分布式调度框架大集合

2018 年 09 月 15 日 18:03:31 路灯下的女孩 阅读数：1634

分布式任务调度框架

1、什么是分布式任务调度?

2、常见的分布式任务调度框架有哪些?

3、分布式任务调度框架的技术选型?

4、分布式任务调度框架的安装与使用?

大对比表格：https://pan.baidu.com/s/1CZAjTFqIhinzlVLnrrMUKQ

分布式任务调度，三个关键词：分布式、任务调度、配置中心。

分布式：平台是分布式部署的，各个节点之间可以无状态和无限的水平扩展；

任务调度：涉及到任务状态管理、任务调度请求的发送与接收、具体任务的分配、任务的具体执行；（这里又会遇到一共要处理哪些任务、任务要分配到哪些机器上处理、任务分发的时候判断哪些机器可以用等问题，所以又需要一个可以感知整个集群运行状态的配置中心）

配置中心：可以感知整个集群的状态、任务信息的注册

一个分布式任务调度系统需要以下内容：

web 模块、server 模块、Scheduler 模块、worker 模块、注册中心。

1、Web 模块：用来提供任务的信息，控制任务的状态、信息展示等。

2、Server 模块：负责接收 web 端传来的任务执行的信息，下发任务调度请求给 Scheduler，会去注册中心进行注册

3、Scheduler 模块：接收 server 端传来的调度请求，将任务进行更加细化的拆分然后下发，到注册中心进行注册，获取到可以干活的 worker。

4、Worker 模块：负责具体的任务执行。

5、注册中心。

1、什么是分布式任务调度?

任务调度是指基于给定的时间点，给定的时间间隔或者给定执行次数自动的执行任务。任务调度是是操作系统的重要组成部分，而对于实时的操作系统，任务调度直接影响着操作系统的实时性能。任务调度涉及到多线程并发、运行时间规则定制及解析、线程池的维护等诸多方面的工作。

WEB 服务器在接受请求时，会创建一个新的线程服务。但是资源有限，必须对资源进行控制，首先就是限制服务线程的最大数目，其次考虑以线程池共享服务的线程资源，降低频繁创建、销毁线程的消耗；然后任务调度信息的存储包括运行次数、调度规则以及运行数据等。一个合适的任务调度框架对于项目的整体性能来说显得尤为重要。

2、常见的任务调度框架有哪些?

我们在实际的开发工作中，或多或少的都会用到任务调度这个功能。常见的分布式任务调度框架有：cronsun、Elastic-job、saturn、lts、TBSchedule、xxl-job 等。

2.1cronsun

crontab 是 Linux 系统里面最简单易用的定时任务管理工具，在 Linux 上由 crond 来周期性的执行指令列表，执行的任务称为 cron job，多个任务就称为 crontab。crontab 任务调度指令的基本格式为：

```
* * * * * command
```

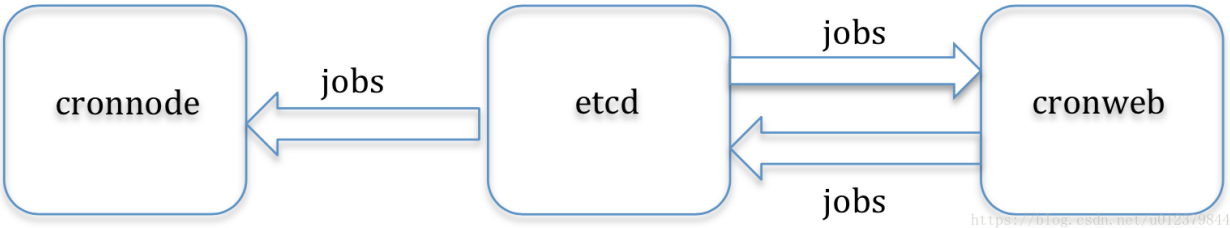
分 时 日 月 周 命令

但是时间久了之后会发现，crontab 会存在一些问题：

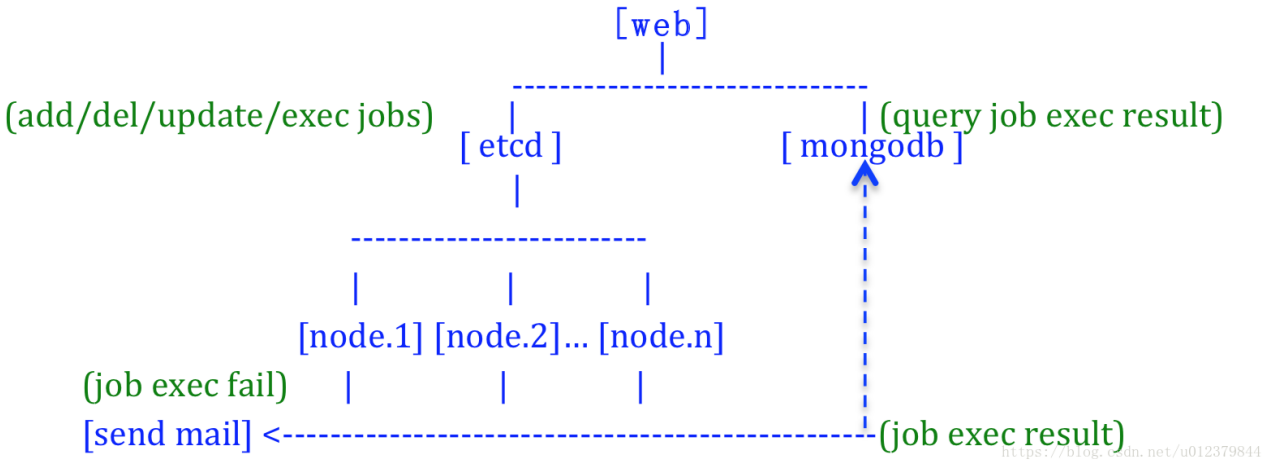
1. 大量的 crontab 分散在各台服务器，带来了很高的维护成本；
2. 任务没有按时执行，过了很长的时间才能发现，需要重试或者排查；
3. crontab 分散在很多集群上，需要一台一台的去查看日志；
4. crontab 存在单点问题，对于不能重复执行的定时任务很伤脑；
5. ……

因此非常需要一个集中管理定时任务的系统，于是就有了 cronsun。cronsun 是一个分布式任务系统，单个节点和 Linux 机器上的 contab 近似，是为了解决多台 Linux 机器上 crontab 任务管理不方便的问题，同时提供了任务高可用的支持（当某个节点死机的时候可以自动调整到正常的节点执行）。与此同时，它还支持界面管理机器上的任务，支持任务失败邮件提醒，安装简单，使用简便，是替换 crontab 的一个不错的选择。

cronsun 中主要有三个组件，都是通过 etcd 通讯的。cronnode 负责节点的分组及节点的状态，cronweb 是用来管理任务的、任务的执行结果都可以在上面看。



cronsun 的系统架构如下图所示，简单的来说就是，所有的任务都会存储在一个分布式 etcd 里，单个 crond 部署成一个服务，也就是图中所示的 node.1、node.2、node.n 等，然后再由 web 界面去管理。如果任务执行失败的话，会发送失败的邮件，当单个节点死机的时候，也会自动调整到正常的节点去执行任务。



cronsun 是在管理后台添加任务的，所以一旦管理后台泄漏出去了，则存在一定的危险性，所以 cronsun 支持 security.json 的安全设置：

```
{
"open": true,
"#users": "允许选择运行脚本的用户", "users": [
"www", "db" ],
"#ext": "允许添加以下扩展名结束的脚本", "ext": [
".cron.sh", ".cron.py" ]
}
```

