Table of Contents

LTS介绍

> LTS(light-task-scheduler)主要用于解决分布式任务调度问题，支持实时任务，定时任务，Cron任务，Repeat任务。有较好的伸缩性，扩展性，健壮稳定性而被多家公司使用，同时也希望开源爱好者一起贡献。

LTS介绍

> LTS(light-task-scheduler)主要用于解决分布式任务调度问题，支持实时任务，定时任务，Cron任务，Repeat任务。有较好的伸缩性，扩展性，健壮稳定性而被多家公司使用，同时也希望开源爱好者一起贡献。

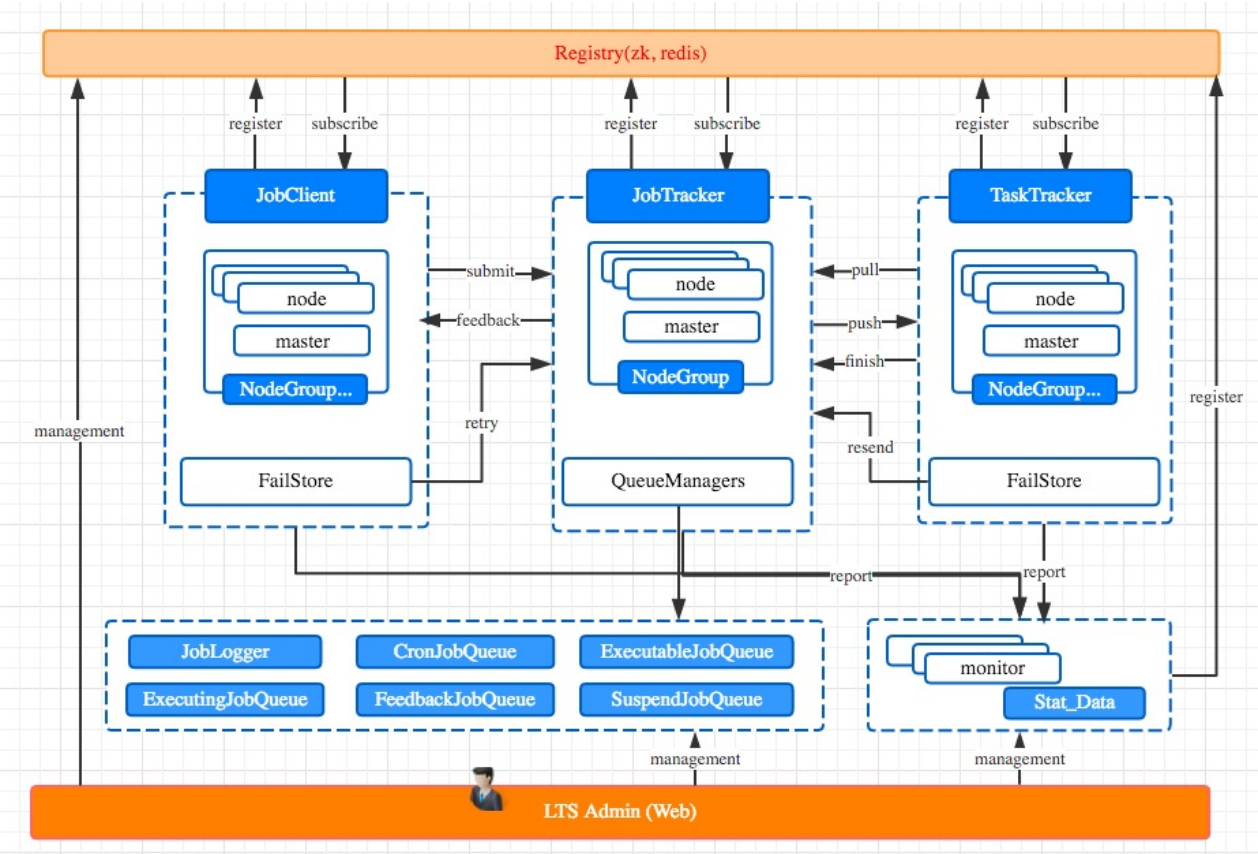
主要功能

1. 支持分布式，解决多点故障，支持动态扩容，容错重试等
2. Spring扩展支持，SpringBoot支持，Spring Quartz Cron任务的无缝接入支持
3. 节点监控支持，任务执行监控支持，JVM监控支持
4. 后台运维操作支持, 可以动态提交，更改，停止 任务

项目地址

- github地址:https://github.com/ltsopensource/light-task-scheduler
- oschina地址:http://git.oschina.net/hugui/light-task-scheduler
- oschina项目收录地址:http://www.oschina.net/p/lts

这两个地址都会同步更新。感兴趣，请加QQ群：109500214 一起探讨、完善。越多人支持，就越有动力去更新，喜欢记得右上角star哈。


Registry(zk, redis)
register
subscribe
register
subscribe
register
subscribe
JobClient
JobTracker
TaskTracker
node
master
NodeGroup...
submit
feedback
node
master
NodeGroup
pull
push
finish
node
master
NodeGroup...
management
retry
resend
register
FailStore
QueueManagers
FailStore
report
report
JobLogger
CronJobQueue
ExecutableJobQueue
ExecutingJobQueue
FeedbackJobQueue
SuspendJobQueue
monitor
Stat_Data
management
management
LTS Admin (Web)

1.1 LTS技术架构

> LTS 着力于解决分布式任务调度问题，将任务的提交者和执行者解耦，解决任务执行的单点故障，支持动态扩容，出错重试等机制。代码程序设计上，参考了优秀开源项目Dubbo，Hadoop的部分思想。

