Kylin任务调度模块

Kylin Job模块

Kylin运行过程中会生成很多不同的Job，这些job是如何调度执行的呢？本文从源码的层面分析kylin是如何完成多个任务的并发执行，以及如何在其他项目中使用这种多线程的模型。

初始化流程

首先，kylin使用了spring框架提供RESTful接口，JobController这个controller实现了InitializingBean接口，这就意味着在初始化bean的时候spring会调用这个类实现的afterPropertiesSet方法执行初始化操作。在初始化函数中会加载kylin的配置文件（文件名为kylin.properties），配置文件的加载首先查找JVM的环境变量KYLIN*CONF，如果不存在则进一步查找KYLIN*HOME，找到其下面的conf目录，然后加载配置文件，并且还会查找该目录下的kylin.properties.override文件是否存在，如果存在则将后者的配置覆盖kylin.properties文件的配置。

在kylin server中分为三种运行模式（可以通过kylin.server.mode配置项配置，默认为all），分别为all、job和query，前两种是可以执行任务的，而query模式下kylin server只提供元数据的操作以及SQL查询，不能执行构建cube、合并cube之类的任务。因此可以看到只有在前两种模式下，该函数会启动一个线程创建一个DefaultScheduler对象，该对象是全局唯一的，然后执行该对象的init方法。

init函数中会首先需要从zookeeper中获取一个lock，这个锁是互斥的，相同的zookeeper的路径只能由一个kylin server实例持有，这个锁是由hbase和zookeeper共同标识的，这意味着不同的kylin server必须使用不同的hbase的元数据表，zookeeper的地址由kylin.storage.url配置项标识，指定quorum和端口，然后再根据kylin.metadata.url配置项查找kylin server使用的元数据存储在hbase的表，默认的hbase表为kylin*metadata，如果这个配置项是hbase:开头的则使用这个配置，否则使用默认的表，加锁的路径为/kylin/job*engine/lock/hbase表名。成功获取锁之后会初始化一个线程池：

```
    //根据配置对象获取manager对象
    executableManager = ExecutableManager.getInstance(jobEngineConfig.getConfig());
    //创建一个大小为1的线程池，这个线程池中周期性的调度查看是否有可执行的任务。
    fetcherPool = Executors.newScheduledThreadPool(1);
   //真正调度任务执行的线程池的大小，默认为10个，使用的队列是无最大上限的。
    int corePoolSize = jobEngineConfig.getMaxConcurrentJobLimit();
    jobPool = new ThreadPoolExecutor(corePoolSize, corePoolSize, Long.MAX_VALUE, TimeUnit.DAYS, new SynchronousQueue<Runnable>());
    //所有正在执行的任务都会保存在context中。
    context = new DefaultContext(Maps.<String, Executable> newConcurrentMap(), jobEngineConfig.getConfig());
     //从元数据库中获取所有的READY状态的任务，置为ERROR，以供接下来重新调度执行。
    for (AbstractExecutable executable : executableManager.getAllExecutables()) {
        if (executable.getStatus() == ExecutableState.READY) {
            executableManager.updateJobOutput(executable.getId(), ExecutableState.ERROR, null, "scheduler initializing work to res
        }
    }
    //将所有RUNNING状态的Job设置为ERROR，以供重新调度。
    executableManager.updateAllRunningJobsToError();
    //进程退出的时候销毁两个线程池，释放zookeeper上的锁
    Runtime.getRuntime().addShutdownHook(new Thread() {
        public void run() {
            logger.debug("Closing zk connection");
            try {
                shutdown();
            } catch (SchedulerException e) {
                logger.error("error shutdown scheduler", e);
            }
        }
    });
    //FetcherRunner线程是周期性的查看其它任务是否可执行的线程，第一次调度的时延为10秒，接下来60秒调度一次。
    fetcher = new FetcherRunner();
    fetcherPool.scheduleAtFixedRate(fetcher, 10, ExecutableConstants.DEFAULT_SCHEDULER_INTERVAL_SECONDS, TimeUnit.SECONDS);
    hasStarted = true;
```

