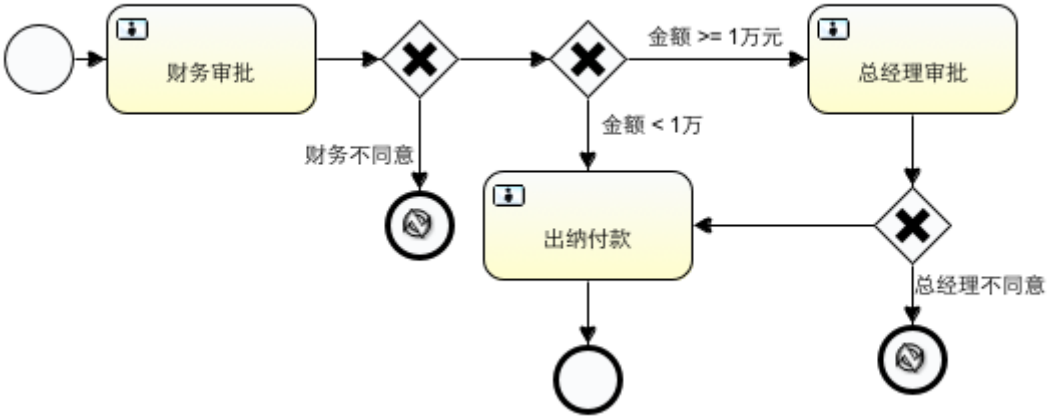


Activiti架构分析及源码详解

风火

二级城市努力折腾的coder

关注他

Activiti架构分析及源码详解

引言

工作流引擎，应用于解决流程审批和流程编排方面等问题，有效的提供了扩展性的支撑。而目前来说，工作流领域也有了相对通行化的标准规范，也就是BPMN2.0。支持这个规范的开源引擎主要有：Activiti，flowable，Jbpm4等。本文着重对Activiti的架构设计进行分析和梳理，同时对流程启动和原子操作的相关代码进行完整走读。

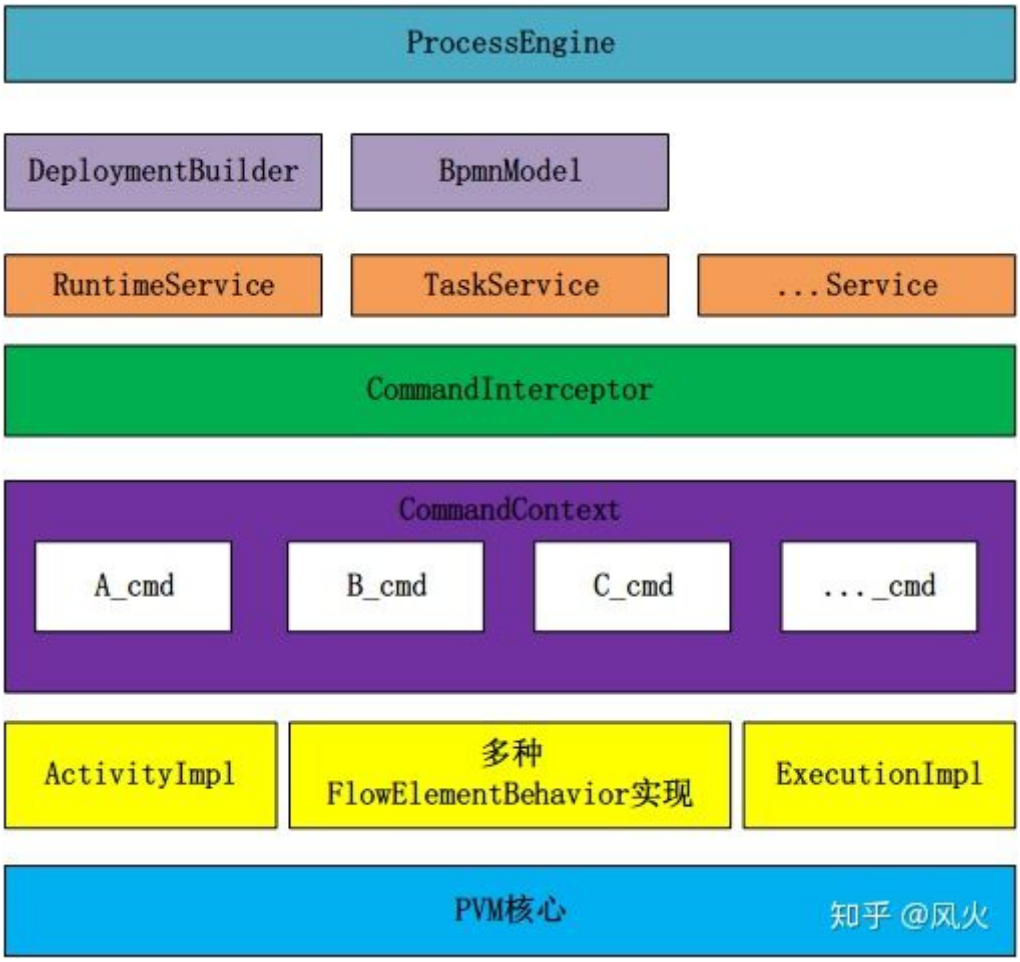
本文的阅读对象需要对Activiti有一定的理解并且已经能够初步的使用Activiti进行流程流转方面开发。

如果对文章内容有疑惑，欢迎加入技术交流群186233599，作者会不定时解答相关问题。

一、Activiti设计解析-架构&领域模型

1.1 架构

Activiti采用了一个分层架构完成自底向上的包装。架构图如下



大致包括：

- 核心接口层，被PVM接口定义。PVM会在后面的章节中详细讲述。
- 核心实现层，基于PVM的思想和接口，定义了一些关键实体包含ActivityImpl（该类抽象了节点实现），FlowElementBehavior实现（该类抽象了节点指令动作），ExecutionImpl（流程执行实体类）
- 命令层，Activiti在编码模式上直接限定整体风格为命令模式。也就是将业务逻辑封装为一个个的Command接口实现类。这样新增一个业务功能时只需要新增一个Command实现即可。这里需要特别提到的是，命令本身需要运行在命令上下文中，也就是CommandContext类对象。
- 命令拦截层，采用责任链模式，通过责任链模式的拦截器层，为命令的执行创造条件。诸如开启事务，创建CommandContext上下文，记录日志等
- 业务接口层，面向业务，提供了各种接口。这部分的接口就不再面向框架开发者了，而是面向框架的使用者。
- 部署层，严格来说，这个与上面说到的并不是一个完整的分层体系。但是为了突出重要性，单独拿出来说。流程运转的前提是流程定义。而流程定义解析就是一切的开始。从领域语言解析为Java的POJO对象依靠的就是部署层。后文还会细说这个环节。
- 流程引擎，所有接口的总入口。上面提到的业务接口层，部署层都可以从流程引擎类中得到。因此这里的流程引擎接口其实类似门面模式，只作为提供入口。

1.1.1 命令模式

Activit整体上采用命令模式进行代码功能解耦。将流程引擎的大部分涉及到客户端的需求让外部以具体命令实现类的方式实现。

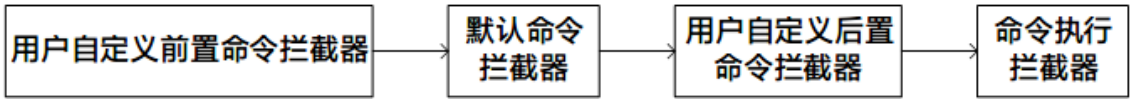
完成这个编码模式，有几个重点类需要关注

- `Command`命令接口，所有的具体命令都需要实现该类，最终业务就是执行该类的execute方法。
- `CommandContext`命令上下文，为具体命令的执行提供上下文支撑。该上下文的生成是依靠命令拦截器中的上下文拦截器`org.activiti.engine.impl.interceptor.CommandContextInterceptor`来生成的。该拦截器会判断是复用当前的上下文还是生成新的上下文。

引擎内的大部分功能都是通过单独的命令完成。

1.1.2 责任链模式

Activiti的命令模式还需要搭配其对应的责任链来完成。具体来说，Activiti中存在一个命令拦截器链条，该命令拦截器链条由几大块的拦截器实现组成，如下



其中重要的默认拦截器有2个：

- 事务拦截器，主要职责是使得后续命令运行在事务环境下。
- CommandContext拦截器，主要职责是在有必要的时候创建CommandContext对象，并在使用完成后关闭该上下文。

1.1.2.1 事务拦截器

事务拦截器是否提供取决于`org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl`的子类对方法`createTransactionInterceptor`的实现。独立使用时的`org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration`该方法返回为空。也就是不提供事务拦截器。此时，命令的运行就无法通过事务拦截器来提供事务环境了。

1.1.2.2 命令上下文拦截器

实现类：org.activiti.engine.impl.interceptor.CommandContextInterceptor。

该拦截器的功能非常重要，可以说是Activiti操作的核心之一。其作用是在后续拦截器执行前检查当前上下文环境，如果不存在CommandContext对象，则创建一个；在后续拦截器执行后，将CommandContext对象close。CommandContext包含了本次操作中涉及到所有的数据对象。

