**睿普云资源调度系统**

**设计说明书**

**一、引言**

**目的**

资源调度系统为睿普云系统进行物理资源调度，当用户进行容器创建，存储创建等行为是，为用户从物理资源池中选择合适的资源进行创建 。

**二、系统总体设计**

**2.1需求概括**

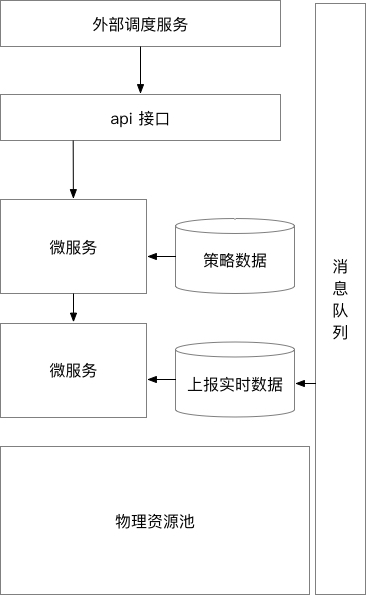
本软件采用传统的软件开发生命周期的方法，采用自顶向下，逐步求精的结构化的软件设计方法。

本软系统要有以下几方面的功能

1. 物理资源调度
2. 默认物理资源调度

# 总体设计

## 2.2总体结构和模块接口设计



系统整体结构框架图

## 2.5模块功能逻辑关系

系统详细的模块信息所示：

系统详细的模块信息表

|  |  |
| --- | --- |
| **主模块** | |
| 模块名称 | 功能简述 |
| 物理资源调度 | 根据调度策略和算法，进行物理资源分配 |
| 默认物理资源调度 | 根据调度策略和算法，进行存储资源分配 |

**2.6设计和描述**

资源调度系统按照固定算法，进行资源计算。计算公式如下：

物理资源权重=cpu 资源 \* 权重 + 内存资源\*权重 – 数量资源\*权重

所有物理资源按照权重排序，优先选择权重高的资源。计算所有资源的之后，从数据库读取策略数据，对资源进行策略分组，按照策略选择最优资源。例如有三台物理机 A，B，C ，其中 A 物理机cpu 资源 C1，B 物理机cpu 资源 C2，C 物理机资源 C3，A 物理机内存资源 M1，B 物理机内存资源 M2，C 物理机内存资源 M3，数量资源是这个物理机上已经创建了多少容器了，A 物理机数量资源 N1，B 物理机数量资源N2，C 物理机数量资源 N3。则 ABC 三台机器的权重分别为：

A 机器权重为：W1 = C1\*w1+M1\*w2+N1\*w3

B 机器权重为：W2 = C2\*w1+M2\*w2+N2\*w3

C 机器权重为：W3 = C3\*w1+M3\*w2+N3\*w3

对 W1,W2,W3进行排序，然后根据策略数据，排除不合适的机器，选择出最优资源。

## **调度器设计概述**

在系统设计领域研究比较多的朋友可以容易地得出一个结论，那就是我们的系统设计——无论是小到一个嵌入式的系统， 还是大到好几百个机器的集群，在设计抽象上都是在不同的层次上重复自己。比如说，如果我们着眼于一个 CPU， 他包括计算单元和一系列的用来加速数据访问的缓存 L1、L2、L3 等，每种缓存具有不同的访问速度。 当我们的视野扩大到整个机器，CPU 又可以被当成一个单元，我们又有内存和硬盘两个层次的储存系统用于加速数据载入。 而在分布式系统中，如果我们把 HDFS 或 S3 看作硬盘，也存在像 Alluxio 这样发挥着类似内存作用的系统。

既然系统在设计上的基本原则都是类似的，那为什么大规模分布式系统的设计这么困难呢？ 这是因为当问题的规模变化了，原先不显著或者容易解决的问题可能会变的难以解决。举例来说， 当我们谈论起进程间通信或者同一个 CPU 不同内核之间的通信时，我们往往不考虑通讯不稳定所带来的问题: 我们无法想象如果一个 CPU 内核无法发送消息到另一个内核的状况。 然而在通过网络通讯的多机机群当中这是无法回避的问题，Paxos、Raft、Zab 等算法被设计出来的原因也在于此。

我们先看一看单机操作系统调度器的发展路线：

1. 最早的调度器是批处理调度器，这种调度器批量调度和执行任务，通过对计算机资源的分时复用来增加资源的利用率。 一般具有较高的吞吐量

2. 某些与外界交互次数频繁的系统对响应时间具有较强的要求，因此发展出了实时操作系统。 实时操作系统的调度器具备低延迟相应外部信号的能力

3. 我们常用的操作系统基本上以批处理的方式调度任务，又通过中断等机制提供实时性的保证， 通过提供灵活的调度策略，在吞吐量和延迟时间当中获得平衡

在分布式系统中的调度器的设计也是相同的。从单机调度器这里我们首先可以总结出调度器设计的三个最基本的需求：

1. 资源的有效利用

2. 信号的实时响应

3. 调度策略的灵活配置

这三个需求在某种程度上来说是相互矛盾的，在面对不同需求的时候需要做出不一样的取舍。

在上述三个需求的基础上，分布式的调度器设计还需要克服很多其他的困难。这些困难往往在单机系统当中并不显著。 比如说：

1. 状态的同步问题。在单机系统中，我们一般使用常规的同步方法，如共享内存和锁机制，就可以很好地保证任务的协调运行了。 这是因为在单机系统上的状态同步比较稳定和容易。然而在分布式系统中，因为网络通讯的不确定性， 使机群中的各个机器对于周围的状态达成一致是非常困难的任务。实际上，在分布式系统中甚至无法通过网络精确同步所有机器的时间！

