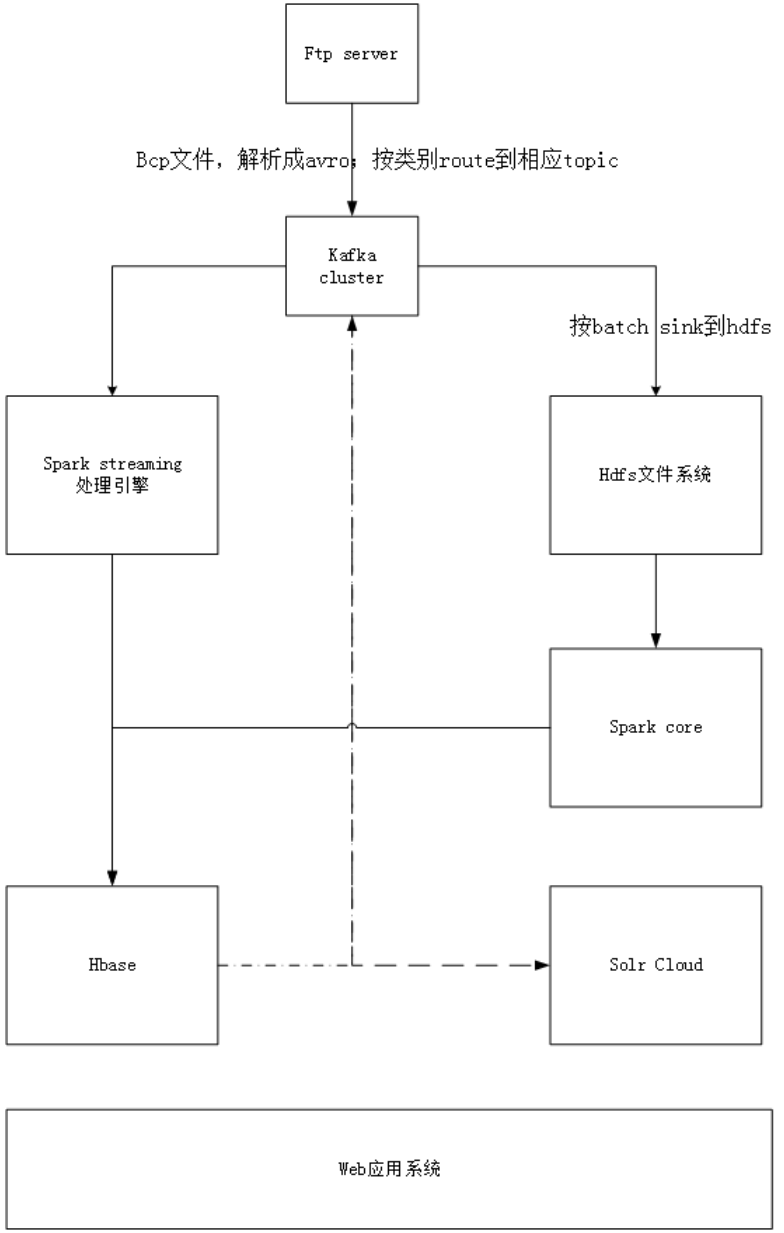
先前技术经历简介

个人开发积累

1、确保开发环境正常（helloworld、wordCount）
2、官方文档，业内团队、前辈的blog
（eg：阿里搜索团队，美团点评团队）
3、intellij下，f2(javadoc),f3(源代码),ctrl+t(继承结构),ctrl+G(被调用)



集群测试

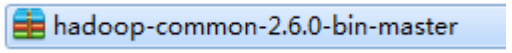
- 各服务web页面可正常，eg:50070、60010 etc

- 功能性测试
  - 相关命令测试，eg:hadoop dfs –ls /
  - Yarn job运行测试：
    - hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar pi 10 100
  - Spark job运行测试：
    - run-example --master yarn-cluster SparkPi 10

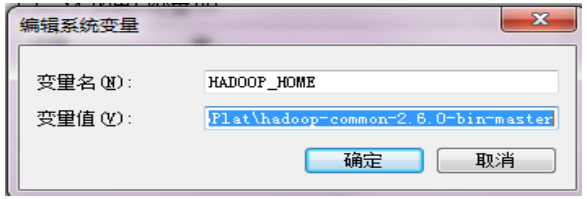
- 性能测试
  - Hdfs测试：
    - hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar randomwriter
  - Sort测试：
    - hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar sort

