SparkContext初始化过程

SparkContext初始化过程需要完成以下几个步骤，创建spark执行环境、创建RDD清理器、创建并初始化UI、获得Hadoop相关的配置以及Executor执行环境、创建任务和调度器TaskScheduler、创建和启动DAGScheduler、启动TaskScheduler、初始化BlockManager、启动测量系统MetricsSystem、创建和启动Executor管理利器、启动ContextClear、更新Spark环境、创建DAGSchedulerSouce和BlockManagerSource、SparkContext对象被激活。

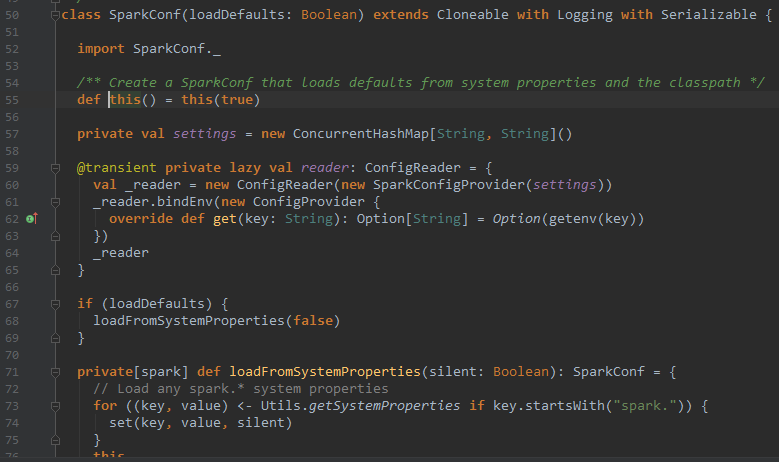
# 总览

## 1.1 SparkConf获得相关的配置信息

初始化过程：



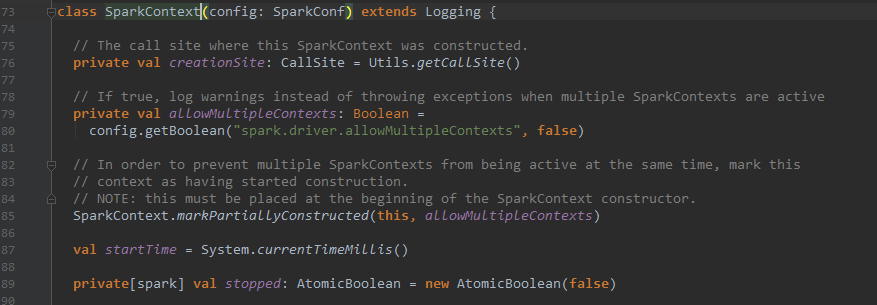
首先定义了SparkConf对象，该对象包含了Spark执行环境所设置的参数。其中：



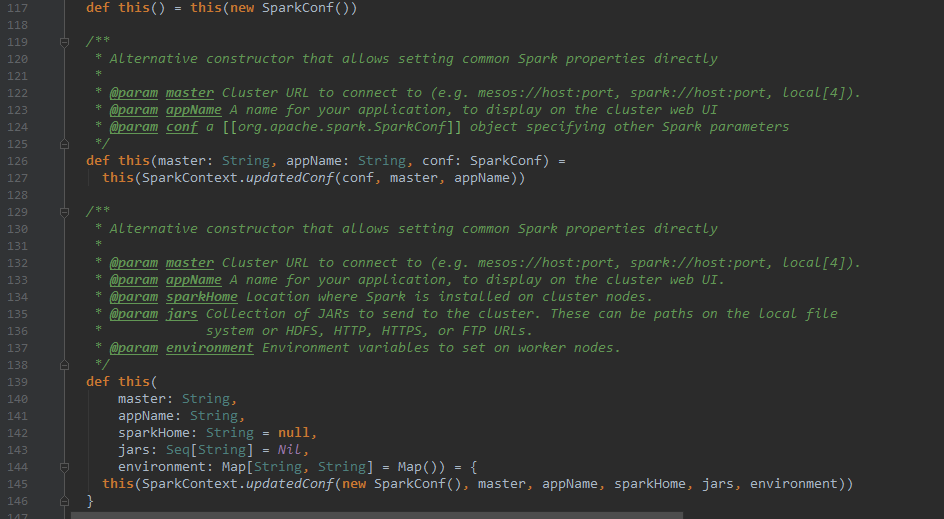
该对象读取系统设置的参数，如果在程序执行时则更新获得的参数配置项，并更新其参数。

## SparkContext

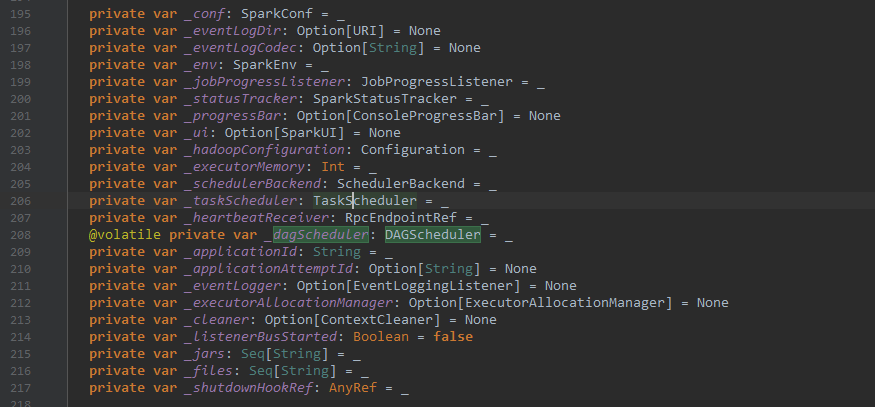
继承了类



几个构造函数：



# SparkContext初始化的变量



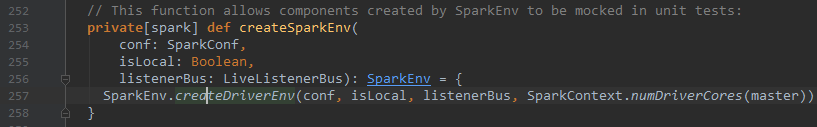
定义了conf获得参数、日志资源等相关、以及系统环境和相关的进度设置

# SparkContext初始化过程

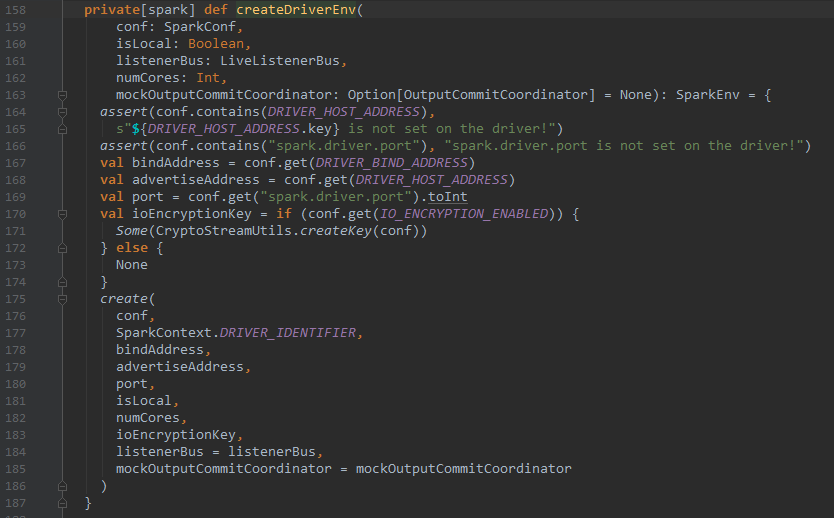
1. 创建Spark的执行环境SparkEnv；
2. 创建RDD清理器metadataCleaner；
3. 创建并初始化UI；
4. 获得Hadoop相关的配置以及Executor执行环境；
5. 创建任务和调度器TaskScheduler；
6. 创建和启动DAGScheduler；
7. 启动TaskScheduler；
8. 初始化BlockManager；
9. 启动测量系统MetricsSystem；
10. 创建和启动Executor管理利器；
11. 启动ContextClear；
12. 更新Spark环境；
13. 创建DAGSchedulerSouce；
14. BlockManagerSource；
15. SparkContext对象被激活。

# 创建SparkEnv

SparkEnv是Spark的执行环境对象，其中包括众多与Executor相关的对象。在Local模式下，Driver创建Executor、local-cluster部署模式或者standlone部署模式下，Woker另起的CoarseGrainedExecutorBackend进程中会创建Executor，故SparkEnv存在于Driver或CoarseGrainedExecutorBackend进程中。

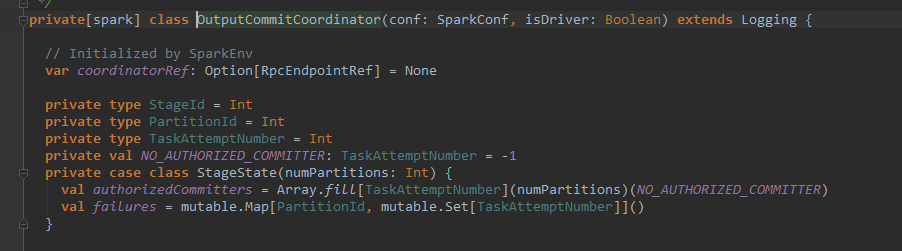


创建SparkEnv时，得到conf参数设置，然后调用createDeriverEnv，创建SparkContext对象。



在调用create函数之前，需要获得相关的信息，例如conf、sparkContextID、主机地址等。最后还监听进度关系，并输出相关的信息。

MockOutPutCommitCoordinator等包含了Stage的输出信息



创建SparkEnv的几个过程：

1. 创建安全管理器SecurityManager；
2. 创建基于Akka的分布式消息系统ActorSystem；
3. 创建Map任务输出跟踪器mapOutTracker；
4. 实例化ShuffleManager；
5. 创建ShuffleMemoryManager；
6. 创建块传输Block Transfer Services；
7. 创建BlockManagerMaster；
8. 创建块管理器BlockManager；
9. 创建广播管理器BroadcastManager；
10. 创建HTTP文件服务器httpFileServer；
11. 创建测量系统MetriesSystem；
12. 创建SparkEnv。

## 4.1 create函数

/\*\*

\* Helper method to create a SparkEnv for a driver or an executor.

