超全大数据面试宝典

V 1.3

来自公众号：五分钟学大数据

版本	时间	描述
V1.0	2020-02-18	创建
V1.2	2020-06-17	新增 spark 、flink 相关
V1.3	2021-03-18	新增 java、JVM、mysql、JUC 等

复习大数据面试题，看这一套就够了！

超全超详细的最新大数据开发面试题

持续更新中...

本文档更新首发于公众号：*五分钟学大数据*

本套面试题堪称史上最全，既有面试技巧，面试流程，还有技术总结，面试真题，包含算法，Java，Mysql，大数据框架，大数据项目等（**持续更新中... 最新版请扫描下方二维码关注公众号：五分钟学大数据，回复【面试宝典】获取**）。

第一版是按照大数据技术进行划分(另一版，可在公众号【五分钟学大数据】后台发送 面试 获取)，第二版是综合版 (此版)。

扫码关注公众号

五分钟学大数据

目　录

第 1 章 找工作流程

1.1 学习技能

1.2 编写简历

简历编写可在公众号【五分钟学大数据】后台发送：简历，获取大数据简历的模板，包含各个行业及各个项目，非常全。

1.3 投简历

如能内推，尽量走内推路线，可免简历筛选，如果找不到内推路径，可在牛客网上搜内推码，上面有很多公司的员工发内推码。
招聘软件推荐 Boss 直聘，这个软件还是很靠谱的，其他的智联招聘，拉勾，前程无忧，猎聘也不错。

1.4 约面试时间

1) 一般需要多轮面试
2) 不想去的公司先去面试，积累面试经验，想去的公司约到最后
3) 合理安排面试时间，给自己尽可能的留一些复习的时间

1.5 面试

1) 人事面试，了解基本情况--笔试/填表
2) 一面：技术经理面试（面试官一般是入职之后你的直接上级，此面很重要）
3) 二面：技术老大面试
4) 三面：架构师/大 Boss 面试
5) .....

1.6 等回复

1) 不要立马入职

1.7 拿 offer

1.8 入职

1.9 准备必备资料，签合同

第 2 章 面试说明

2.1 笔试

1) 试题量：一般两页纸，多的有 3 页纸

2) 考题类型：选择和简答，简答居多

选择题：基础知识类型的选择题（编程语言，数据库，操作系统，大数据要点知识，网络，计算机组成原理等）

简答题：

a. 知识点的详细解释类，比如，MapReduce 的 Shuffle 详细过程

b. 对比之类，Hive 内部表和外部表的区别，MapReduce 和 Spark 的异同？

编程题：算法题（考察逻辑思维，考察解答问题的方式方法等）

数据库类型的 SQL 编写题目（MySQL 或者 Hive）这是必出题，也是分必拿题。否则就全盘覆没了。

SQL 题：必有（Hive 和 MySQL）

场景题：

给你假定一种场景，让你给出解决方案，面试问的最多的也是这种

遇到了问题（集群节点宕机，任务运行出错，数据丢失等），该如何解决？

简单脚本题：

写出一个简单的数据处理脚本，或者运维脚本（会的就自己写，不会的就百度搜，或者求救于小伙伴）

2.2 面试

2.2.1 人事面试

聊基本情况，聊工作经历，聊人生价值观，聊对工作的态度，聊方方面面，就是不聊技术

请事先准备好针对你个人基本情况的面试题

1、比如你原来在上海，为何 10 月份来北京找工作？

2、比如工作经历中，有 4 个月是不上班的，怎么解释？

.....

2.2.2 •手写代码

也就是做题

当然技术面试过程中，也有可能会出现手写代码的问题

经典手写代码：

- 快速排序和归并排序，冒泡排序（优先级由高到低）
- 一个设计模式的实现：比如单例，比如代理，比如装饰器
- 多线程相关，比如两个线程交替执行
- java,mapreduce,scala,spark 的 wordcount
- 画出你的项目的架构图 或者 数据处理流程架构图

2.2.3 技术经理面试

基本只聊技术。

基本套路：

- 先自我介绍
- 然后介绍项目
- 从项目入手
- 先聊业务
- 再切入到技术点
- 问如何实现、项目结构、数据处理流程
- 问遇到过什么难题，怎么解决的，怎么发现的和怎么避免
- 问相类似的场景，如何做技术选型

- 问相类似的突发情况，应该怎么决策快速解决问题

考察广度的同时，也会考察深度

通俗的说，也就是会在不同的方向，不同的领域问各种问题，然后针对你能回答的问题，就深入探讨，以此得知你对这门技术的掌握程度

所以：

考察广度，就是看你的技术领域分布

考察深度，就是看你你对这个技术掌握程度如何

总体来说：

- 聊业务
- 聊技术
- 聊问题
- 聊解决方案
- ....

2.2.4 CTO/技术架构师等面试（如果过了技术经理面试）

到了这种级别的面试，一般考察技术的就少了。

更多的是考察发展，眼光更长远的，考察你的职业生涯规划，考察你的价值观，考察你是否符合公司的长期发展战略

能了解公司更多信息，就切合公司实际去描述

如果不了解公司的，那就尽量按照通用套路说

2.3 机试

少数场景会有的。但是不要慌。按照自己的本事来。

有些是现场的，这种很少。有些是给出需求之后让你回来之后自己做。

2.4 面试考察方式

2.4.1 知识广度

编程语言方向

java 是重点（重中之重），其次：scala、python

java 中的重点考察方向：

集合（优缺点，底层实现，更好的替代方案，如何根据场景选择和使用）

并发（锁，JMM，各种关键字（volite），技术点，线程池，.....）

面向对象

数据库方向

mysql 是重点，其次是 hbase，redis

sql 语句的编写和优化就不说了，没有不考查的

Hadoop 体系/Spark 体系

架构原理

工作机制

典型的常见流程

某个功能的详细分析

问题和运维难点

集群规模/集群规划

任务多少

任务运行总时长

每天数据量

总数据量

多少条记录

每条记录多大

每条记录多少个字段

hive 的总表数等等

其他知识：

ElasticSearch

Flink

机器学习

架构

优化

源码

数据结构

算法

2.4.2 知识深度

追根究底的问各种你答的上来的东西的底层实现细节，直到你答不上来为止，或者到他满意为止

关于 HashMap 的问题：

- 什么是 HashMap，能否自己实现一个？
- 什么时候使用 HashMap，有没有替代品？有没有什么好的 hasmap 使用经验？
- HashMap 和 HashTable 的区别？优缺点？如何选择在那种场景使用？有没有更好的 map 实现类？
- 你知道 HashMap 的内部数据结构么？/ put 和 get 操作的原理么？
- HashMap 的初始长度和扩容策略是怎样的？（什么时候触发扩容，扩大到多少，扩容的时候要考虑什么问题？）
- HashMap 初始化传入的容量参数的值就是 HashMap 实际分配的空间么？
- HashMap 解决 hash 冲突的策略是什么？
- HashMap 能同步/线程安全么？怎么做？
- 了解 HashMap 的条件竞争么？
- JDK7 和 JDK8 的 hashmap 一样么？
- HashMap 的 key 的 hash 计算规则是怎样的？
- HashMap 的 key 有什么要求？为什么最好是 String 或者 Integer 这种类型呢？为什么不要是自定义对象呢？
- 了解 Hash 攻击么？
- 了解 ConcurrentHashMap 么？了解他的工作原理么？和 HashMap 相比较，优势在哪里？

2.5 面试过程最关键的是什么？

1） 不是你说了什么，而是你怎么说

2） 大大方方的聊，放松

2.6 面试要点

1） 乐观开朗，不让要人觉得跟你很难交流，要善于交流，善于倾听

别人说的合理的给予赞同和钦佩

别人说的你不赞同的你不要反对，你可以用另外一种方式表现出你的看法和意见，这是讨论和交流，不是针锋相对

2） 积极向上，所有人都希望融入团队的新人都能给团队增添活力，带给团队乐趣，大家轻松工作，愉快生活

3） 上进好学，不是说别人一说到不会的，你就说我会去学的，而是要表现出我曾经就是这么学过来的

4） 有礼貌有情商有智商，不要看起来傻傻的没见过世面一样的，不妄自菲薄，但是也要谦卑，要表现出我有货，但是知道自己不够，自己正在努力

5） 面试要注意引导。尽量把面试官往你擅长的领域去引导。

6） 关于回答问题，记住，如果不懂这个问题，可以让面试官再叙述一遍的。如果是真回答不上来，就真诚的回答说不知道，不了解，不太清楚。如果你发现你回答的某些问题的答案，面试官在质疑你，你也不要质疑自己。就一口咬死就是这样的。当然不能是离谱的答案还要坚持。

7） 总之一切随机应变

2.7 面试技巧

2.7.1 六个常见问题

1） 你的优点是什么？

大胆的说出自己各个方面的优势和特长

不要写消遣类的爱好，比如爬上，唱歌看电影

2） 你的缺点是什么？

不要谈自己真实问题；用“缺点”衬托自己的优点

3) 你的离职原因是什么？

- 不说前东家坏话，哪怕被伤过
- 合情合理合法
- 不要说超过 1 个以上的原因

4) 您对薪资的期望是多少？

- 非终面不深谈薪资
- 只说区间，不说具体数字
- 底线是不低于当前薪资
- 非要具体数字，区间取中间值，或者当前薪资的+20%

5) 您还有什么想问的问题？

- 这是体现个人眼界和层次的问题
- 问题本身不在于面试官想得到什么样的答案，而在于你跟别的应聘者的对比
- 标准答案：

  公司希望我入职后的 3-6 个月内，给公司解决什么样的问题

  公司（或者对这个部门）未来的战略规划是什么样子的？

6) 您最快多长时间能入职？

一周左右，如果公司需要，可以适当提前

2.7.2 两个注意事项

1) 职业化的语言
2) 职业化的形象

2.7.3 自我介绍（控制在 4 分半以内，不超过 5 分钟）

1) 个人基本信息
2) 工作履历

时间、公司名称、任职岗位、主要工作内容、工作业绩、离职原因

3） 深度沟通（也叫压力面试）

刨根问底下沉式追问（注意是下沉式，而不是发散式的）

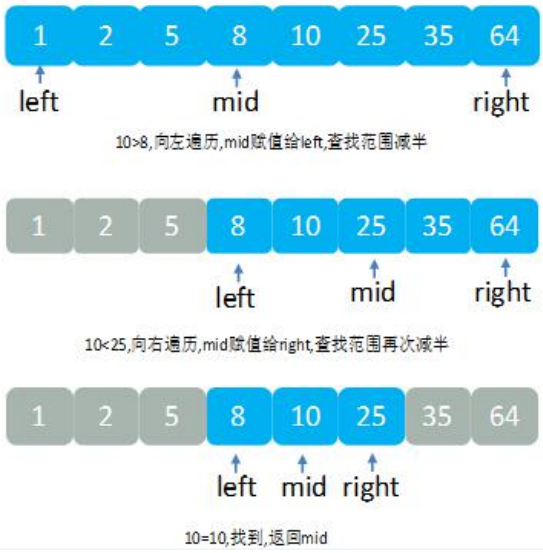
基本技巧：往自己熟悉的方向说

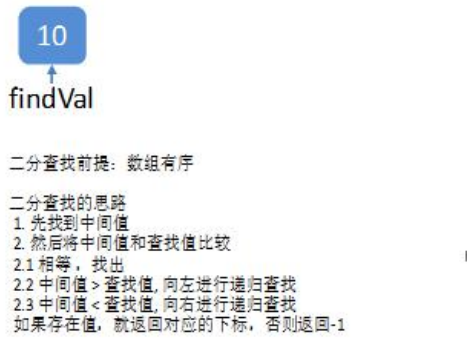
第 3 章 手写代码

3.1 冒泡排序

```java
/**
 * 冒泡排序 时间复杂度 O(n^2) 空间复杂度 O(1)
 */
public class BubbleSort {
    public static void bubbleSort(int[] data) {
        System.out.println("开始排序");
        int arrayLength = data.length;
        for (int i = 0; i < arrayLength - 1; i++) {
            boolean flag = false;
            for (int j = 0; j < arrayLength - 1 - i; j++) {
                if(data[j] > data[j + 1]){
                    int temp = data[j + 1];
                    data[j + 1] = data[j];
                    data[j] = temp;
                    flag = true;
                }
            }
            System.out.println(java.util.Arrays.toString(data));
            if (!flag)
                break;
        }
    }
    public static void main(String[] args) {
        int[] data = { 9, -16, 21, 23, -30, -49, 21, 30, 30 };
        System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
        bubbleSort(data);
        System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
    }
}
```

3.2 二分查找





实现代码：

```
/**
 * 二分查找 时间复杂度 O(log2n);空间复杂度 O(1)
 */
def binarySearch(arr:Array[Int],left:Int,right:Int,findVal:Int): Int={
  if(left>right){//递归退出条件，找不到，返回-1
    -1
  }
  val midIndex = (left+right)/2

  if (findVal < arr(midIndex)){//向左递归查找
    binarySearch(arr,left,midIndex,findVal)
  }else if(findVal > arr(midIndex)){//向右递归查找
    binarySearch(arr,midIndex,right,findVal)
  }else{//查找到，返回下标
    midIndex
  }
}
```

拓展需求：当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到。

代码实现如下：

```
/*
{1,8, 10, 89, 1000, 1000，1234} 当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到，比如这里的 1000.
//分析
1. 返回的结果是一个可变数组 ArrayBuffer
2. 在找到结果时，向左边扫描，向右边扫描 [条件]
```

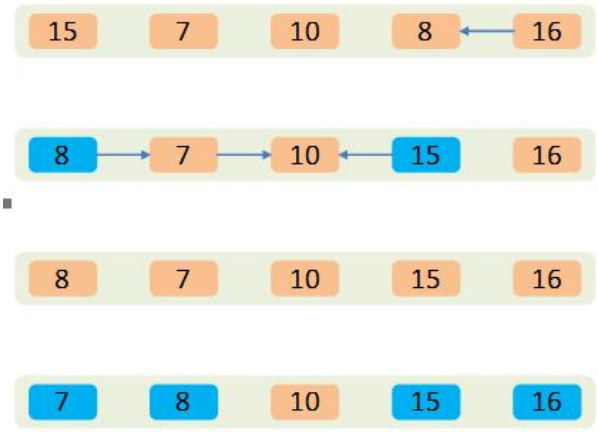
```
3. 找到结果后，就加入到 ArrayBuffer
 */
def binarySearch2(arr: Array[Int], l: Int, r: Int,
                  findVal: Int): ArrayBuffer[Int] = {

  //找不到条件?
  if (l > r) {
    return ArrayBuffer()
  }

  val midIndex = (l + r) / 2
  val midVal = arr(midIndex)
  if (midVal > findVal) {
    //向左进行递归查找
    binarySearch2(arr, l, midIndex - 1, findVal)
  } else if (midVal < findVal) { //向右进行递归查找
    binarySearch2(arr, midIndex + 1, r, findVal)
  } else {
    println("midIndex=" + midIndex)
    //定义一个可变数组
    val resArr = ArrayBuffer[Int]()
    //向左边扫描
    var temp = midIndex - 1
    breakable {
      while (true) {
        if (temp < 0 || arr(temp) != findVal) {
          break()
        }
        if (arr(temp) == findVal) {
          resArr.append(temp)
        }
        temp -= 1
      }
    }
    //将中间这个索引加入
    resArr.append(midIndex)
    //向右边扫描
    temp = midIndex + 1
    breakable {
      while (true) {
        if (temp > arr.length - 1 || arr(temp) != findVal) {
          break()
```

```
        }
        if (arr(temp) == findVal) {
          resArr.append(temp)
        }
        temp += 1
      }
    }
    return resArr
  }
```

3.3 快排



1.取数组最左边为左点(15)，取数组最右边为右点(16)，
取数组中间值为基准点（即将数据分成两组的中间值点10）

```
var l: Int = left
var r: Int = right
var pivot = arr((left + right) / 2)
```

2.左点向右走，直到大于中间值（15>10），此时右点向左走，
直到小于中间值（8<10），交换左右点的值

```
while (arr(l) < pivot) {
    l += 1
}
while (arr(r) > pivot) {
    r -= 1
}
temp = arr(l)
arr(l) = arr(r)
arr(r) = temp
```

3.左点向右走，直到大于中间值（无），此时右点向左走，
直到小于中间值（无），第一轮结束

```
if (l >= r) {
  break()
}
```

4.对中间值左右两边两个数组调用自身排序方法
（递归），将左右两个数组排序

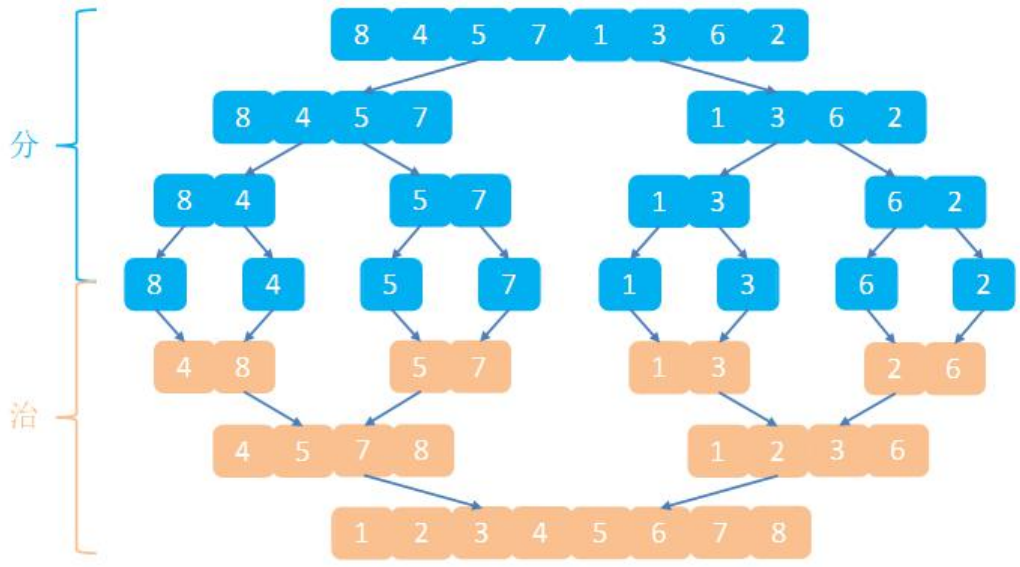
```
if (left < r) {
  quickSort(left, r, arr)
}
if (right > l) {
  quickSort(l, right, arr)
}
```

代码实现：

```
/**
 * 快排
 * 时间复杂度:平均时间复杂度为 O(nlogn)
 * 空间复杂度:O(logn)，因为递归栈空间的使用问题
 */
public class QuickSort {
    public static void main(String[] args) {
        int [] a = {1,6,8,7,3,5,16,4,8,36,13,44};
        QKSourt(a,0,a.length-1);
        for (int i:a) {
            System.out.print(i + " ");
        }
    }
    private static void QKSourt(int[] a, int start, int end) {
        if (a.length < 0){
```

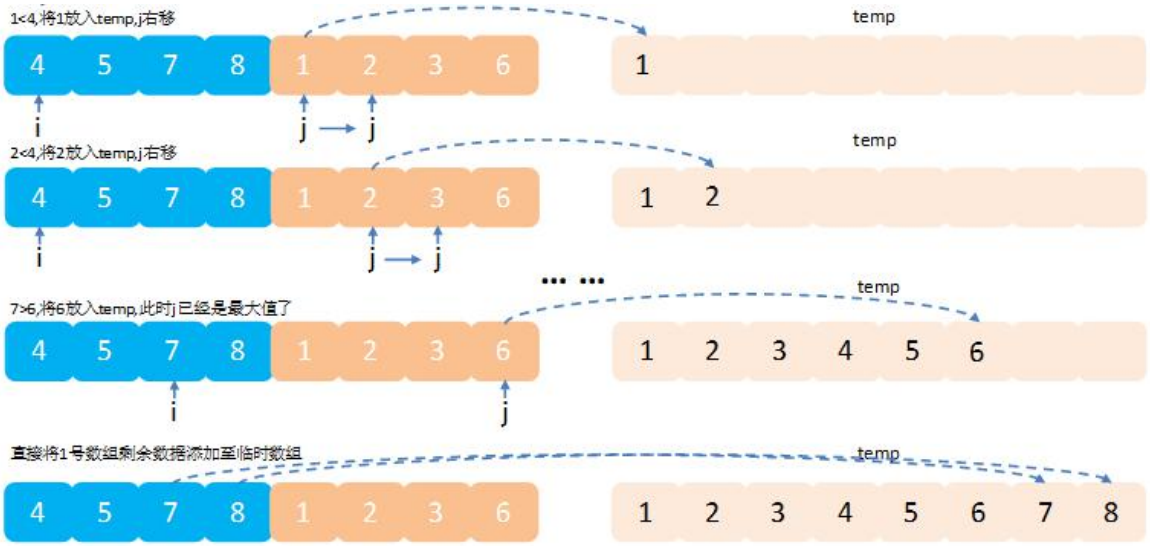
```
            return ;
        }
        if (start >= end){
            return ;
        }
        int left = start;
        int right = end;
        int temp = a[left];
        while (left < right){
            while (left < right && a[right] > temp){
                right -- ;
            }
            a[left] = a[right];
            while (left < right && a[left] < temp){
                left ++ ;
            }
            a[right] = a[left];
        }
        a[left] = temp;
        System.out.println(Arrays.toString(a));
        QKSourt(a, start, left -1);
        QKSourt(a,left+1,end);
    }
}
```

3.4 归并



核心思想：不断的将大的数组分成两个小数组，直到不能拆分为止，即形成了单个值。此时

使用合并的排序思想对已经有序的数组进行合并，合并为一个大的数据，不断重复此过程，直到最终所有数据合并到一个数组为止。



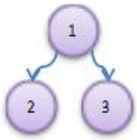
代码实现：

```
/**
 * 快排
 * 时间复杂度:O(nlogn)
 * 空间复杂度:O(n)
 */
def merge(left: List[Int], right: List[Int]): List[Int] = (left, right) match {
  case (Nil, _) => right
  case (_, Nil) => left
  case (x :: xTail, y :: yTail) =>
    if (x <= y) x :: merge(xTail, right)
    else y :: merge(left, yTail)
}
```

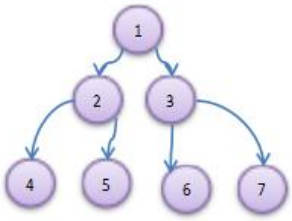
3.5 二叉树之 Scala 实现

3.5.1 二叉树概念

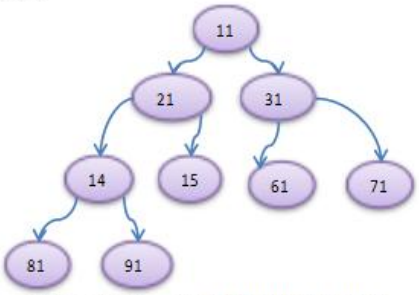
1) 树有很多种，每个节点**最多只能有两个子节点**的一种形式称为二叉树。
2) 二叉树的子节点分为左节点和右节点。
3) 如果该二叉树的**所有叶子节点都在最后一层**，并且结点总数= 2^n -1，n 为层数，则我们称为满二叉树。
4) 如果该二叉树的所有叶子节点都在最后一层或者倒数第二层，而且最后一层的叶子节点在左边连续，倒数第二层的叶子节点在右边连续，我们称为完全二叉树。



二叉树



满二叉树



完全二叉树， 如果把 (61)节点删除，
就不是完全二叉树了,因为叶子节点不连续了

3.5.2 二叉树的特点

1) 树执行查找、删除、插入的时间复杂度都是 O(logN)

2) 遍历二叉树的方法包括前序、中序、后序

3) 非平衡树指的是根的左右两边的子节点的数量不一致

4) 在非空二叉树中，第 i 层的结点总数不超过 ， i>=1；

5) 深度为 h 的二叉树最多有个结点(h>=1)，最少有 h 个结点；

6) 对于任意一棵二叉树，如果其叶结点数为 N0，而度数为 2 的结点总数为 N2，则 N0=N2+1；

3.5.3 二叉树的 Scala 代码实现

定义节点以及前序、中序、后序遍历

```
class TreeNode(treeNo:Int){
  val no = treeNo
  var left:TreeNode = null
  var right:TreeNode = null

  //后序遍历
  def postOrder():Unit={
    //向左递归输出左子树
    if(this.left != null){
      this.left.postOrder
```

```
    }
    //向右递归输出右子树
    if (this.right != null) {
      this.right.postOrder
    }

    //输出当前节点值
    printf("节点信息 no=%d \n",no)
  }

  //中序遍历
  def infixOrder():Unit={
    //向左递归输出左子树
    if(this.left != null){
      this.left.infixOrder()
    }

    //输出当前节点值
    printf("节点信息 no=%d \n",no)

    //向右递归输出右子树
    if (this.right != null) {
      this.right.infixOrder()
    }
  }

  //前序遍历
  def preOrder():Unit={
    //输出当前节点值
    printf("节点信息 no=%d \n",no)

    //向左递归输出左子树
    if(this.left != null){
      this.left.postOrder()
    }

    //向右递归输出右子树
    if (this.right != null) {
      this.right.preOrder()
    }
  }
```

```
  //后序遍历查找
  def postOrderSearch(no:Int): TreeNode = {
    //向左递归输出左子树
    var resNode:TreeNode = null
    if (this.left != null) {
      resNode = this.left.postOrderSearch(no)
    }
    if (resNode != null) {
      return resNode
    }
    if (this.right != null) {
      resNode = this.right.postOrderSearch(no)
    }
    if (resNode != null) {
      return resNode
    }
    println("ttt~~")
    if (this.no == no) {
      return this
    }
    resNode
  }

  //中序遍历查找
  def infixOrderSearch(no:Int): TreeNode = {

    var resNode : TreeNode = null
    //先向左递归查找
    if (this.left != null) {
      resNode = this.left.infixOrderSearch(no)
    }
    if (resNode != null) {
      return resNode
    }
    println("yyy~~")
    if (no == this.no) {
      return this
    }
    //向右递归查找
    if (this.right != null) {
      resNode = this.right.infixOrderSearch(no)
    }
```

```
    return resNode

  }

  //前序查找
  def preOrderSearch(no:Int): TreeNode = {
    if (no == this.no) {
      return this
    }
    //向左递归查找
    var resNode : TreeNode = null
    if (this.left != null) {
      resNode = this.left.preOrderSearch(no)
    }
    if (resNode != null){
      return  resNode
    }
    //向右边递归查找
    if (this.right != null) {
      resNode = this.right.preOrderSearch(no)
    }

    return resNode
  }

  //删除节点
  //删除节点规则
  //1 如果删除的节点是叶子节点，则删除该节点
  //2 如果删除的节点是非叶子节点，则删除该子树

  def delNode(no:Int): Unit = {
    //首先比较当前节点的左子节点是否为要删除的节点
    if (this.left != null && this.left.no == no) {
      this.left = null
      return
    }
    //比较当前节点的右子节点是否为要删除的节点
    if (this.right != null && this.right.no == no) {
      this.right = null
      return
    }
    //向左递归删除
```

```
    if (this.left != null) {
      this.left.delNode(no)
    }
    //向右递归删除
    if (this.right != null) {
      this.right.delNode(no)
    }
  }
}
```

定义二叉树，前序、中序、后序遍历，前序、中序、后序查找，删除节点

```
class BinaryTree {
  var root: TreeNode = null

  //后序遍历
  def postOrder(): Unit = {
    if (root != null) {
      root.postOrder()
    } else {
      println("当前二叉树为空，不能遍历")
    }
  }

  //中序遍历
  def infixOrder(): Unit = {
    if (root != null) {
      root.infixOrder()
    } else {
      println("当前二叉树为空，不能遍历")
    }
  }

  //前序遍历
  def preOrder(): Unit = {
    if (root != null) {
      root.preOrder()
    } else {
      println("当前二叉树为空，不能遍历")
    }
  }
```

```
  //后序遍历查找
  def postOrderSearch(no: Int): TreeNode = {
    if (root != null) {
      root.postOrderSearch(no)
    } else {
      null
    }
  }

  //中序遍历查找
  def infixOrderSeacher(no: Int): TreeNode = {
    if (root != null) {
      return root.infixOrderSearch(no)
    } else {
      return null
    }
  }

  //前序查找
  def preOrderSearch(no: Int): TreeNode = {

    if (root != null) {
      return root.preOrderSearch(no)
    } else {
      //println("当前二叉树为空，不能查找")
      return null
    }
  }

  //删除节点
  def delNode(no: Int): Unit = {
    if (root != null) {
      //先处理一下 root 是不是要删除的
      if (root.no == no) {
        root = null
      } else {
        root.delNode(no)
      }
    }

  }
}
```

3.6 手写 Spark-WordCount

```
val conf: SparkConf =
  new SparkConf().setMaster("local[*]").setAppName("WordCount")

val sc = new SparkContext(conf)

sc.textFile("/input")
  .flatMap(_.split(" "))
  .map((_, 1))
  .reduceByKey(_ + _)
  .saveAsTextFile("/output")

sc.stop()
```

第 4 章 项目架构

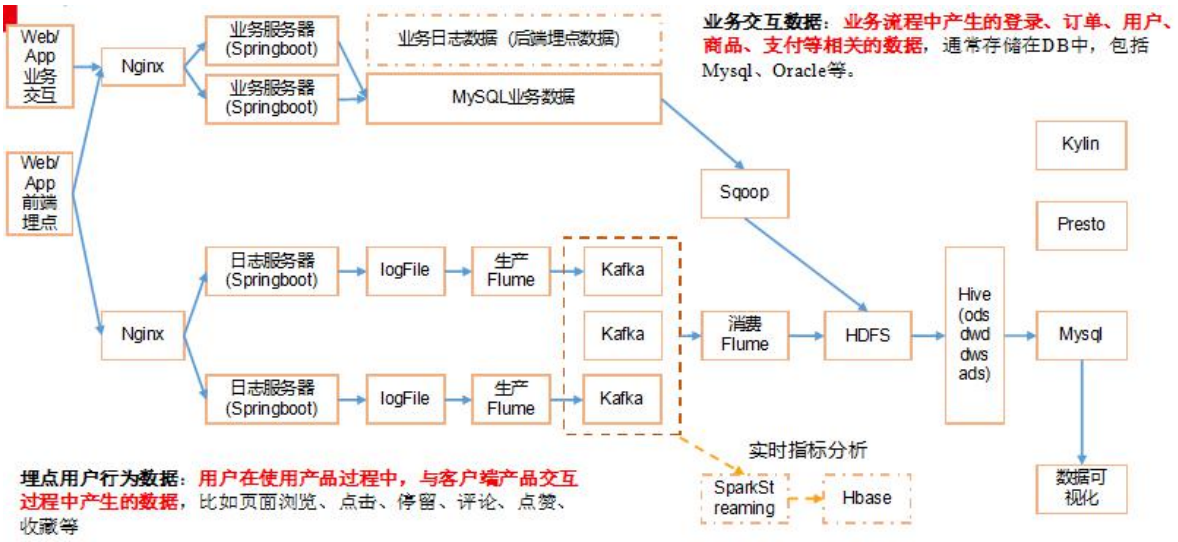
4.1 数仓概念

数据仓库的输入数据源和输出系统分别是什么？

输入系统：埋点产生的用户行为数据、JavaEE 后台产生的业务数据。

输出系统：报表系统、用户画像系统、推荐系统

4.2 系统数据流程设计



4.3 框架版本选型

1） Apache：运维麻烦，组件间兼容性需要自己调研。（一般大厂使用，技术实力雄厚，有专业的运维人员）

2） CDH：国内使用最多的版本，但 CM 不开源，但其实对中、小公司使用来说没有影响（建议使用）

3） HDP：开源，可以进行二次开发，但是没有 CDH 稳定，国内使用较少

4.4 服务器选型

服务器使用物理机还是云主机？

1） 机器成本考虑：

a. 物理机：以 128G 内存，20 核物理 CPU，40 线程，8THDD 和 2TSSD 硬盘，单台报价 4W 出头，需考虑托管服务器费用。一般物理机寿命 5 年左右

b. 云主机，以阿里云为例，差不多相同配置，每年 5W

2） 运维成本考虑：

a. 物理机：需要有专业的运维人员

b. 云主机：很多运维工作都由阿里云已经完成，运维相对较轻松

4.5 集群规模

如何确认集群规模？（假设：每台服务器8T磁盘，128G内存）

（1）每天日活跃用户100万，每人一天平均100条：10万*100条=1000万条

（2）每条日志1K左右，每天1亿条：100000000/1024/1024=约100G

（3）半年内不扩容服务器来算：100G*180天=约18T

（4）保存3副本：18T*3=54T

（5）预留20%-30%Buf=54T/0.7=77T

（6）因此：约8T*10台服务器

4.6 人员配置参考

4.6.1 整体架构

属于研发部，技术总监下面有各个项目组，我们属于数据组，其他还有后端项目组，基础平台等。总监上面就是副总等级别了。其他的还有产品运营部等。

4.6.2 你们部门的职级等级，晋升规则

职级就分初级，中级，高级。晋升规则不一定，看公司效益和职位空缺。

4.6.3 人员配置参考

小型公司（3 人左右）：组长 1 人，剩余组员无明确分工，并且可能兼顾 javaEE 和前端。

中小型公司（3~6 人左右）：组长 1 人，离线 2 人左右，实时 1 人左右（离线一般多于实时），JavaEE 1 人（有或者没有人单独负责 JavaEE，有时是有组员大数据和 JavaEE 一起做，或者大数据和前端一起做）。

