

# jPdl 参考手册

本文描述了 processdefinition.xml 文件的 schema.

## 流程存档(process archive)

一个流程存档是对一个业务流程管理(BPM)的形式化描述.它被打包成为一个 jar 格式的文件,通常以 .par 为扩展名.jBpm 描述一个流程需要三种类型的数据:

**1 声明式的业务流程描述(a declarative description of the business process):** 在 jPdl 中,这些声明位于 **processdefinition.xml** 文件中.本文的大部分都是在解释这一文件的 schema.

**2 程序逻辑(programming logic):** 为了将程序逻辑与 processdefinition.xml 文件中描述的流程联系起来,可以将 java 类包括在流程存档中.所有在流程存档中的类应该位于 /classes 子目录下.

**3 其他的资源文件:** 在工作流引擎的一个客户端中,你可能想要将在运行时可能用到的各种各样的资源文件包括在流程中.例如:与分配给某人的任务相关的表单的描述.对在流程中使用的资源文件的类型, jBpm 并没有强加任何限制.参见 [Other process archive files](#)

## 版本控制机制(Versioning Mechanism)

基本上,jBpm 的版本控制机制可以归结为以下几条原则:

- 每次当一个流程存档部署时,一个新的流程定义(process definition)就在 jBpm 的数据库中创建
- 在部署中,jBpm 分配给流程定义一个版本号.如果流程的名称相同的话,流程存档也认为是相同的.分配版本号时,jBpm 使用 1+(当前具有相同名称的流程定义中最高的版本号)或者 1(如果它是第一版的话).从 jBpm 的 API 你能够取得给定名称的最新的流程定义.
- 一旦一个流程实例(即一次流程执行)根据一个给定的定义启动了之后,流程实例会按照相同的定义执行直到它结束为止.
- 这样的话,每个流程实例将会从最新的定义开始执行,并在它的整个生命周期内都按照相同的定义运行.
- 请注意 jBpm 甚至能够对与一个流程相关的程序逻辑进行版本控制.通过在流程存档中包括 java 类,jBpm 能够将每个流程定义的类分离.

## processdefinition.xml 的 schema

### Document type definition

```
<!DOCTYPE process-definition PUBLIC "-//jBpm/jBpm Mapping DTD 2.0 beta3//EN"
"http://jbpm.org/dtd/processdefinition-2.0-beta3.dtd">
```

*the document type definition of processdefinition.xml*

## process-definition

```
<!ELEMENT process-definition ( description?, swimlane*,
type*,
start-state, ( state | milestone |
process-state |
decision |
fork |
join
)*,
end-state, action* ) >
<!ATTLIST process-definition name CDATA #REQUIRED >
```

*dtd fragment for process-definition*

## state

```
<!ELEMENT state ( description?, assignment?, action*, transition+ ) >
<!ATTLIST state name CDATA #REQUIRED >
```

*dtd fragment for a state*

在 jBpm 中,状态(state)这一术语与有限状态自动机(FSM)或者 UML 状态图中的 state 具有相同的意义.在 jBpm 中进行业务流程建模的目的是为了创建一个软件系统.我们将流程定义的视为软件开发的一个部分.所以 jBpm 从 jBpm 所构成的软件系统这一角度来解释状态这一术语.

状态是 jBpm 的一个核心概念.当开始在 jBpm 中进行流程建模的时候,首先要做的事情就是考虑业务流程的状态.状态将会成为你定义的流程的基本框架.jBpm 围绕状态这一概念进行是有明显的道理的.那就是状态这一概念在编程语言中并没有与之相对应的部分.一个软件程序要么是在运行中要么不是.而一个业务流程一般会有几个相互独立执行的程序逻辑.jBpm 允许对这些相关的程序逻辑之间的状态进行建模.

jPdl 同时也使用变迁(transition),分支(fork),合并(join)和里程碑(milestone)这些概念,用它们来定义状态之间的控制流.请注意这些状态之间的控制流.当一般的(plain)程序逻辑更加适合时,业务流程开发者应该在一个流程事件上标明一个动作(action).jPdl 的哲学就是使用状态管理来扩展 java,并使之与编程语言的能力之间的重叠尽量的小.这就将 jPdl 与其他的业务流程定义语言区分开来,比如 BPEL.其他的流程语言并没有作出这种明确的分离.

