5_工作流应用中经典问题的解决方案

作者：非也　QQ：20674450　Email：nychen2000@163.com

目　　录

1. 流程定义文件的存储与版本控制

1) 题外话

我们常常人为地将系统划分成所谓的“源代码（如 java 文件）”和“配置文件（或配置数据）"；甚至认为开发“源代码”的工作才是编程工作，编写配置文件、调整配置数据的工作是“免编程"。

对此，我非常不以为然！且不说二者本来就没实质区别，都需要开发人员去开发；如果把“配置文件（或配置数据）”发挥至极端，复杂的不行，开发人员需要花更多的精力和时间去学习的话，“免编程”就没有任何价值！

但是，并不能说 java 文件和 Xml 配置文件(尤其是数据库中的配置数据)没有区别。Java 文件需要编译才能运行，xml 配置文件和数据库中的配置数据修改后立即发挥作用。一般情况，下修改 Xml 配置文件和配置数据下比修改 java 文件简单。

因此，在我看来 java 文件也好，xml 配置文件也好.，配置数据也好都是系统的代码。对他们的开发活动都是“编程"。这几种代码都需要某种形式的版本控制。

2) Fire Workflow 流程定义文件的存储与版本控制

Fire Workflow 流程定义文件是整个业务系统代码的一部分，以 xml 的形式表现，开发时存放在/src 目录下，运行时存放在 classpath 中（如/bin 或者/WEB-INF/classes 或者某个 jar 包中）。在运行时通过“流程定义服务”的实现类 org.fireflow.engine.definition.DefinitionService4FileSystem 从 classpath 中读取定义文件。

这种流程文件的存储方案非常简单，但是没有照顾到如下需求：1）在系统上线运行后，流程常常需要调整；在流程定义更新后，系统中仍然有旧版本的流程实例在运行。2）最终用户有自行调整业务流程的需求，存储在 classpath 中的流程定义不能适应这种需求。

因此上述流程文件存储方案适合于系统开发阶段，方便开发人员快速地结合业务数据进行系统测试。而且，Fire Workflow 提供了和流行的 IDE 结合的很好的设计器，使得开发测试业务流程就像开发测试一个 java 类一样简单。

Fire Workflow 流程定义文件还可以存储在后台数据库表 T_FF_DF_WORKFLOWDEF 中。该表可以记录同名流程的各个版本。Fire Workflow 要求每个流程实例只能在自己的流程版本上运行，不能“迁移”到同名流程的其他版本。这种规定不是 Fire Workflow 想“偷懒"，而是由于 Fire Workflow 是基于比较严密的 Petri Net 流转算法，同名的流程不同版本间很可能改变了网络拓扑结构，这种情况下流程实例“迁移”到新的流程版本后无法正确运行。以我的经验看，Fire Workflow 的这种规定适合绝大多数业务场合。

至于用户自行调整流程，将来可以通过 FireWorkflow 的 WEB 工具完成。（该 Web 工具尚未实现 20090210）。

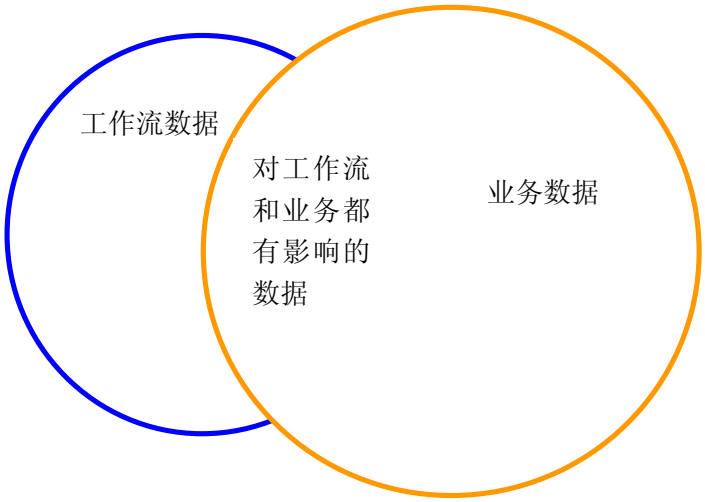
对于存储在 T_FF_DF_WORKFLOWDEF 中的流程定义文件，需要用“流程定义服务"

