

# 尚硅谷大数据技术之 Hadoop (Yarn)

(作者：尚硅谷大数据研发部)

版本 V3.0

## 第 1 章 Yarn 资源调度器

**Yarn** 是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而 **MapReduce** 等运算程序则相当于运行于操作系统之上的应用程序。

### 1.1 Yarn 基本架构

YARN 主要由 ResourceManager、NodeManager、ApplicationMaster 和 Container 等组件构成，如图 1-1 所示。

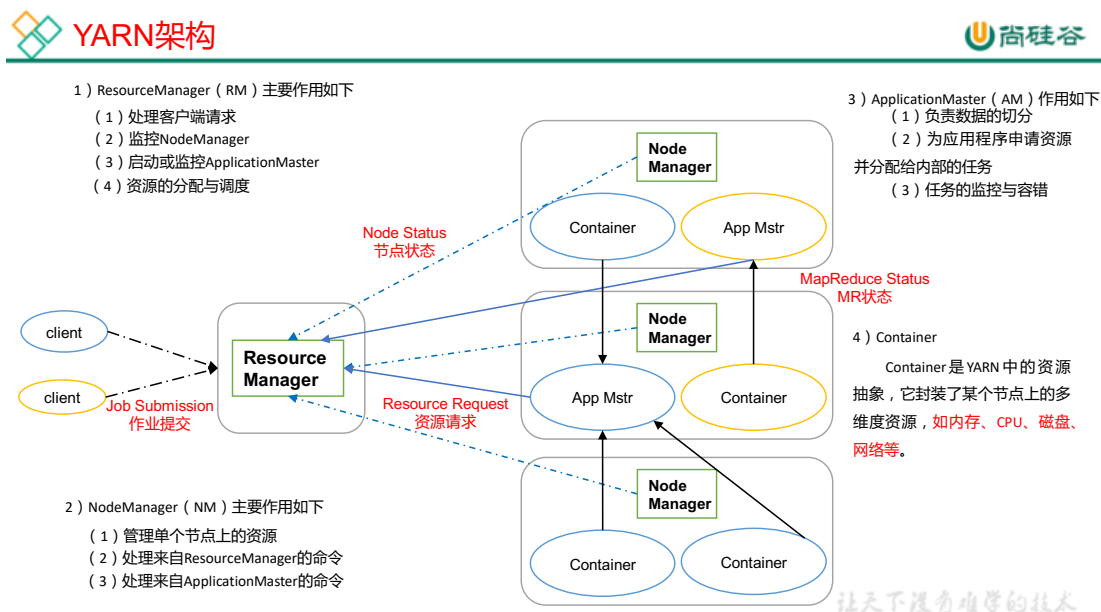


图 1-1 Yarn 基本架构

## 1.2 Yarn 工作机制

1. Yarn 运行机制，如图 1-2 所示。

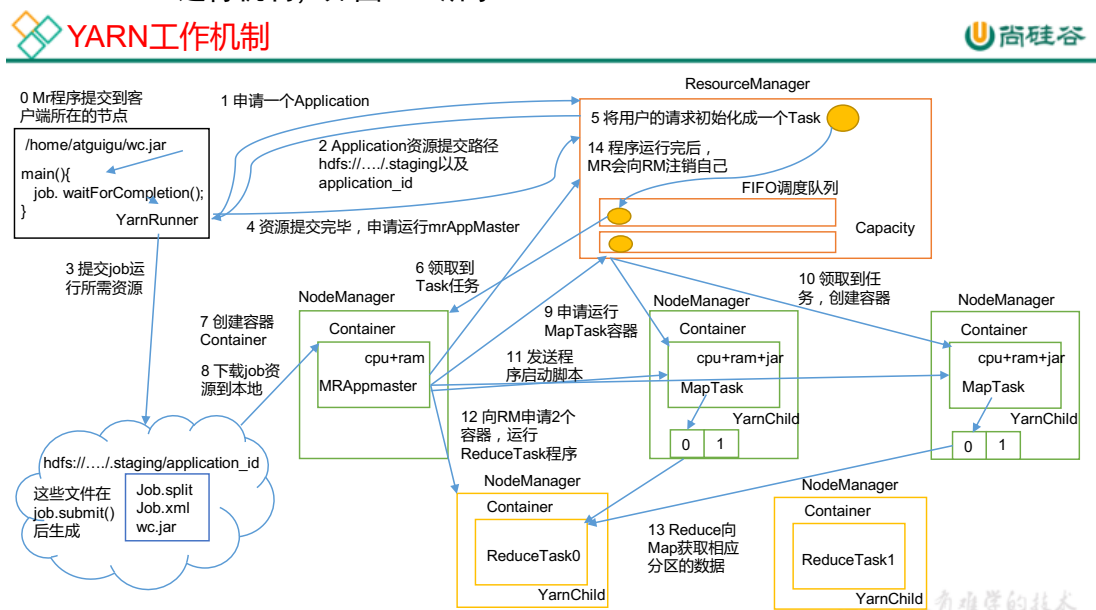


图 1-2 Yarn 工作机制

### 2. 工作机制详解

- (1) MR 程序提交到客户端所在的节点。
- (2) YarnRunner 向 ResourceManager 申请一个 Application。
- (3) RM 将该应用程序的资源路径返回给 YarnRunner。
- (4) 该程序将运行所需资源提交到 HDFS 上。
- (5) 程序资源提交完毕后，申请运行 mrAppMaster。
- (6) RM 将用户的请求初始化成一个 Task。
- (7) 其中一个 NodeManager 领取到 Task 任务。
- (8) 该 NodeManager 创建容器 Container，并产生 MRAppmaster。
- (9) Container 从 HDFS 上拷贝资源到本地。
- (10) MRAppmaster 向 RM 申请运行 MapTask 资源。
- (11) RM 将运行 MapTask 任务分配给另外两个 NodeManager，另两个 NodeManager 分别领取任务并创建容器。
- (12) MR 向两个接收到任务的 NodeManager 发送程序启动脚本，这两个 NodeManager 分别启动 MapTask，MapTask 对数据分区排序。
