# 、4_将 Fire Workflow 嵌入你自己的系统中

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

目　录

# 1. 阅读指南及注意事项

本文所用示例、Engine 安装包、数据库脚本等等都在 Fire-Workflow-Engine-AllInOne.rar 中。该发布包包含如下内容

/dist 目录：包含 Fireworkflow 的 org-fireflow-engine.jar 和 org-fireflow-model.jar，数据库脚本，FireflowContext.xml。

/FireflowExample：一个使用 Fire workflow 的 j2ee 项目示例

/ThirdPartyLibs：engine 所需的第三方支持包

/api_docs：api 文档 （暂缺 2009-02-05）

/src：引擎和 Model 的源代码

我建议你按照章节顺序阅读可能对 Fire workflow 的理解更加深入，如果一开始就扎进第 6 章会不知所云。

# 2. Fire Workflow 就是两个 jar 包和 7 张表

## 1) Jar 包

如果不考虑流程设计器，那么 Fire Workflow 和 hibernate 或者 spring 一样，都是几个 jar 包而已。唯一的区别是，Fire Workflow 还附带了几张表，用于记录流程流转过程中的一些状态。

Fire Workflow 的 jar 包如下表

| 名称 | 用途 |
|---|---|
| org-fireflow-model.jar | 工作流模型部分，例如 WorkflowProcess, Activity, Task, Synchronizer 等等静态对象都在该包中。 |
| org-fireflow-engine.jar | 工作流引擎部分，例如 ProcessInstance，ActivityInstance，TaskInstance 等等动态对象及其驱动算法都在该包中。 |

Fire Workflow 还需要如下第三方支持包

| 名称 | 用途 |
|---|---|
| spring-core.jar，spring-context.jar，spring-beans.jar | Spring 相关的 jar 包。Fire Workflow 的 Engine 的各种服务是通过 Spring IOC 容器组装在一起的。<br>这是缺省的组装方式，实际上你也可以写一个 Factory 将 Engine 组装起来，或者用其他的 IOC 容器进行组装。 |

| | |
|---|---|
| dom4j.jar | 用于流程定义文件的解析 |
| commons-logging.jar，log4j.jar | 用于日志 |
| commons-jexl-1.1.jar | 用于计算转移条件的 EL 表达式 |
| commons-beanutils-1.6.1.jar | 用于 java Bean 的拷贝 |
| Hibernate 相关的 jar 包 | 缺省情况下，engine 通过 hibernate 进行持久化操作。 |
| | |

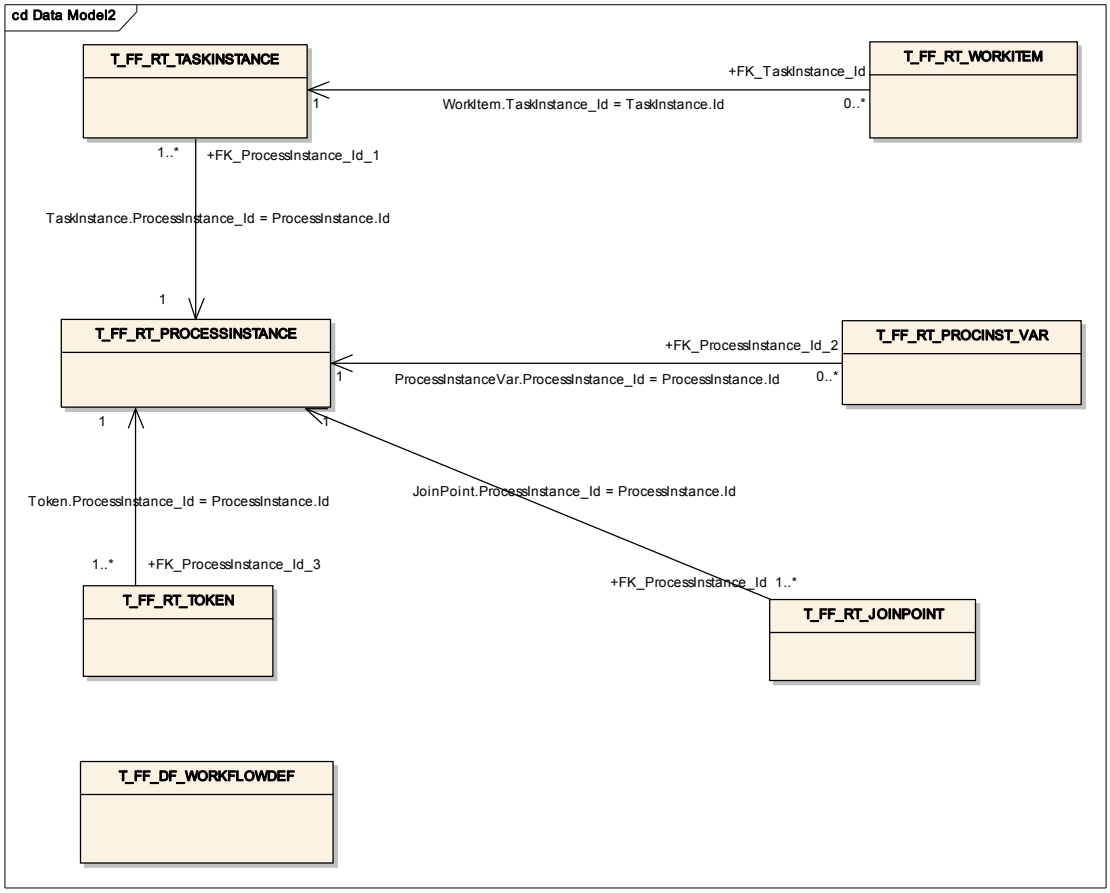
## 2) 表结构

Fire Workflow 的后台表结构如下

| 名称 | 用途 |
|---|---|
| T_FF_DF_WORKFLOWDEF | 存储流程定义文件。在 Fire workflow 中，流程运行时的表不需要和流程定义表做关联，因此没有将流程定义文件分解，直接存为一个大字段。 |
| T_FF_RT_PROCESSINSTANCE | 流程实例表 |
| T_FF_RT_PROCINST_VAR | 流程变量表 |
| T_FF_RT_TASKINSTANCE | 任务实例表 |
| T_FF_RT_WORKITEM | 工单表 |
| T_FF_RT_TOKEN | Token 表，一个流程实例同时可以有任意多个活动的 Token；如果流程实例的活动 Token 数量为 0，即表示流程已经结束了。 |
| T_FF_RT_JOINPOINT | 汇聚点表，用于进行汇聚计算。 |

你或许有点困惑，为什么没有见到 ActivityInstance 之类的表呢？在 Fire Workflow 中，将 StartNodeInstance ， ActivityInstance ， SynchronizerInstance ， EndNodeInstance ，TransitionInstance 等当作内核的对象，内核的对象是所有流程实例共享的无状态的对象，不需要持久化。这样设计的目的之一是减少后台表的数量，相较于 JBPM 的 30 多张后台表，Fire Workflow 是非常苗条的。

