Spark/Scala 实践

来自于编码、Code Review、同事讨论的一些思考总结

Xinyan Lu (xinyanlu@tencent.com)

简洁代码（使用 implicit parameter）

- 当函数的 implicit parameter 缺失时，会自动从调用代码块的上下文中寻找唯一合适的参数（编译时），简化调用时的函数参数
  - 常用的如 SparkContext 对象

```
def prepareSeeds(taskBagIds: Seq[Int])(implicit sc: SparkContext): RDD[String] = {
  // some code
}

implicit val sc = initSparkContext()
val seeds = prepareSeeds(taskBagIds)
```

简洁代码（使用 implicit class）

- 使用 implicit class 可以为一个类“新增”方法，增强可读性，简化重复代码

- 例如，新增一个 *createListPartitionIfNotExist* 方法给 TDWUtil:

```
implicit class TDWUtilWrapper(tdwUtil: TDWUtil) {
  def createListPartitionIfNotExist(table: String,
                                    partition: String,
                                    partitionValue: String): Unit = {
    if (!tdwUtil.partitionExist(table, partition)) {
      tdwUtil.createListPartition(table, partition, partitionValue)
    }
  }
}
```

- 试想，如果在日志 PB 对象上使用，在解析的时候能简化多少代码？

代码简洁（使用 apply）

- 当类或对象有一个主要用途的时候，apply 方法提供一个很好的语法糖
- obj(x) 等价于 obj.apply(x)

代码简洁（少写括号）

- 使用模式匹配时，case 后不需要大括号

```
dateStr.length match {
  case 10 => Calendar.HOUR
  case 8  => Calendar.DATE
  case 6  => Calendar.MONTH
  case _ =>
    throw new IllegalArgumentException(
      s"Invalid argument dateStr: $dateStr.")
}
```

- scala 函数中的 foo(x) 和 foo{x} 是一回事，当 x 是多行的匿名函数时，用后者可以省一对小括号

```
// good
rdd.map {x =>
  // some multi-line code
}
// bad
rdd.map (x => {
  // some multi-line code
})
```

代码简洁（多用简写）

- 多用下划线，甚至不用下划线

```
// bad
collec.foreach(x => println(x))
// good
collec.foreach(println(_))
// better
collec.foreach(println)
```

- 变量的值不重要的时候，也可以用下划线

```
List(1, 2, 3).foreach{ _ => println("Hi") }
```

多用 val，尽量避免用 var

- 同理，不可变集合优先于可变集合
- 使用不可变的好处：
  - 安全，不用担心其他代码会改变它的值
  - 无需维护
  - 提高代码可读性，如果使用 var 的话：
    - 需要通读，才能确定某变量在某片代码上的值
    - 在同一代码片上，是否会随着运行，被别的地方修改了
    - 容易忘记在某个时刻重新初始化，导致 bug
- 如果可以，甚至不使用 val：
  - 一句链式表达式搞定，将结果返回

样例：使用 val 替代 var

```
// bad
var x = 0
if (condition) {
  x = 1
} else {
  x = -1
}

// good
val x = if (condition) 1 else -1
```

```
// bad
val s = Seq(1.0, 2.0, 3.0)
var num = 0
var sum = 0.0
s.foreach{ x =>
  num += 1
  sum += x
}
val mean = sum / num

// good
val mean = s.sum / s.length
```

不使用 getXx / setXx 操作属性

- getXx / setXx 是 Java 的习惯，在 scala 里习惯这样用：

```
class Foo {
  private var v: Int = _

  // getter, no parentheses
  def xx: Int = v

  // setter
  def xx_=(x: Int): Unit = { v = x }
}

val f = new Foo
f.xx = 10
println(f.xx)
```

使用 package object

- 当一些常量 / 辅助函数需要对整个 package 可见时，考虑使用 package object

使用 Option，不要自定义错误符

- 在其他语言里，使用不符合预期的返回值（例如 null）来提醒调用方错误
  - 需要看程序注释进行了解自定义错误符

- 在 scala 里，当返回值是一个 Option[T] 对象时，暗示函数有可能会返回空
  - Option[T] 有两个子类，分别是 Some[T] 和 None[T]
    - Some[T] 是一个仅有 1 个 T 元素的集合，这个元素就是正常值
    - None[T] 不含有任何元素，表示空
- 调用方使用 pattern matching 进行错误处理
- 同理，这套模式也可使用在未初始化的对象上