\*/

**private def create(**

**conf: SparkConf,**

**executorId: String,**

**bindAddress: String,**

**advertiseAddress: String,**

**port: Int,**

**isLocal: Boolean,**

**numUsableCores: Int,**

**ioEncryptionKey: Option[Array[Byte]],**

**listenerBus: LiveListenerBus = null,**

**mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None): SparkEnv = {**

**# 是否是driver**

**val isDriver = executorId == SparkContext.DRIVER\_IDENTIFIER**

**// Listener bus is only used on the driver**

**if (isDriver) {**

**assert(listenerBus != null, "Attempted to create driver SparkEnv with null listener bus!")**

**}**

**# 注册安全管理器**

**val securityManager = new SecurityManager(conf, ioEncryptionKey)**

**ioEncryptionKey.foreach { \_ =>**

**if (!securityManager.isEncryptionEnabled()) {**

**logWarning("I/O encryption enabled without RPC encryption: keys will be visible on the " +**

**"wire.")**

**}**

**}**

**# 获取系统名称**

**val systemName = if (isDriver) driverSystemName else executorSystemName**

**# 注册RPC服务**

**val rpcEnv = RpcEnv.create(systemName, bindAddress, advertiseAddress, port, conf,**

**securityManager, clientMode = !isDriver)**

**// Figure out which port RpcEnv actually bound to in case the original port is 0 or occupied.**

**// In the non-driver case, the RPC env's address may be null since it may not be listening**

**// for incoming connections.**

**# 在分布式环境中，检测是否是驱动程序**

**if (isDriver) {**

**conf.set("spark.driver.port", rpcEnv.address.port.toString)**

**} else if (rpcEnv.address != null) { # 非driver，则监听端口服务**

**conf.set("spark.executor.port", rpcEnv.address.port.toString)**

**logInfo(s"Setting spark.executor.port to: ${rpcEnv.address.port.toString}")**

**}**

// Create an instance of the class with the given name, possibly initializing it with our conf

# 获得实例对象

def **instantiateClass**[T](className: String): T = {

val cls = Utils.classForName(className) # 使用Utils保存相关内容

// L**ook for a constructor taking a SparkConf and a boolean isDriver, then one taking just**

// SparkConf, then one taking no arguments

try {

cls.getConstructor(classOf[SparkConf], java.lang.Boolean.TYPE)

.newInstance(conf, new java.lang.Boolean(isDriver)) # 是否需要创建

.**asInstanceOf**[T]

} catch {

case \_: NoSuchMethodException =>

try {

cls.getConstructor(classOf[SparkConf]).newInstance(conf).asInstanceOf[T]

} catch {

case \_: NoSuchMethodException =>

cls.getConstructor().newInstance().asInstanceOf[T]

}

}

}

// Create an instance of the class named by the given SparkConf property, or defaultClassName

// if the property is not set, possibly initializing it with our conf

def instantiateClassFromConf[T](propertyName: String, defaultClassName: String): T = {

instantiateClass[T](conf.get(propertyName, defaultClassName))

}

val serializer = instantiateClassFromConf[Serializer](

"spark.serializer", "org.apache.spark.serializer.JavaSerializer")

logDebug(s"Using serializer: ${serializer.getClass}")

# 创建序列化对象，序列号管理器

val serializerManager = new **SerializerManager**(serializer, conf, ioEncryptionKey)

val closureSerializer = new JavaSerializer(conf)

# RPC服务

def registerOrLookupEndpoint(

name: String, endpointCreator: => RpcEndpoint):

RpcEndpointRef = {

if (isDriver) {

logInfo("Registering " + name)

rpcEnv.setupEndpoint(name, endpointCreator)

} else {

RpcUtils.makeDriverRef(name, conf, rpcEnv)

}

}

# 广播对象。

val broadcastManager = new **BroadcastManager**(isDriver, conf, securityManager)

# tracker追踪器

val mapOutputTracker = if (isDriver) {

new MapOutputTrackerMaster(conf, broadcastManager, isLocal)

} else {

new MapOutputTrackerWorker(conf)

}

// Have to assign trackerEndpoint after initialization as MapOutputTrackerEndpoint

// requires the MapOutputTracker itself

**mapOutputTracker**.trackerEndpoint = registerOrLookupEndpoint(MapOutputTracker.ENDPOINT\_NAME,

new MapOutputTrackerMasterEndpoint(

rpcEnv, mapOutputTracker.asInstanceOf[MapOutputTrackerMaster], conf))

// Let the user specify short names for shuffle managers

# 设置Shuffle管理器

val **shortShuffleMgrNames** = Map(

"sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].getName,

"tungsten-sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].getName)

val **shuffleMgrName** = **conf.get("spark.shuffle.manager", "sort")**

val shuffleMgrClass =

shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase(Locale.ROOT), shuffleMgrName)

val **shuffleManager** = instantiateClass[**ShuffleManager**](shuffleMgrClass)

# 一些优化参数

val **useLegacyMemoryManager** = conf.getBoolean("**spark.memory.useLegacyMode", false)**

val memoryManager: MemoryManager =

if (useLegacyMemoryManager) {

new StaticMemoryManager(conf, numUsableCores)

} else {

UnifiedMemoryManager(conf, numUsableCores)

}

val blockManagerPort = if (isDriver) {

conf.get(DRIVER\_BLOCK\_MANAGER\_PORT)

} else {

conf.get(BLOCK\_MANAGER\_PORT)

}

# 设置块信息，通过netty服务管理

val blockTransferService =

new NettyBlockTransferService(conf, securityManager, bindAddress, advertiseAddress,

blockManagerPort, numUsableCores)

# 设置**BlockManagerMaster**

val blockManagerMaster = new **BlockManagerMaster**(registerOrLookupEndpoint(

BlockManagerMaster.DRIVER\_ENDPOINT\_NAME,

new BlockManagerMasterEndpoint(rpcEnv, isLocal, conf, listenerBus)),

conf, isDriver)

// NB: blockManager is not valid until initialize() is called later.

# **设置BlockManager**

val blockManager = new **BlockManager**(executorId, rpcEnv, blockManagerMaster,

serializerManager, conf, memoryManager, mapOutputTracker, shuffleManager,

blockTransferService, securityManager, numUsableCores)

# 创建测量系统

val **metricsSystem** = if (isDriver) {

// Don't start metrics system right now for Driver.

// We need to wait for the task scheduler to give us an app ID.

// Then we can start the metrics system.

MetricsSystem.createMetricsSystem("driver", conf, securityManager)

} else {

// We need to set the executor ID before the MetricsSystem is created because sources and

// sinks specified in the metrics configuration file will want to incorporate this executor's

// ID into the metrics they report.

conf.set("spark.executor.id", executorId)

val ms = **MetricsSystem**.**createMetricsSystem**("executor", conf, securityManager)

ms.start()

ms

}

# 输出追踪器

val **outputCommitCoordinator** = mockOutputCommitCoordinator.getOrElse {

new OutputCommitCoordinator(conf, isDriver)

}

val outputCommitCoordinatorRef = registerOrLookupEndpoint("OutputCommitCoordinator",

new OutputCommitCoordinatorEndpoint(rpcEnv, outputCommitCoordinator))

outputCommitCoordinator.coordinatorRef = Some(outputCommitCoordinatorRef)

# 调用并创建SparkEnv对象

val envInstance = new SparkEnv(

executorId,

rpcEnv,

serializer,

closureSerializer,

serializerManager,

mapOutputTracker,

shuffleManager,

broadcastManager,

blockManager,

securityManager,

metricsSystem,

memoryManager,

outputCommitCoordinator,

conf)

// Add a reference to tmp dir created by driver, we will delete this tmp dir when stop() is

// called, and we only need to do it for driver. Because driver may run as a service, and if we

// don't delete this tmp dir when sc is stopped, then will create too many tmp dirs.

