Red Hat Process Automation Manager 7.13

在 Red Hat Process Automation Manager 中开发进程服务

Last Updated: 2024-03-19

Red Hat Process Automation Manager 7.13 在 Red Hat Process Automation Manager 中开发进程服务

法律通告

摘要

本文档论述了如何使用业务流程模型和符号(BPMN)2.0 模型使用红帽流程自动化管理器开发流程服务和案例定义。本文档还描述了流程和问题单管理的概念和选项。

目录

前言

作为业务流程的开发人员，您可以使用 Red Hat Process Automation Manager 来使用业务流程模型和符号(BPMN)2.0 模型来开发流程服务和案例定义。BPMN 流程模型是实现业务目标所需步骤的图形化表示。有关 BPMN 的更多信息，请参阅对象管理组(OMG) 业务流程模型和 Notation 2.0 规格 。

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 *CTO Chris Wright* 的信息。

部分 I. 使用 BPMN 模型设计业务流程

作为业务流程开发人员，您可以在红帽流程自动化管理器或 VS Code 中的 Red Hat Process Automation Manager BPMN 模型器中使用 Business Central 来设计业务流程来满足特定的业务要求。本文档描述了业务流程以及使用 Red Hat Process Automation Manager 中的流程设计人员创建它们的概念和选项。本文档还描述了 Red Hat Process Automation Manager 中的 BPMN2 元素。有关 BPMN2 的详情，请查看业务流程模型和 Notation Version 2.0 规格。

先决条件

- Red Hat JBoss Enterprise Application Platform 7.4 已安装。详情请参阅 *Red Hat JBoss Enterprise Application Platform 7.4 安装指南*。
- Red Hat Process Automation Manager 由 KIE 服务器安装和配置。如需更多信息，*请参阅在 Red Hat JBoss EAP 7.4 上安装和配置 Red Hat Process Automation Manager*。
- Red Hat Process Automation Manager 正在运行，您可以使用 **开发人员** 角色登录到 Business Central。如需更多信息，请参阅 *规划 Red Hat Process Automation Manager 安装* 。

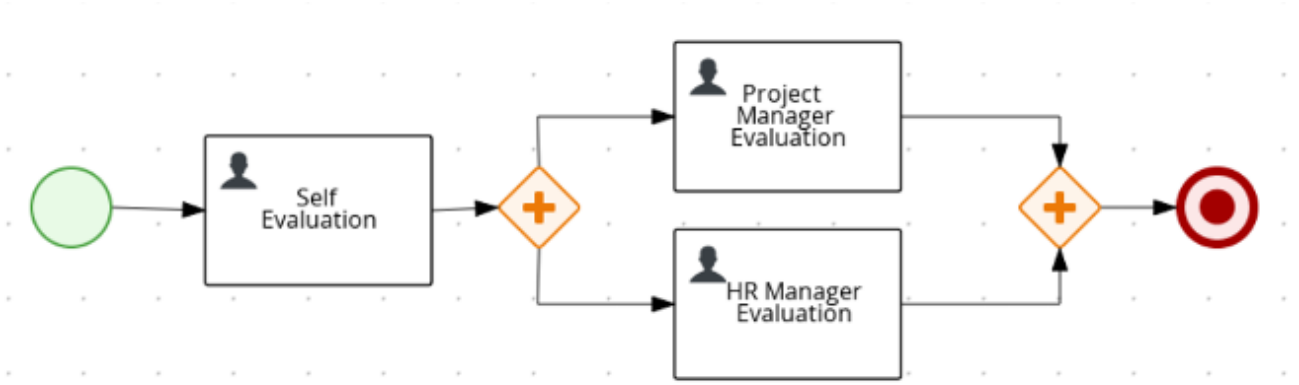
第 1 章 业务流程

业务流程是图表，描述必须执行一系列步骤的顺序，它由预定义的节点和连接组成。每个节点代表进程中的一个步骤，同时连接指定如何从一个节点转换到另一个节点。

典型的业务流程包括以下组件：

- 组成全局元素的标头部分，如进程的名称、导入和变量
- nodes 部分，其中包含属于进程一部分的所有不同节点
- 将这些节点链接到各个节点的连接部分来创建流图表

图 1.1. 业务流程



Red Hat Process Automation Manager 包含传统的流程设计人员以及用于创建业务流程图的新流程设计器。新的过程设计器具有改进的布局和功能集，并持续开发。在新的流程设计器中完全实施传统流程设计器的功能之前，您在 Business Central 中均提供两种设计人员供您使用。

注意

Business Central 中的传统流程设计程序在 Red Hat Process Automation Manager 7.13.5 中已弃用。在以后的 Red Hat Process Automation Manager 发行版本中会删除它。旧流程设计人员将不会再收到任何新的增强功能或功能。如果您打算使用新流程设计器，开始将进程迁移到新的设计人员。在新的进程设计器中创建所有新进程。有关迁移到新设计器的详情，请参考 *Business Central 中的管理项目*。

第 2 章 RED HAT PROCESS AUTOMATION MANAGER BPMN 和 DMN 型号器

Red Hat Process Automation Manager 提供了以下扩展程序或应用程序，您可以使用它们设计业务流程模型和符号(BPMN)流程模型，以及使用图形模型(DMN)决策模型。

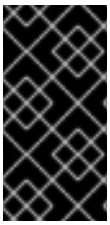
- **Business Central** ：使您能够在相关嵌入式设计人员中查看和设计 BPMN 模型、DMN 模型和测试方案文件。
  要使用 Business Central，您可以设置一个包含 Business Central 的开发环境，用于设计业务规则和流程，以及 KIE 服务器来执行和测试创建的业务规则和流程。

- **Red Hat Process Automation Manager VS Code 扩展**：允许您在 Visual Studio Code(VS Code)中查看和设计 BPMN 模型、DMN 模型和测试场景文件。VS Code 扩展需要 VS Code 1.46.0 或更新版本。
  要安装 Red Hat Process Automation Manager VS Code 扩展，请在 VS Code 中选择 **Extensions** 菜单选项，并搜索并安装 **Red Hat Business Automation Bundle**扩展。

- **独立 BPMN 和 DMN 编辑器**：使您能够查看和设计嵌入在 web 应用程序中的 BPMN 和 DMN 模型。要下载所需的文件，您可以使用 NPM registry 中的 NPM 工件，或直接下载位于 **https://<YOUR_PAGE>/dmn/index.js** 的 DMN 独立编辑器库的 JavaScript 文件。

2.1. 安装 RED HAT PROCESS AUTOMATION MANAGER VS CODE 扩展捆绑包

Red Hat Process Automation Manager 提供了一个 Red Hat Network Automation Bundle VS Code 扩展，它可让您设计决策模型和符号(DMN)决策模型、业务流程模型和符号(BPMN)2.0 业务流程并直接在 VS Code 中测试场景。VS Code 是开发新业务应用的首选集成开发环境(IDE)。Red Hat Process Automation Manager 还提供单独的 DMN Editor 和 BPMN Editor VS Code 扩展（如果需要）。



重要

VS Code 中的编辑器部分与 Business Central 中的编辑器兼容，并且 VS Code 不支持几个 Business Central 功能。

先决条件

- **安装了 VS Code 的最新稳定版本。**

流程

1. **在 VS Code IDE 中，选择 Extensions 菜单选项，并搜索 Red Hat Business Automation Bundle for DMN、Harllse 和 test scenario 文件支持。**

   对于 DMN 或 BPMN 文件支持，您还可以搜索单独的 DMN Editor 或 BPMN Editor 扩展。

2. 当 Red Hat Business Automation Bundle 扩展出现在 VS Code 中时，选择它并点 Install。

3. 要获得最佳 VS Code 编辑器行为，请在扩展安装完成后，重新加载或关闭并重新启动 VS Code 实例。

安装 VS Code 扩展捆绑包后，任何 .dmn、.bpmn 或 .bpmn2 文件都会自动显示为图形模型。此外，您打开或创建的 .scesim 文件自动显示为表格测试场景模型，用于测试您的业务决策功能。

如果 DMN、CEP 或测试场景模型器只打开 DMN、Hardb 或 test scenario 文件的 XML 源，并显示错误消息，请查看报告的错误和模型文件，以确保定义所有元素。

注意

对于新的 DMN 或 BPMN 模型，您还可以在网页浏览器中输入 dmn.new 或 BPMn.new，以在在线模型程序中设计 DMN 或 BPMN 模型。完成创建模型后，您可以点击 Download in the online modeler 页面将 DMN 或 BPMN 文件导入到 VS Code 中的 Red Hat Process Automation Manager 项目中。

2.2. 配置 RED HAT PROCESS AUTOMATION MANAGER 独立编辑器

Red Hat Process Automation Manager 提供独立编辑器，这些编辑器在自包含的库中分发，为每个编辑器提供一个一体化 JavaScript 文件。JavaScript 文件使用全面的 API 来设置和控制编辑器。

您可以使用以下方法安装独立编辑器：

- 手动下载每个 JavaScript 文件
- 使用 NPM 软件包

流程

1. 使用以下方法之一安装独立编辑器：

手动下载每个 JavaScript 文件 ： 对于这个方法，请按照以下步骤操作：

a. **下载 JavaScript 文件。**

b. **将下载的 Javascript 文件添加到您的托管应用程序中。**

c. **将以下 &lt ;script& gt; 标签添加到 HTML 页面：**

DMN 编辑器的 HTML 页面标记

```
<script src="https://<YOUR_PAGE>/dmn/index.js"></script>
```

BPMN 编辑器的 HTML 页面的脚本标签

```
<script src="https://<YOUR_PAGE>/bpmn/index.js"></script>
```

使用 NPM 软件包 ： 对于这个方法，请按照以下步骤操作：

a. **在 package.json 文件中添加 NPM 软件包：**

添加 NPM 软件包

```
npm install @kie-tools/kie-editors-standalone
```

b.

将每个编辑器库导入到 TypeScript 文件：

导入每个编辑器

```
import * as DmnEditor from "@kie-tools/kie-editors-standalone/dist/dmn"
import * as BpmnEditor from "@kie-tools/kie-editors-standalone/dist/bpmn"
```

2.

安装独立编辑器后，使用提供的编辑器 API 打开所需的编辑器，如下例所示，打开 DMN 编辑器。每个编辑器的 API 相同。

打开 DMN 独立编辑器

```
const editor = DmnEditor.open({
  container: document.getElementById("dmn-editor-container"),
  initialContent: Promise.resolve(""),
  readOnly: false,
  origin: "",
  resources: new Map([
    [
      "MyIncludedModel.dmn",
      {
        contentType: "text",
        content: Promise.resolve("")
      }
    ]
  ])
});
```

在 editor API 中使用以下参数：

表 2.1. 示例参数

参数	描述
container	附加编辑器的 HTML 元素。

参数	描述
initialContent	对 DMN 模型内容的承诺。这个参数可以为空，如下例所示： • **Promise.resolve("")** • **Promise.resolve("<DIAGRAM_CONTENT_DIRECTLY_HERE>")** • **fetch("MyDmnModel.dmn").then(content ⇒ content.text())**
ReadOnly （可选）	可让您在编辑器中允许更改。在编辑器中将 设置为 **false**（默认）以允许内容编辑和 **true**。
Origin （可选）	仓库的起源。默认值为 **window.location.origin**。
资源 （可选）	编辑器的资源映射。例如，这个参数用于为 DMN 编辑器提供包含的模型，或为 BPMN 编辑器提供工作项目定义。映射中的每个条目都包含一个资源名称和一个对象，它由 **content -type**（**文本 或二进制 ）和内容**组成（与 **初始 Content** 参数类似）。

返回的对象包含操作编辑器所需的方法。

表 2.2. 返回的对象方法

方法	描述
getContent（）: Promise<string>	返回包含编辑器内容的保证。
setContent(path: string, content: string): void	设置编辑器的内容。
getPreview（）: Promise<string>	返回包含当前图表的 SVG 字符串的保证。
subscribeToContentChanges（callback:(isDirty: boolean)IFL void）:(isDirty: boolean)void	当编辑器中的内容更改并返回用于 unsubscription 的回调时，设置要调用的回调。
subscribe toContentChanges（callback: (isDirty: boolean)IFL void）: void	当内容在编辑器中更改时，取消订阅传递的回调。

方法	描述
markAsSaved(): void	重置编辑器状态，这表示已保存编辑器中的内容。另外，它会激活与内容更改相关的订阅回调。
undo（）: void	在编辑器中取消上次更改。另外，它会激活与内容更改相关的订阅回调。
redo（）: void	在编辑器中恢复最近一次撤消的更改。另外，它会激活与内容更改相关的订阅回调。
close（）: void	关闭编辑器。
getElementPosition(selector: string): Promise<Rect>	提供了一种替代方式，可以在可清空或视频组件中的元素中时扩展标准查询选择器。**选择器** 参数必须遵循 <targetNamespaces **>:::<SELECT** > 格式，如 **Canvas:::MySquare** 或 **Video:::PresenterHand**。此方法返回一个代表元素位置的 **Rect**。
envelopeApi: MessageBusClientApi<KogitoEditorEnvelopeApi>	这是高级编辑器 API。有关高级编辑器 API 的更多信息，请参阅 MessageBusClientApi 和 KogitoEditorEnvelopeApi。

第 3 章 使用 MAVEN 创建和执行 DMN 和 BPMN 模型

您可以使用 Maven archetypes 使用 Red Hat Process Automation Manager VS Code 扩展（而非 Business Central）在 VS Code 中开发 DMN 和 BPMN 模型。然后，您可以根据需要将您的 archetypes 与 Red Hat Process Automation Manager 决策和流程服务集成。这种开发 DMN 和 BPMN 模型的方法对使用 Red Hat Process Automation Manager VS Code 扩展构建新的业务应用程序会很有帮助。

流程

1. 在命令终端中，导航到用于存储新 Red Hat Process Automation Manager 项目的本地文件夹。

2. 输入以下命令使用 Maven archtype 在定义的文件夹中生成项目：

使用 Maven archetype 生成项目

```
mvn archetype:generate \
    -DarchetypeGroupId=org.kie \
    -DarchetypeArtifactId=kie-kjar-archetype \
    -DarchetypeVersion=7.67.0.Final-redhat-00024
```

此命令生成包含所需依赖项的 Maven 项目，并生成所需的目录和文件以构建您的业务应用程序。在开发项目时，您可以使用 Git 版本控制系统（推荐）。

如果要在同一目录中生成多个项目，请通过将 -DgroupId= -D artifactId ={> -DartifactId= <artifactId> 添加到 上一个命令来指定生成的业务应用程序的 artifactId 和 groupId。

3. 在 VS Code IDE 中，单击 File，选择 Open Folder，再导航到使用上一命令生成的文件夹。

4. 在创建第一个资产前，请为您的业务应用程序设置一个软件包，例如 org.kie.Businessapp，并在以下路径中创建相应的目录：

- PROJECT_HOME/src/main/java

- PROJECT_HOME/src/main/resources
- PROJECT_HOME/src/test/resources

例如，您可以为 org.kie . Businessapp 创建 PROJECT_HOME/src/main/java /org/kie/businessapp 软件包。

5. 使用 VS Code 为您的业务应用程序创建资产。您可以使用以下方法创建由 Red Hat Process Automation Manager VS Code 扩展支持的资产：

- 要创建业务流程，请在 PROJECT_HOME/src/main/resources/org/kie/businessapp 目录（如 Process .bpmn ）中使用 .bpmn 或 .bpmn 创建新文件。
- 要创建 DMN 模型，请在 PROJECT_HOME/src/main/resources/org/kie/businessapp 目录中创建一个使用 .dmn 的新文件，如 AgeDecision.dmn。
- 要创建测试场景模拟模型，请在 PROJECT_HOME/src/test/resources/org/kie/businessapp 目录中创建一个带有 .scesim 的新文件，如 TestAgeScenario.scesim。

6. 在 Maven archetype 中创建资产后，导航到命令行中项目的 root 目录（包含 pom.xml），再运行以下命令来构建项目的知识库文章(KJAR)：

```
mvn clean install
```

如果构建失败，请解决命令行错误消息中描述的任何问题，并尝试验证项目，直到构建成功为止。但是，如果构建成功，您可以在 PROJECT_HOME/target 目录中找到业务应用程序的工件。

注意

在开发的每一主要更改后，经常使用 mvn clean install 命令验证您的项目。

您可以使用 REST API 在运行的 KIE 服务器上部署您业务应用程序生成的知识 JAR(KJAR)。有关使用 REST API 的更多信息，请参阅使用 *KIE API 与 Red Hat Process Automation Manager 交互*。

第 4 章 业务流程建模和通知版本 2.0

业务流程建模与 Notation Version 2.0(BPMN2)规范是一个对象管理组(OMG)规格，定义图形表示业务流程的标准，定义元素的执行语义，采用 XML 格式提供进程定义。

进程由进程定义定义或确定。它存在于知识库中，并由其 ID 标识。

表 4.1. 常规进程属性

标签	描述
Name	输入进程的名称。
Documentation	描述进程。此字段中的文本包括在流程文档中（如果适用）。
ID	输入此进程的标识符，如 **orderItems**。
软件包	在 Red Hat Process Automation Manager 项目中输入此过程的软件包位置，如 **org.acme**。
ProcessType	指定进程是公共还是私有。（目前不支持。）
版本	输入进程的构件版本。
临时	如果此进程是一个临时子进程，请选择这个选项。
进程实例描述	输入进程用途的描述。
导入	点击以打开 Imports 窗口，再添加所需的任何数据类型类，也可以选择已定义的数据类型。
可执行	选择这个选项使 Red Hat Process Automation Manager 项目的进程可执行。
SLA 过期日期	输入服务级别协议(SLA)过期日期。
进程变量	为进程添加任何进程变量。进程变量在特定的进程实例中可见。在完成进程创建和销毁时初始化进程变量。变量标签提供对变量行为的更大控制，例如，变量是否标记为 **必需** 还是只读。**有关变量标签的更多信息，请参阅 第 6 章 *变量*。**
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。

标签	描述
全局变量	为进程添加任何全局变量。全局变量对项目中的所有进程实例和资产可见。全局变量通常由业务规则和约束使用，并由规则或约束动态创建。

进程是一组建模元素的容器。它包含通过流对象和流指定业务流程的执行工作流的元素。每个进程都具有自己的 BPMN2 图表。Red Hat Process Automation Manager 包含用于创建 BPMN2 图的新流程设计器，可通过 .bpmn2 扩展打开旧的 BPMN2 图表。新的流程设计器具有改进的布局和功能集，并继续开发。默认情况下，新进程设计器中会创建新的图表。

4.1. RED HAT PROCESS AUTOMATION MANAGER 对 BPMN2 的支持

在 Red Hat Process Automation Manager 中，您可以使用 BPMN 2.0 标准对业务流程建模。然后，您可以使用 Red Hat Process Automation Manager 运行、管理和监控这些业务流程。完整的 BPMN 2.0 规范还包括有关如何代表业务和协作等项目的详细信息。但是，Red Hat Process Automation Manager 只使用可用于指定可执行进程的规范部分。其中包括在 BPMN2 规范的 Common Executable 子类中定义的所有元素和属性，这些元素和属性会扩展一些额外的元素和属性。

下表包含用来指示传统进程设计器、传统和新的进程设计器是否支持 BPMN2 元素的图标列表，还是不支持。

表 4.2. 支持状态图标

键	描述
	在传统和新的流程设计程序中支持
	仅支持旧进程设计器
	不支持

BPMN2 规格中没有图标的元素。

表 4.3. BPMN2 捕获事件

元素名称	Start	Intermediate
无		
消息		
timer		
Error		
escalation		
取消		
补偿		
条件		
Link		
信号		
Multiple		
并行多个		

表 4.4. BPMN2 抛出和非中断事件

元素名称	抛出		非中断	
	结束	Intermediate	Start	Intermediate
无	✓			
消息	✓	✓	✓	✓
timer			✓	✓
Error	✓			
escalation	✓	✓	✓	✓
取消	✗	✗		✗
补偿	✓	✓		
条件			✓	✓
Link		✓		
信号	✓	✓	✓	✓
终止	✓			

元素名称	抛出		非中断	
Multiple	✗	✗	✗	✗
并行多个			✗	✗

表 4.5. BPMN2 元素

元素类型	元素	支持
任务	业务规则	✓
	脚本	✓
	用户任务	✓
	服务任务	✓
子进程，包括多个实例子进程	嵌入式	✓
	临时	✓
	reusable	✓
	事件	✓

元素类型	元素	支持
网关	含	
	专用	
	并行	
	基于事件	
	Complex	
连接对象	序列流程	
	关联流程	
Swimlanes	Swimlanes	
工件	组	
	文本注解	
	数据对象	

有关 BPMN2 的后台和应用程序的详情，请查看 OMG 业务流程模型和符号(BPMN)版本 2.0 规格。

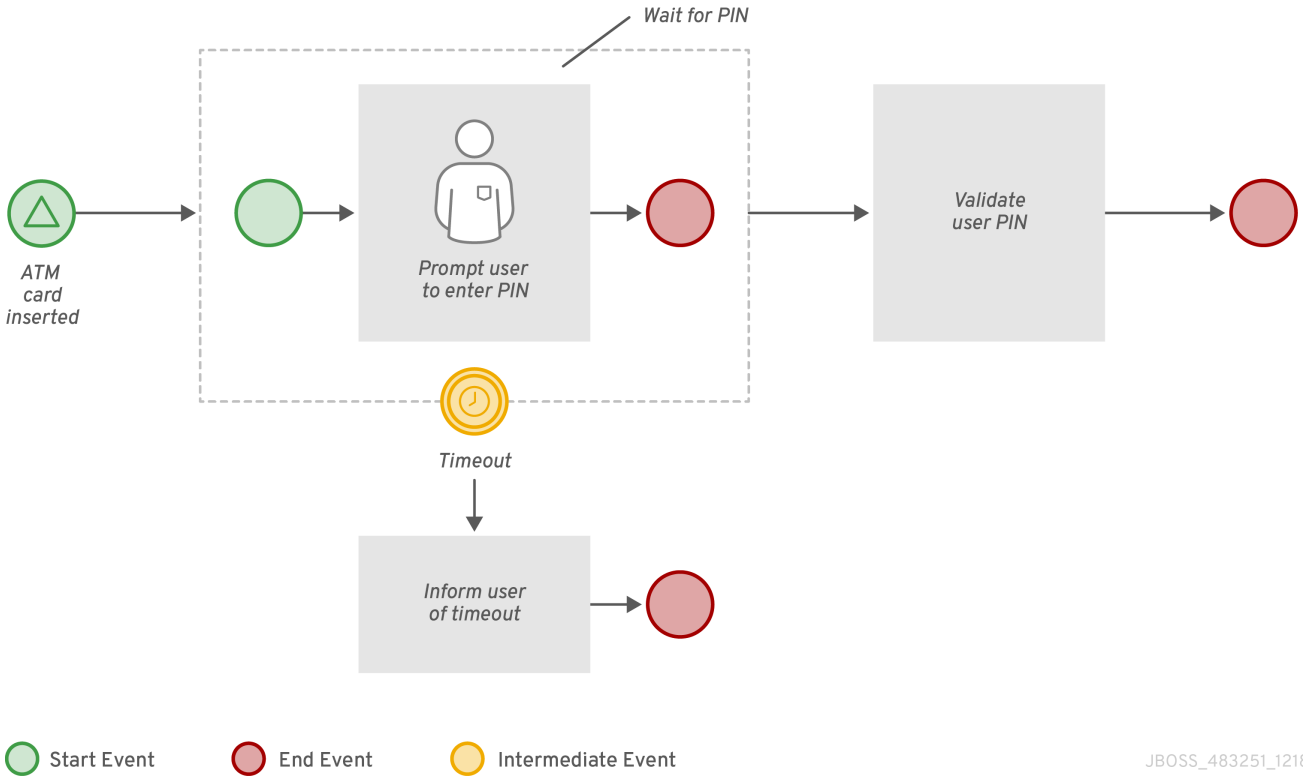
4.2. 流程设计器中的 BPMN2 事件

一个事件是对业务流程发生的情况。BPMN2 支持三种事件类别：

- Start
- 结束
- Intermediate

起始事件捕获事件触发器、结束事件事件触发，中间事件可能会捕获和引发事件触发器。

以下过程图显示了事件示例：



在本例中，会出现以下事件：

- 当收到信号时，会触发 ATM 卡 Inserted 信号启动事件。

- 超时中间事件是基于计时器触发器的中断事件。这意味着，当触发计时器事件时，Wait for PIN 子进程会被取消。

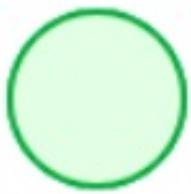
- 根据进程的输入，与 Validate User Pin 任务关联的结束事件，或者与 Timeout 任务关联的最终事件结束进程。

4.2.1. 启动事件

使用 start events 来指示业务流程的开始。启动事件无法有传入的序列流，且必须只有一个传出序列流。您可以在顶级进程、嵌入式子进程、可调用的子进程和事件子进程中使用任何启动事件。

除 none 起始事件外，所有启动事件都会捕获事件。例如，只有在收到引用的信号（事件触发器）时，信号启动事件才会启动该进程。您可以在事件子进程中配置启动事件，以中断或非中断。事件子进程的中断启动事件停止或中断包含或父进程的执行。非中断启动事件不会停止或中断包含或父进程的执行。

表 4.6. 启动事件

启动事件类型	顶级	sub-processes	
		中断	非中断
无			
条件			
补偿			

启动事件类型	顶级	sub-processes	
Error			
escalation			
消息			
信号			
timer			

无

none 启动事件是一个没有触发器条件的启动事件。进程或子进程最多可以包含一个无启动事件，默认情况下在进程或子进程启动时触发该事件，并且传出流将立即执行。

当您在子进程中使用任何启动事件时，进程流的执行将从父进程传输到子进程中，并且不触发启动事件。这意味着，令牌（进程流中的当前位置）从父进程传递到子进程活动，并且没有子进程的开始事件生成自己的令牌。

条件

条件启动事件是一个带有布尔值条件定义的开始事件。当条件首次被评估为 false 时，会触发执行，然后变为 true。只有在实例化开始后该条件被评估为 true 时，进程执行才会开始。

进程可以包含多个条件启动事件。

补偿

使用子进程作为中间事件的目标活动时，使用联合开始事件提交事件子进程。

Error

进程或子进程可以包含多个错误启动事件，当收到带有特定 ErrorRef 属性的错误对象时，会触发这些事件。错误对象可以通过错误结束事件生成。它表示进程结束不正确。收到相应错误对象后，错误启动事件的进程实例将开始执行。在收到错误对象及其传出流时，会立即执行错误启动事件。

escalation

升级启动事件是由升级具有特定升级代码触发的开始事件。进程可以包含多个升级开始事件。当它收到定义的升级对象时，带有升级的进程实例会启动其执行。进程实例化，升级启动事件将立即执行，并收集其传出流。

消息

进程或事件子进程可以包含多个消息启动事件，这些事件由特定消息触发。带有消息启动事件的进程实例仅在收到相应消息后从此事件开始执行。收到消息后，进程会被实例化，其消息启动事件会立即执行（执行传出流）。

由于消息可以通过任意数量的进程和流程元素消耗，包括没有元素，一个消息可以触发多个消息启动事件，因此可以实例化多个进程。

信号

信号启动事件由带有特定信号代码的信号触发。进程可以包含多个信号启动事件。信号启动事件仅在实例收到对应的信号后在进程实例中开始执行。然后，执行信号启动事件，并且执行其传出流。

timer

timer 启动事件是一个带有计时机制的启动事件。进程可以包含多个计时器启动事件，这些事件在进程开始时触发，然后应用计时机制。

当您在子进程中使用计时器启动事件时，将进程流执行从父进程传输到子进程，并且触发计时器启动事件。令牌取自父子进程活动，并且触发了子进程的计时器事件并等待计时器触发。在达到计时定义定义的时间后，将执行传出流。

4.2.2. 中间事件

中间事件驱动业务流程的流。中间事件用于在执行业务流程期间捕获或抛出事件。这些事件放置在开始和结束事件之间，还可用于活动边界，如子进程或人工任务，作为捕获事件。在 BPMN 模型器中，您可以在 Data Output 和 Assignments 字段中为 边界事件设置数据输出，在进一步处理中用于访问进程实例详情。请注意，编译事件不支持设置数据输出变量的功能。

例如，您可以为边界事件设置以下数据输出变量：

- NodeInstance ：计算触发边界事件时要在进一步使用的节点实例详情。
- 信号 ：指定信号的名称。
- 事件 ：计算事件详情。
- workItem ：计算工作项目详情。可以为工作项目或用户任务设置此变量。

边界捕获事件可以配置为中断或非中断。中断边界捕获事件会取消绑定的活动，而非中断事件则不会。

中间事件处理进程执行过程中发生的特定情况。对于中间事件，这种情况是触发器。在进程中，带有外向流的中间事件可以放置在活动边界上。

如果在活动执行过程中发生该事件，则事件会触发其对传出流的执行。一个活动可能有多个边界事件。请注意，根据您在边界中间事件的活动所要求的行为，您可以使用以下中间事件类型：

- 中断：活动执行中断，并触发中间事件的执行。

- **非中断：触发中间事件，活动执行将继续。**

表 4.7. 中间事件

中间事件类型	捕获	边界		抛出
		中断	非中断	
消息				
timer				
Error				
信号				
条件				
补偿				

中间事件类型	捕获	边界		抛出
escalation				
Link				

消息

消息中间事件是一个中间事件，供您用于管理消息对象。使用以下事件之一：

- **抛出消息中间事件根据定义的属性生成消息对象。**
- **捕获消息中间事件，侦听带有定义的属性的消息对象。**

timer

计时器中间事件允许您延迟工作流执行或定期触发工作流执行。它代表一个计时器，可在指定时间段内触发一次或多次。当触发计时器中间事件时，计时器条件（即已定义的时间）将进行检查，并且执行传出流。当计时器中间事件放置在进程工作流中时，它有一个传入的流和一个传出的流。当进入到事件的流传输时，其执行会启动。当计时器中间事件放在活动边界上时，将同时在活动执行时触发执行。

如果计时器元素被取消，如完成或中止包含的进程实例，则会取消计时器。

条件

条件中间事件是一个中间事件，其布尔值条件作为其触发器。当条件评估为 true 及其传出流时，事件会触发进一步的工作流执行。

事件必须定义 Expression 属性。当条件中间事件放入进程工作流中时，它有一个传入的流、一个传出流，并在进入事件的传输时开始执行。当将一个条件中间事件放在活动边界上时，将同时在活动执行的同时触发执行。请注意，如果事件没有中断，则事件触发器在条件为 true 时持续触发。

信号

信号中间事件允许您生成或消耗信号对象。使用以下选项之一：

- 引发信号中间事件根据定义的属性生成信号对象。
- 捕获信号中间事件会侦听带有定义的属性的信号对象。

Error

错误中间事件是一个中间事件，只能用于活动边界。它可让进程响应相应活动中的错误结束事件。活动不能是原子的。当活动完成错误事件时，生成带有相应 ErrorCode 属性的错误对象时，错误中间事件会捕获错误对象并执行会继续传出流。

补偿

补偿中间事件是附加到事务子进程中活动的边界事件。它可以通过处理结束事件或取消结束事件完成。补偿中间事件必须与一个流关联，该流连接至补偿活动。

如果交易子进程使用编译最终事件完成，则执行与边界中间事件相关的活动。执行继续进行相应的流。

escalation

升级中间事件是一个中间事件，供您生成或消耗升级对象。根据事件元素应该执行的操作，您需要使用以下选项之一：

- 丢弃升级中间事件会根据定义的属性生成升级对象。
- 捕获升级中间事件，侦听带有定义的属性的升级对象。

Link

链接中间事件是一个中间事件，使进程图表更易于理解，而无需向进程添加其他逻辑。链接中间事件仅限于单个进程级别，例如，链接中间事件无法与子进程连接。

使用以下选项之一：

- 丢弃链接中间事件根据定义的属性生成链接对象。
- 捕获链路中间事件，侦听带有定义的属性的链接对象。

4.2.3. 结束事件

最终事件用于结束业务流程，并且可能没有任何传出序列流程。业务流程中可能存在多个结束事件。除 none 和终止结束事件外，所有结束事件都会抛出事件。

结束事件表示完成业务流程。最终事件是结束特定工作流的节点。它具有一个或多个传入序列流，没有传出的流。

进程必须至少包含一个结束事件。

在运行时，最终事件会结束进程工作流。最终事件只能完成到达它的工作流，或者进程实例中的所有工作流，具体取决于最终事件类型。

表 4.8. 结束事件

结束事件	图标
无	
消息	
信号	

结束事件	图标
Error	
补偿	
escalation	
终止	

无

none 结束事件指定没有关联进程末尾的任何其他特殊行为。

消息

当流进入消息结束事件时，流完成并且最终事件生成其属性中定义的消息。

信号

引发信号结束事件用于完成进程或子进程流。当执行流进入元素时，执行流将完成并生成由其 SignalRef 属性标识的信号。

Error

引发错误最终事件会完成传入的工作流，这意味着消耗传入的令牌，并产生错误对象。进程或子进程中的其他运行工作流均保持未影响。

补偿

补偿最终事件用于完成事务子进程，并触发附加至子进程活动中编译中间事件的编译。

escalation

升级最终事件完成传入的工作流，这意味着消耗传入的令牌，并生成升级信号，如其属性中定义的升级信号，触发升级过程。

终止

终止最终事件完成指定进程实例中的所有执行流程。正在执行的活动将被取消。如果子进程实例到达终止最终事件，则子进程实例会终止。

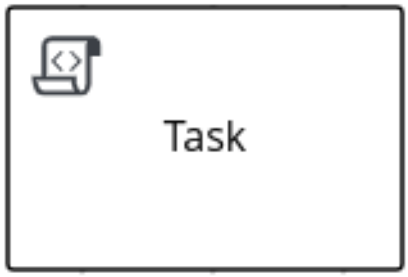
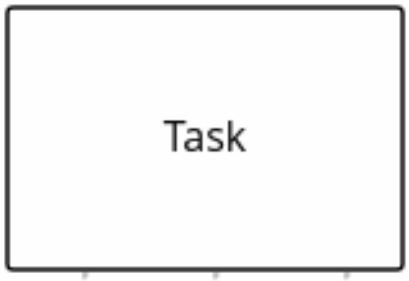
4.3. 流程设计器中的 BPMN2 任务

任务是进程模型中定义的自动活动，也是进程流中工作的最小单元。在 Red Hat Process Automation Manager 进程设计器中提供了 BPMN2 规范中定义的以下任务类型：

- 业务规则任务
- 脚本任务
- 用户任务
- 服务任务
- none 任务

表 4.9. 任务

业务规则任务	

脚本任务	
用户任务	
服务任务	
none 任务	

另外，BPMN2 规格还具备创建自定义任务的能力。有关自定义任务的更多信息，请参阅 第 4.4 节 “Process designer 中的 BPMN2 自定义任务”。

业务规则任务

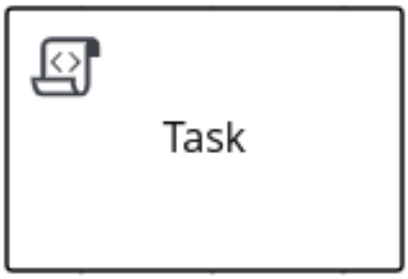
业务规则任务定义了通过 DMN 模型或规则流组做出决策的方法。

当某个进程达到 DMN 模型定义的商业规则任务时，进程引擎会通过提供的输入来执行 DMN 模型决定。

当某个进程到达由规则流组定义的商业规则任务时，进程引擎将开始执行定义的规则流组中的规则。当规则流组中没有更多活跃的规则时，执行将继续到下一元素。在规则流执行期间，可将属于活动规则流组的新激活添加到日程表，因为这些激活已由其他规则更改。

脚本任务

script 任务表示在进程执行期间要执行的脚本。



关联的脚本可以访问进程变量和全局变量。在使用 script 任务前查看以下列表：

- 避免流程中的低级别实施细节。script 任务可用于操作变量，但请考虑在处理更复杂的操作时使用服务任务或自定义任务。
- 确保脚本立即执行，否则使用异步服务任务。
- 避免通过脚本任务联系外部服务。使用服务任务模拟与外部服务的通信。
- 确保脚本不会抛出异常。例如，在该脚本内，应当发现和管理运行时例外，或者转换为可以在进程内处理的信号或错误。

在执行期间到达脚本任务时，将执行该脚本并执行传出流。

用户任务

用户任务是进程工作流中无法由系统自动执行的任务，因此需要人为人类用户干预。

在执行时，User 任务元素作为任务实例化，这些任务出现在一个或多个操作器的任务列表中。如果 User 任务元素定义了 Groups 属性，它将显示在属于该组成员的所有用户的任务列表中。组成员的任何用户都可以声明该任务。

声明后，任务会从其他用户的任务列表中消失。

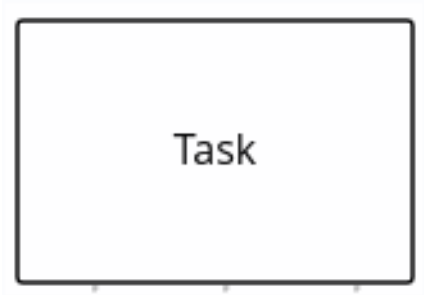
用户任务作为域特定任务实施，作为自定义任务的基础。

服务任务

服务任务是不需要人为干预的任务。它们由外部软件服务自动完成。

none 任务

没有任务在激活时完成。这只适用于概念模型。一个没有任务并不实际由 IT 系统执行。

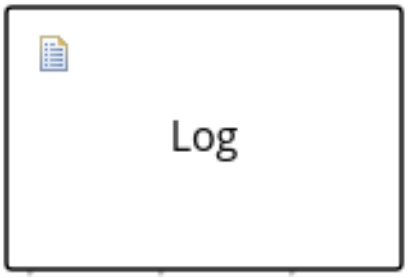


4.4. PROCESS DESIGNER 中的 BPMN2 自定义任务

BPMN2 规范支持扩展 BPMn2:task 元素以在软件实施中创建自定义任务。与标准 BPMN 任务类似，自定义任务标识要在业务流程模型中完成的操作，但它们还包括专门功能，如与特定类型的外部服务（REST、电子邮件或 Web 服务）的兼容性或检查流程中的行为(milestone)。

Red Hat Process Automation Manager 在 BPMN modeler palette 中的 Custom Tasks 下提供以下预定义的自定义任务：

表 4.10. 支持的自定义任务

自定义任务类型	自定义任务节点
REST	
电子邮件	
Log	

自定义任务类型	自定义任务节点
WebService	
milestone	
DecisionTask	
BusinessRuleTask	
KafkaPublishMessages	

有关在 Business Central 中启用或禁用自定义任务的详情，请参考 第 58 章 。

在 BPMN 模型器中，您可以为所选自定义任务配置以下通用属性：

表 4.11. 常规自定义任务属性

标签	描述
Name	标识任务的名称。您还可以双击任务节点来编辑名称。
Documentation	描述任务。此字段中的文本包括在流程文档中（如果适用）。
is Async	确定此任务是否异步调用。
adhoc Autostart	决定这是自动启动的临时任务。这个选项可让任务在进程创建时自动启动，而不是通过信号事件启动。
On Entry Action	定义一个以 Java、JavaScript 或 MVEL 脚本在任务开始时直接执行操作。
在退出操作中	定义一个 Java、JavaScript 或 MVEL 脚本，用于在任务结束时直接执行操作。
SLA 过期日期	当服务级别协议(SLA)过期时，指定持续时间（字符串类型）。您可以在天、分钟、秒和毫秒为单位。例如，**SLA 因为日期** 字段中的 **1m** 值代表一分钟。
分配	定义任务的数据输入和输出。
元数据属性	定义要用于自定义事件监听程序的自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。 **Metadata Attributes** 可启用对 BPMN 图表的新 **metaData** 扩展，并修改整个任务的行为。

REST

其余的自定义任务用于调用远程 RESTful 服务，或者从进程执行 HTTP 请求。

要使用剩余的自定义任务，您可以在进程模型器中设置 URL、HTTP 方法和凭证。当进程到达剩余的自定义任务时，它会生成一个 HTTP 请求，并以字符串形式返回响应。

您可以在 Properties 面板中点击 Assignments 以打开 REST Data I/O 窗口。在 REST Data I/O 窗口中，您可以根据需要配置数据输入和输出。例如，要执行其他自定义任务，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- *URL ：REST 服务的端点 URL。这个属性是必须的。*
- *方法 ：名为 的端点的方法，如 GET 和 POST。默认值为 GET。*
- *ContentType ：发送数据时的数据类型。此属性对于 POST 和 PUT 请求是必需的。*
- *ContentTypeCharset ：为 ContentType 设置字符。*
- *内容 ：您要发送的数据。此属性支持向后兼容性，改为使用 ContentData 属性。*
- *ContentData ：要发送的数据。此属性对于 POST 和 PUT 请求是必需的。*
- *ConnectTimeout ：连接超时（以秒为单位）。默认值为 60000 毫秒。您必须以毫秒为单位提供输入值。*
- *ReadTimeout ：响应时超时（以秒为单位）。默认值为 60000 毫秒。您必须以毫秒为单位提供输入值。*
- *用户名 ：用于身份验证的用户名.*
- *密码 ：用于身份验证的密码。*
- *AuthUrl:正在处理身份验证的 URL。*
- *AuthType ：处理身份验证的 URL 类型。*
-

HandleResponseErrors （可选）：Instructs 处理程序在响应代码失败时抛出错误（除 2XX 除外）。

- *ResultClass: valid name of the response is unmarshalled 的类。如果没有提供，则以字符串格式返回原始响应。*
- *AcceptHeader ：接受标题的值。*
- *AcceptCharset ：接受标头集。*
- *标头 ：要传递 REST 调用的标头，如 content-type=text/html。*

您可以在 Data Outputs 和 Assignments 中添加以下数据输出来存储任务执行的输出：

- *结果 ：输出其他自定义任务的变量（对象类型）。*

电子邮件

电子邮件自定义任务用于从进程发送电子邮件。它包含与其关联的电子邮件正文。

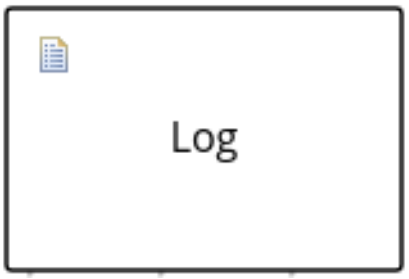
激活电子邮件自定义任务时，电子邮件数据将分配给任务的数据输入属性。电子邮件自定义任务在发送相关电子邮件时完成。

您可以在 Properties 面板中点 Assignments 以打开 Email Data I/O 窗口。在 Email Data I/O 窗口中，您可以根据需要配置数据输入。例如，要执行电子邮件自定义任务，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- *正文 ：电子邮件的正文。*
- *从 ：发件人的电子邮件地址。*
- *主题 ：电子邮件的主题.*
- *到 ：接收者的电子邮件地址。您可以指定多个以分号(;)分隔的电子邮件地址。*
- *模板 （可选）：用于生成电子邮件正文的模板。如果输入，Template 属性会覆盖 Body 参数。*
- *reply-To ：将回复消息发送到的电子邮件地址。*
- *cc: 复制收件人的电子邮件地址。您可以指定多个以分号(;)分隔的电子邮件地址。*
- *BCC: blind copy 的接收者电子邮件地址。您可以指定多个以分号(;)分隔的电子邮件地址。*
- *Attachments ：发送电子邮件到发送的电子邮件。*
- *debug: Flag 来启用 debug 日志记录。*

Log

日志自定义任务用于记录来自进程的消息。当业务流程到达日志自定义任务时，消息数据会被分配给 data 输入属性。



记录相关的消息时会完成日志自定义任务。您可以在 Properties 面板中点 Assignments 以打开 Log Data I/O 窗口。在 Log Data I/O 窗口中，您可以根据需要配置数据输入。例如，要执行日志自定义任务，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- *Message: 来自进程的日志消息。*

WebService

Web 服务自定义任务用于从进程调用 Web 服务。此自定义任务充当 Web 服务客户端，其 Web 服务响应以字符串形式存储。

若要从进程调用 Web 服务，您必须使用正确的任务类型。您可以在 Properties 面板中单击 Assignments 以打开 WS Data I/O 窗口。在 WS Data I/O 窗口中，您可以根据需要配置数据输入和输出。例如，要执行 web 服务任务，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- *端点 ：要调用的 Web 服务的端点位置。*
- *接口 ：服务的名称，如 Weather。*
- *模式 ：服务的模式，如 SYNC、ASYNC 或 ONEWAY。*
- *命名空间 ：Web 服务的命名空间，如 http://ws.cdyne.com/WeatherWS/。*
- *操作 ：要调用的方法名称。*
- *参数 ：要为操作发送的对象或数组。*

- URL ：Web 服务的 URL，如 http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL。

您可以在 Data Outputs 和 Assignments 中添加以下数据输出来存储任务执行的输出：

- 结果 ：web 服务任务的输出变量（对象类型）。

milestone

milestone 代表进程实例内实现的单一点。您可以使用 milestones 标记某些事件，以触发其他任务或跟踪进程的进度。

milestones 可用于关键性能指标(KPI)跟踪或识别仍然完成的任务。milestones 可以在进程中的一个阶段结束时发生，或者是实现其他里程干的结果。

milestones 可以在进程执行过程中达到以下状态：

- Active ：已经为 milestone 节点定义了 milestone 条件，但尚未满足。
- 完成 ：一个 milestone 条件已满足（如果适用）， milestone 已被实现，进程可以继续进行下一任务或可以结束。

您可以在 Properties 面板中点击 Assignments 以打开 Milestone Data I/O 窗口。在 Milestone Data I/O 窗口中，您可以根据需要配置数据输入。例如，要执行一个 milestone，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- 条件: 满足 milestone 的条件。例如，您可以输入使用进程变量的 Java 表达式（字符串数据类型）。

DecisionTask

决策任务用于执行 DMN 图表并从进程调用决策引擎服务。默认情况下，决策任务映射到 DMN 决策。

您可以使用决策任务在流程中做出一个操作决策。决策任务可用于识别需要做出过程中的关键决策。

您可以在 Properties 面板中点 Assignments 以打开 Decision Task Data I/O 窗口。在 Decision Task Data I/O 窗口中，您可以根据需要配置数据输入。例如，要执行决策任务，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- *决策 ：做出流程的决策。*
- *语言 ：决策任务的语言，默认为 DMN。*
- *模型 ：DMN 模型的名称。*
- *命名空间 ：DMN 模型的命名空间。*

BusinessRuleTask

业务规则任务用于评估 DRL 规则并从进程调用决策引擎服务。默认情况下，商业规则任务映射到 DRL 规则。

您可以使用商业规则任务来评估业务流程中的关键业务规则。您可以在 Properties 面板中点 Assignments 以打开 Business Rule Task Data I/O 窗口。在 Business Rule Task I/O 窗口中，您可以根据需要配置数据输入。例如，要执行商业规则任务，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- *KieSessionName: KIE 会话的名称。*
- *KieSessionType: KIE 会话的类型。*
- *语言 ：商业规则任务的语言，默认为 DRL。*

KafkaPublishMessages

Kafka 工作项用于将事件发送到 Kafka 主题。此自定义任务包含一个 work item 处理程序，它使用 Kafka producer 将消息发送到特定的 Kafka 服务器主题。例如，KafkaPublishMessages 任务从进程发布到 Kafka 主题。

您可以在 Properties 面板中点 Assignments 以打开 KafkaPublishMessages Data I/O 窗口。在 KafkaPublishMessages Data I/O 窗口中，您可以根据需要配置数据输入和输出。例如，要执行 Kafka 工作项，在 Data Inputs 和 Assignments 字段中输入以下数据输入：

- *Key ：要发送的 Kafka 消息的密钥。*
- *主题 ： Kafka 主题的名称。*
- *值 ：要发送 Kafka 消息的值。*

您可以在 Data Outputs 和 Assignments 中添加以下数据输出 来存储工作项目执行的输出：

- 结果 ：输出变量（字符串类型）

有关业务流程中的 KafkaPublishMessages 的更多信息，请参阅将 Red Hat Process Automation Manager 与 Red Hat AMQ Streams 集成。

4.5. 在进程设计器中的 BPMN2 子进程

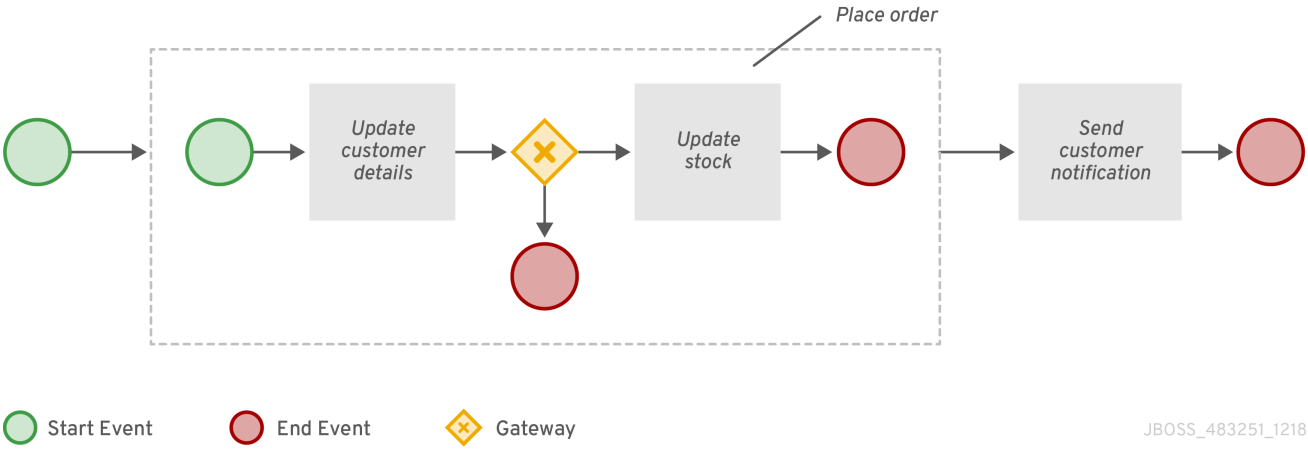
子进程是一个活动，包含节点。您可以在子进程中嵌入主进程的一部分。您还可以在子进程中包含变量定义。这些变量可以被子进程中的所有节点访问。

子进程必须至少有一个传入连接和一个传出连接。子进程中的终止结束事件将结束子进程实例，但不会自动结束父进程实例。当其中没有更多活跃元素时，子进程将结束。

Red Hat Process Automation Manager 中支持以下子流程类型：

- 嵌入式子进程：作为父进程执行一部分的子进程，并共享父进程数据，以及声明其自己的本地子进程变量。
- 临时子进程：一个没有严格的元素执行顺序的子进程。
- 可重复利用的子进程：独立于父进程的子进程。
- 事件子进程：仅在启动事件或计时器上触发的子进程。
- 多实例子进程：被多次实例化的子进程。

在以下示例中，Place order 子流程会检查是否有足够的库存来放置订购并更新库存信息（如果可以放置顺序）。然后，根据是否放置订购，通过主过程通知客户。



嵌入式子进程

嵌入式子进程封装了进程的一部分。它必须包含起始事件和至少一个结束事件。请注意，这个元素允许您定义可由此容器内的所有元素访问的本地子进程变量。

adhoc 子进程

一个临时的子进程或进程包含许多嵌入的内部活动，旨在与典型进程流相比，具有更灵活的排序来执行。与常规进程不同，临时子进程不包含完整的、结构化的 BPMN2 图表描述，例如从启动事件到结束事件。相反，临时子进程仅包含活动、序列流、网关和中间事件。一个临时子进程也可以包含数据对象和数据关联。临时子进程中的活动不需要进入和传出序列流。但是，您可以指定一些包含的活动之间的序列流。使用时，序列流提供与常规进程相同的排序限制。要具有任何含义，中间事件必须具有传出的序列流，并可多次触发，同时临时子进程处于活跃状态。

可重复使用的子进程

可重复使用的子进程会在父进程中出现折叠状态。要配置可重复使用的子进程，请选择可重复使用的子进程，点

，并展开 Implementation/Execution。设置以下属性：

- *名为 Element ：活动调用和实例化的子进程的 ID。*
- *独立 ：如果选中，子进程将作为独立进程启动。如果没有选择，则当父进程被终止时，活跃的子进程将取消。*
- *中止 父项 ：如果未选择，当执行调用进程实例期间出现错误时，非依赖的子进程可以中止父进程。例如，尝试调用子进程或子进程实例中止时出现错误。只有在未选择 独立 属性时，此属性*

才会看到。适用以下规则：

- 如果可重复使用的子进程独立，则无法使用 Abort 父进程。
- 如果可重复使用的子进程不独立，则可使用 Abort 父进程。

- 等待完成 ：如果选中，则不会在名为 子进程实例终止前执行指定的 On Exit Action。当 On Exit Action 完成后，父进程执行将继续执行。默认选择此属性（设置为 true）。

- is Async ：如果应异步调用任务且无法立即执行，则选择该选项。

- 多个实例 ：选择执行指定次数的子进程元素。如果选择， 可用选项如下：
  - MI 执行模式 ：指示多个实例是否并行或按顺序执行。如果设置为 Sequential，则在上一个实例完成前不会创建新的实例。
  - MI Collection 输入 ：选择一个代表创建新实例的元素集合的变量。子进程作为集合大小进行多次实例化。
  - MI Data Input ：指定包含集合中所选元素的变量名称。变量用于访问集合中的元素。
  - MI Collection 输出: Optional 变量， 代表收集多实例节点的输出的元素集合。
  - MI Data Output ：指定添加到 MI Collection 输出 属性中选择的输出集合中的变量名称。
  - MI Completion Condition(mvel): 每个完成的实例评估的 MVEL 表达式，以检查指定的多个实例节点是否可以完成。如果它评估为 true，则所有剩余的实例都将被取消。

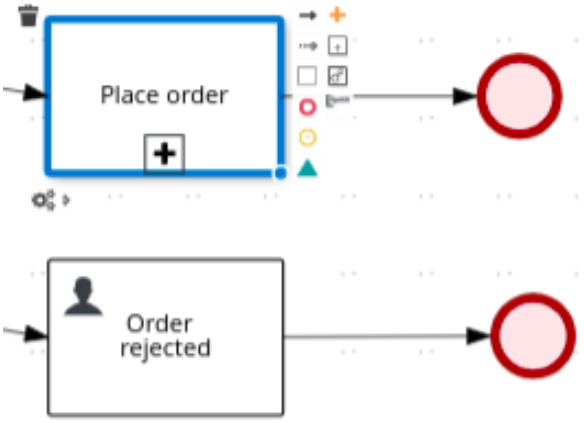
- Entry Action ：Java 或 MVEL 脚本，用于指定任务开头的操作。

- *在 Exit Action 中：Java 或 MVEL 脚本，用于指定任务末尾的操作。*

- *SLA 到期日期：服务级别协议(SLA)过期的日期。您可以在天、分钟、秒和毫秒为单位。例如，SLA 因为日期字段中的 1m 值代表一分钟。*

- *元数据属性 ：添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现元数据属性时实施某些操作的监听程序。*

图 4.1. 可重复使用的子进程属性



Implementation/Execution

Called Element

itorders-data.place-order

Independent

Abort Parent

Wait for Completion

Is Async

Multiple Instance

On Entry Action

Java

On Exit Action

Java

SLA Due Date

Data Assignments

Assignments

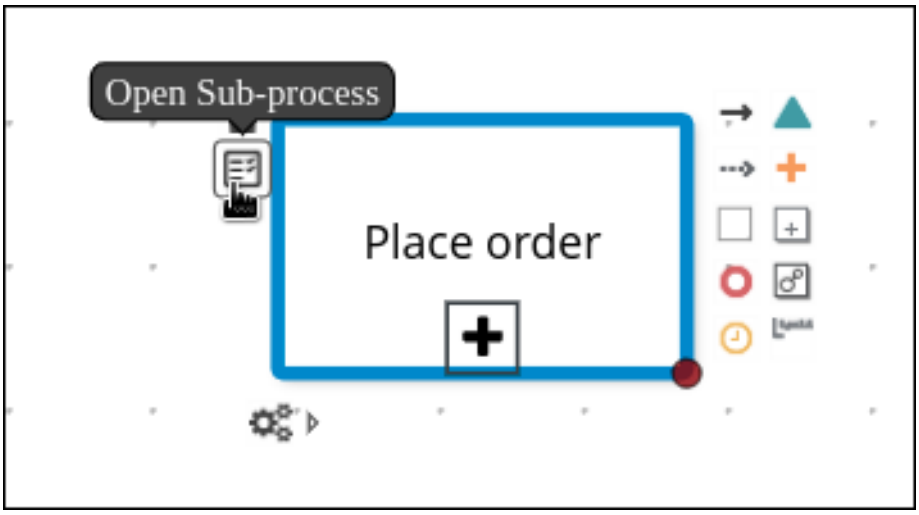
0 data inputs, 0 data outputs

AdvancedData

Metadata Attributes

Name | Value

您可以通过单击主进程中的 Place order 任务，然后单击 Open Sub-process 任务图标，在 Business Central 的新编辑器中打开子进程。



事件子进程

当事件启动事件被触发时，事件子进程会变为活跃状态。它可以中断父进程上下文，或者与其并行运行。

如果没有传出或传入连接，则只有事件或计时器可触发子进程。子进程不是常规控制流的一部分。尽管自包含，但在绑定进程的上下文中执行。

在进程流中使用事件子进程来处理主进程之外发生的事件。例如，在预定了飞行时，可能会发生两个事件：

- 取消图书（中断）
- 检查图书状态（非中断）

您可以使用事件子处理来模拟这两个事件。

多个实例子进程

当多个实例子进程触发时，会多次实例化它。实例按顺序创建或并行创建。如果您设置了连续模式，只有在上一实例完成后才会创建一个新的子进程实例。但是，当您设置并行模式时，会一次性创建所有子进程实例。

多个实例子进程具有一个传入连接和一个传出连接。

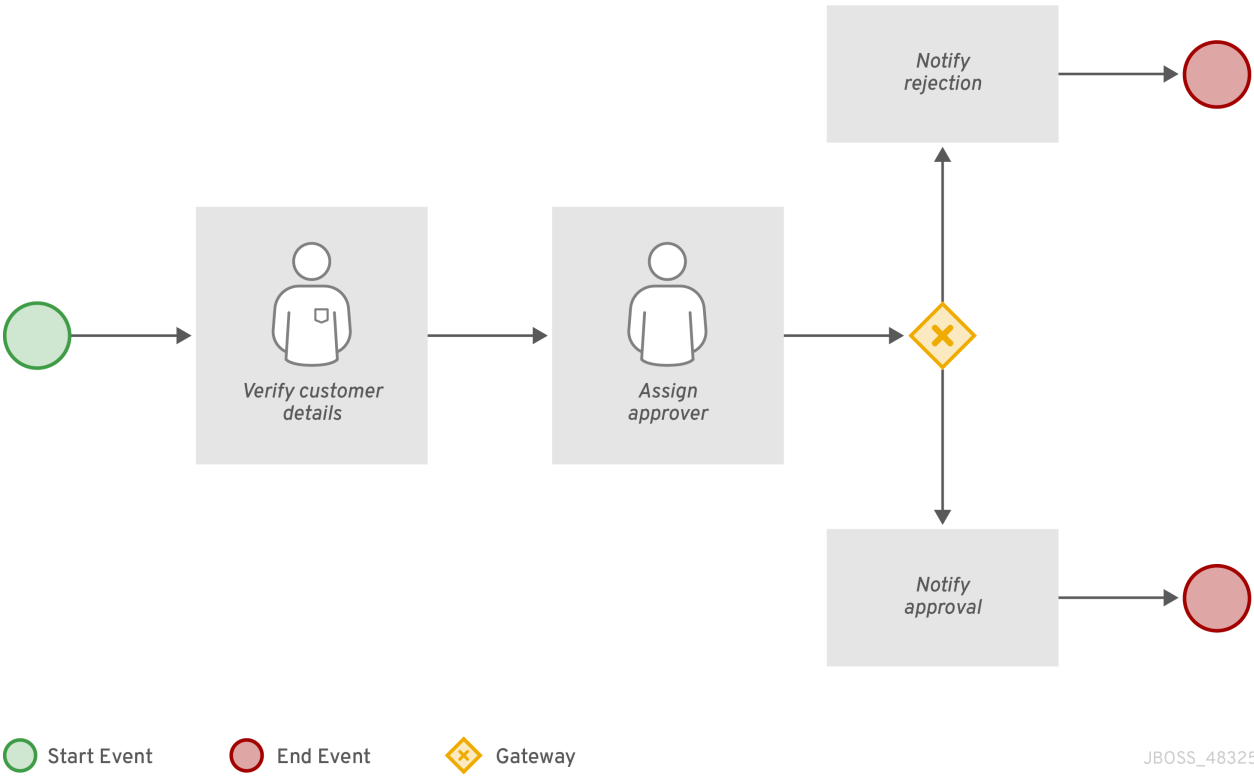
4.6. PROCESS DESIGNER 中的 BPMN2 网关

网关用于通过一组称为聚合机制在工作流中创建或同步分支。BPMN2 支持两种类型的网关：

- 聚合网关，将多个流合并到一个流中
- 分离网关，将一个流分成多个流程

一个网关无法有多个传入和多个传出的流。

在以下处理图中，XOR 网关仅评估其条件评估为 true 的传入流：



在本例中，用户会验证客户详情，并将进程分配给用户进行批准。若获得批准，则会向用户发送批准通知。如果请求的事件被拒绝，则会向用户发送拒绝通知。

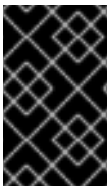
表 4.12. 网关元素

元素类型	图标
专用(XOR)	
含	
并行	
事件	

专用

在专用化网关中，仅选择评估为 true 的第一个传入的流。在聚合网关中，会为每个触发的传入流触发下一节点。

网关只触发一个传出流。约束的流被评估为 true，并且 会采用最低优先级 号。



重要

确保在运行时至少评估为 true 的传出流。否则，进程实例会以运行时异常终止。

聚合网关可让工作流分支在到达网关后马上继续执行传出流。当其中一个传入的流触发网关时，工作流将继续到网关的传出流。如果从多个传入的流触发，它会为每个触发器触发下一个节点。

含

使用一个有包的分离网关，会执行传入的流，并且执行所有评估为 true 的传出流。触发优先级优先级较高的连接前，会触发较低优先级号的连接。优先级会被评估，但 BPMN2 规格不能保证优先级顺序。根据工作流中的 priority 属性，避免其避免。

重要

确保在运行时至少评估为 true 的传出流。否则，进程实例会以运行时异常终止。

合并了包容网关，合并了以前由一站式网关创建的所有传入的流程。它充当包含网关分支的同步入口点。

并行

使用并行网关同步并创建并行流。使用并行分离网关时，会同时处理传入的流。使用聚合并行网关时，网关将等待所有传入的流输入，然后仅触发传出流。

事件

基于事件的网关只分离出来，可让您对可能的事件做出反应，而不是基于数据专用网关，该网关会对进程数据做出反应。传出流将根据发生的事件进行。每次仅占用一个传出流。网关可能充当启动事件，只有发生与基于事件的网关连接的其中一个中间事件时，进程才会实例化。

4.7. BPMN2 在进程设计器中连接对象

连接对象会在两个 BPMN2 元素之间创建一个关联。当连接对象被定向时，关联是连续的，这表示其中一个元素在进程的实例内立即执行。连接对象可以在所关联的进程元素的顶部、右或左边启动和结束。OMG BPMN2 规范允许您使用您的自由，将连接对象放在使流程行为易于理解和遵循的方式中。

BPMN2 支持两种类型的连接对象：

- 序列流：连接进程的元素并定义实例内执行这些元素的顺序。
- 关联流：在没有执行语义的情况下连接进程的元素。关联流可以是不直接的，也可以是单向。

注意

新进程设计器只支持未直接关联流程。旧设计器支持了一个方向和一致性流程。

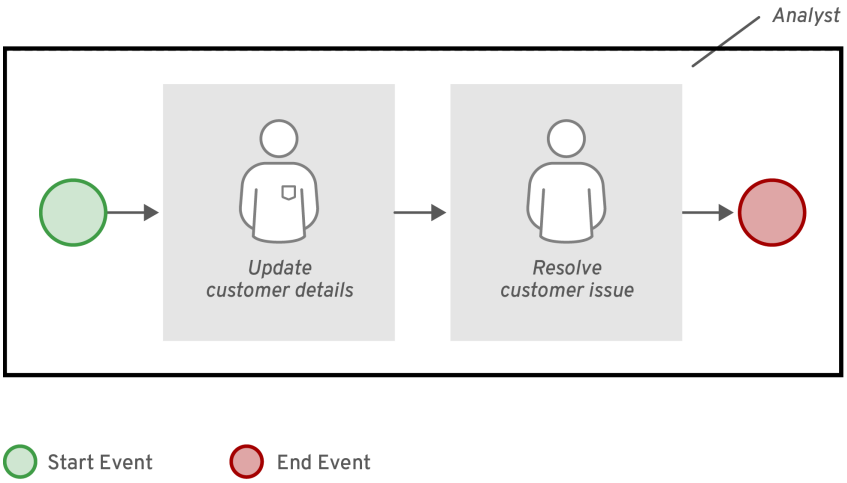
4.8. PROCESS DESIGNER 中的 BPMN2 SWIMLANES

Swimlanes 是以视觉方式对与某个组或用户相关的任务进行分组的进程元素。由于 swimlanes 的 Autoclaim 属性，您可以结合使用用户任务与 swimlanes 来为同一扇区分配多个用户任务。当一个组群的潜在所有者声明 swimlane 中的第一项任务时，其他任务将直接分配给同一所有者。因此，组的其余部分不需要其他任务的声明。Autoclaim 属性可启用与 swimlane 相关的任务自动分配。

注意

如果 swimlane 中剩余的用户任务包含多个预定义的 ActorIds，则不会自动分配用户任务。

在以下示例中，其中的 lane 由两个用户任务组成：



Update Customer Details 和 Resolve Customer Issue 任务中的 Group 项包含价值。启动这个过程后，更新客户详细信息 任务由分析者声明、启动或完成，并声明了 Resolve Customer Issue 任务，并分配给完成第一项任务的用户。但是，如果只有 Update Customer Details 任务包含 analyst 组，且第二项任务不包含用户或组分配，则进程会在第一项任务完成后停止。

您可以禁用 swimlanes 的 Autoclaim 属性。如果禁用了 Autoclaim 属性，则会自动分配与 swimlane 相关的任务。默认情况下，Autoclaim 属性的值设置为 true。如果需要，您还可以从 Business Central 中的项目设置更改 Autoclaim 属性的默认值，或使用部署描述符文件。

要更改 Business Central 中 swimlanes 的 Autoclaim 属性的默认值：

1. *前往项目设置。*
2. *打开 Deployment → Environment 条目。*
3. *在给定字段中输入以下值：*
   - *Name - Autoclaim*
   - *值 - "false"*

如果要在 XML 部署描述符中设置环境条目，请在 kie-deployment-descriptor.xml 文件中添加以下代码：

```
<environment-entries>
  ..
    <environment-entry>
       <resolver>mvel</resolver>
       <identifier>new String ("false")</identifier>
       <parameters/>
       <name>Autoclaim</name>
    </environment-entry>
  ..
</environment-entries>
```

4.9. PROCESS DESIGNER 中的 BPMN2 工件

工件用于提供有关进程的附加信息。工件是指作为进程工作流的一部分在 BPMN2 图表中描述的任何对象。工件没有进入或传出的流对象。工件的目的是提供理解图所需的额外信息。artifacts 表列出了传统进程设计器支持的工件。

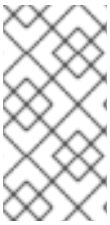
表 4.13. 工件

工件类型	描述
组	组织在整个过程中具有重要意义的任务或进程。新流程设计器不支持组工件。

工件类型	描述
文本注解	为 BPMN2 图提供额外的文本信息。
数据对象	在 BPMN2 图表中显示通过进程的数据流。

4.9.1. 创建数据对象

例如，数据对象以物理和虚拟形式表示在进程中使用的文档。数据对象作为一个页面显示，右上右上角显示。以下流程是创建数据对象的一般概述。



注意

在 Red Hat Process Automation Manager 7.13.5 中，提供了对数据对象的有限支持，它排除了对数据输入、数据输出和关联的支持。

流程

1. 创建业务流程。
2. 在流程设计器中，从工具面板选择 **Artifacts** → **Data Object**。
3. 将数据对象拖放到进程设计器可以撤离，或者点击 **Canvas** 的空白区域。
4. 如有必要，在屏幕右上角点击 **Properties** 图标。
5. 根据需要添加或者定义下表中列出的数据对象信息。

表 4.14. 数据对象参数

标签	描述
Name	数据对象的名称。您还可以双击正在创建的数据对象来编辑名称。

标签	描述
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。 元数据属性可启用对 BPMN 图表的新 **metaData** 扩展，并修改整个数据对象的行为。
类型	选择数据对象类型。  **重要** 当您为数据对象定义数据类型时，您可以在 **Properties** 中的其他数据类型字段中使用相同的数据类型字段，如 **Imports** 和 **Data Assignments**。

6. ***点击 Save。***

第 5 章 在 BUSINESS CENTRAL 中创建业务流程

流程设计器是 Red Hat Process Automation Manager 进程模型程序。modeler 的输出是一个 BPMN 2.0 进程定义文件。该定义用作 Red Hat Process Automation Manager 流程引擎的输入，它根据定义创建进程实例。

本节中的步骤提供了如何创建简单业务流程的一般概述。有关更详细的业务流程示例，请参阅开始使用流程服务。

先决条件

- 您已创建了或导入 Red Hat Process Automation Manager 项目。有关创建项目的更多信息，请参阅 Business Central 中的管理项目。
- 您已创建了所需的用户。用户特权和设置由分配给用户的角色以及用户所属的组控制。有关创建用户的更多信息，请参阅在 Red Hat JBoss EAP 7.4 上安装和配置 Red Hat Process Automation Manager。

流程

1. 在 Business Central 中，转至 Menu → Design → Projects。
2. 点击项目名称以打开项目的资产列表。
3. 点 Add Asset → Business Process。
4. 在 Create new Business Process 向导中输入以下值：
   - 业务流程 ：新业务流程名称
   - 软件包 ：新业务流程的软件包位置，如 com.myspace.myProject
5. 单击 Ok 以打开进程设计程序。

6.

在右上角，点 Properties 
图标并添加您的业务进程属性信息，如处理数据和变量：

a.

向下滚动并展开 Process Data。

b.

点击 Process Variables 旁边的

，并定义您要在处理过程中使用的进程变量。

表 5.1. 常规进程属性

标签	描述
Name	输入进程的名称。
Documentation	描述进程。此字段中的文本包括在流程文档中（如果适用）。
ID	输入此进程的标识符，如 **orderItems**。
软件包	在 Red Hat Process Automation Manager 项目中输入此过程的软件包位置，如 **org.acme**。
ProcessType	指定进程是公共还是私有（如果不适用则为 null）。
版本	输入进程的构件版本。
临时	如果此进程是一个临时子进程，请选择这个选项。
进程实例描述	输入进程用途的描述。
导入	点击以打开 Imports 窗口，再添加为您的进程所需的任何数据对象类。
可执行	选择这个选项使 Red Hat Process Automation Manager 项目的进程可执行。
SLA 过期日期	输入服务级别协议(SLA)过期日期。

标签	描述
进程变量	为进程添加任何进程变量。进程变量在特定的进程实例中可见。在完成进程创建和销毁时初始化进程变量。变量 **标签** 提供对变量行为的更大控制，例如，变量是 **需要** 还是只读。**有关变量标签的更多信息，请参阅 第 6 章 *变量*。**
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。
全局变量	为进程添加任何全局变量。全局变量对项目中的所有进程实例和资产可见。全局变量通常由业务规则和约束使用，并由规则或约束动态创建。

元数据属性条目与 进程变量 标签类似，它们启用了对 BPMN 图表的新 metaData 扩展。但是，process 变量标签会修改特定进程变量的行为，如是否需要某些变量或 只读，而元数据属性是修改整个进程行为的键值定义。

例如，在 BPMN 过程中，以下自定义元数据属性 riskLevel 和 value 对应于用于启动进程的自定义事件监听程序：

*图 **5.1. BPMN 模型器中的 metadata 属性和值示例***

BPMN 文件中的 metadata 属性和值示例

```
<bpmn2:process id="approvals" name="approvals" isExecutable="true"
processType="Public">
  <bpmn2:extensionElements>
    <tns:metaData name="riskLevel">
      <tns:metaValue><![CDATA[low]]></tns:metaValue>
    </tns:metaData>
  </bpmn2:extensionElements>
```

带有元数据值的事件监听程序示例

```
public class MyListener implements ProcessEventListener {
    ...
    @Override
    public void beforeProcessStarted(ProcessStartedEvent event) {
        Map < String, Object > metadata =
event.getProcessInstance().getProcess().getMetaData();
        if (metadata.containsKey("low")) {
            // Implement some action for that metadata attribute
        }
    }
}
```

7. *在流程设计器 canvas 中，使用左侧工具栏来拖放 BPMN 组件，以定义您的业务流程逻辑、连接、事件、任务或其他元素。*

注意

Red Hat Process Automation Manager 中的任务和事件预期一个传入和一个传出流。如果您想设计使用多个传入和多个传出流程的业务流程，请考虑使用网关重新设计业务流程。使用网关可以明显说明逻辑，序列流正在执行。因此，网关被视为多个连接的最佳做法。

但是，如果某个任务或事件必须使用多个连接，则必须将 JVM（Java 虚拟机）系统属性设为 jbpm.enable.multi.con。当 Business Central 和 KIE 服务器在不同服务器上运行时，请确保两者均包含 jbpm.enable.multi.con 系统属性（否则启用），该进程引擎抛出异常。

8. *添加并定义业务流程的所有组件后，点 Save 保存完成的业务流程。*

5.1. 创建商业规则任务

业务规则任务用于通过决策模型和通知(DMN)模型或规则流组做出决策。

流程

1. 创建业务流程。
2. 在进程设计器中，从工具面板选择 **Activities** 工具。
3. 选择 **"业务规则 "**。
4. 点进程设计器的空白区域。
5. 如有必要，在屏幕右上角点击 **Properties** 图标。
6. 根据需要添加或者定义下表中列出的任务信息。

表 5.2. 业务规则任务参数

标签	描述
Name	业务规则任务的名称。您还可以双击的业务规则任务形成以编辑该名称。
规则语言	任务的输出语言。选择 Decision Model and Notation(DMN)或 Drools(DRL)。
规则流组	与这个业务任务关联的规则流组。从列表中选择规则流组，或者指定新规则流组。
On Entry Action	Java、JavaScript 或 MVEL 脚本，用于指定任务开头的操作。
在退出操作中	Java、JavaScript 或 MVEL 脚本，用于指定任务末尾的操作。
is Async	选择 if should异步调用此任务。如果无法即时执行任务，则进行异步任务，例如由外部服务执行的任务。
adhoc Autostart	如果这是应自动启动的临时任务，请选择此项。**adhoc Autostart** 使任务能够在创建进程或案例实例时自动启动，而不是由启动任务启动。它通常用于管理。
SLA 过期日期	服务级别协议(SLA)过期的日期。
分配	点击 添加本地变量。

标签	描述
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。 Metadata Attributes 可启用对 BPMN 图表的新 **metaData** 扩展，并修改整个任务的行为。

7. *点击 **Save**。*

5.2. 创建脚本任务

*脚本任务用于执行使用 **Java**、**JavaScript** 或 **MVEL** 编写的一系列代码。它们包含指定 **script** 任务操作的代码片段。您可以在脚本中包含全局和进程变量。*

*请注意，**MVEL** 接受任何有效的 **Java** 代码，同时还提供对参数嵌套访问的支持。例如，相当于 **Java call person.getName（）** 的 **MVEL** 是 **person.name**。**MVEL** 还提供其他有关 **Java** 和 **MVEL** 表达式的改进，对于业务用户通常更方便。*

流程

1. *创建业务流程。*
2. *在进程设计器中，从工具面板选择 **Activities** 工具。*
3. *选择 **脚本**。*
4. *点进程设计器的空白区域。*
5. *如有必要，在屏幕右上角点击 **Properties** 图标。*
6. *根据需要添加或者定义下表中列出的任务信息。*

表 5.3. 脚本任务参数

标签	描述
Name	脚本任务的名称。您还可以双击 script 任务形图来编辑名称。
Documentation	输入任务的描述。此字段中的文本包含在流程文档中。单击流程设计器左上角的 **Documentation** 选项卡，以查看流程文档。
脚本	在 Java、JavaScript 或 MVEL 中输入由任务执行的脚本，然后选择脚本类型。
is Async	选择 if should异步调用此任务。如果无法即时执行任务，则进行异步任务，例如由外部服务执行的任务。
adhoc Autostart	如果这是应自动启动的临时任务，请选择此项。**adhoc Autostart** 使任务能够在创建进程或案例实例时自动启动，而不是由启动任务启动。它通常用于管理。
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。 **Metadata Attributes** 可启用对 BPMN 图表的新 **metaData** 扩展，并修改整个任务的行为。

7. *点击 **Save**。*

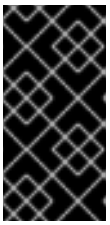
5.3. 创建服务任务

服务任务是根据 Web 服务调用或 Java 类方法执行操作的任务。服务任务示例包括在执行这些任务时发送电子邮件和日志信息。您可以定义与服务任务关联的参数（输入）和结果（输出）。您还可以定义包含所有输入到单个对象中的嵌套参数。要定义嵌套参数，请使用 Wrapped': 'True 在数据分配中创建一个新的工作项目处理程序。Service 任务应该有一个进入的连接和一个传出的连接。

流程

1. *在 **Business Central 中，选择屏幕右上角的 Admin 图标，然后选择 Artifacts。***
2. *单击 **Upload 以打开 Artifact 上传 窗口。***
3. *选择 **.jar 文件并点击***

。



重要

.jar 文件包含数据类型（数据对象）以及 web 服务和 Java 服务任务的 Java 类。

4. 创建一个要使用的项目。

5. 进入您的项目 Settings → Dependencies。

6. 单击 Add from repository，找到上传的 .jar 文件，然后单击 Select。

7. 打开您的项目 Settings → Work Item Handler。

8. 在给定字段中输入以下值：

- Name - Service Task
- Value - new org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession, classLoader)

9. 保存项目。

创建 web 服务任务示例

BPMN2 规格中的服务任务的默认实现是 Web 服务。Web 服务支持基于 Apache CXF 动态客户端，它提供实现 WorkItemHandler 接口的专用服务任务处理程序：

org.jbpm.process.workitem.bpmn2.ServiceTaskHandler

要使用 web 服务创建服务任务，您必须配置 web 服务：

a. 创建业务流程。

b. 如有必要，在屏幕右上角点击 Properties 图标。

c. 点击 Imports 属性中的  打开 Imports 窗口。

d. 单击 WSDL Imports 旁边的 +Add，以导入所需的 WSDL（Web 服务描述语言）。例如：

- 位置 ：http://localhost:8080/sample-ws-1/SimpleService?wsdl

  位置指向服务的 WSDL 文件。

- Namespace: http://bpmn2.workitem.process.jbpm.org/

  命名空间必须与 WSDL 文件中的 targetNamespace 匹配。

e. 在进程设计器中，从工具面板选择 Activities 工具。

f. 选择 Service Task。

g. 点进程设计器的空白区域。

h. 根据需要添加或者定义下表中列出的任务信息。

表 5.4. Web 服务任务参数

标签	描述
Name	service 任务的名称。您还可以双击服务任务形图来编辑名称。
Documentation	输入任务的描述。此字段中的文本包含在流程文档中。单击流程设计器左上角的 Documentation 选项卡，以查看流程文档。
实施	指定一个 Web 服务。
Interface	此服务用于实施脚本，如 **CountriesPortService**。
操作	接口调用的操作，如 **getCountry**。
分配	点击 添加本地变量。
adhoc Autostart	如果这是应自动启动的临时任务，请选择此项。adhoc Autostart 使任务能够在创建进程或案例实例时自动启动，而不是由启动任务启动。它通常用于管理。
is Async	选择 if should异步调用此任务。如果无法即时执行任务，则进行异步任务，例如由外部服务执行的任务。
是多个实例	如果此任务具有多个实例，请选择此项。
MI 执行模式	选择多个实例是否并行或按顺序执行。
MI Collection 输入	指定代表创建新实例的元素集合的变量，如 **inputCountryNames**。
MI 数据输入	指定传输到 web 服务的输入数据分配，如 **Parameter**。
MI Collection 输出	来自 web 服务任务返回的值的数组列表，如 **outputCountries**。
MI 数据输出	指定 web 服务任务的输出数据分配，它将保存在服务器上类执行的结果，如 **Result**。
MI Completion Condition(mvel)	指定每个完成实例上评估的 MVEL 表达式，以检查指定的多个实例节点是否可以完成。
On Entry Action	Java、JavaScript 或 MVEL 脚本，用于指定任务开头的操作。
在退出操作中	Java、JavaScript 或 MVEL 脚本，用于指定任务末尾的操作。
SLA 过期日期	服务级别协议(SLA)过期的日期。

标签	描述
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。 Metadata Attributes 可启用对 BPMN 图表的新 **metaData** 扩展，并修改整个任务的行为。

创建 Java 服务任务示例

使用 Java 方法创建服务任务时，方法只能包含一个参数，并返回单个值。要使用 Java 方法创建服务任务，您必须将 Java 类添加到项目的依赖项中：

a. ***创建业务流程。***

b. ***在进程设计器中，从工具面板选择 Activities 工具。***

c. ***选择 Service Task。***

d. ***点进程设计器的空白区域。***

e. ***如有必要，在屏幕右上角点击 Properties 图标。***

f. ***根据需要添加或者定义下表中列出的任务信息。***

表 5.5. Java 服务任务参数

标签	描述
Name	service 任务的名称。您还可以双击服务任务形图来编辑名称。
Documentation	输入任务的描述。此字段中的文本包含在流程文档中。单击流程设计器左上角的 Documentation 选项卡，以查看流程文档。
实施	指定任务在 Java 中实施。
Interface	用于实施脚本的类，如 **org.xyz.HelloWorld**。
操作	接口调用的方法，如 **sayHello**。

标签	描述
分配	点击 添加本地变量。
adhoc Autostart	如果这是应自动启动的临时任务，请选择此项。**adhoc Autostart** 使任务能够在创建进程或案例实例时自动启动，而不是由启动任务启动。它通常用于管理。
is Async	选择 if should异步调用此任务。如果无法即时执行任务，则进行异步任务，例如由外部服务执行的任务。
是多个实例	如果此任务具有多个实例，请选择此项。
MI 执行模式	选择多个实例是否并行或按顺序执行。
MI Collection 输入	指定代表创建新实例的元素集合的变量，如 **InputCollection**。
MI 数据输入	指定转移到 Java 类的输入数据分配。例如，您可以将输入数据分配设置为 **Parameter** 和 **ParameterType**。**ParameterType** 代表 **Parameter** 的类型，并将参数发送到 Java 方法的执行。
MI Collection 输出	存储在 Java 类返回的值的数组列表中，如 **OutputCollection**。
MI 数据输出	指定 Java 服务任务的输出数据分配，它将保存在服务器上类执行的结果，如 **结果**。
MI Completion Condition(mvel)	指定每个完成实例上评估的 MVEL 表达式，以检查指定的多个实例节点是否可以完成。例如，**OutputCollection.size（）<= 3** 代表没有解决三个以上的人员。
On Entry Action	Java、JavaScript 或 MVEL 脚本，用于指定任务开头的操作。
在退出操作中	Java、JavaScript 或 MVEL 脚本，用于指定任务末尾的操作。
SLA 过期日期	服务级别协议(SLA)过期的日期。
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。 **Metadata Attributes** 可启用对 BPMN 图表的新 **metaData** 扩展，并修改整个任务的行为。

10. ***点击 Save。***

5.4. 创建用户任务

用户任务用于包括人工操作作为业务流程的输入。

流程

1. *创建业务流程。*
2. *在进程设计器中，从工具面板选择 Activities 工具。*
3. *选择用户 。*
4. *将用户任务拖放到进程设计器可以撤离。*
5. *如有必要，在屏幕右上角点击 Properties 图标。*
6. *根据需要添加或者定义下表中列出的任务信息。*

表 5.6. 用户任务参数

标签	描述
Name	用户任务的名称。您还可以双击用户任务组成以编辑名称。
Documentation	输入任务的描述。此字段中的文本包含在流程文档中。单击流程设计器左上角的 Documentation 选项卡，以查看流程文档。
任务名称	人工任务的名称。
subject	为任务输入主题。
actors	负责执行人工任务的操作器。点 Add 添加行，然后从列表中选择 actor，或者点击 New 添加新 actor。
组	负责执行人工任务的组。单击 Add 以添加行，然后从列表中选择一个组，或者单击 New 以添加新组。
分配	此任务的本地变量。点击以打开 Task Data I/O 窗口，然后根据需要添加数据输入和输出。您还可以将 MVEL 表达式添加为数据输入和输出分配。有关 MVEL 语言的更多信息，请参阅 2.0 语言指南。

标签	描述
Reassignments	指定不同的ctor 来完成此任务。
通知	单击 以指定与任务关联的通知。
is Async	选择 if should异步调用此任务。如果无法即时执行任务，则进行异步任务，例如由外部服务执行的任务。
Skippable	如果不需要此任务，请选择此项。
优先级	为任务指定优先级。
描述	输入人工任务的描述。
创建者	创建此任务的用户。
adhoc Autostart	如果这是应自动启动的临时任务，请选择此项。adhoc Autostart 使任务能够在创建进程或案例实例时自动启动，而不是由启动任务启动。它通常用于管理。
多个实例	如果此任务具有多个实例，请选择此项。
On Entry Action	Java、JavaScript 或 MVEL 脚本，用于指定任务开头的操作。
在退出操作中	Java、JavaScript 或 MVEL 脚本，用于指定任务末尾的操作。
内容	脚本的内容。
SLA 过期日期	服务级别协议(SLA)过期的日期。
元数据属性	添加要用于自定义事件监听程序的任何自定义元数据属性名称和值，例如在出现 metadata 属性时实施某些操作的监听程序。 Metadata Attributes 可启用对 BPMN 图表的新**metaData** 扩展，并修改整个任务的行为。

7. *点击 **Save**。*

5.4.1. 设置用户任务分配策略

用户任务分配策略用于为合适的用户自动分配任务。分配策略允许基于关联属性（如潜在所有者、任务优先级和任务数据）更有效的任务分配。org.jbpm.task.assignment.strategy 是 Red Hat Process Automation Manager 中用户任务分配策略的系统属性。您还可以为 Business Central 中的用户任务明确定义分配策略。

先决条件

- *您已在 Business Central 中创建了一个项目。*
- *您必须将 org.jbpm.task.assignment.enabled 系统属性设置为 true。*

流程

1. *创建业务流程。*

   有关在 Business Central 中创建业务流程的详情请参考 第 5 章 在 Business Central 中创建业务流程。

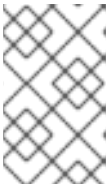
2. *创建用户任务。*

   有关在 Business Central 中创建用户任务的详情，请参考 第 5.4 节 “创建用户任务”。

3. *在屏幕右上角，单击 Properties 图标。*

4. *展开 Implementation/Execution 并点击*
   
   来分配，以打开 Data I/O 窗口。

5. *添加名为 AssignmentStrategy 的数据输入，其类型为 String，使用恒定源，如策略名称。*

   注意

   如果 AssignmentStrategy 设为 null，则不用于任务分配策略。

6. *点 确定。*

   AssignmentStrategy 变量作为数据输入添加到用户任务。

5.5. PROCESS DESIGNER 中的 BPMN2 用户任务生命周期

您可以在进程实例执行过程中触发用户任务元素，以创建用户任务。任务执行引擎的用户任务服务执行用户任务实例。只有在相关用户任务完成后或中止时，进程实例才会继续执行。用户任务生命周期如下：

- 当进程实例进入用户任务元素时，用户任务处于 Created 阶段。
- Create stage 是一个临时阶段，用户任务会立即进入 Ready 阶段。该任务会出现在允许执行任务的所有执行者的任务列表中。
- 当组声明用户任务时，任务将变为 保留。

注意

如果用户任务具有单个潜在的仲裁，则任务在创建后会为这个人分配这个任务。

- 当声明用户任务的监听程序开始执行时，用户任务的状态将变为 InProgress。
- ctor 完成用户任务后，状态会根据执行结果变为 Completed 或 Failed。

另外还有一些其他生命周期方法，包括：

- 委派或转发用户任务，以便用户任务被分配给另一个扇区。
- 撤销用户任务，然后用户任务不再由单个 actor 声明声明，但可供允许采用它的所有执行者使用。
- 暂停和恢复用户任务。
- 停止正在进行的用户任务。

- 跳过一个用户任务，将暂停执行任务。

有关用户任务生命周期的更多信息，请参阅 Web Services human Task 规格。

5.6. PROCESS DESIGNER 中的 BPMN2 任务权限列表

用户权限列表概述了特定用户角色所允许的操作。用户角色如下：

- 潜在所有者：可以声明任务的用户，之前已声明并被释放并转发。可以声明具有 Ready 状态的任务，潜在的所有者成为任务的实际所有者。
- 实际所有者：声明任务并推进任务完成或失败的用户。
- 业务管理员：超级用户可以在任务生命周期的任何时间点修改状态或进度。

以下权限列表代表了修改任务的所有操作的授权。

- + 表示用户角色被允许执行指定的操作。
- - 表示不允许用户角色执行指定操作，或者操作与用户角色不匹配。

表 5.7. 主要操作权限列表

操作	潜在所有者	实际所有者	商业管理员
激活	-	-	+
claim	+	-	+
complete	-	+	+
delegate	+	+	+
FAIL	-	+	+

