Spark 性能调优参数总结

Shuffle 相关

Shuffle 操作大概是对 Spark 性能影响最大的步骤之一（因为可能涉及到排序，磁盘 IO，网络 IO 等众多 CPU 或 IO 密集的操作），这也是为什么在 Spark 1.1 的代码中对整个 Shuffle 框架代码进行了重构，将 Shuffle 相关读写操作抽象封装到 Pluggable 的 Shuffle Manager 中，便于试验和实现不同的 Shuffle 功能模块。例如为了解决 Hash Based 的 Shuffle Manager 在文件读写效率方面的问题而实现的 Sort Base 的 Shuffle Manager。

spark.shuffle.manager

用来配置所使用的 Shuffle Manager，目前可选的 Shuffle Manager 包括默认的 org.apache.spark.shuffle.sort.HashShuffleManager（配置参数值为 hash）和新的 org.apache.spark.shuffle.sort.SortShuffleManager（配置参数值为 sort）。

这两个 ShuffleManager 如何选择呢，首先需要了解他们在实现方式上的区别。

HashShuffleManager，故名思义也就是在 Shuffle 的过程中写数据时不做排序操作，只是将数据根据 Hash 的结果，将各个 Reduce 分区的数据写到各自的磁盘文件中。带来的问题就是如果 Reduce 分区的数量比较大的话，将会产生大量的磁盘文件。如果文件数量特别巨大，对文件读写的性能会带来比较大的影响，此外由于同时打开的文件句柄数量众多，序列化，以及压缩等操作需要分配的临时内存空间也可能会迅速膨胀到无法接受的地步，对内存的使用和 GC 带来很大的压力，在 Executor 内存比较小的情况下尤为突出，例如 Spark on Yarn 模式。

SortShuffleManager，是 1.1 版本之后实现的一个试验性（也就是一些功能和接口还在开发演变中）的 ShuffleManager，它在写入分区数据的时候，首先会根据实际情况对数据采用不同的方式进行排序操作，底线是至少按照 Reduce 分区 Partition 进行排序，这样来至于同一个 Map 任务 Shuffle 到不同的 Reduce 分区中去的所有数据都可以写入到同一个外部磁盘文件中去，用简单的 Offset 标志不同 Reduce 分区的数据在这个文件中的偏移量。这样一个 Map 任务就只需要生成一个 shuffle 文件，从而避免了上述 HashShuffleManager 可能遇到的文件数量巨大的问题

两者的性能比较，取决于内存，排序，文件操作等因素的综合影响。

对于不需要进行排序的 Shuffle 操作来说，如 repartition 等，如果文件数量不是特别巨大，HashShuffleManager 面临的内存问题不大，而 SortShuffleManager 需要额外的根据 Partition 进行排序，显然 HashShuffleManager 的效率会更高。

而对于本来就需要在 Map 端进行排序的 Shuffle 操作来说，如 ReduceByKey 等，使用 HashShuffleManager 虽然在写数据时不排序，但在其它的步骤中仍然需要排序，而 SortShuffleManager 则可以将写数据和排序两个工作合并在一起执行，因此即使不考虑 HashShuffleManager 的内存使用问题，SortShuffleManager 依旧可能更快。

spark.shuffle.sort.bypassMergeThreshold

这个参数仅适用于 SortShuffleManager，如前所述，SortShuffleManager 在处理不需要排序的 Shuffle 操作时，由于排序带来性能的下降。这个参数决定了在这种情况下，当 Reduce 分区的数量小于多少的时候，在 SortShuffleManager 内部不使用 Merge Sort 的方式处理数据，而是与 Hash Shuffle 类似，直接将分区文件写入单独的文件，不同的是，在最后一步还是会将这些文件合并成一个单独的文件。这样通过去除 Sort 步骤来加快处理速度，代价是需要并发打开多个文件，所以内存消耗量增加，本质上是相对 HashShuffleMananger 一个折衷方案。 这个参数的默认值是 200 个分区，如果内存 GC 问题严重，可以降低这个值。

