累加器、广播变量

1. `spark` 中的累加器(`accumulator`) 和广播变量(`broadcast variable`) 都是共享变量（所谓共享，就是在驱动器程序和工作节点之间共享）
    - 累加器用于对信息进行聚合
    - 广播变量用于高效的分发较大的对象

一、累加器

1. 在集群中执行代码时，一个难点是：理解变量和方法的范围、生命周期。下面是一个闭包的例子：

    ```
    counter = 0
    rdd = sc.parallelize(data)

    def increment_counter(x):
        global counter
        counter += x
    rdd.foreach(increment_counter)
    print("Counter value: ", counter)
    ```

    上述代码的行为是不确定的，并且无法按照预期正常工作。

2. 在执行作业时， `spark` 会分解 `RDD` 操作到每个 `executor` 的 `task` 中。在执行之前， `spark` 计算任务的闭包
    - 所谓闭包：指的是 `executor` 要在 `RDD` 上进行计算时，必须对执行节点可见的那些变量和方法
    - 闭包被序列化，并被发送到每个 `executor`
3. 在上述代码中，闭包的变量的副本被发送给每个 `executor` ，当 `counter` 被 `foreach` 函数引用时，它已经不再是驱动器节点的 `counter` 了
    - 虽然驱动器程序中，仍然有一个 `counter` 在内存中；但是对于 `executors` ，它是不可见的。
    - `executor` 看到的只是序列化的闭包的一个副本。所有对 `counter` 的操作都是在 `executor` 的本地进行。
    - 要想正确实现预期目标，则需要使用累加器

1.1 Accumulator

1. 一个累加器(`Accumulator`)变量只支持累加操作
    - 工作节点和驱动器程序对它都可以执行 `+=` 操作，但是只有驱动器程序可以访问它的值。

        > 在工作节点上，累加器对象看起来就像是一个只写的变量

    - 工作节点对它执行的任何累加，都将自动的传播到驱动器程序中。
2. `SparkContext` 的累加器变量只支持基本的数据类型，如 `int、float` 等。
    - 你可以通过 `AccumulatorParam` 来实现自定义的累加器
3. `Accumulator` 的方法：

   - `.add(term)`：向累加器中增加值`term`
4. `Accumulator`的属性：
   - `.value`：获取累加器的值。只可以在驱动器程序中使用
5. 通常使用累加器的流程为：
   - 在驱动器程序中调用`SparkContext.accumulator(init_value)`来创建出带有初始值的累加器
   - 在执行器的代码中使用累加器的`+=`方法或者`.add(term)`方法来增加累加器的值
   - 在驱动器程序中使用累加器的`.value`属性来访问累加器的值

   示例：

```
file=sc.textFile('xxx.txt')
acc=sc.accumulator(0)
def xxx(line):
  global acc #访问全局变量
  if yyy:
    acc+=1
  return zzz
rdd=file.map(xxx)
```

1.2 累加器与容错性

1. `spark`中同一个任务可能被运行多次：
   - 如果工作节点失败了，则`spark`会在另一个节点上重新运行该任务
   - 如果工作节点处理速度比别的节点慢很多，则`spark`也会抢占式的在另一个节点上启动一个投机性的任务副本
   - 甚至有时候`spark`需要重新运行任务来获取缓存中被移出内存的数据
2. 当`spark`同一个任务被运行多次时，任务中的累加器的处理规则：
   - 在行动操作中使用的累加器，`spark`确保每个任务对各累加器修改应用一次
     - 因此：如果想要一个无论在失败还是重新计算时，都绝对可靠的累加器，我们必须将它放在`foreach()`这样的行动操作中
   - 在转化操作中使用的累加器，无法保证只修改应用一次。
     - 转化操作中累加器可能发生不止一次更新
     - 在转化操作中，累加器通常只用于调试目的

二、广播变量

1. 广播变量可以让程序高效的向所有工作节点发送一个较大的只读值
2. `spark`会自动的把闭包中所有引用到的变量都发送到工作节点上。虽然这很方便，但是也很低效。原因有二：
   - 默认的任务发射机制是专门为小任务进行优化的
   - 事实上，你很可能在多个并行操作中使用同一个变量。但是`spark`会为每个操作分别发送。

2.1 Broadcast

1. `Broadcast`变量的`value`中存放着广播的值，该值只会被发送到各节点一次

2. `Broadcast` 的方法：
   - `.destroy()`：销毁当前 `Broadcast` 变量的所有数据和所有 `metadata`。
     - 注意：一旦一个 `Boradcast` 变量被销毁，那么它就再也不能被使用
     - 该方法将阻塞直到销毁完成
   - `.dump(value,f)`：保存 `Broadcast` 变量
   - `.load(path)`：加载 `Broadcast` 变量
   - `.unpersist(blocking=False)`：删除 `Broadcast` 变量在 `executor` 上的缓存备份。
     - 如果在此之后，该 `Broadcast` 被使用，则需要从驱动器程序重新发送 `Broadcast` 变量到 `executor`
     - 参数：
       - `blocking`：如果为 `True`，则阻塞直到 `unpersist` 完成
3. 属性：
   - `.value`：返回 `Broadcast` 变量的值
4. 使用 `Broadcast` 的流程：
   - 通过 `SparkContext.broadcast(xx)` 创建一个 `Broadcast` 变量
   - 通过 `.value` 属性访问该对象的值
   - 该变量只会被发送到各节点一次，应该作为只读值来处理（修改这个值并不会影响到其他的节点）

   示例：

```
bd=sc.broadcast(tuple('name','json'))
def xxx(row):
  s=bd.value[0]+row
  return s
rdd=rdd.map(xxx)
```

2.2 广播的优化

1. 当广播一个较大的值时，选择既快又好的序列化格式非常重要
   - 如果序列化对象的时间较长，或者传送花费的时间太久，则这个时间很容易成为性能瓶颈
2. `spark` 中的 `Java API` 和 `Scala API` 默认使用的序列化库为 `Java` 序列化库，它对于除了基本类型的数组以外的任何对象都比较低效。
   - 你可以使用 `spark.serializer` 属性来选择另一个序列化库来优化序列化过程