开发环境搭建

- 解压文件：hadoop-common-2.6.0-bin-master.zip



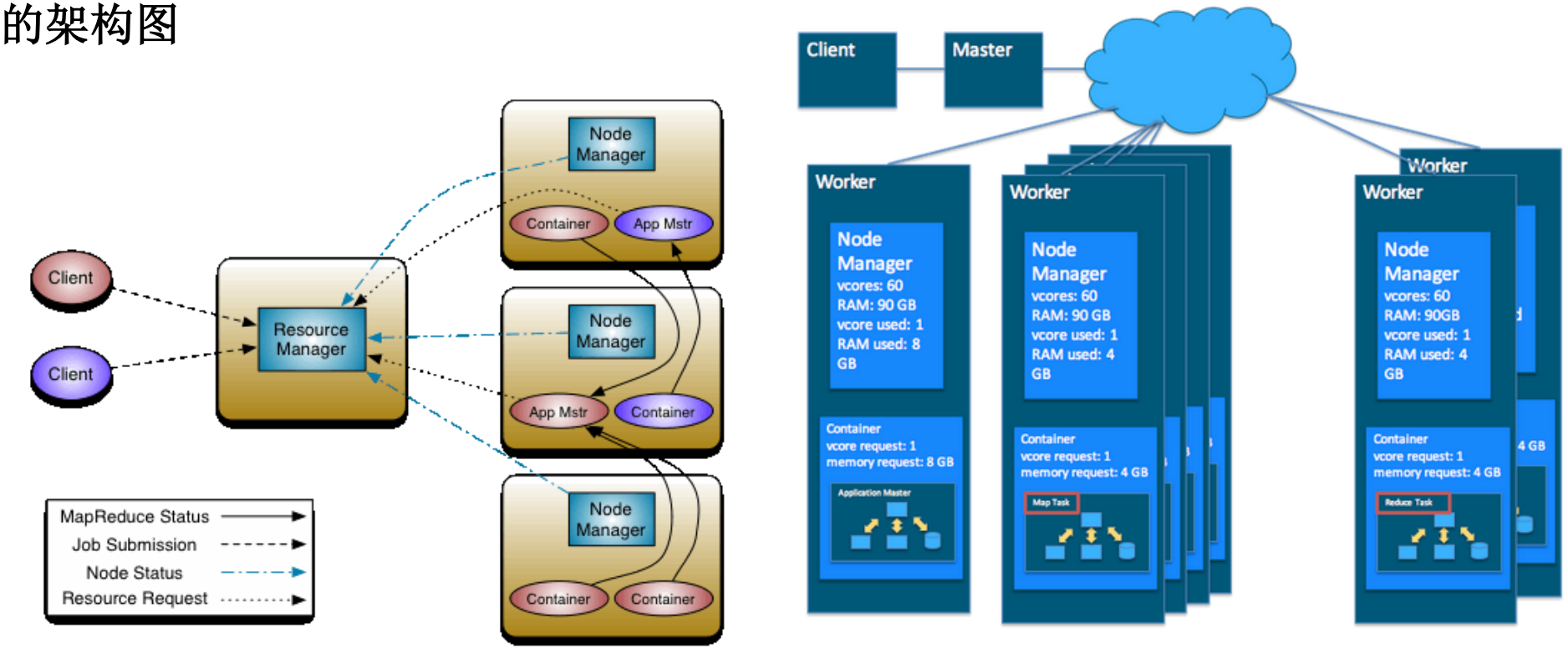
- 配置环境变量：
  - HADOOP_HOME=E:\developPlat\hadoop-common-2.6.0-bin-master
  - Path=……%JAVA_HOME%\bin;%HADOOP_HOME%\bin;……



- 重启电脑
- 运行spark WordCount测试代码

```scala
package com.hikvision.env

import org.apache.spark.{SparkConf, SparkContext}

/**
  * Created by likai14 on 2017/4/21.
  *
  * 测试开发环境
  */
object WordCount {
  val splitor = "\\s+"

  val conf = new SparkConf().setAppName(this.getClass.getSimpleName)
    .setMaster("local")
  val sc = new SparkContext(conf)

  def main(args: Array[String]): Unit = {
    val resultRdd = sc.textFile("file:///F:/dev/datacenter/input/env/*").flatMap(_.split(splitor).map(word => (word, 1))).reduceByKey(_ + _)

    //    resultRdd.foreach(println)

    resultRdd.saveAsTextFile("file:///F:/dev/datacenter/output/env/spark/" + System.currentTimeMillis())

    sc.stop()
  }
}
```

Yarn的架构图



参考：Hadoop Yarn详解

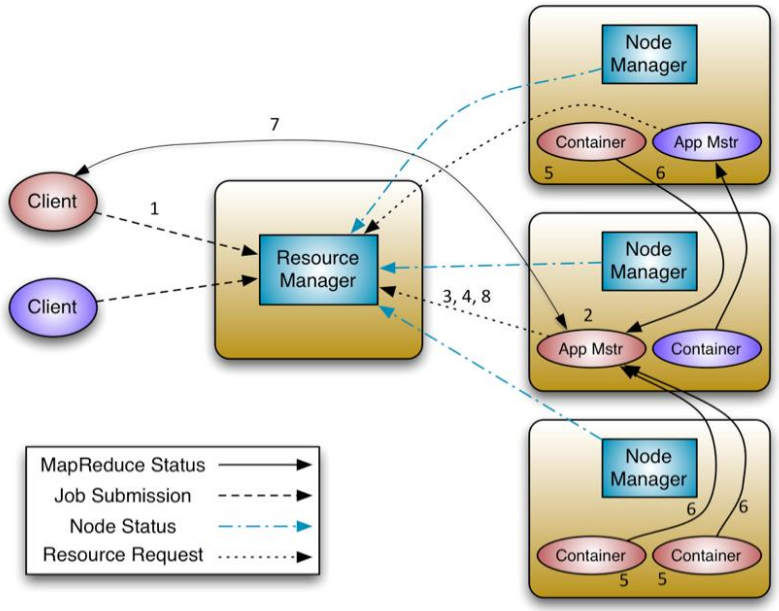
The per-application **ApplicationMaster** is, in effect, a framework specific library and is tasked with negotiating resources from the **ResourceManager** and working with the NodeManager(s) to execute and monitor the tasks, has the responsibility of **negotiating appropriate resource containers** from the Scheduler, **tracking their status and monitoring for progress**.

