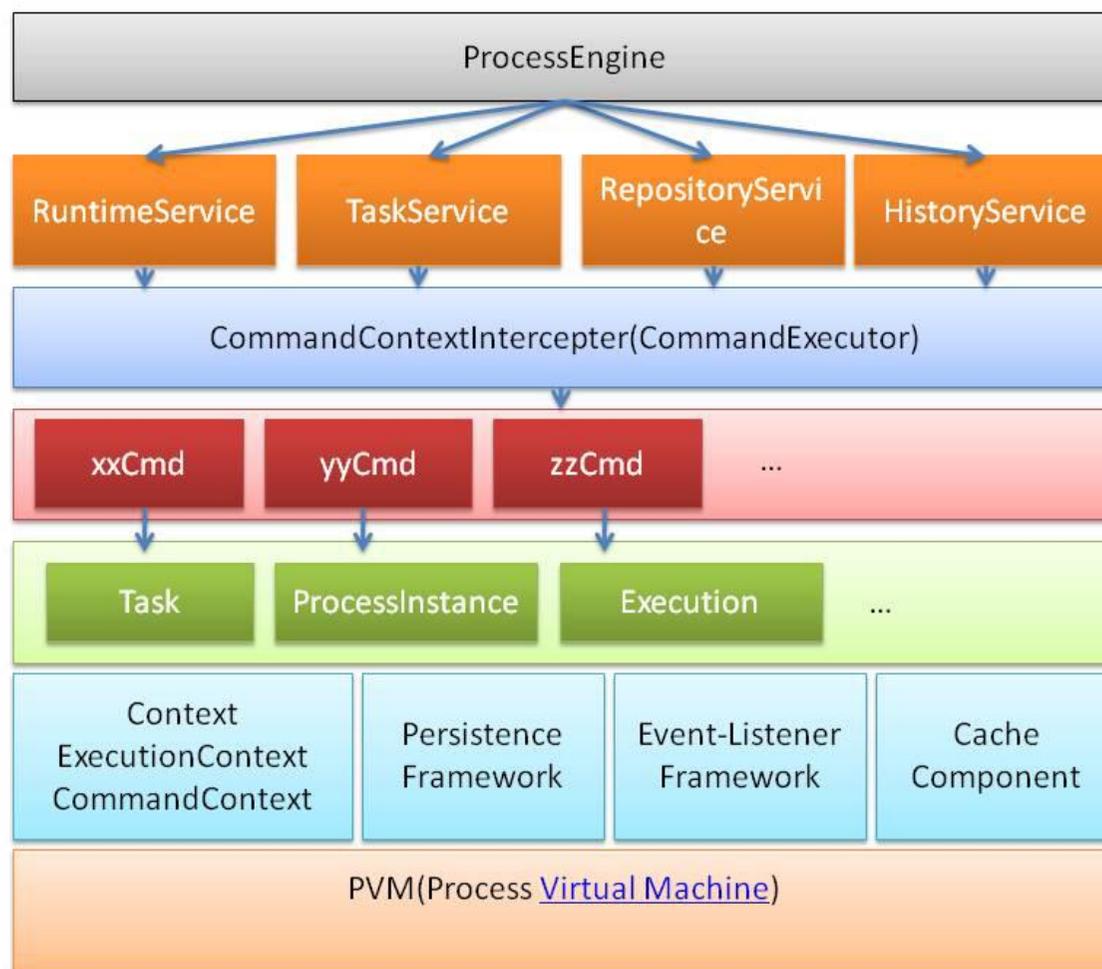


Activiti 源码分析

Activiti 的基础编程框架



Activiti 基于 Spring, ibatis 等开源中间件作为软件平台，在此之上构建了非常清晰的开发框架。上图列出了 Activiti 的核心组件。

1.ProcessEngine: 流程引擎的抽象，对于开发者来说，它是我们使用 Activiti 的 facade，通过它可以获得我们需要的一切服务。

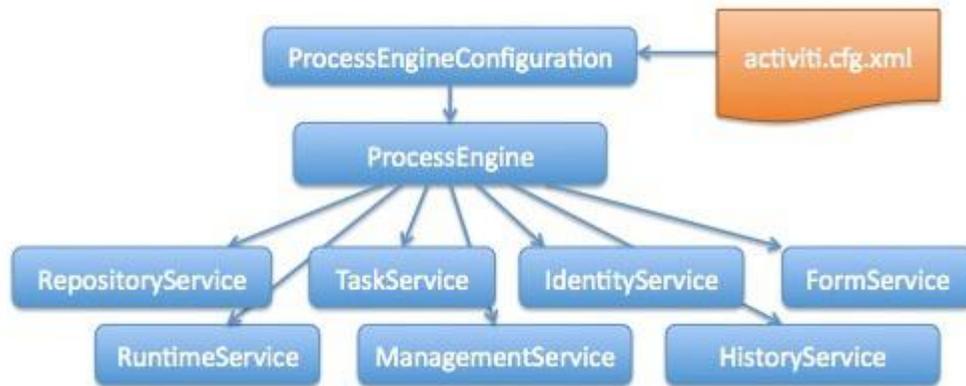
2.XXService (TaskService,RuntimeService,RepositoryService...):Activiti 按照流程的生命周期(定义, 部署, 运行)把不同阶段的服务封装在不同的 Service 中, 用户可以非常清晰地使用特定阶段的接口。通过 ProcessEngine 能够获得这些 Service 实例。TaskService,RuntimeService,RepositoryService 是非常重要的三个 Service:

TaskService: 流程运行过程中, 与每个任务节点相关的接口, 比如 complete,delete,delegate 等等

RepositoryService:流程定义和部署相关的存储服务。

RuntimeService: 流程运行时相关服务, 如 startProcessInstanceByKey.

关于 ProcessEngine 和 XXService 的关系, 可以看下面这张图:



3.CommandContextInterceptor(CommandExecutor):Activiti 使用命令模式作为基础开发模式, 上面 Service 中定义的各个方法都对应相应的命令对象 (xxCmd), Service 把各种请求委托给 xxCmd, xxCmd 来决定命令的接收者, 接收者执行后返回结果。而 CommandContextInterceptor 顾名思义, 它是一个拦截器, 拦截所有命令, 在命令执行前后执行一些公共性操作。比如 CommandContextInterceptor 的核心方法:

Java 代码 

```
1. public <T> T execute(Command<T> command) {
```

```
2.   CommandContext context = commandContextFactory.createCommandContext(command);
3.
4.   try {
5.     //执行前保存上下文
6.     Context.setCommandContext(context);
7.     Context.setProcessEngineConfiguration(processEngineConfiguration);
8.     return next.execute(command); //执行命令
9.
10.  } catch (Exception e) {
11.    context.exception(e);
12.
13.  } finally {
14.    try {
15.      //关闭上下文，内部会 flush session，把数据持久化到 db 等
16.      context.close();
17.    } finally {
```

```
18. //释放上下文

19.     Context.removeCommandContext();

20.     Context.removeProcessEngineConfiguration();

21. }

22. }

23.

24.     return null;

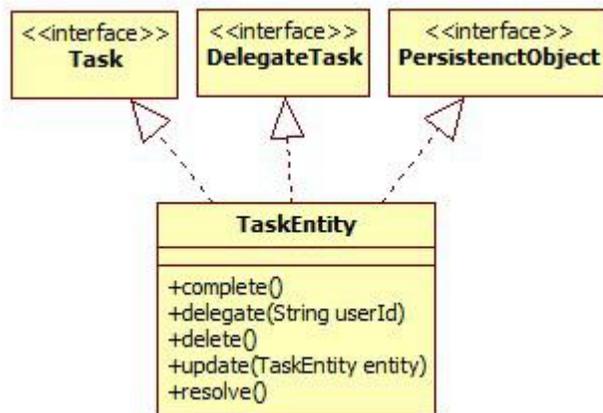
25. }
```