- (13) MrAppMaster 等待所有 MapTask 运行完毕后，向 RM 申请容器，运行 ReduceTask。

(14) ReduceTask 向 MapTask 获取相应分区的数据。

(15) 程序运行完毕后，MR 会向 RM 申请注销自己。

### 1.3 作业提交全过程

1. 作业提交过程之 YARN，如图 1-3 所示。

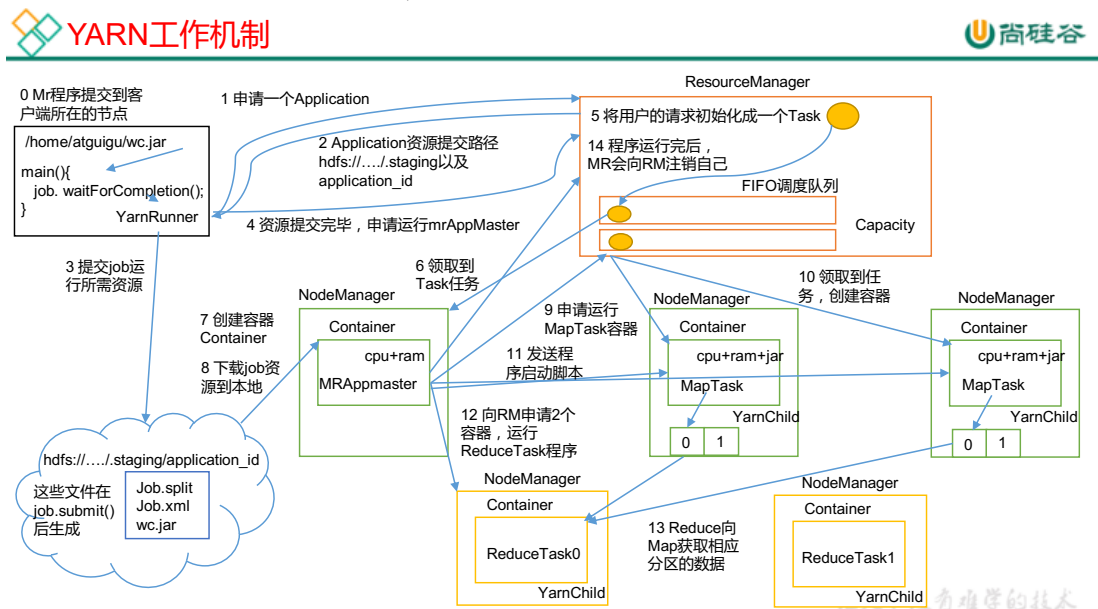


图 1-3 作业提交过程之 Yarn

作业提交全过程详解

#### (1) 作业提交

第 1 步：Client 调用 `job.waitForCompletion` 方法，向整个集群提交 MapReduce 作业。

第 2 步：Client 向 RM 申请一个作业 id。

第 3 步：RM 给 Client 返回该 job 资源的提交路径和作业 id。

第 4 步：Client 提交 jar 包、切片信息和配置文件到指定的资源提交路径。

第 5 步：Client 提交完资源后，向 RM 申请运行 MrAppMaster。

#### (2) 作业初始化

第 6 步：当 RM 收到 Client 的请求后，将该 job 添加到容量调度器中。

第 7 步：某一个空闲的 NM 领取到该 Job。

第 8 步：该 NM 创建 Container，并产生 MRAppmaster。

第 9 步：下载 Client 提交的资源到本地。

#### (3) 任务分配

第 10 步：MrAppMaster 向 RM 申请运行多个 MapTask 任务资源。

第 11 步: RM 将运行 MapTask 任务分配给另外两个 NodeManager, 另两个 NodeManager 分别领取任务并创建容器。

#### (4) 任务运行

第 12 步: MrAppMaster 向两个接收到任务的 NodeManager 发送程序启动脚本, 这两个 NodeManager 分别启动 MapTask, MapTask 对数据分区排序。

