# Spark源码分析之SparkConf

前言

SparkConf类负责管理Spark的所有配置项。在我们使用Spark的过程中，经常需要灵活配置各种参数，来使程序更好、更快地运行，因此也必然要与SparkConf类频繁打交道。了解它的细节不无裨益。

Spark主要提供三种修改配置方法

* **Spark properties:** 可以控制绝大多数应用程序参数，而且可以通过SparkConf对象来设置，也可以通过java系统属性来设置，比如程序中在创建SparkContext之前，使用System.setProperty("xxx","xxx")语句设置相应系统属性。
* **Environment variables:** 可以指定一些各个机器相关的设置，如IP地址，其设置方法是写在每台机器的conf/spark-env.sh文件中。
* **日志配置**：通过log4j.properties实现

SparkConf类的构造方法

class SparkConf(loadDefaults: Boolean) extends Cloneable with Logging with Serializable {

import SparkConf.\_

if (loadDefaults) {

loadFromSystemProperties(false)

}

def this() = this(true)

// ...

}

import语句是从SparkConf类的伴生对象中导入一些东西，它们主要管理过期的，旧版兼容的配置项以及日志输出。Scala中没有java静态（static）概念，类的伴生对象中维护的成员和方法就可以视为类的静态成员和静态方法。

SparkConf类中有一个主构造方法参数loadDefaults，它指示是否要从Java系统属性（即System.getProperties()取得的属性）加载默认的以"spark."开头的配置

Spark配置项的存储

SparkConf内部是采用ConcurrentHashMap来维护所有的配置项键值的，代码如下

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

这自然是考虑到了并发环境下的线程安全性问题。另外，它的键与值类型都为String，说明所有Spark配置项都以字符串形式存储。

设置配置项

要设置spark配置项，有一下三种方法。

直接用Set类方法设置

这是我们开发过程中最常用的方法。SparkConf中提供了多种多样的Set类方法，最基础的set()方法重载如下。

private[spark] def set(key: String, value: String, silent: Boolean): SparkConf = {

if (key == null) {

throw new NullPointerException("null key")

}

if (value == null) {

throw new NullPointerException("null value for " + key)

}

if (!silent) {

logDeprecationWarning(key)

}

settings.put(key, value)

this

}

可见配置项的键值都不能为null。并且包括set()在内的所有Set类方法都返回this，所以支持链式调用，这样使用起来比较简洁。

另外，还有一些方法可以快速设置常用配置项，比如setMaster()、setAppName()、setJars()等函数。它们最终也会调用set()方法。

通过系统属性加载

如果上文提到的loadDefaults参数为true，那么SparkConf会从java系统属性中加载配置项。如果调用无参的辅助构造方法，即直接new SparkConf()的话，也会将loadDefaults设为true。Java系统属性可以通过System.setProperty()方法在程序中动态设置。

SparkConf.loadFromSystemProperties()方法代码

private[spark] def loadFromSystemProperties(silent: Boolean): SparkConf = {

// Load any spark.\* system properties

for ((key, value) <- Utils.getSystemProperties if key.startsWith("spark.")) {

set(key, value, silent)

}

this

}

它使用通用工具类Utils中的方法取得系统属性，过滤出以字符串“spark.”为前缀的键，然后调用set()方法设置键值。由于系统属性相关的参数是一次性初始化的，所以用Set类方法设置的值可以覆盖它们。

克隆SparkConf

SparkConf类继承了Cloneable特征（trait，类似于Java接口的增强版）并覆写了clone()方法，因此SparkConf是可以（深）克隆的。

override def clone: SparkConf = {

val cloned = new SparkConf(false)

settings.entrySet().asScala.foreach { e =>

cloned.set(e.getKey(), e.getValue(), true)

}

cloned

}

虽然ConcourrentHashMap保证线程安全，不会影响SparkConf实例共享，但是在高并发的情况下，锁机制可能会带来性能问题。我们就可以克隆SparkConf到多个组件中，以让它获得相同的配置参数