if (isDriver) {

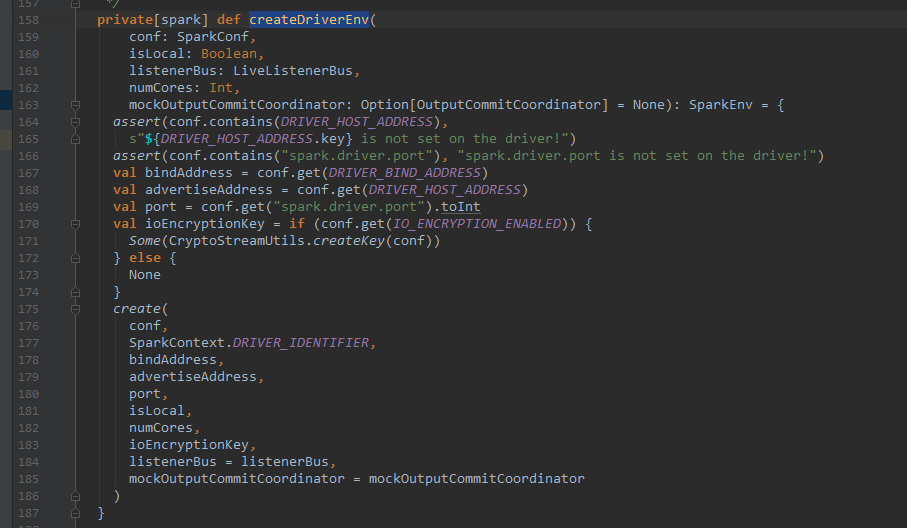
val sparkFilesDir = Utils.createTempDir(Utils.getLocalDir(conf), "userFiles").getAbsolutePath

envInstance.driverTmpDir = Some(sparkFilesDir)

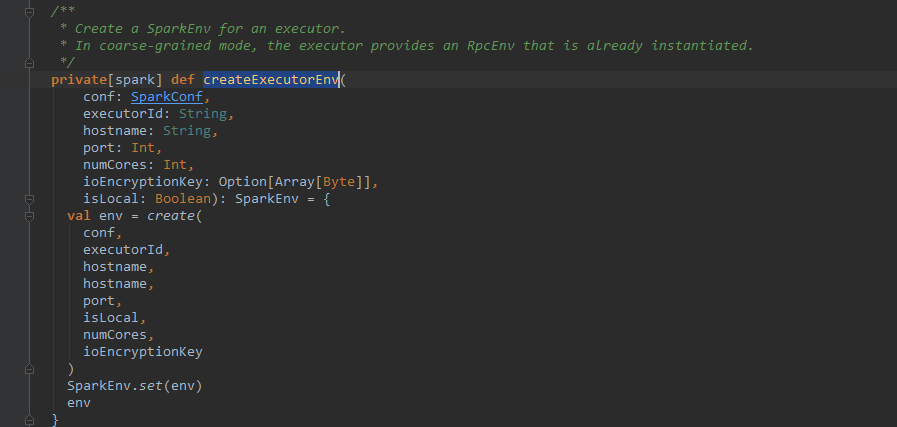
}

envInstance

## 4.2 创建createDriverEnv



## 4.3 创建createExecutorEnv



## 4.4 CoarseGrainedExecutorBackend

**private**[spark] **object** CoarseGrainedExecutorBackend **extends** Logging {

**private def** run(

driverUrl: String,

executorId: String,

hostname: String,

cores: Int,

appId: String,

workerUrl: Option[String],

userClassPath: Seq[URL]) {

Utils.*initDaemon*(log)

SparkHadoopUtil.*get*.runAsSparkUser { () =>

// Debug code

Utils.*checkHost*(hostname)

// Bootstrap to fetch the driver's Spark properties.

**val** executorConf = **new** SparkConf

**val** port = executorConf.getInt("spark.executor.port", 0)

**val** fetcher = RpcEnv.*create*(

"driverPropsFetcher",

hostname,

port,

executorConf,

**new** SecurityManager(executorConf),

clientMode = **true**)

**val** driver = fetcher.setupEndpointRefByURI(driverUrl)

**val** cfg = driver.askSync[SparkAppConfig](RetrieveSparkAppConfig)

**val** props = cfg.sparkProperties ++ *Seq*[(String, String)](("spark.app.id", appId))

fetcher.shutdown()

// Create SparkEnv using properties we fetched from the driver.

**val** driverConf = **new** SparkConf()

**for** ((key, value) <- props) {

// this is required for SSL in standalone mode

**if** (SparkConf.*isExecutorStartupConf*(key)) {

driverConf.setIfMissing(key, value)

} **else** {

driverConf.set(key, value)

}

}

**if** (driverConf.contains("spark.yarn.credentials.file")) {

logInfo("Will periodically update credentials from: " +

driverConf.get("spark.yarn.credentials.file"))

SparkHadoopUtil.*get*.startCredentialUpdater(driverConf)

}

**val** env = SparkEnv.*createExecutorEnv*(

driverConf, executorId, hostname, port, cores, cfg.ioEncryptionKey, isLocal = **false**)

env.rpcEnv.setupEndpoint("Executor", **new** CoarseGrainedExecutorBackend(

env.rpcEnv, driverUrl, executorId, hostname, cores, userClassPath, env))

workerUrl.foreach { url =>

env.rpcEnv.setupEndpoint("WorkerWatcher", **new** WorkerWatcher(env.rpcEnv, url))

}

env.rpcEnv.awaitTermination()

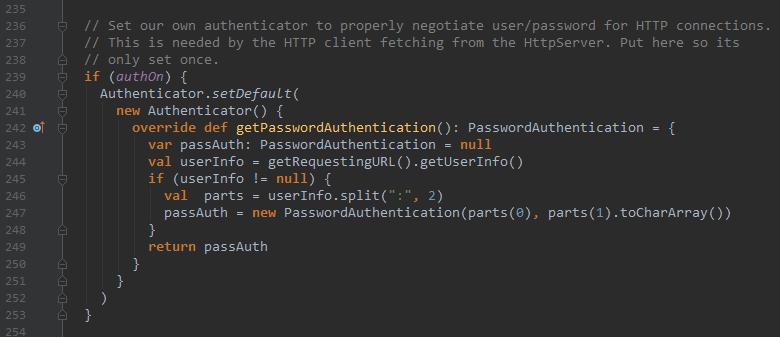
SparkHadoopUtil.*get*.stopCredentialUpdater()

}

}

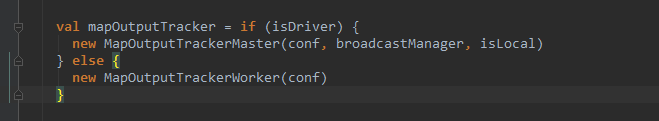
# 安全管理器SecurityManager

SecurityManager主要对权限、账号进行设置，如果基于Hadoop 的Yarn资源管理器，需要证书secret key登录，最后给出默认的口令认证。如果使用认证，那么将执行认证操作。其中认证通信方式为ssl方法。



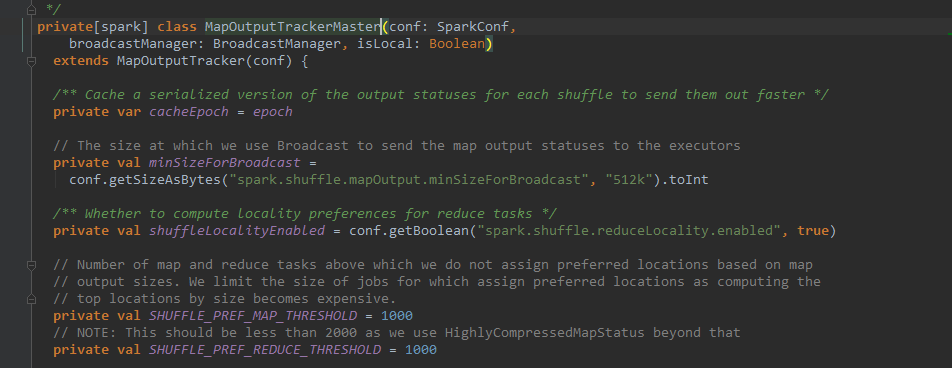
# Map任务输出跟踪器mapOutputTracker

该阶段主要的工作是跟踪map阶段任务状态的输出，便于reduce阶段获取任务的地址和中间输出结果。每个map任务或者reduce任务都会有其唯一的标识，分别为mapID和reduceID。每个reduce任务的输入可能是多个map的输出，reduce会到各个map任务节点上拉取block，该过程称为shuffle。每个shuffle过程都有唯一的标识shuffleID。mapOutputTracker有MapOutputTrackerMaster和MapOutputTrackerWorker