Fire workflow 表结构的关系如下图。




- T_FF_RT_PROCESSINSTANCE 表各字段含义如下图

| Name | Type | Nullable | Default | Storage | Comments |
|---|---|---|---|---|---|
| ID | VARCHAR2 (50 CHAR) | ☐ | | | 流程实例Id |
| PROCESS_ID | VARCHAR2 (50 CHAR) | ☐ | | | 流程Id |
| VERSION | NUMBER (10) | ☐ | | | 流程版本 |
| NAME | VARCHAR2 (100 CHAR) | ☑ | | | 流程名称 |
| DISPLAY_NAME | VARCHAR2 (128 CHAR) | ☑ | | | 流程显示名 |
| STATE | NUMBER (10) | ☑ | | | 流程实例状态 |
| CREATED_TIME | TIMESTAMP (6) | ☑ | | | 流程实例创建时间 |
| STARTED_TIME | TIMESTAMP (6) | ☑ | | | 流程实例启动时间 |
| EXPIRED_TIME | TIMESTAMP (6) | ☑ | | | 流程实例完成期限 |
| END_TIME | TIMESTAMP (6) | ☑ | | | 流程实例结束时间 |
| PARENT_PROCESSINSTANCE_ID | VARCHAR2 (255 CHAR) | ☑ | | | 父流程实例Id |
| PARENT_TASKINSTANCE_ID | VARCHAR2 (255 CHAR) | ☑ | | | 对应的父任务实例Id |

- T_FF_RT_TASKINSTANCE 各字段的含义如下图

| Name | Type | Nullable | Default | Storage | Comments |
|---|---|---|---|---|---|
| ID | VARCHAR2(50 CHAR) | ☐ | | | 任务实例Id |
| BIZ_TYPE | VARCHAR2(50 CHAR) | ☐ | | | 业务类别，用于支持hibernate等orm工具进行父子对象映射 |
| TASK_ID | VARCHAR2(50 CHAR) | ☐ | | | 任务Id |
| ACTIVITY_ID | VARCHAR2(50 CHAR) | ☐ | | | 环节Id |
| NAME | VARCHAR2(50 CHAR) | ☐ | | | 名称 |
| DISPLAY_NAME | VARCHAR2(128 CHAR) | ☑ | | | 显示名 |
| STATE | NUMBER(10) | ☑ | | | 任务实例状态 |
| TASK_TYPE | VARCHAR2(15 CHAR) | ☑ | | | 任务类型，取值 FORM, TOOL, SUBFLOW |
| CREATED_TIME | TIMESTAMP(6) | ☐ | | | 任务实例创建时间 |
| STARTED_TIME | TIMESTAMP(6) | ☑ | | | 任务实例启动时间 |
| EXPIRED_TIME | TIMESTAMP(6) | ☑ | | | 任务实例完成期限 |
| END_TIME | TIMESTAMP(6) | ☑ | | | 任务实例结束时间 |
| ASSIGNMENT_STRATEGY | VARCHAR2(255 CHAR) | ☐ | | | 任务分配策略，取值 ALL, ANY |
| PROCESSINSTANCE_ID | VARCHAR2(255 CHAR) | ☐ | | | 流程实例Id（到T_FF_RT_ProcessInstance的外键） |
| PROCESS_ID | VARCHAR2(255 CHAR) | ☐ | | | 流程Id（冗余字段） |
| VERSION | NUMBER(10) | ☐ | | | 流程版本（冗余字段） |

- T_FF_RT_WorkItem 各字段含义如下

| Name | Type | Nullable | Default | Storage | Comments |
|---|---|---|---|---|---|
| ID | VARCHAR2(50 CHAR) | ☐ | | | 工单Id |
| STATE | NUMBER(10) | ☐ | | | 工单状态 |
| CREATED_TIME | TIMESTAMP(6) | ☐ | | | 工单创建时间 |
| SIGNED_TIME | TIMESTAMP(6) | ☑ | | | 工单签收时间 |
| END_TIME | TIMESTAMP(6) | ☑ | | | 工单结束时间 |
| ACTOR_ID | VARCHAR2(50 CHAR) | ☑ | | | 操作员Id |
| COMMENTS | VARCHAR2(1024 CHAR) | ☑ | | | 备注 |
| TASKINSTANCE_ID | VARCHAR2(50 CHAR) | ☐ | | | 任务实例Id（到T_FF_RT_TaskInstance的外键） |

- T_FF_RT_ProcInst_Var 的各字段含义如下

| Name | Type | Nullable | Default | Storage | Comments |
|---|---|---|---|---|---|
| PROCESSINSTANCE_ID | VARCHAR2(50 CHAR) | ☐ | | | 流程实例ID（到T_FF_RT_ProcessInstance的外键） |
| VALUE | VARCHAR2(255 CHAR) | ☑ | | | 实例变量值 |
| NAME | VARCHAR2(255 CHAR) | ☐ | | | 实例变量名称 |

- T_FF_DF_WorkflowDef 各字段含义如下

| Name | Type | Nullable | Default | Storage | Comments |
|---|---|---|---|---|---|
| ID | VARCHAR2(50 CHAR) | ☐ | | | 本记录的Id |
| PROCESS_ID | VARCHAR2(50 CHAR) | ☐ | | | 流程Id |
| NAME | VARCHAR2(100 CHAR) | ☐ | | | 流程名称 |
| DISPLAY_NAME | VARCHAR2(128 CHAR) | ☑ | | | 流程显示名 |
| DESCRIPTION | VARCHAR2(1024 CHAR) | ☑ | | | 流程描述 |
| VERSION | NUMBER(10) | ☐ | | | 流程版本 |
| PROCESS_CONTENT | CLOB | ☑ | | | 流程定义文件 |
| PUBLISHED | NUMBER(1) | ☑ | | | 发布状态 |
| PUBLISHER | VARCHAR2(50 CHAR) | ☑ | | | 发布人 |
| PUBLISH_TIME | TIMESTAMP(6) | ☑ | | | 发布时间 |

## 3) 小结

简而言之，将 Fire Workflow 嵌入自己的系统非常简单，就是将上述 Fire Workflow 的两个 jar 包和第三方支持包扔到/WEB-INF/lib 目录下。然后在数据库端创建 Fire Workflow 的表结构。

如果你的项目是 J2SE 系统，仍然可以使用 Fire workflow，只要将 Fire workflow 的 jar 包和第三方包加入 class path 即可。

本文第 6 章总结了将 Fire workflow 嵌入系统的详细步骤。

# 3. Fire Workflow 的对象

## 4) 工作流模型对象

工作流静态模型对象分成三类：1)Activity 和 Task， 2)Synchronizer、StartNode、EndNode，3)Transition。模型对象在 org-fireflow-model.jar 中。

### a. Activity 和 Task：

Acitivty 在中文里一般称为“活动”，但是在我的文档里习惯称之为“环节”。Task 是任务，代表具体的业务逻辑，如录入一张表单、调用一段 java 代码或者调用另外一个流程（子流程 Task）。一个环节中可以有多个任务。例如某个入职流程中有一个体检环节，体检环节包含了“检查视力”、“检查肝功能”、“检查心血管功能”等等多个任务。在 Fire workflow 中，体检环节建模如下：


