InforSuite Flow

使用手册

山东中创软件商用中间件股份有限公司

2014 年 08 月

前言

感谢您选用 InforSuite Flow 工作流产品。

InforSuite Flow（以下简称 IS-Flow）是一个基于 J2EE 体系结构并参考国际工作流管理规范开发的工作流产品。IS-Flow 由

- 工作流引擎
- 流程设计器
- 表单设计器
- 管理监控工具
- 标准任务表处理器
- 消息提醒工具
- 产品示例

七部分组成。其中，

- **工作流引擎**对外发布工作流引擎服务并提供一组综合接口，用户可以利用这组接口对要实现的应用流程进行创建、启动、管理与控制。
- **流程设计器**提供图形化的界面进行流程建模，用户可以方便直观地定义所需要的流程。
- **表单设计器**提供图形化的界面对业务模型进行建模，用户可以快速构建应用系统。
- **管理监控工具**基于 B/S 和 C/S 两种结构、对流程进行管理监控。用户可以实时对流程进行查询与控制。
- **标准任务表处理器**提供了 B/S 结构的流程运行环境，用户可以快速开发与工作流相结合的 Web 应用。
- **消息提醒工具**用来辅助当前用户获取流程节点的任务状态。
- **产品示例**提供了用户编程的示例代码。

本手册是针对使用 IS-Flow 工作流产品进行流程应用开发的技术人员编写的。我们假定读者掌握 Java 语言，了解 J2EE 并掌握面向对象的编程技术。

本手册分为 9 个部分：

第 1 章是 IS-Flow 产品介绍，对 IS-Flow 的产品组成及功能作了简要描述。

第 2 章至第 6 章详细介绍了 IS-Flow 产品的功能。其中，

第 2 章 IS-Flow 工作流引擎服务，包括：

- 工作流引擎服务的运行模式，即如何在应用系统中启动工作流引擎服务。
- 工作流引擎服务的类别，即 local、rmi、webservice 方式。
- 工作流引擎服务的事务处理方式。
- 工作流引擎服务的工具代理使用原理。
- 工作流引擎服务的消息提醒功能。
- 工作流引擎服务的事件监听机制。
- 工作流引擎服务的定时启动功能。
- 工作流引擎服务的执行期限功能。

第 3 章 IS-Flow 业务流程建模，介绍建模原理与流程定义导入导出工具的使用。

第 4 章 IS-Flow 流程部署运行，介绍 B/S、C/S 及命令行方式的流程部署和标准任务表处理器中运行流程。

第 5 章介绍 IS-Flow 流程管理监控，介绍 B/S、C/S 方式下对流程的管理监控。

第 6 章 IS-Flow 流程数据分析，介绍了 5 种统计分析模板的原理及分析结果示例。

第 7 章介绍 IS-Flow 用户编程。分为初级编程和高级编程，其中初级编程介绍常用的功能实现，高级编程介绍可以由用户扩展的功能实现。

第 8 章介绍 IS-Flow 常见的工作流应用策略。

第 9 章附录，包括

- IS-Flow 常量表。
- Hibernate 数据库配置常量表。
- IS-Flow 配置文件。
- IS-Flow 工作流优势。
- IS-Flow 工作流特点。

本手册的相关文档：

- 《InforSuite Flow 流程设计器使用手册》
- 《InforSuite Flow 表单设计器使用手册》
- 《InforSuite Flow 管理监控工具使用手册》
- 《InforSuite Flow 流程仿真工具使用手册》
- 《InforSuite Flow 消息提醒工具使用手册》
- 《InforSuite Flow 标准任务表处理器使用手册》
- 《WebService 配置及.NET 平台下 InforSuite Flow 的使用手册》
- 《HelloWorld 使用手册》

使用约定：

%INFORSUITE_HOME%：产品安装目录，如 C:\CVICSE\InforSuite。

山东中创软件商用中间件股份有限公司为 IS-Flow 产品提供全面的技术支持，用户可以通过以下方式获得技术支持：

- 地址:山东中创软件商用中间件股份有限公司(山东省济南市千佛山东路 41-1 号)
- 邮编：250014
- 电话：400-618-6180
- 传真：0531-81753669
- E-mail：support@cvicse.com

- 网址：http://www.inforbus.com

当您需要与 IS-Flow 有关的技术支持时，请提供以下资料：

- 运行环境
- IS-Flow 产品的版本号
- 问题的详细描述

目录

第1章 IS-Flow 产品介绍

本章主要介绍了 IS-Flow 的产品组成和产品功能。

学习本章内容后，您可以从总体上了解 IS-Flow 产品的各个组成部分以及 IS-Flow 产品的所有功能。

1.1. IS-Flow 产品简介

现代社会中，人们对所从事工作的分工越来越细，完成一项工作已经不再是某一个人的事，需要多人之间的协作，以及通过对工作过程的合理组织，才能高效的完成工作。工作流正是为了解决多人之间的组织协作问题而出现的一种新技术，它利用计算机建立业务过程模型，将不同的人、不同的任务组织起来，并控制业务的执行顺序以及任务在不同的人之间的分配。

那么什么是工作流呢？依据工作流管理联盟（Workflow Management Coalition，WfMC）的定义：工作流是指整体或部分业务过程在计算机支持下的自动化。

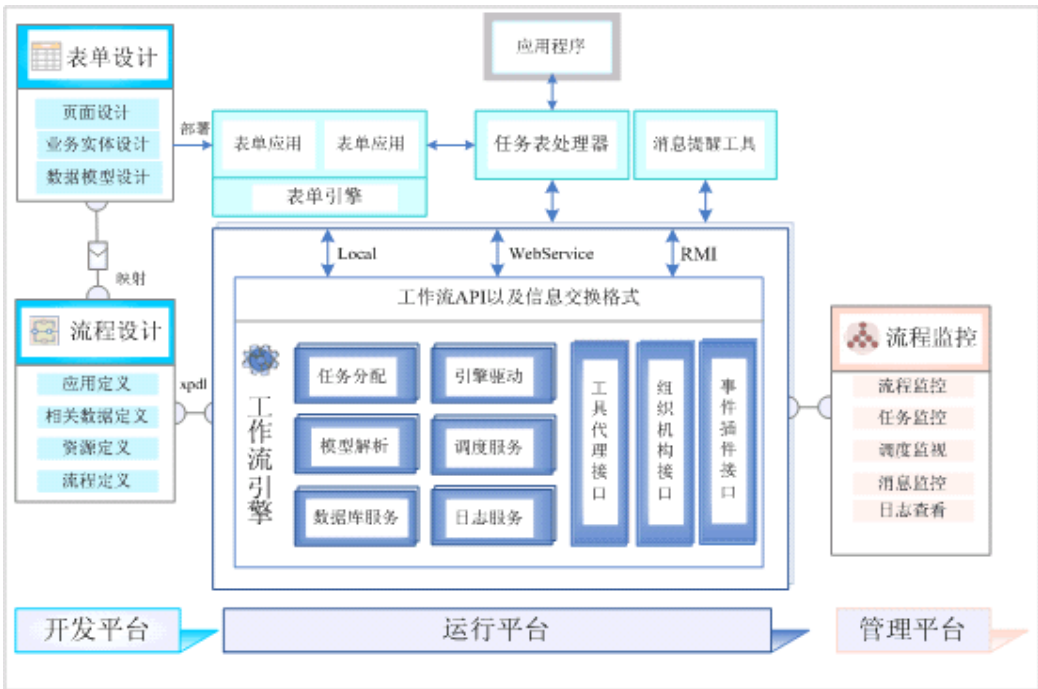
IS-Flow 是中创软件商用中间件有限公司开发的一种工作流中间件产品。它是参考国际工作流管理规范实现的工作流中间件，为工作流自动化和流程再造提供基础平台。IS-Flow 实现了流程逻辑与业务逻辑的分离，能够可视化的进行业务流程的分析、定义和业务单元的组装，从而使应用开发人员更关注于业务逻辑的实现，降低了复杂流程应用的开发难度。

IS-Flow 把业务中的流程控制逻辑从业务逻辑中分离出来，这种分离为应用系统带来了多方面的好处。首先，在开发时，由于对业务过程的控制可以交给 IS-Flow 来做，使得应用开发人员可以专注于业务逻辑的实现，减轻了应用开发的工作量，也使得快速构建应用系统成为可能。其次，在系统运行时，当流程逻辑发生变化时，可以通过 IS-Flow 流程设计器重新定义流程以适应变化的业务过程，使得无需编程而快速

适应流程逻辑的变化成为可能。

1.2. IS-Flow 产品组成

IS-Flow 工作流管理系统由包含多个工作流引擎的工作流执行服务及客户端工具组成。工作流引擎以 WAPI（Workflow Application Programmer Interface，工作流应用程序接口）的形式对外提供流程控制服务。WfMC 规范定义了五类接口来对工作流引擎的行为或者说对外提供的服务进行描述。工作流客户端工具及工作流应用系统使用这些接口并通过工作流执行服务与工作流引擎进行交互。IS-Flow 系统框架见图表 1-1 IS-Flow 产品系统框架：



图表 1-1 IS-Flow 产品系统框架

1.2.1. 工作流引擎

工作流引擎负责解释流程定义、控制流程实例、安排活动的执行顺序、向用户工作表中添加工作项目。工作流引擎维护内部控制数据，这些工作流控制数据包括与各种过程、或者正执行的活动实例相关的内部状态信息。

流程定义数据与（运行时期）工作流相关数据协作，一同用来控制过程中活动的导航、提供活动的转移条件、不同活动的并行执行、顺序执行选项、用户任务、与任务相关的应用入口参数。如果流程定义包括组织模型/角色实体类型，那么完成以上任务，需要访问组织/角色模型数据。

1.2.2. 流程设计器

流程设计器的主要功能如下：

- 以图形化的形式描述业务过程。
- 将定义信息保存到 XPDL 文件中。
- 遵循 WfMC 规范进行流程及相关对象的定义。
- 流程定义的版本控制。
- 流程定义的有效性检查。

流程定义是用来创建一个计算机可以处理的形式的过程描述。应用开发人员可以使用 IS-Flow 所提供的流程设计器来方便的定义一个业务流程，IS-Flow 的流程定义遵循工作流管理联盟的 XPDL 规范。

1.2.3. 表单设计器

表单设计器的主要功能如下：

- 业务实体编辑器提供定制业务实体功能，包括定义业务实体的属性及业务实体之间的关联关系等功能；使用定制好的业务实体文件，可以直接构建应用的代码，并可以从业务实体文件生成数据表文件。

- 数据表编辑器提供定制数据表功能，包括定义表字段的数据类型、在数据库中创建表，并可以从数据表文件生成业务实体文件。
- 表单编辑器用来快速完成表单的定制；表单编辑器提供了图形化的设计界面，提供了文本框等表单设计常用的基本组件。
- 业务人员定制的表单可以与工作流系统相结合，为完成二者的契合，表单设计器提供了相应功能，如设置应用完成监听器、获取工作流相关数据等。

业务人员使用 IS-Flow 所提供的表单设计器，可以快速定制表单，并可将开发出的表单与工作流应用相结合。

1.2.4. 管理监控工具

流程管理监控工具的主要功能如下：

- 流程定义管理。
- 流程管理：查询流程实例与历史流程。
- 流程实例控制，包括挂起、恢复、终止和取消流程实例。
- 活动管理：查询活动实例与历史活动。
- 活动实例控制，包括挂起、恢复、终止和取消活动实例。
- 工作项管理：查询工作项实例与历史工作项。
- 工作项实例控制，包括挂起、恢复、终止和取消工作项实例。
- 用户权限管理。

IS-Flow 所提供的流程管理监控工具，可以使得管理人员能够掌握任意流程实例当前所处的状态和处理情况，能够尽早的发现和解决一些在流程运行中存在的异常现象。

1.2.5. 标准任务表处理器

标准任务表处理器（tasklist）是为用户提供的标准工作流应用。用户使用标准任务表处理器，可以创建流程、启动流程，还可以在线查询和管理流程的相关信息，并可以为用户开发的业务应用提供基本操作，比如删除流程、改变工作项的状态等操作。

标准任务表处理器内置了一个由 InforWeb 提供的单点登录组件。用户登录到任务表处理器后，无需重新登录，即可访问同一 Web 服务器下的其他应用系统。借助单点登录组件实现的功能，任务表处理器可以与其它应用系统交互信息。

用户开发应用系统时，可以直接使用任务表处理器提供的基本操作，从而用户可以专注于业务开发，不再需要过多地关注流程逻辑。

1.2.6. 消息提醒工具

消息提醒工具辅助用户获取当前任务，当有新任务到达时，消息提醒工具会弹出消息提醒用户，并播放预先设置好的提醒铃声。

消息提醒工具由消息服务器和消息客户端工具两部分组成。如果引擎需要发送消息，发送消息前必须启动消息提醒服务，消息提醒客户端通过消息提醒服务注册并接收消息。

启动 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\bin 下的 startup.bat，即启动 InforSuite 应用服务器后，消息提醒服务随即启动。

启动消息提醒服务后，客户端工具可以连接此服务，客户端工具 message.bat 存放在 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\infor\flow\messenger 下，打开 message.bat 文件便启动了消息提醒客户端工具。

1.2.7. 产品示例

IS-Flow 安装包中提供了两个产品示例，位于 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\helloworld 下。HelloWorld 的示例提供了多个展示产品对不同特点的流程定义支持的 XPDL 文件，用户可方便的运行这个示例。另一个示例是用于管理监控的工具，开发人员可以使用这个工具对开发过程中的流程进行调试。这两个示例都提供了源代码供开发人员参考使用。

1.3. IS-Flow 产品功能

1.3.1. 工作流引擎服务

工作流引擎服务是工作流引擎对外提供的服务，它是 IS-Flow 产品功能的核心组成部分。有了工作流引擎服务，才可以进行流程建模、流程部署运行、流程管理监控、流程数据分析等。

目前工作流引擎对外提供三种服务，即 local（本地服务）、rmi（远程方法调用）、webservice（web 服务），同时提供了相应的客户端以及灵活的配置来满足实际应用系统的需求。

关于工作流引擎服务的详细介绍参见 IS-Flow 工作流执行服务。

1.3.2. 业务流程建模

流程建模能力的强弱是工作流产品区别于普通办公自动化系统的因素之一。企业中所存在的业务流程是企业生产、经营过程的反映，必然涉及多部门、多角色的人员之间的分工协作，有些业务流程的运行甚至是跨级别、跨地域、跨季度的在时间与空间上都跨度极广的复杂过程。若支持这样的业务流程，必然要求工作流产品具有极强的流程建模能力。

关于业务流程建模的详细介绍参见 IS-Flow 业务流程建模。

1.3.3. 流程部署运行

流程部署的过程就是将流程建模的结果（即流程定义）导入数据库中。IS-Flow 提供 B/S 和 C/S 及命令行三种方式部署流程定义，其中 IS-Flow 管理平台通过 B/S 方式部署流程定义，IS-Flow 开发平台（即流程设计器）通过 C/S 方式部署流程定义，流程定义导入导出工具通过命令行方式部署流程定义。关于流程部署的详细介绍参见流程部署。

流程运行时期要控制流程的流转，即流程如何一步步按照建模时的逻辑向下传

递。这种规则一般来说是按流程定义的顺序关系，但是很多情况下也会发生例外。例如在实际情况中可能存在流程中当前任务的执行人直接指定下一步要提交的目标活动。IS-Flow 所提供的流程控制功能允许流程在运行过程中打破流程定义的约束而直接创建某种活动。表格 1-1 至 1-4 列出了 IS-Flow 所支持的各种流程控制操作及其功能。

表格 1-1 对流程实例的控制性操作

序号	控制操作	功能	前提
1	创建	创建一个新的流程实例	指定创建的流程定义处于“发布”状态
2	启动	启动刚被创建的流程实例	流程实例处于“not_started”状态
3	终止	终止流程实例	无
4	取消	取消流程实例	无
5	挂起	将处于“running”状态的流程实例转入“suspended”状态	流程实例处于“running”状态
6	恢复	将处于“suspended”状态的流程实例转入“running”状态	流程实例处于“suspended”状态

表格 1-2 对活动实例的控制性操作

序号	控制操作	功能	前提
1	提交 (commit)	1. 结束当前活动实例； 2. 按流程定义创建符合转移条件的所有下一步节点。	所有必做的业务单元处于结束状态
2	跳转	1. 结束当前活动实例； 2. 创建指定的目标的活动	所有必做的业务单元处于结束状态
3	回退	1. 结束当前活动实例； 2. 按指定的活动实例创建其所对应的活动，并将活动的参与者直接分配给指定活动实例的执行人。	无
4	放回	1. 结束当前活动实例； 2. 重新创建相同定义的活动。	活动的任务分配策略为由唯一执行人执行
5	终止	1. 结束当前活动实例； 2. 按流程定义创建符合转移条件的所有下一步节点。	无
6	取消	1. 结束当前活动实例；	无
7	挂起	1. 将当前活动实例的状态由“running”状态设置为“suspended”	活动实例的初始状态为“running”

		状态。	
8	恢复	1. 将当前活动实例的状态由“suspended”状态设置为挂起前的状态。	活动实例的初始状态为“suspended”

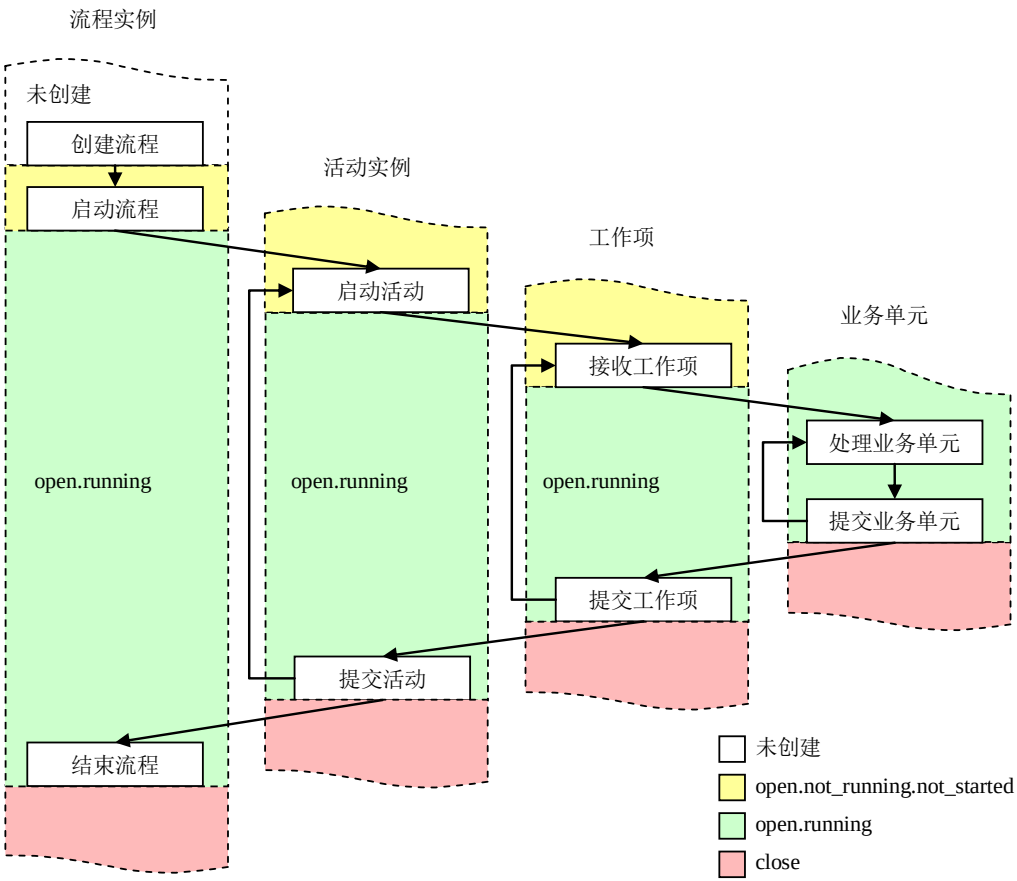
表格 1-3 对工作项的控制性操作

序号	控制操作	功能	前提
1	接收	对任务分配策略为“唯一执行人”的活动，将其活动实例及实例所对应工作项状态设置为“running”。同时，其他人不能再执行接收操作	工作项处于“not_started”状态
2	提交	1. 结束工作项 2. 若工作项所属活动实例的所有工作项都已结束，则结束当前活动实例，创建下一活动实例	所有必做的业务单元都处理完成
3	重定向任务	将已经分配执行人的工作项重新分配执行人	工作项已经被执行人接收

表格 1-4 对应用程序实例的控制性操作

序号	控制操作	功能	前提
1	启动	启动应用程序实例	无
2	结束	提交应用程序实例，状态置为“completed”	无

当对工作流实例对象执行某种控制性操作时，工作流对象实例的状态就会发生改变。每个工作流对象都有从创建到最后被关闭的一个过程，我们称这个过程为工作流对象的生命周期。工作流对象的生命周期见图表 1-2 工作流对象的生命周期：



图表 1-2 工作流对象的生命周期

标准任务表处理器作为 IS-Flow 的辅助工具，提供了业务流程的基本运行环境，它本身是一个 web 应用，可以部署到任何应用服务器上运行。用户使用标准任务表处理器，可以创建流程、启动流程，还可以在线查询和管理流程的相关信息，并可以为

其它应用提供基本操作，比如执行“删除流程”、“改变工作项的状态”等操作。

关于流程部署运行的详细介绍参见 IS-Flow 流程部署运行。

1.3.4. 流程管理监控

IS-Flow 管理监控工具实现了对 IS-Flow 流程的管理监控功能，它可以对正在运行中的流程实例、活动实例、工作项以及在运行中产生的数据进行查询与控制，也可以对历史流程进行查询。使得管理人员能够掌握任意流程实例当前所处的状态和处理情况，能够尽早的发现和解决一些在流程运行中存在的异常现象。例如当发现某个流程由于定义或运行时逻辑上的问题不能继续向下流转时，管理员可以人为的终止流程实例的运行。

目前 IS-Flow 管理监控工具分为 B/S 和 C/S 两种。

关于流程管理监控的详细介绍参见 IS-Flow 流程管理监控。

1.3.5. 流程数据分析

IS-Flow 流程分析功能基于 InforSuite 产品家族中的 InforSuite Report 所提供的报表工具实现。基于 InforSuite Report 可以快速方便的实现各种类型的分析报表的设计、部署、生成、展现、打印和管理。InforSuite Report 支持丰富的报表类型，支持扩展、交叉、主从、主从细、分栏、主从细同时分栏、图表等多样的报表类型，其中图表又包括：

IS-Flow 基于 InforSuite Report 可以快速的定制开发用户所需要的各种类型的流程分析报表，并提供了报表的快速发布功能，以方便流程分析人员的使用。

IS-Flow 提供了丰富的流程分析报表，包括工作完成时间《工作负荷量统计分析报表》、《单一任务超时统计报表》、《任务完成时间统计报表》、《流程执行工作量分析报表》、《流程实例状态统计报表》等。

关于流程数据分析的详细介绍参见 IS-Flow 流程数据分析。

1.3.6. 业务模型设计

表单设计器提供了对应用系统进行快速建模的功能。使用表单设计器，可以根据业务需求快速开发应用系统。

表单设计器全面实现了构建一个应用系统所要跨越的表现层、业务层、持久层三个层次的全部零编程实现的功能，为这三个层次都提供了可视化的建模工具。因此，表单设计器基本的组成部分包括了分别实现这三层可视化建模功能的建模工具。另外，为方便使用，表单设计器也提供了打包部署表单应用的工具。

关于表单设计器的详细介绍参见***《InforSuite Flow 表单设计器使用手册》***。

第2章 IS-Flow 工作流引擎服务

本章详细介绍了 IS-Flow 的工作流引擎服务，包括以下内容：

- 应用系统集成：介绍如何将工作流引擎服务集成到应用系统中，包括工作流引擎服务的运行模式以及工作流引擎服务如何集成应用系统的用户数据。
- 服务类型：介绍工作流引擎提供的三种工作流引擎服务，即 local、rmi 和 webservice 服务。
- 事务处理：介绍工作流引擎的四种事务处理方式，即无事务、独立事务、传递事务和 JTA 事务。
- 工具代理：介绍工作流引擎中工具代理的机制。
- 消息提醒：介绍工作流引擎服务中的消息提醒功能。
- 事件监听：介绍工作流引擎的事件监听机制。
- 定时启动：介绍工作流引擎的流程定时启动功能。
- 执行期限：介绍工作流引擎的执行期限功能。

学习本章内容后，您可以

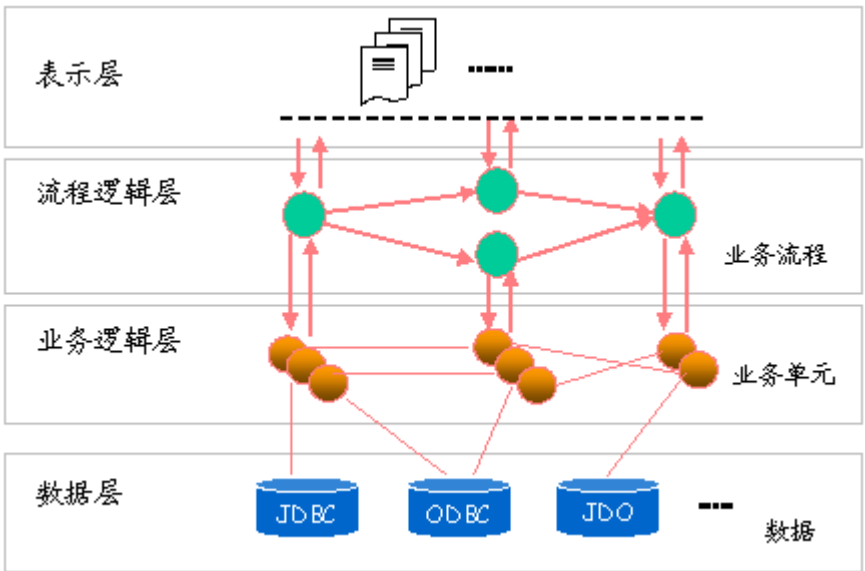
- 将工作流引擎服务集成到应用系统中，即在应用系统中使用工作流引擎服务，以及将应用系统的用户数据集成到工作流引擎服务中。
- 全面了解工作流引擎服务提供的强大功能，如事务处理、工具代理、消息提醒、事件监听、定时启动和执行期限等。

2.1. 将工作流引擎服务集成到应用系统中

2.1.1. 基于 IS-Flow 的应用系统架构

采用 IS-Flow 工作流中间件的系统架构的基本思想在于将业务系统进行充分的分解，逻辑上分解为表示逻辑、流程逻辑、业务逻辑、数据管理逻辑四种基本逻辑，它

们分别对应应用系统的表示层、流程逻辑层、业务逻辑层和数据层，见图表 2-1 基于 IS-Flow 的应用系统逻辑分层。通过这样的分解，使其中任何一层逻辑的修改都不会影响其它三层，从而最大限度的降低系统内部的耦合性，提高系统适应变化的能力。



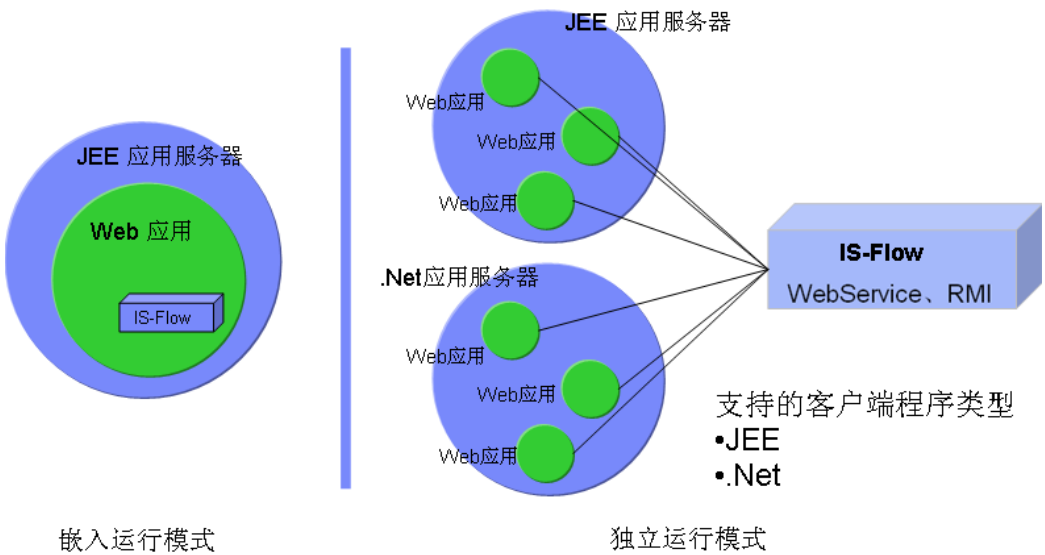
图表 2-1 基于 IS-Flow 的应用系统逻辑分层

业务逻辑层由一些与流程无关的业务单元或应用组件构成，它们通过存取数据库或其它业务对象实现各自的业务逻辑，如计算商品价格、评估贷款申请人信用度等。这些业务单元或应用组件注册到 IS-Flow 中，成为流程中的基本处理单元。流程逻辑层的功能是管理这些业务流程，包括定义、控制业务单元间的数据流和控制流，以及将业务单元的操作映射到业务逻辑层的实际业务对象或应用组件。

将流程逻辑从应用中分离出来，再配以方便直观的图形化流程定义工具 IS-Flow 流程设计器，即可实现开放的、显式的、松耦合的流程，这种流程管理方案可以缩短设计周期并生产出高质量的产品，允许软件通过集合已存在的软件，组装生成新的应用，而不再要求软件从打草稿开始。采用这种解决方案，企业可以更快的创建灵活敏捷的应用系统。

2.1.2. 工作流引擎服务在应用中的运行模式

IS-Flow 作为一个工作流管理系统，应用系统需要在 IS-Flow 之上开发实现自己的系统。IS-Flow 为应用系统提供多种可选的应用模式。工作流引擎服务有以下两种运行模式，见图表 2-2 IS-Flow 的两种模式。



图表 2-2 IS-Flow 的两种模式

◆ **模式一：嵌入运行模式**

在这种模式下，工作流引擎以一个工具包的形式向应用系统提供服务接口，供应用程序在同一个 Java 虚拟机下调用。IS-Flow 不提供独立的工作流执行服务，而是借助应用系统的应用服务器提供服务。

◆ **模式二：独立运行模式**

在这种模式下，应用程序与工作流引擎分别运行在不同的 Java 虚拟机下，工作流执行服务独立运行，对外提供流程控制服务。应用系统需要开发自己的在其系统内运行的任务表处理器。应用程序与工作流引擎之间通过 JMS、JRMP、RMI、WebService

等协议进行通讯，IS-Flow 应用提供基于这些通讯协议的接口。

对于实现业务过程控制所必须的任务表处理器，IS-Flow 也提供了以下两种应用模式：

◆ 模式一：直接使用标准任务表处理器

IS-Flow 提供标准的任务表处理器供应用使用，可以作为最终用户的对任务进行处理的工具。

将应用程序集成到 IS-Flow 系统中来运行，IS-Flow 为应用程序提供运行的环境与用户界面框架。应用系统以 IS-Flow 所提供的客户端工具向最终用户提供业务功能的服务。

◆ 模式二：自己开发应用的任务表处理器

如果 IS-Flow 所提供的标准任务表处理器不能满足应用系统的需要，或者不能与应用程序在界面风格等方面很好的结合到一起，应用可以使用 IS-Flow 所提供的 API 开发自己的任务表处理器。

2.1.3. 用户数据集成

IS-Flow提供两个抽象类（AbstractResourceProvider和DyAbstractResourceProvider）由用户应用来实现，这将提供给工作流引擎从应用数据库中获取用户资源和用户的通道，IS-Flow 通过应用的 AbstractResourceProvider 实现类或 DyAbstractResourceProvider 的实现类来获取应用的用户资源和用户信息。

工作流定义管理器只负责加载执行人和角色定义数据，并不提供添加、修改和删除的有关方法。应用应首先根据业务实现 AbstractResourceProvider 抽象类或 DyAbstractResourceProvider 抽象类，并在工作流配置文件 inforflow.xml 的 resourceManager 元素处指明实现类的位置，见图表 2-3 inforflow.xml 中资源实现类的配置中的选中的字：

```
<!--
        资源管理，系统提供两种默认类型的资源管理方式，分别为
        从用户数据库中获取资源信息:
        应用系统可以扩展自己所需要的资源管理器，扩展时需实现
        com.cvicse.workflow.api.resource.AbstractResourceProvider
        如果应用需要配置参数可以在resourceProperties指定

        HelloWorld应用演示使用com.cvicse.workflow.examples.resource.AppResourceProvider，
        所有的定义信息均在内存中存在，不需要配置resourceProperties下的参数
    -->
  <resourceManager class="com.cvicse.workflow.examples.resource.AppResourceProvider"/>
```

图表 2-3 inforflow.xml 中资源实现类的配置

AbstractResourceProvider 抽象类实现了 ResourceProvider 接口，DyAbstractResourceProvider 抽象类继承 AbstractResourceProvider。在 ResourceProvider 接口中定义了应用参考实现的 10 个方法，见表格 2-1 ResourceProvider 接口的方法列表：

表格 2-1 ResourceProvider 接口的方法列表

方法返回值	方法描述
OrgnizationUnit[]	getOrgnizationUnits() 返回系统所有组织单位对象的数组
Resource	getResource(java.lang.String id) 返回指定资源标示号的资源对象
Resource[]	getResources() 返回系统所有资源对象的数组
Resource[]	getResources(java.lang.String roleId, java.lang.String orgnizationUnitId) 返回指定角色和机构代码的资源对象数组
Role[]	getRoles() 返回系统所有角色对象数组
User	getUser(java.lang.String id) 返回指定执行人标示号的执行人对象
User[]	getUsers() 返回系统所有执行人对象数组

User[]	getUsers(java.lang.String roleId) 返回指定角色的执行人对象数组
User[]	getUsers(java.lang.String roleId, java.lang.String orgnizationUnitId) 返回指定角色和机构代码的执行人对象数组
java.util.Map	getWorkerExtAttrs() 获取执行人拥有的所有扩展属性定义。

关于 AbstractResourceProvider 实现类或 DyAbstractResourceProvider 实现类的详细介绍参见集成用户数据。

2.1.4. 基于 IS-Flow 的一般开发流程

使用 IS-Flow 开发应用主要有以下几个步骤：业务流程分析、业务流程定义、业务流程开发、业务流程运行调试。每一步又包含了几个更小的步骤。

表格 2-2 基于 IS-Flow 的一般开发流程

阶段	子任务			备注
分析阶段	√	第一步	分析业务执行过程	转移、子流程、活动集
	√	深入一点	分析活动要做什么	
	√		分析活动由谁来做	
	√		分析活动对整个流程的影响	
	√	分析细节	分析活动要处理的数据	
	√		分析业务对象与工作流对象之间的关系	
	√		分析影响业务过程的业务数据	
实现阶段	√	实现任务表处理器		
	√	实现用户接口		
	√	实现业务单元		
	√	实现操作		

	√	实现事件插件	

2.2. 服务类型

工作流引擎提供三种类型的工作流引擎服务，即 local 方式、rmi 方式及 webservice 方式。如果工作流引擎服务以嵌入运行模式集成在应用系统中，那么工作流引擎只能提供 local 方式的服务。如果工作流引擎服务以独立运行模式集成在应用系统中，那么工作流引擎可以提供 RMI 方式或 WebService 方式的服务。

2.3. 事务处理

IS-Flow 提供四种事务处理方式，即无事务、独立事务、传递事务及 JTA 事务。工作流引擎默认提供无事务方式的事务处理机制。如果应用系统需要其它类型的事务处理方式，只需修改 IS-Flow 核心配置文件 inforflow.xml 中的相应配置属性，即修改 /inforflow/transaction/@defaultTxType 的值，defaultTxType 在 inforflow.xml 中的位置见图表 2-4 defaultTxType 在 inforflow.xml 中的位置：

图表 2-4 defaultTxType 在 inforflow.xml 中的位置

有关 inforflow.xml 的详细信息参见 inforflow.xml。

表格 2-3 描述了 IS-Flow 四种事务处理方式的对比：

表格 2-3 IS-Flow 四种事务处理方式的对比

事务类型	配置值	适用范围
无事务	NoTxWorkflowContext	小型项目，当并发量不大，或业务过程简单时，可以采取无事务的管理模式。
独立事务	SeparateTxWorkflowContext	应用系统与工作流管理分别管理自己的事务，二者不进行统一控制。这种模式也仅适用于某些小型项目的开发。
传递事务	TransConnTxWorkflowContext	由应用对应用的业务执行过程及流程控制过程进行统一的开始、提交、回滚的事务控制。这是最为安全可靠的应用模式，可以严格的保证业务数据与流程数据的状态一致性。
JTA 事务	JTAWorkflowContext	工作流引擎与应用系统使用符合 JTS 规范的事务管理器进行事务管理，事务的开始与结束由应用系统统一利用 JTS 事务管理规范进行管理。

使用无事务配置或独立事务方式，需要另外在 inforflow.xml 中配置数据库连接参数，如下：

```
<dataBaseProperties>
    <prop key = "driverClassName">com.mysql.jdbc.Driver</prop>
    <prop key = "url">jdbc:mysql://localhost/test</prop>
    <prop key = "username">root</prop>
    <prop key = "password">root</prop>
</dataBaseProperties>
```

使用传递事务连接方式，需要应用向工作流传递数据库连接，并由应用统一管理数据库连接的回滚和提交事务，典型代码如下：

```
//------------------------------应用传递事务模式下的参考代码片段-开始
     //------------------------------Connection为由应用获取的数据库连接对象
      Connection connection = DBManager.getCon();
     //------------------------------设置数据库自动提交SQL语句为false,事务开始
      connection.setAutoCommit(false);
      WfClient client = null;
      try{
            client = WfClientManager.getInstance().getWfClient();
              //------------------------------将数据库连接对象传递给工作流引擎
            client.connect(new WfConnectInfo(username, connection));

              //------------------------------
               //调用工作流API部分
              //------------------------------

              //------------------------------
              //使用 connection 调用应用逻辑部分
              //------------------------------由应用统一提交数据库事务，事务结束
              connection.commit();

      }catch(Exception ex){
              //------------------------------回滚事务
                 connection.rollback()
      }finally{
              //------------------------------释放资源
              if(connection!=null) connection.close();
                 if(client != null) client.disconnect();
      }
//------------------------------应用传递事务模式下的参考代码片段-结束
```

使用独立事务模式，应用在调用一组工作流控制接口之前，除了必须的 connect() 和 disconnect()操作之外，必须显式使用 beginTransaction()/commitTransaction()/rollback() 分别进行开始，提交和回滚事务，典型代码如下：

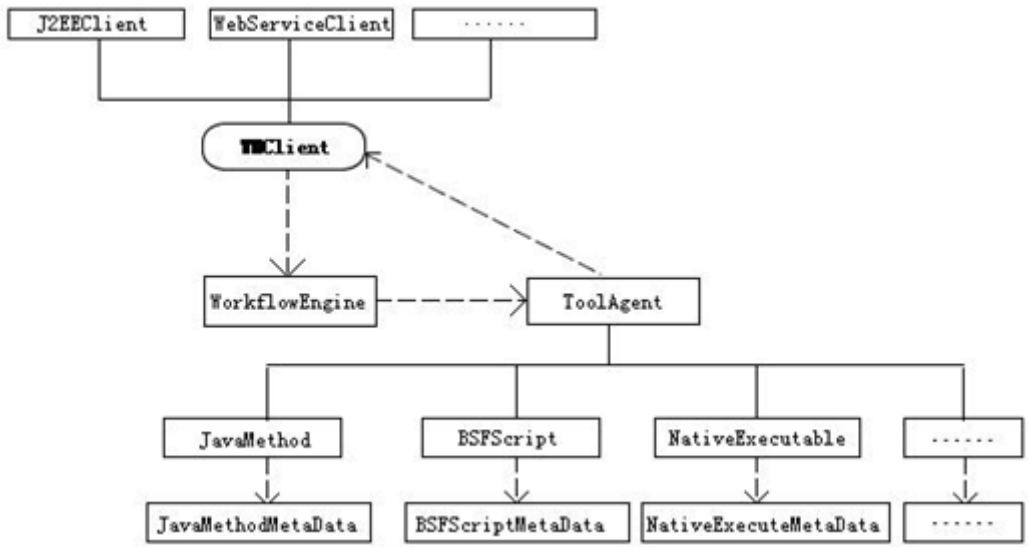
```
//------------------------------独立事务事务模式下的参考代码片段-开始
     //------------------------------Connection为由应用获取的数据库连接对象
      Connection connection = DBManager.getCon();
      WfClient client = null;
      try{
             client = WfClientManager.getInstance().getWfClient();
              //-------------------------使用某用户连接工作流引擎
             client.connect(new WfConnectInfo(username));
              //-------------------------开始工作流的事务
               client.beginTransaction();
              //------------------------------
               //调用工作流API部分
              //------------------------------
              //-------------------------提交工作流的事务
               client.commitTransaction();


             //--------------------设置数据库自动提交SQL语句为false，事务开始
                connection.setAutoCommit(false);;

              //------------------------------
              //使用 connection 调用应用逻辑部分
              //------------------------------
              //------------------------------提交应用的事务
               connection.commit();
      }catch(Exception ex){
              //------------------------------回滚工作流的事务
               client.rollback();
              //------------------------------回滚应用的事务
                connection.rollback();
      }finally{
              //------------------------------释放资源
                if(client != null) client.disconnect();
             if(connection!=null) connection.close();
      }
//------------------------------独立事务模式下的参考代码片段-结束
```

