**流程**

* Java主流程代码，Shell/Python代码对主流程调用，完成控制逻辑
* QA需要分别针对Java主流程代码测试，并添加Python代码的测试
* 增加流程需要修改Python控制逻辑，并做整体逻辑回归
* Shell/Python代码的灵活性较高，实现风格迥异，在业务变化比较频繁的情况下，单元测试变化也较大

**目标**

* 能够增加业务调度的稳定性和健壮性和灵活性
* 将重心放在业务代码的实现上，通过模块化、插件化的方案将业务接入到调度系统中
* 支持流程、业务模块的复用，减少业务平台的开发
* 方便PE的运维，方便日志的查看，方便版本的更新、回滚
* 方便日常运维的操作，支持对流程的启停
* 良好的文档，以及社区有很好的活跃度，长期项目

**开源系统**

* server-sides：[Azkaban](https://link.jianshu.com?t=http:/azkaban.github.io/azkaban/docs/2.5/" \t "_blank)、[Oozie](https://link.jianshu.com?t=http:/oozie.apache.org/" \t "_blank)、[Luigi](https://link.jianshu.com?t=https:/github.com/spotify/luigi" \t "_blank)
* client-sides：[Cascading](https://link.jianshu.com?t=http:/www.cascading.org/" \t "_blank)、[Hamake](https://link.jianshu.com?t=https:/code.google.com/p/hamake/" \t "_blank)

**Oozie VS Azkaban**

**架构**

* Azkaban
  + 主要由三个组件构成：AzkabanWebServer(Jetty)、AzkabanExecutorServer、DataBase(H2/Mysql)
  + WebServer是整个Azkaban工作流系统的主要管理者,负责project管理、用户登录认证、定时执行工作流、跟踪工作流执行进度等
  + ExecutorServer主要负责任务的执行

Azkaban组件

* Oozie
  + 主要部分是Oozie Client、Oozie Server（Tomcat）、DataBase（Debry/Mysql）
  + Oozie Client完成对workflow进行提交、查询、执行
  + 实现
    - Ooozie向Oozie Server提交一个Workflow(job.property)，Server端收到后会根据workflow xml，提交一个map only的MR Job
    - Job在map task中通过Hadoop JobClient将action对应的jar/job.xml提交到JobTracker
    - action job未完成前，map only job一直等待
    - Oozie server通过callback url通知等待action job的完成
    - action job执行成功后会将action对应的状态信息保存到DB中

**主要概念**

* Azkaban
  + Job：最小的执行单元，作为DAG的一个结点
  + Flow：由多个Job组成，并通过dependent配置Job的依赖属性
* Oozie
  + Control Node：工作流的开始、结束以及决定Workflow的执行路径的节点（start、end、kill、decision、fork/join）
  + Action Node：工作流执行的计算任务，支持的类型包括（HDFS、MapReduce、Java、Shell、SSH、Pig、Hive、E-Mail、Sub-Workflow、Sqoop、Distcp）
  + Workflow：由Control Node以及一系列Action Node组成的工作流
  + Coordinator：根据指定Cron信息触发的workflow
  + Bundle：按照组的方式批量管理Coordinator任务，实现集中的启停

**安装部署**

* Azkaban
  + 部署方便：下载解压，配置连接已有Mysql，并启动WebServer和ExecutorServer
  + 支持solo-server的模式以及multi-server的模式
  + multi-server模式下需要配置mysql、JobType plugin
  + 配置用户权限
* Oozie
  + 部署相对复杂，需要配置hadoop的core-site.xml文件增加代理用户的配置，并重启hadoop集群
  + 默认使用derby数据库，推荐使用Mysql数据库
  + 安装环境依赖maven、tomcat、hadoop，同时需要独立安装extJS包
  + 安装之后需要更新sharelib并初始化数据库，容易出现问题

