当当分布式定时任务

1. 普通定时任务的问题

随着系统应用的越来越大，我们经常会将一个大的系统按功能分成多个模块，各个模块中常常要定时执行一些任务。比如，订单系统的超时状态判断、缓存数据的定时更新、定式给用户发邮件，甚至是一些定期计算的业务报表等等。常见的处理方式有线程的while和sleep组合、使用Java自带的Timer定时器触发任务，或者是使用quartz框架。这些都是单节点的解决方案，在分布式环境中，他们无法达到以下要求：

1. 高可用HA：单机版的定时任务调度只能在一台机器上运行，如果程序或者系统出现异常就会导致功能不可用。甚至整个系统的崩溃。虽然程序本身可以实现的足够稳定，但始终有可能遇到非程序引起的故障，比如停电，或者磁盘损坏而导致机器宕机。而这对于一个系统的核心功能来说是不可接受的。

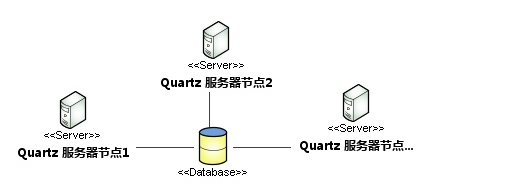
2、单机处理极限：原本1分钟内需要处理1万个订单，但是随着业务量的加大，现在可能需要1分钟内处理10万个订单；原来一个统计需要1小时，现在业务方需要10分钟就统计出来。你也许会说，你也可以多线程、单机多进程处理。不可否认的是多线程并行处理可以提高单位时间的处理效率，但是单机能力毕竟有限，这些限制主要是单机CPU、内存和磁盘，因此始终会有单机处理不过来的情况。

这个时候就需要分布式的定时任务来实现了。业内常用的分布式定式任务解决方案主要有quartz、淘宝的TBSchedule和当当的elastic-job。

1. 常见三种方案的对比

2.1 Quartz解决方案

Quartz的单机版本配置比较容易，功能也比较强大，他也有一个针对集群的解决方案，其集群方案是使用数据库来实现的。集群架构如下：



从上图中看到，一个集群中三个工作节点都有同一份Quartz job的定义，他们都和同一个数据库交互，如果集群中的某一个节点失效，那么这个Job会转移到其他节点上从新执行。由于三个节点上的Job执行代码是一样的，故同一时刻一个任务只能在三个节点的其中一个执行，Quartz使用数据库锁来达到这个目标。具体来说就是在Quartz的集群解决方案里有张表scheduler\_locks，quartz采用了悲观锁（排它锁）的方式对triggers表进行行加锁，以保证任务同步的正确性。一旦某一个节点上面的线程获取了该锁，那么这个Job就会在这台机器上被执行，同时这个锁就会被这台机器占用。同时另外一台机器也会想要触发这个任务，但是锁已经被占用了，就只能等待，直到这个锁被释放。之后会看trigger状态，如果已经被执行了，则不会执行了。

故Quartz的分布式调度策略是以数据库为边界资源的一种异步策略。各个调度器都使用了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：

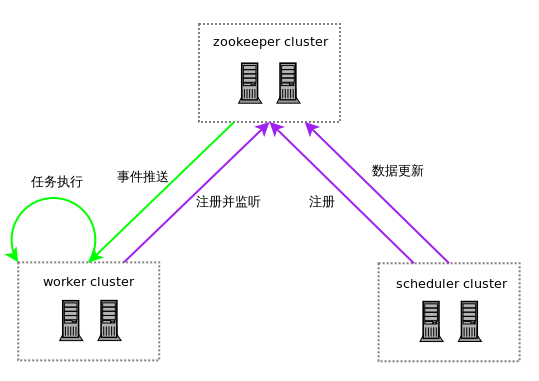
1、集群特性对于高CPU使用率的任务效果很好，但是对于大量的短任务，各个节点都会抢占数据库锁，这样就出现大量的线程等待资源。这种情况随着节点的增加会越来越严重。