2. 容错性问题。由于单机系统的处理能力有限，我们运行任务的规模和同时运行任务的数量都比较有限。 出错的概率和成本都比较低。但是在分布式系统中，由于任务规模变大、任务依赖关系变得更加复杂， 出错的概率大大增加，错误恢复的成本可能也比较高，因此可能需要调度器快速地识别错误并进行恢复操作。

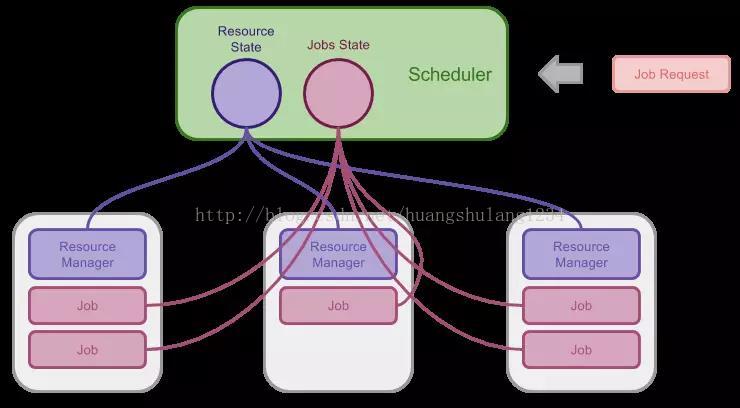
3. 可扩展性的问题。当分布式系统的规模到达一定程度，调度器的可扩展性就可能会成为瓶颈。为了提供高可扩展性， 调度器不但要可以应对管理上千台机器的挑战，也要能够处理动态增减节点这样的问题。

现阶段比较流行的分布式调度器可以归纳为三种类型，在接下来的文章里将会结合具体的案例进行介绍。

## 

## **集中式调度器**

集中式(Centralized)调度器也可以被称为宏(Monolithic)调度器。指的是使用中心化的方式管理资源和调度任务。 也就是说，调度器本身在系统中只存在单个实例，所有的资源请求和任务调度都通过这一个实例进行。 下图展示了集中式调度器的一般模型。可以看到，在这一模型中，资源的使用状态和任务的执行状态都由中央调度器管理。



Centralized Scheduler

按照上面的思路，可以列出集中式调度器在各个方面的表现状况：

1. 适合批处理任务和吞吐量较大、运行时间较长的任务

2. 调度算法只能全部内置在核心调度器当中，灵活性和策略的可扩展性不高

3. 状态同步比较容易且稳定，这是因为资源使用和任务执行的状态被统一管理，降低了状态同步和并发控制的难度

4. 由于存在单点故障的可能性，集中式调度器的容错性一般，有些系统通过热备份 Master 的方式提高可用性

5. 由于所有的资源和任务请求都要由中央调度器处理，集中式调度器的可扩展性较差，容易成为分布式系统吞吐量的瓶颈

尽管应用场合比较局限，集中式调度器仍然是普遍使用的调度器，可以被广泛应用于中小规模的数据平台应用。

### 

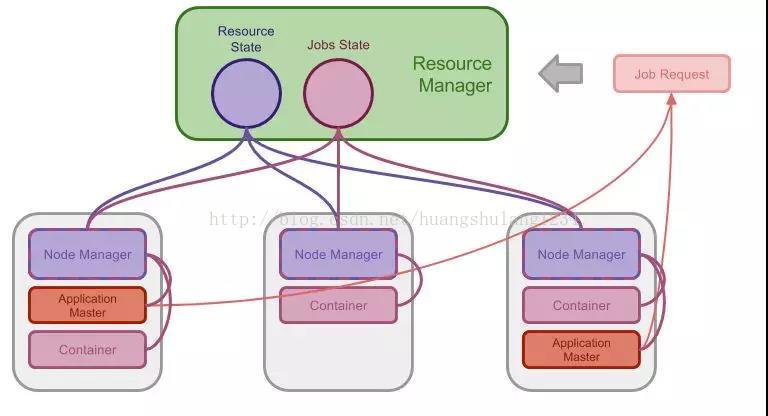
### **集中式调度器案例1: 单机操作系统的调度器**

单机操作系统，如 Windows、Linux 和 macOS 的进程调度器是典型的集中式调度器。 当用户请求执行应用之后，由操作系统将进程载入内存。计算机硬件的所有资源，包括 CPU、内存和硬盘等都由操作系统集中式管理。 当进程需要时，通过系统调用请求操作系统分配资源。如果单机环境中的进程使用了系统级多线程， 这些线程的调度也由系统一并控制。