操作	潜在所有者	实际所有者	商业管理员
forward	+	+	+
nominate	-	-	+
release	-	+	+
remove	-	-	+
resume	+	+	+
skip	+	+	+
start	+	+	+
stop	-	+	+
suspend	+	+	+

5.7. 制作业务流程的副本

您可以在 Business Central 中制作业务流程的副本，并根据需要修改复制的过程。

流程

1. 在业务流程设计器中，单击右上角工具栏中的 Copy。
2. 在 Make a Copy 窗口中，输入复制的业务流程的新名称，选择目标软件包，并选择性地添加一个注释。
3. 点 Make a Copy。
4. 根据需要修改复制的业务流程，点 Save 以保存更新的业务流程。

5.8. 重新定义元素大小并使用缩放功能来查看业务流程

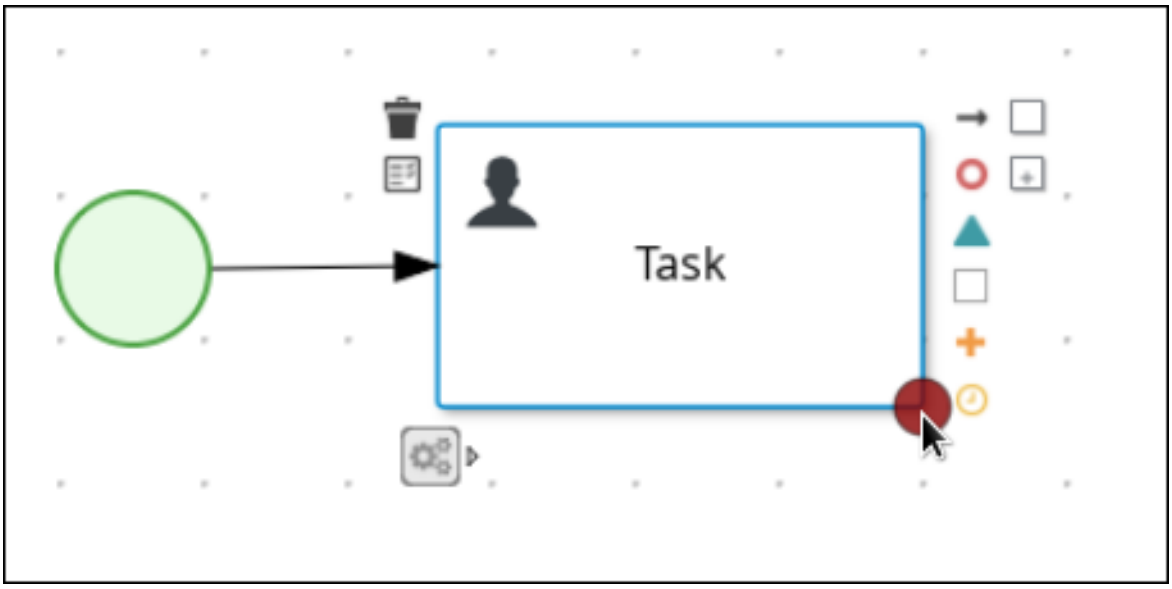
您可以调整业务流程中的单个元素大小，也可以优化或改变您的业务流程视图。

流程

1. 在业务流程设计器中，选择元素并单击元素右下角的红色点。

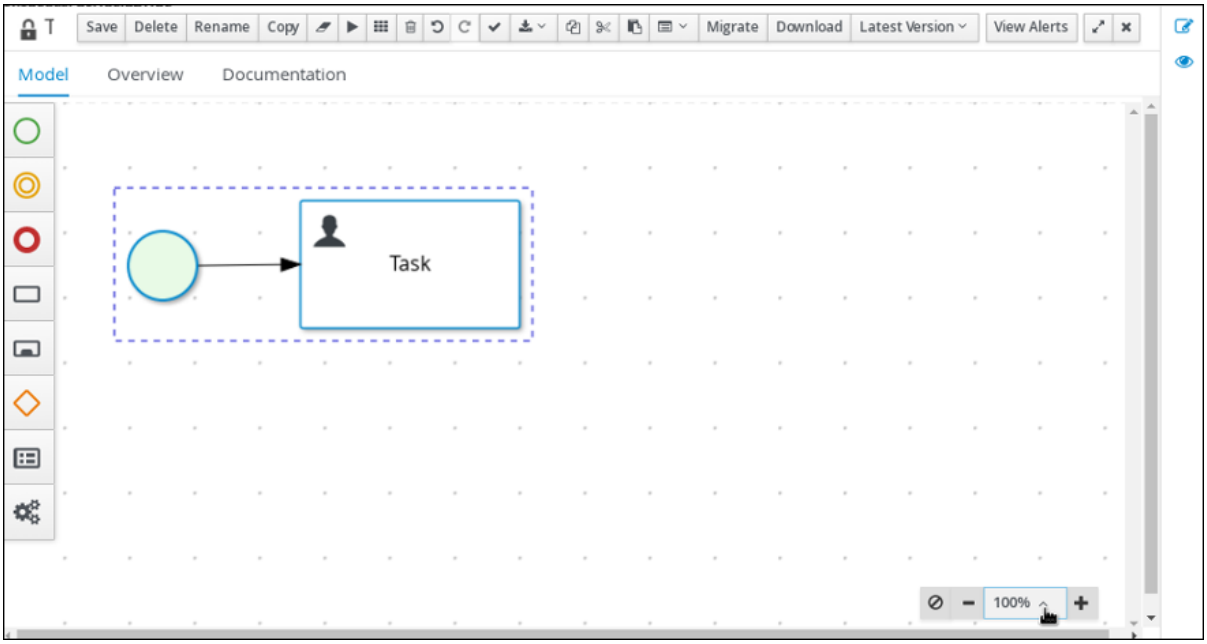
2. 拖动红色圆点来调整元素的大小。

图 5.2. 重新定义元素大小



3. 要放大或横向扩展以查看整个图表，请点击 canvas 右下角的加号或减号。

图 5.3. 放大或缩小业务流程



5.9. 在 BUSINESS CENTRAL 中生成流程文档

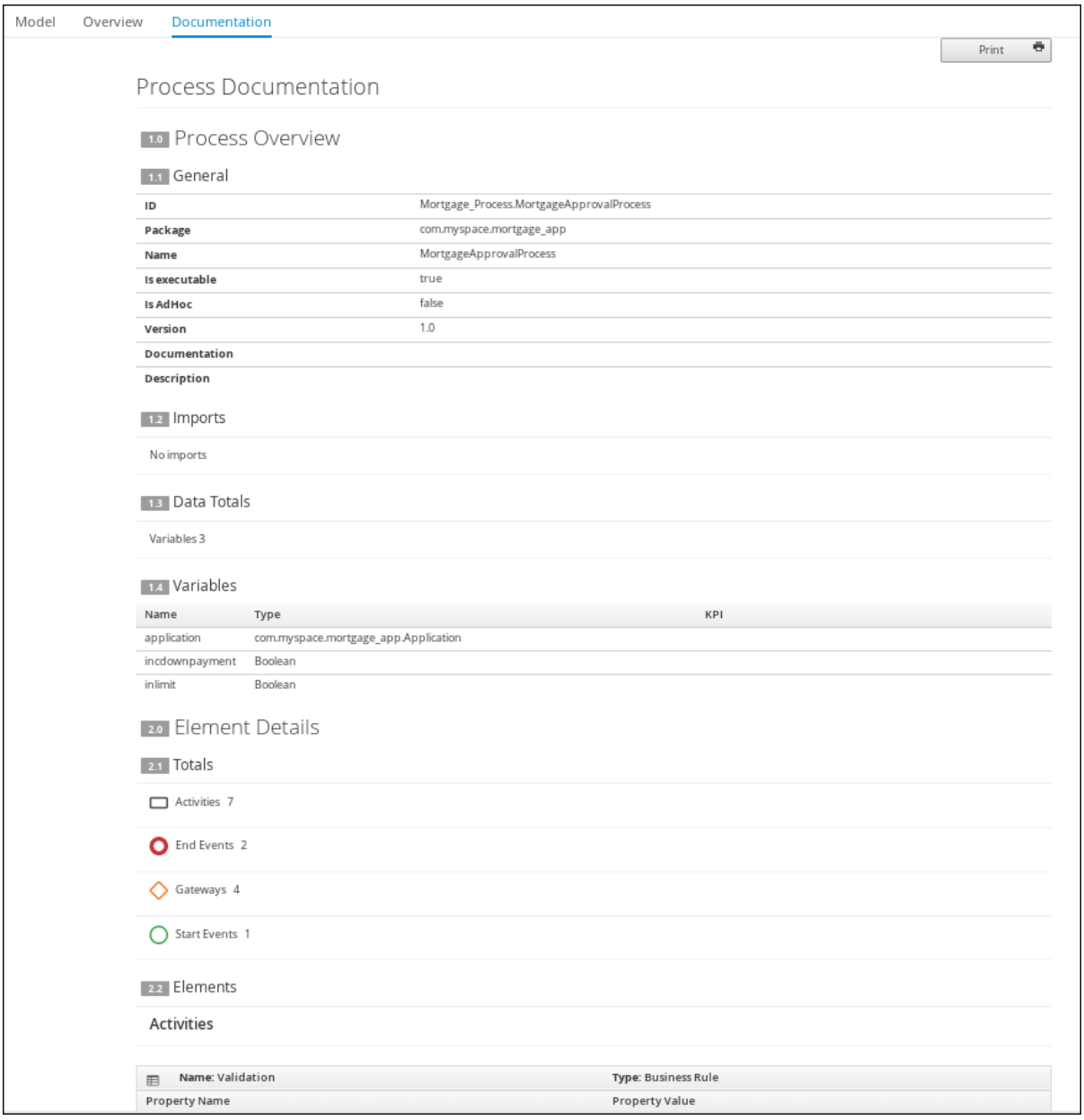
在 Business Central 中的进程设计器中，您可以查看并打印进程定义的报告。进程文档总结了您更轻松地打印和共享的格式(PDF)的组件、数据和可视化流程。

流程

1. *在 Business Central 中，进入包含业务流程的项目并选择流程。*

2. *在进程设计器中，单击 Documentation 选项卡以查看进程文件摘要，然后单击窗口右上角的 Print 以打印 PDF 报告。*

图 5.4. 生成进程文档



第 6 章 变量

变量存储运行时使用的数据。进程设计器使用三种变量类型：

全局变量

全局变量可以在特定会话中的所有进程实例和资产可见。它们主要供业务规则和约束使用，并由规则或约束动态创建。

进程变量

进程变量在 BPMN2 定义文件中作为属性定义，在进程实例中可见。它们在完成进程时初始化和销毁。

本地变量

本地变量与特定进程元素相关联，并在特定进程元素中可用，如活动。在初始化元素上下文时，它们会被初始化，即当执行工作流进入节点并执行 onEntry 操作完成后（如果适用）。当元素上下文被销毁时，它们会被销毁，即当执行工作流离开元素时。

一个元素（如进程、子进程或任务）只能访问其本身和父上下文中的变量。元素无法访问元素中定义的变量。因此，当元素在运行时需要访问变量时，首先搜索其自己的上下文。

如果在元素的上下文中无法直接找到该变量，则会搜索立即父上下文。搜索将继续，直到达到进程上下文。如果是全局变量，则搜索将直接在会话容器上执行。

如果找不到变量，读取访问请求会返回 null，写入访问会生成错误消息，进程将继续执行。根据变量的 ID 搜索。

6.1. 变量标签

为了更好地控制变量行为，您可以在 BPMN 进程文件中标记进程变量和本地变量。标签是简单的字符串值，作为元数据添加到特定变量中。

Red Hat Process Automation Manager 支持以下标签来处理变量和本地变量：

- 必需 ：将 变量设置为要求以启动进程实例。如果进程实例在没有所需变量的情况下启动，则

Red Hat Process Automation Manager 会生成 VariableViolationException 错误。

- ReadOnly ：指示变量仅用于信息目的，且只能在进程实例执行期间设置一次。如果在任何时候修改了只读变量的值，Red Hat Process Automation Manager 会生成 VariableViolationException 错误。

- restricted ：与 VariableGuardProcessEventListener 一起使用的特殊标签，以指示根据所需和现有角色修改变量的权限。

  VariableGuardProcessEventListener 从 DefaultProcessEventListener 扩展，并支持两个不同的构造器：

  - VariableGuardProcessEventListener

    ```
    public VariableGuardProcessEventListener(String requiredRole, IdentityProvider
    identityProvider) {
        this("restricted", requiredRole, identityProvider);
    }
    ```

  - VariableGuardProcessEventListener

    ```
    public VariableGuardProcessEventListener(String tag, String requiredRole,
    IdentityProvider identityProvider) {
        this.tag = tag;
        this.requiredRole = requiredRole;
        this.identityProvider = identityProvider;
    }
    ```

    因此，您必须在带有允许的角色名称和身份提供程序的会话中添加一个事件监听程序，该供应商返回用户角色，如下例所示：

    ```
    ksession.addEventListener(new
    VariableGuardProcessEventListener("AdminRole", myIdentityProvider));
    ```

在上例中，VariableGuardProcessEventListener 方法验证变量是否带有安全约束标签 (restricted)。如果用户没有所需的角色，Red Hat Process Automation Manager 会生成 VariableViolationException 错误。

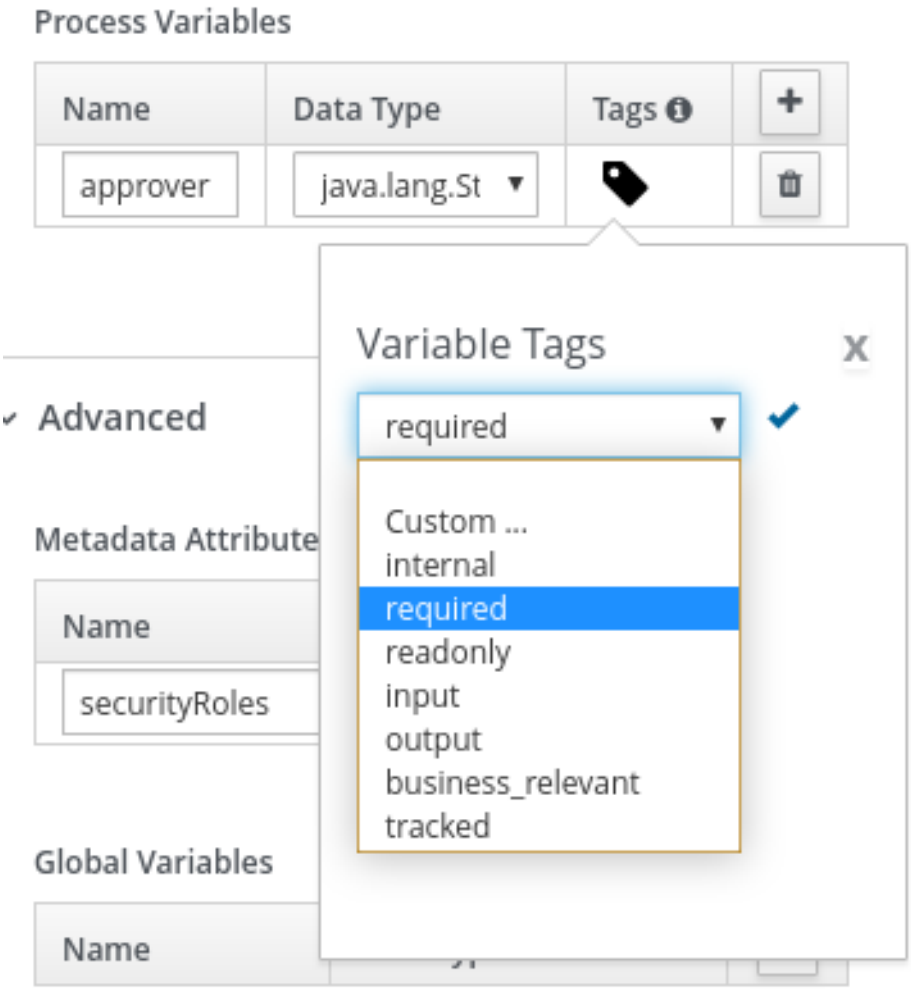
注意

Red Hat Process Automation Manager 不支持 Business Central UI 中显示的变量标签，如 内部、输入、输出、业务相关 和跟踪。

您可以将标签直接添加到 BPMN 进程源文件中，作为 customTags 元数据属性，其格式为 ![CDATA[TAG_NAME]]。

例如，以下 BPMN 进程将 所需的 标签应用到 approver process 变量：

图 6.1. 在 BPMN 型号器中标记的变量示例



在 BPMN 文件中标记的变量示例

```
<bpmn2:property id="approver" itemSubjectRef="ItemDefinition_9" name="approver">
  <bpmn2:extensionElements>
    <tns:metaData name="customTags">
      <tns:metaValue><![CDATA[required]]></tns:metaValue>
```

```
    </tns:metaData>
  </bpmn2:extensionElements>
</bpmn2:property>
```

您可以将多个标签用于适用变量的变量。您还可以在 BPMN 文件中定义自定义变量标签，使变量数据可供 Red Hat Process Automation Manager 处理事件监听程序。自定义标签不会影响 Red Hat Process Automation Manager 运行时，作为标准变量标签，仅用于信息。您以相同的 customTags 元数据属性格式定义自定义变量标签，用于标准 Red Hat Process Automation Manager 变量标签。

6.2. 定义全局变量

全局变量存在于知识会话中，可访问并可由该会话中的所有资产共享。它们属于知识库的特定会话，它们用于将信息传递给引擎。每个全局变量定义其 ID 和项目主题参考。ID 用作变量名称，在进程定义中必须是唯一的。项目主题引用定义了变量存储的数据类型。

重要

在插入事实时，将评估这些规则。因此，如果您使用全局变量来约束事实模式且未设置全局，则系统会返回 NullPointerException。

当带有变量定义的进程添加到会话或者会话初始化时，会初始化全局变量。

全局变量的值通常在分配过程中更改，这是进程变量和活动变量之间的映射。然后，全局 变量与本地活动上下文、本地活动变量或从子上下文直接调用变量相关联。

先决条件

- 您已在 Business Central 中创建了一个项目，它至少包含一个业务流程资产。

流程

1. 开启业务流程资产。
2. 点进程设计器的空白区域。

3. 点击屏幕右上角的 Properties 图标打开 Properties 面板。

4. 如有必要，展开 Process 部分。

5. 在 Global Variables 子部分，点加号图标。

6. 在 名称框中输入变量的名称。

7. 从 Data Type 菜单中选择数据类型。

6.3. 定义进程变量

进程变量在 BPMN2 定义文件中作为属性定义，在进程实例中可见。它们在完成进程时初始化和销毁。

进程变量是在进程上下文中存在的变量，可以由其进程或其子元素访问。进程变量属于特定进程实例，不能被其他进程实例访问。每个进程变量定义其 ID 和项目主题引用：ID 充当变量名称，必须在进程定义中唯一。项目主题引用定义了变量存储的数据类型。

在创建进程实例时初始化进程变量。可以使用 Assignment 更改其值，当全局变量与本地 Activity 上下文关联时，或者直接调用子上下文中的 变量，来更改它们的值。

请注意，进程变量应映射到本地变量。

先决条件

- 您已在 Business Central 中创建了一个项目，它至少包含一个业务流程资产。

流程

1. 开启业务流程资产。

2. 点进程设计器的空白区域。

3. 点击屏幕右上角的 Properties 图标打开 Properties 面板。

4. 如有必要，展开 Process Data 部分。

5. 在 Process Variables 子部分，点加号图标。

6. 在 名称框中输入变量的名称。

7. 从 Data Type 菜单中选择数据类型。

6.4. 定义本地变量

本地变量在其进程元素内提供，如活动。在初始化元素上下文时，它们会被初始化，即当执行工作流进入节点并执行 onEntry 操作完成后（如果适用）。当元素上下文被销毁时，它们会被销毁，即当执行工作流离开元素时。

本地变量的值可以映射到全局或进程变量。这可让您对容纳本地变量的父元素进行相对依赖性。这种隔离或许有助于防止技术例外。

本地变量是存在于进程的子元素上下文中的变量，只能从该上下文中访问。本地变量属于进程的特定元素。

对于任务（除 Script 任务外），您可以在 Assignments 属性中定义 Data Input Assignments 和 Data Output Assignments。Data Input Assignment 定义输入任务的变量，因此提供任务执行所需的条目数据。Data Output Assignments 可以在执行后引用任务的上下文，以获取输出数据。

用户任务显示与正在执行用户任务的ctor 相关的数据。另外，用户任务还要求该人员提供与执行相关的结果数据。

要请求并提供数据，请使用任务表单并将 Data Input Assignment 参数中的数据映射到变量。如果要将数据保留为输出，在 Data Output Assignment 参数中映射用户提供的数据。

先决条件

- *您已在 Business Central 中创建了一个项目，它至少包含一个业务流程资产，其中至少包含一个不是脚本任务的任务。*

流程

1. *开启业务流程资产。*
2. *选择不属于脚本任务的任务。*
3. *点击屏幕右上角的 Properties 图标打开 Properties 面板。*
4. *单击 Assignments 子部分下的框。此时会打开 Task Data I/O 对话框。*
5. *单击 Data Inputs and Assignments or Data Outputs and Assignments 旁边的 Add s。*
6. *在 名称 框中输入本地变量的名称。*
7. *从 Data Type 菜单中选择数据类型。*
8. *选择源或目标，然后点 Save。*

6.5. 编辑进程变量值

启动进程实例后，您可以编辑 Business Central 中的进程变量值。支持的变量类型有：布尔值、Float、Integer 和 Enums。

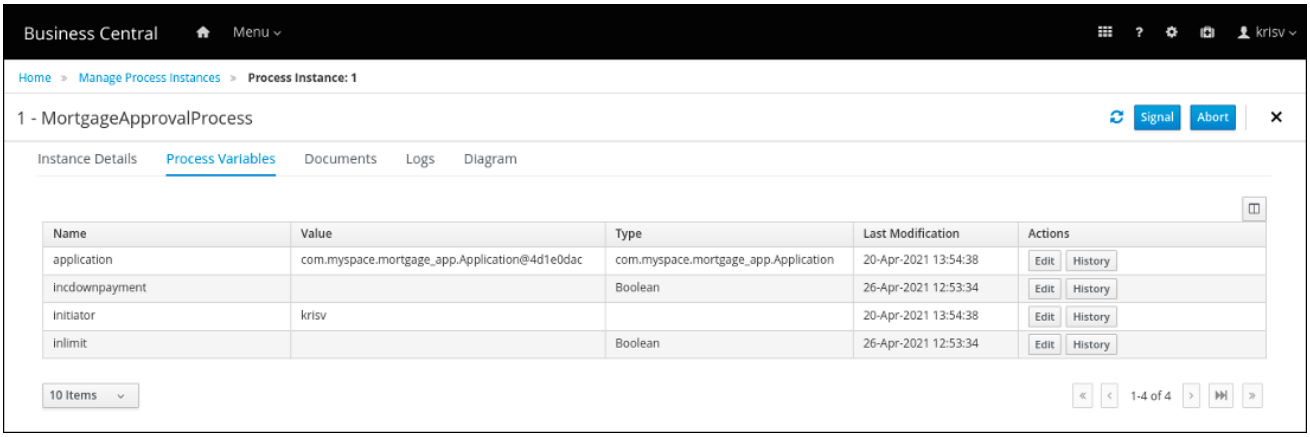
先决条件

- *您已在 Business Central 中创建了项目，并已启动一个进程实例。*

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 选择 Process Variables 选项卡，再点击 Edit 作为您要编辑的变量名称。
3. 添加或更改 变量值， 然后单击保存。

图 6.2. Business Central 中的变量值编辑按钮



第 7 章 操作脚本

操作脚本是定义 Script 属性或元素的拦截器操作的一种代码。操作脚本可以访问全局变量、进程变量和预定义的变量 kcontext。kcontext 是 ProcessContext 接口的实例。有关 kcontext 变量的更多信息，请参阅 ProcessContext Javadoc。

支持 Java 和 MVEL，作为操作脚本定义的选择。MVEL 接受有效的 Java 代码，另外还提供对参数嵌套访问的支持。例如，MVEL 调用 person.name 等同于 Java 调用 person.getName（）。

Java 和 MVEL dialects 中的 action 脚本示例

```
// Java dialect
System.out.println(person.getName());

// MVEL dialect
System.out.println(person.name);
```

您还可以使用操作脚本查看有关进程实例的信息。例如，使用以下命令：

- 返回进程实例的 ID：

  ```
  System.out.println(kcontext.getProcessInstance().getId());
  ```

- 如果进程实例有父实例，则返回父进程实例 ID：

  ```
  System.out.println(kcontext.getProcessInstance().getParentProcessInstanceId());
  ```

- 返回与进程实例相关的进程定义的 ID：

  ```
  System.out.println(kcontext.getProcessInstance().getProcessId());
  ```

- 返回与进程实例相关的进程定义名称：

```
System.out.println(kcontext.getProcessInstance().getProcessName());
```

- **返回进程实例的状态：**

```
System.out.println(kcontext.getProcessInstance().getState());
```

要在操作脚本中设置进程变量，请使用 kcontext.setVariable("VARIABLE_NAME", "VALUE")。

第 8 章 TIMERS

您可以使用计时器在特定周期后触发逻辑，也可以定期重复特定操作。计时器在触发一次或重复一次前等待预定义的时间。

8.1. RED HAT PROCESS AUTOMATION MANAGER 支持的计时器

Red Hat Process Automation Manager 支持两种类型的计时器：

- Quartz：用于 Spring Boot 和 Tomcat 的推荐
- EJB：推荐用于内部和红帽 OpenShift Container Platform 的 Red Hat JBoss EAP

注意

不要将计时器用于以下业务策略：

- 不要使用计时器作为您的进程轮询策略。例如，不直接调用外部服务并添加 1 秒计时器，而是使用 Fuse 注册异步路由。当收到预期响应后，使用 Fuse 回调来触发警报。在您的业务流程模型和符号(BPMN)流程模型之外，创建工作项目处理程序，并将计时器移到工作项目处理程序中。
- 不要在进程执行过程中使用计时器强制安全点或提交。计时器旨在代表业务流程中的时间段（持续时间），而不是实施特定于引擎的行为。

8.2. 使用延迟和周期配置计时器

您可以设置延迟和特定时间段的计时器。延迟指定节点激活后等待时间，周期定义后续触发器激活之间的时间。周期值 0 会产生一个一次性计时器。您可以在 [#d][#h][#m][#s][#s][#[ms]] 表单中指定延迟和周期表达式，指定天数、小时、分钟、秒和 毫秒（默认）。例如，表达式 1h 表示在再次触发计时器前的一小时等待时间。

8.3. 使用 ISO-8601 日期格式配置计时器

您可以使用支持一次性计时器和可重复计时器的 ISO-8601 日期格式配置计时器。您可以将计时器定义为日期和时间表示、时间持续时间或重复间隔。例如：

- 2020-12-24T20:00:00.000+02:00 表示计时器在 8:00 p.m 中完全触发。

- duration PT1S 表示计时器在一秒后触发一次。

- 重复间隔 R/PT1S 表示计时器以任何限值每秒触发。或者，R5/PT1S 会每秒触发计时器五次。

8.4. 使用进程变量配置计时器

您还可以使用进程变量指定计时器，其中包括延迟和周期或 ISO8601 日期格式的字符串表示。当您指定 #{variable} 时，引擎会解析表达式，并将表达式值替换为变量。在进程中，您可以使用以下方法使用计时器：

- 添加计时器事件到进程流。进程激活将启动计时器，并且触发计时器（重复出现或重复），它会激活计时器节点的成功。因此，带有正句点值的计时器的出站连接会多次触发。取消计时器节点时，相关的计时器也会被取消，且不会发生更多触发器。

- 将计时器作为边界事件与子进程或任务关联。

8.5. 更新正在运行的进程实例中的计时器

在某些情况下，需要重新调度的计时器以适应新要求，如更改延迟、周期或重复限制。更新计时器包括许多低级别操作，因此 Red Hat Process Automation Manager 提供以下 命令，以作为原子操作执行与更新计时器相关的低级别操作。以下命令可确保在同一事务中执行所有操作。

org.jbpm.process.instance.command.UpdateTimerCommand

注意

仅支持更新边界计时器事件和中间计时器事件。

您可以通过指定两个强制参数以及三个可选参数集合来 重新调度计时器。

表 8.1. UpdateTimerCommand 类的参数和参数集

参数或参数设置	类型
进程实例 ID(Mandatory)	**long**
计时器节点名称(Mandatory)	**字符串**
delay（可选）	**long**
period（可选）	**long**
重复限制（可选）	**init**

***rescheduling** 时间事件示例*

```
// Start the process instance and record its ID:
long id = kieSession.startProcess(BOUNDARY_PROCESS_NAME).getId();

// Set the timer delay to 3 seconds:
kieSession.execute(new UpdateTimerCommand(id,
BOUNDARY_TIMER_ATTACHED_TO_NAME, 3));
```

第 9 章 约束（CONSTRAINT）

约束是一个布尔值表达式，在执行包含约束的元素时评估。您可以在流程的不同部分使用限制，比如在分散网关中使用限制。

Red Hat Process Automation Manager 支持两种类型的限制，包括：

- 代码约束：在 Java、Javascript、Drools 或 MVEL 中定义的限制。代码约束可以访问工作内存中的数据，包括全局和进程变量。以下代码约束示例包含 人 作为进程中的变量：

  Java 代码约束示例

  ```
  return person.getAge() > 20;
  ```

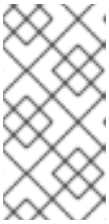
  MVEL 代码约束示例

  ```
  return person.age > 20;
  ```

  Javascript 代码约束示例

  ```
  person.age > 20
  ```

- 规则限制：以 DRL 规则条件的形式定义的限制。规则约束可以访问工作内存中的数据，包括全局变量。但是，规则限制无法直接访问进程中的变量，但使用进程实例。要检索父进程实例的引用，请使用类型 workflow ProcessInstance 的 processInstance 变量。



注意

您可以将进程实例插入到会话中，并根据需要更新它，例如，使用 Java 代码或一个 On-entry、on-exit 或显式操作。

以下示例显示了规则约束，搜索名称与进程中 name 的值相同的人。

带有进程变量分配的规则约束示例

```
processInstance : WorkflowProcessInstance()
Person( name == ( processInstance.getVariable("name") ) )
# add more constraints here ...
```

第 10 章 在 BUSINESS CENTRAL 中部署业务流程

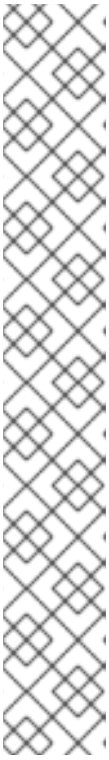
在 Business Central 中设计您的业务流程后，您可以在 Business Central 中构建和部署您的项目，从而使流程可供 KIE 服务器使用。

先决条件

- KIE 服务器已部署并连接到 Business Central。有关 KIE 服务器配置的更多信息，请参阅在 Red Hat JBoss EAP 7.4 上安装和配置 Red Hat Process Automation Manager。

流程

1. 在 Business Central 中，转至 Menu → Design → Projects。
2. 点您要部署的项目。
3. 单击 Deploy。



注意

您还可以选择 Build & Install 选项来构建项目，并将 KJAR 文件发布到配置的 Maven 存储库，而无需部署到 KIE 服务器。在开发环境中，您可以点击 Deploy 将构建的 KJAR 文件部署到 KIE 服务器，而无需停止任何正在运行的实例（如果适用），或者点击 Redeploy 来部署构建的 KJAR 文件并替换所有实例。下次部署或重新部署构建的 KJAR 时，以前的部署单元（KIE 容器）会在同一目标 KIE 服务器中自动更新。在生产环境中，Redeploy 选项被禁用，且只能点击 Deploy 将构建的 KJAR 文件部署到 KIE 服务器上的新部署单元（KIE 容器）。

要配置 KIE 服务器环境模式，请将 org.kie.server.mode 系统属性设置为 org.kie.server.mode=development 或 org.kie.server.mode=production。要在 Business Central 中为对应项目配置部署行为，请转至 Project Settings → General Settings → Version，再切换 Development Mode 选项。默认情况下，Business Central 中的 KIE 服务器和所有新项目均为开发模式。您不能部署打开开发模式的项目，或使用手动将 SNAPSHOT 版本后缀添加到生产模式中的 KIE 服务器。

要查看项目部署详情，可在屏幕顶部的部署横幅或 Deploy 下拉菜单中点击 View deployment details。这个选项将您定向到 Menu → Deploy → Execution Servers 页面。

第 11 章 在 BUSINESS CENTRAL 中执行业务流程

在构建和部署包含您的业务流程的项目后，您可以对业务流程执行定义的功能。

例如，此流程在 Business Central 中使用 Mortgage_Process 示例项目。在这种情况下，您要将数据输入为抵rtgage 应用程序表单中，作为抵rtgage 代理。MortgageApprovalProcess 业务流程运行，并确定申请者是否根据项目中定义的决策规则提供可接受的支付。业务流程的结束了规则测试或申请以增加购买所需的费用。如果应用程序通过了业务规则测试，银行批准者将审核申请并批准或拒绝贷款。

先决条件

- KIE 服务器已部署并连接到 Business Central。有关 KIE 服务器配置的更多信息，请参阅在 Red Hat JBoss EAP 7.4 上安装和配置 Red Hat Process Automation Manager。

流程

1. 在 Business Central 中，转至 Menu → Projects 并选择空格。默认空间为 MySpace。
2. 在窗口的右上角，单击 Add Project 旁边的箭头，然后选择 Try Samples。
3. 选择 Mortgage_Process 示例并点 Ok。
4. 在项目页面中，选择 Mortgage_Process。
5. 在 Mortgage_Process 页面中，单击 Build。
6. 项目构建后，单击 Deploy。
7. 进入 Menu → Manage → Process Definitions。
8. 单击 MortgageApprovalProcess 行中的任意位置，以查看进程详情。

9. 点击 图表 选项卡，在编辑器中查看业务流程图。

10. 点击 New Process Instance 打开 Application 表单，并在表单中输入以下值：

    - 停机支付 ：30000
    - 投影特年 ：10
    - 名称 ：Ivo
    - 年度 Income:60000
    - SSN:123456789
    - 属性的年龄 ：8
    - 属性的地址 ：Brno
    - locale:Rural
    - 班加法案 ：50000

11. 单击 Submit 以启动新进程实例。启动进程实例后，Instance Details 视图将打开。

12. 单击 图表 选项卡，以查看进程图表中的进程流。在各个任务移动过程中，会突出显示进程的状态。

13. 点 Menu → Manage → Tasks。

在本例中，处理相应任务的用户或用户是以下组的成员：

- approver: 用于 Qualify 任务
- 代理 ： 用于检测数据 并增加 故障支付 任务
- Manager: 用于 最后批准 任务

14. 作为批准人，查看 Qualify 任务信息，单击 Claim 以启动该任务，然后选择 Is mortgage application is limit?，然后单击 Complete 以完成任务流。

15. 在 Tasks 页面中，点击 Final Approval 行的任意位置，以打开 Final Approval 任务。

16. 点 Claim 来声明负责任务，然后单击 Complete 以完成 loan Approval process。

注意

Save 和 Release 按钮仅用于暂停批准过程，并在等待某个字段值时保存实例，或者释放另一个用户修改的任务。

第 12 章 测试业务流程

业务流程可以动态更新，这可能会导致错误，因此测试过程业务也是与任何其他开发工件类似的业务流程生命周期的一部分。

业务流程的单元测试可确保具体用例中过程的行为如预期。例如，您可以基于特定输入测试输出。为了简化单元测试，Red Hat Process Automation Manager 包括 org.jbpm.test.JbpmJUnitBaseTestCase 类。

JbpmJUnitBaseTestCase 作为基础测试案例执行，它用于红帽流程自动化管理器相关测试。JbpmJUnitBaseTestCase 提供以下使用区域：

- JUnit 生命周期方法

表 12.1. JUnit 生命周期方法

方法	描述
setUp	此方法标注为 **@Before**。它配置数据源和 **EntityManageronnectionFactoryy**，并删除单例的会话 ID。
tearDown	此方法标注为 **@After**。它移除了历史、关闭实**体管理器** 和数据源，并处理 **RuntimeManager** 和 **RuntimeEngines**。

- 知识库和知识会话管理方法：要创建一个会话，创建 RuntimeManager 和 RuntimeEngine。使用以下方法创建和关闭 RuntimeManager ：

表 12.2. RuntimeManager 和 RuntimeEngine 管理方法

方法	描述
createRuntimeManager	为给定的资产和选定策略创建 **RuntimeManager**。
disposeRuntimeManager	在测试范围内激活的 **RuntimeManager**。
getRuntimeEngine	为给定的上下文创建新的 **RuntimeEngine**。

- 断言：要测试资产的状态，请使用以下方法之一：

表 12.3. RuntimeManager 和 RuntimeEngine 管理方法

表 12.3. RuntimeManager 和 RuntimeEngine 管理方法

断言	描述
assertProcessInstanceActive(long processInstanceId, KieSession ksession)	验证具有给定 **processInstanceId** 的进程实例是否活跃。
assertProcessInstanceCompleted（长 processInstanceId）	验证具有给定 **processInstanceId** 的进程实例是否已完成。如果启用了会话持久性，您可以使用此方法，否则使用 **assertProcessInstanceNotActive(long processInstanceId, KieSession ksession)**。
assertProcessInstanceAborted（长 processInstanceId）	验证具有给定 **processInstanceId** 的进程实例是否已中止。如果启用了会话持久性，您可以使用此方法，否则使用 **assertProcessInstanceNotActive(long processInstanceId, KieSession ksession)**。
assertNodeExists(ProcessInstance process, String… nodeNames)	验证指定进程是否包含给定的节点。
assertNodeActive(long processInstanceId, KieSession ksession, String… name)	验证具有给定 **processInstanceId** 的进程实例是否至少包含一个具有指定节点名称的活跃节点。
assertNodeTriggered(long processInstanceId, String… nodeNames)	验证在执行指定进程实例的过程中是否为每个给定节点触发节点实例。
assertProcessVarExists(ProcessInstance process, String… processVarNames)	验证给定进程是否包含指定的进程变量。
assertProcessNameEquals(ProcessInstance process, String name)	验证给定名称是否与指定进程名称匹配。
assertVersionEquals(ProcessInstance process, String version)	验证给定进程版本是否与指定进程版本匹配。

- ***帮助程序方法：使用以下方法为具有或不使用持久性的给定进程集合创建新的 RuntimeManager 和 RuntimeEngine。有关持久性的更多信息，请参阅 Red Hat Process Automation Manager 中的进程引擎。***

表 12.4. RuntimeManager 和 RuntimeEngine 管理方法

方法	描述
setupPoolingDataSource	配置数据源。
getDs	返回配置的数据源。
getEmf	返回配置的 实体管理器onnectionFactoryy。
getTestWorkItemHandler	返回除了默认的工作项处理程序之外可以注册的测试工作项处理程序。
clearHistory	清除历史记录日志。

以下示例包含一个起始事件、脚本任务和结束事件。JUnit 测试示例会创建一个新的会话，并启动 hello.bpmn 进程，并验证进程实例是否已完成，以及 StartProcess、Hello 和 EndProcess 节点是否已执行。

图 12.1. hello.bpmn 进程的 JUnit 测试示例


```
public class ProcessPersistenceTest extends JbpmJUnitBaseTestCase {

    public ProcessPersistenceTest() {
        super(true, true);
    }

    @Test
    public void testProcess() {

        createRuntimeManager("hello.bpmn");

        RuntimeEngine runtimeEngine = getRuntimeEngine();

        KieSession ksession = runtimeEngine.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello");

        assertProcessInstanceNotActive(processInstance.getId(), ksession);
```

```
        assertNodeTriggered(processInstance.getId(), "StartProcess", "Hello", "EndProcess");
    }
}
```

JbpmJUnitBaseTestCase 支持所有预定义的 RuntimeManager 策略，作为单元测试的一部分。因此，在创建一个 RuntimeManager 作为单个测试的一部分，它足以指定创建 RuntimeManager 时使用的策略。以下示例演示了在任务服务中使用 PerProcessInstance 策略来管理用户任务：

```
public class ProcessHumanTaskTest extends JbpmJUnitBaseTestCase {

    private static final Logger logger =
LoggerFactory.getLogger(ProcessHumanTaskTest.class);

    public ProcessHumanTaskTest() {
        super(true, false);
    }

    @Test
    public void testProcessProcessInstanceStrategy() {
        RuntimeManager manager = createRuntimeManager(Strategy.PROCESS_INSTANCE,
"manager", "humantask.bpmn");
        RuntimeEngine runtimeEngine = getRuntimeEngine(ProcessInstanceIdContext.get());
        KieSession ksession = runtimeEngine.getKieSession();
        TaskService taskService = runtimeEngine.getTaskService();

        int ksessionID = ksession.getId();
        ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello");

        assertProcessInstanceActive(processInstance.getId(), ksession);
        assertNodeTriggered(processInstance.getId(), "Start", "Task 1");

        manager.disposeRuntimeEngine(runtimeEngine);
        runtimeEngine =
getRuntimeEngine(ProcessInstanceIdContext.get(processInstance.getId()));

        ksession = runtimeEngine.getKieSession();
        taskService = runtimeEngine.getTaskService();

        assertEquals(ksessionID, ksession.getId());

        // let John execute Task 1
        List<TaskSummary> list = taskService.getTasksAssignedAsPotentialOwner("john", "en-
UK");
        TaskSummary task = list.get(0);
        logger.info("John is executing task {}", task.getName());
        taskService.start(task.getId(), "john");
        taskService.complete(task.getId(), "john", null);

        assertNodeTriggered(processInstance.getId(), "Task 2");

        // let Mary execute Task 2
        list = taskService.getTasksAssignedAsPotentialOwner("mary", "en-UK");
        task = list.get(0);
        logger.info("Mary is executing task {}", task.getName());
```

```
        taskService.start(task.getId(), "mary");
        taskService.complete(task.getId(), "mary", null);

        assertNodeTriggered(processInstance.getId(), "End");
        assertProcessInstanceNotActive(processInstance.getId(), ksession);
    }
}
```

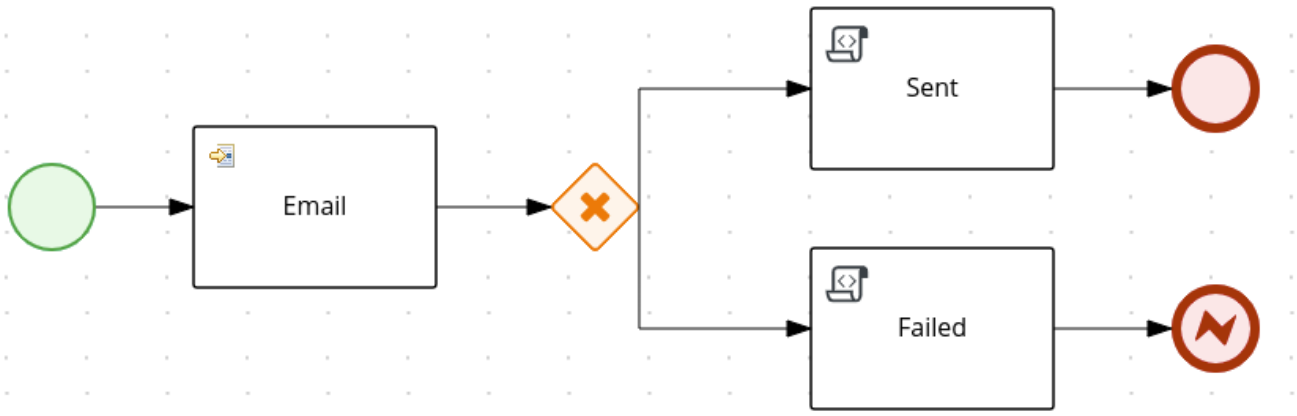
12.1. 测试与外部服务集成

业务流程通常包括调用外部服务。业务流程的单元测试允许您注册测试处理程序，以验证是否正确请求特定服务，并为请求的服务提供测试响应。

要测试与外部服务的交互，请使用 default TestWorkItemHandler 处理程序。您可以注册 TestWorkItemHandler，以收集特定类型的所有工作项目。另外，testWorkItemHandler 包含与任务相关的数据。工作项目代表一个工作单元，如发送特定电子邮件或调用特定服务。TestWorkItemHandler 会在执行过程中验证是否请求特定的工作项，以及相关的数据是否正确。

以下示例演示了如何验证电子邮件任务以及是否收到电子邮件是否引发异常。单元测试使用请求电子邮件时执行的测试处理器，可让您测试与电子邮件相关的数据，如发件人和接收者。在 abortWorkItem（） 方法通知引擎电子邮件发送失败后，单元测试将通过生成错误并记录操作来验证进程是否处理此类情况。在这种情况下，进程实例最终会被中止。

图 12.2. 电子邮件进程示例




```
public void testProcess2() {

    createRuntimeManager("sample-process.bpmn");

    RuntimeEngine runtimeEngine = getRuntimeEngine();

    KieSession ksession = runtimeEngine.getKieSession();

    TestWorkItemHandler testHandler = getTestWorkItemHandler();

    ksession.getWorkItemManager().registerWorkItemHandler("Email", testHandler);
```

```
    ProcessInstance processInstance = ksession.startProcess("com.sample.bpmn.hello2");

    assertProcessInstanceActive(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "StartProcess", "Email");

    WorkItem workItem = testHandler.getWorkItem();
    assertNotNull(workItem);
    assertEquals("Email", workItem.getName());
    assertEquals("me@mail.com", workItem.getParameter("From"));
    assertEquals("you@mail.com", workItem.getParameter("To"));

    ksession.getWorkItemManager().abortWorkItem(workItem.getId());
    assertProcessInstanceNotActive(processInstance.getId(), ksession);
    assertNodeTriggered(processInstance.getId(), "Gateway", "Failed", "Error");

}
```

第 13 章 BUSINESS CENTRAL 中的进程定义和流程实例

进程定义是业务流程模型和符号(BPMN)2.0 文件，充当进程及其 BPMN 图表的容器。进程定义显示有关业务流程的所有可用信息，如任何关联的子进程或参与所选定义的用户和组数量。

进程定义也定义了 导入 条目，供进程定义使用，以及进程定义 的关系 条目。

进程定义的 BPMN2 源

```
<definitions id="Definition"
             targetNamespace="http://www.jboss.org/drools"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"Rule Task
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
             xmlns:g="http://www.jboss.org/drools/flow/gpd"
             xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
             xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
             xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
             xmlns:tns="http://www.jboss.org/drools">

    <process>
      PROCESS
    </process>

    <bpmndi:BPMNDiagram>
     BPMN DIAGRAM DEFINITION
    </bpmndi:BPMNDiagram>

    </definitions>
```

创建、配置并部署包含您的业务流程的项目后，您可以查看 Business Central Menu → Manage → Process Definitions 中所有进程定义的列表。您可以通过单击右上角的刷新按钮，随时刷新已部署的进程定义列表。

进程定义列表显示部署到平台上的所有可用进程定义。单击列出的任何进程定义，以显示对应的进程定义详细信息。这将显示有关进程定义的信息，例如，如果存在关联了子进程，或者进程定义中存在多少个用户和组。进程定义详情页面中的 Diagram 选项卡包含基于 BPMN2 的进程定义图。

在每个选定进程定义中，您可以通过单击右上角的 New Process Instance 按钮来启动进程定义的新进程实例。从可用进程定义启动的实例列在 Menu → Manage → Process Instances 中。

您还可以在 Manage 下拉菜单下为所有用户定义默认的分页选项（流程定义、流程 实例、任务、作业和执行 错误）以及 Menu → Track → Task Inbox 中。

有关 Business Central 中的流程和任务 管理的更多信息，请参阅 Business Central 中管理和监控业务流程。

13.1. 从进程定义页面中启动进程实例

您可以在 Menu → Manage → Process Definitions 中启动进程实例。这可用于您同时处理多个项目或进程定义的环境。

先决条件

- 已在 Business Central 中部署了带有进程定义的项目。

流程

1. 在 Business Central 中，转至 Menu → Manage → Process Definitions。
2. 从列表中选择您要启动新进程实例的进程定义。该定义的详情页面将打开。
3. 单击右上角的 New Process Instance，以启动新的进程实例。
4. 为进程实例提供任何所需信息。
5. 单击 Submit 以创建进程实例。
6. 在 Menu → Manage → Process Instances 中查看新进程实例。

13.2. 从进程实例页面中启动进程实例

您可以创建新进程实例，或者查看 Menu → Manage → Process Instances 中所有运行的进程实例的列表。

先决条件

- 已在 Business Central 中部署了带有进程定义的项目。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 单击右上角的 New Process Instance，再从下拉列表中选择您要启动新进程实例的进程定义。
3. 提供启动新进程实例所需的任何信息。
4. 单击 Start 以创建进程实例。

   新进程实例会出现在 Manage Process Instances 列表中。

13.3. XML 中的进程定义

您可以使用 BPMN 2.0 规范直接以 XML 格式创建进程。这些 XML 进程的语法使用 BPMN 2.0 XML 架构定义进行定义。

进程 XML 文件由以下核心部分组成：

- 进程 ：这是包含不同节点的定义及其属性的进程 XML 的主要部分。进程 XML 文件只包含一个 < process> 元素。此元素包含与进程相关的参数（its 类型、名称、ID 和软件包名称），并且由三个子子组成：一个标题部分，其中进程级别信息（如变量、全局、导入和 lanes ）定义了节点部分，用于定义该进程中的每个节点，以及一个包含进程中所有节点之间的连接部分。
- BPMNDiagram ：这是包含所有图形信息（如节点位置）的进程 XML 文件的下一部分。nodes 部分包含每个节点的特定元素，并为该节点类型定义各种参数和任何子元素。

以下进程 XML 文件片段显示包含启动事件序列的简单进程，一个脚本任务将 "Hello World" 打印到控制台，以及一个结束事件：

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions
  id="Definition"
  targetNamespace="http://www.jboss.org/drools"
  typeLanguage="http://www.java.com/javaTypes"
  expressionLanguage="http://www.mvel.org/2.0"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
  xmlns:g="http://www.jboss.org/drools/flow/gpd"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
  xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
  xmlns:tns="http://www.jboss.org/drools">

  <process processType="Private" isExecutable="true" id="com.sample.hello" name="Hello Process">
    <!-- nodes -->
    <startEvent id="_1" name="Start" />

    <scriptTask id="_2" name="Hello">
      <script>System.out.println("Hello World");</script>
    </scriptTask>

    <endEvent id="_3" name="End" >
      <terminateEventDefinition/>
    </endEvent>

    <!-- connections -->

    <sequenceFlow id="_1-_2" sourceRef="_1" targetRef="_2" />
    <sequenceFlow id="_2-_3" sourceRef="_2" targetRef="_3" />
  </process>

  <bpmndi:BPMNDiagram>
    <bpmndi:BPMNPlane bpmnElement="com.sample.hello" >

      <bpmndi:BPMNShape bpmnElement="_1" >
        <dc:Bounds x="16" y="16" width="48" height="48" />
      </bpmndi:BPMNShape>

      <bpmndi:BPMNShape bpmnElement="_2" >
        <dc:Bounds x="96" y="16" width="80" height="48" />
      </bpmndi:BPMNShape>

      <bpmndi:BPMNShape bpmnElement="_3" >
        <dc:Bounds x="208" y="16" width="48" height="48" />
      </bpmndi:BPMNShape>

      <bpmndi:BPMNEdge bpmnElement="_1-_2" >
        <di:waypoint x="40" y="40" />
```

```
      <di:waypoint x="136" y="40" />
    </bpmndi:BPMNEdge>

    <bpmndi:BPMNEdge bpmnElement="_2-_3" >
      <di:waypoint x="136" y="40" />
      <di:waypoint x="232" y="40" />
    </bpmndi:BPMNEdge>

  </bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>

</definitions>
```

第 14 章 BUSINESS CENTRAL 中的表单

表单是页面的布局定义，定义为 HTML，在进程和任务实例化过程中作为用户的对话框窗口显示。任务表单从用户获取进程和任务实例执行的数据，而进程表单则取进程变量的输入和输出。

然后，使用数据输入分配将输入映射到任务，您可以在任务内部使用。任务完成后，数据映射为数据输出分配，以便为父进程实例提供数据。

14.1. 表单模型器

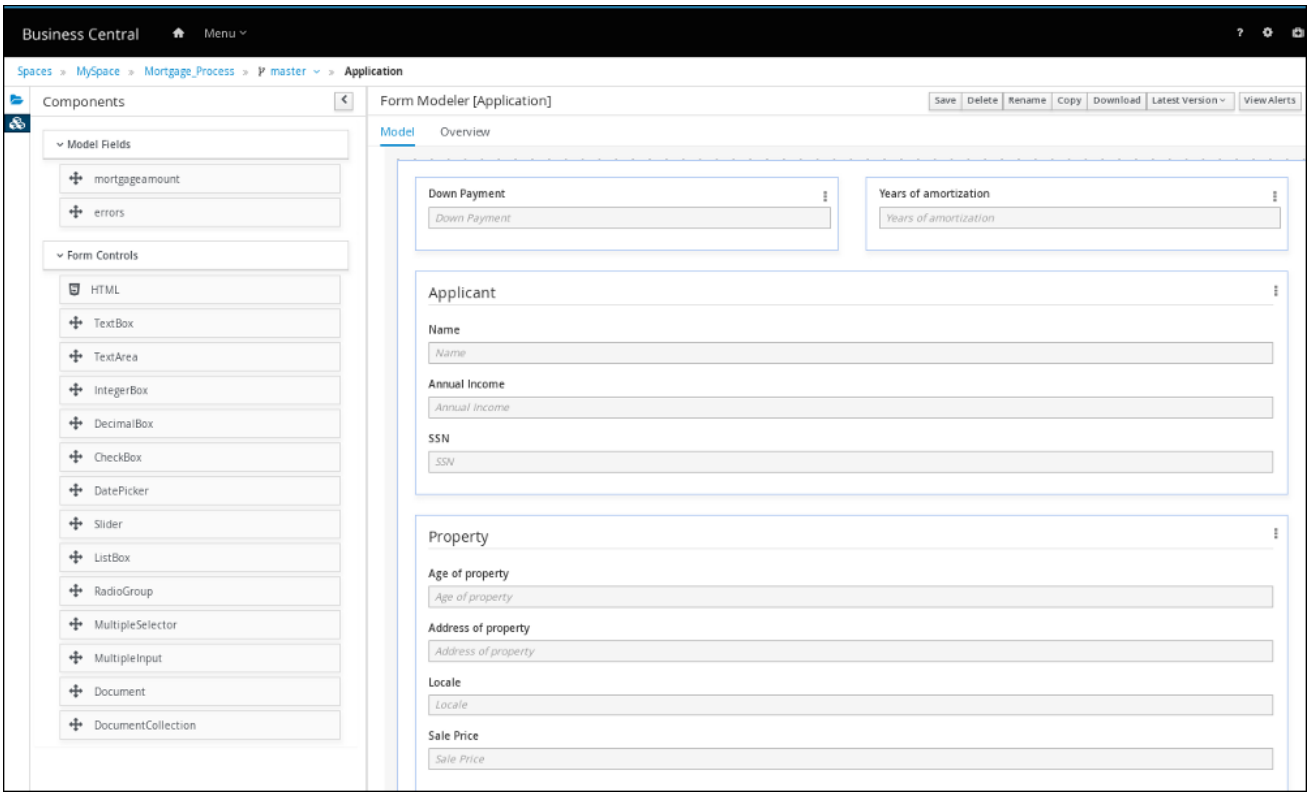
Red Hat Process Automation Manager 提供了一个自定义编辑器来定义表单，称为 Form Modeler。使用 Form Modeler，您可以在不编写代码的情况下为数据对象、任务表单和进程启动表单生成表单。表单模型程序包含一个用于绑定多个数据类型和回调机制的小部件库，用于在表单值更改时发送通知。表单建模器使用基于an 的验证，并支持将表单字段绑定到静态或动态模型。

表单模型程序包括以下功能：

- 表单建模用户界面以进行表单
- 表格从数据模型或 Java 对象自动生成
- Java 对象的数据绑定
- 公式和表达式
- 自定义表单布局
- 嵌入的形式

表单模型程序附带您放入 Canvas 的预定义字段类型来创建表单。

图 14.1. mortgage loan 应用程序表单示例



14.2. 在 BUSINESS CENTRAL 中生成流程和任务表单

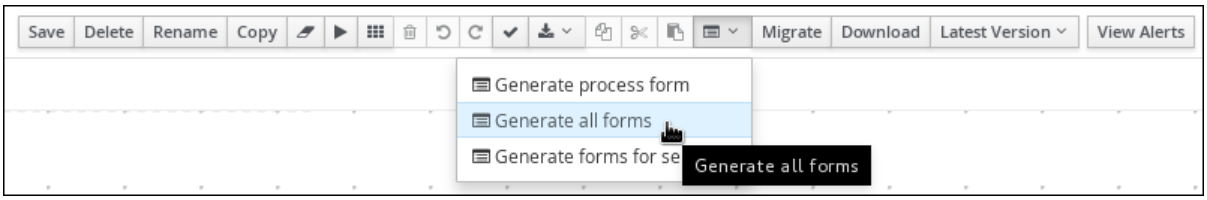
您可以从业务流程中生成流程表单，该表格显示在流程实例化上，供其实例化流程的用户。您还可以从您的业务流程中生成任务表单，在用户任务实例化时显示，当执行流到达任务时，到用户任务的代理。

流程

1. *在 Business Central 中，转至 Menu → Design → Projects。*
2. *点击项目名称以打开资产视图，然后点业务流程名称。*
3. *在进程设计器中，点您要为其创建表单的进程任务（如果适用）。*
4. *在右上角工具栏中，点击 Form Generation 图标并选择您想生成的表单：*
   - *生成进程表单 ：生成整个过程的表单。这是用户启动后必须完成的初始表单。*

- *生成所有表单 ：为整个进程和所有用户任务生成表单。*
- *为选择 生成表单：为所选用户任务节点生成表单。*

图 14.2. 表单生成菜单



表单在项目的根目录中创建。

5. *进入 Business Central 中的项目根目录，单击新的表单名称，然后使用 Form Modeler 定制表单以满足您的要求。*

14.3. 在 BUSINESS CENTRAL 中手动创建表单

您可以从项目资产视图手动创建任务和流程表单。这是生成表单的另一种方法，而无需选择从您的业务流程生成表单。例如，Form Modeler 现在支持从外部数据对象创建表单。

流程

1. *在 Business Central 中，前往 Menu → Design → Projects 并点项目名称。*
2. *点 Add Asset → Form。*
3. *在 Create new Form 窗口中提供以下信息：*
   - *表单名称（必须为唯一）*
   - *软件包名称*

- 模型类型：选择 业务流程 或 数据对象。
  - 对于 业务流程 模型类型，从 Select Process 下拉菜单中选择您的引导过程，然后从 Select Form 下拉菜单中选择您要创建的表单。
  - 对于 Data Object model 类型，请从 Select Data Object from Project 下拉菜单中选择其中一个项目数据对象。

4. 点 Ok 打开 Form Modeler。

5. 在 Form Modeler 左侧的 组件 视图中，展开 Model Fields 和 Form Controls 菜单，并通过将所需字段和表单控制拖到 canvas 来创建一个新的表单。

6. 点 Save 保存您的更改。

14.4. 以表单或流程形式进行文档附件

Red Hat Process Automation Manager 支持使用 Document 表单字段以表单形式进行文档附加。使用 Document 表单字段，您可以上传表单或流程中所需的文档。

要以表单和流程启用文档附件，请完成以下步骤：

1. 设置文档总结策略。

2. 在业务流程中创建文档变量。

3. 将任务输入和输出映射到 document 变量。

14.4.1. 设置文档摘要策略

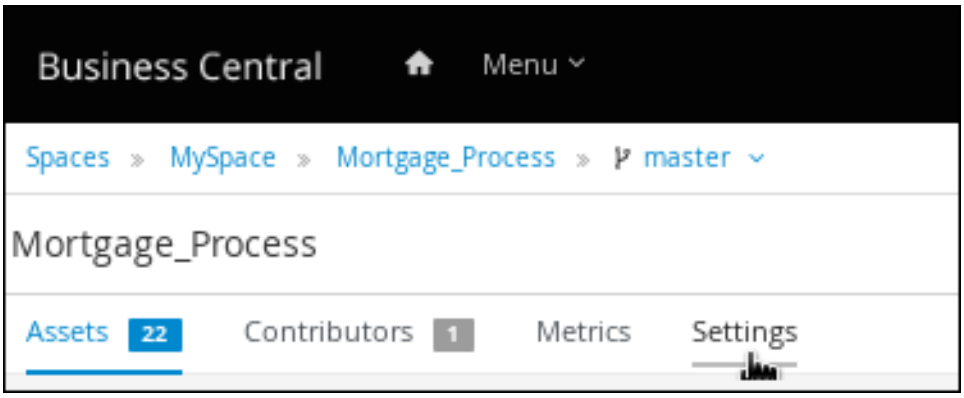
项目的文档总结策略决定了文档与表单和流程相关的存储位置。Red Hat Process Automation Manager 中的默认文档摘要策略是 org.jbpm.document.marshalling.DocumentMarshallingStrategy。此策略使用

DocumentStorageServiceImpl 类，将文档存储在本地存储在 PROJECT_HOME/.docs 文件夹中。您可以在 Business Central 或 kie-deployment-descriptor.xml 文件中为项目设置本文档 marshalling 策略或自定义文档 marshalling 策略。

流程

1. *在 Business Central 中，转至 Menu → Design → Projects。*
2. *选择项目。project Assets 窗口将打开。*
3. *点 Settings 标签页。*

图 14.3. 设置标签页



4. *点 Deployments → MarshallingStrategy → Add Marshalling Strategy。*
5. *在 Name 字段中输入文档摘要策略的标识符，并在 Resolver 下拉菜单中选择对应的解析器类型：*

- *对于单文：输入 org.jbpm.document.marshalling.DocumentMarshallingStrategy 作为文档总结策略，并将解析器类型设置为 Reflection。*
- *对于多个文档：输入 新的 org.jbpm.document.marshalling.DocumentCollectionImplMarshallingStrategy（new org.jbpm.document.marshalling.DocumentMarshallingStrategy（）） 作为文档 marshalling 策略，并将解析器类型设置为 MVEL。*
- *对于自定义文档支持：输入自定义文档总结策略的标识符并选择相关解析器类型。*

6. 单击 **Test** 以验证您的部署描述符文件。

7. 单击 **Deploy to build** 并部署更新的项目。

或者，如果您不使用 **Business Central**，您可以导航到 **PROJECT_HOME/src/main/resources/META_INF/kie-deployment-descriptor.xml**（如果适用）并使用所需的 **< marshalling-strategies >** 元素编辑部署描述符文件。

8. 点击 **Save**。

具有多个文档的部署描述符文件示例

```
<deployment-descriptor
  xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <persistence-unit>org.jbpm.domain</persistence-unit>
 <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
 <audit-mode>JPA</audit-mode>
 <persistence-mode>JPA</persistence-mode>
 <runtime-strategy>SINGLETON</runtime-strategy>
 <marshalling-strategies>
  <marshalling-strategy>
   <resolver>mvel</resolver>
   <identifier>new
org.jbpm.document.marshalling.DocumentCollectionImplMarshallingStrategy(new
org.jbpm.document.marshalling.DocumentMarshallingStrategy());</identifier>
  </marshalling-strategy>
 </marshalling-strategies>
```

14.4.1.1. 使用自定义文档发布内容管理系统(CMS)

项目的文档总结策略决定了文档与表单和流程相关的存储位置。**Red Hat Process Automation Manager** 中的默认文档摘要策略是 **org.jbpm.document.marshalling.DocumentMarshallingStrategy**。此策略使用 **DocumentStorageServiceImpl** 类，将文档存储在本地存储在 **PROJECT_HOME/.docs** 文件夹中。如果要在自定义位置存储表单和流程文档，如集中内容管理系统(CMS)，请在项目中添加自定义文档摘要策略。您可以在 **Business Central** 或 **kie-deployment-descriptor.xml** 文件中设置本文档。

流程

1. 创建自定义 marshalling 策略 .java 文件，其中包含 org.kie.api.marshalling.ObjectMarshallingStrategy 接口的实施。这个接口可让您实施自定义文档总结策略所需的变量持久性。

这个接口中的以下方法可帮助您创建策略：

- 布尔值接受（Object 对象）：确定策略是否可以被汇总指定对象
- byte[] marshal(Context context, ObjectOutput os, Object object): Marshals the specified 对象并将 marshalled 对象返回为 byte[]
- Object unmarshal（Context context, ObjectInputStream is, byte[] 对象, ClassLoader classloader）：阅读作为 byte[] 接收的对象并返回 unmarshalled 对象
- void write(ObjectOutputStream os, Object object): Same 作为 marshal 方法，提供向后兼容
- 对象 read(ObjectInputStream os): Same 作为 unmarshal 方法，为向后兼容提供

以下代码示例是 ObjectMarshallingStrategy 实施示例，用于从 Content Management Interoperability Services(CMIS)系统存储和检索数据：

从 CMIS 系统存储和检索数据的实现示例

```
package org.jbpm.integration.cmis.impl;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.HashMap;

import org.apache.chemistry.opencmis.client.api.Folder;
import org.apache.chemistry.opencmis.client.api.Session;
import org.apache.chemistry.opencmis.commons.data.ContentStream;
import org.apache.commons.io.IOUtils;
import org.drools.core.common.DroolsObjectInputStream;
import org.jbpm.document.Document;
```

```
import org.jbpm.integration.cmis.UpdateMode;

import org.kie.api.marshalling.ObjectMarshallingStrategy;

public class OpenCMISPlaceholderResolverStrategy extends OpenCMISSupport
implements ObjectMarshallingStrategy {

 private String user;
 private String password;
 private String url;
 private String repository;
 private String contentUrl;
 private UpdateMode mode = UpdateMode.OVERRIDE;

 public OpenCMISPlaceholderResolverStrategy(String user, String password, String
url, String repository) {
  this.user = user;
  this.password = password;
  this.url = url;
  this.repository = repository;
 }

 public OpenCMISPlaceholderResolverStrategy(String user, String password, String
url, String repository, UpdateMode mode) {
  this.user = user;
  this.password = password;
  this.url = url;
  this.repository = repository;
  this.mode = mode;
 }

    public OpenCMISPlaceholderResolverStrategy(String user, String password, String
url, String repository, String contentUrl) {
        this.user = user;
        this.password = password;
        this.url = url;
        this.repository = repository;
        this.contentUrl = contentUrl;
    }

    public OpenCMISPlaceholderResolverStrategy(String user, String password, String
url, String repository, String contentUrl, UpdateMode mode) {
        this.user = user;
        this.password = password;
        this.url = url;
        this.repository = repository;
        this.contentUrl = contentUrl;
        this.mode = mode;
    }

 public boolean accept(Object object) {
  if (object instanceof Document) {
   return true;
  }
  return false;
 }
```

```
 public byte[] marshal(Context context, ObjectOutputStream os, Object object) throws
IOException {
  Document document = (Document) object;
  Session session = getRepositorySession(user, password, url, repository);
  try {
   if (document.getContent() != null) {
    String type = getType(document);
    if (document.getIdentifier() == null || document.getIdentifier().isEmpty()) {
     String location = getLocation(document);

     Folder parent = findFolderForPath(session, location);
     if (parent == null) {
      parent = createFolder(session, null, location);
     }
     org.apache.chemistry.opencmis.client.api.Document doc =
createDocument(session, parent, document.getName(), type, document.getContent());
     document.setIdentifier(doc.getId());
     document.addAttribute("updated", "true");
    } else {
     if (document.getContent() != null &&
"true".equals(document.getAttribute("updated"))) {
      org.apache.chemistry.opencmis.client.api.Document doc =
updateDocument(session, document.getIdentifier(), type, document.getContent(),
mode);

      document.setIdentifier(doc.getId());
      document.addAttribute("updated", "false");
     }
    }
   }
   ByteArrayOutputStream buff = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream( buff );
        oos.writeUTF(document.getIdentifier());
        oos.writeUTF(object.getClass().getCanonicalName());
        oos.close();
        return buff.toByteArray();
  } finally {
   session.clear();
  }
 }

 public Object unmarshal(Context context, ObjectInputStream ois, byte[] object,
ClassLoader classloader) throws IOException, ClassNotFoundException {
  DroolsObjectInputStream is = new DroolsObjectInputStream( new
ByteArrayInputStream( object ), classloader );
  String objectId = is.readUTF();
  String canonicalName = is.readUTF();
  Session session = getRepositorySession(user, password, url, repository);
  try {
   org.apache.chemistry.opencmis.client.api.Document doc =
(org.apache.chemistry.opencmis.client.api.Document) findObjectForId(session,
objectId);
   Document document = (Document) Class.forName(canonicalName).newInstance();
   document.setAttributes(new HashMap<String, String>());
```

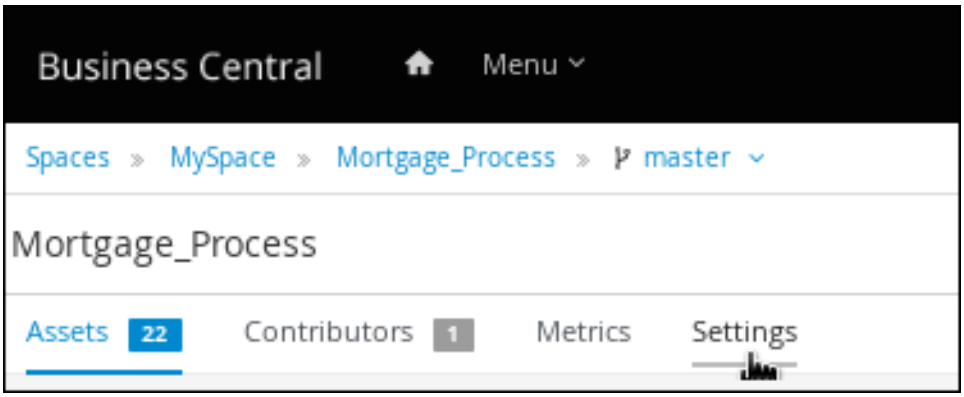
```
   document.setIdentifier(objectId);
   document.setName(doc.getName());
   document.setLastModified(doc.getLastModificationDate().getTime());
   document.setSize(doc.getContentStreamLength());
   document.addAttribute("location", getFolderName(doc.getParents()) +
getPathAsString(doc.getPaths()));
   if (doc.getContentStream() != null && contentUrl == null) {
    ContentStream stream = doc.getContentStream();
    document.setContent(IOUtils.toByteArray(stream.getStream()));
    document.addAttribute("updated", "false");
    document.addAttribute("type", stream.getMimeType());
   } else {
      document.setLink(contentUrl + document.getIdentifier());
   }
   return document;
  } catch(Exception e) {
   throw new RuntimeException("Cannot read document from CMIS", e);
  } finally {
   is.close();
   session.clear();
  }
 }

 public Context createContext() {
  return null;
 }

 // For backward compatibility with previous serialization mechanism
 public void write(ObjectOutputStream os, Object object) throws IOException {
  Document document = (Document) object;
  Session session = getRepositorySession(user, password, url, repository);
  try {
   if (document.getContent() != null) {
    String type = document.getAttribute("type");
    if (document.getIdentifier() == null) {
     String location = document.getAttribute("location");

     Folder parent = findFolderForPath(session, location);
     if (parent == null) {
      parent = createFolder(session, null, location);
     }
     org.apache.chemistry.opencmis.client.api.Document doc =
createDocument(session, parent, document.getName(), type, document.getContent());
     document.setIdentifier(doc.getId());
     document.addAttribute("updated", "false");
    } else {
     if (document.getContent() != null &&
"true".equals(document.getAttribute("updated"))) {
      org.apache.chemistry.opencmis.client.api.Document doc =
updateDocument(session, document.getIdentifier(), type, document.getContent(),
mode);

      document.setIdentifier(doc.getId());
      document.addAttribute("updated", "false");
     }
    }
```

```
    }
    ByteArrayOutputStream buff = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream( buff );
        oos.writeUTF(document.getIdentifier());
        oos.writeUTF(object.getClass().getCanonicalName());
        oos.close();
   } finally {
    session.clear();
   }
  }

  public Object read(ObjectInputStream os) throws IOException,
ClassNotFoundException {
   String objectId = os.readUTF();
   String canonicalName = os.readUTF();
   Session session = getRepositorySession(user, password, url, repository);
   try {
    org.apache.chemistry.opencmis.client.api.Document doc =
(org.apache.chemistry.opencmis.client.api.Document) findObjectForId(session,
objectId);
    Document document = (Document) Class.forName(canonicalName).newInstance();

    document.setIdentifier(objectId);
    document.setName(doc.getName());
    document.addAttribute("location", getFolderName(doc.getParents()) +
getPathAsString(doc.getPaths()));
    if (doc.getContentStream() != null) {
     ContentStream stream = doc.getContentStream();
     document.setContent(IOUtils.toByteArray(stream.getStream()));
     document.addAttribute("updated", "false");
     document.addAttribute("type", stream.getMimeType());
    }
    return document;
   } catch(Exception e) {
    throw new RuntimeException("Cannot read document from CMIS", e);
   } finally {
    session.clear();
   }
  }

}
```

2. 在 Business Central 中，转至 Menu → Design → Projects。

3. 单击项目名称，再单击 Settings。

图 14.4. 设置标签页



4. 点 Deployments → MarshallingStrategy → Add Marshalling Strategy。

5. 在 Name 字段中输入自定义文档 marshalling 策略的标识符，如 org.jbpm.integration.cmis.impl.OpenCMISPlaceholderResolverStrategy。

6. 从 Resolver 下拉菜单中选择相关选项，如本例中的 Reflection。

7. 单击 Test 以验证您的部署描述符文件。

8. 单击 Deploy to build 并部署更新的项目。

或者，如果您不使用 Business Central，您可以导航到 PROJECT_HOME/src/main/resources/META_INF/kie-deployment-descriptor.xml （如果适用）并使用所需的 < marshalling-strategies > 元素编辑部署描述符文件。

具有自定义文档总结策略的部署描述符文件示例

```
<deployment-descriptor
  xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <persistence-unit>org.jbpm.domain</persistence-unit>
 <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
 <audit-mode>JPA</audit-mode>
 <persistence-mode>JPA</persistence-mode>
 <runtime-strategy>SINGLETON</runtime-strategy>
 <marshalling-strategies>
  <marshalling-strategy>
   <resolver>reflection</resolver>
   <identifier>
```

```
      org.jbpm.integration.cmis.impl.OpenCMISPlaceholderResolverStrategy
    </identifier>
  </marshalling-strategy>
</marshalling-strategies>
```

9. 要启用存储在自定义位置的文档要附加到表单和进程，在相关流程中创建文档变量，并将任务输入和输出映射到 Business Central 中的变量。

14.4.2. 在业务流程中创建文档变量

设置文档总结策略后，请在相关流程中创建一个文档变量，以便将文档上传到人工任务，并为 Business Central 中的 流程实例 视图显示文档或文档。

先决条件

- 您已设置了文档总结策略，如 第 14.4.1 节 “设置文档摘要策略” 所述。

流程

1. 在 Business Central 中，转至 Menu → Design → Projects。
2. 点击项目名称以打开资产视图，并点击业务流程名称。
3. 点 canvas 并点窗口右侧的
   
   来打开 Properties 面板。
4. 展开 Process Data 并点
   
   并输入以下值：
   - 名称 ：文档
   - custom Type:org.jbpm.document.Document for a single document or

org.jbpm.document.DocumentCollection for multiple Documentation

14.4.3. 将任务输入和输出到文档变量

如果要在任务表单中查看或修改附件，请在任务输入和输出中创建分配。

先决条件

- *您有一个包含至少包含一个用户任务的业务流程资产的项目。*

流程

1. *在 Business Central 中，转至 Menu → Design → Projects。*
2. *点击项目名称以打开资产视图，并点击业务流程名称。*
3. *点击用户任务并点击窗口右侧的*

打开 Properties 面板。
4. *展开 Implementation/Execution，并在 分配 旁边点*

打开 Data I/O 窗口。
5. *在 Data Inputs and Assignments 下，点 Add 并输入以下值：*
   - *名称 ：taskdoc_in*
   - *Data Type:org.jbpm.document.Document for a single document or org.jbpm.document.DocumentCollection for multiple Documentation*
   - *源 ：文档*

6. *在 Data Outputs and Assignments 下，点 Add 并输入以下值：*

   - *Name:taskdoc_out*
   - *Data Type:org.jbpm.document.Document for a single document or org.jbpm.document.DocumentCollection for multiple Documentation*
   - *目标 ：文档*

   Source 和 Target 字段包含您之前创建的进程变量的名称。

7. *点击 Save。*

第 15 章 高级进程概念和任务

15.1. 在业务流程中调用 DECISION MODEL 和 NOTATION(DMN)服务

您可以使用 Decision Model 和 Notation(DMN)在 Business Central 中的决策要求图(DRD)中以图形化方式为决策服务建模，然后在 Business Central 中将 DMN 服务作为业务流程的一部分调用。业务流程通过识别 DMN 服务并在 DMN 输入和业务流程属性之间映射业务数据，与 DMN 服务交互。

如图所示，此流程使用了示例 TrainStation 项目，它定义了列车路由逻辑。这个示例项目包含以下数据对象和 DMN 组件，用于在用于路由决策逻辑的 Business Central 中设计：

Train 对象示例

```
public class Train {

    private String departureStation;

    private String destinationStation;

    private BigDecimal railNumber;

    // Getters and setters
}
```

图 15.1. Compute Rail DMN 模型示例

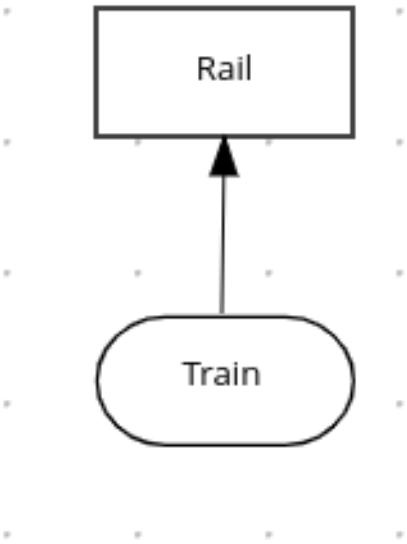
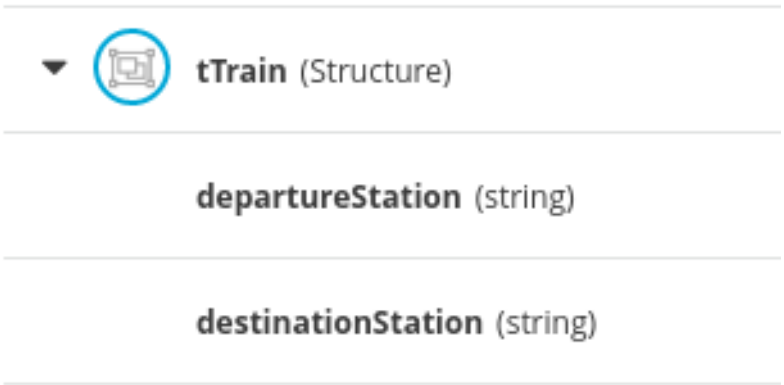


图 15.2. Rail DMN 决策表示例

Rail *(Decision Table)*

F	Train.departureStation *(string)*	Train.destinationStation *(string)*	Rail *(number)*	Description
1	"Prague"	"Hamburg"	5	
2	"Prague"	"Krakow"	2	
3	-	"Belgrade"	1	Just one option to Belgrade
4	"Zagreb"	-	3	Just one option from Zagreb
5	"Graz"	"Vienna"	1	

图 15.3. tTrain DMN 数据类型示例



有关在 Business Central 中创建 DMN 模型的更多信息，请参阅使用 DMN 模型设计决策服务。

先决条件

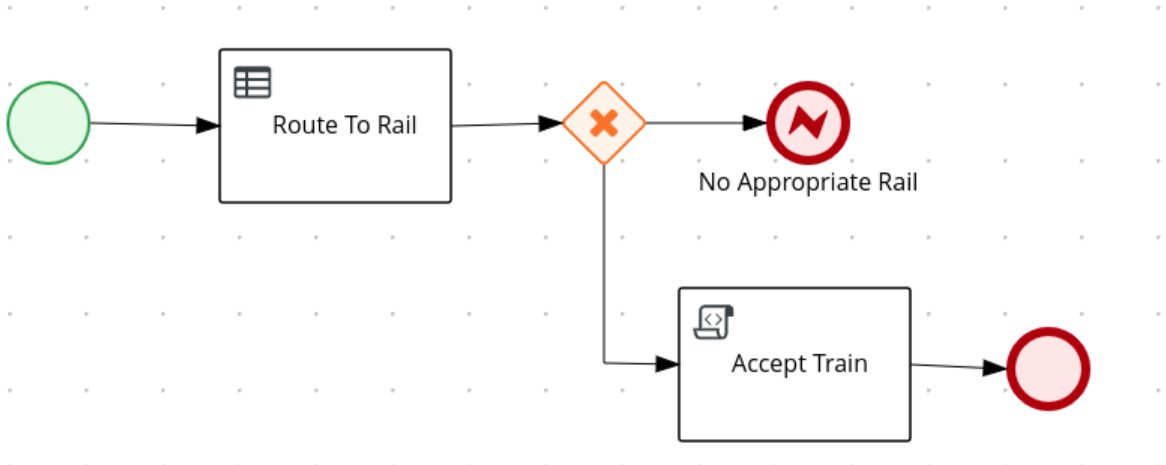
- 所有所需的数据对象和 DMN 模型组件都在项目中定义。

流程

1. 在 Business Central 中，前往 Menu → Design → Projects 并点项目名称。
2. 选择或创建您要在其中调用 DMN 服务的业务流程资产。
3. 在流程设计器中，使用左边工具栏来照常拖放 BPMN 组件，以定义您的整个业务流程逻辑、连接、事件、任务或其他元素。
4. 要在业务流程中整合 DMN 服务，请在左侧工具栏或启动节点选项中添加 业务规则 任务，并在流程流程中的相关位置插入任务。

   在本例中，以下 Accept Train 业务流程将 DMN 服务包含在 Route To Rail 节点中：

   图 15.4. 带有 DMN 服务的 Accept Train 业务流程示例



5. 选择您要用于 DMN 服务的业务规则任务节点，单击进程设计器右上角的 Properties，然后在 Implementation/Execution 下定义以下字段：

- *规则语言：选择 DMN。*
- *命名空间 ：从 DMN 模型文件中输入唯一的命名空间。示例 ： https://www.drools.org/kie-dmn*
- *决策名称 ：输入要在所选进程节点中调用的 DMN 决策节点的名称。示例：R ail*
- *DMN 模型名称 ：输入 DMN 模型名称。示例： 计算 Rail*

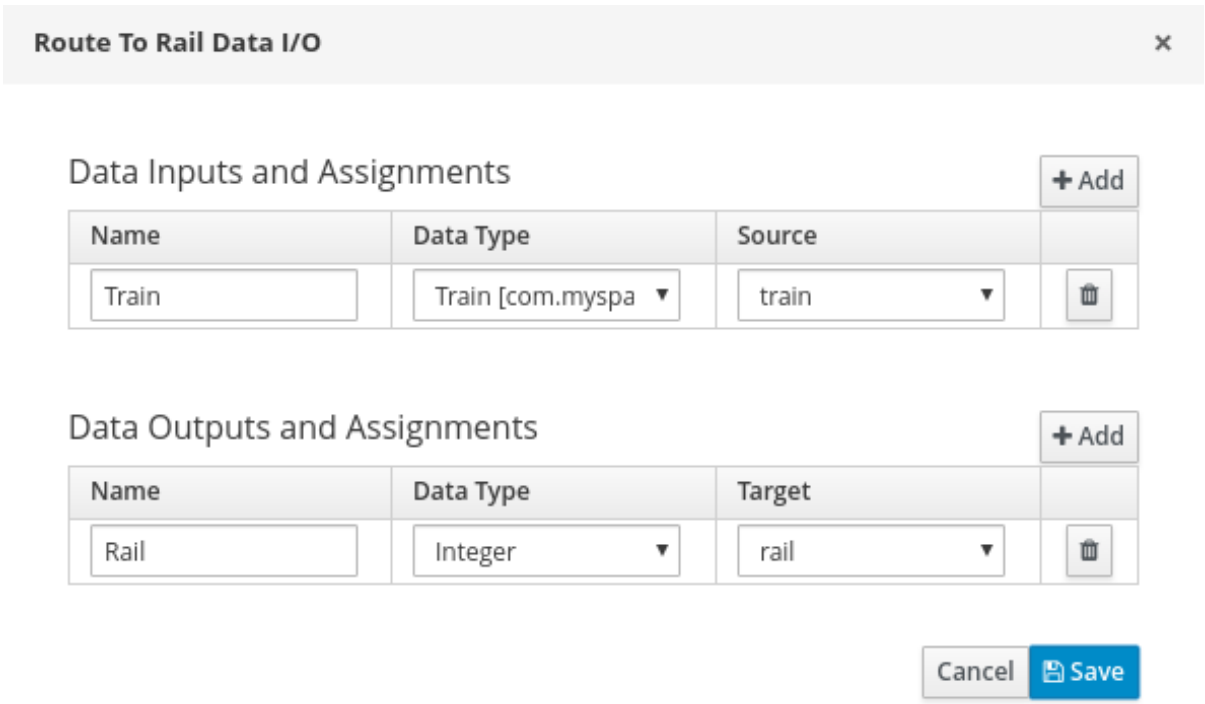
重要

当您浏览 root 节点时，请确保 Namespace 和 DMN Model Name 字段包含与 DMN 图表相同的值。

6. *在 Data Assignments → Assignments 下，点 Edit 图标并添加 DMN 输入和输出数据，以定义 DMN 服务与进程数据之间的映射。*

对于本例中的 Route To Rail DMN 服务节点，您可以为 Train 添加与 DMN 模型中输入节点对应的输入分配，并为 Rail 添加与 DMN 模型中决策节点对应的输出分配。Data Type 必须与您在 DMN 模型中为该节点设置的类型匹配，Source 和 Target 定义是指定对象的相关变量或字段。

图 15.5. Route To Rail DMN 服务节点的输入和输出映射示例



7. *点 Save 保存数据输入和输出数据。*

8. *根据您希望如何处理完成的 DMN 服务，定义您的业务流程的其余部分。*

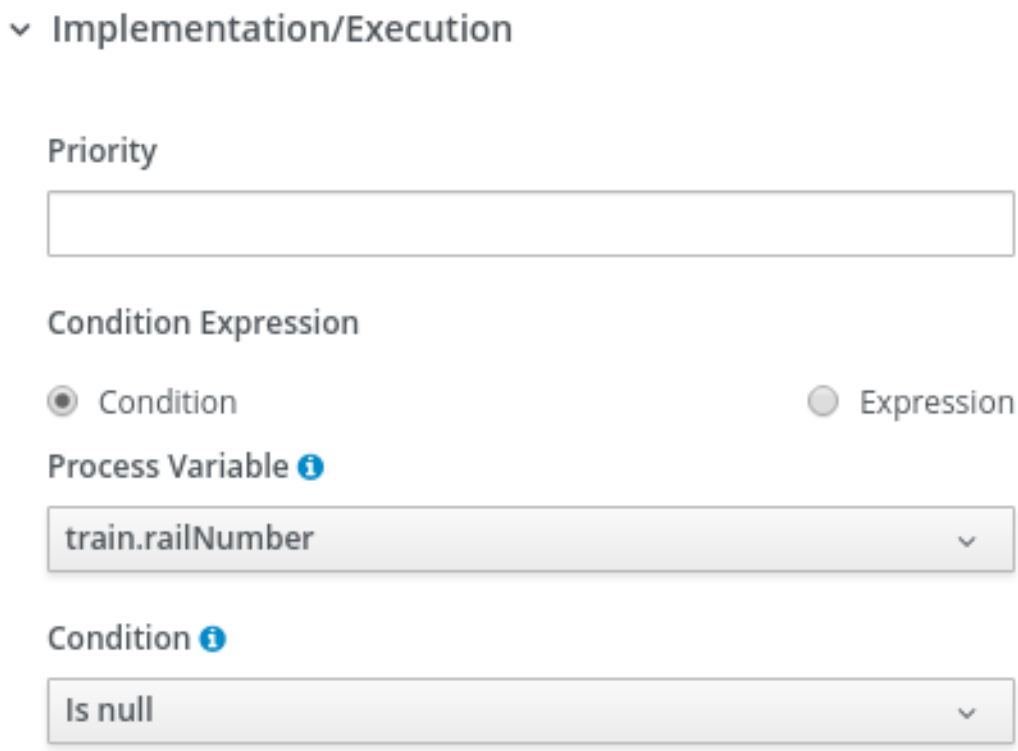
在本例中，Properties → Implementation/Execution → On Exit Action 值被设置为以下代码，以在 Route To Rail DMN 服务完成后存储 rail 号：

On Exit Action的代码示例