1.1.3 流程定义解析

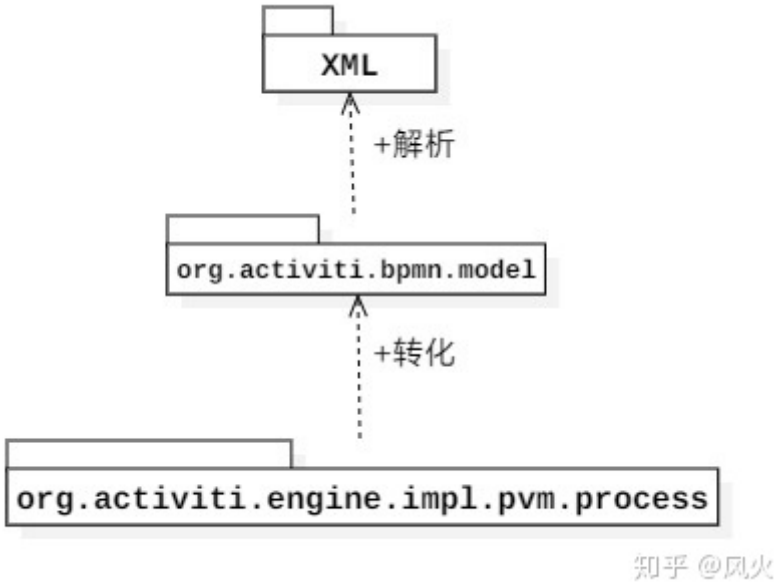
Activiti遵循BPMN2.0规范，因此框架中少不了对BPMN2.0规范的定义文件（XML形式）的解析类。Activiti采用的STAX的拉模型进行XML解析。这里先不分析其具体的解析类的内在联系，而是概念性的阐述下Activiti对解析的概念分层。

首先通过类`org.activiti.bpmn.converter.BpmnXMLConverter`进行XML解析，解析为`org.activiti.bpmn.model`

包下面的与各个XML元素定义对应的POJO类。此时这些POJO类仅仅只是XML文件的一个Java表达。

在通过类`org.activiti.engine.impl.bpmn.parser.BpmnParser`聚合不同的解析类，将上面步骤解析出来的POJO类进一步解析为可以在框架中利用的`org.activiti.engine.impl.pvm.process`包下面的类。典型的代表就是ActivityImpl类。

三者之间的关系简单用图表达就是



1.2 领域模型

Activiti采取了领域中的充血模型作为自己的实现方式。大部分业务逻辑都直接关联在了`org.activiti.engine.impl.persistence.entity.ExecutionEntity`中。由于Activiti采用了MyBatis而非Hibernate这样的O/R Mapping产品作为持久层，因此Activiti在具体的持久化操作上也有自己的独特方式。

1.2.1 数据集中提交

Activiti的持久化机制简单说来就是**数据集中提交**。集中提交还产生了一个额外的作用：自动提交。换句话说，在内存中的实体，如果更新了属性但是没有显示的执行刷新动作，在一个调用的生命周期结束后也会被持久化其最新的状态到数据库。下面来看下详细解释下这个集中提交机制。

在Activiti所有运行期生成的对象都需要实现一个接口`org.activiti.engine.impl.interceptor.Session`，其定义如下

```
public interface Session
{
  void flush();
  void close();
}
```

而Session对象则是由接口`org.activiti.engine.impl.interceptor.SessionFactory`的方法进行生成，其定义如下

```
public interface SessionFactory {
  Class<?> getSessionType();
  Session openSession();
}
```

流程引擎内部持有各种SessionFactory实现，用户也可以自定义注册自己的SessionFactory实现，如果用户希望自定义的对象也可以被集中提交机制处理的话。

在CommandContext中存在一个Map<Class,Session>存储，存储着该CommandContext生命周期内新建的所有Session对象。当一个命令执行完毕后，最终命令上下文CommandContext的close方法会被调用。当执行CommandContext.close()方法时，其内部会按照顺序执行`flushSessions`,`closeSessions`方法。从名字可以看到，第一个方法内部就是执行所有Session对象的flush方法，第二个方法内部就是执行所有的Session对象的close方法。

流程引擎内部有一个Session实现是比较特别的。也就是`org.activiti.engine.impl.db.DbSqlSession`实现。如果有需要更新，删除，插入等操作，该操作是需要通过DbSqlSession来实现的，而实际上该实现会将这些操作缓存在内部。只有在执行flush方法时才会真正的提交到数据库去执行。正是因为如此，所有的数据操作，实际上最终都是要等到CommandContext执行close方法时，才会真正提到到数据库。

理论上，充血模型是在聚合根这个层面上完成的持久化。但是由于Activiti没有采用O/R Mapping框架，于是自己完成了一个类似功能的模块。

1.2.2 PersistentObject

要明白工作流中的数据插入机制，首先要看下实体类的接口`org.activiti.engine.impl.db.PersistentObject`,如下

```
public interface PersistentObject {
  String getId();
  void setId(String id);
  Object getPersistentState();
}
```

Activiti的数据库表都是单字符串主键，每一个实体类都需要实现该接口。在对实体类进行保存的时候，DbSqlSession会调用getId方法判断是否存在ID，如果不存在，则使用ID生成器（该生成器根据策略有几种不同实现，这里不表）生成一个ID并且设置。

而方法`getPersistentState`是用来返回一个持久化状态对象。则方法的使用场合在下一个章节说明

1.2.3 DbSqlSession

该Session内部存在三个重要属性，如下

```
//该属性存储着所有使用insert方法放入的对象
protected Map<Class<? extends PersistentObject>, List<PersistentObject>>
insertedObjects = new HashMap<Class<? extends PersistentObject>,
List<PersistentObject>>();
//该Map结构内存储所有通过该DbSqlSession查询出来的结果，以及update方法放入的对象
protected Map<Class<?>, Map<String, CachedObject>> cachedObjects = new
HashMap<Class<?>, Map<String,CachedObject>>();
//该属性内存储着所有将要执行的删除操作
protected List<DeleteOperation> deleteOperations = new ArrayList<DeleteOperation>
();
```

删除和新增都比较容易理解，就是要此类操作缓存起来，一次性提交到数据库，上文曾提到的数据集中提交就体现在这个地方。而cachedObjects就有些不同了。要解析这个Map结构，首先来看下类`org.activiti.engine.impl.db.DbSqlSession.CachedObject`的结构属性，如下

```
public static class CachedObject {
    protected PersistentObject persistentObject;
    protected Object persistentObjectState;
}
public CachedObject(PersistentObject persistentObject, boolean storeState) {
      this.persistentObject = persistentObject;
      if (storeState) {
        this.persistentObjectState = persistentObject.getPersistentState();
      }
    }
```

通过构造方法可以明白，在新建该对象的时候，通过storeState参数决定是否保存当时的持久化状态。

该Map的数据来源有2处

- 所有通过该DbSqlSession对象执行查询的结果对象都会生成一个对应的CachedObject对象，并且storeState参数为true
- 执行该DbSqlSession对象的update类方法，会将参数用CachedObject对象包装起来，storeState参数为false。

当DbSqlSession执行flush方法时，主要来说是做了数据提交动作

1. 将insertObjects列表中的元素插入到数据库
2. 将deleteOperations列表中的元素遍历执行
3. 执行方法getUpdatedObjects获得要更新的实体对象

方法getUpdatedObjects的逻辑就是遍历所有的CachedObject，同时满足以下条件者则放入要更新的实体集合中

1. 实体的getPersistentState方法不为空
2. 实体的getPersistentState方法返回对象与CachedObject存储的persistentObjectState执行equal判断，结果为false

通过上面可以得知，如果一个实体类在DbSqlSession的生命周期被查询出来，并且其数据内容有了修改，则DbSqlSession刷新时会自动刷新到数据库。

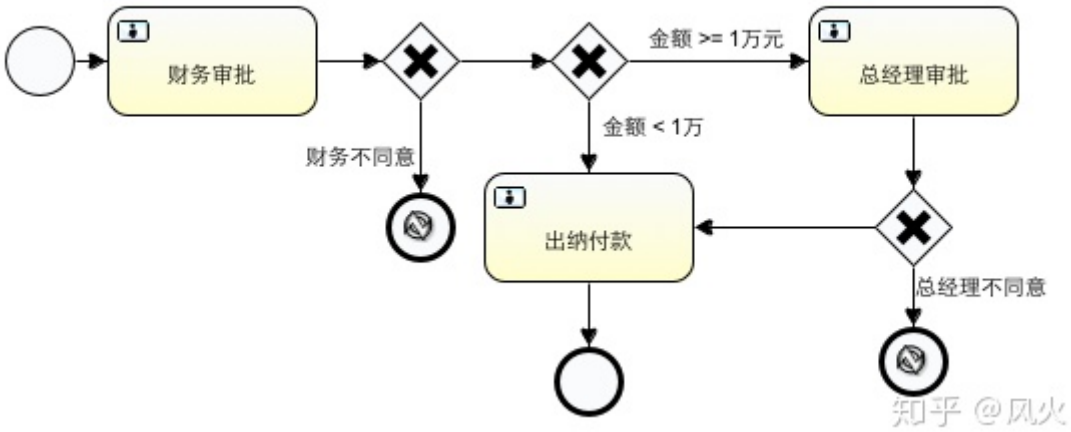
二、Activiti设计解析-PVM执行树

2.1 核心理念

任何框架都是核心理念上发展细化而来。Activiti的核心理念就是流程虚拟机（Process Virtual Machine，以下简称PVM）。PVM试图提供一组API，通过API本身来描述工作流方面的各种可能性。没有了具体实现，也使得PVM本身可以较好的适应各种不同的工作流领域语言，而Activiti本身也是在PVM上的一种实现。

