- 触发 Spark 延迟计算的 Actions 算子主要有两类：一类是将分布式计算结果直接落盘的操作，如 DataFrame 的 write、RDD 的 saveAsTextFile 等；另一类是将分布式结果收集到 Driver 端的操作，如 first、take、collect。
- 通过对比查询计划，我们能够明显看到 UDF 与 SQL functions 的区别。Spark SQL 的 Catalyst Optimizer 能够明确感知 SQL functions 每一步在做什么，因此有足够的优化空间。相反，UDF 里面封装的计算逻辑对于 Catalyst Optimizer 来说就是个黑盒子，除了把 UDF 塞到闭包里面去，也没什么其他工作可做的。
- 性能调优：一是先验的专家经验，二是后验的运行时诊断。Spark 的性能调优可以从应用代码和 Spark 配置项这 2 个层面展开。
  - CPU密集型：（解）压缩、（反）序列化、hash、排序
  - 内存密集型：大量的RDD Cache、DataFrame Cache、数据倾斜场景，等等
  - 磁盘密集型：Shuffle
  - 网络密集型：Shuffle

1.RDD

Resilient Distributed Dataset：弹性分布式数据集

RDD 具有 4 大属性，分别是 partitions、partitioner、dependencies 和 compute 属性。正因为有了这 4 大属性的存在，让 RDD 具有分布式和容错性这两大最突出的特性。

基于数据源和转换逻辑，无论 RDD 有什么差池（如节点宕机造成部分数据分片丢失），在 dependencies 属性记录的父 RDD 之上，都可以通过执行 compute 封装的计算逻辑再次得到当前的 RDD。由 dependencies 和 compute 属性提供的容错能力，为 Spark 分布式内存计算的稳定性打下了坚实的基础，这也正是 RDD 命名中 Resilient 的由来。不同的 RDD 通过 dependencies 和 compute 属性链接在一起，逐渐向纵深延展，构建了一张越来越深的有向无环图，也就是我们常说的 DAG。

弹性描述物体在压缩和拉伸之后具备恢复原状的能力。类比RDD，rdd因为依赖和compute，具备了错误恢复能力，把这个性质称之为弹性。

总的来说，RDD 的 4 大属性又可以划分为两类，横向属性和纵向属性。其中，横向属性锚定数据分片实体，并规定了数据分片在分布式集群中如何分布；纵向属性用于在纵深方向构建 DAG，通过提供重构 RDD 的容错能力保障内存计算的稳定性。

属性名	成员类型	属性含义	RDD特性	刻画方向
partitions	变量	RDD的所有数据分片实体	分布式	横向
partitioner	方法	划分数据分片的规则		
dependencies	变量	生成该RDD所依赖的父RDD	容错性	纵向
compute	方法	生成该RDD的计算接口		

关于 RDD 的特性与核心属性，只要你把如下 2 点牢记于心，我相信在不知不觉中你自然会绕过很多性能上的坑：

横向属性 partitions 和 partitioner 锚定数据分片实体，并且规定了数据分片在分布式集群中如何分布；纵向属性 dependencies 和 compute 用于在纵深方向构建 DAG，通过提供重构 RDD 的容错能力保障内存计算的稳定性。

第5大属性：
PreferredLocations（位置优先策略） 移动数据不如移动计算的原则： 原文链接：https://blog.csdn.net/m0_37582129/article/details/100863749
一、数据本地性 spark在任务调度的时候，会检查数据存储的位置，本着“移动数据不如移动计算”的原则，会优先将task任务部署在其将要处理的数据所在的节点上，可以最大程度地减小数据传输带来的网络开销，减少IO操作，称为**spark的数据本地性**。
问题：总会出现数据移动的情况，即excutor没有起在对应的datanode上，数据在其它节点。理想情况：集群的每个节点上都存储全量的block副本，即每个block都在所有节点上存储，这样就完全不用数据传输了，但同时也大大增加了节点的存储压力，失去了分布式存储的意义。（原则上，HDFS的block备份数是3）
二、本地性的分类
Process_local：Excutor进程（task线程）读取缓存在本地节点上的数据
Node_local：读取本地节点硬盘上的数据
Rack_local：读取存储在同一机架上的其它节点中的数据
Any：读取非本地节点上的数据
No_pref：数据本地性无意义，数据存储在Mysql等外部数据源，不在spark集群节点上
数据读取速度依次递减，网络传输开销依次增大！

infoQ的《权力的游戏：Spark调度系统》：https://www.infoq.cn/article/5aOHzQIaXX6NlHriLtSI

2.DAG和流水线：内存计算

在 Spark 中，内存计算有两层含义：第一层含义就是众所周知的分布式数据缓存，第二层含义是 Stage 内的流水线式计算模式。

第一层含义：分布式数据缓存

只有需要频繁访问的数据集才有必要 cache，对于一次性访问的数据集，cache 不但不能提升执行效率，反而会产生额外的性能开销，让结果适得其反。

第二层含义：Stage 内的流水线式计算模式

同一 Stage 内所有算子融合为一个函数，Stage 的输出结果由这个函数一次性作用在输入数据集而产生。

什么是 DAG?