spark.shuffle.consolidateFiles

这个配置参数仅适用于 HashShuffleMananger 的实现，同样是为了解决生成过多文件的问题，采用的方式是在不同批次运行的 Map 任务之间重用 Shuffle 输出文件，也就是说合并的是不同批次的 Map 任务的输出数据，但是每个 Map 任务所需要的文件还是取决于 Reduce 分区的数量，因此，它并不减少同时打开的输出文件的数量，因此对内存使用量的减少并没有帮助。只是 HashShuffleManager 里的一个折中的解决方案。

需要注意的是，这部分的代码实现尽管原理上说很简单，但是涉及到底层具体的文件系统的实现和限制等因素，例如在并发访问等方面，需要处理的细节很多，因此一直存在着这样那样的 bug 或者问题，导致在例如 EXT3 上使用时，特定情况下性能反而可能下降，因此从 Spark 0.8 的代码开始，一直还没有被标志为 Stable，不是默认采用的方式。此外因为并不减少同时打开的输出文件的数量，因此对性能具体能带来多大的改善也取决于具体的文件数量的情况。所以即使你面临着 Shuffle 文件数量巨大的问题，这个配置参数是否使用，在什么版本中可以使用，也最好还是实际测试以后再决定。

spark.shuffle.spill

shuffle 的过程中，如果涉及到排序，聚合等操作，势必会需要在内存中维护一些数据结构，进而占用额外的内存。如果内存不够用怎么办，那只有两条路可以走，一就是 out of memory 出错了，二就是将部分数据临时写到外部存储设备中去，最后再合并到最终的 Shuffle 输出文件中去。

这里 spark.shuffle.spill 决定是否 Spill 到外部存储设备（默认打开）,如果你的内存足够使用，或者数据集足够小，当然也就不需要 Spill，毕竟 Spill 带来了额外的磁盘操作。

spark.shuffle.memoryFraction / spark.shuffle.safetyFraction

在启用 Spill 的情况下，spark.shuffle.memoryFraction（1.1 后默认为 0.2）决定了当 Shuffle 过程中使用的内存达到总内存多少比例的时候开始 Spill。

通过 spark.shuffle.memoryFraction 可以调整 Spill 的触发条件，即 Shuffle 占用内存的大小，进而调整 Spill 的频率和 GC 的行为。总的来说，如果 Spill 太过频繁，可以适当增加

spark.shuffle.memoryFraction 的大小，增加用于 Shuffle 的内存，减少 Spill 的次数。当然这样一来为了避免内存溢出，对应的可能需要减少 RDD cache 占用的内存，即减小 spark.storage.memoryFraction 的值，这样 RDD cache 的容量减少，有可能带来性能影响，因此需要综合考虑。

由于 Shuffle 数据的大小是估算出来的，一来为了降低开销，并不是每增加一个数据项都完整的估算一次，二来估算也会有误差，所以实际暂用的内存可能比估算值要大，这里 spark.shuffle.safetyFraction（默认为 0.8）用来作为一个保险系数，降低实际 Shuffle 使用的内存阀值，增加一定的缓冲，降低实际内存占用超过用户配置值的概率。

spark.shuffle.spill.compress / spark.shuffle.compress

这两个配置参数都是用来设置 Shuffle 过程中是否使用压缩算法对 Shuffle 数据进行压缩，前者针对 Spill 的中间数据，后者针对最终的 shuffle 输出文件，默认都是 True

理论上说，spark.shuffle.compress 设置为 True 通常都是合理的，因为如果使用千兆以下的网卡，网络带宽往往最容易成为瓶颈。此外，目前的 Spark 任务调度实现中，以 Shuffle 划分 Stage，下一个 Stage 的任务是要等待上一个 Stage 的任务全部完成以后才能开始执行，所以 shuffle 数据的传输和 CPU 计算任务之间通常不会重叠，这样 Shuffle 数据传输量的大小和所需的时间就直接影响到了整个任务的完成速度。但是压缩也是要消耗大量的 CPU 资源的，所以打开压缩选项会增加 Map 任务的执行时间，因此如果在 CPU 负载的影响远大于磁盘和网络带宽的影响的场合下，也可能将 spark.shuffle.compress 设置为 False 才是最佳的方案