```
train.setRailNumber(rail);
```

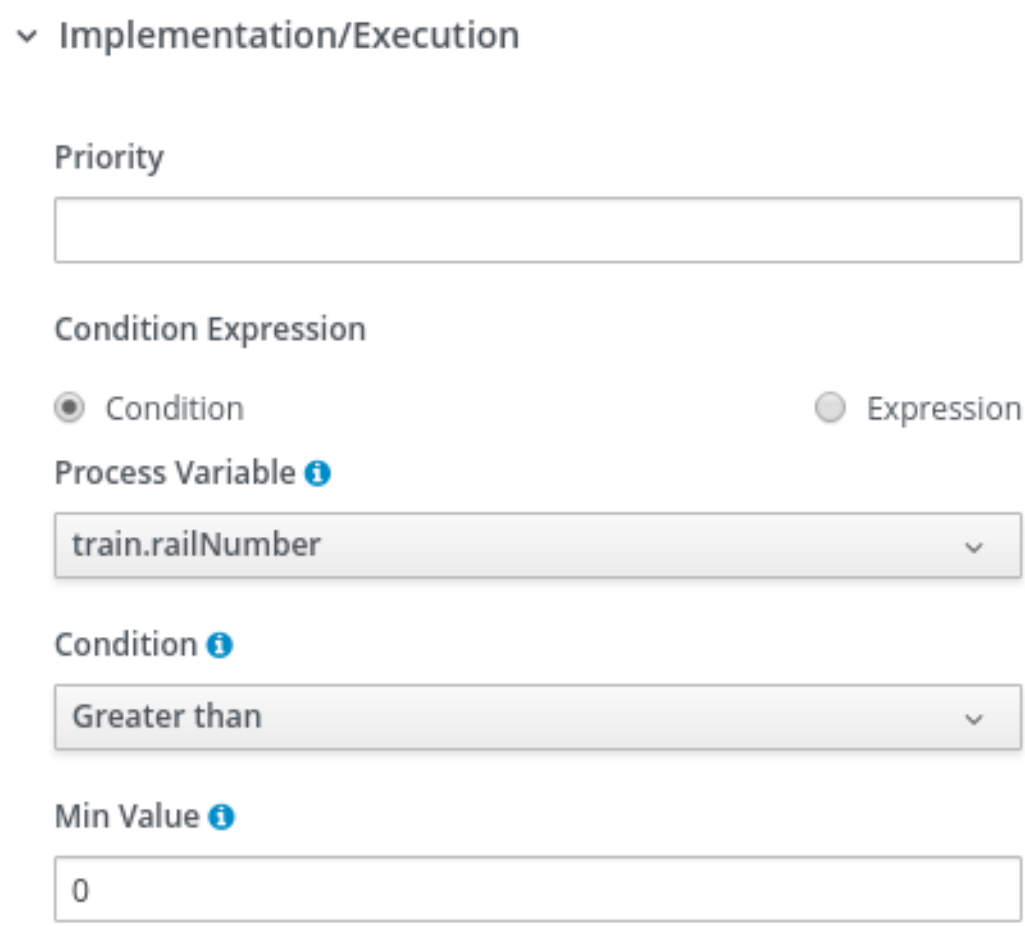
如果无法计算 rail 编号，则进程会达到一个没有正确的 Rail 错误节点，该节点定义了以下条件表达式：

图 15.6. 无适当的 Rail 最终错误节点状况示例



如果计算了 rail 编号，则进程到达 Accept Train 脚本任务，该任务使用以下条件表达式定义：

图 15.7. Accept Train 脚本任务节点的条件示例



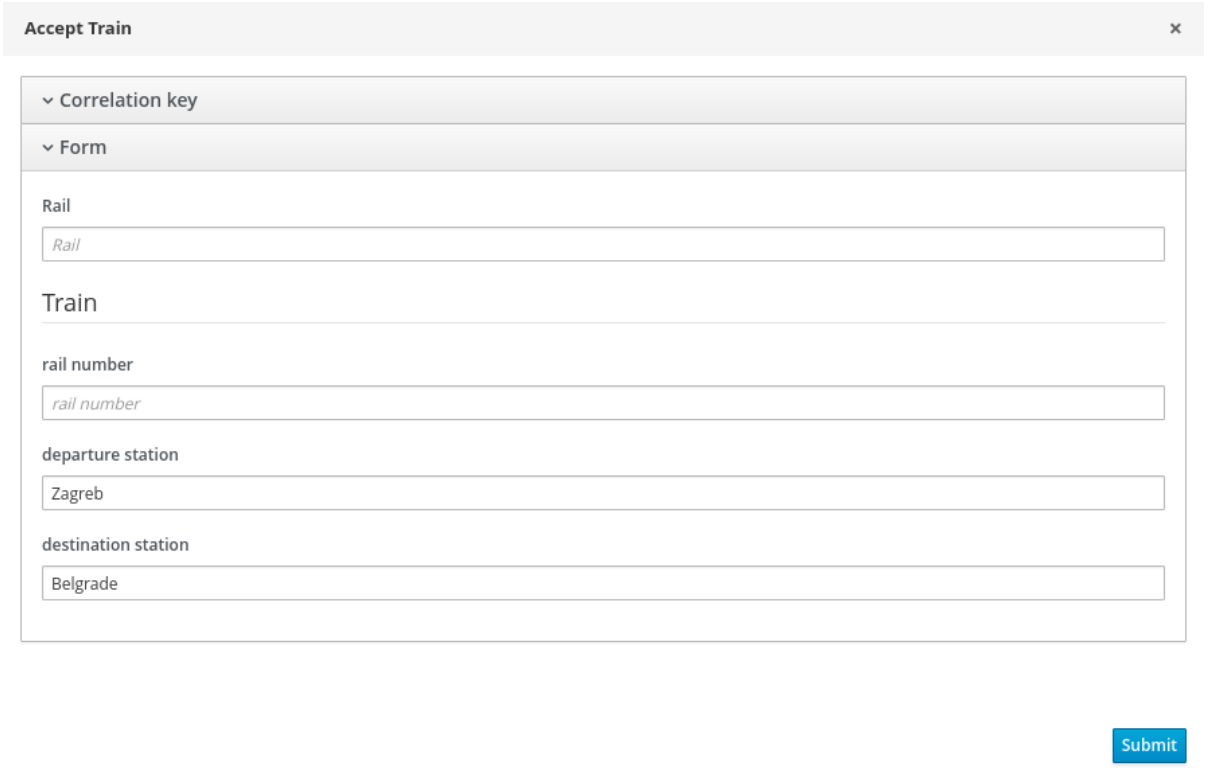
Accept Train 脚本任务还使用 Properties → Implementation/Execution → Script 中的以下脚本打印有关列路由和当前 rail 的消息：

```
com.myspace.trainstation.Train t =
  (com.myspace.trainstation.Train) kcontext.getVariable("train");
System.out.println("Train from: " + t.getDepartureStation() +
                   ", to: " + t.getDestinationStation() +
                   ",  is on rail: " + t.getRailNumber());
```

9. *使用融合的 DMN 服务定义业务流程后，将您的流程保存到流程设计器中，部署项目，然后运行对应的进程定义来调用 DMN 服务。*

在本例中，当您部署 TrainStation 项目并运行对应的进程定义时，打开 Accept Train 进程定义的进程实例表单，并设置 departure station 和 destination station 字段来测试执行：

图 15.8. Accept Train 进程定义的进程实例表单示例



执行完进程后，会在服务器日志中显示含有您指定的列车路由的消息：

Accept Train 进程的服务器日志输出示例

```
Train from: Zagreb, to: Belgrade,  is on rail: 1
```

第 16 章 其他资源

- *进程服务的入门*
- *在 Business Central 中管理和监控业务流程*
- *与进程和任务交互*

部分 II. 与进程和任务交互

作为知识工作，您可以在 Red Hat Process Automation Manager 中使用 Business Central 来运行由市民开发人员开发的业务流程应用程序的流程和任务。业务流程是按照流程流程中所定义执行的一系列步骤。要有效地与流程和任务交互，您必须明确了解业务流程并能够确定流程或任务的当前步骤。您可以启动和停止任务；搜索和过滤任务及处理实例；委派、声明和发布任务；设置任务的到期日期和时间；查看和添加任务历史记录日志；以及查看任务历史记录日志。

先决条件

- 安装了 Red Hat Process Automation Manager。有关安装选项，请参阅 *规划 Red Hat Process Automation Manager 安装*。

第 17 章 BUSINESS CENTRAL 中的业务流程

由 Business Central 中的公民开发人员创建的业务流程应用程序将业务流程作为一系列步骤描述。每个步骤均根据进程流图表执行。进程可由一个或多个较小的离散任务组成。作为知识工作，您将处理业务流程执行期间发生的流程和任务。

例如，使用 Red Hat Process Automation Manager 时，金融机构的抵押部门可以为抵押贷款自动化整个业务流程。当有新的mortgage 请求时，在 mortgage 应用中创建一个新进程实例。因为所有请求都遵循相同的一组规则来处理，因此请确保每个步骤中的一致性。这将产生一个可缩短处理时间和精力的有效流程。

17.1. 知识 WORKER 用户

考虑客户代表处理金融业款请求的客户代表性示例。作为客户账户代表，您可以执行以下任务：

- 接受和拒绝拒绝请求
- 搜索和过滤请求
- 委派、声明和发布请求
- 在请求上设置到期日期和优先级
- 查看和评论请求
- 查看请求历史记录日志

第 18 章 了解 BUSINESS CENTRAL 中的 WORKER 任务

任务是指定用户可以声明和执行的业务流程流的一部分。您可以在 Business Central 中处理 Menu → Track → Task Inbox 中的任务。它显示已登录用户的任务列表。一个任务可以分配给特定用户、多个用户或一组用户。如果某个任务分配给多个用户或一组用户，它会在所有用户的任务列表中看到，任何用户可以声明任务。当用户声明某个任务时，它将从其他用户的任务列表中删除。

18.1. 启动任务

您可以在 Menu → Manage → Tasks 和 Menu → Track → Task Inbox 中启动用户任务。

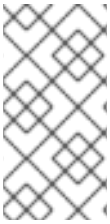
注意

确保您已登录，并且具有启动和停止任务的适当权限。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面中，点任务打开它。
3. 在任务页面的 Work 选项卡上，单击 Start。启动任务后，其状态会更改为 InProgress。

   您可以在 Task Inbox 以及 Manage Tasks 页面上查看任务的状态。



注意

只有具有 process-admin 角色的用户才能在 Manage Tasks 页面上查看任务列表。具有 admin 角色的用户可以访问 Manage Tasks 页面，但它们只看到一个空任务列表。

18.2. 停止任务

您可以从 Tasks 和 Task Inbox 页面中停止用户任务。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。

2. 在 Task Inbox 页面中，点任务打开它。

3. 在任务页面的 Work 选项卡上，单击 Complete。

18.3. 委派任务

在 Business Central 中创建任务后，您可以将它们委托给他人。

注意

使用任何角色分配的用户可以委托、声明或释放用户可见的任务。在 Task Inbox 页面中，Actual Owner 列显示任务当前所有者的名称。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。

2. 在 Task Inbox 页面中，点任务打开它。

3. 在任务页面上，单击 Assignments 选项卡。

4. 在 User 字段中，输入您要委派该任务的用户或组的名称。

5. 点 Delegate。委派任务后，任务的所有者会改变。

18.4. 声明任务

在 Business Central 中创建任务后，您可以声明发布的任务。只有在任务被分配给用户所属的组时，用户才能从 Task Inbox 页面中声明任务。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面中，点任务打开它。
3. 在任务页面的 Work 选项卡上，单击 Claim。
4. 要从 Task Inbox 页面中声明发布的任务，请执行以下任一任务：
   - 在 Actions 列中的三个点中，单击 Claim。
   - 点 Actions 列中的三个点中的 Claim 和 Works 打开、查看和修改任务详情。

声明任务的用户将成为任务的所有者。

18.5. 发布任务

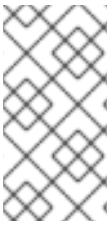
在 Business Central 中创建任务后，您可以发布您的任务供他人声明。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面中，点任务打开它。
3. 在任务页面中，单击 Release。已发布的任务没有所有者。

18.6. 对任务进行批量操作

在 Business Central 中的 Tasks 和 Task Inbox 页面中，您可以通过单个操作对多个任务执行批量操作。



注意

如果不允许基于任务状态指定批量操作，则会显示通知，操作不会在该特定任务上执行。

18.6.1. 批量声明任务

在 Business Central 中创建任务后，您可以广泛声明可用的任务。

流程

1. 在 Business Central 中，完成以下步骤之一：
   - 要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。
   - 要查看 Tasks 页面，请选择 Menu → Manage → Tasks。
2. 要批量声明任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。
3. 在 Bulk Actions 下拉列表中选择 Bulk Claim。
4. 要确认，请单击 Claim selected tasks 窗口。

对于选择每个任务，会显示通知来显示结果。

18.6.2. 批量释放任务

您可以批量释放您拥有的任务供他人使用。

流程

1. 在 Business Central 中，完成以下步骤之一：

   - 要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。
   - 要查看 Tasks 页面，请选择 Menu → Manage → Tasks。

2. 要批量释放任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。

3. 在 Bulk Actions 下拉列表中选择 Bulk Release。

4. 要确认，请单击 Release selected tasks 窗口。

对于选择每个任务，会显示通知来显示结果。

18.6.3. 恢复批量的任务

如果 Business Central 中暂停的任务，您可以批量恢复它们。

流程

1. 在 Business Central 中，完成以下步骤之一：

   - 要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。
   - 要查看 Tasks 页面，请选择 Menu → Manage → Tasks。

2. 要恢复批量的任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。

3. 从 Bulk Actions 下拉列表中，选择 Bulk Resume。

4. 要确认，请点击 Resume selected tasks 窗口。

对于选择每个任务，会显示通知来显示结果。

18.6.4. 批量挂起任务

在 Business Central 中创建任务后，您可以暂停批量的任务。

流程

1. 在 Business Central 中，完成以下步骤之一：

   - 要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。
   - 要查看 Tasks 页面，请选择 Menu → Manage → Tasks。

2. 要批量暂停任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或多个任务。

3. 在 Bulk Actions 下拉列表中选择 Bulk Suspend。

4. 要确认，请点击 Suspend 选择的任务 窗口。

对于选择每个任务，会显示通知来显示结果。

18.6.5. 批量重新分配任务

在 Business Central 中创建任务后，您可以重新分配您的任务并将其委托给其他任务。

流程

1. *在 Business Central 中，完成以下步骤之一：*
   - *要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。*
   - *要查看 Tasks 页面，请选择 Menu → Manage → Tasks。*
2. *要批量重新分配任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。*
3. *在 Bulk Actions 下拉列表中，选择 Bulk Reassign。*
4. *在 任务 重新分配窗口中，输入您要重新分配任务的用户 ID。*
5. *点 Delegate。*

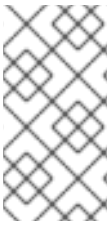
对于选择每个任务，会显示通知来显示结果。

第 19 章 BUSINESS CENTRAL 中的任务过滤

Business Central 提供内置过滤器，可帮助您搜索任务。您可以根据属性（如 Status、Filter by、流程定义 Id 和 Created On ）过滤任务。也可以使用 Advanced Filters 选项创建自定义任务过滤器。新创建的自定义过滤器添加到 Saved Filters 窗格中，请点击 Task Inbox 页面左侧的星号图标来访问该过滤器。

19.1. 管理任务列表列

在 Task Inbox 和 Manage Tasks 窗口中的任务列表中，您可以指定要查看哪些列，并修改列的顺序来更好地管理任务信息。



注意

只有具有 process-admin 角色的用户才能在 Manage Tasks 页面上查看任务列表。具有 admin 角色的用户可以访问 Manage Tasks 页面，但它们只看到一个空任务列表。

流程

1. 在 Business Central 中，前往 Menu → Manage → Tasks 或 Menu → Track → Task Inbox。
2. 在 Manage Task or Task Inbox 页面上，单击 Bulk Actions 右侧的 Show/hide 列 图标。
3. 选择或取消选择要显示的列。当您更改列表时，任务列表中的列会显示或消失。
4. 要重新排列列，将列标题拖到新位置。请注意，在拖动列前，您的指针必须改为以下图示中显示的图标：

5. 要将更改保存为过滤器，请点击 Save Filters，输入名称，然后点 Save。
6. 要使用您的新过滤器，请点击屏幕左侧的 Saved Filters 图标(star)，并从列表中选择您的过滤器。

19.2. 使用基本过滤器过滤任务

Business Central 根据任务（如 状态、Filter by、流程定义 Id 和 Created On ）提供通过任务过滤和搜索的基本过滤器。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面中，点击页面左侧的过滤器图标展开 Filters 窗格，然后选择要使用的过滤器：
   - 状态 ：根据其状态过滤任务。
   - 过滤条件 ：根据 Id、任务、关联密钥、操作所有者或 进程实例描述 属性过滤任务。
   - 进程定义 Id ：根据进程定义 ID 过滤任务。
   - 创建于: 根据其创建日期过滤任务。

您可以使用 Advanced Filters 选项在 Business Central 中创建自定义过滤器。

19.3. 使用高级过滤器过滤任务

您可以使用 Business Central 中的 Advanced Filters 选项创建自定义任务过滤器。

流程

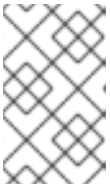
1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面上，点击页面左侧的高级过滤器图标展开 Advanced Filters 面板。

3. 在 Advanced Filters 面板中，输入过滤器名称和描述，然后单击 Add New。

4. 从 Select 列 下拉列表中选择属性，如 Name。下拉列表对 名称 != value1 的内容。

5. 再次点击下拉菜单并选择所需的逻辑查询。对于 Name 属性，选择 等于。

6. 将文本字段的值更改为要过滤的任务的名称。



注意

名称必须与项目业务流程中定义的值匹配。

7. 点击 Save，并按照过滤器定义过滤任务。

8. 单击星号图标，以打开 Saved Filters 窗格。

在 Saved Filters 窗格中，您可以查看保存的高级过滤器。

19.4. 使用 DEFAULT 过滤器管理任务

您可以使用 Business Central 中的 Saved Filter 选项将任务过滤器设置为默认过滤器。每次用户打开页面时，都将执行默认过滤器。

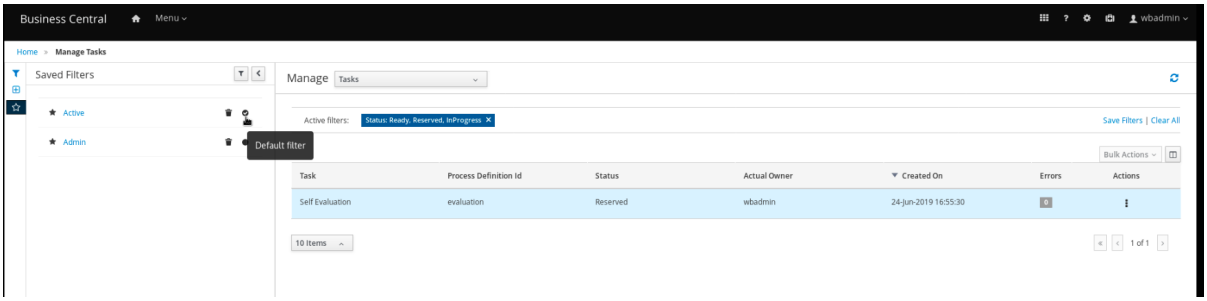
流程

1. 在 Business Central 中，前往 Menu → Track → Task Inbox，或前往 Menu → Manage → Tasks。

2. 在 Task Inbox 页面或 Manage Tasks 页面上，单击页面左侧的星号图标，以展开 Saved Filters 面板。

在 Saved Filters 面板中，您可以查看保存的高级过滤器。

任务或任务的默认过滤器选择



3. 在 Saved Filters 面板中，将保存的任务过滤器设置为默认过滤器。

19.5. 使用基本过滤器查看任务变量

Business Central 提供了基本的过滤器，来查看 管理任务 和任务中的任务变量。您可以使用 Show/hide 列将任务任务变量以列的形式查看。

流程

1. 在 Business Central 中，前往 Menu → Manage → Tasks，或前往 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面上，点击页面左侧的过滤器图标展开 Filters 面板
3. 在 Filters 面板中，选择 Task Name。

   过滤器应用到当前任务列表。
4. 点击任务列表右上角的 Show/hide 列，将显示指定任务 id 的任务变量。
5. 单击星号图标，以打开 Saved Filters 面板。

   在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

19.6. 使用高级过滤器查看任务变量

您可以使用 Business Central 中的 Advanced Filters 选项查看 管理任务和任务收 件中的任务变量。当您创建过滤器时，您可以使用 Show/ hide 列，以列的形式查看任务的任务变量。

流程

1. 在 Business Central 中，前往 Menu → Manage → Tasks，或前往 Menu → Track → Task Inbox。
2. 在 Manage Tasks 页面或 Task Inbox 页面中，点击高级过滤器图标展开 Advanced Filters 面板。
3. 在 Advanced Filters 面板中，输入过滤器的名称和描述，然后单击 Add New。
4. 从 Select 列 列表中，选择 name 属性。该值将更改为 名称 != value1。
5. 在 Select 列 列表中，为逻辑查询选择 等于。
6. 在文本字段中，输入任务的名称。
7. 点 Save，过滤器应用到当前任务列表。
8. 点击任务列表右上角的 Show/hide 列，将显示指定任务 id 的任务变量。
9. 单击星号图标，以打开 Saved Filters 面板。

   在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

第 20 章 在 BUSINESS CENTRAL 中处理实例过滤

Business Central 现在为您提供基本和高级过滤器，以帮助您在整个流程实例过滤和搜索。您可以根据 状态、错误、过滤器、名称、开始日期 和 Last update 等属性过滤 进程。您还可以使用 Advanced Filters 选项创建自定义过滤器。新创建的自定义过滤器添加到 Saved Filters 窗格中，可通过单击 Manage Process Instances 页面左侧的星号图标来访问。

注意

除 经理 或 其余角色以外 的所有用户均可访问和过滤 Business Central 中的进程实例。

20.1. 使用基本过滤器过滤进程实例

Business Central 可根据其属性（如 状态、错误、Filter、名称、开始日期 和 最后更新 ）通过流程实例过滤和搜索提供基本过滤器。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面中，点页面左侧的过滤器图标展开 Filters 窗格，再选择您要使用的过滤器：
   - State ：根据其状态过滤进程实例（活动、Aborted、Completed、Pending 和 Suspended）。
   - 错误 ：过滤至少包含一个或多个错误的进程实例。
   - 过滤符 ：根据 Id、Initiator、关联密钥或 Description 属性过滤进程实例。
   - 名称 ：根据进程定义名称过滤进程实例。
   - 定义 ID ：实例定义的 ID。

- 部署 ID ：实例部署的 ID。
- SLA 合规性：SLA 合规状态（如 满足、Met e、N/A、Pending 和 Violated）。
- 父进程 ID ：父进程实例的 ID。
- 开始日期 ：根据创建日期过滤进程实例。
- 最后更新 ：根据上次修改的日期过滤进程实例。

您还可以使用 Advanced Filters 选项在 Business Central 中创建自定义过滤器。

20.2. 使用高级过滤器过滤进程实例

您可以使用 Business Central 中的 Advanced Filters 选项创建自定义进程实例过滤器。

流程

1. 在 Business Central 中，点击 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面上，单击 Advanced Filters 图标。
3. 在 Advanced Filters 窗格中，输入过滤器的名称和描述，然后单击 Add New。
4. 从 Select 列 下拉列表中选择一个属性，如 processName。processName != value1 下拉列表更改的内容。
5. 再次点击下拉菜单并选择所需的逻辑查询。对于 processName 属性，选择 等于。
6. 将文本字段的值更改为要过滤的进程的名称。

注意

processName 必须与项目业务流程中定义的值匹配。

7. **点 Save，并根据过滤器定义过滤进程。**

8. **单击星号图标，以打开 Saved Filters 窗格。**

   在 Saved Filters 窗格中，您可以查看所有保存的高级过滤器。

20.3. 使用 DEFAULT 过滤器管理进程实例

您可以使用 Business Central 中的 Saved Filter 选项将进程实例过滤器设置为默认过滤器。每次用户打开页面时，都将执行默认过滤器。

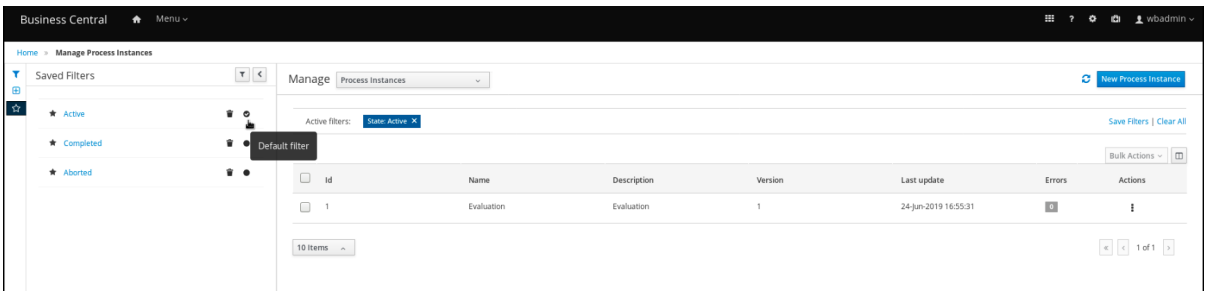
流程

1. **在 Business Central 中，前往 Menu → Manage → Process Instances。**

2. **在 Manage Process Instances 页面上，单击页面左侧的星号图标，展开 Saved Filters 面板。**

   在 Saved Filters 面板中，您可以查看保存的高级过滤器。

进程实例的默认过滤器选择



3. 在 Saved Filters 面板中，将保存的进程实例过滤器设置为默认过滤器。

20.4. 使用基本过滤器查看进程实例变量

Business Central 提供基本过滤器来查看流程实例变量。您可以使用 Show/ hide 列将进程实例变量以列的形式查看。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面上，单击页面左侧的过滤器图标展开 Filters 面板。
3. 在 Filters 面板中，选择 Definition Id。

   过滤器应用于当前进程实例列表。
4. 单击进程实例列表右上角的 Show/hide 列，将显示指定进程 ID 的进程实例变量。
5. 单击星号图标，以打开 Saved Filters 面板。

   在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

20.5. 使用高级过滤器查看进程实例变量

您可以使用 Business Central 中的 Advanced Filters 选项来查看进程实例变量。当您在列 processId 上创建过滤器时，您可以使用 Show/hide 列将进程实例变量查看为列。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面上，单击高级过滤器图标以展开 Advanced Filters 面板。

3. 在 Advanced Filters 面板中，输入过滤器的名称和描述，然后单击 Add New。

4. 从 Select 列 列表中，选择 processId 属性。该值将更改为 processId != value1。

5. 在 Select 列 列表中，为逻辑查询选择 等于。

6. 在文本字段中，输入进程 ID 的名称。

7. 点 Save，过滤器应用到当前进程实例列表中。

8. 单击进程实例列表右上角的 Show/hide 列，将显示指定进程 ID 的进程实例变量。

9. 单击星号图标，以打开 Saved Filters 面板。

   在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

第 21 章 在任务通知中配置电子邮件

之前，只能向 Business Central 中的用户或组发送通知。现在，您也可以直接添加任何电子邮件地址。

先决条件

您已在 Business Central 中创建了一个项目。

流程

1. 创建业务流程。

   有关在 Business Central 中创建业务流程的详情请参考 第 5 章 在 Business Central 中创建业务流程。

2. 创建用户任务。

   有关在 Business Central 中创建用户任务的详情，请参考 第 5.4 节 “创建用户任务”。

3. 在屏幕右上角，单击 Properties 图标。

4. 展开 Implementation/Execution，然后点击 Notifications 旁边的  以打开 Notifications 窗口。

5. 点击 Add。

6. 在 Notifications 窗口中，在 To: email(s) 字段中输入电子邮件地址，以设置任务通知电子邮件的接收者。

   您可以添加用逗号分开的多个电子邮件地址。

7. 输入电子邮件的主题和正文。

8.
点 确定。

您可以在" 通知 "窗口中的 To: email(s) 列中看到添加的电子邮件地址。

9.
点 确定。

第 22 章 设置任务的到期日期和优先级

您可以在 Task Inbox 页面中在 Business Central 中设置优先级、到期日期和时间。请注意，所有用户可能没有设置优先级的权限，以及任务的原因。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面中，点任务打开它。
3. 在任务页面中，点击 Details 选项卡。
4. 在 Due Date 字段中，从日历中选择所需日期以及下拉列表中的到期时间。
5. 在 Priority 字段中，选择所需优先级。
6. 点 Update。

第 23 章 查看并在任务中添加评论

您可以为任务添加注释，并查看 Business Central 中任务的现有注释。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面中，点任务打开它。
3. 在任务页面中，点 Work 选项卡或 Comments 选项卡。
4. 在 Comment 字段中输入任务相关的注释，然后点 Add Comment 图标。

所有任务相关注释都以 工作 和 注释 选项卡中的表格形式显示。

注意

要选中或清除 工作选项卡中的 Show 任务注释，请转至 Business Central 主页，单击 Settings 图标并选择 Process Administration 选项。只有具有 admin 角色的用户才具有启用或禁用此功能的权限。

第 24 章 查看任务的历史记录日志

您可以在任务的 Logs 选项卡中查看 Business Central 中某个任务的历史记录日志。历史记录日志以"Date Time: Task"格式列出所有事件。

流程

1. 在 Business Central 中，转至 Menu → Track → Task Inbox。
2. 在 Task Inbox 页面中，点任务打开它。
3. 在任务页面中，单击 Logs 选项卡。

   在任务生命周期中发生的所有事件都列在 Logs 选项卡中。

第 25 章 查看进程实例的历史记录日志

您可以从 Logs 选项卡查看 Business Central 中进程实例的历史记录日志。日志以 Date Time: Event Node Type: Event Type 格式列出所有事件。

您可以根据 Event Node Type 和 Event Type 来过滤日志。您还可以在日志中查看人节点的详情。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Process Instances 页面中，点您要查看的日志的进程实例。
3. 在实例页面上，单击 Logs 选项卡。
4. 从 Event Node Type 和 Event Type 窗格中选择需要的复选框，以根据需要过滤日志。
5. 要查看与人节点相关的其他信息，请展开 Details。
6. 点击 Reset 恢复到默认过滤器选择。

   在进程实例生命周期中发生的所有事件都列在 Logs 选项卡中。

部分 III. 在 BUSINESS CENTRAL 中管理和监控业务流程

作为系统管理员，您可以在 Red Hat Process Automation Manager 中使用 Business Central 来管理和监控在多个项目上运行的进程实例和任务。从 Business Central，您可以启动新的进程实例，验证所有进程实例的状态，以及中止进程。您可以查看与进程关联的作业和任务列表，以及理解和沟通任何进程错误。

先决条件

- Red Hat JBoss Enterprise Application Platform 7.4 已安装。如需更多信息，请参阅 Red Hat JBoss Enterprise Application Platform 7.4 安装指南。
- 安装了 Red Hat Process Automation Manager。如需更多信息，请参阅 规划 Red Hat Process Automation Manager 安装。
- Red Hat Process Automation Manager 正在运行，您可以使用 process-admin 角色登录到 Business Central。如需更多信息，请参阅 规划 Red Hat Process Automation Manager 安装。

第 26 章 进程监控

Red Hat Process Automation Manager 为您的业务流程提供实时监控，并包括以下功能：

- 业务经理可以实时监控流程。
- 客户可以监控其请求的当前状态。
- 管理员可以轻松监控与进程执行相关的任何错误。

第 27 章 BUSINESS CENTRAL 中的进程定义和流程实例

进程定义是业务流程模型和符号(BPMN)2.0 文件，充当进程及其 BPMN 图表的容器。进程定义显示有关业务流程的所有可用信息，如任何关联的子进程或参与所选定义的用户和组数量。

进程定义也定义了 导入 条目，供进程定义使用，以及进程定义 的关系 条目。

进程定义的 BPMN2 源

```
<definitions id="Definition"
             targetNamespace="http://www.jboss.org/drools"
             typeLanguage="http://www.java.com/javaTypes"
             expressionLanguage="http://www.mvel.org/2.0"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"Rule Task
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL BPMN20.xsd"
             xmlns:g="http://www.jboss.org/drools/flow/gpd"
             xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
             xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
             xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
             xmlns:tns="http://www.jboss.org/drools">

    <process>
      PROCESS
    </process>

    <bpmndi:BPMNDiagram>
     BPMN DIAGRAM DEFINITION
    </bpmndi:BPMNDiagram>

    </definitions>
```

创建、配置并部署包含您的业务流程的项目后，您可以查看 Business Central Menu → Manage → Process Definitions 中所有进程定义的列表。您可以通过单击右上角的刷新按钮，随时刷新已部署的进程定义列表。

进程定义列表显示部署到平台上的所有可用进程定义。单击列出的任何进程定义，以显示对应的进程定义详细信息。这将显示有关进程定义的信息，例如，如果存在关联了子进程，或者进程定义中存在多少个用户和组。进程定义详情页面中的 Diagram 选项卡包含基于 BPMN2 的进程定义图。

在每个选定进程定义中，您可以通过单击右上角的 New Process Instance 按钮来启动进程定义的新进程实例。从可用进程定义启动的实例列在 Menu → Manage → Process Instances 中。

您还可以在 Manage 下拉菜单下为所有用户定义默认的分页选项（流程定义、流程 实例、任务、作业和执行 错误）以及 Menu → Track → Task Inbox 中。

27.1. 从进程定义页面中启动进程实例

您可以在 Menu → Manage → Process Definitions 中启动进程实例。这可用于您同时处理多个项目或进程定义的环境。

先决条件

- 已在 Business Central 中部署了带有进程定义的项目。

流程

1. 在 Business Central 中，转至 Menu → Manage → Process Definitions。
2. 从列表中选择您要启动新进程实例的进程定义。该定义的详情页面将打开。
3. 单击右上角的 New Process Instance，以启动新的进程实例。
4. 为进程实例提供任何所需信息。
5. 单击 Submit 以创建进程实例。
6. 在 Menu → Manage → Process Instances 中查看新进程实例。

27.2. 从进程实例页面中启动进程实例

您可以创建新进程实例，或者查看 Menu → Manage → Process Instances 中所有运行的进程实例的列表。

先决条件

- 已在 Business Central 中部署了带有进程定义的项目。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 单击右上角的 New Process Instance，再从下拉列表中选择您要启动新进程实例的进程定义。
3. 提供启动新进程实例所需的任何信息。
4. 单击 Start 以创建进程实例。

   新进程实例会出现在 Manage Process Instances 列表中。

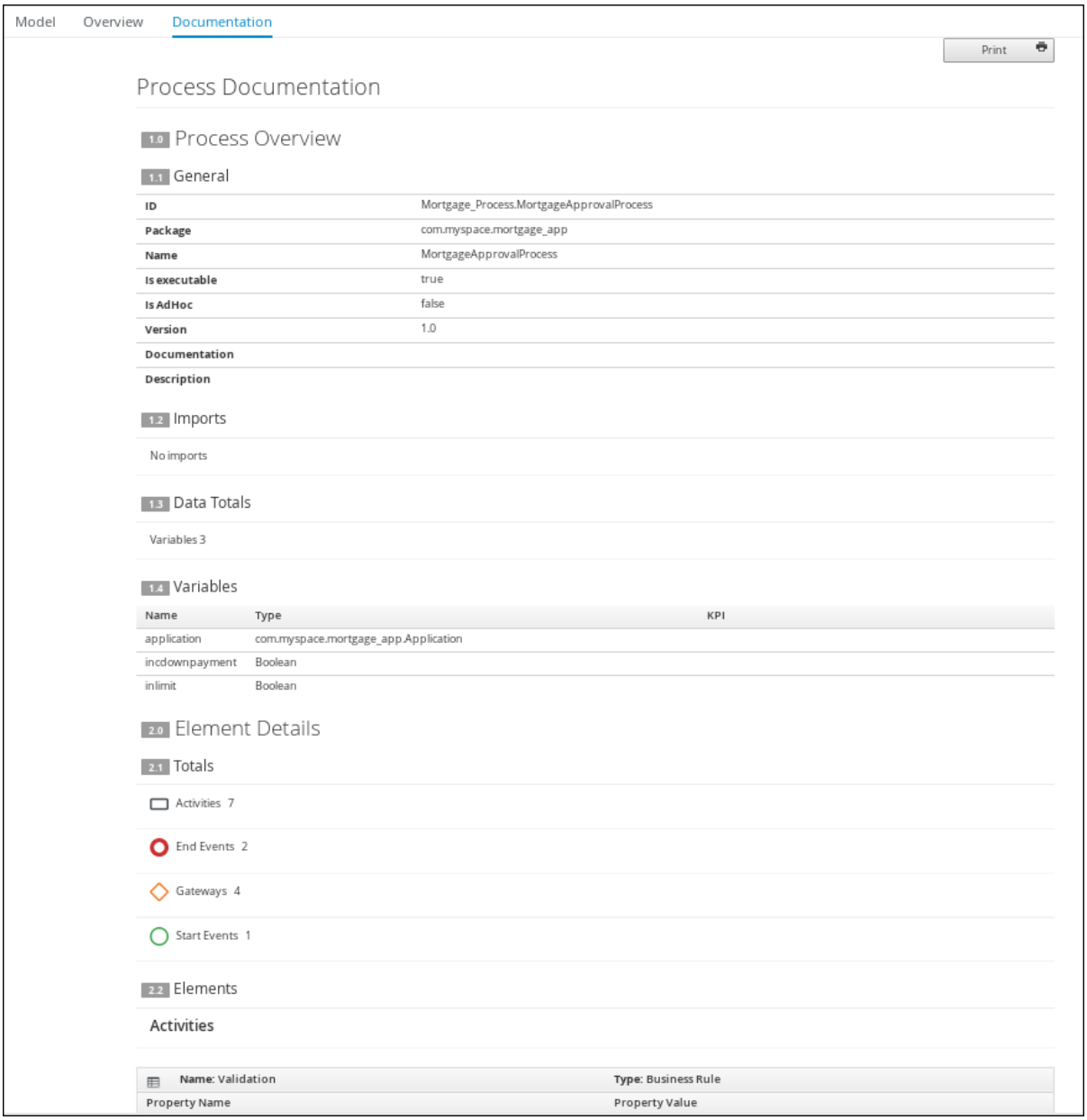
27.3. 在 BUSINESS CENTRAL 中生成流程文档

在 Business Central 中的进程设计器中，您可以查看并打印进程定义的报告。进程文档总结了您更轻松地打印和共享的格式(PDF)的组件、数据和可视化流程。

流程

1. 在 Business Central 中，进入包含业务流程的项目并选择流程。
2. 在进程设计器中，单击 Documentation 选项卡以查看进程文件摘要，然后单击窗口右上角的 Print 以打印 PDF 报告。

图 27.1. 生成进程文档

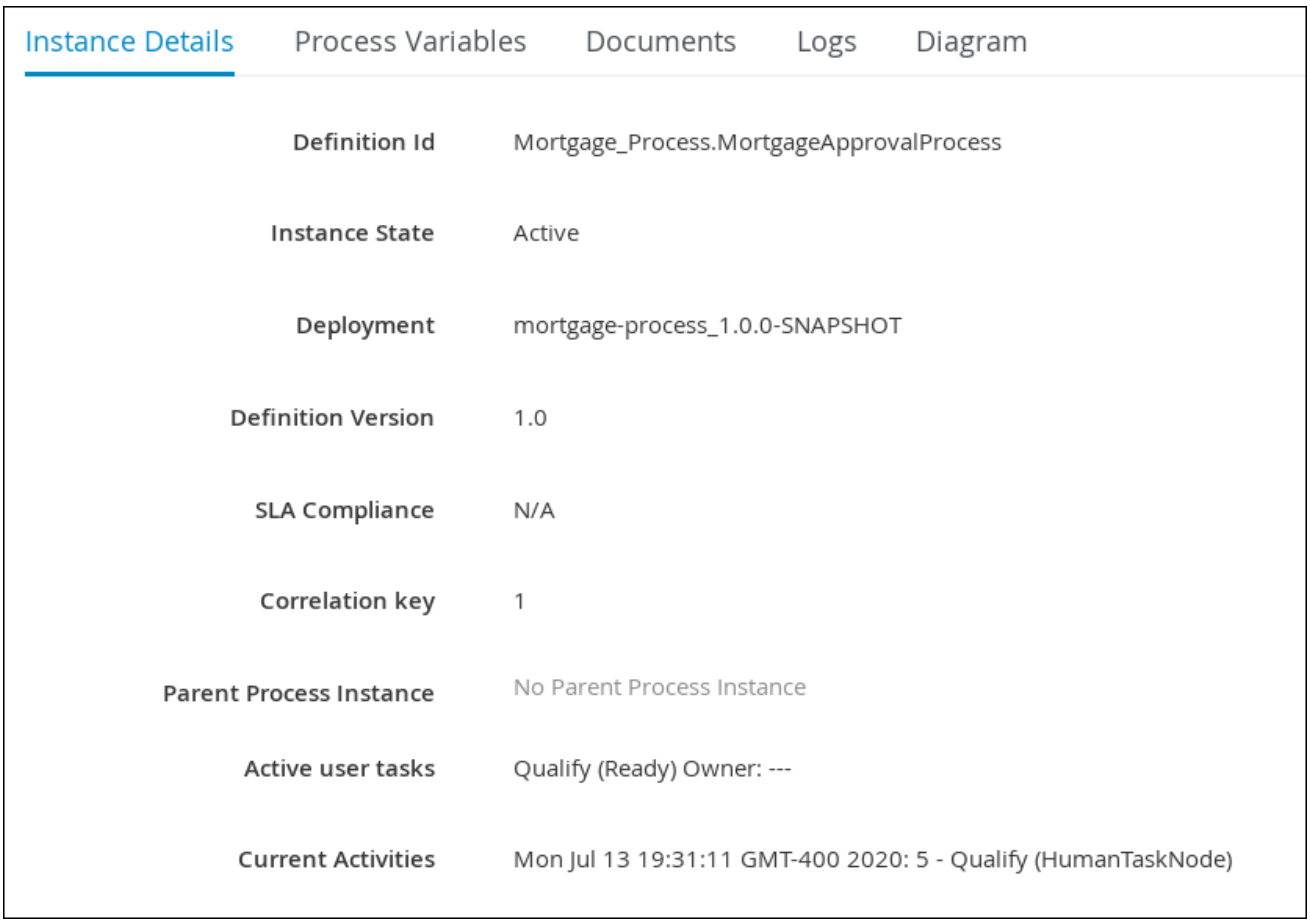

Model
Overview
Documentation
Print
Process Documentation
1.0 Process Overview
1.1 General
ID
Mortgage_Process.MortgageApprovalProcess
Package
com.myspace.mortgage_app
Name
MortgageApprovalProcess
Is executable
true
Is AdHoc
false
Version
1.0
Documentation
Description
1.2 Imports
No imports
1.3 Data Totals
Variables 3
1.4 Variables
Name
Type
KPI
application
com.myspace.mortgage_app.Application
incdownpayment
Boolean
inlimit
Boolean
2.0 Element Details
2.1 Totals
Activities 7
End Events 2
Gateways 4
Start Events 1
2.2 Elements
Activities
Name: Validation
Type: Business Rule
Property Name
Property Value

第 28 章 进程实例管理

要查看进程实例，请在 Business Central 中点击 Menu → Manage → Process Instances。

+ 注意：管理进程实例列表中 的每一行 代表特定进程定义的进程实例。每个实例都有自己的进程正在操作的信息的内部状态。点进程实例查看与进程相关的运行时信息对应的标签页。

图 28.1. 进程实例标签页视图



- 实例 详情：提供与过程中的内容相关的概述。它显示实例的当前状态以及正在执行的当前活动。

- process Variables ：显示由实例操作的所有进程变量，但含有文档的变量除外。您可以编辑 process 变量值并查看其历史记录。

- 文档 ：显示进程包含类型 org.jbpm.Document 的变量，则显示进程文档。这允许对附加文档的访问、 下载和操作。

- 日志 ：显示最终用户的进程实例日志。如需更多信息，请参阅 与进程和任务 交互。

- 图 ：通过 BPMN2 图表跟踪进程实例的进度。正在进行的进程流的节点或节点以红色突出显示。可重复使用的子进程会在父进程中出现折叠状态。双击可重复使用的子进程节点，从父进程图打开其图表。

有关访问 KIE 服务器运行时数据所需的用户凭证和条件的信息，请参阅 规划 Red Hat Process Automation Manager 安装。

28.1. 进程实例过滤

对于 Menu → Manage → Process Instances 中的进程实例，您可以使用 Filters 和 Advanced Filters 面板根据需要对进程实例进行排序。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。

2. 在 Manage Process Instances 页面中，点页面左侧的 Filters 图标，以选择您要使用的过滤器：

   - State ：根据其状态过滤进程实例（活动、Aborted、Completed、Pending 和 Suspended）。

   - 错误 ：过滤至少包含一个或多个错误的进程实例。

   - 按 过滤：根据以下属性过滤进程实例：

     - ID ：按进程实例 ID 过滤。

       输入：Num eric

     - initiator ：按进程实例发起方的用户 ID 过滤。

用户 ID 是唯一的值，它取决于 ID 管理系统。

input： 字符串

- correlation key ：按关联键过滤。

  input： 字符串

- 描述: 按进程实例描述过滤。

  input： 字符串

- 名称 ：根据进程定义名称过滤进程实例。
- 定义 ID ：实例定义的 ID。
- 部署 ID ：实例部署的 ID。
- SLA 合规性：SLA 合规状态（如 满足、Met e、N/A、Pending 和 Violated）。
- 父进程 ID ：父进程的 ID。
- 开始日期 ：根据创建日期过滤进程实例。
- 最后更新 ：根据上次修改的日期过滤进程实例。

您还可以使用 Advanced Filters 选项在 Business Central 中创建自定义过滤器。

28.2. 创建自定义进程实例列表

您可以在 Business Central 中查看 Menu → Manage → Process Instances 中所有正在运行的进程实例的列表。在这个页面中，您可以在运行时管理实例，并监控其执行。您可以自定义显示哪些列、每个页面显示的行数，并过滤结果。您还可以创建自定义进程实例列表。

先决条件

- 已在 Business Central 中部署了带有进程定义的项目。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面中，单击左侧的高级过滤器图标，以打开进程实例 Advanced Filters 选项的列表。
3. 在 Advanced Filters 面板中，输入您想要用于自定义进程实例列表的过滤器的名称和描述，然后点击 Add New。
4. 从过滤器值列表中，选择参数和值来配置自定义进程实例列表，然后单击 Save。

将创建一个新过滤器，并立即应用到进程实例列表中。过滤器也保存在 Saved Filters 列表中。您可以点击 Manage Process Instances 页面左侧的星号图标来访问保存的过滤器。

28.3. 使用默认过滤器管理进程实例

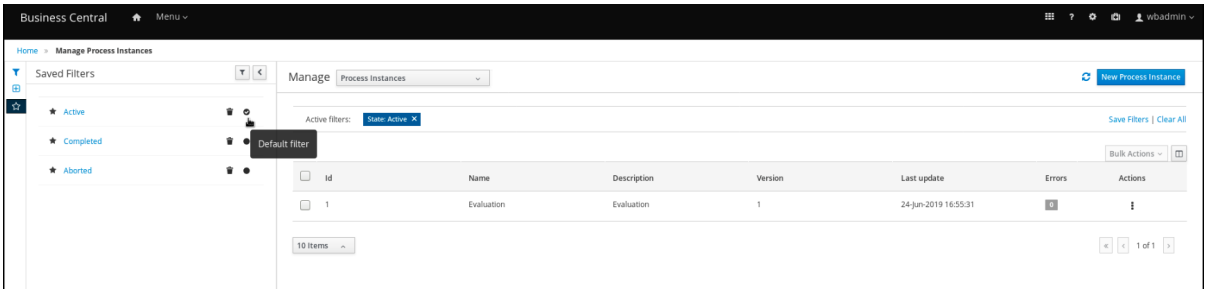
您可以使用 Business Central 中的 Saved Filter 选项将进程实例过滤器设置为默认过滤器。每次用户打开页面时，都将执行默认过滤器。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面上，单击页面左侧的星号图标，展开 Saved Filters 面板。

在 Saved Filters 面板中，您可以查看保存的高级过滤器。

进程实例的默认过滤器选择



3. 在 Saved Filters 面板中，将保存的进程实例过滤器设置为默认过滤器。

28.4. 使用基本过滤器查看进程实例变量

Business Central 提供基本过滤器来查看流程实例变量。您可以使用 Show/ hide 列将进程实例变量以列的形式查看。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面上，单击页面左侧的过滤器图标展开 Filters 面板。
3. 在 Filters 面板中，选择 Definition Id 并从列表中选择一个定义 ID。

   过滤器应用到当前进程实例列表。
4. 在屏幕右上角点击列图标（在 Bulk Actions 右侧），以显示或隐藏进程实例表中的列。
5. 单击星号图标，以打开 Saved Filters 面板。

在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

28.5. 使用高级过滤器查看进程实例变量

您可以使用 Business Central 中的 Advanced Filters 选项来查看进程实例变量。当您在列 processId 上创建过滤器时，您可以使用 Show/hide 列将进程实例变量查看为列。

流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 在 Manage Process Instances 页面上，单击高级过滤器图标以展开 Advanced Filters 面板。
3. 在 Advanced Filters 面板中，输入过滤器的名称和描述，然后单击 Add New。
4. 从 Select 列 列表中，选择 processId 属性。该值将更改为 processId != value1。
5. 从 Select 列 列表中，为查询选择 等于。
6. 在文本字段中，输入进程 ID 的名称。
7. 点 Save，过滤器应用到当前进程实例列表中。
8. 点击进程实例列表右上角的列图标（在 Bulk Actions右侧），将显示指定进程 ID 的进程实例变量。
9. 单击星号图标，以打开 Saved Filters 面板。

在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

28.6. 使用 BUSINESS CENTRAL 中止进程实例

如果进程实例已过时，您可以在 Business Central 中中止进程实例。

流程

1. 在 Business Central 中，转至 Menu → Manage → Process Instances，以查看可用进程实例的列表。
2. 从列表中选择您要中止的进程实例。
3. 在进程详情页面中，单击右上角的 Abort 按钮。

28.7. 来自 BUSINESS CENTRAL 的信号进程实例

您可以从 Business Central 发送信号进程实例。

先决条件

- 已在 Business Central 中部署了带有进程定义的项目。

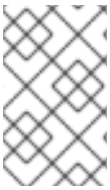
流程

1. 在 Business Central 中，前往 Menu → Manage → Process Instances。
2. 找到所需的进程实例，点  按钮并从下拉菜单中选择 Signal。
3. 填写以下字段：
   - 信号名称: Corresponds 到信号的 SignalRef 或 MessageRef 属性。此字段是必需的。

注意

您还可以通过在 **Message Ref** 值的前面添加 **Message-** 前缀，向进程发送消息事件。

- 信号数据 ：与信号相关的数据.这个字段是可选的。



注意

在使用 **Business Central** 用户界面时，您只能发出信号性中间捕获事件。

28.8. 异步信号事件

当来自不同进程定义的多个进程实例等待相同的信号时，它们会在同一线程中按顺序执行。但是，如果其中一个进程实例引发运行时异常，则所有其他进程实例都会受到影响，通常会导致回滚事务。为了避免这种情况，**Red Hat Process Automation Manager** 支持使用异步信号事件：

- 抛出中间信号事件
- 结束事件

28.8.1. 为中间事件配置异步信号

中间事件驱动业务流程的流。中间事件用于在执行业务流程期间捕获或抛出事件。中间事件处理进程执行过程中发生的特定情况。引发信号中间事件根据定义的属性生成信号对象。

您可以为 **Business Central** 中的中间事件配置异步信号。

先决条件

- 您已在 **Business Central** 中创建了一个项目，它至少包含一个业务流程资产。
- 已在 **Business Central** 中部署了带有进程定义的项目。

流程

1. 开启业务流程资产。
2. 在进程设计器 canvas 中，从左侧工具栏中拖放 Intermediate Signal。
3. 在右上角，点  打开 Properties 面板。
4. 展开 数据分配。
5. 单击 Assignments 子部分下的框。此时会打开 Task Data I/O 对话框。
6. 单击 Data Inputs 和 Assignments 旁边的 Add。
7. 在 Name 字段中输入抛出事件的名称作为 async。
8. 将 Data Type 和 Source 字段留空。
9. 点击 确定。

它将在每个会话中自动设置 executor 服务。这样可保证每个进程实例在不同的事务中都发出信号。

28.8.2. 为结束事件配置异步信号

结束事件表示完成业务流程。除 none 和终止结束事件外，所有结束事件都会抛出事件。引发信号结束事件用于完成进程或子进程流。当执行流进入元素时，执行流将完成并生成由其 SignalRef 属性标识的信号。

您可以为 Business Central 中结束事件配置异步信号。

先决条件

- *您已在 Business Central 中创建了一个项目，它至少包含一个业务流程资产。*
- *已在 Business Central 中部署了带有进程定义的项目。*

流程

1. *开启业务流程资产。*
2. *在进程设计器 canvas 中，从左侧工具栏拖放 End Signal。*
3. *在右上角，点*

打开 Properties 面板。
4. *展开 数据分配。*
5. *单击 Assignments 子部分下的框。此时会打开 Task Data I/O 对话框。*
6. *单击 Data Inputs 和 Assignments 旁边的 Add。*
7. *在 Name 字段中输入抛出事件的名称作为 async。*
8. *将 Data Type 和 Source 字段留空。*
9. *点击 确定。*

它将在每个会话中自动设置 executor 服务。这样可保证每个进程实例在不同的事务中都发出信号。

28.9. 进程实例操作

进程实例管理 API 会公开以下流程引擎和单个进程实例的操作。

- 获取进程节点 - 按实例 id ：返回所有节点，包括进程实例中存在的所有嵌入式子进程。您必须从指定进程实例检索节点，以确保节点存在并包含有效的 ID，以便其他管理操作可以使用它。

- 取消节点实例 - 按进程实例 id 和节点实例 id ：使用进程和节点实例 ID 取消进程实例中的节点实例。

- 重新触发节点实例 - 根据实例 ID 和节点实例 id，通过取消活动节点实例来触发 节点实例，并使用进程和节点实例 ID 创建相同类型的新节点实例。

- 更新计时器 - 根据进程实例 id 和计时器 ID：根据计时器开始所经过的时间更新活跃计时器的过期时间。例如，如果最初创建的计时器为一小时的延时，并且有 30 分钟后，您把它设置为在两小时内更新，它将过期一次，从更新时间起了一半小时。

  - delay ：计时器过期后持续的时间。

  - 周期 ：循环计时器的定时限间隔。

  - 重复限值 ：为循环计时器限制指定数量的过期时间。

- 更新相对于当前时间的定时的计时器 - 按进程实例 ID 和计时器 ID ：根据当前时间更新活跃的计时器过期时间。例如，如果最初创建的计时器为一小时的延时，并且有 30 分钟后，您把它设置为在 2 小时内更新，它将从更新时间起两小时内。

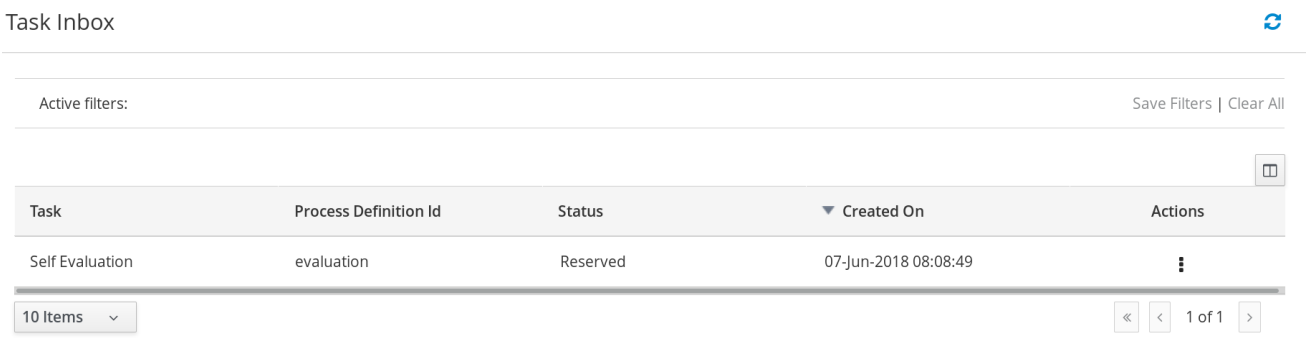
- 列出计时器实例 - 根据进程实例 ID ：为指定进程实例返回所有活跃的计时器。

- 触发节点 - 按进程实例 id 和节点 id：随时触发进程实例中的任何节点。

第 29 章 任务管理

分配给当前用户的任务会出现在 **Business Central** 的 **Menu → Track → Task Inbox** 中。您可以点击任务以打开并开始操作。

可以将用户任务分配给特定用户、多个用户或组群。如果分配给多个用户或一个组，它出现在所有分配用户的任务列表中，并且任何可能的操作器都可以声明任务。当某个任务分配给其他用户时，它不再出现在您的 **Task Inbox** 中。



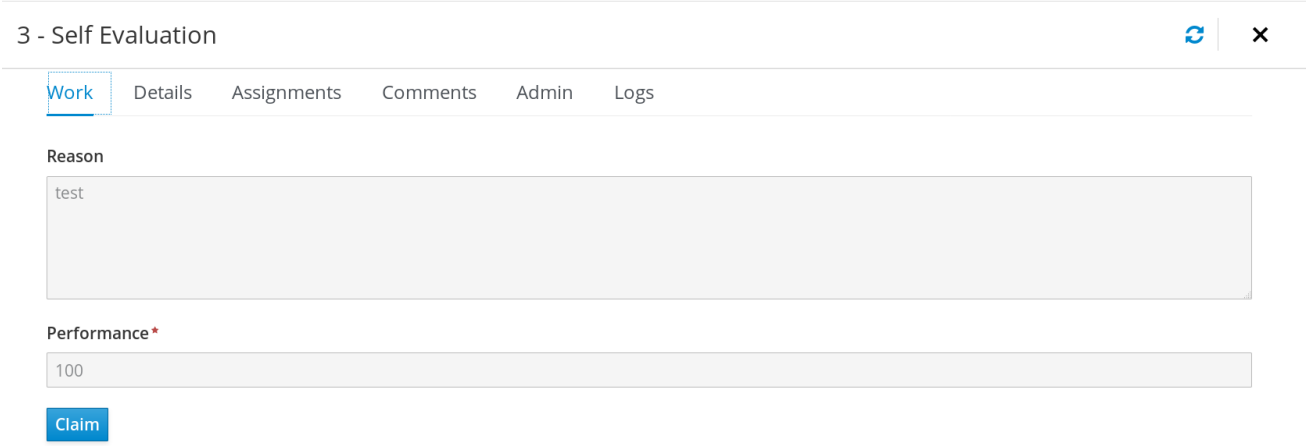
业务管理员可以在 **Business Central** 的 **Tasks** 页面中查看和管理所有用户任务，它位于 **Menu → Manage → Tasks** 下。具有 **admin** 或 **process-admin** 角色的用户可以访问 **Tasks** 页面，但默认情况下无法访问查看和管理任务的权限。

要管理所有任务，必须通过定义以下任一条件，将用户指定为进程管理员：

- 以 任务 **admin** 用户身份 指定用户。默认值为 **Administrator**。
- 用户从属于任务管理员组。默认值为 管理员。

您可以使用 **org.jbpm. ht.admin.user** 和 **org.jbpm.ht.admin.group** 系统属性配置用户和组分配。

您可以通过点击列表中的任务来打开、查看和修改任务详情，如到期日期、优先级或任务描述。任务页面中提供了以下标签页：



- ***work*** *：显示任务和任务所有者的基本详情。您可以点击* ***Claim*** *按钮来声明任务。要撤消声明过程，请点击* ***Release*** *按钮。*
- ***Details*** *：显示任务描述、状态和到期日期等信息。*
- ***分配*** *：显示任务的当前所有者，并允许您将任务委派给另一个人或组。*
- ***注释*** *：显示由任务用户添加的注释。您可以删除现有的注释并添加新注释。*
- ***admin*** *：显示任务的潜在所有者，并允许您将任务转发到另一个人或组。它还显示任务的实际所有者，您可以向任务的实际拥有者发送提醒。*
- ***日志*** *：显示包含任务生命周期事件（如任务启动、声明、完成）的任务日志，对任务字段进行更新（如任务的日期和优先级）。*

您可以点击页面左侧的 ***Filters*** *图标根据可用的过滤器参数过滤任务。有关过滤的详情，请参考* ***第 29.1 节 “任务过滤”****。*

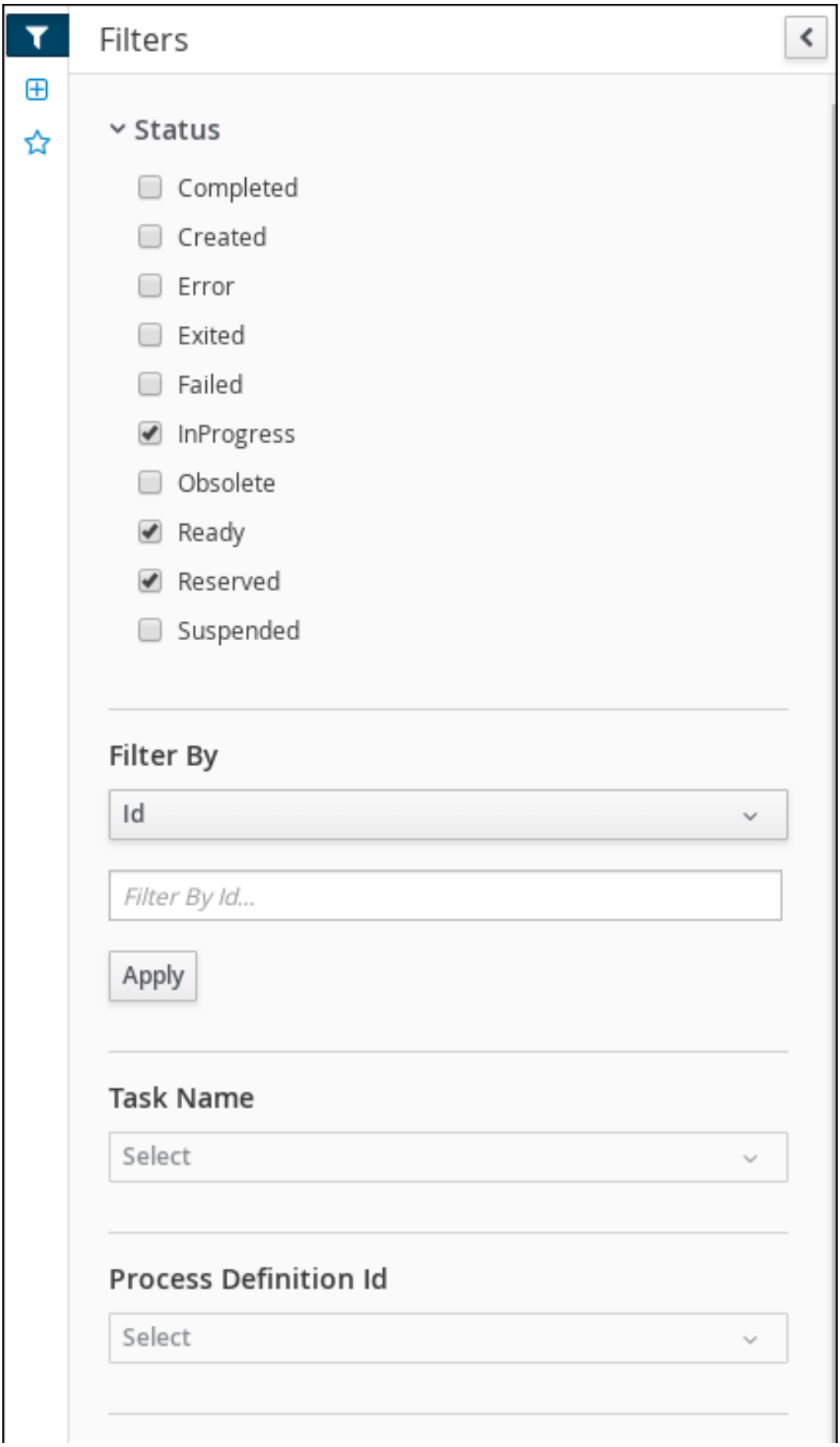
除了这些之外，您还可以创建自定义过滤器，以根据您定义的查询参数过滤任务。有关自定义任务过滤器的更多信息，请参阅 ***第 29.2 节 “创建自定义任务过滤器”****。*

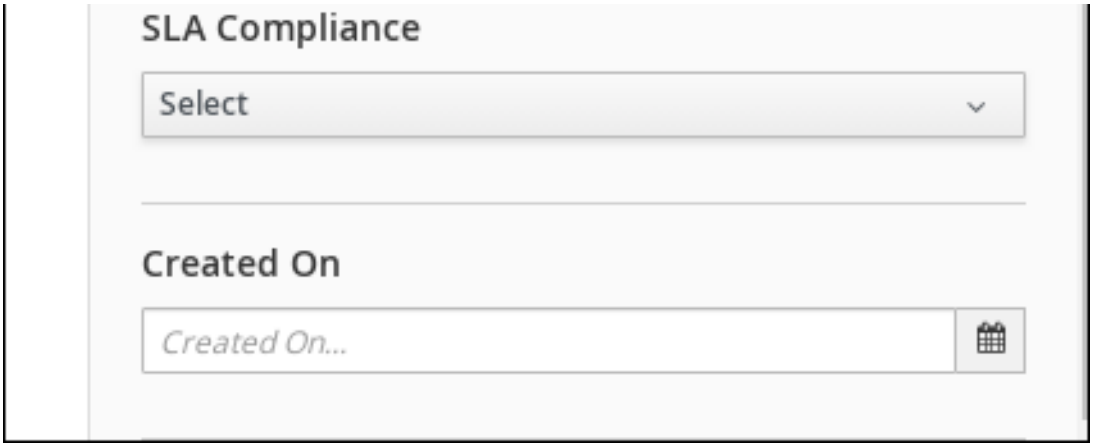
29.1. 任务过滤

对于 ***Menu → Manage → Tasks*** *和* ***Menu → Track → Task Inbox*** *中的任务，您可以使用* ***Filters*** *和*

***Advanced Filters** 面板根据需要对任务进行排序。*

*图 **29.1.** 过滤任务 **-** 默认视图*





Manage Tasks 页面仅可供管理员和处理管理员使用。

您可以根据 Filters 面板中的以下属性过滤任务：

状态

按任务状态过滤。您可以选择多个状态来显示满足任何所选状态的结果。删除 status 过滤器都会显示所有进程，无论状态是什么。

可用的过滤器状态如下：

- *完成*
- *已创建*
- *Error*
- *exited*
- *Failed*
- *InProgress*

- *obsolete*
- *Ready*
- *保留*
- *暂停*

id

按进程实例 ID 进行过滤。

输入：Num eric

任务

按任务名称过滤。

input： 字符串

关联密钥

按关联键过滤。

input： 字符串

实际所有者

按任务所有者过滤。

实际所有者指的是负责执行任务的用户。搜索基于用户 ID，它是一个唯一的值，它取决于 ID 管理系统。

input： 字符串

进程实例描述

按进程实例描述过滤。

input： 字符串

任务名称

按任务名称过滤。

进程定义 Id

按进程定义 Id 过滤。

SLA 合规性

按 SLA 合规状态过滤。

可用的过滤器状态如下：

- *Aborted*
- *满足*
- *N/A*
- *待处理*
- *违反*

Created On

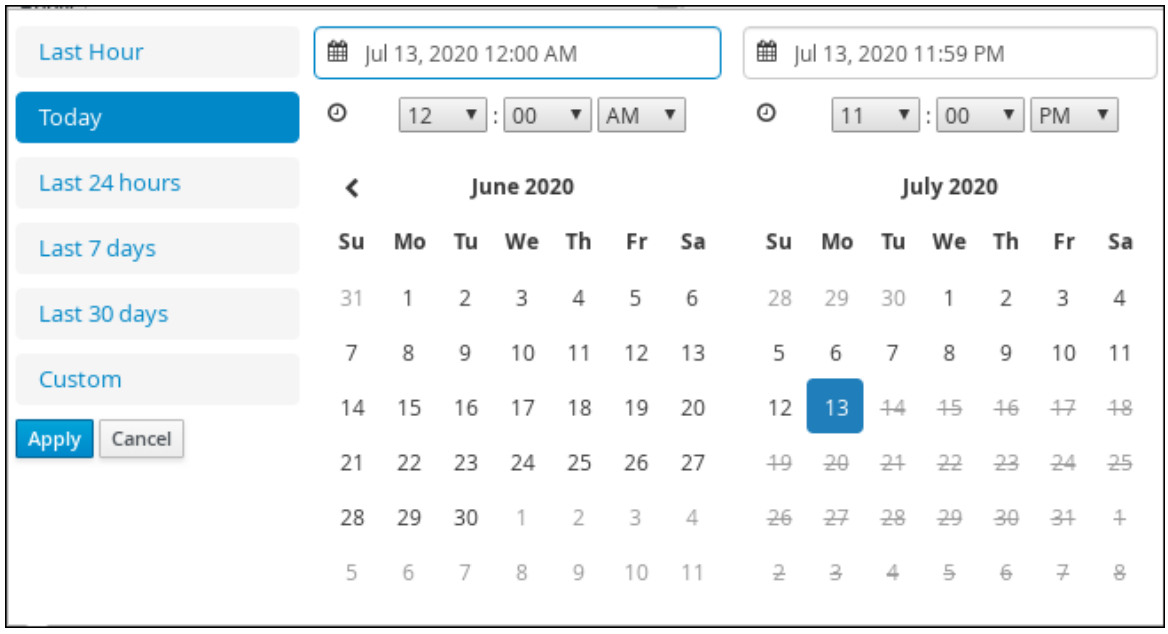
按日期或时间过滤。

此过滤器具有以下快速过滤器选项：

- *最后的 Hour*
- *回顾*
- *最后 24 小时*
- *最后 7 天*
- *最后 30 天*
- *Custom*

选择 Custom 日期和时间过滤将打开一个日历工具，用于选择日期和时间。

图 29.2. 按日期搜索



29.2. 创建自定义任务过滤器

您可以根据 Menu → Manage → Tasks 中提供的查询创建自定义任务过滤器，或者在 Menu → Track → Task 中为分配给当前用户的任务创建自定义任务过滤器。

流程

1. *在 Business Central 中，前往 Menu → Manage → Tasks*

2. *在 Manage Tasks 页面中，单击左侧的高级过滤器图标，以打开 Advanced Filters 选项列表。*

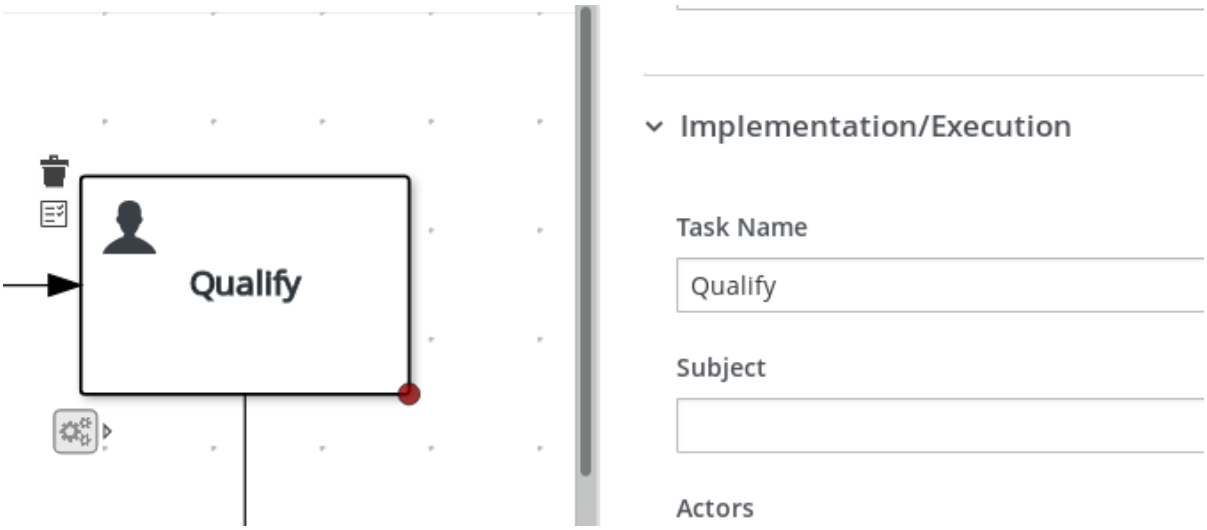
3. *在 Advanced Filters 面板中，输入过滤器的名称和描述，然后单击 Add New。*

4. *在 Select 列 下拉菜单中，选择 名称。*

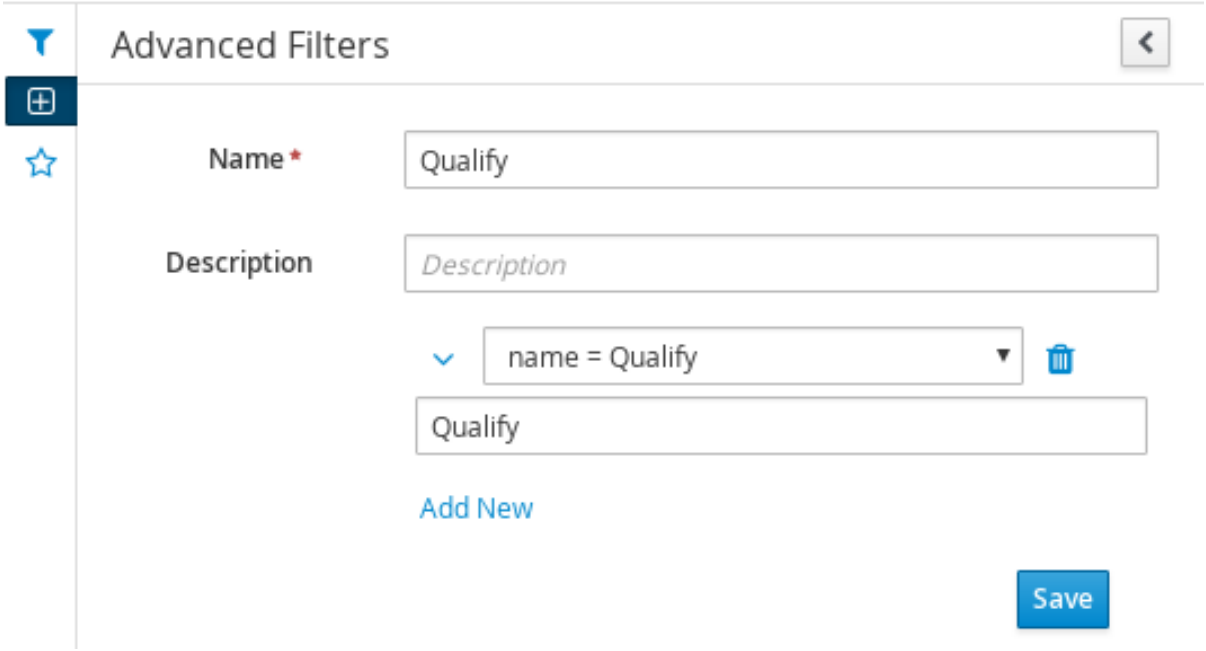
   下拉菜单的内容更改为 名称 != value1。

5. *再次单击下拉菜单，然后选择 等于。*

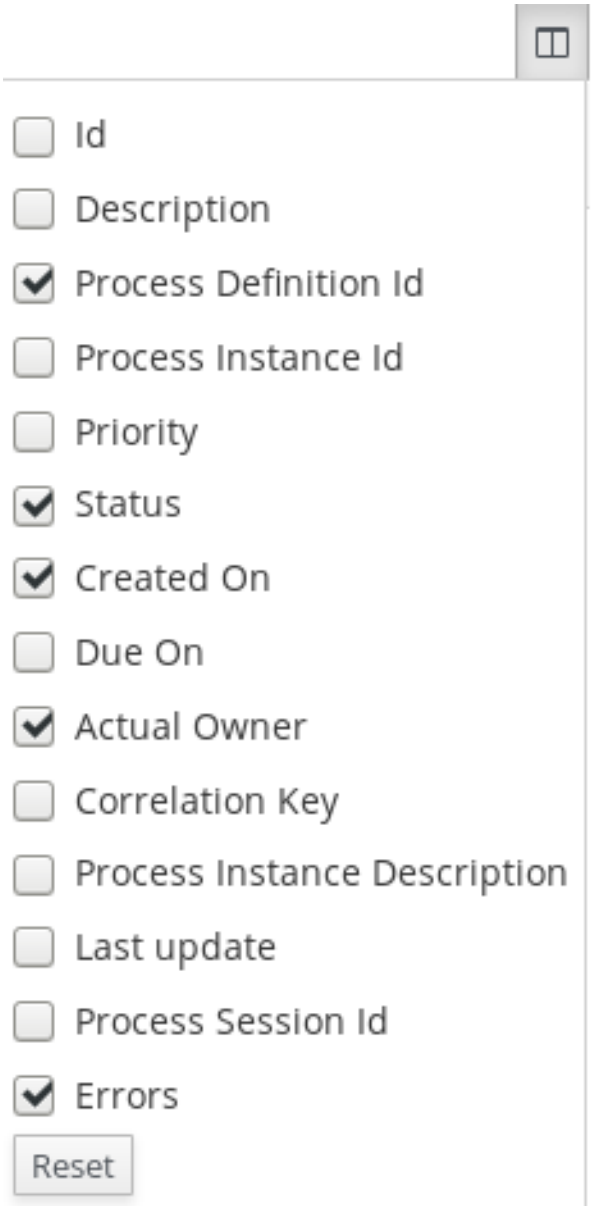
6. *将文本字段的值重写到您要过滤的任务名称。请注意，名称必须与相关业务流程中定义的值匹配：*



7. *点 Ok 保存自定义任务过滤器。*



应用过滤器带有指定限制后，一组可配置的列将基于特定的自定义任务过滤器，并包含以下列选项：



29.3. 使用默认过滤器管理任务

您可以使用 Business Central 中的 Saved Filter 选项将任务过滤器设置为默认过滤器。每次用户打开页面时，都将执行默认过滤器。

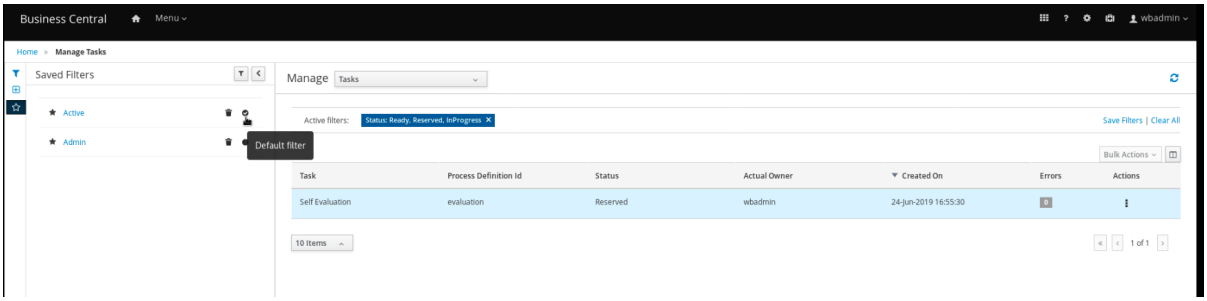
流程

1. 在 Business Central 中，进入 Menu → Track → Task Inbox 或 go to Menu → Manage → Tasks

2. 在 Task Inbox 页面或 Manage Tasks 页面上，单击页面左侧的星号图标，以展开 Saved Filters 面板。

   在 Saved Filters 面板中，您可以查看保存的高级过滤器。

任务或任务的默认过滤器选择



3. 在 Saved Filters 面板中，将保存的任务过滤器设置为默认过滤器。

29.4. 使用基本过滤器查看任务变量

Business Central 提供了基本的过滤器，来查看 管理任务 和任务中的任务变量。您可以使用 Show/hide 列将任务任务变量以列的形式查看。

流程

1. 在 Business Central 中，前往 Menu → Manage → Tasks，或前往 Menu → Track → Task Inbox。

2. 在 Task Inbox 页面上，点击页面左侧的过滤器图标展开 Filters 面板

3. 在 Filters 面板中，选择 Task Name。

   过滤器将应用于当前任务列表。

4. 点击任务列表右上角的 Show/hide 列，将显示指定任务 id 的任务变量。

5. 单击星号图标，以打开 Saved Filters 面板。

   在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

29.5. 使用高级过滤器查看任务变量

您可以使用 Business Central 中的 Advanced Filters 选项查看 管理任务和任务收 件中的任务变量。当您创建过滤器时，您可以使用 Show/ hide 列，以列的形式查看任务的任务变量。

流程

1. 在 Business Central 中，前往 Menu → Manage → Tasks，或前往 Menu → Track → Task Inbox。
2. 在 Manage Tasks 页面或 Task Inbox 页面中，点击高级过滤器图标展开 Advanced Filters 面板。
3. 在 Advanced Filters 面板中，输入过滤器的名称和描述，然后单击 Add New。
4. 从 Select 列 列表中，选择 name 属性。该值将更改为 名称 != value1。
5. 在 Select 列 列表中，为逻辑查询选择 等于。
6. 在文本字段中，输入任务的名称。
7. 点 Save，过滤器应用到当前任务列表。
8. 点击任务列表右上角的 Show/hide 列，将显示指定任务 id 的任务变量。
9. 单击星号图标，以打开 Saved Filters 面板。

   在 Saved Filters 面板中，您可以查看所有保存的高级过滤器。

29.6. 使用 MVEL 表达式在 BUSINESS CENTRAL 中设置任务的优先级

如果您的角色有设置任务优先级的权限，您可以使用 MVEL 表达式来设置 Business Central 中的 Task Inbox 页面中任务的优先级。

先决条件

- *您已在 Business Central 中创建了一个项目。*

流程

1. *在 Business Central 中，转至 Menu → Track → Task Inbox。*
2. *在 Task Inbox 页面中，点任务打开它。*
3. *在任务页面中，点击 Details 选项卡。*
4. *在 Task Priority 字段中，输入 MVEL 表达式。*
5. *点 Update。*

29.7. 在 BUSINESS CENTRAL 中管理自定义任务

自定义任务（工作项目）是可以运行自定义逻辑的任务。您可以在多个业务流程或 Business Central 的所有项目之间自定义和重新利用自定义任务。您还可以在设计器面板中添加自定义元素，包括名称、图标、子类别、输入和输出参数以及文档。Red Hat Process Automation Manager 在 Business Central 中的自定义任务存储库中提供一组自定义任务。您可以启用或禁用默认自定义任务，并将自定义任务上传到 Business Central 中以实施相关流程中的任务。

注意

Red Hat Process Automation Manager 包括一组有限的支持的自定义任务。不支持 Red Hat Process Automation Manager 中未包含的自定义任务。

流程

1. *在 Business Central 中，点右上角的*

并选择 Custom Tasks Administration。

本页列出了自定义任务安装设置，以及用于整个 Business Central 项目中流程的自定义任务。在此页面中启用的自定义任务将在项目级别设置中找到，您可以在其中安装要在进程中使用的每个自定义任务。在项目中安装自定义任务的方式是由您在此 自定义任务 管理页面上的 Settings 下启用或禁用的全局设置来确定。

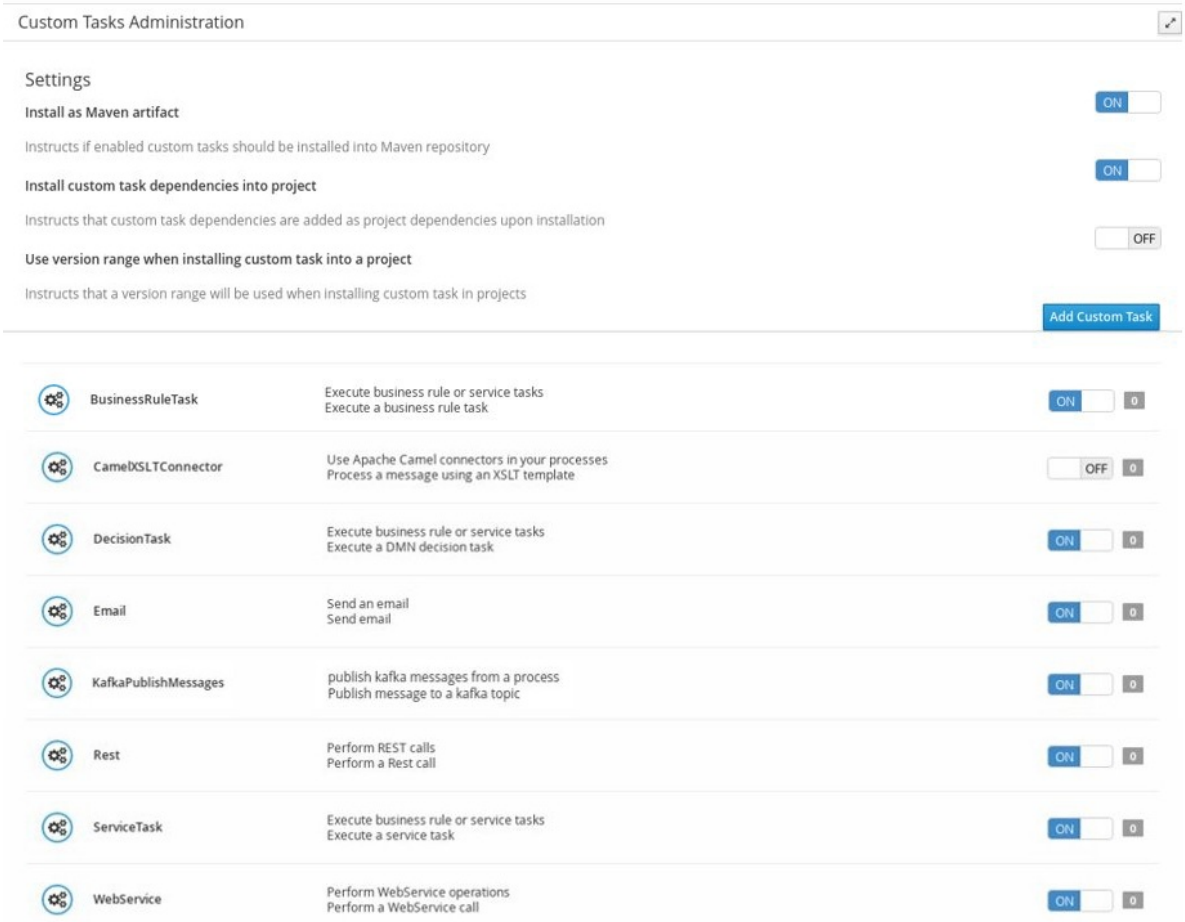
2. 在 Settings 下，启用或禁用每个设置，以确定用户在项目级别安装自定义任务时如何实施可用的自定义任务。

   可用的自定义任务设置如下：

   - 安装为 Maven 工件 ：将自定义任务 JAR 文件上传到配置了 Business Central 的 Maven 存储库（如果不存在）。
   - 将自定义任务依赖项安装到项目中 ：将任何自定义任务依赖项添加到安装任务的项目的 pom.xml 文件中。
   - 在将自定义任务安装到项目 时，请使用版本范围，而不是作为项目依赖项添加的自定义任务的固定版本。示例 ：[7.16， 而不是 7.16.0.Final

3. 根据需要启用或禁用（设置为 ON 或 OFF）任何可用的自定义任务。您在 Business Central 中所有项目的项目级别设置中显示自定义任务。

图 29.3. 启用自定义任务和自定义任务设置



4. *要添加自定义任务，请点击 **Add Custom Task**，浏览到相关的 **JAR** 文件，然后单击 **Upload** 图标。如果类实施了一个 **WorkItemHandler**，您可以通过将该文件单独添加到 **Business Central** 中，用 **.wid** 文件替换注解。*

5. *可选：要删除自定义任务，请点击您要删除的自定义任务行的 **remove** 并点 **Ok** 确认删除。*

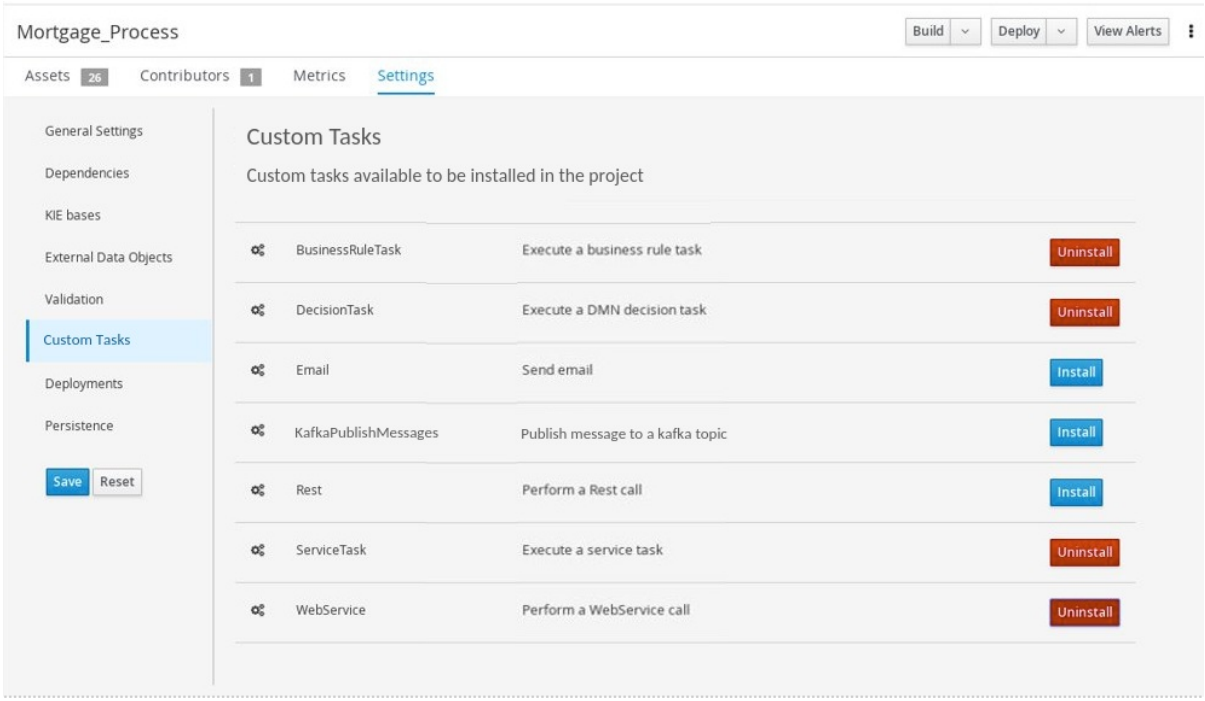
6. *在配置所有必需的自定义任务后，进入 **Business Central** 中的项目，进入 **Project Settings → Custom Tasks** 页面，以查看您启用的可用自定义任务。*

7. *对于每个自定义任务，点 **Install** 使该项目中进程可用的任务，或者点击 **Uninstall** 以从项目中的进程中排除任务。*

8. *如果在安装自定义任务时提示您输入其他信息，请输入所需信息，然后再次单击 **Install**。*

*自定义任务所需的参数取决于任务类型。例如，规则和决策任务需要工件 **GAV** 信息（组 **ID**、**Artifact ID**、版本），电子邮件任务需要主机和端口访问信息，而 **REST** 任务需要 **API** 凭证。其他自定义任务可能不需要任何其他参数。*

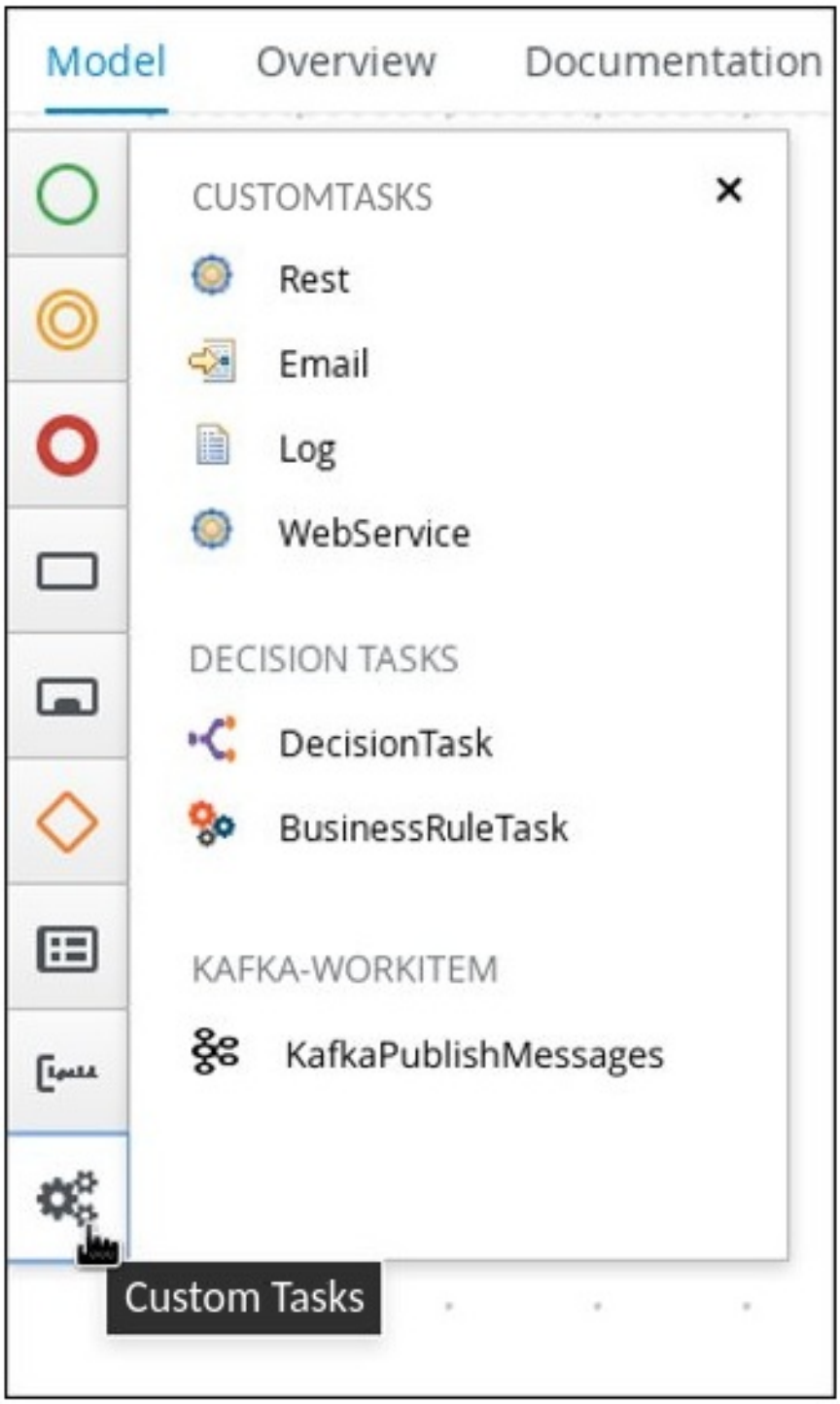
图 29.4. 安装要在进程中使用的自定义任务



9. *点击 **Save**。*

10. *返回到项目页面，选择或在项目中添加业务流程，然后在流程设计器面板中选择 **Custom Tasks** 选项以查看您启用并安装的可用自定义任务：*

图 29.5. 访问在进程设计器中安装的自定义任务



29.8. 用户任务管理

通过用户任务，您可以包括人工操作作为您所创建的业务流程的输入。用户任务管理提供了操作用户和组任务分配、数据处理、基于时间自动通知和重新分配的方法。

Business Central 中提供了以下用户任务操作：

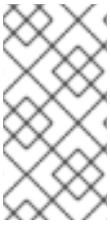
- 按任务 id 添加/删除潜在所有者 ：添加或删除使用任务 ID 的用户和组。
- 添加/删除排除所有者 - 按任务 id: 添加或删除使用任务 ID 的排除所有者。
- 按任务 id 添加/删除业务管理员 ：使用任务 ID 添加或删除业务管理员。
- 添加任务输入 - 按任务 id ：提供在任务使用任务 ID 创建后修改任务输入内容的方法。
- 删除任务输入 - 按任务 id: 使用任务 ID 删除任务输入变量。
- 删除任务输出 - 按任务 id: 使用任务 ID 删除任务输出变量。
- 在给定时间后调度给用户/组的新重新分配 - 按任务 ID 划分：根据时间表达式和任务状态调度自动重新分配：
  - 如果任务 没有 移到 InProgress 状态，则重新启动 : Used。
  - 如果未完成，则重新分配 ：如果任务没有移到 Completed 状态，则使用。
- 根据任务 id，调度在给定时间后向用户授予的用户/组的新电子邮件通知 ：根据时间表达式和任务状态调度自动电子邮件通知：
  - 如果任务 没有 移到 InProgress 状态，则通知没有启动 : Used。
  - 如果任务 没有移到 Completed 状态，则通知未完成: Used。
- 列出调度任务通知 - 按任务 id: 使用任务 ID 返回所有活动任务通知。
- 列出调度任务重新分配 - 按任务 id ：返回所有活动任务重新分配任务 ID。

- 取消任务通知 - 按任务 id 和 notification id: Cancels 和 unschedules 任务通知，使用任务 ID。

- 取消任务重新分配 - 按任务 ID 和重新分配 id ：取消调度任务 ID，重新分配任务 ID。

29.9. 对任务进行批量操作

在 Business Central 中的 Tasks 和 Task Inbox 页面中，您可以通过单个操作对多个任务执行批量操作。



注意

如果不允许基于任务状态指定批量操作，则会显示通知，操作不会在该特定任务上执行。

29.9.1. 批量声明任务

在 Business Central 中创建任务后，您可以广泛声明可用的任务。

流程

1. 在 Business Central 中，完成以下步骤之一：

   - 要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。

   - 要查看 Tasks 页面，请选择 Menu → Manage → Tasks。

2. 要批量声明任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。

3. 在 Bulk Actions 下拉列表中选择 Bulk Claim。

4. 要确认，请单击 Claim selected tasks 窗口。

对于选择每个任务，会显示通知来显示结果。

29.9.2. 批量释放任务

您可以批量释放您拥有的任务供他人使用。

流程

1. 在 Business Central 中，完成以下步骤之一：
   - 要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。
   - 要查看 Tasks 页面，请选择 Menu → Manage → Tasks。
2. 要批量释放任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。
3. 在 Bulk Actions 下拉列表中选择 Bulk Release。
4. 要确认，请单击 Release selected tasks 窗口。

对于选择每个任务，会显示通知来显示结果。

29.9.3. 恢复批量的任务

如果 Business Central 中暂停的任务，您可以批量恢复它们。

流程

1. 在 Business Central 中，完成以下步骤之一：

- *要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。*
- *要查看 Tasks 页面，请选择 Menu → Manage → Tasks。*

2. *要恢复批量的任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。*
3. *从 Bulk Actions 下拉列表中，选择 Bulk Resume。*
4. *要确认，请点击 Resume selected tasks 窗口。*

对于选择每个任务，会显示通知来显示结果。

29.9.4. 批量挂起任务

在 Business Central 中创建任务后，您可以暂停批量的任务。

流程

1. *在 Business Central 中，完成以下步骤之一：*
   - *要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。*
   - *要查看 Tasks 页面，请选择 Menu → Manage → Tasks。*
2. *要批量暂停任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或多个任务。*
3. *在 Bulk Actions 下拉列表中选择 Bulk Suspend。*

4. 要确认，请点击 Suspend 选择的任务 窗口。

对于选择每个任务，会显示通知来显示结果。

29.9.5. 批量重新分配任务

在 Business Central 中创建任务后，您可以重新分配您的任务并将其委托给其他任务。

流程

1. 在 Business Central 中，完成以下步骤之一：
   - 要查看 Task Inbox 页面，请选择 Menu → Track → Task Inbox。
   - 要查看 Tasks 页面，请选择 Menu → Manage → Tasks。
2. 要批量重新分配任务，在 Task Inbox 页面上或 Manage Tasks 页面中，从 Task 表中选择两个或更多任务。
3. 在 Bulk Actions 下拉列表中，选择 Bulk Reassign。
4. 在 任务 重新分配窗口中，输入您要重新分配任务的用户 ID。
5. 点 Delegate。

对于选择每个任务，会显示通知来显示结果。

第 30 章 管理日志数据

Red Hat Process Automation Manager 管理所需的维护运行时数据。它自动删除一些数据，包括以下数据类型：

- 处理实例数据，在进程实例完成后删除。
- 工作项目数据，在工作项目完成后删除。
- 任务实例数据，在完成给定任务所属的进程后移除。

未清理的运行时数据会包括基于所选运行时策略的会话信息数据。

- 单例策略 确保会话信息的运行时数据不会被自动删除。
- 根据请求策略，可以在请求终止时自动删除。
- 当进程实例 映射到完成或中止的会话时，每个进程实例会自动移除进程实例数据。

Red Hat Process Automation Manager 还提供审计日志数据表。您可以使用这些表来跟踪当前和过去的进程实例。默认情况下，Red Hat Process Automation Manager 不会从审计日志表中删除任何数据。

可以通过三种方式来管理和维护审计数据表：

- 您可以使用 Business Central 设置自动清理这些表，如 第 30.1 节 “设置自动清理任务” 所述。
- 您可以使用 Java API 手动删除表中的信息，如 第 30.2 节 “手动清理” 所述。
- 您可以在 Red Hat Process Automation Manager 数据库上运行自定义查询，包括审计日志

表，如 第 30.4 节 “在 Red Hat Process Automation Manager 数据库上运行自定义查询” 所述。

30.1. 设置自动清理任务

您可以在 Business Central 中设置自动清理作业。

流程

1. 在 Business Central 中，前往 Manage > Jobs。
2. 单击新建作业。
3. 输入 Business Key、Due On 和 Retries 字段的值。
4. 在 Type 字段中输入以下命令。

