

---

# 分布式操作系统，PAAS的终极之路

---

## 前言

很多东西都不是突然被发明的，一定会有积累的过程，也就是说他依赖的一些关键技术需要得到解决。这好比燃烧，一定要到了燃点才能烧的起来。

比如，深度学习，很早之前就提出来了，但是现在才火起来，究其原因，是因为刚提出来的时候有两个硬性条件当时达不到

1. 计算能力不足(现在GPU都搞起来了，而且动则几千台服务器，单机服务器性能也提升不是一丁点)
2. 数据规模(如果数据太少，难以发现隐藏的关联关系)

随着大数据以及计算机计算能力的发展，条件具备了，才让现在的深度学习成为可能。

同样的，关于分布式操作系统，很早就被提出来了，但是却一直没有被实现，也没有被重视，原因也在此，以前的条件不具备，但是现在具备了。

现在国内有人在提‘数据分布式操作系统’，基于Mesos来实现的，完成了一套大数据栈的集成，为分布式操作系统卖出了比较坚实的一步。

事实上，构建一个分布式系统需要的前提条件现在都得到满足：

1. 大数据的发展使得很多分布式组件得到广泛应用和成熟
2. 容器技术的发展，使得资源隔离，环境打包得到极大的发展，分布式系统内部甚至可以异构(比如500台Centos,500台Ubuntu)
3. 资源调度系统的蓬勃发展，尤其以Yarn/Mesos 为代表

分布式操作系统由5层构成，类似传统的单机操作系统：

1. 交互层
  - 外部服务调用，类似单机的IP/Port定位，分布式操作系统通过封装 Nginx/Haproxy 实现。

- 人机交互界面，类似单机的Shell终端，通过Web化界面实现。

## 2. 应用层

- 支持.tar.gz, .image, .git 三种文件的安装
- 以Docker为进程模型进行运行
- 进程支持异步或者同步同步通讯，通过系统封装的Zookeeper API 来实现协调。

## 3. 系统服务层

- 基于Yarn实现一套安装程序，可以安装.tar.gz, .image, .git的安装包，动态资源分配。也可以使用传统的指定安装在特定标签的服务器上。
- 基于Yarn实现的一套动态调度系统。职责包括：特定服务的动态扩容/缩容，维持服务实例数(其实也是由动态扩容/缩容来实现)，并且将该变更反馈到交互层(Nginx/Haproxy)
- 基于Yarn实现的一套系统，实现MySQL,Redis这种单Master的高可用性，保证存储的稳定。而类似HBase,Cassandra这些服务本身已经实现了高可用性，不需要分布式系统的系统服务来支撑。
- Kafka消息队列,为应用层提供异步通讯机制