2.4. 工具代理

应用程序通过工具代理被工作流引擎调用，工具代理的相关接口和类的关系见图表 2-5 工具代理调用关系图：

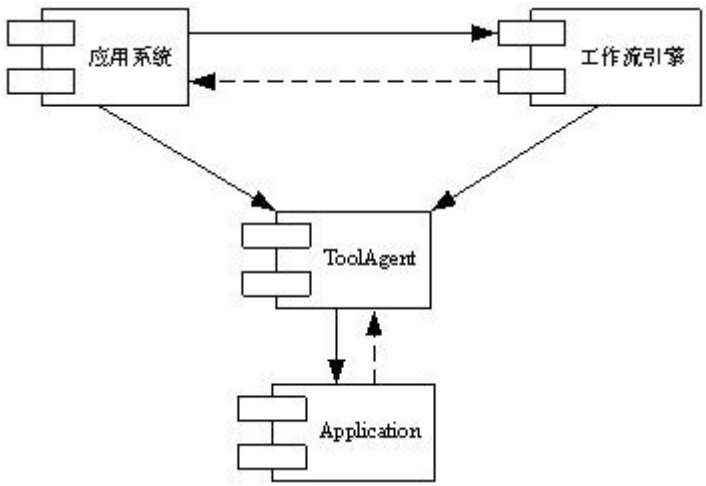


图表 2-5 工具代理调用关系图

对应用可见的接口为图中所示 WfClient，WfClient 可通过 WorkflowEngine 获得工作项，如果工作项的实现为工具，进而获得工作项的工具代理，触发工具实现，在触发工具实现时，通过 WfClient 通知工作流引擎。

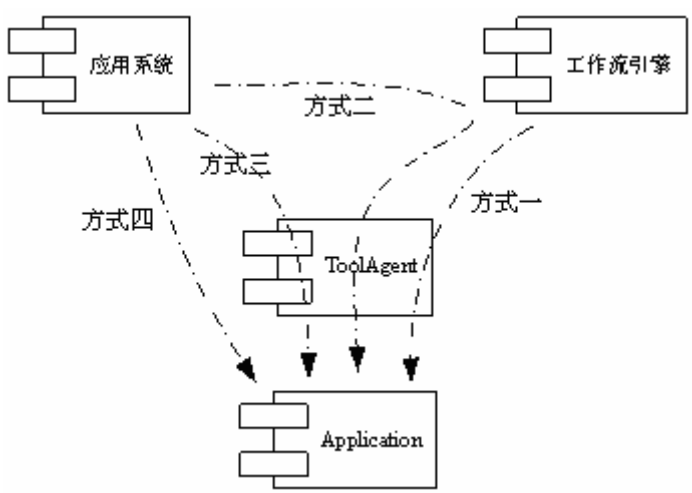
在瘦客户端(B/S 形式)下，应用可不单独开发单独的工具代理实现。不需要序列图中的 invokeApplication，但需要 completeWorkItem 方法将工作项的完成信息通知到工作流引擎，具体的通知时机由应用决定。

在一个基于 IS-Flow 开发的工作流应用系统中，应用系统、工作流引擎服务、工具代理以及被调用的应用之间的关系见图表 2-6 工作流应用系统对象关系图：



图表 2-6 工作流应用系统对象关系图

作为业务组件而存在的应用程序，其被调用的方式由以下几种，见图表 2-7 应用程序调用方式：



图表 2-7 应用程序调用方式

方式一：由工作流引擎自动调用应用程序。在这种情况下，应用程序所属的活动为由引擎自动执行的活动。引擎创建出这个活动后，立即启动活动，并由工作流引擎通过 ToolAgent 自动调用应用程序。这种调用方式要求应用程序的代理实现要遵循 IS-Flow 所规定的接口，以便让引擎知道如何通过代理操作应用程序。

方式二：应用程序何时被执行由应用系统来决定，应用系统首先从引擎获取需要执行的应用程序（标识），选择其中一个应用程序，通过调用工作流引擎所提供的接口（指定要执行的应用程序标识），由工作流引擎根据应用程序所指定的工具代理实例化工具代理对象，并通过工具代理对象激活应用程序。同方式一，这种调用方式也要求应用程序的代理实现要遵循 IS-Flow 所规定的接口，以便让引擎知道如何通过代理操作应用程序。

方式三：应用程序何时被执行由应用系统来决定。应用系统首先从引擎获取需要执行的应用程序（标识），选择其中一个应用程序，应用程序对象本身记录了它所需要的工具代理，获取它的工具代理，然后通过工具代理来激活应用程序的执行。由于是由应用系统来调用工具代理的实现，因此，工具代理的实现可以不遵循 IS-Flow 所规定的接口实现。

方式四：应用程序何时被执行由应用系统来决定。应用系统首先从引擎获取需要执行的应用程序（标识），选择其中一个，并直接执行之。这种方式只适用于应用程序的类型单一的情况下，应用系统不需要对如何调用应用程序做出决策，因此也就省去了工具代理的工作。

表格 2-4 描述了四种应用程序调用方式中，工作流引擎所发挥的作用。

表格 2-4 四种方式下工作流引擎发挥的作用

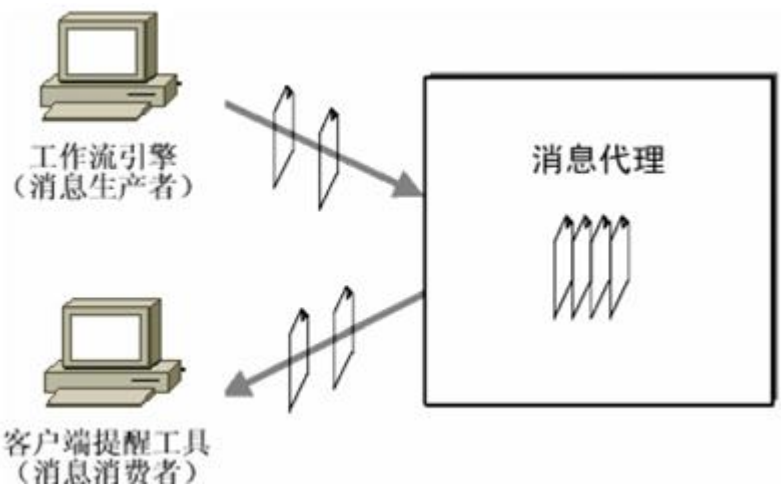
	What	How	When	Do	说明
方式一	√	√	√	√	引擎知道做什么、如何做、何时做，并亲自做。
方式二	√	√	×	√	引擎知道做什么、如何做，不知何时做，由应用系统触发，由引擎负责调用。
方式三	√	√	×	×	引擎知道做什么、如何做，但是不知道何时做，需要应用系统触发并执行。

方式四	√	×	×	×	引擎只知道做什么，但是不知道如何做，需要应用系统自己来确定如何做。

2.5. 消息提醒

消息提醒工具由消息服务器和消息客户端工具两部分组成。如果引擎需要发送消息，发送消息前必须启动消息提醒服务（消息服务器），消息提醒客户端通过消息提醒服务注册并接收消息，向引擎提供有效用户信息，引擎方可将产生的提醒消息通过消息提醒服务发送给消息提醒客户端；工作流引擎做为消息生产者向消息服务器发送消息，消息客户端作为消息消费者通过消息服务器消费消息。

使用消息代理的网络部署图见图表 2-8 系统网络部署图：



图表 2-8 系统网络部署图

在 inforflow.xml 文件中配置工作流引擎与消息服务器连接属性，如图表 2-9 inforflow.xml 中的消息配置所示：

```
<messager default="true">  <!--是否向已注册客户端消息提醒的用户发送消息-->
  <messagerProperties>
    <prop key="url">127.0.0.1</prop>      <!--消息中心的地址-->
    <prop key="port">61616</prop>         <!--消息中心公开的端口号-->
    <prop key="isPersistent">true</prop> <!--发送的消息是否需要持久化-->
  </messagerProperties>
</messager>
```

图表 2-9 inforflow.xml 中的消息配置

有关消息提醒工具的使用参见***《InforSuite Flow 消息提醒工具使用手册》***。

2.6. 事件监听

工作流引擎在运行的过程中可以抛出多种事件，通过这些事件，应用系统可以根据实际业务流程的需要为流程的运行增加业务上的特性。

流程事件监听器接口为 com.cvicse.workflow.api.event.WfProcessInstanceListener，这个接口提供了 14 个方法供用户扩展，如下：

- beforeProcessInstanceCreate：流程实例创建之前进行监听
- afterProcessInstanceCreate：流程实例创建之后进行监听
- beforeProcessInstanceStart：流程实例启动之前进行监听
- afterProcessInstanceStart：流程实例启动之后进行监听
- beforeProcessInstanceTerminated：流程实例终止之前进行监听
- afterProcessInstanceTerminated：流程实例终止之后进行监听
- beforeProcessInstanceAborted：流程实例取消之前进行监听
- afterProcessInstanceAborted：流程实例取消之后进行监听
- beforeProcessInstanceSuspend：流程实例挂起之前进行监听
- afterProcessInstanceSuspend：流程实例挂起之后进行监听
- beforeProcessInstanceResume：流程实例恢复之前进行监听
- afterProcessInstanceResume：流程实例恢复之后进行监听
- beforeProcessInstanceComplete：流程实例处理完成之前进行监听

- afterProcessInstanceComplete：流程实例处理完成之后进行监听

活动事件监听器接口为 com.cvicse.workflow.api.event.WfActivityInstanceListener，这个接口提供了 22 个方法供用户扩展，如下：

- beforeActivityInstanceCreate：活动实例创建之前进行监听
- afterActivityInstanceCreate：活动实例创建之后进行监听
- beforeActivityInstanceStart：活动实例启动之前进行监听
- afterActivityInstanceStart：活动实例启动之后进行监听
- beforeActivityInstanceTerminated：活动实例终止之前进行监听
- afterActivityInstanceTerminated：活动实例终止之后进行监听
- beforeActivityInstanceAborted：活动实例取消之前进行监听
- afterActivityInstanceAborted：活动实例取消之后进行监听
- beforeActivityInstanceSuspend：活动实例挂起之前进行监听
- afterActivityInstanceSuspend：活动实例挂起之后进行监听
- beforeActivityInstanceResume：活动实例恢复之前进行监听
- afterActivityInstanceResume：活动实例恢复之后进行监听
- beforeActivityInstanceComplete：活动实例处理完成之前进行监听
- afterActivityInstanceComplete：活动实例处理完成之后进行监听
- beforeActivityInstanceJump：活动实例跳转之前进行监听
- afterActivityInstanceJump：活动实例跳转之后进行监听
- beforeActivityInstanceBack：活动实例回退之前进行监听
- afterActivityInstanceBack：活动实例回退之后进行监听
- beforeActivityInstancePutback：活动实例放回之前进行监听
- afterActivityInstancePutback：活动实例放回之后进行监听
- beforeActivityInstanceWithdraw：活动实例追回之前进行监听
- afterActivityInstanceWithdraw：活动实例追回之后进行监听

工作项事件监听器接口为 com.cvicse.workflow.api.event.WfWorkItemListener，这个接口提供了 14 个方法供用户扩展，如下：

- beforeWorkItemCreate：工作项创建之前进行监听

- afterWorkItemCreate：工作项创建之后进行监听
- beforeWorkItemAccept：工作项接收之前进行监听
- afterWorkItemAccept：工作项接收之后进行监听
- beforeWorkItemTerminated：工作项终止之前进行监听
- afterWorkItemTerminated：工作项终止之后进行监听
- beforeWorkItemAborted：工作项取消之前进行监听
- afterWorkItemAborted：工作项取消之后进行监听
- beforeWorkItemSuspend：工作项挂起之前进行监听
- afterWorkItemSuspend：工作项挂起之后进行监听
- beforeWorkItemResume：工作项恢复之前进行监听
- afterWorkItemResume：工作项恢复之后进行监听
- beforeWorkItemComplete：工作项处理完成之前进行监听
- afterWorkItemComplete：工作项处理完成之后进行监听

关于事件监听器的实现参见编写事件监听器。

2.7. 定时启动

2.7.1. 流程定时启动

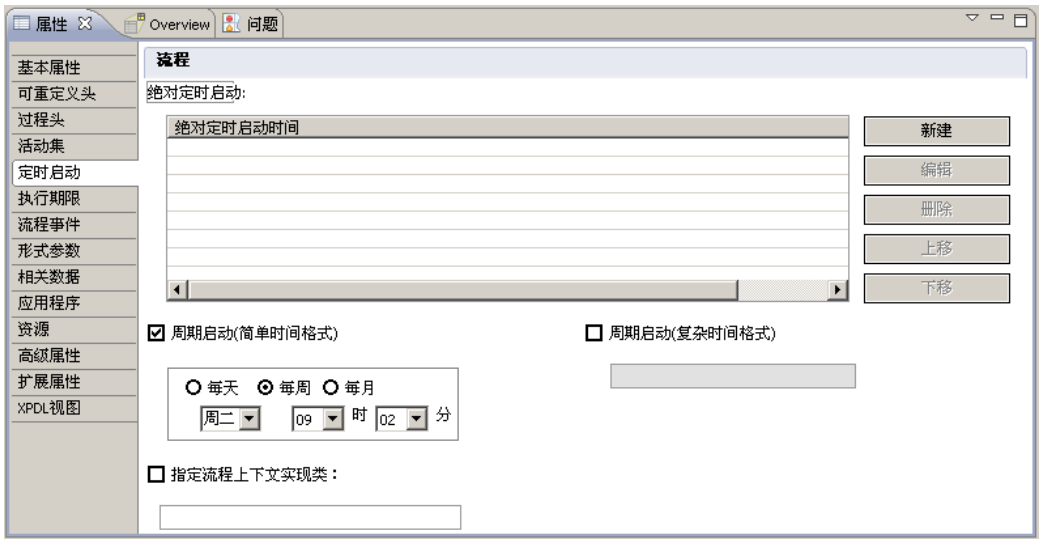
定时启动属于调度的范畴。流程的定时启动是指在自定义的时间内启动流程实例。在业务流程建模时或者流程实例创建后可以使用定时启动功能。

- 业务流程建模使用定时启动。

业务流程建模时可以通过流程设计器设计流程的定时启动，如果流程需要为相关数据赋值，必须设置流程上下文实现类，这个类里实现了对相关数据的赋值。详细信息参见***《InforSuite Flow 流程设计器使用手册》***。

如果业务流程建模时设计了流程的定时启动，那么将流程定义导入数据库后，工作流引擎会在指定的时间里启动流程实例。

图表 2-10 为流程设计器中设置流程定时启动的界面：



图表 2-10 流程设计器设置流程定时启动界面

◆ 流程实例创建后使用定时启动。

流程实例创建后，需要启动流程实例使其进入运行状态。IS-Flow 提供了三种启动流程实例的方式：即时启动、定时启动和图形启动。定时启动分为相对时间启动和绝对时间启动。

如果启动流程实例需要给相关数据赋值，则必须指定流程上下文实现类，这个类实现 com.cvicse.workflow.api.scheduler. ProcessContextProvider 接口。

图表 2-11 为标准任务表处理器的启动流程实例界面：

实例名称	实例ID	发起人	创建时间	状态	定义标识	启动	图形启动	定时启动
测试抢任务	944	User_1	2009-05-18 13:16:10	待处理	WorkAssign:MultiAssign			

图表 2-11 启动流程实例界面

图表 2-12 为标准任务表处理器的定时启动界面：

图表 2-12 流程实例定时启动界面

流程上下文实现类需要用户编程，参见流程上下文实现类。

2.7.2. 活动定时启动

活动定时启动页属于调度的范畴。活动的定时启动是指在自定义的时间内启动活动实例。在流程实例运行到该活动时可以使用定时启动功能。

用户在定义流程时将某个活动配置为定时启动，在流程运行阶段，该活动将定时启动。启动时间可以为“绝对时间”和“相对时间”，相对时间是相对于加入调度的时刻。在流程运行到活动节点时，如果配置了启动时间，则将启动活动的加入调度的队列，在到达指定的启动时间时才真正创建和启动活动实例。

工作流引擎提供接口，可以创建一个自动启动活动，启动时间可以通过参数设置，如果没有参数配置，则去查找扩展属性，如果都没有则抛出异常。

◆ 业务流程建模使用活动定时启动。

业务流程建模时可以通过流程设计器设计活动的定时启动，如果流程需要为相关数据赋值，必须设置流程上下文实现类，这个类里实现了对相关数据的赋值。

打开 BS 设计器，点击“高级属性”中的“定时启动”页签，进入定时启动设置面板，如图 2-13

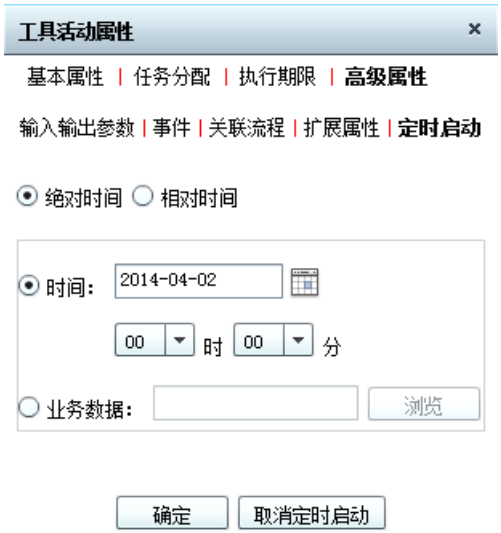


图 2-13 定时启动

选择活动启动的“绝对时间”或“相对时间”。指定的时间可以选择“时间”或指定“业务数据”进行配置。业务数据赋值格式如下：

绝对时间格式：yyyy-mm-dd hh:mm,如 2013-07-09 00:00

相对时间格式：天.时.分.秒

◆ 流程实例运行到配置定时启动的活动实例。

如果业务流程建模时设计了活动的定时启动，那么将流程实例运行到该活动后，仅创建该活动实例但不启动，同时，引擎调度管理器实现对活动启动事件进行调度。工作流引擎会在指定的时间里启动活动实例。

如果启动活动实例配置了相关数据，则需要在运行到该活动之前给该相关数据赋值，如果未赋值则活动立即启动。

在未到达活动启动时间之前，也可以手动启动活动实例，图表 2-14 为标准任务表处理器的启动活动实例界面：

图表 2-14 启动流程活动实例界面

图表 2-15 为标准任务表处理器活动的定时启动界面。

图表 2-15 活动实例定时启动界面

2.8. 执行期限

执行期限是 IS-Flow 提供的任务催办功能，如果在指定时间内流程实例或活动实例没有结束，工作流引擎可以通过设置的执行期限进行相应的处理。

IS-Flow 提供了多种形式的执行期限，如邮件提醒、手机短信等。

执行期限分为流程执行期限和活动执行期限。其中流程执行期限包括：

- 邮件提醒
- 手机短信
- 终止流程
- 自定义事件

活动执行期限包括：

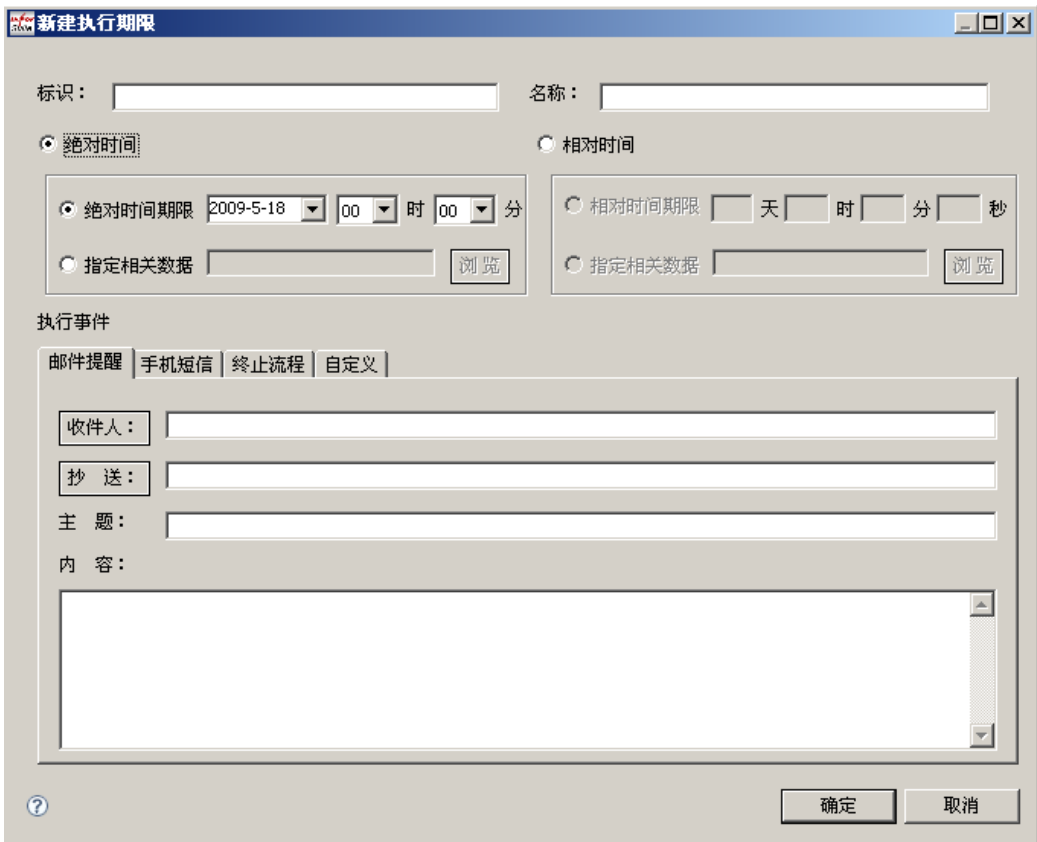
- 邮件提醒
- 手机短信
- 重新分配任务
- 终止工作项
- 自定义事件

其中，执行期限自定义事件需要用户编程，参见编写执行期限自定义事件。

图表 2-16 为流程设计器中设置流程执行期限界面：

图表 2-16 流程设计器中设置流程执行期限界面

图表 2-17 为流程设计器中新建执行期限界面：



图表 2-17 流程设计器中新建执行期限界面

活动执行期限的设置类似流程执行期限，具体的设置信息参见***《InforSuite Flow 流程设计器使用手册》***。

第3章 IS-Flow 业务流程建模

本章详细介绍了业务流程建模，包括以下内容：

- 流程建模指南。介绍了 IS-Flow 进行业务流程建模的原理。
- 流程定义导入导出工具。介绍了流程建模结束后，如何通过这个工具将流程定义导入数据库中，或者从数据库导出到本地的文件系统。

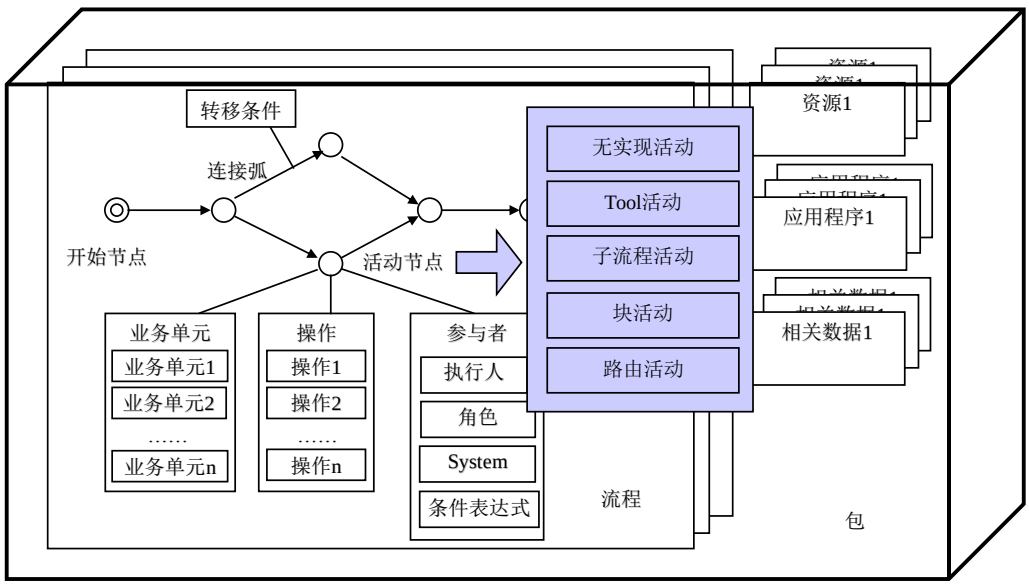
学习本章内容后，您可以：

- 了解 IS-Flow 进行业务流程建模的原理，并结合《*InforSuite Flow 流程设计器使用手册*》使用流程设计器进行流程建模。
- 将流程建模的结果（即流程定义）通过流程定义导入导出工具导入到数据库中供 IS-Flow 工作流引擎服务使用，或者将数据库中的流程定义导出到本地文件系统。

3.1. 流程建模指南

IS-Flow 工作流元模型基于 WfMC 规范实现，是对业务流程所具有的共性的完善的抽象。IS-Flow 在对支持复杂业务流程的分层建模、复杂任务分配方式以及应付易变的业务过程方面都具有独到之处，使之可以轻松应付这些复杂性，降低应用系统的开发难度，也减轻了开发人员的工作量。IS-Flow 设计器是开发人员进行 IS-Flow 流程定义的工具，可对工作流引擎所需的所有元模型进行可视化编辑。

IS-Flow 所支持的工作流元模型见图表 3-1 IS-Flow 的工作流元模型：



图表 3-1 IS-Flow 的工作流元模型

下面详细介绍 IS-Flow 工作流元模型的每个元素。

3.1.1. 包

包就像一个容器，一个包中包含了多个业务相关的流程定义以及这些流程运行时所需要的资源，调用的应用程序，所使用的相关数据。包将这些信息包容在一起，使得对业务流程的描述更加清晰。包中还存放了许多工作流流程定义实体的公共属性（作者、版本、发布状态等）。包中的每一个流程定义都自动继承包中的公共属性，除非在流程定义中那些属性被个别的重新定义。在包中，对象的定义范围是全局的，并且这些对象能被同一包中的任何工作流定义所引用。开发人员可以通过直接的引用包中的对象，提高开发效率，减少工作时间。在 IS-Flow 工作流模型中一个流程定义对应有一个唯一标识的包存在，即一个流程只能存在一个包容器中。

同时还可以引用外部包，设置引用外部包之后，必须重新启动设计器才可以引用外部包中的资源。

目前 IS-Flow 模型支持的包的具体的属性信息如下：

表格 3-1 包的属性列表

包（Package）		
属性名	允许为空	描述
标识	否	包的标识号 ，唯一的值
名称	否	包的名字
可重定义包头（RedefinableHeader）		
属性名	允许为空	描述
作者	是	包定义作者
版本	是	包定义版本
页数	是	正文部分代码页数
国家代码	是	国家代码
包——扩展属性（Package——ExtendedAttributes）		
属性名	允许为空	描述
名称	否	扩展属性名字
值	否	扩展属性值

3.1.2. 流程

3.1.2.1. 概述

流程定义也称为业务流程，在 IS-Flow 工作流元模型中，一个包可以含多个流程定义，每个流程定义都是一个可独立运行的完整的业务流程。一个流程定义包含一个

开始节点、一个结束节点以及多个活动节点。开始节点定义了流程唯一的入口点，结束节点定义了流程唯一的出口点，活动节点的实现类型有五种，分别为：工具活动、自动活动、子流程活动、块活动、路由活动。

流程定义实例化为一个流程自身的容器，提供流程管理（如创建日期、创建人等）信息，或者流程执行中将会用到的相关的信息（如初始化参数，执行优先级等等）。流程定义是包中最重要的定义元素，包内定义的其他元素都是为流程定义服务。

流程定义为流程中其他实体提供上下文信息，在运行阶段流程实例将其上下文信息输入到指定的活动实例定义的输入中，活动实例进而将其上下文信息提供给工作项。

目前 IS-Flow 支持的流程定义的具体的属性信息如下表格 3-2 所示：

表格 3-2 流程定义的属性列表

流程定义（WorkflowProcesses）		
属性名	允许为空	描述
标识	否	流程定义标识号
名称	否	流程定义名字
访问级别	否	流程定义的访问级别，public/private 两类，public 能被外部包访问。
流程定义头（ProcesseHeader）		
属性名	允许为空	描述
创建时间	是	定义的创建时间
描述	是	定义描述
优先级	是	流程定义的优先级
限制	是	流程定义的最长持续时间
可重复定义流程定义头（RedefinableHeader）		
属性名	允许为空	描述
状态	否	流程定义的当前状态：未发布，发布，未测试，需要注意的是只有状态为发布的流程才能在流

		程引擎中被正确的执行。
作者	是	流程定义的创建人
版本	是	流程定义的版本
代码页	是	正文部分使用的代码页
国家代码	是	国家代码

3.1.2.2. 定义流程形式参数

一个流程定义可以定义一个或者多个形式参数，相当于流程启动时要求的输入以及流程结束后的输出，该形式参数的数值在运行阶段作为流程的相关数据（或称为流程上下文）保存。

在流程为手工启动(即非子流程形式)时工作流会自动检查模式为 IN 或者 INOUT 的形式参数对应的相关数据是否为空，若为空，会抛出 WfNoInputException 的异常。

流程实例在运行时为形式参数赋值有两种方式：

- 若该流程不是作为子流程使用，则应使用为流程相关数据赋值的 WAPI。
- 若该流程作为子流程活动所指定的子流程，则应定义子流程活动对应的实际参数，在运行阶段工作流引擎在创建子流程活动并启动该子流程时候，自动将子流程活动的实际参数对应的相关数据的值赋予该子流程的形式参数并作为子流程的相关数据，若子流程活动的实参值为空，自动为子流程的形参赋空值。

流程的输出，即模式为 OUT 或 INOUT 的形式参数只应用于同步执行的子流程，当该子流程结束后，自动将该形式参数对应的相关数据的值更新到父流程中。

3.1.2.3. 定义子流程执行方式

工作流建模工具允许将一个节点的类型定义为子流程活动，启动子流程的方式可以定义为同步或者异步。

- 当子流程的启动方式为同步启动时：

- 子流程启动时父流程变换为挂起状态直至子流程结束；

  若子流程定义了模式为 IN 或 INOUT 的形式参数，工作流引擎自动将子流程活动对应的实参值设置到子流程的的上下文中。

- 子流程结束后需要等待父流程结束才会将数据转移至历史库中。

  若子流程定义了模式为 OUT 或 INOUT 的形式参数，工作流引擎自动将对应的子流程的相关数据设置到父流程的的上下文中。

◆ 当子流程的启动方式为异步启动时：

- 子流程启动时父流程继续运行；

  若子流程定义了模式为 IN 或 INOUT 的形式参数，工作流引擎自动将子流程活动对应的实参值设置到子流程的的上下文中。

- 子流程结束后立即将该子流程的数据转移至历史库中；且不会对父流程有任何影响。

3.1.2.4. 定义流程事件

流程定义允许应用定义流程事件，定义的方式可以选择直接定义，可直接输入流程事件的类名（包含完整的类路径）。流程定义允许定义多个事件类。

用户实现流程事件接口后的事件类必须实现 com.cvicse.workflow.api.event.WfProcessInstanceListener 接口。引擎在加载每个流程定义时，根据此流程定义运行的流程实例将触发流程事件类实现的相应方法。

定义流程事件也可以选择工具代理方式，工具代理实现事件插件包括两种接口：JAVA 方法调用和 WebService 方法调用。首先工具代理开发人员应事先定义好工具代理类，在应用程序定义中新建 JAVA 方法调用事件或者 WebService 方法调用事件；然后为流程定义事件，通过新建工具代理实现事件插件；新建过程中选择什么时候触发事件、触发哪个应用事件，以及相应的应用事件形参与流程的实参对应关系。

事件定义时需要注意逻辑的正确性；例如，定义流程创建前事件时，该事件接口

实现时是不允许对当前流程进行任何逻辑处理，毕竟当前流程还没有被创建，容易出现为当前流程相关数据赋值而抛找不到流程实例的异常，所以对当前流程的操作只能放到流程创建后的事件中。

事件实现的时候，一般除了修改或调用应用上的某些数据、接口外，还需要修改流程的数据来控制流程的逻辑；下面将重点介绍下事件实现中如何有效的调用引擎接口，对流程逻辑进行控制；调用引擎接口目前可以分为两种方式：

首先介绍下面将要用到的两个概念：父连接和子连接。触发事件运行的引擎连接称为父连接，事件实现过程中再次连接引擎称之为子连接。

◆ 方式一：在事件内部直接使用父连接调用工作流引擎。

首先获取 IS-Flow Session 中存放的父连接，通过此引擎连接调用引擎接口，来达到在事件内部调用工作流引擎的目的。这种方式的优点就是，事件实现过程与触发该事件的引擎为同一引擎连接、数据库连接和事务控制，使用比较方便、安全、且效率较高；缺点就是，事件内部无法更换登录工作流引擎的用户信息等。如果不涉及登录用户信息的更改，建议使用该方式。

参考例子代码如下：

事件实现参考代码：

```
//-----------------------------------------------------事件实现代码片段-开始
     /* 实现事件接口 WfActivityInstanceListener 中的下面这个方法 */
public void afterActivityInstanceCreate(WfActivityInstanceEvent arg0)
                throws WfListenerException {
     try {
        /* 首先从已知对象中获取必要信息 */
        WfActivityInstance  ai = arg0.getActivityInstance();
        String pid = ai.getProcessInstanceId();
        /* 获取触发事件引擎(父连接引擎) */
        WAPIExt engine =
        (WAPIExt) SessionManager.getCurrentThreadSession()
```

```
                .get(Global.WORKFLOWENGINE);
        /* 调用父连接引擎接口,操作流程数据 */
        engine.assignProcessInstanceAttribute(engine.getSessionId(),pid,
                        "afterActivityInstanceCreate",
                        "con of afterActivityInstanceCreate");
            } catch (Exception e) {
                e.printStackTrace();
            }
    }
//-----------------------------------------------------事件实现代码片段-结束
```

◆ 方式二：在事件内部建立基于父连接的子连接调用工作流引擎。

在事件内创建基于父连接的子连接。此方式中父连接和子连接具有单独的用户信息等工作流上下文，但共用同一个数据库连接和事务控制，这样可以确保父连接和子连接对象状态一致。这种方式解决了方式一中不能更改用户信息的不足，缺点是：子连接无法控制事务。

子连接构造 WfConnectInfo 时，需要将父连接的 sessionId 传递给子连接，其他操作和正常调用引擎接口相同。其中父连接 sessionId 获取过程为，首先像方式一那样将父引擎 WAPIExt 获取到，然后 getSessionId()即可。参考例子代码如下：

事件实现参考代码：

```
//-----------------------------------------------------事件实现代码片段-开始
     /* 实现事件接口 WfActivityInstanceListener 中的下面这个方法 */
public void afterActivityInstanceCreate(WfActivityInstanceEvent arg0)
        throws WfListenerException {
    try {
    /* 根据父连接 sessionId 建立子连接 */
    WfClient client = WfClientManager.getInstance().getWfClient();
    WAPIExt engine = (WAPIExt) SessionManager.getCurrentThreadSession()
```

```
        .get(Global.WORKFLOWENGINE);
    WfConnectInfo connectInfo =
        new WfConnectInfo("User_1", "", engine.getSessionId());
    /* 子连接连接引擎 */
    client.connect(connectInfo);
    /* 从已知对象中获取必要信息 */
    WfActivityInstance  ai = arg0.getActivityInstance();
    String pid = ai.getProcessInstanceId();
    /* 调用子连接引擎接口,操作流程数据 */
    client.assignProcessInstanceAttribute(pid,
    "afterActivityInstanceCreate","con of afterActivityInstanceCreate");
    client.disconnect();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
//-----------------------------------------------------事件实现代码片段-结束
```

注意：使用方式二连接引擎，其事务操作完全交给了父连接控制，子连接所做的事务提交不会起到任何作用。

3.1.2.5. 定义流程执行期限自定义事件

流程执行期限允许自定义执行事件，定义的方式为直接指定事件的类名（包含完整的类路径）。

自定义事件类必须实现 com.cvicse.workflow.api.scheduler.ExecutionLimitEvent 接口。引擎在加载每个流程定义时，根据此流程定义运行的流程实例将触发流程执行期限的自定义事件类实现的相应方法。

3.1.2.6. 定义流程定时启动

对一些需要在固定时间或者周期性创建和启动的流程，可以为流程定义定时启动属性，定时启动包括绝对定时启动和周期定时启动两种，可以为一个流程定义多个绝对启动时间，还可以为流程定义周期启动属性，周期启动格式包括简单格式和 Unix 格式，两者只能建其一。简单格式的周期启动：可以定义为每天的几时几分、每周周几的几时几分、每月的几日几时几分；Unix 格式的周期启动：用户定义 Unix 格式的定时启动时间，无论是绝对启动还是周期启动，到指定时间时，引擎负责创建并启动流程。

3.1.3. 节点

3.1.3.1. 概述

目前在 IS-Flow 工作流元模型中支持以下四类节点类型:工具活动、子流程活动、块活动、路由活动。

1. 工具活动：当一个活动的实现类型是“工具类型”时，活动可以包含一个或者多个工具，工具可以是应用程序，当为活动增加工具时，需要指明对应此工具的应用程序的形参的输入参数，即实际参数集，实参可以从活动所包含的相关数据中取得。
2. 子流程活动：当一个活动的实现类型是“子流程类型”时，此活动可以引用其它的流程定义。同应用程序的定义相同，也需要定义对应此流程的形参的实参集，实参可以从活动所包含的相关数据中取。此外，还需要指明流程的执行方式（同步还是异步）。在异步执行的情况下，活动的执行持续到子流程的过程实例初始化后。在同步执行的情况下，在子流程实例被初始化后，活动将被挂起。在这个流程实例运行终止后，活动被恢复。在子流程完成时，子流程与父流程间可能要使用返回参数。
3. 块活动：当一个活动的实现类型是“块活动类型”时，此活动可以指定一组活动节点，与子流程类似，但两者的区别是块活动的执行方式只是同步方式执行。另外，块活动不能够单独作为一个流程实例来运行，但子流程可以。

4. 路由活动：可以看作是一个路径，即没有执行者，也没有应用程序。并且路径活动的运行对工作流相关数据，或者应用程序数据没有任何影响。

5. 自动活动：一种特殊的工具活动。其参与者默认分配的是系统执行人，且不可修改。流程运行到该节点会自动调用所配置的业务单元。

表格 3-3 节点的属性信息

活动（Activity）		
属性名	允许为空	描述
标识	否	活动的标识
名称	否	
描述	是	
类型	否	类型有：路由活动、工具活动、块活动、子流程活动，注意不可以通过属性面板更改。当为工具、子流程或块活动类型时，可以设置属于此活动的特殊属性信息。
执行人	否	与工作流参与者关联，也可能是一个表达式，在编辑器的资源页中定义。
启动模式	否	选择：1、自动 2、手工
结束模式	否	选择：1、自动 2、手工
图标	是	自定义图片文件，显示项目的 icons 目录下的文件，选择空的情况下，系统用默认的图标。
X 坐标	否	
Y 坐标	否	
转移约束		
前置条件	是	当节点有多条输入连接弧时候，可以设置为 And（汇聚类型）和 Xor(分支类型)
后置条件	是	当节点有多条输出连接弧时候，可以设置为 And（汇聚类型）和 Xor (分支类型)
子流程活动（SubFlowActivity）		

标识	是	子流程类型活动特有属性，指子流程活动指向的流程的标识。
执行方式	是	子流程类型活动特有属性，指子流程活动指向的流程的执行方式。可选同步或异步。
块活动（BlockActivity）		
块活动标识	是	块活动特有属性，指块活动指向的活动集的标识。

3.1.3.2. 定义任务分配方式

在运行阶段，工具活动首先被实例化为活动实例(WfActivityInstance)，然后根据指定的分配方式产生一个或者多个工作项实例(WfWorkItem)，当所有工作项完成之后才可以结束活动实例，在任务分配时增加了“组”的概念，在同一个活动定义中可以指定多组人来处理，对于每一组人员的人员结构不加任何限制，一个组员可以是一个执行人、一个角色、或者是条件表达式；组员的数量不作限制。以下为几种涉及任务分配的典型应用场景:

- 组间、组内会签模式

  可能执行相同业务活动的多个人分组之后，任务由所有组内的所有人共同完成。比较典型的会签模式的例子是对某种申请的多人审批，如银行信贷管理过程中的贷审会。在会签的过程中，每个人视审批的内容不同，可以做不同的审批工作，为更好的协同每个人的工作，可以由一位秘书进行审批意见的汇总以及将工作情况及时通报给每一个人。

- 组间会签组内抢任务模式

  可能执行相同业务活动的多个人分组之后，每个组只选择一个人来执行任务。

- 组间互斥组内抢任务模式

可能执行相同业务活动的多个人分组之后，任务由其中一个组内的某一个人完成。

- 组间互斥组内会签模式

  可能执行相同业务活动的多个人分组之后，任务由其中一个组内的所有人共同完成。

3.1.3.3. 定义节点的输入输出

在运行阶段，当创建活动实例时工作流引擎将指定的流程上下文设置到活动实例的相关数据中；当活动结束后，工作流引擎将指定的活动上下文更新到流程上下文中。节点的输入输出在节点的扩展属性中指定，其值为相应的相关数据，扩展属性解释见表格 3-4 扩展属性信息。

表格 3-4 扩展属性信息

扩展属性名称	描述
In_Variable	活动实例的输入为该名称对应的相关数据的值，在创建活动实例时，会将该值对应的流程上下文设置到活动上下文中。
Out_Variable	活动实例的输出为该名称对应的相关数据的值，活动实例结束后，会将该值对应的活动上下文更新到流程上下文中。
IN_OUT_Variable	扩展属性对应值即为活动实例的输入，又为活动实例的输出。

3.1.3.4. 定义活动实例事件

节点定义在运行阶段将产生活动实例，活动实例根据节点定义中执行人的分配产生一到多个工作项，IS-Flow 节点定义允许为活动实例和工作项实例定义事件。

节点定义允许应用定义活动实例事件，定义的方式可以为直接定义，可直接输入活动事件监听器实现类的完整包路径名。节点定义允许定义多个活动事件类。

用户实现活动事件接口后的事件类必须实现 com.cvicse.workflow.api.event.