LTS目前支持四种任务：

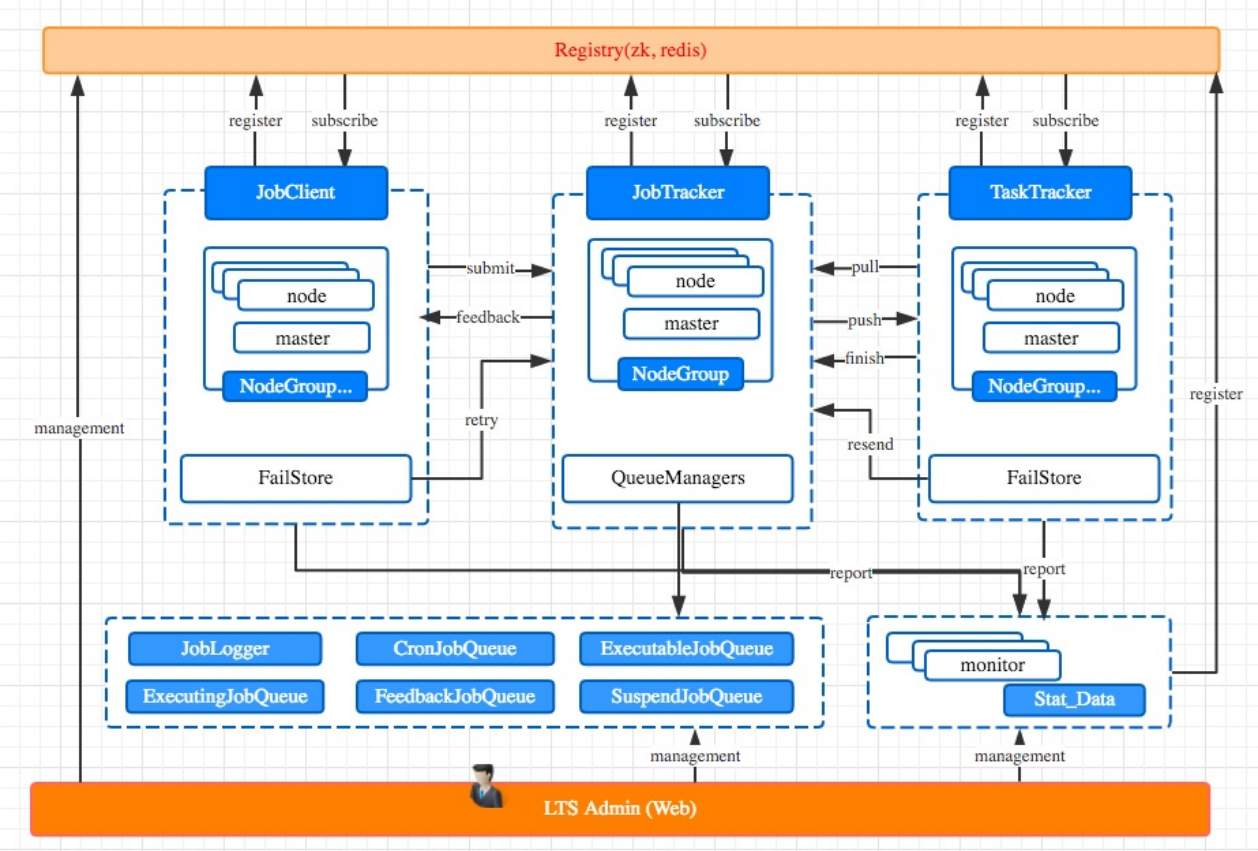
- 实时任务：提交了之后立即就要执行的任务。
- 定时任务：在指定时间点执行的任务，譬如 今天3点执行（单次）。
- Cron任务：CronExpression，和quartz类似（但是不是使用quartz实现的）譬如 0 0/1 * ?
- Repeat任务：譬如每隔5分钟执行一次，重复50次就停止。

架构设计上，**LTS**框架中包含以下五种类型的节点：

- `JobClient` :主要负责提交任务, 并接收任务执行反馈结果。
- `JobTracker` :负责任务调度，接收并分配任务。
- `TaskTracker` :负责执行任务，执行完反馈给JobTracker。
- `LTS-Monitor` :主要负责收集各个节点的监控信息，包括任务监控信息，节点JVM监控信息
- `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。

LTS的这五种节点都是无状态的，都可以部署多个，动态扩容，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。