2.1.1 PVM对流程定义期的描述

首先来看下流程定义本身。在工作流中，流程定义可以图形化的表达为一组节点和连接构成的集合。比如下图

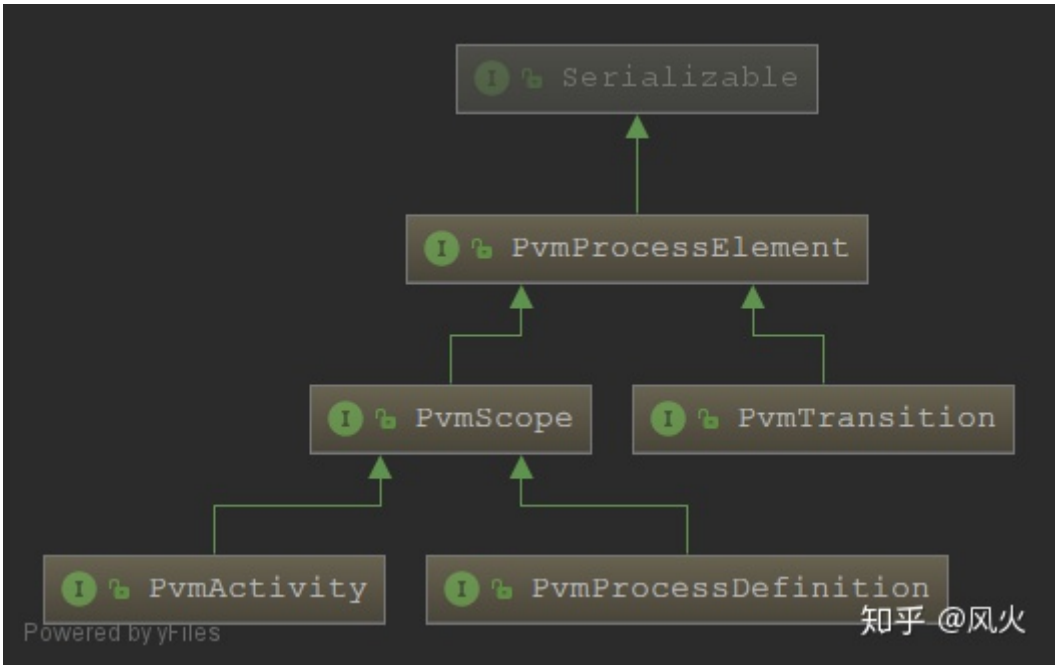


即使没有任何知识也能大概明白这张图表达的是一个流程以及执行顺序的意图。流程定义的表达方式不限，可以使用图形的方式表达，可以使用领域语言，也可以传统的XML（比如Activiti用的就是BPMN2.0 Schema下的XML）。特别的，当前已经有了标准化的BPMN2.0规范。

PVM将流程定义描述为流程元素的集合。再将流程元素细分为2个子类：流程节点和连线。

- 流程节点是某一种动作表达的抽象描述。节点本身是可以嵌套的，也就是节点可以拥有子节点。
- 连线表达是不同节点之间的转移关系。一个连线只能有一个源头节点和一个目标节点。而节点本身可以有任意多的进入连线和外出连线。

从类图的角度也能很好的看出这种关系，流程节点PvmActivity和连线PvmTransition都是流程元素PvmProcessElement。



从类图可以看到PvmActivity继承于PvmScope。这种继承关系表明流程节点本身有其归于的作用域（PvmScope），节点本身也可能是另外一些节点的作用域，这也符合节点可能拥有子节点的原则。关于作用域本身，后文还会再次详细讲解，这里先按下不表。

2.1.2 PVM对流程运行期的描述

通过流程节点和连线，PVM完成了对流程定义的表达。流程定义是一个流程的静态表达，流程执行则是依照流程定义启动的一个运行期表达，每一个流程执行都具备自己唯一的生命周期。流程执行需要具备以下要素：

1. 流程节点的具体执行动作。

2. 流程执行当前处于哪一个流程节点。
3. 流程执行是如何从一个节点运行至下一个节点。
4. 流程执行如何执行流程节点定义的执行动作

针对要素1，Activiti提供了接口`org.activiti.engine.impl.pvm.delegate.ActivityBehavior`。该接口内部仅有一个execute方法。该接口的实现即为不同PvmActivity节点提供了具体动作。ActivityBehavior有丰富的不同实现，对应了流程中丰富的不同功能的节点。每一个PvmActivity对象都会持有一个ActivityBehavior对象。

针对要素2，Activiti提供了接口`org.activiti.engine.impl.pvm.PvmExecution`。该接口有一个方法`PvmActivity getActivity()`。用以返回当前流程执行所处的流程节点。

针对要素3，Activiti提供了接口`org.activiti.engine.impl.pvm.runtime.InterpretableExecution`。接口方法很多，这里取和流程执行运转最重要的2个方法展开，如下

```
public interface InterpretableExecution extends ActivityExecution,
ExecutionListenerExecution, PvmProcessInstance {
  void take(PvmTransition transition);
  void take(PvmTransition transition, boolean fireActivityCompletedEvent);
```

执行方法take，以连线对象作为入参，这会使得流程执行该连线定义的路线。其实现逻辑应该为让流程执行定位于连线源头的活动节点，经由连线对象，到达连线目的地的活动节点。

针对要素4，实际上也是由接口`org.activiti.engine.impl.pvm.runtime.AtomicOperation`来完成的。通过该接口的调用类，此种情况的实现者需要获取当前流程执行所处的活动节点的`ActivityBehavior`对象，执行其`execute`方法来执行节点动作。结合要素3和4，可以看出`AtomicOperation`接口用于执行流程运转中的单一指令，例如根据连线移动，执行节点指令等。分解成单一指令的好处是易于编码和理解。这也契合接口命名中的原子一意。

2.1.3PVM综述

从上面对PVM定义期和运行期的解释可以看出，整个概念体系并不复杂。涉及到的类也不多。正是因为PVM只对工作流中最基础的部分做了抽象和接口定义，使得PVM的实现上有了很多的可能性。

然而也正是由于定义的简单性，实际上这套PVM在转化为实际实现的时候需要额外附加很多的特性才能真正完成框架需求。

2.2 ActivitiImpl与作用域

在解析完成后，一个流程定义中的所有节点都会被解析为ActivityImpl对象。ActivityImpl对象本身可以持有事件订阅（根据BPMN2.0规范，目前有定时，消息，信号三种事件订阅类型）。因为ActivityImpl本身可以嵌套并且可以持有订阅，因此引入作用域概念（Scope）。

一个ActivityImpl在以下两种情况下会被定义为作用域ActivityImpl。

1. 该ActivityImpl是可变范围，则它是一个作用域。可变范围可以理解为该节点的内容定义是可变的。比如流程定义、子流程，其内部内容是可变的。根据BPMN定义，可变范围有：流程定义，子流程，多实例，调用活动。
2. 该ActivityImpl定义了一个上下文用于接收事件。比如：具备边界事件的ActivityImpl，具备事件子流程的ActivityImpl，事件驱动网关，中间事件捕获ActivityImpl。

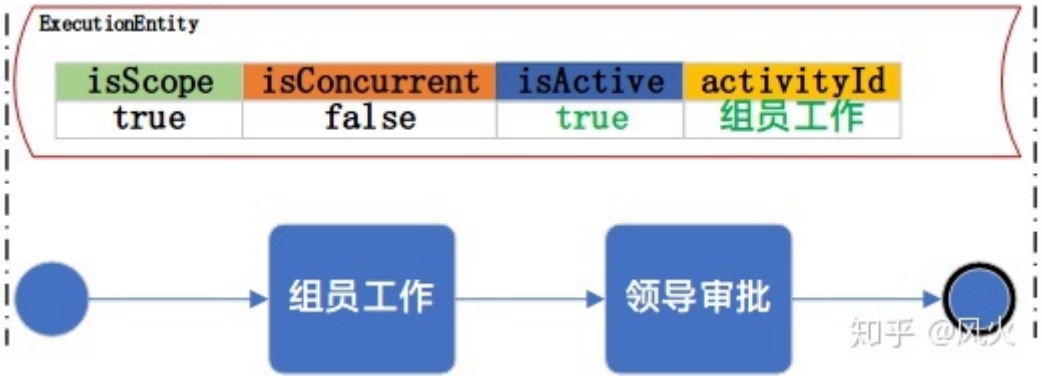
作用域是一个很重要的概念，情况1中作用域定义的是复杂节点的生命周期，情况2中作用域定义的是事件的捕获范围。