JobRunnable线程

在kylin中，每一个Job都的状态处于以下几种之一：READY,RUNNING,ERROR,STOPPED,DISCARDED,SUCCEED;只有处于READY状态的线程才可能被调度执行，每一个Job都会启动一个线程执行，线程的对象为JobRunner，将该线程放入到jobPool 线程池中调度执行。

```
private class JobRunner implements Runnable {

    private final AbstractExecutable executable;

    public JobRunner(AbstractExecutable executable) {
        this.executable = executable;
    }

    @Override
    public void run() {
        try {
            //执行job的处理函数
            executable.execute(context);
            //执行完成之后触发下一次任务的查询，而不是等到下一个60秒。
            fetcherPool.schedule(fetcher, 0, TimeUnit.SECONDS);
        } catch (ExecuteException e) {
            logger.error("ExecuteException job:" + executable.getId(), e);
        } catch (Exception e) {
            logger.error("unknown error execute job:" + executable.getId(), e);
        } finally {
            context.removeRunningJob(executable);
        }
    }
}
```

JobRunner的构造函数是kylin中可执行人Job，这些job都继承自AbstractExecutable虚拟类，这个类定义了虚拟函数protected abstract ExecuteResult doWork(ExecutableContext context) throws ExecuteException;而不是execute函数。该类实现了execute函数，该函数的实现如下：

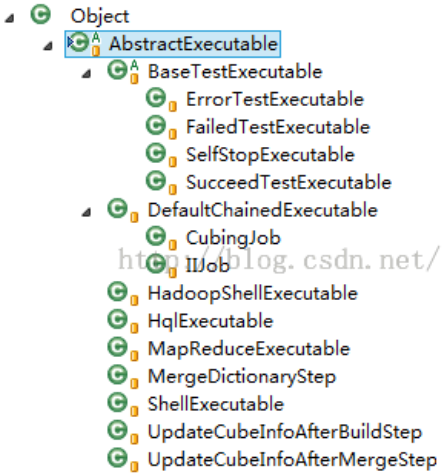
```
public final ExecuteResult execute(ExecutableContext executableContext) throws ExecuteException {

    //print a eye-catching title in log
    LogTitlePrinter.printTitle(this.getName());

    Preconditions.checkArgument(executableContext instanceof DefaultContext);
    ExecuteResult result;
    try {
        onExecuteStart(executableContext);
        result = doWork(executableContext);
    } catch (Throwable e) {
        logger.error("error running Executable", e);
        onExecuteError(e, executableContext);
        throw new ExecuteException(e);
    }
    onExecuteFinished(result, executableContext);
    return result;
}
```

可以看出在所有的任务的入口都是execute函数，该函数将调用所有任务的公共操作，例如执行任务前调用onExecuteStart函数，执行任务调用doWork函数，执行出现异常时调用onExecuteError函数，未抛出异常执行完成之后调用onExecuteFinished函数，这些函数都可以被具体的Job实现已完成具体的需求。

首先我们来分析一下这种任务调度模型，启动了两个线程池，线程池A只能持有一个线程，这个线程就是周期性的检查任务队列中是否有可执行的任务，如果存在可执行的任务则将它作为参数创建一个新的线程对象并交给线程池B调度执行，线程池B调度到这个线程的时候执行线程的run函数，该函数执行具体任务的逻辑，但是这些任务总体执行逻辑是相同的（先执行start，然后doWork最后执行finish），因此都继承自某一个虚拟类，在run函数中调用这个虚拟类的a函数，a函数内实现总体执行逻辑（start、doWork、finish等），然后由不同的子类实现不同的start、doWork、finish等具体函数已实现不同的逻辑。