的另外一个实现类 org.fireflow.engine.definition. DefinitionService4DBMS 进行读取。该类最终调用 IPersistenceService 完成实际的数据库 IO 操作。（DefinitionService4DBMS 尚未测试，将流程定义文件存储到数据库中的 Web 页面尚在开发中，20090210）

2. 业务数据 vs 工作流数据

1) 业务数据及工作流数据的划分和存储

业务数据和工作流数据的划分以及存储方案一直是一个经典的且令人头痛的问题。我认为业务数据和工作流数据不可能截然分开，二者有交集。如下图



上图中哪些数据对工作流和业务都有影响呢？

例如：受理编号、审核意见。这些数据本来应该是业务数据，但是流程在流转的过程中也需要他们。受理编号用于关联流程实例和业务表单，审核意见决定流程的下一步走向。

再例如：流程实例的当前环节和环节状态。这些数据本来是流程数据，但是业务系统经常需要查询这些信息。

上图交集中的数据怎么存储呢？通常有如下几种方案。

- 方案一、工作流数据侵入业务系统表

在业务系统表结构中不但有业务字段，还有工作流相关的字段，例如：流程实例编号、当前环节名称、当前环节状态、下一环节名称等等。

这种方法的优点是，查询统计比较方便。例如某业务在当前在那个环节，处于什么状态，一目了然。

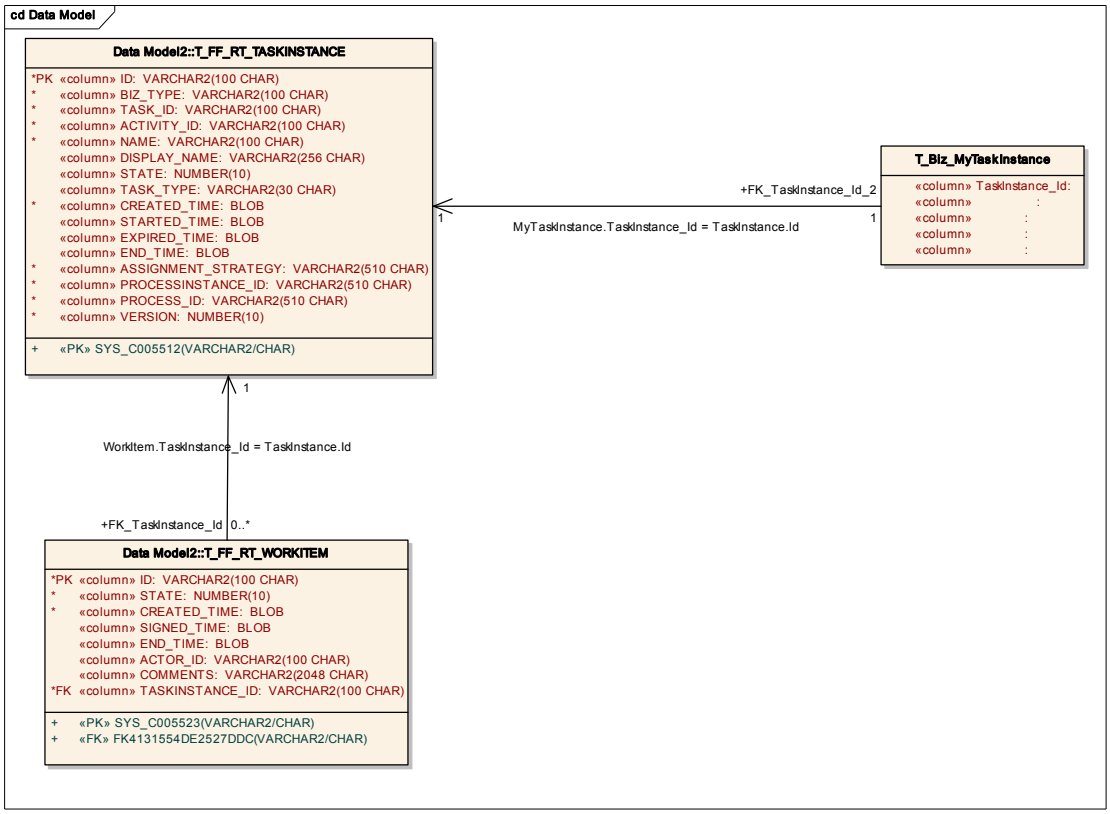
这种方法的缺点是使得业务数据不“纯净”。而且如果业务审批项目比较多，则在每个审批项目的主业务表中都需要嵌入流程字段，会使得流程系统和业务系统耦合得很紧。

