**轻量级分布式任务调度框架**

**使用说明文档**

# 历史记录

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **版本** | **编制** | | **审核** | | **批准** | | **更改记录** |
| **发布人** | **日期** | **审核人** | **日期** | **批准人** | **日期** |
| V0.1 | 潘洪亮 | 2016.3.2 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

# 目录

[历史记录 2](#_Toc444702176)

[目录 3](#_Toc444702177)

[1 分布式作业lts 4](#_Toc444702178)

[1.1 系统概述 4](#_Toc444702179)

[1.2 开发环境 4](#_Toc444702180)

[1.2.1 所需环境 4](#_Toc444702181)

[1.2.2 配置本地仓库 4](#_Toc444702182)

[1.2.3 编译 5](#_Toc444702183)

[1.2.4 运行 5](#_Toc444702184)

[1.3 部署 6](#_Toc444702185)

[1.4 运行 6](#_Toc444702186)

[1.5 开发业务作业 6](#_Toc444702187)

[1.5.1 自动扫描,派发作业 6](#_Toc444702188)

[1.5.2 Lts-admin界面手动添加作业 8](#_Toc444702189)

[1.5.3 Sql语句插入作业 8](#_Toc444702190)

[2 配置中心 8](#_Toc444702191)

[3 Mq rpc 8](#_Toc444702192)

[4 消息推送 8](#_Toc444702193)

[5 缓存 8](#_Toc444702194)

# 分布式作业lts

## 系统概述

<https://github.com/ltsopensource/light-task-scheduler>

readme.md 看几遍理解概念原理

light-task-scheduler\docs\OLD\_README.md 这个里边也有些重要信息

我的fork版本

https://github.com/hongliangpan/light-task-scheduler

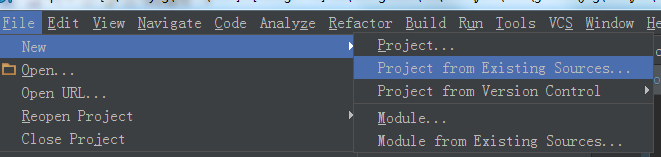
## 开发环境

### 所需环境

Zookeeper或redis作为注册中心【启动zkServer.cmd】

Mysql 数据库

Git、Idea，导入工程，选择父pom文件



### 配置本地仓库

父pom.xml

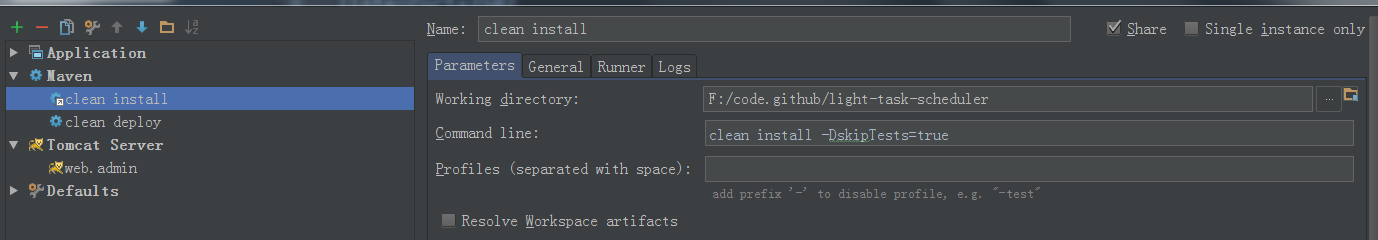
|  |
| --- |
| <distributionManagement>  <repository>  <id>glodon-nexus-releases</id>  <name>Glodon Nexus Release Repository</name>  <url>http://maven.glodon.com/nexus/content/repositories/releases/</url>  </repository>  <snapshotRepository>  <id>glodon-nexus-snapshots</id>  <name>Glodon Nexus Snapshot Repository</name>  <url>http://maven.glodon.com/nexus/content/repositories/snapshots/</url>  </snapshotRepository>  </distributionManagement> |