关于命令模式的细节说明，网上有很多资料，这里不展开。我只是想说一下我看到 **Activiti** 的这种设计之后的两点感受：

- 1) 一个产品或者一个项目，从技术上必须有一个明确的、唯一的开发模型或者叫开发样式（真不知道怎么说恰当），我们常说希望一个团队的所有人写出的代码都有统一的风格，都像是一个人写出来的，很理想化，但做到很难，往往我们都是通过“规范”去约束大家这样做，而规范毕竟是程序之外的东西，主观性很强，不遵守规范的情况屡屡发生。而如果架构师给出了明确的开发模型，并使用一些基础组件加以强化，把程序员要走的路规定清楚，那你想不遵守规范都会很难，因为那意味着你写的东西没发工作。就像 **Activiti** 做的这样，明确以 **Command** 作为基本开发模型，辅之以 **Event-Listener**，这样编程风格的整体性得到了保证。
- 2) 使用命令模式的好处，我这里体会最深的就是 职责分离，解耦。有了 **Command**，各个 **Service** 从角色上说只是一些协调者或者控制者，他不需要知道具体怎么做，他只是把任务交给了各个命令。直接的好处是臃肿的、万能的大类没有了。而这往往是我们平时开发中最深恶痛绝的地方。

4.核心业务对象 (**Task**, **ProcessInstance**, **Execution...**):**org.activiti.engine.impl.persistence.entity** 包下的类是 **Activiti** 的核心业务对象。它们是真正的对象，而不是只有数据没有行为的假对象，搞 **java** 企业级开发的人也许已经习惯了下面的层次划分：**controller->service->dao->entity**, **entity** 只是 **ORMapping** 中数据表的 **java** 对象体现，

没有任何行为（**getter/setter** 不能算行为），对于面向对象来说，这当然是有问题的，记得曾听人说过这样的话“使用面向对象语言进行设计和开发 与 面向对象的设计和开发 是两回事”，面向对象讲究的是“封装”，“多态”，追求的是满足“开-闭”原则的、职责单一的对象社会。如果你认同上述观点，那么相信 **Activiti** 会让你感觉舒服一些，以 **TaskEntity** 为例，其 UML 类图如下：



（图 2：TaskEntity 类）

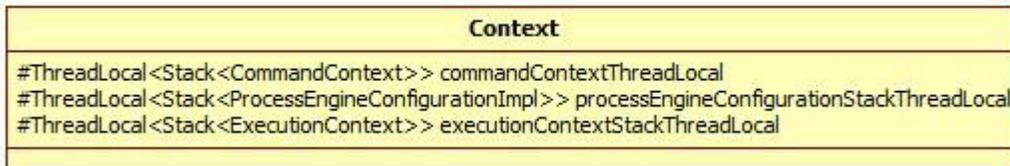
TaskEntity 实现了 3 个接口：**Task**，**DelegateTask** 和 **PersistentObject**。其中 **PersistentObject** 是一个声明接口，表明 **TaskEntity** 需要持久化。接口是一种角色的声明，是一份职责的描述而 **TaskEntity** 就是这个角色的具体扮演者，因此 **TaskEntity** 必须承担如 **complete,delegate** 等职责。

但是这里有些遗憾的是像 **complete** 这么重要的行为没有在 3 个接口中描述，因此“面向抽象”编程对于 **TaskEntity** 来说还没有完全做到。但至少 **Activiti** 告诉我们：

- 1) 牢记面向抽象编程,把职责拆分为不同的接口，让接口来体现对象的职责，而不用去关心这份职责具体由哪个对象实现；
- 2) **entity** 其实可以也应该是真正的对象。

5.Activiti 的上下文组件（Context）

上下文（**Context**）用来保存生命周期很长的、全局性的信息。**Activiti** 的 **Context** 类（在 **org.activiti.engine.impl.context** 包下）保持如下三类信息：



(图 3: Context 类)

CommandContext: 命令上下文，保持每个命令需要的必要资源，如持久化需要的 **session**。

ProcessEngineConfigurationImpl: 流程引擎相关的配置信息。它是整个引擎的全局配置信息，**mailServerHost**，**DataSource** 等。单例。该实例在流程引擎创建时被实例化，其调用 **stack** 如下图：

```

GenericManagerFactory(Class<? extends Session>) - org.activiti.engine.impl.persistence.GenericManagerFactory
├─ initSessionFactory() : void - org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl (20 matches)
│   └─ init() : void - org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl
│       └─ buildProcessEngine() : ProcessEngine - org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl
│           └─ buildProcessEngine(URL) : ProcessEngine - org.activiti.engine.ProcessEngines
│               └─ initProcessEngineFromResource(URL) : ProcessEngineInfo - org.activiti.engine.ProcessEngines
│                   └─ init() : void - org.activiti.engine.ProcessEngines
│                       └─ retry(String) : ProcessEngineInfo - org.activiti.engine.ProcessEngines

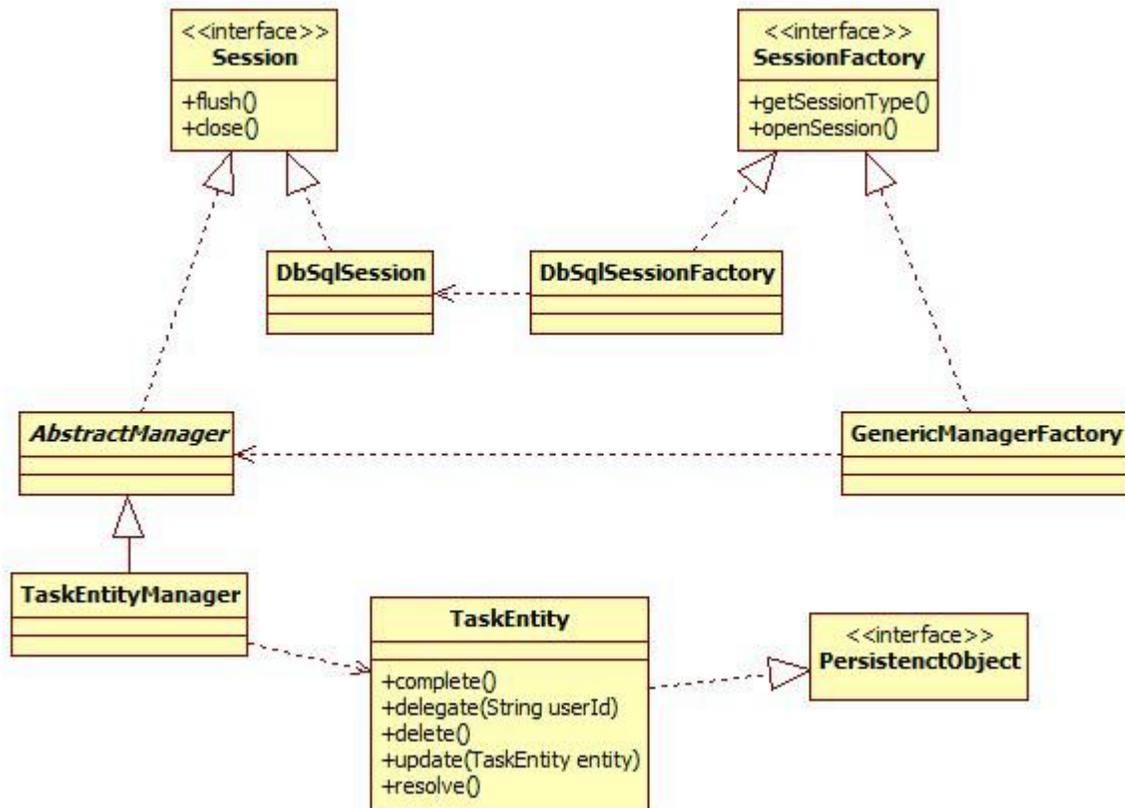
```

(图 4: ProcessEngineConfiguration 的初始化)

ExecutionContext: 刚看到这个类感觉有些奇怪，不明白其作用。看其代码持有 **ExecutionEntity** 这个实例。而 **ExecutionEntity** 是 **Activiti** 中非常重要的一个类，//TODO