### b. Synchronizer、StartNode、EndNode

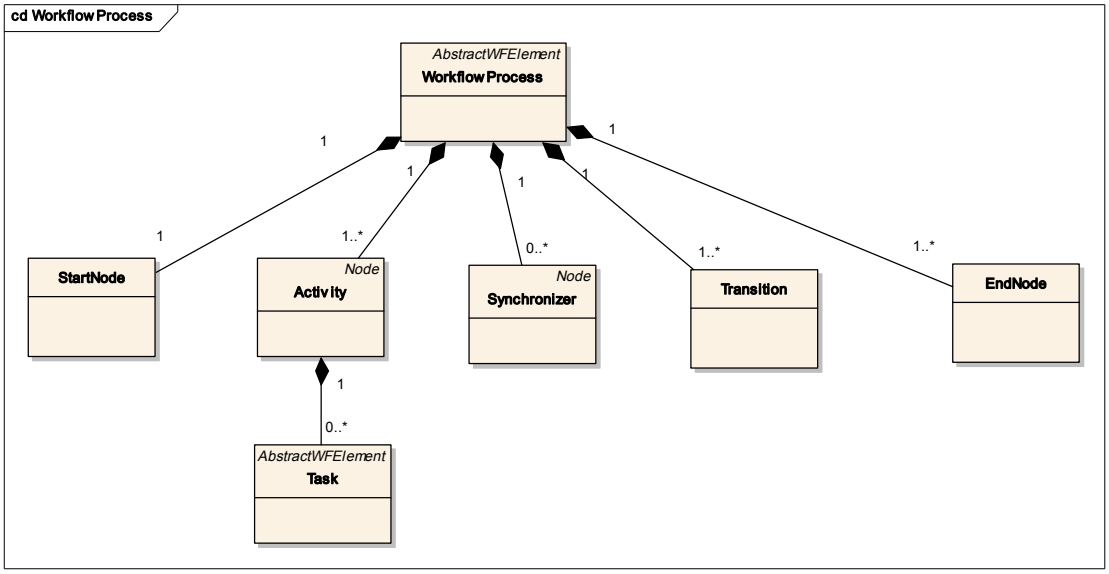
Synchronizer 是同步器。此处的“同步”是一个更加广义的概念，他代表工作流子系统的计算逻辑。开始节点（StartNode）和结束节点（EndNode）是同步器的特例，开始节点是没有输入“边”的同步器，结束节点是没有输出“边”的同步器。

### c. Transition

转移（Transition）在 Fire workflow 并不仅仅是一条连接线，他代表控制权在工作流子系统和业务子系统之间交换。

### d. 各种模型对象（流程元素）的组成关系

各种流程元素的组成关系如下图

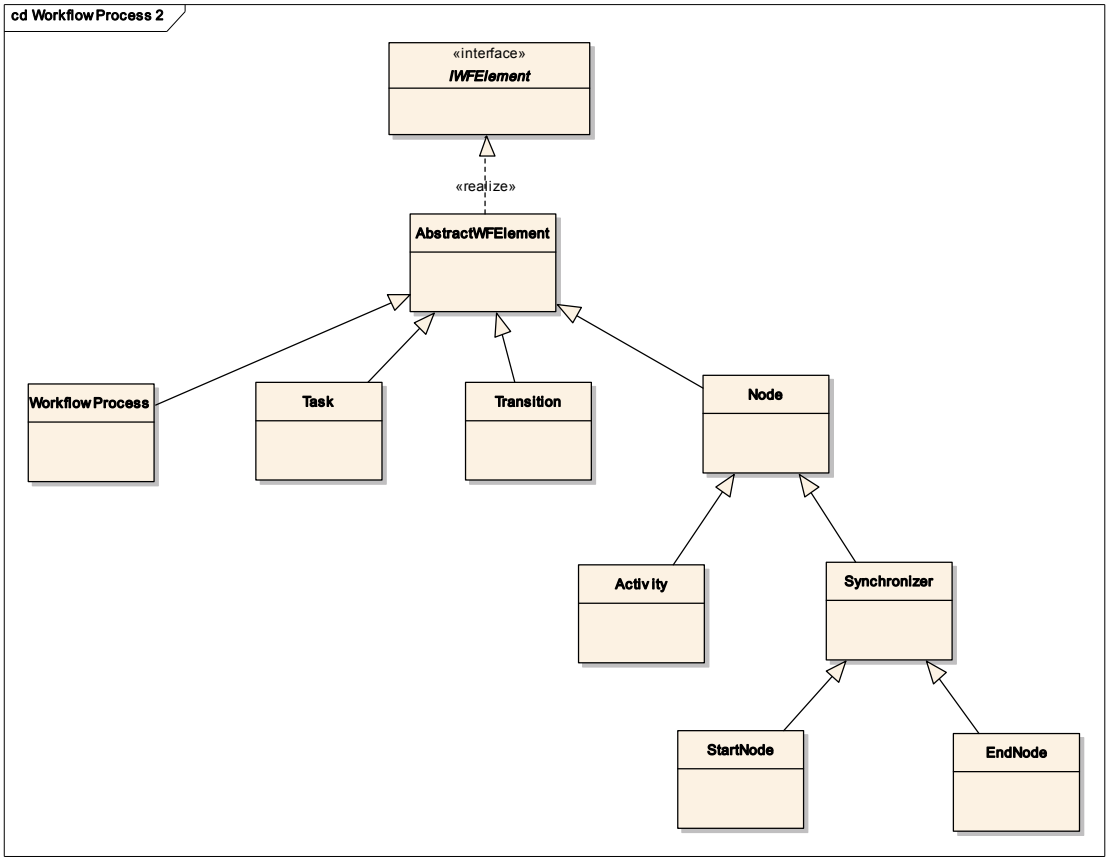



如上图所示，一个流程(WorkflowProcess）有且只有一个开始节点(StartNode)，有一个或者多个环节（Activity)，0 个或者多个同步器(Synchronizer)，一个或者多个结束节点（EndNode)，多个转移(Transition)。

每个环节可以有 0 个或者多个任务（Task)，当环节有 0 个任务时，可以用于表达“略过”的概念（见 《3_各种工作流模式的实现》第“8 略过(skip)”章节)。

## e. 各种流模型对象（程元素）的继承关系

各种流程元素的继承关系如下图




从图中可以看出所有的流程元素皆继承自 AbstractWFElement 抽象类。StartNode 和 EndNode 继承自 Synchronizer，即开始节点和结束节点是特殊的同步器。

# 5) Engine

## a. Engine 的结构

Fire workflow　Engine 的结构如下图。

如图所示，Fire Workflow 把引擎的功能分解成很多的 Service，这些 Service 都“挂接”在 Engine 的“总线”org.fireflow.engine.RuntimeContext 上。下面逐一解释这些 Service。