```
org.jbpm.executor.commands.LogCleanupCommand
```

5. 要配置参数，请完成以下步骤：
   a. 点 Advanced 标签页。
   b. 点 Add Parameter。
   c. 在 Key 列中，输入参数。
   d. 在 Value 列中，输入一个参数。

有关该命令的参数列表，请参阅 第 30.3 节 “从数据库中删除日志”。

6. 点 Create。

Business Central 创建了自动清理作业。

30.2. 手动清理

要执行手动清理，您可以使用审计 Java API。Audit API 由以下区域组成：

表 30.1. Audit API 区域

Name	描述
进程审计	它用于清理 process、节点和变量日志，可在 **jbpm-audit** 模块中访问。 例如，您可以按照如下所示访问该模块： **org.jbpm.process.audit.JPAAuditLogService**
任务审计	它用于清理 **jbpm-human-task-audit** 模块中可访问的任务和事件。 例如，您可以按照如下所示访问该模块： **org.jbpm.services.task.audit.service.TaskJPAAuditService**
执行程序任务	它用于清理 **jbpm-executor** 模块中可访问的 executor 作业和错误。 例如，您可以按照如下所示访问该模块： **org.jbpm.executor.impl.jpa.ExecutorJPAAuditService**

30.3. 从数据库中删除日志

使用 LogCleanupCommand executor 命令清除使用数据库空间的数据。LogCleanupCommand 由逻辑组成，用于自动清理所有或选定数据。

有几个配置选项可与 LogCleanupCommand 一起使用：

表 30.2. LogCleanupCommand 参数表

Name	描述	isclusive
SkipProcessLog	指明进程和节点实例，在命令运行时可以跳过进程和节点日志清理。默认值为 **false**。	不，它与其它参数一起使用。

Name	描述	isclusive
SkipTaskLog	指示任务 audit 和 event log cleanup 已被跳过。默认值为 **false**。	不，它与其它参数一起使用。
SkipExecutorLog	指示是否已跳过 Red Hat Process Automation Manager executor 条目 cleanup。默认值为 **false**。	不，它与其它参数一起使用。
SingleRun	指明作业例程是否只运行一次。默认值为 **false**。	不，它与其它参数一起使用。
NextRun	调度下一个作业执行。默认值为 **24h**。 例如，对于每 12 小时执行的作业，设置为 **12h**。如果您将 **SingleRun** 设置为 **true**，除非同时设置了 **SingleRun** 和 **NextRun**，则计划会被忽略。如果同时设置了这两个程序，则 **NextRun** 调度将具有优先权。ISO 格式可用于设置精确的日期。	不，它与其它参数一起使用。
OlderThan	删除比指定日期旧的日志。日期格式为 **YYYY-MM-DD**。通常，此参数用于单次运行作业。	是，它不与 **OlderThanPeriod** 参数一起使用。
OlderThanPeriod	超过指定计时器表达式的日志将被移除。例如，将 **30d** 设置为删除超过 30 天的日志。	是，它不与 **OlderThan** 参数一起使用。
ForProcess	指定已删除的日志的进程定义 ID。	不，它与其它参数一起使用。
RecordsPerTransaction	表示删除的事务中的记录数量。默认值为 **0，** 表示所有记录。	不，它与其它参数一起使用。
ForDeployment	指定已删除的日志的部署 ID。	不，它与其它参数一起使用。
EmfName	用于执行删除操作的持久性单元名称。	Not applicable

注意

LogCleanupCommand 不会删除任何活跃的实例，如运行的进程实例、任务实例或执行程序作业。

30.4. 在 RED HAT PROCESS AUTOMATION MANAGER 数据库上运行自定义查询

您可以使用 ExecuteSQLQueryCommand executor 命令在红帽流程自动化管理器数据库上运行自定义查询，包括审计日志数据表。您可以设置一个在 Business Central 中运行这个命令的作业。

流程

1. 在 Business Central 中，选择 Manage > Jobs。
2. 单击新建作业。
3. 输入 Business Key、Due On 和 Retries 字段的值。
4. 在 Type 字段中输入以下命令。

```
org.jbpm.executor.commands.ExecuteSQLQueryCommand
```

5. 要配置参数，请完成以下步骤：
   a. 打开 Advanced 选项卡。
   b. 点 Add Parameter。
   c. 在 Key 列中，输入参数值。
   d. 在 Value 列中，输入参数值。

   有关该命令的参数列表，请参阅 第 30.4.1 节 “ExecuteSQLQueryCommand 命令的参数”。

6. 点 Create。Business Central 可创建自定义查询作业。
7. 可选：如果要获取查询的结果，请完成以下步骤：
   a. 在 Business Central 显示的作业列表中，找到您启动的作业。如果列表中不存在该作业，请从 Active 过滤器列表中删除任何过滤器。

b. 记录作业的 id 值。

c. 使用 Web 浏览器，访问位于 < kie_server_address>/docs 的 KIE 服务器上的 Swagger 文档，例如 http://localhost:8080/kie-server/docs/。

d. 单击 GET /server/jobs/{jobId} 请求。

e. 在 jobId 字段中输入您记录的 id 值。

f. 从 withErrors 列表中，选择 true。

g. 从 withData 列表中，选择 true。

h. 点 Execute。

i. 检查 Server 响应 字段。如果 SQL 查询成功，则结果将受 "response-data" 键下。

30.4.1. ExecuteSQLQueryCommand 命令的参数

ExecuteSQLQueryCommand executor 命令在红帽流程自动化管理器数据库上运行自定义查询，包括审计日志表。有关审计日志表的 schema，请参阅 Red Hat Process Automation Manager 中的进程引擎。

您可以为 ExecuteSQLQueryCommand 命令配置以下参数。

表 30.3. ExecuteSQLQueryCommand 参数表

Name	描述
SingleRun	**真**（如果可以触发一次查询）。如果可以多次触发查询，则为 **false**。
EmfName	用于运行查询的持久性单元名称

Name	描述
BusinessKey	与查询一起使用的业务密钥。如果在 Business Central 中配置命令，请使用您为作业设置的业务密钥
SQL	要执行的原生 SQL 查询。前一个带有 **:** 字符的参数
parametersList	SQL 查询中的所有参数列表。使用 **、字符分隔** 参数
SQL 参数名称	SQL 参数的值。为每个 SQL 参数创建单独的命令参数

例如，您可以使用带有两个参数的查询：

```
SELECT * FROM RequestInfo WHERE id = :paramId AND businessKey = :paramKey
```

*为 **ExecuteSQLQueryCommand** 命令设置以下参数：*

- ***SQL : select * FROM RequestInfo WHERE id = :paramId and BusinessKey = :paramKey ;***
- ***parametersList:paramId,paramKey***
- ***paramId: id的值***
- ***paramKey ： businessKey的值***

第 31 章 执行错误管理

当执行错误发生业务流程时，进程将停止并恢复到最新的稳定状态（最安全点）并继续执行。如果整个事务回滚没有处理任何 kind 错误，使进程实例处于之前的等待状态。对向进程引擎发送请求的调用者可见执行错误。

具有进程管理员(process-admin)或管理员（管理员）角色的用户可以访问 Business Central 中的执行错误消息。执行错误消息传递提供以下主要优点：

- 更好的可追溯性
- 关键进程的可见性
- 基于错误情况报告和分析
- 外部系统错误处理和编译

31.1. 查看 BUSINESS CENTRAL 中的进程执行错误

您可以在 Business Central 中的两个位置查看进程错误：

- 菜单 → Manage → Process Instances
- 菜单 → Manage → Execution Errors

在 Manage Process Instances 页面中，Errors 列显示当前进程实例的错误数。

先决条件

- 在 Business Central 中运行进程时发生了错误。

流程

1. 在 Business Central 中，转至 Menu → Manage → Process Instances，并在 错误 列中显示的数字上。

2. 单击 Errors 列中显示的错误数量，以导航到 Manage Execution Errors 页面。

   Manage Execution Errors 页面显示所有进程实例的错误列表。

31.2. 管理执行错误

根据定义，检测到并存储的每个进程错误都不会被某些或部分处理（如果自动错误恢复）。您可以查看已或未确认的过滤错误列表。解除错误可保存用户信息和可追溯时间戳。

流程

1. 在 Business Central 中，选择 Menu → Manage → Execution Errors。

2. 从列表中选择一个错误，以打开 Details 选项卡。Details 选项卡显示有关错误或错误的信息。

3. 单击确认按钮确认错误。您可以在 Manage Execution Errors 页中 选择 Yes 来查看确认的错误。

   如果错误与某个任务相关，则会显示 Go to Task 按钮。

4. 可选： 如果需要，点击 Go to Task 按钮，以查看 Manage Tasks 页面中的相关作业信息。

   在 Manage Tasks 页面中，您可以重启、重新调度或重试对应的任务。

31.3. 错误过滤

要在 Manage Execution Errors 屏幕中执行错误，您可以使用 Filters 面板只显示符合所选条件的错误。

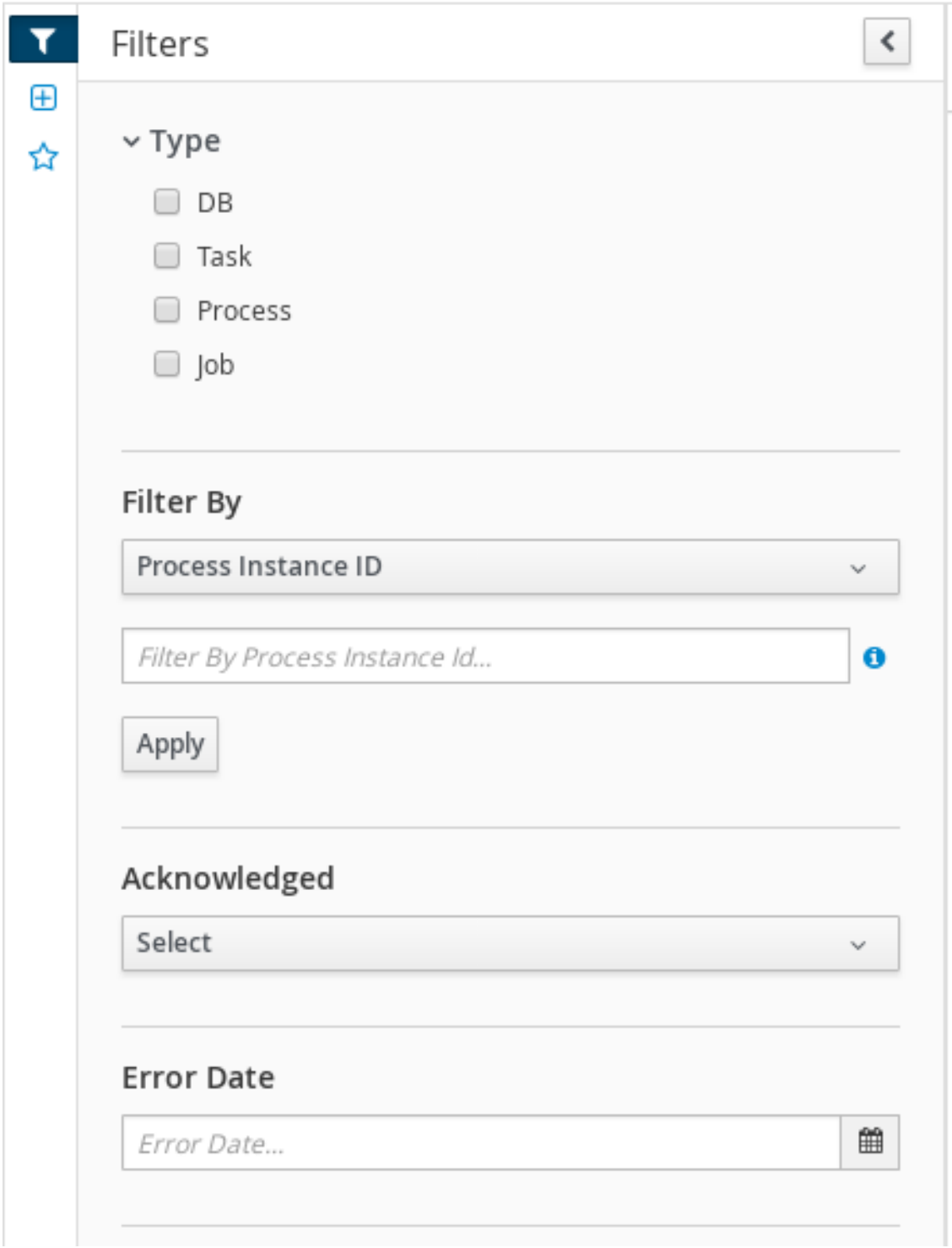
先决条件

- *打开 Manage Execution Errors 屏幕。*

流程

根据需要在屏幕左侧的 Filters 面板中进行更改：

图 31.1. 过滤错误 - 默认视图



类型

根据类型过滤执行错误。您可以选择多个类型过滤器。如果取消选择所有类型，则会显示所有错误，无论类型是什么。

可用的执行错误类型如下：

- DB
- 任务
- Process
- 作业

进程实例 ID

按进程实例 ID 进行过滤。

输入：Num eric

任务 Id

按作业 ID 过滤。作业 ID 在创建作业时自动创建。

输入：Num eric

id

按进程实例 ID 进行过滤。

输入：Num eric

已确认

过滤已确认或尚未确认的错误。

错误日期

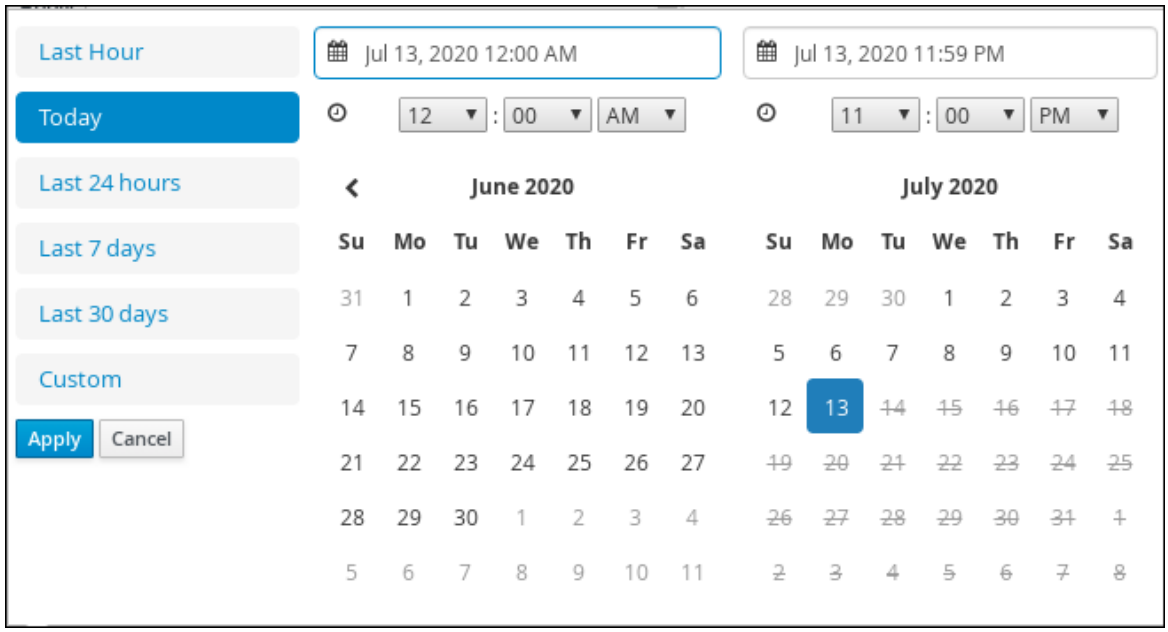
由发生错误的日期或时间过滤。

此过滤器具有以下快速过滤器选项：

- *最后的 Hour*
- *回顾*
- *最后 24 小时*
- *最后 7 天*
- *最后 30 天*
- *Custom*

  选择 Custom 选项以打开日历工具，以选择日期和时间范围。

图 31.2. 按日期搜索



31.4. 自动确认执行错误

默认情况下，当发生错误时，执行错误将不会被确认。为避免需要手动确认每个执行错误，您可以将作业配置为自动识别一些或所有执行错误。

注意

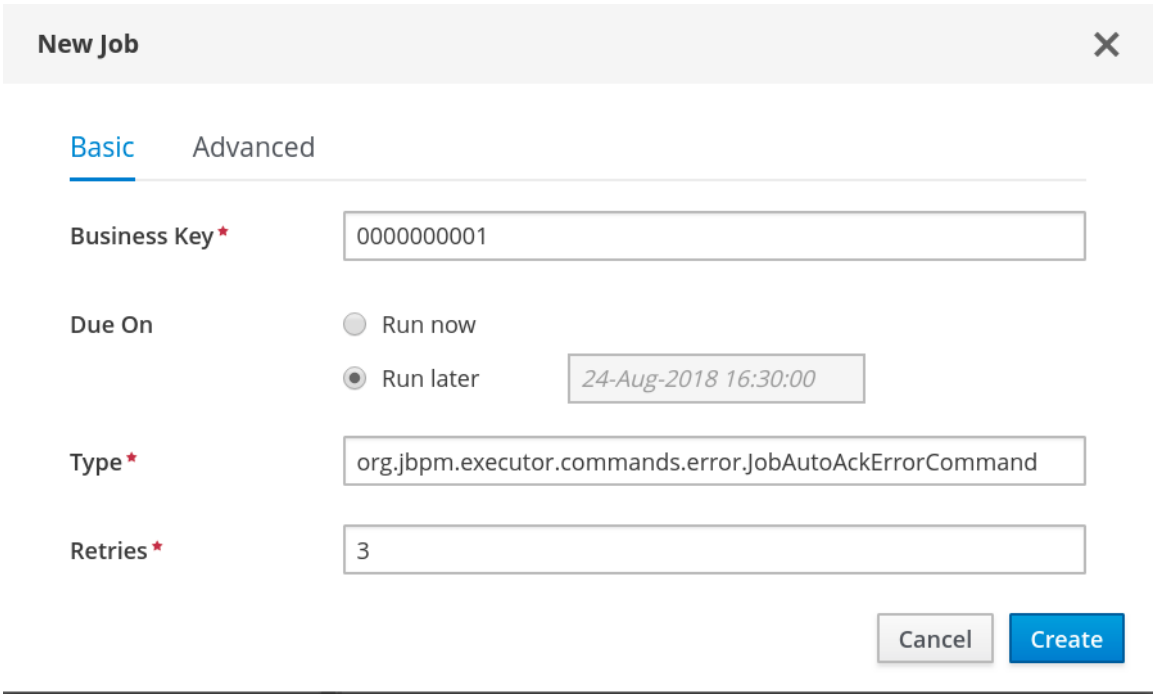
如果您配置了一个自动确认作业，则作业会默认每天运行。要仅自动确认执行错误一次，请将 SingleRun 参数设置为 true。

流程

1. 在 Business Central 中，选择 Menu → Manage → Jobs。
2. 在屏幕右上角，单击 New Job。
3. 在 Business Key 字段中输入作业的任何标识符。
4. 在 Type 字段中输入 auto-acknowledge 作业的类型：
   - org.jbpm.executor.commands.error.JobAutoAckErrorCommand: Acknowledge all type Job 的执行错误，其中作业现在取消、完成或重新调度另一个执行。
   - org.jbpm.executor.commands.error. Task AutoAckErrorCommand: Acknowledge the task with the task in a exit state(completed, failed, exited, obsolete)。
   - org.jbpm.executor.commands.error.ProcessAutoAckErrorCommand: Acknowledge all type of the process instance is been finished（完成或中止），或者错误的来源已经完成的任务。
5. 选择完成任务的时间：
   - 要立即运行作业，请选择 Run now 选项。
   -

要在特定时间运行作业，请选择 稍后运行。Run later 选项旁边会出现一个日期和时间字段。点字段打开日历，并计划作业的特定时间和日期。

图 31.3. 调度自动确认作业示例



6. *默认情况下，在初始运行一次作业后，每天运行一次作业。要更改此设置，请完成以下步骤：*

   a. *点 Advanced 标签页。*

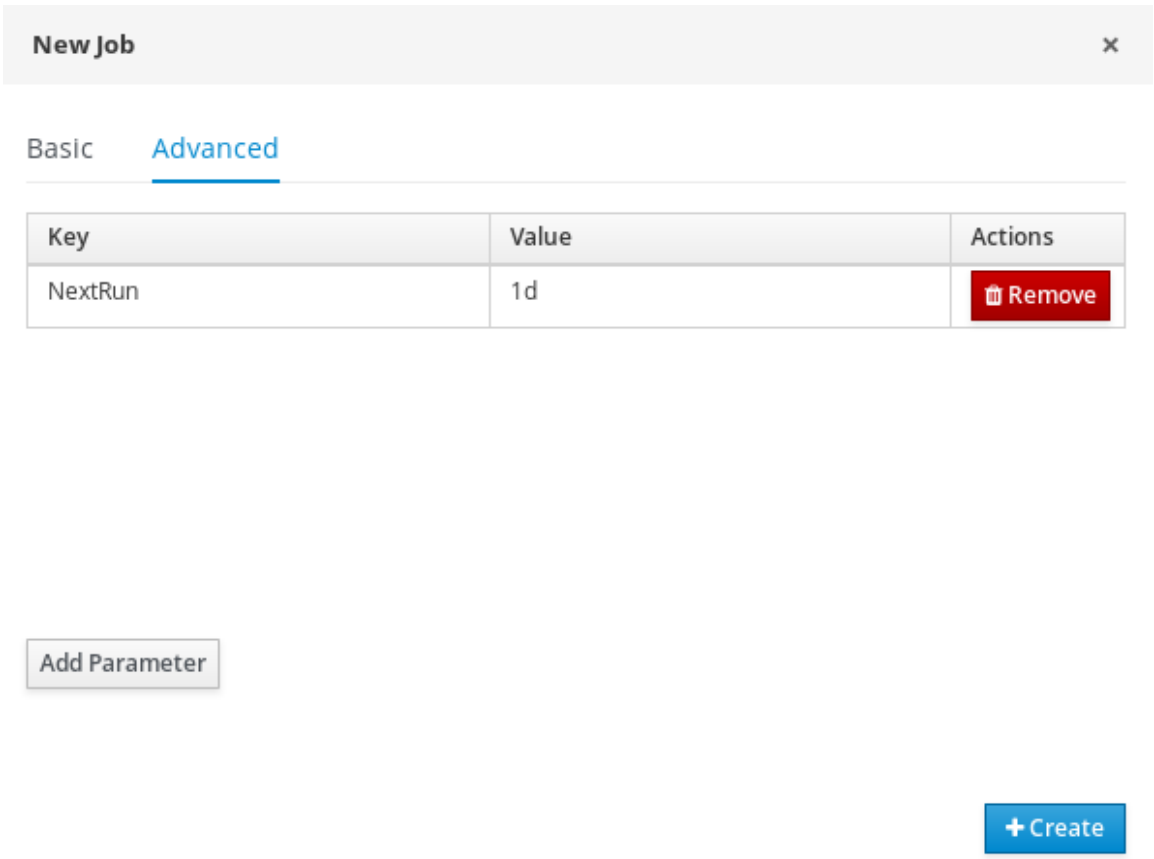
   b. *点 Add Parameter 按钮。*

   c. *输入您要应用到作业的配置参数：*

      - *如果您希望作业只运行一次，使用值 true 添加 SingleRun 参数。*
      - *如果您希望他定期运行作业，使用有效时间表达式的值添加 NextRun 参数，如 2h、5d、1m 等等。*

   d. *可选：要设置自定义实体管理器工厂名称，请输入 EmfName 参数。*

图 31.4. 为自动确认作业设置参数示例



7. *单击 **Create** 以创建作业，再返回到 **Manage Jobs** 页面。*

31.5. 清理错误列表

*进程引擎在 **ExecutionErrorInfo** 数据库表中存储执行错误。如果要永久清理表并删除错误，您可以使用 **org.jbpm.executor.commands.ExecutionErrorCleanupCommand** 命令调度作业。*

*该命令会删除与 **completed** 或 **aborted** 进程实例关联的执行错误。*

流程

1. *在 **Business Central** 中，选择 **Menu** → **Manage** → **Jobs**。*
2. *在屏幕右上角，单击 **New Job**。*
3. *在 **Business Key** 项中输入作业的任何标识符。*

4. 在 Type 字段中输入 org.jbpm.executor.commands.ExecutionErrorCleanupCommand。

5. 选择完成任务的时间：

   - 要立即运行作业，请选择 Run now 选项。
   - 要在特定时间运行作业，请选择 稍后运行。Run later 选项旁边会出现一个日期和时间字段。点字段打开日历，并计划作业的特定时间和日期。

6. 点 Advanced 标签页。

7. 根据需要添加以下任一参数：

   - DateFormat ：参数中日期的格式。如果没有设置，则使用 yyyy-MM-dd，如 SimpleDateFormat 类的模式中所示。
   - EmfName ：用于查询的自定义实体管理器工厂的名称。
   - SingleRun ：计划一个执行的作业。如果设置为 true，则作业一次运行一次，且不会被调度为重复执行。
   - NextRun ：调度作业在一段时间内重复执行。该值必须是有效的时间表达式，例如 1d、5h、10m。
   - OlderThan ：只删除比设置日期旧的错误。该值必须是日期。
   - OlderThanPeriod ：与当前时间相比，只删除超过给定期限的错误。该值必须是有效的时间表达式，例如 1d、5h、10m。
   - ForProcess ：仅删除与进程定义相关的错误。该值必须是进程 definiton 的标识符。

- *ForProcessInstance ：仅删除与进程实例相关的错误。该值必须是进程实例的标识符。*

- *ForDeployment ：仅删除与部署标识符相关的错误。该值必须是部署标识符。*

第 32 章 进程实例迁移

进程实例迁移(PIM)是包含用户界面和后端的独立服务。它被打包为 Quarkus mutable JAR 文件。您可以使用 PIM 服务来定义两个不同的进程定义之间的迁移，称为迁移计划。然后，用户可以将迁移计划应用到特定 KIE 服务器中正在运行的进程实例。

有关 PIM 服务的更多信息，请参阅 KIE（Drools、OptaPlanner 和 jBPM） 中的 处理实例迁移服务。

32.1. 安装进程实例迁移服务

您可以使用流程实例迁移(PIM)服务来创建、导出和执行迁移计划。PIM 服务通过 GitHub 存储库提供。要安装 PIM 服务，请克隆 GitHub 存储库，然后运行该服务并在 Web 浏览器中访问该服务。

先决条件

- 您已在备份的 Red Hat Process Automation Manager 开发环境中定义了进程。
- 已安装 Java 运行时环境(JRE)版本 11 或更高版本。

流程

1. 从 Red Hat Process Automation Manager 7.13 的 Software Downloads 页面中下载 rhpam-7.13.5-add-ons.zip 文件。
2. 提取 rhpam-7.13.5-add-ons.zip 文件。
3. 提取 rhpam-7.13.5-process-migration-service.zip 文件。
4. 输入以下命令来创建数据库表。将 <user > 替换为您的用户名，& lt;host > 替换为本地主机的名称：

```
$ psql -U <user> -h <host> -d rhpam7 -f ~/process-migration/ddl-scripts/postgres/postgresql-quartz-schema.sql
$ psql -U <user> -h <host> -d rhpam7 -f ~/process-migration/ddl-scripts/postgres/postgresql-pim-schema.sql
```

5. 将目录更改为 **process-migration** 目录。

6. 使用文本编辑器，使用以下内容创建 **servers.yaml** 配置文件，并保存在 **process-migration** 目录中。在本例中，将 **&lt ;user_name& gt ;** 和 **<password >** 替换为要登录到 **KieServer** 的凭证。

```
kieservers:
  - host: http://localhost:8080/kie-server/services/rest/server
    username: <user_name>
    password: <password>
```

7. 使用文本编辑器创建 **数据源.yaml** 配置文件，并保存在 **process-migration** 目录中。在本例中，将 **&lt ;user_name& gt ;** 和 **<password >** 替换为要登录到数据库的凭证：

```
quarkus:
  datasource:
    db-kind: postgresql
    jdbc:
      url: jdbc:postgresql://localhost:5432/rhpam7
    username: <user_name>
    password: <password>
```

8. 重建 **quarkus-run.jar** 文件，使其包含 **PostgreSQL** 驱动程序：

```
$ java -jar -Dquarkus.launch.rebuild=true -Dquarkus.datasource.db-kind=postgresql quarkus-app/quarkus-run.jar
```

这个命令的输出应类似以下示例：

```
INFO [io.qua.dep.QuarkusAugmentor] (main) Quarkus augmentation completed in 2657ms
```

9. 运行 **quarkus-app JAR** 文件：

```
$ java -jar -Dquarkus.http.port=8090 -Dquarkus.config.locations=servers.yaml,datasource.yaml quarkus-app/quarkus-run.jar
```

这个命令返回类似以下示例的输出：

```
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
```

```
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___ \_ \____/_/ |_/_/|_/_/|_|\____/___/
2022-03-11 13:04:18,261 INFO  [org.fly.cor.int.lic.VersionPrinter] (main) Flyway Community Edition 7.14.0 by Redgate
2022-03-11 13:04:18,262 INFO  [org.fly.cor.int.dat.bas.BaseDatabaseType] (main) Database: jdbc:postgresql://localhost:5432/rhpam7 (PostgreSQL 13.4)
2022-03-11 13:04:18,280 INFO  [org.fly.cor.int.com.DbMigrate] (main) Current version of schema "public": 1.0
2022-03-11 13:04:18,281 INFO  [org.fly.cor.int.com.DbMigrate] (main) Schema "public" is up to date. No migration necessary.
2022-03-11 13:04:18,601 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Freed 0 triggers from 'acquired' / 'blocked' state.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Recovering 0 jobs that were in-progress at the time of the last shut-down.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Recovery complete.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Removed 0 'complete' triggers.
2022-03-11 13:04:18,603 INFO  [org.qua.imp.jdb.JobStoreCMT] (main) Removed 0 stale fired job entries.
2022-03-11 13:04:18,624 INFO  [org.kie.ser.api.mar.MarshallerFactory] (main) Marshaller extensions init
2022-03-11 13:04:18,710 INFO  [org.kie.pro.ser.imp.KieServiceImpl] (main) Loaded kie server configuration for: org.kie.processmigration.model.config.KieServers$KieServer9579928Impl@4b6b5352
2022-03-11 13:04:18,715 INFO  [org.kie.pro.ser.RecoveryService] (main) Resuming ongoing migrations ...
2022-03-11 13:04:18,856 INFO  [io.quarkus] (main) process-migration-service 7.59.0.Final-redhat-00006 on JVM (powered by Quarkus 2.2.3.Final-redhat-00013) started in 1.443s. Listening on: http://0.0.0.0:8090
2022-03-11 13:04:18,857 INFO  [io.quarkus] (main) Profile prod activated.
2022-03-11 13:04:18,857 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, config-yaml, flyway, hibernate-orm, hibernate-orm-panache, jdbc-db2, jdbc-h2, jdbc-mariadb, jdbc-mssql, jdbc-mysql, jdbc-oracle, jdbc-postgresql, narayana-jta, quartz, resteasy, resteasy-jackson, scheduler, security, security-jdbc, security-ldap, security-properties-file, smallrye-context-propagation, smallrye-health]
```

10. *要访问进程实例迁移应用程序，请在 Web 浏览器中输入 **http://localhost:8090/**。*

11. *出现提示时，输入用户名 **admin** 和密码 **admin1!**。此时会出现 **Process Instance Migration** 控制台。*

32.2. 使用 KEYSTORE VAULT

*您可以使用 **Quarkiverse File Vault** 扩展来存储凭据作为密钥存储文件，并使用 文件 方法使用带有流程实例迁移(PIM)**Keystore Vault** 的密钥存储文件。有关 **Quarkiverse File Vault** 扩展的更多信息，请参阅 **Quarkiverse File Vault**。有关使用 **KeyStore Vault** 的更多信息，请参阅在 **GitHub** 上使用 **Keystore Vault**。如需有关凭证置备的更多信息，请参阅 **Quarkus** 文档中的 **Credentials Provider** 部分。*

注意

您只能对 PIM 配置使用数据库和 KIE 服务器相关凭证。

流程

1. 要在 PIM Keystore Vault 的新密钥存储文件中添加密码，请使用 keytool 命令。例如：

```
$ keytool -importpass -alias pimdb -keystore pimvault.p12 -storepass password -storetype
PKCS12
$ keytool -importpass -alias kieserver -keystore pimvault.p12 -storepass password -storetype
PKCS12
$ keytool -importpass -alias cert -keystore pimvault.p12 -storepass password -storetype
PKCS12
$ keytool -importpass -alias keystore -keystore pimvault.p12 -storepass password -storetype
PKCS12
$ keytool -importpass -alias truststore -keystore pimvault.p12 -storepass password -
storetype PKCS12
```

2. 配置 PIM Keystore Vault 以使用密钥存储文件。例如：

```
quarkus:
 file:
  vault:
   provider:
    pim:
     path: pimvault.p12
     secret: ${vault.storepassword} # This will be provided as a property
```

3. 配置您的应用，以使用 vault 中的凭据。例如：

```
quarkus:
 datasource:
  credentials-provider: quarkus.file.vault.provider.pim.pimdb
kieservers:
 - host: http://localhost:18080/kie-server/services/rest/server
  credentials-provider: quarkus.file.vault.provider.pim.kieserver
```

4. 要使用配置的凭据启动 PIM，将凭证指定为环境变量或系统属性。例如：

- 作为一个环境变量：

```
VAULT_STOREPASSWORD=mysecret java -jar quarkus-app/quarkus-run.jar
```

- 作为系统属性：

```
java -Dvault.storepassword=password -jar quarkus-app/quarkus-run.jar
```

32.3. 创建迁移计划

您可以在流程实例迁移(PIM)服务 Web UI 中定义两个不同的进程定义（称为迁移计划）之间的迁移。

先决条件

- 您已在备份的 Red Hat Process Automation Manager 开发环境中定义了进程。
- 进程实例迁移服务正在运行。

流程

1. 在网页浏览器中输入 http://localhost:8080。
2. 登录到 PIM 服务。
3. 在 Process Instance Migration 页面的右上角，从 KIE Service 列表中选择您要为其添加迁移计划的 KIE Service。
4. 点 Add Plan。此时会打开 Add Migration Plan 向导 窗口。
5. 在 Name 字段中输入迁移计划的名称。
6. 可选：在 Description 字段中输入迁移计划的描述。
7. 点击 Next。

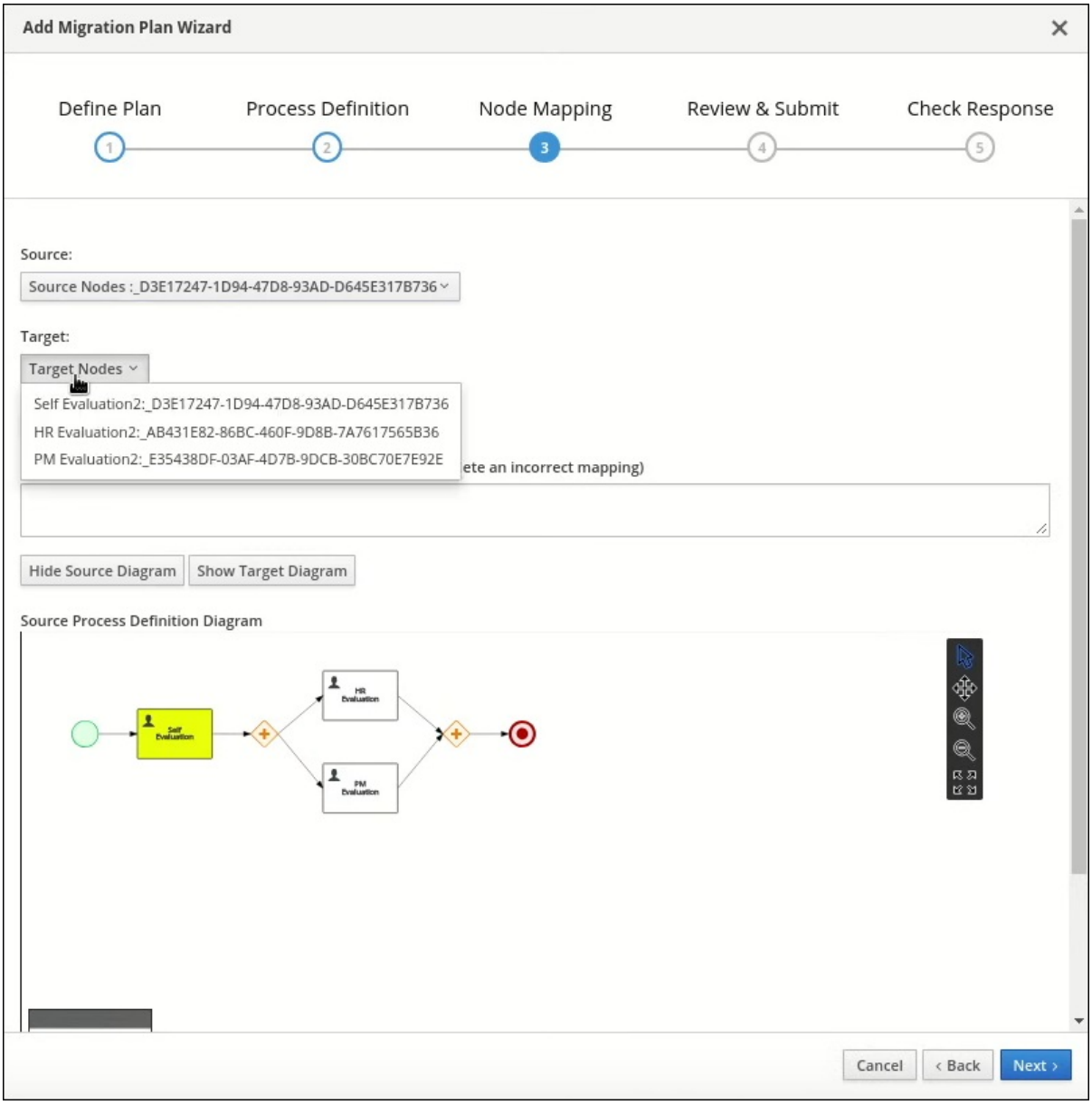
8. *在 Source ContainerID 字段中输入源容器 ID。*

9. *在 Source ProcessId 字段中输入源进程 ID。*

10. *点 Copy Source To Target。*

11. *在 Target ContainerID 字段中，更新目标容器 ID。*

12. *从后端点 Retrieve Definition 并点 Next。*



13. *在 Source Nodes 列表中，选择您要映射的源节点。*

14. *在 Target Nodes 列表中，选择您要映射的目标节点。*

15. *如果没有显示 Source Process Definition 图表 窗格，请单击 Show Source 图表。*

16. *如果没有显示 Target Process Definition 图表 窗格，请单击 Show Target Diagram。*

17. *可选： 要修改图表窗格中的视图，请执行以下任一任务：*

    - *要选择文本，请选择*
      
      图标。

    - *要 pan，选择*
      
      图标。

    - *要缩放，请选择*
      
      图标。

    - *要缩放，请选择*
      
      图标。

    - *要适合查看器，请选择*
      
      图标。

18. 点 Map 这两个节点。

19. 点击 Next。

20. 可选： 要将导出为 JSON 文件，请点击 Export。

21. 在 Review & Submit 选项卡中，检查计划并点击 Submit Plan。

22. 可选： 要将导出为 JSON 文件，请点击 Export。

23. 检查响应并点击 Close。

32.4. 编辑迁移计划

您可以在进程实例迁移(PIM)服务 Web UI 中编辑迁移计划。您可以修改迁移计划名称、描述、指定节点和处理实例。

先决条件

- 您已在备份的 Red Hat Process Automation Manager 开发环境中定义了进程。
- PIM 服务正在运行。

流程

1. 在网页浏览器中输入 http://localhost:8080。
2. 登录到 PIM 服务。
3. 在 Process Instance Migration 页面中，选择您要编辑的迁移计划行的 Edit Migration Plan 

图标。这会打开 Edit Migration Plan 窗口。

4. 在每个标签页中，修改您要更改的详情。

5. 点击 Next。

6. 可选： 要将导出为 JSON 文件，请点击 Export。

7. 在 Review & Submit 选项卡中，检查计划并点击 Submit Plan。

8. 可选： 要将导出为 JSON 文件，请点击 Export。

9. 检查响应并点击 Close。

32.5. 导出迁移计划

您可以使用进程实例迁移(PIM)服务 Web UI 将迁移计划导出为 JSON 文件。

先决条件

- 您已在备份的 Red Hat Process Automation Manager 开发环境中定义了进程。
- PIM 服务正在运行。

流程

1. 在网页浏览器中输入 http://localhost:8080。

2. 登录到 PIM 服务。

3. 在 Process Instance Migration 页面中，选择您要执行的迁移计划行中的 Export Migration

Plan  图标。这时将打开 Export Migration Plan 窗口。

4. 检查并单击 导出。

32.6. 执行迁移计划

您可以在进程实例迁移(PIM)服务 Web UI 中执行迁移计划。

先决条件

- 您已在备份的 Red Hat Process Automation Manager 开发环境中定义了进程。
- PIM 服务正在运行。

流程

1. 在网页浏览器中输入 http://localhost:8080。
2. 登录到 PIM 服务。
3. 在 Process Instance Migration 页面中，选择您要执行的迁移计划行中的 Execute Migration Plan  图标。Execute Migration Plan Wizard 窗口将打开。
4. 在 migration plan 表中，选中您要迁移的每个正在运行的进程实例行的复选框，然后点 Next。
5. 在 Callback URL 字段中，输入回调 URL。

6. 在运行 迁移的权利 中，执行以下任务之一：

   - 要立即执行迁移，现在选择。
   - 要调度迁移，请在文本字段中选择 Schedule，输入日期和时间，如 06/20/2019 10:00 PM。

7. 点击 Next。

8. 可选：要将导出为 JSON 文件，请点击 Export。

9. 点 Execute Plan。

10. 可选：要将导出为 JSON 文件，请点击 Export。

11. 检查响应并单击 关闭。

32.7. 删除迁移计划

您可以删除进程实例迁移(PIM)服务 Web UI 中的迁移计划。

先决条件

- 您已在备份的 Red Hat Process Automation Manager 开发环境中定义了进程。
- PIM 服务正在运行。

流程

1. 在网页浏览器中输入 http://localhost:8080。

2. *登录到 PIM 服务。*

3. *在 Process Instance Migration 页面中，选择您要删除的迁移计划行的 Delete 图标。此时会打开 Delete Migration Plan 窗口。*

4. *单击 Delete 以确认删除。*

部分 IV. 为问题单管理设计和构建案例

作为开发人员，您可以使用 Business Central 来配置用于案例管理的红帽流程自动化管理器资产。

案例管理与业务流程管理(BPM)不同。它更专注于整个案例中处理的实际数据，而不是关注完成目标所采取的步骤序列。案例数据是自动案例处理中最重要的信息，而业务上下文和决策过程是人类工作者的手中。

Red Hat Process Automation Manager 包括 Business Central 中的 IT_Orders 示例项目。本文档引用示例项目来解释案例管理概念并提供示例。

开始使用问题单管理 教程介绍了如何在 Business Central 中创建和测试新的 IT_Orders 项目。在查看本指南中的概念后，按照教程中的步骤操作，确保您能够成功创建、部署和测试您自己的问题单项目。

先决条件

- Red Hat JBoss Enterprise Application Platform 7.4 已安装。有关安装 Red Hat JBoss Enterprise Application Platform 7.4 的详情，请参考 Red Hat JBoss Enterprise Application Platform 7.4 安装指南。
- 安装了 Red Hat Process Automation Manager。有关安装 Red Hat Process Automation Manager 的详情，请参阅 规划 Red Hat Process Automation Manager 安装。
- Red Hat Process Automation Manager 正在运行，您可以使用 用户角色 登录 Business Central。有关用户和权限的信息，请参阅 规划 Red Hat Process Automation Manager 安装。
- The Showcase 应用已被部署。有关如何安装并登录到 Showcase 应用程序的详情，请参阅 使用 Showcase 应用程序进行问题单管理。

第 33 章 问题单管理

案例管理是业务流程管理(BPM)的扩展，可让您管理适应性业务流程。

BPM 是用于自动执行可重复性且具有共同模式的任务，并专注于优化流程。业务流程通常以明确明确定义的路径为实现业务目标而建模。这需要很多可预测性，通常基于大规模生产原则。但是，许多实际应用不能完全描述完成（包括所有可能的路径、偏差和例外）。在某些情况下，使用面向流程的方法可导致难以维护的复杂解决方案。

案例管理为非可预测、无法预测的流程提供了问题解决，而非针对日常预见性任务的 BPM 效率导向的方法。当进程无法预先预测时，它管理一次性情况。案例定义通常包含松散耦合的流程片段，这些片段可以直接连接或间接连接导致特定里程干和最终的业务合作伙伴，而进程能动态管理，以响应运行时发生的变化。

在 Red Hat Process Automation Manager 中，问题单管理包括以下核心流程引擎功能：

- 问题单文件实例
- 每个问题单运行时策略
- 问题单评论
- milestones
- 阶段
- 临时片段
- 动态任务和流程
- 问题单标识符（关联密钥）

- 问题单生命周期（关闭、重新打开、取消、销毁）

问题单定义始终是一个临时进程定义，不需要显式启动节点。该案例定义是业务用例的主要入口点。

进程定义是作为支持问题单的支持构造而成的，可以在问题单定义中按定义调用，也可以根据需要动态上线处理。案例定义定义了以下新对象：

- 活动（必需）
- case file（必需）
- milestones
- 角色
- 阶段

第 34 章 问题单管理模型和符号

您可以使用 Business Central 导入、查看和修改问题单管理模型和符号(CMMN)文件的内容。在编写项目时，您可以导入您的问题单管理模式，然后从 asset 列表中选择它以在标准 XML 编辑器中查看或修改它。

以下 CMMN 结构目前可用：

- 任务（human 任务、流程任务、决策任务、案例任务）
- Discretionary 任务（如上所示）
- 阶段
- milestones
- 问题单文件项
- Sentries（条目和退出）

不支持以下任务：

- 必填
- 重复
- 手动激活

对个别任务相比，Sentries 仅限于在输入和退出条件时对阶段和里程one的支持的条目标准。决策任务默认映射到 DMN 决策。不支持事件监听程序。

Red Hat Process Automation Manager 不提供 CMMN 的任何建模功能，它们只关注执行模型。

第 35 章 问题单文件

案例实例是案例定义的单一实例，封装业务上下文。所有问题单实例数据存储在 case 文件中，该文件可以被所有可能参与特定问题单实例的进程实例访问。每个问题单实例及其案例文件完全与另一个情况隔离。只有分配给所需问题单角色的用户才能访问 case 文件。

问题单文件用作整个案例实例的数据存储库时可使用问题单文件。它包含所有角色、数据对象、数据映射和其他数据。可将此问题单关闭并在以后附加同一问题单文件重新打开。案例实例可以随时关闭，不需要完成特定的解决方案。

该问题单文件还可以包含嵌入式文档、引用、cinder 附加、Web 链接和其他选项。

35.1. 配置问题单 ID 前缀

caseId 参数是一个字符串值，它是问题单实例的标识符。您可以在 Red Hat Process Automation Manager 设计器中配置 Case ID 前缀 来区分不同类型的情况。

以下流程使用 IT_Orders 示例项目来演示如何为特定业务需求创建唯一的问题单 ID 前缀。

先决条件

- IT_Orders 示例项目在 Business Central 中是开放的。

流程

1. 在 Business Central 中，转至 Menu → Design → Projects。如果已有项目，您可以点击 MySpace default 空间并从 Add Project 下拉菜单中选择 Try Samples 来访问示例。如果没有现有项目，请点击 Try samples。
2. 选择 IT_Orders 并点 确定。
3. 在 Assets 窗口中，点击 orderhardware 业务流程以打开设计人员。
4. 点击 canvas 和右上角的空空间，点 Properties  图标。

5. 向下滚动并展开 问题单管理。

6. 在 Case ID Prefix 字段中，输入 ID 值。ID 格式在内部被定义为 ID-XXXXXXXXXX，其中 XXXXXXXXXX 是为问题单实例提供唯一 ID 的生成数字。

   如果没有提供前缀，则默认前缀为 CASE，并带有以下标识符：

   CASE-0000000001

   CASE-0000000002

   CASE-0000000003

   您可以指定任何前缀。例如，如果您指定前缀 IT，则会生成以下标识符：

   IT-0000000001

   IT-0000000002

   IT-0000000003

图 35.1. case ID Prefix 字段

35.2. 配置问题单 ID 表达式

以下流程使用 IT_Orders 示例项目来演示如何设置 metadata 属性键来生成 caseId 的表达式。

先决条件

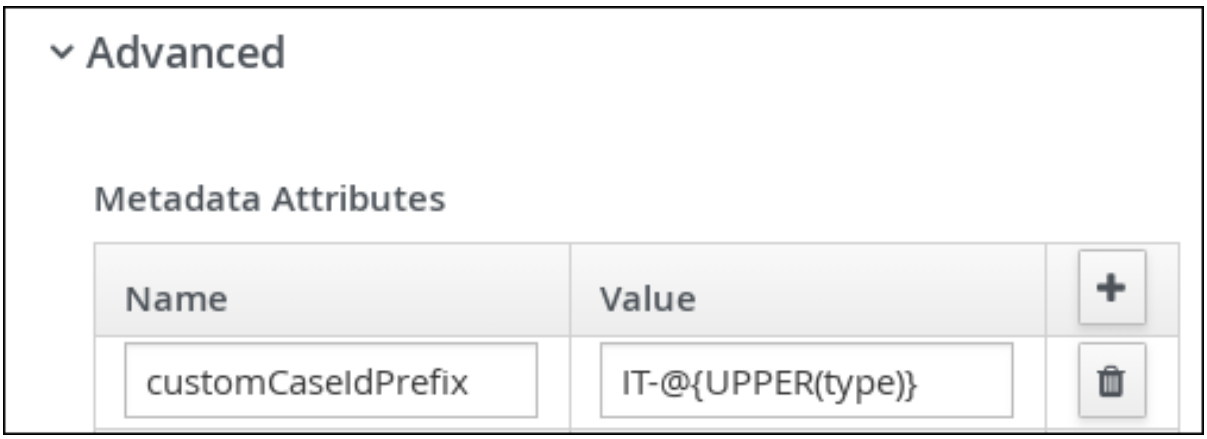
- IT_Orders 示例项目在 Business Central 中是开放的。

流程

1. 在 Business Central 中，转至 Menu → Design → Projects。如果已有项目，您可以点击 MySpace default 空间并从 Add Project 下拉菜单中选择 Try Samples 来访问示例。如果没有现有项目，请点击 Try samples。
2. 选择 IT_Orders 并点 确定。
3. 在 Assets 窗口中，点击 orderhardware 业务流程以打开设计人员。
4. 点击 canvas 和右上角的空空间，点 Properties 图标。
5. 展开 Advanced 菜单，以访问 Metadata Attributes 字段。
6. 为 customCaseIdPrefix 元数据属性指定以下功能之一：
   - LPAD: Left padding
   - RPAD ：合适的 padding
   - TRUNCATE: Truncate

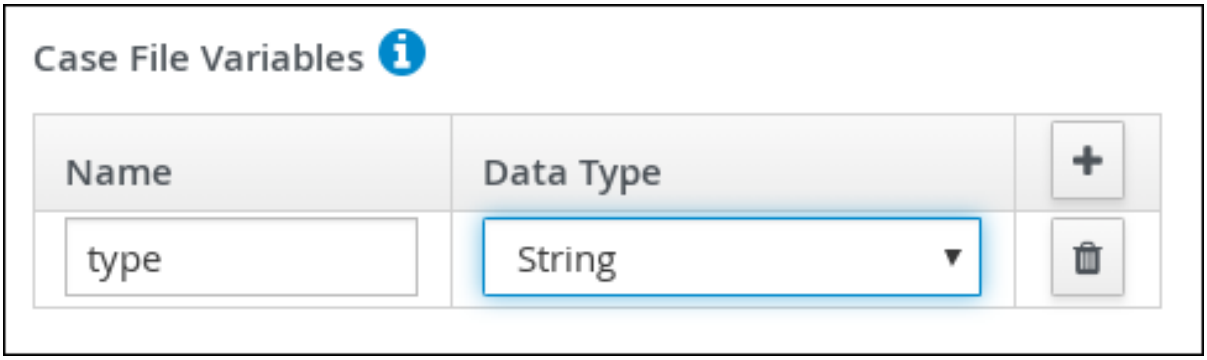
- *UPPER: Upper case*

图 35.2. 为 customCaseIdPrefix 元数据属性设置 UPPER 功能



在本例中，type 是 Case File Variables 字段中设置的变量，用户可以在运行时设置它的值 type1。UPPER 是一个预先构建的功能，用于大写变量，IT- 是静态前缀。结果是动态案例 ID，如 IT-TYPE1-0000000001、IT-TYPE1-0000000002 和 IT-TYPE1-0000000003。

图 35.3. 示例文件变量



如果 customCaseIdPrefixIsSequence case metadata 属性设为 false （默认值为 true），则问题单实例不会创建任何序列，并且 caseIdPrefix 表达式是问题单 ID。例如，如果基于社交安全号生成问题单 ID，则不需要特定的序列或实例标识符。

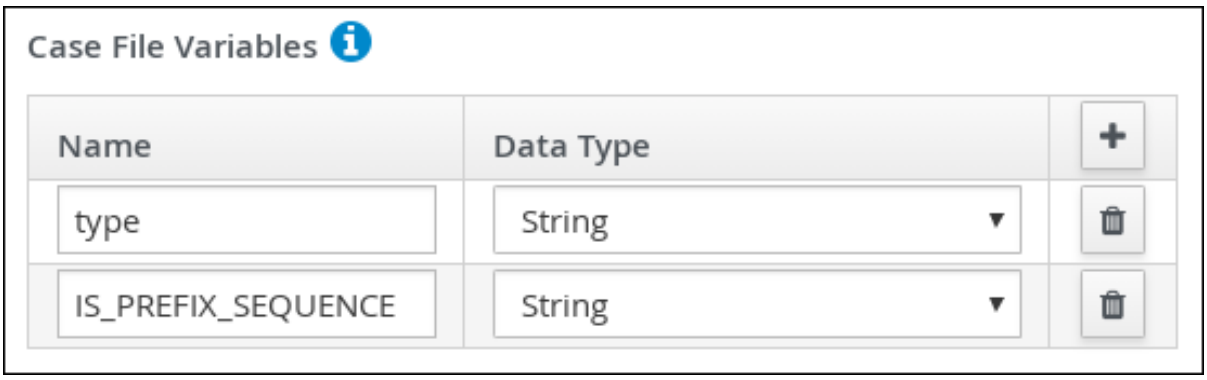
customCaseIdPrefixIsSequence metadata 属性已被添加，并将其设置为 false （默认值为 true）来禁用问题单 ID 的数字序列。如果用于自定义问题单 ID 的表达式已经包含一个问题单文件变量，用于表达唯一业务标识符而不是通用序列值。例如，如果基于社交安全号生成问题单 ID，则不需要特定的序列或实例标识符。例如，SO CIAL_SECURITY_NUMBER 也是作为示例文件变量声明的变量。

图 35.4. customCaseIdPrefixIsSequence 元数据属性

IS_PREFIX_SEQUENCE 问题单文件变量在运行时可选添加为标志，以禁用或启用问题单 ID 的序列生成。例如，无需为医疗保险覆盖范围创建序列后缀。对于多系列保险政策，公司可以将 IS_PREFIX_SEQUENCE 问题单变量设置为 true，以汇总系列的每个成员的序列数。

以静态方式使用 customCaseIdPrefixIsSequence 元数据属性 的结果，或使用 IS_PREFIX_SEQUENCE 问题单文件变量，并在运行时为 false 设置。

图 35.5. IS_PREFIX_SEQUENCE 问题单变量



第 36 章 子问题单

子案例提供了编写由其他情况组成的复杂情况的灵活性。这意味着，您可以将大型和复杂用例分成多个抽象层，甚至多个案例项目。这类似于将进程拆分为多个子进程。

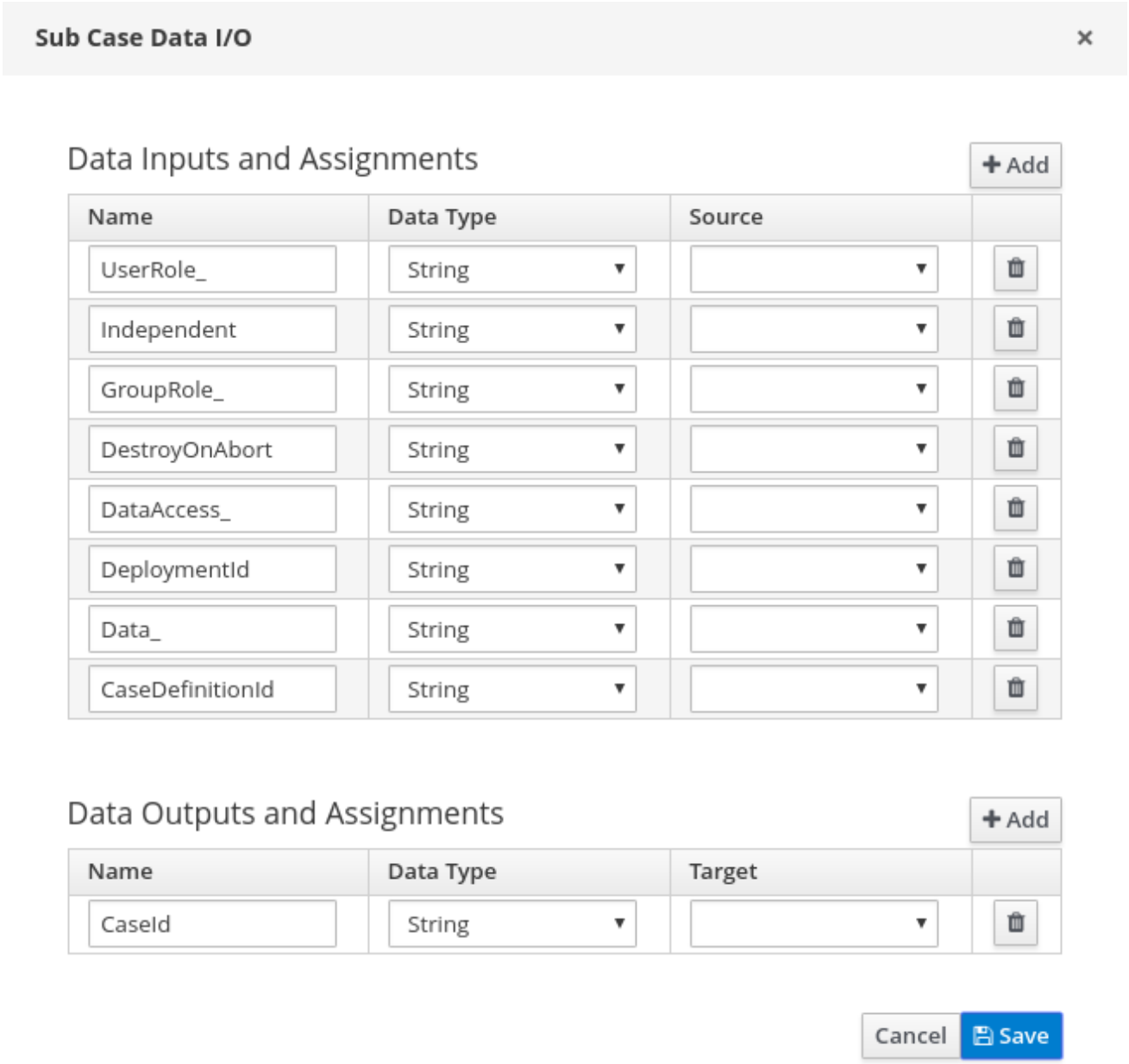
子用例是另一个案例定义，从另一个问题单实例或常规进程实例调用。它具有常规问题单实例的所有功能：

- 它有一个专用的问题单文件。
- 它与其他问题单实例隔离。
- 它有自己的一组问题单角色。
- 它有自己的问题单前缀。

您可以使用流程设计程序在您的问题单定义中添加子问题单。子问题单是您问题单项目中的一个情况，类似于流程中是否有子进程。也可以将子案例添加到常规的业务流程中。这样做可让您在进程实例内启动问题单。

有关在您的问题单定义中添加子问题单的更多信息，请参阅开始使用问题单管理。

Sub Case Data I/O 窗口支持以下输入参数，供您配置和启动子问题单：



独立

可选指示指示进程引擎是否独立于问题单实例。如果是独立，则主问题单实例不会等待其完成。此属性的值默认为 false。

GroupRole_XXX

可选的组到 case 角色映射。属于此例实例的角色名称可以在此处引用，即主案例的参与者可以映射到子案例的参与者。这意味着，分配给主问题单的组会自动分配给子案例，其中 XXX 是角色名称，而 属性的值则是组角色分配的值。

DataAccess_XXX

XXX 是数据项目的名称以及属性的值是访问限制的可选数据访问限制。

DestroyOnAbort

可选指示指示进程引擎在子问题单活动被中止时是否取消或销毁子问题单。默认值为 true。

UserRole_XXX

可选用户到案例角色映射。您可以在此处引用问题单实例角色名称，即主问题单的所有者可以映射到子问题单的所有者。分配给主情况的人员会自动分配给子案例，其中 XXX 是角色名称，而 属性的值则是用户角色分配的值。

Data_XXX

从此情况实例或业务流程到子案例的可选数据映射，其中 XXX 是目标中数据的名称。这个参数可以根据需要提供多次。

DeploymentId

可选部署 ID（或 KIE 服务器上下文中的容器 ID），用于指示目标问题单定义所在的位置。

CaseDefinitionId

要启动的必要问题单定义 ID。

CaseId

子用例的问题单实例 ID 启动后。

第 37 章 临时和动态任务

您可以使用案例管理来执行临时任务，而不是在严格的端到端进程后执行。您还可以在运行时动态添加任务到问题单中。

临时任务在 case 建模阶段定义。未配置为 AdHoc Autostart 的临时任务是可选的，且情形中可能不会使用。因此，它们必须由信号事件或 Java API 触发。

动态任务在案例执行中定义，在案例定义模型中不存在。动态任务可以满足在这种情况下出现的具体需求。它们可添加至这种情况，并在任何时候使用问题单应用程序进行操作，如 Red Hat Process Automation Manager Showcase 应用程序所示。动态任务也可由 Java 和远程 API 调用添加。

动态任务可以是用户或服务活动，而临时任务可以是任何类型的任务。有关任务类型的更多信息，请参阅使用 BPMN 模型设计 业务流程中的"BPMN 2 任务"。

动态进程是来自 case 项目的任何可重复使用的子进程。

在节点的 AdHoc Autostart 属性中配置没有传入连接的临时节点，并在实例启动时自动触发。

临时任务是在问题单定义中配置的可选任务。由于它们是临时的，因此它们必须以某种方式触发，通常是通过信号事件或 Java API 调用来触发。

第 38 章 使用 KIE 服务器 REST API 将动态任务和流程添加到示例中

您可以在运行时添加动态任务和流程，以解决问题单生命周期中可能出现的未预见的更改。特定定义中没有定义动态活动，因此无法向明确的临时任务或进程发送信号。

您可以在问题单中添加以下动态操作：

- 用户任务
- 服务任务（任何作为工作项目实施的类型）
- 可重复使用的子进程

动态用户和服务任务添加到问题单实例中，并立即执行。根据动态任务的性质，它可能开始并等待完成（用户任务）或直接在执行后完成（服务任务）。对于动态子进程，进程引擎需要包含该动态进程的进程定义的 KJAR，以便根据其 ID 查找其 ID 并执行它。此子进程属于这种情况，且有权访问 case 文件中的所有数据。

您可以使用 Swagger REST API 应用程序来创建动态任务和子进程。

先决条件

- 您已登陆到 Business Central，一个问题单实例已使用 Showcase 应用程序启动。有关使用 Showcase 的更多信息，请参阅使用 Showcase 应用程序进行问题单管理。

流程

1. 在网页浏览器中，打开以下 URL：

   http://localhost:8080/kie-server/docs

2. 在 问题单实例 :: Case Management 下打开可用端点的列表。

3. 找到用于创建动态活动的 POST 方法端点。

   POST /server/containers/{id}/cases/instances/{caseId}/tasks

   向问题单实例添加动态任务（根据有效负载影响的用户或服务）。

   POST /server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/tasks

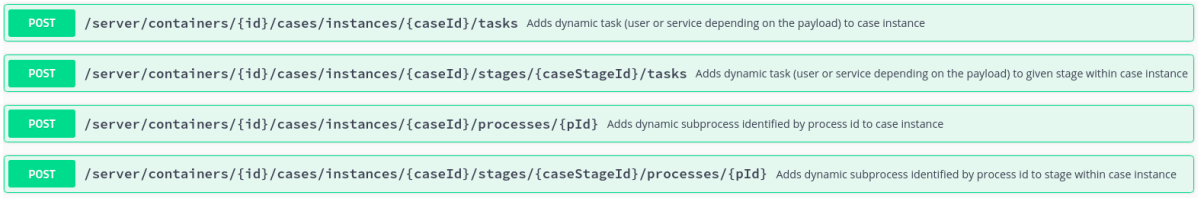
   添加动态任务（根据有效负载的用户或服务）到 case 实例中的特定阶段。

   POST /server/containers/{id}/cases/instances/{caseId}/processes/{pId}

   将进程 ID 标识的动态子进程添加到问题单实例。

   POST /server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/processes/{pId}

   添加由进程 ID 标识的动态子进程到问题单实例中的 stage。



4. 要打开 Swagger UI，请点击创建动态任务或进程所需的 REST 端点。

5. 点 Try it out 并输入创建动态活动所需的参数和正文。

6. 单击 Execute，以使用 REST API 创建动态任务或子进程。

38.1. 使用 KIE 服务器 REST API 创建动态用户任务

您可以使用 REST API 在运行时创建动态用户任务。要创建动态用户任务，您必须提供以下信息：

- 任务名称
- 任务主题（可选，但推荐使用）
- 执行或组（或两者）
- 输入数据

使用以下步骤，使用 Swagger REST API 工具为 Business Central 中提供的 IT_Orders 示例项目创建一个动态用户任务。同一端点可用于 REST API，而没有 Swagger。

先决条件

- 您已使用 Showcase 应用登录 Business Central 和 IT 订单案例实例。有关使用 Showcase 的更多信息，请参阅使用 Showcase 应用程序进行问题单管理。

流程

1. 在网页浏览器中，打开以下 URL：

   http://localhost:8080/kie-server/docs.

2. 在 问题单实例 :: Case Management 下打开可用端点的列表。

3. 点击以下 POST 方法端点打开详情：

   /server/containers/{id}/cases/instances/{caseId}/tasks

4. 点 Try it out，然后输入参数：

表 38.1. 参数

Name	描述
id	**itorders**
caseId	**IT-0000000001**

请求正文

```
{
 "name" : "RequestManagerApproval",
 "data" : {
   "reason" : "Fixed hardware spec",
   "caseFile_hwSpec" : "#{caseFile_hwSpec}"
 },
 "subject" : "Ask for manager approval again",
 "actors" : "manager",
 "groups" : ""
}
```

5. *在 Swagger 应用程序中，点 Execute 创建动态任务。*

此流程创建与案例 IT-000000001 相关的新用户任务。该任务分配到分配给 经理 案例角色的人员。此任务有两个输入变量：

- ***reason***
- ***caseFile_hwSpec: 定义为表达式，允许运行进程或问题单数据的运行时捕获。***

有些任务包含一个表单，为任务提供用户友好的 UI，您可以通过任务名称查找。在 IT Orders 案例中，RequestManagerApproval 任务在其 KJAR 中包括 RequestManagerApproval-taskform.form。

创建之后，该任务将显示在 Business Central 中的 assignee 的 Task Inbox 中。

38.2. 使用 KIE 服务器 REST API 创建动态服务任务

服务任务通常比用户任务复杂，虽然它们可能需要更多的数据才能正确执行。服务任务需要以下信息：

- 名称 ：活动的名称
- nodeType ：用于查找工作项目处理程序的节点类型
- 数据 ：用于正确处理执行的数据映射

在问题单运行时，您可以创建一个使用与用户任务相同的端点的动态服务任务，但使用不同的正文有效负载。

使用 Swagger REST API 通过以下步骤为 Business Central 中提供的 IT_Orders 示例项目创建一个动态服务任务。您可以在不使用 Swagger 的情况下为 REST API 使用同一个端点。

先决条件

- 您已使用 Showcase 应用登录 Business Central 和 IT 订单案例实例。有关使用 Showcase 的更多信息，请参阅使用 Showcase 应用程序进行问题单管理。

流程

1. 在网页浏览器中，打开以下 URL：

   http://localhost:8080/kie-server/docs

2. 在 问题单实例 :: Case Management 下打开可用端点的列表。

3. 点击以下 POST 方法端点打开详情：

   /server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/tasks

4. 点 Try it out，然后输入以下参数：

表 38.2. 参数

Name	描述
id	itorders
caseId	IT-0000000001

请求正文

```
{
 "name" : "InvokeService",
 "data" : {
   "Parameter" : "Fixed hardware spec",
   "Interface" : "org.jbpm.demo.itorders.services.ITOrderService",
   "Operation" : "printMessage",
   "ParameterType" : "java.lang.String"
 },
 "nodeType" : "Service Task"
}
```

5. 在 Swagger 应用程序中，点 Execute 创建动态任务。

在本例中，执行基于 Java 的服务。它由一个带有公共类 org.jbpm.demo.itorders.services.ITOrderService 以及一个 String 参数的公共 printMessage 方法组成。执行后，参数值将传递到执行的方法。

创建服务任务的数字、名称和其他数据类型取决于服务任务的实施。在提供的示例中，使用 org.jbpm.process.workitem.bpmn2.ServiceTaskHandler 处理程序。

注意

对于任何自定义服务任务，请确保在 Work Item Handlers 部分中的部署描述符中注册处理程序，其中 name 与用于创建动态服务任务的 nodeType 相同。有关注册部署描述符的更多信息，请参阅 Red Hat Process Automation Manager 中开发流程服务。

38.3. 使用 KIE 服务器 REST API 创建动态子进程

在创建动态子进程时，只会提供可选数据。在创建动态任务时没有特殊参数。

下面的步骤描述了如何使用 Swagger REST API 为 Business Central 中提供的 IT_Orders 示例项目创建一个动态的子进程任务。同一端点可用于 REST API，而没有 Swagger。

先决条件

- 您已使用 Showcase 应用登录 Business Central 和 IT 订单案例实例。有关使用 Showcase 的更多信息，请参阅使用 Showcase 应用程序进行问题单管理。

流程

1. 在网页浏览器中，打开以下 URL：

   http://localhost:8080/kie-server/docs.

2. 在 问题单实例 :: Case Management 下打开可用端点的列表。

3. 点击以下 POST 方法端点打开详情：

   /server/containers/{id}/cases/instances/{caseId}/processes/{pId}

4. 点 Try it out 并输入以下参数：

   表 38.3. 参数

Name	描述
id	itorders
caseId	IT-0000000001
pId	itorders-data.place-order

pId 是要创建的子进程的进程 ID。

请求正文

```
{
 "placedOrder" : "Manually"
}
```

5. *在 Swagger 应用程序中，单击 Execute 以启动动态子进程。*

在这个示例中，在 IT 顺序中，使用问题单 ID IT-0000000001 开始了订购 的子流程。您可以在 Menu → Manage → Process Instances 下的 Business Central 中看到此过程。

如果描述的示例已正确执行，那么 place-order 进程会出现在进程实例列表中。打开进程的详细信息，请注意，进程的关联键包括 IT 顺序案例实例 ID，而 Process Variables 列表包括变量 placedOrder 及 手动 的值，如 REST API 正文中所述。

第 39 章 注释

如果管理，注释有助于在问题单实例中进行协作，并允许案例 worker 轻松相互通信以交换信息。

注释与问题单实例绑定。案例实例是 case 文件的一部分，因此您可以使用注释对实例执行操作。基于文本的注释可以设置完整的操作，类似于 CRUD（创建、读取、更新和删除）。

第 40 章 问题单角色

case roles 为用户参与案例处理提供额外的抽象层。在管理时，角色、用户和组用于不同的目的。

角色

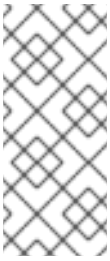
角色驱动问题单实例的授权，用于用户活动分配。用户可以或一个或多个组分配给所有者角色。所有者是谁是该案例。作为案例定义的一部分，角色不限于一组人或组。使用 roles 指定任务分配，而不是将特定的用户或组分配到任务分配，以确保问题单保持动态。

组

组是能够执行特定任务或具有一组指定职责的用户集合。您可以将任何数量的人员分配给一个组，并将任何组分配到某个角色。您可以随时添加或更改组的成员。不要将组硬编码到特定任务。

用户

用户是个人，当您将某个角色分配角色或添加到组中时，可以为他们授予特定的任务。



注意

不要在进程引擎或 KIE Server 中创建一个名为 unknown 的用户。未知 用户帐户是具有超级用户访问权限的保留系统名称。当用户没有登录时，未知 用户帐户执行与 SLA 违反监听程序相关的任务。

以下示例演示了如何使用以下信息将前面的问题单管理概念应用到热保留中：

- 角色 ：guest
- Group:Receptionist,Maid
- 用户:Marilyn

Guest 角色分配会影响关联案例的具体工作，适用于所有案例实例。每个问题单实例都有自己的角色分配。可分配给角色的用户或组数量受案例 卡限制，它在 流程设计和问题单定义中在角色创建期间设置。例如，热预订案例只有一个客户机，而 IT_Orders 示例项目有两个 IT 硬件供应商。

在定义角色时，请确保角色不硬编码到一组人或组，作为问题单定义的一部分，并且每个案例实例都有不同。这就是为什么角色分配很重要。

在问题单启动时或任何时候都可以分配或移除角色分配。虽然角色是可选的，但若使用角色来维护组织工作流。

重要

始终将角色用于任务分配，而不是实际的用户或组名称。这样可保证根据需要尽快进行问题单和用户或组分配。

角色分配到用户或组，并授权在实例启动时执行任务。

40.1. 创建问题单角色

在设计流程设计器时，您可以在问题单定义中创建和定义问题单角色。在案例定义级别上配置了条件角色，使其与处理问题单实例涉及的执行者分开。可将角色分配给用户任务，在整个问题单生命周期内作为联系人引用分配，但它们不会作为特定的用户或用户组来定义。

问题单实例包括实际处理问题单工作的个人。在启动新问题单实例时分配角色。为保持用例灵活，您可以在运行时修改问题单角色分配，虽然这样做不会影响基于之前角色分配创建的任务。分配到某一角色的ctor 非常灵活，但角色本身对于每种情况仍相同。

先决条件

- Business Central 中已存在具有问题单定义的问题单项目。
- 问题单定义资产在流程设计器中被打开。

流程

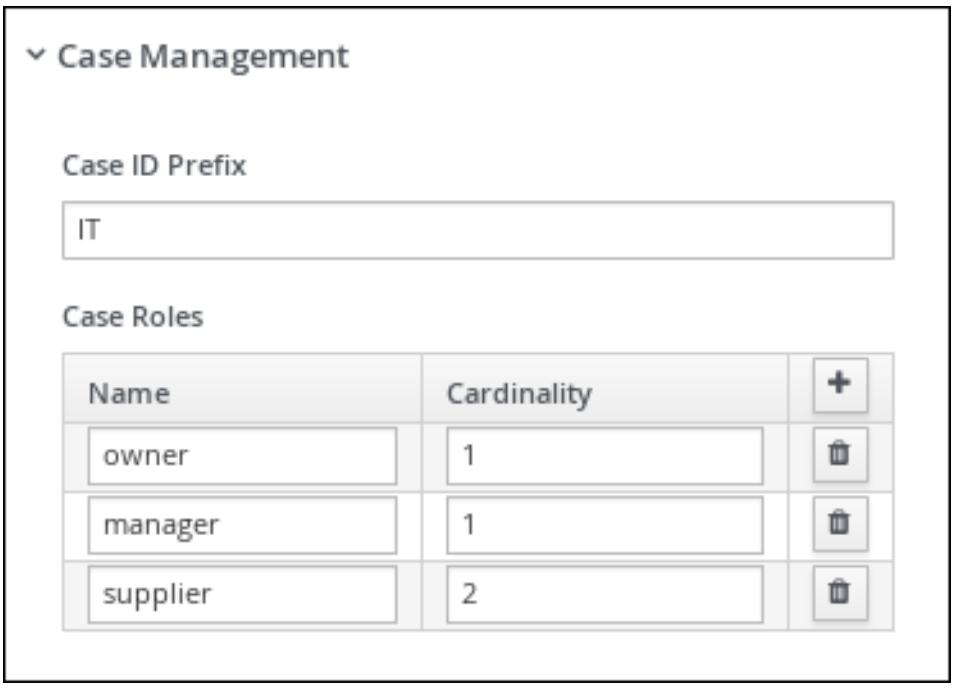
1. 要定义问题单中涉及的角色，请点击编辑器 canvas 中的空空间，然后点击  打开 Properties 菜单。

2.

扩展 问题单管理 以添加问题单角色。

case 角色需要角色的名称和一个问题单卡。case cardinality 是在任何情况下实例分配给该角色的执行者数量。例如，IT_Orders 示例案例管理项目包括以下角色：

图 40.1. ITOrders Case Roles



在这个示例中，您可以将一个ctor（用户或组）分配为问题单 所有者，仅分配一个 manager 角色。供应商 角色可以分配两个活动者。根据具体情况，您可以根据角色配置的问题单卡性将任何数量的执行器分配给特定的角色。

40.2. 角色授权

在使用 Showcase 应用或 REST API 启动新实例时，角色有权执行特定的案例管理任务。

使用以下步骤使用 REST API 启动新的 IT 顺序案例。

先决条件

- IT_Orders 示例项目已导入在 Business Central 中并部署到 KIE 服务器。

流程

1. 使用以下端点创建 POST REST API 调用：

   http://host:port/kie-server/services/rest/server/containers/itorders/cases/itorders.orderhardware/instances

   - 它顺序 ：部署到 KIE 服务器的容器别名。
   - itorders.orderhardware ：问题单定义的名称。

2. 在请求正文中提供以下角色配置：