6.Activiti 的持久化框架（组件）

Activiti 使用 **ibatis** 作为 **ORMapping** 工具。在此基础之上 **Activiti** 设计了自己的持久化框架，看一张图：



(图 5: Activiti 持久化框架)

顶层接口是 `Session` 和 `SessionFactory`，这都非常好理解了。

`Session` 有两个实现类：

`DbSqlSession`：简单点说，`DbSqlSession` 负责 `sql` 表达式的执行。

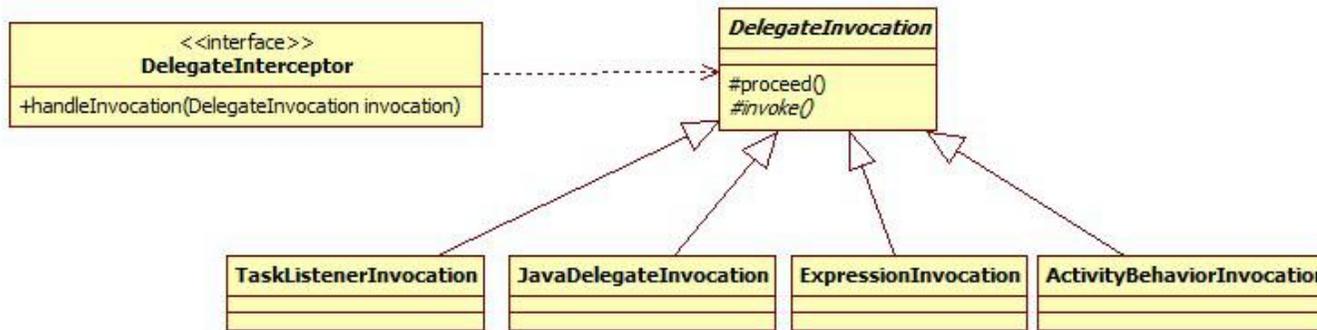
`AbstractManager`：简单点说，`AbstractManager` 及其子类负责面向对象的持久化操作

同理 `DbSqlSessionFactory` 与 `GenericManagerFactory` 的区别就很好理解了。

持久化框架也是在流程引擎建立时初始化的，具体见图 4.

7.Event-Listener 组件

Activiti 允许客户端代码介入流程的执行。为此提供了一个基础组件，看图：



(图 6: 用户代码介入流程的基础组件)

用户可以介入的代码类型包括：TaskListener，JavaDelegate，Expression，ExecutionListener。

ProcessEngineConfigurationImpl 持有 DelegateInterceptor 的某个实例，这样就可以随时非常方便地调用 handleInvocation

8.Cache 组件

对 Activiti 的 cache 实现很感兴趣，但现在我了解到的情况（也许还没有了解清楚）其 cache 的实现还是很简单，在 DbSqlSession 中有 cache 实现：

Java 代码 ☆

1. `protected List<PersistentObject> insertedObjects = new ArrayList<PersistentObject>();`
2. `protected Map<Class<?>, Map<String, CachedObject>> cachedObjects = new HashMap<Class<?>, Map<String, CachedObject>>();`

```
3. protected List<DeleteOperation> deletedObjects = new ArrayList<DeleteOperation>();  
4. protected List<DeserializedObject> deserializedObjects = new ArrayList<DeserializedObject>();
```

也就是说 **Activiti** 就是基于内存的 **List** 和 **Map** 来做缓存的。具体怎么用的呢？以 **DbSqlSession.selectOne** 方法为例：

Java 代码

```
1. public Object selectOne(String statement, Object parameter) {  
2.     statement = dbSqlSessionFactory.mapStatement(statement);  
3.     Object result = sqlSession.selectOne(statement, parameter);  
4.     if (result instanceof PersistentObject) {  
5.         PersistentObject loadedObject = (PersistentObject) result;  
6.         缓存处理  
7.         result = cacheFilter(loadedObject);  
8.     }  
9.     return result;  
10. }
```

Java 代码 ☆

```
1. protected PersistentObject cacheFilter(PersistentObject persistentObject) {  
2.     PersistentObject cachedPersistentObject = cacheGet(persistentObject.getClass(), persistentObject.getId());  
3.     if (cachedPersistentObject!=null) {  
4.         //如果缓存中有就直接返回  
5.         return cachedPersistentObject;  
6.     }  
7.     //否则，就先放入缓存  
8.     cachePut(persistentObject, true);  
9.     return persistentObject;  
10. }
```

Java 代码 ☆

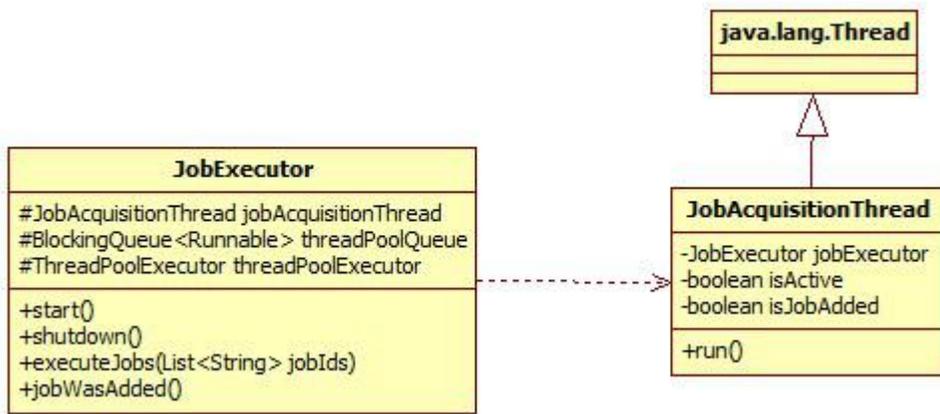
```
1. protected CachedObject cachePut(PersistentObject persistentObject, boolean storeState) {  
2.     Map<String, CachedObject> classCache = cachedObjects.get(persistentObject.getClass());
```

```
3.  if (classCache==null) {  
4.      classCache = new HashMap<String, CachedObject>();  
5.      cachedObjects.put(persistentObject.getClass(), classCache);  
6.  }  
7.  //这里是关键： 一个 CachedObject 包含被缓存的对象本身： persistentObject 和缓存的状态： storeState  
8.  //Activiti 正是根据 storeState 来判别缓存中的数据是否被更新是否与 db 保持一致的。  
9.  CachedObject cachedObject = new CachedObject(persistentObject, storeState);  
10. classCache.put(persistentObject.getId(), cachedObject);  
11. return cachedObject;  
12. }
```

看了 Activiti 的缓存设计，我现在最大的疑问是 Activiti 貌似不支持 cluster，因为其缓存设计是基于单机内存的，这个问题还需要进一步调查。

9.异步执行组件

Activiti 可以异步执行 job（具体例子可以看一下 `ProcessInstance startProcessInstanceByKey(String processDefinitionKey);`），下面简单分析一下其实现过程，还是先看图：



(图 7: 异步执行组件核心类)

JobExecutor 是异步执行组件的核心类，其包含三个主要属性：

- 1) JobAcquisitionThread jobAcquisitionThread: 执行任务的线程 extends java.lang.Thread
- 2) BlockingQueue<Runnable> threadPoolQueue
- 3) ThreadPoolExecutor threadPoolExecutor: 线程池

方法 ProcessEngines 在引擎启动时调用 JobExecutor.start，JobAcquisitionThread 线程即开始工作，其 run 方法不断循环执行 AcquiredJobs 中的 job，执行一次后线程等待一定时间直到超时或者 JobExecutor.jobWasAdded 方法因为有新任务而被调用。

这里发现有一处设计的不够好：JobAcquisitionThread 与 JobExecutor 之间的关系是如此紧密（你中有我，我中有你），那么可以把 JobAcquisitionThread 作为 JobExecutor 的内部类来实现，同时把 ThreadPoolExecutor threadPoolExecutor 交给 JobAcquisitionThread 来管理，JobExecutor 只负责接受任务以及启动、停止等更高级的工作，具体细节委托给 JobAcquisitionThread，责任分解，便于维护，JobExecutor 的代码也会看起来更清晰。