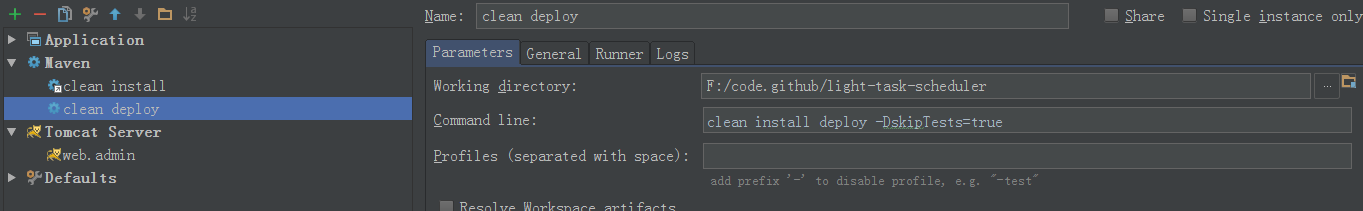
### 编译

clean install

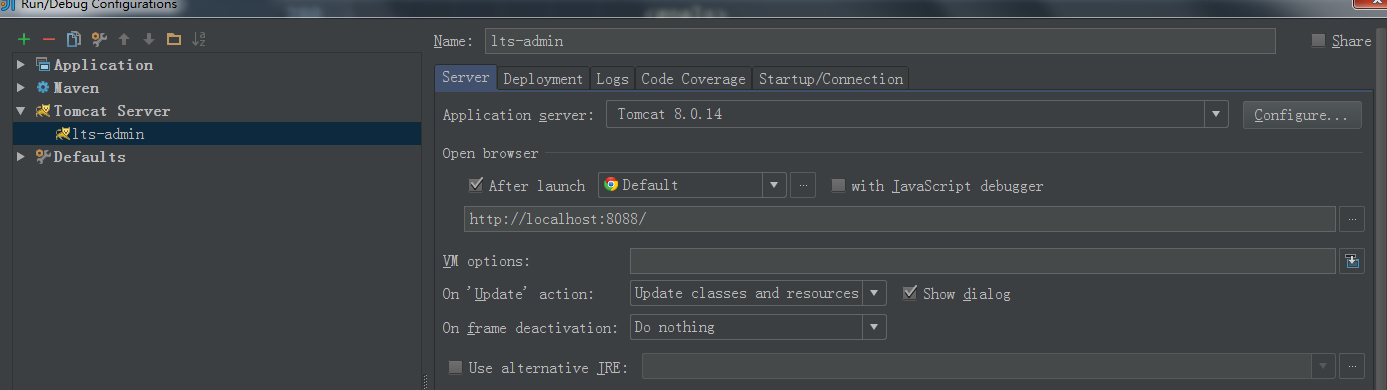
发布到本地仓库

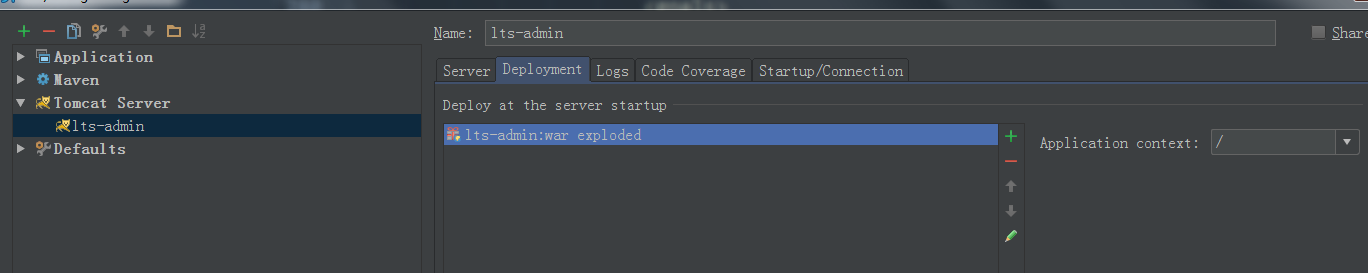
clean install deploy -DskipTests=true





Web.admin





### 运行

1. 自定义配置文件

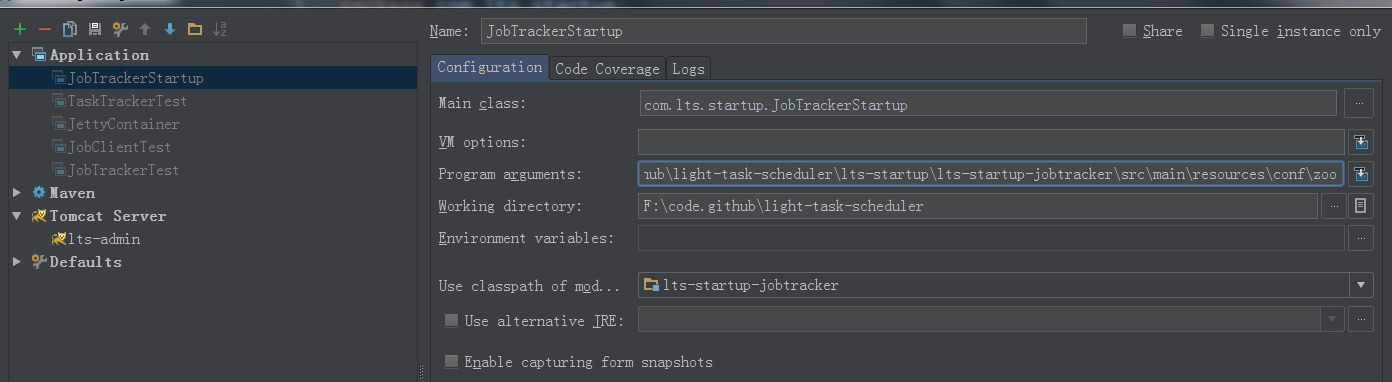
修改配置文件，如数据库连接参数;

conf/zoo/jobtracker.cfg

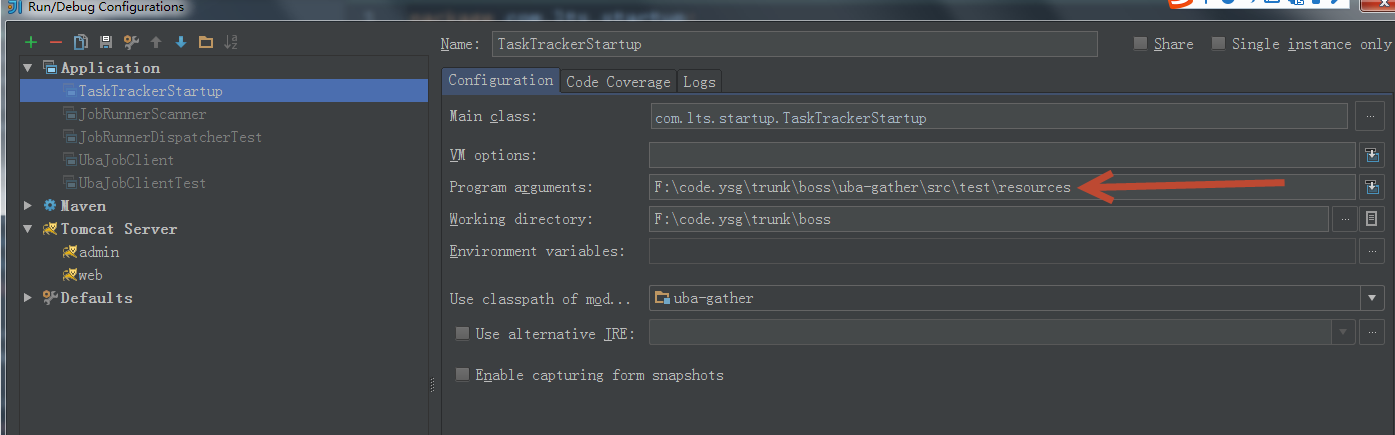
lts-admin.cfg

1. 运行 tomcat lts-admin
2. 运行JobTrackerStartup

第一个参数：为配置文件目录



1. TaskTrackerStartup+ tasktracker.cfg拷贝到业务工程运行



## 部署

运行 build\_all.cmd

lts推荐的部署方式 jobtracker独立部署 jobclient集成到业务模块 tasktracker也集成

## 运行

|  |
| --- |
| sh ./bin/jobtracker.sh zoo start  sh ./bin/lts-admin.sh start  sh ./tasktracker/bin/tasktracker.sh start |