```
{
  "case-data" : {  },
  "case-user-assignments" : {
    "owner" : "cami",
    "manager" : "cami"
  },
  "case-group-assignments" : {
    "supplier" : "IT"
  }
}
```

   这会启动带有定义的角色的新案例，以及自动启动活动，这些活动已启动并准备好继续工作。其中的两个角色为用户分配（所有者和 管理器），第三个角色是组分配（供应商）。

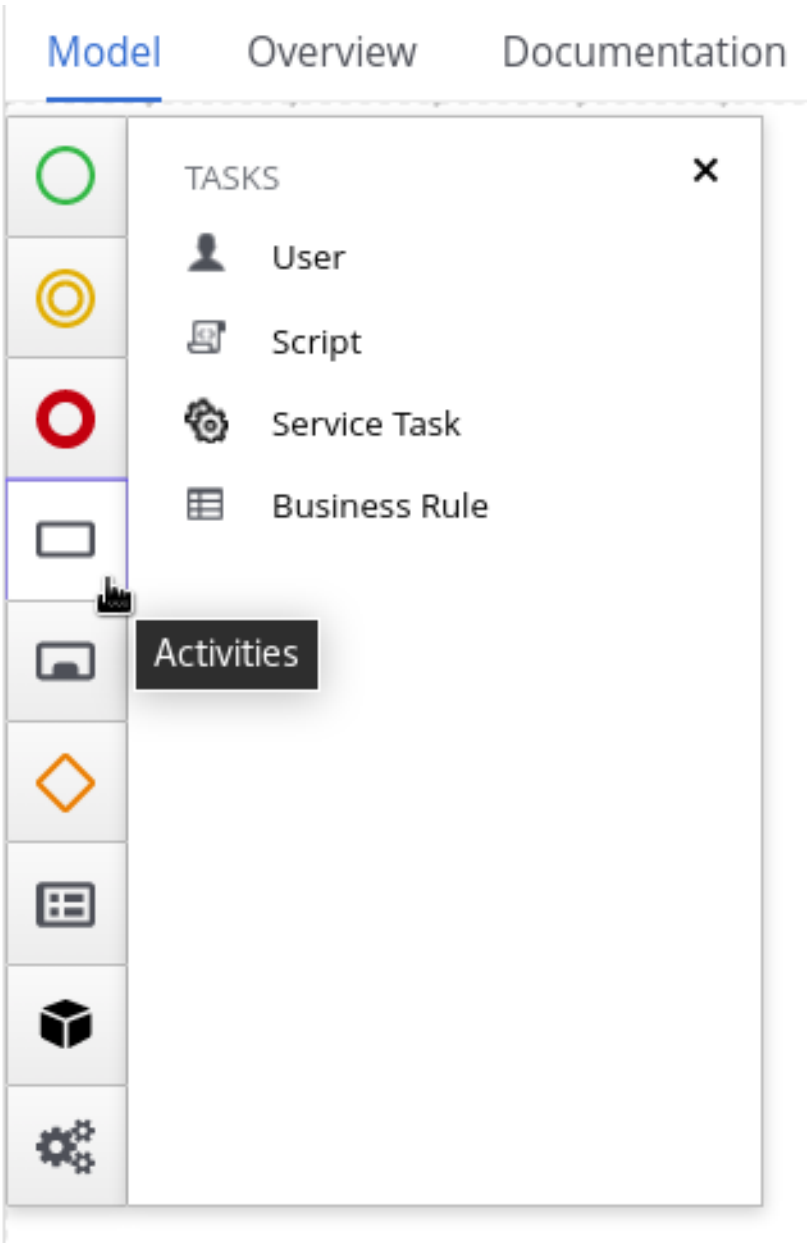
   在案例实例启动成功后，问题单实例会返回 IT-0000000001 问题单 ID。

有关如何使用 Showcase 应用启动新问题单实例的信息，请参阅使用 Showcase 应用程序进行问题单管理。

40.3. 为角色分配任务

案例管理流程需要尽可能灵活，以适应运行时可能会动态发生的更改。这包括更改新案例实例的用户分配，或用于活动。因此，请确保在问题单定义中没有将代码角色硬编码到单个用户或组中。取而代之，可以在条件定义中在任务节点上定义角色分配，以及在创建时分配给角色的用户或组。

Red Hat Process Automation Manager 包含用于简化业务流程创建预定义节点类型。预定义的节点面板位于图表编辑器的左侧。



先决条件

- *创建问题单定义时使用了问题单定义级别配置了条件角色。有关创建问题单角色的更多信息，请参阅 第 40.1 节 “创建问题单角色”。*

流程

1. *在设计器面板中打开 Activities 菜单，并将您要添加到问题单定义的用户或服务任务拖到进程设计器 Canvas 中。*

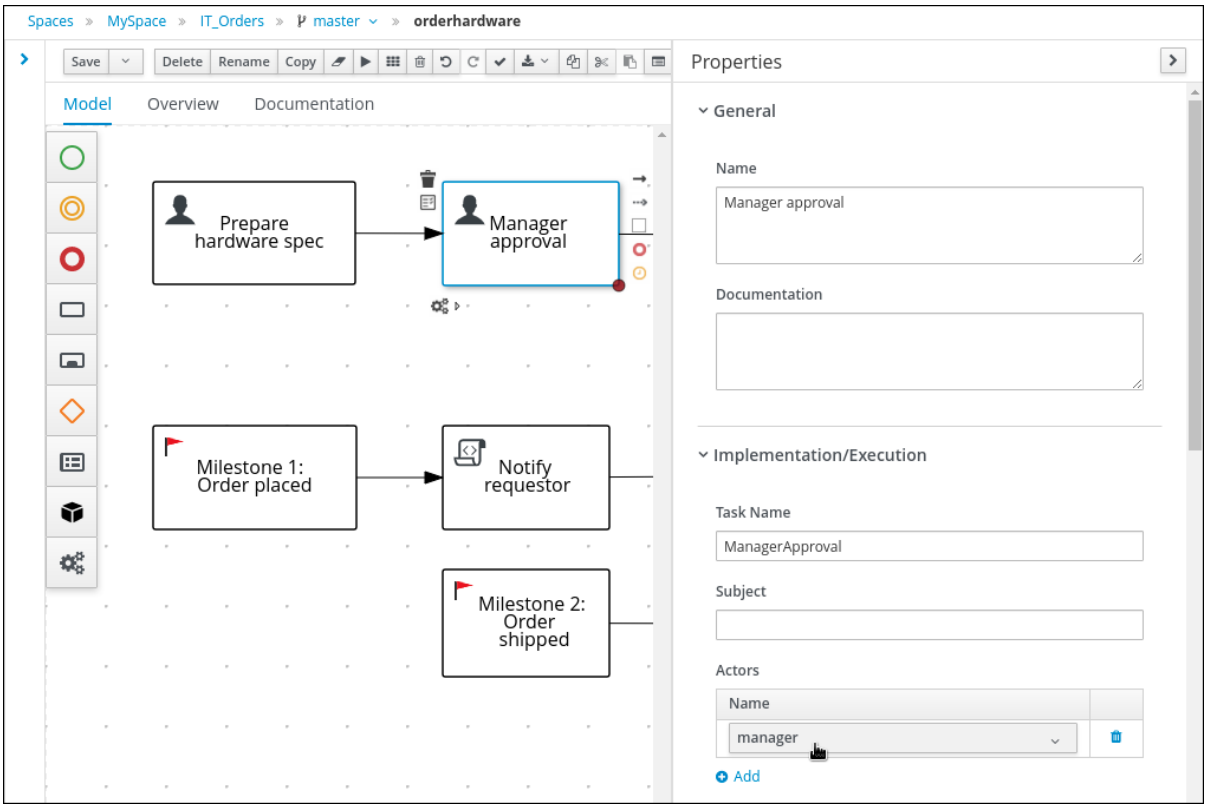
2.

选择任务节点后，点

在设计程序右侧的 Properties 面板。

3. 展开 Implementation/Execution，单击 Actors 属性下的 Add，然后选择 or type 将为其分配任务的角色的名称。您可以以相同的方式使用 Groups 属性来分组分配。

例如，在 IT_Orders 示例项目中，Manager 批准 用户任务被分配给 manager 角色：



在本例中，在 Prepare hardware spec user 任务完成后，分配给 manager 角色的用户将在 Business Central 中接收其 Task Inbox 中的 管理器批准 任务。

在运行案例时，可以更改分配给该角色的用户，但任务本身将继续具有相同的角色分配。例如，最初分配给 manager 角色的人可能需要花时间来关闭（例如，如果他们变为 ill），或者可能会意外离开公司。要在情况下响应这一变化，您可以编辑 manager 角色分配，以便其他人分配与该角色关联的任务。

有关如何在运行时更改角色分配的详情，请参考 第 40.4 节 “使用 Showcase 在运行时修改问题单角色分配” 或 第 40.5 节 “使用 REST API 修改运行时的大小写角色分配”。

40.4. 使用 SHOWCASE 在运行时修改问题单角色分配

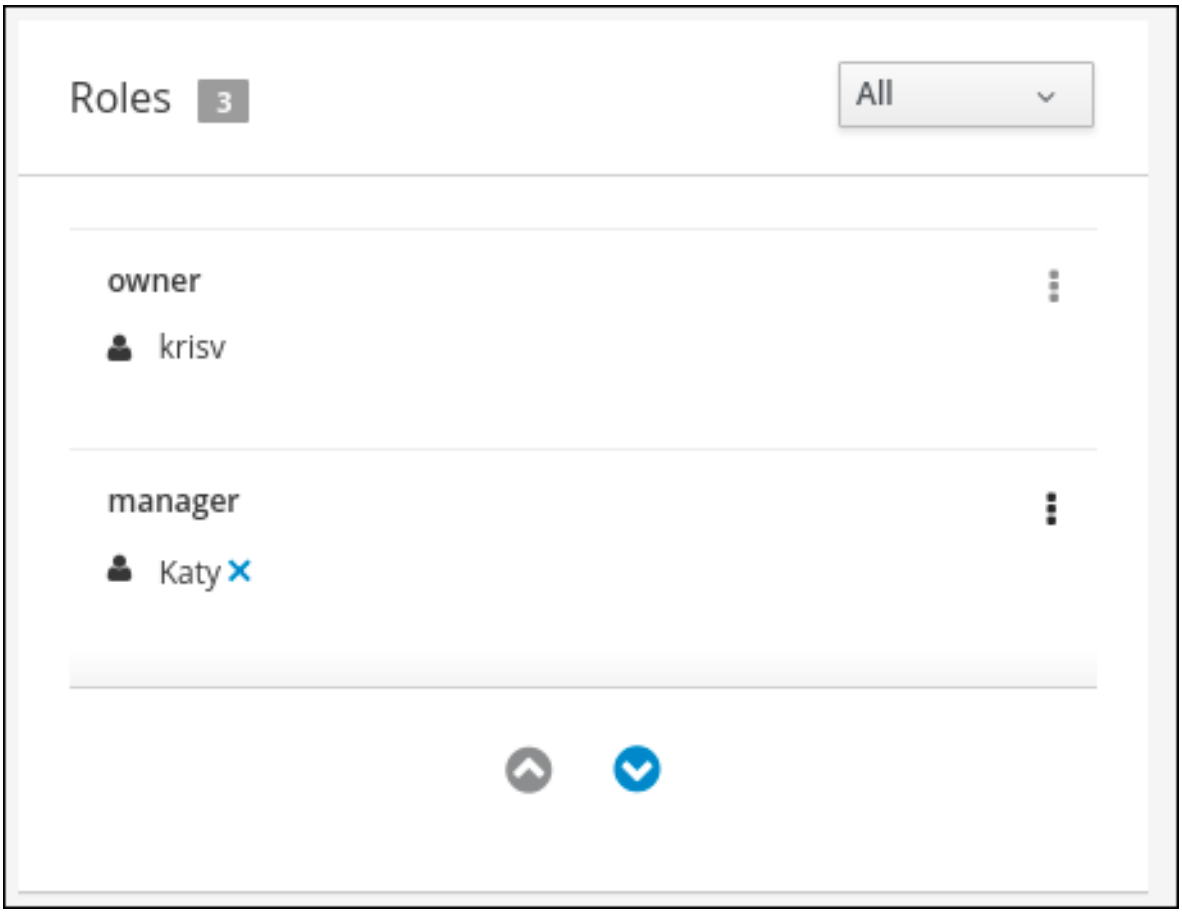
您可以使用 Showcase 应用在条件运行时更改问题单实例角色分配。在问题单定义中定义了角色，并在问题单生命周期中分配给任务。角色无法在运行时改变，因为它们是预定义的，但您可以更改分配给角色的行为，以更改负责执行案例任务的人员。

先决条件

- 具有用户或组的活跃问题单实例已分配给至少一个问题单角色。

流程

1. 在 Showcase 应用程序中，点您要在 Case 列表中 处理的情况来打开问题单概述。
2. 在页面右下角的 Roles 框中找到您要更改的角色分配。



3. 要从角色分配中删除单个用户或组，请点击分配旁边的

。在确认窗口中，点击 Remove 从角色中删除用户或组。
4. 要从角色中删除所有角色分配，请点击角色旁边的

并选择 **Remove all assignments** 选项。在确认窗口中，点 **Remove** 从角色中删除所有用户和组分配。

5. 要将角色分配从一个用户或组改为另一个用户或组，请点击角色旁边的

并选择 **Edit** 选项。

6. 在 **Edit role assignment** 窗口中，删除您要从角色分配中删除的 **assignee** 的名称。在 **Group** 字段中键入您要分配给角色的用户名称。

编辑角色分配时，必须至少分配一个用户或组。

7. 单击 **Assign** 以完成角色分配。

40.5. 使用 REST API 修改运行时的大小写角色分配

您可以使用 **REST API** 或 **Swagger** 应用在运行时更改问题单实例角色分配。在案例定义中定义角色，并在问题单生命周期中分配到任务。角色无法在运行时改变，因为它们是预定义的，但您可以更改分配给角色的行为，以更改负责执行案例任务的人员。

以下流程包含基于 **IT_Orders** 示例项目的示例。您可以使用 **Swagger** 应用或任何其他 **REST API** 客户端或 **Curl** 中的相同的 **REST API** 端点。

先决条件

- **IT** 订单案例实例已使用 所有者、经理 和供应商 角色开始。

流程

1. 使用以下端点上的 **GET** 请求检索当前角色分配列表：

http://localhost:8080/kie-server/services/rest/server/containers/{ID}/cases/instances/{caseId}/roles

表 40.1. 参数

Name	描述
id	itorders
caseId	IT-0000000001

这会返回以下响应：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<case-role-assignment-list>
    <role-assignments>
        <name>owner</name>
        <users>Aimee</users>
    </role-assignments>
    <role-assignments>
        <name>manager</name>
        <users>Katy</users>
    </role-assignments>
    <role-assignments>
        <name>supplier</name>
        <groups>Lenovo</groups>
    </role-assignments>
</case-role-assignment-list>
```

2. *要更改分配给 manager 角色的用户，您必须首先使用 DELETE 从用户 Katy 中删除角色分配。*

/server/containers/{id}/cases/instances/{caseId}/roles/{caseRoleName}

在 Swagger 客户端请求中包含以下信息：

表 40.2. 参数

Name	描述
id	itorders
caseId	IT-0000000001
caseRoleName	Manager
user	Katy

点 Execute。

3. 再次从第一步执行 GET 请求，以检查 manager 角色不再分配用户：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<case-role-assignment-list>
    <role-assignments>
        <name>owner</name>
        <users>Aimee</users>
    </role-assignments>
    <role-assignments>
        <name>manager</name>
    </role-assignments>
    <role-assignments>
        <name>supplier</name>
        <groups>Lenovo</groups>
    </role-assignments>
</case-role-assignment-list>
```

4. 在以下端点上使用 PUT 请求将 Cami 用户分配给 manager 角色：

/server/containers/{id}/cases/instances/{caseId}/roles/{caseRoleName}

在 Swagger 客户端请求中包含以下信息：

表 40.3. 参数

Name	描述
id	itorders
caseId	IT-0000000001
caseRoleName	Manager
user	Cami

点 Execute。

5. 再次从第一步执行 GET 请求，以检查 manager 角色现在是否已分配给 Cami ：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<case-role-assignment-list>
    <role-assignments>
        <name>owner</name>
        <users>Aimee</users>
    </role-assignments>
    <role-assignments>
        <name>manager</name>
        <users>Cami</users>
    </role-assignments>
    <role-assignments>
        <name>supplier</name>
        <groups>Lenovo</groups>
    </role-assignments>
</case-role-assignment-list>
```

第 41 章 阶段

案例管理阶段是一系列任务。阶段是一个临时的子进程，可以使用进程设计器定义，可能包含其他问题单管理节点，如 milestone。当阶段或多个阶段完成时，也可以将 milestone 配置为 completed。因此，一个 milestone 可以通过完成阶段来激活或实现，一个阶段可能会包含 milestone 或多个 milestone。

例如，在病人分类的情况下，第一个阶段可能包括观察，请注意任何明显的物理症状或来自人看重的物理症状或描述，然后是测试的第二阶段，以及用于诊断和处理的第三个阶段。

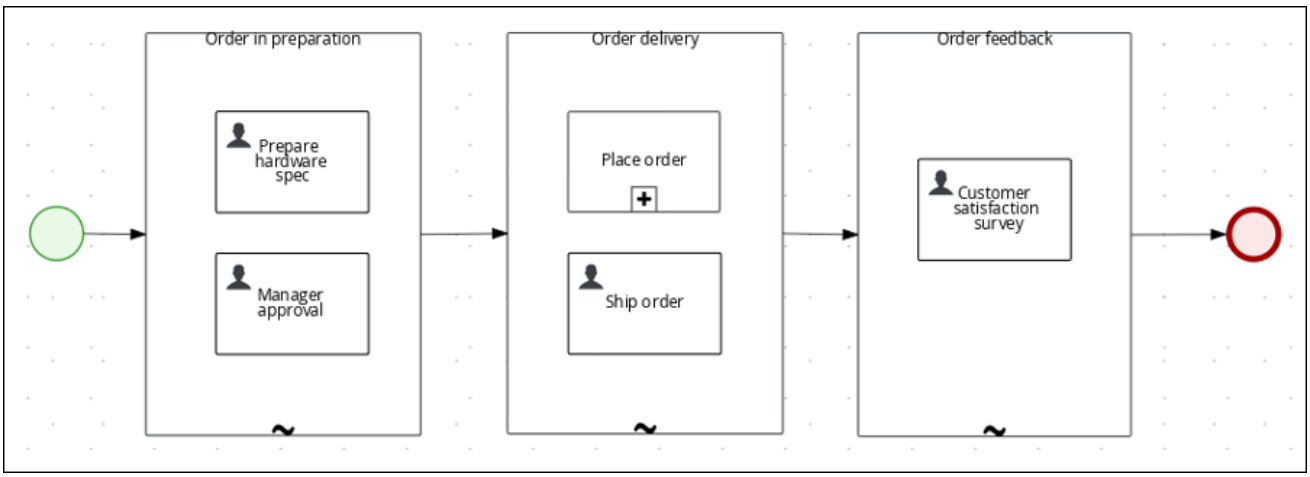
完成阶段有三种方法：

- 按完成条件。
- 按终端结束事件。
- 通过将 Completion Condition 设置为 autocomplete，这在阶段没有活动任务时 自动完成该阶段。

41.1. 定义阶段

可使用流程设计器在 BPMN2 中建模阶段。阶段是以明确定义活动的方式对相关任务进行分组的方式（如果已激活阶段），必须在问题单的下一阶段开始之前完成。例如，也可以通过以下阶段定义 IT_Orders 定义：

图 41.1. IT_Orders 项目阶段示例



流程

1. 从位于图表编辑器左侧的预定义节点面板中，将 Adhoc 子进程节点拖放到设计 canvas 中，并提供 stage 节点的名称。

2. 定义如何激活阶段：

   - 如果进入节点激活阶段，请将该阶段与来自传入节点的序列流行连接。
   - 如果阶段由信号事件激活，请在包含第一步中配置的 stage 名称上配置 SignalRef。
   - 或者，配置 AdHocActivationCondition 属性，以在满足条件时激活阶段。

3. 根据需要重新调整节点大小，以提供为阶段添加任务节点。

4. 将相关的任务添加到阶段，并根据需要配置它们。

5. 可选：为阶段配置完成条件。作为临时子进程，阶段默认配置为自动完成。这意味着，当阶段的所有实例都不再活跃后，阶段将自动完成并触发问题单定义中的下一个活动。

   要更改完成条件，选择 stage 节点并打开 Properties 面板，展开 Implementation/Execution，再修改 AdHocCompletionCondition 属性字段，并为您需要的完成条件修改 AdHocCompletionCondition 属性字段。有关阶段完成条件的更多信息，请参阅 第 41.2 节 “配置阶段激活和完成条件”。

6. 配置阶段后，使用序列流行将其连接到问题单定义中的下一个活动。

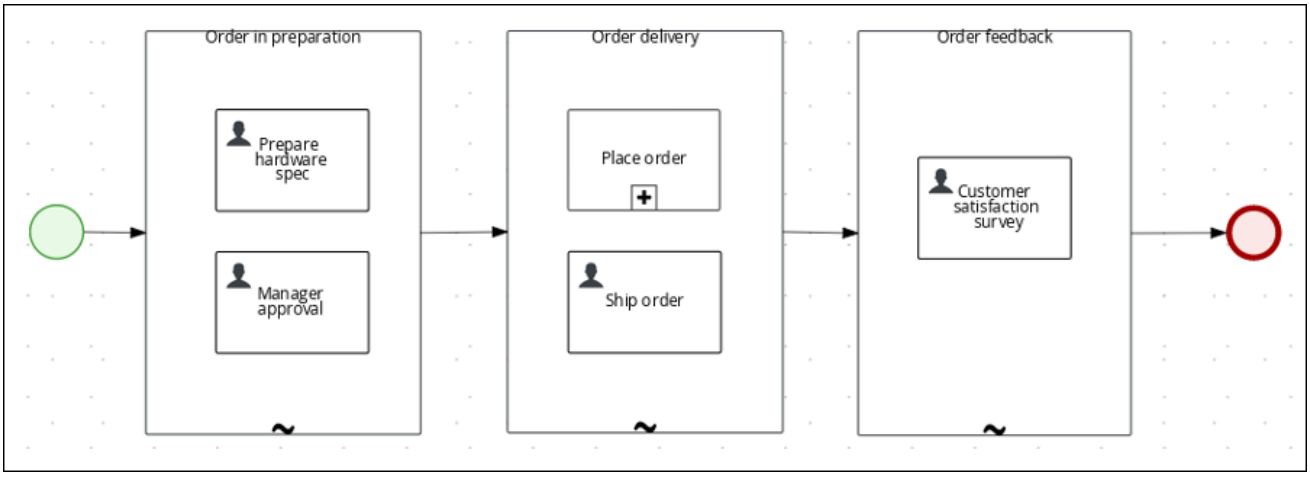
41.2. 配置阶段激活和完成条件

阶段可以由启动节点、中间节点或手动触发阶段，也可以使用 API 调用来手动触发。

您可以使用自由的 Drools 规则使用激活和完成条件配置阶段，这与配置 milestone 完成条件相同。例如，在 IT_Orders 示例项目中，Milestone 2：顺序 已提供完成条件

（org.kie.api.runtime.process.CaseData（data.get("shipped")== true）也可以用作 Order delivery 阶段的完成条件：

图 41.2. IT_Orders 项目阶段示例



还可使用自由的 Drools 规则配置激活条件，以配置 AdHocActivationCondition 属性来激活阶段。

先决条件

- 您已在 Business Central 进程设计器中创建了问题单定义。
- 您已将临时子进程添加到要用作阶段的大小写定义中。

流程

1. 选择 stage 后，点

   在设计程序右侧的 Properties 面板。

2. 展开 实施/执行 并在 AdHocActivationCondition 属性编辑器中定义起始节点的激活条件。例如，设置 autostart: true 以便在新问题单实例启动时自动激活 stage。

3. 默认将 AdHocCompletionCondition 设置为 autocomplete。要改变这种情况，使用 free-form Drools 表达式输入完成条件。例如，设置 org.kie.api.runtime.process.CaseData（data.get("ordered")== true），以在前面显示的示例中激活第二个阶段。

有关 IT_Orders 示例项目中使用的条件的更多信息，请参阅开始使用问题单管理。

41.3. 在 STAGE 中添加动态任务

使用 REST API 请求，可以在运行期间将动态任务添加到问题单阶段。这与在问题单实例中添加动态任务类似，但您还必须定义将任务添加到阶段的 caseStageId。

使用以下步骤将动态任务添加到 Business Central 中可用的 IT_Orders 示例项目中的一个阶段，使用 Swagger REST API 工具。相同的端点可用于在没有 Swagger 的情况下的 REST API。

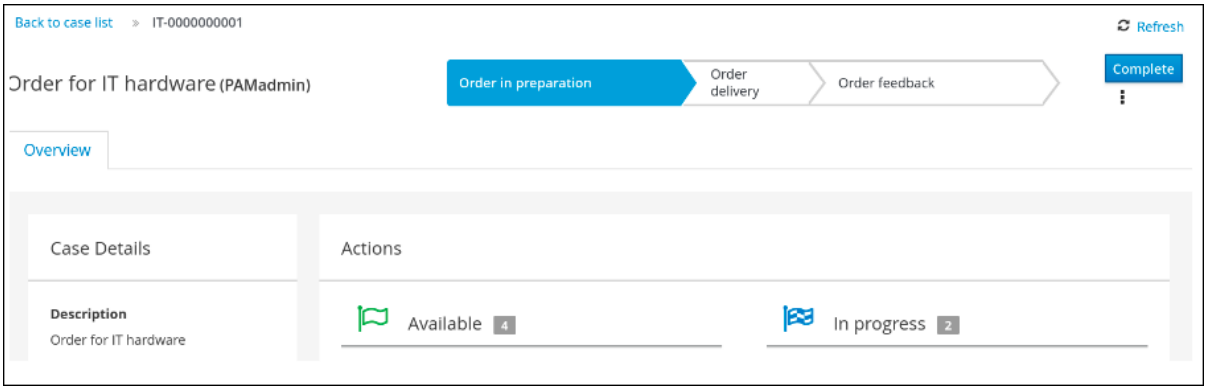
先决条件

- IT_Orders 示例项目 BPMN2 问题单定义已被重新配置为使用阶段，而不是 milestones，如提供的示例所示。有关为问题单管理配置阶段的详情，请参考 第 41.1 节 “定义阶段”。

流程

1. 使用 Showcase 应用启动新的问题单。有关使用 Showcase 的更多信息，请参阅使用 Showcase 应用程序进行问题单管理。

   因为这种情形是使用阶段设计的，所以问题单详情页面会显示阶段跟踪：



   第一阶段在创建问题单实例时自动启动。

2. 作为 manager 用户，在 Menu → Track → Task Inbox 下的 Business Central 中批准硬件规格，然后检查问题单的进度。

   a. 在 Business Central 中，点击 Menu → Manage → Process Instances，再打开活跃

的问题单实例 IT-0000000001。

b. 点击 图表 查看问题单进度。

3. 在网页浏览器中，打开以下 URL：

http://localhost:8080/kie-server/docs.

4. 在 问题单实例 :: Case Management 下打开可用端点的列表。

5. 点击以下 POST 方法端点打开详情：

/server/containers/{id}/cases/instances/{caseId}/stages/{caseStageId}/tasks

6. 点击 Try it out 以完成以下参数：

表 41.1. 参数

Name	描述
id	itorders
caseId	IT-0000000001
caseStageId	订购交付

caseStageId 是创建动态任务的大小写定义中阶段的名称。这可以是任何动态或服务任务有效负载。请参阅 Red Hat Process Automation Manager 中的开发流程服务。

在将动态任务添加到阶段后，必须完成它才能完成阶段，并使问题单进程进入问题单流中的下一项目。

第 42 章 MILESTONES

milestones 是一个特殊的服务任务，可通过将 milestone 节点添加到进程设计器面板，在案例定义设计人员中配置。在创建新问题单定义时，默认包含在设计面板上作为 AdHoc Autostart 的 milestone。默认情况下，新创建的 milestones 默认设置为 AdHoc Autostart。

在阶段结束时，案例管理里程通常发生，但也可是实现其他里程干的结果。milestone 始终需要定义一个条件来跟踪进度。当数据添加到问题单时，milestones 对问题单文件数据做出反应。milestone 代表问题单实例内实现的一个点。它可用于标记某些事件，对于关键性能指示器(KPI)跟踪或识别仍然完成的任务很有用。

在执行时，milestones 可以处于以下任意状态：

- Active ：该条件已在里程one上定义了，但该条件尚未满足。
- 完成: 满足 milestone 条件，达到 milestone，且案例可以继续进行下一任务。
- terminated ：milestone 不再是问题单进程的一部分，因此不再需要。

虽然 milestone 可用或完成，但可以通过信号手动触发，或者在问题单实例启动时自动配置 AdHoc Autostart。可以根据需要多次触发 milestones，但在满足条件时直接实现。

42.1. 配置和触发 MILESTONES

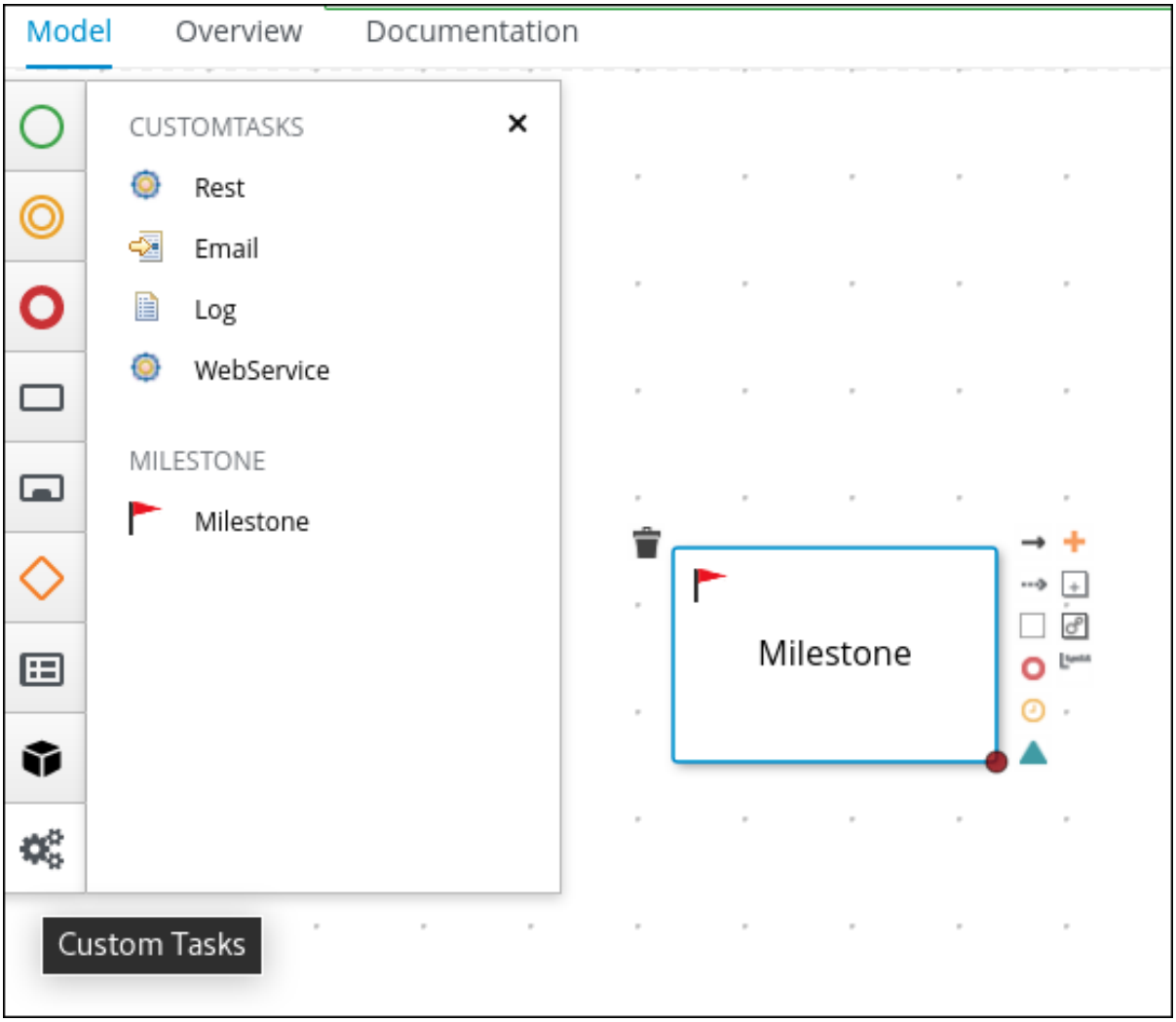
可以将案例 milestones 配置为在问题单实例启动时自动启动，也可以使用信号触发，该信号在问题单设计期间手动配置。

先决条件

- 在 Business Central 中创建了问题单项目。
- 已创建问题单定义。

流程

1. *从位于图表编辑器左侧的预定义节点面板中，将 Milestone 对象拖放到面板。*



2. *选择 milestone 后，点*

打开设计人员右侧的 Properties 面板。

3. *展开 Data Assignments 以添加完成条件。milestones 默认包含一个 Condition 参数。*

4. *要为 milestone 定义完成条件，请从 Source 列表中选择 Constant。该条件必须使用 Drools 语法提供。*

5. *扩展 实施/执行 以配置 AdHoc Autostart 属性。*

- *单击复选框，将 milestones 的此属性设置为 true，以便在问题单实例启动时自动启动。*

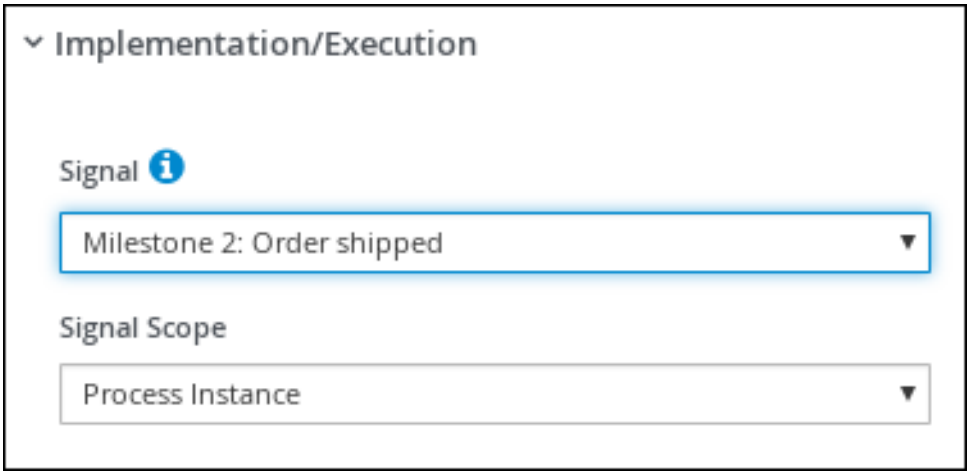
- 将复选框留空，将 milestones 设置为 false，后者由信号事件触发。

6. 可选：配置一个信号事件，以在达到问题单目标后触发 milestone。

   a. 在问题单设计面板中选择信号事件后，打开右侧的 Properties 面板。

   b. 将 Signal Scope 属性设置为 Process Instance。

   c. 打开 SignalRef 表达式编辑器，再键入要触发的 milestone 的名称。



7. 点击 Save。

第 43 章 变量标签

变量存储运行时使用的数据。为了更好地控制变量行为，您可以在 BPMN 问题单文件中标记问题单变量和本地变量。标签是简单的字符串值，作为元数据添加到特定变量中。

Red Hat Process Automation Manager 支持以下标签，用于问题单和本地变量：

- 必需 ：将 变量设置为要求，以便开始一个问题单。如果问题单在没有所需变量的情况下启动， Red Hat Process Automation Manager 会生成 VariableViolationException 错误。
- ReadOnly ：指示变量仅用于信息目的，且只能在执行案例期间设置一次。如果在任何时候修改了只读变量的值， Red Hat Process Automation Manager 会生成 VariableViolationException 错误。
- restricted ：与 VariableGuardProcessEventListener 一起使用的标签，以指示根据现有角色修改变量的权限。如果使用通过新标签名称的第二个构造器， 可以将 restricted 标签替换为任何其他标签名称。

VariableGuardProcessEventListener 类从 DefaultProcessEventListener 类扩展， 支持两个不同的构造器：

- VariableGuardProcessEventListener

```
public VariableGuardProcessEventListener(String requiredRole, IdentityProvider
identityProvider) {
    this("restricted", requiredRole, identityProvider);
}
```

- VariableGuardProcessEventListener

```
public VariableGuardProcessEventListener(String tag, String requiredRole,
IdentityProvider identityProvider) {
    this.tag = tag;
    this.requiredRole = requiredRole;
    this.identityProvider = identityProvider;
}
```

因此， 您必须在带有允许的角色名称和身份提供程序的会话中添加一个事件监听程序， 该供应

商返回用户角色，如下例所示：

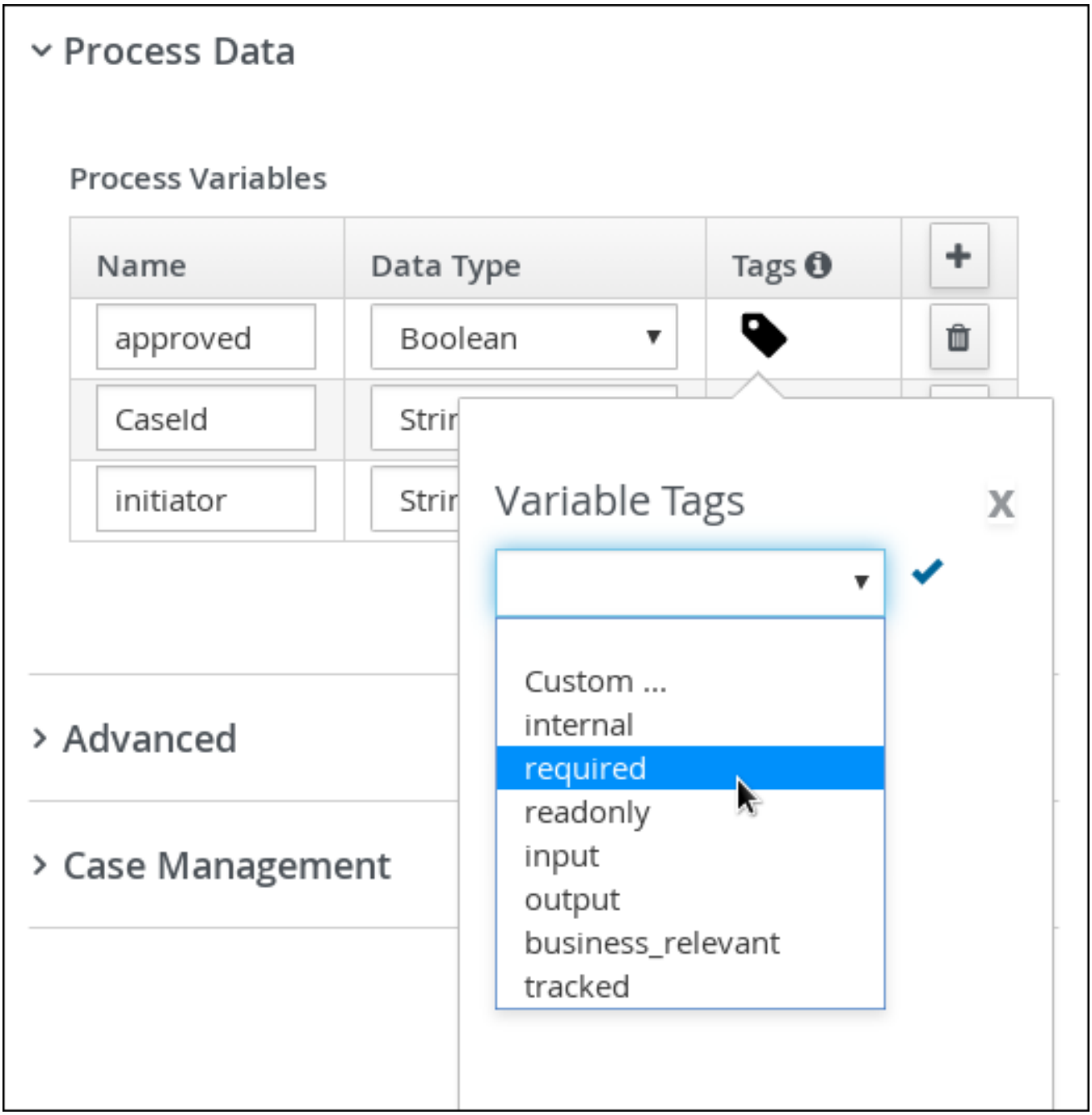
```
ksession.addEventListener(new VariableGuardProcessEventListener("AdminRole",
myIdentityProvider));
```

在上例中，VariableGuardProcessEventListener 方法验证变量是否带有安全约束标签 (restricted)。如果用户没有所需的角色（例如 AdminRole），Red Hat Process Automation Manager 会生成 VariableViolationException 错误。注意：Red Hat Process Automation Manager 不支持在 Business Central UI 中出现的变量标签，如 内部、输入、输出、业务相关和跟踪。

您可以将标签直接添加到 BPMN 进程源文件中，作为 customTags 元数据属性，其格式为 ![CDATA[TAG_NAME]]。

例如，以下 BPMN 进程将 所需的 标签应用到 批准 的进程变量中：

图 43.1. 在 BPMN 型号器中标记的变量示例



在 BPMN 文件中标记的变量示例

```
<bpmn2:property id="approved" itemSubjectRef="ItemDefinition_9" name="approved">
  <bpmn2:extensionElements>
    <tns:metaData name="customTags">
      <tns:metaValue><![CDATA[required]]></tns:metaValue>
    </tns:metaData>
  </bpmn2:extensionElements>
</bpmn2:property>
```

您可以将多个标签用于适用变量的变量。您还可以在 BPMN 文件中定义自定义变量标签，使变量数据可供 Red Hat Process Automation Manager 处理事件监听程序。自定义标签不会影响 Red Hat Process Automation Manager 运行时，作为标准变量标签，仅用于信息。您以相同的 customTags 元数据属性格式定义自定义变量标签，用于标准 Red Hat Process Automation Manager 变量标签。

第 44 章 问题单事件监听程序

CaseEventListener 侦听器用于启动对问题单实例上调用的与问题单相关的事件和操作的通知。通过覆盖特定用例所需的方法来实施问题单事件监听程序。

您可以使用 Menu → Design → PROJECT_NAME → Settings → Deployments 中的 Business Central 中的部署描述符来配置监听程序。

创建新项目时，会使用默认值生成 kie-deployment-descriptor.xml 文件。

CaseEventListener 方法

```
public interface CaseEventListener extends EventListener {

    default void beforeCaseStarted(CaseStartEvent event) {
    };

    default void afterCaseStarted(CaseStartEvent event) {
    };

    default void beforeCaseClosed(CaseCloseEvent event) {
    };

    default void afterCaseClosed(CaseCloseEvent event) {
    };

    default void beforeCaseCancelled(CaseCancelEvent event) {
    };

    default void afterCaseCancelled(CaseCancelEvent event) {
    };

    default void beforeCaseDestroyed(CaseDestroyEvent event) {
    };

    default void afterCaseDestroyed(CaseDestroyEvent event) {
    };

    default void beforeCaseReopen(CaseReopenEvent event) {
    };

    default void afterCaseReopen(CaseReopenEvent event) {
    };

    default void beforeCaseCommentAdded(CaseCommentEvent event) {
    };
```

```
    default void afterCaseCommentAdded(CaseCommentEvent event) {
    };

    default void beforeCaseCommentUpdated(CaseCommentEvent event) {
    };

    default void afterCaseCommentUpdated(CaseCommentEvent event) {
    };

    default void beforeCaseCommentRemoved(CaseCommentEvent event) {
    };

    default void afterCaseCommentRemoved(CaseCommentEvent event) {
    };

    default void beforeCaseRoleAssignmentAdded(CaseRoleAssignmentEvent event) {
    };

    default void afterCaseRoleAssignmentAdded(CaseRoleAssignmentEvent event) {
    };

    default void beforeCaseRoleAssignmentRemoved(CaseRoleAssignmentEvent event) {
    };

    default void afterCaseRoleAssignmentRemoved(CaseRoleAssignmentEvent event) {
    };

    default void beforeCaseDataAdded(CaseDataEvent event) {
    };

    default void afterCaseDataAdded(CaseDataEvent event) {
    };

    default void beforeCaseDataRemoved(CaseDataEvent event) {
    };

    default void afterCaseDataRemoved(CaseDataEvent event) {
    };

    default void beforeDynamicTaskAdded(CaseDynamicTaskEvent event) {
    };

    default void afterDynamicTaskAdded(CaseDynamicTaskEvent event) {
    };

    default void beforeDynamicProcessAdded(CaseDynamicSubprocessEvent event) {
    };

    default void afterDynamicProcessAdded(CaseDynamicSubprocessEvent event) {
    };
}
```

第 45 章 问题单管理的规则

这种情况是数据驱动的，而不是遵循后续流程。解决问题单所需的步骤取决于数据（此情况中涉及的人员提供），或者可将系统配置为根据可用数据触发进一步的操作。在后者的情况下，您可以使用业务规则来决定问题单继续或到达解决方案所需的进一步操作。

在这种情况下，可在任何时候将数据插入到案例文件中。决策引擎持续监控案例文件数据，这意味着规则对问题单文件中包含的数据做出响应。使用规则监控并响应问题单文件数据中的更改提供了一定程度的自动化功能。

45.1. 使用规则驱动器问题单

请参阅 Business Central 中的案例管理 IT_Orders 示例项目。

假设供应商提供的特定硬件规格不正确或无效。供应商需要提供新的有效顺序，以便问题单可以继续。您可以不等待经理拒绝无效的规格并创建供应商的新请求，您可以创建一个在问题单数据表示提供的规格无效时立即作出反应的业务规则。然后可以为供应商创建新的硬件规格请求。

以下步骤演示了如何创建和使用业务规则来执行此情况。

先决条件

- IT_Orders 示例项目在 Business Central 中是开放的，但它不部署到 KIE 服务器。
- ServiceRegistry 是 jbpm-services-api 模块的一部分，它必须在类路径上可用。

注意

如果在 Business Central 之外构建项目，则必须将以下依赖项添加到项目中：

- org.jbpm:jbpm-services-api
- org.jbpm:jbpm-case-mgmt-api

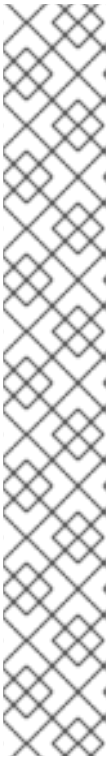
流程

1. 创建名为 **validate-document.drl** 的以下商业规则文件：

```
package defaultPackage;

import java.util.Map;
import java.util.HashMap;
import org.jbpm.casemgmt.api.CaseService;
import org.jbpm.casemgmt.api.model.instance.CaseFileInstance;
import org.jbpm.document.Document;
import org.jbpm.services.api.service.ServiceRegistry;

rule "Invalid document name - reupload"
when
   $caseData : CaseFileInstance()
   Document(name == "invalid.pdf") from $caseData.getData("hwSpec")

then

   System.out.println("Hardware specification is invalid");
   $caseData.remove("hwSpec");
   update($caseData);
   CaseService caseService = (CaseService)
ServiceRegistry.get().service(ServiceRegistry.CASE_SERVICE);
   caseService.triggerAdHocFragment($caseData.getCaseId(), "Prepare hardware
spec", null);
end
```

此商业规则将检测当名为 **invalid.pdf** 的文件被上传到 **case** 文件中。然后，它会删除 无效的.pdf 文档，并创建 **Prepare hardware spec** 用户任务的新实例。

2. 单击 **Deploy** 以构建 **IT_Orders** 项目，并将其部署到 **KIE** 服务器。



注意

您还可以选择 Build & Install 选项来构建项目，并将 KJAR 文件发布到配置的 Maven 存储库，而无需部署到 KIE 服务器。在开发环境中，您可以点击 Deploy 将构建的 KJAR 文件部署到 KIE 服务器，而无需停止任何正在运行的实例（如果适用），或者点击 Redeploy 来部署构建的 KJAR 文件并替换所有实例。下次部署或重新部署构建的 KJAR 时，以前的部署单元（KIE 容器）会在同一目标 KIE 服务器中自动更新。在生产环境中，Redeploy 选项被禁用，且只能点击 Deploy 将构建的 KJAR 文件部署到 KIE 服务器上的新部署单元（KIE 容器）。

要配置 KIE 服务器环境模式，请将 org.kie.server.mode 系统属性设置为 org.kie.server.mode=development 或 org.kie.server.mode=production。要在 Business Central 中为对应项目配置部署行为，请转至 Project Settings → General Settings → Version，再切换 Development Mode 选项。默认情况下，Business Central 中的 KIE 服务器和所有新项目均为开发模式。您不能部署打开开发模式的项目，或使用手动将 SNAPSHOT 版本后缀添加到生产模式中的 KIE 服务器。

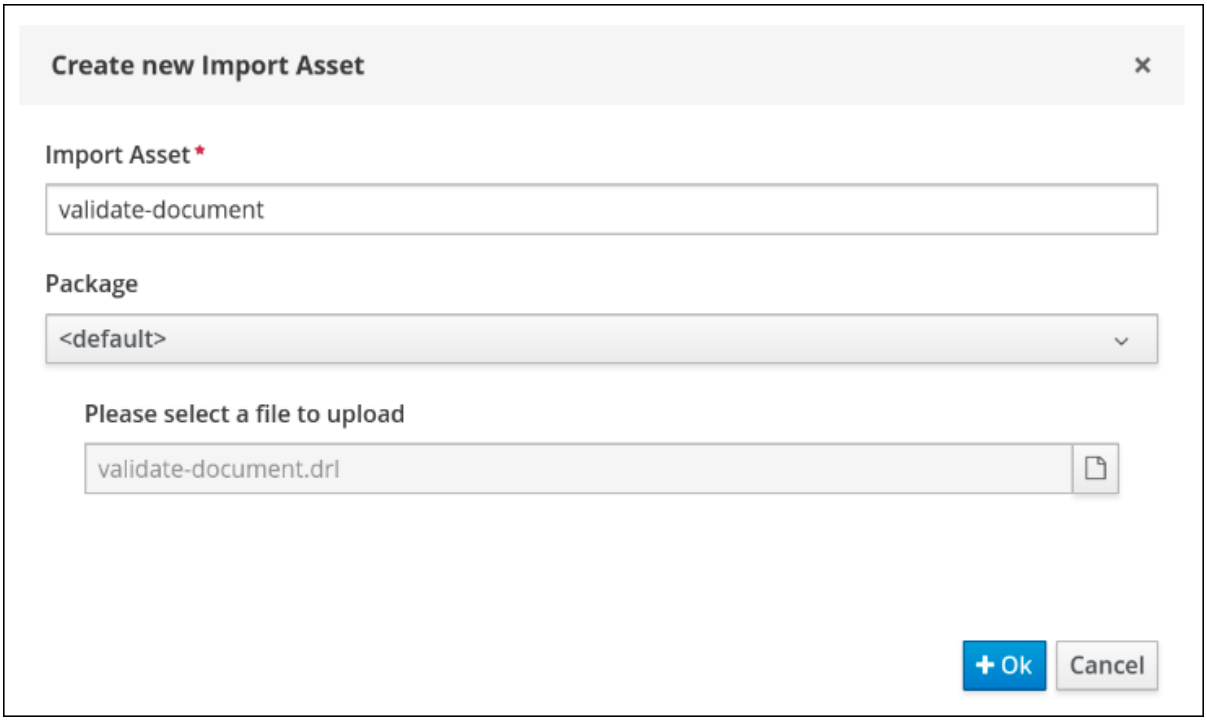
3. 创建名为 invalid.pdf 的文件，并将它保存到本地。

4. 创建名为 valid-spec.pdf 的文件，并将它保存到本地。

5. 在 Business Central 中，前往 Menu → Projects → IT_Orders 来打开 IT_Orders 项目。

6. 单击页面右上角的 Import Asset。

7. 将 validate-document.drl 文件上传到 默认软件包 (src/main/resources)，再单击 Ok。

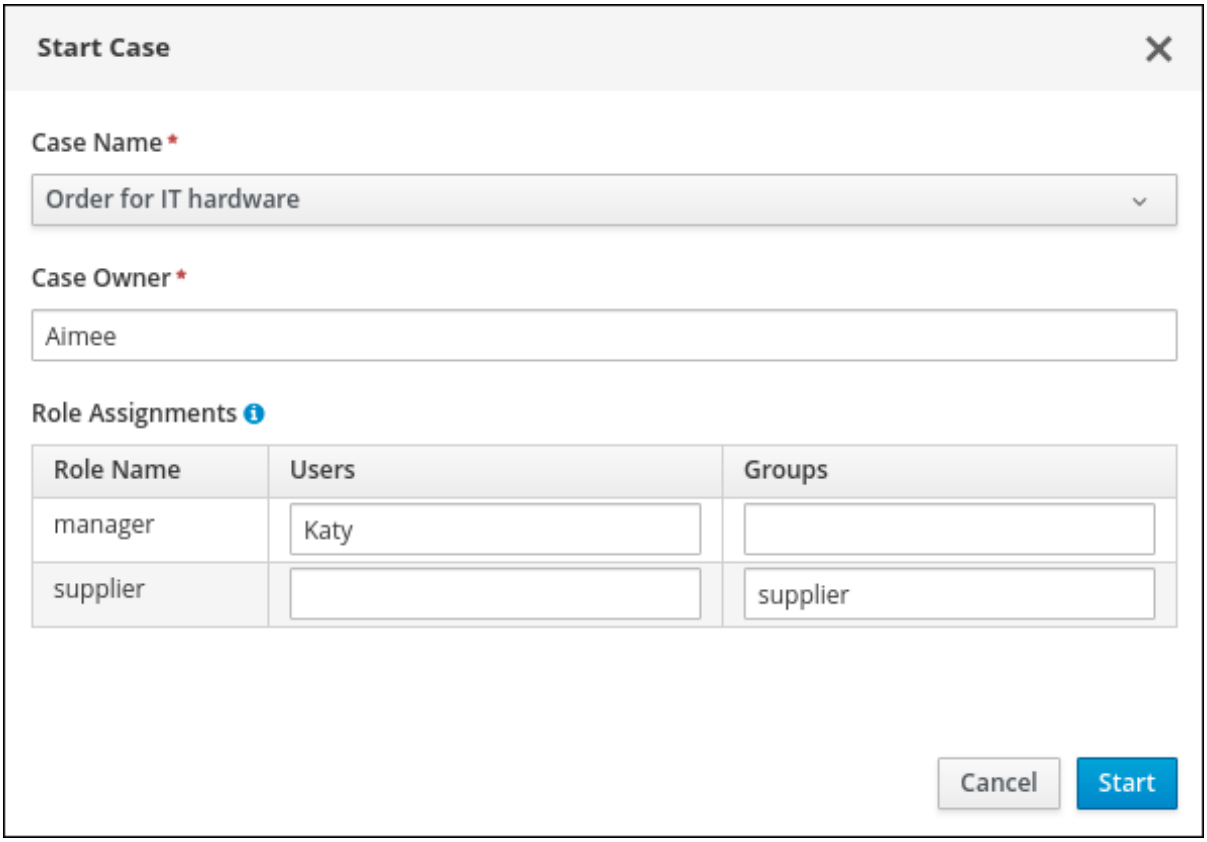


validate-document.drl 规则显示在规则编辑器中。点击 Save 或 close 退出规则编辑器。

8. *点击 Apps 启动程序 （如果已安装）或访问 http://localhost:8080/rhpam-case-mgmt-showcase/jbpm-cm.html 来打开 Showcase 应用程序。*

9. *点 IT_Orders 项目的 Start Case。*

在本示例中，Aimee 是案例 所有者，Katy 是 经理，供应商组是 供应商。



10. *从 **Business Central** 注销，然后以属于 **vendor** 组 的用户身份登录。*

11. *进入 **Menu** → **Track** → **Task Inbox**。*

12. *打开 **Prepare hardware spec** 任务，再点击 **Claim**。这会将任务分配给登录用户。*

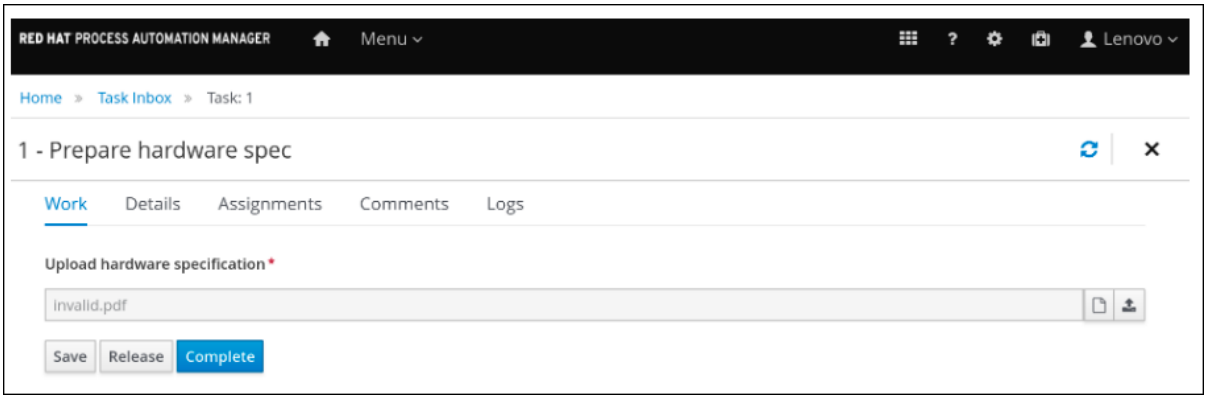
13. *点 **Start** 并点*

    找到 无效的.pdf 硬件规格文件。点击

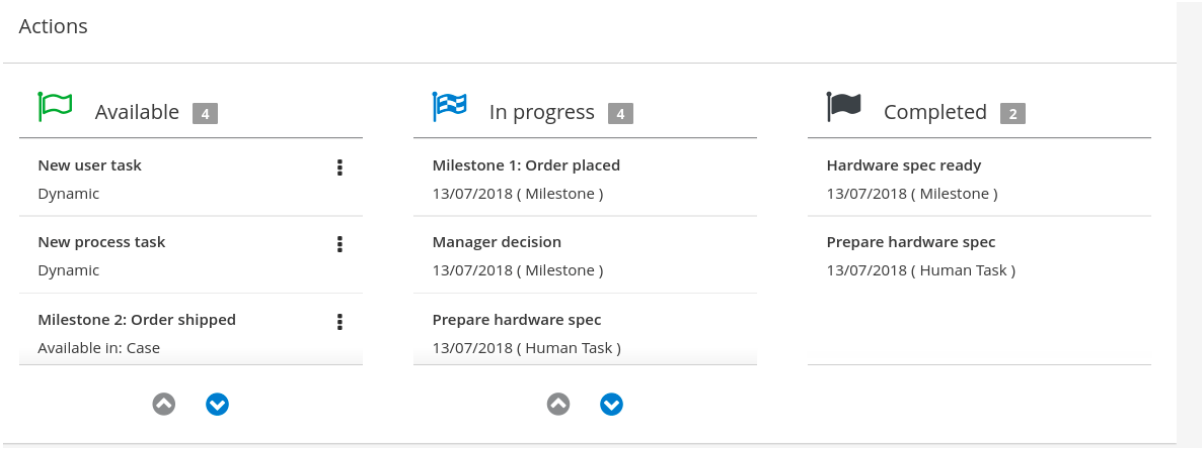
    上传该文件。



14. *点 Complete。*

准备硬件规格的 Task Inbox 中的值是 Ready。

15. *在 Showcase 中，单击右上角的 Refresh。请注意，Completed 列中会出现 Prepare hardware 任务 消息，另一个显示在 In Progress 列中。*



这是因为第一个 Prepare hardware spec 任务已完成，其规格文件 无效。pdf。因此，业务规则会导致任务和文件被丢弃，并创建了新用户任务。

16. *在 Business Central Task Inbox 中，重复前面的步骤来上传 valid-spec.pdf 文件而不是 无效.pdf。*

第 46 章 问题单管理安全

案例在问题单定义级别配置了 case 角色。这些是涉及问题单处理的通用参与者。这些角色可以分配给用户任务，也可以用作联系人参考。角色不会硬编码到特定的用户或组，以便使问题单定义独立于任何给定实例涉及的实际执行者。只要实例处于活动状态，您可以在任何时候修改问题单角色分配，但修改角色分配不会影响之前角色分配创建的任务。

默认启用实例安全性。案例定义可防止不属于该问题单的用户访问数据。除非用户有问题单角色分配（以用户或组成员分配），否则他们无法访问问题单实例。

案例安全性是建议在启动案例实例时分配案例角色的原因之一，因为这会防止将任务分配给本例不应访问的用户。

46.1. 配置问题单管理的安全性

您可以通过将以下系统属性设置为 false 来关闭问题单实例授权：

org.jbpm.cases.auth.enabled

这个系统属性只是一个安全组件，用于实例。另外，您可以使用 case-authorization.properties 文件在执行服务器级别配置问题单操作，它位于执行服务器应用程序的类路径根目录(kie-server.war/WEB-INF/classes)。

对所有可能的案例定义使用简单的配置文件可强制您认为是特定于域的案例管理。案例安全性的 AuthorizationManager 可以插入，它允许您为特定安全处理包含自定义代码。

您可以将以下问题单实例操作限制为问题单角色：

- CANCEL_CASE
- DESTROY_CASE
- REOPEN_CASE

- ***ADD_TASK_TO_CASE***
- ***ADD_PROCESS_TO_CASE***
- ***ADD_DATA***
- ***REMOVE_DATA***
- ***MODIFY_ROLE_ASSIGNMENT***
- ***MODIFY_COMMENT***

先决条件

- ***Red Hat Process Automation Manager KIE 服务器没有运行。***

流程

1. ***在首选的编辑器中打开 JBOSS_HOME/standalone/deployments/kie-server.war/WEB-INF/classes/case-authorization.properties 文件。***

   默认情况下，该文件包含以下操作限制：

```
CLOSE_CASE=owner,admin
CANCEL_CASE=owner,admin
DESTROY_CASE=owner,admin
REOPEN_CASE=owner,admin
```

2. ***根据需要为这些操作添加或删除角色权限：***

   a. ***要删除某个角色执行操作的权限，请在 case-authorization.properties 文件中从该操作的授权角色列表中删除它。例如，从 CLOSE_CASE 操作中删除 admin 角色会将权限设置为所有情况的问题单所有者。***

   b.

要赋予一个角色权限来执行问题单操作，请将其添加到 case-authorization.properties 文件中该操作的授权角色列表中。例如，要允许具有 manager 角色的任何人都执行 CLOSE_CASE 操作，请将其添加到角色列表中，用逗号分隔：

CLOSE_CASE=owner,admin,manager

3. 要为文件中列出的其他问题单操作添加角色限制，请从行中删除 # 并以以下格式列出角色名称：

   OPERATION=role1,role2,roleN

   文件中以 # 开头的操作受到忽略的限制，并可以被涉及的任何人执行。

4. 在分配完角色权限后，保存并关闭 case-authorization.properties 文件。

5. 启动执行服务器。

   问题单授权设置应用到执行服务器中的所有情况。

第 47 章 关闭问题单

当没有执行更多活动且实现业务目标时，可以完成问题单实例，或者预先关闭该实例。通常，问题单所有者在完成所有工作并且满足问题单目标时关闭此情况。关闭问题单时，请考虑添加有关问题单实例关闭的注释。

如果需要，以后可以重新打开已关闭的问题单 ID。当问题单被重新打开时，当问题单被重新打开时，关闭时会处于活跃状态的阶段。

您可以使用 KIE 服务器 REST API 请求或直接在 Showcase 应用程序中远程关闭实例。

47.1. 使用 KIE 服务器 REST API 关闭问题单

您可以使用 REST API 请求来关闭问题单实例。Red Hat Process Automation Manager 包括 Swagger 客户端，其中包括 REST API 请求的端点和文档。另外，您可以使用同一端点来使用首选客户端或 Curl 进行 API 调用。

先决条件

- 已使用 Showcase 启动问题单实例。
- 您可以以具有 admin 角色的用户验证 API 请求。

流程

1. 在网页浏览器中打开 Swagger REST API 客户端：

   http://localhost:8080/kie-server/docs

2. 在 Case Instances :: Case Management 下，使用以下端点打开 POST 请求：

   /server/containers/{id}/cases/instances/{caseId}

3. 点 Try it out 并填写所需参数：

表 47.1. 参数

Name	描述
id	**itorders**
caseId	**IT-0000000001**

4. *可选：包含要包含在问题单文件中的注释。要留言，请在 正文 文本字段中键入内容作为字符串 。*

5. *点 Execute 关闭该问题单。*

6. *要确认问题单已经关闭，打开 Showcase 应用程序并将问题单列表状态更改为 Closed。*

47.2. 在 SHOWCASE 应用程序中关闭问题单

当不需要执行更多活动且实现业务目标时，实例就会完成。问题单完成后，您可以关闭问题单来指示问题单已完成，且不需要进一步工作。当您关闭某个问题单时，请考虑添加有关为何关闭该问题单的具体注释。如果需要，您可以稍后使用相同的问题单 ID 重新打开问题单。

您可以随时使用 Showcase 应用程序关闭问题单实例。通过 Showcase，您可以在关闭问题单前轻松查看问题单的详细信息或留言。

先决条件

- *您登录到 Showcase 应用，是您要关闭的问题单实例的所有者或管理员。*

流程

1. *在 Showcase 应用程序中，找到您要从问题单实例列表中关闭的问题单实例。*

2. *要在不首先查看详情的情况下关闭问题单，点 Close。*

3. *要从问题单详情页面关闭问题单，请点击列表中的问题单将其打开。*

在问题单概览页面中，您可以在问题单中添加注释，并验证您是否根据问题单信息关闭正确的问题单。

4. 点击 Close 关闭问题单。

5. 单击页面左上角的 Back to Case List，以返回到 Showcase case 列表视图。

6. 点 Status 旁边的下拉列表，然后选择 Canceled 以查看关闭和取消的问题单列表。

第 48 章 取消或销毁问题单

如果不再需要的用例，可以取消这些情况，且不需要任何问题单都可以正常工作。之后可以使用相同的问题单实例 ID 和问题单文件数据重新打开取消的用例。在某些情况下，您可能想要永久销毁一个问题单，使其无法重新打开。

问题单只能使用 API 请求来取消或销毁。Red Hat Process Automation Manager 包括 Swagger 客户端，其中包括 REST API 请求的端点和文档。另外，您可以使用同一端点来使用首选客户端或 Curl 进行 API 调用。

先决条件

- 已使用 Showcase 启动问题单实例。
- 您可以以具有 admin 角色的用户验证 API 请求。

流程

1. 在网页浏览器中打开 Swagger REST API 客户端：

   http://localhost:8080/kie-server/docs

2. 在 Case Instances :: Case Management 下，使用以下端点打开 DELETE 请求：

   /server/containers/{id}/cases/instances/{caseId}

   您可以使用 DELETE 请求取消问题单。另外，您还可以使用 destroy 参数销毁问题单。

3. 点 Try it out 并填写所需参数：

表 48.1. 参数

Name	描述
id	itorders

Name	描述
caseId	**IT-0000000001**
destroy	**true** （可选）永久销毁情况。此参数默认为 **false**。）

4. *点 Execute 取消（或销毁）问题单。*

5. *要确认问题单已被取消，请打开 Showcase 应用，并将问题单列表状态更改为 Canceled。如果问题单被销毁，它将不再出现在任何问题单列表中。*

48.1. 从数据库中删除问题单日志

使用 CaseLogCleanupCommand 清理问题单，如取消使用数据库空间的取消问题单。CaseLogCleanupCommand 命令包含自动清理所有或选定问题单的逻辑。

您可以在 CaseLogCleanupCommand 命令中使用以下配置选项：

表 48.2. CaseLogCleanupCommand 参数表

Name	描述	isclusive
SkipProcessLog	在命令运行时，指明进程和节点实例是否以及进程变量日志清理。默认值：**false**	没有，可用于其他参数
SkipTaskLog	在命令运行时，指示任务审计、任务事件和任务变量日志清理是否将被跳过。默认值：**false**	没有，可用于其他参数
SkipExecutorLog	指明在命令运行时，是否跳过 Red Hat Process Automation Manager executor 条目。默认值：**false**	没有，可用于其他参数
SingleRun	指明作业例程是否只运行一次。默认值：**false**	没有，可用于其他参数

Name	描述	isclusive
NextRun	调度下一个作业执行。例如，对于每 12 小时执行的作业，设置为 **12h**。如果您将 **SingleRun** 设置为 **true**，除非同时设置了 **SingleRun** 和 **NextRun**，则计划会被忽略。如果同时设置了这两个程序，则 **NextRun** 调度将具有优先权。ISO 格式可用于设置精确的日期。默认值 **：24h**	没有，可用于其他参数
OlderThan	超过指定日期的旧日志会被删除。日期格式为 **YYYY-MM-DD**。通常，此参数用于单次运行作业。	是，在使用 **OlderThanPeriod** 参数时无法使用
OlderThanPeriod	超过指定计时器表达式的旧日志会被删除。例如，将 30d 设置为删除超过 30 天的日志。	是，在使用 **OlderThan** 参数时无法使用
ForCaseDefId	指定已删除的日志的大小写定义 ID。	没有，可用于其他参数
ForDeployment	指定已删除的日志的部署 ID。	没有，可用于其他参数
EmfName	用于执行删除操作的持久性单元名称。默认值： **org.jbpm.domain**	N/A
DateFormat	指定与时间相关的参数的日期格式。默认值： **yyyy-MM-dd**	没有，可用于其他参数
状态	已删除的日志实例的状态。	没有，可用于其他参数

第 49 章 其他资源

- *问题单管理入门*
- ***使用 Showcase 应用程序进行问题单管理***

部分 V. 使用 SHOWCASE 应用程序进行问题单管理

作为 worker 或流程管理员，您可以使用 Showcase 应用程序管理和监控案例管理应用程序，同时在 Business Central 中执行案例工作。

案例管理与业务流程管理(BPM)不同，它注重于整个案例中处理的实际数据，也减少了完成目标所采取的步骤序列。在处理案例时，问题单数据是最重要的信息，而业务上下文和决策是在人类工作者的手中。

使用本文档，使用 Business Central 中的 IT_Orders 示例管理项目安装 Showcase 应用程序并启动一个问题单实例。使用 Business Central 完成完成 IT 订单案例所需的任务。

先决条件

- Red Hat JBoss Enterprise Application Platform 7.4 已安装。有关安装信息，请参阅 Red Hat JBoss Enterprise Application Platform 7.4 安装指南。
- Red Hat Process Automation Manager 安装在红帽 JBoss EAP 上，并配置了 KIE 服务器。如需更多信息，请参阅在 Red Hat JBoss EAP 7.4 上安装和配置 Red Hat Process Automation Manager。
- KieLoginModule 在 standalone-full.xml 中配置。这需要连接到 KIE 服务器。有关配置 KIE 服务器的更多信息，请参阅 规划 Red Hat Process Automation Manager 安装。
- Red Hat Process Automation Manager 正在运行，您可以使用具有 kie-server 和 用户角色的用户登录 Business Central。如需有关角色的更多信息，请参阅 规划 Red Hat Process Automation Manager 安装。
- IT_Orders 示例项目已导入在 Business Central 中并部署到 KIE 服务器。有关问题单管理的更多信息，请参阅开始使用问题单管理。

第 50 章 问题单管理

案例管理是业务流程管理(BPM)的扩展，可让您管理适应性业务流程。

BPM 是用于自动执行可重复性且具有共同模式的任务，并专注于优化流程。业务流程通常以明确明确定义的路径为实现业务目标而建模。这需要很多可预测性，通常基于大规模生产原则。但是，许多实际应用不能完全描述完成（包括所有可能的路径、偏差和例外）。在某些情况下，使用面向流程的方法可导致难以维护的复杂解决方案。

案例管理为非可预测、无法预测的流程提供了问题解决，而非针对日常预见性任务的 BPM 效率导向的方法。当进程无法预先预测时，它管理一次性情况。案例定义通常包含松散耦合的流程片段，这些片段可以直接连接或间接连接导致特定里程干和最终的业务合作伙伴，而进程能动态管理，以响应运行时发生的变化。

在 Red Hat Process Automation Manager 中，问题单管理包括以下核心流程引擎功能：

- 问题单文件实例
- 每个问题单运行时策略
- 问题单评论
- milestones
- 阶段
- 临时片段
- 动态任务和流程
- 问题单标识符（关联密钥）

- 问题单生命周期（关闭、重新打开、取消、销毁）

问题单定义始终是一个临时进程定义，不需要显式启动节点。该案例定义是业务用例的主要入口点。

进程定义是作为支持问题单的支持构造而成的，可以在问题单定义中按定义调用，也可以根据需要动态上线处理。案例定义定义了以下新对象：

- 活动（必需）
- case file（必需）
- milestones
- 角色
- 阶段

第 51 章 问题单管理显示应用程序

The Showcase 应用程序包含在 Red Hat Process Automation Manager 分发中，以展示应用程序环境中案例管理的功能。展示旨在用作概念验证，旨在展示业务流程管理(BPM)和案例管理之间的交互。您可以使用应用程序启动、关闭、监控和与问题单交互。

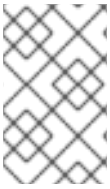
除 Business Central 应用程序和 KIE 服务器外，还必须安装展示功能。需要使用 Showcase 应用程序来启动新的问题单实例，但案例工作仍在 Business Central 中执行。

创建问题单实例并正在正常工作后，您可以点击 Case List 中的问题单打开问题单 Overview 页来监控 Showcase 应用程序中的问题单。

展示支持

The Showcase application 不是 Red Hat Process Automation Manager 的完整部分，用于演示问题单管理。提供的展示是为了鼓励客户对其使用和修改，以满足其具体需求。应用程序本身的内容不提供特定于产品的服务级别协议(SLA)。我们鼓励您报告问题、增强请求以及在显示案例更新中考虑的任何其他反馈。

红帽支持将基于商业合理的使用方法提供使用此模板的指导，不包括在内提供的提供的示例 UI 代码。



注意

产品支持仅限于 Red Hat Process Automation Manager 发行版本。

第 52 章 安装并登录到 SHOWCASE 应用程序

在附加组件 Zip 文件中，Red Hat Process Automation Manager 7.13 发行版本中包括了 Showcase 应用程序。此应用程序的目的是展示红帽流程自动化管理器中案例管理的功能，并可让您与 Business Central 中创建的案例进行交互。您可以在 Red Hat JBoss Enterprise Application Platform 实例或 OpenShift 上安装 Showcase 应用程序。这个步骤描述了如何在红帽 JBoss EAP 中安装 Showcase 应用程序。

先决条件

- Business Central 和 KIE 服务器安装在红帽 JBoss EAP 实例中。
- 您已创建了具有 kie-server 和 用户 角色的用户。只有具有 用户角色 的用户才能登录到 Showcase 应用。用户还需要 kie-server 角色在正在运行的 KIE 服务器上执行远程操作。
- Business Central 未运行。

流程

1. 进入红帽客户门户网站中的 Software Downloads 页面（需要登录），然后从下拉列表中选择产品和版本：
   - 产品： 流程自动化管理器
   - Version: 7.13.5
2. 下载 Red Hat Process Automation Manager 7.13 Add Ons (rhpam-7.13.5-add-ons.zip)。
3. 提取 rhpam-7.13.5-add-ons.zip 文件。rhpam-7.13-case-mgmt-showcase-eap7-deployable.zip 文件位于解压缩的目录中。
4. 将 rhpam-7.13-case-mgmt-showcase-eap7-deployable.zip 存档提取到临时目录。在以下示例中，此目录名为 TEMP_DIR。
5. 将 _TEMP_DIR/rhpam-7.13-case-mgmt-showcase-eap7-deployable/jboss-eap-7.4 目录

的内容复制到 EAP_HOME。

当要求覆盖文件或合并目录时，请选择" 是 "。

警告

确定您复制的 Red Hat Process Automation Manager 部署的名称不会与您在 Red Hat JBoss EAP 实例中的现有部署冲突。

6. *将以下系统属性添加到部署的 jboss-eap-7.4/standalone/configuration/standalone-full.xml 文件中：*

```
<property name="org.jbpm.casemgmt.showcase.url" value="/rhpam-case-mgmt-showcase"/>
```

7. *在终端应用程序中，导航到 EAP_HOME/bin，并运行单机配置来启动 Business Central：*

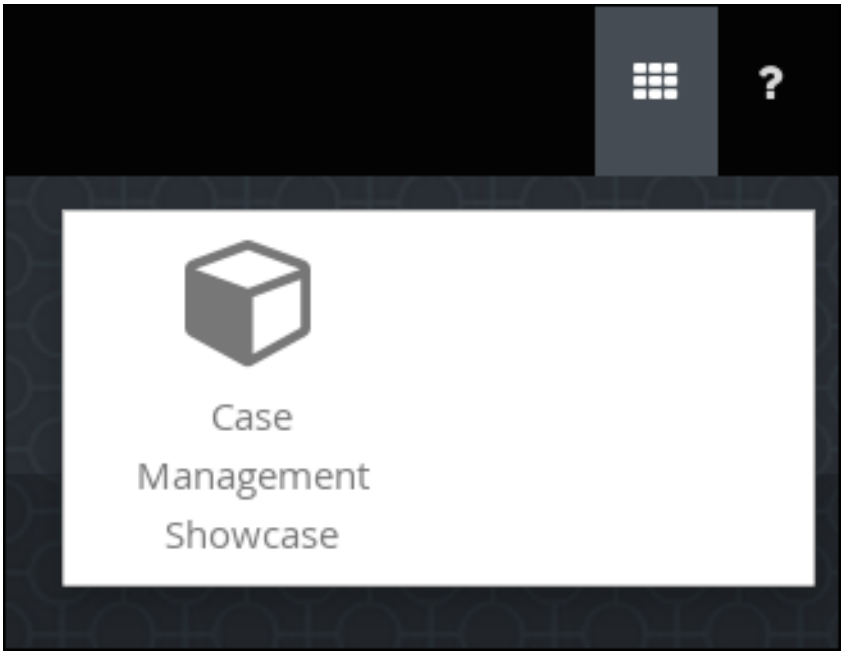
```
./standalone.sh -c standalone-full.xml
```

8. *在 Web 浏览器中，输入 localhost:8080/business-central。*

如果 Red Hat Process Automation Manager 已配置为从域名运行，请使用域名替换 localhost，例如：

```
http://www.example.com:8080/business-central
```

9. *在 Business Central 的右上角，单击 Apps launcher 按钮，在新的浏览器窗口中启动 问题单管理 显示。*



10. ***使用您的 Business Central 用户凭证登录到 Showcase 应用程序。***

第 53 章 问题单角色

case roles 为用户参与案例处理提供额外的抽象层。在管理时，角色、用户和组用于不同的目的。

角色

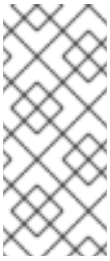
角色驱动问题单实例的授权，用于用户活动分配。用户可以或一个或多个组分配给所有者角色。所有者是谁是该案例。作为案例定义的一部分，角色不限于一组人或组。使用 roles 指定任务分配，而不是将特定的用户或组分配到任务分配，以确保问题单保持动态。

组

组是能够执行特定任务或具有一组指定职责的用户集合。您可以将任何数量的人员分配给一个组，并将任何组分配到某个角色。您可以随时添加或更改组的成员。不要将组硬编码到特定任务。

用户

用户是个人，当您将某个角色分配角色或添加到组中时，可以为他们授予特定的任务。



注意

不要在进程引擎或 KIE Server 中创建一个名为 unknown 的用户。未知 用户帐户是具有超级用户访问权限的保留系统名称。当用户没有登录时，未知 用户帐户执行与 SLA 违反监听程序相关的任务。

以下示例演示了如何使用以下信息将前面的问题单管理概念应用到热保留中：

- 角色 ：guest
- Group:Receptionist,Maid
- 用户:Marilyn

Guest 角色分配会影响关联案例的具体工作，适用于所有案例实例。

重要

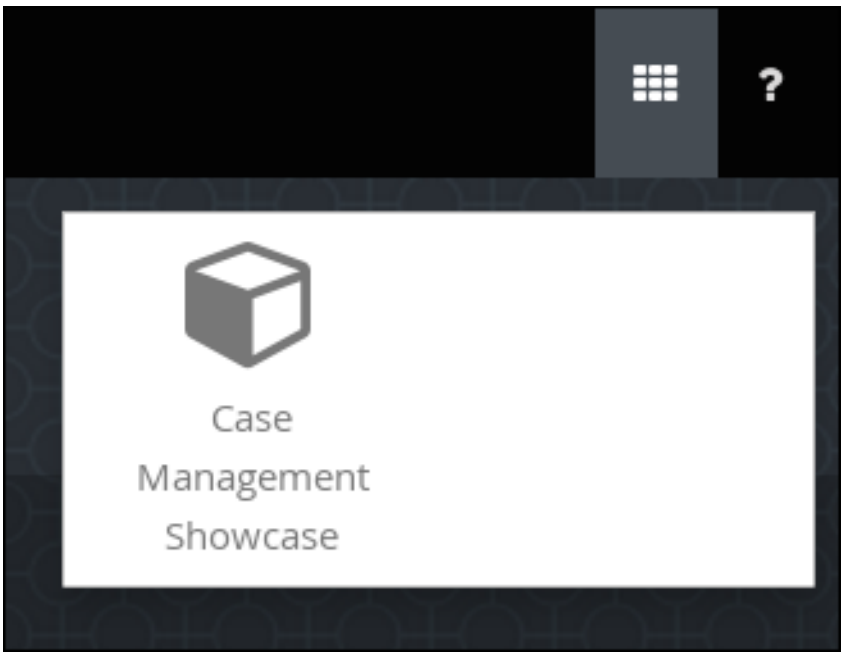
第 54 章

先决条件

-
-
-

流程

1.



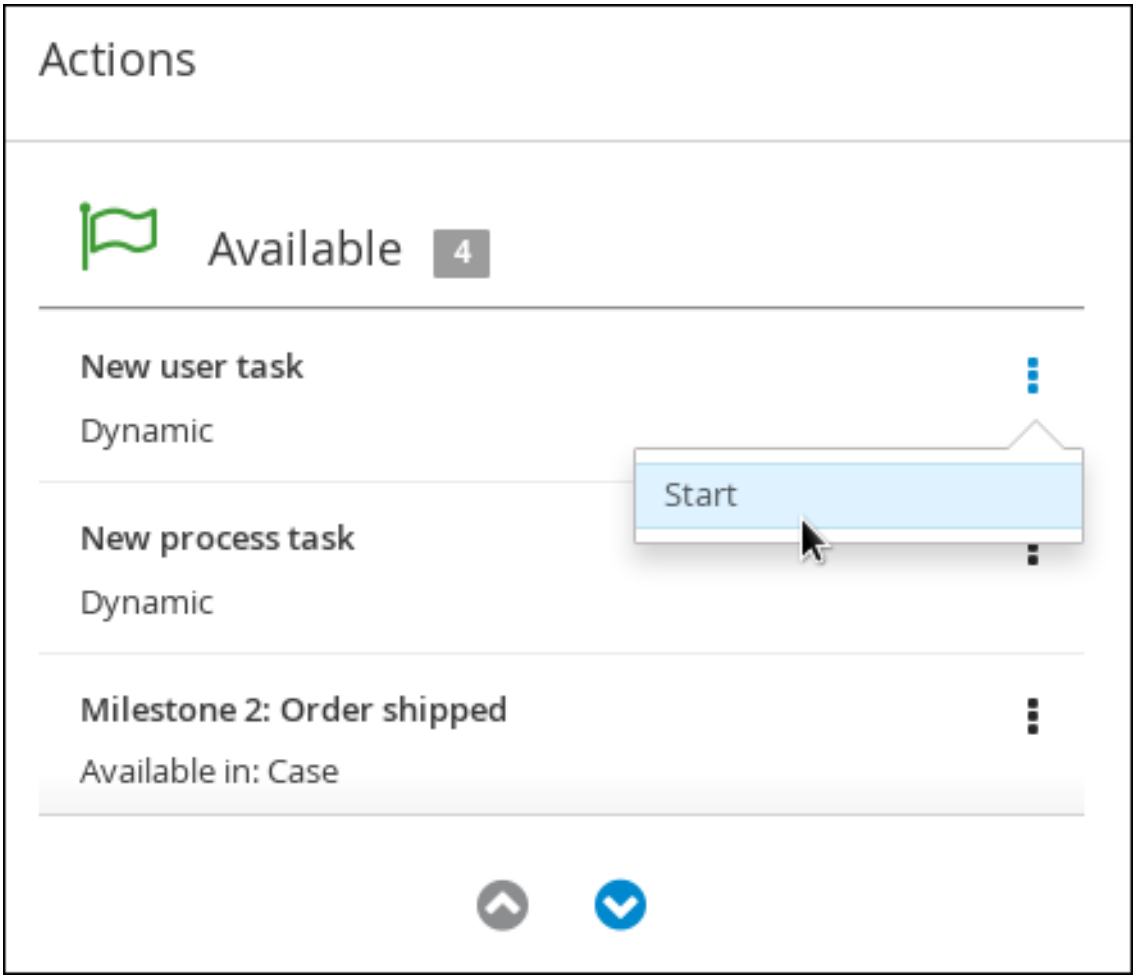
[D]

2.

3.

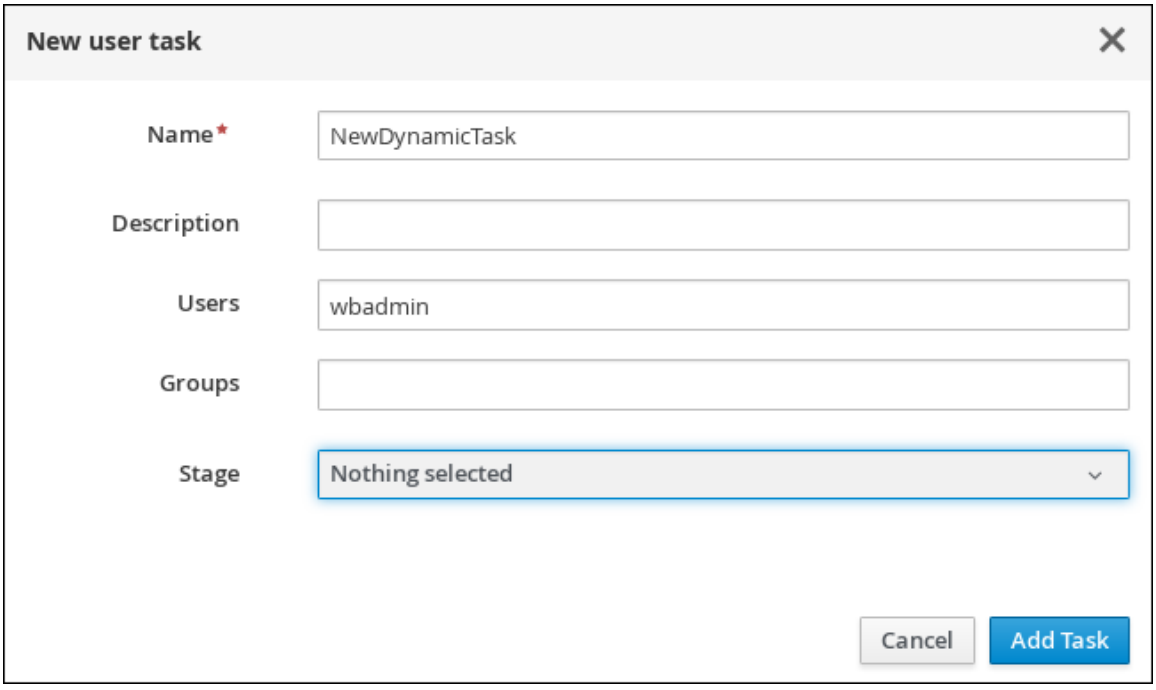
4.

图 54.1.

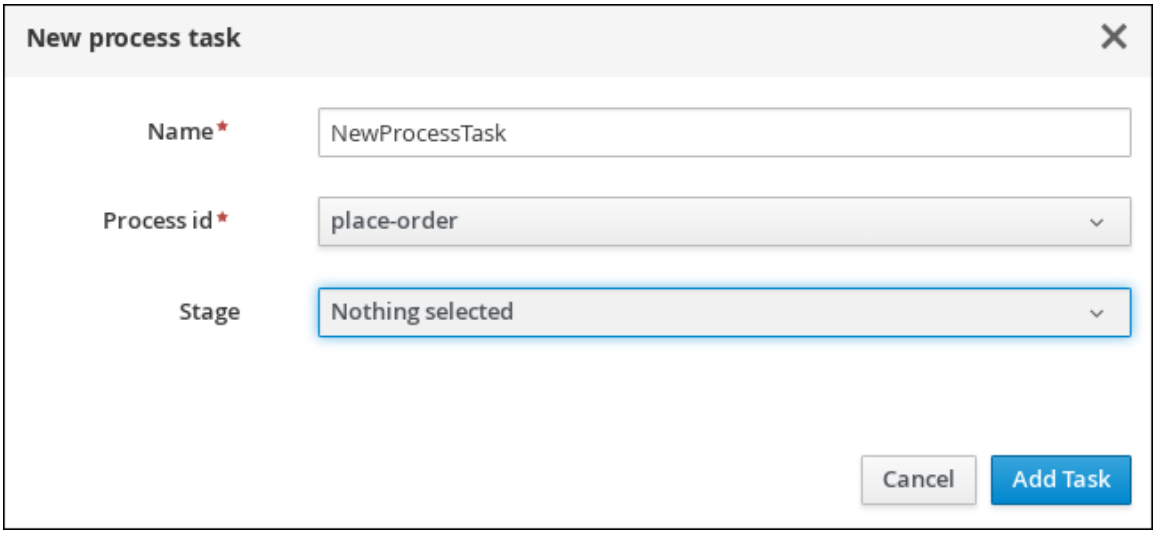

Actions
Available
4
New user task
Dynamic
Start
New process task
Dynamic
Milestone 2: Order shipped
Available in: Case

[D]

-

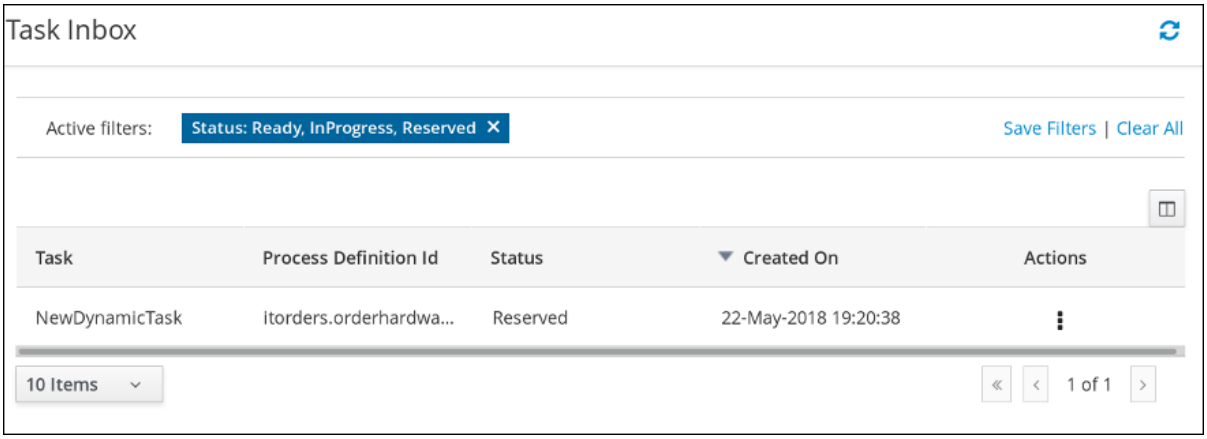

New user task
Name*
NewDynamicTask
Description
Users
wbadmin
Groups
Stage
Nothing selected
Cancel
Add Task

-


New process task
Name*
NewProcessTask
Process id*
place-order
Stage
Nothing selected
Cancel
Add Task

[D]

5.