WfActivityInstanceListener 接口。引擎在加载每个节点定义时，根据此流程定义运行的活动实例将触发活动事件类实现的相应方法。

定义活动事件也可以选择工具代理方式，工具代理实现事件插件包括两种接口：JAVA 方法调用和 WebService 方法调用。首先工具代理开发人员应事先定义好工具代理类，在应用程序定义中新建 JAVA 方法调用事件或者 WebService 方法调用事件；然后为活动定义事件，通过新建工具代理实现事件插件；新建过程中选择什么时候触发事件、触发哪个应用事件，以及相应的应用事件形参与流程的实参对应关系。

事件实现过程中，涉及修改流程数据需要引用父连接或者新建子连接的事件，请参考定义流程事件一节，那里详细描述了事件实现过程如何引用父连接和如何新建子连接。

3.1.3.5. 定义工作项实例事件

节点定义在运行阶段将产生活动实例，活动实例根据节点定义中执行人的分配产生一到多个工作项，IS-Flow 节点定义允许为活动实例和工作项实例定义事件。

节点定义允许应用定义工作项实例事件，定义的方式可以为直接定义，可直接输入工作项事件监听器实现类的完整包路径名。节点定义允许定义多个工作项事件类。

用户实现活动事件接口后的事件类必须实现 com.cvicse.workflow.api.event.WfWorkItemInstanceListener 接口。引擎在加载每个节点定义时，根据此流程定义运行的工作项实例将触发活动事件类实现的相应方法。

定义工作项事件也可以选择工具代理方式，工具代理实现事件插件包括两种接口：JAVA 方法调用和 WebService 方法调用。首先工具代理开发人员应事先定义好工具代理类，在应用程序定义中新建 JAVA 方法调用事件或者 WebService 方法调用事件；然后为流程定义事件，通过新建工具代理实现事件插件；新建过程中选择什么时候触发事件、触发哪个应用事件，以及相应的应用事件形参与流程的实参对应关系。

事件实现过程中，涉及修改流程数据需要引用父连接或者新建子连接的事件，请参考定义流程事件一节，那里详细描述了事件实现过程如何引用父连接和如何新建子连接。

3.1.3.6. 定义活动执行期限自定义事件

活动执行期限允许自定义执行事件，定义的方式为直接指定事件的类名（包含完整的类路径）。

自定义事件类必须实现 com.cvicse.workflow.api.scheduler.ExecutionLimitEvent 接口。引擎在加载每个流程定义时，根据此流程定义运行的流程实例将触发活动执行期限的自定义事件类实现的相应方法。

3.1.4. 连接弧

3.1.4.1. 概述

在流程中，活动节点之间通过连接弧来连接起来，连接弧描述在工作流运行期间转移发生与否的条件，活动实体的控制约束也与转移信息相关。在流程包含分支的情况下执行条件可以通过连接弧的条件来指定，如在一个申请金流程中可以通过指定连接弧的条件是大于 3000 元执行活动 B（上级主管二次审核）和在小于等于 3000 元的情况下执行执行活动 C（财务审核）。

在 IS-Flow 工作流模型中，活动集与连接弧信息是 1 对多的关系，流程与连接弧信息也是 1 对多的关系。连接弧信息的属性有 ID，所属标识号，名称，描述，源节点，目标节点，类型，XPRESS，以及所引用的扩展属性集。

表格 3-5 连接弧属性信息

连接弧信息（Transition）		
属性名	是否允许为空	描述
标识	否	连接弧的唯一标识
名称	否	
描述	是	
源节点	是	在源节点为开始节点的情况下，属性为显示空，不可以通过属性页面直接设置

目标节点	是	在目标节点为结束节点的情况下，属性为空显示，不可以通过属性页面直接设置
类型	否	1:永真（空） 2:条件（condition） 3:缺省（otherwise） 4:异常（exception） 5:默认异常（default exception）
条件	否	转移条件表达式，以工作流相关数据为基础。缺省值：TRUE

对转移信息的类型解释如下：

表格 3-6 转移信息

转移信息：	
永真（空）	没有条件
条件（condition）	转移条件得到满足，转移才会被执行
缺省（otherwise）	说明此转移为缺省转移，如果其他转移的转移条件都不能得到满足，则执行此转移
异常（exception）	异常转移，如果产生了一个异常，并且转移条件得到满足，则执行该转移
默认异常（default exception）	异常缺省转移，如果产生了异常，并且其他异常转移的转移条件都得不到满足，则执行此转移

3.1.4.2. 定义条件连接弧

转移条件是一个真值表达式，用于定义连接弧状态发生转移的规则。这个真值表达式可以包含五种要素：流程相关数据、比较运算符、常量、逻辑运算符(and/or)、优先级(小括号)。其中流程相关数据的书写格式为：$R{相关数据 ID}，常量的书写格式为：{常量值}，比较运行符可以为：“=”、“!=”、“>=”、“<=”、“include”、

“<”、“>”。

如一个相关数据为风险级别，其定义 ID 为 riskLevel，若定义一个风险级别大于 5 小于 1 0 的连接弧定义则为”{5}<$R{riskLevel}<{10}”。

3.1.5. 应用程序

3.1.5.1. 概述

应用程序是工作流过程定义中或者其所在包中的工作流过程需要或调用的应用程序。在 IS-Flow 工作流元模型中可以指定应用类型为普通应用、B/S 方式调用、JAVA 静态方法调用、JSF 方式调用，其中 JSF 方式调用属于 B/S 调用的特殊类型。

应用程序的属性有标识号、名字、描述以及对应的 0 或多个形参，0 或者多个扩展属性。

当执行人为系统执行人时，该活动不需要人工干涉，其业务单元只能是通过工具代理执行的自动程序。

以下为应用程序的基本属性，除此之外，应用程序还可以对应的 0 或多个形参，0 或者多个扩展属性。

表格 3-7 应用程序属性信息

应用程序（Package-Application）		
属性名	是否允许为空	描述
标识	否	
名称	否	
描述	是	
形参	是	
扩展属性	是	
应用类型	否	选择是“业务单元”或是“操作”

作用范围	否	选择全局（整个包）还是具体某一个流程

3.1.5.2. 定义业务单元与操作

所有的应用程序按照实现的功能可以分为业务单元、操作和事件调用三种类型。

业务单元是活动的具体内容，业务单元的完成通常需要调用外部应用实现。

操作是应用对 IS-Flow 所提供的流程控制性操作的封装。

同时业务单元之间可以定义一定的逻辑关系：业务单元的执行前提、业务单元的 start 影响和业务单元的 commit 影响。

业务单元的执行前提，是指要想当前业务单元执行必须满足的前提条件。例如，可以定义某些业务单元做完之后才可执行当前业务单元。

业务单元的 start 影响，是指执行某个应用的 startApplicationInstance("appInstId") 方法的时候，会对其他应用产生定义的影响。例如：可以定义当前应用的 start 影响为 使某个应用状态改变为完成状态，这样当这个应用执行 startApplicationInstance("appInstId")方法时，指定的应用就会改变为完成状态；

业务单元的 commit 影响，是指执行某个应用的 completeApplicationInstance("appInstId")方法的时候，会对其他应用产生定义的影响；例如，可以定义当前应用的 commit 影响为使某个应用状态改变为未启动状态，这样当这个应用执行 completeApplicationInstance("appInstId")方法时，指定的应用就会改变为未启动状态。

3.1.5.3. 定义 Java 方法调用

将应用定义为 JAVA 方法调用时，需要指定 JAVA 类名和方法名。如果一个工具活动包含一个指向该应用的工具，则在运行阶段，则可通过调用 WfClient 接口的 invoke()方法触发该应用。

3.1.5.4. 定义 WebService 方法调用

将应用定义为 WebService 方法调用时，需要指定 WSDL 服务地址、类库文件所在地址、所调用类的全名和方法名。例如：一个活动定义包含一个指向该应用的事件，则在运行阶段，引擎可在指定事件触发时调用该应用；同样流程和工作项也可定义指向该应用的事件。

3.1.5.5. 定义 B/S 方式的调用

将应用定义为 B/S 方式调用时，需要指定 URL 地址。该 URL 地址可以为 JSP 地址，Servlet 地址或者 Struts 等框架的*.do。如果一个工具活动包含一个指向该应用的工具，则在运行阶段，可通过获取定义对象的方法获得该 URL 地址，并设置到应用的工作台中。

3.1.5.6. 定义 EJB 调用

将应用定义为 EJB 调用时，需要指定 EJB 类型（2.0 还是 3.0）、EJB 应用服务器类型、应用服务器 URL 地址、JNDI 名称、接口名和方法名。例如：一个活动定义包含一个指向该应用的事件，则在运行阶段，引擎可在指定事件触发时调用该应用；同样流程和工作项也可定义指向该应用的事件。

3.1.5.7. 定义规则引擎调用

将应用定义为规则引擎调用时，需要指定规则和所调用工具代理类的详细路径（包含包名称），并且指定形式参数的标识和模式。例如：一个“自动活动”定义包含一个指向该应用的工具，则在运行阶段，引擎会自动调用该应用，并根据传递进来的相关数据，通过指定的规则计算出结果，将结果反馈给流程。

3.1.5.8. 定义普通应用

普通应用作为一个通用的应用程序，支持通过扩展属性的方式实现其他应用。默认可以配置应用的名称、标识和工具代理实现类，其他所需要的参数可以以扩展属性和形式参数的方式进行配置，其执行过程由工具代理类进行处理。

3.1.6. 形式参数

在工作流定义和工作流应用程序中，形式参数可以作为属性来使用。形式参数在调用时被传递，并且返回控制句柄。IS-Flow 引擎目前支持基本类型的“java.lang.String”，“java.lang.Integer”，“java.lang.Float”，“java.sql.Date”，“java.sql.Timestamp”，“java.util.Date”，“java.lang.Boolean”7 种，后续版本将会支持更多的类型。

引用形参的对象有流程定义，包内定义的应用程序，流程内定义的应用程序等。

形参的属性有：标识号 ID，索引，模式，描述和数据类型。

表格 3-8 形式参数属性信息

形参（FormalParameters）		
属性名	允许为空	描述
标识	否	参数的唯一标识
索引	否	注意：索引只能使用数字，参数的排列顺序
模式	否	见下
描述	是	
数据类型	是	目前支持基本类型的 7 种： “java.lang.String”,“java.lang.Integer”, “java.lang.Float”, “java.sql.Date”, “java.sql.Timestamp”, “java.util.Date”, “java.lang.Boolean”

3.1.7. 相关数据

流程相关数据就是流程定义自身的相关数据定义，流程定义的相关数据只能被流程定义自身所使用，如果没有定义则使用包定义的相关数据。需要注意的是，如果使用了流程定义的相关数据，那么在数据传递的过程中必须保证数据类型的一致。

表格 3-9 相关数据属性信息

流程—相关数据（Process—DataFields）		
属性名	允许为空	描述
标识	否	数据域的标志号
名称	否	数据域的名字
数据类型	否	数据类型
描述	是	数据域描述信息
流程—数据域—扩展属性（Process—DataFields—ExtendedAttributes）		
属性名	允许为空	描述
名称	是	扩展属性名字
值	是	扩展属性值

3.1.8. 工具活动

当一个活动的实现类型是工具时，可包含 1 个或者多个应用程序，且应定义每个“活动-工具”的实参集和扩展属性。

表格 3-10 工具活动属性信息

活动-工具（Activity-Tool）		
属性名	允许为空	描述
应用标识	否	从系统定义的应用列表中选择
形参集	是	自动从应用中获取
实参集	是	从相关数据列表中选择

描述	是	
扩展属性集		
属性名	允许为空	描述
名称	是	
值	是	

3.1.9. 资源

3.1.9.1. 概述

资源即组织模型，是工作流应用业务系统用来定义企业中人的组织形式的模型。在 WfMC 规范中，并未提供一个灵活的、足够强大的组织模型结构。而只是定义了参与者及参与者类型。

在 IS-Flow 中执行人的定义是由应用实现，应用可以自由的定义自己的执行人，在使用工作流的过程中只需要实现 IS-Flow 的人员管理接口为 IS-Flow 的运转提供所需的应用人员信息。人员管理接口为：

com.cvicse.workflow.api.resources.AbstractResourceProvider

3.1.9.2. 定义角色类型

在设计器中，新建一个类型为 Role 的资源，在活动任务分配处选择角色，角色 ID 即为活动资源 ID。

IS-Flow 在活动任务分配处支持动态指定角色，定义方法如下：

首先，在设计器中相关数据定义里新增一个 String 类型相关数据；例如：roleId，以便在应用编程中动态指定其值。

然后，在设计器中资源定义里新增一个角色(Role 类型)资源，格式如下：$R{roleId}，其中 roleId 为事先定义好的相关数据定义。

最后，在设计器中活动的任务分配里，选择单选框“角色”，并在其下拉框中选中上一步定义的角色“$R{roleId}”。

按照此步骤定义的流程，在应用编程中就可以通过为该相关数据赋值，来达到动态指定角色目的。

3.1.9.3. 定义执行人表达式类型

在设计器中，活动任务分配处选择“执行人表达式”（注：系统会为此活动默认生成资源 ID 为 Express，类型为 SYSTEM），编辑执行人表达式，例如：$P{id} = $S{Process.Creator}，即此活动的资源 ID 与流程的创建者 ID 相同。

3.1.9.4. 定义角色表达式类型

在设计器中，新建一个类型为 Role 的资源，在活动任务分配处选择角色，并且选择“执行人表达式”，即此活动的资源 ID 为角色中包含的执行人并且符合执行人表达式的所有 ID。

3.1.9.5. 定义单个执行人类型

在设计器中，新建一个类型为 Human 的资源，在活动任务分配处选择执行人，执行人 ID 即为活动资源 ID。用户可以为其指定个性化参数（邮箱，手机和电话），还可以为其定义执行期限，当到达指定期限并且还没有完成分配的任务时，则执行相应的事件。

IS-Flow 在活动任务分配处支持动态指定执行人，定义方法如下：

首先，在设计器中相关数据定义里新增一个 String 类型相关数据，例如：userId，以便在应用编程中动态指定其值。

然后，在设计器中资源定义里新增一个执行人（Human 类型）资源，格式如下：$R{userId}，其中 userId 为事先定义好的相关数据定义。

最后，在设计器中活动的任务分配里，选择单选框“单个执行人”，并在其下拉

框中选中上一步定义的执行人“$R{userId}”。

按照此步骤定义的流程，在应用编程中就可以通过为该相关数据赋值，来达到动态指定执行人的目的。

同时，IS-Flow 在活动任务分配处还支持利用流程实例的系统属性来指定活动的执行人，定义方法如下：

首先，在设计器中资源定义里新增一个执行人（Human 类型）资源，格式如下：$S{Process.Creator}，其中 Process.Creator 为流程实例的一个系统属性，其意义为流程实例创建人；流程实例系统属性还包括 Activity[actDefId].Performer 等。

最后，在设计器中活动的任务分配里，选择单选框“单个执行人”，并在其下拉框中选中上一步定义的执行人“$S{Process.Creator}”。

以上描述的功能均可体现在流程设计器中，关于流程设计器的详细使用参见***《InforSuite Flow 流程设计器使用手册》***。

3.2. 流程定义导入导出工具

IS-Flow 提供了一个命令行工具可以将流程定义批量导入数据库或从数据库导出。

3.2.1. 功能简介

在 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\infor\flow\bin\下有一个名为 import.bat 的文件(Linux/Unix 系统下为 import.sh 文件)，该文件主要有以下三个功能：

1. 创建工作流表结构：根据 hibernate 的数据库映射文件创建数据库中工作流的表结构。

2. 导入、导出、查看、卸载流程定义和流程依赖项，将流程定义文件中的流程定义导入到数据库中。IS-Flow 遵循 WfMC 组织规定的 XPDL 规范，在定义阶段，可以使用设计器定义 XML 结构的流程定义，在运行阶段，需要首先将流程定义导入数据库

中。用户还可以将导入后 IS-Flow 引擎中的流程定义和流程依赖项导出到 XPDL 文件中，以便对其进行查看和修改。

3. 启用禁用流程定义模板：提供对流程定义模板的有效性进行控制的方法，以控制把哪些流程定义模板加载到工作流引擎中。如果启用了某个流程定义模板，则该流程定义将被加载到引擎中，当前的状态为未被实例化状态的流程定义模板可以用于用户创建新的流程实例；当前状态为已被实例化的流程定义模板，则只能被以前创建的流程实例所使用，而不能够再创建新的流程实例。如果禁用了某个流程定义模板，则该流程定义将不被加载到引擎中，也就不能够再创建新的流程实例，而不管它当前的状态是已被实例化还是未被实例化。

3.2.2. 配置说明

3.2.2.1. 配置数据库连接

进入 IS-Flow 产品安装目录%INFORSUITE_HOME%\infor\flow\conf\，在 inforflow.xml 文件中配置数据库连接，见表格 3-2 流程定义的属性列表。

```
<dataBaseProperties>
    <prop key = "driverClassName" >COM.ibm.db2.jdbc.net.DB2Driver</prop>
    <prop key = "url" >jdbc:db2://localhost/flow</prop>
    <prop key = "username">db2admin</prop>
    <prop key = "password">db2</prop>
</dataBaseProperties>
```

图表 3-2 inforflow.xml 中数据库连接配置

在 dataBaseProperties 元素中，填写正确的数据库连接参数。可参考表格 3-11 数据库连接属性：

表格 3-11 数据库连接属性

参数	介绍	举例
url	数据库地址	jdbc:db2://localhost/flow

driverClassName	驱动程序	COM.ibm.db2.jdbc.net.DB2Driver
username	用户名	db2admin
password	密码	db2admin

工作流内部数据库连接池实现为 Apache DBCP，故相应参数也可参照 DBCP 的 org.apache.commons.dbcp.BasicDataSource 的 bean 配置，bean 配置参照网页：

http://jakarta.apache.org/commons/dbcp/apidocs/org/apache/commons/dbcp/BasicDataSource.html。

3.2.2.2. Hibernate 属性配置

流程定义导入导出工具使用无事务模式，即在 IS-Flow 产品安装目录 %INFORSUITE_HOME%\infor\flow\conf\ 下的 iniforflow.xml 配置文件中设置 <transaction defaultTxType = "NoTxWorkflowContext">，在 NoTxWorkflowContext 处设置 Hibernate 的参数，见图表 3-3 Hibernate 属性配置：

```
<!--无事务-->
<workflowContext txType ="NoTxWorkflowContext">
    <hibernateProperties>
        <prop key = "hibernate.dialect" >org.hibernate.dialect.DB2Dialect</prop>
        <prop key = "hibernate.show_sql">true</prop>
        <prop key = "hibernate.hbm2ddl.auto">create</prop>
    </hibernateProperties>
</workflowContext>
```

图表 3-3 Hibernate 属性配置

有关 Hibernate 属性的配置，见表格 3-12。

表格 3-12 Hibernate 属性信息

参数	介绍	举例
Hibernate.dialect	选择使用的数据库方言	org.hibernate.dialect.DB2Dialect

Hibernate.show_sql	是否在控制台上输出 sql 语句	True

其他配置请参照 Hibernate Reference：

http://www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html#configuration-optional。

3.2.2.3. 命令行工具客户端连接属性配置

在 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\infor\flow\classes\下 inforflow-Manager-client.xml 配置文件的/InforFlowManagerClient /@parent 处设置客户端的连接方式：本地连接“localManagerClient”和 RMI 方式连接“rmiManagerClient”，如果是本地连接，则不用再单独启动引擎服务；如果设置 RMI 方式连接，则首先需要设置好 rmiManagerClient 的连接属性：服务地址、端口号和服务名称，并需要预先启动好\rmiManagerClient 中设置的引擎服务。配置文件信息见图表 3-4 命令行工具客户端属性配置：

```
<bean id="localManagerClient"
     class="com.cvicse.workflow.api.client.local.LocalManagerClient" >
</bean>

<bean id="rmiManagerClient"
    class="com.cvicse.workflow.api.client.rmi.WfRMIManagerClient"
                    destroy-method="close" init-method="init" lazy-init="true">
    <property name="hostUrl">
        <value>localhost</value>
    </property>
    <property name="hostPort">
        <value>5501</value>
    </property>
    <property name="serviceName">
        <value>InforFlowManagerClient</value>
    </property>
</bean>

<bean id="InforFlowManagerClient" parent="localManagerClient">
<!--bean id="InforFlowManagerClient" parent="rmiManagerClient"-->
</bean>
```

图表 3-4 命令行工具客户端属性配置

3.2.2.4. 命令行工具启动参数配置

命令行工具启动时，需要设置两个参数：XPDL 文件路径和重建表结构参数。如图 5-4 所示。

XPDL 文件路径(/WorkflowUtil/@xpdlRepository)指定用于存放待导入的流程定义文件的目录，可以以绝对目录和相对目录两种方式进行设置。绝对目录是指本地磁盘的绝对目录，相对目录是指相对%INFORSUITE_HOME%\FlowServer\infor\flow 的目录。 (注意：xpdl 文件名不能是中文，并且不能有空格，标识中不能存在特殊字符。)

重建表结构参数(/WorkflowUtil/@auto_create_db)用于命令行工具启动成功后提示用户是否要重建工作流表结构。引擎正常运行后，该参数推荐设置为 false，以免因误操作而清除引擎数据。

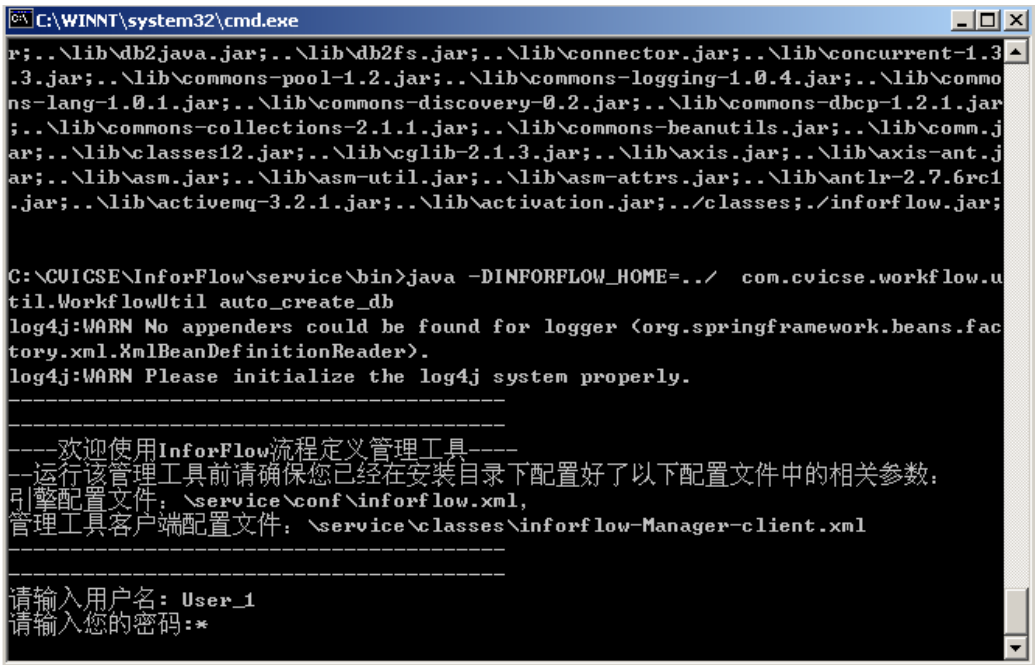
```
<bean id="WorkflowUtil" class="com.cvicse.workflow.util.WorkflowUtil"
                                    init-method="init" lazy-init="true">
    <property name="xpdlRepository">
        <value>xpdl</value>
    </property>
    <property name="auto_create_db">
        <value>true</value>
    </property>
</bean>
```

图表 3-5 重建工作流表结构的属性配置

3.2.3. 功能使用

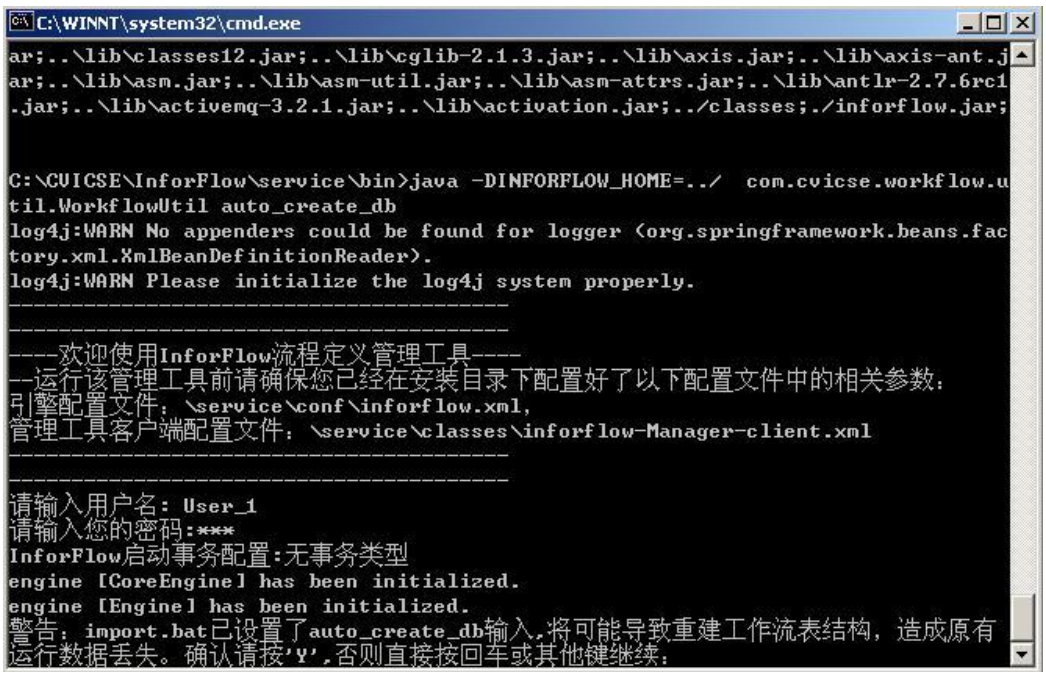
3.2.3.1. 用户登陆与重建工作流表结构

用户设置好以上的各项参数之后，运行 import.bat，将首先出现登录画面，见图表 3-6 用户登录界面：



图表 3-6 用户登录界面

这里的用户名和密码是有权使用工作流执行服务的用户的用户名和密码。输入正确的用户名和密码之后，如果用户将重建工作流表结构参数/WorkflowUtil/@auto_create_db 设置为 true，将出现重建表结构确认画面，见图表 3-7 创建工作流表结构。



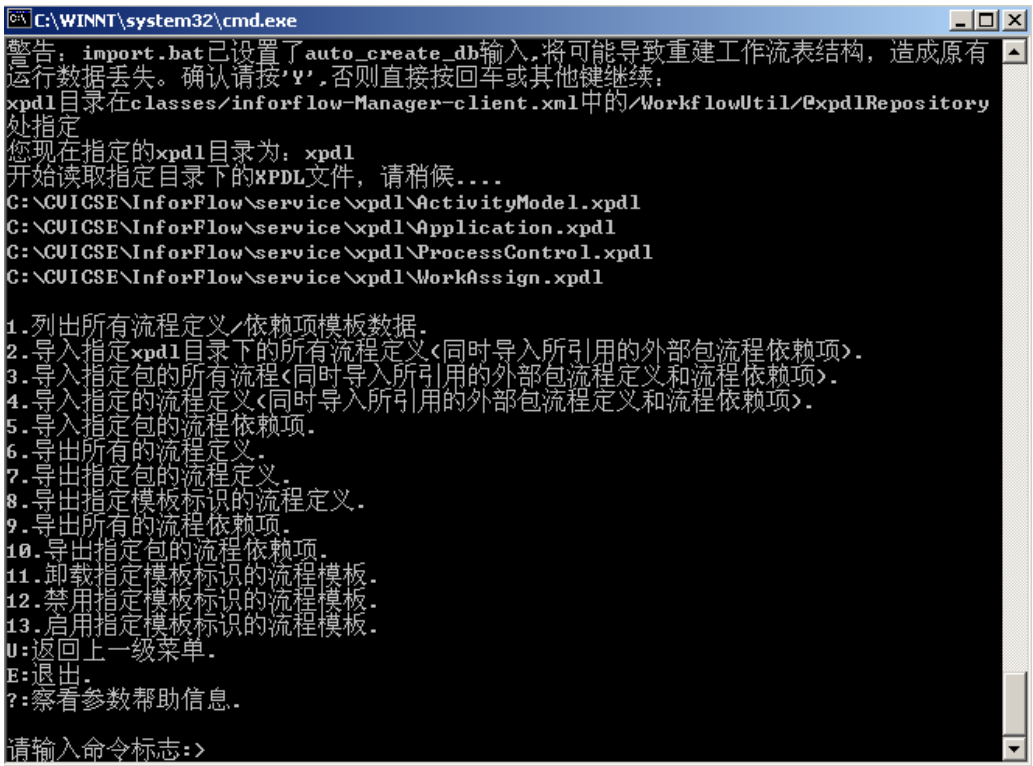
图表 3-7 创建工作流表结构

如果是以 local 方式启动“IS-Flow 流程定义导入导出工具”，按’Y’后，import.bat 将重建数据库结构。这里需要注意的是：重建表结构会致使原来的所有运行数据和模板数据全部丢失，在系统正常运行之后，该功能请慎用！

如果是以 rmi 方式启动“IS-Flow 流程定义导入导出工具”，则无重建表结构选项。

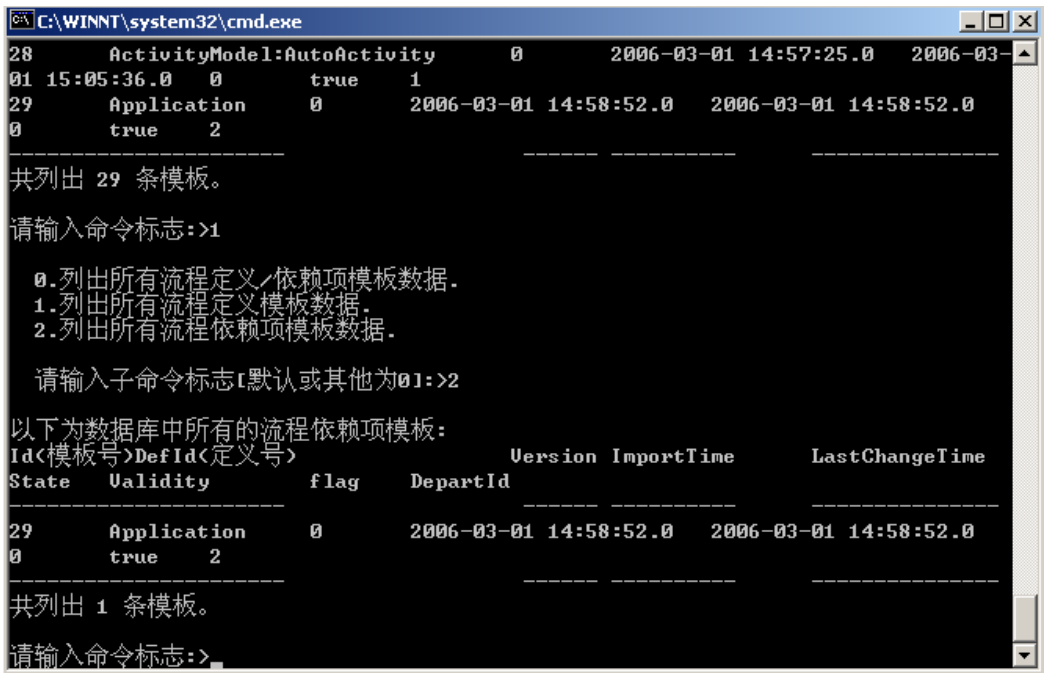
3.2.3.2. 列出所有流程模板数据

执行 import.bat，用户成功登录以后：将出现以下画面，在命令提示符“请输入命令标志:>”后输入“1/2/4/5/6/7/8/9/10/11/12/13/U/E/?”中的一条命令标志，按回车键后即可执行相应的命令。



图表 3-8 命令行工具的提示信息

在输入命令标志提示下，输入'1'后按回车键，将出现如下的二级命令，用户输入命令'1'可列出引擎数据库中所有的流程定义模板记录，输入命令'2'可列出引擎数据库中所有的流程依赖项模板记录，默认、用户输入'0'或者'1'和'2'以外的任意字符，将首先列出所有的流程定义，接着列出所有的流程依赖项模版记录。图表 3-9 列出所有的流程依赖项的示例：



图表 3-9 列出所有流程定义的依赖项

3.2.3.3. 导入流程定义、流程依赖项

3.2.3.3.1. 流程定义模板的版本控制原则

使用 import.bat 可以将指定文件夹中的部分或全部流程定义、流程依赖项导入到引擎数据库中，分别生成流程定义模板和流程依赖项模板。流程依赖项，即流程定义所依赖的外部包的资源、应用程序和相关数据的集合。一般来说，用户在导入流程定义时，如果该流程引用了其他流程，则所需要的流程依赖项会自动被导入到引擎数据库中，所以用户一般不需要单独导入流程依赖项。引擎会自动对导入后的流程定义模板进行版本控制，但不会对流程依赖项进行版本控制。版本控制的原则如下：

1. 流程定义导入后生成流程定义模板，由 IS-Flow 工作流引擎负责流程定义模板版本的控制。

2. 发起新流程时，相同的流程定义号可能有多条流程模板，首先按照流程定义模板的状态和版本标志查找出能够发起流程实例的(即未被实例化的)零条或者唯一的流程定义模板（这个流程模板的标识不一定是标识号最大的流程模板），如果找到了能够发起流程实例的流程定义模板，则发起流程实例。

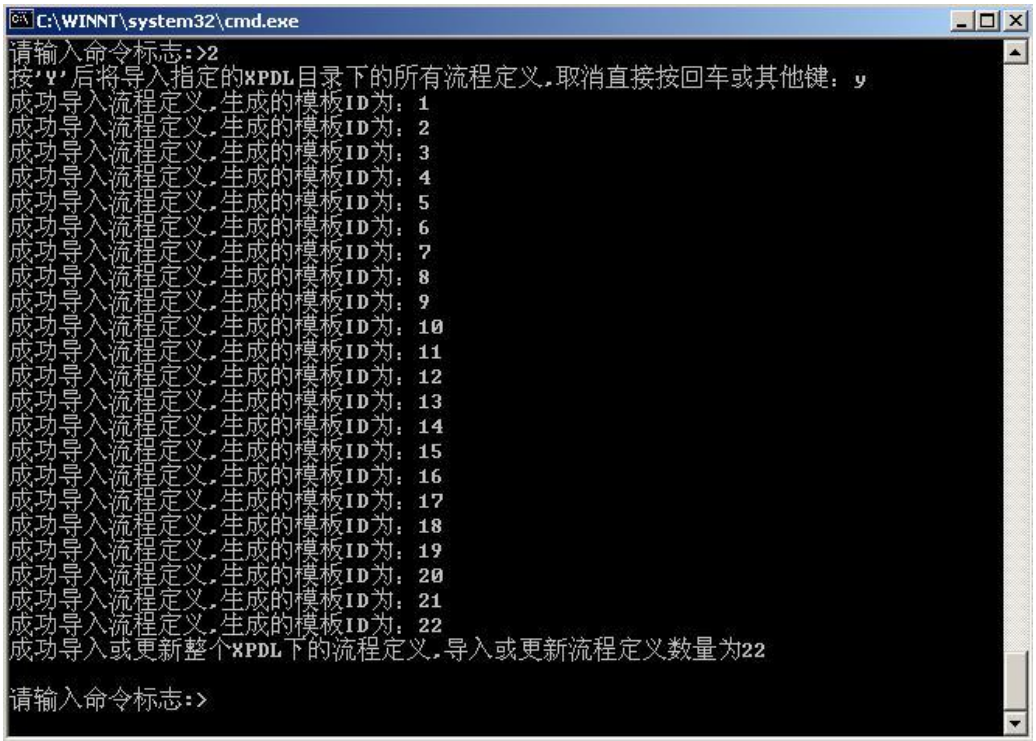
3. 相同的流程定义标识可能有多条流程模板，在导入流程定义时，若不存在该流程定义的模板，则新增加一条记录，否则首先找出该流程定义标识中可发起流程实例的流程模板，若该流程定义模板没有被流程实例引用，则将覆盖该流程定义模版；若已被流程实例引用，则首先按照其流程实例的个数为一个或多个，分别将该流程实例的标志字段设置成单实例化或多实例化状态，使其不能够再发起新的流程实例，但可以被旧的流程实例所使用，然后再增加一条新的流程定义模板。

4. 卸载流程模板时，如果流程模板已被一个或多个流程实例引用，那么工作流引擎会为此流程实例保留其流程模版的备份，否则将从工作流引擎数据库中直接删除。

5. 修改流程实例模板时，首先判断该流程模版被流程实例引用的个数，若只有一个流程实例引用该流程模板，则单实例化该流程模板。如果有多个流程实例引用该流程模板，则首先新建一条该流程模板的拷贝，并实例化该流程模板，然后将发起修改命令的流程实例的流程定义模板引用标识设置成新增流程模板的标识。

3.2.3.3.2. 导入指定的 XPDL 目录下的所有流程定义

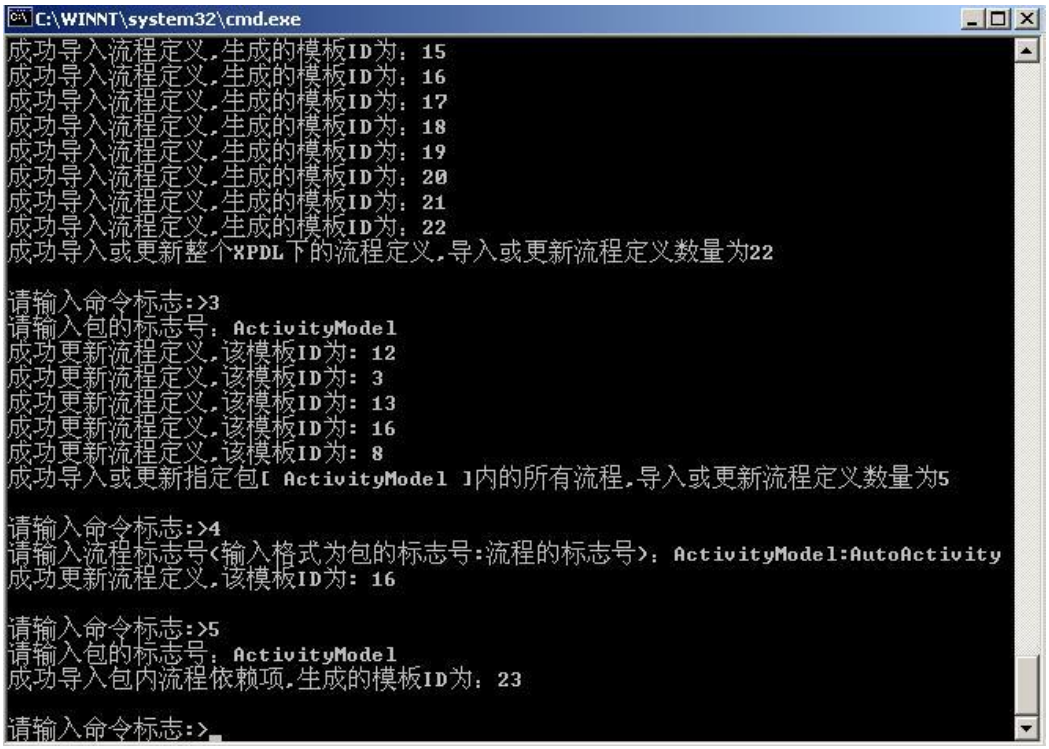
在输入命令标志提示下，输入’2’后按回车键，会有提示信息“按'Y'后将导入指定的 XPDL 目录下的所有流程定义,取消直接按回车或其他键：”，如果用户输入”Y”或”y”，按回车键则会将 inforflow-manager-client.xml 文件中指定的 xpdl 目录下的所有流程定义导入到数据库中，并自动导入所有外部包引用的资源，导入成功后，将提示成功导入的数量，如图表 3-10 所示。如果用户输入”Y”或”y”以外的任意字符或者不输入任何字符，按回车键后，系统会放弃该命令，并回到命令提示符。



图表 3-10 导入所有的流程定义

3.2.3.3.3. 导入指定包的所有流程

在输入命令标志提示下，输入'3'后按回车键，将提示输入包名，如图表 3-11 所示：



图表 3-11 导入流程定义或流程依赖项

输入正确的包标识或者该包的不带扩展名的文件名（如：ActivityModel）后按回车，指定包中所有流程定义、以及其所引用的外部包的流程定义和流程依赖项会一并导入到数据库中。导入成功后，将提示成功导入和导入后的模板标识，如图表 3-11 所示。