- 方案二、将业务数据当作流程变量存储到工作流系统中

由于流程变量一般是以 key-value 方式存储的，不利于查询统计。例如“要根据流水号查询出该业务当前处于哪个环节”实现起来很麻烦。

➢ 方案三、Fire workflow 的方案

Fire workflow 的方案是扩展任务实例表 T_FF_RT_TaskInstance。这个扩展不是在 T_FF_RT_TaskInstance 中增加字段，而是增加一张新表，假设新表的名称是 T_Biz_MyTaskInstance。二者的关系如下图，(字段以 Fireflow Example 中的某商场送货流程为例)。



上述设计虽然冗余了部分业务特征数据(业务流水号、客户名称、商品名称、商品数量)，但是保持业务表的纯净，不会有工作流数据侵入。

最最重要的是，这种设计的使得查询统计非常方便。

例如：如果你想跟踪某个客户购买的商品现在处于那个环节了，只要联合 T_FF_RT_TaskInstance 和 T_Biz_MyTaskInstance 进行查询即可。而且这两个表的数据是 1 对 1 的，通过 TaskInstanceId 进行关联，性能也不会有大的问题。(当然，性能还是有优化的空间，具体见《5_工作流应用中经典问题的解决方案》)。

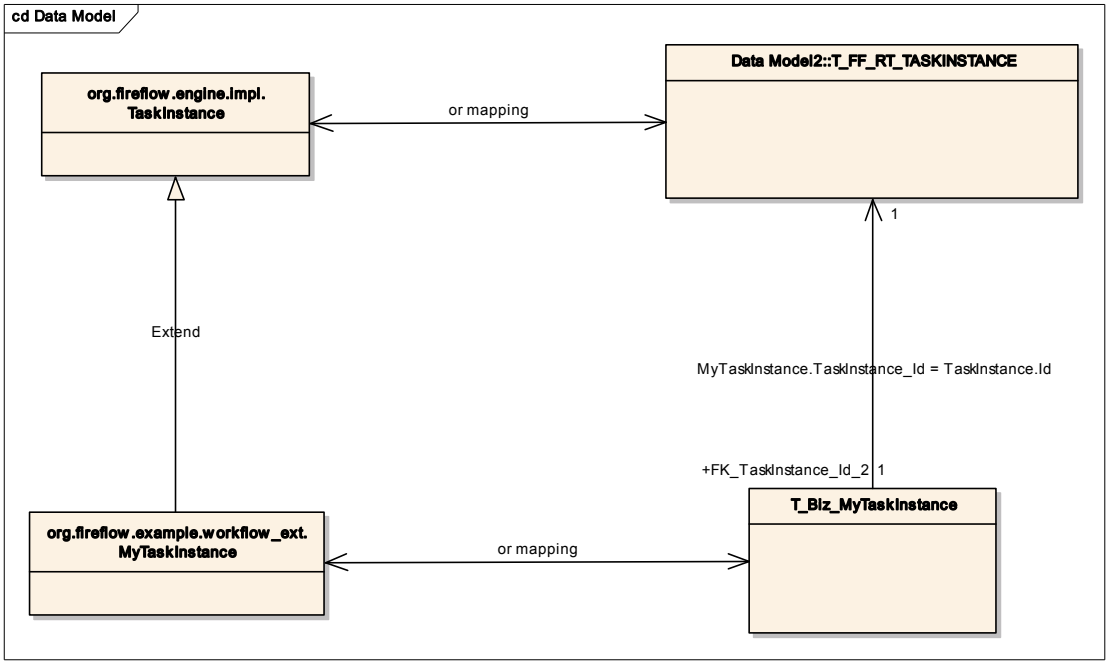
再例如：你需要查询某个客户的货是谁送的，只要联合 T_FF_RT_TaskInstance、T_Biz_MyTaskInstance 和 T_FF_RT_WorkItem 即可。

