

Activiti 用户手册

v 5.21.0

Table of Contents

1. 简介
 - 1.1. 协议
 - 1.2. 下载
 - 1.3. 源码
 - 1.4. 必要的软件
 - 1.4.1. JDK 6+
 - 1.4.2. Eclipse Indigo 和 Juno
 - 1.5. 反馈问题
 - 1.6. 实验性功能
 - 1.7. 内部实现类
2. 开始
 - 2.1. 一分钟入门
 - 2.2. Activiti安装
 - 2.3. Activiti数据库配置
 - 2.4. 引入Activiti jar与依赖
 - 2.5. 下一步
3. 配置 Configuration
 - 3.1. 创建ProcessEngine Creating a ProcessEngine
 - 3.2. ProcessEngineConfiguration bean
 - 3.3. 数据库配置 Database configuration
 - 3.4. JNDI数据源配置 JNDI Datasource Configuration
 - 3.4.1. 使用 Usage
 - 3.4.2. 配置 Configuration
 - 3.5. 支持的数据库 Supported databases
 - 3.6. 创建数据库表 Creating the database tables
 - 3.7. 数据库表名说明 Database table names explained
 - 3.8. 数据库升级 Database upgrade
 - 3.9. 作业执行器与异步执行器（从5.17.0版本起） Job Executor and Async Executor (since version 5.17.0)
 - 3.10. 启用作业执行器 Job executor activation
 - 3.11. 启用异步执行器 Async executor activation
 - 3.12. 配置邮件服务器 Mail server configuration
 - 3.13. 配置历史 History configuration
 - 3.14. 配置在表达式与脚本中暴露的bean Exposing configuration beans in expressions and scripts
 - 3.15. 配置部署缓存 Deployment cache configuration
 - 3.16. 日志 Logging
 - 3.17. 映射诊断上下文 Mapped Diagnostic Contexts
 - 3.18. 事件处理器 Event handlers
 - 3.18.1. 事件监听器实现 Event listener implementation
 - 3.18.2. 配置与安装 Configuration and setup
 - 3.18.3. 在运行时增加监听器 Adding listeners at runtime
 - 3.18.4. 为流程定义增加监听器 Adding listeners to process definitions
 - 执行用户定义逻辑的监听器 Listeners executing user-defined logic
 - 抛出BPMN事件的监听器 Listeners throwing BPMN events
 - 关于流程定义监听器的说明 Notes on listeners on a process-definition
 - 3.18.5. 通过API分发事件 Dispatching events through API
 - 3.18.6. 支持的事件类型 Supported event types
 - 3.18.7. 附加信息 Additional remarks
4. The Activiti API
 - 4.1. 流程引擎API与服务 The Process Engine API and services
 - 4.2. 异常策略 Exception strategy
 - 4.3. 使用Activiti services(Working with the Activiti services)
 - 4.3.1. 部署流程 Deploying the process
 - 4.3.2. 启动流程实例 Starting a process instance
 - 4.3.3. 完成任务 Completing tasks
 - 4.3.4. 暂停与激活流程 Suspending and activating a process

- 4.3.5. 扩展阅读 Further reading
- 4.4. 查询API (Query API)
- 4.5. 变量 Variables
- 4.6. 表达式 Expressions
- 4.7. 单元测试 Unit testing
- 4.8. Debug单元测试(Debugging unit tests)
- 4.9. Web应用中的流程引擎 The process engine in a web application
- 5. 集成Spring (Spring integration)
 - 5.1. ProcessEngineFactoryBean
 - 5.2. 事务 Transactions
 - 5.3. 表达式 Expressions
 - 5.4. 自动部署资源 Automatic resource deployment
 - 5.5. 单元测试 Unit testing
 - 5.6. 通过Hibernate 4.2.x使用JPA (JPA with Hibernate 4.2.x)
 - 5.7. Spring Boot
 - 5.7.1. 兼容性 Compatibility
 - 5.7.2. 开始 Getting started
 - 5.7.3. 更换数据源与连接池 Changing the database and connection pool
 - 5.7.4. REST支持 (REST support)
 - 5.7.5. JPA支持 (JPA support)
 - 5.7.6. 扩展阅读 Further Reading
- 6. 部署 Deployment
 - 6.1. 业务存档 Business archives
 - 6.1.1. 编程方式部署 Deploying programmatically
 - 6.1.2. 使用Activiti Explorer部署 (Deploying with Activiti Explorer)
 - 6.2. 外部资源 External resources
 - 6.2.1. Java类 Java classes
 - 6.2.2. 在流程中使用Spring bean (Using Spring beans from a process)
 - 6.2.3. 创建单独应用 Creating a single app
 - 6.3. 流程定义的版本 Versioning of process definitions
 - 6.4. 提供流程图 Providing a process diagram
 - 6.5. 生成流程图 Generating a process diagram
 - 6.6. 类别 Category
- 7. BPMN 2.0介绍 BPMN 2.0 Introduction
 - 7.1. BPMN是什么? What is BPMN?
 - 7.2. 定义流程 Defining a process
 - 7.3. 准备: 十分钟教程 Getting started: 10 minute tutorial
 - 7.3.1. 先决条件 Prerequisites
 - 7.3.2. 目标 Goal
 - 7.3.3. 用例 Use case
 - 7.3.4. 流程图 Process diagram
 - 7.3.5. XML表现 XML representation
 - 7.3.6. 启动流程实例 Starting a process instance
 - 7.3.7. 任务列表 Task lists
 - 7.3.8. 申领任务 Claiming the task
 - 7.3.9. 完成任务 Completing the task
 - 7.3.10. 结束流程 Ending the process
 - 7.3.11. 代码总结 Code overview
 - 7.3.12. 继续提高 Future enhancements
- 8. BPMN 2.0 结构 BPMN 2.0 Constructs
 - 8.1. 自定义扩展 Custom extensions
 - 8.2. 事件 Events
 - 8.2.1. 事件定义 Event Definitions
 - 8.2.2. 定时器事件定义 Timer Event Definitions
 - 8.2.3. 错误事件定义 Error Event Definitions
 - 8.2.4. 信号事件定义 Signal Event Definitions
 - 抛出信号事件 Throwing a Signal Event
 - 捕获信号事件 Catching a Signal Event
 - 查询信号事件订阅 Querying for Signal Event subscriptions
 - 信号事件范围 Signal event scope
 - 信号事件示例 Signal Event example(s)
 - 8.2.5. 消息事件定义 Message Event Definitions

抛出消息事件 Throwing a Message Event

查询消息事件订阅 Querying for Message Event subscriptions

消息事件示例 Message Event example(s)

8.2.6. 启动事件 Start Events

8.2.7. 空启动事件 None Start Event

描述 Description

图示 Graphical notation

XML表示 XML representation

空启动事件的自定义扩展 Custom extensions for the none start event

8.2.8. 定时器启动事件 Timer Start Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.9. 消息启动事件 Message Start Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.10. 信号启动事件 Signal Start Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.11. 错误启动事件 Error Start Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.12. 结束事件 End Events

8.2.13. 空结束事件 None End Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.14. 错误结束事件 Error End Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.15. 终止结束事件 Terminate End Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.16. 取消结束事件 Cancel End Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.17. 边界事件 Boundary Events

8.2.18. 定时器边界事件 Timer Boundary Event

描述 Description

图示 Graphical Notation

XML表示 XML Representation

边界事件的已知问题 Known issue with boundary events

8.2.19. 错误边界事件 Error Boundary Event

描述 Description

图示 Graphical notation

XML表示 Xml representation

示例 Example

8.2.20. 信号边界事件 Signal Boundary Event

描述 Description

图示 Graphical notation

XML表示 XML representation

示例 Example

8.2.21. 消息边界事件 Message Boundary Event

描述 Description

图示 Graphical notation

XML表示 XML representation

示例 Example

8.2.22. 取消边界事件 Cancel Boundary Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.23. 补偿边界事件 Compensation Boundary Event

描述 Description

图示 Graphical notation

XML表示 XML representation

8.2.24. 捕获中间事件 Intermediate Catching Events

8.2.25. 定时器捕获中间事件 Timer Intermediate Catching Event

描述 Description

图示 Graphical Notation

XML表示 XML Representation

8.2.26. 信号捕获中间事件 Signal Intermediate Catching Event

描述 Description

图示 Graphical notation

XML表示 XML representation

示例 Example

8.2.27. 消息捕获中间事件 Message Intermediate Catching Event

描述 Description

图示 Graphical notation

XML表示 XML representation

示例 Example

8.2.28. 抛出中间事件 Intermediate Throwing Event

8.2.29. 空抛出中间事件 Intermediate Throwing None Event

8.2.30. 信号抛出中间事件 Signal Intermediate Throwing Event

描述 Description

图示 Graphical notation

XML表示 XML representation

示例 Example

8.2.31. 补偿抛出中间事件 Compensation Intermediate Throwing Event

描述 Description

图示 Graphical notation

Xml representation

8.3. 顺序流 Sequence Flow

8.3.1. 描述 Description

8.3.2. 图示 Graphical notation

8.3.3. XML表示 XML representation

8.3.4. 条件顺序流 Conditional sequence flow

描述 Description

图示 Graphical notation

XML表示 XML representation

8.3.5. 默认顺序流 Default sequence flow

描述 Description

图示 Graphical notation

XML表示 XML representation

8.4. 网关 Gateways

8.4.1. 排他网关 Exclusive Gateway

描述 Description

图示 Graphical notation

XML表示 XML representation

8.4.2. 并行网关 Parallel Gateway

描述 Description

图示 Graphical Notation

XML表示 XML representation

8.4.3. 包容网关 Inclusive Gateway

描述 Description

图示 Graphical Notation

XML表示 XML representation

8.4.4. 基于事件的网关 Event-based Gateway

描述 Description

[图示 Graphical notation](#)

[XML表示 XML representation](#)

[示例 Example\(s\)](#)

8.5. 任务 Tasks

8.5.1. 用户任务 User Task

[描述 Description](#)

[图示 Graphical notation](#)

[XML表示 XML representation](#)

[到期日期 Due Date](#)

[用户指派 User assignment](#)

[用于任务指派的Activiti扩展 Activiti extensions for task assignment](#)

[自定义身份联系类型（试验特性） Custom identity link types \(Experimental\)](#)

[通过任务监听器自定义指派 Custom Assignment via task listeners](#)

8.5.2. 脚本任务 Script Task

[描述 Description](#)

[图示 Graphical Notation](#)

[XML表示 XML representation](#)

[脚本中的变量 Variables in scripts](#)

[脚本结果 Script results](#)

[安全性 Security](#)

8.5.3. Java服务任务 Java Service Task

[描述 Description](#)

[图示 Graphical Notation](#)

[XML表示 XML representation](#)

[实现 Implementation](#)

[字段注入 Field Injection](#)

[字段注入与线程安全 Field injection and thread safety](#)

[服务任务的结果 Service task results](#)

[处理异常 Handling exceptions](#)

[抛出BPMN错误 Throwing BPMN Errors](#)

[异常映射 Exception mapping](#)

[异常顺序流 Exception Sequence Flow](#)

[在JavaDelegate中使用Activiti服务 Using an Activiti service from within a JavaDelegate](#)

8.5.4. Web服务任务 Web Service Task

[描述 Description](#)

[图示 Graphical Notation](#)

[XML表示 XML representation](#)

[Web服务任务IO规范 Web Service Task IO Specification](#)

[Web服务任务数据输入关联 Web Service Task data input associations](#)

[Web服务任务数据输出关联 Web Service Task data output associations](#)

8.5.5. 业务规则任务 Business Rule Task

[描述 Description](#)

[图示 Graphical Notation](#)

[XML表示 XML representation](#)

8.5.6. 邮件任务 Email Task

[邮件服务器配置 Mail server configuration](#)

[定义邮件任务 Defining an Email Task](#)

[使用示例 Example usage](#)

8.5.7. Mule任务 Mule Task

[定义Mule任务 Defining an Mule Task](#)

[使用示例 Example usage](#)

8.5.8. Camel任务 Camel Task

[定义Camel任务 Defining a Camel Task](#)

[简单Camel调用示例 Simple Camel Call example](#)

[连通性测试 Ping Pong example](#)

[返回变量 Returning back the variables](#)

[异步连通性测试 Asynchronous Ping Pong example](#)

[从Camel路由实例化工作流 Instantiate workflow from Camel route](#)

8.5.9. 手动任务 Manual Task

[描述 Description](#)

[图示 Graphical Notation](#)

[XML表示 XML representation](#)

8.5.10. Java接收任务 Java Receive Task

描述 [Description](#)

图示 [Graphical notation](#)

XML表示 [XML representation](#)

8.5.11. Shell任务 Shell Task

描述 [Description](#)

定义Shell任务 [Defining a shell task](#)

使用示例 [Example usage](#)

8.5.12. 执行监听器 Execution listener

[执行监听器上的字段注入 Field injection on execution listeners](#)

8.5.13. 任务监听器 Task listener

8.5.14. 多实例 Multi-instance (for each)

描述 [Description](#)

图示 [Graphical notation](#)

XML表示 [Xml representation](#)

[边界事件与多实例 Boundary events and multi-instance](#)

[多实例与执行监听器 Multi instance and execution listeners](#)

8.5.15. 补偿处理器 Compensation Handlers

描述 [Description](#)

图示 [Graphical notation](#)

XML表示 [XML representation](#)

8.6. 子流程与调用活动 Sub-Processes and Call Activities

8.6.1. 子流程 Sub-Process

描述 [Description](#)

图示 [Graphical Notation](#)

XML表示 [XML representation](#)

8.6.2. 事件子流程 Event Sub-Process

描述 [Description](#)

图示 [Graphical Notation](#)

XML表示 [XML representation](#)

示例 [Example](#)

8.6.3. 事务子流程 Transaction subprocess

描述 [Description](#)

图示 [Graphical Notation](#)

XML表示 [XML representation](#)

示例 [Example](#)

8.6.4. 调用活动（子流程） Call activity (subprocess)

描述 [Description](#)

图示 [Graphical Notation](#)

XML表现 [XML representation](#)

[传递变量 Passing variables](#)

示例 [Example](#)

8.7. 事务与并发 Transactions and Concurrency

8.7.1. 异步延续 Asynchronous Continuations

8.7.2. 失败重试 Fail Retry

8.7.3. 排他作业 Exclusive Jobs

[为什么排他作业？ Why exclusive Jobs?](#)

[什么是排他作业？ What are exclusive jobs?](#)

8.8. 流程启动认证 Process Initiation Authorization

8.9. 数据对象 Data objects

9. 表单 Forms

9.1. 表单参数 Form properties

9.2. 外部表单渲染 External form rendering

10. JPA（Java Persistence API Java持久化API）

10.1. 需求 Requirements

10.2. 配置 Configuration

10.3. 使用 Usage

10.3.1. 简单示例 Simple Example

10.3.2. 查询JPA流程变量 Query JPA process variables

10.3.3. 使用Spring bean与JPA的高级示例 Advanced example using Spring beans and JPA

11. 历史 History

11.1. 查询历史 Querying history

- 11.1.1. 历史流程实例查询 [HistoricProcessInstanceQuery](#)
 - 11.1.2. 历史变量实例查询 [HistoricVariableInstanceQuery](#)
 - 11.1.3. 历史活动实例查询 [HistoricActivityInstanceQuery](#)
 - 11.1.4. 历史详情查询 [HistoricDetailQuery](#)
 - 11.1.5. 历史任务实例查询 [HistoricTaskInstanceQuery](#)
 - 11.2. 历史配置 [History configuration](#)
 - 11.3. 审计目的历史 [History for audit purposes](#)
 - 12. Eclipse Designer
 - 12.1. 安装 [Installation](#)
 - 12.2. Activiti Designer编辑器功能 [Activiti Designer editor features](#)
 - 12.3. Activiti Designer BPMN功能 [Activiti Designer BPMN features](#)
 - 12.4. Activiti Designer部署功能 [Activiti Designer deployment features](#)
 - 12.5. 扩展Activiti Designer ([Extending Activiti Designer](#))
 - 12.5.1. 自定义画板 [Customizing the palette](#)
 - 设置扩展 [Extension setup \(Eclipse/Maven\)](#)
 - 在Activiti Designer中应用你的扩展 [Applying your extension to Activiti Designer](#)
 - 为画板添加图形 [Adding shapes to the palette](#)
 - 配置自定义服务任务的运行时执行 [Configuring runtime execution of Custom Service Tasks](#)
 - 参数类型 [Property types](#)
 - [PropertyType.TEXT](#)
 - [PropertyType.MULTILINE_TEXT](#)
 - [PropertyType.PERIOD](#)
 - [PropertyType.BOOLEAN_CHOICE](#)
 - [PropertyType.RADIO_CHOICE](#)
 - [PropertyType.COMBOBOX_CHOICE](#)
 - [PropertyType.DATE_PICKER](#)
 - [PropertyType.DATA_GRID](#)
 - 在画板中禁用默认图形 [Disabling default shapes in the palette](#)
 - 12.5.2. 验证流程图与输出为自定义格式 [Validating diagrams and exporting to custom output formats](#)
 - 创建ProcessValidator扩展 [Creating a ProcessValidator extension](#)
 - 创建ExportMarshaller扩展 [Creating an ExportMarshaller extension](#)
13. Activiti Explorer
 - 13.1. 配置 [Configuration](#)
 - 13.2. 流程图 [Process diagram](#)
 - 13.3. 任务 [Tasks](#)
 - 13.4. 启动流程实例 [Start process instances](#)
 - 13.5. 我的实例 [My instances](#)
 - 13.6. 管理 [Administration](#)
 - 13.7. 报告 [Reporting](#)
 - 13.7.1. 报告数据JSON ([Report data JSON](#))
 - 13.7.2. 实例流程 [Example process](#)
 - 13.7.3. 报告启动表单 [Report start forms](#)
 - 13.7.4. 示例流程 [Example processes](#)
 - 13.8. 改变数据库 [Changing the database](#)
14. Activiti Modeler
 - 14.1. 编辑模型 [Model editing](#)
 - 14.2. 导入现有模型 [Importing existing models](#)
 - 14.3. 将已部署定义转换为可编辑模型 [Convert deployed definitions to a editable model](#)
 - 14.4. 将模型导出为BPMN XML ([Export model to BPMN XML](#))
 - 14.5. 将模型部署至Activiti引擎 [Deploy model to the Activiti Engine](#)
15. REST API
 - 15.1. Activiti REST一般原则 [General Activiti REST principles](#)
 - 15.1.1. 安装与认证 [Installation and Authentication](#)
 - 15.1.2. 配置 [Configuration](#)
 - 15.1.3. 在Tomcat中使用 [Usage in Tomcat](#)
 - 15.1.4. 方法与返回码 [Methods and return-codes](#)
 - 15.1.5. 错误响应体 [Error response body](#)
 - 15.1.6. 请求参数 [Request parameters](#)
 - URL片段 [URL fragments](#)
 - Rest URL查询参数 [Rest URL query parameters](#)
 - JSON体参数 [JSON body parameters](#)
 - 分页与排序 [Paging and sorting](#)

JSON查询变量格式

变量表示 Variable representation

15.2. 部署 Deployment

- 15.2.1. 部署的列表 List of Deployments
- 15.2.2. 获取一个部署 Get a deployment
- 15.2.3. 创建一个新部署 Create a new deployment
- 15.2.4. 删除一个部署 Delete a deployment
- 15.2.5. 列表一个部署中的资源 List resources in a deployment
- 15.2.6. 获取一个部署资源 Get a deployment resource
- 15.2.7. 获取一个部署资源的内容 Get a deployment resource content

15.3. 流程定义 Process Definitions

- 15.3.1. 流程定义的列表 List of process definitions
- 15.3.2. 获取一个流程定义
- 15.3.3. 更新一个流程定义的分类 Update category for a process definition
- 15.3.4. 获取一个流程定义资源的内容 Get a process definition resource content
- 15.3.5. 获取一个流程定义的BPMN模型 Get a process definition BPMN model
- 15.3.6. 暂停一个流程定义 Suspend a process definition
- 15.3.7. 激活一个流程定义 Activate a process definition
- 15.3.8. 获取一个流程定义的所有候选启动者 Get all candidate starters for a process-definition
- 15.3.9. 为一个流程定义添加一个候选启动者 Add a candidate starter to a process definition
- 15.3.10. 从一个流程定义中删除一个候选启动者 Delete a candidate starter from a process definition
- 15.3.11. 从一个流程定义中获取一个候选启动者 Get a candidate starter from a process definition

15.4. 模型 Models

- 15.4.1. 获取模型的列表 Get a list of models
- 15.4.2. 获取一个模型 Get a model
- 15.4.3. 更新一个模型 Update a model
- 15.4.4. 创建一个模型 Create a model
- 15.4.5. 删除一个模型 Delete a model
- 15.4.6. 获取一个模型的编辑器源码 Get the editor source for a model
- 15.4.7. 设置一个模型的编辑器源码 Set the editor source for a model
- 15.4.8. 获取一个模型的附加编辑器源码 Get the extra editor source for a model
- 15.4.9. 设置一个模型的附加编辑器源码 Set the extra editor source for a model

15.5. 流程实例 Process Instances

- 15.5.1. 获取一个流程实例 Get a process instance
- 15.5.2. 删除一个流程实例 Delete a process instance
- 15.5.3. 激活或暂停一个流程实例 Activate or suspend a process instance
- 15.5.4. 启动一个流程实例 Start a process instance
- 15.5.5. 流程实例的列表 List of process instances
- 15.5.6. 查询流程实例 Query process instances
- 15.5.7. 获取一个流程实例的流程图 Get diagram for a process instance
- 15.5.8. 获取流程实例的参与人 Get involved people for process instance
- 15.5.9. 为一个流程实例添加一个参与用户 Add an involved user to a process instance
- 15.5.10. 从一个流程实例中移除一个参与用户 Remove an involved user to from process instance
- 15.5.11. 一个流程实例的变量的列表 List of variables for a process instance
- 15.5.12. 获取一个流程实例的一个变量 Get a variable for a process instance
- 15.5.13. 为一个流程实例创建（或更新）变量 Create (or update) variables on a process instance
- 15.5.14. 为一个流程实例更新一个变量 Update a single variable on a process instance
- 15.5.15. 为一个流程实例创建一个新的二进制变量 Create a new binary variable on a process-instance
- 15.5.16. 为一个流程实例更新一个已有的二进制变量 Update an existing binary variable on a process-instance

15.6. 执行 Executions

- 15.6.1. 获取一个执行 Get an execution
- 15.6.2. 对一个执行进行操作 Execute an action on an execution
- 15.6.3. 获取一个执行中的激活活动 Get active activities in an execution
- 15.6.4. 执行的列表 List of executions
- 15.6.5. 查询执行 Query executions
- 15.6.6. 一个执行中的变量的列表 List of variables for an execution
- 15.6.7. 获取一个执行的一个变量 Get a variable for an execution
- 15.6.8. 为一个执行创建（或更新）变量 Create (or update) variables on an execution
- 15.6.9. 为一个执行更新一个变量 Update a variable on an execution
- 15.6.10. 为一个执行创建一个新的二进制变量 Create a new binary variable on an execution
- 15.6.11. 为一个执行更新一个已有二进制变量 Update an existing binary variable on a process-instance

15.7. 任务 Tasks

- 15.7.1. 获取一个任务 [Get a task](#)
- 15.7.2. 任务的列表 [List of tasks](#)
- 15.7.3. 查询任务 [Query for tasks](#)
- 15.7.4. 更新一个任务 [Update a task](#)
- 15.7.5. 任务操作 [Task actions](#)
- 15.7.6. 删除一个任务 [Delete a task](#)
- 15.7.7. 获取一个任务的所有变量 [Get all variables for a task](#)
- 15.7.8. 从一个任务中获取一个变量 [Get a variable from a task](#)
- 15.7.9. 获取一个变量的二进制数据 [Get the binary data for a variable](#)
- 15.7.10. 为一个任务创建一个新的变量 [Create new variables on a task](#)
- 15.7.11. 为一个任务创建一个新的二进制变量 [Create a new binary variable on a task](#)
- 15.7.12. 为一个任务更新一个已有变量 [Update an existing variable on a task](#)
- 15.7.13. 为一个任务更新一个二进制变量 [Updating a binary variable on a task](#)
- 15.7.14. 为一个任务删除一个变量 [Delete a variable on a task](#)
- 15.7.15. 为一个任务删除所有本地变量 [Delete all local variables on a task](#)
- 15.7.16. 获取一个任务的所有身份关联 [Get all identity links for a task](#)
- 15.7.17. 获取一个任务的组或用户的所有身份关联 [Get all identitylinks for a task for either groups or users](#)
- 15.7.18. 获取一个任务的一个身份关联 [Get a single identity link on a task](#)
- 15.7.19. 为一个任务创建一个身份关联 [Create an identity link on a task](#)
- 15.7.20. 为一个任务删除一个身份关联 [Delete an identity link on a task](#)
- 15.7.21. 为一个任务创建一个新备注 [Create a new comment on a task](#)
- 15.7.22. 获取一个任务的所有备注 [Get all comments on a task](#)
- 15.7.23. 获取一个任务的一个备注 [Get a comment on a task](#)
- 15.7.24. 为一个任务删除一个备注 [Delete a comment on a task](#)
- 15.7.25. 获取一个任务的所有事件 [Get all events for a task](#)
- 15.7.26. 获取一个任务的一个事件 [Get an event on a task](#)
- 15.7.27. 为一个任务创建一个新的附件，带有一个指向外部资源的链接 [Create a new attachment on a task, containing a link to an external resource](#)
- 15.7.28. 为一个任务创建一个新的附件，带有附加文件 [Create a new attachment on a task, with an attached file](#)
- 15.7.29. 获取一个任务的所有附件 [Get all attachments on a task](#)
- 15.7.30. 获取一个任务的一个附件 [Get an attachment on a task](#)
- 15.7.31. 获取一个附件的内容 [Get the content for an attachment](#)
- 15.7.32. 为一个任务删除一个附件 [Delete an attachment on a task](#)
- 15.8. 历史 [History](#)
 - 15.8.1. 获取一个历史流程实例 [Get a historic process instance](#)
 - 15.8.2. 历史流程实例的列表 [List of historic process instances](#)
 - 15.8.3. 查询历史流程实例 [Query for historic process instances](#)
 - 15.8.4. 删除一个历史流程实例 [Delete a historic process instance](#)
 - 15.8.5. 获取一个历史流程实例的身份关联 [Get the identity links of a historic process instance](#)
 - 15.8.6. 获取一个历史流程实例变量的二进制数据 [Get the binary data for a historic process instance variable](#)
 - 15.8.7. 为一个历史流程实例创建一个新的备注 [Create a new comment on a historic process instance](#)
 - 15.8.8. 获取一个历史流程实例的所有备注 [Get all comments on a historic process instance](#)
 - 15.8.9. 获取一个历史流程实例的一个备注 [Get a comment on a historic process instance](#)
 - 15.8.10. 从一个历史流程实例中删除一个备注 [Delete a comment on a historic process instance](#)
 - 15.8.11. 获取一个历史任务实例 [Get a single historic task instance](#)
 - 15.8.12. 获取历史任务实例 [Get historic task instances](#)
 - 15.8.13. 查询历史任务实例 [Query for historic task instances](#)
 - 15.8.14. 删除一个历史任务实例 [Delete a historic task instance](#)
 - 15.8.15. 获取一个历史任务实例的身份关联 [Get the identity links of a historic task instance](#)
 - 15.8.16. 获取一个历史流程实例变量的二进制数据 [Get the binary data for a historic task instance variable](#)
 - 15.8.17. 获取历史活动实例 [Get historic activity instances](#)
 - 15.8.18. 查询历史活动实例 [Query for historic activity instances](#)
 - 15.8.19. 历史变量实例的列表 [List of historic variable instances](#)
 - 15.8.20. 查询历史变量实例 [Query for historic variable instances](#)
 - 15.8.21. 获取历史任务实例变量的二进制数据 [Get the binary data for a historic task instance variable](#)
 - 15.8.22. 获取历史详情 [Get historic detail](#)
 - 15.8.23. 查询历史详情 [Query for historic details](#)
 - 15.8.24. 获取历史详情变量的二进制数据 [Get the binary data for a historic detail variable](#)
- 15.9. 表单 [Forms](#)
 - 15.9.1. 获取表单数据 [Get form data](#)
 - 15.9.2. 提交任务表单数据 [Submit task form data](#)
- 15.10. 数据库表 [Database tables](#)

- 15.10.1. 表的列表 List of tables
- 15.10.2. 获取一个表 Get a single table
- 15.10.3. 获取一个表的列信息 Get column info for a single table
- 15.10.4. 获取一个表的行数据 Get row data for a single table

15.11. 引擎 Engine

- 15.11.1. 获取引擎参数 Get engine properties
- 15.11.2. 获取引擎信息 Get engine info

15.12. 运行时 Runtime

- 15.12.1. 已接收到信号事件 Signal event received

15.13. 作业 Jobs

- 15.13.1. 获取一个作业 Get a single job
- 15.13.2. 删除一个作业 Delete a job
- 15.13.3. 执行一个作业 Execute a single job
- 15.13.4. 获取一个作业的异常栈 Get the exception stacktrace for a job
- 15.13.5. 获取作业的列表 Get a list of jobs

15.14. 用户 Users

- 15.14.1. 获取一个用户 Get a single user
- 15.14.2. 获取用户的列表 Get a list of users
- 15.14.3. 更新一个用户 Update a user
- 15.14.4. 创建一个用户 Create a user
- 15.14.5. 删除一个用户 Delete a user
- 15.14.6. 获取一个用户的图片 Get a user's picture
- 15.14.7. 更新一个用户的图片 Updating a user's picture
- 15.14.8. 列表一个用户的信息 List a user's info
- 15.14.9. 获取一条用户信息 Get a user's info
- 15.14.10. 更新一条用户信息 Update a user's info
- 15.14.11. 创建一条新的用户信息记录 Create a new user's info entry
- 15.14.12. 删除一条用户信息 Delete a user's info

15.15. 组 Groups

- 15.15.1. 获取一个组 Get a single group
- 15.15.2. 获取组的列表 Get a list of groups
- 15.15.3. 更新一个组 Update a group
- 15.15.4. 创建一个组 Create a group
- 15.15.5. 删除一个组 Delete a group
- 15.15.6. 获取一个组中的成员 Get members in a group
- 15.15.7. 为一个组添加一个成员 Add a member to a group
- 15.15.8. 从一个组中删除一个成员 Delete a member from a group

16. CDI集成 CDI integration

16.1. 设置activiti-cdi (Setting up activiti-cdi)

- 16.1.1. 查找流程引擎 Looking up a Process Engine
- 16.1.2. 配置流程引擎 Configuring the Process Engine
- 16.1.3. 部署流程 Deploying Processes

16.2. CDI的基于上下文的流程执行 Contextual Process Execution with CDI

- 16.2.1. 将一个会话关联至一个流程实例 Associating a Conversation with a Process Instance
- 16.2.2. 声明式控制流程 Declaratively controlling the Process
- 16.2.3. 从流程中引用Bean (Referencing Beans from the Process)
- 16.2.4. 使用@BusinessProcessScoped bean (Working with @BusinessProcessScoped beans)
- 16.2.5. 注入流程变量 Injecting Process Variables
- 16.2.6. 接收流程事件 Receiving Process Events
- 16.2.7. 额外功能 Additional Features

16.3. 已知限制 Known Limitations

17. 集成LDAP (LDAP integration)

17.1. 使用 Usage

17.2. 用例 Use cases

17.3. 配置 Configuration

17.4. 参数 Properties

17.5. 在Explorer中集成LDAP (Integrate LDAP in Explorer)

18. 高级 Advanced

18.1. 异步与作业执行器 Async and job executor

- 18.1.1. 异步执行器的设计 Async executor design
- 18.1.2. 作业执行器的设计 Job executor design
- 18.1.3. 异步执行器的优点 Advantages of the Async executor

- 18.1.4. 配置异步执行器 Async executor configuration
- 18.2. 深入流程解析 Hooking into process parsing
- 18.3. 高并发下使用的UUID id生成器 UUID id generator for high concurrency
- 18.4. 多租户 Multitenancy
- 18.5. 执行自定义SQL Execute custom SQL
 - 18.5.1. 基于注解的映射语句 Annotation based Mapped Statements
 - 18.5.2. 基于XML的映射语句 XML based Mapped Statements
- 18.6. 使用ProcessEngineConfigurator进行高级流程引擎配置 Advanced Process Engine configuration with a ProcessEngineConfigurator
- 18.7. 高级查询API：在运行时与历史任务查询间无缝切换 Advanced query API: seamless switching between runtime and historic task querying
- 18.8. 通过覆盖标准SessionFactory自定义身份管理 Custom identity management by overriding standard SessionFactory
- 18.9. 启用安全BPMN 2.0 XML (Enable safe BPMN 2.0 xml)
- 18.10. 事件记录（试验性） Event logging (Experimental)
- 18.11. 禁用批量插入 Disabling bulk inserts
- 18.12. 安全脚本 Secure Scripting
- 19. 使用Activiti-Crystalball仿真（试验性） Simulation with Activiti-Crystalball (Experimental)
 - 19.1. 介绍 Introduction
 - 19.1.1. 简介 Short overview
 - 19.1.2. CrystalBall是独特的 CrystalBall is unique
 - 19.2. 深入CrystalBall (CrystalBall inside)
 - 19.3. 历史分析 History analysis
 - 19.3.1. 历史中的事件 Events from the history.
 - 19.3.2. 回放 PlayBack
 - 19.3.3. 流程引擎调试 Process engine debugger
 - 19.3.4. 重放 Replay
- 20. 工具 Tooling
 - 20.1. JMX
 - 20.1.1. 介绍 Introduction
 - 20.1.2. 快速开始 Quick Start
 - 20.1.3. 属性与操作 Attributes and operations
 - 20.1.4. 配置 Configuration
 - 20.1.5. JMX服务URL (JMX Service URL)
 - 20.2. Maven原型 Maven archetypes
 - 20.2.1. 创建测试用例 Create Test Case
 - 20.3. Docker镜像 Docker images
 - 20.3.1. 介绍 Introduction
 - 20.3.2. 使用 Usage
 - 20.3.3. 构建docker镜像 Build docker image

1. 简介

1.1. 协议

Activiti使用 [the Apache V2 license](#) 协议开源。Activiti Modeler（Web设计器）使用了另一个开源协议 [the LGPL 2.1 license](#)。

1.2. 下载

<http://activiti.org/download.html>

1.3. 源码

Activiti的发布包里包含了大部分源码，这些源码是以jar压缩文件提供的。Activiti的源码可以通过以下链接获得：

<https://github.com/Activiti/Activiti>

1.4. 必要的软件

1.4.1. JDK 6+

Activiti需要JDK 6或以上版本。访问 [Oracle Java SE downloads](#) 点击“下载JDK”按钮。这个页面上有安装文档。安装完成后，执行 `java -version` 校验安装是否成功。能看到JDK的版本信息就说明安装成功了。

1.4.2. Eclipse Indigo 和 Juno

到 [the Eclipse](#) 下载页面选择对应的Eclipse版本下载。把下载的文件解压，然后执行+eclipse+目录下的eclipse文件。手册后续有专门一章 [installing our eclipse designer plugin](#)。

1.5. 反馈问题

Webapp名称	URL	说明角色
----------	-----	------

每一个自重的开发者都应该先看看这个 [提问的智慧](#)。

看完提问的智慧，你可以在 [用户论坛](#) 提问和评论，也可以在我们的 [JIRA 问题追踪系统](#) 创建问题。



虽然Activiti托管在GitHub上，但是不建议使用GitHub的问题追踪系统。如果你想报告问题，不要创建GitHub问题（issue），应该使用我们的JIRA。

1.6. 实验性功能

标记*[EXPERIMENTAL]*的章节介绍的功能 还不够稳定。

+impl.+包下的类都是内部实现类，后续可能发生各种变化。但是，如果是在用户手册中作为配置参数介绍的类，它们是被官方支持的，可以认为是稳定的。

1.7. 内部实现类

在jar文件中，所有包名中包含+impl.+的类（比如，`org.activiti.engine.impl.pvm.delegate`）都是实现类，这些类都是内部类。这些类和接口都不会保证稳定。（不同版本之间可能会有变化。谨慎使用。）

2. 开始

2.1. 一分钟入门

从[Activiti website](#)下载Activiti Explorer的WAR文件后，按照以下步骤使用默认设置运行demo。你需要已经安装Java runtime与Apache Tomcat（事实上，鉴于我们只使用servlet功能，任何web容器都可以运行。但我们主要在Tomcat上进行测试）。

- 将下载的activiti-explorer.war复制到Tomcat的webapps文件夹下。
- 运行Tomcat的bin文件夹下的startup.bat或者startup.sh脚本启动Tomcat。
- Tomcat启动后，打开浏览器访问<http://localhost:8080/activiti-explorer>。使用kermit/kermit登录。

就是这样！Activiti Explorer应用默认使用H2内存数据库。如果你想使用其他数据库配置，请阅读[较长版](#)。

2.2. Activiti安装

要安装Activiti，你需要已经安装Java runtime与Apache Tomcat。同时确认 JAVA_HOME 环境变量已经设置正确。该环境变量的设置方法取决于你的操作系统。

要运行Activiti Explorer与REST web应用，将你从Activiti下载的WAR文件，复制到Tomcat安装目录下的 `webapps` 文件夹中。Explorer应用默认使用内存数据库，示例流程、用户与组。

下表列出demo用户：

用户账号	密码	安全角色
kermit	kermit	admin
gonzo	gonzo	manager
fizzie	fizzie	user

现在你可以访问如下web应用：

Webapp名称	URL	说明
Activiti Explorer	http://localhost:8080/activiti-explorer	流程引擎的用户操作台。使用这个工具来启动新流程、分配任务、查看与接收任务等。这个工具同时可以管理Activiti引擎。

请注意Activiti Explorer示例配置只是用尽可能简单快捷的方式展现Activiti的能力与功能。这*并不*意味着只有这一种使用Activiti的方式。Activiti只是一个jar，它可以被嵌入到任何Java环境中：swing、Tomcat、JBoss、WebSphere，等等。你也可以将Activiti作为典型的、独立运行的BPM服务器。任何在Java里可以做的事情，都可以在Activiti 中做！

2.3. Activiti数据库配置

就像在一分钟示例配置中介绍的，Activiti Explorer默认运行在H2内存数据库上。要让Activiti Explorer使用独立运行的H2数据库或其他数据库，需要修改Activiti Explorer web应用下，WEB-INF/classes目录中的db.properties。

另外，请注意Activiti Explorer默认自动生成演示用户、组、流程定义与模型。要禁用这些设置，需要修改WEB-INF/classes目录下的engine.properties文件。要完全禁用示例设置，可以将所有设置项设为false。你也可以单独禁用或启用其中的某些设置。

```
1 # demo data properties
2 create.demo.users=true
3 create.demo.definitions=true
4 create.demo.models=true
5 create.demo.reports=true
```

2.4. 引入Activiti jar与依赖

我们建议使用Maven(或者Ivy)来引入Activiti的jar与依赖库，因为它简化了我们之间的依赖管理。参

考<http://www.activiti.org/community.html#maven.repository>中的介绍来将必要的jar引入你的项目。

如果不想使用Maven，你也可以自行将jar引入你的项目。从Activiti下载的zip包中的libs文件夹，包含所有Activiti的jar（包括源码jar）。依赖并没有通过这种方式发布。Activiti引擎的依赖列表如下（使用mvn dependency:tree生成）：

```
org.activiti:activiti-engine:jar:5.17.0
+- org.activiti:activiti-bpmn-converter:jar:5.17.0:compile
|   \- org.activiti:activiti-bpmn-model:jar:5.17.0:compile
|       +- com.fasterxml.jackson.core:jackson-core:jar:2.2.3:compile
|       \- com.fasterxml.jackson.core:jackson-databind:jar:2.2.3:compile
|           \- com.fasterxml.jackson.core:jackson-annotations:jar:2.2.3:compile
+- org.activiti:activiti-process-validation:jar:5.17.0:compile
+- org.activiti:activiti-image-generator:jar:5.17.0:compile
+- org.apache.commons:commons-email:jar:1.2:compile
|   +- javax.mail:mail:jar:1.4.1:compile
|   \- javax.activation:activation:jar:1.1:compile
+- org.apache.commons:commons-lang3:jar:3.3.2:compile
+- org.mybatis:mybatis:jar:3.2.5:compile
+- org.springframework:spring-beans:jar:4.0.6.RELEASE:compile
|   \- org.springframework:spring-core:jar:4.0.6.RELEASE:compile
+- joda-time:joda-time:jar:2.6:compile
+- org.slf4j:slf4j-api:jar:1.7.6:compile
+- org.slf4j:jcl-over-slf4j:jar:1.7.6:compile
```

注意：只有使用了邮件任务才必须引入邮件依赖jar。

所有依赖可以在Activiti source code的模块中使用mvn dependency:copy-dependencies下载。

2.5. 下一步

使用Activiti Explorer web应用是一个熟悉Activiti概念与功能的好办法。然而，Activiti的主要目的是用来为你自己的应用添加强大的BPM与工作流功能。下面的章节会帮助你熟悉如何在你的环境中编程使用Activiti：

- 配置章节会教你如何设置Activiti，如何获得ProcessEngine类的实例，他是所有Activiti引擎功能的中心入口。
- API章节会带你了解构成Activiti API的服务。这些服务用简便但强大的方式提供了Activiti引擎的功能，可以使用在任何Java环境下。
- 对深入了解Activiti引擎中流程的编写格式，BPMN 2.0，感兴趣吗？请继续浏览BPMN 2.0章节。

3. 配置 Configuration

3.1. 创建ProcessEngine Creating a ProcessEngine

Activiti流程引擎通过名为activiti.cfg.xml的XML文件进行配置。请注意这种方式与使用Spring创建流程引擎不一样。

获取ProcessEngine，最简单的方式是使用org.activiti.engine.ProcessEngines类：

```
1 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine()
```

这样会从classpath寻找activiti.cfg.xml，并用这个文件中的配置构造引擎。下面的代码展示了一个配置的例子。后续章节会对配置参数进行详细介绍。

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:schemaLocation="http://www.springframework.org/schema/beans
4       http://www.springframework.org/schema/beans/spring-beans.xsd">
5
6   <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
7
8     <property name="jdbcUrl" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
9     <property name="jdbcDriver" value="org.h2.Driver" />
10    <property name="jdbcUsername" value="sa" />
11  </bean>
12 </beans>
```

```

11     <property name="jdbcPassword" value="" />
12
13     <property name="databaseSchemaUpdate" value="true" />
14
15     <property name="jobExecutorActivate" value="false" />
16     <property name="asyncExecutorEnabled" value="true" />
17     <property name="asyncExecutorActivate" value="false" />
18
19     <property name="mailServerHost" value="mail.my-corp.com" />
20     <property name="mailServerPort" value="5025" />
21 </bean>
22 </beans>

```

请注意这个配置XML文件实际上是一个Spring配置文件。但这并不意味着**Activiti**只能用于**Spring**环境！我们只是简单利用Spring内部的解析与依赖注入功能来构造引擎。

也可以通过编程方式使用配置文件，来构造**ProcessEngineConfiguration**对象。也可以使用不同的bean id(例如第3行)。

```

1 ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault();
2 ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource);
3 ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource, String beanName);
4 ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream);
5 ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream, String beanName);

```

也可以不使用配置文件，基于默认创建配置（参考[不同的支持类](#)获得更多信息）。

```

1 ProcessEngineConfiguration.createStandaloneProcessEngineConfiguration();
2 ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration();

```

所有的**ProcessEngineConfiguration.createXXX()**方法都返回**ProcessEngineConfiguration**，并可以继续按需调整。调用**buildProcessEngine()**后，生成一个**ProcessEngine**：

```

1 ProcessEngine processEngine = ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration()
2     .setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB_SCHEMA_UPDATE_FALSE)
3     .setJdbcUrl("jdbc:h2:mem:my-own-db;DB_CLOSE_DELAY=1000")
4     .setAsyncExecutorEnabled(true)
5     .setAsyncExecutorActivate(false)
6     .buildProcessEngine();

```

3.2. ProcessEngineConfiguration bean

activiti.cfg.xml文件中必须包含一个id为'processEngineConfiguration'的bean。

```

1 <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

```

这个bean被用于构建**ProcessEngine**。有多个类可以用于定义**processEngineConfiguration**。这些类用于不同的环境，并各自设置一些默认值。最佳实践是选择（最）匹配你环境的类，以便减少配置引擎需要的参数。下面列出目前可以使用的类（后续版本会提供更多）：

- **org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration**：流程引擎独立运行。Activiti自行处理事务。在默认情况下，数据库检查只在引擎启动时进行（如果Activiti表结构不存在或表结构版本不对，会抛出异常）。
- **org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration**：这是一个便于使用单元测试的类。Activiti自行处理事务。默认使用H2内存数据库。数据库会在引擎启动时创建，并在引擎关闭时删除。使用这个类时，很可能不需要更多的配置（除了使用任务执行器或邮件功能等时）。
- **org.activiti.spring.SpringProcessEngineConfiguration**：在流程引擎处于Spring环境时使用。查看[Spring集成章节](#)获得更多信息。
- **org.activiti.engine.impl.cfg.JtaProcessEngineConfiguration**：用于引擎独立运行，并使用JTA事务的情况。

3.3. 数据库配置 Database configuration

有两种方式配置Activiti引擎使用的数据库。第一种方式是定义数据库的JDBC参数：

- **jdbcUrl**: 数据库的JDBC URL。
- **jdbcDriver**: 特定数据库类型的驱动实现。
- **jdbcUsername**: 用于连接数据库的用户名。
- **jdbcPassword**: 用于连接数据库的密码。

通过提供的JDBC参数构造的数据源，使用默认的**MyBatis**连接池设置。可用下列属性调整这个连接池（来自**MyBatis**文档）：

- **jdbcMaxActiveConnections**: 连接池能够容纳的最大活动连接数量。默认值为10。
- **jdbcMaxIdleConnections**: 连接池能够容纳的最大空闲连接数量。
- **jdbcMaxCheckoutTime**: 连接从连接池“取出”后，被强制返回前的最大时间间隔，单位为毫秒。默认值为20000（20秒）。
- **jdbcMaxWaitTime**: 这是一个底层设置，在连接池获取连接的时间异常长时，打印日志并尝试重新获取连接（避免连接池配置错误造成的永久沉默失败。默认值为20000（20秒）。

数据库配置示例：

```
1 <property name="jdbcUrl" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
2 <property name="jdbcDriver" value="org.h2.Driver" />
3 <property name="jdbcUsername" value="sa" />
4 <property name="jdbcPassword" value="" />
```

也可以使用 `javax.sql.DataSource` 的实现（例如来自 [Apache Commons](#) 的 DBCP）：

```
1 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" >
2   <property name="driverClassName" value="com.mysql.jdbc.Driver" />
3   <property name="url" value="jdbc:mysql://localhost:3306/activiti" />
4   <property name="username" value="activiti" />
5   <property name="password" value="activiti" />
6   <property name="defaultAutoCommit" value="false" />
7 </bean>
8
9 <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
10
11   <property name="dataSource" ref="dataSource" />
12   ...
```

请注意Activiti发布时不包括用于定义数据源的库。需要自行把库（例如来自DBCP）放在你的classpath中。

无论使用JDBC还是数据源方式配置，下列参数都可以使用：

- **databaseType**: 通常不需要专门设置这个参数，因为它可以从数据库连接信息中自动分析得出。只有在自动检测失败时才需要设置。可用值：{h2, mysql, oracle, postgres, mssql, db2}。不使用默认的H2数据库时需要设置这个参数。这个选项会决定创建、删除与查询时使用的脚本。查看“[支持的数据库](#)”章节了解我们支持哪些类型的数据库。
- **databaseSchemaUpdate**: 用于设置流程引擎启动关闭时使用的数据库表结构控制策略。
 - **false** (默认): 当引擎启动时，检查数据库表结构的版本是否匹配库文件版本。版本不匹配时抛出异常。
 - **true**: 构建引擎时，检查并在需要时更新表结构。表结构不存在则会创建。
 - **create-drop**: 引擎创建时创建表结构，并在引擎关闭时删除表结构。

3.4. JNDI数据源配置 JNDI Datasource Configuration

默认情况下，Activiti的数据库配置保存在每个web应用WEB-INF/classes目录下的db.properties文件中。有时这样并不合适，因为这需要用户修改Activiti源码中的db.properties文件并重新编译war包，或者在部署后解开war包并修改db.properties文件。

通过使用JNDI（Java Naming and Directory Interface，Java命名和目录接口）获取数据库连接，连接完全由Servlet容器管理，配置也可以在war部署之外进行管理。同时也比db.properties提供了更多控制连接的参数。

3.4.1. 使用 Usage

要将Activiti Explorer与Activiti Rest web应用从db.properties配置切换至JNDI数据源配置，请打开Spring主配置文件(activiti-webapp-explorer2/src/main/webapp/WEB-INF/activiti-standalone-context.xml与activiti-webapp-rest2/src/main/resources/activiti-context.xml)，并删除名为"dbProperties"与"dataSource"的bean。然后增加下列bean：

```
1 <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
2   <property name="jndiName" value="java:comp/env/jdbc/activitiDB"/>
3 </bean>
```

接下来我们需要新增context.xml文件，其中包含默认的H2配置。也可以用你自己的JNDI配置覆盖它。对于Activiti Explorer，用下列文件替换activiti-webapp-explorer2/src/main/webapp/META-INF /context.xml：

```
1 <Context antiJARLocking="true" path="/activiti-explorer2">
2   <Resource auth="Container"
3     name="jdbc/activitiDB"
4     type="javax.sql.DataSource"
5     scope="Shareable"
6     description="JDBC DataSource"
7     url="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000"
8     driverClassName="org.h2.Driver"
9     username="sa"
10    password=""
```

```

11         defaultAutoCommit="false"
12         initialSize="5"
13         maxWait="5000"
14         maxActive="120"
15         maxIdle="5"/>
16     </Context>

```

对于Activiti REST web应用，新增activiti-webapp-rest2/src/main/webapp/META-INF/context.xml文件，包含下列配置：

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <Context antiJARLocking="true" path="/activiti-rest2">
3      <Resource auth="Container"
4          name="jdbc/activitiDB"
5          type="javax.sql.DataSource"
6          scope="Shareable"
7          description="JDBC DataSource"
8          url="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=-1"
9          driverClassName="org.h2.Driver"
10         username="sa"
11         password=""
12         defaultAutoCommit="false"
13         initialSize="5"
14         maxWait="5000"
15         maxActive="120"
16         maxIdle="5"/>
17  </Context>

```

可选步骤，可以删除Activiti Explorer与Activiti REST web应用中无用的db.properties文件。

3.4.2. 配置 Configuration

根据你使用的servlet容器应用不同，配置JNDI数据源的方式也不同。下面的介绍用于Tomcat，对于其他容器应用，请参考对应的文档。

Tomcat的JNDI资源配置在\$CATALINA_BASE/conf/[engineName]/[hostname]/[warName].xml (对于Activiti Explorer通常是\$CATALINA_BASE/conf/Catalina/localhost/activiti-explorer.xml)。当应用第一次部署时，默认会从Activiti war包中复制context.xml。所以如果存在这个文件则需要替换。例如，如果需要将JNDI资源修改为应用连接MySQL而不是H2，按照下列修改文件：

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <Context antiJARLocking="true" path="/activiti-explorer2">
3      <Resource auth="Container"
4          name="jdbc/activitiDB"
5          type="javax.sql.DataSource"
6          description="JDBC DataSource"
7          url="jdbc:mysql://localhost:3306/activiti"
8          driverClassName="com.mysql.jdbc.Driver"
9          username="sa"
10         password=""
11         defaultAutoCommit="false"
12         initialSize="5"
13         maxWait="5000"
14         maxActive="120"
15         maxIdle="5"/>
16  </Context>

```

3.5. 支持的数据库 Supported databases

下面列出Activiti指定的数据库类型（区分大小写！）。

Activiti数据库类型	示例JDBC URL	备注
h2	jdbc:h2:tcp://localhost/activiti	默认配置的数据库
mysql	jdbc:mysql://localhost:3306/activiti?autoReconnect=true	已使用mysql-connector-java数据库驱动测试
oracle	jdbc:oracle:thin:@localhost:1521:xe	
postgres	jdbc:postgresql://localhost:5432/activiti	
db2	jdbc:db2://localhost:50000/activiti	
mssql	jdbc:sqlserver://localhost:1433;databaseName=activiti (jdbc.driver=com.microsoft.sqlserver.jdbc.SQLServerDriver) OR jdbc:jtds:sqlserver://localhost:1433/activiti (jdbc.driver=net.sourceforge.jtds.jdbc.Driver)	已使用Microsoft JDBC Driver 4.0 (sqljdbc4.jar)与JTDS Driver测试

3.6. 创建数据库表 Creating the database tables

在你的数据库中创建表的最简单方法是：

- 在classpath中增加activiti-engine jar
- 增加合适的数据库驱动
- 在classpath中增加Activiti配置文件(*activiti.cfg.xml*)，指向你的数据库(参考[数据库配置](#))
- 执行*DbSchemaCreate*类的主方法

然而，通常只有数据库管理员可以在数据库中执行DDL语句，在生产环境中这也是最明智的选择。DDL的SQL脚本可以在Activiti下载页面或Activiti发布目录中找到，位于 **database** 子目录。引擎jar (*activiti-engine-x.jar*)的*org/activiti/db/create*包中也有一份(*drop*目录存放删除脚本)。SQL文件的格式为：

```
activiti.{db}.{create|drop}.{type}.sql
```

*db*为支持的数据库，而*type*为

- **engine**: 引擎执行所需的表，必需。
- **identity**: 存储用户、组、用户与组关系的表。这些表是可选的，但在使用引擎自带的默认身份管理时需要使用。
- **history**: 存储历史与审计信息的表。当历史级别设置为*none*时不需要。请注意不使用这些表会导致部分使用历史数据的功能失效（如任务备注）。

MySQL用户请注意：低于5.6.4的MySQL版本不支持*timestamps*或包含毫秒精度的日期。更糟的是部分版本会在创建类似的列时抛出异常，而另一些版本则不会。当使用自动创建/升级时，引擎在执行时会自动修改DDL语句。当使用DDL文件方式建表时，可以使用通用版本，或使用文件名包含*mysql55*的特殊版本（用于5.6.4以下的任何版本）。特殊版本的文件中不会使用毫秒精度的列类型。

具体地说，对于MySQL的版本：

- **<5.6**: 不支持毫秒精度。可以使用DDL文件（使用包含*mysql55*的文件）。可以使用自动创建/升级。
- **5.6.0 - 5.6.3**: 不支持毫秒精度。不可以使用自动创建/升级。建议升级为较新版本的数据库。如果确实需要，可以使用包含*mysql55*的DDL文件。
- **5.6.4+**: 支持毫秒精度。可以使用DDL文件（默认的包含*mysql*的文件）。可以使用自动创建/升级。

请注意在Activiti表已经创建/升级后，更新MySQL数据库，则需要手工修改列类型！

3.7. 数据库表名说明 Database table names explained

Activiti的所有数据库表都以**ACT_**开头。第二部分是说明表用途的两字符标示符。服务API的命名也大略符合这个规则。

- **ACT_RE_***: 'RE'代表**repository**。带有这个前缀的表包含“静态”信息，例如流程定义与流程资源（图片、规则等）。
- **ACT_RU_***: 'RU'代表**runtime**。这些表存储运行时信息，例如流程实例（*process instance*）、用户任务（*user task*）、变量（*variable*）、作业（*job*）等。Activiti只在流程实例运行中保存运行时数据，并在流程实例结束时删除记录。这样保证运行时表小和快。
- **ACT_ID_***: 'ID'代表**identity**。这些表包含身份信息，例如用户、组等。
- **ACT_HI_***: 'HI'代表**history**。这些表存储历史数据，例如已完成的流程实例、变量、任务等。
- **ACT_GE_***: 通用数据。用于不同场景下。

3.8. 数据库升级 Database upgrade

在升级前，请确保你已经（使用数据库的备份功能）备份了数据库。

默认情况下，每次流程引擎创建时会进行版本检查，通常是在你的应用或者Activiti web应用启动的时候。如果Activiti库发现库版本与Activiti数据库表版本不同，会抛出异常。

要进行升级，首先需要将下列配置参数放入你的*activiti.cfg.xml*配置文件：

```
1 <beans >
2
3   <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
4     <!-- ... -->
5     <property name="databaseSchemaUpdate" value="true" />
6     <!-- ... -->
7   </bean>
8
9 </beans>
```

同时，在classpath中加上合适的数据库驱动。升级你应用中的Activiti库，或者启动一个新版本的Activiti，并将它指向旧版本的数据库。将**databaseSchemaUpdate**设置为**true**。当Activiti发现库与数据库表结构不同步时，会自动将数据库表结构升级至新版本。

你还可以直接运行升级DDL语句，也可以从Activiti下载页面获取升级数据库脚本并运行。

3.9. 作业执行器与异步执行器（从5.17.0版本起） Job Executor and Async Executor (since version 5.17.0)

从5.17.0版本开始，在作业执行器之外，Activiti还提供了异步执行器。Activiti引擎可以通过它，以性能更好，也对数据库更友好的方式执行异步作业。

此外，如果在Java EE 7下运行，容器还可以使用符合JSR-236标准的ManagedJobExecutor与ManagedAsyncJobExecutor来管理线程。要启用这个功能，需要在配置中如下加入线程工厂：

```
1 <bean id="threadFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
2   <property name="jndiName" value="java:jboss/ee/concurrency/factory/default" />
3 </bean>
4
5 <bean id="customJobExecutor" class="org.activiti.engine.impl.jobexecutor.ManagedJobExecutor">
6   <!-- ... -->
7   <property name="threadFactory" ref="threadFactory" />
8   <!-- ... -->
9 </bean>
```

如果没有设置线程工厂，上述两个managedxx类都会退化为默认实现（非managed版本）。

3.10. 启用作业执行器 Job executor activation

JobExecutor是管理一组线程的组件，这些线程用于触发定时器（包括后续的异步消息）。在单元测试场景下，使用多线程会很笨重。因此API提供ManagementService.createJobQuery用于查询，以及ManagementService.executeJob用于执行作业。这样作业的执行就可以在单元测试内部控制。为了避免作业执行器的干扰，可以将它关闭。

默认情况下，JobExecutor在流程引擎启动时激活。当你不希望JobExecutor随流程引擎启动时，设置：

```
1 <property name="jobExecutorActivate" value="false" />
```

3.11. 启用异步执行器 Async executor activation

AsyncExecutor是管理线程池的组件，这个线程池用于触发定时器与异步任务。

默认情况下，由于历史原因，当使用JobExecutor时，AsyncExecutor不生效。然而我们建议使用新的AsyncExecutor代替JobExecutor，通过定义两个参数实现

```
1 <property name="asyncExecutorEnabled" value="true" />
2 <property name="asyncExecutorActivate" value="true" />
```

asyncExecutorEnabled参数用于启用异步执行器，代替老的作业执行器。第二个参数asyncExecutorActivate命令Activiti引擎在启动时启动异步执行器线程池。

3.12. 配置邮件服务器 Mail server configuration

配置邮件服务器是可选的。Activiti支持在业务流程中发送电子邮件。发送电子邮件需要配置有效的SMTP邮件服务器。查看[电子邮件任务](#)了解配置选项。

3.13. 配置历史 History configuration

可以选择自定义历史存储的配置。你可以通过调整配置影响历史功能。查看[历史配置](#)了解细节。

```
1 <property name="history" value="audit" />
```

3.14. 配置在表达式与脚本中暴露的bean Exposing configuration beans in expressions and scripts

默认情况下，所有通过activiti.cfg.xml或你自己的Spring配置文件声明的bean，都可以在表达式与脚本中使用。如果你希望限制配置文件中bean的可见性，可以使用流程引擎配置的beans参数。ProcessEngineConfiguration中的beans参数是一个map。当你配置这个参数时，只有在这个map中声明的bean可以在表达式与脚本中使用。bean会使用你在map中指定的名字暴露。

3.15. 配置部署缓存 Deployment cache configuration

鉴于流程定义信息不会改变，为了避免每次使用流程定义时都读取数据库，所有的流程定义都会（在解析后）被缓存。默认情况下，这个缓存没有限制。要限制流程定义缓存，加上如下的参数

```
1 <property name="processDefinitionCacheLimit" value="10" />
```

设置这个参数，会将默认的hashmap替换为LRU缓存，以进行限制。当然，参数的“最佳”取值，取决于总的流程定义数量，以及实际使用的流程定义数量。

你也可以注入自己的缓存实现。它必须是一个实现了 `org.activiti.engine.impl.persistence.deploy.DeploymentCache` 接口的 bean:

```
1 <property name="processDefinitionCache">
2   <bean class="org.activiti.MyCache" />
3 </property>
```

配置规则缓存(rules cache)可以使用类似的名 `knowledgeBaseCacheLimit` 与 `knowledgeBaseCache` 的参数。只有在流程中使用规则任务(rules task)时才需要设置。

3.16. 日志 Logging

自Activiti 5.12版本起，使用SLF4J作为日志框架，替代了之前使用的java.util.logging。所有日志(activiti, spring, mybatis, ...)通过SLF4J路由，并允许你自行选择日志实现。

默认情况下，**Activiti**引擎依赖不会提供**SLF4J**绑定jar。你需要自行将其加入你的项目，以便使用所选的日志框架。如果没有加入实现jar，SLF4J会使用NOP-logger。这时除了一条警告外，任何日志都不会记录。可以从<http://www.slf4j.org/codes.html#StaticLoggerBinder>获取关于绑定的更多信息。

使用Maven可以添加类似这样（这里使用log4j）的依赖，请注意你还需要加上版本：

```
1 <dependency>
2   <groupId>org.slf4j</groupId>
3   <artifactId>slf4j-log4j12</artifactId>
4 </dependency>
```

activiti-explorer与activiti-rest web应用配置为使用Log4j绑定。所有的activiti-*模块运行测试时也会使用Log4j。

重要提示：当使用classpath中带有**commons-logging**的容器时：为了将spring的日志路由至SLF4j，需要使用桥接（参考<http://www.slf4j.org/legacy.html#jclOverSLF4J>）。如果你的容器提供了commons-logging实现，请按照<http://www.slf4j.org/codes.html#release>页面的指示来保证稳定性。

使用Maven的示例（省略了版本）：

```
1 <dependency>
2   <groupId>org.slf4j</groupId>
3   <artifactId>jcl-over-slf4j</artifactId>
4 </dependency>
```

3.17. 映射诊断上下文 Mapped Diagnostic Contexts

从5.13版本开始，Activiti支持SLF4J的映射诊断上下文特性。与需要日志记录的信息一起，下列基本信息也会传递给底层日志记录器：

- processDefinition Id 作为 mdcProcessDefinitionID
- processInstance Id 作为 mdcProcessInstanceID
- execution Id 作为 mdcExecutionId

默认情况下这些信息都不会被日志记录，但可以通过配置日志记录器，以使用想要的格式，与其他日志信息一起显示。例如在log4j中进行如下简单的布局定义，就可以让日志记录器显示上述信息：

```
1 log4j.appender.consoleAppender.layout.ConversionPattern=ProcessDefinitionId=%X{mdcProcessDefinitionID}
   executionId=%X{mdcExecutionId}
   mdcProcessInstanceID=%X{mdcProcessInstanceID}
   mdcBusinessKey=%X{mdcBusinessKey} %m%n
```

在系统任务很关键的情况下这很有用，可以通过例如日志分析器进行日志的严格检查。

3.18. 事件处理器 Event handlers

Activiti 5.15引入了事件机制。它可以让你在引擎中发生多种事件的时候得到通知。查看[所有支持的事件类型](#)了解可用的事件。

可以只为特定种类的事件注册监听器，而不是在任何类型的事件发送时都被通知。可以[通过配置](#)增加引擎全局的事件监听器，[在运行时通过API](#)增加引擎全局的事件监听器，也可以 [在BPMN XML文件为个别流程定义](#)增加事件监听器。

所有被分发的都是 `org.activiti.engine.delegate.event.ActivitiEvent` 的子类。事件（在可用时）提供 `type`，`executionId`，`processInstanceID` 与 `processDefinitionID`。部分事件含有关于发生事件的上下文信息。关于事件包含的附加信息，请参阅[所有支持的事件类型](#)。

3.18.1. 事件监听器实现 Event listener implementation

对事件监听器的唯一要求，是要实现 `org.activiti.engine.delegate.event.ActivitiEventListener` 接口。下面是一个监听器实现的例子，它将接收的所有事件打印至标准输出，并对作业执行相关的事件特别处理。：

```
1 public class MyEventListener implements ActivitiEventListener {
2
3     @Override
4     public void onEvent(ActivitiEvent event) {
5         switch (event.getType()) {
6
7             case JOB_EXECUTION_SUCCESS:
8                 System.out.println("A job well done!");
9                 break;
10
11             case JOB_EXECUTION_FAILURE:
12                 System.out.println("A job has failed...");
13                 break;
14
15             default:
16                 System.out.println("Event received: " + event.getType());
17         }
18     }
19
20     @Override
21     public boolean isFailOnException() {
22         // onEvent方法中的逻辑并不重要，日志失败异常可以被忽略.....
23         return false;
24     }
25 }
```

`isFailOnException()` 方法决定了当事件分发后，`onEvent(..)` 方法抛出异常时的行为。若返回 `false`，忽略异常；返回 `true`，异常不会被忽略而会被上抛，使当前执行的命令失败。如果事件是API调用（或其他事务操作，例如作业执行）的一部分，事务将被回滚。如果事件监听器中并不是重要的业务操作，建议返回 `false`。

Activiti提供了少量基础实现，以简化常用的事件监听器用例。它们可以被用作监听器的示例或基类：

- `org.activiti.engine.delegate.event.BaseEntityEventListener`: 事件监听器基类，可用来监听实体（entity）相关事件，特定或所有实体的事件都可以。它隐藏了类型检测，提供了4个需要覆盖的方法：`onCreate(..)`、`onUpdate(..)`与`onDelete(..)`在实体创建、更新及删除时调用；对所有其他实体相关事件，`onEntityEvent(..)`会被调用。

3.18.2. 配置与安装 Configuration and setup

在流程引擎中配置的事件监听器会在流程引擎启动时生效，引擎重启后也会保持有效。

`eventListeners` 参数配置为 `org.activiti.engine.delegate.event.ActivitiEventListener` 实例的列表(list)。与其他地方一样，你可以声明内联bean定义，也可以用 `ref` 指向已有的bean。下面的代码片段在配置中增加了一个事件监听器，无论任何类型的事件分发时，都会得到通知：

```
1 <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
2     ...
3     <property name="eventListeners">
4         <list>
5             <bean class="org.activiti.engine.example.MyEventListener" />
6         </list>
7     </property>
8 </bean>
```

要在特定类型的事件分发时得到通知，使用 `typedEventListeners` 参数，取值为map。map的key为逗号分隔的事件名字列表（或者一个事件的名字），取值为 `org.activiti.engine.delegate.event.ActivitiEventListener` 实例的列表。下面的代码片段在配置中增加了一个事件监听器，它会在作业执行成功或失败时得到通知：

```
1 <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
2     ...
3     <property name="typedEventListeners">
4         <map>
5             <entry key="JOB_EXECUTION_SUCCESS,JOB_EXECUTION_FAILURE" >
6                 <list>
7                     <bean class="org.activiti.engine.example.MyJobEventListener" />
8                 </list>
9             </entry>
10        </map>
11    </property>
12 </bean>
```


事件分发的顺序由加入监听器的顺序决定。首先，所有普通(`eventListeners` 参数定义的)事件监听器按照他们在 `list` 里的顺序被调用；之后，如果某类型的事件被分发，则该类型(`typedEventListeners` 参数定义的)监听器被调用。

3.18.3. 在运行时增加监听器 Adding listeners at runtime

可以使用API(`RuntimeService`)为引擎增加或删除额外的事件监听器：

```
1  /**
2   * 新增一个监听器，分发器会在所有事件分发时通知。
3   * @param listenerToAdd 要新增的监听器
4   */
5  void addEventListener(ActivitiEventListener listenerToAdd);
6
7  /**
8   * 新增一个监听器，在给定类型的事件发生时被通知。
9   * @param listenerToAdd 要新增的监听器
10  * @param types 监听器需要监听的事件类型
11  */
12 void addEventListener(ActivitiEventListener listenerToAdd, ActivitiEventType... types);
13
14 /**
15  * 从分发器中移除给定监听器。该监听器不再被通知，无论该监听器注册为监听何种类型。
16  * @param listenerToRemove 要移除的监听器
17  */
18 void removeEventListener(ActivitiEventListener listenerToRemove);
```

请注意，运行时新增的监听器在引擎重启后不会保持。

3.18.4. 为流程定义增加监听器 Adding listeners to process definitions

可以为某一流程定义增加监听器。只有与该流程定义相关，或使用该流程定义启动的流程实例相关的事件，才会调用这个监听器。监听器实现可以用完全限定类名（fully qualified classname）定义；也可以定义为表达式，该表达式能被解析为实现监听器接口的bean；也可以配置为抛出消息（message）/信号（signal）/错误（error）的BPMN事件。

执行用户定义逻辑的监听器 Listeners executing user-defined logic

下面的代码片段为流程定义增加了2个监听器。第一个监听器接收任何类型的事件，使用完全限定类名定义。第二个监听器只在作业成功执行或失败时被通知，使用流程引擎配置中 `beans` 参数定义的bean作为监听器。

```
1  <process id="testEventListeners">
2    <extensionElements>
3      <activiti:eventListener class="org.activiti.engine.test.MyEventListener" />
4      <activiti:eventListener delegateExpression="${testEventListener}" events="JOB_EXECUTION_SUCCESS,JOB_EXECUTION_FAILURE" />
5    </extensionElements>
6  </process>
```

实体相关的事件也可以在流程定义中增加监听器，只有在特定实体类型的事件发生时得到通知。下面的代码片段展示了如何设置。可以使用实体的所有（第一个例子）事件，或只使用实体的特定类型（第二个例子）事件。

```
1  <process id="testEventListeners">
2    <extensionElements>
3      <activiti:eventListener class="org.activiti.engine.test.MyEventListener" entityType="task" />
4      <activiti:eventListener delegateExpression="${testEventListener}" events="ENTITY_CREATED" entityType="task" />
5    </extensionElements>
6  </process>
```

`entityType` 可用的值有：`attachment`（附件），`comment`（备注），`execution`（执行），`identity-link`（认证关系），`job`（作业），`process-instance`（流程实例），`process-definition`（流程定义），`task`（任务）。

抛出BPMN事件的监听器 Listeners throwing BPMN events

[试验功能]

处理分发的事件的另一个方法，是抛出BPMN事件。请牢记在心，只有特定种类的Activiti事件类型，抛出BPMN事件才合理。例如，在流程实例被删除时抛出BPMN事件，会导致错误。下面的代码片段展示了如何在流程实例中抛出信号，向外部流程（全局）抛出信号，在流程实例中抛出消息事件，以及在流程实例中抛出错误事件。这里不使用 `class` 或 `delegateExpression`，而要使用 `throwEvent` 属性，以及一个附加属性，用于指定需要抛出的事件类型。

```
<process id="testEventListeners">
```

```

1   <extensionElements>
2     <activiti:eventListener throwEvent="signal" signalName="My signal" events="TASK_ASSIGNED" />
3   </extensionElements>
4 </process>
5

```

```

1 <process id="testEventListeners">
2   <extensionElements>
3     <activiti:eventListener throwEvent="globalSignal" signalName="My signal" events="TASK_ASSIGNED" />
4   </extensionElements>
5 </process>

```

```

1 <process id="testEventListeners">
2   <extensionElements>
3     <activiti:eventListener throwEvent="message" messageName="My message" events="TASK_ASSIGNED" />
4   </extensionElements>
5 </process>

```

```

1 <process id="testEventListeners">
2   <extensionElements>
3     <activiti:eventListener throwEvent="error" errorCode="123" events="TASK_ASSIGNED" />
4   </extensionElements>
5 </process>

```

如果需要使用额外的逻辑判断是否需要抛出BPMN事件，可以扩展Activiti提供的监听器类。通过在你的子类中覆盖 `isValidEvent(ActivitiEvent event)`，可以阻止抛出BPMN事件。相关的类为 `org.activiti.engine.test.api.event.SignalThrowingEventListenerTest`，`org.activiti.engine.impl.bpmn.helper.MessageThrowingEventListener` 与 `org.activiti.engine.impl.bpmn.helper.ErrorThrowingEventListener`。

关于流程定义监听器的说明 Notes on listeners on a process-definition

- 事件监听器只能作为 `extensionElements` 的子元素，声明在 `process` 元素上。不能在个别节点（activity）上定义（事件）监听器。
- `delegateExpression` 中的表达式，与其他表达式（例如在网关中的）不一样，不可以访问执行上下文。只能引用在流程引擎配置中 `beans` 参数定义的bean，或是使用 `spring`（且没有定义 `beans` 参数）时，引用任何实现了监听器接口的 `spring bean`。
- 使用监听器的 `class` 属性时，只会创建唯一一个该类的实例。请确保监听器实现不依赖于成员变量，或确保多线程/上下文的使用安全。
- 如果 `events` 属性使用了不合法的事件类型，或者使用了不合法的 `throwEvent` 值，会在流程定义部署时抛出异常（导致部署失败）。如果 `class` 或 `delegateExecution` 指定了不合法的值（不存在的类，不存在的bean引用，或者代理类没有实现监听器接口），在流程启动（或该流程定义的第一个有效事件分发给这个监听器）时，会抛出异常。请确保引用的类在 `classpath` 中，并且保证表达式能够解析为有效的实例。

3.18.5. 通过API分发事件 Dispatching events through API

我们通过API提供事件分发机制，可以向任何在引擎中注册的监听器分发自定义事件。建议（但不强制）只分发类型为 `CUSTOM` 的 `ActivitiEvents`。可以使用 `RuntimeService` 分发事件：

```

1 /**
2  * 将给定事件分发给所有注册监听器。
3  * @param event 要分发的事件。
4  *
5  * @throws ActivitiException 当分发事件发生异常，或者{@link ActivitiEventDispatcher}被禁用。
6  * @throws ActivitiIllegalArgumentException 当给定事件不可分发
7  */
8 void dispatchEvent(ActivitiEvent event);

```

3.18.6. 支持的事件类型 Supported event types

下表列出引擎中的所有事件类型。每种类型对应 `org.activiti.engine.delegate.event.ActivitiEventType` 中的一个枚举值。

Table 1. Supported events

Event name	Description	Event classes
ENGINE_CREATED	本监听器附着的流程引擎已经创建，并可以响应API调用。	<code>org.activiti...ActivitiEvent</code>
ENGINE_CLOSED	本监听器附着的流程引擎已经关闭，不能再对该引擎的进行API调用。	<code>org.activiti...ActivitiEvent</code>

Event name	Description	Event classes
ENTITY_CREATED	新的实体已经创建。该实体包含在本事件里。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
ENTITY_INITIALIZED	新的实体已经创建并完全初始化。如果任何子实体作为该实体的一部分被创建，本事件会在子实体创建/初始化后触发，与 <code>ENTITY_CREATE</code> 事件相反。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
ENTITY_UPDATED	实体已经更新。该实体包含在本事件里。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
ENTITY_DELETED	实体已经删除。该实体包含在本事件里。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
ENTITY_SUSPENDED	实体已经挂起。该实体包含在本事件里。会为 <code>ProcessDefinitions</code> （流程定义）， <code>ProcessInstances</code> （流程实例）与 <code>Tasks</code> （任务）分发本事件。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
ENTITY_ACTIVATED	实体已被激活。该实体包含在本事件里。会为 <code>ProcessDefinitions</code> , <code>ProcessInstances</code> 与 <code>Tasks</code> 分发本事件。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
JOB_EXECUTION_SUCCESS	作业已经成功执行。该作业包含在本事件里。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
JOB_EXECUTION_FAILURE	作业执行失败。该作业与异常包含在本事件里。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code> and <code>org.activiti...</code> <code>ActivitiExceptionEvent</code>
JOB_RETRIES_DECREMENTED	作业重试次数已经由于执行失败而减少。该作业包含在本事件里。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
TIMER FIRED	定时器已经被触发。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
JOB_CANCELED	作业已经被取消。该作业包含在本事件里。作业会由于API调用取消，任务完成导致关联的边界定时器取消，也会由于新流程定义的部署而取消。	<code>org.activiti...</code> <code>ActivitiEntityEvent</code>
ACTIVITY_STARTED	节点开始执行	<code>org.activiti...</code> <code>ActivitiActivityEvent</code>
ACTIVITY_COMPLETED	节点成功完成	<code>org.activiti...</code> <code>ActivitiActivityEvent</code>
ACTIVITY_CANCELLED	节点将要取消。节点的取消有三个原因（ <code>MessageEventSubscriptionEntity</code> , <code>SignalEventSubscriptionEntity</code> , <code>TimerEntity</code> ）。	<code>org.activiti...</code> <code>ActivitiActivityCancelledEvent</code>
ACTIVITY_SINGALED	节点收到了一个信号	<code>org.activiti...</code> <code>ActivitiSignalEvent</code>
ACTIVITY_MESSAGE_RECEIVED	节点收到了一个消息。事件在节点接收消息前分发。消息接收后，会为该节点分发 <code>ACTIVITY_SIGNAL</code> 或 <code>ACTIVITY_STARTED</code> 事件，取决于其类型（边界事件，或子流程启动事件）。	<code>org.activiti...</code> <code>ActivitiMessageEvent</code>

Event name	Description	Event classes
ACTIVITY_ERROR_RECEIVED	节点收到了错误事件。在节点实际处理错误前分发。该事件的 activityId 含有处理错误的节点的引用。如果错误被成功传递，后续会为节点发送 ACTIVITY_SIGNALLED 或 ACTIVITY_COMPLETE 消息。	org.activiti... ActivitiErrorEvent
UNCAUGHT_BPMN_ERROR	抛出了未捕获的BPMN错误。流程没有该错误的处理器。该事件的 activityId 为空。	org.activiti... ActivitiErrorEvent
ACTIVITY_COMPENSATE	节点将要被补偿。该事件包含将要执行补偿的节点id。	org.activiti... ActivitiActivityEvent
VARIABLE_CREATED	创建了流程变量。本事件包含变量名、取值与关联的执行和任务（若有）。	org.activiti... ActivitiVariableEvent
VARIABLE_UPDATED	更新了已有变量。本事件包含变量名、取值与关联的执行和任务（若有）。	org.activiti... ActivitiVariableEvent
VARIABLE_DELETED	删除了已有变量。本事件包含变量名、最后取值与关联的执行和任务（若有）。	org.activiti... ActivitiVariableEvent
TASK_ASSIGNED	任务分派给了用户。该任务包含在本事件里。	org.activiti... ActivitiEntityEvent
TASK_CREATED	任务已经创建。本事件在 ENTITY_CREATE 事件之后分发。若该任务是流程的一部分，本事件会在任务监听器执行前触发。	org.activiti... ActivitiEntityEvent
TASK_COMPLETED	任务已经结束。本事件在 ENTITY_DELETE 事件前分发。若该任务是流程的一部分，本事件会在流程前进之前触发，并且会跟随一个 ACTIVITY_COMPLETE 事件，指向代表该任务的节点。	org.activiti... ActivitiEntityEvent
PROCESS_COMPLETED	流程完成。在最后一个节点的 ACTIVITY_COMPLETED 事件后分发。当流程实例没有任何路径可以继续时，流程结束。	org.activiti... ActivitiEntityEvent
PROCESS_CANCELLED	流程已经被取消。在流程实例从运行时删除前分发。流程实例使用API调用 RuntimeService.deleteProcessInstance 取消。	org.activiti... ActivitiCancelledEvent
MEMBERSHIP_CREATED	用户加入了一个组。本事件包含了相关的用户和组的id。	org.activiti... ActivitiMembershipEvent
MEMBERSHIP_DELETED	用户从一个组中移出。本事件包含了相关的用户和组的id。	org.activiti... ActivitiMembershipEvent
MEMBERSHIPS_DELETED	组的所有用户将被移出。本事件在用户移出前抛出，因此关联关系仍然可以访问。因为性能原因，不会再为每个被移出的用户抛出 MEMBERSHIP_DELETED 事件。	org.activiti... ActivitiMembershipEvent

引擎中所有的 **ENTITY_*** 事件都与实体关联。下表列出每个实体分发的实体事件：

- ENTITY_CREATED, ENTITY_INITIALIZED, ENTITY_DELETED**: Attachment（附件），Comment（备注），Deployment（部署），Execution（执行），Group（组），IdentityLink（身份关联），Job（作业），Model（模型），ProcessDefinition（流程定义），ProcessInstance（流程实例），Task（任务），User（用户）。
- ENTITY_UPDATED**: Attachment, Deployment, Execution, Group, IdentityLink, Job, Model, ProcessDefinition, ProcessInstance, Task, User.
- ENTITY_SUSPENDED, ENTITY_ACTIVATED**: ProcessDefinition, ProcessInstance/Execution, Task.

3.18.7. 附加信息 Additional remarks

监听器只会被通知所在引擎分发的事件。因此如果你使用不同的引擎，在同一个数据库上运行，只有该监听器注册的引擎生成的事件，会分发给该监听器。其他引擎生成的事件不会分发给这个监听器，不论这些引擎是否运行在同一个JVM下。

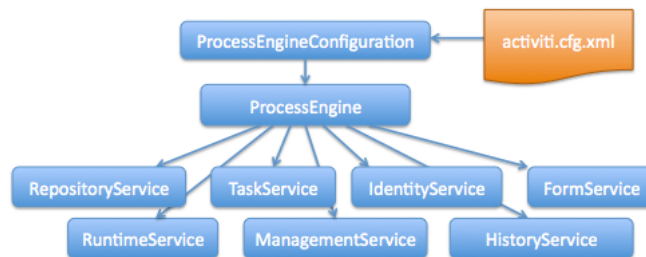
某些事件类型（与实体相关）暴露了目标实体。按照事件类型的不同，有时实体不能被更新（例如实体已经被删除）。如果可能的话，请使用事件暴露的 **EngineServices** 安全操作引擎。即使这样，更新、操作事件中暴露的实体仍然需要小心。

历史不会分发实体事件，因为它们都有对应的运行时实体分发事件。

4. The Activiti API

4.1. 流程引擎API与服务 The Process Engine API and services

引擎API是与Activiti交互的最常用手段。中心入口是 **ProcessEngine**，像配置章节中介绍的一样，可以使用多种方式创建。使用 **ProcessEngine**，可以获得包含工作流/BPM方法的多种服务。**ProcessEngine**与服务对象都是线程安全的，因此可以在整个服务器中保存一份引用。



```
1 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
2
3 RuntimeService runtimeService = processEngine.getRuntimeService();
4 RepositoryService repositoryService = processEngine.getRepositoryService();
5 TaskService taskService = processEngine.getTaskService();
6 ManagementService managementService = processEngine.getManagementService();
7 IdentityService identityService = processEngine.getIdentityService();
8 HistoryService historyService = processEngine.getHistoryService();
9 FormService formService = processEngine.getFormService();
```

ProcessEngines.getDefaultProcessEngine() 在第一次被调用时将初始化并构建流程引擎，在之后的调用都会返回相同的流程引擎。流程引擎的创建通过 **ProcessEngines.init()** 实现，关闭由 **ProcessEngines.destroy()** 实现。

ProcessEngines会扫描所有 **activiti.cfg.xml** 与 **activiti-context.xml** 文件。对于所有的 **activiti.cfg.xml** 文件，流程引擎会以标准Activiti方式构

建： **ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(inputStream).buildProcessEngine()**。

对于所有的 **activiti-context.xml** 文件，流程引擎会以Spring的方式构建：首先构建Spring应用上下文，然后从该上下文中获取流程引擎。

所有的服务都是无状态的。这意味着你可以很容易的在集群环境的多个节点上运行Activiti，使用同一个数据库，而不用担心上一次调用实际在哪台机器上执行。不论在哪里执行，对任何服务的任何调用都是幂等（idempotent）的。

RepositoryService很可能是使用Activiti引擎要用的第一个服务。这个服务提供了管理与控制 **deployments**（部署）与 **process definitions**（流程定义）的操作。在这里简单说明一下，流程定义是BPMN 2.0流程的Java等价副本，展现流程中每一步的结构与行为。**deployment**是 Activiti引擎中的包装单元，一个部署中可以包含多个BPMN 2.0 xml文件，以及其他资源。开发者可以决定在一个部署中包含的内容，可以是单各流程的BPMN 2.0 xml文件，也可以包含多个流程及其相关资源（如'hr-processes'部署可以包含所有与人力资源流程相关的的东西）。**RepositoryService**可用于 **deploy**（部署）这样的包。部署意味着将它上传至引擎，引擎将在储存至数据库之前检查与分析所有的流程。从这里开始，系统知道了这个部署，部署中包含的所有流程都可以启动。

此外，这个服务还可以：

- 查询引擎已知的部署与流程定义。
- 暂停或激活部署中的某些流程，或整个部署。暂停意味着不能再对它进行操作，激活是其反操作。
- 读取各种资源，比如部署中保存的文件，或者引擎自动生成的流程图。
- 读取POJO版本的流程定义。使用它可以用Java而不是xml的方式检查流程。

RepositoryService提供的是静态信息（也就是不会改变，至少不会经常改变的信息），而**RuntimeService**就完全相反。它可以启动流程定义的新流程实例。前面介绍过，**process definition**（流程定义）定义了流程中不同步骤的结构与行为。流程实例则是流程定义的实际执行。同一时刻，一个流程定义通常有多个运行中的实例。**RuntimeService**也用于读取与存储 **process variables**（流程变量）。流程变量是给定流程持有的数据，可以在流程的许多构造中使用（例如排他网关exclusive gateway 经常使用流程变量决定流程下一步要选择的路径）。**RuntimeService**还可以用于查询流程实例与执行（execution）。执行代表了BPMN 2.0中的 **'token'** 概念。通常执行是指向流程实例当前位置的指针。最后，**RuntimeService**还可以在流程实例等待外部触发时使用，以便流程可以继续运

行。流程有许多 `wait states`（暂停状态），`RuntimeService` 服务提供了许多操作用于“通知”流程实例，告知已经接收到外部触发，使流程实例可以继续运行。

对于像Activiti这样的BPM引擎来说，核心是需要人类用户实际操作的任务。所有任务相关的东西都组织在 `TaskService` 中，例如

- 查询分派给用户或组的任务
- 创建 *standalone*（独立运行）任务。这是一种没有关联到流程实例的任务。
- 决定任务的执行用户（*assignee*），或者将用户通过某种方式与任务关联。
- 认领（*claim*）与完成（*complete*）任务。认领是指某人决定成为任务的执行用户，也即他将会完成这个任务。完成任务是指“做这个任务要求的工作”，通常是填写某种表单。

`IdentityService` 很简单。它用于管理（创建，更新，删除，查询……）组与用户。请重点注意，`Activiti` 实际上在运行时并不做任何用户检查。例如任务可以分派给任何用户，而引擎并不会验证系统中是否存在该用户。这是因为 `Activiti` 有时要与 `LDAP`、`Active Directory` 等服务结合使用。

`FormService` 是可选服务。也就是说 `Activiti` 没有它也能很好地运行，而不必牺牲任何功能。这个服务引入了 *start form*（开始表单）与 *task form*（任务表单）的概念。开始表单是在流程实例启动前显示的表单，而任务表单是用户完成任务时显示的表单。`Activiti` 可以在 `BPMN 2.0` 流程定义中定义这些表单。表单服务通过简单的方式暴露这些数据。再次重申，表单不一定要嵌入流程定义，因此这个服务是可选的。

`HistoryService` 暴露所有 `Activiti` 引擎收集的历史数据。当执行流程时，引擎会保存许多数据（可以配置），例如流程实例启动时间，谁在执行哪个任务，完成任务花费的事件，每个流程实例的执行路径，等等。这个服务主要提供查询这些数据的能力。

`ManagementService` 通常在用 `Activiti` 编写用户应用时不需要使用。它可以用于读取数据库表与表原始数据的信息，也提供了对作业（*job*）的查询与管理操作。`Activiti` 中很多地方都使用作业，例如定时器（*timer*），异步操作（*asynchronous continuation*），延时暂停/激活（*delayed suspension/activation*）等等。后续会详细介绍这些内容。

参考 [javadocs](#) 了解服务操作与引擎API的更多信息。

4.2. 异常策略 Exception strategy

`Activiti` 的异常基类是 `org.activiti.engine.ActivitiException`，是未检查异常（unchecked exception）。在任何API操作时都可能会抛出这个异常，[javadoc](#) 记录了每个方法可能发生的异常。例如，从 `TaskService` 中摘录：

```
1 /**
2  * 当任务成功执行时调用。
3  * @param taskId 需要完成的任务id，不能为null。
4  * @throws ActivitiObjectNotFoundException 若给定id找不到任务。
5  */
6 void complete(String taskId);
```

在上例中，如果传递的id找不到任务，会抛出异常。并且，由于javadoc中明确要求taskId不能为null，因此如果传递了 `null` 值，会抛出 `ActivitiIllegalArgumentException` 异常。

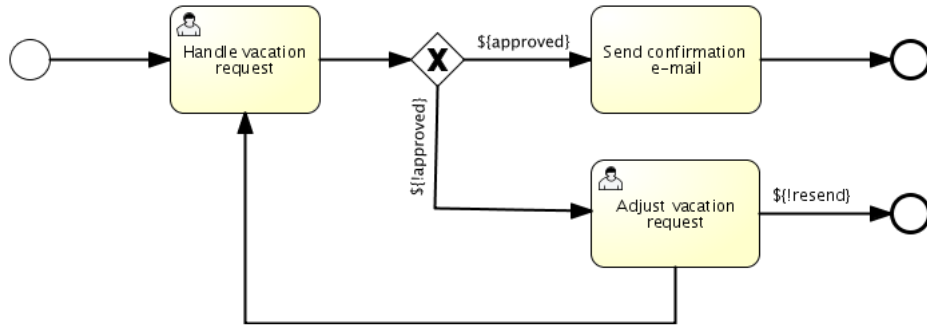
尽管我们想避免过大的异常层次结构，我们还是添加了下述在特定情况下抛出的异常子类。所有流程执行与API调用中发生的错误，如果不符合下面列出的异常，会统一抛出 `ActivitiExceptions`。

- `ActivitiWrongDbException`: 当 `Activiti` 引擎检测到数据库表结构版本与引擎版本不匹配时抛出。
- `ActivitiOptimisticLockingException`: 当对同一数据实体的并发访问，导致数据存储发生乐观锁时抛出。
- `ActivitiClassLoadingException`: 当需要载入的类（如 `JavaDelegates`, `TaskListeners`, ...）无法找到，或载入时发生错误时抛出。
- `ActivitiObjectNotFoundException`: 当请求或要操作的对象不存在时抛出。
- `ActivitiIllegalArgumentException`: 这个异常说明调用 `Activiti` API 时使用了不合法的参数。可能是引擎配置中的不合法值，或者是API调用传递的不合法参数，也可能是流程定义中的不合法值。
- `ActivitiTaskAlreadyClaimedException`: 当调用 `taskService.claim(...)`，而该任务已经被认领时抛出。

4.3. 使用Activiti services(Working with the Activiti services)

如前所述，与 `Activiti` 引擎交互的方式，是使用 `org.activiti.engine.ProcessEngine` 类实例暴露的服务。下面的示例假定你已经有可运行的 `Activiti` 环境，也就是说，你可以访问有效的 `org.activiti.engine.ProcessEngine`。如果你只是简单地想尝试下面的代码，可以下载或克隆 `Activiti` 单元测试模板，导入你的IDE，在 `org.activiti.MyUnitTest` 单元测试中增加一个 `testUserguideCode()` 方法。

这段小教程的最终目标是生成一个业务流程，模拟公司中简单的请假流程：



4.3.1. 部署流程 Deploying the process

所有有关“静态”数据（例如流程定义）的东西，都可以通过**RepositoryService**访问。从概念上说，所有这种静态数据，都是Activiti引擎“仓库（repository）”中的内容。

在 `src/test/resources/org/activiti/test` 资源目录（如果没有使用单元测试模板，也可以是其他任何地方）下创建名为 `VacationRequest.bpmn20.xml` 的xml文件，写入下列内容。请注意这个章节不会解释例子中用到的xml的结构。如果需要，请先阅读 [BPMN 2.0 章节 the BPMN 2.0 chapter](#) 了解这种结构。

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <definitions id="definitions"
3     targetNamespace="http://activiti.org/bpmn20"
4     xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xmlns:activiti="http://activiti.org/bpmn">
7
8     <process id="vacationRequest" name="Vacation request">
9
10         <startEvent id="request" activiti:initiator="employeeName">
11             <extensionElements>
12                 <activiti:formProperty id="numberOfDays" name="Number of days" type="long" value="1" required="true"/>
13                 <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyy)" datePattern="dd-MM-yyyy hh:mm"
14 type="date" required="true" />
15                 <activiti:formProperty id="vacationMotivation" name="Motivation" type="string" />
16             </extensionElements>
17         </startEvent>
18         <sequenceFlow id="flow1" sourceRef="request" targetRef="handleRequest" />
19
20         <userTask id="handleRequest" name="Handle vacation request" >
21             <documentation>
22                 ${employeeName} would like to take ${numberOfDays} day(s) of vacation (Motivation: ${vacationMotivation}).
23             </documentation>
24             <extensionElements>
25                 <activiti:formProperty id="vacationApproved" name="Do you approve this vacation" type="enum" required="true">
26                     <activiti:value id="true" name="Approve" />
27                     <activiti:value id="false" name="Reject" />
28                 </activiti:formProperty>
29                 <activiti:formProperty id="managerMotivation" name="Motivation" type="string" />
30             </extensionElements>
31             <potentialOwner>
32                 <resourceAssignmentExpression>
33                     <formalExpression>management</formalExpression>
34                 </resourceAssignmentExpression>
35             </potentialOwner>
36         </userTask>
37         <sequenceFlow id="flow2" sourceRef="handleRequest" targetRef="requestApprovedDecision" />
38
39         <exclusiveGateway id="requestApprovedDecision" name="Request approved?" />
40         <sequenceFlow id="flow3" sourceRef="requestApprovedDecision" targetRef="sendApprovalMail">
41             <conditionExpression xsi:type="tFormalExpression">${vacationApproved == 'true'}</conditionExpression>
42         </sequenceFlow>
43
44         <task id="sendApprovalMail" name="Send confirmation e-mail" />
45         <sequenceFlow id="flow4" sourceRef="sendApprovalMail" targetRef="theEnd1" />
46         <endEvent id="theEnd1" />
47
48         <sequenceFlow id="flow5" sourceRef="requestApprovedDecision" targetRef="adjustVacationRequestTask">
49             <conditionExpression xsi:type="tFormalExpression">${vacationApproved == 'false'}</conditionExpression>
50         </sequenceFlow>
51
52         <userTask id="adjustVacationRequestTask" name="Adjust vacation request">
53             <documentation>
54                 Your manager has disapproved your vacation request for ${numberOfDays} days.
55                 Reason: ${managerMotivation}
56             </documentation>
57             <extensionElements>
58                 <activiti:formProperty id="numberOfDays" name="Number of days" value="${numberOfDays}" type="long"

```

```

59 required="true"/>
60 <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyy)" value="${startDate}"
61 datePattern="dd-MM-yyyy hh:mm" type="date" required="true" />
62 <activiti:formProperty id="vacationMotivation" name="Motivation" value="${vacationMotivation}" type="string" />
63 <activiti:formProperty id="resendRequest" name="Resend vacation request to manager?" type="enum" required="true">
64 <activiti:value id="true" name="Yes" />
65 <activiti:value id="false" name="No" />
66 </activiti:formProperty>
67 </extensionElements>
68 <humanPerformer>
69 <resourceAssignmentExpression>
70 <formalExpression>${employeeName}</formalExpression>
71 </resourceAssignmentExpression>
72 </humanPerformer>
73 </userTask>
74 <sequenceFlow id="flow6" sourceRef="adjustVacationRequestTask" targetRef="resendRequestDecision" />
75
76 <exclusiveGateway id="resendRequestDecision" name="Resend request?" />
77 <sequenceFlow id="flow7" sourceRef="resendRequestDecision" targetRef="handleRequest">
78 <conditionExpression xsi:type="tFormalExpression">${resendRequest == 'true'}</conditionExpression>
79 </sequenceFlow>
80
81 <sequenceFlow id="flow8" sourceRef="resendRequestDecision" targetRef="theEnd2">
82 <conditionExpression xsi:type="tFormalExpression">${resendRequest == 'false'}</conditionExpression>
83 </sequenceFlow>
84 <endEvent id="theEnd2" />
85
</process>
</definitions>

```

你必须首先部署（deploy）流程，以使Activiti引擎可以识别它。部署意味着引擎会将BPMN 2.0 xml文件解析为可执行的东西，并为部署中包含的每个流程定义创建新的数据库记录。这样，引擎重启后，仍能获取已部署的流程：

```

1 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
2 RepositoryService repositoryService = processEngine.getRepositoryService();
3 repositoryService.createDeployment()
4 .addClasspathResource("org/activiti/test/VacationRequest.bpmn20.xml")
5 .deploy();
6
7 Log.info("Number of process definitions: " + repositoryService.createProcessDefinitionQuery().count());

```

在[部署](#)章节阅读更多部署相关信息。

4.3.2. 启动流程实例 Starting a process instance

向Activiti引擎部署流程定义后，可以用它启动流程实例。每个流程定义都可以有多个流程实例。流程定义就像是“蓝图”，而流程实例在运行时执行它。

所有与流程运行时状态相关的东西都可以在**RuntimeService**中找到。启动流程实例有多种不同的方法。在下列代码片段中，使用流程定义xml中定义的key启动流程实例。在启动流程实例时，我们也设置了一些流程变量（**process variables**），因为第一个用户任务（**user task**）的描述（**description**）中的表达式（**expression**）需要用到它们。流程变量的使用很普遍，因为它们为流程定义的流程实例赋予了意义。流程变量使每个流程实例与其他实例不同。

```

1 Map<String, Object> variables = new HashMap<String, Object>();
2 variables.put("employeeName", "Kermit");
3 variables.put("numberOfDays", new Integer(4));
4 variables.put("vacationMotivation", "I'm really tired!");
5
6 RuntimeService runtimeService = processEngine.getRuntimeService();
7 ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("vacationRequest", variables);
8
9 // Verify that we started a new process instance
10 Log.info("Number of process instances: " + runtimeService.createProcessInstanceQuery().count());