DAG 全称 Direct Acyclic Graph，中文叫有向无环图。顾名思义，DAG 是一种“图”。我们知道，任何一种图都包含两种基本元素：顶点（Vertex）和边（Edge），顶点通常用于表示实体，而边则代表实体间的关系。**在 Spark 的 DAG 中，顶点是一个个 RDD，边则是 RDD 之间通过 dependencies 属性构成的父子关系。**

那 DAG 是怎么生成的呢?

从开发者的视角出发，DAG 的构建是通过在分布式数据集上不停地调用算子来完成的。

Stages 的划分：

简单地说，从开发者构建 DAG，到 DAG 转化的分布式任务在分布式环境中执行，其间会经历如下 4 个阶段：

- 回溯 DAG 并划分 Stages
- 在 Stages 中创建分布式任务
- 分布式任务的分发
- 分布式任务的执行

刚才我们说了，内存计算的第二层含义在 stages 内部，因此这一讲我们只要搞清楚 DAG 是怎么划分 Stages 就够了。

如果用一句话来概括从 DAG 到 Stages 的转化过程，那应该是：**以 Actions 算子为起点，从后向前回溯 DAG，以 Shuffle 操作为边界去划分 Stages。**

Stage 中的内存计算：

MapReduce 提供两类计算抽象，分别是 Map 和 Reduce：Map 抽象允许开发者通过实现 map 接口来定义数据处理逻辑；Reduce 抽象则用于封装数据聚合逻辑。MapReduce 计算模型最大的问题在于，**所有操作之间的数据交换都以磁盘为媒介。**这种频繁的磁盘 I/O 必定会拖累用户应用端到端的执行性能。

我们刚刚把流水线比作内存，这意味着每一个算子计算得到的中间结果都会在内存中缓存一份，以备下一个算子运算，这个过程与开发者在应用代码中滥用 RDD cache 简直如出一辙。如果你曾经也是逢 RDD 便 cache，应该不难想象，采用这种计算模式，Spark 的执行性能不见得比 MapReduce 强多少，尤其是在 Stages 中的算子数量较多的时候。

既然不是简单地把数据和计算挪到内存，那 Stage 内的流水线式计算模式到底长啥样呢？在 Spark 中，流水线计算模式指的是：**在同一 Stage 内部，所有算子融合为一个函数，Stage 的输出结果由这个函数一次性作用在输入数据集而产生。**这也正是内存计算的第二层含义。

因此你看，**所谓内存计算，不仅仅是指数据可以缓存在内存中，更重要的是让我们明白了，通过计算的融合来大幅提升数据在内存中的转换效率，进而从整体上提升应用的执行性能。**
由于计算的融合只发生在 Stages 内部，而 Shuffle 是切割 Stages 的边界，因此一旦发生 Shuffle，内存计算的代码融合就会中断。但是，当我们对内存计算有了多方位理解以后，就不会一股脑地只想到用 cache 去提升应用的执行性能，而是会更主动地想办法尽量避免 Shuffle，让应用代码中尽可能多的部分融合为一个函数，从而提升计算效率。

DAG 以 Shuffle 为边界划分 Stages，那你知道 Spark 是根据什么来判断一个操作是否会引入 Shuffle 的呢?

- rdd 会有 dep 属性，用来区分是否是 shuffle 生成的 rdd. 而 dep 属性的确定主要是根据子 rdd 是否依赖父 rdd 的某一部分数据，这个就得看他两的分区器(如果 tranf/action 有的话)。如果分区器一致，就不会产生 shuffle。
- 两个RDD之间的依赖分为宽依赖和窄依赖，窄依赖即父RDD的Partition数据对子RDD的Partition数据的关系是多对一或者一对一的，宽依赖即父RDD的Partition对子RDD的关系是一对多的，所以宽依赖存在数据分发的情况，会引入shuffle；spark处理rdd时会判断，出现宽依赖的时候会停止压入rdd到本stage，新建stage，开始压栈。

在 Spark 中，同一 Stage 内的所有算子会融合为一个函数。你知道这一步是怎么做到的吗?

