Toy Spark 文档

选题

我们组的选题是实现一个 mini 版本的 Spark。

我们希望尽可能多的实现原始 Spark 的功能，不在性能上面下过多的功夫。目前实现的功能已经超过了作业要求中要求实现的部分许多，而性能上和 Spark 尚有一定差距（大约 10 倍）。

我们希望保留 Spark 的如下特点：

- 用户只需编写串行程序
- 自动并行化和分布式执行
- 计算过程在内存中完成

实现功能

系统完成情况

详细的实现情况可以参阅最后一次展示的 PPT。这里列出比较重要的：

- Dataset 之间的依赖关系可以是一个有向无环图
- 支持在内存中缓存一个 dataset 的计算结果

实现的 API

- `generate`：产生数据，它的参数有两个，第一个参数是一个列表，例如 `List(4, 5, 5)` 表示你希望有三个节点，每个节点上分别有 4、5、5 个分片；第二个参数是一个 lambda 表达式，指定了生成数据的逻辑，这个 lambda 有两个参数，第一个参数是节点的编号，从 0 开始，第二个参数是这个节点上分片的编号，每个节点都是从 0 开始的。
- `read`：从 HDFS 读入数据。（TODO HTX）
- `map`、`filter`、`flatMap`、`distinct`：这几个的含义和使用比较显然。
- `repartition` 提供一个参数，是一个表示如何重新分片的列表，意义同 `generate` 中的那个参数。
- `groupByKey`：如果一个 dataset 的内容是键值对形式（`Dataset[(K, V)]`），会把它按照相同的 key 进行合并，得到 `Dataset[(K, List[V])]`。
- `reduceByKey`：如果一个 dataset 的内容是键值对形式，会进行 group 之后在内部进行 reduce。
- `unionWith`、`intersectionWith`：对两个 dataset 求并或交。
- `cartestianWith`：求两个 dataset 的笛卡尔积。
- `joinWith`：如果一个 dataset 的内容是键值对形式（`Dataset[(K, V)]`），另一个也是，并且键的类型相同（`Dataset[(K, R)]`），则按照 key 在组里进行笛卡尔积，得到 `Dataset[(K, (V, R))]`。
- `reduce`、`collect`、`count`、`take`：这些是一些 actions，含义和 Spark 类似。
- `saveAsSequenceFile`：TODO HTX
- `saveAsSingleFile`：TODO HTX
- `save`：对一个 Dataset 进行缓存。

实现的 API 可以用来做什么

Spark 能做的许多简单的工作都可以用 Toy-Spark 完成，例如迭代地计算 PageRank：

```
def randomSourceURL()      = Random.nextPrintableChar() + Random.nextInt(10)
def randomDestinationURL() = Random.nextPrintableChar() + Random.nextInt(10)

val iters = 10
val links = Dataset
  .generate(List(4, 4, 4), (_, _) => List.fill(1000)(randomSourceURL(),
randomDestinationURL()))
  .distinct()
  .groupByKey()
links.save()
var ranks = links.map({ case (k, _) => (k, 1.0) })

for (_ <- 1 to iters) {
  val contribs = links
    .joinWith(ranks)
    .flatMap({
      case (_, (urls: List[Any], rank: Double)) =>
        val size = urls.size
        urls.map(url => (url, rank / size))
    })
  ranks = contribs
    .reduceByKey((x, y) => x.asInstanceOf[Double] + y.asInstanceOf[Double])
    .map({ case (k, v: Double) => (k, 0.15 + 0.85 * v) })
}

val output = ranks.collect(Nil)
output.foreach(tup => println(s"${tup._1} has rank: ${tup._2}"))
```

为了简单起见，URL 都是随机生成的。在实际的使用中你可以从 HDFS 读入，或者在 lambda 中读入相关的文件。可以看到我们的 Toy-Spark 支持：

- Dataset 的缓存和复用
- Dataset 依赖关系是一个有向无换图
- Scala 的控制结构和 Toy-Spark 可以一起使用，例如上面的 `for`

而且使用体验和 Spark 基本一致。

使用说明

如果需要使用，在 `main` 函数中先使用 `Communication.initialize(args)`，然后写自己要执行的逻辑，然后使用 `Communication.close()` 回收资源。这一步可以仿照已有的 Main.scala 文件。

然后提供一个 config.json 文件。该文件的格式如下：

```
{
    "master": {
        "ip": "172.21.0.20",
        "port": "23333"
    },
    "workers": [
        {
            "ip": "172.21.0.4"
        },
        {
            "ip": "172.21.0.33"
        }
    ],
```

```
    "hdfs": {
        "coreSitePath": "/home/ubuntu/hadoop-2.7.7/etc/hadoop/core-site.xml",
        "url": "hdfs://master:9000"
    }
}
```