3) 必要的扩展

现在的问题是 T_Biz_MyTaskInstance 中的数据怎么填入呢？

Hibernate 的父子对象映射机制给我带来了便利。我们知道 org.fireflow.engine.impl.TaskInstance 映射到 T_FF_RT_TaskInstance。那么我们扩展 org.fireflow.engine.impl.TaskInstance 类，得到

org.fireflow.example.workflow_ext.MyTaskInstance。他们的继承关系和映射关系如下图。



仅仅扩展 org.fireflow.engine.impl.TaskInstance 还是没有解决问题，还需要扩展 org.fireflow.engine.taskinstance.BasicTaskInstanceManager。

BaiscTaskInstanceManager 的作用是为工作流引擎创建 org.fireflow.engine.impl.TaskInstance 实例。其代码很简单，如下图。

```
1.  public class BasicTaskInstanceManager extends AbstractTaskInstanceManager {
2.
3.      /*
4.       * (non-Javadoc)
5.       *
6.       * @see org.fireflow.engine.taskinstance.AbstractTaskInstanceManager#createTaskInstance(
7.       *      org.fireflow.model.task.Task)
8.       */
9.      @Override
10.     public ITaskInstance createTaskInstance(IToken token, Task task, Activity activity)throw
11.
12.         TaskInstance taskInstance = new TaskInstance();
13.
14.         return taskInstance;
15.
16.     }
17. }
```

我们扩展 BaiscTaskInstanceManager 得到 org.fireflow.example.workflow_ext.MyTaskInstanceManager。该类负责给工作流引擎提供 MyTaskInstance 实例同时填充 MyTaskInstance 中的业务字段。因为 MyTaskInstance 是 TaskInstance 子类，所以在 Engine 中和 TaskInstance 一样是“畅行无阻”的。MyTaskInstanceManager 代码如下图。

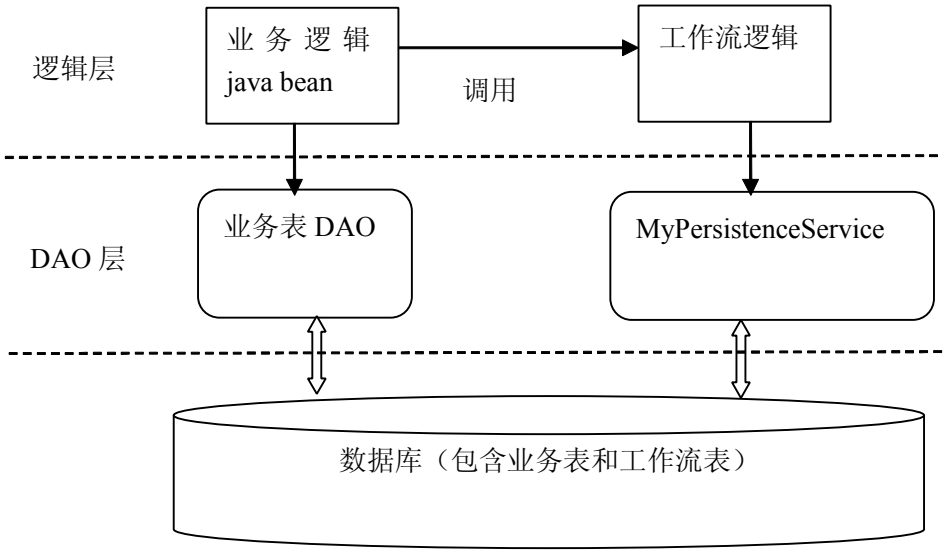
MyTaskInstanceManager 必须注册到 FireflowContext.xml 中，见 6-4)-a)章节。为了使得 MyTaskInstance 实例能够被 hibernate 持久化，还必须将 MyTaskInstance.hbm.xml 配置文件加入到 hibernate 的映射文件列表中。

```java
public class MyTaskInstanceManager extends BasicTaskInstanceManager {
    @Override
    public ITaskInstance createTaskInstance(IToken arg0, Task arg1,
            Activity arg2) throws EngineException {
        MyTaskInstance taskInst = new MyTaskInstance();
        IProcessInstance procInst = arg0.getProcessInstance();

        String sn = (String)procInst.getProcessInstanceVariable("sn");
        taskInst.setSn(sn);

        String customerName = (String)procInst.getProcessInstanceVariable("customerName");
        taskInst.setCustomerName(customerName);

        String goodsName = (String)procInst.getProcessInstanceVariable("goodsName");
        taskInst.setGoodsName(goodsName);

        Long quantity = (Long)procInst.getProcessInstanceVariable("quantity");
        taskInst.setQuantity(quantity);

        return taskInst;
    }
}
```

