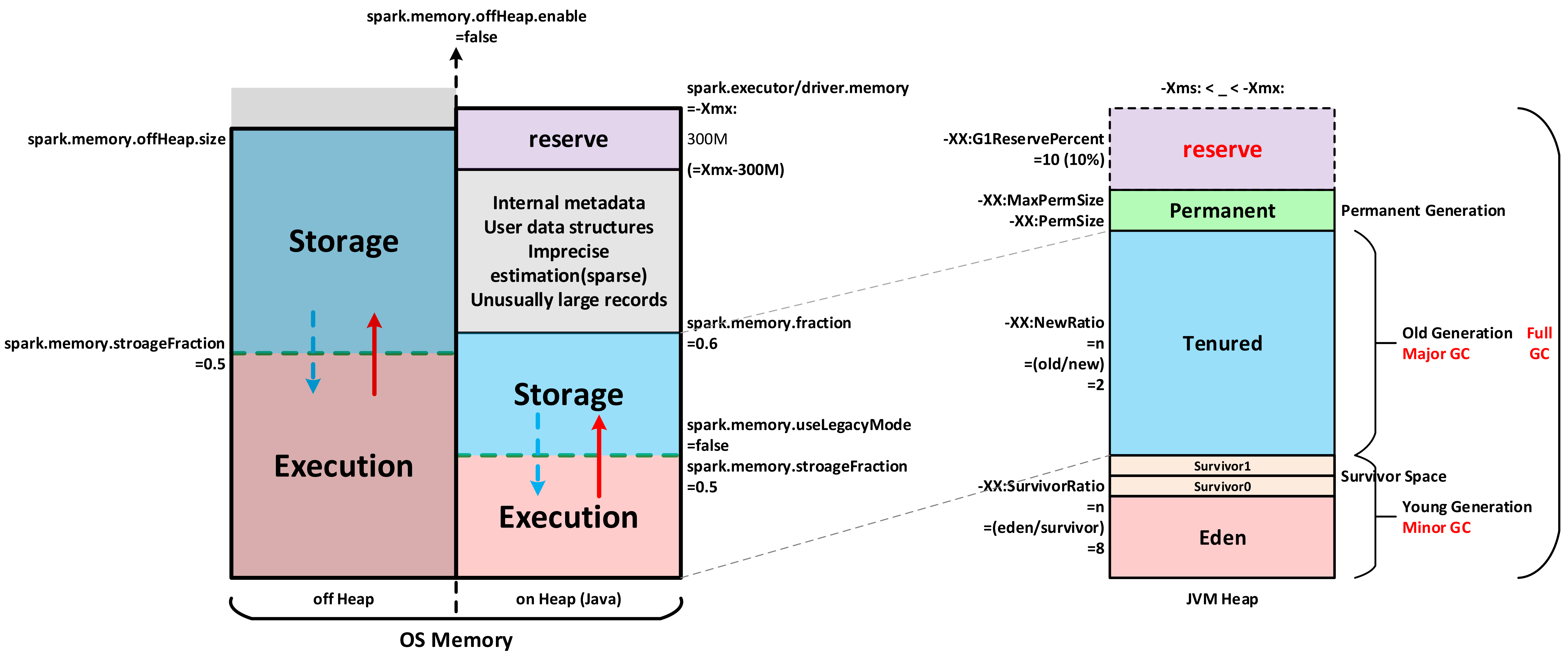


Problem: conflict between Execution and Tenured

- Too many GCs, many full GCs
  not enough memory for executing tasks.
- Too many minor GCs, but not many major GCs
  allocating more memory for Eden.
- OldGen is close to begin full
  reduce spark.memory.fraction
  decrease YoungGen
  large -XX:NewRatio
-
-

org.apache.spark.scheduler.LiveListenerBus

```
eventQueue = new LinkedBlockingQueue[SparkListenerEvent](EVENT_QUEUE_CAPACITY)
```

阻塞定长的事件队列：塞满后，丢去新事件（新事件很可能与触发资源回收相关）

```
def postToAll(event: E): Unit = {
  // JavaConverters can create a JIterableWrapper if we use asScala.
  // However, this method will be called frequently. To avoid the wrapper cost, here we use
  // Java Iterator directly.
  val iter = listeners.iterator
  while (iter.hasNext) {
    val listener = iter.next()
    try {
      doPostEvent(listener, event)
    } catch {
      case NonFatal(e) =>
        logError(s"Listener ${Utils.getFormattedClassName(listener)} threw an exception", e)
    }
  }
}
```

