Spark Context

Spark的执行过程如下图所示：



其中Spark Driver用于提交用户应用程序，其初始化围绕着SparkContext，相当于Spark应用程序的发送机引擎。SparkContext的初始化步骤如下：

# 初始化SparkConf

SparkContext的配置参数由SparkConf（主构造参数）负责，实现代码如下：

*class SparkContext(config: SparkConf) extends Logging {*

*// The call site where this SparkContext was constructed.private*

*val creationSite: CallSite = Utils.getCallSite()……*

*}*

上面代码截图中的CallSite存储了线程栈中最靠近栈顶的用户类以及最靠近栈底的Scala或者Spark核心类信息。SparkConf使用ConcurrentHashMap对象保存配置：

*private val settings = new ConcurrentHashMap[String, String]()*

# 创建Spark执行环境SparkEnv

SparkEnv是Spark的执行环境对象，其中包括众多与Executor执行相关的对象，其存在于Driver或者CoarseGranedExecutorBackend进程。创建SparkEnv主要使用createDriverEnv，该方法有三个参数：conf, isLocal和listenerBus。

*sc = new SparkContext(conf) {*

*override private[spark] def createSparkEnv(*

*conf: SparkConf,*

*isLocal: Boolean,*

*listenerBus: LiveListenerBus): SparkEnv = {*

*outputCommitCoordinator = spy(new OutputCommitCoordinator(conf, isDriver = true))*

*SparkEnv.createDriverEnv(conf, isLocal, listenerBus,*

*SparkContext.numDriverCores(master), Some(outputCommitCoordinator))*

*}*

*}*

SparkEnv.createDriverEnv的构造步骤如下：

1. 创建安全管理器SecurityManager

SecurityManager主要有用于权限设置，比如使用yarn作为资源调度框架时，用于生成secret key进行登录，最后给当前系统设置默认的口令认证实例。

*private val secretKey = generateSecretKey()*

*if (authOn) {*

*Authenticator.setDefault(*

*new Authenticator() {*

*override def getPasswordAuthentication(): PasswordAuthentication = {*

*var passAuth: PasswordAuthentication = null*

*val userInfo = getRequestingURL().getUserInfo()*

*if (userInfo != null) {*

*val parts = userInfo.split(":", 2)*

*passAuth = new PasswordAuthentication(parts(0), parts(1).toCharArray())*

*}*

*return passAuth*

*}*

*}*

*)*

*}*

Spark支持通过共享秘钥进行认证，在Spark on YARN部署模式下通过修改启用认证功能的参数为：*spark.authenticate*来配置，配置后自动产生并分发共享秘钥，每个应用程序都使用唯一的共享秘钥。此参数控制Spark通信协议使用共享秘钥进行认证，这种认证方式基于握手机制，以确保通信双方都有相同的共享秘钥才能通信，如果共享秘钥不一致，则双方将无法通信。

SecurityManager主要用于以下：

1. Web UI，通过参数spark.ui.filters来实现Spark UI的Security，例如用户实现不让其他用户看到敏感数据(spark.ui.view.acls)，通过设置ACL列表（spark.modify.acls）控制哪些用户可以修改正在运行的spark应用程序，开启参数：spark.acls.enable。
2. Event log：使用事件日志记录功能（spark.eventLog.dir），可以通过securityMananger赋予合适的权限，保证日志文件的安全性，开启参数：spark.eventLog.enabled，其处理的实现为SparkListenerEvent: StageCompleted,TaskStarted等
3. 网络安全，主要是Spark用于网络通信的配置，包括配置HTTP协议的SSL，RPC终端及block transfer service的SASL加密及Shuffle文件的加密。
4. 基于Netty的RpcEnv，分布式通信系统

RpcEnv是Spark集群RPC通信的基础服务环境，用于各个组件之间的通信，每个节点之间（Driver或者Worker）组件的Endpoint和对应EndpointRef之间的信息通信和方法调用都是通过RpcEnv作协调，通信底层是通过Netty NIO框架实现（2.0之后统一由Netty替换成Akka，实现了通信传输统一化）。SparkEnv中的RpcEnv的使用如下：

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

*val rpcEnv = RpcEnv.create(systemName, bindAddress, advertiseAddress, port.getOrElse(-1), conf,*

*securityManager, numUsableCores, !isDriver)*

RpcEnv分为Driver及Executor System，下图是RpcEnv的几个主要类：



1. RpcEnv（类似于ActorySystem对象），RPC环境，所有的RpcEndpoint需要注册到该对象中用于接受消息，注册时指定name,RpcEnv将会处理从RpcEndpointRef和远程节点发送过来的消息，然后发送给消息的Endpoint处理，对于接收到的异常使用RpcCallContext处理。其实现类为NettyRpcEnv，该类中包含Netty Server及对消息的处理逻辑，NettyRpcEnvFactory用于创建一个RpcEnv。
2. RpcAddress，用来表示主机名和端口号，其伴生对象用于从URL等构造RpcAddress对象
3. RpcTimeout，超时时间，其中awaitResult在规定的时间内返回结果对象，伴生对象主要是配置文件中获取时间值后生成该对象
4. RpcEnvConfig，用于构建RpcEnv的配置对象
5. RpcEndpoint，进程间调用的Endpoint，当一个消息到来时
6. RpcEndpointRef，一个远程RpcEndpoint的引用，通过它可以给远程Endpoint发送消息，可以是同步或者异步，其映射为一个地址，主要的实现类为NettyRpcEndpointRef，其通信端为client:TranspointClient

Driver和CoarseGrainedSchedulerBackend之间的交互过程如下图所示：



Driver启动过程中RpcEnv初始化后将endpoint写入到SparkConf中，代码如下：

*if (isDriver) {*

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

*}*

Client及Cluster模式下,Spark AM将SparkConf发送给CoarseGrainedSchedulerBackend，启动后调用registerRpcEndpoint，将启动的Endpoint发送给Driver<

*def registerRpcEndpoint(name: String, endpoint: RpcEndpoint): NettyRpcEndpointRef = {*

*val addr = RpcEndpointAddress(nettyEnv.address, name)*

*val endpointRef = new NettyRpcEndpointRef(nettyEnv.conf, addr, nettyEnv)*

*synchronized {*

*if (stopped) {*

*throw new IllegalStateException("RpcEnv has been stopped")*

*}*

*if (endpoints.putIfAbsent(name, new EndpointData(name, endpoint, endpointRef)) != null) {*

*throw new IllegalArgumentException(s"There is already an RpcEndpoint called $name")*

*}*

*val data = endpoints.get(name)*

*endpointRefs.put(data.endpoint, data.ref)*

*receivers.offer(data) // for the OnStart message*

*}*

*endpointRef*

*}*

Driver与CoarseGrainedSchedulerBackend之间以Message进行交互，Driver处理过程如下：

*private class MessageLoop extends Runnable {*

*override def run(): Unit = {*

*try {*

*while (true) {*

*try {*

*val data = receivers.take()*

*if (data == PoisonPill) {*

*receivers.offer(PoisonPill)*

*return*

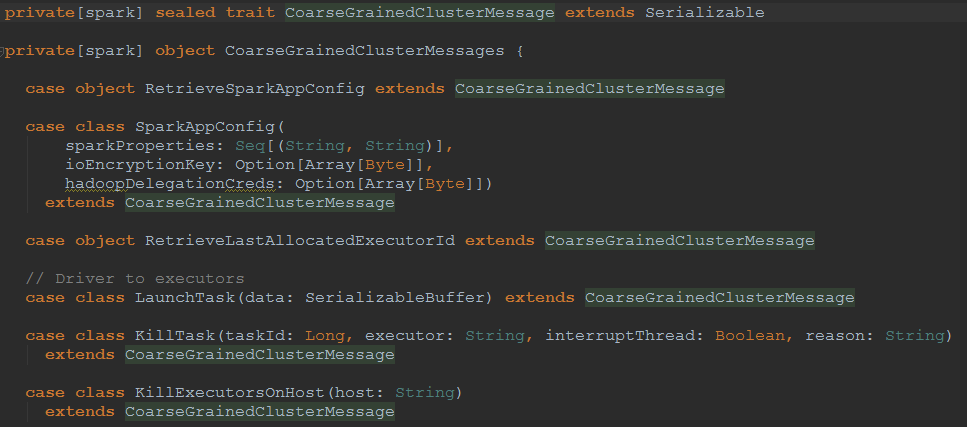
*}*

*data.inbox.process(Dispatcher.this)*

*} ……*

*}*

Message如下：



1. Spark序列化

序列化常用于网络传输和数据持久化以便于存储和传输，Spark通过两种方式来创建序列化器，如下所示：

*val serializer = instantiateClassFromConf[Serializer](*

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

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

*val closureSerializer = new JavaSerializer(conf)*

1. 创建广播管理器BroadcastManager

共享变量允许开发人员在每个节点缓存只读的变量，而不是在任务之间传递这些变量。例如，使用广播变量能够高效地在集群每个节点创建大数据集的复本。同时，Spark还使用高效的广播算法分发这些变量，从而减少通信的开销。可以调用SparkContext.broadcast(v)创建一个广播变量v，该广播变量封装在v变量中，可使用获取该变量的value的方法进行访问。当广播变量创建后，在集群中所有函数都以变量v代表该广播变量，并且该变量v一次性分发到各节点上。