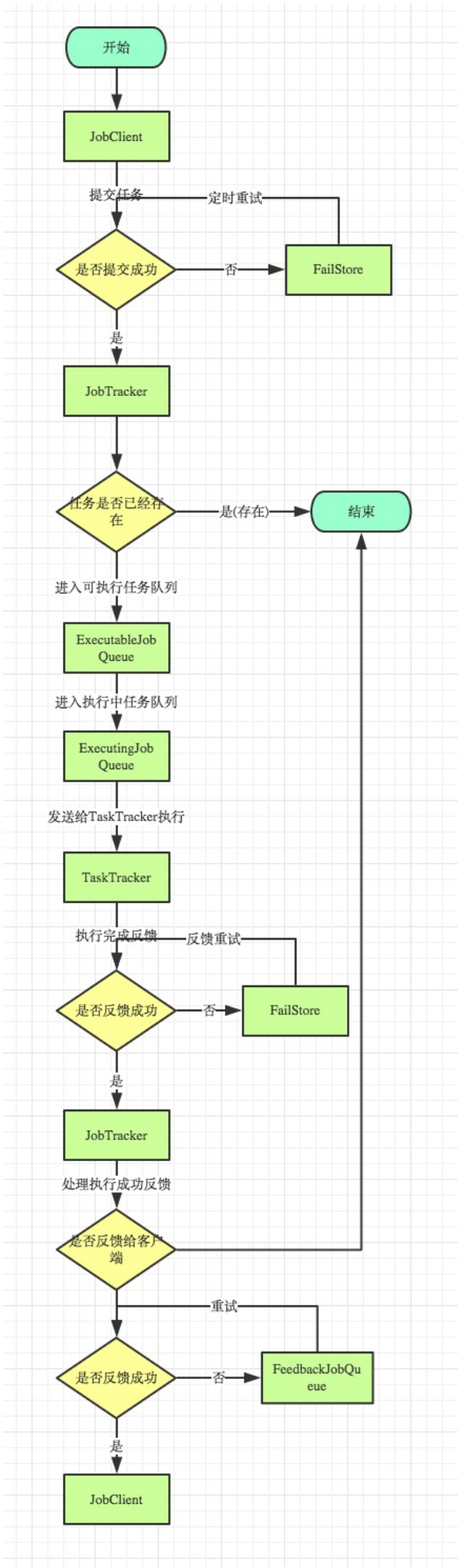
架构图



- **Registry**： 注册中心，LTS提供多种实现，目前支持 `zookeeper` （推荐）和 `redis`，主要用于LTS的节点信息暴露和master节点选举。
- **FailStore**：失败存储，主要用于在部分场景远程RPC调用失败的情况，采取现存储本地KV文件系统，待远程通信恢复的时候再进行数据补偿。目前FailStore场景，主要有 `RetryJobClient` 提交**任务失败的时候，存储FailStore； `TaskTracker` 返回任务执行结果给 `JobTracker` 的失败 时候，FailStore； `TaskTracker` 提交 `BizLogger` 的失败的时候，存储FailStore. 目前FailStore有四种实现：leveldb，rocksdb，berkeleydb，mapdb（当然用户也可以实现扩展接口实现自己的FailStore）
- **QueueManager**：任务队列，目前提供mysql（推荐）和mongodb两种实现（同样的用户可以自己扩容展示其他的，譬如oracle等），主要存储任务数据和任务执行日志等。
- **RPC**：远程RPC通信框架，目前也支持多种实现，LTS自带有netty和mina，用户可以自行选择，或者自己SPI扩展实现其他的。
- **NodeGroup**：节点组，同一个节点组中的任何节点都是对等的，等效的，对外提供相同的服务。譬如 `TaskTracker` 中有10个nodeGroup都是 `send_msg` 的节点组，专门执行发送短信的任务。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。
- **ClusterName**：LTS集群，就如上图所示，整个图就是一个集群，包含LTS的五种节点。

LTS执行流程

1.实时任务执行流程。


开始
JobClient
提交任务
定时重试
是否提交成功
否
FailStore
是
JobTracker
任务是否已经存在
是(存在)
结束
进入可执行任务队列
ExecutableJob Queue
进入执行中任务队列
ExecutingJob Queue
发送给TaskTracker执行
TaskTracker
执行完成反馈
反馈重试
是否反馈成功
否
FailStore
是
JobTracker
处理执行成功反馈
是否反馈给客户端
重试
是否反馈成功
否
FeedbackJobQueue
是
JobClient

2.Cron任务执行流程

3.Repeat任务执行流程

LTS特性

1、Spring/Spring Boot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30s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些参数。
> - 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。

8、动态扩容

> 因为LTS各个节点都是无状态的，所以支持动态增加删除节点，达到负载均衡的目的

和其他解决方案的比较

主要根据LTS支持的几种任务（实时任务、定时任务、Cron任务，Repeat任务）和其他一些开源框架在应用场景上做比较。

实时任务，实时执行

> 这种场景下，当任务量比较小的时候，单机都可以完成的时候，自己采用线程池或者去轮训数据库取任务的方式（或者其他方式）就可以解决，但如果是任务执行时间比较长或者任务量比较大，单机不足以满足需求的时候，就需要自己去做分布式的功能，还有很重要的是，怎么做容错，怎么保证同一个任务只被一个节点执行，怎么解决执行失败异常的情形等等，你就需要自己去做很多事情，头可能就大了。这时候 LTS 就派上用场了，因为这些问题 LTS 都帮你解决了，你只需关注你的业务逻辑，而不用为上面的这些事情而烦恼。当然这时候有人可能会想到如果用 MQ 去解决，利用 MQ 的异步去解耦，也同时可以实现分布还有容错等。当然有时候是可以的，为什么说是可以的呢，因为 LTS 的架构也和 MQ 的类似， JobClient 相当于 MQ 的 Producer , JobTracker 相当于 MQ 的 Broker , TaskTracker 相当于 MQ 的 Consumer ，经过我这么一说，是不是觉得貌似是很像哈。但是我又为什么说是有时候是可以的呢，而不是一定是可以的呢，因为如果你同一个任务（消息）提交 MQ 两次， MQ 队列中有两条同样的任务消息，那么当你这个任务不能有两个节点同时执行的时候（同时执行一个任务可能出现各种问题），MQ 就不能满足了，因为他不知道你这两条消息是同一个任务，它会把这两条消息可能会发给两个不同的节点同时执行（或者同一个节点的不同线程去执行），这时候你就需要自己去做一些事情去保证同一个任务不能同时被两个线程（或节点）去执行问题，这时候头又大了，那么 LTS 又派上用场了，以为 LTS 就可以保证这一点。说到任务调度，很多人一下就想到了 QuartZ ，对于这种实时任务的情况， QuartZ 是不太适合的，它也不能很好的解决故障转移（譬如执行中的节点突然停掉了， QuartZ 不能将这个执行中的任务立马分配给其他节点执行，最多设置了 QuartZ 的可恢复性，在停掉的节点重启之后重新执行该任务，但如果这个节点再也不启动起来了呢？那就只能呵呵了）等问题，这类场景 QuartZ 就不做比较了。有些人可能问，说了这么多，你倒是举个例子呀。嗯，我举几个例子： 1 ，发短信验证码，这种可以用 MQ 去实现，也可以单机去实现（如果你量不大的话），当然 LTS 也是可以的，如果你量非常非常大的话，建议还是用性能比较好的 MQ 代替 2 ，实时的给在线用户算数据，触发者是用户自己（自己手动点），但是算任务的只能同时由一个线程去执行，这是就可以用 LTS 了。