### 

### **集中式调度器案例2: Hadoop YARN**

对于集中式调度器，我们重点介绍 Hadoop YARN 这个案例。下图是将集中式调度器的一般模型替换成 YARN 当中术语之后的示意图：

Hadoop YARN

Hadoop YARN 的特点可以总结为：

1. 集中式的资源管理和调度器被称为 ResourceManager，所有的资源的空闲和使用情况都由 ResourceManager 管理， ResourceManager 也负责监控任务的执行

2. 集群中的每个节点都运行着一个 NodeManager，这个 NodeManager 管理本地的资源占用和任务执行并将这些状态同步给 ResourceManager

3. 集群中的任务运行在 Container 当中，YARN 使用 Container 作为资源分配的抽象单位，每个 Container 会被分配一些本地资源和运算资源等

熟悉 YARN 的朋友可能知道 YARN 存在一个 ApplicationMaster 的概念，当一个应用被启动之后， 一个 ApplicationMaster 会先在集群中被启动，随后 ApplicationMaster 会向 ResourceMaster 申请新的资源并调度新的任务。这一模型好像看起来和后面介绍的双层调度器特别是 Mesos 的设计有点相似， 但一般仍认为 YARN 是 Monolithic 设计的调度器。这主要是因为：

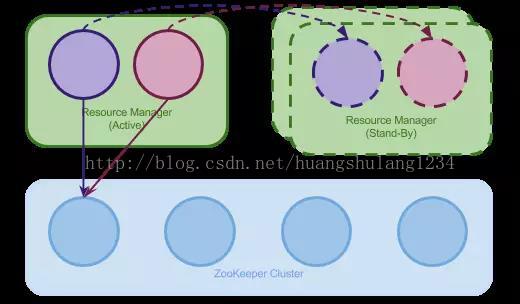
1. ApplicationMaster 其实也是运行在一个 Container 里的 YARN job

2. ApplicationMaster 虽然决定 Job 如何被激发，但是仍然需要请求 ResourceMaster 申请资源和启动新的 Job

3. ApplicationMaster 启动的 Job 也会由 ResourceMaster 进行监控，其启动所需的本地资源和运算资源都由 ResourceMaster 负责分配并通知 NodeManager 具体执行

在 YARN 中，为了防止 ResourceManager 出错退出，可以设计多个 Stand-By Master， Stand-By Master 一直处于运行状态并和 ResourceManager 注册在同一个 ZooKeeper 集群中。

Active 的 ResourceManager 会定期保存自己的状态到 ZooKeeper，当其失败退出后， 一个 Stand-By Master 会被选举出来成为新的 Manager。



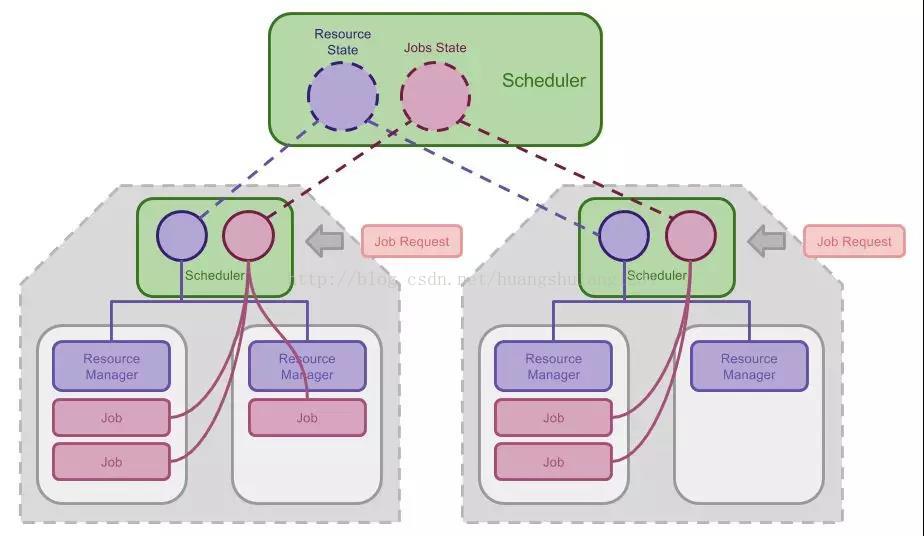
adoop YARN: High Availability

作为一个分布式资源管理和调度器， YARN 与其竞争对手相比功能其实比较薄弱。 譬如默认情况下 YARN 只能对 Memory 资源施加限制(如果一个 Job 使用了超过许可的 Memory， YARN 会直接杀死进程)。尽管其调度接口提供了对 CPU Cores 的抽象， 但 YARN 默认情况下对任务使用 CPU 核数并没有任何限制。

不过若运行在 Linux 环境下， 在较新版本的 YARN 中可以配置 cgroup 限制资源使用。

## 双层调度器

前面提到，集中式调度器的主要缺点在于单点模型容错性和可扩展性较差，容易成为性能瓶颈。在一般的数据密集型应用当中， 解决这一问题的主要方法是分区。下图是双层调度器的一般模型：

2-level Scheduler

在双层调度器当中，资源的使用状态同时由分区调度器和中央调度器管理，但是中央调度器一般只负责宏观的大规模的资源分配， 因此业务压力较小。分区调度器负责管理自己分区的所有资源和任务，一般只有当所在分区资源无法满足需求时， 才将任务冒泡到中央调度器处理。

