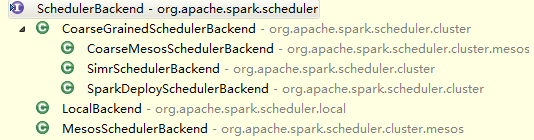
在TaskResultGetter处理的时候，成功和失败分别向Scheduler调用handleSuccessfulTask和handleFailedTask方法。

handleSuccessfulTask在DAGScheduler里，会发出CompletionEvent事件，这一步结尾工作会有很多细节处理，这里先不阅读了。

handleFailedTask的话，只要TaskSetManager不是zombie，task没有被kill，那么会继续调用backend.reviveOffers()来re-run。

# SchedulerBackend

除了YARN外（YARN单独在子项目里），其他backend的继承体系如下：



SchedulerBackend提供的接口模型类似Mesos，需要子类实现这几个方法

start()

stop()

reviveOffers()

defaultParallelism()

killTask()

## Local模式

不分析

## 粗粒度模式

粗粒度模式包括Standalone模式和Mesos。

CoarseGrainedSchedulerBackend(TaskSchedulerImpl, ActorSystem)

对SchedulerBackend里方法的实现，都是围绕Actor展开的，Actor负责任务通信。

DriverActor做的事情。

### Standalone Deploy

SparkDeploySchedulerBackend基本都是复用 CoarseGrainedSchedulerBackend的逻辑，只是还实现了AppClientListener的接口。

所以资源管理这块应该和CoarseGrainedSchedulerBackend一致。

### Mesos粗粒度

## Mesos细粒度模式

## YARN

# 其他类

## SerializableWritable

专门用于提供Writable子类序列化能力的类。一般适用于InputSplit，JobConf类。

提供read, writer方法。