Kylin Job介绍

AbstractExecutable类的继承关系如下图：



- 其中BaseTestExecutable我们不用关心，DefaultChainedExecutable是用于链式执行多个job的job，它的内部可以链接多个job，然后按照job的加入顺序依次执行，它的子类包括CubingJob和IIJob分别用于实现构建普通的cube和构建倒排索引（这个暂时没有用到）。
- HadoopShellExecutable用于执行类似于在hadoop的shell下提交的mapreduce任务，它的参数必须包含一个mapreduce的job class，然后执行这个mapreduce任务，并且会同步等待这个任务的执行完成，只检查该任务是否执行成功。
- MapReduceExecutable同样也是为了执行一个mapreduce任务，但是和HadoopShellExecutable不同的是它使用异步的方式执行提交的mapreduce任务，它不会一直等待任务的执行完成，而是提交完成之后立即返回，然后周期性的访问hadoop任务状态的url（根据resourceManager地址）查看任务的执行状态，如果执行完成则会调用getCounter获取任务的一些统计信息，检查状态的时延可以由kylin.job.yarn.app.rest.check.interval.seconds配置项设置，默认为60s，所以在kylin中可以看到对于普通的任务一般使用HadoopShellExecutable执行，而对于关键的任务（cube构建过程中的任务）会使用MapReduceExecutable执行mapreduce任务。
- HqlExecutable用于执行多条hive sql语句，语句通过配置项传递，目前看到的唯一执行hive sql的地方在于创建cube的第一步需要从hive中获取原始数据，但是目前没有使用HqlExecutable，而是选择直接调用shell命令hive -e执行。
- ShellExecutable用于执行一个shell命令，如刚才提到的执行hive -e执行原始数据的生成。目前shell可以支持命令在当前主机执行或者在远端主机上执行，可以通过kylin.job.run.as.remote.cmd配置决定是否在远端执行，kylin.job.remote.cli.hostname指定了远端主机的主机名，kylin.job.remote.cli.username和kylin.job.remote.cli.password分别指定了登陆用户名和密码，实现是通过SSH登陆到远端主机再执行相应的命令。
- UpdateCubeInfoAfterBuildStep和UpdateCubeInfoAfterMergeStep分别是用于在构建cube和合并cube完成之前的最后一步，这一步主要是更新一些统计信息，例如cube的大小，读写hdfs的大小等。

DefaultChainedExecutable执行流程

在kylin中主要使用的是MapReduceExecutable来执行构建cube的任务，而在这些类中DefaultChainedExecutable是比较特殊的，因为它本身并不会执行任务的逻辑，而是相当于多个具体job的容器，顺序的执行这些job，下面看一下这个类。在onExecuteStart函数中它会将自身的状态设置为RUNNING，而doWork函数如下：

```
@Override
protected ExecuteResult doWork(ExecutableContext context) throws ExecuteException {
    List<? extends Executable> executables = getTasks();
    final int size = executables.size();
    for (int i = 0; i < size; ++i) {
        Executable subTask = executables.get(i);
        if (subTask.isRunnable()) {
            return subTask.execute(context);
        }
    }
    return new ExecuteResult(ExecuteResult.State.SUCCEED, null);
}
```

可以看出它会顺序的从executables 数组获取job然后查看job是否可以执行（状态是否为READY），但是令人诧异的是每一个job执行完成之后它并不会调度执行下一个job，而是直接返回了，这也就意味着它只会执行executables链表中的第一个可执行的job，那接下来的怎么办呢？

这其中的奥秘在于每次成功执行一个job之后会调用这个job的onExecuteFinished函数，根据上面的逻辑，每执行完一个job都会跳出doWork函数执行onExecuteFinished函数，而在DefaultChainedExecutable的onExecuteFinished函数中，它会顺序的检查每一个任务的执行状况，如果最近一个任务执行失败，则标记整个job执行失败，如果成功则检查是否全部任务都执行成功，如果是则将整个任务标记为成功，然后检查是否有任何一个任务执行失败，如果是则将这个标记为失败（这一步理论上不需要再检查，因为每一个job完成之后都会检查），如果没有任何job失败并且也没有全部执行成功则再次将自身标记为READY就可以返回了。

虽然每次只执行了executables链表中的第一个可执行的job，但是每次执行完成之后都会将自身标记为READY，回想起之前JobRunner线程在每次job执行execute函数之后立即调度下一次查看是否有READY的job，这样在DefaultChainedExecutable对象中的前一个job执行完成之后就会立即调度下一个job执行（因为前一个任务的状态不再是READY），并且提供了检查每次任务执行完成都检查完成状态的逻辑，这样的结构还是挺巧妙的。

总结

上面大致介绍了kylin中不同类型的job以及比较复杂的DefaultChainedExecutable的具体调度流程，并且对kylin整个任务调度框架有了一定的了解，这种设计也是值得我们学习的地方，对于做后端的设计经常会遇到这种任务执行的需求，可以尝试用这种方式完成。