```

4.3.3. 完成任务 Completing tasks

流程启动时，第一步是一个用户任务。这个步骤必须由系统用户操作。一般会提供“待办任务”列出所有需要该用户处理的任务。下面的代码片段展示如何进行这种列表的查询：

```

1 // 获取management组的所有任务
2 TaskService taskService = processEngine.getTaskService();
3 List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();
4 for (Task task : tasks) {
5     Log.info("Task available: " + task.getName());
6 }

```


我们需要结束这个任务才能使流程实例继续运行。对于Activiti引擎来说，就是`complete`（完成）这个任务。下面的代码片段展示了如何操作：

```
1 Task task = tasks.get(0);
2
3 Map<String, Object> taskVariables = new HashMap<String, Object>();
4 taskVariables.put("vacationApproved", "false");
5 taskVariables.put("managerMotivation", "We have a tight deadline!");
6 taskService.complete(task.getId(), taskVariables);
```

现在流程实例会继续向下一步进行。在这个例子里，下一步允许雇员填写一个表单，用来修改提交的请假申请。雇员可以重新提交请假申请，这会使流程从开始任务重新开始运行。

4.3.4. 暂停与激活流程 Suspending and activating a process

可以暂停流程定义。当流程定义暂停后，不能再创建新的流程实例（会抛出异常）。使用`RepositoryService`暂停流程定义：

```
1 repositoryService.suspendProcessDefinitionByKey("vacationRequest");
2 try {
3     runtimeService.startProcessInstanceByKey("vacationRequest");
4 } catch (ActivitiException e) {
5     e.printStackTrace();
6 }
```

要重新激活流程定义，可以调用`repositoryService.activateProcessDefinitionXXX`方法。

也可以暂停流程实例。当流程实例暂停后，不能进行流程操作（例如完成任务会抛出异常），作业（如定时器）也不会执行。可以调用`runtimeService.suspendProcessInstance`暂停流程实例。调用`runtimeService.activateProcessInstanceXXX`重新激活流程实例。

4.3.5. 扩展阅读 Further reading

在前面的章节，我们大致介绍了Activiti的功能。我们会在未来扩展这些内容，覆盖更多的Activiti API。当然，与其他开源项目一样，最好的学习方法是研究代码与阅读Javadocs！

4.4. 查询API (Query API)

从引擎中查询数据有两种方式：查询API与原生(native)查询。查询API可以使用链式API，通过编程方式进行类型安全的查询。你可以在查询中增加各种条件（所有条件都用做AND逻辑），也可以明确指定排序。下面是示例代码：

```
1 List<Task> tasks = taskService.createTaskQuery()
2     .taskAssignee("kermit")
3     .processVariableValueEquals("orderId", "0815")
4     .orderByDueDate().asc()
5     .list();
```

有时你需要更强大的查询，例如使用OR操作符查询，或者使用查询API不能满足查询条件要求。我们为这种需求提供了原生查询，可以自己写SQL查询。返回类型由使用的查询对象决定，数据也会映射到正确的对象中，如Task, ProcessInstance, Execution....查询会在数据库中进行，因此你需要使用数据库中定义的表名与列名。这需要了解内部数据结构，因此建议小心使用原生查询。数据库表名可以通过API读取，这样可以将依赖关系减到最小。

```
1 List<Task> tasks = taskService.createNativeTaskQuery()
2     .sql("SELECT count(*) FROM " + managementService.getTableName(Task.class) + " T WHERE T.NAME_ = #{taskName}")
3     .parameter("taskName", "gonzoTask")
4     .list();
5
6 long count = taskService.createNativeTaskQuery()
7     .sql("SELECT count(*) FROM " + managementService.getTableName(Task.class) + " T1, "
8         + managementService.getTableName(VariableInstanceEntity.class) + " V1 WHERE V1.TASK_ID_ = T1.ID_")
9     .count();
```

4.5. 变量 Variables

流程实例按步骤执行时，需要同时也使用一些数据。在Activiti中，这些数据称作`variables`(变量)，并会存储在数据库中。变量可以用在表达式中（例如在排他网关中用于选择正确的出口路径），用在java服务任务（java service task）中用于调用外部服务（例如为服务调用提供输入或结果存储），等等。

流程实例可以拥有变量（称作`process variables`，流程变量），执行(`executions`)——流程当前活动节点的指针，以及用户任务也可以拥有变量。流程实例可以持有任意数量的变量，每个变量都存储在`ACT_RU_VARIABLE`数据库表的一行中。

任何`startProcessInstanceXXX`方法都有一个可选参数，用于在流程实例创建并启动时设置变量。例如，在`RuntimeService`中：

```
1 ProcessInstance startProcessInstanceByKey(String processDefinitionKey, Map<String, Object> variables);
```

也可以在流程执行中加入变量。例如(*RuntimeService*):

```
1 void setVariable(String executionId, String variableName, Object value);
2 void setVariableLocal(String executionId, String variableName, Object value);
3 void setVariables(String executionId, Map<String, ? extends Object> variables);
4 void setVariablesLocal(String executionId, Map<String, ? extends Object> variables);
```

请注意可以为给定执行（请记住流程实例由一颗执行的树tree of executions组成）设置*local*（局部）变量。局部变量将只在该执行中可见，而对执行树的上层则不可见。这可以用于 数据不应该在流程实例级别传播，或者变量在流程实例的不同路径中有不同的值（例如使用并行路径时）的情况。

像下面展示的，可以读取变量。请注意*TaskService*中有类似的方法。这意味着任务与执行一样，可以持有局部变量，其生存期为任务持续的时间。

```
1 Map<String, Object> getVariables(String executionId);
2 Map<String, Object> getVariablesLocal(String executionId);
3 Map<String, Object> getVariables(String executionId, Collection<String> variableNames);
4 Map<String, Object> getVariablesLocal(String executionId, Collection<String> variableNames);
5 Object getVariable(String executionId, String variableName);
6 <T> T getVariable(String executionId, String variableName, Class<T> variableClass);
```

变量通常用于Java代理（Java delegates），表达式（expressions），执行（execution），任务监听器（tasklisteners），脚本（scripts）等等。在这些结构中，提供了当前的*execution*或*task*对象，可用于变量的设置、读取。简单示例如下：

```
1 execution.getVariables();
2 execution.getVariables(Collection<String> variableNames);
3 execution.getVariable(String variableName);
4
5 execution.setVariables(Map<String, object> variables);
6 execution.setVariable(String variableName, Object value);
```

请注意也可以使用上例中方法的*local*（局部变量）版本。

由于历史（与向后兼容的）原因，当调用上述任何方法时，引擎实际上会从数据库中取出所有变量。也就是说，如果你有10个变量，使用*getVariable("myVariable")*获取其中的一个，实际上其他9个变量也会从数据库取出并缓存。这并不坏，因为后续的调用可以不必再读取数据库。比如，你的流程定义包含三个连续的服务任务 *service task*（因此它们在同一个数据库事务里），在第一个服务任务里通过一次调用获取全部变量，也许比在每个服务任务里分别获取需要的变量要好。请注意对读取 与设置变量都是这样。

当然，如果使用大量变量，或者你希望精细控制数据库查询与流量，上述做法并不合适。从Activiti 5.17版本起，引入了可以更精细控制的方法。这个方法有一个可选的参数，告诉引擎是否需要在幕后将所有变量读取并缓存：

```
1 Map<String, Object> getVariables(Collection<String> variableNames, boolean fetchAllVariables);
2 Object getVariable(String variableName, boolean fetchAllVariables);
3 void setVariable(String variableName, Object value, boolean fetchAllVariables);
```

当*fetchAllVariables*参数为*true*时，行为与上面描述的完全一样：读取或设置一个变量时，所有的变量都将被读取并缓存。

而在参数值为*false*时，会使用明确的查询，其他变量不会被读取或缓存。只有指定的变量的值会被缓存，用于后续使用。

4.6. 表达式 Expressions

Activiti使用UEL进行表达式解析。UEL代表*Unified Expression Language*，是EE6规范的一部分（查看EE6规范了解更多信息）。为了在所有环境上支持UEL标准的所有最新特性，我们使用JUEL的修改版本。

表达式可以用于例如Java服务任务 *Java Service tasks*, 执行监听器 *Execution Listeners*, 任务监听器 *Task Listeners* 与 条件流 *Conditional sequence flows*。尽管有值表达式与方法表达式两种表达式，通过Activiti的抽象，使它们都可以在需要*expression*（表达式）的地方使用。

- 值表达式 **Value expression**: 解析为一个值。默认情况下，所有流程变量都可以使用。（若使用Spring）所有的Spring bean也可以用在表达式里。例如：

```
${myVar}
${myBean.myProperty}
```

- 方法表达式 **Method expression**: 注入一个方法，可以带或不带参数。当注入不带参数的方法时，要确保在方法名后添加空括号（以避免与值表达式混淆）。传递的参数可以是字面值（literal value），也可以是表达式，它们会被自动解析。例如：

```
    ${printer.print()}
    ${myBean.addNewOrder('orderName')}
    ${myBean.doSomething(myVar, execution)}
```

请注意，表达式支持解析（包括比较）原始类型（primitive）、bean、list、array（数组）与map。

除了所有流程变量外，还有一些默认对象可在表达式中使用：

- **execution**: 持有进行中执行（execution）额外信息的 **DelegateExecution**。
- **task**: 持有当前任务（task）额外信息的 **DelegateTask**。请注意：只在任务监听器的表达式中可用。
- **authenticatedUserId**: 当前已验证的用户id。如果没有已验证的用户，该变量不可用。

更多实际使用例子，请查看[Spring中的表达式 Expressions in Spring](#), [Java服务任务 Java Service tasks](#), [执行监听器 Execution Listeners](#), [任务监听器 Task Listeners](#)或者[条件流 Conditional sequence flows](#)。

4.7. 单元测试 Unit testing

业务流程是软件项目的必要组成部分，也需要使用测试一般应用逻辑的方法，也就是单元测试，对它们进行测试。Activiti是嵌入式的Java引擎，因此为业务流程编写单元测试就与编写一般的单元测试一样简单。

Activiti支持JUnit版本3与4的单元测试风格。按照JUnit 3的风格，必须扩展

（extended）**org.activiti.engine.test.ActivitiTestCase**。它通过保护（protected）成员变量提供对ProcessEngine与服务的访问。在测试的**setup()**中，processEngine会默认使用classpath中的**activiti.cfg.xml**资源初始化。如果要指定不同的配置文件，请覆盖**getConfigurationResource()**方法。当使用相同的配置资源时，流程引擎会静态缓存，用于多个单元测试。

通过扩展**ActivitiTestCase**，你可以使用**org.activiti.engine.test.Deployment**注解测试方法。在测试运行前，会部署与测试类在同一个包下的格式为**testClassName.testMethod.bpmn20.xml**的资源文件。在测试结束时，会删除这个部署，包括所有相关的流程实例，任务，等等。也可以使用**Deployment**注解显式指定资源位置。查看该类以获得更多信息。

综上所述，JUnit 3风格的测试看起来类似：

```
1 public class MyBusinessProcessTest extends ActivitiTestCase {
2
3     @Deployment
4     public void testSimpleProcess() {
5         runtimeService.startProcessInstanceByKey("simpleProcess");
6
7         Task task = taskService.createTaskQuery().singleResult();
8         assertEquals("My Task", task.getName());
9
10        taskService.complete(task.getId());
11        assertEquals(0, runtimeService.createProcessInstanceQuery().count());
12    }
13 }
```

要使用JUnit 4的风格书写单元测试并达成同样的功能，必须使用**org.activiti.engine.test.ActivitiRule** Rule。这样能够通过它的getter获得流程引擎与服务。对于**ActivitiTestCase**（上例），包含**Rule**就可以使用**org.activiti.engine.test.Deployment**注解（参见上例解释其用途及配置），并且会自动在classpath中寻找默认配置文件。当使用相同的配置资源时，流程引擎会静态缓存，用于多个单元测试。

下面的代码片段展示了JUnit 4风格的测试与**ActivitiRule**的用法。

```
1 public class MyBusinessProcessTest {
2
3     @Rule
4     public ActivitiRule activitiRule = new ActivitiRule();
5
6     @Test
7     @Deployment
8     public void ruleUsageExample() {
9         RuntimeService runtimeService = activitiRule.getRuntimeService();
10        runtimeService.startProcessInstanceByKey("ruleUsage");
11
12        TaskService taskService = activitiRule.getTaskService();
13        Task task = taskService.createTaskQuery().singleResult();
14        assertEquals("My Task", task.getName());
15
16        taskService.complete(task.getId());
17        assertEquals(0, runtimeService.createProcessInstanceQuery().count());
18    }
19 }
```

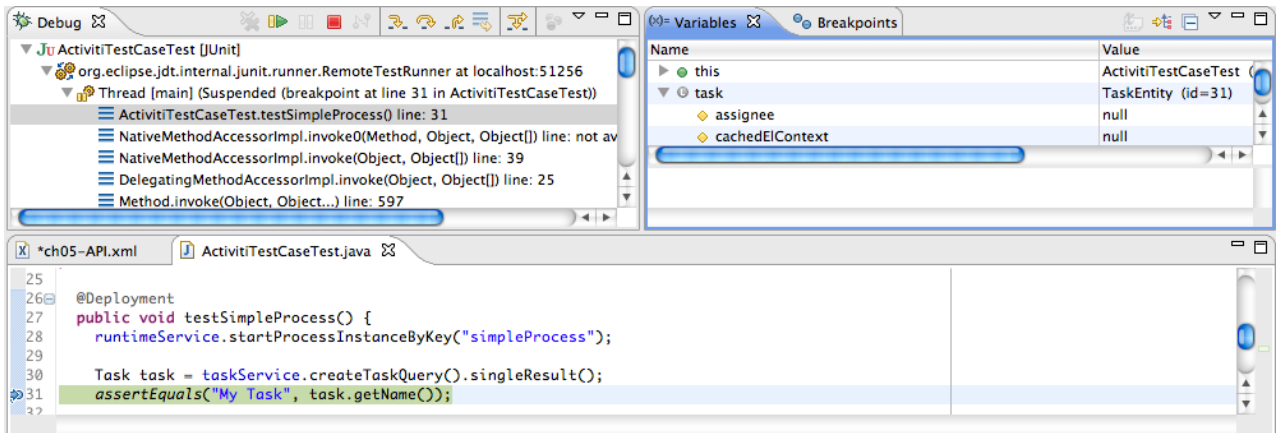
4.8. Debug单元测试(Debugging unit tests)

当使用H2内存数据库进行单元测试时，下面的介绍可以让你在debug过程中容易地检查Activiti数据库中的数据。截图来自Eclipse，但原理应该与其他IDE相似。

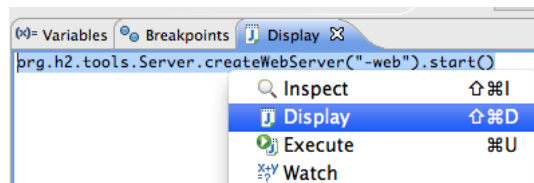
假设我们的单元测试的某处放置了**breakpoint**（断点）。在Eclipse里可以通过在代码左侧条上双击实现：

```
27 public void testSimpleProcess() {
28     runtimeService.startProcessInstanceByKey("simpleProcess");
29
30     Task task = taskService.createTaskQuery().singleResult();
31     assertEquals("My Task", task.getName());
32 }
```

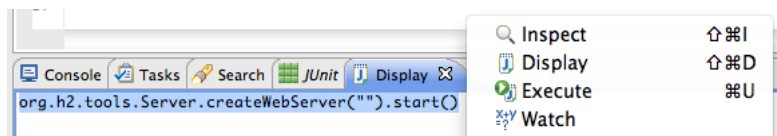
如果我们在**debug**模式（在测试类中右键，选择“Run as”，然后选择“JUnit test”）下运行单元测试，测试进程会在断点处暂停，这样我们就可以在右上窗口中查看测试中的变量。



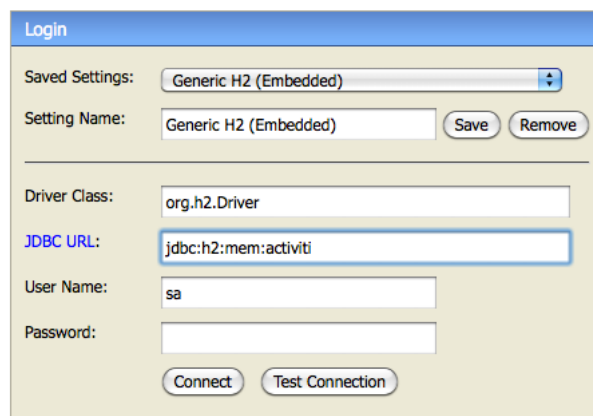
要检查Activiti的数据，打开**Display**窗口（如果没有找到这个窗口，打开 Window→Show View→Other，然后选择**Display**），并键入（可以使用代码补全）**org.h2.tools.Server.createWebServer("-web").start()**



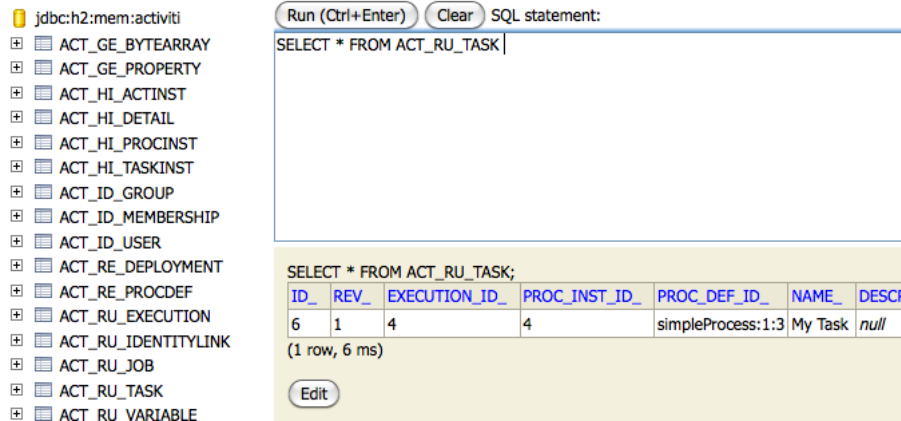
选中刚键入的行并右键点击。然后选择'Display'（或者用快捷方式执行）



现在打开浏览器并访问<http://localhost:8082>，填入内存数据库的JDBC URL（默认为**jdbc:h2:mem:activiti**），然后点击connect按钮。



现在你可以看到Activiti的数据，可以用来理解你的单元测试执行流程的方式是什么，以及为什么这样。



4.9. Web应用中的流程引擎 The process engine in a web application

ProcessEngine 是线程安全的类，可以很容易地在多个线程间共享。在web应用中，这意味着可以在容器启动时创建引擎，并在容器关闭时关闭引擎。

下面的代码片段展示了如何在纯Servlet环境中，简单的通过 **ServletContextListener** 初始化与销毁流程引擎。

```
1 public class ProcessEnginesServletContextListener implements ServletContextListener {
2
3     public void contextInitialized(ServletContextEvent servletContextEvent) {
4         ProcessEngines.init();
5     }
6
7     public void contextDestroyed(ServletContextEvent servletContextEvent) {
8         ProcessEngines.destroy();
9     }
10
11 }
```

contextInitialized 方法委托给 **ProcessEngines.init()**。它会在classpath中查找 **activiti.cfg.xml** 资源文件，并为每个配置分别创建 **ProcessEngine**（例如多个jar都包含配置文件）。如果在classpath中有多个这样的资源文件，请确保它们都使用不同的名字。需要使用流程引擎时，可以获取通过

```
1 ProcessEngines.getDefaultProcessEngine()
```

或者

```
1 ProcessEngines.getProcessEngine("myName");
```

当然，就像[配置章节 configuration section](#)中介绍的，还可以使用各种不同的方式创建流程引擎。

context-listener的 **contextDestroyed** 方法委托给 **ProcessEngines.destroy()**。它会妥善关闭所有已初始化的流程引擎。

5. 集成Spring (Spring integration)

尽管完全可以脱离Spring使用Activiti，我们仍提供了很多非常好的集成特性，将在这一章节介绍。

5.1. ProcessEngineFactoryBean

ProcessEngine 可以被配置为普通的Spring bean。入口是 **org.activiti.spring.ProcessEngineFactoryBean** 类。这个bean处理流程引擎配置，并创建流程引擎。这意味着在Spring中，创建与设置参数与[配置章节 configuration section](#)中介绍的一样。集成Spring的配置与引擎bean为：

```
1 <bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
2     ...
3 </bean>
4
5 <bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
6     <property name="processEngineConfiguration" ref="processEngineConfiguration" />
7 </bean>
```

请注意 **processEngineConfiguration** bean现在使用 **org.activiti.spring.SpringProcessEngineConfiguration** 类。

5.2. 事务 Transactions

我们会一步一步地解释(Activiti)发行版里，Spring示例中的 `SpringTransactionIntegrationTest`。下面是我们示例中使用的Spring配置文件（`SpringTransactionIntegrationTest-context.xml`）。下面的小节包含了 `dataSource`（数据源），`transactionManager`（事务管理器），`processEngine`（流程引擎）与 Activiti引擎服务。

将 `DataSource` 传递给 `SpringProcessEngineConfiguration`（使用“`dataSource`”参数）时，Activiti会在内部使用 `org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy` 对得到的数据源进行包装（wrap）。这是为了保证从数据源获取的SQL连接与Spring的事务可以协同工作。也就是说不需要在Spring配置中对数据源进行代理（proxy）。尽管仍然可以将 `TransactionAwareDataSourceProxy` 传递给 `SpringProcessEngineConfiguration`——在这种情况下，不会再进行包装。

请确保如果自行在Spring配置中声明了 `TransactionAwareDataSourceProxy`，不会将它用在已经配置Spring事务的资源上（例如 `DataSourceTransactionManager` 与 `JPATransactionManager` 就需要未代理的数据源）。

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:context="http://www.springframework.org/schema/context"
3       xmlns:tx="http://www.springframework.org/schema/tx"
4       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans
6 http://www.springframework.org/schema/beans/spring-beans.xsd
7
8 http://www.springframework.org/schema/context
9 http://www.springframework.org/schema/context/spring-context-2.5.xsd
10
11 http://www.springframework.org/schema/tx
12 http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
13
14 <bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
15   <property name="driverClass" value="org.h2.Driver" />
16   <property name="url" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
17   <property name="username" value="sa" />
18   <property name="password" value="" />
19 </bean>
20
21 <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
22   <property name="dataSource" ref="dataSource" />
23 </bean>
24
25 <bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
26   <property name="dataSource" ref="dataSource" />
27   <property name="transactionManager" ref="transactionManager" />
28   <property name="databaseSchemaUpdate" value="true" />
29   <property name="jobExecutorActivate" value="false" />
30 </bean>
31
32 <bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
33   <property name="processEngineConfiguration" ref="processEngineConfiguration" />
34 </bean>
35
36 <bean id="repositoryService" factory-bean="processEngine" factory-method="getRepositoryService" />
37 <bean id="runtimeService" factory-bean="processEngine" factory-method="getRuntimeService" />
38 <bean id="taskService" factory-bean="processEngine" factory-method="getTaskService" />
39 <bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService" />
40 <bean id="managementService" factory-bean="processEngine" factory-method="getManagementService" />
41 ...
```

这个Spring配置文件的余下部分包含了在这个示例中要用到的bean与配置：

```
1 <beans>
2 ...
3 <tx:annotation-driven transaction-manager="transactionManager"/>
4
5 <bean id="userBean" class="org.activiti.spring.test.UserBean">
6   <property name="runtimeService" ref="runtimeService" />
7 </bean>
8
9 <bean id="printer" class="org.activiti.spring.test.Printer" />
10
11 </beans>
```

使用任何Spring的方式创建应用上下文（application context）。在这个例子中，可以使用classpath中的XML资源配置来创建Spring应用上下文：

```
1 ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(
2   "org/activiti/examples/spring/SpringTransactionIntegrationTest-context.xml");
```

或者在单元测试中：


```
1 | @ContextConfiguration("classpath:org/activiti/spring/test/transaction/SpringTransactionIntegrationTest-context.xml")
```

现在就可以获取服务bean，并反射调用(invoke)它们的方法。ProcessEngineFactoryBean会为服务加上额外的拦截器(interceptor)，为Activiti服务方法设置Propagation.REQUIRED事务语义(transaction semantics)。因此，我们可以像这样使用repositoryService部署流程：

```
1 | RepositoryService repositoryService =
2 |     (RepositoryService) applicationContext.getBean("repositoryService");
3 | String deploymentId = repositoryService
4 |     .createDeployment()
5 |     .addClasspathResource("org/activiti/spring/test/hello.bpmn20.xml")
6 |     .deploy()
7 |     .getId();
```

还有另一种方法也可以使用。在这个例子中，userBean.hello()方法被Spring事务包围，Activiti服务方法的调用会加入这个事务。

```
1 | UserBean userBean = (UserBean) applicationContext.getBean("userBean");
2 | userBean.hello();
```

UserBean看起来像这样。请记住在上面的Spring bean配置中，我们已经将repositoryService注入了userBean。

```
1 | public class UserBean {
2 |
3 |     /** 已经由Spring注入 */
4 |     private RuntimeService runtimeService;
5 |
6 |     @Transactional
7 |     public void hello() {
8 |         // 这里可以在你的领域模型 (domain model) 中进行事务操作，
9 |         // 它会与Activiti RuntimeService的startProcessInstanceByKey
10 |         // 合并在同一个事务里
11 |         runtimeService.startProcessInstanceByKey("helloProcess");
12 |     }
13 |
14 |     public void setRuntimeService(RuntimeService runtimeService) {
15 |         this.runtimeService = runtimeService;
16 |     }
17 | }
```

5.3. 表达式 Expressions

当使用ProcessEngineFactoryBean时，默认BPMN流程中所有的表达式 expressions 都可以“看见”所有的Spring bean。通过可以配置的map，可以限制表达式能使用的bean，甚至可以完全禁止表达式使用bean。下面的例子只暴露了一个bean（printer），可以使用“printer”作为key访问。要完全禁止表达式使用bean，可以将SpringProcessEngineConfiguration的‘beans’参数设为空list。如果不设置‘beans’参数，则上下文中的所有bean都将可以使用。

```
1 | <bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
2 |     ...
3 |     <property name="beans">
4 |         <map>
5 |             <entry key="printer" value-ref="printer" />
6 |         </map>
7 |     </property>
8 | </bean>
9 |
10 | <bean id="printer" class="org.activiti.examples.spring.Printer" />
```

现在可以在表达式中使用这个暴露的bean了：例如，SpringTransactionIntegrationTest `hello.bpmn20.xml` 展示了如何通过UEL方法表达式(method expression)注入Spring bean：

```
1 | <definitions id="definitions">
2 |
3 |     <process id="helloProcess">
4 |
5 |         <startEvent id="start" />
6 |         <sequenceFlow id="flow1" sourceRef="start" targetRef="print" />
7 |
8 |         <serviceTask id="print" activiti:expression="#{printer.printMessage()}" />
9 |         <sequenceFlow id="flow2" sourceRef="print" targetRef="end" />
10 |
11 |         <endEvent id="end" />
12 |
13 |     </process>
```

```
14
15 </definitions>
```

其中 **Printer** 为:

```
1 public class Printer {
2
3     public void printMessage() {
4         System.out.println("hello world");
5     }
6 }
```

Spring bean配置（上面已经展示过）为:

```
1 <beans>
2     ...
3
4     <bean id="printer" class="org.activiti.examples.spring.Printer" />
5
6 </beans>
```

5.4. 自动部署资源 Automatic resource deployment

集成Spring还提供了部署资源的特殊方式。在流程引擎配置中，可以指定一组资源。当流程引擎被创建时，这些资源都会被扫描并部署。有过滤器用于阻止重复部署。只有当资源确实发生变化时，才会重新部署至Activiti数据库。在Spring容器经常重启（例如测试时）的时候，这很有用。

这里有个例子:

```
1 <bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
2     ...
3     <property name="deploymentResources"
4         value="classpath*:org/activiti/spring/test/autodeployment/autodeploy.*.bpmn20.xml" />
5 </bean>
6
7 <bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
8     <property name="processEngineConfiguration" ref="processEngineConfiguration" />
9 </bean>
```

默认情况下，这个配置会将符合这个过滤器的所有资源组织在一起，作为Activiti引擎的一个部署。重复检测过滤器将作用于整个部署，避免重复地部署未改变资源。有时这不是你想要的。例如，如果用这种方式部署了一组资源，即使只有其中的一个资源发生了改变，整个部署都会被认为已经改变，因此这个部署中所有的所有流程定义都会被重新部署。这将导致每个流程定义都会刷新版本号（流程定义id会变化），即使实际上只有一个流程发生了变化。

可以使用 **SpringProcessEngineConfiguration** 中的额外参数+**deploymentMode**，定制部署的选择方式。这个参数定义了在一组符合过滤器的资源中，组织部署的方式。默认这个参数有3个可用值:

- **default**: 将所有资源组织在一个部署中，整体用于重复检测过滤。这是默认值，在未设置这个参数时也会用这个值。
- **single-resource**: 为每个资源创建一个单独的部署，并用于重复检测过滤。当你希望单独部署每一个流程定义，并且在它发生变化时创建新的流程定义版本，应该使用这个值。
- **resource-parent-folder**: 为同一个目录下的资源创建一个单独的部署，并用于重复检测过滤。这个参数值可以为大多数资源创建独立的部署。同时仍可以通过将部分资源放在同一个目录下，将它们组织在一起。这里有一个将 **deploymentMode** 设置为 **single-resource** 的例子:

```
1 <bean id="processEngineConfiguration"
2     class="org.activiti.spring.SpringProcessEngineConfiguration">
3     ...
4     <property name="deploymentResources" value="classpath*:activiti/*.bpmn" />
5     <property name="deploymentMode" value="single-resource" />
6 </bean>
```

如果上述 **deploymentMode** 的参数值不能满足要求，还可以自定义组织部署的行为。创建 **SpringProcessEngineConfiguration** 的子类，并覆盖 **getAutoDeploymentStrategy(String deploymentMode)** 方法。这个方法决定了对于给定的 **deploymentMode** 参数值，使用何种部署策略。

5.5. 单元测试 Unit testing

与Spring集成后，业务流程可以非常简单地使用标准的 **Activiti测试工具 Activiti testing facilities**进行测试。下面的例子展示了如何通过典型的基于Spring的单元测试，对业务流程进行测试:


```

1  @RunWith(SpringJUnit4ClassRunner.class)
2  @ContextConfiguration("classpath:org/activiti/spring/test/junit4/springTypicalUsageTest-context.xml")
3  public class MyBusinessProcessTest {
4
5      @Autowired
6      private RuntimeService runtimeService;
7
8      @Autowired
9      private TaskService taskService;
10
11     @Autowired
12     @Rule
13     public ActivitiRule activitiSpringRule;
14
15     @Test
16     @Deployment
17     public void simpleProcessTest() {
18         runtimeService.startProcessInstanceByKey("simpleProcess");
19         Task task = taskService.createTaskQuery().singleResult();
20         assertEquals("My Task", task.getName());
21
22         taskService.complete(task.getId());
23         assertEquals(0, runtimeService.createProcessInstanceQuery().count());
24     }
25 }
26

```

请注意要让这个例子可以正常工作，需要在Spring配置中定义`org.activiti.engine.test.ActivitiRule` bean（在上面的例子中通过auto-wiring注入）。

```

1  <bean id="activitiRule" class="org.activiti.engine.test.ActivitiRule">
2      <property name="processEngine" ref="processEngine" />
3  </bean>

```

5.6. 通过Hibernate 4.2.x使用JPA (JPA with Hibernate 4.2.x)

要在Activiti引擎的服务任务或者监听器逻辑中使用Hibernate 4.2.x JPA，需要添加Spring ORM的额外依赖。对Hibernate 4.1.x或更低则不需要。需要添加的依赖为：

```

1  <dependency>
2      <groupId>org.springframework</groupId>
3      <artifactId>spring-orm</artifactId>
4      <version>${org.springframework.version}</version>
5  </dependency>

```

5.7. Spring Boot

Spring Boot是一个应用框架，按照[官网](#)的介绍，可以轻松地创建独立运行的，生产级别的，基于`Spring`的应用，并且可以“直接运行”。坚持使用`Spring`框架与第三方库，使你可以轻松地开始使用。大多数`Spring Boot`应用只需要很少的`Spring`配置。

要获得更多关于Spring Boot的信息，请查阅<http://projects.spring.io/spring-boot/>

Activiti与Spring Boot的集成目前只是试验性的。我们已经与Spring的提交者共同开发，但为时尚早。我们欢迎试用并提供反馈。

5.7.1. 兼容性 Compatibility

Spring Boot需要JDK 7运行时环境。可以通过调整配置，在JDK6下运行。请查阅Spring Boot的文档。

5.7.2. 开始 Getting started

Spring Boot提倡约定大于配置。要开始工作，简单地在你的项目中添加`spring-boot-starters-basic`依赖。例如在Maven中：

```

1  <dependency>
2      <groupId>org.activiti</groupId>
3      <artifactId>activiti-spring-boot-starter-basic</artifactId>
4      <version>${activiti.version}</version>
5  </dependency>

```

就这么简单。这个依赖会自动向classpath添加正确的Activiti与Spring依赖。现在你可以编写Spring Boot应用了：

```

1  import org.springframework.boot.SpringApplication;
2  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
3  import org.springframework.context.annotation.ComponentScan;
4  import org.springframework.context.annotation.Configuration;
5
6  @Configuration
7  @ComponentScan

```

```

8  @EnableAutoConfiguration
9  public class MyApplication {
10
11      public static void main(String[] args) {
12          SpringApplication.run(MyApplication.class, args);
13      }
14
15  }

```

Activiti需要数据库存储数据。如果你运行上面的代码，会得到提示性的异常信息，指出需要在classpath中添加数据库驱动依赖。现在添加H2数据库依赖：

```
1 <dependency>
2     <groupId>com.h2database</groupId>
3     <artifactId>h2</artifactId>
4     <version>1.4.183</version>
5 </dependency>
```

应用这次可以启动了。你会看到类似这样的输出：

```
. _ _ _ _ _  
/\ / _ '- _ _ _ (-) - _ _ _ \ \ \ \  
( ( ) \_ | ' _ | ' _ | | ' _ V _ | \ \ \ \  
\V _ _ )| _ | | | | | | | ( _ | ) ) ) )  
' | _ | . _ | _ | _ | _ | _ \ , | / / / /  
===== |_ ===== |_ =/_/_/_/  
:: Spring Boot ::           (v1.1.6.RELEASE)  
  
MyApplication                : Starting MyApplication on ...  
s.c.a.AnnotationConfigApplicationContext : Refreshing  
org.springframework.context.annotation.AnnotationConfigApplicationContext@33cb5951:  
  startup date [Wed Dec 17 15:24:34 CET 2014]; root of context hierarchy  
a.s.b.AbstractProcessEngineConfiguration : No process definitions were  
found using the specified path (classpath:/processes/**/*.bpmn20.xml).  
o.activiti.engine.impl.db.DbSqlSession   : performing create on engine  
with resource org/activiti/db/create/activiti.h2.create.engine.sql  
o.activiti.engine.impl.db.DbSqlSession   : performing create on history  
with resource org/activiti/db/create/activiti.h2.create.history.sql  
o.activiti.engine.impl.db.DbSqlSession   : performing create on identity  
with resource org/activiti/db/create/activiti.h2.create.identity.sql  
o.a.engine.impl.ProcessEngineImpl        : ProcessEngine default created  
o.a.e.i.a.DefaultAsyncJobExecutor        : Starting up the default async  
job executor [org.activiti.spring.SpringAsyncExecutor].  
o.a.e.i.a.AcquireTimerJobsRunnable       : {} starting to acquire async  
jobs due  
o.a.e.i.a.AcquireAsyncJobsDueRunnable    : {} starting to acquire async  
jobs due  
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX  
exposure on startup  
MyApplication                          : Started MyApplication in  
2.019 seconds (JVM running for 2.294)
```

只是在classpath中添加依赖，并使用`@EnableAutoConfiguration`注解，就会在幕后发生很多事情：

- 自动创建了内存数据库（因为classpath中有H2驱动），并传递给Activiti流程引擎配置
- 创建并暴露了Activiti ProcessEngine bean
- 所有的Activiti服务都暴露为Spring bean
- 创建了Spring Job Executor

并且，`processes`目录下的任何BPMN 2.0流程定义都会被自动部署。创建`processes`目录，并在其中创建示例流程定义（命名为`one-task-process.bpmn20.xml`）：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions
3     xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
4     xmlns:activiti="http://activiti.org/bpmn"
5     targetNamespace="Examples">
6
7     <process id="oneTaskProcess" name="The One Task Process">
8         <startEvent id="theStart" />
9         <sequenceFlow id="flow1" sourceRef="theStart" targetRef="theTask" />
10        <userTask id="theTask" name="my task" />
11        <sequenceFlow id="flow2" sourceRef="theTask" targetRef="theEnd" />
12        <endEvent id="theEnd" />
13    </process>
14 </definitions>
```

```

13     </process>
14
15 </definitions>

```

然后添加下列代码，以测试部署是否生效。*CommandLineRunner*是一个特殊的Spring bean，在应用启动时执行：

```

1  @Configuration
2  @ComponentScan
3  @EnableAutoConfiguration
4  public class MyApplication {
5
6      public static void main(String[] args) {
7          SpringApplication.run(MyApplication.class, args);
8      }
9
10     @Bean
11     public CommandLineRunner init(final RepositoryService repositoryService,
12                                   final RuntimeService runtimeService,
13                                   final TaskService taskService) {
14
15         return new CommandLineRunner() {
16             @Override
17             public void run(String... strings) throws Exception {
18                 System.out.println("Number of process definitions : "
19                                   + repositoryService.createProcessDefinitionQuery().count());
20                 System.out.println("Number of tasks : " + taskService.createTaskQuery().count());
21                 runtimeService.startProcessInstanceByKey("oneTaskProcess");
22                 System.out.println("Number of tasks after process start: " + taskService.createTaskQuery().count());
23             }
24         };
25     }
26 }
27
28 }

```

会得到这样的输出：

```

Number of process definitions : 1
Number of tasks : 0
Number of tasks after process start : 1

```

5.7.3. 更换数据源与连接池 Changing the database and connection pool

上面也提到过，Spring Boot的约定大于配置。默认情况下，如果classpath中只有H2，就会创建内存数据库，并传递给Activiti流程引擎配置。

可以简单地通过提供DataSource bean来覆盖默认配置，来更换数据源。我们在这里使用*DataSourceBuilder*类，这是Spring Boot的辅助类。如果classpath中有Tomcat, HikariCP 或者 Commons DBCP，就会（按照这个顺序，先是Tomcat）选择一个（作为连接池）。例如，要切换到MySQL数据库：

```

1  @Bean
2  public DataSource database() {
3      return DataSourceBuilder.create()
4          .url("jdbc:mysql://127.0.0.1:3306/activiti-spring-boot?characterEncoding=UTF-8")
5          .username("alfresco")
6          .password("alfresco")
7          .driverClassName("com.mysql.jdbc.Driver")
8          .build();
9  }

```

从Maven依赖中移除H2，并为classpath添加MySQL驱动与Tomcat连接池：

```

1  <dependency>
2      <groupId>mysql</groupId>
3      <artifactId>mysql-connector-java</artifactId>
4      <version>5.1.34</version>
5  </dependency>
6  <dependency>
7      <groupId>org.apache.tomcat</groupId>
8      <artifactId>tomcat-jdbc</artifactId>
9      <version>8.0.15</version>
10 </dependency>

```

应用这次启动后，可以看到使用了MySQL作为数据库（也使用了Tomcat连接池框架）：

```

org.activiti.engine.impl.db.DbSqlSession : performing create on
engine with resource
org/activiti/db/create/activiti.mysql.create.engine.sql
org.activiti.engine.impl.db.DbSqlSession : performing create on
history with resource
org/activiti/db/create/activiti.mysql.create.history.sql
org.activiti.engine.impl.db.DbSqlSession : performing create on
identity with resource
org/activiti/db/create/activiti.mysql.create.identity.sql

```

多次重启应用，会发现任务的数量增加了（H2内存数据库在关闭后会丢失，而MySQL不会）。

5.7.4. REST支持 (REST support)

通常在嵌入的Activiti引擎之上，需要提供REST API（用于与公司的不同服务交互）。Spring Boot让这变得很容易。在classpath中添加下列依赖：

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   <version>${spring.boot.version}</version>
5 </dependency>

```

创建一个新的Spring服务类，并创建两个方法：一个用于启动流程，另一个用于获得给定任务办理人的任务列表。在这里我们简单地包装了Activiti调用，但很明显在实际使用场景中会比这复杂得多。

```

1 @Service
2 public class MyService {
3
4     @Autowired
5     private RuntimeService runtimeService;
6
7     @Autowired
8     private TaskService taskService;
9
10    @Transactional
11    public void startProcess() {
12        runtimeService.startProcessInstanceByKey("oneTaskProcess");
13    }
14
15    @Transactional
16    public List<Task> getTasks(String assignee) {
17        return taskService.createTaskQuery().taskAssignee(assignee).list();
18    }
19
20 }

```

现在可以用@RestController来注解类，以创建REST端点（endpoint）。在这里我们简单地委派给了上面定义的服务。

```

1 @RestController
2 public class MyRestController {
3
4     @Autowired
5     private MyService myService;
6
7     @RequestMapping(value="/process", method= RequestMethod.POST)
8     public void startProcessInstance() {
9         myService.startProcess();
10    }
11
12    @RequestMapping(value="/tasks", method= RequestMethod.GET, produces=MediaType.APPLICATION_JSON_VALUE)
13    public List<TaskRepresentation> getTasks(@RequestParam String assignee) {
14        List<Task> tasks = myService.getTasks(assignee);
15        List<TaskRepresentation> dtos = new ArrayList<TaskRepresentation>();
16        for (Task task : tasks) {
17            dtos.add(new TaskRepresentation(task.getId(), task.getName()));
18        }
19        return dtos;
20    }
21
22    static class TaskRepresentation {
23
24        private String id;
25        private String name;
26
27        public TaskRepresentation(String id, String name) {
28            this.id = id;
29            this.name = name;
30        }
31    }
32 }

```

```

30     }
31
32     public String getId() {
33         return id;
34     }
35     public void setId(String id) {
36         this.id = id;
37     }
38     public String getName() {
39         return name;
40     }
41     public void setName(String name) {
42         this.name = name;
43     }
44
45 }
46
47 }

```

自动组件扫描(`@ComponentScan`)会找到我们添加在应用类上的`@Service`与`@RestController`。再次运行应用类，现在可以与REST API交互了。例如使用cURL:

```

curl http://localhost:8080/tasks?assignee=kermit
[]

curl -X POST http://localhost:8080/process
curl http://localhost:8080/tasks?assignee=kermit
[{"id":"10004","name":"my task"}]

```

5.7.5. JPA支持 (JPA support)

要为Spring Boot中的Activiti添加JPA支持，增加下列依赖:

```

1 <dependency>
2   <groupId>org.activiti</groupId>
3   <artifactId>activiti-spring-boot-starter-jpa</artifactId>
4   <version>${activiti.version}</version>
5 </dependency>

```

这会加入Spring的配置，以及JPA用的bean。默认使用Hibernate作为JPA提供者。

创建一个简单的实体类:

```

1 @Entity
2 class Person {
3
4     @Id
5     @GeneratedValue
6     private Long id;
7
8     private String username;
9
10    private String firstName;
11
12    private String lastName;
13
14    private Date birthDate;
15
16    public Person() {
17    }
18
19    public Person(String username, String firstName, String lastName, Date birthDate) {
20        this.username = username;
21        this.firstName = firstName;
22        this.lastName = lastName;
23        this.birthDate = birthDate;
24    }
25
26    public Long getId() {
27        return id;
28    }
29
30    public void setId(Long id) {
31        this.id = id;
32    }
33
34    public String getUsername() {
35        return username;
36    }
37

```

```

38     public void setUsername(String username) {
39         this.username = username;
40     }
41
42     public String getFirstName() {
43         return firstName;
44     }
45
46     public void setFirstName(String firstName) {
47         this.firstName = firstName;
48     }
49
50     public String getLastName() {
51         return lastName;
52     }
53
54     public void setLastName(String lastName) {
55         this.lastName = lastName;
56     }
57
58     public Date getBirthDate() {
59         return birthDate;
60     }
61
62     public void setBirthDate(Date birthDate) {
63         this.birthDate = birthDate;
64     }
65 }

```

默认情况下，如果没有使用内存数据库，不会自动创建数据库表。在classpath中创建__application.properties_文件并加入下列参数：

```
spring.jpa.hibernate.ddl-auto=update
```

添加下列类：

```

1 public interface PersonRepository extends JpaRepository<Person, Long> {
2
3     Person findByUsername(String username);
4
5 }

```

这是一个Spring存储(repository)，提供了直接可用的增删改查。我们添加了通过username查找Person的方法。Spring会基于约定自动实现它（也就是使用names属性）。

现在进一步增强我们的服务：

- 在类上添加@*Transactional*。请注意，通过上面添加的JPA依赖，之前使用的DataSourceTransactionManager会自动替换为JpaTransactionManager。
- *startProcess*增加了任务办理人参数，用于查找Person，并将Person JPA对象作为流程变量存入流程实例。
- 添加了创建示例用户的方法。CommandLineRunner使用它打桩数据库。

```

1 @Service
2 @Transactional
3 public class MyService {
4
5     @Autowired
6     private RuntimeService runtimeService;
7
8     @Autowired
9     private TaskService taskService;
10
11     @Autowired
12     private PersonRepository personRepository;
13
14     public void startProcess(String assignee) {
15
16         Person person = personRepository.findByUsername(assignee);
17
18         Map<String, Object> variables = new HashMap<String, Object>();
19         variables.put("person", person);
20         runtimeService.startProcessInstanceByKey("oneTaskProcess", variables);
21     }
22
23     public List<Task> getTasks(String assignee) {
24         return taskService.createTaskQuery().taskAssignee(assignee).list();
25     }
26
27     public void createDemoUsers() {

```



```

28         if (personRepository.findAll().size() == 0) {
29             personRepository.save(new Person("jbarrez", "Joram", "Barrez", new Date()));
30             personRepository.save(new Person("trademakers", "Tijs", "Rademakers", new Date()));
31         }
32     }
33 }
34 }

```

CommandLineRunner现在为:

```

1  @Bean
2  public CommandLineRunner init(final MyService myService) {
3
4      return new CommandLineRunner() {
5          public void run(String... strings) throws Exception {
6              myService.createDemoUsers();
7          }
8      };
9  }
10 }

```

RestController也有小改动（只展示新方法），以配合上面的改动。HTTP POST现在有了body，存有办理人用户名:

```

@RestController
public class MyRestController {

    @Autowired
    private MyService myService;

    @RequestMapping(value="/process", method= RequestMethod.POST)
    public void startProcessInstance(@RequestBody StartProcessRepresentation startProcessRepresentation) {
        myService.startProcess(startProcessRepresentation.getAssignee());
    }

    ...

    static class StartProcessRepresentation {

        private String assignee;

        public String getAssignee() {
            return assignee;
        }

        public void setAssignee(String assignee) {
            this.assignee = assignee;
        }
    }
}

```

最后，为了试用Spring-JPA-Activiti集成，我们在流程定义中，将Person JPA对象的id指派为任务办理人:

```

1  <userTask id="theTask" name="my task" activiti:assignee="${person.id}"/>

```

现在可以通过在POST body中提供用户名，启动一个新的流程实例:

```

curl -H "Content-Type: application/json" -d '{"assignee": "jbarrez"}' http://localhost:8080/process

```

也可以使用Person id获取任务列表:

```

curl http://localhost:8080/tasks?assignee=1

[{"id": "12505", "name": "my task"}]

```

5.7.6. 扩展阅读 Further Reading

很明显还有很多Spring Boot相关的内容还没有提及，例如简单的JTA集成，构建能在主流应用服务器上运行的war文件。还有很多关于Spring Boot集成的内容:

- Actuator支持
- Spring Integration支持

- Rest API集成：启动Spring应用中嵌入的Activiti Rest API
- Spring Security支持

目前这些领域都是初版，未来会不断演进。

6. 部署 Deployment

6.1. 业务存档 Business archives

要部署流程，需要将它们包装在业务存档里。业务存档是Activiti引擎的部署单元，也就是一个zip文件。可以包含BPMN 2.0流程，任务表单，规则，与其他类型的文件。总的来说，业务存档包含一组已命名的资源。

当部署业务存档时，会扫描具有 `.bpmn20.xml` 或 `.bpmn` 扩展名的BPMN文件。每一个这种文件都会被解析，并可以包含多个流程定义。



业务存档中的Java类不会添加至classpath。业务存档中，所有流程定义使用的自定义类（例如Java服务任务service tasks或者事件监听器实现event listener implementations），都应该放在用于运行流程的activiti引擎的classpath下。

6.1.1. 编程方式部署 Deploying programmatically

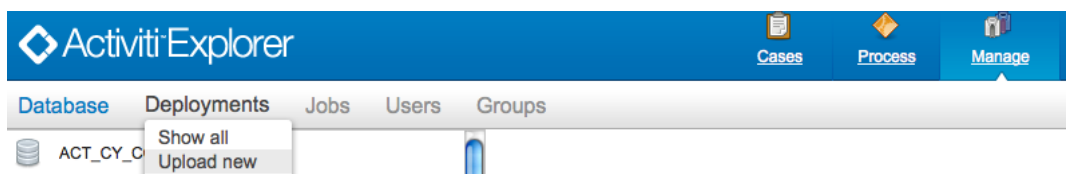
从zip文件部署业务存档，可以这样做：

```
1 String barFileName = "path/to/process-one.bar";
2 ZipInputStream inputStream = new ZipInputStream(new FileInputStream(barFileName));
3
4 repositoryService.createDeployment()
5     .name("process-one.bar")
6     .addZipInputStream(inputStream)
7     .deploy();
```

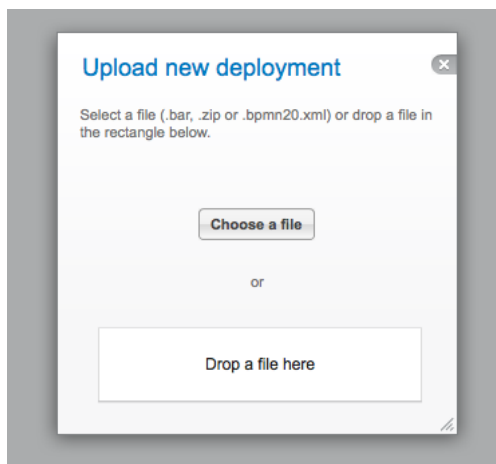
也可以从不同资源构建部署。查看javadoc获取更多信息。

6.1.2. 使用Activiti Explorer部署 (Deploying with Activiti Explorer)

Activiti Explorer web应用，可以通过web应用用户界面，上传bar文件（或者单独的 `bpmn20.xml` 文件）。选择 *Management* 页签并点击 *Deployment*：



会弹出窗口，让你选择电脑中的文件，或者（如果你的浏览器支持）可以直接拖放文件到指定区域。



6.2. 外部资源 External resources

流程定义保存在Activiti数据库中。这些流程定义，在使用服务任务、执行监听器或执行Activiti配置文件中定义的Spring bean时，可以引用委托类。这些类与Spring配置文件，需要对所有可能运行这个流程定义的流程引擎都可用。

6.2.1. Java类 Java classes

所有流程中用到的自定义类（例如服务任务、事件监听器、任务监听器等中，用到的JavaDelegate），在流程启动时，都需要存在于引擎的classpath中。

然而在业务存档部署时，classpath中不是必须要有这些类。这意味着使用Ant部署新业务存档时，你的代理类不必放在classpath中。

当使用演示配置，且希望添加自定义类时，需要在activiti-explorer或activiti-rest web应用库中，添加包含你的自定义类的jar。别忘了也要添加你的自定义类的依赖（若有）。或者，也可以将你的依赖添加到Tomcat的库文件夹，`${tomcat.home}/lib`中。

6.2.2. 在流程中使用Spring bean (Using Spring beans from a process)

当在表达式或脚本中使用Spring bean时，执行该流程定义的引擎需要可以使用这些bean。如果你自行构建web应用，并按照spring集成章节 the spring integration section的介绍，在上下文中配置流程引擎，就可以直接使用。但也请牢记在心，如果使用Activiti rest web应用，就需要更新它的上下文配置。用包含你的Spring上下文配置的 `activiti-context.xml` 文件，替换 `activiti-rest/lib/activiti-cfg.jar` jar文件中的 `activiti.cfg.xml`。

6.2.3. 创建单独应用 Creating a single app

如果不想费心保证所有流程引擎都在classpath中含有所有需要的代理类，以及保证它们都使用了正确的Spring配置，也可以考虑将Activiti rest web应用嵌入你自己的web应用，也就是说只有一个单独的 `ProcessEngine`。

6.3. 流程定义的版本 Versioning of process definitions

BPMN并没有版本的概念。这其实很好，因为可执行的BPMN流程文件很可能已经作为你的开发项目的一部分，保存在版本管理系统仓库中了（例如 Subversion，Git，或者Mercurial）。流程定义的版本在部署时创建。在部署时，Activiti会在保存至Activiti数据库前，为 `ProcessDefinition` 指定版本。

对于业务存档中的每个流程定义，下列步骤都会执行，以初始化 `key`，`version`，`name` 与 `id` 参数：

- XML文件中的流程定义 `id` 属性作为流程定义的 `key` 参数。
- XML文件中的流程定义 `name` 属性作为流程定义的 `name` 参数。如果未给定 `name` 属性，会使用 `id` 作为 `name`。
- 当每个 `key` 的流程第一次部署时，指定版本为1。对其后所有使用相同 `key` 的流程定义，部署时版本会在该 `key` 当前已部署的最高版本号基础上加1。 `key` 参数用于区分流程定义。
- `id` 参数设置为 `{processDefinitionKey}:{processDefinitionVersion}:{generated-id}`，其中 `generated-id` 是一个唯一数字，用以保证在集群环境下，流程定义缓存中，流程 `id` 的唯一性。

以下面的流程为例

```
1 <definitions id="myDefinitions" >
2   <process id="myProcess" name="My important process" >
3     ...
```

当部署这个流程定义时，数据库中的流程定义会是这个样子：

id	key	name	version
myProcess:1:676	myProcess	My important process	1

如果我们现在部署同一个流程的更新版本（例如改变部分用户任务），且保持流程定义的 `id` 不变，那么流程定义表中会包含下面的记录：

id	key	name	version
myProcess:1:676	myProcess	My important process	1
myProcess:2:870	myProcess	My important process	2

当调用 `runtimeService.startProcessInstanceByKey("myProcess")` 时，会使用版本 `2` 的流程定义，因为这是这个流程定义的最新版本。

如果再创建第二个流程，如下定义并部署至Activiti，表中会增加第三行。

```
1 <definitions id="myNewDefinitions" >
2   <process id="myNewProcess" name="My important process" >
3     ...
```

表将显示类似：

id	key	name	version
myProcess:1:676	myProcess	My important process	1
myProcess:2:870	myProcess	My important process	2
myNewProcess:1:1033	myNewProcess	My important process	1

请注意新流程的key与第一个流程的不同。即使name是相同的（我们可能本应该也改变它），Activiti也只用id属性来区分流程。因此新的流程部署时版本为1。

6.4. 提供流程图 Providing a process diagram

部署可以添加流程图图片。这个图片将存储在Activiti数据库中，并可以使用API访问。这个图片可以用在Activiti Explorer中，使流程形象化。

如果在classpath中，有一个org/activiti/expenseProcess.bpmn20.xml 流程，key为'expense'。则流程图图片会使用下列命名约定（按此顺序）：

- 如果部署中有图片资源，并且它的名字为BPMN 2.0 XML文件名，加上流程key以及图片后缀，则使用这个图片。在我们的例子中，就是org/activiti/expenseProcess.expense.png（或者.jpg/gif）。如果一个BPMN 2.0 XML文件中有多流程定义，这个方式就很合理，因为每一个流程图的文件名中都有流程key。
- 如果没有这种图片，就会寻找部署中匹配BPMN 2.0 XML文件名的图片资源。在我们的例子中，就是org/activiti/expenseProcess.png。请注意这就意味着同一个BPMN 2.0文件中的每一个流程定义，都会使用同一个流程图图片。很显然，如果每个BPMN 2.0 XML文件中只有一个流程定义，就没有问题。

用编程方式部署的例子：

```
1 repositoryService.createDeployment()
2   .name("expense-process.bar")
3   .addClasspathResource("org/activiti/expenseProcess.bpmn20.xml")
4   .addClasspathResource("org/activiti/expenseProcess.png")
5   .deploy();
```

图片资源可用下面的API获取：

```
1 ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
2   .processDefinitionKey("expense")
3   .singleResult();
4
5 String diagramResourceName = processDefinition.getDiagramResourceName();
6 InputStream imageStream = repositoryService.getResourceAsStream(
7   processDefinition.getDeploymentId(), diagramResourceName);
```

6.5. 生成流程图 Generating a process diagram

如果部署时没有按上小节介绍的提供图片，且流程定义中包含必要的“图形交换（diagram interchange）”信息，Activiti引擎会生成流程图。

可以用与部署时提供图片完全相同的方法获取图片资源。



如果由于某种原因，不需要或不希望在部署时生成流程图，可以在流程引擎配置中设置isCreateDiagramOnDeploy参数：

```
1 <property name="createDiagramOnDeploy" value="false" />
```

这样就不会生成流程图了。

6.6. 类别 Category

部署与流程定义都可以定义类别。流程定义的类别使用BPMN文件中的<definitions ... targetNamespace="yourCategory" .../>设置。

部署的类别可用API如此设定：

```
1 repositoryService
2   .createDeployment()
3   .category("yourCategory")
4   ...
5   .deploy();
```

7. BPMN 2.0介绍 BPMN 2.0 Introduction

7.1. BPMN是什么？ What is BPMN？

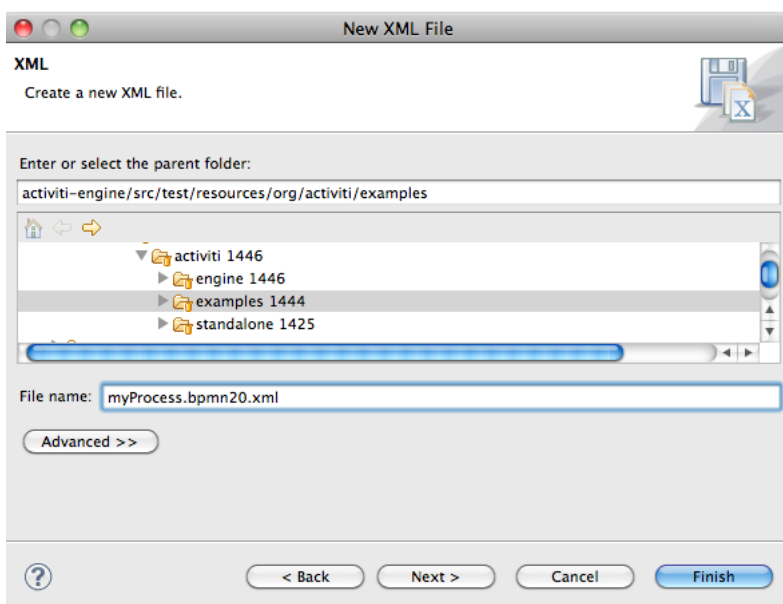
查看我们关于BPMN 2.0的FAQ条目。

7.2. 定义流程 Defining a process



这个介绍的写作，基于使用Eclipse IDE创建与编辑文件。但其实只有很少的部分使用了Eclipse的特性。可以使用你喜欢的任何其他工具创建包含BPMN 2.0的XML文件。

创建一个新的XML文件（在任意项目上右击，选择New→Other→XML-XML File）并命名。确保该文件名以.bpmn20.xml或.bpmn结尾，因为只有这样，引擎才会在部署时选择这个文件。



BPMN 2.0概要（schema）的根元素（root element）是`definitions`元素。在这个元素中，可以定义多个流程定义（然而我们建议在每个文件中，只有一个流程定义。这样可以简化已部署流程的管理）。下面显示的是一个空流程定义。请注意`definitions`元素最少需要包含`xmlns`与`targetNamespace`声明。`targetNamespace`可以为空，用于对流程定义进行分类。

```
1 <definitions
2   xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:activiti="http://activiti.org/bpmn"
4   targetNamespace="Examples">
5
6   <process id="myProcess" name="My First Process">
7     ..
8   </process>
9
10 </definitions>
```

BPMN 2.0 XML概要，除了使用Eclipse中的XML分类配置，也可以使用在线概要。

```
1 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2 xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
3   http://www.omg.org/spec/BPMN/2.0/20100501/BPMN20.xsd
```

`process`元素有两个属性：

- **id**: 必填属性，映射为Activiti `ProcessDefinition` 对象的`key`参数。可以使用`RuntimeService`中的`startProcessInstanceByKey`方法，使用`id`来启动这个流程定义的新流程实例。这个方法总会使用流程定义的最后部署版本。

```
1 ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("myProcess");
```

- 请注意这与调用 `startProcessInstanceById` 方法不同。 `startProcessInstanceById` 方法的参数为Activiti引擎在部署时生成的字符串id，可以通过调用 `processDefinition.getId()` 方法获取。生成id的格式为 `key:version`，长度限制为64字符。如果有 `ActivitiException` 显示生成id过长，请限制流程key参数（即这个id字段）的文字长度。
- **name**: 可选属性，映射为 `ProcessDefinition` 的 `name` 参数。引擎自己不会使用这个参数，可以用于例如，在用户界面上显示更用户友好的名字。

[[10minutetutorial]]

7.3. 准备：十分钟教程 Getting started: 10 minute tutorial

这个章节包含了一个（很简单的）业务流程，用于介绍一些基本的Activiti概念，以及Activiti API。

7.3.1. 先决条件 Prerequisites

这个教程假设你已经运行了Activiti演示配置，并使用独立的H2服务器。编辑 `db.properties` 并设置 `jdbc.url=jdbc:h2:tcp://localhost/activiti`，然后按照H2文档的介绍运行独立服务器。

7.3.2. 目标 Goal

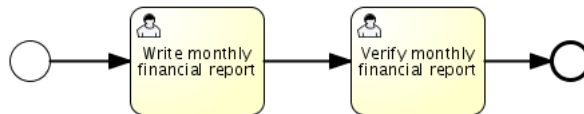
这个教程的目标是学习Activiti以及BPMN 2.0的一些基础概念。最后成果是一个简单的Java SE程序，部署了一个流程定义，并通过Activiti引擎API与流程进行交互。当然，在这个教程里学到的东西，也可以基于你的业务流程，用于构建你自己的web应用程序。

7.3.3. 用例 Use case

用例很直接：有一个公司，叫做BPMCorp。在BPMCorp中，每月需要为投资人撰写一份金融报告，由会计部门负责。在报告完成后，需要上层经理中的一位进行审核，然后才能发给所有投资人。

7.3.4. 流程图 Process diagram

上面描述的业务流程，可以使用 [Activiti Designer](#) 可视地画出。但是在这个教程里，我们自己写XML，这样可以学习更多。这个流程的图形化BPMN 2.0记录像这样：



我们看到的是一个空启动事件 **none Start Event**（左边的圆圈），接下来是两个用户任务 **User Tasks**: *'Write monthly financial report'*（撰写月度金融报告）与 *'Verify monthly financial report'*（审核月度金融报告）。最后是空结束事件 **none end event**（右边的粗线条圆圈）。

7.3.5. XML表现 XML representation

这个业务流程的XML版本（*FinancialReportProcess.bpmn20.xml*）像下面显示的一样。很容易认出流程的主要元素（点击链接可以跳转到BPMN 2.0结构的详细章节）：

- (空)开始事件 (none) start event 是流程的入口点（*entry point*）。
- 用户任务 **User Tasks** 的声明表示了流程中的人工任务。请注意第一个任务分配给 *accountancy* 组，而第二个任务分配给 *management* 组。查看用户任务分配章节 [the section on user task assignment](#) 了解关于用户与组如何分配用户任务的更多信息。
- 流程在到达空结束事件 **none end event** 时结束。
- 各元素间通过顺序流 **sequence flows** 链接。顺序流用 `source` 与 `target` 定义顺序流的流向（*direction*）。

```
1 <definitions id="definitions"
2   targetNamespace="http://activiti.org/bpmn20"
3   xmlns:activiti="http://activiti.org/bpmn"
4   xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL">
5
6   <process id="financialReport" name="Monthly financial report reminder process">
7
8     <startEvent id="theStart" />
9
10    <sequenceFlow id="flow1" sourceRef="theStart" targetRef="writeReportTask" />
11
12    <userTask id="writeReportTask" name="Write monthly financial report" >
13      <documentation>
14        Write monthly financial report for publication to shareholders.
15      </documentation>
16      <potentialOwner>
17        <resourceAssignmentExpression>
18          <formalExpression>accountancy</formalExpression>
19        </resourceAssignmentExpression>
```



```

20         </potentialOwner>
21     </userTask>
22
23     <sequenceFlow id='flow2' sourceRef='writeReportTask' targetRef='verifyReportTask'
24 />
25
26     <userTask id="verifyReportTask" name="Verify monthly financial report" >
27         <documentation>
28             Verify monthly financial report composed by the accountancy department.
29             This financial report is going to be sent to all the company shareholders.
30         </documentation>
31         <potentialOwner>
32             <resourceAssignmentExpression>
33                 <formalExpression>management</formalExpression>
34             </resourceAssignmentExpression>
35         </potentialOwner>
36     </userTask>
37
38     <sequenceFlow id='flow3' sourceRef='verifyReportTask' targetRef='theEnd' />
39
40     <endEvent id="theEnd" />
41
42 </process>
43
</definitions>

```

7.3.6. 启动流程实例 Starting a process instance

现在我们已经创建了业务流程的流程定义。使用这样的流程定义，可以创建流程实例。在这个例子中，一个流程实例将对应一个特定月份的一次财经报告创建与审核工作。所有流程实例共享相同的流程定义。

要用给定的流程定义创建流程实例，需要首先部署（**deploy**）流程定义。部署流程定义意味着两件事：

- 流程定义将会存储在**Activiti**引擎配置的持久化数据库中。因此通过部署业务流程，保证了引擎在重启后也能找到流程定义。
- BPMN 2.0流程文件会解析为内存中的对象模型。这个模型可以通过**Activiti API**操纵。

更多关于部署的信息可以在[部署专门章节](#)中找到。

与该章节的描述一样，部署有很多种方式。一种是通过下面展示的API。请注意所有与**Activiti**引擎的交互都要通过它的服务（**services**）。

```

1 Deployment deployment = repositoryService.createDeployment()
2   .addClasspathResource("FinancialReportProcess.bpmn20.xml")
3   .deploy();

```

现在可以使用在流程定义中定义的**id**（参见XML文件中的流程元素）启动新流程实例。请注意这个**id**在**Activiti**术语中被称作**key**。

```

1 ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("financialReport");

```

这会创建流程实例，并首先通过开始事件。在开始事件后，会沿着所有出口顺序流（在这个例子中只有一个）继续，并到达第一个任务（撰写月度金融报告 *write monthly financial report*）。这时，**Activiti**引擎会在持久化数据库中存储一个任务。同时，会解析这个任务附加的分配用户或组，也保存在数据库中。请注意，**Activiti**引擎会持续执行流程步骤，直到到达等待状态 *wait state*，例如用户任务。在这种等待状态时，流程实例的当前状态会存储在数据库中，并保持这个状态，直到用户决定完成任务。这时，引擎会继续执行，直到遇到新的等待状态，或者流程结束。如果在这期间引擎重启或崩溃，流程的状态也仍在数据库中安全并妥善的保存。

在任务创建后，**startProcessInstanceByKey**方法会返回，因为用户任务活动是一个等待状态。在这个例子里，这个任务分配给一个组。这意味着这个组的每一个成员都是处理这个任务的候选人 **candidate**。

现在可以将这些整合起来，创建一个简单的Java程序。创建一个新的Eclipse项目，在它的classpath中添加**Activiti jar**与依赖（可以在**Activiti**发行版的**libs**目录下找到）。在能够调用**Activiti**服务前，需要首先构建**ProcessEngine**（流程引擎），用于访问服务。这里我们使用**standalone**独立配置，这个配置会构建**ProcessEngine**，并使用与演示配置中相同的数据库。

可以从[这里](#)下载流程定义XML。这个文件包含了上面展示的XML，同时包含了必要的BPMN图形交互信息 **diagram interchange information**，用于在**Activiti**的工具中可视化展示流程。

```

1 public static void main(String[] args) {
2
3     // 创建Activiti流程引擎 Create Activiti process engine
4     ProcessEngine processEngine = ProcessEngineConfiguration
5         .createStandaloneProcessEngineConfiguration()
6         .buildProcessEngine();
7
8     // 获取Activiti服务 Get Activiti services
9     RepositoryService repositoryService = processEngine.getRepositoryService();
10    RuntimeService runtimeService = processEngine.getRuntimeService();
11

```

```

12 // 部署流程定义 Deploy the process definition
13 repositoryService.createDeployment()
14     .addClasspathResource("FinancialReportProcess.bpmn20.xml")
15     .deploy();
16
17 // 启动流程实例 Start a process instance
18 runtimeService.startProcessInstanceByKey("financialReport");
19 }

```

7.3.7. 任务列表 Task lists

现在可以通过添加下列逻辑，获取这个任务：

```

1 List<Task> tasks = taskService.createTaskQuery().taskCandidateUser("kermit").list();

```

请注意传递给这个操作的用户需要是 **accountancy** 组的成员，因为在流程定义中是这么声明的：

```

1 <potentialOwner>
2   <resourceAssignmentExpression>
3     <formalExpression>accountancy</formalExpression>
4   </resourceAssignmentExpression>
5 </potentialOwner>

```

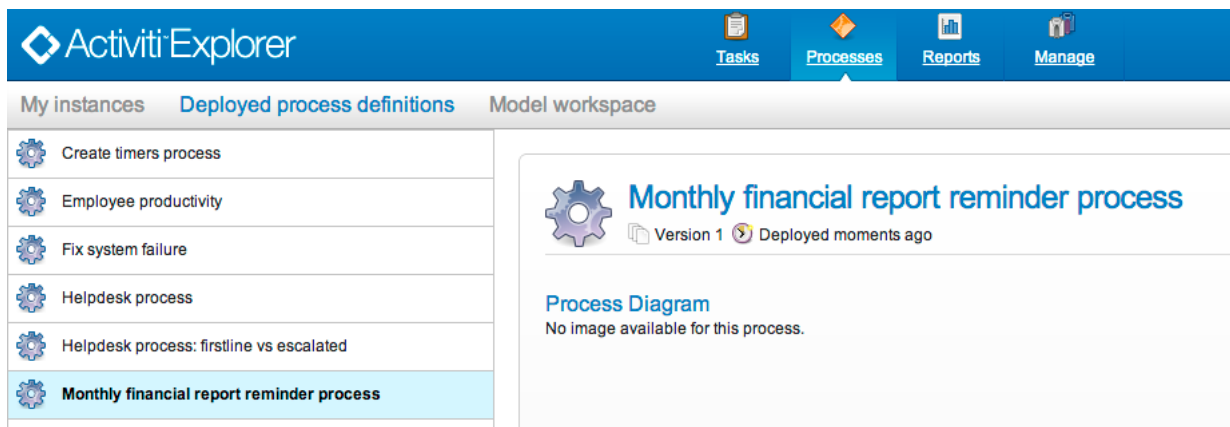
也可以使用任务查询API，用组名查得相同结果。可以在代码中添加下列逻辑：

```

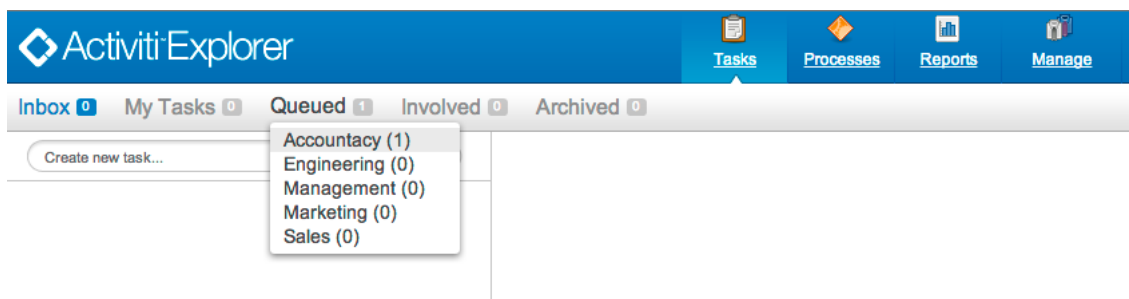
1 TaskService taskService = processEngine.getTaskService();
2 List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("accountancy").list();

```

因为我们将 **ProcessEngine** 配置为使用与演示配置中相同的数据库，因此可以登录 **Activiti Explorer**。默认情况下，**accountancy** 组中没有用户。使用 **kermit/kermit** 登录，点击 **Groups**，然后 "Create group (创建组)"。然后点击 **Users**，并向组中添加 **fizzie**。现在使用 **fizzie/fizzie** 登录，就会发现在选择了 **Processes** 页面，点击 '**Monthly financial report**（月度金融报告）' 的 '**Actions**' 栏的 '**Start Process**（开始流程）' 链接后，可以启动我们的业务流程。



前面已经解释过，流程会执行到第一个用户任务。因为使用 **Fizzie** 登录，就可以看到在启动流程实例后，他有一个新的候选任务（candidate task）。选择 **Tasks** 页面来查看这个新任务。请注意即使流程是由其他人启动的，**accountancy** 组中的每一个人仍然都能看到这个候选任务。



7.3.8. 申领任务 Claiming the task

会计师（**accountancy** 组的成员）现在需要申领任务。申领任务后，这个用户会成为任务的执行人（**assignee**），这个任务也会从 **accountancy** 组的其他成员的任务列表中消失。申领任务通过编程方式如下实现：

```

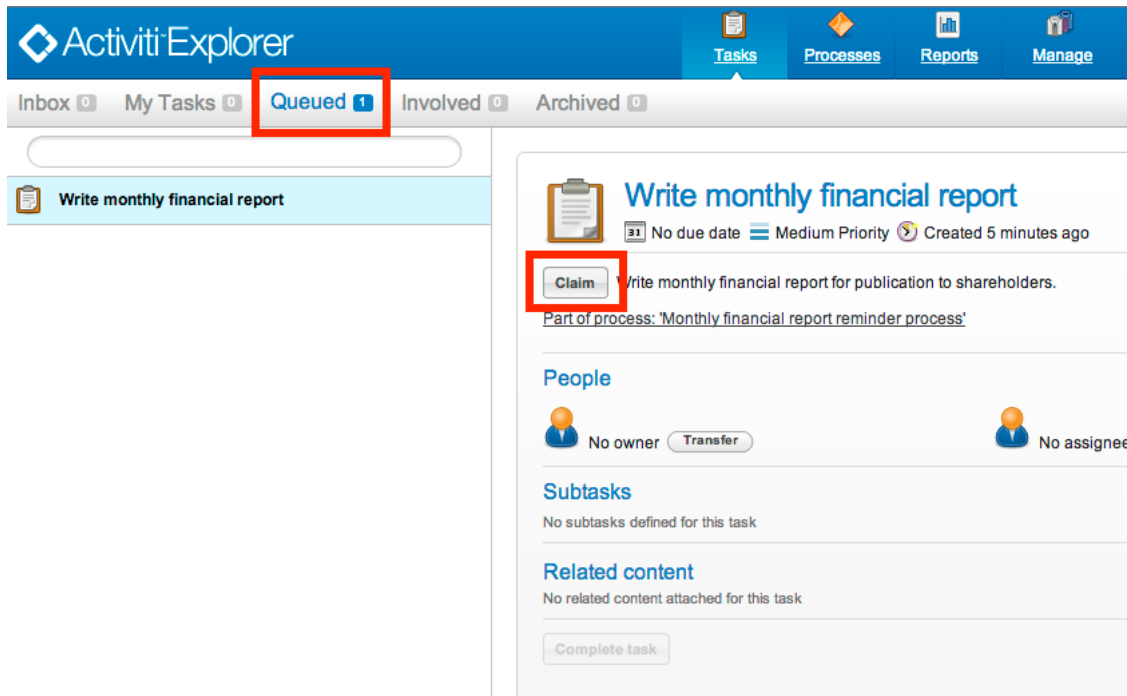
1 taskService.claim(task.getId(), "fizzie");

```

这个任务现在在申领任务者的个人任务列表中。

```
1 | List<Task> tasks = taskService.createTaskQuery().taskAssignee("fozzie").list();
```

在Activiti Explorer UI中，点击`claim`按钮会执行相同操作。这个任务会转移到登录用户的个人任务列表中。也可以看到任务执行人变更为当前登录用户。



7.3.9. 完成任务 Completing the task

会计师（`accountancy`组的成员）现在需要开始撰写金融报告了。一旦报告完成，他就可以完成任务。这意味着这个任务的所有工作都已经完成。

```
1 | taskService.complete(task.getId());
```

对于Activiti引擎来说，这是个外部信号，指示流程实例需要继续执行。任务本身会从运行时数据中移除，并继续这个任务唯一的出口转移（`outgoing transition`），将执行移至第二个任务（`'verification of the report'` 审核月度报告）。与上面介绍的第一个任务使用的相同的机制，会用于为第二个任务分配执行人。有一点小区别，这个任务会分配给`management`组。

在演示设置中，完成任务可以通过点击任务列表中的`complete`按钮。因为Fozzie不是经理，我们需要登出Activiti Explorer，并用`kermi`（他是经理）登录。第二个任务现在可以在未分配任务列表中看到。

7.3.10. 结束流程 Ending the process

与之前完全相同的方式，可以获取并申领审核任务。完成这个第二个任务，会将流程执行移至结束事件，并结束流程实例。这个流程实例与所有相关的运行时执行数据都会从数据库中移除。

登录至Activiti Explorer可以验证这一点，流程执行的存储表中找不到记录。

Tasks

Processes

Reports

Manage

Database

Deployments

Active Processes

Suspended Processes

Jobs

Users

Groups

Administration

ACT_ID_MEMBERSHIP (14)

ACT_ID_USER (3)

ACT_RE_DEPLOYMENT (3)

ACT_RE_MODEL (1)

ACT_RE_PROCDEF (11)

ACT_RU_EVENT_SUBSCR (0)

ACT_RU_EXECUTION (0)

ACT_RU_IDENTITYLINK (0)

ACT_RU_JOB (0)

ACT_RU_TASK (0)

ACT_RU_VARIABLE (0)

ACT_RU_EXECUTION

Table contains no rows.

也可以通过编程方式，使用 `historyService` 验证流程已经结束

```

1 HistoryService historyService = processEngine.getHistoryService();
2 HistoricProcessInstance historicProcessInstance =
3 historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
4 System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());

```

7.3.11. 代码总结 Code overview

将之前章节的所有代码片段整合起来，会得到类似这样的代码（这段代码考虑到了你可能已经使用Activiti Explorer UI启动了一些流程实例。代码中总是获取任务列表而不是一个任务，因此总能执行）：

```

1 public class TenMinuteTutorial {
2
3     public static void main(String[] args) {
4
5         // 创建Activiti流程引擎 Create Activiti process engine
6         ProcessEngine processEngine = ProcessEngineConfiguration
7             .createStandaloneProcessEngineConfiguration()
8             .buildProcessEngine();
9
10        // 获取Activiti服务 Get Activiti services
11        RepositoryService repositoryService = processEngine.getRepositoryService();
12        RuntimeService runtimeService = processEngine.getRuntimeService();
13
14        // 部署流程定义 Deploy the process definition
15        repositoryService.createDeployment()
16            .addClasspathResource("FinancialReportProcess.bpmn20.xml")
17            .deploy();
18
19        // 启动流程实例 Start a process instance
20        String procId = runtimeService.startProcessInstanceByKey("financialReport").getId();
21
22        // 获取第一个任务 Get the first task
23        TaskService taskService = processEngine.getTaskService();
24        List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("accountancy").list();
25        for (Task task : tasks) {
26            System.out.println("Following task is available for accountancy group: " + task.getName());
27
28            // 申领 claim it
29            taskService.claim(task.getId(), "fozzie");
30        }
31
32        // 验证Fozzie获取了任务 Verify Fozzie can now retrieve the task
33        tasks = taskService.createTaskQuery().taskAssignee("fozzie").list();
34        for (Task task : tasks) {
35            System.out.println("Task for fozzie: " + task.getName());
36
37            // 完成任务 Complete the task
38            taskService.complete(task.getId());
39        }
40
41        System.out.println("Number of tasks for fozzie: "
42            + taskService.createTaskQuery().taskAssignee("fozzie").count());
43
44        // 获取并申领第二个任务 Retrieve and claim the second task
45        tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();

```

```

46     for (Task task : tasks) {
47         System.out.println("Following task is available for management group: " + task.getName());
48         taskService.claim(task.getId(), "kermit");
49     }
50
51     // 完成第二个任务并结束流程 Completing the second task ends the process
52     for (Task task : tasks) {
53         taskService.complete(task.getId());
54     }
55
56     // 验证流程已经结束 verify that the process is actually finished
57     HistoryService historyService = processEngine.getHistoryService();
58     HistoricProcessInstance historicProcessInstance =
59         historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
60     System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());
61 }
62
63 }

```

7.3.12. 继续提高 Future enhancements

可以看出这个业务流程太简单了，不能实际使用。然而，随着继续浏览Activiti中可用的BPMN 2.0结构，可以增强业务流程通过：

- 定义网关 **gateway** 执行选择。这样，经理可以驳回金融报告，并重新为会计师创建任务。
- 定义并使用变量 **variables**。这样可以存储或引用报告，并可以在表单中显示它。
- 在流程结束处定义服务任务 **service task**，将报告发送给每一个投资人。
- 等等。

8. BPMN 2.0 结构 BPMN 2.0 Constructs

本章节包含了Activiti支持的BPMN 2.0结构，以及对BPMN标准的自定义扩展。

8.1. 自定义扩展 Custom extensions

BPMN 2.0标准对流程的所有的参与者都是个好东西。最终用户不需要因为依赖专利解决方案，而被供应商“绑架”。Activiti之类的开源框架，也可以提供与大型供应商的解决方案相同（经常是更好;-）的实现。有了BPMN 2.0标准，从大型供应商解决方案向Activiti的转变，就变得简单平滑。

然而标准的缺点，是它通常是不同公司（不同观点）大量讨论与妥协的结果。作为阅读BPMN 2.0 XML流程定义的开发者，有时会觉得某些结构或做事方法太笨重了。Activiti将开发者的感受放在最高优先，因此引入了一些**Activiti BPMN extensions**（扩展）。这些“扩展”并不在BPMN 2.0规格中，有些是新结构，有些是对特定结构的简化。

尽管BPMN 2.0规格明确指出可以支持自定义扩展，我们仍做了如下保证：

- 自定义扩展的前提是，做事情的标准方式总可以进行更简化的改造。因此当你决定使用自定义扩展时，不用担心无路可退（仍然可以用标准方式）。
- 使用自定义扩展时，总是通过为新的XML元素、属性等提供**activiti:**命名空间前缀，明确标识出来。

因此是否使用自定义扩展，完全取决于你自己。有些其他因素会影响选择（图形化编辑器的使用，公司策略，等等）。我们提供扩展，只是因为相信，标准中的某些地方可以用更简单或效率更高的方式处理。请不要吝啬给我们反馈对扩展的评价（正面的和/或负面的），也可以给我们提供关于自定义扩展的新想法。说不定某一天，你的想法会成为规范的一部分！

8.2. 事件 Events

事件通常用于为流程生命周期中发生的事情建模。事件总是图形化为圆圈。在BPMN 2.0中，有两种主要的事件分类：捕获（**catching**）与抛出（**throwing**）事件。

- 捕获：当流程执行到达这个事件时，会等待直到触发器动作。触发器的类型，由其中的图标，或者说XML中的类型声明而定义。捕获事件与抛出事件显示上的区别，是其内部的图标没有填充（也就是说，是白色的）。
- 抛出：当流程执行到达这个事件时，会触发一个触发器。触发器的类型，由其中的图标，或者说XML中的类型声明而定义。抛出事件与捕获事件显示上的区别，是其内部的图标填充为黑色。

8.2.1. 事件定义 Event Definitions

事件定义，定义了事件的语义。没有事件定义的话，事件就“不做什么特别的事情”。例如一个没有事件定义的开始事件，并不限定具体是什么启动了流程。如果为这个开始事件添加事件定义（例如定时器事件定义），就声明了启动流程的“类型”（例如对于定时器事件定义，就是到达了特定的时间点）。

8.2.2. 定时器事件定义 Timer Event Definitions

定时器事件，是由定义的定时器触发的事件。可以用于**开始事件 start event**，**中间事件 intermediate event**，或**边界事件 boundary event**。定时器事件的行为，取决于所使用的业务日历（**business calendar**）。定时器事件有默认的业务日历，但也可以为每个定时器事件定义，定义业务日历。

```

1 <timerEventDefinition activiti:businessCalendarName="custom">
2   ...
3 </timerEventDefinition>

```

其中businessCalendarName指向流程引擎配置中的业务日历。如果省略业务日历定义，就使用默认业务日历。

定时器定义必须且只能使用下列的一种元素：

- **timeDate**。这个方式指定了ISO 8601格式的固定时间。在这个时间点，会触发触发器。例如：

```

1 <timerEventDefinition>
2   <timeDate>2011-03-11T12:13:14</timeDate>
3 </timerEventDefinition>

```

- **timeDuration**。要定义在触发前，定时器需要等待多长时间，可以用timeDuration作为timerEventDefinition的子元素来指定。使用ISO 8601格式（BPMN 2.0规范要求）。例如（等待10天）：

```

1 <timerEventDefinition>
2   <timeDuration>P10D</timeDuration>
3 </timerEventDefinition>

```

- **timeCycle**。指定重复周期，可用于周期性启动流程，或者为超期用户任务多次发送提醒。这个元素可以使用两种格式。第一种是按照ISO 8601标准定义的循环时间周期。例如（三次重复间隔，每次间隔为10小时）：

```

1 <timerEventDefinition>
2   <timeCycle activiti:endDate="2015-02-25T16:42:11+00:00">R3/PT10H</timeCycle>
3 </timerEventDefinition>

```

也可以指定endDate，作为timeCycle的可选属性，或者像这样直接写在时间表达式的结尾：**R3/PT10H/\${EndDate}**。当到达endDate时，应用会停止，并为该任务创建其他作业。可以使用ISO 8601标准的静态值，比如"2015-02-25T16:42:11+00:00"。也可以使用变量\${EndDate}

```

1 <timerEventDefinition>
2   <timeCycle>R3/PT10H/${EndDate}</timeCycle>
3 </timerEventDefinition>

```

如果同时使用了两种指定方式，则系统会使用属性方式定义的endDate。

目前只有BoundaryTimerEvents与CatchTimerEvent支持EndDate功能。

另外，也可以使用cron表达式指定定时周期。下面的例子展示了一个整点启动，每5分钟触发的触发器：

```
0 0/5 * * * ?
```

请参考[这个教程](#)了解如何使用cron表达式。

请注意：与普通的Unix cron不同，第一个符号代表的是秒。

重复时间周期更适用于使用相对时间，也就是从某个特定时间点开始计算（比如用户任务开始的时间）。而cron表达式可以使用绝对时间，因此绝对适合用于[定时启动事件 timer start events](#)。

可以在定时事件定义中使用表达式，也就是使用流程变量调整定时器定义。这个流程变量必须是包含合适时间格式的字符串，ISO 8601（或者对于循环类型，cron）。

```

1 <boundaryEvent id="escalationTimer" cancelActivity="true" attachedToRef="firstLineSupport">
2   <timerEventDefinition>
3     <timeDuration>${duration}</timeDuration>
4   </timerEventDefinition>
5 </boundaryEvent>

```

请注意：定时器只有在作业或者异步执行器启用时才能触发（也就是说，需要在activiti.cfg.xml中，将jobExecutorActivate或者asyncExecutorActivate设置为true。因为默认情况下，作业与异步执行器都是禁用的）。

8.2.3. 错误事件定义 Error Event Definitions

重要提示：BPMN错误与Java异常不是一回事。事实上，这两者毫无共同点。BPMN错误事件是建模业务异常（business exceptions）的方式。而Java异常使用它们自己的方式处理。

```
<endEvent id="myErrorEndEvent">
```



```

1   <errorEventDefinition errorRef="myError" />
2   </endEvent>
3

```

8.2.4. 信号事件定义 Signal Event Definitions

信号事件，是引用具名信号的事件。信号是全局范围（广播）的事件，并会被传递给所有激活的处理器（等待中的流程实例/捕获信号事件 catching signal events）。

信号事件定义使用 `signalEventDefinition` 元素声明。其 `signalRef` 属性引用一个 `signal` 元素，该 `signal` 元素需要声明为 `definitions` 根元素的子元素。下面摘录一个流程，使用中间事件（intermediate event）抛出与捕获信号事件。

```

1   <definitions... >
2       <!-- 声明信号 -->
3       <signal id="alertSignal" name="alert" />
4
5       <process id="catchSignal">
6           <intermediateThrowEvent id="throwSignalEvent" name="Alert">
7               <!-- 信号事件定义 -->
8               <signalEventDefinition signalRef="alertSignal" />
9           </intermediateThrowEvent>
10          ...
11          <intermediateCatchEvent id="catchSignalEvent" name="On Alert">
12              <!-- 信号事件定义 -->
13              <signalEventDefinition signalRef="alertSignal" />
14          </intermediateCatchEvent>
15          ...
16      </process>
17  </definitions>

```

两个 `signalEventDefinition` 引用同一个 `signal` 元素。

抛出信号事件 Throwing a Signal Event

信号可以由流程实例使用BPMN结构抛出，也可以通过编程方式使用Java API抛出。下面 `org.activiti.engine.RuntimeService` 中的方法可以用于编程方式抛出信号：

```

1   RuntimeService.signalEventReceived(String signalName);
2   RuntimeService.signalEventReceived(String signalName, String executionId);

```

`signalEventReceived(String signalName);` 与 `signalEventReceived(String signalName, String executionId);` 的区别，是前者在全局范围，为所有已订阅处理器抛出信号（广播），而后者只为指定的执行传递信号。

捕获信号事件 Catching a Signal Event

信号事件可用信号捕获中间事件（intermediate catch signal event）或者信号边界事件（signal boundary event）捕获。

查询信号事件订阅 Querying for Signal Event subscriptions

可以查询订阅了某一信号事件的所有执行：

```

1   List<Execution> executions = runtimeService.createExecutionQuery()
2       .signalEventSubscriptionName("alert")
3       .list();

```

可以使用 `signalEventReceived(String signalName, String executionId)` 方法为这些执行传递这个信号。

信号事件范围 Signal event scope

默认情况下，信号事件在流程引擎全局广播。这意味着你可以在一个流程实例中抛出一个信号事件，而不同流程定义的不同流程实例都会响应这个事件。

然而，有时也会希望只在同一个流程实例中响应信号事件。例如在流程实例中使用异步机制，而两个或多个活动彼此互斥的时候。

要限制信号事件的范围（scope），在信号事件定义中添加（非BPMN 2.0标准！）`scope`属性：

```

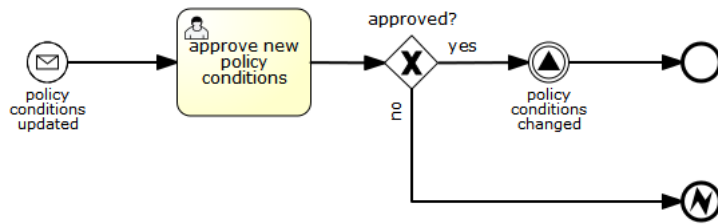
1   <signal id="alertSignal" name="alert" activiti:scope="processInstance"/>

```

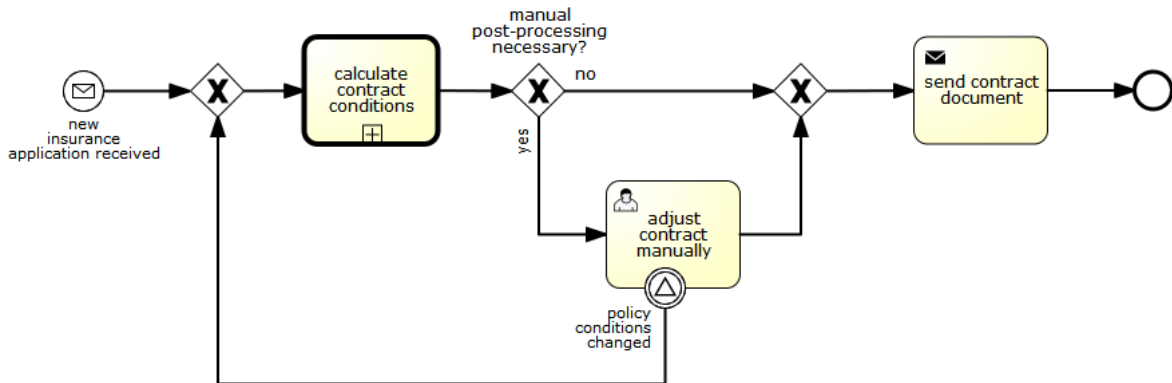
这个属性的默认值为“*global*（全局）”。

信号事件示例 Signal Event example(s)

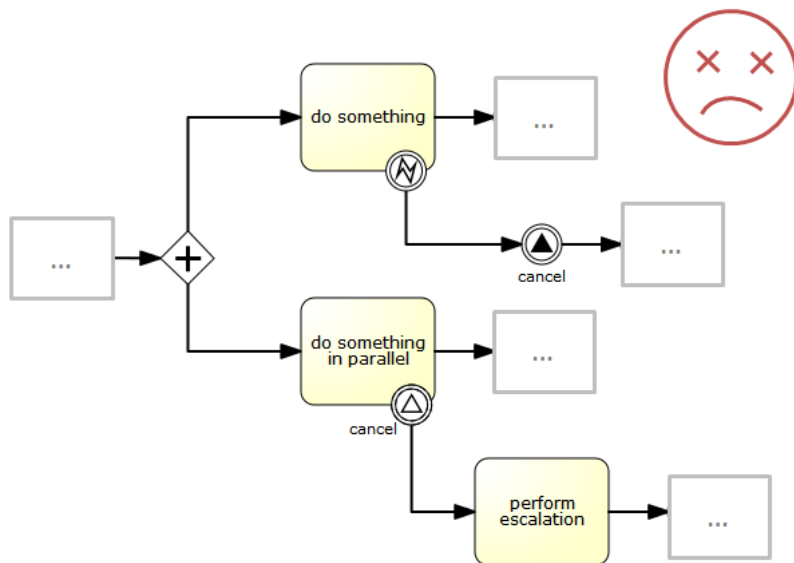
下面是一个关于两个不同的流程通过信号通信的例子。第一个流程在保险政策更新或变更时启动。在变更由人工审核之后，会抛出信号事件，指出政策已经发生了变更：



这个事件可以被所有感兴趣的流程实例捕获。下面是一个订阅这个事件的流程的例子。



请注意：要理解信号事件会广播给所有激活的处理器，这很重要。这意味着在上面的例子中，所有捕获这个信号的流程实例，都会接收这个信号。在这个例子中这就是我们期望的。然而，有的情况下，不希望使用广播方式。考虑下面的流程：



Activiti不支持上面流程中描述的模式。理想情况是，在执行“do something”任务时抛出的错误，由错误边界事件捕获，并通过信号抛出事件传播至执行的并行分支，最终中断“do something in parallel”任务。到目前为止Activiti会按照预期效果执行。然而，由于信号的广播效应，它也会被传播至所有其他订阅了这个信号事件的流程实例。这可能并非我们希望的效果。

请注意：信号事件与特定的流程实例无关，而是会广播给所有流程实例。如果你需要只为某一特定的流程实例传递信号，则需要使用 `signalEventReceived(String signalName, String executionId)` 手动建立关联，并使用适当的查询机制 [query mechanisms](#)。

8.2.5. 消息事件定义 Message Event Definitions

消息事件，是指引用具名消息的事件。消息具有名字与载荷。与信号不同，消息事件只有一个接收者。

消息事件定义使用 `messageEventDefinition` 元素声明。其 `messageRef` 属性引用一个 `message` 元素，该 `message` 元素需要声明为 `definitions` 根元素的子元素。下面摘录一个流程，声明了两个消息事件，并由开始事件与消息捕获中间事件（intermediate catching message event）引用。

```

1 <definitions id="definitions"
2   xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:activiti="http://activiti.org/bpmn"
4   targetNamespace="Examples"
5   xmlns:tns="Examples">
6
7   <message id="newInvoice" name="newInvoiceMessage" />

```

```

8      <message id="payment" name="paymentMessage" />
9
10     <process id="invoiceProcess">
11
12         <startEvent id="messageStart" >
13             <messageEventDefinition messageRef="newInvoice" />
14         </startEvent>
15         ...
16         <intermediateCatchEvent id="paymentEvt" >
17             <messageEventDefinition messageRef="payment" />
18         </intermediateCatchEvent>
19         ...
20     </process>
21
22 </definitions>

```

抛出消息事件 Throwing a Message Event

作为可嵌入的流程引擎，Activiti不关心实际接收消息。因为这可能与环境相关，并进行平台定义的操作，例如连接至JMS（Java Messaging Service, Java消息服务）队列（Queue）/主题（Topic），或者处理Webservice或者REST请求。因此接收消息需要作为应用的一部分，或者是流程引擎所嵌入的基础框架中的一部分，由你自行实现。

在应用中接收到消息后，需要决定如何处理它。如果这个消息需要启动新的流程实例，可以选择下面由runtime服务提供的方法中的一种：

```

1 ProcessInstance startProcessInstanceByMessage(String messageName);
2 ProcessInstance startProcessInstanceByMessage(String messageName, Map<String, Object> processVariables);
3 ProcessInstance startProcessInstanceByMessage(String messageName, String businessKey, Map<String, Object>
  processVariables);

```

这些方法可以使用引用的消息启动流程实例。

如果需要由已有的流程实例接收消息，需要首先将消息与特定的流程实例关联（查看后续章节），然后触发等待中的执行，让其继续。runtime服务提供了下列方法，根据消息事件的订阅，触发执行：

```

1 void messageEventReceived(String messageName, String executionId);
2 void messageEventReceived(String messageName, String executionId, HashMap<String, Object> processVariables);

```

查询消息事件订阅 Querying for Message Event subscriptions

- 对于消息启动事件，消息事件的订阅与特定的流程定义相关。这种类型的消息订阅，可以使用 **ProcessDefinitionQuery** 查询：