定时任务

> 某个时间点触发这种场景下，和实时任务相比，只有一个不同，就是要指定一个时间点去执行，可能是 1 个小时之后，可能是 1 天之后，也可能是 1 天，小时之后。有些人可能用轮训的业务数据库的方式去解决，轮训业务数据库有什么问题呢，当然你数据量很小我就不说了。如果你数据量还比较大的情况下，轮训数据库，势必会影响业务查询，如果有其他业务查询的话。还有就是对于分布式的支持不是很好，还有当表中存在多种不同执行（延迟）时间的任务，这个轮训频率就比较关键了，太短，影响性能，太长，影响业务，执行不及时，导致任务执行延迟太久，等等。这时候如果用MQ ，虽然有些 MQ 支持延迟队列 (RabbitMQ , RocketMQ 等），但他们都是指定的一些特定的 Level 级别延迟，但是不支持任意时间精度，譬如， 1 min , 5 min . 10 min 等等，但如果是 7 分钟，或者 20 天之后呢。如果 MQ 支持任意时间精度，那么它的性能就只能呵呵了，这种情况 MQ 就排除了，但是如果 MQ 的这些特定的 Level 刚好满足你的需求，那么选 MQ 也是可以的。再说说 Quartz吧， Quartz 可以支持定时任务，支持某个时间点触发，也支持集群，它在架构上是分布式的，没有负责几种管理的节点。 Quartz 是通过数据库行级锁的方式实现多线程之间任务争用的问题。行锁有嘟些特点呢，开销大，加锁慢，会出现死锁，并发度相比表级锁，页级锁高一点。但是在任务量比较大的时候，并发度较大的时候，行级锁就显得比较吃力了，而且很容易发生死锁。那么 LTS 是怎么解决并发性的问题的呢， LTS 采用预加载和乐观锁的方式，批量的将部分要执行的任务预加载到内存中，所以在取任务的时候只需要两步： 1 ，从内存中取到一个任务，当然内存中保证同一个线程拿到同一个任务是很容易的也是很高效的， 2 ，将拿到的这个任务对应的数据库记录锁住，那么这里采用乐观锁， CAS 的方式去修改记录（如果任务已经被别的节点拿走了，那么重新执行 1 , 2 步，这种已经被别的节点拿走的情况，主要是在多个 JobTracker 的情形下，单个 JobTracker 不会出现这种情况，但是在多个 JobTracker 下，内存中的预加载数据采用不同步长的方式来减小两个 JobTracker 内存中数据重复的概率，很好的解决了这个问题，这里稍微提下》 ，所以这个时候LTS相对于QuartZ 的优势一下就体现出来了。还有就是上面说的 Quartz 对故障转移做的不是很好。还有就是当 QuartZ 对应的 MySQL 数据库挂了，这时候问题就来了客户端提交的任务提交不成功了，那么有人会想将这些数据保存在内存中，等 MySOL 重启起来了再重试提交，那么如果你当前节点也挂了呢，你内存中的数据就会全部丢失了，所以这时候你需要自己额外的去做一些失败任务本地持久化的工作，不过如果你用LTS的话， LTS 支持Failstore ，任务提交失败了，自动帮你本地持久化，然后待 JobTracker 可用的时候重试，不管你是 JobTracker 挂了，还是 JobTracker 对应的数据库挂了，都是 ok 的。举个例子吧，在一个小时之后给某些用户发短信，或者当用户点击退款操作之后，从点击退货的这个时间点开始， n 天后将这个退款关闭。

周期性任务(Cron,Repeat)

> 这种场景下，和定时任务相比，不一样的地方，就只有，这个是会重复执行的，相当于重复执行的定时任务。所以这种场景下的对比，可以继续考照定时任务的对比。 LTS在这种场景下提供的特性有，提供统一的后台监控和后台管理。当某次定时任务执行失败，会执行重试操作，并提供执行日志，