**工作流定义**

* Azkaban
  + Job和Flow的定义直接通过key-value的属性文件配置
  + Azkaban内建支持的两种JobType只有Shell和Java，其它的JobType需要以插件的形式接入Azkaban
  + 依赖关系直接通过dependent属性配置
* Oozie
  + 基于hPDL语言描述的配置文件（XML）
  + 通过control flow node以及一系列action node确定一个workflow
  + Coordinator有两种触发条件：时间触发、数据触发
  + 支持参数化的定义Workflow的配置以及Java EL函数，多种配置传入的方式
    - configure-default.xml
    - global域
    - configuration域
    - job.property

**工作流发布**

* Azkaban
  + 将用户定义的job的property配置文件以及依赖的lib都打包在同一个zip中
  + 通过WebUI或者通过ajax接口将zip包提交到WebServer
  + 定时任务的发布需要在WebUI上直接设置
* Oozie
  + 将workflow、coordinator、bundle的配置以及依赖的lib都放在统一的目录中，并上传到hdfs上
  + 通过job.property配置指定任务的hdfs根路径
  + 通过命令行的方式，将workflow提交

**WebUI**

* Azkaban
  + Azkaban所有任务的提交、监控、执行、编辑都可以通过前端完成
* Oozie
  + 只支持Workflow当前以及历史状态的查询
  + action的log需要通过oozie提交的hadoop job的日志查看
  + cloudera HUE提供了更加nice的UI，支持oozie任务查看、编辑、操作的功能

**运维**

* Azkaban
  + 通过Web端可以完成任务的提交、编辑、执行、查看
  + 能够通过Web端完成任务的重启以及暂停
* Oozie
  + Oozie原生Web页面能够看到工作流的图形化定义、log日志、workflow状态以及action的执行状态
  + 通过HUE可以更加方便的在Web页面上完成workflow的启动、停止、恢复

**日志、监控**

* Azkaban
  + 支持stdout日志的直接输出，支持配置SLA以及邮件报警
* Oozie
  + stdout日志只能通过点击oozie启动的mr-job中查看，ssh action没法查看stdout日志
  + 支持SLA Alters
  + 基础的E-Mail通知，后续可以自定义实现报警的Action
  + 支持JMS API，可以对接JMS provider，由下游自己获取状态并做报警，默认可以配置ActiveMQ

**扩展**

* Azkaban
  + 非常好的模块化和插件化的结构，支持自定义JobType或者插件，本身支持的类型并不多
* Oozie
  + 可以扩展自定义类型的Action
  + 支持自定义EL函数

**Oozie方案**

**使用场景**

* 需要按顺序并能够并行处理的工作流（DAG）
* 对运行结果或异常需要报警、重启
* 需要Cron的任务
* 适用于批量处理的任务，不适合实时的情况

**优点**

* Oozie与Hadoop生态系统紧密结合，提供做种场景的抽象
* Oozie有更强大的社区支持，文档
* Job提交到hadoop集群，server本身并不启动任何job
* 通过control node/action node能够覆盖大多数的应用场景
* Coordinator支持时间、数据触发的启动模式
* 支持参数化和EL语言定义workflow，方便复用
* 结合HUE，能够方便的对workflow查看以及运维
* 结合HUE，能够完成workflow在前端页面的编辑、提交
* 支持action之间内存数据的交互（默认2K）
* 支持workflow的rerun（从某一个节点重启）