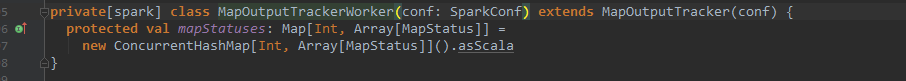


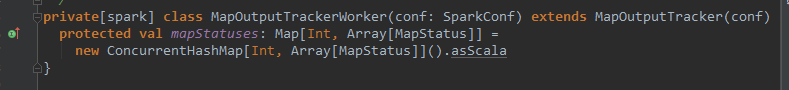
## 6.1 MapOutputTrackerMaster

在主节点master上的工作。



## mapOutputTrackerWorker

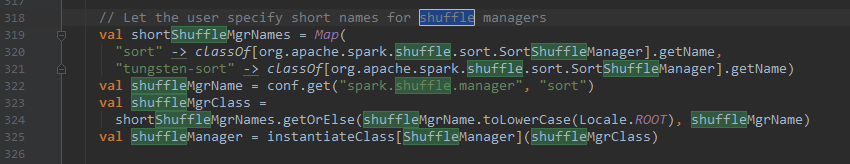


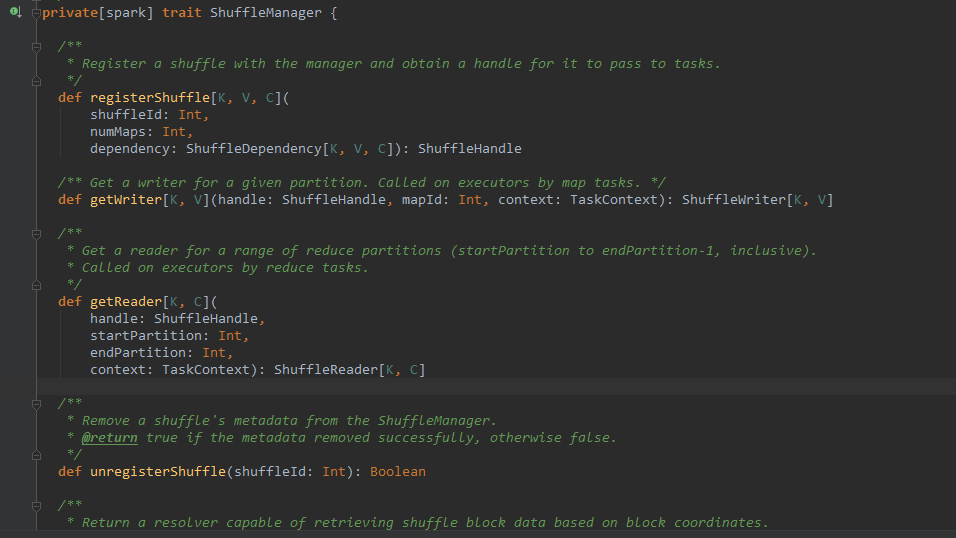


# 实例化ShuffleManager

ShuffleManager负责管理本地及远程的block数据的shuffle操作。ShuffleManager默认通过反射方式生成SortShuffleManager实例。通过修改属性spark.shuffle.manager 为hash的方法控制使用HashShuffleManager。HashShuffleManager通过IndexShuffleBlockManager间接操作BlockManager中的DiskBlockManager将map结果写入本地，并根据ShuffleID、mapID写入索引文件，也通过MapOutputTrackerMaster中维护的mapStatue从本地或者其他远程节点读取数据。

由于reduce在多个节点上，因此通过shuffle将所有的reduce的输入汇总起来，需要shuffle过程。





# BroadcastManager广播管理器

广播管理器需要得到配置信息和安全注册信息。，用于将配置信息和序列化后的RDD、job、已经ShuffleDependecy等信息进行存储。代码如下。



broadcastManager必须在初始化方法initialize被调用后，才能生效。Initialize方法利用反射生成广播工厂实例broadcastFactory

private[spark] class BroadcastManager(

val isDriver: Boolean,

conf: SparkConf,

securityManager: SecurityManager)

extends Logging {

private var initialized = false

private var broadcastFactory: BroadcastFactory = null

initialize()

// Called by SparkContext or Executor before using Broadcast

private def initialize() {

synchronized {

if (!initialized) {

broadcastFactory = new TorrentBroadcastFactory

broadcastFactory.initialize(isDriver, conf, securityManager)

initialized = true

}

}

}

def stop() {

broadcastFactory.stop()

}

private val nextBroadcastId = new AtomicLong(0)

# 代理工厂方法**broadcastFactory的newBroadcast方法生成广播对象。同理的**unbroadcast类似。

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

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

}

def unbroadcast(id: Long, removeFromDriver: Boolean, blocking: Boolean) {

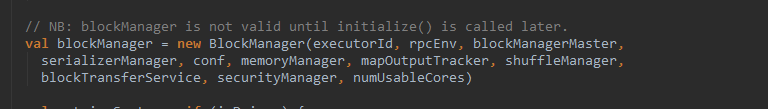
broadcastFactory.unbroadcast(id, removeFromDriver, blocking)

}

}

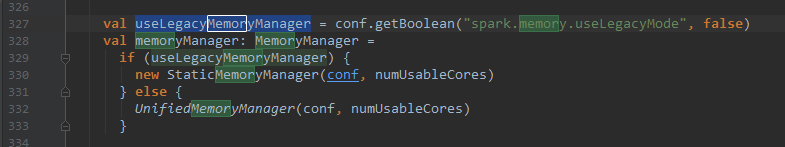
# 块管理器BlockManager

BlockManger负责对Block管理，只有在BlockManager的初始化方法initlize被调用后，才有效。BlockManager作为存储系统的一部分。



# 内存管理

两种管理方法StaticMemoryManager和*UnifiedMemoryManager*，通过spark.memory.useLegacyMode设置内存管理方法，其中跟core是具有相关性，



## 10.1 StaticMemoryManager管理方法

*部分内容：https://blog.csdn.net/dabokele/article/details/51475469*

StaticMemoryManager各部分内存静态划分好后便不可变化。根据**`*spark.shuffle.memoryFraction*`** *和*`***spark.storage.memoryFraction*`**设置相关的选项。

1. **变量**

maxUnrollMemory由maxStorageMemory(该方法在MemoryManager中被定义)乘以spark.storage.unrollFraction（默认值0.2）来确定。

也就是说在storage内存中，有一部分会被用于unroll。由于Spark允许序列化和非序列化两种方式存储数据，对于序列化的数据，必须要先展开后才能使用。unroll部分空间就是用于展开序列化数据的。这部分空间是动态分配的

private val maxUnrollMemory: Long = {

(maxStorageMemory \* conf.getDouble("spark.storage.unrollFraction", 0.2)).toLong

}

**2. 方法**

**（1）getMaxStorageMemory(conf: SparkConf)**

伴生对象中的方法，用于获取storage部分内存大小，计算过程如下：

private def getMaxStorageMemory(conf: SparkConf): Long = {

val systemMaxMemory = conf.getLong("spark.testing.memory", Runtime.getRuntime.maxMemory)

val memoryFraction = conf.getDouble("spark.storage.memoryFraction", 0.6)

val safetyFraction = conf.getDouble("spark.storage.safetyFraction", 0.9)

(systemMaxMemory \* memoryFraction \* safetyFraction).toLong

}

systemMaxMemory是当前Executor的内存大小，虽然可以由参数spark.testing.memory来设定，但是这个参数一般用于做测试，在生产上不建议设置。

memoryFraction是storage内存占整个systemMaxMemory内存的比例，由参数spark.storage.memoryFraction（默认值0.6）来设定。同时为了避免出现OOM的情况，会设定一个安全系数spark.storage.safetyFraction(默认值0.9）。

**（2）getMaxExecutionMemory(conf: SparkConf)**

伴生对象中的方法。用于获取execution部分内存大小。计算过程如下：

private def getMaxExecutionMemory(conf: SparkConf): Long = {

val systemMaxMemory = conf.getLong("spark.testing.memory", Runtime.getRuntime.maxMemory)

val memoryFraction = conf.getDouble("spark.shuffle.memoryFraction", 0.2)

val safetyFraction = conf.getDouble("spark.shuffle.safetyFraction", 0.8)

(systemMaxMemory \* memoryFraction \* safetyFraction).toLong

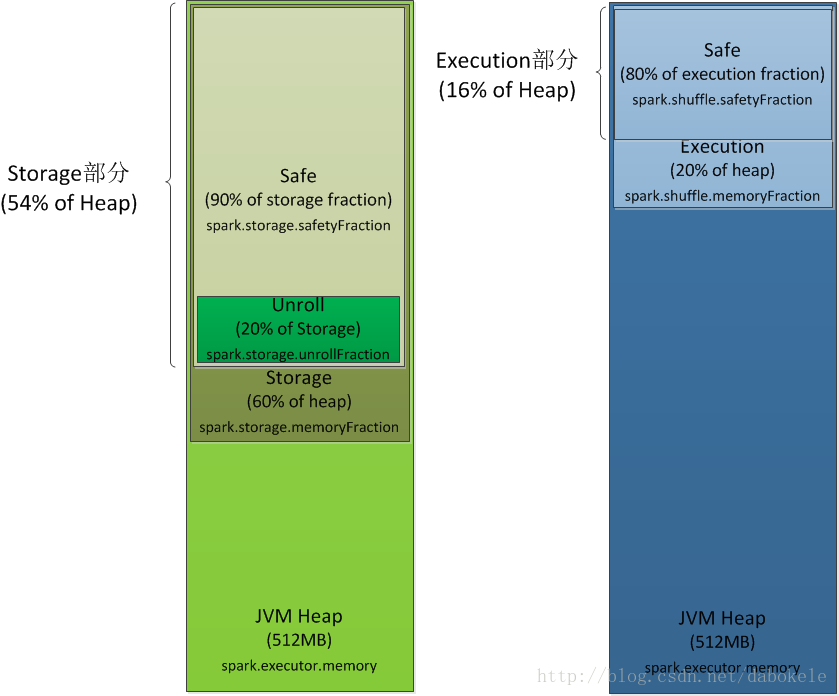
}

memoryFraction即execution部分占所有能使用内存的百分比，由参数spark.shuffle.memoryFraction（默认值是0.2）来确定。

safetyFraction是execution部分的一个安全阈值，由参数spark.shuffle.safetyFraction（默认值是0.8）来确定。

总结一下，如果不引入safety的话，整个executor内存的60%用于storage，20%用于execution，剩下20%用于其他。在引入safetyFraction后，默认情况下storage占了整个executor内存的54%，execution占了16%，那么最终还剩下30%内存用于其他用途。注意在storage和execution中的safetyFraction是不一样的，execution部分的safety值更低。