2.3 ExecutionEntity

ExecutionEntity的含义是一个流程定义被启动后的执行实例，代表着流程的运行期状态。在Activiti的设计中，**事件订阅，流程变量等都是与一个具体的ExecutionEntity相关的。**其本身有几个重要的属性：

- isScope：该属性为真时，意味该执行实例在执行一个具备作用域的ActivityImpl节点或者执行一个流程定义。更简单一些，意味着该实例正在执行一个作用域活动。
- isConcurrent：该属性为真时，意味与该执行实例正在执行的活动节点同属相同作用域的节点可能正并发被其他执行实例执行（比如并行网关后面的2个并行任务）。
- isActive：该属性为真时，意味该执行实例正在执行一个简单ActivityImpl（不包含其他ActivityImpl的ActivityImpl）
- isEventScope：该属性为真时，意味该执行实例是为了后期补偿而进行的变量保存所创建的执行实例。由于流程执行中的变量都需要与ExecutionEntity挂钩，而补偿是需要原始变量的快照。为了满足这个需求，创建出一个专用于此的ExecutionEntity。
- activityId：该ExecutionEntity正在执行的ActivityImpl的id。正在执行意味着几种情况：进入该节点，执行该节点动作，离开该节点。如果是等待子流程的完成，则该属性为null。

上面对ExecutionEntity的解释仍然抽象。如果直观的看，可以认为ExecutionEntity是某一种生命周期的体现，其内部属性随着不同的情况而变化。如下图所示：



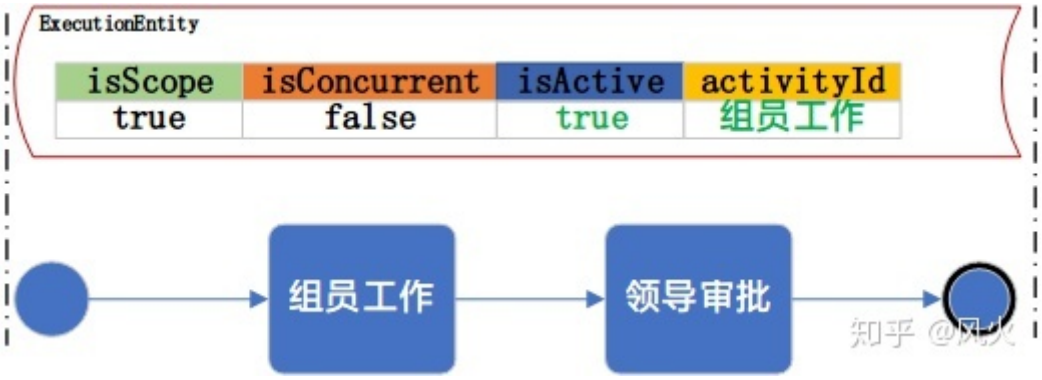
随着流程的启动，会创建一个`ExecutionEntity`对象。该`ExecutionEntity`生命周期与整个流程相同，而其中的`isScope`和`isConcurrent`在创建之初固定，并且不会改变。而`isActive`和`activityId`随着流程的推进则会不断变化。

`ExecutionEntity`是用来反映流程的推进情况的，实际上，往往一个`ExecutionEntity`不足以支撑全部的BPMN功能。因此实现上，Activiti是通过一个树状结构的`ExecutionEntity`结构来反映流程推进情况。创建之初的`ExecutionEntity`对象随着流程的推进会不断的分裂和合并，`ExecutionEntity`树也会不断的生长和修剪。在流程的推进过程中会遇到4种基本情况

1. 单独的非作用域节点
2. 单独的作用域节点
3. 并发的非作用域节点
4. 并发的作用域节点

2.3.1 单独的非作用域节点

此种情况可以如下图所示

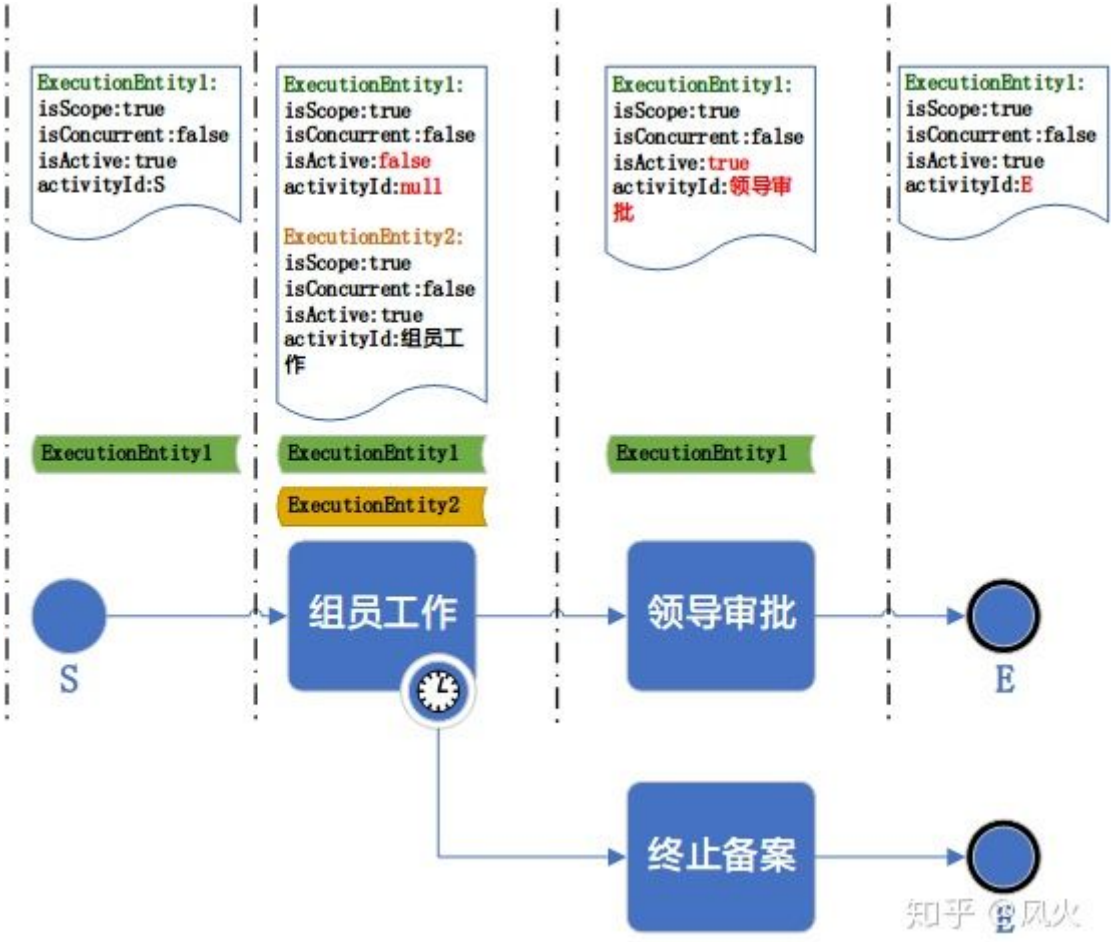


在流程前进的构成中，遇到单独的非作用域节点，`ExecutionEntity`一直处于激活状态，只不过随着流程前进，其activityId指向会不断变化。如上图所示，会经历：开始节点、组员工作、领导审批、结束节点4个不同的值。

实际上，这些节点都是在作用域<流程定义>之下，而`ExecutionEntity`代表的正是该作用域，因此其isScope属性为true。

2.3.2 单独的作用域节点

如果流程推进中遇到单独的作用域节点，则当前执行对象`ExecutionEntity`应该创建一个作用域子执行（isScope为true的ExecutionEntity）。整个变化过程可以如下图所示



当准备进入节点<组员工作>时，`ExecutionEntity1`冻结，并且创建出子执行`ExecutionEntity2`。`ExecutionEntity2`的isScope属性也为true。