2、Quartz的分布式只是解决了高可用的问题，节点失效任务仍可以转移到其它地方执行，但是同一时刻一个任务还是只能在单个节点上执行，故还是会有单机处理的极限问题。

2.2 TBSchedule

TBSchedule是一个支持分布式的调度框架，能让一种批量任务或者不断变化的任务，被动态的分配到多个主机的JVM中，不同的线程组中并行执行。基于ZooKeeper的纯Java实现，由Alibaba开源。这里的ZooKeeper对于TBSchedule来说是一个NoSQL，用于存储策略、任务、心跳信息数据，集群中添加的一个新的节点会在ZooKeeper中创建一个代表当前服务器的一个唯一性路径（临时节点）。

TBSchesule对分布式的支持包括Scheduler的分布式和Worker的分布式，其网络部署架构图如下：



Worker和Scheduler均以ZooKeeper为注册中心，所有数据以节点及节点内容的形式注册，通过定时汇报主机状态保持存活在ZooKeeper上。通过ZooKeeper来实现任务调度的高可用和分片。它有以下特点：

优点：

* 宿主服务器可以实现动态的扩容和资源回收
* 提供web端的调度界面，可通过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
* 任务分片，同一个任务可以拆分成多个部分在不同的节点同时执行，提高任务的执行效率和集群利用率

缺点：

* 时间调度表达式不支持quartz的时间表达式的格式，不够灵活
* 底层任务的执行使用timer而非线程池执行任务调度，不够高效
* 框架提供的作业类型较为单一

2.3 Elastic-Job

Elastic-Job是当当开源的分布式调度解决方案，由两个相互独立的子项目Elastic-Job-Lite和Elastic-Job-Cloud组成。Elastic-Job-Lite定位为轻量级无中心化解决方案，使用jar包的形式提供分布式任务的协调服务。这种方式对原有代码的侵入性比较小，所以一般我们只要使用Elastic-Job-Lite就好。

Elastic-Job-Lite没有宿主程序，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。任务的唯一性依赖于zookeeper节点控制。Elastic-Job-Lite的分布式部署全靠ZooKeeper来同步状态和原数据。实现高可用的任务只需将分片总数设置为1，并把开发的Jar包部署于多个服务器上执行，任务将会以1主N从的方式执行。一旦本次执行任务的服务器崩溃，其他执行任务的服务器将会在下次作业启动时选择一个替补执行。如果开启了失效转移，那么功能效果更好，可以保证在本次作业执行时崩溃，备机之一立即启动替补执行。

其有以下特点：

* 分布式调度协调：Zookeeper保证节点的有效性
* 弹性扩容缩容：同TBSchedule一样，通过zookeeper实现
* 失效转移：节点失效后，分片任务可以转移到其他节点执行
* 错过执行作业重触发：错过执行的任务会被加入队列，被重新触发
* 作业分片一致性：保证同一分片在分布式环境中仅一个执行实例
* 支持并行调度：
* 支持作业生命周期操作
* 丰富的作业类型：Simple、DataFlow、Script
* Spring整合以及命名空间提供
* 运维平台

elastic-job结合了quartz非常优秀的时间调度功能，并且利用ZooKeeper实现了灵活的分片策略。除此之外，还加入了大量实用的监控和管理功能，以及其开源社区活跃、文档齐全等优点，是分布式任务调度框架的推荐选择。

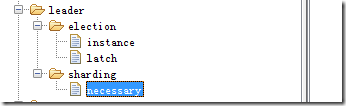
1. Job分片原理

数据分片的目的在于把一个任务分散到不同的机器上运行，既可以解决单机计算能力上限的问题，也能降低部分任务失败对整体系统的影响。elastic-job并不直接提供数据处理的功能，框架只会将分片项分配至各个运行中的作业服务器（其实是Job实例，部署在一台机器上的多个Job实例也能分片），开发者需要自行处理分片项与真实数据的对应关系。框架也预置了一些分片策略：平均分配算法策略，作业名哈希值奇偶数算法策略，轮转分片策略。同时也提供了自定义分片策略的接口。

