

大数据高端人才专项计划



東北大學
Northeastern University



HORIZON
昊宸科技



- 1 介绍Spark 核心
- 2 Spark 编程基础
- 3 广播变量和累加器
- 4 实例应用



The image features a clear blue sky as the background. Several wind turbines are visible, with white towers and blades that have orange and white striped tips. In the center, the text "Spark Core" is written in a white, sans-serif font. This text is overlaid on a series of overlapping, semi-transparent circles in shades of light blue, teal, and lavender. The overall composition is clean and modern, suggesting a focus on clean energy or technology.

Spark Core

Spark 所需Hadoop 的知识点



- HDFS基本架构
- HDFS核心概念: Block , 文件, 目录
- HDFS 基本操作, 读写目录/文件
- YARN 基本架构

ResourceManager, NodeManager

ApplicationMaster

Container

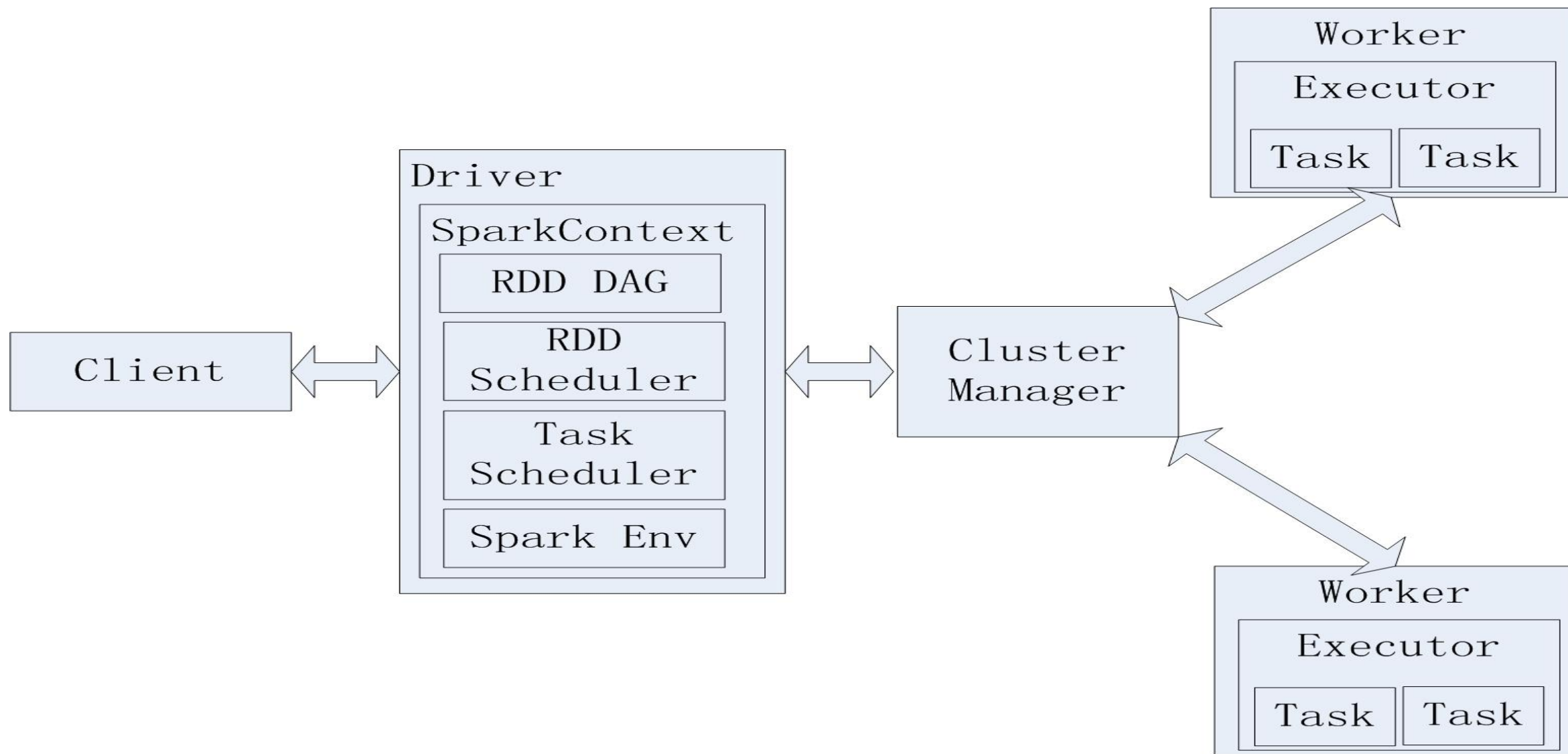
➤ RDD Resilient Distributed Datasets 弹性分布式数据集

分布在集群中的只读对象集合 (由多个 **Partition** 构成)

可以存储在磁盘或内存中 (多种存储级别)

通过并行“转换”操作构造

失效后自动重构



- **应用程序(Application):** 基于Spark的用户应用程序，包含了一个Driver Program 和集群中多个的Executor。
- **Cluster Manager:** 在Standalone 模式中即为Master(主节点)，控制整个集群，监控Worker。在yarn中类似于Resource Manager。
- **Worker:** 从节点，负责计算的节点，启动Executor 或 Driver。在yarn中相当于NodeManager，负责计算节点的控制。
- **执行单元(Executor):** 是为某Application运行在Node上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上，每个Application都有各自独立的Executors。
- **操作(Operation):** 作用于RDD的各种操作分为Transformation和Action。

- **驱动程序(Driver Program):** 运行 Application 的 main() 函数并且创建 SparkContext，通常用 SparkContext 代表 Driver Program。
- **Job:** 由 spark 的 Action 算子触发，有多少个 action 算子就有多少个 Job。
- **Stage:** 每个 Job 都会根据 RDD 的宽窄依赖关系被切分为多个 Stage。
- **Task:** 一个分区对应一个 Task，Task 执行 RDD 中对应 Stage 中包含的算子。
- **有向无环图(DAG)：** Directed Acycle Graph，反应 RDD 之间的依赖关系。
- **有向无环图调度器(DAG Scheduler)：** 根据 Job 构建基于 Stage 的 DAG，并提交 Stage 给 TaskScheduler。

- local (本地运行)

单机运行，通常用于测试。

- Standalone(独立模式)

独立运行在一个集群中。

- YARN/mesos

运行在资源管理系统上，比如YARN或mesos

Spark On YARN 存在两种模式

yarn – client

yarn – cluster

➤ Spark On YARN 提交命令举例

```
./spark-submit --class com.esoft.als.RunRecommender \  
--master yarn-client \  
--driver-memory 10g \  
--executor-memory 10g \  
--executor-cores 20 \  
--num-executors 6 \  
hdfs://192.168.88.104:8020/data/ALS.jar \  
hdfs://192.168.88.104:8020/data/
```