相比集中式调度器，双层调度器某一分区内的资源分配和工作安排可以由具体的任务本身进行定制， 因此大大增强了使用的灵活性，可以同时对高吞吐和低延迟的两种场景提供良好的支持。每个分区可以独立运行， 降低了单点故障导致系统崩溃的概率，增加了可用性和可扩展性。但是反过来也导致状态同步和维护变得比较困难。

尽管主要思路是一致的，但双层调度器在实现上的变种比较丰富，本文接下来使用案例进行介绍。

### 

### **双层调度器案例1: 协程调度器**

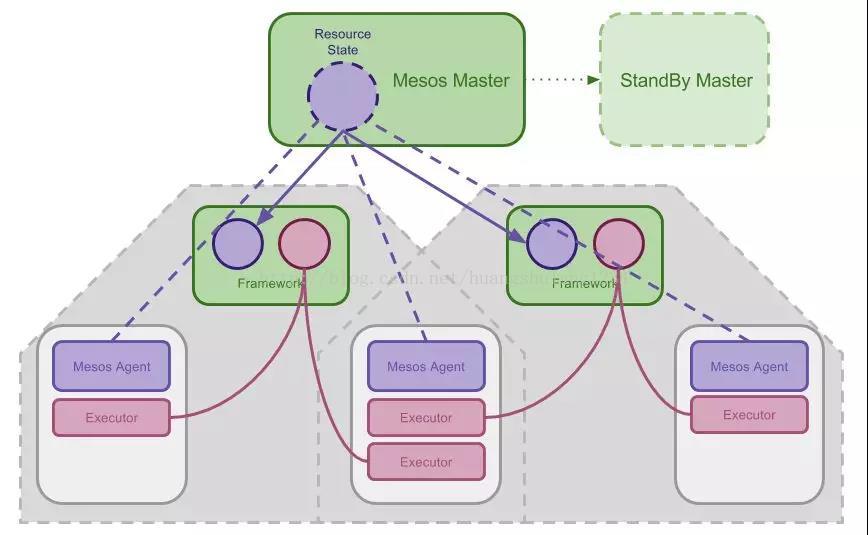
单机操作系统的单个进程为了避免系统级多线程上下文切换的成本，可以自行实现进程内的调度器，如 Golang 运行时的 Goroutine 调度器。在这一模型下，一个进程内部的资源就相当于一个分区，分区内的资源由运行时提供的调度器预先申请并自行管理。 运行时环境只有当资源耗尽时才会向系统请求新的资源，从而避免频繁的系统调用。

提出这个例子的主要目的在于说明类似的优化思路其实也被应用于分布式系统，再次证明了系统设计分层重复的特点！

### 

### **双层调度器案例2: Mesos**

Mesos 是和 YARN 几乎同一时间发展起来的任务和资源调度系统。这一调度系统实现了完整的资源调度功能， 并使用 Linux Container 技术对资源的使用进行限制。和 YARN 一样，Mesos 系统也包括一个独立的 Mesos Master 和运行在每个节点上的 Mesos Agent，而后者会管理节点上的资源和任务并将状态同步给 Master。 在 Mesos 里，任务运行在 Executor 里。下图是 Mesos 的主要架构

Mesos Scheduler

值得注意的是，Mesos 分区的单位并不是单个节点，是可以将一个节点当中的资源划分到多个区的。 也就是说，在 Mesos 里，分区是逻辑的和动态的。

把 Mesos 看作一种双层的资源调度系统设计主要基于以下几点：

1. 与一般通过 Master 请求资源不同，Mesos 提出了 Framework 的概念，每个 Framework 相当于一个独立的调度器， 可以实现自己的调度策略

2. Master 掌握对整个集群资源的的状态，通过 Offer (而不是被动请求) 的方式通知每个 Framework 可用的资源有哪些

3. Framework 根据自己的需求决定要不要占有 Master Offer 的资源，如果占有了资源，这些资源接下来将完全由 Framework 管理

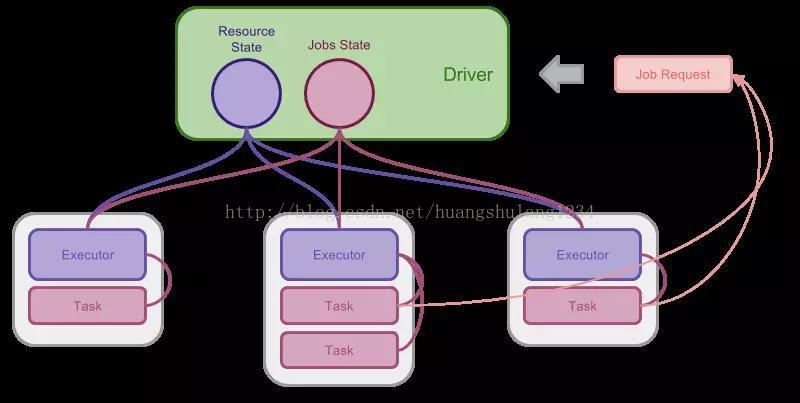
4. Framework 通过灵活分配自己占有的资源调度任务并执行，并不需要通过 Master 完成这一任务