```

1 ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
2     .messageEventSubscription("newCallCenterBooking")
3     .singleResult();

```

因为对于一个消息，只能有一个流程定义订阅，因此这个查询总是返回0或1个结果。如果流程定义更新了，只有该流程定义的最新版本会订阅这个消息事件。

- 对于消息捕获中间事件（intermediate catch message event），消息事件的订阅与特定的执行相关。这种类型的消息订阅，可以使用 **ExecutionQuery** 查询：

```

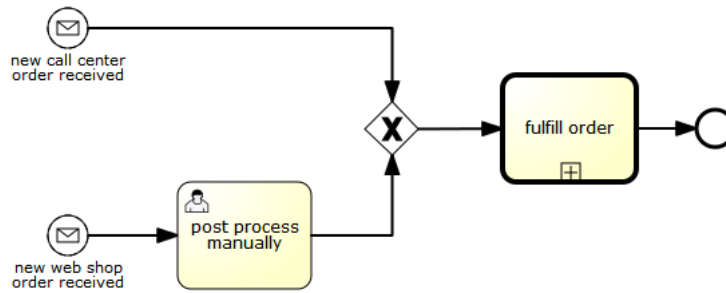
1 Execution execution = runtimeService.createExecutionQuery()
2     .messageEventSubscriptionName("paymentReceived")
3     .variableValueEquals("orderId", message.getOrderId())
4     .singleResult();

```

这种查询通常有关联查询，并且通常需要了解流程（在这个例子里，对于给定的orderId，至多只有一个流程实例）。

消息事件示例 Message Event example(s)

下面是一个流程的例子，可以使用两种不同的消息启动：



在流程需要通过不同的方式响应不同的启动事件，但是后续使用统一的方式处理时，这就很有用。

8.2.6. 启动事件 Start Events

启动事件指明了流程的起点。启动事件的类型（流程在消息到达时启动，在指定的时间间隔后启动，等等），定义了流程如何启动，并显示为启动事件中的小图标。在XML中，类型由子元素声明来定义。

启动事件“随时捕获”：概念上，事件（随时）等候，直到特定的触发器被触发。

在启动事件中，可以使用下列Activiti专用参数：

- **initiator**: 指明保存认证用户（authenticated user）id用的变量名。在流程启动时，该id会使用这个变量名被保存。例如：

```
1 <startEvent id="request" activiti:initiator="initiator" />
```

认证用户必须通过 `IdentityService.setAuthenticatedUserId(String)` 方法，在try-finally块中设置，像这样：

```
1 try {
2     identityService.setAuthenticatedUserId("bono");
3     runtimeService.startProcessInstanceByKey("someProcessKey");
4 } finally {
5     identityService.setAuthenticatedUserId(null);
6 }
```

这段代码在集成在Activiti Explorer应用中。因此可以与[表单](#)一起使用。

8.2.7. 空启动事件 None Start Event

描述 Description

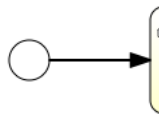
“空”启动事件，技术上指的是没有特别指定启动流程实例的触发器。这意味着引擎无法预知何时启动流程实例。空启动事件用于流程实例通过调用下列`startProcessInstanceByXXX` API方法启动的情况。

```
1 ProcessInstance processInstance = runtimeService.startProcessInstanceByXXX();
```

请注意：子流程（subprocess）总是有空启动事件。

图示 Graphical notation

空启动事件用空心圆圈表示，中间没有图标（也就是说没有触发器）。



XML表示 XML representation

空启动事件的XML表示格式，就是普通的启动事件声明，而没有任何子元素（其他种类的启动事件都有子元素，用于声明其类型）。

```
1 <startEvent id="start" name="my start event" />
```

空启动事件的自定义扩展 Custom extensions for the none start event

formKey: 引用表单模板，用户需要在启动新流程实例时填写该表单。可以在[表单](#)章节找到更多信息。例如：

```
1 <startEvent id="request" activiti:formKey="org/activiti/examples/taskforms/request.form" />
```

8.2.8. 定时器启动事件 Timer Start Event

描述 Description

定时器启动事件，用于在指定时间创建流程实例。在流程只需要启动一次，或者流程需要在特定的时间间隔重复启动时，都可以使用。

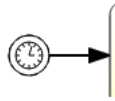
请注意：子流程不能有定时器启动事件。

请注意：定时器启动事件，在流程部署的同时就开始计时。不需要调用`startProcessInstanceByXXX`，尽管也不禁止使用启动流程的方法。调用`startProcessInstanceByXXX`时也会启动流程。

请注意：当部署带有定时器启动事件的流程的新版本时，上一版本的定时器作业会被移除。这是因为通常并不希望旧版本的流程仍然自动启动新的流程实例。

图示 Graphical notation

定时器启动事件，用其中有一个钟表图标的圆圈来表示。



XML表示 XML representation

定时器启动事件的XML表示格式，是普通的启动事件声明，加上定时器定义子元素。请参考[定时器定义](#)了解详细配置方法。

示例：流程会启动4次，间隔5分钟，从2011年3月11日，12:13开始

```
1 <startEvent id="theStart">
2   <timerEventDefinition>
3     <timeCycle>R4/2011-03-11T12:13/PT5M</timeCycle>
4   </timerEventDefinition>
5 </startEvent>
```

示例：流程会在选定的时间启动一次

```
1 <startEvent id="theStart">
2   <timerEventDefinition>
3     <timeDate>2011-03-11T12:13:14</timeDate>
4   </timerEventDefinition>
5 </startEvent>
```

8.2.9. 消息启动事件 Message Start Event

描述 Description

消息启动事件，使用具名消息启动流程实例。它让我们可以使用消息名，有效地在一组可选的启动事件中选择正确的启动事件。

当部署具有一个或多个消息启动事件的流程定义时，会考虑下列因素：

- 消息启动事件的名字，在给定流程定义中，必须是唯一的。一个流程定义不得包含多个同名的消息启动事件。如果流程定义中有两个或多个消息启动事件引用 同一个消息，也即两个或多个消息启动事件引用了具有相同消息名字的消息，则Activiti在部署这个流程定义时，会抛出异常。
- 消息启动事件的名字，在所有已部署的流程定义中，必须是唯一的。如果流程定义中，一个或多个消息启动事件，引用了已经部署的另一流程定义中消息启动事件的消息名，则Activiti在部署这个流程定义时，会抛出异常。
- 流程版本：在部署流程定义的新版本时，会取消上一版本的消息订阅。即使新版本中并没有这个消息事件，仍然如此（取消上版本的消息订阅）。

当启动流程实例时，可以使用下列 `RuntimeService` 中的方法，触发消息启动事件：

```
1 ProcessInstance startProcessInstanceByMessage(String messageName);
2 ProcessInstance startProcessInstanceByMessage(String messageName, Map<String, Object> processVariables);
3 ProcessInstance startProcessInstanceByMessage(String messageName, String businessKey, Map<String, Object> processVariables);
```

`messageName` 是由 `message` 元素的 `name` 属性决定的名字。`message` 元素被 `messageEventDefinition` 的 `messageRef` 属性引用。当启动流程实例时，请考虑下列因素：

- 只有顶层流程（top-level process）才支持消息启动事件。嵌入式子流程不支持消息启动事件。
- 如果一个流程定义中有多个消息启动事件，`runtimeService.startProcessInstanceByMessage(...)` 允许选择合适的启动事件。
- 如果一个流程定义中有多个消息启动事件，与一个空启动事件，则 `runtimeService.startProcessInstanceByKey(...)` 与 `runtimeService.startProcessInstanceById(...)` 会使用空启动事件启动流程实例。

- 如果一个流程定义中有多个消息启动事件，而没有空启动事件，则 `runtimeService.startProcessInstanceByKey(...)` 与 `runtimeService.startProcessInstanceById(...)` 会抛出异常。
- 如果一个流程定义中只有一个消息启动事件，则 `runtimeService.startProcessInstanceByKey(...)` 与 `runtimeService.startProcessInstanceById(...)` 会使用这个消息启动事件启动新流程实例。
- 如果流程由调用活动（call activity）启动，则消息启动事件只有在下列情况下才被支持
 - 除了消息启动事件，流程还有唯一的空启动事件
 - 或者流程只有唯一的消息启动事件，而没有其他启动事件。

图示 Graphical notation

消息启动事件，用其中有一个消息事件标志的圆圈表示。这个标志并未填充，用以表示捕获（接收）行为。



XML表示 XML representation

消息启动事件的XML表示格式，为普通启动事件声明，加上messageEventDefinition子元素：

```

1 <definitions id="definitions"
2   xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:activiti="http://activiti.org/bpmn"
4   targetNamespace="Examples"
5   xmlns:tns="Examples">
6
7   <message id="newInvoice" name="newInvoiceMessage" />
8
9   <process id="invoiceProcess">
10
11     <startEvent id="messageStart" >
12       <messageEventDefinition messageRef="tns:newInvoice" />
13     </startEvent>
14     ...
15   </process>
16
17 </definitions>

```

8.2.10. 信号启动事件 Signal Start Event

描述 Description

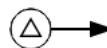
信号启动事件，使用具名信号启动流程实例。这个信号可以由流程实例中的信号抛出中间事件（intermediary signal throw event），或者API（`runtimeService.signalEventReceivedXXX`方法）触发。这些情况下，所有拥有相同名字信号启动事件的流程定义都会被启动。

请注意这些情况下，都可以选择异步还是同步启动流程实例。

需要为API传递的 `signalName`，是由 `signal` 元素的 `name` 属性决定的名字。`signal` 元素被 `signalEventDefinition` 的 `signalRef` 属性所引用。

图示 Graphical notation

信号启动事件，用其中有一个信号事件标志的圆圈表示。这个标志并未填充，用以表示捕获（接收）行为。



XML表示 XML representation

信号启动事件的XML表示格式，为普通启动事件声明，加上signalEventDefinition子元素：

```

1 <signal id="theSignal" name="The Signal" />
2
3 <process id="processWithSignalStart1">
4   <startEvent id="theStart">
5     <signalEventDefinition id="theSignalEventDefinition" signalRef="theSignal" />
6   </startEvent>
7   <sequenceFlow id="flow1" sourceRef="theStart" targetRef="theTask" />
8   <userTask id="theTask" name="Task in process A" />

```



```

9      <sequenceFlow id="flow2" sourceRef="theTask" targetRef="theEnd" />
10      <endEvent id="theEnd" />
11 </process>

```

8.2.11. 错误启动事件 Error Start Event

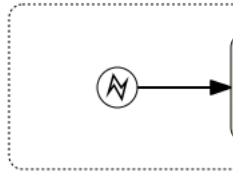
描述 Description

错误启动事件，可用于触发事件子流程（Event Sub-Process）。错误启动事件不能用于启动流程实例。

错误启动事件总是中断。

图示 Graphical notation

错误启动事件，用其中有一个错误事件标志的圆圈表示。这个标志并未填充，用以表示捕获（接收）行为。



XML表示 XML representation

错误启动事件的XML表示格式，为普通启动事件声明，加上errorEventDefinition子元素：

```

1 <startEvent id="messageStart" >
2     <errorEventDefinition errorRef="someError" />
3 </startEvent>

```

8.2.12. 结束事件 End Events

结束事件标志着（子）流程的（分支的）结束。结束事件总是抛出（型）事件。这意味着当流程执行到达结束事件时，会抛出一个结果。结果的类型由事件内部的黑色图标描绘。在XML表示中，类型由子元素声明给出。

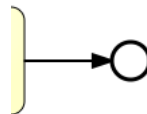
8.2.13. 空结束事件 None End Event

描述 Description

“空”结束事件，意味着当到达这个事件时，抛出的结果没有特别指定。因此，引擎除了结束当前执行分支之外，不会多做任何事情。

图示 Graphical notation

空结束事件，用其中没有图标（没有结果类型）的粗圆圈表示。



XML表示 XML representation

空事件的XML表示格式，为普通结束事件声明，没有任何子元素（其它种类的结束事件都有子元素，用于声明其类型）。

```

1 <endEvent id="end" name="my end event" />

```

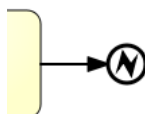
8.2.14. 错误结束事件 Error End Event

描述 Description

当流程执行到达错误结束事件时，结束执行的当前分支，并抛出错误。这个错误可以使用匹配的[错误边界中间事件 intermediate boundary error event](#)捕获。如果找不到匹配的错误边界事件，将会抛出异常。

图示 Graphical notation

错误结束事件，用内部有一个错误图标（全黑）的标准结束事件（粗圆圈）表示。错误图标是全黑的，代表抛出的含义。



XML表示 XML representation

错误结束事件，表示为结束事件，加上`errorEventDefinition`子元素：

```
1 <endEvent id="myErrorEndEvent">
2   <errorEventDefinition errorRef="myError" />
3 </endEvent>
```

`errorRef`属性可以引用在流程外定义的`error`元素：

```
1 <error id="myError" errorCode="123" />
2 ...
3 <process id="myProcess">
4   ...
```

`error`的`errorCode`用于查找匹配的错误捕获边界事件。如果`errorRef`不匹配任何已定义的`error`，则该`errorRef`会用做`errorCode`的快捷方式。这个快捷方式是Activiti特有的。下面的代码片段在功能上是相同的。

```
1 <error id="myError" errorCode="error123" />
2 ...
3 <process id="myProcess">
4   ...
5   <endEvent id="myErrorEndEvent">
6     <errorEventDefinition errorRef="myError" />
7   </endEvent>
8   ...
```

与下面的功能相同

```
1 <endEvent id="myErrorEndEvent">
2   <errorEventDefinition errorRef="error123" />
3 </endEvent>
```

请注意`errorRef`必须遵从BPMN 2.0概要（schema），且必须是合法的QName。

8.2.15. 终止结束事件 Terminate End Event

描述 Description

当到达终止结束事件时，当前的流程实例或子流程会被终止。概念上说，当执行到达终止结束事件时，会判断第一个范围 `scope`（流程或子流程）并终止它。请注意在BPMN 2.0中，子流程可以是嵌入式子流程，调用活动，事件子流程，或事务子流程。有一条通用规则：当存在多实例的调用过程或嵌入式子流程时，只会终止一个实例，其他的实例与流程实例不会受影响。

可以添加一个可选属性`terminateAll`。当其为`true`时，无论该终止结束事件在流程定义中的位置，也无论它是否在子流程（甚至是嵌套子流程）中，都会终止（根）流程实例。

图示 Graphical notation

终止结束事件，用内部有一个全黑圆的标准结束事件（粗圆圈）表示。



XML表示 XML representation

终止结束事件，表示为结束事件，加上`terminateEventDefinition`子元素。

请注意`terminateAll`属性是可选的（默认为`false`）。

```
1 <endEvent id="myEndEvent">
2   <terminateEventDefinition activiti:terminateAll="true"></terminateEventDefinition>
3 </endEvent>
```

8.2.16. 取消结束事件 Cancel End Event

[EXPERIMENTAL]

描述 Description

取消结束事件，只能与bpmn事务子流程（bpmn transaction subprocess）一起使用。当到达取消结束事件时，会抛出取消事件，且必须由取消边界事件（cancel boundary event）捕获。之后这个取消边界事件将取消事务，并触发补偿（compensation）。

图示 Graphical notation

取消结束事件，用内部有一个取消图标（粗圆圈）表示。取消图标是全黑的，代表抛出的含义。



XML表示 XML representation

取消结束事件，表示为结束事件，加上`cancelEventDefinition`子元素。

```
1 <endEvent id="myCancelEndEvent">
2   <cancelEventDefinition />
3 </endEvent>
```

8.2.17. 边界事件 Boundary Events

边界事件是捕获（型）事件，依附在活动（`activity`）上（边界事件永远不会抛出）。这意味着当活动运行时，事件在监听特定类型的触发器。当事件捕获时，活动会被终止，并沿该事件的出口顺序流继续。

所有的边界事件都用相同的方式定义：

```
1 <boundaryEvent id="myBoundaryEvent" attachedToRef="theActivity">
2   <XXXEventDefinition/>
3 </boundaryEvent>
```

边界事件由下列（元素）定义：

- 唯一标识符（流程范围）
- 通过`attachedToRef`属性定义的，对该事件所依附的活动的引用。请注意边界事件，与其所依附的活动，定义在相同级别（也就是说，边界事件并不包含在活动内部）。
- 定义了边界事件的类型的，`XXXEventDefinition`形式的XML子元素（例如`TimerEventDefinition`，`ErrorEventDefinition`，等等）。查阅特定边界事件类型，以了解更多细节。

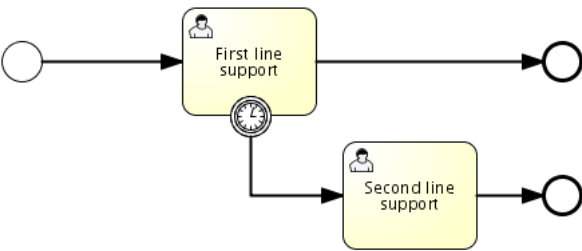
8.2.18. 定时器边界事件 Timer Boundary Event

描述 Description

定时器边界事件的行为像是跑表与闹钟。当执行到达边界事件所依附的活动时，启动定时器。当定时器触发时（例如在特定事件间隔后），活动会被中断，沿着边界事件继续执行。

图示 Graphical Notation

定时器边界事件，用内部有一个定时器图标（圆圈）表示。



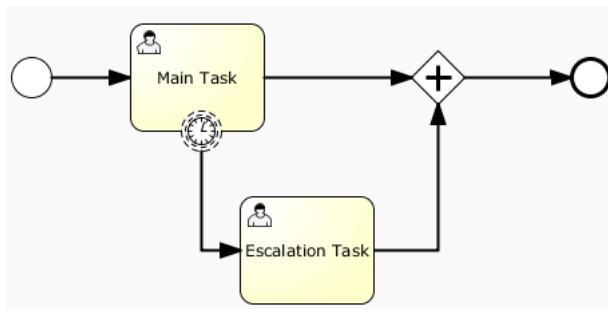
XML表示 XML Representation

定时器边界事件与一般边界事件一样定义。其中类型子元素为`timerEventDefinition`元素。

```
1 <boundaryEvent id="escalationTimer" cancelActivity="true" attachedToRef="firstLineSupport">
2   <timerEventDefinition>
3     <timeDuration>PT4H</timeDuration>
4   </timerEventDefinition>
5 </boundaryEvent>
```

请参考[定时器事件定义](#)了解定时器配置的细节。

上面的例子在图示中，圆圈画为虚线：



其典型使用场景，是发送额外的升级邮件，但不中断正常的流程流向。

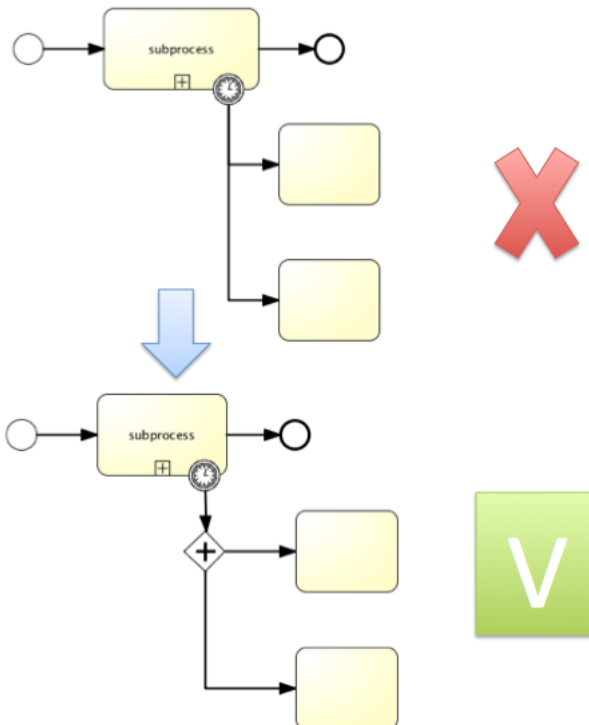
在BPMN 2.0中，中断与非中断定时器事件是不同的。默认为中断。非中断意味着最初的活动不会被中断，而会保留。并会创建额外的执行，用于处理事件的出口转移（outgoing transition）。在XML表示中，*cancelActivity*属性设置为false。

```
1 <boundaryEvent id="escalationTimer" cancelActivity="false" attachedToRef="firstLineSupport"/>
```

请注意：定时器边界事件只有在作业或异步执行器启用时才能触发（也就是说，需要在 `activiti.cfg.xml` 中，将 *jobExecutorActivate* 或者 *asyncExecutorActivate* 设置为 `true`。因为默认情况下，作业与异步执行器都是禁用的。）

边界事件的已知问题 Known issue with boundary events

所有类型的边界事件，都有一个关于并发的已知问题。不能在边界事件上附加多个出口顺序流（查看问题 [ACT-47](#)）。这个问题的解决方案，是使用一条出口顺序流，指向并行网关。



8.2.19. 错误边界事件 Error Boundary Event

描述 Description

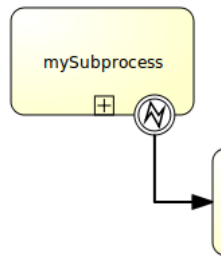
在活动边界上的错误捕获中间（事件），或简称错误边界事件，捕获其依附的活动范围内抛出的错误。

在 [嵌入式子流程](#) 或者 [调用活动](#) 上定义错误边界事件最有意义，因为子流程会为其中的所有活动创建范围。错误由 [错误结束事件](#) 抛出。这样的错误会逐层向其上级父范围传播，直到找到一个错误边界事件的范围，该范围定义了匹配的错误事件定义。

当错误事件被捕获时，边界事件定义所在的活动会被销毁，同时销毁其中所有的当前执行（例如，并行活动，嵌套子流程，等等）。流程执行沿着边界事件的出口顺序流继续。

图示 Graphical notation

错误边界事件，用内部有一个错误图标（两层圆圈）的标准中间事件（两层圆圈）表示。错误图标是白色的，代表捕获的含义。



XML表示 Xml representation

错误边界事件与标准边界事件一样定义：

```
1 <boundaryEvent id="catchError" attachedToRef="mySubProcess">
2   <errorEventDefinition errorRef="myError"/>
3 </boundaryEvent>
```

在边界事件中，`errorRef`引用一个流程元素外定义的错误：

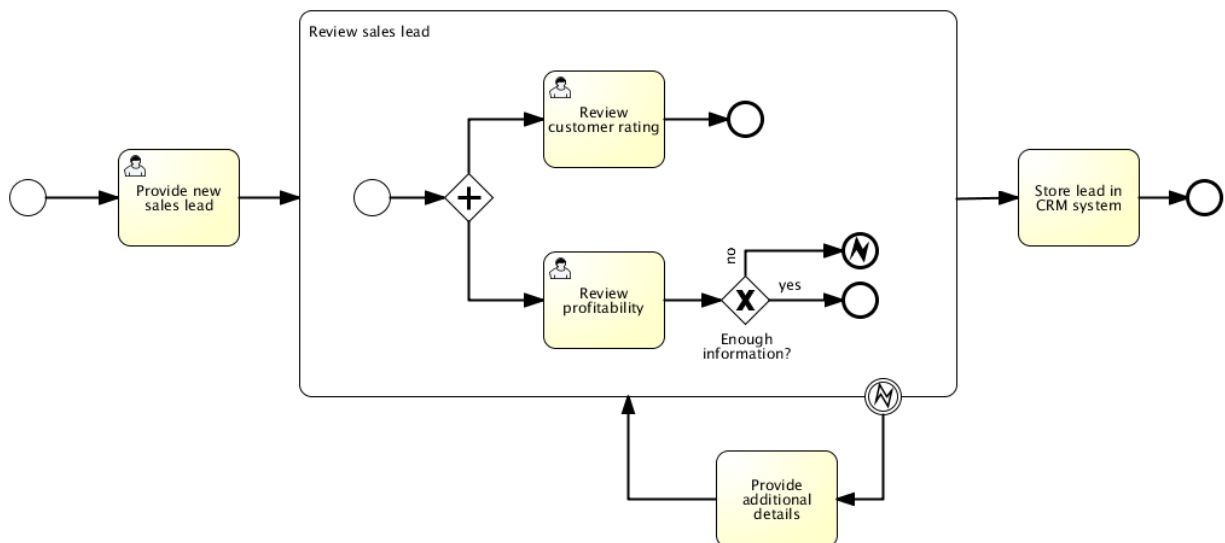
```
1 <error id="myError" errorCode="123" />
2 ...
3 <process id="myProcess">
4 ...
```

`errorCode`用于匹配捕获的错误：

- 如果省略了`errorRef`，错误边界事件会捕获所有错误事件，无论`error`的`errorCode`是什么。
- 如果提供了`errorRef`，并且其引用了存在的`error`，则边界事件只会捕获相同错误代码的错误。
- 如果提供了`errorRef`，但BPMN 2.0文件中没有定义`error`，则`errorRef`会用作`errorCode`（与错误结束事件类似）。

示例 Example

下面的示例流程展示了如何使用错误结束事件。当'*Review profitability* (审核盈利能力)'用户任务完成，并指出提供的信息不足时，会抛出错误。当这个错误被子流程边界捕获时，'*Review sales lead* (审核销售线索)'子流程中的所有运行中活动都会被销毁（即使'*Review customer rating* 审核客户等级'还没有完成），并创建'*Provide additional details* (提供更多信息)'用户任务。



这个流程作为演示配置的示例提供。可以在`org.activiti.examples.bpmn.event.error`包中找到流程XML与单元测试。

8.2.20. 信号边界事件 Signal Boundary Event

描述 Description

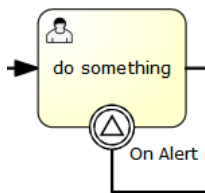
依附在活动边界上的信号捕获中间（事件），或简称信号边界事件，捕获与其信号定义具有相同信号名的信号。

请注意：与其他事件例如错误边界事件不同的是，信号边界事件不只是捕获其所依附范围抛出的信号。信号边界事件为全局范围（广播）的，意味着信号可以从任何地方抛出，甚至是不同的流程实例。

请注意：与其他事件如错误事件不同，信号在被捕获后不会被消耗。如果有两个激活的信号边界事件，捕获相同的信号事件，则两个边界事件都会被触发，哪怕它们不在同一个流程实例里。

图示 Graphical notation

信号边界事件，用内部有一个信号图标（两层圆圈）表示。信号图标是白色的，代表捕获的含义。



XML表示 XML representation

信号边界事件与标准边界事件一样定义：

```
1 <boundaryEvent id="boundary" attachedToRef="task" cancelActivity="true">
2   <signalEventDefinition signalRef="alertSignal"/>
3 </boundaryEvent>
```

示例 Example

查看[信号事件定义](#)章节内容。

8.2.21. 消息边界事件 Message Boundary Event

描述 Description

在活动边界上的消息捕获中间（事件），或简称消息边界事件，捕获与其消息定义具有相同消息名的消息。

图示 Graphical notation

消息边界事件，用内部有一个消息图标（两层圆圈）表示。信号图标是白色的，代表捕获的含义。



请注意消息边界事件既可以是中断型的（右边），也可以是非中断型的（左边）。

XML表示 XML representation

消息边界事件与标准边界事件一样定义：

```
1 <boundaryEvent id="boundary" attachedToRef="task" cancelActivity="true">
2   <messageEventDefinition messageRef="newCustomerMessage"/>
3 </boundaryEvent>
```

示例 Example

查看[消息事件定义](#)章节内容。

8.2.22. 取消边界事件 Cancel Boundary Event

[EXPERIMENTAL]

描述 Description

依附在事务子流程边界上的取消捕获中间（事件），或简称取消边界事件，在事务取消时触发。当取消边界事件触发时，首先会中断当前范围的所有活动执行。接下来，启动事务范围内所有有效的的补偿边界事件（**compensation boundary event**）。补偿会同步执行，也就是说在离开事务前，边界事件会等待补偿完成。当补偿完成时，使用取消边界事件的出口顺序流，离开事务子流程。

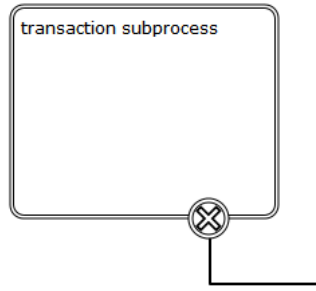
请注意：一个事务子流程只允许一个取消边界事件。

请注意：如果事务子流程中有嵌套的子流程，只有成功完成的子流程才会触发补偿。

请注意：如果取消边界事件放置在具有多实例特性的事务子流程上，如果一个实例触发了取消，则边界事件将取消所有实例。

图示 Graphical notation

取消边界事件，用内部有一个取消图标（两层圆圈）表示。取消图标是白色的（未填充），代表捕获的含义。



XML表示 XML representation

取消边界事件与标准边界事件一样定义：

```
1 <boundaryEvent id="boundary" attachedToRef="transaction" >
2   <cancelEventDefinition />
3 </boundaryEvent>
```

因为取消边界事件总是中断型的，因此不需要 `cancelActivity` 属性。

8.2.23. 补偿边界事件 Compensation Boundary Event

[EXPERIMENTAL]

描述 Description

依附在活动边界上的补偿捕获中间（事件），或简称补偿边界事件，可以为活动附加补偿处理器。

补偿边界事件必须通过直接关联的方式，引用单个的补偿处理器。

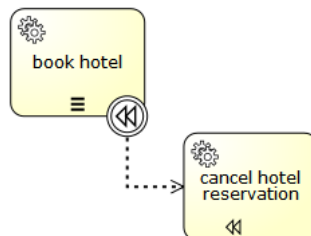
补偿边界事件与其它边界事件的活动策略不同。其它边界事件，例如信号边界事件，当其依附的活动启动时激活；当离开该活动时，会被解除，并取消相应的事件订阅。而补偿边界事件不是这样。补偿边界事件在其依附的活动成功完成时激活，同时创建补偿事件的相应订阅。当补偿事件被触发，或者相应的流程实例结束时，才会移除订阅。请考虑下列因素：

- 当补偿被触发时，补偿边界事件关联的补偿处理器会被调用，次数与其依附的活动成功完成的次数相同。
- 如果补偿边界事件依附在具有多实例特性的活动上，则会为每一个实例创建补偿事件订阅。
- 如果补偿边界事件依附在位于循环内部的活动上，则每次该活动执行时，都会创建一个补偿事件订阅。
- 如果流程实例结束，则取消补偿事件的订阅。

请注意：嵌入式子流程不支持补偿边界事件。

图示 Graphical notation

补偿边界事件，用内部有一个补偿图标（两层圆圈）的标准中间事件（两层圆圈）表示。补偿图标是白色的（未填充），代表捕获的含义。另外，补偿边界事件使用单向连接关联补偿处理器，如下图所示：



XML表示 XML representation

补偿边界事件与标准边界事件一样定义：

```
1 <boundaryEvent id="compensateBookHotelEvt" attachedToRef="bookHotel" >
2   <compensateEventDefinition />
3 </boundaryEvent>
4
5 <association associationDirection="One" id="a1" sourceRef="compensateBookHotelEvt" targetRef="undoBookHotel" />
6
7 <serviceTask id="undoBookHotel" isForCompensation="true" activiti:class="..." />
```

补偿边界事件在活动完成后才激活，因此不支持 `cancelActivity` 属性。

8.2.24. 捕获中间事件 Intermediate Catching Events

所有的捕获中间事件都使用相同方式定义：

```
1 <intermediateCatchEvent id="myIntermediateCatchEvent" >
2   <XXXEventDefinition/>
3 </intermediateCatchEvent>
```

捕获中间事件由下列（元素）定义

- 唯一标识符（流程范围）
- 定义了捕获中间事件类型的，*XXXEventDefinition*形式的XML子元素（例如*TimerEventDefinition*等）。查阅特定中间捕获事件类型，以了解更多细节。

8.2.25. 定时器捕获中间事件 Timer Intermediate Catching Event

描述 Description

定时器捕获中间事件的行为像是跑表。当执行到达捕获事件活动（catching event activity）时，启动定时器；当定时器触发时（例如在一段时间间隔后），沿定时器中间事件的出口顺序流继续执行。

图示 Graphical Notation

定时器中间事件，用内部有定时器图标（带指针的圆）的中间捕获事件表示。



XML表示 XML Representation

定时器中间事件与捕获中间事件一样定义。指定类型的子元素为*timerEventDefinition*元素。

```
1 <intermediateCatchEvent id="timer">
2   <timerEventDefinition>
3     <timeDuration>PT5M</timeDuration>
4   </timerEventDefinition>
5 </intermediateCatchEvent>
```

查看[定时器事件定义](#)了解详细配置。

8.2.26. 信号捕获中间事件 Signal Intermediate Catching Event

描述 Description

信号捕获中间事件，捕获与其引用的信号定义具有相同信号名称的信号。

请注意：与其他事件如错误事件不同，信号在被捕获后不会被消耗。如果有两个激活的信号中间事件，捕获相同的信号事件，则两个中间事件都会被触发，哪怕它们不在同一个流程实例里。

图示 Graphical notation

信号捕获中间事件，用内部有信号图标（带三角形的圆）的标准中间事件（两层圆圈）表示。信号图标是白色的（未填充），代表捕获的含义。



XML表示 XML representation

信号中间事件与捕获中间事件一样定义。指定类型的子元素为*signalEventDefinition*元素。

```
1 <intermediateCatchEvent id="signal">
2   <signalEventDefinition signalRef="newCustomerSignal" />
3 </intermediateCatchEvent>
```

示例 Example

查看[信号事件定义](#)章节。

8.2.27. 消息捕获中间事件 Message Intermediate Catching Event

描述 Description

消息捕获中间事件，捕获特定名字的消息。

图示 Graphical notation

消息捕获中间事件，用内部有消息图标（带信封的圆）的标准中间事件（两层圆圈）表示。消息图标是白色的（未填充），代表捕获的含义。



XML表示 XML representation

消息中间事件与[捕获中间事件](#)一样定义。指定类型的子元素为**messageEventDefinition**元素。

```
1 <intermediateCatchEvent id="message">
2   <messageEventDefinition signalRef="newCustomerMessage" />
3 </intermediateCatchEvent>
```

示例 Example

查看[消息事件定义](#)章节。

8.2.28. 抛出中间事件 Intermediate Throwing Event

所有的抛出中间事件都使用相同方式定义：

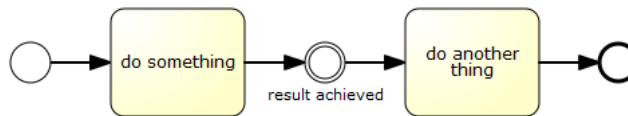
```
1 <intermediateThrowEvent id="myIntermediateThrowEvent" >
2   <XXXEventDefinition/>
3 </intermediateThrowEvent>
```

抛出中间事件由下列（元素）定义

- 唯一标识符（流程范围）
- 定义了抛出中间事件类型的，*XXXEventDefinition*形式的XML子元素（例如[signalEventDefinition](#)等）。查阅特定中间抛出事件类型，以了解更多细节。

8.2.29. 空抛出中间事件 Intermediate Throwing None Event

下面的流程图展示了空中间事件的简单例子，其用于指示流程已经到达了某种状态。



基本上添加一个[执行监听器 execution listener](#)后，空中间事件就可以成为很好的监视某些KPI（Key Performance Indicators 关键绩效指标）的钩子。

```
1 <intermediateThrowEvent id="noneEvent">
2   <extensionElements>
3     <activiti:executionListener class="org.activiti.engine.test.bpmn.event.IntermediateNoneEventTest$MyExecutionListener"
4     event="start" />
5   </extensionElements>
6 </intermediateThrowEvent>
```

你也可以添加一些自己的代码，将部分事件发送给你的BAM（Business Activity Monitoring 业务活动监控）工具，或者DWH（Data Warehouse 数据仓库）。引擎本身不会在事件中做任何事情，只是从中穿过。

8.2.30. 信号抛出中间事件 Signal Intermediate Throwing Event

描述 Description

[信号](#)抛出中间事件，抛出已定义信号的信号事件。

在Activiti中，信号会广播至所有的激活的处理器（也就是说，所有的捕获信号事件）。信号可以同步或异步地发布。

- 在默认配置中，信号同步地传递。这意味着抛出（信号的）流程实例会等待，直到信号传递至所有的捕获（信号的）流程实例。所有的捕获流程实例也会在与抛出流程实例相同的事务中，也就是说如果收到通知的流程实例中，有一个实例产生了技术错误（抛出异常），则所有相关的实例都会失败。
- 信号也可以异步地传递。这是由到达抛出信号事件时，激活的是哪一个（发送）处理器来决定的。对于每个激活的处理器，JobExecutor会为其存储并传递一个异步通知消息，*asynchronous notification message*（作业 Job）。

图示 Graphical notation

消息抛出中间事件，用内部有信号图标的标准中间事件（两层圆圈）表示。信号图标是黑色的（已填充），代表抛出的含义。



XML表示 XML representation

信号中间事件与抛出中间事件一样定义。指定类型的子元素为**signalEventDefinition**元素。

```
1 <intermediateThrowEvent id="signal">
2   <signalEventDefinition signalRef="newCustomerSignal" />
3 </intermediateThrowEvent>
```

异步信号事件像这样定义：

```
1 <intermediateThrowEvent id="signal">
2   <signalEventDefinition signalRef="newCustomerSignal" activiti:async="true" />
3 </intermediateThrowEvent>
```

示例 Example

查看[信号事件定义](#)章节。

8.2.31. 补偿抛出中间事件 Compensation Intermediate Throwing Event

[EXPERIMENTAL]

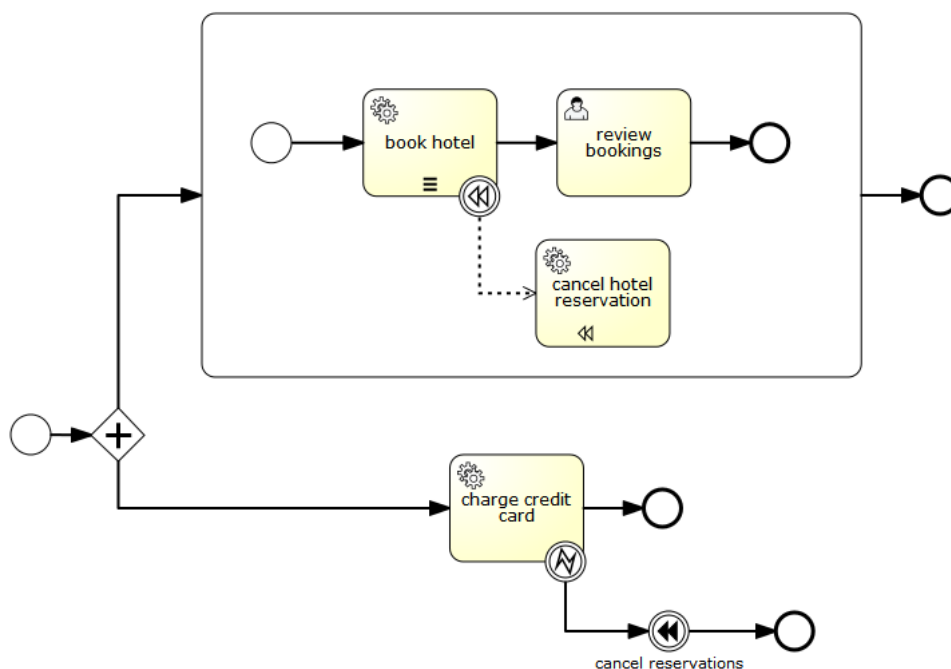
描述 Description

补偿抛出中间事件，可用于触发补偿。

触发补偿：补偿既可以为设计的活动触发，也可以为补偿事件所在的范围触发。补偿由活动所关联的补偿处理器执行。

- 抛出补偿时，活动关联的补偿处理器执行的次数，与活动成功完成的次数相同。
- 如果为当前范围抛出了补偿，则当前范围中所有的活动都会被补偿，包括并行分支上的活动。
- 补偿分层触发：如果将要被补偿的活动是一个子流程，则该子流程中所有的活动都会触发补偿。如果该子流程有嵌套的活动，则会递归地抛出补偿。然而，补偿不会传播至流程的上层：如果子流程中触发了补偿，该补偿不会传播至子流程范围外的活动。BPMN规范指出，补偿为“与子流程在相同级别”的活动触发。
- 在Activiti中，补偿按照执行的相反顺序运行。这意味着最后完成的活动会第一个补偿，等等。
- 补偿抛出中间事件，可用于补偿已经成功完成的事务子流程。

请注意：如果抛出补偿的范围中有一个子流程，而该子流程包含有关联了补偿处理器的活动，则当抛出补偿时，只有当 该子流程成功完成的情况，补偿才会传播至该子流程。如果子流程内嵌套的部分活动已经完成，并附加了补偿处理器，则如果包含这些活动的子流程还没有完成，这些补偿处理器不会执行。参考下面的例子：



在这个流程中，有两个并行的执行。一个执行嵌入子流程，另一个执行“charge credit card（信用卡付款）”活动。假定两个执行都已开始，且第一个并行执行正等待用户完成“review bookings（检查预定）”任务。第二个执行进行了“charge credit card（信用卡付款）”活动的操作，抛出了一个错误，导致“cancel reservations（取消预订）”事件触发补偿。这时并行子流程还未完成，意味着补偿不会传播至该子流程，因此不会执行“cancel hotel reservation（取消酒店预订）”补偿处理器。而如果“cancel reservations（取消预订）”运行前，这个用户任务（因此该嵌入式子流程也）已经完成，则补偿会传播至该嵌入式子流程。

流程变量：当补偿嵌入式子流程时，用于执行补偿处理器的执行，可以以变量在子流程完成时所处的状态，访问子流程的局部流程变量。围了实现这一点，会为范围执行（为执行子流程所创建的执行）所关联的流程变量，进行快照。意味着：

- 子流程范围内创建的并行执行所添加的变量，补偿执行器无法访问。
- 上层的执行关联的流程变量（例如流程实例的执行关联的流程变量），不在该快照中：补偿处理器（本就）可以以其在抛出补偿时所处的状态，访问这些流程变量。
- 只会为嵌入式子流程，而不会为其他活动，进行变量快照。

目前的限制：

- 目前不支持 `waitForCompletion="false"`。当补偿抛出中间事件触发补偿时，只有在补偿成功完成时，才会离开该事件。
- 补偿现在由并行执行来运行。该并行执行按照补偿活动完成的逆序启动。**Activiti**的后续版本可能会添加选项，使补偿可以按（活动完成的）顺序运行。
- 补偿不会传播至调用活动（call activity）生成的子流程。

图示 Graphical notation

补偿抛出中间事件，用内部有补偿图标（两层圆圈）的标准中间事件（两层圆圈）表示。补偿图标是黑色的（已填充），代表抛出的含义。



Xml representation

补偿中间事件与抛出中间事件一样定义。指定类型的子元素为**compensateEventDefinition**元素。

```
1 <intermediateThrowEvent id="throwCompensation">
2   <compensateEventDefinition />
3 </intermediateThrowEvent>
```

另外，**activityRef**可选项可用于为指定的范围/活动触发补偿：

```
1 <intermediateThrowEvent id="throwCompensation">
2   <compensateEventDefinition activityRef="bookHotel" />
3 </intermediateThrowEvent>
```

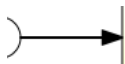
8.3. 顺序流 Sequence Flow

8.3.1. 描述 Description

顺序流是流程中两个元素间的连接器。当流程执行中，一个元素被访问后，会沿着所有的出口顺序流继续。这意味着BPMN 2.0的默认性质是并行的：两个出口顺序流，会创建两个独立的，并行的执行路径。

8.3.2. 图示 Graphical notation

顺序流，用从源元素指向目标元素的箭头表示。箭头总是指向目标元素。



8.3.3. XML表示 XML representation

顺序流需要有流程唯一的**id**，以及对存在的源与目标元素的引用。

```
1 <sequenceFlow id="flow1" sourceRef="theStart" targetRef="theTask" />
```

8.3.4. 条件顺序流 Conditional sequence flow

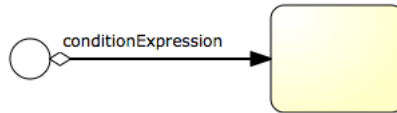
描述 Description

在顺序流上可以定义条件。当离开BPMN 2.0活动时，默认行为是计算其出口顺序流上的条件。当条件计算为**true**时，选择该出口顺序流。如果该方法选择了多条顺序流，则会生成多个执行，流程会以并行方式继续。

请注意：上面的介绍对BPMN 2.0活动（与事件）有效，但不适用于网关（gateway）。不同类型的网关，会用不同的方式处理带有条件的顺序流。

图示 Graphical notation

条件顺序流，用起点带有小菱形的一般顺序流表示。条件表达式挨着顺序流显示。



XML表示 XML representation

条件顺序流的XML表示格式，为含有**conditionExpression**（条件表达式）子元素的普通顺序流。请注意目前只支持**tFormalExpressions**。省略**xsi:type=""**定义会默认为唯一支持的表达式类型。

```

1 <sequenceFlow id="flow" sourceRef="theStart" targetRef="theTask">
2   <conditionExpression xsi:type="tFormalExpression">
3     <![CDATA[${order.price > 100 && order.price < 250}]]>
4   </conditionExpression>
5 </sequenceFlow>
  
```

目前conditionalExpressions只能使用**UEL**，详细信息可以在[表达式](#)章节找到。使用的表达式需要能解析为boolean值，否则当计算条件时会抛出异常。

- 下面的例子，通过典型的JavaBean的方式，使用getter引用流程变量的数据。

```

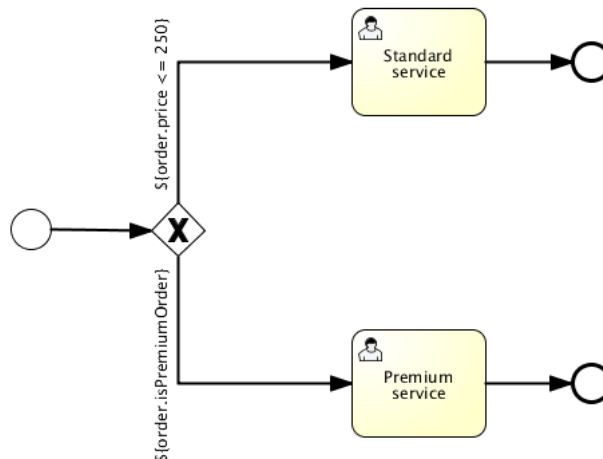
1 <conditionExpression xsi:type="tFormalExpression">
2   <![CDATA[${order.price > 100 && order.price < 250}]]>
3 </conditionExpression>
  
```

- 这个例子调用了个解析为boolean值的方法。

```

1 <conditionExpression xsi:type="tFormalExpression">
2   <![CDATA[${order.isStandardOrder()}]]>
3 </conditionExpression>
  
```

Activiti发行版中包含了下列示例流程，展示值表达式与方法表达式的使用（参见[org.activiti.examples.bpmn.expression](#)）。



8.3.5. 默认顺序流 Default sequence flow

描述 Description

所有的BPMN 2.0任务与网关，都可以使用默认顺序流。这种顺序流只有当没有其他顺序流可以选择时，才会被选择为活动的出口顺序流。默认顺序流上的条件会被忽略。

图示 Graphical notation

默认顺序流，用起点带有“斜线”标记的一般顺序流表示。



XML表示 XML representation

活动的默认顺序流，由该活动的**default**属性定义。下面的XML片段展示了一个排他网关（exclusive gateway），带有默认顺序流**flow 2**。只有当**conditionA**与**conditionB**都计算为false时，默认顺序流才会被选择为网关的出口顺序流。


```

1 <exclusiveGateway id="exclusiveGw" name="Exclusive Gateway" default="flow2" />
2 <sequenceFlow id="flow1" sourceRef="exclusiveGw" targetRef="task1">
3   <conditionExpression xsi:type="tFormalExpression">${conditionA}</conditionExpression>
4 </sequenceFlow>
5 <sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="task2"/>
6 <sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="task3">
7   <conditionExpression xsi:type="tFormalExpression">${conditionB}</conditionExpression>
8 </sequenceFlow>

```

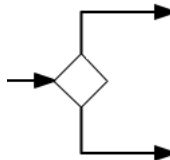
对应下面的图示：

（原图缺失）

8.4. 网关 Gateways

网关用于控制执行的流向（或者按BPMN 2.0描述的，执行的`token`标志）。网关可以消耗与生成标志。

网关用其中带有图标的菱形表示。该图标显示了网关的类型。



8.4.1. 排他网关 Exclusive Gateway

描述 Description

排他网关（也叫异或网关 *XOR gateway*，或者更专业的，基于数据的排他网关 *exclusive data-based gateway*），用于为流程中的决策建模。当执行到达这个网关时，所有出口顺序流会按照它们定义的顺序进行计算。条件计算为`true`的顺序流（当没有设置条件时，认为顺序流定义为`true`）会被选择用于继续流程。

请注意这里出口顺序流的含义与BPMN 2.0中的一般情况不一样。一般情况下，所有条件计算为`true`的顺序流，都会被选择继续，并行执行。而使用排他网关时，只会选择一条顺序流。当多条顺序流的条件都计算为`true`时，其中在XML中定义的第一条（也只有这条）会被选择，用于继续流程。如果没有可选的顺序流，会抛出异常。

图示 Graphical notation

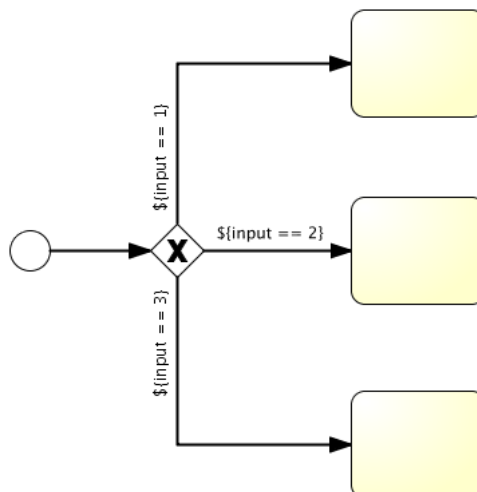
排他网关，用内部带有'X'图标的标准网关（菱形）表示，'X'图标代表异或（XOR）的含义。请注意内部没有图标的网关默认为排他网关。BPMN 2.0规范不允许在同一个流程中，混合使用带有及没有X的菱形标志。



XML表示 XML representation

排他网关的XML表示格式很直接：一行定义网关的XML，而条件表达式定义在出口顺序流上。查看[条件顺序流](#)章节了解这种表达式的可用选项。

以下的模型为例：



用XML表示如下:

```
1 <exclusiveGateway id="exclusiveGw" name="Exclusive Gateway" />
2
3 <sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="theTask1">
4   <conditionExpression xsi:type="tFormalExpression">${input == 1}</conditionExpression>
5 </sequenceFlow>
6
7 <sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="theTask2">
8   <conditionExpression xsi:type="tFormalExpression">${input == 2}</conditionExpression>
9 </sequenceFlow>
10
11 <sequenceFlow id="flow4" sourceRef="exclusiveGw" targetRef="theTask3">
12   <conditionExpression xsi:type="tFormalExpression">${input == 3}</conditionExpression>
13 </sequenceFlow>
```

8.4.2. 并行网关 Parallel Gateway

描述 Description

网关也可以用于对流程中并行的建模。在流程模型中引入并行的最简单的网关，就是并行网关。它可以将执行分支（*fork*）为多条路径，也可以合并（*join*）执行的多条入口路径。

并行网关的功能，基于其入口与出口顺序流：

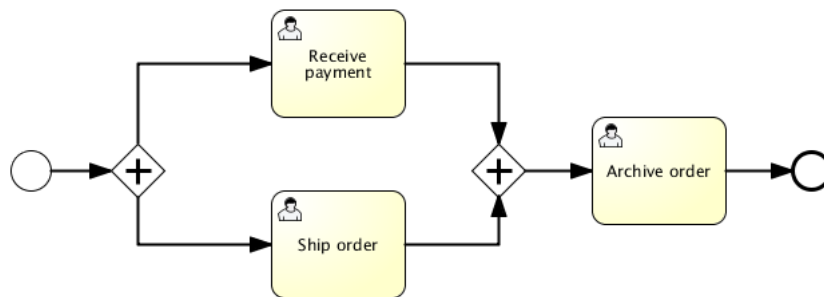
- 分支：所有的出口顺序流都并行执行，为每一条顺序流创建一个并行执行。
- 合并：所有到达并行网关的并行执行，都在网关处等待，直到每一条入口顺序流都有一个执行到达。然后流程经过该合并网关继续。

请注意，如果并行网关同时具有多条入口与出口顺序流，可以同时具有分支与合并的行为。在这种情况下，网关首先合并所有入口顺序流，然后分裂为多条并行执行路径。

与其他网关类型的重要区别，是并行网关不计算条件。如果连接到并行网关的顺序流上定义了条件，条件会被简单地忽略。

图示 Graphical Notation

并行网关，用内部带有‘加号’图标网关（菱形）表示，代表与（AND）的含义。



XML表示 XML representation

定义并行网关需要一行XML:

```
1 <parallelGateway id="myParallelGateway" />
```

实际行为（分支，合并或两者皆有），由连接到该并行网关的顺序流定义。

例如，上面的模型表现为下面的XML:

```
1 <startEvent id="theStart" />
2 <sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork" />
3
4 <parallelGateway id="fork" />
5 <sequenceFlow sourceRef="fork" targetRef="receivePayment" />
6 <sequenceFlow sourceRef="fork" targetRef="shipOrder" />
7
8 <userTask id="receivePayment" name="Receive Payment" />
9 <sequenceFlow sourceRef="receivePayment" targetRef="join" />
10
11 <userTask id="shipOrder" name="Ship Order" />
12 <sequenceFlow sourceRef="shipOrder" targetRef="join" />
13
14 <parallelGateway id="join" />
15 <sequenceFlow sourceRef="join" targetRef="archiveOrder" />
16
```

```

17 <userTask id="archiveOrder" name="Archive Order" />
18 <sequenceFlow sourceRef="archiveOrder" targetRef="theEnd" />
19
20 <endEvent id="theEnd" />

```

在上面的例子中，当流程启动后，会创建两个任务：

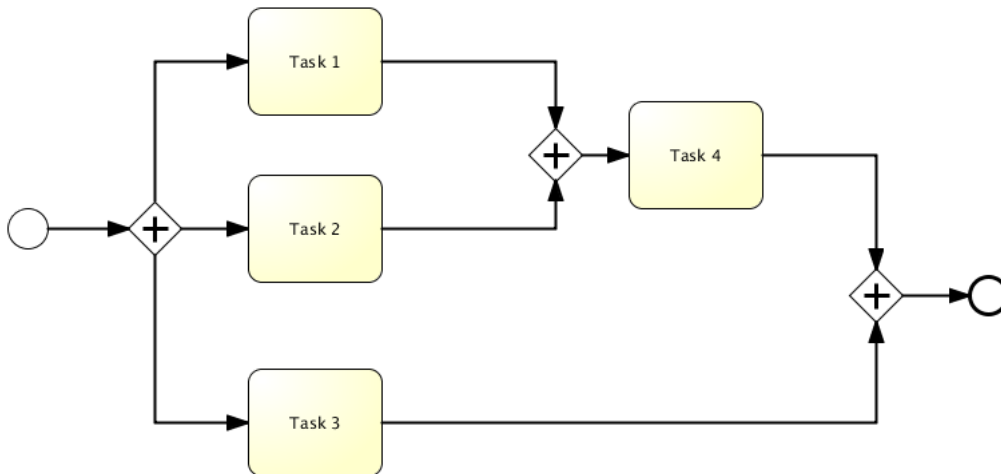
```

1 ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
2 TaskQuery query = taskService.createTaskQuery()
3     .processInstanceId(pi.getId())
4     .orderByTaskName()
5     .asc();
6
7 List<Task> tasks = query.list();
8 assertEquals(2, tasks.size());
9
10 Task task1 = tasks.get(0);
11 assertEquals("Receive Payment", task1.getName());
12 Task task2 = tasks.get(1);
13 assertEquals("Ship Order", task2.getName());

```

当这两个任务完成后，第二个并行网关会合并这两个执行，并且由于只有一条出口顺序流，不会再创建并行执行路径，只会激活**Archive Order**(存档订单)任务。

请注意并行网关不需要“平衡”（也就是说，对应的并行网关，其入口/出口顺序流的数量不需要匹配）。并行网关会简单地等待所有入口顺序流，并为每一条出口顺序流创建并行执行，不受流程模型中的其他结构影响。因此，下面的流程在BPMN 2.0中是合法的：



8.4.3. 包容网关 Inclusive Gateway

描述 Description

包容网关可被视作排他网关与并行网关的组合。与排他网关一样，可以在出口顺序流上定义条件，包容网关会计算它们。然而主要的区别是，包容网关与并行网关一样，可以选择多于一条（出口）顺序流。

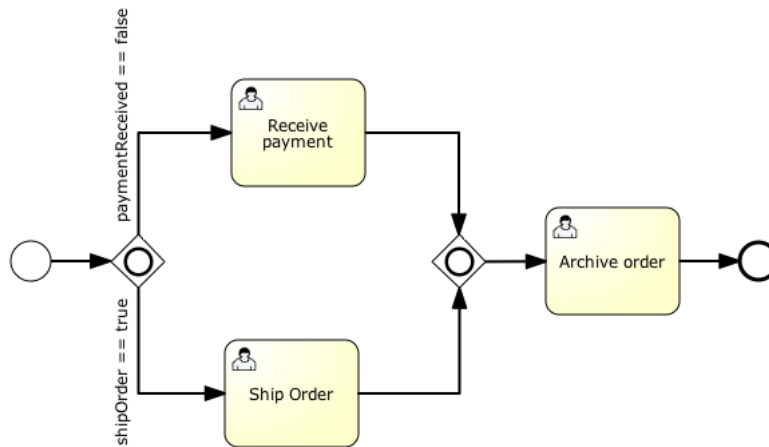
包容网关的功能，基于其入口与出口顺序流：

- 分支：所有出口顺序流的条件都会被计算，对于条件计算为**true**的顺序流，流程会并行地沿其继续，为每一条顺序流创建一个并行执行。
- 合并：所有到达包容网关的并行执行，都会在网关处等待，直到每一条具有流程标志的入口顺序流，都有一个执行到达。这是与并行网关的重要区别。换句话说，包容网关只会等待将会被执行的入口顺序流。在合并后，流程穿过合并并行网关继续。

请注意，如果包容网关同时具有多条入口与出口顺序流，可以同时具有分支与合并的行为。在这种情况下，网关首先合并所有具有流程标志的入口顺序流，然后为条件计算为**true**的出口顺序流，分裂为多条并行执行路径。

图示 Graphical Notation

包容网关，用内部带有‘圆圈’图标网关（菱形）表示。



XML表示 XML representation

定义包容网关需要一行XML:

```
1 <inclusiveGateway id="myInclusiveGateway" />
```

实际行为（分支，合并或两者皆有），由连接到该包容网关的顺序流定义。

例如，上面的模型表现为下面的XML:

```

1 <startEvent id="theStart" />
2 <sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork" />
3
4 <inclusiveGateway id="fork" />
5 <sequenceFlow sourceRef="fork" targetRef="receivePayment" >
6   <conditionExpression xsi:type="tFormalExpression">${paymentReceived == false}</conditionExpression>
7 </sequenceFlow>
8 <sequenceFlow sourceRef="fork" targetRef="shipOrder" >
9   <conditionExpression xsi:type="tFormalExpression">${shipOrder == true}</conditionExpression>
10 </sequenceFlow>
11
12 <userTask id="receivePayment" name="Receive Payment" />
13 <sequenceFlow sourceRef="receivePayment" targetRef="join" />
14
15 <userTask id="shipOrder" name="Ship Order" />
16 <sequenceFlow sourceRef="shipOrder" targetRef="join" />
17
18 <inclusiveGateway id="join" />
19 <sequenceFlow sourceRef="join" targetRef="archiveOrder" />
20
21 <userTask id="archiveOrder" name="Archive Order" />
22 <sequenceFlow sourceRef="archiveOrder" targetRef="theEnd" />
23
24 <endEvent id="theEnd" />

```

在上面的例子中，当流程启动后，如果流程变量`paymentReceived == false`且`shipOrder == true`，将会创建两个任务。如果只有一个流程变量等于`true`，则只会创建一个任务。如果没有条件计算为`true`，会抛出异常，并可通过指定默认出口顺序流避免。在下面的例子中，只有`ship order`（传递订单）一个任务会被创建:

```

1 HashMap<String, Object> variableMap = new HashMap<String, Object>();
2   variableMap.put("receivedPayment", true);
3   variableMap.put("shipOrder", true);
4   ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
5 TaskQuery query = taskService.createTaskQuery()
6   .processInstanceId(pi.getId())
7   .orderByTaskName()
8   .asc();
9
10 List<Task> tasks = query.list();
11 assertEquals(1, tasks.size());
12
13 Task task = tasks.get(0);
14 assertEquals("Ship Order", task.getName());

```

当这个任务完成后，第二个包容网关会合并这两个执行，并且由于只有一条出口顺序流，不会再创建并行执行路径，只会激活`Archive Order`(存档订单)任务。

请注意包容网关不需要“平衡”（也就是说，对应的包容网关，其入口/出口顺序流的数量不需要匹配）。包容网关会简单地等待所有入口顺序流，并为每一条出口顺序流创建并行执行，不受流程模型中的其他结构影响。

8.4.4. 基于事件的网关 Event-based Gateway

描述 Description

基于事件的网关，允许基于事件做选择。网关的每一条出口顺序流，都需要连接至一个捕获中间事件。当流程执行到达基于事件的网关时，网关类似等待状态地动作：执行被暂停。并且，为每一条出口顺序流，创建一个事件订阅。

请注意基于事件的网关，其出口顺序流与一般的顺序流不同。这些顺序流从不实际被执行。相反，它们允许流程引擎决定，当执行到达一个基于事件的网关时，需要订阅什么事件。基于下列约束：

- 一个基于事件的网关，必须有两条或更多的出口顺序流。
- 基于事件的网关，只能连接至 `intermediateCatchEvent`（捕获中间事件）类型的元素（Activiti不支持基于事件的网关后，连接接收任务，Receive Task）。
- 连接至基于事件的网关的 `intermediateCatchEvent`，必须只有一个入口顺序流。

图示 Graphical notation

基于事件的网关，用内部带有特殊图标的网关（菱形）表示。

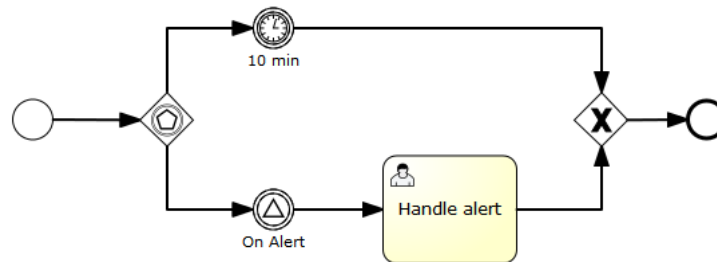


XML表示 XML representation

用于定义基于事件的网关的XML元素为 `eventBasedGateway`。

示例 Example(s)

下面的流程，是带有基于事件的网关的流程的例子。当执行到达基于事件的网关时，流程执行被暂停。并且，流程实例订阅alert信号事件，并创建一个10分钟后触发的定时器。这使得流程引擎等待10分钟，并等待信号事件。如果信号在10分钟内触发，则定时器会被取消，执行沿着信号继续。如果信号未被触发，执行会在定时器到时后继续，并取消信号订阅。



```
1 <definitions id="definitions"
2   xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:activiti="http://activiti.org/bpmn"
4   targetNamespace="Examples">
5
6   <signal id="alertSignal" name="alert" />
7
8   <process id="catchSignal">
9
10    <startEvent id="start" />
11
12    <sequenceFlow sourceRef="start" targetRef="gw1" />
13
14    <eventBasedGateway id="gw1" />
15
16    <sequenceFlow sourceRef="gw1" targetRef="signalEvent" />
17    <sequenceFlow sourceRef="gw1" targetRef="timerEvent" />
18
19    <intermediateCatchEvent id="signalEvent" name="Alert">
20      <signalEventDefinition signalRef="alertSignal" />
21    </intermediateCatchEvent>
22
23    <intermediateCatchEvent id="timerEvent" name="Alert">
24      <timerEventDefinition>
25        <timeDuration>PT10M</timeDuration>
26      </timerEventDefinition>
27    </intermediateCatchEvent>
28
29    <sequenceFlow sourceRef="timerEvent" targetRef="exGw1" />
30    <sequenceFlow sourceRef="signalEvent" targetRef="task" />
31
32    <userTask id="task" name="Handle alert"/>
33  </process>
34 </definitions>
```

```

33
34         <exclusiveGateway id="exGw1" />
35
36         <sequenceFlow sourceRef="task" targetRef="exGw1" />
37         <sequenceFlow sourceRef="exGw1" targetRef="end" />
38
39         <endEvent id="end" />
40     </process>
41 </definitions>

```

8.5. 任务 Tasks

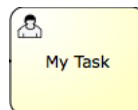
8.5.1. 用户任务 User Task

描述 Description

“用户任务”用于建模需要人工执行的任务。当流程执行到达用户任务时，会为指派至该任务的用户或组的任务列表创建一个新任务。

图示 Graphical notation

用户任务，用左上角有一个小用户图标为标准任务（圆角矩形）表示。



XML表示 XML representation

用户任务在XML中如下定义。*id*是必须属性，*name*是可选属性。

```

1 <userTask id="theTask" name="Important task" />

```

一个用户任务也可以有一个描述（description）。事实上任何BPMN 2.0元素都可以有一个描述。描述使用附加的**documentation**元素定义。

```

1 <userTask id="theTask" name="Schedule meeting" >
2   <documentation>
3     Schedule an engineering meeting for next week with the new hire.
4   </documentation>

```

描述文本可以从任务中，使用标准Java方式获取：

```

1 task.getDescription()

```

到期日期 Due Date

每个任务都有一个字段，标志该任务的到期日期。可以使用查询API，查询在给定日期前或后到期的任务。

有一个Activiti的扩展，可以在任务定义中指定表达式，以在任务创建时，设定初始到期日期。该表达式必须解析为 `java.util.Date`，`java.util.String` (ISO8601格式)，ISO8601时间长度（例如PT50M），或者 `null`。例如，可以使用在流程里前一个表单中输入的日期，或者由前一个服务任务计算出的日期。如果使用的是时间长度，则到期日期基于当前时间加上给定长度计算。例如当dueDate使用“PT30M”时，任务在从现在起30分钟后到期。

```

1 <userTask id="theTask" name="Important task" activiti:dueDate="${dateVariable}"/>

```

任务的到期日期，也可以使用 `TaskService`，或者在 `TaskListener` 中使用传递的 `DelegateTask` 修改。

用户指派 User assignment

一个用户任务可以直接指派给用户。可以通过定义**humanPerformer**子元素实现。这个**humanPerformer**定义需要**resourceAssignmentExpression**来实际定义用户。目前，只支持**formalExpressions**。

```

1 <process >
2
3   ...
4
5   <userTask id='theTask' name='important task' >
6     <humanPerformer>
7       <resourceAssignmentExpression>
8         <formalExpression>kermit</formalExpression>
9       </resourceAssignmentExpression>
10    </humanPerformer>
11  </userTask>

```


只有一个用户可被指定为任务的`humanPerformer`。在Activiti术语中，这个用户被称作办理人（**assignee**）。拥有办理人的任务，在其他人的任务列表中不可见，而可以在该办理人的个人任务列表中看到。

特定用户办理的任务，可以通过TaskService如下获取：

```
1 List<Task> tasks = taskService.createTaskQuery().taskAssignee("kermit").list();
```

任务也可以放在用户的候选任务列表中。在这个情况下，需要使用**potentialOwner**（潜在用户）结构。用法与`humanPerformer`结构类似。请注意需要为表达式中的每一个元素指定其为用户还是组（引擎无法自行判断）。

```
1 <process >
2
3   ...
4
5   <userTask id='theTask' name='important task' >
6     <potentialOwner>
7       <resourceAssignmentExpression>
8         <formalExpression>user(kermit), group(management)</formalExpression>
9       </resourceAssignmentExpression>
10    </potentialOwner>
11  </userTask>
```

定义了**potential owner**结构的任务，可用如下方法获取（或类似于指派用户任务，使用TaskQuery查询）：

```
1 List<Task> tasks = taskService.createTaskQuery().taskCandidateUser("kermit");
```

将获取所有kermit作为候选用户的任务，也就是说，表达式含有`user(kermit)`的任务，也将获取所有指派给kermit为其成员的组的任务（例如`group(management)`，如果kermit是这个组的成员，并且使用Activiti身份组件）。组会在运行时解析，并可通过IdentityService（身份服务）管理。

如果并未指定给定字符串是用户还是组，引擎默认其为组。因此下列代码与声明了`group(accountancy)`一样。

```
1 <formalExpression>accountancy</formalExpression>
```

用于任务指派的Activiti扩展 Activiti extensions for task assignment

很明显，当指派关系不复杂时，这种用户与组的指派方式十分笨重。为避免这种复杂性，可以在用户任务上使用**自定义扩展**。

- **assignee**（办理人）属性：这个自定义扩展用于直接将一个用户任务指派至一个给定用户。

```
1 <userTask id="theTask" name="my task" activiti:assignee="kermit" />
```

与使用上面定义的**humanPerformer**结构完全相同。

- **candidateUsers**（候选用户）属性：这个自定义扩展用于为一个任务指定候选用户。

```
1 <userTask id="theTask" name="my task" activiti:candidateUsers="kermit, gonzo" />
```

与使用上面定义的**potentialOwner**结构完全相同。请注意不需要像在**potential owner**中一样，使用`user(kermit)`的声明，因为这个属性只能用于用户。

- **candidateGroups**（候选组）attribute：这个自定义扩展用于为一个任务指定候选组。

```
1 <userTask id="theTask" name="my task" activiti:candidateGroups="management, accountancy" />
```

与使用上面定义的**potentialOwner**结构完全相同。请注意不需要像在**potential owner**中一样，使用`group(management)`的声明，因为这个属性只能用于组。

- **candidateUsers**与**candidateGroups**可以定义在同一个用户任务上。

请注意：尽管Activiti提供了身份管理组件，通过IdentityService暴露，但并不会检查给定的用户是否在身份组件中存在。这样Activiti在嵌入应用时，可以与已有的身份管理解决方案集成。

自定义身份联系类型（试验特性） Custom identity link types (Experimental)

[EXPERIMENTAL]

在[用户指派](#)中定义过，BPMN标准支持单个指派用户即**humanPerformer**，或者一组用户构成**potentialOwners**潜在用户池。另外，Activiti为用户任务定义了[扩展属性元素](#)，代表任务的办理人或者候选用户。

Activiti支持的身份联系类型有：

```
1 public class IdentityLinkType {
2     /* Activiti原生角色 Activiti native roles */
3     public static final String ASSIGNEE = "assignee";
4     public static final String CANDIDATE = "candidate";
5     public static final String OWNER = "owner";
6     public static final String STARTER = "starter";
7     public static final String PARTICIPANT = "participant";
8 }
```

BPMN标准与Activiti示例身份认证是用户与组。在前一章节提到过，Activiti的身份管理实现并不适用于生产环境，而需要在支持的认证概要下扩展。

如果需要添加额外的联系类型，可按照下列语法，使用自定义资源作为扩展元素：

```
1 <userTask id="theTask" name="make profit">
2     <extensionElements>
3         <activiti:customResource activiti:name="businessAdministrator">
4             <resourceAssignmentExpression>
5                 <formalExpression>user(kermit), group(management)</formalExpression>
6             </resourceAssignmentExpression>
7         </activiti:customResource>
8     </extensionElements>
9 </userTask>
```

自定义联系表达式添加至**TaskDefinition**类：

```
1 protected Map<String, Set<Expression>> customUserIdentityLinkExpressions =
2     new HashMap<String, Set<Expression>>();
3 protected Map<String, Set<Expression>> customGroupIdentityLinkExpressions =
4     new HashMap<String, Set<Expression>>();
5
6 public Map<String,
7     Set<Expression>> getCustomUserIdentityLinkExpressions() {
8     return customUserIdentityLinkExpressions;
9 }
10
11 public void addCustomUserIdentityLinkExpression(String identityLinkType,
12     Set<Expression> idList)
13     customUserIdentityLinkExpressions.put(identityLinkType, idList);
14 }
15
16 public Map<String,
17     Set<Expression>> getCustomGroupIdentityLinkExpressions() {
18     return customGroupIdentityLinkExpressions;
19 }
20
21 public void addCustomGroupIdentityLinkExpression(String identityLinkType,
22     Set<Expression> idList) {
23     customGroupIdentityLinkExpressions.put(identityLinkType, idList);
24 }
```

并将会在运行时，由**UserTaskActivityBehavior handleAssignments**方法填写。

最后，需要扩展**IdentityLinkType**类，以支持自定义身份联系类型：

```
1 package com.yourco.engine.task;
2
3 public class IdentityLinkType
4     extends org.activiti.engine.task.IdentityLinkType
5 {
6     public static final String ADMINISTRATOR = "administrator";
7
8     public static final String EXCLUDED_OWNER = "excludedOwner";
9 }
```

通过任务监听器自定义指派 Custom Assignment via task listeners

如果上面的方式仍不能满足要求，可以在创建事件（create event）上使用[任务监听器](#)，代理自定义指派逻辑：

```
1 <userTask id="task1" name="My task" >
2     <extensionElements>
3         <activiti:taskListener event="create" class="org.activiti.MyAssignmentHandler" />
4     </extensionElements>
5 </userTask>
```

```

4   </extensionElements>
5   </userTask>

```

传递至 `TaskListener` 实现的 `DelegateTask`，可用于设置办理人与候选用户/组：

```

1  public class MyAssignmentHandler implements TaskListener {
2
3      public void notify(DelegateTask delegateTask) {
4          // Execute custom identity Lookups here
5
6          // and then for example call following methods:
7          delegateTask.setAssignee("kermit");
8          delegateTask.addCandidateUser("fozzie");
9          delegateTask.addCandidateGroup("management");
10         ...
11     }
12
13 }

```

当使用Spring时，可以按上面章节的介绍使用自定义指派属性，并代理至使用[任务监听器](#)、带有[表达式](#)的Spring bean，监听任务创建事件。在下面的例子中，通过调用 `ldapService` Spring bean的 `findManagerOfEmployee` 方法，设置办理人。传递的 `emp` 参数是一个流程变量。

```

1  <userTask id="task" name="My Task" activiti:assignee="${ldapService.findManagerForEmployee(emp)}"/>

```

也可以用于候选用户与组：

```

1  <userTask id="task" name="My Task" activiti:candidateUsers="${ldapService.findAllSales()}" />

```

请注意调用方法的返回类型必须是 `String` 或 `Collection<String>`（候选用户与组）：

```

1  public class FakeLdapService {
2
3      public String findManagerForEmployee(String employee) {
4          return "Kermit The Frog";
5      }
6
7      public List<String> findAllSales() {
8          return Arrays.asList("kermit", "gonzo", "fozzie");
9      }
10
11 }

```

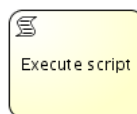
8.5.2. 脚本任务 Script Task

描述 Description

脚本任务是自动化的活动。当流程执行到达脚本任务时，会执行相应的脚本。

图示 Graphical Notation

脚本任务，用左上角有一个小“脚本”图标的标准BPMN 2.0任务（圆角矩形）表示。



XML表示 XML representation

脚本任务通过指定 `script` 与 `scriptFormat` 定义。

```

1  <scriptTask id="theScriptTask" name="Execute script" scriptFormat="groovy">
2      <script>
3          sum = 0
4          for ( i in inputArray ) {
5              sum += i
6          }
7      </script>
8  </scriptTask>

```

scriptFormat属性的值，必须是兼容[JSR-223](#)（Java 平台脚本）的名字。默认情况下，JavaScript包含在每一个JDK中，因此不需要添加任何jar。如果想使用其它（兼容JSR-223的）脚本引擎，需要在classpath中添加相应的jar，并使用适当的名字。例如，Activiti单元测试经常使用Groovy，因为其语法与Java十分相似。

请注意Groovy脚本引擎与groovy-all jar捆绑在一起。在2.0版本以前，脚本引擎是Groovy jar的一部分。因此，现在必须添加如下依赖：

```
1 <dependency>
2   <groupId>org.codehaus.groovy</groupId>
3   <artifactId>groovy-all</artifactId>
4   <version>2.x.x</version>
5 </dependency>
```

脚本中的变量 Variables in scripts

到达脚本引擎的执行可以访问的所有流程变量，都可以在脚本中使用。在这个例子里，脚本变量

```
1 <script>
2   sum = 0
3   for ( i in inputArray ) {
4     sum += i
5   }
6 </script>
```

也可以简单地调用`execution.setVariable("variableName", variableValue)`，在脚本中设置流程变量。默认情况下，变量不会自动储存（请注意，在Activiti 5.12以前是这样的！）。可以将scriptTask的autoStoreVariables参数设置为true，以自动保存任何在脚本中定义的变量（例如上例中的sum）。然而，最佳实践不是这么做，而是直接调用`execution.setVariable()`，因为在JDK近期的一些版本中，某些脚本语言不能自动保存变量。查看[这个链接](#)了解更多信息。

```
1 <scriptTask id="script" scriptFormat="JavaScript" activiti:autoStoreVariables="false">
```

这个参数的默认值为false，意味着这个参数将在脚本任务定义中忽略，所有声明的变量将只在脚本执行期间有效。

在脚本中设置变量的例子：

```
1 <script>
2   def scriptVar = "test123"
3   execution.setVariable("myVar", scriptVar)
4 </script>
```

请注意：下列名字被保留，不能用于变量名：out, out:print, lang:import, context, elcontext。

脚本结果 Script results

脚本任务的返回值，可以通过为脚本任务定义的'activiti:resultVariable'属性设置流程变量名，指定为已经存在的，或者新的流程变量。指定的已有值的流程变量，会被脚本执行的结果值覆盖。当不指定结果变量名时，脚本结果值将被忽略。

```
1 <scriptTask id="theScriptTask" name="Execute script" scriptFormat="juel" activiti:resultVariable="myVar">
2   <script>#{echo}</script>
3 </scriptTask>
```

在上面的例子中，脚本执行的结果（解析表达式#{echo}的值），将在脚本完成后，设置为名为'myVar'的流程变量。

安全性 Security

当使用javascript作为脚本语言时，可以使用“安全脚本（secure scripting）”。参见[安全脚本章节](#)。

8.5.3. Java服务任务 Java Service Task

描述 Description

Java服务任务用于执行外部的Java类。

图示 Graphical Notation

服务任务，用左上角有一个小齿轮图标的圆角矩形表示。



XML表示 XML representation

有四种方法声明如何调用Java逻辑：

- 指定实现了JavaDelegate或ActivityBehavior的类
- 对解析为代理对象的表达式求值
- 调用方法表达式
- 对值表达式求值

要指定流程执行时调用的类，需要使用**activiti:class**属性提供全限定类名（fully qualified classname）。

```
1 <serviceTask id="javaService"
2     name="My Java Service Task"
3     activiti:class="org.activiti.MyJavaDelegate" />
```

查看[实现章节](#)，了解关于如何使用这种类的更多信息。

也可以使用解析为对象的表达式。该对象必须遵循的规则，与使用**activiti:class**创建的对象规则相同（查看[更多](#)）。

```
1 <serviceTask id="serviceTask" activiti:delegateExpression="${delegateExpressionBean}" />
```

这里，**delegateExpressionBean**是一个实现了**JavaDelegate**接口的bean，在Spring容器中定义。

要指定需要计算的UEL方法表达式，使用**activiti:expression**属性。

```
1 <serviceTask id="javaService"
2     name="My Java Service Task"
3     activiti:expression="#{printer.printMessage()}" />
```

将在名为**printer**的对象上调用**printMessage**方法（不带参数）。

也可以为表达式中使用的方法传递变量。

```
1 <serviceTask id="javaService"
2     name="My Java Service Task"
3     activiti:expression="#{printer.printMessage(execution, myVar)}" />
```

将在名为**printer**的对象上调用**printMessage**方法。传递的第一个参数为**DelegateExecution**，名为**execution**，在表达式上下文中默认可用。传递的第二个参数，是当前执行中，名为**myVar**变量的值。

可以使用**activiti:expression**属性指定需要计算的UEL值表达式。

```
1 <serviceTask id="javaService"
2     name="My Java Service Task"
3     activiti:expression="#{split.ready}" />
```

会调用名为**split**的bean的**ready**参数的getter方法，**getReady**（不带参数）。该对象会被解析为执行的流程变量或（如果可用的话）Spring上下文中的bean。

实现 Implementation

要实现可以在流程执行中调用的类，需要实现**org.activiti.engine.delegate.JavaDelegate**接口，并在**execute**方法中提供所需逻辑。当流程执行到达该活动时，会执行方法中定义的逻辑，并按照BPMN 2.0的默认方法离开活动。

让我们创建一个Java类的示例，可用于将流程变量String改为大写。这个类需要实现**org.activiti.engine.delegate.JavaDelegate**接口，因此需要实现**execute(DelegateExecution)**方法。这个方法就是引擎将调用的方法，需要实现业务逻辑。可以通过**DelegateExecution**接口（点击链接获取该接口操作的详细Javadoc）访问流程实例信息，如流程变量等。

```
1 public class ToUppercase implements JavaDelegate {
2
3     public void execute(DelegateExecution execution) throws Exception {
4         String var = (String) execution.getVariable("input");
5         var = var.toUpperCase();
6         execution.setVariable("input", var);
7     }
8
9 }
```

请注意：只会为**serviceTask**上定义的**Java**类创建一个实例。所有流程实例共享同一个类实例，用于调用**execute(DelegateExecution)**。这意味着该类不能有任何成员变量，并需要是线程安全的，因为它可能会在不同线程中同时执行。这也影响了**字段注入**的使用方法。（译者注：原文可能较老，不正确。5.21中，**activiti:class**指定的类，会在流程实例启动时，为每个活动，分别进行实例化。不过，当该活动在流程中重复执行，或者为多实例时，使用的都会是同一个类实例。）

在流程定义中（如通过**activiti:class**）引用的类，不会在部署时实例化。只有当流程执行第一次到达该类使用的地方时，才会创建该类的实例。如果找不到这个类，会抛出**ActivitiException**。这是因为部署时的环境（更准确的说**classpath**），与实际运行的环境经常不一样。例如当使用**ant**或者**Activiti Explorer**中业务存档上传的方式部署的流程，其**classpath**中并没有流程引用的类。

[内部：非公有实现类]也可以使用实现了**org.activiti.engine.impl.pvm.delegate.ActivityBehavior**接口的类。该实现可以访问更强大的**ActivityExecution**，可以例如影响流程的控制流程。请注意这并不是很好的实践，需要避免这么使用。因此，建议只有在高级使用场景下，并且你确知在做什么的时候，才使用**ActivityBehavior**接口。

字段注入 Field Injection

可以为代理类的字段注入值。支持下列注入方式：

- 字符串常量
- 表达式

如果可以的话，会按照Java Bean命名约定（例如，**firstName**成员使用setter **setFirstName(...)**），通过代理类的公有setter方法，注入变量。如果该字段没有可用的setter，会直接设置该代理类的私有成员的值。有的环境中，**SecurityManagers**不允许修改私有字段，因此为想要注入的字段，暴露一个公有setter方法，是更安全的做法。

不论在流程定义中声明的是什么类型的值，注入对象的**setter**/私有字段的类型，总是**org.activiti.engine.delegate.Expression**。解析表达式后，可以被转型为合适的类型。

当使用'**activiti:class**'属性时，支持字段注入。也可以在使用'**activiti:delegateExpression**'属性时，进行字段注入，然而因为线程安全的考虑，需要有特殊的规则（参见下一章节）。

下面的代码片段展示了如何为类中声明的字段注入常量值。请注意按照BPMN 2.0 XML概要的要求，在实际字段注入声明前，需要先声明'**extensionElements**'XML元素。

```
1 <serviceTask id="javaService"
2   name="Java service invocation"
3   activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
4   <extensionElements>
5     <activiti:field name="text" stringValue="Hello World" />
6   </extensionElements>
7 </serviceTask>
```

ToUpperCaseFieldInjected类有一个字段**text**，为**org.activiti.engine.delegate.Expression**类型。当调用**text.getValue(execution)**时，会返回配置的字符串**Hello World**：

```
1 public class ToUpperCaseFieldInjected implements JavaDelegate {
2
3   private Expression text;
4
5   public void execute(DelegateExecution execution) {
6     execution.setVariable("var", ((String)text.getValue(execution)).toUpperCase());
7   }
8
9 }
```

另外，对于较长文本（例如邮件内容），可以使用'**activiti:string**'子元素：

```
1 <serviceTask id="javaService"
2   name="Java service invocation"
3   activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
4   <extensionElements>
5     <activiti:field name="text">
6       <activiti:string>
7         This is a long string with a lot of words and potentially way longer even!
8       </activiti:string>
9     </activiti:field>
10  </extensionElements>
11 </serviceTask>
```

要在运行时动态解析注入的值，可以使用表达式。这种表达式可以使用流程变量，或者Spring定义的bean（如果使用Spring）。像**服务任务实现**中提到的，当服务任务中使用**activiti:class**属性时，该Java类的实例在所有流程实例中共享。要动态地为字段注入值，可以在**org.activiti.engine.delegate.Expression**中注入值或方法表达式，它们会通过**execute**方法传递的**DelegateExecution**计算/调用。

下面的示例类，使用了注入的表达式，并使用当前的 `DelegateExecution` 解析它们。调用 `generBean` 方法时传递的是 `gender` 变量。完整的代码与测试可以在 `org.activiti.examples.bpmn.servicetask.JavaServiceTaskTest.testExpressionFieldInjection` 中找到

```
1 <serviceTask id="javaService" name="Java service invocation"
2   activiti:class="org.activiti.examples.bpmn.servicetask.ReverseStringsFieldInjected">
3
4   <extensionElements>
5     <activiti:field name="text1">
6       <activiti:expression>${genderBean.getGenderString(gender)}</activiti:expression>
7     </activiti:field>
8     <activiti:field name="text2">
9       <activiti:expression>Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}</activiti:expression>
10    </activiti:field>
11  </extensionElements>
12 </serviceTask>
```

```
1 public class ReverseStringsFieldInjected implements JavaDelegate {
2
3   private Expression text1;
4   private Expression text2;
5
6   public void execute(DelegateExecution execution) {
7     String value1 = (String) text1.getValue(execution);
8     execution.setVariable("var1", new StringBuffer(value1).reverse().toString());
9
10    String value2 = (String) text2.getValue(execution);
11    execution.setVariable("var2", new StringBuffer(value2).reverse().toString());
12  }
13 }
```

另外，为避免XML太过冗长，可以将表达式设置为属性，而不是子元素。

```
1 <activiti:field name="text1" expression="${genderBean.getGenderString(gender)}" />
2 <activiti:field name="text1" expression="Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}" />
```

字段注入与线程安全 Field injection and thread safety

总的来说，在服务任务中使用Java代理与字段注入是线程安全的。然而，有些情况下不能保证线程安全，取决于设置，或Activiti运行所在的环境。

当使用 `activiti:class` 属性时，使用字段注入总是线程安全的（译者注：仍不完全安全，如对于多实例服务任务，使用的是同一个类实例）。对于引用了某个类的每一个服务任务，都会实例化新的实例，并且在创建实例时注入一次字段。在不同的任务或流程定义中多次使用同一个类没有问题。

当使用 `activiti:expression` 属性时，不能使用字段注入。只能通过方法调用传递变量，并且这总是线程安全的。

当使用 `activiti:delegateExpression` 属性时，代理实例的线程安全性，取决于表达式解析的方式。如果该代理表达式在多个任务与/或流程定义中重复使用，并且表达式总是返回相同的示例，则字段注入不是线程安全的。让我们看几个例子。

假设表达式为 `${factory.createDelegate(someVariable)}`，其中 `factory` 为引擎可用的Java bean（例如使用Spring集成时的Spring bean），并在每次表达式解析时，创建新的实例。这种情况下，使用字段注入时，没有线程安全性问题：每次表达式解析时，新实例的字段都会注入。

然而，如果表达式为 `${someJavaDelegateBean}`，解析为 `JavaDelegate` 的实现，并且在创建单例的环境（如Spring）中运行。当在不同的任务和/或流程定义中使用这个表达式时，表达式总会解析为相同的实例。这种情况下，使用字段注入不是线程安全的。例如：

```
1 <serviceTask id="serviceTask1" activiti:delegateExpression="${someJavaDelegateBean}">
2   <extensionElements>
3     <activiti:field name="someField" expression="${input * 2}"/>
4   </extensionElements>
5 </serviceTask>
6
7 <!-- other process definition elements -->
8
9 <serviceTask id="serviceTask2" activiti:delegateExpression="${someJavaDelegateBean}">
10  <extensionElements>
11    <activiti:field name="someField" expression="${input * 2000}"/>
12  </extensionElements>
13 </serviceTask>
```

这段示例代码有两个服务任务，使用同一个代理表达式，但是 `expression` 字段填写不同的值。如果该表达式解析为相同的实例，就会在并发场景下，注入 `someField` 字段时出现竞争条件。

最简单的解决方案，为

- 重写Java代理，以使用表达式，并将所需数据通过方法参数传递给代理。
- 或者，在每次代理表达式解析时，返回代理类的新实例。这意味着这个bean的scope（范围）必须是**prototype**（原型）（例如在代理类上加上@Scope(SCOPE_PROTOTYPE)注解）。

在Activiti 5.21版本中，可以通过配置流程引擎配置，禁用代理表达式上使用字段注入。需要设置`delegateExpressionFieldInjectionMode`参数（取`org.activiti.engine.impl.cfg.DelegateExpressionFieldInjectionMode`枚举中的值）。

可使用下列选项：

- **DISABLED**（禁用）：当使用代理表达式时，完全禁用字段注入。不会再尝试进行字段注入。这是最安全的方式，保证线程安全。
- **COMPATIBILITY**（兼容）：在这个模式下，行为与5.21版本之前完全一样：可以在代理表达式中使用字段注入，如果代理类中没有定义该字段，会抛出异常。这是最不线程安全的模式，但可以保证历史版本兼容性，也可以在代理表达式只在一个任务中使用的时候（因此不会产生并发竞争条件），安全使用。
- **MIXED**（混合）：可以在使用代理表达式时注入，但当代理中没有定义字段时，不会抛出异常。这样可以在部分代理中使用注入（例如不是单例时），而在部分代理中不使用注入。
- Activiti 5.x版本的默认模式为**COMPATIBILITY**（兼容）。
- Activiti 6.x版本的默认模式为**MIXED**（混合）。

例如，假设使用**MIXED**模式，并使用Spring集成，在Spring配置中定义了如下bean：

```
1 <bean id="singletonDelegateExpressionBean"
2   class="org.activiti.spring.test.fieldinjection.SingletonDelegateExpressionBean" />
3
4 <bean id="prototypeDelegateExpressionBean"
5   class="org.activiti.spring.test.fieldinjection.PrototypeDelegateExpressionBean"
6   scope="prototype" />
```

第一个bean是一般的Spring bean，因此是单例的。第二个的scope为**prototype**，因此每次请求这个bean时，Spring容器都会返回一个新实例。

在以下流程定义中：

```
1 <serviceTask id="serviceTask1" activiti:delegateExpression="${prototypeDelegateExpressionBean}">
2   <extensionElements>
3     <activiti:field name="fieldA" expression="${input * 2}" />
4     <activiti:field name="fieldB" expression="${1 + 1}" />
5     <activiti:field name="resultVariableName" stringValue="resultServiceTask1" />
6   </extensionElements>
7 </serviceTask>
8
9 <serviceTask id="serviceTask2" activiti:delegateExpression="${prototypeDelegateExpressionBean}">
10  <extensionElements>
11    <activiti:field name="fieldA" expression="${123}" />
12    <activiti:field name="fieldB" expression="${456}" />
13    <activiti:field name="resultVariableName" stringValue="resultServiceTask2" />
14  </extensionElements>
15 </serviceTask>
16
17 <serviceTask id="serviceTask3" activiti:delegateExpression="${singletonDelegateExpressionBean}">
18  <extensionElements>
19    <activiti:field name="fieldA" expression="${input * 2}" />
20    <activiti:field name="fieldB" expression="${1 + 1}" />
21    <activiti:field name="resultVariableName" stringValue="resultServiceTask1" />
22  </extensionElements>
23 </serviceTask>
24
25 <serviceTask id="serviceTask4" activiti:delegateExpression="${singletonDelegateExpressionBean}">
26  <extensionElements>
27    <activiti:field name="fieldA" expression="${123}" />
28    <activiti:field name="fieldB" expression="${456}" />
29    <activiti:field name="resultVariableName" stringValue="resultServiceTask2" />
30  </extensionElements>
31 </serviceTask>
```

有四个服务任务，第一、二个使用`${prototypeDelegateExpressionBean}`代理表达式，第三、四个使用`${singletonDelegateExpressionBean}`代理表达式。

先看原型bean：

```
1 public class PrototypeDelegateExpressionBean implements JavaDelegate {
2
3   public static AtomicInteger INSTANCE_COUNT = new AtomicInteger(0);
4 }
```

```

5     private Expression fieldA;
6     private Expression fieldB;
7     private Expression resultVariableName;
8
9     public PrototypeDelegateExpressionBean() {
10         INSTANCE_COUNT.incrementAndGet();
11     }
12
13     @Override
14     public void execute(DelegateExecution execution) throws Exception {
15
16         Number fieldAValue = (Number) fieldA.getValue(execution);
17         Number fieldValueB = (Number) fieldB.getValue(execution);
18
19         int result = fieldAValue.intValue() + fieldValueB.intValue();
20         execution.setVariable(resultVariableName.getValue(execution).toString(), result);
21     }
22
23 }

```

在运行上面流程定义的一个流程实例后，检查`INSTANCE_COUNT`，会得到2。这是因为每次`${prototypeDelegateExpressionBean}`解析时，都会创建新实例。可以看到三个`Expression`成员字段的注入没有任何问题。

而在原型bean中，有一点区别：

```

1 public class SingletonDelegateExpressionBean implements JavaDelegate {
2
3     public static AtomicInteger INSTANCE_COUNT = new AtomicInteger(0);
4
5     public SingletonDelegateExpressionBean() {
6         INSTANCE_COUNT.incrementAndGet();
7     }
8
9     @Override
10    public void execute(DelegateExecution execution) throws Exception {
11
12        Expression fieldAExpression = DelegateHelper.getFieldExpression(execution, "fieldA");
13        Number fieldA = (Number) fieldAExpression.getValue(execution);
14
15        Expression fieldBExpression = DelegateHelper.getFieldExpression(execution, "fieldB");
16        Number fieldB = (Number) fieldBExpression.getValue(execution);
17
18        int result = fieldA.intValue() + fieldB.intValue();
19
20        String resultVariableName = DelegateHelper.getFieldExpression(execution,
21 "resultVariableName").getValue(execution).toString();
22        execution.setVariable(resultVariableName, result);
23    }
24 }

```

`INSTANCE_COUNT`总是1，因为是单例模式。在这个代理中，没有`Expression`成员字段。因为我们使用的是`MIXED`模式，可以这样用。而在`COMPATIBILITY`模式下，就会抛出异常，因为需要有成员字段。这个bean也可以使用`DISABLED`模式，但会禁用上面进行了字段注入的原型bean。

在代理的代码里，使用了`org.activiti.engine.delegate.DelegateHelper`。它提供了一些有用的工具方法，用于执行相同的逻辑，并且在单例中是线程安全的。与注入`Expression`不同，它通过`getFieldExpression`读取。这意味着在服务任务的XML里，字段定义与单例bean完全相同。查看上面的XML代码，可以看到定义是相同的，只是实现逻辑不同。

（技术提示：`getFieldExpression`直接读取BpmnModel，并在方法执行时创建表达式，因此是线程安全的）。

- 在Activiti 5.x版本中，（由于架构缺陷）不能在`ExecutionListener`或`TaskListener`中使用`DelegateHelper`。要保证监听器的线程安全，仍需使用表达式，或确保每次解析代理表达式时，都创建新实例。
- 在Activiti 6.x版本中，在`ExecutionListener`或`TaskListener`中可以使用`DelegateHelper`。例如在6.x版本中，下列代码可以使用`DelegateHelper`：

```

1 <extensionElements>
2   <activiti:executionListener
3     delegateExpression="${testExecutionListener}" event="start">
4     <activiti:field name="input" expression="${startValue}" />
5     <activiti:field name="resultVar" stringValue="processStartValue" />
6   </activiti:executionListener>
7 </extensionElements>

```

其中`testExecutionListener`解析为`ExecutionListener`接口的一个实现的实例：

```
@Component("testExecutionListener")
```

```

1 public class TestExecutionListener implements ExecutionListener {
2
3     @Override
4     public void notify(DelegateExecution execution) {
5         Expression inputExpression = DelegateHelper.getFieldExpression(execution, "input");
6         Number input = (Number) inputExpression.getValue(execution);
7
8         int result = input.intValue() * 100;
9
10        Expression resultVarExpression = DelegateHelper.getFieldExpression(execution, "resultVar");
11        execution.setVariable(resultVarExpression.getValue(execution).toString(), result);
12    }
13
14 }
15

```

服务任务的结果 Service task results

服务执行的返回值（仅对使用表达式的服务任务），可以通过为脚本任务定义的'**activiti:resultVariable**'属性设置流程变量名，指定为已经存在的，或者新的流程变量。指定的已有值的流程变量，会被服务执行的结果值覆盖。当不指定结果变量名时，服务执行的结果值将被忽略。

```

1 <serviceTask id="aMethodExpressionServiceTask"
2     activiti:expression="#{myService.doSomething()}"
3     activiti:resultVariable="myVar" />

```

在上例中，服务执行的结果（流程变量或Spring bean中，使用'**myService**'名字获取的对象，调用'**doSomething()**'方法的返回值），在服务执行完成后，会设置为名为'**myVar**'的流程变量。

处理异常 Handling exceptions

当执行自定义逻辑时，通常需要捕获特定的业务异常，并在流程中处理。Activiti提供了不同的方法。

抛出BPMN错误 Throwing BPMN Errors

可以在服务任务或脚本任务的用户代码中抛出BPMN错误。要这么做，可以在Java代理、脚本、表达式与代理表达式中，抛出特殊的**ActivitiException**，叫做**BpmnError**。引擎会捕获这个异常，并将其转发至合适的错误处理器，例如异常边界事件，或者错误事件子程序。