若指定的包标识不存在，或者指定包中没有任何流程定义，将出现提示：“指定包[……]不存在，或者没有任何流程定义”，其中省略号代表用户的输入。

3.2.3.3.4. 导入指定的流程定义

在输入命令标志提示下，输入’4’后按回车键，将提示输入流程定义标识号，该标识号格式为“流程定义所在包的包标识**:**流程定义表示”组合而成。

输入正确的流程定义名后按回车键，可将指定的流程定义、其所引用的流程定义和所引用的外部包的依赖项一并导入到数据库中。导入成功后，将提示成功导入的数量和导入后的模板标识，见图表 3-11 导入流程定义或流程依赖项。

若流程定义标志号不符合“包名：流程定义名”的格式，或者流程定义标志号虽然合法，但其指定的流程定义不存在，将出现提示：“指定的流程定义[……]不存在”，其中省略号代表用户的输入。

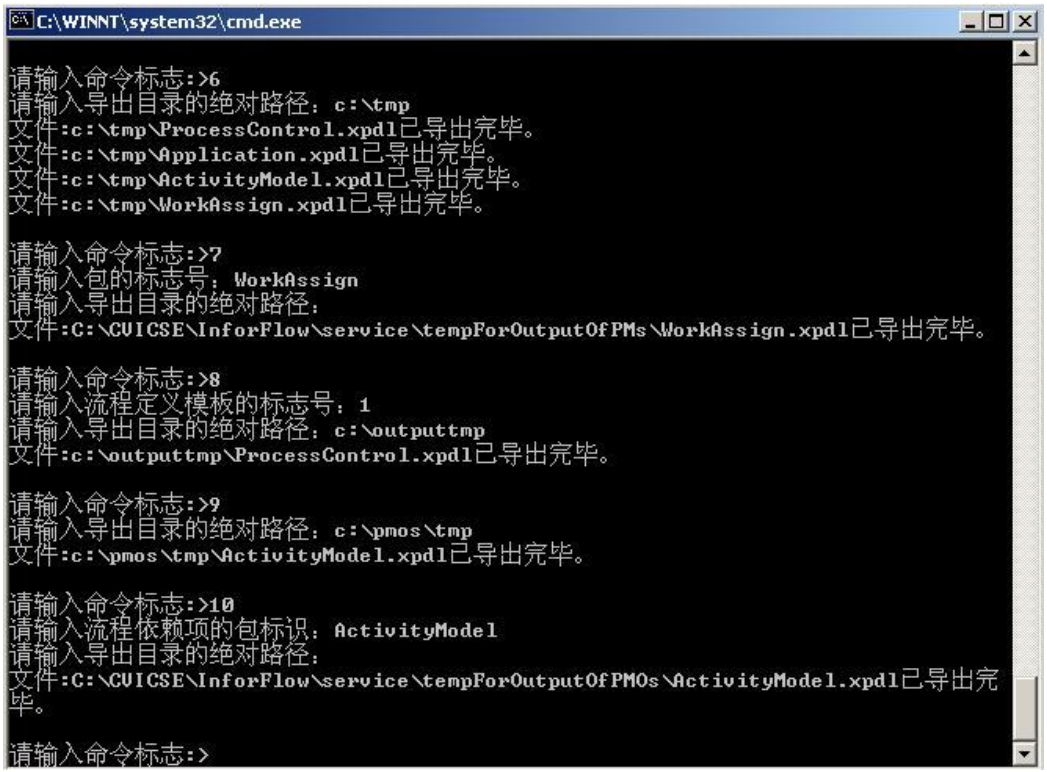
3.2.3.3.5. 导入指定包的流程依赖项

在输入命令标志提示下，输入’5’后按回车键，将提示输入包标识，输入正确的包标识或代表导入包的正确文件名（不带后缀）后按回车键，可将该包的流程依赖项导入到引擎数据库中，导入成功后，将提示成功导入和导入后的模板标识，见图表 3-11 导入流程定义或流程依赖项。

3.2.3.4. 导出流程定义、流程依赖项

3.2.3.4.1. 导出所有的流程定义

在输入命令标志提示下，输入’**6**’后按回车键，数据库中的所有流程定义会被导出到相应的文件中，系统会自动把导入前属于同一个包的流程定义导入到同一个文件中。用户可以输入导出文件的绝对目录，如果用户不输入任何路径而直接按回车，默认导出文件的位置为 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\infor\flow\tempForOutputOfPMs\中。导出成功后，将出现以下提示，见图表 3-12 导出流程定义或流程依赖项：



图表 3-12 导出流程定义或流程依赖项

3.2.3.4.2. 导出指定包的流程定义

在输入命令标志提示下，输入'**7**'后按回车键，将提示输入包标识，然后提示输入包的标识号，输入正确的流程定义标识后按回车键，会提示输入导出目录的绝对路径，用户可以输入导出文件的绝对目录，如果用户不输入任何路径而直接按回车，默认导出文件的位置为%INFORSUITE_HOME%\FlowServer\infor\flow\tempForOutputOfPMs\目录中。然后该包中所有的流程定义、所引用的流程定义和所引用的外部包的依赖项一并导出到相应文件中，导出成功后，将出现如图表 3-12 所示信息。如果用户输入的包标识不存在，则会提示：“指定的包[……]不存在流程定义模板！”其中，“……”

代表用户的输入信息。

3.2.3.4.3. 导出指定模板标识的流程定义

在输入命令标志提示下，输入'**8**'后按回车键，将提示输入流程定义模板标识，用户输入流程定义模板标识后按回车键，会提示输入导出目录的绝对路径，导出成功后，该流程定义便被导出到特定的文件中，并显示如图表 3-12 所示信息。

3.2.3.4.4. 导出所有的流程依赖项

在输入命令标志提示下，输入'**9**'后按回车键，数据库中的所有流程依赖项会被导出到相应的文件中。用户可以输入导出文件的绝对目录，如果用户不输入任何路径而直接按回车，默认导出文件的位置为 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\infor\flow\tempForOutputOfPMs\中（注意：该默认导出目录与流程定义的默认导出目录很接近）。导出成功后，将出现如图表 3-12 所示信息。

3.2.3.4.5. 导出指定包的流程依赖项

在输入命令标志提示下，输入'**10**'后按回车键，将提示输入流程依赖项的包标识，然后提示输入导出目录的绝对路径，导出成功后，会出现如图表 3-12 所示信息。如果用户输入的包标识不存在，则会提示：“指定的流程依赖项[……]不存在！”其中，“……”代表用户的输入信息。

3.2.3.5. 卸载流程模板

在输入命令标志提示下，输入'**11**'后按回车键，将提示输入待卸载的流程模板标识，用户输入待卸载的流程模板标识后按回车键，卸载成功后出现的信息见图表 3-13 卸载流程模板：

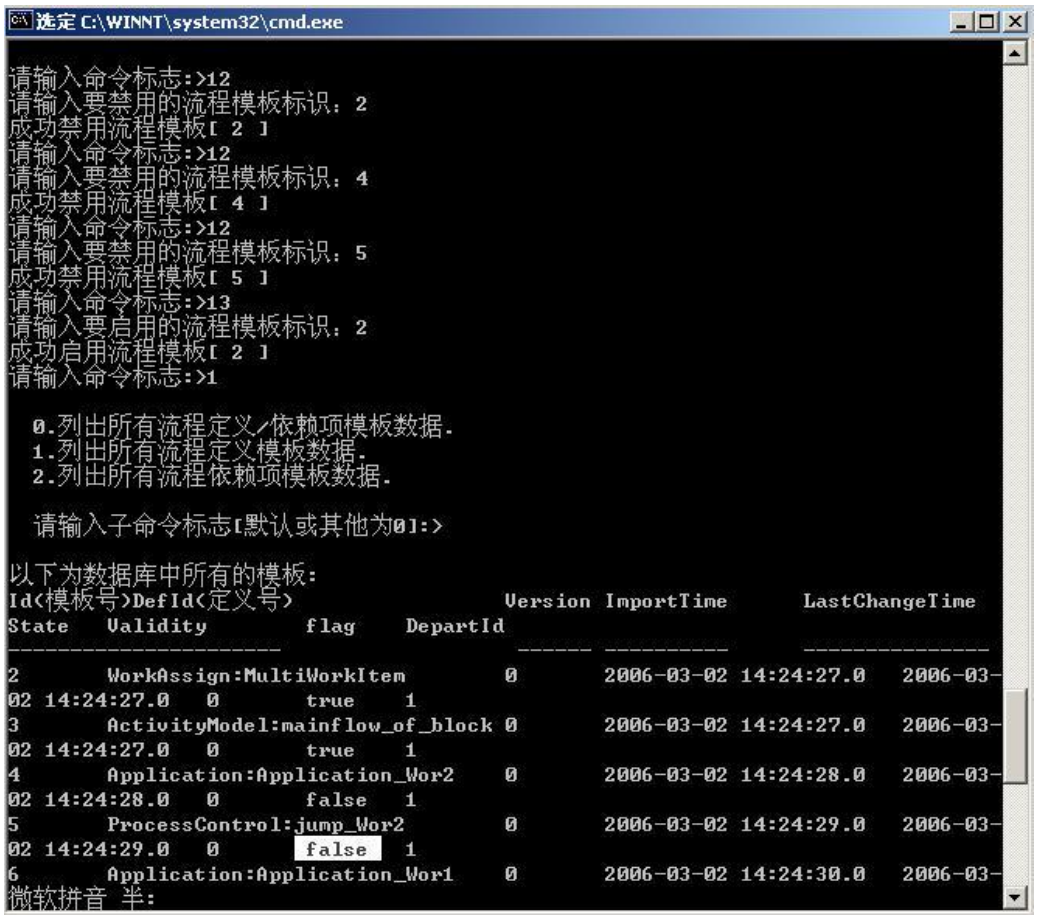
图表 3-13 卸载流程模板

卸载某流程模版时，如果流程模板已被一个或多个流程实例引用，那么工作流引擎会为此流程实例保留其流程模版的备份，否则将从工作流引擎数据库中直接删除，如图表 3-13 所示。流程模版的备份使用其 state 属性作为标志位，而且其将不能再创建新的流程实例，但可以被已有的流程实例所使用。

3.2.3.6. 启用禁用流程模板

3.2.3.6.1. 禁用流程模板

在输入命令标志提示下，输入'**12**'后按回车键，将提示输入待禁用的流程模板标识，用户输入待禁用的流程模板标识后按回车键，禁用成功后会出现的信息见图表 3-14 流程模板的启用禁用：



图表 3-14 流程模板的启用禁用

禁用成功后，用户输入命令”**1**”列出所有的流程模板，可看到该禁用的模板的 flag 标志位为”false”，如图表 3-14 中高亮所示。流程模板被禁用代表着该流程模板已被禁用，它既不能发起新的流程实例，也不能被其他流程实例所引用。

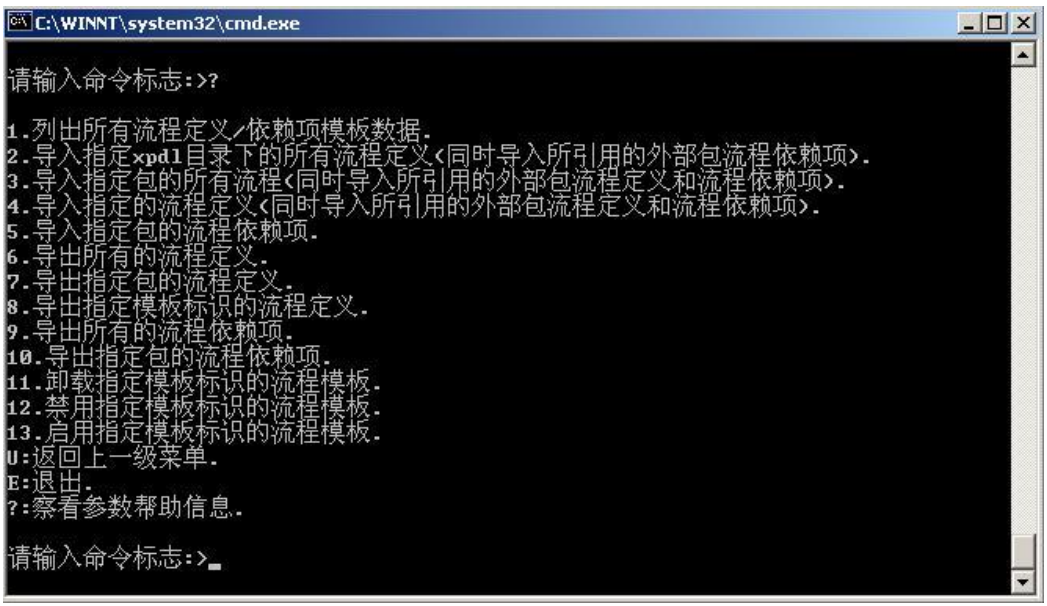
3.2.3.6.2. 启用流程模板

在输入命令标志提示下，输入’**13**’后按回车键，将提示输入待启用的流程模板标

识，用户输入待启用的流程模板标识后按回车键，启用成功后会出现如图表 3-14 所示信息。

3.2.3.7. 显示帮助信息

在输入命令标志提示下，输入'**?**'后按回车键，将显示帮助信息，即命令列表，见图表 3-15 帮助信息：



图表 3-15 帮助信息

第4章 IS-Flow 流程部署运行

本章介绍 IS-Flow 如何部署流程定义、如何通过标准任务表处理器运行流程。

学习本章内容后，您可以：

- 了解 IS-Flow 通过三种方式部署流程定义，并结合相关参考文档进行流程定义的部署。
- 结合《*InforSuite Flow 标准任务表处理器使用手册*》运行流程。

4.1. 流程部署

IS-Flow 流程部署是指将流程定义（即 xpdl 文件）通过工具导入到指定数据库的 wr_model 表中。一个流程定义对应 wr_model 表的一条记录，如果一个流程包含子流程，那么子流程也对应 wr_model 表的一条记录。

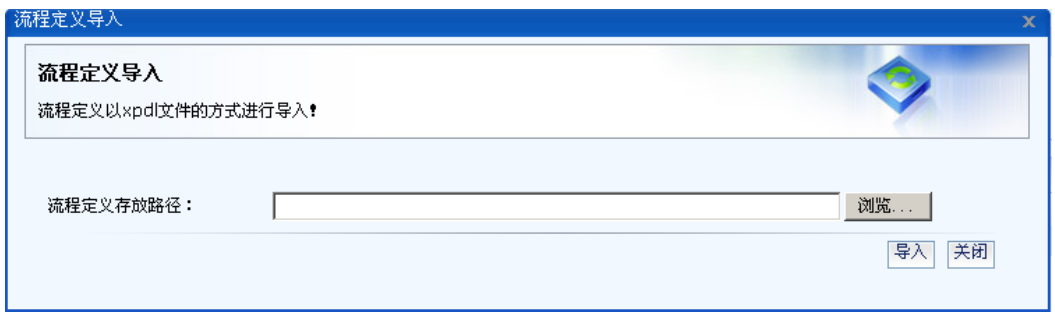
目前，IS-Flow 提供三种方式的流程部署工具，即 B/S 方式、C/S 方式和命令行方式。其中 B/S 方式由 IS-Flow 管理平台实现，C/S 方式由 IS-Flow 流程设计器实现，命令行方式由 IS-Flow 流程定义导入导出工具实现。

4.1.1. B/S 方式

进入 IS-Flow 管理平台，进入“流程定义管理”界面，控制面板如图表 4-1 所示：

图表 4-1 IS-Flow 管理平台流程定义管理界面控制面板

点击“导入”按钮，如图表 4-2 所示：

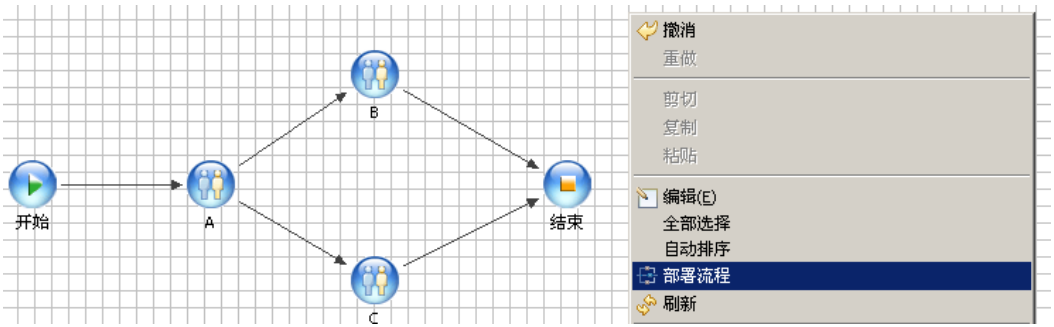


图表 4-2 流程定义导入界面

点击“浏览”，选择相应路径下的流程定义后点击“导入”按钮，导入完成。

4.1.2. C/S 方式

业务流程建模完成后，可以通过流程设计器直接部署流程定义到数据库中，如图表 4-3 所示：



图表 4-3 流程设计器中部署流程

关于流程设计器中部署流程的详细介绍参见***《InforSuite Flow 流程设计器使用手册》***。

4.1.3. 命令行方式

IS-Flow 提供了命令行方式部署流程定义。这种方式可以批量导入流程定义，关于它的详细使用参见导入流程定义、流程依赖项。

4.2. 流程运行

IS-Flow 工作流对象（包括流程实例、活动实例和工作项实例）的生命周期如 1.3.3 节中的图表 1-2 所示。标准任务表处理器提供了工作流对象整个生命周期的运行环境。

标准任务表处理器是一个 web 应用，位于产品安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\tasklist\下。关于部署 tasklist.war 及其相应的配置详见***《InforSuite Flow 标准任务表处理器使用手册》***，这里只展示其运行时的效果。

标准任务表处理器应用启动后，进入登陆页面，输入用户名 User_1，不用输入密码，如图表 4-4 所示：

图表 4-4 登录标准任务表处理器

进入标准任务表处理器之后，左侧菜单栏显示标准任务表处理器的功能列表，如任务查询、新建流程、综合查询和管理监控等。首先需要创建流程实例，点击“创建流程实例”，页面中显示流程定义列表，从中选择一个流程定义，并输入流程实例名（可为空）、机构代码（可为空）等信息，点击“创建”按钮，创建流程实例，如图表 4-5 所示：

流程定义名称	流程定义ID	流程模板ID	流程定义描述
◉ 测试抢任务	WorkAssign:MultiAssign	16	
○ 多个活动	multiact_complex:multiact	14	测试子流程活动和块活动
○ 综合查询	ClientService:query	13	
○ blockActTrans	testBlockActTrans:blockActTrans	12	
○ SubFlow_2	multiact_complex:SubFlow_2	11	
○ 单元测试	junit_test:junit_test	8	
○ 顺序流程	WorkflowPattern:Sequence	6	
○ 分支流程	WorkflowPattern:Branch	4	
○ SubFlow_2	multiact:SubFlow_2	3	
○ 多个活动	multiact:multiact	1	
流程实例名：	测试抢任务	机构代码：	
创建 启动 取消			

图表 4-5 创建流程实例

流程实例创建成功后，会显示如图表 4-6 所示的提示信息：

图表 4-6 流程实例创建成功后的提示信息

流程实例创建成功后，需要启动流程实例，如图表 4-7 所示：

实例名称	实例ID	发起人	创建时间	状态	定义标识	启动	图形启动	定时启动
测试抢任务	944	User_1	2009-05-18 13:16:10	待处理	WorkAssign:MultiAssign			
testListProcessInstanceAttributes	430	system	2009-04-10 15:35:12	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	429	system	2009-04-10 15:34:02	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	427	system	2009-04-10 15:05:33	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	426	system	2009-04-10 14:52:42	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	422	system	2009-04-10 14:29:46	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	217	system	2009-04-10 13:25:38	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	216	system	2009-04-10 11:56:44	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	215	system	2009-04-10 11:52:17	待处理	WorkflowPattern:Branch			
testListProcessInstanceAttributes	214	system	2009-04-10 11:51:56	待处理	WorkflowPattern:Branch			

图表 4-7 待启动的流程实例列表

在流程实例列表中启动创建成功的流程实例。启动成功后，可以进行流程实例的查询、管理、完成工作项等操作。

此外，标准任务表处理器与 InforSuite SSO 结合实现了单点登录、认证授权等功能。

关于标准任务表处理器的详细介绍参见***《InforSuite Flow 标准任务表处理器使用手册》***。

第5章 IS-Flow 流程管理监控

本章介绍 IS-Flow 的流程管理监控功能。IS-Flow 提供两种方式的流程管理监控，即 B/S 方式和 C/S 方式。其中 B/S 方式由 IS-Flow 管理平台实现，C/S 方式由 IS-Flow 开发平台的流程仿真调试工具完成。

学习本章内容后，您可以：

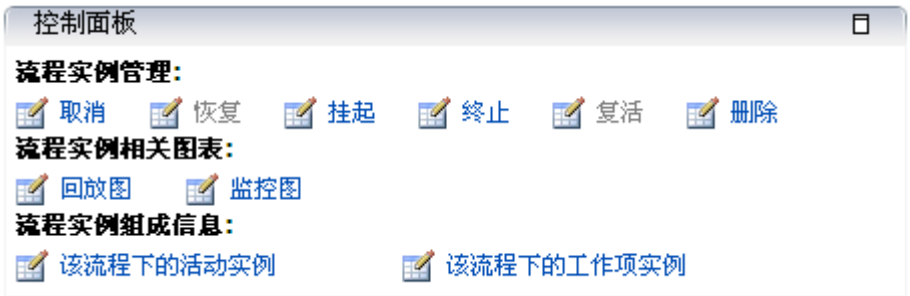
- 了解 IS-Flow 通过 B/S 和 C/S 两种方式进行流程的管理监控，关于如何使用工具进行流程的管理监控参见相应的使用手册。

5.1. B/S 方式

IS-Flow 管理平台提供了 B/S 方式的流程管理监控。其中流程管理包括流程定义管理和流程实例管理，流程监控包括流程实例回放图、监控图等。流程定义管理控制面板见图表 5-1 所示，图表中的每一个按钮都对应相应的管理功能。

图表 5-1 流程定义管理控制面板

流程实例管理控制面板见图表 5-2 所示，图表中的每个按钮都对应相应的管理或监控功能。



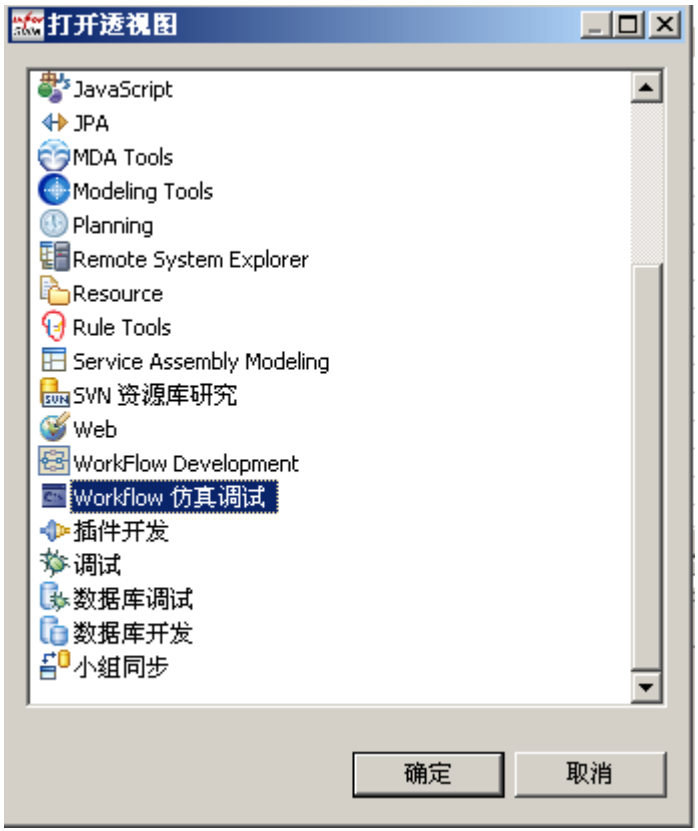
图表 5-2 流程实例管理控制面板

关于 B/S 方式流程管理监控的详细介绍参见***《InforSuite Flow 管理监控工具使用手册》***。

5.2. C/S 方式

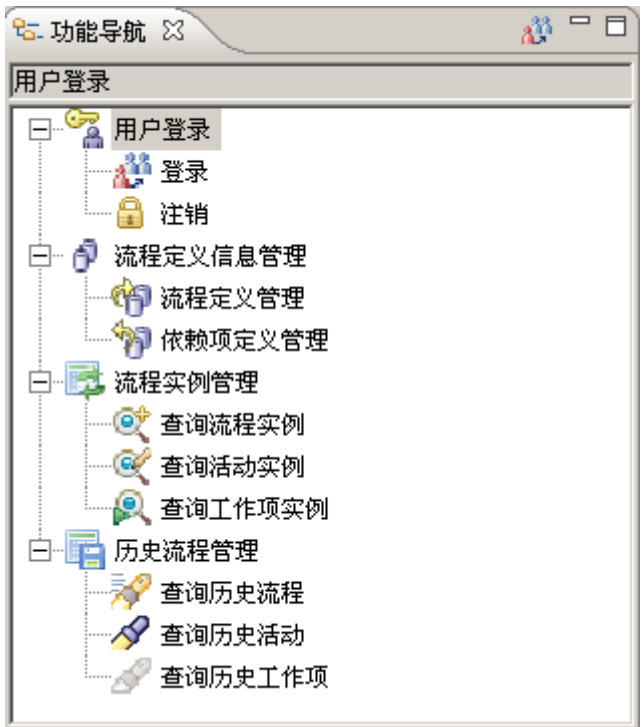
IS-Flow 开发平台提供了 C/S 方式的流程管理监控，即仿真调试工具。

打开 IS-Flow 流程设计器“Workflow 仿真调试”视图，



图表 5-3 IS-Flow 开发平台“打开透视图”窗口

开发平台功能导航窗口见图表 5-4 所示：



图表 5-4 功能导航

在功能导航窗口中，每个菜单都对应相应的管理功能。点击“流程定义管理”，流程定义列表如图表 5-5 所示：

状态	模板标识	流程标识	流程名称	机构标识	模板类型	导入时间	最后修改时间	生效日期	失效日期
	16	WorkAssign:MultiAssign	测试抢任务		文件	2009-04-09 15:54:17	2009-04-09 15:54:17.0		
	14	multiact_complex:multiact	多个活动		文件	2009-04-09 15:54:17	2009-04-09 15:54:17.0	2008-11-01	2008-11-01
	13	ClientService:query	综合查询		文件	2009-04-09 15:54:16	2009-04-09 15:54:16.0		
	12	testBlockActTrans:blockActTrans	blockActTrans		文件	2009-04-09 15:54:16	2009-04-09 15:54:16.0	2008-10-28	2008-10-28
	11	multiact_complex:SubFlow_2	SubFlow_2		文件	2009-04-09 15:54:16	2009-04-09 15:54:16.0		
	8	junit_test:junit_test	单元测试		文件	2009-04-09 15:54:15	2009-04-09 15:54:15.0	2008-09-24	2008-09-24
	6	WorkflowPattern:Sequence	顺序流程		文件	2009-04-09 15:54:15	2009-04-09 15:54:15.0		
	4	WorkflowPattern:Branch	分支流程		文件	2009-04-09 15:54:15	2009-04-09 15:54:15.0		
	3	multiact:SubFlow_2	SubFlow_2		文件	2009-04-09 15:54:14	2009-04-09 15:54:14.0		
	1	multiact:multiact	多个活动		文件	2009-04-09 15:54:13	2009-04-09 15:54:13.0	2008-11-01	2008-11-01

图表 5-5 流程定义管理

在流程定义列表中，单击鼠标右键，可以进行对流程定义的管理，如导出、卸载、禁用等。

状态	模板标识	流程标识	流程名称	机构标识	模板类型	导入时间	最后修改时间	生效日期	失效日期
	16	WorkAssign:MultiAssign	测试抢任务		文件	2009-04-09 15:54:17	2009-04-09 15:54:17.0		
	14	multiact_complex:multiact	多个活动		文件	2009-04-09 15:54:17	2009-04-09 15:54:17.0	2008-11-01	2008-11-01
	13	ClientService:query	综合查询			2009-04-09 15:54:16	2009-04-09 15:54:16.0		
	12	testBlockActTrans:blockActTra	blockActTra			2009-04-09 15:54:16	2009-04-09 15:54:16.0	2008-10-28	2008-10-28
	11	multiact_complex:SubFlow_2	SubFlow_2			2009-04-09 15:54:16	2009-04-09 15:54:16.0		
	8	junit_test:junit_test	单元测试			2009-04-09 15:54:15	2009-04-09 15:54:15.0	2008-09-24	2008-09-24
	6	WorkflowPattern:Sequence	顺序流程			2009-04-09 15:54:15	2009-04-09 15:54:15.0		
	4	WorkflowPattern:Branch	分支流程			2009-04-09 15:54:15	2009-04-09 15:54:15.0		
	3	multiact:SubFlow_2	SubFlow_2		文件	2009-04-09 15:54:14	2009-04-09 15:54:14.0		
	1	multiact:multiact	多个活动		文件	2009-04-09 15:54:13	2009-04-09 15:54:13.0	2008-11-01	2008-11-01

同样，在功能导航窗口中点击“流程实例管理”、“活动实例管理”等，可以看到相应的实例列表，在结果行单击右键可以对实例进行相应的管理功能。

关于 C/S 方式流程管理监控的详细介绍参见***《InforSuite Flow 流程仿真工具使用手册》***。

第6章 IS-Flow 流程数据分析

本章介绍了 IS-Flow 对流程数据的统计分析功能，主要体现在以下方面：

- 工作量负荷统计
- 单一任务超时统计
- 任务完成时间统计
- 流程执行工作量分析
- 流程实例状态统计

统计分析的结果均以报表的形式展现。

学习本章内容后，您可以：

- 了解 IS-Flow 如何通过流程数据统计分析出具有实际应用价值的结果。
- 结合*《InforSuite Flow 标准任务表处理器使用手册》*根据应用系统需求定制报表，并流程数据展示在报表中。

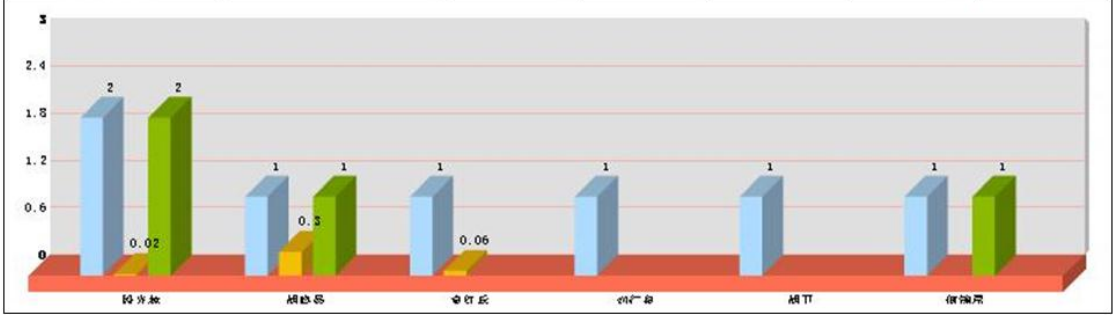
6.1. 工作量负荷统计

工作量负荷统计是指，统计每个人在某一时间段内处理完成的任务笔数、平均/最大/最小处理时间、当前待处理任务笔数。图表 6-1 工作量负荷统计展示了一个示例的统计结果：

工作量负荷统计

日期:2007-7-1～2008-8-8

人员姓名	处理总任务数	处理时间			任务数	
		平均值	最大值	最小值	在处理	待处理
段光林	2				2	0
胡晓曼	1				1	0
袁红兵	1	0.02	0.02	0.02	0	0
刘仁泉	1	0.3	0.3	0.3	0	0
胡卫	1	0.06	0.06	0.06	0	0
何锦星	1				1	0



图表 6-1 工作量负荷统计

6.2. 单一任务超时统计

单一任务超时统计是指，统计某段时间内某流程定义所产生的所有实例中每项任务的处理总笔数，其中超时处理的笔数，以及占任务总数的比例。图表 6-2 单一任务超时统计展示了一个示例的统计结果：

单一任务超时统计

年度	流程名称	任务名称	总任务数	超时任务数	超时百分比
	调增开票限额	结束	0	0	
		局长审批	5	0	0.00%
		开始	0	0	
		科长审批	5	0	0.00%
		税源管理科调查	5	0	0.00%
		税政审批	3	0	0.00%
		文书发放	5	0	0.00%
		文书受理	6	6	100.00%
		修改票种核定	5	0	0.00%
	非正常户认定流程	结束	0	0	
		开始	0	0	
		判断纳税人是否存在	2	0	0.00%

图表 6-2 单一任务超时统计

6.3. 任务完成时间统计

任务完成时间统计是指，按月度统计所定义的每个业务流程所产生的所有实例的总笔数，其中超时处理的笔数，以及占流程总数的比例。图表 6-3 任务完成时间统计展示了一个示例的统计结果：

业务流程(内资企业设立登记流程)任务完成时间统计

日期:2007-7-1～2008-08-08

单位：天

任务名称	平均完成时间	最大完成时间	最小完成时间
登记受理	0.03	0.05	0
辅查	0.01	0.03	0
核实调查	0.01	0.03	0
户管制定	0.005	0.02	0
数据维护	4.86	19.26	0.06
主查	0.025	0.04	0.01

共统计流程数(笔)： 3

统计时间:2007-8-24

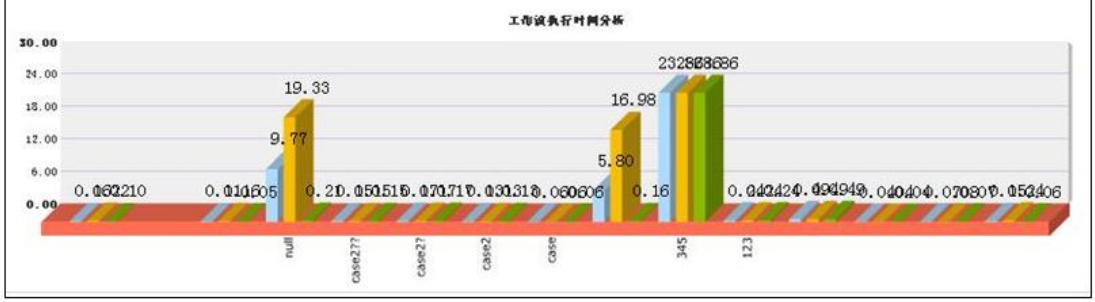
图表 6-3 任务完成时间统计

6.4. 流程执行工作量分析

流程执行工作量分析是指，统计某段时间内，某流程定义所产生的所有实例中每项任务所花费的工作量，平均/最大/最小工作量。图表 6-4 流程执行工作量分析展示了一个示例的统计分析结果：

工作流执行工作量分析

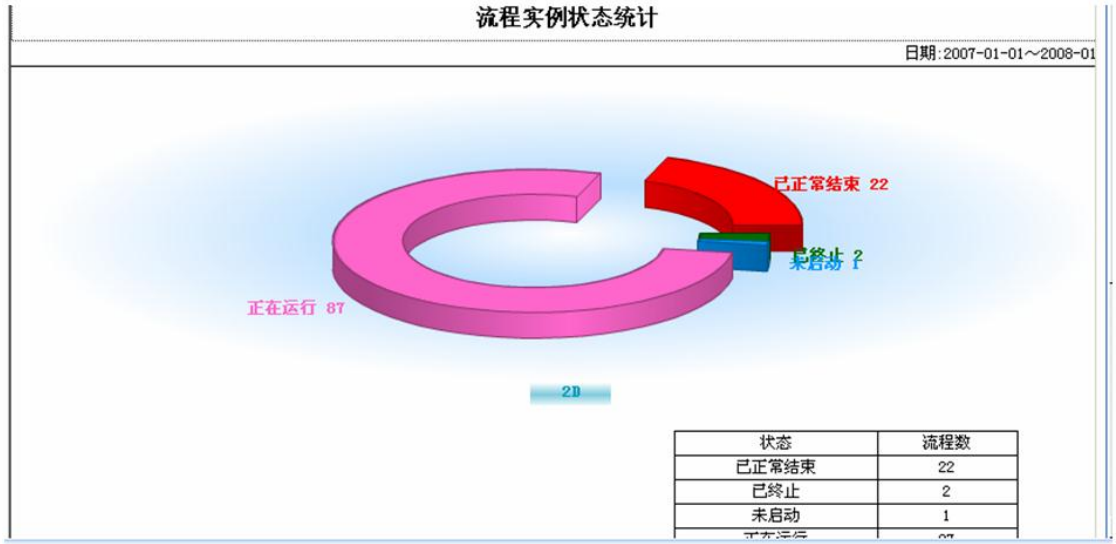
年度	流程名称	流程总数	执行时间分析（小时）			工作量分析（人时 ）		
			平均值	最大值	最小值	平均值	最大值	最小值
2007	延期缴纳税款申请流程	2	0.16	0.22	0.10	0.16	0.22	0.10
	非正常户认定流程	2	0.00	0.00	0.00	0.00	0.00	0.00
	内资企业设立登记流程	2	0.11	0.16	0.05	2.55	19.34	0.07
	null	2	9.77	19.33	0.21	2.55	19.34	0.07
	case2??	1	0.15	0.15	0.15	2.55	19.34	0.07
	case2?	1	0.17	0.17	0.17	2.55	19.34	0.07
	case2	1	0.13	0.13	0.13	2.55	19.34	0.07
	case	1	0.06	0.06	0.06	2.55	19.34	0.07
	调增开票限额	3	5.80	16.98	0.16	8.30	23.86	0.16
	345	1	23.86	23.86	23.86	8.30	23.86	0.16
	123	1	0.24	0.24	0.24	8.30	23.86	0.16
	设备申请流程	1	0.49	0.49	0.49	0.68	0.68	0.68
	设备采购流程	1	0.04	0.04	0.04	0.04	0.04	0.04
	证照领用流程	3	0.07	0.08	0.07	0.07	0.08	0.07
	因公出国（境）工作的审批流程	2	0.15	0.24	0.06	0.19	0.24	0.13



图表 6-4 流程执行工作量分析

6.5. 流程实例状态统计

流程实例状态统计是指，统计不同状态流程实例的数量。图表 6-5 流程实例状态统计展示了一个示例的统计结果：



图表 6-5 流程实例状态统计

以上是 IS-Flow 结合 InforSuite Report 提供的统计分析报表。如果这些统计分析报表不能满足应用系统的需求，用户可以参考***《InforSuite Flow 标准任务表处理器使用手册》***定制流程数据展示的统计分析报表。

关于 IS-Flow 流程数据分析的详细介绍参见***《InforSuite Flow 标准任务表处理器使用手册》***。

第7章 IS-Flow 用户编程

本章介绍 IS-Flow 用户编程，共分为三部分，包括：

- 简介。介绍 IS-Flow 提供的 API 接口。
- 初级编程。介绍基本的 API 接口调用，即如何通过 API 接口进行流程实例的创建、启动等操作。此部分用来教用户如何调用工作流引擎提供的 API 接口来实现相应的功能，不需要用户实现接口。
- 高级编程。介绍 IS-Flow 提供的扩展功能，如 IS-Flow 集成应用系统的用户数据、事件处理等。此部分需要用户实现 IS-Flow 提供的接口。

学习本章内容后，您可以：

- 通过调用 IS-Flow 提供的 API 接口实现流程相关的功能。
- 实现 IS-Flow 提供的扩展接口完成相应的扩展功能。

7.1. 简介

符合 WfMC 规范的工作流引擎针对功能不同的客户端应用系统，定义了五类接口，通过这五种接口的定义，详细的描述了工作流引擎对外发布的功能，这五类接口的功能分别为：

接口一定义了符合 WfMC 规范的工作流元模型结构以及用于描述一个业务流程的 XPDL 流程定义语言。

接口二定义了供应用程序调用的接口，完成流程控制、流程信息查询等功能。

接口三定义了工作流引擎调用应用的接口规范。

接口四定义了与其它工作流引擎之间进行交互的接口。

接口五定义了管理监控接口。

本章描述了 IS-Flow 工作流引擎所提供的流程控制、流程信息查询等功能，即描述了接口二所要求的功能。

应用调用工作流引擎的所有 API 接口均在 com.cvicse..workflow.api 包或子包下，见表格 7-1 工作流引擎 API 接口列表：

表格 7-1 工作流引擎 API 接口列表

包名	包含的 API
com.cvicse.workflow.api	包括行为接口 WfClient 以及各种定义和运行信息的实体接口
com.cvicse.workflow.api.event	允许用户在引擎运行时插入的各种事件插件
com.cvicse.workflow.api.exception	工作流抛出的异常，其根异常为 RuntimeException
com.cvicse.workflow.api.query	为构造综合查询条件所用
com.cvicse.workflow.api.resource	资源接口，包括应用系统必须实现的 AbstractResourceProvider 以及资源接口。

7.2. 搭建编程环境

进行用户编程，首先要搭建编程环境。IS-Flow 用户编程需要搭建服务器端和客户端。服务器端负责发布工作流引擎的服务，如 local 方式或 rmi 方式或 webservice 方式的服务。客户端通过配置访问工作流引擎服务来完成与工作流引擎的交互。

7.2.1. 服务器端

工作流引擎可以发布三种方式的服务，如果工作流引擎服务以嵌入运行模式集成到应用系统中，那么工作流引擎只能发布 local 方式的服务。如果工作流引擎服务以独立运行模式集成到应用系统中，那么工作流引擎可以发布 rmi 和 WebService 方式的服务，下面具体讲述工作流引擎服务在不同的运行模式下如何发布服务。