- 单就rdd算子来说，在同一个stage内部，spark不会真的去创建、合成一个函数链，而是通过不同rdd算子Iterator的嵌套，在逻辑上，形成一个函数链。这里我们说“捏合”，坦白说不够严谨，不过重点在于表达“内存计算”的第二层含义。在后面要讲的tungsten，它是真的真的会把一个stage内部的code，在运行时on the fly地重构出一份新的代码出来。这两者有本质的不同。
- WSCG很好，不过，这个是只有Spark SQL才能享受到的特性，也就是当你使用DataFrame、Dataset或是SQL进行开发的时候，才能享受到这个特性。对于纯粹的RDD API来说，所谓的“捏合”，其实是一种伪“捏合”，它是通过同一个Stage内部多个RDD算子compute函数嵌套的方式，来完成“捏合”。

spark shuffle前后的分区数是如何计算的?

- Map阶段的并行度，会沿用父RDD的并行度，比如沿用HadoopRDD的并行度，这样的话，就是源文件原始的分片数量。Reduce阶段，可以通过repartition来调整，如果没有调整，默认按照spark.sql.shuffle.partitions来走。

spark里cache的正确姿势是什么？是直接df.cache()还是val cacheDf = df.cache()呢？另外不管cache还是persist都是lazy的，所以有必要紧接着一句df.count()让它马上执行吗？因为这样会平白无故多一个job，不知道是不是画蛇添足了?

- df.cache() df.count
- 或是
- val cacheDf = df.cache()
- cacheDf.count
- 都可以，action是必需的，没有action，不会触发缓存的计算和存储，这可不是画蛇添足哈~

说个最极端的情况，如果对一个dataframe Read以后做了一堆不会触发shuffle 的操作，最后又调用了一下coalesce(1)，然后write ，那是不是就意味着从读数据开始的所有操作都会在一个executor上完成?

- 取决于你如何调用coalesce(1, shuffle = false/true)，分两种情况。
- 1. shuffle = false，就像你说的，所有操作，从一开始，并行度都是1，都在一个executor计算，显然，这个时候，整个作业非常慢，奇慢无比
- 2. shuffle = true，这个时候，coalesce就会引入shuffle，切割stage。coalesce之前，用源数据DataFrame的并行度，这个时候是多个Executors真正的并行计算；coalesce之后，也就是shuffle之后，并行度下降为1，所有父RDD的分区，全部shuffle到一个executor，交给一个task去计算。显然，相比前一种，这种实现在执行效率上，更好一些。因此，如果业务应用必须要这么做，推荐这一种实现方法。

3.调度系统-数据不动代码动

在日常的开发与调优工作中，为了充分利用硬件资源，我们往往需要手工调节任务并行度来提升 CPU 利用率，控制任务并行度的参数是 Spark 的配置项：spark.default.parallelism。增加并行度确实能够充分利用闲置的 CPU 线程，但是，parallelism 数值也不宜过大，过大反而会引入过多的调度开销，得不偿失。

Spark 的调度系统是如何工作的?

Spark 调度系统的核心职责是，**先将用户构建的 DAG 转化为分布式任务，结合分布式集群资源的可用性，基于调度规则依序把分布式任务分发到执行器。**

事实上，Spark 调度系统的工作流程包含如下 5 个步骤：

- 将 DAG 拆分为不同的运行阶段 Stages；
- 创建分布式任务 Tasks 和任务组 TaskSet；
- 获取集群内可用的硬件资源情况；
- 按照调度规则决定优先调度哪些任务 / 组；
- 依序将分布式任务分发到执行器 Executor。

调度系统中的核心组件有哪些?

Spark 调度系统包含 3 个核心组件，分别是 **DAGScheduler**、**TaskScheduler** 和 **SchedulerBackend**。这 3 个组件都运行在 **Driver 进程**中，它们通力合作将用户构建的 DAG 转化为分布式任务，再把这些任务分发给集群中的 Executors 去执行。

TaskScheduler：

充当中介的角色，TaskScheduler 的调度策略分为两个层次，一个是不同 Stages 之间的调度优先级，一个是 Stages 内不同任务之间的调度优先级。

- Stages 之间的任务调度，TaskScheduler 提供了 2 种调度模式，分别是 FIFO（先到先得）和 FAIR（公平调度）。
- 同一个 Stages 内部不同任务之间的调度，TaskScheduler 会优先挑选那些满足本地性级别要求的任务进行分发。

Spark 调度系统的原则是尽可能地让数据呆在原地、保持不动，同时尽可能地把承载计算任务的代码分发到离数据最近的地方，从而最大限度地降低分布式系统中的网络开销。
毕竟，分发代码的开销要比分发数据的代价低太多，这也正是“数据不动代码动”这个说法的由来。
总的来说，TaskScheduler 根据本地性级别遴选出待计算任务之后，先对这些任务进行序列化。然后，交给 SchedulerBackend，SchedulerBackend 根据 ExecutorData 中记录的 RPC 地址和主机地址，再将序列化的任务通过网络分发到目的主机的 Executor 中去。最后，Executor 接收到任务之后，把任务交由内置的线程池，线程池中的多线程则并发地在不同数据分片之上执行任务中封装的数据处理函数，从而实现分布式计算。