获取配置项

获取配置项只有一个途径，即调用Get类方法。Get类方法同样有很多实现，基于get()和getOption()如下所示。

def get(key: String): String = {

getOption(key).getOrElse(throw new NoSuchElementException(key))

}

def get(key: String, defaultValue: String): String = {

getOption(key).getOrElse(defaultValue)

}

def getOption(key: String): Option[String] = {

Option(settings.get(key)).orElse(getDeprecatedConfig(key, settings))

}

获取配置项时，会同时检查过期配置（getDeprecatedConfig()方法是伴生对象中定义的），并且会使用Scala Option来包装返回的结果，对于有值（Some）和无值（None）的情况可以灵活处理。

另外，Get类方法中有不少涉及数据类型转换和单位转换，如getDouble()、getLong()、getSizeAsMb()、getTimeAsSeconds()等等，这些函数都是为了使用。

校验配置项

SparkConf中有一个方法validateSettings()，用来校验配置项。它的源码很长，但是逻辑比较简单，主要是对过期配置项进行警告，以及对非法设置或不兼容的配置项抛出异常。

总结

本文通过SparkConf类的部分源码，简述了SparkConf的构造方法、配置存储，以及设置、获取、校验配置项的方法逻辑。

SparkConf是SparkContext初始化的必备前提。了解了SparkConf，就可以分析复杂得多的SparkContext了。

**应用配置：**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.submit.deployMode |  | client/cluster |
| spark.master |  | spark应用连接的master |
| spark.local.dir | /tmp | 用于spark应用中空间划分，map输出结果、RDD结果存储 |
| spark.driver.cores | 1 | 在cluster模式中有效，driver进程使用的core数量 |
| spark.app.name |  | 应用名称，在UI和日志中使用 |
| spark.driver.maxResultSize | 1g | spark中每一个action计算所有分区的序列化结果大小，超出这个值，程序将会终止 |
| spark.driver.memory | 1g | driver进程运行的内存大小。可以在程序代码中通过SparkConf进行设置，也可以在spark-submit中使用--driver-memory进行设置 |
| spark.extraListeners |  | 实现SparkListener的监听器，在SparkContext初始化时创建出来 |
| spark.logConf | false | SparkContext中有效的日志设置（INFO级） |

**运行环境：**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.jars |  | driver和executor将会加载的jars |
| spark.files |  | 每一个worker执行的工作目录 |
| spark.redaction.regex | "(?i)secret|password|token".r | 正则表达式决定在driver和executor环境中，哪些spark配置属性和环境变量包含敏感信息。那些匹配这个正则匹配表达式的变量的值在UI和日志中显示时，会自动进行加密处理 |
| spark.executor.logs.rolling.strategy |  | 设置执行程序日志的滚动策略。默认情况下，它被禁用。它可以设置为“时间”（基于时间的滚动）或“大小”（基于大小的滚动）。对于“时间”，用spark.executor.logs.rolling.time.interval设置滚动间隔。对于“大小”，用于spark.executor.logs.rolling.maxSize设置滚动的最大文件大小。 |
| spark.executor.logs.rolling.time.interval | daily | 设置执行程序日志将被滚动的时间间隔。默认情况下禁用滚动。有效值是daily，hourly，minutely或在几秒钟内的任何时间间隔。请参阅spark.executor.logs.rolling.maxRetainedFiles 自动清洁旧日志。 |
| spark.executor.logs.rolling.maxSize | 1MB | 设置文件的最大大小（以字节为单位），执行程序日志将通过该字节翻转。默认情况下禁用滚动。请参阅spark.executor.logs.rolling.maxRetainedFiles 自动清洁旧日志。 |
| spark.executor.logs.rolling.maxRetainedFiles |  | 设置将由系统保留的最新滚动日志文件的数量。旧的日志文件将被删除。默认情况下禁用。 |
| spark.executor.logs.rolling.enableCompression | false | 启用执行程序日志压缩。如果已启用，则将压缩已滚动的执行程序日志。默认情况下禁用。 |
| spark.cleaner.periodicGC.interval | 30min | 控制触发垃圾回收的频率。仅当弱引用被垃圾收集时，此上下文清除程序才会触发清理。在具有大型驱动程序JVM的长时间运行的应用程序中，驱动程序上的内存压力很小，这可能偶尔发生或根本不发生。根本不清理可能会导致执行程序在一段时间后耗尽磁盘空间。 |
| spark.cleaner.referenceTracking | true | 启用或禁用上下文清理。 |
| spark.cleaner.referenceTracking.blocking | true | 控制清理线程是否应该阻止清除任务（除了由spark.cleaner.referenceTracking.blocking.shuffleSpark属性控制的shuffle）。 |
| spark.cleaner.referenceTracking.blocking.shuffle | false | 控制清理线程是否应阻止随机清理任务。 |
| spark.cleaner.referenceTracking.cleanCheckpoints | false | 控制是否在引用超出范围时清除检查点文件。 |

