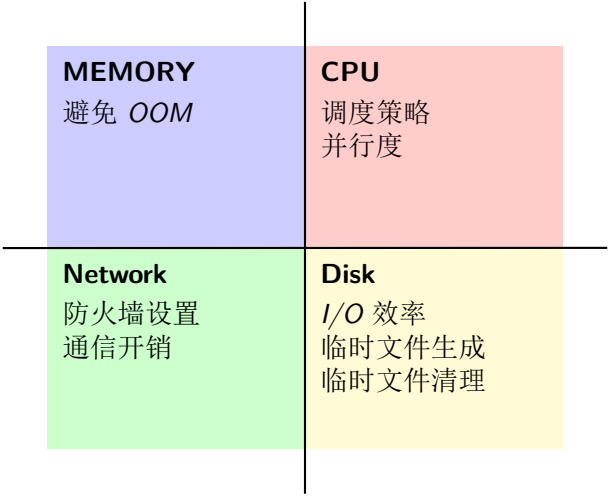
Spark 部署中的关键问题解决之道

许鹏

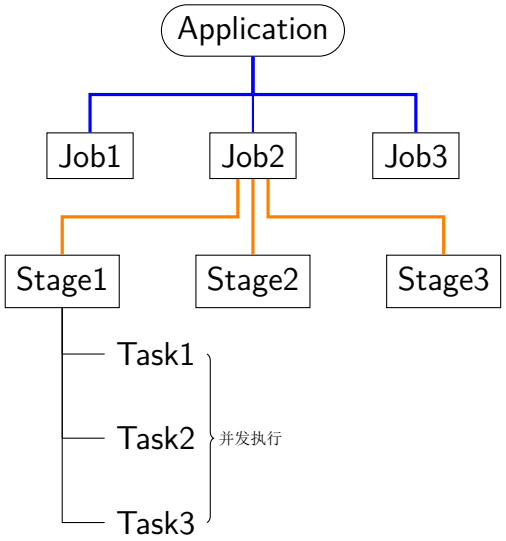
March 25, 2015

Spark 资源管理概述



RDD

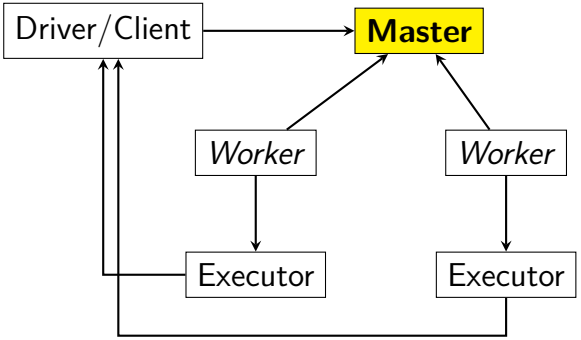
Application 规划阶段



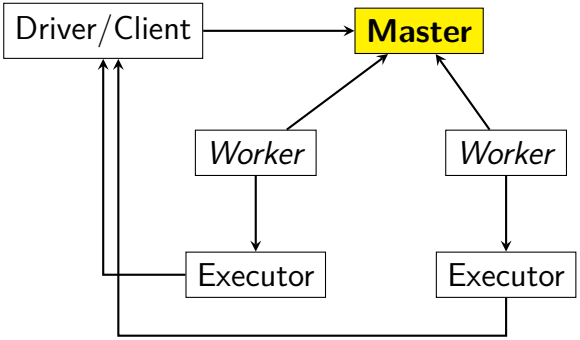
task 数目

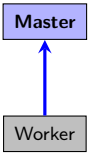
每个 Stage 中 Task 数目取决于读入数据的 Partition 数
而真正可以并行处理的 Task 数目则取决于可用的计算资源，即 CPU Core 数目