```

1 public class ThrowBpmnErrorDelegate implements JavaDelegate {
2
3     public void execute(DelegateExecution execution) throws Exception {
4         try {
5             executeBusinessLogic();
6         } catch (BusinessException e) {
7             throw new BpmnError("BusinessExceptionOccurred");
8         }
9     }
10
11 }

```

构造函数的参数是错误代码，将被用于决定处理这个错误的错误处理器。参见[错误边界事件](#)了解如何捕获BPMN错误。

这个机制只应该用于业务错误，需要通过流程中定义的错误边界事件或错误事件子流程处理。技术错误应该通过其他异常类型表现，并且通常不在流程内部处理。

异常映射 Exception mapping

也可以使用**mapException**扩展，直接将Java异常映射至业务异常（错误）。单一映射是最简单的格式：

```

1 <serviceTask id="servicetask1" name="Service Task" activiti:class="...">
2     <extensionElements>
3         <activiti:mapException
4             errorCode="myErrorCode1">org.activiti.SomeException</activiti:mapException>
5     </extensionElements>
6 </serviceTask>

```

在上面的代码中，如果服务任务抛出了**org.activiti.SomeException**的实例，则会被捕获，并被转换为带有给定**errorCode**的BPMN异常（错误）。从这里开始，可以与普通BPMN异常（错误）完全一样地处理。

其他异常会依照没有映射被处理，将传播至API调用者。

也可以在一行中，使用**includeChildExceptions**属性，映射特定异常的所有子异常。

```

1 <serviceTask id="servicetask1" name="Service Task" activiti:class="...">
2     <extensionElements>

```

```

3      <activiti:mapException errorCode="myErrorCode1"
4          includeChildExceptions="true">org.activiti.SomeException</activiti:mapException>
5      </extensionElements>
6  </serviceTask>

```

上面的代码中，Activiti会将任何直接或间接的 `SomeException` 的子类，转换为带有给定错误代码的BPMN错误。当未指定 `includeChildExceptions` 时，视为“false”。

最普通的是默认映射。默认映射是一个没有类的映射，可以匹配任何Java异常：

```

1  <serviceTask id="servicetask1" name="Service Task" activiti:class="...">
2      <extensionElements>
3          <activiti:mapException errorCode="myErrorCode1"/>
4      </extensionElements>
5  </serviceTask>

```

映射会按照顺序检查，从上至下，使用第一个匹配的映射，除了默认映射。默认映射将只在所有映射都不能成功匹配时使用。只有第一个没有类的映射会当做默认映射处理。默认映射忽略 `includeChildExceptions`。

异常顺序流 Exception Sequence Flow

[内部：非公有实现类]

也可以选择在发生异常时，将流程执行路由至另一条路径。下面的例子展示了如何做。

```

1  <serviceTask id="javaService"
2      name="Java service invocation"
3      activiti:class="org.activiti.ThrowsExceptionBehavior">
4  </serviceTask>
5
6  <sequenceFlow id="no-exception" sourceRef="javaService" targetRef="theEnd" />
7  <sequenceFlow id="exception" sourceRef="javaService" targetRef="fixException" />

```

在这里，这个服务任务具有两条出口顺序流，分别称为 `exception` 与 `no-exception`。这些顺序流id会在发生异常时，用于控制流程流向：

```

1  public class ThrowsExceptionBehavior implements ActivityBehavior {
2
3      public void execute(ActivityExecution execution) throws Exception {
4          String var = (String) execution.getVariable("var");
5
6          PvmTransition transition = null;
7          try {
8              executeLogic(var);
9              transition = execution.getActivity().findOutgoingTransition("no-exception");
10         } catch (Exception e) {
11             transition = execution.getActivity().findOutgoingTransition("exception");
12         }
13         execution.take(transition);
14     }
15 }
16 }

```

在JavaDelegate中使用Activiti服务 Using an Activiti service from within a JavaDelegate

有的时候，需要在Java服务任务中使用Activiti服务（例如当调用活动不符合需求时，通过RuntimeService启动流程实例）。`org.activiti.engine.delegate.DelegateExecution`可以方便地通过`org.activiti.engine.EngineServices`接口使用这些服务：

```

1  public class StartProcessInstanceTestDelegate implements JavaDelegate {
2
3      public void execute(DelegateExecution execution) throws Exception {
4          RuntimeService runtimeService = execution.getEngineServices().getRuntimeService();
5          runtimeService.startProcessInstanceByKey("myProcess");
6      }
7
8  }

```

通过这个接口可以访问所有Activiti服务API。

使用这些API调用造成的所有数据变更，都处在当前事务中。在具有依赖注入的环境，如Spring或CDI中，使用或不使用激活JTA的数据源，也都可以使用。例如，下面的代码片段与上面的代码具有相同功能，但RuntimeService是通过注入而不是通过`org.activiti.engine.EngineServices`接口获得的。

```
@Component("startProcessInstanceDelegate")
```

```

1 public class StartProcessInstanceTestDelegateWithInjection {
2
3     @Autowired
4     private RuntimeService runtimeService;
5
6     public void startProcess() {
7         runtimeService.startProcessInstanceByKey("oneTaskProcess");
8     }
9
10 }
11

```

重要技术提示：在当前事务中进行的服务调用，产生或修改的数据是在服务任务执行前完成的，因此更改还未刷入数据库。所有API调用都通过处理数据库数据而生效，这意味着这些未提交的修改在服务任务的API调用中“不可见”。

8.5.4. Web服务任务 Web Service Task

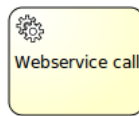
[EXPERIMENTAL]

描述 Description

Web服务任务用于同步调用外部的Web服务。

图示 Graphical Notation

Web服务任务，与Java服务任务显示地一样。



XML表示 XML representation

要使用Web服务，需要导入其操作，以及复杂的类型。通过使用指向Web服务的WSDL的导入标签（import tag），可以自动完成这些：

```

1 <import importType="http://schemas.xmlsoap.org/wsdl/"
2         location="http://localhost:63081/counter?wsdl"
3         namespace="http://webservice.activiti.org/" />

```

上面的声明告知Activiti导入定义，但并不创建条目定义（item definition）与消息。假设我们需要调用一个名为‘prettyPrint’的方法，我们需要为请求与回复消息，创建相应的消息与条目定义：

```

1 <message id="prettyPrintCountRequestMessage" itemRef="tns:prettyPrintCountRequestItem" />
2 <message id="prettyPrintCountResponseMessage" itemRef="tns:prettyPrintCountResponseItem" />
3
4 <itemDefinition id="prettyPrintCountRequestItem" structureRef="counter:prettyPrintCount" />
5 <itemDefinition id="prettyPrintCountResponseItem" structureRef="counter:prettyPrintCountResponse" />

```

在声明服务任务前，需要定义实际引用Web服务的BPMN接口与操作。基本上，是定义“接口”与所需的“操作”。我们对每一个操作都重复使用之前定义的传入与传出消息。例如，下面的声明定义了“counter”接口，与“prettyPrintCountOperation”操作：

```

1 <interface name="Counter Interface" implementationRef="counter:Counter">
2     <operation id="prettyPrintCountOperation" name="prettyPrintCount Operation"
3         implementationRef="counter:prettyPrintCount">
4         <inMessageRef>tns:prettyPrintCountRequestMessage</inMessageRef>
5         <outMessageRef>tns:prettyPrintCountResponseMessage</outMessageRef>
6     </operation>
7 </interface>

```

现在可以通过使用##WebService实现，声明Web服务任务，并引用Web服务操作。

```

1 <serviceTask id="webService"
2     name="Web service invocation"
3     implementation="##WebService"
4     operationRef="tns:prettyPrintCountOperation">

```

Web服务任务IO规范 Web Service Task IO Specification

除非使用简化方法处理输入与输出数据关联（见下），否则需要为每个Web服务任务声明IO规范，指出任务的输入与输出是什么。这个方法很简单，也兼容BPMN 2.0。在prettyPrint例子中，根据之前声明的条目定义，定义输入与输出：

```
<ioSpecification>
```

```

1      <dataInput itemSubjectRef="tns:prettyPrintCountRequestItem" id="dataInputOfServiceTask" />
2      <dataOutput itemSubjectRef="tns:prettyPrintCountResponseItem" id="dataOutputOfServiceTask" />
3      <inputSet>
4          <dataInputRefs>dataInputOfServiceTask</dataInputRefs>
5      </inputSet>
6      <outputSet>
7          <dataOutputRefs>dataOutputOfServiceTask</dataOutputRefs>
8      </outputSet>
9  </ioSpecification>
10

```

Web服务任务数据输入关联 Web Service Task data input associations

有两种指定数据输入关联的方式：

- 使用表达式
- 使用简化方法

要使用表达式指定数据输入关联，需要定义条目的源与目标，并指定每个条目字段的关联。下面的例子中我们指定了条目的`prefix`与`suffix`字段：

```

1  <dataInputAssociation>
2      <sourceRef>dataInputOfProcess</sourceRef>
3      <targetRef>dataInputOfServiceTask</targetRef>
4      <assignment>
5          <from>${dataInputOfProcess.prefix}</from>
6          <to>${dataInputOfServiceTask.prefix}</to>
7      </assignment>
8      <assignment>
9          <from>${dataInputOfProcess.suffix}</from>
10         <to>${dataInputOfServiceTask.suffix}</to>
11     </assignment>
12 </dataInputAssociation>

```

另一方面，也可以使用简化方法。`'sourceRef'`元素是一个Activiti变量名，而`'targetRef'`是条目定义的参数。在下面的例子中，将`'PrefixVariable'`变量的值关联至`'prefix'`字段，并将`'SuffixVariable'`变量的值关联至`'suffix'`字段。

```

1  <dataInputAssociation>
2      <sourceRef>PrefixVariable</sourceRef>
3      <targetRef>prefix</targetRef>
4  </dataInputAssociation>
5  <dataInputAssociation>
6      <sourceRef>SuffixVariable</sourceRef>
7      <targetRef>suffix</targetRef>
8  </dataInputAssociation>

```

Web服务任务数据输出关联 Web Service Task data output associations

有两种指定数据输出关联的方式：

- 使用表达式
- 使用简化方法

要使用表达式指定数据输出关联，需要定义目标变量与源表达式。这种方法很直接，与数据输入关联类似：

```

1  <dataOutputAssociation>
2      <targetRef>dataOutputOfProcess</targetRef>
3      <transformation>${dataOutputOfServiceTask.prettyPrint}</transformation>
4  </dataOutputAssociation>

```

另一方面，也可以使用简化方法。`'sourceRef'`是条目定义的参数，而`'targetRef'`元素是一个Activiti变量名。这种方法很直接，与数据输入关联类似：

```

1  <dataOutputAssociation>
2      <sourceRef>prettyPrint</sourceRef>
3      <targetRef>OutputVariable</targetRef>
4  </dataOutputAssociation>

```

8.5.5. 业务规则任务 Business Rule Task

[EXPERIMENTAL]

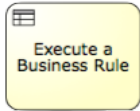
描述 Description

业务规则任务用于同步执行一条或多条规则。**Activiti**使用名为**Drools Expert**的**Drools**规则引擎执行业务规则。目前，业务规则中包含的**.drl**文件，必须与定义了业务规则服务并执行规则的流程定义，一起部署。这意味着流程中使用的所有**.drl**文件都需要打包在流程**BAR**文件中，与任务表单类似。要了解为**Drools Expert**创建业务规则的更多信息，请访问位于**JBoss Drools**的**Drools**文档。

如果想要插入自己的规则任务实现，例如，希望通过不同方法使用**Drools**，或者想使用完全不同的规则引擎，则可以使用**BusinessRuleTask**的**class**或**expression**属性。这样它会与**服务任务**的行为完全相同。

图示 Graphical Notation

业务规则任务，显示为带有表格图标的圆角矩形。



XML表示 XML representation

要执行一条或多条，与流程定义在同一个**BAR**文件中部署的业务规则，需要定义输入与结果变量。输入变量可以用流程变量的列表定义，使用逗号分隔。输出变量只能有一个变量名，将执行业务规则数处对象存储至流程变量。请注意结果变量会包含对象的**list**。如果没有指定结果变量名，会使用默认的 **org.activiti.engine.rules.OUTPUT**。

下面的业务规则任务，执行与流程定义一起部署的所有业务规则：

```
1 <process id="simpleBusinessRuleProcess">
2
3   <startEvent id="theStart" />
4   <sequenceFlow sourceRef="theStart" targetRef="businessRuleTask" />
5
6   <businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="{order}"
7     activiti:resultVariable="rulesOutput" />
8
9   <sequenceFlow sourceRef="businessRuleTask" targetRef="theEnd" />
10
11  <endEvent id="theEnd" />
12
13 </process>
```

也可以将业务规则任务配置为只执行部署的**.drl**文件中的一组规则。要做到这一点，需要指定规则名字的列表，用逗号分隔。

```
1 <businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="{order}"
2   activiti:rules="rule1, rule2" />
```

这个例子中只会执行**rule1**与**rule2**。

也可以定义需要从执行中排除的规则列表。 **execution**。

```
1 <businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="{order}"
2   activiti:rules="rule1, rule2" exclude="true" />
```

这个例子中与流程定义一起部署在同一个**BAR**文件中的所有规则都会被执行，除了**rule1**与**rule2**。

前面提到过，还可以自行处理**BusinessRuleTask**的实现：

```
1 <businessRuleTask id="businessRuleTask" activiti:class="{MyRuleServiceDelegate}" />
```

这样业务规则任务与服务任务的行为完全一样，但仍保持业务规则任务的图标，显示在这里处理业务规则。

8.5.6. 邮件任务 Email Task

Activiti可以通过自动邮件服务任务，增强业务流程。可以向一个或多个收信人发送邮件，支持**cc**，**bcc**，**HTML**内容，等等。请注意邮件任务不是**BPMN 2.0**规范的“官方”任务（因此也没有专用图标）。因此，在**Activiti**中，邮件任务实现为一种特殊的服务任务。

邮件服务器配置 Mail server configuration

Activiti引擎通过支持**SMTP**的外部邮件服务器发送邮件。要发送邮件，引擎需要了解如何连接邮件服务器。可以在**activiti.cfg.xml**配置文件中设置下面的参数：

参数	必填？	描述
mailServerHost	否	邮件服务器的主机名（如 mail.mycorp.com）。默认为 localhost

参数	必填?	描述
mailServerPort	是，如果不使用默认端口	邮件服务器的SMTP端口。默认值为25
mailServerDefaultFrom	否	若用户没有提供地址，默认使用的邮件发件人地址。默认为 <code>activiti@activiti.org</code>
mailServerUsername	若服务器需要	部分邮件服务器发信时需要进行认证。默认为空。
mailServerPassword	若服务器需要	部分邮件服务器发信时需要进行认证。默认为空。
mailServerUseSSL	若服务器需要	部分邮件服务器要求ssl通信。默认设置为false。
mailServerUseTLS	若服务器需要	部分邮件服务器要求TLS通信（例如gmail）。默认设置为false。

定义邮件任务 Defining an Email Task

邮件任务实现为特殊的[服务任务](#)，通过将服务任务的`type`定义为`'mail'`设置。

```
1 <serviceTask id="sendMail" activiti:type="mail">
```

邮件任务通过[字段注入](#)配置。这些参数的值可以使用EL表达式，将在流程执行运行时解析。可以设置下列参数：

参数	必填?	描述
to	是	邮件的收信人。可以使用逗号分隔的列表定义多个接收人
from	否	邮件的发信人地址。如果不设置，会使用 默认配置 的地址
cc	否	邮件的抄送人。可以使用逗号分隔的列表定义多个接收人
bcc	否	邮件的密送人。可以使用逗号分隔的列表定义多个接收人
charset	否	可以修改邮件的字符集，对许多非英语语言很必要。
html	否	邮件的HTML内容
text	否	邮件的内容，普通非富文本的邮件。对于不支持富文本内容的客户端，可以与 <code>html</code> 一起使用。客户端会退回为纯文本格式。
htmlVar	否	存储邮件HTML内容的流程变量名。与 <code>html</code> 参数的最大区别，是这个参数会在邮件任务发送前，使用其内容进行表达式替换。
textVar	否	存储邮件纯文本内容的流程变量名。与 <code>text</code> 参数的最大区别，是这个参数会在邮件任务发送前，使用其内容进行表达式替换。
ignoreException	否	处理邮件时的失败，是否抛出 <code>ActivitiException</code> 。默认设置为false。
exceptionVariableName	否	当处理邮件时的失败，由于 <code>ignoreException = true</code> 设置而不会抛出异常，则使用给定名字的变量保存失败信息

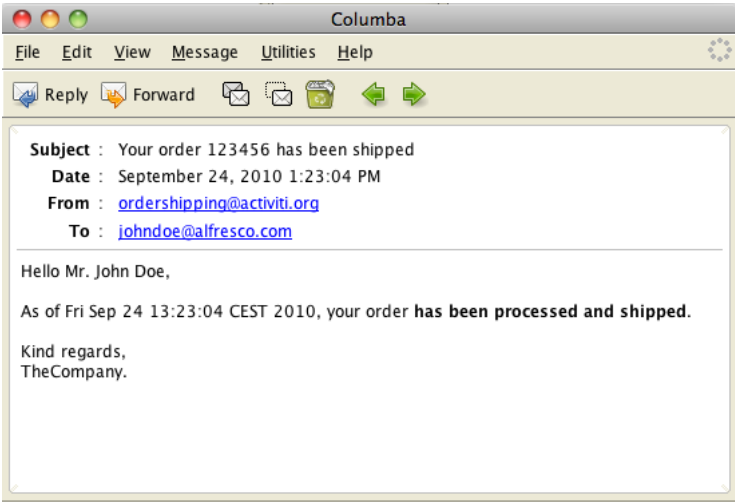
使用示例 Example usage

参数	必填?	描述
----	-----	----

下面的XML代码片段展示了使用邮件任务的示例。

```
1 <serviceTask id="sendMail" activiti:type="mail">
2   <extensionElements>
3     <activiti:field name="from" stringValue="order-shipping@thecompany.com" />
4     <activiti:field name="to" expression="${recipient}" />
5     <activiti:field name="subject" expression="Your order ${orderId} has been shipped" />
6     <activiti:field name="html">
7       <activiti:expression>
8         <![CDATA[
9           <html>
10            <body>
11              Hello ${male ? 'Mr.' : 'Mrs.' } ${recipientName},<br/><br/>
12
13              As of ${now}, your order has been <b>processed and shipped</b>.<br/><br/>
14
15              Kind regards,<br/>
16
17              TheCompany.
18            </body>
19          </html>
20        ]]>
21      </activiti:expression>
22    </activiti:field>
23  </extensionElements>
24 </serviceTask>
```

产生如下结果:



8.5.7. Mule任务 Mule Task

Mule任务可以向Mule发送消息，增强Activiti的集成特性。请注意Mule任务不是BPMN 2.0规范的“官方”任务（因此也没有专用图标）。因此，在Activiti中，Mule任务实现为一种特殊的服务任务。

定义Mule任务 Defining an Mule Task

Mule任务实现为特殊的服务任务，通过将服务任务的type定义为'mule'设置。

```
1 <serviceTask id="sendMule" activiti:type="mule">
```

Mule任务通过字段注入配置。这些参数的值可以使用EL表达式，将在流程执行运行时解析。可以设置下列参数:

参数	必填?	描述
endpointUrl	是	希望调用的Mule终端（endpoint）。
language	是	计算payloadExpression字段所用的语言。
payloadExpression	是	消息的载荷表达式
resultVariable	否	存储调用结果的变量名。

使用示例 Example usage

下面的XML代码片段展示了使用Mule任务的示例。

```

1 <extensionElements>
2   <activiti:field name="endpointUrl">
3     <activiti:string>vm://in</activiti:string>
4   </activiti:field>
5   <activiti:field name="language">
6     <activiti:string>juel</activiti:string>
7   </activiti:field>
8   <activiti:field name="payloadExpression">
9     <activiti:string>"hi"</activiti:string>
10  </activiti:field>
11  <activiti:field name="resultVariable">
12    <activiti:string>theVariable</activiti:string>
13  </activiti:field>
14 </extensionElements>

```

8.5.8. Camel任务 Camel Task

Camel任务可以向Mule发送与接收消息，增强Activiti的集成特性。请注意Camel任务不是BPMN 2.0规范的“官方”任务（因此也没有专用图标）。因此，在Activiti中，Camel任务实现为一种特殊的服务任务。还请注意要使用Camel任务功能，需要在项目中包含Activiti Camel模块。

定义Camel任务 Defining a Camel Task

Camel任务实现为特殊的**服务任务**，通过将服务任务的`type`定义为'camel'设置。

```

1 <serviceTask id="sendCamel" activiti:type="camel">

```

流程定义本身只需要在服务任务上定义Camel类型。集成逻辑都通过Camel容器代理。默认情况下Activiti引擎在Spring容器中查找camelContext bean。camelContext bean定义了由Camel容器装载的Camel路由。在下面的例子中，路由通过给定的Java包装载，但也可以自行在Spring配置中直接定义路由。

```

1 <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
2   <packageScan>
3     <package>org.activiti.camel.route</package>
4   </packageScan>
5 </camelContext>

```

可以在[Camel网站](#)找到关于Camel路由的更多文档。这篇文档中只通过几个小例子展示基本概念。在第一个例子中，在Activiti工作流中进行最简单的Camel调用。叫做SimpleCamelCall。

如果想要定义多个Camel上下文bean，并且/或使用不同的bean名字，可以在Camel任务定义中像这样覆盖：

```

1 <serviceTask id="serviceTask1" activiti:type="camel">
2   <extensionElements>
3     <activiti:field name="camelContext" stringValue="customCamelContext" />
4   </extensionElements>
5 </serviceTask>

```

简单Camel调用示例 Simple Camel Call example

这个例子相关的所有文件，都可以在activiti-camel模块的 org.activiti.camel.examples.simpleCamelCall包中找到。目的是简单启动一个camel路由。首先需要配置了上面提到的路由的Spring上下文。下面的代码用做这个目的：

```

1 <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
2   <packageScan>
3     <package>org.activiti.camel.examples.simpleCamelCall</package>
4   </packageScan>
5 </camelContext>

```

```

1 public class SimpleCamelCallRoute extends RouteBuilder {
2
3   @Override
4   public void configure() throws Exception {
5     from("activiti:SimpleCamelCallProcess:simpleCall").to("log:org.activiti.camel.examples.SimpleCamelCall");
6   }
7 }

```

路由只是记录消息体，不做更多事情。请注意from终端的格式，包含三个部分：

终端Uri部分	描述

终端Url部分	描述
activiti	引用Activiti终端
SimpleCamelCallProcess	流程名
simpleCall	流程中Camel服务的名字

现在路由已经正确配置，可以访问Camel。下面需要像这样定义工作流：

```

1 <process id="SimpleCamelCallProcess">
2   <startEvent id="start"/>
3   <sequenceFlow id="flow1" sourceRef="start" targetRef="simpleCall"/>
4
5   <serviceTask id="simpleCall" activiti:type="camel"/>
6
7   <sequenceFlow id="flow2" sourceRef="simpleCall" targetRef="end"/>
8   <endEvent id="end"/>
9 </process>

```

连通性测试 Ping Pong example

示例已经可以工作，但实际上Camel与Activiti之间并没有通信，因此没有太多价值。在这个例子里，将试着从Camel接收与发送消息。将 发送一个字符串，Camel在其上连接一些东西，并返回作为结果。发送部分比较普通，以变量的格式将信息发送给Camel服务。这是我们的调用代码：

```

1 @Deployment
2 public void testPingPong() {
3     Map<String, Object> variables = new HashMap<String, Object>();
4
5     variables.put("input", "Hello");
6     Map<String, String> outputMap = new HashMap<String, String>();
7     variables.put("outputMap", outputMap);
8
9     runtimeService.startProcessInstanceByKey("PingPongProcess", variables);
10    assertEquals(1, outputMap.size());
11    assertNotNull(outputMap.get("outputValue"));
12    assertEquals("Hello World", outputMap.get("outputValue"));
13 }

```

“input”变量是实际上是Camel路由的输入，而outputMap用于捕获Camel传回的结果。流程像是这样：

```

1 <process id="PingPongProcess">
2   <startEvent id="start"/>
3   <sequenceFlow id="flow1" sourceRef="start" targetRef="ping"/>
4   <serviceTask id="ping" activiti:type="camel"/>
5   <sequenceFlow id="flow2" sourceRef="ping" targetRef="saveOutput"/>
6   <serviceTask id="saveOutput" activiti:class="org.activiti.camel.examples.pingPong.SaveOutput" />
7   <sequenceFlow id="flow3" sourceRef="saveOutput" targetRef="end"/>
8   <endEvent id="end"/>
9 </process>

```

请注意SaveOutput服务任务，将“Output”变量从上下文中取出，存储至上面提到的OutputMap。现在需要了解变量如何发送至 Camel，以及如何返回。这就需要了解Camel行为（Behavior）的概念。变量与Camel通信的方式可以通过CamelBehavior配置。在这个例子里使用默认配置，其它配置在后面会进行简短介绍。下面的代码配置了期望的Camel行为：

```

1 <serviceTask id="serviceTask1" activiti:type="camel">
2   <extensionElements>
3     <activiti:field name="camelBehaviorClass" stringValue="org.activiti.camel.impl.CamelBehaviorCamelBodyImpl" />
4   </extensionElements>
5 </serviceTask>

```

如果不指定行为，则会设置为org.activiti.camel.impl.CamelBehaviorDefaultImpl。这个行为将以相同名字，将变量复制到Camel参数。对于返回值，无论选择什么行为，如果Camel消息体是一个map，则其中的每个元素都将复制为变量，否则整个对象 将复制为名“camelBody”的特定变量。了解这些后，Camel路由总结为第二个例子：

```

1 @Override
2 public void configure() throws Exception {
3     from("activiti:PingPongProcess:ping").transform().simple("${property.input} World");
4 }

```

在这个路由中，字符串"world"会在结尾连接上名为"input"的参数，结果作为消息体。可以通过Java服务任务检查"camelBody"变量，并复制到"outputMap"，并可通过测试用例检查。既然这个例子使用默认行为，就让我们看看还有什么其他选择。在每个Camel路由的开始处，流程实例id会复制为名为"PROCESS_ID_PROPERTY"的Camel参数。之后会用于将流程实例与Camel路由相关联，也可以在Camel路由中使用。

Activiti中已经有可以使用三种不同的行为。可以通过修改路由URL中特定的部分，覆写行为。这里有个在URL中重载已有行为的例子：

```
1 from("activiti:asyncCamelProcess:serviceTaskAsync2?copyVariablesToProperties=true").
```

下表展示了三种可用的Camel行为：

行为	Url中	描述
CamelBehaviorDefaultImpl	copyVariablesToProperties	将Activiti变量复制为Camel参数
CamelBehaviorCamelBodyImpl	copyCamelBodyToBody	只将名为"camelBody"的Activiti变量复制为Camel消息体
CamelBehaviorBodyAsMapImpl	copyVariablesToBodyAsMap	将一个map中的所有Activiti变量复制为Camel消息体

上表解释了Activiti变量如何传递给Camel。下表解释了Camel变量如何返回至Activiti。只能在路由URL中配置。

Url	描述
Default	如果Camel消息体是一个map，则将其中每一对象复制为Activiti变量；否则将整个Camel消息体复制为"camelBody" Activiti变量
copyVariablesFromProperties	将Camel参数以同名复制为Activiti变量
copyCamelBodyToBodyAsString	与default相同，但如果Camel消息体不是map，则首先将其转换为字符串，然后再复制为"camelBody"
copyVariablesFromHeader	额外将Camel头复制为Activiti的同名变量

返回变量 Returning back the variables

上面提到的传递变量，不论是从Camel到Activiti还是反过来，都只用于变量传递的开始侧。要特别注意，由于Activiti的非阻塞行为，Activiti不会自动向Camel返回变量。因此，提供了特殊的语法。可以在Camel路由URL中，以`var.return.someVariableName`的格式，使用一个或多个参数。与这些参数同名，但没有`var.return`部分的变量，会被认为是输出变量，因此将会以相同的名字复制回Camel参数。例如在如下路由中：

```
from("direct:start").to("activiti:process?var.return.exampleVar").to("mock:result");
```

名为`exampleVar`的Activiti变量，将被认为是输出变量，因此会以同名复制回Camel参数。

异步连通性测试 Asynchronous Ping Pong example

上面的例子都是同步的。工作流停止，直到Camel路由结束并返回。有时，需要Activiti工作流继续运行。为了这个目的，Camel服务任务的异步功能就很有用。可以通过将Camel服务任务的异步参数设置为true，启用这个功能。

```
1 <serviceTask id="serviceAsyncPing" activiti:type="camel" activiti:async="true"/>
```

设置这个特性后，Camel路由会由Activiti作业执行器异步启动。如果定义了Camel路由队列，Activiti流程会继续执行Camel服务任务之后的活动。Camel路由会与流程执行完全异步地执行。如果需要在流程定义的某处等待Camel服务任务的响应，可以使用接收任务（receive task）。

```
1 <receiveTask id="receiveAsyncPing" name="Wait State" />
```

流程实例会等待，直到接收到信号，例如来自Camel。在Camel中，可以通过向合适的Activiti终端发送消息，来为流程实例发送信号。

```
1 from("activiti:asyncPingProcess:serviceAsyncPing").to("activiti:asyncPingProcess:receiveAsyncPing");
```

（译者注：原文如此。可能为缺失了的to终端的定义：）

- “activiti”字符串常量
- 流程名
- 接收任务名

从Camel路由实例化工作流 Instantiate workflow from Camel route

上面的所有例子，都是先启动Activiti工作流，然后在工作流中启动Camel路由。也可以反过来。可以在已经启动的Camel路由中实例化工作流。与为接收任务发送消息很类似，除了最后一部分。这是一个简单的路由：

```
1 from("direct:start").to("activiti:camelProcess");
```

可以看到url有两部分，第一部分是“activiti”字符串常量，第二个名字是流程的名字。很明显流程需要已经部署，并且可以通过引擎配置启动。

也可以在Camel头中，将流程起动人设置为某个已认证用户id。要这么做，首先需要在流程定义中指定起动人变量：

```
1 <startEvent id="start" activiti:initiator="initiator" />
```

然后在Camel头中的CamelProcessInitiatorHeader指定用户id。Camel路由会如下定义：

```
1 from("direct:startWithInitiatorHeader")
2 .setHeader("CamelProcessInitiatorHeader", constant("kermit"))
3 .to("activiti:InitiatorCamelCallProcess?processInitiatorHeaderName=CamelProcessInitiatorHeader");
```

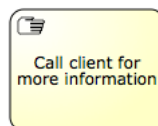
8.5.9. 手动任务 Manual Task

描述 Description

手动任务定义了BPMN引擎外部的任务。用于建模引擎不需要了解的某项工作，或者其他系统或用户界面。对于引擎来说，手动任务将按直接穿过活动处理，在流程执行到达时，自动继续流程。

图示 Graphical Notation

手动任务，表现为左上角带有“手型”图标圆角矩形。



XML表示 XML representation

```
1 <manualTask id="myManualTask" name="Call client for more information" />
```

8.5.10. Java接收任务 Java Receive Task

描述 Description

接收任务，是等待特定消息到达的简单任务。目前，我们只为这个任务实现了Java语义。当流程执行到达接收任务时，流程状态将提交至持久化存储。这意味着流程将保持等待状态，直到引擎接收到特定的消息，并将触发流程通过接收任务。

图示 Graphical notation

接收任务，表现为右上角带有消息图标的任务（圆角矩形）。消息图标是白色的（黑色消息图标代表发送的含义）。



XML表示 XML representation

```
1 <receiveTask id="waitState" name="wait" />
```

要使流程实例从当前的等待状态，如接收任务中继续，需要使用到达接收任务的执行id，调用runtimeService.signal(executionId)。下面的代码片段展示了如何操作：


```

1 ProcessInstance pi = runtimeService.startProcessInstanceByKey("receiveTask");
2 Execution execution = runtimeService.createExecutionQuery()
3     .processInstanceId(pi.getId())
4     .activityId("waitState")
5     .singleResult();
6 assertNotNull(execution);
7
8 runtimeService.signal(execution.getId());

```

8.5.11. Shell任务 Shell Task

描述 Description

Shell任务可以运行Shell脚本与命令。请注意Shell任务不是BPMN 2.0规范的“官方”任务（因此也没有专用图标）。

定义Shell任务 Defining a shell task

Shell任务实现为特殊的**服务任务**，通过将服务任务的`type`定义为'`shell`'设置。

```

1 <serviceTask id="shellEcho" activiti:type="shell">

```

Shell任务通过**字段注入**配置。这些参数的值可以使用EL表达式，将在流程执行运行时解析。可以设置下列参数：

参数	必填？	类型	描述	默认值
command	是	String	要执行的Shell命令。	
arg0-5	否	String	参数0至参数5	
wait	否	true/false	如果可能，是否等待Shell进程终止。	true
redirectError	否	true/false	将标准错误（standard error）并入标准输出（standard output）。	false
cleanEnv	否	true/false	Shell进程不继承当前环境。	false
outputVariable	否	String	保存输出的变量名	不会记录输出。
errorCodeVariable	否	String	保存结果错误代码的变量名	不会注册错误级别。
directory	否	String	Shell进程的默认目录	当前目录

使用示例 Example usage

下面的XML代码片段展示了使用Shell任务的例子。会运行"`cmd /c echo EchoTest`" Shell脚本，等待其结束，并将结果放入`resultVar`。

```

1 <serviceTask id="shellEcho" activiti:type="shell" >
2   <extensionElements>
3     <activiti:field name="command" stringValue="cmd" />
4     <activiti:field name="arg1" stringValue="/c" />
5     <activiti:field name="arg2" stringValue="echo" />
6     <activiti:field name="arg3" stringValue="EchoTest" />
7     <activiti:field name="wait" stringValue="true" />
8     <activiti:field name="outputVariable" stringValue="resultVar" />
9   </extensionElements>
10 </serviceTask>

```

8.5.12. 执行监听器 Execution listener

兼容性提示：在5.3版本后，我们发现执行监听器、任务监听器（task listeners）与表达式仍然在非公开API中。这些类在

`org.activiti.engine.impl`子包

中。`org.activiti.engine.impl.pvm.delegate.ExecutionListener`，`org.activiti.engine.impl.pvm.delegate.TaskListener`

与`org.activiti.engine.impl.pvm.el.Expression`已被废弃。从现在起，应该使

用`org.activiti.engine.delegate.ExecutionListener`，`org.activiti.engine.delegate.TaskListener`

与`org.activiti.engine.delegate.Expression`。在新的公开可用的API中，对

`ExecutionListenerExecution.getEventSource()`的访问已被移除。除了编译器的废弃警告，现有代码可以正常运行。但是请考虑切换至新的公开API接口（包名中不带有`impl`）。

执行监听器可以在流程执行中发生特定的事件时，执行外部Java代码或计算表达式。可以被捕获的事件有：

- 流程实例的start（启动）和end（结束）。
- take（进行）转移（transition）。
- 活动的start和end。
- 网关的start和end。
- 中间事件的start和end。
- 启动事件的end，和结束事件的start。

下面的流程定义包含了三个执行监听器：

```
1 <process id="executionListenersProcess">
2
3   <extensionElements>
4     <activiti:executionListener class="org.activiti.examples.bpmn.executionlistener.ExampleExecutionListenerOne"
5 event="start" />
6   </extensionElements>
7
8   <startEvent id="theStart" />
9   <sequenceFlow sourceRef="theStart" targetRef="firstTask" />
10
11  <userTask id="firstTask" />
12  <sequenceFlow sourceRef="firstTask" targetRef="secondTask">
13    <extensionElements>
14      <activiti:executionListener class="org.activiti.examples.bpmn.executionlistener.ExampleExecutionListenerTwo" />
15    </extensionElements>
16  </sequenceFlow>
17
18  <userTask id="secondTask" >
19    <extensionElements>
20      <activiti:executionListener expression="${myPojo.myMethod(execution.event)}" event="end" />
21    </extensionElements>
22  </userTask>
23  <sequenceFlow sourceRef="secondTask" targetRef="thirdTask" />
24
25  <userTask id="thirdTask" />
26  <sequenceFlow sourceRef="thirdTask" targetRef="theEnd" />
27
28  <endEvent id="theEnd" />
29
30 </process>
```

第一个执行监听器将在流程启动时得到通知。这个监听器是一个外部Java类（例如 `ExampleExecutionListenerOne`），并且需要实现 `org.activiti.engine.delegate.ExecutionListener` 接口。当该事件发生时（这里是 `start` 事件），会调用 `notify(ExecutionListenerExecution execution)` 方法。

```
1 public class ExampleExecutionListenerOne implements ExecutionListener {
2
3   public void notify(ExecutionListenerExecution execution) throws Exception {
4     execution.setVariable("variableSetInExecutionListener", "firstValue");
5     execution.setVariable("eventReceived", execution.getEventName());
6   }
7 }
```

也可以使用实现了 `org.activiti.engine.delegate.JavaDelegate` 接口的代理类。这些代理类也可以用于其他的结构，例如服务任务的代理。

第二个执行监听器在 `take`（进行）转移时被调用。请注意 `listener` 元素并未定义 `event`，因为在转移上只会触发 `take` 事件。当监听器定义在转移上时，`event` 属性的值将被忽略。

最后一个执行监听器在 `secondTask` 活动结束时被调用。监听器声明中没有使用 `class`，而是定义了 `expression`，并将在事件触发时计算/调用。

```
1 <activiti:executionListener expression="${myPojo.myMethod(execution.eventName)}" event="end" />
```

与其他表达式一样，可以使用与解析 `execution` 变量。因为 `execution` 实现对象有一个暴露事件名的参数，因此可以使用 `execution.eventName` 向你的方法传递事件名。

执行监听器也支持使用 `delegateExpression`，与服务任务类似。

```
1 <activiti:executionListener event="start" delegateExpression="${myExecutionListenerBean}" />
```

在Activiti 5.12中，我们也引入了新的执行监听器类型，`org.activiti.engine.impl.bpmn.listener.ScriptExecutionListener`。这个脚本执行监听器，可以为一个执行监听器事件执行一段脚本逻辑。

```
1 <activiti:executionListener event="start" class="org.activiti.engine.impl.bpmn.listener.ScriptExecutionListener" >
2   <activiti:field name="script">
3     <activiti:string>
4       def bar = "BAR"; // local variable
5       foo = "FOO"; // pushes variable to execution context
6       execution.setVariable("var1", "test"); // test access to execution instance
7       bar // implicit return value
8     </activiti:string>
9   </activiti:field>
10  <activiti:field name="language" stringValue="groovy" />
11  <activiti:field name="resultVariable" stringValue="myVar" />
12 </activiti:executionListener>
```

执行监听器上的字段注入 Field injection on execution listeners

当使用通过 `class` 属性配置的执行监听器时，可以使用字段注入。与[服务任务字段注入](#)使用完全相同的机制，可以在那里看到字段注入提供的各种可能用法。

下面的代码片段展示了简单的示例流程，有一个使用了字段注入的执行监听器。

```
1 <process id="executionListenersProcess">
2   <extensionElements>
3     <activiti:executionListener class="org.activiti.examples.bpmn.executionListener.ExampleFieldInjectedExecutionListener"
4     event="start">
5       <activiti:field name="fixedValue" stringValue="Yes, I am " />
6       <activiti:field name="dynamicValue" expression="${myVar}" />
7     </activiti:executionListener>
8   </extensionElements>
9
10  <startEvent id="theStart" />
11  <sequenceFlow sourceRef="theStart" targetRef="firstTask" />
12
13  <userTask id="firstTask" />
14  <sequenceFlow sourceRef="firstTask" targetRef="theEnd" />
15
16  <endEvent id="theEnd" />
</process>
```

```
1 public class ExampleFieldInjectedExecutionListener implements ExecutionListener {
2
3   private Expression fixedValue;
4
5   private Expression dynamicValue;
6
7   public void notify(ExecutionListenerExecution execution) throws Exception {
8     execution.setVariable("var", fixedValue.getValue(execution).toString() + dynamicValue.getValue(execution).toString());
9   }
10 }
```

`ExampleFieldInjectedExecutionListener` 类连接两个字段（一个是固定值，另一个是动态值），并将其存储在 `'var'` 流程变量中。

```
1 @Deployment(resources =
2 {"org/activiti/examples/bpmn/executionListener/ExecutionListenersFieldInjectionProcess.bpmn20.xml"})
3 public void testExecutionListenerFieldInjection() {
4   Map<String, Object> variables = new HashMap<String, Object>();
5   variables.put("myVar", "listening!");
6
7   ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("executionListenersProcess", variables);
8
9   Object varSetByListener = runtimeService.getVariable(processInstance.getId(), "var");
10  assertNotNull(varSetByListener);
11  assertTrue(varSetByListener instanceof String);
12
13  // Result is a concatenation of fixed injected field and injected expression
14  assertEquals("Yes, I am listening!", varSetByListener);
15 }
```

请注意，关于线程安全的规则与服务任务相同。请阅读[相应章节](#)了解更多信息。

8.5.13. 任务监听器 Task listener

任务监听器用于在特定的任务相关事件发生时，执行自定义的Java逻辑或表达式。

任务监听器只能在流程定义中作为**用户任务**的子元素。请注意，任务监听器是一个Activiti自定义结构，因此也需要作为**BPMN 2.0 extensionElements**，放在**activiti**命名空间下。

```
1 <userTask id="myTask" name="My Task" >
2   <extensionElements>
3     <activiti:taskListener event="create" class="org.activiti.MyTaskCreateListener" />
4   </extensionElements>
5 </userTask>
```

任务监听器支持下列属性：

- **event**（事件）（必填）：任务监听器将被调用的任务事件类型。可用的事件有：
 - **create**（创建）：当任务已经创建，并且所有任务参数都已经设置时触发。
 - **assignment**（指派）：当任务已经指派给某人时触发。请注意：当流程执行到达用户任务时，**create**事件触发前，首先触发**assignment**事件。这看起来不是自然顺序，但是有实际原因的：当收到**create**事件时，我们通常希望查看任务的所有参数，包括办理人。
 - **complete**（完成）：当任务已经完成，从运行时数据中删除前触发。
 - **delete**（删除）：在任务即将被删除前触发。请注意当任务通过**completeTask**正常完成时也会触发。
- **class**：需要调用的代理类。这个类必须实现**org.activiti.engine.delegate.TaskListener**接口。

```
1 public class MyTaskCreateListener implements TaskListener {
2
3   public void notify(DelegateTask delegateTask) {
4     // Custom logic goes here
5   }
6
7 }
```

也可以使用**字段注入**，为代理类传递流程变量或执行。请注意代理类的实例在流程部署时创建（与Activiti中其它的代理类一样），这意味着该实例会在所有流程实例执行中共享。

- **expression**：（不能与**class**属性一起使用）：指定在事件发生时要执行的表达式。可以为被调用的对象传递**DelegateTask**对象与事件名（使用**task.eventName**）作为参数。

```
1 <activiti:taskListener event="create" expression="${myObject.callMethod(task, task.eventName)}" />
```

- **delegateExpression**：可以指定一个能够解析为**TaskListener**接口实现类对象的表达式。与服务任务类似。

```
1 <activiti:taskListener event="create" delegateExpression="${myTaskListenerBean}" />
```

- 在Activiti 5.12中，我们也引入了新的执行监听器类型，**org.activiti.engine.impl.bpmn.listener.ScriptTaskListener**。这个脚本任务监听器，可以为一个任务监听器事件执行一段脚本逻辑。

```
1 <activiti:taskListener event="complete" class="org.activiti.engine.impl.bpmn.listener.ScriptTaskListener" >
2   <activiti:field name="script">
3     <activiti:string>
4       def bar = "BAR"; // local variable
5       foo = "FOO"; // pushes variable to execution context
6       task.setOwner("kermit"); // test access to task instance
7       bar // implicit return value
8     </activiti:string>
9   </activiti:field>
10  <activiti:field name="language" stringValue="groovy" />
11  <activiti:field name="resultVariable" stringValue="myVar" />
12 </activiti:taskListener>
```

8.5.14. 多实例 Multi-instance (for each)

描述 Description

多实例活动是在业务流程中，为特定步骤定义重复的方式。在编程概念中，多实例匹配**for each**结构：可以为给定集合中的每一条目，顺序或并行地，执行特定步骤，甚至是整个子流程。

多实例是一个普通活动，加上定义（被称作“多实例特性”）的额外参数，会使得活动在运行时被多次执行。下列活动可以成为多实例活动：

- 用户任务
- 脚本任务

- [Java服务任务](#)
- [Web服务任务](#)
- [业务规则任务](#)
- [邮件任务](#)
- [人工任务](#)
- [接收任务](#)
- [（嵌入式）子流程](#)
- [调用活动](#)

网关与事件不能成为多实例。

按照规范的要求，所有用于为每个实例创建执行的父执行，都有下列变量：

- **nrOfInstances**: 实例总数
- **nrOfActiveInstances**: 当前活动的，也就是说未完成的，实例数量。对于顺序多实例，这个值总为1。
- **nrOfCompletedInstances**: 已经完成的实例数量

可以通过调用 `execution.getVariable(x)` 方法，获取这些值。

另外，每个创建的执行，都有执行本地变量（也就是说，对其他执行不可见，也不存储在流程实例级别）：

- **loopCounter**: 代表给定实例在 *foreach* 循环中的 *index*。可以通过Activiti的 **elementIndexVariable** 属性为 *loopCounter* 变量重命名。

图示 Graphical notation

如果一个活动是多实例，将通过在该活动底部的三条短线表示。三条竖线代表实例会并行执行，而三条横线代表顺序执行。



XML表示 Xml representation

要将活动变成多实例，该活动的XML元素必须有 **multiInstanceLoopCharacteristics** 子元素

```
1 <multiInstanceLoopCharacteristics isSequential="false|true">
2 ...
3 </multiInstanceLoopCharacteristics>
```

isSequential 属性代表了活动的实例为顺序还是并行执行。

实例的数量在进入活动时，计算一次。有不同方法可以配置数量。一个方法是通过 **loopCardinality** 子元素，直接指定数字。

```
1 <multiInstanceLoopCharacteristics isSequential="false|true">
2   <loopCardinality>5</loopCardinality>
3 </multiInstanceLoopCharacteristics>
```

也可以使用解析为正整数的表达式：

```
1 <multiInstanceLoopCharacteristics isSequential="false|true">
2   <loopCardinality>${nrOfOrders-nrOfCancellations}</loopCardinality>
3 </multiInstanceLoopCharacteristics>
```

另一个定义实例数量的方法，是使用 **loopDataInputRef** 子元素，指定一个集合流程变量的名字。对集合中的每一个条目，都会创建一个实例。可以使用 **inputDataItem** 子元素，将集合中的该条目设置给实例。在下面的XML示例中展示：

```
1 <userTask id="miTasks" name="My Task ${loopCounter}" activiti:assignee="${assignee}">
2   <multiInstanceLoopCharacteristics isSequential="false">
3     <loopDataInputRef>assigneeList</loopDataInputRef>
4     <inputDataItem name="assignee" />
5   </multiInstanceLoopCharacteristics>
6 </userTask>
```

假设变量 `assigneeList` 包含 `[kermit, gonzo, fozzie]`。在上面的代码中，会并行创建三个用户任务。每一个执行都有一个名为 `assignee` 的流程变量，含有集合中的一个值，并在这个例子中被用于指派用户任务。

`loopDataInputRef`与`inputDataItem`的缺点是 1)名字很难记 2)由于BPMN 2.0概要的限制，不能使用表达式。Activiti通过在`multiInstanceCharacteristics`上提供`collection`与`elementVariable`属性解决了这些问题：

```
1 <userTask id="miTasks" name="My Task" activiti:assignee="${assignee}">
2   <multiInstanceLoopCharacteristics isSequential="true"
3     activiti:collection="${myService.resolveUsersForTask()}" activiti:elementVariable="assignee" >
4   </multiInstanceLoopCharacteristics>
5 </userTask>
```

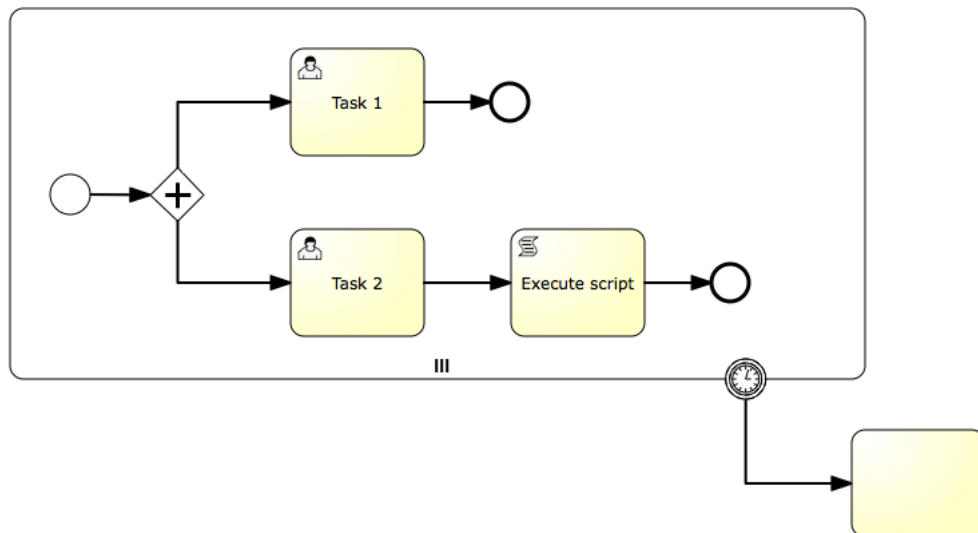
多实例活动在所有实例都完成时结束。然而，也可以指定一个表达式，在每个实例结束时计算。当表达式计算为`true`时，销毁所有剩余的实例，并且结束多实例活动，继续流程。这个表达式必须通过`completionCondition`子元素定义。

```
1 <userTask id="miTasks" name="My Task" activiti:assignee="${assignee}">
2   <multiInstanceLoopCharacteristics isSequential="false"
3     activiti:collection="assigneeList" activiti:elementVariable="assignee" >
4     <completionCondition>${nrOfCompletedInstances/nrOfInstances >= 0.6 }</completionCondition>
5   </multiInstanceLoopCharacteristics>
6 </userTask>
```

在这个例子中，会为`assigneeList`集合中的每个元素创建并行实例。然而，当60%的任务完成时，其他的任务将被删除，流程继续运行。

边界事件与多实例 Boundary events and multi-instance

多实例是普通活动，因此可以在其边界定义边界事件。对于中断边界事件，当捕获事件时，活动中的所有实例都会被销毁。以下面的多实例子流程为例：



当时器触发时，子流程的所有实例都会被销毁，无论有多少实例，或者哪个内部活动还未完成。

多实例与执行监听器 Multi instance and execution listeners

（Activiti 5.18及以上版本可用）

有一个关于执行监听器与多实例一起使用的警告。以下面的BPMN 2.0 XML代码片段为例，其定义在`multiInstanceLoopCharacteristics` XML元素的相同级别：

```
1 <extensionElements>
2   <activiti:executionListener event="start" class="org.activiti.MyStartListener"/>
3   <activiti:executionListener event="end" class="org.activiti.MyEndListener"/>
4 </extensionElements>
```

对于普通的BPMN活动，会在活动开始于结束时调用一次监听器。

然而，当该活动为多实例时，行为有区别：

- 当进入多实例活动时，在任何内部活动执行前，抛出启动事件。这时`loopCounter`变量还未设置（为`null`）。
- 每个实际执行的活动，抛出一个启动事件。这时`loopCounter`变量已经设置。

对结束事件类似：

- 当离开实际活动时，抛出一个结束事件。这时`loopCounter`变量已经设置。

- 当多实例活动整体完成时，抛出一个结束事件。这时`loopCounter`变量未设置。

例如：

```

1 <subProcess id="subprocess1" name="Sub Process">
2   <extensionElements>
3     <activiti:executionListener event="start" class="org.activiti.MyStartListener"/>
4     <activiti:executionListener event="end" class="org.activiti.MyEndListener"/>
5   </extensionElements>
6   <multiInstanceLoopCharacteristics isSequential="false">
7     <loopDataInputRef>assignees</loopDataInputRef>
8     <inputDataItem name="assignee"></inputDataItem>
9   </multiInstanceLoopCharacteristics>
10  <startEvent id="startevent2" name="Start"></startEvent>
11  <endEvent id="endevent2" name="End"></endEvent>
12  <sequenceFlow id="flow3" name="" sourceRef="startevent2" targetRef="endevent2"></sequenceFlow>
13 </subProcess>

```

在这个例子中，假设`assignees`有三个条目。在运行时会发生如下事情：

- 多实例整体抛出一个启动事件。调用一次`start`执行监听器，`loopCounter`与`assignee`变量均未设置（也就是说为`null`）。
- 每一个活动实例抛出一个启动事件。调用三次`start`执行监听器，`loopCounter`与`assignee`变量均已设置（也就是说不为`null`）。
- 因此启动执行监听器总共被调用四次。

请注意当`multiInstanceLoopCharacteristics`不是定义在子元素上，也是如此。例如上面的简单用户任务的例子，也合理适用这一点。

8.5.15. 补偿处理器 Compensation Handlers

描述 Description

[EXPERIMENTAL]

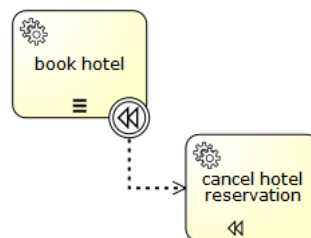
如果一个活动要用于补偿另一个活动的影响，可以声明为补偿处理器。补偿处理器不在普通流程中，只在抛出补偿事件时才会执行。

补偿处理器不得有入口或出口顺序流。

补偿处理器必须通过单向连接，关联一个补偿边界事件。

图示 Graphical notation

如果一个活动是补偿处理器，则会在下部中间显示补偿事件图标。下面摘录的流程图展示了一个带有补偿边界事件的服务任务，并关联至一个补偿处理器。请注意补偿处理器图标显示在“cancel hotel reservation（取消酒店预订）”服务任务的下部中间。



XML表示 XML representation

要将一个活动声明为补偿处理器，需要将`isForCompensation`属性设置为`true`：

```

1 <serviceTask id="undoBookHotel" isForCompensation="true" activiti:class="...">
2 </serviceTask>

```

8.6. 子流程与调用活动 Sub-Processes and Call Activities

8.6.1. 子流程 Sub-Process

描述 Description

子流程是包含其他的活动、网关、事件等的活动。其本身构成一个流程，并作为更大流程的一部分。子流程完全在父流程中定义（这就是为什么经常被称作嵌入式子流程）。

子流程有两个主要的使用场景：

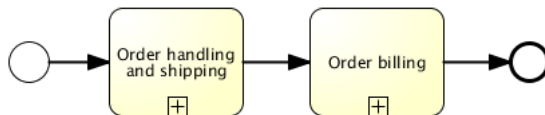
- 子流程可以分层建模。很多建模工具都可以折叠子流程，隐藏子流程的所有细节，而只显示业务流程的高层端到端总览。
- 子流程创建了新的事件范围。在子流程执行中抛出的事件，可以通过子流程边界上的边界事件捕获。因此为该事件创建了限制在子流程内的范围。

使用子流程也要注意以下几点：

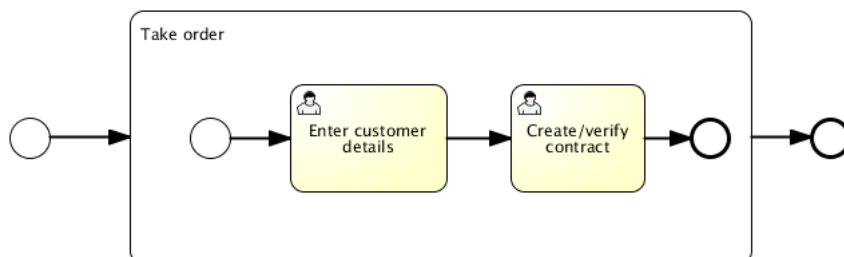
- 子流程只能有一个空启动事件，而不允许有其他类型的启动事件。请注意BPMN 2.0规范允许省略子流程的启动与结束事件，然而当前的Activiti实现并不支持省略。
- 顺序流不能跨越子流程边界。

图示 Graphical Notation

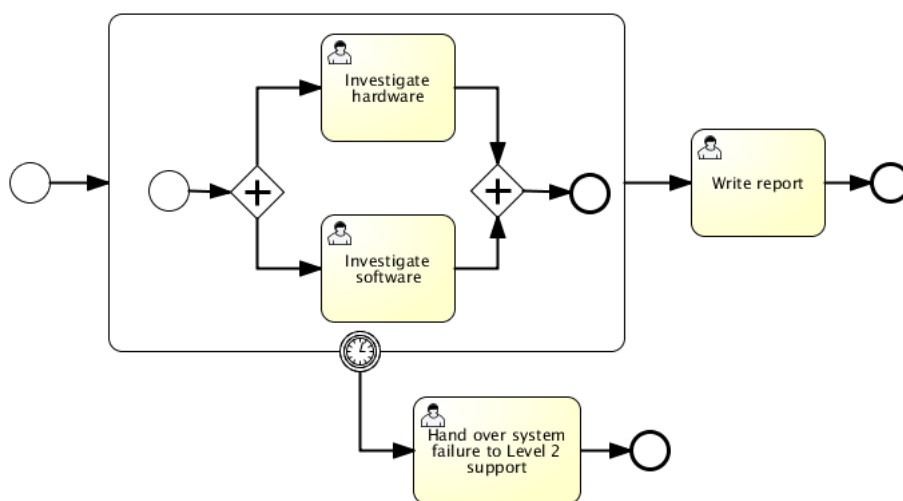
子流程表示为标准活动，即圆角矩形。若折叠了子流程，则只显示其名字与一个加号，提供了流程的高层概览：



若展开了子流程，则子流程的所有步骤都在子流程边界内显示：



使用子流程的一个主要原因，是为特定事件定义范围。下面的流程模型展示了这种用法：*investigate software/investigate hardware*（调查硬件/调查软件）两个任务需要并行执行，且需要在给定时限内，在*Level 2 support*（二级支持）响应前完成。在这里，定时器的范围（即需要按时完成的的活动）通过子流程限制。



XML表示 XML representation

子流程通过`subprocess`元素定义。子流程中的所有活动、网关、事件等，都需要附在这个元素内。

```

1  <subProcess id="subProcess">
2
3  <startEvent id="subProcessStart" />
4
5  ... other Sub-Process elements ...
6
7  <endEvent id="subProcessEnd" />
8
9  </subProcess>

```

8.6.2. 事件子流程 Event Sub-Process

描述 Description

事件子流程是BPMN 2.0新定义的。事件子流程，是通过事件触发的子流程。可以在流程级别，或者任何子流程级别，添加事件子流程。用于触发事件子流程的事件，使用启动事件配置。因此可知，不能在事件子流程中使用空启动事件。事件子流程可以通过例如消息

事件、错误事件、信号时间、定时器事件或补偿事件触发。对启动事件的订阅，在事件子流程的宿主范围（流程实例或子流程）创建时创建。当该范围销毁时，删除订阅。

事件子流程可以是中断或不中断的。中断的子流程将取消当前范围内的任何执行。非中断的事件子流程将创建新的并行执行。宿主范围内的每个活动，只能触发一个中断事件子流程，而非中断事件子流程可以多次触发。子流程是否是中断的，通过触发事件子流程的启动事件配置。

事件子流程不能有任何入口或出口顺序流。事件子流程是由事件触发的，因此入口顺序流不合逻辑。当事件子流程结束时，要么同时结束当前范围（中断事件子流程的情况），要么是非中断子流程创建的并行执行结束。

目前的限制：

- Activiti只支持中断事件子流程。
- Activiti只支持错误启动事件与消息启动事件触发事件子流程。

图示 Graphical Notation

事件子流程，表示为点线边框的嵌入式子流程。



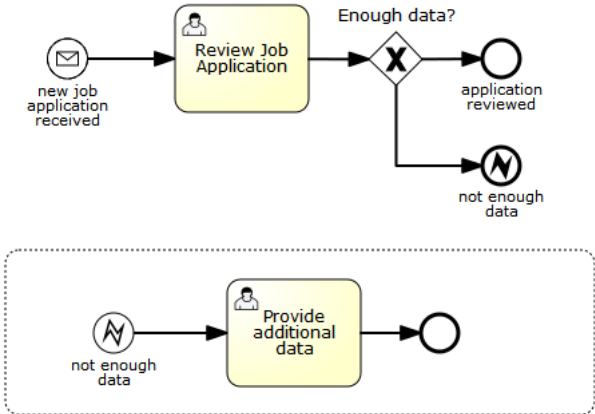
XML表示 XML representation

事件子流程的XML表示格式，与嵌入式子流程相同。但需要将 `triggeredByEvent` 属性设置为 `true`：

```
1 <subProcess id="eventSubProcess" triggeredByEvent="true">
2   ...
3 </subProcess>
```

示例 Example

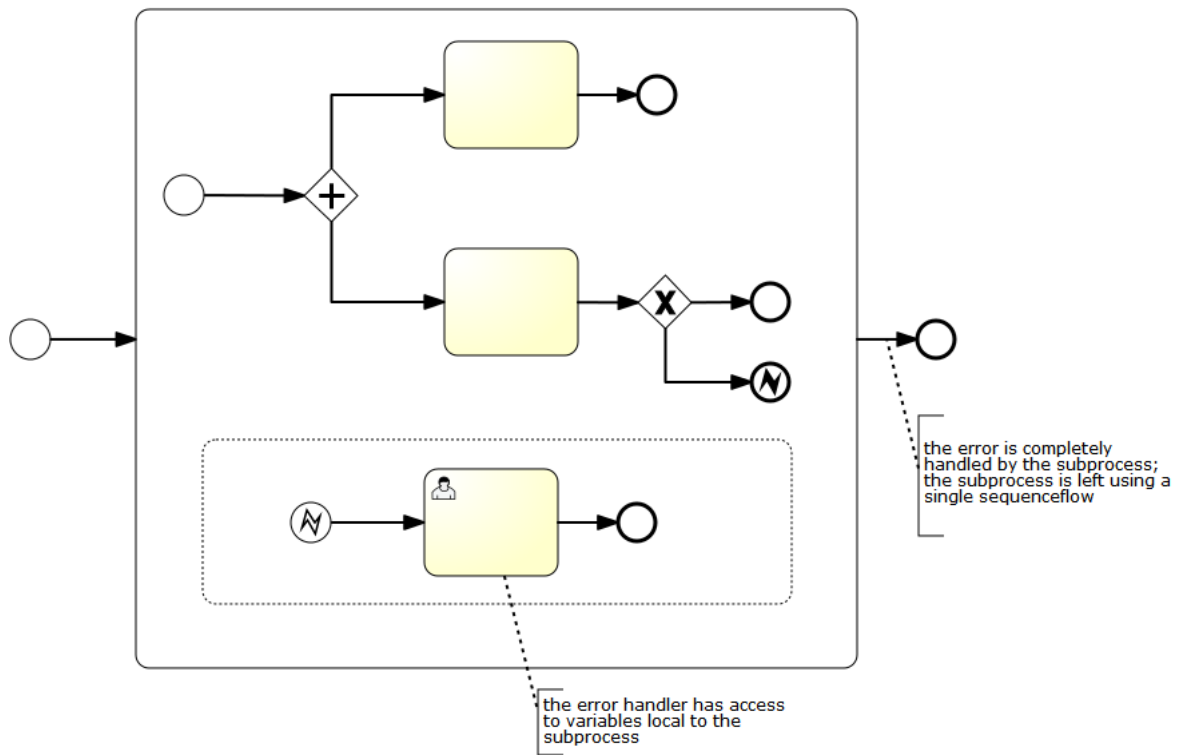
下面是使用错误启动事件触发事件子流程的例子。该事件子流程位与“流程级别”，即流程实例的范围：



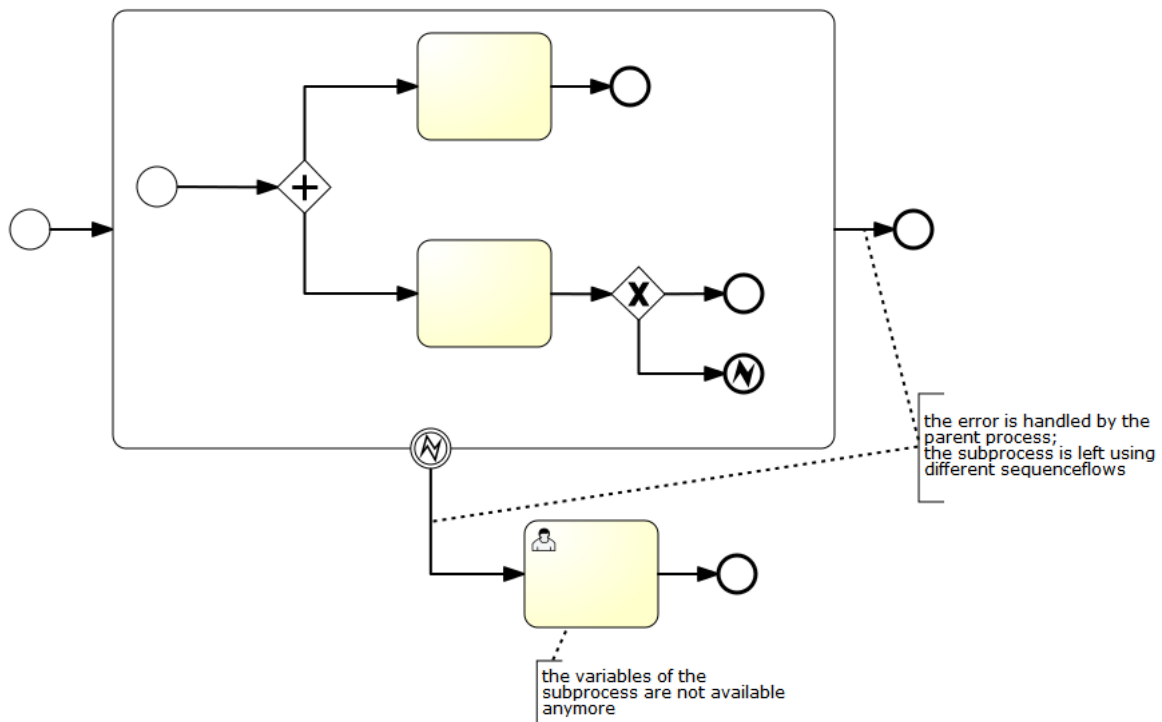
事件子流程在XML是这样的：

```
1 <subProcess id="eventSubProcess" triggeredByEvent="true">
2   <startEvent id="catchError">
3     <errorEventDefinition errorRef="error" />
4   </startEvent>
5   <sequenceFlow id="flow2" sourceRef="catchError" targetRef="taskAfterErrorCatch" />
6   <userTask id="taskAfterErrorCatch" name="Provide additional data" />
7 </subProcess>
```

前面已经指出，事件子流程也可以添加到嵌入式子流程内。若添加到嵌入式子流程内，将可替代边界事件的功能。考虑下面两个流程图，嵌入式子流程都抛出错误事件，该错误事件都被捕获，并由用户任务处理。



对比：



两种情况下都执行相同的任务。然而，两种模型选择有如下不同：

- 嵌入式（事件）子流程使用其宿主范围的执行来执行。这意味着嵌入式（事件）子流程可以访问其范围的局部变量。当使用边界事件时，创建用于执行嵌入式子流程的执行，将被边界事件的出口顺序流删除。这意味着嵌入式子流程创建的变量将不再可用。
- 使用事件子流程时，事件完全由其所在的子流程处理。当使用边界事件时，事件由其父流程处理。

这两个区别可以帮助你判断，使用边界事件还是嵌入式（事件）子流程，哪个更适合解决特定的流程建模/实现问题。

8.6.3. 事务子流程 Transaction subprocess

[EXPERIMENTAL]

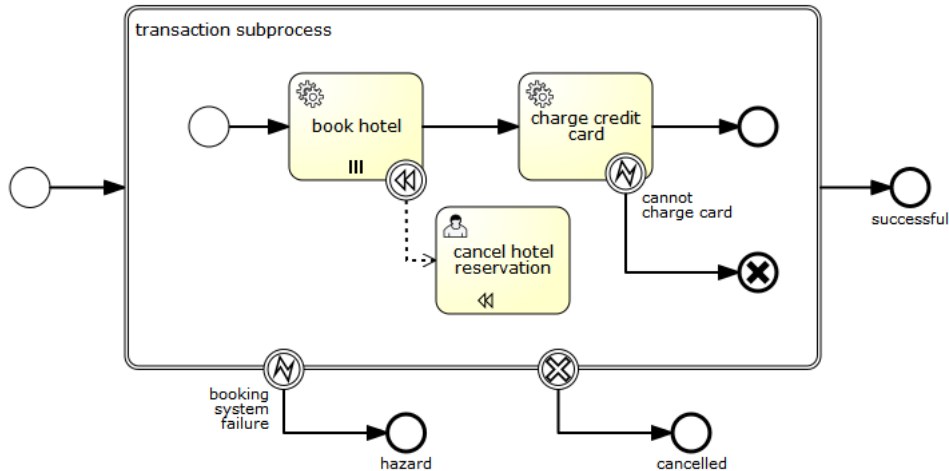
描述 Description

事务子流程是一种嵌入式子流程，可用于将多个活动组织在一个事务里。事务是工作的逻辑单元，可以组织一组独立活动，使得它们可以一起成功或失败。

事务的可能结果：事务有三种不同的结果：

- 若未被取消，或被意外终止，则事务成功。若事务子流程成功，将使用出口顺序流离开。若流程后面抛出了补偿事件，成功的事务可以被补偿。请注意：与“普通”嵌入式子流程一样，可以使用补偿抛出中间事件，在事务成功完成后补偿。
- 若执行到达取消结束事件时，事务被取消。在这种情况下，所有执行都将被终止并移除。只会保留一个执行，设置为取消边界事件，并将触发补偿。在补偿完成后，事务子流程通过取消边界事件的出口顺序流离开。
- 若由于抛出了错误结束事件，且未被事务子流程所在的范围捕获，则事务会被意外终止（错误被事件子流程的边界捕获也一样）。在这种情况下，不会进行补偿。

下面的流程图展示了三种不同的结果：



与**ACID**事务的关系：要注意不要将BPMN事务子流程与技术（ACID）事务混淆。BPMN事务子流程不是划分技术事务范围的方法。要理解Activiti中的事务管理，请阅读[并发与事务](#)章节。BPMN事务与技术事务有如下区别：

- **ACID**事务技术上生存期短暂，而BPMN事务可以持续几小时，几天甚至几个月才完成。（考虑一个场景，事务包括的活动中有一个用户任务。通常人的响应时间要比程序长。或者，在另一个场景下，BPMN事务可能等待某些业务事件发生，像是特定订单的填写完成。）这些操作通常要比更新数据库字段，或者使用事务队列存储消息，花长得多的时间完成。
- 因为不可能将业务活动的持续时间限定为技术事务的范围，一个BPMN事务通常会生成多个ACID事务。
- 因为一个BPMN事务可以生成多个ACID事务，就不再使用ACID特性。例如，考虑上面的流程例子。假设“book hotel（预订酒店）”与“charge credit card（信用卡付款）”操作在分开的ACID事务中处理。再假设“book hotel（预订酒店）”活动已经成功。这时，因为已经进行了预订酒店操作，而还没有进行信用卡扣款，就处在中间不一致状态（intermediary inconsistent state）。在ACID事务中，会顺序进行不同的操作，因此也处在中间不一致状态。在这里不一样的是，不一致状态在事务范围外可见。例如，如果通过外部预订服务进行预定，则使用该预订服务的其他部分将能看到酒店已被预订。这意味着，当时用业务事务时，完全不使用隔离参数（的确，当使用ACID事务时，我们通常也释放隔离，以保证高并发级别。但可以细粒度地控制，而中间不一致状态也只会存在与一小段时间内）。
- BPMN业务事务也不使用传统方式回滚。因为它生成多个ACID事务，在BPMN事务取消时，部分ACID事务可能已经提交。这样它们没法回滚。

因为BPMN事务天生需要长时间运行，因此就需要区别处理缺乏隔离与回滚机制。在实际使用中，通常只能通过领域特定（domain specific）的方式解决这些问题：

- 回滚通过补偿实现。如果在事务范围内抛出了取消事件，所有成功执行，并带有补偿处理器的活动，带来的影响，将被补偿。
- 缺乏隔离通常使用特定领域的解决方案来处理。例如，在上面的例子里，在我们确定第一个客户可以付款前，一个酒店房间可能被第二个客户预定。这可能不满足业务预期，预订服务可能会选择允许一定量的超量预定。
- 另外，由于事务可以由于意外而终止，预订服务需要处理这种情况，酒店房间已经预定，但从未付款（因为事务可能已经终止）。在这种情况下，预定服务可能选择这种策略，一个酒店房间有最大预留时间，若到时还未付款，则取消预订。

总结一下：**ACID**事务提供了这些问题的通用解决方案（回滚，隔离级别，与探索输出 **heuristic outcomes**），但仍然需要在实现业务事务时，为这些问题寻找特定领域的解决方案。

目前的限制：

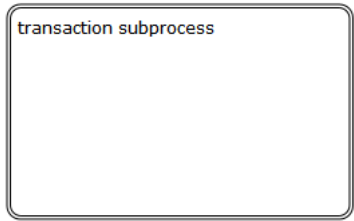
- BPMN规范要求，流程引擎响应底层事务协议提交的事务。例如在底层协议中发生了取消事件，则取消事务。作为可嵌入的引擎，Activiti当前不支持这点。（查看下面关于一致性的段落，了解其后果。）

基于**ACID**事务与乐观锁（**optimistic concurrency**）的一致性：BPMN事务在如下情况保证一致性：所有活动都成功完成；或若部分活动不能执行，则所有已完成活动都被补偿。两种方法都可以得到一致性状态。然而，认识到这一点很重要：Activiti中，BPMN事务的一致性模型，位与流程执行的一致性模型之上。Activiti以事务的方式执行流程。通过乐观锁标记处理并发。在Activiti中，BPMN的错误、取消与补偿事件，都建立在相同的ACID事务与乐观锁之上。例如，只有在实际到达时，取消结束事件才能触发补偿。如果由于服务任务抛出了未检查异常，导致其未实际到达；或者，由于底层ACID事务中的其他操作，将事务设置为rollback-only（回滚）状态，导

致补偿处理器的操作不能提交；或者，当两个并行执行到达一个取消结束事件时，补偿会被两次触发，并由于乐观锁异常而失败。所有这些都是想说明，当在 **Activiti** 中实现 **BPMN** 事务时，与实施“普通”流程与子流程，需要遵守相同的规则。因此要有效地保证一致性，需要将乐观锁、事务执行模型纳入考虑 范围，以实现流程。

图示 Graphical Notation

事务子流程，使用带有两层边框的**嵌入式子流程**表示。



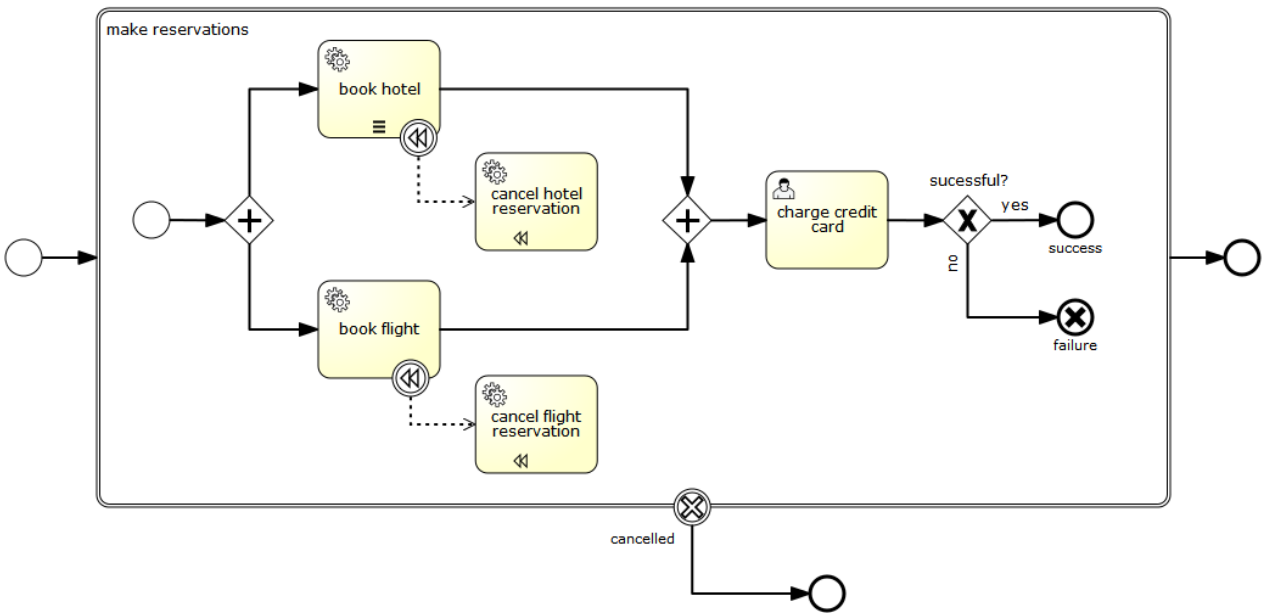
XML表示 XML representation

事务子流程，在XML中通过 **transaction** 标签表示：

```
1 <transaction id="myTransaction" >
2   ...
3 </transaction>
```

示例 Example

下面是一个事务子流程的例子：



8.6.4. 调用活动（子流程） Call activity (subprocess)

描述 Description

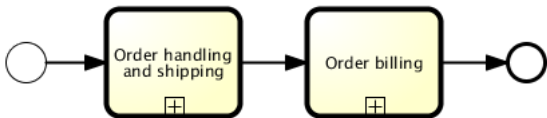
BPMN 2.0区分一般的**子流程**，通常也称作嵌入式子流程，与调用活动，尽管它们看起来很像。从概念上说，两者都在流程执行到达该活动时，调用一个子流程。

区别在于，调用活动引用一个流程定义外部的流程，而**subprocess**嵌入在原有流程定义内。调用活动的主要使用场景，是它有一个可重复使用的流程定义，可以在多个其他流程定义中调用。

当流程执行到达**call activity**时，会创建一个新的执行，作为到达调用活动的执行的子执行。这个子执行之后用于执行子流程，潜在地创建了类似普通流程的并行子执行。父执行将等待子流程完成，之后沿原流程继续执行。

图示 Graphical Notation

调用过程，表现为带有粗边框（折叠与展开都是）的**子流程**。取决于建模工具，调用过程可以展开，但默认表现形式为折叠形式。



XML表现 XML representation

调用活动是一个普通活动，需要有通过其**key**引用流程定义的**calledElement**。在实际使用中，这通常意味着在**calledElement**中使用流程的**id**。

```
1 <callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess" />
```

请注意子流程的流程定义在运行时解析。这意味着如果需要的话，子流程可以与调用流程分别部署。

传递变量 Passing variables

可以向子流程传递流程变量，反之亦然。数据将在子流程启动时复制到子流程，并在其结束时复制回主流程。

```
1 <callActivity id="callSubProcess" calledElement="checkCreditProcess" >
2   <extensionElements>
3     <activiti:in source="someVariableInMainProcess" target="nameOfVariableInSubProcess"
4   />
5     <activiti:out source="someVariableInSubProcess" target="nameOfVariableInMainProcess"
6   />
  </extensionElements>
</callActivity>
```

使用Activiti扩展，作为BPMN标准元素**dataInputAssociation**与**dataOutputAssociation**的扩展。它们需要按照BPMN 2.0标准的方式声明流程变量。

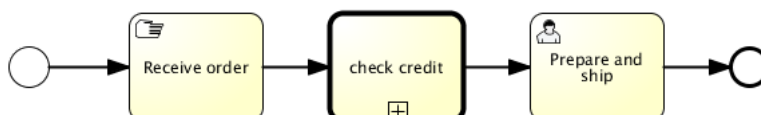
也可以在这里使用表达式：

```
1 <callActivity id="callSubProcess" calledElement="checkCreditProcess" >
2   <extensionElements>
3     <activiti:in sourceExpression="{x+5}" target="y" />
4     <activiti:out source="{y+5}" target="z" />
5   </extensionElements>
6 </callActivity>
```

因此最终 $z = y + 5 = x + 5 + 5$

示例 Example

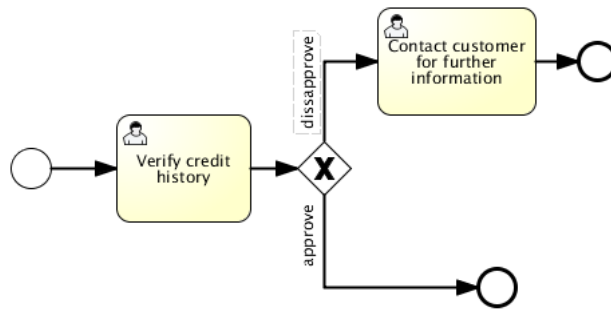
下面的流程图展示了简单的订单处理。因为检查客户的信用额度在许多其他流程中都常见，因此将**check credit step**（检查信用额度步骤）建模为调用活动。



流程像是下面这样：

```
1 <startEvent id="theStart" />
2 <sequenceFlow id="flow1" sourceRef="theStart" targetRef="receiveOrder" />
3
4 <manualTask id="receiveOrder" name="Receive Order" />
5 <sequenceFlow id="flow2" sourceRef="receiveOrder" targetRef="callCheckCreditProcess" />
6
7 <callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess" />
8 <sequenceFlow id="flow3" sourceRef="callCheckCreditProcess" targetRef="prepareAndShipTask" />
9
10 <userTask id="prepareAndShipTask" name="Prepare and Ship" />
11 <sequenceFlow id="flow4" sourceRef="prepareAndShipTask" targetRef="end" />
12
13 <endEvent id="end" />
```

子流程像是下面这样：

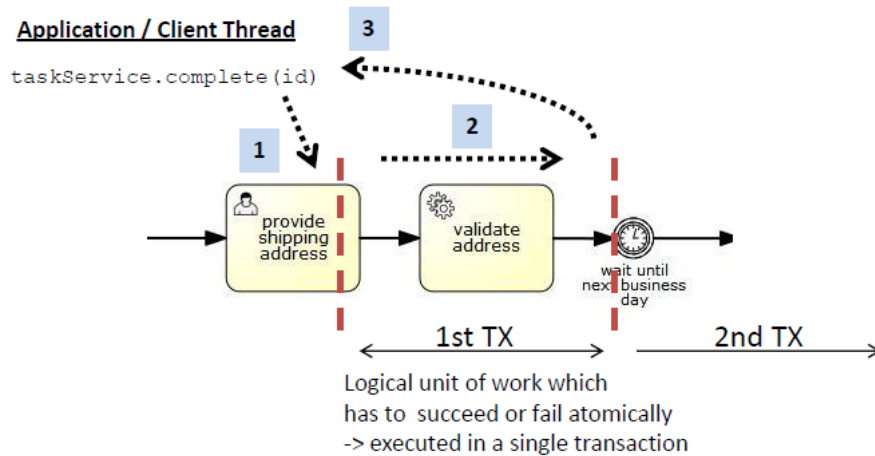


与子流程的流程定义相比没什么特别。也可以不通过其他流程调用而使用。

8.7. 事务与并发 Transactions and Concurrency

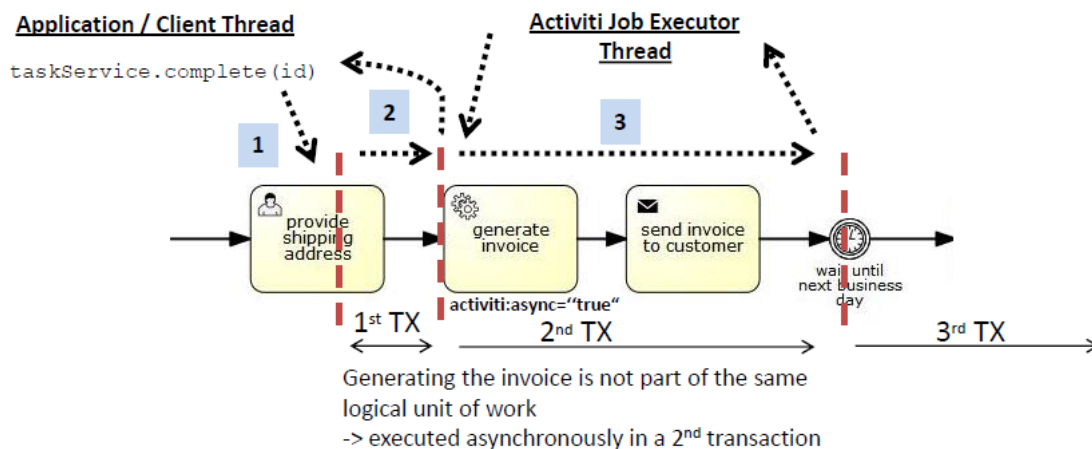
8.7.1. 异步延续 Asynchronous Continuations

Activiti以事务方式执行流程，并可按照你的需求配置。让我们从Activiti一般如何为事务划分范围开始介绍。如果Activiti被触发（也就是说，启动流程，完成任务，为执行发送信号），Activiti将沿流程继续，直到到达每个执行路径的等待状态。更具体地说，它以深度优先方式搜索流程图，并在每个执行分支都到达等待状态时返回。等待状态是“之后”再执行的任务，意味着Activiti将当前执行持久化，并等待再次触发。触发可以来自外部来源，例如用户任务或消息接受任务，也可以来自Activiti自身，例如定时器事件。以下面的图片说明：



这是一个BPMN流程的片段，有一个用户任务，一个服务任务，与一个定时器事件。完成用户任务与验证地址（validate address）在同一个工作单元内，因此需要原子性地（atomically）成功或失败。这意味着如果服务任务抛出了异常，我们会想要回滚当前事务，以便执行返回到用户任务，而用户任务仍然存在于数据库中。这也是Activiti的默认行为。在（1）中，应用或客户端线程完成任务。在相同的线程中，Activiti执行服务并继续，直到到达等待状态，在这个例子中，是定时器事件（2）。然后将控制权返回至调用者（3），同时提交事务（如果事务由Activiti开启）。

在有的情况下，这不是我们想要的。有时我们需要在流程中，自定义地控制事务边界，以便为工作的逻辑单元划分范围。这就需要使用异步延续。考虑下面的流程（片段）：



这次我们完成用户任务，生成发票，并将发票发送给客户。这次发票的生成不再是同一个工作单元的一部分，因此我们不希望当发票生成失败时，回滚用户任务。因此我们希望Activiti做的，是完成用户任务（1），提交事务，并将控制权返回给调用程序。然后我们希望后台线程中，异步地生成发票。这个后台线程就是Activiti作业执行器（事实上是一个线程池），它周期性地将作业保存至数据库。因

此在幕后，当到达"generate invoice（生成发票）"任务时，会为Activiti创建“消息”作业，以继续流程，并将其持久化到数据库中。这个作业之后会被作业执行器选中并执行。我们也会为本地的作业执行器进行提示，告知其有新作业到来，以提升性能。

要使用这个特性，可以使用`activiti:async="true"`扩展。因此，服务任务会像是这样：

```
1 <serviceTask id="service1" name="Generate Invoice" activiti:class="my.custom.Delegate" activiti:async="true" />
```

可以为下列BPMN任务类型指定`activiti:async`：任务，服务任务，脚本任务，业务规则任务，发送任务，接收任务，用户任务，子流程，调用活动

对于用户任务，接收任务与其他等待状态来说，异步延续允许我们可以在一个独立的线程/事务中启动执行监听器。

8.7.2. 失败重试 Fail Retry

默认配置下，如果作业执行中有任何异常，Activiti将3次重试执行作业。对异步任务作业也是这样。有时需要更灵活的配置。可以配置两个参数：

- 重试的次数
- 重试的间隔

这两个参数可以通过`activiti:failedJobRetryTimeCycle`元素配置。这有一个简单的例子：

```
1 <serviceTask id="failingServiceTask" activiti:async="true"
2   activiti:class="org.activiti.engine.test.jobexecutor.RetryFailingDelegate">
3   <extensionElements>
4     <activiti:failedJobRetryTimeCycle>R5/PT7M</activiti:failedJobRetryTimeCycle>
5   </extensionElements>
6 </serviceTask>
```

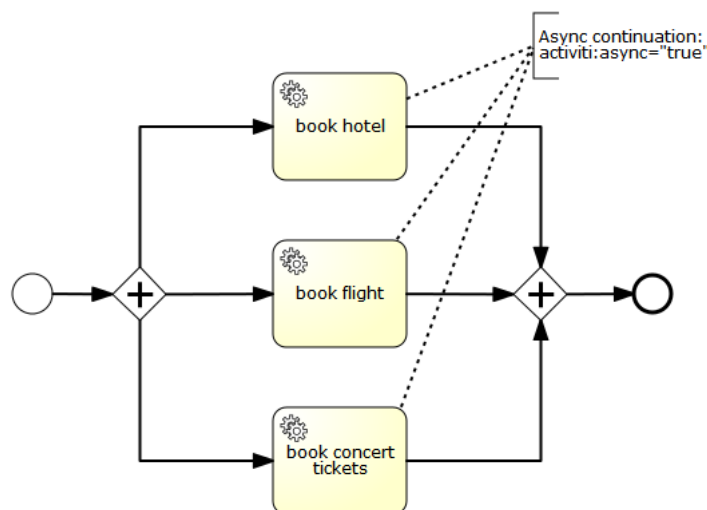
时间周期表达式遵循ISO 8601标准，与定时器事件表达式一样。上面的例子，让作业执行器重试5次，并在每次重试前等待7分钟。

8.7.3. 排他作业 Exclusive Jobs

从Activiti 5.9开始，JobExecutor确保同一个流程实例的作业永远不会并发执行。为什么这样？

为什么排他作业？ Why exclusive Jobs?

考虑下面的流程定义：



我们有一个并行网关，之后是三个服务任务，都使用异步延续执行。其结果是，数据库中添加三个作业。当作业储存在数据库后，就可以使用JobExecutor处理。JobExecutor获取作业，并将其代理至工作线程的线程池，由它们实际执行作业。这意味着通过使用异步延续，可以将工作分派至线程池（在集群场景下，甚至是在集群中跨越多个线程池）。通常这都是好事。然而，也有固有问题：一致性。考虑服务任务后的并行合并。当服务任务的执行完成时，到达并行合并，并需要决定等待其他执行，还是需要继续向前。这意味着，对于每一个到达并行合并的分支，都需要选择继续执行，还是需要等待其他分支上的一个或多个其他执行。

为什么这是问题呢？这是因为服务任务配置为使用异步延续，有可能所有相应的作业都同时被作业执行器处理，并代理至不同的工作线程。结果是服务执行的事务，与到达并行合并的3个独立执行所在的事务，会发生重叠。如果这样，每一个独立事务都“看”不到，其他事物并发地到达了同样的并行合并，并因此判断需要等待其他事务。然而，如果每个事务都判断需要等待其他事务，在并行合并后不会有继续流程的事务，而流程实例也就会永远保持这个状态。

Activiti如何解决这个问题呢？Activiti使用乐观锁，基于数据进行判断，而数据可能不是当前值（因为其他事务可能在我们提交前修改了这个数据，我们确保会在每个事务中都增加同一个数据库记录行的版本号）。这样，无论哪个事务第一个提交，都将成功，而其他的会抛出乐观锁异常并失败。这解决了上面流程中讨论的问题：如果多个执行并发到达并行合并，它们都判断需要等待，增加其父执行（流

程实例)的版本号,并尝试提交。无论哪个执行第一个提交,都可以成功提交,而其他的将会抛出乐观锁异常并失败。因为这些执行由作业触发,Activiti会在等待给定时间后,重试执行相同的作业,期望这一次通过同步的网关。

这是好的解决方案么?我们已经看到,乐观锁使Activiti能够避免不一致。它确保了我们不会“在合并网关卡住”,意味着:要么所有的执行都通过网关,要么数据库中的作业能确保可以重试通过它。然而,尽管这是一个持久化与一致性角度的完美解决方案,仍然不一定总是更高层次的理想行为:

- Activiti只会为同一个作业,重试一个固定的最大次数(默认配置为'3'次)。在这之后,作业仍然保存在数据库中,但不会再重试。这意味着需要手动操作来触发作业。
- 如果一个作业有非事务性的副作用,将不会由于事务失败而回滚。例如,如果"book concert tickets(预定音乐会门票)"服务与Activiti不在同一个事务中,则如果重试执行作业,将预定多张票。

什么是排他作业? What are exclusive jobs?

排他作业不能与同一个流程实例中的其他排他作业同时执行。考虑上面展示的流程:如果我们将服务任务都声明为排他的,则JobExecutor将确保相关的作业都不会并发执行。相反,它将确保不论何时从特定流程实例中获取了排他作业,都将从同一个流程实例中获取所有其他的排他作业,并将它们代理至同一个工作线程。这保证了作业的顺序执行。

如何启用这个特性?从Activiti 5.9起,排他作业成为默认配置。所有异步延续与定时器事件,都因此默认成为排他的。另外,如果希望作业成为非排他的,可以使用`activiti:exclusive="false"`配置。例如,下面的服务任务是异步,但非排他的。

```
1 <serviceTask id="service" activiti:expression="${myService.performBooking(hotel, dates)}" activiti:async="true"
  activiti:exclusive="false" />
```

这是好的解决方案么?有很多人问我们这是否是好的解决方案。他们的顾虑是,这将阻止并行“操作”,因此会有性能问题。再一次,需要考虑以下两点:

- 如果你是专家,并且知道你在做什么(并理解“为什么排他作业?”章节的内容),可以关掉排他。除此之外,对大多数用户来说,异步延续与定时器能够正常工作才更直观。
- 事实上不会有性能问题。只有在重负载下才会有性能问题。重负载意味着作业执行器的所有的工作线程都一直忙碌。对于排他作业,Activiti会简单的根据负载不同进行分配。排他作业意味着同一个流程实例的作业,都将在同一个线程中顺序执行。但是请想一下:有多于一个流程实例。而其他流程实例的作业将被代理至其他线程,并将并发执行。这意味着Activiti不会并发执行同一个流程实例的排他作业,但仍然并发执行多个实例。从总吞吐量角度来看,可以期望大多数场景下都将导致独立的实例更快地完成。此外,执行同一个流程实例中下一个作业所需的数据,将已经在执行集群节点中缓存。如果作业与节点没有这种关系,则数据可能需要重新从数据库中获取。

8.8. 流程启动认证 Process Initiation Authorization

默认情况下,任何人都可以启动已部署流程定义的新流程实例。流程启动认证功能可以定义用户与组,这样Web客户端可以选择性的限制能够启动新流程实例的用户。请注意Activiti引擎不会用任何方式验证认证定义。这个功能只是为了开发人员可以简化Web客户端认证规则的实现。语法与为用户任务指派用户的语法类似。可以使用`<activiti:potentialStarter>`标签,将用户或组指派为流程的潜在启动者。这里有一个例子:

```
1 <process id="potentialStarter">
2   <extensionElements>
3     <activiti:potentialStarter>
4       <resourceAssignmentExpression>
5         <formalExpression>group2, group(group3), user(user3)</formalExpression>
6       </resourceAssignmentExpression>
7     </activiti:potentialStarter>
8   </extensionElements>
9
10  <startEvent id="theStart"/>
11  ...
```

在上面摘录的XML中, `user(user3)`直接引用用户`user3`,而`group(group3)`引用组`group3`。组没有默认标志。也可以使用`<process>`标签,名为`<activiti:candidateStarterUsers>`与`<activiti:candidateStarterGroups>`的属性。这里有一个例子:

```
1 <process id="potentialStarter" activiti:candidateStarterUsers="user1, user2"
2   activiti:candidateStarterGroups="group1">
3   ...
```

这些属性可以同时使用。

在流程启动认证定义后,开发者可以使用下列方法获取该认证定义。这段代码获取可以由给定用户启动的流程定义列表:

```
1 processDefinitions = repositoryService.createProcessDefinitionQuery().startableByUser("userxxx").list();
```

也可以获取给定流程定义中,所有定义为潜在启动者的身份联系

```
1 identityLinks = repositoryService.getIdentityLinksForProcessDefinition("processDefinitionId");
```

下面的例子展示了如何获取能够启动给定流程的用户列表：

```
1 List<User> authorizedUsers = identityService().createUserQuery().potentialStarter("processDefinitionId").list();
```

用完全相同的方法，可以获取配置为给定流程定义的潜在启动者的组列表：

```
1 List<Group> authorizedGroups = identityService().createGroupQuery().potentialStarter("processDefinitionId").list();
```

8.9. 数据对象 Data objects

[EXPERIMENTAL]

BPMN提供了将数据对象定义为流程或子流程元素的一部分的可能性。根据BPMN规范，可以包含复杂的XML结构，并可以从XSD定义中引入。作为Activiti支持的第一批数据对象，支持下列XSD类型：

```
1 <dataObject id="dObj1" name="StringTest" itemSubjectRef="xsd:string"/>
2 <dataObject id="dObj2" name="BooleanTest" itemSubjectRef="xsd:boolean"/>
3 <dataObject id="dObj3" name="DateTest" itemSubjectRef="xsd:datetime"/>
4 <dataObject id="dObj4" name="DoubleTest" itemSubjectRef="xsd:double"/>
5 <dataObject id="dObj5" name="IntegerTest" itemSubjectRef="xsd:int"/>
6 <dataObject id="dObj6" name="LongTest" itemSubjectRef="xsd:long"/>
```

数据对象的定义，将使用'name'属性值作为新变量的名字，自动转换为流程变量。另外，Activiti也提供了扩展元素，用于为变量设置默认值。下面的BPMN代码片段提供了示例：

```
1 <process id="dataObjectScope" name="Data Object Scope" isExecutable="true">
2   <dataObject id="dObj123" name="StringTest123" itemSubjectRef="xsd:string">
3     <extensionElements>
4       <activiti:value>Testing123</activiti:value>
5     </extensionElements>
6   </dataObject>
7   ...
```

9. 表单 Forms

Activiti提供了一个方便灵活的方法，为你的业务流程的人工步骤添加表单。我们支持两种使用表单的方式：表单参数的内置表单渲染，以及外部表单渲染。

9.1. 表单参数 Form properties

所有与业务流程相关的信息，要么包含在流程变量里，要么可以通过流程变量引用。Activiti支持将复杂的Java对象，以 **Serializable** 对象的方式存储为流程变量，而JPA实体或者整个XML文档将存储为 **String**。

启动流程与完成用户任务是参与流程的地方。与人交流需要使用某些用户界面技术渲染表单。为了简化多用户界面技术，流程定义可以包含转换逻辑，将流程变量中的复杂的Java对象转换为参数的 **Map<String,String>**。

然后任何用户界面技术，都可以使用暴露这些参数信息的Activiti API方法，在这些参数的基础上构建表单。这些参数可以提供对流程变量的专门（也更受限）视图。用于显示表单的参数是 **FormData** 的返回值。例如