**内存管理：**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.memory.fraction | 0.6 | 堆内存中用于执行、混洗和存储（缓存）的比例。这个值越低，则执行中溢出到磁盘越频繁，同时缓存被逐出内存也更频繁。这个配置的目的，是为了留出用户自定义数据结构、内部元数据使用的内存。推荐使用默认值 |
| spark.memory.storageFraction | 0.5 | 缓存数据(cached data)不会被逐出内存写入磁盘的总量，表示一个相对于 spark.memory.fraction的比例。这个越高，那么执行混洗等操作用的内存就越少，从而溢出磁盘就越频繁。 |
| spark.memory.offHeap.enabled | true | 如果true，Spark会尝试使用堆外内存。启用后，spark.memory.offHeap.size必须为正数。 |
| spark.memory.offHeap.size | 0 | 堆外内存分配的大小（绝对值）。这个设置不会影响堆内存的使用，所以你的执行器总内存必须适应JVM的堆内存大小。必须要设为正数。并且前提是 spark.memory.offHeap.enabled=true. |
| spark.memory.useLegayMode |  |  |
| spark.storgae.unrollMemoryThreshold |  | 在展开任何block之前请求的初始内存 |
| spark.storage.safetyFraction | 0.9 | jvm中的storage memory的安全比例大小，默认值90%，留一个安全内存区的原因可能是内存使用大小的计算不是一个精确值，留有余地可以避免一个可能很大的对象放进去之后导致内存不够。所以如果storagememory绝对值很大的话，理论上这个比例可以设置高一点 |
| spark.buffer.pageSize |  |  |

