面试加强相关的内容点

1. zookeeper阶段

- 1- zookeeper中有那些类型的节点, 各个节点有什么特点

```
有四种类型: 永久节点  临时节点  永久顺序节点(永久序列化节点) 临时顺序节点(临时序列化节点)

永久节点:  只有客户端显示执行删除操作的时候, 才会被删除, 不允许出现同名节点, 可以拥有子节点
临时节点:  会话结束, 节点自动删除, 不允许创建子节点, 不允许同名节点

顺序(序列化): 在节点名称后面会自动追加一个不断增长的序列号, 可以创建同名的节点
```

- 2- zookeeper集群是如何进行选举

```
初始集群的启动: 根据节点依次启动,每个节点都有一个myid, 默认各个zookeeper会将票投给id最大的, 同时过半即产生leader

当集群第二次启动, 或者主节点出现宕机后, 如何选举: 各个节点依次启动, 首先会将票投给那个事务id最大的节点,如果id都一致,那么就会将票投给id最大的, 同时过半即产生leader


事务ID(ZID): 当想zookeeper写入数据的时候, 每一次写入, 事务id就会不断的增大, 当各个节点的数据不统一的时候, 往往是那个事务id最大的其数据完整性也是最好的
```

- 3- zookeeper的watch监听机制

```
zookeeper的watch的监听机制, 可以对zookeeper中各个节点进行监听工作, 包括节点的创建 删除 修改 查询 都是可以监听到的
对应
那么在实际使用中, 主要是基于这个watch监听机制, 来实现对其他集群的管理工作, 例如当其他的集群某个节点启动了, 那么就会自动在zookeeper上创建一个临时节点, 此时通过watch监听机制, 即可监听到有节点被创建了, 就能感知到对应的集群中某个节点启动了


zookeeper中watch的监听机制是一次性, 一旦触发就没有了, 需要重新设置监听
```

2. HDFS阶段

- 1- HADOOP集群的架构, 架构中各个节点的作用是什么?

```
hadoop是一个海量数据存储和计算的平台, hadoop中主要包含了有:  HDFS  YARN  MR

HDFS: 分布式存储框架
    NameNode:  HDFS的主节点  接收用户的读写请求, 管理各个从节点, 管理HDFS的元数据
    DataNode:  HDFS的从节点, 主要负责 存储实际的数据, 和NameNode建立心跳,定时向主节点汇报各个管理的块信息, 以及汇报当前运行的状态信息
    secondaryNameNode: HDFS的辅助节点, 辅助NameNode进行元数据的管理工作, 如果是高可用的架构, 此节点就变更为 journalNode集群节点

```

```
Yarn: 是一个统一的资源管理平台
    ResourceManager: Yarn的主节点  负责Yarn集群整体资源的分配管理,底层是基于调度器 接收任务的请求, 负责启动AppMaster

    AppMaster(applicationMaster): 不属于一个节点 仅仅属于一个进程, 一个任务就会有一个AppMaster节点 负责资源的申请, 任务的分配 , 与任务相关的管理工作

    NodeManager: Yarn的从节点 负责执行AppMaster分配过来的任务,向主节点汇报自己的资源信息


MR: 分布式计算的引擎  (分而治之的思想)
    MapTask:  负责分的操作  负责分布式并行计算操作
    reduceTask: 负责合的操作 负责聚合统计处理操作
```

- 2- HDFS的三大机制: 心跳机制 负载均衡机制 副本机制

```
心跳机制:  DataNode默认向NameNode每隔3秒钟汇报一次, 主要汇报的内容有二个部分: DataNode的状态信息 和DataNode存储的所有块信息, 如果DataNode在某一时刻没有汇报信息, 可能被NameNode认为已经宕机了, 一般都是先进入假死状态, 如果连续超过10次都没有收到心跳包, 才会认为是真的宕机了

负载均衡机制: 主要是用于保证各个DataNode中的block块的数量大致相等的, 同时nameNode还会将数据优先保存在存储余量比较大的DataNode节点上

副本机制: Block块可以有多个副本, 放置数据丢失, 提高数据的可靠性, 默认为3个副本
    机架感知原理:
        第一个副本放置在离当前客户端最近的一个机架上的某一个服务器中
        第二个副本放置在同第一个副本机架的另一个机器上
        第三个副本放置在另一个机架的某台机器上
```