可以用SparkContext将一个变量广播到所有Executor上，使得所有Executor都能获取这个变量代表的数据。在SparkEnv中BroadcastManager用于将配置信息和序列化后的RDD、Job以及ShuffleDependency等信息在本地存储，如果为了容灾，也会复制到其他节点上。BroadcastManager的代码实现：

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

BroadcastManater必须在初始化方法initialize被调用后才能生效，initialize方法实际上利用反射生成广播工厂实现broadFactory（配置参数：spark.broadcast.factory，默认为org.apache.sp

ark.broadcast.TorrentBroadcastFactory）。类图如下所示：



BroadcastManager的实现的实现如下：

*// Called by SparkContext or Executor before using Broadcast*

*private def initialize() {*

*synchronized {*

*if (!initialized) {*

*broadcastFactory = new TorrentBroadcastFactory*

*broadcastFactory.initialize(isDriver, conf, securityManager)*

*initialized = true*

*}*

*}*

*}*

*private val nextBroadcastId = new AtomicLong(0)*

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

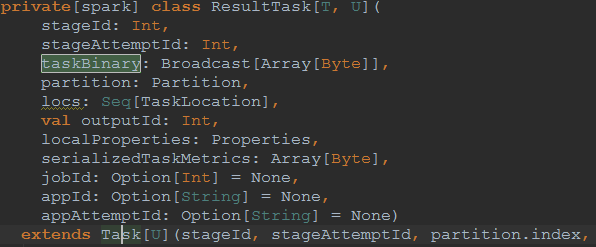
*broadcastFactory.newBroadcast[T](value\_, isLocal, nextBroadcastId.getAndIncrement())*

*}*

SparkContext.broadcast执行过程如下：



1. Driver使用SparkContext#broadcast(value)，将需要broadcast的数据先写入到Driver的BlockManager（StorageLevel为MEMORY\_AND\_DISK）中。
2. 在Driver中，通过writeBlocks，先把data序列化成ChunkedByteBuffer，然后切割成BLOCK\_SIZE大小的data block，然后将block存放到Driver的Block Manager中，并将信息写入到meta信息中并通知BlockManagerMaster（Driver和Executor均可访问的全局信息）。
3. CoarseGrainedSchedulerBackend在接收到Task(ShuffleMapTask和ResultTask)后，Task的初始化参数中包含Broadcast，举例如下：



Executor在serialized task（反序列化task, BroadCast[]].value）后，会调用TorrentBroadcast

.readBroadcastTask，获取BroadCast对象，源码如下所示：

*private def readBroadcastBlock(): T = Utils.tryOrIOException {*

*TorrentBroadcast.synchronized {*

*setConf(SparkEnv.get.conf)*

*val blockManager = SparkEnv.get.blockManager*

*//从本地BlockManager读取数据*

*blockManager.getLocalValues(broadcastId) match {*

*case Some(blockResult) =>*

*if (blockResult.data.hasNext) {*

*val x = blockResult.data.next().asInstanceOf[T]*

*releaseLock(broadcastId)*

*x*

*} else {*

*throw new SparkException(s"Failed to get locally stored broadcast data: $broadcastId")*

*}*

*//从本地获取不到数据，连接Driver的BlockManagerMaster获取data的meta信息，然后基于元数据，然后fetch dataBlock。*

*case None =>*

*logInfo("Started reading broadcast variable " + id)*

*val startTimeMs = System.currentTimeMillis()*

*val blocks = readBlocks()*

*logInfo("Reading broadcast variable " + id + " took" + Utils.getUsedTimeMs(startTimeMs))*

*try {*

*val obj = TorrentBroadcast.unBlockifyObject[T](*

*blocks.map(\_.toInputStream()), SparkEnv.get.serializer, compressionCodec)*

*// Store the merged copy in BlockManager so other tasks on this executor don't*

*// need to re-fetch it.*

*val storageLevel = StorageLevel.MEMORY\_AND\_DISK*

*if (!blockManager.putSingle(broadcastId, obj, storageLevel, tellMaster = false)) {*

*throw new SparkException(s"Failed to store $broadcastId in BlockManager")*

*}*

*obj*

*} finally {*

*blocks.foreach(\_.dispose())*

*}*

*}*

*}*

*}*

https://www.kancloud.cn/kancloud/spark-internals/45238

1. map任务输出跟踪器MapOutputTracker

mapOutputTracker用于跟踪map阶段任务的输出状态，此状态便于reduce阶段获取地址及中间输出结果。每个Map或Reduce任务都会有其唯一标识，分别是mapId和reduceId。每个reduce任务的输入可能是多个map任务的输出，reduce会对各个map任务的所在节点拉取Block，这一过程称为Shuffle。每个Shuffle过程都有唯一标识shuffleId。在SparkEnv中MapOutputTracker的创建如下：

*val mapOutputTracker = if (isDriver) {*

*new MapOutputTrackerMaster(conf, broadcastManager, isLocal)*

*} else {*

*new MapOutputTrackerWorker(conf)*

*}*

其中MapOutputTracker类图如下所示：



MapOutputTrackerMaster内部使用ShuffleStatus来维护各个Map任务的输出标志，其中ShuffleStatus中使用*mapStatus:Array[MapStatus](numPartitions)，*mapStatus维护了mapId和map输出的Block地址BlockManagerId，所以reduce任务知道如何获取map任务中的中间输出。其中MapOutputTrackerMaster种的ShuffleStatus还使用cachedSerializedMapStatus：Array[Byte]维护序列化后的任务的输出字节数组。如上面代码，Driver和Executor处理的MapOutputTracker的方式不同：

1. 如果应用程序是Driver，则创建MapOutputTrackerMaster，然后创建MapOutputTracker-

MasterEndpoint，并且注册到RpcEnv中。

1. Executor，则创建MapOutputTrackerWorker，并从RpcEnv中获取MapOutputTrackerMaster-

Endpoint

无论是Driver还是Executor，最后都有MapOutputTracker的属性trackerEndpoint的引用，代码如下：

*def registerOrLookupEndpoint(*

*name: String, endpointCreator: => RpcEndpoint):*

*RpcEndpointRef = {*

*if (isDriver) {*

*logInfo("Registering " + name)*

*rpcEnv.setupEndpoint(name, endpointCreator)*

*} else {*

*RpcUtils.makeDriverRef(name, conf, rpcEnv)*

*}}*

Map任务执行的状态由Executor持有的MapOutputTrackerMasterEndpoint发送消息，将其同步到mapoutputTracker的shuffleStatus和cacheSerializedShuffleStatus中。

1. 在Driver中，SparkEnv将trackerEndpoint注册到RpcEnv中：

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

*new MapOutputTrackerMasterEndpoint(*

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

1. 在Executor中，MapOutputTrackerWorker，通过初始化，通过SparkEnv获取Endpoint，如下所示

*env.mapOutputTracker.asInstanceOf[MapOutputTrackerWorker].updateEpoch(task.epoch)*

1. ShuffleManager

Shuffle是MapReduce框架中的特定阶段，Map输出的结果被Reduce使用时，需要通过Shuffle过程将Map结果分发到Reducer上去。ShuffleManager负责管理本地及远程的block数据的shuffle操作，包括shuffle过程的执行、计算和处理。在Spark 1.2之前Shuffle的计算引擎是HashShuffleManager，但是会产生大量的中间磁盘文件，影响性能。因此Spark 1.2之后使用的ShuffleManager改成了SortShuffleManager，类图如下所示：



IndexShuffleBlockResolver间接操作BlockManager将map结果写入本地，并根据shuffleId、mapId写入索引文件，同时通过MapOutputTrackerMaster中维护的mapStatus从本地或者其他远程节点读取文件。ShuffleManager的实现如下：

*val shortShuffleMgrNames = Map(*

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

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

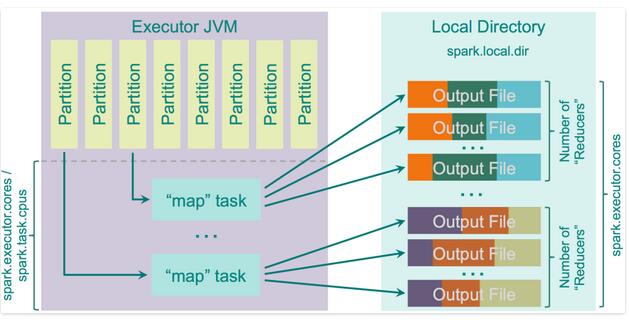
*val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")*