## 4. 文件系统

- HDFS分布式文件系统
- MySQL高可用支持事务的存储引擎
- Redis高可用缓存组件
- HBase高可用KV存储组件

## 5. 内核层(Borg)

- 集成分布式Shell引擎
- 集成Yarn资源管理引擎
- 典型Master-Slave 结构

具体对比如下：

以Yarn为内核的分布式系统

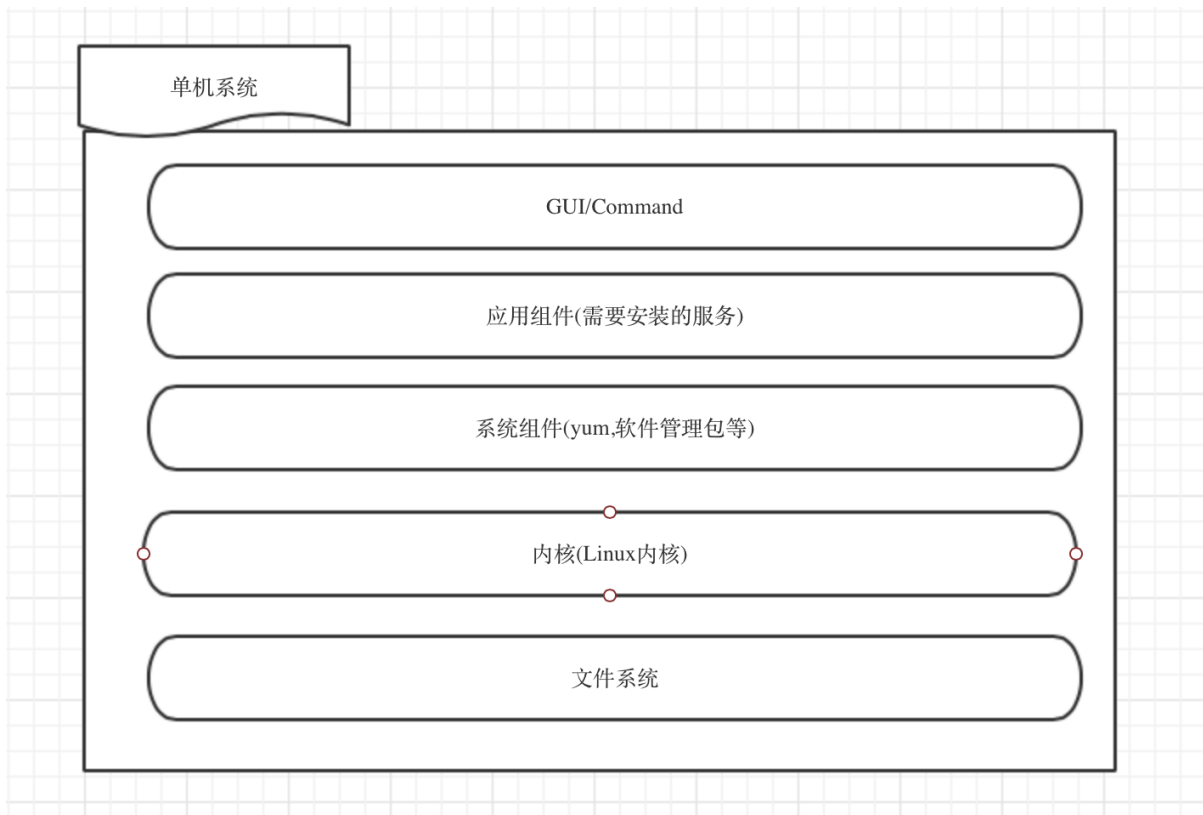
交互层(Web页面/API)

应用组件(需要部署的服务)

系统组件(zookeeper/kafka/容器调度,类似kubenet)

内核(Yarn,分布式shell引擎)

文件系统(HDFS/MySQL/HBase/Redis)



事实上，我们都是将一些成熟的组件或者包裹一层标准的API,或是直接以其API作为标准结构，构成我们的基础的系统级别的API。

---

## 分布式操作系统的意义

IASS + 分布式系统 构建起了一个完整的IT平台。具有如下特点：

1. 在这个平台上，只有两类应用：
  - 系统组件，用来增强系统的功能。典型如动态扩容等功能
  - 应用程序，业务代码，比如站点，或者某个API服务
2. 支持App Store. 所有应用都是APP,部署过程就是APP的安装过程，和现在的单机系统保持了高度一致。
3. 规范化。为了实现自动运维等，我们不再是为了某个功能开发一堆的难以复用的系统，他们都会拥有切实的规范，并且以系统组件或者应用程序的方式被以标准的方式安装进

分布式系统中。

---

## 分布式操作系统组件

前面我们提及了分布式系统的分层。我们现在详细看看这些分层的组件的详细情况。

- 分布式调度内核 (Borg(Yarn/Mesos)), 可以跨服务器进行资源调度
- 存储, 下面的都会作为操作系统的标准API对外提供, 一个应用只要部署到分布式系统中, 就有下列存储器可用:
  - 分布式文件系统 (HDFS)
  - 支持事物存储MySQL集群
  - NoSQL存储 HBase
  - 内存缓存 Redis
- 进程模型 (以Docker为代表的容器技术)。这块很简单, 在分布式操作系统运行的一切都是容器。
- 分布式系统可执行程序。我们采用标准的安装包分发机制。支持下列三种:

```
* .tar.gz 普通的解压即可启动的安装包, 会通过Docker包裹运行
* .git 源码, 分布式系统系统可通过类似Maven等构建工具编译即可使用
* .image 容器镜像, 分布式系统可直接加载并且运行的可执行镜像
```