1. 嵌入运行模式下发布 local 服务

如果工作流引擎服务以嵌入运行模式集成到应用系统中，应用系统需要将 IS-Flo

w 核心 jar 包（inforflow.jar）及其依赖的 jar 包（参见 7.2.3. 节 inforflow.jar 依赖包说明）和配置文件放在相应的目录下，具体信息参见 IS-Flow 产品提供的嵌入式标准任务表处理器应用 tasklist.war，此应用位于 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\tasklist\inset\下。

2. 独立运行模式下发布 RMI 服务

如果工作流引擎服务以独立运行模式集成到应用系统中，应用系统需要工作流引擎发布 RMI 服务，那么只需在 IS-Flow 管理平台中配置相应的工作流服务实例，启动工作流服务实例即可发布 RMI 服务。

3. 独立运行模式下发布 WebService 服务

如果工作流引擎服务以独立运行模式集成到应用系统中，应用系统需要工作流引擎发布 WebService 服务，那么只需要将产品安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\下的 flowservice_axis1.4.war 或 flowservice_cxf..war 部署到任何 Web 服务器上即可。

7.2.2. 客户端

客户端负责连接工作流引擎发布的服务。连接工作流引擎服务需要配置 inforflow-client.xml 以及将 IS-Flow 提供的 API 接口（inforflow_api.jar）放置早应用系统依赖的 lib 目录下，inforflow_api.jar 可以在 IS-Flow 产品提供的独立式标准任务表处理器应用中获取，独立式的应用 tasklist.war 位于 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\tasklist\independence\下。

客户端根据应用系统连接的服务类型配置 inforflow-client.xml， 具体设置参见 inforflow-client.xml。

进行客户端编程时，应用系统需要依赖的 jar 包以及配置文件如下：

- inforflow_api.jar，放在应用依赖的 lib 下。
- inforflow-client.xml，放在应用的 src 根目录下。
- inforflow-image.xml，放在应用的 src 根目录下。

◆ inforflow-cache-client.xml，放在应用的 src 根目录下。

关于配置文件 inforflow-client.xml 、 inforflow-image-client.xml 和 inforflow-cache-client.xml 的配置参见 IS-Flow 配置文件。它们可以在 IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\tasklist\independence\下的 tasklist.war 中获取到。

7.2.3. inforflow.jar 依赖包

工作流引擎核心 jar 包为 inforflow.jar，它可以在 flowservice_axis1.4.war 或 flowservice_cxf..war 中获取。它的依赖包整理如下：

必须的		
包名	**版本**	**说明**
antlr-2.7.5H3.jar	2.7.7	Hibernate3 必须，解析 HSQL，此包会与 weblogic 相冲突；修改 weblogic 启动脚本，优先加载此包。
asm-attrs.jar		必须，辅助 CGLIB 生成字节码
asm-util.jar		必须，辅助 CGLIB 生成字节码
asm.jar	2.2.3	必须，辅助 CGLIB 生成字节码
cglib-2.1.jar	2.1.3	Hibernate3 使用 CGLIB 动态生成 PO 码
commons-beanutils.jar	1.7.0	Hibernate3 必须
commons-collections-3.1.jar	3.1	Hibernate3 必须，Qaurtz1.6.0 必须
commons-dbcp-1.2.1.jar	1.2.1	InforFlow 内部数据库连接池
commons-lang-1.0.1.jar	2.4	Hibernate3 必须
commons-logging-1.0.4.jar	1.1.1	Hibernate3,Spring 必须
commons-pool-1.2.jar	1.2	InforFlow 内部数据库连接池需要
concurrent-1.3.2.jar	1.3.4	Hibernate3 若采用 TreeCache 时必须
connector.jar		JCA 接口。可选。
dom4j-1.4.jar	1.6.1	Hibernate 解析*.hbm.xml 必须

ehcache-1.2.4.jar	1.2.4	InforFlow 内存缓存必须
jaxen-1.1.jar	1.1	Dom4j-1.6.1 需要
hibernate3.jar	3.2.6	
jgroups-2.2.9.jar		Hibernate3 若采用 replicated caches 时必须
log4j-1.2.9.jar	1.2.14	InforFlow 必须
spring.jar	2.0.2	必须
verifycode.jar		InorFlow 验证版本控制
cviccpr.jar		
xalan-2.4.0.jar	2.4.0	Xpdl 层引用到 （这 3 个 jar 暂时删除，待处理）
xercesImpl.jar		
xml-apis.jar		
servlet-api.jar		InforFlow 编译时需要
ehcache-1.1.jar	1.2.4	Hibernate 推荐的缓存包
backport-util-concurrent-2.1.jar		InforSuite V6 新增
commons-codec-1.3.jar		
inforsuite-kernel-2.0.1.jar		
inforsuite-system-2.0.1.jar		
jdom-1.0.jar		

可选的		
包名	**版本**	**说明**
classes12.jar		Oracle 数据库驱动
db2fs.jar db2java.jar db2jcc.jar		DB2 数据库驱动
derbyclient.jar		derby 数据库驱动
httpunit.jar		综合测试必须

Js.jar		综合测试_测试自动化时用来处理 javascript
nekohtml.jar		综合测试必须
xmlParserAPIs.jar		综合测试必须
junit-3.8.1.jar	3.8.1	单元测试必须
mm.mysql-2.0.4-bin.jar		MYSQL 数据库驱动
	3.1.7	mysql-connector-java-3.1.7-bin.jar
mail.jar	J2EE 提供	邮件提醒必须
msbase.jar		MS SQLServer 数据库驱动
mssqlserver.jar		
msutil.jar		
jconn2.jar		Sybase 数据库驱动
jtds-1.2.jar		第三方驱动包（the open source JDBC driver for Microsoft SQL Server and Sybase）
jai_codec.jar	1_1_2_01	SUN 提供，流程监控图导出功能必须
jai_core.jar	1_1_2_01	SUN 提供，流程监控图导出功能必须
quartz-1.6.0.jar	1.5.1	定时服务包,若使用引擎的定时启动流程，任务催办等功能所必须.
comm.jar		SUN 提供的串口通讯包，手机短消息提醒必须。
RXTXcomm.jar		手机短消息提醒必须。
jSMSEngine.jar	1.2.8	开源手机短消息工具包，屏蔽了短信发送设备与串口通讯的细节。手机短消息提醒必须。
smslib.jar		手机短消息提醒必须。
		JMS 1.1 的开源实现，JMS 消息提醒必须。

inforbroker-core.jar		替换 activemq-3.2.1.jar
inforbroker-optional.jar		
inforbroker-ra.jar		
geronimo-spec-j2ee-management-1.0-rc4.jar		Geronimo 版本的 jmx 驱动包，JMS 消息提醒必须
	1.1	替换为 geronimo-j2ee-management_1.1_spec-1.0.jar
geronimo-spec-jms-1.1-rc4.jar		Geronimo 版本的 jms 驱动包, JMS 消息提醒必须
geronimo-spec-jta-1.0.1B-rc4.jar		Geronimo 版本的 jta 驱动包(去掉原来的 jta.jar), JMS 消息提醒必须
wsdl4j-1.6.1.jar	1.6.1	Web Service 之 Cxf 和 Axis 共用
axis.jar axis-ant.jar（可选） jaxrpc.jar activation.jar commons-discovery-1.0.jar saaj.jar	1.2 1.0	Web Service 之 Axis 专用
retroguard.jar		类库混淆所需包
ldap.jar		Ldap 支持
utilities.jar		
resolver.jar		
antlr-runtime.jar		Drools 依赖包；xercesImpl.jar 和 xml-apis.jar 引擎 xpdl 层已经依赖
core-3.2.3.v_686_R32x.jar		
drools-compiler.jar		
drools-core.jar		
drools-decisiontables.jar		

drools-jsr94.jar		
janino-2.5.10.jar		
jsr94.jar		
jxl.jar		
mvel14.jar		
xpp3.jar		
xpp3_min.jar		
xstream.jar		
xercesImpl.jar		
xml-apis.jar		

备注 1.

增加 cxf 功能 jar 包整理
包名
14.新增 cxf 依赖包列表如下： abdera-core-0.4.0-incubating.jar abdera-extensions-html-0.4.0-incubating.jar abdera-extensions-json-0.4.0-incubating.jar abdera-extensions-main-0.4.0-incubating.jar abdera-i18n-0.4.0-incubating.jar abdera-parser-0.4.0-incubating.jar abdera-server-0.4.0-incubating.jar aopalliance-1.0.jar axiom-api-1.2.7.jar axiom-impl-1.2.7.jar commons-httpclient-3.1.jar cxf-2.1.1.jar

cxf-manifest.jar
FastInfoset-1.2.2.jar
geronimo-activation_1.1_spec-1.0.2.jar
geronimo-annotation_1.0_spec-1.1.1.jar
geronimo-javamail_1.4_spec-1.3.jar
geronimo-jaxws_2.1_spec-1.0.jar
geronimo-servlet_2.5_spec-1.2.jar
geronimo-stax-api_1.0_spec-1.0.1.jar
geronimo-ws-metadata_2.0_spec-1.1.2.jar
htmlparser-1.0.5.jar
jaxb-api-2.1.jar
jaxb-impl-2.1.6.jar
jaxb-xjc-2.1.6.jar
jettison-1.0.jar
jetty-6.1.9.jar
jetty-util-6.1.9.jar
jra-1.0-alpha-4.jar
js-1.6R7.jar
jsr311-api-0.6.jar
neethi-2.0.4.jar
opensaml-1.1.jar
saaj-api-1.3.jar
saaj-impl-1.3.jar
slf4j-api-1.3.1.jar
slf4j-jdk14-1.3.1.jar
stax-utils-20060502.jar
velocity-1.4.jar
velocity-dep-1.4.jar

wss4j-1.5.4.jar
wstx-asl-3.2.4.jar
xmlbeans-2.3.0.jar
xml-resolver-1.2.jar
XmlSchema-1.4.2.jar
xmlsec-1.4.0.jar

7.3. 初级编程

7.3.1. IS-Flow 工作流引擎初始化

初始化工作流引擎目的是为了创建工作流的运行环境，初始化模式根据连接方式的不同可分为：本地服务(local)、RMI 远程服务、WebService 远程服务三种模式。其中，本地服务又分 web 应用模式下的本地服务和类调用模式下的本地服务两种。

初始化工作流引擎涉及到的配置文件有如下几个：

- 引擎服务启动时的配置文件 inforflow.xml： web 应用模式启动本地化引擎服务时需要将该文件放在应用的\WEB-INF\目录下。
- web 应用模式下的配置文件 web.xml：启动 web 服务前需要将该文件放在应用的\WEB-INF\目录下。

上述配置文件也可以参照 IS-Flow 提供标准任务表处理器(tasklist)应用。

7.3.1.1. 本地服务

本地服务，即由本地启动工作流执行服务，又分 web 应用模式下的本地服务和类调用模式下的本地服务两种。都需要配置工作流引擎的初始化环境 inforflow.xml。

web 应用模式通过本地连接的 InitialServlet 执行工作流引擎的初始化工作，并需要通过配置应用下的\WEB-INF\web.xml 来指定初始化类路径和初始化环境读取的配置文件。

web.xml 的配置如下：

Web 应用模式下初始化工作流引擎参考代码：

```
//--------------------------------------初始化工作流引擎 web.xml 配置片段-开始
    <servlet>
     <servlet-name>InitServlet</servlet-name>

<servlet-class>com.cvicse.workflow.web.InitialServlet</servlet-class>
     <init-param>
       <param-name>SystemConfigFilePath</param-name>
       <param-value>/WEB-INF/inforflow.xml</param-value>
       <description>inforflow config file path and name</description>
     </init-param>
     <load-on-startup>1</load-on-startup>
     </servlet>
//--------------------------------------初始化工作流引擎 web.xml 配置片段-结束
```

- 选择初始化类 com.cvicse.workflow.web.InitialServlet。
- 配置初始化时参数：IS-Flow 初始化参数的系统变量为 SystemConfigFilePath，指定其值/WEB-INF/inforflow.xml。

类调用模式需要在类的入口处首先调用引擎的 WorkflowConfig 执行工作流引擎的初始化，程序调用示例如下：

类初始化工作流引擎参考代码：

```
//----------------------------初始化工作流引擎 inforflow.xml 配置片段-开始
     /* 指定初始化文件 */
     File file = new File(
InforFlowTestCase.class.getResource("inforflow.xml").getFile());
    /* 初始化配置文件，解析初始化参数 */
     WorkflowConfig.initBeanFactory(file);
//----------------------------初始化工作流引擎 inforflow.xml 配置片段-结束
```

当然，本地服务都需要配置类路径下的 inforflow-client.xml 配置文件指定使用的是 localClient。

7.3.1.2. RMI 远程服务

启动 InforSuite 应用服务器后，默认提供一个 RMI 远程服务，用户可以根据应用系统的需要配置自己的工作流服务实例，启动后即可发布 RMI 远程服务。

7.3.1.3. WebService 远程服务

WebService 远程服务，需要配置工作流引擎的初始化环境：将 IS-Flow 安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow 下的 flowservice 应用部署到 InforSuite 应用服务器上，即可提供 IS-Flow 的 WebService 远程服务；修改 flowservice 应用下的 inforflow.xml 配置文件，来配置工作流引擎的初始化环境。

7.3.2. 应用系统与工作流引擎服务的连接方式

应用连接工作流引擎服务有三种方式：本地连接、RMI 远程连接、WebService 服务远程连接。

本地连接情况下，由本地启动服务器；首先需要配置工作流引擎的初始化环境，通过配置 web.xml 来指定初始化类路径和初始化环境读取的配置文件；通过配置 inforflow-client.xml 来指定服务引擎。

RMI 远程连接情况下，由远程启动服务器；客户端只需配置 inforflow-client.xml 来指定服务引擎。

WebService 服务远程连接情况下，由远程启动 WEB 服务服务器；客户端只需配置 inforflow-client.xml 来指定 WEB 服务地址信息。根据客户端编码语言的不同，又可以分为 Java 的 WebService 服务远程连接和.Net 的 WebService 服务远程连接两种，后者具体可参见 ***《WebService 配置及.NET 平台下 InforSuite Flow 的使用手册》***，在此不做介绍。

连接工作流引擎服务涉及的配置文件如下：

- inforflow-client.xml，用于配置客户端应用于引擎服务的连接方式，该文件需放在应用的\WEB-INF\classes 目录下。另外，inforflow-image-client.xml 用于应用程序访问工作流引擎的导出图片功能，其存放位置与 inforflow-client.xml 相同。

7.3.2.1. 本地连接

本地连接需要首先按照 7.1.1.2 节中的方式配置本地服务的配置文件，然后配置类路径下的 inforflow-client.xml 配置文件指定使用的是 localClient。

inforflow-client.xml 的配置如下：

本地连接工作流引擎的配置文件参考代码：

```
//--------------------应用连接工作流引擎的配置文件inforflow-client.xml配置片段-开始 <beans>
    <bean id="localClient"
       class="com.cvicse.workflow.api.client.local.LocalClient"
 singleton="false"/>
    <bean id="InforFlowClient"
          class="com.cvicse.workflow.api.client.WfClientImpl"
singleton="false">
        <constructor-arg><ref bean="localClient"/></constructor-arg>
    </bean>
      </beans>
//------------------应用连接工作流引擎的配置文件 inforflow-client.xml 配置片段-结束
```

- 设置 localClient 类路径 com.cvicse.workflow.api.client.local.LocalClient；
- 设置 InforFlowClient 类路径 com.cvicse.workflow.api.client.WfClientImpl；
- 选择 localClient 接口为服务接口 <ref bean="localClient"/>。

7.3.2.2. RMI 远程连接

首先需要启动工作流引擎的 RMI 远程服务，然后配置类路径下的 inforflow-client.xml 配置文件指定使用的是 rmiClient。

inforflow-client.xml 的配置如下：

RMI 方式连接工作流引擎的配置文件参考代码：

```
//--------------------应用连接工作流引擎的配置文件inforflow-client.xml配置片段-开始<beans>
    <bean id="rmiClient"
          class="com.cvicse.workflow.api.client.rmi.WfRMIClient"
destroy-method="close" init-method="init" lazy-init="true">
        <property name="hostUrl">
            <value>192.168.51.53</value>
        </property>
        <property name="hostPort">
            <value>5501</value>
        </property>
        <property name="serviceName">
            <value>InforFlowService</value>
        </property>
    </bean>

    <bean id="InforFlowClient"
          class="com.cvicse.workflow.api.client.WfClientImpl"
singleton="false">
        <constructor-arg><ref bean="rmiClient"/></constructor-arg>
    </bean>
      </beans>
//------------------应用连接工作流引擎的配置文件 inforflow-client.xml 配置片段-结束
```

- 设置 rmiClient 类路径 com.cvicse.workflow.api.client.rmi.WfRMIClient；
- 指定连接远程服务器的各属性值；
- 设置 InforFlowClient 类路径 com.cvicse.workflow.api.client.WfClientImpl；

◆ 选择 rmiClient 接口为服务接口 <ref bean="rmiClient"/>。

7.3.2.3. WebService 服务远程连接

首先需要启动工作流引擎的 WebService 远程服务，然后配置类路径下的 inforflow-client.xml 配置文件指定使用的是 wsClient。

目前工作流引擎提供两种 WebService 容器（CXF 和 AXIS1.4）的 WebService 服务。如果应用系统使用 JDK1.4，则可以使用 AXIS1.4 容器的 WebService 服务。如果应用系统使用 JDK1.5，则可以使用 CXF 容器和 AXIS1.4 容器的 WebService 服务。下面分别介绍一下 CXF 容器和 AXIS1.4 容器下工作流引擎相应的配置。

IS-Flow 产品安装目录%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\下有两个 war 包，即 flowservice_axis1.4.war 和 flowservice_cxf.war。其中 flowservice_axis1.4.war 是基于 AXIS1.4 容器的 web 应用，在 web 服务器上部署这个 war 就可以启动工作流引擎的 WebService 服务。flowservice_cxf.war 是基于 CXF 容器的 web 应用，在 web 服务器上部署此应用即可启动工作流引擎的 WebService 服务。

客户端连接工作流引擎的 WebService 服务需要配置 inforflow-client.xml，此文件位于应用的 classes 根目录下，关于 WebService 客户端的配置如下：

inforflow-client.xml 的配置如下：

Web 应用模式下初始化工作流引擎参考代码：

```
//--------------应用连接工作流引擎的配置文件inforflow-client.xml配置片段-开始
     <beans>
    <bean id="wsClient"
          class="com.cvicse.workflow.api.client.webservice.WebServiceClient"
destroy-method="close" init-method="init" lazy-init="true">
        <property name="serviceAddress">
    <value>http://localhost:6522/flowservice/services/WsClient</value>
        </property>
    </bean>
    <bean id="InforFlowClient"
          class="com.cvicse.workflow.api.client.WfClientImpl"
```

```
singleton="false">
        <constructor-arg><ref bean="wsClient"/></constructor-arg>
    </bean>
      </beans>
//---------------应用连接工作流引擎的配置文件 inforflow-client.xml 配置片段-结束
```

- 设置 wsClient 类路径。如果连接 AXIS 容器的 Webservice 服务，设置 id 为 wsClient 的类为 com.cvicse.workflow.api.client.webservice.WebServiceClient；如果连接 CXF 的 WebService 服务，设置 id 为 wsClient 的类为 com.cvicse.workflow.api.client.webservice.cxf.WebServiceClient；
- 指定连接远程服务器的 serviceAddress 属性值；
- 设置 InforFlowClient 类路径 com.cvicse.workflow.api.client.WfClientImpl；
- 选择 wsClient 接口为服务接口 <ref bean=" wsClient"/>。

7.3.3. 连接工作流引擎

包 com.cvicse.workflow.api 下的 WfClient 接口包含工作流引擎的所有行为接口，包括获取定义或实例对象，控制实例对象等。调用这些接口前需要调用 WfClient 接口的 connect()方法，在调用工作流接口后使用 disconnect()方法断开与工作流引擎的连接。

若应用使用 connect()方法与工作流引擎建立一个连接，工作流引擎在内存中将记录当前线程连接的执行人信息，并产生连接到工作流引擎的唯一会话 ID。在连接后调用引擎 API 时，工作流引擎将校验应用是否连接工作流引擎；在执行接受工作项，完成工作项等 API 时，引擎还将引用执行人 ID 信息，并进行校验。

7.3.3.1. 获取 WfClient

通过 WfClientManager.getInstance().getWfClient()方法可以获取 WfClient 接口；

WfClient 有三种实现方式，即 Local 客户端、RMI 客户端和 WebService 客户端，由 inforflow-client.xml 配置实现。具体配置 inforflow-client.xml 参照第 7.1.2 节。

7.3.3.2. 建立连接

调用工作流引擎的方法，需首先建立与工作流引擎的连接。建立连接时需要传递 WfConnectInfo 参数，构造 WfConnectInfo 实例可使用以下构造方法：

构造 WfConnectInfo 实例的方法：
WfConnectInfo(java.lang.String userId) 构造简单的连接信息对象
WfConnectInfo(java.lang.String userId，java.sql.Connection connection) 构造连接信息对象
WfConnectInfo(java.lang.String userId，java.lang.String password) 构造简单的连接信息对象
WfConnectInfo(java.lang.String userId，java.lang.String password，java.sql.Connection connection) 构造连接信息对象
WfConnectInfo(java.lang.String userId，java.lang.String password，java.sql.Connection connection，char txType) 构造连接信息对象

其中比较复杂的连接对象：

带有数据库连接对象的构造参数表示由应用将数据库连接传递给工作流引擎，并由应用统一控制数据库事务，参照第 2.3 节 IS-Flow 的事务处理方式；

带有用户名密码的构造参数表示需要 IS-Flow 进行授权鉴证服务；

带有 txType 事务类型的构造参数表示此次连接工作流引擎需要该指定类型的事务，具体的 txType 参照附录 IS-Flow 常量表

连接工作流的参考代码：

```
//----------------------------------------建立与工作流引擎连接的代码片段-开始
      /* 构造请求服务相关的鉴别信息 */
     WfConnectInfo conInfo = new WfConnectInfo("userId");
     WfClientManager cm = WfClientManager.getInstance();
```

```
      /* 首先获取客户端接口对象 */
      WfClient client = cm.getWfClient();
      //--------
      //使用 client 的行为接口
      //--------
     /* 建立与工作流引擎连接 */
      client.connect(conInfo);
//----------------------------------------建立与工作流引擎连接的代码片段-结束
```

7.3.3.3. 断开连接

断开与工作流引擎的连接。

断开与工作流连接的参考代码：

```
//----------------------------------------断开与工作流引擎连接的代码片段-开始
   /* 断开与工作流引擎连接*/
   client.disconnect();
//----------------------------------------断开与工作流引擎连接的代码片段-结束
```

7.3.4. 流程控制

流程控制函数用于改变一个或多个流程实例的状态。工作流产品的终端用户应用程序会调用这些 API。但是，某些 API 调用或者某些 API 调用的参数，可能会影响多个用户，因此在正常情况下被限制为只能被流程管理员所使用。

7.3.4.1. 获取流程定义列表

查询并列出满足过滤器选择标准的所有流程定义。

获取流程定义列表的参考代码：

```
//-----------------------------------------获取流程定义列表代码片段-开始
        /* 构造请求服务相关的鉴别信息 */
        WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象并连接 */
         WfClient client = cm.getWfClient();
         client.connect(conInfo);
         WfFilter filter = null;
         /* 列出满足过滤器选择标准的所有流程定义，
         如果 filer 为空则列出所有的流程定义 */
         List procDefs = client.listProcessDefinitions(filter);
        /* 断开与工作流引擎连接*/
         client.disconnect();
//-----------------------------------------获取流程定义列表代码片段-结束
```

这个命令可以用于管理员或流程管理者得到一个流程定义列表，以查看哪些流程对哪些特定的人员是可启动的。通过设置 `WfFilter` 指导工作流引擎打开查询，提供一个对某个特定的工作流参与者可用的流程定义列表，有一些可能是这个参与者可启动的。组织内的所有流程并不是都能被所有的参与者启动的。该功能的用途之一是让一个工作流参与者查看能启动哪些流程，这个参与者下一步很有可能将会选择一个并启动。

工作流引擎同时还提供了一个便捷的流程定义查询接口：“获取流程定义第一个节点包括指定执行人的所有流程定义”。

获取流程定义列表的参考代码：

```
//--------------------获取流程定义列表的代码片段-开始--------------------
       /* 构造请求服务相关的鉴别信息 */
             WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 获取流程定义第一个节点包括指定执行人的所有流程定义*/
    List procDefs = client.listBeginerProcessDefinitions("userId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//--------------------获取流程定义列表的代码片段-结束--------------------
```

7.3.4.2. 获取流程定义

根据“获取流程定义列表”返回的 procDefs 获取流程定义：

获取流程定义的参考代码：

```
//-----------------------------------------获取流程定义代码片段-开始
      WfProcessDefinition procDef =
(WfProcessDefinition)procDefs.get(i);
//-----------------------------------------获取流程定义代码片段-结束
```

该功能的作用是从“获取流程定义列表”中返回的 List 对象中获取一个流程定义。

7.3.4.3. 创建流程实例

创建一个流程定义的实例。此命令的结果是工作流引擎创建一个指定流程定义的流程实例。

创建流程实例的参考代码：

```
//-------------------------------------------创建流程实例的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
        WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        String procDefId = "test";
        /*创建流程实例 */
        String procInstId = client.createProcessInstance(
  procDefId,"流程实例 1");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-------------------------------------------创建流程实例的代码片段-结束
```

该功能返回一个用于启动该流程实例的临时的流程实例标识号。

7.3.4.4. 设置流程实例机构代码

流程实例创建后，可以指定当前流程实例的机构代码，用以限制整个流程实例在运转过程中的任务分配；任务分配只有在抢任务的情况下受流程实例的机构代码所限制，执行人的所属机构满足流程实例机构才能接受任务，会签点和单一执行人不受流程实例的机构代码限制；流程实例在不设置机构代码的情况下，当前流程实例的机构代码默认为空，流程实例在运行阶段不受机构代码的限制。

设置指定流程实例的机构代码参考代码：

```
//---------------------------------------设置指定流程实例的机构代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
        WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*设置指定流程实例机构代码 */
        String finalProcInstId = client. resetProcessInstCode (
  "procInstId ", "instcode");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//---------------------------------------设置指定流程实例的机构代码片段-结束
```

在流程运行阶段，通过此方法可以随时修改流程实例的机构代码，这样同一流程实例内即能够实现上报到上级机构的功能，又能够实现指示下级机构工作的功能，最终实现一个流程可以在不同机构内的流转。

7.3.4.5. 启动指定流程实例

流程实例的启动可以分为应用直接启动指定流程实例和引擎动态启动指定流程实例两种方式。应用直接启动指的是由应用直接调用引擎提供的启动流程实例接口，引擎即时启动指定的流程实例；引擎动态启动指的是应用调用引擎提供的动态定时启动接口，设置流程启动的时间（绝对时间格式为“yyyy-MM-dd HH:mm”，相对时间格式为“d.h.m.s”，相对点是引擎服务器当前的时间）及其它信息，引擎将在指定时间自动启动该流程实例。

启动指定流程实例的参考代码(注释部分为引擎动态启动指定流程实例)：

```
//----------------------------------------启动指定流程实例的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*应用直接启动指定流程实例 */
        String finalProcInstId =
  client.startProcess("userId", "procInstId");
        /**
        /*应用设置引擎动态启动指定流程实例 */
        client.setProcessStartTime(procInstId,timeString,
  isAbsoluteTime,initContxtClassName);
        */
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------启动指定流程实例的代码片段-结束
```

启动指定的流程实例。此命令(startProcess)指示工作流引擎开始执行一个流程，这个流程已经创建了一个实例。当一个流程通过此命令被启动时，流程的第一个活动将被启动。返回的流程实例 ID 在此流程实例的生命周期内有效。

7.3.4.6. 终止流程实例

终止指定流程实例的参考代码：

```
//----------------------------------------终止指定流程实例的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
```

```
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*终止指定流程实例 */
        client.terminateProcessInstance("procInstId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------终止指定流程实例的代码片段-结束
```

结束一个流程实例。此命令提供了一种结束流程而不必取消流程实例的能力，将未结束的活动终止掉，各活动的相关数据值被拷贝到流程实例对象中。

7.3.4.7. 取消流程实例

取消指定流程实例的参考代码：

```
//----------------------------------------取消指定流程实例的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*取消指定流程实例 */
        client.abortProcessInstance("procInstId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
```

```
//---------------------------------------取消指定流程实例的代码片段-结束
```

取消一个流程实例。此命令提供取消流程实例的能力，将所有未结束的活动取消掉，各活动的相关数据值将不被拷贝到流程实例对象中。

7.3.4.8. 获取指定流程实例状态

获取指定流程实例状态的参考代码：

```
//---------------------------------------获取指定流程实例状态的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*获取一个指定的流程实例*/
        WfProcessInstance procInst =
        client.getProcessInstance ("procInstId");
        /*获取流程实例状态 */
        String state = procInst.getState();
        /* 断开与工作流引擎连接 */
        client.disconnect();
//---------------------------------------获取指定流程实例状态的代码片段-结束
```

该功能可以获取到指定流程实例的状态字符串。

7.3.4.9. 获取流程实例的属性列表

获取流程实例属性列表的参考代码：

```
//--------------------------------------获取流程实例属性列表的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        WfFilter filter = null;
        /*列出满足过滤器选择标准的所有流程实例的属性 */
        Map attrs = client.listProcessInstanceAttributes(
  "procInstId", filter);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//--------------------------------------获取流程实例属性列表的代码片段-结束
```

返回一个满足过滤条件的指定流程实例的属性列表。该功能的用途之一是，让应用向工作流引擎询问已分配给流程实例的可用属性，而后将此列表提供给应用用户。

返回流程相关数据标识-值。其中 key 为在 xpdl 中所定义的流程相关数据的标识号，value 为此相关数据在运行过程中的值，其类型为定义中所指定的相关流程的类型。

7.3.4.10. 获取流程实例属性列表中指定属性的值

返回指定属性的值。

获取流程实例属性列表中指定属性的值参考代码：

```
//-------------------------获取流程实例属性列表中指定属性的值的代码片段-开始
        Object attrValue = attrs.get(attrName);
//-------------------------获取流程实例属性列表中指定属性的值的代码片段-结束
```

7.3.4.11. 获取指定流程实例指定属性的值

根据输入的流程实例 ID 和流程属性 ID 返回流程属性值。

获取指定流程实例指定属性的值的参考代码：

```
//--------------------------------获取指定流程实例指定属性的值的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
         WfClientManager cm = WfClientManager.getInstance();
         /* 首先获取客户端接口对象 */
         WfClient client = cm.getWfClient();
         /* 建立与工作流引擎连接 */
         client.connect(conInfo);
         /*获取流程实例指定属性的值  */
         Object attrValue =
             client.getProcessInstanceAttributeValue("procInstId"      ,
"attrID");
         /* 断开与工作流引擎连接 */
         client.disconnect();
//--------------------------------获取指定流程实例指定属性的值的代码片段-结束
```

7.3.4.12. 分配一个属性值给指定流程实例属性

分配一个属性给流程实例。该功能通知工作流引擎为一个流程实例的属性赋值或改变流程实例的一个属性的值。流程实例的属性是流程控制和流程相关数据的一类。

分配一个属性值给指定流程实例属性参考代码：

```
//-----------------------------------------分配流程实例属性的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        String attrValue = "example";
        /*分配一个属性值给指定流程实例属性 */
        client.assignProcessInstanceAttribute("procInstId", "attrID",
                                              attrValue);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------分配流程实例属性的代码片段-结束
```

注意：流程属性值的长度默认为 100 个字节，如果希望修改流程属性的长度可以通过如下操作完成：将 inforflow.jar 中的 com 目录解压，修改 com\cvicse\workflow\datastore\db\entity 目录下的 ProcessRelevantDataEntity.xml 文件和 WHProcessRelevantDataEntity.xml 文件，将两个文件中的<property name="value" type="java.lang.String" length="100" access = "field"/>中的 length 改成期望的值。修改完成以后需要重新生成数据库。

下面还需要重点介绍下对象类型流程属性值，其赋值方法同基本类型流程属性值的赋值方法，即将代码片断中的 `attrValue` 声明为相应的对象类型；比基本类型流程属性赋值麻烦的是：

工作流引擎的 web 应用模式为嵌入模式（本地连接）的情况下，需将 com.cvicse.workflow.api.DataFieldFactory 的实现类及需要赋值的对象类放到 web 应用的类路径

下；具体实现可参考引擎内部的例子，com.cvicse.workflow.examples.datafieldfactory.CreditDataFieldFactory 和 com.cvicse.workflow.examples.custombean.CreditBean；

工作流引擎的 web 应用模式为独立模式（远程连接）的情况下，既需将 com.cvicse.workflow.api.DataFieldFactory 的实现类及需要赋值的对象类放到 web 应用的类路径下，又需将其放到引擎服务端的类路径下。

对象类型的属性值，记录在工作流数据库中的值为当前对象标志属性的值；这样在通过引擎接口获取对象类型属性值的时候，是通过获取出来工作流数据库中的对象标志属性的值，然后通过 DataFieldFactory 的实现类根据标志属性的值获取出对象类型属性值；这样需要用户编程的 DataFieldFactory 的实现类以及定义过程中的对象标志属性，需要用户详细设计，以达到预期效果。

7.3.4.13. 复活流程实例

通过执行此命令，可以将已结束的流程实例（既正常结束、终止和取消的流程实例）恢复为流程结束前一样的状态，当前流程实例的活动和工作项等信息同步恢复到结束前的状态。

复活流程实例参考代码：

```
//-----------------------------------------------复活流程实例的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
        WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 复活指定的流程实例 */
        client.reviveProcess("procInstId");
```

```
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-------------------------------------------复活流程实例的代码片段-结束
```

注意：由于各数据库对工作流表主键生成机制支持的不同，需要作相应配置才能支持 Sybase 和 SQLServer 这两个数据库的流程实例复活。配置方法详见“技术问答”第 4 条。

不支持未启动流程实例复活。对流程结束前流程实例状态为挂起状态的流程实例进行复活，复活后的流程实例变为运行时状态，相当于同时执行了恢复操作。

7.3.4.14. 删除流程实例

删除流程实例。该功能通知工作流引擎删除一个运行时的或历史的流程实例。该操作在以后将提供对恢复删除的支持。

删除流程实例参考代码：

```
//-------------------------------------------删除流程实例的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        String attrValue = "example";
        /*删除一个运行时的流程实例*/
        client. removeProcessInstance("procInstId",false,false);
        /* 断开与工作流引擎连接 */
```

```
        client.disconnect();
//-------------------------------------------删除流程实例的代码片段-结束
```

7.3.5. 活动控制

活动控制函数用于改变一个或多个活动实例可操作的状态。工作流产品的终端用户应用程序会调用这些 API。但是，某些 API 调用，或者某些调用的参数，可能会影响多个用户，因此正常情况下被限制为只能被流程管理员所使用。

7.3.5.1. 获取活动实例的状态

获取活动实例状态的参考代码：

```
//-------------------------------------------获取活动实例状态的代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*获取指定活动实例 */
        WfActivityInstance actInst
  = client.getActivityInstance("actInstId");
        /*获取活动实例的状态 */
        String state = actInst.getState();
        /* 断开与工作流引擎连接 */
```

```
        client.disconnect();
//----------------------------------------获取活动实例状态的代码片段-结束
```

该功能返回指定活动实例的状态字符串。

7.3.5.2. 改变指定活动实例的状态

改变指定活动实例的状态的参考代码：

```
//----------------------------------------改变指定活动实例状态的代码片段-开始
   /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*改变指定活动实例的状态 */
        client.changeActivityInstanceState("actInstId", "newState");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------改变指定活动实例状态的代码片段-结束
```

改变指定的活动实例状态。该功能用于让一个活动实例暂时改变为一个指定的状态，如 suspended。此命令的执行将导致一个指定的活动实例转移到新的状态。

7.3.5.3. 活动的提交

活动的提交也就是将该活动结束，把该活动的相关数据提交给流程，再由流程来控制下一步走向。

注意：

- 如果活动定义的完成策略为“自动”，活动的提交由工作流引擎自动的执行；只有在活动定义的完成策略为“手动”的时候，需要客户端编写程序进行活动的提交；
- 当活动实例的所有工作项都完成的情况下才能提交活动。

活动的提交的参考代码：

```
//--------------------------------------------------活动的提交代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 活动的提交，将目标 ID 设置为空 */
        client.commitActivityInstance ("actInstId", null);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//--------------------------------------------------活动的提交代码片段-结束
```

7.3.5.4. 活动的跳转

通过调用此接口来实现“自由流”式的理想流程模型，即不用按照流程定义的约束来创建下一节点，通过指定下一步节点来实现流程的正常流转。

活动的跳转的参考代码：

```
//-------------------------------------------------活动的跳转代码片段-开始
 /* 构造请求服务相关的鉴别信息 */
        WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*活动的跳转，将目标 ID 设置为想要提交到的定义节点*/
        client.commitActivityInstance("actInstId",
                                      "targetActivityDefinitionId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-------------------------------------------------活动的跳转代码片段-结束
```

7.3.5.5. 获取所有可回退定义节点 ID

工作流支持选择性回退，可回退节点列表可通过工作流引擎获取；回退节点并不是能回退到所有节点，必须要满足引擎规范的条件。

获取所有可回退定义节点 ID 的参考代码：

```
//-----------------------------------获取所有可回退定义节点 ID 代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
```

```
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*获取所有可回退定义节点 ID */
       List actDefIds =
             client. listBackableActivities ("actInstId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------获取所有可回退定义节点 ID 代码片段-结束
```

7.3.5.6. 活动的回退

整个流程运转到当前节点，如果发现前几步的某个节点处理的不理想，可以通过调用活动的回退这个接口，实现让不理想的工作重新处理。

通过“获取所有可回退定义节点 ID”来选择回退的活动定义 ID。

活动的回退参考代码：

```
//-------------------------------------------------活动的回退代码片段-开始
      /* 构造请求服务相关的鉴别信息 */
           WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*活动的回退，将目标 ID 设置为想要回退到的活动实例定义 ID */
        client.backActivityInstance("actInstId",
                                "targetActivityDefinitionId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
```

```
//-------------------------------------------------活动的回退代码片段-结束
```

如果 targetActivityDefinitionId 的值为 null；系统将默认为回退到运行时的上一步节点。

7.3.5.7. 活动的放回

可以将活动的放回封装为操作，当前活动通过放回操作可以实现对活动的重做。

工作项的放回参考代码：

```
//-------------------------------------------------活动的放回代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
         WfClientManager cm = WfClientManager.getInstance();
         /* 首先获取客户端接口对象 */
         WfClient client = cm.getWfClient();
         /* 建立与工作流引擎连接 */
         client.connect(conInfo);
         /*活动的放回 */
         client.putbackActivityInstance ("actInstId");
         /* 断开与工作流引擎连接 */
         client.disconnect();
//-------------------------------------------------活动的放回代码片段-结束
```

7.3.5.8. 获取活动实例的属性列表

获取活动实例的属性列表参考代码：

```
//--------------------------------------获取活动实例的属性列表代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        WfFilter filter = null;
        /*获取活动实例属性列表 */
        Map attrs = client.listActivityInstanceAttributes("actInstId",
                                                          filter);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//--------------------------------------获取活动实例的属性列表代码片段-结束
```

返回满足过滤条件的指定活动的属性列表。该功能的用途是，让应用向工作流引擎询问已分配给活动实例的可用属性，而后将此列表提供给应用用户。

获取活动实例属性列表中指定的属性值参考代码：

```
//-------------------------获取活动实例属性列表中指定的属性值的代码片段-开始
        Object attrValue = attrs.get("attrId");
//-------------------------获取活动实例属性列表中指定的属性值的代码片段-结束
```

返回活动实例属性列表中指定的属性值。

7.3.5.9. 获取指定的活动实例的属性

获取指定的活动实例的属性参考代码：

```
//-----------------------------------获取指定的活动实例的属性代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*获取指定的活动实例的属性 */
        Object attrValue =
        client.getActivityInstanceAttributeValue(
  "actInstId","attrId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------获取指定的活动实例的属性代码片段-结束
```

返回指定活动实例的指定属性的值。

7.3.5.10. 分配一个属性值给指定的活动属性

分配一个属性值给指定的活动属性参考代码：

```
//--------------------------------分配一个属性值给指定的活动属性代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
```

```
        String attrValue = "Test";
        /*分配指定属性一个值 */
        client.assignActivityInstanceAttribute("actInstId","attrId",
                                                    attrValue);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//------------------------------分配一个属性值给指定的活动属性代码片段-结束
```

分配一个属性值给指定的活动属性。该功能通知工作流引擎为一个活动实例的属性赋值或改变活动实例的一个属性的值。活动实例的属性是活动控制和活动相关数据的一类。

注意：活动属性值的长度默认为 100 个字节，如果希望修改活动属性的长度可以通过如下操作完成：将 inforflow.jar 中的 com 目录解压，修改 com\cvicse\workflow\datastore\db\entity 目录下的 ActivityRelevantDataEntity.xml 文件和 WHActivityRelevantDataEntity.xml，将两个文件中的 <property name="value" type="java.lang.String" length="100" access = "field"/>中的 length 改成期望的值。修改完成以后需要重新生成数据库。

