1 任务调度的基本用法

1.1Quartz Scheduler 开源框架

Quartz 是开源任务调度框架中的翘首，是 java 业务界事实上的任务调度标准。

Quartz 提供了强大任务调度机制，而且使用简单。Quartz 允许开发人员灵活地定义触发器的调度时间表，并可以对触发器和任务进行关联映射。

此外，Quartz 提供了调度运行环境的持久化机制，可以保存并恢复调度现场，即使系统因故障关闭，任务调度现场数据并不会丢失。此外，Quartz 还提供了组件式的侦听器、各种插件、线程池等功能。

1.2 Quartz 的核心元素

Quartz 主要涉及以下元素概念：

scheduler：任务调度器

trigger：触发器，用于定义任务调度时间规则

job：任务，即被调度的任务

misfire：错过的，指本来应该被执行但实际没有被执行的任务调度

其中 trigger 和 job 是任务调度的元数据， scheduler 是实际执行调度的控制器。

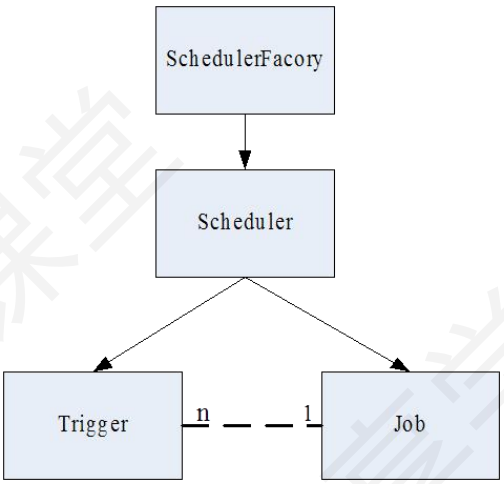
trigger 是用于定义调度时间的元素，即按照什么时间规则去执行任务。Quartz 中主要提供了四种类型的 trigger：SimpleTrigger，CronTirgger，DateIntervalTrigger，和 NthIncludedDayTrigger。这四种 trigger 可以满足企业应用中的绝大部分需求。我们将在企业应用一节中进一步讨论四种 trigger 的功能。

job 用于表示被调度的任务。主要有两种类型的 job：无状态的（stateless）和有状态的（stateful）。对于同一个 trigger 来说，有状态的 job 不能被并行执行，只有上一次触发的任务被执行完之后，才能触发下一次执行。Job 主要有两种属性：volatility 和 durability，其中 volatility 表示任务是否被持久化到数据库存储，而 durability 表示在没有 trigger 关联的时候任务是否被保留。两者都是在值为 true 的时候任务被持久化或保留。

一个 job 可以被多个 trigger 关联，但是一个 trigger 只能关联一个 job。

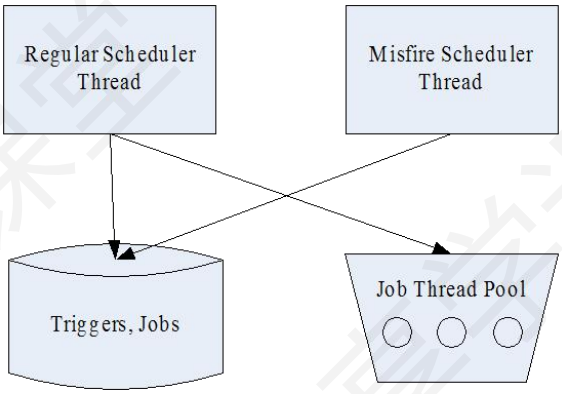
scheduler 由 scheduler 工厂创建：DirectSchedulerFactory 或者 StdSchedulerFactory。 一般使用 StdSchedulerFactory 工厂较多。 Scheduler 主要有三种：RemoteMBeanScheduler， RemoteScheduler 和 StdScheduler。最常用的为 StdScheduler。