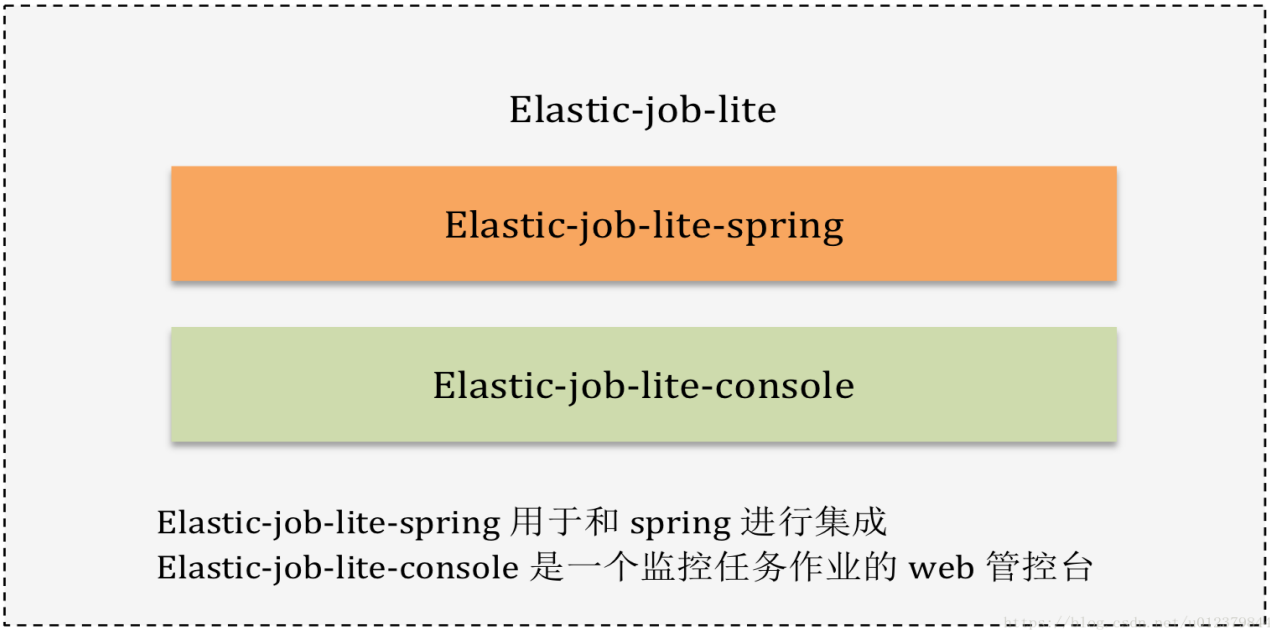
如以上设置开启安全限制，则添加和执行任务的时候只允许选择配置里面指定的用户来执行脚本，并且脚本的扩展名要在配置的脚本的扩展名限制的列表里面。

2.2、Elastic-job

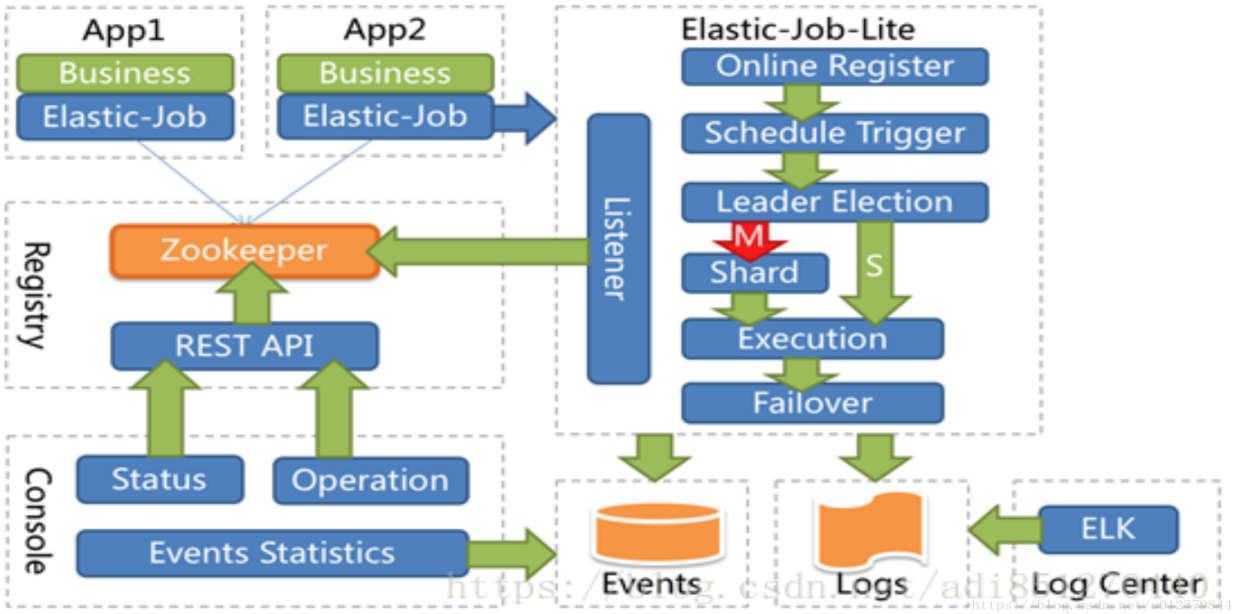
Elastic-job 是当当开源的一款非常好用的作业框架，Elastic-job 在 2.x 之后，出现了两个相互独立的产品线：Elastic-job-lite 和 Elastic-job-cloud。