性能调优案例回顾
方式1：

```
/**
实现方式1
输入参数：模板文件路径，用户兴趣字符串
返回值：用户兴趣字符串对应的索引值
*/

//函数定义
def findIndex(templatePath: String, interest: String): Int = {
val source = Source.fromFile(filePath, "UTF-8")
val lines = source.getLines().toArray
source.close()
val searchMap = lines.zip(0 until lines.size).toMap
searchMap.getOrElse(interest, -1)
}

//Dataset中的函数调用
findIndex(filePath, "体育-篮球-NBA-湖人")
```

方式2：

```
/**
实现方式2
输入参数：模板文件路径，用户兴趣字符串
返回值：用户兴趣字符串对应的索引值
*/

//函数定义
val findIndex: (String) => (String) => Int = {
(filePath) =>
val source = Source.fromFile(filePath, "UTF-8")
val lines = source.getLines().toArray
source.close()
val searchMap = lines.zip(0 until lines.size).toMap
(interest) => searchMap.getOrElse(interest, -1)
}
val partFunc = findIndex(filePath)

//Dataset中的函数调用
partFunc("体育-篮球-NBA-湖人")
```

- 2 种实现方式的本质区别在于，函数中 2 个计算步骤的分布式计算过程不同。在第 1 种实现方式中，函数是一个接收两个形参的普通标量函数，Dataset 调用这个函数在千亿级样本上做 Label encoding。
- 在 Spark 任务调度流程中，该函数在 Driver 端交由 DAGScheduler 打包为 Tasks，经过 TaskScheduler 调度给 SchedulerBackend，最后由 SchedulerBackend 分发到集群中的 Executors 中去执行。这意味着集群中的每一个 Executors 都需要执行函数中封装的两个计算步骤，要知道，第一个步骤中遍历文件内容并建立字典的计算开销还是相当大的。
- 反观第 2 种实现方式，2 个计算步骤被封装到一个高阶函数中。用户代码先在 Driver 端用模板文件调用这个高阶函数，完成第一步计算建立字典的过程，同时输出一个只带一个形参的标量函数，这个标量函数内携带了刚刚建好的映射字典。最后，Dataset 将这个标量函数作用于千亿样本之上做 Label encoding。
- 发现区别了吗？在第 2 种实现中，函数的第一步计算只在 Driver 端计算一次，分发给集群中所有 Executors 的任务中封装的是携带了字典的标量函数。然后在 Executors 端，Executors 在各自的数据分片上调用该函数，省去了扫描模板文件、建立字典的开销。最后，我们只需要把样本中的用户兴趣传递进去，函数就能以 O(1) 的查询效率返回数值结果。

- 对于一个有着成百上千 Executors 的分布式集群来说，这 2 种不同的实现方式带来的性能差异还是相当可观的。因此，如果你能把 Spark 调度系统的工作原理牢记于心，我相信在代码开发或是 review 的过程中，你都能够意识到第一个计算步骤会带来的性能问题。这种开发过程中的反思，其实就是在潜移默化地建立以性能为导向的开发习惯。

小结：

结合这 5 个步骤，我们深入分析了 Spark 调度系统的工作原理，我们可以从核心职责和核心原则这两方面来归纳：

- Spark 调度系统的核心职责是，**先将用户构建的 DAG 转化为分布式任务，结合分布式集群资源的可用性，基于调度规则依序把分布式任务分发到执行器 Executors；**
- Spark 调度系统的核心原则是，**尽可能地让数据呆在原地、保持不动，同时尽可能地把承载计算任务的代码分发到离数据最近的地方（Executors 或计算节点），从而最大限度地降低分布式系统中的网络开销。**

数据尽量不动，比如有部分数据在节点A，那么移动计算难道不是要在节点A上启动Executor才可以进行计算吗？但是Executor不是在申请资源的时候就确定了在哪几个节点上启动Executor吗？

- 资源调度和任务调度是分开的。
- 资源调度主要看哪些节点可以启动executors，是否能满足executors所需的cpu数量要求，这个时候，不会考虑任务、数据本地性这些因素。资源调度完成之后，在任务调度阶段，spark负责计算每个任务的本地性，效果就是task明确知道自己应该调度到哪个节点，甚至是哪个executors。最后scheduler Backend会把task代码，分发到目标节点的目标executors，完成任务调度，实现数据不动代码动。
- 所以，二者是独立的，不能混为一谈