*val shuffleMgrClass =*

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

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

SortShuffle的过程如下所示：



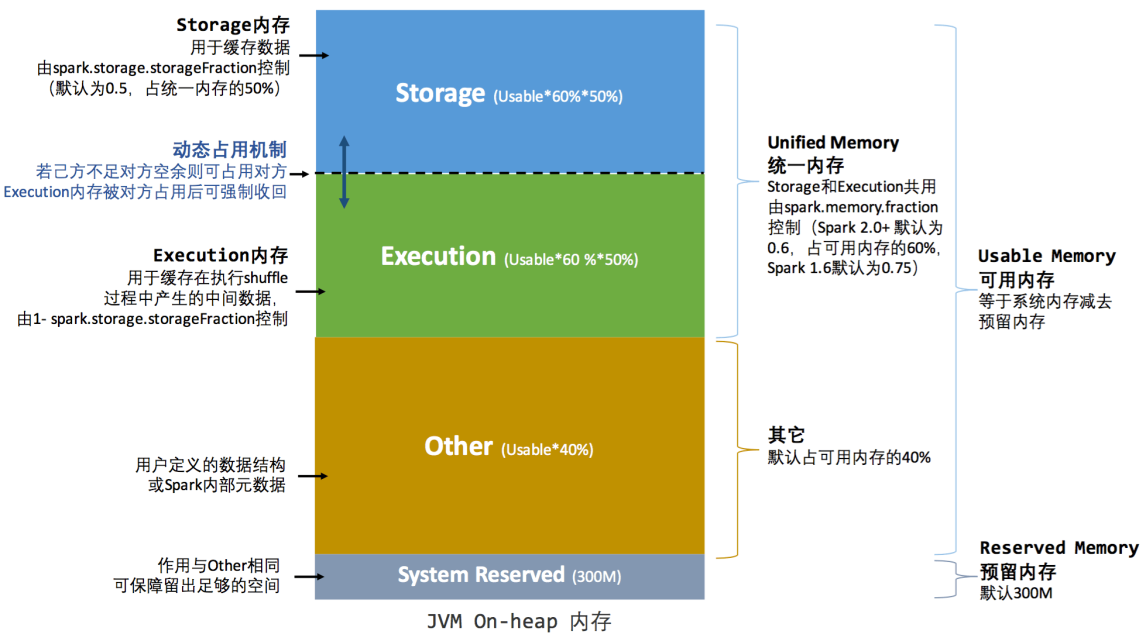
Spark在Sort Shuffle的reduce之前提前用算法进行排序。

1. 创建内存管理MemoryManager

Spark内存管理使用的接口是MemoryManager，在Spark1.6之后，引入新的内存管理模型，类图如下所示：



MemoryManager的实现类为UnifiedMemoryManager，实现的内存管理模式如下所示：



主要分成三个主要内存区域：

1. Reserved Memory，系统预留内存，在Spark 1.6中大小是300MB
2. Spark Memory，由Spark管理的内存，配置参数：spark.memory.fraction，内存大小为：

*(JAVA Heap – Reserved Memory)\*spark.memory.fraction*

该内存分成两个部分：Storage Memory和Execution Memory，比例关系：

*spark.memory.storageFraction*配置。StorageMemory缓存Spark数据，也用来做unroll序列化数据的临时空间，广播变量以block的形式存储在这里。Execution Memory，存储执行task过程中的一些对象，也可以用来shuffle Map端的中间缓存。

1. User Memory，用来存储用户数据，如RDD transformations过程使用的数据结构。

Spark 1.6之后默认为统一管理（Unified Memory Manager），之前采用的静态管理Static Memory Manager仍为保留，可以通过配置spark.memory.useLegacyMode参数启用，如下所示：

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

*val memoryManager: MemoryManager =*

*if (useLegacyMemoryManager) {*

*new StaticMemoryManager(conf, numUsableCores)*

*} else {*

*UnifiedMemoryManager(conf, numUsableCores)*

*}*

1. 块传输服务BlockTransferService

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服务及客户端，获取远程节点上的Block集合：

*val blockTransferService =*

*new NettyBlockTransferService(conf, securityManager, bindAddress, advertiseAddress,*

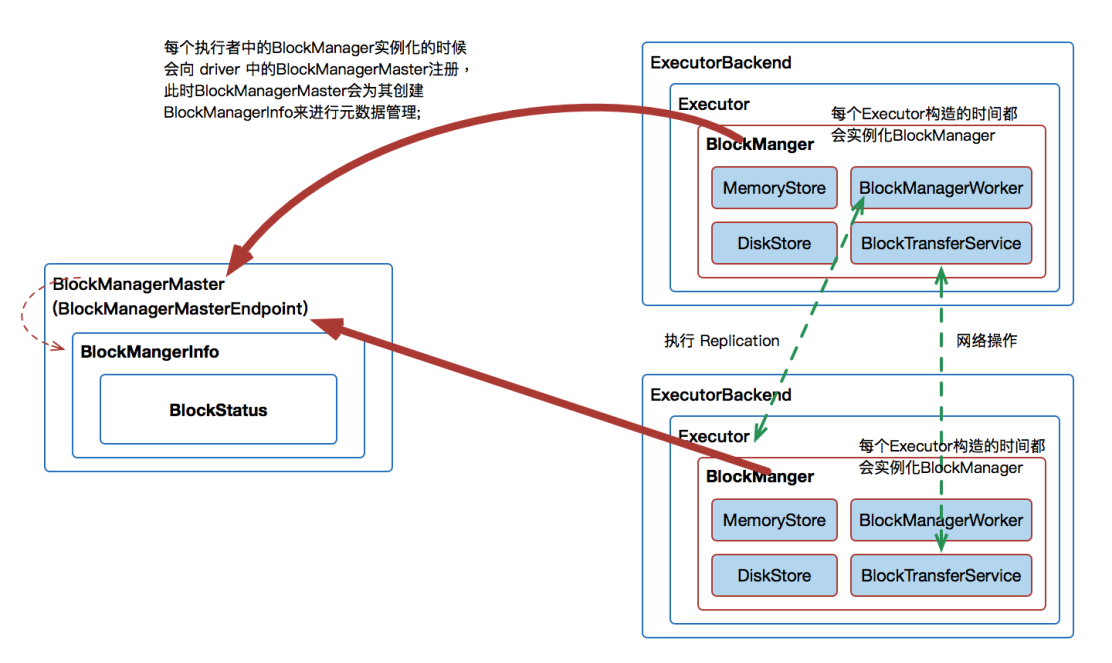
*blockManagerPort, numUsableCores)*

类图如下所示：



1. BlockManagerMaster及BlockManager

BlockManager负责对Block管理，作为存储系统的一部分，BlockManagerMaster负责对Block管理和协调，具体操作依赖于BlockManagerMasterEndpoint，Driver和Executor处理BlockManagerMaster的方式不同，如下所示：



1. 在Driver端，创建BlockManagerMaster，并且注册到RpcEnv中，源码如下所示：

*val blockManagerMaster = new BlockManagerMaster(registerOrLookupEndpoint(*

*BlockManagerMaster.DRIVER\_ENDPOINT\_NAME,*

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

*conf, isDriver)*

1. 如果当前应用程序是Executor，则从RpcEnv中找到BlockManagerMasterEndpoint中

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

*serializerManager, conf, memoryManager, mapOutputTracker, shuffleManager,*

*blockTransferService, securityManager, numUsableCores)*

无论是Driver还是Executor，最后BlockManagerMaster属性driverEndpoint将持有对BlockManagerMasterEndpoint的引用。

应用程序启动后，会将BlockManagerMaster注册到Driver中，启动ExecutorBackend时会实例化BlockManager并通过远程通信的方式注册给BlockManagerMaster，实际上是Executor中的BlockManager注册给Driver中的BlockManagerMasterEndpoint，代码如下所示：

*private val slaveEndpoint = rpcEnv.setupEndpoint(*

*"BlockManagerEndpoint" + BlockManager.ID\_GENERATOR.next,*

*new BlockManagerSlaveEndpoint(rpcEnv, this, mapOutputTracker))*

*val idFromMaster = master.registerBlockManager(*

*id,*

*maxOnHeapMemory,*

*maxOffHeapMemory,*

*slaveEndpoint)*

BlockManager相关的类图如下所示：



在BlockManager中：

1. MemoryStore是BlockManager中专门负责内存数据存储和读写的类
2. DiskStore是BlockManager中专门负责磁盘数据存储和读写的类
3. DiskBlockManager，管理LogicalBlock与Disk上的PhysicalBlock之间的映射关系并负责磁盘的文件的创建、读写等。
4. 创建测量系统MetricsSystem

MetricsSystem是Spark的测量系统，创建MetricsSystem的代码如下所示：

*val metricsSystem = if (isDriver) {*

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

*} else {*

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

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

*ms.start()*

*ms*

*}*

上面使用createMetricsSystem方法实际上创建的MetricsSystem，代码如下：

*def createMetricsSystem(*

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

*new MetricsSystem(instance, conf, securityMgr)*

*}*

构造MetricsSystem的过程最重要的是调用MetricsConfig的initialize方法，代码如下：

*def initialize() {*

*setDefaultProperties(properties)*

*loadPropertiesFromFile(conf.getOption("spark.metrics.conf"))*

*val prefix = "spark.metrics.conf."*

*conf.getAll.foreach {*

*case (k, v) if k.startsWith(prefix) =>*

*properties.setProperty(k.substring(prefix.length()), v)*

*case \_ =>*

*}*

*perInstanceSubProperties = subProperties(properties, INSTANCE\_REGEX)*

*if (perInstanceSubProperties.contains(DEFAULT\_PREFIX)) {*

*val defaultSubProperties = perInstanceSubProperties(DEFAULT\_PREFIX).asScala*

*for ((instance, prop) <- perInstanceSubProperties if (instance != DEFAULT\_PREFIX);*

*(k, v) <- defaultSubProperties if (prop.get(k) == null)) {*

*prop.put(k, v)*

*}}}*

从上面实现可以看出，MetricsConfig的initialize方法主要负责加载metrics.properties文件中的属性文件，并对属性进行初始化转换。