中型公司（5~10 人左右）：组长 1 人，离线 3~5 人左右（离线处理、数仓），实时 2 人左右，JavaEE 1 人左右（负责对接 JavaEE 业务），前端 1 人左右（有或者没有人单独负责前端）。

中大型公司（5~20 人左右）：组长 1 人，离线 5~10 人（离线处理、数仓），实时 5 人左右，JavaEE2 人左右（负责对接 JavaEE 业务），前端 1 人（有或者没有人单独负责前端）。（发展比较良好的中大型公司可能大数据部门已经细化拆分，分成多个大数据组，分别负责不同业务）

上面只是参考配置，因为公司之间差异很大，例如 ofo 大数据部门只有 5 个人左右，因此根据所选公司规模确定一个合理范围，在面试前必须将这个人员配置考虑清楚，回答时要非常确定。

第 5 章 项目涉及技术

5.1 Linux&Shell 相关总结

5.1.1 Linux 常用命令

序号	命令	命令解释
1	top	查看内存
2	df -h	查看磁盘存储情况
3	iotop	查看磁盘 IO 读写(yum install iotop 安装)
4	iotop -o	直接查看比较高的磁盘读写程序
5	netstat -tunlp | grep 端口号	查看端口占用情况
6	uptime	查看报告系统运行时长及平均负载
7	ps aux	查看进程

5.1.2 Shell 常用工具

awk、sed、cut、sort

5.2 Hadoop 相关总结

5.2.1 Hadoop 常用端口号

- dfs.namenode.http-address:50070
- dfs.datanode.http-address:50075
- SecondaryNameNode 辅助名称节点端口号：50090
- dfs.datanode.address:50010
- fs.defaultFS:8020 或者 9000
- yarn.resourcemanager.webapp.address:8088
- 历史服务器 web 访问端口：19888

5.2.2 Hadoop 配置文件以及简单的 Hadoop 集群搭建

1) 配置文件：

core-site.xml、hdfs-site.xml、mapred-site.xml、yarn-site.xml

hadoop-env.sh、yarn-env.sh、mapred-env.sh、slaves

2) 简单的集群搭建过程：

JDK 安装

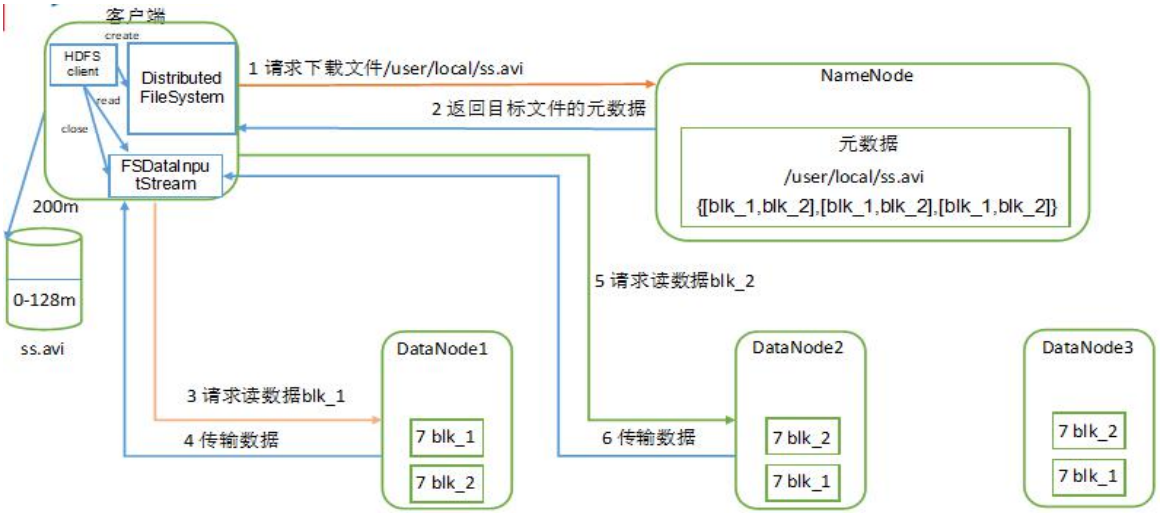
配置 SSH 免密登录

配置 hadoop 核心文件:

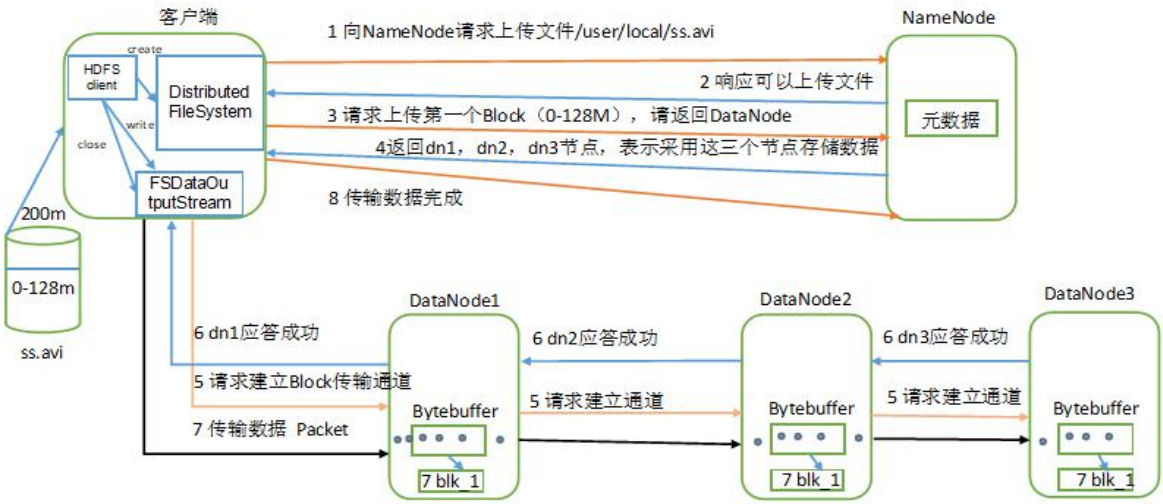
格式化 namenode

5.2.3 HDFS 读流程和写流程

HDFS 读数据：

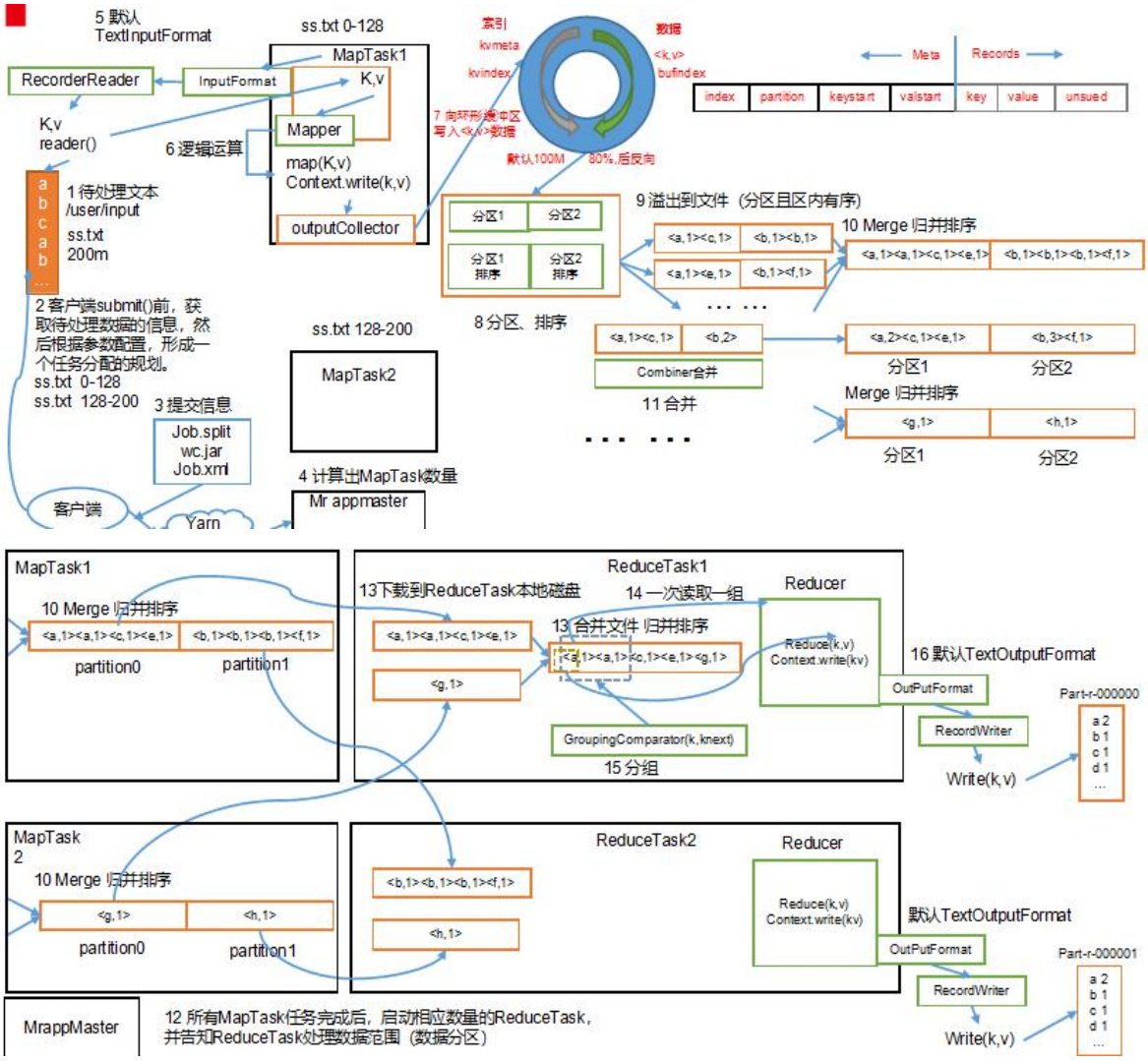


HDFS 写数据：



5.2.4 MapReduce 的 Shuffle 过程及 Hadoop 优化（包括：压缩、小文件、集群优化）

MapReduce 的详细工作流程：



一、Shuffle 机制

1） Map 方法之后 Reduce 方法之前这段处理过程叫 Shuffle

2） Map 方法之后，数据首先进入到分区方法，把数据标记好分区，然后把数据发送到环形缓冲区；环形缓冲区默认大小 100m，环形缓冲区达到 80%时，进行溢写；溢写前对数据进行排序，排序按照对 key 的索引进行字典顺序排序，**排序的手段快排**；溢写产生大量溢写文件，需要对溢写文件进行**归并排序**；对溢写的文件也可以进行 Combiner 操作，前提是汇总操作，求平均值不行。最后将文件按照分区存储到磁盘，等待 Reduce 端拉取。

3） 每个 Reduce 拉取 Map 端对应分区的数据。拉取数据后先存储到内存中，内存不够了，再存储到磁盘。拉取完所有数据后，采用**归并排序**将内存和磁盘中的数据都进行排序。在进入 Reduce 方

法前，可以对数据进行分组操作。

二、Hadoop 优化

1) HDFS 小文件影响

a. 影响 NameNode 的寿命，因为文件元数据存储在 NameNode 的内存中

b. 影响计算引擎的任务数量，比如每个小的文件都会生成一个 Map 任务

2) 数据输入小文件处理：

a. 合并小文件：对小文件进行归档（Har）、自定义 Inputformat 将小文件存储成 SequenceFile 文件。

b. 采用 ConbinFileInputFormat 来作为输入，解决输入端大量小文件场景。

c. 对于大量小文件 Job，可以开启 JVM 重用。

3) Map 阶段

a. 增大环形缓冲区大小。由 100m 扩大到 200m

b. 增大环形缓冲区溢写的比例。由 80%扩大到 90%

c. 减少对溢写文件的 merge 次数。

d. 不影响实际业务的前提下，采用 Combiner 提前合并，减少 I/O。

4) Reduce 阶段

a. 合理设置 Map 和 Reduce 数：两个都不能设置太少，也不能设置太多。太少，会导致 Task 等待，延长处理时间；太多，会导致 Map、Reduce 任务间竞争资源，造成处理超时等错误。

b. 设置 Map、Reduce 共存：调整 slowstart.completedmaps 参数，使 Map 运行到一定程度后，Reduce 也开始运行，减少 Reduce 的等待时间。

c. 规避使用 Reduce，因为 Reduce 在用于连接数据集的时候将会产生大量的网络消耗。

d. 增加每个 Reduce 去 Map 中拿数据的并行数

e. 集群性能可以的前提下，增大 Reduce 端存储数据内存的大小。

5) IO 传输

a. 采用数据压缩的方式，减少网络 IO 的的时间。安装 Snappy 和 LZOP 压缩编码器。

b. 使用 SequenceFile 二进制文件

6) 整体

a. MapTask 默认内存大小为 1G，可以增加 MapTask 内存大小为 4-5g

b. ReduceTask 默认内存大小为 1G，可以增加 ReduceTask 内存大小为 4-5g

c. 可以增加 MapTask 的 cpu 核数，增加 ReduceTask 的 CPU 核数

d. 增加每个 Container 的 CPU 核数和内存大小

e. 调整每个 Map Task 和 Reduce Task 最大重试次数

三、压缩

压缩格式	Hadoop 自带？	算法	文件扩展名	支持切分	换成压缩格式后，原来的程序是否需要修改
DEFLATE	是，直接使用	DEFLATE	.deflate	否	和文本处理一样，不需要修改
Gzip	是，直接使用	DEFLATE	.gz	否	和文本处理一样，不需要修改
bzip2	是，直接使用	bzip2	.bz2	是	和文本处理一样，不需要修改
LZO	否，需要安装	LZO	.lzo	是	需要建索引，还需要指定输入格式
Snappy	否，需要安装	Snappy	.snappy	否	和文本处理一样，不需要修改

提示：如果面试过程问起，我们一般回答压缩方式为 Snappy，特点速度快，缺点无法切分（可以回答在链式 MR 中，Reduce 端输出使用 bzip2 压缩，以便后续的 map 任务对数据进行 split）

四、切片机制

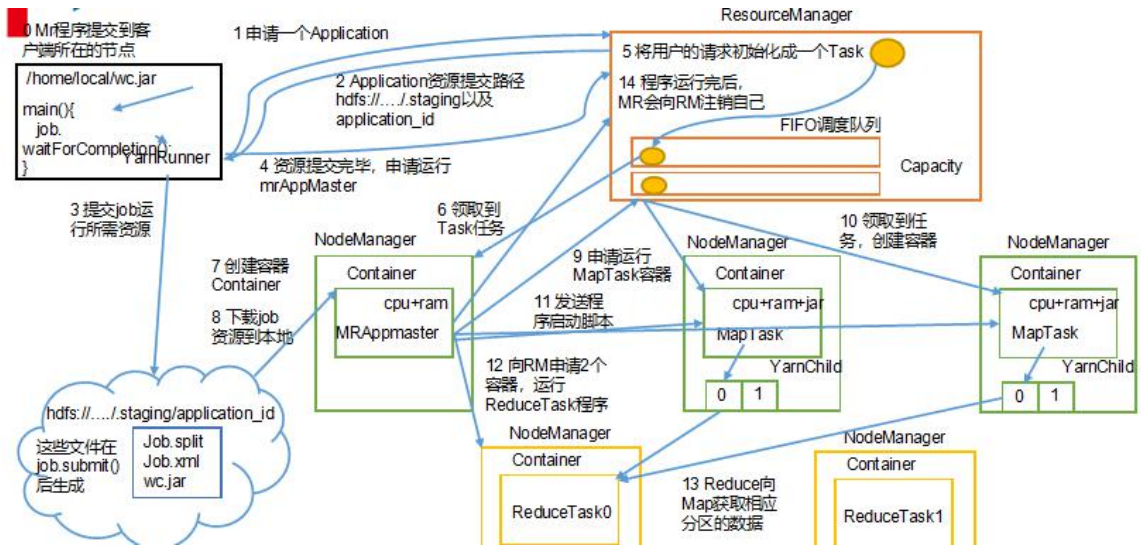
1) 简单地按照文件的内容长度进行切片

2) 切片大小，默认等于 Block 大小

3) 切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

提示：切片大小公式：max(0,min(Long_max,blockSize))

5.2.5 Yarn 的 Job 提交流程



5.2.6 Yarn 的默认调度器、调度器分类、以及他们之间的区别

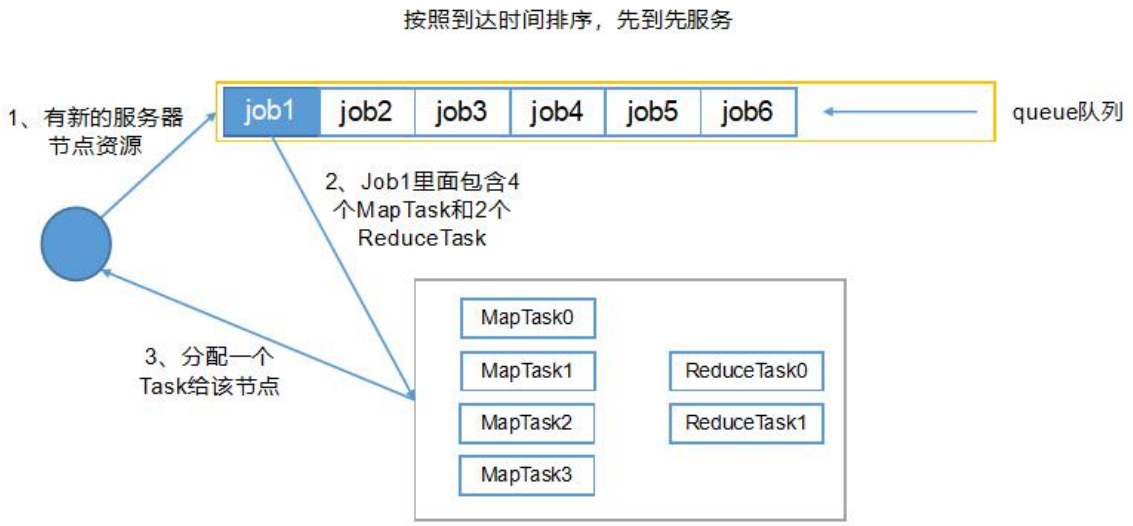
1） Hadoop 调度器重要分为三类：

FIFO（先进先出调度器）、Capacity Scheduler（容量调度器）和 Fair Sceduler（公平调度器）。

Hadoop2.7.2 默认的资源调度器是 容量调度器

2） 区别：

FIFO 调度器：先进先出，同一时间队列中只有一个任务在执行。



容量调度器：多队列；每个队列内部先进先出，同一时间队列中只有一个任务在执行。队列的并行度为队列的个数。

公平调度器：多队列；每个队列内部按照缺额大小分配资源启动任务，同一时间队列中有多个任务执行。队列的并行度大于等于队列的个数。

3） 一定要强调生产环境中不是使用的 FifoScheduler，面试的时侯会发现候选人大概了解这几种调度器的区别，但是问在生产环境用哪种，却说使用的 FifoScheduler（**企业生产环境一定不会用这个调度的**）

5.2.7 项目经验之 LZO 压缩

启用 lzo 的压缩方式对于小规模集群是很有用处，压缩比率大概能降到原始日志大小的 1/3。同时解压缩的速度也比较快。

Hadoop 默认不支持 LZO 压缩，如果需要支持 LZO 压缩，需要添加 jar 包，并在 hadoop 的 cores-site.xml 文件中添加相关压缩配置。

1） 环境准备

maven（下载安装，配置环境变量，修改 sitting.xml 加阿里云镜像）

gcc-c++

zlib-devel

autoconf

automake

libtool

通过 yum 安装即可

```
yum -y install gcc-c++ lzo-devel zlib-devel autoconf automake libtool
```

2） 下载、安装并编译 LZO

```
wget http://www.oberhumer.com/opensource/lzo/download/lzo-2.10.tar.gz
tar -zxvf lzo-2.10.tar.gz
cd lzo-2.10
```

```
./configure -prefix=/usr/local/hadoop/lzo/
make
make install
```

3） 编译 hadoop-lzo 源码

a. 下载 hadoop-lzo 的源码

```
下载地址：https://github.com/twitter/hadoop-lzo/archive/master.zip
```

b. 解压之后，修改 pom.xml

```
<hadoop.current.version>2.7.2</hadoop.current.version>
```

c. 声明两个临时环境变量

```
export C_INCLUDE_PATH=/usr/local/hadoop/lzo/include
export LIBRARY_PATH=/usr/local/hadoop/lzo/lib
```

d. 进入 hadoop-lzo-master，执行 maven 编译命令

```
mvn package -Dmaven.test.skip=true
```

e. 进入 target，将 hadoop-lzo-0.4.21-SNAPSHOT.jar 放到 hadoop 的 classpath 下

```
如${HADOOP_HOME}/share/hadoop/common
```

f. 修改 core-site.xml 增加配置支持 LZO 压缩

```
<configuration>
        <property>
            <name>io.compression.codecs</name>
            <value>
            org.apache.hadoop.io.compress.GzipCodec,
            org.apache.hadoop.io.compress.DefaultCodec,
            org.apache.hadoop.io.compress.BZip2Codec,
            org.apache.hadoop.io.compress.SnappyCodec,
            com.hadoop.compression.lzo.LzoCodec,
            com.hadoop.compression.lzo.LzopCodec
            </value>
        </property>
        <property>
            <name>io.compression.codec.lzo.class</name>
            <value>com.hadoop.compression.lzo.LzoCodec</value>
        </property>
</configuration>
```

注意：

LZO 本身是不支持分片的，但是我们给 LZO 压缩的文件加上索引，就支持分片了

5.2.8 Hadoop 参数调优

1） 在 hdfs-site.xml 文件中配置多目录，最好提前配置好，否则更改目录需要重新启动集群

2） NameNode 有一个工作线程池，用来处理不同 DataNode 的并发心跳以及客户端并发的元数据操作。

dfs.namenode.handler.count=20 * log2(Cluster Size)，比如集群规模为 10 台时，此参数设置为 60

3） 编辑日志存储路径 dfs.namenode.edits.dir 设置与镜像文件存储路径 dfs.namenode.name.dir 尽量分开，达到最低写入延迟

4） 服务器节点上 YARN 可使用的物理内存总量，默认是 8192（MB），注意，如果你的节点内存资源不够 8GB，则需要调减小这个值，而 YARN 不会智能的探测节点的物理内存总量。yarn.nodemanager.resource.memory-mb

5） 单个任务可申请的最多物理内存量，默认是 8192（MB）。yarn.scheduler.maximum-allocation-mb

5.2.9 项目经验之基准测试

Hadoop 带有一些基准测试程序，可以最少的准备成本轻松运行。基准测试被打包在测试程序 JAR 文件中，通过无参调用 JAR 文件可以得到其列表

1） 查看信息：

```
[root@node1 hadoop-2.6.0-cdh5.14.0]# bin/hadoop jar
share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.6.0-cdh5.14.0-tests.jar
```

2） 写入速度测试

测试写入 10 个 10M 文件

```
[root@node1 hadoop-2.6.0-cdh5.14.0]# bin/hadoop jar
share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.6.0-cdh5.14.0-tests.jar
TestDFSIO -write -nrFiles 10 -fileSize 10MB
```

3） 测试读文件速度

```
[root@node1 hadoop-2.6.0-cdh5.14.0]# bin/hadoop jar
share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.6.0-cdh5.14.0-tests.jar
TestDFSIO -read -nrFiles 10 -fileSize 10MB
```

4） 命令查看

```
[root@node1 hadoop-2.6.0-cdh5.14.0]# cat TestDFSIO_results.log
----- TestDFSIO ----- : write
```

```
           Date & time: Fri Nov 08 10:43:20 CST 2019
       Number of files: 10
Total MBytes processed: 100.0
     Throughput mb/sec: 4.551039912620034
Average IO rate mb/sec: 5.2242536544799805
 IO rate std deviation: 2.1074511594328245
    Test exec time sec: 52.394

----- TestDFSIO ----- : read
           Date & time: Fri Nov 08 10:45:28 CST 2019
       Number of files: 10
Total MBytes processed: 100.0
     Throughput mb/sec: 73.85524372230428
Average IO rate mb/sec: 135.5804901123047
 IO rate std deviation: 111.20953898062095
    Test exec time sec: 28.231
```

5.2.10 Hadoop 宕机

1） 如果 MR 造成系统宕机。此时要控制 Yarn 同时运行的任务数，和每个任务申请的最大内存。调整参数：yarn.scheduler.maximum-allocation-mb（单个任务可申请的最多物理内存量，默认是 8192MB）

2） 如果写入文件过量造成 NameNode 宕机。那么调高 Kafka 的存储大小，控制从 Kafka 到 HDFS 的写入速度。高峰期的时候用 Kafka 进行缓存，高峰期过去数据同步会自动跟上。

5.3 Zookeeper 相关总结

5.3.1 选举机制

半数机制

三个核心选举原则：

1） Zookeeper 集群中只有超过半数以上的服务器启动，集群才能正常工作；

2） 在集群正常工作之前，myid 小的服务器给 myid 大的服务器投票，直到集群正常工作，选出 Leader；

3） 选出 Leader 之后，之前的服务器状态由 Looking 改变为 Following，以后的服务器都是 Follower

比如：

集群半数以上存活，集群可用

假设有五台服务器组成的 zookeeper 集群，它们的 id 从 1-5，同时它们都是最新启动的

1） 1 启动，选自己

2） 2 启动，选自己（比 1 大，12 选 2）

3） 3 启动，选自己（123 都选 3，超过半数）当选 leader

4） 4 启动，已有 leader3

5） 5 启动，已有 leader3

5.3.2 常用命令

ls、get、create

5.4 Flume 相关总结

5.4.1 Flume 组成，Put 事务，Take 事务

1） flume 组成，Put 事务，Take 事务

Taildir Source：断点续传、多目录。Flume1.6 以前需要自己自定义 Source 记录每次读取文件位置，实现断点续传。

File Channel：数据存储在磁盘，宕机数据可以保存。但是传输速率慢。适合对数据传输可靠性要求高的场景，比如，金融行业。

Memory Channel：数据存储在内存中，宕机数据丢失。传输速率快。适合对数据传输可靠性要求不高的场景，比如，普通的日志数据。

Kafka Channel：减少了 Flume 的 Sink 阶段，提高了传输效率。

Source 到 Channel 是 Put 事务

Channel 到 Sink 是 Take 事务

5.4.2 Flume 拦截器

1） 拦截器注意事项

项目中自定义了：ETL 拦截器和区分类型拦截器。

采用两个拦截器的优缺点：优点，模块化开发和可移植性；缺点，性能会低一些

2） 自定义拦截器步骤

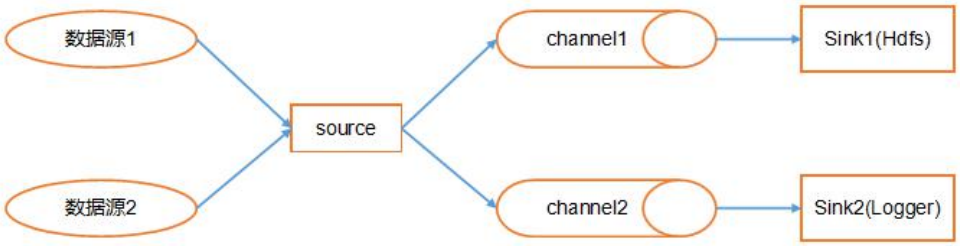
a. 实现 Interceptor

b. 重写四个方法

- initialize 初始化
- public Event intercept(Event event) 处理单个 Event
- public List<Event> intercept(List<Event> events) 处理多个 Event，在这个方法中调用 Event intercept(Event event)
- close 方法

c. 静态内部类，实现 Interceptor.Builder

5.4.3 Flume Channel 选择器



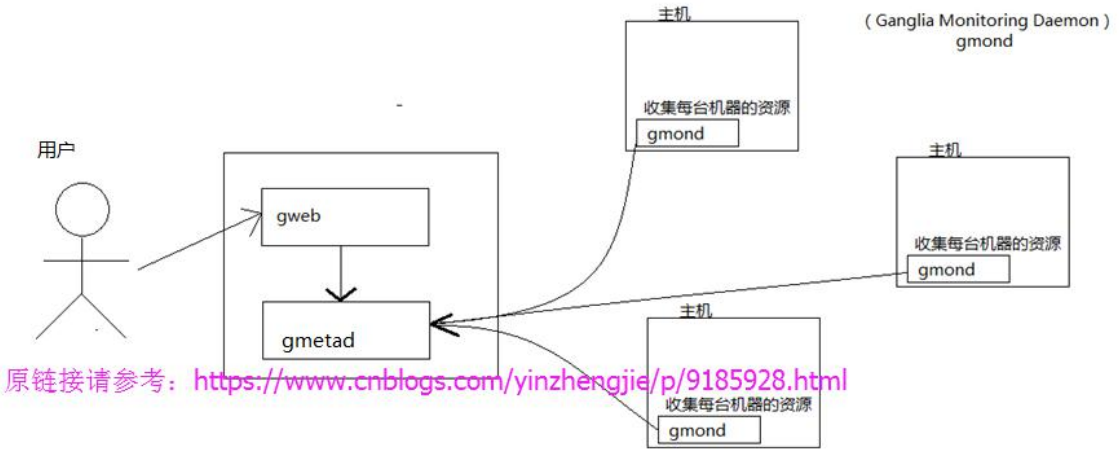
Channel Selectors，可以让不同的项目日志通过不同的Channel到不同的Sink中去。官方文档上Channel Selectors 有两种类型:Replicating Channel Selector (default)和 Multiplexing Channel Selector

这两种Selector的区别是:Replicating 会将source过来的events发往所有channel,而Multiplexing可以选择该发往哪些Channel。

5.4.4 Flume 监控器

Ganglia 监控

加州伯克利大学千禧计划的其中一个开源项目.是一个集群汇总监控用的的软件，和 Cacti 不同，cacti 是详细监控集群中每台服务器的运行状态，而 Ganglia 是将集群中的服务器数据进行汇总然后监控。有时通过 cacti 或者 zabbix（监控软件）看不出来的集群总体负载问题，却能够在 Ganglia 中体现。被监控的主机(即 client)安装 ganglia-gmond 并启动该进程。服务器端需要安装 gmetad 和 web 程序。大致大构图如下：



