



[1. 配置文件 \(IOC\)](#)

[2. PVM 架构](#)

[2.1. Environment](#)

[2.2. Service](#)

[2.3. Command and Command Service](#)

[2.4. PVM Models: ProcessDefinition, ActivityBehaviour, Transition, Event](#)

[2.5. API 整体架构图](#)

[3. PVM Model\(模型\)解析](#)

[3.1. ActivityBehaviour\(内部节点\)](#)

[3.2. ExternalActivityBehaviour\(外部节点\)](#)

[3.3. 基本流程的实现](#)

[3.4. 事件机制 \(Event and EventListener\)](#)

[Bibliography](#)

Abstract

下面主要是介绍了本工作流组件的基础架构以及应用.

1. 配置文件 (IOC)

Abstract

配置文件是很关键的一步,同时 Configuration API 可以看做是工作流组件的一个入口 API,其他很多重要的 Service,都是通过这个 API 生成的.

在默认情况下,包里有 `jbpm.cfg.xml`,这个就是他的配置文件,以下是它的内容:

```
<jbpm-configuration>
    <import resource="jbpm.default.cfg.xml" />
    <import resource="jbpm.tx.jta.cfg.xml" />
    <import resource="jbpm.jpdl.cfg.xml" />
    <import resource="jbpm.identity.cfg.xml" />
    <import resource="jbpm.jobexecutor.cfg.xml" />
</jbpm-configuration>
```

现在再继续看下 `jbpm.default.cfg.xml` 配置文件:

```
<process-engine-context>
    <repository-service />
    <repository-cache />
    <execution-service />
    <history-service />
    <management-service />
    <identity-service />
    <task-service />
    <hibernate-configuration>
        <cfg resource="jbpm.hibernate.cfg.xml" />
    </hibernate-configuration>
</process-engine-context>
```



```
</hibernate-configuration>

.....

</process-engine-context>

<transaction-context>
  <repository-session />
  <db-session />
  <message-session />
  <timer-session />
  <history-session />
  <mail-session>
    <mail-server>
      <session-properties resource="jbpm.mail.properties" />
    </mail-server>
  </mail-session>
</transaction-context>
```

这个配置文件主要包含了"process-engine-context"和 'transaction-context'的配置.

现在可以来看下本 workflow 组件的 IOC 实现机制.

首先是 Context 接口,可以从这里存储,获得对象.

```
Object get(String key);
<T> T get(Class<T> type);
Object set(String key, Object value);
```

可以从 Context 中获取到组件,对于 IOC 容器来说,一般情况下都会提供一种加载的方式,比如从 xml 文件进行加载、从资源文件进行加载。



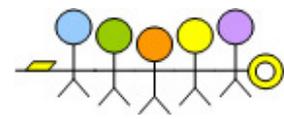
该组件是通过 `WireParser` 来解析 `xml`,然后创建并把对象存放在 `WireContext`. `WireContext` 这个类负责存放,提取对象,内部用一个 `Map` 来存储已经创建的对象实例,可以简单得把他看成是 `IOC` 的一个实现类. 从 `WireContext` 的 `javadoc`,我们可以看出,他主要是跟 `WireDefinition`, `Descriptor` 打交道. `WireContext` 里面包含了一个 `WireDefinition`,而 `WireDefinition` 里面包含了一系列的 `Descriptor`.每个 `Descriptor` 负责创建和初始化该对象. 比如我们可以看到 `IntegerDescriptor`, `FloatDescriptor`, `ObjectDescriptor` 等等. 我们来看下 `Descriptor` 的接口:

```
/**
 * constructs the object.
 * @param wireContext {@link WireContext} in which the object is created. This is also the
 {@link WireContext}
 * where the object will search for other object that may be needed during the initialization
 phase.
 * @return the constructed object.
 */
Object construct(WireContext wireContext);

/**
 *called by the WireContext to initialize the specified object.
 */
void initialize(Object object, WireContext wireContext);
```

`Descriptor` 对象的创建可以直接通过 `Java` 对象的实例化,比如(`new IntegerDescriptor(..)`),也可以通过 `xml` 的配置文件来实现.可以说我们更经常用 `xml` 来配置,所以就有了 `Binding` 的概念. `Binding` 类最主要的任务就是把 `XML DOM` 到 `Java` 对象的转换. `Bindings` 是把 `Binding` 归类了一下而已. 以下是 `Binding` 的接口.

```
public interface Binding {
    String getCategory();
}
```