1. 创建SparkEnv

当所有的基础组件准备好后，使用下面的代码创建执行环境SparkEnv：

*val envInstance = new SparkEnv(*

*executorId,*

*rpcEnv,*

*serializer,*

*closureSerializer,*

*serializerManager,*

*mapOutputTracker,*

*shuffleManager,*

*broadcastManager,*

*blockManager,*

*securityManager,*

*metricsSystem,*

*memoryManager,*

*outputCommitCoordinator,*

*conf)*

# 创建RDD清理器，ContextCleaner

ContextCleaner用于清理那些超出应用范围的RDD、ShuffleDependency和Broadcast对象，在SparkContext中实例化如下：

*\_cleaner =*

*if (\_conf.getBoolean("spark.cleaner.referenceTracking", true)) {*

*Some(new ContextCleaner(this))*

*} else {*

*None*

*}*

*\_cleaner.foreach(\_.start())*

ContextCleaner类组成如下：

1. referenceQueue，缓存顶级AnyRef引用
2. referenceBuffer，缓存AnyRef的虚引用
3. listeners，缓存清理工作的监听器数组
4. cleaningThread，用于具体清理工作的线程

ContextCleaner的工作原理和listenerBus一样，也采用监听器模式，由线程来处理，此线程实际上只是调用keepCleaning方法，代码如下：

*private def keepCleaning(): Unit = Utils.tryOrStopSparkContext(sc) {*

*while (!stopped) {*

*try {*

*val reference = Option(referenceQueue.remove(ContextCleaner.REF\_QUEUE\_POLL\_TIMEOUT))*

*.map(\_.asInstanceOf[CleanupTaskWeakReference])*

*synchronized {*

*reference.foreach { ref =>*

*logDebug("Got cleaning task " + ref.task)*

*referenceBuffer.remove(ref)*

*ref.task match {*

*case CleanRDD(rddId) =>*

*doCleanupRDD(rddId, blocking = blockOnCleanupTasks)*

*case CleanShuffle(shuffleId) =>*

*doCleanupShuffle(shuffleId, blocking = blockOnShuffleCleanupTasks)*

*case CleanBroadcast(broadcastId) =>*

*doCleanupBroadcast(broadcastId, blocking = blockOnCleanupTasks)*

*case CleanAccum(accId) =>*

*doCleanupAccum(accId, blocking = blockOnCleanupTasks)*

*case CleanCheckpoint(rddId) =>*

*doCleanCheckpoint(rddId)*

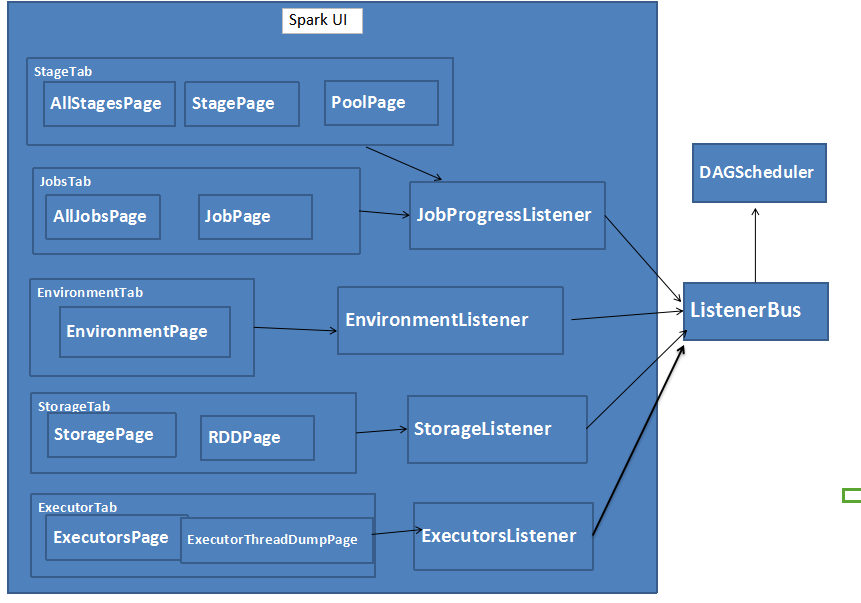
*}*

*}*

*}}*

# 创建并初始化Spark UI

SparkUI提供监控功能，用浏览器能访问具有样式及布局并提供丰富监控数据的页面。在SparkUI中采用事件监听机制，事件的处理是异步的，当前线程可以继续执行后续逻辑，线程池中的线程可以被重用，这样可以提供系统的并发度。发送的线程存入缓存，由定时调度器取出后，分配给监听此事件的监听器对监控数据进行更新，SparkUI架构如下所示：



上图中的各个组件：

1. DAGScheduler，主要产生各类SparkListenerEvent源头，将各种SparkListenerEvent发送给listenerBus的事件队列中
2. listenerBus通过定时器将SparkListenerEvent事件匹配到具体的SparkListener，改变SparkListener中的统计监听数据，最终由SparkUI的界面展示。

下图是SparkListener的实现：



1. listenerBus详解

listnerBus的类型是LiveListenerBus，其实现了监听器模型，通过监听事件触发对各种监听器监听状态信息的修改，达到UI界面的数据刷新消息。LiveListenerBus由以下部分组成：



1. 事件阻塞队列，类型为LinkedBlockingQueue[SparkListenerEvent]
2. 监听器数组，类型为CopyOnWriteArrayList[AsyncEventQueue]，存放各类监听器SparkListener。其中AsyncEventQueue对应事件类型的队列，其中有线程堆对该队列中的事件进行处理。
3. 事件匹配监听器的线程，此Thread不断拉取LinkedBlockingQueue中的事件，遍历监听器，调用监听器的方法，如下所示：

*private val dispatchThread = new Thread(s"spark-listener-group-$name") {*

*setDaemon(true)*

*override def run(): Unit = Utils.tryOrStopSparkContext(sc) {*

*dispatch()*

*}*

*}*

*private def dispatch(): Unit = LiveListenerBus.withinListenerThread.withValue(true) {*

*try {*

*var next: SparkListenerEvent = eventQueue.take()*

*while (next != POISON\_PILL) {*

*val ctx = processingTime.time()*

*try {*

*super.postToAll(next)*

*} finally {*

*ctx.stop()*

*}*

*eventCount.decrementAndGet()*

*next = eventQueue.take()*

*}*

*eventCount.decrementAndGet()*

*} catch {*

*case ie: InterruptedException =>*

*logInfo(s"Stopping listener queue $name.", ie)*

*}*

*}*

LiveListenerBus中调用addToQueue将SparkListener添加到Queue中，代码如下：

*private def addToQueue(listener: SparkListenerInterface, queue: String): Unit = synchronized {*

*if (stopped.get()) {*

*throw new IllegalStateException("LiveListenerBus is stopped.")*

*}*

*queues.asScala.find(\_.name == queue) match {*

*case Some(queue) =>*

*queue.addListener(listener)*

*case None =>*

*val newQueue = new AsyncEventQueue(queue, conf, metrics)*

*newQueue.addListener(listener)*

*if (started.get()) {*

*newQueue.start(sparkContext)*

*}*

*queues.add(newQueue)*

*}*

*}*

AsyncEventQueue启动后，调用postToAll(event)定义在父类SparkListenerBus中，代码如下：

*protected override def doPostEvent(*

*listener: SparkListenerInterface,*

*event: SparkListenerEvent): Unit = {*

*event match {*

*case stageSubmitted: SparkListenerStageSubmitted =>*

*listener.onStageSubmitted(stageSubmitted)*

*case stageCompleted: SparkListenerStageCompleted =>*

*listener.onStageCompleted(stageCompleted)*

*case jobStart: SparkListenerJobStart =>*

*listener.onJobStart(jobStart)*

*case jobEnd: SparkListenerJobEnd =>*

*listener.onJobEnd(jobEnd)*

*case taskStart: SparkListenerTaskStart =>*

*listener.onTaskStart(taskStart)*

*case taskGettingResult: SparkListenerTaskGettingResult =>*

*listener.onTaskGettingResult(taskGettingResult)*

*case taskEnd: SparkListenerTaskEnd =>*

*listener.onTaskEnd(taskEnd)*

*case environmentUpdate: SparkListenerEnvironmentUpdate =>*

*listener.onEnvironmentUpdate(environmentUpdate)*

*case blockManagerAdded: SparkListenerBlockManagerAdded =>*

*listener.onBlockManagerAdded(blockManagerAdded)*

*case blockManagerRemoved: SparkListenerBlockManagerRemoved =>*

*listener.onBlockManagerRemoved(blockManagerRemoved)*

*case unpersistRDD: SparkListenerUnpersistRDD =>*

*listener.onUnpersistRDD(unpersistRDD)*

*case applicationStart: SparkListenerApplicationStart =>*

*listener.onApplicationStart(applicationStart)*

*case applicationEnd: SparkListenerApplicationEnd =>*

*listener.onApplicationEnd(applicationEnd)*

*case metricsUpdate: SparkListenerExecutorMetricsUpdate =>*

*listener.onExecutorMetricsUpdate(metricsUpdate)*

*case executorAdded: SparkListenerExecutorAdded =>*

*listener.onExecutorAdded(executorAdded)*

*case executorRemoved: SparkListenerExecutorRemoved =>*

*listener.onExecutorRemoved(executorRemoved)*

*case executorBlacklisted: SparkListenerExecutorBlacklisted =>*

*listener.onExecutorBlacklisted(executorBlacklisted)*

*case executorUnblacklisted: SparkListenerExecutorUnblacklisted =>*

*listener.onExecutorUnblacklisted(executorUnblacklisted)*

*case nodeBlacklisted: SparkListenerNodeBlacklisted =>*

*listener.onNodeBlacklisted(nodeBlacklisted)*

*case nodeUnblacklisted: SparkListenerNodeUnblacklisted =>*

*listener.onNodeUnblacklisted(nodeUnblacklisted)*

*case blockUpdated: SparkListenerBlockUpdated =>*

*listener.onBlockUpdated(blockUpdated)*

*case speculativeTaskSubmitted: SparkListenerSpeculativeTaskSubmitted =>*

*listener.onSpeculativeTaskSubmitted(speculativeTaskSubmitted)*

*case \_ => listener.onOtherEvent(event)*

*}*

*}*

1. JobProgressListener

以JobProgressListener为例讲解SparkListener，通过监听ListenerBus中的事件更新任务进度，SparkStatusTracker和Spark UI实际上是通过JobProgressListener来实现任务状态跟踪的，创建代码如下:

*\_jobProgressListener = new JobProgressListener(\_conf)*

*listenerBus.addToStatusQueue(jobProgressListener)*

JobProgressListner的作用是通过HashMap、ListBuffer等数据结构存储JobId及对应的JobUIData信息，并按照激活、完成、失败等Job状态统计。

其实现了onJobStart,onJobEnd,onStageCompleted,onTaskStart,onTaskEnd等方法，这些方法在listenerBus的驱动下，改变JobProgressListener中的各种Job、Stage相关的数据。

3）SparkUI的创建和初始化

Spark UI的创建，代码清单如下：

*\_ui =*

*if (conf.getBoolean("spark.ui.enabled", true)) {*

*Some(SparkUI.createLiveUI(this, \_conf, \_jobProgressListener,*

*\_env.securityManager, appName, startTime = startTime))*

*} else {*

*None*

*}*

*\_ui.foreach(\_.bind())*

如果不需要Spark UI服务，可以将属性spark.ui.enabled修改为false,其中createLiveUI实际上是调用create方法，代码如下：

*private def create(*

*sc: Option[SparkContext],*

*conf: SparkConf,*

*addListenerFn: SparkListenerInterface => Unit,*

*securityManager: SecurityManager,*

*appName: String,*

*basePath: String = "",*

*jobProgressListener: Option[JobProgressListener] = None,*

*lastUpdateTime: Option[Long] = None,*

*startTime: Long): SparkUI = {*

*val \_jobProgressListener: JobProgressListener = jobProgressListener.getOrElse {*

*val listener = new JobProgressListener(conf)*

*addListenerFn(listener)*

*listener*

*}*

*val environmentListener = new EnvironmentListener*

*val storageStatusListener = new StorageStatusListener(conf)*

*val executorsListener = new ExecutorsListener(storageStatusListener, conf)*

*val storageListener = new StorageListener(storageStatusListener)*

*val operationGraphListener = new RDDOperationGraphListener(conf)*

*addListenerFn(environmentListener)*

*addListenerFn(storageStatusListener)*

*addListenerFn(executorsListener)*

*addListenerFn(storageListener)*

*addListenerFn(operationGraphListener)*

*new SparkUI(sc, conf, securityManager, environmentListener, storageStatusListener,*

*executorsListener, \_jobProgressListener, storageListener, operationGraphListener,*

*appName, basePath, lastUpdateTime, startTime)*

*}*

在代码中可以看到，除了JobProgressListener是外部传入之外，又增加了一些SparkListener：

1. EnvironmentListener，用于对JVM参数，Spark属性、Java系统属性、classpath等进行监控
2. StorageStatusListener，维护Executor的存储状态
3. ExecutorsListener，将Executor的信息展示在ExecutorsTab
4. StorageListener，将Executor相关存储信息展示在BlockManagerUI

在initialize方法中会组织前端页面的各个Tab和Page的展示和布局，代码如下所示：

*def initialize() {*

*val jobsTab = new JobsTab(this)*

*attachTab(jobsTab)*

*val stagesTab = new StagesTab(this)*

*attachTab(stagesTab)*

*attachTab(new StorageTab(this))*

*attachTab(new EnvironmentTab(this))*

*attachTab(new ExecutorsTab(this))*

*attachHandler(createStaticHandler(SparkUI.STATIC\_RESOURCE\_DIR, "/static"))*

*attachHandler(createRedirectHandler("/", "/jobs/", basePath = basePath))*

*attachHandler(ApiRootResource.getServletHandler(this))*

*// These should be POST only, but, the YARN AM proxy won't proxy POSTs*

*attachHandler(createRedirectHandler(*

*"/jobs/job/kill", "/jobs/", jobsTab.handleKillRequest, httpMethods = Set("GET", "POST")))*

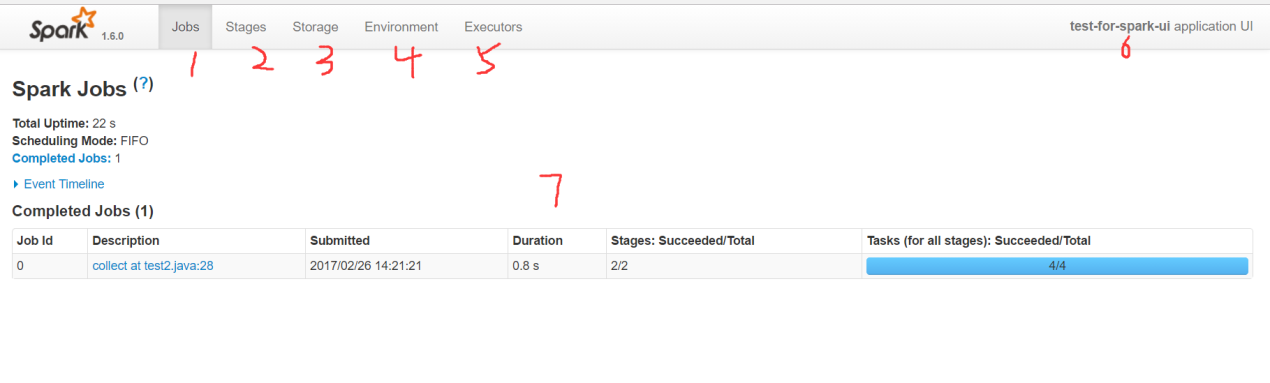
*attachHandler(createRedirectHandler(*

*"/stages/stage/kill", "/stages/", stagesTab.handleKillRequest,*

*httpMethods = Set("GET", "POST")))*

*}*

Spark UI的主页面对应的Listener如下所示：



1. Spark UI的页面布局与展示

以JobTabs为例，展示所有Job的进度、状态信息，JobTabs复用Spark UI的killEnabled、SparkContext及JobProgressListener，包括AllJobsPage和JobPage两个页面，如下所示：

*val killEnabled = parent.killEnabled*

*val jobProgresslistener = parent.jobProgressListener*

*val executorListener = parent.executorsListener*

*val operationGraphListener = parent.operationGraphListener*

*def isFairScheduler: Boolean =*

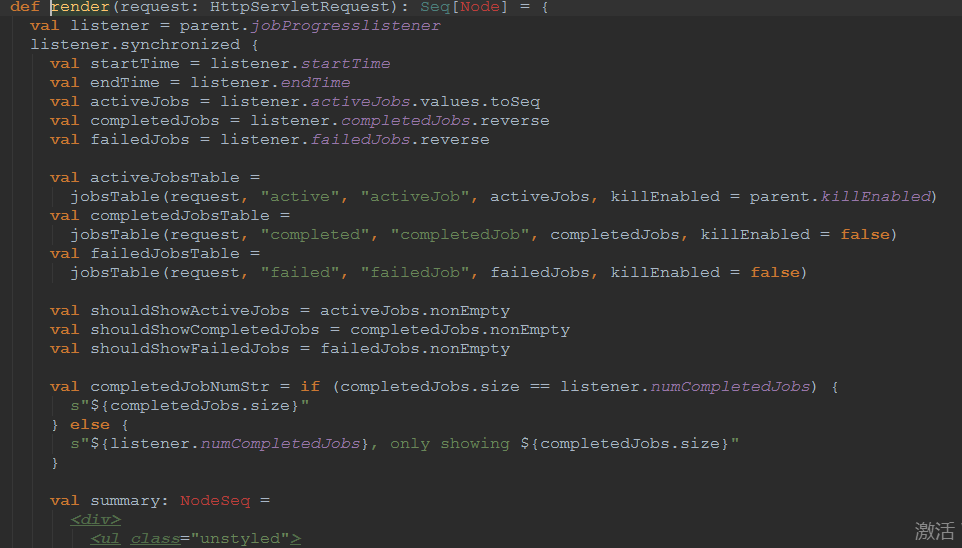
*jobProgresslistener.schedulingMode == Some(SchedulingMode.FAIR)*

*def getSparkUser: String = parent.getSparkUser*

*attachPage(new AllJobsPage(this))*

*attachPage(new JobPage(this))*

AllJobsPage由render来渲染，利用JobProgressListener中的统计监控数据生成激活、完成、失败等状态的Job摘要信息，并调用jobsTable方法生成表格等html元素，最终使用UIUtils的headerSparkPage封装好css、js、header及页面布局等，代码清单如下：