对于 spark.shuffle.spill.compress 而言，情况类似，但是 spill 数据不会被发送到网络中，仅仅是临时写入本地磁盘，而且在一个任务中同时需要执行压缩和解压缩两个步骤，所以对 CPU 负载的影响会更大一些，而磁盘带宽（如果标配 12HDD 的话）可能往往不会成为 Spark 应用的主要问题，所以这个参数相对而言，或许更有机会需要设置为 False。

总之，Shuffle 过程中数据是否应该压缩，取决于 CPU/DISK/NETWORK 的实际能力和负载，应该综合考虑。

Storage 相关配置参数

spark.local.dir

Spark 用于写中间数据，如 RDD Cache，Shuffle，Spill 等数据的位置，那么有什么可以注意的呢。

首先，最基本的当然是我们可以配置多个路径（用逗号分隔）到多个磁盘上增加整体 IO 带宽。

其次，目前的实现中，Spark 是通过对文件名采用 hash 算法分布到多个路径下的目录中去，如果你的存储设备有快有慢，比如 SSD+HDD 混合使用，那么你可以通过在 SSD 上配置更

多的目录路径来增大它被 Spark 使用的比例，从而更好地利用 SSD 的 IO 带宽能力。当然这只是一种变通的方法，终极解决方案还是应该像目前 HDFS 的实现方向一样，让 Spark 能够感知具体的存储设备类型，针对性的使用。

需要注意的是，在 Spark 1.0 以后，SPARK_LOCAL_DIRS (Standalone, Mesos) or LOCAL_DIRS (YARN)参数会覆盖这个配置。比如 Spark On YARN 的时候，Spark Executor 的本地路径依赖于 Yarn 的配置，而不取决于这个参数。

spark.executor.memory

Executor 内存的大小，和性能本身当然并没有直接的关系，但是几乎所有运行时性能相关的内容都或多或少间接和内存大小相关。这个参数最终会被设置到 Executor 的 JVM 的 heap 尺寸上，对应的就是 Xmx 和 Xms 的值

理论上 Executor 内存当然是多多益善，但是实际受机器配置，以及运行环境，资源共享，JVM GC 效率等因素的影响，还是有可能需要为它设置一个合理的大小。多大算合理，要看实际情况

Executor 的内存基本上是 Executor 内部所有任务共享的，而每个 Executor 上可以支持的任务的数量取决于 Executor 所管理的 CPU Core 资源的多少，因此你需要了解每个任务的数据规模的大小，从而推算出每个 Executor 大致需要多少内存即可满足基本的需求。

如何知道每个任务所需内存的大小呢，这个很难统一的衡量，因为除了数据集本身的开销，还包括算法所需各种临时内存空间的使用，而根据具体的代码算法等不同，临时内存空间的开销也不同。但是数据集本身的大小，对最终所需内存的大小还是有一定的参考意义的。

通常来说每个分区的数据集在内存中的大小，可能是其在磁盘上源数据大小的若干倍（不考虑源数据压缩，Java 对象相对于原始裸数据也还要算上用于管理数据的数据结构的额外开销），需要准确的知道大小的话，可以将 RDD cache 在内存中，从 BlockManager 的 Log 输出可以看到每个 Cache 分区的大小（其实也是估算出来的，并不完全准确）

如： BlockManagerInfo: Added rdd_0_1 on disk on sr438:41134 (size: 495.3 MB)