同样，Mesos 也可以通过 Stand-By Master 的方法提供 Master 节点的高可用性。 Mesos 已经被广泛应用于各类集群的管理，但是其 Offer-Accept 的资源申请可能不是特别容易理解。 对于想要自行编写调度策略的人，Frameworks 的抽象比较并不容易掌握。由于 Framework 要先占有了资源才能使用， 设计不够良好的 Framework 可能会导致资源浪费和资源竞争/死锁。

### 

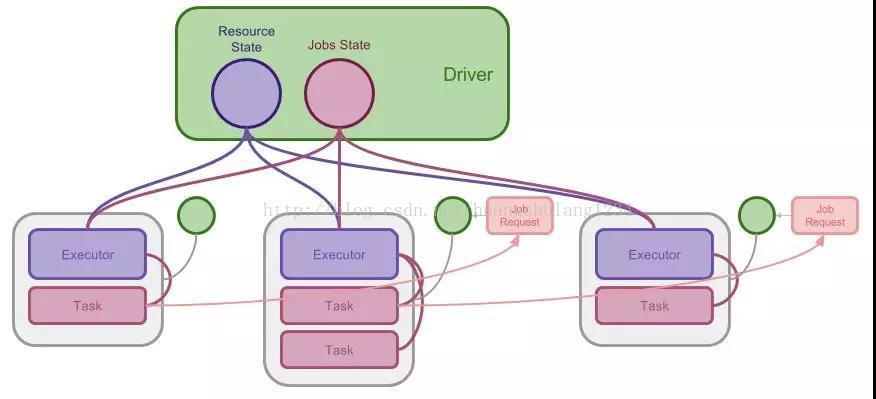
### **双层调度器案例3：Spark 和 Spark Drizzle**

Spark 为了调度和执行自己基于 DAG 模型的计算，自己实现了一个集中式的调度器， 这个调度器的 Master 被称为 Driver，当 Driver 运行起来的之后，会向上层的 Scheduler 申请资源调度起 Executor 进程。Executor 将会一直保持待机，等候 Driver 分配任务并执行，直到任务结束为止。

Spark 传统的调度模型

Spark 和 YARN 这样的集中式调度器放在一起可以认为是通过迂回的方式实现了双层调度器。 就好像单机进程自己实现协程调度器一样，Spark Driver 预先申请的资源可以认为是在申请分区资源， 申请到的资源将由 Driver 自行管理和使用。

有趣的是，在 2017 年的 SOSP 上，Spark 为了解决流处理计算当中调度延迟较大的问题， 提出了一种新的调度模型 Drizzle，在原来调度模型的基础上，又再次实现了双层调度。 下图是 Spark Drizzle 的设计模型

Spark Drizzle 的调度模型

Drizzle 使得调度 Spark Streaming 任务的延迟由最低 500ms 降低到 200ms 左右，让 Spark Streaming 在低延迟处理的问题上获得了突破性的进展。要搞清楚 Drizzle 提出的目的和解决的问题， 首先要理解以下几点：

1. Spark Streaming 的实现方式实际上是 Micro Batch，也就是说流式输入的数据在这里仍然被切分成一个一个 Batch 进行处理

2. 传统的 Spark 调度器会在前序任务完成之后，根据之前任务输出的规模和分布，通过一定的算法有策略地调度新的任务， 以便于获得更好的处理速度和降低资源浪费

3. 在这一过程中，Exector 需要在前续任务完成后通知 Scheduler，之后由 Scheduler 调度新的任务。 在传统 Batch 处理模式下，这种模型效果很好，但是在 Streaming 的场景下存在很多问题

4. 在 Streaming 场景下，每个 Batch 的数据量较小，因此任务可能会需要频繁与 Scheduler 交互， 因为存在这一交互过程的 Overhead，Streaming 处理的过程中最低的延迟也要 500ms 以上

为了得到更低的延迟性且保留 Micro Batch 容错性强且易于执行 Checkpoint 的优点，Drizzle 在原来的模型上做了一些优化：

1. 在每个节点运行一个 LocalScheduler

2. 中央调度器 Driver 在执行 Streaming 处理任务时，根据计算的 DAG 图模型，预先调度某一个 Job 的后序 Job，后序 Job 会被放置在 LocalScheduler 上

3. 后序 Job 默认在 LocalScheduler 上是沉睡状态，但是前面的 Job 可以知道后序 Job 在哪个节点上， 因此当前面的任务完成后，可以直接激活后序任务

4. 当后序任务被激活之后，前序任务和后序任务可以直接通过网络请求串流结果

可以看到 Drizzle 的主要思路就是根据用户程序生成的图模型，预先 Schedule 一些任务， 使得前序任务知道后序任务的位置，在调度时避免再请求中央调度器 Driver。 同时 Drizzle 也采取了其他一些方法，比如将多个 Micro Batch 打包在一起，借由 LocalScheduler 自行本地调度等等方式减少延迟。