在这里必须说明一点，在我的设计中 MyTaskInstanceManager 的业务字段来自于流程变量（如上图）。因此，如果希望填充或者变更 MyTaskInstance 实例中业务字段的值，必须先调用 IProcessInstance.setProcessInstanceVariable(String name,Object value)设置相关的变量，然后再执行流程操作。当然，你也可以考虑更好的策略。

3. 流程数据存取设计与事务一致性

Fire Workflow 的流程数据的存取都是通过 org.fireflow.engine.persistence.IPersistenceSerivce 的实现类完成的，假设这个实现类的名字是 MyPersistenceService。那么 MyPersistenceService 其实就是一个存取多张工作流表的 DAO 而已，和你的业务系统的 DAO 处于相同的地位，没有任何区别。示意图如下。



如果你的系统采用的是申明式事务，将事务申明添加到 MyPersistanceService 上即可以实现业务操作和流程操作在同一个事务中。

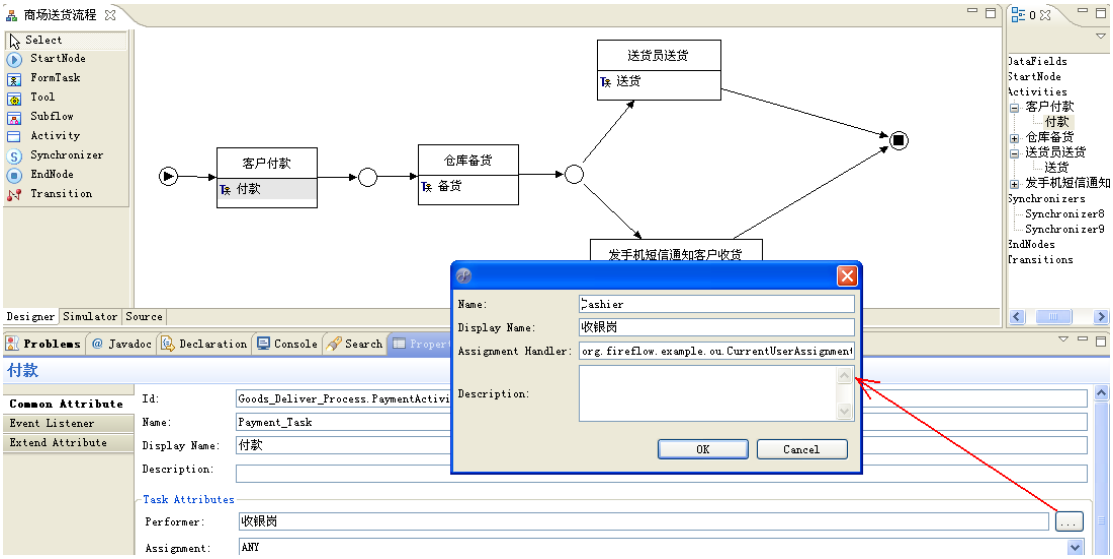
如果你的系统采用的是编码实现的事务，只要将调用流程引擎的代码和和业务逻辑代码放在同一个事务中即可。在 Fire Workflow Example 中，采用的是编码实现的事务，可以参考。

Fire Workflow 已经提供了 org.fireflow.engine.persistence. IPersistenceSerivce 一个缺省实现类 org.fireflow.engine.persistence. hibernate.PersistenceServiceHibernateImpl，该类基于 hibernate 的，你当然也可以自行编写基于 iBATIS，Toplink，甚至 JDBC 的实现类。

4. 与用户管理系统的接口

Fire workflow 认为，工作流系统是整个业务系统的一部分，不应该有自己独立的用户管理系统。在 Fire Workflow 中，与用户管理系统接口的地方只有一个，那就是：新产生的工单分配给“谁”？

Fire Workflow 规定，在流程定义时，所有的 Form 类型的 Task 都必须设置其 Performer 属性。如下图：



Performer 是一个复合属性，其本身有如下属性

属性名称	说明
Name	Performer 的 Name，说白了就是角色名称
Display Name	显示名称，即角色的中文名称
Assignment Handler (*这个 Handler 是最关键的*)	业务系统中一个实现了 org.fireflow.engine.ou. IAssignmentHandler 接口的类。该类的作用就是：在运行时，根据角色名称返回该角色的所有成员，然后把这些成员的 ID 列表交给工作流系统。工作流系统会根据输入的 ID 列表分配工单。
Description	角色描述。

Assignment Handler 实现类的示例参见 Fire Workflow Example 的 org.fireflow.example.ou.