2.2.1、Elastic-job-lite

Elastic-job-lite 定位为轻量级无中心化的解决方案，使用 jar 包的形式提供分布式任务的协调服务，外部依赖仅依赖于 zookeeper。



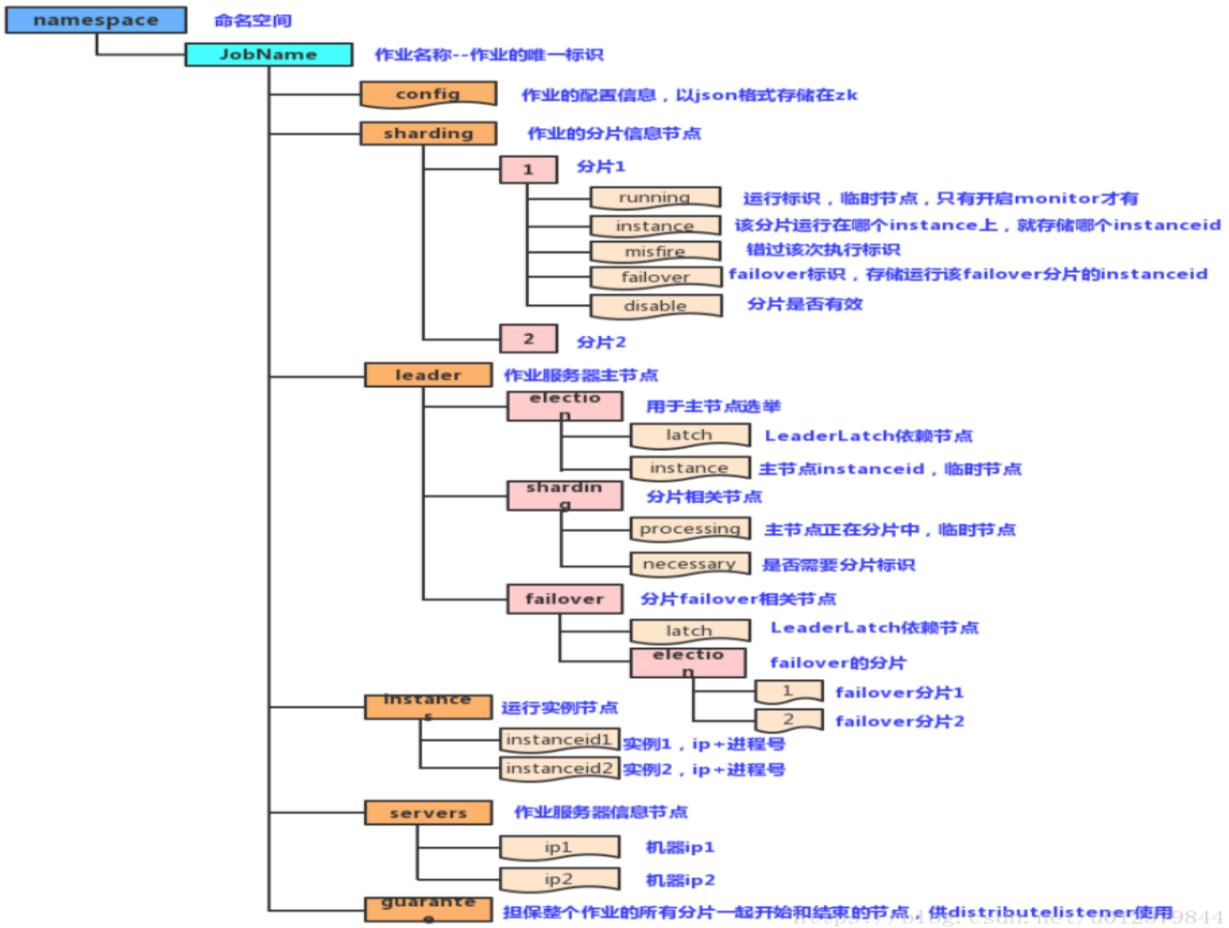
Elastic-job-lite 的架构图如下图所示：



从上面的框架图中可以看出，Elastic-job-lite 框架使用 zookeeper 作为注册中心，Elastic-job-lite 框架通过监听感知 zookeeper 数据的变化，并做相应的处理；运维平台也仅是通过读取 zk 数据来展现作业状态，或是更新 zk 数据修改全局配置。运维平台和 Elastic-job-lite 没有直接的关系，完全解耦合。Elastic-job-lite 并不直接提供数据处理的功能，框架只会将分片项分配给各个正在运行中的服务器，分片项与真是数据的对应关系需要开发者在应用程序中自行处理。

Elastic-job-lite 并无作业调度中心节点，而是基于部署作业框架的程序在到达相应时间点时各自触发调度。注册中心仅用于作业注册和监控信息存储，而主作业节点仅用于处理分片和清理的功能。

（1）注册中心的数据结构



我们先来了解一下该框架在 zookeeper 上的节点情况。首先注册中心在命名的空间下创建作业名称节点（作业名称用来区分不同的作业，一旦修改名称，则认为是新的作业），作业名称节点下又包含 5 个子节点：

config：保存作业的配置信息，以 JSON 格式存储

sharding：保存作业的分片信息，它的子节点是分片项序号，从零开始，至分片总数减一

leader：该节点保存作业服务器主节点的信息，分为 election、sharding 和 failover 三个子节点，分别用于主节点的选举、分片和失效转移

instances：该节点保存的是作业运行实例的信息，子节点是当前作业运行实例的主键

servers：该节点保存作业服务器的信息，子节点是作业服务器的 IP 地址

（2）实现原理

1. 第一台服务器上线触发主服务器选举，主服务器一旦下线，则重新触发选举，选举过程中阻塞，只有当主服务器选举完成，才会去执行其他的任务；

2. 某服务器上线时会自动将服务器的信息注册到注册中心，下线时会自动更新服务器的状态；
1. 主节点选举，服务器上下线，分片总数变更均更新重新分片标记；
2. 定时任务触发时，如需重新分片，则通过主服务器分片，分片过程中阻塞，分片结束后才可以执行任务。如分片过程中主服务器下线，则先选举主服务器在分片；
3. 由上一项说明可知，为了维持作业运行时的稳定性，运行过程中只会标记分片的状态，不会重新分片，分片仅可能发生在下次任务触发前；
4. 每次分片都会按照 ip 排序，保证分片结果不会产生较大的波动；
5. 实现失效转移功能，在某台服务器执行完毕后主动抓取未分配的分片，并且在某台服务器下线后主动寻找可用的服务器执行任务。

elastic 底层的任务调度还是使用的 quartz，通过 zookeeper 来动态给 job 节点分片。如果很大体量的用户需要我们在特定的时间段内计算完成，那么我们肯定是希望我们的任务可以通过集群达到水平的扩展，集群里的每个节点都处理部分的用户，不管用户的数量有多大，我们只需要增加机器就可以了。举个例子：比如我们希望 3 台机器跑 job，我么将我们的任务分成 3 片，框架通过 zk 的协调，最终会让 3 台机器分配到 0，1，2 的任务片，比如 server0->0、server1->1、server2->2，当 server0 执行时，可以只查询 id%3==0 的用户，server1 可以只查询 id%3==1 的用户，server2 可以只查询 id%3==2 的用户。

在以上的基础上再增加一个 server3，此时，server3 分不到任何的分片，没有分到任务分片的程序将不执行。如果此时 server2 挂了，那么 server2 被分到的任务分片将会分配给 server3，所以 server3 就会代替 server2 执行。如果此时 server3 也挂了，那么框架也会自动的将 server3 的任务分片随机分配到 server0 或者 server1，那么就可能成：server0->0、server1->1,2。

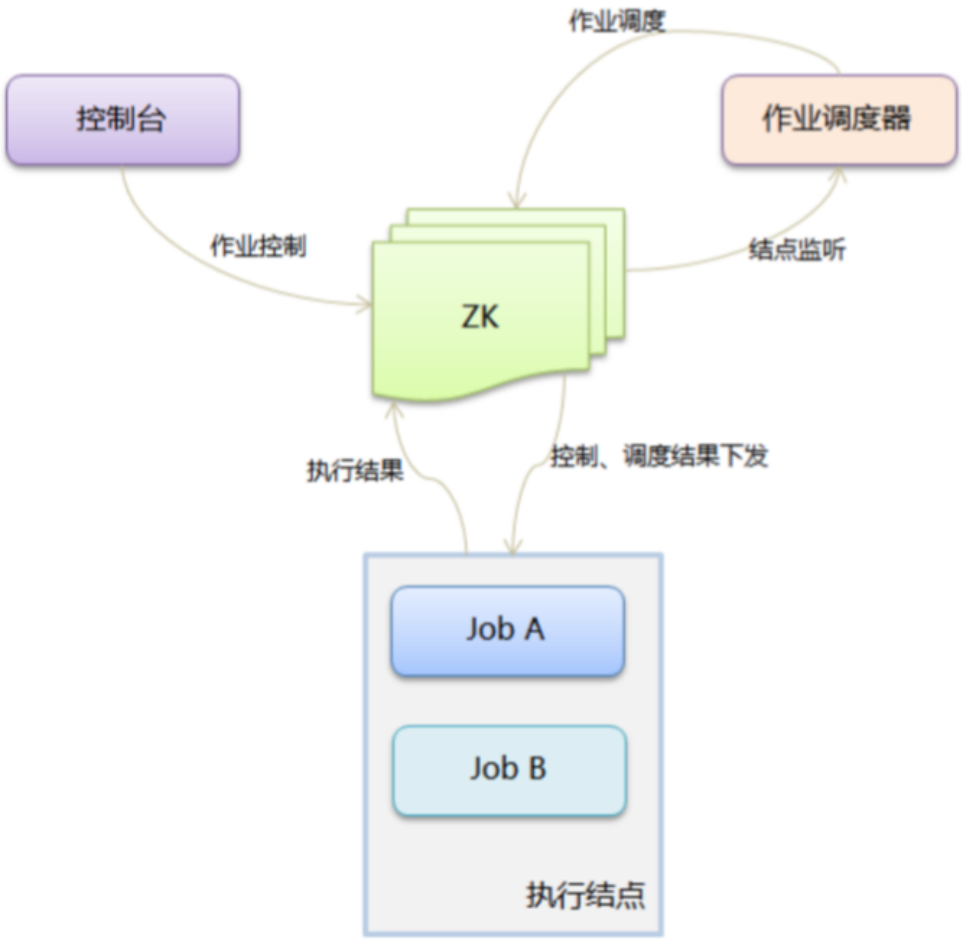
这种特性称之为**弹性扩容**。

2.2.2、Elastic-job-cloud

Elastic-job-cloud 包含了 Elastic-job-lite 的全部功能，它是以私有云平台的方式提供集资源、调度以及分片为一体的全量级解决方案，依赖于 Mesos 和 Zookeeper，它额外提供了资源治理、应用分发以及进程隔离等服务。他们两个提供同一套 API 开发作业，开发者仅需一次开发，然后可根据需要以 lite 或 cloud 的方式部署。