单线程、串行处理分发时间到所有注册过的事件处理器：碰到慢速事件处理器（比如log）会耽误整个事件处理效率。

```
protected override def doPostEvent(
    listener: SparkListenerInterface,
    event: SparkListenerEvent): Unit = {
  event match {
    case stageSubmitted: SparkListenerStageSubmitted =>
      listener.onStageSubmitted(stageSubmitted)
    case stageCompleted: SparkListenerStageCompleted =>
      listener.onStageCompleted(stageCompleted)
```

进入某个时间处理器，还需逐一（串行）case所有类型进行对应处理：严重影响了事件处理的效率。

org.apache.spark.util.EventLoop

```scala
private val eventQueue: BlockingQueue[E] = new LinkedBlockingDeque[E]()
```

阻塞非定长事件队列：在DAGScheduler中当提交任务速度超过处理速度时，最终OOM(在非UDE使用的高并发使用中).

```scala
private val eventThread = new Thread(name) {
  setDaemon(true)

  override def run(): Unit = {
    try {
      while (!stopped.get) {
        val event = eventQueue.take()
        try {
          onReceive(event)
        } catch {
          case NonFatal(e) =>
            try {
              onError(e)
            } catch {
              case NonFatal(e) => logError("Unexpected error in " + name, e)
            }
        }
      }
    } catch {
      case ie: InterruptedException => // exit even if eventQueue is not empty
      case NonFatal(e) => logError("Unexpected error in " + name, e)
    }
  }
}
```

单线程、串行的事件处理逻辑，阻碍了事件处理的效率。

org.apache.spark.ContextCleaner

```scala
/** Start the cleaner. */
def start(): Unit = {
  cleaningThread.setDaemon(true)
  cleaningThread.setName("Spark Context Cleaner")
  cleaningThread.start()
  periodicGCService.scheduleAtFixedRate(new Runnable {
    override def run(): Unit = System.gc()
  }, periodicGCInterval, periodicGCInterval, TimeUnit.SECONDS)
}
```

通过System.gc()回收内存，如果存在：-XX:-DisableExplicitGC，则该功能废了，存在OOM风险。

```scala
private val cleaningThread = new Thread() { override def run() { keepCleaning() }}
```

单线程进行清理，无法应对高并发的情况。

```scala
def registerRDDForCleanup(rdd: RDD[_]): Unit = {
def registerShuffleForCleanup(shuffleDependency: ShuffleDependency[_, _, _]): Unit = {
def registerBroadcastForCleanup[T](broadcast: Broadcast[T]): Unit = {
/** Register a RDDCheckpointData for cleanup when it is garbage collected. */
def registerRDDCheckpointDataForCleanup[T](rdd: RDD[_], parentId: Int): Unit = {
  registerForCleanup(rdd, CleanCheckpoint(parentId))
}

/** Register an object for cleanup. */
private def registerForCleanup(objectForCleanup: AnyRef, task: CleanupTask): Unit = {
  referenceBuffer.add(new CleanupTaskWeakReference(task, objectForCleanup, referenceQueue))
}
```

RDD、 Shuffle、Broadcast、RDDCheckpoint通过注册，一备任务完成回收。（而“任务完成”这一事件通过分布式传递，LiveSparkListener处理（参考前文）

org.apache.spark.ContextCleaner

能否回收，回收效率，最终依托于JVM的对象引用机制。（Soft、Weak、Phantom、Final推到最后一波回收）

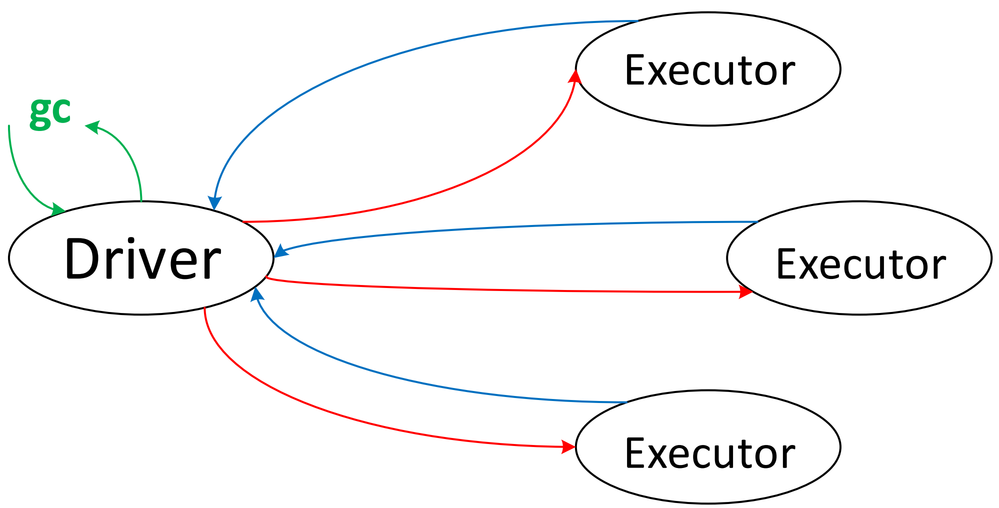
低效的删除操作。

```
/** Keep cleaning RDD, shuffle, and broadcast state. */
private def keepCleaning(): Unit = Utils.tryOrStopSparkContext(sc) {
  while (!stopped) {
    try {
      val reference = Option(referenceQueue.remove(ContextCleaner.REF_QUEUE_POLL_TIMEOUT))
        .map(_.asInstanceOf[CleanupTaskWeakReference])
      // Synchronize here to avoid being interrupted on stop()
      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)
          }
        }
      }
    } catch {
      case ie: InterruptedException if stopped => // ignore
      case e: Exception => logError("Error in cleaning thread", e)
    }
  }
}
```