AssignmentHandler。代码如下图。

```
public class AssignmentHandler implements IAssignmentHandler {
    public void assign(IAssignable arg0, String arg1) throws EngineException ,KenelException
        String roleName = arg1==null?"":arg1.trim();
        //获得角色中所有的用户
        List users = OUManagementMock.getInstance().getAllUsersForRole(roleName);
        if (users==null || users.size()==0){
            throw new EngineException("没有任何用户和角色"+arg1+"相关联，无法分配任务。");
        }
        List userIds = new Vector();
        for (int i=0;i<users.size();i++){
            userIds.add(((User)users.get(i)).getId());
        }

        //将用户ID的列表交给工作流系统
        arg0.asignToActors(userIds);
    }
}
```

此处，或许有人有疑问：我们的系统的角色都是用户自己定义的啊，系统发布时没有任何角色，而 Fire Workflow 要求在设计时就确定角色名称，不合理。

我确实认为角色是用户自己定义的，但是为什么不能在系统设计阶段根据系统的业务特点预定义一些系统内置的角色呢？在我看来，不但可以这样做，而且理应这样做。业务系统的角色名称可以五花八门，但是业务的特征是恒定的，不管角色取什么名称，都是指同一个对象。所以对于这种角色应该预定义在系统中。

5. 与业务表单的接口

Fire Workflow 不提供所谓的“表单设计器”，提供表单设计器是越俎代庖；Fire Workflow 也不会对业务的表单本身做任何的处理；Fire Workflow 仅仅在定义文件中记录 Form 类型的 Task 各种表单的 Uri，使得系统在运行时可以找到当前 TaskInstance 的各种表单的 Uri，然后对其进行处理。

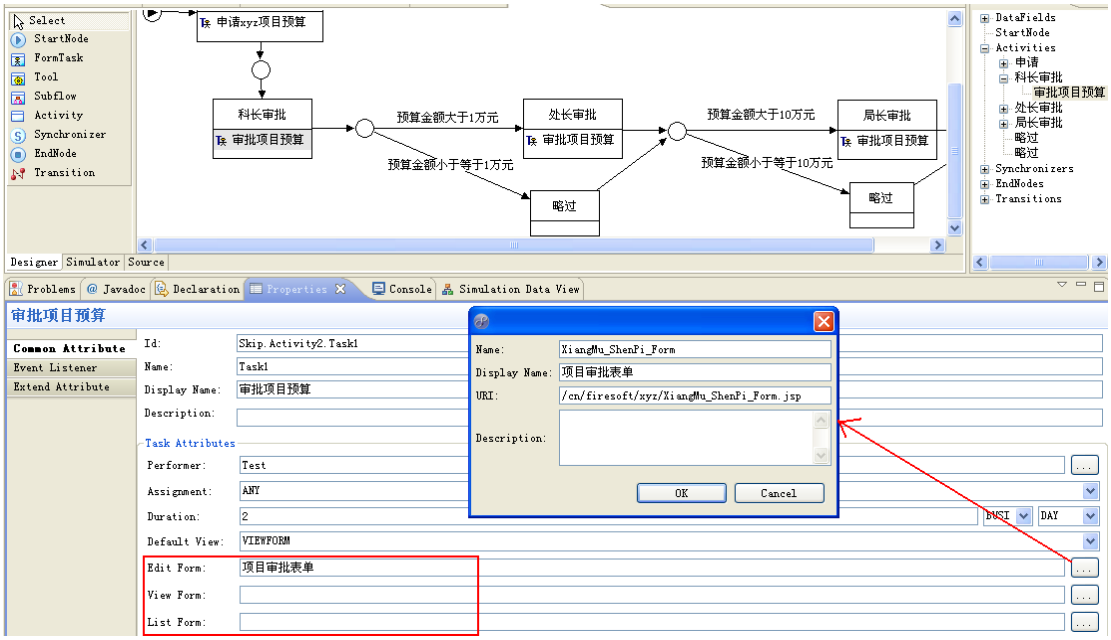
Form 类型的 Task 有如下属性记录表单信息。

属性名称	说明
Default View	缺省视图，取值为 EditForm、ViewForm、ListForm，缺省值为 ViewForm。很多情况下，一个 Task 可能对应多个表单（视图），该属性记录 Task 缺省表单（视图）的类型。 工作流引擎本身不需要使用该属性，仅仅为了方便业务开发而设置。
Edit Form	可编辑的表单信息，在此填入相关的 url 可以用于链接业务表单。该属性是一个复合属性，具体见下一个表格。
View Form	只读表单信息。该属性是一个复合属性，具体见下一个表格。
List Form	列表表单信息。该属性是一个复合属性，具体见下一个表格。