- 3- HDFS的数据读写流程

```
写入数据的流程:
* 1- 客户端向NameNode发起写入数据的请求, NameNode接收到写入请求后, 首先会判断对应目录下是否存在要写入的文件, 如果有理解报错, 说写入的文件以存在, 如果不存在, 还会校验是否有写入数据权限, 如果没有, 直接报错, 报权限不足, 如果既不存在, 也有权限, 就会给客户端返回可以写入

* 2- 客户端开始对要写入的文件进行切割操作, 默认是按照128M一个快来进行切割, 切割完后, 拿着第一个Block块再次请求NameNode, 询问应该放置到那个位置下

* 3- NameNode接收到请求后, 会根据网络拓扑关系 机架感知原理  副本机制 从所有的DataNode选择一些合适的DataNode的节点地址返回给客户端

* 4- DataNode接收到NameNode返回的DataNode节点列表后, 首先会先连接列表中第一个DataNode, 然后让第一个DataNode连接第二次, 第二个连接第三次, 以此类推, 形成一个数据传输管道, 同时反向还会构建一条ACK确认管道

* 5- 客户端开始进行写入数据, 数据通过数据包的形式来进行传输, 每一个数据包为64kb大小, 当第一个DataNode接收到数据后, 然后传递给第二个, 第二个传递给第三个 完成数据保存同时每个节点接收到, 还是在ACK中进行确认操作

* 6- 当客户端发现各个节点都收到消息后, 开始传入下一个数据包, 依次类推, 知道将这个block所有的数据全部写完
```

```
* 7- 接着会拿着第二次block再次请求NameNode，询问，第二个block块应该存储在那些位置中，后续循环执行 3~7 直到将所有block全部都写完


读取数据的流程:
* 1- 客户端向NameNode发起读取数据的请求，NameNode接收到读取数据请求后，首先判断对应目录下是否有这个数据，如果没有，直接报错，返回不存在此文件，如果有，接着判断是否有读取数据的权限，如果没有，直接报错，如果有，那么就直接根据网络拓扑关系，机架感知原理 返回对应文件各个block所在的DataNode的地址，如果block比较多，可能此时先返回一部分，如果块比较少，那么直接全部都返回了

* 2- 客户端，根据返回地址，可以选择并行的方式，或者串行的方式连接各个DataNode开始读取数据，采用IO流的方式，直接获取

* 3- 如果nameNode仅返回了一部分，那么客户端会再次请求Namenode获取下一批block所在的地址，然后重复第二步，不断的将数据全部读取到本地

* 4- 当客户端全部读取完成后，将各个block按照顺序拼接在一起，形成了最终的文件
```

- 4- HDFS的secondaryNameNode是如何辅助管理元数据

```
* 1- SNN会每隔一定的时间，检测NameNode是否需要进行checkpoint(1个小时/128M)
* 2- SNN一旦达到对应阈值，就会让Namenode执行checkpoint，滚动形成一个新的edit文件
* 3- SNN将之前整个edit文件和对应fsimage通过HTTP请求的方式将其拉取到SNN所在的节点上
* 4- 将edits文件和fsimage文件读取到内存中，进行内存合并操作，将其合并为一个新的fsimage文件(fsimage.checkpoint)
* 5- 将新的fsimage文件重新发送会给namenode，放置到namenode指定的位置下即可
* 6- NameNode将fsimage.checkpoint 重命名为 fsimage文件


也正因为如何，secondaryNameNode应该是要具备和nameNode等量内存的空间大小
```

3. MR阶段

- 1- MR的核心思想是什么?

```
分而治之：将一个任务拆分为多个子任务，分别运行在不同节点上进行并行的计算操作，各个节点计算完成后，将各个节点结果进行汇总合并处理即可
```

- 2- MR的执行流程:

```
MapTask执行流程：(Map端shuffle：  分区 排序 规约)
1- MapTask开始读取数据，在读取数据的时候，会根据文件大小的对文件数据逻辑切片，默认按照128M进行切片，有多少个片,最终就会有多少个MapTask来执行，形成k1 和 v1  ，k1表示行的起始偏移量  v1表示的每一行的数据

2- 开始执行Map逻辑(自定义)处理 ，接收k1和v1 将其转换为 k2和v2

3- 将转换后的k2和v2写入的环形缓冲区(默认:100M),随着不断的写入，环形缓冲区的数据会越来越多