org.apache.spark.ContextCleaner

```scala
/** Perform shuffle cleanup. */
def doCleanupShuffle(shuffleId: Int, blocking: Boolean): Unit = {
  try {
    logDebug("Cleaning shuffle " + shuffleId)
    mapOutputTrackerMaster.unregisterShuffle(shuffleId)
    blockManagerMaster.removeShuffle(shuffleId, blocking)
    listeners.asScala.foreach(_.shuffleCleaned(shuffleId))
    logInfo("Cleaned shuffle " + shuffleId)
  } catch {
    case e: Exception => logError("Error cleaning shuffle " + shuffleId, e)
  }
}
```

blockManager里边的操作都是需要跨越多台机器，这一操作比较慢，尤其是在集群大量跑着任务时；
当出现Network IO问题时，缺乏鲁棒性，极端情况下可能这根调用链压根就丢失；



执行级别的垃圾（Shuffle，Broadcast，Task Binary…）严重依赖的Driver端的jvm gc触发（另一种形式的SPOF）；
涉及很多往返通信，IO出问题，清除流程非常脆弱；

org.apache.spark.ContextCleaner

```
/** Register an RDD for cleanup when it is garbage collected. */
def registerRDDForCleanup(rdd: RDD[_]): Unit = {
  registerForCleanup(rdd, CleanRDD(rdd.id))
}
```

第一类：RDD。仅当一个RDD的*storageLevel*为StorageLevel.*NONE*时，并且调用persist时，该RDD才会通过注册为回收，后续依靠Driver端的GC触发ContextCleanr然后把清理操作传导到Executor，通过BlockManager执行清楚。对于offheap：直接通过buffer的depose清除；对于onheap：将对应的Block标记清除后，最终依赖Executor端的GC回收。

```
def registerAccumulatorForCleanup(a: AccumulatorV2[_, _]): Unit = {
  registerForCleanup(a, CleanAccum(a.id))
}
```

第二类：Accumulator。这个在Spark SQL中执行聚合计算时用得比较多，相当于一个分布式的累加器，该累加器的回收完全依赖于ContextCleaner，与所从属于的某个Job无关，当然回收与否也与其对应的Job无关。

```
/** Register a ShuffleDependency for cleanup when it is garbage collected. */
def registerShuffleForCleanup(shuffleDependency: ShuffleDependency[_, _, _]): Unit = {
  registerForCleanup(shuffleDependency, CleanShuffle(shuffleDependency.shuffleId))
}
```

第三类：Shuffle Block。做Shuffle操作时引起的一些临时数据，这些数据一般为Execution Data。当一个ShuffleDependency初始化时，把其注册到ContextCleaner，其回收与从属的Job无关，完全依赖ContextCleaner回收。

```
/** Register a Broadcast for cleanup when it is garbage collected. */
def registerBroadcastForCleanup[T](broadcast: Broadcast[T]): Unit = {
  registerForCleanup(broadcast, CleanBroadcast(broadcast.id))
}
```

第四类：Braodcast Block。做广播操作时，注册到ContextCleaner，后期广播数据的回收完全依赖ContextCleaner，与Job无关。

```
/** Register a RDDCheckpointData for cleanup when it is garbage collected. */
def registerRDDCheckpointDataForCleanup[T](rdd: RDD[_], parentId: Int): Unit = {
  registerForCleanup(rdd, CleanCheckpoint(parentId))
}
```

第五类：RDDCheckpointData。对RDD做Checkpiont操作时，注册到ContextCleaner，但该处的数据大部分存在磁盘中，（与Memory关系不大），其回收完全依赖ContextCleaner，与其对应Job无关。

应用数据RDD，由用户管理（是否cache，是否unpersist）；执行期数据由ContextCleaner负责（非常类似Jvm中的GC）；Executor执行回收时，发现该Block正在读，阻塞等待，当读锁释放后，做回收操作。