查看日志

lts-bin/logs

lts-bin/tasktracker/logs

## 开发业务作业

### 自动扫描,派发作业

|  |
| --- |
| // 作业开发部署  /\*  自动扫描作业，并派发[根据注解的type参数进行派发]  1. 配置JobRunnerDispatcher为JobRunner  tasktracker.cfg文件中  # JobRunner 任务执行类  jobRunnerClass=com.lts.tasktracker.jobdispatcher.JobRunnerDispatcher  2. 开发新的作业  //添加 类注解 @RunnerTask(type= "type")  @JobRunnerAnnotation(type= "type1")  public class GatherMetricJobRunner implements JobRunner  3.自动扫描作业JobRunnerDispatcher,配置扫描的package  static {  try {  JobRunnerScanner.scan("com.glodon.ysg", JOB\_RUNNER\_MAP);  } catch (Exception e) {  LOGGER.error(e.getMessage(),e);  }  }  4. 自行开发的作业，把jar包，放入tasktracker\lib  重启tasktracker，就自动运行  5. 配置参数  # JobRunner 任务执行类  jobRunnerClass=com.lts.tasktracker.jobdispatcher.JobRunnerDispatcher  # 自动扫描作业包，逗号分隔  jobRunnerScannerPackages=com.glodon.ysg,com.lts.job  # 是否内嵌JobClient  isEmbedJobClient=true  # 提交作业类，初始化一些作业，可以通过admin-web修改作业信息  jobSubmitterClass=com.lts.tasktracker.jobdispatcher.UbaJobSubmitter  # 自定义的cron，可以通过admin-web修改作业信息  mySubmitCronExpression=0 53/10 2-3 \* \* ? |

### Lts-admin界面手动添加作业

### Sql语句插入作业

需要插入数据到lts\_cron\_job\_queue和lts\_executable\_job\_queue

job\_id相同

|  |
| --- |
| INSERT INTO `lts\_cron\_job\_queue` (`id`, `job\_id`, `priority`, `retry\_times`, `task\_id`, `gmt\_created`, `gmt\_modified`, `submit\_node\_group`, `task\_tracker\_node\_group`, `ext\_params`, `is\_running`, `task\_tracker\_identity`, `need\_feedback`, `cron\_expression`, `trigger\_time`) VALUES ('1', '07418C5F7DF705AD78505BBF864FDF7B', '100', '0', 'gaq\_function\_infos', '1456972127562', '1456972127562', 'uba\_taskTracker', 'uba\_taskTracker', '{\"product\":\"gaq\",\"type\":\"function\_infos\",\"size\":\"300\"}', '0', NULL, '0', '0 30 10-18 \* \* ?', NULL); |

|  |
| --- |
| INSERT INTO `lts\_executable\_job\_queue\_uba\_tasktracker` (`id`, `job\_id`, `priority`, `retry\_times`, `task\_id`, `gmt\_created`, `gmt\_modified`, `submit\_node\_group`, `task\_tracker\_node\_group`, `ext\_params`, `is\_running`, `task\_tracker\_identity`, `need\_feedback`, `cron\_expression`, `trigger\_time`) VALUES ('1', '07418C5F7DF705AD78505BBF864FDF7B', '100', '0', 'gaq\_function\_infos', '1456972128156', '1456972128156', 'uba\_taskTracker', 'uba\_taskTracker', '{\"product\":\"gaq\",\"type\":\"function\_infos\",\"size\":\"300\"}', '0', NULL, '0', '0 30 10-18 \* \* ?', '1456972200000'); |

# Lts知识库[使用问题记录]

## 作者博客

http://huguifuture.iteye.com/blog/2190274

## Can not found JobTracker

adminweb界面添加任务 ,错误提示: Can not found JobTracker.

lts-admin、jobtracker、 tasktracker、jobclient集群名称需要保持一致

|  |
| --- |
| # 集群名称 clusterName=uba\_cluster |

配置文件的参数获取

|  |
| --- |
| Config  for (Map.Entry<String, String> config : cfg.getConfigs().entrySet()) {  jobTracker.addConfig(config.getKey(), config.getValue()); }  public void setParameter(String key, String value) {  parameters.put(key, value); } |