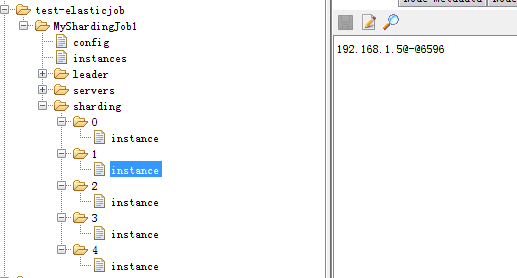
elastic-job的分片是通过zookeeper来实现的。分片的分片由主节点分配，如下三种情况都会触发主节点上的分片算法执行：

* 新的Job实例加入集群
* 现有的Job实例下线，如果下线的是leader节点，那么先选举然后触发分片算法的执行
* 主节点选举

上述三种情况，会让zookeeper上leader节点的sharding节点上多出来一个necessary的临时节点，主节点每次执行Job前，都会去看一下这个节点，如果有则执行分片算法。



分片的执行结果会存储在zookeeper上，如下图，5个分片，每个分片应该由哪个Job实例来运行都已经分配好。分配的过程就是上面触发分片算法之后的操作。分配完成之后，各个Job实例就会在下次执行的时候使用上这个分配结果。



每个job实例任务触发前都会获取本任务在本实例上的分片情况（直接和上图zookeeper上instance节点比对某一个分片是否该有这个Job实例执行），然后封装成shardingContext，传递给调用任务的实际执行方法execute执行。

综上所述，分片的目的就是把一个任务分散到不同的机器上运行，它解决了单机计算能力上限的问题、降低部分任务失败对整体系统的影响，elastic-job自带了一些分片策略：平均分配算法、作业名哈希值奇偶数算法、轮转分片。提供自定义分片策略的接口分片策略：平均分配算法、作业名哈希值奇偶数算法、轮转分片。并且也提供自定义分片策略的接口，供使用者扩展。

1. 作业的调度

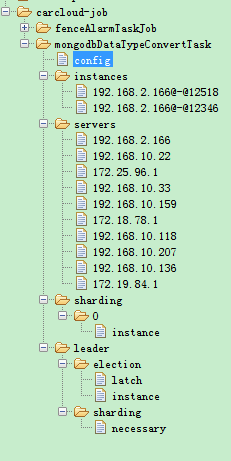
elastic-job中作业的调度任务的执行需要包含两大步骤：任务的配置和任务的注册

4.1 任务的配置

由于elastic-job内部使用quartz作为任务调度框架，任务的配置的相关的基础信息也是和quartz一致的。elastic-job的任务配置类在quartz的基础上额外封装了分片策略，监控作业相关参数以及与注册中心的时间误差秒数等配置项。

4.2 任务的注册

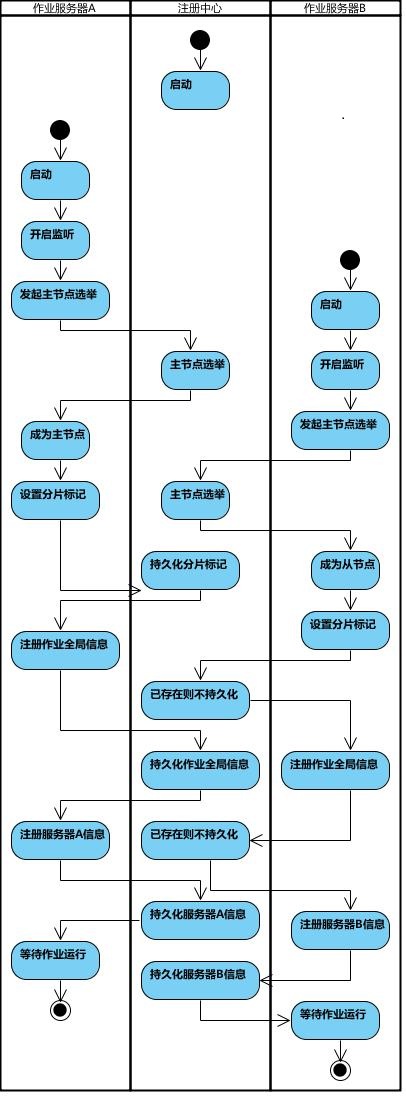
elastic-job是通过zookeeper进行任务协调和故障转移的，任务的注册也就是把任务注册到zookeeper里面去。任务的注册包含在任务的启动过程中。根节点是项目的名称，下面一级是任务的名称。鉴于Zookeeper节点不支持修改的操作，故任务一旦创建则不能修改任务的名称，如果修改名称将视为新的任务，创建新的节点。任务名称节点下又包含5个数据子节点，分别是config, instances, leader, servers和sharding。如下图：