Task Inbox
Active filters:
Status: Ready, InProgress, Reserved
Save Filters | Clear All
Task
Process Definition Id
Status
Created On
Actions
NewDynamicTask
itorders.orderhardwa...
Reserved
22-May-2018 19:20:38
10 Items
1 of 1

[D]

a.

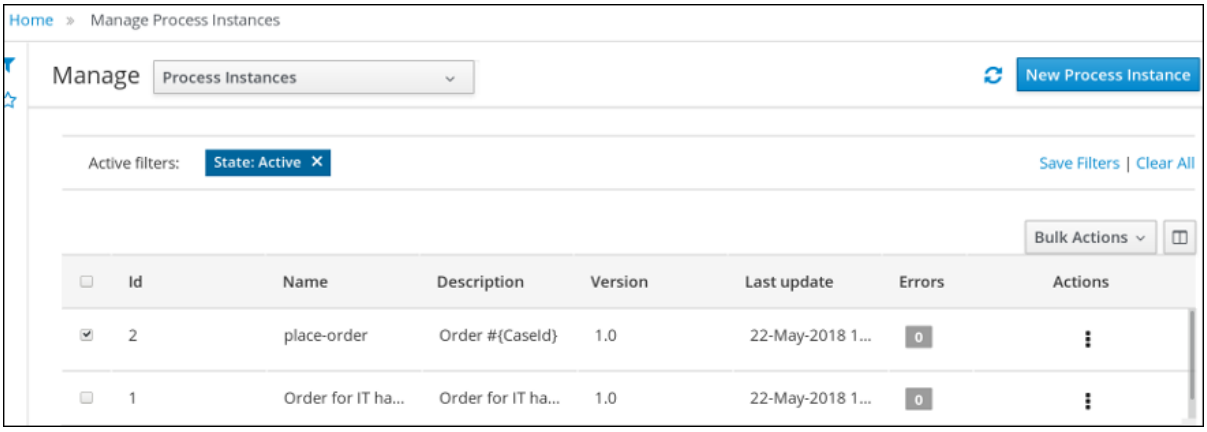
b.

c.

d.

e.

6.


Home » Manage Process Instances
Manage
Process Instances
New Process Instance
Active filters:
State: Active
Save Filters | Clear All
Bulk Actions
Id
Name
Description
Version
Last update
Errors
Actions
2
place-order
Order #{CaseId}
1.0
22-May-2018 1...
0
1
Order for IT ha...
Order for IT ha...
1.0
22-May-2018 1...
0

[D]

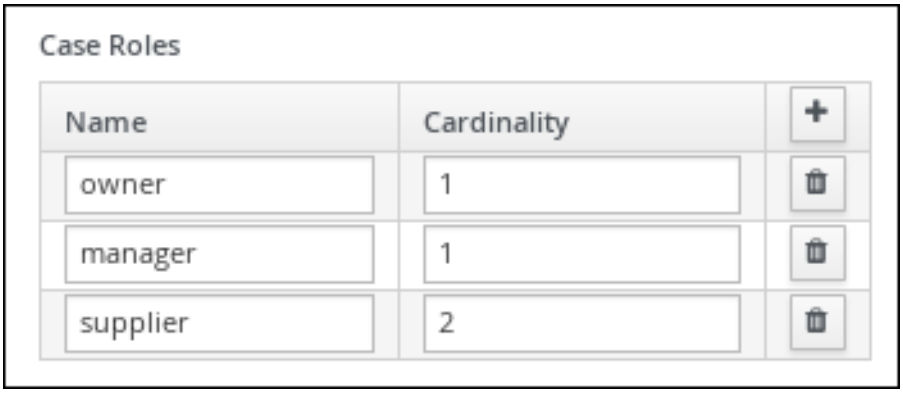
a.

b.

第 55 章

-
-
-

图 55.1.



[D]

先决条件

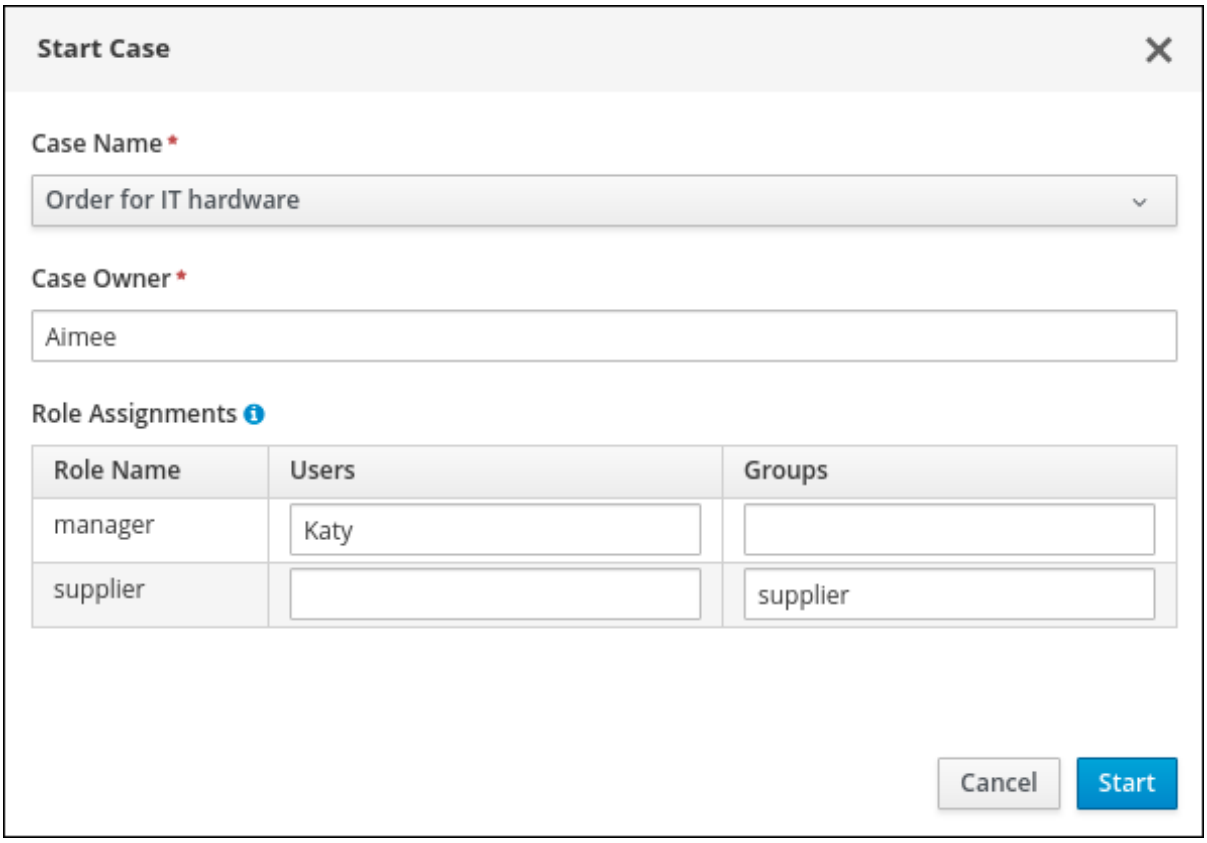
-
-

流程

ルセ

1.

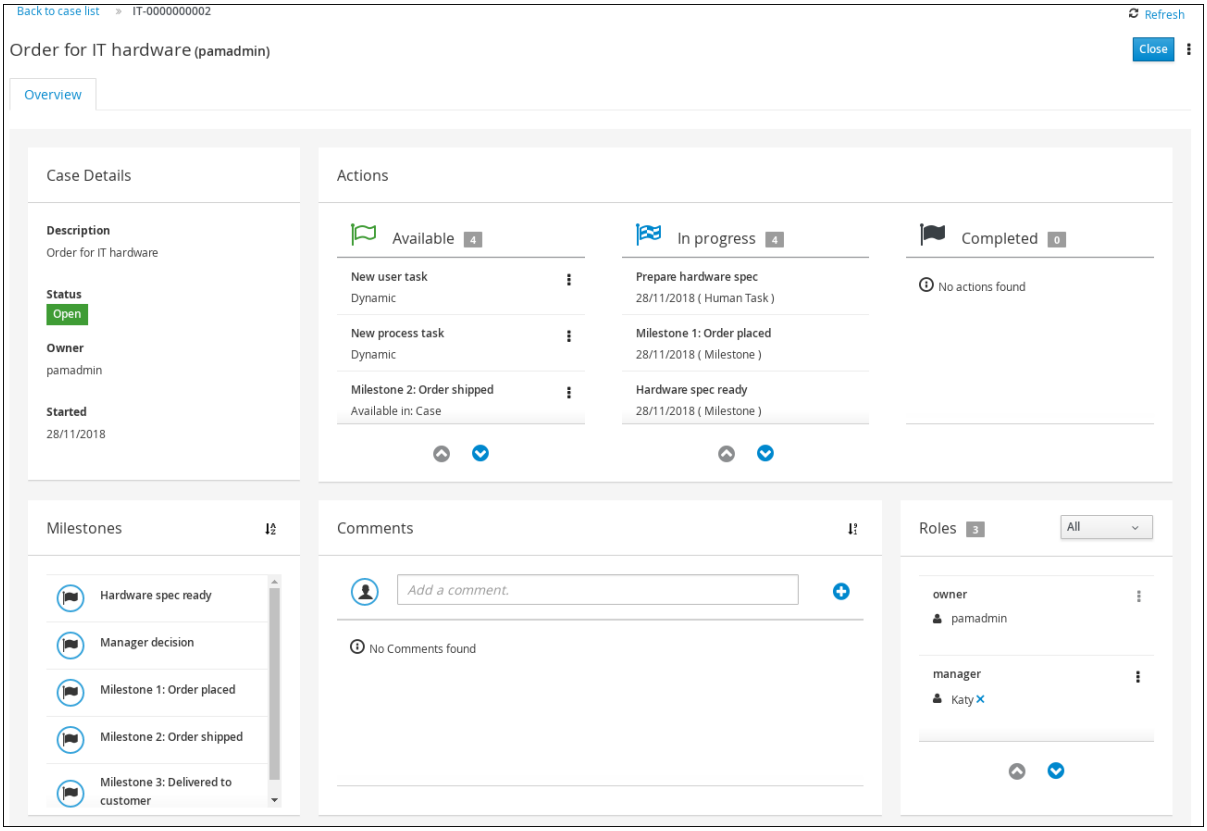
2.


Start Case
Case Name *
Order for IT hardware
Case Owner *
Aimee
Role Assignments
Role Name
Users
Groups
manager
Katy
supplier
supplier
Cancel
Start

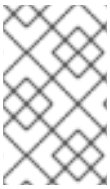
[D]

3.

4.


Back to case list
IT-0000000002
Refresh
Order for IT hardware (pamadmin)
Close
Overview
Case Details
Description
Order for IT hardware
Status
Open
Owner
pamadmin
Started
28/11/2018
Actions
Available 4
New user task
Dynamic
New process task
Dynamic
Milestone 2: Order shipped
Available in: Case
In progress 4
Prepare hardware spec
28/11/2018 (Human Task)
Milestone 1: Order placed
28/11/2018 (Milestone)
Hardware spec ready
28/11/2018 (Milestone)
Completed 0
No actions found
Milestones
Hardware spec ready
Manager decision
Milestone 1: Order placed
Milestone 2: Order shipped
Milestone 3: Delivered to customer
Comments
Add a comment.
No Comments found
Roles 3
All
owner
pamadmin
manager
Katy

[D]



注意

第 56 章

-
-
-
-

先决条件

-

流程

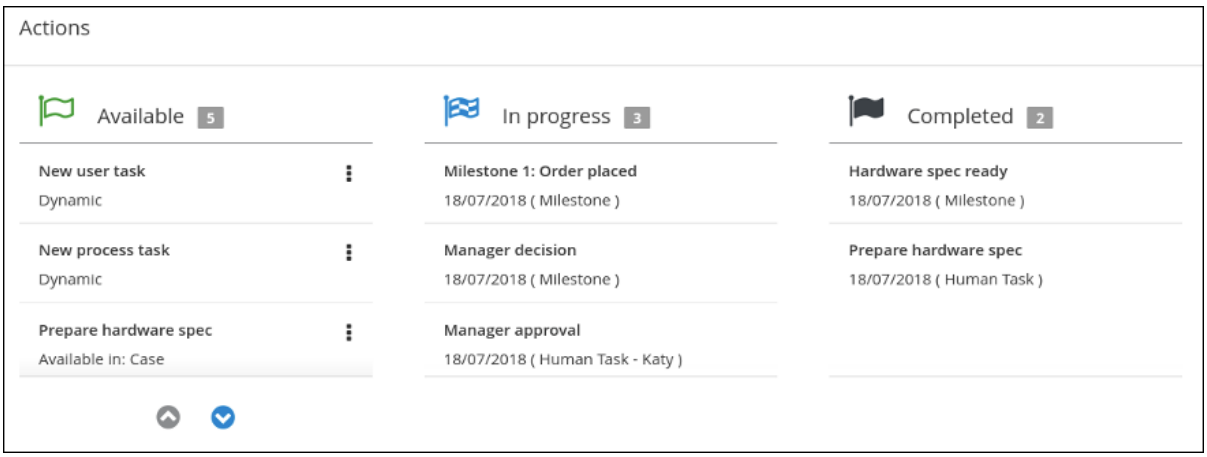
1.
2.
3.
4.


Home » Task Inbox » Task: 1
1 - Prepare hardware spec
Work
Details
Assignments
Comments
Logs
Upload hardware specification *
valid-spec.pdf
Save
Release
Complete

[D]

5.

6.


Actions
Available 5
New user task
Dynamic
New process task
Dynamic
Prepare hardware spec
Available in: Case
In progress 3
Milestone 1: Order placed
18/07/2018 (Milestone)
Manager decision
18/07/2018 (Milestone)
Manager approval
18/07/2018 (Human Task - Katy)
Completed 2
Hardware spec ready
18/07/2018 (Milestone)
Prepare hardware spec
18/07/2018 (Human Task)

[D]

7.

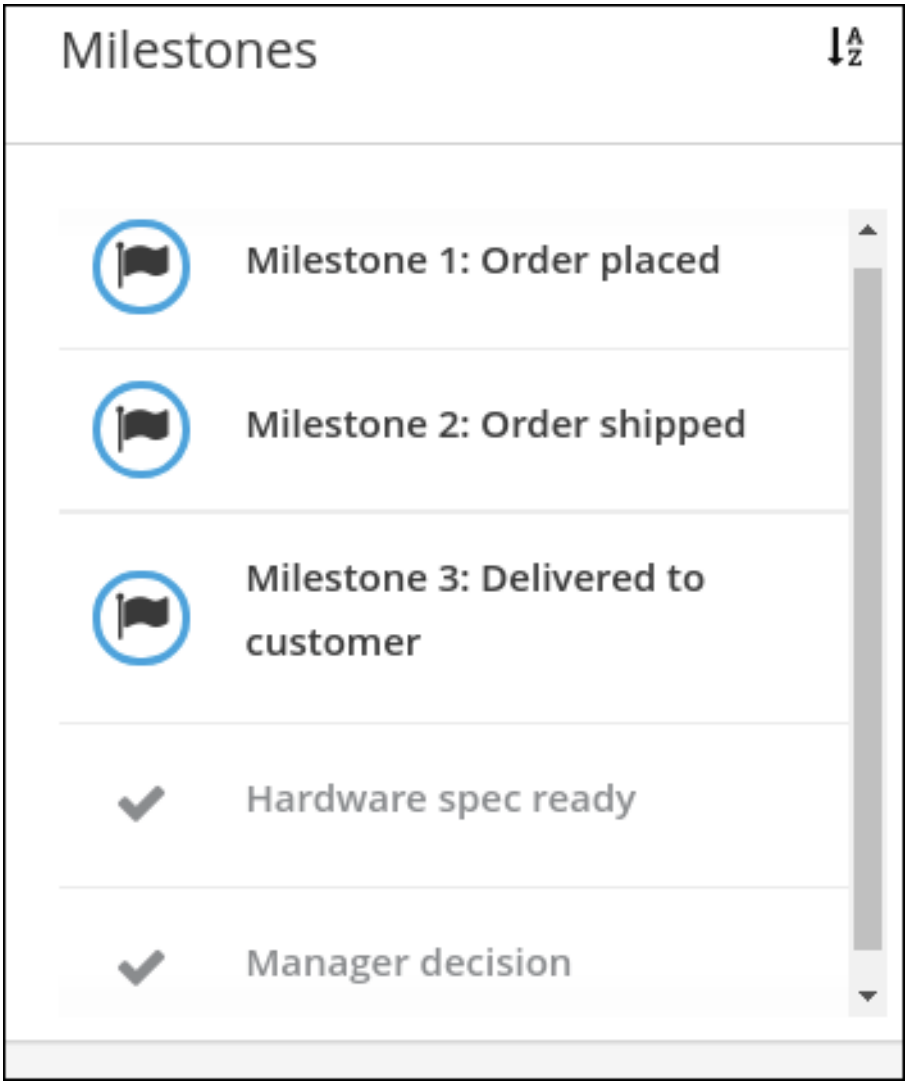
a.

b.

8.

a.

b.

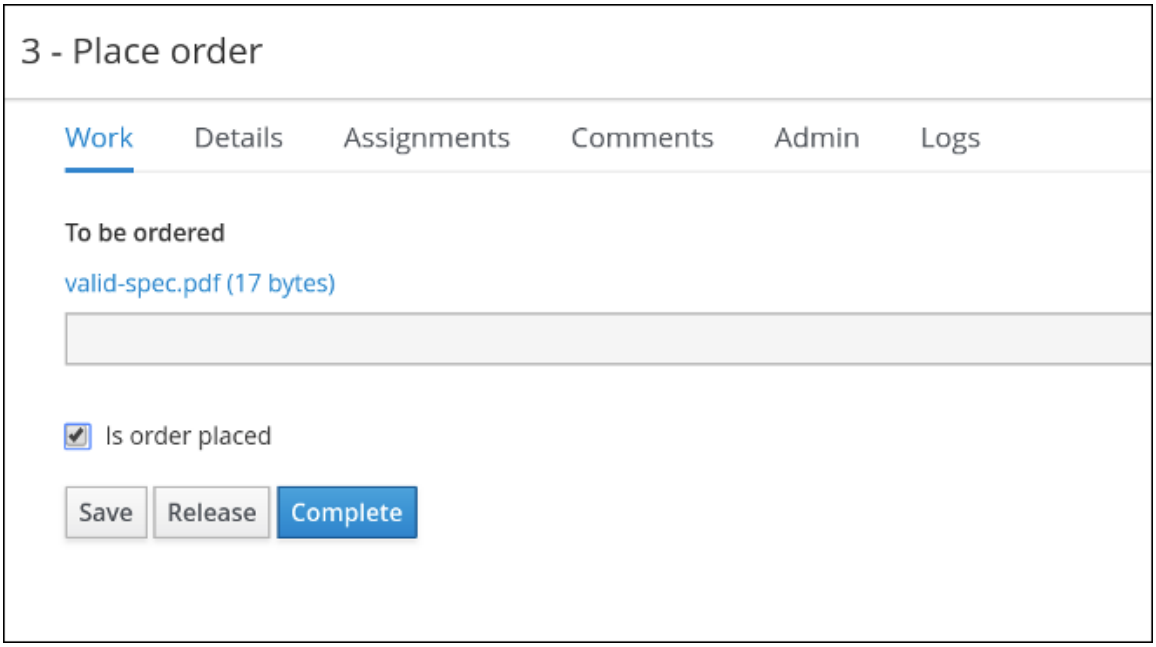

Milestones
Milestone 1: Order placed
Milestone 2: Order shipped
Milestone 3: Delivered to customer
Hardware spec ready
Manager decision

[D]

9.

a.

b.


3 - Place order
Work
Details
Assignments
Comments
Admin
Logs
To be ordered
valid-spec.pdf (17 bytes)
Is order placed
Save
Release
Complete

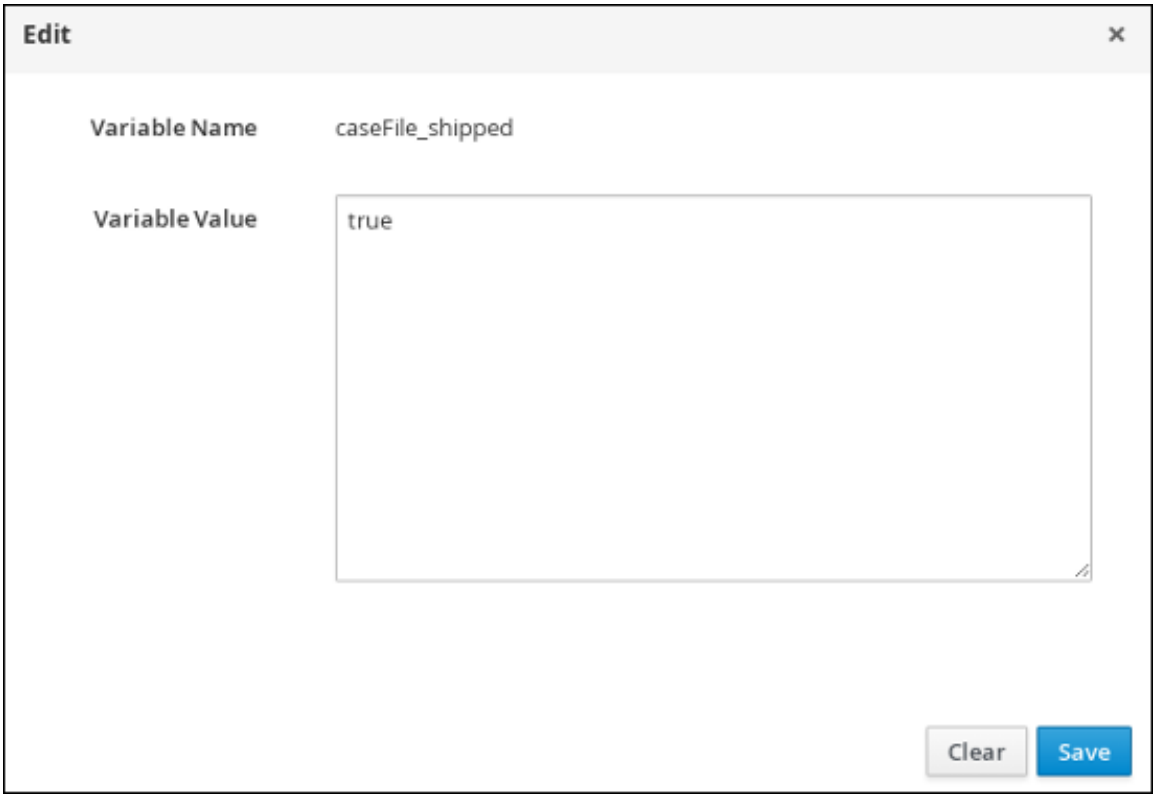
[D]

c.

10.

a.

b.


Edit
Variable Name
caseFile_shipped
Variable Value
true
Clear
Save

[D]

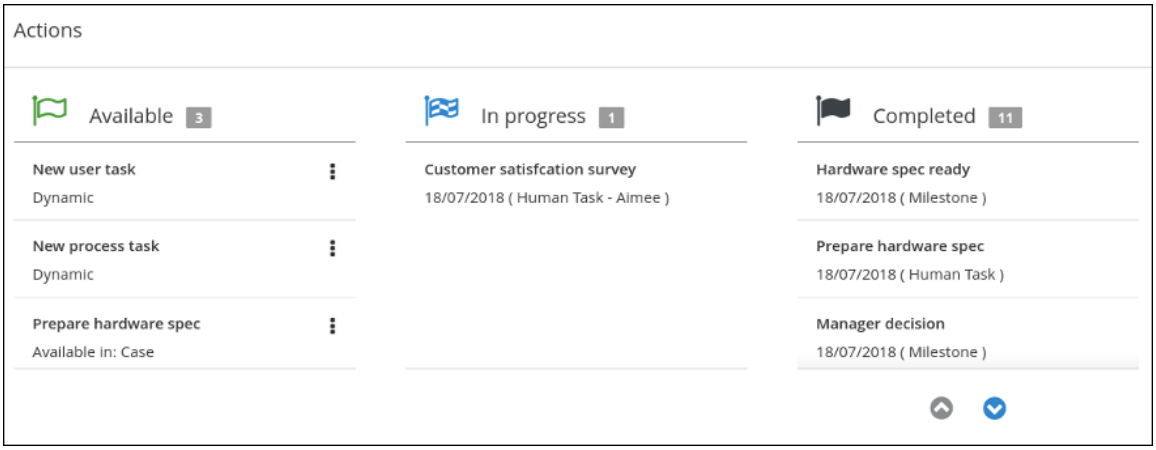
c.

11.

a.

b.

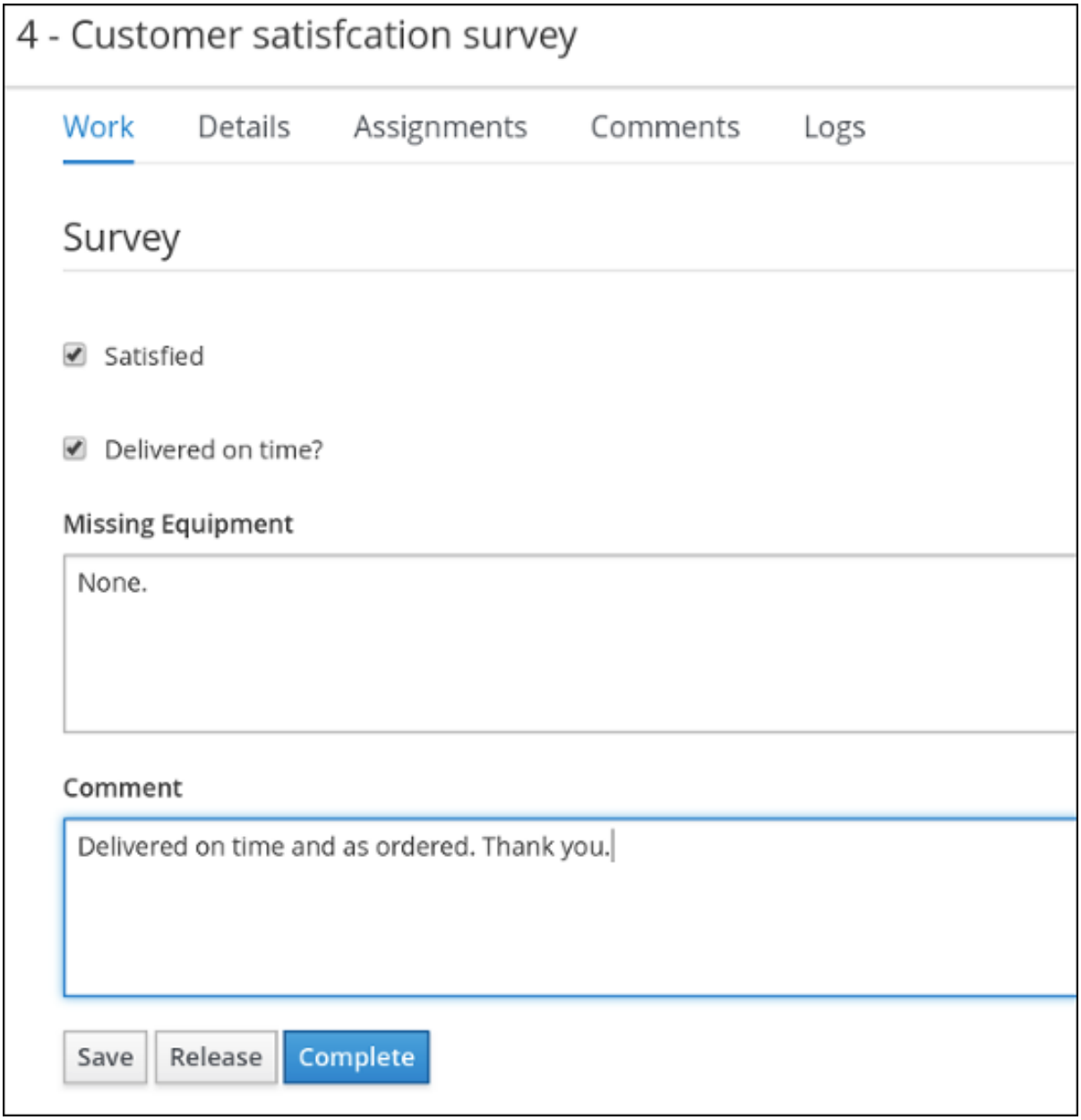
c.


Actions
Available 3
New user task
Dynamic
New process task
Dynamic
Prepare hardware spec
Available in: Case
In progress 1
Customer satisfcation survey
18/07/2018 (Human Task - Aimee)
Completed 11
Hardware spec ready
18/07/2018 (Milestone)
Prepare hardware spec
18/07/2018 (Human Task)
Manager decision
18/07/2018 (Milestone)

[D]

12.

13.


4 - Customer satisfcation survey
Work
Details
Assignments
Comments
Logs
Survey
Satisfied
Delivered on time?
Missing Equipment
None.
Comment
Delivered on time and as ordered. Thank you.
Save
Release
Complete

[D]

14.

15.

a.

b.

16.


Comments
Add a comment.
wbadmin
10/12/2018
All done and ready to close

[D]

17.

第 57 章 其他资源

-

-

部分 VI.

先决条件

-
-
- *已安装 Maven。*
-
-

第 58 章

注意

流程

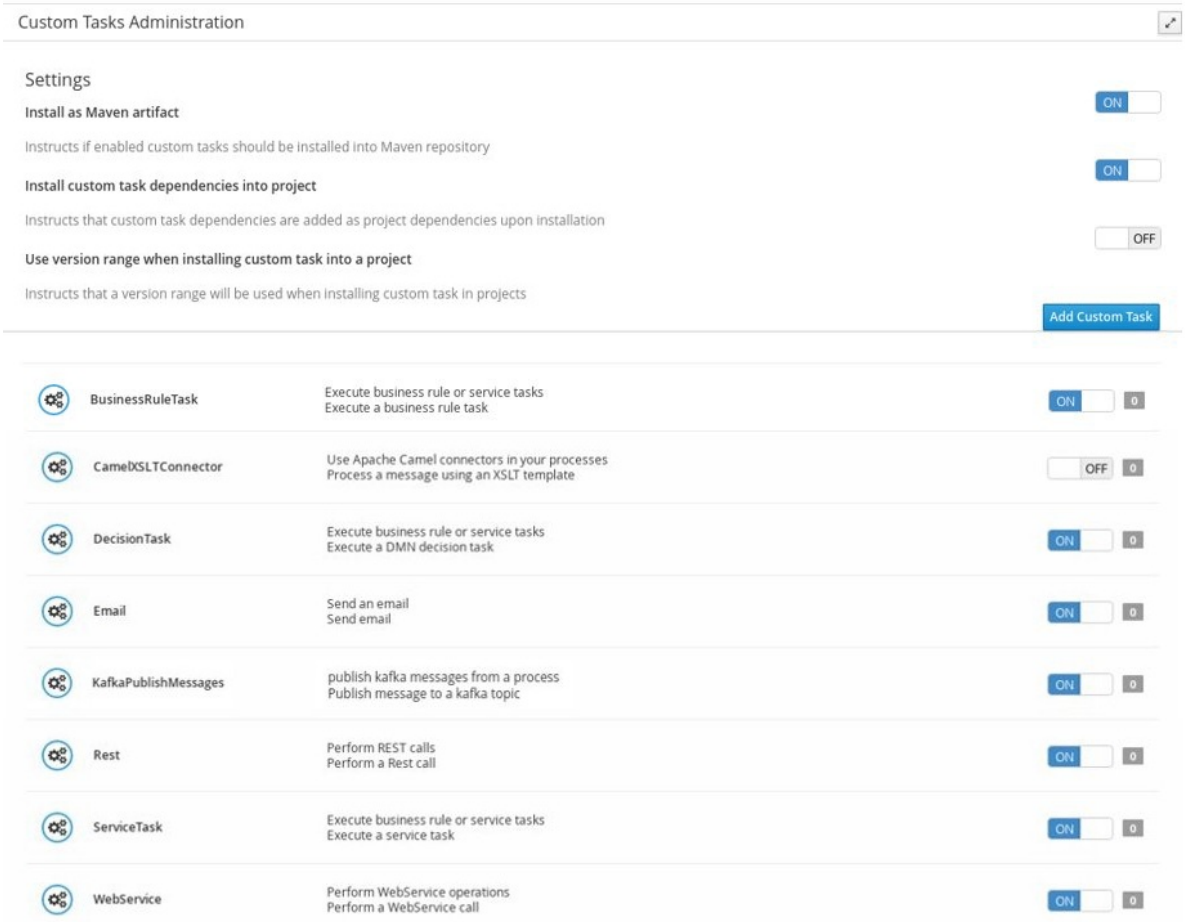
1.

2.

-
-
-

3.

图 58.1.


Custom Tasks Administration
Settings
Install as Maven artifact
Instructs if enabled custom tasks should be installed into Maven repository
ON
Install custom task dependencies into project
Instructs that custom task dependencies are added as project dependencies upon installation
ON
Use version range when installing custom task into a project
Instructs that a version range will be used when installing custom task in projects
OFF
Add Custom Task
BusinessRuleTask
Execute business rule or service tasks
Execute a business rule task
ON 0
CamelXSLTConnector
Use Apache Camel connectors in your processes
Process a message using an XSLT template
OFF 0
DecisionTask
Execute business rule or service tasks
Execute a DMN decision task
ON 0
Email
Send an email
Send email
ON 0
KafkaPublishMessages
publish kafka messages from a process
Publish message to a kafka topic
ON 0
Rest
Perform REST calls
Perform a Rest call
ON 0
ServiceTask
Execute business rule or service tasks
Execute a service task
ON 0
WebService
Perform WebService operations
Perform a WebService call
ON 0

[D]

4.

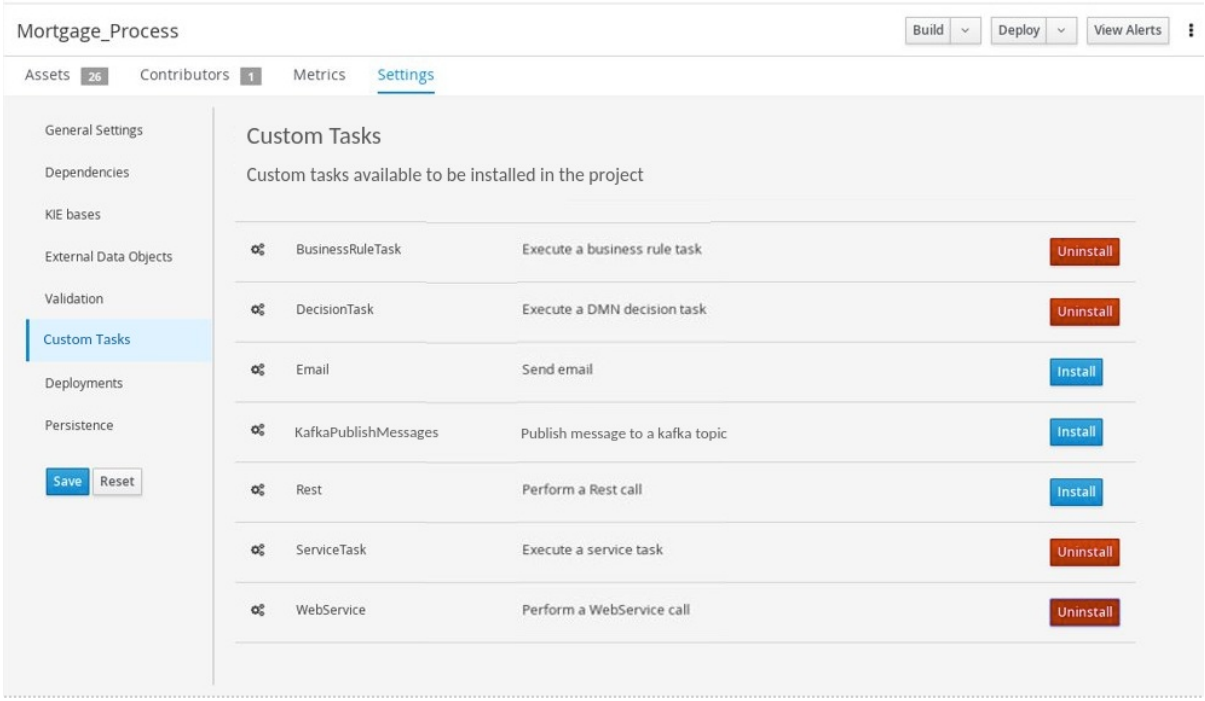
5.

6.

7.

8.

图 58.2.

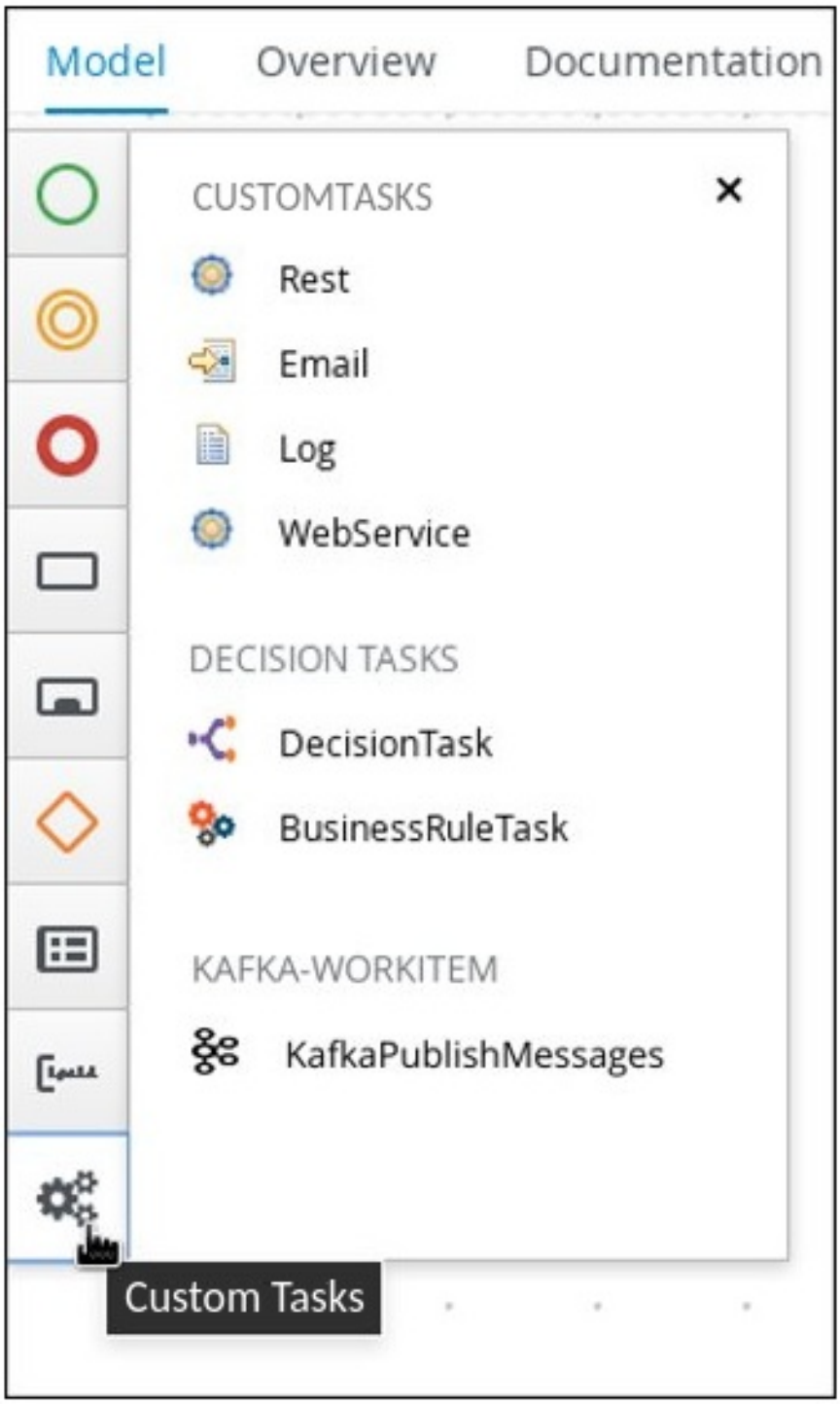


[D]

9. *点击 **Save**。*

10.

图 58.3.


Model
Overview
Documentation
CUSTOMTASKS
Rest
Email
Log
WebService
DECISION TASKS
DecisionTask
BusinessRuleTask
KAFKA-WORKITEM
KafkaPublishMessages
Custom Tasks

[D]

第 59 章

流程

1.

```
$ mkdir workitem-home
```

2.

注意

```
<settings>
  <profiles>
    <profile>
      <id>my-profile</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
        <repository>
          <id>redhat-ga</id>
          <url>http://maven.repository.redhat.com/ga/</url>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
          <releases>
            <enabled>true</enabled>
          </releases>
        </repository>
        ...
      </repositories>
    </profile>
```

```
  </profiles>
  ...
</settings>
```

3.

-
-

4.

```
$ mvn archetype:generate \
-DarchetypeGroupId=org.jbpm \
-DarchetypeArtifactId=jbpm-workitems-archetype \
-DarchetypeVersion=<redhat-library-version> \
-Dversion=1.0.0-SNAPSHOT \
-DgroupId=com.redhat \
-DartifactId=myworkitem \
-DclassPrefix=MyWorkItem
```

表 59.1.

参数	描述
-DarchetypeGroupId	
-DarchetypeArtifactId	
-DarchetypeVersion	
-Dversion	
-DgroupId	
-DartifactId	
-DclassPrefix	

例如：

```
assembly/
```

```
  assembly.xml
src/
  main/
    java/
      com/
        redhat/
          MyWorkItemWorkItemHandler.java
    repository/
    resources/
  test/
    java/
      com/
        redhat/
          MyWorkItemWorkItemHandlerTest.java
          MyWorkItemWorkItemIntegrationTest.java
    resources/
      com/
        redhat/
pom.xml
```

5.

6.

```
$ mvn clean package
```

表 59.2.

参数	描述
myworkitems-<version>.jar	
myworkitems-<version>.zip	

第 60 章

表 60.1.

描述	

```
public class MyWorkItemWorkItemHandler extends AbstractLogOrThrowWorkItemHandler {

 public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
   try {
    RequiredParameterValidator.validate(this.getClass(), workItem);

    // sample parameters
    String sampleParam = (String) workItem.getParameter("SampleParam");
    String sampleParamTwo = (String) workItem.getParameter("SampleParamTwo");

    // complete workitem impl...

    // return results
    String sampleResult = "sample result";
    Map<String, Object> results = new HashMap<String, Object>();
    results.put("SampleResult", sampleResult);
    manager.completeWorkItem(workItem.getId(), results);
   } catch(Throwable cause) {
    handleException(cause);
   }
 }

 @Override
 public void abortWorkItem(WorkItem workItem, WorkItemManager manager) {
   // similar
 }
}
```

表 60.2.

参数	描述
RequiredParameterValidator.validate(this.getClass(), workItem);	
String sampleParam = (String) workItem.getParameter("SampleParam");	
results.put("SampleResult", sampleResult);	
manager.completeWorkItem(workItem.getId(), results);	
abortWorkItem()	

注意

第 61 章

-
-

61.1. @WID ANNOTATION

@Wid Example

```
@Wid(widfile="MyWorkItemDefinitions.wid",
  name="MyWorkItemDefinitions",
  displayName="MyWorkItemDefinitions",
  icon="",
  defaultHandler="mvel: new com.redhat.MyWorkItemWorkItemHandler()",
  documentation = "myworkitem/index.html",
  parameters={
    @WidParameter(name="SampleParam", required = true),
    @WidParameter(name="SampleParamTwo", required = true)
  },
  results={
    @WidResult(name="SampleResult")
  },
  mavenDepends={
    @WidMavenDepends(group="com.redhat",
    artifact="myworkitem",
    version="7.52.0.Final-example-00007")
  },
  serviceInfo={
    @WidService(category = "myworkitem",
    description = "${description}",
    keywords = "",
    action = @WidAction(title = "Sample Title"),
    authinfo = @WidAuth(required = true,
    params = {"SampleParam", "SampleParamTwo"},
    paramsdescription = {"SampleParam", "SampleParamTwo"},
```

```
        referencesite = "referenceSiteURL"))
    }
)
```

表 61.1. @Wid parameter descriptions

	描述
@Wid	
widfile	
name	
displayName	
icon	
description	
defaultHandler	
文档	
@WidParameter	 **注意**
@WidResult	 **注意**
@WidMavenDepends	
@WidService	可选。
@WidAction	
@WidAuth	

61.2.

```
[
  [
    "name" : "MyWorkItemDefinitions",
    "displayName" : "MyWorkItemDefinitions",
    "category" : "",
    "description" : "",
    "defaultHandler" : "mvel: new com.redhat.MyWorkItemWorkItemHandler()",
    "documentation" : "myworkitem/index.html",
    "parameters" : [
      "SampleParam" : new StringDataType(),
      "SampleParamTwo" : new StringDataType()
    ],
    "results" : [
      "SampleResult" : new StringDataType()
    ],
    "mavenDependencies" : [
      "com.redhat:myworkitem:7.52.0.Final-example-00007"
    ],
    "icon" : ""
  ]
]
```

表 61.2.

描述	
name	
displayName	
icon	
category	
description	
defaultHandler	

描述	
文档	
parameters	
results	
mavenDependencies	

-
-

警告

第 62 章

-
-

62.1.

62.2.

流程

1.
2. *点 Upload。*
3.
4.
5.

62.3.

例如：

> *<startup location>/repositories/kie/global/com/redhat/myworkitem/1.0.0-SNAPSHOT/myworkitems-1.0.0-SNAPSHOT.jar*

第 63 章

注意

-
-

63.1.

流程

1.
2.
3.
4.

5.

6.

注意

-

-

-

7.

63.2.

流程

1.

2.

-

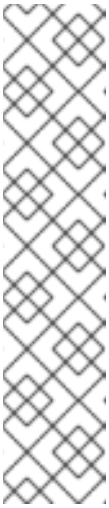
```
<work-item-handler>
  <resolver>mvel</resolver>
  <identifier>new com.redhat.MyWorkItemWorkItemHandler()</identifier>
  <parameters/>
  <name>MyWorkItem</name>
</work-item-handler>
```

-

```
<work-item-handler>
  <resolver>reflection</resolver>
  <identifier>com.redhat.MyWorkItemWorkItemHandler</identifier>
  <parameters/>
  <name>MyWorkItem</name>
</work-item-handler>
```

- *对于 Spring，请添加以下内容并确保标识符是 Spring bean 的标识符：*

```
<work-item-handler>
  <resolver>spring</resolver>
  <identifier>beanIdentifier</identifier>
  <parameters/>
  <name>MyWorkItem</name>
</work-item-handler>
```



注意

如果您使用 Spring 来发现和配置 Spring Bean，则可以使用 org.springframework.stereotype.Component 类注解来自动注册工作项目处理程序。

在工作项目处理程序中，在工作项目处理程序类声明前添加注释 @Component ("<Name >")。例如：@Component ("MyWorkItem")公共类 MyWorkItemWorkItemHandler 扩展 AbstractLogOrThrowWorkItemHandler {

第 64 章 放置自定义任务

当在 Red Hat Process Automation Manager 中注册自定义任务时，它会出现在进程设计程序面板中。自定义任务按照对应的 WID 文件中的条目命名并分类。

先决条件

- 在 Red Hat Process Automation Manager 中注册了一个自定义任务。更多信息请参阅 第 63 章 。
- 自定义任务按照对应的 WID 文件命名并分类。有关 WID 文件位置或格式化的详情，请参考 第 61 章 。

流程

1. 在 Business Central 中，前往 Menu → Design → Projects 并点一个项目。
2. 选择您要添加自定义任务的业务流程。
3. 从面板中选择自定义任务，并将其拖到 BPMN2 图表中。
4. 可选：更改自定义任务属性。例如，更改对应的 WID 文件中的数据输出和输入。

注意

如果 WID 文件不在项目中的 Others 类别中可见，且项目其他类别中不可见 Work Item 定义 对象，您必须注册自定义任务。有关注册自定义任务 第 63 章 的更多信息。

部分 VII. RED HAT PROCESS AUTOMATION MANAGER 中的处理引擎

作为业务流程的分析师或开发人员，您对红帽流程自动化管理器中的流程引擎的理解可以帮助您设计更有效的业务资产和更具扩展性的进程管理架构。流程引擎实施红帽流程自动化管理器中的业务流程管理 (BPM)模式，并管理和执行构成流程的业务资产。本文档描述了在 Red Hat Process Automation Manager 中创建业务流程管理系统和流程服务时，流程引擎的概念和功能。

第 65 章 RED HAT PROCESS AUTOMATION MANAGER 中的处理引擎

流程引擎实施红帽流程自动化管理器中的业务流程管理(BPM)范式。BPM 是一个业务方法，可在企业内实现建模、测量和优化流程。

在 BPM 中，可重复的业务流程以工作流图表示。业务流程模型和符号(BPMN)规范定义此图表的可用元素。进程引擎实现 BPMN 2.0 规范的大子集。

借助流程引擎，业务分析员可以自行开发图。开发人员可实施代码中流的每个元素的业务逻辑，从而成为可执行的业务流程。用户可以执行 业务流程，并根据需要与之交互。分析师可以生成反映流程效率的指标。

工作流图由多个节点组成。BPMN 规格定义了多种节点，包括以下主体类型：

- 事件 ：代表进程或进程外发生的某些节点。典型的事件是进程的开始和结尾。事件可以向其他进程丢弃消息并捕获这些消息。图上的圆形代表事件。
- 活动 ：代表必须采取的操作的节点（无论是自动还是有用户参与）。典型的事件是任务，它代表进程中采取的操作，以及对子进程的调用。图中的轮轮形图代表活动。
- 网关 ：分支或合并节点。典型的网关评估表达式，并根据结果继续其中一个执行路径。图中的诊断图代表网关。

当用户启动进程时，会创建一个进程实例。进程实例包含一组存储在进程变量中的数据或 上下文。进程 实例的状态 包括所有上下文数据，以及当前活动节点（或者在某些情形中，一些活动节点）。

当用户启动进程时，可以初始化其中的一些变量。活动可以从进程变量读取并写入进程变量。网关可以评估进程变量以确定执行路径。

例如，shop 中的购买流程可能是一个业务流程。用户 cart 可以是初始进程上下文。在执行结束时，进程上下文可以包含支付确认和发运跟踪详情。

另外，您可以使用 Business Central 中的 BPMN 数据模型程序为进程变量中数据设计模型。

工作流图由 XML 业务流程定义 以代码表示。事件、网关和子进程调用的逻辑在业务流程定义中定义。

在引擎中实施一些任务类型（例如脚本任务和标准决策引擎规则任务）。对于其他任务类型，包括所有自定义任务，当任务必须执行进程引擎时使用 Work Item Handler API 执行调用。引擎外部的代码可以实施此 API，为实施各种任务提供灵活的机制。

流程引擎包含许多预定义的任务类型。这些类型包括运行用户 Java 代码、调用 Java 方法或 Web 服务的服务任务、调用决策引擎服务的决策任务，以及调用决策引擎服务和其他自定义任务（如 REST 和数据库调用）的脚本任务。

另一种预定义 任务是用户任务，其中包括与用户的交互。进程中的用户任务可以分配给用户和组。

进程引擎使用 KIE API 与其他软件组件交互。您可以在 KIE 服务器上作为服务运行业务流程，并使用 KIE API 的 REST 实现与其交互。或者，您可以将业务流程嵌入到应用程序中，并使用 KIE API Java 调用与它们进行交互。在这种情况下，您可以在任何 Java 环境中运行流程引擎。

Business Central 包括一个用户界面，供用户执行人工任务，以及为人工任务创建 Web 表单的表单建模器。但是，您也可以实施一个自定义用户界面，该界面使用 KIE API 与进程引擎交互。

进程引擎支持以下附加功能：

- 支持使用 JPA 标准对进程信息的持久性。持久性保留每个进程实例的状态和上下文（进程变量中的数据），以便在某些时间重启或停机时它们不会丢失。您可以使用 SQL 数据库引擎来存储持久性信息。
- 用于使用 JTA 标准执行进程元素的可插拔支持。如果您使用 JTA 事务管理器，则业务流程的每个元素都作为事务启动。如果元素没有完成，进程实例的上下文将恢复到元素启动前的状态。
- 支持自定义扩展代码，包括新节点类型和其他进程语言。
- 支持有关各种事件通知的自定义监听程序类。
- 支持将正在运行的进程实例迁移到其进程定义的新版本

流程引擎还可与其他独立的核心服务集成：

- 人工任务服务 可在人工需要参与过程中时管理用户任务。它完全可插拔，默认实施基于 WS-HumanTask 规格。人工任务服务管理任务、任务列表、任务表单以及一些更高级的功能，如升级、委派和基于规则的分配。

- 历史记录日志 可以存储关于在进程引擎中执行所有进程的所有信息。虽然运行时持久性存储所有活动进程实例的当前状态，但需要历史记录日志来确保访问历史信息。历史记录日志包含所有当前和历史状态，了解所有活动和完成的进程实例。您可以使用日志来查询与执行进程实例相关的任何信息，以进行监控和分析。

其他资源

- 使用 BPMN 模型设计业务流程

- 使用 KIE API 与 Red Hat Process Automation Manager 交互

- 公共 KIE API 的 Java 文档

第 66 章 进程引擎的核心引擎 API

流程引擎执行业务流程。要定义流程，您可以创建 业务资产，包括流程定义和自定义任务。

您可以使用 Core Engine API 来加载、执行和管理进程引擎中的进程。

有几种级别的控制：

- 在最低级别，您可以直接创建一个 KIE 基础 和 KIE 会话。KIE 基础代表业务流程中的所有资产。KIE 会话是进程引擎中运行业务流程实例的实体。此级别提供精细的控制，但需要明确声明和配置进程实例、任务处理程序、事件处理程序以及您的代码中的其他进程引擎实体。
- 您可以使用 RuntimeManager 类来管理会话和进程。此类使用可配置策略为所需的进程实例提供会话。它自动配置 KIE 会话和任务服务间的交互。它推断了不再需要的进程引擎实体，确保资源的优化使用。您可以使用流畅的 API 将 RuntimeManager 与必要的业务资产实例化，并配置其环境。
- 您可以使用 Services API 来管理进程的执行。例如，部署服务将业务资产部署到引擎，形成部署单元。进程服务从这个部署单元运行进程。

  如果要在应用程序中嵌入进程引擎，Services API 是最方便的选项，因为它隐藏了配置和管理引擎的内部详情。
- 最后，您可以部署一个 KIE 服务器，从 KJAR 文件并运行进程中加载业务资产。KIE 服务器为加载和管理进程提供了 REST API。您还可以使用 Business Central 管理 KIE 服务器。

  如果您使用 KIE 服务器，则不需要使用内核引擎 API。有关在 KIE 服务器上部署和管理流程的详情，请参考打包和部署 Red Hat Process Automation Manager 项目。

有关所有公共流程引擎 API 调用的完整参考信息，请参阅 Java 文档。其他 API 类也存在于代码中，但它们是可以在以后的版本中更改的内部 API。在您开发和维护的应用程序中使用公共 API。

66.1. KIE 基础和 KIE 会话

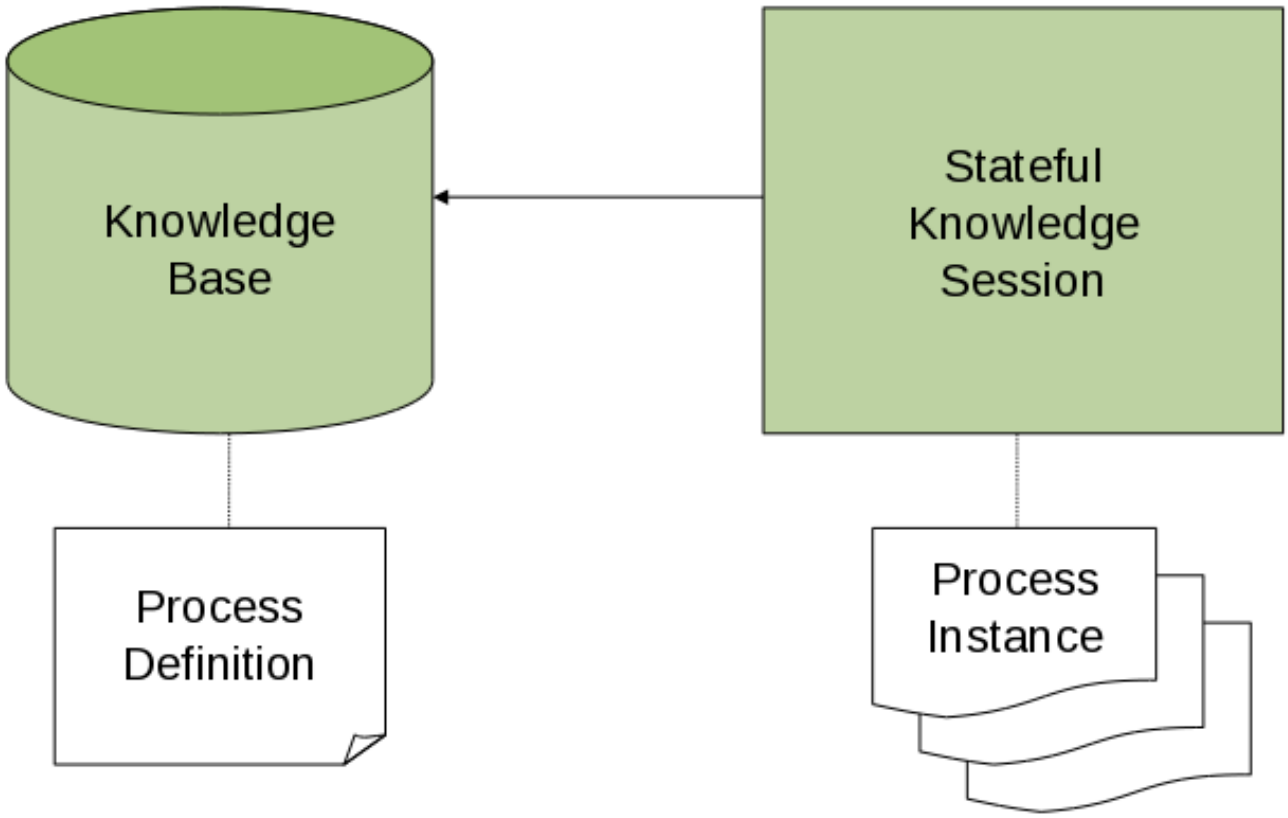
KIE 基础 包含对进程相关的所有进程定义和其他资产的引用。该引擎将使用此 KIE 基础查找流程的所有信息，或根据需要查找多个进程。

您可以从各种源（如类路径、文件系统或进程存储库）将资产加载到 KIE 库中。创建 KIE 基础是一个资源中心操作，因为它涉及从各种源加载和解析资产。您可以动态修改 KIE 基础，在运行时添加或删除进程定义和其他资产。

创建 KIE 基础后，您可以根据这个 KIE 基础实例化 KIE 会话。使用此 KIE 会话，根据 KIE 库中的定义运行进程。

当您使用 KIE 会话启动进程时，会创建一个新 进程实例。此实例维护了一个特定的进程状态。同一 KIE 会话中的不同实例可以使用相同的进程定义，但具有不同的状态。

图 66.1. 进程引擎中的 KIE 基础和 KIE 会话



例如，如果您开发一个应用程序来处理销售订购，您可以创建一个或多个进程定义来确定如何处理订购。启动应用程序时，首先需要创建一个包含这些进程定义的 KIE 基础。然后，您可以基于这个 KIE 基础创建一个会话。当新的销售订单进入时，按顺序启动新的进程实例。这个过程实例包含特定销售请求的进程状态。

您可以为相同的 KIE 基础创建多个 KIE 会话，并可在同一 KIE 会话中创建进程的多个实例。创建 KIE 会话，并在 KIE 会话中创建进程实例使用的资源比创建 KIE 基础少。如果您修改 KIE 基础，则使用它的所有 KIE 会话都可以自动使用修改。

在大多数简单的用例中，您可以使用单个 KIE 会话来执行所有进程。如果需要，也可以使用多个会话。例如，如果想要让不同客户订购处理完全独立，则可以为每个客户创建一个 KIE 会话。出于可扩展性的原因，您还可以使用多个会话。

在典型的应用程序中，您不需要直接创建 KIE 基础或 KIE 会话。但是，在使用进程引擎 API 的其他级别时，您可以和此级别定义的 API 元素进行交互。

66.1.1. KIE 基础

KIE 基础包括应用程序执行业务流程可能需要的所有流程定义和其他资产。

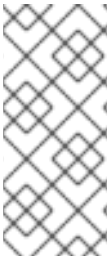
要创建 KIE 基础，请使用 KieHelper 实例从各种资源（如类路径或文件系统）加载进程，并创建新的 KIE 基础。

以下代码片段演示了如何创建只包含一个进程定义（从类路径加载）的 KIE 基础。

创建包含进程定义的 KIE 基础

```
KieHelper kieHelper = new KieHelper();
KieBase kieBase = kieHelper
  .addResource(ResourceFactory.newClassPathResource("MyProcess.bpmn"))
  .build();
```

ResourceFactory 类与从文件加载资源、URL、InputStream、Reader 和其它来源的方法类似。



注意

创建 KIE 基础的这种"手动"流程比其它其它替代方案简单，但会使应用程序难以维护。对于您希望在长时间内开发和维护的应用程序，使用其他创建 KIE 基础（如 RuntimeManager 类或 Services API）的方法。

66.1.2. KIE 会话

创建并加载 KIE 基础后，您可以创建一个 KIE 会话与进程引擎交互。您可以使用此会话启动和管理进程以及信号事件。

以下代码片段基于之前创建的 KIE 基础创建一个会话，然后启动进程实例，并引用进程定义中的 ID。

创建 KIE 会话并启动进程实例

```
KieSession ksession = kbase.newKieSession();
ProcessInstance processInstance = ksession.startProcess("com.sample.MyProcess");
```

66.1.3. ProcessRuntime 接口

KieSession 类会公开 ProcessRuntime 接口，它定义与进程交互的所有会话方法，如下所示：

ProcessRuntime 接口的定义

```
/**
* Start a new process instance.  Use the process (definition) that
* is referenced by the given process ID.
*
* @param processId  The ID of the process to start
* @return the ProcessInstance that represents the instance of the process that was started
*/
  ProcessInstance startProcess(String processId);

  /**
* Start a new process instance.  Use the process (definition) that
* is referenced by the given process ID.  You can pass parameters
* to the process instance as name-value pairs, and these parameters set
* variables of the process instance.
 *
* @param processId  the ID of the process to start
 * @param parameters  the process variables to set when starting the process instance
* @return the ProcessInstance that represents the instance of the process that was started
   */
  ProcessInstance startProcess(String processId,
                               Map<String, Object> parameters);

  /**
   * Signals the process engine that an event has occurred. The type parameter defines
   * the type of event and the event parameter can contain additional information
```

```
     * related to the event.  All process instances that are listening to this type
     * of (external) event will be notified.  For performance reasons, use this type of
     * event signaling only if one process instance must be able to notify
     * other process instances. For internal events within one process instance, use the
     * signalEvent method that also include the processInstanceId of the process instance
     * in question.
     *
     * @param type the type of event
     * @param event the data associated with this event
     */
    void signalEvent(String type,
                     Object event);

    /**
     * Signals the process instance that an event has occurred. The type parameter defines
     * the type of event and the event parameter can contain additional information
     * related to the event.  All node instances inside the given process instance that
     * are listening to this type of (internal) event will be notified.  Note that the event
     * will only be processed inside the given process instance.  All other process instances
     * waiting for this type of event will not be notified.
     *
     * @param type the type of event
     * @param event the data associated with this event
     * @param processInstanceId the id of the process instance that should be signaled
     */
    void signalEvent(String type,
                     Object event,
                     long processInstanceId);

    /**
     * Returns a collection of currently active process instances.  Note that only process
     * instances that are currently loaded and active inside the process engine are returned.
     * When using persistence, it is likely not all running process instances are loaded
     * as their state is stored persistently.  It is best practice not to use this
     * method to collect information about the state of your process instances but to use
     * a history log for that purpose.
     *
     * @return a collection of process instances currently active in the session
     */
    Collection<ProcessInstance> getProcessInstances();

    /**
     * Returns the process instance with the given ID.  Note that only active process instances
     * are returned. If a process instance has been completed already, this method returns
     * null.
     *
     * @param id the ID of the process instance
     * @return the process instance with the given ID, or null if it cannot be found
     */
    ProcessInstance getProcessInstance(long processInstanceId);

    /**
     * Aborts the process instance with the given ID. If the process instance has been
completed
     * (or aborted), or if the process instance cannot be found, this method will throw an
     * IllegalArgumentException.
```

```
     *
     * @param id the ID of the process instance
     */
    void abortProcessInstance(long processInstanceId);

    /**
     * Returns the WorkItemManager related to this session. This object can be used to
     * register new WorkItemHandlers or to complete (or abort) WorkItems.
     *
     * @return the WorkItemManager related to this session
     */
    WorkItemManager getWorkItemManager();
```

66.1.4. 关联密钥

在使用进程时，您可能需要将业务标识符分配给进程实例，然后使用标识符来引用实例，而不存储生成的实例 ID。

为提供这样的功能，流程引擎使用 CorrelationKey 接口，它可定义 CorrelationProperties。实施 CorrelationKey 的类可以具有描述它的单个属性，也可以是多功能集。属性的值或多个属性的值组合指的是唯一的实例。

KieSession 类实施 CorrelationAwareProcessRuntime 接口，以支持关联功能。这个接口会公开以下方法：

CorrelationAwareProcessRuntime 接口方法

```
    /**
     * Start a new process instance.  Use the process (definition) that
     * is referenced by the given process ID.  You can pass parameters
     * to the process instance (as name-value pairs), and these parameters set
     * variables of the process instance.
     *
     * @param processId  the ID of the process to start
     * @param correlationKey custom correlation key that can be used to identify the process
instance
     * @param parameters  the process variables to set when starting the process instance
     * @return the ProcessInstance that represents the instance of the process that was started
     */
    ProcessInstance startProcess(String processId, CorrelationKey correlationKey,
Map<String, Object> parameters);

    /**
```

```
    * Create a new process instance (but do not yet start it).  Use the process
    * (definition) that is referenced by the given process ID.
    * You can pass to the process instance (as name-value pairs),
    * and these parameters set variables of the process instance.
    * Use this method if you need a reference to the process instance before actually
    * starting it.  Otherwise, use startProcess.
    *
    * @param processId  the ID of the process to start
    * @param correlationKey custom correlation key that can be used to identify the process
instance
    * @param parameters  the process variables to set when creating the process instance
    * @return the ProcessInstance that represents the instance of the process that was
created (but not yet started)
    */
    ProcessInstance createProcessInstance(String processId, CorrelationKey correlationKey,
Map<String, Object> parameters);

    /**
    * Returns the process instance with the given correlationKey.  Note that only active
process instances
    * are returned.  If a process instance has been completed already, this method will return
    * null.
    *
    * @param correlationKey the custom correlation key assigned when the process instance
was created
    * @return the process instance identified by the key or null if it cannot be found
    */
    ProcessInstance getProcessInstance(CorrelationKey correlationKey);
```

关联通常用于长时间运行的进程。如果要永久存储关联信息，您必须启用持久性。

66.2. 运行时管理器

RuntimeManager 类在进程引擎 API 中提供一个层，可简化并赋予其使用情况。此类封装和管理 KIE 基础和 KIE 会话，以及为进程中的所有任务提供处理程序的任务服务。运行时管理器中的 KIE 会话和任务服务已配置为彼此使用，您不需要提供这样的配置。例如，您不需要注册人工任务处理程序，并确保它连接到所需的服务。

运行时管理器根据预定义的策略管理 KIE 会话。可用的策略如下：

- singleton ：运行时管理器维护一个 KieSession，并将其用于所有请求的进程。

- 每个请求 ：运行时管理器为每个请求创建一个新的 KieSession。

- 每个进程 实例 ：运行时管理器在进程实例和 KieSession 之间维护映射，每当使用给定进程实例时始终提供相同的 KieSession。

无论策略是什么，RuntimeManager 类都在进程引擎组件初始化和配置确保相同的功能：

- KieSession 实例使用相同的工厂（基于内存或 JPA）加载的实例。

- work item 处理程序在每个 KieSession 实例中注册（从数据库加载或新创建）。

- 事件监听程序（进程、Agenda 和WorkingMemory）在每个 KIE 会话中注册，无论会话是否从数据库加载，还是从数据库加载。

- 任务服务被配置为以下所需组件：

  - JTA 事务管理器

  - 与用于 KieSession 实例使用的实体管理器工厂相同

  - 可以在环境中配置的 UserGroupCallback 实例

运行时管理器还启用对进程引擎的完全处理。它提供了在不再需要时分散 RuntimeEngine 实例的专用方法，释放它可能获取的所有资源。

以下代码显示了 RuntimeManager 接口的定义：

RuntimeManager 接口的定义

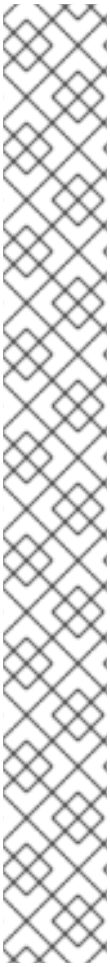
```
public interface RuntimeManager {
```

```
 /**
  * Returns a <code>RuntimeEngine</code> instance that is fully initialized:
  * <ul>
  *  <li>KieSession is created or loaded depending on the strategy</li>
  *  <li>TaskService is initialized and attached to the KIE session (through a listener)</li>
  *  <li>WorkItemHandlers are initialized and registered on the KIE session</li>
  *  <li>EventListeners (process, agenda, working memory) are initialized and added to the KIE
session</li>
  * </ul>
  * @param context the concrete implementation of the context that is supported by given
<code>RuntimeManager</code>
  * @return instance of the <code>RuntimeEngine</code>
  */
   RuntimeEngine getRuntimeEngine(Context<?> context);

   /**
    * Unique identifier of the <code>RuntimeManager</code>
    * @return
    */
   String getIdentifier();

   /**
    * Disposes <code>RuntimeEngine</code> and notifies all listeners about that fact.
    * This method should always be used to dispose <code>RuntimeEngine</code> that is not
needed
    * anymore. <br/>
    * Do not use KieSession.dispose() used with RuntimeManager as it will break the internal
    * mechanisms of the manager responsible for clear and efficient disposal.<br/>
    * Disposing is not needed if <code>RuntimeEngine</code> was obtained within an active
JTA transaction,
    * if the getRuntimeEngine method was invoked during active JTA transaction, then
disposing of
    * the runtime engine will happen automatically on transaction completion.
    * @param runtime
    */
   void disposeRuntimeEngine(RuntimeEngine runtime);

   /**
    * Closes <code>RuntimeManager</code> and releases its resources. Call this method
when
    * a runtime manager is not needed anymore. Otherwise it will still be active and operational.
    */
   void close();

}
```

RuntimeManager 类还提供 RuntimeEngine 类，它包括访问底层进程引擎组件的方法：

RuntimeEngine 接口的定义

```
public interface RuntimeEngine {

 /**
  * Returns the <code>KieSession</code> configured for this <code>RuntimeEngine</code>
  * @return
  */
   KieSession getKieSession();

    /**
  * Returns the <code>TaskService</code> configured for this <code>RuntimeEngine</code>
  * @return
  */
   TaskService getTaskService();
}
```



注意

RuntimeManager 类的标识符在运行时执行过程中用作 deploymentId。例如，当 任务 持久化时，标识符会被保留为 任务的 deploymentId。任务的 deploymentID 在任务完成时将其与 RuntimeManager 关联，进程实例会恢复。

相同的 deploymentId 也作为 externalId 在历史记录日志表中保留。

如果您在创建 RuntimeManager 实例时不指定标识符，则根据策略应用默认值（例如，对于 PerProcessInstanceRuntimeManager）。这意味着您的应用程序在其整个生命周期中使用与 RuntimeManager 类相同的部署。

如果在应用程序中维护多个运行时管理器，您必须为每个 RuntimeManager 实例指定一个唯一标识符。

例如，部署服务维护多个运行时管理器，并使用 KJAR 文件的 GAV 值作为标识符。相同的逻辑在 Business Central 和 KIE 服务器中使用，因为它们依赖于部署服务。

注意

当您需要从处理程序或侦听器中与进程引擎或任务服务交互时，您可以使用 RuntimeManager 界面获取给定进程的 RuntimeEngine 实例，然后使用 RuntimeEngine 实例来检索 KieSession 或 TaskService 实例。此方法可确保引擎的正确状态根据所选的策略进行管理。

66.2.1. 运行时管理器策略

RuntimeManager 类支持以下管理 KIE 会话策略。

单例策略

此策略指示运行时管理器维护单一 RuntimeEngine 实例（并依次使用单个 KieSession 和 TaskService 实例）。对运行时引擎的访问会被同步，因此线程安全，但因同步而出现性能损失。

对此策略用于简单用例。

此策略具有以下特征：

- 它的内存占用少，具有运行时引擎的单一实例和任务服务。
- 在设计和使用方面简单而紧凑。
- 由于同步访问，它非常适合流程引擎上低到中等负载。
- 在这个策略中，因为单个 KieSession 实例，所有状态对象（如事实）都直接对所有进程实例可见，反之亦然。
- 该策略不是上下文。从单例 RuntimeManager 检索 RuntimeEngine 实例时，您不需要将 Context 实例考虑在内。通常，您可以将 EmptyContext.get（） 用作上下文，虽然还接受 null 参数。
- 在这个策略中，运行时管理器会跟踪 KieSession 的 ID，以便在 RuntimeManager 重启后使用相同的会话。ID 做为一个序列化的文件存储在文件系统中，具体取决于环境，可以是以下目录之一：

- *jbpm.data.dir 系统属性的值*
- *jboss.server.data.dir 系统属性的值*
- *java.io.tmpdir 系统属性的值*

警告

Singleton 策略和 EJB Timer 调度程序的组合可能会引发 Hibernate 问题。不要在生产环境中使用这个组合。EJB 计时器调度程序是 KIE 服务器中的默认调度程序。

每个请求策略

此策略指示运行时管理器为每个请求提供一个新的 RuntimeEngine 实例。在一个事务中对进程引擎执行一个或多个调用被视为一个请求。

必须在单一事务中使用同一个 RuntimeEngine 实例，以确保状态正确。否则，在下一个调用中无法看到一个调用中的操作。

这个策略是无状态的，因为进程状态仅在请求内保留。当请求完成后，RuntimeEngine 实例会被永久销毁。如果使用持久性，与 KIE 会话相关的信息也会从持久性数据库中删除。

此策略具有以下特征：

- *它为每一请求提供完全隔离的进程引擎和任务服务操作。*
- *它是完全无状态的，因为事实仅存储在请求的持续时间内。*
- *它非常适合高负载、无状态进程，在请求之间不能保留事实或计时器。*

- 在这个策略中，KIE 会话仅在请求生命周期中可用，并在请求结束时销毁。

- 该策略不是上下文。从每个请求的 RuntimeManager 检索 RuntimeEngine 实例时，您不需要考虑 Context 实例。通常，您可以将 EmptyContext.get（） 用作上下文，虽然还接受 null 参数。

每个进程实例策略

此策略指示 RuntimeManager 在 KIE 会话和进程实例之间保持严格的关系。每个 KieSession 都可用，只要它所属的 ProcessInstance 才可用。

此策略提供了使用流程引擎的高级功能的最灵活方法，如进程实例之间的规则评估和隔离。它可最大化性能并减少同步带来的潜在瓶颈。同时，与请求策略不同，它会将 KIE 会话的数量减少到实际进程实例数，而不是请求总数。

此策略具有以下特征：

- 它为每个进程实例提供隔离。

- 它在 KieSession 和 ProcessInstance 之间有一个严格的关系，以确保它始终为给定进程实例 提供相同的 KieSession。

- 它将 KieSession 与 ProcessInstance 的生命周期合并，当进程实例完成或中止时，它们都被处理。

- 它可在进程实例范围内维护数据，如事实和计时器。只有进程实例有权访问数据。

- 它引入了一些开销，因为需要查找和加载进程实例的 KieSession。

- 它将验证每个 KieSession 的使用，使其不能用于其他进程实例。如果另一个进程实例使用相同的 KieSession，则抛出异常。

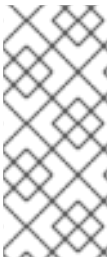
- 该策略是上下文，接受以下上下文实例：

- EmptyContext 或 null: 在启动进程实例时使用，因为还没有可用的进程实例 ID
- ProcessInstanceIdContext: 在创建进程实例后使用
- CorrelationKeyContext: 用作 ProcessInstanceIdContext 的替代选择，使用自定义（业务）密钥而不是进程实例 ID

66.2.2. 运行时管理器的典型用法场景

运行时管理器的典型用法场景由以下阶段组成：

- 在应用程序启动时，完成以下阶段：
  - 构建一个 RuntimeManager 实例，并在应用程序的整个生命周期内保留它，因为它是 thread-safe，并可同时访问。
- 在请求时，完成以下阶段：
  - 从 RuntimeManager 获取 RuntimeEngine，使用正确的上下文实例由您为 RuntimeManager 类配置的策略决定。
  - 从 RuntimeEngine 中获取 KieSession 和 TaskService 对象。
  - 将 KieSession 和 TaskService 对象用于操作，如 startProcess 或 completeTask。
  - 完成处理后，使用 Runtime Manager.dispose RuntimeEngine 方法处理。
- 在应用程序关闭时，完成以下阶段：
  - 关闭 RuntimeManager 实例。



注意

当从活跃的 JTA 事务中的 RuntimeEngine 获得 RuntimeEngine 时，您不需要在结束时退出 RuntimeEngine，因为 RuntimeManager 会自动对事务完成处理 RuntimeEngine （禁止完成状态：提交或回滚）。

以下示例演示了如何构建 RuntimeManager 实例，并获取 RuntimeEngine 实例（封装了 KieSession 和 TaskService 类）：

构建 RuntimeManager 实例，然后获取 RuntimeEngine 和 KieSession

```
// First, configure the environment to be used by RuntimeManager
RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
.newDefaultInMemoryBuilder()
.addAsset(ResourceFactory.newClassPathResource("BPMN2-ScriptTask.bpmn2"),
ResourceType.BPMN2)
.get();

// Next, create the RuntimeManager - in this case the singleton strategy is chosen
RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newSingletonRuntimeManager(environment);

// Then get RuntimeEngine from the runtime manager, using an empty context because
singleton does not keep track
// of runtime engine as there is only one
RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());

// Get the KieSession from the RuntimeEngine - already initialized with all handlers,
listeners, and other requirements
// configured on the environment
KieSession ksession = runtimeEngine.getKieSession();

// Add invocations of the process engine here,
// for example, ksession.startProcess(processId);

// Finally, dispose the runtime engine
manager.disposeRuntimeEngine(runtimeEngine);
```

这个示例提供了使用 RuntimeManager 和 RuntimeEngine 类的最简单或最小方法。它具有以下特征：

- KieSession 实例创建在内存中，使用 newDefaultInMemoryBuilder 构建器。

- 单个进程（作为资产添加）可用于执行。

- TaskService 类通过 LocalHTWorkItemHandler 接口配置并附加到 KieSession 实例，以支持进程中的用户任务功能。

66.2.3. 运行时环境配置对象

RuntimeManager 类封装了内部进程引擎复杂性，如创建、处理和注册处理程序。

它还提供对流程引擎配置的精细控制。要设置此配置，您必须创建一个 RuntimeEnvironment 对象，然后使用它创建 RuntimeManager 对象。

以下定义显示了 RuntimeEnvironment 接口中可用的方法：

RuntimeEnvironment 接口中的方法

```
public interface RuntimeEnvironment {

/**
 * Returns <code>KieBase</code> that is to be used by the manager
 * @return
 */
  KieBase getKieBase();

   /**
    * KieSession environment that is to be used to create instances of
<code>KieSession</code>
    * @return
    */
   Environment getEnvironment();

   /**
    * KieSession configuration that is to be used to create instances of
<code>KieSession</code>
    * @return
    */
   KieSessionConfiguration getConfiguration();

   /**
```

```
    * Indicates if persistence is to be used for the KieSession instances
    * @return
    */
   boolean usePersistence();

   /**
    * Delivers a concrete implementation of <code>RegisterableItemsFactory</code> to obtain
handlers and listeners
    * that is to be registered on instances of <code>KieSession</code>
    * @return
    */
   RegisterableItemsFactory getRegisterableItemsFactory();

   /**
    * Delivers a concrete implementation of <code>UserGroupCallback</code> that is to be
registered on instances
    * of <code>TaskService</code> for managing users and groups.
    * @return
    */
   UserGroupCallback getUserGroupCallback();

   /**
    * Delivers a custom class loader that is to be used by the process engine and task service
instances
    * @return
    */
   ClassLoader getClassLoader();

   /**
    * Closes the environment, permitting closing of all dependent components such as
ksession factories
    */
   void close();
```

66.2.4. 运行时环境构建程序

要创建包含所需数据的 RuntimeEnvironment 实例，请使用 RuntimeEnvironmentBuilder 类。此类提供了一个流畅的 API，用于通过预定义的设置配置 RuntimeEnvironment 实例。

以下定义显示了 RuntimeEnvironmentBuilder 接口中的方法：

RuntimeEnvironmentBuilder 接口中的方法

```
public interface RuntimeEnvironmentBuilder {
```

```
public RuntimeEnvironmentBuilder persistence(boolean persistenceEnabled);

public RuntimeEnvironmentBuilder entityManagerFactory(Object emf);

public RuntimeEnvironmentBuilder addAsset(Resource asset, ResourceType type);

public RuntimeEnvironmentBuilder addEnvironmentEntry(String name, Object value);

public RuntimeEnvironmentBuilder addConfiguration(String name, String value);

public RuntimeEnvironmentBuilder knowledgeBase(KieBase kbase);

public RuntimeEnvironmentBuilder userGroupCallback(UserGroupCallback callback);

public RuntimeEnvironmentBuilder registerableItemsFactory(RegisterableItemsFactory
factory);

public RuntimeEnvironment get();

public RuntimeEnvironmentBuilder classLoader(ClassLoader cl);

public RuntimeEnvironmentBuilder schedulerService(Object globalScheduler);
```

使用 RuntimeEnvironmentBuilderFactory 类，获取 RuntimeEnvironmentBuilder 的实例。除了空实例没有设置外，您还可以通过运行时管理器的几个预配置的配置选项集合获取构建器。

以下定义显示了 RuntimeEnvironmentBuilderFactory 接口中的方法：

RuntimeEnvironmentBuilderFactory 接口的方法

```
public interface RuntimeEnvironmentBuilderFactory {

 /**
    * Provides a completely empty <code>RuntimeEnvironmentBuilder</code> instance to
manually
    * set all required components instead of relying on any defaults.
    * @return new instance of <code>RuntimeEnvironmentBuilder</code>
    */
   public RuntimeEnvironmentBuilder newEmptyBuilder();

   /**
    * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
    * <ul>
    *  <li>DefaultRuntimeEnvironment</li>
```

```
     * </ul>
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder();

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * but does not have persistence for the process engine configured so it will only store
process instances in memory
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultInMemoryBuilder();

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This method is tailored to work smoothly with KJAR files
     * @param groupId group id of kjar
     * @param artifactId artifact id of kjar
     * @param version version number of kjar
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId, String artifactId,
String version);

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This method is tailored to work smoothly with KJAR files and use the kbase and ksession
settings in the KJAR
     * @param groupId group id of kjar
     * @param artifactId artifact id of kjar
     * @param version version number of kjar
     * @param kbaseName name of the kbase defined in kmodule.xml stored in kjar
     * @param ksessionName name of the ksession define in kmodule.xml stored in kjar
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
```

```
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(String groupId, String artifactId,
String version, String kbaseName, String ksessionName);

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * This method is tailored to work smoothly with KJAR files and use the release ID defined
in the KJAR
     * @param releaseId <code>ReleaseId</code> that described the kjar
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId releaseId);

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
   * This method is tailored to work smoothly with KJAR files and use the kbase, ksession, and
release ID settings in the KJAR
     * @param releaseId <code>ReleaseId</code> that described the kjar
     * @param kbaseName name of the kbase defined in kmodule.xml stored in kjar
     * @param ksessionName name of the ksession define in kmodule.xml stored in kjar
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newDefaultBuilder(ReleaseId releaseId, String
kbaseName, String ksessionName);

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
     * It relies on KieClasspathContainer that requires the presence of kmodule.xml in the
META-INF folder which
     * defines the kjar itself.
     * Expects to use default kbase and ksession from kmodule.
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
```

```
    public RuntimeEnvironmentBuilder newClasspathKmoduleDefaultBuilder();

    /**
     * Provides default configuration of <code>RuntimeEnvironmentBuilder</code> that is
based on:
     * <ul>
     *  <li>DefaultRuntimeEnvironment</li>
     * </ul>
    * It relies on KieClasspathContainer that requires the presence of kmodule.xml in the META-
INF folder which
     * defines the kjar itself.
     * @param kbaseName name of the kbase defined in kmodule.xml
     * @param ksessionName name of the ksession define in kmodule.xml
     * @return new instance of <code>RuntimeEnvironmentBuilder</code> that is already
preconfigured with defaults
     *
     * @see DefaultRuntimeEnvironment
     */
    public RuntimeEnvironmentBuilder newClasspathKmoduleDefaultBuilder(String
kbaseName, String ksessionName);
```

运行时管理器还提供 TaskService 对象作为 RuntimeEngine 对象的集成组件（配置为与 KIE 会话通信）。如果您使用默认构建器之一，则任务服务的以下配置设置已存在：

- 持久性单元名称设置为 org.jbpm.persistence.jpa （用于进程引擎和任务服务）。
- 人工任务处理程序在 KIE 会话中注册。
- 基于 JPA 的历史日志事件监听程序在 KIE 会话中注册。
- 触发规则任务评估的事件监听程序(fireAllRules)在 KIE 会话上注册。

66.2.5. 为运行时引擎注册处理程序和监听程序

如果您使用运行时管理器 API，则运行时引擎对象代表进程引擎。

要使用自己的处理程序或监听程序扩展运行时引擎，您可以实施 RegisterableItemsonnectionFactory y 接口，然后使用

RuntimeEnvironmentBuilder.registerableItemsFactory（） 方法将其包含在运行时环境中。然后，运行时管理器自动为它创建的每个运行时引擎添加处理程序或监听程序。

以下定义显示了 RegisterableItemsFactory 接口中的方法：

RegisterableItemsFactory 接口的方法

```
/**
 * Returns new instances of <code>WorkItemHandler</code> that will be registered on
<code>RuntimeEngine</code>
 * @param runtime provides <code>RuntimeEngine</code> in case handler need to make use
of it internally
 * @return map of handlers to be registered - in case of no handlers empty map shall be
returned.
 */
  Map<String, WorkItemHandler> getWorkItemHandlers(RuntimeEngine runtime);

  /**
 * Returns new instances of <code>ProcessEventListener</code> that will be registered on
<code>RuntimeEngine</code>
 * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make
use of it internally
 * @return list of listeners to be registered - in case of no listeners empty list shall be returned.
 */
  List<ProcessEventListener> getProcessEventListeners(RuntimeEngine runtime);

  /**
 * Returns new instances of <code>AgendaEventListener</code> that will be registered on
<code>RuntimeEngine</code>
 * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make
use of it internally
 * @return list of listeners to be registered - in case of no listeners empty list shall be returned.
 */
  List<AgendaEventListener> getAgendaEventListeners(RuntimeEngine runtime);

  /**
 * Returns new instances of <code>WorkingMemoryEventListener</code> that will be
registered on <code>RuntimeEngine</code>
 * @param runtime provides <code>RuntimeEngine</code> in case listeners need to make
use of it internally
 * @return list of listeners to be registered - in case of no listeners empty list shall be returned.
 */
  List<WorkingMemoryEventListener> getWorkingMemoryEventListeners(RuntimeEngine
runtime);
```