## assignment

```
<ELEMENT assignment EMPTY >
<!ATTLIST assignment swimlane CDATA #IMPLIED
assignment (optional|required) #IMPLIED
authentication (optional|required|verify) #IMPLIED >
```

*dtd fragment for an assignment*

当一个流程执行到某一个状态时,工作流引擎会一直等待直到它得到了某个外部的触发(使用jBpm的API方法[ExecutionService.endOfState\(...\)](#)进行触发).在这一方法中,jBpm将会计算流程实例的下一个状态.

因此一个状态能够视作对一个外部参与者的依赖.这个外部参与者可以是一个人,一组人或者是一个系统.有许多方法可以使得业务流程中的状态(state)与任务(task)相关联,这些方法基本可以分为两种:

### 基于客户的分配(Client based assignment)

在这种策略中,jBpm 的用户要自己管理它们用户的任务列表.jBpm 在这种情况下仅仅是用作为有限状态自动机的一个执行引擎.jBpm 的责任就是计算并且跟踪流程执行的状态,而客户端的责任则是确保每个人都知道自己要做什么.典型的,客户端就是负责找出某一状态下的所有的流程实例(或者令牌(token)).

### 基于流程的分配(Process based assignment)

(这一分配策略仅仅在 jBpm 2.0 beta3 之前的版本才支持)

在这一分配策略中,jBpm 负责为参与者分配状态,并且跟踪任务列表.在 jBpm 中,流程执行的进度使用令牌来跟踪.一个令牌有一个指针指向一个状态和一个参与者.在 jBpm 中,一个参与者都是通过一般的 `java.lang.String` 来进行引用的.为了表明 jBpm 如何将令牌分配给参与者,有几个相关联的事件.我们下面一个个进行分析:

无论何时,客户端启动了一个新的流程实例([starts a new process instance](#))或者通知了某个状态的结束([end of a state](#)), jBpm开始为这一流程实例计算下一个状态.关于分配,首先我们想说的一件事就是这些API方法的第一个参数就是这一参与者.这一参数是用来表明这一方法是由谁来执行的.在启动一个新的流程实例的情况下,一个根令牌(root-token)在开始状态(start-state)被创建.在状态结束(endOfState)的情况下,需要一个tokenId作为一个参数以标明它处在哪个状态(a tokenId needs to be specified as a parameter which is in some state).然后,jBpm将会为这一令牌开始计算新的状态.为了简化,我们暂时先不考虑并发的这个问题.这一令牌将会经过变迁和节点(node)直到它到达了一个状态节点.那时候,这一令牌需要被分配给某个参与者.在选择参与者([selection of the actor](#))之后, jBpm将会把这一相关的参与者与这一令牌关联起来并且调用方法(startProcessInstance或者endOfState)然后返回.

这样,在某个状态的令牌有了对这个参与者的引用之时,就意味着流程实例开始等待这一参与者给jBpm引擎提供一个外部的触发.在这种情况下,流程中的一个状态对应于一个给用户的任务.jBpm通过搜索所有的已经被分配给给定的参与者的令牌来计算任务列表([the tasklist](#)).现在,参与者可以从列表中选择令牌并再次通知这一状态的结束.

分配还包括了更多的尚未考虑的属性:分配(**assignment**)和授权(**authentication**).assignment属性有两个值:可选的(optional)或者必需的(required).必需的意味着在执行到某一状态时,jBpm将会检查这一令牌是否已经实际分配给了一个参与者.如果没有通过检查的话,会抛出一个AssignmentException异常.如果分配是可选的话(默认情况),jBpm允许一个令牌离开某一状态时仍然还未分配参与者.authentication属性明确了哪个参与者可以通知状态结束这一约束.这一参与者通过 [endOfState](#)方法中的actorId这一参数来标明.

Assignment has to more attributes that are not yet covered : assignment and authentication. **optional** 属性(默认情况)意味着不必要标明这一状态结束具体是由谁来发出的.required 意味着必须标明一个参与者,而 **verify** 意味着状态结束必须由那个分配了令牌的参与者来发出.