反过来说，如果你的 Executor 的数量和内存大小受机器物理配置影响相对固定，那么你就需要合理规划每个分区任务的数据规模，例如采用更多的分区，用增加任务数量（进而需要更多的批次来运算所有的任务）的方式来减小每个任务所需处理的数据大小。

spark.storage.memoryFraction

如前面所说 spark.executor.memory 决定了每个 Executor 可用内存的大小，而 spark.storage.memoryFraction 则决定了在这部分内存中有多少可以用于 Memory Store 管理 RDD Cache 数据，剩下的内存用来保证任务运行时各种其它内存空间的需要。

spark.executor.memoryFraction 默认值为 0.6，官方文档建议这个比值不要超过 JVM Old Gen 区域的比值。这也很容易理解，因为 RDD Cache 数据通常都是长期驻留内存的，理论上也就是说最终会被转移到 Old Gen 区域（如果该 RDD 还没有被删除的话），如果这部分数据允许的尺寸太大，势必把 Old Gen 区域占满，造成频繁的 FULL GC。

如何调整这个比值，取决于你的应用对数据的使用模式和数据的规模，粗略的来说，如果频繁发生 Full GC，可以考虑降低这个比值，这样 RDD Cache 可用的内存空间减少（剩下的部分 Cache 数据就需要通过 Disk Store 写到磁盘上了），会带来一定的性能损失，但是腾出更多的内存空间用于执行任务，减少 Full GC 发生的次数，反而可能改善程序运行的整体性能

spark.streaming.blockInterval

这个参数用来设置 Spark Streaming 里 Stream Receiver 生成 Block 的时间间隔，默认为 200ms。具体的行为表现是具体的 Receiver 所接收的数据，每隔这里设定的时间间隔，就从 Buffer 中生成一个 StreamBlock 放进队列，等待进一步被存储到 BlockManager 中供后续计算过程使用。理论上来说，为了每个 Streaming Batch 间隔里的数据是均匀的，这个时间间隔当然应该能被 Batch 的间隔时间长度所整除。总体来说，如果内存大小够用，Streaming 的数据来得及处理，这个 blockInterval 时间间隔的影响不大，当然，如果数据 Cache Level 是 Memory+Ser，即做了序列化处理，那么 BlockInterval 的大小会影响序列化后数据块的大小，对于 Java 的 GC 的行为会有一些影响。

此外 spark.streaming.blockQueueSize 决定了在 StreamBlock 被存储到 BlockMananger 之前，队列中最多可以容纳多少个 StreamBlock。默认为 10，因为这个队列 Poll 的时间间隔是 100ms，所以如果 CPU 不是特别繁忙的话，基本上应该没有问题。

压缩和序列化相关

spark.serializer

默认为 org.apache.spark.serializer.JavaSerializer，可选 org.apache.spark.serializer.KryoSerializer，实际上只要是 org.apache.spark.serializer 的子类就可以了,不过如果只是应用,大概你不会自己去实现一个的。

序列化对于 spark 应用的性能来说,还是有很大影响的,在特定的数据格式的情况下,KryoSerializer 的性能可以达到 JavaSerializer 的 10 倍以上,当然放到整个 Spark 程序中来考量,比重就没有那么大了,但是以 Wordcount 为例，通常也很容易达到 30%以上的性能提升。而对于一些 Int 之类的基本类型数据，性能的提升就几乎可以忽略了。KryoSerializer 依赖 Twitter 的 Chill 库来实现，相对于 JavaSerializer，主要的问题在于不是所有的 Java Serializable 对象都能支持。

需要注意的是，这里可配的 Serializer 针对的对象是 Shuffle 数据，以及 RDD Cache 等场合，而 Spark Task 的序列化是通过 spark.closure.serializer 来配置，但是目前只支持 JavaSerializer，所以等于没法配置啦。