核心元素间关系如下图：



1.3Quartz 的线程

在 Quartz 中，有两类线程，Scheduler 调度线程和任务执行线程，其中任务执行线程通常使用一个线程池维护一组线程。

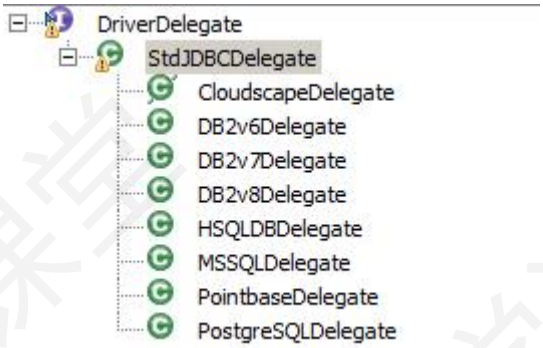


Scheduler 调度线程主要有两个： 执行常规调度的线程，和执行 misfired trigger 的线程。常规调度线程轮询存储的所有 trigger，如果有需要触发的 trigger，即到达了下一次触发的时间，则从任务执行线程池获取一个空闲线程，执行与该 trigger 关联的任务。Misfire 线程是扫描所有的 trigger，查看是否有 misfired trigger，如果有的话根据 misfire 的策略分别处理。

1.4数据存储

Quartz 中的 trigger 和 job 需要存储下来才能被使用。Quartz 中有两种存储方式：RAMJobStore, JobStoreSupport，其中 RAMJobStore 是将 trigger 和 job 存储在内存中，而 JobStoreSupport 是基于 jdbc 将 trigger 和 job 存储到数据库中。RAMJobStore 的存取速度非常快，但是由于其在系统被停止后所有的数据都会丢失，所以在通常应用中，都是使用 JobStoreSupport。

在 Quartz 中，JobStoreSupport 使用一个驱动代理来操作 trigger 和 job 的数据存储：StdJDBCDelegate。StdJDBCDelegate 实现了大部分基于标准 JDBC 的功能接口，但是对于各种数据库来说，需要根据其具体实现的特点做某些特殊处理，因此各种数据库需要扩展 StdJDBCDelegate 以实现这些特殊处理。Quartz 已经自带了一些数据库的扩展实现，可以直接使用，如下图所示：



1.5 Quartz 的用法

1.5.1 配置目标任务

使用 Spring Quartz 实现 Job 任务有两种方式，一种是实现 org.quartz.Job 接口，这个不推荐。另一种不需要继承，只需要在配置文件中定义 org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean，并指定它的 targetObject 属性为 Job 任务类，targetMethod 属性为任务方法就可以了。

```
/**
 * 普通业务类
 * @param serviceBean
 * @return
 */
@Bean(name = "serviceBeanDetail")
public MethodInvokingJobDetailFactoryBean serviceBeanDetail(XXXService serviceBean) {
    MethodInvokingJobDetailFactoryBean jobDetail = new MethodInvokingJobDetailFactoryBean();
    // 是否并发执行
    jobDetail.setConcurrent(false);
    // 需要执行的实体bean
    jobDetail.setTargetObject(serviceBean);
    // 需要执行的方法
    jobDetail.setTargetMethod("business");
    return jobDetail;
}
```

业务bean

要执行的业务方法

targetObject 属性指定目标对象。

1.5.2 配置触发器

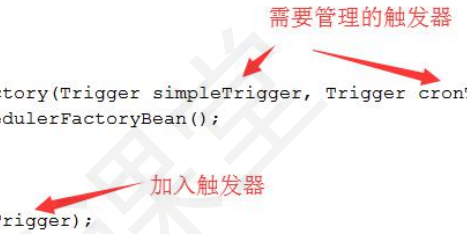
```
//cron触发器
@Bean(name = "cronTrigger")
public CronTriggerFactoryBean cronTrigger(JobDetail serviceBeanDetail) {
    CronTriggerFactoryBean triggerFactoryBean = new CronTriggerFactoryBean();
    triggerFactoryBean.setJobDetail(serviceBeanDetail);
    triggerFactoryBean.setCronExpression("0/6 * * * * ?");
    return triggerFactoryBean;
}
```

目标任务

触发规则

1.5.3 配置调度工厂

```java
/**
 * 调度工厂，将所有的触发器引入
 * @return
 */
@Bean(name = "scheduler")
public SchedulerFactoryBean schedulerFactory(Trigger simpleTrigger, Trigger cronTrigger) {
    SchedulerFactoryBean bean = new SchedulerFactoryBean();
    // 延时启动，应用启动1秒后
    bean.setStartupDelay(1);
    // 注册触发器
    bean.setTriggers(simpleTrigger,cronTrigger);
    return bean;
}
```