2.3、saturn

Saturn（定时任务调度系统）是唯品会自主研发的分布式的定时任务的调度平台，它是基于 Elastic-job 版本 1 开发的。目标是取代传统的 Linux Cron/Spring Batch Job/Quartz 的方式，做到全域统一配置、统一监控、任务高可用以及分片。Saturn 的任务可以使用多种语言开发，比如 python、Go、Shell、Java、Php 等。



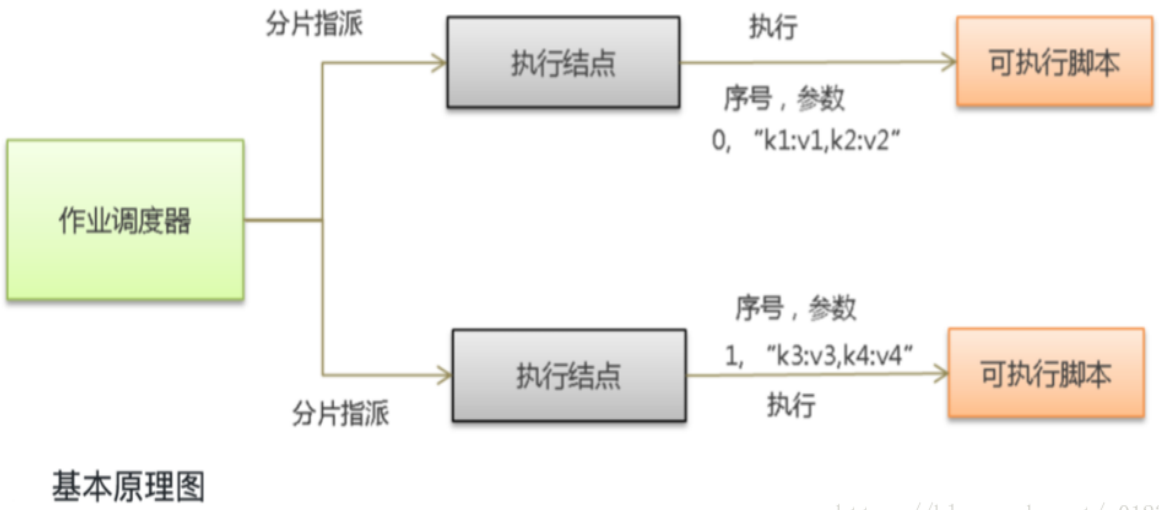
系统逻辑架构图

Saturn 包括两大部分，Saturn Console 和 Saturn Executor。Console 是一个 WEB UI，用来对作业/Executor 的管理，统计报表展现等。他同时也是整个调度系统的大脑：将作业任务分配到各 Executor。Executor 是执行任务的 worker：按照作业配置的要求去执行部署于 Executor 所在容器或物理机当中的作业脚本和代码。Saturn 高度依赖于 zookeeper，每个 executor 及调度服务都会在 zookeeper 上进行注册，确保调度程序能够及时得到 executor 的状态。

Saturn 定时任务调度的最小单位是分片，即任务的一个执行单元。Saturn 的基本任务就是将任务分成多个分片，并将每个分片通过算法调度到对应的 executor 上去执行。

2.3.1、Staurn 基本原理

Saturn 的基本原理是将作业在逻辑上划分为若干个分片，通过作业分片调度器将作业分片指派给特定的执行节点。执行节点通过 quartz 触发执行作业的具体实现，在执行的时候，会将分片序号和参数作为参数传入。作业的实现逻辑需分析分片序号和分片参数，并以此为依据来调用具体的实现（比如一个批量处理数据库的作业，可以划分 0 号分片处理 1-10 号数据库，1 号分片可以处理 11-20 号数据库）。



基本原理图

2.3.2、Saturn 作业调度算法

（1）方案的设计

原理是给每个作业分片一个负载值和优先执行节点（prefer list），当需要重新分片时，参考作业优先设定和执行节点的负载值来进行域内节点之间的资源分配，从而达到资源平衡。

（2）前置条件

A：每个分片都引入一个负载值（load），由用户通过 Saturn UI 界面输入

B：为每一个作业引入新的属性 prefer list（优先列表，或者叫欲分配列表），由管理员通过 ui 界面编辑

C：作业引入启用状态（enabled/disabled），用户通过 UI 界面改变这个状态；启用状态的作业会被节点执行，且不可编辑、删除，不可对 prefer list 进行调整，禁用状态的作业不会被执行

（3）实施步骤

第一步，摘取；第二步，放回（将这些作业分片按照负载值从大到小顺序逐个分配给负载最小的执行节点）。

（3.1）executor 上线

摘取：

第一步，找出新上线节点的全部可执行作业列表；对于每个作业，判断 prefer list 中是否包含了新上线的节点；如果是，则摘取其中全部的分片；这些已经处理过的作业称为预处理作业；

第二步，从新上线节点的作业列表中减去预分配作业，然后使用以下的方法依次摘取：

1. 假如上线的 executor 为 a，它能处理的作业类型为 j1，j2（已减去预分配列表）。遍历当前域下的 executor 列表，拿掉全部作业类型为 j1，j2 的分片，加上尚未分配的 j1，j2 作业分片列表，作为算法的待分配列表
2. 在处理每个节点时，每拿掉一个作业分片后判断被拿掉的负载（load）是否已经超过了自身处理前总负载（load）的 1/n（n 为当前 executor 节点的总数量），如果超过，则本执行节点摘取完成，继续处理下一个执行节点；如果不超过则继续摘取，直到超过（大于等于）为止。

放回：

a．构造需要添加的作业分片列表，我们起名为待分配列表，长度为 n，待分配列表按照负载（load）从大到小排序，排序时需保证相同作业的所有分片时连续的

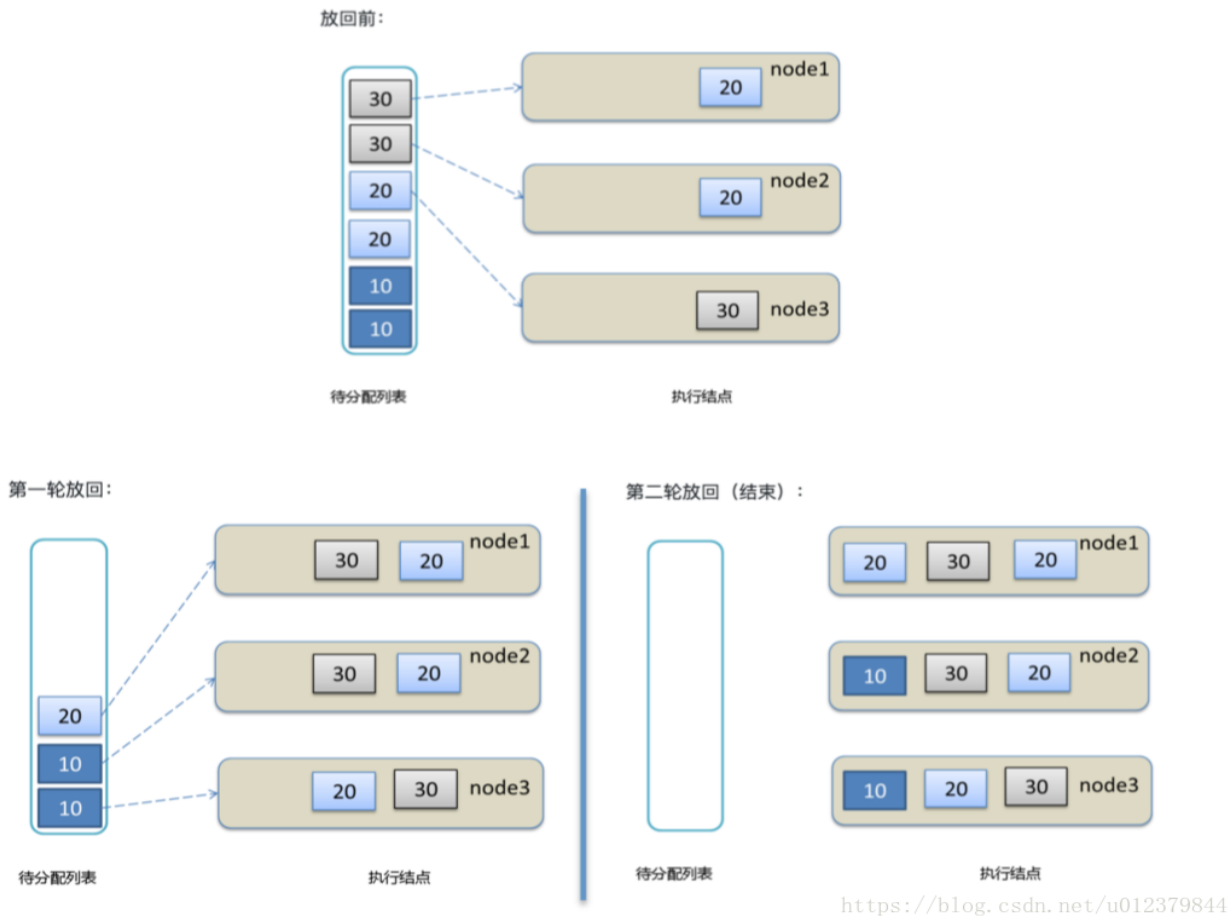
b．构造每种作业类型的 executor 列表（如果有 prefer list，且有存活，则该作业的 executor 列表就是 prefer list），得到一个 map<jobName,executorList>’

