分布式：

• 什么是 CAP 理论？

• CAP 中的 P 是什么意思？

• 为什么说分布式系统，只能在 C、A 中二选一？

• 结合实际应用，CP、AP 该怎么选择？

**1. HDFS分块处理的好处**

**本质原因**：

* **分布式计算优化**：分块（128MB/256MB）使数据可并行处理，每个Block可被独立调度到不同节点计算，最大化利用集群资源
* **容错与经济性**：块损坏只需重新复制该块（而非整个文件），副本机制（默认3副本）保证数据安全
* **适应硬件限制**：突破单机磁盘容量限制，支持PB级存储
* **计算本地化（Data Locality）**：计算任务可直接调度到数据所在节点，减少网络传输（**面试核心点**：强调网络IO是分布式系统最大瓶颈）

**数仓视角**：分块设计是数仓大规模ETL的基础，使Spark/Hive能并行处理海量数据

**3. 集合接口设计**

**图表**

代码

**关键点**：

* Map不是Collection子接口，因语义不同（键值对 vs 单元素集合）
* Java集合框架设计体现**关注点分离**原则
* **工程意义**：防止API污染（如Map不需要add()方法）

**4. 类名冲突优先级**

**加载顺序**：

1. 当前包 > 导入包（IDE智能提示可调整）
2. JVM类加载遵循 **"双亲委派"** 机制：
   * Bootstrap ClassLoader (JDK核心类)
   * Extension ClassLoader
   * Application ClassLoader (用户类)

* **危险场景**：自定义**java.lang.String**不会被加载（安全机制阻止）

**数据开发实践**：避免与JDK类同名，使用**com.company.project**包前缀

**6. Lambda清晰性的本质**

java

// 传统写法

list.stream()

.filter(new Predicate<String>() {

public boolean test(String s) {

return s.length() > 3;

}

})

// Lambda写法

list.stream().filter(s -> s.length() > 3)

**技术原理**：

* **类型擦除**：编译器自动推断**Predicate<String>**类型
* **IDE增强**：IntelliJ的**流调试器**可视化数据流
* **函数式编程**：每个Lambda对应一个操作步骤（filter/map/reduce）

**分布式意义**：Lambda是Spark RDD算子链式调用的基础

**9. 泛型无法识别的原因**

**场景复现**：

java

List<Pair<String, Long>> list = Arrays.asList(

new Pair<>("A", 1L), // 编译器无法推断Pair的泛型

new Pair<>("B", 2L)

);

**根本原因**：Java泛型在编译时擦除类型信息，运行时无法获取

**解决方案**：

* 显式声明：**new Pair<String, Long>("A", 1L)**
* 使用工具类：**Lists.newArrayList(new Pair<>("A", 1L))** (Guava)

**10. Flume Interceptor作用**

**图表**

代码

**核心功能**：

* **数据清洗**：过滤无效事件（如Nginx日志中的健康检查请求）
* **字段增强**：添加主机名、时间戳等元数据
* **路由控制**：根据内容分发到不同Channel（如按日志级别分流）
* **数据脱敏**：移除敏感字段（手机号、身份证）

**数仓价值**：在数据入湖前完成预处理，降低下游计算成本

**11. 行动算子 vs 转换算子**

| **特性** | **转换算子（sortBy）** | **行动算子（collect）** |
| --- | --- | --- |
| **执行时机** | 延迟执行（Lazy） | 立即触发Job |
| **返回值** | 新RDD | 非RDD（Scala集合） |
| **数据移动** | 无跨节点移动 | 数据拉取到Driver |
| **容错** | 依赖Lineage重建 | 无重建能力 |

**关键理解**：**collect().foreach()**是**反模式**！原因：

1. 数据从Executor拉回Driver（网络瓶颈）
2. 单机处理丧失分布式优势
3. Driver可能OOM

**正确做法**：直接用**rdd.foreach()**分布式执行

**13. startWith方法本质**

* **函数签名**：**String.startsWith(prefix: String): Boolean**
* **分布式应用**：常用于日志分类

scala

val errorLogs = logs.filter(\_.startsWith("ERROR"))

* **优化技巧**：在分布式系统中避免使用正则匹配（性能差）

**14. reduceByKey分区排序谜题**

**根本原因**：HashPartitioner的分区算法

scala

partition = key.hashCode() % numPartitions

* **分区数<Key数**：多个Key被分到同一分区，分区内数据顺序不确定
* **分区数≥Key数**：每个Key独立分区，按分区编号顺序输出

**工程启示**：不要依赖reduceByKey的输出顺序！需显式调用**sortByKey**

**15. map产生Null元组异常**

**场景**：

scala

rdd.map(x => (x, null)).collect() // 抛出NPE

**原因**：

1. Spark内部使用**Product2**（Tuple2的父特质）存储KV
2. Tuple2的**\_1**和**\_2**不允许为null（Scala设计约束）
3. 序列化时调用**\_2.toString**导致NPE

**解决方案**：

scala

rdd.map(x => (x, Option.empty[String])) // 使用Option封装

**16. SparkSession vs SparkContext**

| **维度** | **JavaSparkContext** | **SparkSession** |
| --- | --- | --- |
| **API时代** | Spark 1.x (RDD API) | Spark 2.x+ (统一入口) |
| **功能范围** | 仅RDD操作 | RDD + DataFrame + SQL |
| **创建方式** | **new JavaSparkContext()** | **SparkSession.builder()** |
| **元数据** | 无元数据管理 | 内置Catalog管理表信息 |

**演进意义**：SparkSession是**结构化数据**处理的里程碑，支撑Delta Lake等现代数仓

**17. 小文件性能问题本质**

**三层瓶颈**：

1. **存储层**：HDFS NameNode内存消耗（每个文件≈150B元数据）
2. **计算层**：
   * 任务启动开销 > 数据处理时间（MapReduce/Spark Task调度耗时）
   * 大量随机IO（机械硬盘寻道时间成为瓶颈）
3. **查询层**：列存格式（Parquet/ORC）的**Footer读取放大**

**字节优化方案**：

* **写入时合并**：**spark.sql.shuffle.partitions**控制输出文件数
* **定期Compaction**：Delta Lake的**OPTIMIZE**命令
* **查询加速**：使用FileIndex跳过无关文件（如Iceberg的Manifest）

**18. 分布式打印 vs Driver打印**

| **场景** | **rdd.foreach(println)** | **rdd.collect().foreach(println)** |
| --- | --- | --- |
| **执行位置** | Executor节点（分布式） | Driver单机 |
| **输出位置** | 各Executor的stdout日志 | Driver控制台 |
| **网络开销** | 无 | 全量数据网络传输 |
| **内存风险** | 无OOM风险 | Driver可能OOM |
| **调试用途** | 查看分布式数据处理过程 | 小数据集快速验证 |
| **生产环境行为** | **禁用**（日志淹没集群） | 绝对禁止（稳定性风险） |

**字节最佳实践**：

* 调试用：**rdd.take(10).foreach(println)**
* 生产环境：日志写入分布式存储（HDFS/S3），通过日志系统查看