PVM:Process Virtual Machine,流程虚拟机 API 暴露了流程虚拟机的 POJO 核心，流程虚拟机 API 描述了一个工作流流程必备的组件，这些组件包括：

PvmProcessDefinition: 流程的定义，形象点说就是用户画的那个图。静态含义。

PvmProcessInstance: 流程实例，用户发起的某个 **PvmProcessDefinition** 的一个实例，动态含义。

PvmActivity: 流程中的一个节点

PvmTransition: 衔接各个节点之间的路径，形象点说就是图中各个节点之间的连接线。

PvmEvent: 流程执行过程中触发的事件

以上这些组件很好地对一个流程进行了建模和抽象。每个组件都有很清晰的角色和职责划分。另外，有了这些 API，我们可以通过编程的方式，用代码来“画”一个流程图并让他 run 起来，例如：

Java 代码 ☆

```
1. PvmProcessDefinition processDefinition = new ProcessDefinitionBuilder()
2.     .createActivity("a").initial().<strong style="background-color: #ff0000;">behavior</strong>(new WaitState())
3.     .transition("b").endActivity().createActivity("b")
4.     .behavior(new WaitState()).transition("c").endActivity()
5.     .createActivity("c").behavior(new WaitState()).endActivity()
6.     .buildProcessDefinition();
7. PvmProcessInstance processInstance = processDefinition
8.     .createProcessInstance();
9. processInstance.start();
```

```
10. PvmExecution activityInstance = processInstance.findExecution("a");

11. assertNotNull(activityInstance);

12. activityInstance.signal(null, null);

13. activityInstance = processInstance.findExecution("b");

14. assertNotNull(activityInstance);

15. activityInstance.signal(null, null);

16. activityInstance = processInstance.findExecution("c");

17. assertNotNull(activityInstance);
```

以上代码都很简单，很好理解，只有一点需要说明一下，粗体红色背景的 `behavior` 方法，为一个 `PvmActivity` 增加 `ActivityBehavior`，这是干什么呢？`ActivityBehavior` 是一个 `interface`，其接口声明很简单：

Java 代码 ☆

```
1. /**
2.  * @author Tom Baeyens
3.  */
4. public interface ActivityBehavior {
5.
```

```
6. void execute(ActivityExecution execution) throws Exception;  
7. }
```

我的理解：Activiti 把完成一个 PvmActivity 的行为单独建模封装在 ActivityBehavior 中。execute 方法只有一个参数 ActivityExecution，为啥这么设计？

为了更好地理解 ActivityBehavior 的作用，我们以 TaskEntity.complete 方法为例,分析其执行过程，先看 complete 的代码：

Java 代码 

```
1. public void complete() {  
2.     fireEvent(TaskListener.EVENTNAME_COMPLETE);  
3.  
4.     Context  
5.     .getCommandContext()  
6.     .getTaskManager()  
7.     .deleteTask(this, TaskEntity.DELETE_REASON_COMPLETED, false);  
8.  
9.     if (executionId!=null) {  
10.         getExecution().signal(null, null);  
}
```

```
11. }
```

```
12. }
```

代码很简单，也很好理解（可能出乎我们的意料，因为完成一个 **task**，其实有很多事情要做的）：

1.fireEvent: 通知 Listener，本任务完成了。

2.数据持久化相关的动作

3.getExecution().signal(null, null): 发信号，这里面隐藏的东西就多了，总体来说，完成了当前任务流程怎么走，怎么生成新的任务都是在这里完成的。

进去看看：

Java 代码 

```
1. public void signal(String signalName, Object signalData) {  
2.     ensureActivityInitialized();  
3.     SignallableActivityBehavior activityBehavior = (SignallableActivityBehavior) activity.getActivityBehavior();  
4.     try {  
5.         activityBehavior.signal(this, signalName, signalData);  
6.     } catch (RuntimeException e) {  
7.         throw e;  
8.     } catch (Exception e) {  
9.         throw new PvmException("couldn't process signal '"+signalName+"' on activity '"+activity.getId()+"': "+e.getMessage(), e);
```

```
10. }
```

```
11. }
```

`ExecutionEntity.signal` 方法核心工作就是把发信号的工作委托给 `PvmActivity` 的 `activityBehavior`。这里的设计存在问题，很显然其触犯了一个代码的坏味道：消息链。它让 `ExecutionEntity` 没有必要地与 `SignallableActivityBehavior` 产生了耦合，更好的做法应该是 `PvmActivity` 提供 `signal` 方法，其内部调用 `ActivityBehavior` 完成发信号工作。

其实看看 `PvmActivity` 的接口声明，我不免也有疑问，本来属于 `PvmActivity` 的很重要的职责在其接口声明中都没有体现，why??

Java 代码 

```
1. /**  
2.  * @author Tom Baeyens  
3.  */  
4. public interface PvmActivity extends PvmScope {  
5.  
6.  boolean isAsync();  
7.  
8.  PvmScope getParent();  
9.
```

```
10. List<PvmTransition> getIncomingTransitions();  
11.  
12. List<PvmTransition> getOutgoingTransitions();  
13.  
14. PvmTransition findOutgoingTransition(String transitionId);  
15. }
```

把思路拉回来，我们继续看 `activityBehavior.signal` 方法内部的具体实现。

//待续

Activiti 5.3 : 流程活动自动与手工触发执行

分类: [Workflow](#) 2011-03-23 12:51 2849 人阅读 [评论\(2\)](#) [收藏](#) [举报](#)

活动 [exceptionclassstringobjectdeployment](#)

Activiti 5.3 支持流程活动自动执行与手工触发执行。其中,自动执行是指,在启动流程之前,准备流程所需要的控制流程进度的变量数据,启动流程之后,无需外部干预,就能够按照预定义的流程执行;手工触发执行是指,执行到流程中某个个结点后流程暂时停止运行,直到收到外部发送的信号以后,才会继续向前推进,这样情况可以更加精细地控制流程。

下面主要通过基于 **Activiti 5.3** 的<parallelGateway>、<serviceTask>、<receiveTask>、<userTask>元素来看一下。首先,我们在测试的过程中,用到 JUnit 3.x,为了方便,这里给了一层封装,代码如下所示:

[\[java\]](#) [view plain](#) [copy](#)

```
1. package org.shirdrn.workflow.activiti;
2.
3. import junit.framework.TestCase;
4.
5. import org.activiti.engine.FormService;
6. import org.activiti.engine.HistoryService;
7. import org.activiti.engine.IdentityService;
8. import org.activiti.engine.ManagementService;
9. import org.activiti.engine.ProcessEngine;
10. import org.activiti.engine.ProcessEngines;
11. import org.activiti.engine.RepositoryService;
12. import org.activiti.engine.RuntimeService;
13. import org.activiti.engine.TaskService;
14.
15. /**
16.  * @author shirdrn
17.  */
18. public abstract class AbstractTest extends TestCase {
19.
20.     private ProcessEngine processEngine;
21.     protected String deploymentId;
```

```
22.     protected RepositoryService repositoryService;
23.     protected RuntimeService runtimeService;
24.     protected TaskService taskService;
25.     protected FormService formService;
26.     protected HistoryService historyService;
27.     protected IdentityService identityService;
28.     protected ManagementService managementService;
29.
30.     @Override
31.     protected void setUp() throws Exception {
32.         super.setUp();
33.         if(processEngine==null) {
34.             processEngine = ProcessEngines.getDefaultProcessEngine();
35.         }
36.         repositoryService = processEngine.getRepositoryService();
37.         runtimeService = processEngine.getRuntimeService();
38.         taskService = processEngine.getTaskService();
39.         formService = processEngine.getFormService();
40.         historyService = processEngine.getHistoryService();
41.         identityService = processEngine.getIdentityService();
42.         managementService = processEngine.getManagementService();
43.         initialize();
44.     }
45.
46.     @Override
47.     protected void tearDown() throws Exception {
48.         super.tearDown();
49.         destroy();
50.     }
51.
52.     protected abstract void initialize() throws Exception;
53.
54.     protected abstract void destroy() throws Exception;
```