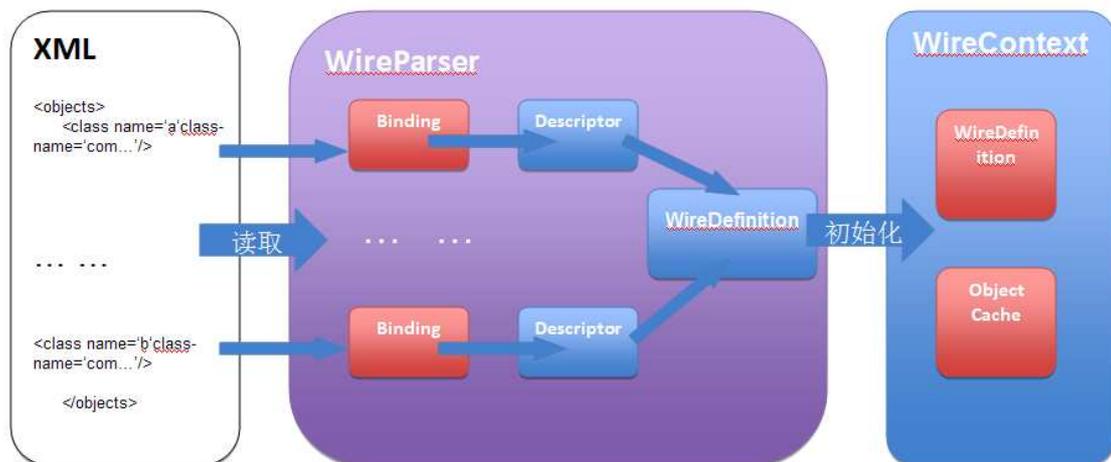


```
/** does this binding apply to the given element? */  
  
boolean matches(Element element);  
  
/** translates the given element into a domain model java object.  
 * Use the parse to report problems.  
 */  
  
Object parse(Element element, Parse parse, Parser parser);  
  
}
```

如果想看实现,我们可以看下 IdentityServiceBinding, RepositoryServiceBinding 等等.这里注意,在该组件的实现中,WireDescriptorBinding 是根据 tagName 来解析的. 所以,从它的 xml 配置文件,到 ProcessEngine 对象的构建,是这样的一个流程.

jbpm.cfg.xml -> jBPMConfigurationParser -> Binding -> Descriptor --> WireContext

或者更清楚的,我们可以看下下面这张图^[1].



我们不妨也看下 ConfigurationTest 测试.

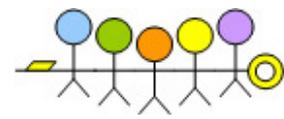




```
public void testConfigurationServices() {  
  
    ProcessEngine processEngine = new Configuration()  
  
        .setXmlString(  
  
            "<jbpm-configuration>" +  
  
            " <process-engine-context>" +  
  
            " <repository-service />" +  
  
            " <execution-service />" +  
  
            " <management-service />" +  
  
            " </process-engine-context>" +  
  
            "</jbpm-configuration>"  
  
        )  
  
        .buildProcessEngine();  
  
    assertNotNull(processEngine);  
  
    assertNotNull(processEngine.getExecutionService());  
  
    assertNotNull(processEngine.getManagementService());  
  
}
```

Configuration 类是入口,你可以从 *Configuration* 类中创建 *ProcessEngine*,而从 *ProcessEngine* 中获取到 *RepositoryService*, *ExecutionService*, *TaskService* 等等. *Configuration* 类里有两个实现类,一个是 *JbpmConfiguration*,这是默认组件自带的 *Configuration* 解析器,另外一个为 *SpringConfiguration*,这个是本工作流组件和 *Spring* 的集成.在这里,我们就只看下 *JbpmConfiguration* 的实现,在 *JbpmConfiguration* 类里,我们可以看到他是这么去调用 *Parser* 来解析 xml 的.

```
protected void parse(StreamInput streamSource) {  
  
    isConfigured = true;  
  
    JbpmConfigurationParser.getInstance()  
  
        .createParse()
```



```
.pushObject(this)

.setStreamSource(streamSource)

.execute()

.checkErrors("jbpm configuration " + streamSource);

}
```

在这里,我们可以看到,该 workflow 组件的配置文件是有两个元素组成的,一个是 `process-engine-context`, 另外一个则是 `transaction-context`. 其中 `process-engine-context` 里面的元素是包括了对外发布的服务, 比如 `repository-service`, `execution-service` 等等. 而 `transaction-context` 则包括了他的内部真正实现, 比如 `repository-service` 对应 `repository-session`. 也可以简单的把 `repository-service` 看做是 API, `repository-session` 看做是 SPI. 这样做的好处是, SPI 的实现, 对 API 一点影响都没有, 很大程度上提供了一个容易集成的特性.