The Scheduler(可插拔:Capacity vs Fair) is responsible for **allocating resources** to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no **monitoring or tracking of status** for the application. Also, it offers no guarantees about **restarting failed tasks** either due to application failure or hardware failures. The Scheduler performs its scheduling function based **the resource requirements** of the applications; it does so based on **the abstract notion of a resource *Container*** which incorporates elements such as memory, cpu, disk, network etc.

The ApplicationsManager is responsible for **accepting job-submissions, negotiating the first container** for executing the application specific ApplicationMaster and provides the service **for restarting the ApplicationMaster container** on failure.

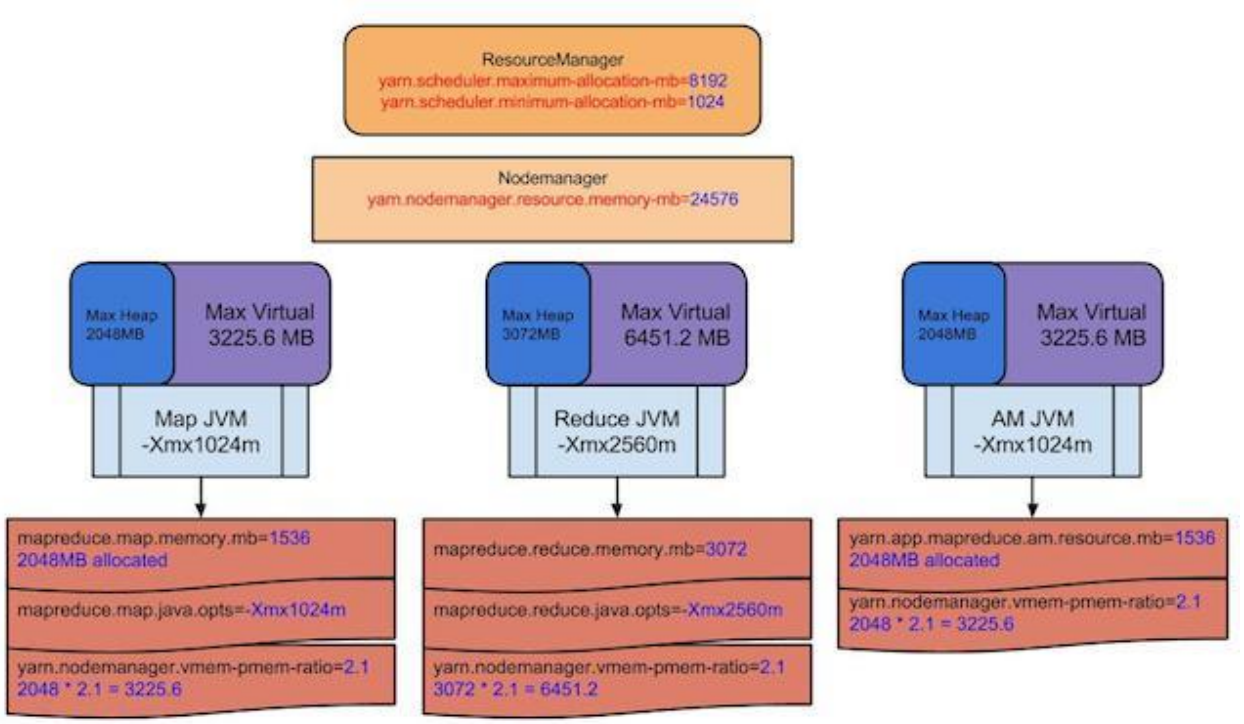
Yarn request分析

1. 应用程序提交
2. 启动应用的ApplicationMaster实例
3. ApplicationMaster实例管理应用程序的执行



1. client向RM(ResourceManager)提交应用并请求一个AM(ApplicationMaster)实例
2. RM找到可以运行一个Container的NodeManager，在这Container启动AM实例
3. AM向RM进行注册，注册之后客户端就可以查询RM获得自己AM的详细信息，以后就可以和自己的AM直接交互了
4. 在平常的操作过程中，AM根据resource-request协议向RM发送resource-request（<resource-name, priority, resource-requirement, number-of-containers>）请求
5. 当Container被成功分配之后，AM通过向NodeManager发送container-launch-specification信息来启动Container， container-launch-specification信息包含了能够让Container和AM交流所需要的资料
6. 应用程序的代码在启动的Container中运行，并把运行的进度、状态等信息通过application-specific协议发送给AM
7. 在应用程序运行期间，提交应用的客户端主动和AM交流获得应用的运行状态、进度更新等信息，交流的协议也是application-specific协议
8. 一但应用程序执行完成并且所有相关工作也已经完成，AM向RM取消注册然后关闭，用到所有的Container也归还给系统

集群服务配置→Hadoop篇



[具体参考：Yarn 内存分配管理机制及相关参数配置]