参考

<http://spark.apache.org/docs/1.6.3/running-on-yarn.html>

<http://spark.apache.org/docs/2.2.0/submitting-applications.html>

Spark 运行模式-本地模式



➤ 什么是本地模式

将Spark应用以多线程方式，直接运行在本地，便于调试。

➤ 本地模式，Driver 和 Executor 在一起

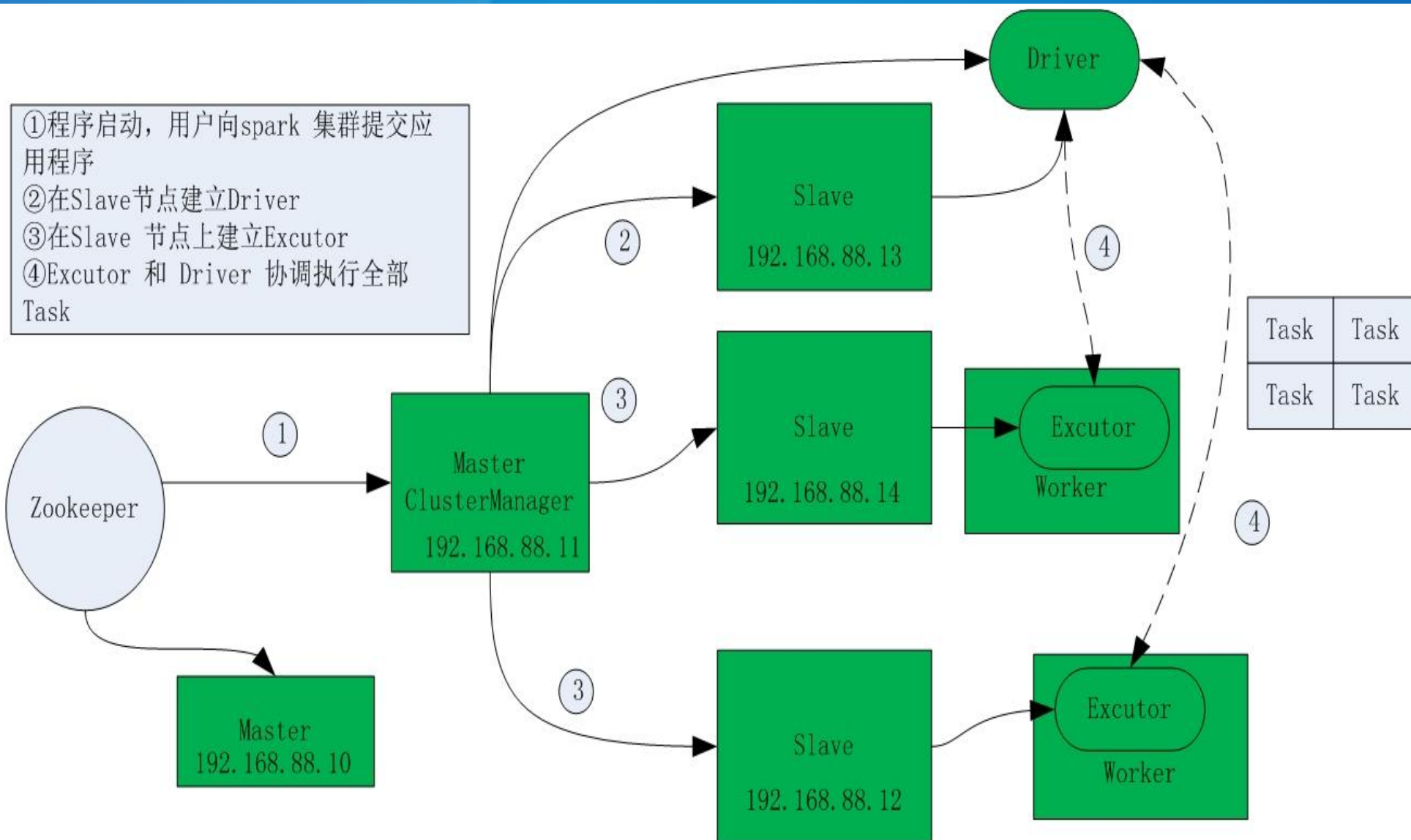
local: 分一个核心

local[k]: 给Driver 和 executor 分几个核心

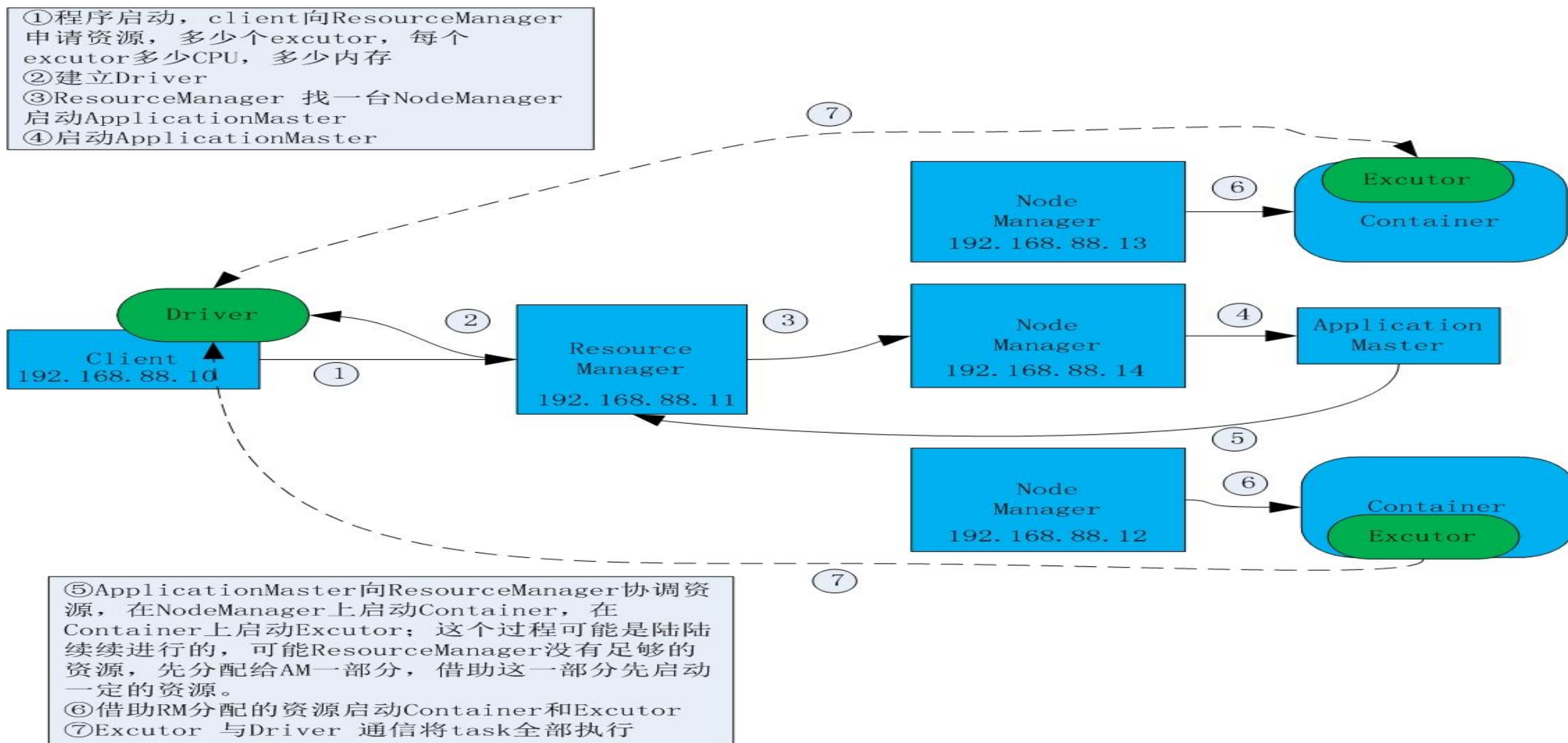
local[*]:给Driver 和 executor 分 cpu 线程数目相同的核心