## 发送作业,接收作业

JobPusher JobSender

JobPushProcessor

# 系统架构

## 框架概况

LTS 有主要有以下四种节点：

* JobClient：主要负责提交任务, 并接收任务执行反馈结果。
* JobTracker：负责接收并分配任务，任务调度。
* TaskTracker：负责执行任务，执行完反馈给JobTracker。
* 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。

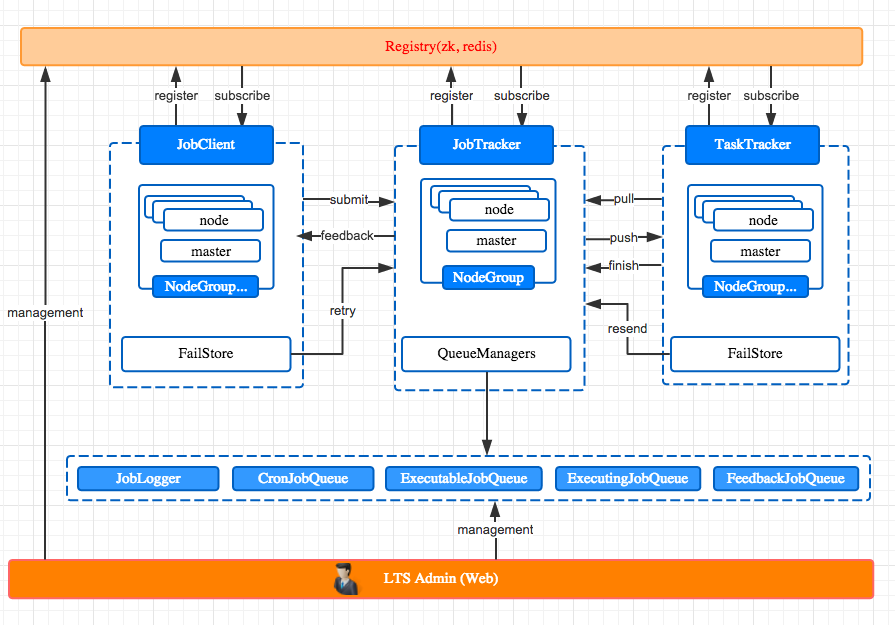
其中JobClient，JobTracker，TaskTracker节点都是无状态的。 可以部署多个并动态的进行删减，来实现负载均衡，实现更大的负载量, 并且框架采用FailStore策略使LTS具有很好的容错能力。

LTS注册中心提供多种实现（Zookeeper，redis等），注册中心进行节点信息暴露，master选举。(Mongo or Mysql)存储任务队列和任务执行日志, netty or mina做底层通信, 并提供多种序列化方式fastjson, hessian2, java等。