c．从待分配列表中依次取出第 0 到第 n-1 个作业分片 jobi

d．从 map 中取出可运行 jobi 的 executor 列表 listi

e．将 jobi 分配给 listi 中负载总和最小的 executor

举例如下：



（3.2）executor 下线

摘取：取出下线的 executor 当前分配到的全部作业分片，作为算法的待分配列表

放回：使用平衡算法逐个处理待分配列表中的作业分片

（3.3）作业启动

摘取：从所有 executor 中摘取将被启动作业的全部分片作为算法的待分配列表

放回：使用调整后的平衡算法放回

（3.4）作业停止

摘取：将被停止的作业分片从各节点删除

返回：无

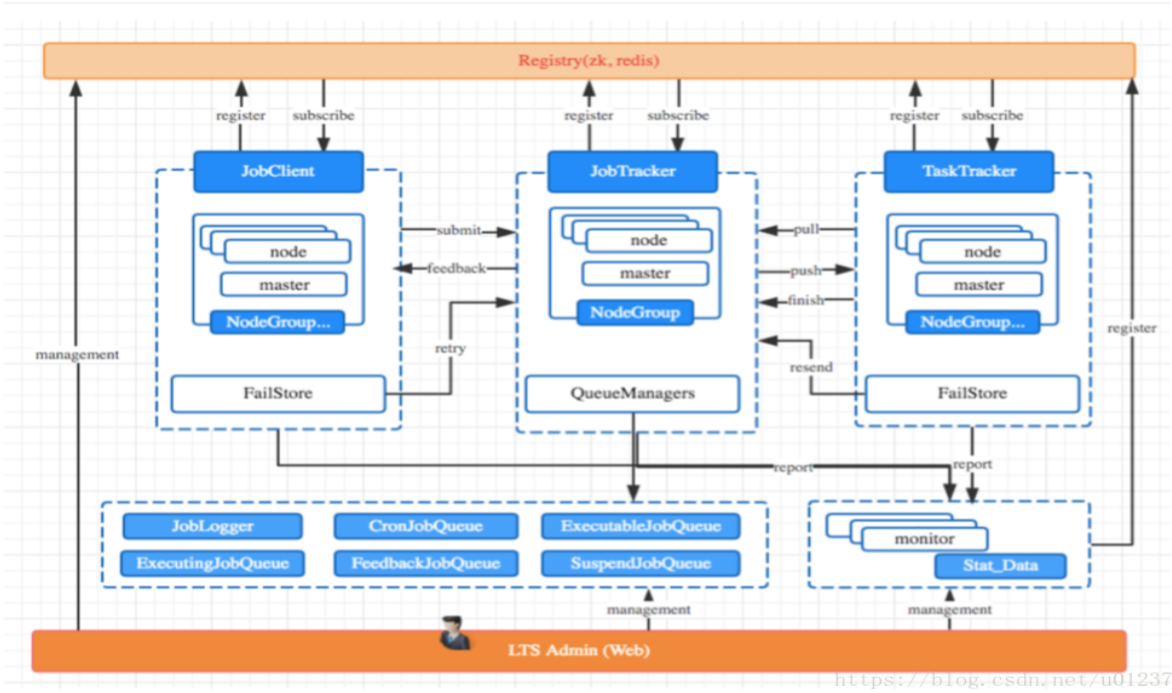
注：Saturn 架构文档请见 https://github.com/vipshop/Saturn/wiki/Saturn 架构文档

2.4、lts

LTS 是一个轻量级分布式任务调度框架，主要用于解决分布式任务的调度问题，支持实时任务、定时任务和 Cron 任务，有较好的伸缩性、扩展性以及健壮稳定性。他参考 hadoop 的思想，主要有以下四个节点：

1. JobClient：主要负责提交任务，并接收任务执行的反馈结果
2. JobTracker：负责接收并分配任务，任务调度
3. TaskTracker：负责执行任务，执行完反馈给 JobTracker
4. LTS-Admin：（管理后台）主要负责节点管理，任务队列管理，监控管理等

其中 JobClient、JobTracker、TaskTracker 是无状态的，可以部署多个并动态的进行删减，来实现负载均衡，实现更大的负载量，并且框架采用 FailStore 策略使得 LTS 具有很好的容错能力。



一个典型的定时任务，大概的执行流程如下：

1. 添加任务以后在注册中心进行注册，zk 集群会暴露各个节点的信息，进行 master 节点选举等
2. JobClient 将任务进行提交，如果成功的话将进行下一步；否则的话进入 FailStore，重试
3. JobTracker 接收并分配任务，如果任务已经存在，则结束；否则任务进入可执行队列 ExecutableJobQueue，接着进入执行中任务队列 ExecutingJobQueue，最后发送给 TaskTracker 进行执行

4. TaskTracker 执行完毕后，将结果反馈给客户端；如果反馈成功，则回到 JobClient 执行下一个任务；否则的话进入 FeedbackJobQueue 重试

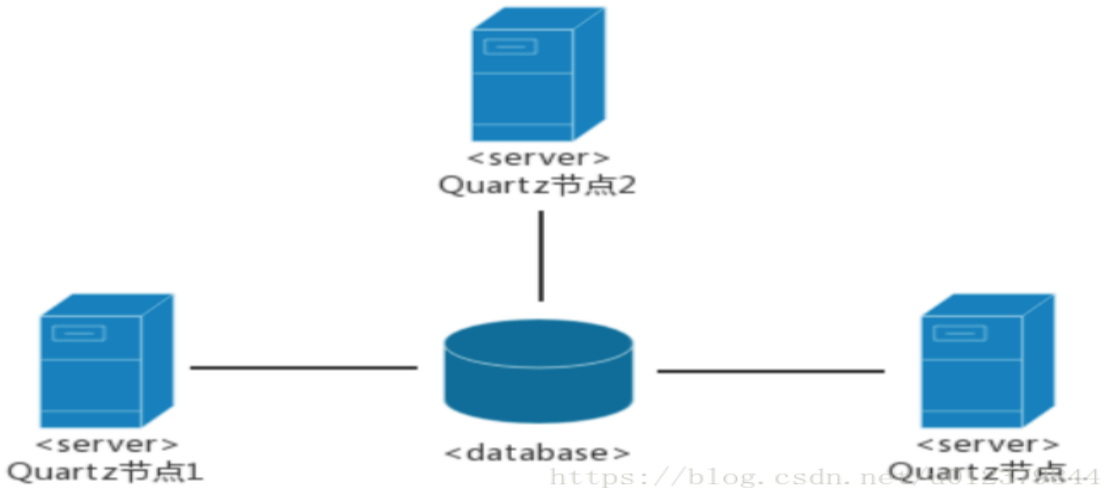
2.5、quartz

Quartz 是 OpenSymphony 开源组织在任务调度领域的一个开源项目，完全基于 java 实现。作为一个优秀的开源框架，Quartz 具有以下特点：强大的调度功能、灵活的应用方式、分布式和集群能力，另外作为 spring 默认的调度框架，很容易实现与 Spring 集成，实现灵活可配置的调度功能。