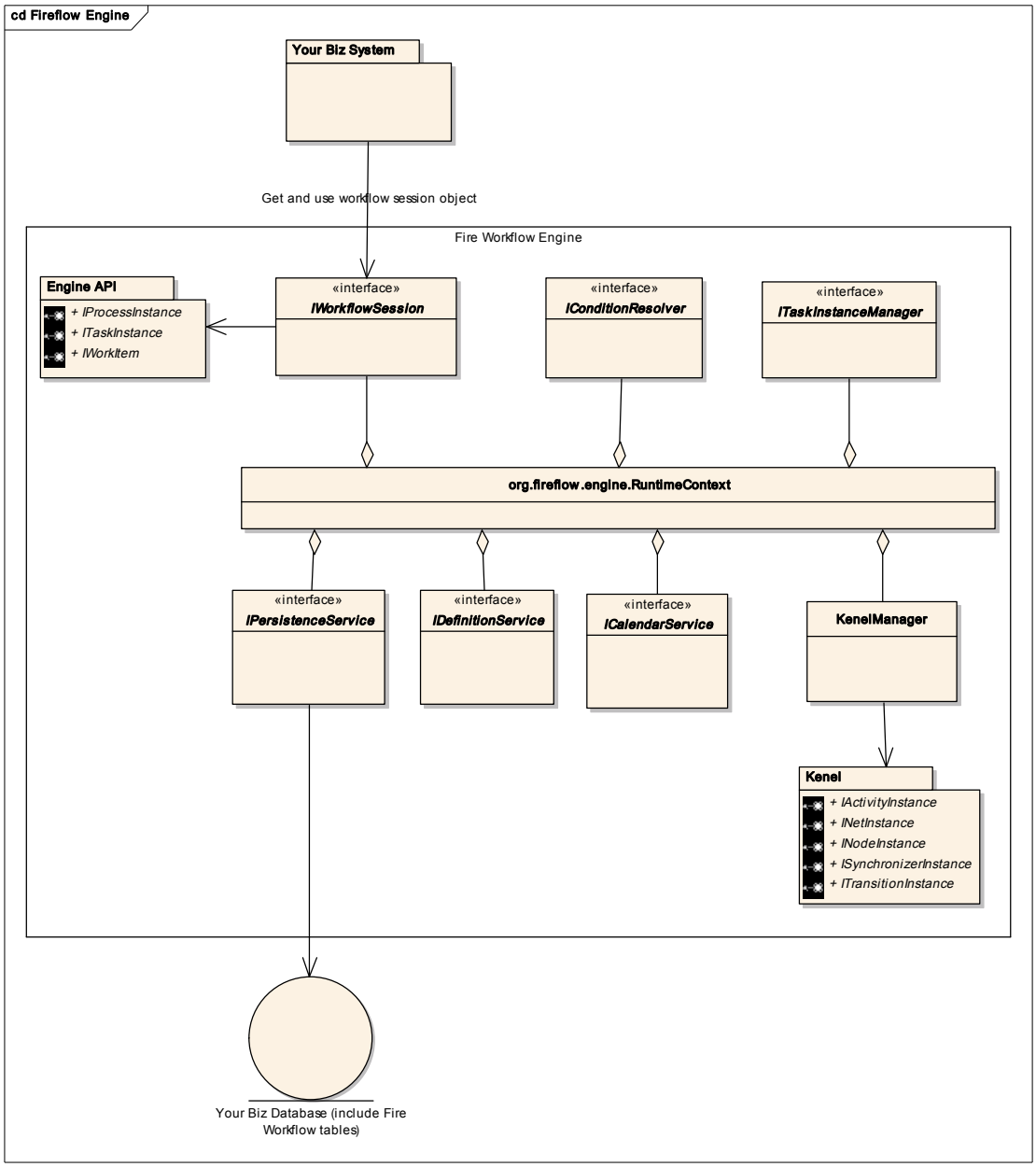
- IWorkflowSession

这是业务代码调用 Engine API 的入口，通过 IWorkflowSession 你可以创建 IProcessInstance，获得相应的 TaskInstance，IWorkItem 等等对象。

显然，此处应用了 Facade 模式，你可以将 IWorkflowSession 类比 Jdbc 中的 connection。通过 Connection 可以获得 Statement，ResultSet 等等对象。

那么在系统中如何取得 IWorkflowSession 对象呢？Fire workflow 缺省情况下是通过 Spring 将各种 Service 组装（即注入）到 RuntimeContext，所以首先要将 FireflowContext.xml 引入到你的系统中，这是一个 spring 配置文件；然后通过类似如下代码获得 IWorkflowSession。具体见第 3、第 4 章节

```
RuntimeContext rtCtx = mySpringBeanFactory.getBean("runtimeContext");
IWorkflowSession workflowSession = rtCtx.getWorkflowSession();
```




- KenelManager

即内核管理器。Fire Workflow 的内核实际上是“工作流逻辑网”的执行机。工作流逻辑网的相关概念见《9_fireflow 技术原理》，它是 Petri Net 改造后的新的网结构。

内核管理器的职责是根据流程定义文件创建和维护工作流逻辑网实例 INetInstance。INetInstance 在 Petri Net 定义的执行规则下一步一步驱动流程实例执行。

- IPersistenceService

存储服务。Fire Workflow 缺省情况下使用 hibernate 进行数据库存取。如果你的系统不是使用 hibernate，则重新实现该类，然后通过修改 FireflowContext.xml 配置，将你的存储服务实现类注入到 RuntimeContext 中。更详细的内容见《6_工作流引擎的结构及其扩展》

- IDefinitionService

流程定义服务。该服务负责根据流程 ID 和版本号获得流程定义对象 WorkflowDefinition。从该对象可以获得 WorkflowProcess，即真正的流程定义。

Fire Workflow 缺省提供两种实现，一种实现是 org.fireflow.engine.definition.DefinitionService4FileSystem。该实现类从文件系统中获得流程定义对象，在开发阶段使用该类比较方便。该类从 class path 中读取流程定义文件，因此你在项目中设计流程时，推荐将流程定义文件置于/src 或者其子目录中。DefinitionService4FileSystem 忽略流程的版本，直接读取当前的流程定义文件。

另一个实现是 org.fireflow.engine.definition.DefinitionService4DBMS。该实现类从数据库表 T_FF_DF_WORKFLOWDEF 中获得流程定义文件。因为表 T_FF_DF_WORKFLOWDEF 中保存了流程的版本号，因此该类在产品真正运行时使用。

在 FireflowContext.xml 修改相关的配置即可实现这两个类的切换。

- ICalendarService

日历服务。日历服务负责获取系统时间和计算 TaskInstance 的 ExpiredDate。缺省实现中，系统时间是返回 new Date()，也只考虑了周六、周日作为节假日的情况。你可以扩展该类获取数据库时间所谓系统时间，增加节假日配置。

- IConditionResolver

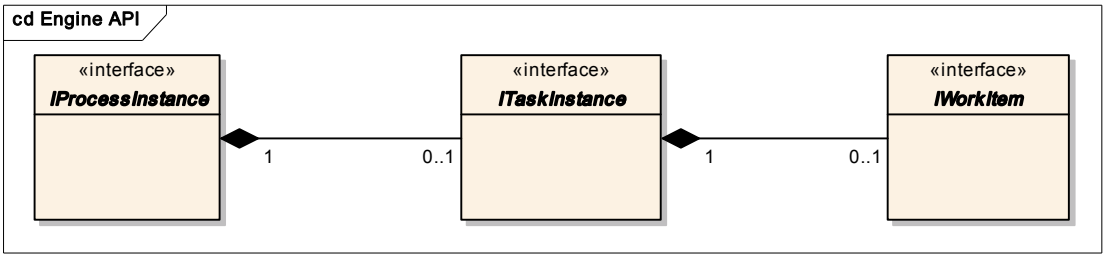
转移条件解析器，用于计算转移条件中的 EL 表达式的值。

- ITaskInstanceManager

任务管理器，负责创建任务实例。缺省实现是 BasicTaskInstanceManager，作为一个真实的应用一般需要扩展这个类。具体原因见《5_工作流应用中经典问题的解决方案》，在此不喧宾夺主地描述了。