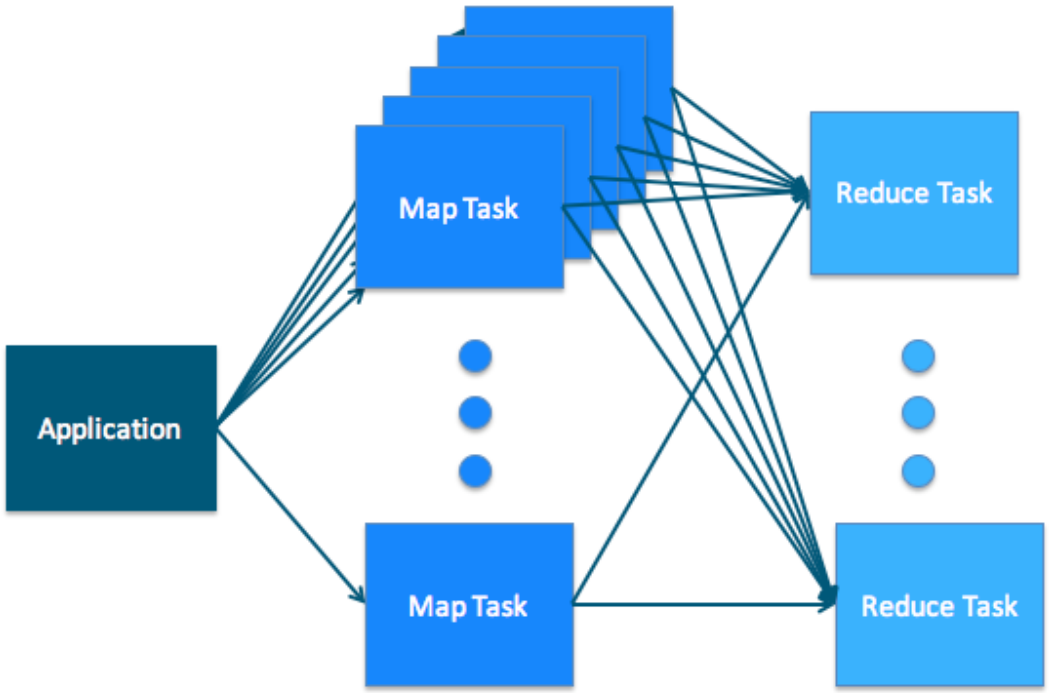
vim yarn-site.xml

```
<!—配置分配给yarn资源池的内存总量 -->
    <property>
      <name>yarn.nodemanager.resource.memory-mb</name>
      <value>102400</value>    </property>
<!—配置分配给yarn资源池的cpu虚拟core -->
    <property>
      <name>yarn.nodemanager.resource.cpu-vcores</name>
      <value>20</value>    </property>
<!—yarn资源池分配给container的最大内存量 -->
    <property>
      <name>yarn.scheduler.maximum-allocation-mb</name>
      <value>24024</value>    </property>
<!—yarn资源池分配给container的最大cpuxunicore量-->
    <property>
      <name>yarn.scheduler.maximum-allocation-vcores</name>
      <value>20</value>    </property>
```

vim mapred-site.xml

```
<!—每个 Map Task 需要的内存量 -->
    <property>
      <name>mapreduce.map.memory.mb</name>
      <value>2048</value></property>
<!—每个 Reduce Task 需要的内存量 -->
    <property>
      <name>mapreduce.reduce.memory.mb</name>
      <value>24024</value></property>
<!—maps 中对 jvm child 设置更大的堆大小 -->
    <property>
      <name>mapreduce.map.java.opts</name>
      <value>-Xmx2048m</value></property>
<!—reduces 对 child jvms Larger heap-size 设置 -->
    <property>
      <name>mapreduce.reduce.java.opts</name>
      <value>-Xmx24024m</value></property>
<!—标注 Deprecateded；由上面两项替代实现； -->
    <property>
      <name>mapred.child.java.opts</name>
      <value>-Xmx2048m</value>    </property>
```