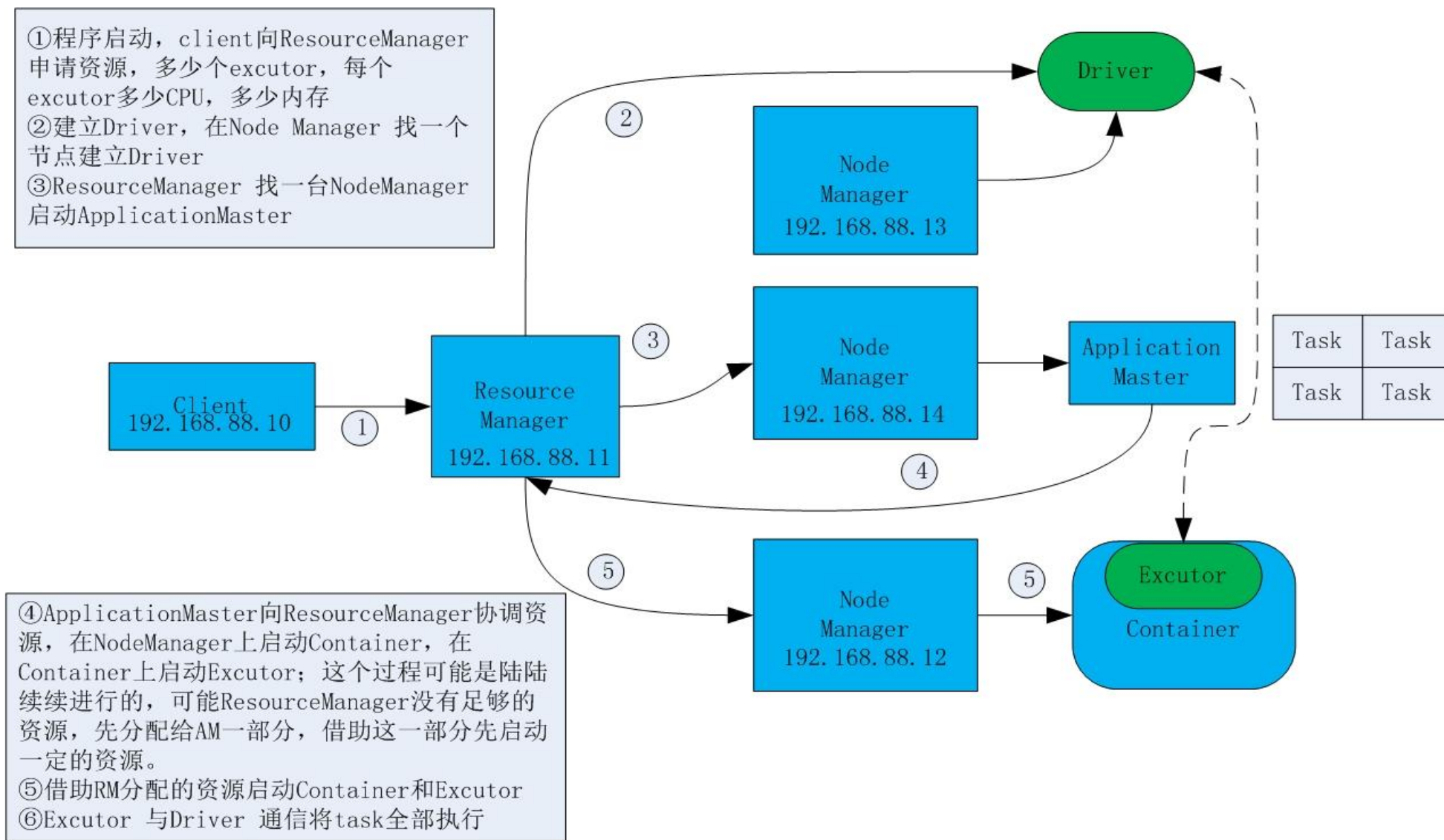
Spark 任务提交流程-Standalone



Spark 任务提交流程-yarn-client



Spark 任务提交流程-yarn-cluster



Spark 编程基础



➤ 创建 SparkContext 对象

封装 spark 执行环境信息（每一个Spark 程序有且仅有一个 SparkContext）

```
17/06/01 15:05:27 INFO BlockManagerMaster: Registered BlockManager
Exception in thread "main" org.apache.spark.SparkException: Only one SparkContext may be running in this JVM (see SPARK
org.apache.spark.SparkContext.<init>(SparkContext.scala:82)
com.horizon.sparkcore.FirstSparkDemo$.main(FirstSparkDemo.scala:12)
com.horizon.sparkcore.FirstSparkDemo.main(FirstSparkDemo.scala)
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
java.lang.reflect.Method.invoke(Method.java:498)
```