第 13 步: MrAppMaster 等待所有 MapTask 运行完毕后, 向 RM 申请容器, 运行 ReduceTask。

第 14 步: ReduceTask 向 MapTask 获取相应分区的数据。

第 15 步: 程序运行完毕后, MR 会向 RM 申请注销自己。

#### (5) 进度和状态更新

YARN 中的任务将其进度和状态(包括 counter)返回给应用管理器, 客户端每秒(通过 `mapreduce.client.progressmonitor.pollinterval` 设置)向应用管理器请求进度更新, 展示给用户。

#### (6) 作业完成

除了向应用管理器请求作业进度外, 客户端每 5 秒都会通过调用 `waitForCompletion()` 来检查作业是否完成。时间间隔可以通过 `mapreduce.client.completion.pollinterval` 来设置。作业完成之后, 应用管理器和 Container 会清理工作状态。作业的信息会被作业历史服务器存储以备之后用户核查。

## 2. 作业提交过程之 MapReduce, 如图 1-4 所示

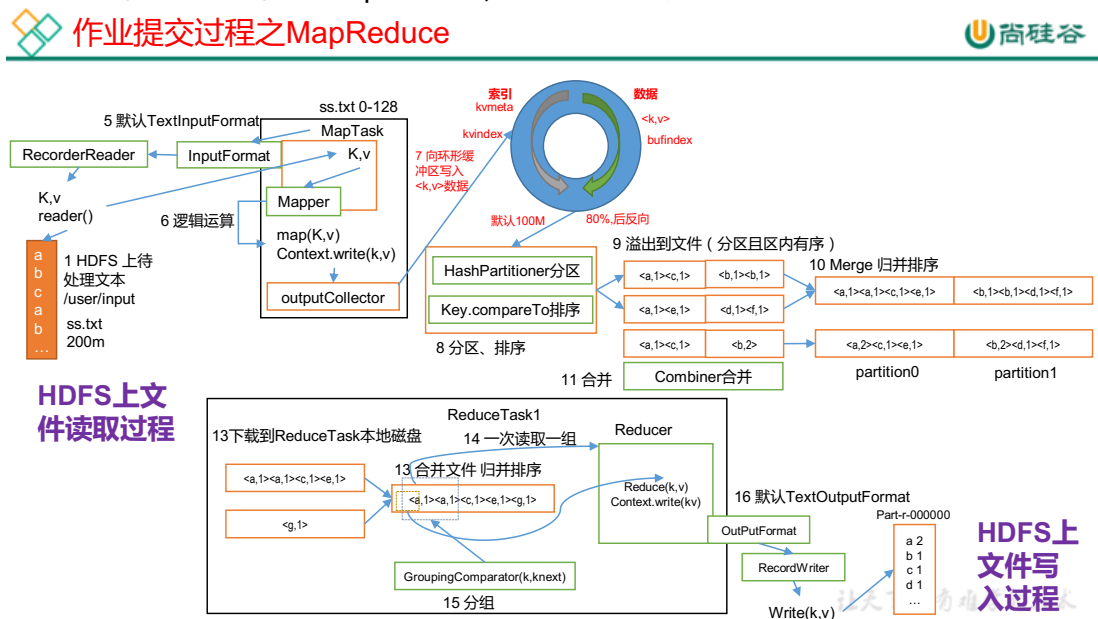


图 1-4 作业提交过程之 MapReduce

## 1.4 资源调度器

目前，Hadoop 作业调度器主要有三种：FIFO、Capacity Scheduler 和 Fair Scheduler。

Hadoop3.1.3 默认的资源调度器是 Capacity Scheduler。

具体设置详见：yarn-default.xml 文件

```
<property>
  <description>The class to use as the resource
scheduler.</description>
  <name>yarn.resourcemanager.scheduler.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.scheduler
.capacity.CapacityScheduler</value>
</property>
```