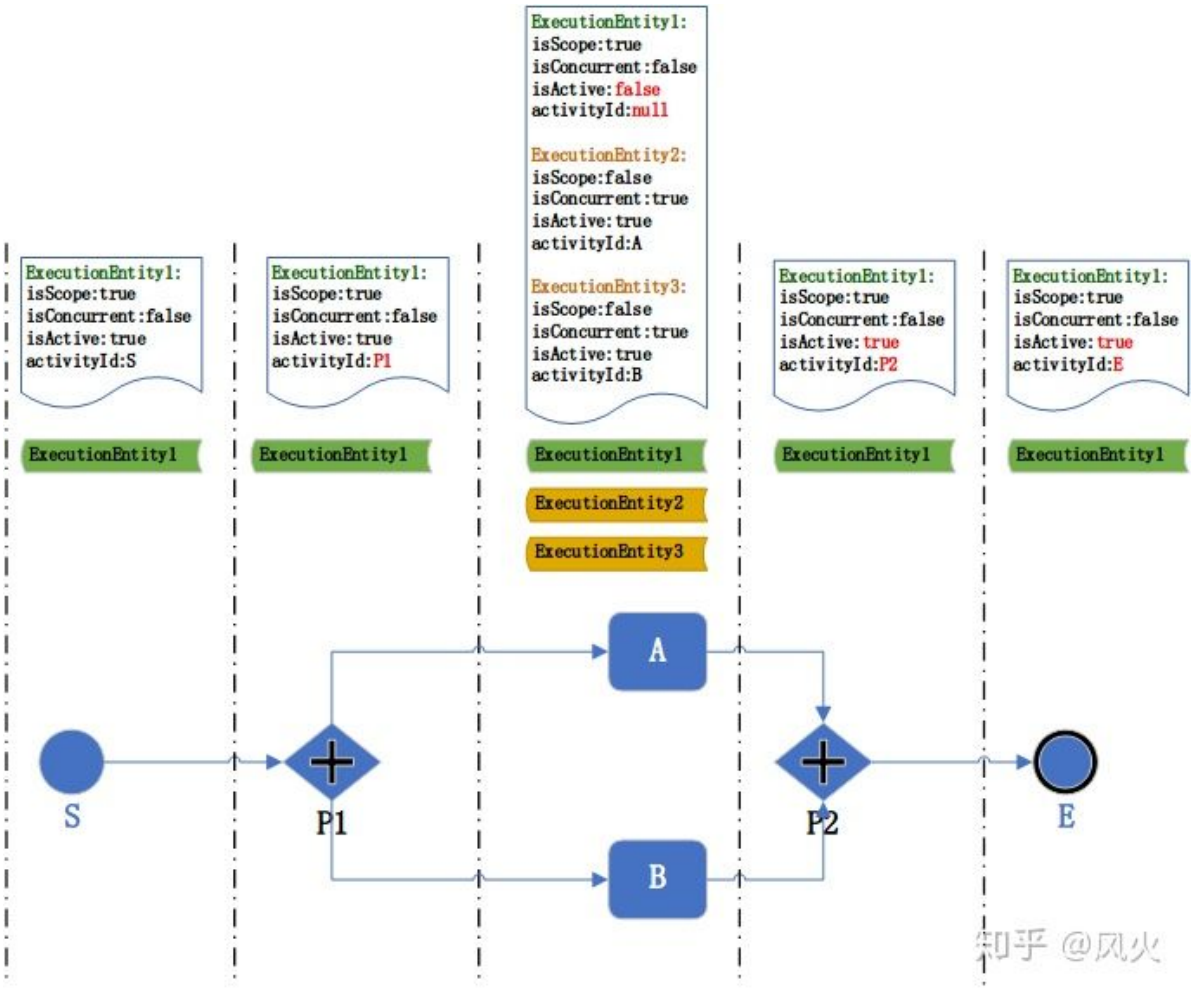
`ExecutionEntity1`的isScope为true，是因为该执行实例负责整个流程定义的事件订阅，`ExecutionEntity2`的isScope为true，是因为该执行实例负责节点<组员工作>的事件订阅。

前面提到过，事件订阅与某一个具体的执行实例相关。当节点<组员工作>完成时，也就是事件订阅所在的作用域要被摧毁时，对应的事件订阅也要被删除。此时额外的 `ExecutionEntity` 就特别方便，只要删除该 `ExecutionEntity`，顺便删除相关的事件订阅即可，在这里就是删除 `ExecutionEntity2`。

删除 `ExecutionEntity2`，并且激活父执行 `ExecutionEntity1`。随着流程推进，`ExecutionEntity1` 更换指向的activityId。

2.3.3 并发的非作用域节点

流程推进中节点存在多个外出连线，则可以根据需要创建多个并发的子执行，每一个子执行对应一个连线。如下图所示

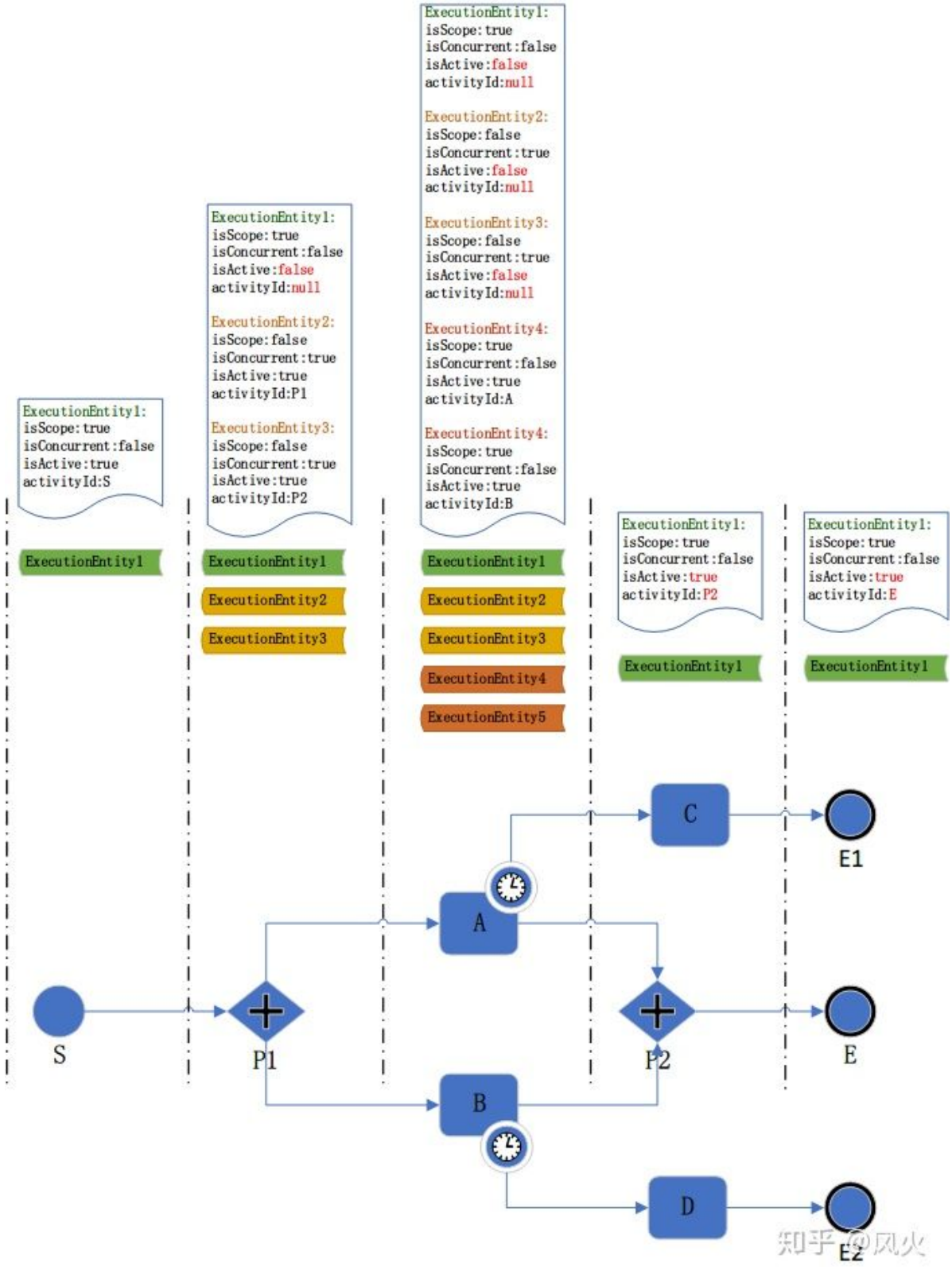


当流程节点A、B被激活时，`ExecutionEntity1` 会有2个并发的子执行 `ExecutionEntity2` 和 `ExecutionEntity3`。这两个子执行的 `isConcurrent` 属性均为true，因为节点A和B都是在相同的作用域（流程定义）下被并发的执行。

当节点A、B执行完毕后，并发的子执行被删除，父执行重新被激活，继续后面的节点。

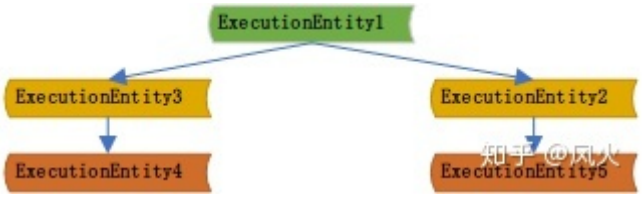
2.3.4 并发的作用域节点

流程推进中遇到并发节点，并且节点为作用域节点，情况就会如下所示



当流程运行至P1节点，P1节点有多个出线。根据出线数目创建2个子执行，此时2个子执行均为并发的，且从P1作为出线的源头节点，因此2个子执行的activityId均为P1。