各节点的作业如下：

* config节点：任务的配置信息

包括执行任务的类、cron表达式、分片算法、分片数、分片参数等

* instances节点：同一个Job下的elastic-job的部署实例，是临时节点，作业实例上线时注册，下线时自动清理。命名:IP+@-@+PID
* leader节点：通过zookeeper的主节点选举，选出来的主节点信息

election：下面的instance节点显式了当前主节点的实例ID

latch节点也是一个永久节点用于选举时候的实现分布式锁

sharding节点下面有一个临时节点：necessary，是否需要重新分片的标记

* servers节点：任务实例的信息，任务实例的IP地址
* sharding节点：任务的分片信息

子节点是分片项序号，从零开始，至分片总数减一。每个分片项下的子节点用于控制和记录分片运行状态。分片计算的结果也就体现在这instance上

4.3 任务的启动

任务的启动过程，就是任务实例和zookeeper进行交互的过程。每个实例在启动过程中，会把自身的信息注册到zookeeper中去。并完成选举和分片策略的设置，也就是完成上文一些zookeeper节点的创建和持久化。

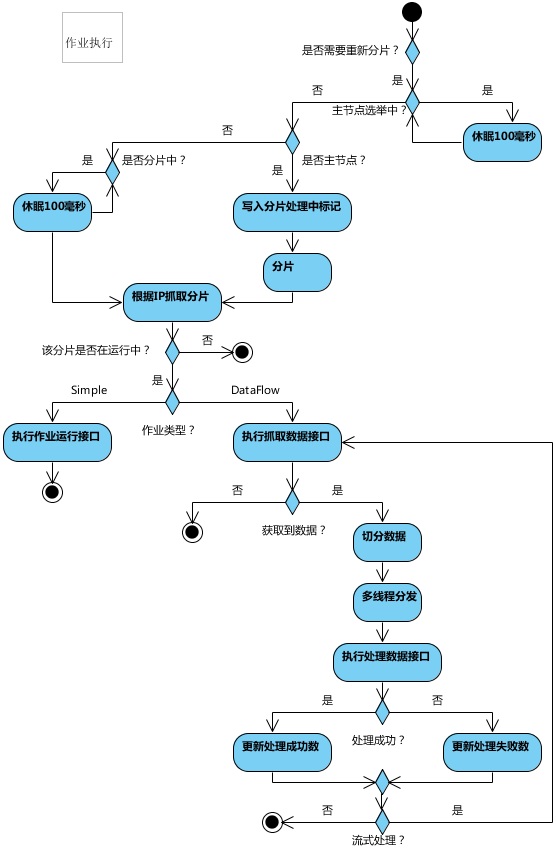
上图是单个任务的启动过程

4.4 任务的执行

任务的执行依赖于quartz job的触发。elastic-job的LiteJob类继承自quartz的Job类，在任务触发的时候，添加额外的逻辑处理。

LiteJob把任务的执行分为执行前，执行中（业务代码的执行）和执行后三个阶段。在执行前阶段中主要实现分片策略的执行，记录事件，执行监听事件等等。而执行后阶段主要处理错过执行的相关任务以及执行监听事件。我们可以在这三个作业的生命周期中添加自己的处理逻辑。