如何使用

第一步，从 `github` 或者 `oschina` 上 `git clone` 或者 `download zip` 下来。

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

环境准备

1. Java JDK

> 因为LTS是使用Java语言编写的，所以必须要有个Java编译运行环境，目前LTS支持JDK1.6及以上版本。

2. Maven

> LTS项目是基于Maven做项目依赖管理的，所以用户机器上需要配置Maven环境

3. Zookeeper/Redis

> 因LTS目前支持Zookeeper和Redis作为注册中心，主要用于节点信息暴露、监听、master节点选举。用于选择其一即可，建议zookeeper。

4. Mysql/Mongodb

> LTS目前支持Mysql和mongodb作为任务队列的存储引擎。用户同样的选择其中一个即可。

打包部署

> 目前已经上传到了maven中央仓库了，可以不用手动编译了（下面的可以忽略）
> http://search.maven.org/#search%7Cga%7C1%7Ccom.github.ltsopensource

1. 你的项目也是Maven构建

> 如果你的项目是 maven 构建的话,需要将上面这些工程的 jar 包上传到你们自己的 maven 服务器上。然后再引入相应的 jar 即可。上传的话,可以直接在工程的 `pom.xml` 文件中添加 distributionManagement 即可。譬如

```
<distributionManagement>
        <repository>
            <id>nexus-releases</id>
            <name>Your Nexus Release Repository</name>
 <url>http://maven.xxxxxx.com/nexus/content/repositories/releases/</url>
        </repository>
        <snapshotRepository>
            <id>nexus-snapshots</id>
            <name>Your Nexus Snapshot Repository</name>            <url>http://ma
ven.xxxxxx.com/nexus/content/repositories/snapshots/</url>
        </snapshotRepository>
    </distributionManagement>
```

2. Jar方式

> 如果你是直接使用 jar 包的方式,那么你先需要安装 maven,然后通过 maven 来编译整个工程,然后将编译出来的 jar 包引入即可。在 `light-task-scheduler` 根路径执行 `mvn clean install -U -DskipTests`

3. 构建脚本启动的bin包

提供 `(cmd)windows` 和 `(shell)linux` 两种版本脚本来进行编译和部署:

1. 运行根目录下的 `sh build.sh` 或 `build.cmd` 脚本，会在 `dist` 目录下生成 `lts-{version}-bin` 文件夹
2. 下面是其目录结构，其中bin目录主要是JobTracker和LTS-Admin的启动脚本。 `jobtracker` 中是 JobTracker的配置文件和需要使用到的jar包， `lts-admin` 是LTS-Admin相关的war包和配置文件。 lts-{version}-bin的文件结构

```
-- lts-${version}-bin
    |-- bin
    |   |-- jobtracker.cmd
    |   |-- jobtracker.sh
    |   |-- lts-admin.cmd
    |   |-- lts-admin.sh
    |   |-- lts-monitor.cmd
    |   |-- lts-monitor.sh
    |   |-- tasktracker.sh
    |-- conf
    |   |-- log4j.properties
    |   |-- lts-admin.cfg
    |   |-- lts-monitor.cfg
    |   |-- readme.txt
    |   |-- tasktracker.cfg
    |   |-- zoo
    |       |-- jobtracker.cfg
    |       |-- log4j.properties
    |       |-- lts-monitor.cfg
    |-- lib
    |   |-- *.jar
    |-- war
        |-- jetty
        |   |-- lib
        |       |-- *.jar
        |-- lts-admin.war
```

1. JobTracker启动。如果你想启动一个节点，直接修改下 `conf/zoo` 下的配置文件，然后运行 `sh jobtracker.sh zoo start` 即可，如果你想启动两个JobTracker节点，那么你需要拷贝一份zoo,譬如命名为 `zoo2` ,修改下 `zoo2` 下的配置文件，然后运行 `sh jobtracker.sh zoo2 start` 即可。logs文件夹下生成 `jobtracker-zoo.out` 日志。
2. LTS-Admin启动.修改 `conf/lts-monitor.cfg` 和 `conf/lts-admin.cfg` 下的配置，然后运行 `bin` 下的 `sh lts-admin.sh` 或 `lts-admin.cmd` 脚本即可。logs文件夹下会生成 `lts-admin.out` 日志，启动成功在日志中会打印出访问地址，用户可以通过这个访问地址访问了。

4. 部署建议

> Admin后台: 建议Admin后台单独部署，默认会嵌入一个Monitor
>
> Monitor：默认在Admin后台进程中有一个，如果一个不够，也可以单独启动多个
>
> JobTracker: 建议单独部署
>
> JobClient:,这个是提交任务的工程,一般是和业务相关的,所以会放在业 务工程中, 当然也要看业务场景
>
> TaskTracker,这个因为是跑任务的,具体看业务场景, 一般情况下也可以是独立部署

快速开始

1. 准备工作

1.1 下载例子，并导入进IDEA或Eclipse

> 例子地址 ： https://github.com/ltsopensource/lts-examples

1.2 下载Admin

> 地址：http://git.oschina.net/hugui/light-task-scheduler/blob/master/dist/lts-1.6.8-bin.zip

1.3 启动zookeeper

> 这里假设是 127.0.0.1 端口 2181

1.4 启动mysql

> 这里假设是 127.0.0.1:3306 用户名密码：root, root
>
> 创建数据库：假设数据库名称是 lts

2. 跑例子

2.1 启动JobTracker

> 启动 lts-examples/lts-example-jobtracker/lts-example-jobtracker-java 下 com.github.ltsopensource.example.java.Main 类, 出现 `com.github.ltsopensource.core.cluster.JobNode - [LTS] ========== Start success` 日志表示启动成功

2.2 启动TaskTracker

> 启动 lts-examples/lts-example-tasktracker/lts-example-tasktracker-java 下 com.github.ltsopensource.example.java.Main 类, 出现 `com.github.ltsopensource.core.cluster.JobNode - [LTS] ========== Start success` 日志表示启动成功