```

```
4- 当缓冲区中数据达到80%的时候，就会触发溢写的机制，将数据溢写到磁盘上，形成一个小文件,在溢写的过程中，会对数据进行分区操作(给每一条数据打上分区的标记),同时对数据排序操作，如果设置了规约(combiner)也会在此时来运行的

5- 当MapTask执行完成后，将缓冲池区最后一部分数据全部溢写出来，开始对数据进行合并操作，形成一个最终的结果文件即可，在合并的过程中，依然会存在排序和规约的操作，形成的结果文件就是分区好 排好序 规好约的结果数据

reduceTask执行流程: (reduce端shuffle: 分组)
1- 各个reduceTask去MapTask中拉取属于自己分区的数据,在拉取的过程中，也是先将其拉取到内存中，如果内存不足将其溢写到磁盘上，最后对数据进行排序合并操作 (排序的目的就是为了能够更好进行分组操作)

2- 对数据进行分组操作，将相同key的数据放置在一起，分好一组数据，就会触发执行一次reduce的逻辑,将k2和v2转换为k3和v3

3- 结果进行输出，直到将所有的分组全部处理完成，整个reduce的操作也就全部的结束
```

4. Yarn阶段

- 1- Yarn如何提交MR的流程

```
第一步: 客户端提交运行一个MR的程序给Yarn集群，Yarn主节点首先会先检查提交任务请求是否OK
第二步: 如果OK，RM会随机选择一台NodeManager，启动这个任务对应的一个管理者: AppMaster(一个任务一个AppMaster)
第三步: AppMaster进行启动，启动后会和RM进行汇报，已经启动成功了，同时也会和主节点简历心跳包
第四步: AppMaster开始进行资源的申请，根据任务的对应资源要求，向RM申请资源，通过心跳包的形式，将需要申请资源信息发送给RM
第五步: RM接收到资源申请的信息后，将这个资源申请交给其底层的资源调度器(FIFO 容量 公平)，来负责完整资源的分配工作，如果当前没有资源，那么就需要等待即可，如果有资源，就会立即分配好，等待AppMaster来拉取
第六步: AppMaster会不断的询问RM(心跳)，是否已经准备好资源，如果发现已经准备好资源，那么就会立即拉取过来
第七步: AppMaster会根据资源信息(资源信息会采用container容器的方式返回)，连接对应的nodeManager，开始进行任务的分配
第八步: 将任务分配给对应的nodemanager进行执行操作，nodemanager收到任务后，占用相关的资源，启动运行，同时一旦占用资源后，后续nodemanager汇报的时候，就会告知RM资源已经被占用了
第九步: AppMaster不断的监听各个NN是否已经执行完成，如果MapTask全部都执行完成后，就会通知ReduceTask可以拉取数据，完成reduceTask的运行操作
第十步: 当所有的Task全部都执行完成后，AppMaster就会通知给RM 任务以及执行完成，RM回收资源，通知AppMaster可以关闭了
```

- 2- Yarn的三种调度方案是什么, 以及各有什么特征

```
1- FIFO Scheduler(先金先出调度器) : 只有一个队列，将所有的提交任务全部放置在一个队列中，根据提交的顺序，依次的分配资源进行运行操作
    好处: 配置简单
    弊端: 不合适共享集群,因为大的应用会直接将所有的资源全部都拿到了，导致后续的其他的任务无法运行

2- Capacity Scheduler(容量调度器): 可以对资源进行划分，形成多个队列，不同队列管理不同范围资源，比如说，分配队列，分别管理 30% 50% 20% 资源，这样可以让不同任务，放置到不同队列中，既可以保证大任务运行，也可以让小任务有资源可用
```

```
    好处：多队列，可以多任务的运行，可以进行资源的划分
    弊端：如果是一个庞大的任务，是无法拿到全部的资源，虽然支持资源抢占，但是存在一定抢占比例

    注意：此调度器是Apache版本的Hadoop默认使用的

3- Fair Scheduler(公平调度器)：使用公平调度器，不需要预留一定的资源，因为调度器会在所有运行的作业中进行动态平衡，当第一个大的任务运行的时候，公平调度器可以将全部的资源分配给第一个任务，当第二个大的任务过来后，会要求第一个任务释放出50%的资源，交给第二个任务来进行运行，当第二个任务运行完成后，会将资源在归还给第一个任务

    好处：可以让大的任务使用全部的资源，同时也可以满足其他的任务也可以有资源运行
    弊端：如果任务比较多，会导致大任务迟迟无法结束，因为资源一直被其他的任务在使用中


    注意：此调度器是CDH版本的hadoop默认使用的
```