**缺点**

* 学习门槛比较高，需要丰富的Action配置方法，需要写大量的XML配置
* 不支持动态的fork，不支持loop
* 应用中有动态参数不好支持

| **Airflow** | **Azkaban** | **Conductor** | **Oozie** | **AWS Step函数** |
| --- | --- | --- | --- | --- |
| **所有者** | Apache  （以前是Airbnb） | LinkedIn | Netflix公司 | 阿帕奇 | 亚马逊 |
| **社区** | 很活跃 | 有点活跃 | 活性 | 活性 | N / A |
| **历史** | 4年 | 7年 | 1.5年 | 8年 | 1.5年 |
| **主要目的** | 通用批处理 | Hadoop作业调度 | 微服务编排 | Hadoop作业调度 | 通用工作流 |
| **流程定义** | Python | 自定义DSL | JSON | XML | JSON |
| **支持单节点** | 是 | 是 | 是 | 是 | N / A |
| **快速演示设置** | 是 | 是 | 是 | 没有 | N / A |
| **支持HA** | 是 | 是 | 是 | 是 | 是 |
| **单点故障** | 是 （单一调度程序） | 是 （单个Web和调度程序组合节点） | 没有 | 没有 | 没有 |
| **HA额外要求** | Celery / Dask / Mesos + Load Balancer + DB | D B | 负载均衡器（Web节点）+ DB | 负载均衡器（Web节点）+ DB + Zookeeper | 本地人 |
| **Cron Job** | 是 | 是 | 没有 | 是 | 是 |
| **执行模型** | 推 | 推 | 轮询 | 轮询 | 未知 |
| **Rest API触发器** | 是 | 是 | 是 | 是 | 是 |
| **参数化执行** | 是 | 是 | 是 | 是 | 是 |
| **由外部事件触发** | 是 | 没有 | 没有 | 是 | 是 |
| **本机等待任务支持** | 是 | 没有 | 是（需要外部信号） | 没有 | 是 |
| **回填支持** | 是 | 没有 | 没有 | 是 | 没有 |
| **本机Web身份验证** | LDAP /密码 | XML密码 | 没有 | 没有 | 没有 |
| **监控** | 是 | 有限 | 有限 | 是 | 有限 |
| **可扩展性** | 取决于执行程序设置 | 好 | 很好 | 很好 | 很好 |

## 

## 声明

我不是任何这些引擎的专家，但已经使用了其中的一些（Airflow和Azkaban）并检查了代码，对于其他一些产品，我要么只阅读代码（Conductor）或文档（Oozie / AWS步骤函数），由于大多数是OSS项目，我当然可能错过了某些未记录的功能或社区贡献的插件。如果你发现任何错误，我很乐意更新。

底线：阅读本文时请自行判断。

## Airflow

### 优点

     与所有其他解决方案相比，Airflow是一种功能超强的引擎，你不仅可以使用插件来支持各种作业，包括数据处理作业：Hive，Pig（尽管你也可以通过shell命令提交它们），以及通过文件/ db entry / s3来触发的一般流程管理，或者等待来自Web端点的预期输出，但它也提供了一个很好的UI，允许你通过代码/图形检查DAG（工作流依赖性），并监视作业的实时执行。

目前充满活力的社区也可以高度定制Airflow。你可以使用本地执行程序通过单个节点运行所有作业，或通过Celery / Dask / Mesos编排将它们分发到一组工作节点。

### 缺点

      气流本身仍然不是很成熟（实际上Oozie可能是这里唯一的“成熟”引擎），调度程序需要定期轮询调度计划并将作业发送给执行程序，这意味着它将不断地从“盒子”中甩出大量的日志。

由于它通过“滴答”定期轮询工作，你的工作不能保证“实时”安排，随着并发工作数量的增加，这会变得更糟。

同时，由于你有一个集中式调度程序，如果它出现故障或卡住，你的正在运行的作业将不会像执行程序的作业那样受到影响，但是不会安排新的作业了。当你使用HA设置运行时，这尤其令人困惑，其中你有多个Web节点，调度程序，代理（通常是Celery案例中的消息队列），多个执行程序。当调度程序因任何原因而卡住时，你在Web UI中看到的所有任务都在运行，但实际上它们实际上并没有向前运行，而执行程序却高兴地报告它们没问题。换句话说，默认监控仍然远非银弹。

从初看起来，Web UI非常好用，然而，它有时会让新用户感到困惑。我的DAG运行是什么意思，我的任务竟然没有状态？这些图表也不是搜索友好的，更不用说一些功能还远远没有详细记录（尽管文档看起来确实很好，我的意思是，与Oozie相比，后者似乎已经过时了）。