到此，整个配置就完成了。

1.6spring task 调度器用法

Spring 从 3.0 开始增加了自己的任务调度器，它是通过扩展 java.util.concurrent 包下面的类来实现的，它也使用 Cron 表达式。

使用 spring task 非常简单，只需要给定时任务类添加@Component 注解，给任务方法添加@Scheduled(cron = "0/5 * * * * ?")注解，并让 Spring 扫描到该类即可。

如果定时任务很多，可以配置 executor 线程池，这里 executor 的含义和 java.util.concurrent.Executor 是一样的，pool-size 的大小官方推荐为 5~10。scheduler 的 pool-size 是 ScheduledExecutorService 线程池。

```java
@Configuration
@EnableScheduling
public class ScheduleConfig implements SchedulingConfigurer {

    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.setScheduler(taskExecutor());
    }

    //配置线程池
    @Bean(destroyMethod="shutdown")
    public Executor taskExecutor() {
        return Executors.newScheduledThreadPool(10);
    }
}
```

2 分布式任务调度

在系统需要运行大量耗时定时任务的场景下，单使用类似 Quartz 或者 spring task 等定时任务框架无法满足对并发处理性能、监控管理及运维拓展的要求。

此时，定时任务主要面临以下几个新问题：

- 多节点重复执行某一任务
- 大量的任务管理困难
- 某些大型任务耗时超长，需要切分给多台机器并行执行

简单地说，如果仅仅面向解决第一个问题，我们可以借助分布式锁，来规避节点的执行难题（有兴趣的可参考分布式锁的使用）。下面我们主要介绍 elastic-job 和 xxl-job 这两个框架

共同点： E-Job 和 X-job 都有广泛的用户基础和完整的技术文档，都能满足定时任务的基本功能需求。

不同点：

X-Job 侧重的业务实现的简单和管理的方便，学习成本简单，失败策略和路由策略丰富。推荐使用在“用户基数相对少，服务器数量在一定范围内，任务数量多”的情景下使用

E-Job 关注的是数据，增加了弹性扩容和数据分片的思路，以便于更大限度的利用分布式服务器的资源。但是学习成本相对高些，推荐在“数据量庞大，且部署服务器数量较多”时使用

2.1E-Job 分布式调度

Elastic-Job 是一个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。

基于 quartz 定时任务框架为基础的，因此具备 quartz 的大部分功能

使用 zookeeper 做协调，调度中心，更加轻量级

支持任务的分片

支持弹性扩容,可以水平扩展，当任务再次运行时，会检查当前的服务器数量，重新分片，分片结束之后才会继续执行任务

失效转移，容错处理，当一台调度服务器宕机或者跟 zookeeper 断开连接之后，会立即停止作业，然后再去寻找其他空闲的调度服务器，来运行剩余的任务

提供运维界面，可以管理作业和注册中心。

我们主要介绍 Elastic-Job-Lite 的去中心化解决方案

2.1.1 使用场景

当今互联网项目大多为微服务，单模块可能在两个实例以上的数量，定时器就会出现多实例同时执行的情况。同时，我们的某些任务可能还存在过大，需要切片来加速执行等需求，归结如下：

- 分布式调度协调
- 弹性扩容缩容
- 失效转移
- 错过执行作业重触发
- 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
- 自诊断并修复分布式不稳定造成的问题
- 支持并行调度
- 支持作业生命周期操作
- 丰富的作业类型
- Spring 整合以及命名空间提供
- 运维平台

Elastic-Job-Lite 以其简单易用，无中心化方式，非常适合用来简单整合普通业务。

2.1.2 添加依赖

```
<dependency>
    <groupId>com.github.yinjihuan</groupId>
    <artifactId>elastic-job-spring-boot-starter</artifactId>
    <version>1.0.2</version>
</dependency>
```

2.1.3 配置

ps: 需要 zookeeper 支持，请提前搭建好。

配置 application.properties， 加入以下配置：

```
# zk 注册中心
elastic.job.zk.serverLists=127.0.0.1:2181
elastic.job.zk.namespace=enjoy_elastic7
spring.main.allow-bean-definition-overriding=true
```