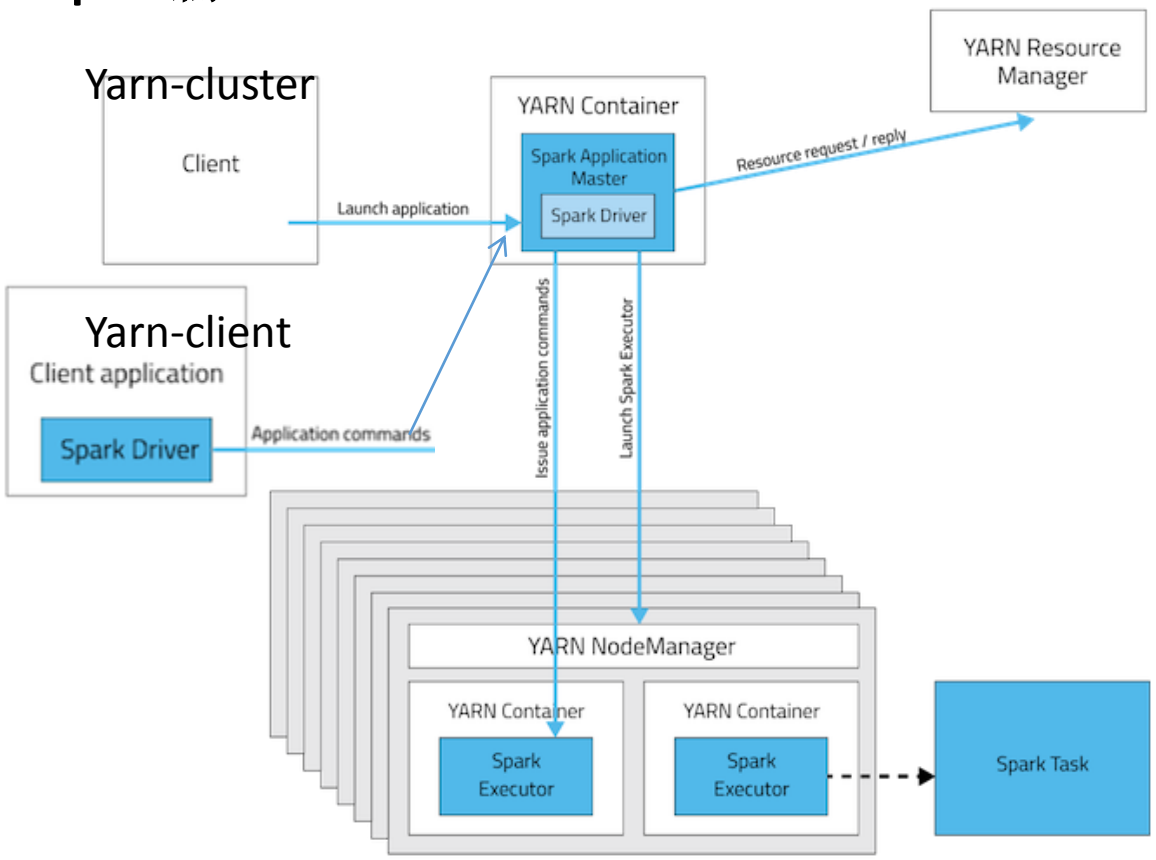
编程模型




```java
public class WorldCountMR {
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable( value: 1);
        private Text word = new Text();
        @Override
        public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            String[] words = value.toString().split( regex: "\\s+");
            for (String w : words) {
                word.set(w);
                context.write(word, one);
            }
        }
    }
    public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
    public static void run(String[] args) throws IOException, ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf, jobName: "word count");
        job.setJarByClass(WorldCountMR.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        int exitCode = job.waitForCompletion( verbose: true) ? 0 : 1;
        System.exit(exitCode);
    }
```

IO format：数据输入输出格式

Spark篇



	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark Slave
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and Workers
Supports Spark Shell?	No	Yes	Yes

Spark插拔式资源管理

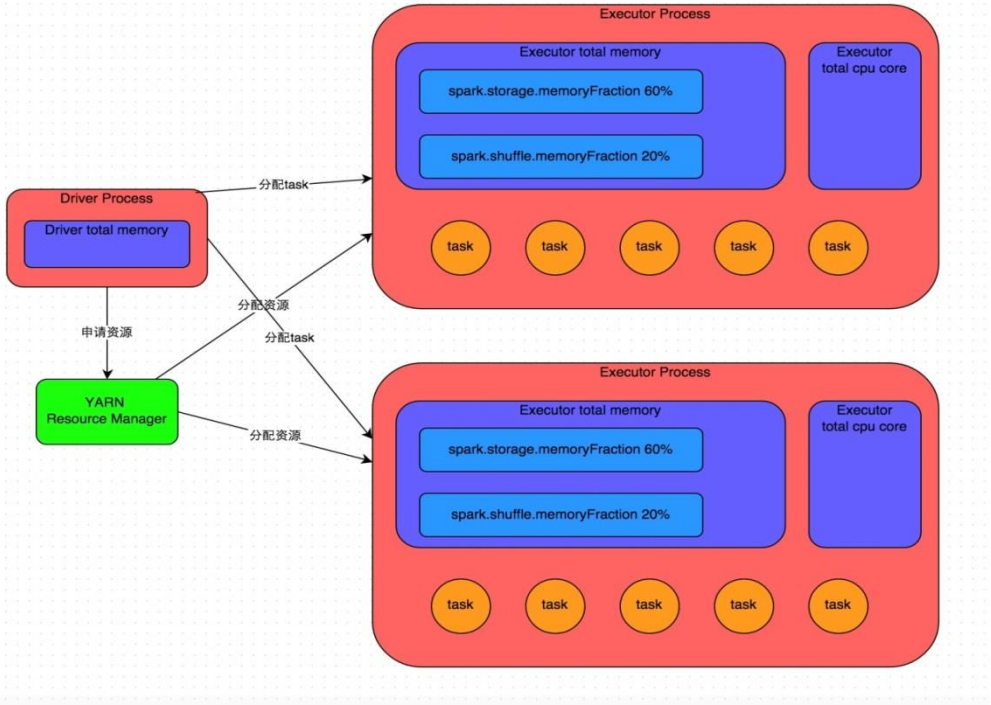
Spark支持Yarn,Mesos,Standalone三种集群部署模式，它们的共同点：Master服务(Yarn ResourceManager,Mesos master,Spark standalone)来决定哪些应用可以运行以及在哪什么时候运行，Slave服务(Yarn NodeManger)运行在每个节点上，节点上实际运行着Executor进程，此外还监控着它们的运行状态以及资源的消耗