其中 `master` 填 master 的 IP 地址和监听端口；`workers` 填各个 worker 的 IP 地址；`hdfs` TODO HTX。

最后，部署时使用 `sbt assembly` 命令得到 fat jar。在各个节点上使用 `java -jar foo.jar <n>` 运行。其中把 `<n>` 换成一个数字，对于 master 来说，这个数字是 0；对于各个 worker 来说，这个数字是它在 config.json 的 `workers` 数组中的下标（从 1 开始）。可以参考 scripts 目录下的部署和运行脚本。

实现细节

名词约定

为了后续叙述的方便，这里先约定一些名词。由于 Toy Spark 的实现没有参考 Spark 的实现，所以二者的名词可能有所差异。此外，也会有一些 Spark 中没有的名词。

Transformation 和 action 的定义和 Spark 一致。

Master 节点与 Worker 节点

把节点（计算机）分成两类：***master 节点*** 和 ***worker 节点***。其中 master 节点只有一个，负责协调各个 worker 节点的工作；worker 节点可以有很多个。Worker 节点会负责实际的计算，而 master 节点除了协调外，也会负责计算，因此可以认为 master 节点是 worker 节点的超集。

Manager 线程、Communicator 线程与 Executor 线程

由于每个节点都负责计算，因此这里的三种线程在各个节点上都存在。

- **Manager 线程**：负责协调自己节点上的所有线程，以及负责处理来自各个 executor 线程的通信请求。
- **Communicator 线程**：负责完成实际的和 executor 线程的通信工作。
- **Executor 线程**：负责完成实际的计算工作，例如 `map` 操作。

其中，如果使用非阻塞的 IO 的话，communicator 线程应该是可以避免的。为了实现的简单起见，这里使用 communicator 线程专门负责和其它 executor 线程之间的网络通信。

Dataset、Partition 与 Record

- **Dataset** 的概念和 Spark 中的 RDD 是一致的。
- 一个 dataset 会被实现为多个 **partitions**，每个 partition 是由一个线程来负责的。一个节点上可能有多个 partition。
- 每个 partition 中有多个 record。

Direct Dependency 与 Shuffle Dependency

如果一个 dataset 依赖上游（即之前的）dataset 是像 `map` 这样的，考虑一个 partition 的依赖时不需要考虑其它的 partition 的话，那么这种 dependency 就被称作是 direct dependency。反之，称为 shuffle dependency。实质上的区别就在于下游的 partition 中的数据是否只需要上游的一个 parititon 就可以确定，如果是就是 direct 的，否则是 shuffle 的。

Job、Stage 与 Task

这三个概念比较抽象，因此使用一个例子加以说明。假设用户要估算 Pi 的值，使用的方法为先使用随机的方法产生一些 x 坐标，然后产生一些 y 坐标，随后使用笛卡尔积将它们合并成二元组，然后使用面积比来估算 Pi。

能够获取一个用户希望获得的计算结果的计算过程称为一个 ***job***。例如用户希望使用 Toy-Spark 来计算 Pi 的值，那么这件事情本身就是一个 job。一次计算可能包含多个 job，例如用户希望使用同样的随机数据计算其它的积分，那么每个积分都是一个独立的 job。**直观来讲，每个 action 操作都是一个 job。**

一个 ***stage*** 是连续的最长的无 shuffle dependency 计算过程。在例子中，我们在合并成二元组后，可能希望先对数据进行归一化把坐标变换到 [-1, 1] 中，这个过程可以看作是一个 `map` 操作；然后我们需要只保留在单位圆中的那些点，这个过程是一个 `filter` 操作。这些操作都不含 shuffle，因此同属一个 stage。此外一个 stage 还有如下的特点：它一点是多个计算的“链条”，中间不会出现分叉的情况。

一个 stage 的计算需要多个线程合作完成，每个线程负责完成的是 stage 中的一小部分。这一小部分被称作是一个 ***task***。线程在计算一个 task 时可以只关注这个 task 如何运行，不用关心其它 task 的流程，也不需要知道它们的存在。

大体运行流程

1. 用户提供计算逻辑代码与节点握手

这一步中用户需要提供一个程序。这个程序会在所有节点上运行，如果运行的节点是 master 节点的话，还会返回适合的结果。

在正式的计算之前，程序必须以 `Communication.initalize` 开始，这个函数中各个节点会进行握手以建立通信。

2. Job 划分与 Stage 划分

Job 的划分比较简单。对于计算逻辑中的每个 action，我们都可以得到一个 job。

而对于每个 job，可以对其进行 stage 的划分。由于一个 stage 需要是一个“链条”，而 job 的计算过程可能有分叉和汇合，因此需要进行一些处理。

> **什么是分叉与汇合**
>
> 分叉指的是一个 dataset 被多次利用的情况，例如：
>
> ```
> val dataset = foo.map(x => x)
> val bar1 = dataset.map(x => x * x)
> val bar2 = dataset.filter(x => {if (x % 2) {true} else {false}})
> ```
>
> `dataset` 这里产生了分叉。
>
> ---
>
> 汇合指的是多个 dataset 被使用来计算一个 dataset 的情况，典型的场景就是计算笛卡尔积（两个 dataset 被用于计算一个 dataset）。