Standalone 集群组成

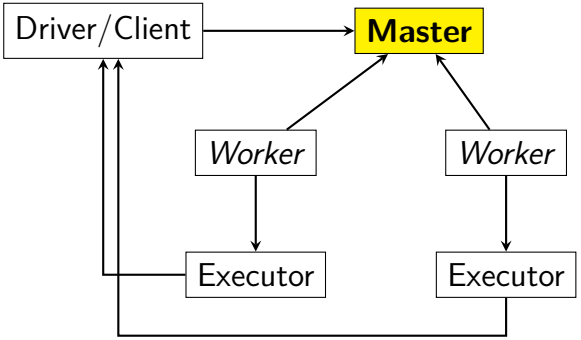


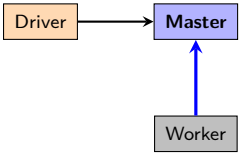
Standalone 集群组成



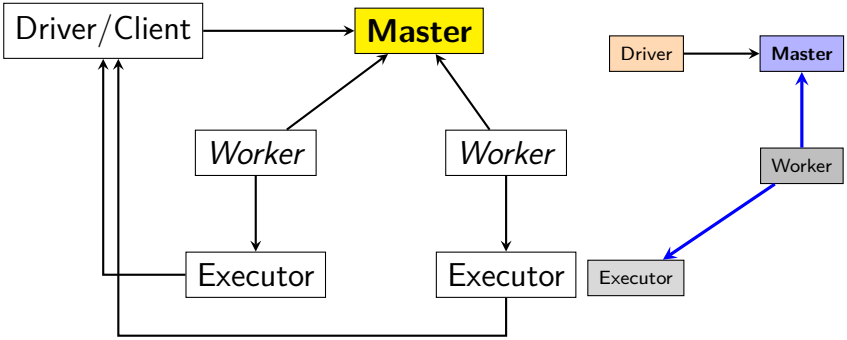


Standalone 集群组成

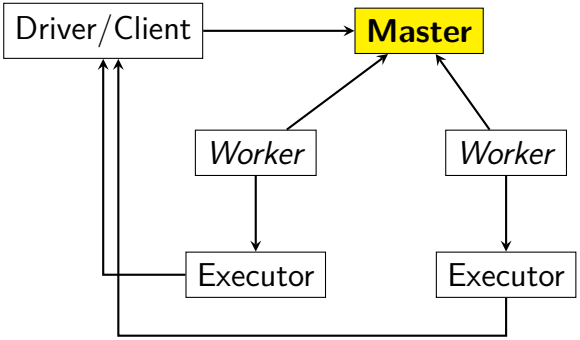


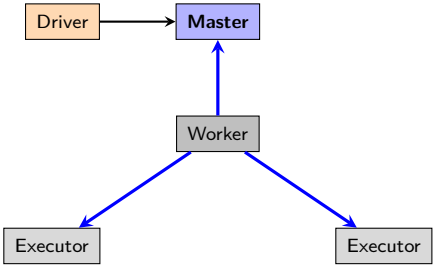


Standalone 集群组成

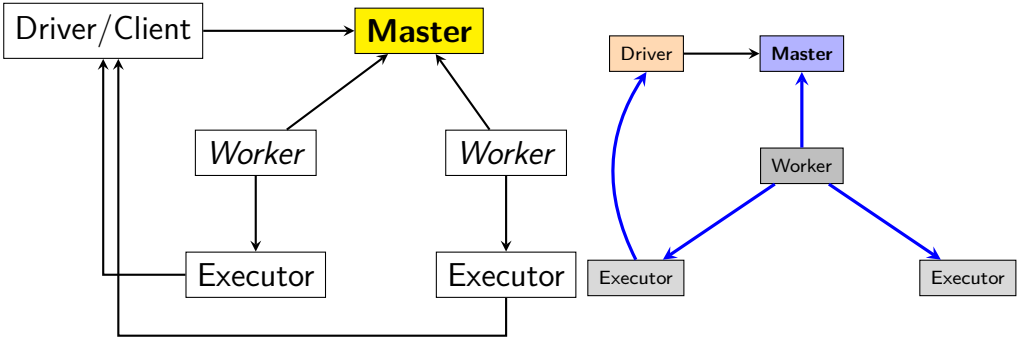


Standalone 集群组成





Standalone 集群组成

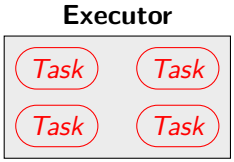


CPU

Task 到系统线程的映射关系

任务到线程的映射

1. Task 运行于 Executor 所在的 JVM 进程,Task 运行在具体的线程当中
2. 在同一个 Executor 中并行执行的 Task 个数取决于 Worker 上报给 Master 时声明的 Core 数目



在同一台机器中启动多个 Worker

spark-env.sh

假设要在一台有 24 核的机器上启动 4 个 worker

```
export SPARK_WORKER_INSTANCES=4
export SPARK_WORKER_CORES=6
```

两者相乘的结果不要超过 CPU 的物理核数 WORKER_INSTANCES $*$ WORKER_CORES $<$ PHYSICAL_CORES

无密码登录

```
cd $HOME
ssh-keygen -t dsa
cd .ssh
cat id_dsa.pub >> authorized_keys
$SPARK_HOME/sbin/start_slaves.sh
```

调度策略

通过 *spark.scheduler.mode* 来设置调度策略
通过多线程，同一个应用 (application) 可以提交多个作业 (job)，多个作业的调度策略有如下选择