如果想了解更多的关于coping with group assignments,请参见[faqs](#).

## swimlane

```
<!ELEMENT swimlane ( description?, delegation? ) >
<!ATTLIST swimlane name CDATA #REQUIRED >
```

### *dtd fragment for swimlane*

一般的,一个人会负责一个流程中的多个状态.在jBpm中,这是通过创建一个泳道(swimlane)并将所有那个参与者负责的的状态分配道那个泳道中来完成的.一个泳道在一个业务流程中可以被视为一个参与者在这一流程中的角色名称.jBpm对于泳道的解释与UML 1.5 中对于用到的解释相同. 当一个流程执行到某一给定泳道中的一个状态时,引擎开始计算参与者(the [actor is calculated](#)).

```
public interface AssignmentHandler {
String selectActor( AssignmentContext assignerContext );
}
```

### *the [AssignmentHandler](#) interface*

在一个泳道中的委托(delegation)标记指的是一个 AssignmentHandler 的实现.然后那一参与者就使用与这一泳道系统的名称存储在一个流程变量中.下一次当流程到达这一泳道的某一状态时,jBpm 将会注意到这一变量并分配这一令牌给存储在这一变量中的参与者.

所以泳道([swimlane](#))是在流程级别([a process level](#))上定义的,状态([state](#))在分配([assignment](#))中对泳道([swimlane](#))进行引用.

这一计算的结果将会被存储在一个与这一泳道有相同名字的流程变量中.这样当下一个状态到达相同的泳道时,这一状态就会被分配给相同的参与者而不必再次使用 AssignmentHandler. 由于泳道与参与者的关系是存储在一个变量中的,可以通过更新这变量来操纵这一关系.

## variable and type

```
<!ELEMENT type ( description?, (delegation|transient), variable* ) >
<!ATTLIST type java-type CDATA #IMPLIED >
<!ELEMENT transient EMPTY >
<!ELEMENT variable EMPTY >
<!ATTLIST variable name CDATA #REQUIRED >
```

### *dtd fragment for type and variable*

## Variables

一个变量(variable)是一个与流程实例(即一次流程执行)相关联的键值对(key-value pair).键值对的key是java.lang.String类型的,the value is any [POJO](#)'s of any java type.所以即使是jBpm不知道的java类型也能够在流程变量中使用.

变量存储了一个流程实例的上下文信息(context information).变量能够通过三种方式进行设置:

- ExecutionService.startProcessInstance( String actorId, Long definitionId, **Map variables**, String transitionName )
- ExecutionService.endOfState( String actorId, Long tokenId, **Map variables**, String transitionName )
- ExecutionService.setVariables( String actorId, Long tokenId, **Map variables** )
- ExecutionContext.setVariable( String name, Object value )
- ExecutionService.getVariables( String actorId, Long tokenId )
- ExecutionContext.getVariable( String name )

当使用变量时,我们尝试在 jBpm-API 中尽量模仿类似 java.util.Map 的语义.这就是说一个变量仅仅当它被插入(inserted (read:set))的时候才进行实例化,并且任何的 java 类型都可以作为值.

## Type

一个类型(type)标明了 jBpm 应该怎样在数据库中存储这一变量的值.jBpm 有一个文本域(text field)用以存储值.因此对象和文本之间的转化是通过一个序列化器(Serializer)来进行的:

```
public interface Serializer {
String serialize( Object object );
Object deserialize( String text );
}
```

### *the [Serializer](#) interface*

一个类型可以看成是对一个序列化器的引用.jBpm 包括了以下 java 类型的序列化器的默认实现:

```
java.lang.String
java.lang.Long
java.lang.Double
```

### *by default supported java-types*

## Variable-type matching

默认支持的 java 变量类型不需要在 processdefinition.xml 进行声明.jBpm 有一种半自动的声明机制:当一个变量实例化的时候,jBpm 尝试通过检查变量值的 java 类型计算变量的类型.如果变量值是一个默认的支持类型,那么 jBpm 就使用这一类型.否则的话, jBpm 检查变量值的 java 类型是否在 processdefinition.xml(java-type 属性)中有对应的声明.请注意在查找这一匹配的过程中,jBpm 也会将变量值的超类型(super-type)考虑进去.如果没有找到对应的类型,jBpm 将这一变量视为一个临时变量(Transient Variables).