5. Spark阶段

- 1- Spark中repartition 和 coalesce区别是什么呢?

```
相同点：
    重分区的函数，都可以改变RDD的分区数，都会返回一个新的RDD

区别点：
    repartition 可以增大分区，也可以减少分区，但是会存在shuffle
    coalesce 可以减少分区，默认不能增大分区，默认不会有shuffle ，如果想增大分区，需要将参数2设置为True

    repartition的底层其实就是coalesce，知识将coalesce的函数的参数2设置为True，参数2表示是否有shuffle
```

- 2- RDD的缓存 和 检查点的区别

```
适用场景：
    当一个RDD被多方重复使用的时候，或者一个RDD计算非常的复杂的时候，我们可以设置缓存或者检查点，将RDD的结果保存下来，下次就可以直接使用了，不需要在重新计算

区别：
    存储位置：缓存是将数据可以存储在内存，磁盘或者堆外内存中国，而检查点是将数据存储在磁盘/HDFS(主要)
    生命周期：
        缓存是临时存储，当Spark应用结束后，缓存也就会自动失效，或者调用 unpersist也会清除缓存
        检查点一旦保存了，就是永久的，只要不手动删除，检查点的数据会一直存在，即使程序已经退出
    血缘关系：
        缓存是不会截断依赖关系，因为缓存是一个临时存储，当失效后，可以通过血缘关系，重新计算
        检查点会截断依赖关系，因为检查点认为数据是可以进行可靠存储的，比如存储到HDFS，不会发送丢失,不需要重新计算
```

- 3- Spark的宽窄依赖关系

```
窄依赖: 上游的RDD的分区的数据. 被下游的分区完全的继承, 那么我将这样的依赖关系称为窄依赖关系
窄依赖关系之间是没有shuffle
    目的: 支持并行的计算

宽依赖: 上游的RDD的分区数据, 分发的下游的多个分区所接收, 中间存在shuffle的操作
    目的: 划分Stage
```

- 4- 如何生产DAG执行流程图

```
1- 当Spark程序遇到一个action的算子后, 就会触发一个Job任务的执行, 一个Spark应用程序可能会触发多个Job任务, 每一个Job任务都会产生一个DAG执行流程图

2- 首先会将这个action算子所依赖的所有的RDD全部都加载到一个stage阶段中

3- 接着对这个stage开始从后往前进行回溯操作, 依次判断RDD之间的依赖关系, 如果遇到宽依赖, 将其拆分开, 形成一个stage即可, 依次判断直到将所有的RDD全部判断为完成, 形成最终的DAG执行流程图
```