2.PVM 架构

这里,我们看下 PVM 概念和架构,这也是整个项目的核心所在.

PVM (Process Virtual Machine), 主要是想作为一个开发平台,在这个平台上,可以很方便的开发 workflow, 服务编制 (orchestration), BPM 等等. 比如说 说 jPDL 这套语法的内部实现就是基于 PVM 的. 将来基于 PVM 可以开发一个符合 WS-BPEL 2.0 的模块. PVM 可以简单的看成是一个状态机. 我们接下去看下里面的几个重要概念.

2.1. Environment

Environment 的概念,主要有以下的作用.

1. 使 `process` 能在不同的环境下之行, 比如可以在标准的 Java, 企业级 Java, Seam 或者 Spring 环境下运行.
2. 他可以存放着不同的 `context` 对象, 比如我们之前所看到的配置文件中的 `process-engine` 的 `Context`, `transaction` 的 `Context` 等.

在代码中, 我们想获得其他的组件时, 或者变量时, 我们总是使用 `Environment` 的 API, 我们可以看下它的 API 的几个重要方法.



```
public abstract <T> T get(Class<T> type);

/** searches an object based on type. The search doesn't take superclasses of the
context elements

* into account.

* @return the first object of the given type or null in case no such element was
found.

*/

public abstract <T> T get(Class<T> type, String[] searchOrder);
```

Environment 内部是使用 *Map* 来保存不同的 *Context* 对象，*Environment* 对象的创建主要是有 *EnvironmentFactory* 来负责。*Environment* 是被保存在 *ThreadLocal* 中，如果有多个 *Environment*，那么是选择把 *Environment(s)* 放在一个堆栈中，然后再存放在 *ThreadLocal* 里。

2.2. Service

Service 就是 *PVM* 对外发布的服务，包括 *RepositoryService* (比如部署 *jpdl* 文件)，*ExecutionService* (负责管理执行)，*TaskService*，*ManagementService* 等等。可以说这是给用户调用的 *API*。比如我们看下他们的一些方法。

RepositoryService 接口的一些方法：

```
public interface RepositoryService {

    NewDeployment createDeployment();

    ProcessDefinitionQuery createProcessDefinitionQuery();

    ...

}
```

ExecutionService 的一些方法：

```
public interface ExecutionService {

    ProcessInstance startProcessInstanceById(String processDefinitionId);
```



```
ProcessInstance signalExecutionById(String executionId);  
  
...  
}
```

ManagementService 的接口:

```
public interface ManagementService {  
  
    void executeJob(long jobDbid);  
  
    JobQuery createJobQuery();  
  
}
```

2.3. Command and Command Service

Command 概念的引入,主要是想对所有的操作做一个封装.可以说上面每个 *Service* 的方法的实现都是通过实现一个 *Command* 来操作,然后通过 *CommandService* 调用到后面具体的实现.我们先看下 *Command* 的接口.

```
public interface Command<T> extends Serializable {  
  
    T execute(Environment environment) throws Exception;  
  
}
```

很简单,很典型的 *Command* 模式.

我们接下来看 *CommandService* 接口,顾名思义,他主要是负责来执行 *Command(s)*的操作.所以其他 *Service* 的实现都是通过 *CommandService* 来调用到后面的实现,可以把 *CommandService* 看做一个桥梁的作用.看一下 *CommandService* 的接口.

```
public interface CommandService {  
  
    /**  
  
    * @throws JbpmException if command throws an exception.  
  
    */  
  
}
```