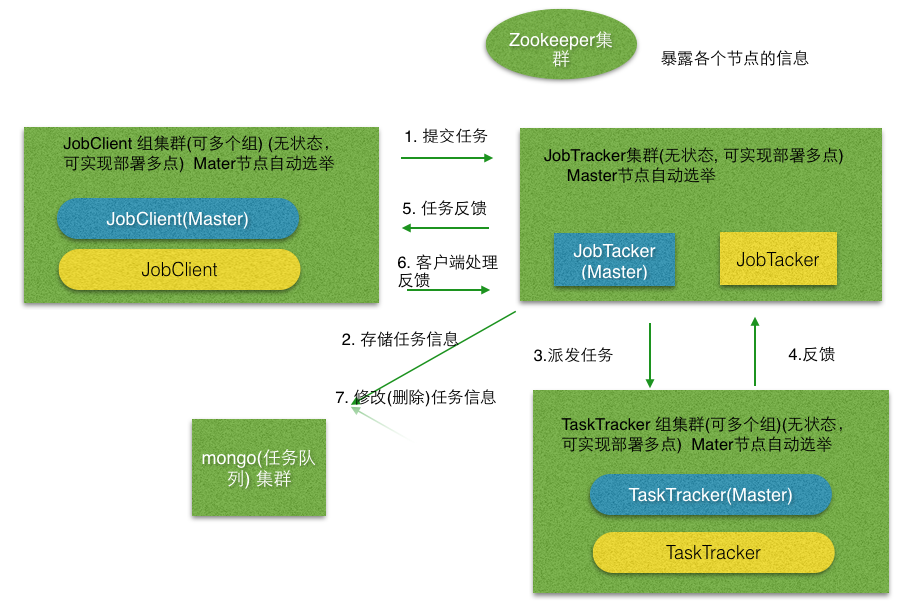
LTS支持任务类型：

* 实时任务：提交了之后立即就要执行的任务。
* 定时任务：在指定时间点执行的任务，譬如 今天3点执行（单次）。
* Cron任务：CronExpression，和quartz类似（但是不是使用quartz实现的）譬如 0 0/1 \* \* \* ?

## 架构图



## 架构图



## 概念说明

### 节点组

1. 英文名称 NodeGroup,一个节点组等同于一个小的集群，同一个节点组中的各个节点是对等的，等效的，对外提供相同的服务。
2. 每个节点组中都有一个master节点，这个master节点是由LTS动态选出来的，当一个master节点挂掉之后，LTS会立马选出另外一个master节点，框架提供API监听接口给用户。

### 节点组:

* 1. 一个节点组等同于一个集群，同一个节点组中的各个节点是对等的，外界无论连接节点组中的任务一个节点都是可以的。
* 2. 每个节点组中都有一个master节点(master宕机，会自动选举出新的master节点)，框架会提供接口API来监听master节点的变化，用户可以自己使用master节点做自己想做的事情。
* 3. JobClient和TaskTracker都可以存在多个节点组。譬如 JobClient 可以存在多个节点组。 譬如：JobClient 节点组为 ‘lts\_WEB’ 中的一个节点提交提交一个 只有节点组为’lts\_TRADE’的 TaskTracker 才能执行的任务。
* 4. (每个集群中)JobTacker只有一个节点组。
* 5. 多个JobClient节点组和多个TaskTracker节点组再加上一个JobTacker节点组, 组成一个大的集群。

### FailStore

1. 顾名思义，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之后，存储在本地，等待远程通信恢复的时候，再将数据提交。
2. FailStore主要用户JobClient的任务提交，TaskTracker的任务反馈，TaskTracker的业务日志传输的场景下。
3. FailStore目前提供几种实现：leveldb,rocksdb,berkeleydb,mapdb,ltsdb，用于可以自由选择使用哪种,用户也可以采用SPI扩展使用自己的实现。