内存使用情况如下图：



上图中的Unroll部分，并不是静态分配后不变的，它只是表示Unroll部分的内存最多占了整个storage部分的20%，当storage部分对内存需求比较大时，会使用Unroll部分的内存，当有unroll部分内存申请时，storage部分会释放一些内存以满足unroll部分的申请。unroll部分内存的上限是Storage部分的20%。

**（3）acquireExecutionMemory(numBytes: Long, taskAttemptId: Long, memoryMode: MemoryMode)**

申请execution部分内存。根据传入的taskAttemptId，以及需要的内存数numBytes，和当前的MemoryMode是ON\_HEAP还是OFF\_HEAP，从对应的execution内存池中申请内存。这里进一步调用ExecutionMemoryPool的acquireMemory方法进行内存的申请。

ExecutionMemoryPool#acquireMemory方法在最后一部分会介绍到。

**（4）acquireStorageMemory(blockId: BlockId, numBytes: Long, evictedBlocks: Buffer)**

申请storage部分内存。在保证申请的内存数numBytes小于maxStorageMemory后，向storage内存池申请numBytes内存。进一步调用StorageMemoryPool的acquireMemory方法进行内存的申请。

StorageMemoryPool#acquireMemory的执行逻辑在本文最后会有描述。

**（5）qcquireUnrollMemory(blockId: BlockId, numBytes: Long, evictedBlocks: Buffer)**

根据传入numBytes，申请unroll部分内存。首先获取当前storage内存池中unroll部分使用的内存数currentUnrollMemory，以及当前storage内存池剩余内存数freeMemory。内存足够时，直接从storage内存池分配numBytes内存。如果内存不足，则会从storage内存池先释放出一部分内存。整个unroll部分使用的内存不能超过maxUnrollMemory。

override def acquireUnrollMemory(

blockId: BlockId,

numBytes: Long,

evictedBlocks: mutable.Buffer[(BlockId, BlockStatus)]): Boolean = synchronized {

val currentUnrollMemory = storageMemoryPool.memoryStore.currentUnrollMemory

val freeMemory = storageMemoryPool.memoryFree

// When unrolling, we will use all of the existing free memory, and, if necessary,

// some extra space freed from evicting cached blocks. We must place a cap on the

// amount of memory to be evicted by unrolling, however, otherwise unrolling one

// big block can blow away the entire cache.

val maxNumBytesToFree = math.max(0, maxUnrollMemory - currentUnrollMemory - freeMemory)

// Keep it within the range 0 <= X <= maxNumBytesToFree

val numBytesToFree = math.max(0, math.min(maxNumBytesToFree, numBytes - freeMemory))

storageMemoryPool.acquireMemory(blockId, numBytes, numBytesToFree, evictedBlocks)

}

/\*\*

\* A [[MemoryManager]] that statically partitions the heap space into disjoint regions.

\*

\* The sizes of the execution and storage regions are determined through

\* `spark.shuffle.memoryFraction` and `spark.storage.memoryFraction` respectively. The two regions are cleanly separated such that neither usage can borrow memory from the other.

\*/

private[spark] class StaticMemoryManager(

conf: SparkConf,

maxOnHeapExecutionMemory: Long,

override val maxOnHeapStorageMemory: Long,

numCores: Int)

extends MemoryManager(

conf,

numCores,

maxOnHeapStorageMemory,

maxOnHeapExecutionMemory) {

def this(conf: SparkConf, numCores: Int) {

this(

conf,

StaticMemoryManager.getMaxExecutionMemory(conf),

StaticMemoryManager.getMaxStorageMemory(conf),

numCores)

}

// The StaticMemoryManager does not support off-heap storage memory:

offHeapExecutionMemoryPool.incrementPoolSize(offHeapStorageMemoryPool.poolSize)

offHeapStorageMemoryPool.decrementPoolSize(offHeapStorageMemoryPool.poolSize)

// Max number of bytes worth of blocks to evict when unrolling

private val maxUnrollMemory: Long = {

(maxOnHeapStorageMemory \* conf.getDouble("spark.storage.unrollFraction", 0.2)).toLong

}

override def maxOffHeapStorageMemory: Long = 0L

override def acquireStorageMemory(

blockId: BlockId,

numBytes: Long,

memoryMode: MemoryMode): Boolean = synchronized {

require(memoryMode != MemoryMode.OFF\_HEAP,

"StaticMemoryManager does not support off-heap storage memory")

if (numBytes > maxOnHeapStorageMemory) {

// Fail fast if the block simply won't fit

logInfo(s"Will not store $blockId as the required space ($numBytes bytes) exceeds our " +

s"memory limit ($maxOnHeapStorageMemory bytes)")

false

} else {

onHeapStorageMemoryPool.acquireMemory(blockId, numBytes)

}

}

override def acquireUnrollMemory(

blockId: BlockId,

numBytes: Long,

memoryMode: MemoryMode): Boolean = synchronized {

require(memoryMode != MemoryMode.OFF\_HEAP,

"StaticMemoryManager does not support off-heap unroll memory")

val currentUnrollMemory = onHeapStorageMemoryPool.memoryStore.currentUnrollMemory

val freeMemory = onHeapStorageMemoryPool.memoryFree

// When unrolling, we will use all of the existing free memory, and, if necessary,

// some extra space freed from evicting cached blocks. We must place a cap on the

// amount of memory to be evicted by unrolling, however, otherwise unrolling one

// big block can blow away the entire cache.

val maxNumBytesToFree = math.max(0, maxUnrollMemory - currentUnrollMemory - freeMemory)

// Keep it within the range 0 <= X <= maxNumBytesToFree

val numBytesToFree = math.max(0, math.min(maxNumBytesToFree, numBytes - freeMemory))

onHeapStorageMemoryPool.acquireMemory(blockId, numBytes, numBytesToFree)

}

private[memory]

override def acquireExecutionMemory(

numBytes: Long,

taskAttemptId: Long,

memoryMode: MemoryMode): Long = synchronized {

memoryMode match {

case MemoryMode.ON\_HEAP => onHeapExecutionMemoryPool.acquireMemory(numBytes, taskAttemptId)

case MemoryMode.OFF\_HEAP => offHeapExecutionMemoryPool.acquireMemory(numBytes, taskAttemptId)

}

}

}

private[spark] object StaticMemoryManager {

private val MIN\_MEMORY\_BYTES = 32 \* 1024 \* 1024

/\*\*

\* Return the total amount of memory available for the storage region, in bytes.

\*/

private def getMaxStorageMemory(conf: SparkConf): Long = {

val systemMaxMemory = conf.getLong("spark.testing.memory", Runtime.getRuntime.maxMemory)

val memoryFraction = conf.getDouble("spark.storage.memoryFraction", 0.6)

val safetyFraction = conf.getDouble("spark.storage.safetyFraction", 0.9)

(systemMaxMemory \* memoryFraction \* safetyFraction).toLong

}

/\*\*

\* Return the total amount of memory available for the execution region, in bytes.

\*/

private def getMaxExecutionMemory(conf: SparkConf): Long = {

val systemMaxMemory = conf.getLong("spark.testing.memory", Runtime.getRuntime.maxMemory)

if (systemMaxMemory < MIN\_MEMORY\_BYTES) {

throw new IllegalArgumentException(s"System memory $systemMaxMemory must " +

s"be at least $MIN\_MEMORY\_BYTES. Please increase heap size using the --driver-memory " +

s"option or spark.driver.memory in Spark configuration.")

}

if (conf.contains("spark.executor.memory")) {

val executorMemory = conf.getSizeAsBytes("spark.executor.memory")

if (executorMemory < MIN\_MEMORY\_BYTES) {

throw new IllegalArgumentException(s"Executor memory $executorMemory must be at least " +

s"$MIN\_MEMORY\_BYTES. Please increase executor memory using the " +

s"--executor-memory option or spark.executor.memory in Spark configuration.")