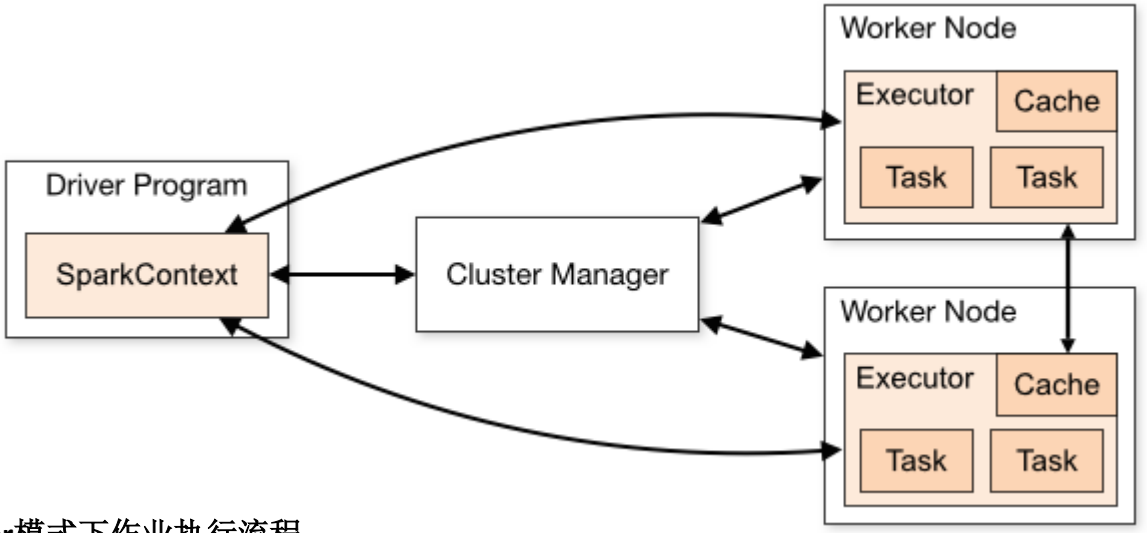
Spark On Yarn的优势

1. Spark支持资源动态共享，运行于Yarn的框架都共享一个集中配置好的资源池
2. 可以很方便利用Yarn的资源调度特性来做分类·，隔离以及优先级控制负载，拥有更灵活的调度策略
3.Yarn可以自由地选择executor数量
4.Yarn是唯一支持Spark安全的集群管理器，使用Yarn，Spark可以运行于Kerberized Hadoop之上，在它们进程之间进行安全认证

- **Application:** This may be a single job, a sequence of jobs, a long-running service issuing new commands as needed or an interactive exploration session.
- **Spark Driver:** The Spark driver is the process running the spark context (which represents the application session). This driver is responsible for converting the application to a directed graph of individual steps to execute on the cluster. There is one driver per application.
- **Spark Application Master:** The Spark Application Master is responsible for negotiating resource requests made by the driver with YARN and finding a suitable set of hosts/containers in which to run the Spark applications. There is one Application Master per application.
- **Spark Executor:** A single JVM instance on a node that serves a single Spark application. An executor runs multiple tasks over its lifetime, and multiple tasks concurrently. A node may have several Spark executors and there are many nodes running Spark Executors for each client application.
- **Spark Task:** A Spark Task represents a unit of work on a partition of a distributed dataset.

Spark篇





Yarn-cluster模式下作业执行流程：

1. 客户端生成作业信息提交给ResourceManager(RM)
2. RM在某一个NodeManager(由Yarn决定)启动container并将Application Master(AM)分配给该NodeManager(NM)
3. NM接收到RM的分配，启动Application Master并初始化作业，此时这个NM就称为Driver
4. Application向RM申请资源，分配资源同时通知其他NodeManager启动相应的Executor
5. Executor向NM上的Application Master注册汇报并完成相应的任务

在Spark On Yarn模式下，每个Spark Executor作为一个Yarn container在运行

- **Block，InputSplit(**不能跨越文件)
- **InputSplit**生成具体的**Task**, InputSplit与Task是一一对应
- 具体的Task每个都会被分配到集群上的某个**Executor**去执行
- 每个Executor由若干**core**组成，每个Executor的每个core一次只能执行一个Task。
- 每个Task执行的结果就是生成了目标**RDD**的一个**partiton**
- **注意:** 这里的core是虚拟的core而不是物理CPU核，可以理解为Executor的一个工作线程。
- Task被执行的并发度 = Executor数目 * 每个Executor核数。
- partition的数目：对于数据读入阶段，例如sc.textFile，输入文件被划分为多少InputSplit,就会需要多少初始Task。1、在Map阶段partition数目保持不变。2、在Reduce阶段，RDD的聚合会触发shuffle操作，聚合后的RDD的partition数目跟具体操作有关，例如repartition操作会聚合成指定分区数，还有一些算子是可配置的。

- **Driver Program**: The driver program is responsible for managing the job flow and scheduling tasks that will run on the executors.
- **Executors**: Executors are processes that run computation and store data for a Spark application.
- **Cluster Manager**: Cluster Manager is responsible for starting executor processes and where and when they will be run. Spark supports pluggable cluster manager, it supports (YARN, Mesos, and its own “standalone” cluster manager)

job监控页面实战

```
[root@node68 04]# spark-submit --master yarn-cluster --class com.hikvision.env.bussiness.EtlOracle2HdfsSimplePartAdv ./env-test.j
ar  10 10000 100000 20
```

node68:18080

Spark 2.1.0-hdh4.0.0 **History Server**

Event log directory: hdfs://node68:8020/var/log/spark_hislog

Last updated: 2017-5-9, 20:18:59

1

Search:

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
application_1494233131806_0006	Spark Pi	2017-05-09 08:15:23	2017-05-09 08:15:30	6 s	root	2017-05-09 08:15:30	Download

Spark 2.1.0-hdh4.0.0 Jobs Stages Storage Environment Executors

Spark Jobs (?)

