大数据训练营 — 模块九
Hadoop/Spark 核心源码学习

极客时间

金澜涛

目录

大数据基础算法：

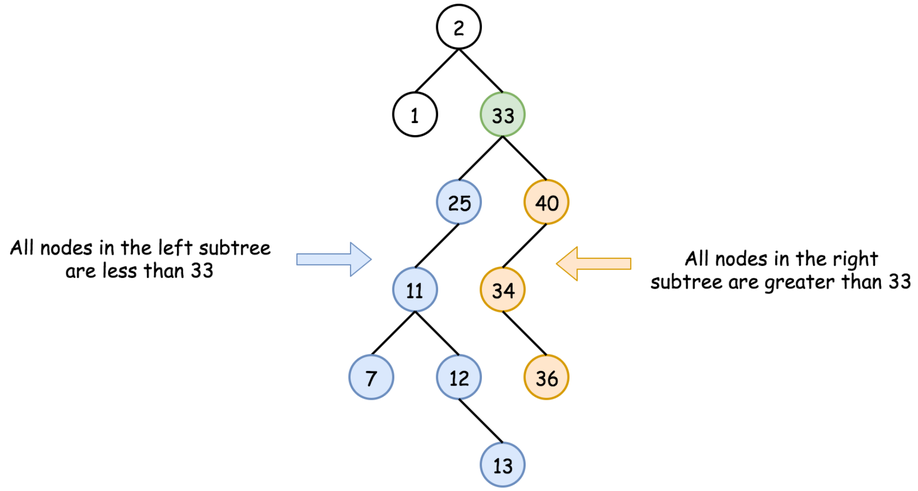
- B 树
- LSM 树
- 布隆过滤器
- 跳表
- 并归排序
- Scala 基础
- 函数式编程

目录

1. 大数据基础算法

二叉搜索树B树

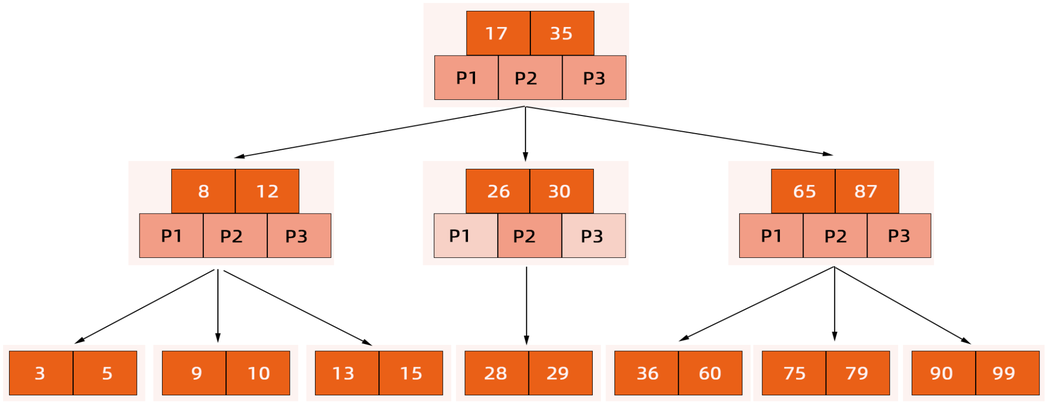
- 二叉搜索树是一棵二叉树，每个节点都有以下特性：
  - 大于左子树上任意一个节点的值
  - 小于右子树上任意一个节点的值
  - 递归



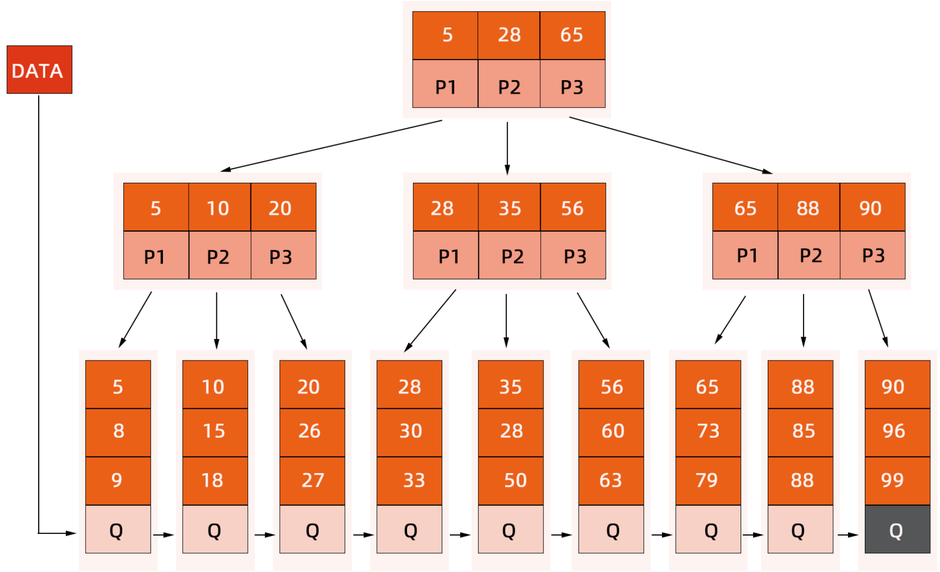

```
class Solution {
  public TreeNode searchBST(TreeNode root, int val) {
    if (root == null || val == root.val) return root;

    return val < root.val ? searchBST(root.left, val):  searchBST(root.right, val);
  }
}
```

B-树和B+树

- 为于减少树的高度，B-树是一种多路搜索树，并不是二叉的。



- B+树是B-树的变体，也是一种多路搜索树。



B+Tree 索引

- B+Tree 是 MySQL 使用最频繁的一个索引数据结构，是 InnoDB 和 MyISAM 存储引擎模式的索引类型。相对 Hash 索引，B+Tree 在查找单条记录的速度比不上 Hash 索引，但是因为更适合排序等操作，所以它更受欢迎，毕竟不可能只对数据库进行单条记录的操作。
- B+Tree 所有索引数据都在叶子节点上，并且增加了顺序访问指针，每个叶子节点都有指向相邻叶子节点的指针。这样做是为了提高区间效率，例如查询 key 为从18到49的所有数据记录，当找到18后，只要顺着节点和指针顺序遍历就可以以此向访问到所有数据节点，极大提高了区间查询效率。
- 数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点需要一次 I/O 就可以完全载入，大大减少磁盘 I/O 读写。

LSM 树

- LSM 树并不像 B+ 树、红黑树一样是一颗严格的树状数据结构，它其实是一种存储结构，目前HBase、LevelDB、RocksDB 这些 NoSQL 存储都是采用的 LSM 树
- LSM 树的核心特点是利用顺序写来提高写性能
- 这种设计对读取操作是非常不利的，因为需要在读取的过程中，通过归并所有文件来读取所对应的 KV，这是非常消耗 IO 资源的
- HBase 牺牲了读的性能，来实现顺序写入
- 同时 HBase 又通过如下方式进行提高读性能。
  - 合并 SSTable 文件的个数
  - 布隆过滤器