```
1 StartFormData FormService.getStartFormData(String processDefinitionId)
```

或者

```
1 TaskFormdata FormService.getTaskFormData(String taskId)
```

默认情况下，内建表单引擎能够“看到”参数与流程变量。因此如果任务表单参数1对1匹配流程变量，则不需要专门声明。例如，对于下列声明：

```
1 <startEvent id="start" />
```

当执行到达startEvent时，所有流程变量都可用。然而

```
1 formService.getStartFormData(String processDefinitionId).getFormProperties()
```

将为空，因为并未指定映射。

在上面的例子中，所有提交的参数将被存储为流程变量。这意味着简单地在表单中添加输入框，就可以存储新变量。

参数从流程变量衍生出来，但不是必须存储为流程变量。例如，流程变量可以是类地址的JPA实体。而用户界面技术使用的 **StreetName** 表单参数，可以通过 **#{address.street}** 表达式连接。

类似的，表单中用户需要提交的参数可以存储为流程变量，也可以作为某个流程变量的嵌套参数，使用UEL值表达式，如 **#{address.street}**。

提交的参数的默认行为，是存储为流程变量，除非使用 **formProperty** 声明指定。

流程也可以在表单参数与流程变量之间进行转换。

例如：

```
1 <userTask id="task">
2   <extensionElements>
3     <activiti:formProperty id="room" />
4     <activiti:formProperty id="duration" type="long"/>
5     <activiti:formProperty id="speaker" variable="SpeakerName" writable="false" />
6     <activiti:formProperty id="street" expression="#{address.street}" required="true" />
7   </extensionElements>
8 </userTask>
```

- **room** 表单参数将作为String，映射为 **room** 流程变量
- **duration** 表单参数将作为java.lang.Long，映射为 **duration** 流程变量
- **speaker** 表单参数将被映射为 **SpeakerName** 流程变量。将只在TaskFormData对象中可用。若提交了speaker参数，将抛出ActivitiException。类似的，使用 **readable="false"** 属性，可以将参数从FormData中排除，但仍然可以在提交时处理。
- **street** 表单参数将作为String，映射为 **address** 流程变量的Java bean参数 **street**。如果在提交时没有提供这个字段，**required="true"**将抛出异常。

也可以提供类型元数据，作为 **StartFormData FormService.getStartFormData(String processDefinitionId)** 与 **TaskFormdata FormService.getTaskFormData(String taskId)** 方法返回的FormData的一部分

我们支持下列表单参数类型：

- **string** (org.activiti.engine.impl.form.StringFormType)
- **long** (org.activiti.engine.impl.form.LongFormType)
- **enum** (org.activiti.engine.impl.form.EnumFormType)
- **date** (org.activiti.engine.impl.form.DateFormType)
- **boolean** (org.activiti.engine.impl.form.BooleanFormType)

对每个声明的表单参数，下列 **FormProperty** 信息都可以通过 **List<FormProperty> formService.getStartFormData(String processDefinitionId).getFormProperties()** 与 **List<FormProperty> formService.getTaskFormData(String taskId).getFormProperties()** 方法获取

```
1 public interface FormProperty {
2   /**
3    * 在{@Link FormService#submitStartFormData(String, java.util.Map)}
4    * 或{@Link FormService#submitTaskFormData(String, java.util.Map)}
5    * 中提交参数时使用的key
6    *
7    * the key used to submit the property in {@Link FormService#submitStartFormData(String, java.util.Map)}
8    * or {@Link FormService#submitTaskFormData(String, java.util.Map)} */
9   String getId();
10
11  /** 显示标签 the display Label */
12  String getName();
13
14  /** 在本接口中定义的类型，例如{@Link #TYPE_STRING}
15   * one of the types defined in this interface like e.g. {@Link #TYPE_STRING} */
16  FormType getFormType();
17
18  /** 这个参数需要显示的可选值
19   * optional value that should be used to display in this property */
20  String getValue();
21
22  /** 这个参数是否需要读取用于在表单中显示，并可通过
23   * {@Link FormService#getStartFormData(String)}
24   * 与{@Link FormService#getTaskFormData(String)}
25   * 方法访问。
26   *
27   * is this property read to be displayed in the form and made accessible with the methods
```

```

28     * {@Link FormService#getStartFormData(String)} and {@Link FormService#getTaskFormData(String)}. */
29     boolean isReadable();
30
31     /** 用户提交表单时是否可以包含这个参数? is this property expected when a user submits the form? */
32     boolean isWritable();
33
34     /** 输入框中是否必填这个参数 is this property a required input field */
35     boolean isRequired();
36 }

```

例如:

```

1  <startEvent id="start">
2    <extensionElements>
3      <activiti:formProperty id="speaker"
4        name="Speaker"
5        variable="SpeakerName"
6        type="string" />
7
8      <activiti:formProperty id="start"
9        type="date"
10       datePattern="dd-MMM-yyyy" />
11
12     <activiti:formProperty id="direction" type="enum">
13       <activiti:value id="left" name="Go Left" />
14       <activiti:value id="right" name="Go Right" />
15       <activiti:value id="up" name="Go Up" />
16       <activiti:value id="down" name="Go Down" />
17     </activiti:formProperty>
18
19   </extensionElements>
20 </startEvent>

```

所有这些信息都可以通过API获取。类型名可以通过 `formProperty.getType().getName()` 获取，日期格式可以通过 `formProperty.getType().getInformation("datePattern")` 获取，枚举值可以通过 `formProperty.getType().getInformation("values")` 获取。

Activiti Explorer支持表单参数，并会按照表单定义渲染表单。下面的XML代码片段

```

1  <startEvent>
2    <extensionElements>
3      <activiti:formProperty id="numberOfDays" name="Number of days" value="{numberOfDays}" type="long" required="true"/>
4      <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyyy)" value="{startDate}" datePattern="dd-MM-
5      yyyy hh:mm" type="date" required="true" />
6      <activiti:formProperty id="vacationMotivation" name="Motivation" value="{vacationMotivation}" type="string" />
7    </extensionElements>
8  </userTask>

```

当使用Activiti Explorer时，将会渲染为流程启动表单

9.2. 外部表单渲染 External form rendering

API也支持使用在Activiti引擎之外渲染的，你自己的任务表单。下面的步骤解释了在自行渲染任务表单时，可以使用的钩子。

本质上，渲染表单所需的所有数据，都组装在这两个方法之一中：`StartFormData FormService.getStartFormData(String processDefinitionId)`与`TaskFormdata FormService.getTaskFormData(String taskId)`。

提交表单参数可以通过`ProcessInstance FormService.submitStartFormData(String processDefinitionId, Map<String,String> properties)`与`void FormService.submitTaskFormData(String taskId, Map<String,String> properties)`完成。

要了解表单参数如何映射为流程变量，查看表单参数 [Form properties](#)

可以将任何表单模板资源，放在部署的业务存档中（如果希望将它们按版本与流程存储在一起）。作为部署中的资源，可以使用 `String ProcessDefinition.getDeploymentId()` 与 `++InputStream RepositoryService.getResourceAsStream(String deploymentId, String resourceName);` 获取。这就是你的模板定义文件，可以用于在你的应用中渲染/显示表单。

除了任务表单，也可以为任何目的，使用访问部署资源的能力。

`<userTask activiti:formKey="..."` 属性，由API通过 `String FormService.getStartFormData(String processDefinitionId).getFormKey()` 与 `String FormService.getTaskFormData(String taskId).getFormKey()` 暴露。可以用它保存部署中模板的全名（如 `org/activiti/example/form/my-custom-form.xml`），但并非必须。例如，也可以在表单参数中保存普通的key，并用算法或变换得到实际需要使用的模板。在你需要使用不同的用户界面技术，渲染不同的表单时很有用。例如，一个表单在普通屏幕尺寸的Web应用中使用，另一个表单在手机小屏幕中使用，甚至可以为IM表单或邮件表单提供模板。

10. JPA（Java Persistence API Java持久化API）

可以使用JPA实体作为流程变量，这样可以：

- 基于流程变量更新已有JPA实体。流程变量可以在用户任务的表单中填写，或者通过服务任务生成。
- 重用已有的领域模型，而不需要写专门的服务用于读取与更新实体值。
- 基于已有实体做决策（网关）。
- ...

10.1. 需求 Requirements

只能支持完全满足下列条件的实体：

- 实体需要使用JPA注解配置，字段与参数访问器都支持。也可以使用映射的父类。
- 实体需要有使用 `@Id` 注解的主键，不支持复合主键（`@EmbeddedId` 与 `@IdClass`）。Id字段/参数可以是任何JPA规范支持的类型：原生类型与其包装器（除了boolean）、`String`、`BigInteger`、`BigDecimal`、`java.util.Date` 与 `java.sql.Date`。

10.2. 配置 Configuration

要使用JPA实体，引擎必须引用 `EntityManagerFactory`。可以通过配置引用，或者提供持久化单元名（Persistence Unit Name）来实现。用作变量的JPA实体将将被自动检测，并按情况处理。

下面的示例配置使用 `jpaPersistenceUnitName`：

```
1 <bean id="processEngineConfiguration"
2   class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
3
4 <!-- Database configurations -->
5 <property name="databaseSchemaUpdate" value="true" />
6 <property name="jdbcUrl" value="jdbc:h2:mem:JpaVariableTest;DB_CLOSE_DELAY=1000" />
7
8 <property name="jpaPersistenceUnitName" value="activiti-jpa-pu" />
9 <property name="jpaHandleTransaction" value="true" />
10 <property name="jpaCloseEntityManager" value="true" />
11
12 <!-- job executor configurations -->
13 <property name="jobExecutorActivate" value="false" />
14
15 <!-- mail server configurations -->
16 <property name="mailServerPort" value="5025" />
17 </bean>
```

下面的示例配置提供了我们自己定义的 `EntityManagerFactory`（在这个例子里，是一个open-jpa实体管理器）。请注意这段代码只包含了与本例相关的bean，省略了其他的。带有open-jpa实体管理器的完整的可用示例，可以在 `activiti-spring-examples (/activiti-spring/src/test/java/org/activiti/spring/test/jpa/JPASpringTest.java)` 中找到。

```
1 <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
2   <property name="persistenceUnitManager" ref="pum"/>
3   <property name="jpaVendorAdapter">
4     <bean class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter">
5       <property name="databasePlatform" value="org.apache.openjpa.jdbc.sql.H2Dictionary" />
6     </bean>
7   </property>
8 </bean>
9
10 <bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
11   <property name="dataSource" ref="dataSource" />
12   <property name="transactionManager" ref="transactionManager" />
13   <property name="databaseSchemaUpdate" value="true" />
14   <property name="jpaEntityManagerFactory" ref="entityManagerFactory" />
15   <property name="jpaHandleTransaction" value="true" />
16   <property name="jpaCloseEntityManager" value="true" />
```



```

17     <property name="jobExecutorActivate" value="false" />
18 </bean>

```

也可以在编程构建引擎时，使用相同的配置，例如：

```

1 ProcessEngine processEngine = ProcessEngineConfiguration
2 .createProcessEngineConfigurationFromResourceDefault()
3 .setJpaPersistenceUnitName("activiti-pu")
4 .buildProcessEngine();

```

配置参数：

- **jpaPersistenceUnitName**：要使用的持久化单元的名字。（要确保该持久化单元在classpath中可用。根据规范，默认位置为/META-INF/persistence.xml）。**jpaEntityManagerFactory**与**jpaPersistenceUnitName**二选一。
- **jpaEntityManagerFactory**：对实现了**javax.persistence.EntityManagerFactory**的bean的引用，将用于载入实体，并刷入更新。**jpaEntityManagerFactory**与**jpaPersistenceUnitName**二选一。
- **jpaHandleTransaction**：标示引擎是否需要启动事务，并在使用**EntityManager**实例后提交/回滚。当使用**Java Transaction API (JTA)**时，设置为false。
- **jpaCloseEntityManager**：标示引擎是否需要关闭其从**EntityManagerFactory**获取的**EntityManager**实例。当**EntityManager**由容器管理时（例如，使用扩展持久化上下文 **Extended Persistence Context**时，不支持将范围限制为单一事务）设置为false。

10.3. 使用 Usage

10.3.1. 简单示例 Simple Example

可以在Activiti源代码的JPAVariableTest中找到使用JPA变量的例子。我们会一步一步解释

JPAVariableTest.testUpdateJPAEntityValues。

首先，基于**META-INF/persistence.xml**，为我们的持久化单元创建一个**EntityManagerFactory**。它包含了需要包含在持久化单元内的类，以及一些厂商特定配置。

在这个测试里我们使用简单实体，它有一个id以及一个**String**值参数，用于持久化。在运行测试前，先创建一个实体并保存。

```

1 @Entity(name = "JPA_ENTITY_FIELD")
2 public class FieldAccessJPAEntity {
3
4     @Id
5     @Column(name = "ID_")
6     private Long id;
7
8     private String value;
9
10    public FieldAccessJPAEntity() {
11        // JPA需要的空构造方法 Empty constructor needed for JPA
12    }
13
14    public Long getId() {
15        return id;
16    }
17
18    public void setId(Long id) {
19        this.id = id;
20    }
21
22    public String getValue() {
23        return value;
24    }
25
26    public void setValue(String value) {
27        this.value = value;
28    }
29 }

```

启动一个新的流程实例，将这个实体加入变量。与其他变量一样，它们都会在引擎中持久化存储。当下一次请求这个变量时，将会根据存储的类与Id，从**EntityManager**载入。

```

1 Map<String, Object> variables = new HashMap<String, Object>();
2 variables.put("entityToUpdate", entityToUpdate);
3
4 ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("UpdateJPAValuesProcess", variables);

```

我们流程定义的第一个节点，是一个**服务任务**，将调用**entityToUpdate**上的**setValue**方法。它将解析为我们之前启动流程实例时设置的JPA变量，并使用当前引擎的上下文关联的**EntityManager**载入。


```

1 <serviceTask id='theTask' name='updateJPAEntityTask'
2   activiti:expression='${entityToUpdate.setValue('updatedValue')}' />

```

当服务任务完成时，流程实例在流程定义中定义的用户任务处等待，让我们可以查看流程实例。在这时，`EntityManager` 已经刷入，对实体的修改也已经存入数据库。当我们使用 `entityToUpdate` 变量的值时，将重新载入，我们会得到 `value` 参数设置为 `updatedValue` 的实体。

```

1 // 流程 'UpdateJPAValuesProcess' 中的服务任务应已设置了 entityToUpdate 的 value。
2 // Servicetask in process 'UpdateJPAValuesProcess' should have set value on entityToUpdate.
3 Object updatedEntity = runtimeService.getVariable(processInstance.getId(), "entityToUpdate");
4 assertTrue(updatedEntity instanceof FieldAccessJPAEntity);
5 assertEquals("updatedValue", ((FieldAccessJPAEntity)updatedEntity).getValue());

```

10.3.2. 查询JPA流程变量 Query JPA process variables

可以查询以特定JPA实体作为变量值的流程实例与执行。请注意对于 `ProcessInstanceQuery` 与 `ExecutionQuery` 的JPA实体查询，只支持 `variableValueEquals(name, entity)`。

而 `variableValueNotEquals`、`variableValueGreaterThan`、`variableValueGreaterThanOrEqual`、`variableValueLessThan` 与 `variableValueLessThanOrEqual` 方法都不支持，并会在值传递为JPA实体时，抛出 `ActivitiException`。

```

1 ProcessInstance result = runtimeService.createProcessInstanceQuery()
2   .variableValueEquals("entityToQuery", entityToQuery).singleResult();

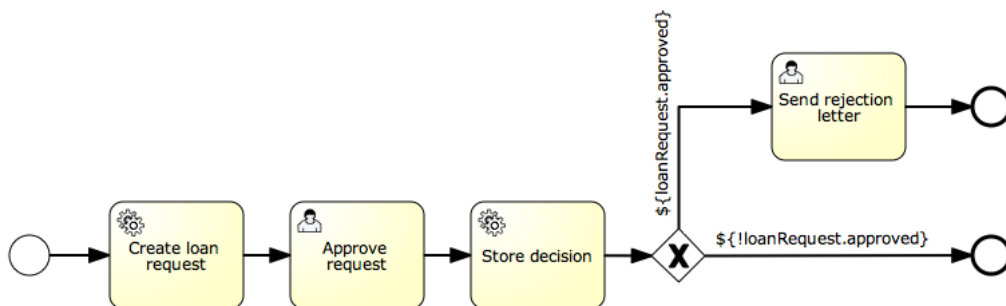
```

10.3.3. 使用Spring bean与JPA的高级示例 Advanced example using Spring beans and JPA

可以在 `activiti-spring-examples` 中找到更高级的例子，`JPASpringTest`。它描述了下属简单用例：

- 一个已有的Spring bean，使用已有的JPA实体，用于存储贷款申请。
- 使用Activiti，可以通过该bean获取该实体，并将其用作流程中的变量。流程定义如下步骤：
 - 创建新的LoanRequest（贷款申请）的服务任务，使用已有的 `LoanRequestBean`，并使用启动流程时接收的变量（例如，从启动表单）。创建的实体作为变量存储，使用 `activiti:resultVariable` 将表达式结果存储为变量。
 - 让经理可以审核申请并批准/驳回的用户任务，该选择将会存储为boolean变量 `approvedByManager`。
 - 更新贷款申请实体的服务任务，以便其可以与流程同步。
 - 依据 `approved` 实体参数的值，使用一个排他网关，选择下一步采用哪条路径：若申请被批准，结束流程；否则，产生一个额外任务（Send rejection letter 发送拒信），以便客户可以收到拒信得到通知。

请注意这个流程不包含任何表单，因为它只用于单元测试。



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions id="taskAssigneeExample"
3   xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmlns:activiti="http://activiti.org/bpmn"
6   targetNamespace="org.activiti.examples">
7
8   <process id="LoanRequestProcess" name="Process creating and handling loan request">
9     <startEvent id='theStart' />
10    <sequenceFlow id='flow1' sourceRef='theStart' targetRef='createLoanRequest' />
11
12    <serviceTask id='createLoanRequest' name='Create loan request'
13      activiti:expression='${loanRequestBean.newLoanRequest(customerName, amount)}'
14      activiti:resultVariable="loanRequest"/>
15    <sequenceFlow id='flow2' sourceRef='createLoanRequest' targetRef='approveTask' />
16
17    <userTask id="approveTask" name="Approve request" />
18    <sequenceFlow id='flow3' sourceRef='approveTask' targetRef='approveOrDisapprove' />
19

```

```

20     <serviceTask id='approveOrDissapprove' name='Store decision'
21         activiti:expression="${loanRequest.setApproved(approvedByManager)}" />
22     <sequenceFlow id='flow4' sourceRef='approveOrDissapprove' targetRef='exclusiveGw' />
23
24     <exclusiveGateway id="exclusiveGw" name="Exclusive Gateway approval" />
25     <sequenceFlow id="endFlow1" sourceRef="exclusiveGw" targetRef="theEnd">
26         <conditionExpression xsi:type="tFormalExpression">${loanRequest.approved}</conditionExpression>
27     </sequenceFlow>
28     <sequenceFlow id="endFlow2" sourceRef="exclusiveGw" targetRef="sendRejectionLetter">
29         <conditionExpression xsi:type="tFormalExpression">${!loanRequest.approved}</conditionExpression>
30     </sequenceFlow>
31
32     <userTask id="sendRejectionLetter" name="Send rejection letter" />
33     <sequenceFlow id='flow5' sourceRef='sendRejectionLetter' targetRef='theOtherEnd' />
34
35     <endEvent id='theEnd' />
36     <endEvent id='theOtherEnd' />
37 </process>
38
39 </definitions>

```

尽管上面的例子很简单，但也展示了组合使用JPA与Spring以及带参数方法表达式的威力。这个流程完全不需要自定义Java代码（当然除了Spring bean），大幅加速了开发。

11. 历史 History

历史是捕获流程执行过程中发生的事情，并将其永久存储的组件。与运行时数据相反，历史数据在流程实例完成以后，仍会保持在数据库中。

有5个历史实体：

- **HistoricProcessInstance** 保存当前与已结束流程实例的信息。
- **HistoricVariableInstance** 保存流程变量或任务变量的最新值。
- **HistoricActivityInstance** 保存活动（流程中的节点）的单一执行信息。
- **HistoricTaskInstance** 保存当前与过去（完成并删除的）任务实例的信息。
- **HistoricDetail** 保存与历史流程实例，活动实例或任务实例有关的多种信息。

因为数据库为过去与当前进行中的实例都保存历史实体，因此你可能希望查询这些表，以减少访问运行时流程实例数据，并提高运行时执行性能。

之后，这些信息将在Activiti Explorer中暴露。并且，也将用于生成报告。

11.1. 查询历史 Querying history

可以使用API查询全部5种历史实体，HistoryService暴露的

createHistoricProcessInstanceQuery()、**createHistoricVariableInstanceQuery()**、**createHistoricActivityInstanceQuery()** 与 **createHistoricTaskInstanceQuery()** 方法。

下面是一些例子，展示了历史查询API的一些用法。关于各用法的全部描述可以在javadoc中找到，在 **org.activiti.engine.history** 包中。

11.1.1. 历史流程实例查询 HistoricProcessInstanceQuery

取得所有流程中，前10个花费最多时间完成（最长持续时间）的，定义为'XXX'，已完成的**HistoricProcessInstances**。

```

1 historyService.createHistoricProcessInstanceQuery()
2     .finished()
3     .processDefinitionId("XXX")
4     .orderByProcessInstanceDuration().desc()
5     .listPage(0, 10);

```

11.1.2. 历史变量实例查询 HistoricVariableInstanceQuery

在已完成的，id为'XXX'的流程实例中，取得所有**HistoricVariableInstances**，以变量名排序。

```

1 historyService.createHistoricVariableInstanceQuery()
2     .processInstanceId("XXX")
3     .orderByVariableName.desc()
4     .list();

```

11.1.3. 历史活动实例查询 HistoricActivityInstanceQuery

取得最新的，服务任务类型的，已完成的，流程定义的id为XXX的，**HistoricActivityInstance**。

```

1 historyService.createHistoricActivityInstanceQuery()
2   .activityType("serviceTask")
3   .processDefinitionId("XXX")
4   .finished()
5   .orderByHistoricActivityInstanceEndTime().desc()
6   .listPage(0, 1);

```

11.1.4. 历史详情查询 HistoricDetailQuery

下一个例子，取得id为123的流程中，所有变量的更新记录。这个查询只会返回 **HistoricVariableUpdate**。请注意有可能某个变量名有多个 **HistoricVariableUpdate** 实体，代表流程中的每一次变量更新。可以使用 **orderByTime**（变量更新的时间）或 **orderByVariableRevision**（运行时变量更新时的版本号），按其发生顺序排序。

```

1 historyService.createHistoricDetailQuery()
2   .variableUpdates()
3   .processInstanceId("123")
4   .orderByVariableName().asc()
5   .list()

```

这个例子，取得流程id为"123"的，任何任务中或启动时提交的，所有**表单参数**。这个查询只返回 **HistoricFormProperties**。

```

1 historyService.createHistoricDetailQuery()
2   .formProperties()
3   .processInstanceId("123")
4   .orderByVariableName().asc()
5   .list()

```

最后一个例子，取得id为"123"的任务进行的所有变量更新操作。将返回该任务设置的所有变量（任务局部变量）的 **HistoricVariableUpdates**，而不会返回流程实例中设置的。

```

1 historyService.createHistoricDetailQuery()
2   .variableUpdates()
3   .taskId("123")
4   .orderByVariableName().asc()
5   .list()

```

可以在 **TaskListener** 中使用 **TaskService** 或 **DelegateTask** 设置任务局部变量：

```

1 taskService.setVariableLocal("123", "myVariable", "Variable value");

```

```

1 public void notify(DelegateTask delegateTask) {
2     delegateTask.setVariableLocal("myVariable", "Variable value");
3 }

```

11.1.5. 历史任务示例查询 HistoricTaskInstanceQuery

取得所有任务中，前10个花费最多时间完成（最长持续时间）的，已完成的 **HistoricTaskInstance**。

```

1 historyService.createHistoricTaskInstanceQuery()
2   .finished()
3   .orderByHistoricTaskInstanceDuration().desc()
4   .listPage(0, 10);

```

取得删除原因包含"invalid"，最后一次指派给'kermit'用户的 **HistoricTaskInstance**。

```

1 historyService.createHistoricTaskInstanceQuery()
2   .finished()
3   .taskDeleteReasonLike("%invalid%")
4   .taskAssignee("kermit")
5   .listPage(0, 10);

```

11.2. 历史配置 History configuration

可以使用org.activiti.engine.impl.history.HistoryLevel枚举（或在5.11版本前， **ProcessEngineConfiguration** 中定义的 **HISTORY** 常量），以编程方式配置历史级别：

```

1 ProcessEngine processEngine = ProcessEngineConfiguration
2   .createProcessEngineConfigurationFromResourceDefault()

```

```
3     .setHistory(HistoryLevel.AUDIT.getKey())
4     .buildProcessEngine();
```

也可以在activiti.cfg.xml或Spring上下文中配置级别：

```
1 <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
2   <property name="history" value="audit" />
3   ...
4 </bean>
```

可以配置下列历史级别：

- **none**（无）：跳过所有历史存档。对于运行时流程执行来说，是性能最高的配置，但是不会保存任何历史信息。
- **activity**（活动）：存档所有流程实例与活动实例。在流程实例结束时，顶级流程实例变量的最新值，将被复制为历史流程实例。不会存档细节。
- **audit**（审计）：默认级别。将存档所有流程实例，活动实例，并保持变量值以及所有提交的表单参数持续同步，以保证表单的所有用户操作都可追踪、可审计。
- **full**（完全）：历史存档的最高级别，因此也最慢。这个级别存储所有 **audit** 级别存储的信息，加上所有其他可用细节，主要是流程变量的更新。

在 **Activiti 5.11** 版本以前，历史级别保存在数据库中（**ACT_GE_PROPERTY** 表，参数名为 **historyLevel**）。从 **5.11** 开始，这个值不再使用，并从数据库中忽略/删除。现在历史可以在 **2** 个引擎的启动间切换，而不会由于前一个引擎启动修改了级别，而抛出异常。

11.3. 审计目的历史 History for audit purposes

如果至少配置为 **audit** 级别，则通过 **FormService.submitStartFormData(String processDefinitionId, Map<String, String> properties)** 与 **FormService.submitTaskFormData(String taskId, Map<String, String> properties)** 方法提交的所有参数都将被记录。

表单参数可以通过查询API，像这样读取：

```
1 historyService
2     .createHistoricDetailQuery()
3     .formProperties()
4     ...
5     .list();
```

在这个情况下，只会返回 **HistoricFormProperty** 类型的历史详情。

如果在调用提交方法前，使用 **IdentityService.setAuthenticatedUserId(String)** 设置了认证用户，则该提交了表单的认证用户可以在历史中访问。对于启动表单使用 **HistoricProcessInstance.getStartUserId()**，对于任务表单使用 **HistoricActivityInstance.getAssignee()**。

12. Eclipse Designer

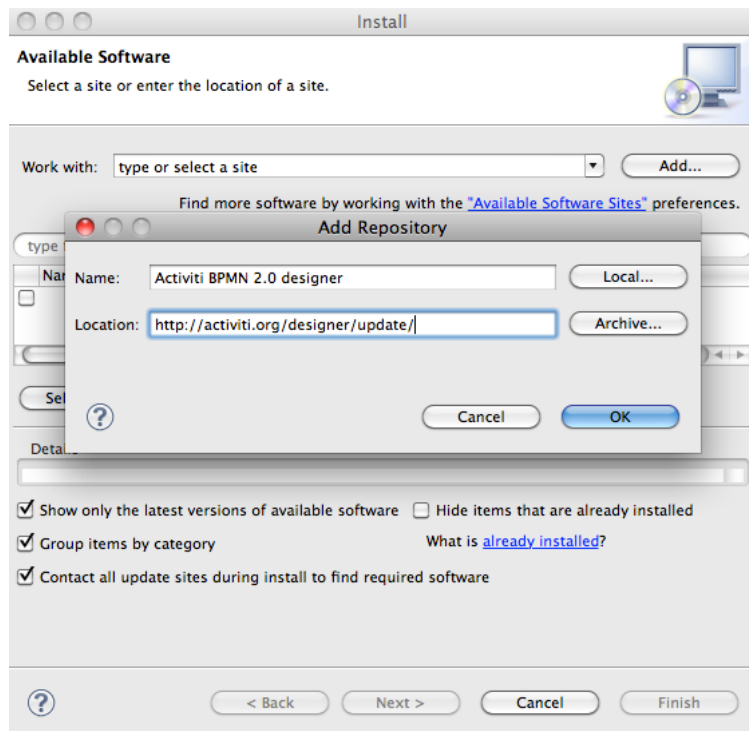
Activiti提供了名为Activiti Eclipse Designer的Eclipse插件，可以用于图形化地建模、测试与部署BPMN 2.0流程。

12.1. 安装 Installation

下面的安装指导在 **Eclipse Kepler** 与 **Indigo** 进行了验证。请注意不支持 **Eclipse Helios**。

选择 **Help → Install New Software**。在下图面板中，点击 **Add** 按钮，并填写下列字段：

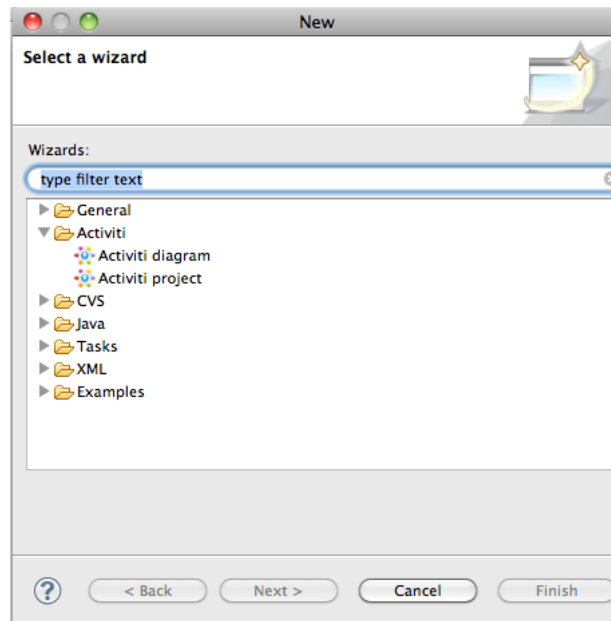
- *Name:*Activiti BPMN 2.0 designer
- *Location:*http://activiti.org/designer/update/



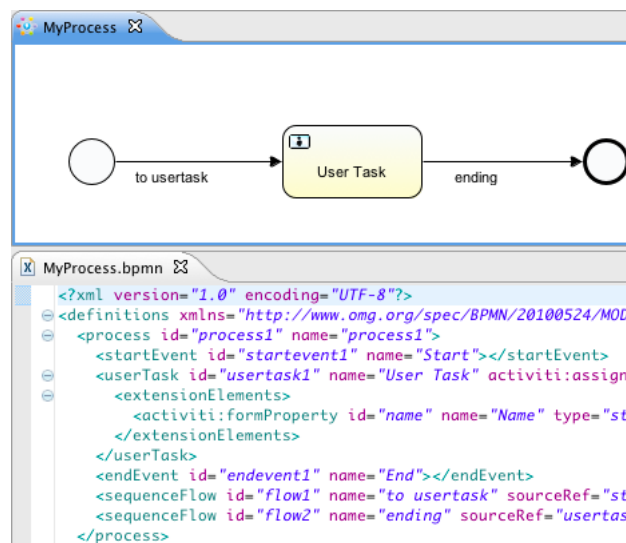
确保“Contact all updates sites..”复选框已选中，因为这样Eclipse就可以下载需要的所有插件。

12.2. Activiti Designer编辑器功能 Activiti Designer editor features

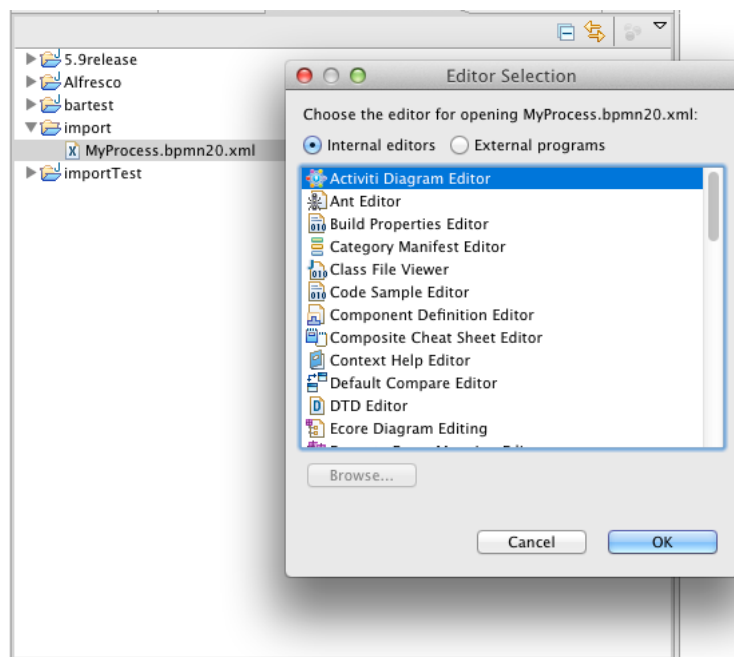
- 创建Activiti项目与流程图（diagram）。



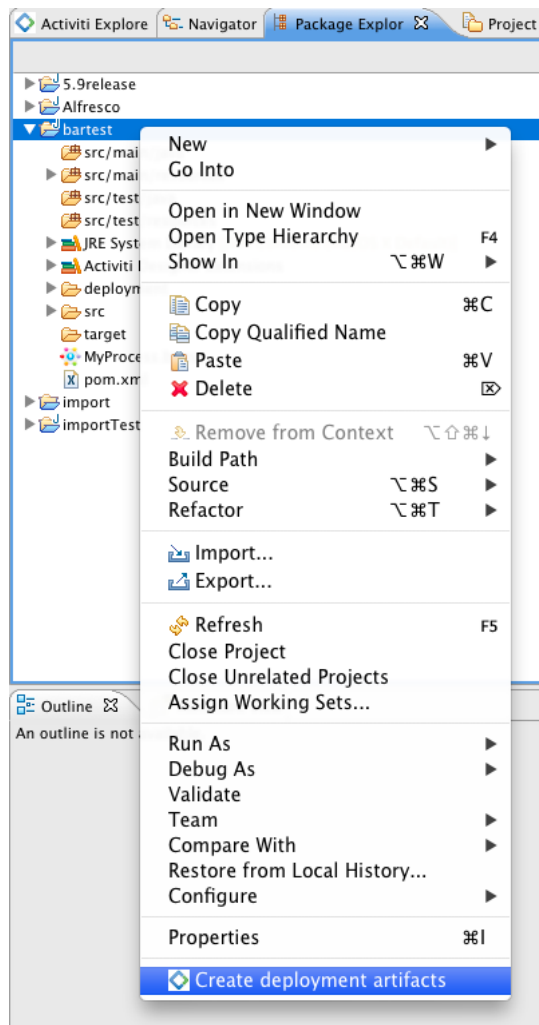
- Activiti Designer在创建新的Activiti流程图时，会创建一个.bpmn文件。当使用Activiti Diagram Editor（Activiti流程图编辑器）视图打开时，将提供图形化的模型画布与画板。这个文件也可以使用XML编辑器打开，将显示流程定义的 BPMN 2.0 XML元素。因此Activiti Designer只用一个文件，既是流程图，也是BPMN 2.0 XML。请注意在Activiti 5.9版本中，还不支持使用.bpmn扩展名作为流程定义的部署包。因此Activiti Designer的“create deployment artifacts（创建部署包）”功能，将生成一个BAR文件，与一个包含.bpmn文件内容.bpmn20.xml文件。也可以方便的自己重命名。请注意，也可以使用Activiti Diagram Editor打开.bpmn20.xml文件。



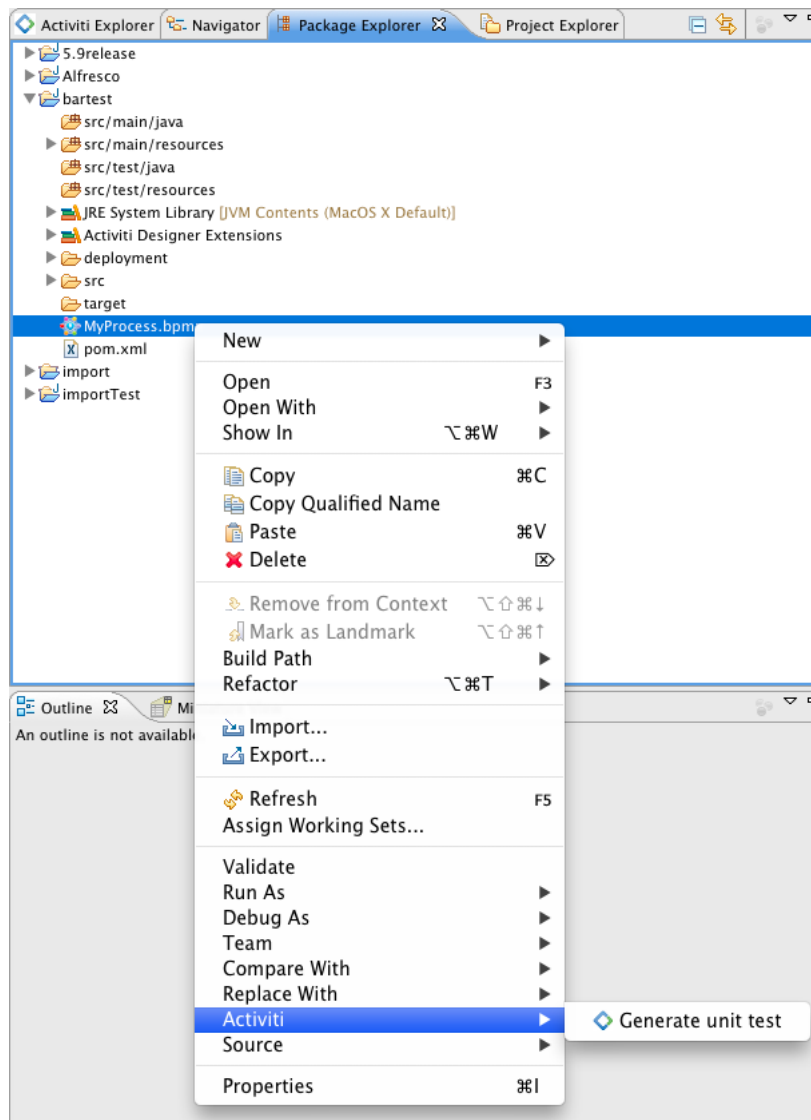
- 可以将BPMN 2.0 XML文件导入Activiti Designer，会自动创建流程图。只需要将BPMN 2.0 XML文件复制到项目中，并使用Activiti Diagram Editor视图打开它。Activiti Designer使用文件中的BPMN DI信息来创建流程图。如果BPMN 2.0 XML文件中没有BPMN DI信息，则不会创建流程图。



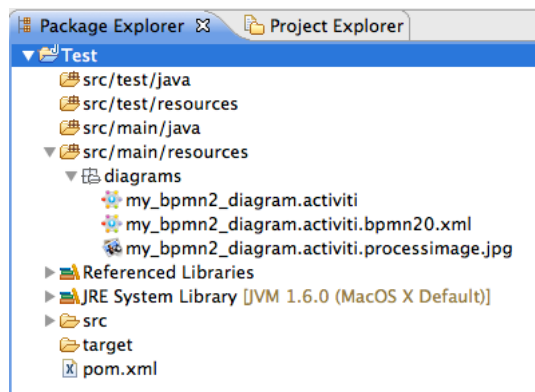
- 要进行部署，可以使用Activiti Designer创建BAR文件，或JAR文件。在包浏览器中的Activiti项目上点击右键，在弹出菜单的下方选择 *Create deployment artifacts*（创建部署包）选项。要了解关于Designer部署功能的更多信息，请查看 [部署](#) 章节。



- 生成单元测试（在包浏览器中的BPMN 2.0 XML文件上点击右键，选择`generate unit test`生成单元测试）。将创建一个单元测试及运行在嵌入式H2数据库上的Activiti配置。这样就可以运行单元测试，来测试你的流程定义。

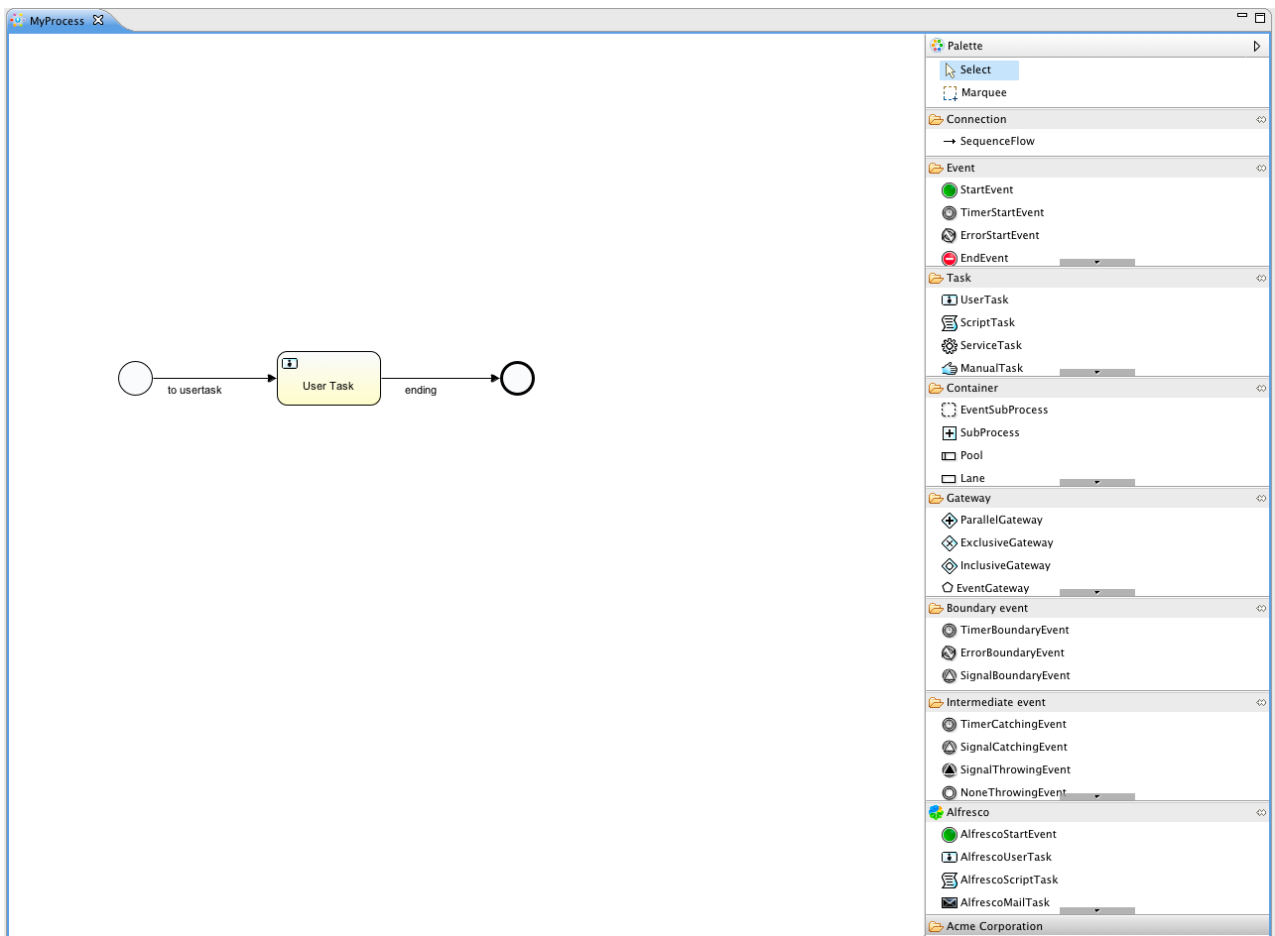


- Activiti项目可以生成Maven项目。要配置依赖，需要运行`mvn eclipse:eclipse`。请注意在流程设计时，不需要Maven依赖。只在运行单元测试时才需要依赖。

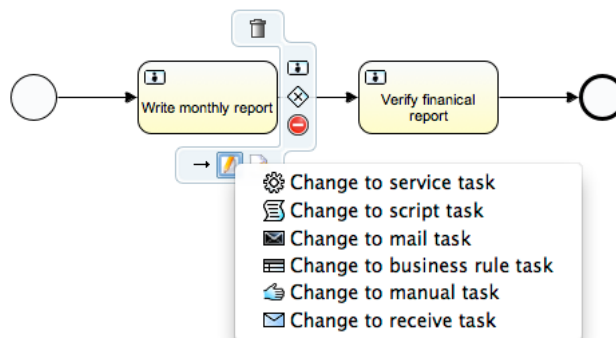


12.3. Activiti Designer BPMN功能 Activiti Designer BPMN features

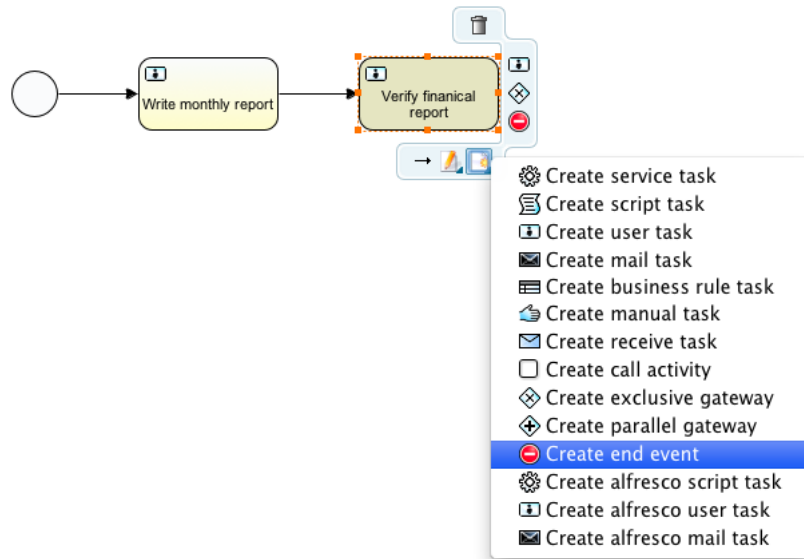
- 支持空启动事件，错误启动事件，定时器启动事件，空结束事件，错误结束事件，顺序流，并行网关，排他网关，包容网关，事件网关，嵌入子流程，事件子流程，调用活动，泳道，泳道，脚本任务，用户任务，服务任务，邮件任务，手动任务，业务规则任务，接收任务，定时器边界事件，错误边界事件，信号边界事件，定时器捕获事件，信号捕获事件，信号抛出事件，空抛出事件，与四个Alfresco特有元素（用户，脚本，邮件任务与启动事件）。



- 可以在元素上悬停并选择新的任务类型，快速改变任务的类型。



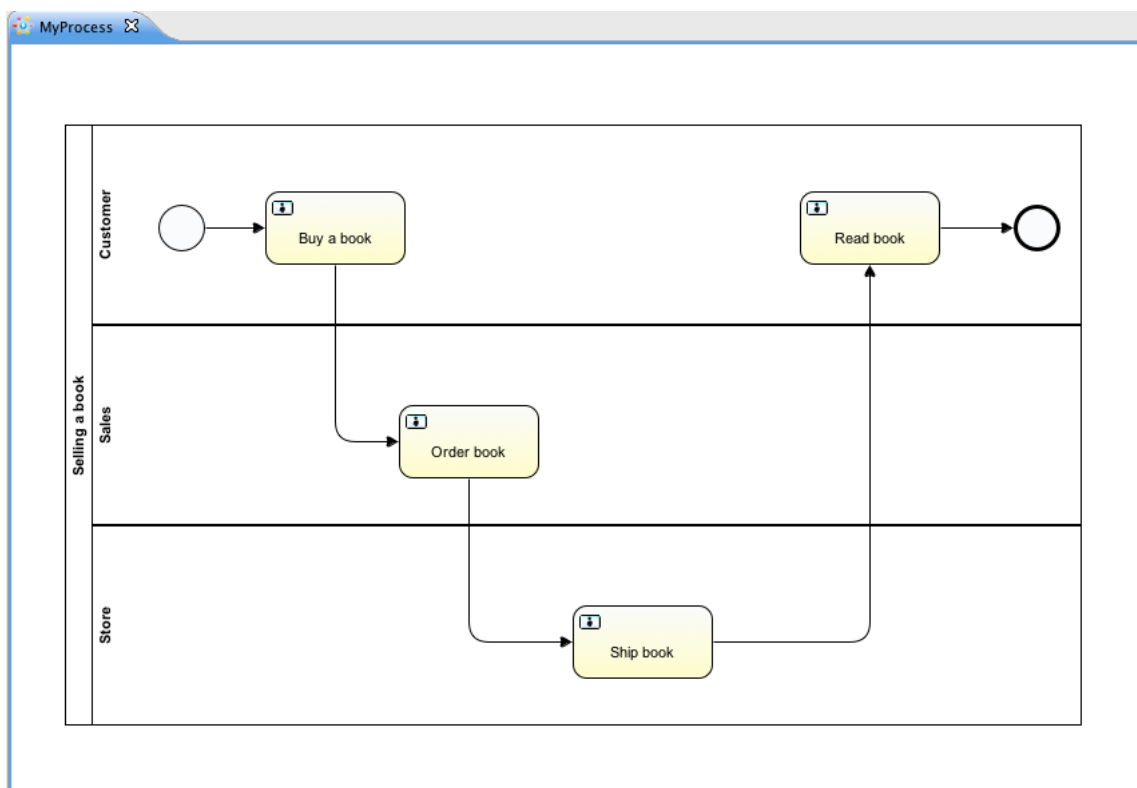
- 可以在元素上悬停并选择新的元素类型，快速添加新的元素。



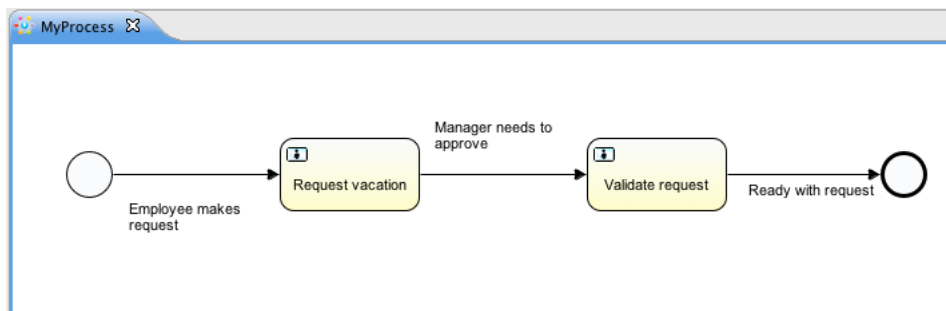
- Java服务任务支持Java类，表达式或代理表达式配置。另外也可以配置字段扩展。

General	Type:	<input type="radio"/> Java class <input checked="" type="radio"/> Expression <input type="radio"/> Delegate expression												
Main config	Expression:	<code>\${printer.printMessage}</code>												
Listeners	Result variable:													
	Fields:	<table border="1"> <thead> <tr> <th>Field name</th> <th>String value / Expression</th> </tr> </thead> <tbody> <tr> <td>message</td> <td>hello</td> </tr> <tr><td> </td><td> </td></tr> <tr><td> </td><td> </td></tr> <tr><td> </td><td> </td></tr> <tr><td> </td><td> </td></tr> </tbody> </table>	Field name	String value / Expression	message	hello								
Field name	String value / Expression													
message	hello													

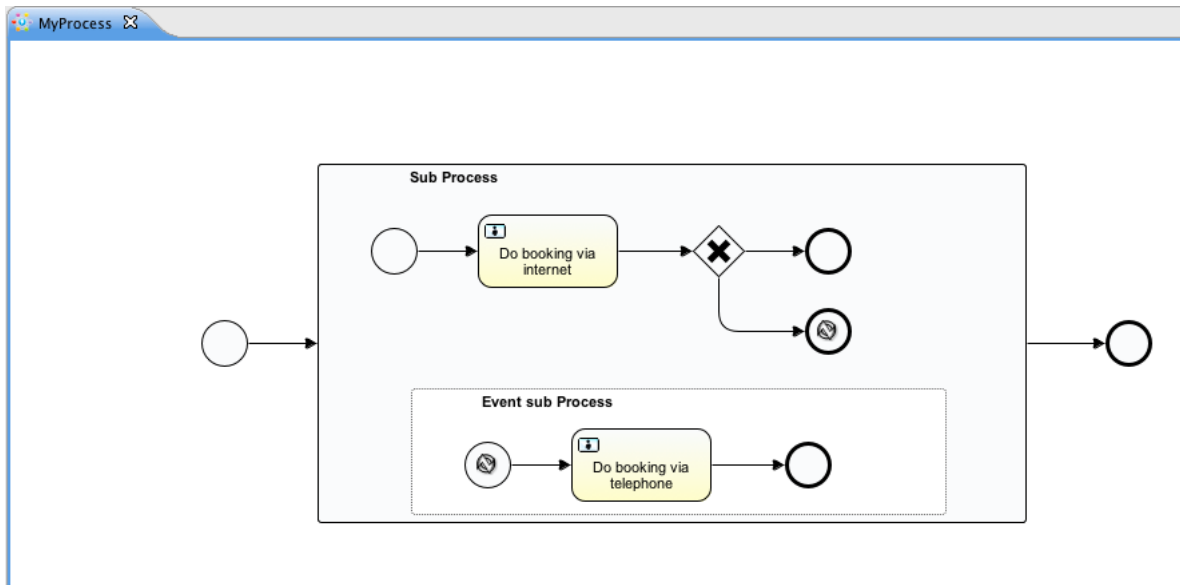
- 支持泳池与泳道。但因为Activiti将不同的泳池认作不同的流程定义，因此最好只使用一个泳池。如果使用多个泳池，要小心不要在泳池间画顺序流，否则会在Activiti引擎中部署流程时发生错误。可以在一个泳池中添加任意多的泳道。



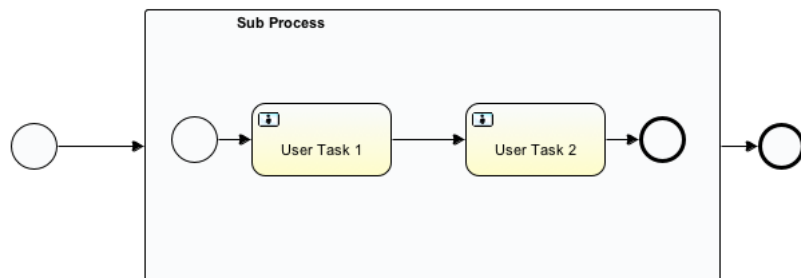
- 可以通过填写name参数，为顺序流添加标签。可以决定放置标签的位置，位置将保存为BPMN 2.0 XML DI信息的一部分。



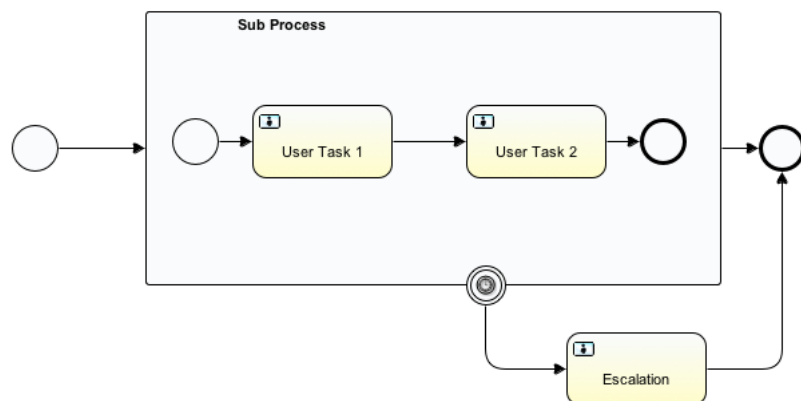
- 支持事件子流程。



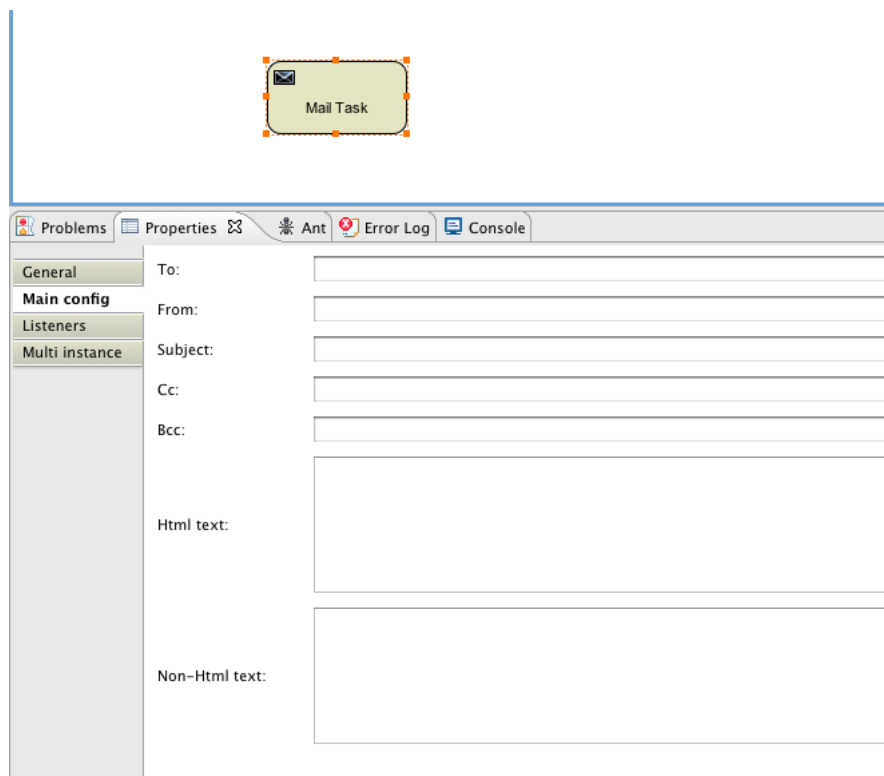
- 支持展开嵌入式子流程。也可以在一个嵌入式子流程中加入另一个嵌入式子流程。



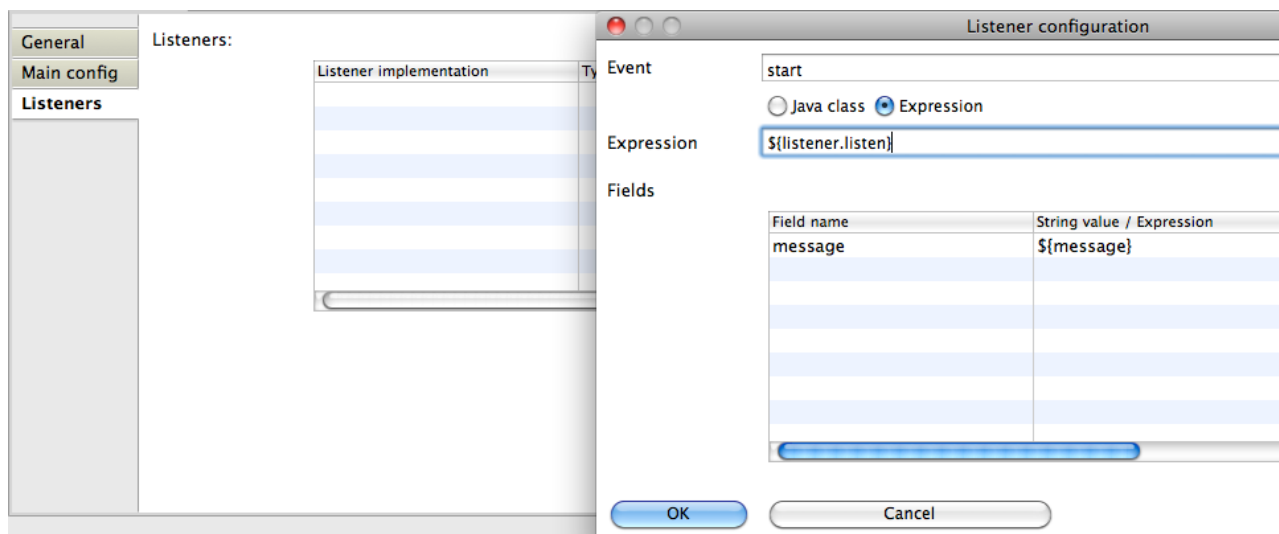
- 支持在任务与嵌入式子流程上的定时器边界事件。然而，在Activiti Designer中，在用户任务或嵌入式子流程上使用定时器边界事件最合理。



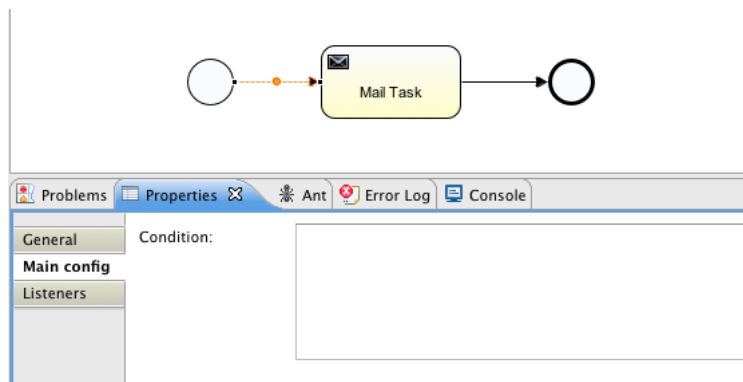
- 支持额外的Activiti扩展，例如邮件任务，用户任务的候选人配置，或脚本任务配置。



- 支持Activiti执行与任务监听器。也可以为执行监听器添加字段扩展。

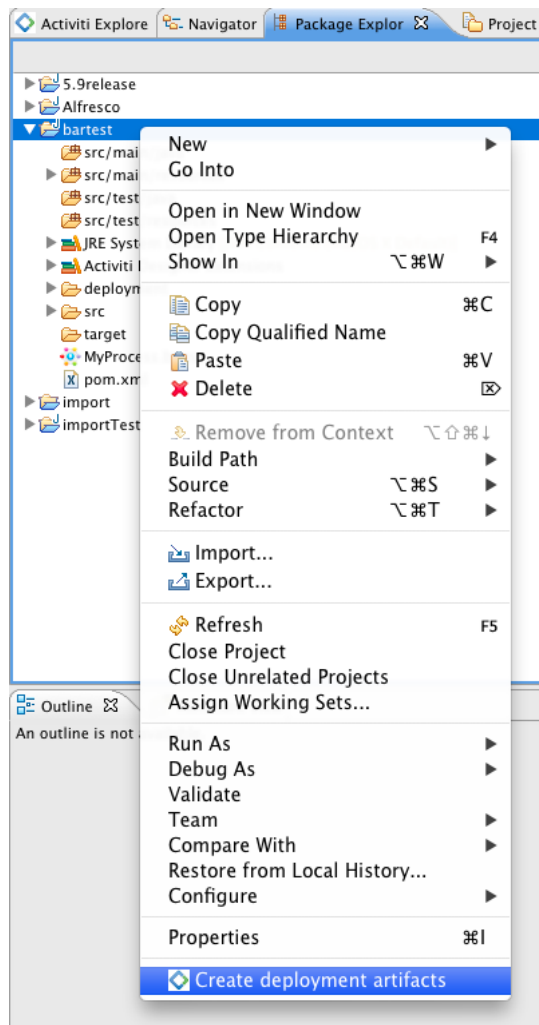


- 支持在顺序流上添加条件。

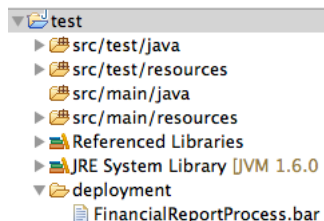


12.4. Activiti Designer部署功能 Activiti Designer deployment features

在Activiti引擎上部署流程定义与任务表单并不困难。需要有一个包含有流程定义BPMN 2.0 XML文件的BAR文件，与可选的用于在Activiti Explorer中查看的任务表单和流程图片。在Activiti Designer中，创建BAR文件十分简单。在完成流程实现后，只要在包浏览器中的Activiti项目上点击右键，在弹出菜单下方选择**Create deployment artifacts**（创建部署包）选项。



然后就会创建一个部署目录，包含BAR文件，与可能的JAR文件。其中JAR文件包含Activiti项目中的Java类。



这样就可以在Activiti Explorer的部署页签中，将这个文件上传至Activiti引擎。

如果项目包含Java类，部署时要多做些工作。在这种情况下，Activiti Designer的**Create deployment artifacts**（创建部署包）操作也会创建包含编译后类的JAR文件。这个JAR文件必须部署在Activiti Tomcat安装目录的activiti-XXX/WEB-INF/lib目录下。这将为Activiti引擎的classpath添加这些类。

12.5. 扩展Activiti Designer (Extending Activiti Designer)

可以扩展Activiti Designer提供的默认功能。这段文档介绍了可以使用哪些扩展，如何使用，并提供了一些例子。在建模业务流程时，如果默认功能不能满足需要，需要额外的功能，或有领域专门需求的时候，扩展Activiti Designer就很有用。扩展Activiti Designer分为两个不同领域，扩展画板与扩展输出格式。两种方式都需要专门的方法，与不同的技术知识。



扩展Activiti Designer需要专业知识，更确切地说，Java编程的知识。取决于你想要创建的扩展类型，你可能需要熟悉Maven，Eclipse，OSGi，Eclipse扩展与SWT。

12.5.1. 自定义画板 Customizing the palette

可以自定义为用户建模流程提供的画板。画板是形状的集合，显示在画布的右侧，可以将形状拖放至画布中的流程图上。在默认画板中可以看到，默认形状进行了分组（被称为“抽屉 drawer”），如事件，网关，等等。Activiti Designer提供了两种选择，用于自定义画板中的抽屉与形状：

- 将你自己的形状/节点添加到已有或新建的抽屉
- 禁用Activiti Designer提供的部分或全部BPMN 2.0默认形状，除了连线与选择工具

要自定义画板，需要创建一个JAR文件，并加入Activiti Designer安装目录（后面介绍[如何做](#)）。这个JAR文件叫做扩展（*extension*）。通过编写扩展中包含的类，就能让Activiti Designer知道你需要自定义什么。要做到这个，你的类需要实现特定的接口。有一个集成类库，包含这些接口以及需要加入classpath的用于扩展的基类。

可以在下列地方找到代码示例：Activiti源码的 `projects/designer` 目录下的 `examples/money-tasks` 目录。



可以使用你喜欢的任何工具设置项目，并使用你选择的构建工具构建JAR。在下面的介绍中，假设使用Eclipse Kepler或Indigo，并使用Maven（3.x）作为构建工具。但任何设置都可以创建相同的结果。

设置扩展 Extension setup (Eclipse/Maven)

下载并解压缩Eclipse（应该可以使用最新版本），与Apache Maven近期的版本（3.x）。如果使用2.x版本的Maven，可能会在构建项目时遇到错误，因此请确保版本是最新的。我们假设你已经熟悉Eclipse中的基本功能以及Java编辑器。可以使用Eclipse的Maven功能，或直接从命令行运行Maven命令。

在Eclipse中创建一个新项目。可以是通用类型项目。在项目的根路径创建一个 `pom.xml` 文件，以包含Maven项目配置。同时创建 `src/main/java` 与 `src/main/resources` 目录，这是Maven约定的Java源文件与资源文件目录。打开 `pom.xml` 文件并添加下列行：

```
1 <project
2   xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
5
6   <modelVersion>4.0.0</modelVersion>
7
8   <groupId>org.acme</groupId>
9   <artifactId>money-tasks</artifactId>
10  <version>1.0.0</version>
11  <packaging>jar</packaging>
12  <name>Acme Corporation Money Tasks</name>
13  ...
14 </project>
```

可以看到，这只是一个基础的pom.xml文件，为项目定义了一个 `groupId`，`artifactId` 与 `version`。我们会创建一个定制项，包含一个money业务的自定义节点。

在 `pom.xml` 文件中添加这些依赖，将集成库添加至项目依赖：

```
1 <dependencies>
2   <dependency>
3     <groupId>org.activiti.designer</groupId>
4     <artifactId>org.activiti.designer.integration</artifactId>
5     <version>5.12.0</version> <!-- Use the current Activiti Designer version -->
6     <scope>compile</scope>
7   </dependency>
8 </dependencies>
9 ...
10 <repositories>
11   <repository>
12     <id>Activiti</id>
13     <url>https://maven.alfresco.com/nexus/content/groups/public/</url>
14   </repository>
15 </repositories>
```

最后，在 `pom.xml` 文件中，添加 `maven-compiler-plugin` 配置，设置Java源码级别为1.5以上（参见下面的代码片段）。要使用注解需要这个配置。也可以为Maven包含用于生成JAR的 `MANIFEST.MF` 文件。这不是必须的，但可以在这个manifest中使用特定参数，为你的扩展提供名字（这个名字可以在设计器的特定位置显示，主要用于在设计器中有多个扩展时使用）。如果想要这么做，在 `pom.xml` 中添加下列代码片段：

```
1 <build>
2   <plugins>
3     <plugin>
4       <artifactId>maven-compiler-plugin</artifactId>
5       <configuration>
6         <source>1.5</source>
7         <target>1.5</target>
8         <showDeprecation>true</showDeprecation>
9         <showWarnings>true</showWarnings>
10        <optimize>true</optimize>
11      </configuration>
12    </plugin>
13    <plugin>
14      <groupId>org.apache.maven.plugins</groupId>
15      <artifactId>maven-jar-plugin</artifactId>
16      <version>2.3.1</version>
17      <configuration>
```



```

18     <archive>
19     <index>true</index>
20     <manifest>
21     <addClasspath>false</addClasspath>
22     <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
23     </manifest>
24     <manifestEntries>
25     <ActivitiDesigner-Extension-Name>Acme Money</ActivitiDesigner-Extension-Name>
26     </manifestEntries>
27     </archive>
28 </configuration>
29 </plugin>
30 </plugins>
31 </build>

```

扩展的名字使用 `ActivitiDesigner-Extension-Name` 参数描述。现在只剩下让Eclipse按照 `pom.xml` 的指导设置项目。因此打开命令行，并转到Eclipse工作空间中你项目的根目录。然后执行下列Maven命令：

```
mvn eclipse:eclipse
```

等待构建完成。刷新项目（使用项目上下文菜单（右键点击），并选择 **Refresh** 刷新）。现在Eclipse项目中应该已经建立了 `src/main/java` 与 `src/main/resources` 源码目录。



当然也可以使用 `m2eclipse` 插件，并简单地在项目的上下文菜单（右键点击）中启用 **Maven** 依赖管理。然后在项目的上下文菜单中选择 **Maven** > **Update project configuration**（更新项目配置）。这也将配置源代码目录。

这就完成了配置。现在可以开始为Activiti Designer创建自定义项了！

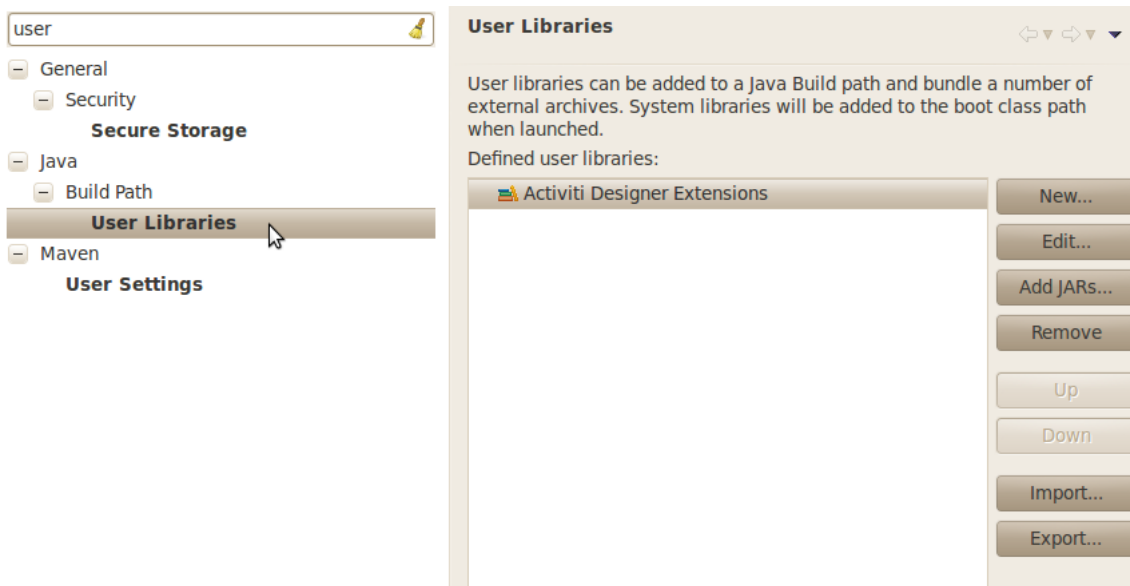
在Activiti Designer中应用你的扩展 Applying your extension to Activiti Designer

你也许想知道如何将你的扩展加入Activiti Designer，以便应用你的自定义项。需要这些步骤：

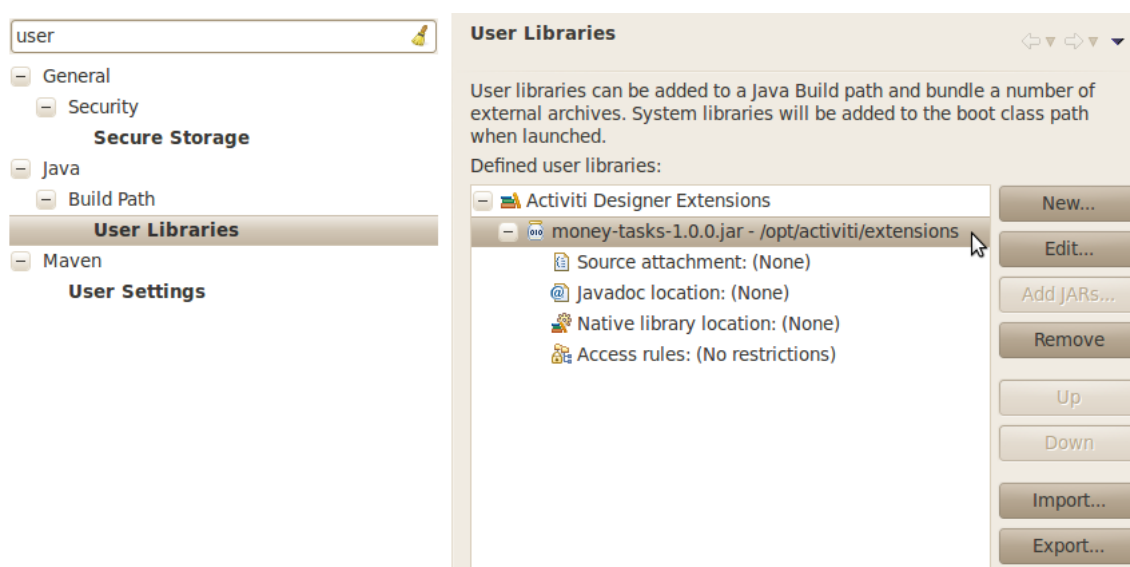
- 创建扩展JAR（例如，使用Maven构建时，在项目中运行 `mvn install`）后，需要将扩展传递至Activiti Designer安装的计算机；
- 将扩展存储在硬盘上，方便记忆的位置。请注意：必须保存在Activiti Designer的Eclipse工作空间之外——将扩展保存在工作空间内，会导致弹出错误消息弹框，扩展将不可用；
- 启动Activiti Designer，从菜单中，选择 **Window** > **Preferences**
- 在Preferences界面，键入 `user` 作为关键字。将可以看到在Eclipse中 **Java** 段落内， **User Libraries** 的选项。



- 选择 **User Libraries** 选项，将在右侧显示树形界面，可以添加库。应该可以看到一个默认组，可以用于添加Activiti Designer的扩展（根据Eclipse安装不同，也可能看到几个其他的）。



- 选择 **Activiti Designer Extensions** 组，并点击 **Add JARs...** 按钮。跳转至存储扩展的目录，并选择希望添加的扩展文件。完成后，配置界面会将扩展作为 **Activiti Designer Extensions** 组的成员进行显示，像下面这样。



- 点击 **OK** 按钮保存并关闭配置对话框。**Activiti Designer Extensions** 会自动添加至你创建的新 **Activiti** 项目。可以在导航条或包管理器的项目树下的用户库条目中看到。如果工作空间中已经有了 **Activiti** 项目，也可以看到组中显示了新扩展，像下面这样。



打开的流程图将在其画板上显示新扩展的图形（或者禁用部分图形，取决于扩展中的配置）。如果已经打开了流程图，关闭并重新打开就能在画板上看到变化。

为画板添加图形 Adding shapes to the palette

项目配置完后，可以很轻松的为画板添加图形。每个添加的图形都表现为JAR中的一个类。请注意这些类并不是 **Activiti** 引擎运行时使用的类。在扩展中可以为每个图形描述 **Activiti Designer** 可用的参数。在这些图形中，也可以定义运行时特性，并将由引擎在流程实例到达该节点时使用。运行时特性可以使用任何 **Activiti** 对普通 **ServiceTask** 支持的选项。查看[这个章节](#)了解更多信息。

图形的类是简单的 **Java** 类，加上一些注解。这个类需要实现 **CustomServiceTask** 接口，但不应该直接实现这个接口，而应该扩展 **AbstractCustomServiceTask** 基类（目前必须直接扩展这个类，而不能在中间使用 **abstract** 类）。在这个类的 **Javadoc** 中，可以看到其默认提供的，与需要覆盖的方法介绍。覆盖可以实现很多功能，例如为画板及画布中的图形提供图标（两个可以不一样），或者指定你希望节点实现的基图形（活动，时间，网关）。

```

1  * @author John Doe
2  * @version 1
3  * @since 1.0.0
4  */
5  public class AcmeMoneyTask extends AbstractCustomServiceTask {
6  ...
7  }
8

```

需要实现 `getName()` 方法，来决定节点在画板上的名字。也可以将节点放在自己的抽屉中，并提供图标，只需要覆盖 `AbstractCustomServiceTask` 的对应方法就可以。如果希望提供图标，请确保放在JAR的 `src/main/resources` 包中，需要是16X16像素的JPEG或PNG格式图片。你要提供的路径是到这个目录的相对路径。

可以通过在类中添加成员，并使用 `@Property` 注解，来为形状添加参数。像这样：

```

1  @Property(type = PropertyType.TEXT, displayName = "Account Number")
2  @Help(displayHelpShort = "提供一个账户编码 Provide an account number", displayHelpLong = HELP_ACCOUNT_NUMBER_LONG)
3  private String accountNumber;

```

可以使用多种 `PropertyType` 值，在这个章节中详细描述。可以通过将 `required` 属性设置为 `true`，将一个字段设为必填。如果用户没有填写这个字段，将会提示消息，背景也会变红。

如果想要确保类中多个参数在参数界面上的显示顺序，需要指定 `@Property` 注解的 `order` 属性。

可以看到有个 `@Help` 注解，它用于为用户提供一些填写字段的指导。也可以在类本身上使用 `@Help` 注解——这个信息将在显示给用户的参数表格最上面显示。

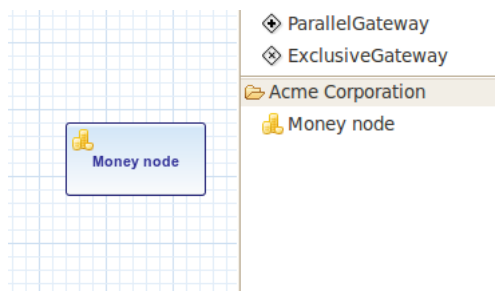
下面是 `MoneyTask` 详细介绍的列表。添加了一个备注字段，也可以看到节点包含了一个图标。

```

1  /**
2   * @author John Doe
3   * @version 1
4   * @since 1.0.0
5   */
6  @Runtime(javaDelegateClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
7  @Help(displayHelpShort = "创建一个新的账户 Creates a new account", displayHelpLong = "使用给定的账户编码，创建一个新的账户 Creates a new account using the account number specified")
8  public class AcmeMoneyTask extends AbstractCustomServiceTask {
9
10     private static final String HELP_ACCOUNT_NUMBER_LONG = "提供一个可用作账户编码的编码。 Provide a number that is suitable as an account number.";
11
12
13     @Property(type = PropertyType.TEXT, displayName = "Account Number", required = true)
14     @Help(displayHelpShort = "提供一个账户编码 Provide an account number", displayHelpLong = HELP_ACCOUNT_NUMBER_LONG)
15     private String accountNumber;
16
17
18     @Property(type = PropertyType.MULTILINE_TEXT, displayName = "Comments")
19     @Help(displayHelpShort = "提供备注 Provide comments", displayHelpLong = "可以为节点添加备注，以提供详细说明。 You can add comments to the node to provide a brief description.")
20     private String comments;
21
22
23     /*
24      * (non-Javadoc)
25      *
26      * @see org.activiti.designer.integration.servicetask.AbstractCustomServiceTask #contributeToPaletteDrawer()
27      */
28     @Override
29     public String contributeToPaletteDrawer() {
30         return "Acme Corporation";
31     }
32
33     @Override
34     public String getName() {
35         return "Money node";
36     }
37
38     /*
39      * (non-Javadoc)
40      *
41      * @see org.activiti.designer.integration.servicetask.AbstractCustomServiceTask #getSmallIconPath()
42      */
43     @Override
44     public String getSmallIconPath() {
45         return "icons/coins.png";
46     }
47 }

```

如果使用这个图形扩展 `Activiti Designer`，画板与相应的图形将像是这样：



money任务的参数界面在下面显示。请注意 `accountNumber` 字段的必填信息。

在创建流程图、填写参数字段时，用户可以使用静态文本，或者使用流程变量的表达式（如"This little piggy went to `#{piggyLocation}`"）。一般来说，用户可以在text字段自由填写任何文本。如果你希望用户使用表达式，并（使用 `@Runtime`）为 `CustomServiceTask` 添加运行时行为，请确保在代理类中使用 `Expression` 字段，以便表达式可以在运行时正确解析。可以在[这个章节](#)找到更多关于运行时行为的信息。

字段的帮助信息由每个参数右侧的按钮提供。点击该按钮将弹出显示下列内容。

配置自定义服务任务的运行时执行 `Configuring runtime execution of Custom Service Tasks`

当设置好字段，并将扩展应用至Designer后，用户就可以在建模流程时，配置服务任务的这些参数。在大多数情况下，会希望在Activiti执行流程时，使用这些用户配置参数。要做到这一点，必须告诉Activiti，当流程到达你 `CustomServiceTask` 时，需要使用哪个类。

有一个特别的注解， `@Runtime`，用于指定 `CustomServiceTask` 的运行时特性。这里有些如何使用的例子：

```
1 @Runtime(javaDelegateClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
```

使用时， `CustomServiceTask` 将会表现为流程建模BPMN中的一个普通的 `ServiceTask`。Activiti提供了多种方法定义 `ServiceTask` 的运行时特性。因此， `@Runtime` 可以使用Activiti提供的三个属性中的一个：

- `javaDelegateClass` 在BPMN输出中映射为 `activiti:class`。指定一个实现了 `JavaDelegate` 的类的全限定类名。
- `expression` 在BPMN输出中映射为 `activiti:expression`。指定一个需要执行的方法的表达式，例如一个Spring Bean中的方法。当使用这个选项时，不应在字段上指定任何 `@Property` 注解。下面有更详细的说明。
- `javaDelegateExpression` 在BPMN输出中映射为 `activiti:delegateExpression`。指定一个实现了 `JavaDelegate` 的类的表达式。

如果在类中为Activiti提供了可以注入的成员，就可以将用户的参数至注入到运行时类中。名字需要与 `CustomServiceTask` 的成员名一致。查看用户手册的[这个部分](#)了解更多信息。请注意从Designer的5.11.0版本开始，可以为动态字段值使用 `Expression` 接口。这意味着Activiti Designer中参数的值必须要是表达式，并且这个表达式将在之后注入 `JavaDelegate` 实现类的 `Expression` 参数中。



可以在 `CustomServiceTask` 的成员上使用 `@Property` 注解，但如果使用 `@Runtime` 的 `expression` 属性，则 `@Property` 注解将不会生效。原因是指定的表达式将被 `Activiti` 尝试解析为方法，而不是类。因此，不会有对类的注入。如果在 `@Runtime` 注解中使用 `expression`，则注解为 `@Property` 的成员将被 `Designer` 忽略。`Designer` 不会将它们渲染为节点参数页面的可编辑字段，也不会为这些参数在流程的 `BPMN` 中生成输出。



请注意不应该在你的扩展 `JAR` 中包括运行时类，因为它与 `Activiti` 库是分离的。`Activiti` 需要在运行时能够找到它们，因此需要将其放在 `Activiti` 引擎的 `classpath` 中。

`Designer` 代码树中的示例项目包含了配置 `@Runtime` 的不同选项的例子。可以从查看 `money-tasks` 项目开始。引用代理类的示例在 `money-delegates` 项目中。

参数类型 Property types

这个章节介绍了 `CustomServiceTask` 能够使用的参数类型，可以将类型设置为 `PropertyType` 的值。

PropertyType.TEXT

创建如下所示的单行文本字段。可以是必填字段，并将验证消息作为提示信息显示。验证失败会将字段的背景变为浅红色。

Account Number (*): This field is required ?

PropertyType.MULTILINE_TEXT

创建如下所示的多行文本字段（高度固定为80像素）。可以是必填字段，并将验证消息作为提示信息显示。验证失败会将字段的背景变为浅红色。

Comments (*): This field is required ?

PropertyType.PERIOD

创建一个组合编辑框，可以使用转盘控件编辑每一个单位的数量，来指定一段时间长度，结果如下所示。可以是必填字段（含义是不能所有的值都是0，也就是至少有一个部分要有非零值），并将验证消息作为提示信息显示。验证失败会将整个字段的背景变为浅红色。字段的值保存为 `1y 2mo 3w 4d 5h 6m 7s` 格式的字符串，代表1年，2月，3周，4天，6分钟及7秒。即使有部分为0，也总是存储整个字符串。

Processing Time (*): 0 y, 4 mo, 0 w, 4 d, 0 h, 0 m, 0 s ?

PropertyType.BOOLEAN_CHOICE

创建一个单独的 `boolean` 复选框，或者开关选择。请注意可以在 `Property` 注解上指定 `required` 属性，但不会生效，不然用户就无法选择是否选中复选框。流程图中存储的值为 `java.lang.Boolean.toString(boolean)`，其结果为 `"true"` 或 `"false"`。

VIP Customer: ☒ ?

PropertyType.RADIO_CHOICE

创建如下所示的一组单选按钮。选中任何一个单选按钮都自动排除任何其他的选择（也就是说，单选）。可以是必填字段，并将验证消息作为提示信息显示。验证失败会将组的背景变为浅红色。

这个参数类型需要注解的类成员同时使用 `@PropertyItems` 注解（例如如下所示）。可以使用这个额外的注解，以字符串数组的方式，指定条目的列表。需要为每一个条目添加两个数组项：第一个，用于显示的标签；第二个，用于存储的值。

```
1 @Property(type = PropertyType.RADIO_CHOICE, displayName = "提款限额 Withdrawl limit", required = true)
2 @Help(displayHelpShort = "最大每日提款限额 The maximum daily withdrawl amount ", displayHelpLong = "选择从该账户中每日最大能提
3 取的额度。 Choose the maximum daily amount that can be withdrawn from the account.")
4 @PropertyItems({ LIMIT_LOW_LABEL, LIMIT_LOW_VALUE, LIMIT_MEDIUM_LABEL, LIMIT_MEDIUM_VALUE, LIMIT_HIGH_LABEL,
5                  LIMIT_HIGH_VALUE })
6 private String withdrawlLimit;
```

Withdrawl limit (*): ☐ Low (250) ☒ High (2500) ☐ Medium (1000) ?

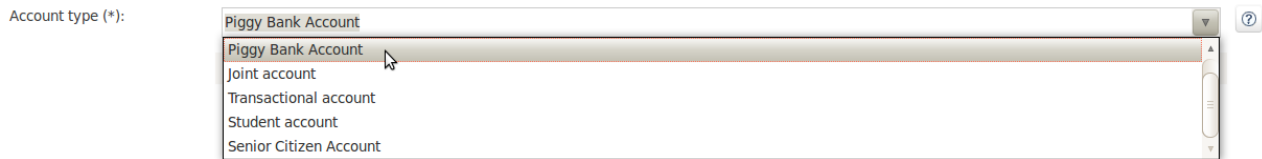
Withdrawl limit (*): ☐ Low (250) ☐ High (2500) ☐ Medium (1000) This field is required ?