任务调度的时候不考虑可用内存大小吗？

- 我们分资源调度和任务调度两种情况来说。
- Spark在做任务调度之前，SchedulerBackend封装的调度器，比如Yarn、Mesos、Standalone，实际上已经完成了资源调度，换句话说，整个集群有多少个containers/executors，已经是一件确定的事情了。而且，每个Executors的CPU和内存，也都是确定的了（因为你启动Spark集群的时候，使用配置项指定了每个Executors的CPU和内存分别是多少）。资源调度器在做资源调度的时候，确实是同时需要CPU和内存信息的。
- 资源调度完成后，Spark开始任务调度，你的问题，其实是任务调度范畴的问题。也就是TaskScheduler在准备调度任务的时候，要事先知道都有哪些Executors可用，注意，是可用。也就是TaskScheduler的核心目的，在于获取“可用”的Executors。
- 现在来回答你的问题，也就是：为什么ExecutorData不存储于内存相关的信息。答案是：不需要。一来，TaskScheduler要达到目的，它只需知道Executors是否有空闲CPU、有几个空闲CPU就可以了，有这些信息就足以让他决定是否把tasks调度到目标Executors上去。二来，每个Executors的内存总大小，在Spark集群启动的时候就确定了，因此，ExecutorData自然是没必要记录像Total Memory这样的冗余信息。

- 再来说Free Memory，首先，我们说过，Spark对于内存的预估不准，再者，每个Executors的可用内存都会随着GC的执行而动态变化，因此，ExecutorData记录的Free Memory，永远都是过时的信息，TaskScheduler拿到这样的信息，也没啥用。一者是不准，二来确实没用，因为TaskScheduler拿不到数据分片大小这样的信息，TaskScheduler在Driver端，而数据分片是在目标Executors，所以TaskScheduler拿到Free Memory也没啥用，因为它也不能判断说：task要处理的数据分片，是不是超过了目标Executors的可用内存。
- 综上，ExecutorData的数据结构中，只保存了CPU信息，而没有记录内存消耗等信息。

第二种用部分函数的例子里，是节约了哪步操作呢？读文件应该只要Driver读一次就够了。但是zipWithIndex生成的map呢，由于没有把它广播出去，那应该还是每个task都会被拷贝一份全量的map吧，感觉性能提升也不应该那么明显？

- 由于map没有用广播，所以每个task都会携带这个map，有额外的网络和内存存储开销。
- 但是，你要看跟谁比，跟广播比，第二种写法确实不如广播，但如果你跟第一种实现比，性能提升会非常明显。在第一种实现下，task不会携带map，而是在Executor临时去读文件、临时创建那个map，这个重复的计算开销，远大于task分发携带map带来的网络和内存开销。
- 简言之， 你说的非常对，不过咱们这一讲要强调的关键是调度系统，因此后来并没有用广播进一步优化，讲道理来说，一定是广播的实现方式是最优的。

4.存储系统

Spark 存储系统是为谁服务的?

Spark 存储系统用于存储 3 个方面的数据，分别是 **RDD 缓存、Shuffle 中间文件、广播变量**。

RDD缓存：

RDD 缓存指的是将 RDD 以缓存的形式物化到内存或磁盘的过程。对于一些计算成本和访问频率都比较高的 RDD 来说，缓存有两个好处：

- 一是通过截断 DAG，可以降低失败重试的计算开销；
- 二是通过对缓存内容的访问，可以有效减少从头计算的次数，从整体上提升作业端到端的执行性能。

Shuffle中间文件：

Shuffle 的计算过程可以分为 2 个阶段：

- Map 阶段：Shuffle writer 按照 Reducer 的分区规则将中间数据写入本地磁盘；
- Reduce 阶段：Shuffle reader 从各个节点下载数据分片，并根据需要进行聚合计算。

Shuffle 中间文件实际上就是 Shuffle Map 阶段的输出结果，这些结果会以文件的形式暂存于本地磁盘。在 Shuffle Reduce 阶段，Reducer 通过网络拉取这些中间文件用于聚合计算，如求和、计数等。在集群范围内，Reducer 想要拉取属于自己的那部分中间数据，就必须要知道这些数据都存储在哪些节点，以及什么位置。而这些关键的元信息，正是由 Spark 存储系统保存并维护的。因此你看，**没有存储系统，Shuffle 是玩不转的**。

广播变量：

广播变量往往用于在集群范围内分发访问频率较高的小数据。利用存储系统，**广播变量可以在 Executors 进程范畴内保存全量数据**。这样一来，对于同一 Executors 内的所有计算任务，应用就能够以 Process local 的本地性级别，来共享广播变量中携带的全量数据了。

存储系统的基本组件有哪些?