Drizzle 模型可以说是双层模型的又一种另类体现。然而这种模型主要的缺点是必须要预先知道计算任务的图模型和依赖关系， 否则就无法发挥作用。

## 

## **共享状态调度器**

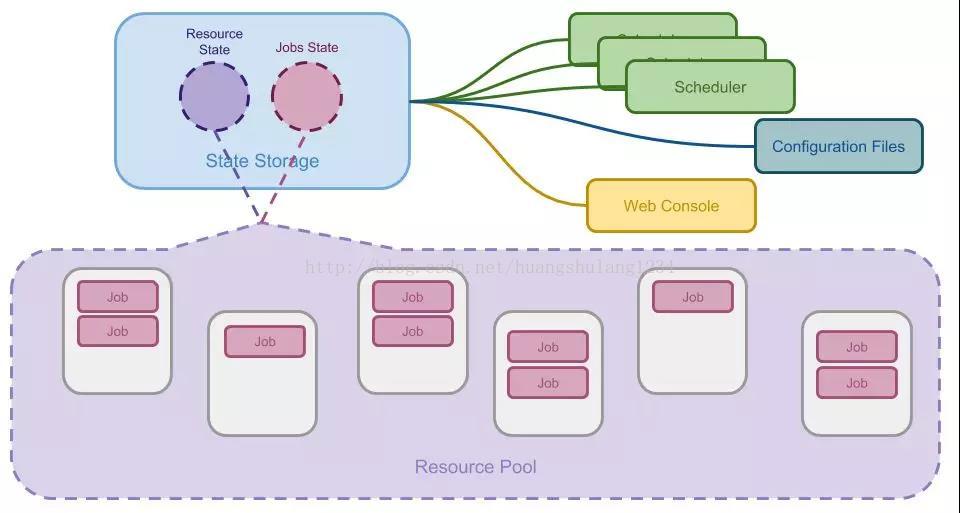
通过前面两种模型的介绍，可以发现集群中需要管理的状态主要包括以下两种：

1. 系统中资源分配和使用的状态

2. 系统中任务调度和执行的状态

在集中式调度器里，这两个状态都由中心调度器管理，并且一并集成了调度等功能。 双层调度器模式里，这两个状态分别由中央调度器和次级调度器管理。 集中式调度器可以容易地保证全局状态的一致性但是可扩展性不够， 双层调度器对共享状态的管理较难达到好的一致性保证，也不容易检测资源竞争和死锁。

为了解决这些问题，一种新的调度器架构被设计出来。 这种架构基本上沿袭了集中式调度器的模式，通过将中央调度器肢解为多个服务以提供更好的伸缩性。 这种调度器的核心是共享的集群状态，因此可以被称为共享状态调度器。

共享状态调度器

共享状态调度架构为了提供高可用性和可扩展性，将除共享状态之外的功能剥离出来成为独立的服务。 这种设计可以类比为单机操作系统的微内核设计。在这种设计中，内核只负责提供最核心的资源管理接口， 其他的内核功能都被实现为独立的服务，通过调用内核提供的 API 完成工作。

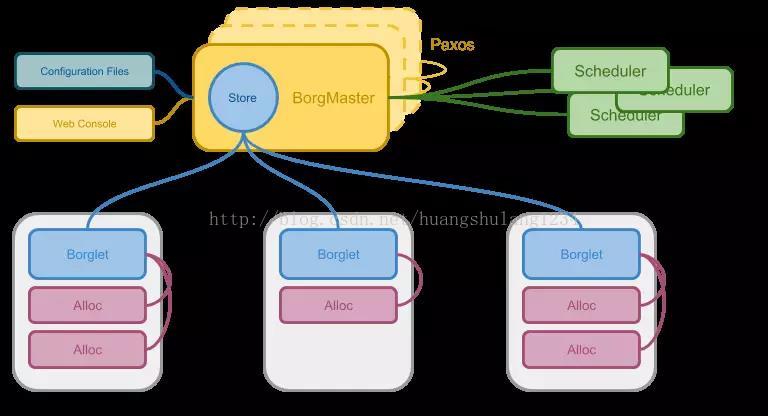
共享状态调度器的设计近些年来越来越受欢迎，这两年炙手可热的 Kubernetes 和它的原型 Borg 都是采用这种架构。最近由加州大学伯克利分校知名实验室 RISELab 提出的号称要取代 Spark 分布式计算系统 Ray 也是如此。下面将对这些案例进行介绍。

### 

### **共享状态调度器案例1: Borg/Kubernetes**

根据相关论文，Borg 在初期开发的时候使用的是集中式调度器的设计，所有功能都被集中在 BorgMaster 当中，之后随着对灵活性和可扩展性的要求，逐步切换到共享状态模型或者说微内核模型上面去。 Google 的工程师们总结了 Borg 的经验教训，将这些概念集合在 Kubernetes 当中开源出来， 成为了近些年来最炙手可热的资源管理框架。

在这里我们依然以 Borg 为例进行介绍，Kubernetes 在具体的设计上是与 Borg 基本一致的。 下图是 Borg 设计架构示意图：

Borg 资源调度架构

Borg 资源调度架构的设计可以总结为以下几点：

1. 一个数据中心的集群可以被组织成一个 Borg 当中的 Cell

2. 在一个 Borg 的 Cell 当中，资源的管理类似于集中式调度器的设计——集群资源由 BorgMaster 统一管理， 每一个节点上运行着 Borglet 定时将本机器的状态与 BorgMaster 同步