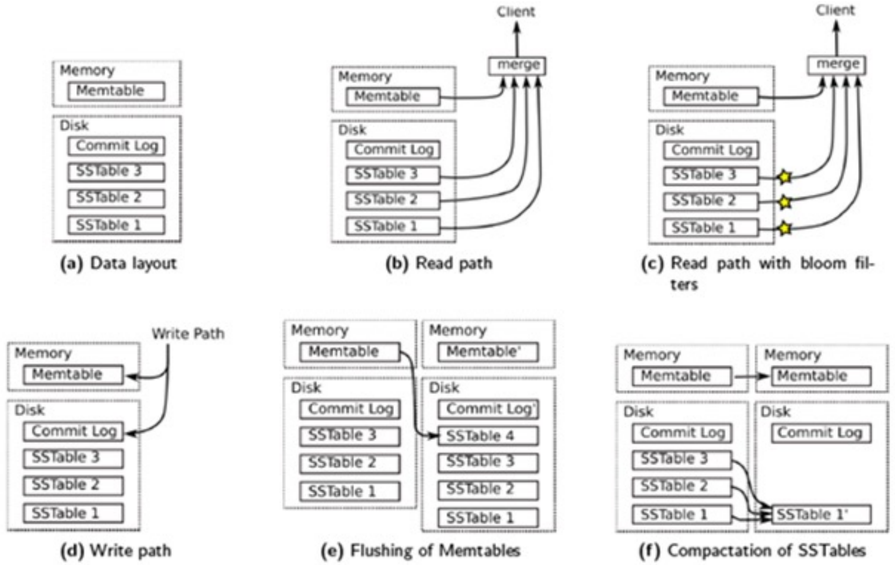
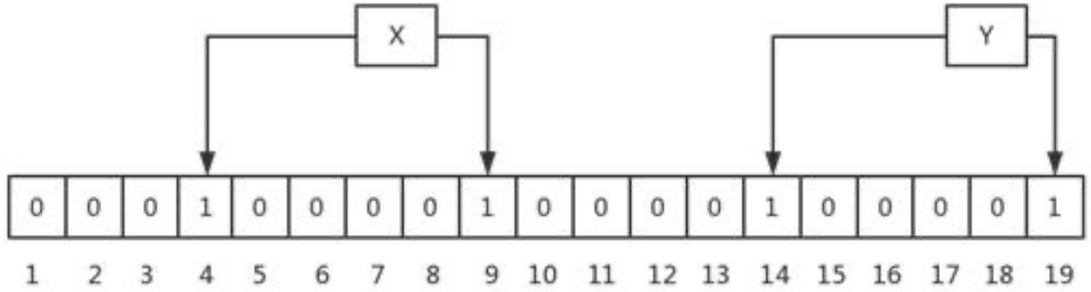


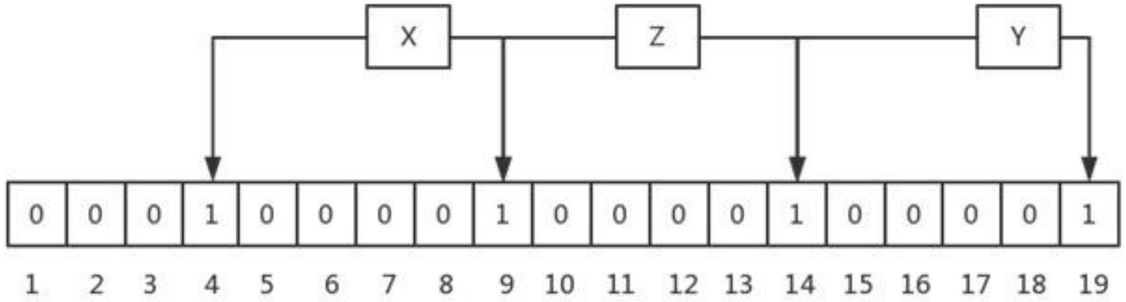
Figure 3.11.: Storage Layout – Log Structured Merge Trees (taken from [Lip09, slides 26–31])

布隆过滤器

- 布隆过滤器是一种基于概率的数据结构，主要用来判断某个元素是否在集合内。它具有运行速度快、占用内存小的优点，但是有一定的误识别率和删除困难的问题。它能够告诉你某个元素一定不在集合内或可能在集合内。



- 插入了两个元素，X 和 Y，X 的两次 hash 取模后的值分别为4,9，因此，4和9位被置成1；Y 的两次 hash 取模后的值分别为14和19，因此14和19位被置成1。



作业：代码实现布隆过滤器

极客时间 训练营

- 面试题：你有一个网站并且拥有很多访客，每当有用户访问时，你想知道这个 IP 是不是第一次访问你的网站（使用 guava 的布隆过滤器）。
- 代码实现布隆过滤器

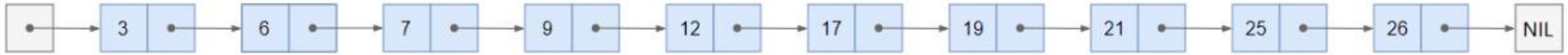
```
public class MyBloomFilter {
  public void add(String value)
  public boolean contains(String value)
}
```

- 一些应用
  - 避免缓存穿透
  - 网页爬虫对 URL 去重
  - 反垃圾邮件，从数十亿个垃圾邮件列表中判断某邮箱是否垃圾邮箱
  - 使用布隆过滤器避免推荐给用户已经读过的文章
  - NoSQL/KV 读加速

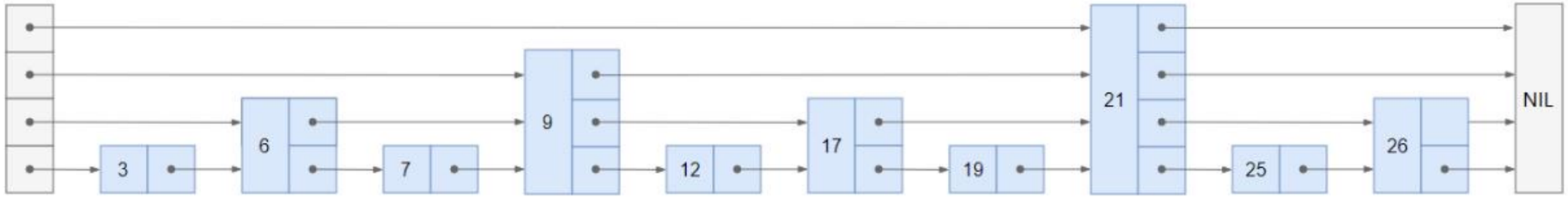
跳表

极客时间 训练营

- 跳表（skip list）以空间换时间提高查找性能，是一种插入/删除/搜索都是 O(log n) 的数据结构。它最大的优势是容易实现、效率更高。
- 二分查找确实很快，但是插入和删除元素的时候，为了保证元素的有序性，就需要大量的移动元素了。
- 在链表中，如果要搜索一个数，需要从头到尾比较每个元素是否匹配，直到找到匹配的数为止，即时间复杂度是 O(n)。同理，插入一个数并保持链表有序，需要先找到合适的插入位置再执行插入，总计也是O(n)的时间复杂度。