➤ 创建 RDD

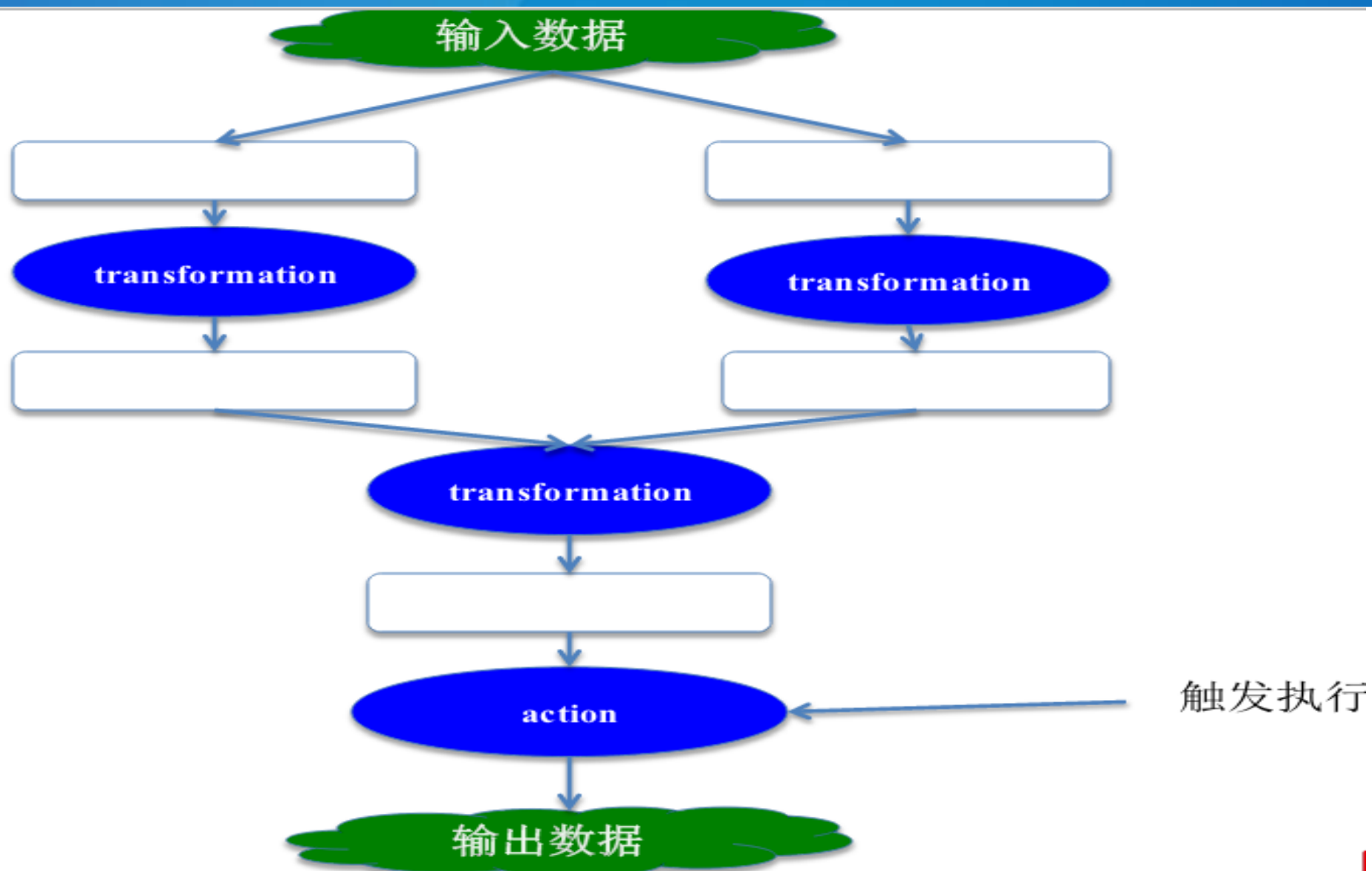
从 Scala 集合 或 Hadoop 数据集上创建

➤ 在 RDD 之上进行转换和 action

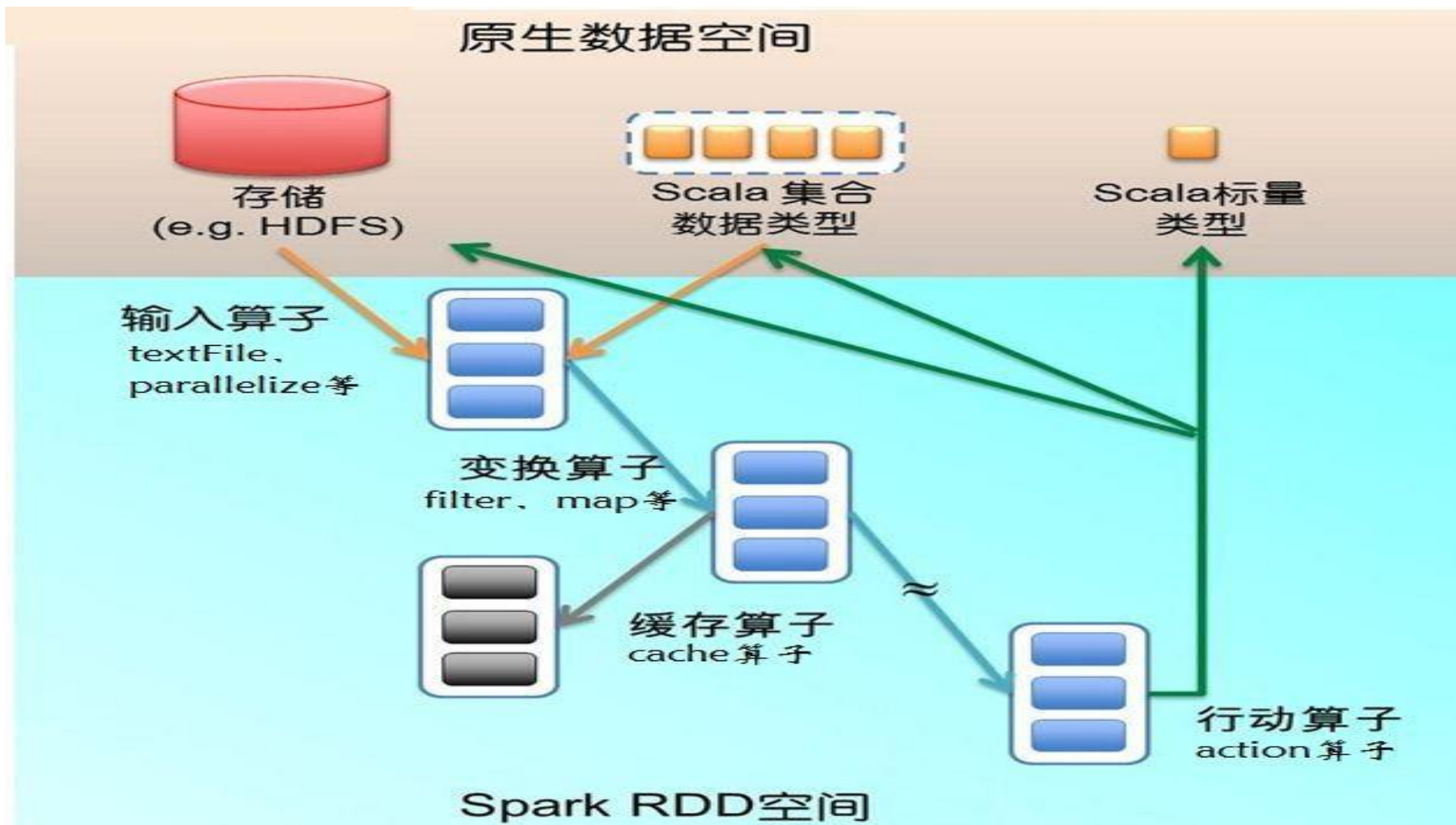
Spark 提供了多种转换和 action 函数

➤ 返回结果

保存到 HDFS 中，或直接打印出来



RDD 惰性执行



Spark 程序设计— 创建 SparkContext



- 创建 conf 和 SparkContext , 封装了Spark 配置信息 (Spark 1.x)

```
val conf = new SparkConf().setAppName(appName)  
val sc = new SparkContext(conf)
```

- 创建 SparkSession 封装了调度器等信息 (Spark 2.x)

```
val spark = SparkSession.builder().appName(appname).master('local[4]').getOrCreate()
```

- 1. `sc.parallelize(List(1, 2, 3), 2)` 对List(1,2,3) 进行并行化 , 并行度为2 (2个task 在跑)
- 2.启动 10个 map task 进行处理
 - ① `val slices = 10`
 - ② `val n = 100000 * slices` 每个map 迭代10w 次
 - ③ `val count = sc.parallelize(1 to n, slices).map { i => // 10个partition 平分这些数据`
 - ④ `val x = random * 2 - 1`
 - ⑤ `val y = random * 2 - 1`
 - ⑥ `f (x*x + y*y < 1) 1 else 0`
 - ⑦ `}.reduce(_ + _)`

➤ 1. 文本文件 TextInputFormat

`sc.textFile("file.txt")` // 将本地文本文件加载成 **RDD**

`sc.textFile("directory/*.txt")` // 将某类文本文件加载成 **RDD**

`sc.textFile("hdfs://nn:9000/path/file")` // hdfs 文件或目录

➤ 2.sequenceFile文件 SequenceFileInputFormat

`sc.sequenceFile("file.txt")` //将本地二进制文件加载成**RDD**

`sc.sequenceFile[String, Int] ("hdfs://nn:9000/path/file")`

➤ 3.使用任意自定义的Hadoop InputFormat

`sc.hadoopFile(path, inputFmt, keyClass, valClass)`

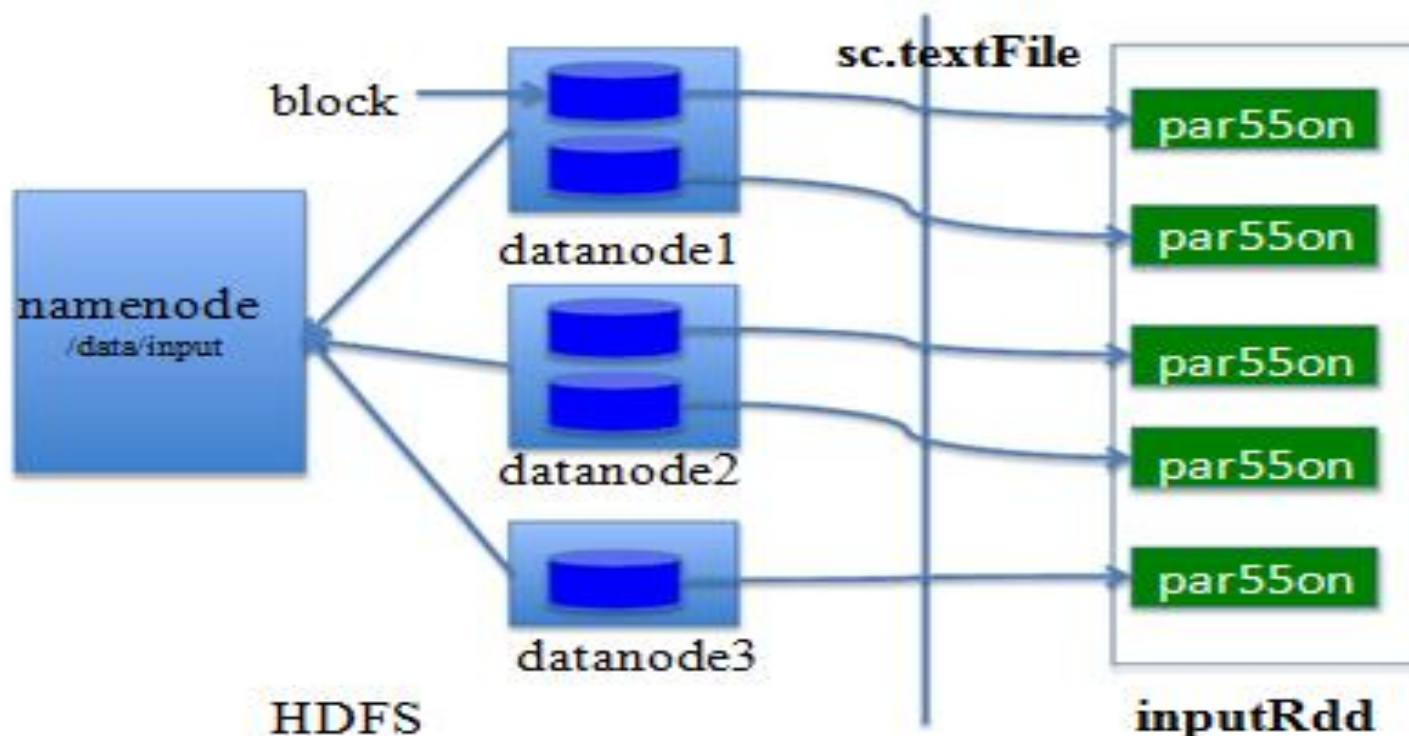
Spark 程序设计—创建RDD:HDFS



`inputRdd = sc.textFile("/data/input")` //配置了hadoop 客户端（配置文件）之后可以通过HDFS 访问文件

`inputRdd = sc.textFile("file:///data/input")` //读取本地文件系统的文件

`inputRdd = sc.textFile("hdfs://namenode:8020/data/input")` //读取HDFS 上的文件



Transformations	$map(f : T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]$ $filter(f : T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]$ $flatMap(f : T \Rightarrow Seq[U]) : RDD[T] \Rightarrow RDD[U]$ $sample(fraction : Float) : RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling) $groupByKey() : RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$ $reduceByKey(f : (V, V) \Rightarrow V) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $union() : (RDD[T], RDD[T]) \Rightarrow RDD[T]$ $join() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$ $cogroup() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$ $crossProduct() : (RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$ $mapValues(f : V \Rightarrow W) : RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning) $sort(c : Comparator[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $partitionBy(p : Partitioner[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Actions	$count() : RDD[T] \Rightarrow Long$ $collect() : RDD[T] \Rightarrow Seq[T]$ $reduce(f : (T, T) \Rightarrow T) : RDD[T] \Rightarrow T$ $lookup(k : K) : RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs) $save(path : String) : \text{Outputs RDD to a storage system, e.g., HDFS}$