Quartz 的核心元素如下：

1. Scheduler：任务调度器，是实际执行任务调度的控制器
2. Trigger；触发器，用于定义任务调度的时间规则
3. Calendar：它是一些日历特定时间的集合，一个 Trigger 可以包含多个 Calendar，以便于排除或包含某些时间点
4. JobDetail：用来描述 Job 实现类及其他相关的静态信息，如 Job 的名字、关联监听器等信息
5. Job：是一个接口，只有一个方法 void execute(JobExecutionContext context)，开发者实现该接口定义运行任务，JobExecutionContext 类提供了调度上下文的各种信息

Quartz 的单机版大家应该都比较熟悉，它的集群方案是使用数据库来实现的。集群架构如下：

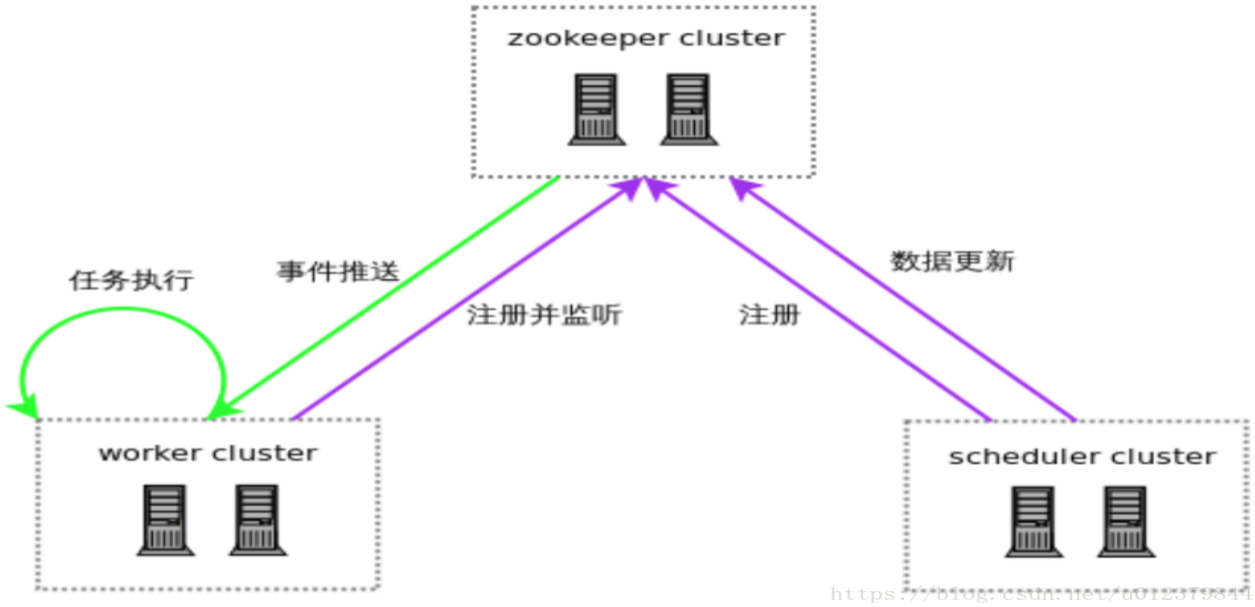


上图 3 个节点在数据库中都有同一份 Job 定义，如果某一个节点失效，那么 Job 会在其他节点上执行。因为每个节点上的代码都是一样的，那么如何保证只有一台机器上触发呢？答案是使用了数据库锁。在 quartz 集群解决方案了有张 scheduler_locks，采用了悲观锁的方式对 triggers 表进行了行加锁，以保证任务同步的正确性。

简单来说，quartz 的分布式调度策略是以数据库为边界的一种异步策略。各个调度器都遵守一个基于数据库锁的操作规则从而保证了操作的唯一性，同时多个节点的异步运行保证了服务的可靠。但这种策略有自己的局限性：集群特性对于高 CPU 使用率的任务效果特别好，但是对于大量的短任务，各个节点都会抢占数据库锁，这样就出现大量的线程等待资源。Quartz 的分布式只解决了任务高可用的问题，并没有解决任务分片的问题，还是会有单机处理的极限。

2.6、TBSchedule

TBSchedule 是一款非常优秀的分布式调度框架，广泛应用于阿里巴巴、淘宝、支付宝、京东、汽车之家等很多互联网企业的流程调度系统。TBSchedule 在时间调度方面虽然没有 quartz 强大，但是它支持分片的功能。和 quartz 不同的是，TBSchedule 使用 zk 来实现任务调度的高可用和分片。纯 java 开发。



TBSchedule 项目实际上可以分为两部分。1）schedule 管理控制台。负责控制、监控任务执行状态。2）实际执行 job 的客户端程序。在实际使用时，需要先启动 zk，然后部署 TBSchedule web 界面的管理控制台，最后启动实际执行 job 的客户端程序。这里的 zk 并不实际控制任务调度，它只是负责与 N 台执行 job 任务的客户端进行通讯，协调、管理、监控这些机器的运行信息。实际分配任务的是管理控制台，控制台从 zk 获取 job 的运行信息。TBSchedule 通过控制 ZNode 的创建、修改、删除来间接控制 job 的执行，执行任务的客户端监听它们对应 ZNode 的状态更新事件，从而达到 TBSchedule 控制 job 执行的目的。特点：

1. TBSchedule 的分布式机制是通过灵活的 Sharding 方式实现的，比如可以按所有数据的 ID 按 10 取模分片、按月份分片等，根据不同的场景由客户端配置分片规则。

2. TBSchedule 的宿主服务器可以进行动态的扩容和资源回收，这个特点主要是因为它后端依赖的 zooKeeper，这里的 zooKeeper 对于 TBSchedule 来说相当于 NoSQL，用于存储策略、任务、心跳等信息数据，他的数据结构类似于文件系统的目录结构，他的节点有临时节点、持久节点之分。一个新的服务器上线后，会在 zk 中创建一个代表当前服务器的一个唯一性路径（临时节点），并且新上线的服务器会和 zk 保持长连接，当通信断开后，节点会自动删除。
3. TBSchedule 会定时扫描当前服务器的数量，重新进行任务分配。
4. TBSchedule 不仅提供了服务端的高性能调度服务，还提供了一个 scheduleConsole war 随着宿主应用的部署直接部署到服务器，可以通过 web 的方式对调度的任务、策略进行监控管理，以及实时更新调整。