2.3 启动JobClient并提交任务

> 启动 lts-examples/lts-example-jobclient/lts-example-jobclient-java 下 com.github.ltsopensource.example.java.Main 类, 出现 `com.github.ltsopensource.core.cluster.JobNode - [LTS] ========== Start success` 日志表示启动成功, 这个例子里面在jobclient启动完成之后，提交了四个不同的任务。在提交之后，你回收到任务执行完成的日志，并查看tasktracker端，也会有执行的日志

到这里一个LTS的任务执行链路已经成功打通，但是还没有一个可视化的界面提供用户查看，操作等，下面再介绍启动Admin后台

3. 启动Admin

3.1 解压前面下载到的lts-1.6.8-beta1-bin.zip

> 文件结构大概如下

```
-- lts-${version}-bin
    |-- bin
    |   |-- jobtracker.cmd
    |   |-- jobtracker.sh
    |   |-- lts-admin.cmd
    |   |-- lts-admin.sh
    |   |-- lts-monitor.cmd
    |   |-- lts-monitor.sh
    |   |-- tasktracker.sh
    |-- conf
    |   |-- log4j.properties
    |   |-- lts-admin.cfg
    |   |-- lts-monitor.cfg
    |   |-- readme.txt
    |   |-- tasktracker.cfg
    |   |-- zoo
    |       |-- jobtracker.cfg
    |       |-- log4j.properties
    |       |-- lts-monitor.cfg
    |-- lib
    |   |-- *.jar
    |-- war
        |-- jetty
        |   |-- lib
        |       |-- *.jar
        |-- lts-admin.war
```

> 1. 修改 conf/lts-monitor.cfg 和 conf/lts-admin.cfg 下的配置，如果有需要的话 (如果需要改下端口，添加 port=端口号 到 lts-admin.cfg中)
> 2. 启动windows下运行 bin/lts-admin.cmd, linux 下运行bin/lts-admin.sh 访问 http://localhost:8081/index.htm

3.2 自己部署tomcat

> 拷贝 war/lts-admin.war 到tomcat的部署目录，解压之后，修改相应的配置文件即可（conf/lts-monitor.cfg 和 conf/lts-admin.cfg）

参数说明

基本参数

参数	是否必须	默认值	使用范围	
registryAddress	必须	无	JobClient,JobTracker,TaskTracker	setRegistryA
clusterName	必须	无	JobClient,JobTracker,TaskTracker	setCluste
listenPort	必须	35001	JobTracker	set
dataPath	可选	user.home	JobClient,TaskTracker,JobTracker	setDa
bindIp	可选	本机ip	JobClient,TaskTracker,JobTracker	se
nodeGroup	可选	无	JobClient,TaskTracker	setNod
workThreads	可选	64	TaskTracker	setWorkT
jobRunnerClass	可选	无	TaskTracker	setJobRunn

扩展参数

扩展参数全部是在configs下面, 有以下几种使用方式

1. 硬编码方式，jobClient.addConfig("key", "value");
2. spring配置方式：

```
<property name="configs">
 <props>
     <prop key="key">value</prop>
 </props>
</property>
```

3. springboot或LTS自带的properties模式

```
## 譬如
lts.tasktracker.configs.job.fail.store=mapdb
```

参数列表

参数	默认值	
lts.job.processor.thread	32 + AVAILABLE_PROCESSOR * 5	J
java.compiler	javassist	J
lts.remoting.serializable.default	fastjson	J
lts.remoting	netty	J
job.fail.store	leveldb	Job rock 现, le

jdbc.datasource.provider	mysql	
zk.client	zkclient	J
job.logger	mysql	
lts.logger	slf4j	J
lts.json	fastjson	J
job.queue	mongo	
jdbc.url	无	
jdbc.username	无	
jdbc.password	无	
mongo.addresses	无	
mongo.database	无	
mongo.username	无	
mongo.password	无	
lts.http.cmd.port	无	JobT

tasktracker.stop.working.enable	false	
lts.monitor.report.interval	1	J
job.preloader.size	300	
job.preloader.factor	0.2	
job.submit.maxQPS	500	
job.submit.lock.acquire.timeout	100	
jobtracker.job.retry.interval.millis	30	
job.max.retry.times	10	
remoting.req.limit.enable	false	
remoting.req.limit.maxQPS	5000	
remoting.req.limit.acquire.timeout	50	
jobtracker.executing.job.fix.check.interval.seconds	30	
jobtracker.executing.job.fix.deadline.seconds	20	
job.pull.frequency	1	
lb.machine.res.check.enable	false	

lb.memoryUsedRate.max	0.9	
lb.cpuUsedRate.max	0.9	
monitor.select.loadbalance	random	J
jobtracker.select.loadbalance	random	
jobclient.select.loadbalance	random	