SparkUI创建好后，需要调用父类WebUI的bind方法，绑定服务和端口，bind方法中主要的代码如下：

*serverInfo = Some(startJettyServer(host, port, sslOptions, handlers, conf, name))*

# Hadoop相关配置及Executor环境变量的设置

1. Hadoop相关配置信息

Spark默认使用HDFS作为分布式文件系统，所以需要获取Hadoop相关配置信息，代码如下：

*\_hadoopConfiguration = SparkHadoopUtil.get.newConfiguration(\_conf)*

获取的配置信息包括：

*if (conf != null) {*

*//将Amazon S3文件系统的的accessid和key加载到hadoop conf*

*val keyId = System.getenv("AWS\_ACCESS\_KEY\_ID")*

*val accessKey = System.getenv("AWS\_SECRET\_ACCESS\_KEY")*

*if (keyId != null && accessKey != null) {*

*hadoopConf.set("fs.s3.awsAccessKeyId", keyId)*

*hadoopConf.set("fs.s3n.awsAccessKeyId", keyId)*

*hadoopConf.set("fs.s3a.access.key", keyId)*

*hadoopConf.set("fs.s3.awsSecretAccessKey", accessKey)*

*hadoopConf.set("fs.s3n.awsSecretAccessKey", accessKey)*

*hadoopConf.set("fs.s3a.secret.key", accessKey)*

*val sessionToken = System.getenv("AWS\_SESSION\_TOKEN")*

*if (sessionToken != null) {*

*hadoopConf.set("fs.s3a.session.token", sessionToken)*

*}}*

*//将spark.hadoop开头的属性复制到hadoop conf*

*appendSparkHadoopConfigs(conf, hadoopConf)*

*//将spark.buffer.size复制到为hadoop conf的io.file.buffer.size*

*val bufferSize = conf.get("spark.buffer.size", "65536")*

*hadoopConf.set("io.file.buffer.size", bufferSize)*

*}*

如果指定了SPARK\_YARN\_MODE属性，则会使用YarnSparkHadoopUtil，默认使用SparkHadoopUtil。

1. Executor环境变量

对Executor的环境变量的处理，下面代码所示：

*\_executorMemory = \_conf.getOption("spark.executor.memory")*

*.orElse(Option(System.getenv("SPARK\_EXECUTOR\_MEMORY")))*

*.orElse(Option(System.getenv("SPARK\_MEM"))*

*.map(warnSparkMem))*

*.map(Utils.memoryStringToMb)*

*.getOrElse(1024)*

*for { (envKey, propKey) <- Seq(("SPARK\_TESTING", "spark.testing"))*

*value <- Option(System.getenv(envKey)).orElse(Option(System.getProperty(propKey)))} {*

*executorEnvs(envKey) = value*

*}*

*Option(System.getenv("SPARK\_PREPEND\_CLASSES")).foreach { v =>*

*executorEnvs("SPARK\_PREPEND\_CLASSES") = v*

*}*

*executorEnvs("SPARK\_EXECUTOR\_MEMORY") = executorMemory + "m"*

*executorEnvs ++= \_conf.getExecutorEnv*

*executorEnvs("SPARK\_USER") = sparkUser*

executorEnvs包含的环境变量将会在注册应用的过程中发送给Master， Master给Worker发送调度后，最终使用executorEnvs提供的信息启动Executor。

可以通过配置spark.executor.memory指定executor占用的内存大小，也可以配置系统变量SPARK\_EXECUTOR\_MEMORY或者SPARK\_MEM对其大小进行设置。

# 创建任务调度TaskScheduler

TaskScheduler是SparkContext的重要组成部分，负责任务的提交，并且请求集群管理器对任务进行调度。TaskScheduler可以看做为调度的客户端，创建的代码如下：

*val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)*

*\_schedulerBackend = sched*

*\_taskScheduler = ts*

在createTaskSheduler中根据master配置匹配的部署模式，创建TaskSchedulerImpl，并生成不同的schedulerBackend，匹配Yarn模式代码如下：

*case masterUrl =>*

*val cm = getClusterManager(masterUrl) match {*

*case Some(clusterMgr) => clusterMgr*

*case None => throw new SparkException("Could not parse Master URL: '" + master + "'")*

*}*

*try {*

*val scheduler = cm.createTaskScheduler(sc, masterUrl)*

*val backend = cm.createSchedulerBackend(sc, masterUrl, scheduler)*

*cm.initialize(scheduler, backend)*

*(backend, scheduler)*

*}*

1. 创建TaskSchedulerImpl

TaskSchedulerImpl的构造过程如下：

1. 从SparkConf中读取配置信息，包括每个任务分配的cpu数、调度模式（FAIR/FIFO，默认是FIFO，可以通过修改属性spark.scheduler.mode来改变等）
2. 创建TaskResultGetter，通过线程池Executors.newFixedThreadPool创建，默认4个线程（以task-result-getter）开头，线程开头默认是Executors.defaultThreadFactory对worker上的Executor发送的Task的执行结果进行处理。

TaskSchedulerImpl的实现代码如下：

*var dagScheduler: DAGScheduler = null*

*var backend: SchedulerBackend = null*

*val mapOutputTracker = SparkEnv.get.mapOutputTracker.asInstanceOf[MapOutputTrackerMaster]*

*private var schedulableBuilder: SchedulableBuilder = null*

*// default scheduler is FIFO*

*private val schedulingModeConf = conf.get(SCHEDULER\_MODE\_PROPERTY, SchedulingMode.FIFO.toString)*

*val schedulingMode: SchedulingMode =*

*try {*

*SchedulingMode.withName(schedulingModeConf.toUpperCase(Locale.ROOT))*

*}.....*

*val rootPool: Pool = new Pool("", schedulingMode, 0, 0)*

*private[spark] var taskResultGetter = new TaskResultGetter(sc.env, this)*

TaskSchedulerImpl的调度模式由FAIR和FIFO两种，任务最终的调度都是落实到接口SchedulerBackend的具体实现上。在YARN 集群中，Spark SchedulerBackend的实现类为：

*override def createSchedulerBackend(sc: SparkContext,*

*masterURL: String,*

*scheduler: TaskScheduler): SchedulerBackend = {*

*sc.deployMode match {*

*case "cluster" =>*

*new YarnClusterSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)*

*case "client" =>*

*new YarnClientSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)*

*case \_ =>*

*throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")*

*}*

*}*

1. TaskSchedulerImpl的初始化

创建完TaskSchedulerImpl和SchedulerBackend后，对TaskSchedulerImpl的调用方法initialize进行初始化。以默认的FIFO调度为例，TaskSchedulerImpl的初始化过程如下：

*def initialize(backend: SchedulerBackend) {*

*this.backend = backend*

*schedulableBuilder = {*

*schedulingMode match {*

*case SchedulingMode.FIFO =>*

*new FIFOSchedulableBuilder(rootPool)*

*case SchedulingMode.FAIR =>*

*new FairSchedulableBuilder(rootPool, conf)*

*case \_ =>*

*throw new IllegalArgumentException(s"Unsupported $SCHEDULER\_MODE\_PROPERTY: " +*

*s"$schedulingMode")*

*}*

*}*

*schedulableBuilder.buildPools()*

*}*

1. 使TaskSchedulerImpl持有对backend的引用
2. 创建Pool，缓存调度队列、调度算法及TaskSetManager集合等信息
3. 创建FIFOSchedulableBuilder，用来操作Pool中的调度队列。

# 创建和启动DAGScheduler

DAGScheduler主要用于在任务正式提交给TaskSchedulerImpl提交之前做一些准备工作，包括创建Job,将DAG中的RDD划分到不同的Stage,提交Stage等。创建DAGScheduler的代码如下：

*\_dagScheduler = new DAGScheduler(this)*

DAGScheduler主要维护的数据结构为JobId与StageId的关系，Stage,ActiveJob以及缓存RDD的partitions的位置信息，代码如下：

*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 stageIdToStage = new HashMap[Int, Stage]*

*private[scheduler] val shuffleIdToMapStage = new HashMap[Int, ShuffleMapStage]*

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

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

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

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

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

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

*private val failedEpoch = new HashMap[String, Long]*

*private [scheduler] val outputCommitCoordinator = env.outputCommitCoordinator*

*private val closureSerializer = SparkEnv.get.closureSerializer.newInstance()*

在构造DAGScheduler的时候，会创建DAGSchedulerEventProcessLoop，该对象用于处理JobSubmitted,MapStageSubmitted,ExecutorAdded等消息，源码如下：

*private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {*

*case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>*

*dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)*

*case MapStageSubmitted(jobId, dependency, callSite, listener, properties) =>*

*dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite, listener, properties)*