Spark 存储系统是一个囊括了众多组件的复合系统，如 BlockManager、BlockManagerMaster、MemoryStore、DiskStore 和 DiskBlockManager 等等。不过，家有千口、主事一人，**BlockManager 是其中最为重要的组件，它在 Executors 端负责统一管理和协调数据的本地存取与跨节点传输。**

- 对外，BlockManager 与 Driver 端的 BlockManagerMaster 通信，不仅定期向 BlockManagerMaster 汇报本地数据元信息，还会不定时按需拉取全局数据存储状态。另外，不同 Executors 的 BlockManager 之间也会以 Server/Client 模式跨节点推送和拉取数据块。
- 对内，BlockManager 通过组合存储系统内部组件的功能来实现数据的存与取、收与发。

BlockManager服务于RDD 缓存、Shuffle 中间文件和广播变量这 3 个服务对象。

Spark 存储系统提供了两种存储抽象：**MemoryStore 和 DiskStore。BlockManager 正是利用它们来分别管理数据在内存和磁盘中的存取。**

- 广播变量的全量数据存储在 Executors 进程中，因此它由 MemoryStore 管理。
- Shuffle 中间文件往往会落盘到本地节点，所以这些文件的落盘和访问就要经由 DiskStore。
- RDD 缓存会稍微复杂一些，由于 RDD 缓存支持内存缓存和磁盘缓存两种模式，因此我们要视情况而定，缓存在内存中的数据会封装到 MemoryStore，缓存在磁盘上的数据则交由 DiskStore 管理。

有了 MemoryStore 和 DiskStore，我们暂时解决了数据“存在哪儿”的问题。但是，这些数据该以“什么形式”存储到 MemoryStore 和 DiskStore 呢？对于数据的存储形式，**Spark 存储系统支持两种类型：对象值（Object Values）和字节数组（Byte Array）**。

- 对象值压缩为字节数组的过程叫做序列化；
- 字节数组还原成原始对象值的过程就叫做反序列化。

由此可见，**对象值和字节数组二者之间存在着一种博弈关系**，也就是所谓的“以空间换时间”和“以时间换空间”，两者之间该如何取舍，我们还是要看具体的应用场景。**核心原则就是：如果想省地儿，你可以优先考虑字节数组；如果想以最快的速度访问对象，还是对象值更直接一些。** 不过，这种选择的烦恼只存在于 MemoryStore 之中，而 DiskStore 只能存储序列化后的字节数组，毕竟，凡是落盘的东西，都需要先进行序列化。

透过 RDD 缓存看 MemoryStore：

MemoryStore 同时支持存储对象值和字节数组这两种不同的数据形式，并且统一采用 MemoryEntry 数据抽象对它们进行封装。

在 RDD 的语境下，我们往往用数据分片（Partitions/Splits）来表示一份分布式数据，但在存储系统的语境下，我们经常会用数据块（Blocks）来表示数据存储的基本单元。**在逻辑关系上，RDD 的数据分片与存储系统的 Block 一一对应，也就是说一个 RDD 数据分片会被物化成一个内存或磁盘上的 Block。**

总的来说，RDD 数据分片、Block 和 MemoryEntry 三者之间是一一对应的，当所有的 RDD 数据分片都物化为 MemoryEntry，并且所有的（Block ID, MemoryEntry）对都记录到 **LinkedHashMap** 字典之后，RDD 就完成了数据缓存到内存的过程。

- 当storage memory不足，spark需要删除rdd cache的时候，遵循的是lru，那么问题来了，它咋实现的lru，答案就是它充分利用LinkedHashMap。

MemoryStore为什么没有使用普通的HashMap，而是采用了更加复杂的LinkedHashMap来维护缓存列表?

> LinkedHashMap也是一种HashMap，它的特别之处在于，其在内部用一个双向链表来维护键值对的顺序，每个键值对同时存储在哈希表、和双向链表中。
>
> **我们来看LinkedHashMap的特性：**
>
> 插入有序，这个好理解，就是在链表后追加元素
>
> 访问有序，这个就厉害了。访问有序说的是，对一个kv操作，不管put还是get，元素都会被挪到链表的末尾
>
> 在Spark RDD Cache的场景下，第一个特性不重要，重要的是第二个特性。当Storage Memory不足，Spark需要删除RDD Cache的时候，遵循的是LRU。那么问题来了，Spark是怎么实现的LRU的呢？答案就是它充分利用LinkedHashMap第二个特性，啥也不用做，就轻松地做到了这一点。根据LinkedHashMap访问有序的特性，最近访问过的元素都会依次挪到链表末尾，那么从链表表头开始，依次往后遍历，就都是那些“最近最少访问”的倒霉蛋。因此，当需要删除RDD Cache Block的时候，只需要从前往后依次删掉链表元素及其对应的Block就好了。