- 服务注册API
  - 基于Zookeeper实现
  - 任何抛在分布式系统中的应用都需要通过接口注册, 类似windows注册表
  - 提供HTTP/RPC注册接口
- 进程通讯
  - 进程异步通讯模型(消息队列)
  - 进程通讯模型(HTTP/RPC)
- 用户态组件

- 特点为直接基于Yarn/Mesos开发
- 类似单操作系统里的用户态
- 典型如： Hadoop MR 实现， Spark Yarn 实现， 增强了操作系统功能
- 应用程序
  - 多实例
  - 应用的互调需要透过分布式系统
  - 多实例扩容等则需要用户态组件来进行增强
- UI系统(Web化可视界面)
- 系统中的应用程序定位
  - 单机操作系统为 IP+端口
  - 在分布式中， 内部定位通过服务注册API,外部调用， 则透过Nginx/Haproxy暴露， 内部的服务变更(比如扩容后)， 系统会自动调整Nginx/Haproxy的配置。

---

## 分布式操作系统如何现存应用进行交互

容器技术使得所有应用可以被操作系统运行起来。并且可以吻合Yarn内核对资源控制的要求。但是我们可能需要对被容器包括起来应用更细致的控制。我们先来看两个概念。

### 1. 哑应用

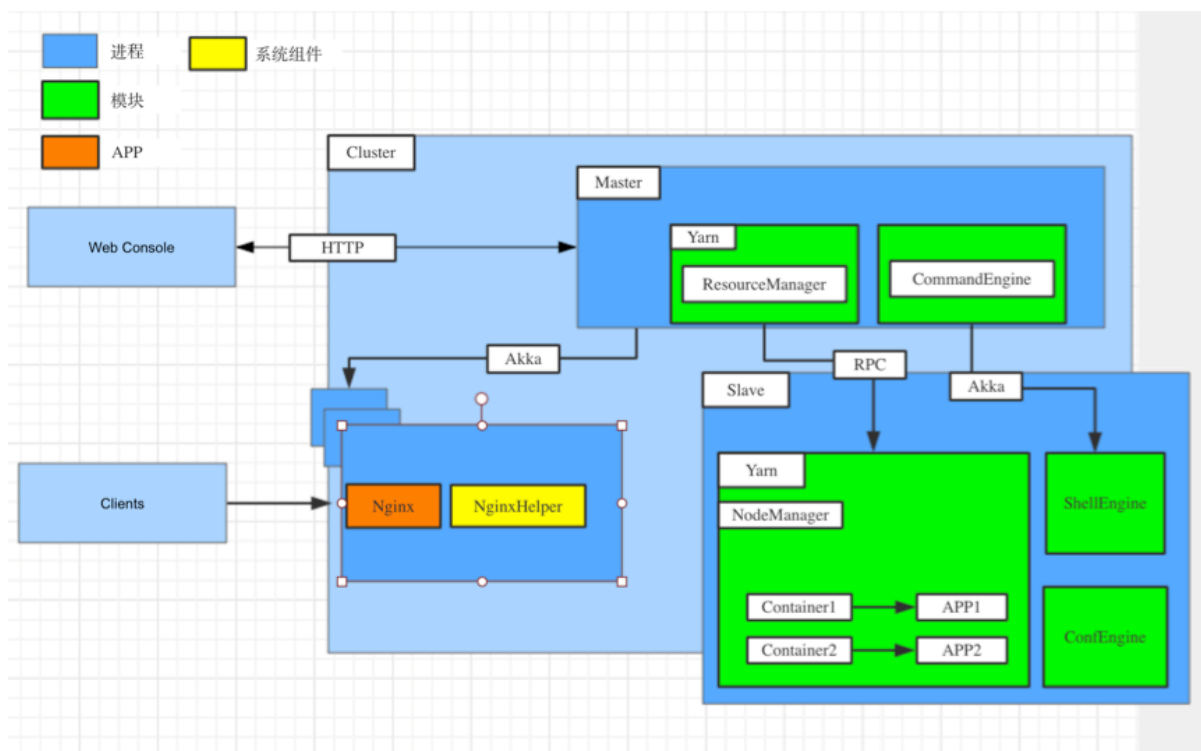
所谓哑应用指的是无法和分布式系统直接进行交互， 分布式系统也仅仅透过容器能进行生命周期的控制， 比如关闭或者开启的应用。典型的比如MySQL,Nginx等这些基础应用。他们一般有自己特有的交互方式， 譬如命令行或者socket协议或者HTTP协议。