- 5- Spark RDD 的job调度流程: Driver内部运行流程

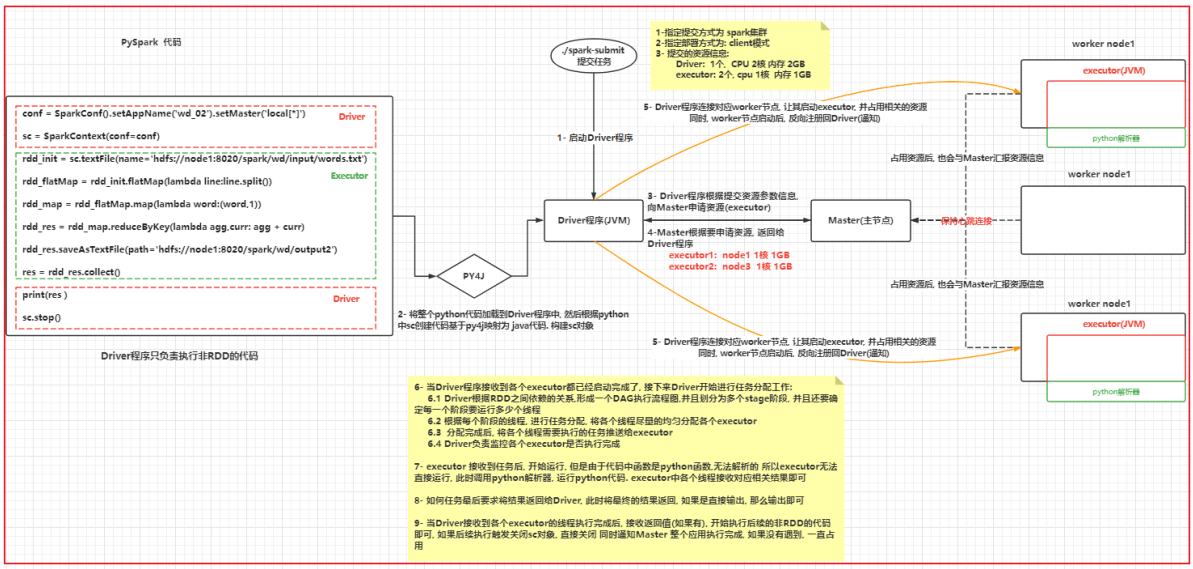
```
    Spark程序, 遇到一个action算子, 就会产生一个Job的任务, 首先就会先SparkContext对象, 同时在其底层也会创建DAGScheduler 和 TaskScheduler

    首先DAGScheduler负责生成DAG的执行流程图, 划分有多少个阶段, 并且确定每个阶段内部有多少个线程, 然后将每个阶段的线程放置到一个TaskSet中,然后提交到TaskScheduler
    接着TaskScheduler接收到TaskSet后,然后依次遍历每一个阶段的TaskSet, 将其尽可能均衡分配给对应的executor来运行, 从而完成所有的阶段的执行操作
    最后Driver负责任务的监控管理即可....
```

- 6- Spark的shuffle的机制

```
发展史:
    先有 HashShuffle --> SortShuffle --> 钨丝计划(优化阶段) --> 钨丝计划合并到SortShuffle ---> 删除HashShuffle统一合并到SortShuffle(2.0版本以后)

Hashshuffle:
    优化前: 上游的RDD的每个分片都会产生和下游分区数量相等的文件的数量, 每个文件对应下游的一个分区的数据, 这样导致产生大量的分区的文件, 对IO影响也是非常大的 最终影响效率

    优化后: 把原有由上游的每一个RDD生成与下游等同的的分区数量 转换为 由每一个executor来生成与下游等同的分区的数量, 从而减少的分区文件数量的产生,从而降低了IO, 提升了效率

SortShuffle的机制: 两种机制  普通机制 和 bypass机制
    普通机制:  将处理的数据先写入到内存中, 当内存中数据达到一定的阈值后, 就会触发溢写, 将数据溢写到磁盘上, 在溢写的时候会对数据进行分区操作, 以及排序操作, 形成的文件分好区排好序的数据, 溢写完成后, 还会将多个溢写的文件的数据合并为最终的大的文件数据, 同时这个文件数据还会携带有一个索引文件, 用于后续加载读取文件中数据
    bypass机制: 比普通机制少了排序的操作, 所以在某些情况下bypass的机制执行效率可能会高于普通机制, 毕竟干的活少了
        使用条件的:
            1- 要求RDD的分区数量不能超过200个
```

```
                2- 要求上游的RDD不允许进行提前聚合的操作

            如果满足了这两个特性， 系统会自动的采用bypass的方案
```