*case StageCancelled(stageId, reason) =>*

*dagScheduler.handleStageCancellation(stageId, reason)*

*case JobCancelled(jobId, reason) =>*

*dagScheduler.handleJobCancellation(jobId, reason)*

*case JobGroupCancelled(groupId) =>*

*dagScheduler.handleJobGroupCancelled(groupId)*

*case AllJobsCancelled =>*

*dagScheduler.doCancelAllJobs()*

*case ExecutorAdded(execId, host) =>*

*dagScheduler.handleExecutorAdded(execId, host)*

*case ExecutorLost(execId, reason) =>*

*val workerLost = reason match {*

*case SlaveLost(\_, true) => true*

*case \_ => false*

*}*

*dagScheduler.handleExecutorLost(execId, workerLost)*

*case WorkerRemoved(workerId, host, message) =>*

*dagScheduler.handleWorkerRemoved(workerId, host, message)*

*case BeginEvent(task, taskInfo) =>*

*dagScheduler.handleBeginEvent(task, taskInfo)*

*case SpeculativeTaskSubmitted(task) =>*

*dagScheduler.handleSpeculativeTaskSubmitted(task)*

*case GettingResultEvent(taskInfo) =>*

*dagScheduler.handleGetTaskResult(taskInfo)*

*case completion: CompletionEvent =>*

*dagScheduler.handleTaskCompletion(completion)*

*case TaskSetFailed(taskSet, reason, exception) =>*

*dagScheduler.handleTaskSetFailed(taskSet, reason, exception)*

*case ResubmitFailedStages =>*

*dagScheduler.resubmitFailedStages()*

*}*

# TaskScheduler启动

TaskScheduler在创建后，进行启动：

*override def start() {*

*backend.start()*

*if (!isLocal && conf.getBoolean("spark.speculation", false)) {*

*logInfo("Starting speculative execution thread")*

*speculationScheduler.scheduleWithFixedDelay(new Runnable {*

*override def run(): Unit = Utils.tryOrStopSparkContext(sc) {*

*checkSpeculatableTasks()*

*}*

*}, SPECULATION\_INTERVAL\_MS, SPECULATION\_INTERVAL\_MS, TimeUnit.MILLISECONDS)*

*}*

*}*

实际上调用的是backend的start方法，以YarnSchedulerBackend为例，启动时会创建Endpoint，并注册到RpcEnv中，如下所示：

*private val yarnSchedulerEndpoint = new YarnSchedulerEndpoint(rpcEnv)*

*private val yarnSchedulerEndpointRef = rpcEnv.setupEndpoint(*

*YarnSchedulerBackend.ENDPOINT\_NAME, yarnSchedulerEndpoint)*

1. 创建CoarseGrainedSchedulerBackend

创建本地SchedulerBackend，核心在于启动CoarseGrainedSchedulerBackend，代码如下所示：

*private val reviveThread =*

*ThreadUtils.newDaemonSingleThreadScheduledExecutor("driver-revive-thread")*

*override def onStart() {*

*// Periodically revive offers to allow delay scheduling to work*

*val reviveIntervalMs = conf.getTimeAsMs("spark.scheduler.revive.interval", "1s")*

*reviveThread.scheduleAtFixedRate(new Runnable {*

*override def run(): Unit = Utils.tryLogNonFatalError {*

*Option(self).foreach(\_.send(ReviveOffers))*

*}*

*}, 0, reviveIntervalMs, TimeUnit.MILLISECONDS)*

*}*

*override def receive: PartialFunction[Any, Unit] = {*

*case StatusUpdate(executorId, taskId, state, data) =>*

*scheduler.statusUpdate(taskId, state, data.value)*

*if (TaskState.isFinished(state)) {*

*executorDataMap.get(executorId) match {*

*case Some(executorInfo) =>*

*executorInfo.freeCores += scheduler.CPUS\_PER\_TASK*

*makeOffers(executorId)*

*case None =>*

*// Ignoring the update since we don't know about the executor.*

*logWarning(s"Ignored task status update ($taskId state $state) " +*

*s"from unknown executor with ID $executorId")*

*}*

*}*

*case ReviveOffers =>*

*makeOffers()*

*case KillTask(taskId, executorId, interruptThread, reason) =>*

*executorDataMap.get(executorId) match {*

*case Some(executorInfo) =>*

*executorInfo.executorEndpoint.send(*

*KillTask(taskId, executorId, interruptThread, reason))*

*case None =>*

*// Ignoring the task kill since the executor is not registered.*

*logWarning(s"Attempted to kill task $taskId for unknown executor $executorId.")*

*}*

*case KillExecutorsOnHost(host) =>*

*scheduler.getExecutorsAliveOnHost(host).foreach { exec =>*

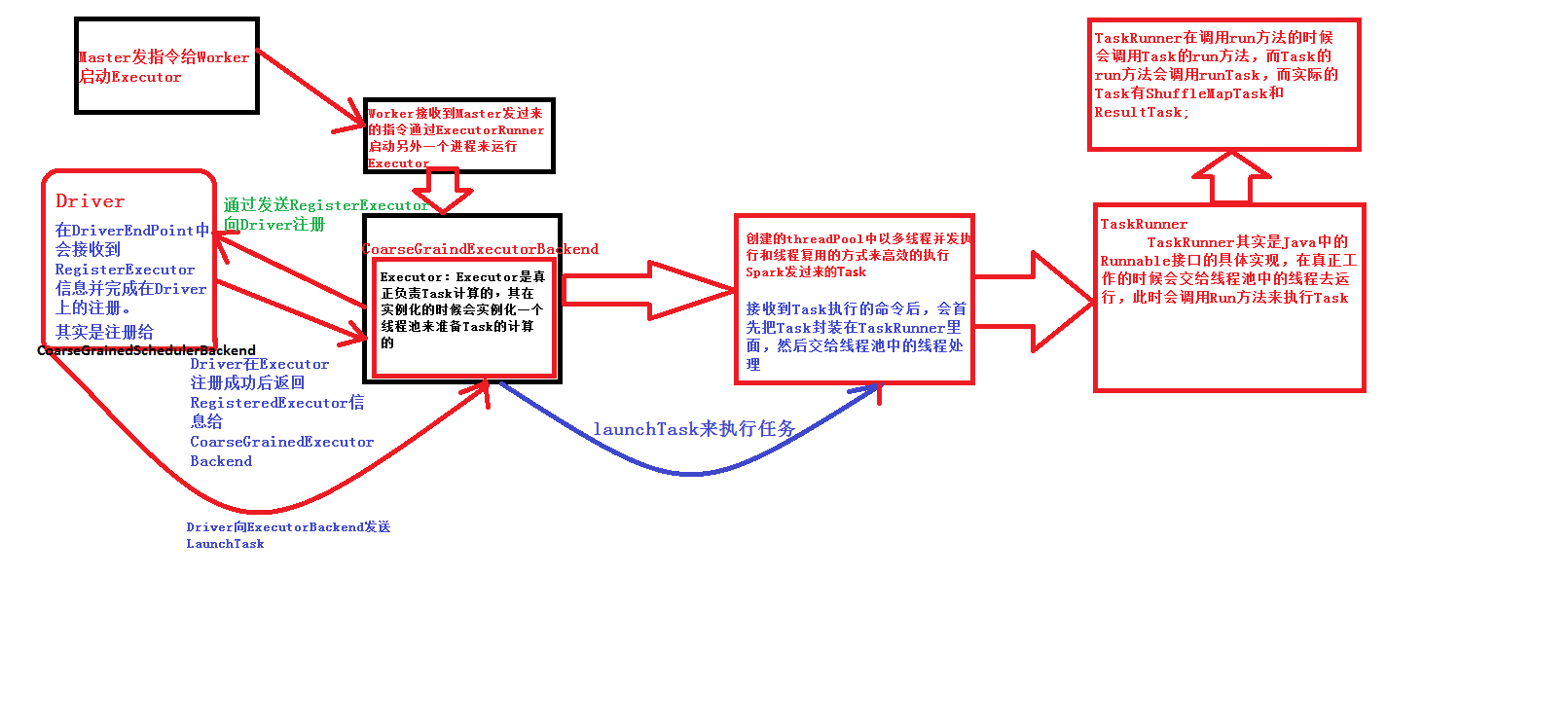
*killExecutors(exec.toSeq, replace = true, force = true)*

*}*

*}*

2）启动Executor及心跳

Executor负责执行Task，对应进程为CoarseGrainedExecutorBackend，启动流程如下所示：



1. CoarseGrainedExecutorBackend的启动过程如下：

*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 { () =>*

*Utils.checkHost(hostname)*

*val executorConf = new SparkConf*

*val fetcher = RpcEnv.create("driverPropsFetcher",hostname,-1,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)*

*}*

*cfg.hadoopDelegationCreds.foreach { hadoopCreds =>*

*val creds = SparkHadoopUtil.get.deserialize(hadoopCreds)*

*SparkHadoopUtil.get.addCurrentUserCredentials(creds)*

*}*

*val env = SparkEnv.createExecutorEnv(*

*driverConf, executorId, hostname, 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()*