AutoConfigure

> 这里会优先讲一种AutoConfigure配置，主要为解决配置上的简单和多样化，参考SpringBoot的属性配置实现。

这个功能最主要的目的就是减少硬编码的编写，通过配置文件的方式来简化。最主要的功能就是将property映射成java对象。

譬如通过配置，然后生成先这个对象

```
@ConfigurationProperties(prefix = "lts.tasktracker")
public class TaskTrackerProperties {
    /**
     * 节点标识(可选)
     */
    private String identity;
    /**
     * 集群名称
     */
    private String clusterName;
    /**
     * zookeeper地址
     */
    private String registryAddress;
    /**
     * 执行绑定的本地ip
     */
    private String bindIp;
    /**
     * 额外参数配置
     */
    private Map<String, String> configs = new HashMap<String, String>();
    /**
     * 节点Group
     */
    private String nodeGroup;
    /**
     * FailStore数据存储路径
     */
    private String dataPath;
    /**
     * 工作线程,默认64
     */
    private int workThreads;

    private Level bizLoggerLevel;

    private DispatchRunner dispatchRunner;
```

```
    private Class<?> jobRunnerClass;

    //-------- Getter/Setter方法省略

    public static class DispatchRunner {
        /**
         * 是否使用shardRunner
         */
        private boolean enable = false;
        /**
         * shard的字段,默认taskId
         */
        private String shardValue;

        public boolean isEnable() {
            return enable;
        }

        public void setEnable(boolean enable) {
            this.enable = enable;
        }

        public String getShardValue() {
            return shardValue;
        }

        public void setShardValue(String shardValue) {
            this.shardValue = shardValue;
        }
    }
}
```

prefix="lts.tasktracker" 表示配置文件中的lts.tasktracker开头的所有配置才会映射 那么配置文件即可这样写：

```
lts.tasktracker.cluster-name=test_cluster
lts.tasktracker.registry-address=zookeeper://127.0.0.1:2181
lts.tasktracker.work-threads=64
lts.tasktracker.node-group=test_trade_TaskTracker
lts.tasktracker.dispatch-runner.enable=true
lts.tasktracker.dispatch-runner.shard-value=taskId
lts.tasktracker.configs.job.fail.store=mapdb
```

也可以这样写 (大写)

```
LTS.TASKTRACKER.CLUSTER-NAME=test_cluster
LTS.TASKTRACKER.REGISTRY-ADDRESS=zookeeper://127.0.0.1:2181
LTS.TASKTRACKER.WORK-THREADS=64
LTS.TASKTRACKER.NODE-GROUP=test_trade_TaskTracker
LTS.TASKTRACKER.DISPATCH-RUNNER.ENABLE=true
LTS.TASKTRACKER.DISPATCH-RUNNER.SHARD-VALUE=taskId
LTS.TASKTRACKER.CONFIGS.job.fail.store=mapdb
```

也可以这样写(小写+下划线分隔符)

```
lts_tasktracker_cluster-name=test_cluster
lts_tasktracker_registry-address=zookeeper://127.0.0.1:2181
lts_tasktracker_work-threads=64
lts_tasktracker_node-group=test_trade_TaskTracker
lts_tasktracker_dispatch-runner_enable=true
lts_tasktracker_dispatch-runner_shard-value=taskId
lts_tasktracker_configs_job.fail.store=mapdb
```

也可以这样写（大写+下划线分隔符）

```
LTS_TASKTRACKER_CLUSTER-NAME=test_cluster
LTS_TASKTRACKER_REGISTRY-ADDRESS=zookeeper://127.0.0.1:2181
LTS_TASKTRACKER_WORK-THREADS=64
LTS_TASKTRACKER_NODE-GROUP=test_trade_TaskTracker
LTS_TASKTRACKER_DISPATCH-RUNNER_ENABLE=true
LTS_TASKTRACKER_DISPATCH-RUNNER_SHARD-VALUE=taskId
LTS_TASKTRACKER_CONFIGS_job.fail.store=mapdb
```

... 等等，其实还有好多种，这里就不列举了

编码方式

> 本小节主要讲下使用硬编码方式启动各个节点

Spring注解方式

包引入说明

1. JobTracker,JobClient,TaskTracker都需要引入的包

1.1 lts-core

```
<dependency>
    <groupId>com.github.ltsopensource</groupId>
    <artifactId>lts-core</artifactId>
    <version>${lts版本号}</version>
</dependency>
```

1.2 zk客户端包

二选一, 通过 addConfig("zk.client", "可选值: curator, zkclient, lts") 设置, 如果用lts,可以不用引入包

`zkclient`

```
<dependency>
    <groupId>com.github.sgroschupf</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.1</version>
</dependency>
```

`curator`

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>2.9.1</version>
</dependency>
```

`zookeeper包`

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>${zk.version}</version>
    <exclusions>
        <exclusion>
            <groupId>org.jboss.netty</groupId>
            <artifactId>netty</artifactId>
        </exclusion>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

1.3 通讯包

netty或者mina, 二选一, 通过 addConfig("lts.remoting", "可选值: netty, mina") 设置

`netty`

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.0.20.Final</version>
</dependency>
```

`mina`

```
<dependency>
    <groupId>org.apache.mina</groupId>
    <artifactId>mina-core</artifactId>
    <version>2.0.9</version>
</dependency>
```

1.4 json包

fastjson或者jackson, 二选一, 通过 addConfig("lts.json", "可选值: fastjson, jackson") 设置

`fastjson`

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.7</version>
</dependency>
```

`jackson`

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.6.3</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.3</version>
</dependency>
```

1.5 日志包

可以选用 slf4j, jcl, log4j, 或者使用jdk原生logger

LoggerFactory.setLoggerAdapter("可选值: slf4j, jcl, log4j, jdk"), 不手动设置, 默认按这个顺序加载

`log4j`

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.16</version>
</dependency>
```

`slf4j`

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
</dependency>
```

`jcl`

```
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging-api</artifactId>
    <version>1.1</version>
</dependency>
```

1.6 如果需要spring的话,需要引入lts-spring及spring的相关包

```
<dependency>
    <groupId>com.github.ltsopensource</groupId>
    <artifactId>lts-spring</artifactId>
    <version>${lts版本号}</version>
</dependency>
```