使用 lazy val

- 在 scala 里，单例 object 是 lazy 的 （用到时才初始化）
  - 一个工程里超多单例不会相互影响性能
- 但一个单例实例化时，其成员均会初始化
  - 对于一个庞大的单例，有必要对其成员使用 lazy val 进行惰性求值
  - 特别是耗时较久的初始化
    - 例如：含有读取配置文件、建立数据库连接等 IO 操作

需要抛异常的时候就坚决抛异常

- 尽量不要使用 case _: Exception => … 来捕获全部异常
- 强行屏蔽异常，可能会导致出问题的时候难以排查源头

善用 Intellij Idea 和 scalafmt 插件

- 重视 Idea 的智能提示
  - 英文拼写错误
  - 编码时提示错误（例如类型错误）
  - Code inspection (settings -> editor -> inspections)
    - 完整看一遍 inspection 也大有裨益
- menu -> code -> auto format imports (Ctrl + Alt + O)
- menu -> code -> reformat with scalafmt (Ctrl + shift + L)
  - 仅需默认配置，在代码风格上就没有什么问题了

用模式匹配来注释 RDD 里的元组

- 在 RDD 里，最常使用的元素是元组。特别是，二元组的第 1 元素和第 2 元素分别对应了 PairedRDD 中的 key 和 value。
- 随着不断进行 RDD 操作，元组会一直发生变化（位置、含义），需要良好的注释才能理解代码。
- 使用模式匹配来进行元组“自解释”：

```
curr.foreach { meta =>
  meta.foreach {
    case (bagId, (embedding, threshold, auc)) =>
      // some code
  }
}
```

写数据之前，先删 HDFS 目录

- 同理，先删 TDW 表的数据分区
- 尽管这违背MR/Spark的哲学，但在集群频繁升级、迁移、重启的现实下：
  - 执行中的任务被杀，数据写到一半
  - 重启后任务自动拉起，但输出目录已存在，重试任务会失败
  - 人为干预等于加班

```
def deleteHdfsPath(sc: SparkContext, path: String): Boolean = {
  val p = new Path(path)
  val fs = p.getFileSystem(sc.hadoopConfiguration)
  if (fs.exists(p)) {
    fs.delete(p, true)
  } else {
    false
  }
}
```

选择 Persist RDD 的 StorageLevel

- 根据 sharkdtu 的实验（参考 KM 文章）：
  - 内存充足时，优先考虑使用 MEMORY_ONLY
  - 当内存不足以 Cache 住中间数据时：
    - 建议用 MEMORY_ONLY_SER (spark.rdd.compress=true) 或者 DISK_ONLY
    - 不要用MEMORY_AND_DISK，MEMORY_AND_DISK可能会频繁地触发Spark的内存管理，增加Spill以及GC的开销

大胆使用 groupByKey

- groupByKey 不是禁用的 API，只是由于性能缘故需要避免使用
- 并不是所有操作都能很容易地转成 reduceByKey 或者 combineByKey 操作，在数据量可控（性能可控）的情况下，为了代码的可读性，大胆使用 groupByKey

减少需要输入的命令行参数

- 对于例行任务，用命令行参数指定 HDFS 路径/ TDW 表：
  - 重复输入，一个任务的输出通常是其下游的输入
  - 路径较长，编辑困难（易错）

- 称之为可变常量
  - 一旦指定，运行时保持不变
  - 不损害灵活性的前提下，使用 config 文件进行配置

样例：使用 TypeSafe 的 Config Library

application.conf (有层次结构的配置)

```
hdfs {
  // 注意，如果是HDFS目录的话，请以"/"结尾。
 base_dir = "hdfs://tl-nn-tdw.tencent-distribute.com:54310/stage/outface/sng/gdt/lookalike/"
  // 种子目录
 seed_dir = ${hdfs.base_dir}"/history_seed/"
  // 配置文件目录
 conf_dir = ${hdfs.base_dir}"/history_conf/"
  // 训练数据目录
 train_data_dir = ${hdfs.base_dir}"/history_train_data/"
}
```

code.scala

```
import com.typesafe.config.ConfigFactory
val config = ConfigFactory.load()
config.getString("hdfs.seed_dir")
```

不要随意新增第三方库依赖

- 不加限制地依赖第三方库会导致 Jar 打包体积急速变大
  - 需要上传几十上百兆的 Jar
- 像日期类、Http 连接类均有 JDK 自带的实现（满足绝大多数需求）
- 如果实在要用，优先使用 Spark 集群中已有的同一版本的第三方库，同时将这个库设为 Provided
  - Spark 任务运行的 Web UI 能显示当前集群所有的 Jar