### 1. 伴生组件

因为有了哑应用的存在， 操作系统为了能够和这些应用交互， 所以有了伴生组件的存在。这些伴生组件和哑应用具有相同的生命周期。伴生组件其实是哑应用的Proxy,从而使得哑应用可以和分布式系统进行交流。典型的比如， 某个服务被关停后， 该事件会被操作系统获知， 操作系统会将该事件发送给Nginx的伴生组件， 伴生组件转化为Nginx能够识别的指令， 将停止的服务从Nginx的ProxyBackend列表中剔除。

---

## 分布式系统原型-猛犸



这是一个典型的master-slave结构。

Master进程包含：

1. ResourceManager. 基于Yarn封装的一套组件。
2. CommandEngine, 命令系统, 通过Akka可以向Slave的ShellEngine发布任何Shell脚本指令
3. APPEngine, APP部署支持, APP信息存储查询等。比如根据安装包的后缀不同会将安装包丢给不同的安装组件去安装等。

Slave进程：

1. NodeManager. 基于Yarn封装的资源管理组件
2. ShellEngine. 执行由Master的CommandEngine发布的指令(目前支持Shell)
3. ConfEngine. 如果系统发现用户安装了Zookeeper,则会请求确认开启, 从而支持配置文件管理。

Web Console:

猛犸的默认交互端。通过该Console可以进行应用安装，管理等。

通讯协议:

调用者	服务者	协议
Web Console	Master	HTTP
CommandEngine	ShellEngine	AKKA
ResourceManager	NodeManager	HADOOP RPC

猛犸系统透过APPEngine支持两种应用/资源管理模式:

1. 静态安装模型。也就是传统的‘指定服务器’部署模式。APPEngine默认透过CommandEngine做这种支持。基本步骤为:

1. 上传安装包
2. 选择服务器
3. 收集参数
4. 生成配置文件
5. 分发安装包
6. 安装
7. 进入管理界面

2, 3 两个步骤完全配置化。4,5,6步骤如果有需要，应用可以按照指定格式提供相应的脚本即可。

为了能够管理，如果下载的是【哑应用】，则需要撰写一个指定格式的shell脚本并添加到tar.gz包里即可。通常只需要两行即可，分布式系统便能够完成启动，关闭，监控进程存活等功能。

应用的安装信息并不会存储在master上，而是存储在每台Slave上。由Slave通过心跳上报到Master端。静态模型中，Master是完全无状态的。

2. 动态安装模型

该模式下，所有资源由Yarn内核进行动态分配管理。我们提供了一个DynamicDeploy的



系统组件(需要额外安装)。如果用户提交的应用是.image后缀并且跳过服务器选择, 则AppEngine会将该安装包交给DynamicDeploy, DynamicDeploy会按下面的流程进行处理:

1. DynamicDeploy向ResourceManager提交资源资源申请, ResourceManager启动ApplicationMaster
2. ApplicationMaster启动DynamicDeploy的Driver(Master)
3. ApplicationMaster向Resource Manager申请资源, Resource Manager根据集群资源情况为其分配YARN Container,
4. 这些Container会连接Driver。Driver发布启动指令给各个Container.Container会执行download image,load image,run image 三部分指令
5. run指令会自动添加cpu,内存, 磁盘配额。并且给Docker的容器配置一个随机端口
6. Container会将Docker容器的IP,端口上报给Driver,Driver会将这些发送给APPEngine.
7. APPEngine会把这些数据发送给Zookeeper或者其他存储介质
8. Nginx系统组件(需要事先在猛犸中安装)会监听这些变化, 更新到Nginx中
9. 完成部署

这里本质上Container,Nginx系统组件 除了拥有一些额外功能外, 都是伴生对象。因为我们安装的大部分应用, 都是【哑应用】。

## 存储

在分布式操作系统中, 可提供分布式文件系统(HDFS),也可以提供‘硬件级别’的磁盘, 还有高层次支持事务的MySQL集群, 高速缓存Redis集群, 优秀的Key/Value存储 HBase等。

以MySQL为例, 在分布式操作系统中是如何实现高可用的呢?