}

}

val memoryFraction = conf.getDouble("spark.shuffle.memoryFraction", 0.2)

val safetyFraction = conf.getDouble("spark.shuffle.safetyFraction", 0.8)

(systemMaxMemory \* memoryFraction \* safetyFraction).toLong

}

}

## *UnifiedMemoryManager*

是一种动态内存管理方法，该memoryManager主要是使得execution部分和storage部分的内存不像之前由比例参数限定住，而是两者可以互相借用内存。execution和storage总的内存上限由参数｀spark.memory.fraction（默认0.75）来设定的，这个比例是相对于整个JVM heap来说的。

Storage部分可以申请Execution部分的所有空闲内存，直到Execution内存不足时向Storage发出信号为止。当Execution需要更多内存时，Storage部分会向磁盘spill数据，直到把借用的内存都还上为止。

同样的Execution部分也能向Storage部分借用内存，当Storage需要内存时，Execution中的数据不会马上spill到磁盘，因为Execution使用的内存发生在计算过程中，如果数据丢失就会到账task计算失败。Storage部分只能等待Execution部分主动释放占用的内存。

**1. 变量**

（１）RESERVED\_SYSTEM\_MEMORY\_BYTES

伴生对象的一个属性，值为300MB，是Execution和Storage之外的一部分内存，为系统保留。

private val RESERVED\_SYSTEM\_MEMORY\_BYTES = 300 \* 1024 \* 1024

**2. 方法**

（1）getMaxMemory(conf: SparkConf)

伴生对象的方法。获取execution和storage部分能够使用的总内存大小。计算过程如下：

private def getMaxMemory(conf: SparkConf): Long = {

val systemMemory = conf.getLong("spark.testing.memory", Runtime.getRuntime.maxMemory)

val reservedMemory = conf.getLong("spark.testing.reservedMemory",

if (conf.contains("spark.testing")) 0 else RESERVED\_SYSTEM\_MEMORY\_BYTES)

val minSystemMemory = reservedMemory \* 1.5

if (systemMemory < minSystemMemory) {

throw new IllegalArgumentException(s"System memory $systemMemory must " +

s"be at least $minSystemMemory. Please use a larger heap size.")

}

val usableMemory = systemMemory - reservedMemory

val memoryFraction = conf.getDouble("spark.memory.fraction", 0.75)

(usableMemory \* memoryFraction).toLong

}

systemMemory即Executor的内存大小。systemMemory要求最小为reservedMemory的1.5倍，否则直接抛出异常信息。

reservedMemory是为系统保留的内存大小，可以由参数spark.testing.reservedMemory确定，默认值为上面的300MB。如果为默认值的话，那么对应的会要求systemMemory最小为450MB。

memoryFraction是整个execution和storage共用的最大内存比例，由参数spark.memory.fraction（默认值0.75）来决定。那么还剩下0.25的内存作为User Memory部分使用。

那么对一个1GB内存的Executor来说，在默认情况下，可使用的内存大小为（1024 - 300） \* 0.75 = 543MB

（2）maxStorageMemory

storage部分最大内存数。由最大内存数减去ON\_HEAP的execution使用的内存大小即可得到。

override def maxStorageMemory: Long = synchronized {

maxMemory - onHeapExecutionMemoryPool.memoryUsed

}

（3）acquireExecutionMemory(numBytes: Long, taskAttemptId: Long, memoryMode: MemoryMode)

为当前的taskAttemptId申请最多numBytes的内存，如果内存不足则返回0。

由于这里涉及到的都是Executor JVM Heap中的内存，所以如果是OFF\_HEAP模式，直接从offHeapExecution内存池分配。对memoryMode为ON\_HEAP的进行如下处理。

ExecutionMemoryPool#acquireMemory

onHeapExecutionMemoryPool.acquireMemory(numBytes, taskAttemptId, maybeGrowExecutionPool, computeMaxExecutionPoolSize)

maybeGrowExecutionPool方法会去释放Storage中保存的数据所占用的内存，收缩Storage部分内存大小，从而增大Execution部分。当Execution部分剩余内存小于numBytes时，执行如下逻辑

val memoryReclaimableFromStorage = math.max(storageMemoryPool.memoryFree, storageMemoryPool.poolSize - storageRegionSize)

if (memoryReclaimableFromStorage > 0) {

// Only reclaim as much space as is necessary and available:

val spaceReclaimed = storageMemoryPool.shrinkPoolToFreeSpace(

math.min(extraMemoryNeeded, memoryReclaimableFromStorage))

onHeapExecutionMemoryPool.incrementPoolSize(spaceReclaimed)

}

如果memoryReclaimableFromStorage大于0，表示storage部分能够分配一些内存给Execution部分，这个值最多不能超过此刻storage内存池的剩余空闲内存。然后取出spaceReclaimed的内存给Execution部分，实时调整Storage和Execution内存池的大小。

在内存区域调整后，会重新计算当前Execution内存池大小computeMaxExecutionPoolSize。然后调用ExecutionMemoryPool#acquireMemory方法向Execution内存池申请内存。该方法在本文最后会有描述。

（4）acquireStorageMemory(blockId: BlockId, numBytes: Long, evictedBlocks: Buffer)

首先申请的storage内存numBytes不能超过storage部分内存的最大值maxStorageMemory。

然后当storage部分内存不足以满足此次申请时，尝试向execution内存池借用内存，借到的内存大小为min(execution内存池剩余内存，numBytes)，并且实时调整execution和storage内存池的大小，如下面的代码所描述的。

if (numBytes > storageMemoryPool.memoryFree) {

// There is not enough free memory in the storage pool, so try to borrow free memory from

// the execution pool.

val memoryBorrowedFromExecution = Math.min(onHeapExecutionMemoryPool.memoryFree, numBytes)

onHeapExecutionMemoryPool.decrementPoolSize(memoryBorrowedFromExecution)

storageMemoryPool.incrementPoolSize(memoryBorrowedFromExecution)

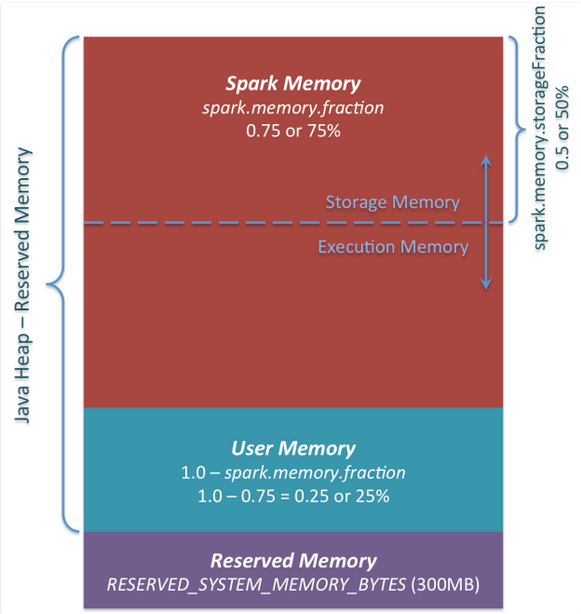
}

最后，向storageMemoryPool申请numBytes的内存。这一部分逻辑在本文最后StorageMemoryPool#acquireMemory中会有详细描述。

（5）acquireUnrollMemory

直接调用方法（4）从storage部分申请内存。

UnifiedMemoryManager的内存分配情况如下图所示：UnifiedMemoryManager内存管理。



该图片参考自：https://0x0fff.com/spark-memory-management/#comment-1188

private[spark] class UnifiedMemoryManager private[memory] (

conf: SparkConf,

val maxHeapMemory: Long,

onHeapStorageRegionSize: Long,

numCores: Int)

extends MemoryManager(

conf,

numCores,

onHeapStorageRegionSize,

maxHeapMemory - onHeapStorageRegionSize) {

private def assertInvariants(): Unit = {

assert(onHeapExecutionMemoryPool.poolSize + onHeapStorageMemoryPool.poolSize == maxHeapMemory)

assert(

offHeapExecutionMemoryPool.poolSize + offHeapStorageMemoryPool.poolSize == maxOffHeapMemory)

}

assertInvariants()

override def maxOnHeapStorageMemory: Long = synchronized {

maxHeapMemory - onHeapExecutionMemoryPool.memoryUsed

}

override def maxOffHeapStorageMemory: Long = synchronized {

maxOffHeapMemory - offHeapExecutionMemoryPool.memoryUsed

}

/\*\*

\* Try to acquire up to `numBytes` of execution memory for the current task and return the

\* number of bytes obtained, or 0 if none can be allocated.

\*

\* This call may block until there is enough free memory in some situations, to make sure each

\* task has a chance to ramp up to at least 1 / 2N of the total memory pool (where N is the # of

\* active tasks) before it is forced to spill. This can happen if the number of tasks increase

\* but an older task had a lot of memory already.

\*/

override private[memory] def acquireExecutionMemory(

numBytes: Long,

taskAttemptId: Long,

memoryMode: MemoryMode): Long = synchronized {

assertInvariants()

assert(numBytes >= 0)

val (executionPool, storagePool, storageRegionSize, maxMemory) = memoryMode match {

case MemoryMode.ON\_HEAP => (

onHeapExecutionMemoryPool,

onHeapStorageMemoryPool,

onHeapStorageRegionSize,

maxHeapMemory)

case MemoryMode.OFF\_HEAP => (

offHeapExecutionMemoryPool,

offHeapStorageMemoryPool,

offHeapStorageMemory,

maxOffHeapMemory)

}

/\*\*

\* Grow the execution pool by evicting cached blocks, thereby shrinking the storage pool.