## b. Engine API

你的业务代码在调用 Fire Workflow Engine 时需要涉及到的对象如下图



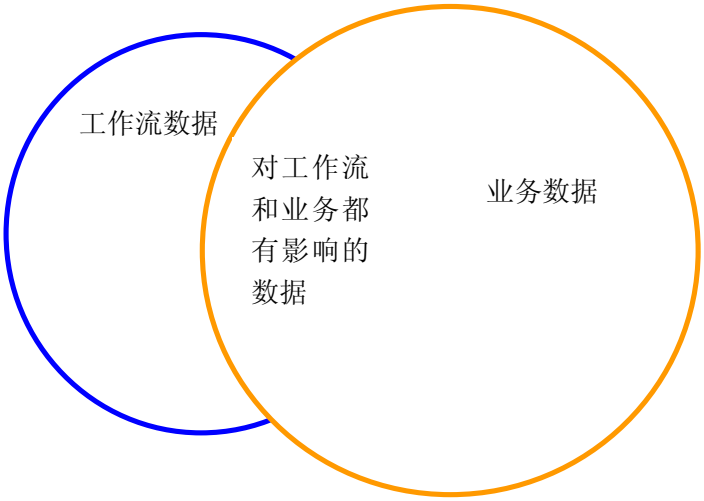

IProcessInstance 是流程实例，包含多个任务实例 ITaskInstance，每个任务实例可以包含 0 个或者多个 IWorkItem。各对象具体的方法说明见 API 文档。

你或许会疑惑，为什么业务代码不需要调用到 ITaskInstance，ISynchronizerInstance，和 ITransitionInstance。在 Fire workflow 中你在一般情况下确实不需要关心这些对象，具体原因我们留在以后慢慢探讨。在 Fire workflow 中，将上述对象划归到 kenel 中，而且设计成无状态的对象，以减少数据库表。

# 4. 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。

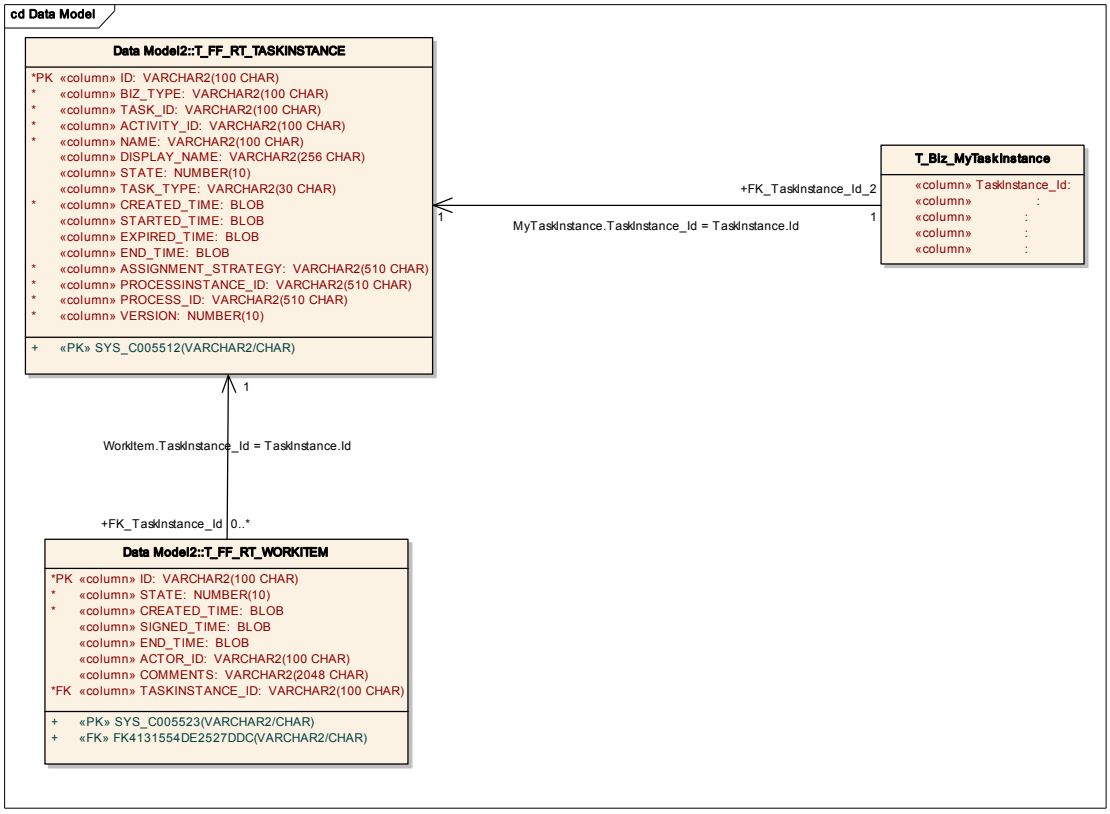
- 方案二、将业务数据当作流程变量存储到工作流系统中

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。

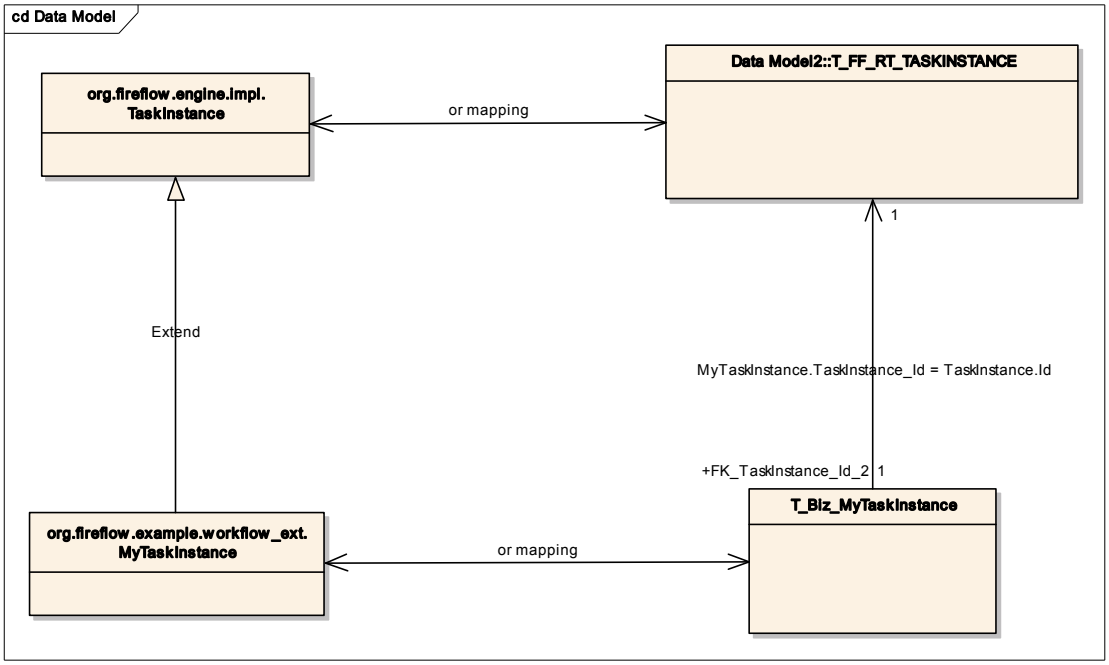
例如：如果你想跟踪某个客户购买的商品现在处于那个环节了，只要联合 T_FF_RT_TaskInstance 和 T_Biz_MyTaskInstance 进行查询即可。而且这两个表的数据是 1 对 1 的，通过 TaskInstanceId 进行关联，性能也不会有大的问题。（当然，性能还是有优化的空间，具体见《5_工作流应用中经典问题的解决方案》）。