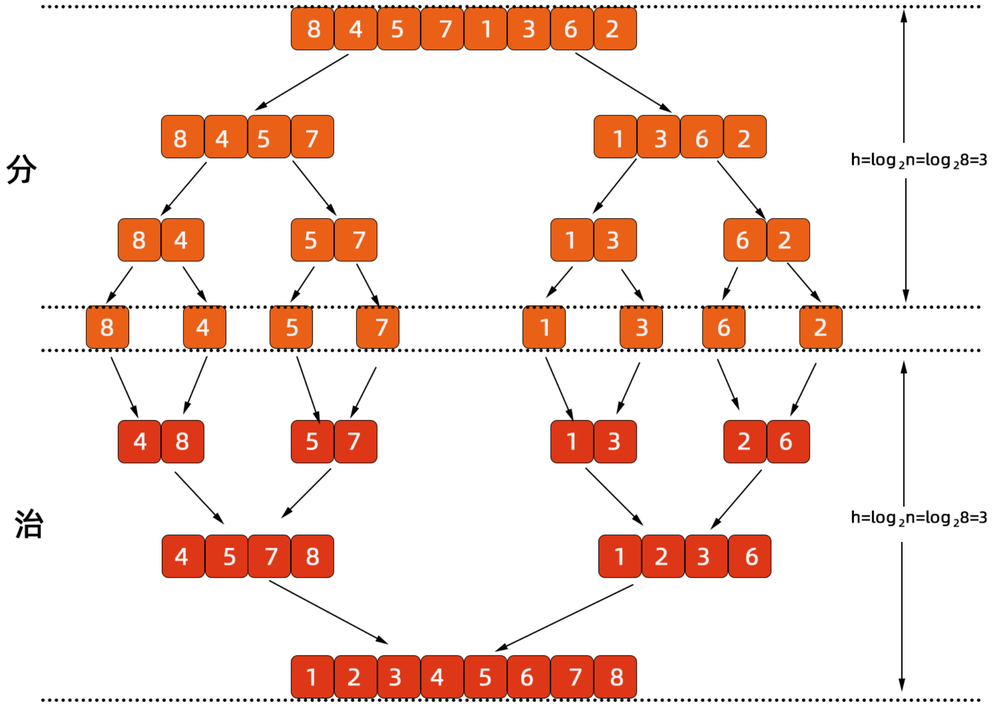


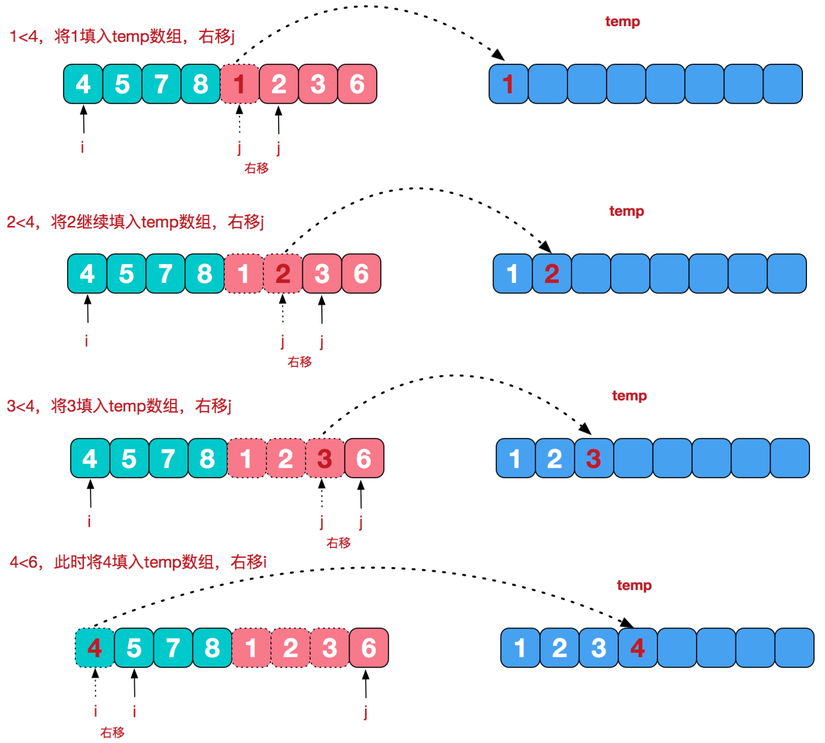
- 那么如何提高搜索的速度呢？很简单，做个索引：



并归排序

- 并归排序算法是一种分治模式，在数据库中经常使用
- 不能全部放在内存中的排序称为外排，而外排最常用的技术就是归并外排





2. Scala 基础

val 和 var

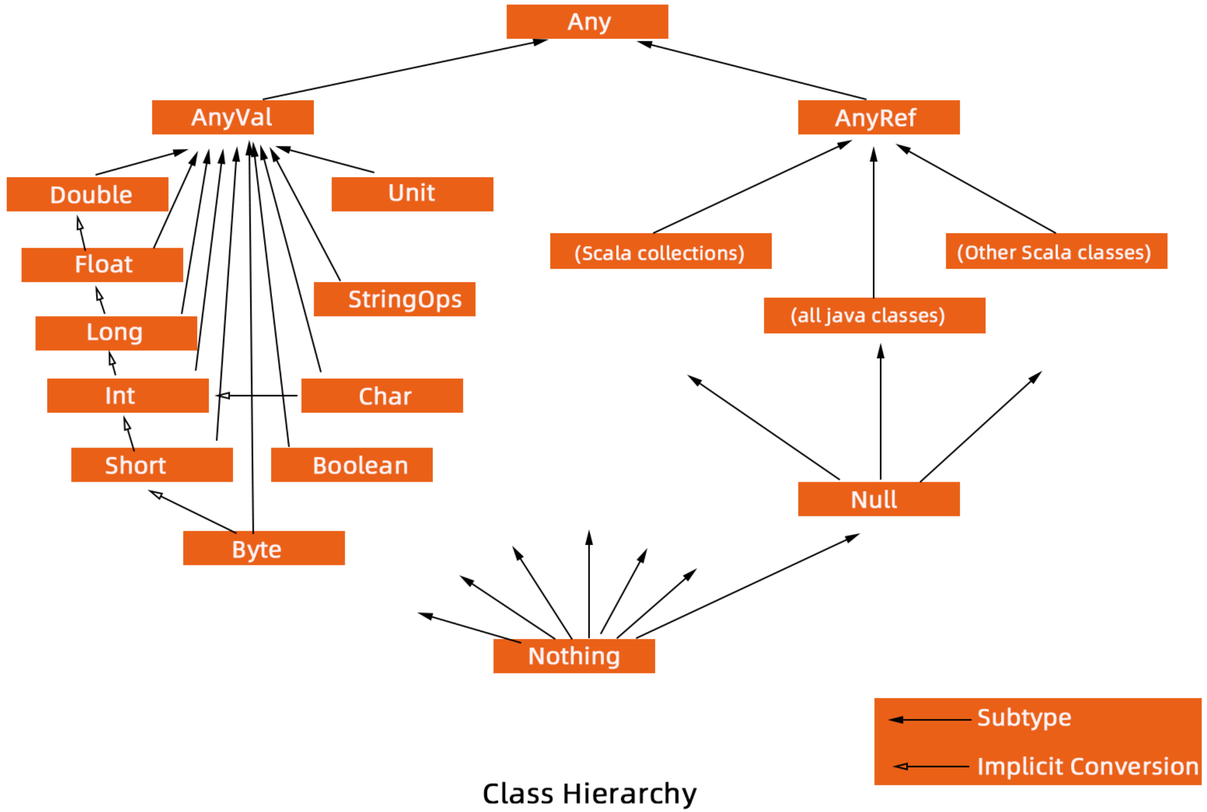
极客时间 训练营

- Scala 声明变量有两种方式，一个用 val，一个用 var
- val 也可以叫常量，相当于加了 Java 的 final 关键字
- val 定义的变量不能改变其引用的对象本身，但是可以改变其引用的对象的其他属性
- 尽可能使用 val
- 类型不需要指定，可自行推断

```
var v1 = ''code''
var v2: Int = 2
val v3 = 1 // 相当于final修饰
val v4: Double = 3.14
val v5 = Array(1, 2, 3)
v5(1) = 10
```

类

- Scala 所有的值都是类对象，继承自一个统一的根类型 Any。
- Null 是所有引用类型的子类型，null 是 Null 的唯一实例。
- Nothing 是所有类型的子类型，Nothing 没有实例，但是可以用来定义类型。例如：如果一个方法抛出异常，则异常的返回值类型就是 Nothing。
- Unit 类型用来标识过程，也就是没有返回值的函数，类似于 Java 里的 void。Unit 只有一个实例：()。