进程引擎提供 RegisterableItemsFactory 的默认实现。您可以扩展这些实施来定义自定义处理程序和监听程序。

以下可用的实现可能很有用：

- org.jbpm.runtime.manager.impl.SimpleRegisterableItemsFactory: 最容易的实施。它没有任何预定义的内容，使用反射来生成基于给定类名称的处理程序和监听程序的实例。
- org.jbpm.runtime.manager.impl.DefaultRegisterableItemsFactory ：简单实施的扩展引入了与默认运行时环境构建器相同的默认值，仍然提供与 Simple 实施相同的功能。
- org.jbpm.runtime.manager.impl.cdi.InjectableRegisterableItemsFactory ：为 CDI 环境量身定制的默认实现的扩展，并提供 CDI 风格方法来使用制作者查找处理程序和监听器。

66.2.5.1. 使用文件注册工作项处理程序

您可以通过在 CustomWorkItem.conf 文件中定义文件并将该文件放置到类路径中，注册简单的工作项目处理程序（无状态或依赖于 KieSession 状态）。

流程

1. 在类路径的根目录下的 META-INF 子目录中创建一个名为 drools.session.conf 的文件。对于 Web 应用，目录为 WEB-INF/classes/META-INF。
2. 在 drools.session.conf 文件中添加以下行：

```
drools.workItemHandlers = CustomWorkItemHandlers.conf
```

3. 在同一目录中创建一个名为 CustomWorkItemHandlers.conf 的文件。
4. 在 CustomWorkItemHandlers.conf 文件中，使用 MVEL 风格定义自定义工作项目处理程序，如下例所示：

```
[
  "Log": new org.jbpm.process.instance.impl.demo.SystemOutWorkItemHandler(),
  "WebService": new
```

```
    org.jbpm.process.workitem.webservice.WebServiceWorkItemHandler(ksession),
      "Rest": new org.jbpm.process.workitem.rest.RESTWorkItemHandler(),
      "Service Task" : new org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession)
    ]
```

结果

您列出的工作项目处理程序是为应用程序创建的任何 KIE 会话注册，无论应用程序使用了运行时管理器 API。

66.2.5.2. 在 CDI 环境中注册处理程序和监听程序

如果您的应用使用运行时管理器 API 并在 CDI 环境中运行，则您的类可以实施专用制作者接口，为所有运行时引擎提供自定义工作项目处理程序和事件监听程序。

要创建工作项处理程序，您必须实施 WorkItemHandlerProducer 接口。

WorkItemHandlerProducer 接口的定义

```
public interface WorkItemHandlerProducer {

    /**
     * Returns a map of work items (key = work item name, value=  work item handler instance)
     * to be registered on the KieSession
     * <br/>
     * The following parameters are accepted:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     *
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the
producer to filter out
     * and provide valid instances for given owner
     * @param params - the owner might provide some parameters, usually KieSession,
TaskService, RuntimeManager instances
     * @return map of work item handler instances (recommendation is to always return new
instances when this method is invoked)
     */
    Map<String, WorkItemHandler> getWorkItemHandlers(String identifier, Map<String, Object>
params);
}
```

要创建事件监听程序，您必须实现 EventListenerProducer 接口。给事件监听器制作者添加正确的限定符，以指示它所提供的监听程序类型。使用以下注解之一：

- @process for ProcessEventListener
- @agenda for AgendaEventListener
- @WorkingMemory for WorkingMemoryEventListener

EventListenerProducer 接口的定义

```
public interface EventListenerProducer<T> {

    /**
     * Returns a list of instances for given (T) type of listeners
     * <br/>
     * The following parameters are accepted:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     * @param identifier - identifier of the owner - usually RuntimeManager that allows the
producer to filter out
     * and provide valid instances for given owner
     * @param params - the owner might provide some parameters, usually KieSession,
TaskService, RuntimeManager instances
     * @return list of listener instances (recommendation is to always return new instances
when this method is invoked)
     */
    List<T> getEventListeners(String identifier, Map<String, Object>  params);
}
```

通过在 META-INF 子目录中包含 bean 归档，将您的接口实施打包为 bean 归档。将 bean 归档放置在应用程序类路径上，例如，在 Web 应用的 WEB-INF/lib 中。基于 CDI 的运行时管理器发现软件包，并在它从数据存储中创建或加载的每个 KieSession 中注册工作项目处理程序和事件监听程序。

进程引擎为制作者提供某些参数，以启用有状态和高级操作。例如，处理程序或监听器可以使用参数

来发送进程引擎，或当出错时向进程实例发出信号。进程引擎以参数的形式提供以下组件：

- KieSession
- TaskService
- RuntimeManager

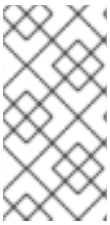
另外，RuntimeManager 类实例的标识符作为参数提供。您可以对标识符应用过滤来决定此 RuntimeManager 实例是否接收处理程序和监听程序。

66.3. 进程引擎中的服务

进程引擎提供一组高级别服务，在运行时管理器 API 上运行。

该服务提供了在您的应用程序中嵌入流程引擎的最便捷方式。KIE 服务器还在内部使用这些服务。

使用服务时，您不需要自行实施运行时管理器、运行时引擎、会话和其他流程引擎实体。但是，您可以根据需要通过服务访问底层 RuntimeManager 对象。



注意

如果您将 EJB 远程客户端用于服务 API，则 RuntimeManager 对象不可用，因为它们在序列化后不会在客户端上正确操作。

66.3.1. 处理引擎服务的模块

进程引擎服务作为一组模块提供。这些模块按照它们的框架依赖项来分组。您可以选择合适的模块，且只使用这些模块，而不让应用程序依赖于其他模块使用的框架。

可用的模块如下：

- jBPM-services-api ：只有 API 类和接口
- jBPM-kie-services ：一个在纯 Java 中服务 API 的代码实施，而无需任何框架依赖关系
- jBPM-services-cdi: 在核心服务实施之上的 CDI 打包程序
- jBPM-services-ejb-api ：服务 API 的扩展，以支持 EJB 要求
- jBPM-services-ejb-impl: EJB 封装在核心服务实施之上
- jBPM-services-ejb-timer ：基于 EJB 计时器服务的调度程序服务来支持基于时间的操作，如计时器事件和截止时间
- jBPM-services-ejb-client ：是 EJB 远程客户端实施，目前仅支持红帽 JBoss EAP

66.3.2. 部署服务

部署服务会在进程引擎中部署和取消部署单元。

部署单元 代表 KJAR 文件的内容。部署单元包括商业资产，如流程定义、规则、表单和数据模型。部署单元后，您可以执行它定义的进程。您还可以查询可用的部署单元。

每个部署单元都有一个唯一标识符字符串 deploymentId，也称为 deploymentUnitId。您可以使用此标识符将任何服务操作应用到部署单元。

在此服务的典型用例中，您可以同时加载和卸载多个 KJAR，并在需要时同时执行进程。

以下代码示例显示了对部署服务的简单使用。

使用部署服务

```
// Create deployment unit by providing the GAV of the KJAR
DeploymentUnit deploymentUnit = new KModuleDeploymentUnit(GROUP_ID, ARTIFACT_ID,
VERSION);
// Get the deploymentId for the deployed unit
String deploymentId = deploymentUnit.getIdentifier();
// Deploy the unit
deploymentService.deploy(deploymentUnit);
// Retrieve the deployed unit
DeployedUnit deployed = deploymentService.getDeployedUnit(deploymentId);
// Get the runtime manager
RuntimeManager manager = deployed.getRuntimeManager();
```

以下定义显示了完整的 DeploymentService 接口：

DeploymentService 接口的定义

```
public interface DeploymentService {

    void deploy(DeploymentUnit unit);

    void undeploy(DeploymentUnit unit);

    RuntimeManager getRuntimeManager(String deploymentUnitId);

    DeployedUnit getDeployedUnit(String deploymentUnitId);

    Collection<DeployedUnit> getDeployedUnits();

    void activate(String deploymentId);

    void deactivate(String deploymentId);

    boolean isDeployed(String deploymentUnitId);
}
```

66.3.3. 定义服务

当您使用部署服务部署进程定义时，定义服务会自动扫描定义，解析流程，并提取进程引擎所需的信息。

您可以使用定义服务 API 检索有关进程定义的信息。该服务直接从 BPMN2 进程定义中提取这些信息。可用的信息如下：

- 进程定义，如 ID、名称和描述
- 进程变量，包括每个变量的名称和类型
- 在进程中使用的可重复使用的 子进程（若有）
- 代表域特定活动 的服务任务
- 用户任务 包括分配信息
- 带有输入和输出信息的任务 数据

以下代码示例显示了对定义服务的简单使用。processID 必须与您使用部署服务部署的 KJAR 文件中的进程定义的 ID 对应。

使用定义服务

```
String processId = "org.jbpm.writedocument";

Collection<UserTaskDefinition> processTasks =
bpmn2Service.getTasksDefinitions(deploymentUnit.getIdentifier(), processId);

Map<String, String> processData =
bpmn2Service.getProcessVariables(deploymentUnit.getIdentifier(), processId);

Map<String, String> taskInputMappings =
bpmn2Service.getTaskInputMappings(deploymentUnit.getIdentifier(), processId, "Write a
Document" );
```

您还可以使用 定义服务扫描您作为 BPMN2 兼容 XML 内容提供的定义，而无需使用 KJAR 文

件。buildProcessDefinition 方法提供此功能。

以下定义显示了完整的 DefinitionService 接口：

DefinitionService 接口的定义

```
public interface DefinitionService {

    ProcessDefinition buildProcessDefinition(String deploymentId, String bpmn2Content,
ClassLoader classLoader, boolean cache) throws IllegalArgumentException;

    ProcessDefinition getProcessDefinition(String deploymentId, String processId);

    Collection<String> getReusableSubProcesses(String deploymentId, String processId);

    Map<String, String> getProcessVariables(String deploymentId, String processId);

    Map<String, String> getServiceTasks(String deploymentId, String processId);

    Map<String, Collection<String>> getAssociatedEntities(String deploymentId, String
processId);

    Collection<UserTaskDefinition> getTasksDefinitions(String deploymentId, String
processId);

    Map<String, String> getTaskInputMappings(String deploymentId, String processId, String
taskName);

    Map<String, String> getTaskOutputMappings(String deploymentId, String processId, String
taskName);

}
```

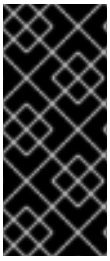
66.3.4. 进程服务

部署和定义服务在流程引擎中准备流程数据。要根据此数据执行进程，请使用 process 服务。进程服务支持与进程引擎执行环境交互，包括以下操作：

- 启动新进程实例

- 将进程作为单一事务运行
- 使用现有的进程实例，例如，信号处理事件，获取信息详情，以及设置变量的值
- 使用工作项

进程服务也是命令 executor。您可以使用它对 KIE 会话执行命令以扩展其功能。



重要

进程服务针对运行时操作进行了优化。在需要运行进程或更改进程实例时使用它，如信号事件或更改变量。例如，若要读取操作，显示可用的进程实例，请使用运行时数据服务。

以下代码示例显示了部署和运行进程：

使用部署过程及进程服务部署并运行进程

```
KModuleDeploymentUnit deploymentUnit = new KModuleDeploymentUnit(GROUP_ID,
ARTIFACT_ID, VERSION);

deploymentService.deploy(deploymentUnit);

long processInstanceId = processService.startProcess(deploymentUnit.getIdentifier(),
"customtask");

ProcessInstance pi = processService.getProcessInstance(processInstanceId);
```

startProcess 方法预期 deploymentId 作为第一个参数。使用此参数，当应用程序可能有多个部署时，您可以在特定部署中启动进程。

例如，您可能会从不同的 KJAR 文件中部署相同进程的不同版本。然后，您可以使用正确的 deploymentId 启动所需的版本。

以下定义显示了完整的 ProcessService 接口：

ProcessService 接口的定义

```
public interface ProcessService {

 /**
  * Starts a process with no variables
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @return process instance IDentifier
  * @throws RuntimeException in case of encountered errors
  * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
  * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active
  */
 Long startProcess(String deploymentId, String processId);

 /**
  * Starts a process and sets variables
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @param params process variables
  * @return process instance IDentifier
  * @throws RuntimeException in case of encountered errors
  * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
  * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active
  */
   Long startProcess(String deploymentId, String processId, Map<String, Object> params);

 /**
  * Starts a process with no variables and assigns a correlation key
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @param correlationKey correlation key to be assigned to the process instance - must be
unique
  * @return process instance IDentifier
  * @throws RuntimeException in case of encountered errors
  * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
    * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active
  */
 Long startProcess(String deploymentId, String processId, CorrelationKey correlationKey);
```

```
 /**
  * Starts a process, sets variables, and assigns a correlation key
  *
  * @param deploymentId deployment identifier
  * @param processId process identifier
  * @param correlationKey correlation key to be assigned to the process instance - must be
unique
  * @param params process variables
  * @return process instance IDentifier
  * @throws RuntimeException in case of encountered errors
  * @throws DeploymentNotFoundException in case a deployment with the given deployment
identifier does not exist
    * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active
  */
   Long startProcess(String deploymentId, String processId, CorrelationKey correlationKey,
Map<String, Object> params);

   /**
    * Run a process that is designed to start and finish in a single transaction.
    * This method starts the process and returns when the process completes.
    * It returns the state of process variables at the outcome of the process
    *
    * @param deploymentId deployment identifier for the KJAR file of the process
    * @param processId process identifier
    * @param params process variables
    * @return the state of process variables at the end of the process
    */
     Map<String, Object> computeProcessOutcome(String deploymentId, String processId,
Map<String, Object> params);

    /**
     * Starts a process at the listed nodes, instead of the normal starting point.
     * This method can be used for restarting a process that was aborted. However,
     * it does not restore the context of a previous process instance. You must
     * supply all necessary variables when calling this method.
     * This method does not guarantee that the process is started in a valid state.
     *
     * @param deploymentId deployment identifier
     * @param processId process identifier
     * @param params process variables
     * @param nodeIds list of BPMN node identifiers where the process must start
     * @return process instance IDentifier
     * @throws RuntimeException in case of encountered errors
     * @throws DeploymentNotFoundException in case a deployment with the given
deployment identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active
     */
    Long startProcessFromNodeIds(String deploymentId, String processId, Map<String,
Object> params, String... nodeIds);

    /**
    * Starts a process at the listed nodes, instead of the normal starting point,
    * and assigns a correlation key.
    * This method can be used for restarting a process that was aborted. However,
```

```
     * it does not restore the context of a previous process instance. You must
     * supply all necessary variables when calling this method.
     * This method does not guarantee that the process is started in a valid state.
     *
     * @param deploymentId deployment identifier
     * @param processId process identifier
     * @param key correlation key (must be unique)
     * @param params process variables
     * @param nodeIds list of BPMN node identifiers where the process must start.
     * @return process instance IDentifier
     * @throws RuntimeException in case of encountered errors
     * @throws DeploymentNotFoundException in case a deployment with the given
deployment identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active
     */
    Long startProcessFromNodeIds(String deploymentId, String processId, CorrelationKey key,
Map<String, Object> params, String... nodeIds);

    /**
     * Aborts the specified process
     *
     * @param processInstanceId process instance unique identifier
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    void abortProcessInstance(Long processInstanceId);

    /**
     * Aborts the specified process
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId process instance unique identifier
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    void abortProcessInstance(String deploymentId, Long processInstanceId);

    /**
   * Aborts all specified processes
   *
   * @param processInstanceIds list of process instance unique identifiers
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
   */
    void abortProcessInstances(List<Long> processInstanceIds);

    /**
     * Aborts all specified processes
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceIds list of process instance unique identifiers
     * @throws DeploymentNotFoundException in case the deployment unit was not found
```

```
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    void abortProcessInstances(String deploymentId, List<Long> processInstanceIds);

    /**
   * Signals an event to a single process instance
   *
   * @param processInstanceId the process instance unique identifier
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
   */
    void signalProcessInstance(Long processInstanceId, String signalName, Object event);

    /**
     * Signals an event to a single process instance
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId the process instance unique identifier
     * @param signalName the ID of the signal in the process
     * @param event the event object to be passed with the event
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    void signalProcessInstance(String deploymentId, Long processInstanceId, String
signalName, Object event);

    /**
   * Signal an event to a list of process instances
   *
   * @param processInstanceIds list of process instance unique identifiers
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
   */
    void signalProcessInstances(List<Long> processInstanceIds, String signalName, Object
event);

    /**
     * Signal an event to a list of process instances
     *
     * @param deploymentId deployment to which the process instances belong
     * @param processInstanceIds list of process instance unique identifiers
     * @param signalName the ID of the signal in the process
     * @param event the event object to be passed with the event
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    void signalProcessInstances(String deploymentId, List<Long> processInstanceIds, String
```

```
signalName, Object event);

  /**
    * Signal an event to a single process instance by correlation key
    *
    * @param correlationKey the unique correlation key of the process instance
    * @param signalName the ID of the signal in the process
    * @param event the event object to be passed in with the event
    * @throws DeploymentNotFoundException in case the deployment unit was not found
    * @throws ProcessInstanceNotFoundException in case a process instance with the given
key was not found
    */
   void signalProcessInstanceByCorrelationKey(CorrelationKey correlationKey, String
signalName, Object event);

  /**
   * Signal an event to a single process instance by correlation key
   *
   * @param deploymentId deployment to which the process instance belongs
   * @param correlationKey the unique correlation key of the process instance
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed in with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given
key was not found
   */
  void signalProcessInstanceByCorrelationKey(String deploymentId, CorrelationKey
correlationKey, String signalName, Object event);

  /**
   * Signal an event to given list of correlation keys
   *
   * @param correlationKeys list of unique correlation keys of process instances
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed in with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with one of the
given keys was not found
   */
  void signalProcessInstancesByCorrelationKeys(List<CorrelationKey> correlationKeys,
String signalName, Object event);

  /**
   * Signal an event to given list of correlation keys
   *
   * @param deploymentId deployment to which the process instances belong
   * @param correlationKeys list of unique correlation keys of process instances
   * @param signalName the ID of the signal in the process
   * @param event the event object to be passed in with the event
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with one of the
given keys was not found
   */
  void signalProcessInstancesByCorrelationKeys(String deploymentId, List<CorrelationKey>
correlationKeys, String signalName, Object event);
```

```
    /**
     * Signal an event to a any process instance that listens to a given signal and belongs to a
given deployment
     *
     * @param deployment identifier of the deployment
     * @param signalName the ID of the signal in the process
     * @param event the event object to be passed with the event
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     */
    void signalEvent(String deployment, String signalName, Object event);

    /**
   * Returns process instance information. Will return null if no
   * active process with the ID is found
   *
   * @param processInstanceId The process instance unique identifier
   * @return Process instance information
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   */
    ProcessInstance getProcessInstance(Long processInstanceId);

    /**
     * Returns process instance information. Will return null if no
     * active process with the ID is found
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @return Process instance information
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     */
    ProcessInstance getProcessInstance(String deploymentId, Long processInstanceId);

    /**
   * Returns process instance information. Will return null if no
   * active process with that correlation key is found
   *
   * @param correlationKey correlation key assigned to the process instance
   * @return Process instance information
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   */
    ProcessInstance getProcessInstance(CorrelationKey correlationKey);

    /**
     * Returns process instance information. Will return null if no
     * active process with that correlation key is found
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param correlationKey correlation key assigned to the process instance
     * @return Process instance information
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     */
    ProcessInstance getProcessInstance(String deploymentId, CorrelationKey correlationKey);

    /**
   * Sets a process variable.
   * @param processInstanceId The process instance unique identifier
```

```
   * @param variableId The variable ID to set
   * @param value The variable value
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
   */
    void setProcessVariable(Long processInstanceId, String variableId, Object value);

    /**
     * Sets a process variable.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @param variableId The variable id to set.
     * @param value The variable value.
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    void setProcessVariable(String deploymentId, Long processInstanceId, String variableId,
Object value);

    /**
   * Sets process variables.
   *
   * @param processInstanceId The process instance unique identifier
   * @param variables map of process variables (key = variable name, value = variable value)
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
   */
    void setProcessVariables(Long processInstanceId, Map<String, Object> variables);

    /**
     * Sets process variables.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @param variables map of process variables (key = variable name, value = variable value)
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    void setProcessVariables(String deploymentId, Long processInstanceId, Map<String,
Object> variables);

    /**
   * Gets a process instance variable.
   *
   * @param processInstanceId the process instance unique identifier
   * @param variableName the variable name to get from the process
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
   */
    Object getProcessInstanceVariable(Long processInstanceId, String variableName);
```

```
    /**
     * Gets a process instance variable.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId the process instance unique identifier
     * @param variableName the variable name to get from the process
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    Object getProcessInstanceVariable(String deploymentId, Long processInstanceId, String
variableName);

  /**
   * Gets a process instance variable values.
   *
   * @param processInstanceId The process instance unique identifier
   * @throws DeploymentNotFoundException in case the deployment unit was not found
   * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
   */
  Map<String, Object> getProcessInstanceVariables(Long processInstanceId);

  /**
     * Gets a process instance variable values.
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId The process instance unique identifier
     * @throws DeploymentNotFoundException in case the deployment unit was not found
     * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
     */
    Map<String, Object> getProcessInstanceVariables(String deploymentId, Long
processInstanceId);

  /**
   * Returns all signals available in current state of given process instance
   *
   * @param processInstanceId process instance ID
   * @return list of available signals or empty list if no signals are available
   */
    Collection<String> getAvailableSignals(Long processInstanceId);

    /**
     * Returns all signals available in current state of given process instance
     *
     * @param deploymentId deployment to which the process instance belongs
     * @param processInstanceId process instance ID
     * @return list of available signals or empty list if no signals are available
     */
    Collection<String> getAvailableSignals(String deploymentId, Long processInstanceId);

  /**
   * Completes the specified WorkItem with the given results
   *
```

```
     * @param id workItem ID
     * @param results results of the workItem
     * @throws DeploymentNotFoundException in case the deployment unit was not found
      * @throws WorkItemNotFoundException in case a work item with the given ID was not
found
     */
      void completeWorkItem(Long id, Map<String, Object> results);

      /**
       * Completes the specified WorkItem with the given results
       *
       * @param deploymentId deployment to which the process instance belongs
       * @param processInstanceId process instance ID to which the work item belongs
       * @param id workItem ID
       * @param results results of the workItem
       * @throws DeploymentNotFoundException in case the deployment unit was not found
       * @throws WorkItemNotFoundException in case a work item with the given ID was not
found
       */
      void completeWorkItem(String deploymentId, Long processInstanceId, Long id,
Map<String, Object> results);

      /**
       * Abort the specified workItem
       *
       * @param id workItem ID
       * @throws DeploymentNotFoundException in case the deployment unit was not found
       * @throws WorkItemNotFoundException in case a work item with the given ID was not
found
       */
      void abortWorkItem(Long id);

      /**
       * Abort the specified workItem
       *
       * @param deploymentId deployment to which the process instance belongs
       * @param processInstanceId process instance ID to which the work item belongs
       * @param id workItem ID
       * @throws DeploymentNotFoundException in case the deployment unit was not found
       * @throws WorkItemNotFoundException in case a work item with the given ID was not
found
       */
      void abortWorkItem(String deploymentId, Long processInstanceId, Long id);

      /**
       * Returns the specified workItem
       *
       * @param id workItem ID
       * @return The specified workItem
       * @throws DeploymentNotFoundException in case the deployment unit was not found
       * @throws WorkItemNotFoundException in case a work item with the given ID was not
found
       */
      WorkItem getWorkItem(Long id);

      /**
```

```
    * Returns the specified workItem
    *
    * @param deploymentId deployment to which the process instance belongs
    * @param processInstanceId process instance ID to which the work item belongs
    * @param id workItem ID
    * @return The specified workItem
    * @throws DeploymentNotFoundException in case the deployment unit was not found
    * @throws WorkItemNotFoundException in case a work item with the given ID was not
found
    */
   WorkItem getWorkItem(String deploymentId, Long processInstanceId, Long id);

   /**
    * Returns active work items by process instance ID.
    *
    * @param processInstanceId process instance ID
    * @return The list of active workItems for the process instance
    * @throws DeploymentNotFoundException in case the deployment unit was not found
  * @throws ProcessInstanceNotFoundException in case a process instance with the given ID
was not found
    */
   List<WorkItem> getWorkItemByProcessInstance(Long processInstanceId);

   /**
    * Returns active work items by process instance ID.
    *
    * @param deploymentId deployment to which the process instance belongs
    * @param processInstanceId process instance ID
    * @return The list of active workItems for the process instance
    * @throws DeploymentNotFoundException in case the deployment unit was not found
    * @throws ProcessInstanceNotFoundException in case a process instance with the given
ID was not found
    */
   List<WorkItem> getWorkItemByProcessInstance(String deploymentId, Long
processInstanceId);


   /**
    * Executes the provided command on the underlying command executor (usually
KieSession)
    * @param deploymentId deployment identifier
    * @param command actual command for execution
    * @return results of the command execution
    * @throws DeploymentNotFoundException in case a deployment with the given
deployment identifier does not exist
    * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active for restricted commands (for example, start process)
    */
   public <T> T execute(String deploymentId, Command<T> command);

   /**
    * Executes the provided command on the underlying command executor (usually
KieSession)
    * @param deploymentId deployment identifier
    * @param context context implementation to be used to get the runtime engine
    * @param command actual command for execution
```

```
     * @return results of the command execution
     * @throws DeploymentNotFoundException in case a deployment with the given
deployment identifier does not exist
     * @throws DeploymentNotActiveException in case the deployment with the given
deployment identifier is not active for restricted commands (for example, start process)
     */
    public <T> T execute(String deploymentId, Context<?> context, Command<T> command);

}
```

66.3.5. Runtime Data Service

您可以使用运行时数据服务检索与进程相关的所有运行时信息，如启动进程实例并执行的节点实例。

例如，您可以构建基于列表的 UI，以根据运行时数据服务提供的信息，显示进程定义、处理实例、给定用户的任务和其他数据。

此服务经过优化，以便尽可能高效地提供所有所需信息。

以下示例演示了此服务的各种用法。

检索所有进程定义

```
Collection definitions = runtimeDataService.getProcesses(new QueryContext());
```

检索活跃进程实例

```
Collection<processinstancedesc> instances = runtimeDataService.getProcessInstances(new
QueryContext());
```

检索特定进程实例的活跃节点

```
Collection<nodeinstancedesc> instances =
runtimeDataService.getProcessInstanceHistoryActive(processInstanceId, new
QueryContext());
```

检索分配给用户 john的任务

```
List<tasksummary> taskSummaries =
runtimeDataService.getTasksAssignedAsPotentialOwner("john", new QueryFilter(0, 10));
```

运行时数据服务方法支持两个重要参数，即 QueryContext 和 QueryFilter。QueryFilter 是 QueryContext 的扩展。您可以使用这些参数来管理结果集，提供分页、排序和排序。您还可以在搜索用户任务时应用额外的过滤。

以下定义显示了 RuntimeDataService 接口方法：

RuntimeDataService 接口的定义

```
public interface RuntimeDataService {
 /**
  * Represents type of node instance log entries
  *
  */
 enum EntryType {

        START(0),
        END(1),
        ABORTED(2),
        SKIPPED(3),
        OBSOLETE(4),
        ERROR(5);
 }

    // Process instance information
```

```
    /**
     * Returns a list of process instance descriptions
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the available process
instances
     */
    Collection<ProcessInstanceDesc> getProcessInstances(QueryContext queryContext);

    /**
     * Returns a list of all process instance descriptions with the given statuses and initiated by
<code>initiator</code>
     * @param states list of possible state (int) values that the {@link ProcessInstance} can
have
     * @param initiator the initiator of the {@link ProcessInstance}
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
     *         the given criteria (states and initiator)
     */
    Collection<ProcessInstanceDesc> getProcessInstances(List<Integer> states, String
initiator, QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions found for the given process ID and
statuses and initiated by <code>initiator</code>
     * @param states list of possible state (int) values that the {@link ProcessInstance} can
have
     * @param processId ID of the {@link Process} (definition) used when starting the process
instance
     * @param initiator initiator of the {@link ProcessInstance}
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
     *         the given criteria (states, processId, and initiator)
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByProcessId(List<Integer> states,
String processId, String initiator, QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions found for the given process name and
statuses and initiated by <code>initiator</code>
     * @param states list of possible state (int) values that the {@link ProcessInstance} can
have
     * @param processName name (not ID) of the {@link Process} (definition) used when
starting the process instance
     * @param initiator initiator of the {@link ProcessInstance}
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
     *         the given criteria (states, processName and initiator)
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByProcessName(List<Integer>
states, String processName, String initiator, QueryContext queryContext);

    /**
```

```
    * Returns a list of process instance descriptions found for the given deployment ID and
statuses
    * @param deploymentId deployment ID of the runtime
    * @param states list of possible state (int) values that the {@link ProcessInstance} can
have
    * @param queryContext control parameters for the result, such as sorting and paging
    * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
    *         the given criteria (deploymentId and states)
    */
   Collection<ProcessInstanceDesc> getProcessInstancesByDeploymentId(String
deploymentId, List<Integer> states, QueryContext queryContext);

   /**
    * Returns process instance descriptions found for the given processInstanceId. If no
descriptions are found, null is returned. At the same time, the method
    * fetches all active tasks (in status: Ready, Reserved, InProgress) to provide the
information about what user task is keeping each instance
    * and who owns the task (if the task is already claimed by a user)
    * @param processInstanceId ID of the process instance to be fetched
    * @return process instance information, in the form of a {@link ProcessInstanceDesc}
instance
    */
   ProcessInstanceDesc getProcessInstanceById(long processInstanceId);

   /**
    * Returns the active process instance description found for the given correlation key. If
none is found, returns null. At the same time it
    * fetches all active tasks (in status: Ready, Reserved, InProgress) to provide information
about which user task is keeping each instance
    * and who owns the task (if the task is already claimed by a user)
    * @param correlationKey correlation key assigned to the process instance
    * @return process instance information, in the form of a {@link ProcessInstanceDesc}
instance
    */
   ProcessInstanceDesc getProcessInstanceByCorrelationKey(CorrelationKey
correlationKey);

   /**
    * Returns process instances descriptions (regardless of their states) found for the given
correlation key. If no descriptions are found, an empty list is returned
    * This query uses 'LIKE' to match correlation keys so it accepts partial keys. Matching
    * is performed based on a 'starts with' criterion
    * @param correlationKey correlation key assigned to the process instance
    * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
    *         the given correlation key
    */
   Collection<ProcessInstanceDesc> getProcessInstancesByCorrelationKey(CorrelationKey
correlationKey, QueryContext queryContext);

   /**
    * Returns process instance descriptions, filtered by their states, that were found for the
given correlation key. If none are found, returns an empty list
    * This query uses 'LIKE' to match correlation keys so it accepts partial keys. Matching
    * is performed based on a 'starts with' criterion
```

```
    * @param correlationKey correlation key assigned to process instance
    * @param states list of possible state (int) values that the {@link ProcessInstance} can
have
    * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
    *         the given correlation key
    */
   Collection<ProcessInstanceDesc>
getProcessInstancesByCorrelationKeyAndStatus(CorrelationKey correlationKey,
List<Integer> states, QueryContext queryContext);

   /**
    * Returns a list of process instance descriptions found for the given process definition ID
    * @param processDefId ID of the process definition
    * @param queryContext control parameters for the result, such as sorting and paging
    * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
    *         the given criteria (deploymentId and states)
    */
   Collection<ProcessInstanceDesc> getProcessInstancesByProcessDefinition(String
processDefId, QueryContext queryContext);

   /**
    * Returns a list of process instance descriptions found for the given process definition ID,
filtered by state
    * @param processDefId ID of the process definition
    * @param states list of possible state (int) values that the {@link ProcessInstance} can
have
    * @param queryContext control parameters for the result, such as sorting and paging
    * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that match
    *         the given criteria (deploymentId and states)
    */
   Collection<ProcessInstanceDesc> getProcessInstancesByProcessDefinition(String
processDefId, List<Integer> states, QueryContext queryContext);

   /**
    * Returns process instance descriptions that match process instances that have the given
variable defined, filtered by state
    * @param variableName name of the variable that process instance should have
    * @param states list of possible state (int) values that the {@link ProcessInstance} can
have. If null, returns only active instances
    * @param queryContext control parameters for the result, such as sorting and paging
    * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that have the given variable defined
    */
   Collection<ProcessInstanceDesc> getProcessInstancesByVariable(String variableName,
List<Integer> states, QueryContext queryContext);

   /**
    * Returns process instance descriptions that match process instances that have the given
variable defined and the value of the variable matches the given variableValue
    * @param variableName name of the variable that process instance should have
    * @param variableValue value of the variable to match
    * @param states list of possible state (int) values that the {@link ProcessInstance} can
have. If null, returns only active instances
```

```
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the process
instances that have the given variable defined with the given value
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByVariableAndValue(String
variableName, String variableValue, List<Integer> states, QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions that have the specified parent
     * @param parentProcessInstanceId ID of the parent process instance
     * @param states list of possible state (int) values that the {@link ProcessInstance} can
have. If null, returns only active instances
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessInstanceDesc} instances representing the available process
instances
     */
    Collection<ProcessInstanceDesc> getProcessInstancesByParent(Long
parentProcessInstanceId, List<Integer> states, QueryContext queryContext);

    /**
     * Returns a list of process instance descriptions that are subprocesses of the specified
process, or subprocesses of those subprocesses, and so on. The list includes the full
hierarchy of subprocesses under the specified parent process
     * @param processInstanceId ID of the parent process instance
     * @return list of {@link ProcessInstanceDesc} instances representing the full hierarchy of
this process
     */
    Collection<ProcessInstanceDesc>
getProcessInstancesWithSubprocessByProcessInstanceId(Long processInstanceId,
List<Integer> states, QueryContext queryContext);

    // Node and Variable instance information

    /**
     * Returns the active node instance descriptor for the given work item ID, if the work item
exists and is active
     * @param workItemId identifier of the work item
     * @return NodeInstanceDesc for work item if it exists and is still active, otherwise null is
returned
     */
    NodeInstanceDesc getNodeInstanceForWorkItem(Long workItemId);

    /**
     * Returns a trace of all active nodes for the given process instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return
     */
    Collection<NodeInstanceDesc> getProcessInstanceHistoryActive(long processInstanceId,
QueryContext queryContext);

    /**
     * Returns a trace of all executed (completed) nodes for the given process instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return
```

```
    */
    Collection<NodeInstanceDesc> getProcessInstanceHistoryCompleted(long
processInstanceId, QueryContext queryContext);

    /**
     * Returns a complete trace of all executed (completed) and active nodes for the given
process instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return {@link NodeInstance} information, in the form of a list of {@link
NodeInstanceDesc} instances,
     *         that come from a process instance that matches the given criteria (deploymentId,
processId)
     */
    Collection<NodeInstanceDesc> getProcessInstanceFullHistory(long processInstanceId,
QueryContext queryContext);

    /**
     * Returns a complete trace of all events of the given type (START, END, ABORTED,
SKIPPED, OBSOLETE or ERROR) for the given process instance
     * @param processInstanceId unique identifier of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @param type type of events to be returned (START, END, ABORTED, SKIPPED,
OBSOLETE or ERROR). To return all events, use {@link #getProcessInstanceFullHistory(long,
QueryContext)}
     * @return collection of node instance descriptions
     */
    Collection<NodeInstanceDesc> getProcessInstanceFullHistoryByType(long
processInstanceId, EntryType type, QueryContext queryContext);


    /**
     * Returns a trace of all nodes for the given node types and process instance ID
     * @param processInstanceId unique identifier of the process instance
     * @param nodeTypes list of node types to filter nodes of the process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return collection of node instance descriptions
     */
    Collection<NodeInstanceDesc> getNodeInstancesByNodeType(long processInstanceId,
List<String> nodeTypes, QueryContext queryContext);

    /**
     * Returns a trace of all nodes for the given node types and correlation key
     * @param correlationKey correlation key
     * @param states list of states
     * @param nodeTypes list of node types to filter nodes of process instance
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return collection of node instance descriptions
     */
    Collection<NodeInstanceDesc>
getNodeInstancesByCorrelationKeyNodeType(CorrelationKey correlationKey,  List<Integer>
states, List<String> nodeTypes, QueryContext queryContext);


    /**
     * Returns a collection of all process variables and their current values for the given
```

```
process instance
     * @param processInstanceId process instance ID
     * @return information about variables in the specified process instance,
     *         represented by a list of {@link VariableDesc} instances
     */
    Collection<VariableDesc> getVariablesCurrentState(long processInstanceId);

    /**
     * Returns a collection of changes to the given variable within the scope of a process
instance
     * @param processInstanceId unique identifier of the process instance
     * @param variableId ID of the variable
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return information about the variable with the given ID in the specified process
instance,
     *         represented by a list of {@link VariableDesc} instances
     */
    Collection<VariableDesc> getVariableHistory(long processInstanceId, String variableId,
QueryContext queryContext);


    // Process information


    /**
     * Returns a list of process definitions for the given deployment ID
     * @param deploymentId deployment ID of the runtime
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessDefinition} instances representing processes that match
     *         the given criteria (deploymentId)
     */
    Collection<ProcessDefinition> getProcessesByDeploymentId(String deploymentId,
QueryContext queryContext);

    /**
     * Returns a list of process definitions that match the given filter
     * @param filter regular expression
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of {@link ProcessDefinition} instances with a name or ID that matches the
given regular expression
     */
    Collection<ProcessDefinition> getProcessesByFilter(String filter, QueryContext
queryContext);

    /**
     * Returns all process definitions available
     * @param queryContext control parameters for the result, such as sorting and paging
     * @return list of all available processes, in the form a of a list of {@link ProcessDefinition}
instances
     */
    Collection<ProcessDefinition> getProcesses(QueryContext queryContext);

    /**
     * Returns a list of process definition identifiers for the given deployment ID
     * @param deploymentId deployment ID of the runtime
     * @param queryContext control parameters for the result, such as sorting and paging
```

```
    * @return list of all available process id's for a particular deployment/runtime
    */
   Collection<String> getProcessIds(String deploymentId, QueryContext queryContext);

   /**
    * Returns process definitions for the given process ID regardless of the deployment
    * @param processId ID of the process
    * @return collection of {@link ProcessDefinition} instances representing the {@link
Process}
    *         with the specified process ID
    */
   Collection<ProcessDefinition> getProcessesById(String processId);

   /**
    * Returns the process definition for the given deployment and process identifiers
    * @param deploymentId ID of the deployment (runtime)
    * @param processId ID of the process
    * @return {@link ProcessDefinition} instance, representing the {@link Process}
    *         that is present in the specified deployment with the specified process ID
    */
   ProcessDefinition getProcessesByDeploymentIdProcessId(String deploymentId, String
processId);

 // user task query operations

 /**
  * Return a task by its workItemId
  * @param workItemId
  * @return @{@link UserTaskInstanceDesc} task
  */
   UserTaskInstanceDesc getTaskByWorkItemId(Long workItemId);

 /**
  * Return a task by its taskId
  * @param taskId
  * @return @{@link UserTaskInstanceDesc} task
  */
 UserTaskInstanceDesc getTaskById(Long taskId);

 /**
  * Return a task by its taskId with SLA data if the withSLA param is true
  * @param taskId
  * @param withSLA
  * @return @{@link UserTaskInstanceDesc} task
  */
 UserTaskInstanceDesc getTaskById(Long taskId, boolean withSLA);

 /**
  * Return a list of assigned tasks for a Business Administrator user. Business
  * administrators play the same role as task stakeholders but at task type
  * level. Therefore, business administrators can perform the exact same
  * operations as task stakeholders. Business administrators can also observe
  * the progress of notifications
  *
  * @param userId identifier of the Business Administrator user
  * @param filter filter for the list of assigned tasks
```

```
     * @return list of @{@link TaskSummary} task summaries
     */
    List<TaskSummary> getTasksAssignedAsBusinessAdministrator(String userId, QueryFilter
    filter);

    /**
        * Return a list of assigned tasks for a Business Administrator user for with one of the listed
        * statuses
        * @param userId identifier of the Business Administrator user
        * @param statuses the statuses of the tasks to return
        * @param filter filter for the list of assigned tasks
        * @return list of @{@link TaskSummary} task summaries
        */
    List<TaskSummary> getTasksAssignedAsBusinessAdministratorByStatus(String userId,
    List<Status> statuses, QueryFilter filter);

    /**
     * Return a list of tasks that a user is eligible to own
     *
     * @param userId identifier of the user
     * @param filter filter for the list of tasks
     * @return list of @{@link TaskSummary} task summaries
     */
    List<TaskSummary> getTasksAssignedAsPotentialOwner(String userId, QueryFilter filter);

    /**
     * Return a list of tasks the user or user groups are eligible to own
     *
     * @param userId identifier of the user
     * @param groupIds a list of identifiers of the groups
     * @param filter filter for the list of tasks
     * @return list of @{@link TaskSummary} task summaries
     */
    List<TaskSummary> getTasksAssignedAsPotentialOwner(String userId, List<String>
    groupIds, QueryFilter filter);

    /**
     * Return a list of tasks the user is eligible to own and that are in one of the listed
     * statuses
     *
     * @param userId identifier of the user
     * @param status filter for the task statuses
     * @param filter filter for the list of tasks
     * @return list of @{@link TaskSummary} task summaries
     */
    List<TaskSummary> getTasksAssignedAsPotentialOwnerByStatus(String userId,
    List<Status> status, QueryFilter filter);

    /**
     * Return a list of tasks the user or groups are eligible to own and that are in one of the listed
     * statuses
     * @param userId identifier of the user
     * @param groupIds filter for the identifiers of the groups
     * @param status filter for the task statuses
     * @param filter filter for the list of tasks
     * @return list of @{@link TaskSummary} task summaries
```

```
 */
 List<TaskSummary> getTasksAssignedAsPotentialOwner(String userId, List<String>
groupIds, List<Status> status, QueryFilter filter);

 /**
  * Return a list of tasks the user is eligible to own, that are in one of the listed
  * statuses, and that have an expiration date starting at <code>from</code>. Tasks that do not
have expiration date set
  * will also be included in the result set
  *
  * @param userId identifier of the user
  * @param status filter for the task statuses
  * @param from earliest expiration date for the tasks
  * @param filter filter for the list of tasks
  * @return list of @{@link TaskSummary} task summaries
  */
 List<TaskSummary> getTasksAssignedAsPotentialOwnerByExpirationDateOptional(String
userId, List<Status> status, Date from, QueryFilter filter);

 /**
  * Return a list of tasks the user has claimed, that are in one of the listed
  * statuses, and that have an expiration date starting at <code>from</code>. Tasks that do not
have expiration date set
  * will also be included in the result set
  *
  * @param userId identifier of the user
  * @param strStatuses filter for the task statuses
  * @param from earliest expiration date for the tasks
  * @param filter filter for the list of tasks
  * @return list of @{@link TaskSummary} task summaries
  */
 List<TaskSummary> getTasksOwnedByExpirationDateOptional(String userId, List<Status>
strStatuses, Date from, QueryFilter filter);

 /**
  * Return a list of tasks the user has claimed
  *
  * @param userId identifier of the user
  * @param filter filter for the list of tasks
  * @return list of @{@link TaskSummary} task summaries
  */
 List<TaskSummary> getTasksOwned(String userId, QueryFilter filter);

 /**
  * Return a list of tasks the user has claimed with one of the listed
  * statuses
  *
  * @param userId identifier of the user
  * @param status filter for the task statuses
  * @param filter filter for the list of tasks
  * @return list of @{@link TaskSummary} task summaries
  */
 List<TaskSummary> getTasksOwnedByStatus(String userId, List<Status> status, QueryFilter
filter);

 /**
```

```
 * Get a list of tasks the Process Instance is waiting on
 *
 * @param processInstanceId identifier of the process instance
 * @return list of task identifiers
 */
List<Long> getTasksByProcessInstanceId(Long processInstanceId);

/**
 * Get filter for the tasks the Process Instance is waiting on that are in one of the
 * listed statuses
 *
 * @param processInstanceId identifier of the process instance
 * @param status filter for the task statuses
 * @param filter filter for the list of tasks
 * @return list of @{@link TaskSummary} task summaries
 */
List<TaskSummary> getTasksByStatusByProcessInstanceId(Long processInstanceId,
List<Status> status, QueryFilter filter);

  /**
 * Get a list of task audit logs for all tasks owned by the user, applying a query filter to the list
of tasks
 *
 *
 * @param userId identifier of the user that owns the tasks
 * @param filter filter for the list of tasks
 * @return list of @{@link AuditTask} task audit logs
 */
  List<AuditTask> getAllAuditTask(String userId, QueryFilter filter);

  /**
 * Get a list of task audit logs for all tasks that are active and owned by the user, applying a
query filter to the list of tasks
 *
 * @param userId identifier of the user that owns the tasks
 * @param filter filter for the list of tasks
 * @return list of @{@link AuditTask} audit tasks
 */
  List<AuditTask> getAllAuditTaskByStatus(String userId, QueryFilter filter);

  /**
 * Get a list of task audit logs for group tasks (actualOwner == null) for the user, applying a
query filter to the list of tasks
 *
 * @param userId identifier of the user that is associated with the group tasks
 * @param filter filter for the list of tasks
 * @return list of @{@link AuditTask} audit tasks
 */
  List<AuditTask> getAllGroupAuditTask(String userId, QueryFilter filter);


  /**
 * Get a list of task audit logs for tasks that are assigned to a Business Administrator user,
applying a query filter to the list of tasks
 *
 * @param userId identifier of the Business Administrator user
```

```
   * @param filter filter for the list of tasks
   * @return list of @{@link AuditTask} audit tasks
   */
    List<AuditTask> getAllAdminAuditTask(String userId, QueryFilter filter);

    /**
     * Gets a list of task events for the given task
     * @param taskId identifier of the task
     * @param filter for the list of events
   * @return list of @{@link TaskEvent} task events
     */
    List<TaskEvent> getTaskEvents(long taskId, QueryFilter filter);

    /**
     * Query on {@link TaskSummary} instances
     * @param userId the user associated with the tasks queried
     * @return {@link TaskSummaryQueryBuilder} used to create the query
     */
    TaskSummaryQueryBuilder taskSummaryQuery(String userId);

    /**
     * Gets a list of {@link TaskSummary} instances for tasks that define a given variable
     * @param userId the ID of the user associated with the tasks
     * @param variableName the name of the task variable
     * @param statuses the list of statuses that the task can have
     * @param queryContext the query context
     * @return a {@link List} of {@link TaskSummary} instances
     */
    List<TaskSummary> getTasksByVariable(String userId, String variableName, List<Status>
statuses, QueryContext queryContext);

    /**
     * Gets a list of {@link TaskSummary} instances for tasks that define a given variable and
the variable is set to the given value
     * @param userId the ID of the user associated with the tasks
     * @param variableName the name of the task variable
     * @param variableValue the value of the task variable
     * @param statuses the list of statuses that the task can have
     * @param context the query context
     * @return a {@link List} of {@link TaskSummary} instances
     */
    List<TaskSummary> getTasksByVariableAndValue(String userId, String variableName,
String variableValue, List<Status> statuses, QueryContext context);

}
```

66.3.6. 用户任务服务

用户任务服务涵盖了单个任务的整个生命周期，您可以使用该服务从开始到结束来管理用户任务。

任务查询不是用户任务服务的一部分。使用运行时数据服务查询任务。使用用户任务服务在一个任务上有作用域操作，包括以下操作：

- 修改所选属性
- 访问任务变量
- 访问任务附加
- 访问任务评论

用户任务服务也是命令 executor。您可以使用它来执行自定义任务命令。

以下示例显示了启动进程并与进程中的任务交互：

在此过程中启动进程并与用户任务交互

```
long processInstanceId =
processService.startProcess(deployUnit.getIdentifier(), "org.jbpm.writedocument");

List<Long> taskIds =
runtimeDataService.getTasksByProcessInstanceId(processInstanceId);

Long taskId = taskIds.get(0);

userTaskService.start(taskId, "john");
UserTaskInstanceDesc task = runtimeDataService.getTaskById(taskId);

Map<String, Object> results = new HashMap<String, Object>();
results.put("Result", "some document data");
userTaskService.complete(taskId, "john", results);
```

66.3.7. 基于 Quartz 的计时器服务

流程引擎使用 Quartz 提供集群就绪计时器服务。您可以随时使用服务处理或载入 KIE 会话。服务可以管理 KIE 会话的时长，以便相应地触发每个计时器。

以下示例显示了集群环境的基本 Quartz 配置文件：

集群环境的 Quartz 配置文件

```
#============================================================================

# Configure Main Scheduler Properties
#============================================================================


org.quartz.scheduler.instanceName = jBPMClusteredScheduler
org.quartz.scheduler.instanceId = AUTO

#============================================================================

# Configure ThreadPool
#============================================================================


org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 5
org.quartz.threadPool.threadPriority = 5

#============================================================================

# Configure JobStore
#============================================================================


org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.class=org.quartz.impl.jdbcjobstore.JobStoreCMT
org.quartz.jobStore.driverDelegateClass=org.quartz.impl.jdbcjobstore.StdJDBCDelegate
org.quartz.jobStore.useProperties=false
org.quartz.jobStore.dataSource=managedDS
org.quartz.jobStore.nonManagedTXDataSource=nonManagedDS
org.quartz.jobStore.tablePrefix=QRTZ_
org.quartz.jobStore.isClustered=true
org.quartz.jobStore.clusterCheckinInterval = 20000

#============================================================================
# Configure Datasources
#============================================================================
org.quartz.dataSource.managedDS.jndiURL=jboss/datasources/psbpmsDS
org.quartz.dataSource.nonManagedDS.jndiURL=jboss/datasources/quartzNonManagedDS
```

您必须修改上例以符合您的环境。

66.3.8. 查询服务

查询服务提供基于 Dashbuilder 数据集的高级搜索功能。

通过这种方法，您可以控制如何从底层数据存储中检索数据。您可以将复杂的 JOIN 语句用于外部表，如 JPA 实体表或自定义系统数据库表。

查询服务围绕以下两组操作构建：

- 管理操作：
  - 注册查询定义
  - 替换查询定义
  - 取消注册（删除）查询定义
  - 获取查询定义
  - 获取所有注册的查询定义
- 运行时操作：
  - 基于 QueryParam 的简单查询作为过滤器提供程序

- 基于 QueryParamBuilder 作为过滤器提供程序的高级查询

Dashbuilder 数据集支持多个数据源，如 CSV、SQL 和 Elastic Search。但是，流程引擎使用基于 RDBMS 的后端，并专注于基于 SQL 的数据集。

因此，进程引擎查询服务是 Dashbuilder 数据集功能的子集，可通过简单的 API 实现高效查询。

66.3.8.1. 查询服务的关键类

查询服务依赖于以下键类：

- QueryDefinition ：代表数据集的定义。该定义由唯一名称、SQL 表达式（查询） 和源（即查询）以及执行查询时使用的数据源的 JNDI 名称组成。
- QueryParam ：代表单个查询参数或条件的基本结构。此结构由列名称、运算符和预期值组成。
- QueryResultMapper ：将原始数据集数据的类（箭头和列）映射到对象表示。
- QueryParamBuilder ：构建查询过滤器的类，以调用查询定义。

QueryResultMapper

QueryResultMapper 将从数据库(dataset)获取的数据映射到对象表示。它与 ORM 提供程序（如 hibernate ）类似，后者将表映射到实体。

很多对象类型可用于代表 dataset 结果。因此，现有映射程序可能并不总是适合您的需要。QueryResultMapper 中的映射程序是可插拔的，您可以在需要时提供自己的映射程序，以便将 dataset 数据转换为您需要的任何类型。

进程引擎提供以下映射程序：

- org.jbpm.kie.services.impl.query.mapper.ProcessInstanceQueryMapper，使用名称 ProcessInstances注册

- *org.jbpm.kie.services.impl.query.mapper.ProcessInstanceWithVarsQueryMapper，名称为 ProcessInstancesWithVariables*
- *org.jbpm.kie.services.impl.query.mapper.ProcessInstanceWithCustomVarsQueryMapper，在名称 ProcessInstancesWithCustomVariables中注册*
- *org.jbpm.kie.services.impl.query.mapper.UserTaskInstanceQueryMapper，使用名称 UserTasks注册*
- *org.jbpm.kie.services.impl.query.mapper.UserTaskInstanceWithVarsQueryMapper，名称为 UserTasksWithVariables*
- *org.jbpm.kie.services.impl.query.mapper.UserTaskInstanceWithCustomVarsQueryMapper，使用名称 UserTasksWithCustomVariables注册*
- *org.jbpm.kie.services.impl.query.mapper.TaskSummaryQueryMapper，使用 name TaskSummaries注册*
- *org.jbpm.kie.services.impl.query.mapper.RawListQueryMapper，使用名称 RawList注册*

每个 QueryResultMapper 使用唯一字符串名称注册。您可以使用此名称查找映射程序，而不是引用完整类名称。在使用服务的 EJB 远程调用时，此功能尤为重要，因为它可避免依赖客户端上的特定实施。

要根据字符串名称引用 QueryResultMapper，请使用 NamedQueryMapper，这是 jbpm-services-api 模块的一部分。此类充当委托（中继委派）并在执行查询时查找实际映射程序。

使用 NamedQueryMapper

```
queryService.query("my query def", new
NamedQueryMapper<Collection<ProcessInstanceDesc>>("ProcessInstances"), new
QueryContext());
```

QueryParamBuilder

QueryParamBuilder 提供了为数据集构建过滤器的高级方法。

默认情况下，在使用接受零或更多 QueryParam 实例的 QueryService 时，所有这些参数都会与 AND 运算符一起加入，因此数据条目必须与所有这些项匹配。

但是，有时需要更复杂的参数关系。您可以使用 QueryParamBuilder 构建在发布查询时提供过滤器的自定义构建器。

QueryParamBuilder 的一个现有实现在进程引擎中可用。它涵盖了基于 核心功能的默认 QueryParams 。

这些核心功能是基于 SQL 的条件，包括以下条件：

- IS_NULL
- NOT_NULL
- EQUALS_TO
- NOT_EQUALS_TO
- LIKE_TO
- GREATER_THAN

- GREATER_OR_EQUALS_TO
- LOWER_THAN
- LOWER_OR_EQUALS_TO
- 介于
- IN
- NOT_IN

在调用查询前，进程引擎会多次调用 QueryParamBuilder 接口的构建方法，同时方法返回非null值。由于这种方法，您可以构建复杂的过滤器选项，无法通过一个简单的 QueryParams 列表表示。

以下示例显示了 QueryParamBuilder 的基本实施。它依赖于 DashBuilder Dataset API。

QueryParamBuilder的基本实施

```
public class TestQueryParamBuilder implements QueryParamBuilder<ColumnFilter> {

  private Map<String, Object> parameters;
  private boolean built = false;
  public TestQueryParamBuilder(Map<String, Object> parameters) {
     this.parameters = parameters;
  }

  @Override
  public ColumnFilter build() {
     // return null if it was already invoked
     if (built) {
        return null;
     }

     String columnName = "processInstanceId";

     ColumnFilter filter = FilterFactory.OR(
           FilterFactory.greaterOrEqualsTo((Long)parameters.get("min")),
```

```
            FilterFactory.lowerOrEqualsTo((Long)parameters.get("max")));
        filter.setColumnId(columnName);

        built = true;
        return filter;
    }

}
```

在实施构建器后，您可以在对 QueryService 服务执行查询时使用此类实例，如下例所示：

使用 QueryService 服务运行查询

```
queryService.query("my query def", ProcessInstanceQueryMapper.get(), new QueryContext(),
paramBuilder);
```

66.3.8.2. 在典型的场景中使用查询服务

以下流程概述了代码使用查询服务的典型方法。

流程

1. 定义数据集，这是您要使用的数据的视图。使用 services API 中的 QueryDefinition 类来完成此操作：

   定义数据集

   ```
   SqlQueryDefinition query = new SqlQueryDefinition("getAllProcessInstances",
   "java:jboss/datasources/ExampleDS");
   query.setExpression("select * from processinstancelog");
   ```

这个示例代表了最简单的查询定义。

构造器需要以下参数：

- 在运行时标识查询的唯一名称
- 用于对此定义执行查询的 JNDI 数据源名称

setExpression（） 方法的参数是构建数据集视图的 SQL 语句。查询服务中的查询使用来自此视图的数据，并根据需要过滤此数据。

2. 注册查询：

注册查询

```
queryService.registerQuery(query);
```

3. 如果需要，从 dataset 中收集所有数据，而无需过滤：

从 dataset 收集所有数据

```
Collection<ProcessInstanceDesc> instances =
queryService.query("getAllProcessInstances", ProcessInstanceQueryMapper.get(),
new QueryContext());
```

此简单的查询使用 QueryContext 中的默认值进行分页和排序。

4. 如果需要，使用 QueryContext 对象来更改分页和排序的默认值：

使用 QueryContext 对象更改默认值

```
QueryContext ctx = new QueryContext(0, 100, "start_date", true);

Collection<ProcessInstanceDesc> instances =
queryService.query("getAllProcessInstances", ProcessInstanceQueryMapper.get(),
ctx);
```

5. 如果需要，使用查询来过滤数据：

使用查询过滤数据

```
// single filter param
Collection<ProcessInstanceDesc> instances =
queryService.query("getAllProcessInstances", ProcessInstanceQueryMapper.get(),
new QueryContext(), QueryParam.likeTo(COLUMN_PROCESSID, true, "org.jbpm%"));

// multiple filter params (AND)
Collection<ProcessInstanceDesc> instances =
queryService.query("getAllProcessInstances", ProcessInstanceQueryMapper.get(),
new QueryContext(),
 QueryParam.likeTo(COLUMN_PROCESSID, true, "org.jbpm%"),
 QueryParam.in(COLUMN_STATUS, 1, 3));
```

使用查询服务时，您可以定义要获取哪些数据以及如何过滤它。不适用 JPA 提供程序或其他类似限制。您可以为您的环境定制数据库查询以提高性能。

66.3.9. 高级查询服务

高级查询服务提供相应的功能，用于根据进程和任务属性、进程变量和用户任务的内部变量搜索进程和任务。搜索会自动覆盖进程引擎中的所有现有进程。

属性和变量的名称和所需值在 QueryParam 对象中定义。

进程属性包括进程实例 ID、关联键、进程定义 ID 和部署 ID。任务属性包括任务名称、所有者和状态。

可用的搜索方法如下：

- queryProcessByVariables ：根据进程属性和进程变量值列表搜索进程实例。要包含在结果中，进程实例必须具有列出的属性以及进程变量中列出的值。
- queryProcessByVariablesAndTask ：根据进程属性、进程变量值和任务变量值的列表搜索处理实例。要包含在结果中，进程实例必须具有列出的属性以及进程变量中列出的值。它还必须在任务变量中包含列出的值的任务。
- queryUserTasksByVariables ：根据任务属性、任务变量值和进程变量值列表搜索用户任务。要包含在结果中，任务必须具有其任务变量中列出的属性和列出的值。它还必须包含在进程中，其进程变量中列出的值。

该服务由 AdvanceRuntimeDataService 类提供。此类的接口还定义预定义的任务和进程属性名称。

AdvanceRuntimeDataService 接口的定义

```
public interface AdvanceRuntimeDataService {

  String TASK_ATTR_NAME = "TASK_NAME";
  String TASK_ATTR_OWNER = "TASK_OWNER";
  String TASK_ATTR_STATUS = "TASK_STATUS";
  String PROCESS_ATTR_INSTANCE_ID = "PROCESS_INSTANCE_ID";
  String PROCESS_ATTR_CORRELATION_KEY = "PROCESS_CORRELATION_KEY";
  String PROCESS_ATTR_DEFINITION_ID = "PROCESS_DEFINITION_ID";
  String PROCESS_ATTR_DEPLOYMENT_ID = "PROCESS_DEPLOYMENT_ID";
  String PROCESS_COLLECTION_VARIABLES = "ATTR_COLLECTION_VARIABLES";

  List<ProcessInstanceWithVarsDesc> queryProcessByVariables(List<QueryParam>
attributes,
    List<QueryParam> processVariables, QueryContext queryContext);

  List<ProcessInstanceWithVarsDesc> queryProcessByVariablesAndTask(List<QueryParam>
attributes,
    List<QueryParam> processVariables, List<QueryParam> taskVariables,
```

```
        List<String> potentialOwners, QueryContext queryContext);

    List<UserTaskInstanceWithPotOwnerDesc> queryUserTasksByVariables(List<QueryParam>
attributes,
        List<QueryParam> taskVariables, List<QueryParam> processVariables,
        List<String> potentialOwners, QueryContext queryContext);
}
```

66.3.10. 进程实例迁移服务

进程实例迁移服务是一个将进程实例从一个部署迁移到另一个部署的实用程序。进程或任务变量不受迁移的影响。不过，新部署可以使用不同的进程定义。

迁移进程时，进程实例迁移服务还会自动迁移该进程的所有子进程、这些子进程的子进程，等等。如果您在不迁移父进程的情况下尝试迁移子进程，则迁移会失败。

对于最简单的流程迁移，请让活动进程实例完成并在新部署中启动新的进程实例。如果这种方法不适用于您的需要，请在启动进程实例迁移前请考虑以下问题：

- 向后兼容性
- 数据更改
- 需要节点映射

在可能的情况下，通过扩展流程定义来创建向后兼容的进程。例如，从进程定义中删除节点会破坏兼容性。如果进行这样的更改，您必须提供节点映射。如果某个活跃进程实例位于已删除的节点上，则进程实例迁移会使用节点映射。

节点映射包含从旧进程定义中映射到目标节点 ID 的源节点 ID。您只能将同一类型的节点映射到用户任务，如用户任务。

Red Hat Process Automation Manager 提供多个迁移服务实现：

实施迁移服务的 ProcessInstanceMigrationService 接口的方法

```
public interface ProcessInstanceMigrationService {
 /**
 * Migrates a given process instance that belongs to the source deployment into the target
process ID that belongs to the target deployment.
 * The following rules are enforced:
 * <ul>
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the
report for the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instance to be migrated
belongs
 * @param processInstanceId ID of the process instance to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instance should be
migrated
 * @return returns complete migration report
 */
 MigrationReport migrate(String sourceDeploymentId, Long processInstanceId, String
targetDeploymentId, String targetProcessId);
 /**
 * Migrates a given process instance (with node mapping) that belongs to source deployment
into the target process ID that belongs to the target deployment.
 * The following rules are enforced:
 * <ul>
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the
report for the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instance to be migrated
belongs
 * @param processInstanceId ID of the process instance to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instance should be
migrated
 * @param nodeMapping node mapping - source and target unique IDs of nodes to be mapped
- from process instance active nodes to new process nodes
 * @return returns complete migration report
 */
 MigrationReport migrate(String sourceDeploymentId, Long processInstanceId, String
targetDeploymentId, String targetProcessId, Map<String, String> nodeMapping);
 /**
 * Migrates given process instances that belong to the source deployment into a target
process ID that belongs to the target deployment.
 * The following rules are enforced:
```

```
 * <ul>
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the
report for the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instances to be migrated
belong
 * @param processInstanceIds list of process instance IDs to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instances should be
migrated
 * @return returns complete migration report
 */
 List<MigrationReport> migrate(String sourceDeploymentId, List<Long> processInstanceIds,
String targetDeploymentId, String targetProcessId);
 /**
 * Migrates given process instances (with node mapping) that belong to the source
deployment into a target process ID that belongs to the target deployment.
 * The following rules are enforced:
 * <ul>
 * <li>the source deployment ID must point to an existing deployment</li>
 * <li>the process instance ID must point to an existing and active process instance</li>
 * <li>the target deployment must exist</li>
 * <li>the target process ID must exist in the target deployment</li>
 * </ul>
 * Returns a migration report regardless of migration being successful or not; examine the
report for the outcome of the migration.
 * @param sourceDeploymentId deployment to which the process instances to be migrated
belong
 * @param processInstanceIds list of process instance ID to be migrated
 * @param targetDeploymentId ID of the deployment to which the target process belongs
 * @param targetProcessId ID of the process to which the process instances should be
migrated
 * @param nodeMapping node mapping - source and target unique IDs of nodes to be mapped
- from process instance active nodes to new process nodes
 * @return returns list of migration reports one per each process instance
 */
 List<MigrationReport> migrate(String sourceDeploymentId, List<Long> processInstanceIds,
String targetDeploymentId, String targetProcessId, Map<String, String> nodeMapping);
}
```

要在 KIE 服务器上迁移进程实例，请使用以下实施：这些方法与 ProcessInstanceMigrationService 接口中的方法类似，为 KIE 服务器部署提供相同的迁移实施。

在 ProcessAdminServicesClient 界面中，为 KIE 服务器部署实施迁移服务的方法

```
public interface ProcessAdminServicesClient {

    MigrationReportInstance migrateProcessInstance(String containerId, Long
processInstanceId, String targetContainerId, String targetProcessId);

    MigrationReportInstance migrateProcessInstance(String containerId, Long
processInstanceId, String targetContainerId, String targetProcessId, Map<String, String>
nodeMapping);

    List<MigrationReportInstance> migrateProcessInstances(String containerId, List<Long>
processInstancesId, String targetContainerId, String targetProcessId);

    List<MigrationReportInstance> migrateProcessInstances(String containerId, List<Long>
processInstancesId, String targetContainerId, String targetProcessId, Map<String, String>
nodeMapping);
}
```

您可以一次性迁移单个进程实例或多个进程实例。如果您迁移多个进程实例，每个实例都会在一个单独的事务中迁移，以确保迁移不会影响相互影响。

迁移完成后，迁移方法 会返回 包含以下信息的 MigrationReport 对象：

- 迁移的开始和结束日期。
- 迁移结果（成功或失败）。
- INFO、WARN 或 ERROR 类型的日志条目。ERROR 消息终止迁移。

以下示例显示了进程实例迁移：

在 KIE 服务器部署中迁移进程实例

```
import org.kie.server.api.model.admin.MigrationReportInstance;
import org.kie.server.api.marshalling.MarshallingFormat;
```

```
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;

public class ProcessInstanceMigrationTest{

 private static final String SOURCE_CONTAINER = "com.redhat:MigrateMe:1.0";
  private static final String SOURCE_PROCESS_ID = "MigrateMe.MigrateMev1";
 private static final String TARGET_CONTAINER = "com.redhat:MigrateMe:2";
  private static final String TARGET_PROCESS_ID = "MigrateMe.MigrateMeV2";

 public static void main(String[] args) {

  KieServicesConfiguration config =
KieServicesFactory.newRestConfiguration("http://HOST:PORT/kie-
server/services/rest/server", "USERNAME", "PASSWORD");
  config.setMarshallingFormat(MarshallingFormat.JSON);
  KieServicesClient client = KieServicesFactory.newKieServicesClient(config);

  long sourcePid = client.getProcessClient().startProcess(SOURCE_CONTAINER,
SOURCE_PROCESS_ID);

    // Use the 'report' object to return migration results.
  MigrationReportInstance report =
client.getAdminClient().migrateProcessInstance(SOURCE_CONTAINER,
sourcePid,TARGET_CONTAINER, TARGET_PROCESS_ID);

  System.out.println("Was migration successful:" + report.isSuccessful());

  client.getProcessClient().abortProcessInstance(TARGET_CONTAINER, sourcePid);

 }
}
```

已知进程实例迁移的限制

以下情况可能会导致迁移失败或不正确的迁移：

- *新的或修改的任务需要迁移的进程实例中没有可用的输入。*
- *您可以在活动任务之前修改任务，其中的更改会对进一步处理产生影响。*
- *您删除当前处于活跃状态的人工任务。要替换人工任务，您必须将它映射到另一人任务。*
- *您可以在单个活动任务中添加一个并行的新任务。因为没有激活 AND 网关中的所有分支，*

进程会卡住。

- 您删除活跃的计时器事件（数据库中不会更改这些事件）。
- 您可以在活跃的任务中修复或更新输入和输出（任务数据没有被迁移）。

如果将映射应用到任务节点，则只会映射任务节点名称和描述。其他任务字段（包括 TaskName 变量）没有映射到新任务。

66.3.11. 部署和不同的进程版本

部署服务将商业资产放入执行环境中。然而，在某些情况下需要其他管理才能使资产在正确的上下文中可用。值得注意的是，如果部署多个版本的同一进程，您必须确保进程实例使用正确的版本。

部署激活和取消激活

在某些情况下，一些进程实例在部署上运行，然后将同一进程的新版本添加到运行时环境中。

您可能决定此进程定义的新实例必须使用新版本，而现有活动实例则应该继续使用以前的版本。

要启用这种情况，请使用以下部署服务方法：

- 激活 ：激活部署，以便它可用于交互。您可以列出其进程定义，并启动此部署的新进程实例。
- 取消激活 ：取消激活部署。禁用 选项，以列出进程定义，并在部署中启动新的进程实例。但是，您可以继续操作已经激活的进程实例，例如信号事件并与用户任务交互。

您可以使用此功能在项目版本间平稳过渡，而无需流程实例迁移。

调用进程的最新版本

如果您需要使用最新版本的项目进程，您可以使用 latest 关键字与服务中的多个操作进行交互。只有在进程标识符在所有版本中都保持不变时，才支持此方法。

以下示例说明了其功能。

初始部署单元是 org.jbpm:HR:1.0。它包含招聘流程的第一个版本。

几周后，您将开发新版本，并将其部署为 org.jbpm:HR.2.0。它包括招聘流程的版本 2。

如果要调用进程并确保您使用最新版本，您可以使用以下部署 ID：

> org.jbpm.HR:latest

如果您使用此部署 ID，则进程引擎会找到项目的最新版本。它使用以下标识符：

- *Group Id :org.jbpm*
- *artifactId:HR*

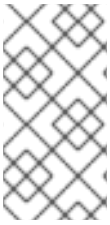
版本号由 Maven 规则进行比较，以查找最新版本。

以下代码示例演示了部署多个版本，并与最新版本交互：

部署多个进程版本，并与最新版本交互

```
KModuleDeploymentUnit deploymentUnitV1 = new KModuleDeploymentUnit("org.jbpm", "HR",
"1.0");
deploymentService.deploy(deploymentUnitV1);

long processInstanceId = processService.startProcess("org.jbpm:HR:LATEST",
"customtask");
ProcessInstanceDesc piDesc =
runtimeDataService.getProcessInstanceById(processInstanceId);

// We have started a process with the project version 1
assertEquals(deploymentUnitV1.getIdentifier(), piDesc.getDeploymentId());

// Next we deploy version 2
```

```
KModuleDeploymentUnit deploymentUnitV2 = new KModuleDeploymentUnit("org.jbpm", "HR",
"2.0");
deploymentService.deploy(deploymentUnitV2);

processInstanceId = processService.startProcess("org.jbpm:HR:LATEST", "customtask");
piDesc = runtimeDataService.getProcessInstanceById(processInstanceId);

// This time we have started a process with the project version 2
assertEquals(deploymentUnitV2.getIdentifier(), piDesc.getDeploymentId());
```



注意

这个功能也可以在 KIE 服务器 REST API 中提供。使用部署 ID 发送请求时，您可以使用 LATEST 作为版本标识符。

其他资源

- 使用 KIE API 与 Red Hat Process Automation Manager 交互

66.3.12. 部署同步

流程引擎服务包括部署同步器，将可用部署存储到数据库中，包括部署每个部署的部署描述符。

同步器还会监控此表，使其与可能使用相同的数据源的其他安装同步。当在一个集群中运行或 Business Central 和自定义应用程序必须在同一工件上运行时，这个功能尤为重要。

默认情况下，在运行核心服务时，您必须配置同步。对于 EJB 和 CDI 扩展，自动启用同步。

以下代码示例配置同步：

配置同步

```
TransactionalCommandService commandService = new TransactionalCommandService(emf);

DeploymentStore store = new DeploymentStore();
store.setCommandService(commandService);
```

```
DeploymentSynchronizer sync = new DeploymentSynchronizer();
sync.setDeploymentService(deploymentService);
sync.setDeploymentStore(store);

DeploymentSyncInvoker invoker = new DeploymentSyncInvoker(sync, 2L, 3L,
TimeUnit.SECONDS);
invoker.start();
....
invoker.stop();
```

借助此配置，部署每三秒钟同步，初始延迟为 2 秒。

66.4. 进程引擎中的线程

我们可引用两种类型的多线程：逻辑和技术。技术多线程涉及多个线程或启动的进程，例如通过 Java 或 C 程序。在 BPM 过程中发生逻辑多线程，例如，在进程到达并行网关后。在执行逻辑中，原始进程被分成两个以并行方式运行的进程。

流程引擎代码使用一个技术线程实施逻辑多线程。

这种设计选择的原因是，多个（技术）线程必须能够在同一进程上相互交流状态信息。这个要求会产生一些复杂情况。在线程间安全通信需要额外的逻辑，以及避免竞争条件和死锁所需的额外开销，可以降低使用这些线程的性能优势。

通常，进程引擎执行一系列操作。例如，当进程引擎在进程中遇到脚本任务时，它会同步执行脚本并等待其完成，然后再继续执行。同样，如果进程遇到并行网关，进程引擎按顺序触发每个传出分支，另一个是这样。

这是因为执行几乎总是瞬间，这意味着它非常快，生成几乎没有开销。因此，顺序执行不会产生用户可能会发现的任何影响。

您提供的过程中的任何代码也执行同步，进程引擎在继续操作前等待其完成。例如，如果您使用 Thread.sleep(…) 作为自定义脚本的一部分，则进程引擎线程在休眠期间被阻断。

当进程到达服务任务时，进程引擎还会异步调用处理程序，并等待 completeWorkItem(…) 方法返回，然后再继续执行。如果您的服务处理程序不瞬间，请在您的代码中单独实施异步执行。

例如，您的服务任务可能会调用外部服务。远程调用此服务和等待结果的延迟可能非常显著。因此，异步调用此服务。您的处理程序必须仅调用该服务，然后从方法返回，然后在结果可用时通知进程引擎。同时，流程引擎可以继续执行进程。

人工任务是需要异步调用的服务的典型示例。人工任务需要人工响应请求，流程引擎不得等待这一响应。

触发人工任务节点时，人工任务处理程序仅在所分配人的任务列表上创建一个新任务。然后，流程引擎可以继续在剩余的流程中继续执行（如有必要）。当用户完成任务时，处理程序会异步通知流程引擎。

66.5. 进程引擎中的执行错误

任何流程引擎执行的一部分，包括任务服务，可以抛出异常。一个例外是扩展 java.lang.Throwable 的任何类。

在进程级别上处理一些例外。值得注意的是，工作项目处理程序可能会引发自定义异常，用于指定错误处理的子进程。如需有关开发工作项目处理程序的信息，请参阅 自定义任务和工作处理程序。

如果没有处理异常并到达进程引擎，它将变成 执行错误。当发生执行错误时，进程引擎会回滚当前的事务并使进程处于之前稳定状态。之后，进程引擎将继续从该点执行进程。

对向进程引擎发送请求的调用者可见执行错误。流程引擎还包括用于处理执行错误的扩展机制，并存储有关它们的信息。这种机制由以下组件组成：

- ExecutionErrorManager ：错误处理的入口点。此类与 RuntimeManager 集成，后者负责为其提供底层 KieSession 和 TaskService。ExecutionErrorManager 提供对执行错误处理机制中的其他类的访问。

  当进程引擎创建 RuntimeManager 实例时，它还会创建一个对应的 ExecutionErrorManager 实例。

- ExecutionErrorHandler ：错误处理的主要类。这个类在进程引擎中实施，您通常不需要自定义或扩展它。ExecutionErrorHandler 调用错误过滤器，以处理特定错误并调用 ExecutionErrorStorage 来存储错误信息。

ExecutionErrorHandler 绑定到 RuntimeEngine 的生命周期；在创建新运行时引擎时会创建它，并在运行时引擎被销毁。ExecutionErrorHandler 的单个实例在给定的执行上下文或事务中使用。KieSession 和 TaskService 都使用该实例来告知处理已处理节点或任务的错误处理。有关以下事件通知 ExecutionErrorHandler ：

- 启动节点实例的处理
- 完成节点实例的处理
- 任务实例的处理
- 完成任务实例的处理

ExecutionErrorHandler 使用此信息记录错误的上下文，特别是错误本身未提供进程上下文信息。例如，数据库例外没有任何进程信息。

- ExecutionErrorStorage ：用于执行错误信息的可插拔存储类。

当进程引擎创建 RuntimeManager 实例时，它还会创建一个对应的 ExecutionErrorStorage 实例。然后 ExecutionErrorHandler 类调用这个 ExecutionErrorStorage 实例，以存储每个执行错误的信息。

默认的存储实施使用数据库表存储每个错误的所有可用信息。不同错误类型可能有不同的详细级别，因为某些错误可能不允许提取详细信息。

- 处理特定类型的执行错误的多个 过滤器。您可以添加自定义过滤器。

默认情况下，记录每个执行错误作为 未确认的。您可以使用 Business Central 查看所有记录的执行错误并确认它们。您还可以创建自动确认所有或执行错误的作业。

有关使用 Business Central 查看执行错误以及创建可自动确认错误的作业，请参阅 Business Central 中管理和监控业务流程的信息。

66.5.1. 执行错误类型和过滤器

执行错误处理会尝试捕获和处理任何类型的错误。但是，用户需要以不同的方式处理不同的错误。另外，针对不同类型的错误，还提供了不同的详细信息。

错误处理机制支持可插拔 过滤器。每个过滤器处理特定类型的错误。您可以通过不同的方式添加处理特定错误的过滤器，覆盖默认处理。

过滤器是 ExecutionErrorFilter 接口的实现。此接口构建 ExecutionError 的实例，稍后使用 ExecutionErrorStorage 类存储。

ExecutionErrorFilter 接口有以下方法：

- accept ：指示某个错误是否可以被过滤器处理
- 过滤器: 处理错误并返回 ExecutionError 实例
- getPriority ：指示此过滤器的优先级

执行错误处理程序会单独处理每个错误。对于每个错误，它开始调用所有注册过滤器的 接受 方法，从具有较低优先级值的过滤器开始。如果过滤器的 接受 方法返回为 true，处理程序会调用 过滤器 的过滤器方法，不会调用任何其他过滤器。

由于优先级系统，只有一个过滤器处理任何错误。更为专业的过滤器具有较低优先级值。任何专用过滤器不接受的错误，会到达具有更高优先级值的通用过滤器。

ServiceLoader 机制提供 ExecutionErrorFilter 实例。要注册自定义过滤器，将其完全限定的类名称添加到服务项目的 META-INF/services/org.kie.internal.runtime.error.ExecutionErrorFilter 文件中。

Red Hat Process Automation Manager 附带以下执行错误过滤器：

表 66.1. ExecutionErrorFilters

类名称	类型	优先级
org.jbpm.runtime.manager.impl.error.filters.ProcessExecutionErrorFilter	Process	100
org.jbpm.runtime.manager.impl.error.filters.TaskExecutionErrorFilter	任务	80
org.jbpm.runtime.manager.impl.error.filters.DBExecutionErrorFilter	DB	200
org.jbpm.executor.impl.error.JobExecutionErrorFilter	作业	100

根据优先级最低值，为过滤器赋予更高的执行顺序。因此，执行错误处理器按照以下顺序调用这些过滤器：

1. 任务
2. Process
3. 作业
4. DB

66.6. 进程引擎中的事件监听程序

每次进程或任务更改其生命周期中的不同点时，进程引擎都会生成事件。您可以开发接收和处理此类事件的类。此类称为 事件监听程序。

进程引擎将事件对象传递到此类。对象提供对相关信息的访问。例如，如果事件与进程节点相关，对象则提供对进程实例和节点实例的访问。

66.6.1. 事件监听程序的接口

您可以使用以下接口为进程引擎开发事件监听程序。

66.6.1.1. 进程事件监听程序的接口

您可以开发实现 ProcessEventListener 接口的类。此类可以侦听与进程相关的事件，如启动或完成进程或输入和离开节点。

以下源代码显示了 ProcessEventListener 接口的不同方法：

ProcessEventListener 接口

```
public interface ProcessEventListener
    extends
    EventListener {

    /**
     * This listener method is invoked right before a process instance is being started.
     * @param event
     */
    void beforeProcessStarted(ProcessStartedEvent event);

    /**
     * This listener method is invoked right after a process instance has been started.
     * @param event
     */
    void afterProcessStarted(ProcessStartedEvent event);

    /**
     * This listener method is invoked right before a process instance is being completed (or
aborted).
     * @param event
     */
    void beforeProcessCompleted(ProcessCompletedEvent event);

    /**
     * This listener method is invoked right after a process instance has been completed (or
aborted).
     * @param event
     */
    void afterProcessCompleted(ProcessCompletedEvent event);

    /**
     * This listener method is invoked right before a node in a process instance is being
triggered
     * (which is when the node is being entered, for example when an incoming connection
triggers it).
     * @param event
```

```
     */
    void beforeNodeTriggered(ProcessNodeTriggeredEvent event);

    /**
     * This listener method is invoked right after a node in a process instance has been
triggered
     * (which is when the node was entered, for example when an incoming connection
triggered it).
     * @param event
     */
    void afterNodeTriggered(ProcessNodeTriggeredEvent event);

    /**
     * This listener method is invoked right before a node in a process instance is being left
     * (which is when the node is completed, for example when it has performed the task it was
     * designed for).
     * @param event
     */
    void beforeNodeLeft(ProcessNodeLeftEvent event);

    /**
     * This listener method is invoked right after a node in a process instance has been left
     * (which is when the node was completed, for example when it performed the task it was
     * designed for).
     * @param event
     */
    void afterNodeLeft(ProcessNodeLeftEvent event);

    /**
     * This listener method is invoked right before the value of a process variable is being
changed.
     * @param event
     */
    void beforeVariableChanged(ProcessVariableChangedEvent event);

    /**
     * This listener method is invoked right after the value of a process variable has been
changed.
     * @param event
     */
    void afterVariableChanged(ProcessVariableChangedEvent event);

    /**
     * This listener method is invoked right before a process/node instance's SLA has been
violated.
     * @param event
     */
    default void beforeSLAViolated(SLAViolatedEvent event) {}

    /**
     * This listener method is invoked right after a process/node instance's SLA has been
violated.
     * @param event
     */
    default void afterSLAViolated(SLAViolatedEvent event) {}
```

```
    /**
     * This listener method is invoked when a signal is sent
     * @param event
     */
    default void onSignal(SignalEvent event) {}

    /**
     * This listener method is invoked when a message is sent
     * @param event
     */
    default void onMessage(MessageEvent event) {}
}
```

您可以实施任何这些方法来处理相应的事件。

有关进程引擎传递给方法的事件类的定义，请参阅 Java 文档中的 org.kie.api.event.process 软件包。

您可以使用事件类的方法检索包含事件中涉及实体的所有信息的其他类。

以下示例是节点相关事件的一部分，如 afterNodeLeft（），并检索进程实例和节点类型。

在节点相关事件中检索进程实例和节点类型

```
WorkflowProcessInstance processInstance = event.getNodeInstance().getProcessInstance()
NodeType nodeType = event.getNodeInstance().getNode().getNodeType()
```

66.6.1.2. 任务生命周期事件监听程序的接口

您可以开发实现 TaskLifecycleEventListener 接口的类。此类可以侦听与任务生命周期相关的事件，如分配所有者或完成任务。

以下源代码显示了 TaskLifecycleEventListener 接口的不同方法：

TaskLifecycleEventListener 接口

```
public interface TaskLifeCycleEventListener extends EventListener {

    public enum AssignmentType {
        POT_OWNER,
        EXCL_OWNER,
        ADMIN;
    }

    public void beforeTaskActivatedEvent(TaskEvent event);
    public void beforeTaskClaimedEvent(TaskEvent event);
    public void beforeTaskSkippedEvent(TaskEvent event);
    public void beforeTaskStartedEvent(TaskEvent event);
    public void beforeTaskStoppedEvent(TaskEvent event);
    public void beforeTaskCompletedEvent(TaskEvent event);
    public void beforeTaskFailedEvent(TaskEvent event);
    public void beforeTaskAddedEvent(TaskEvent event);
    public void beforeTaskExitedEvent(TaskEvent event);
    public void beforeTaskReleasedEvent(TaskEvent event);
    public void beforeTaskResumedEvent(TaskEvent event);
    public void beforeTaskSuspendedEvent(TaskEvent event);
    public void beforeTaskForwardedEvent(TaskEvent event);
    public void beforeTaskDelegatedEvent(TaskEvent event);
    public void beforeTaskNominatedEvent(TaskEvent event);
    public default void beforeTaskUpdatedEvent(TaskEvent event){};
    public default void beforeTaskReassignedEvent(TaskEvent event){};
    public default void beforeTaskNotificationEvent(TaskEvent event){};
    public default void beforeTaskInputVariableChangedEvent(TaskEvent event, Map<String,
Object> variables){};
    public default void beforeTaskOutputVariableChangedEvent(TaskEvent event, Map<String,
Object> variables){};
    public default void beforeTaskAssignmentsAddedEvent(TaskEvent event, AssignmentType
type, List<OrganizationalEntity> entities){};
    public default void beforeTaskAssignmentsRemovedEvent(TaskEvent event,
AssignmentType type, List<OrganizationalEntity> entities){};

    public void afterTaskActivatedEvent(TaskEvent event);
    public void afterTaskClaimedEvent(TaskEvent event);
    public void afterTaskSkippedEvent(TaskEvent event);
    public void afterTaskStartedEvent(TaskEvent event);
    public void afterTaskStoppedEvent(TaskEvent event);
    public void afterTaskCompletedEvent(TaskEvent event);
    public void afterTaskFailedEvent(TaskEvent event);
    public void afterTaskAddedEvent(TaskEvent event);
    public void afterTaskExitedEvent(TaskEvent event);
    public void afterTaskReleasedEvent(TaskEvent event);
    public void afterTaskResumedEvent(TaskEvent event);
    public void afterTaskSuspendedEvent(TaskEvent event);
    public void afterTaskForwardedEvent(TaskEvent event);
    public void afterTaskDelegatedEvent(TaskEvent event);
    public void afterTaskNominatedEvent(TaskEvent event);
    public default void afterTaskReassignedEvent(TaskEvent event){};
```

```
    public default void afterTaskUpdatedEvent(TaskEvent event){};
    public default void afterTaskNotificationEvent(TaskEvent event){};
    public default void afterTaskInputVariableChangedEvent(TaskEvent event, Map<String,
Object> variables){};
    public default void afterTaskOutputVariableChangedEvent(TaskEvent event, Map<String,
Object> variables){};
    public default void afterTaskAssignmentsAddedEvent(TaskEvent event, AssignmentType
type, List<OrganizationalEntity> entities){};
    public default void afterTaskAssignmentsRemovedEvent(TaskEvent event,
AssignmentType type, List<OrganizationalEntity> entities){};

}
```

您可以实施任何这些方法来处理相应的事件。

有关进程引擎传递给方法的事件类的定义，请参阅 Java 文档中的 org.kie.api.task 软件包。

您可以使用事件类的方法检索代表任务、任务上下文和任务元数据的类。

66.6.2. 对事件监听器调用的时间

一个事件监听器调用的数量是 before 和 after events，如 beforeNodeLeft（） 和 afterNodeLeft（），beforeTaskActivatedEvent（） 和 afterTaskActivatedEvent（）。

事件调用的 before 和 after 通常会像堆栈一样工作。如果事件 A 直接导致事件 B，则会出现以下调用序列：

- 前 A
- B 之前
- B 后
- A 后

例如，如果保留节点 X 触发节点 Y，则所有与触发节点 Y 的事件调用在节点 X 的 beforeNodeLeft 和 afterNodeLeft 调用之间发生。

同样，如果启动进程直接导致某些节点启动，则所有 nodeTriggered 和 nodeLeft 事件调用在 beforeProcessStarted 和 afterProcessStarted 调用之间发生。

此方法反映了事件之间原因和效果。但是，事件调用 的时间和 顺序并不总是直观。例如，在对过程中某些节点的 afterNodeLeft 调用后，可能会出现 afterProcessStarted 调用。

通常，在发生特定事件时获得通知，请使用 before 调用事件。只有在要确保与这个事件相关的所有处理均已结束时，才使用 after 调用。例如，当您要获得与启动特定进程实例关联的所有步骤时，请通知。

根据节点的类型，一些节点可能只生成 nodeLeft 调用，而其他节点只能生成 nodeTriggered 调用。例如，捕获中间事件节点不会生成 nodeTriggered 调用，因为它们不会被另一个进程节点触发。同样，引发中间事件节点不会生成 nodeLeft 调用，因为这些节点没有到另一节点的传出连接。

66.6.3. 事件监听程序开发实践

进程引擎在处理事件或任务期间调用事件监听程序。这个调用发生在进程引擎事务和块执行中。因此，事件监听器可能会影响进程引擎的逻辑和性能。

为确保最小中断，请遵循以下准则：

- 任何操作都必须尽可能短。
- 侦听器类不能具有 状态。进程引擎可以随时销毁和重新创建侦听器类。
- 如果监听器修改任何存在于监听器方法范围之外的资源，请确保资源在当前事务中被列出。事务可能会被回滚。在这种情况下，如果修改的资源不是事务的一部分，则资源的状态会不一致。

  红帽 JBoss EAP 提供的数据库相关资源始终被纳入当前事务中。在其他情况下，检查您使用的运行时环境的信息。

- 不要使用依赖于不同事件监听器执行顺序的逻辑。
- 不要在监听器内包含与进程引擎之外的不同实体的交互。例如，不要包含事件通知的 REST 调用。反之，使用进程节点来完成此类调用。一个例外是日志信息的输出，但日志记录监听器必须尽可能简单。
- 您可以使用侦听器修改事件中涉及的进程或任务的状态，例如，更改其变量。
- 您可以使用监听程序与进程引擎交互，例如发送信号或与事件中不涉及的进程实例交互。