\*

\* When acquiring memory for a task, the execution pool may need to make multiple

\* attempts. Each attempt must be able to evict storage in case another task jumps in

\* and caches a large block between the attempts. This is called once per attempt.

\*/

def maybeGrowExecutionPool(extraMemoryNeeded: Long): Unit = {

if (extraMemoryNeeded > 0) {

// There is not enough free memory in the execution pool, so try to reclaim memory from

// storage. We can reclaim any free memory from the storage pool. If the storage pool

// has grown to become larger than `storageRegionSize`, we can evict blocks and reclaim

// the memory that storage has borrowed from execution.

val memoryReclaimableFromStorage = math.max(

storagePool.memoryFree,

storagePool.poolSize - storageRegionSize)

if (memoryReclaimableFromStorage > 0) {

// Only reclaim as much space as is necessary and available:

val spaceToReclaim = storagePool.freeSpaceToShrinkPool(

math.min(extraMemoryNeeded, memoryReclaimableFromStorage))

storagePool.decrementPoolSize(spaceToReclaim)

executionPool.incrementPoolSize(spaceToReclaim)

}

}

}

/\*\*

\* The size the execution pool would have after evicting storage memory.

\*

\* The execution memory pool divides this quantity among the active tasks evenly to cap

\* the execution memory allocation for each task. It is important to keep this greater

\* than the execution pool size, which doesn't take into account potential memory that

\* could be freed by evicting storage. Otherwise we may hit SPARK-12155.

\*

\* Additionally, this quantity should be kept below `maxMemory` to arbitrate fairness

\* in execution memory allocation across tasks, Otherwise, a task may occupy more than

\* its fair share of execution memory, mistakenly thinking that other tasks can acquire

\* the portion of storage memory that cannot be evicted.

\*/

def computeMaxExecutionPoolSize(): Long = {

maxMemory - math.min(storagePool.memoryUsed, storageRegionSize)

}

executionPool.acquireMemory(

numBytes, taskAttemptId, maybeGrowExecutionPool, computeMaxExecutionPoolSize)

}

override def acquireStorageMemory(

blockId: BlockId,

numBytes: Long,

memoryMode: MemoryMode): Boolean = synchronized {

assertInvariants()

assert(numBytes >= 0)

val (executionPool, storagePool, maxMemory) = memoryMode match {

case MemoryMode.ON\_HEAP => (

onHeapExecutionMemoryPool,

onHeapStorageMemoryPool,

maxOnHeapStorageMemory)

case MemoryMode.OFF\_HEAP => (

offHeapExecutionMemoryPool,

offHeapStorageMemoryPool,

maxOffHeapStorageMemory)

}

if (numBytes > maxMemory) {

// Fail fast if the block simply won't fit

logInfo(s"Will not store $blockId as the required space ($numBytes bytes) exceeds our " +

s"memory limit ($maxMemory bytes)")

return false

}

if (numBytes > storagePool.memoryFree) {

// There is not enough free memory in the storage pool, so try to borrow free memory from

// the execution pool.

val memoryBorrowedFromExecution = Math.min(executionPool.memoryFree,

numBytes - storagePool.memoryFree)

executionPool.decrementPoolSize(memoryBorrowedFromExecution)

storagePool.incrementPoolSize(memoryBorrowedFromExecution)

}

storagePool.acquireMemory(blockId, numBytes)

}

override def acquireUnrollMemory(

blockId: BlockId,

numBytes: Long,

memoryMode: MemoryMode): Boolean = synchronized {

acquireStorageMemory(blockId, numBytes, memoryMode)

}

}

object UnifiedMemoryManager {

// Set aside a fixed amount of memory for non-storage, non-execution purposes.

// This serves a function similar to `spark.memory.fraction`, but guarantees that we reserve

// sufficient memory for the system even for small heaps. E.g. if we have a 1GB JVM, then

// the memory used for execution and storage will be (1024 - 300) \* 0.6 = 434MB by default.

private val RESERVED\_SYSTEM\_MEMORY\_BYTES = 300 \* 1024 \* 1024

def apply(conf: SparkConf, numCores: Int): UnifiedMemoryManager = {

val maxMemory = getMaxMemory(conf)

new UnifiedMemoryManager(

conf,

maxHeapMemory = maxMemory,

onHeapStorageRegionSize =

(maxMemory \* conf.getDouble("spark.memory.storageFraction", 0.5)).toLong,

numCores = numCores)

}

/\*\*

\* Return the total amount of memory shared between execution and storage, in bytes.

\*/

private def getMaxMemory(conf: SparkConf): Long = {

val systemMemory = conf.getLong("spark.testing.memory", Runtime.getRuntime.maxMemory)

val reservedMemory = conf.getLong("spark.testing.reservedMemory",

if (conf.contains("spark.testing")) 0 else RESERVED\_SYSTEM\_MEMORY\_BYTES)

val minSystemMemory = (reservedMemory \* 1.5).ceil.toLong

if (systemMemory < minSystemMemory) {

throw new IllegalArgumentException(s"System memory $systemMemory must " +

s"be at least $minSystemMemory. Please increase heap size using the --driver-memory " +

s"option or spark.driver.memory in Spark configuration.")

}

// SPARK-12759 Check executor memory to fail fast if memory is insufficient

if (conf.contains("spark.executor.memory")) {

val executorMemory = conf.getSizeAsBytes("spark.executor.memory")

if (executorMemory < minSystemMemory) {

throw new IllegalArgumentException(s"Executor memory $executorMemory must be at least " +

s"$minSystemMemory. Please increase executor memory using the " +

s"--executor-memory option or spark.executor.memory in Spark configuration.")

}

}

val usableMemory = systemMemory - reservedMemory

val memoryFraction = conf.getDouble("spark.memory.fraction", 0.6)

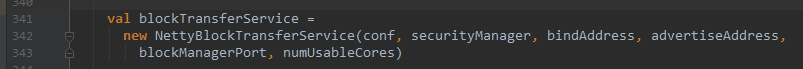
(usableMemory \* memoryFraction).toLong

}

}

# BlockTransferService服务

BlockTransferService默认为NettyBlockTransferService，使用netty提供的异步事件驱动的网络应用框架，提供web服务和客户端，获取远程节点上的Block集合。



源码：

/\*\*

\* A BlockTransferService that uses Netty to fetch a set of blocks at time.

\*/

private[spark] class NettyBlockTransferService(

conf: SparkConf,

securityManager: SecurityManager,

bindAddress: String,

override val hostName: String,

\_port: Int,

numCores: Int)

extends BlockTransferService {

// TODO: Don't use Java serialization, use a more cross-version compatible serialization format.

private val serializer = new JavaSerializer(conf)

private val authEnabled = securityManager.isAuthenticationEnabled()

private val transportConf = SparkTransportConf.fromSparkConf(conf, "shuffle", numCores)

private[this] var transportContext: TransportContext = \_

private[this] var server: TransportServer = \_

private[this] var clientFactory: TransportClientFactory = \_

private[this] var appId: String = \_

override def init(blockDataManager: BlockDataManager): Unit = {

val rpcHandler = new NettyBlockRpcServer(conf.getAppId, serializer, blockDataManager)

var serverBootstrap: Option[TransportServerBootstrap] = None

var clientBootstrap: Option[TransportClientBootstrap] = None

if (authEnabled) {

serverBootstrap = Some(new AuthServerBootstrap(transportConf, securityManager))

clientBootstrap = Some(new AuthClientBootstrap(transportConf, conf.getAppId, securityManager))

}

transportContext = new TransportContext(transportConf, rpcHandler)

clientFactory = transportContext.createClientFactory(clientBootstrap.toSeq.asJava)

server = createServer(serverBootstrap.toList)

appId = conf.getAppId

logInfo(s"Server created on ${hostName}:${server.getPort}")

}

/\*\* Creates and binds the TransportServer, possibly trying multiple ports. \*/

private def **createServer**(bootstraps: List[TransportServerBootstrap]): TransportServer = {

def startService(port: Int): (TransportServer, Int) = {

val server = transportContext.createServer(bindAddress, port, bootstraps.asJava)

(server, server.getPort)

}

Utils.startServiceOnPort(\_port, startService, conf, getClass.getName).\_1

}

override def **fetchBlocks**(

host: String,

port: Int,

execId: String,

blockIds: Array[String],

listener: BlockFetchingListener,

shuffleFiles: Array[File]): Unit = {

logTrace(s"Fetch blocks from $host:$port (executor id $execId)")

try {

val blockFetchStarter = new RetryingBlockFetcher.BlockFetchStarter {

override def createAndStart(blockIds: Array[String], listener: BlockFetchingListener) {

val client = clientFactory.createClient(host, port)

new OneForOneBlockFetcher(client, appId, execId, blockIds.toArray, listener,

transportConf, shuffleFiles).start()

}

}

val maxRetries = transportConf.maxIORetries()

if (maxRetries > 0) {

// Note this Fetcher will correctly handle maxRetries == 0; we avoid it just in case there's

// a bug in this code. We should remove the if statement once we're sure of the stability.

new RetryingBlockFetcher(transportConf, blockFetchStarter, blockIds, listener).start()

} else {

blockFetchStarter.createAndStart(blockIds, listener)

}

} catch {

case e: Exception =>

logError("Exception while beginning fetchBlocks", e)

blockIds.foreach(listener.onBlockFetchFailure(\_, e))

}

}

override def port: Int = server.getPort

override def **uploadBlock**(

hostname: String,

port: Int,

execId: String,

blockId: BlockId,

blockData: ManagedBuffer,

level: StorageLevel,

classTag: ClassTag[\_]): Future[Unit] = {

val result = Promise[Unit]()

val client = clientFactory.createClient(hostname, port)

// StorageLevel and ClassTag are serialized as bytes using our JavaSerializer.