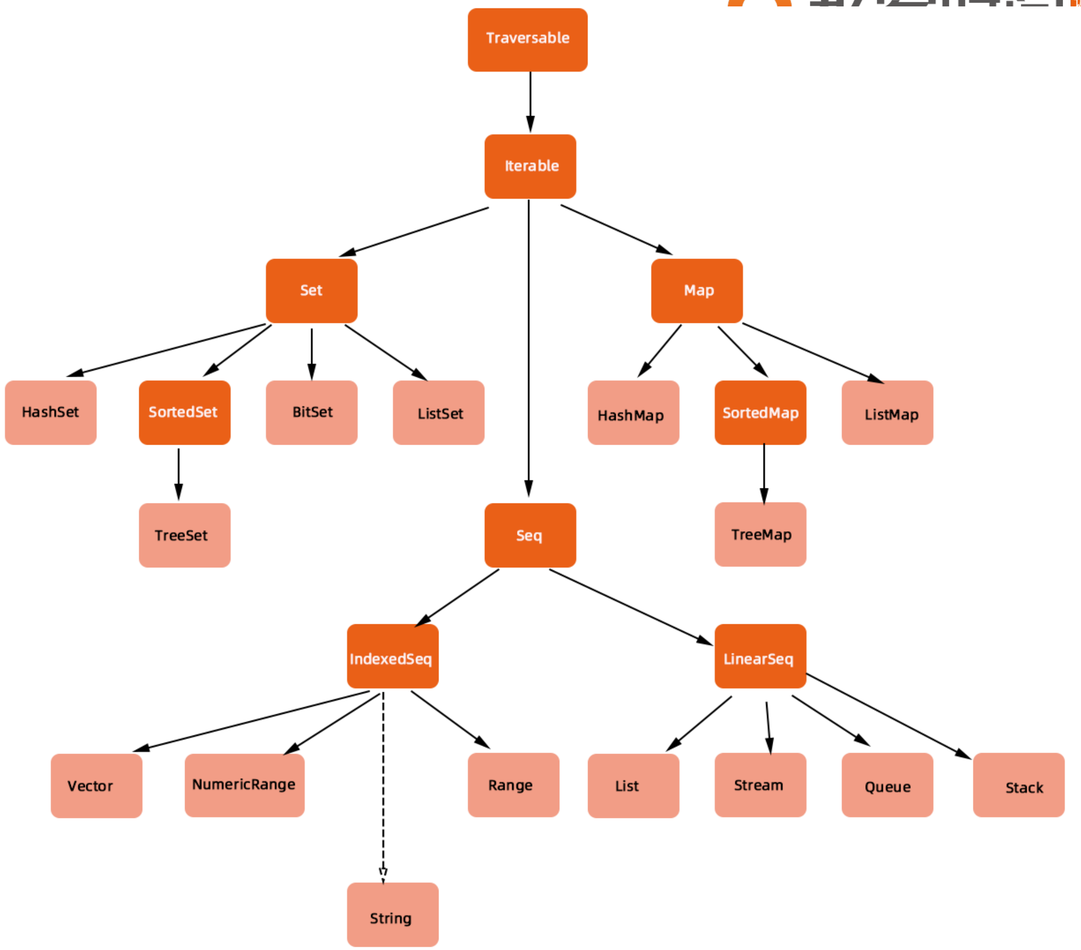
Class Hierarchy

循环

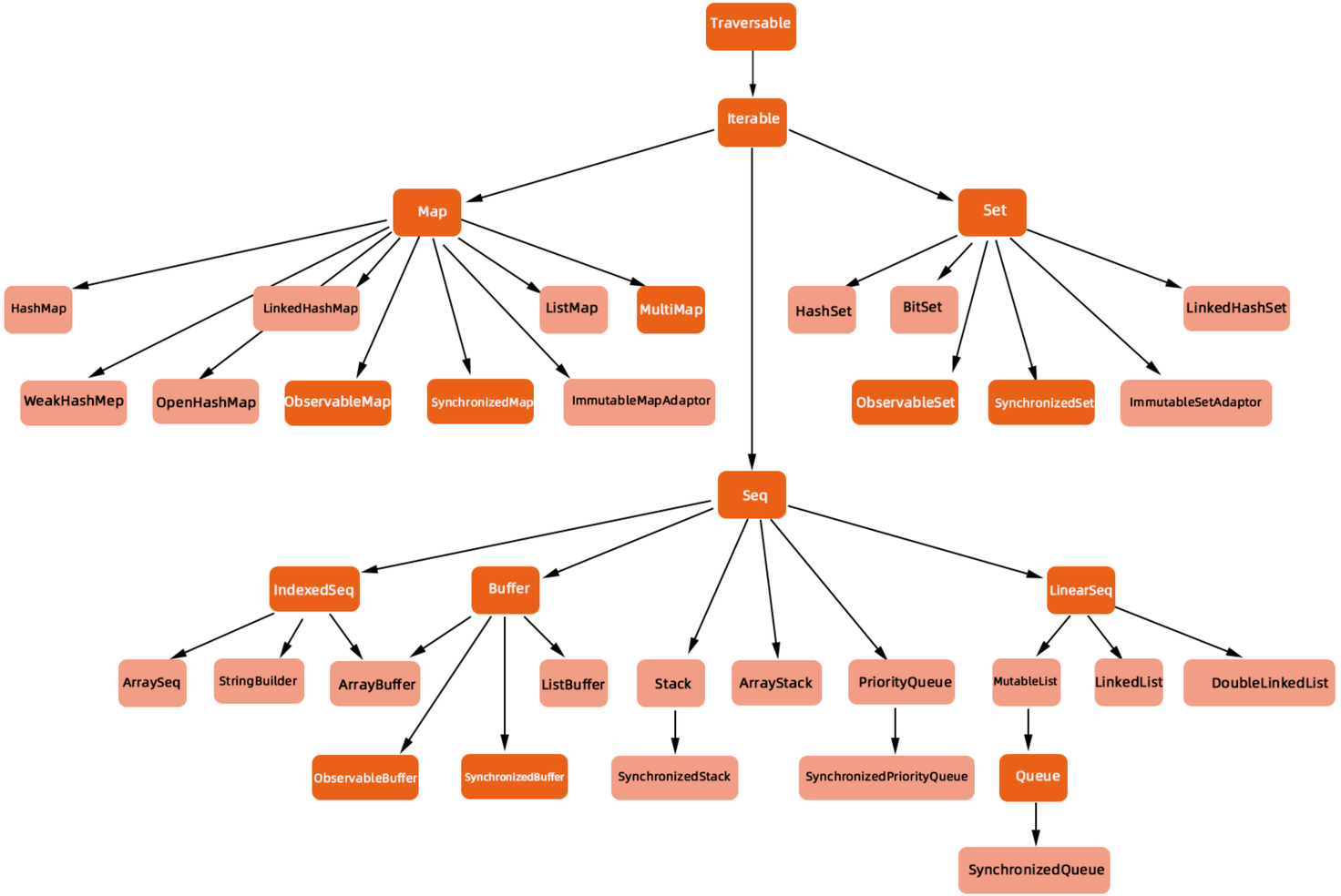
- while 循环：和 Java 一样
- for 循环
  - for 不支持 continue 和 break
  - for (i <- 表达式)
  - for (i <- 1 to 3) 和 for (i <- 1 until 3) 和 for(i <- (1 until 3).reverse)
- 循环守卫
  - for(i <- 1 to 10 if i %3= 0)
- 循环变量
  - for(i <- 1 to 3; j = 4 - i)
- 嵌套循环
  - for(i <- 1 to 3; j <- 1 to 3)
- 循环返回值
  - val x = for(i <- 1 to 10) yield i

集合操作

```
// 空列表
val empty = Nil
// ::构造列表
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
// 定义整型 List
val x = List(1, 2, 3, 4)
// 定义 Set
var x = Set(1, 3, 5, 7)
// 创建两个不同类型元素的元组
val x = (10, "Runoob")
// 定义 Map
val x = Map("one" -> 1, "two" -> 2, "three" -> 3)
```



集合操作


Traversable
Iterable
Map
Set
HashMap
LinkedHashMap
ListMap
MultiMap
HashSet
BitSet
LinkedHashSet
WeakHashMep
OpenHashMap
ObservableMap
SynchronizedMap
ImmutableMapAdaptor
ObservableSet
SynchronizedSet
ImmutableSetAdaptor
Seq
IndexedSeq
Buffer
LinearSeq
ArraySeq
StringBuilder
ArrayBuffer
ListBuffer
Stack
ArrayStack
PriorityQueue
MutableList
LinkedList
DoubleLinkedList
ObservableBuffer
SynchronizedBuffer
SynchronizedStack
SynchronizedPriorityQueue
Queue
SynchronizedQueue

case class

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

- 伴生对象

  编译 Person.scala 会产生两个 class，Person.class 和 Person$class. 编译器自动添加了一个伴生对象 object Person

- apply() 方法

  伴生对象 Person 里面默认实现了创建对象的 apply() 方法，创建实例的时候不需要使用关键字 new，可以直接通过 Person(age, name) 得到一个实例对象

- 字段默认加上了 val

  age 和 name 都被定义成了 val（final）

- toString()、hashCode() 和 equals()