更多 Kryo 序列化相关优化配置,可以参考 http://spark.apache.org/docs/latest/tuning.html#data-serialization 一节

spark.rdd.compress

这个参数决定了 RDD Cache 的过程中，RDD 数据在序列化之后是否进一步进行压缩再储存到内存或磁盘上。当然是为了进一步减小 Cache 数据的尺寸，对于 Cache 在磁盘上而言，绝对大小大概没有太大关系，主要是考虑 Disk 的 IO 带宽。而对于 Cache 在内存中，那主要就是考虑尺寸的影响，是否能够 Cache 更多的数据，是否能减小 Cache 数据对 GC 造成的压力等。

这两者，前者通常不会是主要问题，尤其是在 RDD Cache 本身的目的就是追求速度，减少重算步骤，用 IO 换 CPU 的情况下。而后者，GC 问题当然是需要考量的，数据量小，占用空间少，GC 的问题大概会减轻，但是是否真的需要走到 RDD Cache 压缩这一步，或许用其它方式来解决可能更加有效。

所以这个值默认是关闭的，但是如果在磁盘 IO 的确成为问题或者 GC 问题真的没有其它更好的解决办法的时候，可以考虑启用 RDD 压缩。

spark.broadcast.compress

是否对 Broadcast 的数据进行压缩，默认值为 True。

Broadcast 机制是用来减少运行每个 Task 时，所需要发送给 TASK 的 RDD 所使用到的相关数据的尺寸，一个 Executor 只需要在第一个 Task 启动时，获得一份 Broadcast 数据，之后的 Task 都从本地的 BlockManager 中获取相关数据。在 1.1 版本以后的代码中，RDD 本身也改为以 Broadcast 的形式发送给 Executor（之前的实现 RDD 本身是随每个任务发送的），因此基本上不太需要显式的决定哪些数据需要 broadcast 了。

因为 Broadcast 的数据需要通过网络发送，而在 Executor 端又需要存储在本地 BlockMananger 中，加上最新的实现，默认 RDD 通过 Boradcast 机制发送，因此大大增加了 Broadcast 变量的比重，所以通过压缩减小尺寸，来减少网络传输开销和内存占用，通常都是有利于提高整体性能的。

什么情况可能不压缩更好呢，大致上个人觉得同样还是在网络带宽和内存不是问题的时候，如果 Driver 端 CPU 资源很成问题（毕竟压缩的动作基本都在 Driver 端执行），那或许有调整的必要。

spark.io.compression.codec

RDD Cache 和 Shuffle 数据压缩所采用的算法 Codec，默认值曾经是使用 LZF 作为默认 Codec，最近因为 LZF 的内存开销的问题，默认的 Codec 已经改为 Snappy。

LZF 和 Snappy 相比较，前者压缩率比较高（当然要看具体数据内容了，通常要高 20%左右），但是除了内存问题以外，CPU 代价也大一些（大概也差 20%~50%？）

在用于 Shuffle 数据的场合下，内存方面，应该主要是在使用 HashShuffleManager 的时候有可能成为问题，因为如果 Reduce 分区数量巨大，需要同时打开大量的压缩数据流用于写文件，进而在 Codec 方面需要大量的 buffer。但是如果使用 SortShuffleManager，由于 shuffle 文件数量大大减少，不会产生大量的压缩数据流，所以内存开销大概不会成为主要问题。

剩下的就是 CPU 和压缩率的权衡取舍，和前面一样，取决于 CPU/网络/磁盘的能力和负载，个人认为 CPU 通常更容易成为瓶颈。所以要调整性能，要不不压缩，要不使用 Snappy 可能性大一些？

对于 RDD Cache 的场合来说，绝大多数场合都是内存操作或者本地 IO，所以 CPU 负载的问题可能比 IO 的问题更加突出，这也是为什么 spark.rdd.compress 本身默认为不压缩，如果要压缩，大概也是 Snappy 合适一些？

schedule 调度相关

大概会是你针对自己的集群第一步就会配置的参数，这里多少就其内部机制做一些解释。

spark.cores.max

一个集群最重要的参数之一，当然就是 CPU 计算资源的数量。spark.cores.max 这个参数决定了在 Standalone 和 Mesos 模式下，一个 Spark 应用程序所能申请的 CPU Core 的数量。如果你没有并发跑多个 Spark 应用程序的需求，那么可以不需要设置这个参数，默认会使用 spark.deploy.defaultCores 的值（而 spark.deploy.defaultCores 的值默认为 Int.Max，也就是不限制的意思）从而应用程序可以使用所有当前可以获得的 CPU 资源。