- 7- Spark 与 pyspark的交互流程： on Spark集群 部署模式为Client




```
1- 首先在现在提交的节点上启动一个Driver的程序
2- Driver程序启动后， 执行Mian函数， 首先创建SparkContext对象(底层基于py4j,识别python的创建方式， 将其映射为Java代码)
3- 连接Spark集群的主节点， 根据配置信息的要求， 向主节点申请资源， 用于启动executor
    说明: 资源可以自定义， 也可以使用默认值
4- Master接收到资源申请后， 开始进行分配资源， 底层的分配资源模式为FIFO(先进先出),然后将分配好的资源返回给Driver程序
5- Driver程序连接对应worker节点， 通知worker启动executor进程， 并且占用相应的资源，worker启动完成后， 返回注册给Driver程序(表示已经启动完成了)
6- Driver接收到各个executor都启动完成消息后， 开始处理代码:  任务分配
    6.1 首先会加载所有的RDD的算子 基于RDD算子之间的依赖的关系， 形成DAG执行流程图， 划分stage， 并且确定每个阶段应该运行多少个线程， 以及每个线程应该提交给那个executor来运行(任务分配)
    6.2 Driver程序通知对应的executor程序， 来执行具体的任务
    6.3 Executor接收到任务信息后， 启动线程， 开始执行处理即可: executor在执行的时候，由于RDD的中使用大量的python的函数， 这些函数executor程序是无法解决运行， 因为executor是一个JVM程序， 此时调用Python解析器. 来执行python的函数， executor接收执行后的结果即可
    6.4 executor在执行的过程中， 如果发现最终的结果需要返回给Driver， 直接返回给Driver. 如果不需要返回， 直接输出结果即可
    6.5 Driver程序监听executor执行的状态信息， 当executor都执行完成后， Driver认为任务以及执行完成了

7- 当任务执行完成后， Driver执行后续的非RDD的代码了， 当执行sc.stop的时候 通知Master执行完成 Master回收资源， Driver退出即可
```

- 8- Spark SQL如何转换为Spark RDD的流程 (HIVE 如何转换为MR 比较类似的)

```
1- 接收客户端提交过来的SQL/DSL. 首先会先校验SQL/DSL的语法是否正确, 如果通过校验, 基于SQL/DSL的执行顺序, 生成一颗抽象语法树(AST)

2- 对AST的抽象语法树加入元数据, 确定一共要涉及到有那些字段, 字段的类型是什么, 以及涉及表的相关的元数据信息, 加入元数据后, 形成一个优化前的逻辑执行计划

3- 对未优化前的逻辑执行计划进行优化, 整个优化通过优化器来实施, 在优化器中匹配对应的优化的规则, 实施优化,而优化器提交有上百种的优化的方案, 例如 谓词下推  列值裁剪  最终通过优化形成一个优化后的逻辑执行计划
4- 将优化后的逻辑执行计划, 转换为物理执行计划, 但是在转换过程中, 可能会产生多个物理执行计划, 此时内部是通过一个代价函数, 从多个优化的方案中选择出一个最优的方案, 将其转换为物理执行计划
5- 通过物理执行计划匹配Spark SQL提供的RDD的代码模板. 基于代码生成器转换为最终的RDD代码, 然后将RDD代码提交到集群运行
6- 后续整个流程, 就是JOB的调度流程 和 spark的交互流程 与 RDD是一致的
```