2. 对于JobTracker端

2.1 必须引入的包:

```
<dependency>
    <groupId>com.github.ltsopensource</groupId>
    <artifactId>lts-jobtracker</artifactId>
    <version>${lts版本号}</version>
</dependency>
```

2.2 除了基础包之外还需要引入任务队列的包(可以是mongo或者mysql)

`mysql`

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.26</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.14</version>
</dependency>
```

`mongo`

```
<dependency>
    <groupId>org.mongodb.morphia</groupId>
    <artifactId>morphia</artifactId>
    <version>1.0.0-rc1</version>
</dependency>
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.0.2</version>
</dependency>
```

3. JobClient需要引入的包

必须引入的包

```
<dependency>
    <groupId>com.github.ltsopensource</groupId>
    <artifactId>lts-jobclient</artifactId>
    <version>${project.version}</version>
</dependency>
```

FailStore存储包(四选一)

通过 jobClient.addConfig("job.fail.store", "可选值: leveldb, mapdb, berkeleydb, rocksdb") 设置

```
<!--mapdb -->
<dependency>
    <groupId>org.mapdb</groupId>
    <artifactId>mapdb</artifactId>
    <version>2.0-beta10</version>
</dependency>
<!-- leveldb -->
<dependency>
    <groupId>org.fusesource.leveldbjni</groupId>
    <artifactId>leveldbjni-all</artifactId>
    <version>1.2.7<version>
</dependency>
<!-- berkeleydb  -->
<dependency>
    <groupId>com.sleepycat</groupId>
    <artifactId>je</artifactId>
    <version>5.0.73</version>
</dependency>
<!-- rocksdb -->
<dependency>
    <groupId>org.rocksdb</groupId>
    <artifactId>rocksdbjni</artifactId>
    <version>3.10.1</version>
</dependency>
```

3. TaskTracker需要引入的包

必须引入的包

```
<dependency>
    <groupId>com.github.ltsopensource</groupId>
    <artifactId>lts-tasktracker</artifactId>
    <version>${project.version}</version>
</dependency>
```

FailStore存储包(四选一)

通过 taskTracker.addConfig("job.fail.store", "可选值: leveldb, mapdb, berkeleydb, rocksdb") 设置

```
<!--mapdb -->
<dependency>
    <groupId>org.mapdb</groupId>
    <artifactId>mapdb</artifactId>
    <version>2.0-beta10</version>
</dependency>
<!-- leveldb -->
<dependency>
    <groupId>org.fusesource.leveldbjni</groupId>
    <artifactId>leveldbjni-all</artifactId>
    <version>1.2.7<version>
</dependency>
<!-- berkeleydb  -->
<dependency>
    <groupId>com.sleepycat</groupId>
    <artifactId>je</artifactId>
    <version>5.0.73</version>
</dependency>
<!-- rocksdb -->
<dependency>
    <groupId>org.rocksdb</groupId>
    <artifactId>rocksdbjni</artifactId>
    <version>3.10.1</version>
</dependency>
```

SpringBoot支持

HttpCmd

PreLoader

不依赖上一周期任务

任务分发

1. 任务由JobTracker分发给TaskTracker有两种途径
    i. TaskTracker会定时发送pull请求给JobTracker, 默认1s一次, 在发送pull请求之前，会检查当前TaskTracker是否有可用的空闲线程，如果没有则不会发送pull请求，同时也会检查本节点机器资源是否足够，主要是检查cpu和内存使用率，默认超过90%就不会发送pull请求，当JobTracker收到TaskTracker节点的pull请求之后，再从任务队列中取出相应的已经到了执行时间点的任务 push给TaskTracker，这里push的个数等于TaskTracker的空余线程数。
    ii. 还有一种途径是，每个TaskTracker线程处理完当前任务之后，在反馈给JobTracker的时候，同时也会询问JobTracker是否有新的任务需要执行，如果有JobTracker会同时返回给TaskTracker一个新的任务执行。所以在任务量足够大的情况下，每个TaskTracker基本上是满负荷的执行的。

常见问题

1. LTS支持JDK版本是多少

> LTS 支持JDK1.6及以上版本，如果是自己下载源码编译，需要注意编译jdk版本和运行时的jdk版本，运行时jdk版本不能小于编译时候的jdk版本。很常见的一个问题是，编译在1.8版本，然后运行时候小于1.8版本，就会出现以下异常：
> java.lang.NoSuchMethodError:
> java.util.concurrent.ConcurrentHashMap.keySet()Ljava/util/concurrent/ConcurrentHashMap$KeySetView

2. LTS需要引用哪些jar

> 具体引用可以参考lts-example例子中的引用方式
>
> LTS对于第三方jar的依赖基本上都是provided的scope，而且大部分都提供了多种实现供用户选择，用户可以通过参数来动态选择哪个实现。一方面就是为了防止lts引用的jar和用户自带引入的jar冲突，而可以使用其他实现方式来解决。当然lts也提供了丰富的扩展机制，用户可以自定义SPI扩展实现相应的扩展。

3. Leveldb问题

> 报错： java.lang.UnsatisfiedLinkError: Could not load library. Reasons: [no leveldbjni64-1.8 in java.library.path, no leveldbjni-1.8 in java.library.path, no leveldbjni in java.library.path 原因是操作系统缺少某些类库
>
> 解决办法：请更换job.fail.store为mapdb或者其他