- Transformation : 将一个RDD通过一种规则，映射成另一种RDD
- Action : 返回结果或者保存结果，只有action才会触发程序的执行

- 1.创建RDD

```
val listRdd = sc.parallelize(List(1, 2, 3), 3)
```

- 2.将 RDD 传入函数,生成新的 RDD

```
val squares = listRdd.map(x => x*x) // {1, 4, 9}
```

- 3.对RDD中元素进行过滤,生成新的RDD

```
val even = squares.filter(_ % 2 == 0) // {4}
```

- 4.将一个元素映射成多个,生成新的RDD

```
nums.flatMap(x => 1 to x) // => {1, 1, 2, 1, 2, 3}
```


➤ 1.创建RDD

```
val nums = sc.parallelize(List(1, 2, 3), 2)
```

➤ 2.将RDD保存为本地集合 (返回到driver端)

```
nums.collect() // => Array(1, 2, 3)
```

➤ 3.返回前K个元素

```
nums .take(2) // => Array(1, 2)
```

➤ 4.计算元素总数

```
nums.count() // => 3
```

➤ 5.合并集合元素

```
nums .reduce(_ + _) // => 6
```

➤ 6.将RDD写到HDFS中

```
nums.saveAsTextFile("hdfs://nn:8020/output")
```

```
val pets: RDD[(String, Int)] = sc.parallelize(List(("cat", 1), ("dog", 1), ("cat", 2)))
```

```
pets.reduceByKey(_ + _) // => {(cat, 3), (dog, 1)}
```

```
pets.groupByKey() // => {(cat, Seq(1, 2)), (dog, Seq(1))}
```

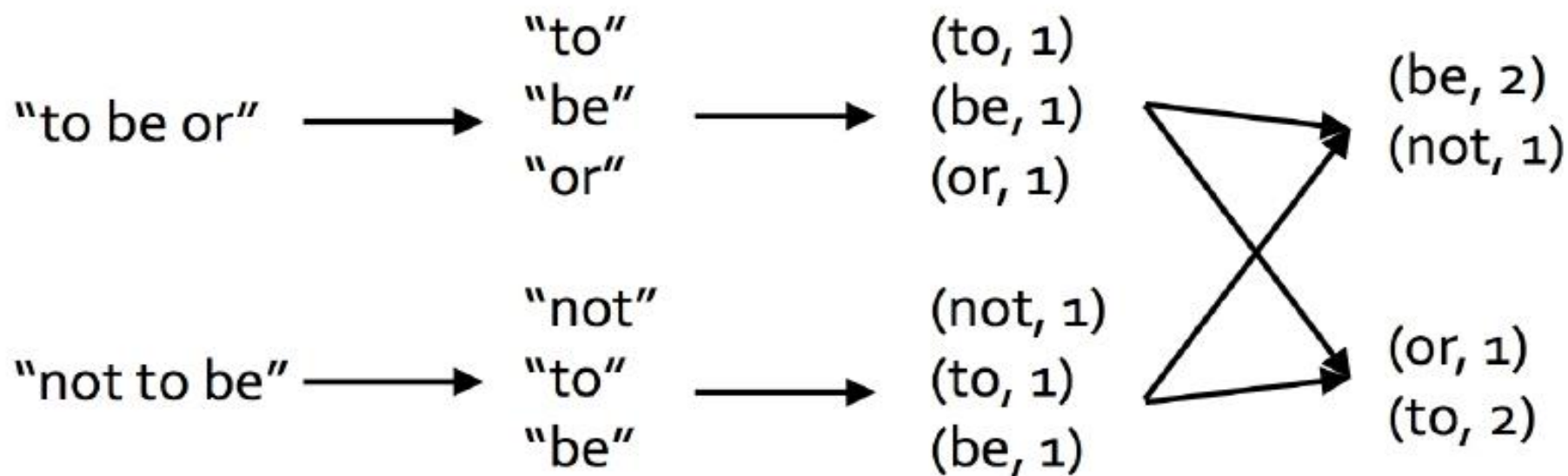
```
pets.sortByKey() // => {(cat, 1), (cat, 2), (dog, 1)}
```

World Count



```
val lines = sc.textFile("hamlet.txt")
```

```
val counts = lines.flatMap(line => line.split(" "))  
                    .map(word => (word, 1))  
                    .reduceByKey(_ + _)
```



应用程序举例 Join



```
val visits = sc.parallelize(List(  
  ("index.html", "1.2.3.4"),  
  ("about.html", "3.4.5.6"),  
  ("index.html", "1.3.3.1")))  
  
val pageNames = sc.parallelize(List(  
  ("index.html", "Home"), ("about.html", "About")))  
  
visits.join(pageNames)  
// ("index.html", ("1.2.3.4", "Home"))  
// ("index.html", ("1.3.3.1", "Home"))  
// ("about.html", ("3.4.5.6", "About"))  
  
visits.cogroup(pageNames)  
// ("index.html", (Seq("1.2.3.4", "1.3.3.1"), Seq("Home")))  
// ("about.html", (Seq("3.4.5.6"), Seq("About")))
```

- 1.所有key/value RDD 操作符均包含一个可选参数,表示reduce task 并行度

words.reduceByKey(_ + _, 5)