PropertyType.COMBOBOX_CHOICE

创建如下所示的，带有固定选项的下拉框。可以是必填字段，并将验证消息作为提示信息显示。验证失败会将下拉框的背景变为浅红色。

这个参数类型需要注解的类成员同时使用 `@PropertyItems` 注解（例如如下所示）。可以使用这个额外的注解，以字符串数组的方式，指定条目的列表。需要为每一个条目添加两个数组项：第一个，用于显示的标签；第二个，用于存储的值。

```
1 @Property(type = PropertyType.COMBOBOX_CHOICE, displayName = "账户类型 Account type", required = true)
2 @Help(displayHelpShort = "账户的类型 The type of account", displayHelpLong = "从选项列表中选择账户的类型 Choose a type of
3 account from the list of options")
4 @PropertyItems({ ACCOUNT_TYPE_SAVINGS_LABEL, ACCOUNT_TYPE_SAVINGS_VALUE, ACCOUNT_TYPE_JUNIOR_LABEL,
5 ACCOUNT_TYPE_JUNIOR_VALUE, ACCOUNT_TYPE_JOINT_LABEL,
6 ACCOUNT_TYPE_JOINT_VALUE, ACCOUNT_TYPE_TRANSACTIONAL_LABEL, ACCOUNT_TYPE_TRANSACTIONAL_VALUE, ACCOUNT_TYPE_STUDENT_LABEL,
ACCOUNT_TYPE_STUDENT_VALUE,
ACCOUNT_TYPE_SENIOR_LABEL, ACCOUNT_TYPE_SENIOR_VALUE })
private String accountType;
```

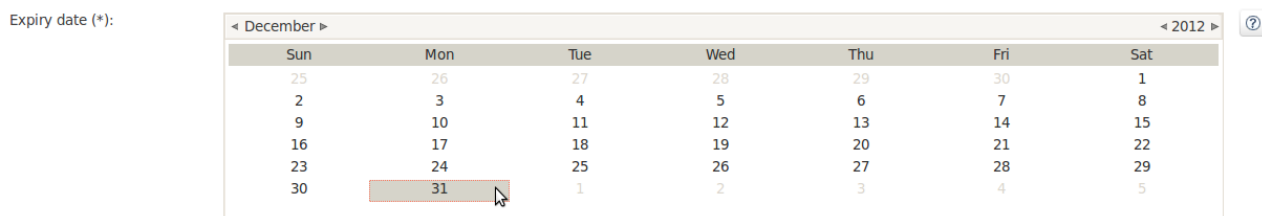


PropertyType.DATE_PICKER

创建如下所示的日期选择控件。可以是必填字段，并将验证消息作为提示信息显示（请注意，这个控件会自动填入当前系统时间，因此值很难为空）。验证失败会将控件的背景变为浅红色。

这个参数类型需要注解的类成员同时使用 `@DatePickerProperty` 注解（例如如下所示）。可以使用这个额外的注解，指定在流程图中存储日期时使用的日期格式，以及要用于显示的日期选择类型。这些属性都是可选的，当没有指定时会使用默认值（`@DatePickerProperty` 注解的静态变量）。`dateTimePattern` 属性应该使用 `SimpleDateFormat` 类支持的格式。当使用 `swtStyle` 属性时，应该指定 `SWT` 的 `DateTime` 控件支持的整形值，因为将使用这个控件渲染这个类型的参数。

```
1 @Property(type = PropertyType.DATE_PICKER, displayName = "过期日期 Expiry date", required = true)
2 @Help(displayHelpShort = "账户过期的日期 The date the account expires ", displayHelpLong = "选择一个日期，如果账户未在该日期前展
3 期，则将过期。 Choose the date when the account will expire if no extended before the date.")
4 @DatePickerProperty(dateTimePattern = "MM-dd-yyyy", swtStyle = 32)
private String expiryDate;
```



PropertyType.DATA_GRID

创建一个如下所示的数据表格控件。数据表格可以让用户输入任意行数据，并为每一行输入固定列数的值（每一组行列的组合代表一个单元格）。用户可以添加与删除行。

这个参数类型需要注解的类成员同时使用 `@DataGridProperty` 注解（例如如下所示）。可以使用这个额外的注解，指定数据表格的细节属性。需要用 `itemClass` 属性引用另一个类，来决定表格中有哪些列。`Activiti Designer` 期望其成员类型为 `List`。按照约定，可以将 `itemClass` 属性的类用作其泛型类型。如果，例如，在表格中编辑一个杂货清单，用 `GroceryListItem` 类定义表格的列。在 `CustomServiceTask` 中，可以这样引用它：

```
1 @Property(type = PropertyType.DATA_GRID, displayName = "杂货清单 Grocery List")
2 @DataGridProperty(itemClass = GroceryListItem.class)
3 private List<GroceryListItem> groceryList;
```

与 `CustomServiceTask` 一样，当使用数据表格时，"itemClass" 可以使用相同的注解指定字段类型，目前支持 `TEXT`，`MULTILINE_TEXT` 与 `PERIOD`。你会注意到不论其 `PropertyType` 是什么，表格都会为每个字段创建一个单行文本控件。这是为了表格保持整洁与可读。如果考虑下 `PERIOD` 这种 `PropertyType` 的显示模式，就可以想象出它绝不适合在表格的单元格中显示。对于 `MULTILINE_TEXT` 与 `PERIOD`，会为每个字段添加双击机制，并会为该 `PropertyType` 弹出更大的编辑器。数值将在用户点击OK后存储至字段，因此可以在表格中显示。

必选属性使用与普通 `TEXT` 字段类似的方式处理，当任何字段失去焦点时，会验证整个表格。验证失败的单元格，背景色将变为浅红色。

默认情况下，这个组件允许用户添加行，但不能决定行的顺序。如果希望允许排序，需要将 `orderable` 属性设置为 `true`，这将在每一行末尾启用按钮，以将该行在表格内上移或下移。



目前，这个参数类型不能正确注入运行时类。

Account managers:

	First name ?	Last name ?	Authorization period ?	
1.	<input type="text" value="John"/>	<input type="text" value="Doe"/>	<input type="text" value="0y 2mo 0w 0d 0h 0m 0s"/>	<div>↓</div> <div>×</div>
2.	<input type="text" value="Felix"/>	<div style="background-color: #f8d7da; height: 20px;"></div>	<div style="background-color: #f8d7da; height: 20px;"></div>	<div>↑</div> <div>×</div>
<div>+ Add item</div> <div>This field is required</div>				

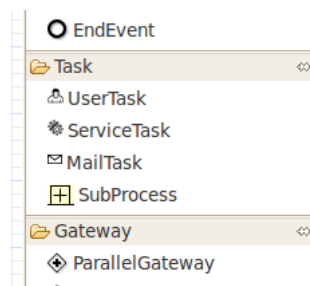
在画板中禁用默认图形 Disabling default shapes in the palette

这种自定义需要在你的扩展中引入一个实现了 `DefaultPaletteCustomizer` 接口的类。不应该直接实现这个接口，而要扩展 `AbstractDefaultPaletteCustomizer` 基类。目前，这个类不提供任何功能，但 `DefaultPaletteCustomizer` 未来的版本中会提供更多功能，这样基类将提供更多合理的默认值，这样你的扩展将在未来的版本中更好用。

扩展 `AbstractDefaultPaletteCustomizer` 需要实现一个方法，`disablePaletteEntries()`，并必须返回一个 `PaletteEntry` 值的 `list`。请注意如果从默认集合中移除图形，导致某个抽屉中没有图形，则该抽屉也会被移除。如果需要禁用所有的默认图形，只需要在结果中添加 `PaletteEntry.ALL`。作为例子，下面的代码禁用了画板中的手动任务和脚本任务图形。

```
1 public class MyPaletteCustomizer extends AbstractDefaultPaletteCustomizer {
2
3     /*
4      * (non-Javadoc)
5      *
6      * @see org.activiti.designer.integration.palette.DefaultPaletteCustomizer#disablePaletteEntries()
7      */
8     @Override
9     public List<PaletteEntry> disablePaletteEntries() {
10         List<PaletteEntry> result = new ArrayList<PaletteEntry>();
11         result.add(PaletteEntry.MANUAL_TASK);
12         result.add(PaletteEntry.SCRIPT_TASK);
13         return result;
14     }
15
16 }
```

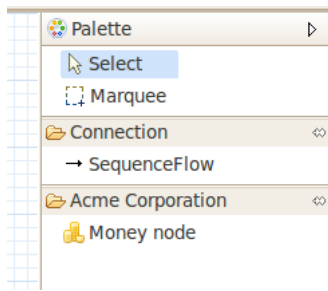
应用这个扩展的结果在下图显示。可以看到，在 `Tasks` 抽屉中不再显示手动任务与脚本任务图形。



要禁用所有默认图形，需要使用类似下面的代码。

```
1 public class MyPaletteCustomizer extends AbstractDefaultPaletteCustomizer {
2
3     /*
4      * (non-Javadoc)
5      *
6      * @see org.activiti.designer.integration.palette.DefaultPaletteCustomizer#disablePaletteEntries()
7      */
8     @Override
9     public List<PaletteEntry> disablePaletteEntries() {
10         List<PaletteEntry> result = new ArrayList<PaletteEntry>();
11         result.add(PaletteEntry.ALL);
12         return result;
13     }
14
15 }
```

结果像是这样（请注意画板中不再显示默认图形所在的抽屉）：



12.5.2. 验证流程图与输出为自定义格式 Validating diagrams and exporting to custom output formats

除了自定义画板，也可以为Activiti Designer创建扩展，来进行流程图验证，以及将流程图的信息保存为Eclipse工作空间中的自定义资源。可以通过内建的扩展点实现，这个章节将介绍如何做。



保存功能最近正在重构。我们仍在开发验证功能。下面的文档记录的是旧的情况，并将在新功能可用后更新。

Activiti Designer可以编写用于验证流程图的扩展。默认情况已经可以在工具中验证BPMN结构，但你也可以添加自己的，如果希望验证额外的条目，例如建模约定，或者CustomServiceTask中的参数值。这些扩展被称作Process Validators。

也可以在Activiti Designer保存流程图时，发布为其它格式。这些扩展被称作Export Marshallers，将在每次用户进行保存操作时，由Activiti Designer自动调用。这个行为可以在Eclipse配置对话框中，为每一种扩展检测出的格式，分别启用或禁用。Designer会根据用户的配置，确保在保存流程图时，调用你的ExportMarshaller。

通常，会想要将ProcessValidator与ExportMarshaller一起使用。例如有一些CustomServiceTask，带有一些希望在流程中使用的参数。然而，在生成流程前，希望验证其中一些值。联合使用ProcessValidator与ExportMarshaller是最佳的方式，Activiti Designer也允许你无缝拼接扩展。

要创建一个ProcessValidator或ExportMarshaller，需要创建与扩展画板不同的扩展类型。原因很简单：你的代码会需访问比集成库中提供的更多的API。特别是，会需要使用Eclipse的类。因此从一开始，就需要创建一个Eclipse插件（可以使用Eclipse的PDE支持完成），并将其打包为自定义Eclipse产品或特性。解释开发Eclipse插件的所有细节，已经不是本用户手册的范畴，因此下面的介绍仅限于扩展Activiti Designer的功能。

扩展包需要依赖下列库：

- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.activiti.designer.eclipse
- org.activiti.designer.libs
- org.activiti.designer.util

可选的，如果希望在扩展中使用，可以通过Designer使用org.apache.commons.lang包。

ProcessValidator与ExportMarshaller都是通过扩展基类创建的。这些基类从其父类AbstractDiagramWorker继承了一些有用的方法。使用这些方法，可以创建在Eclipse问题视图中显示的提示信息，警告，错误标记，以便用户了解错误与重要的信息。可以以Resources与InputStreams的格式获取流程图的信息，这些信息由DiagramWorkerContext提供，在AbstractDiagramWorker中可用。

不论是ProcessValidator还是ExportMarshaller中，做任何事情前最好调用clearMarkers()；这将清除任何已有的标记（标记自动连接至操作，清除一个操作的标记不会影响其他操作的标记）。例如：

```
1 // 首先清除流程图的标记 Clear markers for this diagram first
2 clearMarkersForDiagram();
```

也需要使用（DiagramWorkerContext中）提供的进度监控，将你的进度报告给用户，因为验证与/或保存操作可能花费很多时间，而用户只能等待。报告进度需要了解如何使用Eclipse的功能。查看[这篇文章](#)了解详细概念与用法。

创建ProcessValidator扩展 Creating a ProcessValidator extension



审核中！

在你的plugin.xml文件中，创建一个org.activiti.designer.eclipse.extension.validation.ProcessValidator扩展点的扩展。这个扩展点需要扩展AbstractProcessValidator类。

```
1 <?eclipse version="3.6"?>
2 <plugin>
3   <extension
```

```

4     point="org.activiti.designer.eclipse.extension.validation.ProcessValidator">
5     <ProcessValidator
6         class="org.acme.validation.AcmeProcessValidator">
7     </ProcessValidator>
8 </extension>
9 </plugin>

```

```

1 public class AcmeProcessValidator extends AbstractProcessValidator {
2 }

```

需要实现一些方法。最重要的是实现 `getValidatorId()`，为验证器返回全局唯一ID。这将使你可以在 `ExportMarshaller` 中调用它，或者在其他 `ExportMarshaller` 中让其他人调用你的验证器。实现 `getValidatorName()`，为验证器返回逻辑名字。这个名字将在对话框中显示给用户。`getFormatName()` 可以返回这个验证器通常验证的流程图类型。

验证工作通过 `validateDiagram()` 方法实现。从这里开始，就是你自己的功能代码了。然而，通常你会想从获取流程中的所有节点开始，这样就可以迭代访问，收集、比较与验证数据了。这段代码展示了如何进行这些操作：

```

1 final EList<EObject> contents = getResourceForDiagram(diagram).getContents();
2 for (final EObject object : contents) {
3     if (object instanceof StartEvent ) {
4         // 验证启动事件 Perform some validations for StartEvents
5     }
6     // 其它节点类型与验证 Other node types and validations
7 }

```

别忘了在验证过程中调用 `addProblemToDiagram()` 与/或 `addWarningToDiagram()` 等等。确保在结束时返回正确的boolean结果，以指示验证成功还是失败。可以由后续调用的 `ExportMarshaller` 判断下一步操作。

创建ExportMarshaller扩展 Creating an ExportMarshaller extension

在你的 `plugin.xml` 文件中，创建一个 `org.activiti.designer.eclipse.extension.ExportMarshaller` 扩展点的扩展。这个扩展点需要扩展 `AbstractExportMarshaller` 类。这个基类提供了一些在保存为你自己的格式时有用的方法，但最重要的是提供了将资源保存至工作空间，以及调用验证器的功能。

Designer的示例目录下有一个示例实现。这个示例展示了如何使用基类中的方法完成基本操作，例如访问流程图的 `InputStream`，使用其 `BpmnModel`，以及将资源保存至工作空间。

```

1 <?eclipse version="3.6"?>
2 <plugin>
3     <extension
4         point="org.activiti.designer.eclipse.extension.ExportMarshaller">
5         <ExportMarshaller
6             class="org.acme.export.AcmeExportMarshaller">
7         </ExportMarshaller>
8     </extension>
9 </plugin>

```

```

1 public class AcmeExportMarshaller extends AbstractExportMarshaller {
2 }

```

需要实现一些方法，例如 `getMarshallerName()` 与 `getFormatName()`。这些方法用来为用户显示选项，并在流程对话框中显示信息，因此请确保你返回的描述反映了正在进行的操作。

大部分工作主要在 `doMarshallDiagram()` 方法中进行。

如果需要先进行一些验证，可以直接从保存器中调用验证器。从验证器可以获得boolean结果，就可以知道验证是否成功。在大多数情况下，在流程图验证失败时不会想要进行保存，但你也可以选择仍然继续，甚至在验证失败时创建不同的资源。

一旦获取了所有需要的数据，就可以调用 `saveResource()` 方法创建保存有数据的文件。在一个保存器中，可以调用 `saveResource()` 任意多次；因此一个验证器可以创建多于一个输出文件。

可以使用 `AbstractDiagramWorker` 类的 `saveResource()` 方法构建输出资源的文件名。可以使用一些有用的变量用于创建文件名，例如 `_original-filename__my-format-name.xml`。这些变量在Javadocs中描述，通过 `ExportMarshaller` 接口定义。如果希望自行解析保存位置，也可以在一个字符串（例如一个路径）上使用 `resolvePlaceholders()`。`getURIRelativeToDiagram()` 会为你调用它。

应该使用提供的进度监控将你的进度报告给用户。[这篇文章](#)描述了如何做。

13. Activiti Explorer

Activiti Explorer是一个web应用，包含在从Activiti网站下载的Activiti中。Explorer不是一个完成的，最终用户可用的应用，而是用于实践与展示Activiti的功能。因此Explorer更像是一个示例，或者为在自己的应用中使用Activiti的用户提供灵感。另外，Explorer使用内

存数据库，但也可以轻松切换至你自己的数据库（查看WEB-INF目录下的applicationContext文件）。

登录应用后，将看到这些大图标，展示了主要功能。



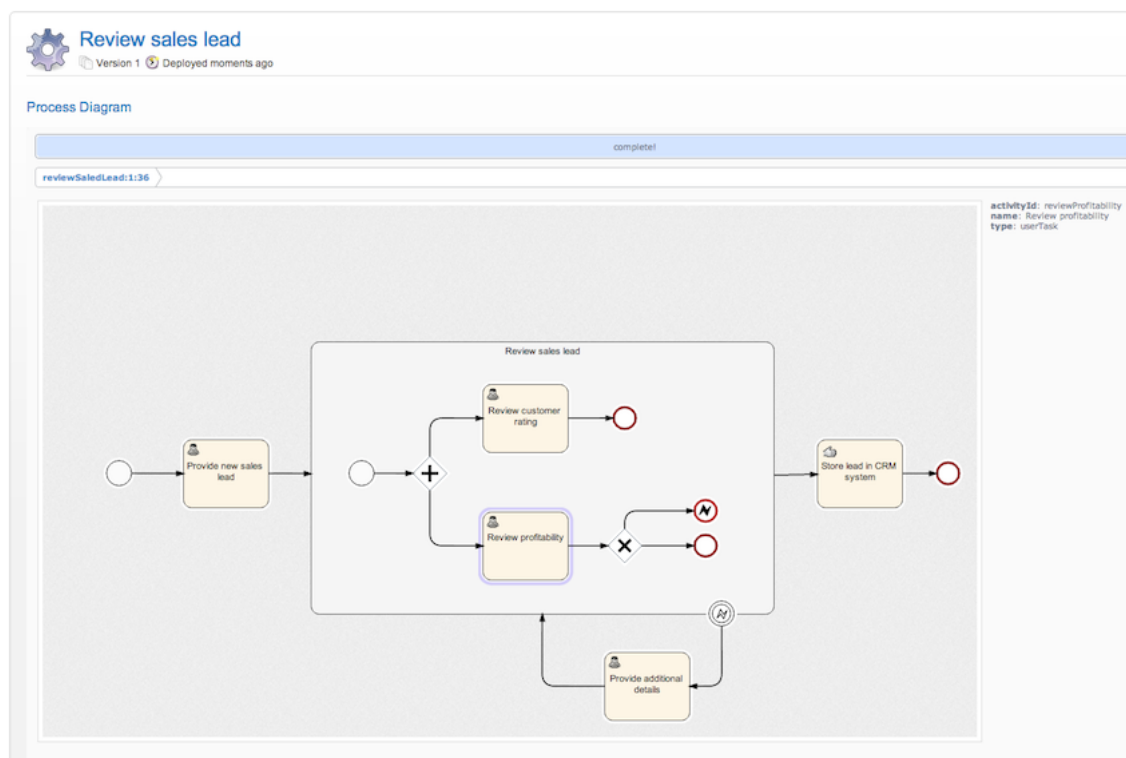
- **Tasks:** 任务管理功能。在这里可以看到指派给你的运行中用户任务的表单，或你可以申领的组任务。Explorer可以关联内容，将工作分为子任务，将人作为不同角色引入，等等。Explorer也可以创建不关联至任何流程的独立任务。
- **Process:** 展示已部署的流程定义，也可以启动新的流程实例。
- **Reporting:** 生成报告以及浏览之前保存的报告。查看[报告章节](#)了解更多细节。
- **Manage:** 只有当用户登陆为管理员权限时才可见。可以管理Activiti引擎：管理用户与组，执行与查看卡住的作业，查看数据库，以及部署新的流程定义。

13.1. 配置 Configuration

Activiti Explorer使用Spring Java配置启动Activiti引擎。可以修改WEB-INF/classes目录下的engine.properties文件，定义小部分参数。如果需要高级配置选项，可以修改同在WEB-INF/classes目录下的activiti-custom-context.xml文件，来定义Activiti流程引擎配置。该文件已经以注释形式提供了示例配置。

13.2. 流程图 Process diagram

Explorer包含了动态生成流程定义的总览的功能，使用Raphaël JavaScript框架。这个流程图只能在流程定义XML中包含BPMN DI信息时才能生成。如果流程定义XML中没有BPMN DI信息，而部署中包含有流程定义图片，则会显示该图片。



如果不希望使用JavaScript流程定义总览，可以在ui.properties文件中禁用

```
1 | activiti.ui.jsdiagram = false
```

另外，要在Explorer中显示流程图，也可以在任何你想要的地方引入流程图。下面的URL将基于流程定义id，显示流程定义图：

```
http://localhost:8080/activiti-explorer/diagram-viewer/index.html?processDefinitionId=reviewSaledLead:1:36
```

也可以通过添加processInstanceId请求参数，显示流程实例的当前状态，像是这样：

```
http://localhost:8080/activiti-explorer/diagram-viewer/index.html?processDefinitionId=reviewSaledLead:1:36&processInstanceId=41
```

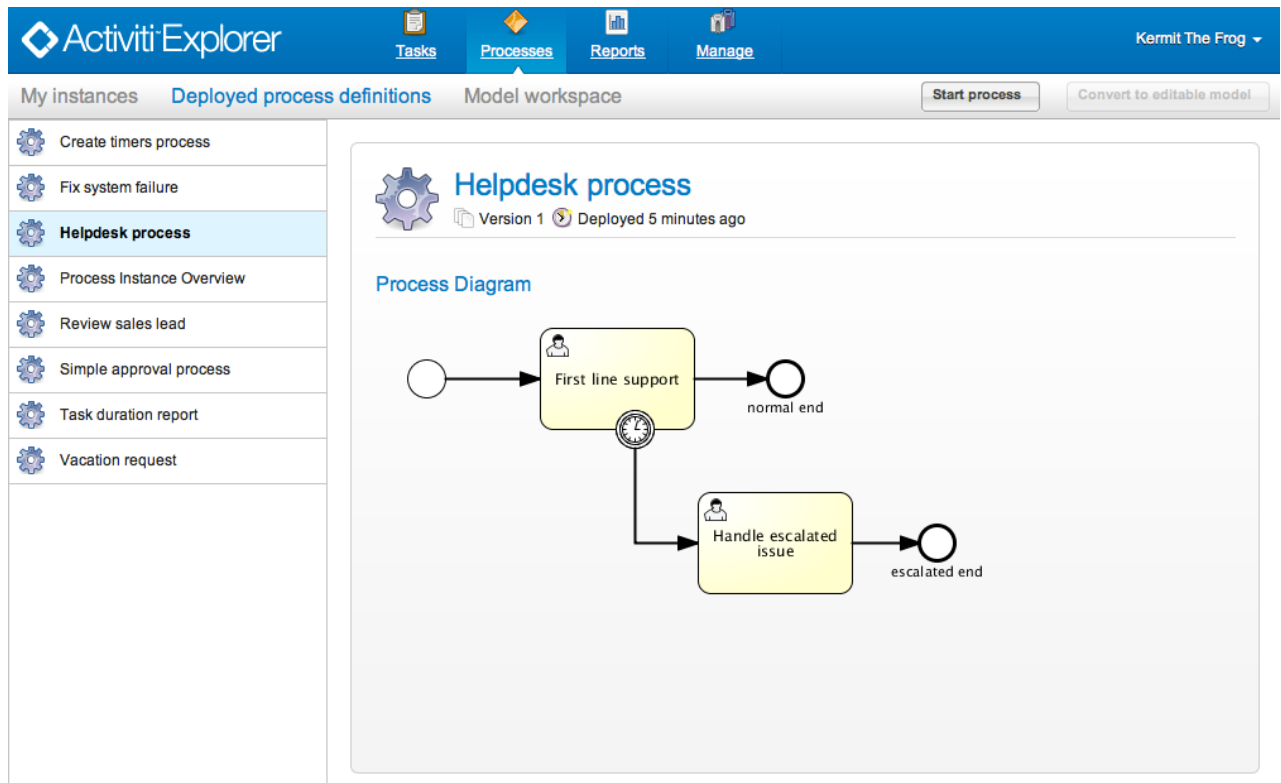
13.3. 任务 Tasks



- **Inbox:** 展示指派给当前登录用户的任务。
- **My tasks:** 展示当前登录用户作为属主的任务。当创建独立任务时，当前用户自动成为该任务的属主。
- **Queued:** 展示你所在的不同组。这里的任务必须先进行申领，然后才能完成。
- **Involved:** 展示当前登陆用户作为下列角色之一的任务：(1)相关人员（也就是说候选用户或参与者），(2)办理人，或者(3)属主。
- **Archived** 包含过去（历史）的任务。

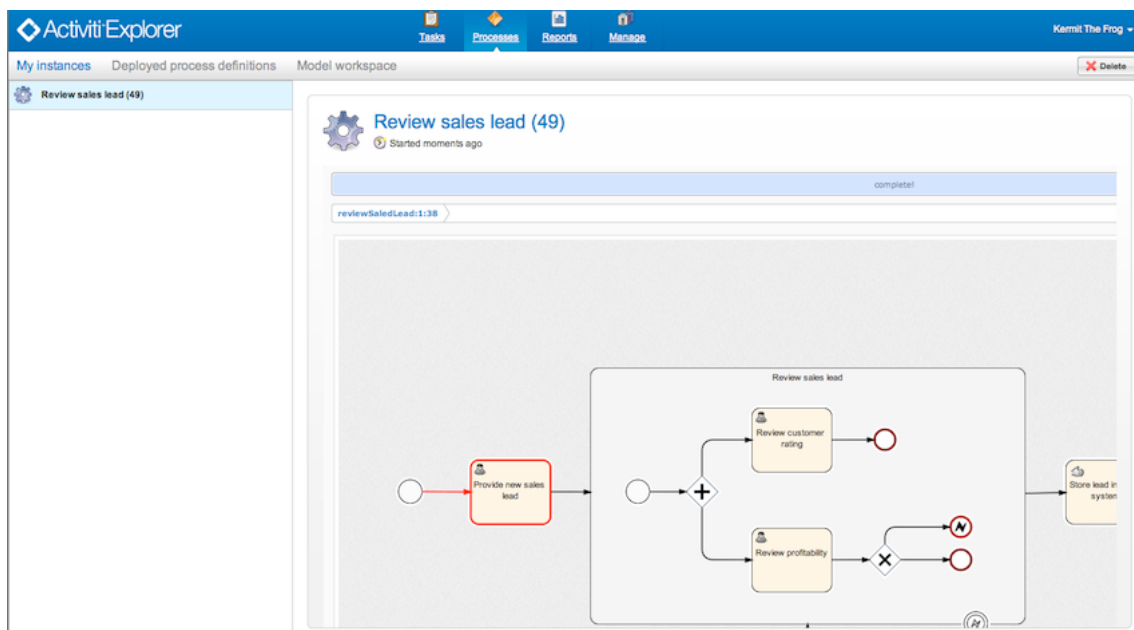
13.4. 启动流程实例 Start process instances

Process definitions（流程定义）页签可以查看Activiti引擎中部署的所有流程定义。可以使用右上角的按钮启动新的流程实例。如果流程定义了启动表单，则会在启动流程实例前显示该表单。



13.5. 我的实例 My instances

My instances页签，展示你当前有未完成用户任务的所有流程实例。同时也显示该流程实例的当前活动，以及存储的流程变量。



13.6. 管理 Administration

管理功能只有在登录用户是安全组 **admin** 的成员时才可用。当点击 **Manage** 图标时，可用下列页签：

- **Database:** 展示数据库内容。当部署流程或排错时十分有用。

DatabaseDeploymentsJobsUsersGroups

ACT_HI_DETAIL (8)

ACT_HI_PROCINST (5)

ACT_HI_TASKINST (12)

ACT_ID_GROUP (8)

ACT_ID_INFO (6)

ACT_ID_MEMBERSHIP (12)

ACT_ID_USER (3)

ACT_RE_DEPLOYMENT (2)

ACT_RE_PROCDEF (6)

ACT_RU_EXECUTION (8)

ACT_RU_IDENTITYLINK (2)

ACT_RU_JOB (1)

ACT_RU_TASK (8)

ACT_RU_VARIABLE (11)

ACT_RU_VARIABLE

ID_	REV_	TYPE_	NAME_	EXECUTION_ID_	PROC_INST_ID_
145	1	string	initiator	144	144
149	1	string	initiator	148	148
155	1	string	customerName	148	148
156	1	null	potentialProfit	148	148
157	1	string	details	148	148
169	1	string	employeeName	168	168
173	1	long	numberOfDays	168	168

- **Deployments:** 展示引擎当前的部署，并查看部署的内容（流程定义，图片，业务规则，等等）。

DatabaseDeploymentsJobsUsersGroups

activiti-engine-examples.bar

myProcess.bpmn20.xml

activiti-engine-examples.bar

Deployed 2 hours ago

Process Definitions

[Expense process](#)

[Fix system failure](#)

[Helpdesk process](#)

[Review sales lead](#)

[Vacation request](#)

Resources

[org/activiti/examples/adhoc/Expense_process.adhoc_Expense_process.png](#)

[org/activiti/examples/adhoc/Expense_process.bpmn20.xml](#)

[org/activiti/examples/bpmn/event/error/reviewSalesLead.bpmn20.xml](#)

点击 **deployment** 页签也可以上传新的部署。选择电脑中的一个业务存档或者一个 bpmn20.xml 文件，或者简单地拖放至特定区域，就可以部署新的业务流程。

Upload new deployment

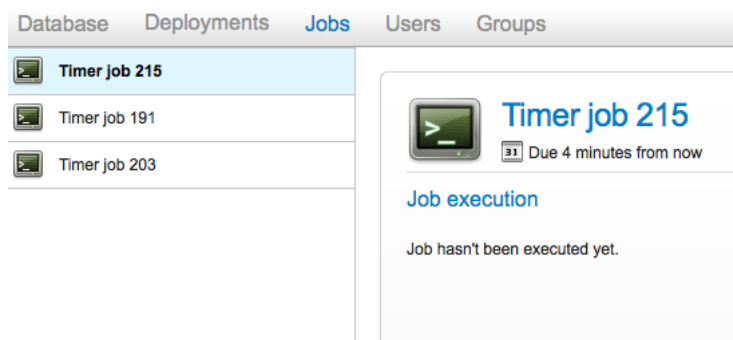
Select a file (.bar, .zip or .bpmn20.xml) or drop a file in the rectangle below.

Choose a file

or

Drop a file here

- **Jobs:** 在左侧展示当前的作业（定时器，等等），也可以手动执行它们（例如，在到时前触发定时器）。如果作业执行失败（例如邮件服务器无法连接），也会显示异常



- **Users and Groups:** 管理用户与组：创建、编辑与删除用户与组。将用户关联至组，以赋予更多权限，或使他们可以查看分派给特定组的任务。



13.7. 报告 Reporting

Activiti Explorer提供了一些报告的例子，也可以很容易地为系统添加新的报告。报告功能组织在'Reports'页签下。



重要：要使报告能工作，Explorer需要配置为不是none的历史级别。默认配置满足这个要求。

报告页签目前有两个子页签：

- **Generate reports:** 展示系统已有的所有报告类型的列表。可以运行来生成报告。
- **Saved reports:** 展示之前保存的所有报告的列表。请注意这些是个人保存的报告，不能查看其他人保存的报告。

用于创建报告中列表与图表的数据由流程生成。虽然初看有些奇怪，但使用流程生成报告数据有几个优点

- 流程可以直接访问Activiti引擎内部，并可以直接访问引擎使用的数据库。
- 作业执行器可以用于任何其他流程。这意味着可以异步生成流程，或者同步地执行一些步骤。也意味着可以使用定时器，例如，在特定时间点生成报告。
- 可以使用已有工具与已有概念创建新报告。并且，不需要新的概念、服务或应用。部署或上传新的报告与部署一个新流程是一样的。
- 可以使用BPMN 2.0结构。这意味着所有的东西，比如并行步骤，基于数据做分支选择，或者甚至是生成过程中请求用户输入都可以使用。

用于生成报告数据的流程定义需要是'activiti-report'类型的，这样才能在Explorer的已有报告列表中看到。“报告流程”可以简单，也可以任意复杂。唯一的要求，是流程要生成一个叫做reportData的变量，这个变量必须是一个表示JSON对象的字节数组。该变量存储在Activiti的历史表中（因此要求引擎必须启用历史），用于后续保存报告时获取。

13.7.1. 报告数据JSON（Report data JSON）

报告流程必须生成一个`reportData`变量，这是一个代表了需要显示给用户的数据的JSON，要像下面这样：

```
1 {
2   "title": "My Report",
3   "datasets": [
4     {
5       "type": "lineChart",
6       "description": "My first chart",
7       "xaxis": "Year"
8       "yaxis": "Total sales"
9       "data":
10      {
11        "2010": 50,
12        "2011": 33,
13        "2012": 17,
14        "2013": 87,
15      }
16    }
17  ]
18 }
```

这个JSON将在Explorer运行时获取，并将用于生成图表或列表。JSON中的元素是：

- **title:** 整个报告的总标题
- **datasets:** 与报告中各图表与列表对应的数据集的数组。
- **type:** 每个数据集都有一个类型。这个类型将用于决定如何渲染数据。目前支持的值有：**pieChart**、**lineChart**、**barChart**与**list**。
- **description:** 每个图表都可选一个描述，将显示在报告中。
- **x与yaxis:** 只对**lineChart**类型可用。描述图表坐标轴名字的可选参数。
- **data:** 实际的数据，是一个带有键值对元素的JSON对象。

13.7.2. 实例流程 Example process

下面的例子展示了一个‘process instance overview（流程实例总览）’报告。流程本身十分简单，只有一个使用JavaScript生成JSON数据集的脚本任务（除了启动与结束）。尽管Explorer中所有的例子都使用脚本，但也完全可以使用Java服务任务。运行流程的结果就是包含数据的`reportData`变量。

重要提示：下面的例子只能在JDK 7+使用。原因是旧JDK版本中的JavaScript引擎（*Rhino*）不够先进，不能使用类似下面使用的结构撰写脚本。之后有一个兼容JDK 6+的例子。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn"
4   xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
5   xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema"
6   expressionLanguage="http://www.w3.org/1999/XPath"
7   targetNamespace="activiti-report">
8
9   <process id="process-instance-overview-report" name="Process Instance Overview" isExecutable="true">
10
11     <startEvent id="startevent1" name="Start" />
12     <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="generateDataset" />
13
14     <scriptTask id="generateDataset" name="Execute script" scriptFormat="JavaScript"
15       activiti:autoStoreVariables="false">
16       <script><![CDATA[
17
18         importPackage(java.sql);
19         importPackage(java.lang);
20         importPackage(org.activiti.explorer.reporting);
21
22         var result =
23 ReportingUtil.executeSelectSqlQuery("SELECT PD.NAME_, PD.VERSION_ ,
24 count(*) FROM ACT_HI_PROCIINST PI inner join ACT_RE_PROCDEF PD on
25 PI.PROC_DEF_ID_ = PD.ID_ group by PROC_DEF_ID_");
26
27         var reportData = {};
28         reportData.datasets = [];
29
30         var dataset = {};
31         dataset.type = "pieChart";
32         dataset.description = "Process instance overview (" + new java.util.Date() + ")";
33         dataset.data = {};
34
35         while (result.next()) { // process results one row at a time
36           var name = result.getString(1);
37           var version = result.getLong(2);
38           var count = result.getLong(3);
39           dataset.data[name + " (v" + version + ")"] = count;
40         }
41       ]]>
42     </script>
43   </process>
44 </definitions>
```



```

41         reportData.datasets.push(dataset);
42
43         execution.setVariable("reportData", new java.lang.String(JSON.stringify(reportData)).getBytes("UTF-8"));
44     ]]></script>
45 </scriptTask>
46 <sequenceFlow id="flow3" sourceRef="generateDataset" targetRef="theEnd" />
47
48 <endEvent id="theEnd" />
49
50 </process>
51
52 </definitions>

```

除了流程XML顶端的标准的XML行，主要的区别是`targetNamespace`设置为`activiti-report`，为部署的流程定义添加了同名的类型。

脚本的前几行是为了避免重复写包名而进行的引入。要关注的第一行是使用`ReportingUtil`查询`Activiti`数据库，其结果是生成了一个普通的`JDBC ResultSet`。在查询下面的几行，`JavaScript`功能轻松地创建了要用的JSON。这个JSON按照[要求](#)生成。

脚本的最后一行看起来有些奇怪。首先是使用`JavaScript`函数`JSON.stringify()`将JSON对象转换为字符串，然后将这个字符串保存为一个字节数组变量。原因是技术性的：字节数组没有大小限制，而字符串有。这就是为什么`JavaScript`字符串必须要转换为`Java`字符串，因为这样就可以转换为字节形式。

兼容JDK 6（与更高）的相同流程看起来有点区别。不能使用原生的JSON功能，因此需要提供一些辅助类（`ReportData`与`Dataset`）：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn"
4   xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
5   xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema"
6   expressionLanguage="http://www.w3.org/1999/XPath"
7   targetNamespace="activiti-report">
8
9   <process id="process-instance-overview-report" name="Process Instance Overview" isExecutable="true">
10
11     <startEvent id="startevent1" name="Start" />
12     <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="generateDataset" />
13
14     <scriptTask id="generateDataset" name="Execute script" scriptFormat="js" activiti:autoStoreVariables="false">
15       <script><![CDATA[
16
17         importPackage(java.sql);
18         importPackage(java.lang);
19         importPackage(org.activiti.explorer.reporting);
20
21         var result =
22 ReportingUtil.executeSelectSqlQuery("SELECT PD.NAME_, PD.VERSION_ ,
23 count(*) FROM ACT_HI_PROINST PI inner join ACT_RE_PROCDEF PD on
24 PI.PROC_DEF_ID_ = PD.ID_ group by PROC_DEF_ID_");
25
26
27         var reportData = new ReportData();
28         var dataset = reportData.newDataset();
29         dataset.type = "pieChart";
30         dataset.description = "Process instance overview (" + new java.util.Date() + ")"
31
32
33         while (result.next()) { // process results one row at a time
34           var name = result.getString(1);
35           var version = result.getLong(2);
36           var count = result.getLong(3);
37           dataset.add(name + " (v" + version + ")", count);
38         }
39
40         execution.setVariable("reportData", reportData.toBytes());
41
42       ]]></script>
43     </scriptTask>
44     <sequenceFlow id="flow3" sourceRef="generateDataset" targetRef="theEnd" />
45
46     <endEvent id="theEnd" />
47
48   </process>
49
50 </definitions>

```

13.7.3. 报告启动表单 Report start forms

报告通过普通流程生成，因此可以使用普通的表单功能。简单的为启动事件添加启动表单，这样`Explorer`就将在生成流程前为用户显示表单。

```

<startEvent id="startevent1" name="Start">

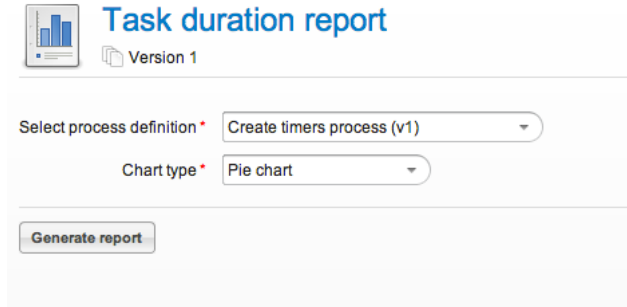
```

```

1  <extensionElements>
2    <activiti:formProperty id="processDefinition" name="Select process definition" type="processDefinition" required="true"
3  />
4    <activiti:formProperty id="chartType" name="Chart type" type="enum" required="true">
5      <activiti:value id="pieChart" name="Pie chart" />
6      <activiti:value id="barChart" name="Bar chart" />
7    </activiti:formProperty>
8  </extensionElements>
9  </startEvent>

```

为用户渲染的是一个典型的表单：



表单的参数将在流程启动时提交，与普通执行变量一样，可以在生成数据的脚本中使用：

```

1  var processDefinition = execution.getVariable("processDefinition");

```

13.7.4. 示例流程 Example processes

默认情况下，Explorer包含四个示例报告：

- **Employee productivity**（雇员生产力）：这个报告演示了折线图的使用，并使用了启动表单。这个报告中使用的脚本也比其他例子中的复杂，因为获取的数据，在存储至报告数据前，会由脚本进行处理。
- **Helpdesk - firstline vs escalated**（帮助中心——一线对比升级）：展示了饼图的使用，并结合了两个不同数据库查询的结果。
- **Process instance overview**（流程实例总览）：使用多个数据集的示例报告。这个报告包含有一个饼图，以及相同数据的列表视图，这样展示了如何使用多个数据集生成一个带有多个图表的页面。
- **Task duration**（任务持续时间）：另一个使用了启动表单的例子，并使用了相应的数据动态建立SQL查询。

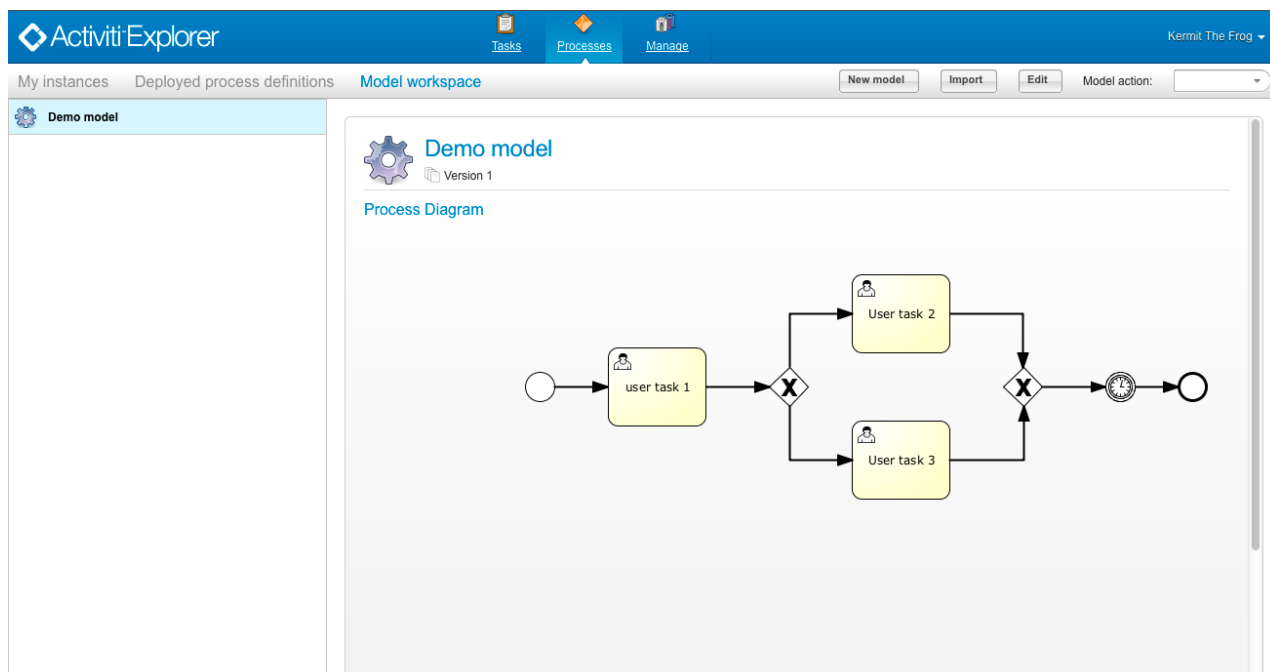
13.8. 改变数据库 Changing the database

要修改Explorer在演示配置中使用的数据库，需要修改 `apps/apache-tomcat-6.x/webapps/activiti-explorer/WEB-INF/classes/db.properties` 配置文件。并且，在classpath中放入合适的数据库驱动（Tomcat共享库，或放在 `apps/apache-tomcat-6.x/webapps/activiti-explorer/WEB-INF/lib/` 下）。

14. Activiti Modeler

Activiti Modeler是一个BPMN web建模器组件，内置在Activiti Explorer web应用中。Modeler是 [Signavio核心组件](#) 的分支项目。Activiti Modeler从5.17.0版本起基于Angular JS（之前是基于Ext-JS的应用）。Activiti Modeler的Angular JS部分基于LGPL协议发布。与之前版本的Activiti Modeler（Signavio核心组件）的主要区别，是新的Modeler作为Activiti项目的一部分维护与开发。Activiti Modeler的目标，是支持所有BPMN元素与Activiti支持的扩展。

使用默认配置运行Activiti Explorer时，在模型工作空间内将有一个示例流程。



14.1. 编辑模型 Model editing

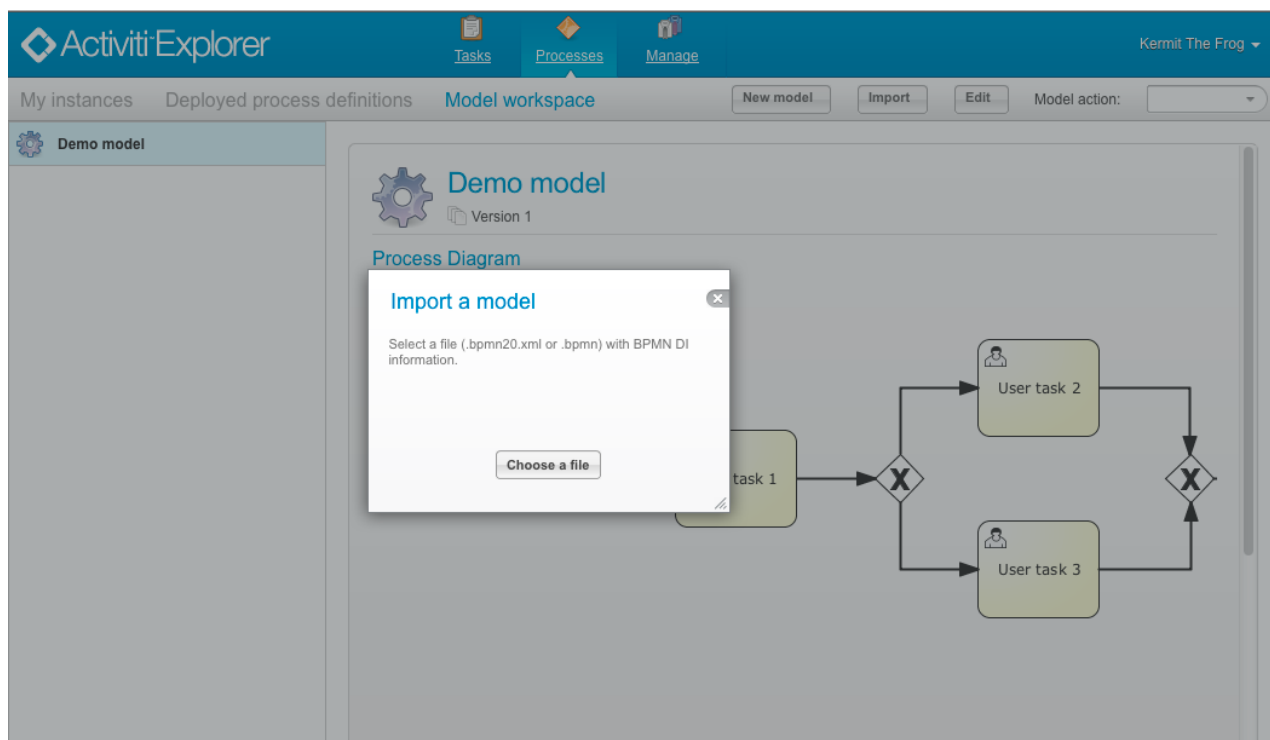
在模型工作空间点击**edit**按钮时，会在建模器中打开模型。屏幕的左侧是BPMN元素与Activiti扩展的画板。在需要时可以将新元素拖放至画布上。在屏幕底部可以填写选中元素的属性。在示例截图中选中了一个用户任务，可以填写用户任务属性，例如办理人，表单参数与到期日期。要返回 Activiti Explorer，可以点击屏幕右上角的关闭按钮。

The screenshot shows the Activiti Modeler web application. The top bar includes the 'Alfresco Activiti' logo and a toolbar with various editing tools. The left sidebar contains a tree view of BPMN elements: Start Events, Activities, Structural, Gateways, Boundary Events, Intermediate Catching Events, Intermediate Throwing Events, End Events, Swimlanes, and Artifacts. The main canvas displays the same process diagram as in the previous screenshot, but with 'user task 1' selected and highlighted with a dashed border. The bottom panel shows the properties for 'user task 1'.

▼ user task 1			
Id :	No value	Name :	user task 1
Documentation :	No value	Asynchronous :	<input type="checkbox"/>
Exclusive :	<input type="checkbox"/>	Execution listeners :	No execution listeners configured
Multi-Instance type :	None	Cardinality (Multi-instance) :	No value
Collection (Multi-instance) :	No value	Element variable (Multi-instance) :	No value
Completion condition (Multi-instance) :	No value	Is for compensation :	<input type="checkbox"/>

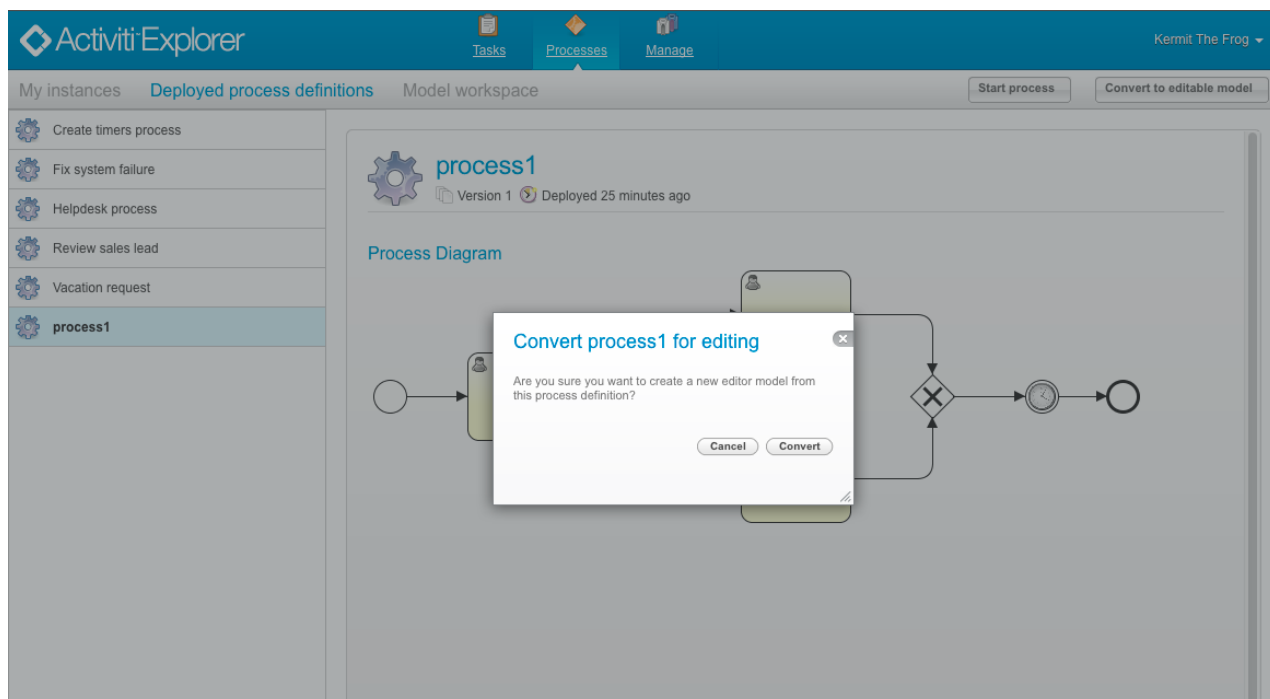
14.2. 导入现有模型 Importing existing models

也可以将现有模型导入模型工作空间，以在Activiti Modeler中编辑它们。点击**import**按钮，选择一个.bpmn或者.bpmn20.xml文件。请注意这个BPMN XML文件需要包含BPMN DI信息。



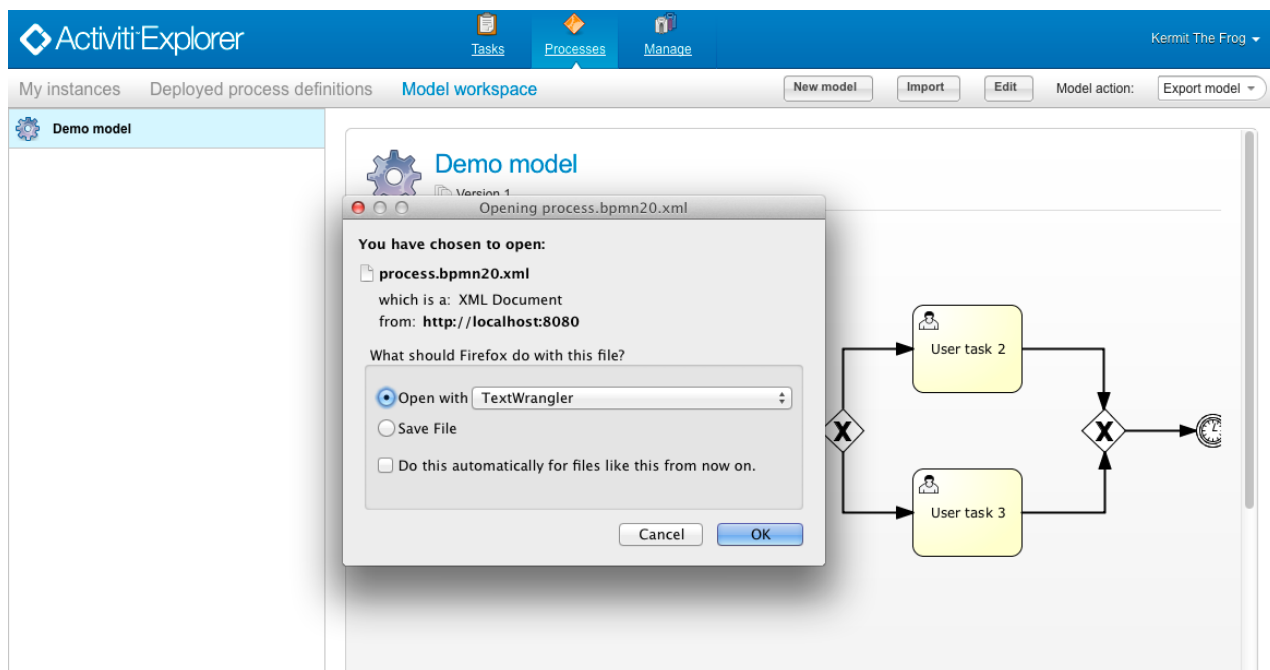
14.3. 将已部署定义转换为可编辑模型 Convert deployed definitions to a editable model

已部署的流程定义可以转换为能够使用Activiti Modeler编辑的模型。请注意该流程定义需要包含BPMN DI信息。



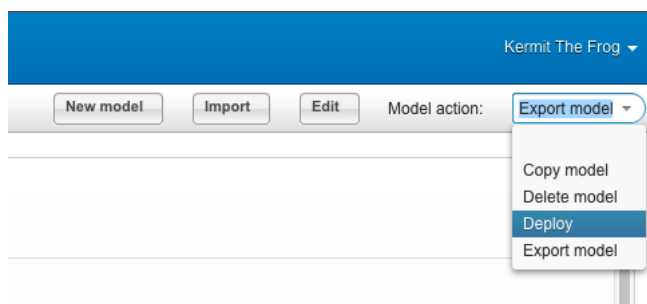
14.4. 将模型导出为BPMN XML (Export model to BPMN XML)

模型工作空间中的模型可以导出为BPMN XML文件。在模型动作选项框中选择导出选项。



14.5. 将模型部署至Activiti引擎 Deploy model to the Activiti Engine

当模型包含了运行所需的所有参数，就可以部署至Activiti引擎。在模型动作选项框中选择部署选项。



15. REST API

15.1. Activiti REST一般原则 General Activiti REST principles

15.1.1. 安装与认证 Installation and Authentication

Activiti在引擎中包含了REST API，可以通过在servlet容器如Apache Tomcat中，部署activiti-rest.war文件来安装。但是也可以在其他web应用中使用，只要在你的应用中包含这些servlet与其映射，并在classpath中添加所有activiti-rest依赖即可。

默认情况下Activiti引擎连接至一个H2内存数据库。可以修改WEB-INF/classes文件夹下的db.properties文件中的数据库设置。REST API使用JSON格式 (<http://www.json.org>)，基于Spring MVC (<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html>) 构建。

默认情况下，所有REST资源都需要有一个有效的的Activiti已认证用户，使用基础HTTP访问认证，因此在请求时，可以在HTTP头添加 **Authorization: Basic ...==**，也可以请求url中包含用户名与密码（例如 <http://username:password@localhost...>）。

建议使用基础认证时，同时使用HTTPS。

15.1.2. 配置 Configuration

Activiti REST web应用使用Spring Java Configuration来启动Activiti引擎、定义基础认证安全使用Spring security，以及为特定的变量处理定义变量转换。可以修改WEB-INF/classes目录下的engine.properties文件，定义少量参数。如果需要高级配置选项，可以在activiti-custom-context.xml文件中覆盖默认的Spring bean，这个文件也在WEB-INF/classes目录下。该文件中已经以注释形式提供了示例配置。也可以在这里通过定义一个新的命名为 restResponsefactory的Spring bean，覆盖默认的RestResponseFactory，并使用自定义实现类。

15.1.3. 在Tomcat中使用 Usage in Tomcat

由于Tomcat中的默认安全参数，默认不能使用已转义斜线符（%2F与%5C）（返回400结果）。这可能会影响部署资源与其数据URL，因为URL可能隐含已转义斜线符。

当发现非预期的400结果时，设置下列系统参数 `-Dorg.apache.tomcat.util.buf.UDecoder.ALLOW_ENCODED_SLASH=true`。

最佳实践是（post/put JSON时），在下面描述的HTTP请求中，永远将Accept与Content-Type头设置为application/json。

15.1.4. 方法与返回码 Methods and return-codes

Table 2. HTTP方法与相应操作

方法	描述
<code>GET</code>	获取单个资源，或获取一组资源。
<code>POST</code>	创建一个新资源。在查询结构太复杂，不能放入 <code>GET</code> 请求的查询URL中时，也用于执行资源查询。
<code>PUT</code>	更新一个已有资源的参数。也用于在已有资源上的调用动作。
<code>DELETE</code>	删除一个已有资源。

Table 3. HTTP方法响应码

响应	描述
<code>200 - OK</code>	操作成功，返回响应（ <code>GET</code> 与 <code>PUT</code> 请求）。
<code>201 - 已创建</code>	操作成功，已经创建了实体，并在响应体中返回（ <code>POST</code> 请求）。
<code>204 - 无内容</code>	操作成功，已经删除了实体，因此没有返回的响应体（ <code>DELETE</code> 请求）。
<code>401 - 未认证</code>	操作失败。操作要求设置认证头。如果请求中有认证头，则提供的鉴证并不合法，或者用户未被授权进行该操作。
<code>403 - 禁止</code>	操作被禁止，且不应重试。这不是鉴证或授权的问题，而是说明不允许该操作。例如：删除一个运行中流程的任务是且永远是不允许的，无论该用户或流程/任务的状态。
<code>404 - 未找到</code>	操作失败。请求的资源未找到。
<code>405 - 不允许的方法</code>	操作失败。使用的方法不能用于该资源。例如，更新（ <code>PUT</code> ）部署资源将导致 <code>405</code> 状态。
<code>409 - 冲突</code>	操作失败。该操作导致更新一个已被其他操作更新的资源，因此本更新不再有效。也可以表明正在为一个集合创建一个资源，但该集合中已经使用了该标识符。
<code>415 - 不支持的媒体类型</code>	操作失败。请求体包含了不支持的媒体类型。也会发生在请求体JSON中包含了未知的属性或值，但没有正确的格式/类型来接受的情况下。
<code>500 - 服务器内部错误</code>	操作失败。执行操作时发生了未知异常。响应体中包含了错误的细节。

HTTP响应的media-type总是`application/json`，除非请求的是二进制内容（例如部署资源数据）。这时将使用内容的media-type。

15.1.5. 错误响应体 Error response body

当发生错误时（客户端与服务器端都可能，4XX及5XX状态码），响应体会包含一个描述了发生的错误的对象。任务未找到时的404状态的例子：

```
1 {
2   "statusCode" : 404,
3   "errorMessage" : "Could not find a task with id '444'."
4 }
```

15.1.6. 请求参数 Request parameters

URL片段 URL fragments

作为url的一部分的参数（例如，`http://host/actviti-rest/service/repository/deployments/{deploymentId}`中的`deploymentId`参数），如果包含特殊字符，则需要进行合适的转义（参见URL编码或百分号编码）。大多数框架都内建了这个功能，但要记得考虑它。特别是对可能包含斜线符的段落（例如部署资源），就是必须的。

Rest URL查询参数 Rest URL query parameters

作为查询字符串添加在URL中的参数（例如<http://host/activiti-rest/service/deployments?name=Deployment>中的name参数）可以使用下列类型，也会在相应的REST-API文档中提到：

类型	默认值	格式	描述
Table 4. URL查询参数类型			
类型	格式		
String	纯文本参数。可以包含任何URL允许的合法字符。对于XXXLike参数，字符串可能会包含通配符%（需要进行URL编码）。可以进行like搜索，例如，'Tas%'将匹配所有以'Tas'开头的值。		
Integer	整形参数。只能包含数字型非十进制值（原文如此，下同），在-2.147.483.648至2.147.483.647之间。		
Long	长整形参数。只能包含数字型非十进制值，在-9.223.372.036.854.775.808至9.223.372.036.854.775.807之间。		
Boolean	boolean型参数。可以为true或false。任何其他值都会导致'405 - 错误请求'响应。		
Date	日期型参数。使用ISO-8601日期格式（参考wikipedia中的ISO-8601），使用时间与日期组分（例如2013-04-03T23:45Z）。		

JSON体参数 JSON body parameters

Table 5. JSON参数类型

类型	格式
String	纯文本参数。对于XXXLike参数，字符串可能会包含通配符%。可以进行like搜索。例如，'Tas%'将匹配所有以'Tas'开头的值。
Integer	整形参数，使用JSON数字。只能包含数字型非十进制值（原文如此，下同），在-2.147.483.648至2.147.483.647之间。
Long	长整形参数，使用JSON数字。只能包含数字型非十进制值，在-9.223.372.036.854.775.808至9.223.372.036.854.775.807之间。
Date	日期型参数，使用JSON文本。使用ISO-8601日期格式（参考wikipedia中的ISO-8601），使用时间与日期组分（例如2013-04-03T23:45Z）。

分页与排序 Paging and sorting

分页与排序参数可以作为查询字符串加入URL中（例如<http://host/activiti-rest/service/deployments?sort=name>中的name参数）。

Table 6. 查询变量参数

参数	默认值	描述
sort	各查询实现不同	排序键的名字，在各查询实现中默认值与可用值都不同。
order	asc	排序顺序，可以是'asc'（顺序）或'desc'（逆序）。
start	0	对结果分页的参数。默认结果从0开始。
size	10	对结果分页的参数。默认大小为10。

JSON查询变量格式

```
1 {
2   "name" : "variableName",
3   "value" : "variableValue",
4   "operation" : "equals",
}
```



```
5 | "type" : "string"
6 | }
```

参数名	必填	描述
-----	----	----

Table 7. JSON查询变量参数

参数	必填	描述
name	否	包含在查询中的变量名。在有些使用'equals'的查询中可以为空，查询任意变量名为给定值的资源。
value	是	包含在查询中的变量值，需要使用给定类型的正确格式。
operator	是	查询使用的操作，可以为下列值： <code>equals</code> ， <code>notEquals</code> ， <code>equalsIgnoreCase</code> ， <code>notEqualsIgnoreCase</code> ， <code>lessThan</code> ， <code>greaterThan</code> ， <code>lessThanOrEquals</code> ， <code>greaterThanOrEquals</code> 与 <code>like</code> 。
type	否	所用变量的类型。当省略时，会从 <code>value</code> 参数推理类型。任何JSON文本值都认为是 <code>string</code> 类型，JSON boolean值认为是 <code>boolean</code> 类型，JSON数字认为是 <code>long</code> 或 <code>integer</code> ，取决于数字的大小。建议在有疑惑时明确指定类型。其他支持的类型列在下面。

Table 8. 默认查询JSON类型

类型名	描述
string	值处理转换为 <code>java.lang.String</code> 。
short	值处理转换为 <code>java.lang.Integer</code> 。
integer	值处理转换为 <code>java.lang.Integer</code> 。
long	值处理转换为 <code>java.lang.Long</code> 。
double	值处理转换为 <code>java.lang.Double</code> 。
boolean	值处理转换为 <code>java.lang.Boolean</code> 。
date	值处理转换为 <code>java.util.Date</code> 。JSON字符串将使用ISO-8601日期格式转换。

变量表示 Variable representation

当使用变量时（执行/流程与任务），读取与写入时REST-api都使用一些通用原则与JSON格式。变量的JSON表示像是这样：

```
1 | {
2 |   "name" : "variableName",
3 |   "value" : "variableValue",
4 |   "valueUrl" : "http://...",
5 |   "scope" : "local",
6 |   "type" : "string"
7 | }
```

Table 9. 变量的JSON属性

参数	必填	描述
name	是	变量名。
value	否	变量的值。当写入变量且省略了 <code>value</code> 时，会使用 <code>null</code> 作为value。

参数名 valueUrl 参数	必填 否	描述 值
		描述 当读取 <code>binary</code> 或 <code>serializable</code> 类型的变量时，这个属性将指向可用于获取原始二进制数据的URL。
scope	否	变量的范围。如果值为 <code>local</code> ，则变量明确定义在其请求的资源上。如果值为 <code>global</code> ，则变量定义在其父上（或者父树中的任意父）。当写入变量且省略了 <code>scope</code> 时，使用 <code>global</code> 。
type	否	变量的类型。查看下面的表格了解类型的更多信息。当写入变量且省略了这个值时，将使用请求的原始JSON属性类型推断，限制在 <code>string</code> 、 <code>double</code> 、 <code>integer</code> 与 <code>boolean</code> 中。建议总是包含类型，以确保不会错误推断类型。

Table 10. 变量类型

类型名	描述
string	值按照 <code>java.lang.String</code> 处理。写入变量时使用原始JSON文本。
integer	值按照 <code>java.lang.Integer</code> 处理。按约定写入变量时使用JSON数字，失败则退回JSON文本。
short	值按照 <code>java.lang.Short</code> 处理。按约定写入变量时使用JSON数字，失败则退回JSON文本。
long	值按照 <code>java.lang.Long</code> 处理。按约定写入变量时使用JSON数字，失败则退回JSON文本。
double	值按照 <code>java.lang.Double</code> 处理。按约定写入变量时使用JSON数字，失败则退回JSON文本。
boolean	值按照 <code>java.lang.Boolean</code> 处理。按约定写入变量时使用JSON <code>boolean</code> 。
date	值按照 <code>java.util.Date</code> 处理。写入变量时将转换为ISO-8601日期格式。
binary	二进制变量，按照字节数组处理。 <code>value</code> 属性为 <code>null</code> ， <code>valueUrl</code> 包含指向原始二进制流的URL。
serializable	代表序列化的Java对象。与 <code>binary</code> 类型一样， <code>value</code> 属性为 <code>null</code> ， <code>valueUrl</code> 包含指向原始二进制流的URL。所有可序列化的变量（不是上述任意类型的）将被暴露为这个类型的变量。

可以使用自定义JSON表示，以支持额外的变量类型（既可以是简单值，也可以是复杂/嵌套的JSON对象）。通过扩展 `org.activiti.rest.service.api.RestResponseFactory` 的 `initializeVariableConverters()` 方法，可以添加额外的 `org.activiti.rest.service.api.engine.variable.RestVariableConverter` 类，来将你的POJO转换为适合通过REST传输的格式，以及将REST值转换为POJO。实际转换JSON使用Jackson。

15.2. 部署 Deployment

使用 `tomcat` 时，请阅读在 [Tomcat](#) 中使用。

15.2.1. 部署的列表 List of Deployments

GET repository/deployments

Table 11. URL查询参数

--

参数	必填	值	描述
name	否	String	只返回给定名字的部署。
nameLike	否	String	只返回名字like给定名字的部署。
category	否	String	只返回给定分类的部署。
categoryNotEquals	否	String	只返回不是给定分类的部署。
tenantId	否	String	只返回给定tenantId的部署。
tenantIdLike	否	String	只返回tenantId like给定值的部署。
withoutTenantId	否	Boolean	如果值为true，则只返回没有设置tenantId的部署。如果值为false，则忽略withoutTenantId参数。
sort	否	id（默认），name，deploytime或tenantId	用于排序的参数，与'order'一起使用。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 12. REST返回码

返回码	描述
200	代表请求成功。

成功响应体：

```
1 {
2   "data": [
3     {
4       "id": "10",
5       "name": "activiti-examples.bar",
6       "deploymentTime": "2010-10-13T14:54:26.750+02:00",
7       "category": "examples",
8       "url": "http://localhost:8081/service/repository/deployments/10",
9       "tenantId": null
10    }
11  ],
12  "total": 1,
13  "start": 0,
14  "sort": "id",
15  "order": "asc",
16  "size": 1
17 }
```

15.2.2. 获取一个部署 Get a deployment

```
GET repository/deployments/{deploymentId}
```

Table 13. 获取一个部署 - URL参数

参数	必填	值	描述
deploymentId	是	String	要获取的部署的id。

Table 14. 获取一个部署 - 响应码

响应码	描述
200	代表已找到并返回部署。
404	代表未找到请求的部署。

成功响应体：

```
1 {
2   "id": "10",
3   "name": "activiti-examples.bar",
4   "deploymentTime": "2010-10-13T14:54:26.750+02:00",
5   "category": "examples",
6   "url": "http://localhost:8081/service/repository/deployments/10",
7   "tenantId" : null
8 }
```

15.2.3. 创建一个新部署 Create a new deployment

```
POST repository/deployments
```

请求体：

请求体需要包含 *multipart/form-data* 类型的数据。请求中需要只有一个文件，多余的文件将被忽略。部署名是传入的文件字段的名字。如果要在一个部署中部署多个资源，需要将资源压缩为zip文件，并确保文件名以 `.bar` 或 `.zip` 结尾。

可以在请求体中传递名为 `tenantId` 的额外参数（表单字段）。这个字段的值将指定部署所在的租户（tenant）的id。

Table 15. 创建一个新部署 - 响应码

响应码	描述
201	代表成功创建部署
400	代表请求体中没有内容，或部署不支持content的mime-type。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "id": "10",
3   "name": "activiti-examples.bar",
4   "deploymentTime": "2010-10-13T14:54:26.750+02:00",
5   "category": null,
6   "url": "http://localhost:8081/service/repository/deployments/10",
7   "tenantId" : "myTenant"
8 }
```

15.2.4. 删除一个部署 Delete a deployment

```
DELETE repository/deployments/{deploymentId}
```

Table 16. 删除一个部署 - URL 参数

参数	必填	值	描述
deploymentId	是	String	要删除的部署的id。

Table 17. 删除一个部署 - 响应码

响应码	描述
204	代表已找到并删除了部署。响应体设置为空。
404	代表未找到请求的部署。

15.2.5. 列表一个部署中的资源 List resources in a deployment

```
GET repository/deployments/{deploymentId}/resources
```

Table 18. 列表一个部署中的资源 - URL 参数

参数	必填	值	描述

响应码	必填	描述
deploymentId	是	String 要获取资源的部署的id。

Table 19. 列表一个部署中的资源 - 响应码

响应码	描述
200	代表已找到部署，并已返回资源的列表。
404	代表未找到请求的部署。

成功响应体：

```

1  [
2    {
3      "id": "diagrams/my-process.bpmn20.xml",
4      "url": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resources/diagrams%2Fmy-
5      process.bpmn20.xml",
6      "dataUrl": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resourcedata/diagrams%2Fmy-
7      process.bpmn20.xml",
8      "mediaType": "text/xml",
9      "type": "processDefinition"
10   },
11   {
12     "id": "image.png",
13     "url": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resources/image.png",
14     "dataUrl": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resourcedata/image.png",
15     "mediaType": "image/png",
16     "type": "resource"
17   }
18 ]

```

- **mediaType**: 包含了资源的媒体类型。使用（可插入的）**MediaTypeResolver** 解析，默认包含有限数量的媒体类型映射。
- **type**: 资源的类型，可用值为：
 - **resource**: 原始资源。
 - **processDefinition**: 包含一个或多个流程定义的资源。通过部署器挑选。
 - **processImage**: 代表流程定义的图形化输出的资源。

结果JSON中的**dataUrl**参数包含了获取该二进制资源的实际URL。

15.2.6. 获取一个部署资源 Get a deployment resource

```
GET repository/deployments/{deploymentId}/resources/{resourceId}
```

Table 20. 获取一个部署资源 - URL 参数

参数	必填	值	描述
deploymentId	是	String	请求的资源所在的部署的id。
resourceId	是	String	要获取的资源的id。请确保如果包含斜线符，需要对 resourceId 进行URL编码。例如，使用' diagrams%2Fmy-process.bpmn20.xml '代替' diagrams/Fmy-process.bpmn20.xml '。

Table 21. 获取一个部署资源 - 响应码

响应码	描述
200	代表已找到部署与资源，并已返回资源。
404	代表未找到请求的部署，或者该部署中没有给定id的资源。状态描述包含了额外信息。

成功响应体：

```
1 {
2   "id": "diagrams/my-process.bpmn20.xml",
3   "url": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resources/diagrams%2Fmy-
4 process.bpmn20.xml",
5   "dataUrl": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resourcedata/diagrams%2Fmy-
6 process.bpmn20.xml",
7   "mediaType": "text/xml",
   "type": "processDefinition"
}
```

- `mediaType`：包含了资源的媒体类型。使用（可插入的）`MediaTypeResolver`解析，默认包含有限数量的媒体类型映射。
- `type`：资源的类型，可用值为：
 - `resource`：原始资源。
 - `processDefinition`：包含一个或多个流程定义的资源。通过部署器挑选。
 - `processImage`：代表流程定义的图形化输出的资源。

15.2.7. 获取一个部署资源的内容 Get a deployment resource content

```
GET repository/deployments/{deploymentId}/resourcedata/{resourceId}
```

Table 22. 获取一个部署资源的内容 - URL 参数

参数	必填	值	描述
deploymentId	是	String	请求的资源所在的部署的id。
resourceId	是	String	要获取的资源的id。请确保如果包含斜线符，需要对 resourceId 进行URL编码。例如，使用' diagrams%2Fmy-process.bpmn20.xml '代替' diagrams/Fmy-process.bpmn20.xml '。

Table 23. 获取一个部署资源的内容 - 响应码

响应码	描述
200	代表已找到部署与资源，并已返回资源。
404	代表未找到请求的部署，或者该部署中没有给定id的资源。状态描述包含了额外信息。

成功响应体：

响应体将包含所请求资源的二进制资源内容。响应的content-type与资源'mimeType'参数返回的类型相同。同时将设置content-disposition头，让浏览器可以下载文件而不是直接显示。

15.3. 流程定义 Process Definitions

15.3.1. 流程定义的列表 List of process definitions

```
GET repository/process-definitions
```

Table 24. 流程定义的列表 - URL 参数

参数	必填	值	描述
version	否	integer	只返回给定版本的流程定义。
name	否	String	只返回给定名字的流程定义。

参数 nameLike	必填 否	类型 String	描述 只返回名字like给定名字的流程定义。
key	否	String	只返回给定key的流程定义。
keyLike	否	String	只返回key like给定key的流程定义。
resourceName	否	String	只返回给定资源名的流程定义。
resourceNameLike	否	String	只返回资源名like给定资源名的流程定义。
category	否	String	只返回给定分类的流程定义
categoryLike	否	String	只返回分类名like给定名字的流程定义。
categoryNotEquals	否	String	只返回不是给定分类的流程定义。
deploymentId	否	String	只返回给定id的部署中的流程定义。
startableByUser	否	String	只返回给定用户可以启动的流程定义。
latest	否	Boolean	只返回流程定义的最新版本。 只能与'key'及'keyLike'参数一起使用，同时使用任何其它参数都将导致400响应。
suspended	否	Boolean	如果值为 <code>true</code> ，则只返回暂停的流程定义。如果值为 <code>false</code> ，则只返回活动的流程定义（未暂停的）。
sort	否	<i>name</i> （默认）， <i>id</i> , <i>key</i> , <i>category</i> , <i>deploymentId</i> 与 <i>version</i>	用于排序的参数，与'order'一起使用。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 25. 流程定义的列表 - 响应码

响应码	描述
200	代表请求成功，并已返回流程定义。
400	代表某个参数格式错误，或者'latest'与'key', 'keyLike'以外的其他参数一起使用。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "id" : "oneTaskProcess:1:4",
5       "url" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
6       "version" : 1,
7       "key" : "oneTaskProcess",
8       "category" : "Examples",
9       "suspended" : false,
10      "name" : "The One Task Process",
11      "description" : "This is a process for testing purposes",
12      "deploymentId" : "2",
13      "deploymentUrl" : "http://localhost:8081/repository/deployments/2",
```



```

14     "graphicalNotationDefined" : true,
15     "resource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.xml",
16     "diagramResource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.png",
17     "startFormDefined" : false
18   }
19 ],
20 "total": 1,
21 "start": 0,
22 "sort": "name",
23 "order": "asc",
24 "size": 1
25 }

```

- **graphicalNotationDefined**: 代表流程定义中包含有图形信息（BPMN DI）。
- **resource**: 包含实际部署的BPMN 2.0 XML。
- **diagramResource**: 包含流程的图形化表示。如果没有可用流程图则为null。

15.3.2. 获取一个流程定义

```
GET repository/process-definitions/{processDefinitionId}
```

Table 26. 获取一个流程定义 - URL 参数

参数	必填	值	描述
processDefinitionId	是	String	要获取的流程定义的id。

Table 27. 获取一个流程定义 - 响应码

响应码	描述
200	代表已找到并已返回流程定义。
404	代表未找到请求的流程定义。

成功响应体:

```

1  {
2    "id" : "oneTaskProcess:1:4",
3    "url" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
4    "version" : 1,
5    "key" : "oneTaskProcess",
6    "category" : "Examples",
7    "suspended" : false,
8    "name" : "The One Task Process",
9    "description" : "This is a process for testing purposes",
10   "deploymentId" : "2",
11   "deploymentUrl" : "http://localhost:8081/repository/deployments/2",
12   "graphicalNotationDefined" : true,
13   "resource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.xml",
14   "diagramResource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.png",
15   "startFormDefined" : false
16 }

```

- **graphicalNotationDefined**: 代表流程定义中包含有图形信息（BPMN DI）。
- **resource**: 包含实际部署的BPMN 2.0 XML。
- **diagramResource**: 包含流程的图形化表示。如果没有可用流程图则为null。

15.3.3. 更新一个流程定义的分类 Update category for a process definition

```
PUT repository/process-definitions/{processDefinitionId}
```

JSON体:

```

1  {
2    "category" : "updatedcategory"
3  }

```

Table 28. 更新一个流程定义的分类 - 响应码

响应码	描述
200	代表已修改流程的分类。
400	代表请求体中未定义分类。
404	代表未找到请求的流程定义。

成功响应体：参见 `repository/process-definitions/{processDefinitionId}` 的响应。

15.3.4. 获取一个流程定义资源的内容 Get a process definition resource content

```
GET repository/process-definitions/{processDefinitionId}/resourcedata
```

Table 29. 获取一个流程定义资源的内容 - URL 参数

参数	必填	值	描述
processDefinitionId	是	String	要获取资源的流程定义的id。

响应：

与 `GET repository/deployment/{deploymentId}/resourcedata/{resourceId}` 完全一样的响应码/响应体。

15.3.5. 获取一个流程定义的BPMN模型 Get a process definition BPMN model

```
GET repository/process-definitions/{processDefinitionId}/model
```

Table 30. 获取一个流程定义的BPMN模型 - URL 参数

参数	必填	值	描述
processDefinitionId	是	String	要获取模型的流程定义的id。

Table 31. 获取一个流程定义的BPMN模型 - 响应码

响应码	描述
200	代表已找到流程定义，并已返回模型。
404	代表未找到请求的流程定义。

响应体：响应体是一个代表了 `org.activiti.bpmn.model.BpmnModel` 的JSON，包含所有流程定义模型。

```
1 {
2   "processes": [
3     {
4       "id": "oneTaskProcess",
5       "xmlRowNumber": 7,
6       "xmlColumnNumber": 60,
7       "extensionElements": {
8       },
9     },
10    "name": "The One Task Process",
11    "executable": true,
12    "documentation": "One task process description",
13  ],
14 }
15 }
```

15.3.6. 暂停一个流程定义 Suspend a process definition

```
PUT repository/process-definitions/{processDefinitionId}
```

JSON体：

```
1 {
2   "action": "suspend",
3   "includeProcessInstances": "false",
4 }
```

```
4 | "date" : "2013-04-15T00:42:12Z"
5 | }
```

响应码	必填	描述	必填	描述
-----	----	----	----	----

Table 32. 暂停一个流程定义 - JSON体参数

参数	描述	必填
action	要进行的操作， <code>activate</code> 或 <code>suspend</code> 。	是
includeProcessInstances	是否同时暂停/激活该流程定义的运行中流程实例。如果省略，则流程实例保持原有状态。	否
date	要进行暂停/激活操作的日期（ISO-8601）。如果省略，则暂停/激活立刻生效。	否

Table 33. 暂停一个流程定义 - 响应码

响应码	描述
200	代表已暂停流程。
404	代表未找到请求的流程定义。
409	代表请求的流程定义之前已经暂停。

成功响应体：参见 `repository/process-definitions/{processDefinitionId}` 的响应

15.3.7. 激活一个流程定义 Activate a process definition

```
PUT repository/process-definitions/{processDefinitionId}
```

JSON体：

```
1 | {
2 |   "action" : "activate",
3 |   "includeProcessInstances" : "true",
4 |   "date" : "2013-04-15T00:42:12Z"
5 | }
```

参见暂停流程定义的JSON体参数。

Table 34. 激活一个流程定义 - 响应码

响应码	描述
200	代表已激活流程。
404	代表未找到请求的流程定义。
409	代表请求的流程定义之前已经激活。

成功响应体：参见 `repository/process-definitions/{processDefinitionId}` 的响应

15.3.8. 获取一个流程定义的所有候选启动者 Get all candidate starters for a process-definition

```
GET repository/process-definitions/{processDefinitionId}/identitylinks
```

Table 35. 获取一个流程定义的所有候选启动者 - URL 参数

参数	必填	值	描述
processDefinitionId	是	String	要获取身份关联的流程定义的id。

Table 36. 获取一个流程定义的所有候选启动者 - 响应码

--

响应码	描述
200	代表已找到流程定义，并已返回请求的身份关联。
404	代表未找到请求的流程定义。

成功响应体：

```
1  [
2    {
3      "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/groups/admin",
4      "user": null,
5      "group": "admin",
6      "type": "candidate"
7    },
8    {
9      "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
10     "user": "kermit",
11     "group": null,
12     "type": "candidate"
13   }
14 ]
```

15.3.9. 为一个流程定义添加一个候选启动者 Add a candidate starter to a process definition

POST repository/process-definitions/{processDefinitionId}/identitylinks

Table 37. 为一个流程定义添加一个候选启动者 - URL 参数

参数	必填	值	描述
processDefinitionId	是	String	流程定义的id。

请求体（用户）：

```
1  {
2    "user" : "kermit"
3  }
```

请求体（组）：

```
1  {
2    "groupId" : "sales"
3  }
```

Table 38. 为一个流程定义添加一个候选启动者 - 响应码

响应码	描述
201	代表已找到流程定义，并已添加身份关联。
404	代表未找到请求的流程定义。

成功响应体：

```
1  {
2    "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
3    "user": "kermit",
4    "group": null,
5    "type": "candidate"
6  }
```

15.3.10. 从一个流程定义中删除一个候选启动者 Delete a candidate starter from a process definition

DELETE repository/process-definitions/{processDefinitionId}/identitylinks/{family}/{identityId}

Table 39. 从一个流程定义中删除一个候选启动者 - URL 参数

--

参数	必填	值	描述
processDefinitionId	是	String	流程定义的id。
family	是	String	为 <code>users</code> 或 <code>groups</code> ，取决于身份关联的类型。
identityId	是	String	要从候选启动者中移除的 <code>userId</code> 或 <code>groupId</code> 。

Table 40. 从一个流程定义中删除一个候选启动者 - 响应码

响应码	描述
204	代表已找到流程定义，并已移除该身份关联。响应体设置为空。
404	代表未找到请求的流程定义，或者流程定义中并没有匹配url的身份关联。

成功响应体：

```
1 {
2   "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
3   "user": "kermit",
4   "group": null,
5   "type": "candidate"
6 }
```

15.3.11. 从一个流程定义中获取一个候选启动者 Get a candidate starter from a process definition

```
GET repository/process-definitions/{processDefinitionId}/identitylinks/{family}/{identityId}
```

Table 41. 从一个流程定义中获取一个候选启动者 - URL 参数

参数	必填	值	描述
processDefinitionId	是	String	流程定义的id。
family	是	String	为 <code>users</code> 或 <code>groups</code> ，取决于身份关联的类型。
identityId	是	String	要获取的候选启动者的 <code>userId</code> 或 <code>groupId</code> 。

Table 42. 从一个流程定义中获取一个候选启动者 - 响应码

响应码	描述
200	代表已找到流程定义，并已返回身份关联。
404	代表未找到请求的流程定义，或者流程定义中并没有匹配url的身份关联。

成功响应体：

```
1 {
2   "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
3   "user": "kermit",
4   "group": null,
5   "type": "candidate"
6 }
```

15.4. 模型 Models

15.4.1. 获取模型的列表 Get a list of models

```
GET repository/models
```

Table 43. 获取模型的列表 - URL 查询参数

able 43. 获取模型的列表 - URL 查询参数

响 应 码	必 填 参 数	描 述 值	描 述	
	id	否	String	只返回给定id的模型。
	category	否	String	只返回给定分类的模型。
	categoryLike	否	String	只返回分类like给定值的模型。 使用%字符作为通配符。
	categoryNotEquals	否	String	只返回不是给定分类的模型。
	name	否	String	只返回给定名字的模型。
	nameLike	否	String	只返回名字like给定值的模型。 使用%字符作为通配符。
	key	否	String	只返回给定key的模型。
	deploymentId	否	String	只返回在给定部署中部署的模型。
	version	否	Integer	只返回给定版本的模型。
	latestVersion	否	Boolean	如果值为true，则只返回最新版本的模型。最好与key联合使用。如果值为false，忽略本参数，返回所有版本。
	deployed	否	Boolean	如果值为true，只返回已部署的模型。如果值为false，只返回未部署的模型（deploymentId为null）。
	tenantId	否	String	只返回给定tenantId的模型。
	tenantIdLike	否	String	只返回tenantId like给定值的模型。
	withoutTenantId	否	Boolean	如果值为true，则只返回没有设置tenantId的模型。如果值为false，则忽略withoutTenantId参数。
	sort	否	id（默认），category，createTime，key，lastUpdateTime，name，version或tenantId	用于排序的参数，与'order'一起使用。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 44. 获取模型的列表 - 响应码

响 应 码	描 述
200	代表查询成功，并已返回模型。
400	代表传递的参数格式错误。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "name": "Model name",
5       "key": "Model key",
6       "category": "Model category",
```

```

7         "version":2,
8         "metaInfo":"Model metainfo",
9         "deploymentId":"7",
10        "id":"10",
11        "url":"http://localhost:8182/repository/models/10",
12        "createTime":"2013-06-12T14:31:08.612+0000",
13        "lastUpdateTime":"2013-06-12T14:31:08.612+0000",
14        "deploymentUrl":"http://localhost:8182/repository/deployments/7",
15        "tenantId":null
16    },
17
18    ...
19
20    ],
21    "total":2,
22    "start":0,
23    "sort":"id",
24    "order":"asc",
25    "size":2
26 }

```

15.4.2. 获取一个模型 Get a model

GET repository/models/{modelId}

Table 45. 获取一个模型 - URL 参数

参数	必填	值	描述
modelId	是	String	要获取的模型的id。

Table 46. 获取一个模型 - 响应码

响应码	描述
200	代表已找到并返回模型。
404	代表未找到请求的模型。

成功响应体:

```

1  {
2      "id":"5",
3      "url":"http://localhost:8182/repository/models/5",
4      "name":"Model name",
5      "key":"Model key",
6      "category":"Model category",
7      "version":2,
8      "metaInfo":"Model metainfo",
9      "deploymentId":"2",
10     "deploymentUrl":"http://localhost:8182/repository/deployments/2",
11     "createTime":"2013-06-12T12:31:19.861+0000",
12     "lastUpdateTime":"2013-06-12T12:31:19.861+0000",
13     "tenantId":null
14 }

```

15.4.3. 更新一个模型 Update a model

PUT repository/models/{modelId}

请求体:

```

1  {
2      "name":"Model name",
3      "key":"Model key",
4      "category":"Model category",
5      "version":2,
6      "metaInfo":"Model metainfo",
7      "deploymentId":"2",
8      "tenantId":"updatedTenant"
9  }

```


所有的请求值都是可选的。例如，可以只在请求体的JSON对象中包含'name'属性，则只更新模型的名字，而不影响其它任何字段。若明确包含了一个属性，并设置为null，则模型值将更新为null。例如：

响应码

`{"metaInfo" : null}`将清空模型的metaInfo。

描述

Table 47. 更新一个模型 - 响应码

响应码	描述
200	代表已找到并更新了模型。
404	代表未找到请求的模型。

成功响应体：

```
1 {
2   "id": "5",
3   "url": "http://localhost:8182/repository/models/5",
4   "name": "Model name",
5   "key": "Model key",
6   "category": "Model category",
7   "version": 2,
8   "metaInfo": "Model metaInfo",
9   "deploymentId": "2",
10  "deploymentUrl": "http://localhost:8182/repository/deployments/2",
11  "createTime": "2013-06-12T12:31:19.861+0000",
12  "lastUpdateTime": "2013-06-12T12:31:19.861+0000",
13  "tenantId": "updatedTenant"
14 }
```

15.4.4. 创建一个模型 Create a model

POST repository/models

请求体：

```
1 {
2   "name": "Model name",
3   "key": "Model key",
4   "category": "Model category",
5   "version": 1,
6   "metaInfo": "Model metaInfo",
7   "deploymentId": "2",
8   "tenantId": "tenant"
9 }
```

所有的请求值都是可选的。例如，可以只在请求体的JSON对象中包含'name'属性，则只设置模型的名字，其它所有字段都为null。

Table 48. 创建一个模型 - 响应码

响应码	描述
201	代表已创建模型。

成功响应体：

```
1 {
2   "id": "5",
3   "url": "http://localhost:8182/repository/models/5",
4   "name": "Model name",
5   "key": "Model key",
6   "category": "Model category",
7   "version": 1,
8   "metaInfo": "Model metaInfo",
9   "deploymentId": "2",
10  "deploymentUrl": "http://localhost:8182/repository/deployments/2",
11  "createTime": "2013-06-12T12:31:19.861+0000",
12  "lastUpdateTime": "2013-06-12T12:31:19.861+0000",
13  "tenantId": "tenant"
14 }
```

15.4.5. 删除一个模型 Delete a model

DELETE repository/models/{modelId}

Table 49. 删除一个模型 - URL 参数

参数	必填	值	描述
modelId	是	String	要删除的模型的id。

Table 50. 删除一个模型 - 响应码

响应码	描述
204	代表已找到并删除了模型。响应体设置为空。
404	代表未找到请求的模型。

15.4.6. 获取一个模型的编辑器源码 Get the editor source for a model

GET repository/models/{modelId}/source
--

Table 51. 获取一个模型的编辑器源码 - URL 参数

参数	必填	值	描述
modelId	是	String	模型的id。

Table 52. 获取一个模型的编辑器源码 - 响应码

响应码	描述
200	代表已找到模型，并已返回源码。
404	代表未找到请求的模型。

成功响应体：响应体包含了模型的原始编辑器源码。无论源码的content是什么，响应的content-type都设置为 `application/octet-stream`。

15.4.7. 设置一个模型的编辑器源码 Set the editor source for a model

PUT repository/models/{modelId}/source
--

Table 53. 设置一个模型的编辑器源码 - URL 参数

参数	必填	值	描述
modelId	是	String	模型的id。

请求体：

请求需要是 `multipart/form-data` 类型的。需要有唯一的file-part，包含源码的二进制值。

Table 54. 设置一个模型的编辑器源码 - 响应码

响应码	描述
200	代表已找到模型，并已更新源码。
404	代表未找到请求的模型。

成功响应体：响应体包含了模型的原始编辑器源码。无论源码的content是什么，响应的content-type都设置为 `application/octet-stream`。

15.4.8. 获取一个模型的附加编辑器源码 Get the extra editor source for a model

GET repository/models/{modelId}/source-extra
--

Table 55. 获取一个模型的附加编辑器源码 - URL 参数

参数	必填	值	描述
----	----	---	----

响应码 modelId	必填 是	描述 String	描述 模型的id。
----------------	---------	--------------	--------------

Table 56. 获取一个模型的附加编辑器源码 - 响应码

响应码	描述
200	代表已找到模型，并已返回源码。
404	代表未找到请求的模型。

成功响应体：响应体包含了模型的原始编辑器源码。无论源码的content是什么，响应的content-type都设置为 `application/octet-stream`。

15.4.9. 设置一个模型的附加编辑器源码 Set the extra editor source for a model

```
PUT repository/models/{modelId}/source-extra
```

Table 57. 设置一个模型的附加编辑器源码 - URL 参数

参数	必填	值	描述
modelId	是	String	模型的id。

请求体：

请求需要是 `multipart/form-data` 类型的。需要有唯一的file-part，包含源码的二进制值。

Table 58. 设置一个模型的附加编辑器源码 - 响应码

响应码	描述
200	代表已找到模型，并已更新附加源码。
404	代表未找到请求的模型。

成功响应体：响应体包含了模型的原始编辑器源码。无论源码的content是什么，响应的content-type都设置为 `application/octet-stream`。

15.5. 流程实例 Process Instances

15.5.1. 获取一个流程实例 Get a process instance

```
GET runtime/process-instances/{processInstanceId}
```

Table 59. 获取一个流程实例 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要获取的流程实例的id。

Table 60. 获取一个流程实例 - 响应码

响应码	描述
200	代表已找到并已返回流程实例。
404	代表未找到请求的流程实例。

成功响应体：

```
1 {
2   "id": "7",
3   "url": "http://localhost:8182/runtime/process-instances/7",
4   "businessKey": "myBusinessKey",
5   "suspended": false,
6   "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
```

```
7 | "activityId": "processTask",
8 | "tenantId": null
9 | }
```

15.5.2. 删除一个流程实例 Delete a process instance

DELETE runtime/process-instances/{processInstanceId}

Table 61. 删除一个流程实例 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要删除的流程实例的id。

Table 62. 删除一个流程实例 - 响应码

响应码	描述
204	代表已找到并删除了流程实例。响应体设置为空
404	代表未找到请求的流程实例。

15.5.3. 激活或暂停一个流程实例 Activate or suspend a process instance

PUT runtime/process-instances/{processInstanceId}

Table 63. 激活或暂停一个流程实例 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要激活/暂停的流程实例的id。

请求体（暂停）：

```
1 | {
2 |   "action": "suspend"
3 | }
```

请求体（激活）：

```
1 | {
2 |   "action": "activate"
3 | }
```

Table 64. 激活或暂停一个流程实例 - 响应码

响应码	描述
200	代表已找到流程实例，并执行了操作。
400	代表提供了非法的操作。
404	代表未找到请求的流程实例。
409	代表请求的流程实例操作无法执行，因为流程实例之前已经激活/暂停了。

15.5.4. 启动一个流程实例 Start a process instance

POST runtime/process-instances

请求体（通过流程定义id启动）：

```
1 | {
2 |   "processDefinitionId": "oneTaskProcess:1:158",
3 |   "businessKey": "myBusinessKey",
```

```

4   "variables": [
5     {
6       "name": "myVar",
7       "value": "This is a variable",
8     }
9   ]
10  }

```

请求体（通过流程定义`key`启动）：

```

1  {
2    "processDefinitionKey": "oneTaskProcess",
3    "businessKey": "myBusinessKey",
4    "tenantId": "tenant1",
5    "variables": [
6      {
7        "name": "myVar",
8        "value": "This is a variable",
9      }
10   ]
11  }

```

请求体（通过消息启动）：

```

1  {
2    "message": "newOrderMessage",
3    "businessKey": "myBusinessKey",
4    "tenantId": "tenant1",
5    "variables": [
6      {
7        "name": "myVar",
8        "value": "This is a variable",
9      }
10   ]
11  }

```

请求体中只能使用 `processDefinitionId`、`processDefinitionKey` 与 `message` 中的一个。`businessKey`、`variables` 与 `tenantId` 参数是可选的。如果省略了 `tenantId`，则将使用默认租户。关于变量格式的更多信息可以在[REST变量章节](#)找到。请注意提供的变量范围将被忽略，流程变量总是 `local` 的。

Table 65. 启动一个流程实例 - 响应码

响应码	描述
201	代表已创建流程实例。
400	代表（通过 <code>id</code> 或 <code>key</code> ）未找到流程定义，或者发送给定消息并未启动流程，或者传递了不合法的变量。状态描述中包含了关于错误的额外信息。

成功响应体：

```

1  {
2    "id": "7",
3    "url": "http://localhost:8182/runtime/process-instances/7",
4    "businessKey": "myBusinessKey",
5    "suspended": false,
6    "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
7    "activityId": "processTask",
8    "tenantId" : null
9  }

```

15.5.5. 流程实例的列表 List of process instances

```
GET runtime/process-instances
```

Table 66. 流程实例的列表 - URL 查询参数

参数	必填	值	描述
id	否	String	只返回给定id的流程实例。

参数	必填	值	描述
processDefinitionKey	否	String	只返回给定流程定义key的流程实例。
processDefinitionId	否	String	只返回给定流程定义id的流程实例。
businessKey	否	String	只返回给定businessKey的流程实例。
involvedUser	否	String	只返回给定用户参与的流程实例。
suspended	否	Boolean	如果值为 <code>true</code> ，则只返回暂停的流程实例。如果值为 <code>false</code> ，则只返回未暂停（激活）的流程实例。
superProcessInstanceId	否	String	只返回给定父流程实例id的流程实例（使用调用活动的流程）。
subProcessInstanceId	否	String	只返回给定子流程实例id的流程实例（通过调用活动启动的流程）。
excludeSubprocesses	否	Boolean	只返回不是子流程的流程实例。
includeProcessVariables	否	Boolean	是否在结果中包含流程变量。
tenantId	否	String	只返回给定tenantId的流程实例。
tenantIdLike	否	String	只返回tenantId like给定值的流程实例。
withoutTenantId	否	Boolean	如果值为 <code>true</code> ，则只返回没有设置tenantId的流程实例。如果值为 <code>false</code> ，则忽略withoutTenantId参数。
sort	否	String	排序字段，需要是 <code>id</code> （默认）， <code>processDefinitionId</code> ， <code>tenantId</code> 或 <code>processDefinitionKey</code> 中的一个。

可以在这个URL中使用通用分页与排序查询参数。

Table 67. 流程实例的列表 - 响应码

响应码	描述
200	代表请求成功，并已返回流程实例。
400	代表传递的参数格式错误。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "id": "7",
5       "url": "http://localhost:8182/runtime/process-instances/7",
6       "businessKey": "myBusinessKey",
7       "suspended": false,
8       "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
9       "activityId": "processTask",
10      "tenantId": null
11    }
12  ],
13
14  ],
15  "total": 2,
16  "start": 0,
17  "sort": "id",
18  "order": "asc",
19  "size": 2
20 }
```

15.5.6. 查询流程实例 Query process instances

POST query/process-instances

请求体:

```
1 {
2   "processDefinitionKey": "oneTaskProcess",
3   "variables":
4   [
5     {
6       "name" : "myVariable",
7       "value" : 1234,
8       "operation" : "equals",
9       "type" : "long"
10    }
11  ]
12 }
```

请求体可以包含所有在[流程实例的列表URL](#)查询中可用的过滤器。另外，也可以在查询中包含一个变量的数组，使用[这里描述](#)的格式。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 68. 查询流程实例 - 响应码

响应码	描述
200	代表请求成功，并已返回流程实例。
400	代表传递的参数格式错误。状态描述中包含了额外信息。

成功响应体:

```
1 {
2   "data": [
3     {
4       "id": "7",
5       "url": "http://localhost:8182/runtime/process-instances/7",
6       "businessKey": "myBusinessKey",
7       "suspended": false,
8       "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
9       "activityId": "processTask",
10      "tenantId" : null
11    }
12  ],
13
14  ],
15  "total": 2,
16  "start": 0,
17  "sort": "id",
18  "order": "asc",
19  "size": 2
20 }
```

15.5.7. 获取一个流程实例的流程图 Get diagram for a process instance

GET runtime/process-instances/{processInstanceId}/diagram

Table 69. 获取一个流程实例的流程图 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要获取流程图的流程实例的id。

Table 70. 获取一个流程实例的流程图 - 响应码

响应码	描述
200	代表已找到流程实例，并已返回流程图。
400	代表已找到请求的流程实例，但该流程未包含任何图形信息（BPMN: DI），因此不能创建流程图。

响应码 404	必填	描述 代表未找到请求的流程实例。	描述
------------	----	---------------------	----

成功响应体:

```

1 {
2   "id": "7",
3   "url": "http://localhost:8182/runtime/process-instances/7",
4   "businessKey": "myBusinessKey",
5   "suspended": false,
6   "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
7   "activityId": "processTask"
8 }
```

15.5.8. 获取流程实例的参与者 Get involved people for process instance

GET runtime/process-instances/{processInstanceId}/identitylinks

Table 71. 获取流程实例的参与者 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要查询关联的流程实例的id。

Table 72. 获取流程实例的参与者 - 响应码

响应码	描述
200	代表已找到流程实例，并已返回关联。
404	代表未找到请求的流程实例。

成功响应体:

```

1 [
2   {
3     "url": "http://localhost:8182/runtime/process-instances/5/identitylinks/users/john/customType",
4     "user": "john",
5     "group": null,
6     "type": "customType"
7   },
8   {
9     "url": "http://localhost:8182/runtime/process-instances/5/identitylinks/users/paul/candidate",
10    "user": "paul",
11    "group": null,
12    "type": "candidate"
13  }
14 ]
```

请注意 `groupId` 永远为null，因为只有用户才能参与流程实例。

15.5.9. 为一个流程实例添加一个参与用户 Add an involved user to a process instance

POST runtime/process-instances/{processInstanceId}/identitylinks

Table 73. 为一个流程实例添加一个参与用户 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要添加关联的流程实例。

请求体:

```

1 {
2   "userId": "kermit",
3   "type": "participant"
4 }
```

`userId`与`type`都是必填的。

Table 74. 为一个流程实例添加一个参与用户 - 响应码

响应码	描述
201	代表已找到流程实例，并已返回关联。
400	代表请求体中未包含 <code>userId</code> 或 <code>type</code> 。
404	代表未找到请求的流程实例。

成功响应体：

```
1 {
2   "url": "http://localhost:8182/runtime/process-instances/5/identitylinks/users/john/customType",
3   "user": "john",
4   "group": null,
5   "type": "customType"
6 }
```

请注意`groupId`永远为`null`，因为只有用户才能参与流程实例。

15.5.10. 从一个流程实例中移除一个参与用户 Remove an involved user to from process instance

```
DELETE runtime/process-instances/{processInstanceId}/identitylinks/users/{userId}/{type}
```

Table 75. 从一个流程实例中移除一个参与用户 - URL 参数

参数	必填	值	描述
<code>processInstanceId</code>	是	String	流程实例的id。
<code>userId</code>	是	String	要删除关联的用户的id。
<code>type</code>	是	String	要删除的关联的类型。

Table 76. 从一个流程实例中移除一个参与用户 - 响应码

响应码	描述
204	代表已找到流程实例，并已删除关联。响应体设置为空。
404	代表未找到请求的流程实例，或者要删除的关联不存在。响应状态中包含了关于错误的额外信息。

成功响应体：

```
1 {
2   "url": "http://localhost:8182/runtime/process-instances/5/identitylinks/users/john/customType",
3   "user": "john",
4   "group": null,
5   "type": "customType"
6 }
```

请注意`groupId`永远为`null`，因为只有用户才能参与流程实例。

15.5.11. 一个流程实例的变量的列表 List of variables for a process instance

```
GET runtime/process-instances/{processInstanceId}/variables
```

Table 77. 一个流程实例的变量的列表 - URL 参数

参数	必填	值	描述
<code>processInstanceId</code>	是	String	要列表变量的流程实例的id。

Table 78. 一个流程实例的变量的列表 - 响应码

响应码	描述
-----	----

响应码	描述
200	代表已找到流程实例，并已返回变量。
404	代表未找到请求的流程实例。

成功响应体：

```
1  [
2    {
3      "name": "intProcVar",
4      "type": "integer",
5      "value": 123,
6      "scope": "local"
7    },
8    {
9      "name": "byteArrayProcVar",
10     "type": "binary",
11     "value": null,
12     "valueUrl": "http://localhost:8182/runtime/process-instances/5/variables/byteArrayProcVar/data",
13     "scope": "local"
14   }
15 ]
```

如果变量是二进制变量或序列化值，则 `valueUrl` 指向获取原始值的URL。如果是一个简单变量，则在响应中显示值。请注意只会返回 `local` 范围的变量，因为流程实例变量没有 `global` 范围。

15.5.12. 获取一个流程实例的一个变量 Get a variable for a process instance

```
GET runtime/process-instances/{processInstanceId}/variables/{variableName}
```

Table 79. 获取一个流程实例的一个变量 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要获取变量的流程实例的id。
variableName	是	String	要获取的变量的名字。

Table 80. 获取一个流程实例的一个变量 - 响应码

响应码	描述
200	代表已找到流程实例及变量，并已返回变量。
400	代表请求体不完整，或者包含不合法值。状态描述中包含了关于错误的额外信息。
404	代表未找到请求的流程实例，或者流程实例中没有给定名字的变量。状态描述中包含了关于错误的额外信息。

成功响应体：

```
1  {
2    "name": "intProcVar",
3    "type": "integer",
4    "value": 123,
5    "scope": "local"
6  }
```

如果变量是二进制变量或序列化值，则 `valueUrl` 指向获取原始值的URL。如果是一个简单变量，则在响应中显示值。请注意只会返回 `local` 范围的变量，因为流程实例变量没有 `global` 范围。

15.5.13. 为一个流程实例创建（或更新）变量 Create (or update) variables on a process instance

```
POST runtime/process-instances/{processInstanceId}/variables
```

```
PUT runtime/process-instances/{processInstanceId}/variables
```

当使用 **POST** 时，会创建所有传递的变量。如果流程实例中已经存在某个变量，则请求结果为错误（409 - 冲突）。当使用 **PUT** 时，会创建流程实例中不存在的变量；已存在变量的将会被覆盖，而没有错误。

参数	必填	值	描述
----	----	---	----

Table 81. 为一个流程实例创建（或更新）变量 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要操作变量的流程实例的id。

请求体：

```
[
  {
    "name": "intProcVar",
    "type": "integer",
    "value": 123
  },
  ...
]
```

请求体的数组中可以传递任意数量的变量。可以在[REST变量章节](#)找到关于变量格式的更多信息。请注意范围将被忽略，流程实例中只能设置 **local** 变量。