1. 先进先出调度器（FIFO），如图 1-5 所示

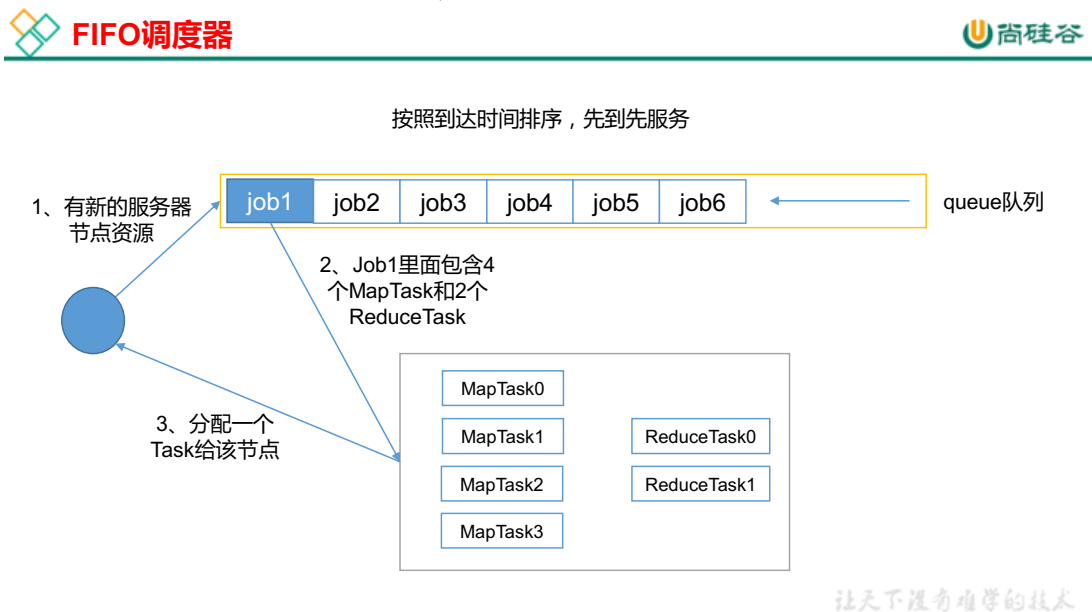


图 1-5 FIFO 调度器

Hadoop 最初设计目的是支持大数据批处理作业，如日志挖掘、Web 索引等作业，

为此，Hadoop 仅提供了一个非常简单的调度机制：FIFO，即先来先服务，在该调度机制下，所有作业被统一提交到一个队列中，Hadoop 按照提交顺序依次运行这些作业。

但随着 Hadoop 的普及，单个 Hadoop 集群的用户量越来越大，不同用户提交的应用程序往往具有不同的服务质量要求，典型的应用有以下几种：

批处理作业：这种作业往往耗时较长，对时间完成一般没有严格要求，如数据挖掘、机器学习等方面的应用程序。

交互式作业：这种作业期望能及时返回结果，如 SQL 查询（Hive）等。

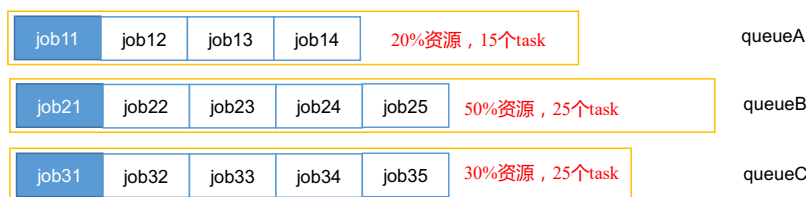
生产性作业：这种作业要求有一定的资源保证，如统计值计算、垃圾数据分析等。

此外，这些应用程序对硬件资源需求量也是不同的，如过滤、统计类作业一般为 CPU 密集型作业，而数据挖掘、机器学习作业一般为 I/O 密集型作业。因此，简单的 FIFO 调度策略不仅不能满足多样化需求，也不能充分利用硬件资源。

## 2. 容量调度器 (Capacity Scheduler)，如图 1-6 所示

### 容量调度器

按照到达时间排序，先到先服务



- 1、支持多个队列，每个队列可配置一定的资源量，每个队列采用FIFO调度策略。
- 2、为了防止同一个用户的作业独占队列中的资源，该调度器会对同一用户提交的作业所占资源量进行限定。
- 3、首先，计算每个队列中正在运行的任务数与其应该分得的计算资源之间的比值，选择一个该比值最小的队列——最闲的。
- 4、其次，按照作业优先级和提交时间顺序，同时考虑用户资源限制和内存限制对队列内任务排序。
- 5、三个队列同时按照任务的先后顺序依次执行，比如，job11、job21和job31分别排在队列最前面，先运行，也是并行运行。

让天下没有难学的技术

图 1-6 容量调度器

Capacity Scheduler Capacity Scheduler 是 Yahoo 开发的多用户调度器，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用。而当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。