3. 为了增加可用性，BorgMaster 使用了 Stand-By Master 的模式。也就是说同时运行着 BorgMaster 的多个热备份， 当 Active 的 BorgMaster 出现失败，新的 Master 会被选取出来

4. 为了增加可扩展性和灵活性，BorgMaster 的大部分功能被剥离出来成为独立的服务。最终，BorgMaster 只剩下维护集群资源和任务状态这唯一一个功能，包括 Scheduler 在内的所有其他服务都独立运行

5. 独立运行的每个 Scheduler 可以运行自己的调度策略，它们定时从 BorgMaster 同步集群资源状态， 根据自己的需要做出修改，然后通过 API 同步回 BorgMaster，从而实现调度功能

可以看到，Borg 的共享状态调度架构其实是集中式调度的改进，由于承载调度逻辑的调度器都运行在独立的服务里， 对于 BorgMaster 的请求压力得到了某种程度的缓解。使用微内核设计模式，BorgMaster 自己包含的逻辑就比较简单了， 系统的鲁棒性、灵活性和可扩展性得到了增强。

在 Borg 中，任务的隔离和资源限制使用了 Linux 的 cgroup 机制。在 Kubernetes 当中，这一机制被 Container 技术替代， 实际上的功能是等价的。

Borg 的共享状态设计看似简单，其实具体实现仍然比较复杂。事实上，集中式的状态管理仍然会成为瓶颈。 随着集群规模的扩展和状态的规模扩大，State Storage 必须使用分布式数据储存机制来保证可用性和低延迟。

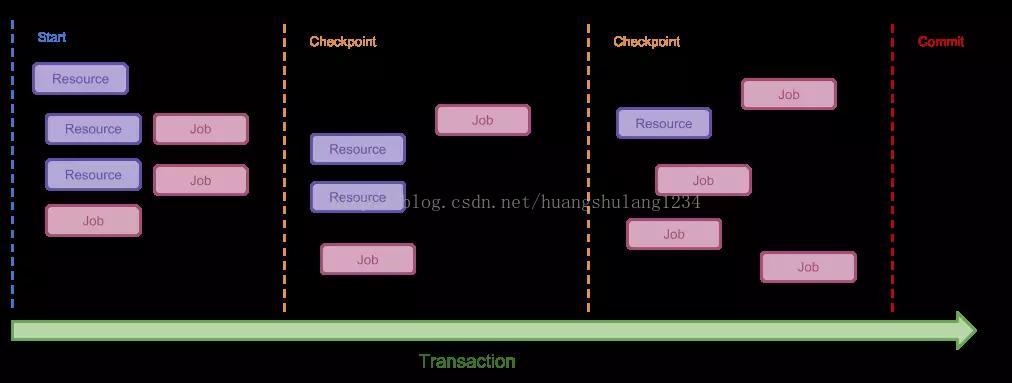
共享状态架构的设计和双层设计的最大区别是， 共享状态被抽取出来由一个统一的组件管理。从其他的各种服务的角度来看， 共享状态提供的调用接口和集中式调度的状态管理是一样的。 这种设计通过封装内部细节的方式降低了外部服务编写的复杂度，体现了系统设计里封装复杂模块的思想。

### 

### **共享状态调度器案例2：Omega**

上面介绍的 Borg 是共享状态最典型的一个示例。尽管 BorgMaster 已经为其他服务的编程提供了简单的接口， 但是仍然没有降低状态一致性同步的难度——BorgMaster 和服务的编写着仍然需要考虑很多并发控制的方法， 防止对共享状态的修改出现 Race Condition 或死锁的现象。如何为其他服务和调度策略提供一层简单的抽象， 使得任务的调度能兼顾吞吐量、延迟和并发安全呢？

Omega 使用事务(Transaction)解决共享状态一致性管理的问题。这一思路非常直观——如果将数据库储存的数据看作共享状态， 那么数据库就是是共享状态管理的最成熟、最通用的解决方案！事务更是早已被开发者们熟悉而且证明非常成熟和好用的并发抽象。

事务调度策略

Omega 将集群中资源的使用和任务的调度看作数据库中的条目，在一个应用执行的过程当中， 调度器可以分步请求多种资源，当所有资源依次被占用并使任务执行完成，这个 Transaction 就会被成功 Commit。

Omega 的设计借鉴了很多数据库设计的思路，比如：

1. Transaction 设计保留了一般事务的诸多特性，如嵌套 Transaction 或者 Checkpoint。 当资源无法获取或任务执行失败，事务将会被回滚到上一个 Checkpoint 那里

2. Omega 可以实现传统数据库的死锁检测机制，如果检测到死锁，可以安全地撤销一个任务或其中的一些步骤

3. Omega 使用了乐观锁，也就是说申请的资源不会立刻被加上排他锁，只有需要真正分配资源或进行事务提交的时候才会检查锁的状态， 如果发现出现了 Race Condition 或其他错误，相关事务可以被回滚

4. Omega 可以像主流数据库一样定义 Procedure ，这些 Procedure 可以实现一些简单的逻辑， 用于对用户的资源请求进行合法性的验证(如优先级保证、一致性校验、资源请求权限管理等)

**三、系统功能描述**

**3.1 物理资源详细流程描述**

物理资源调度主要是调用物理机资源用来创建容器。需要传入 cpu，内存，存储，集群等参数进行计算。这里介绍下策略数据：策略数据分为 CPU 指标，内存，硬盘，磁盘 IO，外网出带宽，外网如带宽等维度。用户可以定义指标。当超过指标时，则该物理资源不能满足分配条件。如下为策略的表结构：