**执行行为：**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.broadcast.blockSize | 4mb | TorrentBroadcastFactory每个分片大小。太大会减少广播时候的并发数（更慢了）；如果太小，BlockManager可能会给出性能提示。 |
| spark.broadcast.checksum | true | Whether to enable checksum for broadcast |
| spark.broadcast.UDFCompressionThreshold | 1mb | 超过这个阈值，user-defined function和Python RDD命令将通过broadcast被压缩 |
| spark.storage.memoryMapThreshold | 2m | spark从磁盘上读取一个块后，映射到内存块的最小大小。这阻止了spark映射过小的内存块。通常，内存映射块是有开销的，应该比接近或小于操作系统的页大小 |
| spark.storage.replication.proactive | false | 为RRD blocks主动进行块复制，默认为false。如果存在可用的副本，那些因为executor执行失败而丢失的缓存RDD将会被补充。系统将尝试将block副本数恢复到初始数值 |
| spark.storage.replication.policy | RandomBlockReplicationPolicy | 复制策略 |
| spark.storage.replication.topologyMapper | DefaultTopologyMapper |  |
| spark.storage.cachedPeersTtl | 60\*1000 |  |
| spark.storage.maxReplicationFailures | 1 |  |
| spark.storage.blockManagerTimeoutIntervalMs | 60s | 指定检查BlockManager超时的时间间隔 |
| spark.storage.blockManagerSlaveTimeoutMs | Network.NETWORK\_TIMEOUT.defaultValueString  120s |  |
| spark.diskStore.subDirectories | 64 | spark.local.dir中列出的每个路径中的子目录数 |
| spark.storage.cleanupFilesAfterExecutorExit | **true** | 档executor退出时是否清除外部shuffle服务未提供得到文件 |
| spark.block.failures.beforeLocationRefresh | **5** | 块管理器从driver上更新block位置的最大失败次数 |
| spark.files.ignoreCorruptFiles | **false** | 是否忽略损毁的文件，如果设成true，当遇到损坏或者不存在的文件或者文本时，spark job将继续运行并返回已经读取的数据 |
| spark.files.ignoreMissingFiles | **false** | 是否忽略丢失的文件 |
| spark.files.maxPartitionBytes | **128MB** | 读取文件时，打包到单个分区的最大字节数 |
| spark.files.openCostInBytes | **4MB** | 小文件合并阈值，小于该参数就会被合并到一个partition内。  默认4194304 (4 MB) 。这个参数在将多个文件放入一个partition时被用到，宁可设置的小一些，因为在partition操作中，小文件肯定会比大文件快。 |
| spark.buffer.write.chunkSize | **64MB** | 将缓存数据写出时的Chunk大小 |
| spark.checkpoint.compress | **false** | 是否压缩RDD的checkpoints，如果压缩将使用spark.io.compression.codec |
| spark.storage.unrollMemoryCheckPeriod | 16 | 内存检查周期，用来决定当我们尝试在内存中unroll给定的block时，多久检查一次是否有请求更多内存的需求 |
| spark.storage.unrollMemoryGrowthFactor | 1.5 | Memory to request as a multiple of the size that used to unroll the block |
|  |  |  |

动态分配

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.dynamicAllocation.enabled | false | 是否启用动态资源分配特性，启用后，executor的个数会根据工作负载动态的调整（增加或减少）。注意，目前在YARN模式下不用。更详细信息，请参考：here该特性依赖于 spark.shuffle.service.enabled 的启用。同时还和以下配置相关：spark.dynamicAllocation.minExecutors, spark.dynamicAllocation.maxExecutors以及 spark.dynamicAllocation.initialExecutors |
| spark.dynamicAllocation.executorIdleTimeout | 60s | 动态分配特性启用后，空闲时间超过该配置时间的执行器都会被移除 |
| spark.dynamicAllocation.cachedExecutorIdleTimeout | infinity | 动态分配特性启用后，包含缓存数据的执行器如果空闲时间超过该配置设置的时间，则被移除。 |
| spark.dynamicAllocation.initialExecutors | spark  .dynamicAllocation  .minExecutors | 动态分配开启后，执行器的初始个数 |
| spark.dynamicAllocation.maxExecutors | infinity | 动态分配开启后，执行器个数的上限 |
| spark.dynamicAllocation.minExecutors | 0 | 动态分配开启后，执行器个数的下限 |
| spark.dynamicAllocation.schedulerBacklogTimeout | 1s | 动态分配启用后，如果有任务积压的持续时间长于该配置设置的时间，则申请新的执行器。 |
| spark.dynamicAllocation.sustainedSchedulerBacklogTimeout |  | 和spark.dynamicAllocation.schedulerBacklogTimeout类似，只不过该配置对应于随后持续的执行器申请。 |