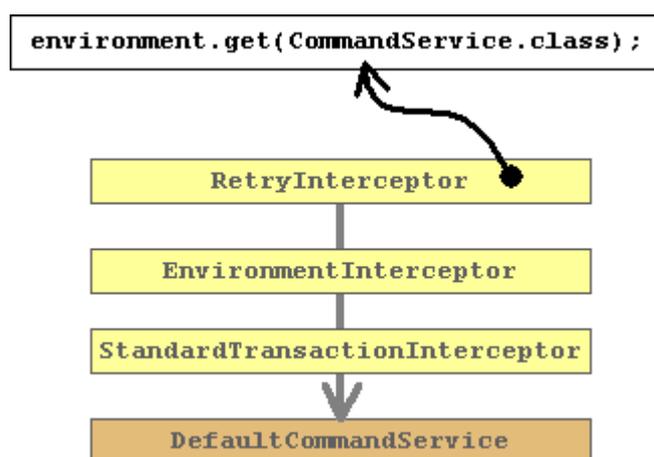


```
*/  
<T> T execute(Command<T> command);  
}
```

`CommandService` 还有一个特性,就是可以配置 `Interceptor`,比如事务就是在这一 `Service` 中来配置.看下 `CommandService` 的配置.

```
<command-service>  
  <retry-interceptor />  
  <environment-interceptor />  
  <standard-transaction-interceptor />  
</command-service>
```

这里,在执行真正的 `CommandServiceImpl` 之前,会先之前 `retry-Interceptor`,`environment-interceptor` 等等.这里的 `Interceptor` 的顺序是跟配置的顺序一致的.比如这样的配置,那就是 `retry-interceptor` 在 `environment-interceptor` 之前执行.我们看下面这个图.



可以说 `CommandService` 是最适合处理横截面(`cross-cutting`)的问题,比如事务,安全等的. `CommandService` 默认下是 `DefaultCommandService`,如果是在使用 `ejb` 的情况下,它的实现类就是 `EjbLocalCommandService` 和 `EjbRemoteCommandService` 类. 所以,这里是远程调用,还是本地调用,对用户而言,都是透明的.



1. `retry-interceptor` 的主要作用当碰到 `StaleObjectException`(也就是 `optimistic locking` 失败)时,过一定的时间后再尝试.
2. `environment-interceptor` 的主要作用就是,保证 `command` 是在 `Environment` 范围内执行,也就是在执行 `command` 之前,`openEnvironment`,执行之后,`closeEnvironment`.
3. `standard-transaction-interceptor` 的主要作用就是处理事务.
4. `authorization-interceptor`,主要是处理权限校验.

2.4. PVM Models: ProcessDefinition, ActivityBehaviour, Transition, Event

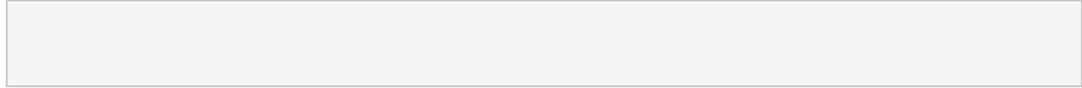
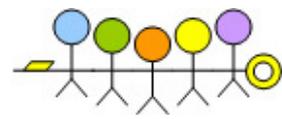
`ProcessDefinition` 是一个定义好的工作流程. `OpenProcessDefinition` 里面包含了启始的 `Activity`. 流程的走向是通过 `Activity` 的流向来组成的. `Transition` 就是用来连接 `Activity` 而后来构成一个流程的. 一个工作流的工作引擎,最基本的两个功能:一个是设计好当前的工作流程,第二个是有个东西需要来体现当前走到流程的哪一步,那么 PVM 中的 `Execution API` 就是这个作用. 至于最后一个 `Event`,就是你可以定义一些事件,比如当流程进入到某一个 `Activity` 的时候,促发 `email`. `Event` 和 `Activity` 最大的区别在于 `Event` 本身不会构成对流程走向的改变.

我们先看下 `ActivityBehaviour` 的接口.

```
public interface ActivityBehaviour extends Serializable {  
  
    void execute(ActivityExecution execution) throws Exception;  
  
}
```

就是到这个 `Activity`,需要执行的操作都在 `execute` 方法里. 还有一种 `ActivityBehaviour`,就是属于 `wait state`,也就是会停留在这个节点上,需要外部的一些触发,才会继续执行下去,这种情况,需要实现的接口就是 `ExternalActivityBehaviour`, 接口如下.

```
public interface ExternalActivityBehaviour extends ActivityBehaviour {  
  
    //handles an external trigger.  
  
    void signal(ActivityExecution execution, String signalName, Map<String, ?> parameters) throws  
    Exception;  
  
}
```