6. Kafka阶段

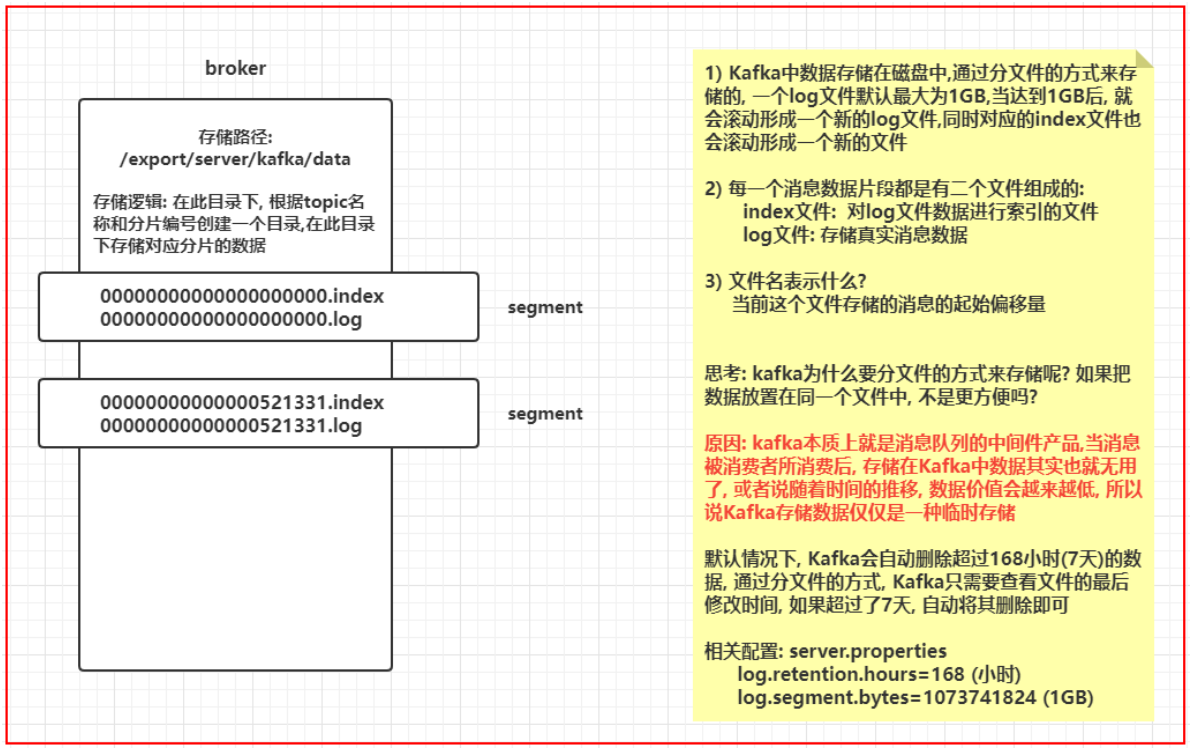
- 1- Topic的分片和副本机制

```
分片: 逻辑概念
    相当于将一个Topic(大容器),拆分为多个块(多个小容器), 多个块合并在一起就是一个Topic(大容器), 可以将各个块分布在不同的节点上
    作用:
        1- 提高读写的效率: 分片可以分布在不同的节点上, 在进行读写的时候, 可以让多个节点一起负责
        2- 分布式存储: 解决单台阶段存储容量有限的问题
    注意: 分片的数量是可以创建N多个, 与集群的节点数量是没有关系的

副本: 物理概念
    针对每个分片的数据, 都可以设置备份, 可以将其备份多个

    目的:  提高数据的可靠性 防止数据丢失

    注意: 副本的数据最多和集群的节点的数量保持一致, 但是一般建议设置2个 或者 3个 最多了
```

- 2- Kafka如何保证数据不丢失

```
生产端如何保证: ACK确认机制
    0: 生产者只管发送数据, 不关心不接收Broker给予的响应
    1: 生产者将数据发送到Broker端, 需要等待Broker端对应的topic上对应分片上的主副本接收到消息后, 才认为发送成功了
    -1/all: 生产者将数据发送到Broker端, 需要等待Broker端对应的Topic上对应分片上的所有副本都接收到消息后, 才认为发送成功了

broker端如何保证: 磁盘 + 多副本 + ack=-1

消费端如何保证: 消息的偏移量
    由Broker端记录每个消费者的消费的偏移量信息, 当消费者来消费数据的时候, 根据group.id在broker上找到对应的消费者消费为位置, 然后从这个位置开始消费即可, 如果没有这个消费者, 默认从当前开始消费, 消费完成后, 消费端需要汇报消息偏移量到broker, 告知已经消费完成了

    不会导致导致数据丢失, 但是会导致重复消费的问题
```

- 3- Kafka的消息的存储机制



- 4- Kafka的生产者的数据分发策略

```
    生产者将数据生产到Broker后， 最终只能有topic的其中一个分片来接收数据， 但是最终是由那个分片来接收数据呢,这就是生产者的数据分发策略

Kafka支持的分发策略:
    1- 随机分发策略 : python客户端支持的， Java 不支持
    2- Hash取模的分发策略: 根据key进行Hash取模分发
    3- 指定分发策略:
    4- 粘性分发策略(2.4版本下: 轮询策略) : Java客户端支持 但是Python客户端不支持
    5- 自定义分发策略
```

- 5- Kafka的消费者负载均衡的机制

```
规定:
    1- 在一个消费者组内， 消费者的数量最多和所监听的topic的分片数量是相等的， 如果有大于分片数量的消费者， 一定会有消费者处于闲置状态

    2- 在一个消费者组内， topic的一个分片的数据只能被一个消费者所消费， 不允许出现一个分片被多个消费者所接收情况， 而一个消费者是可以接收多个分片的数据
```

7- 建模的问题

- 你是否有参与数仓建模呢? 如何做的呢?

```
回答的点：
    1- 采用什么建模的模型：星型模型  雪花模型 星座模型
    2- 确定表中事实和维度表：如何确认事实和维度  （参考新零售中 讲解数仓概念的时候，会涉及的）
    3- 确认会用到那些表，然后在数仓中构建表：建表的思考点
```