参考链接：https://www.cnblogs.com/yinzhengjie/p/9798739.html

5.4.5 Flume 采集数据会丢失吗?（防止数据丢失的机制）

不会，Channel 存储可以存储在 File 中，数据传输自身有事务。

5.4.6 Flume 内存

开发中在 flume-env.sh 中设置 JVM heap 为 4G 或更高，部署在单独的服务器上（4 核 8 线程 16G 内存）

-Xmx（设定程序运行期间最大可占用的内存大小）与-Xms（设定程序启动时占用内存大小）最好设置一致，减少内存抖动带来的性能影响，如果设置不一致容易导致频繁 fullgc。

5.4.7 FileChannel 优化

通过配置 dataDirs 指向多个路径，每个路径对应不同的硬盘，增大 Flume 吞吐量。

官方说明如下：

```
Comma separated list of directories for storing log files. Using multiple directories on separate disks can improve file channel peformance
```

checkpointDir 和 backupCheckpointDir 也尽量配置在不同硬盘对应的目录中，保证 checkpoint 坏掉后，可以快速使用 backupCheckpointDir 恢复数据

5.4.8 HDFS Sink 小文件处理

1) HDFS 存入大量小文件，有什么影响？

元数据层面：每个小文件都有一份元数据，其中包括文件路径，文件名，所有者，所属组，权限，创建时间等，这些信息都保存在 Namenode 内存中。所以小文件过多，会占用 Namenode 服务器大量内存，影响 Namenode 性能和使用寿命

计算层面：默认情况下 MR 会对每个小文件启用一个 Map 任务计算，非常影响计算性能。同时也影响磁盘寻址时间。

2) HDFS 小文件处理

官方默认的这三个参数配置写入 HDFS 后会产生小文件，hdfs.rollInterval、hdfs.rollSize、hdfs.rollCount

基于以上 hdfs.rollInterval=3600，hdfs.rollSize=134217728，hdfs.rollCount =0，hdfs.roundValue=10，hdfs.roundUnit= second 几个参数综合作用，效果如下：

（1）tmp 文件在达到 128M 时会滚动生成正式文件

（2）tmp 文件创建超 10 分时会滚动生成正式文件

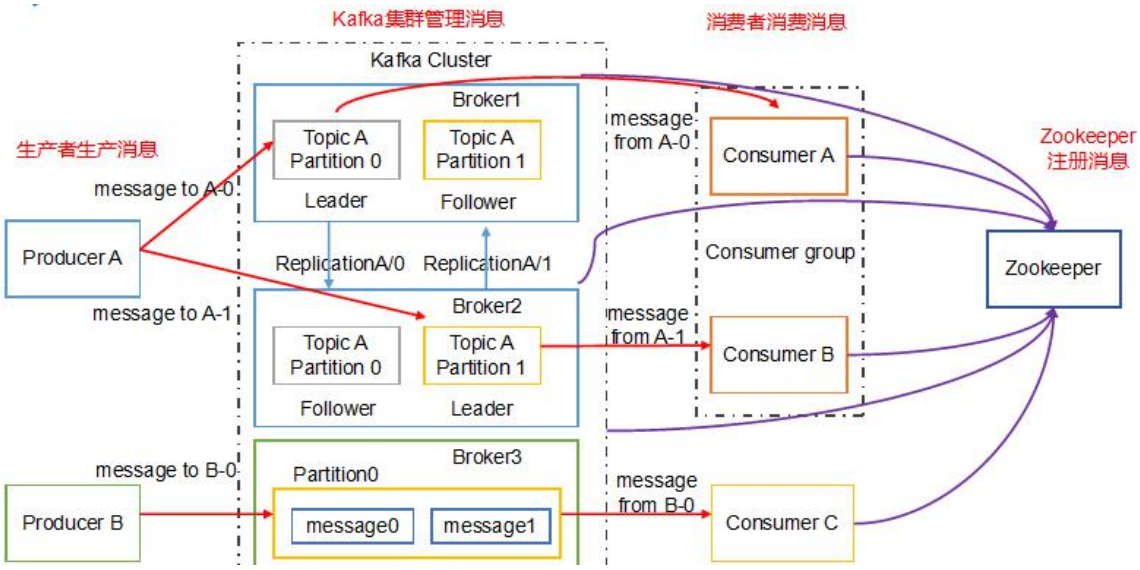
举例：在 2018-01-01 05:23 的时侯 sink 接收到数据，那会产生如下 tmp 文件：

/itcast/20180101/itcast.201801010520.tmp

即使文件内容没有达到 128M，也会在 05:33 时滚动生成正式文件

5.5 Kafka 相关总结

5.5.1 Kafka 架构



5.5.2 Kafka 压测

Kafka 官方自带压力测试脚本（kafka-consumer-perf-test.sh 、kafka-producer-perf-test.sh）。Kafka 压测时，可以查看到哪个地方出现了瓶颈（CPU，内存，网络 IO）。一般都是网络 IO 达到瓶颈。

5.5.3 Kafka 的机器数量

Kafka 机器数量=2*（峰值生产速度*副本数/100）+1

5.5.4 Kafka 的日志保存时间

7 天

5.5.5 Kafka 的硬盘大小

每天的数据量*7 天

5.5.6 Kafka 监控

公司自己开发的监控器；

开源的监控器：KafkaManager、KafkaMonitor

5.5.7 Kakfa 分区数

分区数并不是越多越好，一般分区数不要超过集群机器数量。分区数越多占用内存越大（ISR 等），一个节点集中的分区也就越多，当它宕机的时候，对系统的影响也就越大。

分区数一般设置为：3-10 个

5.5.8 副本数设定

一般我们设置成 2 个或 3 个，很多企业设置为 2 个。

5.5.9 多少个 Topic

通常情况：多少个日志类型就多少个 Topic。也有对日志类型进行合并的。

5.5.10 Kafka 丢不丢数据

Ack=0，相当于异步发送，消息发送完毕即 offset 增加，继续生产。

Ack=1，leader 收到 leader replica 对一个消息的接受 ack 才增加 offset，然后继续生产。

Ack=-1，leader 收到所有 replica 对一个消息的接受 ack 才增加 offset，然后继续生产。

5.5.11 Kafka 的 ISR 副本同步队列

ISR（In-Sync Replicas），副本同步队列。ISR 中包括 Leader 和 Follower。如果 Leader 进程挂掉，会在 ISR 队列中选择一个服务作为新的 Leader。有 replica.lag.max.messages（延迟条数）和 replica.lag.time.max.ms（延迟时间）两个参数决定一台服务是否可以加入 ISR 副本队列，在 0.10 版本移除了 replica.lag.max.messages 参数，防止服务频繁的进去队列。

任意一个维度超过阈值都会把 Follower 剔除出 ISR，存入 OSR（Outof-Sync Replicas）列表，新加入的 Follower 也会先存放在 OSR 中。

5.5.12 Kafka 中数据量计算

每天总数据量 100g，每天产生 1 亿条日志， 10000 万/24/60/60=1150 条/每秒钟

平均每秒钟：1150 条

低谷每秒钟：400 条

高峰每秒钟：1150 条*（2-20 倍）=2300 条->23000 条

每条日志大小：0.5k-2k

每秒多少数据量：2.3M-20MB

5.5.13 Kafka 挂掉

1） Flume 记录

2） 日志有记录

3） 短期没事

5.5.14 Kafka 消息数据积压，Kafka 消费能力不足怎么处理？

1） 如果是 Kafka 消费能力不足，则可以考虑增加 Topic 的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）

2） 如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

5.5.15 Kafka 的再平衡机制

5.5.15.1 什么是再平衡

所谓的再平衡，指的是在 kafka consumer 所订阅的 topic 发生变化时发生的一种分区重分配机制。一般有三种情况会触发再平衡：

- consumer group 中的新增或删除某个 consumer，导致其所消费的分区需要分配到组内其他的 consumer 上；
- consumer 订阅的 topic 发生变化，比如订阅的 topic 采用的是正则表达式的形式，如 test-* 此时如果有一个新建了一个 topic test-user，那么这个 topic 的所有分区也是会自动分配给当前的 consumer 的，此时就会发生再平衡；
- consumer 所订阅的 topic 发生了新增分区的行为，那么新增的分区就会分配给当前的 consumer，此时就会触发再平衡。

Kafka 提供的再平衡策略主要有三种：Round Robin，Range 和 Sticky，默认使用 Range。这三种分配策略的主要区别在于：

- Round Robin：会采用轮询的方式将当前所有的分区依次分配给所有的 consumer；
- Range：首先会计算每个 consumer 可以消费的分区个数，然后按照顺序将指定个数范围的分区分配给各个 consumer；
- Sticky：这种分区策略是最新版本中新增的一种策略，其主要实现了两个目的：

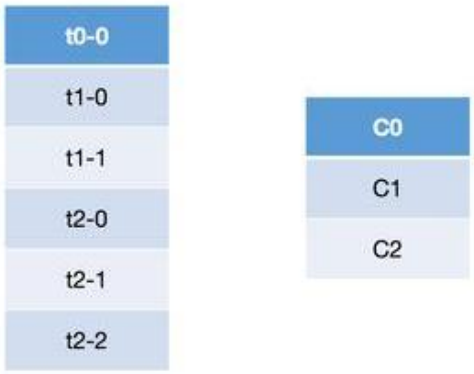
- 将现有的分区尽可能均衡的分配给各个 consumer，存在此目的的原因在于 Round Robin 和 Range 分配策略实际上都会导致某几个 consumer 承载过多的分区，从而导致消费压力不均衡；
- 如果发生再平衡，那么在重新分配前的基础上会尽力保证当前未宕机的 consumer 所消费的分区不会被分配给其他的 consumer 上；

本文主要会通过几个示例来对上面讲解的三种分区重分配策略的基本实现原理进行讲解。

5.5.15.2 Round Robin

关于 Round Robin 重分配策略，其主要采用的是一种轮询的方式分配所有的分区，该策略主要实现的步骤如下。这里我们首先假设有三个 topic：t0、t1 和 t2，这三个 topic 拥有的分区数分别为 1、2 和 3，那么总共有六个分区，这六个分区分别为：t0-0、t1-0、t1-1、t2-0、t2-1 和 t2-2。这里假设我们有三个 consumer：C0、C1 和 C2，它们订阅情况为：C0 订阅 t0，C1 订阅 t0 和 t1，C2 订阅 t0、t1 和 t2。那么这些分区的分配步骤如下：

1） 首先将所有的 partition 和 consumer 按照字典序进行排序，所谓的字典序，就是按照其名称的字符串顺序，那么上面的六个分区和三个 consumer 排序之后分别为：



2） 然后依次以按顺序轮询的方式将这六个分区分配给三个 consumer，如果当前 consumer 没有订阅当前分区所在的 topic，则轮询的判断下一个 consumer：

- 尝试将 t0-0 分配给 C0，由于 C0 订阅了 t0，因而可以分配成功；
- 尝试将 t1-0 分配给 C1，由于 C1 订阅了 t1，因而可以分配成功；
- 尝试将 t1-1 分配给 C2，由于 C2 订阅了 t1，因而可以分配成功；

- 尝试将 t2-0 分配给 C0，由于 C0 没有订阅 t2，因而会轮询下一个 consumer；
- 尝试将 t2-0 分配给 C1，由于 C1 没有订阅 t2，因而会轮询下一个 consumer；
- 尝试将 t2-0 分配给 C2，由于 C2 订阅了 t2，因而可以分配成功；
- 同理由于 t2-1 和 t2-2 所在的 topic 都没有被 C0 和 C1 所订阅，因而都不会分配成功，最终都会分配给 C2。

按照上述的步骤将所有的分区都分配完毕之后，最终分区的订阅情况如下：

C0	t0-0
C1	t1-0
C2	t1-1、t2-0、t2-1、t2-2

从上面的步骤分析可以看出，轮询的策略就是简单的将所有的 partition 和 consumer 按照字典序进行排序之后，然后依次将 partition 分配给各个 consumer，如果当前的 consumer 没有订阅当前的 partition，那么就会轮询下一个 consumer，直至最终将所有的分区都分配完毕。但是从上面的分配结果可以看出，轮询的方式会导致每个 consumer 所承载的分区数量不一致，从而导致各个 consumer 压力不均一。

5.5.15.3 Range（默认策略）

所谓的 Range 重分配策略，就是首先会计算各个 consumer 将会承载的分区数量，然后将指定数量的分区分配给该 consumer。这里我们假设有两个 consumer：C0 和 C1，两个 topic：t0 和 t1，这两个 topic 分别都有三个分区，那么总共的分区有六个：t0-0、t0-1、t0-2、t1-0、t1-1 和 t1-2。那么 Range 分配策略将会按照如下步骤进行分区的分配：

- 需要注意的是，Range 策略是按照 topic 依次进行分配的，比如我们以 t0 进行讲解，其首先会获取 t0 的所有分区：t0-0、t0-1 和 t0-2，以及所有订阅了该 topic 的 consumer：C0 和 C1，并且会将这些分区和 consumer 按照字典序进行排序；
- 然后按照平均分配的方式计算每个 consumer 会得到多少个分区，如果没有除尽，则会将多出来的分区依次计算到前面几个 consumer。比如这里是三个分区和两个 consumer，那么每

个 consumer 至少会得到 1 个分区，而 3 除以 2 后还余 1，那么就会将多余的部分依次算到前面几个 consumer，也就是这里的 1 会分配给第一个 consumer，总结来说，那么 C0 将会从第 0 个分区开始，分配 2 个分区，而 C1 将会从第 2 个分区开始，分配 1 个分区；

- 同理，按照上面的步骤依次进行后面的 topic 的分配。

最终上面六个分区的分配情况如下：

C0	t0-0、t0-1、t1-0、t1-1
C1	t0-2、t1-2

可以看到，如果按照 Range 分区方式进行分配，其本质上是依次遍历每个 topic，然后将这些 topic 的分区按照其所订阅的 consumer 数量进行平均的范围分配。这种方式从计算原理上就会导致排序在前面的 consumer 分配到更多的分区，从而导致各个 consumer 的压力不均衡。

5.5.15.4 Sticky

Sticky 策略是新版本中新增的策略，顾名思义，这种策略会保证再分配时已经分配过的分区尽量保证其能够继续由当前正在消费的 consumer 继续消费，当然，前提是每个 consumer 所分配的分区数量都大致相同，这样能够保证每个 consumer 消费压力比较均衡。关于这种分配方式的分配策略，我们分两种情况进行讲解，即初始状态的分配和某个 consumer 宕机时的分配情况。

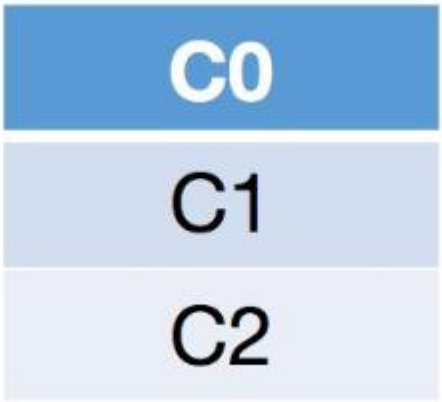
5.5.15.4.1 初始分配

初始状态分配的特点是，所有的分区都还未分配到任意一个 consumer 上。这里我们假设有三个 consumer：C0、C1 和 C2，三个 topic：t0、t1 和 t2，这三个 topic 分别有 1、2 和 3 个分区，那么总共的分区为：t0-0、t1-0、t1-1、t2-0、t2-1 和 t2-2。关于订阅情况，这里 C0 订阅了 t0，C1 订阅了 t0 和 1，C2 则订阅了 t0、t1 和 t2。这里的分区分配规则如下：

1） 首先将所有的分区进行排序，排序方式为：首先按照当前分区所分配的 consumer 数量从低到高进行排序，如果 consumer 数量相同，则按照分区的字典序进行排序。这里六个分区由于所在的 topic 的订阅情况各不相同，因而其排序结果如下：

分区排序结果	订阅的consumer数量	订阅的consumer
t2-0	1	C2
t2-1		
t2-2		
t1-0	2	C1、C2
t1-1		
t0-0	3	C0、C1、C2

2） 然后将所有的 consumer 进行排序，其排序方式为：首先按照当前 consumer 已经分配的分区数量有小到大排序，如果两个 consumer 分配的分区数量相同，则会按照其名称的字典序进行排序。由于初始时，这三个 consumer 都没有分配任何分区，因而其排序结果即为其按照字典序进行排序的结果：



3） 然后将各个分区依次遍历分配给各个 consumer，首先需要注意的是，这里的遍历并不是 C0 分配完了再分配给 C1，而是每次分配分区的时候都整个的对所有的 consumer 从头开始遍历分配，如果当前 consumer 没有订阅当前分区，则会遍历下一个 consumer。然后需要注意的是，在整个分配的过程中，各个 consumer 所分配的分区数是动态变化的，而这种变化是会体现在各个 consumer 的排序上的，比如初始时 C0 是排在第一个的，此时如果分配了一个分区给 C0，那么 C0 就会排到最后，因为其拥有的分区数是最多的。上面的六个分区整体的分配流程如下：

a. 首先将 t2-0 尝试分配给 C0，由于 C0 没有订阅 t2，因而分配不成功，继续轮询下一个 consumer；

b. 然后将 t2-0 尝试分配给 C1，由于 C1 没有订阅 t2，因而分配不成功，继续轮询下一个

consumer；

c. 接着将 t2-0 尝试分配给 C2，由于 C2 订阅了 t2，因而分配成功，此时由于 C2 分配的分区数发生变化，各个 consumer 变更后的排序结果为：

C0
C1
C2

d. 接下来的 t2-1 和 t2-2，由于也只有 C2 订阅了 t2，因而其最终还是会分配给 C2，最终在 t2-0、t2-1 和 t2-2 分配完之后，各个 consumer 的排序以及其分区分配情况如下：

C0	
C1	
C2	t2-0、t2-1、t2-2

e. 接着继续分配 t1-0，首先尝试将其分配给 C0，由于 C0 没有订阅 t1，因而分配不成功，继续轮询下一个 consumer；

f. 然后尝试将 t1-0 分配给 C1，由于 C1 订阅了 t1，因而分配成功，此时各个 consumer 以及其分配的分区情况如下：

C0	
C1	t1-0
C2	t2-0、t2-1、t2-2

g. 同理，接下来会分配 t1-1，虽然 C1 和 C2 都订阅了 t1，但是由于 C1 排在 C2 前面，因而该分区会分配给 C1，即：

C0	
C1	t1-0、t1-1
C2	t2-0、t2-1、t2-2

h. 最后，尝试将 t0-0 分配给 C0，由于 C0 订阅了 t0，因而分配成功，最终的分配结果为：

C0	t0-0
C1	t1-0、t1-1
C2	t2-0、t2-1、t2-2

上面的分配过程中，需要始终注意的是，虽然示例中的 consumer 顺序始终没有变化，但这是由于各个分区分配之后正好每个 consumer 所分配的分区数量的排序结果与初始状态一致。这里读者也可以比较一下这种分配方式与前面讲解的 Round Robin 进行对比，可以很明显的发现，Sticky 重分配策略分配得更加均匀一些。

5.5.15.4.2 模拟 consumer 宕机

由于前一个示例中最终的分区分配方式模拟宕机的情形比较简单，因而我们使用另一种订阅策略。这里我们的示例的 consumer 有三个：C0、C1 和 C2，topic 有四个：t0、t1、t2 和 t3，每个 topic 都有两个分区，那么总的分区有：t0-0、t0-1、t1-0、t1-1、t2-0、t2-1、t3-0 和 t3-1。这里的订阅情况为三个 consumer 订阅所有的主题，那么如果按照 Sticky 的分区分配策略，初始状态时，分配情况如下，读者可以按照前一示例讲解的方式进行推算：

C0	t0-0、t1-1、t3-0
C1	t0-1、t2-0、t3-1
C2	t1-0、t2-1

这里我们假设在消费的过程中，C1 发生了宕机，此时就会发生再平衡，而根据 Sticky 策略，其再分配步骤如下：

1） 首先会将宕机之后未分配的分区进行排序，排序方式为：首先按照分区所拥有的 consumer 数量从低到高进行排序，如果 consumer 数量相同，则按照分区的字典序进行排序。这里需要注意的是，由于只有 C1 宕机，因而未分配的分区为：t0-1、t2-0 和 t3-1，排序之后的结果为：

分区排序结果	订阅的consumer数量	订阅的consumer
t0-1	4	C0、C1、C2、C3
t2-0	4	C0、C1、C2、C3
t3-1	4	C0、C1、C2、C3

2） 然后将所有的 consumer 进行排序，排序方式为：首先将 consumer 按照其所拥有的 consumer 数量从小到大排序，如果数量相同，则按照 consumer 名称的字典序进行排序，排序结果如下：

C2	t1-0、t2-1
C0	t0-0、t1-1、t3-0

3） 接着依次遍历各个分区，将其分配给各个 consumer，需要注意的是，在分配的过程中，consumer 所分配的分区数量是在变化的，而这种变化是会反应在 consumer 的排序上的：

a. 首先尝试将 t0-1 分配给 C2，由于 C2 订阅了 t0，因而可以分配成功，此时 consumer 排

序和分区分配情况如下，需要注意的是，虽然分配之后，C2 和 C0 的分区数量相同，但是由于按照字典序，C0 在 C2 前面，因而排序情况还是会发生变化：

C0	t0-0、t1-1、t3-0
C2	t1-0、t2-1、t0-1

b. 然后尝试将 t2-0 分配给 C0，由于 C0 订阅了 t2，因而分配可以成功，此时 consumer 排序和分区分配情况如下：

C2	t1-0、t2-1、t0-1
C0	t0-0、t1-1、t3-0、t2-0

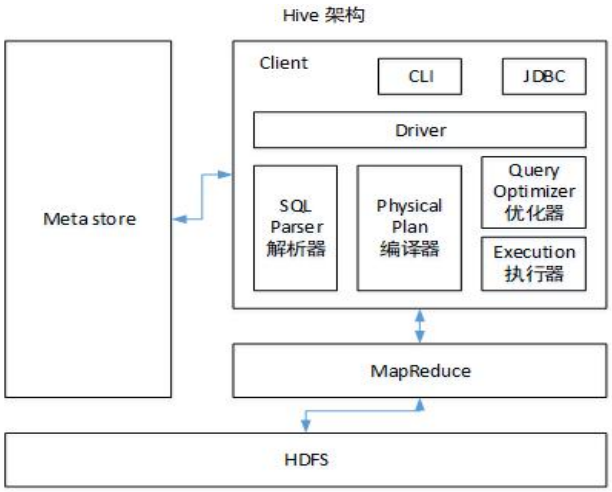
c. 最后尝试分配 t3-1 给 C2，由于 C2 订阅了 t3，因而分配可以成功，此时 consumer 排序与分区分配情况如下：

C0	t0-0、t1-1、t3-0、t2-0
C2	t1-0、t2-1、t0-1、t3-1

在上面的分区分配过程中，我们可以看到，由于分区的不断分配，各个 consumer 所拥有的分区数量也在不断变化，因而其排序情况也在变化，但是最终可以看到，各个分区是均匀的分配到各个 consumer 的，并且还保证了当前 consumer 已经消费的分区是不会分配到其他的 consumer 上的。

5.6 Hive 相关总结

5.6.1 Hive 的架构



5.6.2 Hive 和数据库比较

Hive 和数据库除了拥有类似的查询语言，再无类似之处。

1) 数据存储位置

Hive 存储在 HDFS。数据库将数据保存在块设备或者本地文件系统中。

2) 数据更新

Hive 中不建议对数据的改写。而数据库中的数据通常是需要经常进行修改的，

3) 执行延迟

Hive 执行延迟较高。数据库的执行延迟较低。当然，这个是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。

4) 数据规模

Hive 支持很大规模的数据计算；数据库可以支持的数据规模较小。

5.6.3 内部表和外部表

1) 内部表：当我们删除一个内部表时，Hive 也会删除这个表中数据。内部表不适合和其他工具共享数据。

2) 外部表：删除该表并不会删除掉原始数据，删除的是表的元数据

5.6.4 4 个 By 区别

1) Sort By：分区内有序；

2) Order By：全局排序，只有一个 Reducer；

3) Distrbute By：类似 MR 中 Partition，进行分区，结合 sort by 使用。

4) Cluster By：当 Distribute by 和 Sorts by 字段相同时，可以使用 Cluster by 方式。Cluster by 除了具有 Distribute by 的功能外还兼具 Sort by 的功能。但是排序只能是升序排序，不能指定排序规则为 ASC 或者 DESC。

5.6.5 窗口函数

RANK() 排序相同时会重复，总数不会变

DENSE_RANK() 排序相同时会重复，总数会减少

ROW_NUMBER() 会根据顺序计算

1) OVER()：指定分析函数工作的数据窗口大小，这个数据窗口大小可能会随着行的变而变化

2） CURRENT ROW：当前行

3） n PRECEDING：往前 n 行数据

4） n FOLLOWING：往后 n 行数据

5） UNBOUNDED：起点，UNBOUNDED PRECEDING 表示从前面的起点， UNBOUNDED FOLLOWING 表示到后面的终点

6） LAG(col,n)：往前第 n 行数据

7） LEAD(col,n)：往后第 n 行数据

8） NTILE(n)：把有序分区中的行分发到指定数据的组中，各个组有编号，编号从 1 开始，对于每一行，NTILE 返回此行所属的组的编号。注意：n 必须为 int 类型。

5.6.6 自定义 UDF、UDTF

在项目中是否自定义过 UDF、UDTF 函数，以及用他们处理了什么问题，及自定义步骤？

1） 自定义过。

2） 用 UDF 函数解析公共字段；用 UDTF 函数解析事件字段。

自定义 UDF：继承 UDF，重写 evaluate 方法

自定义 UDTF：继承自 GenericUDTF，重写 3 个方法：initialize(自定义输出的列名和类型)，process（将结果返回 forward(result)），close

为什么要自定义 UDF/UDTF，因为自定义函数，可以自己埋点 Log 打印日志，出错或者数据异常，方便调试.

5.6.7 Hive 优化

1） MapJoin

如果不指定 MapJoin 或者不符合 MapJoin 的条件，那么 Hive 解析器会将 Join 操作转换成 Common Join，即：在 Reduce 阶段完成 join。容易发生数据倾斜。可以用 MapJoin 把小表全部加载到内存在 map 端进行 join，避免 reducer 处理。

2） 行列过滤

列处理：在 SELECT 中，只拿需要的列，如果有，尽量使用分区过滤，少用 SELECT *。

行处理：在分区剪裁中，当使用外关联时，如果将副表的过滤条件写在 Where 后面，那么就会先全表关联，之后再过滤。

3) 采用分桶技术

4) 采用分区技术

5) 合理设置 Map 数

a. 通常情况下，作业会通过 input 的目录产生一个或者多个 map 任务。

主要的决定因素有：input 的文件总个数，input 的文件大小，集群设置的文件块大小。

b. 是不是 map 数越多越好？

答案是否定的。如果一个任务有很多小文件（远远小于块大小 128m），则每个小文件也会被当做一个块，用一个 map 任务来完成，而一个 map 任务启动和初始化的时间远远大于逻辑处理的时间，就会造成很大的资源浪费。而且，同时可执行的 map 数是受限的。

c. 是不是保证每个 map 处理接近 128m 的文件块，就高枕无忧了？

答案也是不一定。比如有一个 127m 的文件，正常会用一个 map 去完成，但这个文件只有一个或者两个小字段，却有几千万的记录，如果 map 处理的逻辑比较复杂，用一个 map 任务去做，肯定也比较耗时。

针对上面的问题 2 和 3，我们需要采取两种方式来解决：即减少 map 数和增加 map 数；

6) 小文件进行合并

在 Map 执行前合并小文件，减少 Map 数：CombineHiveInputFormat 具有对小文件进行合并的功能（系统默认的格式）。HiveInputFormat 没有对小文件合并功能。

7) 合理设置 Reduce 数

Reduce 个数并不是越多越好

a. 过多的启动和初始化 Reduce 也会消耗时间和资源；

b. 另外，有多少个 Reduce，就会有多少个输出文件，如果生成了很多个小文件，那么如果这些小文件作为下一个任务的输入，则也会出现小文件过多的问题；

在设置 Reduce 个数的时候也需要考虑这两个原则：处理大数据量利用合适的 Reduce 数；使单个 Reduce 任务处理数据量大小要合适；

8) 常用参数

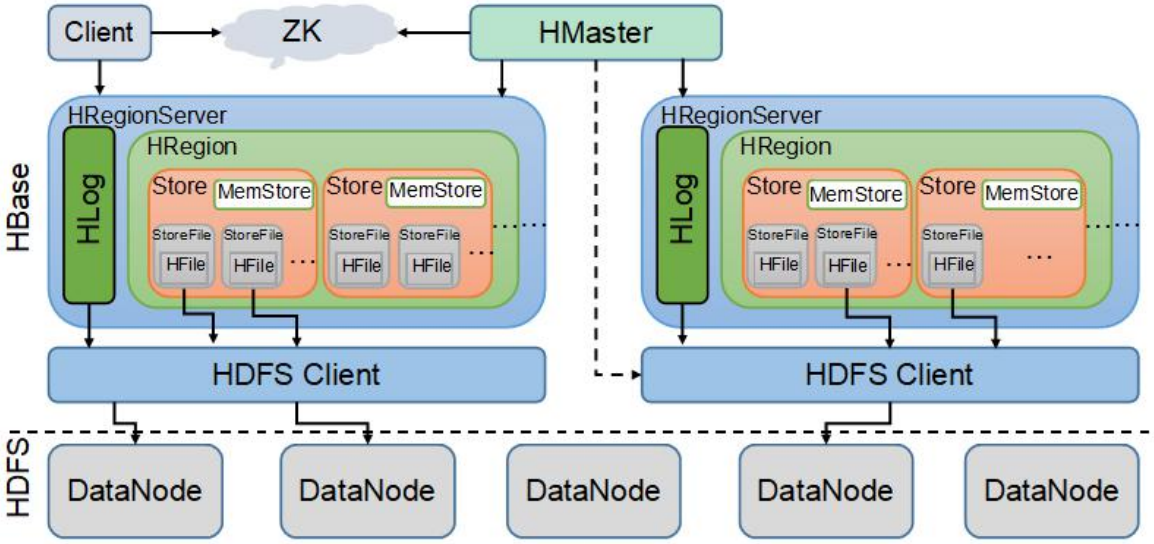
// 输出合并小文件

```
SET hive.merge.mapfiles = true; -- 默认 true，在 map-only 任务结束时合并小文件
SET hive.merge.mapredfiles = true; -- 默认 false，在 map-reduce 任务结束时合并小文件
SET hive.merge.size.per.task = 268435456; -- 默认 256M
```