当运行到A、B节点时，由于2个节点均为作用域节点，因此还会再创建2个子执行。此时 `ExecutionEntity3` 和 `ExecutionEntity3` 冻结。 `ExecutionEntity4` 和 `ExecutionEntity5` 所执行的节点在各自的作用域下均无并发操作，因此其isScope属性为true，isConcurrent属性为false。这5个执行实例构成的执行树如下



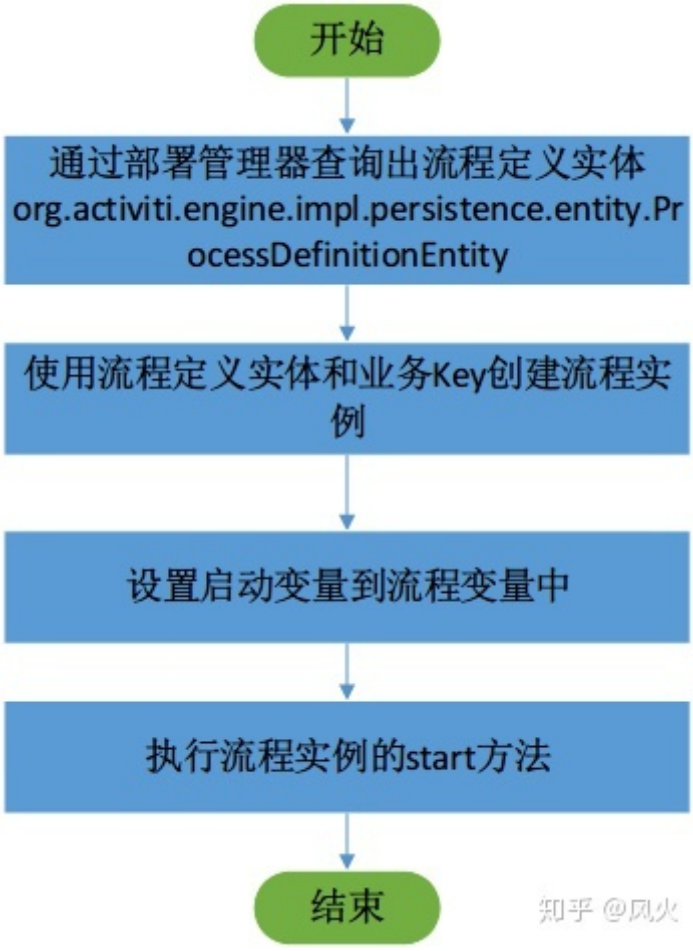
当A、B节点完成时，首先是各自的作用域被删除，因此`ExecutionEntity4`和`ExecutionEntity5`首先被删除，`ExecutionEntity3`和`ExecutionEntity4`激活。而后汇聚于P2节点，因此`ExecutionEntity3`和`ExecutionEntity4`删除，`ExecutionEntity1`被激活，继续执行剩下的节点。

三、代码解析-流程启动

3.1 流程说明

流程启动依靠的是命令类：`org.activiti.engine.impl.cmd.StartProcessInstanceCmd`。

该命令的整体流程如下



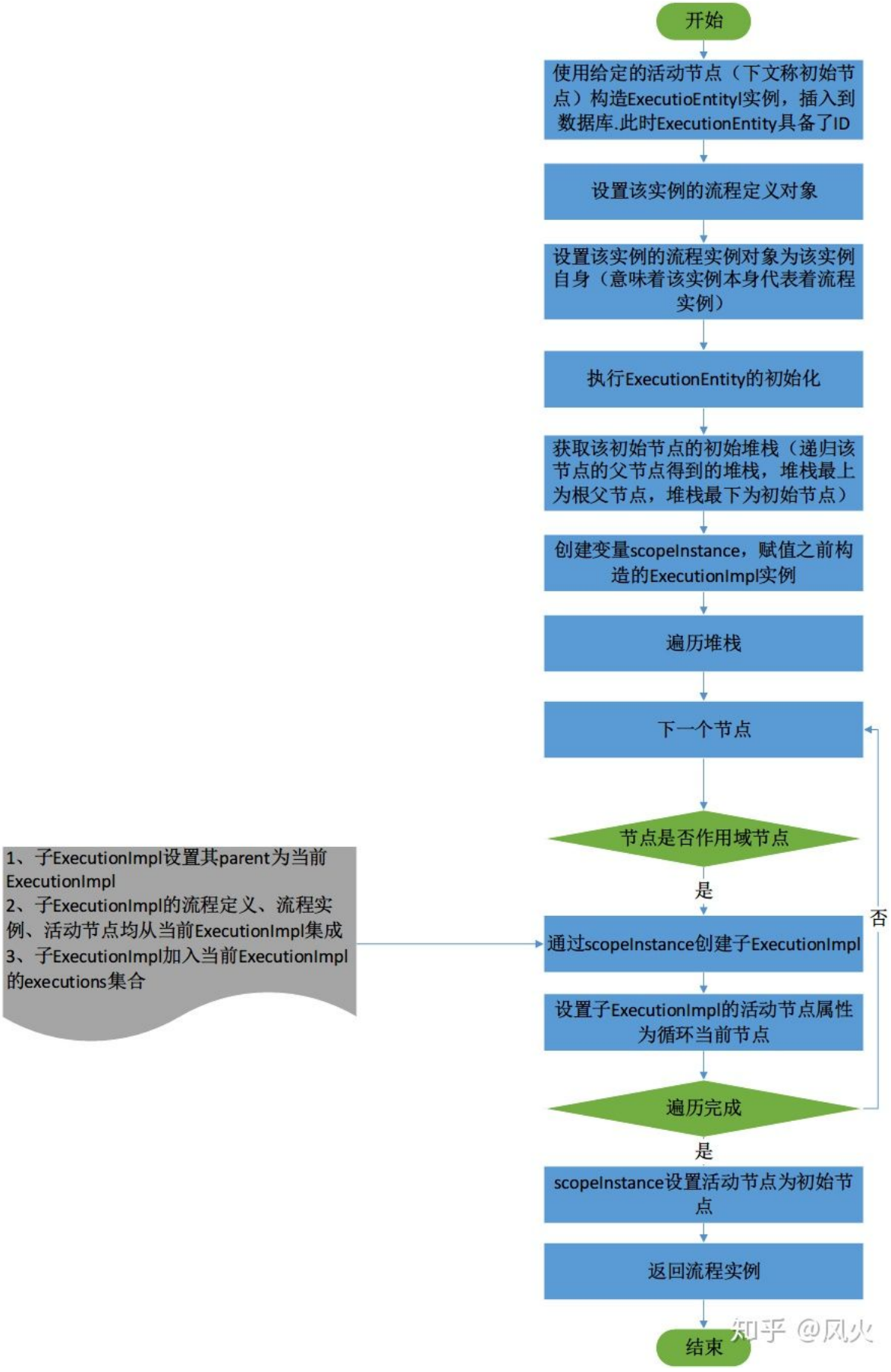
部署管理器的查询和运行期关系不大，先忽略。两个流程着重展开：

1. 流程定义实体创建流程实例
2. 流程实例执行启动

3.1.1 流程定义实体创建流程实例

流程如下

其中**指定初始滑动节点创建实例**本身展开后的流程如下



ExecutionEntity的初始化需要单独说下，流程如下

在创建流程的逻辑的尾部是一个循环流程。该循环的目的是为了创建正确的ExecutionImpl树（以下简称执行树）。本质上该方法是创建一个流程实例，并且将流程当前运行节点定位到指定的节点。而工作流的正确执行依赖于执行树的正确分裂和整合。因此就需要为指定的节点创建其上游的执行树实例。使得在效果上看起来就和流程自动执行到当前节点类似（执行树类似，节点运行历史则无相似，实际上也无历史节点）。

而如果指定的初始节点就是流程定义的初始节点，则循环就不存在意义了。

3.1.2 流程实例启动

流程实例的启动的内容，就是执行原子操作: `org.activiti.engine.impl.pvm.runtime.AtomicOperationProcessStart`。关于原子操作单独阐述。

3.2 额外补充

3.2.1 ActivityImpl的parent属性

`org.activiti.engine.impl.pvm.process.ActivityImpl`类中有一个属性parent，类型为`org.activiti.engine.impl.pvm.process.ScopeImpl`。在解析的时候，该属性为当前节点的作用域节点。根据作用域节点的定义，该属性的取值有两种可能的类型，一种是`org.activiti.engine.impl.pvm.process.ActivityImpl`，另外一种是

`org.activiti.engine.impl.pvm.process.ProcessDefinitionImpl`。

第二种情况意味着该节点是直属于流程定义的节点了。

四、代码解析-原子操作

4.1 说明

原子操作是一个接口`org.activiti.engine.impl.pvm.runtime.AtomicOperation`。从名字也可以看出，该接口的作用就是执行流程实例中的一个单步操作。下面分阶段说明