总之，Capacity Scheduler 主要有以下几个特点：

- ①容量保证。管理员可为每个队列设置资源最低保证和资源使用上限，而所有提交到该队列的应用程序共享这些资源。
- ②灵活性，如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其他队列释放的资源会归还给该队列。这种资源灵活分配的方式可明显提高资源利用率。
- ③多重租赁。支持多用户共享集群和多应用程序同时运行。为防止单个应用程序、用户或者队列独占集群中的资源，管理员可为之增加多重约束（比如单个应用程序同时运行的任务数等）。

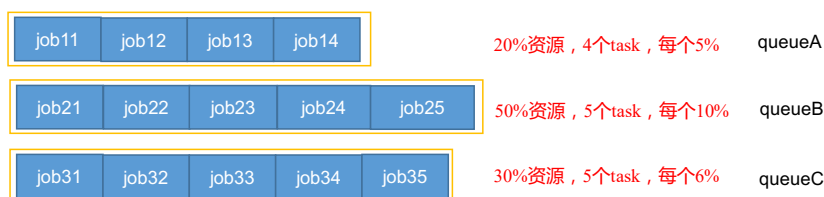
④安全保证。每个队列有严格的 ACL 列表规定它的访问用户，每个用户可指定哪些用户允许查看自己应用程序的运行状态或者控制应用程序（比如杀死应用程序）。此外，管理员可指定队列管理员和集群系统管理员。

⑤动态更新配置文件。管理员可根据需要动态修改各种配置参数，以实现在线集群管理。

### 3. 公平调度器（Fair Scheduler），如图 1-7 所示



同队列所有任务共享资源，在时间尺度上获得公平的资源

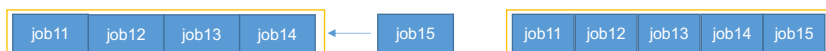


- 支持多队列多作业，每个队列可以单独配置
- 同一队列的作业按照其优先级分享整个队列的资源，并发执行
- 每个作业可以设置最小资源值，调度器会保证作业获得其以上的资源

让天下没有难学的技术



理想：



现实：



- 公平调度器设计目标是：在时间尺度上，所有作业获得公平的资源。某一时刻一个作业应获资源和实际获取资源的差距叫“缺额”
- 调度器会优先为缺额大的作业分配资源

让天下没有难学的技术

图 1-7 公平调度器

Fair Scheduler Fair Scheduler 是 Facebook 开发的多用户调度器。

公平调度器的目的是让所有的作业随着时间的推移，都能平均地获取等同的共享资源！当有作业提交上来，系统会将空闲的资源分配给新的作业！每个任务大致上会获取平等数量的资源！和传统的调度策略不同的是



它会让小的任务在合理的时间完成，同时不会让需要长时间运行的耗费大量资源的应用挨饿！

同 Capacity Scheduler 类似，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用；当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。

当然，Fair Scheduler 也存在很多与 Capacity Scheduler 不同之处，这主要体现在以下几个方面：

- ① 资源公平共享。在每个队列中，Fair Scheduler 可选择按照 FIFO、Fair 或 DRF 策略为应用程序分配资源。其中，Fair 策略(默认)是一种基于最大最小公平算法实现的资源多路复用方式，默认情况下，每个队列内部采用该方式分配资源。这意味着，如果一个队列中有两个应用程序同时运行，则每个应用程序可得到  $1/2$  的资源；如果三个应用程序同时运行，则每个应用程序可得到  $1/3$  的资源。

### 最大最小算法

#### 不加权

有一四个用户的集合,资源需求分别是 2, 2.6, 4, 5, 其资源总能力为 10, 为其计算最大最小公平分配

解决方法:

我们通过几轮的计算来计算最大最小公平分配.

第一轮,我们暂时将资源划分成 4 个大小为 2.5 的.由于这超过了用户 1 的需求,这使得剩了 0.5 个均匀的分配给剩下的 3 个人资源,给予他们每个 2.66.这又超过了用户 2 的需求,所以我们拥有额外的 0.066...来分配给剩下的两个用户,给予每个用户  $2.5 + 0.66... + 0.033... = 2.7$ .因此公平分配是:用户 1 得到 2, 用户 2 得到 2.6, 用户 3 和用户 4 每个都得到 2.7.

#### 加权

有一四个用户的集合,资源需求分别是 4, 2, 10, 4, 权重分别是 2.5, 4, 0.5, 1, 资源总能力是 16, 为其计算最大最小公平分配.