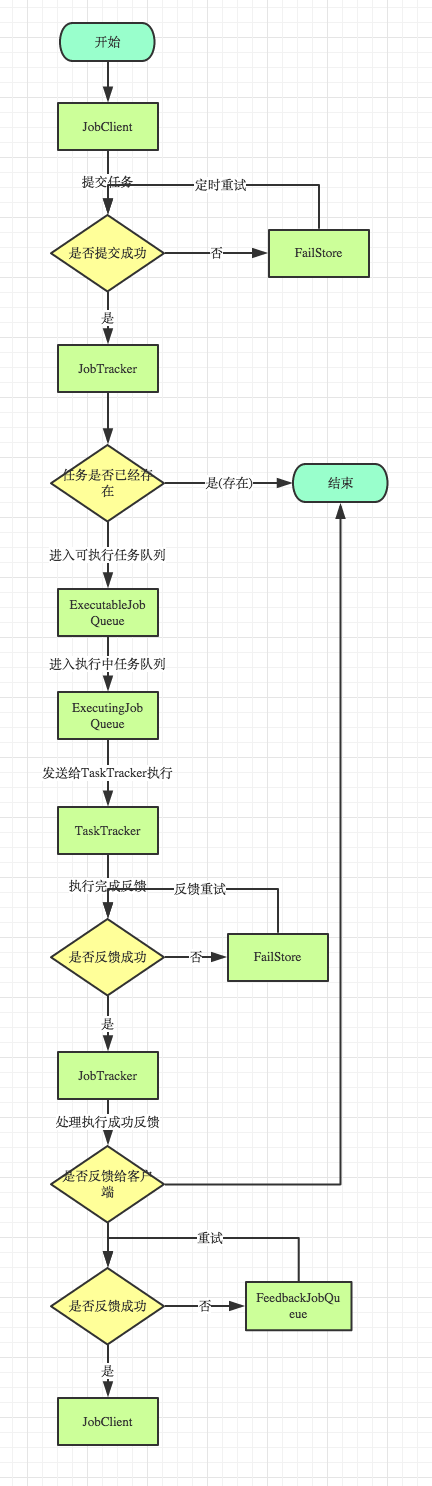
## 流程

### 工作流程

* 1. JobClient 提交一个 任务 给 JobTracker, 这里我提供了两种客户端API, 一种是如果JobTracker 不存在或者提交失败，直接返回提交失败。另一种客户端是重试客户端, 如果提交失败，先存储到本地FailStore(可以使用NFS来达到同个节点组共享leveldb文件的目的,多线程访问,已经做了文件锁处理)，返回给客户端提交成功的信息，待JobTracker可用的时候，再将任务提交。
* 2. JobTracker收到JobClient提交来的任务,将任务存入任务队列。JobTracker等待TaskTracker的Pull请求，然后将任务Push给TaskTracker去执行。
* 3. TaskTracker收到JobTracker分发来的任务之后，然后从线程池中拿到一个线程去执行。执行完毕之后，再反馈任务执行结果给JobTracker（成功or 失败[失败有失败错误信息]），如果发现JobTacker不可用，那么存储本地FailStore，等待JobTracker可用的时候再反馈。反馈结果的同时，询问JobTacker有没有新的任务要执行。
* 4. JobTacker收到TaskTracker节点的任务结果信息。根据任务信息决定要不要反馈给客户端。不需要反馈的直接删除,需要反馈的,直接反馈,反馈失败进入FeedbackQueue, 等待重新反馈。
* 5. JobClient收到任务执行结果，进行自己想要的逻辑处理。

### 流程图

下图是一个标准的实时任务执行流程。



## 特性

### 1、Spring支持

LTS可以完全不用Spring框架，但是考虑到很用用户项目中都是用了Spring框架，所以LTS也提供了对Spring的支持，包括Xml和注解，引入lts-spring.jar即可。

### 2、业务日志记录器

在TaskTracker端提供了业务日志记录器，供应用程序使用，通过这个业务日志器，可以将业务日志提交到JobTracker，这些业务日志可以通过任务ID串联起来，可以在LTS-Admin中实时查看任务的执行进度。

### 3、SPI扩展支持

SPI扩展可以达到零侵入，只需要实现相应的接口，并实现即可被LTS使用，目前开放出来的扩展接口有

1. 对任务队列的扩展，用户可以不选择使用mysql或者mongo作为队列存储，也可以自己实现。
2. 对业务日志记录器的扩展，目前主要支持console，mysql，mongo，用户也可以通过扩展选择往其他地方输送日志。

### 4、故障转移

当正在执行任务的TaskTracker宕机之后，JobTracker会立马将分配在宕机的TaskTracker的所有任务再分配给其他正常的TaskTracker节点执行。

### 5、节点监控

可以对JobTracker，TaskTracker节点进行资源监控，任务监控等，可以实时的在LTS-Admin管理后台查看，进而进行合理的资源调配。

### 6、多样化任务执行结果支持

LTS框架提供四种执行结果支持，EXECUTE\_SUCCESS，EXECUTE\_FAILED，EXECUTE\_LATER，EXECUTE\_EXCEPTION，并对每种结果采取相应的处理机制，譬如重试。

* EXECUTE\_SUCCESS: 执行成功,这种情况，直接反馈客户端（如果任务被设置了要反馈给客户端）。
* EXECUTE\_FAILED：执行失败，这种情况，直接反馈给客户端，不进行重试。
* EXECUTE\_LATER：稍后执行（需要重试），这种情况，不反馈客户端，重试策略采用1min，2min，3min的策略，默认最大重试次数为10次，用户可以通过参数设置修改这个重试次数。
* EXECUTE\_EXCEPTION：执行异常, 这中情况也会重试(重试策略，同上)

### 7、FailStore容错

采用FailStore机制来进行节点容错，Fail And Store，不会因为远程通信的不稳定性而影响当前应用的运行。具体FailStore说明，请参考概念说明中的FailStore说明。