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

## 2) 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。

# 5. 常见流程操作在 Fire Workflow 中的实现

我们用下面的案例来认识常见的流程操作。假设有某业务流程如下：受理-->审核-->归档。

首先考察一下受理岗操作员的操作。首先，操作员填写表单，保存数据；同时系统应该给该业务创建新的业务流程实例。为了能够使得业务流程实例和业务数据相关联，需要给业务流程设置流程变量，将业务数据的关键字段设置到流程实例中去。在实际业务中，大多数情况下受理岗操作员一保存数据，流程即流转到审批环节，不需要操作员再做额外的操作。因此，系统应该自动启动并自动完成第一个环节及其工单。总结一下受理岗的流程操作如下表。

| 岗位名称 | 操作员的操作 | | 业务系统自动完成的流程操作 |
|---|---|---|---|
| | 业务操作 | 流程操作 | |
| 受理岗 | 1、填写表单并保存 | 无 | 1、创建流程实例<br>2、设置流程变量<br>3、运行流程实例使之启动“受理”环节<br>4、自动结束“受理”环节， |

| | | | |
|---|---|---|---|
| | | | 实例及其工单，是流程流转到“审核”环节 |

再考察一下审核岗操作员的操作。一般情况下，操作员需要签收待审核的工单，然后填写审核意见，然后保存审核意见。在实际业务中，既可以在保存审核意见的同时，自动结束相关的工单，使得流程流转到下一个环节；也可以，单独提供一个按钮，让操作员点击一下才结束工单。此处假设需要操作员点击一下按钮才结束工单。总结一下审核岗的流程操作如下。

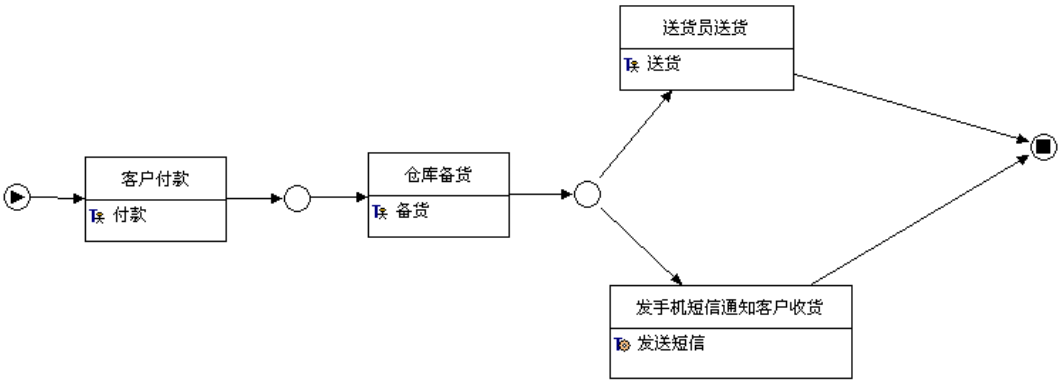
| 岗位名称 | 操作员的操作 | | 业务系统自动完成的流程操作 |
|---|---|---|---|
| | 业务操作 | 流程操作 | |
| 审核岗 | 2、填写审核意见并保存 | 1、签收工单<br>3、点击相关按钮结束工单 | 无 |

最后考察一下归档岗的操作。同审核岗类似，操作员首先签收相关的工单，然后填写必要的业务信息并保存，最后点击按钮结束工单。总结该岗位的流程操作如下。

| 岗位名称 | 操作员的操作 | | 业务系统自动完成的流程操作 |
|---|---|---|---|
| | 业务操作 | 流程操作 | |
| 归档岗 | 2、填写相关业务信息并保存 | 1、签收工单<br>3、点击相关按钮结束工单 | 无 |

总结：由上面的案列可知，除了第一个环节的流程操作稍有不同之外，后续环节的流程操作是雷同的，都是操作员“签收工单-->录入业务数据并保存-->结束工单”。整个流程实例的结束是工作流引擎自动完成的，不需要任何外部操作。

下面我们详细讲解这些常见的业务流程操作如何调用 Fire Workflow 的接口来实现。下面的代码在 FireflowExample 中，这个 Example 模拟某商场送货流程。流程图如下。




## 1) 创建流程实例、设置流程实例变量、自动完成第一个环节的工单

在 FireflowExample 中，收银员收到货款，录入客户、货物、金额等信息即可，流程会自动流转到仓库备货环节。因此业务代码需要将启动流程、设置流程实例变量、自动完成第一个环节这三个流程操作一并完成。相关代码在 org.fireflow.example.mbeans.PaymentBean.doSave()中。如下图。

如图所示，第 6 行保存业务数据。

第 8 行获得 IWorkflowSession 对象，其中 workflowRuntimeContext 是由 JSF 的机制注入的，workflowRuntimeContext 来自 FireflowContext.xml。