解决方法:

第一步是标准化权重,将最小的权重设置为 1.这样权重集合更新为 5, 8, 1, 2.这样需要的资源是  $5 + 8 + 1 + 2 = 16$  份.因此将资源划分成 16 份.在资源分配的每一轮,我们按照权重的比例来划分资源,因此,在第一轮,,用户分别获得 5, 8, 1, 2 单元的资源,用户 1 得到了 5 个资源,但是只需要 4, 所以多了 1 个资源,同样的,用户 2 多了 6 个资源.用户 3 和用户 4 拖欠了,因为他们的配额低于



需求.现在我们有 7 个资源可以分配给用户 3 和用户 4.他们的权重分别是 1 和 2,给予用户 3 额外的  $7 \times 1/3$  资源和用户 4 额外的  $7 \times 2/3$  资源.这会导致用户 4 的配额达到了  $2 + 7 \times 2/3 = 6.666$ ,超过了需求.所以我们将额外的 2.666 单元给用户 3,最终获得  $1 + 7/3 + 2.666 = 6$  单元.最终的分配是,4,2,6,4,这就是带权重的最大最小公平分配.

## DRF

DRF(Dominant Resource Fairness)。我们之前说的资源，都是单一标准，例如只考虑内存(也是 yarn 默认的情况)。但是很多时候我们资源有很多种，例如内存，CPU，网络带宽等，这样我们很难衡量两个应用应用应该分配的资源比例

那么在 YARN 中，我们用 DRF 来决定如何调度：假设集群有 10T mem 和 100 CPU，而应用 A 需要(2 CPU, 300GB)，应用 B 需要(6 CPU, 100GB)。则两种应用分别需要系统(2%, 3%)和(6%, 1%)的资源，这就意味着 A 是 mem 主导的, B 是 CPU 主导的，并且它俩的比例是 3% : 6% = 1:2。这样两者就按照 1:2 的比例去分配资源。

②支持资源抢占。当某个队列中有剩余资源时，调度器会将这些资源共享给其他队列，而当该队列中有新的应用程序提交时，调度器要为其回收资源。为了尽可能降低不必要的计算浪费，调度器采用了先等待再强制回收的策略，即如果等待一段时间后尚有未归还的资源，则会进行资源抢占：从那些超额使用资源的队列中杀死一部分任务，进而释放资源。

`yarn.scheduler.fair.preemption=true` 通过该配置开启资源抢占。

③负载均衡。Fair Scheduler 提供了一个基于任务数目的负载均衡机制，该机制尽可能将系统中的任务均匀分配到各个节点上。此外，用户也可以根据自己的需要设计负载均衡机制。