透过 Shuffle 看 DiskStore：

DiskStore 中数据的存取本质上就是字节序列与磁盘文件之间的转换，它通过 putBytes 方法把字节序列存入磁盘文件，再通过 getBytes 方法将文件内容转换为数据块。

MemoryStore 采用链式哈希字典来维护类似的元数据，DiskStore 这个狡猾的家伙并没有亲自维护这些元数据，而是请了 DiskBlockManager 这个给力的帮手。

- DiskBlockManager 的主要职责就是，记录逻辑数据块 Block 与磁盘文件系统中物理文件的对应关系，每个 Block 都对应一个磁盘文件。同理，每个磁盘文件都有一个与之对应的 Block ID，这就好比货架上的每一件货物都有唯一的 ID 标识。

- Spark 默认采用 SortShuffleManager 来管理 Stages 间的数据分发，在 Shuffle write 过程中，有 3 类结果文件：temp_shuffle_XXX、shuffle_XXX.data 和 shuffle_XXX.index。Data 文件存储分区数据，它是由 temp 文件合并而来的，而 index 文件记录 data 文件内不同分区的偏移地址。Shuffle 中间文件具体指的就是 data 文件和 index 文件，temp 文件作为暂存盘文件最终会被删除。
- 在 Shuffle write 的不同阶段，Shuffle manager 通过 BlockManager 调用 DiskStore 的 putBytes 方法将数据块写入文件。文件由 DiskBlockManager 创建，文件名就是 putBytes 方法中的 Block ID，这些文件会以“temp_shuffle”或“shuffle”开头，保存在 spark.local.dir 目录下的子目录里。
- 在 Shuffle read 阶段，Shuffle manager 再次通过 BlockManager 调用 DiskStore 的 getBytes 方法，读取 data 文件和 index 文件，将文件内容转化为数据块，最终这些数据块会通过网络分发到 Reducer 端进行聚合计算。

spark 做shuffle的时候，shuffle write 要写入磁盘，是否可以直接通过内存传输?

- 可以的。Shuffle write的中间文件，都会落盘到spark.local.dir这个配置项指定的文件目录，如果内存足够大的话，可以用Ramdisk开辟一块内存，用来当作是文件系统。
- 然后，把spark.local.dir指向到这里，那么中间文件实际上就是暂存到了内存中，这样就能实现完全的“内存计算”，也就是计算的整个过程，从头到尾，都是在内存里面完成。

5.spark内存管理

内存管理模式

在管理方式上，Spark 会区分堆内内存（On-heap Memory）和堆外内存（Off-heap Memory）。这里的“堆”指的是 JVM Heap，因此堆内内存实际上就是 Executor JVM 的堆内存；堆外内存指的是通过 Java Unsafe API，像 C++ 那样直接从操作系统中申请和释放内存空间。

堆内内存：

其中，**堆内内存的申请与释放统一由 JVM 代劳**。比如说，Spark 需要内存来实例化对象，JVM 负责从堆内分配空间并创建对象，然后把对象的引用返回，最后由 Spark 保存引用，同时记录内存消耗。反过来也是一样，Spark 申请删除对象会同时记录可用内存，JVM 负责把这样的对象标记为“待删除”，然后再通过垃圾回收（Garbage Collection，GC）机制将对象清除并真正释放内存。

在这样的管理模式下，**Spark 对内存的释放是有延迟的**，因此，当 Spark 尝试估算当前可用内存时，很有可能会高估堆内的可用内存空间。Spark 对堆内内存占用的预估往往不够精确，高估可用内存往往会为 OOM 埋下隐患。

堆外内存：

堆外内存则不同，Spark 通过调用 Unsafe 的 allocateMemory 和 freeMemory 方法直接在操作系统内存中申请、释放内存空间，这听上去是不是和 C++ 管理内存的方式很像呢？这样的内存管理方式自然不再需要垃圾回收机制，也就免去了它带来的频繁扫描和回收引入的性能开销。更重要的是，空间的申请与释放可以精确计算，因此 Spark 对堆外可用内存的估算会更精确，对内存的利用率也更有把握。

内存区域的划分

堆内内存：
堆内内存的划分方式和堆外差不多，Spark 也会划分出用于执行和缓存的两份内存空间。
不仅如此，Spark 在堆内还会划分出一片叫做 User Memory 的内存空间，它用于存储开发者自定义数据结构。
除此之外，Spark 在堆内还会预留出一小部分内存空间，叫做 Reserved Memory，它被用来存储各种 Spark 内部对象，例如存储系统中的 BlockManager、DiskBlockManager 等等。