*}*

1. CoarseGrainedExecutorBackend启动后，通过发送RegisterExecutor消息向Driver进行注册，下面的ref相当于Driver，如下所示：

*rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>*

*// This is a very fast action so we can use "ThreadUtils.sameThread"*

*driver = Some(ref)*

*ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extractLogUrls))*

1. Driver收到Executor的注册消息RegisterExecutor后，将Executor存入Driver的缓存中，如下：

*case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>*

*…….*

*val executorAddress = if (executorRef.address != null) {*

*executorRef.address*

*} else {*

*context.senderAddress*

*}*

*addressToExecutorId(executorAddress) = executorId*

*totalCoreCount.addAndGet(cores)*

*totalRegisteredExecutors.addAndGet(1)*

*val data = new ExecutorData(executorRef, executorRef.address, hostname,*

*cores, cores, logUrls)*

*CoarseGrainedSchedulerBackend.this.synchronized {*

*executorDataMap.put(executorId, data)*

*if (currentExecutorIdCounter < executorId.toInt) {*

*currentExecutorIdCounter = executorId.toInt*

*}*

*executorRef.send(RegisteredExecutor)*

*context.reply(true)*

*SparkListenerExecutorAdded(System.currentTimeMillis(), executorId, data))*

*makeOffers()*

*}*

在Driver中通过ExecutorData封装并注册ExecutorBackend信息到Driver的内存数据结构executorMapData中：

*private val executorDataMap = new HashMap[String, ExecutorData]*

1. CoarseGrainedExecutorBackend接收Driver发送的RegisterdExecutor后，启动executor:

*executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false*

Executor启动后，实例化一个线程池来准备Task的计算：

*private val threadPool = {*

*val threadFactory = new ThreadFactoryBuilder()*

*.setDaemon(true)*

*.setNameFormat("Executor task launch worker-%d")*

*.setThreadFactory(new ThreadFactory {*

*override def newThread(r: Runnable): Thread =*

*new UninterruptibleThread(r, "unused") // thread name will be set by ThreadFactoryBuilder*

*})*

*.build()*

*Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]*

*}*

*private val executorSource = new ExecutorSource(threadPool, executorId)*

创建的线程池threadPool以多线程并发执行和线程复用的方式高效执行Spark发送过来的Task。

1. 当Driver发送过来Task后，其实是发送给CoarseGrainedExecutorBackend这个RpcEndpoint，而不是发送给Executor

*case LaunchTask(data) =>*

*if (executor == null) {*

*exitExecutor(1, "Received LaunchTask command but executor was null")*

*} else {*

*val taskDesc = TaskDescription.decode(data.value)*

*executor.launchTask(this, taskDesc)*

*}*

1. Executor执行launchTask来执行任务，

*def launchTask(context: ExecutorBackend, taskDescription: TaskDescription): Unit = {*

*val tr = new TaskRunner(context, taskDescription)*

*runningTasks.put(taskDescription.taskId, tr)*

*threadPool.execute(tr)*

*}*

http://blog.csdn.net/u013063153/article/details/54909115

# 启动测量系统MetricsSystem

MetricsSystem使用codahale提供的第三方测量仓库Metrics，其相关概念如下：

1. Instance： 指定谁在使用测量系统， 区分为Master、Worker、Application、Driver和Executor
2. Source: 指定了从哪里收集测量数据
3. Sink： 指定了往哪里输出测量数据，目前提供的Sink有ConsoleSink,CsvSink,JmxSink，MetricsServlet和GraphiteSink等。Spark目前使用MetricsServlet作为默认的Sink

MetricsSystem的启动过程包括如下步骤：

1. 注册Sources
2. 注册Sinks
3. 给Sink增加Jetty的ServletContextHandler

启动代码如下：

*\_env.metricsSystem.start()*

*// Attach the driver metrics servlet handler to the web ui after the metrics system is started.*

*\_env.metricsSystem.getServletHandlers.foreach(handler => ui.foreach(\_.attachHandler(handler)))*

MetricsSystem启动完毕后，会遍历与Sinks有关的ServletContextHandler，并调用attachHandler将其绑定到SparkUI上。MetricsSystem的启动如下：

*def start() {*

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

*running = true*

*StaticSources.allSources.foreach(registerSource)*

*registerSources()*

*registerSinks()*

*sinks.foreach(\_.start)*

*}*

1. 注册Source，registerSource方法用于注册Sources，告诉测量系统从哪里收集测量数据，代码如下：

*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)*

*}*

*}*

*}*

1. 从metricsConfig获取Driver的properties，默认为创建MetricsSystem的过程中解析：

*prop.setProperty("\*.sink.servlet.class", "org.apache.spark.metrics.sink.MetricsServlet")*

1. 用正则匹配Driver的properties中以source.开头的属性，然后将属性中的Source反射得到的实例加入ArrayBuffer[Source]
2. 将每个source的metricRegistry注册到ConcurrentMap<String,Metric> metrics
3. 注册Sinks

registerSinks方法用于注册Sinks，即告诉测量系统MetricsSystem往哪里输出测量数据，实现代码如下：

*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*

*}}}}}*

1. 从Driver的properties中用正则匹配sink.开头的属性，如*{"\*.sink.servlet.class", "org.apache.spark.metrics.sink.MetricsServlet"}*
2. 将子属性class对应的类MetricsServlet反射得到MetricsServlet实例，如果属性为servlet，将其设置为metricsSystem；如果是Sink，则加入到ArrayBuffer[Sink]中
3. 给Sinks增加Jetty的ServletContextHandler

为了能够在SparkUI访问到测量数据，所以需要给Sinks增加Jetty的ServletContextHandler，这里用到的MetricsSystem的getServletHandlers方法实现如下：

*def getServletHandlers: Array[ServletContextHandler] = {*

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

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

*}*

调用了metricsServlet的getHandlers，其实现如下：

*def getHandlers(conf: SparkConf): Array[ServletContextHandler] = {*

*Array[ServletContextHandler](*

*createServletHandler(servletPath,*

*new ServletParams(request => getMetricsSnapshot(request), "text/json"), securityMgr, conf)*

*)}*

最终生成处理的/metrics/json请求的ServletContextHandler，而请求的真正处理由getMetricsSnapshot方法，利用fastjson解析，生成ServletContextHandler通过SparkUI的attachHandler方法，被绑定到SparkUI。

# 创建和启动Executor分配管理器ExecutorAllocationManager

ExecutorAllocationManager用于对已分配的Executor进行管理，创建和启动ExecutorAllocationManager的代码如下：

*val dynamicAllocationEnabled = Utils.isDynamicAllocationEnabled(\_conf)*

*\_executorAllocationManager =*

*if (dynamicAllocationEnabled) {*

*schedulerBackend match {*

*case b: ExecutorAllocationClient =>*

*Some(new ExecutorAllocationManager(*

*schedulerBackend.asInstanceOf[ExecutorAllocationClient], listenerBus, \_conf))*

*case \_ =>*

*None*

*}*

*} else {*

*None*

*}*

*\_executorAllocationManager.foreach(\_.start())*

默认不会创建ExecutionAllocationManager，通过修改属性：*spark.dynamicAllocations.enable*来创建，可以设置动态分配最小或者最大的Executor数量，每个Executor可以运行的Task数量等配置信息，并对配置信息进行校验。

在start方法中将ExecutorAllocationListener加入listenerBus中，ExecutorAllocationListener通过监听listenerBus里的事件，动态添加、删除Executor。并且通过Thread不断添加Executor，遍历Executor，将超时的Executor杀掉并移除，其实现与其他SparkListener类似，关键代码如下：

*def start(): Unit = {*

*listenerBus.addToManagementQueue(listener)*

*val scheduleTask = new Runnable() {*

*override def run(): Unit = {*

*try {*

*schedule()*

*} catch {*

*case ct: ControlThrowable =>*

*throw ct*

*case t: Throwable =>*

*logWarning(s"Uncaught exception in thread ${Thread.currentThread().getName}", t)*

*}*

*}*

*}*

*executor.scheduleWithFixedDelay(scheduleTask, 0, intervalMillis, TimeUnit.MILLISECONDS)*

*client.requestTotalExecutors(numExecutorsTarget, localityAwareTasks, hostToLocalTaskCount)*

*}*

# 将SparkContext标记为激活

SparkContext初始化的最后将当前SparkContext的状态从contextBeingConstruncted（正在构建中）改为activeContext（已激活），代码如下：

*private[spark] def setActiveContext(*

*sc: SparkContext,*

*allowMultipleContexts: Boolean): Unit = {*

*SPARK\_CONTEXT\_CONSTRUCTOR\_LOCK.synchronized {*

*assertNoOtherContextIsRunning(sc, allowMultipleContexts)*

*contextBeingConstructed = None*

*activeContext.set(sc)*

*}*

*}*

Spark学习：http://blog.csdn.net/qq\_21383435/article/details/78645708

https://0x0fff.com/spark-memory-management/

https://www.cnblogs.com/yangsy0915/p/6058611.html

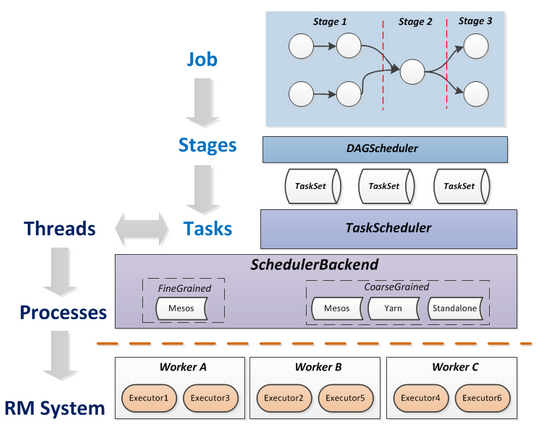
Shuffle内存分析：

https://github.com/JerryLead/SparkInternals/blob/master/markdown/4-shuffleDetails.md

http://blog.csdn.net/dabokele/article/details/51475469 https://www.ibm.com/developerworks/cn/analytics/library/ba-cn-apache-spark-memory-management/index.html?ca=drs-&utm\_source=tuicool&utm\_medium=referral

https://www.cnblogs.com/jcchoiling/p/6507162.html

http://www.infoq.com/cn/news/2017/08/development-practice-reference



http://www.mamicode.com/info-detail-530178.html