Wait State (也就是实现 ExternalActivityBehaviour)的促发,是通过 Transition 来完成的,来看下 Transition 这个接口.

```
public interface Transition extends ObservableElement {  
  
    /** the activity from which this transition leaves. */  
    Activity getSource();  
  
    /** the activity in which this transition arrives. */  
    Activity getDestination();  
  
}
```

在 pvm 中,也包括了对 event 的支持,event 的接口如下.

```
public interface EventListener extends Serializable {  
  
    void notify(EventListenerExecution execution) throws Exception;  
  
}
```

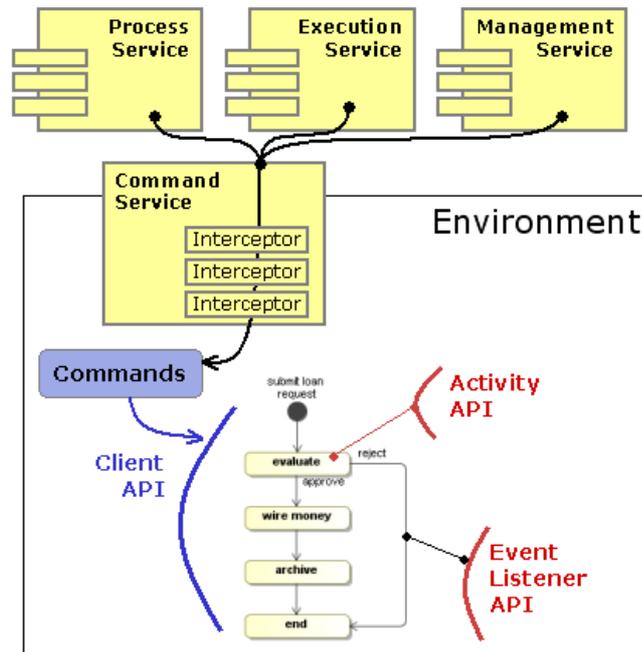
如我们之前所说的, *ProcessDefinition* 是由 Activity, Transition 以及 Event 组成的, ProcessDefinition 是由 *ProcessDefinitionBuilder* 的 API 来创建.我们稍微看下这个 API 的使用.

```
ClientProcessDefinition definition = ProcessDefinitionBuilder.startProcess("jeffProcess")
```

这里,我们注意到返回的是 *ClientProcessDefinition*, 那么他和 *ProcessDefinition* 的区别在哪儿呢? *ClientProcessDefinition* 是继承 *ProcessDefinition*, 他还包括了一些负责创建和启动 *ProcessInstance* 的方法. 类似的,你会发现有 *ClientProcessInstance* 和 *ProcessInstance* 接口,他们的区别也是 *ClientProcessInstance* 多了负责启动的 start 方法.

2.5. API 整体架构图

我们最后从整体上来看下这些 API 之间的联系,这个图片很清楚的描述了他的架构思路



这里需要注意的是,本身 PVM 内部的流程定义模型是 POJO 的,所以如果你只是想测试流程的正确性,你只需要直接使用 Client 的 API,比如 ClientProcessDefinition, ClientProcessInstance 等的 API. 不需要去调用 RepositoryService, ProcessEngineService. 这些 Service 是针对你把 ProcessDefinition 保存在数据库的情况下才需要用的.

3. PVM Model(模型)解析

这里,我们将利用 PVM 所提供的 Model,来实现一个基本的工作流引擎.

3.1. ActivityBehaviour(内部节点)

正如我们之前所说的,ActivityBehaviour 是整个流程的定义核心所在,我们再看下它的 API.

```
public interface ActivityBehaviour extends Serializable {
    void execute(ActivityExecution execution) throws Exception;
}
```

当之行到 `ActivityBehaviour` 的时候,整个流程的走向完全是由他的 `execute()` 方法来决定。比如你可以调用 `execution.end()` 来结束这个流程,或者调用 `execution.waitForSignal()` 进入一个等待状态。我们接下去来实现一个很简单的 `ActivityBehaviour`。

```
public class Display implements ActivityBehaviour {
    String message;

    public Display(String message) {
        this.message = message;
    }

    public void execute(ActivityExecution execution) {
        System.out.println(message);
    }
}
```

如果我们在 `execute()` 方法中,没有显示的调用 `execution` 中的方法,比如 `waitForSignal()`,`take(transition)` 等方法,那么默认的顺序是这样的。