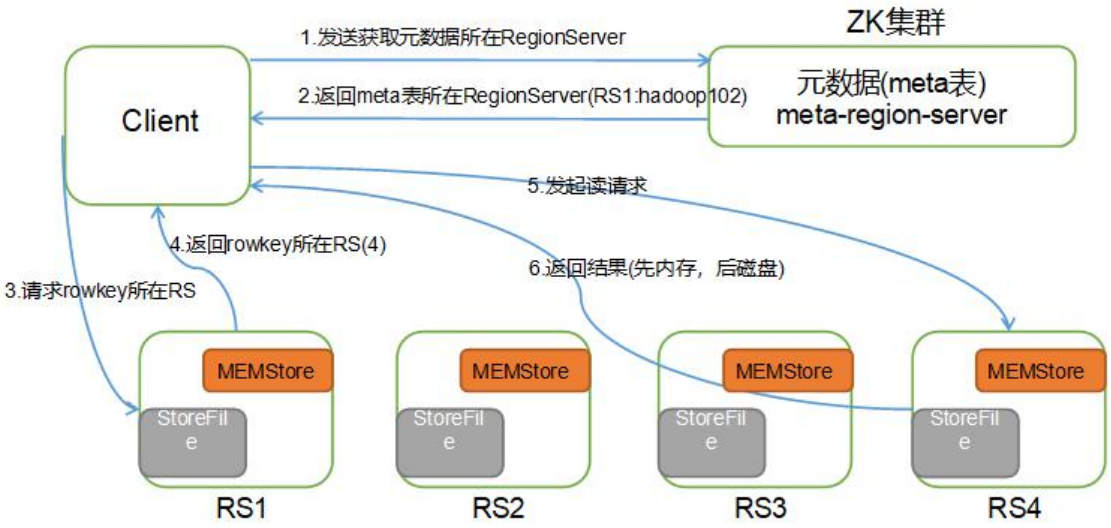
```
SET hive.merge.smallfiles.avgsize = 16777216; -- 当输出文件的平均大小小于该值时，启动一个独立的 map-reduce 任务进行文件 merge
```

5.7 HBase 相关总结

5.7.1 HBase 存储结构



5.7.2 读流程



1） Client 先访问 zookeeper，从 meta 表读取 region 的位置，然后读取 meta 表中的数据。meta 中又存储了用户表的 region 信息；

2） 根据 namespace、表名和 rowkey 在 meta 表中找到对应的 region 信息；

3） 找到这个 region 对应的 regionserver；

4） 查找对应的 region；

5） 先从 MemStore 找数据，如果没有，再到 BlockCache 里面读；

6） BlockCache 还没有，再到 StoreFile 上读(为了读取的效率)；

7） 如果是从 StoreFile 里面读取的数据，不是直接返回给客户端，而是先写入 BlockCache，再返回给客户端。

5.7.3 写流程

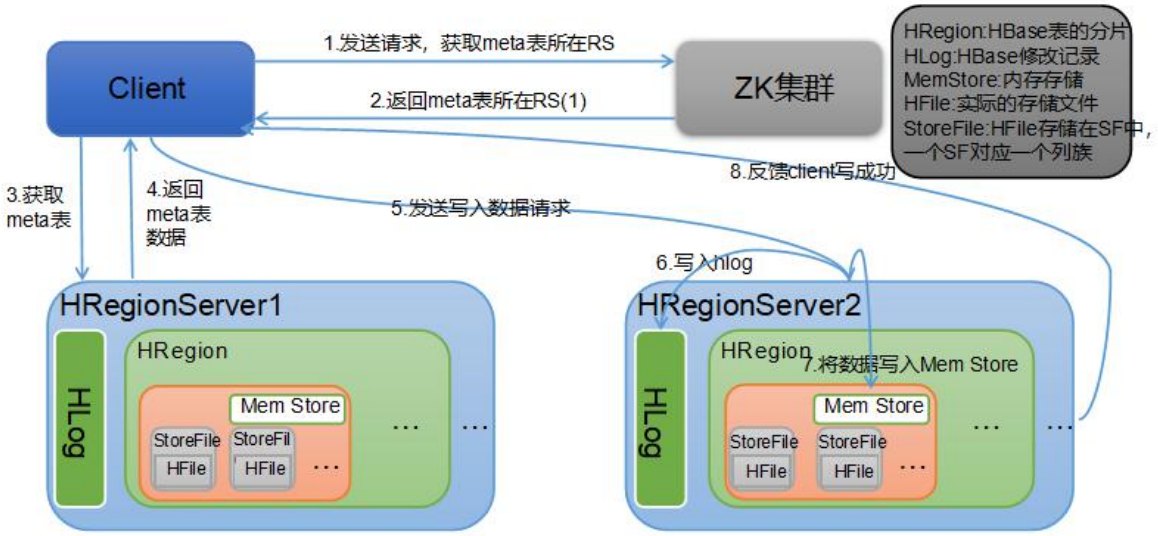


图 2 HBase 写数据流程

1） Client 向 HregionServer 发送写请求；

2） HRegionServer 将数据写到 HLog（write ahead log）。为了数据的持久化和恢复；

3） HRegionServer 将数据写到内存（MemStore）；

4） 反馈 Client 写成功。

5.7.4 数据 flush 过程

1） 当 MemStore 数据达到阈值（默认是 128M，老版本是 64M），将数据刷到硬盘，将内存中的数据删除，同时删除 HLog 中的历史数据；

2） 并将数据存储到 HDFS 中；

3） 在 HLog 中做标记点。

5.7.5 数据合并过程

1） 当数据块达到 3 块，Hmaster 触发合并操作，Region 将数据块加载到本地，进行合并；

2） 当合并的数据超过 256M，进行拆分，将拆分后的 Region 分配给不同的 HregionServer 管理；

3） 当 HregionServer 宕机后，将 HregionServer 上的 hlog 拆分，然后分配给不同的 HregionServer 加载，修改.META.；

4） 注意：HLog 会同步到 HDFS。

5.7.6 hbase-default.xml 中相关参数

```
Flush:
	<!--当 memstore 的大小超过这个值的时候，会 flush 到磁盘。128M-->
	<property>
		<name>hbase.hregion.memstore.flush.size</name>
		<value>134217728</value>
	</property>
	<!--单个 regionserver 的全部 memstore 的最大值。超过这个值总容量(Max Heap=983.4 M)*0.4，
	一个新的 put 插入操作会被挂起，强制执行 flush 操作。 -->
	<property>
		<name>hbase.regionserver.global.memstore.upperLimit</name>
		<value>0.4</value>
	</property>
	<!--当强制执行 flush 操作的时候，当低于这个值的时候，flush 会停止。默认是堆大小的 35% 。 -->
	<property>
		<name>hbase.regionserver.global.memstore.lowerLimit</name>
		<value>0.35</value>
	</property>


Compact:
	<!--当一个 HStore 含有多于这个值的 HStoreFiles(每一个 memstore flush 产生一个 HStoreFile)的时候，会执行一个合并操作，把这 HStoreFiles
写成一个-->
	<property>
		<name>hbase.hstore.compactionThreshold</name>
		<value>3</value>
	</property>
	<!--一个 Region 中的所有 HStoreFile 的 major compactions 的时间间隔。默认是 1 天。-->
	<property>
		<name>hbase.hregion.majorcompaction</name>
		<value>86400000</value>
	</property>


Split:
	<!--最大 HStoreFile 大小。若某个列族的 HStoreFile 增长达到这个值，这个 Hegion 会被切割成两个。 默认：10G.-->
	<property>
		<name>hbase.hregion.max.filesize</name>
		<value>10737418240</value>
```

```
    </property>
```

5.7.7 rowkey 设计原则

1） rowkey 长度原则（Rowkey 的长度被很多开发者建议说设计在 10~100 个字节，不过建议是越短越好，不要超过 16 个字节）

2） rowkey 散列原则

3） rowkey 唯一原则

5.7.8 RowKey 如何设计

1） 生成随机数、hash、散列值

2） 字符串反转

5.8 Sqoop 参数

```
/opt/module/sqoop/bin/sqoop import \
--connect \
--username \
--password \
--target-dir \
--delete-target-dir \
--num-mappers \
--fields-terminated-by    \
--query    "$2" ' and $CONDITIONS;'
```

5.8.1 Sqoop 导入导出 Null 存储一致性问题

Hive 中的 Null 在底层是以“\N”来存储，而 MySQL 中的 Null 在底层就是 Null，为了保证数据两端的一致性。在导出数据时采用--input-null-string 和--input-null-non-string 两个参数。导入数据时采用--null-string 和--null-non-string。

5.8.2 Sqoop 数据导出一致性问题

1）场景 1：如 Sqoop 在导出到 Mysql 时，使用 4 个 Map 任务，过程中有 2 个任务失败，那此时 MySQL 中存储了另外两个 Map 任务导入的数据，此时老板正好看到了这个报表数据。而开发工程师发现任务失败后，会调试问题并最终将全部数据正确的导入 MySQL，那后面老板再次看报表数据，发现本次看到的数据与之前的不一致，这在生产环境是不允许的。

官网：http://sqoop.apache.org/docs/1.4.6/SqoopUserGuide.html

```
Since Sqoop breaks down export process into multiple transactions, it is possible that a failed export job may result in partial data being committed to the database. This can further lead to subsequent jobs failing due to
```

```
insert collisions in some cases, or lead to duplicated data in others. You can overcome this problem by specifying a staging table via the  --staging-table  option which acts as an auxiliary table that is used to stage exported data. The staged data is finally moved to the destination table in a single transaction.
```

- staging-table 方式

```
sqoop export --connect jdbc:mysql://192.168.137.10:3306/user_behavior --username root --password 123456 --table app_cource_study_report --columns watch_video_cnt,complete_video_cnt,dt --fields-terminated-by "\t" --export-dir "/user/hive/warehouse/tmp.db/app_cource_study_analysis_${day}" --staging-table app_cource_study_report_tmp --clear-staging-table --input-null-string '\N'
```

2）场景 2：设置 map 数量为 1 个（不推荐，面试官想要的答案不只这个）

多个 Map 任务时，采用 - staging-table 方式，仍然可以解决数据一致性问题。

5.8.3 Sqoop 底层运行的任务是什么

只有 Map 阶段，没有 Reduce 阶段的任务。

5.8.4 Sqoop 数据导出的时候一次执行多长时间

Sqoop 任务 5 分钟-2 个小时的都有。取决于数据量。

5.9 Scala 相关总结

5.9.1 元组

1） 元组的创建

```
val tuple1 = (1, 2, 3, "heiheihei")
println(tuple1)
```

2） 元组数据的访问，注意元组元素的访问有下划线，并且访问下标从 1 开始，而不是 0

```
val value1 = tuple1._4
println(value1)
```

3） 元组的遍历

```
方式 1：
for (elem <- tuple1.productIterator  ) {
   print(elem)
}
println()
方式 2：
tuple1.productIterator.foreach(i => println(i))
tuple1.produIterator.foreach(print(_))
```

5.9.2 隐式转换

隐式转换函数是以 implicit 关键字声明的带有单个参数的函数。这种函数将会自动应用，将值从一种类型转换为另一种类型。

```
implicit def a(d: Double) = d.toInt
//不加上边这句你试试
val i1: Int = 3.5
println(i1)
```

5.9.3 函数式编程理解

1）Scala 中函数的地位：一等公民

2） Scala 中的匿名函数(函数字面量)

3）Scala 中的高阶函数

4）Scala 中的闭包

5）Scala 中的部分应用函数

6）Scala 中的柯里化函数

5.9.4 样例类

case class Person(name:String,age:Int)

一般使用在 ds=df.as[Person]

5.9.5 柯里化

函数编程中，接受多个参数的函数都可以转化为接受单个参数的函数，这个转化过程就叫柯里化，柯里化就是证明了函数只需要一个参数而已。其实我们刚的学习过程中，已经涉及到了柯里化操作，所以这也印证了，柯里化就是以函数为主体这种思想发展的必然产生的结果。

1）柯里化的示例

```
def mul(x: Int, y: Int) = x * y
println(mul(10, 10))
def mulCurry(x: Int) = (y: Int) => x * y
println(mulCurry(10)(9))
def mulCurry2(x: Int)(y:Int) = x * y
println(mulCurry2(10)(8))
```

2）柯里化的应用

比较两个字符串在忽略大小写的情况下是否相等，注意，这里是两个任务：

- 全部转大写（或小写）
- 比较是否相等

针对这两个操作，我们用一个函数去处理的思想，其实无意间也变成了两个函数处理的思想。

示例如下：

```
val a = Array("Hello", "World")
val b = Array("hello", "world")
println(a.corresponds(b)(_.equalsIgnoreCase(_)))
其中 corresponds 函数的源码如下：
def corresponds[B](that: GenSeq[B])(p: (A,B) => Boolean): Boolean = {

val i = this.iterator

val j = that.iterator
```

```
while (i.hasNext && j.hasNex  t)

  if (!p(i.next(), j.next()    ))

    return fals
!i.hasNext && !j.hasNext
}
```

尖叫提示：不要设立柯里化存在义这样的命题，柯里化，是面向函数思想的必然产生结果。

5.9.6 闭包

一个函数把外部的那些不属于自己的对象也包含(闭合)进来。

案例 1：

```
def minusxy(x: Int) = (y: Int) => x - y
这就是一个闭包：
1) 匿名函数(y: Int) => x -y 嵌套在 minusxy 函数中。
2) 匿名函数(y: Int) => x -y 使用了该匿名函数之外的变量 x
3) 函数 minusxy 返回了引用了局部变量的匿名函数
```

案例 2

```
def minusxy(x: Int) = (y: Int) => x - y
val f1 = minusxy(10)
val f2 = minusxy(10)
println(f1(3) + f2(3))
此处 f1,f2 这两个函数就叫闭包。
```

5.9.7 Some、None、Option 的正确使用

```
val map = Map("Tom"-> 23)
map("Jack") // 抛出异常 java.util.NoSuchElementException: key not found: Jack
map.get("Jack") // None
map("Tom") // 23
map.get("Tom") // Some(23)

使用模式匹配取出最后结果
val optionAge = map.get("Tom")
val age = optionAge match {
 case Some(x) => optionAge.get
 case None => 0
}
```

5.10 Spark 相关总结

5.10.1 Spark 有几种部署方式？请分别简要论述

1) Local:运行在一台机器上，通常是练手或者测试环境。

2) Standalone:构建一个基于 Master+Slaves 的资源调度集群，Spark 任务提交给 Master 运行。是 Spark 自身的一个调度系统。

3) Yarn: Spark 客户端直接连接 Yarn，不需要额外构建 Spark 集群。有 yarn-client 和

yarn-cluster 两种模式，主要区别在于：Driver 程序的运行节点。

4) Mesos：国内大环境比较少用。

5.10.2 Spark 任务使用什么进行提交，javaEE 界面还是脚本

Shell 脚本。

5.10.3 Spark 提交作业参数（重点）

参考答案：

https://blog.csdn.net/gamer_gyt/article/details/79135118

1) 在提交任务时的几个重要参数

executor-cores —— 每个 executor 使用的内核数，默认为 1，官方建议 2-5 个，我们企业是 4 个

num-executors —— 启动 executors 的数量，默认为 2

executor-memory —— executor 内存大小，默认 1G

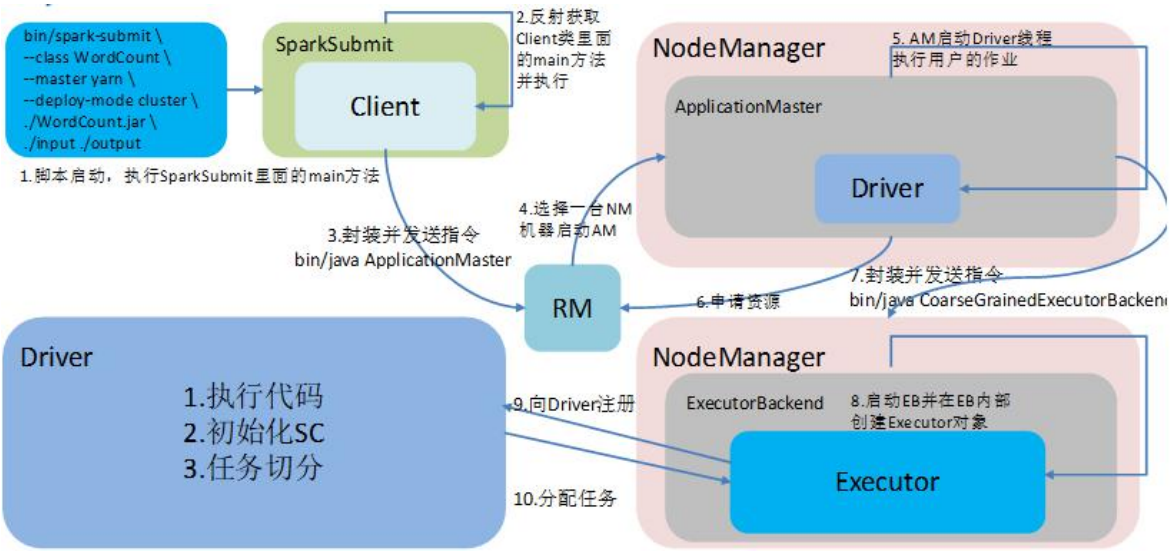
driver-cores —— driver 使用内核数，默认为 1

driver-memory —— driver 内存大小，默认 512M

2) 给一个提交任务的样式

```
spark-submit \
  --master local[5]  \
  --driver-cores 2   \
  --driver-memory 8g \
  --executor-cores 4 \
  --num-executors 10 \
  --executor-memory 8g \
  --class PackageName.ClassName XXXX.jar \
  --name "Spark Job Name" \
  InputPath      \
  OutputPath
```

5.10.4 简述 Spark 的架构与作业提交流程（画图讲解，注明各个部分的作用）（重点）



5.10.5 如何理解 Spark 中的血统概念（RDD）（笔试重点）

RDD 在 Lineage 依赖方面分为两种 Narrow Dependencies 与 Wide Dependencies 用来解决数据容错时的高效性以及划分任务时候起到重要作用。

5.10.6 简述 Spark 的宽窄依赖，以及 Spark 如何划分 stage，每个 stage 又根据什么决定 task 个数？（笔试重点）

Stage：根据 RDD 之间的依赖关系的不同将 Job 划分成不同的 Stage，遇到一个宽依赖则划分一个 Stage。

Task：Stage 是一个 TaskSet，将 Stage 根据分区数划分成一个个的 Task。

5.10.7 请列举 Spark 的 transformation 算子（不少于 8 个），并简述功能（重点）

1） map（func）：返回一个新的 RDD，该 RDD 由每一个输入元素经过 func 函数转换后组成.

2） mapPartitions(func)：类似于 map，但独立地在 RDD 的每一个分片上运行，因此在类型为 T 的 RD 上运行时，func 的函数类型必须是 Iterator[T] => Iterator[U]。假设有 N 个元素，有 M 个分区，那么 map 的函数的将被调用 N 次,而 mapPartitions 被调用 M 次,一个函数一次处理所有分区。

3） reduceByKey（func，[numTask]）：在一个(K,V)的 RDD 上调用，返回一个(K,V)的 RDD，使用定的 reduce 函数，将相同 key 的值聚合到一起，reduce 任务的个数可以通过第二个可选的参数来设置。

4) aggregateByKey (zeroValue:U,[partitioner: Partitioner]) (seqOp: (U, V) => U,combOp: (U, U) => U: 在 kv 对的 RDD 中，，按 key 将 value 进行分组合并，合并时，将每个 value 和初始值作为 seq 函数的参数，进行计算，返回的结果作为一个新的 kv 对，然后再将结果按照 key 进行合并，最后将每个分组的 value 传递给 combine 函数进行计算（先将前两个 value 进行计算，将返回结果和下一个 value 传给 combine 函数，以此类推），将 key 与计算结果作为一个新的 kv 对输出。

5) combineByKey(createCombiner: V=>C, mergeValue: (C, V) =>C, mergeCombiners: (C, C) =>C):

对相同 K，把 V 合并成一个集合。

a. createCombiner: combineByKey() 会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的某个元素的键相同。如果这是一个新的元素,combineByKey()会使用一个叫作 createCombiner()的函数来创建那个键对应的累加器的初始值

b. mergeValue: 如果这是一个在处理当前分区之前已经遇到的键，它会使用 mergeValue() 方法将该键的累加器对应的当前值与这个新的值进行合并

c. mergeCombiners: 由于每个分区都是独立处理的， 因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器， 就需要使用用户提供的 mergeCombiners() 方法将各个分区的结果进行合并。

…

根据自身情况选择比较熟悉的算子加以介绍。

5.10.8 请列举 Spark 的 action 算子（不少于 6 个），并简述功能（重点）

1) reduce：

2) collect:

3) first：

4) take：

5) aggregate：

6) countByKey：

7) foreach：

8） saveAsTextFile：

5.10.9 请列举会引起 Shuffle 过程的 Spark 算子，并简述功能。

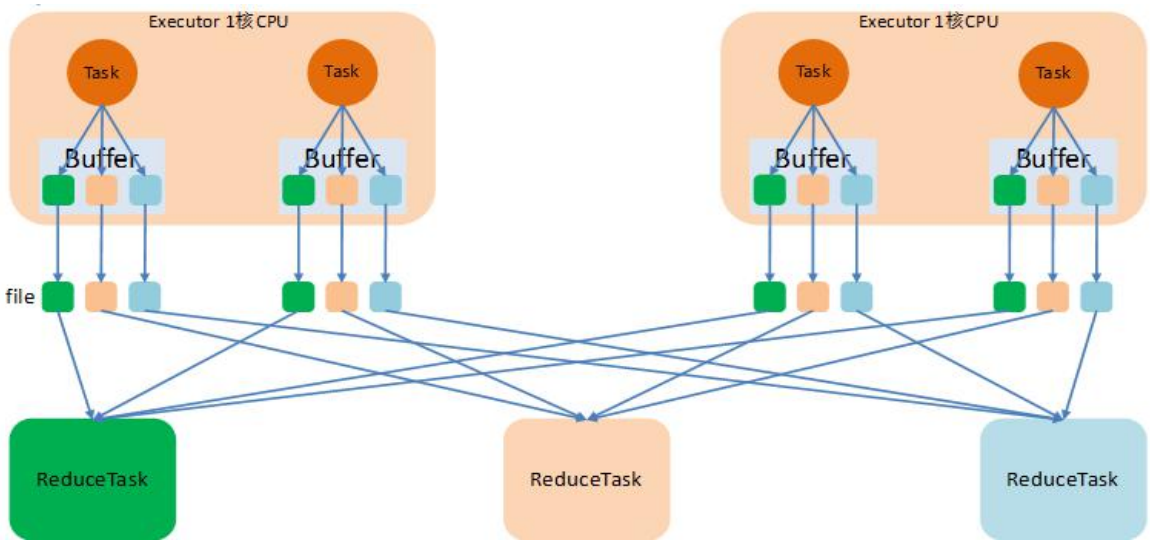
reduceBykey：

groupByKey：

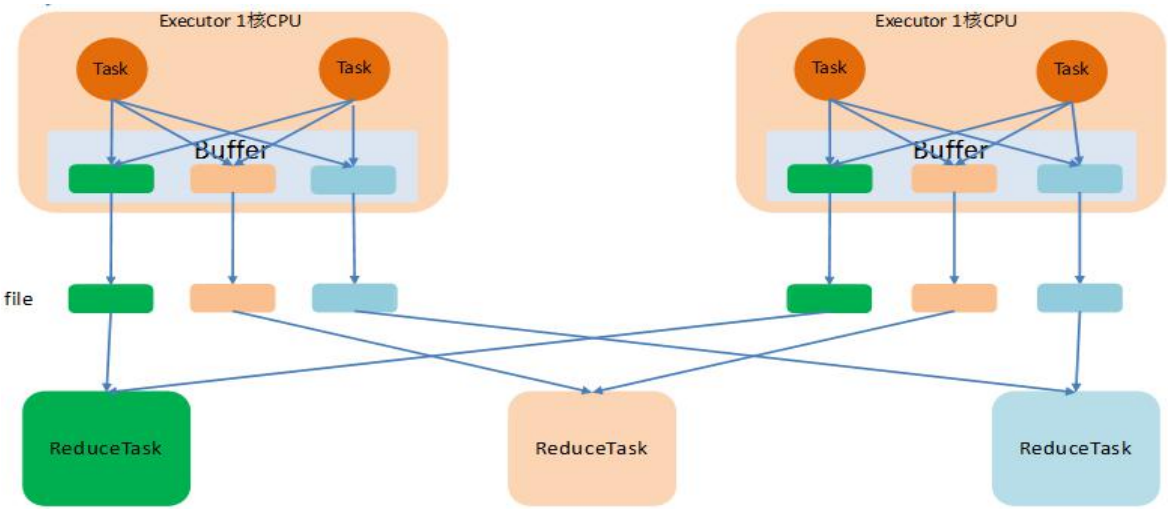
…ByKey:

5.10.10 简述 Spark 的两种核心 Shuffle（HashShuffle 与 SortShuffle）的工作流程（包括未优化的 HashShuffle、优化的 HashShuffle、普通的 SortShuffle 与 bypass 的 SortShuffle）（重点）

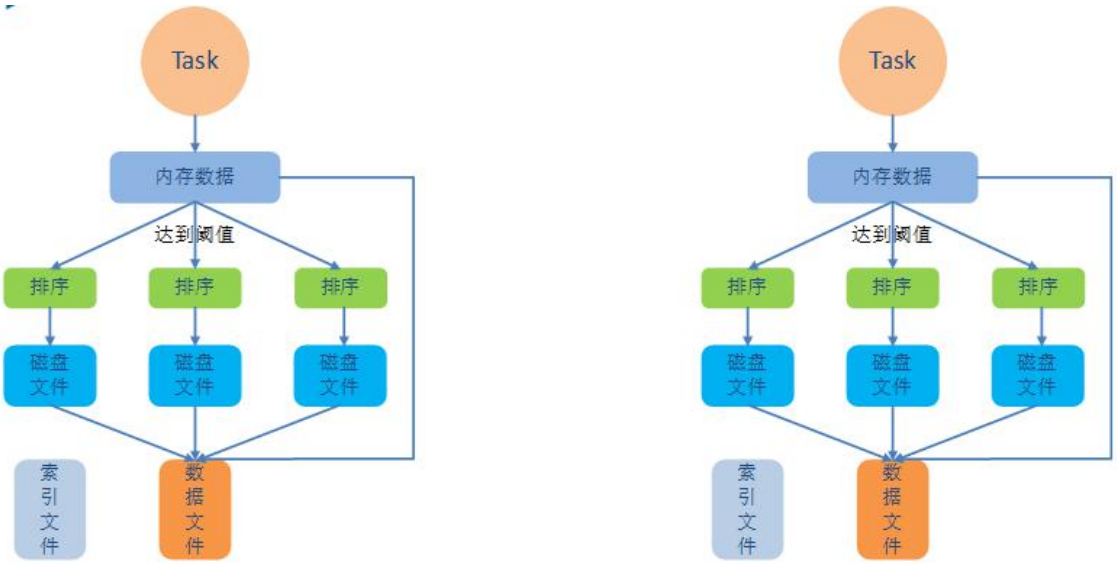
未经优化的 HashShuffle：



优化后的 Shuffle：



普通的 SortShuffle：



当 shuffle read task 的数量小于等于 spark.shuffle.sort。

bypassMergeThreshold 参数的值时（默认为 200），就会启用 bypass 机制。

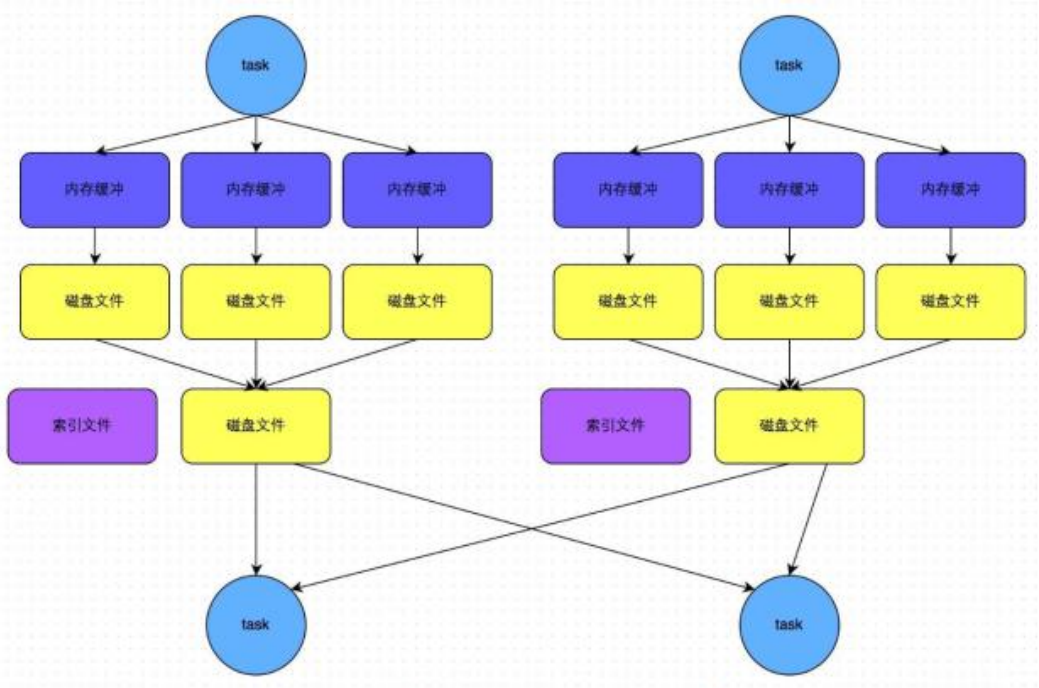


图 6-5 bypass 运行机制的 SortShuffleManager 工作原理

5.10.11 Spark 常用算子 reduceByKey 与 groupByKey 的区别，哪一种更具优势？（重点）

reduceByKey：按照 key 进行聚合，在 shuffle 之前有 combine（预聚合）操作，返回结果是 RDD[k,v]。

groupByKey：按照 key 进行分组，直接进行 shuffle。

开发指导：reduceByKey 比 groupByKey，建议使用。但是需要注意是否会影响业务逻辑。

5.10.12 Repartition 和 Coalesce 关系与区别

1) 关系：

两者都是用来改变 RDD 的 partition 数量的，repartition 底层调用的就是 coalesce 方法：coalesce(numPartitions, shuffle = true)

2) 区别：

repartition 一定会发生 shuffle，coalesce 根据传入的参数来判断是否发生 shuffle

一般情况下增大 rdd 的 partition 数量使用 repartition，减少 partition 数量时使用 coalesce

5.10.13 分别简述 Spark 中的缓存机制（cache 和 persist）与 checkpoint 机制，并指出两者的区别与联系

都是做 RDD 持久化的

cache:内存，不会截断血缘关系，使用计算过程中的数据缓存。

checkpoint：磁盘，截断血缘关系，在 ck 之前必须没有任何任务提交才会生效，ck 过程会额外提交一次任务。

5.10.14 简述 Spark 中共享变量（广播变量和累加器）的基本原理与用途。（重点）

累加器（accumulator）是 Spark 中提供的一种分布式的变量机制，其原理类似于 mapreduce，即分布式的改变，然后聚合这些改变。累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。而广播变量用来高效分发较大的对象。

共享变量出现的原因：

通常在向 Spark 传递函数时，比如使用 map() 函数或者用 filter() 传条件时，可以使用驱动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。

Spark 的两个共享变量，累加器与广播变量，分别为结果聚合与广播这两种常见的通信模式突破了这一限制。

5.10.15 当 Spark 涉及到数据库的操作时，如何减少 Spark 运行中的数据库连接数？

使用 foreachPartition 代替 foreach，在 foreachPartition 内获取数据库的连接。

5.10.16 简述 SparkSQL 中 RDD、DataFrame、DataSet 三者的区别与联系？（笔试重点）

1） RDD

优点:

编译时类型安全

编译时就能检查出类型错误

面向对象的编程风格

直接通过类名点的方式来操作数据

缺点:

序列化和反序列化的性能开销

无论是集群间的通信，还是 IO 操作都需要对对象的结构和数据进行序列化和反序列化。

GC 的性能开销，频繁的创建和销毁对象，势必会增加 GC

2） DataFrame

DataFrame 引入了 schema 和 off-heap

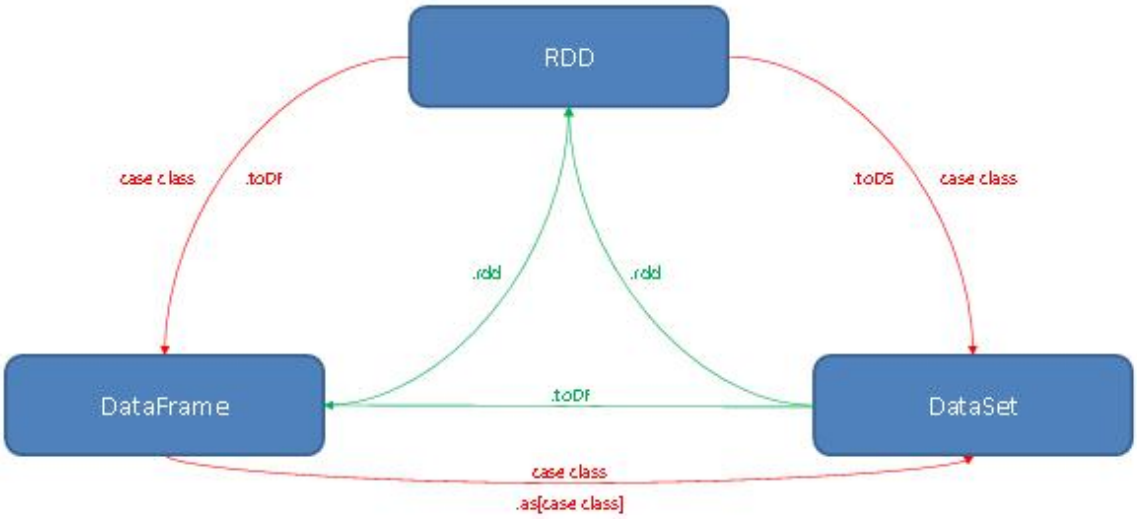
schema ： RDD 每一行的数据，结构都是一样的，这个结构就存储在 schema 中。 Spark 通过 schema 就能够读懂数据，因此在通信和 IO 时就只需要序列化和反序列化数据，而结构的部分就可以省略了。

3） DataSet

DataSet 结合了 RDD 和 DataFrame 的优点，并带来的一个新的概念 Encoder。

当序列化数据时，Encoder 产生字节码与 off-heap 进行交互，能够达到按需访问数据的效果，而不用反序列化整个对象。Spark 还没有提供自定义 Encoder 的 API，但是未来会加入。

三者之间的转换：



5.10.17 SparkSQL 中 join 操作与 left　join 操作的区别？

join 和 sql 中的 inner join 操作很相似，返回结果是前面一个集合和后面一个集合中匹配成功的，过滤掉关联不上的。

leftJoin 类似于 SQL 中的左外关联 left outer join，返回结果以第一个 RDD 为主，关联不上的记录为空。

部分场景下可以使用 left semi join 替代 left join：

因为 left semi join 是 in(keySet) 的关系，遇到右表重复记录，左表会跳过,性能更高，而 left join 则会一直遍历。但是 left semi join 中最后 select 的结果中只许出现左表中的列名，因为右表只有 join key 参与关联计算了

5.10.18 SparkStreaming 有哪几种方式消费 Kafka 中的数据，它们之间的区别是什么？　（重点）

一、基于 Receiver 的方式

这种方式使用 Receiver 来获取数据。Receiver 是使用 Kafka 的高层次 Consumer API 来实现的。receiver 从 Kafka 中获取的数据都是存储在 Spark Executor 的内存中的（如果突然数据暴增，大量 batch 堆积，很容易出现内存溢出的问题），然后 Spark Streaming 启动的 job 会去处理那些数据。

然而，在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用 Spark Streaming 的预写日志机制（Write Ahead Log，WAL）。

该机制会同步地将接收到的 Kafka 数据写入分布式文件系统（比如 HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复。

二、基于 Direct 的方式

这种新的不基于 Receiver 的直接方式，是在 Spark 1.3 中引入的，从而能够确保更加健壮的机制。替代掉使用 Receiver 来接收数据后，这种方式会周期性地查询 Kafka，来获得每个 topic+partition 的最新的 offset，从而定义每个 batch 的 offset 的范围。当处理数据的 job 启动时，就会使用 Kafka 的简单 consumer api 来获取 Kafka 指定 offset 范围的数据。

优点如下：

简化并行读取：如果要读取多个 partition，不需要创建多个输入 DStream 然后对它们进行 union 操作。Spark 会创建跟 Kafka partition 一样多的 RDD partition，并且会并行从 Kafka 中读取数据。所以在 Kafka partition 和 RDD partition 之间，有一个一对一的映射关系。

高性能：如果要保证零数据丢失，在基于 receiver 的方式中，需要开启 WAL 机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka 自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到 WAL 中。而基于 direct 的方式，不依赖 Receiver，不需要开启 WAL 机制，只要 Kafka 中作了数据的复制，那么就可以通过 Kafka 的副本进行恢复。

一次且仅一次的事务机制。

三、对比：

基于 receiver 的方式，是使用 Kafka 的高阶 API 来在 ZooKeeper 中保存消费过的 offset 的。这是消费 Kafka 数据的传统方式。这种方式配合着 WAL 机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为 Spark 和 ZooKeeper 之间可能是不同步的。

基于 direct 的方式，使用 kafka 的简单 api，Spark Streaming 自己就负责追踪消费的 offset，并保存在 checkpoint 中。Spark 自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。

在实际生产环境中大都用 Direct 方式

5.10.19 简述 SparkStreaming 窗口函数的原理（重点）

窗口函数就是在原来定义的 SparkStreaming 计算批次大小的基础上再次进行封装，每次计算多个批次的数据，同时还需要传递一个滑动步长的参数，用来设置当次计算任务完成之后下一

次从什么地方开始计算。

图中 time1 就是 SparkStreaming 计算批次大小，虚线框以及实线大框就是窗口的大小，必须为批次的整数倍。虚线框到大实线框的距离（相隔多少批次），就是滑动步长。

5.10.20 请手写出 wordcount 的 Spark 代码实现（Scala）（手写代码重点）

```
val conf: SparkConf = new SparkConf().setMaster("local[*]").setAppName("WordCount")
val sc = new SparkContext(conf)
sc.textFile("/input")
  .flatMap(_.split(" "))
  .map((_,1))
  .reduceByKey(_+_)
  .saveAsTextFile("/output")
  sc.stop()
```

5.10.21 如何使用 Spark 实现 topN 的获取（描述思路或使用伪代码）（重点）

方法 1：

（1）按照 key 对数据进行聚合（groupByKey）

（2）将 value 转换为数组，利用 scala 的 sortBy 或者 sortWith 进行排序（mapValues）数据量太大，会 OOM。

方法 2：

（1）取出所有的 key

（2）对 key 进行迭代，每次取出一个 key 利用 spark 的排序算子进行排序

方法 3：

（1）自定义分区器，按照 key 进行分区，使不同的 key 进到不同的分区

（2）对每个分区运用 spark 的排序算子进行排序

5.10.22 京东：调优之前与调优之后性能的详细对比（例如调整 map 个数，map 个数之前多少、之后多少，有什么提升）

这里举个例子。比如我们有几百个文件，会有几百个 map 出现，读取之后进行 join 操作，会非常的慢。这个时候我们可以进行 coalesce 操作，比如 240 个 map，我们合成 60 个 map，也

就是窄依赖。这样再 shuffle，过程产生的文件数会大大减少。提高 join 的时间性能。

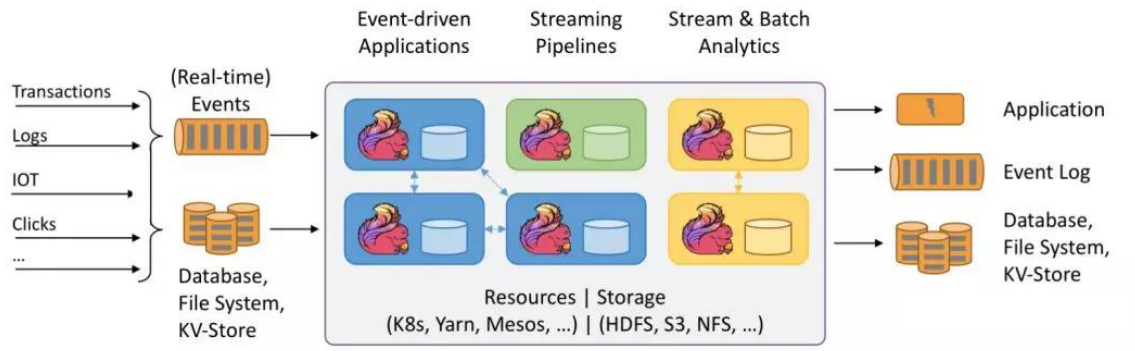
5.11Flink 相关总结

5.11.1 简单介绍一下 Flink

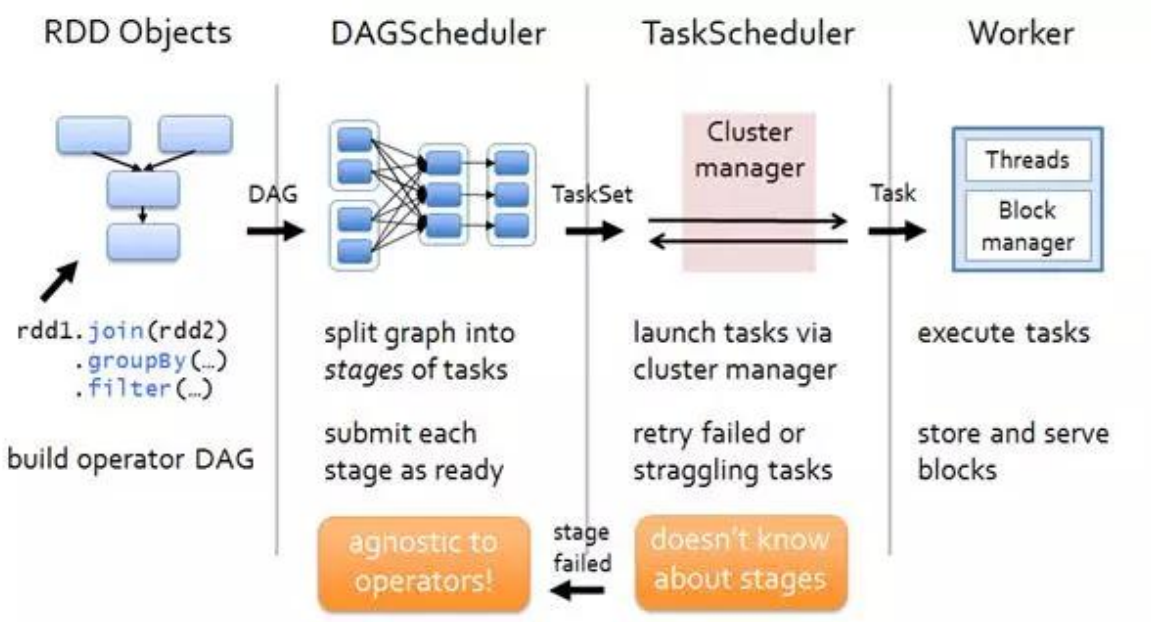
Flink 核心是一个流式的数据流执行引擎，其针对数据流的分布式计算提供了数据分布、数据通信以及容错机制等功能。基于流执行引擎，Flink 提供了诸多更高抽象层的 API 以便用户编写分布式任务：DataSet API， 对静态数据进行批处理操作，将静态数据抽象成分布式的数据集，用户可以方便地使用 Flink 提供的各种操作符对分布式数据集进行处理，支持 Java、Scala 和 Python。DataStream API，对数据流进行流处理操作，将流式的数据抽象成分布式的数据流，用户可以方便地对分布式数据流进行各种操作，支持 Java 和 Scala。Table API，对结构化数据进行查询操作，将结构化数据抽象成关系表，并通过类 SQL 的 DSL 对关系表进行各种查询操作，支持 Java 和 Scala。此外，Flink 还针对特定的应用领域提供了领域库，例如：Flink ML，Flink 的机器学习库，提供了机器学习 Pipelines API 并实现了多种机器学习算法。Gelly，Flink 的图计算库，提供了图计算的相关 API 及多种图计算算法实现。

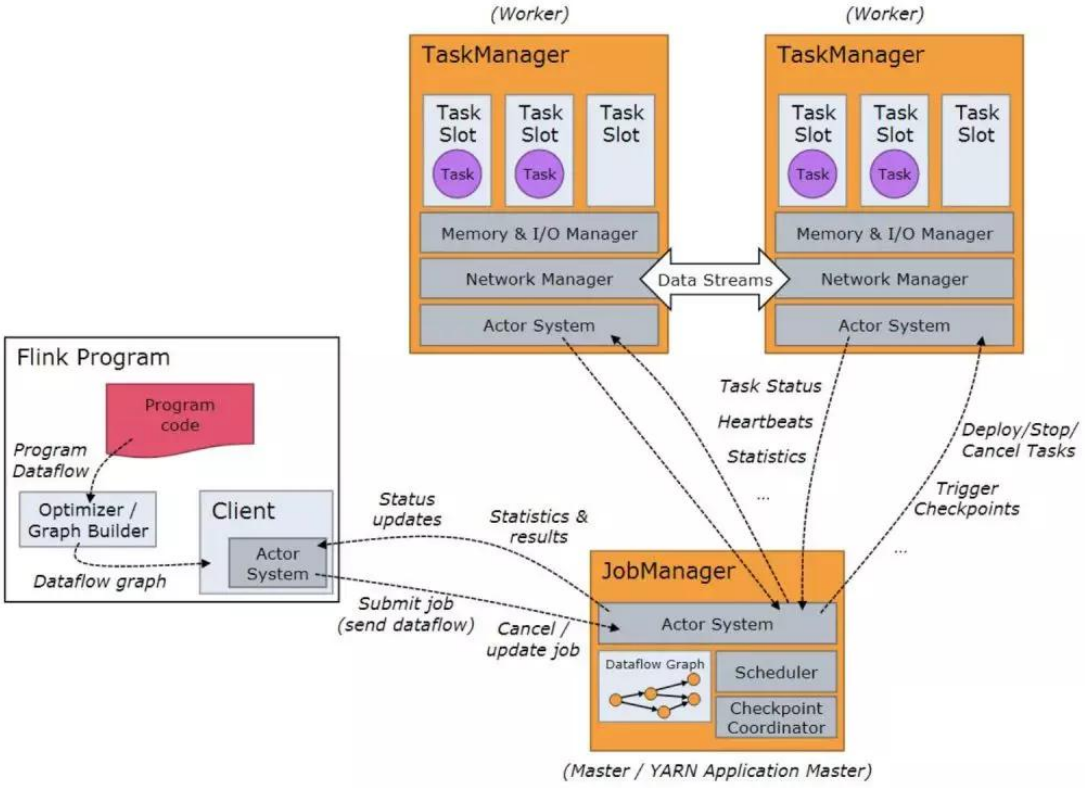
5.11.2 Flink 相比 Spark Streaming 有什么区别？

架构模型上:Spark Streaming 的 task 运行依赖 driver 和 executor 和 worker，当然 driver 和 excutor 还依赖于集群管理器 Standalone 或者 yarn 等。而 Flink 运行时主要是 JobManager、TaskManage 和 TaskSlot。另外一个最核心的区别是：Spark Streaming 是微批处理，运行的时候需要指定批处理的时间，每次运行 job 时处理一个批次的数据；Flink 是基于事件驱动的，事件可以理解为消息。事件驱动的应用程序是一种状态应用程序，它会从一个或者多个流中注入事件，通过触发计算更新状态，或外部动作对注入的事件作出反应。



任务调度上：Spark Streaming 的调度分为构建 DGA 图，划分 stage，生成 taskset，调度 task 等步骤，而 Flink 首先会生成 StreamGraph，接着生成 JobGraph，然后将 jobGraph 提交给 Jobmanager 由它完成 jobGraph 到 ExecutionGraph 的转变，最后由 jobManager 调度执行。



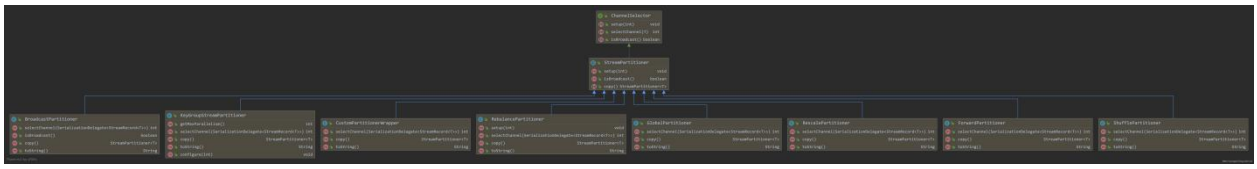


时间机制上：flink 支持三种时间机制事件时间，注入时间，处理时间，同时支持 watermark 机制处理滞后数据。Spark Streaming 只支持处理时间，Structured streaming 则支持了事件时间和 watermark 机制。

容错机制上：二者保证 exactly-once 的方式不同。spark streaming 通过保存 offset 和事务的方式；Flink 则使用两阶段提交协议来解决这个问题。

5.11.3 Flink 中的分区策略有哪几种？

Flink 中默认提供了八大分区策略(也叫分区器)。八大分区策略继承关系图



ChannelSelector: 接口，决定将记录写入哪个 Channel。有 3 个方法:

1) void setup(int numberOfChannels): 初始化输出 Channel 的数量。

2) int selectChannel(T record): 根据当前记录以及 Channel 总数，决定应将记录写入下游哪个 Channel。八大分区策略的区别主要在这个方法的实现上。

3) boolean isBroadcast(): 是否是广播模式。决定了是否将记录写入下游所有 Channel。

4) StreamPartitioner:抽象类，也是所有流分区器的基类。

注意:

这里以及下边提到的 Channel 可简单理解为下游 Operator 的某个实例。Flink 中改变并行度,默认 RebalancePartitioner 分区策略。分区策略,可在 Flink WebUI 上直观看出,如 REBALANCE,即使用了 RebalancePartitioner 分区策略;SHUFFLE,即使用了 ShufflePartitioner 分区策略。

1) GlobalPartitioner: DataStream => DataStream

GlobalPartitioner,GLOBAL 分区。将记录输出到下游 Operator 的第一个实例。

selectChannel 实现

```
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
     //对每条记录，只选择下游 operator 的第一个 Channel
     return 0;
}
```

API 使用

```
dataStream
    .setParallelism(2)
    // 采用 GLOBAL 分区策略重分区
    .global()
    .print()
    .setParallelism(1);
```

2) ShufflePartitioner: DataStream => DataStream

ShufflePartitioner，SHUFFLE 分区。将记录随机输出到下游 Operator 的每个实例。

selectChannel 实现

```
private Random random = new Random();
@Override
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
    //对每条记录，随机选择下游 operator 的某个 Channel
     return random.nextInt(numberOfChannels);
}
```

API 使用

```
dataStream
    .setParallelism(2)
    // 采用 SHUFFLE 分区策略重分区
    .shuffle()
```

```
    .print()
    .setParallelism(4);
```

3) RebalancePartitioner: DataStream => DataStream

RebalancePartitioner,REBALANCE 分区。将记录以循环的方式输出到下游 Operator 的每个实例。

selectChannel 实现

```
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
    //第一条记录，输出到下游的第一个 Channel;第二条记录，输出到下游的第二个 Channel...如此循环
      nextChannelToSendTo = (nextChannelToSendTo + 1) % numberOfChannels;
      return nextChannelToSendTo;
}
```

API 使用

```
dataStream
        .setParallelism(2)
        // 采用 REBALANCE 分区策略重分区
        .rebalance()
        .print()
        .setParallelism(4);
```

4) RescalePartitioner: DataStream => DataStream

RescalePartitioner,RESCALE 分区。基于上下游 Operator 的并行度，将记录以循环的方式输出到下游 Operator 的每个实例。举例：上游并行度是 2，下游是 4，则上游一个并行度以循环的方式将记录输出到下游的两个并行度上;上游另一个并行度以循环的方式将记录输出到下游另两个并行度上。若上游并行度是 4，下游并行度是 2，则上游两个并行度将记录输出到下游一个并行度上；上游另两个并行度将记录输出到下游另一个并行度上。

selectChannel 实现

```
private int nextChannelToSendTo = -1;
@Override
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
      if (++nextChannelToSendTo >= numberOfChannels) {
            nextChannelToSendTo = 0;
      }
      return nextChannelToSendTo;
}
```

API 使用

```
dataStream
    .setParallelism(2)
    // 采用 RESCALE 分区策略重分区
    .rescale()
    .print()
    .setParallelism(4);
```

5) BroadcastPartitioner: DataStream => DataStream

BroadcastPartitioner,BROADCAST 分区。广播分区将上游数据集输出到下游 Operator 的每个实例中。适合于大数据集 Join 小数据集的场景。

selectChannel 实现

```
@Override
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
    //广播分区不支持选择 Channel,因为会输出到下游每个 Channel 中
      throw new UnsupportedOperationException("Broadcast partitioner does not support select channels.");
}
@Override
public boolean isBroadcast() {
    //启用广播模式，此时 Channel 选择器会选择下游所有 Channel
      return true;
}
```

API 使用

```
dataStream
    .setParallelism(2)
    // 采用 BROADCAST 分区策略重分区
    .broadcast()
    .print()
    .setParallelism(4);
```

6) ForwardPartitioner

ForwardPartitioner,FORWARD 分区。将记录输出到下游本地的 operator 实例。ForwardPartitioner 分区器要求上下游算子并行度一样。上下游 Operator 同属一个 SubTasks。

selectChannel 实现

```
@Override
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
      return 0;
}
```

API 使用

```
dataStream
    .setParallelism(2)
    // 采用 FORWARD 分区策略重分区
    .forward()
    .print()
    .setParallelism(2);
```

7) KeyGroupStreamPartitioner(HASH 方式):

KeyGroupStreamPartitioner,HASH 分区。将记录按 Key 的 Hash 值输出到下游 Operator 实例。

selectChannel 实现

```
@Override
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
      K key;
      try {
            key = keySelector.getKey(record.getInstance().getValue());
      } catch (Exception e) {
            throw new RuntimeException("Could not extract key from " + record.getInstance().getValue(), e);
      }
      return KeyGroupRangeAssignment.assignKeyToParallelOperator(key, maxParallelism, numberOfChannels);
}
// KeyGroupRangeAssignment 中的方法
public static int assignKeyToParallelOperator(Object key, int maxParallelism, int parallelism) {
      return computeOperatorIndexForKeyGroup(maxParallelism, parallelism, assignToKeyGroup(key, maxParallelism));
}
// KeyGroupRangeAssignment 中的方法
public static int assignToKeyGroup(Object key, int maxParallelism) {
      return computeKeyGroupForKeyHash(key.hashCode(), maxParallelism);
}
// KeyGroupRangeAssignment 中的方法
public static int computeKeyGroupForKeyHash(int keyHash, int maxParallelism) {
      return MathUtils.murmurHash(keyHash) % maxParallelism;
}
```

API 使用

```
dataStream
    .setParallelism(2)
    // 采用 HASH 分区策略重分区
    .keyBy((KeySelector<Tuple3<String, Integer, String>, String>) value -> value.f0)
    .print()
    .setParallelism(4);
```

8) CustomPartitionerWrapper

CustomPartitionerWrapper,CUSTOM 分区。通过 Partitioner 实例的 partition 方法(自定义的)将记录输出到下游。

selectChannel 实现

```
Partitioner<K> partitioner;
KeySelector<T, K> keySelector;
public CustomPartitionerWrapper(Partitioner<K> partitioner, KeySelector<T, K> keySelector) {
      this.partitioner = partitioner;
      this.keySelector = keySelector;
}
@Override
public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
      K key;
      try {
            key = keySelector.getKey(record.getInstance().getValue());
      } catch (Exception e) {
            throw new RuntimeException("Could not extract key from " + record.getInstance(), e);
      }
      return partitioner.partition(key, numberOfChannels);
}
```

自定义分区器将指定的 Key 分到指定的分区

```
// 自定义分区器，将不同的 Key(用户 ID)分到指定的分区
// key: 根据 key 的值来分区
// numPartitions: 下游算子并行度
static class CustomPartitioner implements Partitioner<String> {
      @Override
      public int partition(String key, int numPartitions) {
          switch (key){
              case "user_1":
                  return 0;
              case "user_2":
                  return 1;
              case "user_3":
                  return 2;
              default:
                  return 3;
          }
      }
  }
```

使用自定义分区器

```
dataStream
```

```
.setParallelism(2)
// 采用 CUSTOM 分区策略重分区
.partitionCustom(new CustomPartitioner(),0)
.print()
.setParallelism(4);
```

5.11.4 Flink 的并行度有了解吗？Flink 中设置并行度需要注意什么？

Flink 程序由多个任务（Source、Transformation、Sink）组成。任务被分成多个并行实例来执行，每个并行实例处理任务的输入数据的子集。任务的并行实例的数量称之为并行度。Flink 中人物的并行度可以从多个不同层面设置：操作算子层面(Operator Level)、执行环境层面(Execution Environment Level)、客户端层面(Client Level)、系统层面(System Level)。Flink 可以设置好几个 level 的 parallelism,其中包括 Operator Level、Execution Environment Level、Client Level、System Level 在 flink-conf.yaml 中通过 parallelism.default 配置项给所有 execution environments 指定系统级的默认 parallelism；在 ExecutionEnvironment 里头可以通过 setParallelism 来给 operators、data sources、data sinks 设置默认的 parallelism；如果 operators、data sources、data sinks 自己有设置 parallelism 则会覆盖 ExecutionEnvironment 设置的 parallelism。

5.11.5 Flink 支持哪几种重启策略？分别如何配置？

重启策略种类:固定延迟重启策略(Fixed Delay Restart Strategy)故障率重启策略(Failure Rate Restart Strategy）无重启策略（No Restart Strategy）Fallback 重启策略（Fallback Restart Strategy）

5.11.6 Flink 的分布式缓存有什么作用？如何使用？

Flink 提供了一个分布式缓存，类似于 hadoop，可以使用户在并行函数中很方便的读取本地文件，并把它放在 taskmanager 节点中，防止 task 重复拉取。

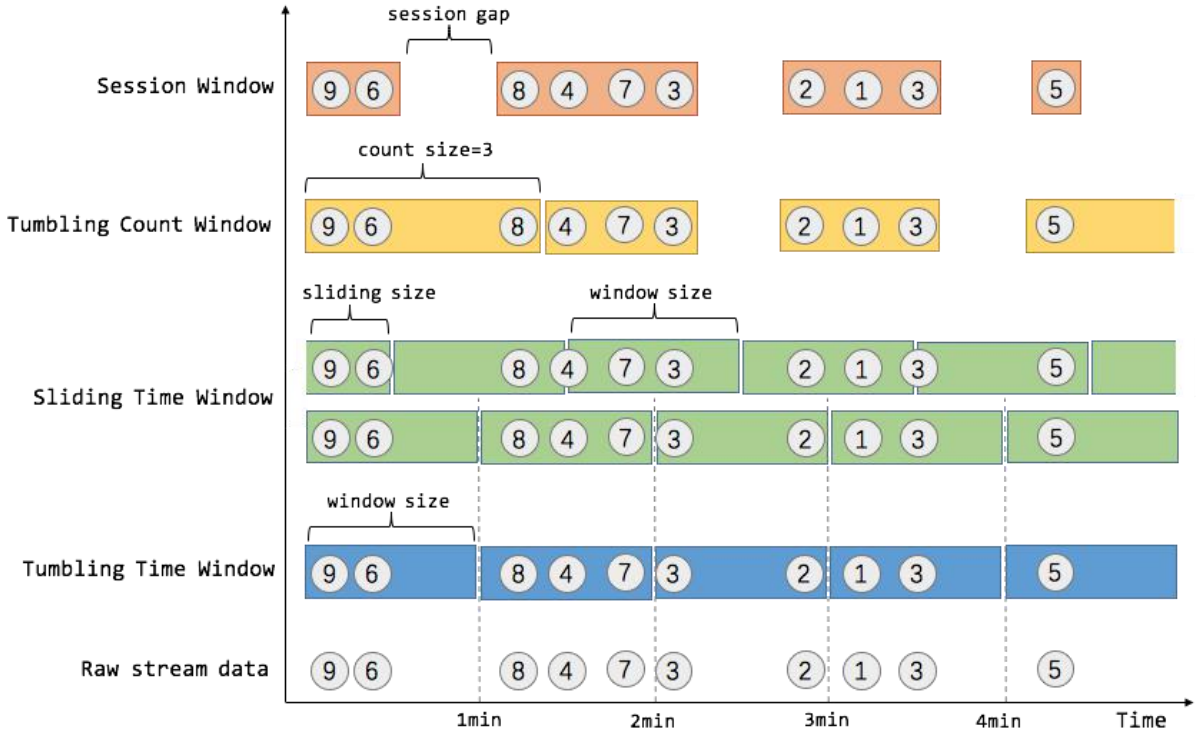
此缓存的工作机制如下：程序注册一个文件或者目录(本地或者远程文件系统，例如 hdfs 或者 s3)，通过 ExecutionEnvironment 注册缓存文件并为它起一个名称。

当程序执行，Flink 自动将文件或者目录复制到所有 taskmanager 节点的本地文件系统，仅会执行一次。用户可以通过这个指定的名称查找文件或者目录，然后从 taskmanager 节点的本地文件系统访问它。

5.11.7 Flink 中的广播变量，使用广播变量需要注意什么事项？

在 Flink 中，同一个算子可能存在若干个不同的并行实例，计算过程可能不在同一个 Slot 中进行，不同算子之间更是如此，因此不同算子的计算数据之间不能像 Java 数组之间一样互相访问，而广播变量 Broadcast 便是解决这种情况的。我们可以把广播变量理解为是一个公共的共享变量，我们可以把一个 dataset 数据集广播出去，然后不同的 task 在节点上都能够获取到，这个数据在每个节点上只会存在一份。

5.11.8 Flink 中对窗口的支持包括哪几种？说说他们的使用场景



1） Tumbling Time Window

假如我们需要统计每一分钟中用户购买的商品的总数，需要将用户的行为事件按每一分钟进行切分，这种切分被成为翻滚时间窗口（Tumbling Time Window）。翻滚窗口能将数据流切分成不重叠的窗口，每一个事件只能属于一个窗口。

2） Sliding Time Window

我们可以每 30 秒计算一次最近一分钟用户购买的商品总数。这种窗口我们称为滑动时间窗口（Sliding Time Window）。在滑窗中，一个元素可以对应多个窗口。

3） Tumbling Count Window

当我们想要每 100 个用户购买行为事件统计购买总数，那么每当窗口中填满 100 个元素了，

就会对窗口进行计算，这种窗口我们称之为翻滚计数窗口（Tumbling Count Window），上图所示窗口大小为 3 个。

4) Session Window

在这种用户交互事件流中，我们首先想到的是将事件聚合到会话窗口中（一段用户持续活跃的周期），由非活跃的间隙分隔开。如上图所示，就是需要计算每个用户在活跃期间总共购买的商品数量，如果用户 30 秒没有活动则视为会话断开（假设 raw data stream 是单个用户的购买行为流）。一般而言，window 是在无限的流上定义了一个有限的元素集合。这个集合可以是基于时间的，元素个数的，时间和个数结合的，会话间隙的，或者是自定义的。Flink 的 DataStream API 提供了简洁的算子来满足常用的窗口操作，同时提供了通用的窗口机制来允许用户自己定义窗口分配逻辑。

5.11.9 Flink 中的 State Backends 是什么？有什么作用？分成哪几类？说说他们各自的优缺点？

Flink 流计算中可能有各种方式来保存状态：

1) 窗口操作

2) 使用了 KV 操作的函数

3) 继承了 CheckpointedFunction 的函数

当开始做 checkpointing 的时候，状态会被持久化到 checkpoints 里来规避数据丢失和状态恢复。选择的状态存储策略不同，会导致状态持久化如何和 checkpoints 交互。