  使用 age 和 name 构建

- 实现 Serializable 接口

trait

- 相当于 Java 中的接口，但可以定义抽象方法，也可以定义字段和方法的实现。
- 在没有自己的实现方法体时，可以认为它和 Java 的 interface 是等价的。
- 可以使用 extends 或 with 关键字把 trait 混入类中。

```
// 定义一个 trait
trait HasLegs {
    // 定义一个抽象字段
    val legs: Int
    // 定义一个具体的方法
    def walk(){
        println("Use" + legs + "legs to walk")
    }
    // 定义一个抽象方法
    def fly()
}
```

Option

极客时间 训练营

- 目的是为了避免使用 null，在 Java 里 null 是一个关键字，不是一个对象，所以对它调用任何方法都是非法的。
- Option 类型用来表示一个值是可选的（有值或无值）
- Option[T] 是一个类型为 T 的可选值的容器：如果值存在，Option[T] 就是一个 Some[T] ，如果不存在，Option[T] 就是对象 None
- Option 常用方法：
  - def isEmpty：Boolean 检测可选类型值是否为 None，是的话返回 true，否则返回 false。
  - def isDefined：Boolean 等价于 !isEmpty。
  - def getOrElse(default: => B): B 如果选项包含有值，返回选项值，否则返回设定的默认值。
  - def orElse(alternative: => Option[B]): Option[B] 如果选项包含有值返回选项，否则返回 alternative。
  - def orNull 如果选项包含有值返回选项值，否则返回 null。

模式匹配

极客时间 训练营

```
val nameMaybe: Option[String] = ...
nameMaybe match {
  case Some(name) =>
    println(name.trim.toUppercase)
  case None =>
    println("No name value")
}
```

```
v match {
    case x: Int => println("Int " + x)
    case y: Double if(y >= 0) => println("Double "+ y)
    case z: String => println("String " + z)
    case _ => throw new Exception("not match exception")
}
```

```
try {
    val r = 10 / 0
} catch {
    case ex: ArithmeticException=> println("捕获了除数为零的算数异常")
    case ex: Exception => println("捕获了异常")
}
```

3. 函数式编程

返回值

极客时间 训练营

- Scala 中，不需要使用 return 来返回函数的值，函数最后一行语句的值，就是函数的返回值。
- 将函数赋值给变量时，必须在函数后面加上“ _”或“(_)”。

```
def sayHello(name: String):String = {
   "Hello, " + name
}
```

函数赋值

极客时间 训练营

- Scala 中的函数是一等公民，可以直接将函数作为值赋值给变量。

```
def sayHello(name: String):String = {
    "Hello, " + name
}
val hi = sayHello _
val hey = sayHello(_)
hi("Lisa")
hey("Polo")
```

匿名函数

极客时间 训练营

- Scala 定义匿名函数的语法是 (参数名: 参数类型) => 函数体。

```
val hi = (name: String) => println("Hello, " + name)
hi("Lisa")
```

高阶函数

极客时间 训练营

- Scala 的函数是一等公民，可以直接将函数作为参数传入其他函数，也可以作为其他函数的返回值。
- 接收函数作为参数的函数，或返回值为函数的函数，被称作高阶函数（higher-order function）。

```
val hi = (name: String) => println("Hello, " + name)

def higherOrderHi(func: (String) => Unit, name: String) {
  func(name)
}

higherOrderHi(hi, "Lisa")
```

```
def higherOrderHi(hi: String) = (name: String) => println(hi+ ", " + name)

val hi = higherOrderHi("hello")

hi("Lisa")
```

常用高阶函数

极客时间 训练营

- map：对传入的每个元素都进行映射，返回一个处理后的元素。
  - Array(1, 2, 3, 4, 5).map(_ * 10)
- foreach：对传入的每个元素都进行处理，但是没有返回值。
  - Array(1, 2, 3, 4, 5).map(_ * 10).foreach(println _)
- filter：对传入的每个元素都进行条件判断，如果对元素返回 true，则保留该元素，否则过滤掉该元素。
  - (1 to 50).filter(_ % 10 == 0)
- reduceLeft：从左侧元素开始，进行 reduce 操作，即先对元素1和元素2进行处理，然后将结果与元素3处理，再将结果与元素4处理，依次类推，即为 reduce。
  - (1 to 4).reduceLeft(_ * _)

闭包

- 闭包是一个函数，返回值依赖于函数外部的一个或多个变量。

```
val multiplier = (i:Int) => i * 10
val multiplier = (i:Int) => i * factor
```

- 在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形参，在函数被调用时，i 被赋予一个新的值。但 factor 不是形参，而是自由变量。

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

- 这里引入一个自由变量 factor，这个变量定义在函数外面。
- 函数变量 multiplier 成为一个"闭包"，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数，形成闭包。

Spark 闭包问题

极客时间 训练营

- bin/spark-shell

```
scala> class User(name: String)
scala> val user1 = new User("jinlantao")
scala> val rdd1 = sc.makeRDD(List(user1))
scala> rdd1.foreach(println)
ERROR Utils: Exception encountered
java.io.NotSerializableException: $line14.$read$$iw$$iw$User
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
  at java.io.ObjectOutputStream.writeArray(ObjectOutputStream.java:1378)
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1174)
  at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
  at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)
  at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)
  at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
  at java.io.ObjectOutputStream.defaultWriteObject(ObjectOutputStream.java:441)
  at org.apache.spark.rdd.ParallelCollectionPartition.$anonfun$writeObject$1(ParallelCollectionRDD.scala:58)
  at scala.runtime.java8.JFunction0$mcV$sp.apply(JFunction0$mcV$sp.java:23)
  at org.apache.spark.util.Utils$.tryOrIOException(Utils.scala:1405)
```

- case class User(name: String)

柯里化 Currying

极客时间 训练营

- 柯里化指将原来接收 N 个参数的一个函数，转换为 N 个函数。

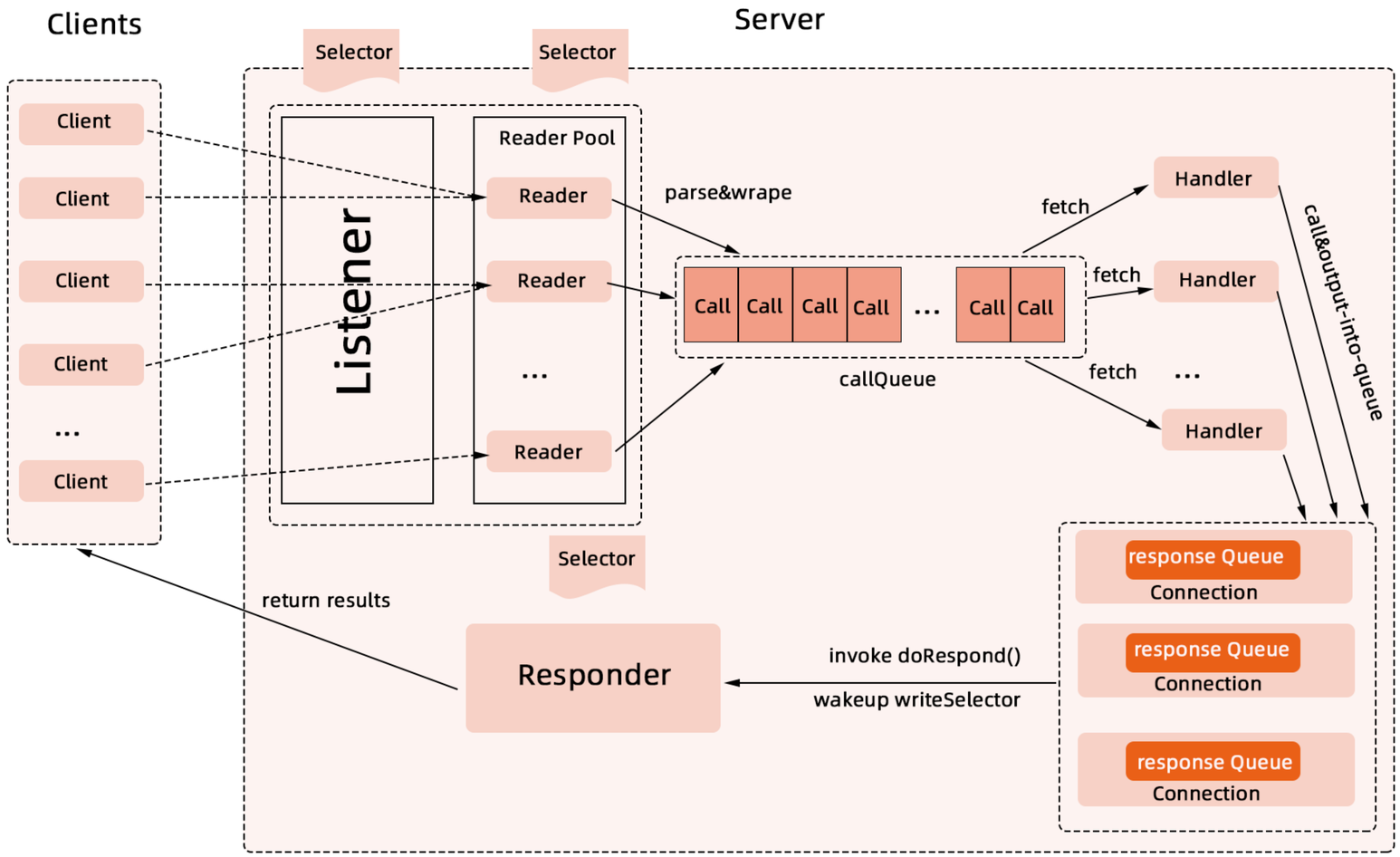
```
def twoAdd1(a: Int, b: Int) = a + b
add(1, 2)

def twoAdd2(a: Int) = (b: Int) => a + b
add(1)(2)

def twoAdd3(a: Int)(b: Int) = a + b
add(1)(2)

def threeAdd1(a: Int, b: Int, c: Int) = a + b + c
add(1, 2, 3)

def threeAdd3(a: Int)(b: Int)(c: Int) = a + b + c
add(1)(2)(3)

def threeAdd2?
```