User: root
Total Uptime: 6 s
Scheduling Mode: FIFO
Completed Jobs: 1

2

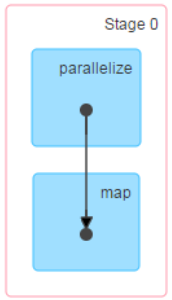
▸ Event Timeline

Completed Jobs (1)

Job Id	Description	Submitted
0	reduce at SparkPi.scala:38	2017/05/09 16:15:29

Status: SUCCEEDED
Completed Stages: 1

▸ Event Timeline
▾ DAG Visualization



3

Completed Stages (1)

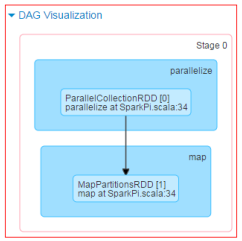
Stage Id	Description
0	reduce at SparkPi.scala:38

Spark 2.1.0-hdh4.0.0 Jobs Stages Storage Environment Executors

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 0.2 s
Locality Level Summary: Process local: 10

▾ DAG Visualization



4

▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 10 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	7 ms	7 ms	9 ms	9 ms	89 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

▾ Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks
1 stdout stderr	node68:60080	0.6 s	5	0	0	5
2 stdout stderr	node68:56251	0.7 s	5	0	0	5

Tasks (10)

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time
0	0	0	SUCCESS	PROCESS_LOCAL	2 / node68 stdout stderr	2017/05/09 16:15:29	89 ms	

集群服务配置→Spark篇

num-executors
参数说明：用于设置Spark作业总共要用多少个Executor进程来执行。Driver在向YARN集群管理器申请资源时，YARN集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的Executor进程。如果不设置的话，默认只会给你启动少量的Executor进程，此时你的Spark作业的运行速度是非常慢的。
参数调优建议：每个Spark作业的运行一般设置50~100个左右的Executor进程比较合适，设置太少或太多的Executor进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。
executor-memory
参数说明：用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能，而且跟常见的JVM OOM异常，也有直接的关联。
参数调优建议：每个Executor进程的内存设置4G~8G较为合适。但是这只是一个参考值，具体的设置还是得根据不同部门的资源队列来定。可以看看自己团队的资源队列的最大内存限制是多少，num-executors乘以executor-memory，就代表了你的Spark作业申请到的总内存量（也就是所有Executor进程的内存总和），这个量是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的总内存量最好不要超过资源队列最大总内存的1/3~1/2，避免你自己的Spark作业占用了队列所有的资源，导致别的同学的作业无法运行。
executor-cores
参数说明：用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core数量越多，越能够快速地执行完分配给自己的所有task线程。
参数调优建议：Executor的CPU core数量设置为2~4个较为合适。同样得根据不同部门的资源队列来定，可以看看自己的资源队列的最大CPU core限制是多少，再依据设置的Executor数量，来决定每个Executor进程可以分配到几个CPU core。同样建议，如果是跟他人共享这个队列，那么num-executors * executor-cores不要超过队列总CPU core的1/3~1/2左右比较合适，也是避免影响其他同学的作业运行。
driver-memory
参数说明：用于设置Driver进程的内存。
参数调优建议：Driver内存通常不设置，设置1G就够。如果使用collect将RDD的数据全部拉取到Driver上，须确保Driver的内存足够大，否则OOM。

实例：spark-submit --num-executors 15 --executor-cores 1 --executor-memory 1g --master yarn --deploy-mode cluster \--class com.chaosdata.etl.load.oracle.OracleDataLoad

Transformation
map(func)
filter(func)
flatMap(func)
mapPartitions(func)
mapPartitionsWithIndex(func)
sample(withReplacement, fraction, seed) Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.取样
union(otherDataset)
intersection(otherDataset) Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks]))
groupByKey([numTasks])
reduceByKey(func, [numTasks])
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])
sortByKey([ascending], [numTasks])
join(otherDataset, [numTasks])
cogroup(otherDataset, [numTasks]) datasets of type (K, V) and (K, W), returns (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(otherDataset)笛卡尔积
pipe(command, [envVars]) Pipe each partition of the RDD through a shell command,
coalesce(numPartitions) 减少rdd 分区的数量，运行经过filter处理过大dataset时高效
repartition(numPartitions)
repartitionAndSortWithinPartitions(partitioner)

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Self define eg:saveAsNewApiHadoopDataSet