第 11 行、12 行通过 IWorkflowSession 的接口创建流程实例。

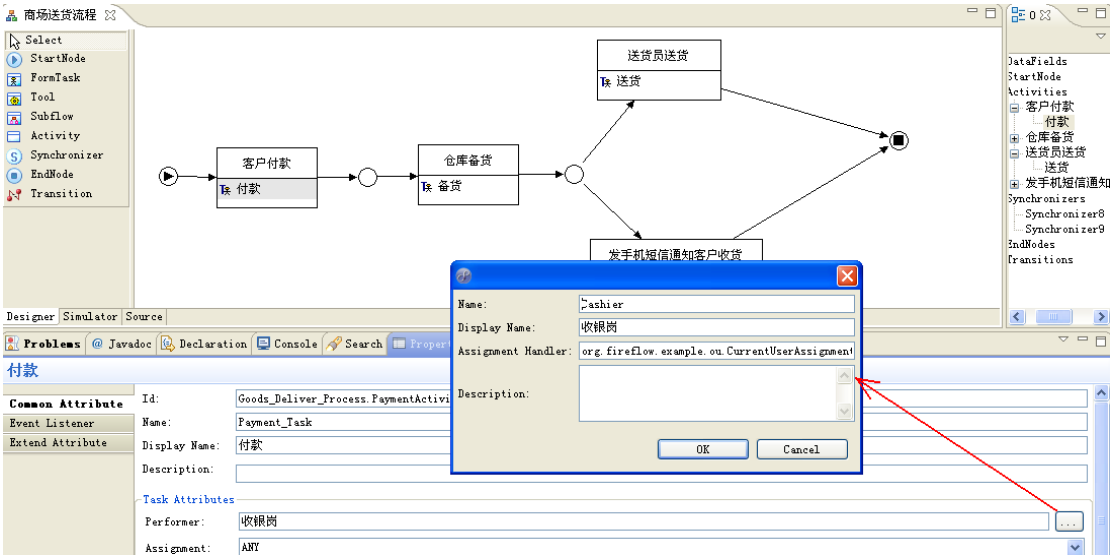
第 14 行至 22 行设置流程实例变量。

第 24 行启动流程实例。

第 26 行至 33 行自动完成当前操作员在当前流程是实例的 WorkItem，即由“客户付款”这个 Task 产生的 WorkItem。

```
1.  public String doSave() {
2.      this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {
3.                  protected void doInTransactionWithoutResult(
4.                          TransactionStatus transactionStatus) {
5.                      //一、执行业务业务操作，保存业务数据
6.                      paymentInfoDao.save(paymentInfo);
7.                      // 二、开始执行流程操作
8.                      IWorkflowSession workflowSession = workflowRuntimeContext.getWorkflowSes
9.                      try {
10.                         // 1、创建流程实例
11.                         IProcessInstance procInst = workflowSession
12.                                 .createProcessInstance("Goods_Deliver_Process");
13.                         // 2、设置流程变量/业务属性字段
14.                         procInst.setProcessInstanceVariable("sn", paymentInfo
15.                                 .getSn());//设置交易顺序号
16.                         procInst.setProcessInstanceVariable("goodsName",
17.                                 paymentInfo.getGoodsName());//货物名称
18.                         procInst.setProcessInstanceVariable("quantity",
19.                                 paymentInfo.getQuantity());//数量
20.                         procInst.setProcessInstanceVariable("mobile",
21.                                 paymentInfo.getCustomerMobile());//客户电话
22.                         procInst.setProcessInstanceVariable("customerName", paymentInfo.getC
23.                         // 3、启动流程实例,run()方法启动实例并创建第一个环节实例、分派任务
24.                         procInst.run();
25.                         // 4、根据业务情况，收银环节应该立即对该环节的workitem执行complete操作
26.                         FacesContext facesContext = FacesContext.getCurrentInstance();
27.                         HttpSession httpSession = (HttpSession)facesContext.getExternalConte
28.                         User currentUser = (User)httpSession.getAttribute("CURRENT_USER");
29.                         List workitems = workflowSession.findMyTodoWorkItems(currentUser.get
30.                         if (workitems!=null && workitems.size()>0){
31.                             IWorkItem wi = (IWorkItem)workitems.get(0);
32.                             wi.complete();
33.                         }
34.                     } catch (EngineException e) {
35.                         transactionStatus.setRollbackOnly();//回滚
36.                         e.printStackTrace();
37.                     } catch (KenelException e) {
38.                         transactionStatus.setRollbackOnly();//回滚
39.                         e.printStackTrace();
40.                     }
41.                     paymentInfo = new PaymentInfo();
42.                 }
43.             });
44.     return "SEFL";
```

你或许有一个疑问，流程实例启动后，第一个环节的 WorkItem 一定会分派给当前操作员吗？如果没有派给当前操作员则第 26 行至 33 行的操作不会成功。通过下面的方法可以保证第一个环节的 WorkItem 会分派给当前操作员。首先付款 Task 的 Performer 属性设置如下图：




由上图可知“付款”任务是由 org.fireflow.example.ou.CurrentUserAssignmentHandler 完成的。我们看一下这个类的代码，如下。从下面的代码第 8 行可以看出，系统将当前用户作为认为实例的操作员，因此自动完成第一个环节的工单可以顺利实现。

```
1.  public class CurrentUserAssignmentHandler implements IAssignmentHandler{
2.
3.      public void assign(IAssignable arg0, String arg1) throws EngineException,KenelException
4.          FacesContext facesContext = FacesContext.getCurrentInstance();
5.          HttpSession httpSession = (HttpSession)facesContext.getExternalContext().getSession(
6.          User currentUser = (User)httpSession.getAttribute("CURRENT_USER");
7.          //将当前用户设置为操作员
8.          arg0.asignToActor(currentUser.getId(),false);
9.      }
10. }
```

## 2) FireWorkflow 的任务分配机制

Fire Workflow 把“获得角色中的所有操作员”这个工作委派给业务系统自行实现，实现的方法就是在业务系统中实现 Fire Workflow 的接口 org.fireflow.engine.ou.IAssignmentHandler。然后把这个实现类在相关的 Task 的 Performer 属性中申明。如上节所述。

## 3) 签收工单

对于仓管岗的操作员和送货岗的操作员，其流程操作都是签收工单、完成工单。我们首先看一下签收工单如何实现。

在业务实际中，仓管员应该可以看到一个“待办任务列表”，如下图。仓管员可以选择其中一个或者多个工单进行签收。


获得待办任务列表的代码参考 org.fireflow.example.mbeans.MyWorkItemBean.doQueryMyToDoWorkItems；如下图。第 15 行获得待办任务。

```
1.  /**
2.   * 查询待办任务
3.   *
4.   * @return
5.   */
6.  public String doQueryMyToDoWorkItems() {
7.      IWorkflowSession wflsession = workflowRuntimeContext
8.              .getWorkflowSession();
9.      FacesContext facesContext = FacesContext.getCurrentInstance();
10.     HttpSession httpSession = (HttpSession) facesContext
11.             .getExternalContext().getSession(true);
12.
13.     User currentUser = (User) httpSession.getAttribute("CURRENT_USER");
14.
15.     myTodoWorkitems = wflsession.findMyTodoWorkItems(currentUser.getId());
16.
17.     return "SELF";
18. }
```

签收相关的代码参考 org.fireflow.example.mbeans.MyWorkItemBean.signWorkItem()；调用 IWorkItem.sign()完成工单的签收。如下图。

```java
/**
 * 签收工单
 *
 * @return
 */
public String signWorkItem() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    Map params = facesContext.getExternalContext().getRequestParameterMap();
    final String workItemId = this.selectedWorkItemId;
    this.transactionTemplate
            .execute(new TransactionCallbackWithoutResult() {
                @Override
                protected void doInTransactionWithoutResult(
                        TransactionStatus transactionStatus) {
                    IWorkflowSession wflsession = workflowRuntimeContext
                            .getWorkflowSession();
                    IWorkItem wi = wflsession.findWorkItemById(workItemId);
                    try {
                        if (wi != null
                                && wi.getState() == IWorkItem.INITIALIZED) {
                            wi.sign();
                        }
                    } catch (EngineException e) {
                        e.printStackTrace();
                    } catch (KenelException e) {
                        e.printStackTrace();
                    }
                }
            });
    doQueryMyToDoWorkItems();
    return "SELF";
}
```

## 4) 完成工单

完成工单的代码参考 org.fireflow.example.mbeans.MyWorkItemBean.completeWorkItem(); 如下图

```java
/**
 * 结束当前的workitem
 *
 * @return
 */
public String completeWorkItem() {
    final String workItemId = selectedWorkItemId;
    this.transactionTemplate
            .execute(new TransactionCallbackWithoutResult() {
                @Override
                protected void doInTransactionWithoutResult(
                        TransactionStatus arg0) {
                    if (workItemId != null) {
                        try {
                            IWorkflowSession wflSession = workflowRuntimeContext
                                    .getWorkflowSession();
                            IWorkItem wi = wflSession
                                    .findWorkItemById(workItemId);
                            if (wi != null
                                    && wi.getState() == IWorkItem.STARTED) {
                                wi.complete();
                            }
                        } catch (EngineException e) {
                            e.printStackTrace();
                            arg0.setRollbackOnly();
                        } catch (KenelException e) {
                            e.printStackTrace();
                            arg0.setRollbackOnly();
                        }
                    }
                }
            });
    this.doQueryMyToDoWorkItems();
    return "MyWorkItemView";
}
```