2.7、xxl-job

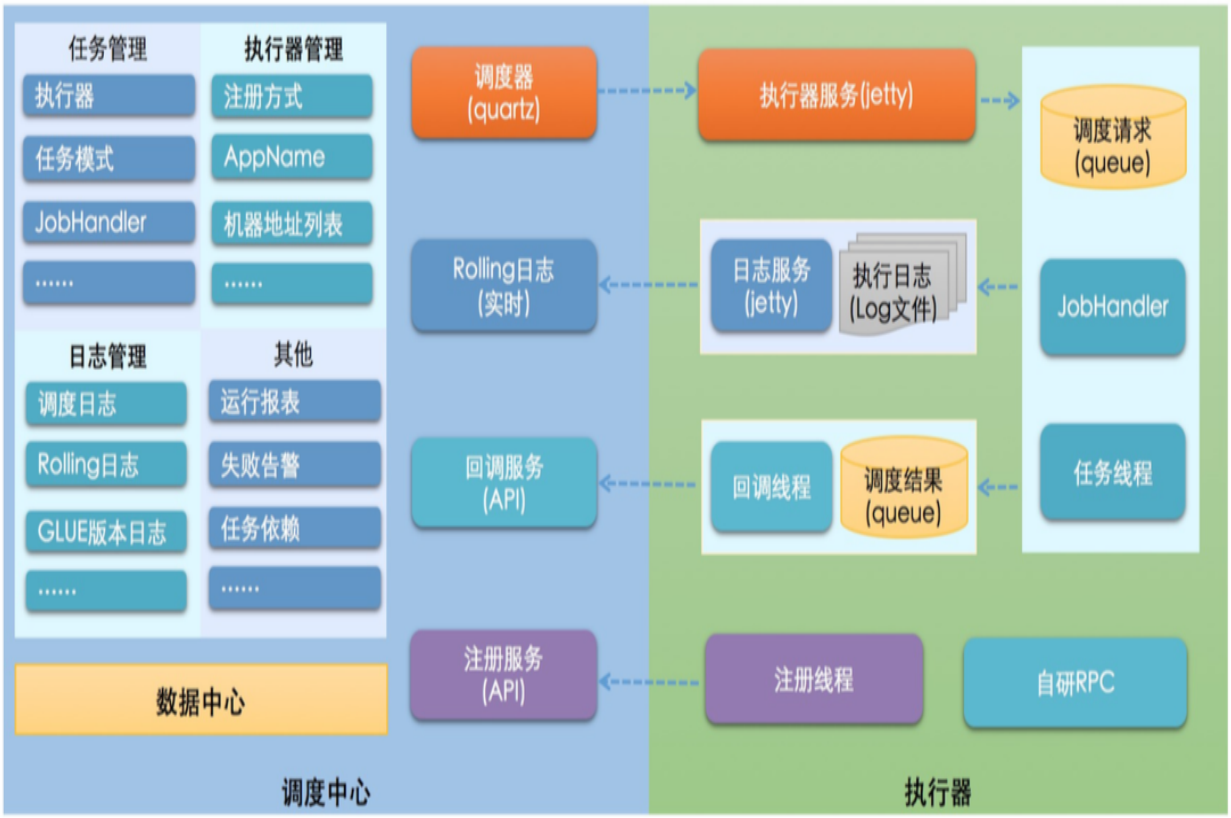
xxl-job 是一个轻量级的分布式任务调度框架，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。

xxl-job 的设计思想为：

（1）将调度行为抽象形成“调度中心”公共平台，而平台自身并不承担业务逻辑，“调度中心”负责发起调度请求

（2）将任务抽象成分散的 JobHandler，交由执行器统一管理，执行器负责接收调度请求并执行对应的 JobHandler 中业务逻辑

因此，“调度”和“任务”可以互相解偶，提高系统整体的稳定性和扩展性。



XXL-JOB架构图 v1.9 https://blog.csdn.net/u012379844

xxl-job 系统的组成分为：

（1）调度模块（调度中心）：负责管理调度信息，按照调度配置发出调度请求，自身不承担业务代码。调度系统与任务解耦，提高了系统可用性和稳定性，同时调度系统性能不再受限于任务模块；支持可视化、简单且动态的管理调度信息，包括任务新建，更新，删除，GLUE 开发和任务报警等，所有上述操作都会实时生效，同时支持监控调度结果以及执行日志，支持执行器 Failover。

（2）执行模块（执行器）：负责接收调度请求并执行任务逻辑。任务模块专注于任务的执行等操作，开发和维护更加简单和高效；接收“调度中心”的执行请求、终止请求和日志请求等。

Xxl-job 的执行流程：

首先准备一个将要执行的任务，任务开启后到执行器中注册任务的信息，加载执行器的配置文件，初始化执行器的信息，然后执行器 start。在 admin 端配置任务信息，配置执行器的信息。就可以控制任务的状态了。

xxl-job 的特性为：

1. 简单：支持通过 web 页面对任务进行 CRUD 操作，操作简单

2. 动态：支持动态修改任务状态、暂停/恢复任务，以及终止运行中的任务，即时生效
3. 调度中心 HA（中心式）：调度采用中心式设计，“调度中心”基于集群 Quartz 实现并支持集群部署，可保证调度中心 HA
4. 执行器 HA：任务分布式执行，任务执行器支持集群部署，可保证任务执行 HA
5. 注册中心：执行器会周期性自动注册任务并触发执行。同时，也支持手动录入执行器地址
6. 弹性扩容缩容：一旦有新的执行器机器上线或下线，下次调度时会重新分配任务
7. 路由策略：执行器集群部署时提供丰富的路由策略，包括：第一个、最后一个、轮询、随机、最不经常使用、故障转移等
8. 故障转移：任务路由策略选择"故障转移"情况下，如果执行器集群中某一台机器故障，将会自动 Failover 切换到一台正常的执行器发送调度请求。
9. 阻塞处理策略：调度过于密集执行器来不及处理时的处理策略，策略包括：单机串行、丢弃后续调度、覆盖之前调度
10. 任务超时控制：支持自定义任务超时时间，任务运行超时将会主动中断任务；
11. 任务失败重试：支持自定义任务失败重试次数，当任务失败时将会按照预设的失败重试次数主动进行重试；
12. 失败处理策略；调度失败时的处理策略，默认提供失败告警、失败重试等策略；
13. 分片广播任务：执行器集群部署时，任务路由策略选择"分片广播"情况下，一次任务调度将会广播触发集群中所有执行器执行一次任务，可根据分片参数开发分片任务；
14. 动态分片：分片广播任务以执行器为维度进行分片，支持动态扩容执行器集群从而动态增加分片数量，协同进行业务处理；在进行大数据量业务操作时可显著提升任务处理能力和速度。
15. 事件触发：除了"Cron 方式"和"任务依赖方式"触发任务执行之外，支持基于事件的触发任务方式。调度中心提供触发任务单次执行的 API 服务，可根据业务事件灵活触发。
16. 任务进度监控：支持实时监控任务进度；
17. Rolling 实时日志：支持在线查看调度结果，并且支持以 Rolling 方式实时查看执行器输出的完整的执行日志；
18. GLUE：提供 Web IDE，支持在线开发任务逻辑代码，动态发布，实时编译生效，省略部署上线的过程。支持 30 个版本的历史版本回溯。
19. 脚本任务：支持以 GLUE 模式开发和运行脚本任务，包括 Shell、Python、NodeJS 等类型脚本；
20. 任务依赖：支持配置子任务依赖，当父任务执行结束且执行成功后将会主动触发一次子任务的执行, 多个子任务用逗号分隔；

21.一致性：“调度中心”通过 DB 锁保证集群分布式调度的一致性, 一次任务调度只会触发一次执行；

22.自定义任务参数：支持在线配置调度任务入参，即时生效；

23.调度线程池：调度系统多线程触发调度运行，确保调度精确执行，不被堵塞；

24.数据加密：调度中心和执行器之间的通讯进行数据加密，提升调度信息安全性；

25.邮件报警：任务失败时支持邮件报警，支持配置多邮件地址群发报警邮件；

26.推送 maven 中央仓库: 将会把最新稳定版推送到 maven 中央仓库, 方便用户接入和使用；

27.运行报表：支持实时查看运行数据，如任务数量、调度次数、执行器数量等；以及调度报表，如调度日期分布图，调度成功分布图等；

28.全异步：系统底层实现全部异步化，针对密集调度进行流量削峰，理论上支持任意时长任务的运行；

29.国际化：调度中心支持国际化设置，提供中文、英文两种可选语言，默认为中文；

> xxl-job-lite 的执行器实际是一个 ConcurrentHashMap 容器。

3、任务调度框架的技术选型？

1、Quartz：Java 事实上的定时任务标准，但是关注点在于定时任务而非数据，虽然实现了高可用，但是缺少分布式并行调度的功能，性能低。

2、TBSchedule：阿里早期开源的分布式任务调度系统。代码略陈旧，使用的是 Timer 而不是线程池执行任务调度。TBSchedule 的作业类型比较单一，只能是获取/处理数据一种模式，文档缺失比较严重。