4.2 AbstractEventAtomicOperation

该抽象类是众多实现类的基类。其代码如下

```
public abstract class AbstractEventAtomicOperation implements AtomicOperation {

  public boolean isAsync(InterpretableExecution execution) {
    return false;
  }

  public void execute(InterpretableExecution execution) {
      //获取当前执行对象的作用域对象。具体由子类提供。
    ScopeImpl scope = getScope(execution);
      //从作用域对象中获取指定事件的监听器。事件名称由子类提供。
    List<ExecutionListener> exectionListeners =
scope.getExecutionListeners(getEventName());
    int executionListenerIndex = execution.getExecutionListenerIndex();

    if (exectionListeners.size()>executionListenerIndex) {
      execution.setEventName(getEventName());
      execution.setEventSource(scope);
      ExecutionListener listener =
exectionListeners.get(executionListenerIndex);
      try {
        listener.notify(execution);
      } catch (RuntimeException e) {
        throw e;
      } catch (Exception e) {
        throw new PvmException("couldn't execute event listener :
"+e.getMessage(), e);
      }
      execution.setExecutionListenerIndex(executionListenerIndex+1);
      execution.performOperation(this);

    } else {
      execution.setExecutionListenerIndex(0);
      execution.setEventName(null);
      execution.setEventSource(null);

      eventNotificationsCompleted(execution);
    }
  }

  protected abstract ScopeImpl getScope(InterpretableExecution execution);
  protected abstract String getEventName();
```

```
  protected abstract void eventNotificationsCompleted(InterpretableExecution
execution);
}
```

整个抽象类的逻辑概括而言，就是将获取当前执行实例的作用域对象（具体由子类提供），执行其中特定事件（事件名由子类提供）的监听器。

在全部的监听器执行完毕后，执行子类的特定逻辑。

4.3 AtomicOperationProcessStart

该操作用于流程启动。但是并不执行真正的启动动作。只是设置了当前执行实例的活动节点为`org.activiti.engine.impl.pvm.runtime.StartingExecution`中存储的活动节点。然后执行原子操作`org.activiti.engine.impl.pvm.runtime.AtomicOperationProcessStartInitial`。

本质上来说，只是执行了一个设置的动作。

```
public class AtomicOperationProcessStart extends AbstractEventAtomicOperation {

  @Override
  protected ScopeImpl getScope(InterpretableExecution execution) {
    return execution.getProcessDefinition();
  }

  @Override
  protected String getEventName() {
    return org.activiti.engine.impl.pvm.PvmEvent.EVENTNAME_START;
  }

  @Override
  protected void eventNotificationsCompleted(InterpretableExecution execution) {
      if (Context.getProcessEngineConfiguration() != null &&
Context.getProcessEngineConfiguration().getEventDispatcher().isEnabled()) {
        Map<String, Object> variablesMap = null;
        try {
          variablesMap = execution.getVariables();
        } catch (Throwable t) {
          // In some rare cases getting the execution variables can fail (JPA
entity load failure for example)
          // We ignore the exception here, because it's only meant to include
variables in the initialized event.
        }

Context.getProcessEngineConfiguration().getEventDispatcher().dispatchEvent(

ActivitiEventBuilder.createEntityWithVariablesEvent(ActivitiEventType.ENTITY_INI
TIALIZED,
                    execution, variablesMap, false));
      Context.getProcessEngineConfiguration().getEventDispatcher()

.dispatchEvent(ActivitiEventBuilder.createProcessStartedEvent(execution,
variablesMap, false));
    }

    ProcessDefinitionImpl processDefinition = execution.getProcessDefinition();
    StartingExecution startingExecution = execution.getStartingExecution();
```

```
    List<ActivityImpl> initialActivityStack =
processDefinition.getInitialActivityStack(startingExecution.getInitial());
    execution.setActivity(initialActivityStack.get(0));
    execution.performOperation(PROCESS_START_INITIAL);
  }
}
```

4.4 AtomicOperationProcessStartInitial

代码如下

```
public class AtomicOperationProcessStartInitial extends
AbstractEventAtomicOperation {

  @Override
  protected ScopeImpl getScope(InterpretableExecution execution) {
    return (ScopeImpl) execution.getActivity();
  }

  @Override
  protected String getEventName() {
    return org.activiti.engine.impl.pvm.PvmEvent.EVENTNAME_START;
  }

  @Override
  protected void eventNotificationsCompleted(InterpretableExecution execution) {
    ActivityImpl activity = (ActivityImpl) execution.getActivity();
    ProcessDefinitionImpl processDefinition = execution.getProcessDefinition();
    StartingExecution startingExecution = execution.getStartingExecution();
     //从开始节点开始的，该判断均为真。
    if (activity==startingExecution.getInitial()) {
      execution.disposeStartingExecution();
      execution.performOperation(ACTIVITY_EXECUTE);
    } else {
      List<ActivityImpl> initialActivityStack =
processDefinition.getInitialActivityStack(startingExecution.getInitial());
      int index = initialActivityStack.indexOf(activity);
      activity = initialActivityStack.get(index+1);

      InterpretableExecution executionToUse = null;
      if (activity.isScope()) {
        executionToUse = (InterpretableExecution)
execution.getExecutions().get(0);
      } else {
        executionToUse = execution;
      }
      executionToUse.setActivity(activity);
      executionToUse.performOperation(PROCESS_START_INITIAL);
    }
  }
}
```

4.5 AtomicOperationTransitionNotifyListenerEnd

该原子操作的目的仅是为了执行节点上的`end`事件监听器。监听器执行完毕后，就执行下一个原子操作`AtomicOperationTransitionDestroyScope`

4.6 AtomicOperationTransitionNotifyListenerStart

该原子操作的目的是为了执行节点上start事件监听器。在执行完毕后，会判断执行实例的当前节点是否可以执行。判断的依据该节点和连接线节点

4.3 AtomicOperationActivityExecute

该原子操作的作用实际上就是取出该执行实例当前的活动节点，并且执行该活动节点的行为定义。行为定义通过接口`org.activiti.engine.impl.pvm.delegate.ActivityBehavior`定义。不同的节点行为由不同的子类完成

```
public class AtomicOperationActivityExecute implements AtomicOperation {

  private static Logger log =
LoggerFactory.getLogger(AtomicOperationActivityExecute.class);

  public boolean isAsync(InterpretableExecution execution) {
    return false;
  }

  public void execute(InterpretableExecution execution) {
    ActivityImpl activity = (ActivityImpl) execution.getActivity();

    ActivityBehavior activityBehavior = activity.getActivityBehavior();
    if (activityBehavior==null) {
      throw new PvmException("no behavior specified in "+activity);
    }

    log.debug("{} executes {}: {}", execution, activity,
activityBehavior.getClass().getName());

    try {
        if(Context.getProcessEngineConfiguration() != null &&
Context.getProcessEngineConfiguration().getEventDispatcher().isEnabled()) {

Context.getProcessEngineConfiguration().getEventDispatcher().dispatchEvent(

ActivitiEventBuilder.createActivityEvent(ActivitiEventType.ACTIVITY_STARTED,
                          execution.getActivity().getId(),
                          (String) execution.getActivity().getProperty("name"),
                          execution.getId(),
                          execution.getProcessInstanceId(),
                          execution.getProcessDefinitionId(),
                          (String) activity.getProperties().get("type"),

activity.getActivityBehavior().getClass().getCanonicalName()));
      }

      activityBehavior.execute(execution);
    } catch (RuntimeException e) {
      throw e;
```

```
    } catch (Exception e) {
      LogMDC.putMDCExecution(execution);
      throw new PvmException("couldn't execute activity
<"+activity.getProperty("type")+" id=\""+activity.getId()+"\" ...>:
"+e.getMessage(), e);
    }
  }
}
```