# 6. 将 **Fire workflow** 嵌入你的系统的详细步骤

## 1) 第一步、将相关的 Jar 包扔到你的项目中去

将第一章节介绍的 Fire workflow　jar 包和第三方 jar 包扔到你的 WEB-INF/lib 目录中去。如果你的系统是 J2SE 系统则扔到你的 Class path 中去。

## 2) 第二步、创建相关的表结构

利用/dist/dbscript 中的数据库脚本创建 Fire workflow 所需的表结构。目前只提供了 oracle 和 mysql 的版本。

## 3) 第三步、配置 FireflowContext.xml

（Fire workflow 缺省使用 spring 进行 Service 装配，如果你的系统不是使用 spring 或者你不熟悉 spring，则解决方案以后再讨论。）

将/dist/FireflowContext.xml 拷贝并引入到你的项目中去。需要进行如下配置。

### a. 首先配置 task instance manager

如第 4 章所述，实际的应用中都需要定制 task instance manager，所以该 task instance manager 必须配置到 FireflowContext 中，替换缺省实现。在 Fireflow Example 中替换如下图。

```
<!--
<bean id="taskInstanceManager"
      class="org.fireflow.engine.taskinstance.BasicTaskInstanceManager">
</bean>
 -->
<bean id="taskInstanceManager"
      class="org.fireflow.example.workflow_ext.MyTaskInstanceManager">
</bean>
```

### b. 然后配置 PersistenceService

Fire workflow 缺省利用 hibernate 实现数据持久化操作，所以需要将你的实际的 SessionFactory Name 告知 Fire workflow。具体方法是设置 persistenceService 的 sessionFactory 属性。Fireflow Example 设置情况如下图

```
    <!-- **************Persistence Begin******************** -->
<bean id="persistenceService"
      class="org.fireflow.engine.persistence.hibernate.PersistenceServiceHibernateImpl">
    <property name="sessionFactory">
      <ref bean="MyHibernateSessionFactory" />
    </property>
</bean>
```

### c. 注册流程定义文件

如第 3 章所述，流程定义服务提供了两个缺省实现，一个实现是从数据库表

T_FF_DF_WORKFLOWDEF 读取流程定义。这个类还没有测试过，所以我们暂时只能用 org.fireflow.engine.definition.DefinitionService4FileSystem。该类要求你指定所用到的流程定义文件在 classpath 中的全名。Fireflow Example 设置情况如下图

```
    <!-- **************workflow Definitions Begin*******  -->
    <bean id="definitionService"
    class="org.fireflow.engine.definition.DefinitionService4FileSystem">
        <property name="definitionFiles">
            <list>
                <value>
                /org/fireflow/example/workflowdefinition/example_workflow.xml
                </value>
            </list>
        </property>
    </bean>
        <!-- **************workflow Definitions End*******  -->
```

## 4) 将工作流的 hibernate 映射文件加入到 hibernate 映射文件列表中

这些映射文件既包括 Fire Workflow Engine 本身的，也包括 MyTaskInstance.hbm.xml，如下图。特别注意：WEB-INF/lib/org-fireflow-engine.jar 名称要与实际情况相匹配。

```
<!-- session factory, 指定了lobHandler -->
<bean id="MyHibernateSessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource">
        <ref bean="MyDataSource" />
    </property>
    <property name="lobHandler">
        <ref bean="oracleLobHandler" />
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.Oracle9Dialect
            </prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
    <property name="mappingJarLocations">
        <list>
            <value>WEB-INF/lib/org-fireflow-engine.jar</value>
        </list>
    </property>
    <property name="mappingResources">
        <list>
            <value>org/fireflow/example/data/PaymentInfo.hbm.xml</value>
            <value>
                org/fireflow/example/workflow_ext/MyTaskInstance.hbm.xml
            </value>
        </list>
    </property>
</bean>
```

## 5) 第五步、编码调用 Fire Workflow

按照第 5 章所描述的方法在业务适当的业务代码中创建流程、签收工单、结束工单。

# 7. 关于 FireflowExample 项目

## 1) 项目介绍

本文所涉及的代码都在 FireflowExample 项目中，该项目模拟了某商场送货流程。您可以用 Eclipse 打开该 Project。

该 Example 所使用的程序框架是 JSF(MyFaces1.5+facelet1.1)+Spring+hibernate。

## 2) 建立 Example 的数据库表

Example 提供的缺省的数据库脚本是 Oracle 系统的，在目录“FireflowExample \dbscript \oracle”中。

- MySQL

如果你使用 MySQL 数据库，则可以用 Fire-Workflow-Engine-All-In-One\dist\dbscript 下面的 schema-export-4-mysql.sql 创建工作流引擎表结构，

示例程序业务表结构你需要根据 hibernage 的映射文件“FireflowExample \src \org \fireflow \example\data\TradeInfo.hbm.xml”和“FireflowExample \src \org\ fireflow \example \workflow_ext \MyTaskInstance.hbm.xml”自行创建。

- SQLSERVER 或者其他数据库

如果你使用 SQL Server 或者其他数据库，则所有的表结构需要你自己根据 hibernate 映射文件创建。（以后会完善）

工作流引擎的映射文件在引擎源代码的如下目录中“org\fireflow\engine\persistence\hibernate”。

示例程序业务表结构创建方法和 MySQL 一样。

## 3) 发布运行

本示例程序在 Tomcat 5.5.x 版本下测试通过。

## 4) 登录系统进行操作

通过如下地址登录系统 http://IP:Port/FireflowExample。登录界面如下，界面上列出了系

统的测试用户，你可以通过这些测试用户登录系统进行操作。


| 测试用户列表 | | | |
|---|---|---|---|
| 角色名称 | 用户名 | 密码 | 中文名 |
| 收银岗 | Cashier1 | 123456 | 收银员1 |
| | Cashier2 | 123456 | 收银员2 |
| 仓管岗 | WarehouseKeeper1 | 123456 | 仓管员1 |
| | WarehouseKeeper2 | 123456 | 仓管员2 |
| 送货岗 | Deliveryman1 | 123456 | 送货员1 |
| | Deliveryman2 | 123456 | 送货员2 |
| | Deliveryman3 | 123456 | 送货员3 |
| 管理岗 | manager | 123456 | 管理员 |

## 5) 各角色的流程操作特点

收银员作为流程的第一个岗位，收银完成后，流程即流转到备货环节，因此收银员的“我的待办任务”总是空的。

除了第一个环节的岗位有“收银”菜单作为业务入口外，其他后续岗位理论上都以“我的待办任务”作为业务入口。因此，要求“待办箱”能够“智能地”打开工单对应业务操作界面，代码“org.fireflow.example.mbeans.MyWorkItemBean.openForm()”向你展示如何办到这一点，如下图

```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";
    }
}
```

另外，你也可以关注一下 org.fireflow.example.mbeans.MyWorkItemBean.doQueryMyHaveDoneWorkItems()，该方法想你展示如何通过 IWorkflowSession 实现系统没有预先定义的一些功能。