55. }

这里面，主要是在测试之前做一些初始化工作，主要包括流程引擎实例的构建，及其流程提供的基本服务。下面测试会用到该抽象类。

自动执行

`<serviceTask>`元素，可以实现自动活动，语法如下所示：

[xhtml] view plaincopy

```
1. <serviceTask id="serviceTaskId" name="serviceTaskName"
2.   iviti:class="org.shindrn.workflow.activiti.gateway.ServiceTaskClass"/>
```

其中，`activiti:class`属性为该结点对应的处理类，该类要求实现 `org.activiti.engine.delegate.JavaDelegate` 接口，该接口定义如下所示：

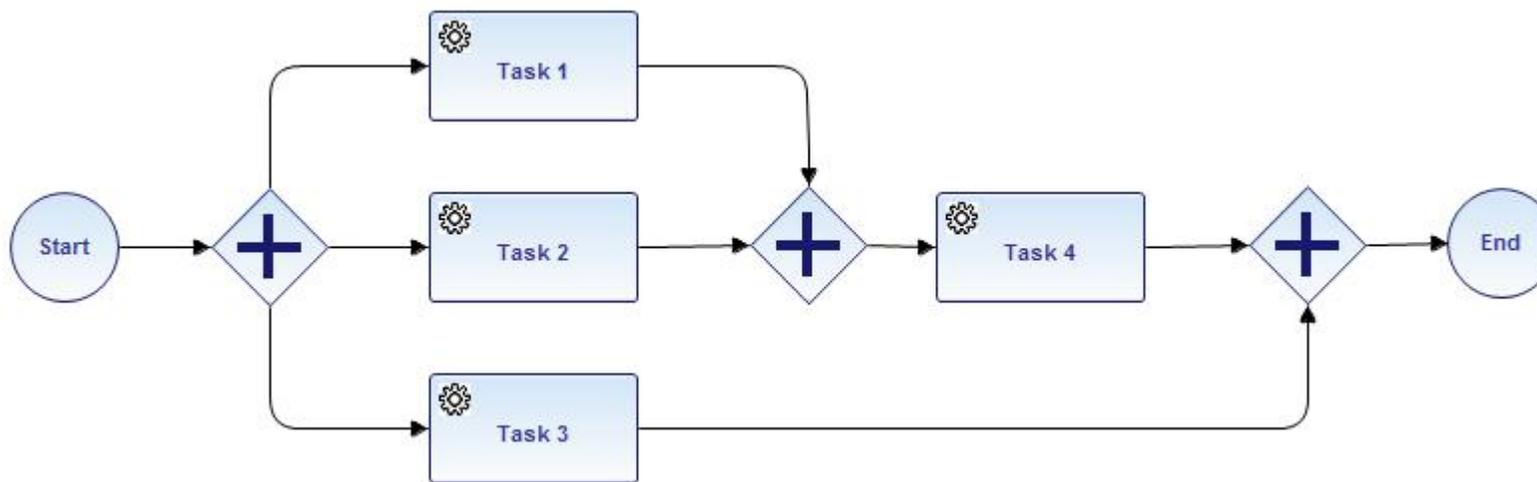
[java] view plaincopy

```
1. package org.activiti.engine.delegate;
2.
3. public interface JavaDelegate {
4.
5.     void execute(DelegateExecution execution) throws Exception;
6.
7. }
```

`execute` 方法的参数 `DelegateExecution execution` 可以在流程中各个结点之间传递流程变量。

下面给出一个具体的例子：

自动执行的流程，如图所示：



对应的流程定义文件为 GatewayTest.testAutomaticForkJoin.bpmn20.xml，如下所示：

[\[c-sharp\] view plaincopy](#)

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn" xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC" xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema" expressionLanguage="http://www.w3.org/1999/XPath" targetNamespace="http://www.activiti.org/test">
3.   <process id="AutomaticParalellBasedForkJoin" name="AutomaticParalellBasedForkJoin">
4.     <startEvent id="startevent7" name="Start"></startEvent>
5.     <parallelGateway id="parallelgateway12" name="Fork"></parallelGateway>
6.     <serviceTask id="servicetask3" name="Task 1" activiti:class="org.shirdrn.workflow.activiti.gateway.ServiceTask1"></serviceTask>
7.     <serviceTask id="servicetask4" name="Task 2" activiti:class="org.shirdrn.workflow.activiti.gateway.ServiceTask2"></serviceTask>
8.     <serviceTask id="servicetask5" name="Task 3" activiti:class="org.shirdrn.workflow.activiti.gateway.ServiceTask3"></serviceTask>
9.     <parallelGateway id="parallelgateway13" name="First Join"></parallelGateway>
10.    <serviceTask id="servicetask6" name="Task 4" activiti:class="org.shirdrn.workflow.activiti.gateway.ServiceTask4"></serviceTask>
11.    <parallelGateway id="parallelgateway14"></parallelGateway>
12.    <endEvent id="endevent7" name="End"></endEvent>
13.    <sequenceFlow id="flow45" name="" sourceRef="startevent7" targetRef="parallelgateway12"></sequenceFlow>
14.    <sequenceFlow id="flow46" name="" sourceRef="parallelgateway12" targetRef="servicetask3"></sequenceFlow>
  
```

```
15. <sequenceFlow id="flow47" name="" sourceRef="parallelgateway12" targetRef="servicetask4"></sequenceFlow>
16. <sequenceFlow id="flow48" name="" sourceRef="parallelgateway12" targetRef="servicetask5"></sequenceFlow>
17. <sequenceFlow id="flow49" name="" sourceRef="servicetask3" targetRef="parallelgateway13"></sequenceFlow>
18. <sequenceFlow id="flow50" name="" sourceRef="servicetask4" targetRef="parallelgateway13"></sequenceFlow>
19. <sequenceFlow id="flow51" name="" sourceRef="servicetask5" targetRef="parallelgateway14"></sequenceFlow>
20. <sequenceFlow id="flow52" name="" sourceRef="parallelgateway13" targetRef="servicetask6"></sequenceFlow>
21. <sequenceFlow id="flow53" name="" sourceRef="servicetask6" targetRef="parallelgateway14"></sequenceFlow>
22. <sequenceFlow id="flow54" name="" sourceRef="parallelgateway14" targetRef="endevent7"></sequenceFlow>
23. </process>
24. </definitions>
```

上述流程定义中，一共定义了 4 个 ServiceTask，模拟实现代码如下所示：

[java] view plaincopy

```
1. package org.shirdrn.workflow.activiti.gateway;
2.
3. import java.util.logging.Logger;
4.
5. import org.activiti.engine.delegate.DelegateExecution;
6. import org.activiti.engine.delegate.JavaDelegate;
7.
8. public class ServiceTask1 implements JavaDelegate {
9.
10.     private final Logger log = Logger.getLogger(ServiceTask1.class.getName());
11.
12.     @Override
13.     public void execute(DelegateExecution execution) throws Exception {
14.         Thread.sleep(10000);
15.         log.info("variables=" + execution.getVariables());
16.         execution.setVariable("task1", "I am task 1");
17.         log.info("I am task 1.");
18.     }
```

19. }

[java] view plaincopy

```
1. package org.shirdrn.workflow.activiti.gateway;
2.
3. import java.util.logging.Logger;
4.
5. public class ServiceTask2 implements JavaDelegate {
6.
7.     private final Logger log = Logger.getLogger(ServiceTask2.class.getName());
8.
9.     @Override
10.    public void execute(DelegateExecution execution) throws Exception {
11.        Thread.sleep(10000);
12.        log.info("variavles=" + execution.getVariables());
13.        execution.setVariable("task2", "I am task 2");
14.        log.info("I am task 2.");
15.    }
16. }
```

[java] view plaincopy