1. 如果当前的节点有默认的 `outgoing transition`,那么就调用这个默认的 `transition`。
2. 如果当前的节点有父节点,那么就调用到父节点。
3. 最后,如果都没有,那就调用 `execution.end()` 方法。

我们先用这个 `Display`,来创建下面的 `process`。

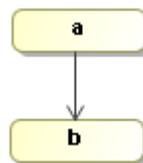
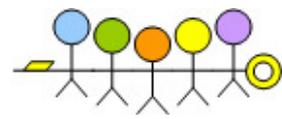


Figure 1. Display example process

```
ClientProcessDefinition processDefinition = ProcessDefinitionBuilder.startProcess("helloworld")
```



```
.startActivity("a", new Display("Hello"))  
  
.initial()  
  
.transition("b")  
  
.endActivity().  
  
.startActivity("b", new Display("World"))  
  
.endActivity()  
  
.endProcess();
```

然后,调用

```
processDefinition.startProcessInstance();
```

就会得到如下的结果

```
Hello  
  
World
```

我们这个 Display 的节点就是采用的隐式 execution 执行方法.

细心的你会发现,我们在定义 Process Definition 的时候,总是要在调用一个 initial()方法. 这是设定当前这个节点为流程的起始节点.

3.2. ExternalActivityBehaviour(外部节点)

外部节点就是代表着,这个活动还需要系统外部的配合,比如说人工的配合才能使得这个流程继续下去. 我们一般称这种的节点为 Wait State. 因为他需要一直等待,直至外部活动的促发,然后流程才继续. 这种的节点需要实现 ExternalActivityBehaviour 的 API.

```
public interface ExternalActivityBehaviour extends ActivityBehaviour {  
  
    //handles an external trigger.
```



```
void signal(ActivityExecution execution, String signalName, Map<String, ?> parameters) throws  
Exception;  
}
```

跟内部节点类似,执行到 `ExternalActivityBehaviour` 的时候,也是执行它的 `execute()`方法,但是一般来说,在外部活动的 `execute()`方法中,会调用 `execution.waitForSignal()`方法,使得 `activity` 进入一个等待状态. 直到外部调用 `signal()`方法来使得流程再次从等待状态变成激活.一般来说在 `signal()`方法中,会调用 `execution.take(signalName)`根据 `signalName`(也就是 `transition name`)去找到下一个节点,然后把整个流程走到下一个节点.

很简单的一个例子是,比如说一个申请审批的流程,员工递交一份申请上去,然后继续就进入一个 `wait state` 的状态,因为他需要经理的审批(也就是一个人工的活动),那么经理可以选择一个 `ok` 的 `signalName`,使得整个流程进入到下一个节点,这里就好比是结束的节点,又或者使得整个流程直接结束.

我们接下来实现一个简单的 `WaitState`,实现 `ExternalActivityBehaviour` 的接口.

```
public class WaitState implements ExternalActivityBehaviour {  
  
    public void execute(ActivityExecution execution) {  
  
        execution.waitForSignal();  
  
    }  
  
    public void signal(ActivityExecution execution,  
  
                        String signalName,  
  
                        Map<String, Object> parameters) {  
  
        execution.take(signalName);  
  
    }  
  
}
```

一样的,我们来看一个简单的从 `a->b` 的流程.这次不同的是,`a` 和 `b` 都是 `wait state`.

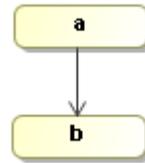


Figure 2. The external activity example process

ProcessDefinition 的定义

```
ClientProcessDefinition pd = ProcessDefinitionBuilder.startProcess("helloworld")

    .startActivity("a", new WaitState())

    .initial()

    .transition("b", "b-transition")

    .endActivity()

    .startActivity("b", new WaitState())

    .endActivity()

    .endProcess();
```

启动这个 ProcessDefinition

```
ClientProcessInstance instance = pd.startProcessInstance();

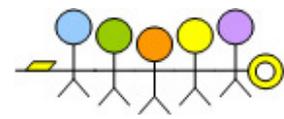
instance.isActive("a")
```

在启动之后,因为执行到 a 的时候,是一个 wait state,所以,当前的流程活动应该是指向 a. 如果要到 b 这个 activity,那么就需要调用

```
instance.signal("b-transition");

instance.isActive("b")
```

那么,你就会发现,经过我们调用 signal 方法,instance 根据所提供的 transitionName (b-transition),找到下一个节点,也就是 b. 但因为 b 也是一个 wait state,所以此刻,整个流程就停留在了 b 节点身上.