④调度策略配置灵活。Fair Scheduler 允许管理员为每个队列单独设置调度策略（当前支持 FIFO、Fair 或 DRF 三种）。

⑤提高小应用程序响应时间。由于采用了最大最小公平算法，小作业可以快速获取资源并运行完成

## 1.5 容量调度器多队列提交案例

### 1.5.1 需求

Yarn 默认的容量调度器是一条单队列的调度器，在实际使用中会出现单个任务阻塞整

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

个队列的情况。同时，随着业务的增长，公司需要分业务限制集群使用率。这就需要我们按照业务种类配置多条任务队列。

### 1.5.2 配置多队列的容量调度器

默认 Yarn 的配置下，容量调度器只有一条 Default 队列。在 capacity-scheduler.xml 中可以配置多条队列，并降低 default 队列资源占比：

```
<!-- 指定多队列 -->
<property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>default,hive</value>
</property>
<!-- 指定 default 队列的额定容量 -->
<property>
    <name>yarn.scheduler.capacity.root.default.capacity</name>
    <value>40</value>
</property>
<!-- 指定 hive 队列的额定容量 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.capacity</name>
    <value>60</value>
</property>
<!-- 指定 default 队列允许单用户占用的资源占比 -->
<property>
    <name>yarn.scheduler.capacity.root.default.user-limit-factor</name>
    <value>1</value>
</property>
<!-- 指定 hive 队列允许单用户占用的资源占比 -->
<property>
    <name>yarn.scheduler.capacity.root.hive.user-limit-factor</name>
    <value>1</value>
</property>
<!-- 指定 default 队列的最大容量-->
<property>
    <name>yarn.scheduler.capacity.root.default.maximum-capacity</name>
    <value>60</value>
</property>
<!-- 指定 hive 队列的最大容量-->
<property>
    <name>yarn.scheduler.capacity.root.hive.maximum-capacity</name>
    <value>80</value>
</property>
<!-- 指定 default 队列的状态 -->
<property>
    <name>yarn.scheduler.capacity.root.default.state</name>
```

```
<value>RUNNING</value>
</property>
<!-- 指定 hive 队列的状态 -->
<property>
  <name>yarn.scheduler.capacity.root.hive.state</name>
  <value>RUNNING</value>
</property>
<!-- 指定 default 队列允许哪些用户提交 job-->
<property>
  <name>yarn.scheduler.capacity.root.default.acl_submit_applications</name>
  <value>*</value>
</property>
<!-- 指定 hive 队列允许哪些用户提交 job-->
<property>
  <name>yarn.scheduler.capacity.root.hive.acl_submit_applications</name>
  <value>*</value>
</property>
<!-- 指定 default 队列允许哪些用户进行管理-->
<property>
  <name>yarn.scheduler.capacity.root.default.acl_administer_queue</name>
  <value>*</value>
</property>
<!-- 指定 hive 队列允许哪些用户进行管理-->
<property>
  <name>yarn.scheduler.capacity.root.hive.acl_administer_queue</name>
  <value>*</value>
</property>
<!-- 指定 default 队列允许哪些用户提交配置优先级的 job-->
<property>
<name>yarn.scheduler.capacity.root.default.acl_application_max_priority</name>
  <value>*</value>
</property>
<!--指定 hive 队列允许哪些用户提交配置优先级的 job -->


<property>
  <name>yarn.scheduler.capacity.root.hive.acl_application_max_priority</name>
  <value>*</value>
</property>
<!-- 指定 default 队列允许 job 运行的最大时间-->
<property>
  <name>yarn.scheduler.capacity.root.default.maximum-application-lifetime
  </name>
  <value>-1</value>
</property>
```

```

<!-- 指定 hive 队列允许 job 运行的最大时间-->
<property>
  <name>yarn.scheduler.capacity.root.hive.maximum-application-lifetime
  </name>
  <value>-1</value>
</property>
<!-- 指定 default 队列允许 job 运行的默认时间-->
<property>
  <name>yarn.scheduler.capacity.root.default.default-application-lifetime
  </name>
  <value>-1</value>
</property>
<!-- 指定 hive 队列允许 job 运行的最大时间-->
<property>
  <name>yarn.scheduler.capacity.root.hive.default-application-lifetime
  </name>
  <value>-1</value>
</property>

```

在配置完成后，重启 Yarn，就可以看到两条队列：



## NEW, NEW\_SAVING, SUBMITTED

Cluster

- About
- Nodes
- Node Labels
- Applications
- NEW
- NEW\_SAVING
- SUBMITTED
- ACCEPTED
- RUNNING
- FINISHED
- FAILED
- KILLED
- Scheduler**

Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Free
0	0	0	0	0	0 B	24 GB	0 B

Scheduler Metrics

Scheduler Type	Scheduling Resource Type
Capacity Scheduler	[MEMORY]

**Application Queues**

Legend: Capacity Used Used (over capacity) Max Capacity

- [-] root
  - [+] default
  - [+] hive

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime
Showing 0 to 0 of 0 entries					

图 1-8 多队列

### 1.5.3 向 Hive 队列提交任务

默认的任务提交都是提交到 default 队列的。如果希望向其他队列提交任务，需要在 Driver 中声明：

```

public class WcDriver {
    public static void main(String[] args) throws IOException, ClassNotFoundException,
        InterruptedException {
        Configuration configuration = new Configuration();

```

```
configuration.set("mapred.job.queue.name", "hive");

//1. 获取一个 Job 实例
Job job = Job.getInstance(configuration);

//2. 设置类路径
job.setJarByClass(WcDriver.class);

//3. 设置 Mapper 和 Reducer
job.setMapperClass(WcMapper.class);
job.setReducerClass(WcReducer.class);

//4. 设置 Mapper 和 Reducer 的输出类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

job.setCombinerClass(WcReducer.class);

//5. 设置输入输出文件
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

//6. 提交 Job
boolean b = job.waitForCompletion(true);
System.exit(b ? 0 : 1);
}
}
```