4.4 AtomicOperationTransitionDestroyScope

```
public class AtomicOperationTransitionDestroyScope implements AtomicOperation {

  private static Logger log =
LoggerFactory.getLogger(AtomicOperationTransitionDestroyScope.class);

  public boolean isAsync(InterpretableExecution execution) {
    return false;
  }

  @SuppressWarnings("unchecked")
  public void execute(InterpretableExecution execution) {
    InterpretableExecution propagatingExecution = null;

    ActivityImpl activity = (ActivityImpl) execution.getActivity();
    /**
    * 如果当前的活动节点具备作用域。这就意味着最初的时候，有一个处于激活状态的执行实例在执行该节点，以下称初始执行实例。
    * 从这样的节点退出要考虑几种情况：
    * 一、单独的作用域节点。此时的执行树情况是：非激活的非作用域执行实例-激活的作用域执行实例（初始执行实例）。此时要离开作用域节点，首先是销毁激活的作用域执行实例（初始执行实例），激活初始执行实例的父实例，使用该父实例执行后续的出线动作。
    * 二、并行的作用域节点。此时的执行树情况是：非激活的非作用域并发执行实例-非激活的
    */
    if (activity.isScope()) {

      InterpretableExecution parentScopeInstance = null;
      // if this is a concurrent execution crossing a scope boundary
      if (execution.isConcurrent() && !execution.isScope()) {
        // first remove the execution from the current root
        InterpretableExecution concurrentRoot = (InterpretableExecution)
execution.getParent();
        parentScopeInstance = (InterpretableExecution)
execution.getParent().getParent();

        log.debug("moving concurrent {} one scope up under {}", execution,
parentScopeInstance);
        List<InterpretableExecution> parentScopeInstanceExecutions =
(List<InterpretableExecution>) parentScopeInstance.getExecutions();
        List<InterpretableExecution> concurrentRootExecutions =
(List<InterpretableExecution>) concurrentRoot.getExecutions();
        // if the parent scope had only one single scope child
        if (parentScopeInstanceExecutions.size()==1) {
          // it now becomes a concurrent execution
          parentScopeInstanceExecutions.get(0).setConcurrent(true);
        }
```

```
        concurrentRootExecutions.remove(execution);
        parentScopeInstanceExecutions.add(execution);
        execution.setParent(parentScopeInstance);
        execution.setActivity(activity);
        propagatingExecution = execution;

        // if there is only a single concurrent execution left
        // in the concurrent root, auto-prune it.  meaning, the
        // last concurrent child execution data should be cloned into
        // the concurrent root.
        if (concurrentRootExecutions.size()==1) {
          InterpretableExecution lastConcurrent =
concurrentRootExecutions.get(0);
          if (lastConcurrent.isScope()) {
            lastConcurrent.setConcurrent(false);

          } else {
            log.debug("merging last concurrent {} into concurrent root {}",
lastConcurrent, concurrentRoot);

            // We can't just merge the data of the lastConcurrent into the
concurrentRoot.
            // This is because the concurrent root might be in a takeAll-loop.
So the
            // concurrent execution is the one that will be receiving the take
            concurrentRoot.setActivity((ActivityImpl)
lastConcurrent.getActivity());
            concurrentRoot.setActive(lastConcurrent.isActive());
            lastConcurrent.setReplacedBy(concurrentRoot);
            lastConcurrent.remove();
          }
        }

      } else if (execution.isConcurrent() && execution.isScope()) {
        /**
        * 根据算法，这种情况不会出现。源代码中，这部分也属于todo的内容。
        */
      }
        else {
        /**
        * 这个条件是执行实例的scope属性为真。此时销毁当前的执行实例，使用其父执行实例继续后面
的流程
        */
        propagatingExecution = (InterpretableExecution) execution.getParent();
        propagatingExecution.setActivity((ActivityImpl)
execution.getActivity());
        propagatingExecution.setTransition(execution.getTransition());
        propagatingExecution.setActive(true);
        log.debug("destroy scope: scoped {} continues as parent scope {}",
execution, propagatingExecution);
        execution.destroy();
        //删除与该执行实例相关的一切，包括：定时工作，各种任务，事件订阅，用户流程关系，最后删
除自身。
        execution.remove();
      }
    } else {
      //如果离开的是一个非作用域节点，则仍然使用当前的执行实例作为下一个节点的执行实例
      propagatingExecution = execution;
```

```
    }

    // if there is another scope element that is ended
    ScopeImpl nextOuterScopeElement = activity.getParent();
    TransitionImpl transition = propagatingExecution.getTransition();
    ActivityImpl destination = transition.getDestination();
    /**
    * 考虑当前的节点可能是子流程或者活动调用中的节点。那么就需要离开当前的作用域范围，回到更上层的作用域下。因此需要判断目的地是否和源头节点处于同一个作用域。如果不是同一个作用域，则不断向上回溯
    */
    if (transitionLeavesNextOuterScope(nextOuterScopeElement, destination)) {
      propagatingExecution.setActivity((ActivityImpl) nextOuterScopeElement);
      propagatingExecution.performOperation(TRANSITION_NOTIFY_LISTENER_END);
    } else {
      propagatingExecution.performOperation(TRANSITION_NOTIFY_LISTENER_TAKE);
    }
  }

  public boolean transitionLeavesNextOuterScope(ScopeImpl nextScopeElement,
ActivityImpl destination) {
    return !nextScopeElement.contains(destination);
  }
}
```

4.5 AtomicOperationTransitionNotifyListenerTake

该原子操作的目的就是为了执行在连接线上的监听器。在执行完毕后，就准备执行目标活动节点。这里关于目标节点还存在一个选择的问题。并不是直接执行连接线上的目标活动节点。而是从目标活动节点出发，选择和执行实例当前活动节点同属同一个作用域的目标活动节点或其父节点。

```
public class AtomicOperationTransitionNotifyListenerTake implements
AtomicOperation {

  private static Logger log =
LoggerFactory.getLogger(AtomicOperationTransitionNotifyListenerTake.class);

  public boolean isAsync(InterpretableExecution execution) {
    return false;
  }

  public void execute(InterpretableExecution execution) {
    TransitionImpl transition = execution.getTransition();

    List<ExecutionListener> executionListeners =
transition.getExecutionListeners();
    int executionListenerIndex = execution.getExecutionListenerIndex();
    /**
    * 整个if的功能就是不断判断监听器是否被执行完毕。都执行完毕后走入到else的部分。
    */
    if (executionListeners.size()>executionListenerIndex) {

execution.setEventName(org.activiti.engine.impl.pvm.PvmEvent.EVENTNAME_TAKE);
      execution.setEventSource(transition);
      ExecutionListener listener =
executionListeners.get(executionListenerIndex);
      try {
```

```
        listener.notify(execution);
      } catch (RuntimeException e) {
        throw e;
      } catch (Exception e) {
        throw new PvmException("couldn't execute event listener :
"+e.getMessage(), e);
      }
      execution.setExecutionListenerIndex(executionListenerIndex+1);
      execution.performOperation(this);

    } else {
        if (log.isDebugEnabled()) {
            log.debug("{} takes transition {}", execution, transition);
        }
      execution.setExecutionListenerIndex(0);
      execution.setEventName(null);
      execution.setEventSource(null);

      ActivityImpl activity = (ActivityImpl) execution.getActivity();
      ActivityImpl nextScope = findNextScope(activity.getParent(),
transition.getDestination());
      execution.setActivity(nextScope);

      // Firing event that transition is being taken
      if(Context.getProcessEngineConfiguration() != null &&
Context.getProcessEngineConfiguration().getEventDispatcher().isEnabled()) {

Context.getProcessEngineConfiguration().getEventDispatcher().dispatchEvent(

ActivitiEventBuilder.createSequenceFlowTakenEvent(ActivitiEventType.SEQUENCEFLOW
_TAKEN, transition.getId(),
                        activity.getId(), (String)
activity.getProperties().get("name") ,(String)
activity.getProperties().get("type"),
activity.getActivityBehavior().getClass().getCanonicalName(),
                        nextScope.getId(), (String)
nextScope.getProperties().get("name"), (String)
nextScope.getProperties().get("type"),
nextScope.getActivityBehavior().getClass().getCanonicalName()));
      }

      execution.performOperation(TRANSITION_CREATE_SCOPE);
    }
  }

  /** finds the next scope to enter.  the most outer scope is found first */
  public static ActivityImpl findNextScope(ScopeImpl outerScopeElement,
ActivityImpl destination) {
    ActivityImpl nextScope = destination;
    while( (nextScope.getParent() instanceof ActivityImpl)
           && (nextScope.getParent() != outerScopeElement)
         ) {
      nextScope = (ActivityImpl) nextScope.getParent();
    }
    return nextScope;
  }
}
```

4.6 AtomicOperationTransitionCreateScope

该原子操作是为了确认进入的节点是否具备作用域。如果具备作用域，则将目前的执行实例冻结。并且创建出新的执行实例，用于执行作用域节点；如果不具备作用域，则无效果。

在确认完毕后，执行原子操作`AtomicOperationTransitionNotifyListenerStart`

```
public class AtomicOperationTransitionCreateScope implements AtomicOperation {

  private static Logger log =
LoggerFactory.getLogger(AtomicOperationTransitionCreateScope.class);

  public boolean isAsync(InterpretableExecution execution) {
    ActivityImpl activity = (ActivityImpl) execution.getActivity();
    return activity.isAsync();
  }

  public void execute(InterpretableExecution execution) {
    InterpretableExecution propagatingExecution = null;
    ActivityImpl activity = (ActivityImpl) execution.getActivity();
    if (activity.isScope()) {
      //为作用域活动创建一个新的执行实例，是该原子操作的主要目的
      propagatingExecution = (InterpretableExecution)
execution.createExecution();
      propagatingExecution.setActivity(activity);
      propagatingExecution.setTransition(execution.getTransition());
      execution.setTransition(null);
      execution.setActivity(null);
      execution.setActive(false);
      log.debug("create scope: parent {} continues as execution {}", execution,
propagatingExecution);
      //这里是另外一个重点。在一个流程实例初始化的时候，会对当前流程所处的作用域对象（可能是流程定义或者是作用域活动进行处理，具体表现是为该作用域对象上的定时事件，消息事件，信号事件执行注册动作。分别是放入定时调度器，在数据库新增事件订阅）
      propagatingExecution.initialize();

    } else {
      propagatingExecution = execution;
    }


propagatingExecution.performOperation(AtomicOperation.TRANSITION_NOTIFY_LISTENER
_START);
  }
}
```

发布于 2019-12-22 00:53

Activiti

软件架构

源代码