**调度**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.locality.wait | 3s | 为了数据本地性最长等待时间（spark会根据数据所在位置，尽量让任务也启动于相同的节点，然而可能因为该节点上资源不足等原因，无法满足这个任务分配，spark最多等待这么多时间，然后放弃数据本地性）。数据本地性有多个级别，每一级别都是等待这么多时间（同一进程、同一节点、同一机架、任意）。你也可以为每个级别定义不同的等待时间，需要设置spark.locality.wait.node等。如果你发现任务数据本地性不佳，可以增加这个值，但通常默认值是ok的。 |
| spark.scheduler.listenerbus.eventqueue.capacity | 10000 | spark事件监听队列容量，默认10000，必须为正值，增加可能会消耗更多内存 |
| spark.locality.wait.process | spark.locality.wait | 自定义位置等待进程位置。这会影响尝试访问特定执行程序进程中的缓存数据的任务。 |
| spark.locality.wait.node | spark.locality.wait | 自定义位置等待节点位置。例如，您可以将此值设置为0以跳过节点位置并立即搜索机架位置（如果您的群集有机架信息）。 |
| spark.locality.wait.rack | spark.locality.wait | 自定义本地等待机架位置。 |
| spark.reducer.maxSizeInFlight | 48mb | 除非另有说明，否则在MiB中同时从每个reduce任务获取的映射输出的最大大小。由于每个输出都需要我们创建一个缓冲区来接收它，这表示每个reduce任务的固定内存开销，所以除非你有大量的内存，否则保持它很小。 |
| spark.speculation.quantile | 0.75 | 对于一个stage来说，完成多少百分比才开始检查慢任务，并启动推测执行任务。 |
| spark.speculation.multiplier | 1.5 | 比任务平均执行时间慢多少倍的任务会被认为是慢任务。 |
| spark.speculation.interval | 100ms | Spark检查慢任务的时间间隔。 |
| spark.speculation | false | 如果设为true，将会启动推测执行任务。这意味着，如果stage中有任务执行较慢，他们会被重新调度到别的节点上执行。 |
| spark.task.cpus | 1 | 每个任务分配的CPU core。 |
| spark.task.maxFailures | 4 | 单个任务最大失败次数。应该>=1。最大重试次数 = spark.task.maxFailures – 1 |
| spark.scheduler.revive.interval | 1s | 调度器复活worker的间隔时间。 |
| spark.scheduler.mode | FIFO | 提交到同一个SparkContext上job的调度模式（scheduling mode）。另一个可接受的值是FAIR，而FIFO只是简单的把job按先来后到排队。对于多用户服务很有用。 |
| spark.scheduler.maxRegisteredResourcesWaitingTime | 30s | 调度开始前，向集群管理器注册使用资源的最大等待时间。 |
| spark.scheduler.minRegisteredResourcesRatio |  | 调度启动前，需要注册得到资源的最小比例（注册到的资源数 / 需要资源总数）（YARN模式下，资源是执行器；独立部署和Mesos粗粒度模式下时资源是CPU core【spark.cores.max是期望得到的资源总数】）。可以设为0.0~1.0的一个浮点数。不管job是否得到了最小资源比例，最大等待时间都是由spark.scheduler.maxRegisteredResourcesWaitingTime控制的。 |
| spark.scheduler.allocation.file |  |  |

Task

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.task.cpus | 1 | 每个task分配的cpu数 |
| spark.task.maxFailures | 4 | 在放弃这个job前允许的最大失败次数，重试次数为该参数 |
| spark.task.maxDirectResultSize | 1m |  |
| spark.task.reaper.enabled | false | 赋予spark监控有权限去kill那些失效的task |
| spark.task.reaper.killTimeout | -1 | 当一个被kill的task过了多久还在running，就会把那个executor给kill掉 |
| spark.task.reaper.pollingInterval | 10s | 轮询被kill掉的task的时间间隔，如果还在running，就会打印warn日志，默认10s |
| spark.task.reaper.threadDump | true | 线程回收是否产生日志 |
| spark.blacklist.enabled | false | 是否列入黑名单。如果为true，当一个executor失败好几次时，会被列入黑名单，防止后续task派发到这个executor。 |