针对这个参数需要注意的是，这个参数对 Yarn 模式不起作用，YARN 模式下，资源由 Yarn 统一调度管理，一个应用启动时所申请的 CPU 资源的数量由另外两个直接配置 Executor 的数量和每个 Executor 中 core 数量的参数决定。

SPARK_EXECUTOR_INSTANCES/SPARK_EXECUTOR_CORES

--num-executors / --executor-cores

（历史原因造成，不同运行模式下的一些启动参数个人认为还有待进一步整合）

此外，在 Standalone 模式等后台分配 CPU 资源时，目前的实现中，在 spark.cores.max 允许的范围内，基本上是优先从每个 Worker 中申请所能得到的最大数量的 CPU core 给每个 Executor，因此如果人工限制了所申请的 Max Core 的数量小于 Standalone 和 Mesos 模式所管理的 CPU 数量，可能发生应用只运行在集群中部分节点上的情况（因为部分节点

所能提供的最大 CPU 资源数量已经满足应用的要求），而不是平均分布在集群中。通常这不会是太大的问题，但是如果涉及数据本地性的场合，有可能就会带来一定的必须进行远程数据读取的情况发生。理论上，这个问题可以通过两种途径解决：一是 Standalone 和 Mesos 的资源管理模块自动根据节点资源情况，均匀分配和启动 Executor，二是和 Yarn 模式一样，允许用户指定和限制每个 Executor 的 Core 的数量。

spark.task.cpus

这个参数在字面上的意思就是分配给每个任务的 CPU 的数量，默认为 1。实际上，这个参数并不能真的控制每个任务实际运行时所使用的 CPU 的数量，比如你可以通过在任务内部创建额外的工作线程来使用更多的 CPU（至少目前为止，将来任务的执行环境是否能通过 LXC 等技术来控制还不好说）。它所发挥的作用，只是在作业调度时，每分配出一个任务时，对已使用的 CPU 资源进行计数。也就是说只是理论上用来统计资源的使用情况，便于安排调度。因此，如果你期望通过修改这个参数来加快任务的运行，那还是赶紧换个思路吧。这个参数的意义，个人觉得还是在你真的在任务内部自己通过任何手段，占用了更多的 CPU 资源时，让调度行为更加准确的一个辅助手段。

spark.scheduler.mode

这个参数决定了单个 Spark 应用内部调度的时候使用 FIFO 模式还是 Fair 模式。是的，你没有看错，这个参数只管理一个 Spark 应用内部的多个没有依赖关系的 Job 作业的调度策略。

如果你需要的是多个 Spark 应用之间的调度策略，那么在 Standalone 模式下，这取决于每个应用所申请和获得的 CPU 资源的数量（暂时没有获得资源的应用就 Pending 在那里了），基本上就是 FIFO 形式的，谁先申请和获得资源，谁就占用资源直到完成。而在 Yarn 模式下，则多个 Spark 应用间的调度策略由 Yarn 自己的策略配置文件所决定。

那么这个内部的调度逻辑有什么用呢？如果你的 Spark 应用是通过服务的形式，为多个用户提交作业的话，那么可以通过配置 Fair 模式相关参数来调整不同用户作业的调度和资源分配优先级。

spark.locality.wait

spark.locality.wait 和 spark.locality.wait.process，spark.locality.wait.node, spark.locality.wait.rack 这几个参数影响了任务分配时的本地性策略的相关细节。