## Transient variables

一些变量不需要持续化到数据库中.'to.email.address'这一变量在 endOfState()这一 jBpm-API 中被使用.状态有一个离开的变迁(leaving transition),这一变迁有一个动作(action)是发送一封 email 到一个给定的地址.如果这是变量'to.email.address'唯一的用处的话,那么它并不需要进行持续化.因此,jBpm 支持临时变量.临时变量并未存储在数据库中,它们仅仅能够被用 jBpm-API 调用方法的内部.(Transient variables are not stored in the database and

can only be used inside the jBpm-API method call they're being supplied.)换句话说,临时变量的作用域是在 jBpm-API 方法调用期间.

## start-state

```
<!ELEMENT start-state ( description?, transition+ ) >
<!ATTLIST start-state name CDATA #REQUIRED swimlane CDATA #IMPLIED >
```

### *dtd fragment for start-state*

开始状态(start-state)是一个流程实例的独特的状态,所有的流程实例都是从这个状态开始的.请注意,在流程实例开始的时候(process-instance-start-time),you can already feed variables in the process.另一个重要的概念是 you can have multiple transitions leaving the start-state.在这种情况下,你需要标明当你开始一个流程实例的时候要引发的是哪个变迁.

## milestone

```
<!ELEMENT milestone ( description?, action*, transition ) >
<!ATTLIST milestone name CDATA #REQUIRED>
```

### *dtd fragment for a milestone*

一个里程碑(milestone)就是一种特殊的状态,通过这种状态,可以同步两个并发执行.当一条路径的执行需要等待另一条路径上的事件发生的时候可以使用里程碑.如果一个里程碑没有达到,一个执行就必须在里程碑处进行等待直到另一个并发路径上的执行也达到里程碑为止.如果里程碑已经达到,这一执行就可以直接通过这一里程碑.关于更多的里程碑的信息和图形演示动画,请参见[the workflow patterns](#).一个里程碑状态与一个或多个动作相关,这些action通知里程碑的到达.这些action能够通过使用默认的

ActionHandler(**org.jbpm.delegation.action.MilestoneReachedActionHandler**)进行建模.所以通知jBpm引擎某个里程碑已经到达的action可以在processdefinition.xml中如下调度:

```
...
<milestone name="theMilestone">
<transition to="stateAfterMilestone" />
</milestone>
...
<state name="stateBeforeReachingMilestone" swimlane="initiator">
<transition to="stateAfterReachingMilestone">
<action>
<delegation
class="org.jbpm.delegation.action.MilestoneReachedActionHandler">theMilestone
</delegation>
</action>
</transition>
</state>
...
```

### *modelling a milestone in the processdefinition.xml*

## process-state

```
<!ELEMENT process-state ( description?, delegation, action*, transition+ ) >
<!ATTLIST process-state name CDATA #REQUIRED>
```

### *dtd fragment for a process-state*

一个流程状态对应与一个超流程(super-process)的调用.当执行到达流程状态时,父流程启动一个子流程.在子流程的执行流程中,父流程仍然处于流程状态.当子流程结束时,父流程就离开了流程状态.

## **decision**

```
<!ELEMENT decision ( description?, delegation, action*, transition+ ) >
<!ATTLIST decision name CDATA #REQUIRED>
```

### *dtd fragment for a decision*

一个决策(decision)决定了多个路径的执行.如果你是一个程序员,那么把它假想成是一个 if-then-else 结构就可以了.当然,一个决策能够有任意多的离开变迁.请注意,当一个工作流引擎需要从根据上下文(即变量)来选择采用那条路由的话,可以使用决策模型进行建模.另一种选择是,采用多个变迁离开一个状态来进行建模.在那种情况下,jBpm 客户端必须通过选择变迁的名称作为 endOfState 方法调用的一个参数以决定触发哪个离开的变迁.

## **fork**

```
<!ELEMENT fork ( description?, delegation?, action*, transition+ ) >
<!ATTLIST fork name CDATA #REQUIRED
corresponding-join CDATA #IMPLIED>
```

### *dtd fragment for a fork*

一个分支(fork)产生多个并发的执行路径.可以通过 ForkHandler 接口来指定自定义的分支行为.但是默认的行为(当这一分支中没有标明委托(delegation)时)是为这一分支的每个离开变迁产生一个子令牌(child-token).所以仅仅当高级的奇特并发时,你才会需要实现自定义的 ForkHandler.