3、详见分布式调度框架对比表格～

4、分布式任务调度框架的安装与使用？

4.1、Elastic-job

1、环境准备：

jdk1.7+、zookeeper3.4.6+、maven3.0.4+

2、安装 zookeeper3.4.12 并启动

```
localhost:~ guolujie$ brew install zookeeper
==> Downloading https://homebrew.bintray.com/bottles/zookeeper-3.4.12.high_
######################################################################## 10
==> Pouring zookeeper-3.4.12.high_sierra.bottle.tar.gz
==> Caveats
To have launchd start zookeeper now and restart at login:
  brew services start zookeeper
Or, if you don't want/need a background service you can just run:
  zkServer start
==> Summary
🍺  /usr/local/Cellar/zookeeper/3.4.12: 242 files, 32.9MB
localhost:~ guolujie$ brew services start zookeeper
==> Successfully started `zookeeper` (label: homebrew.mxcl.zookeeper)
```


这里 zookeeper 占用了 2181 端口。

```
localhost:~ guolujie$ brew services start zookeeper
==> Successfully started `zookeeper` (label: homebrew.mxcl.zookeeper)
localhost:~ guolujie$ zkCli
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is enabled
[zk: localhost:2181(CONNECTING) 0]
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
```


3、创建简单任务

添加依赖：

```
<dependency>
    <groupId>com.dangdang</groupId>
    <artifactId>elastic-job-lite-core</artifactId>
    <version>2.1.5</version>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-client</artifactId>
    <version>2.11.1</version>
</dependency>
```

写一个简单的任务：

```
public class MyElasticJob implements SimpleJob {
    public void execute(ShardingContext context){
        switch (context.getShardingItem()){
            case 0:{
                System.out.println("MyElastic-Job - 0");
                break;
            }
            case 1:{
                System.out.println("MyElastic-Job - 1");
                break;
            }
            case 2:{
                System.out.println("MyElastic-Job - 2");
                break;
            }
            default:{
                System.out.println("MyElastic-Job - default");
            }
        }
    }
}
```


在项目入口处添加作业的配置和 zk 的配置：

```
private static CoordinatorRegistryCenter createRegistryCenter() {
    CoordinatorRegistryCenter regCenter = new ZookeeperRegistryCenter(new
            ZookeeperConfiguration( serverLists: "localhost:2181",  namespace: "elastic-job-demo"));
    regCenter.init();
    return regCenter;
}

private static  LiteJobConfiguration createJobConfiguration() {
    // 定义作业核心配置
    JobCoreConfiguration simpleCoreConfig = JobCoreConfiguration.newBuilder( jobName: "demoSimpleJob",  cron: "0/15 * * * * ?",  shardingTotalCount: 10).build();

    // 定义SIMPLE类型配置
    SimpleJobConfiguration simpleJobConfig = new SimpleJobConfiguration(simpleCoreConfig, MyElasticJob.class.getCanonicalName());

    // 定义Lite作业根配置
    LiteJobConfiguration simpleJobRootConfig = LiteJobConfiguration.newBuilder(simpleJobConfig).build();
    return simpleJobRootConfig;
}

@Bean
public CommandLineRunner commandLineRunner() {
    return (String... args) -> {
        new JobScheduler(createRegistryCenter(), createJobConfiguration()).init();
    };
}
```


运行，得到结果：

4、下载 Elastic-job-lite 源码，使用 maven 进行打包。在 elastic-job-lite/elastic-job-lite-console/target/elastic-job-lite-console-3.0.0.M1-SNAPSHOT/中，然后解压，会有 start.bat 和 start.sh 两个脚本，启动。

浏览器中输入 localhost:8899，就可以管理任务了。

4.2、xxl-job-lite

1、调度数据库初始化，tables_xxl-job.sql

2、下载源码：包括调度中心+公共依赖+执行器示例

3、配置部署“调度中心”：修改数据库配置——将项目进行打包——将 xxl-job-admin 包部署到 tomcat 上

4、输入 localhost:8080/xxl-job-admin 即可访问调度中心

5、配置部署执行器：xxl-job-executor-sample-springboot 打成 jar 包直接运行，其他的打成 war 包部署在 tomcat 上。

6、写一个任务，运行，去执行器上进行注册，然后调度中心配置执行器信息，添加任务

附录

1、etcd

etcd 是一个开源的、分布式的键值对数据存储系统，提供共享配置、服务的注册和发现。etcd 内部采用 raft 协议作为一致性算法，是基于 Go 语言实现的。

2、zookeeper

zookeeper 是一个开源的分布式协调服务，它为分布式应用提供了高效且可靠的分布式协调服务，提供了诸如统一命名空间服务、配置服务和分布式锁等分布式基础服务。

3、分布式锁

假如我们由三台机器，每台机器上都有一个进程。假设我们在第一台机器上挂载了一个资源，三个进程都要来竞争这个资源。我们不希望这三个进程同时来访问，那么就需要有一个协调器，来让他们有序的对该资源进行访问。这个协调器就是我们所说的那个锁，比如说“进程 1”在使用该资源的时候，就会先去获得锁，“进程 1”就对该资源保持独占，这样其他的进程就无法访问该资源。“进程 1”用完该资源后就会将锁释放掉，让其他的进程来获得锁。因此这个锁机制就能保证我们的进程有序的访问该资源。就称作为“分布式锁”，是分布式协调技术实现的核心内容

4、分片

任务的分布式执行，需要将一个任务拆分为多个独立的任务项，然后由分布式的服务器分别执行某一个或几个分片项。

5、单点故障

通常分布式系统采用主从模式，就是一个主控机连接多个处理节点。主节点负责分发任务，从节点负责处理任务，当我们的主节点发生故障时，那么整个系统就瘫痪了，这就叫做单点故障。

传统的解决办法：

就是准备一个备用节点，这个备用节点定期给当前主节点发送 ping 包，主节点收到 ping 包后向备用节点发送回复 Ack，当备用节点收到回复后就会认为主节点还活着，让他继续提供服务。

当主节点挂了，那么备用节点就收不到 Ack 回复了，然后备用节点就代替它成为了主节点。

但是存在一个安全隐患，那就是当发生网络故障时，备用节点收不到主节点的回复 Ack，他会认为主节点死了，它会代替主节点成为新的主节点。

zookeeper 解决方案：

在引入了 zookeeper 后我们启用了两个主节点，A 和 B 启动后他们都会去 Zookeeper 去注册一个节点，假设 A 注册的节点为 master-01，B 注册的节点为 master-02，注册完之后进行选举，编号最小的节点将被选举为主节点。

如果 A 挂了，它在 zookeeper 注册的节点将会被自动删除，Zookeeper 感知到节点的变化，然后再次发出选举，这时候 B 将获胜成为新的主节点。如果 A 恢复了，它会去 zookeeper 再注册一个节点，编号为 master-03。这时 zookeeper 感知到节点的变化，会再次发起选举，此时还是 B 胜出。那么 B 继续担任主节点，A 则成为备用节点。

6、Mesos

——像用一台电脑一样使用整个数据中心

是 Apache 下的开源分布式资源管理框架，它被称为分布式系统的内核，是以与 Linux 内核同样的原则而创建的，不同点仅仅是在于抽象的层面。使用 ZooKeeper 实现 Master 和 Slave 的容错。

7、FailStore 策略

FailStrore，顾名思义就是 Fail and Store，这个主要是用于失败了存储的，主要用于节点容错，当远程数据交互失败后，存储在本地，等待远程通讯恢复后，再将数据进行提交。