7.3.5.11. 取消活动实例

取消指定活动实例的参考代码：

```
//-----------------------------------------------取消活动实例代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
        WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
```

```
        client.abortActivityInstance ("actInstId",false);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------------取消活动实例代码片段-结束
```

取消一个活动实例。此命令提供取消活动实例的能力，将指定的未结束的活动实例取消掉，其相关数据值将不被拷贝到流程实例对象中，如果指定创建下一个节点的活动实例，则工作流引擎会按照流程定义自动创建出下一个活动实例，否则将不自动创建下一个活动实例。

7.3.6. 流程运行状况

通过定义过滤器 WfFilter 可以分别获取未启动的流程实例、运行中的流程实例、挂起的流程实例、终止掉的流程实例和取消掉的流程实例，可以获取与一个工作流参与者或一组工作流参与者相关的流程实例，也可以获取指定流程实例的运行状况；流程运行状况查询的请求可以由一个正常的工作流参与者发出，也可以由想查看其管理范围内某个人的工作进度的管理者或流程管理员发出。

7.3.6.1. 获取流程实例列表

获取流程实例列表参考代码：

```
//-----------------------------------------获取流程实例列表代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
```

```
        client.connect(conInfo);
        WfFilter filter = null;
        /* 获取流程实例列表 */
        List procInsts = client.listProcessInstances(filter);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------获取流程实例列表代码片段-结束
```

返回满足过滤条件的流程实例列表；该功能可被用于各种流程实例查询，如用此命令建立一个查询已结束或挂起流程实例的列表。

如果 filter 为空，则返回所有的流程实例列表。

7.3.6.2. 获取指定流程实例

获取指定的流程实例参考代码：

```
//----------------------------------------获取指定的流程实例代码片段-开始
      /* 构造请求服务相关的鉴别信息 */
           WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*获取指定的流程实例 */
        WfProcessInstance procInst =
                        client.getProcessInstance("procInstId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------获取指定的流程实例代码片段-结束
```

返回一个指定的流程实例记录。可根据此流程实例获取流程实例内的信息及其运行状况。

7.3.6.3. 生成流程实例运行图

IS-Flow 向用户提供了生成监控图的独立引擎，使用生成监控图功能的时候，需要配置 inforflow-image-client.xml 文件，可以选择嵌入引擎模式或者独立引擎模式，详细介绍请参考“其他设置信息”一章。

通过引擎的生成流程实例运行图功能，可以清晰直观的看到整个流程运转过程；引擎已提供获取运行中流程实例监控图和获取历史流程实例监控图的方法，具体实现过程请参考下面代码片断：

生成流程实例运行图参考代码：

```
//----------------------------------------生成流程实例运行图代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfImageClientManager cm = WfImageClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfImageClient imageclient = cm.getWfImageClient();
        /* 建立与工作流引擎连接 */
        imageclient.connect(conInfo);
        /*生成指定流程实例的运行图 */
        WfMonitorImage image =
    imageclient.getProcessInstance("procInstId",false);
        File main = File.createTempFile("Process_" + procInstId,
   ".JPEG", new File("curPath"));
        OutputStream os = new FileOutputStream(main);
        image.write(os);
        os.close();
```

```
        /* 断开与工作流引擎连接 */
        imageclient.disconnect();
//--------------------------------------生成流程实例运行图代码片段-结束
```

返回一个指定流程实例运行图的 WfMonitorImage 对象；可根据此对象生成运行图片直观的查看此流程实例运行状况，当然也可获取历史流程运行图查看历史流程的运行结果。

值得注意的是，WfMonitorImage 对象的获取也可以通过接口 public WfMonitorImage getWfMonitorImage(String processInstanceId, WfMonitorImageConfiguration configuration)获得流程监控图的 WfMonitorImage 对象，该对象使用给定绘图配置生成；其中 configuration 对象可以对生成的流程监控图的图片元素（各活动节点的图片）和监控颜色（运行的活动、结束的活动等配置不同的显示颜色）进行定制；例子代码中不带参数 configuration 对象时，使用工作流默认图片元素和监控颜色，图片元素使用目录%INFORSUITE_HOME%\repository\com\cvicse\inforflow\icons 下的图标作为默认值，监控颜色默认值为 WfMonitorImageConfiguration 中 stateColors（一组状态颜色 Map）对象，其中，

- 工作流活动状态 Global.NOT_STARTED 对应颜色 Color.green，
- 状态 Global.RUNNING 对应 颜色 Color.red，
- 状态 Global.SUSPENDED 对应颜色 Color.gray，
- 状态 Global.COMPLETED 对应 颜色 Color.yellow，
- 状态 Global.TERMINATED 对应颜色 Color.blue，
- 状态 Global.ABORTED 对应 颜色 Color.orange，
- 未被创建的活动背景颜色为 Color.white。

生成的流程进度图中活动节点的状态展示是如下逻辑：

1. 未结束的流程中展示活动的状态是取的当前活动的状态，即 curstate 字段（如果一个活动上有多个实例，会优先展示结束的那个实例的状态）
2. 已经结束的流程，流程图中展示活动的状态取的是 howclosed 字段，（如果一个活动上有多个实例，会优先展示最后的那个实例的状态）

IS-Flow 引擎提供了通过浏览器访问 Servlet 的方式产生流程实例运行图的方法。具体访问方式如下：

1) 通过 Local 或 Rmi 方式访问流程实例运行图，直接输入应用的 Url 地址：应用 url/ProcessPicture?proInstId=[流程实例号]即可；访问历史流程监控图还需要输入参数 isProcessClosed=true，访问运行时流程实例图片的时候也可加上参数 isProcessClosed=false。例如访问 helloworld 应用下流程实例号为 3 的流程图，可以写成：
http://localhost:8080/helloworld/ProcessPicture?procInstId=3；或者：
http://localhost:8080/helloworld/ProcessPicture?procInstId=3&isProcessClosed=false；如果访问 helloworld 应用下历史流程实例号为 5 的流程图，可以写成：
http://localhost:8080/helloworld/ProcessPicture?procInstId=5&isProcessClosed=true。
2) 通过 WebService 方式访问流程实例运行图，访问路径与应用无关，直接输入服务的 Url：服务的 Url/flowservice/ProcessPicture?procInstId=[流程实例号] 即可访问流程实例运行图；同上访问历史监控图需要添加参数 isProcessClosed=true。此时需要运行 run_nosec.bat 而不是 run.bat 来启动 WebService 服务，这种启动方式不使用 Web 服务器的安全策略，直接使用 JDK 的安全策略。例如通过本地默认 WebService 服务访问流程实例号为 10 的流程实例图的方式如下：
http://localhost:6522/flowservice/ProcessPicture?procInstId=10；访问历史流程实例号为 5 的历史流程实例监控图的方式如下：
http://localhost:6522/flowservice/ProcessPicture?procInstId=5&isProcessClosed=true。

7.3.7. 活动运行状况

活动运行状况方法着重于描述工作已经完成、即将要做，或者将工作和单一参与

者或者参与者组关联起来等等。一个运行状况查询可能被任何一个希望知道自己工作范围内的工作进度的人发起，它可能是一个普通的参与者或者是一个流程的管理员。

运行状况 API 的建立不仅能够从宏观上（流程）上察看工作进度，也能更进一步的查看一个单一活动实例的状态。

7.3.7.1. 获取活动实例列表

获取活动实例列表参考代码：

```
//-----------------------------------------获取活动实例列表代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        WfFilter filter = null;
        /*获取活动实例列表 */
        List actInsts = client.listActivityInstances(filter);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------获取活动实例列表代码片段-结束
```

该功能返回所有符合条件的活动实例列表。

该功能将用于定位较大的范围的活动实例，例如查询出已经完成或者正在处理的所有活动实例，当请求的内容为空的时候，返回所有的活动实例。

7.3.7.2. 获取指定活动实例

获取指定活动实例参考代码：

```
//----------------------------------------获取活动实例列表代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 获取指定的活动实例*/
        WfActivityInstance actInst =
                          client.getActivityInstance("actInstId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------获取活动实例列表代码片段-结束
```

该方法返回指定的活动实例，可根据返回的活动实例获取该活动的相应信息。

7.3.8. 工作项控制

活动和工作项都描述了一项工作，但是他们存在的时间点不同，并且一个活动可能对应了多个工作项，而且他们有各自的特有操作。

当一个流程定义完成以后（定义阶段），一个活动被用来描述有什么工作需要被处理；在运行期间，一个活动已经做好了处理的准备，而且已经有一个或多个的工作候选人，那么一个或多个工作项将被创建出来并放到候选人的工作列表中。

7.3.8.1. 获取工作项列表

获取工作项列表参考代码：

```
//-------------------------------------------获取工作项列表代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        WfFilter filter = null;
        /*获取工作项列表 */
        List workItems = client.listWorkItems(filter);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-------------------------------------------获取工作项列表代码片段-结束
```

通过构造 WfFilter 指定返回某一个参与者或者参与者组的符合某些条件的工作项列表。请求者可以代表他人或自己发出此请求，或者是一个管理员想知道哪些工作已分配给了一个指定的人或工作组而发出此请求。

7.3.8.2. 获取指定的工作项

获取指定的工作项参考代码：

```
//-------------------------------------------获取指定的工作项代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
```

```
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*获取指定的工作项 */
        WfWorkItem workItem = client.getWorkItem("workItemId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------获取指定的工作项代码片段-结束
```

返回指定的工作项，根据该工作项获取其相应信息。

7.3.8.3. 接收指定的工作项

处理工作项的状态为未接收状态的工作项，进一步再处理该工作项。

接收指定的工作项参考代码：

```
//----------------------------------------接收指定的工作项代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 接收指定的工作项 */
        client.acceptWorkItem("workItemId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------接收指定的工作项代码片段-结束
```

7.3.8.4. 完成指定的工作项

完成指定的工作项参考代码：

```
//-----------------------------------------完成指定的工作项代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
              /* 完成指定的工作项 */
        client.completeWorkItem("workItemId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------完成指定的工作项代码片段-结束
```

该功能允许工作流的参与者通知工作流引擎，指定的工作项已经完成。

一个工作项只对应一个活动，但是一个活动允许存在多个工作项，工作项的完成并不代表活动的所有工作已经完成。

7.3.8.5. 获取工作项状态

获取工作项状态参考代码：

```
//-----------------------------------------获取工作项状态代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
```

```
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*获取指定工作项 */
        WfWorkItem wi = client. getWorkItem ("workItemId");
        /*获取工作项状态 */
        String states = wi. getState ();
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------------获取工作项状态代码片段-结束
```

该功能返回指定工作项的状态。

7.3.8.6. 改变工作项状态

改变工作项状态参考代码：

```
//-----------------------------------------------改变工作项状态代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 改变工作项状态 */
        client.changeWorkItemState("workItemId","newState");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------------改变工作项状态代码片段-结束
```

改变指定的工作项的状态。该功能能够将工作项的状态暂时的更改成一个明确的

状态例如“运行中”、“未启动”。

7.3.8.7. 重新分配工作项

重新分配工作项参考代码：

```
//-------------------------------------------重新分配工作项代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
         WfClientManager cm = WfClientManager.getInstance();
         /* 首先获取客户端接口对象 */
         WfClient client = cm.getWfClient();
         /* 建立与工作流引擎连接 */
         client.connect(conInfo);
         /*重新分配工作项  */
         client.reassignWorkItem("sourceUser","targetUser",
                                 "workItemId");
         /* 断开与工作流引擎连接 */
         client.disconnect();
//-------------------------------------------重新分配工作项代码片段-结束
```

该功能允许将分配给一个参与者的工作项重新分配给另外一个参与者（加入到工作列表中）。分配的前提是重新分配的执行人需要有权限处理该工作项。

7.3.8.8. 任务代理执行人

IS-Flow 系统支持为原执行人指定代理人的功能；指定代理人后，在为原执行人创建未接收任务的时候，同时也为原执行人的代理人创建未接收任务，实现原执行人与其代理人抢任务的需求；其中会签或者单一执行人的情况下，为代理人创建任务，但是不再为原执行人创建任务；用户使用代理执行人功能时，只需设置原执行人 User 对象的 proxyUser 属性。

在 2.1.3 章节介绍了“用户数据集成”，IS-Flow 通过应用的 AbstractResourceProvider 实现类来获取应用的用户资源和用户信息。

如何修改原执行人 User 对象的 proxyUser 属性，来达到设置任务的代理执行人和取消任务的代理执行人？首先，在用户接口实现中，遍历每个 User 对象时为其设置 proxyUser 属性，此 proxyUser 属性是由用户提供，其值为 null 或 "" 时 IS-Flow 系统不再为代理执行人创建工作项，这样用户可以通过修改业务系统 proxyUser 的值来控制设置代理执行人和取消代理执行人。

在任务项未接收前，可以临时指定代理人，可通过接口 addWorkItem(String actInstId, String groupId, String participantId)针对某个活动单独为代理人添加未接收任务，任务项执行完毕，临时代理人被自动取消。

也可通过接口 deleteWorkItem("workItemId", ture)取消代理人的未接收任务。

在取消代理执行人后，引擎为原执行人提供了夺回被代理任务的接口 retakeProxyWorkItem (String sourceUser, String workItemId)：

1) 代理执行人工作项为未接收状态的，原执行人夺回时将代理执行人的未接收工作项删除，即不允许代理执行人抢任务；
2) 代理执行人工作项为运行状态的，原执行人夺回时需要将代理执行人的运行工作项重分配到原执行人。

夺回/临时指定/取消 代理执行人任务参考代码：

```
//-----------------------------夺回/临时指定/取消 代理执行人任务代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 临时指定代理执行人任务 */
```

```
        client.addWorkItem("actInstId", "groupId", "participantId");
        /* 取消代理执行人未接收任务 */
        client.deleteWorkItem ("workItemId", true);
        /* 夺回代理执行人任务 */
        client.retakeProxyWorkItem ("sourceUser", "workItemId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------夺回/临时指定/取消 代理执行人任务代码片段-结束
```

7.3.8.9. 获取工作项属性列表

获取工作项属性列表参考代码：

```
//----------------------------------------获取工作项属性列表代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        WfFilter filter = null;
        /* 获取工作项属性列表 */
        Map attrs = client.listWorkItemAttributes(
  "workItemId",filter);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------获取工作项属性列表代码片段-结束
```

该功能返回一个工作项的所有符合查询条件的属性列表。

7.3.8.10. 获取工作项属性列表中的值

获取工作项属性列表中的值参考代码：

```
//-----------------------------------获取工作项属性列表中的值代码片段-开始
        Object attrValue = attrs.get ("attrId");
//-----------------------------------获取工作项属性列表中的值代码片段-结束
```

7.3.8.11. 获取工作项属性值

获取工作项属性值参考代码：

```
//-------------------------------------------获取工作项属性值代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
         WfClientManager cm = WfClientManager.getInstance();
         /* 首先获取客户端接口对象 */
         WfClient client = cm.getWfClient();
         /* 建立与工作流引擎连接 */
         client.connect(conInfo);
         /*获取工作项属性值  */
         ObjectattrValue =
            client.getWorkItemAttributeValue("workItemId","attrId");
         /* 断开与工作流引擎连接 */
         client.disconnect();
//-------------------------------------------获取工作项属性值代码片段-结束
```

该功能返回指定工作项的指定属性的属性值。

7.3.8.12. 工作项属性赋值

为工作项属性赋值参考代码：

```
//-----------------------------------------为工作项属性赋值代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        String attrValue = "example";
        /*为工作项属性赋值 */

    client.assignWorkItemAttribute("workItemId","attrId" ,attrValue);
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------为工作项属性赋值代码片段-结束
```

该功能通知工作流引擎修改指定工作项的指定属性的值。

注意：工作项属性值的长度默认为100 个字节，如果希望修改工作项属性的长度可以通过如下操作完成：将 inforflow.jar 中的 com 目录解压，修改 com\cvicse\workflow\datastore\db\entity 目录下的 WorkItemRelevantDataEntity.xml 文件和 WHWorkItemRelevantDataEntity.xml 文件，将两个文件中的语句 <property name="value" type="java.lang.String" length="100" access = "field"/> 中的length 改成期望的值，修改完成以后需要重新生成数据库。

7.3.9. 工具调用

以下提供的这组功能用于提供工具代理服务以及调用和控制工作项的外部应用程序。

工作流专门提供的一套用于调用外部应用程序的功能接口，该功能接口被用于工作流系统的组件中（引擎和应用程序客户端）来控制专用的应用程序驱动，我们称之

为工具代理。工具代理启动和结束应用程序，通过工作流相关数据信息的交互来控制应用程序的运行状态。

7.3.9.1. 获取应用程序列表

获取应用程序列表参考代码：

```
//-----------------------------------------获取应用程序列表代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        String procDefId = "test";
        /* 获取应用程序列表 */
        List appInsts = client.listApplicationInstances ("workItemId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------获取应用程序列表代码片段-结束
```

该功能返回指定工作项的所有应用程序。

7.3.9.2. 获取指定的应用实例

获取指定的应用实例参考代码：

```
//-----------------------------------------获取指定的应用实例代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
```

```
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 获取指定的应用实例 */
        WfApplicationInstance appInst = client.getApplicationInstance
                                                ("appInstId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------获取指定的应用实例代码片段-结束
```

该功能返回指定的应用实例。

7.3.9.3. 启动应用实例

启动应用实例参考代码：

```
//----------------------------------------------启动应用实例代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /*启动应用实例 */
        client.startApplicationInstance ("appInstId");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//----------------------------------------------启动应用实例代码片段-结束
```

该功能通知工作流引擎启动指定应用实例，改变应用实例状态为“处理中”。

7.3.9.4. 结束应用实例

结束应用实例参考代码：

```
//-----------------------------------------------结束应用实例代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
         WfClientManager cm = WfClientManager.getInstance();
         /* 首先获取客户端接口对象 */
         WfClient client = cm.getWfClient();
         /* 建立与工作流引擎连接 */
         client.connect(conInfo);
         /* 结束应用实例 */
         client.completeApplicationInstance ("appInstId");
         /* 断开与工作流引擎连接 */
         client.disconnect();
//-----------------------------------------------结束应用实例代码片段-结束
```

该功能通知工作流引擎已完成对应用实例的调用，改变应用实例的状态为“处理完成”。

7.3.10. 工作时间计算

根据工作流引擎当前配置的日历，计算一个时间段中的绝对工作时间长度，即除去日历中定义的非工作时间之外的纯工作时间段长度。此接口主要用于计算工作流实例的两个状态之间的绝对工作时间，比如流程实例从创建到结束的绝对工作时间长度。

工作时间计算参考代码：

```
//-----------------------------------------------工作时间计算代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfClientManager cm = WfClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfClient client = cm.getWfClient();
        /* 建立与工作流引擎连接 */
        client.connect(conInfo);
        /* 工作时间计算 */
              client. calculateWorkingTime ("startTimeInMills", "endTimeInMills");
        /* 断开与工作流引擎连接 */
        client.disconnect();
//-----------------------------------------------工作时间计算代码片段-结束
```

"startTimeInMills"为开始时间点(单位为微秒)，"endTimeInMills"为结束时间点(单位为微秒)。如果用户没有为引擎正确的配置好日历，则会抛出找不到日历异常。此时用户应该检查 inforflow.xml 文件中是否正确的设置了日历。

日历的配置详见 7.5 节 calendar.xml 文件的配置。

7.3.11. 流程定义信息

IS-Flow 除了对用户提供了运行流程信息和历史流程信息的获取、修改接口外，还提供了对流程定义信息获取的接口，方便用户根据流程定义信息做一些逻辑上的判断，例如常用的一些流程定义、活动定义、应用定义、工具定义的一些扩展属性等；这里就不再一一介绍流程定义信息的获取了，用户可以通过查看 IS-Flow 提供的 api，清晰的了解到流程中各定义信息的获取。

7.3.11.1. 修改流程定义模板

在实际生产环境中经常会碰到这样的情况，设计出来的流程定义经过测试发布到生产环境上后，由于测试过程的疏忽或者需求上的临时变更，在生产运行过程中发现某些不如人意的漏洞甚至是严重缺陷；这个时候我们需要在不影响这些模板的运行实例正常运转的条件下，对这些流程模板进行修改，尽可能弥补或者修复这些缺陷所带来的损失；通过修改流程定义模板功能可以实现这样的需求。

例如：流程实例 A 在运转过程中，发现流程实例 A 对应流程模板 MA 中的某个活动定义信息不对，为了不影响当前流程实例 A 的继续运行，同时不影响根据流程模板 MA 已经发起的其他流程实例，通过修改流程模板 MA 的方式来修复模板的缺陷活动定义，进而保证流程实例 A 能继续向下运转，让其他未发现此缺陷的流程实例避免遇到此缺陷。

由于 IS-Flow 引擎已提供的各项功能，使修改流程模板要受到一些限制，不能够像新建流程定义那样对流程模板进行随意修改，毕竟根据当前模板已经发起了不少流程实例，这些流程实例要严格根据模板进行运转；例如引擎提供的回退功能，如果你把流程模板中的某个活动节点删除掉，而回退的时候还要往这个活动节点回退，这样就很容易使引擎报找不到活动实例对应活动节点模板的错误。

因此，修改流程模板功能仅提供对某些定义信息的修改，例如：流程定义中的参与者信息、应用信息、相关数据信息、活动信息、活动下的工具信息、活动下的任务分配信息以及连接弧信息；对于那些增加节点、删除节点、调整节点位置、变更流程执行路径等较大范围的调整，修改流程模板功能未提供修改接口；

可以说，流程定义较大范围的调整，应该属于流程逻辑上的需求变更，则必须使用导入工具升级流程模板的版本，保证以后新发起流程按照新的模板运行，已经运转的流程只能按照原模板继续运转。

下面举例介绍下修改流程定义模板的接口：

调用此功能接口前，首先需要确定要修改流程定义哪部分内容，是针对流程还是针对活动，是修改还是增加还是删除等，确定选用哪个接口；例如，给流程模板增加应用定义，需要调用 WfManagerClient 的 addProcessApplication 接口；然后还需要知

道为流程增加应用，需要增加哪些应用属性，这部分可以参考 xpdl 文件中应用定义各属性元素，也可以对照设计器，最终理解 api 注释中提示的那些属性参数，构造 map 对象当参数传入接口。

具体实现方法可参考下面代码片断：

修改流程定义模板参考代码：

```
//-----------------------------------------修改流程定义模板代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
         WfConnectInfo conInfo = new WfConnectInfo("userId");
      WfManagerClientFactory cf = WfManagerClientFactory.getInstance();
         /* 首先获取客户端接口对象 */
         WfManagerClient managerClient = cf.getWfManagerClient();
         /* 建立与工作流引擎连接 */
         managerClient.connect(conInfo);
         /* 为流程模板增加应用定义 */
         Map properties = new HashMap();
         properties.put(WfManagerClient.app_name,"test");
         properties.put(WfManagerClient.app_description,"test_add");

         Map extAttrs = new HashMap();
         extAttrs.put(WfManagerClient.app_Type, "Application");
         extAttrs.put(WfManagerClient.app_ApplicationType, "Jsp");
         extAttrs.put(WfManagerClient.app_url, "/helloworld.jsp");
         extAttrs.put("test", "test_addApplication");

         Map formalPars = new HashMap();
         managerClient.addProcessApplication("procModelId",
  "test", properties, extAttrs, formalPars);
         /* 断开与工作流引擎连接 */
```

```
        managerClient.disconnect();
//-----------------------------------------修改流程定义模板代码片段-结束
```

修改流程定义模板此类的接口，对使用人员要求比较高，需要对流程定义信息相当明确，尤其在为流程模板增加或者修改元素的时候，需对被操作元素各属性理解，并能与实际需求结合达到增加或修改的目的。

7.3.11.2. 获取流程定义图

IS-Flow 的生成监控图引擎，提供了对流程定义图导出的功能；通过此功能可以在应用中随时获取流程定义图片用于展示，具体实现方法可参考下面代码片断：

生成流程定义图参考代码：

```
//-----------------------------------------生成流程定义图代码片段-开始
       /* 构造请求服务相关的鉴别信息 */
            WfConnectInfo conInfo = new WfConnectInfo("userId");
        WfImageClientManager cm = WfImageClientManager.getInstance();
        /* 首先获取客户端接口对象 */
        WfImageClient imageclient = cm.getWfImageClient();
        /* 建立与工作流引擎连接 */
        imageclient.connect(conInfo);
        /*生成指定流程模板的流程定义图 */
        WfMonitorImage image =
         imageClient.getProcessDefImage ("procModelId");
       File main = File.createTempFile("ProcModel_" + procModelId,
   ".JPEG", new File("curPath"));
        OutputStream os = new FileOutputStream(main);
        image.write(os);
        os.close();
        /* 断开与工作流引擎连接 */
```

```
        imageclient.disconnect();
//-------------------------------------------生成流程定义图代码片段-结束
```

同生成流程实例运行图，IS-Flow 引擎也提供了通过浏览器访问 Servlet 的方式产生流程定义图的方法。具体访问方式如下：

1) 通过 Local 或 Rmi 方式访问流程定义图，直接输入应用的 Url 地址：应用 url/ProcessPicture?proModelId=[流程定义模板号]即可。例如访问 helloworld 应用下流程模板号为 3 的流程定义图，可以写成：http://localhost:8080/helloworld/ProcessPicture?procModelId=3。
2) 通过 WebService 方式访问流程定义图，访问路径与应用无关，直接输入服务的 Url：服务的 Url/flowservice/ProcessPicture?procModelId=[流程定义模板号] 即可访问流程定义图。此时需要运行 run_nosec.bat 而不是 run.bat 来启动 WebService 服务，这种启动方式不使用 Web 服务器的安全策略，直接使用 JDK 的安全策略。例如通过本地默认 WebService 服务访问流程定义模板号为 10 的流程定义图的方式如下：http://localhost:6522/flowservice/ProcessPicture?procModelId=10。

7.4. 高级编程

7.4.1. 集成用户数据

本节介绍如何通过编程将应用系统中的用户数据集成到工作流引擎中，也即在工作流引擎中使用应用系统的用户数据。IS-Flow 提供两个抽象类供用户扩展，即

- com.cvicse.workflow.api.resources.AbstractResourceProvider
- com.cvicse.workflow.api.resources.DyAbstractResourceProvider。

其中，AbstractResourceProvider 的实现类通过读取数据库的方式将应用系统的用户数据集成到工作流引擎中。DyAbstractResourceProvider 的实现类除通过数据库方式外还可以通过在程序代码中动态生成用户数据并集成到工作流引擎中。一般情况下，实现 AbstractResourceProvider 即可满足多数应用系统的需求。如果应用系统需要通过

接口在程序代码中动态生成用户数据，则需要实现抽象类 DyAbstractResourceProvider。

下面介绍用户数据集成的具体实现方式。

◆ 方式一，实现 AbstractResourceProvider。

需要做的工作如下：

1. 实现抽象类
   com.cvicse.workflow.api.resources.AbstractResourceProvider
2. 将 AbstractResourceProvider 的实现类配置到 inforflow.xml 中（参见用户数据集成）

工作流引擎给出了默认的用户接口实现类，即 com.cvicse.workflow.examples.resource.AppResourceProvider。参见 IS-Flow 产品示例 helloworld 应用下的源文件 AppResourceProvider.java。

◆ 方式二，实现 DyAbstractResourceProvider。

需要做的工作如下：

1. 实现抽象类
   com.cvicse.workflow.api.resources.DyAbstractResourceProvider
2. 将 DyAbstractResourceProvider 的实现类配置到 inforflow.xml 中（参见 2.1.3 节）

DyAbstractResourceProvider 接口中的方法，见表格 7-2，

表格 7-2 DyAbstractResourceProvider 接口中的方法

方法返回值	方法描述
User	getUser(String userId, WfActivityInstance wfActivityInstance) 根据活动实例信息和执行人标示号返回执行人对象
User[]	getUsers(WfActivityInstance wfActivityInstance) 根据活动实例信息返回执行人对象数组
User[]	getUsers(String roleId, String orgnizationUnitId, WfActivityInstance wfActivityInstance) 根据活动实例信息返回指定角色和机构代码的执行人对象数组

User[]	getUsers(String roleId, WfActivityInstance wfActivityInstance) 根据活动实例信息返回指定角色的执行人对象数组
boolean	validateResource(String resourceId, String password, String resourceType) 返回效验资源信息结果

下面的示例代码展示了方法 getUser(String userId, WfActivityInstance wfActivityInstance)的实现。

```
public User getUser(String userId,
WfActivityInstance actInst) {
    String procInstId = actInst.getProcessInstanceId();
    String procDefId = actInst.getProcDefId().trim();
    String actDefId = actInst.getActDefinitionId().trim();
    if("指定流程定义ID".equals(procDefId) &&
       "指定活动定义ID".equalsIgnoreCase(actDefId)){
        return new User("id", "张三", "描述");
    }else{
        return getUser(userId);
    }
}
```

7.4.2. 编写流程上下文实现类

定义流程定时启动时，如果流程具有相关数据，只需实现接口 com.cvicse.workflow.api.scheduler.ProcessContextProvider 给流程相关数据赋值。

接口 ProcessContextProvider.java 的代码如下：

ProcessContextProvider.java:

```
package com.cvicse.workflow.api.scheduler;

import java.util.Map;

/**
 * 定时启动流程时，需要初始化相关数据，应用可实现此接口，
 *并配置在流程定时启动中，<br>
 * 工作流引擎在执行定时启动流程时会动态加载实现类，并设置到流程实例中。
```

```
 */
public interface ProcessContextProvider {
    /**
     * 获取定时启动时所需要的流程上下文，
     * key:相关数据ID,value:相关数据的值
     * @return java.util.Map
     */
    Map getProcessContextProviderMap();
}
```

该接口实现类 ProcessContextImpl 的示例代码如下：

ProcessContextProviderImpl.java:

```
package com.cvicse.workflow.api.scheduler;

import java.util.HashMap;
import java.util.Map;

public class ProcessContextImpl
implements ProcessContextProvider {

    public Map getProcessContextProviderMap() {
        Map context = new HashMap();
        //假定相关数据id为number
        context.put("number", "100000");
        return context;
    }
}
```

在流程设计器的流程“定时启动”窗口中，指定流程上下文实现类中输入实现类的名称，如图表 7-1 所示：

图表 7-1 设置流程上下文实现类

7.4.3. 编写事件监听器

7.4.3.1. 流程事件监听器

流程事件监听器为接口 com.cvicse.workflow.api.event.WfProcessInstanceListener，用户可以根据应用的需求实现接口中相应的方法。工作流引擎提供了该接口的默认实现类 ProcessListenerExample，它位于包 com.cvicse.workflow.examples 下。关于 ProcessListenerExample 的详细内容参考 IS-Flow 产品示例 helloworld 应用下的源码 ProcessListenerExample.java。

7.4.3.2. 活动事件监听器

活动事件监听器为接口 com.cvicse.workflow.api.event.WfActivityInstanceListener，用户可以根据应用的需求实现接口中相应的方法。工作流引擎提供了该接口的默认实现类 ProcessListenerExample2，它位于包 com.cvicse.workflow.examples 下。关于 ActivityListenerExample2 的详细内容参考 IS-Flow 产品示例 helloworld 应用下的源码 ActivityListenerExample2.java。

7.4.3.3. 工作项事件监听器

流程事件监听器为接口 com.cvicse.workflow.api.event.WfWorkItemListener，用户可以根据应用的需求实现接口中相应的方法。工作流引擎提供了该接口的默认实现类 WorkItemListenerExample，它位于包 com.cvicse.workflow.examples 下。关于 WorkItemListenerExample 的详细内容参考 IS-Flow 产品示例 helloworld 应用下的源码 WorkItemListenerExample.java。

7.4.4. 编写执行期限自定义事件

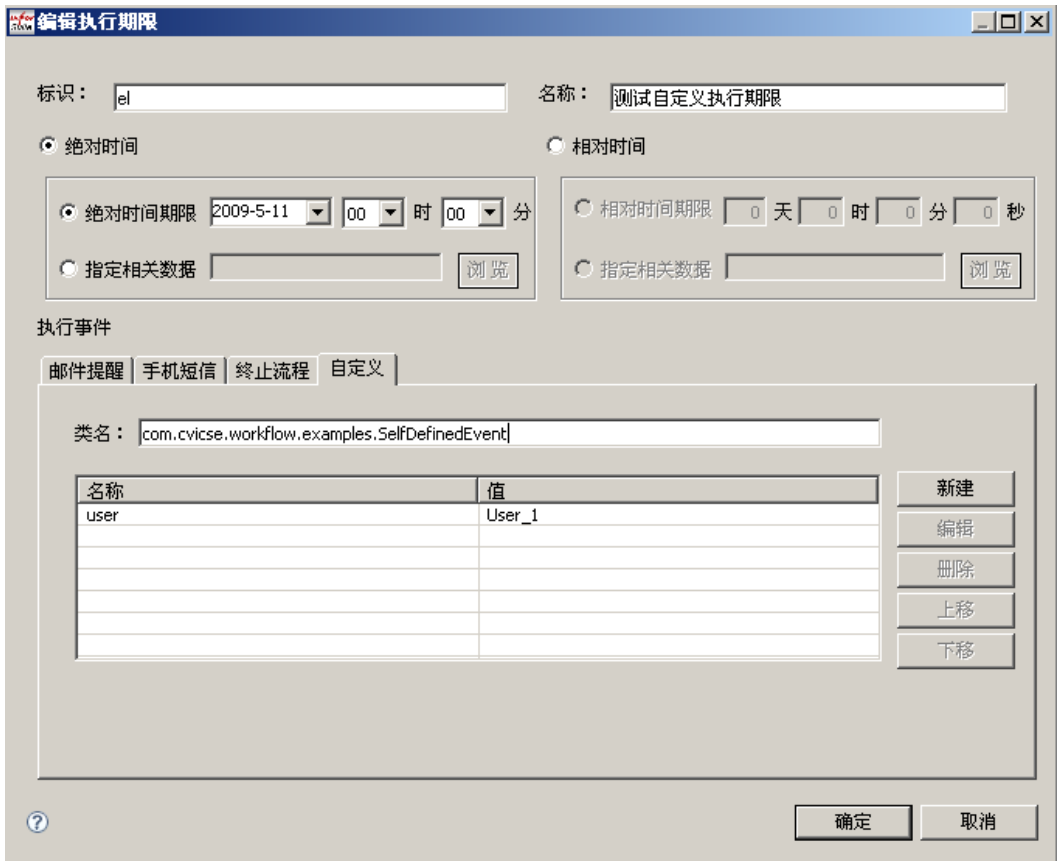
7.4.4.1. 流程执行期限自定义事件

自定义流程执行期限事件需要实现接口 com.cvicse.workflow.api.scheduler.ExecutionLimitEvent。接口 ExecutionLimitEvent 中的方法见表格 7-3 ExecutionLimitEvent 中的方法：

表格 7-3 ExecutionLimitEvent 中的方法

方法返回值	方法描述
void	executeEvent(List contextList) /** * 应用实现的催办方法 * @param contextList 用于存放用户在自定义事件中定义的上下文信息，以及流程实例标识或者工作项实例标识(按照该执行期限定义在流程上还是活动上而定)。 *contextList包含的对象为com.cvicse.workflow.api.xpdlext.Context。 *其中存放工作项实例标识的com.cvicse.workflow.api.xpdlext.Context对象key为InforFlowKernelConstants.JOB_CONTEXT_WORKITEMINSTANCEID *存放工作项实例标识的com.cvicse.workflow.api.xpdlext.Context对象的key为InforFlowKernelConstants.JOB_CONTEXT_PROCESSINSTANCEID */

流程设计器中流程执行期限自定义事件窗口如图表 7-2 所示：



图表 7-2 流程设计器中流程执行期限自定义事件窗口

自定义事件类 SeflDefinedEvent 的代码片段如下：

SelfDefinedEvent.java:

```
package com.cvicse.workflow.examples;

import java.util.List;

import com.cvicse.workflow.api.scheduler.ExecutionLimitEvent;
import com.cvicse.workflow.api.xpdlext.Context;
```

```
public class SelfDefinedEvent implements ExecutionLimitEvent {

    public void executeEvent(List contextList) {
        //获取自定义事件的属性值，这里只获取第一个属性
        Context context = (Context)contextList.get(0);
        String name = context.getName();
        String value = context.getValue();
        //获取相应的属性之后，很容易编写事件的代码，这里略。
        // TODO
    }
}
```

7.4.4.2. 活动执行期限自定义事件

与流程执行期限自定义事件类似，自定义活动执行期限事件需要实现接口 com.cvicse.workflow.api.scheduler.ExecutionLimitEvent。具体详见流程执行期限自定义事件。

7.4.5. 扩展工作流引擎

用户扩展工作流引擎，只需继承 com.cvicse.workflow.core.DefaultWAPIDecorator 类。工作流引擎提供了 3 个扩展引擎的示例，示例源码 AppWorkflowEngine.java、AppWorkflowEngine2.java 和 AppWorkflowEngine3.java 位于 IS-Flow 产品示例 helloworld 应用 classes 目录的 com.cvicse.workflow.examples 包下。

7.4.6. 扩展工具代理类

工作流引擎提供了默认的工具代理类，如 WebService 工具代理类、EJB 代理类等。如果这些工具代理类不能满足应用系统的需求，那么用户可以自定义扩展，只需实现 com.cvicse.workflow.toolagent.AbstractToolAgent。

7.4.7. 扩展缓存功能

工作流引擎提供了服务器端的缓存功能，用户可以根据业务系统的需要来扩展。

扩展缓存功能需要做的工作如下：

1. 扩展接口 com.cvicse.workflow.api.ehcache.WorkflowCacheObject。
2. 编写一个实现类，实现 1 中的扩展接口。这个实现类就是缓存功能用到的缓存对象，它的属性为应用系统中用到的工作流对象的属性。
3. 扩展抽象类 com.cvicse.workflow.api.ehcache.WorkflowCache。
4. 扩展抽象类 com.cvicse.workflow.ehcache.pool.CacheObjectsPool。
5. 扩展抽象类 com.cvicse.workflow.examples.ehcache.listener.CreateWorkItemListener
6. 扩展抽象类 com.cvicse.workflow.examples.ehcache.listener.DestroyWorkItemListener
7. 扩展抽象类 com.cvicse.workflow.examples.ehcache.listener.UpdateWorkItemListener
8. 配置 inforflow-cache.xml

下面给出每个步骤的示例代码。

- 扩展接口 com.cvicse.workflow.api.ehcache.WorkflowCacheObject。

XNCacheObject.java:

```
package com.cvicse.workflow.examples.ehcache;

import com.cvicse.workflow.api.ehcache.WorkflowCacheObject;

public interface XNCacheObject extends WorkflowCacheObject{
    public String getProcessInstanceId();
    public void setProcessInstanceId(String id);
    public String getProcessInstanceName();
    public void setProcessInstanceName(String name);
    public String getActivityInstanceId();
    public void setActivityInstanceId(String id);
    public String getActivityInstanceName();
    public void setActivityInstanceName(String name);
```

```
    public String getWorkItemId();
    public void setWorkItemId(String workItemId);
    public String getPerformer();
    public void setPerformer(String performer);
    public String getStartTime();
    public void setStartTime(String startTime);
    public String getCurState();
    public void setCurState(String curState);
    public String getActDefId();
    public void setActDefId(String actDefId);
}
```

◆ 编写一个实现类，实现 1 中的扩展接口。

XNCacheObjectImpl.java:

```
package com.cvicse.workflow.examples.ehcache;

public class XNCacheObjectImpl implements XNCacheObject{

 private static final long serialVersionUID = 7348973458883318601L;

    private String processInstanceId;
    private String processInstanceName;
    private String activityInstanceId;
    private String activityInstanceName;
    private String workItemId;
    private String performer;
    private String startTime;
    private String curState;
    private String actDefId;
    /**
     * 属性的setter和getter方法省略
     */
    ...
}
```

◆ 扩展抽象类 com.cvicse.workflow.api.ehcache.WorkflowCache。

XNWorkflowCache.java:

```
package com.cvicse.workflow.examples.ehcache;

import java.util.List;
```

```
import com.cvicse.workflow.api.WfClient;
import com.cvicse.workflow.api.WfClientManager;
import com.cvicse.workflow.api.WfConnectInfo;
import com.cvicse.workflow.api.WfWorkItem;
import com.cvicse.workflow.api.ehcache.WorkflowCache;
import com.cvicse.workflow.api.ehcache.WorkflowCacheObject;
import com.cvicse.workflow.ehcache.CacheUtil;

public class XNWorkflowCache extends WorkflowCache {
    private static final long serialVersionUID = 1L;

    public XNWorkflowCache(String cacheNames, String
indexProperty) {
        super(cacheNames, indexProperty);
    }
    protected void initializeCache(String name, Object key) {
    }
    protected Object initializeObject(Object key) {
        return null;
    }
    public void initAllRunningProcess() {
        WfClient client =
WfClientManager.getInstance().getWfClient();
        try {
            WfConnectInfo connectInfo = new
WfConnectInfo("User_1");
            client.connect(connectInfo);

            List list = client.listWorkItems(null);
            WorkflowCacheObject db = null;
            XNCacheObject obj = null;;
            if (CacheUtil.getInstance().isUseCacheObjectsPool()) {
                for (int i = 0; i < list.size(); i++) {
                    WfWorkItem workItem = (WfWorkItem) list.get(i);
                    db =
(WorkflowCacheObject)CacheUtil.getInstance().getObjectsPool().g
et();
                    obj = (XNCacheObject) db.getMyself();
                    obj.setWorkItemId(workItem.getId());
                    obj.setPerformer(workItem.getParticipant());
```

```
                    obj.setStartTime(workItem.getReceiveTime() +
"");
                    obj.setActivityInstanceId(
workItem.getActivityInstanceId());

    obj.setActivityInstanceName(workItem.getName());
                    obj.setProcessInstanceId(
workItem.getProcessInstanceId());
                    obj.setCurState(workItem.getState());
                    obj.setActDefId(workItem.getActDefId());
                    put(db);
                }
            }else {
                for (int i = 0; i < list.size(); i++) {
                    WfWorkItem workItem = (WfWorkItem) list.get(i);
                    obj = new XNCacheObjectImpl();
                    obj.setWorkItemId(workItem.getId());
                    obj.setPerformer(workItem.getParticipant());
                    obj.setStartTime(workItem.getReceiveTime() +
"");
                    obj.setActivityInstanceId(
workItem.getActivityInstanceId());

    obj.setActivityInstanceName(workItem.getName());
                    obj.setProcessInstanceId(
workItem.getProcessInstanceId());
                    obj.setCurState(workItem.getState());
                    obj.setActDefId(workItem.getActDefId());
                    put(obj);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            try {
                client.disconnect();
            } catch (RuntimeException e1) {
                e1.printStackTrace();
            }
        }
    }
```

```
}
```