3.3. 基本流程的实现

接下来,我们基于前面两种节点的实现,来实现一个稍微比较正式的流程(loan process).

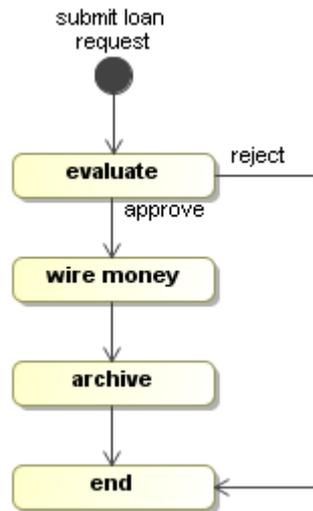


Figure 3. The loan process

ProcessDefinition 的定义

```
ClientProcessDefinition pd = ProcessDefinitionBuilder.startProcess("loanprocess")
    .startActivity("submit loan request", new Display("submit a loan request"))
    .initial()
    .transition("evaluate", "evaluate-transition")
    .endActivity()
    .startActivity("evaluate", new WaitState())
    .transition("wiremoney", "approve")
    .transition("end", "reject")
    .endActivity()
    .startActivity("wiremoney", new Display("wire the money"))
    .transition("archive")
    .endActivity()
```

```

.startActivity("archive", new WaitState())

.transition("end", "done")

.endActivity()

.startActivity("end", new WaitState())

.endActivity()

.endProcess();

```

启动这个 processInstance

```

instance = pd.startProcessInstance();

```

启动这个 processInstance 后,它开始点在 submit loan request 这个节点,后面经过 Display 这个节点,默认走到了 evaluate 这个节点. 因为 evaluate 是个 wait state,所以流程停在了 evaluate.

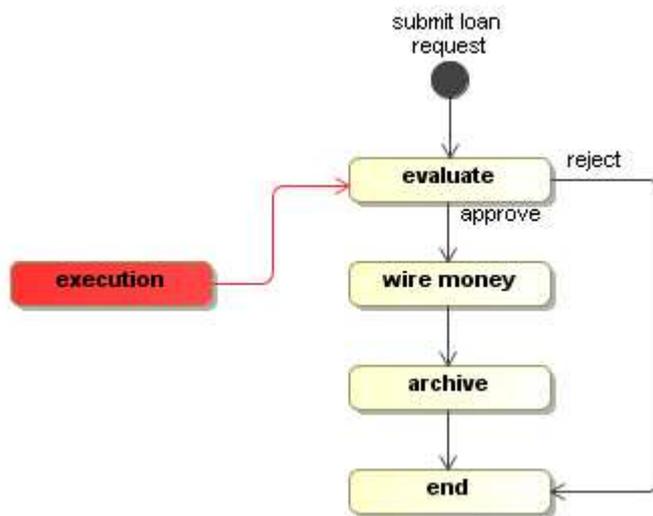


Figure 4. Execution positioned in the 'evaluate' activity

现在呢, evaluate 这个节点有两条支路,一个是 approve,指向 wiremoney 节点;另外一个 reject,直接走向 end. 假设我们选择 approve 这条支路.

```

instance.signal("approve");

```

那么,我们就走向了 wiremoney 这个节点,因为 wiremoney 是个 Display 节点,所以它显示完后,默认的走向下一个节点,archive.

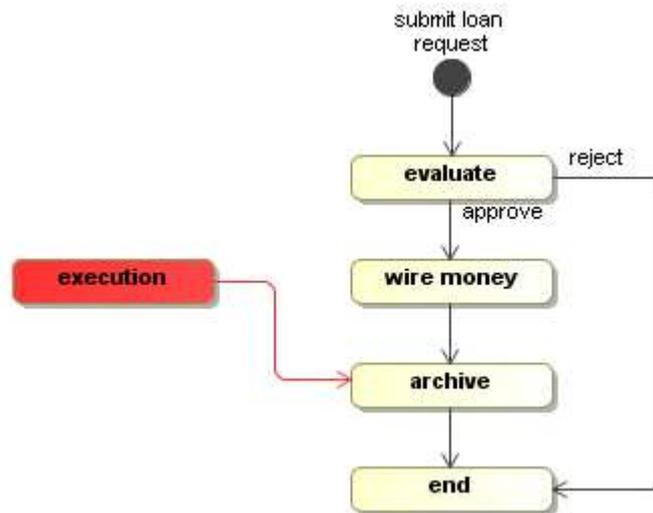


Figure 5. Execution positioned in 'archive' activity

同样的,因为 archive 节点是个 wait state,所以需要再一次的 signal,才能走到 end 这个节点.

```
instance.signal("done");
```

这样的话,整个流程就会走向了 end 节点.