Table 82. 为一个流程实例创建（或更新）变量 - 响应码

响应码	描述
201	代表已找到流程实例，并已创建变量。
400	代表请求体不完整，或含有非法值。状态描述中包含了关于错误的额外信息。
404	代表未找到请求的流程实例。
409	代表已找到流程实例，但其已包含了给定名字的变量（只在使用 POST 方法时抛出）。改用更新方法。

成功响应体：

```
[
  {
    "name": "intProcVar",
    "type": "integer",
    "value": 123,
    "scope": "local"
  },
  ...
]
```

15.5.14. 为一个流程实例更新一个变量 Update a single variable on a process instance

```
PUT runtime/process-instances/{processInstanceId}/variables/{variableName}
```

Table 83. 为一个流程实例更新一个变量 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要操作变量的流程实例的id。
variableName	是	String	要更新的变量的名字。

请求体：

```
1 {
2   "name": "intProcVar"
3   "type": "integer"
```

```
4     "value":123
5 }
}
```

请求体的数组中可以传递任意数量的变量。可以在[REST变量章节](#)找到关于变量格式的更多信息。请注意范围将被忽略，流程实例中只能设置 **local** 变量。

Table 84. 为一个流程实例更新一个变量 - 响应码

响应码	描述
200	代表已找到流程实例与变量，并已更新变量。
404	代表未找到请求的流程实例，或者流程实例中没有给定名字的变量。状态描述中包含了关于错误的额外信息。

成功响应体:

```
1 {
2   "name": "intProcVar",
3   "type": "integer",
4   "value": 123,
5   "scope": "local"
6 }
```

如果变量是二进制变量或序列化值，则 **valueUrl** 指向获取原始值的URL。如果是一个简单变量，则在响应中显示值。请注意只会返回 **local** 范围的变量，因为流程实例变量没有 **global** 范围。

15.5.15. 为一个流程实例创建一个新的二进制变量 Create a new binary variable on a process-instance

```
POST runtime/process-instances/{processInstanceId}/variables
```

Table 85. 为一个流程实例创建一个新的二进制变量 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要创建新变量的流程实例的 id。

请求体: 请求需要是 **multipart/form-data** 类型的。需要有唯一的 **file-part**，包含变量的二进制值。另外，也可以使用下列额外的 **form-fields**:

- **name**: 变量需要的名字。
- **type**: 创建的变量的类型。如果省略，则使用 **binary**，并且将请求中的二进制数据保存为字节数组。

成功响应体:

```
1 {
2   "name" : "binaryVariable",
3   "scope" : "local",
4   "type" : "binary",
5   "value" : null,
6   "valueUrl" : "http://.../runtime/process-instances/123/variables/binaryVariable/data"
7 }
```

Table 86. 为一个流程实例创建一个新的二进制变量 - 响应码

响应码	描述
201	代表已创建变量，并已返回结果。
400	代表缺少要创建的变量的名字。状态消息提供了额外信息。
404	代表未找到请求的流程实例。
409	代表流程实例中已经有给定名字的变量。改用PUT方法更新任务变量。

响应码	必填	描述
415		代表序列化数据中包含了一个运行Activiti引擎的JVM中不存在的类的对象，因此不能反序列化。

15.5.16. 为一个流程实例更新一个已有的二进制变量 Update an existing binary variable on a process-instance

PUT runtime/process-instances/{processInstanceId}/variables

Table 87. 为一个流程实例更新一个已有的二进制变量 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要更新变量的流程实例的id。

请求体：请求需要是 **multipart/form-data** 类型的。需要有唯一的 **file-part**，包含变量的二进制值。另外，也可以使用下列额外的 form-fields：

- name**：变量需要的名字。
- type**：创建的变量的类型。如果省略，则使用 **binary**，并且将请求中的二进制数据保存为字节数组。

成功响应体：

```

1 {
2   "name" : "binaryVariable",
3   "scope" : "local",
4   "type" : "binary",
5   "value" : null,
6   "valueUrl" : "http://.../runtime/process-instances/123/variables/binaryVariable/data"
7 }
```

Table 88. 为一个流程实例更新一个已有的二进制变量 - 响应码

响应码	描述
200	代表已更新变量，并已返回结果。
400	代表缺少要更新的变量的名字。状态消息提供了额外信息。
404	代表未找到请求的流程实例，或者流程实例中没有给定名字的变量。
415	代表序列化数据中包含了一个运行Activiti引擎的JVM中不存在的类的对象，因此不能反序列化。

15.6. 执行 Executions

15.6.1. 获取一个执行 Get an execution

GET runtime/executions/{executionId}

Table 89. 获取一个执行 - URL 参数

参数	必填	值	描述
executionId	是	String	要获取的执行的id。

Table 90. 获取一个执行 - 响应码

响应码	描述
200	代表已找到并返回执行。
404	代表未找到执行

成功响应体：

```

1  {
2    "id": "5",
3    "url": "http://localhost:8182/runtime/executions/5",
4    "parentId": null,
5    "parentUrl": null,
6    "processInstanceId": "5",
7    "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
8    "suspended": false,
9    "activityId": null,
10   "tenantId": null
11 }

```

15.6.2. 对一个执行进行操作 Execute an action on an execution

PUT runtime/executions/{executionId}

Table 91. 对一个执行进行操作 - URL 参数

参数	必填	值	描述
executionId	是	String	要操作的执行的id。

请求体（为一个执行发信号）：

```

1  {
2    "action": "signal"
3  }

```

请求体（为一个执行发信号接收事件）：

```

1  {
2    "action": "signalEventReceived",
3    "signalName": "mySignal",
4    "variables": [ ]
5  }

```

通知执行：已经收到了一个信号事件。要求有一个 **signalName** 参数。可以传递可选的 **variables**，将在进行操作前设置到执行中。

请求体（为一个执行发消息接收事件）：

```

1  {
2    "action": "messageEventReceived",
3    "messageName": "myMessage",
4    "variables": [ ]
5  }

```

通知执行：已经收到了一个消息事件。要求有一个 **messageName** 参数。可以传递可选的 **variables**，将在进行操作前设置到执行中。

Table 92. 对一个执行进行操作 - 响应码

响应码	描述
200	代表已找到执行，并进行了操作。
204	代表已找到执行，进行了操作，该操作导致执行结束。
400	代表请求了非法的操作，请求中缺少必要的参数，或者传递了非法的变量。状态描述中包含了关于错误的额外信息。
404	代表未找到执行

成功响应体（当执行并未因该操作结束时）：

```

1  {
2    "id": "5",
3    "url": "http://localhost:8182/runtime/executions/5",
4    "parentId": null,
5    "parentUrl": null,
6    "processInstanceId": "5",
7    "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",

```



```
8     "suspended":false,
9     "activityId":null,
10    "tenantId" : null
11 }
```

15.6.3. 获取一个执行中的激活活动 Get active activities in an execution

```
GET runtime/executions/{executionId}/activities
```

返回执行中与所有子执行中（以及它们的子执行，递归）激活的所有活动。

Table 93. 获取一个执行中的激活活动 - URL 参数

参数	必填	值	描述
executionId	是	String	要获取活动的执行的id。

Table 94. 获取一个执行中的激活活动 - 响应码

响应码	描述
200	代表已找到执行，并已返回活动。
404	代表未找到执行

成功响应体：

```
1 [
2   "userTaskForManager",
3   "receiveTask"
4 ]
```

15.6.4. 执行的列表 List of executions

```
GET runtime/executions
```

Table 95. 执行的列表 - URL 查询参数

参数	必填	值	描述
id	否	String	只返回给定id的执行。
activityId	否	String	只返回给定activity id的执行。
processDefinitionKey	否	String	只返回给定流程定义key的执行。
processDefinitionId	否	String	只返回给定流程定义id的执行。
processInstanceId	否	String	只返回给定流程实例中的执行。
messageEventSubscriptionName	否	String	只返回订阅了给定名字的消息的执行。
signalEventSubscriptionName	否	String	只返回订阅了给定名字的信号的执行。
parentId	否	String	只返回给定执行的直接子执行。
tenantId	否	String	只返回给定tenantId的执行。
tenantIdLike	否	String	只返回tenantId like给定值的执行。
withoutTenantId	否	Boolean	如果值为true，则只返回未设置tenantId的执行。如果值为false，则忽略withoutTenantId参数。

响应码 sort	必填 否	值 String	描述 排序字段，需要
			为 <code>processInstanceId</code> （默认）， <code>processDefinitionId</code> ， <code>processDefinitionKey</code> 或 <code>tenantId</code> 。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 96. 执行的列表 - 响应码

响应码	描述
200	代表请求成功，并已返回执行。
400	代表传递的参数格式错误。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "id": "5",
5       "url": "http://localhost:8182/runtime/executions/5",
6       "parentId": null,
7       "parentUrl": null,
8       "processInstanceId": "5",
9       "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
10      "suspended": false,
11      "activityId": null,
12      "tenantId": null
13    },
14    {
15      "id": "7",
16      "url": "http://localhost:8182/runtime/executions/7",
17      "parentId": "5",
18      "parentUrl": "http://localhost:8182/runtime/executions/5",
19      "processInstanceId": "5",
20      "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
21      "suspended": false,
22      "activityId": "processTask",
23      "tenantId": null
24    }
25  ],
26  "total": 2,
27  "start": 0,
28  "sort": "processInstanceId",
29  "order": "asc",
30  "size": 2
31 }
```

15.6.5. 查询执行 Query executions

```
POST query/executions
```

请求体：

```
1 {
2   "processDefinitionKey": "oneTaskProcess",
3   "variables":
4   [
5     {
6       "name" : "myVariable",
7       "value" : 1234,
8       "operation" : "equals",
9       "type" : "long"
10    }
11  ],
12  "processInstanceVariables":
13  [
14    {
15      "name" : "processVariable",
16      "value" : "some string",
17      "operation" : "equals",
18      "type" : "string"
19    }
20  ]
21 }
```

20]
21	}

响应码	描述
-----	----

请求体可以包含所有在[列表执行URL](#)查询中可用的过滤器。另外，也可以在查询中包含一个 `variables` 与 `processInstanceVariables` 的数组，使用[这里描述](#)的格式。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 97. 查询执行 - 响应码

响应码	描述
200	代表请求成功，并已返回执行。
400	代表传递的参数格式错误。状态描述中包含了额外信息。

成功响应体：

1	{
2	"data":[
3	{
4	"id":"5",
5	"url":"http://localhost:8182/runtime/executions/5",
6	"parentId":null,
7	"parentUrl":null,
8	"processInstanceId":"5",
9	"processInstanceUrl":"http://localhost:8182/runtime/process-instances/5",
10	"suspended":false,
11	"activityId":null,
12	"tenantId":null
13	},
14	{
15	"id":"7",
16	"url":"http://localhost:8182/runtime/executions/7",
17	"parentId":"5",
18	"parentUrl":"http://localhost:8182/runtime/executions/5",
19	"processInstanceId":"5",
20	"processInstanceUrl":"http://localhost:8182/runtime/process-instances/5",
21	"suspended":false,
22	"activityId":"processTask",
23	"tenantId":null
24	}
25],
26	"total":2,
27	"start":0,
28	"sort":"processInstanceId",
29	"order":"asc",
30	"size":2
31	}

15.6.6. 一个执行中的变量的列表 List of variables for an execution

GET runtime/executions/{executionId}/variables?scope={scope}
--

Table 98. 一个执行中的变量的列表 - URL 参数

参数	必填	值	描述
executionId	是	String	要列表变量的执行的id。
scope	否	String	可以为 <code>local</code> 或 <code>global</code> 。若省略，则同时返回本地与全局范围的变量。

Table 99. 一个执行中的变量的列表 - 响应码

响应码	描述
200	代表已找到执行，并已返回变量。
404	代表未找到请求的执行。

成功响应体：

--

```

1  [
2    {
3      "name": "intProcVar",
4      "type": "integer",
5      "value": 123,
6      "scope": "global"
7    },
8    {
9      "name": "byteArrayProcVar",
10     "type": "binary",
11     "value": null,
12     "valueUrl": "http://localhost:8182/runtime/process-instances/5/variables/byteArrayProcVar/data",
13     "scope": "local"
14   }
15 ]
16
17 ]

```

如果变量是二进制变量或序列化值，则 `valueUrl` 指向获取原始值的URL。如果是一个简单变量，则在响应中显示值。

15.6.7. 获取一个执行的一个变量 Get a variable for an execution

```
GET runtime/executions/{executionId}/variables/{variableName}?scope={scope}
```

Table 100. 获取一个执行的一个变量 - URL 参数

参数	必填	值	描述
executionId	是	String	要获取变量的执行的id
variableName	是	String	要获取的变量的名字。
scope	否	String	<code>local</code> 或 <code>global</code> 。若省略，则（存在时）返回本地变量。不存在本地变量时，则（存在时）返回全局变量。

Table 101. 获取一个执行的一个变量 - 响应码

响应码	描述
200	代表已找到执行与变量，并已返回变量。
400	代表请求体不完整，或者包含不合法值。状态描述中包含了关于错误的额外信息。
404	代表未找到请求的执行，或者在请求的范围内执行没有给定名字的变量（若省略了范围查询条件，则本地与全局范围中都没有该变量）。状态描述中包含了关于错误的额外信息。

成功响应体：

```

1  {
2    "name": "intProcVar",
3    "type": "integer",
4    "value": 123,
5    "scope": "local"
6  }

```

如果变量是二进制变量或序列化值，则 `valueUrl` 指向获取原始值的URL。如果是一个简单变量，则在响应中显示值。

15.6.8. 为一个执行创建（或更新）变量 Create (or update) variables on an execution

```
POST runtime/executions/{executionId}/variables
```

```
PUT runtime/executions/{executionId}/variables
```

当使用 `POST` 时，会创建所有传递的变量。如果执行中已经存在某个变量，则请求结果为错误（409 - 冲突）。当使用 `PUT` 时，会创建执行中不存在的变量；已存在变量的将会被覆盖，而没有错误。

参数码	必填	描述	描述
-----	----	----	----

Table 102. 为一个执行创建（或更新）变量 - URL 参数

参数	必填	值	描述
executionId	是	String	要操作变量的执行的id。

请求体:

1	[
2	{
3	"name": "intProcVar",
4	"type": "integer",
5	"value": 123,
6	"scope": "local"
7	}
8	
9	
10	
11]

请注意只能提供相同范围的变量。如果请求体数组中包含了不同范围的变量，则请求结果为错误（400 - 错误请求）。请求体的数组中可以传递任意数量的变量。可以在[REST变量章节](#)找到关于变量格式的更多信息。

Table 103. 为一个执行创建（或更新）变量 - 响应码

响应码	描述
201	代表已找到执行，并已创建变量。
400	代表请求体不完整，或者包含不合法值。状态描述中包含了关于错误的额外信息。
404	代表未找到请求的执行。
409	代表已找到执行，但其已包含了给定名字的变量（只在使用 <code>POST</code> 方法时抛出）。改用更新方法。

成功响应体:

1	[
2	{
3	"name": "intProcVar",
4	"type": "integer",
5	"value": 123,
6	"scope": "local"
7	}
8	
9	
10	
11]

15.6.9. 为一个执行更新一个变量 Update a variable on an execution

PUT runtime/executions/{executionId}/variables/{variableName}

Table 104. 为一个执行更新一个变量 - URL 参数

参数	必填	值	描述
executionId	是	String	要更新变量的执行的id。
variableName	是	String	要更新的变量的名字。

请求体:

1	{
2	"name": "intProcVar"
3	"type": "integer"

```
4     "value":123,
5     "scope":"global"
6 }
```

可以在[REST变量章节](#)找到关于变量格式的更多信息。

Table 105. 为一个执行更新一个变量 - 响应码

响应码	描述
200	代表已找到执行与变量，并已更新变量。
404	代表未找到请求的执行，或者执行没有给定名字的变量。状态描述中包含了关于错误的额外信息。

成功响应体：

```
1  {
2    "name":"intProcVar",
3    "type":"integer",
4    "value":123,
5    "scope":"global"
6 }
```

如果变量是二进制变量或序列化值，则 `valueUrl` 指向获取原始值的URL。如果是一个简单变量，则在响应中显示值。

15.6.10. 为一个执行创建一个新的二进制变量 Create a new binary variable on an execution

```
POST runtime/executions/{executionId}/variables
```

Table 106. 为一个执行创建一个新的二进制变量 - URL 参数

参数	必填	值	描述
executionId	是	String	要创建新变量的执行的id。

请求体：请求需要是 `multipart/form-data` 类型的。需要有唯一的 `file-part`，包含变量的二进制值。另外，也可以使用下列额外的 `form-fields`：

- `name`：变量需要的名字。
- `type`：创建的变量的类型。如果省略，则使用 `binary`，并且将请求中的二进制数据保存为字节数组。
- `scope`：创建变量的范围。如果省略，则使用 `local`。

成功响应体：

```
1  {
2    "name" : "binaryVariable",
3    "scope" : "local",
4    "type" : "binary",
5    "value" : null,
6    "valueUrl" : "http://.../runtime/executions/123/variables/binaryVariable/data"
7  }
```

Table 107. 为一个执行创建一个新的二进制变量 - 响应码

响应码	描述
201	代表已创建变量，并已返回结果。
400	代表缺少要创建的变量的名字。状态消息提供了额外信息。
404	代表未找到请求的执行。
409	代表执行已包含了给定名字的变量（只在使用POST方法时抛出）。改用PUT方法更新变量。

响应码 415	必填	描述 代表序列化数据中包含了一个运行Activiti引擎的JVM中不存在的类的对象，因此不能反序列化。
------------	----	--

15.6.11. 为一个执行更新一个已有二进制变量 Update an existing binary variable on a process-instance

```
PUT runtime/executions/{executionId}/variables/{variableName}
```

Table 108. 为一个执行更新一个已有二进制变量 - URL 参数

参数	必填	值	描述
executionId	是	String	要更新变量的执行的id。
variableName	是	String	要更新的变量的名字。

请求体：请求需要是 **multipart/form-data** 类型的。需要有唯一的 **file-part**，包含变量的二进制值。另外，也可以使用下列额外的 form-fields：

- **name**：变量需要的名字。
- **type**：创建的变量的类型。如果省略，则使用 **binary**，并且将请求中的二进制数据保存为字节数组。
- **scope**：创建变量的范围。如果省略，则使用 **local**。

成功响应体：

```
1 {
2   "name" : "binaryVariable",
3   "scope" : "local",
4   "type" : "binary",
5   "value" : null,
6   "valueUrl" : "http://.../runtime/executions/123/variables/binaryVariable/data"
7 }
```

Table 109. 为一个执行更新一个已有二进制变量 - 响应码

响应码	描述
200	代表已更新变量，并已返回结果。
400	代表缺少要更新的变量的名字。状态消息提供了额外信息。
404	代表未找到请求的执行，或者执行中没有给定名字的变量。
415	代表序列化数据中包含了一个运行Activiti引擎的JVM中不存在的类的对象，因此不能反序列化。

15.7. 任务 Tasks

15.7.1. 获取一个任务 Get a task

```
GET runtime/tasks/{taskId}
```

Table 110. 获取一个任务 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取的任务的id。

Table 111. 获取一个任务 - 响应码

响应码	描述
200	代表已找到并返回了任务。
404	代表未找到请求的任务。

成功响应体:

```
1 {
2   "assignee" : "kermit",
3   "createTime" : "2013-04-17T10:17:43.902+0000",
4   "delegationState" : "pending",
5   "description" : "Task description",
6   "dueDate" : "2013-04-17T10:17:43.902+0000",
7   "execution" : "http://localhost:8182/runtime/executions/5",
8   "id" : "8",
9   "name" : "My task",
10  "owner" : "owner",
11  "parentTask" : "http://localhost:8182/runtime/tasks/9",
12  "priority" : 50,
13  "processDefinition" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
14  "processInstance" : "http://localhost:8182/runtime/process-instances/5",
15  "suspended" : false,
16  "taskDefinitionKey" : "theTask",
17  "url" : "http://localhost:8182/runtime/tasks/8",
18  "tenantId" : null
19 }
```

- `delegationState`: 任务的代理状态, 可以为 `null`、`"pending"` 或 `"resolved"`。

15.7.2. 任务的列表 List of tasks

```
GET runtime/tasks
```

Table 112. 任务的列表 - URL 查询参数

参数	必填	值	描述
name	否	String	只返回给定名字的任务。
nameLike	否	String	只返回名字like给定名字的任务。
description	否	String	只返回给定描述的任务。
priority	否	Integer	只返回给定优先级的任务。
minimumPriority	否	Integer	只返回优先级高于给定值的任务。
maximumPriority	否	Integer	只返回优先级低于给定值的任务。
assignee	否	String	只返回指派至给定用户的任务。
assigneeLike	否	String	只返回办理人like给定值的任务。
owner	否	String	只返回属主为给定用户的任务。
ownerLike	否	String	只返回属主like给定值的任务。
unassigned	否	Boolean	只返回未指派的任务。如果传递 <code>false</code> , 则忽略本参数。
delegationState	否	String	只返回给定代理状态的任务。可用值为 <code>pending</code> 与 <code>resolved</code> 。
candidateUser	否	String	只返回可以被给定用户申领的任务。包括用户为候选人, 以及用户所在组为候选组的任务。

参数	必填 否	值	描述
candidateGroup	否	String	只返回可以被给定组中的用户申领的任务。
candidateGroups	否	String	只返回可以被给定组中的用户申领的任务。逗号分隔值。
involvedUser	否	String	只返回给定用户参与的任务。
taskDefinitionKey	否	String	只返回给定任务定义id的任务。
taskDefinitionKeyLike	否	String	只返回任务定义id like给定值的任务。
processInstanceId	否	String	只返回给定id流程实例中的任务。
processInstanceBusinessKey	否	String	只返回给定businessKey流程实例中的任务。
processInstanceBusinessKeyLike	否	String	只返回businessKey like给定值的流程实例中的任务。
processDefinitionKey	否	String	只返回给定key的流程定义的流程实例中的任务。
processDefinitionKeyLike	否	String	只返回key like给定值的流程定义的流程实例中的任务。
processDefinitionName	否	String	只返回给定名字的流程定义的流程实例中的任务。
processDefinitionNameLike	否	String	只返回名字like给定值的流程定义的流程实例中的任务。
executionId	否	String	只返回给定id的执行中的任务。
createdOn	否	ISO Date	只返回给定日期创建的任务。
createdBefore	否	ISO Date	只返回给定日期之前创建的任务。
createdAfter	否	ISO Date	只返回给定日期之后创建的任务。
dueOn	否	ISO Date	只返回给定日期到期的任务。
dueBefore	否	ISO Date	只返回给定日期前到期的任务。
dueAfter	否	ISO Date	只返回给定日期后到期的任务。
withoutDueDate	否	boolean	只返回没有到期日期的任务。如果值为false则忽略本参数。
excludeSubTasks	否	Boolean	只返回不是另一个任务的子任务的任务。

参数名 active	必填 否	数据类型 Boolean	描述 如果值为true，则只返回未暂停（要么所在流程未暂停，要么根本不在流程中）的任务。如果值为false，则只返回已暂停流程实例中的任务。
includeTaskLocalVariables	否	Boolean	是否在结果中包含本地变量。
includeProcessVariables	否	Boolean	是否在结果中包含流程变量。
tenantId	否	String	只返回给定tenantId的任务。
tenantIdLike	否	String	只返回tenantId like给定值的任务。
withoutTenantId	否	Boolean	如果值为true，则只返回未设置tenantId的任务。如果值为false，则忽略withoutTenantId参数。
candidateOrAssigned	否	String	选择已被申领，或已指派给用户，或等待用户（候选用户或组）申领的任务。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 113. 任务的列表 - 响应码

响应码	描述
200	代表请求成功，并已返回任务。
400	代表传递的参数格式错误，或'delegationState'使用了不合法的值（不是'pending'与'resolved'）。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "assignee" : "kermit",
5       "createTime" : "2013-04-17T10:17:43.902+0000",
6       "delegationState" : "pending",
7       "description" : "Task description",
8       "dueDate" : "2013-04-17T10:17:43.902+0000",
9       "execution" : "http://localhost:8182/runtime/executions/5",
10      "id" : "8",
11      "name" : "My task",
12      "owner" : "owner",
13      "parentTask" : "http://localhost:8182/runtime/tasks/9",
14      "priority" : 50,
15      "processDefinition" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
16      "processInstance" : "http://localhost:8182/runtime/process-instances/5",
17      "suspended" : false,
18      "taskDefinitionKey" : "theTask",
19      "url" : "http://localhost:8182/runtime/tasks/8",
20      "tenantId" : null
21    }
22  ],
23  "total": 1,
24  "start": 0,
25  "sort": "name",
26  "order": "asc",
27  "size": 1
28 }
```

15.7.3. 查询任务 Query for tasks

```
POST query/tasks
```

请求体:

```
1 {
2   "name" : "My task",
3   "description" : "The task description",
4
5   ...
6
7   "taskVariables" : [
8     {
9       "name" : "myVariable",
10      "value" : 1234,
11      "operation" : "equals",
12      "type" : "long"
13    }
14  ],
15
16   "processInstanceVariables" : [
17     {
18       ...
19     }
20  ]
21 }
22 }
```

支持的JSON参数字段与[获取任务集合](#)的参数一模一样（除了candidateGroupIn，只能在POST任务查询REST服务中使用），但是使用JSON体参数而不是URL参数，可以进行更高级的查询，也可以避免请求URI太长的错误。另外，查询可以通过任务与流程变量进行过滤。`taskVariables`与`processInstanceVariables`都是JSON数组，包含[这里描述](#)的格式的对象。

Table 114. 查询任务 - 响应码

响应码	描述
200	代表请求成功，并已返回任务。
400	代表传递的参数格式错误，或'delegationState'使用了不合法的值（不是'pending'与'resolved'）。状态描述中包含了额外信息。

成功响应体:

```
1 {
2   "data": [
3     {
4       "assignee" : "kermit",
5       "createTime" : "2013-04-17T10:17:43.902+0000",
6       "delegationState" : "pending",
7       "description" : "Task description",
8       "dueDate" : "2013-04-17T10:17:43.902+0000",
9       "execution" : "http://localhost:8182/runtime/executions/5",
10      "id" : "8",
11      "name" : "My task",
12      "owner" : "owner",
13      "parentTask" : "http://localhost:8182/runtime/tasks/9",
14      "priority" : 50,
15      "processDefinition" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
16      "processInstance" : "http://localhost:8182/runtime/process-instances/5",
17      "suspended" : false,
18      "taskDefinitionKey" : "theTask",
19      "url" : "http://localhost:8182/runtime/tasks/8",
20      "tenantId" : null
21    }
22  ],
23   "total": 1,
24   "start": 0,
25   "sort": "name",
26   "order": "asc",
27   "size": 1
28 }
```

15.7.4. 更新一个任务 Update a task

```
PUT runtime/tasks/{taskId}
```

JSON体:

```
{
```

```

1  "assignee" : "assignee",
2  "delegationState" : "resolved",
3  "description" : "New task description",
4  "dueDate" : "2013-04-17T13:06:02.438+02:00",
5  "name" : "New task name",
6  "owner" : "owner",
7  "parentTaskId" : "3",
8  "priority" : 20
9  }
10

```

所有的请求值都是可选的。例如，可以只在请求体的JSON对象中包含'assignee'属性，则只更新任务的办理人，而不影响其它任何字段。若明确包含了一个属性，并设置为null，则任务值将更新为null。例如：`{"dueDate" : null}`将清空任务的到期日期。

Table 115. 更新一个任务 - 响应码

响应码	描述
200	代表任务已被更新。
404	代表未找到请求的任务。
409	代表请求的任务已经被并发同时更新。

成功响应体：参见 `runtime/tasks/{taskId}` 的响应。

15.7.5. 任务操作 Task actions

POST `runtime/tasks/{taskId}`

完成一个任务 - JSON 体:

```

1  {
2    "action" : "complete",
3    "variables" : []
4  }

```

完成任务。可以使用 `variables` 参数传递可选的变量数组。可以在[REST变量章节](#)找到关于变量格式的更多信息。请注意提供的变量范围将被忽略，变量将设置在父范围中，除非一个本地范围中存在该变量。这种情况下将会覆盖。

与 `TaskService.completeTask(taskId, variables)` 调用的行为相同。

申领一个任务 - JSON 体:

```

1  {
2    "action" : "claim",
3    "assignee" : "userWhoClaims"
4  }

```

使用给定办理人申领任务。如果办理人为 `null`，则取消任务的指派，可以重新申领。

代理一个任务 - JSON 体:

```

1  {
2    "action" : "delegate",
3    "assignee" : "userToDelegateTo"
4  }

```

将任务代理给给定办理人。办理人为必填。

解决一个任务 - JSON 体:

```

1  {
2    "action" : "resolve"
3  }

```

解决任务代理。任务指派回任务的属主（若有）。

Table 116. 任务操作 - 响应码

响应码	描述

响应码	必填	描述
200		代表已执行操作。
400		代表请求体包含非法值，或者当操作需要时没有提供办理人。
404		代表未找到请求的任务。
409		代表操作由于冲突无法执行。任务已被并发同时更新，或者在进行' claim '操作时，任务已被其他用户申领。

成功响应体：参见 `runtime/tasks/{taskId}` 的响应。

15.7.6. 删除一个任务 Delete a task

<code>DELETE runtime/tasks/{taskId}?cascadeHistory={cascadeHistory}&deleteReason={deleteReason}</code>
--

Table 117. 删除一个任务 - URL 参数

参数	必填	值	描述
taskId	是	String	要删除的任务的id。
cascadeHistory	否	Boolean	是否在删除任务时，删除历史任务实例（如果可用）。如果未设置，则默认值为false。
deleteReason	否	String	任务删除的原因。如果 <code>cascadeHistory</code> 值为true，则忽略本参数。

Table 118. 删除一个任务 - 响应码

响应码	描述
204	代表已找到并删除任务。响应体设置为空。
403	代表由于请求的任务是工作流的一部分，无法删除。
404	代表未找到请求的任务。

15.7.7. 获取一个任务的所有变量 Get all variables for a task

<code>GET runtime/tasks/{taskId}/variables?scope={scope}</code>

Table 119. 获取一个任务的所有变量 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取变量的任务的id。
scope	否	String	要返回的变量的范围。若值为' local '，则只返回任务本地变量。若值为' global '，则只返回任务的父执行树中的变量。若省略该参数，则本地与全局变量都会返回。

Table 120. 获取一个任务的所有变量 - 响应码

响应码	描述
200	代表已找到任务，并已返回请求的变量。
404	代表未找到请求的任务。

成功响应体:

```
1  [
2  {
3      "name" : "doubleTaskVar",
4      "scope" : "local",
5      "type" : "double",
6      "value" : 99.99
7  },
8  {
9      "name" : "stringProcVar",
10     "scope" : "global",
11     "type" : "string",
12     "value" : "This is a ProcVariable"
13  }
14 ]
15
16
17 ]
```

变量作为一个JSON数组返回。在通用[REST变量章节](#)中可以找到全部响应体的描述。

15.7.8. 从一个任务中获取一个变量 Get a variable from a task

```
GET runtime/tasks/{taskId}/variables/{variableName}?scope={scope}
```

Table 121. 从一个任务中获取一个变量 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取变量的任务的id。
variableName	是	String	要获取的变量的名字。
scope	否	String	要返回的变量的范围。若值为' local '，则只返回任务本地变量。若值为' global '，则只返回任务的父执行树中的变量。若省略该参数，如果存在本地变量就会返回，否则返回全局变量。

Table 122. 从一个任务中获取一个变量 - 响应码

响应码	描述
200	代表已找到任务，并已返回请求的变量。
404	代表未找到请求的任务，或者任务（在给定范围中）没有给定名字的变量。状态消息提供了额外信息。

成功响应体:

```
1  {
2      "name" : "myTaskVariable",
3      "scope" : "local",
4      "type" : "string",
5      "value" : "Hello my friend"
6  }
```

在通用[REST变量章节](#)中可以找到全部响应体的描述。

15.7.9. 获取一个变量的二进制数据 Get the binary data for a variable

```
GET runtime/tasks/{taskId}/variables/{variableName}/data?scope={scope}
```

Table 123. 获取一个变量的二进制数据 - URL 参数

参数	必填	值	描述

参数 taskId	必填 是	描述 String	描述 要获取变量的任务的id。
variableName	是	String	要获取数据的变量的名字。只能使用 <code>binary</code> 与 <code>serializable</code> 类型的变量。如果使用其它任何类型的变量，将返回 <code>404</code> 。
scope	否	String	要返回的变量的范围。若值为 <code>local</code> ，则只返回任务本地变量。若值为 <code>global</code> ，则只返回任务的父执行树中的变量。若省略该参数，如果存在本地变量就会返回，否则返回全局变量。

Table 124. 获取一个变量的二进制数据 - 响应码

响应码	描述
200	代表已找到任务，并已返回请求的变量。
404	代表未找到请求的任务，或者任务（在给定的范围中）没有给定名字的变量，或者变量没有可用的二进制流。状态消息提供了额外信息。

成功响应体：响应体包含了变量的二进制值。如果变量是 `binary` 类型，则不论变量的 `content` 或者请求的 `accept-type` 头是什么，响应的 `content-type` 都将设置为 `application/octet-stream`。如果变量是 `serializable` 类型，则 `content-type` 会使用 `application/x-java-serialized-object`。

15.7.10. 为一个任务创建一个新的变量 Create new variables on a task

POST runtime/tasks/{taskId}/variables

Table 125. 为一个任务创建一个新的变量 - URL 参数

参数	必填	值	描述
taskId	是	String	要创建新变量的任务的id。

创建简单（非二进制）变量的请求体：

<pre>1 [2 { 3 "name" : "myTaskVariable", 4 "scope" : "local", 5 "type" : "string", 6 "value" : "Hello my friend" 7 }, 8 { 9 } 10] 11]</pre>

请求体需要是一个数组，包含有一个或多个代表要创建的变量的JSON对象

- `name`: 变量需要的名字。
- `scope`: 创建变量的范围。如果省略，则使用 `local`。
- `type`: 创建的变量的类型。如果省略，则转换为原始JSON值的类型（`string`、`boolean`、`integer`或`double`）。
- `value`: 变量值。

可以在[REST变量章节](#)找到关于变量格式的更多信息。

成功响应体：

--

```

1  [
2  {
3      "name" : "myTaskVariable",
4      "scope" : "local",
5      "type" : "string",
6      "value" : "Hello my friend"
7  },
8  {
9
10 }
11 ]

```

Table 126. 为一个任务创建一个新的变量 - 响应码

响应码	描述
201	代表已创建变量，并已返回结果。
400	代表缺少要创建的变量名，或者尝试为独立任务（未关联至流程）创建 <code>global</code> 变量，或者请求中包含了空的变量数组，或者请求中没有包含变量数组。状态消息提供了额外信息。
404	代表未找到请求的任务。
409	代表任务中已有给定名字的变量。改用PUT方法更新任务变量。

15.7.11. 为一个任务创建一个新的二进制变量 Create a new binary variable on a task

```
POST runtime/tasks/{taskId}/variables
```

Table 127. 为一个任务创建一个新的二进制变量 - URL 参数

参数	必填	值	描述
taskId	是	String	要创建新变量的任务的id。

请求体：请求需要是`multipart/form-data`类型的。需要有唯一的`file-part`，包含变量的二进制值。另外，也可以使用下列额外的`form-fields`：

- `name`：变量需要的名字。
- `scope`：创建变量的范围。如果省略，则使用`local`。
- `type`：创建的变量的类型。如果省略，则使用`binary`，并且将请求中的二进制数据保存为字节数组。

成功响应体：

```

1  {
2      "name" : "binaryVariable",
3      "scope" : "local",
4      "type" : "binary",
5      "value" : null,
6      "valueUrl" : "http://.../runtime/tasks/123/variables/binaryVariable/data"
7  }

```

Table 128. 为一个任务创建一个新的二进制变量 - 响应码

响应码	描述
201	代表已创建变量，并已返回结果。
400	代表缺少要创建的变量名，或者尝试为独立任务（未关联至流程）创建 <code>global</code> 变量。状态消息提供了额外信息。
404	代表未找到请求的任务。
409	代表任务中已有给定名字的变量。改用PUT方法更新任务变量。
415	代表序列化数据中包含了一个运行Activiti引擎的JVM中不存在的类的对象，因此不能反序列化。

15.7.12. 为一个任务更新一个已有变量 Update an existing variable on a task

```
PUT runtime/tasks/{taskId}/variables/{variableName}
```

Table 129. 为一个任务更新一个已有变量 - URL 参数

参数	必填	值	描述
taskId	是	String	要更新变量的任务的id。
variableName	是	String	要更新的变量的名字。

更新简单（非二进制）变量的请求体：

```
1 {
2   "name" : "myTaskVariable",
3   "scope" : "local",
4   "type" : "string",
5   "value" : "Hello my friend"
6 }
```

请求体需要是一个数组，包含有一个或多个代表要创建的变量JSON对象

- **name**: 变量需要的名字。
- **scope**: 创建变量的范围。如果省略，则使用 **local**。
- **type**: 创建的变量的类型。如果省略，则转换为原始JSON值的类型（string、boolean、integer或double）。
- **value**: 变量值。

可以在[REST变量章节](#)找到关于变量格式的更多信息。

成功响应体：

```
1 {
2   "name" : "myTaskVariable",
3   "scope" : "local",
4   "type" : "string",
5   "value" : "Hello my friend"
6 }
```

Table 130. 为一个任务更新一个已有变量 - 响应码

响应码	描述
200	代表已更新变量，并已返回结果。
400	代表缺少要创建的变量名，或者尝试为独立任务（未关联至流程）创建 global 变量。状态消息提供了额外信息。
404	代表未找到请求的任务，或者任务在给定范围中没有给定名字的变量。状态消息提供了额外信息。

15.7.13. 为一个任务更新一个二进制变量 Updating a binary variable on a task

```
PUT runtime/tasks/{taskId}/variables/{variableName}
```

Table 131. 为一个任务更新一个二进制变量 - URL 参数

参数	必填	值	描述
taskId	是	String	要更新变量的任务的id。
variableName	是	String	要更新的变量的名字。

请求体：请求需要是 **multipart/form-data** 类型的。需要有唯一的 **file-part**，包含变量的二进制值。另外，也可以使用下列额外的 **form-fields**：

- **name**: 变量需要的名字。

- `scope`: 创建变量的范围。如果省略，则使用 `local`。
- `type`: 创建的变量的类型。如果省略，则使用 `binary`，并且将请求中的二进制数据保存为字节数组。

成功响应体：

```

1 {
2   "name" : "binaryVariable",
3   "scope" : "local",
4   "type" : "binary",
5   "value" : null,
6   "valueUrl" : "http://.../runtime/tasks/123/variables/binaryVariable/data"
7 }
```

Table 132. 为一个任务更新一个二进制变量 - 响应码

响应码	描述
200	代表已更新变量，并已返回结果。
400	代表缺少要创建的变量名，或者尝试为独立任务（未关联至流程）创建 <code>global</code> 变量。状态消息提供了额外信息。
404	代表未找到请求的任务，或者任务在给定范围中没有给定名字的变量。
415	代表序列化数据中包含了一个运行 <code>Activiti</code> 引擎的 JVM 中不存在的类的对象，因此不能反序列化。

15.7.14. 为一个任务删除一个变量 Delete a variable on a task

```
DELETE runtime/tasks/{taskId}/variables/{variableName}?scope={scope}
```

Table 133. 为一个任务删除一个变量 - URL 参数

参数	必填	值	描述
<code>taskId</code>	是	String	要删除的变量所在任务的id。
<code>variableName</code>	是	String	要删除的变量的名字。
<code>scope</code>	否	String	要删除的变量所在的范围。可以为 <code>local</code> 或 <code>global</code> 。若省略，则使用 <code>local</code> 。

Table 134. 为一个任务删除一个变量 - 响应码

响应码	描述
204	代表已找到并删除任务变量。响应体设置为空。
404	代表未找到请求的任务，或者任务在给定范围中没有给定名字的变量。状态消息提供了额外信息。

15.7.15. 为一个任务删除所有本地变量 Delete all local variables on a task

```
DELETE runtime/tasks/{taskId}/variables
```

Table 135. 为一个任务删除所有本地变量 - URL 参数

参数	必填	值	描述
<code>taskId</code>	是	String	要删除的变量所在任务的id。

Table 136. 为一个任务删除所有本地变量 - 响应码

响应码	描述

响应码	必填	描述
204		代表已删除所有本地任务变量。响应体设置为空。
404		代表未找到请求的任务。

15.7.16. 获取一个任务的所有身份关联 Get all identity links for a task

GET runtime/tasks/{taskId}/identitylinks
--

Table 137. 获取一个任务的所有身份关联 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取身份关联的任务的id。

Table 138. 获取一个任务的所有身份关联 - 响应码

响应码	描述
200	代表已找到任务，并已返回请求的身份关联。
404	代表未找到请求的任务。

成功响应体：

<pre> 1 [2 { 3 "userId" : "kermit", 4 "groupId" : null, 5 "type" : "candidate", 6 "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/users/kermit/candidate" 7 }, 8 { 9 "userId" : null, 10 "groupId" : "sales", 11 "type" : "candidate", 12 "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/groups/sales/candidate" 13 }, 14 ... 15] </pre>
--

15.7.17. 获取一个任务的组或用户的所有身份关联 Get all identitylinks for a task for either groups or users

GET runtime/tasks/{taskId}/identitylinks/users
GET runtime/tasks/{taskId}/identitylinks/groups

只返回目标为用户或组的身份关联。响应体及状态码与获取一个任务的完整身份关联列表时完全相同。

15.7.18. 获取一个任务的一个身份关联 Get a single identity link on a task

GET runtime/tasks/{taskId}/identitylinks/{family}/{identityId}/{type}

Table 139. 获取一个任务的一个身份关联 - URL 参数

参数	必填	值	描述
taskId	是	String	任务的id。
family	是	String	可以为 <code>groups</code> 或 <code>users</code> ，取决于身份目标的类型。
identityId	是	String	身份的id。
type	是	String	身份关联的类型。

Table 140. 获取一个任务的一个身份关联 - 响应码

--

响应码	必填	描述
200		代表已找到任务及身份关联，并已返回。
404		代表未找到请求的任务，或者任务中没有请求的身份关联。状态消息提供了额外信息。

成功响应体：

```
1 {
2   "userId" : null,
3   "groupId" : "sales",
4   "type" : "candidate",
5   "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/groups/sales/candidate"
6 }
```

15.7.19. 为一个任务创建一个身份关联 Create an identity link on a task

```
POST runtime/tasks/{taskId}/identitylinks
```

Table 141. 为一个任务创建一个身份关联 - URL 参数

参数	必填	值	描述
taskId	是	String	任务的id。

响应体（用户）：

```
1 {
2   "userId" : "kermit",
3   "type" : "candidate",
4 }
```

响应体（组）：

```
1 {
2   "groupId" : "sales",
3   "type" : "candidate",
4 }
```

Table 142. 为一个任务创建一个身份关联 - 响应码

响应码	描述
201	代表已找到任务，并已创建身份关联。
404	代表未找到任务，或任务中没有请求的身份关联。状态消息提供了额外信息。

成功响应体：

```
1 {
2   "userId" : null,
3   "groupId" : "sales",
4   "type" : "candidate",
5   "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/groups/sales/candidate"
6 }
```

15.7.20. 为一个任务删除一个身份关联 Delete an identity link on a task

```
DELETE runtime/tasks/{taskId}/identitylinks/{family}/{identityId}/{type}
```

Table 143. 为一个任务删除一个身份关联 - URL 参数

参数	必填	值	描述

参数名	必填	类型	描述
taskId	是	String	任务的id。
family	是	String	可以为 <code>groups</code> 或 <code>users</code> ，取决于身份目标的类型。
identityId	是	String	身份的id。
type	是	String	身份关联的类型。

Table 144. 为一个任务删除一个身份关联 - 响应码

响应码	描述
204	代表已找到任务与身份关联，并已删除身份关联。响应体设置为空。
404	代表未找到任务，或任务中没有请求的身份关联。状态消息提供了额外信息。

15.7.21. 为一个任务创建一个新备注 Create a new comment on a task

```
POST runtime/tasks/{taskId}/comments
```

Table 145. 为一个任务创建一个新备注 - URL 参数

参数	必填	值	描述
taskId	是	String	要创建备注的任务的id。

请求体:

```
1 {
2   "message" : "This is a comment on the task.",
3   "saveProcessInstanceId" : true
4 }
```

参数 `saveProcessInstanceId` 是可选的，如果为 `true`，则在备注中保存任务的流程实例id。

成功响应体:

```
1 {
2   "id" : "123",
3   "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
4   "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-
5 instances/100/comments/123",
6   "message" : "This is a comment on the task.",
7   "author" : "kermit",
8   "time" : "2014-07-13T13:13:52.232+08:00",
9   "taskId" : "101",
10  "processInstanceId" : "100"
11 }
```

Table 146. 为一个任务创建一个新备注 - 响应码

响应码	描述
201	代表已创建备注，并已返回结果。
400	代表请求中缺少备注。
404	代表未找到请求的任务。

15.7.22. 获取一个任务的所有备注 Get all comments on a task

```
GET runtime/tasks/{taskId}/comments
```


Table 147. 获取一个任务的所有备注 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取备注的任务的id。

成功响应体:

```

1  [
2  {
3      "id" : "123",
4      "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
5      "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-
6 instances/100/comments/123",
7      "message" : "This is a comment on the task.",
8      "author" : "kermit",
9      "time" : "2014-07-13T13:13:52.232+08:00",
10     "taskId" : "101",
11     "processInstanceId" : "100"
12 },
13 {
14     "id" : "456",
15     "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/456",
16     "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-
17 instances/100/comments/456",
18     "message" : "This is another comment on the task.",
19     "author" : "gonzo",
20     "time" : "2014-07-13T13:13:52.232+08:00",
21     "taskId" : "101",
22     "processInstanceId" : "100"
23 }
24 ]

```

Table 148. 获取一个任务的所有备注 - 响应码

响应码	描述
200	代表已找到任务，并已返回备注。
404	代表未找到请求的任务。

15.7.23. 获取一个任务的一个备注 Get a comment on a task

```
GET runtime/tasks/{taskId}/comments/{commentId}
```

Table 149. 获取一个任务的一个备注 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取备注的任务的id。
commentId	是	String	备注的id。

成功响应体:

```

1  {
2      "id" : "123",
3      "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
4      "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-
5 instances/100/comments/123",
6      "message" : "This is a comment on the task.",
7      "author" : "kermit",
8      "time" : "2014-07-13T13:13:52.232+08:00",
9      "taskId" : "101",
10     "processInstanceId" : "100"
11 }

```

Table 150. 获取一个任务的一个备注 - 响应码

响应码	描述
200	代表已找到任务与备注，并已返回备注。

响应码 404	必填	描述 代表未找到请求的任务，或任务没有给定ID的备注。
------------	----	--------------------------------

15.7.24. 为一个任务删除一个备注 Delete a comment on a task

DELETE runtime/tasks/{taskId}/comments/{commentId}
--

Table 151. 为一个任务删除一个备注 - URL 参数

参数	必填	值	描述
taskId	是	String	要删除备注的任务的id。
commentId	是	String	备注的id。

Table 152. 为一个任务删除一个备注 - 响应码

响应码	描述
204	代表已找到任务与备注，并已删除备注。响应体设置为空。
404	代表未找到请求的任务，或任务没有给定id的备注。

15.7.25. 获取一个任务的所有事件 Get all events for a task

GET runtime/tasks/{taskId}/events

Table 153. 获取一个任务的所有事件 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取事件的任务的id。

成功响应体：

<pre> 1 [2 { 3 "action" : "AddUserLink", 4 "id" : "4", 5 "message" : ["gonzo", "contributor"], 6 "taskUrl" : "http://localhost:8182/runtime/tasks/2", 7 "time" : "2013-05-17T11:50:50.000+0000", 8 "url" : "http://localhost:8182/runtime/tasks/2/events/4", 9 "userId" : null 10 } 11] 12] </pre>
--

Table 154. 获取一个任务的所有事件 - 响应码

响应码	描述
200	代表已找到任务，并已返回事件。
404	代表未找到请求的任务。

15.7.26. 获取一个任务的一个事件 Get an event on a task

GET runtime/tasks/{taskId}/events/{eventId}

Table 155. 获取一个任务的一个事件 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取事件的任务的id。

响应码 eventid	必填 是	描述 String	描述 事件的id。
----------------	---------	--------------	--------------

成功响应体:

```

1 {
2   "action" : "AddUserLink",
3   "id" : "4",
4   "message" : [ "gonzo", "contributor" ],
5   "taskUrl" : "http://localhost:8182/runtime/tasks/2",
6   "time" : "2013-05-17T11:50:50.000+0000",
7   "url" : "http://localhost:8182/runtime/tasks/2/events/4",
8   "userId" : null
9 }
```

Table 156. 获取一个任务的一个事件 - 响应码

响应码	描述
200	代表已找到任务与事件，并已返回事件。
404	代表未找到请求的任务，或者任务中没有给定ID的事件。

15.7.27. 为一个任务创建一个新的附件，带有一个指向外部资源的链接 **Create a new attachment on a task, containing a link to an external resource**

```
POST runtime/tasks/{taskId}/attachments
```

Table 157. 为一个任务创建一个新的附件，带有一个指向外部资源的链接 - URL 参数

参数	必填	值	描述
taskId	是	String	要创建附件的任务的id。

请求体:

```

1 {
2   "name":"Simple attachment",
3   "description":"Simple attachment description",
4   "type":"simpleType",
5   "externalUrl":"http://activiti.org"
6 }
```

要创建一个新的附件，只有附件名是必填的。

成功响应体:

```

1 {
2   "id":"3",
3   "url":"http://localhost:8182/runtime/tasks/2/attachments/3",
4   "name":"Simple attachment",
5   "description":"Simple attachment description",
6   "type":"simpleType",
7   "taskUrl":"http://localhost:8182/runtime/tasks/2",
8   "processInstanceUrl":null,
9   "externalUrl":"http://activiti.org",
10  "contentUrl":null
11 }
```

Table 158. 为一个任务创建一个新的附件，带有一个指向外部资源的链接 - 响应码

响应码	描述
201	代表已创建附件，并已返回结果。
400	代表请求中缺少附件名。
404	代表未找到请求的任务。

15.7.28. 为一个任务创建一个新的附件，带有附加文件 Create a new attachment on a task, with an attached file

```
POST runtime/tasks/{taskId}/attachments
```

Table 159. 为一个任务创建一个新的附件，带有附加文件 - URL 参数

参数	必填	值	描述
taskId	是	String	要创建附件的任务的id。

请求体：请求需要是 `multipart/form-data` 类型的。需要有唯一的file-part，包含变量的二进制值。另外，也可以使用下列额外的 form-fields：

- **name**：附件需要的名字。
- **description**：附件的描述，可选。
- **type**：附件的类型，可选。支持任意字符串或合法的HTTP content-type。

成功响应体：

```
1 {
2   "id": "5",
3   "url": "http://localhost:8182/runtime/tasks/2/attachments/5",
4   "name": "Binary attachment",
5   "description": "Binary attachment description",
6   "type": "binaryType",
7   "taskUrl": "http://localhost:8182/runtime/tasks/2",
8   "processInstanceUrl": null,
9   "externalUrl": null,
10  "contentUrl": "http://localhost:8182/runtime/tasks/2/attachments/5/content"
11 }
```

Table 160. 为一个任务创建一个新的附件，带有附加文件 - 响应码

响应码	描述
201	代表已创建附件，并已返回结果。
400	代表请求中缺少附件名，或者请求中没有文件。错误消息中包含了额外信息。
404	代表未找到请求的任务。

15.7.29. 获取一个任务的所有附件 Get all attachments on a task

```
GET runtime/tasks/{taskId}/attachments
```

Table 161. 获取一个任务的所有附件 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取附件的任务的id。

成功响应体：

```
1 [
2   {
3     "id": "3",
4     "url": "http://localhost:8182/runtime/tasks/2/attachments/3",
5     "name": "Simple attachment",
6     "description": "Simple attachment description",
7     "type": "simpleType",
8     "taskUrl": "http://localhost:8182/runtime/tasks/2",
9     "processInstanceUrl": null,
10    "externalUrl": "http://activiti.org",
11    "contentUrl": null
12  },
13  {
14    "id": "5",
15    "url": "http://localhost:8182/runtime/tasks/2/attachments/5",
16    "name": "Binary attachment",
17    "description": "Binary attachment description",
```

```

18     "type": "binaryType",
19     "taskUrl": "http://localhost:8182/runtime/tasks/2",
20     "processInstanceUrl": null,
21     "externalUrl": null,
22     "contentUrl": "http://localhost:8182/runtime/tasks/2/attachments/5/content"
23 }
24 ]

```

Table 162. 获取一个任务的所有附件 - 响应码

响应码	描述
200	代表已找到任务，并已返回附件。
404	代表未找到请求的任务。

15.7.30. 获取一个任务的一个附件 Get an attachment on a task

```
GET runtime/tasks/{taskId}/attachments/{attachmentId}
```

Table 163. 获取一个任务的一个附件 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取附件的任务的id。
attachmentId	是	String	附件的id。

成功响应体：

```

1 {
2   "id": "5",
3   "url": "http://localhost:8182/runtime/tasks/2/attachments/5",
4   "name": "Binary attachment",
5   "description": "Binary attachment description",
6   "type": "binaryType",
7   "taskUrl": "http://localhost:8182/runtime/tasks/2",
8   "processInstanceUrl": null,
9   "externalUrl": null,
10  "contentUrl": "http://localhost:8182/runtime/tasks/2/attachments/5/content"
11 }

```

- **externalUrl - contentUrl:** 如果附件为外部资源，则 **externalUrl** 中包含外部内容的URL。如果附件内容保存在Activiti引擎中，则 **contentUrl** 中将包含可以流式获取二进制内容的URL。
- **type:** 可以是任意值。如果包含了合法格式的media-type（例如application/xml、text/plain），则二进制内容的HTTP响应的content-type将设置为给定值。

Table 164. 获取一个任务的一个附件 - 响应码

响应码	描述
200	代表已找到任务与附件，并已返回附件。
404	代表未找到请求的任务，或任务中没有给定ID的附件。

15.7.31. 获取一个附件的内容 Get the content for an attachment

```
GET runtime/tasks/{taskId}/attachment/{attachmentId}/content
```

Table 165. 获取一个附件的内容 - URL 参数

参数	必填	值	描述
taskId	是	String	要获取附件数据的任务的id。
attachmentId	是	String	附件的id。如果附件指向外部URL而不是保存在Activiti内部，则将返回 404 。

Table 166. 获取一个附件的内容 - 响应码

响应码	必填	描述
200		代表已找到任务与附件，并已返回请求的内容。
404		代表未找到请求的任务，或者任务中没有给定id的附件，或者附件没有可用的二进制流。状态消息提供了额外信息。

成功响应体：

响应体包含了二进制内容。默认情况下，响应的content-type设置为 `application/octet-stream`，除非附件类型包含了合法的content-type。

15.7.32. 为一个任务删除一个附件 Delete an attachment on a task

DELETE runtime/tasks/{taskId}/attachments/{attachmentId}
--

Table 167. 为一个任务删除一个附件 - URL 参数

参数	必填	值	描述
taskId	是	String	要删除附件的任务的id。
attachmentId	是	String	附件的id。

Table 168. 为一个任务删除一个附件 - 响应码

响应码	描述
204	代表已找到任务与附件，并已删除附件。响应体设置为空。
404	代表未找到请求的任务，或者任务中没有给定ID的附件。

15.8. 历史 History

15.8.1. 获取一个历史流程实例 Get a historic process instance

GET history/historic-process-instances/{processInstanceId}
--

Table 169. 获取一个历史流程实例 - 响应码

响应码	描述
200	代表可以找到历史流程实例。
404	代表找不到历史流程实例。

成功响应体：

<pre>1 { 2 "data": [3 { 4 "id" : "5", 5 "businessKey" : "myKey", 6 "processDefinitionId" : "oneTaskProcess%3A1%3A4", 7 "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4", 8 "startTime" : "2013-04-17T10:17:43.902+0000", 9 "endTime" : "2013-04-18T14:06:32.715+0000", 10 "durationInMillis" : 86400056, 11 "startUserId" : "kermit", 12 "startActivityId" : "startEvent", 13 "endActivityId" : "endEvent", 14 "deleteReason" : null, 15 "superProcessInstanceId" : "3", 16 "url" : "http://localhost:8182/history/historic-process-instances/5", 17 "variables": null, 18 "tenantId":null 19 } 20], 21 "total": 1, 22 "start": 0,</pre>

```
23 | "sort": "name",
24 | "order": "asc",
25 | "size": 1
26 | }
```

15.8.2. 历史流程实例的列表 List of historic process instances

GET history/historic-process-instances

Table 170. 历史流程实例的列表 - URL 参数

参数	必填	值	描述
processInstanceId	否	String	历史流程实例的id。
processDefinitionKey	否	String	历史流程实例的流程定义key。
processDefinitionId	否	String	历史流程实例的流程定义id。
businessKey	否	String	历史流程实例的businessKey。
involvedUser	否	String	历史流程实例的参与用户。
finished	否	Boolean	历史流程实例是否已结束。
superProcessInstanceId	否	String	可选的历史流程实例的父流程的id
excludeSubprocesses	否	Boolean	只返回不是子流程的历史流程实例。
finishedAfter	否	Date	只返回在该日期之后完成的历史流程实例。
finishedBefore	否	Date	只返回在该日期之前完成的历史流程实例。
startedAfter	否	Date	只返回在该日期之后启动的历史流程实例。
startedBefore	否	Date	只返回在该日期之前启动的历史流程实例。
startedBy	否	String	只返回由该用户启动的历史流程实例。
includeProcessVariables	否	Boolean	是否也返回历史流程实例变量。
tenantId	否	String	只返回给定tenantId的实例。
tenantIdLike	否	String	只返回tenantId like给定值的实例。
withoutTenantId	否	Boolean	如果值为true，则只返回未设置tenantId的实例。如果值为false，则忽略withoutTenantId参数。

可以在这个URL中使用通用分页与排序查询参数。

Table 171. 历史流程实例的列表 - 响应码

响应码	描述
200	代表可以查询历史流程实例。

响应码 400	描述 代表传递的某个参数格式错误。状态描述中包含了额外信息。
------------	-----------------------------------

成功响应体:

```

1  {
2    "data": [
3      {
4        "id" : "5",
5        "businessKey" : "myKey",
6        "processDefinitionId" : "oneTaskProcess%3A1%3A4",
7        "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
8        "startTime" : "2013-04-17T10:17:43.902+0000",
9        "endTime" : "2013-04-18T14:06:32.715+0000",
10       "durationInMillis" : 86400056,
11       "startUserId" : "kermit",
12       "startActivityId" : "startEvent",
13       "endActivityId" : "endEvent",
14       "deleteReason" : null,
15       "superProcessInstanceId" : "3",
16       "url" : "http://localhost:8182/history/historic-process-instances/5",
17       "variables": [
18         {
19           "name": "test",
20           "variableScope": "local",
21           "value": "myTest"
22         }
23       ],
24       "tenantId":null
25     }
26   ],
27   "total": 1,
28   "start": 0,
29   "sort": "name",
30   "order": "asc",
31   "size": 1
32 }

```

15.8.3. 查询历史流程实例 Query for historic process instances

POST query/historic-process-instances

请求体:

```

1  {
2    "processDefinitionId" : "oneTaskProcess%3A1%3A4",
3
4
5    "variables" : [
6      {
7        "name" : "myVariable",
8        "value" : 1234,
9        "operation" : "equals",
10       "type" : "long"
11     }
12   ]
13 }

```

支持的JSON参数字段与[获取历史流程实例集合](#)的参数一模一样，但是使用JSON体参数而不是URL参数，可以进行更高级的查询，也可以避免请求URI太长的错误。另外，查询可以通过流程变量进行过滤。`variables`参数是一个JSON数组，包含[这里描述](#)的格式的对象。

Table 172. 查询历史流程实例 - 响应码

响应码	描述
200	代表请求成功，并已返回实例。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体:

```

1  {
2    "data": [

```



```

3      {
4        "id" : "5",
5        "businessKey" : "myKey",
6        "processDefinitionId" : "oneTaskProcess%3A1%3A4",
7        "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
8        "startTime" : "2013-04-17T10:17:43.902+0000",
9        "endTime" : "2013-04-18T14:06:32.715+0000",
10       "durationInMillis" : 86400056,
11       "startUserId" : "kermit",
12       "startActivityId" : "startEvent",
13       "endActivityId" : "endEvent",
14       "deleteReason" : null,
15       "superProcessInstanceId" : "3",
16       "url" : "http://localhost:8182/history/historic-process-instances/5",
17       "variables": [
18         {
19           "name": "test",
20           "variableScope": "local",
21           "value": "myTest"
22         }
23       ],
24       "tenantId":null
25     }
26   ],
27   "total": 1,
28   "start": 0,
29   "sort": "name",
30   "order": "asc",
31   "size": 1
32 }

```

15.8.4. 删除一个历史流程实例 Delete a historic process instance

```
DELETE history/historic-process-instances/{processInstanceId}
```

Table 173. 响应码

响应码	描述
200	代表已删除该历史流程实例。
404	代表未找到该历史流程实例。

15.8.5. 获取一个历史流程实例的身份关联 Get the identity links of a historic process instance

```
GET history/historic-process-instance/{processInstanceId}/identitylinks
```

Table 174. 响应码

响应码	描述
200	代表请求成功，并已返回身份关联。
404	代表未找到流程实例。

成功响应体:

```

1  [
2    {
3      "type" : "participant",
4      "userId" : "kermit",
5      "groupId" : null,
6      "taskId" : null,
7      "taskUrl" : null,
8      "processInstanceId" : "5",
9      "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/5"
10   }
11 ]

```

15.8.6. 获取一个历史流程实例变量的二进制数据 Get the binary data for a historic process instance variable

```
GET history/historic-process-instances/{processInstanceId}/variables/{variableName}/data
```

Table 175. 获取一个历史流程实例变量的二进制数据 - 响应码

响应码	必填	描述
200		代表已找到流程实例，并已返回请求的变量数据。
404		代表未找到请求的流程实例，或者流程实例中没有给定名字的变量，或者给定名字的变量没有可用的二进制流。状态消息提供了额外信息。

成功响应体：

响应体包含了变量的二进制值。如果变量是 `binary` 类型，则不论变量的 `content` 或者请求的 `accept-type` 头是什么，响应的 `content-type` 都将设置为 `application/octet-stream`。如果变量是 `serializable` 类型，则 `content-type` 会使用 `application/x-java-serialized-object`。

15.8.7. 为一个历史流程实例创建一个新的备注 Create a new comment on a historic process instance

```
POST history/historic-process-instances/{processInstanceId}/comments
```

Table 176. 为一个历史流程实例创建一个新的备注 - URL 参数

参数	必填	值	描述
<code>processInstanceId</code>	是	String	要创建备注的流程实例的id。

请求体：

```
1 {
2   "message" : "This is a comment.",
3   "saveProcessInstanceId" : true
4 }
```

参数 `saveProcessInstanceId` 是可选的，如果为 `true`，则在备注中保存任务的流程实例id。

成功响应体：

```
1 {
2   "id" : "123",
3   "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
4   "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-
5 instances/100/comments/123",
6   "message" : "This is a comment on the task.",
7   "author" : "kermit",
8   "time" : "2014-07-13T13:13:52.232+08:00",
9   "taskId" : "101",
10  "processInstanceId" : "100"
}
```

Table 177. 为一个历史流程实例创建一个新的备注 - 响应码

响应码	描述
201	代表已创建备注，并已返回结果。
400	代表请求中缺少备注。
404	代表未找到请求的历史流程实例。

15.8.8. 获取一个历史流程实例的所有备注 Get all comments on a historic process instance

```
GET history/historic-process-instances/{processInstanceId}/comments
```

Table 178. 获取一个历史流程实例的所有备注 - URL 参数

参数	必填	值	描述
<code>processInstanceId</code>	是	String	要获取备注的流程实例的id。

成功响应体:

```
1 [
2   {
3     "id" : "123",
4     "processInstanceId" : "http://localhost:8081/activiti-rest/service/history/historic-process-
5 instances/100/comments/123",
6     "message" : "This is a comment on the task.",
7     "author" : "kermit",
8     "time" : "2014-07-13T13:13:52.232+08:00",
9     "processInstanceId" : "100"
10  },
11  {
12    "id" : "456",
13    "processInstanceId" : "http://localhost:8081/activiti-rest/service/history/historic-process-
14 instances/100/comments/456",
15    "message" : "This is another comment.",
16    "author" : "gonzo",
17    "time" : "2014-07-14T15:16:52.232+08:00",
18    "processInstanceId" : "100"
19  }
20 ]
```

Table 179. 获取一个历史流程实例的所有备注 - 响应码

响应码	描述
200	代表已找到流程实例，并已返回备注。
404	代表未找到请求的任务。

15.8.9. 获取一个历史流程实例的一个备注 Get a comment on a historic process instance

```
GET history/historic-process-instances/{processInstanceId}/comments/{commentId}
```

Table 180. 获取一个历史流程实例的一个备注 - URL 参数

参数	必填	值	描述
processInstanceId	是	String	要获取备注的历史流程实例的id。
commentId	是	String	备注的id。

成功响应体:

```
1 {
2   "id" : "123",
3   "processInstanceId" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100/comments/456",
4   "message" : "This is another comment.",
5   "author" : "gonzo",
6   "time" : "2014-07-14T15:16:52.232+08:00",
7   "processInstanceId" : "100"
8 }
```

Table 181. 获取一个历史流程实例的一个备注 - 响应码

响应码	描述
200	代表已找到历史流程实例与备注，并已返回备注。
404	代表未找到请求的历史流程实例，或者历史流程实例中没有给定id的备注。

15.8.10. 从一个历史流程实例中删除一个备注 Delete a comment on a historic process instance

```
DELETE history/historic-process-instances/{processInstanceId}/comments/{commentId}
```

Table 182. 从一个历史流程实例中删除一个备注 - URL 参数

--

参数	必填	值	描述
响应码 processInstanceId	必填 是	描述 String	描述 要删除备注的历史流程实例的id。
参数	必填	值	描述
commentId	是	String	备注的id。

Table 183. 从一个历史流程实例中删除一个备注 - 响应码

响应码	描述
204	代表已找到历史流程实例与备注，并已删除备注。响应体设置为空。
404	代表未找到请求的流程实例，或历史流程实例中没有给定ID的备注

15.8.11. 获取一个历史任务实例 Get a single historic task instance

GET history/historic-task-instances/{taskId}
--

Table 184. 获取一个历史任务实例 - 响应码

响应码	描述
200	代表可以找到历史任务实例。
404	代表不能找到历史任务实例。

成功响应体：

<pre> 1 { 2 "id" : "5", 3 "processDefinitionId" : "oneTaskProcess%3A1%3A4", 4 "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4", 5 "processInstanceId" : "3", 6 "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3", 7 "executionId" : "4", 8 "name" : "My task name", 9 "description" : "My task description", 10 "deleteReason" : null, 11 "owner" : "kermit", 12 "assignee" : "fozzie", 13 "startTime" : "2013-04-17T10:17:43.902+0000", 14 "endTime" : "2013-04-18T14:06:32.715+0000", 15 "durationInMillis" : 86400056, 16 "workTimeInMillis" : 234890, 17 "claimTime" : "2013-04-18T11:01:54.715+0000", 18 "taskDefinitionKey" : "taskKey", 19 "formKey" : null, 20 "priority" : 50, 21 "dueDate" : "2013-04-20T12:11:13.134+0000", 22 "parentTaskId" : null, 23 "url" : "http://localhost:8182/history/historic-task-instances/5", 24 "variables" : null, 25 "tenantId":null 26 }</pre>
--

15.8.12. 获取历史任务实例 Get historic task instances

GET history/historic-task-instances

Table 185. 获取历史任务实例 - URL 参数

参数	必填	值	描述
taskId	否	String	历史任务实例的id。
processInstanceId	否	String	历史任务实例所在流程实例的id。

参数 processDefinitionKey	必填 否	值 String	描述 历史任务实例所在流程定义key。
processDefinitionKeyLike	否	String	历史任务实例所在流程，流程定义key匹配给定值。
processDefinitionId	否	String	历史任务实例所在流程定义的定义id。
processDefinitionName	否	String	历史任务实例所在流程定义名。
processDefinitionNameLike	否	String	历史任务实例所在流程，流程定义名匹配给定值。
processBusinessKey	否	String	历史任务实例所在流程实例businessKey。
processBusinessKeyLike	否	String	历史任务实例所在流程，流程实例businessKey匹配给定值。
executionId	否	String	历史任务实例的所在执行的id。
taskDefinitionKey	否	String	流程任务部分的任务定义key。
taskName	否	String	历史任务实例的任务名。
taskNameLike	否	String	历史任务实例的任务名'like'操作。
taskDescription	否	String	历史任务实例的任务描述。
taskDescriptionLike	否	String	历史任务实例的任务描述'like'操作。
taskDefinitionKey	否	String	历史任务实例从流程定义得到的任务标识符。
taskDeleteReason	否	String	历史任务实例的任务删除原因。
taskDeleteReasonLike	否	String	历史任务实例的任务删除原因'like'操作。
taskAssignee	否	String	历史任务实例的办理人。
taskAssigneeLike	否	String	历史任务实例的办理人'like'操作。
taskOwner	否	String	历史任务实例的属主。
taskOwnerLike	否	String	历史任务实例的属主'like'操作。
taskInvolvedUser	否	String	历史任务实例的一个相关用户。
taskPriority	否	String	历史任务实例的优先级。
finished	否	Boolean	是否已完成历史任务实例。
processFinished	否	Boolean	是否为已完成流程实例的历史任务实例。

参数 parentTaskId	必填 否	描述 String	描述 历史任务实例的可选父任务的id。
dueDate	否	Date	只返回到期日期等于该日期的历史任务实例。
dueDateAfter	否	Date	只返回到期日期在该日期之后的历史任务实例。
dueDateBefore	否	Date	只返回到期日期在该日期之前的历史任务实例。
withoutDueDate	否	Boolean	只返回未设置到期日期的历史任务实例。如果值为 false ，则忽略本参数。
taskCompletedOn	否	Date	只返回在该日期完成的历史任务实例。
taskCompletedAfter	否	Date	只返回在该日期之后完成的历史任务实例。
taskCompletedBefore	否	Date	只返回在该日期之前完成的历史任务实例。
taskCreatedOn	否	Date	只返回在该日期创建的历史任务实例。
taskCreatedBefore	否	Date	只返回在该日期之前创建的历史任务实例。
taskCreatedAfter	否	Date	只返回在该日期之后创建的历史任务实例。
includeTaskLocalVariables	否	Boolean	是否同时返回历史任务实例的本地变量。
includeProcessVariables	否	Boolean	是否同时返回历史任务实例的全局变量。
tenantId	否	String	只返回给定tenantId的历史任务实例。
tenantIdLike	否	String	只返回tenantId like给定值的历史任务实例。
withoutTenantId	否	Boolean	如果值为 true ，则只返回未设置tenantId的历史任务实例。如果值为 false ，则忽略 withoutTenantId 参数。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 186. 获取历史任务实例 - 响应码

响应码	描述
200	代表可以查询历史任务实例。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
```

```

4      "id" : "5",
5      "processDefinitionId" : "oneTaskProcess%3A1%3A4",
6      "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
7      "processInstanceId" : "3",
8      "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
9      "executionId" : "4",
10     "name" : "My task name",
11     "description" : "My task description",
12     "deleteReason" : null,
13     "owner" : "kermit",
14     "assignee" : "fozzie",
15     "startTime" : "2013-04-17T10:17:43.902+0000",
16     "endTime" : "2013-04-18T14:06:32.715+0000",
17     "durationInMillis" : 86400056,
18     "workTimeInMillis" : 234890,
19     "claimTime" : "2013-04-18T11:01:54.715+0000",
20     "taskDefinitionKey" : "taskKey",
21     "formKey" : null,
22     "priority" : 50,
23     "dueDate" : "2013-04-20T12:11:13.134+0000",
24     "parentTaskId" : null,
25     "url" : "http://localhost:8182/history/historic-task-instances/5",
26     "taskVariables": [
27       {
28         "name": "test",
29         "variableScope": "local",
30         "value": "myTest"
31       }
32     ],
33     "processVariables": [
34       {
35         "name": "processTest",
36         "variableScope": "global",
37         "value": "myProcessTest"
38       }
39     ],
40     "tenantId":null
41   }
42 ],
43 "total": 1,
44 "start": 0,
45 "sort": "name",
46 "order": "asc",
47 "size": 1
48 }

```

15.8.13. 查询历史任务实例 Query for historic task instances

POST query/historic-task-instances

查询历史任务实例 - 请求体:

```

1  {
2    "processDefinitionId" : "oneTaskProcess%3A1%3A4",
3    ...
4
5    "variables" : [
6      {
7        "name" : "myVariable",
8        "value" : 1234,
9        "operation" : "equals",
10       "type" : "long"
11      }
12    ]
13  }

```

支持的JSON参数字段与[获取历史任务实例集合](#)的参数一模一样，但是使用JSON体参数而不是URL参数，可以进行更高级的查询，也可以避免请求URI太长的错误。另外，查询可以通过流程变量进行过滤。`variables`参数是一个JSON数组，包含[这里描述](#)的格式的对象。

Table 187. 查询历史任务实例 - 响应码

响应码	描述
200	代表请求成功，并已返回任务。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体:

```
1 {
2   "data": [
3     {
4       "id" : "5",
5       "processDefinitionId" : "oneTaskProcess%3A1%3A4",
6       "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
7       "processInstanceId" : "3",
8       "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
9       "executionId" : "4",
10      "name" : "My task name",
11      "description" : "My task description",
12      "deleteReason" : null,
13      "owner" : "kermit",
14      "assignee" : "fozzie",
15      "startTime" : "2013-04-17T10:17:43.902+0000",
16      "endTime" : "2013-04-18T14:06:32.715+0000",
17      "durationInMillis" : 86400056,
18      "workTimeInMillis" : 234890,
19      "claimTime" : "2013-04-18T11:01:54.715+0000",
20      "taskDefinitionKey" : "taskKey",
21      "formKey" : null,
22      "priority" : 50,
23      "dueDate" : "2013-04-20T12:11:13.134+0000",
24      "parentTaskId" : null,
25      "url" : "http://localhost:8182/history/historic-task-instances/5",
26      "taskVariables": [
27        {
28          "name": "test",
29          "variableScope": "local",
30          "value": "myTest"
31        }
32      ],
33      "processVariables": [
34        {
35          "name": "processTest",
36          "variableScope": "global",
37          "value": "myProcessTest"
38        }
39      ],
40      "tenantId":null
41    }
42  ],
43  "total": 1,
44  "start": 0,
45  "sort": "name",
46  "order": "asc",
47  "size": 1
48 }
```

15.8.14. 删除一个历史任务实例 Delete a historic task instance

```
DELETE history/historic-task-instances/{taskId}
```

Table 188. 响应码

响应码	描述
200	代表已删除历史任务实例。
404	代表未找到该历史任务实例。

15.8.15. 获取一个历史任务实例的身份关联 Get the identity links of a historic task instance

```
GET history/historic-task-instance/{taskId}/identitylinks
```

Table 189. 响应码

响应码	描述
200	代表请求成功，并已返回身份关联。
404	代表未找到该任务实例。

成功响应体:


```

1  [
2  {
3    "type" : "assignee",
4    "userId" : "kermi",
5    "groupId" : null,
6    "taskId" : "6",
7    "taskUrl" : "http://localhost:8182/history/historic-task-instances/5",
8    "processInstanceId" : null,
9    "processInstanceUrl" : null
10 }
11 ]

```

15.8.16. 获取一个历史流程实例变量的二进制数据 Get the binary data for a historic task instance variable

```
GET history/historic-task-instances/{taskId}/variables/{variableName}/data
```

Table 190. 获取一个历史流程实例变量的二进制数据 - 响应码

响应码	描述
200	代表已找到任务实例，并已返回请求的变量数据。
404	代表未找到请求的任务实例，或流程实例中没有给定名字的变量，或者变量没有可用的二进制流。状态消息提供了额外信息。

成功响应体：

响应体包含了变量的二进制值。如果变量是 `binary` 类型，则不论变量的 `content` 或者请求的 `accept-type` 头是什么，响应的 `content-type` 都将设置为 `application/octet-stream`。如果变量是 `serializable` 类型，则 `content-type` 会使用 `application/x-java-serialized-object`。

15.8.17. 获取历史活动实例 Get historic activity instances

```
GET history/historic-activity-instances
```

Table 191. 获取历史活动实例 - URL 参数

参数	必填	值	描述
activityId	否	String	活动实例的id。
activityInstanceId	否	String	历史活动实例的id。
activityName	否	String	历史活动实例的名字。
activityType	否	String	历史活动实例的元素类型。
executionId	否	String	历史活动实例的执行的id。
finished	否	Boolean	是否已完成的历史活动实例。
taskAssignee	否	String	历史活动实例的办理人。
processInstanceId	否	String	历史活动实例的流程实例的id。
processDefinitionId	否	String	历史活动实例的流程定义的id。
tenantId	否	String	只返回给定tenantId的实例。
tenantIdLike	否	String	只返回tenantId like给定值的实例。
withoutTenantId	否	Boolean	如果值为 <code>true</code> ，则只返回未设置tenantId的实例。如果值为 <code>false</code> ，则忽略 <code>withoutTenantId</code> 参数。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 192. 获取历史活动实例 - 响应码

响应码	描述
200	代表可以查询历史活动实例。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "id" : "5",
5       "activityId" : "4",
6       "activityName" : "My user task",
7       "activityType" : "userTask",
8       "processDefinitionId" : "oneTaskProcess%3A1%3A4",
9       "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
10      "processInstanceId" : "3",
11      "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
12      "executionId" : "4",
13      "taskId" : "4",
14      "calledProcessInstanceId" : null,
15      "assignee" : "fozzie",
16      "startTime" : "2013-04-17T10:17:43.902+0000",
17      "endTime" : "2013-04-18T14:06:32.715+0000",
18      "durationInMillis" : 86400056,
19      "tenantId":null
20    }
21  ],
22  "total": 1,
23  "start": 0,
24  "sort": "name",
25  "order": "asc",
26  "size": 1
27 }
```

15.8.18. 查询历史活动实例 Query for historic activity instances

```
POST query/historic-activity-instances
```

请求体：

```
1 {
2   "processDefinitionId" : "oneTaskProcess%3A1%3A4"
3 }
```

支持的JSON参数字段与[获取历史任务实例集合（原文如此）](#)的参数一模一样，但是使用JSON体参数而不是URL参数，可以进行更高级的查询，也可以避免请求URI太长的错误。

Table 193. 查询历史活动实例 - 响应码

响应码	描述
200	代表请求成功，并已返回活动。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "id" : "5",
5       "activityId" : "4",
6       "activityName" : "My user task",
7       "activityType" : "userTask",
8       "processDefinitionId" : "oneTaskProcess%3A1%3A4",
9       "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
10      "processInstanceId" : "3",
11      "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
12      "executionId" : "4",
```

```

13     "taskId" : "4",
14     "calledProcessInstanceId" : null,
15     "assignee" : "fozzie",
16     "startTime" : "2013-04-17T10:17:43.902+0000",
17     "endTime" : "2013-04-18T14:06:32.715+0000",
18     "durationInMillis" : 86400056,
19     "tenantId":null
20   }
21 ],
22 "total": 1,
23 "start": 0,
24 "sort": "name",
25 "order": "asc",
26 "size": 1
27 }

```

15.8.19. 历史变量实例的列表 List of historic variable instances

GET history/historic-variable-instances

Table 194. 历史变量实例的列表 - URL 参数

参数	必填	值	描述
processInstanceId	否	String	历史变量实例的流程实例的id。
taskId	否	String	历史变量实例的任务的id。
excludeTaskVariables	否	Boolean	是否从结果中排除任务变量。
variableName	否	String	历史变量实例的变量名。
variableNameLike	否	String	历史变量实例的变量名'like'操作。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 195. 历史变量实例的列表 - 响应码

响应码	描述
200	代表可以查询历史变量实例。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体:

```

1  {
2    "data": [
3      {
4        "id" : "14",
5        "processInstanceId" : "5",
6        "processInstanceId" : "http://localhost:8182/history/historic-process-instances/5",
7        "taskId" : "6",
8        "variable" : {
9          "name" : "myVariable",
10         "variableScope": "global",
11         "value" : "test"
12       }
13     }
14   ],
15   "total": 1,
16   "start": 0,
17   "sort": "name",
18   "order": "asc",
19   "size": 1
20 }

```

15.8.20. 查询历史变量实例 Query for historic variable instances

POST query/historic-variable-instances

请求体:

```
1 {
2   "processDefinitionId" : "oneTaskProcess%3A1%3A4",
3   ...
4
5   "variables" : [
6     {
7       "name" : "myVariable",
8       "value" : 1234,
9       "operation" : "equals",
10      "type" : "long"
11    }
12  ]
13 }
```

支持的JSON参数字段与[获取历史变量实例集合](#)的参数一模一样，但是使用JSON体参数而不是URL参数，可以进行更高级的查询，也可以避免请求URI太长的错误。另外，查询可以通过流程变量进行过滤。`variables`参数是一个JSON数组，包含[这里描述](#)的格式的对象。

Table 196. 查询历史变量实例 - 响应码

响应码	描述
200	代表请求成功，并已返回变量。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体:

```
1 {
2   "data": [
3     {
4       "id" : "14",
5       "processInstanceId" : "5",
6       "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/5",
7       "taskId" : "6",
8       "variable" : {
9         "name" : "myVariable",
10        "variableScope": "global",
11        "value" : "test"
12      }
13    }
14  ],
15  "total": 1,
16  "start": 0,
17  "sort": "name",
18  "order": "asc",
19  "size": 1
20 }
```

15.8.21. 获取历史任务实例变量的二进制数据 [Get the binary data for a historic task instance variable](#)

```
GET history/historic-variable-instances/{varInstanceId}/data
```

Table 197. 获取历史任务实例变量的二进制数据 - 响应码

响应码	描述
200	代表已找到变量实例，并已返回请求的变量数据。
404	代表未找到请求的变量实例，或变量实例中没有给定名字的变量（原文如此），或变量没有可用的二进制流。状态消息提供了额外信息。

成功响应体:

响应体包含了变量的二进制值。如果变量是 `binary` 类型，则不论变量的content或者请求的accept-type头是什么，响应的content-type都将设置为 `application/octet-stream`。如果变量是 `serializable` 类型，则content-type会使用 `application/x-java-serialized-object`。

15.8.22. 获取历史详情 [Get historic detail](#)

GET history/historic-detail

Table 198. 获取历史详情 - URL 参数

参数	必填	值	描述
id	否	String	历史详情的id。
processInstanceId	否	String	历史详情的流程实例的id。
executionId	否	String	历史详情的执行的id。
activityInstanceId	否	String	历史详情的活动实例的id。
taskId	否	String	历史详情的任务的id。
selectOnlyFormProperties	否	Boolean	是否只在结果中返回表单参数。
selectOnlyVariableUpdates	否	Boolean	是否只在结果中返回变量更新。

可以在这个URL中使用通用[分页与排序查询参数](#)。

Table 199. 获取历史详情 - 响应码

响应码	描述
200	代表可以查询历史详情。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体：

<pre>1 { 2 "data": [3 { 4 "id" : "26", 5 "processInstanceId" : "5", 6 "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/5", 7 "executionId" : "6", 8 "activityInstanceId": "10", 9 "taskId" : "6", 10 "taskUrl" : "http://localhost:8182/history/historic-task-instances/6", 11 "time" : "2013-04-17T10:17:43.902+0000", 12 "detailType" : "variableUpdate", 13 "revision" : 2, 14 "variable" : { 15 "name" : "myVariable", 16 "variableScope": "global", 17 "value" : "test" 18 }, 19 "propertyId": null, 20 "propertyValue": null 21 } 22], 23 "total": 1, 24 "start": 0, 25 "sort": "name", 26 "order": "asc", 27 "size": 1 28 }</pre>
--

15.8.23. 查询历史详情 Query for historic details

POST query/historic-detail

请求体：

<pre>{ "processInstanceId" : "5", }</pre>

支持的JSON参数字段与[获取历史详情集合](#)的参数一模一样，但是使用JSON体参数而不是URL参数，可以进行更高级的查询，也可以避免请求URI太长的错误。

Table 200. 查询历史详情 - 响应码

响应码	描述
200	代表请求成功，并已返回历史详情。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

成功响应体：

```
1 {
2   "data": [
3     {
4       "id" : "26",
5       "processInstanceId" : "5",
6       "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/5",
7       "executionId" : "6",
8       "activityInstanceId": "10",
9       "taskId" : "6",
10      "taskUrl" : "http://localhost:8182/history/historic-task-instances/6",
11      "time" : "2013-04-17T10:17:43.902+0000",
12      "detailType" : "variableUpdate",
13      "revision" : 2,
14      "variable" : {
15        "name" : "myVariable",
16        "variableScope": "global",
17        "value" : "test"
18      },
19      "propertyId" : null,
20      "propertyValue" : null
21    }
22  ],
23  "total": 1,
24  "start": 0,
25  "sort": "name",
26  "order": "asc",
27  "size": 1
28 }
```

15.8.24. 获取历史详情变量的二进制数据 Get the binary data for a historic detail variable

GET history/historic-detail/{detailId}/data

Table 201. 获取历史详情变量的二进制数据 - 响应码

响应码	描述
200	代表已找到历史详情实例，并已返回请求的变量数据。
404	代表未找到请求的历史详情实例，或历史详情实例中没有给定名字的变量，或变量没有可用的二进制流。状态消息提供了额外信息。

成功响应体：

响应体包含了变量的二进制值。如果变量是 `binary` 类型，则不论变量的 `content` 或者请求的 `accept-type` 头是什么，响应的 `content-type` 都将设置为 `application/octet-stream`。如果变量是 `serializable` 类型，则 `content-type` 会使用 `application/x-java-serialized-object`。

15.9. 表单 Forms

15.9.1. 获取表单数据 Get form data

GET form/form-data

Table 202. 获取表单数据 - URL 参数

参数	必填	值	描述
----	----	---	----

响应码 taskId	必填 是（如果没有 processDefinitionId	描述 String	描述 需要获取表单数据的相应任务的id。
processDefinitionId	是（如果没有taskId）	String	需要获取启动事件表单数据的相应流程定义的id。

Table 203. 获取表单数据 - 响应码

响应码	描述
200	代表可以查询表单数据。
404	代表未找到该表单数据。

成功响应体:

```
1 {
2   "data": [
3     {
4       "formKey" : null,
5       "deploymentId" : "2",
6       "processDefinitionId" : "3",
7       "processDefinitionUrl" : "http://localhost:8182/repository/process-definition/3",
8       "taskId" : "6",
9       "taskUrl" : "http://localhost:8182/runtime/task/6",
10      "formProperties" : [
11        {
12          "id" : "room",
13          "name" : "Room",
14          "type" : "string",
15          "value" : null,
16          "readable" : true,
17          "writable" : true,
18          "required" : true,
19          "datePattern" : null,
20          "enumValues" : [
21            {
22              "id" : "normal",
23              "name" : "Normal bed"
24            },
25            {
26              "id" : "kingsize",
27              "name" : "Kingsize bed"
28            }
29          ]
30        }
31      ]
32    }
33  ],
34  "total": 1,
35  "start": 0,
36  "sort": "name",
37  "order": "asc",
38  "size": 1
39 }
```

15.9.2. 提交任务表单数据 Submit task form data

POST form/form-data

任务表单的请求体:

```
1 {
2   "taskId" : "5",
3   "properties" : [
4     {
5       "id" : "room",
6       "value" : "normal"
7     }
8   ]
9 }
```

启动事件表单的请求体:

```

1  {
2    "processDefinitionId" : "5",
3    "businessKey" : "myKey",
4    "properties" : [
5      {
6        "id" : "room",
7        "value" : "normal"
8      }
9    ]
10 }

```

Table 204. 提交任务表单数据 - 响应码

响应码	描述
200	代表请求成功，并已提交表单数据。
400	代表传递的某个参数格式错误。状态描述中包含了额外信息。

启动事件表单数据的成功响应体（任务表单数据没有响应）：

```

1  {
2    "id" : "5",
3    "url" : "http://localhost:8182/history/historic-process-instances/5",
4    "businessKey" : "myKey",
5    "suspended": false,
6    "processDefinitionId" : "3",
7    "processDefinitionUrl" : "http://localhost:8182/repository/process-definition/3",
8    "activityId" : "myTask"
9  }

```

15.10. 数据库表 Database tables

15.10.1. 表的列表 List of tables

```
GET management/tables
```

Table 205. 表的列表 - 响应码

响应码	描述
200	代表请求成功。

成功响应体：

```

1  [
2    {
3      "name": "ACT_RU_VARIABLE",
4      "url": "http://localhost:8182/management/tables/ACT_RU_VARIABLE",
5      "count": 4528
6    },
7    {
8      "name": "ACT_RU_EVENT_SUBSCR",
9      "url": "http://localhost:8182/management/tables/ACT_RU_EVENT_SUBSCR",
10     "count": 3
11   }
12 ]
13

```

15.10.2. 获取一个表 Get a single table

```
GET management/tables/{tableName}
```

Table 206. 获取一个表 - URL 参数

参数	必填	值	描述
tableName	是	String	要获取的表的名字。

成功响应体：


```
1 {
2     "name": "ACT_RE_PROCDEF",
3     "url": "http://localhost:8182/management/tables/ACT_RE_PROCDEF",
4     "count": 60
5 }
```

Table 207. 获取一个表 - 响应码

响应码	描述
200	代表存在该表，并已返回表记录数。
404	代表不存在请求的表。

15.10.3. 获取一个表的列信息 Get column info for a single table

```
GET management/tables/{tableName}/columns
```

Table 208. 获取一个表的列信息 - URL 参数

参数	必填	值	描述
tableName	是	String	要获取的表的名字。

成功响应体:

```
1 {
2     "tableName": "ACT_RU_VARIABLE",
3     "columnNames": [
4         "ID_",
5         "REV_",
6         "TYPE_",
7         "NAME_"
8     ],
9
10    ],
11    "columnTypes": [
12        "VARCHAR",
13        "INTEGER",
14        "VARCHAR",
15        "VARCHAR"
16    ],
17
18    ]
19 }
```

Table 209. 获取一个表的列信息 - 响应码

响应码	描述
200	代表存在该表，并已返表的列信息。
404	代表不存在请求的表。

15.10.4. 获取一个表的行数据 Get row data for a single table

```
GET management/tables/{tableName}/data
```

Table 210. 获取一个表的行数据 - URL 参数

参数	必填	值	描述
tableName	是	String	要获取的表的名字。

Table 211. 获取一个表的行数据 - URL 查询参数

参数	必填	值	描述

响应码 start	必填 否	描述 integer	描述 要读取的第一行的index。默认为0。
size	否	Integer	要读取的行数，从start开始。默认为10。
orderAscendingColumn	否	String	结果行要用来排序的列名，顺序。
orderDescendingColumn	否	String	结果行要用来排序的列名，逆序。

成功响应体：

```
1 {
2   "total":3,
3   "start":0,
4   "sort":null,
5   "order":null,
6   "size":3,
7
8   "data":[
9     {
10      "TASK_ID_":"2",
11      "NAME_":"var1",
12      "REV_":1,
13      "TEXT_":"123",
14      "LONG_":123,
15      "ID_":"3",
16      "TYPE_":"integer"
17    }
18  ]
19
20
21 }
22
23 }
```

Table 212. 获取一个表的行数据 - 响应码

响应码	描述
200	代表存在该表，并已返回表的行数据。
404	代表不存在请求的表。

15.11. 引擎 Engine

15.11.1. 获取引擎参数 Get engine properties

```
GET management/properties
```

返回引擎内部使用的参数的只读视图。

成功响应体：

```
1 {
2   "next.dbid": "101",
3   "schema.history": "create(5.15)",
4   "schema.version": "5.15"
5 }
```

Table 213. 获取引擎参数 - 响应码

响应码	描述
200	代表已返回参数。

15.11.2. 获取引擎信息 Get engine info

GET management/engine			
响应码	请求头	响应头	响应体
返回本REST服务使用的引擎的只读视图。			
成功响应体：			
<pre>1 { 2 "name":"default", 3 "version":"5.15", 4 "resourceUrl":"file://activiti/activiti.cfg.xml", 5 "exception":null 6 }</pre>			

Table 214. 获取引擎信息 - 响应码

响应码	描述
200	代表已返回引擎信息。

15.12. 运行时 Runtime

15.12.1. 已接收到信号事件 Signal event received

POST runtime/signals

通知引擎，已经接收到一个信号事件，而不直接关联到特定的执行。

JSON体：

<pre>1 { 2 "signalName": "My Signal", 3 "tenantId" : "execute", 4 "async": true, 5 "variables": [6 {"name": "testVar", "value": "This is a string"} 7] 8 } 9 }</pre>
--

Table 215. 已接收到信号事件 - JSON体参数

参数	描述	必填
signalName	信号的名字	是
tenantId	需要处理信号事件的租户的ID	否
async	如果值为 true ，则会异步处理信号。返回码将为 202 - 已接受 ，代表请求已接受，但还未执行。如果值为 false ，则会立即处理信号，在执行成功完成后才将返回结果（ 200 - OK ）。如果省略，默认值为 false 。	否
variables	作为载荷与信号一同传递的变量（以通用变量格式）的数组。如果 async 设置为 true 则不能使用，否则结果为错误。	否

成功响应体：

Table 216. 已接收到信号事件 - 响应码

响应码	描述
200	代表已处理信号，且没有发生错误。
202	代表已将信号放入作业队列，可以开始执行。

响应码	必填	描述
400		未处理信号。缺少信号名，或变量与异步一同使用。响应体中包含错误的额外信息。

15.13. 作业 Jobs

15.13.1. 获取一个作业 Get a single job

GET management/jobs/{jobId}

Table 217. 获取一个作业 - URL 参数

参数	必填	值	描述
jobId	是	String	要获取的作业的id。

成功响应体：

<pre>1 { 2 "id": "8", 3 "url": "http://localhost:8182/management/jobs/8", 4 "processInstanceId": "5", 5 "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5", 6 "processDefinitionId": "timerProcess:1:4", 7 "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/timerProcess%3A1%3A4", 8 "executionId": "7", 9 "executionUrl": "http://localhost:8182/runtime/executions/7", 10 "retries": 3, 11 "exceptionMessage": null, 12 "dueDate": "2013-06-04T22:05:05.474+0000", 13 "tenantId": null 14 }</pre>

Table 218. 获取一个作业 - 响应码

响应码	描述
200	代表存在该作业，并已返回。
404	代表不存在请求的作业。

15.13.2. 删除一个作业 Delete a job

DELETE management/jobs/{jobId}

Table 219. 删除一个作业 - URL 参数

参数	必填	值	描述
jobId	是	String	要删除的作业的id。

Table 220. 删除一个作业 - 响应码

响应码	描述
204	代表已找到并已删除作业。响应体设置为空。
404	代表未找到请求的作业。

15.13.3. 执行一个作业 Execute a single job

POST management/jobs/{jobId}

JSON体：

{

```
1  "action" : "execute"
2  }
3
```

Table 221. 执行一个作业 - JSON体参数

参数	描述	必填
action	要进行的操作，只支持execute。	是

Table 222. 执行一个作业 - 响应码

响应码	描述
204	代表已执行该作业。响应体设置为空。
404	代表未找到请求的作业。
500	代表执行作业时发生异常。状态描述中包含了错误的额外细节。如果需要，之后可以获取完整的异常栈。

15.13.4. 获取一个作业的异常栈 Get the exception stacktrace for a job

```
GET management/jobs/{jobId}/exception-stacktrace
```

Table 223. 获取一个作业的异常栈 - URL 参数

参数	描述	必填
jobId	要获取异常栈的作业的id。	是

Table 224. 获取一个作业的异常栈 - 响应码

响应码	描述
200	代表已找到请求的作业，并已返回异常栈。响应包含原始的异常栈，并总是用text/plain的Content-type。
404	代表未找到请求的作业，或作业没有异常栈。状态描述中包含了错误的额外信息。。

15.13.5. 获取作业的列表 Get a list of jobs

```
GET management/jobs
```

Table 225. 获取作业的列表 - URL 查询参数

参数	描述	类型
id	只返回给定id的作业	String
processInstanceId	只返回给定id的流程中的作业	String
executionId	只返回给定id的执行中的作业	String
processDefinitionId	只返回给定流程定义的作业	String
withRetriesLeft	如果值为true，则只返回还能重试的作业。如果值为false，则忽略本参数。	Boolean
executable	如果值为true，则只返回可执行的作业。如果值为false，则忽略本参数。	Boolean

响应码	描述	类型
timersOnly	如果值为 <code>true</code> ，则只返回定时器作业。 如果值为 <code>false</code> ，则忽略本参数。不能与 <code>'messagesOnly'</code> 一起使用	Boolean
messagesOnly	如果值为 <code>true</code> ，则只返回消息作业。如果值为 <code>false</code> ，则忽略本参数。不能与 <code>'timersOnly'</code> 一起使用	Boolean
withException	如果值为 <code>true</code> ，则只返回执行时发生了异常的作业。如果值为 <code>false</code> ，则忽略本参数。	Boolean
dueBefore	只返回将在给定日期前执行的作业。使用本参数不会返回没有执行日期的作业。	Date
dueAfter	只返回将在给定日期后执行的作业。使用本参数不会返回没有执行日期的作业。	Date
exceptionMessage	只返回带有给定异常消息的作业	String
tenantId	只返回给定 <code>tenantId</code> 的作业。	String
tenantIdLike	只返回 <code>tenantId</code> like 给定值的作业。	String
withoutTenantId	如果值为 <code>true</code> ，则只返回未设置 <code>tenantId</code> 的作业。如果值为 <code>false</code> ，则忽略 <code>withoutTenantId</code> 参数。	Boolean
sort	用于排序结果的字段，需要为 <code>id</code> ， <code>dueDate</code> ， <code>executionId</code> ， <code>processInstanceId</code> ， <code>retries</code> 或 <code>tenantId</code> 中的一个。	String

可以在这个URL中使用通用[分页与排序查询参数](#)。

成功响应体：