首先我们需要 **在逻辑上** 去掉所有的分叉。既然分叉的实质是要重复利用某个 dataset，我们就假定每次要重复利用的时候，总是根据该 dataset 的上游历史来重新计算出这个 dataset 的实际数据，这样一来就不存在分叉的问题了。不过，这么做会导致性能低下。为了弥补这个问题，每个 dataset 可以通过执行 `save` 操作来把自己缓存到内存中，这样一来“重新计算”的时候实际上直接是从内存中读出该 dataset 的值，间接实现了“重复利用”的效果。

现在一个 job 只有汇合了，整个计算过程可以使用一棵树来表示。我们实现的三个汇合操作 `unionWith`、`intersectionWith` 和 `cartesianWith` 都是二元操作，并且有如下的特点：汇合完成后的 partition 数在和一个操作数的 partition 数相同时，它可以和该操作数构成 direct dependency，和另一个操作数构成 shuffle dependency。**我们规定所有的汇合操作都必须满足如上的形式，以简化设计。**这样一来，在汇合处的 stage 划分总是可以完成（direct dependency 的可以分进一个 stage；shuffle dependency 的是另一个 stage，而且这个 stage 的最后一个计算就是在汇合操作这里）。

对于非汇合处的 stage 划分就没有这么复杂了：如果是 direct dependency 就并入 stage，如果是 shuffle dependency 就开始一个新的 stage。

3. 计算各个 Stage

Stage 划分完成后，对于每个 job 需要确定这个 job 对应 stage 的计算顺序。计算顺序由如下规则递归确定，从最后的 action 操作开始：

- 如果这个操作前面的 shuffle dependency 不是汇合操作，则应该先完成前面的计算后，执行 shuffle dependency 再执行自己的 stage；
- 如果这个操作前面是一个汇合操作，则应该在计算汇合操作的 shuffle dependency 分支后再计算 direct dependency 的分支。

这么说可能有些抽象，下面使用另一种更加直观的表述。前面提到去掉了分叉的计算过程实际上是一棵树。计算过程可以看作是从树叶到树根的过程。这里的一个问题是，先从哪个树叶开始？回答是从树根开始往回走，遇到汇合操作就选择 shuffle dependency 的分支，直到走到树叶，这个树叶是第一个计算的树叶。从这个树叶开始计算直到它对应的 stage 结束，此时整个计算树上就少了一个分支。如此递归下去即可。

实际上，对于 job 也需要安排顺序，不过由于 job 在代码中有一个“天然的”顺序，这一步可以忽略。

每个 Stage 的运行流程

Stage 的特性

进行了 Stage 的划分以及排序后可以保证：

- **Stage 内部的所有 dependency 都是 direct dependency** 这一点是 stage 的定义，在构造 stage 的时候已经保证了
- **Stage 中如遇到需要 shuffle dependency 的情况，对应的数据已经准备好了** 这一点由 stage 的排序方式保证，注意汇合时我们先计算了 shuffle dependency 的分支
- **Stage 各个环节没有分支和汇合，是一个“链条”** 这一点也由 stage 的定义和构造保证

一个 Stage 的处理

在确认了上述 stage 特性后，可以确定一个 stage 是如何处理的。

在计算一个 stage 之前，我们先看一下其中有没有哪个 dataset 已经执行过 `save` 操作了。如果有的话，就不用计算它前面的那些 dataset 了，而是可以直接从这里开始。

首先一个 stage 要获得开始计算的数据。这里有两种情况，一种是这个 stage 没有上游的 stage，数据是自己产生的，那么该 stage 的第一个 dataset 应该是使用 `Dataset.generate` 或者 `Dataset.read` 得到的；另一种是这个 stage 需要使用一个 shuffle dependency 从上游读取数据，这个 stage 在计算开始时就要先进行网络通信来获取自己需要的数据了，这个情况的例子是 `repartition`。

获得了数据后，就可以开始 stage 的计算了。在 stage 的计算过程中，可能遇到一个节点是汇合节点的情况。如果遇到这种情况的话，就要进行网络通信以获取需要的数据了。此外，在计算每个 dataset 时，还需要查看它是否有 `save` 的标记，如果有的话，就要将其 `save` 起来以备他用。