分布式操作系统中, MySQL 以容器状态运行, 文件通过Volumn挂载在实际磁盘中, 实现单Master多Slave结构, 我们称之为一个Cluster,系统允许多个Cluster存在。

值得注意的是, MySQL这些信息都会在系统的注册表中找到。

我们基于Yarn开发了一套傍生组件, 该组件是专门监控这种单Master节点类型的系统, 一旦安装了MySQL,系统会自动启用该组件, 定时监控Master的可用性, 一旦发现Master不可以用, 则找到数据最新的Slave, 提升为Master,并且变更注册表信息(如IP,端口等), 所有使用该MySQL的服务组件都会得到通知, 从而实现动态变更。

同理，Redis Cluster也是通过相同的方式实现高可用。

## 应用安装

前面我们提到，分布式操作系统支持三种类型的APP，分别为

1. .tar.gz 比如Hadoop
2. .image docker 镜像文件
3. .git git源码

这些APP会放在分布式系统中的AppStore中。用户可以制作好上面的APP安装包后上传到AppStore中，也可以直接使用已经制作好的安装包。

分布式系统针对类似Hadoop,或者MySQL,或者普通的Web应用都有单独的安装策略。

## 服务注册API

基于Zookeeper集群实现高可用。由系统提供Http API实现注册。注册机制为URL地址。每个服务都由一个URL构成，URL可以使用URL参数进行区分。任何一个安装的应用都会将自己的信息注册到该API上。

## 进程通讯

涉及到分布式系统内部通讯以及分布式系统间通讯。

系统内部异步通讯通过内置的Kaffka来实现。对应用来说是透明的。同步通讯则直接使用Http/RPC即可。

都需要透过服务注册API，也就是必须经过分布式操作系统。

而如果外部，比如用户浏览器，或者第三方不在分布式操作系统的应用想调用系统内部服务，则透过已经封装好的 Nginx/Haproxy 来完成。

## MapReduce/Spark/MPI等工具在分布式系统的定位

Yarn的可编程性让我们可以开发一些系统组件，从而让系统有了新的能力，比如MapReduce/Spark 等，让分布式系统有了执行批量离线(或者准实时)的功能。我们认为这是对分布式系统的一种增强。因为这套应用是直接基于分布式系统内核编程的。

## 分布式系统的自动容量规划机制

我们发现单机桌面软件运行时，大部分情况是不需要你写资源申请的。而事实上，即使在实际的线上部署，部署者也很难确切的知道我应该要多少CPU,多少内存比较合理，当然，磁盘理论可以预估的。

前面我们提到，类似于Windows的注册表，一个进程启动后需要将自己告知系统，而且他如果要调用其他的进程需要通过分布式系统的注册API来完成，一旦获得依赖的其他进程信息，则通过操作系统提供的异步通讯模型-消息队列，或者直接的通讯模型(HTTP/RPC)来完成实际的数据传递。

由此，我们可以知道，当一个服务被启动，分布式系统很容易获取到这个应用的依赖，包括对外的RPC调用,Http调用，数据库调用，缓存调用（如果应用框架支持，这些调用关系是通过配置文件静态分析出来的）。这和安卓一样，一个应用需要申明他是否需要联网，是否需要自启动，是否需要XXX。用户只需要提供一个预估调用量，或者需要他能承受的一个调用量给系统，系统结合这些数值，可以自动计算出需要启动多少个容器，配置多少CPU,多少内存，计算的方式非常多，其中还有一个小技巧，是可以参看有着类似外部依赖的已经部署在线上的服务，参看他的数值。当然，用户也可以直接通过配置文件告诉系统自己需要的资源申请。

系统一旦计算出来后，用户确认即可，防止出现错误，这个时候，系统采用的是贪婪算法，它会采用比预估值大一倍的容量来启动这个服务群集。接着根据实际运行结果，比如一周的运行数据(各个容器的CPU,内存等))，来确定一个更合适的值，这个时候可以通过服务的自动伸缩性来解决减少服务的。