CREATE TABLE `alerm\_policy\_condition` (

`id` varchar(32) NOT NULL,

`quota` int(11) DEFAULT NULL COMMENT '指标1、cpu 2、内存 3、硬盘 4、网络 5、磁盘io读 6、磁盘io写 7、外网出带宽 8、外网入带宽',

`op` int(11) DEFAULT NULL COMMENT '操作符1、= 2、> 3、<',

`threshold\_value` float DEFAULT NULL COMMENT '阈值',

`statistics\_cyc` int(11) DEFAULT NULL COMMENT '统计周期',

`last\_cyc` int(11) DEFAULT NULL COMMENT '持续周期',

`policy\_id` varchar(32) DEFAULT NULL COMMENT '策略id',

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='告警策略条件表'

为了提高性能，策略会保存在 redis 缓存中。

**3.2默认物理资源详细流程描述**

默认物理资源分配是计算存储分配的算法，这个算法没不需要提供 cpu 内存等参数进行计算，计算算法通物理资源分配。

# 四、接口设计

## 4.1接口列表

POST /scheduler/resourceSchedulerNullParam 默认资源调度

POST /scheduler/resourceScheduler 物理资源调度

# 4.2出错处理设计

## 出错输出信息

接口返回结果数据格式为JSON，返回的错误信息在最外层根据HTTP统一错误码进行判断对应结果如下：

200 - 服务器成功返回网页

404 - 请求的网页不存在

503 - 服务不可用

1xx（临时响应）表示临时响应并需要请求者继续执行操作的状态代码。

100 （继续） 请求者应当继续提出请求。服务器返回此代码表示已收到请求的第一部分，正在等待其余部分。

101 （切换协议） 请求者已要求服务器切换协议，服务器已确认并准备切换。

2xx （成功）

表示成功处理了请求的状态代码。

200 （成功） 服务器已成功处理了请求。通常，这表示服务器提供了请求的网页。

201 （已创建） 请求成功并且服务器创建了新的资源。

202 （已接受） 服务器已接受请求，但尚未处理。

203 （非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。

204 （无内容） 服务器成功处理了请求，但没有返回任何内容。

205 （重置内容） 服务器成功处理了请求，但没有返回任何内容。

206 （部分内容） 服务器成功处理了部分 GET 请求。

3xx （重定向）

表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向。

300 （多种选择） 针对请求，服务器可执行多种操作。服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。

301 （永久移动） 请求的网页已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。

302 （临时移动） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

303 （查看其他位置） 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。

304 （未修改） 自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。

305 （使用代理） 请求者只能使用代理访问请求的网页。如果服务器返回此响应，还表示请求者应使用代理。

307 （临时重定向） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

4xx（请求错误）

这些状态代码表示请求可能出错，妨碍了服务器的处理。

400 （错误请求） 服务器不理解请求的语法。

401 （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403 （禁止） 服务器拒绝请求。

404 （未找到） 服务器找不到请求的网页。

405 （方法禁用） 禁用请求中指定的方法。

406 （不接受） 无法使用请求的内容特性响应请求的网页。

407 （需要代理授权） 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理。

408 （请求超时） 服务器等候请求时发生超时。

409 （冲突） 服务器在完成请求时发生冲突。服务器必须在响应中包含有关冲突的信息。

410 （已删除） 如果请求的资源已永久删除，服务器就会返回此响应。

411 （需要有效长度） 服务器不接受不含有效内容长度标头字段的请求。

412 （未满足前提条件） 服务器未满足请求者在请求中设置的其中一个前提条件。

413 （请求实体过大） 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。

414 （请求的 URI 过长） 请求的 URI（通常为网址）过长，服务器无法处理。

415 （不支持的媒体类型） 请求的格式不受请求页面的支持。

416 （请求范围不符合要求） 如果页面无法提供请求的范围，则服务器会返回此状态代码。

417 （未满足期望值） 服务器未满足"期望"请求标头字段的要求。

5xx（服务器错误）

这些状态代码表示服务器在尝试处理请求时发生内部错误。 这些错误可能是服务器本身的错误，而不是请求出错。

500 （服务器内部错误） 服务器遇到错误，无法完成请求。

501 （尚未实施） 服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回此代码。

502 （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。

503 （服务不可用） 服务器目前无法使用（由于超载或停机维护）。通常，这只是暂时状态。

504 （网关超时） 服务器作为网关或代理，但是没有及时从上游服务器收到请求。

505 （HTTP 版本不受支持） 服务器不支持请求中所用的 HTTP 协议版本。

如果最外层返回码为200，其他业务错误类型在第二次中进行标识。