配置很简单，只需要加入 zk 注册中心地址，和 job 名称空间即可 2.3 定时器实现方法编写

2.1.4 定时任务编写

Elastic-job 总共提供了三种类型的定时任务：Simple 类型定时任务、Dataflow 类型定时任务和 Script 类型定时任务。其中，Script 类型作业意为脚本类型作业，支持 shell，python 和 perl 等所有类型脚本，应用得不太多。

SimpleJob 需要实现 SimpleJob 接口，是未经过任何封装的定时任务简单实现，与 quartz 原生接口相似。

Dataflow 类型的定时任务主要用于处理数据流，需实现 DataflowJob 接口。该接口可以提供 2 个方法可供覆盖，分别用于抓取(fetchData)和处理(processData)数据。

推荐使用 ElasticJobConf 注解方式，如：

```
@ElasticJobConf(name = "EnjoySimpleJob",cron = "0/5 * * * * ?"
      ,shardingItemParameters = "0=beijing,1=shanghai",shardingTotalCount = 2
      ,listener = "com.enjoy.handle.MessageElasticJobListener"
      ,jobExceptionHandler = "com.enjoy.handle.CustomJobExceptionHandler"
)
```

ElasticJobConf 的常用的参数释义如下：

cron	cron 表达式，用于配置作业触发时间
sharding-total-count	作业分片总数
sharding-item-parameters	分片序列号和参数用等号分隔，多个键值对用逗号分隔 分片序列号从 0 开始，不可大于或等于作业分片总数 如：0=a,1=b,2=c
job-parameter	作业自定义参数，可以配置多个相同的作业，但是用不同的参数作为不同的调度实例

misfire	是否开启错过任务重新执行
listener	任务开始和结束时，自定义的处理功能
jobExceptionHandler	任务异常时，自定义的处理

2.1.5 任务启动

对于 springboot 来说，使用 EnableElasticJob 标注启动 elasticjob 即可

2.2 X-Job 分布式调度

XXL-JOB 是一个轻量级分布式任务调度平台，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。现已开放源代码并接入多家公司线上产品线，开箱即用。与 E-Job 走无中心化方式不同，XXL-JOB 是中心集权方式。

2.2.1 使用场景

xxl-job 是一款基于 spring， quartz，netty 开源定时任务框架，解决的问题是不用每一个 job-client 都需要集成 quartz，管理 cron 配置，尤其 job 很多的时候当容器启动会执行一堆 job，影响启动速度。

原理简单而言就是由 job 配置中心管理通过 quartz 控制客户端 job 触发时机，然后通过 netty rpc 调用执行客户端的具体实现。这样中心化的方式可以极大改善 job 的管理成本，还可以配置集群。下面是其全景图：

xxl-job 的官方文档非常详细，可以到这里查看：https://www.xuxueli.com/xxl-job/#/

本文，我们主要讲解一个其基本的搭建与使用流程

2.2.2 部署调度中心

作为中心化的调度平台，xxl 的调度中心当然是核心中的核心。

其作用：统一管理任务调度平台上调度任务，负责触发调度执行，并且提供任务管理平台。

你可以 github 下载：https://github.com/xuxueli/xxl-job

下载好代码包后，第一步执行 doc 中的 db 脚本，创建好数据库表。

然后更改调度中心配置文件地址：/xxl-job/xxl-job-admin/src/main/resources/xxl-job-admin.properties

```
### 调度中心JDBC链接
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/xxl-job?Unicode=true&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root_pwd
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

### 报警邮箱
spring.mail.host=smtp.qq.com
spring.mail.port=25
spring.mail.username=xxx@qq.com
spring.mail.password=xxx
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true

### 登录账号
xxl.job.login.username=admin
xxl.job.login.password=123456

### 调度中心通讯TOKEN，用于调度中心和执行器之间的通讯进行数据加密，非空时启用
xxl.job.accessToken=

### 调度中心国际化设置，默认为中文版本，值设置为“en”时切换为英文版本
xxl.job.i18n=
```


启动：直接启动 XxlJobAdminApplication 即可

2.2.3 调度中心集群

在生产环境中，需要提升调度系统容灾和可用性，调度中心支持集群部署，只需要多台部署连接同一套数据库即可。

- DB 配置保持一致；
- 登陆账号配置保持一致；
- 集群机器时钟保持一致（单机集群忽视）；

建议：推荐通过 nginx 为调度中心集群做负载均衡，分配域名。

调度中心访问、执行器回调配置、调用 API 服务等操作均通过该域名进行。

2.2.4 部署执行器项目

执行器就是我们的业务系统，负责接收“调度中心”的调度并执行业务任务；下面简介集成步骤：

2.2.4.1步骤一：引入 xxl-job-core 的依赖

```
<!-- job-job-core -->
<dependency>
    <groupId>com.xuxueli</groupId>
    <artifactId>xxl-job-core</artifactId>
```

```
    <version>2.0.2</version>
</dependency>
```