**Shuffle行为**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.io.encryption.enabled | false | 启用IO加密。目前除了Mesos之外的所有模式都支持。建议在使用此功能时启用RPC加密。 |
| spark.io.encryption.keySizeBits | 128 | IO加密密钥大小（位）。支持的值为128,192和256。 |
| spark.io.encryption.keygen.algorithm | HmacSHA1 | 生成IO加密密钥时使用的算法。Java Cryptography体系结构标准算法名称文档的KeyGenerator部分中描述了支持的算法。 |
| spark.io.crypto.cipher.transformation | AES/CTR/NoPadding |  |
| spark.shuffle.accurateBlockThreshold | 100MB | HighlyCompressedMapStatus中高于该阈值的shuffle block大小会被准确记录。这有助于通过避免在获取shuffle块时低估shuffle块大小来防止OOM。 |
| spark.shuffle.registration.timeout | 5000 | 单位毫秒，注册外部shuffle服务的超时时间 |
| spark.shuffle.registration.maxAttempts | 3 | 注册外部shuffle服务的最大尝试次数 |
| spark.reducer.maxBlocksInFlightPerAddress | Int.MaxValue | 该配置限制每个reduce任务从一个端口获取remote blocks的数量。当一次从给定地址请求大量块时，这可能会使正在运行的exector或node manager崩溃。当启用externl shuffle时，这对于减少node manager的负载特别有用。您可以通过将其设置为较低的值来缓解此问题。 |
| spark.maxRemoteBlockSizeFetchToMem | 200MB | 当Block大小高于此阈值时，远程块将被提取到磁盘。这是为了避免巨大的请求占用太多内存。我们可以通过设置特定值（例如200m）来启用此配置。请注意，此配置将影响随机提取和块管理器远程块提取。对于启用了外部shuffle服务的用户，只有当外部shuffle服务比Spark 2.2更新时，才能使用此功能。 |
| spark.shuffle.file.buffer | 32k | 每一个Shuffle输出流的内存buffer大小.这个buffer能减少shuffle文件的创建和磁盘寻址。 |
| spark.shuffle.spill.diskWriteBufferSize | 1MB | 当the sorted records到磁盘文件上时，使用的buffer大小 |
| spark.shuffle.spill.numElementsForceSpillThreshold | Integer.MAX\_VALUE | 在使用UnsafeShuffleWriter时,如果内存中的数据超过这个值则对当前内存数据进行排序并写入磁盘临时文件。 |
| spark.shuffle.mapOutput.parallelAggregationThreshold | 10000000 | 用于配置使用多线程聚合的阈值，聚合的并行度由JVM中可用core数和Mappers \* Shuffle partitions / threshold + 1中的小值决定 |
| spark.shuffle.sort.initialBufferSize | 4MB | 初始化的排序缓冲大小，默认为4096Byte(4KB,这个比SortedBaseShuffleWrite那个初始化的缓冲大小要小多了,那个是5M,反正也会扩容的) |
| spark.shuffle.compress | true | 是否压缩map任务的输出文件。通常来说，压缩是个好主意。使用的压缩算法取决于 spark.io.compression.codec |
| spark.shuffle.spill.compress | true | 是否在shuffle阶段压缩溢出到磁盘的数据。压缩算法取决于spark.io.compression.codec |
| spark.shuffle.spill.initialMemoryThreshold | 5MB | 开始追踪collection内存使用情况的初始化阈值 |
| spark.shuffle.spill.batchSize | 10000 | shuffle过程中读和写serializers的批数据大小 |
| spark.shuffle.sort.bypassMergeThreshold | 200 | 在基于排序（sort）的shuffle管理器中，如果没有map端聚合的话，就会最多存在这么多个reduce分区。 |
| spark.shuffle.manager | sort | shuffle数据的实现方式。可用的有”sort”和”hash“。基于排序（sort）的shuffle内存利用率更高，并且从1.2开始已经是默认值了。 |
| spark.shuffle.reduceLocality.enabled | true | Whether to compute locality preferences for reduce tasks |
| spark.shuffle.mapOutput.minSizeForBroadcast | 512k | 使用Broadcast的方式发送map output statuses给executors的大小 |
| spark.shuffle.mapOutput.dispatcher.numThreads | 8 |  |
| spark.shuffle.detectCorrupt | true | 在shuffle过程中是否检测获取的blocks中有损坏的 |
| spark.shuffle.detectCorrupt.useExtraMemory | false | 如果为true，部分compressed/encrypted stream在de-compressed/de-crypted时，会使用extra memroy来检测坏块。任何IOException触发将导致任务重试，如果下次还是出现同样的错误，将扔出FetchFailedException |
| spark.shuffle.sync | false | 是否强制将未完成的数据写入磁盘 |
| spark.shuffle.unsafe.fastMergeEnabled | true | 是否执行快速溢出合并 |
| spark.shuffle.sort.useRadixSort | true | 是否使用基数排序来对内存中的分区ID进行排序。基数排序更快但在添加指针时需要额外的内存来保留内存 |
| spark.shuffle.minNumPartitionsToHighlyCompress | 200 | 确定MapStatus是否应使用HighlyCompressedMapStatus的分区数 |
| spark.barrier.sync.timeout | 365d | 从barrier task 调用的barrier()函数超时的时间 |
| spark.reducer.maxSizeInFlight | 48mb | 这个配置限制了远程获取blocks的大小。当集群的主机数增加，可能导致大量的inbound connections to one or more nodes，从而导致workers不堪重负挂掉。通过限制获取请求的数量，可以减轻这种情况 |
|  |  |  |