- def threeAdd2(a: Int) = (b: Int) => (c: Int) => a + b + c

4. Hadoop RPC 源码

PRC Server 框架结构

1. 建立连接
2. 接受请求
3. 处理请求
4. 返回结果



主要代码块

- Server.Listener
  - run() 监听来自客户端的 Socket 连接请求，通过 Selector 监听 OP_ACCEPT 事件
  - doAccept() 收到来自客户端的 Socket 连接请求后初始化连接 Connection，从 readers 线程池中选一个 Reader 线程处理，并在 readSelector 注册一个 OP_READ 事件
- Server.Reader
  - run() 监听当前 Reader 对象的连接中是否有 RPC 请求到达
  - doRead() 找到 Connection 对象并读取 RPC 请求
- Server.Connection
  - readAndProcess -> processOneRpc -> processData -> callQueue (Handler)
- Server.Handler
  - run()
  - setupResponse() -> doResponse() -> processResponse()
- Server.Responder
  - run() 监听 OP_WRITE 事件
  - doAsyncWrite()
  - processResponse()

参考链接：https://github.com/LantaoJin/commons-rpc

学习型项目

- https://github.com/LantaoJin/commons-rpc
- https://github.com/LantaoJin/DistributedSystemUsingJavaNIO

参考链接：https://github.com/LantaoJin/commons-rpc

5. Spark Core 源码

Job 和 Stage 的生成和提交

极客时间 训练营

- RDD.count()
  - def count(): Long = sc.runJob(this, Utils.getIteratorSize _).sum
- SparkContext.runJob()
- DAGScheduler.runJob()
- DAGScheduler.submitJob()
  - eventProcessLoop.post(JobSubmitted)
- DAGSchedulerEventProcessLoop.onReceive() -> DAGScheduler.handleJobSubmitted() // 开始 Stage 的划分
  - createResultStage()
    - getOrCreateParentStages() // 获取其 Parent Stages，即 ShuffleMapStage 列表
    - new ResultStage() // 生成最后一个 Stage，即 ResultStage
  - 生成 ActiveJob 对象
  - listenerBus.post(SparkListenerJobStart)
  - submitStage(finalStage) // Stage 提交

Job 和 Stage 的生成和提交

- submitStage() //方法是一个递归方法
  - getMissingParentStages(stage).sortBy(_.id) //先找出 parent stages 并提交
  - 再提交自己
  - submitMissingTasks(stage, jobId.get) //真正的提交是提交 TaskSet
- submitMissingTasks()
  - taskIdToLocations() // 生成具有 location 信息的 task 列表
  - listenerBus.post(SparkListenerStageSubmitted)
  - taskBinary = sc.broadcast(taskBinaryBytes) // 序列化 task 并广播
  - TaskSchedulerImpl.submitTasks(new TaskSet()) // 封装成 taskSet 并提交
- CoarseGrainedSchedulerBackend.reviveOffers()
  - driverEndpoint.send(ReviveOffers)

Task 的序列化、发送和执行

- DriverEndpoint.receive() -> case ReviveOffers => makeOffer()
- makeOffer()
  - TaskSchedulerImpl.resourceOffers() // 根据资源和 task 的 location 进行本地化
  - launchTasks(taskDescs)
    - executorData.executorEndpoint.send(LaunchTask())
- CoarseGrainedExecutorBackend.receive() -> case LaunchTask(data)
- Executor.launchTask(this, taskDesc)
  - val tr = new TaskRunner() + threadPool.execute(tr)
- TaskRunner.run()

Task 的序列化、发送和执行

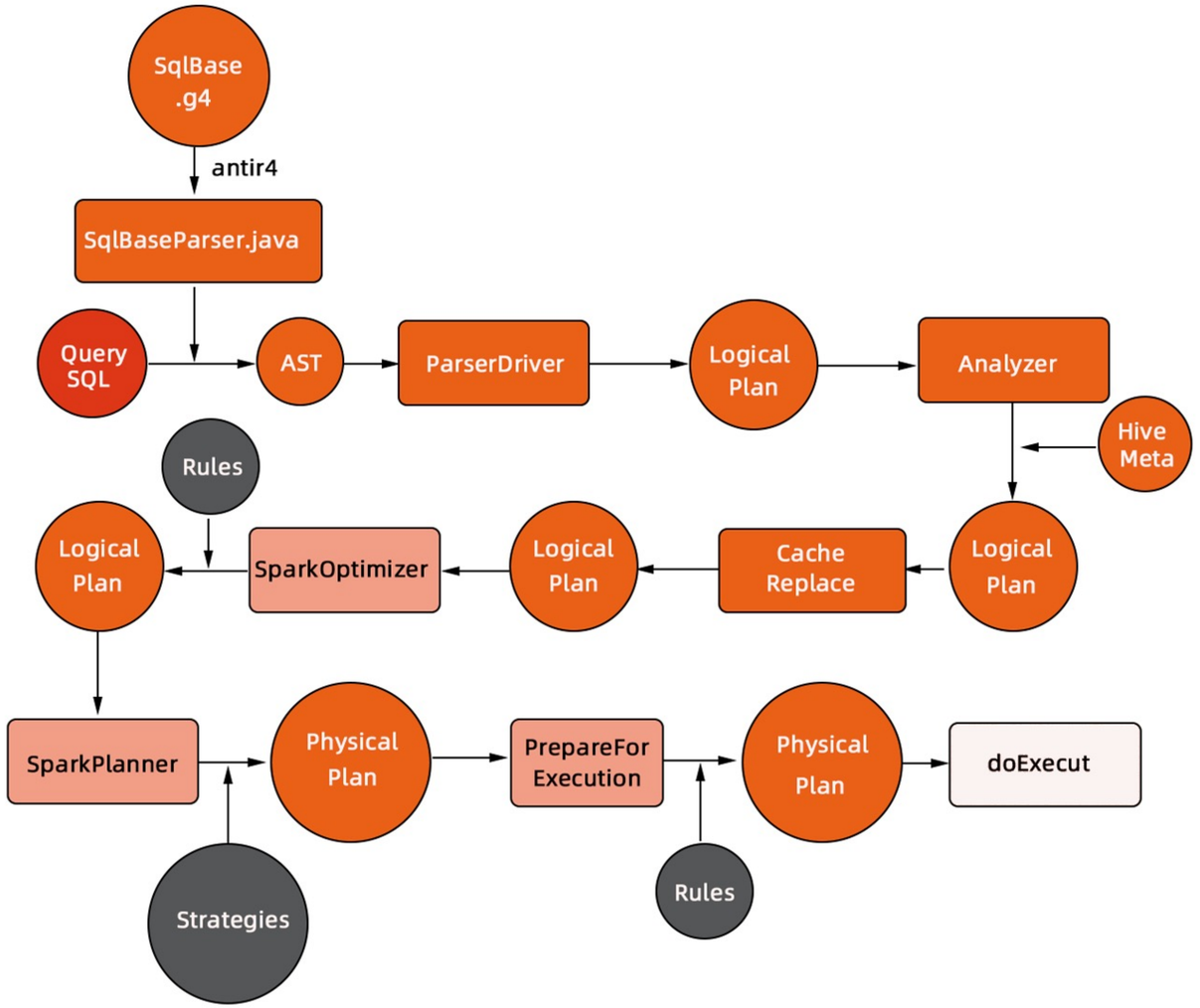
- task = ser.deserialize[Task[Any]] // 反序列化 task
- val value = task.run()
  - runTask(context) // ShuffleMapTask.runTask() 或者 ResultTask.runTask()
- ShuffleMapTask.runTask()
  - val rddAndDep = ser.deserialize // 反序列化出RDD对象和Dep对象
  - ShuffleWriteProcessor.write() // 写shuffle文件和mapStatus
- ResultTask.runTask()
  - val rddAndDep = ser.deserialize // 反序列化出RDD对象和func函数
  - func(context, rdd.iterator(partition, context)) // 调用
  - 在我们这个 count 的例子中，func 就是 Utils.getIteratorSize()
- val serializedResult = resultSer.serialize(value) // 将结果(mapStatus或result）序列化
- CoarseGrainedExecutorBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)
  - driverRef.send(StatusUpdate)