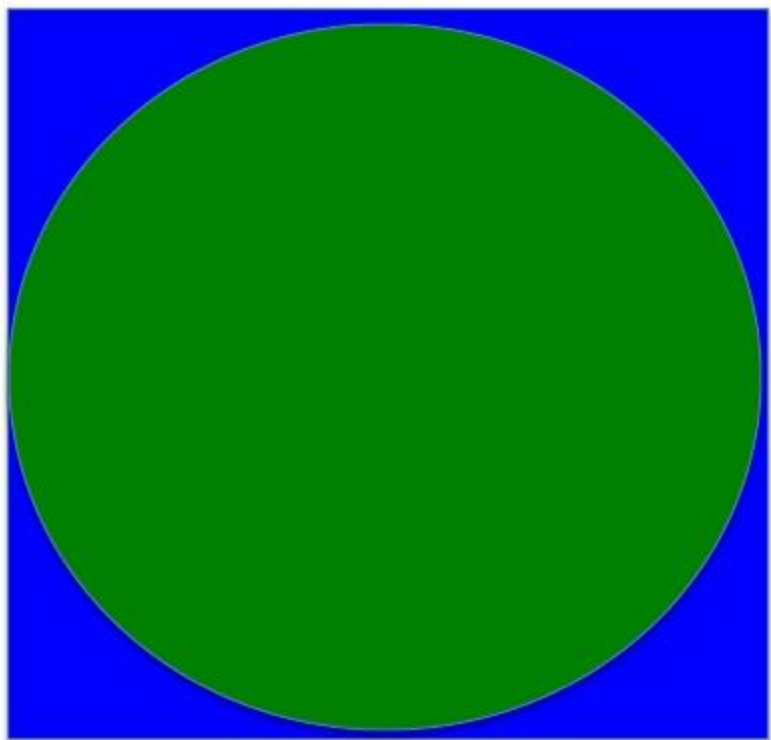
words.groupByKey(5)

visits.join(pageViews, 5)

- 2.用户也可以通过修改spark.default.parallelism设置默认并行度

默认并行度为 最初的 **RDD partition** 数目

Spark 内存计算实例，计算PI近似值



1. 假设正方形边长为 x , 则：正方形面积为： $x*x$
2. 圆的面积为： $\pi*(x/2)*(x/2)$ 两者之比为： $4/\pi$
面积比 = $4/\pi$ $\pi = 4/\text{正方形面积}/\text{圆面积}$
3. 随机产生位于正方形内的点 x 个，假设位于圆中的有 y 个
则：
 $\pi = 4*y/x$ 当 $x \rightarrow \infty$ 时， π 逼近真实值

Spark 累加器和广播变量



Spark 程序设计——Accumulator



- **Accumulator(累加器,计数器)** 利用上面pi 的例子
类似于 **MapReduce** 中的 **counter**,将数据从一个节点发送到其他各个节点上去;
通常用于监控,调试,记录符合某类特征的数据数目等
- **Accumulator 使用**

```
val total_counter = sc.accumulator(0L, "total_counter")  
val counter0 = sc.accumulator(0L, "counter0")  
val counter1 = sc.accumulator(0L, "counter1")
```

定义累加器

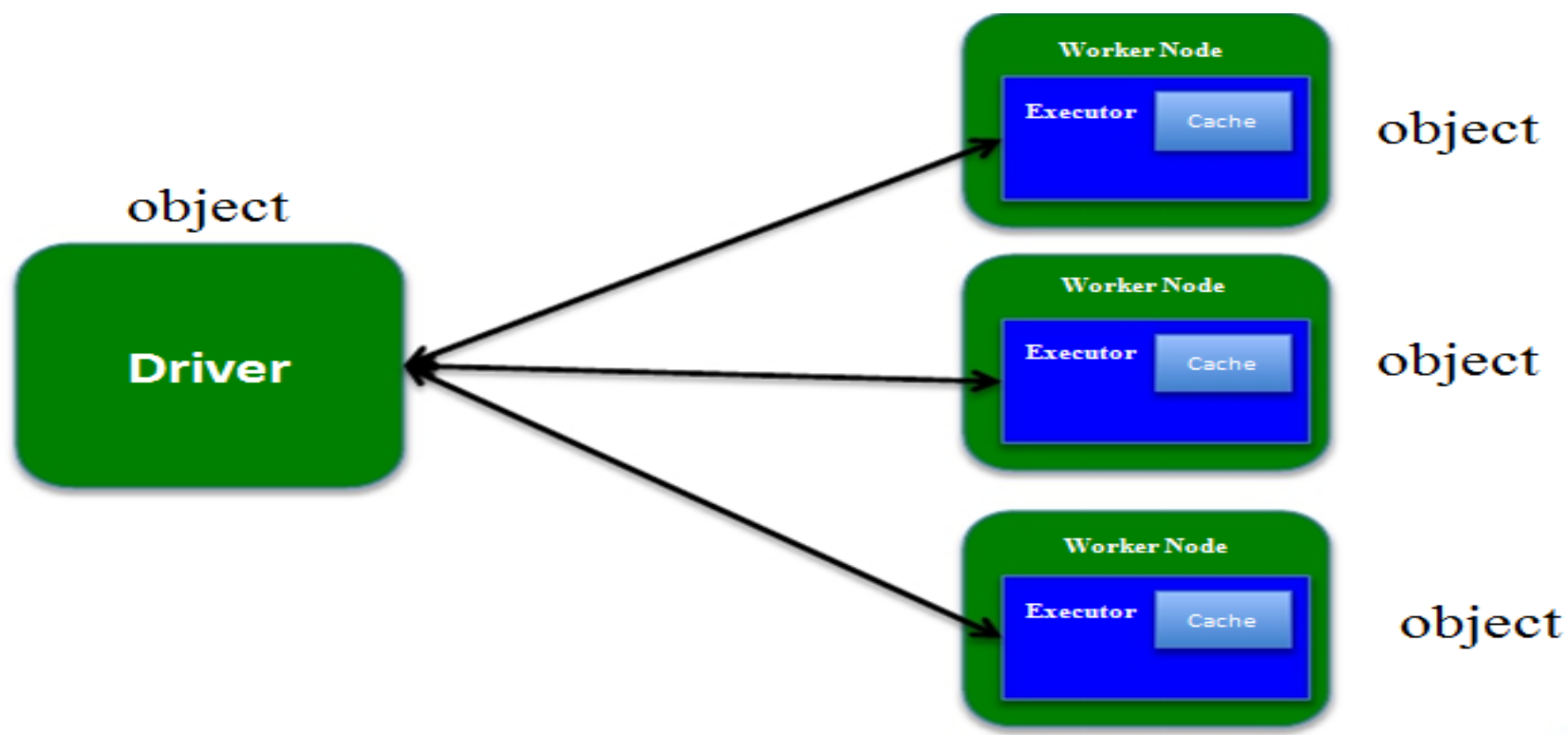
```
val count = sc.parallelize(1 to n, slices).map { i =>  
  total_counter += 1  
  val x = random * 2 - 1  
  val y = random * 2 - 1  
  if (x*x + y*y < 1) {  
    counter1 += 1  
  } else {  
    counter0 += 1  
  }  
  if (x*x + y*y < 1) 1 else 0  
}.reduce(_ + _)
```

累加器 + 1

➤ 广播机制

高效分发大对象，比如字典（**map**），集合（**set**）等，每个**executor**一份，而不是每个**task**一份；包括**HttpBroadcast**和**TorrentBroadcast**两种