建表的思考点：

```
1- 数据的同步方式是什么？
    全量覆盖同步
        处理逻辑：在建表的时候，不需要构建分区表，每一次都是将之前的数据全部删除，然后全部都重新导入一遍
        适用于：数据量比较少，而且不需要维护历史变化行为
    仅新增同步
        处理逻辑：在建表的时候，需要构建分区表，分区字段是以更新的周期一致即可，比如更新的周期为天，分区字段也应该以天为准，每一次导入上一天的新增的数据
        适用于：数据量比较大，而且不需要维护历史变化行为（并不代表表不存在变化，只不过这个变化对我们分析没有任何影响）
    新增及更新同步
        处理逻辑：在建表的时候，需要构建分区表，分区字段是以更新的周期一致即可，比如更新的周期为天，分区字段也应该以天为准，每一次导入上一天的新增及更新的数据
        适用于：数据量比较大，而且需要维护历史变化行为

    全量同步：
        处理逻辑：在建表的时候，需要构建分区表，分区字段以更新的周期一致即可，比如更新周期以天为基准，分区的字段也为天即可，每一次导入的时候，都是将整个数据集全部导入到一个新的分区中，后期定期删除老的历史数据(比如：仅保留最近一周)
        适用于：数据量比较少，而且还需要维护历史变化，同时维护周期不需要特别长
        说明：此种方式相对较少

    在项目中：
        对于一些配置型的表，是不需要维护历史变化，一旦发现变更，整个计算工作都是需要重新计算的. 而且配置型的表数据体量也不大，所以采用全量覆盖的方式
        对于一些事实表(业务表)，比如 投保表，理赔表 退保表 可以认为没有变更的行为，所有整个项目中业务表都是采用仅新增同步的方式，当然在面试中，也可以说一部分表存在变更行为，采用新增及更新的同步方式

    在学习环境中，为了简单一些，本次全部采用全量覆盖的同步方式

2- 表是否为内部表还是外部表呢？
        判断的依据：是否对数据有绝对的控制权，如果没有 必须是外部表，如果有 随意

        项目中：
            说辞一：由于后续都是自己构建表，自己通过SQOOP方式采集数据，所以对表数据有绝对的控制权，采用内部表
            说辞二：由于业务库数据是先数据导入到HDFS上指定位置上，然后后期通过HIVE建表映射对应目录下的数据，此时构建的是外部表 （推荐）

        本次学习中：直接构建内部表，后续通过SQOOP直接导入即可

3- 表是否为分区表还是分桶表呢？
        分区表：划分文件夹，将数据分在不同的文件夹中，当查询数据时候，通过分区字段获取对应分区下的数据，从而减少数据扫描量，提高查询效率

```

```
        分桶表: 分文件
            主要作用:  1- 数据采样 2- 提升Join的效率(Bucket Map Join  以及 SMB Join)

        项目中:
            除了全量覆盖同步表以外，其他所有的表都是分区表，不存在分桶表，因为数据以天来划分的，每天的数据量相对较少，后续不存在数据采样的情况，以及分桶Join优化的情况，所以不需要使用分桶表

        本次学习中: 为了方便,除了APP层以外，所有的层次都会采用普通表，不是分区 也不是分桶

4- 表选择什么存储格式和压缩方案呢?
    存储格式: 一般都是 ORC / TEXT FILE
    压缩格式: 一般都是 SNAPPY / GZ

    存储格式: 如果数据直接对接的普通文本文件的操作，只能使用textFile 否则大多数都是ORC
    压缩格式: 读多写少 采用 SNAPPY 写多读少 采用 GZ

    如果空间充足 ，没有特殊要求，建议统一采用SNAPPY

    在项目中: 所有表都是采用ORC存储格式 + ODS层为GZ压缩 + 其他层次为 SNAPPY 压缩

    学习环境: textFile + 不压缩

5- 表中字段应该如何选择呢?
    ODS层: 业务库有那些表，表中有那些字段，对应在ODS层建那些表，表中有对应相关字段 额外根据同步方式，是否添加分区字段
    其他层次: 不同层次 需要单独分析处理
```