2.2.4.2步骤二：执行器配置

在 application.properties 下的配置信息信息如下：

```
### 调度器的地址----- 发消息
xxl.job.admin.addresses=http://localhost:8080/xxl-job-admin
### 当前执行器的标识名称，同一个名字的执行器构成集群
xxl.job.executor.appname=xxl-enjoy
# 执行器与调度器通信的 ip / port
xxl.job.executor.ip=
xxl.job.executor.port=9991
### job-job, access token
xxl.job.accessToken=
### job-job log path
xxl.job.executor.logpath=/logs/xxl/job
### job-job log retention days
xxl.job.executor.logretentiondays=-1
```

调度中心部署跟地址 [选填]：如调度中心集群部署存在多个地址则用逗号分隔。执行器将会使用该地址进行"执行器心跳注册"和"任务结果回调"；为空则关闭自动注册；

xxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin

执行器 AppName [选填]：执行器心跳注册分组依据；为空则关闭自动注册

xxl.job.executor.appname= xxl-enjoy

执行器 IP [选填]：默认为空表示自动获取 IP，多网卡时可手动设置指定 IP，该 IP 不会绑定 Host 仅作为通讯实用；地址信息用于 "执行器注册" 和 "调度中心请求并触发任务"；

xxl.job.executor.ip=

执行器端口号 [选填]：小于等于 0 则自动获取；默认端口为 9999，单机部署多个执行器时，注意要配置不同执行器端口；

xxl.job.executor.port=9999

执行器通讯 TOKEN [选填]：非空时启用；

xxl.job.accessToken=

执行器运行日志文件存储磁盘路径 [选填] ：需要对该路径拥有读写权限；为空则使用默认路径；

xxl.job.executor.logpath=/data/applogs/xxl-job/jobhandler

执行器日志保存天数 [选填] ：值大于 3 时生效，启用执行器 Log 文件定期清理功能，否则不生效；

xxl.job.executor.logretentiondays=-1

2.2.4.3步骤三：执行器组件配置

需要将上面的执行器各项参数，配置到`XxlJobSpringExecutor`组件中，创建如下：

```
@Bean(initMethod = "start", destroyMethod = "destroy")
public XxlJobSpringExecutor xxlJobExecutor() {
    logger.info(">>>>>>>>>>> job-job config init.");
    XxlJobSpringExecutor xxlJobSpringExecutor = new XxlJobSpringExecutor();
    xxlJobSpringExecutor.setAdminAddresses(adminAddresses);
    xxlJobSpringExecutor.setAppName(appName);
    xxlJobSpringExecutor.setIp(ip);
    xxlJobSpringExecutor.setPort(port);
    xxlJobSpringExecutor.setAccessToken(accessToken);
    xxlJobSpringExecutor.setLogPath(logPath);
    xxlJobSpringExecutor.setLogRetentionDays(logRetentionDays);

    return xxlJobSpringExecutor;
}
```

2.2.4.4步骤四：编写任务

任务类必须实现`IJobHandler`接口，它的`execute`方法即为执行器执行入口

```
@JobHandler(value="demoJobHandler")
@Service
public class EnjoySharding extends IJobHandler {
   @Autowired
   private EnjoyBusiness enjoyBusiness;

   @Override
   public ReturnT<String> execute(String param) throws Exception {
      ShardingUtil.ShardingVO shardingVO = ShardingUtil.getShardingVo();

      int index = shardingVO.getIndex();
      int total = shardingVO.getTotal();
      enjoyBusiness.process(index,total,param);
      return SUCCESS;
   }
}
```

`@JobHandler(value="enjoySharding")`所标的名字，`demoJobHandler`专为调度中指代本任务

2.2.4.5调度中心新建任务

登录调度中心，点击下图所示“新建任务”按钮，新建示例任务。然后，参考下面截图中任务的参数配置，点击保存。