- FIFO 先入先出
- FAIR 公平调度

避免单个应用占尽所有 CPU

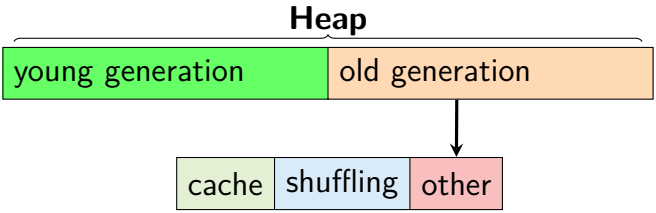
在 **spark-env.sh** 中设置 SPARK_MASTER_OPTS 为应用设定能获得的 CPU 个数的默认值

```
export SPARK_MASTER_OPTS="-Dspark.deploy.defaultCores=12"
```

应用程序可以在 **spark_defaults.conf** 中设置 *spark.cores.max* 来修改所要获取的最大核数

Memory

内存布局



NewRatio 的默认值是 2, 意味着 Old Generation 可以占用 2/3 的 HeapSize

shuffle	spark.shuffle.memoryFraction	0.2
cache	spark.storage.memoryFraction	0.6

表中两者相加的总和不能超过 *Old Generation* 所能占用的最大比例

Serialized vs. Deserialized

为了减少内存使用的压力，建议在 Cache 的时候将 Java Object 序列化存储

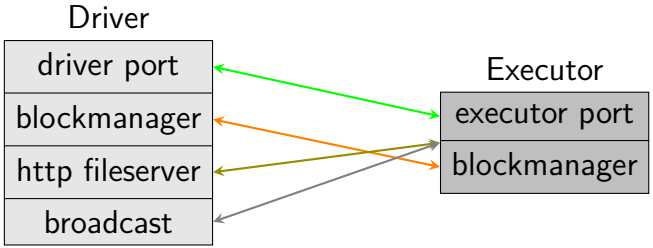
- DISK_ONLY
- DISK_ONLY_2
- MEMORY_ONLY
- MEMORY_ONLY_2
- MEMORY_ONLY_SER
- MEMORY_ONLY_SER_2
- MEMORY_AND_DISK
- MEMORY_AND_DISK_2
- MEMORY_AND_DISK_SER
- MEMORY_AND_DISK_SER_2

网络

集群内节点之间的网络连接

- spark.broadcast.port
- spark.driver.port
- spark.fileserver.port
- spark.blockManager.port
- spark.executor.port

上述端口号都是随机的，这为防火墙设置带来困难。



防火墙设置

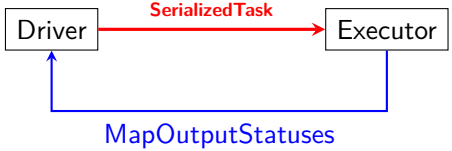
全部基于 tcp

hostname	确保组成集群的所有机器中的/etc/hosts 内容一致
spark_local_ip	在 conf/spark-env.sh 中正确设置了本机 IP 地址
firewall	打开相应的端口

防火墙规则

由于 worker 和 executor 打开的端口是随机的，不得不采用简单暴力的方式来设置防火墙
iptables -I INPUT 1 -p tcp -s *192.168.0.0/24* -j ACCEPT

spark.akka.framesize



消息类型

SerializedTask	Driver 需要将 Task 序列化后传递给 Executor
MapOutputStatuses	包含 Executor 将 Task 的执行结果, 如果结果超过 framesize, 将先存储在 BlockManager，然后 Driver 再从相应的 BlockManager 获取

问题重演

在 spark-defaults.conf 中将 *spark.akka.framesize* 设置为 1, 单位为 M

```
val ll = 1L to 2L*1024L*1024L toList
sc.makeRDD(ll,1).collect
```

问题规避

尝试使用 broadcast

```
val ll = 1L to 2L*1024L*1024L toList
sc.broadcast(ll)
sc.makeRDD(ll,1).collect
```

方法 2：加大 spark.akka.framesize 的值

通信开销

- 减少不必要的心跳检测
- 消息包压缩后传输, 使用 KryoSerializer

data locality

数据的远近会极大影响 Spark 任务的执行速度，当前支持以下几种不同的数据位置偏好，在调度的时候尽可能采取就近执行。

1. process_local 数据在同一进程内
2. node_local 数据在同一台机器中
3. no_pref 数据位置的远近不影响任务执行性能
4. rack_local 同一机架
5. any 任意机器