通常,一个分支有其相关的合并(join),这二者共同定义了一个并发块.在默认的分支和合并行为中,仅仅支持严格的嵌套.默认的分叉和合并不支持穿越多个并发块边界之间的的变迁.

```
public interface ForkHandler {
void fork( ForkContext forkContext ) throws ExecutionException;
}
```

### *the ForkHandler interface*

## **join**

```
<!ELEMENT join ( description?, delegation?, action*, transition ) >
<!ATTLIST join name CDATA #REQUIRED
corresponding-fork CDATA #IMPLIED>
```

### *dtd fragment for a join*

一个分支合并多个并发的执行路径.可以通过 `JoinHandler` 接口标明自定义的合并行为.但是默认的行为(当没有在合并中指定委托时)是所有产生的令牌在对应的分支中聚合.在合并中最后一个到达的令牌将会触发父令牌离开合并的这一变迁.所以只有当遇到高级的奇特并发时才需要考虑自己实现一个自定义的 `JoinHandler`.

约束:一个合并仅仅能够有一个离开的变迁.

```
public interface JoinHandler {  
void join( JoinContext joinContext ) throws ExecutionException;  
}
```

### *the JoinHandler interface*

## end-state

```
<!ELEMENT end-state EMPTY >  
<!ATTLIST end-state name CDATA #REQUIRED>
```

### *dtd fragment for end-state*

一个流程定义有且仅有一个终态(end-state).当一个流程实例执行到达终态时,这个流程实例就结束了.

## transition

```
<!ELEMENT transition ( action* )>  
<!ATTLIST transition name CDATA #IMPLIED to CDATA #REQUIRED>
```

### *dtd fragment for a transition*

变迁(transition)用以标明节点之间的有向连接.变迁元素(指的是 `JPdl` 中的 `transition element`)应该放置在节点这个变迁离开的节点的内部.

## action

```
<!ELEMENT action ( delegation ) >  
<!ATTLIST action event-type (process-start|process-end|  
state-enter|state-leave|state-after-assignment|  
milestone-enter|milestone-leave|  
decision-enter|decision-leave|  
fork-enter|fork-every-leave|  
join-every-enter|join-leave|  
transition) #IMPLIED>
```

### *dtd fragment for an action*

一个动作(action)是一段 `java` 代码,在流程执行过程中触发的一个事件时 workflow 引擎会执行这段代码.

动作一直被定义为流程定义元素(`process-definition-element`,例如 **process-definition, state, transition, decision**等等)的子元素.父元素加上事件类型(`event-type`)精确定义了业务流程中动作执行的时间.正如你能想象的,一个动作的可能的事件类型取决与包含动作的这一元素.事件类型名称(`event-type-names`)已经暗示了它们所能应用的元素.你能够在 [the javadocs of EventType](#)找到完整的描述.

```
public interface ActionHandler {  
void execute( ExecutionContext executionContext );  
}
```

*the ActionHandler interface*

## delegation

```
<!ELEMENT delegation ( #PCDATA ) >  
<!ATTLIST delegation class CDATA #REQUIRED>
```

*dtd fragment for a delegation*

作用：解释包含的元素和委托类(delegation class)要实现的接口之间的联系

## 其他的流程存档文件(Other process archive files)

当一个流程存档部署之后,processdefinition.xml文件被解析,相关的信息被存储到jbpm数据库中.你添加到到流程存档的所有其他文件都存储在数据库或文件系统上并且与创建的流程定义相关联.使用jbpm API作为客户端的话,你可以通过

**ExecutionReadService.getFile( Long processDefinitionId, String fileName )**方法来访问这些文件.一个流程存档和一个流程定义的不同与版本控制机制([versioning mechanism](#))有关.

本文是对[jPdl Reference Manual](#)的一个中文翻译,由于刚刚接触 workflow,有失偏颇之处敬请原谅.

倪跃 2005年3月29日

nybonbon@Gmail.com