```
1. package org.shirdrn.workflow.activiti.gateway;
2.
3. import java.util.logging.Logger;
4.
5. public class ServiceTask3 implements JavaDelegate {
6.
7.     private final Logger log = Logger.getLogger(ServiceTask3.class.getName());
8.
9.     @Override
10.    public void execute(DelegateExecution execution) throws Exception {
11.        Thread.sleep(10000);
```

```
12.     log.info("variavles=" + execution.getVariables());
13.     execution.setVariable("task3", "I am task 3");
14.     log.info("I am task 3.");
15.     }
16. }
```

[java] view plaincopy

```
1.  package org.shirdrn.workflow.activiti.gateway;
2.
3.  import java.util.logging.Logger;
4.
5.  public class ServiceTask4 implements JavaDelegate {
6.
7.     private final Logger log = Logger.getLogger(ServiceTask4.class.getName());
8.
9.     @Override
10.    public void execute(DelegateExecution execution) throws Exception {
11.        Thread.sleep(10000);
12.        log.info("variavles=" + execution.getVariables());
13.        execution.setVariable("task4", "I am task 4");
14.        log.info("I am task 4.");
15.    }
16. }
```

测试代码，如下所示：

[java] view plaincopy

```
1.  package org.shirdrn.workflow.activiti.gateway;
2.
3.  import org.activiti.engine.runtime.ProcessInstance;
4.  import org.activiti.engine.test.Deployment;
5.  import org.shirdrn.workflow.activiti.AbstractTest;
```

```
6.
7. /**
8.  * @author shirdrn
9.  */
10. public class AutomaticParallelGatewayTest extends AbstractTest {
11.
12.     private String deploymentId;
13.
14.     @Override
15.     protected void initialize() throws Exception {
16.         deploymentId = repositoryService.createDeployment()
17.             .addClasspathResource("diagrams/GatewayTest.testAutomaticForkJoin.bpmn20.xml")
18.             .deploy().getId();
19.     }
20.
21.     @Override
22.     protected void destroy() throws Exception {
23.         repositoryService.deleteDeployment(deploymentId, true);
24.     }
25.
26.     @Deployment
27.     public void testForkJoin() {
28.         ProcessInstance pi = runtimeService.startProcessInstanceByKey("AutomaticParalellBasedForkJoin");
29.         assertEquals(true, pi.isEnded());
30.     }
31. }
```

只需要启动一个流程实例，它会自动执行到结束。这种情况下，你不需要关注流程的执行进度，而只需要把精力集中在每个结点的处理逻辑（通常是简单或者复杂的商业逻辑）上，运行结果如下所示：

[\[xhtml\]](#) [view plain](#) [copy](#)

```
1. 2011-3-23 11:50:12 org.shirdrn.workflow.activiti.gateway.ServiceTask1 execute
2. 信息: variables={}
```

```

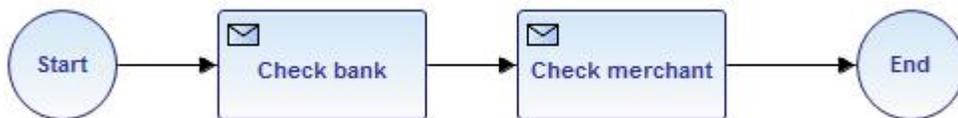
3. 2011-3-23 11:50:12 org.shirdrn.workflow.activiti.gateway.ServiceTask1 execute
4. 信息: I am task 1.
5. 2011-3-23 11:50:22 org.shirdrn.workflow.activiti.gateway.ServiceTask2 execute
6. 信息: variavles={task1=I am task 1}
7. 2011-3-23 11:50:22 org.shirdrn.workflow.activiti.gateway.ServiceTask2 execute
8. 信息: I am task 2.
9. 2011-3-23 11:50:32 org.shirdrn.workflow.activiti.gateway.ServiceTask4 execute
10. 信息: variavles={task1=I am task 1, task2=I am task 2}
11. 2011-3-23 11:50:32 org.shirdrn.workflow.activiti.gateway.ServiceTask4 execute
12. 信息: I am task 4.
13. 2011-3-23 11:50:42 org.shirdrn.workflow.activiti.gateway.ServiceTask3 execute
14. 信息: variavles={task1=I am task 1, task2=I am task 2, task4=I am task 4}
15. 2011-3-23 11:50:42 org.shirdrn.workflow.activiti.gateway.ServiceTask3 execute
16. 信息: I am task 3.
  
```

手工触发执行

通过<receiveTask>和<userTask>元素都可以实现流程的手工触发执行。

基于<receiveTask>

实现的流程，如图所示：



对应的流程定义文件 Task.ReceiveTask.bpmn20.xml，如下所示：

[\[xhtml\]](#) [view](#) [plaincopy](#)

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn" xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC" xmlns:omgdi="http://www.
  
```

```
g.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema" expressionLanguage="http://www.w3.org/1999/XPath" targetNamespace="http://
/www.activiti.org/test">
3. <process id="MyReceiveTask" name="MyReceiveTask">
4.   <startEvent id="startevent4" name="Start"></startEvent>
5.   <receiveTask id="receivetask1" name="Check bank">
6.     <extensionElements>
7.       <activiti:executionListener event="start" class="org.shirdrn.workflow.activiti.task.CheckBankReceiveTask"></activiti:executionListener>
8.     </extensionElements>
9.   </receiveTask>
10.  <receiveTask id="receivetask2" name="Check merchant">
11.    <extensionElements>
12.      <activiti:executionListener event="start" class="org.shirdrn.workflow.activiti.task.CheckMerchantReceiveTask"></activiti:executionListener>
13.    </extensionElements>
14.  </receiveTask>
15.  <endEvent id="endevent5" name="End"></endEvent>
16.  <sequenceFlow id="flow16" name="" sourceRef="startevent4" targetRef="receivetask1"></sequenceFlow>
17.  <sequenceFlow id="flow17" name="" sourceRef="receivetask1" targetRef="receivetask2"></sequenceFlow>
18.  <sequenceFlow id="flow18" name="" sourceRef="receivetask2" targetRef="endevent5"></sequenceFlow>
19. </process>
20.</definitions>
```

上述流程定义中，对应的两个处理类，代码分别如下所示：

[java] view plaincopy

```
1. package org.shirdrn.workflow.activiti.task;
2.
3. import java.util.HashMap;
4. import java.util.logging.Logger;
5.
6. import org.activiti.engine.delegate.DelegateExecution;
7. import org.activiti.engine.delegate.JavaDelegate;
8.
```

```
9. public class CheckBankReceiveTask implements JavaDelegate {
10.
11.     private final Logger log = Logger.getLogger(CheckBankReceiveTask.class.getName());
12.
13.     @SuppressWarnings("unchecked")
14.     @Override
15.     public void execute(DelegateExecution execution) throws Exception {
16.         log.info("i am CheckBankReceiveTask.");
17.         System.out.println("in : " + execution.getVariables());
18.         ((HashMap<String, Object>)execution.getVariables().get("in")).put("next", "CheckBankTask");
19.         ((HashMap<String, Object>)execution.getVariables().get("out")).put("reponse", "subprocess:CheckBankReceiveTask->CheckMerchantReceiveTask");
20.     }
21. }
```

[java] view plaincopy

```
1. package org.shirdrn.workflow.activiti.task;
2.
3. import java.util.HashMap;
4. import java.util.logging.Logger;
5.
6. import org.activiti.engine.delegate.DelegateExecution;
7. import org.activiti.engine.delegate.JavaDelegate;
8.
9. public class CheckMerchantReceiveTask implements JavaDelegate {
10.
11.     private final Logger log = Logger.getLogger(CheckMerchantReceiveTask.class.getName());
12.
13.     @SuppressWarnings("unchecked")
14.     @Override
15.     public void execute(DelegateExecution execution) throws Exception {
16.         log.info("i am CheckMerchantReceiveTask.");
```

```
17.         System.out.println("in : " + execution.getVariables());
18.         ((HashMap<String, Object>)execution.getVariables().get("in")).put("previous", "CheckMerchantReceiveTask");
19.     }
20. }
```