// Everything else is encoded using our binary protocol.

val metadata = JavaUtils.bufferToArray(serializer.newInstance().serialize((level, classTag)))

// Convert or copy nio buffer into array in order to serialize it.

val array = JavaUtils.bufferToArray(blockData.nioByteBuffer())

client.**sendRpc**(new UploadBlock(appId, execId, blockId.toString, metadata, array).toByteBuffer,

new RpcResponseCallback {

override def onSuccess(response: ByteBuffer): Unit = {

logTrace(s"Successfully uploaded block $blockId")

result.success((): Unit)

}

override def onFailure(e: Throwable): Unit = {

logError(s"Error while uploading block $blockId", e)

result.failure(e)

}

})

result.future

}

override def close(): Unit = {

if (server != null) {

server.close()

}

if (clientFactory != null) {

clientFactory.close()

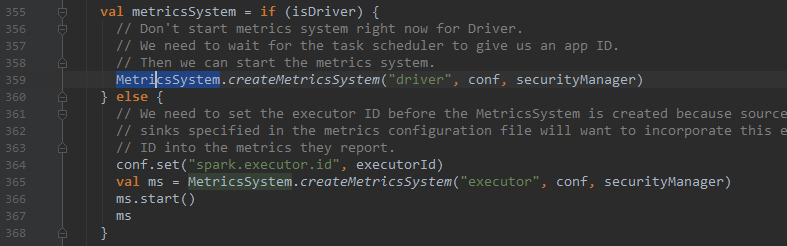
}

}

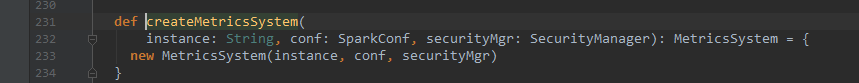
}

# 测量系统MetriceSystem

Spark通过metricsSystem创建测量系统。其中分为driver端和executor端的测量方法。



通过创建MetriceSystem对象的方法创建测量系统。



/\*\*

\* Spark Metrics System, created by a specific "instance", combined by source,

\* sink, periodically polls source metrics data to sink destinations.

\*

\* "instance" specifies "who" (the role) uses the metrics system. In Spark, there are several roles

\* like master, worker, executor, client driver. These roles will create metrics system

\* for monitoring. So, "instance" represents these roles. Currently in Spark, several instances

\* have already implemented: master, worker, executor, driver, applications.

\*

\* "source" specifies "where" (source) to collect metrics data from. In metrics system, there exists

\* two kinds of source:

\* 1. Spark internal source, like MasterSource, WorkerSource, etc, which will collect

\* Spark component's internal state, these sources are related to instance and will be

\* added after a specific metrics system is created.

\* 2. Common source, like JvmSource, which will collect low level state, is configured by

\* configuration and loaded through reflection.

\*

\* "sink" specifies "where" (destination) to output metrics data to. Several sinks can

\* coexist and metrics can be flushed to all these sinks.

\*

\* Metrics configuration format is like below:

\* [instance].[sink|source].[name].[options] = xxxx

\*

\* [instance] can be "master", "worker", "executor", "driver", "applications" which means only

\* the specified instance has this property.

\* wild card "\*" can be used to replace instance name, which means all the instances will have

\* this property.

\*

\* [sink|source] means this property belongs to source or sink. This field can only be

\* source or sink.

\*

\* [name] specify the name of sink or source, if it is custom defined.

\*

\* [options] represent the specific property of this source or sink.

\*/

private[spark] class MetricsSystem private (

val instance: String,

conf: SparkConf,

securityMgr: SecurityManager)

extends Logging {

private[this] val metricsConfig = new MetricsConfig(conf)

private val sinks = new mutable.ArrayBuffer[Sink]

private val sources = new mutable.ArrayBuffer[Source]

private val registry = new MetricRegistry()

private var running: Boolean = false

// Treat MetricsServlet as a special sink as it should be exposed to add handlers to web ui

private var metricsServlet: Option[MetricsServlet] = None

/\*\*

\* Get any UI handlers used by this metrics system; can only be called after start().

\*/

def getServletHandlers: Array[ServletContextHandler] = {

require(running, "Can only call getServletHandlers on a running MetricsSystem")

metricsServlet.map(\_.getHandlers(conf)).getOrElse(Array())

}

metricsConfig.initialize()

def start() {

require(!running, "Attempting to start a MetricsSystem that is already running")

running = true

StaticSources.allSources.foreach(registerSource)

registerSources()

registerSinks()

sinks.foreach(\_.start)

}

def stop() {

if (running) {

sinks.foreach(\_.stop)

} else {

logWarning("Stopping a MetricsSystem that is not running")

}

running = false

}

def report() {

sinks.foreach(\_.report())

}

/\*\*

\* Build a name that uniquely identifies each metric source.

\* The name is structured as follows: <app ID>.<executor ID (or "driver")>.<source name>.

\* If either ID is not available, this defaults to just using <source name>.

\*

\* @param source Metric source to be named by this method.

\* @return An unique metric name for each combination of

\* application, executor/driver and metric source.

\*/

private[spark] def buildRegistryName(source: Source): String = {

val metricsNamespace = conf.get(METRICS\_NAMESPACE).orElse(conf.getOption("spark.app.id"))

val executorId = conf.getOption("spark.executor.id")

val defaultName = MetricRegistry.name(source.sourceName)

if (instance == "driver" || instance == "executor") {

if (metricsNamespace.isDefined && executorId.isDefined) {

MetricRegistry.name(metricsNamespace.get, executorId.get, source.sourceName)

} else {

// Only Driver and Executor set spark.app.id and spark.executor.id.

// Other instance types, e.g. Master and Worker, are not related to a specific application.

if (metricsNamespace.isEmpty) {

logWarning(s"Using default name $defaultName for source because neither " +

s"${METRICS\_NAMESPACE.key} nor spark.app.id is set.")

}

if (executorId.isEmpty) {

logWarning(s"Using default name $defaultName for source because spark.executor.id is " +

s"not set.")

}

defaultName

}

} else { defaultName }

}

def getSourcesByName(sourceName: String): Seq[Source] =

sources.filter(\_.sourceName == sourceName)

def registerSource(source: Source) {

sources += source

try {

val regName = buildRegistryName(source)

registry.register(regName, source.metricRegistry)

} catch {

case e: IllegalArgumentException => logInfo("Metrics already registered", e)

}

}

def removeSource(source: Source) {

sources -= source

val regName = buildRegistryName(source)

registry.removeMatching(new MetricFilter {

def matches(name: String, metric: Metric): Boolean = name.startsWith(regName)

})

}

private def registerSources() {

val instConfig = metricsConfig.getInstance(instance)

val sourceConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.SOURCE\_REGEX)

// Register all the sources related to instance

sourceConfigs.foreach { kv =>

val classPath = kv.\_2.getProperty("class")

try {

val source = Utils.classForName(classPath).newInstance()

registerSource(source.asInstanceOf[Source])

} catch {

case e: Exception => logError("Source class " + classPath + " cannot be instantiated", e)

}

}

}

private def registerSinks() {

val instConfig = metricsConfig.getInstance(instance)

val sinkConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.SINK\_REGEX)

sinkConfigs.foreach { kv =>

val classPath = kv.\_2.getProperty("class")

if (null != classPath) {

try {

val sink = Utils.classForName(classPath)

.getConstructor(classOf[Properties], classOf[MetricRegistry], classOf[SecurityManager])

.newInstance(kv.\_2, registry, securityMgr)

if (kv.\_1 == "servlet") {

metricsServlet = Some(sink.asInstanceOf[MetricsServlet])

} else {

sinks += sink.asInstanceOf[Sink]

}

} catch {

case e: Exception =>

logError("Sink class " + classPath + " cannot be instantiated")

throw e

}

}

}

}

}

private[spark] object MetricsSystem {

val SINK\_REGEX = "^sink\\.(.+)\\.(.+)".r

val SOURCE\_REGEX = "^source\\.(.+)\\.(.+)".r

private[this] val MINIMAL\_POLL\_UNIT = TimeUnit.SECONDS

private[this] val MINIMAL\_POLL\_PERIOD = 1

def checkMinimalPollingPeriod(pollUnit: TimeUnit, pollPeriod: Int) {

val period = MINIMAL\_POLL\_UNIT.convert(pollPeriod, pollUnit)

if (period < MINIMAL\_POLL\_PERIOD) {

throw new IllegalArgumentException("Polling period " + pollPeriod + " " + pollUnit +

" below than minimal polling period ")

}

}

def createMetricsSystem(

instance: String, conf: SparkConf, securityMgr: SecurityManager): MetricsSystem = {

new MetricsSystem(instance, conf, securityMgr)

}

}