Edit Form，View Form，List Form 都是复合属性，他们各自的属性见下表

属性名称	说明
Name	表单的名称
Display Name	表单的显示名
URI	表单的 URI。工作流系统并不关心该 URI，只要业务系统自己能够该 URI 的含义就可以了。
Description	表单描述。

表单设置界面如下图



在 Fire Workflow Example 中展示了表单的应用。在仓管员、送货员以及系统中所有的操作员看到的“我的待办任务”实际上是同一页面（当然，不同的操作员看到的内容是不一样的），点击这些任务，系统自动导航到各自的处理页面。这种自动导航到相应的处理页面有赖于 Task 中存储的表单 URI 信息。相关的代码可以参考 org.fireflow.example.mbeans.MyWorkItemBean.openForm()。截图如下：

```java
/**
 * 打开工单对应的业务操作界面
 *
 * @return
 */
public String openForm() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    try {
        final String workItemId = this.selectedWorkItemId;
        IWorkflowSession wflsession = workflowRuntimeContext
                .getWorkflowSession();
        IWorkItem wi = wflsession.findWorkItemById(workItemId);

        if (wi != null && wi.getState() == IWorkItem.STARTED) {
            facesContext.getExternalContext().getRequestMap().put(
                    "CURRENT_WORKITEM", wi);
            String formUri = wi.getTaskInstance().getTask().getEditForm()
                    .getUri();
            return formUri;
        } else {
            this.doQueryMyToDoWorkItems();
            return "SELF";
        }
    } catch (EngineException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        this.doQueryMyToDoWorkItems();
        return "SELF";
    }
}
```

6. 退回与取回

在《3_各种工作流模式的实现》一文中讨论了自由流、循环、略过这几种流程模式。常常我们还会听到“退回”和“取回”这两个词，这两个流程操作实际上也没有精确的定义。按照我的理解是这样的。

退回：收到工单的操作员拒绝业务操作，将流程退回到上一个环节。

取回：结束自己工单的操作员发现业务做错了或者其他的缘故，要将下一环节已经产生的工单“强行”取消掉。待修正后重新流转到下一个环节。

总之，退回和取回都是在相邻的两个流程环节之间“折腾”，我感觉这种情况很多时候没有必要“折腾”到工作流里面去，可以在业务代码的层面解决。

首先以取回为例，可以这样规范业务逻辑：在下一个环节没有签收工单之前，上一个环节的操作员有权修改业务数据。就这么简单的一个规则可以避免工作流来回流转的麻烦，更重要的是，很多场合下用户会觉得系统的操作更加方便。

再以退回为例，既然收到工单的操作员发现有问题，那就不签收，通知上一个环节操作员修正之后再签收就可以了。

上述方案是一种解决途径，如果觉得不严密，也可以通过工作流来解决。在 Fire Workflow 看来“退回/取回”都是“循环”的特例。退回可以调用 IWorkItem.loopTo(String nextActivityId),

对于取回目前没有直截了当的方法供调用，（20090210）。

7. 工单签收与业务实际中的材料移交

在 Fire Workflow 中，工单签收的意义首先是对工作任务的“认领”。因为在 Fire Workflow 会把任务会分配给角色中的所有成员，如果不通过签收的方式“认领”工单，可能会造成多个人干同一件事的情况（非会签的情况下）。工单被一个操作员签收后，其他操作员就没有权限处理该单业务了。

工单签收在某种情况下还代表业务材料的交接，即，我签收工单代表纸质材料已经移交到了我的手上。但是这种含义往往与实际情况不太符合，例如：在政府部门的审批业务中，业务材料通常情况下是由一个部门的内勤收集后统一交到下一个部门的内勤手上，材料移交在两个内勤之间发生；这种情况下，接受材料的内勤不能够签收工单，否则业务领导就看不到本应由他处理的工作任务了。在这种情况下，材料移交不能记录在工作流系统中，而是记录在业务系统中。