上面还用到一个 `org.shirdrn.workflow.activiti.subprocess.Merchant` 类，该类必须支持序列化，如下所示：

[\[java\] view plaincopy](#)

```
1. package org.shirdrn.workflow.activiti.subprocess;
2.
3. import java.io.Serializable;
4.
5. public class Merchant implements Serializable {
6.     private static final long serialVersionUID = 1L;
7.     public Merchant(String merchantId, int priority, short serviceType, short status) {
8.         super();
9.         this.merchantId = merchantId;
10.        this.priority = priority;
11.        this.serviceType = serviceType;
12.        this.status = status;
13.    }
14.    public Merchant(String merchantId) {
15.        this(merchantId, -1, (short)0, (short)0);
16.    }
17.    private String merchantId;
18.    private int priority = -1;
19.    private short serviceType = 0;
20.    private short status = 0;
21.    public String getMerchantId() {
22.        return merchantId;
23.    }
24.    public void setMerchantId(String merchantId) {
```

```
25.     this.merchantId = merchantId;
26. }
27. public int getPriority() {
28.     return priority;
29. }
30. public void setPriority(int priority) {
31.     this.priority = priority;
32. }
33. public short getServiceType() {
34.     return serviceType;
35. }
36. public void setServiceType(short serviceType) {
37.     this.serviceType = serviceType;
38. }
39. public short getStatus() {
40.     return status;
41. }
42. public void setStatus(short status) {
43.     this.status = status;
44. }
45. @Override
46. public String toString() {
47.     return "Merchant[" + merchantId + "];
48. }
49. }
```

测试用例，代码如下所示：

[java] view plaincopy

```
1. package org.shirdrn.workflow.activiti.task;
2.
3. import java.util.HashMap;
```

```
4. import java.util.List;
5. import java.util.Map;
6.
7. import org.activiti.engine.repository.Deployment;
8. import org.activiti.engine.runtime.Execution;
9. import org.activiti.engine.runtime.ProcessInstance;
10. import org.shirdrn.workflow.activiti.AbstractTest;
11. import org.shirdrn.workflow.activiti.subprocess.Merchant;
12.
13. /**
14.  * @author shirdrn
15.  */
16. public class MyReceiveTaskTest extends AbstractTest {
17.
18.     @Override
19.     protected void initialize() throws Exception {
20.         Deployment deployment = repositoryService
21.             .createDeployment()
22.             .addClasspathResource(
23.                 "diagrams/Task.ReceiveTask.bpmn20.xml")
24.             .deploy();
25.         deploymentId = deployment.getId();
26.     }
27.
28.     @Override
29.     protected void destroy() throws Exception {
30.         repositoryService.deleteDeployment(deploymentId, true);
31.     }
32.
33.     public void testSubProcess() {
34.         // prepare data packet
35.         Map<String, Object> variables = new HashMap<String, Object>();
36.         Map<String, Object> subVariables = new HashMap<String, Object>();
```

```
37.     variables.put("maxTransCount", 1000000);
38.     variables.put("merchant", new Merchant("ICBC"));
39.     variables.put("protocol", "UM32");
40.     variables.put("repository", "10.10.38.99:/home/shirdrn/repository");
41.     variables.put("in", subVariables);
42.     variables.put("out", new HashMap<String, Object>());
43.
44.     // start process instance
45.     ProcessInstance pi = runtimeService.startProcessInstanceByKey("MyReceiveTask", variables);
46.     List<Execution> executions = runtimeService.createExecutionQuery().list();
47.     assertEquals(1, executions.size());
48.
49.     Execution execution = runtimeService.createExecutionQuery().singleResult();
50.     runtimeService.setVariable(execution.getId(), "type", "receiveTask");
51.     runtimeService.signal(execution.getId());
52.     assertEquals(1, executions.size());
53.
54.     execution = runtimeService.createExecutionQuery().list().get(0);
55.     assertNotNull(execution);
56.     runtimeService.setVariable(execution.getId(), "oper", "shirdrn");
57.     runtimeService.signal(execution.getId());
58. }
59.
60. }
```

运行结果如下所示:

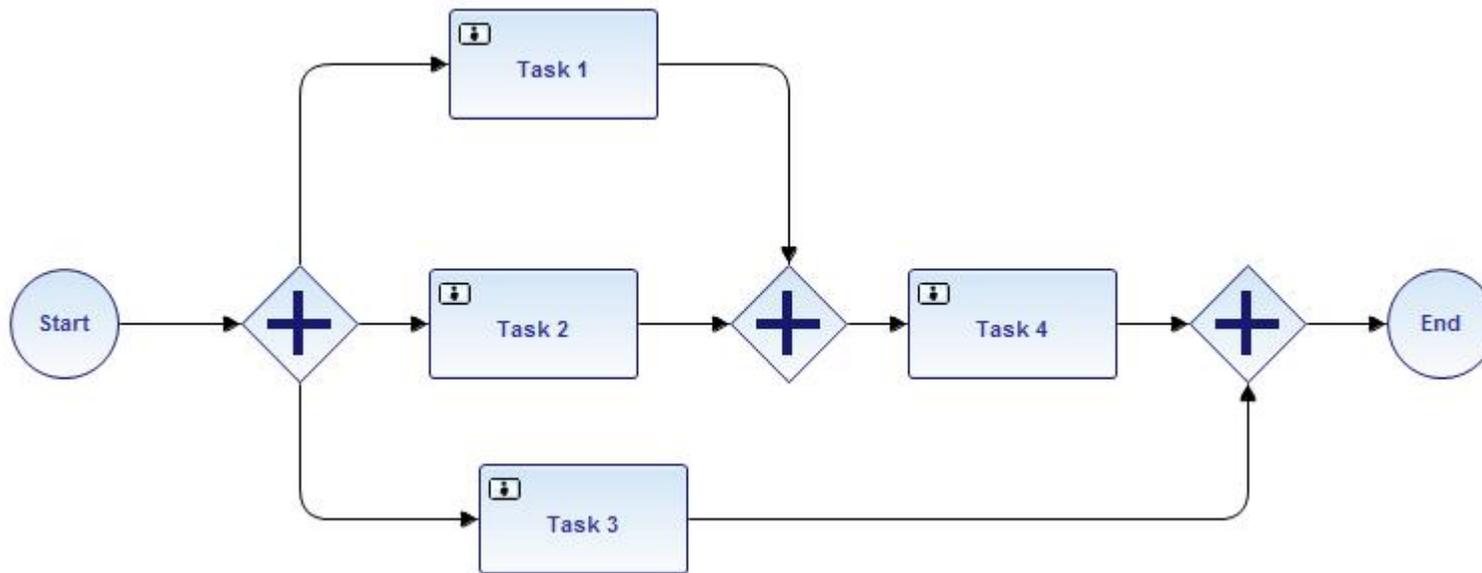
[\[xhtml\]](#) view plaincopy

1. 2011-3-23 12:51:35 org.shirdrn.workflow.activiti.task.CheckBankReceiveTask execute
2. 信息: i am CheckBankReceiveTask.
3. in : {protocol=UM32, repository=10.10.38.99:/home/shirdrn/repository, merchant=Merchant[ICBC], maxTransCount=1000000, in={}, out={}}
4. 2011-3-23 12:51:35 org.shirdrn.workflow.activiti.task.CheckMerchantReceiveTask execute
5. 信息: i am CheckMerchantReceiveTask.

```
6. in : {protocol=UM32, repository=10.10.38.99:/home/shirdrn/repository, merchant=Merchant[ICBC], maxTransCount=1000000, type=receiveTask, in={}, out={}
```

基于<userTask>

实现的流程，如图所示：



对应的流程定义文件，如下所示：

[\[xhtml\]](#) [view plaincopy](#)

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <definitions id="definitions"
3.     xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:activiti="http://activiti.org/bpmn"
4.     targetNamespace="Umpay">
5.     <process id="ParalellBasedForkJoin">
6.         <startEvent id="theStart" />
7.         <sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork" />
8.         <parallelGateway id="fork" name="Fork" />
9.         <sequenceFlow sourceRef="fork" targetRef="task1" />
10.        <sequenceFlow sourceRef="fork" targetRef="task2" />

```