- 扩展抽象类 com.cvicse.workflow.ehcache.pool.CacheObjectsPool。

XNCacheObjectsPool.java:

```
package com.cvicse.workflow.examples.ehcache;

import com.cvicse.workflow.ehcache.pool.CacheObjectsPool;

public class XNCacheObjectsPool extends CacheObjectsPool {

    private int  count = 0;

    public XNCacheObjectsPool(int minObjectsNum) {
        super(minObjectsNum);
    }

    public XNCacheObjectsPool() {
        super();
    }

    public Object initCachedObject() {
        return new XNCacheObjectImpl();
    }
}
```

- 扩展抽象类 com.cvicse.workflow.examples.ehcache.listener. CreateWorkItemListener

XNCreateWorkItemListener.java:

```
package com.cvicse.workflow.examples.ehcache.listener;

import java.util.List;

import com.cvicse.workflow.api.WfActivityInstance;
import com.cvicse.workflow.api.WfClient;
import com.cvicse.workflow.api.WfClientManager;
import com.cvicse.workflow.api.WfConnectInfo;
import com.cvicse.workflow.api.WfFilter;
import com.cvicse.workflow.api.WfProcessInstance;
import com.cvicse.workflow.api.WfWorkItem;
```

```
import com.cvicse.workflow.api.ehcache.WorkflowCacheObject;
import
com.cvicse.workflow.api.ehcache.listener.CreateWorkItemListener;
import com.cvicse.workflow.api.query.entity.RunningEntity;
import com.cvicse.workflow.ehcache.CacheUtil;
import com.cvicse.workflow.ehcache.ParallelCache;
import com.cvicse.workflow.examples.ehcache.XNCacheObject;
import com.cvicse.workflow.examples.ehcache.XNCacheObjectImpl;

public class XNCreateWorkItemListener
extends CreateWorkItemListener {

    public XNCreateWorkItemListener() {
    }

    public XNCreateWorkItemListener(ParallelCache manager) {
        super(manager);
    }

    public void afterCreate(WfWorkItem workItem,
WfActivityInstance activity, WfProcessInstance process)
throws Exception {
        if (CacheUtil.getInstance().isUseCacheObjectsPool()) {
            WorkflowCacheObject  db =  (WorkflowCacheObject)
CacheUtil.getInstance().getObjectsPool().get();
            XNCacheObject obj= (XNCacheObject) db.getMyself();
            obj.setWorkItemId(workItem.getId());
            obj.setPerformer(workItem.getParticipant());
            obj.setStartTime(workItem.getReceiveTime() + "");
            obj.setActivityInstanceId(
workItem.getActivityInstanceId());
            obj.setActivityInstanceName(workItem.getName());
            obj.setProcessInstanceId(
workItem.getProcessInstanceId());
            obj.setCurState(workItem.getState());
            obj.setActDefId(workItem.getActDefId());

            //这里new一个新的Element会不会有问题？产生重复记录？
            getManager().put(db);
//加入到manager之后，obj变成一个被加强的受控对象。
        }else {
```

```
            XNCacheObject obj = new XNCacheObjectImpl();
            obj.setWorkItemId(workItem.getId());
            obj.setPerformer(workItem.getParticipant());
            obj.setStartTime(workItem.getReceiveTime() + "");
            obj.setActivityInstanceId(
workItem.getActivityInstanceId());
            obj.setActivityInstanceName(workItem.getName());
            obj.setProcessInstanceId(
workItem.getProcessInstanceId());
            obj.setCurState(workItem.getState());
            obj.setActDefId(workItem.getActDefId());

            //这里new一个新的Element会不会有问题？产生重复记录？
            getManager().put(obj);
//加入到manager之后，obj变成一个被加强的受控对象。
            //测试是否是被加强。
    //updateWorkitemPerformer(workitem.getWorkitemId(), "002");
        }
    }

    public void afterReviveProcess(String procInstId)
throws Exception {
        WfClient client =
WfClientManager.getInstance().getWfClient();
        WfConnectInfo connectInfo = new WfConnectInfo("System");
        client.connect(connectInfo);
        List wis = client.listWorkItems(new
WfFilter(RunningEntity.WorkItemProcessId,
WfFilter.EXPRESSION_EQUAL, procInstId));
        client.disconnect();

        WfWorkItem workItem;
        WorkflowCacheObject db = null;
        XNCacheObject obj = null;
        if (CacheUtil.getInstance().isUseCacheObjectsPool()) {
            for(int i = 0; i < wis.size(); i++){
                workItem = (WfWorkItem)wis.get(i);
                db =  (WorkflowCacheObject)
CacheUtil.getInstance().getObjectsPool().get();
                obj = (XNCacheObject) db.getMyself();
```

```
                obj.setWorkItemId(workItem.getId());
                obj.setPerformer(workItem.getParticipant());
                obj.setStartTime(workItem.getReceiveTime() + "");
                obj.setActivityInstanceId(
workItem.getActivityInstanceId());
                obj.setActivityInstanceName(workItem.getName());
                obj.setProcessInstanceId(
workItem.getProcessInstanceId());
                obj.setCurState(workItem.getState());
                obj.setActDefId(workItem.getActDefId());
                getManager().put(db);
            }
        }else {
            for(int i = 0; i < wis.size(); i++){
                workItem = (WfWorkItem)wis.get(i);
                obj = new XNCacheObjectImpl();
                obj.setWorkItemId(workItem.getId());
                obj.setPerformer(workItem.getParticipant());
                obj.setStartTime(workItem.getReceiveTime() + "");
                obj.setActivityInstanceId(
workItem.getActivityInstanceId());
                obj.setActivityInstanceName(workItem.getName());
                obj.setProcessInstanceId(
workItem.getProcessInstanceId());
                obj.setCurState(workItem.getState());
                obj.setActDefId(workItem.getActDefId());

                //这里new一个新的Element会不会有问题？产生重复记录？
                getManager().put(obj);
//加入到manager之后，obj变成一个被加强的受控对象。
            }
        }
    }
}
```

- 扩展抽象类 com.cvicse.workflow.examples.ehcache.listener. DestroyWorkItem Listener

XNDestroyWorkItemListener.java:

```
package com.cvicse.workflow.examples.ehcache.listener;

import java.util.List;

import com.cvicse.workflow.api.WfActivityInstance;
import com.cvicse.workflow.api.WfWorkItem;
import com.cvicse.workflow.api.ehcache.
listener.DestroyWorkItemListener;
import com.cvicse.workflow.ehcache.ParallelCache;
import com.cvicse.workflow.examples.ehcache.XNCacheObject;

public class XNDestroyWorkItemListener extends
DestroyWorkItemListener {
    public XNDestroyWorkItemListener() {
    }

    public XNDestroyWorkItemListener(ParallelCache manager) {
        super(manager);
    }

    public void destroyActivityInstance(WfActivityInstance
activityInstance) throws Exception {
        List list = getManager().get("activityInstanceId",
activityInstance.getId());
        for(int i = 0; i < list.size(); i++){
            XNCacheObject ao = (XNCacheObject)list.get(i);
            if(ao != null){
                getManager().remove(ao);
            }
        }
    }

    public void destroyProcess(String procInstId)
throws Exception {
        List list = getManager().get("processInstanceId",
procInstId);
        for(int i = 0; i < list.size(); i++){
            XNCacheObject ao = (XNCacheObject)list.get(i);
            if(ao != null){
                getManager().remove(ao);
            }
```

```
        }
    }

    public void destroyWorkItem(WfWorkItem workItem)
throws Exception {
        XNCacheObject ao =
(XNCacheObject)getManager().get(workItem.getId());
        getManager().remove(ao);
    }

    public void destroyWorkItem(String userId, String state)
 throws Exception{
        //TODO 联合主键查找删除。
    }
}
```

- 扩展抽象类 com.cvicse.workflow.examples.ehcache.listener. UpdateWorkItemListener

XNUpdateWorkItemListener.java:

```
package com.cvicse.workflow.examples.ehcache.listener;

import java.util.List;

import com.cvicse.workflow.api.ehcache.
listener.UpdateWorkItemListener;
import com.cvicse.workflow.ehcache.ParallelCache;
import com.cvicse.workflow.examples.ehcache.XNCacheObject;

public class XNUpdateWorkItemListener extends
UpdateWorkItemListener {
    public XNUpdateWorkItemListener() {
    }

    public XNUpdateWorkItemListener(ParallelCache manager) {
        super(manager);
    }

    public void changeActivityState(String actInstId,
String previousState,String curState, String startTime,
String lastStateChangeTime,String completeTime)
```

```
throws Exception {
        List list = getManager().get("activityInstanceId",
actInstId);
        for(int i = 0; i < list.size(); i++){
            XNCacheObject ao = (XNCacheObject)list.get(i);
            if(ao != null){
            ao.setStartTime(startTime +
"------------时间赋值成功");
            }
        }
    }

    public void changeProcessState(String procInstId,
String previousState,String curState,
String lastStateChangeTime) throws Exception {
        List list = getManager().get("processInstanceId",
procInstId);
        for(int i = 0; i < list.size(); i++){
            XNCacheObject ao = (XNCacheObject)list.get(i);
            if(ao != null){
                ao.setProcessInstanceName(
"------------修改流程状态成功");
            }
        }
    }

    public void changeWorkItemState(String workItemId,
String previousState,String curState,
String receiveTime,
String lastStateChangeTime,String completeTime)
throws Exception {
        XNCacheObject ao =
(XNCacheObject)getManager().get(workItemId);
        if(ao != null){
            ao.setStartTime(receiveTime);
            ao.setCurState(curState);
        }
    }

    public void updateActivityFromNodes(String actInstId,
List fn) throws Exception {
```

```
        List list = getManager().get("activityInstanceId",
actInstId);
        for(int i = 0; i < list.size(); i++){
            XNCacheObject ao = (XNCacheObject)list.get(i);
            if(ao != null){
                ao.setActivityInstanceName("------------修改
fromnodes成功");
            }
        }
    }

    public void updateProcessInstCode(String procInstId,
String instCode) throws Exception{
        List list = getManager().get("processInstanceId",
procInstId);
        for(int i = 0; i < list.size(); i++){
            XNCacheObject ao = (XNCacheObject)list.get(i);
            if(ao != null){
                ao.setProcessInstanceName("------------
修改instCode成功");
            }
        }
    }

    public void updateWorkItemPerformer(String workItemId,
String newParticipantId) throws Exception{
        XNCacheObject ao =
(XNCacheObject)getManager().get(workItemId);
        if(ao != null) ao.setPerformer(newParticipantId);
    }

}
```

◆ 配置 inforflow-cache.xml

inforflow-cache.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="parallelCache"
```

```
        class="com.cvicse.workflow.examples.
ehcache.XNWorkflowCache" init-method="init">
        <constructor-arg type="java.lang.String"
value="processInstanceId,activityInstanceId,performer"/>
        <constructor-arg type="java.lang.String"
value="workItemId"/>
        <property name="cacheManager">
            <ref local="cacheManager" />
        </property>
    </bean>

    <bean id="cacheManager"
          class="org.springframework.cache.
                 ehcache.EhCacheManagerFactoryBean">
        <property name="configLocation">
            <value>classpath:ehcache.xml</value>
        </property>
    </bean>

    <bean id="createWorkItemListener"
         class="com.cvicse.workflow.examples.
                ehcache.listener.XNCreateWorkItemListener"
          singleton="false">
        <property name="manager">
            <ref bean="parallelCache"/>
        </property>
    </bean>

    <bean id="destroyWorkItemListener"
         class="com.cvicse.workflow.examples.
                ehcache.listener.XNDestroyWorkItemListener"
        singleton="false">
        <property name="manager">
            <ref bean="parallelCache"/>
        </property>
    </bean>

    <bean id="updateWorkItemListener"
         class="com.cvicse.workflow.examples.
                ehcache.listener.XNUpdateWorkItemListener"
        singleton="false">
```

```
        <property name="manager">
            <ref bean="parallelCache"/>
        </property>
    </bean>

    <bean id="CacheObjectsPool"
        class="com.cvicse.workflow.examples.
                ehcache.XNCacheObjectsPool"
        singleton="true" lazy-init="true">
        <constructor-arg type="int" value="1000"/>
        <property name="manager">
            <ref bean="parallelCache"/>
        </property>
    </bean>

    <bean id="InforFlowCacheUtil"
           class="com.cvicse.workflow.ehcache.CacheUtil"
      singleton="true">
        <property name="useCache">
            <value>true</value>
        </property>
        <property name="manager">
            <ref bean="parallelCache"/>
        </property>
        <property name="createWorkItemListener">
            <ref bean="createWorkItemListener"/>
        </property>
        <property name="destroyWorkItemListener">
            <ref bean="destroyWorkItemListener"/>
        </property>
        <property name="updateWorkItemListener">
            <ref bean="updateWorkItemListener"/>
        </property>
        <property name="useCacheObjectsPool">
            <value>true</value>
        </property>
        <property name="objectsPool">
            <ref bean="CacheObjectsPool"/>
        </property>
    </bean>
</beans>
```

第8章 应用策略

本章介绍常见的工作流应用策略。

学习本章内容后，你可以：

- 使用 IS-Flow 处理应用系统中业务流程的复杂需求。

1. 开发业务流程系统过程中，流程方面需要重点关注哪些过程？

答：流程方面重点关注过程顺序如下：

- 启动工作流引擎服务，详见服务器端。
- 设计流程定义，详见流程建模指南。
- 部署流程定义，详见 IS-Flow 流程部署运行。
- 流程运行框架开发，参考 helloworld 应用或者标准任务表处理器应用，位于%INFORSUITE_HOME%\FlowServer\docs\demo\flow\java\目录下。
- 流程管理监控，详见 IS-Flow 流程管理监控。
- 流程数据分析，详见 IS-Flow 流程数据分析。

2. 流程运行框架开发，需要关注哪些过程？

答：流程运行框架开发重点关注过程顺序如下：

- 获取流程定义列表，详见获取流程定义列表。
- 创建流程实例，详见创建流程实例。
- 设置流程实例初始化参数，详见分配一个属性值给指定流程实例属性。
- 启动流程实例，详见启动指定流程实例。
- 查询待办任务列表，详见 helloworld 应用或者标准任务表处理器应用。

- 工作台。
- 流程控制操作，常见流程控制操作如下：
  - 提交
  - 回退
  - 跳转
  - 放回
  - 终止流程

其中一些过程可以根据实际需要进行合并，常见的合并，例如：创建流程实例、设置流程实例初始化参数、启动流程实例 这三个过程一般合并为一个过程。

重点辅助过程如下（参见 helloworld 应用或者标准任务表处理器应用）：

- 经手已结束任务查询
- 经手未结束任务查询
- 待处理任务查询
- 可追回任务查询
- 可取回代理任务查询
- 被挂起任务查询
- 综合查询

3. 如何保证应用系统与工作流引擎服务事务一致性？

答：如果工作流引擎服务以独立运行模式集成到应用系统中，事务类型建议使用独立事务方式，如果工作流引擎服务以嵌入运行模式集成到应用系统中，事务类型建议使用传递事务或独立事务。

4. 流程运行时，第一个节点比较难以处理，怎么办？

答：在 IS-Flow 中的第一个节点是一个比较特殊的节点：这个节点除了具有其他节点的功能之外，在这个节点上的人可能需要发起流程，它的处理需要一定的技巧。

常用的处理方式是这样的：填写流程发起前必要的业务信息并保存到业务数据库中，流程发起人创建流程实例，为流程实例设置必要的相关数据，启动流程，然后直接接受该流程产生的工作项并提交工作项。

5. 可否动态选择下一个节点的执行人？

答：IS-Flow 具有灵活强大的任务分配功能，依据应用需求以及灵活度的不同，IS-Flow 具有以下五种解决方案。针对该问题，可参考以下方案中的第二种、第三种和第四种方案。

下面四种方案中将提到“用户接口实现”，该概念是指工作流定义了一个组织机构接口，由用户应用来实现，该接口实现提供了工作流引擎从应用数据库中获取用户资源和用户的通道。

第一种，任务分配可定义为固定的执行人或角色。

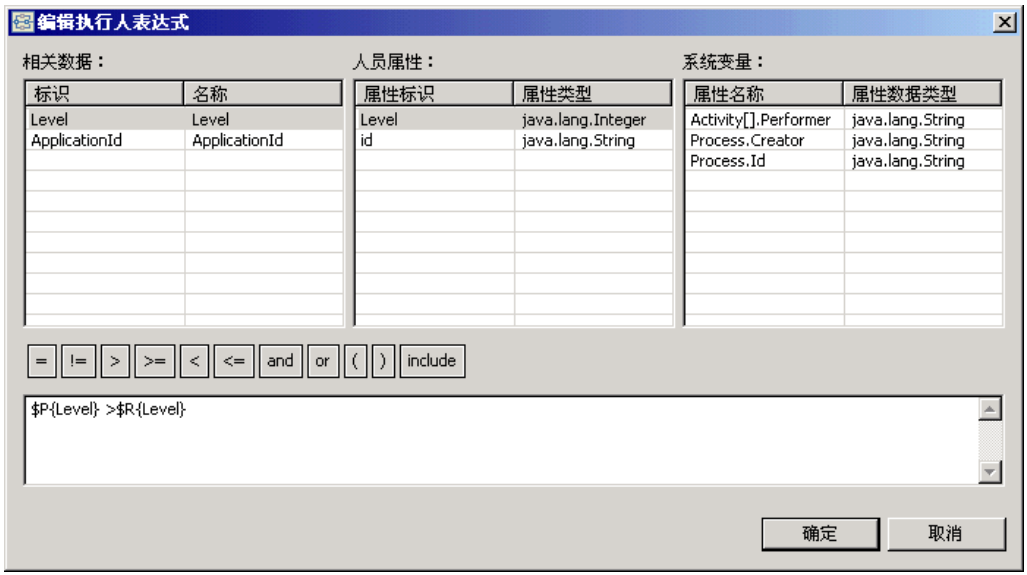
这是最常用的任务分配方式，用户在流程定义时只需定义节点的执行人或角色 ID，引擎运行时根据用户接口实现分配工作项。

第二种，任务分配可定义为动态的执行人或角色。

这种方式是通过动态的为流程实例的相关数据赋值，完全由应用在流程运行过程中动态指定活动（已定义动态执行人或角色的活动）的执行人或角色。

第三种，任务分配具有一定的规则，执行人可定义出与任务分配相关的属性。

针对这种情形，工作流可将节点的任务分配定义为执行人表达式，见图表 8-1 编辑执行人表达式：



图表 8-1 编辑执行人表达式

其中相关数据部分代表要输入的实际业务值，人员属性部分代表与任务分配相关的执行人的扩展属性，将流程实例运行的实际业务值和人员扩展属性做表达式比较即可确定一个人或者一组人。

应用负责定义人员的扩展属性，扩展属性应是和任务分配相关的概念，比如“人员级别”，“业务领域”等，然后为每一个具体的流程实例的相关数据赋值，由工作流负责解析表达式，并实现任务分配。赋值的方式可以是人工选取或者程序自动执行。

以“人员级别超过某个值才有处理该工作的权限”为例，可以将表达式定义为”$P{Level}>$R{Level}”，应用首先以接口实现的方式将每个执行人的级别属性$P{Level}配置到工作流引擎中，若流程 A 的某个活动实例需要级别大于 5 的执行人处理，则通过工作流的接口设置相关数据$R{Level}的值为 5，工作流在创建活动实例时根据实际数据自动解析定义中的表达式并分配工作项。

使用这种任务分配方式，应用也可以通过工作流接口“预览”根据表达式规则所界定的所有可能的执行人。

第四种，任务分配具有一定的规则，但执行人很难或无法定义出与任务分配相关的属性。这种情况下，人员与任务分配的关系可能会根据每一个流程实例的不同而不同，如矩阵式组织模型中的任务分配。

针对这种任务分配方式，工作流提供了动态用户接口。动态用户接口的概念继承自上面提到的用户接口实现，但接口参数中增加了流程运行时的数据，如工作流对象的实例 ID，定义 ID 等。

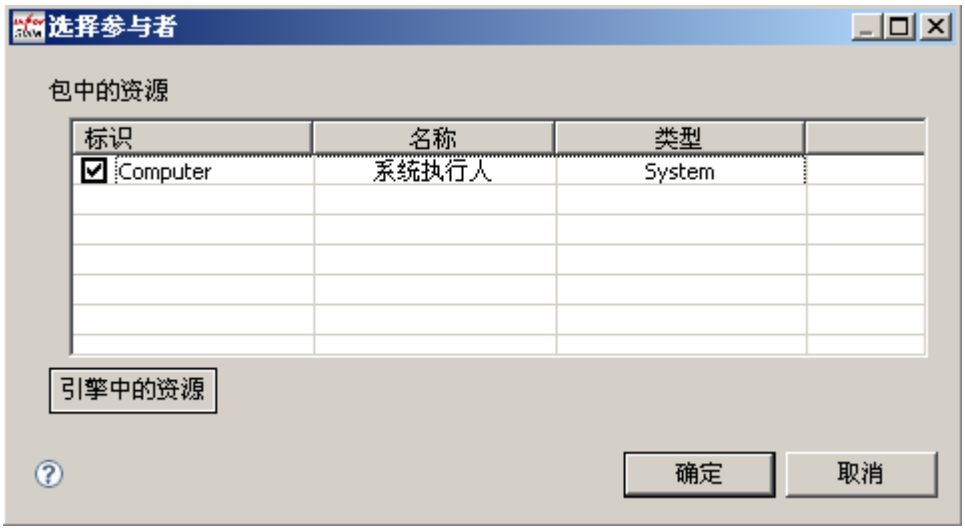
应用通过动态用户接口实现可获取流程运行中的数据，进而根据规则筛选出执行人，工作流根据此接口实现返回的数据创建工作项。

第五种，任务分配没有规则，甚至大部分节点定义的任务分配情形都不同。

工作流公开了任务分配接口，用户可以实现该接口，配置到节点定义中，工作流引擎实例化该节点时，将使用该接口实现替换默认的任务分配实现，通过该接口实现返回的执行人数据创建工作项。这种方案最为灵活，但也最为复杂，需要为单独节点定义任务分配实现。

6. IS-Flow 如何处理自动执行的活动？

答：IS-Flow 的节点类型之一为自动节点，其任务分配的执行人为“系统执行人”，见图表 8-2 设置系统执行人：



图表 8-2 设置系统执行人

自动节点一旦被初始化，将自动执行定义在节点上的业务单元，进行自动的业务处理，并触发下一个节点的实例化。

7. 如何建立工作流的索引？

答：应用可以根据构造了什么查询条件，最终向工作流表产生了什么 SQL，根据数据库分析工具进行索引优化。

其中工作流查询实体和数据库字段之间的对应关系如下：

表格 8-1 工作流查询实体与数据库字段关系表

Entity 查询字段	数据库表		字段
RunningEntity.ProcessId	WR_Process		id
RunningEntity.ProcessInstcode			instCode
RunningEntity.ProcessModelId			procModelId
RunningEntity.ProcessCreateTime			createTime
RunningEntity.ProcessCreator			creator
RunningEntity.ProcessCurState			curState
RunningEntity.ProcessLastStateChangeTime			lastStateChang
RunningEntity.ProcessPriority			priority

RunningEntity.AssignMode			assignMode
RunningEntity.ProcessDefId			procDefId
RunningEntity.ProcessName			name
RunningEntity.ProcessDescription			description
RunningEntity.ProcessContextName	WR_Process_Context		dataFieldId
RunningEntity.ProcessContextValue			value
RunningEntity.ActivityDefId	WR_Activity		actDefId
RunningEntity.ActivityId			id
RunningEntity.ActivityProcessId			procInstId
RunningEntity.ActivityBlockActInstId			blockActInstId
RunningEntity.ActivityInstcode			instCode
RunningEntity.CreateNum			createNum
RunningEntity.FromActivity			fromNodes
RunningEntity.ActivityCreateTime			createTime
RunningEntity.ActivityPerformer			performer
RunningEntity.ActivityCurState			curState
RunningEntity.ActivityLastStateChangeTime			lastStateChang
RunningEntity.ActivityPriority			priority
RunningEntity.IsBack			isBack
RunningEntity.ActivityName			name
RunningEntity.ActivityDescription			description
RunningEntity.ActivityType			activityType
RunningEntity.ActorType			actorType
RunningEntity.ActivityActor			actor
RunningEntity.ActivityDuration			duration
RunningEntity.ActivityLevel			activityLimit
RunningEntity.WorkerAssignExpression			expression
RunningEntity.ActivityContextName	WR_Activity_Context		dataFieldId
RunningEntity.ActivityContextValue			value
RunningEntity.WorkItemProcessId	WR_WorkItem		procInstId
RunningEntity.WorkItemInstcode			instCode
RunningEntity.WorkItemActivityId			actInstId
RunningEntity.WorkItemId			id
RunningEntity.WorkItemPerformer			participant
RunningEntity.WorkItemReceiveTime			receiveTime
RunningEntity.WorkItemLastStateChangeTime			lastStateChang
RunningEntity.WorkItemCompleteTime			completeTime
RunningEntity.WorkItemPreviousState			previousState
RunningEntity.WorkItemCurrentState			curState
RunningEntity.WorkItemPriority			priority
RunningEntity.WorkItemName			name
RunningEntity.WorkItemProcessModelId			procModelId
RunningEntity.WorkItemCreateTime			createTime

RunningEntity.WorkItemContextName	WR_WorkItem_Context		dataFieldId
RunningEntity.WorkItemContextValue			Value
HistoryEntity.ProcessId	WH_Process		id
HistoryEntity.ProcessInstcode			instCode
HistoryEntity.ProcessModelId			procModelId
HistoryEntity.ProcessCreateTime			createTime
HistoryEntity.ProcessCreator			creator
HistoryEntity.ProcessPriority			priority
HistoryEntity.AssignMode			assignMode
HistoryEntity.ProcessDefId			procDefId
HistoryEntity.ProcessName			name
HistoryEntity.ProcessDescription			description
HistoryEntity.ProcessContextName	WH_Process_Context		dataFieldId
HistoryEntity.ProcessContextValue			value
HistoryEntity.ActivityDefId	WH_Activity		actDefId
HistoryEntity.ActivityId			id
HistoryEntity.ActivityProcessId			procInstId
HistoryEntity.ActivityBlockActInstId			blockActInstId
HistoryEntity.ActivityInstcode			instCode
HistoryEntity.CreateNum			createNum
HistoryEntity.FromActivity			fromNodes
HistoryEntity.ActivityCreateTime			createTime
HistoryEntity.ActivityPerformer			performer
HistoryEntity.ActivityCurState			curState
HistoryEntity.ActivityPriority			priority
HistoryEntity.IsBack			isBack
HistoryEntity.ActivityName			name
HistoryEntity.ActivityDescription			description
HistoryEntity.ActivityType			activityType
HistoryEntity.ActorType			actorType
HistoryEntity.ActivityActor			actor
HistoryEntity.ActivityDuration			duration
HistoryEntity.ActivityLevel			activityLimit
HistoryEntity.WorkerAssignExpression			expression
HistoryEntity.ActivityContextName	WH_Activity_Context		dataFieldId
HistoryEntity.ActivityContextValue			value
HistoryEntity.WorkItemProcessId	WH_WorkItem		procInstId
HistoryEntity.WorkItemInstcode			instCode
HistoryEntity.WorkItemActivityId			actInstId
HistoryEntity.WorkItemId			id
HistoryEntity.WorkItemPerformer			participant
HistoryEntity.WorkItemReceiveTime			receiveTime
HistoryEntity.WorkItemCompleteTime			completeTime

HistoryEntity.WorkItemPreviousState			previousState
HistoryEntity.WorkItemCurrentState			curState
HistoryEntity.WorkItemPriority			priority
HistoryEntity.WorkItemName			name
HistoryEntity.WorkItemProcessModelId			procModelId
HistoryEntity.WorkItemCreateTime			createTime
HistoryEntity.WorkItemContextName	WH_WorkItem_Context		dataFieldId
HistoryEntity.WorkItemContextValue			value

第9章 附录

9.1. IS-Flow 常量表

- 流程定义状态表：

表格 9-1 流程定义状态表

状态描述符	值类型	说明
UNDER_REVISION	char	已保存，未发布
RELEASED	char	已保存，已发布
UNDER_TEST	char	正在修改，未发布

- 流程状态类型表：

表格 9-2 流程状态类型表

状态描述符	值类型	值	说明
Global.ABORTED	String	"closed.aborted"	工作流对象实例被取消
Global.COMPLETED	String	"closed.completed"	工作流对象实例正常结束
Global.TERMINATED	String	"closed.terminated"	工作流对象实例已终止
Global.RUNNING	String	"open.Running"	工作流对象实例正在处理
Global.SUSPENDED	String	"open.not_running.suspended"	工作流对象实例已挂起

Global.NOT_STARTED	String	"open.not_running.not_started"	工作流对象实例尚未启动(包括工作项未被接收)

◆ 活动类型描述：

表格 9-3 活动类型表

类型描述符	值类型	值	说明
Global.ACTIVITY_TYPE_SUBFLOW	String	"0"	子流程活动
Global.ACTIVITY_TYPE_NO	String	"1"	无实现活动
Global.ACTIVITY_TYPE_ROUTE	String	"2"	路由活动
Global.ACTIVITY_TYPE_TOOL	String	"3"	工具活动
Global.ACTIVITY_TYPE_BLOCK	String	"4"	块活动

◆ 定义对象描述：

表格 9-4 定义对象描述

类型描述符	值类型	值	说明
Global.IN	String	"In"	输入
Global.OUT	String	"Out"	输出
Global.INOUT	String	"In and out"	输入并且输出
Global.ASSIGNPOLICY_ONE	String	"one"	活动分配策略为一个人
Global.ASSIGNPOLICY_ALL	String	"all"	活动分配策略为所有人

Global.STARTMODE_AUTOMATIC	String	"Automatic"	活动的启动方式为自动
Global.STARTMODE_MANUAL	String	"Manual"	活动的启动方式为手动

◆ 资源描述：

表格 9-5 资源描述

类型描述符	值类型	值	说明
Global.RESOURCE_ROLE	String	"0"	资源为角色
Global.RESOURCE_MAN	String	"1"	资源为单一执行人
Global.RESOURCE_EXPRESION	String	"2"	资源为表达式
Global.RESOURCE_ROLE_EXPRESION	String	"3"	资源为同时满足角色和表达式的所有执行人
Global.RESOURCE_COMPUTER	String	"4"	资源为 System

◆ 事务类型描述：

表格 9-6 事务类型表

类型描述符	值类型	值	说明
Global.NO_TX	char	‘0’	无事务类型
Global.TRAN_TX	char	‘1’	传递连接事务类型
Global.SEP_TX	char	‘2’	独立事务类型
Global.JTA_TX	char	‘3’	JTA 事务类型

◆ TOOL 定义对象描述：

表格 9-7 TOOL 定义对象描述

类型描述符	值类型	值	说明
Global.TOOL_INDEX	String	"Index "	应用程序在活动的定义的顺序
Global. TOOL_PRECONDITION	String	"PreCondition "	应用程序在活动中执行的前提条件
Global. TOOL_STARTEFFECT	String	"StartEffect"	应用程序启动对其它应用程序的影响
Global. TOOL_COMMITEFFECT	String	"CommitEffect"	应用程序提交对其它应用程序的影响
Global.TOOL_MUSTBEDONE	String	"MustBeDone"	应用程序在活动中是否必做
Global.TOOL_DISPOSEDTIME	String	"DisposedTime"	应用程序定义的处理次数
Global.TOOL_URL	String	"url"	应用程序定义的URL

◆ 流程定义模板类型和状态描述：

表格 9-8 流程定义模板类型和状态描述

属性名	类型描述符	值类型	值	说明
ModelType	Global. DEF_OBJECTCONENT	String	"0"	流程定义模板中存储的格式是xpdl 文本内容和 definition 对象
	Global. DEF_FILECONTENT	String	"1"	流程定义模板中存储的格式是definition 对象
	Global. DEF_RESOURCECONTENT	String	"2"	流程定义模板中存储的格式是流程依赖项
State	Global. PROCMODEL_IMPORTED	int	0	流程定义模板的状态参数：已导入
	Global. PROCMODEL_SINGLE_REFERANCED	int	1	流程定义模板的状态参数：已被单引用

	Global. PROCMODEL_MULTI_REFERANCED	int	2	流程定义模板的状态参数：已被多引用
	Global. PROCMODEL_SINGEL_INSTANCED	int	3	流程定义模板的状态参数：已被单实例化
	Global. PROCMODEL_MULTI_INSTANCED	int	4	流程定义模板的状态参数：已被多实例化
	Global. PROCMODEL_EDITED	int	5	流程定义模板的状态参数：已被修改

9.2. Hibernate 数据库配置常量表

在 inforflow.xml 配置文件配置 Hibernate 所必须的 SessionFactory 资源处，需要配置 hibernate 针对数据库的各种参数，典型参数如下表格所示，更详细的配置信息可以参照 Hibernate Reference：

http://www.hibernate.org/hib_docs/reference/zh-cn/html/session-configuration.html#configuration-optional-dialects

表格 9-9 数据库配置常量表

数据库类型	参数名	参数值示例
HSQL	hibernate.dialect	org.hibernate.dialect.HSQLDialect
	hibernate.connection.url	jdbc:hsqldb:hsql://localhost 或 jdbc:hsqldb:test
	hibernate.connection.driver_class	org.hsqldb.jdbcDriver
MySQL	hibernate.dialect	org.hibernate.dialect.MySQLInnoDBDialect org.hibernate.dialect.MySQLMyISAMDialect

		org.hibernate.dialect.MySQLDialect 注释：Inno 和 MyISAM 是两种 Mysql 的表存储格式，IS-Flow 的主键生成策略不适用于 Inno 格式的存储格式。 具体参照： http://dev.mysql.com/doc/refman/5.0/en/innodb-auto-increment-column.html 一般需要配置成 org.hibernate.dialect.MySQLDialect 参数即可。
	hibernate.connection.url	jdbc:mysql:///test
	hibernate.connection.driver_class	com.mysql.jdbc.Driver
	注意：需设置 MySQL 数据库的默认编码格式为 GBK，否则，根据数据库的版本和用户默认编码格式的不同，可能会有中文乱码问题。	
Oracle	hibernate.dialect	org.hibernate.dialect.Oracle9Dialect 或 org.hibernate.dialect.OracleDialect
	hibernate.connection.url	jdbc:oracle:oci8:@flow
	hibernate.connection.driver_class	oracle.jdbc.driver.OracleDriver
PostgreSQL	hibernate.dialect	org.hibernate.dialect.PostgreSQLDialect
	hibernate.connection.url	jdbc:postgresql:template1
	hibernate.connection.driver_class	org.postgresql.Driver
DB2	hibernate.dialect	org.hibernate.dialect.DB2Dialect

	hibernate.connection.url	jdbc:db2:test
	hibernate.connection.driver_class	COM.ibm.db2.jdbc.app.DB2Driver
Sybase	hibernate.dialect	org.hibernate.dialect.SybaseDialect
	hibernate.connection.url	jdbc:sybase:Tds:co3061835-a:5000/tempdb
	hibernate.connection.driver_class	com.sybase.jdbc2.jdbc.SybDriver
MS SQL Server	hibernate.dialect	org.hibernate.dialect.SQLServerDialect
	hibernate.connection.url	jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=inforflow
	hibernate.connection.driver_class	com.microsoft.jdbc.sqlserver.SQLServerDriver

9.3. IS-Flow 配置文件

本节介绍 IS-Flow 使用的配置文件，包括：

- inforflow.xml，配置工作流引擎服务。
- inforflow-client.xml，配置工作流客户端连接与工作流引擎服务的连接。
- inforflow-Manager.xml，配置流程定义导入导出工具。
- inforflow-image-client.xml，配置工作流图片客户端与图片服务的连接。
- inforflow-cache-client.xml，配置缓存客户端与缓存服务的连接。
- calendar.xml，配置工作流日志功能。
- quartz.properties，配置工作流调度功能。
- javagroup.properties，配置工作流集群功能。
- lockTableManagerInit.properties，配置流程锁管理工具。

9.3.1. inforflow.xml

这是 IS-Flow 工作流系统的核心配置文件，其作用是设置工作流引擎服务器的参数信息。其文件的内容（部分）见表格 9-10 inforflow.xml 片段。

表格 9-10 inforflow.xml 片段

```
<inforflow iconsRepository="icons" isAuth="true"
    isAudit="true" hostname="" servicePort="5501"
    serviceName="WorkFlowService" clustering="false"
    supportScheduler="true" modelInitType="lazy">
  <engine name="Engine"
    class="com.cvicse.workflow.core.WorkflowEngine"
    depend="CoreEngine">
    <engineProperties>
      <prop key="connectValidate">true</prop>
    </engineProperties>
  </engine>
  <engine name="AuthenticationEngine"
    class="com.cvicse.workflow.authentication.
    AuthenticationEngine"
    depend="CoreEngine">
    <engineProperties>
      <prop key="file">conf/authentication.xml</prop>
    </engineProperties>
  </engine>
  <engine name="CoreEngine"
          class="com.cvicse.workflow.core.CoreEngineAdapter"/>
  <storage name="MemoryStorage"
class="com.cvicse.workflow.datastore.memory.
WorkflowEntityMemoryManager"/>
  <storage name="DBStorage"
class="com.cvicse.workflow.datastore.db.
WorkflowEntityPersistentManager"/>
  <transaction defaultTxType="NoTxWorkflowContext">
    <dataBaseProperties>
      <prop key="driverClassName">
        oracle.jdbc.driver.OracleDriver</prop>
      <prop key="url">
        jdbc:oracle:thin:@127.0.0.1:1521:flow</prop>
      <prop key="username">system</prop>
```

```
    <prop key="password">system</prop>
    </dataBaseProperties>
```

下面我们来分析一下文件每一部分的内容，首先看

```
<inforflow iconsRepository="icons" isAuth="true" isAudit="true"
  hostname="" servicePort="5501" serviceName="WorkFlowService"
  clustering="false"                    supportScheduler="true"
  modelInitType="lazy">
```

iconsRepository 设置导出图片时需要图标目录的相对路径或绝对路径，如果使用相对路径，并且启动 Java Application 时必须设置系统属性 INFORFLOW_HOME，通过-DINFORFLOW_HOME 指明 IS-Flow 应用的根目录，同时图标目录要在 INFORFLOW_HOME 下，此图标目录下需包括开始节点图标、结束节点图标、块活动节点图标、子流程活动节点图标、工具活动节点图标、路由活动节点图标，可使用安装目录%INFORSUITE_HOME% \repository\com\cvicse\inforflow\icons 下的图标作为默认值（*注意：默认图标名称不能变更*），当然可以通过扩展 com.cvicse.workflow.api.image.WfMonitorImageConfiguration 类来为指定活动实例指定图标目录下的图标。

isAuth 设置工作流引擎是否进行授权，若关闭请在 isAuth 处填写"false"，servicePort 和 serviceName 为实现 RMI 服务形式启动的工作流的配置，clustering 设置工作流引擎是否应用在集群系统中。

supportScheduler 设置工作流引擎是否进行支持调度，若关闭请在 supportScheduler 处填写"false"，此种情况不会初始化调度相关的服务。如果不配置，则默认是 true。

modelInitType 设置工作流引擎启动时加载流程定义模板的方式，目前支持“全量加载”和“智能加载”两种方式。其配置的关键字分别是“all”和“lazy”。全量加载方式，会将所有的有效状态的模板加载到内存；智能加载方式仅加载最新的流程定义模板。如果不配置，则默认是全量加载方式。

```
  <transaction defaultTxType="NoTxWorkflowContext">
```

在 defaultTxType 处定义默认的事务类型，其值的含义为："NoTxWorkflowContext"：不使用事务，适用于对事务要求不严格的项目开发，或者产品测试阶段，一般导入 XPDL 应使用 NoTxWorkflowContext，即工作流内部获取数据

库连接；"SeparateTxWorkflowContext"：应用事务和工作流事务事务分离，工作流使用内部的数据库连接池；"TransConnTxWorkflowContext"：应用向工作流传递数据库连接，由应用管理统一事务；"JTAWorkflowContext"：JTA 事务。