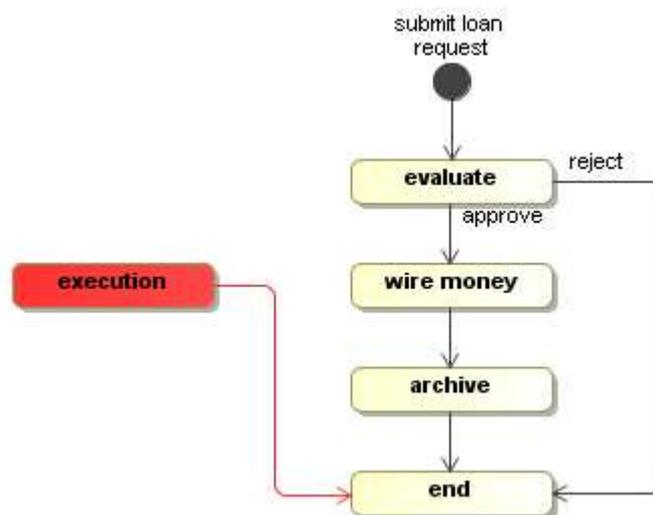




Figure 6. Execution positioned in the 'end' activity

3.4. 事件机制 (Event and EventListener)

事件的订阅可以通过实现 `EventListener` 来实现.

```
public interface EventListener extends Serializable {  
    void notify(EventListenerExecution execution) throws Exception;  
}
```

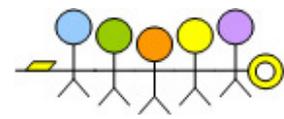
`Event` 概念的引入,主要是为了弥补分析员(`Business Analyst`)和开发人员(`Developer`)之间的不同需求.
`developer` 可以使用 `Event` 在一些节点上来做一些操作(比如说操作数据库),这样呢,也不会影响整个流程,
所以分析员不用去关心这些具体的 `Event`, 他们只需要看流程是否跟他们所期望的是一致的.

具体的 `Event` 是由 `ObservableElement` 和 `EventName` 来构成的.

```
public interface EventListenerExecution extends OpenExecution {  
    void fire(String eventName, ObservableElement eventSource);  
}
```

我们来实现一个简单的 `EventListener`, 叫 `PrintLn`

```
public class PrintLn implements EventListener {  
  
    String message;  
  
    public PrintLn(String message) {  
        this.message = message;  
    }  
}
```



```
public void notify(EventListenerExecution execution) throws Exception {  
  
    System.out.println(message);  
  
}  
  
}
```

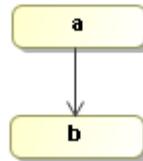


Figure 7. The PrintLn listener process

我们看下是怎么来定义一个具备有 Events 的 ProcessDefinition:

```
ClientProcessDefinition pd = ProcessDefinitionBuilder.startProcess("ab")  
  
    .startActivity("a", new Display("Testing Event"))  
  
    .initial()  
  
    .transition("b")  
  
    .startEvent(Event.END)  
  
    .listener(new PrintLn("leaving a"))  
  
    .listener(new PrintLn("second message while leaving a"))  
  
    .endEvent()  
  
    .startEvent(Event.TAKE)  
  
    .listener(new PrintLn("taking transition"))  
  
    .endEvent()  
  
    .endActivity()  
  
    .startActivity("b", new WaitState())  
  
    .startEvent(Event.START)  
  
    .listener(new PrintLn("entering b"))  
  
    .endEvent()
```



```
.endActivity()  
  
.endProcess();
```

我们可以看到,一个事件可以有无穷多个的 Listener(s).

至此,我们主要看了 PVM 里面内部 Model 的一些设计,一些核心的概念和 API.

[1] 图来自于 <http://www.blogjava.net/RongHao/archive/2009/05/07/269465.html>

[2] <http://www.jboss.org/jbossidentity>