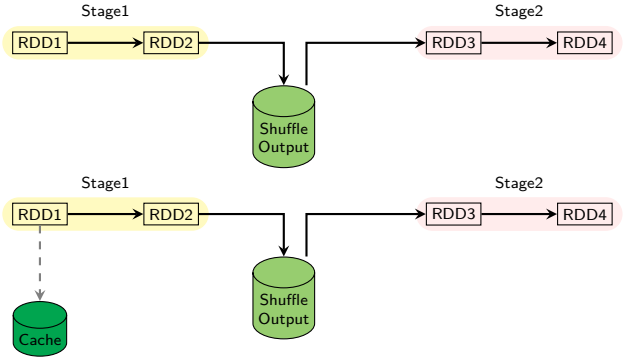
示例

通过 getPreferredLocations 来获取数据的地址信息
HadoopRDD.scala getPreferredLocations
Spark-Cassandra-Connector 中的 CassandraRDD
如果要将任务尽可能的分布到不同机器上，可以将 spark.localilty.wait 设置的很小

磁盘

磁盘

1. shuffle
2. cache
3. checkpoint



临时文件的存放和清理

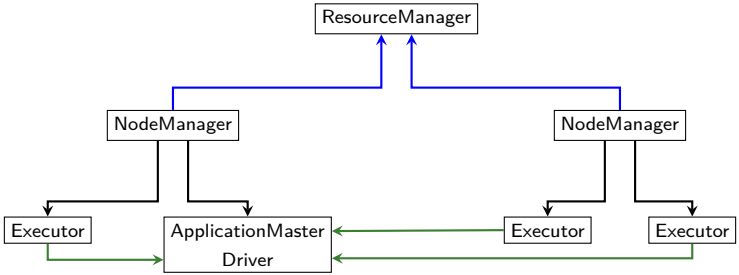
SPARK_WORKER_DIR	存放依赖文件和 executor 执行生成的日志信息	$SPARK_HOME/work
SPARK_LOCAL_DIRS	存放 shuffle 和 RDD 缓存以及相应的第三方依赖包	/tmp

存放于 SPARK_LOCAL_DIRS 目录下的内容会在 Application 退出时被自动清除，对于那些需要长时间运行的程序，可以通过指定 spark.cleaner.ttl 来进行定时清理。这里存在一个问题，就是包含有第三方依赖的 cache 包不会被自动清除。

存放于 SPARK_WORKER_DIR 目录下的内容默认不会被自动清除，需要通过指定 spark.worker.cleanup.enabled 来进行定时清除。在定时清除时，只会将已经运行完成且超过指定时长的文件夹清理掉。

YARN

Deploy Spark On YARN



内存都去哪了

- ApplicationMaster
- Driver
- Executor

参数名称	参数含义
yarn.scheduler.minimum-allocation-mb	AM 申请的最小内存
yarn.scheduler.maximum-allocation-mb	AM 申请的最大内存
yarn.nodemanager.resource.memory-mb	nodemanager 上报的内存数
yarn.nodemanager.resource.cpu-vcores	nodemanager 上报的 CPU 核数

实际申请量大于指定值

memoryOverhead 会导致实际申请的内存大于配置文件中指定的值，源码见 YarnAllocator.scala 和 YarnSparkHadoopUtil.scala

```
yarn.nodemanager.resource.memory-mb = 49152     # 48G
yarn.scheduler.maximum-allocation-mb = 24576    # 24G
SPARK_EXECUTOR_INSTANCES=1
SPARK_EXECUTOR_MEMORY=18G
SPARK_DRIVER_MEMORY=4G
```

$4 + 18 = 22 < 24$，理论上可以启动两个不同的应用程序，但实际上 Driver 会占用 5G，而 Executor 会申请多达 20G，$20 + 5 = 25 > 24$

用于 Cache 的内存容量

计算公式

(SPECIFIED_MEMORY – MEMORY_USED_BY_RUNTIME)*spark.storage.memoryFraction*spark.storage.safetyFraction

计算资源的动态调整

- 通过将 spark.dynamicAllocation.enabled 设置为 true，可以实现计算资源的动态调整，目前仅支持 YARN 模式。
- 仅当应用 A 在超过指定时间后不再有作业运行，应用 A 会将相应的 Executor 移除
- 当原本服务于应用 A 的 Executor 被动态移除后，Executor 产生的 Shuffle 结果会被保存，但原本存储于该 Executor 上的 Cache 结果不再可用。

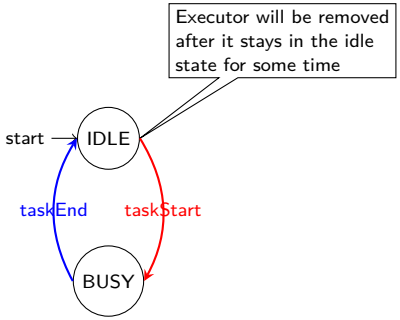


Figure: ExecutorAllocationManager 下的 Executor 状态机

Thank you

新浪微博：徽沪一郎
个人博客：cnblogs.com/hseagle