这样，这个任务在集群提交时，就会提交到 hive 队列：

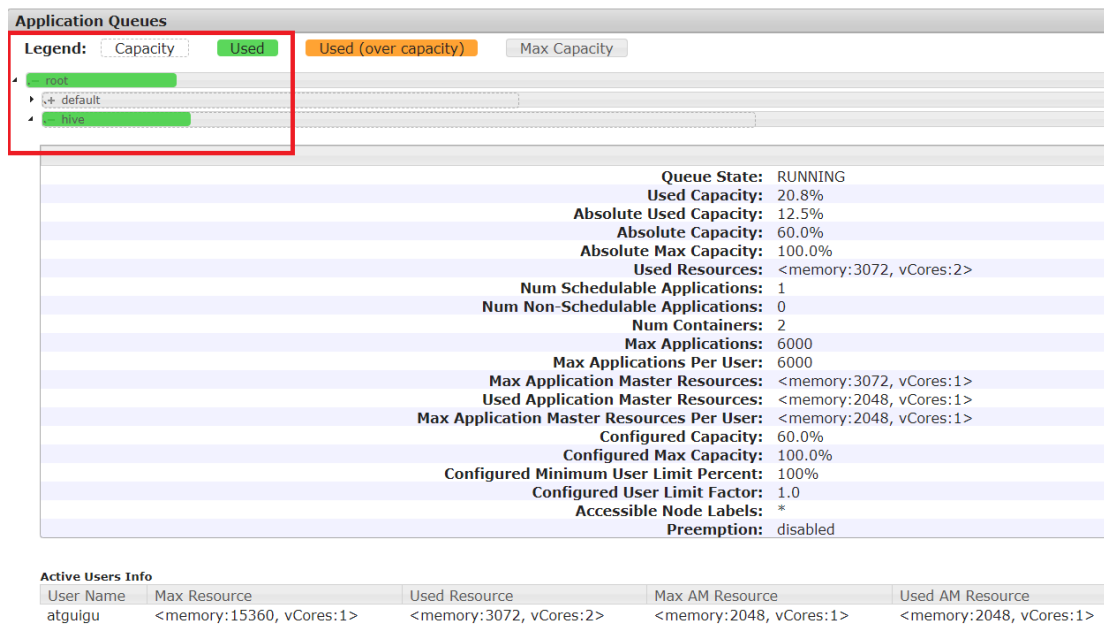


图 1-9 公平调度器

## 1.6 任务的推测执行

### 1. 作业完成时间取决于最慢的任务完成时间

一个作业由若干个 Map 任务和 Reduce 任务构成。因硬件老化、软件 Bug 等，某些任务可能运行非常慢。

思考：系统中有 99% 的 Map 任务都完成了，只有少数几个 Map 老是进度很慢，完不成，怎么办？

### 2. 推测执行机制

发现拖后腿的任务，比如某个任务运行速度远慢于任务平均速度。为拖后腿任务启动一个备份任务，同时运行。谁先运行完，则采用谁的结果。

### 3. 执行推测任务的前提条件

- (1) 每个 Task 只能有一个备份任务
- (2) 当前 Job 已完成的 Task 必须不小于 0.05 (5%)
- (3) 开启推测执行参数设置。mapred-site.xml 文件中默认是打开的。

```
<property>
  <name>mapreduce.map.speculative</name>
  <value>true</value>
  <description>If true, then multiple instances of some map
tasks may be executed in parallel.</description>
</property>
```

```
<property>
  <name>mapreduce.reduce.speculative</name>
  <value>true</value>
  <description>If true, then multiple instances of some reduce
tasks may be executed in parallel.</description>
</property>
```

#### 4. 不能启用推测执行机制情况

- (1) 任务间存在严重的负载倾斜;
- (2) 特殊任务, 比如任务向数据库中写数据。

#### 5. 算法原理, 如图 1-10 所示



#### 推测执行算法原理



假设某一时刻, 任务T的执行进度为progress, 则可通过一定的算法推测出该任务的最终完成时刻estimateEndTime。另一方面, 如果此刻为该任务启动一个备份任务, 则可推断出它可能的完成时刻estimateEndTime', 于是可得出以下几个公式:

$$\begin{aligned} \text{estimatedRunTime} &= (\text{currentTimestamp} - \text{taskStartTime}) / \text{progress} \\ \text{推测运行时间 (60s)} &= (\text{当前时刻 (6)} - \text{任务启动时刻 (0)}) / \text{任务运行比例 (10\%)} \\ \text{estimateEndTime} &= \text{estimatedRunTime} + \text{taskStartTime} \\ \text{推测执行完时刻 60} &= \text{推测运行时间 (60s)} + \text{任务启动时刻 (0)} \\ \text{estimateEndTime'} &= \text{currentTimestamp} + \text{averageRunTime} \\ \text{备份任务推测完成时刻 (16)} &= \text{当前时刻 (6)} + \text{运行完成任务的平均时间 (10s)} \end{aligned}$$

- 1) MR总是选择 ( estimateEndTime- estimateEndTime' ) 差值最大的任务, 并为之启动备份任务。
- 2) 为了防止大量任务同时启动备份任务造成的资源浪费, MR为每个作业设置了同时启动的备份任务数目上限。
- 3) 推测执行机制实际上采用了经典的优化算法: 以空间换时间, 它同时启动多个相同任务处理相同的数据, 并让这些任务竞争以缩短数据处理时间。显然, 这种方法需要占用更多的计算资源。在集群资源紧缺的情况下, 应合理使用该机制, 争取在多用少量资源的情况下, 减少作业的计算时间。

让天下没有难学的技术

图 1-10 推测执行算法原理