Spark 中任务的处理需要考虑所涉及的数据的本地性的场合，基本就两种，一是数据的来源是 HadoopRDD; 二是 RDD 的数据来源来自于 RDD Cache（即由 CacheManager 从 BlockManager 中读取，或者 Streaming 数据源 RDD）。其它情况下，如果不涉及 shuffle

操作的 RDD，不构成划分 Stage 和 Task 的基准，不存在判断 Locality 本地性的问题，而如果是 ShuffleRDD，其本地性始终为 No Prefer，因此其实也无所谓 Locality。

在理想的情况下，任务当然是分配在可以从本地读取数据的节点上时（同一个 JVM 内部或同一台物理机器内部）的运行时性能最佳。但是每个任务的执行速度无法准确估计，所以很难在事先获得全局最优的执行策略，当 Spark 应用得到一个计算资源的时候，如果没有可以满足最佳本地性需求的任务可以运行时，是退而求其次，运行一个本地性条件稍差一点的任务呢，还是继续等待下一个可用的计算资源已期望它能更好的匹配任务的本地性呢？

这几个参数一起决定了 Spark 任务调度在得到分配任务时，选择暂时不分配任务，而是等待获得满足进程内部/节点内部/机架内部这样的不同层次的本地性资源的最长等待时间。默认都是 3000 毫秒。

基本上，如果你的任务数量较大和单个任务运行时间比较长的情况下，单个任务是否在数据本地运行，代价区别可能比较显著，如果数据本地性不理想，那么调大这些参数对于性能优化可能会有一定的好处。反之如果等待的代价超过带来的收益，那就不要考虑了。

特别值得注意的是：在处理应用刚启动后提交的第一批任务时，由于当作业调度模块开始工作时，处理任务的 Executors 可能还没有完全注册完毕，因此一部分的任务会被放置到 No Prefer 的队列中，这部分任务的优先级仅次于数据本地性满足 Process 级别的任务，从而被优先分配到非本地节点执行，如果的确没有 Executors 在对应的节点上运行，或者的确是 No Prefer 的任务（如 shuffleRDD），这样做确实是比较优化的选择，但是这里的实际情况只是这部分 Executors 还没来得及注册上而已。这种情况下，即使加大本节中这几个参数的数值也没有帮助。针对这个情况，有一些已经完成的和正在进行中的 PR 通过例如动态调整 No Prefer 队列，监控节点注册比例等等方式试图来给出更加智能的解决方案。不过，你也可以根据自身集群的启动情况，通过在创建 SparkContext 之后，主动 Sleep 几秒的方式来简单的解决这个问题。

spark.speculation

spark.speculation 以及 spark.speculation.interval, spark.speculation.quantile, spark.speculation.multiplier 等参数调整 Speculation 行为的具体细节，Speculation 是在任务调度的时候，如果没有适合当前本地性要求的任务可供运行，将跑得慢的任务在空闲计算资源上再度调度的行为，这些参数调整这些行为的频率和判断指标，默认是不使用 Speculation 的。

通常来说很难正确的判断是否需要 Speculation，能真正发挥 Speculation 用处的场合，往往是某些节点由于运行环境原因，比如 CPU 资源由于某种原因被占用，磁盘损坏导致 IO 缓慢造成任务执行速度异常的情况，当然前提是你的分区任务不存在仅能被执行一次，或者不能同时执行多个拷贝等情况。Speculation 任务参照的指标通常是其它任务的执行时间，而实际的任务可能由于分区数据尺寸不均匀，本来就会有时间差异，加上一定的调度和 IO 的随机性，所以如果一致性指标定得过严，Speculation 可能并不能真的发现问题，反而增加了不必要的任务开销，定得过宽，大概又基本相当于没用。

个人觉得，如果你的集群规模比较大，运行环境复杂，的确可能经常发生执行异常，加上数据分区尺寸差异不大，为了程序运行时间的稳定性，那么可以考虑仔细调整这些参数。否则还是考虑如何排除造成任务执行速度异常的因数比较靠铺一些。