整个执行流程图如下：



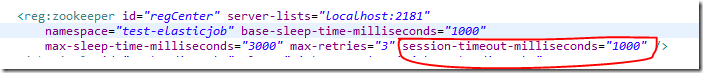
上图中画出了两种作业类型的执行过程，需要注意的是，无论哪种作业类型，在执行之前，都需要进行主节点选取和Job分片的步骤。分片完以后，执行具体的作业，这里有一点区别是，SimpleJob是一种简单类型作业，未经任何封装的类型。SimpleJob接口仅提供单一方法用于覆盖，此方法将定时执行。Dataflow类型用于处理数据流，需实现DataflowJob接口。该接口提供2个方法可供覆盖，分别用于抓取(fetchData)和处理(processData)数据。这里的fetchData方法的返回值为null或集合长度为空时，作业才停止抓取，否则作业将一直运行下去。

1. 失效转移的实现

所谓失效转移，就是指如果在任务执行过程中有一个执行实例挂了，那么之前被分配到这个实例的任务（或者分片）会在下次任务执行之前被重新分配到其他正常节点实例上执行。

5.1 简单的HA

当某一个任务实例节点宕机，它就会断开与zookeeper的连接，这会触发elastic-job主节点的重新分片逻辑。elastic-job启动任务节点以后生成的zookeeper中的instance节点是一个临时节点。Zookeeper的临时节点能在任务实例出现问题与zookeeper断开以后，触发zookeeper的节点移除的事件，从而重新调整分区或者运行节点。既然是临时节点，就可以在zookeeper中配置sessionTimeoutMs参数。在使用spring的elastic-job配置中在如下地方配置：



这里的session-timeout-milliseconds参数配置的就是当前节点与zookeeper的连接超时时间，当超过这个时间，zookeeper没有收到节点的信息，他就会认为该节点已经失效，从而触发删除临时节点，这样在下次任务执行时，就会触发任务的重新分片，从而达到失效转移，但是需要注意的是，如果任务的执行频率比session-timeout-milliseconds时间短，那么可能会出现任务分片丢失的情况。所以这并不是一种真正的实时的失效转移策略。

5.2 复杂的失效转移

elastic-job的任务配置有个failover，如果开启设置为true的时候，会启动真正的失效转移：，elastic-job的任务又两个配置failover（默认值为false）和monitorExecution（默认值是true）。只有对monitorExecution为true的情况下才可以开启失效转移。

IMG_256

elastic-job监听的是zk的instance节点删除事件。如果任务配置了failover等于true，其中某个instance与zk失去联系或被删除，并且失效的节点又不是本身，就会触发失效转移逻辑。

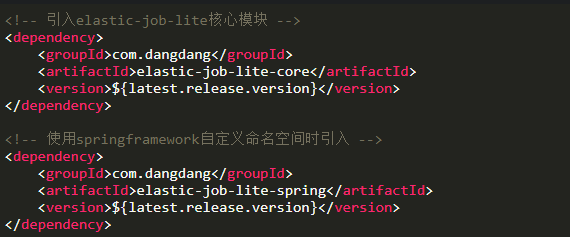
通过上面的配置，任务就会失效转移，即在执行任务的过程中遇见异常的情况，这个分片任务可以在其他节点再次执行。这个和上面的HA不同，对于HA，如果任务终止，那么不会在其他任务实例上再次重新执行。等到下次重新分片才会再次执行。

其具体过程如下：

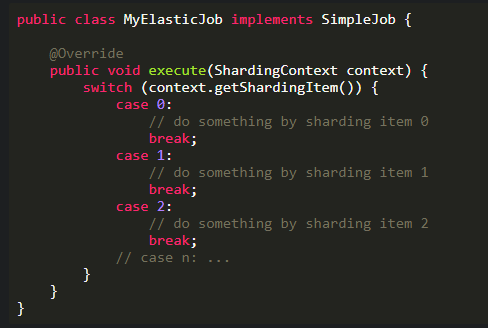
* leader节点：下面创建failover节点以及items节点
* 存活着的多个任务实例收到zk节点丢失的事件
* 某个任务实例选举获得执行权
* sharding节点的分片上添加failover节点
* 任务重新执行，相应的节点和数据删除

1. 环境搭建与任务实现

6.1 引入Maven依赖：

elastic-job-lite-core、elastic-job-lite-spring

6.2 实现具体的接口，管理具体数据和分片的关系

6.3 Spring命名空间的配置