在 stage 计算完成后，需要等待其它的 dataset 来读取这一数据，具体的处理在下一小节叙述。

通信与缓存机制实现

在这一节中，需要首先对计算图上的每个节点进行编号，这个号码需要在所有的节点上一致。将这一号码称作是 `datasetID`。

如何产生 `datasetID`？在进行 stage 分割的过程中，每遇到一个 dataset，都去一个全局的词典里面看看有没有这个 dataset 的 `datasetID`。如果有的话不进行任何处理，如果没有的话就给它分配一个 ID。这么做是因为在不同的机器上面，逻辑上同属一个 dataset 的 dataset 实例实际是不同的对象，但是我们希望它们具有某个统一的标示。完成这一标示后，在一个机器上总是可以通过 dataset 实例来获取它的 `datasetID`。

通信机制的实现

通信可能在 stage 计算开始时或中途触发。不论何时触发，总是可以在本地的机器上得到对应的上游 dataset 对象，进入获取其 `datasetID`。有了 `datasetID` 后即可跨机器读取数据。

我们以 `repartition` 为例来讲述如何进行通信，先描述以 `repartition` 结束的 stage 如何处理，再描述以 `repartition` 开始的 stage 如何处理。

以 `repartition` 结束的 stage 在结束时会将自己的数据存放到每个机器的全局的 context 中。可以理解成里面有一个从 `(partitionID, executorID)` 到实际数据的词典，是一个“发送缓冲区”。当 executor 结束 stage 时，它会在这个词典中添加自己的 entry，然后通知 manager 自己已经完成。

以 `repartition` 开始的 stage（即某个 `repartition` 的下游 stage），第一步是读取数据。它会提供如下数据以定位自己到底需要哪些数据：

- **读取方式描述**。这个 stage 想要上游 partition 中的一部分数据，还是全部数据？在这个场景下我们只需要一部分的数据（对于类似 `intersection` 这样的操作就需要用到全部的数据）。
- **自己的 `nodeID` 和 `partitionID`**。`nodeID` 指的是节点的编号，`partitionID` 是这个节点上的 partition 的编号。提供这两个值是为了获取自己需要的那一部分数据。
- **一个随机种子**。在读取上游 partition 中只属于自己的那一部分数据时，需要上游 partition 在提供数据时引入一定的随机性（`repartition` 伴随着 random shuffle）。但是，按照设计，上游 partition 无法获取到下游的任何数据，它不知道下游 stage 有多少个 partition 这样的信息，因此这种信息需要下游提供。只要保证下游 stage 中这些 partition 产生的随机种子是一样的就可以了。

上游 partition 获取到这些数据后可以确定具体要放回哪一部分的数据。对于 `repartition` 而言，实现是将产生一些随机的索引，将 partition 对应索引上的值返回。保证所有的下游 partition 产生的索引是所有可能索引的一个划分即可（彼此之间不相交，且并集是全集）。通信的监听是由 manager 线程完成的，收到请求后它会开启一个 communicator 来专门负责此次通信。

值得注意的是，每个 partition 应该只被发送一次就可以从“发送缓冲区”中删除了，因为逻辑上讲在消除了分叉后每个 partition 只会被使用一次。不过也可以通过调用 `save` 方法来实现一次计算多次使用，但在这里而言，我们设计成把通信和缓存分离开来，即使通信机制不需要知道缓存机制的存在，也可以达到一次计算多次使用的目的。

通行时不是所有的数据都需要传输。为了表示这种传输时数据的选择策略，我们引入了一个叫做 `SamplingType` 的东西。它指明了如何挑选要传输的数据，例如 `FullSampling` 表示的是传输所有数据（在 `collect` 时使用）；`HashSampling` 表示根据元素的 hash 值决定它是否应该被发送往某个下游 dataset（在 `intersectionWith` 的时候被用到）。我们一共实现了六种选择策略。

缓存机制的实现

所谓的缓存机制指的就是调用 `save` 方法后发生的事情。在代码中调用 `save` 方法会给 dateset 打上相应的标记。计算时：

- 如果是第一次要使用这个 dataset，就把值算出来，然后存到本地全局 context 的一个映射中（同样是 `(datasetID, partitionID)` 到实际数据的映射）。
- 如果不是第一次使用这个 dataset，则直接从前述 context 中读取。

显然，由于被 `save` 的 dataset 极有可能被多次调用，即使它被使用了也不应该移出缓存。

目前的缓存机制是使用内存实现的，但是它也可以很容易地使用文件的方式来实现。

测试结果与分析

TODO HTX，注意要分析

分工情况

张洋：

- 设计和实现 Toy-Spark 的大体框架
- 四次 PPT 制作和展示
- 撰写文档

郝天翔：

- HDFS 的读写支持
- 一些 transformation 和 action 的实现
- 正确性测试与性能测试