```
11. <sequenceFlow sourceRef="fork" targetRef="task3" />
12. <userTask id="task1" name="Task 1">
13.     <extensionElements>
14.         <activiti:taskListener event="complete"
15.             class="org.shirdrn.workflow.activiti.gateway.Task1Listener" />
16.     </extensionElements>
17. </userTask>
18. <sequenceFlow sourceRef="task1" targetRef="firstJoin" />
19. <userTask id="task2" name="Task 2">
20.     <extensionElements>
21.         <activiti:taskListener event="complete"
22.             class="org.shirdrn.workflow.activiti.gateway.Task2Listener" />
23.     </extensionElements>
24. </userTask>
25. <sequenceFlow sourceRef="task2" targetRef="firstJoin" />
26. <userTask id="task3" name="Task 3">
27.     <extensionElements>
28.         <activiti:taskListener event="complete"
29.             class="org.shirdrn.workflow.activiti.gateway.Task3Listener" />
30.     </extensionElements>
31. </userTask>
32. <sequenceFlow sourceRef="task3" targetRef="secondJoin" />
33. <parallelGateway id="firstJoin" name="First Join" />
34. <sequenceFlow sourceRef="firstJoin" targetRef="task4" />
35. <userTask id="task4" name="Task 4">
36.     <extensionElements>
37.         <activiti:taskListener event="complete"
38.             class="org.shirdrn.workflow.activiti.gateway.Task4Listener" />
39.     </extensionElements>
40. </userTask>
41. <sequenceFlow sourceRef="task4" targetRef="secondJoin" />
42. <parallelGateway id="secondJoin" />
43. <sequenceFlow sourceRef="secondJoin" targetRef="theEnd" />
```

```
44.     <endEvent id="theEnd" />
45. </process>
46. </definitions>
```

我们看一下上述定义中，如下配置片段：

[\[xhtml\]](#) view plaincopy

```
1. <userTask id="task1" name="Task 1">
2.     <extensionElements>
3.         <activiti:taskListener event="complete"
4.             class="org.shirdrn.workflow.activiti.gateway.Task1Listener" />
5.     </extensionElements>
6. </userTask>
```

`<activiti:taskListener>`元素的 `event` 属性，它一共包含三种事件：`"create"`、`"assignment"`、`"complete"`，分别表示结点执行处理逻辑的时机为：在处理类实例化时、在结点处理逻辑被指派时、在结点处理逻辑执行完成时，可以根据自己的需要进行指定。

上述流程定义中，4 个任务结点对应的处理类，代码分别如下所示：

[\[java\]](#) view plaincopy

```
1. package org.shirdrn.workflow.activiti.gateway;
2.
3. import java.util.logging.Logger;
4.
5. import org.activiti.engine.delegate.DelegateTask;
6. import org.activiti.engine.impl.pvm.delegate.TaskListener;
7.
8. public class Task1Listener implements TaskListener {
9.
10.     private final Logger log = Logger.getLogger(Task1Listener.class.getName());
11.
12.     @Override
```

```
13.     public void notify(DelegateTask delegateTask) {
14.         try {
15.             Thread.sleep(10000);
16.         } catch (InterruptedException e) {
17.             e.printStackTrace();
18.         }
19.         log.info("I am task 1.");
20.     }
21. }
```

[java] view plaincopy

```
1.  package org.shirdrn.workflow.activiti.gateway;
2.
3.  import java.util.logging.Logger;
4.
5.  public class Task2Listener implements TaskListener {
6.
7.      private final Logger log = Logger.getLogger(Task2Listener.class.getName());
8.
9.      @Override
10.     public void notify(DelegateTask delegateTask) {
11.         try {
12.             Thread.sleep(10000);
13.         } catch (InterruptedException e) {
14.             e.printStackTrace();
15.         }
16.         log.info("I am task 2.");
17.     }
18. }
```

[java] view plaincopy

```
1.  package org.shirdrn.workflow.activiti.gateway;
```

```
2.
3. import java.util.logging.Logger;
4.
5. public class Task3Listener implements TaskListener {
6.
7.     private final Logger log = Logger.getLogger(Task3Listener.class.getName());
8.
9.     @Override
10.    public void notify(DelegateTask delegateTask) {
11.        try {
12.            Thread.sleep(5000);
13.        } catch (InterruptedException e) {
14.            e.printStackTrace();
15.        }
16.        log.info("I am task 3.");
17.    }
18. }
```

[java] view plaincopy

```
1. package org.shirdrn.workflow.activiti.gateway;
2.
3. import java.util.logging.Logger;
4.
5. public class Task4Listener implements TaskListener {
6.
7.     private final Logger log = Logger.getLogger(Task4Listener.class.getName());
8.
9.     @Override
10.    public void notify(DelegateTask delegateTask) {
11.        try {
12.            Thread.sleep(5000);
13.        } catch (InterruptedException e) {
```

```
14.         e.printStackTrace();
15.     }
16.     log.info("I am task 4.");
17. }
18. }
```

测试用例，代码如下所示：

[\[java\]](#) [view plaincopy](#)

```
1. package org.shirdrn.workflow.activiti.gateway;
2.
3. import java.util.Date;
4. import java.util.List;
5.
6. import org.activiti.engine.runtime.ProcessInstance;
7. import org.activiti.engine.task.Task;
8. import org.activiti.engine.task.TaskQuery;
9. import org.activiti.engine.test.Deployment;
10. import org.shirdrn.workflow.activiti.AbstractTest;
11.
12. /**
13.  * @author shirdrn
14.  */
15. public class ParallelGatewayTest extends AbstractTest {
16.
17.     private String deploymentId;
18.     private Date start = null;
19.     private Date end = null;
20.
21.     @Override
22.     protected void initialize() throws Exception {
23.         deploymentId = repositoryService.createDeployment()
24.             .addClasspathResource("diagrams/GatewayTest.testForkJoin.bpmn20.xml")
```

```
25.         .deploy().getId();
26.     }
27.
28.     @Override
29.     protected void destroy() throws Exception {
30.         repositoryService.deleteDeployment(deploymentId, true);
31.     }
32.
33.     @Deployment
34.     public void testUnbalancedForkJoin() {
35.         ProcessInstance pi = runtimeService.startProcessInstanceByKey("ParalellBasedForkJoin");
36.         TaskQuery query = taskService.createTaskQuery().processInstanceId(pi.getId()).orderByTaskName().asc();
37.
38.         List<Task> tasks = query.list();
39.         assertEquals(3, tasks.size());
40.         start = new Date();
41.         for(Task task : tasks) {
42.             taskService.complete(task.getId());
43.             end = new Date();
44.             System.out.println("" + (end.getTime()-start.getTime()) + "ms.");
45.         }
46.
47.         tasks = query.list();
48.         assertEquals(1, tasks.size());
49.         for(Task task : tasks) {
50.             taskService.complete(task.getId());
51.             end = new Date();
52.             System.out.println("" + (end.getTime()-start.getTime()) + "ms.");
53.         }
54.         end = new Date();
55.         System.out.println("" + (end.getTime()-start.getTime()) + "ms.");
56.     }
57. }
```

运行结果如下所示:

[xhtml] view plaincopy

1. 2011-3-23 12:50:09 org.shirdrn.workflow.activiti.gateway.Task1Listener notify
2. 信息: I am task 1.
3. 10031ms.
4. 2011-3-23 12:50:19 org.shirdrn.workflow.activiti.gateway.Task2Listener notify
5. 信息: I am task 2.
6. 20078ms.
7. 2011-3-23 12:50:24 org.shirdrn.workflow.activiti.gateway.Task3Listener notify
8. 信息: I am task 3.
9. 25093ms.
10. 2011-3-23 12:50:29 org.shirdrn.workflow.activiti.gateway.Task4Listener notify
11. 信息: I am task 4.
12. 30172ms.
13. 30172ms.