**网络**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.network.crypto.enabled | false | 使用commons-crypto库为RPC和块传输服务启用加密。需要spark.authenticate启用。 |
| spark.network.remoteReadNioBufferConversion |  |  |
| spark.network.timeout | 120s | 所有网络交互的默认超时。这个配置是以下属性的默认值：  spark.core.connection.ack.wait.timeout,  spark.akka.timeout,  spark.storage.blockManagerSlaveTimeoutMs,  spark.shuffle.io.connectionTimeout,spark.rpc.askTimeout or  spark.rpc.lookupTimeout |
| spark.network.timeoutInterval |  |  |
| spark.network.timeout | 120s | 所有网络交互的默认超时。这个配置是以下属性的默认值：  spark.core.connection.ack.wait.timeout,  spark.akka.timeout,  spark.storage.blockManagerSlaveTimeoutMs,  spark.shuffle.io.connectionTimeout,spark.rpc.askTimeout or  spark.rpc.lookupTimeout |
| spark.rpc.connect.threads |  |  |
| spark.rpc.io.numConnectionsPerPeer |  |  |
| spark.rpc.io.threads |  |  |
| spark.rpc.lookupTimeout | 120s | RPC远程端点查询超时。 |
| spark.rpc.message.maxSize | 128 | “控制平面”通信中允许的最大消息大小（以MB为单位）; 通常仅适用于执行程序和驱动程序之间发送的映射输出大小信息。如果您正在运行具有数千个map和reduce任务的作业并查看有关RPC消息大小的消息，请增加此值。 |
| spark.rpc.netty.dispatcher.numThreads |  |  |
| spark.rpc.numRetries | 3 | RPC任务最大重试次数。RPC任务最多重试这么多次。 |
| spark.rpc.retry.wait | 3s | RPC请求操作重试前等待时间。 |
| spark.driver.host | (local hostname) | driver主机名称，用于和executors已经独立部署时集群master通信 |
| spark.driver.port | 0 | driver进程端口，用于和executors已经独立部署时集群master通信 |
| spark.driver.supervise | false | 如果为true，则在失败且退出状态为非零时自动重新启动驱动程序。仅在Spark独立模式或Mesos集群部署模式下有效。 |
| spark.driver.bindAddress | （spark.driver.host的值） | 主机名或IP地址绑定侦听套接字的位置。此配置会覆盖SPARK\_LOCAL\_IP环境变量（请参见下文）。 它还允许将与本地地址不同的地址通告给执行者或外部系统。例如，在运行具有桥接网络的容器时，这很有用。为了正常工作，需要从容器的主机转发驱动程序使用的不同端口（RPC，块管理器和UI）。 |
| spark.blockManager.port | （随机） | 所有块管理器的端口都可以监听。它们存在于executor和driver中。 |
| spark.driver.blockManager.port | （  spark.blockManager.port的值  ） | 如果不能使用与executor相同的端口，则driver端使用该端口 |