Task/Stage/Job 的结束

- CoarseGrainedSchedulerBackend.receive() -> case StatusUpdate
  - TaskSchedulerImpl.statusUpdate() // 对成功或失败任务进行处理
    - TaskResultGetter.enqueueSuccessfulTask(taskSet, tid, serializedData)
  - makeOffers(executorId) // 完成了一个task，获得资源释放，继续尝试提交
- task-result-getter.run()
  - TaskSchedulerImpl.handleSuccessfulTask() 或 TaskSchedulerImpl.handleFailedTask()
    - TaskSetManager.handleSuccessfulTask()
      - sched.backend.killTask // 杀到其他还在跑的task attempt
      - sched.dagScheduler.taskEnded()
        - DAGSchedulerEventProcessLoop.post(CompletionEvent)
      - maybeFinishTaskSet() // 将这个Stage的runningTasks置为0
- DAGSchedulerEventProcessLoop.receive() -> case CompletionEvent
  - handleTaskCompletion() // 处理task完成后Stage的情况，如failStage，如果runningTasks==0，markStageAsFinished()
    - listenerBus.post(SparkListenerStageCompleted())

6. Spark SQL 源码

一条 SQL 语句在 sql() 中的执行过程



7. AQE 源码

入口

- QueryExecution.executedPlan
  - preparations
    - Option(InsertAdaptiveSparkPlan(AdaptiveExecutionContext())
  - InsertAdaptiveSparkPlan.apply
    - AdaptiveSparkPlanExec.applyPhysicalRules(plan, planSubqueriesRule) //子查询
    - retrun AdaptiveSparkPlanExec(newPlan)
  - AdaptiveSparkPlanExec.doExecute()
    - getFinalPhysicalPlan()
      - createQueryStages(currentPhysicalPlan) // 递归构建QueryStage
        - newQueryStage

AQE 物理规则

极客时间 训练营

- newQueryStage中val optimizedPlan = applyPhysicalRules

  PlanAdaptiveDynamicPruningFilters(this),

  ReuseAdaptiveSubquery(context.subqueryCache),

  OptimizeSkewedJoin,

  OptimizeSkewInRebalancePartitions,

  CoalesceShufflePartitions(context.session),

  OptimizeLocalShuffleReader

  ApplyColumnarRulesAndInsertTransitions(sessionState.columnarRules),

  CollapseCodegenStages()

newQueryStage 的再优化

- getFinalPhysicalPlan()
  - val stage = newQueryStage()中根据规则获得 QueryStageExec
    - ShuffleQueryStageExec
    - BroadcastQueryStageExec
  - reorderedNewStages //BroadcastQueryStageExec 需要前置
  - stage.materialize()
    - ShuffleExchangeExec.mapOutputStatisticsFuture // submit MapStage
      - sparkContext.submitMapStage(shuffleDependency)
    - BroadcastQueryStageExec.materializeWithTimeout
      - BroadcastExchangeExec.submitBroadcastJob // BC 是 job
  - val logicalPlan = replaceWithQueryStagesInLogicalPlan //获得新的逻辑计划
  - val (newPhysicalPlan, newLogicalPlan) = reOptimize(logicalPlan)
    - AQEOptimizer.execute(logicalPlan)

AQE 逻辑优化规则

极客时间 训练营

- AQEOptimizer

```
Batch("Propagate Empty Relations", fixedPoint,
  AQEPropagateEmptyRelation,
  ConvertToLocalRelation,
  UpdateAttributeNullability),
Batch("Dynamic Join Selection", Once, DynamicJoinSelection)
```

finalPhysicalPlan 的执行

极客时间 训练营

- getFinalPhysicalPlan

```
// Run the final plan when there's no more unfinished stages.
currentPhysicalPlan = applyPhysicalRules(
      result.newPlan,
      finalStageOptimizerRules,
      Some((planChangeLogger, "AQE Final Query Stage Optimization")))
isFinalPlan = true
executionId.foreach(onUpdatePlan(_, Seq(currentPhysicalPlan)))
currentPhysicalPlan
```

- doExecute()
  - val rdd = getFinalPhysicalPlan().execute()