```
1 {
2   "data":[
3     {
4       "id":"13",
5       "url":"http://localhost:8182/management/jobs/13",
6       "processInstanceId":"5",
7       "processInstanceUrl":"http://localhost:8182/runtime/process-instances/5",
8       "processDefinitionId":"timerProcess:1:4",
9       "processDefinitionUrl":"http://localhost:8182/repository/process-definitions/timerProcess%3A1%3A4",
10      "executionId":"12",
11      "executionUrl":"http://localhost:8182/runtime/executions/12",
12      "retries":0,
13      "exceptionMessage":"Can't find scripting engine for 'unexistinglanguage'",
14      "dueDate":"2013-06-07T10:00:24.653+0000",
15      "tenantId":null
16    }
17  ],
18  "total":2,
19  "start":0,
20  "sort":"id",
21  "order":"asc",
22  "size":2
23 }
```

Table 226. 获取作业的列表 - 响应码

响应码	描述
200	代表已返回请求的作业。

响应码	必填	描述	类型	描述
400				代表在url查询参数中使用了非法的值，或 <code>'messagesOnly'</code> 与 <code>'timersOnly'</code> 同时用作参数。状态描述中包含了错误的额外信息。

15.14. 用户 Users

15.14.1. 获取一个用户 Get a single user

GET identity/users/{userId}

Table 227. 获取一个用户 - URL 参数

参数	必填	值	描述
userId	是	String	要获取的用户的id。

成功响应体：

<pre>1 { 2 "id": "testuser", 3 "firstName": "Fred", 4 "lastName": "McDonald", 5 "url": "http://localhost:8182/identity/users/testuser", 6 "email": "no-reply@activiti.org" 7 }</pre>
--

Table 228. 获取一个用户 - 响应码

响应码	描述
200	代表存在该用户，并已返回。
404	代表不存在请求的用户。

15.14.2. 获取用户的列表 Get a list of users

GET identity/users

Table 229. 获取用户的列表 - URL 查询参数

参数	描述	类型
id	只返回给定id的用户	String
firstName	只返回给定firstname的用户	String
lastName	只返回给定lastname的用户	String
email	只返回给定email的用户	String
firstNameLike	只返回firstname like给定值的用户。使用%作为通配符。	String
lastNameLike	只返回lastname like给定值的用户。使用%作为通配符。	String
emailLike	只返回email like给定值的用户。使用%作为通配符。	String
memberOfGroup	只返回给定组的成员用户	String
potentialStarter	只返回给定id的流程定义的潜在启动者用户。	String

响应码	描述	排序	类型
sort	用于排序结果的字段，需要为id，	描述	String
	firstName, lastname或email中的一个。		

可以在这个URL中使用通用[分页与排序查询参数](#)。

成功响应体：

```
1 {
2   "data":[
3     {
4       "id":"anotherUser",
5       "firstName":"Tijs",
6       "lastName":"Barrez",
7       "url":"http://localhost:8182/identity/users/anotherUser",
8       "email":"no-reply@alfresco.org"
9     },
10    {
11      "id":"kermit",
12      "firstName":"Kermit",
13      "lastName":"the Frog",
14      "url":"http://localhost:8182/identity/users/kermit",
15      "email":null
16    },
17    {
18      "id":"testuser",
19      "firstName":"Fred",
20      "lastName":"McDonald",
21      "url":"http://localhost:8182/identity/users/testuser",
22      "email":"no-reply@activiti.org"
23    }
24  ],
25  "total":3,
26  "start":0,
27  "sort":"id",
28  "order":"asc",
29  "size":3
30 }
```

Table 230. 获取用户的列表 - 响应码

响应码	描述
200	代表已返回请求的用户。

15.14.3. 更新一个用户 Update a user

```
PUT identity/users/{userId}
```

JSON体：

```
1 {
2   "firstName":"Tijs",
3   "lastName":"Barrez",
4   "email":"no-reply@alfresco.org",
5   "password":"pass123"
6 }
```

所有的请求值都是可选的。例如，可以只在请求体的JSON对象中包含'firstName'属性，则只更新用户的firstName，而不影响其它任何字段。若明确包含了一个属性，并设置为null，则用户值将更新为null。例如：`{"firstName" : null}`将清空用户的firstName。

Table 231. 更新一个用户 - 响应码

响应码	描述
200	代表已更新该用户。
404	代表未找到请求的用户。
409	代表请求的用户已被并发同时更新。

成功响应体：参见 `identity/users/{userId}` 的响应。

15.14.4. 创建一个用户 Create a user

响应码	描述
POST identity/users	

JSON体：

<pre>{ "id": "tjjs", "firstName": "Tjjs", "lastName": "Barrez", "email": "no-reply@alfresco.org", "password": "pass123" }</pre>

Table 232. 创建一个用户 - 响应码

响应码	描述
201	代表已创建该用户。
400	代表缺少用户的id。

成功响应体：参见 `identity/users/{userId}` 的响应。

15.14.5. 删除一个用户 Delete a user

DELETE identity/users/{userId}

Table 233. 删除一个用户 - URL 参数

参数	必填	值	描述
userId	是	String	要删除的用户的id。

Table 234. 删除一个用户 - 响应码

响应码	描述
204	代表已找到并删除该用户。响应体设置为空。
404	代表未找到请求的用户。

15.14.6. 获取一个用户的图片 Get a user's picture

GET identity/users/{userId}/picture

Table 235. 获取一个用户的图片 - URL 参数

参数	必填	值	描述
userId	是	String	要获取图片的用户的id。

响应体：

响应体包含原始图片数据，代表用户的图片。响应的Content-type与创建图片时设置的mimeType有关。

Table 236. 获取一个用户的图片 - 响应码

响应码	描述
200	代表已找到用户，并已在响应体中返回了图片。
404	代表未找到请求的用户，或用户没有设置图片。状态描述中包含了错误的额外信息。

15.14.7. 更新一个用户的图片 Updating a user's picture

```
GET identity/users/{userId}/picture
```

Table 237. 更新一个用户的图片 - URL 参数

参数	必填	值	描述
userId	是	String	要更新图片的用户的id。

请求体:

请求体: 请求需要是 `multipart/form-data` 类型的。需要有唯一的file-part, 包含变量的二进制值。另外, 也可以使用下列额外的 form-fields:

- `contentType`: 可选的上传图片的mime-type。如果省略, 则默认使用 `image/jpeg` 作为该图片的mime-type。

Table 238. 更新一个用户的图片 - 响应码

响应码	描述
200	代表已找到该用户, 并已更新图片。响应体设置为空。
404	代表未找到请求的用户。

15.14.8. 列表一个用户的信息 List a user's info

```
PUT identity/users/{userId}/info
```

Table 239. 列表一个用户的信息 - URL 参数

参数	必填	值	描述
userId	是	String	要获取信息的用户的id。

响应体:

```
1  [
2    {
3      "key": "key1",
4      "url": "http://localhost:8182/identity/users/testuser/info/key1"
5    },
6    {
7      "key": "key2",
8      "url": "http://localhost:8182/identity/users/testuser/info/key2"
9    }
10 ]
```

Table 240. 列表一个用户的信息 - 响应码

响应码	描述
200	代表已找到用户, 并已返回信息 (key与url) 的列表。
404	代表未找到请求的用户。

15.14.9. 获取一条用户信息 Get a user's info

```
GET identity/users/{userId}/info/{key}
```

Table 241. 获取一条用户信息 - URL 参数

参数	必填	值	描述
userId	是	String	要获取信息的用户的id。
key	是	String	要获取的用户信息的key。

响应体:

```
1 {
2   "key": "key1",
3   "value": "Value 1",
4   "url": "http://localhost:8182/identity/users/testuser/info/key1"
5 }
```

Table 242. 获取一条用户信息 - 响应码

响应码	描述
200	代表已找到用户，该用户也有给定key的信息。
404	代表未找到请求的用户，或用户没有给定key的信息。状态描述中包含了关于错误的额外信息。

15.14.10. 更新一条用户信息 Update a user's info

```
PUT identity/users/{userId}/info/{key}
```

Table 243. 更新一条用户信息 - URL 参数

参数	必填	值	描述
userId	是	String	要更新信息的用户的id。
key	是	String	要更新的用户信息的key。

请求体:

```
1 {
2   "value": "The updated value"
3 }
```

响应体:

```
1 {
2   "key": "key1",
3   "value": "The updated value",
4   "url": "http://localhost:8182/identity/users/testuser/info/key1"
5 }
```

Table 244. 更新一条用户信息 - 响应码

响应码	描述
200	代表已找到用户，并已更新信息。
400	代表请求体中缺少value。
404	代表未找到请求的用户，或用户没有给定key的信息。状态描述中包含了关于错误的额外信息。

15.14.11. 创建一条新的用户信息记录 Create a new user's info entry

```
POST identity/users/{userId}/info
```

Table 245. 创建一条新的用户信息记录 - URL 参数

参数	必填	值	描述
userId	是	String	要创建信息的用户的id。

请求体:

```
|
```

```
1 {
2   "key": "key1",
3   "value": "The value"
4 }
```

响应体:

```
1 {
2   "key": "key1",
3   "value": "The value",
4   "url": "http://localhost:8182/identity/users/testuser/info/key1"
5 }
```

Table 246. 创建一条新的用户信息记录 - 响应码

响应码	描述
201	代表已找到用户，并已创建信息。
400	代表请求体中缺少key或value。状态描述中包含了关于错误的额外信息。
404	代表未找到请求的用户。
409	代表用户已有给定key的信息记录，改用更新资源实例（PUT）。

15.14.12. 删除一条用户信息 Delete a user's info

```
DELETE identity/users/{userId}/info/{key}
```

Table 247. 删除一条用户信息 - URL 参数

参数	必填	值	描述
userId	是	String	要删除信息的用户的id。
key	是	String	要删除的用户信息的key。

Table 248. 删除一条用户信息 - 响应码

响应码	描述
204	代表已找到用户，并已删除给定key的信息。响应体设置为空。
404	代表未找到请求的用户，或用户没有给定key的信息。状态描述中包含了关于错误的额外信息。

15.15. 组 Groups

15.15.1. 获取一个组 Get a single group

```
GET identity/groups/{groupId}
```

Table 249. 获取一个组 - URL 参数

参数	必填	值	描述
groupId	是	String	要获取的组的id。

成功响应体:

```
1 {
2   "id": "testgroup",
3   "url": "http://localhost:8182/identity/groups/testgroup",
4   "name": "Test group",
5   "type": "Test type"
6 }
```

Table 250. 获取一个组 - 响应码

响应码	描述	描述	类型
200		代表存在并已返回该组。	
404		代表不存在请求的组。	

15.15.2. 获取组的列表 Get a list of groups

GET identity/groups

Table 251. 获取组的列表 - URL 查询参数

参数	描述	类型
id	只返回给定id的组	String
name	只返回给定名字的组	String
type	只返回给定类型的组	String
nameLike	只返回名字like给定值的组。使用%作为通配符。	String
member	只返回有给定用户名作为成员的组。	String
potentialStarter	只返回给定id的流程定义的潜在启动者组。	String
sort	用于排序结果的字段，需要为id, name或type中的一个。	String

可以在这个URL中使用通用分页与排序查询参数。

成功响应体：

<pre>1 { 2 "data":[3 { 4 "id":"testgroup", 5 "url":"http://localhost:8182/identity/groups/testgroup", 6 "name":"Test group", 7 "type":"Test type" 8 } 9], 10 "total":3, 11 "start":0, 12 "sort":"id", 13 "order":"asc", 14 "size":3 15 }</pre>

Table 252. 获取组的列表 - 响应码

响应码	描述
200	代表已返回请求的组。

15.15.3. 更新一个组 Update a group

PUT identity/groups/{groupId}

JSON体：

<pre>1 { 2 "name":"Test group", 3 "type":"Test type" 4 }</pre>
--

所有的请求值都是可选的。例如，可以只在请求体的JSON对象中包含'name'属性，则只更新组的名字，而不影响其它任何字段。若明确包含了一个属性，并设置为null，则组值将更新为null。

响应码	描述	描述
-----	----	----

Table 253. 更新一个组 - 响应码

响应码	描述
200	代表已更新组。
404	代表未找到请求的组。
409	代表请求的组已被并发同时更新。

成功响应体：参见 `identity/groups/{groupId}` 的响应。

15.15.4. 创建一个组 Create a group

POST <code>identity/groups</code>

JSON体：

1	{
2	" id ":"testgroup",
3	" name ":"Test group",
4	" type ":"Test type"
5	}

Table 254. 创建一个组 - 响应码

响应码	描述
201	代表已创建组。
400	代表缺少组的id。

成功响应体：参见 `identity/groups/{groupId}` 的响应。

15.15.5. 删除一个组 Delete a group

DELETE <code>identity/groups/{groupId}</code>

Table 255. 删除一个组 - URL 参数

参数	必填	值	描述
groupId	是	String	要删除的组的id。

Table 256. 删除一个组 - 响应码

响应码	描述
204	代表已找到并已删除组。响应体设置为空。
404	代表未找到请求的组。

15.15.6. 获取一个组中的成员 Get members in a group

`identity/groups/members` 不能使用GET。使用 `identity/users?memberOfGroup=sales` URL 获取特定组中的所有用户。

15.15.7. 为一个组添加一个成员 Add a member to a group

POST <code>identity/groups/{groupId}/members</code>

Table 257. 为一个组添加一个成员 - URL 参数

参数	必填	值	描述
groupId	是	String	要添加成员的组的id。

JSON体:

```
1 {
2   "userId": "kermit"
3 }
```

Table 258. 为一个组添加一个成员 - 响应码

响应码	描述
201	代表已找到组，并已添加成员。
404	代表请求体中没有包含userId。
404	代表未找到请求的组。
409	代表请求的用户已经是组的成员。

响应体:

```
1 {
2   "userId": "kermit",
3   "groupId": "sales",
4   "url": "http://localhost:8182/identity/groups/sales/members/kermit"
5 }
```

15.15.8. 从一个组中删除一个成员 Delete a member from a group

```
DELETE identity/groups/{groupId}/members/{userId}
```

Table 259. 从一个组中删除一个成员 - URL 参数

参数	必填	值	描述
groupId	是	String	要移除成员的组的id。
userId	是	String	要移除的用户的id。

Table 260. 从一个组中删除一个成员 - 响应码

响应码	描述
204	代表已找到组，并已删除成员。响应体设置为空。
404	代表未找到请求的组，或用户不是组的成员。状态描述中包含了关于错误的额外信息。

响应体:

```
1 {
2   "userId": "kermit",
3   "groupId": "sales",
4   "url": "http://localhost:8182/identity/groups/sales/members/kermit"
5 }
```

16. CDI集成 CDI integration

activiti-cid模块平衡了Activiti的可配置性与cid的可扩展性。activiti-cdi最突出的特点是:

- 支持@BusinessProcessScoped bean（生命周期绑定至流程实例的Cdi bean），
- 用于从流程中解析Cdi bean（包括EJB）的自定义EI解析器，
- 使用注解对流程实例进行声明式控制，
- Activiti关联cdi事件总线，
- 可以与Java EE，Java SE，以及Spring一起工作，
- 支持单元测试。

```

1 <dependency>
2   <groupId>org.activiti</groupId>
3   <artifactId>activiti-cdi</artifactId>
4   <version>5.x</version>
5 </dependency>

```

16.1. 设置activiti-cdi (Setting up activiti-cdi)

Activiti cdi可以在不同环境中安装。在这个章节我们主要浏览配置选项。

16.1.1. 查找流程引擎 Looking up a Process Engine

cid扩展需要能访问流程引擎，因此会在运行时查找`org.activiti.cdi.spi.ProcessEngineLookup`接口的实现。cdi模块提供了默认的为`org.activiti.cdi.impl.LocalProcessEngineLookup`的实现，使用`ProcessEngines-Utility`类查找流程引擎，默认配置下使用`ProcessEngines#NAME_DEFAULT`查找流程引擎。需要扩展这个类并设置一个名字。请注意：在classpath中需要有`activiti.cfg.xml`配置文件。

Activiti cdi使用`java.util.ServiceLoader` SPI解析`org.activiti.cdi.spi.ProcessEngineLookup`实例。为了提供接口的自定义实现，需要在部署中添加名为`META-INF/services/org.activiti.cdi.spi.ProcessEngineLookup`的纯文本文件，在其中指定实现的全限定类名。



如果不提供自定义的`org.activiti.cdi.spi.ProcessEngineLookup`实现，则Activiti会使用默认的`LocalProcessEngineLookup`实现。在这种情况下，只需要在classpath中提供`activiti.cfg.xml`（参见下一章节）即可。

16.1.2. 配置流程引擎 Configuring the Process Engine

配置方式取决于选用的流程引擎查找策略（上一章节）。这里，我们关注与`LocalProcessEngineLookup`一起使用的可用配置选项，需要在classpath中提供一个Spring `activiti.cfg.xml`文件。

Activiti根据底层事务管理策略的不同，提供不同的`ProcessEngineConfiguration`实现。`activiti-cdi`模块不关注事务，也就意味着可以使用任何事务管理策略（甚至是Spring抽象事务）。按照约定，cdi模块提供两个自定义的`ProcessEngineConfiguration`实现：

- `org.activiti.cdi.CdiJtaProcessEngineConfiguration`：`activiti JtaProcessEngineConfiguration`的子类，可用于Activiti使用JTA管理事务的情况
- `org.activiti.cdi.CdiStandaloneProcessEngineConfiguration`：`StandaloneProcessEngineConfiguration`的子类，可用于Activiti使用简单JDBC事务的情况。

下面是一个JBoss 7的`activiti.cfg.xml`文件示例：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5 http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- 查找JTA事务管理器 Lookup the JTA-Transaction manager -->
8     <bean id="transactionManager" class="org.springframework.jndi.JndiObjectFactoryBean">
9         <property name="jndiName" value="java:jboss/TransactionManager"></property>
10        <property name="resourceRef" value="true" />
11    </bean>
12
13    <!-- 流程定义配置 process engine configuration -->
14    <bean id="processEngineConfiguration"
15          class="org.activiti.cdi.CdiJtaProcessEngineConfiguration">
16        <!-- 查找默认的Jboss数据源 Lookup the default Jboss datasource -->
17        <property name="dataSourceJndiName" value="java:jboss/datasources/ExampleDS" />
18        <property name="databaseType" value="h2" />
19        <property name="transactionManager" ref="transactionManager" />
20        <!-- 使用外部管理事务 using externally managed transactions -->
21        <property name="transactionsExternallyManaged" value="true" />
22        <property name="databaseSchemaUpdate" value="true" />
23    </bean>
24 </beans>

```

这是Glassfish 3.1.1中的样子（假设已正确配置了名为`jdbc/activiti`的数据源）：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5 http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- 查找JTA事务管理器 Lookup the JTA-Transaction manager -->
8     <bean id="transactionManager" class="org.springframework.jndi.JndiObjectFactoryBean">
9         <property name="jndiName" value="java:appserver/TransactionManager"></property>
10        <property name="resourceRef" value="true" />

```



```

11     </bean>
12
13     <!-- 流程定义配置 process engine configuration -->
14     <bean id="processEngineConfiguration"
15           class="org.activiti.cdi.CdiJtaProcessEngineConfiguration">
16         <property name="dataSourceJndiName" value="jdbc/activiti" />
17         <property name="transactionManager" ref="transactionManager" />
18         <!-- 使用外部管理事务 using externally managed transactions -->
19         <property name="transactionsExternallyManaged" value="true" />
20         <property name="databaseSchemaUpdate" value="true" />
21     </bean>
</beans>

```

请注意上面的配置需要"spring-context"模块:

```

1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>spring-context</artifactId>
4   <version>3.0.3.RELEASE</version>
5 </dependency>

```

Java SE环境中的配置与[创建一个流程引擎](#)章节中的示例一样, 使用"StandaloneProcessEngineConfiguration"代替"CdiStandaloneProcessEngineConfiguration".

16.1.3. 部署流程 Deploying Processes

可以使用标准activiti API ([RepositoryService](#)) 部署流程。另外, `activiti-cdi`也提供了自动部署流程的功能, 使用classpath顶层的 `processes.xml` 文件提供流程列表。这是一个processes.xml文件的例子:

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <!-- 列表需要部署的流程 list the processes to be deployed -->
3 <processes>
4   <process resource="diagrams/myProcess.bpmn20.xml" />
5   <process resource="diagrams/myOtherProcess.bpmn20.xml" />
6 </processes>

```

16.2. CDI的基于上下文的流程执行 Contextual Process Execution with CDI

本章节我们将介绍Activiti cdi扩展使用的基于上下文的流程执行模型。BPMN业务流程通常是一个长期运行的交互动作, 包含用户与系统的任务。在运行时, 流程分割为独立工作单元的集合, 由用户与/或应用逻辑操作。在activiti-cdi中, 流程实例可以关联至一个cdi作用域, 这个关联代表了一个工作单元。如果工作单元很复杂 这就特别有用, 例如若一个用户任务由多个不同表单的复杂顺序组成, 并需要在交互过程中保持"非流程作用域 (non-process-scoped)"状态。

在默认配置中, 流程实例关联至"broadest (广播)"活动作用域, 一开始为会话, 并在会话上下文未激活时退化为请求。

16.2.1. 将一个会话关联至一个流程实例 Associating a Conversation with a Process Instance

当解析@BusinessProcessScoped bean, 或注入流程变量时, 会依赖一个激活的cdi作用域与一个流程实例的已有关联。Activiti-cdi提供了 `org.activiti.cdi.BusinessProcess` bean用于控制该关联, 特别是:

- `startProcessBy(...)`方法, 镜像了Activiti `RuntimeService` 服务暴露的对应方法, 用于启动并关联一个业务流程,
- `resumeProcessById(String processInstanceId)`, 用于将给定id关联至流程实例,
- `resumeTaskById(String taskId)`, 用于将给定id关联至任务 (以及扩展至相关的流程实例)。

当完成了一个工作单元 (例如一个用户任务) 时, 可以调用 `completeTask()` 方法, 解除流程实例与会话/请求的关联。这将通知Activiti 当前任务已完成, 并使流程实例继续运行。

请注意 `BusinessProcess` bean是一个@Named bean, 意味着可以使用表达式语言调用暴露的服务, 例如在JSF页面中调用。下面的JSF2代码片段启动了一个新的会话, 并将其关联至一个用户任务实例, 其id作为请求参数传递 (例如 `pageName.jsf?taskId=XX`):

```

1 <f:metadata>
2   <f:viewParam name="taskId" />
3   <f:event type="preRenderView" listener="#{businessProcess.startTask(taskId, true)}" />
4 </f:metadata>

```

16.2.2. 声明式控制流程 Declaratively controlling the Process

Activiti可以使用注解, 声明式启动流程实例以及完成任务。`@org.activiti.cdi.annotation.StartProcess` 注解可以通过"key"或"name"启动一个流程实例。请注意流程实例在注解的方法返回之后启动。例如:

```

1 @StartProcess("authorizeBusinessTripRequest")
2 public String submitRequest(BusinessTripRequest request) {
3   // 进行操作 do some work

```

```

4         return "success";
5     }

```

取决于Activiti的配置，被注解的方法代码以及流程实例的启动将处于同一个事务中。`@org.activiti.cdi.annotation.CompleteTask`的使用方式相同：

```

1 @CompleteTask(endConversation=false)
2 public String authorizeBusinessTrip() {
3     // 进行操作 do some work
4     return "success";
5 }

```

`@CompleteTask`注解提供了完成当前会话的能力。默认行为是在调用Activiti返回后结束回话。可以像上面的例子一样，禁用结束会话。

16.2.3. 从流程中引用Bean (Referencing Beans from the Process)

Activiti-cdi使用自定义解析器，将CDI bean暴露给Activiti EI。因此可以像这样在流程中引用bean：

```

1 <userTask id="authorizeBusinessTrip" name="Authorize Business Trip"
2     activiti:assignee="#{authorizingManager.account.username}" />

```

其中"authorizingManager"可以是生产者方法提供的bean：

```

1 @Inject @ProcessVariable Object businessTripRequesterUsername;
2
3 @Produces
4 @Named
5 public Employee authorizingManager() {
6     TypedQuery<Employee> query = entityManager.createQuery("SELECT e FROM Employee e WHERE e.account.username='"
7         + businessTripRequesterUsername + "'", Employee.class);
8     Employee employee = query.getSingleResult();
9     return employee.getManager();
10 }

```

可以使用`activiti:expression="myEjb.method()"`扩展，在服务任务中调用一个EJB中的业务方法。请注意这需要在`MyEjb`类上使用`@Named`注解。

16.2.4. 使用@BusinessProcessScoped bean (Working with @BusinessProcessScoped beans)

使用activiti-cdi，可以将一个bean的生命周期绑定在一个流程实例上。因此，提供了名为 `BusinessProcessContext` 的自定义的上下文实现。`BusinessProcessScoped` bean的实例将作为流程变量存储在当前流程实例中。`BusinessProcessScoped` bean需要是可持久化（`PassivationCapable`，例如`Serializable`）的。下面是一个流程作用域bean的例子：

```

1 @Named
2 @BusinessProcessScoped
3 public class BusinessTripRequest implements Serializable {
4     private static final long serialVersionUID = 1L;
5     private String startDate;
6     private String endDate;
7     // ...
8 }

```

有时希望在没有关联至流程实例的情况下使用流程作用域bean，例如在流程启动前。如果当前没有激活的流程实例，则`BusinessProcessScoped` bean的实例将临时存储在本地作用域（也就是会话或请求中，取决于上下文）。如果该作用域之后关联至一个业务流程实例，则会将bean实例刷入该流程实例。

16.2.5. 注入流程变量 Injecting Process Variables

可以注入流程变量。Activiti-CDI支持

- 使用`@Inject \[additional qualifiers\] Type fieldName`类型安全地注入`@BusinessProcessScoped` bean
- 使用`@ProcessVariable(name?)`限定名不安全地注入其它流程变量：

```

1 @Inject @ProcessVariable Object accountNumber;
2 @Inject @ProcessVariable("accountNumber") Object account

```

要在EL中引用流程变量，有类似的选择：

- `@Named @BusinessProcessScoped` bean可以直接引用，
- 其它流程变量可以通过`ProcessVariables` bean引用：

```
#{processVariables['accountNumber']}
```

16.2.6. 接收流程事件 Receiving Process Events

[EXPERIMENTAL]

Activiti可以关联至CDI事件总线。这样可以使用标准CDI事件机制获取流程事件。要为Activiti启用CDI事件支持，需要在配置中启用相应的处理监听器：

```
1 <property name="postBpmnParseHandlers">
2   <list>
3     <bean class="org.activiti.cdi.impl.event.CdiEventSupportBpmnParseHandler" />
4   </list>
5 </property>
```

这样Activiti就被配置为使用CDI事件总线发布事件。下面介绍如何在CDI bean中接收流程事件。事件通知是类型安全的。流程事件的类型是`org.activiti.cdi.BusinessProcessEvent`。下面是一个简单的事件观察者方法的例子：

```
1 public void onProcessEvent(@Observes BusinessProcessEvent businessProcessEvent) {
2     // 处理事件 handle event
3 }
```

观察者将会被通知所有事件。如果需要限制观察者接收的事件，可以添加限定注解：

- `@BusinessProcess`：限制事件为特定的流程定义。例如：`@Observes @BusinessProcess("billingProcess") BusinessProcessEvent evt`
- `@StartActivity`：使用特定的活动限制事件。例如：`@Observes @StartActivity("shipGoods") BusinessProcessEvent evt`将在进入id为"shipGoods"的活动时调用。
- `@EndActivity`：使用特定的活动限制事件。例如：`@Observes @EndActivity("shipGoods") BusinessProcessEvent evt`将在离开id为"shipGoods"的活动时调用。
- `@TakeTransition`：使用特定的路径限制事件。
- `@CreateTask`：使用特定任务的创建限制事件。
- `@DeleteTask`：使用特定任务的删除限制事件。
- `@AssignTask`：使用特定任务的指派限制事件。
- `@CompleteTask`：使用特定任务的完成限制事件。

上面的限定名可以自由组合。例如，要接收离开"shipmentProcess"中的"shipGoods"活动时生成的所有事件，可以撰写下面的观察者方法：

```
1 public void beforeShippingGoods(@Observes @BusinessProcess("shippingProcess") @EndActivity("shipGoods")
2 BusinessProcessEvent evt) {
3     // 处理事件 handle event
4 }
```

在默认配置中，事件监听器将在上下文的相同事务中同步调用。CDI事务性观察者（CDI transactional observer，只能与JavaEE/EJB一起使用）可以在事件交给观察者方法时控制。使用事务性观察者，可以例如保证观察者只在触发事件的事务成功 时才被通知：

```
1 public void onShipmentSucceeded(@Observes(during=TransactionPhase.AFTER_SUCCESS) @BusinessProcess("shippingProcess")
2 @EndActivity("shipGoods") BusinessProcessEvent evt) {
3     // 给客户发送邮件。 send email to customer.
4 }
```

16.2.7. 额外功能 Additional Features

- 可以注入流程引擎与服务：`@Inject ProcessEngine, RepositoryService, TaskService, ...`
- 可以注入当前的流程实例与任务：`@Inject ProcessInstance, Task,`
- 可以注入当前的businessKey：`@Inject @BusinessKey String businessKey,`
- 可以注入当前的流程实例id：`@Inject @ProcessInstanceId String pid +`

16.3. 已知限制 Known Limitations

尽管activiti-cdi依靠SPI实现，并设计为“移动性扩展”，但只使用Weld进行了测试。

17. 集成LDAP (LDAP integration)

公司通常已经有了LDAP（Lightweight Directory Access Protocol，轻量级目录访问协议）系统形式的用户与组存储。从5.14版本开始，Activiti提供了一个立即可用的解决方案，可以简单配置 Activiti与LDAP系统连接的方式。

在Activiti 5.14版本以前，也可以为Activiti集成LDAP。然而，在5.14中大幅简化了配置。但配置LDAP的“老”方法仍然可用。实际上，简化配置只是在“老”基础上进行的包装。

17.1. 使用 Usage

要在你的项目中添加LDAP集成代码，简单地在pom.xml中添加下列依赖：

```
1 <dependency>
2   <groupId>org.activiti</groupId>
3   <artifactId>activiti-ldap</artifactId>
4   <version>latest.version</version>
5 </dependency>
```

17.2. 用例 Use cases

目前LDAP集成有两大使用场景：

- 通过IdentityService进行认证。例如在使用Activiti Explorer，并需要通过LDAP登录时很有用。
- 获取一个组中的用户。例如在查询一个特定用户可见的任务（也就是说，特定候选组的任务）时很重要。

17.3. 配置 Configuration

在Activiti中集成LDAP系统通过在流程引擎配置的 `configurators` 小节注入 `org.activiti.ldap.LDAPConfigurator` 完成。这个类高度可扩展：如果默认实现不能满足使用场景，可以轻松地覆盖方法，许多依赖的bean也是可插拔的。

这是一个示例配置（请注意：当然，在程式创建引擎时完全类似）。现在不需要太关注这些参数，我们会在下一章节详细介绍。

```
1 <bean id="processEngineConfiguration" class="...SomeProcessEngineConfigurationClass">
2   ...
3   <property name="configurators">
4     <list>
5       <bean class="org.activiti.ldap.LDAPConfigurator">
6
7         <!-- 服务器连接参数 Server connection params -->
8         <property name="server" value="ldap://localhost" />
9         <property name="port" value="33389" />
10        <property name="user" value="uid=admin, ou=users, o=activiti" />
11        <property name="password" value="pass" />
12
13        <!-- 查询参数 Query params -->
14        <property name="baseDn" value="o=activiti" />
15        <property name="queryUserById" value="(&(objectClass=inetOrgPerson)(uid={0}))" />
16        <property name="queryUserByFullNameLike" value="(&(objectClass=inetOrgPerson)(|({0}=*{1}*)({2}=*{3}*)))" />
17      />
18
19      <property name="queryGroupsForUser" value="(&(objectClass=groupOfUniqueNames)(uniqueMember={0}))" />
20
21      <!-- 属性配置 Attribute config -->
22      <property name="userIdAttribute" value="uid" />
23      <property name="userFirstNameAttribute" value="cn" />
24      <property name="userLastNameAttribute" value="sn" />
25      <property name="userEmailAttribute" value="mail" />
26
27      <property name="groupIdAttribute" value="cn" />
28      <property name="groupNameAttribute" value="cn" />
29
30    />
31  />
32 />
```

17.4. 参数 Properties

可以为 `org.activiti.ldap.LDAPConfigurator` 设置下列参数：

Table 261. LDAP配置参数

参数名	描述	类型	默认值
server	LDAP系统的服务器。例如'ldap://localhost:33389'	String	
port	LDAP系统运行的端口	int	
user	连接LDAP系统的用户id	String	

参数名	描述	类型	默认值
password	连接LDAP系统的密码	String	
initialContextFactory	连接LDAP的InitialContextFactory的名字	String	com.sun.jndi ldap.LdapCtxFactory
securityAuthentication	连接LDAP系统时使用的'java.naming.security.authentication'参数	String	simple
customConnectionParameters	用于设置没有专门setter的所有LDAP连接参数。例如 http://docs.oracle.com/javase/tutorial/jndi/ldap/jndi.html 中的自定义参数。这些参数用于配制连接池，指定安全选项，等等。当创建到LDAP系统的连接时，会提供所有设置的参数。	Map<String, String>	
baseDn	开始搜索用户与组的基础“区分名”（distinguished name, DN）	String	
userBaseDn	开始搜索用户的基础“区分名”（distinguished name, DN）。如果未提供，则使用baseDn（见上）	String	
groupBaseDn	开始搜索组的基础“区分名”（distinguished name, DN）。如果未提供，则使用baseDn（见上）	String	
searchTimeLimit	在LDAP中搜索时的超时时间，以毫秒计	long	一个小时
queryUserById	使用 userId 搜索用户时执行的查询。例如：(&(objectClass=inetOrgPerson)(uid={0}))。将返回LDAP中所有class为'inetOrgPerson'，并匹配'uid'属性的值的对象。在例子中，用户id使用{0}注入。如果对于特定的LDAP配置，只使用该查询不满足要求，则可以使用不同的LDAPQueryBuilder插件，提供比查询更高的自由度。	string	
queryUserByFullNameLike	使用全名搜索用户时执行的查询。例如：(&(objectClass=inetOrgPerson) (&({0}={1})(&({2}={3})))。将返回LDAP中所有class为'inetOrgPerson'，并匹配first name与last name的值的对象。请注意{0}注入为firstNameAttribute（在上面定义），{1}与{3}为搜索文本，{2}为 lastNameAttribute。如果对于特定的LDAP配置，只使用该查询不满足要求，则可以使用不同的LDAPQueryBuilder插件，提供比查询更高的自由度。	string	
queryGroupsForUser	查找特定 用户的组时执行的查询。例如：(&(objectClass=groupOfUniqueNames)(uniqueMember={0}))。将返回class为'groupOfUniqueNames'，且提供的DN（匹配用户的DN）为'uniqueMember'的对象。在例子中，用户 id使用{0}注入。如果对于特定的LDAP配置，只使用该查询不满足要求，则可以使用不同的LDAPQueryBuilder插件，提供比查询更高的自由度。	string	
userIdAttribute	匹配用户id的属性的名字。这个属性用于查找用户对象，并完成LDAP对象与Activiti用户对象间的映射。	string	
userFirstNameAttribute	匹配用户first name的属性的名字。这个属性用于查找用户对象，并完成LDAP对象与Activiti用户对象间的映射。	string	
userLastNameAttribute	匹配用户last name的属性的名字。这个属性用于查找用户对象，并完成LDAP对象与Activiti用户对象间的映射。	string	
groupIdAttribute	匹配组id的属性的名字。这个属性用于查找组对象，并完成LDAP对象与Activiti用户对象间的映射。	string	

参数名	描述	类型	默认值
groupNameAttribute	匹配组name的属性的名字。这个属性用于查找组对象，并完成LDAP对象与Activiti用户对象间的映射。	String	
groupTypeAttribute	匹配组type的属性的名字。这个属性用于查找组对象，并完成LDAP对象与Activiti用户对象间的映射。	String	

下面的参数用于自定义默认行为或引入组缓存：

Table 262. 高级参数

参数名	描述	类型	默认值
ldapUserManagerFactory	如果默认实现不符合要求，设置一个自定义的LDAPUserManagerFactory实现。	LDAPUserManagerFactory的实例	
ldapGroupManagerFactory	如果默认实现不符合要求，设置一个自定义的LDAPGroupManagerFactory实现。	LDAPGroupManagerFactory的实例	
ldapMemberShipManagerFactory	如果默认实现不符合要求，设置一个自定义的LDAPMembershipManagerFactory实现。请注意很少出现这种情况，因为一般使用LDAP系统管理成员信息。	LDAPMembershipManagerFactory的实例	
ldapQueryBuilder	如果默认实现不符合要求，设置一个自定义的查询构建器。当在LDAP系统中使用LDAPUserManager或LDAPGroupManager进行实际查询时，会使用LDAPQueryBuilder的实例。默认实现使用在本实例中设置的参数，例如queryGroupsForUser与queryUserById	org.activiti.Idap.LDAPQueryBuilder的实例	
groupCacheSize	用于设置组缓存的大小。这是一个为用户缓存组的LRU缓存，可以避免每次需要查询用户所在组的时候都访问LDAP系统。 如果值小于0则不会使用缓存。默认值为-1，因此不会进行缓存。	int	-1
groupCacheExpirationTime	设置组缓存的过期时间，以毫秒计。当获取了一个特定用户的组，且存在组缓存时，会将组存储在缓存中，持续本参数设置的时间。也就是说，如果在00:00获取了一个组，过期时间为30分钟，则00:30之后进行的获取该用户组操作都不会使用该缓存，而是会从LDAP系统中重新获取。同样的，在00:00 - 00:30间会从该缓存中获取。	long	一个小时

使用活动目录（Active Directory）时请注意：Activiti论坛的用户报告，在使用活动目录时，需要将'InitialDirContext'设置为Context.REFERRAL。可以通过customConnectionParameters map按上面介绍的方法传递。

17.5. 在Explorer中集成LDAP (Integrate LDAP in Explorer)

- 在 `activiti-standalone-context.xml` 中添加上面介绍的LDAP配置

- 在WEB-INF/lib中添加activiti-ldap jar
- 移除demoDataGenerator bean，因为它会尝试添加用户（在集成LDAP时不允许）
- 为activiti-ui.context中的explorerApp bean添加下列配置：

```
1 <property name="adminGroups">
2   <list>
3     <value>admin</value>
4   </list>
5 </property>
6 <property name="userGroups">
7   <list>
8     <value>user</value>
9   </list>
10 </property>
```

直接使用你自己的值替换即可。需要使用的值是组的id（通过groupIdAttribute配置）。上面的配置将使'admin'组中的所有成员都成为Activiti Explorer的管理员用户，对于user组也类似。任何不匹配这些值的组都将成为'assignment'组，也就是说可以将任务指派给他们。

18. 高级 Advanced

下面的章节介绍了Activiti的高级用例，超出了一般的BPMN 2.0流程执行的范畴。因此，要理解这里的材料需要对Activiti足够熟练与精通。

18.1. 异步与作业执行器 Async and job executor

从5.17.0版本开始，Activiti在已有的作业执行器之外，还提供了异步执行器。在Activiti引擎中，两个执行器都处理定时器与异步作业，因此只能启用一个执行器。这个章节将介绍异步执行器与作业执行器的区别，以及为什么我们推荐使用异步执行器。请注意默认情况下Activiti使用已有的作业执行器，因为我们不希望没有进行明确的配置就启用异步执行器。

18.1.1. 异步执行器的设计 Async executor design

异步执行器包含了一个用于执行定时器与异步作业的线程池。当启用时，引擎将会使用已经持久化的异步作业实体调用异步执行器，然后线程池将异步地执行该作业。这是与老的作业执行器的主要区别，因为在异步作业已被持久化的情况下，作业执行器将会轮询数据库获取新作业。如果找到了一个作业，将锁定并执行它，这意味着大量额外的数据库通信。异步执行器会直接执行异步作业，而不会先轮询数据库。如果异步作业是一个排他作业，异步执行器将首先锁定流程实例的执行，如果成功则将执行作业，并之后解锁流程实例执行。如果锁定流程实例失败，则会重试。

定时器作业的逻辑与异步作业不同。对于定时器作业，异步与作业执行器的实现相近。异步执行器将轮询数据库中到期的定时器作业。然后会锁定并执行作业。

18.1.2. 作业执行器的设计 Job executor design

作业执行器也包含了一个用于执行定时器与异步作业的线程池。当启用时，作业执行器会轮询数据库，获取异步作业与定时器作业的到期时间。如果找到了一个作业，执行器将锁定并执行该作业，这意味着大量额外的数据库通信。相对应的，异步执行器会直接执行异步作业，而不会首先轮询数据库。

18.1.3. 异步执行器的优点 Advantages of the Async executor

- 更少的数据库查询，因为异步作业不通过轮询数据库就执行
- 对于非排他作业，不会再遇到OptimisticLockingExceptions（乐观锁异常）
- 排他作业现在锁定在流程实例级别，而不是在作业执行器中那样，使用笨重的逻辑查询排他作业。

18.1.4. 配置异步执行器 Async executor configuration

可以为异步执行器定义线程池大小与其他配置项。我们建议查看异步执行器的默认设置，核实其是否符合你的流程实例的要求。

要覆盖异步执行器的默认设置，需要在流程引擎配置中注入一个新的bean，像是：

```
1 <property name="asyncExecutor" ref="asyncExecutor" />
```

可以覆盖默认的异步执行器，以覆盖默认设置，当然也可以从DefaultAsyncJobExecutor类扩展。下面列出的配置中，DefaultAsyncJobExecutor的参数使用了新的值：

```
1 <bean id="asyncExecutor" class="org.activiti.engine.impl.asyncexecutor.DefaultAsyncJobExecutor">
2   <property name="corePoolSize" value="10" />
3   <property name="maxPoolSize" value="50" />
4   <property name="keepAliveTime" value="3000" />
5   <property name="queueSize" value="200" />
6   <property name="maxTimerJobsPerAcquisition" value="2" />
7   <property name="maxAsyncJobsDuePerAcquisition" value="2" />
8   <property name="defaultAsyncJobAcquireWaitTimeInMillis" value="1000" />
```

```

9     <property name="defaultTimerJobAcquireWaitTimeInMillis" value="1000" />
10    <property name="timerLockTimeInMillis" value="60000" />
11    <property name="asyncJobLockTimeInMillis" value="60000" />
12 </bean>

```

Table 263. 异步执行器配置选项

名字	默认值	描述
corePoolSize	2	线程池中为执行作业保留的最小线程数量。
maxPoolSize	10	线程池中为执行作业保留的最大线程数量。
keepAliveTime	5000	在销毁执行作业所用的线程前，需要保持活动的时间（以毫秒计）。默认设置为5000。设置为非默认的0值会消耗资源，但在有大量执行作业的时候，可以避免总是创建新线程。
queueSize	100	放置待执行作业的队列的大小。
maxTimerJobsPerAcquisition	1	在一次查询中获取的将到期定时器作业的数量。
maxAsyncJobsDuePerAcquisition	1	在一次数据库查询中获取的异步作业的数量。
defaultAsyncJobAcquireWaitTimeInMillis	10000	在两次执行异步作业查询之间等待的时间，以毫秒计。
defaultTimerJobAcquireWaitTimeInMillis	10000	在两次执行将到期定时器作业查询之间等待的时间，以毫秒计。
timerLockTimeInMillis	300000	在重试之前，定时器作业进行锁定操作的时间，以毫秒计。在该时间之后，Activiti引擎会视为定时器作业已经失败，并将重试锁定。
asyncJobLockTimeInMillis	300000	在重试之前，异步作业进行锁定操作的时间，以毫秒计。在该时间之后，Activiti引擎会视为异步作业已经失败，并将重试锁定。

18.2. 深入流程解析 Hooking into process parsing

一个BPMN 2.0 XML需要解析为Activiti的内部模型，才能在Activiti引擎中执行。解析发生在部署流程时；或没有在内存中找到流程的时候，这时将会从数据库获取XML。

对于每一个流程，`BpmnParser`类都会创建一个新的`BpmnParser`实例。这个实例是所有在解析时要做的事情的容器。解析本身很简单：对于每一个BPMN 2.0元素，引擎中都有一个对应的`org.activiti.engine.parse.BpmnParseHandler`的实例。因此，解析器会将一个BPMN 2.0元素类映射到一个`BpmnParseHandler`实例。默认情况下，Activiti使用`BpmnParseHandler`实例处理所有支持的元素，并用其为流程的步骤附加执行监听器，以创建历史。

可以在Activiti引擎中添加`org.activiti.engine.parse.BpmnParseHandler`的自定义实例。常见使用场景是，例如为特定步骤添加执行监听器，用于向某个事件处理队列触发事件。Activiti内部使用这种方式处理历史。要添加这种自定义处理器，需要调整Activiti配置：

```

1  <property name="preBpmnParseHandlers">
2    <list>
3      <bean class="org.activiti.parsing.MyFirstBpmnParseHandler" />
4    </list>
5  </property>
6
7  <property name="postBpmnParseHandlers">
8    <list>
9      <bean class="org.activiti.parsing.MySecondBpmnParseHandler" />
10     <bean class="org.activiti.parsing.MyThirdBpmnParseHandler" />
11   </list>
12 </property>

```


在 `preBpmnParseHandlers` 参数中配置的 `BpmnParseHandler` 实例的列表将添加在任何默认处理器之前。类似的，`postBpmnParseHandlers` 中的将添加在默认处理器之后。在顺序会影响自定义解析处理器中包含的逻辑时很重要。

`org.activiti.engine.parse.BpmnParseHandler` 是一个简单的接口：

```
1 public interface BpmnParseHandler {
2
3     Collection<Class>? extends BaseElement>> getHandledTypes();
4
5     void parse(BpmnParse bpmnParse, BaseElement element);
6
7 }
```

`getHandledTypes()` 方法返回该解析器处理的所有类型的集合。通过集合的泛型决定了可用的类型是 `BaseElement` 的子类。也可以扩展 `AbstractBpmnParseHandler` 类，并覆盖 `getHandledType()` 方法，它只返回一个类而不是一个集合。这个类也包含了一些默认解析处理器共享的辅助方法。当解析器遇到任何该方法的返回类型时，将调用 `BpmnParseHandler` 实例。在下面的例子里，当遇到BPMN 2.0 XML中包含的流程时，将会执行 `executeParse` 方法（这是一个类型转换方法，取代了 `BpmnParseHandler` 接口中的普通 `parse` 方法）中的逻辑。

```
1 public class TestBPMNParseHandler extends AbstractBpmnParseHandler<Process> {
2
3     protected Class<? extends BaseElement> getHandledType() {
4         return Process.class;
5     }
6
7     protected void executeParse(BpmnParse bpmnParse, Process element) {
8         ..
9     }
10
11 }
```

重要提示：在撰写自定义解析处理器时，不要使用任何用于解析BPMN 2.0结构的内部类。这将导致很难查找bug。实现一个自定义处理器安全的做法是实现 `BpmnParseHandler` 接口，或扩展内部抽象类 `org.activiti.engine.impl.bpmn.parser.handler.AbstractBpmnParseHandler`。

可以（但不常见）替换默认用于将BPMN 2.0元素解析为Activiti内部模型的 `BpmnParseHandler` 实例。可以通过下面的代码片段实现：

```
1 <property name="customDefaultBpmnParseHandlers">
2     <list>
3         ...
4     </list>
5 </property>
```

简单的例子是用于将所有服务任务都强制异步执行：

```
1 public class CustomUserTaskBpmnParseHandler extends ServiceTaskParseHandler {
2
3     protected void executeParse(BpmnParse bpmnParse, ServiceTask serviceTask) {
4
5         // 进行常规操作 Do the regular stuff
6         super.executeParse(bpmnParse, serviceTask);
7
8         // 保证异步 Make always async
9         ActivityImpl activity = findActivity(bpmnParse, serviceTask.getId());
10        activity.setAsync(true);
11    }
12
13 }
```

18.3. 高并发下使用的UUID id生成器 UUID id generator for high concurrency

在某些（非常）高并发负载的情况下，默认的id生成器可能会由于不能够够快地获取新的id块而产生异常。每一个流程引擎都有一个id生成器。默认的id生成器在数据库中保留一个块的id，这样其他引擎就不能使用同一个块中的id。在引擎操作时，当默认的id生成器发现id块已经用完，就会启动一个新的事务，来获取一个新的块。在（非常）有限的使用场景下，当负载非常高时可能导致问题。对于大多数用例来说，默认的id生成器已经足够使用了。默认的 `org.activiti.engine.impl.db.DbIdGenerator` 也有一个 `idBlockSize` 参数，用于配置保留的id块的大小，可以调整获取id的行为。

默认的id生成器的替代品是 `org.activiti.engine.impl.persistence.StrongUuidGenerator`，它会在本地生成一个唯一的UUID，并将其用作所有实体的标识符。因为UUID不需要访问数据库就能生成，因此在非常高并发的使用场景下更合适。请注意取决于机器，性能可能与默认的id生成器不同（更好更坏都有可能）。

可以在activiti配置中，像下面这样配置UUID生成器：

```
1 <property name="idGenerator">
2   <bean class="org.activiti.engine.impl.persistence.StrongUuidGenerator" />
3 </property>
```

使用UUID id生成器需要添加下列额外依赖：

```
1 <dependency>
2   <groupId>com.fasterxml.uuid</groupId>
3   <artifactId>java-uuid-generator</artifactId>
4   <version>3.1.3</version>
5 </dependency>
```

18.4. 多租户 Multitenancy

总的来说，多租户是一个软件为多个不同组织提供服务的概念。其核心是数据是隔离的，一个组织不能看到其他组织的数据。在这个语境中，一个这样的组织（或部门、团队……）被称为一个租户（*tenant*）。

请注意它与多实例安装方式有本质区别，其中多实例安装是指每一个组织都分别运行一个Activiti流程引擎实例（并使用不同的数据库账户）。尽管Activiti比较轻量级，运行一个流程引擎实例不会花费太多资源，但多实例安装仍然增加了复杂性与维护量。但是，在某些使用场景中，多实例安装可能是正确的解决方案。

Activiti中的多租户主要围绕着隔离数据实现。要注意Activiti并不强制多租户规则。这意味着当查询与使用数据时，并不会验证进行操作的用户是否属于正确的租户。这应该在调用Activiti引擎的层次实现。Activiti确保可以存储租户信息，并在获取流程数据时使用。

在Activiti流程引擎中部署流程定义时，可以传递一个租户标识符（*tenant identifier*）。这是一个字符串（例如一个UUID，部门id，等等……），限制为256个字符长，唯一标识租户：

```
1 repositoryService.createDeployment()
2   .addClassPathResource(...)
3   .tenantId("myTenantId")
4   .deploy();
```

在部署时传递一个租户id带有下列含义：

- 部署中包含的所有流程定义都将从该部署集成租户标识符。
- 从这些流程定义启动的所有流程实例都将从流程定义继承租户标识符。
- 在执行流程实例时，运行时创建的所有任务都将从流程实例继承租户标识符。独立任务也可以有租户标识符。
- 执行流程实例时创建的所有执行都将流程实例继承租户标识符。
- 触发一个信号抛出事件（在流程内或通过API）时可以提供租户标识符。这个信号将只在该租户的上下文中执行：也就是说，如果有多个使用相同名字的信号捕获事件，只会调用带有正确租户标识符的事件。
- 所有作业（定时器与异步延续）要么从流程定义（例如定时器启动事件），要么从流程实例（运行时创建的作业，例如异步延续）继承租户标识符。这可以用于在自定义作业执行器中为部分租户设置优先级。
- 所有历史实体（历史流程实例、任务与活动）都从其对应的运行时对象继承租户标识符。
- 另外，模型也可以有租户标识符（模型在例如Activiti Modeler存储BPMN 2.0模型的时候使用）。

为了实际使用流程数据上的租户标识符，所有查询API都可以通过租户过滤。例如（也可以使用其他实体的对应查询实现替换）：

```
1 runtimeService.createProcessInstanceQuery()
2   .processInstanceTenantId("myTenantId")
3   .processDefinitionKey("myProcessDefinitionKey")
4   .variableValueEquals("myVar", "someValue")
5   .list();
```

查询API也可以使用*like*语义通过租户标识符过滤，也可以过滤掉没有租户标识符的实体。

重要的实现细节：由于数据库的原因（更确切地说，唯一约束的null处理），默认的代表没有租户的租户标识符为空字符串。（流程定义key，流程定义版本，租户标识符）的组合需要是唯一的（并且通过数据库约束检查）。也请注意租户标识符不能设置为null，不然会影响查询，因为某些数据库（Oracle）将空字符串当做null值（这就是为什么*withoutTenantId*查询不检查空字符串还是null）。这意味着同一个流程定义（有相同的流程定义key）可以为多个租户部署，每一个租户都有他们自己的版本。未使用租户时不会影响使用。

请注意上面所说都不与在集群中运行多个Activiti实例冲突。

[试验性] 可以调用repositoryService的changeDeploymentTenantId(String deploymentId, String newTenantId)方法修改租户标识符。这将修改每一处之前继承的租户标识符。在从非多租户环境迁移至多租户配置时很有用。查看该方法的Javadoc了解更多细节信息。

18.5. 执行自定义SQL Execute custom SQL

Activiti API可以通过高级API与数据库交互。例如，要获取数据，查询API与原生（Native）查询API各有用武之地。然而，在某些用例下，可能不够灵活。下面的章节描述了如何在Activiti数据存储中执行完全自定义的SQL语句（select、insert、update与delete都可以），且完全在配置的流程引擎范围内（例如因此可以使用事务设置）。

要定义自定义SQL语句，activiti引擎使用其底层框架MyBatis的功能。可以在[MyBatis用户手册](#)中阅读更多信息。

18.5.1. 基于注解的映射语句 Annotation based Mapped Statements

当使用基于注解的映射语句时，首先要做的是创建一个ByBatis映射类。例如，假设在某个用例中，不需要所有的任务数据，而只需要其中很少一部分。可以通过映射类完成，像是这样：

```
1 public interface MyTestMapper {
2
3     @Select("SELECT ID_ as id, NAME_ as name, CREATE_TIME_ as createTime FROM ACT_RU_TASK")
4     List<Map<String, Object>> selectTasks();
5
6 }
```

该映射类必须像下面这样提供给流程引擎配置：

```
1 ...
2 <property name="customMybatisMappers">
3     <set>
4         <value>org.activiti.standalone.cfg.MyTestMapper</value>
5     </set>
6 </property>
7 ...
```

请注意这是一个接口。底层的MyBatis框架会构造一个它的实例，并在运行时使用。也请注意方法的返回值没有类型，而只是一个map的list（代表了带有列数据的行的列表）。如果需要，可以通过MyBatis映射类设置类型。

要执行上面的查询，必须使用`managementService.executeCustomSql`方法。这个方法使用一个`CustomSqlExecution`实例。这是一个包装器，将引擎需要处理的内部数据隐藏起来。

不幸的是，Java泛型让它没有本应该的那么可读。下面的两个泛型类是映射类与其返回类型类。然而，实际的逻辑就是简单的调用映射方法，并返回其结果（若有）。

```
1 CustomSqlExecution<MyTestMapper, List<Map<String, Object>>> customSqlExecution =
2     new AbstractCustomSqlExecution<MyTestMapper, List<Map<String, Object>>>(MyTestMapper.class) {
3
4     public List<Map<String, Object>> execute(MyTestMapper customMapper) {
5         return customMapper.selectTasks();
6     }
7
8 };
9
10 List<Map<String, Object>> results = managementService.executeCustomSql(customSqlExecution);
```

在这个例子里，上面列出的映射实体只包含`id`、`name`与创建时间，而不是完整的任务对象。

上面的方法可以使用任何SQL。另一个更复杂的例子：

```
1 @Select({
2     "SELECT task.ID_ as taskId, variable.LONG_ as variableValue FROM ACT_RU_VARIABLE variable",
3     "inner join ACT_RU_TASK task on variable.TASK_ID_ = task.ID_",
4     "where variable.NAME_ = #{variableName}"
5 })
6 List<Map<String, Object>> selectTaskWithSpecificVariable(String variableName);
```

使用这个方法，会将任务表与变量表联合。只选择变量有特定名字的记录，并返回任务id与对应的数字值。

对于使用基于注解的映射语句的实际例子，请查看单元测试`org.activiti.standalone.cfg.CustomMybatisMapperTest`与`src/test/java/org/activiti/standalone/cfg/`、`src/test/resources/org/activiti/standalone/cfg/`目录中的其它类与资源。

18.5.2. 基于XML的映射语句 XML based Mapped Statements

当使用基于XML的映射语句时，语句在XML文件中定义。对于不需要整个任务数据，而只需要其中很少一部分的用例来说，XML文件像是下面这样：

```
1 <mapper namespace="org.activiti.standalone.cfg.TaskMapper">
2
3     <resultMap id="customTaskResultMap" type="org.activiti.standalone.cfg.CustomTask">
4         <id property="id" column="ID_" jdbcType="VARCHAR"/>
5         <result property="name" column="NAME_" jdbcType="VARCHAR"/>
6     </resultMap>
7
8     <select id="selectTaskWithSpecificVariable" resultMap="customTaskResultMap">
9         SELECT task.ID_ as taskId, variable.LONG_ as variableValue FROM ACT_RU_VARIABLE variable
10        inner join ACT_RU_TASK task on variable.TASK_ID_ = task.ID_
11        where variable.NAME_ = #{variableName}
12     </select>
13 </mapper>
```

```

6      <result property="createTime" column="CREATE_TIME_" jdbcType="TIMESTAMP" />
7  </resultMap>
8
9  <select id="selectCustomTaskList" resultMap="customTaskResultMap">
10     select RES.ID_, RES.NAME_, RES.CREATE_TIME_ from ACT_RU_TASK RES
11 </select>
12
13 </mapper>

```

结果映射为`org.activiti.standalone.cfg.CustomTask`类的实例，像是下面这样：

```

1 public class CustomTask {
2
3     protected String id;
4     protected String name;
5     protected Date createTime;
6
7     public String getId() {
8         return id;
9     }
10    public String getName() {
11        return name;
12    }
13    public Date getCreateTime() {
14        return createTime;
15    }
16 }

```

必须像下面这样为流程引擎配置提供映射XML文件：

```

1 ...
2 <property name="customMybatisXMLMappers">
3     <set>
4         <value>org/activiti/standalone/cfg/custom-mappers/CustomTaskMapper.xml</value>
5     </set>
6 </property>
7 ...

```

语句可以如下执行：

```

1 List<CustomTask> tasks = managementService.executeCommand(new Command<List<CustomTask>>() {
2
3     @SuppressWarnings("unchecked")
4     @Override
5     public List<CustomTask> execute(CommandContext commandContext) {
6         return (List<CustomTask>) commandContext.getDbSqlSession().selectList("selectCustomTaskList");
7     }
8 });

```

对于需要更复杂语句的用例，XML映射语句很有帮助。因此Activiti内部就使用XML映射语句，它可以确保底层功能。

假设某个用例下，需要基于id、name、type、userId等字段，查询附件数据！要实现这个用例，可以创建一个扩展了`org.activiti.engine.impl.AbstractQuery`的查询类`AttachmentQuery`，像下面这样：

```

1 public class AttachmentQuery extends AbstractQuery<AttachmentQuery, Attachment> {
2
3     protected String attachmentId;
4     protected String attachmentName;
5     protected String attachmentType;
6     protected String userId;
7
8     public AttachmentQuery(ManagementService managementService) {
9         super(managementService);
10    }
11
12    public AttachmentQuery attachmentId(String attachmentId){
13        this.attachmentId = attachmentId;
14        return this;
15    }
16
17    public AttachmentQuery attachmentName(String attachmentName){
18        this.attachmentName = attachmentName;
19        return this;
20    }
21
22    public AttachmentQuery attachmentType(String attachmentType){
23        this.attachmentType = attachmentType;
24        return this;
25    }
26 }

```

```

25     }
26
27     public AttachmentQuery userId(String userId){
28         this.userId = userId;
29         return this;
30     }
31
32     @Override
33     public long executeCount(CommandContext commandContext) {
34         return (Long) commandContext.getDbSqlSession()
35             .selectOne("selectAttachmentCountByQueryCriteria", this);
36     }
37
38     @Override
39     public List<Attachment> executeList(CommandContext commandContext, Page page) {
40         return commandContext.getDbSqlSession()
41             .selectList("selectAttachmentByQueryCriteria", this);
42     }

```

请注意在扩展`AbstractQuery`时，扩展类需要为super构造器传递一个`ManagementService`的实例，并需要实现`executeCount`与`executeList`来调用映射语句。

包含映射语句的XML文件像是下面这样：

```

1  <mapper namespace="org.activiti.standalone.cfg.AttachmentMapper">
2
3      <select id="selectAttachmentCountByQueryCriteria" parameterType="org.activiti.standalone.cfg.AttachmentQuery"
4      resultType="long">
5          select count(distinct RES.ID_)
6          <include refid="selectAttachmentByQueryCriteriaSql"/>
7      </select>
8
9      <select id="selectAttachmentByQueryCriteria" parameterType="org.activiti.standalone.cfg.AttachmentQuery"
10     resultMap="org.activiti.engine.impl.persistence.entity.AttachmentEntity.attachmentResultMap">
11         ${limitBefore}
12         select distinct RES.* ${limitBetween}
13         <include refid="selectAttachmentByQueryCriteriaSql"/>
14         ${orderBy}
15         ${limitAfter}
16     </select>
17
18     <sql id="selectAttachmentByQueryCriteriaSql">
19         from ${prefix}ACT_HI_ATTACHMENT RES
20         <where>
21             <if test="attachmentId != null">
22                 RES.ID_ = #{attachmentId}
23             </if>
24             <if test="attachmentName != null">
25                 and RES.NAME_ = #{attachmentName}
26             </if>
27             <if test="attachmentType != null">
28                 and RES.TYPE_ = #{attachmentType}
29             </if>
30             <if test="userId != null">
31                 and RES.USER_ID_ = #{userId}
32             </if>
33         </where>
34     </sql>
35 </mapper>

```

可以在语句中使用例如分页、排序、表名前缀等功能（因为`parameterType`为`AbstractQuery`的子类）。请注意可以使用已定义的`org.activiti.engine.impl.persistence.entity.AttachmentEntity.attachmentResultMap`来映射结果。

最后，`AttachmentQuery`可以如下使用：

```

1  ....
2  // 获取附件的总数 Get the total number of attachments
3  long count = new AttachmentQuery(managementService).count();
4
5  // 获取id为10025的附件 Get attachment with id 10025
6  Attachment attachment = new AttachmentQuery(managementService).attachmentId("10025").singleResult();
7
8  // 获取前10个附件 Get first 10 attachments
9  List<Attachment> attachments = new AttachmentQuery(managementService).listPage(0, 10);
10
11 // 获取用户kermit上传的所有附件 Get all attachments uploaded by user kermit
12 attachments = new AttachmentQuery(managementService).userId("kermit").list();
13 ....

```

对于使用基于XML的映射语句的实际例子，请查看单元测试`org.activiti.standalone.cfg.CustomMybatisXMLMapperTest`与`src/test/java/org/activiti/standalone/cfg/`、`src/test/resources/org/activiti/standalone/cfg/`目录中的其它类与资源。

18.6. 使用ProcessEngineConfigurator进行高级流程引擎配置 Advanced Process Engine configuration with a ProcessEngineConfigurator

深入流程引擎配置的高级方法是使用`ProcessEngineConfigurator`。方法是创建一个`org.activiti.engine.cfg.ProcessEngineConfigurator`接口的实现，并注入到流程引擎配置中：

```
1 <bean id="processEngineConfiguration" class="...SomeProcessEngineConfigurationClass">
2
3   ...
4
5   <property name="configurators">
6     <list>
7       <bean class="com.mycompany.MyConfigurator">
8         ...
9       </bean>
10    </list>
11  </property>
12
13  ...
14
15 </bean>
```

这个接口需要实现两个方法。`configure`方法，使用一个`ProcessEngineConfiguration`实例作为参数。可以使用这个方式添加自定义配置，并且这个方法会保证在流程引擎创建之前，所有默认配置已经完成之后调用。另一个方法是`getPriority`方法，可以指定配置器的顺序，以备某些配置器对其他的有依赖。

这种配置器的一个例子是[LDAP集成](#)，其中配置器用于将默认的用户与组管理类，替换为可以处理LDAP用户存储的实现。因此基本上配置器可以相当大地改变或调整流程引擎，也意味着非常高级的使用场景。另一个例子是使用自定义的版本替换流程引擎缓存：

```
1 public class ProcessDefinitionCacheConfigurator extends AbstractProcessEngineConfigurator {
2
3   public void configure(ProcessEngineConfigurationImpl processEngineConfiguration) {
4     MyCache myCache = new MyCache();
5     processEngineConfiguration.setProcessDefinitionCache(enterpriseProcessDefinitionCache);
6   }
7
8 }
```

也可以使用[ServiceLoader](#)方法，从classpath中自动发现流程引擎配置器。这意味着包含配置器实现的jar必须放在classpath下，并在jar的`META-INF/services`目录下包含名为`org.activiti.engine.cfg.ProcessEngineConfigurator`的文件。文件的内容必须是自定义实现的全限定类名。当流程引擎启动时，日志会显示找到这些配置器：

```
INFO org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl - Found 1 auto-discoverable Process Engine Configurators
INFO org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl - Found 1 Process Engine Configurators in total:
INFO org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl - class org.activiti.MyCustomConfigurator
```

请注意ServiceLoader方法可能在某些环境不能运行。可以通过ProcessEngineConfiguration的`enableConfiguratorServiceLoader`参数明确禁用（默认为true）。

18.7. 高级查询API：在运行时与历史任务查询间无缝切换 Advanced query API: seamless switching between runtime and historic task querying

任何BPM用户界面的核心组件都是任务列表。一般来说，最终用户操作运行时的任务，在收件箱中通过不同设置进行过滤。有时也需要在这些列表中显示历史任务，并进行类似的过滤。为了简化代码，`TaskQuery`与`HistoricTaskInstanceQuery`有共同的父接口，其中包含了所有公共操作（大多数操作都是公共的）。

这个公共接口是`org.activiti.engine.task.TaskInfoQuery`类。`org.activiti.engine.task.Task`与`org.activiti.engine.task.HistoricTaskInstance`都有公共父类`org.activiti.engine.task.TaskInfo`（并带有公共参数），并将作为例如`list()`方法的返回值。然而，有时Java泛型会帮倒忙：如果想要直接使用`TaskInfoQuery`类型，将会是这样：

```
1 TaskInfoQuery<? extends TaskInfoQuery<?,?>, ? extends TaskInfo> taskInfoQuery
```

呃.....好吧。为了“解决”这个问题，可以使用`org.activiti.engine.task.TaskInfoQueryWrapper`类来避免泛型（下面的代码来自REST的代码，将返回一个任务列表，且用户可以选择查看进行中还是已完成的任务）：

```
1 TaskInfoQueryWrapper taskInfoQueryWrapper = null;
2 if (runtimeQuery) {
```



```

3         taskInfoQueryWrapper = new TaskInfoQueryWrapper(taskService.createTaskQuery());
4     } else {
5         taskInfoQueryWrapper = new TaskInfoQueryWrapper(historyService.createHistoricTaskInstanceQuery());
6     }
7
8     List<? extends TaskInfo> taskInfos = taskInfoQueryWrapper.getTaskInfoQuery().or()
9         .taskNameLike("%k1%")
10        .taskDueAfter(new Date(now.getTime() + (3 * 24L * 60L * 60L * 1000L)))
11        .endOr()
12        .list();

```

18.8. 通过覆盖标准SessionFactory自定义身份管理 Custom identity management by overriding standard SessionFactory

如果不想像LDAP集成中那样，使用完整的`ProcessEngineConfigurator`实现，但仍然希望将自定义的身份管理插入框架中，那么也可以直接覆盖`ProcessEngineConfiguration`中的`SessionFactory`类。在Spring中，可以简单地通过向`ProcessEngineConfiguration` bean定义添加下面的代码实现：

```

1 <bean id="processEngineConfiguration" class="...SomeProcessEngineConfigurationClass">
2
3     ...
4
5     <property name="customSessionFactories">
6         <list>
7             <bean class="com.mycompany.MyGroupManagerFactory"/>
8             <bean class="com.mycompany.MyUserManagerFactory"/>
9         </list>
10    </property>
11
12    ...
13
14 </bean>

```

`MyGroupManagerFactory`与`MyUserManagerFactory`需要实现`org.activiti.engine.impl.interceptor.SessionFactory`接口。对`openSession()`的调用，需要返回实际管理身份的自定义类的实现。对于组，需要是继承`org.activiti.engine.impl.persistence.entity.GroupEntityManager`的类，对于用户管理，需要是继承`org.activiti.engine.impl.persistence.entity.UserEntityManager`的类。下面的代码样例包含了一个自定义的组管理器工厂：

```

1 package com.mycompany;
2
3 import org.activiti.engine.impl.interceptor.Session;
4 import org.activiti.engine.impl.interceptor.SessionFactory;
5 import org.activiti.engine.impl.persistence.entity.GroupIdentityManager;
6
7 public class MyGroupManagerFactory implements SessionFactory {
8
9     @Override
10    public Class<?> getSessionType() {
11        return GroupIdentityManager.class;
12    }
13
14    @Override
15    public Session openSession() {
16        return new MyCompanyGroupManager();
17    }
18
19 }

```

实际工作由这个工厂创建的`MyCompanyGroupManager`进行。但不需要覆盖`GroupEntityManager`的所有成员，只需要覆盖使用场景需要的那些即可。下面的样例展示了可能的样子（只展示一部分成员）：

```

1 public class MyCompanyGroupManager extends GroupEntityManager {
2
3     private static Logger log = LoggerFactory.getLogger(MyCompanyGroupManager.class);
4
5     @Override
6     public List<Group> findGroupsByUser(String userId) {
7         log.debug("findGroupByUser called with userId: " + userId);
8         return super.findGroupsByUser(userId);
9     }
10
11    @Override
12    public List<Group> findGroupByQueryCriteria(GroupQueryImpl query, Page page) {
13        log.debug("findGroupByQueryCriteria called, query: " + query + " page: " + page);
14        return super.findGroupByQueryCriteria(query, page);
15    }
16

```

```

17     @Override
18     public long findGroupCountByQueryCriteria(GroupQueryImpl query) {
19         log.debug("findGroupCountByQueryCriteria called, query: " + query);
20         return super.findGroupCountByQueryCriteria(query);
21     }
22
23     @Override
24     public Group createNewGroup(String groupId) {
25         throw new UnsupportedOperationException();
26     }
27
28     @Override
29     public void deleteGroup(String groupId) {
30         throw new UnsupportedOperationException();
31     }
32 }

```

在适当的方法中添加你自己的实现，以插入自己的身份管理解决方案。需要自行判断要覆盖基类中的那些成员。例如下面的调用：

```

1 long potentialOwners = identityService.createUserQuery().memberOfGroup("management").count();

```

会调用 *UserIdentityManager* 接口的下列成员：

```

1 List<User> findUserByQueryCriteria(UserQueryImpl query, Page page);

```

LDAP集成中的代码包含了如何实现这些的完整示例。可以在GitHub查看代码，特别是 *LDAPGroupManager* 与 *LDAPUserManager*。

18.9. 启用安全BPMN 2.0 XML (Enable safe BPMN 2.0 xml)

在大多数情况下，部署至Activiti引擎的BPMN 2.0流程都在例如开发团队的严格控制下。然而，有的时候能够向引擎上传任意的BPMN 2.0 XML很诱人。在这种情况下，需要考虑动机不良的用户可能会像[这里](#)描述的一样，搞坏服务器。

要避免上面链接中描述的攻击，可以在流程引擎配置中设置 *enableSafeBpmnXml* 参数：

```

1 <property name="enableSafeBpmnXml" value="true"/>

```

默认情况下这个功能是禁用的！原因是它依赖 *StaxSource* 类。而不幸的是，某些平台（例如JDK6，JBoss，等等）不能使用这个类（由于过时的XML解析器实现），因此不能启用安全BPMN 2.0 XML功能。

如果Activiti运行的平台支持，请一定要启用这个功能。

18.10. 事件记录（试验性） Event logging (Experimental)

从Activiti 5.16开始，引入了（试验性）的事件记录机制。记录机制基于Activiti引擎的事件机制的一般用途，并默认禁用。其思想是，来源于引擎的事件会被捕获，并创建一个包含了所有事件数据（甚至更多）的映射，提供给 *org.activiti.engine.impl.event.logger.EventFlusher*，由它将这些数据刷入其他地方。默认情况下，使用简单的基于数据库的事件处理/刷入，会使用Jackson将上述映射序列化为JSON，并将其作为 *EventLogEntryEntity* 接口存入数据库。如果不使用事件记录，可以删除这个表。

要启用数据库记录：

```

1 processEngineConfiguration.setEnableDatabaseEventLogging(true);

```

或在运行时：

```

1 databaseEventLogger = new EventLogger(processEngineConfiguration.getClock());
2 runtimeService.addEventListener(databaseEventLogger);

```

可以扩展 *EventLogger* 类。如果默认的数据库记录不符合要求，需要覆盖 *createEventFlusher()* 方法返回一个 *org.activiti.engine.impl.event.logger.EventFlusher* 接口的实例。可以通过Activiti的 *managementService.getEventLogEntries(startLogNr, size)* 获取 *EventLogEntryEntity* 实例。

容易看出这个表中的数据可以通过JSON放入大数据NoSQL存储，例如MongoDB，Elastic Search，等等。也容易看出这里使用的类（*org.activiti.engine.impl.event.logger.EventLogger/EventFlusher* 与许多其他 *EventHandler* 类）是可插入的，可以按你的使用场景调整（例如不将JSON存入数据库，而是将其直接发送到一个队列或大数据存储）。

请注意这个事件记录机制是额外于Activiti的“传统”历史管理器的。尽管所有数据都在数据库表中，但并未对查询或快速恢复做优化。实际使用场景是末端审计并将其存入大数据存储。

18.11. 禁用批量插入 Disabling bulk inserts

默认情况下，引擎会对同一个数据库表的多个插入语句组合在一起，作为批量插入，这样能够提高性能，并已在所有支持的数据库中测试与实现了。

然而，支持与测试过的数据库，可能有某个特定版本不支持批量插入（例如有报告说DB2在z/OS上不支持，尽管一般来说DB2是支持的），可以在流程引擎配置中禁用批量插入：

```
1 <property name="bulkInsertEnabled" value="false" />
```

18.12. 安全脚本 Secure Scripting

试验性：安全脚本功能随Activiti 5.21发布。

默认情况下，使用脚本任务时，执行的脚本与Java代理具有相似的能力。可以完全访问JVM，永远运行（无限循环），或占用大量内存。然而，Java代理需要撰写并放在classpath的jar中，与流程定义的生命周期不同。最终用户一般不会撰写Java代理，因为这基本上是开发者的工作。

然而脚本是流程定义的一部分，具有相同的生命周期。脚本任务不需要额外的jar部署步骤，而是在流程部署后就可以执行。有时，脚本任务中的脚本不是由开发者撰写的。这就有一个上面提到的问题：脚本可以完全访问JVM，也可以在执行脚本时阻塞许多系统资源。因此允许来自几乎任何人的脚本不是一个好主意。

要解决这个问题，可以启用安全脚本功能。目前，这个功能只实现了javascript脚本。要启用它，向你的项目添加activiti-secure-javascript依赖。使用Maven时：

```
1 <dependency>
2   <groupId>org.activiti</groupId>
3   <artifactId>activiti-secure-javascript</artifactId>
4   <version>${activiti.version}</version>
5 </dependency>
```

添加这个依赖会同时引入Rhino依赖（参见<https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino>）。Rhino是一个用于JDK的javascript引擎。过去包含在JDK版本6与7中，并已被Nashorn引擎取代。然而，Rhino项目仍然在继续开发。许多功能（包括Activiti用于实现安全脚本的）都在之后才加入。在撰写本手册的时候，Nashorn还没有实现安全脚本功能需要的功能。

这意味着脚本之间可能要做一些（基本很少）改变（例如，Rhino使用importPackage，而Nashorn使用load()）。这些改变与将脚本从JDK 7切换至8相似。

通过专门的Configurator对象配置安全脚本，并在流程引擎实例化之前将其传递给流程引擎配置：

```
1 SecureJavascriptConfigurator configurator = new SecureJavascriptConfigurator()
2   .setWhiteListedClasses(new HashSet<String>(Arrays.asList("java.util.ArrayList")))
3   .setMaxStackDepth(10)
4   .setMaxScriptExecutionTime(3000L)
5   .setMaxMemoryUsed(3145728L)
6   .setNrOfInstructionsBeforeStateCheckCallback(10);
7
8 processEngineConfig.addConfigurator(configurator);
```

可以使用下列设置：

- **enableClassWhiteListing**: 为true时，会黑名单所有类。希望运行的所有类都需要添加入白名单，这样就严格控制了暴露给脚本的东西。默认为false。
- **whiteListedClasses**: 一个全限定类名字字符串的集合，表示允许脚本中使用的类。例如，要在脚本中暴露execution对象，需要在这个集合中添加org.activiti.engine.impl.persistence.entity.ExecutionEntity字符串。默认为空。
- **maxStackDepth**: 限制在脚本中调用函数时的最大栈深度。可以用于避免由于递归调用脚本中定义的方法，而导致的栈溢出异常。默认为-1（禁用）。
- **maxScriptExecutionTime**: 脚本允许运行的最大时间。默认为-1（禁用）。
- **maxMemoryUsed**: 脚本允许使用的最大内存数量，以字节计。请注意脚本引擎自己也要需要一定量的内存，也会算在这里。默认为-1（禁用）。
- **nrOfInstructionsBeforeStateCheckCallback**: 脚本每执行x个指令，就通过回调函数进行一次最大脚本执行时间与内存检测。请注意这不是指脚本指令，而是指java字节码指令（这意味着一行脚本可能有上百行字节码指令）。默认为100。

请注意：maxMemoryUsed设置只能用于支持com.sun.management.ThreadMXBean#getThreadAllocatedBytes()方法的JVM。Oracle JDK支持它。

也有ScriptExecutionListener与ScriptTaskListener的安全形式：org.activiti.scripting.secure.listener.SecureJavascriptExecutionListener与org.activiti.scripting.secure.listener.SecureJavascriptTaskListener。

像这样使用：

```

1 <activiti:executionListener event="start"
2 class="org.activiti.scripting.secure.listener.SecureJavaScriptExecutionListener">
3   <activiti:field name="script">
4     <activiti:string>
5       <![CDATA[
6         execution.setVariable('test');
7       ]]>
8     </activiti:string>
9   </activiti:field>
10  <activiti:field name="language" stringValue="javascript" />
    </activiti:executionListener>

```

演示不安全脚本以及如何通过安全脚本功能将其变得安全的例子，可以查看[GitHub上的单元测试](#)

19. 使用Activiti-Crystalball仿真（试验性） Simulation with Activiti-Crystalball (Experimental)

19.1. 介绍 Introduction

19.1.1. 简介 Short overview

activiti-crystalball（CrystalBall）是一个用于Activiti业务流程管理平台的仿真引擎。CrystalBall仿真可以用于：

- 支持决策——对于生产工作流（例如，我们是否要在系统中增加更多资源，以满足截止日期？）。
- 优化与理解——测试变更并理解其影响。
- 培训——仿真可以在员工铺开使用前进行培训。
- ...

19.1.2. CrystalBall是独特的 CrystalBall is unique

不需要：

- 创建单独的仿真模型与引擎。
- 为仿真创建不同的报告。
- 为仿真引擎提供大量数据。

CrystalBall仿真器基于Activiti。这就是为什么可以很轻易地复制数据并启动仿真，也可以从历史中重放工作流行为。

19.2. 深入CrystalBall (CrystalBall inside)

CrystalBall是一个离散事件仿真器。最简单的实现是org.activiti.crystalball.simulator.SimpleSimulationRun。

```

1 init();
2
3 SimulationEvent event = removeSimulationEvent();
4
5 while (!simulationEnd(event)) {
6   executeEvent(event);
7   event = removeSimulationEvent();
8 }
9
10 close();

```

SimulationRun也可以执行不同来源生成的仿真事件（参见[回放](#)）。

19.3. 历史分析 History analysis

仿真的一个使用场景是用于分析历史。生产环境不能用于重现并查找bug。也几乎不可能将流程引擎调整到生产环境发生bug时一样的状态，问题不在硬件而是：

- 时间——流程实例可能需要好几个月。
- 并发——流程实例运行时可能与其它的实例相互影响，只有在它们都并行执行的时候才会发现问题。
- 用户——大量用户可以参与流程实例执行。将流程实例调整到错误发生时的状态会花费大量精力。

仿真可以容易地定位上面提到的问题。仿真的时间是虚拟的，不依赖于真实时间。Activiti流程引擎本身就是虚拟的，因此不需要创建用于仿真试验的虚拟流程引擎。在这种场景下并发也可以自然地定位。用户的行为可以记录并回放，回放可以按照记录，或者预测并按需生成。

最好的分析历史的方式是将其重现。重现在真实生活正很难实现，但可以通过仿真实现。

19.3.1. 历史中的事件 Events from the history.

重现历史的最重要的事情，是收集所有影响过系统状态的事件。假设我们的流程通过用户事件驱动（例如申领，完成任务……）。在这个场景下，我们可以使用两种事件源：

- 流程历史——当前只支持原始`activiti-crystalball`项目。
- 记录的活动事件。我们可以将`ActivitiEventListener`加入想要记录事件的引擎。记录的事件会被存储用于之后的分析。基础的实现是`org.activiti.crystalball.simulator.delegate.event.impl.InMemoryRecordActivitiEventListener`：

```
1  @Override
2  public void onEvent(ActivitiEvent event) {
3      Collection<SimulationEvent> simulationEvents = transform(event);
4      store(simulationEvents);
5  }
```

事件将被存储。我们可以继续再次重现历史。

19.3.2. 回放 PlayBack

回放的优点是可以一遍又一遍的播放，直到我们完全理解发生了什么。`Crystalball`仿真基于真实数据与真实用户行为，这是`Crystalball`的优势。

理解回放工作方式的最好的方法，是基于JUnit测试`org.activiti.crystalball.simulator.delegate.event.PlaybackRunTest`逐步地解释。测试仿真的流程是最简单的：

```
1  <process id="theSimplestProcess" name="Without task Process">
2      <documentation>This is a process for testing purposes</documentation>
3
4      <startEvent id="theStart"/>
5      <sequenceFlow id="flow1" sourceRef="theStart" targetRef="theEnd"/>
6      <endEvent id="theEnd"/>
7
8  </process>
```

流程已被部署，并将实际使用以及仿真运行。

- 记录事件

```
1  // 获取带有记录监听器的流程引擎来记录事件 get process engine with record listener to log events
2  ProcessEngine processEngine = (new RecordableProcessEngineFactory(THE_SIMPLEST_PROCESS, listener)).getObject();
3
4  // 使用变量启动流程实例 start process instance with variables
5  Map<String, Object> variables = new HashMap<String, Object>();
6  variables.put(TEST_VARIABLE, TEST_VALUE);
7  processEngine.getRuntimeService().startProcessInstanceByKey(SIMPLEST_PROCESS, BUSINESS_KEY, variables);
8
9  // 检查流程引擎状态——历史中应该有一个流程实例
10 // check process engine status - there should be one process instance in the history
11 checkStatus(processEngine);
12
13 // 关闭并销毁流程引擎 close and destroy process engine
14 EventRecorderTestUtils.closeProcessEngine(processEngine, listener);
15 ProcessEngines.destroy();
```

上面的代码片段将在`startProcessInstanceByKey`方法调用后记录`ActivitiEventType.ENTITY_CREATED`。

- 启动仿真运行 start simulation run

```
1  final SimpleSimulationRun.Builder builder = new SimpleSimulationRun.Builder();
2  // 初始化仿真运行 init simulation run
3  // 获取流程引擎工厂——与RecordableProcessEngineFactory的唯一区别是没有添加记录监听器
4  // get process engine factory - the only difference from RecordableProcessEngineFactory that log listener is not added
5  DefaultSimulationProcessEngineFactory simulationProcessEngineFactory = new
6  DefaultSimulationProcessEngineFactory(THE_SIMPLEST_PROCESS);
7  // 配置仿真运行 configure simulation run
8  builder.processEngine(simulationProcessEngineFactory)
9      // 从记录的事件设置回放事件日历 set playback event calendar from recorded events
10     .eventCalendar(new PlaybackEventCalendarFactory(new SimulationEventComparator(), listener.getSimulationEvents()))
11     // 为仿真事件设置处理器 set handlers for simulation events
12     .customEventHandlerMap(EventRecorderTestUtils.getHandlers());
13 SimpleSimulationRun simRun = builder.build();
14
15 simRun.execute(new NoExecutionVariableScope());
16
17 // 检查状态——在记录事件方法中使用的相同方法 check the status - the same method which was used in record events method
18 checkStatus(simulationProcessEngineFactory.getObject());
19
20 // 关闭并销毁流程引擎 close and destroy process engine
21
```

```
simRun.getProcessEngine().close();
ProcessEngines.destroy();
```

更高级的回放例子在org.activiti.crystalball.simulator.delegate.event.PlaybackProcessStartTest中

19.3.3. 流程引擎调试 Process engine debugger

回放限制了只能一下子执行所有仿真事件（例如启动流程，完成任务）。调试可以将执行分割为小步骤，并在步骤之间观察流程引擎的状态。

SimpleSimulationRun实现了SimulationDebugger接口。SimulationDebugger可以一步一步地执行仿真事件，将仿真运行到特定时间。

```
1  /**
2   * 可以在调试模式运行仿真 Allows to run simulation in debug mode
3   */
4  public interface SimulationDebugger {
5      /**
6       * 初始化仿真运行 initialize simulation run
7       * @param execution - 传递变量与运行仿真的变量范围 variable scope to transfer variables from and to simulation run
8       */
9      void init(VariableScope execution);
10
11     /**
12      * 前进一步仿真事件 step one simulation event forward
13      */
14     void step();
15
16     /**
17      * 继续仿真运行 continue in the simulation run
18      */
19     void runContinue();
20
21     /**
22      * 执行仿真运行直到simulationTime（仿真时间） execute simulation run till simulationTime
23      */
24     void runTo(long simulationTime);
25
26     /**
27      * 执行仿真运行直到特定类型的仿真事件 execute simulation run till simulation event of the specific type
28      */
29     void runTo(String simulationEventType);
30
31     /**
32      * 关闭仿真运行 close simulation run
33      */
34     void close();
35 }
```

要实际查看流程引擎调试器，运行SimpleSimulationRunTest

19.3.4. 重放 Replay

重放需要创建另一个流程引擎实例。回放并不影响“真实”环境，而是需要仿真试验配置。重放则运行在“真实”流程引擎上。重放在运行中的流程引擎上执行仿真事件。因此重放使用真实时间。真实时间意味着仿真事件被预定将会立即执行。

下面的例子展示了如何重放一个流程实例。同样的技术可以用在回放中，以播放一个流程实例。（ReplayRunTest）测试的第一部分初始化流程引擎，启动一个流程实例，并完成流程实例的任务。

```
1  ProcessEngine processEngine = initProcessEngine();
2
3  TaskService taskService = processEngine.getTaskService();
4  RuntimeService runtimeService = processEngine.getRuntimeService();
5
6  Map<String, Object> variables = new HashMap<String, Object>();
7  variables.put(TEST_VARIABLE, TEST_VALUE);
8  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(USER_TASK_PROCESS, BUSINESS_KEY,
9  variables);
10
11  Task task = taskService.createTaskQuery().taskDefinitionKey("userTask").singleResult();
12  TimeUnit.MILLISECONDS.sleep(50);
13  taskService.complete(task.getId());
```

使用的流程引擎是基础的InMemoryStandaloneProcessEngine以及

- InMemoryRecordActivitiEventListener（已经在回放中使用过）用于记录Activiti事件，并将其转换为仿真事件。
- UserTaskExecutionListener——当创建了新的用户任务，且新的任务是表单重放流程引擎时，将任务完成事件预定至事件日历。

测试的下一部分在原流程的相同流程引擎上启动了仿真调试器。重放事件处理器使用StartReplayProcessEventHandler取代了StartProcessEventHandler。StartReplayProcessEventHandler取得流程实例Id，将其用于重放以及在初始化阶段预定流程实例启动。

在处理阶段`StartProcessEventHandler`使用一个预留的变量启动新的流程实例。变量名字为"`_replay.processInstanceId`"。这个变量用于存储需要重放的流程的id。与`SimpleSimulationRun`相比，`ReplaySimulationRun`并不：

- 创建与关闭流程引擎实例。
- 修改仿真时间。（真实时间不能修改）

```
final SimulationDebugger simRun = new ReplaySimulationRun(processEngine,
    getReplayHandlers(processInstance.getId()));
```

现在重放流程实例可以启动了。在一开始流程实例还没有运行。在历史中已经有一个已完成的流程实例。在初始化之后，在事件日历中有一个仿真事件——用于启动一个流程实例，以重放已经完成的流程实例。

```
1  simRun.init();
2
3  // 原始流程已经完成—不应该有任何运行中的流程实例/任务
4  // original process is finished - there should not be any running process instance/task
5  assertEquals(0, runtimeService.createProcessInstanceQuery().processDefinitionKey(USERTASK_PROCESS).count());
6  assertEquals(0, taskService.createTaskQuery().taskDefinitionKey("userTask").count());
7
8  simRun.step();
9
10 // 重放流程已启动 replay process was started
11 assertEquals(1, runtimeService.createProcessInstanceQuery().processDefinitionKey(USERTASK_PROCESS).count());
12 // 应该有一个任务 there should be one task
13 assertEquals(1, taskService.createTaskQuery().taskDefinitionKey("userTask").count());
```

当创建任务时，`UserTaskExecutionListener`创建了新的仿真事件，以完成用户任务。

```
1  simRun.step();
2
3  // 用户任务已完成—重放流程已经完成 userTask was completed - replay process was finished
4  assertEquals(0, runtimeService.createProcessInstanceQuery().processDefinitionKey(USERTASK_PROCESS).count());
5  assertEquals(0, taskService.createTaskQuery().taskDefinitionKey("userTask").count());
```

仿真结束，我们可以继续另一个流程实例启动，或任何其他事件。这次我们关闭`simRun`与流程引擎。

```
1  simRun.close();
2  processEngine.close();
3  ProcessEngines.destroy();
```

20. 工具 Tooling

20.1. JMX

20.1.1. 介绍 Introduction

可以使用标准Java管理扩展（JMX）技术连接Activiti引擎，以获取信息或改变其行为。可以使用任何标准的JMX客户端。启用与禁用作业执行器、部署新的流程定义文件或删除它们等等操作，都可以通过JMX完成，而不需要写一行代码。

20.1.2. 快速开始 Quick Start

默认情况下没有启用JMX。要使用默认配置启动它，只要使用Maven或其他方法，将`activiti-jmx` jar文件加入classpath即可。如果使用Maven，可以在`pom.xml`中添加下列行以添加合适的依赖：

```
1  <dependency>
2    <groupId>org.activiti</groupId>
3    <artifactId>activiti-jmx</artifactId>
4    <version>latest.version</version>
5  </dependency>
```

在添加依赖并构建流程引擎后，就可以使用JMX连接了。可以使用在标准JDK发行版中提供的`jconsole`。在本地线程列表中，可以看到包含Activiti的JVM。如果由于任何原因没有在“本地进程”中列出合适的JVM，可以尝试使用这个URL从“远程进程”中连接：

```
service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/activiti
```

可以在日志文件中找到正确的本地URL。连接之后，可以看到标准JVM状态与MBeans。可以选择MBeans页签，并在右侧面板选择"`org.activiti.jmx.Mbeans`"，以浏览Activiti专用的MBean。选择任何MBean，都可以查询信息或修改配置。这个截图展示了`jconsole`的样子：

MBean	类型	默认值	名字	描述	描述
不只是jconsole，任何JMX客户端都可以访问MBeans。大多数数据中心监控工具都有可以连接至JMX MBeans的连接器。					

20.1.3. 属性与操作 Attributes and operations

这里有一个目前可用的属性与操作的列表。这个列表可能根据需要在未来版本中扩展。

MBean	类型	名字	描述
ProcessDefinitionsMBean	属性	processDefinitions	已部署流程定义的Id, Name, Version, IsSuspended参数，是一个字符串的list
	属性	deployments	当前部署的Id, Name, TenantId参数
	方法	getProcessDefinitionById(String id)	给定id流程定义的+Id+, Name, Version与IsSuspended参数
	方法	deleteDeployment(String id)	使用给定Id删除部署
	方法	suspendProcessDefinitionById(String id)	使用给定Id暂停流程定义
	方法	activatedProcessDefinitionById(String id)	使用给定Id激活流程定义
	方法	suspendProcessDefinitionByKey(String id)	使用给定key暂停流程定义
	方法	activatedProcessDefinitionByKey(String id)	使用给定key激活流程定义
	方法	deployProcessDefinition(String resourceName, String processDefinitionFile)	部署流程定义文件
JobExecutorMBean	属性	isJobExecutorActivated	作业执行器是否启用
	方法	setJobExecutorActivate(Boolean active)	按照给定boolean值启用或禁用作业执行器

20.1.4. 配置 Configuration

JMX使用最常使用的配置作为默认配置来简化部署。然而也可以很容易的修改默认配置，可以通过编程方式，或通过配置文件。下列代码展示了如何在配置文件中配置：

```
1 <bean id="processEngineConfiguration" class="...SomeProcessEngineConfigurationClass">
2   ...
3   <property name="configurators">
4     <list>
5       <bean class="org.activiti.management.jmx.JMXConfigurator">
6         <property name="connectorPort" value="1912" />
7         <property name="serviceUrlPath" value="/jmxrmi/activiti" />
8       </bean>
9     </list>
10    ...
11  </property>
12 </bean>
```

下表展示了可配置的参数与其默认值：

名字	默认值	描述
disabled	false	若值为true，即使已添加依赖也不会启动JMX

名字	默认值	描述
行 参数 domain	解释 org.activiti.jmx.Mbeans	MBean的域
createConnector	true	若值为true，则为已启动的MbeanServer创建一个连接器
MBeanDomain	DefaultDomain	MBean服务器的域
registryPort	1099	出现在服务URL中作为注册端口
serviceUrlPath	/jmxrmi/activiti	出现在服务URL中
connectorPort	-1	如果大于0，则出现在服务URL中作为连接端口

20.1.5. JMX服务URL (JMX Service URL)

JMX服务URL为下列格式：

```
service:jmx:rmi://<hostName>:<connectorPort>/jndi/rmi://<hostName>:<registryPort>/<serviceUrlPath>
```

`hostName`会自动设置为机器的网络名。可以配置`connectorPort`、`registryPort`与`serviceUrlPath`。

如果`connectionPort`小于0，则服务URL中没有对应的部分，简化为：

```
service:jmx:rmi:///jndi/rmi:///:<hostname>:<registryPort>/<serviceUrlPath>
```

20.2. Maven原型 Maven archetypes

20.2.1. 创建测试用例 Create Test Case

在开发过程中，有时在应用中实际实现前，创建一个小测试用例来测试想法或功能很有用，因为这样可以用测试来隔离主题。JUnit测试用例也是交流bug报告与功能需求的推荐工具。在一份bug报告或功能需求jira单中附加一个测试用例，可以有效减少修复时间。

为了便于创建测试用例，可以使用maven原型。通过使用这个原型，可以快速创建标准测试用例。原型应该在标准仓库中已经有了。如果没有，可以简单的安装在你的本地maven仓库目录中：在`tooling/archetypes`目录下键入`mvn install`。

下列命令创建单元测试项目：

```
mvn archetype:generate \
-DarchetypeGroupId=org.activiti \
-DarchetypeArtifactId=activiti-archetype-unittest \
-DarchetypeVersion=5.21.0 \
-DgroupId=org.myGroup \
-DartifactId=myArtifact
```

每个参数的效果在下表解释：

Table 264. 单元测试生成原型参数

行	参数	解释
1	archetypeGroupId	原型的Group id。需要为 org.activiti
2	archetypeArtifactId	原型的Artifact id。需要为 activiti-archetype-unittest
3	archetypeVersion	生成的测试项目中使用的Activiti版本
4	groupId	生成的测试项目的Group id
5	artifactId	生成的测试项目的Artifact id

生成的项目的目录结构像是这样：

```
.
├─ pom.xml
├─ src
│   └─ test
```



```
├─ java
│   └─ org
│       └─ myGroup
│           └─ MyUnitTest.java
└─ resources
    ├── activiti.cfg.xml
    ├── log4j.properties
    └─ org
        └─ myGroup
            └─ my-process.bpmn20.xml
```

可以修改java单元测试用例与其对应的流程模型，或者添加新的单元测试用例与流程模型。如果使用该项目来表述一个bug或功能，测试用例应该在初始时失败，然后在修复了预期的bug或实现了预期的功能以后成功。请确保在发送之前键入`mvn clean`清理项目。

20.3. Docker镜像 Docker images

20.3.1. 介绍 Introduction

Docker是一个神奇的虚拟化工具。可以将一个应用与其所有依赖打包为一个标准的软件开发单元。人们可以创建持久化的镜像，并将其发布至共享服务器。其他人可以获取该镜像，并轻松启动与运行该软件。

Activiti docker镜像的目的是快速运行activiti explorer与rest-api，而不需要为设置开发环境而处理复杂的细节。对于希望尽快运行、测试与使用Activiti的人来说十分有益。对展示演示也很有价值。

20.3.2. 使用 Usage

第一步是安装docker，如果还没有安装的话。[安装](#)十分简单明了。几乎所有主流OS都有可用的二进制安装文件。在安装Docker之后，可以使用这个命令，将activiti镜像与其依赖取回本地机器：

```
docker pull activiti/activiti-single-image:latest explorer
```

取决于Internet连接速度，可能需要一段时间下载镜像。只有第一次或共享服务器版本更新之后，才需要下载。

下一步是运行镜像与创建容器：

```
docker run -p 8080:8080 -t -i activiti/activiti-single-image:latest explorer
```

就是这样，镜像已经启动并运行了explorer web应用。可以使用这个URL访问：

```
localhost:8080/activiti-explorer
```

本地机器的端口号可以通过修改第二部分8080:8080参数修改。

最后的参数决定了运行的应用。有两个可选项：

- explorer
- rest

选择“rest”则会运行rest-api。例如这个URL会获取部署的列表：

```
http://kermit:kermit@localhost:8080/activiti-rest/service/repository/deployments
```

如果没有提供参数，默认为“explorer”。

20.3.3. 构建docker镜像 Build docker image

Docker使用tooling/dockerImage/singleImage文件夹下的Dockerfile构建。一个名为“buildImage.sh”的工具集合文件包含了构建镜像所需的命令行。如果构建的镜像需要由其他用户使用，需要使用这个命令将其发布至 docker共享服务器：

```
docker push activiti/activiti-single-image:latest
```