Broadcast Variables

用于共享数据，应用：布控报警

Accumulators

Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

Spark sql oracle数据导出实战

```scala
/**
  * dataframe结合
  *
  * @return
  */
def unionTableReducer: (DataFrame, DataFrame) => DataFrame = (x: DataFrame, y: DataFrame) => x.union(y)

/**
  * 构建分页查询语句
  *
  * @param index
  * @param interval
  * @return
  */
def query(index: Int, interval: Int): String = {
  val basic = " ( select a.*,rownum as rn from ZHCX_BDQ.BDQ_CXB_QG a ) b "

  val condition = " where  b.rn between " + ((index - 1) * interval + 1) + " AND " + (index) * interval


  " ( select * from " + basic + condition + " ) c"
}
```

```scala
val jdbcDF = (1 to index.toInt).map(index => {
  ssc.read.format("jdbc").options(
    Map("url" -> "jdbc:oracle:thin:@10.17.139.66:1521:xe",
      "user" -> "sparta",
      "password" -> "sparta",
      "owner" -> "SPARTA",
      "dbtable" -> query(index, interval), //核心
      "fetch.size" -> fetchSize,
      "driver" -> "oracle.jdbc.driver.OracleDriver")).load()
}
).reduce(unionTableReducer)

//重新分区
val rdd = jdbcDF.rdd.repartition(partition).cache()
```

开发调优

1. 避免创建重复的RDD
2. 尽可能复用同一个RDD
3. 对多次使用的RDD进行持久化
    1. cache()方法表示：使用非序列化的方式将RDD中的数据全部尝试持久化到内存中
    2. persist()方法表示：手动选择持久化级别，并使用指定的方式进行持久化
4. 尽量避免使用shuffle类算子
    1. 比如实现join时，优先考虑：Broadcast与map结合进行join
5. 使用map-side预聚合的shuffle操作
6. 使用高性能的算子
    1. 使用reduceByKey/aggregateByKey替代groupByKey
    2. 使用mapPartitions替代普通map
    3. 使用filter之后进行coalesce操作
    4. 使用repartitionAndSortWithinPartitions替代repartition与sort类操作
7. 广播大变量
    1. 默认在task粒度，广播后在executor粒度
8. 使用Kryo优化序列化性能
9. 优化数据结构（相对）

数据倾斜调优

数据倾斜发生时的现象

1. 绝大多数task执行得都非常快，但个别task执行极慢。比如，总共有1000个task，997个task都在1分钟之内执行完了，但是剩余两三个task却要一两个小时。这种情况很常见。
2. 原本能够正常执行的Spark作业，某天突然报出OOM（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

原理：数据倾斜只会发生在shuffle过程中；某个节点上的一个task对应的数据量特别大；

数据倾斜的解决方案

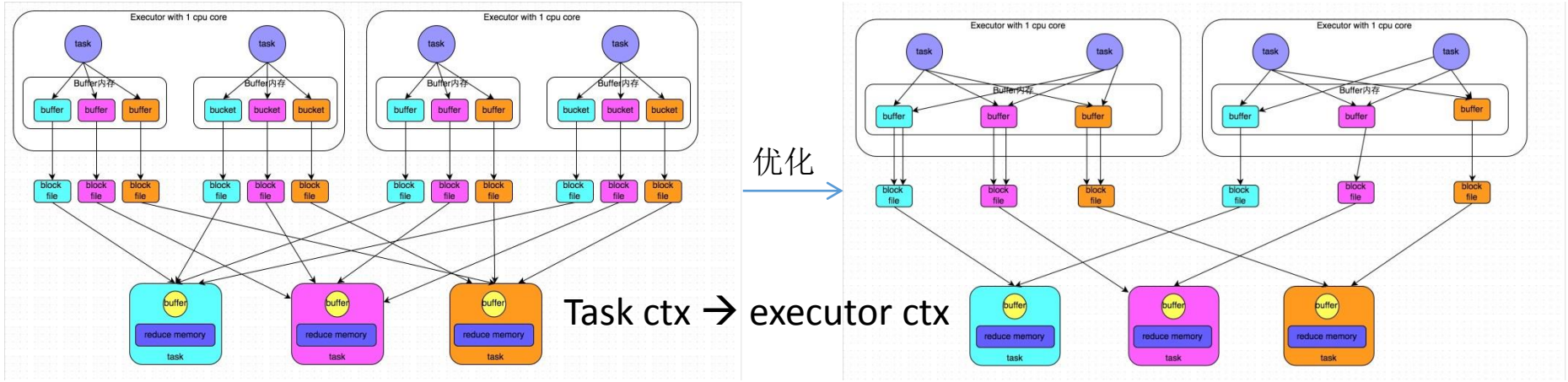
1. 使用Hive ETL预处理数据
2. 过滤少数导致倾斜的key
3. 提高shuffle操作的并行度
4. 两阶段聚合（局部聚合+全局聚合）
5. 将reduce join转为map join
6. 采样倾斜key并分拆join操作
7. 使用随机前缀和扩容RDD进行join　　reduceByKey left key加range前缀；那么right key 必须扩容range.size倍（穷举）
8. 多种方案组合使用

shuffle调优

大多数Spark作业的性能主要就是消耗在了shuffle环节，她包含many disk IO、序列化、net IO等操作。So want more，有必要对shuffle调优。but，影响Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占little。因此大家务必把握住调优的基本原则，千万不要舍本逐末。

HashShuffleManager

shuffle read的拉取过程是一边拉取一边进行聚合的。每个shuffle read task都会有一个自己的buffer缓冲，每次都只能拉取与buffer缓冲相同大小的数据，然后通过内存中的一个Map进行聚合等操作。



SortShuffleManager

Normal vs bypass

1. 磁盘写机制不同
2. 不会进行排序