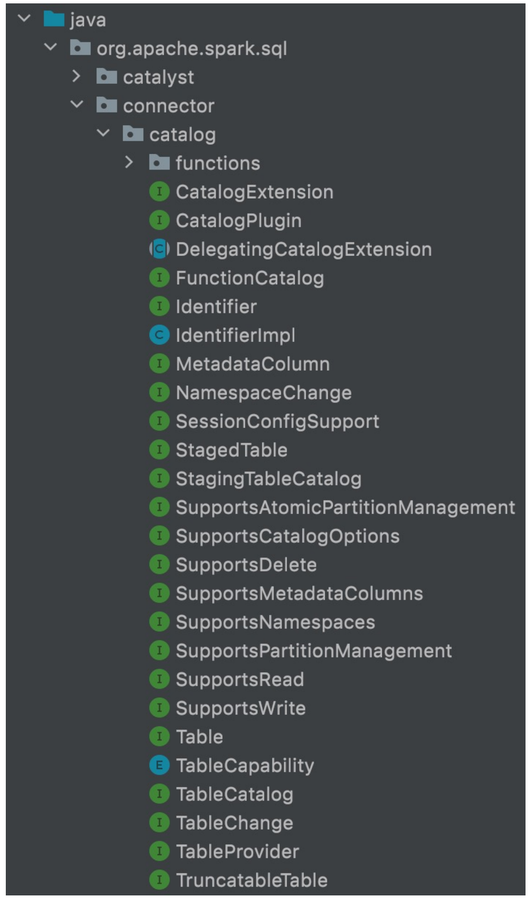
8. DataSource V2 源码

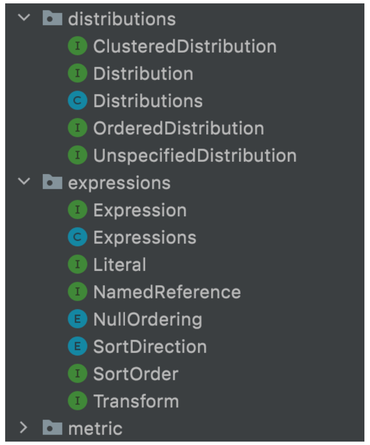
DataSourceV1 的不足

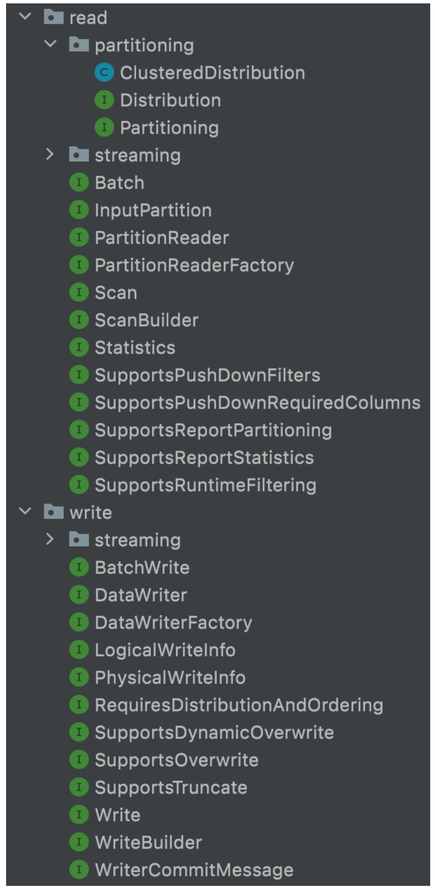
- 部分接口依赖 SQLContext 和 DataFrame
- 扩展能力有限，难以下推其他算子
- 缺乏对列式存储读取的支持
- 缺乏分区和排序信息
- 写操作不支持事务
- 不支持流处理

Package

- Java 代码：package org.apache.spark.sql.connector

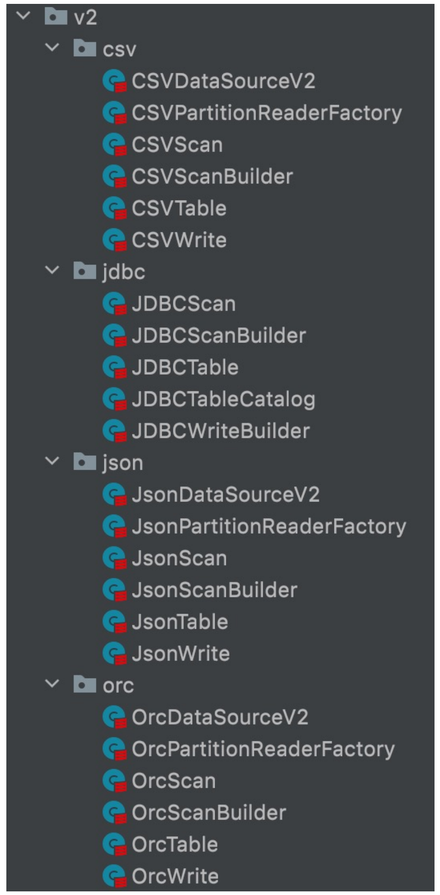


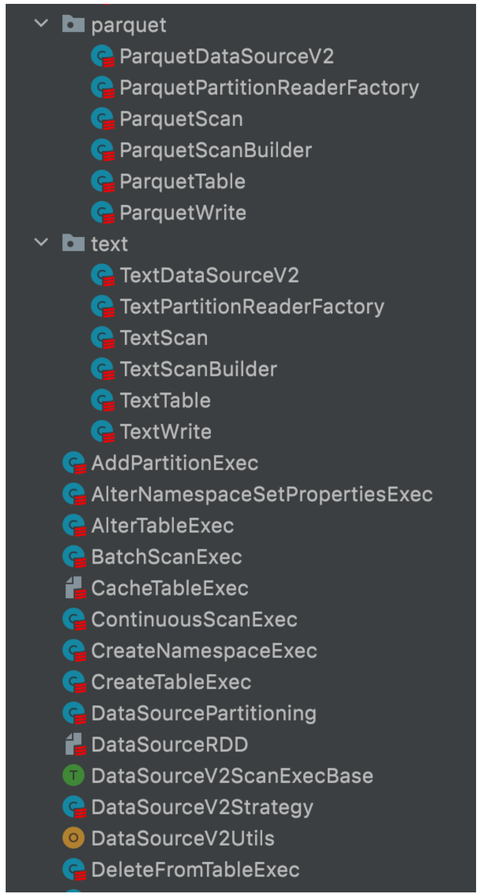


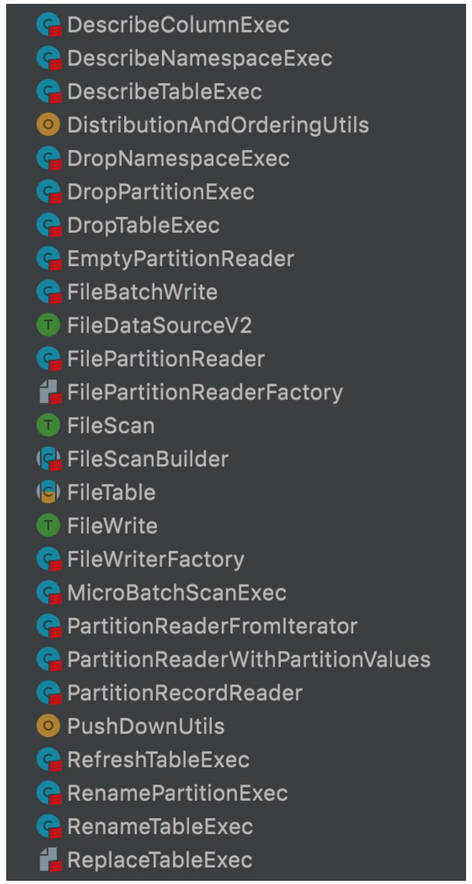


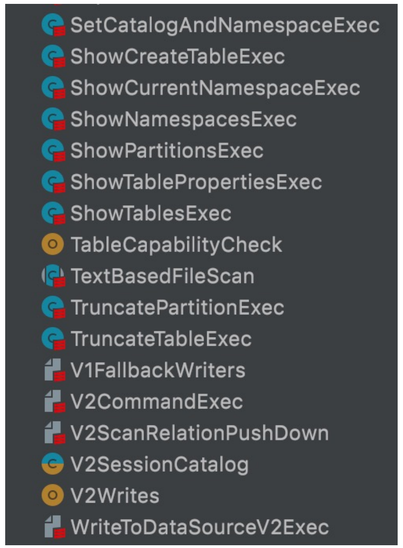
Package

- Scala 代码：package org.apache.spark.sql.execution.datasources.v2









JDBC V2

```scala
case class JDBCTable(ident: Identifier, schema: StructType, jdbcOptions: JDBCOptions)
  extends Table with SupportsRead with SupportsWrite {

  override def name(): String = ident.toString

  override def capabilities(): util.Set[TableCapability] = {
    Set(BATCH_READ, V1_BATCH_WRITE, TRUNCATE).asJava
  }

  override def newScanBuilder(options: CaseInsensitiveStringMap): JDBCScanBuilder = {
    val mergedOptions = new JDBCOptions(
      jdbcOptions.parameters.originalMap ++ options.asCaseSensitiveMap().asScala)
    JDBCScanBuilder(SparkSession.active, schema, mergedOptions)
  }

  override def newWriteBuilder(info: LogicalWriteInfo): WriteBuilder = {
    val mergedOptions = new JdbcOptionsInWrite(
      jdbcOptions.parameters.originalMap ++ info.options.asCaseSensitiveMap().asScala)
    JDBCWriteBuilder(schema, mergedOptions)
  }
}
```

选做作业：RCFile 的 DSV2 实现

THANKS

极客时间 | 训练营