Flink 内部提供了这些状态后端:

1) MemoryStateBackend

2) FsStateBackend

3) RocksDBStateBackend

如果没有其他配置，系统将使用 MemoryStateBackend。

5.11.10 Flink 中的时间种类有哪些？各自介绍一下？

Flink 中的时间与现实世界中的时间是不一致的，在 flink 中被划分为事件时间，摄入时间，处理时间三种。如果以 EventTime 为基准来定义时间窗口将形成 EventTimeWindow,要求消息本身就应该携带 EventTime 如果以 IngesingtTime 为基准来定义时间窗口将形成 IngestingTimeWindow,以 source 的 systemTime 为准。如果以 ProcessingTime 基准来定义时间

窗口将形成 ProcessingTimeWindow，以 operator 的 systemTime 为准。

5.11.11 WaterMark 是什么？是用来解决什么问题？如何生成水印？水印的原理是什么？

Watermark 是 Apache Flink 为了处理 EventTime 窗口计算提出的一种机制,本质上也是一种时间戳。watermark 是用于处理乱序事件的，处理乱序事件通常用 watermark 机制结合 window 来实现。详细参考：https://www.jianshu.com/p/1c2542f11da0

5.11.12 Flink 的 table 和 SQL 熟悉吗？Table API 和 SQL 中 TableEnvironment 这个类有什么作用

TableEnvironment 是 Table API 和 SQL 集成的核心概念。它负责：

A) 在内部 catalog 中注册表

B) 注册外部 catalog

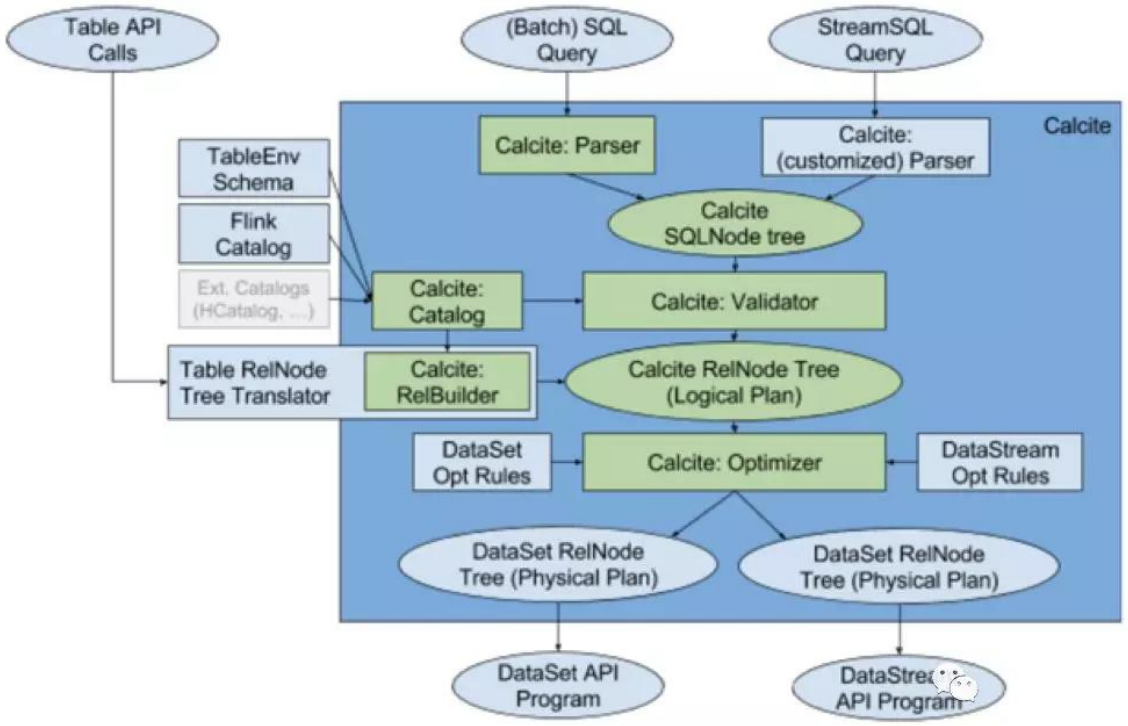
C) 执行 SQL 查询

D) 注册用户定义（标量，表或聚合）函数

E) 将 DataStream 或 DataSet 转换为表

F) 持有对 ExecutionEnvironment 或 StreamExecutionEnvironment 的引用

5.11.13 Flink 如何实现 SQL 解析的呢？



StreamSQL API 的执行原理如下：

1、用户使用对外提供 Stream SQL 的语法开发业务应用；

2、用 calcite 对 StreamSQL 进行语法检验，语法检验通过后，转换成 calcite 的逻辑树节点；最终形成 calcite 的逻辑计划；

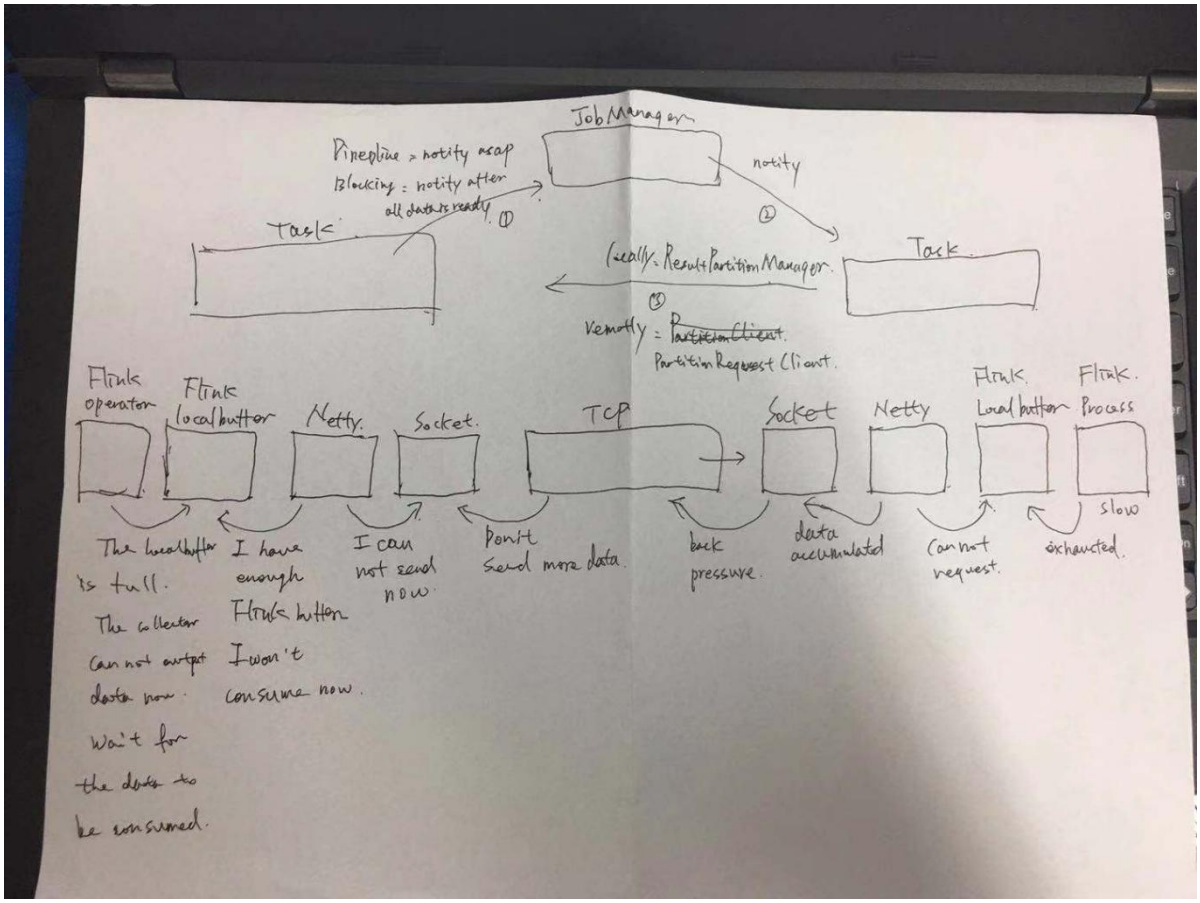
3、采用 Flink 自定义的优化规则和 calcite 火山模型、启发式模型共同对逻辑树进行优化，生成最优的 Flink 物理计划；

4、对物理计划采用 janino codegen 生成代码，生成用低阶 API DataStream 描述的流应用，提交到 Flink 平台执行详细参考：https://cloud.tencent.com/developer/article/1471612

5.11.14 Flink 是如何做到批处理与流处理统一的？

Flink 设计者认为：有限流处理是无限流处理的一种特殊情况，它只不过在某个时间点停止而已。Flink 通过一个底层引擎同时支持流处理和批处理。详细参考：https://cloud.tencent.com/developer/article/1501348

5.11.15 Flink 中的数据传输模式是怎么样的？



大概的原理，上游的 task 产生数据后，会写在本地的缓存中，然后通知 JM 自己的数据已经好了，JM 通知下游的 Task 去拉取数据，下游的 Task 然后去上游的 Task 拉取数据，形成链条。

但是在何时通知 JM？这里有一个设置，比如 pipeline 还是 blocking，pipeline 意味着上游哪怕产生一个数据，也会去通知，blocking 则需要缓存的插槽存满了才会去通知，默认是 pipeline。

虽然生产数据的是 Task，但是一个 TaskManager 中的所有 Task 共享一个 NetworkEnvironment，下游的 Task 利用 ResultPartitionManager 主动去上游 Task 拉数据，底层利用的是 Netty 和 TCP 实现网络链路的传输。

那么，一直都在说 Flink 的背压是一种自然的方式，为什么是自然的了？

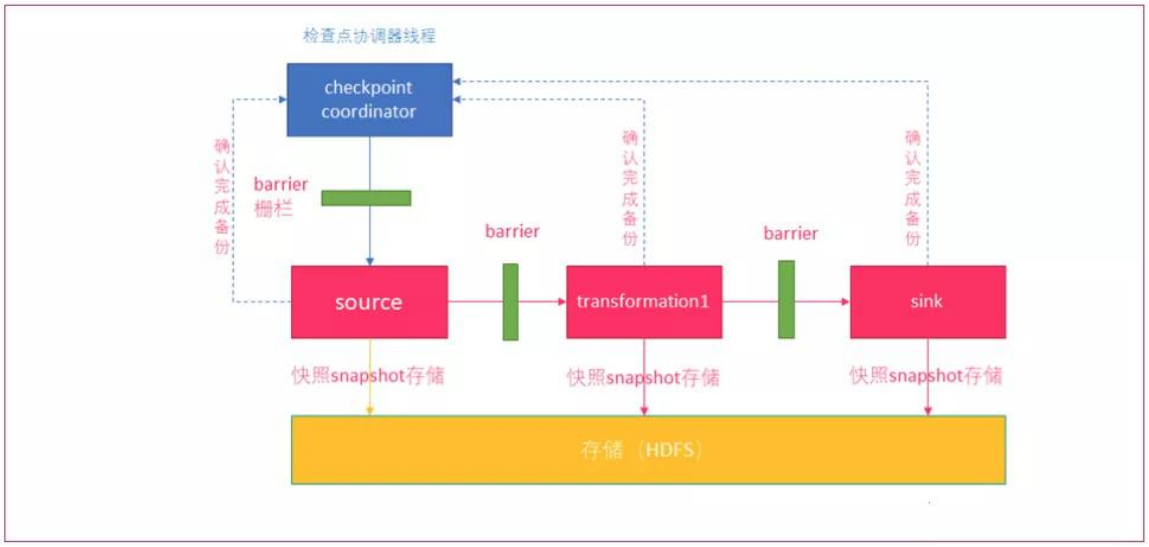
从上面的图中下面的链路中可以看到，当下游的 process 逻辑比较慢，无法及时处理数据时，他自己的 local buffer 中的消息就不能及时被消费，进而导致 netty 无法把数据放入 local buffer，进而 netty 也不会去 socket 上读取新到达的数据，进而在 tcp 机制中，tcp 也不会从上游的 socket 去读取新的数据，上游的 netty 也是一样的逻辑，它无法发送数据，也就不能从上游的 localbuffer 中消费数据，所以上游的 localbuffer 可能就是满的，上游的 operator 或者

process 在处理数据之后进行 collect.out 的时候申请不能本地缓存，导致上游的 process 被阻塞。这样，在这个链路上，就实现了背压。

如果还有相应的上游，则会一直反压上去，一直影响到 source，导致 source 也放慢从外部消息源读取消息的速度。一旦瓶颈解除，网络链路畅通，则背压也会自然而然的解除。

5.11.16 Flink 的容错机制

Flink 基于分布式快照与可部分重发的数据源实现了容错。用户可自定义对整个 Job 进行快照的时间间隔，当任务失败时，Flink 会将整个 Job 恢复到最近一次快照，并从数据源重发快照之后的数据。



详细参考：https://www.jianshu.com/p/1fca8fb61f86

5.11.17 Flink 在使用 Window 时出现数据倾斜，你有什么解决办法？

注意：这里 window 产生的数据倾斜指的是不同的窗口内积攒的数据量不同，主要是由源头数据的产生速度导致的差异。核心思路：1.重新设计 key 2.在窗口计算前做预聚合可以参考这个：https://blog.csdn.net/it_lee_j_h/article/details/88641894

5.11.18 Flink 任务，delay 极高，请问你有什么调优策略？

首先要确定问题产生的原因，找到最耗时的点，确定性能瓶颈点。比如任务频繁反压，找到反压点。主要通过：资源调优、作业参数调优。资源调优即是对作业中的 Operator 的并发数（parallelism）、CPU（core）、堆内存（heap_memory）等参数进行调优。作业参数调优包括：

并行度的设置，State 的设置，checkpoint 的设置。

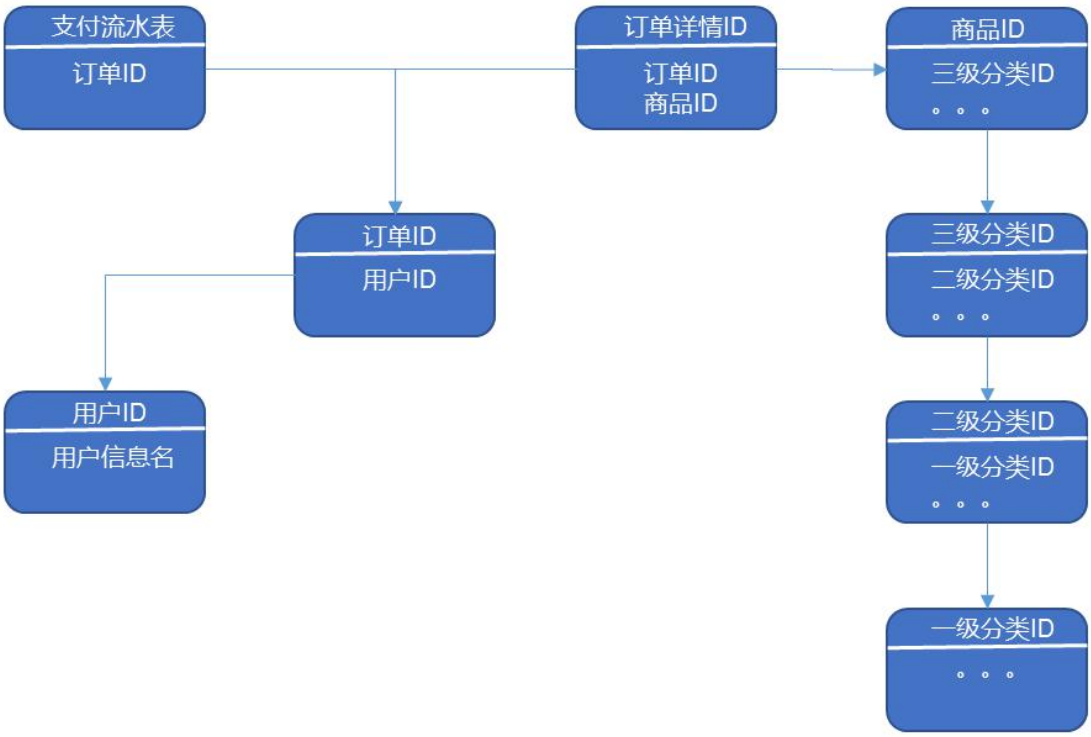
第 6 章 业务交互数据分析

6.1 电商常识

SKU（库存量单位）：一台银色、128G 内存的、支持联通网络的 iPhoneX

SPU（标准产品单位）：iPhoneX

6.2 电商业务流程



6.3 业务表关键字段

6.3.1 订单表（order_info）

标签	含义
id	订单编号
total_amount	订单金额

order_status	订单状态
user_id	用户 id
payment_way	支付方式
out_trade_no	支付流水号
create_time	创建时间
operate_time	操作时间

6.3.2 订单详情表（order_detail）

标签	含义
id	订单编号
order_id	订单号
user_id	用户 id
sku_id	商品 id
sku_name	商品名称
order_price	商品价格
sku_num	商品数量
create_time	创建时间

6.3.3 商品表

标签	含义
id	skuId
spu_id	spuid
price	价格
sku_name	商品名称
sku_desc	商品描述
weight	重量
tm_id	品牌 id
category3_id	品类 id
create_time	创建时间

6.3.4 用户表

标签	含义
id	用户 id
name	姓名
birthday	生日
gender	性别
email	邮箱
user_level	用户等级
create_time	创建时间

6.3.5 商品一级分类表

标签	含义

id	id
name	名称

6.3.6 商品二级分类表

标签	含义
id	id
name	名称
category1_id	一级品类 id

6.3.7 商品三级分类表

标签	含义
id	id
name	名称
Category2_id	二级品类 id

6.3.8 支付流水表

标签	含义
id	编号
out_trade_no	对外业务编号
order_id	订单编号
user_id	用户编号
alipay_trade_no	支付宝交易流水编号
total_amount	支付金额
subject	交易内容
payment_type	支付类型
payment_time	支付时间

订单表跟订单详情表有什么区别？

订单表的订单状态会变化，订单详情表不会，因为没有订单状态。

订单表记录 user_id，订单 id 订单编号，订单的总金额 order_status，支付方式，订单状态等。

订单详情表记录 user_id，商品 sku_id ，具体的商品信息（商品名称 sku_name，价格 order_price，数量 sku_num）

6.4 MySql 中表的分类

实体表，维度表，事务型事实表，周期性事实表

其实最终可以把事务型事实表，周期性事实表统称实体表，实体表，维度表统称维度表

订单表（order_info）（周期型事实表）

订单详情表（order_detail）(事务型事实表)

商品表(实体表)

用户表(实体表)

商品一级分类表(维度表)

商品二级分类表(维度表)

商品三级分类表(维度表)

支付流水表(事务型实体表)

6.5 同步策略

序号	数据库中表名	表中文名称	同步策略
1	order_info	订单表	新增及变化
2	order_detail	订单详情	增量
3	sku_info	商品表	全量
4	user_info	用户表	全量
5	base_category1	商品一级分类表	全量
6	base_category2	商品二级分类表	全量
7	base_category3	商品三级分类表	全量
8	payment_info	支付流水表	增量

实体表，维度表统称维度表，每日全量或者每月（更长时间）全量

事务型事实表：每日增量

周期性事实表：拉链表

6.6 关系型数据库范式理论

1NF：属性不可再分割（例如不能存在 5 台电脑的属性，坏处：表都没法用）

2NF：不能存在部分函数依赖（例如主键（学号+课名）-->成绩，姓名，但学号--》姓名，所以姓名部分依赖于主键（学号+课名），所以要去除，坏处：数据冗余）

3NF：不能存在传递函数依赖（学号--》宿舍种类--》价钱，坏处：数据冗余和增删异常）

Mysql 关系模型：关系模型主要应用与 OLTP 系统中，为了保证数据的一致性以及避免冗余，所以大部分业务系统的表都是遵循第三范式的。

Hive 维度模型：维度模型主要应用于 OLAP 系统中，因为关系模型虽然冗余少，但是在大规模数据，跨表分析统计查询过程中，会造成多表关联，这会大大降低执行效率。所以 HIVE 把相关各种表整理成两种：事实表和维度表两种。所有维度表围绕着事实表进行解释。

6.7 数据模型

雪花模型、星型模型和星座模型

（在维度建模的基础上又分为三种模型：星型模型、雪花模型、星座模型。）

星型模型（一级维度表），雪花（多级维度），星座模型（星型模型+多个事实表）

6.8 业务数据数仓搭建

sqoop

导数据的原理是 mapreduce,

import 把数据从关系型数据库 导到 数据仓库，自定义 InputFormat，

export 把数据从数据仓库 导到 关系型数据库，自定义 OutputFormat，

用 sqoop 从 mysql 中将八张表的数据导入数仓的 ods 原始数据层

全量无条件，增量按照创建时间，增量+变化按照创建时间或操作时间。

origin_data

sku_info 商品表（每日导全量）

user_info 用户表（每日导全量）

base_category1 商品一级分类表（每日导全量）

base_category2 商品二级分类表（每日导全量）

base_category3 商品三级分类表（每日导全量）

order_detail 订单详情表（每日导增量）

payment_info 支付流水表（每日导增量）

order_info 订单表（每日导增量+变化）

6.8.1 ods 层

（八张表，表名，字段跟 mysql 完全相同）

从 origin_data 把数据导入到 ods 层，表名在原表名前加 ods_

6.8.2 dwd 层

对 ODS 层数据进行判空过滤。对商品分类表进行维度退化(降维)。其他数据跟 ods 层一模一样

订单表 dwd_order_info

订单详情表 dwd_order_detail

用户表 dwd_user_info

支付流水表 dwd_payment_info

商品表 dwd_sku_info

其他表字段不变，唯独商品表，通过关联 3 张分类表，增加了

```
category2_id` string COMMENT '2id',
`category1_id` string COMMENT '3id',
`category3_name` string COMMENT '3',
`category2_name` string COMMENT '2',
`category1_name` string COMMENT '1',
```

小结：

1) 维度退化要付出什么代价？或者说会造成什么样的需求处理不了？

如果被退化的维度，还有其他业务表使用，退化后处理起来就麻烦些。

还有如果要删除数据，对应的维度可能也会被永久删除。

2) 想想在实际业务中还有那些维度表可以退化

城市的三级分类（省、市、县）等

6.8.3 dws 层

从订单表 dwd_order_info 中获取 下单次数 和 下单总金额

从支付流水表 dwd_payment_info 中获取 支付次数 和 支付总金额

从事件日志评论表 dwd_comment_log 中获取评论次数

最终按照 user_id 聚合，获得明细，跟之前的 mid_id 聚合不同

6.9 需求一：GMV 成交总额

从用户行为宽表中 dws_user_action，根据统计日期分组，聚合，直接 sum 就可以了。

6.10 需求二：转化率

6.10.1 新增用户占日活跃用户比率表

从日活跃数表 ads_uv_count 和 日新增设备数表 ads_new_mid_count 中取即可。

6.10.2 用户行为转化率表

从用户行为宽表 dws_user_action 中取，下单人数（只要下单次数>0）,支付人数（只要支付次数>0）

从日活跃数表 ads_uv_count 中取活跃人数，然后对应的相除就可以了。

6.11 需求三：品牌复购率

需求：以月为单位统计，购买 2 次以上商品的用户

6.11.1 用户购买商品明细表（宽表）

6.11.2 品牌复购率表

从用户购买商品明细宽表 dws_sale_detail_daycount 中，根据品牌 id--sku_tm_id 聚合，计算每个品牌购买的总次数，购买人数 a=购买次数>=1，两次及以上购买人数 b=购买次数>=2，三次及以上购买人数 c=购买次数>=3，单次复购率=b/a，多次复购率=c/a

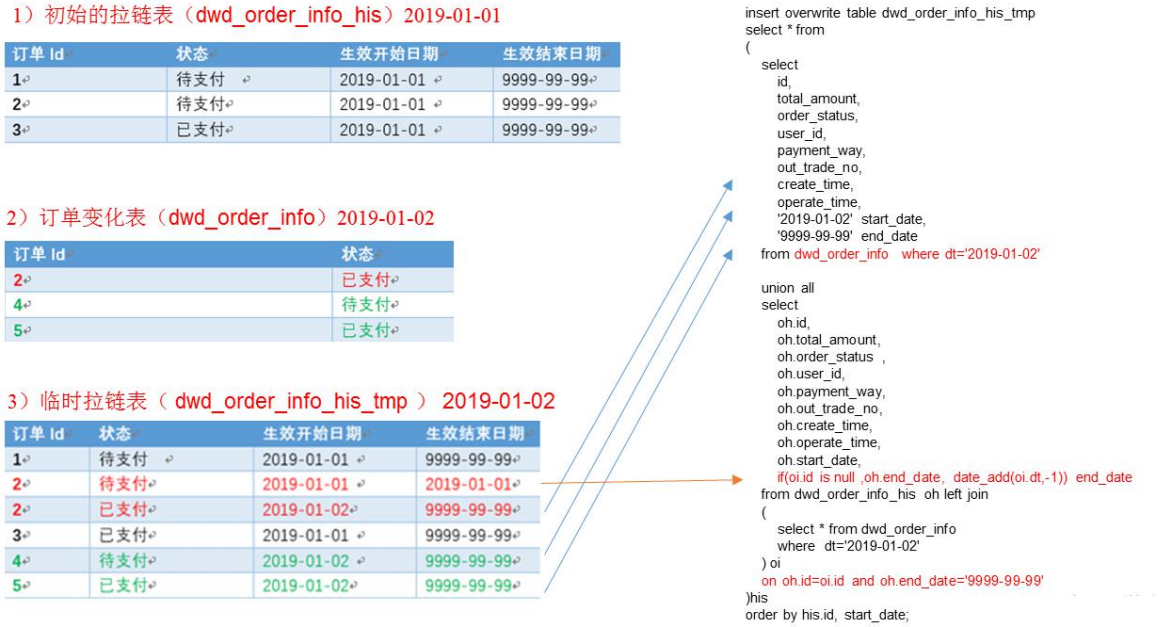
6.12 项目中有多少张宽表

宽表要 3-5 张，用户行为宽表，用户购买商品明细行为宽表，商品宽表，购物车宽表，物流宽表、登录注册、售后等。

1） 为什么要建宽表

需求目标，把每个用户单日的行为聚合起来组成一张多列宽表，以便之后关联用户维度信息后进行，不同角度的统计分析。

6.13 拉链表



订单表拉链表 dwd_order_info_his

```
`id` string COMMENT '订单编号',
`total_amount` decimal(10,2) COMMENT '订单金额',
`order_status` string COMMENT '订单状态',
`user_id` string COMMENT '用户 id' ,
```

```
`payment_way` string COMMENT '支付方式',
`out_trade_no` string COMMENT '支付流水号',
`create_time` string COMMENT '创建时间',
`operate_time` string COMMENT '操作时间' ,
`start_date`  string COMMENT '有效开始日期',
`end_date`  string COMMENT '有效结束日期'
```

1）创建订单表拉链表，字段跟拉链表一样，只增加了有效开始日期和有效结束日期

初始日期，从订单变化表 ods_order_info 导入数据，且让有效开始时间=当前日期，有效结束日期=9999-99-99

（从 mysql 导入数仓的时候就只导了新增的和变化的数据 ods_order_info,dwd_order_info 跟 ods_order_info 基本一样，只多了一个 id 的判空处理）

2）建一张拉链临时表 dwd_order_info_his_tmp，字段跟拉链表完全一致

3）新的拉链表中应该有这几部分数据，

（1）增加订单变化表 dwd_order_info 的全部数据

（2）更新旧的拉链表左关联订单变化表 dwd_order_info，关联字段：订单 id, where 过滤出 end_date 只等于 9999-99-99 的数据，如果旧的拉链表中的 end_date 不等于 9999-99-99，说明已经是终态了，不需要再更新

如果 dwd_order_info.id is null ， 没关联上，说明数据状态没变，让 end_date 还等于旧的 end_date

如果 dwd_order_info.id is not null ， 关联上了，说明数据状态变了，让 end_date 等于当前日期-1

把查询结果插入到拉链临时表中

4）把拉链临时表覆盖到旧的拉链表中

第 7 章 项目中遇到过哪些问题

7.1 Hadoop 宕机

1） 如果 MR 造成系统宕机。此时要控制 Yarn 同时运行的任务数，和每个任务申请的最大内存。

调整参数：yarn.scheduler.maximum-allocation-mb（单个任务可申请的最多物理内存量，默认是8192MB）

2） 如果写入文件过量造成NameNode宕机。那么调高Kafka的存储大小，控制从Kafka到HDFS的写入速度。高峰期的时候用Kafka进行缓存，高峰期过去数据同步会自动跟上。

7.2 Ganglia监控

Ganglia监控Flume发现尝试提交的次数大于最终成功的次数

1） 增加Flume内存

2） 增加Flume台数

7.3 Flume小文件

Flume上传文件到HDFS时参数大量小文件?