回填设计在某些情况下是好的，但在其他情况下非常容易出错。如果你的cron计划已禁用并且稍后重新启用，那么它会尝试追赶，如果你的工作不是幂等的，那么就会发生真实的无可挽回的事情。

## Azkaban

### 优点

    在所有引擎中，Azkaban可能是最容易开箱即用的。UI非常直观且易于使用。调度和REST API工作得很好。

有限的HA设置开箱即用。不需要负载均衡器，因为你只能有一个Web节点。你可以配置它如何选择执行程序节点然后才能将作业推送到它，它通常看起来非常好，只要有足够的容量来执行程序节点，就可以轻松运行数万个作业。

### 缺点

    作为通用编排引擎，它没有非常丰富的功能，但可能不是最初设计的目的，它的优势在于对Hadoop / Pig / Hive的原生支持，尽管你也可以使用命令行实现这些功能，但它本身不能通过Airflow等外部资源触发工作，也不支持工作等待模式。虽然你可以通过java代码/脚本实现比较繁忙的工作，但这会导致资源利用率下降。

与其他文档和配置相比，文档和配置通常有点混乱。它可能不应该推荐为初学者使用，设计很好但是你最好有一个大型数据中心来运行执行程序，因为当执行程序耗尽资源而没有额外的监视功能时，调度会停止。与其他代码相比，整体代码质量有点朝向低端，所以它通常只有在资源不成问题时才能很好地扩展。

设置/设计不是云友好的。你几乎应该拥有稳定的裸机，而不是动态分配具有动态IP的虚拟实例。如果机器消失，调度会“向南飘移”。

监控部分通过JMX可接受（似乎没有记录）。但是，如果你的机器负载很重，它通常不会很好，因为端点可能会卡住。

## Conductor

### 优点

   将Conductor引入本次竞赛有点不公平，因为它的真正目的是微服务编排，无论这意味着什么，它的HA模型涉及一定数量的服务器，它们位于负载均衡器后面，将任务放入消息队列中，工作节点将轮询这个队列，这意味着你不太可能遇到停滞的调度。在API参数化执行的帮助下，如果你正确设置负载均衡器/服务发现层，它实际上非常擅长调度和扩展。

### 缺点

   用户界面需要更多的提高，目前监控非常有限。虽然通用调度可能已经足够好了。

这是开箱即用的裸机。甚至没有运行shell脚本的本机支持，尽管通过python实现任务工作者很容易通过提供的示例完成工作。

## Oozie

### 优点

Oozie通过db设置提供了一个看似可靠的HA模型（貌似b / c我没有看到它），它为Hadoop相关工作提供本机支持，因为它是为该生态系统构建的。

### 缺点

对于通用流程调度而言，不是一个非常好的候选者，因为XML定义对于定义轻量级作业非常冗长和繁琐。

它还需要相当多的外设设置。你需要一个zookeeper集群，一个db，一个负载均衡器，每个节点都需要运行像Tomcat这样的Web应用程序容器。初始设置也需要一些时间，这对初次使用的用户来说是不友好的。

## Step函数

### 优点

亚马逊云的步骤函数是相当新（2016年12月推出），然而，未来似乎很有希望。凭借云平台和lambda函数的HA特性，它几乎感觉它可以轻松地无限扩展（与其他人相比）。

它还为通用工作流处理提供了一些有用的功能，如等待支持和基于输出的动态分支。

它也相当便宜：如果你没有运行成千上万的工作，这可能比运行你自己的集群更好。

### 缺点

只能由AWS用户使用。如果你还不是其中之一，那就木有办法了。

Lambda需要额外的工作来进行生产级迭代/部署。

没有用户界面（很好，但它实际上只是一个控制台）。因此，如果你需要任何级别的监控，你需要自己使用cloudwatch构建它。