```
<dataBaseProperties>
    <prop key="driverClassName">
          oracle.jdbc.driver.OracleDriver</prop>
    <prop key="url">
          jdbc:oracle:thin:@127.0.0.1:1521:flow</prop>
    <prop key="username">system</prop>
    <prop key="password">system</prop>
 </dataBaseProperties>
```

在使用无事务和独立事务情况下使用，如工作流的演示程序 HelloWorld 即使用此配置，在工作流单元测试阶段中，测试大部分方法应均使用此配置，在此可配置工作流数据库连接数据库驱动，数据库 URL，用户名和密码。

注意：*如果用户所配置数据库为 MySQL 数据库，推荐用户将 MySQL 数据库的默认编码格式设置为 GBK 格式，否则依据 MySQL 版本和用户编码设置的不同，可能会有中文数据信息显示为乱码的现象。如果应用中显示的中文信息为乱码，则用户应该首先设置 MySQL 数据库的默认编码格式设置为 GBK 格式，然后重启 MySQL 服务并重新创建用户所设置的数据库及数据表。*

1. 无事务类型的设置信息，其中 hibernateProperties 属性设置中 hibernate.dialect 设置了使用的数据库类型，hibernate.show_sql 设置了是否在日志中记录 sql 的执行情况，hibernate.connection.useUnicode 设置是否使用 Unicode 编码，hibernate.connection.characterEncoding 设置使用的编码方式，hibernate.hbm2ddl.auto 设置中 create 表示在启动时重新创建数据库结构，update 表示在启动时更新数据库结构。

```
<workflowContext txType="SeparateTxWorkflowContext">
<hibernateProperties>
       <prop key="hibernate.dialect">
org.hibernate.dialect.MySQLDialect</prop>
     <prop key="hibernate.show_sql">false</prop>
</hibernateProperties>
```

```
</workflowContext>
```

2. 独立事务类型的设置信息，hibernateProperties 设置信息的含义类同上面的无事务类型。

```
 <workflowContext txType="TransConnTxWorkflowContext">
<hibernateProperties>
       <prop key="hibernate.dialect">
org.hibernate.dialect.MySQLDialect</prop>
       <prop key="hibernate.show_sql">false</prop>
       </hibernateProperties>
</workflowContext>
```

3. 传递连接类型的设置信息，hibernateProperties 设置信息的含义类同上面的无事务类型。

```
    <workflowContext txType="JTAWorkflowContext">
       <hibernateProperties>
       <prop key="hibernate.dialect">
              org.hibernate.dialect.MySQLDialect</prop>
       <prop key="hibernate.show_sql">false</prop>
       <prop key="hibernate.transaction.factory_class">
              org.hibernate.transaction.JTATransactionFactory</prop>
        <prop key="jta.UserTransaction">
              javax.transaction.UserTransaction</prop>
        <prop key="hibernate.jndi.url">
              t3://localhost:7001</prop>
        <prop key="hibernate.jndi.class">
              weblogic.jndi.WLInitialContextFactory</prop>
        <prop key="hibernate.query.factory_class">
             org.hibernate.hql.ast.ASTQueryTranslatorFactory
        </prop>
       </hibernateProperties>
    </workflowContext>
```

4. JTA 事务型的设置信息，hibernateProperties 设置信息的含义类同上面的无事务

类型。

```
<externalReferenceDataField/>
```

若用户需要使用自定义实体类型的相关数据，就在这里配置 XML 以及实体的包路径，需要注意的是包类的路径需要同时在 classpath 中添加，如果在 classpath 并不存在此类，IS-Flow 启动时会有异常产生。

```
<resourceManager
class="com.cvicse.workflow.examples.resource.AppResourceProvider"/>
```

这一部分是设置资源管理，IS-Flow 工作流系统提供两种默认类型的资源管理方式：从用户数据库中获取资源信息，和应用系统扩展自己所需要的资源管理器，扩展时需实现 com.cvicse.workflow.api.resource.AbstractResourceProvider 如果应用需要配置参数可以在 resourceProperties 指定。

若用户需要设置邮件服务器和短信设备的信息，在这里可以配置。

```
<executionLimit>
        <mailProperties><prop key="isAuth">false</prop>
            <!--邮件服务器是否需要身份验证，若为true，则需要配置用户名
    (authUser)和口令(authPassword)-->
            <prop key="authUser">test</prop>
            <prop key="authPassword">test</prop>
            <prop key="sender">inforflow@cvicse.com</prop>
             <!--发送邮件的邮箱地址-->
            <prop key="host">192.168.2.21</prop>
            <!--发送邮件的邮件服务器地址-->
        </mailProperties>
         <!--短信发送设备与计算机连接的串口号-->
         <mobileProperties>
            <prop key="comName">COM1</prop>
            <!--Windows平台，如"COM1"，"COM2"等-->
            <!--prop key = "comName">/dev/ttyS0</prop-->
            <!--Linux、Unix 等平台，如"/dev/ttyS0"，"/dev/ttyS1"等-->
        </mobileProperties>
 </executionLimit>
```

在上面 xml 中<prop key="host">192.168.2.21</prop>可以指定邮件服务

器的地址；

在<prop key="sender">inforflow@cvicse.com</prop>中指定发件人的邮箱地址；

在<prop key="isAuth">false</prop>中指定发送邮件是否需要服务器验证，该值与邮件服务器的设置有关，如果这里设置为 true，则需要配置

```
<prop key="authUser">test</prop>
<prop key="authPassword">test</prop>
```

的值。

在<prop key="comName">COM1</prop> 中可以设置发送短信的串口。

用户需要引擎进行消息提醒时，需将 JMS 消息中心配置选项赋值为 true；同时如果引擎需要发送消息，发送消息前必须启动消息中心，消息提醒客户端通过消息中心注册并接收消息。

```
<messager default="false">
    <!--是否向已注册客户端消息提醒的用户发送消息-->
    <messagerProperties>
        <prop key="url">localhost</prop>
        <!--消息中心的地址-->
        <prop key="port">61616</prop>
        <!--消息中心公开的端口号-->
        <prop key="isPersistent">true</prop>
        <!--发送的消息是否需要持久化-->
        </messagerProperties>
 </messager>
```

其中，消息中心的地址以及消息中心公开的端口号是依赖于消息服务引擎端的 broker.xml 的配置。

配置日历功能：当在定时启动、执行期限、计算工作流实例对象等操作需要除去非工作时间时，需要在 Inforflow.xml 文件中配置工作日历的管理类路径。代码如下：

```
<calendarManager useCalendar="true"
  class="com.cvicse.workflow.calendar.CalendarManagerImpl">
```

```
        <prop key="useCalendarInStartJob">false</prop>
        <!-- 是否在定时启动的调度中使用日历功能-->
        <prop key="useCalendarInScheduleJob">true</prop>
<!-- 是否在执行期限的调度中使用日历功能 -->
  </calendarManager>
```

"useCaleandar"属性表示是否在工作流引擎中使用日历，取值为"true"或"false"，"class"属性为日历管理类的类路径，此路径可以是 IS-Flow 引擎默认的日历管理实现类，也可以是用户自定义的日历管理实现类。

配置时间管理功能：支持用户配置时间管理类，用于生成系统的时间，便于多个系统之间系统同步。扩展时需实现 com.cvicse.workflow.api.time.TimeManager。代码如下：

```
      <timeManager
class="com.cvicse.workflow.examples.time.AppTimeManager"/>
```

配置监控工作流引擎的管理类：用于管理工作流的配置文件，扩展时需实现 com.cvicse.workflow.api.monitor.FlowMonitorManager。代码如下：

```
      <monitorManager class=
"com.cvicse.workflow.examples.monitor.AppFlowMonitorManager"/>
```

如果在 web 应用中嵌入使用 IS-Flow 工作流系统，需要在 web 应用的 web.xml 中设置 inforflow.xml 的位置，增加如下内容到 web.xml。

```
<servlet>
    <servlet-name>InitServlet</servlet-name>
    <servlet-class>
com.cvicse.workflow.web.InitialServlet</servlet-class>
    <init-param>
      <param-name>SystemConfigFilePath</param-name>
      <param-value>/WEB-INF/inforflow.xml</param-value>
      <description>inforflow config file path and name
</description>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

这是一个典型的 web 应用的</servlet>设置。声明了 IS-Flow 工作流系统应用在 web 应用中时配置文件的存放位置，使用 web.xml 加载 servlet 的方式，在启动带有工作流的应用时，由 WorkflowConfig 初始化 inforflow.xml。

9.3.2. inforflow-client.xml

这是一个标准的 spring beans 设置文件，其作用是设置工作流客户端与引擎服务器的连接方式。其文件的内容见表格 9-11 inforflow-client.xml。

表格 9-11 inforflow-client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="localClient"

     class="com.cvicse.workflow.api.client.local.LocalClient"
     singleton="false"/>

    <bean id="rmiClient"
     class="com.cvicse.workflow.api.client.rmi.WfRMIClient"
destroy-method="close" singleton="false" init-method="init">
        <property name="hostUrl">
            <value>localhost</value>
        </property>
        <property name="hostPort">
            <value>5501</value>
        </property>
        <property name="serviceName">
            <value>InforFlowService</value>
        </property>
    </bean>

    <bean id="wsClient"
         class=
```

```
"com.cvicse.workflow.api.client.webservice.WebServiceClient"
destroy-method="close" singleton="false" init-method="init">
       <property name="serviceAddress">
           <value>
          http://localhost:8080/flowservice/services/WsClient
           </value>
       </property>
   </bean>

   <bean id="InforFlowClient"
         class="com.cvicse.workflow.api.client.WfClientImpl"
    singleton="false">
   <constructor-arg><ref bean="localClient"/>
    </constructor-arg>
   <!--<constructor-arg><ref
   bean="rmiClient"/></constructor-arg> -->
   <!--<constructor-arg><ref
   bean="wsClient"/></constructor-arg> -->
   </bean>
 </beans>
```

通过上面表格，可以看出文件的内容非常简单，下面来分析一下文件的内容，首先看

```
   <bean id="localClient"
class="com.cvicse.workflow.api.client.local.LocalClient"
   singleton="false"/>
```

这一部分声明了一个工作流客户端与引擎服务器的连接采用本地连接实现的 java bean 信息，这部分不需要用户修改。

再继续向下看，

```
   <bean id="rmiClient"
         class="com.cvicse.workflow.api.client.rmi.WfRMIClient"
    destroy-method="close" singleton="false" init-method="init">
       <property name="hostUrl">
           <value>localhost</value>
       </property>
       <property name="hostPort">
           <value>5501</value>
```

```
        </property>
        <property name="serviceName">
            <value>InforFlowService</value>
        </property>
    </bean>
```

这一部分声明了一个工作流客户端与引擎服务器的连接采用RMI连接实现的java bean信息，其中localhost设置了RMI服务器的URL地址（如：192.168.51.53），5501设置了使用的端口号，InforFlowService 设置了RMI服务的名称。用户可根据自己的实际情况修改这3个参数。

再继续向下看，

```
    <bean id="wsClient"
          class=
"com.cvicse.workflow.api.client.webservice.WebServiceClient"
destroy-method="close" singleton="false" init-method="init">
        <property name="serviceAddress">
            <value>
            http://localhost:8080/flowservice/services/WsClient
            </value>
        </property>
    </bean>
```

这一部分声明了以WebService形式提供工作流执行服务的java bean信息，其中http://localhost:8080/flowservice/services/WsClient 设置了WebService服务的地址，其中127.0.0.1设置了WebService服务器的IP，8080设置了使用的端口号，flowservice设置了提供WebService服务的应用名称。用户可根据自己的实际情况修改这个参数。

```
    <bean id="InforFlowClient"
          class="com.cvicse.workflow.api.client.WfClientImpl"
          singleton="false">
        <constructor-arg><ref bean="localClient"/>
        </constructor-arg>
    </bean>
```

最后这一部分设置了用户当前的通讯连接选择使用前面的哪一种连接方式，如上

显示的内容，说明用户选择的连接方式是本地连接，如果使用 RMI 连接的方式就应该修改为如下所示：

```
<bean id="InforFlowClient"
    class="com.cvicse.workflow.api.client.WfClientImpl" singleton="false">
<constructor-arg><ref bean="rmiClient"/>
</constructor-arg>
    </bean>
```

如果使用 WebService 的方式就应该修改为如下所示：

```
<bean id="InforFlowClient"
    class ="com.cvicse.workflow.api.client.WfClientImpl" singleton="false">
<constructor-arg><ref bean="wsClient"/>
</constructor-arg>
    </bean>
```

9.3.3. inforflow-Manager-client.xml

该文件主要用于对流程定义导入导出工具的一些参数进行配置，其结构与 inforflow-client.xml 文件相似，只是比后者多了对 WorkflowUtil 项的配置。与 inforflow-client.xml 文件结构相同的部分请参见 inforflow-client.xml 文件，对 WorkflowUtil 项的配置请详见 3.2.2.4 小节。

9.3.4. inforflow-image-client.xml

该配置文件其作用是设置工作流图片导出客户端与图片服务的连接方式。其文件的内容见表格 9-12 inforflow-image-client.xml。

表格 9-12 inforflow-image-client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>
    <bean id="localImageClient"
       class="com.cvicse.workflow.api.image.LocalImageClient" >
    </bean>
    <bean id="rmiImageClient"

class="com.cvicse.workflow.api.image.WfMonitorImageRMIClient"
destroy-method="close"
       init-method="init" lazy-init="true">
       <property name="hostUrl">
           <value>127.0.0.1</value>
       </property>
       <property name="hostPort">
           <value>5501</value>
       </property>
       <property name="serviceName">
           <value>InforFlowImageProvider</value>
       </property>
    </bean>
    <bean id="InforFlowClient" parent="localImageClient">
    <!--bean id="InforFlowClient" parent="rmiImageClient"-->
    </bean>
  </beans>
```

下面我们来分析一下文件的内容，首先看

```
 <bean id="localImageClient"
        class="com.cvicse.workflow.api.image.LocalImageClient" >
    </bean>
```

这一部分声明了，一个图片导出客户端与图片服务的连接采用本地连接。这部分不需要用户修改。

再继续向下看，

```
 <bean id="rmiImageClient"
  class="com.cvicse.workflow.api.image.WfMonitorImageRMIClient"
      destroy-method="close"
       init-method="init" lazy-init="true">
       <property name="hostUrl">
           <value>127.0.0.1</value>
```

```
        </property>
        <property name="hostPort">
            <value>5501</value>
        </property>
        <property name="serviceName">
            <value>InforFlowImageProvider</value>
        </property>
    </bean>
```

这一部分声明了，一个图片导出客户端与图片服务的连接采用 RMI 连接及实现的 java bean 信息，其中 localhost 设置了 RMI 服务器的 URL 地址（如：192.168.51.198），5501 设置了使用的端口号，用户可根据自己的实际情况修改这 2 个参数；InforFlowImageProvider 设置了 RMI 服务的名称，InforFlowImageProvider (RMI 服务名称)是已经在引擎中注册过的服务名称，一般用户不需要修改。

再继续向下看，

```
    <bean id="InforFlowClient" parent="localImageClient">
    </bean>
```

最后这一部分设置了用户当前的通讯连接选择使用前面的哪一种连接方式。如上显示的内容，说明用户选择的连接方式是本地连接，如果使用 RMI 连接的方式就应该修改为如下所示：

```
    <bean id="InforFlowClient" parent="rmiImageClient">
    </bean>
```

9.3.5. inforflow-cache-client.xml

这是一个标准的 spring beans 设置文件，其作用是设置工作流缓存客户端与引擎服务器的连接方式。其文件的内容如表格 9-13 inforflow-cache-client.xml：

表格 9-13 inforflow-cache-client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
```

```
    <bean id="localCacheClient"
       class="com.cvicse.workflow.api.ehcache.LocalCacheClient"
     singleton="false">
    </bean>

    <bean id="rmiCacheClient"
         class="com.cvicse.workflow.api.ehcache.RMICacheClient"
destroy-method="close" singleton="false" init-method="init">
        <property name="hostUrl">
            <value>localhost</value>
        </property>
        <property name="hostPort">
            <value>5501</value>
        </property>
        <property name="serviceName">
            <value>InforFlowCacheProvider</value>
        </property>
    </bean>

    <bean id="wsCacheClient"
         class="com.cvicse.workflow.api.client.
webservice.WebServiceCacheClient" destroy-method="close"
singleton="false" init-method="init">
        <property name="serviceAddress">
            <value>
        http://localhost:6522/flowservice/services/WsClient
            </value>
        </property>
    </bean>

    <bean id="InforFlowCacheClient" parent="localCacheClient"/>
    <!--bean id="InforFlowCacheClient" parent="rmiCacheClient"/
     -->
    <!--bean id="InforFlowCacheClient" parent="wsCacheClient"/
     -->

</beans>
```

通过上面表格，可以看出文件的内容非常简单，下面来分析一下文件的内容，首先看

```
<bean id="localCacheClient"
   class="com.cvicse.workflow.api.ehcache.LocalCacheClient"
singleton="false">
</bean>
```

这一部分声明了一个工作流缓存客户端与缓存服务的连接采用本地连接实现的 java bean 信息，这部分不许要用户修改。

再继续向下看，

```
<bean id="rmiCacheClient"
      class="com.cvicse.workflow.api.ehcache.RMICacheClient"
 destroy-method="close" singleton="false" init-method="init">
    <property name="hostUrl">
        <value>localhost</value>
    </property>
    <property name="hostPort">
        <value>5501</value>
    </property>
    <property name="serviceName">
        <value>InforFlowCacheProvider</value>
    </property>
</bean>
```

这一部分声明了一个工作流缓存客户端与缓存服务的连接采用 RMI 连接实现的 java bean 信息，其中 localhost 设置了 RMI 服务器的 URL 地址（如：192.168.51.53），5501 设置了使用的端口号，InforFlowService 设置了 RMI 服务的名称。用户可根据自己的实际情况修改这 3 个参数。

再继续向下看，

```
<bean id="wsCacheClient"
  class="com.cvicse.workflow.api.client.
webservice.WebServiceCacheClient" destroy-method="close"
singleton="false" init-method="init">
    <property name="serviceAddress">
        <value>
        http://localhost:6522/flowservice/services/WsClient
        </value>
    </property>
```

```
    </bean>
```

这一部分声明了以 WebService 形式提供工作流缓存服务的 java bean 信息，其中 http://localhost:8080/flowservice/services/WsClient 设置了工作流缓存的 WebService 服务地址，其中 127.0.0.1 设置了 WebService 服务的 IP，8080 设置了使用的端口号，flowservice 设置了提供 WebService 服务的应用名称。用户可根据自己的实际情况修改这个参数。

```
<bean id="InforFlowCacheClient" parent="localCacheClient"/>
```

最后这一部分设置了用户当前的通讯连接选择使用前面的哪一种连接方式。如上显示的内容，说明用户选择的连接方式是本地连接，如果使用 RMI 连接的方式就应该修改为如下所示：

```
<bean id="InforFlowCacheClient" parent="rmiCacheClient"/>
```

如果使用 WebService 的方式就应该修改为如下所示：

```
<bean id="InforFlowCacheClient" parent="wsCacheClient"/>
```

9.3.6. calendar.xml

用户可以通过修改 classes/calendar.xml 文件配置符合本公司习惯的日历，也可以通过编码开发自己的日历对象、日历管理类，或者单独开发自定义的日历管理类。该日历功能仅对相对时间和周期时间有效，对绝对时间无效。

在使用日历时，相对时间的定时启动、执行期限会根据日历按照工作时间跨度计算出下一次执行时间。周期定时启动时，如果触发时间点为非工作日时间，则会抛弃此次启动事件。

日工作日历，用于定义每工作日的固定工作时间，定义时的取值范围为：0:00 - 23:59。“splite”节点的个数必须为 1 个或者两个。

```
    <daily-calendar>
```

```
        <splite from="13:00" to="17:30" />
        <splite from="8:30" to="12:00" />
    </daily-calendar>
```

周工作日历,用于定义每周的非工作日，定义时的取值范围为：Monday、Tuesday、Wednesday、Thursday、Friday、Saturday、Sunday。

basecalendar 属性的值只能是“daily-calendar”和“”中的一个，前者表示在日工作日历的基础上定义周日历，即周日历自动包含日工作日历；后者或者其他值则表示该周工作日历是从零开始定义的。

```
    <weekly-calendar basecalendar="daily-calendar">
        <excluds>
            <excluded>Saturday</excluded>
            <excluded>Sunday</excluded>
        </excluds>
    </weekly-calendar>
```

年度日历，用于定义每年的固定节假日，填写格式为："月.日"，如"10.1"代表每年的十月一日，即国庆节。

basecalendar 属性的值只能是“weekly-calendar”、“daily-calendar”或“”中的一个，“weekly-calendar”表示在周工作日历的基础上定义年度日历，即年度日历自动包含周工作日历；“daily-calendar”表示在日工作日历的基础上定义年度日历，即年度日历自动包含日工作日历；“”或者其他值则表示该年度日历是从零开始定义的。“basecalendar”属性可以逐级继承。

```
    <annual-calendar basecalendar="weekly-calendar">
        <!--固定节假日列表-->
        <excluds>
            <excluded>1.1</excluded>
            <excluded>1.2</excluded>
            <excluded>1.3</excluded>
            <excluded>5.1</excluded>
            <excluded>5.2</excluded>
            <excluded>5.3</excluded>
            <excluded>5.4</excluded>
            <excluded>5.5</excluded>
            <excluded>5.6</excluded>
```

```
            <excluded>5.7</excluded>
            <excluded>10.1</excluded>
            <excluded>10.2</excluded>
            <excluded>10.3</excluded>
            <excluded>10.4</excluded>
            <excluded>10.5</excluded>
            <excluded>10.6</excluded>
            <excluded>10.7</excluded>
        </excluds>
```

因为固定节假日而需要在非工作日中加班的日期列表，如国庆节前后的周末等；该部分定义强制包括的部分，不受周日历的影响。

```
        <includs>
            <included>4.29</included>
            <included>4.30</included>
            <included>9.30</included>
            <included>10.8</included>
        </includs>
    </annual-calendar>
```

配置 IS-Flow 中使用哪一个日历，默认情况下，IS-Flow 中应使用年度日历，而年度日历应该基于周日历，即自动包含周日历，进而自动包含日工作日历，所以年度日历的排除规则为年度日历、周日历和日工作日历所排除日期(时间)的总和减去年度日历中所强制包括的非工作日部分。

```
    <calendar>
        <bean id="annual-calendar"
           name="flowClendar" update="true"/>
    </calendar>
```

9.3.7. quartz.properties

Quartz 提供两种基本作业存储类型。第一种类型叫做 RAMJobStore，它利用通常的内存来持久化调度程序信息。这种作业存储类型最容易配置、构造和运行。对许多应用来说，这种作业存储已经足够了。

然而，因为调度程序信息是存储在被分配给 JVM 的内存里面，所以，当应用程

序停止运行时，所有调度信息将被丢失。如果你需要在重新启动之间持久化调度信息，则将需要第二种类型的作业存储。

第二种类型的作业存储实际上提供两种不同的实现，但两种实现一般都称为 JDBC 作业存储。两种 JDBC 作业存储都需要 JDBC 驱动程序和后台数据库来持久化调度程序信息。这两种类型的不同在于你是否想要控制数据库事务或者释放控制给应用服务器例如 BEA's WebLogic 或 Jboss。这两种 JDBC 作业存储是：

- JobStoreTX：当你想要控制事务或工作在非应用服务器环境中是使用；
- JobStoreCMT：当你工作在应用服务器环境中和想要容器控制事务时使用。

目前 IS-Flow 调度处理上的默认配置为 RAMJobStore，利用内存来持久化调度程序信息；

然而，为避免重新启动 IS-Flow 服务后，调度信息丢失，IS-Flow 调度处理上需要使用 JDBC 作业存储，需要调度程序维护调度信息；

鉴于 IS-Flow 既可以独立启动服务，又可以嵌入到各服务器中去，下面举例说明 IS-Flow 使用 JobStoreTX 这种 JDBC 作业存储：

首先，通过安装目录 service\quartzSQL\下的建表脚本，在指定的库中使用对应脚本创建 quartz 的表结构；一般是在重新创建 IS-Flow 表结构的时候，需要同时重新创建 quartz 的表结构；

然后，通过配置 quartz.properties 文件，IS-Flow 引擎的调度信息可以被持久化保存。下面对 quartz.properties 文件需要配置的元素做简单说明。

表格 9-14 quartz.properties

quartz.properties：

```
#============================================================
# Configure Main Scheduler Properties
#============================================================
org.quartz.scheduler.instanceName = TestScheduler
org.quartz.scheduler.instanceId = AUTO
#============================================================
# Configure ThreadPool
#============================================================
```

```
org.quartz.threadPool.class =
    org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 3
org.quartz.threadPool.threadPriority = 5
#============================================================
# Configure JobStore
#============================================================
org.quartz.jobStore.misfireThreshold = 60000
#save the jobStore items in RAM
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
#save the jobStore items in database
org.quartz.jobStore.class =
    org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass =
    org.quartz.impl.jdbcjobstore.DB2v8Delegate
org.quartz.jobStore.useProperties = false
org.quartz.jobStore.dataSource = myDS
org.quartz.jobStore.tablePrefix = QRTZ_
org.quartz.jobStore.isClustered = false
#============================================================
# Configure Datasources for org.quartz.jobStore.dataSource
#============================================================
org.quartz.dataSource.myDS.driver =
    COM.ibm.db2.jdbc.net.DB2Driver
org.quartz.dataSource.myDS.URL = jdbc:db2://127.0.0.1/flow
org.quartz.dataSource.myDS.user = db2admin
#如果数据库密码为空，则必须注释掉下一句密码配置行，否则会有异常发生。
org.quartz.dataSource.myDS.password = db2
org.quartz.dataSource.myDS.maxConnections = 5
```

最后，正常启动引擎服务，即可将 IS-Flow 引擎的调度信息持久化保存到指定的数据库中，由调度程序维护调度信息。

9.3.8. 引擎集群配置

IS-Flow 引擎支持引擎服务器的集群，也支持流程定义的热部署，即流程定义导入或发生其他变动时，引擎会对流程模板自动进行同样的变化，而不用再重新启动引擎服务去被动更新变动后的流程模板。为了实现上述两种功能，您需要在 inforflow.xml

配置文件中，设置 clustering="true"，即设置引擎支持集群。此时，IS-Flow 引擎采用集群配置文件/classes/javagroup.properties 中的默认设置，该默认配置一般可以满足大多数系统的需要。

注意：*一个集群群组内的所有组员都必须配置相同的数据库参数，才能保证集群的正常运行。*

当然，用户也可以在/classes/javagroup.properties 中设置自己的集群配置属性，该文件设置了集群服务的协议堆栈（cache.cluster.properties)和群组名称（cache.cluster.busName）。如果用户对该文件进行了修改，则必须保证集群内的每一个组员都使用相同的配置文件。该配置文件默认内容见表格 9-15 javagroup.properties：

表格 9-15 javagroup.properties

```
cache.cluster.properties=UDP(mcast_addr=224.0.0.35;
                          mcast_port=45566;ip_ttl=32;\

mcast_send_buf_size=150000;mcast_recv_buf_size=80000):\
              PING(timeout=2000;num_initial_members=3):\
              MERGE2(min_interval=5000;
              max_interval=10000):FD_SOCK:\
              VERIFY_SUSPECT(timeout=1500):\
              pbcast.NAKACK(gc_lag=50;
              retransmit_timeout=300,600,1200,2400,4800):\
              UNICAST(timeout=5000):\
              pbcast.STABLE(desired_avg_gossip=20000):\
              FRAG(frag_size=8096;down_thread=false;
              up_thread=false):\
              pbcast.GMS(join_timeout=5000;
              join_retry_timeout=2000;\
              shun=false;print_local_addr=true)
cache.cluster.multicast.ip=231.12.21.132
cache.cluster.busName=flowgroup
```

cache.cluster.properties 属性设置了集群所使用的堆栈协议，这里使用了 UDP 协议（消息的收发）、PING 协议（组员的发现）、MERGE2 协议（群组的分离与合并）、VERIFY_SUSPECT 协议（组员的意外当机检测）、NAKACK 协议、UNICAST 协议、

STABLE 协议、FRAG 协议和 GMS 协议，并对这些协议的参数进行了设置。了解更多的协议，具体请参见 JGroups 的堆栈协议。

cache.cluster.busName 属性设置了集群群组的名称，在同一个局域网内的多个服务器，会根据群组名称的不同而被划分为不同的集群群组，不同的群组之间可以互不干扰。

9.3.9. 流程锁管理工具配置

当使用集群模式运行时，由于工作流执行服务异常终止等原因，可能会导致用于控制对流程实例的并发访问的锁固化在数据库中，不能被及时释放，从而导致流程实例不能被其它用户访问。这时，可以使用流程锁管理工具来清除被固化的锁。管理流程锁工具启动时，首先从 ClassPath 的 lockTableManagerInit.properties 文件中读取数据库连接参数，用命令行启动该文件后，出现命令行提示符：“Lock =>”，在该提示符下输入合法的命令行参数即可。

流程锁的配置文件 classes\lockTableManagerInit.properties 的内容格式见表格 9-16 lockTableManagerInit.properties。

表格 9-16 lockTableManagerInit.properties

```
driver=COM.ibm.db2.jdbc.net.DB2Driver
path=
url=jdbc:db2://192.168.51.15/flow21
user=infordb2
password=infordb2
```

其中 driver 为数据库的驱动参数；path 为数据库的表名称，如果用户按照 IS-Flow 引擎自动创建的表结构运行，该参数可以不设置；url 为数据库的地址；user 和 password 分别为登录数据库时的用户名和密码。

该工具使用时的具体操作如下所示：(注：①命令参数不区分大小写；②[]内表示可选参数。)

1. 数据查询

(1) list [all]：查询数据表中的全部记录；

(2) list process_instance_id：列出 process_instance_id 所指定的进程编号的流程锁数据。

2. 数据删除

(1) del [all]：删除数据表中的全部记录，

(2) del process_instance_id：删除 process_instance_id 所指定的进程编号的流程锁数据。

3. 查看帮助

? 显示命令行操作的帮助信息。

9.3.10. broker.xml

当配置引擎进行消息提醒时，必须首先启动消息服务引擎，供 IS-Flow 引擎产生消息；消息服务引擎启动所依赖的配置文件为 broker.xml，下面详细介绍下 broker.xml 各属性。

表格 9-17 broker.xml

```
<broker>
  <connector>
    <tcpServerTransport uri="tcp://localhost:61616"
                backlog="1000" useAsyncSend="true"
                maxOutstandingMessages="50"/>
  </connector>
  <persistence>
  <journalPersistence directory="../derby/var/journal">
    <!-- you can point this to a different datasource -->
          <jdbcPersistence dataSourceRef="derby-ds"/>
  </journalPersistence>
  </persistence>
</broker>
```

其中需要关心的属性为 uri，这个参数指定了消息中心的地址和消息中心公开的端口，在配置 inforflow.xml 时需要根据这个 uri 参数进行配置；消息日志持久化目录属性，建议保持原配置，以便保证清晰的目录结构，如上配置默认保存在 service 的

derby/var/journal 目录下；至于持久化所使用的数据源，例如"derby-ds"需要在 broker.xml 中进行声明，下面介绍如何声明数据源"derby-ds"。

如上，IS-Flow 提供了默认的数据源配置；IS-Flow 默认使用 derby 数据库的内嵌驱动方式，这样就不需要单独启动 derby 数据库的服务，方便使用。

broker.xml：

```
<bean id="derby-ds"
class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName">
      <value>org.apache.derby.jdbc.EmbeddedDriver</value>
    </property>
    <property name="url">
  <value>
jdbc:derby:../derby/target/data/derbydb;create=true
</value>
    </property>
    <property name="username">
      <value></value>
    </property>
    <property name="password">
      <value></value>
    </property>
    <property name="poolPreparedStatements">
      <value>true</value>
    </property>
  </bean>
```

9.3.11. message.xml

当配置引擎进行消息提醒时，必须首先启动消息服务引擎，供 IS-Flow 引擎产生消息；同时客户端如果想接收到消息，则必须启动消息提醒客户端工具，消息提醒客户端工具的所依赖的配置文件为%INFORSUITE_HOME%\FlowServer\infor\flow\messenger\data 目录下的 message.xml 文件，下面详细介绍下 message.xml 各属性。

表格 9-18 message.xml

```
<?xml version="1.0" encoding="GBK"?>
<Messenger>
  <config>
    <user>
      <prop key="name">User_1</prop>
      <prop
key="password">Jo=79-t&gt;/r\i-D3ycQH):DJ"U_KT!I5#@@</prop>
      <prop key="autoLogin">false</prop>
      <prop key="receiveMessage">true</prop>
    </user>
    <jms>
       <prop key="url">localhost</prop>
      <prop key="port">61616</prop>
    </jms>
    <sound>
      <prop key="selectSound">true</prop>
      <prop key="filePath">E:\trippygaia1.mid</prop>
    </sound>
  </config>
  <contents>
    <content>
    </content>
  </contents>
</Messenger>
```

其中各属性介绍如下：

参数	介绍	举例
name	工作流引擎登录用户名	User_1
password	工作流引擎登录密码	“”
autoLogin	是否自动登录	false
receiveMessage	是否接收消息	true
url	消息服务器地址	192.168.51.139
port	消息服务器端口	61616

selectSound	是否打开声音	true
filePath	所选择的音乐名称的地址	E:\trippygaia1.mid

9.4. IS-Flow 产品注册相关错误码

下面列出 IS-Flow 运行时，出现的与产品注册相关的常见现象及相应解决方法。

错误码	现象及原因描述	解决办法
12000	版权控制文件不存在	拷贝安装目录下 license.info 到类路径下
12001	License 模块不存在,使用超过许可范围	重新申请并激活授权码
12003	读取版权文件控制项属性时异常，属性值被修改或为空	重新申请并激活授权码。
12004	产品使用超过许可期限	重新申请并激活授权码
12005	cpus 控制项属性值被修改	重新申请并激活授权码
12006	serial 控制项属性值被修改	重新申请并激活授权码
12007	units 控制项属性值被修改	重新申请并激活授权码
12008	版权控制文件读写操作异常	重新申请并激活授权码
12009	ip 控制项属性值被修改	重新申请并激活授权码
12012	在 license 文件中查找匹配的 license 出错	重新申请并激活授权码
12013	licensee 控制项属性值被修改	重新申请并激活授权码
12017	签名时出现异常	重新申请并激活授权码
12018	签名使用的密钥异常	重新申请并激活授权码
12019	反签名出现异常，可能是 formal 控制项被修改或者签名被修改	重新申请并激活授权码
12020	时间格式匹配异常	重新申请并激活授权码
12021	有属性值被修改，当前属性值不完整	重新申请并激活授权码

12022	formal 项不是 false 也不是 true，不合法	重新申请并激活授权码
12023	expiration 控制项属性值被修改	重新申请并激活授权码
12031	license 文件操作异常	重新申请并激活授权码
12032	license 文件根节点查找和添加异常	重新申请并激活授权码
12033	向根节点添加节点时异常	重新申请并激活授权码
12034	使用错误的生成 license 的 jar 包	重新申请并激活授权码
12035	安装机标识对不上	重新申请并激活授权码
12036	验证文件时发现验证的是模版,没生成文件	重新申请并激活授权码
12037	生成文件时发现模版不对,具体用什么模版应该具体分析	重新申请并激活授权码

9.5. IS-Flow 产品运行相关错误码

以下是产品运行过程中常见的异常错误码。

如果出现**服务端未处理异常**，如 NullPointException 等；客户端统一封装成 WfUnHandleException，客户端判定所抛异常是否为**服务端未处理异常**的方式，即无法通过所抛异常信息获取到异常错误码。

异常类型	名称	错误码
WfUnHandleException	服务端未处理异常	ERROR:01
ToolAgentGeneralException	工具代理基本异常	ERROR:02
TransactionException	事务异常	ERROR:03
WfApplicationInstancesUncompletedException	在完成活动时，若应用实例尚未完成则抛出此异常	ERROR:04
WfAttributeAssignExcepton	相关数据赋值异常	ERROR:05
WfCannotJumpException	源活动无法跳转到目的活动产生的异常	ERROR:06

WfConnectException	连接失败异常	ERROR:07
WfException	工作流基本异常	ERROR:08
WfExecuteLimitEventException	工作流在处理催办任务时抛出的异常	ERROR:09
WfInitializeException	工作流引擎初始化异常	ERROR:10
WfInvalidActivityDefinitionException	不合法的活动定义	ERROR:11
WfInvalidActivityInstaneException	不合法的活动实例	ERROR:12
WfInvalidApplicationInstanceException	不合法的应用实例	ERROR:13
WfInvalidAttributeException	不合法的属性	ERROR:14
WfInvalidAuthenticationException	无效的鉴别信息	ERROR:15
WfInvalidDefinitionIdException	无效的定义 ID 标志	ERROR:16
WfInvalidFilterException	不合法的过滤器	ERROR:17
WfInvalidObjectException	无效的对象异常的父类	ERROR:18
WfInvalidParameterException	无效的参数异常	ERROR:19
WfInvalidProcessDefinitionException	不合法的流程定义	ERROR:20
WfInvalidProcessInstanceException	不合法的流程属性异常	ERROR:21
WfInvalidWorkItemException	不合法的工作项异常	ERROR:22
WfIoException	原做为无效的流程的形参数输入	ERROR:23
WfJMSException	接受到底层的 JMSException 后，转化为工作流的 WfJMSException 异常	ERROR:24
WfListenerException	基于 InforFlow 的应用在编写事件插件时可抛给工作流的基本异	ERROR:25

	常	
WfNoDataAuthenticationException	没有操作某数据的权限异常	ERROR:26
WfNoFunctionAuthenticationException	没有执行某功能的权限异常	ERROR:27
WfNoInputException	流程或者活动要求的输入信息为空时抛出的异常	ERROR:28
WfNoMoreActivityDefinitionException	工作流无法获得指定标志号的活动定义	ERROR:29
WfNoMoreActivityInstanceException	工作流无法获得指定标志号的活动实例或活动历史	ERROR:30
WfNoMoreApplicationInsatnceException	通过应用实例 ID 无法获取指定的应用实例	ERROR:31
WfNoMoreAttributeException	通过数据域 ID 无法在实例中获取指定的相关数据的值	ERROR:32
WfNoMoreDataException	无数据异常的基类，用于工作流无法获得指定标志号的定义对象或者实例对象时抛出的异常	ERROR:33
WfNoMoreDataFieldException	工作流无法获得指定标志号的数据域(相关数据)定义	ERROR:34
WfNoMoreProcessDefinitionException	工作流无法获得指定标志号的流程定义	ERROR:35
WfNoMoreProcessInstanceException	工作流无法获得指定标志号的流程实例或者流程历史实例	ERROR:36
WfNoMoreWorkItemException	工作流无法获得指定	ERROR:37

	标志号的工作项实例	
WfRMINotSupportException	RMI 客户端不支持该操作	ERROR:38
WfSchedulerException	工作流在处理定时服务时抛出的异常	ERROR:39
WfTransitionNotAllowedException	状态不允许转移	ERROR:40
WorkflowStoreException	数据库底层异常	ERROR:41
WfCannotWithdrawException	无法追回到目标活动实例产生的异常	ERROR:42
WfInvalidConfigFileException	无效的配置文件异常	ERROR:43

9.6. IS-Flow 工作流优势

相对同类工作流产品，InforSuite Flow 产品具有的优势如下：

- **遵循标准，架构领先**
  - 参照 WfMC（工作流管理联盟）标准，遵循 XPDL 规范；
  - 全面的应用整合的支撑能力，基于统一认证服务实现对应用系统的流程集成；
  - 基于轻量级的微内核，柔性的体系架构，提供可插拔的扩展机制，随需应变，快速构建；
  - 开放的工作流平台中间件。
- **功能强大，性能卓越**
  - 支持面向服务的可视化构件组装；
  - 支持多种任务分配策略，支持多组任务分配，以一种更自然的流程表达方式来表达多人同一时间协同完成一项工作的情况；
  - 强大的规则引擎，简化规则的修改，灵活应对业务变化；支持与 LDAP 的良好集成；

- 企业级工作流引擎，高性能、高并发的支持能力，运用多种先进技术和机制，集群、内存驻留管理技术、流程实例数据和流程历史数据的分离、全面的事务管理机制等。

◆ **贴近需求，灵活机动**

- 图形化流程设计、管理及监控，基于 Eclipse 架构，提供标准的可扩展的集成开发环境；
- 更贴近国内工作流应用场景需求，更周到的本地化服务。

9.7. IS-Flow 工作流特点

InforSuite Flow 工作流元模型基于 WfMC 规范实现，是对业务流程所具有的共性的完善和抽象。InforSuite Flow 在对支持复杂业务流程的分层建模、复杂任务分配方式以及应付易变的业务过程方面都具有独到之处，降低了应用系统的开发难度，也减轻了开发人员的工作量。

InforSuite Flow 的产品特点如下：

◆ **遵循国际规范**

- 参照 WfMC（工作流管理联盟）标准，遵循 XPDL 规范
- 遵循 OMG（对象管理组织）规范

◆ **平台无关性**

- 操作系统无关性
- 数据库无关性
- 应用服务器无关性

◆ **流程控制的灵活性**

- 支持动态的分支选择与合并
- 支持流程的动态回退与跳转
- 支持动态任务分配
- 提供灵活、丰富的编程接口
- 支持用户自定义条件的综合查询

◆ **流程建模的可扩展性**

- 支持对流程、节点、工作项的属性进行扩展，以适应业务建模的需求
- 支持对工作流引擎的扩展，以解释流程定义时所扩展的各种业务相关的属性
- 支持调度服务中日历功能扩展，以满足不同行业实际日历需求
- 支持自定义执行期限和事件，以满足应用系统的特殊执行期限和事件要求
- 支持事务、日志、导出图片格式、用户权限校验、待办任务缓存等的扩展