堆外内存：
Spark 把堆外内存划分为两块区域：一块用于执行分布式任务，如 Shuffle、Sort 和 Aggregate 等操作，这部分内存叫做 Execution Memory；一块用于缓存 RDD 和广播变量等数据，它被称为 Storage Memory。

对于性能调优来说，我们在前三块内存的利用率上有比较大的发挥空间，因为业务应用主要消耗的就是它们，也即 Execution memory、Storage memory 和 User memory。而预留内存我们却动不得，因为这块内存仅服务于 Spark 内部对象，业务应用不会染指。

堆内内存中：保留内存300M，用户内存为20*0.2=4GB，Storage内存为20*0.8*0.6=9.6GB，Execution内存为20*0.8*0.4=6.4GB
堆外内存中：Storage内存为10*0.6=6G，Execution内存为10*0.4=4G

执行与缓存内存
在所有的内存区域中，最重要的无疑是缓存内存和执行内存，而内存计算的两层含义也就是数据集缓存和 Stage 内的流水线计算，对应的就是 Storage Memory 和 Execution Memory。
在 Spark 1.6 版本之前，Execution Memory 和 Storage Memory 内存区域的空间划分是静态的，一旦空间划分完毕，不同内存区域的用途就固定了。也就是说，即便你没有缓存任何 RDD 或是广播变量，Storage Memory 区域的空闲内存也不能用来执行 Shuffle 中的映射、排序或聚合等操作，因此宝贵的内存资源就被这么白白地浪费掉了。在 1.6 版本之后，**Spark 推出了统一内存管理模式。统一内存管理指的是 Execution Memory 和 Storage Memory 之间可以相互转化**，尽管两个区域由配置项 spark.memory.storageFraction 划定了初始大小，但在运行时，结合任务负载的实际情况，Storage Memory 区域可能被用于任务执行（如 Shuffle），Execution Memory 区域也有可能存储 RDD 缓存。执行任务相比缓存任务，在内存抢占上有着更高的优先级，执行任务主要分为两类：**一类是 Shuffle Map 阶段的数据转换、映射、排序、聚合、归并等操作；另一类是 Shuffle Reduce 阶段的数据排序和聚合操作。它们所涉及的数据结构，都需要消耗执行内存。**

抢占规则：

- 如果对方的内存空间有空闲，双方就都可以抢占；
- 对于 RDD 缓存任务抢占的执行内存，当执行任务有内存需要时，RDD 缓存任务必须立即归还抢占的内存，涉及的 RDD 缓存数据要么落盘、要么清除；
- 对于分布式计算任务抢占的 Storage Memory 内存空间，即便 RDD 缓存任务有收回内存的需要，也要等到任务执行完毕才能释放。

开启堆外内存后，分配的内存空间是多大？这时候还会分配堆内内存吗？

- 具体大小可以通过参数来配置哈，堆内也一样，都是用参数开调控。不过需要注意，堆内、堆外的内存，互相之间不共享。也就是一开始你的task用off heap，后来用着用着发现不够了，这个时候是不能去占有堆内内存的，所以即便堆内有空闲，也还是会oom。所以在划分堆内堆外之前，要提前计划好，如果怕麻烦，就都用堆内。tungsten对于堆内的内存管理做的也很好，大多数场景都问题不大
- 开启堆外之后，执行任务默认会走堆外，堆外用尽了，后续的任务才会走堆内。对于缓存来说，如果你明确指定了用off heap，那就是明确走堆外，如果你不明确指定，那么默认走堆内。

堆外内存存在的意义是什么，有什么场景是一定需要堆外内存么？

- spark官方建议谨慎使用堆外内存，为啥呢？
- 原因其实很简单，在于堆外堆内的空间互不share，也就是说，你的task最开始用堆外，用着用着发现不够了，这个时候即使堆内还有空闲，task也没法用，所以照样会oom。
- 内存本来就有限，再强行划分出两块隔离的区域，其实反而增加了管理难度。tungsten在堆内其实也用内存页管理内存（Tungsten的相关优化，可以参考后面Tungsten那一讲），也用压缩的二进制数据结构，因此gc效率往往可以保障，这也是为什么官方推荐就用堆内就可以了。
- 回答你的问题，我不觉得有什么场景一定要用堆外，就我看来，对于开发者来说，堆外更多地是一种备选项，是Optional的。不过，尽管如此，我们还是要知道堆外、堆内各自有哪些优缺点、优劣势，这样在结合应用场景做选择的时候，也能有的放矢~

想利用堆外让spark去管理数据、加速执行效率，只有off heap那两个参数，一个用来enable(spark.memory.offHeap.enabled=true)、一个指定大小(spark.memory.offHeap.size)。这两个才是正儿八经的off heap key configs。