Fire Workflow 也可以用编码的方式设置工单不需要签收，参见接口 IAssignable.asignToActor(String actorId, boolean needSign)的说明文档。（是否需要给 Task 增加一个属性来设置工单要不要签收还在考虑中，20090210）

8. 流程“自定义”与“自调整”

我已经在 FAQ 中申明了，我反对“流程自定义”这个说法的。尽管很多客户整天嚷嚷着要流程自定义、表单自定义；也尽管很多公司声称他们的“平台”是免编程的，几乎什么都可以自定义。但是，有几个最终用户会自己在一个所谓的平台上定义出自己需要的业务呢？我认为几乎没有。所以，“流程自定义”是一个假议题，顶多是一个商业议题而不是一个技术议题。

但是，最终用户在某种程度上调整业已存在的流程是一个真实的需求。Fire Workflow 考虑提供一个 Web 形式的流程“自调整”工具给最终用户，该工具是流程设计器的变种，可以对系统中已经存在的流程进行某种程度的调整。Web 形式的“流程调整器”采用什么技术实现，允许用户调整那些流程参数有待考虑（20090210）。

9. 工作流系统的性能问题

我曾经在项目中应用过 Jbpm，发现其性能不佳。首先，Jbpm 表结构太复杂，导致每次流程操作都要执行大量 SQL 语句。更严重的问题是，JBPM 的很多表数据增长迅速（具体没有统计，大概是 jbpm_taskactorpool 等），长时间运行后，系统性能肯定有隐患。

以我的经验，工作流系统运行性能主要存在于以下两个方面

➢ 数据库 IO 的数量

在 MIS 系统中，多执行一条不必要的语句对性能都有影响！因为 MIS 系统用户众多，而且每个用户持续不断地办理业务，如果某个业务操作多执行一条不必要的 SQL 语句，那么一个人一天下来可能执行上百条不必要的语句，100 个人一天要执行上万条不必要的 SQL 语句。这么多无用的 SQL 语句对性能的影响是非常明显的。

因此 Fire Workflow 尽量减少数据库 IO。首先，Fire Workflow 简化表结构，目前 Fire Workflow 总共只有 7 张表，除去一张表存储流程定义文件，实际用于引擎计算的表只有 6 张。另一方面，Fire Workflow 尽量优化 SQL 语句，尤其是查询语句，尽量避免 N+1 次查询的情况。

下表根据 Workflow Example 统计了预览版本的引擎中各种流程操作的 SQL 执行情况。

流程操作名称	**SQL 执行情况**
创建流程实例	1 条 Insert 语句（插入 T_FF_RT_ProcessInstance 表）
设置流程实例变量	N 条 Insert 语句（每个变量 1 条）
启动流程实例	1 条 Update 语句
创建任务实例和工单	4 条 Insert 语句 N 条插入 T_FF_RT_WorkItem 的语句
查询待办工单	1 条查询语句（将 T_FF_RT_TaskInstance 、T_FF_RT_WorkItem，T_Biz_MyTaskInstance 中的数据一次性查询出来）
签收工单	2 条 Update 语句， 1 条 Delete 语句
结束工单并启动下一个环节	约 12 条语句 N 条插入 T_FF_RT_WorkItem 的语句
查询已办任务	1 条查询语句（将 T_FF_RT_TaskInstance 、T_FF_RT_WorkItem，T_Biz_MyTaskInstance 中的数据一次性查询出来）

➢ 流程表中的数据量

流程系统各个表的数据量是影响性能的另一个关键因素，尤其影响“待办任务”的查询。Fire Workflow 已经通过初步的优化，将每个表的数据增长速度压缩到最低。

同时，Fire Workflow 正在考虑流程数据分割策略。流程数据有这样一个特点，在业务稳定的情况下，整个系统中活动的流程实例是比较固定的，而且数据量也不大。然而随着运行时间的增长，系统中终止的流程实例会越来越多，数据量会越来越大。这种已经终止的流程实例数据一般情况下不会用到，然而却严重影响数据库操作的速度。因此非常有必要设计一种策略将这部分数据分割出去。Fire Workflow 接下来需要考虑这个问题。