调整hdfs.rollInterval、hdfs.rollSize、hdfs.rollCount这三个参数的值。

7.4 Kafka挂掉

1） Flume记录

2） 日志有记录

3） 短期没事

7.5 Kafka消息数据积压，Kafka消费能力不足怎么处理？

1） 如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）

2） 如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

7.6 Kafka数据重复

在下一级消费者中去重。（redis、SparkStreaming）

7.7 Mysql 高可用

Hive 的 metadata 存储在 MySql 中（配置 MySql 的高可用（主从复制和读写分离和故障转移））

7.8 自定义 UDF 和 UDTF 解析和调试复杂字段

自定义 UDF（extends UDF 实现 evaluate 方法） 解析公共字段

自定义 UDTF(extends Genertic UDTF->实现三个方法 init(指定返回值的名称和类型)、process(处理字段一进多出)、close 方法) -> 更加灵活以及方便定义 bug

7.9 Sqoop 数据导出 Parquet

Ads 层数据用 Sqoop 往 MySql 中导入数据的时候，如果用了 orc（Parquet）不能导入，需转化成 text 格式

7.10 Sqoop 数据导出控制

Sqoop 中导入导出 Null 存储一致性问题：

Hive 中的 Null 在底层是以“\N”来存储，而 MySQL 中的 Null 在底层就是 Null，为了保证数据两端的一致性。在导出数据时采用--input-null-string 和 --input-null-non-string 两个参数。导入数据时采用--null-string 和 --null-non-string。

7.11 Sqoop 数据导出一致性问题

当 Sqoop 导出数据到 MySql 时，使用 4 个 map 怎么保证数据的一致性

因为在导出数据的过程中 map 任务可能会失败，可以使用—staging-table –clear-staging

```
sqoop export --connect jdbc:mysql://192.168.137.10:3306/user_behavior --username root --password 123456 --table app_cource_study_report --columns watch_video_cnt,complete_video_cnt,dt --fields-terminated-by "\t" --export-dir "/user/hive/warehouse/tmp.db/app_cource_study_analysis_${day}" --staging-table app_cource_study_report_tmp --clear-staging-table --input-null-string '\N'
```

任务执行成功首先在 tmp 临时表中，然后将 tmp 表中的数据复制到目标表中（这个时候可以使用事务，保证事务的一致性）

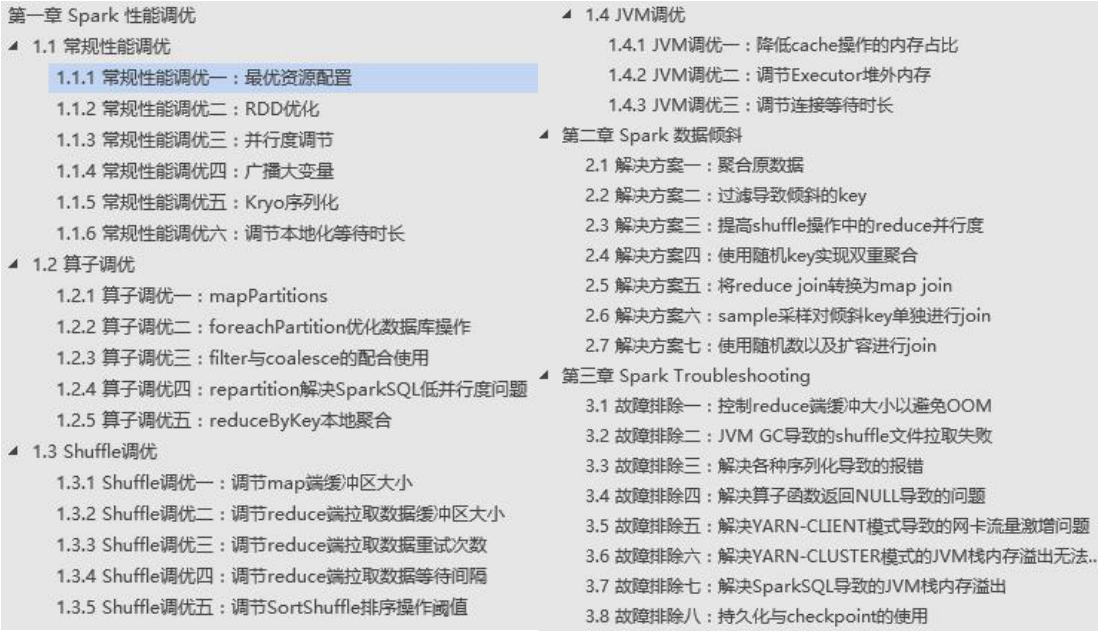
7.12 SparkStreaming 优雅关闭

如何优雅的关闭 SparkStreaming 任务（将写好的代码打包，Spark-Submit）

Kill -9 xxx ？

开启另外一个线程每 5 秒监听 HDFS 上一个文件是否存在。如果检测到存在，调用 ssc.stop()方法关闭 SparkStreaming 任务（当你要关闭任务时，可以创建你自定义监控的文件目录）

7.13 Spark OOM、数据倾斜解决



第 8 章 项目经验

8.1 框架经验

8.1.1 Hadoop

1） Hadoop 集群基准测试（HDFS 的读写性能、MapReduce 的计算能力测试）

2） 一台服务器一般都有很多个硬盘插槽（插了几个插槽）

如果不配置 datanode.data.dir 多目录，每次插入一块新的硬盘都需要重启服务器

配置了即插即用

3） Hdfs 参数调优

Namenode 有一个工作线程池，用来处理与 datanode 的心跳（报告自身的健康状况和文件恢复请求）和元数据请求 dfs.namenode.handler.count=20 * log2(Cluster Size)

4）编辑日志存储路径 dfs.namenode.edits.dir 设置与镜像文件存储路径

dfs.namenode.name.dir 尽量分开，达到最低写入延迟（提高写入的吞吐量）

5）YARN 参数调优 yarn-site.xml

（1）服务器节点上 YARN 可使用的物理内存总量，默认是 8192（MB）

（2）单个任务可申请的最多物理内存量，默认是 8192（MB）。

6）HDFS 和硬盘空闲控制在 70%以下。

8.1.2 Flume

1） Flume 内存配置为 4G（flume-env.sh 修改）

2） FileChannel 优化

通过配置 dataDirs 指向多个路径，每个路径对应不同的硬盘，增大 Flume 吞吐量。

checkpointDir 和 backupCheckpointDir 也尽量配置在不同硬盘对应的目录中，保证 checkpoint 坏掉后，可以快速使用 backupCheckpointDir 恢复数据

3） Sink：HDFS Sink 小文件处理

这三个参数配置写入 HDFS 后会产生小文件，hdfs.rollInterval、hdfs.rollSize、hdfs.rollCount

8.1.3 Kafka

1） Kafka 的吞吐量测试（测试生产速度和消费速度）

2） Kafka 内存为 6G（不能超过 6G）

3） Kafka 数量确定：2 * 峰值生产速度（m/s）* 副本数 / 100 + 1 = ?

4） Kafka 中的数据量计算

每天数据总量 100g(1 亿条) 10000 万/24/60/60 = 1150 条/s

平均每秒钟：1150 条

低谷每秒：400 条

高峰每秒钟：1150 * 10 = 11000 条

每条日志大小： 1K 左右

每秒多少数据量：20MB

5) Kafka 消息数据积压，Kafka 消费能力不足怎么处理？

a. 如果是 Kafka 消费能力不足，则可以考虑增加 Topic 的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）

b. 如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

8.1.4 Tez 引擎优点（略过）？

Tez 可以将多个有依赖的作业转换为一个作业，这样只需写一次 HDFS，且中间节点较少，从而大大提升作业的计算性能。

8.1.5 Sqoop 参数

1) Sqoop 导入导出 Null 存储一致性问题

2) Sqoop 数据导出一致性问题

- staging-table 方式 --clear-staging

3) Sqoop 数据导出的时候一次执行多长时间

Sqoop 任务 5 分钟-2 个小时的都有。取决于数据量。

8.1.6 Azkaban 每天执行多少个任务

1) 每天集群运行多少 job?

2) 每个任务的资源是如何分配的？

3) 多个指标（200）*6=1200（1000-2000 个 job）

4) 每天集群运行多少个 task? 1000*（5-8）=5000 多个

5) 任务挂了怎么办？运行成功或者失败都会发邮件

Zip a.job b.job c.job job.zip 把压缩的 zip 包放到 azkaban 的 web 界面上提交（指定 sechduler）

8.2 业务经验

8.2.1 ODS 层采用什么压缩方式和存储格式？

压缩采用 Snappy，存储采用 orc，压缩比是 100g 数据压缩完 10g 左右。

8.2.2 DWD 层做了哪些事？

1) 数据清洗

a. 空值去除

b. 过滤核心字段无意义的数据，比如订单表中订单 id 为 null，支付表中支付 id 为空

c. 对手机号、身份证号等敏感数据脱敏

d. 对业务数据传过来的表进行维度退化和降维。

e. 将用户行为宽表和业务表进行数据一致性处理

```
select case when a is null then b else a end as JZR,
    ...
from A
```

2) 清洗的手段

Sql、mr、rdd、kettle、Python（项目中采用 sql 进行清除）

3) 清洗掉多少数据算合理

1 万条数据清洗掉 1 条。

8.2.3 DWS 层做了哪些事？

1) DWS 层有 3-5 张宽表（处理 100-200 个指标　70%以上的需求）

具体宽表名称：用户行为宽表，用户购买商品明细行为宽表，商品宽表，购物车宽表，物流宽表、登录注册、售后等。

2) 哪个宽表最宽？大概有多少个字段？

最宽的是用户行为宽表。大概有 60-100 个字段

3) 具体用户行为宽表字段名称

评论、打赏、收藏、关注--商品、关注--人、点赞、分享、好价爆料、文章发布、活跃、签到、补签卡、幸运屋、礼品、金币、电商点击、gmv

```
CREATE TABLE `app_usr_interact`(
```

```
`stat_dt` date COMMENT '互动日期',
`user_id` string COMMENT '用户 id',
`nickname` string COMMENT '用户昵称',
`register_date` string COMMENT '注册日期',
`register_from` string COMMENT '注册来源',
`remark` string COMMENT '细分渠道',
`province` string COMMENT '注册省份',
`pl_cnt` bigint COMMENT '评论次数',
`ds_cnt` bigint COMMENT '打赏次数',
`sc_add` bigint COMMENT '添加收藏',
`sc_cancel` bigint COMMENT '取消收藏',
`gzg_add` bigint COMMENT '关注商品',
`gzg_cancel` bigint COMMENT '取消关注商品',
`gzp_add` bigint COMMENT '关注人',
`gzp_cancel` bigint COMMENT '取消关注人',
`buzhi_cnt` bigint COMMENT '点不值次数',
`zhi_cnt` bigint COMMENT '点值次数',
`zan_cnt` bigint COMMENT '点赞次数',
`share_cnts` bigint COMMENT '分享次数',
`bl_cnt` bigint COMMENT '爆料数',
`fb_cnt` bigint COMMENT '好价发布数',
`online_cnt` bigint COMMENT '活跃次数',
`checkin_cnt` bigint COMMENT '签到次数',
`fix_checkin` bigint COMMENT '补签次数',
`house_point` bigint COMMENT '幸运屋金币抽奖次数',
`house_gold` bigint COMMENT '幸运屋积分抽奖次数',
`pack_cnt` bigint COMMENT '礼品兑换次数',
`gold_add` bigint COMMENT '获取金币',
`gold_cancel` bigint COMMENT '支出金币',
`surplus_gold` bigint COMMENT '剩余金币',
`event` bigint COMMENT '电商点击次数',
`gmv_amount` bigint COMMENT 'gmv',
`gmv_sales` bigint COMMENT '订单数')
PARTITIONED BY ( `dt` string)
```

4) 商品详情 ----- 购物车 ----- 订单 ------ 付款的转换比率

5% 30% 70%

5) 每天的 GMV 是多少，哪个商品卖的最好？每天下单量多少？

a. 100 万的日活每天大概有 10 万人购买，平均每人消费 100 元，一天的 GMV 在 1000 万

b. 面膜，每天销售 5000 个

c. 每天下单量在 10 万左右

8.2.4 分析过哪些指标（一分钟至少说出 30 个指标）

1) 离线指标

网站流量指标 独立访问数 UV 页面访客数 PV

流量质量指标类 跳出率 平均页面访问时长 人均页面访问数

2) 购物车类指标

加入购物车次数 加入购物车买家次数 加入购物车商品数 购物车支付转化率

3) 下单类指标

下单笔数 下单金额 下单买家数 浏览下单转化率

4) 支付类指标

支付金额 支付买家数 支付商品数 浏览-支付买家转化率

下单-支付金额转化率 下单-支付买家数转换率

5) 交易类指标

交易成功订单数 交易成功金额 交易成功买家数 交易成功商品数

交易失败订单数 交易失败订单金额 交易失败买家数

交易失败商品数 退款总订单量 退款金额 退款率

6) 市场营销活动指标

新增访问人数 新增注册人数 广告投资回报率 UV 订单转化率

7) 风控类指标

买家评价数 买家上传图片数 买家评价率 买家好评率 买家差评率

物流平均配送时间

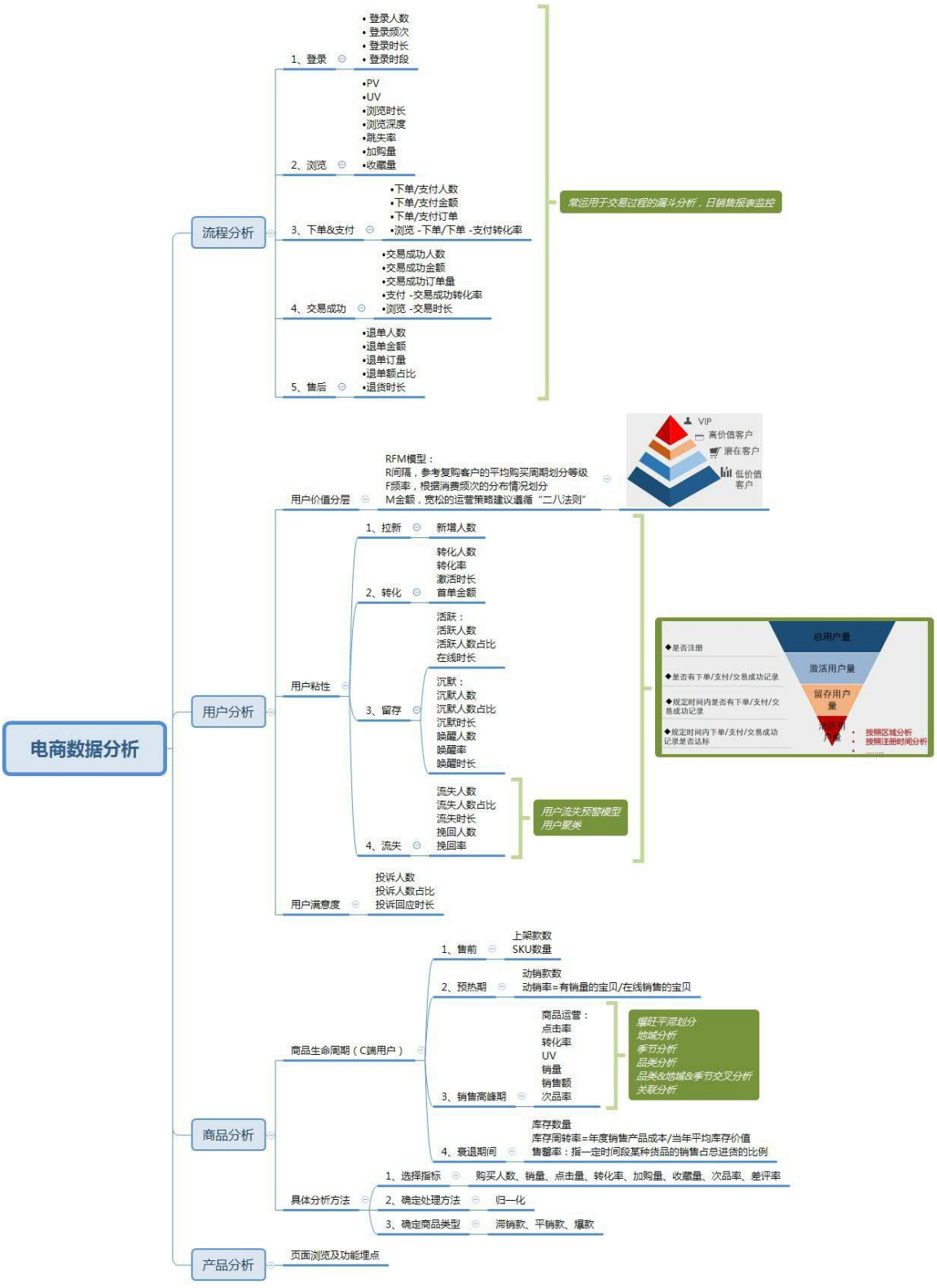
8) 投诉类指标

发起投诉数 投诉率 撤销投诉(申诉数)

9) 商品类指标

产品总数 SKU 数 SPU 数

上架商品 SKU 数 上架商品 SPU 数 上架商品数



日活跃用户，

月活跃用户，

各区域 Top10 商品统计，

季度商品品类点击率 top10，

用户留存，

月 APP 的用户增长人数，

广告区域点击数 top3，

活跃用户每天在线时长，

投诉人数占比，

沉默用户占比，

用户的新鲜度，

商品上架的 sku 数，

同种品类的交易额排名，

统计买家的评价率，

用户浏览时长，

统计下单的数量，

统计支付的数量，

统计退货的数量，

用户的（日活、月活、周活），

统计流失人数

日活，周活，月活，沉默用户占比，增长人数，活跃用户占比，在线时长统计，歌曲访问数，歌曲访问时长，各地区 Top10 歌曲统计 ,投诉人数占比，投诉回应时长，留存率，月留存率，转化率，GMV，复购 vip 率，vip 人数，歌榜，挽回率，粉丝榜，打赏次数，打赏金额，发布歌曲榜单，歌曲热度榜单，歌手榜单，用户年龄组，vip 年龄组占比，收藏数榜单，评论数

1) 用户活跃数统计（日活，月活，周活）

2) 某段时间的新增用户/活跃用户数

3) 页面单跳转化率统计

4) 活跃人数占比（占总用户比例）

5) 在线时长统计（活跃用户每天在线时长）

6) 统计本月的人均在线时长

7) 订单产生效率（下单的次数与访问次数比）

8） 页面访问时长（单个页面访问时长）

9） 统计本季度付款订单

10）统计某广告的区城点击数 top3

11）统计本月用户的流失人数

12）统计本月流失人数占用户人数的比例

13）统计本月 APP 的用户增长人数

14）统计本月的沉默用户

15）统计某时段的登录人数

16）统计本日用户登录的次数平均值

17）统计用户在某类型商品中的浏览深度（页面转跳率）

18）统计用户从下单开始到交易成功的平均时长

19）Top10 热门商品的统计

20）统计下单的数量

21）统计支付的数量

22）统计退货的数量

23）统计动销率（有销量的商品/在线销售的宝贝）

24）统计支付转化率

25）统计用户的消费频率

26）统计商品上架的 SKU 数

27）统计同种品类的交易额排名

28）统计按下单退款排序的 top10 的商品

29）统计本 APP 的投诉人数占用户人数的比例

30）用户收藏商品

8.2.5 分析过最难的两个指标，现场手写

最近连续 3 周活跃用户数：

最近3周连续活跃的用户：通常是周一对前3周的数据做统计，该数据一周计算一次。

1）创建表

```
drop table if exists ads_continuity_wk_count;
create external table ads_continuity_wk_count(
    `dt` string COMMENT '统计日期,一般用结束周周日日期,如果每天计算一次,可用当天日期',
    `wk_dt` string COMMENT '持续时间',
    `continuity_count` bigint
)
row format delimited fields terminated by '\t'
location '/warehouse/gmall/ads/ads_continuity_wk_count';
```

前一周	前二周	前三周
100	101	101
101	102	104
102		

2）导入数据

```
insert into table ads_continuity_wk_count
select
    '2019-02-20',
    concat(date_add(next_day('2019-02-20','MO'),-7*3),'_',date_add(next_day('2019-02-20','MO'),-1)),
    count(*)
from
(
    select mid_id
    from dws_uv_detail_wk
    where wk_dt>=concat(date_add(next_day('2019-02-20','MO'),-7*2),'_',date_add(next_day('2019-02-20','MO'),-7*2-1))
    and wk_dt<=concat(date_add(next_day('2019-02-20','MO'),-1),'_',date_add(next_day('2019-02-20','MO'),-1))
    group by mid_id
    having count(*)=3
)t1;
```

按照设备id对每周活跃表分组

统计次数等于3

100 1
101 3
102 2
104 1

最近 7 天连续 3 天活跃用户数：

1）创建表

```
drop table if exists ads_continuity_uv_count;
create external table ads_continuity_uv_count(
    `dt` string COMMENT '统计日期',
    `wk_dt` string COMMENT '最近7天日期',
    `continuity_count` bigint
) COMMENT '连续活跃设备数'
row format delimited fields terminated by '\t'
location '/warehouse/gmall/ads/ads_continuity_uv_count';
```

2）导入数据

```
insert into table ads_continuity_uv_count
select
    '2019-02-12',
    concat(date_add('2019-02-12',-6),'_','2019-02-12'),
    count(*)
from
(
    select mid_id
    from
    (
        select mid_id
        from
        (
            select
                mid_id,
                date_sub(dt,rank) date_dif
            from
            (
                select
                    mid_id,
                    dt,
                    rank() over(partition by mid_id order by dt) rank
                from dws_uv_detail_day
                where dt>=date_add('2019-02-12',-6) and dt<='2019-02-12'
            )t1
        )t2
        group by mid_id,date_dif
        having count(*)>=3
    )t3
    group by mid_id
)t4;
```

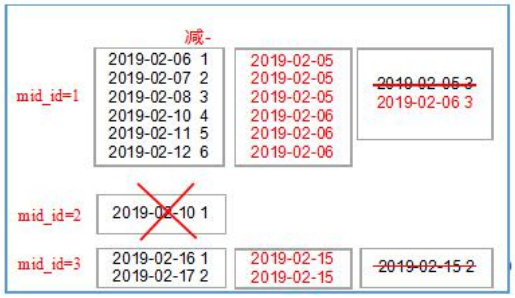
2、计算用户活跃日期及排名之间的差值

查询出最近7天的活跃用户，并对用户活跃日期进行排名

3、对同用户及差值分组，统计差值个数

4、将差值相同个数大于等于3的数据取出

5、对数据去重



8.2.6 数据仓库每天跑多少张表，大概什么时候运行，运行多久？

基本一个项目建一个库，表格个数为初始的原始数据表格加上统计结果表格的总数。（一般 70-100 张表格）

每天 0：30 开始运行。

所有离线数据报表控制在 8 小时之内

大数据实时处理部分控制在 5 分钟之内。

8.2.7 数仓中使用的哪种文件存储格式

常用的包括：textFile，rcFile，ORC，Parquet，一般企业里使用 ORC 或者 Parquet，因为是列式存储，且压缩比非常高，所以相比于 textFile，查询速度快，占用硬盘空间少

8.2.8 数仓中用到过哪些 Shell 脚本及具体功能

1) 集群启动停止脚本（Hadoop、Flume、Kafka、Zookeeper）

2) Sqoop 和数仓之间的导入导出脚本

3) 数仓层级之间的数据导入脚本。

8.2.9 项目中用过的报表工具

Echarts、kibana、supesrset

8.2.10 测试相关

1) 公司有多少台测试服务器？

测试服务器一般三台

2) 测试数据哪来的？

一部分自己写 Java 程序自己造，一部分从生产环境上取一部分。

3) 如何保证写的 sql 正确性

需要造一些特定的测试数据，测试。

离线数据和实时数据分析的结果比较。

4) 测试环境什么样？

测试环境的配置是生产的一半

5) 测试之后如何上线？

上线的时候，将脚本打包，提交 git。先发邮件抄送经理和总监，运维。通过之后跟运维一起上线。

8.2.11 项目实际工作流程

1) 先与产品讨论，看报表的各个数据从哪些埋点中取

2) 将业务逻辑过程设计好，与产品确定后开始开发

3) 开发出报表 SQL 脚本，并且跑几天的历史数据，观察结果

4) 将报表放入调度任务中，第二天给产品看结果。

5) 周期性将表结果导出或是导入后台数据库，生成可视化报表

8.2.12 项目中实现一个需求大概多长时间

刚入职第一个需求大概需要 7 天左右。

对业务熟悉后，平均一天一个需求。

影响时间的因素：开会讨论需求、表的权限申请、测试等

8.2.13 项目在 3 年内迭代次数，每一个项目具体是如何迭代的。

差不多一个月会迭代一次。就产品或我们提出优化需求，然后评估时间。每周我们都会开会做下周计划和本周总结。

有时候也会去预研一些新技术。

8.2.14 项目开发中每天做什么事

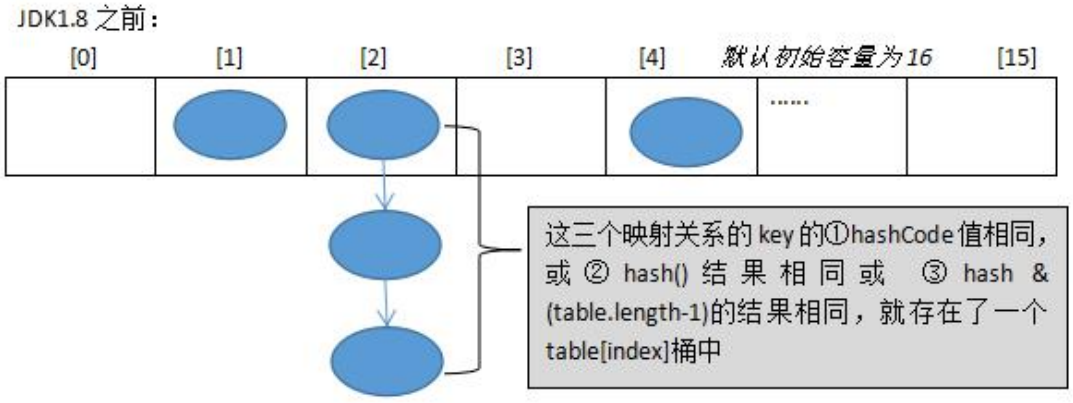
新需求比如埋点或是报表来了之后，需要设计做的方案，设计完成之后跟产品讨论，再开发。

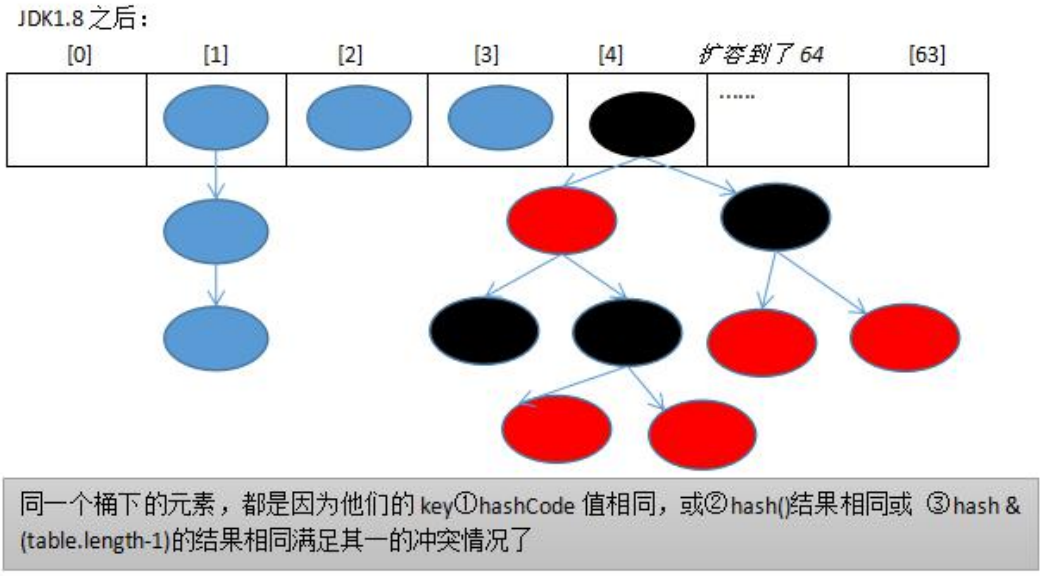
数仓的任何步骤出现问题，需要查看问题，比如日活，月活下降等。

第 9 章 JavaSE（答案精简）

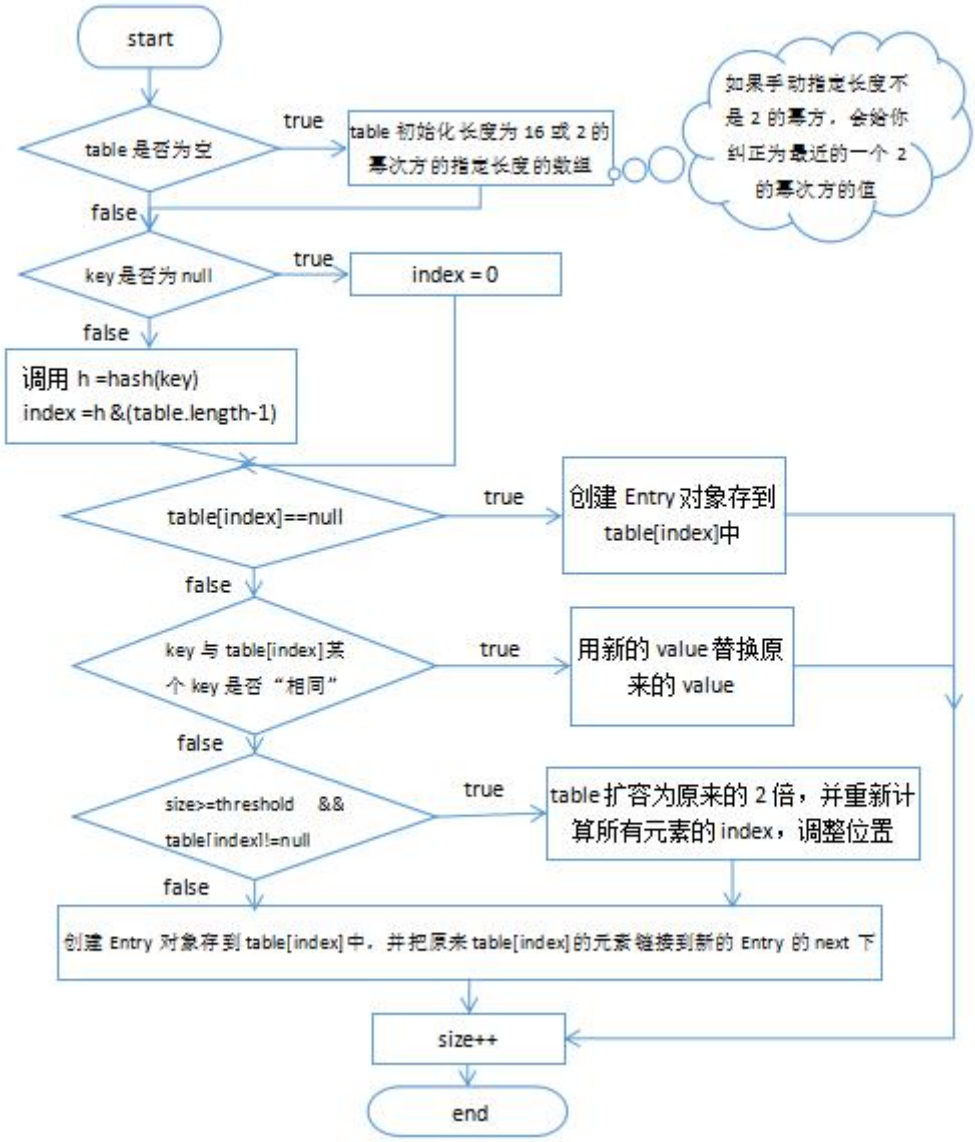
9.1 hashMap 底层源码，数据结构

hashMap 的底层结构在 jdk1.7 中由数组+链表实现，在 jdk1.8 中由数组+链表+红黑树实现，以数组+链表的结构为例。

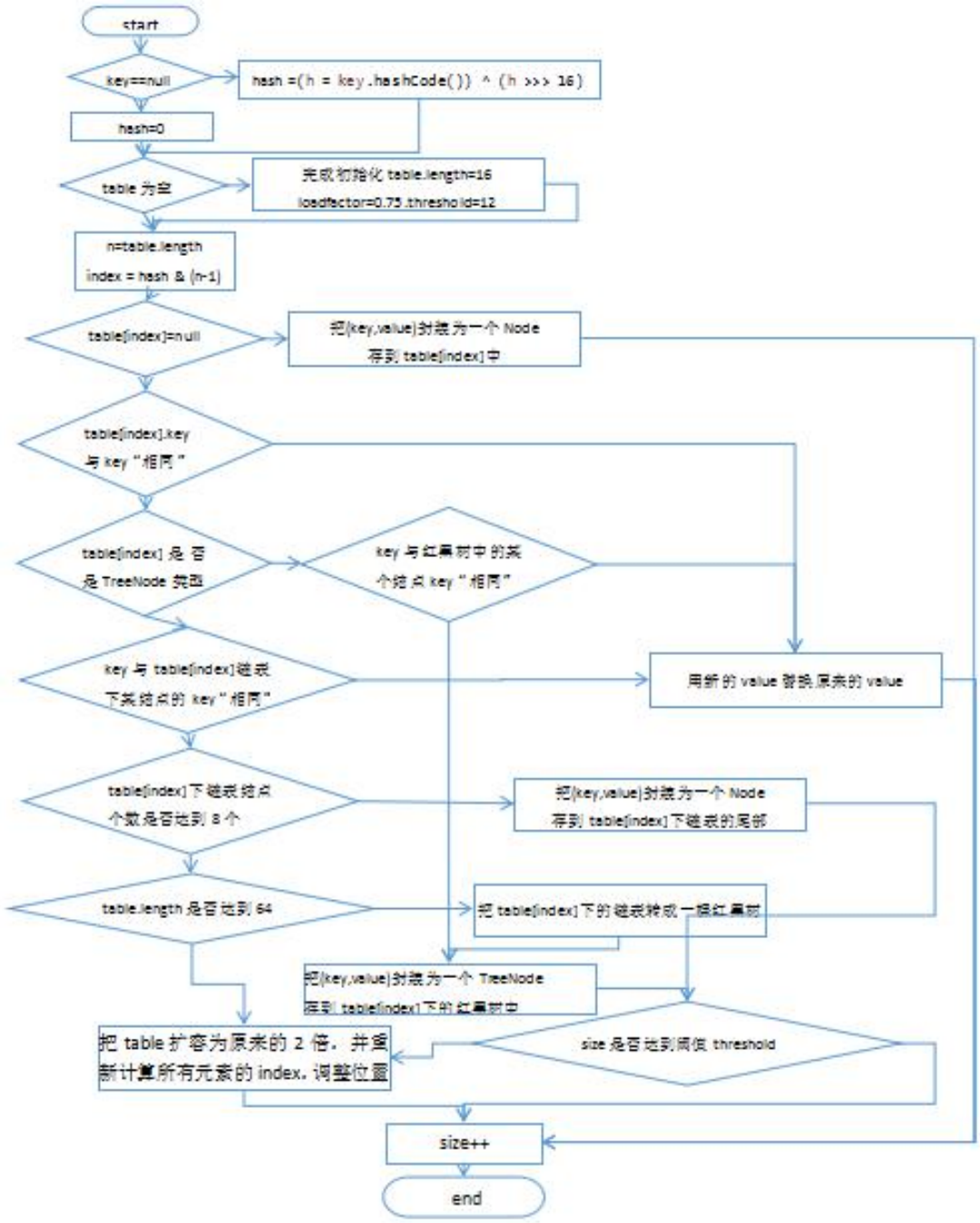




JDK1.8 之前 Put 方法：



JDK1.8 之后 Put 方法：



9.2 Java 自带有哪几种线程池？

1） newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。这种类型的线程池特点是：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为 Interger. MAX_VALUE)，这样