**压缩和序列化**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.broadcast.compress | true | 是否在发送之前压缩广播变量。一般来说是个好主意。压缩将使用spark.io.compression.codec。 |
| spark.io.compression.snappy.blockSize | 32k | 在使用Snappy压缩编解码器的情况下，在Snappy压缩中使用的块大小（以字节为单位）。使用Snappy时，降低此块大小也会降低随机内存使用量。 |
| spark.io.compression.lz4.blockSize | 32k | 在使用LZ4压缩编解码器的情况下，LZ4压缩中使用的块大小（以字节为单位）。当使用LZ4时，降低此块大小也会降低随机内存使用量。 |
| spark.io.compression.codec | lz4 | 用于压缩内部数据的编解码器，例如RDD分区，事件日志，广播变量和随机输出。默认情况下，spark提供了四个编解码器：lz4，lzf， snappy，和zstd。您还可以使用完全限定类名来指定的编解码器，例如 org.apache.spark.io.LZ4CompressionCodec，org.apache.spark.io.LZFCompressionCodec，org.apache.spark.io.SnappyCompressionCodec，和org.apache.spark.io.ZstdCompressionCodec |
| spark.io.compression.zstd.bufferSize | 32k | 在使用Zstd压缩编解码器的情况下，在Zstd压缩中使用的缓冲区大小（以字节为单位）。降低此大小将降低使用Zstd时的随机内存使用量，但由于过多的JNI调用开销，可能会增加压缩成本。 |
| spark.io.compression.zstd.level | 1 | Zstd压缩编解码器的压缩级别。增加压缩级别将导致更好的压缩，代价是更多的CPU和内存。 |
| spark.io.warning.largeFileThreshold | 1GB | 当spark加载一个大文件时，如果文件大小超出这个阈值，日志将告警可能原因 |
| spark.eventLog.compression.codec | lz4 |  |
| spark.rdd.compress | false | 是否压缩序列化RDD分区（例如，用于StorageLevel.MEMORY\_ONLY\_SERJava和Scala或StorageLevel.MEMORY\_ONLYPython）。可以节省大量空间，但需要额外的CPU时间。压缩将使用spark.io.compression.codec。 |
| spark.serializer | org.apache.spark.serializer。JavaSerializer | 用于序列化将通过网络发送或需要以序列化形式缓存的对象的类。Java序列化的默认设置适用于任何Serializable Java对象但速度很慢，因此我们建议 在需要速度时使用org.apache.spark.serializer.KryoSerializer和配置Kryo序列化。可以是任何子类 。 org.apache.spark.Serializer |

**部署**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.deploy.recoveryMode |  | 恢复模式设置，用于在失败并重新启动时使用群集模式恢复提交的Spark作业。这仅适用于使用Standalone或Mesos运行时的群集模式。 |
| spark.deploy.zookeeper.url |  | 当`spark.deploy.recoveryMode`设置为ZOOKEEPER时，此配置用于设置要连接的zookeeper URL。 |
| spark.deploy.zookeeper.dir |  | 当`spark.deploy.recoveryMode`设置为ZOOKEEPER时，此配置用于将zookeeper目录设置为存储恢复状态。 |

**UI**

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.master.rest.enabled | false |  |
| spark.master.rest.port | 6066 |  |
| spark.master.ui.port | 8088 |  |
| spark.ui.retainedJobs | 1000 | SparkUI和status API最多保留多少个spark作业的数据（当然是在垃圾回收之前） |
| spark.ui.retainedStages | 1000 | SparkUI和status API最多保留多少个spark步骤（stage）的数据（当然是在垃圾回收之前） |
| spark.ui.retainedTasks |  |  |
| spark.ui.retainedDeadExecutors |  |  |