➤ HttpBroadcast 与 TorrentBroadcast （P2P 的机制，节点非常多时，互相通信）



- broadcast 必须是只读变量

由于要广播到多台机器上，那么如果一个节点被更新，其他节点没法同步更新，因此Spark目前仅支持只读变量。

- broadcast 到Executor而不是到每个task

因为每个 task 是一个线程，而且同在一个进程运行 tasks都属于同一个 application .因此每个节点executor 上放一份就可以被所有 task 共享。

- HttpBroadcast 就是每个 executor 通过的 http 协议连接 driver 并从 driver 那里 fetch data（主流使用TorrentBroadcat）

- TorrentBroadcast，这类似于大家常用的BitTorrent技术。基本思想就是将data分块成 datablocks,然后executor fetch 到了一些 datablocks，那么这个 executor 就可以被当作 data server 了，随着 fetch 的 executor 越来越多，有更多的 data server 加入，data 就很快能传播到全部的 executor 那里去了。

➤ 广播使用

```
val data = Set(1, 2, 4, 6, ..... ) // 大小为128MB  
val rdd = sc.parallelize(1to 6, 2)  
val observedSizes= rdd.map(_ => data.size)
```

效率如何呢？ ？ ？

区别是什么？

```
val data = Set(1, 2, 4, 6, ..... ) // 大小为128MB  
val bdata = sc.broadcast(data)  
val rdd = sc.parallelize(1to 1000000, 100)  
val observedSizes= rdd.map(_ =>  
  bdata.value.size ....)
```

将大小为128MB
的Set广播出去

Spark 程序设计——cache 的基本概念与使用



➤ Spark RDD Cache

允许将RDD缓存到内存中或磁盘上，以便于重用

Spark提供了多种缓存级别，以便于用户根据实际需求进行调整

```
➤ val NONE = new StorageLevel(false, false, false, false)
val DISK_ONLY = new StorageLevel(true, false, false, false)
val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
val MEMORY_ONLY = new StorageLevel(false, true, false, true)
val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
val OFF_HEAP = new StorageLevel(false, false, true, false)
```

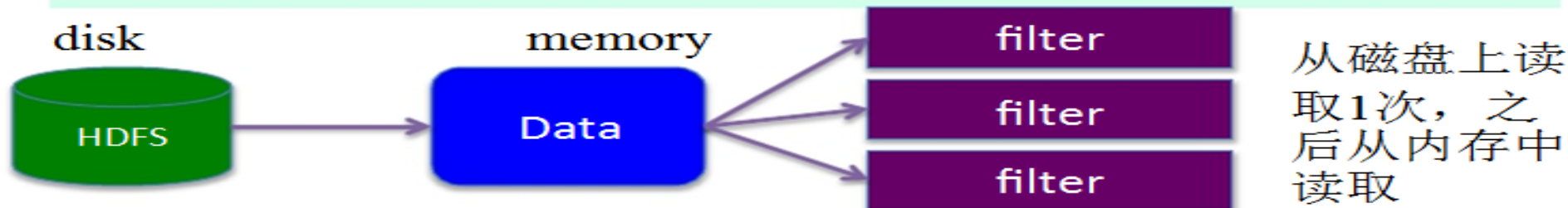
➤ RDD cache使用

```
val data = sc.textFile("hdfs://nn:8020/input")
data.cache()
//data.persist(StorageLevel.DISK_ONLY_2)
```

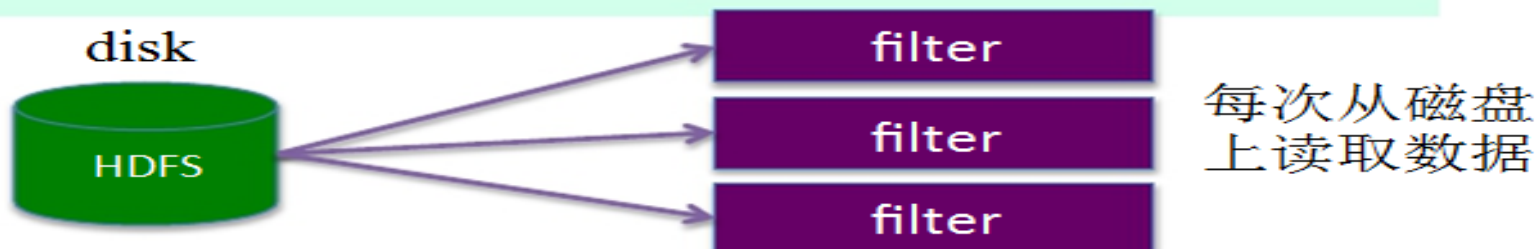
Spark 程序设计——深入了解cache



```
val data = sc.textFile("hdfs://nn:8020/input")
data.cache()
data.filter(_.startsWith("error")).count
data.filter(_.endsWith("hadoop")).count
data.filter(_.endsWith("hbase")).count
.....
```

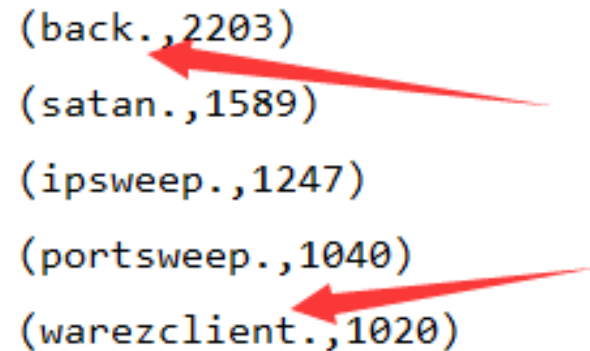


```
val data = sc.textFile("hdfs://nn:8020/input")
data.filter(_.startsWith("error")).count
data.filter(_.endsWith("hadoop")).count
data.filter(_.endsWith("hbase")).count
.....
```



- 1. 数据集net.gz为网络流量数据，数据集每条记录展现每个连接的信息，最后一列为攻击的标签
- (1) 请统计，出现的攻击类型对应的攻击次数
- (2) 为了输入给算法，请将RDD 类型转换为RDD[Labelpoint] (Labelpoint为Spark Vector)

```
(back.,2203)  
(satan.,1589)  
(ipsweep.,1247)  
(portsweep.,1040)  
(warezclient.,1020)
```



- 2. 数据集art.txt中存储id 和 姓名（用制表符/t 分割），art_alias.txt 存储正确ID 和错误ID 的对应关系，编写程序，将art.txt 中id 到art_alias.txt中匹配出正确的ID,并组成新的RDD[(id,name)]。
- 提示：在处理art.txt 数据集过程中会遇到数据不合规的问题，合理使用Some 和 None 解决此问题。



联系我们

如何联系我们...



联系
我们

地址：沈阳市和平区三好街84-8号易购大厦319A

电话：024-88507865

邮箱：horizon@syhc.com.cn

公司网站：<http://www.syhc.com.cn>



THANKS