可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为 1 分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用 CachedThreadPool 时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

2) newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。FixedThreadPool 是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

3) newSingleThreadExecutor

创建一个单线程化的 Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

4) newScheduleThreadPool

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。延迟 3 秒执行。

9.3 HashMap 和 HashTable 区别

1) 线程安全性不同

HashMap 是线程不安全的，HashTable 是线程安全的，其中的方法是 Synchronize 的，在多线程并发的情况下，可以直接使用 HashTabl，但是使用 HashMap 时必须自己增加同步处理。

2) 是否提供 contains 方法

HashMap 只有 containsValue 和 containsKey 方法；HashTable 有 contains、containsKey 和 containsValue 三个方法，其中 contains 和 containsValue 方法功能相同。

3) key 和 value 是否允许 null 值

Hashtable 中，key 和 value 都不允许出现 null 值。HashMap 中，null 可以作为键，这样的

键只有一个；可以有一个或多个键所对应的值为 null。

4) 数组初始化和扩容机制

HashTable 在不指定容量的情况下的默认容量为 11，而 HashMap 为 16，Hashtable 不要求底层数组的容量一定要为 2 的整数次幂，而 HashMap 则要求一定为 2 的整数次幂。

Hashtable 扩容时，将容量变为原来的 2 倍加 1，而 HashMap 扩容时，将容量变为原来的 2 倍。

9.4 TreeSet 和 HashSet 区别

HashSet 是采用 hash 表来实现的。其中的元素没有按顺序排列，add()、remove()以及 contains()等方法都是复杂度为 O(1)的方法。

TreeSet 是采用树结构实现(红黑树算法)。元素是按顺序进行排列，但是 add()、remove()以及 contains()等方法都是复杂度为 O(log (n))的方法。它还提供了一些方法来处理排序的 set，如 first(), last(), headSet(), tailSet()等等。

9.5 String buffer 和 String build 区别

1) StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，

2) 只是 StringBuffer 中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是线程不安全的。

3) 在单线程程序下，StringBuilder 效率更快，因为它不需要加锁，不具备多线程安全而 StringBuffer 则每次都需要判断锁，效率相对更低

9.6 Final、Finally、Finalize

final：修饰符（关键字）有三种用法：修饰类、变量和方法。修饰类时，意味着它不能再派生出新的子类，即不能被继承，因此它和 abstract 是反义词。修饰变量时，该变量使用中不被改变，必须在声明时给定初值，在引用中只能读取不可修改，即为常量。修饰方法时，也同样只能使用，不能在子类中被重写。

finally：通常放在 try…catch 的后面构造最终执行代码块，这就意味着程序无论正常执行

还是发生异常，这里的代码只要 JVM 不关闭都能执行，可以将释放外部资源的代码写在 finally 块中。

finalize：Object 类中定义的方法，Java 中允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写 finalize() 方法可以整理系统资源或者执行其他清理工作。

9.7 ==和 Equals 区别

== ： 如果比较的是基本数据类型，那么比较的是变量的值

如果比较的是引用数据类型，那么比较的是地址值（两个对象是否指向同一块内存）

equals:如果没重写 equals 方法比较的是两个对象的地址值。

如果重写了 equals 方法后我们往往比较的是对象中的属性的内容

equals 方法是从 Object 类中继承的，默认的实现就是使用==

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

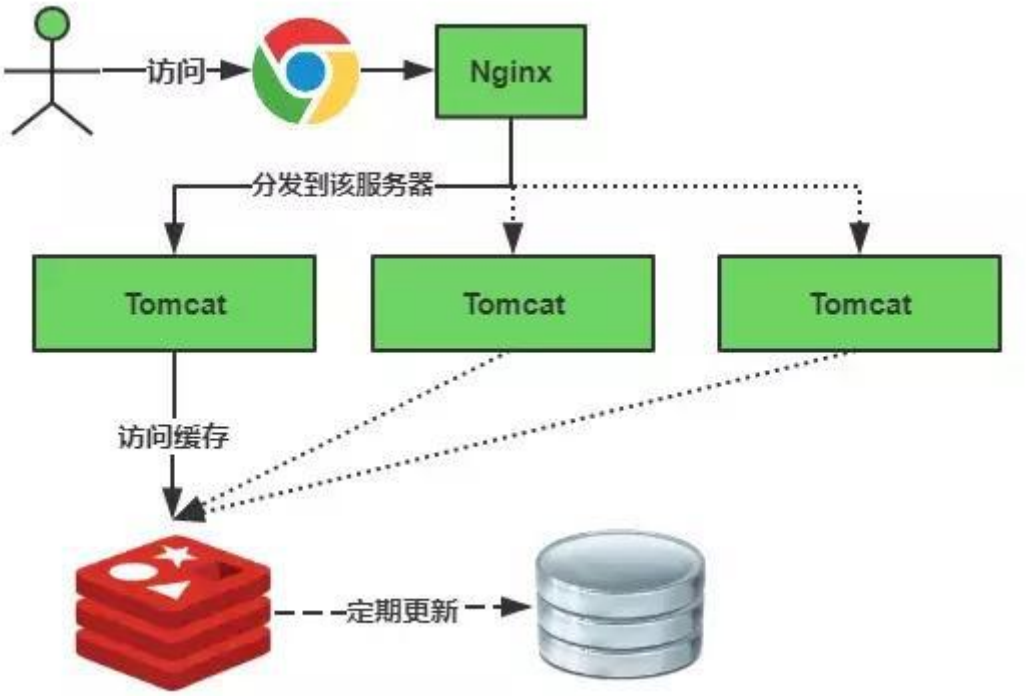
第 10 章 Redis（答案精简）

10.1 缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

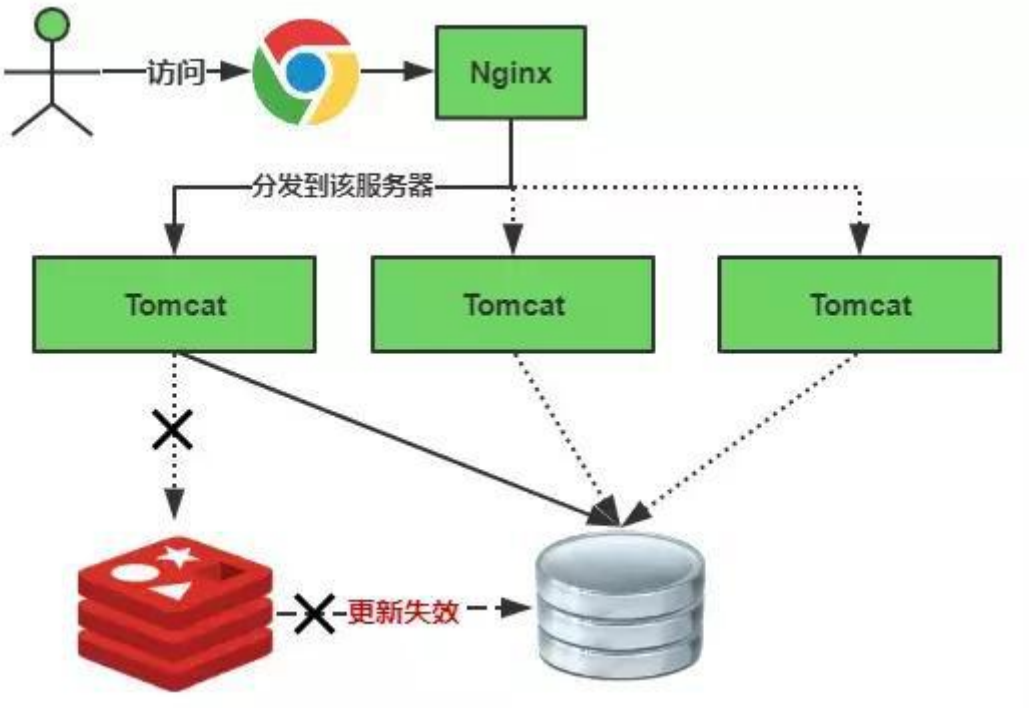
缓存雪崩

缓存雪崩我们可以简单的理解为：**由于原有缓存失效，新缓存未到期间**(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

缓存正常从 Redis 中获取，示意图如下：



缓存失效瞬间示意图如下：



缓存失效时的雪崩效应对底层系统的冲击非常可怕！大多数系统设计者考虑用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。还有一个简单方案就时讲缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降

低，就很难引发集体失效的事件。

以下简单介绍两种实现方式的伪代码：

（1）碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队，伪代码如下：

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
    String lockKey = cacheKey;

    String cacheValue = CacheHelper.get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        synchronized(lockKey) {
            cacheValue = CacheHelper.get(cacheKey);
            if (cacheValue != null) {
                return cacheValue;
            } else {
                //这里一般是sql查询数据
                cacheValue = GetProductListFromDB();
                CacheHelper.Add(cacheKey, cacheValue, cacheTime);
            }
        }
        return cacheValue;
    }
}
```

加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间 key 是锁着的，这是过来 1000 个请求 999 个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

注意：加锁排队的解决方式分布式环境的并发问题，有可能还要解决分布式锁的问题；线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

（2）还有一个解决办法解决方案是：给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存，实例伪代码如下：

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
    //缓存标记
    String cacheSign = cacheKey + "_sign";

    String sign = CacheHelper.Get(cacheSign);
    //获取缓存值
    String cacheValue = CacheHelper.Get(cacheKey);
    if (sign != null) {
        return cacheValue; //未过期，直接返回
    } else {
        CacheHelper.Add(cacheSign, "1", cacheTime);
        ThreadPool.QueueUserWorkItem((arg) -> {
            //这里一般是 sql查询数据
            cacheValue = GetProductListFromDB();
            //日期设缓存时间的2倍，用于脏读
            CacheHelper.Add(cacheKey, cacheValue, cacheTime * 2);
        });
        return cacheValue;
    }
}
```

解释说明：

1、缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际 key 的缓存；

2、缓存数据：它的过期时间比缓存标记的时间延长 1 倍，例：标记缓存时间 30 分钟，数据缓存设置为 60 分钟。 这样，当缓存标记 key 过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

关于缓存崩溃的解决方法，这里提出了三种方案：使用锁或队列、设置过期标志更新缓存、为 key 设置不同的缓存失效时间，还有一各被称为“二级缓存”的解决方法，有兴趣的读者可以自行研究。

缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

另外也有一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";

    String cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    }

    cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        //数据库查询不到，为空
        cacheValue = GetProductListFromDB();
        if (cacheValue == null) {
            //如果发现为空，设置个默认值，也缓存起来
            cacheValue = string.Empty;
        }
        CacheHelper.Add(cacheKey, cacheValue, cacheTime);
        return cacheValue;
    }
}
```

把空结果，也给缓存起来，这样下次同样的请求就可以直接返回空了，即可以避免当查询的值为空时引起的缓存穿透。同时也可以单独设置个缓存区域存储空值，对要查询的 key 进行预先校验，然后再放行给后面的正常缓存处理逻辑。

缓存预热

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决思路：

1） 直接写个缓存刷新页面，上线时手工操作下；

2） 数据量不大，可以在项目启动的时候自动进行加载；

3） 定时刷新缓存；

缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis 默认的有 6 种策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

1） 定时去清理过期的缓存；

2） 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的 key 是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

1） 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

2） 警告：有些服务在一段时间内成功率有波动（如在 95~100%之间），可以自动降级或人工降级，并发送告警；

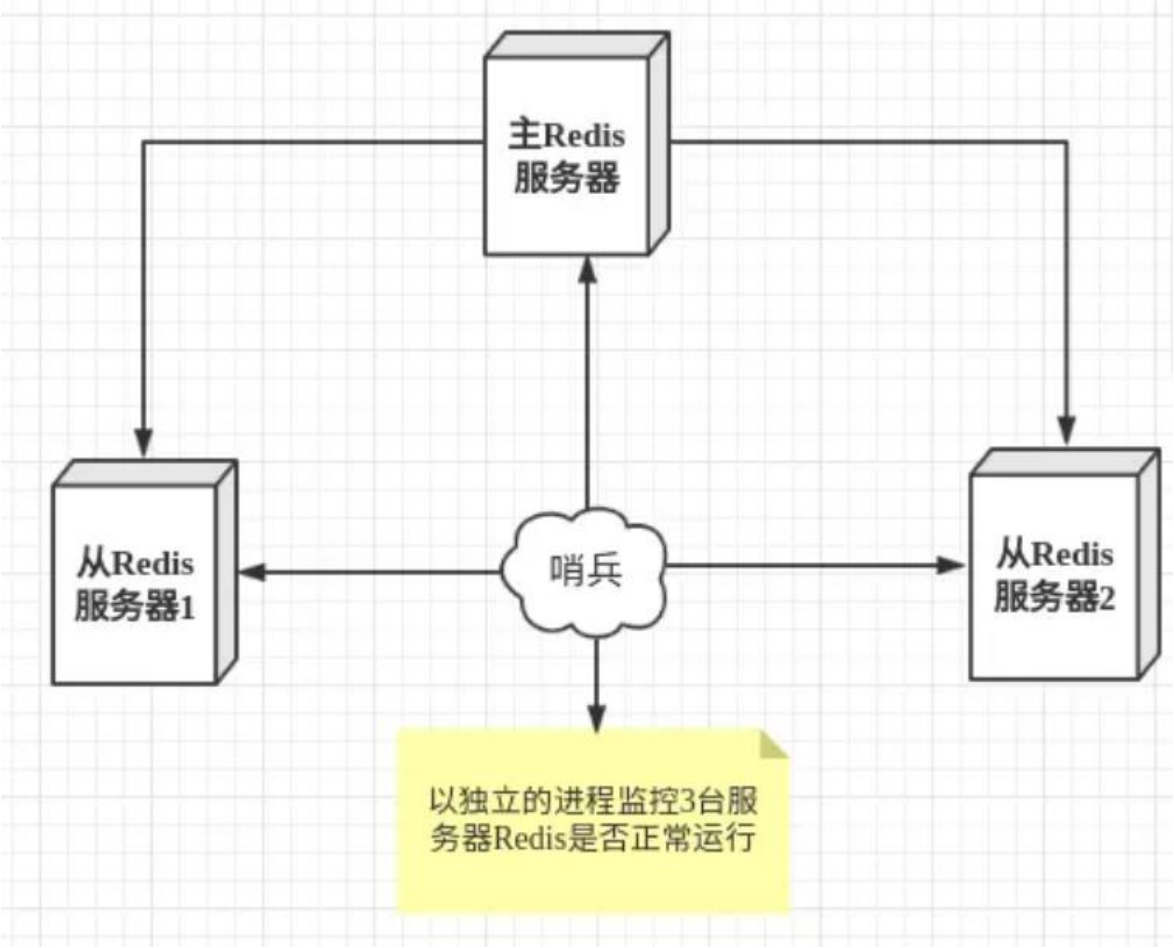
3） 错误：比如可用率低于 90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

4） 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

10.2 哨兵模式

主从切换技术的方法是：当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这不是一种推荐的方式，更多时候，我们优先考虑哨兵模式。

哨兵模式是一种特殊的模式，首先 Redis 提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是哨兵通过发送命令，等待 Redis 服务器响应，从而监控运行的多个 Redis 实例。



这里的哨兵有两个作用

1） 通过发送命令，让 Redis 服务器返回监控其运行状态，包括主服务器和从服务器。

2） 当哨兵监测到 master 宕机，会自动将 slave 切换成 master，然后通过发布订阅模式通知其

他的从服务器，修改配置文件，让它们切换主机。

然而一个哨兵进程对 Redis 服务器进行监控，可能会出现问题，为此，我们可以使用多个哨兵进行监控。各个哨兵之间还会进行监控，这样就形成了多哨兵模式。

用文字描述一下故障切换（failover）的过程。假设主服务器宕机，哨兵 1 先检测到这个结果，系统并不会马上进行 failover 过程，仅仅是哨兵 1 主观的认为主服务器不可用，这个现象成为主观下线。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行 failover 操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为客观下线。这样对于客户端而言，一切都是透明的。

10.3 数据类型

string	字符串
list	可以重复的集合
set	不可以重复的集合
hash	类似于 Map<String,String>
zset(sorted set)	带分数的 set

10.4 持久化

1) RDB 持久化：

   a. 在指定的时间间隔内持久化

   b. 服务 shutdown 会自动持久化

   c. 输入 bgsave 也会持久化

2) AOF ： 以日志形式记录每个更新操作

   Redis 重新启动时读取这个文件，重新执行新建、修改数据的命令恢复数据。

   保存策略：

   推荐（并且也是默认）的措施为每秒持久化一次，这种策略可以兼顾速度和安全性。

   缺点：

   a. 比起 RDB 占用更多的磁盘空间

b. 恢复备份速度要慢

c. 每次读写都同步的话，有一定的性能压力

d. 存在个别 Bug，造成恢复不能

选择策略：

官方推荐：

如果对数据不敏感，可以选单独用 RDB；不建议单独用 AOF，因为可能出现 Bug;如果只是做纯内存缓存，可以都不用

10.5 悲观锁

所谓悲观锁：具有强烈的独占和排他特性。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

简单来说：执行操作前假设当前的操作肯定（或有很大几率）会被打断（悲观）。基于这个假设，我们在做操作前就会把相关资源锁定，不允许自己执行期间有其他操作干扰。

悲观锁的并发性能差，但是能保证不会发生脏数据的可能性小一点。

10.6 乐观锁

执行操作前假设当前操作不会被打断（乐观）。基于这个假设，我们在做操作前不会锁定资源，万一发生了其他操作的干扰，那么本次操作将被放弃。Redis 使用的就是乐观锁。

10.7 redis 是单线程的，为什么那么快

1) 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。

2) 数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的

3) 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

4） 使用多路 I/O 复用模型，非阻塞 IO

5） 使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

第 11 章 MySql

11.1 MyISAM 与 InnoDB 的区别

对比项	MyISAM	InnoDB
外键	不支持	支持
事务	不支持	支持
行表锁	表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作	行锁,操作时只锁某一行，不对其它行有影响， 适合高并发的操作
缓存	只缓存索引，不缓存真实数据	不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响

11.2 索引

数据结构：B+Tree

一般来说能够达到 range 就可以算是优化了

口诀（两个法则加 6 种索引失效的情况）

全值匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

LIKE 百分写最右，覆盖索引不写*；

不等空值还有 OR，索引影响要注意；

VAR 引号不可丢，SQL 优化有诀窍。

11.3 b-tree 和 b+tree 的区别

1） B-树的关键字和记录是放在一起的，叶子节点可以看作外部节点，不包含任何信息；B+树的非叶子节点中只有关键字和指向下一个节点的索引，记录只放在叶子节点中。

2） 在 B-树中，越靠近根节点的记录查找时间越快，只要找到关键字即可确定记录的存在；而 B+树中每个记录的查找时间基本是一样的，都需要从根节点走到叶子节点，而且在叶子节点中还要再比较关键字。

11.4 MySQL 的事务

一、事务的基本要素（ACID）

1） 原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位

2） 一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏 。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

3） 隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱，在 A 取钱的过程结束前，B 不能向这张卡转账。

4） 持久性（Durability）：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

二、事务的并发问题

1） 脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是脏数据

2） 不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数据作了更新并提交，导致事务 A 多次读取同一数据时，结果 不一致

3） 幻读：系统管理员 A 将数据库中所有学生的成绩从具体分数改为 ABCDE 等级，但是系统管理员 B 就在这个时候插入了一条具体分数的记录，当系统管理员 A 改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

三、MySQL 事务隔离级别

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是
串行化（serialize）	否	否	否

11.5 常见面试 sql

1) 用一条 SQL 语句查询出每门课都大于 80 分的学生姓名

name	kecheng	fenshu
张三	语文	81
张三	数学	75
李四	语文	76
李四	数学	90
王五	语文	81
王五	数学	100
王五	英语	90

A: select distinct name from table where name not in (select distinct name from table where fenshu<=80)

B：select name from table group by name having min(fenshu)>80

2) 学生表

自动编号	学号	姓名	课程编号	课程名称	分数
1	2005001	张三	0001	数学	69
2	2005002	李四	0001	数学	89

3　　2005001　张三　　0001　　数学　　69

删除除了自动编号不同，　其他都相同的学生冗余信息

A: delete tablename where　自动编号　not in(select min(自动编号) from tablename group by 学号，　姓名，　课程编号，　课程名称，　分数)

3)　一个叫 team 的表，里面只有一个字段 name,一共有 4 条纪录，分别是 a,b,c,d,对应四个球队，现在四个球队进行比赛，用一条 sql 语句显示所有可能的比赛组合

```
select a.name, b.name
from team a, team b
where a.name < b.name
```

4)　面试题：怎么把这样一个

year　　month amount

1991　　1　　1.1

1991　　2　　1.2

1991　　3　　1.3

1991　　4　　1.4

1992　　1　　2.1

1992　　2　　2.2

1992　　3　　2.3

1992　　4　　2.4

查成这样一个结果

year m1　m2　m3　m4

1991 1.1 1.2 1.3 1.4

1992 2.1 2.2 2.3 2.4

答案

```
select year,
```

```
(select amount from  aaa m where month=1  and m.year=aaa.year) as m1,
(select amount from  aaa m where month=2  and m.year=aaa.year) as m2,
(select amount from  aaa m where month=3  and m.year=aaa.year) as m3,
(select amount from  aaa m where month=4  and m.year=aaa.year) as m4
from aaa  group by year
```

5) 说明：复制表(只复制结构,源表名：a 新表名：b)

SQL: select * into b from a where 1<>1 (where1=1，拷贝表结构和数据内容)

```
ORACLE:create table b
As
Select * from a where 1=2
```

[<>（不等于）(SQL Server Compact)

比较两个表达式。 当使用此运算符比较非空表达式时，如果左操作数不等于右操作数，则结果为 TRUE。 否则，结果为 FALSE。]

6) 原表:

```
courseid coursename score
----------------------------------
1 java 70
2 oracle 90
3 xml 40
4 jsp 30
5 servlet 80
----------------------------------
```

为了便于阅读,查询此表后的结果显式如下(及格分数为 60):

```
courseid coursename score mark
------------------------------------------------
1 java 70 pass
2 oracle 90 pass
```

3 xml 40 fail

4 jsp 30 fail

5 servlet 80 pass

写出此查询语句

select courseid, coursename ,score ,if(score>=60, "pass","fail") as mark from course

7) 表名：购物信息

购物人	商品名称	数量
A	甲	2
B	乙	4
C	丙	1
A	丁	2
B	丙	5

……

给出所有购入商品为两种或两种以上的购物人记录

答：select * from 购物信息 where 购物人 in (select 购物人 from 购物信息 group by 购物人 having count(*) >= 2);

8) info 表

date	result
2005-05-09	win
2005-05-09	lose
2005-05-09	lose
2005-05-09	lose
2005-05-10	win
2005-05-10	lose
2005-05-10	lose

如果要生成下列结果，该如何写 sql 语句?

win lose

2005-05-09 2 2

2005-05-10 1 2

答案：

(1) select date, sum(case when result = "win" then 1 else 0 end) as "win", sum(case when result = "lose" then 1 else 0 end) as "lose" from info group by date;

(2) select a.date, a.result as win, b.result as lose

from

(select date, count(result) as result from info where result = "win" group by date) as a

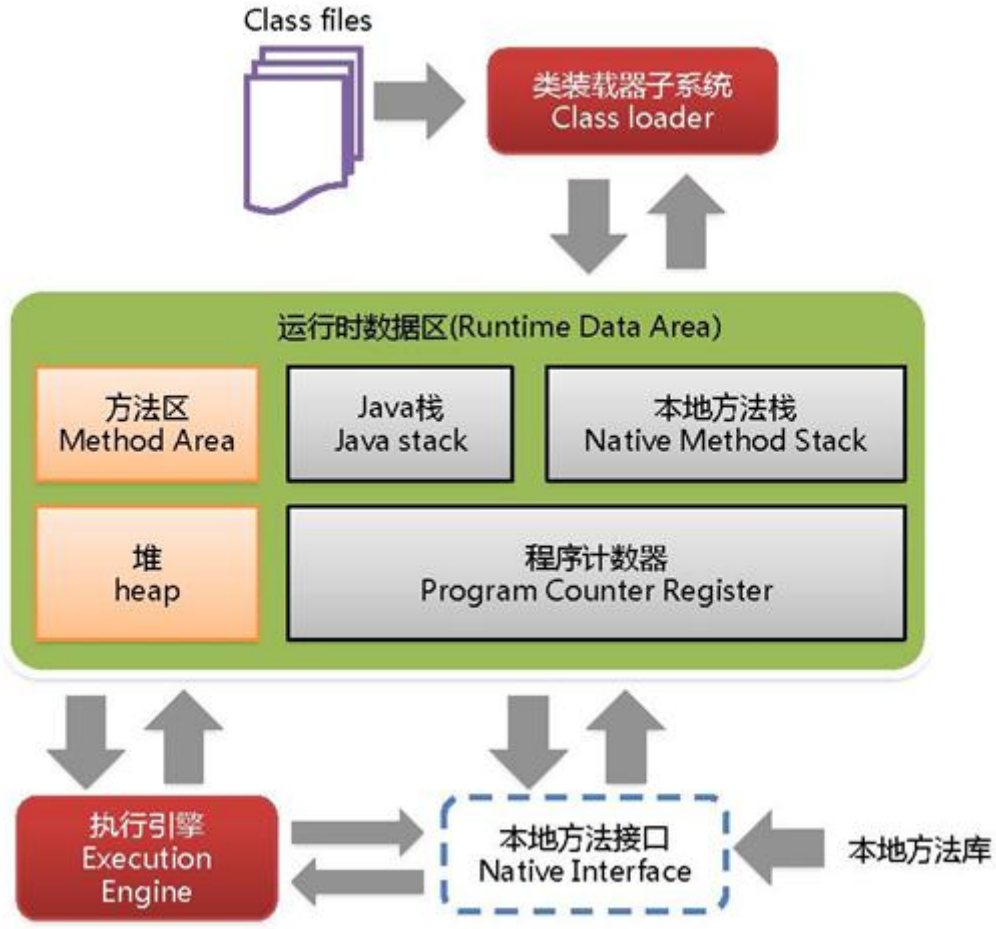
join

(select date, count(result) as result from info where result = "lose" group by date) as b

on a.date = b.date;

第 12 章　JVM

12.1 JVM 内存分哪几个区，每个区的作用是什么?



java 虚拟机主要分为以下几个区:

1)　方法区：

a.　有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生 GC，在这里进行的 GC 主要是对方法区里的常量池和对类型的卸载

b.　方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。

c.　该区域是被线程共享的。

d.　方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

2)　虚拟机栈:

a. 虚拟机栈也就是我们平常所称的栈内存,它为 java 方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。

b. 虚拟机栈是线程私有的，它的生命周期与线程相同。

c. 局部变量表里存储的是基本数据类型、returnAddress 类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定

d. 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式

e. 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

3) 本地方法栈：

本地方法栈和虚拟机栈类似，只不过本地方法栈为 Native 方法服务。

4) 堆：

java 堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

5) 程序计数器：

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个 java 虚拟机规范没有规定任何 OOM 情况的区域。

12.2 Java 类加载过程?

Java 类加载需要经历一下几个过程：

1) 加载

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

a. 通过一个类的全限定名获取该类的二进制流。

b. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。

c. 在内存中生成该类的 Class 对象，作为该类的数据访问入口。

2） 验证

验证的目的是为了确保 Class 文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

a. 文件格式验证：验证字节流是否符合 Class 文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.

b. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。

c. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。

d. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

e. 准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

3） 解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

4） 初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。

12.3 java 中垃圾收集的方法有哪些?

1） **引用计数法** 应用于：微软的 COM/ActionScrip3/Python 等

a. 如果对象没有被引用，就会被回收，缺点：需要维护一个引用计算器

2) **复制算法** 年轻代中使用的是 Minor GC，这种 GC 算法采用的是复制算法(Copying)

a. 效率高，缺点：需要内存容量大，比较耗内存

b. 使用在占空间比较小、刷新次数多的新生区

3) **标记清除** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a. 效率比较低，会差生碎片。

4) **标记压缩** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a. 效率低速度慢，需要移动对象，但不会产生碎片。

5) **标记清除压缩** 标记清除-标记压缩的集合，多次 GC 后才 Compact

a. 使用于占空间大刷新次数少的养老区，是 3 4 的集合体

12.4 如何判断一个对象是否存活?(或者 GC 对象的判定方法)

判断一个对象是否存活有两种方法:

1) 引用计数法

2) 可达性算法(引用链法)

12.5 什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1) 启动类加载器(Bootstrap ClassLoader)用来加载 java 核心类库，无法被 java 程序直接引用。

2) 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3) 系统类加载器（system class loader）也叫应用类加载器：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

4) 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

12.6 简述 Java 内存分配与回收策略以及 Minor GC 和 Major GC（full GC）

内存分配：

1） 栈区：栈分为 java 虚拟机栈和本地方法栈

2） 堆区：堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。堆区是 gc 的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为 Eden 区，主要放新创建对象，From survivor 和 To survivor 保存 gc 后幸存下的对象，默认情况下各自占比 8:1:1。

3） 方法区：被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被 Java 虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代（permanment generation）

4） 程序计数器：当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。线程私有的。

回收策略以及 Minor GC 和 Major GC：

1） 对象优先在堆的 Eden 区分配。

2） 大对象直接进入老年代。

3） 长期存活的对象将直接进入老年代。

当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC.Minor GC 通常发生在新生代的 Eden 区，在这个区的对象生存期短，往往发生 GC 的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC,但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。

第 13 章 JUC

13.1 Synchronized 与 Lock 的区别

1） Synchronized 能实现的功能 Lock 都可以实现，而且 Lock 比 Synchronized 更好用，更灵活。

2） Synchronized 可以自动上锁和解锁；Lock 需要手动上锁和解锁

13.2 Runnable 和 Callable 的区别

1） Runnable 接口中的方法没有返回值；Callable 接口中的方法有返回值

2） Runnable 接口中的方法没有抛出异常；Callable 接口中的方法抛出了异常

3） Runnable 接口中的落地方法是 call 方法；Callable 接口中的落地方法是 run 方法

13.3 什么是分布式锁

当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。分布式锁可以将标记存在内存，只是该内存不是某个进程分配的内存而是公共内存，如 Redis，通过 set (key,value,nx,px,timeout)方法添加分布式锁。

13.4 什么是分布式事务

分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。