66.6.4. 注册事件监听程序

KieSession 类实施 RuleRuntimeEventManager 接口，它提供注册、删除和列出事件监听程序的方法，如下表所示。

RuleRuntimeEventManager 接口的方法

```
void addEventListener(AgendaEventListener listener);
void addEventListener(RuleRuntimeEventListener listener);
void removeEventListener(AgendaEventListener listener);
void removeEventListener(RuleRuntimeEventListener listener);
Collection<AgendaEventListener> getAgendaEventListeners();
Collection<RuleRuntimeEventListener> getRuleRintimeEventListeners();
```

但是，在典型的情况下，不要使用这些方法。

如果使用 RuntimeManager 接口，您可以使用 RuntimeEnvironment 类来注册事件监听程序。

如果使用 Services API，您可以在项目中的 META-INF/services/org.jbpm.services.task.deadlines.NotificationListener 文件中添加事件监听程序的完全限定类名称。服务 API 还注册一些默认监听程序，包括 org.jbpm.services.task.deadlines.notifications.impl.email.EmailNotificationListener，它可以为事件发送电子邮件通知。

要排除默认监听器，您可以将监听器的完全限定名称添加到 org.kie.jbpm.notification_listeners.exclude JVM 系统属性。

66.6.5. KieRuntimeLogger 事件监听程序

KieServices 软件包包含 KieRuntimeLogger 事件监听程序，您可以添加到 KIE 会话。您可以使用此监听程序创建审计日志。此日志包含运行时发生的所有不同事件。

注意

这些日志记录器主要用于调试目的。在业务级别流程分析方面，它们可能过于详细。

侦听器实施以下日志记录器类型：

- 控制台日志记录器：该日志记录器将所有事件写入控制台。此日志记录器的完全限定类名称是 org.drools.core.audit.WorkingMemoryConsoleLogger。
- 文件日志记录器：该日志记录器使用 XML 表示将所有事件写入文件。您可以使用 IDE 中的日志文件来生成在执行过程中发生的事件的树形视觉化。此日志记录器的完全限定类名称是 org.drools.core.audit.WorkingMemoryFileLogger。

  文件日志记录器仅在关闭日志记录器时或日志记录器中的事件数量达到预定义的级别时将事件写入磁盘。因此，它不适用于在运行时调试进程。
- 线程文件日志记录器：此日志记录器在指定时间间隔后将事件写入文件中。您可以使用此日志记录器在调试进程时实时视觉化进度。此日志记录器的完全限定类名称是 org.drools.core.audit.ThreadedWorkingMemoryFileLogger。

在创建日志记录器时，您必须将 KIE 会话作为参数传递。文件日志记录器还需要创建日志文件的名称。线程文件日志记录器需要保存事件的间隔（以毫秒为单位）。

始终关闭应用程序末尾的日志记录器。

下例演示了使用文件日志记录器。

使用文件日志记录器

```
import org.kie.api.KieServices;
import org.kie.api.logger.KieRuntimeLogger;
...
KieRuntimeLogger logger = KieServices.Factory.get().getLoggers().newFileLogger(ksession,
"test");
// add invocations to the process engine here,
// e.g. ksession.startProcess(processId);
...
logger.close();
```

由基于文件的日志记录器创建的日志文件包含有关进程运行时发生的所有事件的基于 XML 的概览。

66.7. 进程引擎配置

您可以使用多个控制参数来更改流程引擎默认行为，以满足您的环境的要求。

这些参数设置为 JVM 系统属性，通常是在启动应用服务器等程序时使用 -D 选项。

表 66.2. 控制参数

Name	可能的值	默认值	描述
jbpm.ut.jndi.lookup	字符串		没有访问默认名称时使用的备选 JNDI 名称 (**java:comp/UserTransaction**)。 注意：在给定的运行时环境中名称必须有效。如果没有访问默认用户事务 JNDI 名称，则不要使用此变量。
jbpm.enable.multi.con	**true|false**	**false**	启用对活动的多个传入和传出序列流程
jbpm.business.calendar.properties	字符串	**/jbpm.business.calendar.properties**	商业日历配置文件的替代类路径位置

Name	可能的值	默认值	描述
jbpm.overdue.timer.delay	Long	2000	指定超过计时器以便允许正确初始化的延迟（以毫秒为单位）
jbpm.process.name.comparator	字符串		按名称启用进程的其他比较器类，默认使用 **NumberVersionComparator** 比较器
jbpm.loop.level.disabled	**true**|**false**	**true**	在使用 XOR 网关时，启用或禁用对高级循环支持的循环迭代跟踪
org.kie.mail.session	字符串	**mail/jbpmMailSession**	任务 Deadlines 使用的邮件会话的替代 JNDI 名称
jbpm.usergroup.callback.properties	字符串	**/jbpm.usergroup.callback.properties**	用户组回调实现的其他类路径位置（LDAP、DB）
jbpm.user.group.mapping	字符串	**${jboss.server.config.dir}/roles.properties**	**JBossUserGroupCallbackImpl**的 **roles.properties** 文件的替代位置
jbpm.user.info.properties	字符串	**/jbpm.user.info.properties**	用户 info 配置的替代类路径位置（供 **LDAPUserInfoImpl**使用）
org.jbpm.ht.user.separator	字符串	**,**	用户任务的其他作用和组群的分隔符
org.quartz.properties	字符串		用于激活基于 Quartz 的计时器服务的 Quartz 配置文件的位置
jbpm.data.dir	字符串	**${JBoss.server.data.dir}**（如果可用），否则 **${java.io.tmpdir}**	存储进程引擎生成的数据文件的位置
org.kie.executor.pool.size	整数	**1**	进程引擎执行器的线程池大小
org.kie.executor.retry.count	整数	3	错误时由进程引擎执行程序尝试的重试次数

Name	可能的值	默认值	描述
org.kie.executor.interval	整数	0	进程引擎执行器检查待处理作业的频率（以秒为单位）。如果值为 0，则检查会在 executor 启动期间运行一次。
org.kie.executor.disabled	**true**|**false**	**true**	禁用进程引擎执行器
org.kie.store.services.class	字符串	**org.drools.persistence.jpa.KnowledgeStoreServiceImpl**	实施 **KieStoreServices** 的类的完全限定名称，负责引导 **KieSession** 实例
org.kie.jbpm.notification_listeners.exclude	字符串		即使在其他情况下必须排除的事件监听程序的完全限定名称，即使它们被使用。使用逗号分隔多个名称。例如，您可以添加 **org.jbpm.services.task.deadlines.notifications.impl.email.EmailNotificationListener**，以排除默认的电子邮件通知监听程序。
org.kie.jbpm.notification_listeners.include	字符串		必须包括的事件监听程序的完全限定名称。使用逗号分隔多个名称。如果您设置了此属性，则只包括此属性中的监听程序，所有其他监听程序也会被排除。

第 67 章 流程引擎中的持久性和事务

进程引擎实施进程状态的持久性。该实施使用带有 SQL 数据库后端的 JPA 框架。它还可以将审计日志信息存储在数据库中。

流程引擎还支持使用 JTA 框架执行进程的事务，依靠持久后端支持事务。

67.1. 进程运行时状态的持久性

进程引擎支持运行进程实例的运行时状态的持久性存储。由于它存储了运行时状态，如果进程引擎停止或遇到问题，它可以继续执行进程实例。

进程引擎还会持久存储当前和之前进程状态的进程定义和历史日志。

您可以使用 JPA 框架指定的 persistence.xml 文件在 SQL 数据库中配置持久性。您可以插入不同的持久性策略。有关 persistence.xml 文件的详情请参考 第 67.4.1 节 “在 persistence.xml 文件中配置”。

默认情况下，如果您没有在进程引擎中配置持久性，则进程信息（包括进程实例状态）不会被具有持久性。

当进程引擎启动进程时，它会创建一个 进程实例，它代表在该特定上下文中执行进程。例如，在执行处理销售订购的进程时，会为每个销售请求创建一个进程实例。

进程实例包含进程的当前运行时状态和上下文，包括任何进程变量的当前值。但是，它不包括关于该进程过去状态历史记录的信息，因为此信息不需要持续执行进程。

当进程实例的运行时状态具有持久性时，您可以在进程引擎失败或停止时恢复所有正在运行的进程的执行状态。您还可以从内存中删除特定进程实例，然后在以后恢复它。

如果您将进程引擎配置为使用持久性，它会自动将运行时状态存储到数据库中。您不需要在代码中触发持久性。

当您从数据库恢复进程引擎状态时，所有实例都会自动恢复到其上次记录的状态。如果进程实例被触发，例如，已过期的计时器、完成进程实例请求的任务或向进程实例发送信号时，进程实例会自动恢复。

您不需要加载单独的实例，并手动触发其执行。

流程引擎还会根据需要自动重新加载进程实例。

67.1.1. 安全持久性点

进程引擎在执行过程中，将进程实例的状态保存在 安全点上。

当进程实例启动或恢复之前等待状态的执行时，进程引擎将继续执行，直到无法执行更多操作。如果不能执行更多操作，这表示进程已完成，或者已到达等待状态。如果进程包含多个并行路径，则所有路径都必须到达等待状态。

在执行过程中，这个点被视为一个安全点。此时，进程引擎将进程实例的状态以及受执行影响的任何其他进程实例存储在持久性存储中。

67.2. 持久性审计日志

流程引擎可以存储关于进程实例执行的信息，包括实例的连续历史状态。

很多情况下，这些信息很有用。例如，您可能希望验证已为特定进程实例执行哪些操作，或监控和分析特定进程的效率。

但是，在运行时数据库中存储历史记录信息会导致数据库迅速增加的大小，并会影响到持久层的性能。因此，历史记录日志信息会单独存储。

进程引擎基于进程在执行过程中生成的事件创建日志。它使用事件监听程序机制接收事件并提取必要的信息，然后将此信息保留在数据库中。jbpm-audit 模块包含一个事件监听程序，它利用 JPA 在数据库中存储与进程相关的信息。

您可以使用过滤器来限制日志信息的范围。

67.2.1. 进程引擎审计日志数据模型

您可以查询进程引擎审计日志的日志信息，以便在不同的情况下使用它，例如为一个特定的进程实例创建历史记录日志，或分析特定进程的所有实例的性能。

审计日志数据模型是默认的实现。根据您的用例，您还可以定义自己的数据模型来存储所需信息。您可以使用进程事件监听程序来提取信息。

数据模型包含三个实体：一个用于处理实例信息，一个用于节点实例信息，一个用于处理变量实例信息。

ProcessInstanceLog 表包含有关进程实例的基本日志信息。

表 67.1. ProcessInstanceLog 表字段

字段	描述	nullable
id	日志实体的主密钥和 ID	不是 NULL
correlationKey	此进程实例的关联	
duration	自从开始日期开始以来，此进程实例的实际持续时间	
end_date	适用时，进程实例的结束日期	
externalId	用于与一些元素关联的可选外部标识符，如部署 ID	
user_identity	启动进程实例的用户的可选标识符	
结果	进程实例的结果。如果进程实例完成并带有错误事件，则此字段包含错误代码。	
parentProcessInstanceId	父进程实例的进程实例 ID（如果适用）	
processid	进程的 ID	
processinstanceid	进程实例 ID	不是 NULL
processname	进程的名称	
processtype	实例的类型（处理或问题单）	
processversion	进程的版本	

字段	描述	nullable
sla_due_date	根据服务级别协议(SLA)的到期日期	
slaCompliance	SLA 合规级别	
start_date	进程实例的开始日期	
status	映射到进程实例状态的进程实例的状态	

NodeInstanceLog 表包含有关各个进程实例内执行哪些节点的更多信息。每当从其进入的连接中输入某个节点实例时，或通过其中一个传出连接退出，则有关该事件的信息都会存储在此表中。

表 67.2. NodeInstanceLog 表字段

字段	描述	nullable
id	日志实体的主密钥和 ID	不是 NULL
连接	导致此节点实例的序列流的实际识别符	
log_date	事件的日期	
externalId	用于与一些元素关联的可选外部标识符，如部署 ID	
NODEID	进程定义中对应节点的节点 ID	
nodeinstanceid	节点实例 ID	
nodename	节点的名称	
nodetype	节点的类型	
processid	进程实例正在执行的进程 ID	
processinstanceid	进程实例 ID	不是 NULL
sla_due_date	根据服务级别协议(SLA)的到期日期。	
slaCompliance	SLA 合规级别	

字段	描述	nullable
type	事件的类型(0 = enter, 1 = exit)	不是 NULL
workItemId	（可选，仅适用于某些节点类型）工作项目的标识符	
nodeContainerId	容器的标识符，如果节点位于嵌入式子进程节点中	
referenceId	引用标识符	
影响	如果节点是调度的事件类型，则原始节点实例 ID 和作业 ID。您可以使用这些信息再次触发作业。	

VariableInstanceLog 表包含有关变量实例中更改的信息。默认情况下，进程引擎在变量更改其值后生成日志条目。流程引擎也可以在更改之前记录条目。

表 67.3. VariableInstanceLog 表字段

字段	描述	nullable
id	日志实体的主密钥和 ID	不是 NULL
externalId	用于与一些元素关联的可选外部标识符，如部署 ID	
log_date	事件的日期	
processid	进程实例正在执行的进程 ID	
processinstanceid	进程实例 ID	不是 NULL
oldvalue	在日志进行时，之前变量的值	
value	在日志进行时，变量的值	
variableid	进程定义中的变量 ID	
variableinstanceid	变量实例的 ID	

AuditTaskImpl 表包含有关用户任务的信息。

表 67.4. AuditTaskImpl 表字段

字段	描述	nullable
id	任务日志实体的主键和 ID	
activationTime	激活此任务的时间	
actualOwner	分配给此任务的实际所有者。仅当所有者声明任务时才会设置这个值。	
createdBy	创建此任务的用户	
createdOn	任务创建日期	
deploymentId	此任务所属的部署的 ID	
description	任务的描述	
dueDate	由于在此任务上设置的日期	
name	任务的名称	
parentId	父任务 ID	
priority	任务的优先级	
processId	此任务所属的进程定义 ID	
processInstanceId	与这个任务关联的进程实例 ID	
processSessionId	用于创建此任务的 KIE 会话 ID	
status	任务的当前状态	
taskId	任务标识符	
workItemId	在进程中分配给这个任务 ID 的工作项目的标识符	
lastModificationDate	进程实例状态最后在持久性数据库中记录的日期和时间	

BAMTaskSummary 表收集有关 BAM 引擎用于构建 chart 和仪表板的任务的信息。

表 67.5. BAMTaskSummary table 字段

字段	描述	nullable
pk	日志实体的主密钥和 ID	不是 NULL
createdDate	任务创建日期	
duration	任务创建的时间	
endDate	任务到达最终状态时的日期（完成、退出、失败、跳过）	
processinstanceid	进程实例 ID	
startDate	任务启动时的日期	
status	任务的当前状态	
taskId	任务标识符	
taskName	任务的名称	
userId	分配给任务的用户 ID	
optlock	作为 optimistic 锁定值的 version 字段	

TaskVariableImpl 表包含有关任务变量实例的信息。

表 67.6. TaskVariableImpl 表字段

字段	描述	nullable
id	日志实体的主密钥和 ID	不是 NULL
modificationDate	最近修改变量的日期	
name	任务的名称	
processid	进程实例正在执行的进程 ID	
processinstanceid	进程实例 ID	
taskId	任务标识符	
type	变量的类型：任务的输入或输出	

字段	描述	nullable
value	变量值	

TaskEvent 表包含有关任务实例中更改的信息。声明、启动和停止 等操作存储在此表中，以提供给定任务发生的事件的时间表视图。

表 67.7. TaskEvent 表字段

字段	描述	nullable
id	日志实体的主密钥和 ID	不是 NULL
logTime	保存此事件的日期	
message	事件日志信息	
processinstanceid	进程实例 ID	
taskId	任务标识符	
type	事件类型。类型与任务的生命周期阶段对应	
userId	分配给任务的用户 ID	
workItemId	为任务分配的工作项目的标识符	
optlock	作为 optimistic 锁定值的 version 字段	
correlationKey	进程实例的关联密钥	
processType	进程实例的类型（进程或问题单）	
currentOwner	任务的当前所有者	

67.2.2. 用于将进程事件日志存储在数据库中的配置

要使用默认数据模型在数据库中记录进程历史记录信息，您必须在您的会话中注册日志记录器。

在 KIE 会话中注册日志记录器

```
KieSession ksession = ...;
ksession.addProcessEventListener(AuditLoggerFactory.newInstance(Type.JPA, ksession,
null));

// invoke methods for your session here
```

要指定存储信息的数据库，您必须修改 persistence.xml 文件，使其包含审计日志类：ProcessInstanceLog、NodeInstanceLog 和 VariableInstanceLog。

修改后的 persistence.xml 文件，其中包括审计日志类

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<persistence
  version="2.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
  http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

    <class>org.jbpm.process.audit.ProcessInstanceLog</class>
    <class>org.jbpm.process.audit.NodeInstanceLog</class>
    <class>org.jbpm.process.audit.VariableInstanceLog</class>

    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.connection.release_mode" value="after_transaction"/>
      <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform"/>
```

```
    </properties>
  </persistence-unit>
</persistence>
```

67.2.3. 将进程事件日志发送到 JMS 队列的配置

当进程引擎以默认审计日志实施的形式将事件存储在数据库中时，数据库操作会异步完成（与进程实例的实际执行相同）。此操作需要时间，在高度加载的系统上可能会对数据库性能造成一些影响，特别是当历史记录日志和运行时数据都存储在同一数据库中时。

作为替代方案，您也可以使用流程引擎提供的基于 JMS 的日志记录器。您可以将此日志记录器配置为将进程日志条目作为消息提交到 JMS 队列，而不是直接将它们保留在数据库中。

您可以将 JMS 日志记录器配置为事务性，以避免在处理引擎事务回滚时数据不一致。

使用 JMS 审计日志记录器

```
ConnectionFactory factory = ...;
Queue queue = ...;
StatefulKnowledgeSession ksession = ...;
Map<String, Object> jmsProps = new HashMap<String, Object>();
jmsProps.put("jbpm.audit.jms.transacted", true);
jmsProps.put("jbpm.audit.jms.connection.factory", factory);
jmsProps.put("jbpm.audit.jms.queue", queue);
ksession.addProcessEventListener(AuditLoggerFactory.newInstance(Type.JMS, ksession,
jmsProps));

// invoke methods one your session here
```

这只是配置 JMS 审计日志记录器的可能方法之一。您可以使用 AuditLoggerFactory 类来设置额外的配置参数。

67.2.4. 审核变量

默认情况下，进程和任务变量的值以字符串表示形式存储在审计表中。要创建非字符串变量类型的字

符串，进程引擎调用 variable.toString（） 方法。如果将自定义类用于变量，您可以为类实施此方法。在很多情况下，这代表已经足够。

但是，有时日志中的字符串表示可能不足，特别是在需要按进程或任务变量进行有效查询时。例如，Person 对象用作变量的值，其结构可能如下：

Person 对象示例，用作进程或任务变量值

```
public class Person implements Serializable {

    private static final long serialVersionUID = -5172443495317321032L;
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

toString（） 方法提供人类可读的格式。但是，搜索可能不足。示例字符串值为 Person [name="john", age="34"]。通过大量搜索字符串来查找年龄 34 的人会使数据库查询效率低下。

要启用更有效的搜索，您可以使用 VariableIndexer 对象审核变量，这会提取审计日志中存储的变量的相关部分。

VariableIndexer 接口的定义

```
/**
 * Variable indexer that transforms a variable instance into another representation (usually string)
 * for use in log queries.
 *
 * @param <V> type of the object that will represent the indexed variable
 */
public interface VariableIndexer<V> {

    /**
     * Tests if this indexer can index a given variable
     *
     * NOTE: only one indexer can be used for a given variable
     *
     * @param variable variable to be indexed
     * @return true if the variable should be indexed with this indexer
     */
    boolean accept(Object variable);

    /**
     * Performs an index/transform operation on the variable. The result of this operation can be
     * either a single value or a list of values, to support complex type separation.
     * For example, when the variable is of the type Person that has name, address, and phone fields,
     * the indexer could build three entries out of it to represent individual fields:
     * person = person.name
     * address = person.address.street
     * phone = person.phone
     * this configuration allows advanced queries for finding relevant entries.
     * @param name name of the variable
     * @param variable actual variable value
     * @return
     */
    List<V> index(String name, Object variable);
}
```

默认索引器使用 toString（） 方法为单个变量生成单个审计条目。其他索引器可以从索引单个变量返回对象列表。

要启用对 Person 类型的有效查询，您可以构建将 Person 实例索引到单独的审计条目中的自定义索引程序，一个用于代表该年龄。

Person 类型的索引程序示例

```
public class PersonTaskVariablesIndexer implements TaskVariableIndexer {

    @Override
    public boolean accept(Object variable) {
        if (variable instanceof Person) {
            return true;
        }
        return false;
    }

    @Override
    public List<TaskVariable> index(String name, Object variable) {

        Person person = (Person) variable;
        List<TaskVariable> indexed = new ArrayList<TaskVariable>();

        TaskVariableImpl personNameVar = new TaskVariableImpl();
        personNameVar.setName("person.name");
        personNameVar.setValue(person.getName());

        indexed.add(personNameVar);

        TaskVariableImpl personAgeVar = new TaskVariableImpl();
        personAgeVar.setName("person.age");
        personAgeVar.setValue(person.getAge()+"");

        indexed.add(personAgeVar);

        return indexed;
    }

}
```

进程引擎可在 Person 类型时使用此索引器索引值，而所有其他变量则以默认 到String（） 方法进行索引。现在，要查询指向年龄 34 人的进程实例或任务，您可以使用以下查询：

- 变量名称： person.age

- 变量值： 34

因为没有使用 LIKE 类型查询，数据库服务器可以优化查询，并使其在大量数据上高效。

自定义索引器

进程引擎支持处理和任务变量的索引器。但是，它将不同的接口用于索引器，因为它们必须生成代表变量审计视图的不同类型的对象。

您必须实施以下接口来构建自定义索引器：

- 对于进程变量： org.kie.internal.process.ProcessVariableIndexer
- 对于任务变量： org.kie.internal.task.api.TaskVariableIndexer

您必须对以下任意接口实施两种方法：

- accept ：指示某个类型是否由此索引程序处理。进程引擎期望只有一个索引程序可以索引给定变量值，因此它会使用接受该类型的第一个索引器。
- 索引 ：索引值，生成对象或对象列表（通常是字符串）以包含在审计日志中。

在实施接口后，您必须将此实施打包为 JAR 文件，并在以下其中一个文件中列出实施：

- 对于进程变量，META-INF/services/org.kie.internal.process.Process.ProcessVariableIndexer 文件，列出进程变量索引器（每行单类名）的完全限定类名称。
- 对于任务变量，META-INF/services/org.kie.internal.task.api.TaskVariableIndexer 文件，它列出了任务变量索引器（每行单类名称）

ServiceLoader 机制使用这些文件发现索引器。当索引进程或任务变量时，进程引擎会检查注册的索引器以查找接受变量值的索引程序。如果没有其他索引程序接受值，则进程引擎将使用 toString（） 方

法应用默认的索引程序。

67.3. 进程引擎中的事务

流程引擎支持 Java 事务 API(JTA)事务。

进程引擎的当前版本不支持纯本地事务。

如果没有在应用程序内提供事务界限，流程引擎会在一个单独的事务中对进程引擎执行每个方法调用。

另外，您可以在应用程序代码中指定事务界限，例如，将多个命令合并到一个事务中。

67.3.1. 注册事务管理器

您必须在环境中注册事务管理器才能使用用户定义的事务。

以下示例代码注册事务管理器并使用 JTA 调用来指定事务边界。

注册事务管理器和使用事务

```
// Create the entity manager factory
EntityManagerFactory emf =
EntityManagerFactoryManager.get().getOrCreate("org.jbpm.persistence.jpa");
TransactionManager tm = TransactionManagerServices.getTransactionManager();

// Set up the runtime environment
RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
.newDefaultBuilder()
.addAsset(ResourceFactory.newClassPathResource("MyProcessDefinition.bpmn2"),
ResourceType.BPMN2)
    .addEnvironmentEntry(EnvironmentName.TRANSACTION_MANAGER, tm)
    .get();

// Get the KIE session
RuntimeManager manager =
RuntimeManagerFactory.Factory.get().newPerRequestRuntimeManager(environment);
RuntimeEngine runtime = manager.getRuntimeEngine(ProcessInstanceIdContext.get());
KieSession ksession = runtime.getKieSession();
```

```
// Start the transaction
UserTransaction ut = InitialContext.doLookup("java:comp/UserTransaction");
ut.begin();

// Perform multiple commands inside one transaction
ksession.insert( new Person( "John Doe" ) );
ksession.startProcess("MyProcess");

// Commit the transaction
ut.commit();
```

您必须在 root 类路径中提供 jndi.properties 文件，以创建 JNDI InitialContextonnectionFactoryy 对象，因为与事务相关的对象（如 UserTransaction、TransactionManager 和 TransactionSynchronizationRegistry ）在 JNDI 中注册。

如果您的项目包含 jbpm-test 模块，则默认已包含此文件。

否则，您必须使用以下内容创建 jndi.properties 文件：

jndi.properties 文件的内容

```
java.naming.factory.initial=org.jbpm.test.util.CloseSafeMemoryContextFactory
org.osjava.sj.root=target/test-classes/config
org.osjava.jndi.delimiter=/
org.osjava.sj.jndi.shared=true
```

此配置假设 simple-jndi:simple-jndi 工件存在于项目的类路径中。您还可以使用不同的 JNDI 实现。

默认情况下会使用 Narayana JTA 事务管理器。如果要使用不同的 JTA 事务管理器，您可以更改 persistence.xml 文件以使用所需的事务管理器。例如，如果您的应用程序在红帽 JBoss EAP 7 或更高版本上运行，您可以使用 JBoss 事务管理器。在这种情况下，更改 persistence.xml 文件中的事务管理器属性：

JBoss 事务管理器的 persistence.xml 文件中的事务管理器属性

```
<property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform" />
```

警告

使用 RuntimeManager 类的 Singleton 策略与 JTA 事务（UserTransaction 或 CMT）的 Singleton 创建了一个竞争条件。这个竞争条件可能会导致 IllegalStateException 异常，与 进程实例 XXX 类似的消息是断开连接。

要避免这种竞争条件，在用户应用程序代码中调用事务时，请明确同步 KieSession 实例。

```
synchronized (ksession) {
   try {
      tx.begin();

      // use ksession
      // application logic

      tx.commit();
   } catch (Exception e) {
      //...
   }
}
```

67.3.2. 配置容器管理事务

例如，如果您将进程引擎嵌入到由容器管理的事务(CMT)模式中执行的应用程序中，您必须完成额外的配置。如果应用程序在没有允许 CMT 应用从 JNDI 访问 UserTransaction 实例（例如，WebSphere Application）运行，则此配置尤为重要。

流程引擎中的默认事务管理器实施依赖于 UserTransaction 查询事务状态，然后使用该状态来确定是否启动事务。在阻止访问 UserTransaction 实例的环境中，此实施会失败。

要在 CMT 环境中启用正确执行，流程引擎提供一个专用的事务管理器实施：org.jbpm.persistence.jta.ContainerManagedTransactionManager。此事务管理器要求事务处于活动状态，并且始终在调用 getStatus（） 方法时返回 ACTIVE。开始、提交和 回滚 等操作是 no-op 方法，因为事务管理器无法在容器管理的事务模式中影响这些操作。

注意

在执行代码的过程中，必须将引擎引发的任何例外传播到容器，以确保容器在必要时回滚事务。

要配置此事务管理器，请完成此流程中的步骤。

流程

1. *在您的代码中，在创建或载入会话前将事务管理器和持久性上下文管理器插入到环境中：*

将事务管理器和持久性上下文管理器插入到环境中

```
Environment env = EnvironmentFactory.newEnvironment();
env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);
env.set(EnvironmentName.TRANSACTION_MANAGER, new
ContainerManagedTransactionManager());
env.set(EnvironmentName.PERSISTENCE_CONTEXT_MANAGER, new
JpaProcessPersistenceContextManager(env));
env.set(EnvironmentName.TASK_PERSISTENCE_CONTEXT_MANAGER, new
JPATaskPersistenceContextManager(env));
```

2. *在 persistence.xml 文件中，配置 JPA 提供程序。以下示例使用 hibernate 和 WebSphere Application Server。*

在 persistence.xml 文件中配置 JPA 供应商

```
<property name="hibernate.transaction.factory_class"
value="org.hibernate.transaction.CMTTransactionFactory"/>
<property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.WebSphereJtaPlatform"/>
```

3. 要分散 KIE 会话，请不要直接选择它。相反，执行 org.jbpm.persistence.jta.ContainerManagedTransactionDisposeCommand 命令。此命令可确保会话在完成当前事务时被处理。在以下示例中，ksession 是您要处理的 KieSession 对象。

使用 ContainerManagedTransactionDisposeCommand 命令执行 KIE 会话

```
ksession.execute(new ContainerManagedTransactionDisposeCommand());
```

直接调整会话会导致事务完成异常，因为进程引擎注册事务同步来清理会话状态。

67.3.3. 事务重试

当进程引擎提交事务时，有时提交操作会失败，因为同时提交另一个事务。在这种情况下，进程引擎必须重试事务。

如果多次重试失败，则事务会永久失败。

您可以使用 JVM 系统属性来控制重试进程。

表 67.8. 重试提交事务的系统属性

属性	值	默认	描述
org.kie.optlock.retries	整数	5	此属性描述了进程引擎永久重试事务的次数。
org.kie.optlock.delay	整数	50	第一次重试前的延迟时间（以毫秒为单位）。
org.kie.optlock.delayFactor	整数	4	每次后续重试增加延迟时间的倍数。使用默认值时，进程引擎在第一个重试前等待 50 毫秒，在第二个重试前等待 200 毫秒，即第三个重试前 800 毫秒，以此类推。

67.4. 在流程引擎中配置持久性

如果您在不配置任何持久性的情况下使用进程引擎，它不会将运行时数据保存到任何数据库；默认没有可用的内存数据库。如果出于性能原因或者想要自己管理持久性的原因，您可以使用这个模式。

要在流程引擎中使用 JPA 持久性，您必须对其进行配置。

配置通常需要添加必要的依赖项、配置数据源并创建配置了持久性的进程引擎类。

67.4.1. 在 persistence.xml 文件中配置

要使用 JPA 持久性，您必须在类路径中添加 persistence.xml 持久性配置，以配置 JPA 以使用 Hibernate 和 H2 数据库（或者您需要的任何其他数据库）。将此文件放到项目的 META-INF 目录中。

persistence.xml 文件示例

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence
      version="2.0"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
      http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
      xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <persistence-unit name="org.jbpm.persistence.jpa" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jbpm-ds</jta-data-source>
    <mapping-file>META-INF/JBPMorm.xml</mapping-file>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.jbpm.persistence.processinstance.ProcessInstanceInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationKeyInfo</class>
    <class>org.jbpm.persistence.correlation.CorrelationPropertyInfo</class>
    <class>org.jbpm.runtime.manager.impl.jpa.ContextMappingInfo</class>

    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.max_fetch_depth" value="3"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.connection.release_mode" value="after_transaction"/>
          <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform"/>
```

```
    </properties>
  </persistence-unit>
</persistence>
```

示例引用 jdbc/jbpm-ds 数据源。有关配置数据源的步骤，请参阅 第 67.4.2 节 “为流程引擎持久性配置数据源”。

67.4.2. 为流程引擎持久性配置数据源

要在流程引擎中配置 JPA 持久性，您必须提供一个代表数据库后端的数据源。

如果您在应用服务器（如 Red Hat JBoss EAP）中运行您的应用程序，您可以使用应用服务器设置数据源，例如在 部署 目录中添加数据源。有关创建数据源的说明，请参阅应用服务器的文档。

如果您将应用程序部署到 Red Hat JBoss EAP，您可以通过在 部署 目录中创建配置文件来创建数据源：

红帽 JBoss EAP 数据源配置文件示例

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/jbpm-ds</jndi-name>
    <connection-url>jdbc:h2:tcp://localhost/~/test</connection-url>
    <driver-class>org.h2.jdbcx.JdbcDataSource</driver-class>
    <user-name>sa</user-name>
    <password></password>
  </local-tx-datasource>
</datasources>
```

如果您的应用程序在普通的 Java 环境中运行，您可以使用 Red Hat Process Automation Manager 提供的 kie-test-util 模块中的 DataSourceFactory 类来使用 Narayana 和 Tomcat DBCP。请查看以下代码片段。这个示例使用 H2 内存中的数据库，与 Narayana 和 Tomcat DBCP 结合使用。

配置 H2 内存中数据库数据源的代码示例

```
Properties driverProperties = new Properties();
driverProperties.put("user", "sa");
driverProperties.put("password", "sa");
driverProperties.put("url", "jdbc:h2:mem:jbpm-db;MVCC=true");
driverProperties.put("driverClassName", "org.h2.Driver");
driverProperties.put("className", "org.h2.jdbcx.JdbcDataSource");
PoolingDataSourceWrapper pdsw = DataSourceFactory.setupPoolingDataSource("jdbc/jbpm-ds", driverProperties);
```

67.4.3. 持久性的依赖项

持久性需要特定的 JAR 构件依赖项。

jbpm-persistence-jpa.jar 文件始终是必需的。此文件包含可在需要时保存运行时状态的代码。

根据您使用的持久性解决方案和数据库，您可能需要额外的依赖项。默认配置组合包括以下组件：

- Hibernate 作为 JPA 持久供应商
- H2 内存中数据库
- 基于 JTA 的事务管理 Narayana
- 用于连接池功能的 Tomcat DBCP

此配置需要以下额外依赖项：

- jbpm-persistence-jpa (org.jbpm)

- *drools-persistence-jpa (org.drools)*
- *persistence-api (javax.persistence)*
- *hibernate-entitymanager (org.hibernate)*
- *hibernate-annotations (org.hibernate)*
- *hibernate-commons-annotations (org.hibernate)*
- *hibernate-core (org.hibernate)*
- *commons-collections (commons-collections)*
- *dom4j (org.dom4j)*
- *jta (javax.transaction)*
- *narayana-jta (org.jboss.narayana.jta)*
- *tomcat-dbcp (org.apache.tomcat)*
- *jboss-transaction-api_1.2_spec (org.jboss.spec.javax.transaction)*
- *javassist (javassist)*
- *slf4j-api (org.slf4j)*

- slf4j-jdk14 (org.slf4j)
- simple-jndi (simple-jndi)
- h2 (com.h2database)
- jBPM-test （仅限于测试的org.jbpm）不要将这个工件包括在 production 应用程序中

67.4.4. 创建具有持久性的 KIE 会话

如果您的代码直接创建 KIE 会话，您可以使用 JPAKnowledgeService 类来创建 KIE 会话。这个方法提供了对底层配置的完整访问权限。

流程

1. 使用 JPAKnowledgeService 类创建一个 KIE 会话会话，基于 KIE 基础、KIE 会话配置（如果需要）和一个环境。该环境必须包含对用于持久性的实体管理器工厂的引用。

创建具有持久性的 KIE 会话

```
// create the entity manager factory and register it in the environment
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory( "org.jbpm.persistence.jpa" );
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY, emf );

// create a new KIE session that uses JPA to store the runtime state
StatefulKnowledgeSession ksession =
JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );
int sessionId = ksession.getId();

// invoke methods on your method here
ksession.startProcess( "MyProcess" );
ksession.dispose();
```

2. 要根据特定会话 ID 从数据库重新创建会话，请使用

JPAKnowledgeService.loadStatefulKnowledgeSession（） 方法：

从持久性数据库重新创建 KIE 会话

```
// re-create the session from database using the sessionId
ksession = JPAKnowledgeService.loadStatefulKnowledgeSession(sessionId, kbase,
null, env );
```

67.4.5. 运行时管理器中的持久性

如果您的代码使用 RuntimeManager 类，请使用 RuntimeEnvironmentBuilder 类来配置用于持久性的环境。默认情况下，运行时管理器会搜索 org.jbpm.persistence.jpa persistence 单元。

以下示例创建了一个带有空上下文的 KieSession。

使用运行时管理器创建带有空上下文的 KIE 会话

```
RuntimeEnvironmentBuilder builder = RuntimeEnvironmentBuilder.Factory.get()
    .newDefaultBuilder()
    .knowledgeBase(kbase);
RuntimeManager manager = RuntimeManagerFactory.Factory.get()
    .newSingletonRuntimeManager(builder.get(), "com.sample:example:1.0");
RuntimeEngine engine = manager.getRuntimeEngine(EmptyContext.get());
KieSession ksession = engine.getKieSession();
```

前面的示例需要 KIE 基础作为 kbase 参数。您可以使用类路径上的 kmodule.xml KJAR 描述符来构建 KIE 基础。

从 kmodule.xml KJAR 描述符构建 KIE 基础

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();
KieBase kbase = kContainer.getKieBase("kbase");
```

kmodule.xml 描述符文件可以包含用于扫描的资源软件包的属性，以查找和部署进程引擎工作流。

kmodule.xml 描述符文件示例

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase" packages="com.sample"/>
</kmodule>
```

要控制持久性，您可以使用 RuntimeEnvironmentBuilder::entityManagerFactory 方法。

在运行时管理器中控制持久性配置

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("org.jbpm.persistence.jpa");

RuntimeEnvironment runtimeEnv = RuntimeEnvironmentBuilder.Factory
        .get()
        .newDefaultBuilder()
        .entityManagerFactory(emf)
        .knowledgeBase(kbase)
        .get();

StatefulKnowledgeSession ksession = (StatefulKnowledgeSession)
RuntimeManagerFactory.Factory.get()
        .newSingletonRuntimeManager(runtimeEnv)
        .getRuntimeEngine(EmptyContext.get())
        .getKieSession();
```

在这个示例中创建 ksession KIE 会话后，您可以在 ksession 中调用方法，例如

StartProcess（）。进程引擎会在配置的数据源中保留运行时状态。

您可以使用进程实例 ID 从持久性存储恢复进程实例。runtime Manager 会自动重新创建所需的会话。

使用进程实例 ID 从持久性数据库重新创建 KIE 会话

```
RuntimeEngine runtime =
manager.getRuntimeEngine(ProcessInstanceIdContext.get(processInstanceId));

KieSession session = runtime.getKieSession();
```

67.5. 在 RED HAT PROCESS AUTOMATION MANAGER 中以独立数据库模式持久保留进程变量

当您创建要在您定义的进程中使用的进程变量时，红帽流程自动化管理器将这些进程变量存储在默认数据库架构中。您可以在单独的数据库架构中保留进程变量，以便在维护和实施流程数据方面具有更大的灵活性。

例如，在您的独立数据库架构中保留进程变量可以帮助您执行以下任务：

- 以人类可读格式维护进程变量
- 将变量提供给 Red Hat Process Automation Manager 之外的服务
- 清除 Red Hat Process Automation Manager 中默认数据库表的日志，而不丢失进程变量数据

注意

此流程只适用于处理变量。此过程不适用于问题单变量。

先决条件

- 您已在 Red Hat Process Automation Manager 中定义了您要在其中实施变量的进程。
- 如果要将变量保存在 Red Hat Process Automation Manager 之外的数据库架构中，您可以创建一个数据源和您要使用的独立数据库 schema。有关创建数据源的详情，请参考 配置 Business Central 设置和属性。

流程

1. 在用作进程变量的数据对象文件中，添加以下元素来配置变量持久性：

为变量持久性配置 Person.java 对象示例

```
@javax.persistence.Entity (1)
@javax.persistence.Table(name = "Person") (2)
public class Person extends org.drools.persistence.jpa.marshaller.VariableEntity (3)
implements java.io.Serializable { (4)

 static final long serialVersionUID = 1L;

 @javax.persistence.GeneratedValue(strategy =
javax.persistence.GenerationType.AUTO, generator = "PERSON_ID_GENERATOR")
 @javax.persistence.Id (5)
 @javax.persistence.SequenceGenerator(name = "PERSON_ID_GENERATOR",
sequenceName = "PERSON_ID_SEQ")
 private java.lang.Long id;

 private java.lang.String name;

 private java.lang.Integer age;

 public Person() {
 }

 public java.lang.Long getId() {
  return this.id;
 }

 public void setId(java.lang.Long id) {
  this.id = id;
 }

 public java.lang.String getName() {
  return this.name;
 }

 public void setName(java.lang.String name) {
```

```
 this.name = name;
}

public java.lang.Integer getAge() {
 return this.age;
}

public void setAge(java.lang.Integer age) {
 this.age = age;
}

public Person(java.lang.Long id, java.lang.String name,
  java.lang.Integer age) {
 this.id = id;
 this.name = name;
 this.age = age;
}

}
```

1

将数据对象配置为持久性实体。

2

定义用于 data 对象的数据库表名称。

3

创建单独的 MappedVariable 映射表，用于维护此数据对象和相关进程实例之间的关系。如果您不需要此关系，则不需要扩展 VariableEntity 类。如果没有此扩展，数据对象仍会保留，但没有包含额外的数据。

4

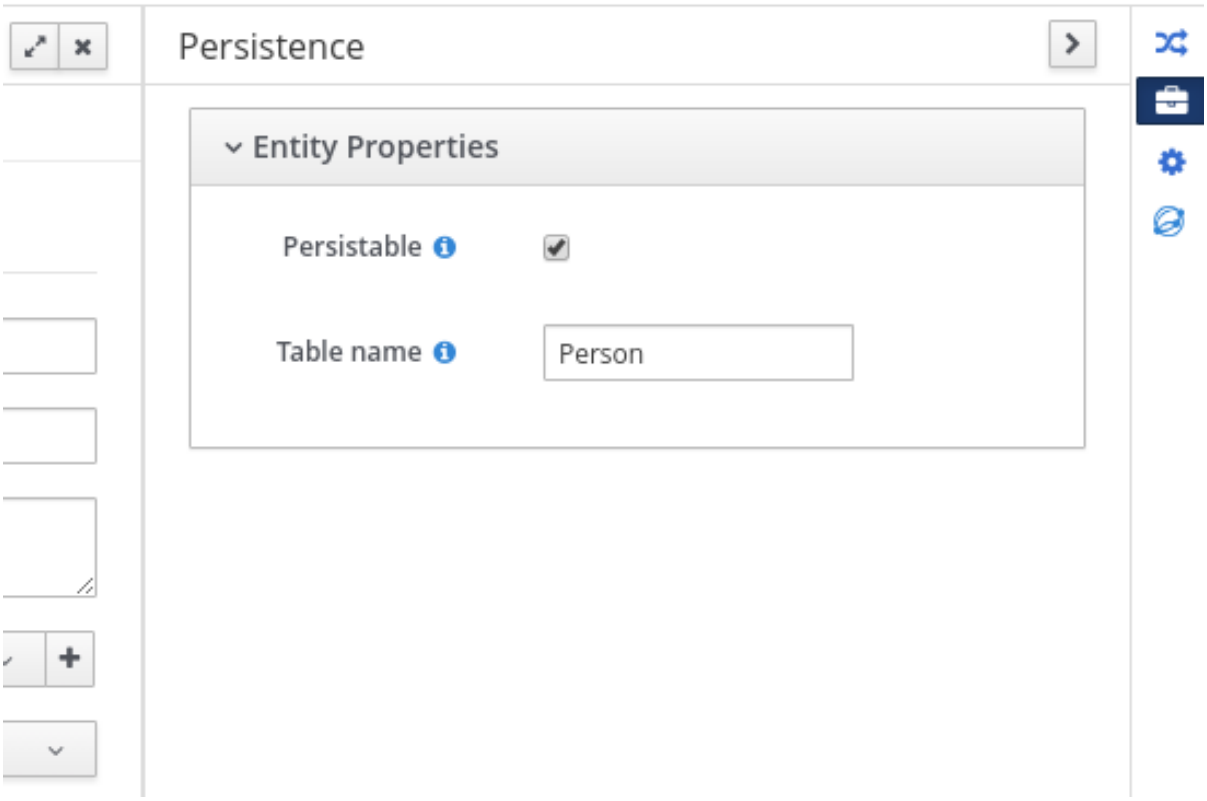
将数据对象配置为 serializable 对象。

5

为对象设置持久性 ID。

要使用 Business Central 使数据对象持久，请导航到项目中的 data 对象文件，单击窗口右上角的 Persistence 图标，并配置持久性行为：

图 67.1. Business Central 中的持久性配置



2. *在项目的 pom.xml 文件中，添加以下用于持久性支持的依赖关系。这个依赖关系包含您在 data 对象中配置的 VariableEntity 类。*

持久性的项目依赖性

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-persistence-jpa</artifactId>
  <version>${rhpam.version}</version>
  <scope>provided</scope>
</dependency>
```

3. *在项目的 ~/META-INF/kie-deployment-descriptor.xml 文件中，配置 JPA marshalling 策略以及要与 marshaller 搭配使用的持久性单元。对于定义为实体的对象，需要 JPA marshalling 策略和持久性单元。*

JPA marshaller 和 persistence 单元在 kie-deployment-descriptor.xml 文件中配置

```
<marshalling-strategy>
  <resolver>mvel</resolver>
  <identifier>new
org.drools.persistence.jpa.marshaller.JPAPlaceholderResolverStrategy("myPersistenc
eUnit", classLoader)</identifier>
  <parameters/>
</marshalling-strategy>
```

4. 在项目的 ~/META-INF 目录中，创建一个 persistence.xml 文件，用于指定您要保留的进程变量：

带有数据源配置的 persistence.xml 文件示例

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd">
   <persistence-unit name="myPersistenceUnit" transaction-type="JTA">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source> (1)
      <class>org.space.example.Person</class>
      <exclude-unlisted-classes>true</exclude-unlisted-classes>
      <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
         <property name="hibernate.max_fetch_depth" value="3"/>
         <property name="hibernate.hbm2ddl.auto" value="update"/>
         <property name="hibernate.show_sql" value="true"/>
         <property name="hibernate.id.new_generator_mappings" value="false"/>
         <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform"/>
      </properties>
   </persistence-unit>
</persistence>
```

(1) 设置进程变量保留的数据源

要使用 Business Central 配置 marshalling 策略、持久性单元和数据源，请导航至项目 Settings → Deployments → Marshalling Strategy 和 Project Settings → Persistence ：

图 67.2. JPA marshaller 配置 Business Central

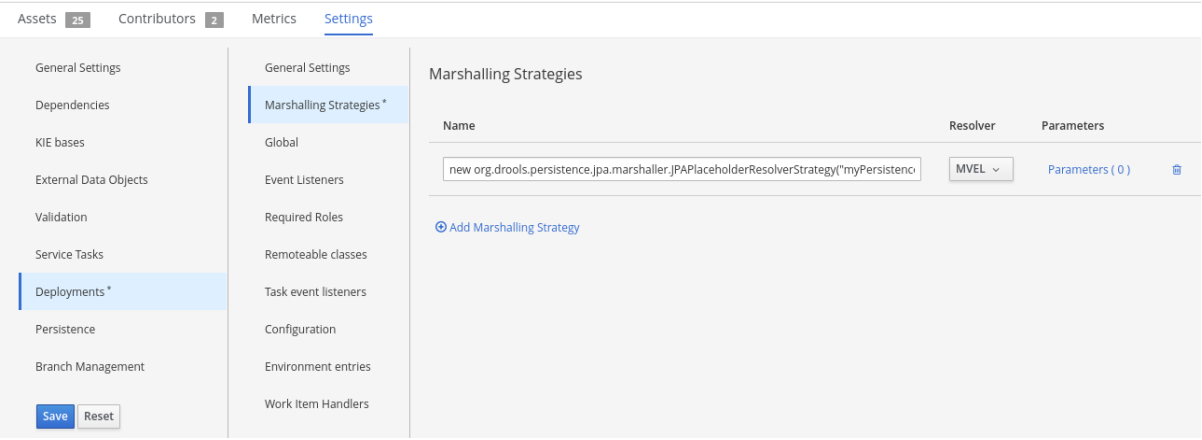
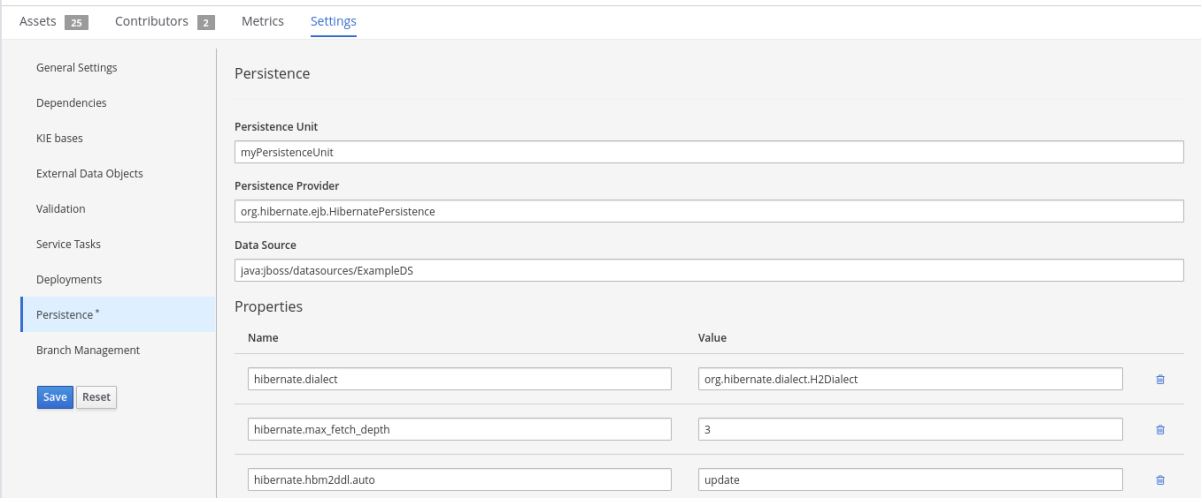


图 67.3. Business Central 中的持久性单元和数据源配置



第 68 章 与 JAVA 框架集成

您可以将流程引擎与多个行业标准 Java 框架（如 Apache Maven、CDI、Spring 和 EJB）集成。

68.1. 与 APACHE MAVEN 集成

进程引擎将 Maven 用于两个主要目的：

- 要创建 KJAR 工件，它们是进程引擎可安装到用于执行的运行时环境的部署单元
- 管理用于构建嵌入进程引擎的应用程序的依赖关系

68.1.1. Maven 工件作为部署单元

进程引擎提供了一种机制，可从 Apache Maven 工件部署进程。这些工件采用 JAR 文件格式，称为 KJAR 文件，或者通知 KJARs。KJAR 文件包括定义 KIE 基础和 KIE 会话的描述符。它还包含业务资产，包括流程定义，流程引擎可加载到 KIE 基础中。

KJAR 文件描述符由名为 kie-deployment-descriptor.xml 的 XML 文件表示。描述符可以为空，在这种情况下，会应用默认配置。它还可以为 KIE 基本和 KIE 会话提供自定义配置。

空 kie-deployment-descriptor.xml 描述符。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<deployment-descriptor xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <persistence-unit>org.jbpm.domain</persistence-unit>
  <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
  <audit-mode>JPA</audit-mode>
  <persistence-mode>JPA</persistence-mode>
  <runtime-strategy>SINGLETON</runtime-strategy>
  <marshalling-strategies/>
  <event-listeners/>
  <task-event-listeners/>
  <globals/>
  <work-item-handlers />
  <environment-entries/>
  <configurations/>
```

```
    <required-roles/>
    <remoteable-classes/>
</deployment-descriptor>
```

带有一个空的 kie-deployment-descriptor.xml 描述符，会应用以下默认配置：

- 创建一个默认的 KIE 基础，其特征如下：
  - 它包含 KJAR 文件中所有软件包的所有资产
  - 其事件处理模式被设置为 云
  - 它的 equivalenthaviour 设置为 identity
  - 它声明性声明性声明已被禁用
  - 对于 CDI 应用，其范围设置为 ApplicationScope
- 创建单一默认的无状态 KIE 会话，特征如下：
  - 它绑定到单个 KIE 基础
  - 其时钟类型设置为 实时
  - 对于 CDI 应用，其范围设置为 ApplicationScope
- 创建单一默认有状态 KIE 会话，特征如下：

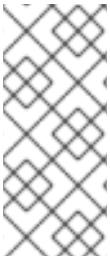
- 它绑定到单个 KIE 基础
- 其时钟类型设置为 实时
- 对于 CDI 应用，其范围设置为 ApplicationScope

如果您不想使用默认值，可以使用 kie-deployment-descriptor.xml 文件更改所有配置设置。您可以在 XSD schema 中找到此文件的所有元素的完整规格。

以下示例显示了配置运行时引擎的自定义 kie-deployment-descriptor.xml 文件。这个示例配置最常见的选项，并包含一个工作项目处理程序。您还可以使用 kie-deployment-descriptor.xml 文件来配置其他选项。

自定义 kie-deployment-descriptor.xml 文件示例

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<deployment-descriptor xsi:schemaLocation="http://www.jboss.org/jbpm deployment-descriptor.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <persistence-unit>org.jbpm.domain</persistence-unit>
  <audit-persistence-unit>org.jbpm.domain</audit-persistence-unit>
  <audit-mode>JPA</audit-mode>
  <persistence-mode>JPA</persistence-mode>
  <runtime-strategy>SINGLETON</runtime-strategy>
  <marshalling-strategies/>
  <event-listeners/>
  <task-event-listeners/>
  <globals/>
  <work-item-handlers>
    <work-item-handler>
      <resolver>mvel</resolver>
      <identifier>new org.jbpm.process.workitem.bpmn2.ServiceTaskHandler(ksession,
classLoader)</identifier>
      <parameters/>
      <name>Service Task</name>
    </work-item-handler>
  </work-item-handlers>
  <environment-entries/>
  <configurations/>
  <required-roles/>
  <remoteable-classes/>
</deployment-descriptor>
```



注意

如果使用 RuntimeManager 类，则此类创建 KieSession 实例，而不是 KieContainer 类。但是，kie-deployment-descriptor.xml 模型始终用作构造过程的基础。KieContainer 类始终创建 KieBase 实例。

您可以使用 GAV（组、工件、版本）值引用 KJAR 工件，如任何其他 Maven 工件。当从 KJAR 文件部署单元时，进程引擎使用 GAV 值作为 KIE API 中的发行 ID。您可以使用 GAV 值将 KJAR 工件部署到运行时环境中，例如：KIE 服务器。

68.1.2. 使用 Maven 管理依赖项

构建嵌入进程引擎的项目时，请使用 Apache Maven 配置进程引擎所需的所有依赖项。

进程引擎提供一组 BOMs（资料的 Bills）来简化声明工件依赖项。

使用项目的顶级 pom.xml 文件来定义用于嵌入进程引擎的依赖关系管理，如下例所示。示例包括主要的运行时依赖项，这些依赖项适用于应用程序是否在 servlet 容器中部署或作为独立应用。

这个示例还包括应用程序使用进程引擎的应用程序的版本属性。根据需要调整组件和版本列表。您可以查看产品团队 在 Github 存储库中的父 pom.xml 文件中 测试的第三方依赖项版本。

用于嵌入进程引擎的 Maven 依赖项管理设置

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <version.org.drools>
  </version.org.drools>
  <version.org.jbpm>7.67.0.Final-redhat-00024</version.org.jbpm>
  <hibernate.version>5.3.17.Final</hibernate.version>
  <hibernate.core.version>5.3.17.Final</hibernate.core.version>
  <slf4j.version>1.7.26</slf4j.version>
  <jboss.javaee.version>1.0.0.Final</jboss.javaee.version>
  <logback.version>1.2.9</logback.version>
  <h2.version>1.3.173</h2.version>
  <narayana.version>5.9.0.Final</narayana.version>
```

```
    <jta.version>1.0.1.Final</jta.version>
    <junit.version>4.13.1</junit.version>
  </properties>
  <dependencyManagement>
    <dependencies>
      <!-- define Drools BOM -->
      <dependency>
        <groupId>org.drools</groupId>
        <artifactId>drools-bom</artifactId>
        <type>pom</type>
        <version>${version.org.drools}</version>
        <scope>import</scope>
      </dependency>
      <!-- define jBPM BOM -->
      <dependency>
        <groupId>org.jbpm</groupId>
        <artifactId>jbpm-bom</artifactId>
        <type>pom</type>
        <version>${version.org.jbpm}</version>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

在使用进程引擎 Java API(KIE API)的模块中，声明所需的进程引擎依赖项和模块所需的其他组件，如下例所示：

使用 KIE API 的模块的依赖项

```
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-flow</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-flow-builder</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-bpmn2</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jbpm</groupId>
      <artifactId>jbpm-persistence-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jbpm</groupId>
```

```
  <artifactId>jbpm-human-task-core</artifactId>
</dependency>
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-runtime-manager</artifactId>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>${slf4j.version}</version>
</dependency>
```

如果您的应用程序使用持久性和事务，您必须添加实施 JTA 和 JPA 框架的工件。在实际部署前测试工作流组件需要其他依赖项。

以下示例定义了包括 JPA 的 Hibernate、用于持久性的 H2 数据库、JTA 的 Narayana 以及测试所需的组件的依赖项。这个示例使用 test 范围。根据您的应用程序需要调整这个示例。对于生产环境，请删除 测试 范围。

进程引擎的测试模块依赖项示例

```
<!-- test dependencies -->
<dependency>
  <groupId>org.jbpm</groupId>
  <artifactId>jbpm-shared-services</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>${hibernate.version}</version>
  <scope>test</scope>
</dependency>
```

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>${hibernate.core.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>${h2.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>jboss-transaction-api_1.2_spec</groupId>
  <artifactId>org.jboss.spec.javax.transaction</artifactId>
  <version>${jta.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.narayana.jta</groupId>
  <artifactId>narayana-jta</artifactId>
  <version>${narayana.version}</version>
  <scope>test</scope>
</dependency>
```

使用这个配置，您可以在应用程序中嵌入进程引擎，并使用 KIE API 与进程、规则和事件交互。

Maven 存储库

要使用 Red Hat 产品版本的 Maven 依赖项，您必须在顶层 pom.xml 文件中配置 Red Hat JBoss Enterprise Maven 存储库。有关此软件仓库的详情，请查看 JBoss Enterprise Maven Repository。

或者，从红帽客户门户网站的软件下载页面下载 rhpam-7.13.5-maven-repository.zip 产品交付的文件，并将此文件的内容作为本地 Maven 存储库提供。https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=rhpam&productChanged=yes

68.2. 与 CDI 集成

流程引擎支持自动与 CDI 集成。您可以在 CDI 框架中使用大多数 API，而无需任何修改。

进程引擎还提供一些专门用于 CDI 容器的专用模块。最重要的模块是 jbpm-services-cdi，它为处理引擎服务提供 CDI 封装器。您可以使用这些打包程序在 CDI 应用程序中集成进程引擎。模块提供以下一

组服务：

- DeploymentService
- ProcessService
- UserTaskService
- RuntimeDataService
- DefinitionService

这些服务可用于在任何其他 CDI Bean 中注入。

68.2.1. CDI 部署服务

DeploymentService 服务在运行时环境中部署并取消部署部署单元。当您使用这个服务部署单元时，部署单元就可以执行，并创建一个 RuntimeManager 实例。您还可以使用 DeploymentService 检索以下对象：

- 给定部署 ID 的 RuntimeManager 实例
- 代表给定部署 ID 的完整部署单元的 DeployedUnit 实例
- 部署服务已知的所有部署单元列表

默认情况下，部署服务不会将部署单元的信息保存到任何持久性存储。在 CDI 框架中，使用该服务的组件可以保存和恢复部署单元信息，例如使用数据库、文件系统或存储库。

部署服务会在部署和取消部署中触发 CDI 事件。使用服务的组件可以处理这些事件来存储部署，并在取消部署时将其从存储中移除。

- 部署单元时会触发带有 @Deploy qualifier 的 DeploymentEvent
- 带有 @Undeploy qualifier 的 DeploymentEvent 在单元的 undeployment 上触发

您可以使用 CDI 观察程序机制来获取这些事件的通知。

以下示例收到单元部署中的通知，并可以保存部署：

部署事件处理示例

```
public void saveDeployment(@Observes @Deploy DeploymentEvent event) {
    // Store deployed unit information
    DeployedUnit deployedUnit = event.getDeployedUnit();
}
```

以下示例在部署单元时收到通知，并可从存储中删除部署：

处理非部署事件的示例

```
public void removeDeployment(@Observes @Undeploy DeploymentEvent event) {
    // Remove deployment with the ID event.getDeploymentId()
}
```

DeploymentService 服务有几个实现，因此您必须使用限定符来指示 CDI 容器注入特定的实施。每一实施部署都必须存在 DeploymentUnit 的 匹配。

流程引擎提供 KmoduleDeploymentService 实施。这种实现旨在与 KmoduleDeploymentUnits 合作，它们是 KJAR 文件中所含的小描述符。这个实现是大多数用例中典型的解决方案。此实施的限定符是

@Kjar。

68.2.2. CDI 的表单供应商服务

FormProviderService 服务提供对表单的访问，通常显示在用户界面中用于进程表单和用户任务表单的用户界面。

该服务依赖于隔离形式的供应商的概念，可提供不同功能并由不同的技术提供支持。FormProvider 接口描述了表单供应商的合同。

FormProvider 接口的定义

```
public interface FormProvider {

  int getPriority();

  String render(String name, ProcessDesc process, Map<String, Object> renderContext);

  String render(String name, Task task, ProcessDesc process, Map<String, Object>
renderContext);
}
```

FormProvider 接口的实现必须定义优先级值。当 FormProviderService 服务需要呈现表单时，它会按优先级顺序调用可用提供程序。

优先级越低，提供商获得的优先级越高。例如，在优先级为 10 的供应商之前，评估优先级为 5 的供应商。对于每个所需表单，服务按照其优先级顺序迭代可用的提供程序，直至其中之一来提供内容。在最糟糕的情况中，返回一个简单的基于文本的表单。

进程引擎提供 FormProvider 的以下实现：

- 提供 Form Modeller 工具中创建的表单的供应商，优先级为 2
- 基于 FreeMarker 的实施，它支持流程和任务表单，优先级为 3

- 默认表单供应商返回简单的基于文本的表单，如果不存在其他供应商提供任何内容，其使用方法为 1000

68.2.3. CDI 运行时数据服务

RuntimeDataService 服务提供对在运行时可用的数据的访问，包括以下数据：

- 要执行的可用进程，带有各种过滤器
- 活跃的进程实例，带有不同的过滤器
- 进程实例历史记录
- 进程实例变量
- 进程实例的活跃和完成节点

RuntimeDataService 的默认实现会观察部署事件，并索引所有部署的进程以将其公开给调用组件。

68.2.4. CDI 定义服务

DefinitionService 服务提供对作为 BPMN2 XML 定义一部分存储的进程详细信息的访问。

注意

在使用提供信息的任何方法前，调用 buildProcessDefinition（） 方法，以使用从 BPMN2 内容检索的进程信息填充存储库。

BPMN2DataService 实现提供对以下数据的访问：

- 给定进程定义过程的整体描述

- 在进程定义中找到的所有用户任务的集合
- 有关用户任务节点定义的输入的信息
- 有关为用户任务节点定义的输出的信息
- 在给定进程定义中定义的可重用进程的 ID（call 活动）
- 有关在给定进程定义中定义的进程变量的信息
- 有关包含在进程定义中的所有组织实体（用户和组）的信息。根据具体进程定义，用户和组返回的值可包含以下信息：
  - 实际用户或组名称
  - 用于在运行时获取实际用户或组名称的进程变量，例如： #{manager}

68.2.5. CDI 集成配置

要在 CDI 框架中使用 jbpm-services-cdi 模块，您必须提供一些 bean 来满足所含服务实施的依赖项。

根据使用场景，需要几个 Bean：

- 实体管理器和实体管理器工厂
- 用于人工任务的用户组回调
- 身份提供程序将经过身份验证的用户信息传递给服务

在 JEE 环境中运行（如红帽 JBoss EAP）时，以下制作者 bean 满足 jbpm-services-cdi 模块的所有要求。

制作者 bean 满足 JEE 环境中的 jbpm-services-cdi 模块的所有要求

```
public class EnvironmentProducer {

    @PersistenceUnit(unitName = "org.jbpm.domain")
    private EntityManagerFactory emf;

    @Inject
    @Selectable
    private UserGroupInfoProducer userGroupInfoProducer;

    @Inject
    @Kjar
    private DeploymentService deploymentService;

    @Produces
    public EntityManagerFactory getEntityManagerFactory() {
        return this.emf;
    }

    @Produces
    public org.kie.api.task.UserGroupCallback produceSelectedUserGroupCalback() {
        return userGroupInfoProducer.produceCallback();
    }

    @Produces
    public UserInfo produceUserInfo() {
        return userGroupInfoProducer.produceUserInfo();
    }

    @Produces
    @Named("Logs")
    public TaskLifeCycleEventListener produceTaskAuditListener() {
        return new JPATaskLifeCycleEventListener(true);
    }

    @Produces
    public DeploymentService getDeploymentService() {
        return this.deploymentService;
    }

    @Produces
    public IdentityProvider produceIdentityProvider {
        return new IdentityProvider() {
            // implement IdentityProvider
        };
    }
}
```

应用的 Bean.xml 文件必须为用户组 info 回调启用正确的替代选择。这种替代选择基于 @Selectable qualifier。

在 Bean.xml 文件'中，为用户组信息回调的定义

```
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee https://docs.jboss.org/cdi/beans_1_0.xsd">

  <alternatives>
    <class>org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer</class>
  </alternatives>

</beans>
```

注意

org.jbpm.kie.services.cdi.producer.JAASUserGroupInfoProducer 是一个示例值。这个值通常适合于红帽 JBoss EAP，因为它可重复利用应用服务器上的安全设置，无论服务器使用哪种安全方法，如 LDAP 或数据库。

另外，您还可以提供一些其他制作者来提供 WorkItemHandlers 和 Process、Agenda、WorkingMemory 事件监听程序。您可以通过实现以下接口来提供这些组件：

处理引擎与 CDI 集成的工作项目处理程序制作界面

```
/**
 * Enables providing custom implementations to deliver WorkItem name and WorkItemHandler instance pairs
 * for the runtime.
 * <br/>
 * This interface is invoked by the RegisterableItemsFactory implementation (in particular InjectableRegisterableItemsFactory
 * in the CDI framework) for every KieSession. Always return new instances of objects to avoid unexpected
```

```
 * results.
 *
 */
public interface WorkItemHandlerProducer {

    /**
     * Returns map of work items(key = work item name, value = work item handler instance)
     * to be registered on KieSession
     * <br/>
     * The following parameters might be given:
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     *
     * @param identifier - identifier of the owner - usually the RuntimeManager. This parameter
allows the producer to filter out
     * and provide valid instances for a given owner
     * @param params - the owner might provide some parameters, usually KieSession,
TaskService, RuntimeManager instances
     * @return map of work item handler instances (always return new instances when this
method is invoked)
     */
    Map<String, WorkItemHandler> getWorkItemHandlers(String identifier, Map<String, Object>
params);
}
```

用于处理引擎与 CDI 集成的事件监听程序制作者接口

```
/**
 * Enables defining custom producers for known EventListeners. There might be several
 * implementations that might provide a different listener instance based on the context in
which they are executed.
 * <br/>
 * This interface is invoked by the RegisterableItemsFactory implementation (in particular,
InjectableRegisterableItemsFactory
 * in the CDI framework) for every KieSession. Always return new instances of objects to avoid
unexpected results.
 *
 * @param <T> type of the event listener - ProcessEventListener, AgendaEventListener,
WorkingMemoryEventListener
 */
public interface EventListenerProducer<T> {

    /**
     * Returns list of instances for given (T) type of listeners
     * <br/>
     * Parameters that might be given are:
```

```
     * <ul>
     *  <li>ksession</li>
     *  <li>taskService</li>
     *  <li>runtimeManager</li>
     * </ul>
     * @param identifier - identifier of the owner - usually RuntimeManager. This parameter
allows the producer to filter out
     * and provide valid instances for given owner
     * @param params - the owner might provide some parameters, usually KieSession,
TaskService, RuntimeManager instances
     * @return list of listener instances (always return new instances when this method is
invoked)
     */
    List<T> getEventListeners(String identifier, Map<String, Object>  params);
}
```

实施这两个接口的 Bean 会在运行时收集并在 RuntimeManager 类构建 KieSession 实例时调用。

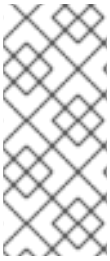
68.2.5.1. 作为 CDI Bean 的运行时管理器

您可以将 RuntimeManager 类作为 CDI Bean 注入应用程序中的任何其他 CDI Bean。必须正确生成 RuntimeEnvironment 类，以启用 RuntimeManager 实例的正确初始化。

以下 CDI 限定符引用现有的运行时管理器策略：

- @Singleton
- @PerRequest
- @PerProcessInstance

有关运行时管理器的详情，请参考 第 66.2 节 “运行时管理器”。



注意

虽然您可以直接注入 RuntimeManager 类，但对于大多数框架用例（如 CDI、EJB 或 Spring）的解决方案正在使用服务。流程引擎服务使用运行时管理器实施许多最佳实践。

要使用运行时管理器，您必须将 RuntimeEnvironment 类添加到 第 68.2.5 节 “CDI 集成配置” 部分中定义的 producer 中。

提供 RuntimeEnvironment 类的 producer bean

```
public class EnvironmentProducer {

  //Add the same producers as for services

  @Produces
  @Singleton
  @PerRequest
  @PerProcessInstance
  public RuntimeEnvironment produceEnvironment(EntityManagerFactory emf) {

    RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
        .newDefaultBuilder()
        .entityManagerFactory(emf)
        .userGroupCallback(getUserGroupCallback())

.registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(beanManager, null))
        .addAsset(ResourceFactory.newClassPathResource("BPMN2-ScriptTask.bpmn2"),
ResourceType.BPMN2)
        .addAsset(ResourceFactory.newClassPathResource("BPMN2-UserTask.bpmn2"),
ResourceType.BPMN2)
        .get();
    return environment;
  }
}
```

在本例中，单一制作者方法可以通过在方法级别上指定所有运行时管理器策略，为所有运行时管理器策略提供 RuntimeEnvironment 类。

当完整的制作者可用时，RuntimeManager 类可以注入到应用程序的 CDI bean 中：

注入 RuntimeManager 类

```
public class ProcessEngine {

    @Inject
    @Singleton
    private RuntimeManager singletonManager;

    public void startProcess() {

        RuntimeEngine runtime = singletonManager.getRuntimeEngine(EmptyContext.get());
        KieSession ksession = runtime.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("UserTask");

        singletonManager.disposeRuntimeEngine(runtime);
    }
}
```

如果您注入 RuntimeManager 类，则应用程序中只能有一个 RuntimeManager 实例。在典型的情形中，使用 DeploymentService 服务根据需要创建 RuntimeManager 实例。

作为 DeploymentService 的替代方案，您可以注入 RuntimeManageronnectionFactory y 类，然后应用程序就可以使用它来创建 RuntimeManager 实例。在这种情况下，仍需要 EnvironmentProducer 定义。以下示例显示了一个简单的 ProcessEngine bean。

ProcessEngine bean 示例

```
public class ProcessEngine {

    @Inject
    private RuntimeManagerFactory managerFactory;

    @Inject
    private EntityManagerFactory emf;

    @Inject
    private BeanManager beanManager;

    public void startProcess() {
        RuntimeEnvironment environment = RuntimeEnvironmentBuilder.Factory.get()
                .newDefaultBuilder()
```

```
            .entityManagerFactory(emf)
            .addAsset(ResourceFactory.newClassPathResource("BPMN2-ScriptTask.bpmn2"),
ResourceType.BPMN2)
            .addAsset(ResourceFactory.newClassPathResource("BPMN2-UserTask.bpmn2"),
ResourceType.BPMN2)

.registerableItemsFactory(InjectableRegisterableItemsFactory.getFactory(beanManager, null))
            .get();

        RuntimeManager manager =
managerFactory.newSingletonRuntimeManager(environment);
        RuntimeEngine runtime = manager.getRuntimeEngine(EmptyContext.get());
        KieSession ksession = runtime.getKieSession();

        ProcessInstance processInstance = ksession.startProcess("UserTask");

        manager.disposeRuntimeEngine(runtime);
        manager.close();
    }

}
```

68.3. 与 SPRING 集成

虽然可以通过几种方式将流程引擎与 Spring 框架一起使用，但最常使用两种方法

- 直接使用 Runtime Manager API
- 使用进程引擎服务

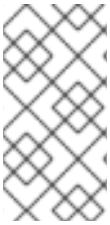
这两种方法都经过测试并有效。

如果您的应用程序只需要使用一个运行时管理器，请使用直接运行时管理器 API，因为这是在 Spring 应用程序中使用流程引擎的最简单方法。

如果您的应用程序需要使用运行时管理器的多个实例，请使用进程引擎服务，通过提供动态运行时环境来封装最佳实践。

68.3.1. 在 Spring 中直接使用运行时管理器 API

运行时管理器在同步过程中管理进程引擎和任务服务。有关运行时管理器的详情，请参考 第 66.2 节 “运行时管理器”。



注意

Red Hat Process Automation Manager 不支持使用 Spring MVC 应用程序中的 EJB 计时器。

要在 Spring 框架中设置运行时管理器，请使用以下 factoryan：

- org.kie.spring.factorybeans.RuntimeEnvironmentFactoryBean
- org.kie.spring.factorybeans.RuntimeManagerFactoryBean
- org.kie.spring.factorybeans.TaskServiceFactoryBean

这些 factoryan 提供了一种标准方法来为您的 Spring 应用程序配置 spring.xml 文件。

68.3.1.1. RuntimeEnvironmentFactoryBean bean

RuntimeEnvironmentFactoryBean 工厂 bean 生成 RuntimeEnvironment 的实例。创建 RuntimeManager 实例需要这些实例。

bean 支持创建带有不同默认配置的以下 RuntimeEnvironment 实例类型：

- DEFAULT ：运行时管理器的默认设置或最常用的配置
- EMPTY ：您可以手动配置完全空环境
- DEFAULT_IN_MEMORY ：与 DEFAULT 相同的配置，但没有运行时引擎的持久性

- DEFAULT_KJAR ：与 DEFAULT 相同的配置，但资产从 KJAR 工件（由发行版本 ID 或 GAV 值标识）中加载

- DEFAULT_KJAR_CL ：配置基于 KJAR 构件中的 kmodule.xml 描述符构建

必要属性取决于所选的类型。但是，所有类型都必须显示相关信息。这个要求意味着必须提供以下信息之一：

- knowledgeBase

- 资产

- releaseId

- GroupId, artifactId, version

对于 DEFAULT、DEFAULT_KJAR 和 DEFAULT_KJAR_CL 类型，还必须通过提供以下参数来配置持久性：

- 实体管理器工厂

- 事务管理器

事务管理器必须是 Spring 事务管理器，因为基于此事务管理器对持久性和事务支持进行配置。

另外，您还可以提供一个 EntityManager 实例，而不是从 EntityManagerFactory 创建新实例，例如，您可以使用 Spring 的共享实体管理器。

所有其他属性都是可选的。它们可以覆盖由所选运行时环境类型决定的默认值。

68.3.1.2. RuntimeManageronnectionFactoryyBean bean

RuntimeManagerFactoryBean factoryan 根据提供的 RuntimeEnvironment 实例生成给定类型的 RuntimeManager 实例。

支持的类型与运行时管理器策略对应：

- 单例
- PER_REQUEST
- PER_PROCESS_INSTANCE

如果没有指定类型，默认类型是 SINGLETON。

标识符是一个强制属性，因为每个运行时管理器都必须唯一标识。此工厂创建的所有实例都会被缓存，因此可以使用 destroy 方法正确处理它们（关闭（））。

68.3.1.3. TaskServiceFactoryBean an

TaskServiceFactoryBean factoryan 根据给定属性生成 TaskService 实例。您必须提供以下强制属性：

- 实体管理器工厂
- 事务管理器

事务管理器必须是 Spring 事务管理器，因为基于此事务管理器对持久性和事务支持进行配置。

另外，您还可以提供一个 EntityManager 实例，而不是从 EntityManagerFactory 创建新实例，例如，您可以使用 Spring 的共享实体管理器。

您还可以为任务服务实例设置额外的可选属性：

- ***userGroupCallback ：任务服务必须使用的 UserGroupCallback 的实施，默认值为 MVELUserGroupCallbackImpl***

- ***UserInfo ：任务服务必须使用的 UserInfo 实施，默认值为 DefaultUserInfo***

- ***侦听器 ： TaskLifeCycleEventListener 监听器列表，必须在各种任务中获得通知***

此 factoryan 创建任务服务的一个实例。按照设计，此实例必须在 Spring 环境中的所有 Bean 中共享。

68.3.1.4. 使用 Spring 应用程序配置示例运行时管理器

以下流程是 Spring 应用程序中的一个运行时管理器的完整配置示例。

流程

1. ***配置实体管理器工厂和事务管理器：***

在 spring.xml 文件中配置实体管理器工厂和事务管理器

```
<bean id="jbpmEMF"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="org.jbpm.persistence.spring.jta"/>
</bean>

<bean id="jbpmEM"
class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
  <property name="entityManagerFactory" ref="jbpmEMF"/>
</bean>

<bean id="narayanaUserTransaction" factory-method="userTransaction"
class="com.arjuna.ats.jta.UserTransaction" />

<bean id="narayanaTransactionManager" factory-method="transactionManager"
class="com.arjuna.ats.jta.TransactionManager" />

<bean id="jbpmTxManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
  <property name="transactionManager" ref="narayanaTransactionManager" />
  <property name="userTransaction" ref="narayanaUserTransaction" />
</bean>
```

这些设置定义了以下持久性配置：

- JTA 事务管理器（由 Narayana JTA 提供支持，用于单元测试或 servlet 容器）
- org.jbpm.persistence.spring.jta persistence 单元的实体管理器工厂

2. 配置业务流程资源：

在 spring.xml 文件中配置业务流程资源

```
<bean id="process" factory-method="newClassPathResource"
class="org.kie.internal.io.ResourceFactory">
  <constructor-arg>
    <value>jbpm/processes/sample.bpmn</value>
  </constructor-arg>
</bean>
```

这些设置定义可用于执行的单个进程。资源名称是 sample.bpmn，它必须在类路径上可用。您可以将类路径用作包含尝试进程引擎的资源的简单方法。

3. 使用实体管理器、事务管理器和资源配置 RuntimeEnvironment 实例：

在 spring.xml 文件中配置 RuntimeEnvironment 实例

```
<bean id="runtimeEnvironment"
class="org.kie.spring.factorybeans.RuntimeEnvironmentFactoryBean">
  <property name="type" value="DEFAULT"/>
  <property name="entityManagerFactory" ref="jbpmEMF"/>
  <property name="transactionManager" ref="jbpmTxManager"/>
  <property name="assets">
    <map>
```

```
    <entry key-ref="process"><util:constant static-
field="org.kie.api.io.ResourceType.BPMN2"/></entry>
   </map>
  </property>
</bean>
```

这些设置定义了运行时管理器的默认运行时环境。

4. 根据环境创建一个 RuntimeManager 实例：

```
<bean id="runtimeManager"
class="org.kie.spring.factorybeans.RuntimeManagerFactoryBean" destroy-method="close">
  <property name="identifier" value="spring-rm"/>
  <property name="runtimeEnvironment" ref="runtimeEnvironment"/>
</bean>
```

结果

在这些步骤后，您可以使用运行时管理器使用实体管理器和 JTA 事务管理器在 Spring 环境中执行进程。

您可以在 存储库 中找到针对不同策略的完整 Spring 配置文件。

68.3.1.5. Spring 框架中的运行时管理器的额外配置选项

除了使用 EntityManagerFactory 类和 JTA 事务管理器的配置外，如 第 68.3.1.4 节 “使用 Spring 应用程序配置示例运行时管理器” 所述，您还可以使用 Spring 框架中的运行时管理器的其他配置选项：

- JTA 和 SharedEntityManager 类
- 本地持久性单元和 实体管理器工厂 类
- 本地持久性单元和 SharedEntityManager 类

如果您的应用程序配置了本地持久性单元，并使用 AuditService 服务查询进程引擎历史记录数据，

您必须将 org.kie.api.runtime.EnvironmentName.USE_LOCAL_TRANSACTIONS 环境条目添加到 RuntimeEnvironment 实例配置：

spring.xml 文件中为本地持久性单元的 RuntimeEnvironment 实例配置

```
<bean id="runtimeEnvironment"
class="org.kie.spring.factorybeans.RuntimeEnvironmentFactoryBean">
...
    <property name="environmentEntries" ref="env" />
 </bean>
 ...

 <util:map id="env" key-type="java.lang.String" value-type="java.lang.Object">
 <entry>
  <key>
   <util:constant
    static-field="org.kie.api.runtime.EnvironmentName.USE_LOCAL_TRANSACTIONS" />
  </key>
  <value>true</value>
 </entry>
</util:map>
```

您可以在存储库中找到更多配置选项示例：配置文件和 测试案例。

68.3.2. 使用 Spring 处理引擎服务

您可能想要创建一个动态的 Spring 应用程序，您可以在不重启应用程序的情况下添加和删除业务资产，如流程定义、数据模型、规则和表单。

在这种情况下，使用流程引擎服务。进程引擎服务被设计为与框架无关，单独的模块可在所需框架的附加中实现。

jbpm-kie-services 模块包含服务的代码逻辑。Spring 应用程序可以使用这些纯 Java 服务。

您必须在 Spring 应用程序中添加的唯一代码来配置流程引擎服务，这是 IdentityProvider 接口的实现。这种实现取决于您的安全配置。以下实现示例使用 Spring Security，但可能没有涵盖 Spring 应用程序的所有可用安全功能。

使用 Spring Security 实施 IdentityProvider 接口

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.kie.internal.identity.IdentityProvider;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;

public class SpringSecurityIdentityProvider implements IdentityProvider {

 public String getName() {

  Authentication auth = SecurityContextHolder.getContext().getAuthentication();
  if (auth != null && auth.isAuthenticated()) {
   return auth.getName();
  }
  return "system";
 }

 public List<String> getRoles() {
  Authentication auth = SecurityContextHolder.getContext().getAuthentication();
  if (auth != null && auth.isAuthenticated()) {
   List<String> roles = new ArrayList<String>();

   for (GrantedAuthority ga : auth.getAuthorities()) {
    roles.add(ga.getAuthority());
   }

   return roles;
  }

  return Collections.emptyList();
 }

 public boolean hasRole(String role) {
  return false;
 }

}
```

68.3.2.1. 使用 Spring 应用程序配置流程引擎服务

以下流程是 Spring 应用程序内处理引擎服务的完整配置示例。

流程

1. **配置转换：**

在 spring.xml 文件中配置事务

```
<context:annotation-config />
<tx:annotation-driven />
<tx:jta-transaction-manager />

<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager" />
```

2. **配置 JPA 和持久性：**

在 spring.xml 文件中配置 JPA 和持久性

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" depends-
on="transactionManager">
  <property name="persistenceXmlLocation" value="classpath:/META-INF/jbpm-
persistence.xml" />
</bean>
```

3. **配置安全性和用户和组群信息供应商：**

在 spring.xml 文件中配置安全性和组信息提供程序

```
<util:properties id="roleProperties" location="classpath:/roles.properties" />

<bean id="userGroupCallback"
class="org.jbpm.services.task.identity.JBossUserGroupCallbackImpl">
  <constructor-arg name="userGroups" ref="roleProperties"></constructor-arg>
```

```
</bean>

<bean id="identityProvider" class="org.jbpm.spring.SpringSecurityIdentityProvider"/>
```

4. ***配置运行时管理器工厂。这个因素是 Spring 上下文了解，因此它能够以正确的方式与 Spring 容器交互，并支持必要的服务，包括事务命令服务和任务服务：***

在 spring.xml 文件中配置运行时管理器工厂

```
<bean id="runtimeManagerFactory"
class="org.kie.spring.manager.SpringRuntimeManagerFactoryImpl">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="userGroupCallback" ref="userGroupCallback"/>
</bean>

<bean id="transactionCmdService"
class="org.jbpm.shared.services.impl.TransactionalCommandService">
  <constructor-arg name="emf" ref="entityManagerFactory"></constructor-arg>
</bean>

<bean id="taskService" class="org.kie.spring.factorybeans.TaskServiceFactoryBean"
destroy-method="close">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
  <property name="transactionManager" ref="transactionManager"/>
  <property name="userGroupCallback" ref="userGroupCallback"/>
  <property name="listeners">
    <list>
      <bean class="org.jbpm.services.task.audit.JPATaskLifeCycleEventListener">
        <constructor-arg value="true"/>
      </bean>
    </list>
  </property>
</bean>
```

5. ***将流程引擎服务配置为 Spring Bean：***

在 spring.xml 文件中将进程引擎服务配置为 Spring Bean

```
<!-- Definition service -->
<bean id="definitionService"
class="org.jbpm.kie.services.impl.bpmn2.BPMN2DataServiceImpl"/>

<!-- Runtime data service -->
<bean id="runtimeDataService" class="org.jbpm.kie.services.impl.RuntimeDataServiceImpl">
  <property name="commandService" ref="transactionCmdService"/>
  <property name="identityProvider" ref="identityProvider"/>
  <property name="taskService" ref="taskService"/>
</bean>

<!-- Deployment service -->
<bean id="deploymentService"
class="org.jbpm.kie.services.impl.KModuleDeploymentService" depends-
on="entityManagerFactory" init-method="onInit">
  <property name="bpmn2Service" ref="definitionService"/>
  <property name="emf" ref="entityManagerFactory"/>
  <property name="managerFactory" ref="runtimeManagerFactory"/>
  <property name="identityProvider" ref="identityProvider"/>
  <property name="runtimeDataService" ref="runtimeDataService"/>
</bean>

<!-- Process service -->
<bean id="processService" class="org.jbpm.kie.services.impl.ProcessServiceImpl" depends-
on="deploymentService">
  <property name="dataService" ref="runtimeDataService"/>
  <property name="deploymentService" ref="deploymentService"/>
</bean>

<!-- User task service -->
<bean id="userTaskService" class="org.jbpm.kie.services.impl.UserTaskServiceImpl"
depends-on="deploymentService">
  <property name="dataService" ref="runtimeDataService"/>
  <property name="deploymentService" ref="deploymentService"/>
</bean>

<!-- Register the runtime data service as a listener on the deployment service so it can
receive notification about deployed and undeployed units -->
<bean id="data"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean" depends-
on="deploymentService">
  <property name="targetObject" ref="deploymentService"></property>
  <property name="targetMethod"><value>addListener</value></property>
  <property name="arguments">
  <list>
     <ref bean="runtimeDataService"/>
  </list>
  </property>
</bean>
```

结果

您的 Spring 应用程序可以使用流程引擎服务。

68.4. 与 EJB 集成

进程引擎为企业 Java Beans(EJB)提供完整的集成层。这个层支持本地和远程 EJB 互动。

以下模块提供 EJB 服务：

- jBPM-services-ejb-api ：使用特定于 EJB 的接口和对象扩展 jbpm-services-api 模块的 API 模块
- jBPM-services-ejb-impl:用于核心服务的 EJB 扩展
- jBPM-services-ejb-timer ：基于 EJB Timer 服务的进程引擎调度程序服务实施
- jBPM-services-ejb-client ：用于远程交互的 EJB 远程客户端实施，默认情况下支持 Red Hat JBoss EAP

EJB 层基于流程引擎服务。虽然您使用远程接口，它提供了与核心模块相同的功能。

主要限制会影响部署服务，如果将其用作远程 EJB 服务，则只支持以下方法：

- deploy()
- imageink（）
- activate()
- deactivate()

- *isDeployed()*

其他方法将被排除，因为它们返回运行时对象实例，如 RuntimeManager，这无法通过远程接口使用。

所有其他服务都提供与核心模块中包含的版本相同的 EJB 功能。

68.4.1. EJB 服务的实现

作为进程引擎核心服务的扩展，EJB 服务提供基于 EJB 的执行语义，并基于各种特定于 EJB 的功能。

- *DeploymentServiceEJBImpl 作为带有容器管理的并发的 EJB 单例实施。其锁定类型设置为 写入。*
- *DefinitionServiceEJBImpl 作为带有容器管理的并发的 EJB 单例实施。其整体锁定类型设置为 read，对于 buildProcessDefinition（） 方法，锁定类型设置为 write。*
- *ProcessServiceEJBImpl 作为无状态会话 bean 实现。*
- *RuntimeDataServiceEJBImpl 作为 EJB 单例实施。对于多数方法，锁定类型被设置为 读取。对于以下方法，锁定类型被设置为 写入 ：*
  - *onDeploy()*
  - *onUnDeploy()*
  - *onActivate()*
  - *onDeactivate()*

- UserTaskServiceEJBImpl 作为无状态会话 bean 实现。

Transactions

EJB 容器管理 EJB 服务中的事务。因此，您不需要在应用程序代码中设置任何事务管理器或用户事务。

用户身份提供程序

默认身份提供程序基于 EJBContext 接口，并依赖于名称和角色的调用者主体信息。IdentityProvider 接口提供与角色相关的两种方法：

- getRoles（） 返回空列表，因为 EJBContext 接口不提供获取特定用户的所有角色的选项
- 具有Role（） 委派给上下文的 isCallerInRole（） 方法

为确保 EJB 环境可以使用有效信息，您必须遵循标准的 JEE 安全实践来进行身份验证和授权用户。如果没有为 EJB 服务配置验证或授权，则始终假定一个匿名用户。

如果使用不同的安全模型，您可以使用 CDI 样式注入 IdentityProvider 对象用于 EJB 服务。在本例中，创建一个有效的 CDI Bean，它将实现 org.kie.internal.identity.IdentityProvider 接口，并使此 bean 可供您的应用程序注入。这种实施优先于基于 EJBContext的身份提供商。

部署同步

部署同步默认为启用，并且尝试每 3 秒同步任何部署。它作为带有容器管理的并发的 EJB 单例实施。其锁定类型设置为 写入。它使用 EJB 计时器服务来调度同步作业。

EJB 调度程序服务

进程引擎使用调度程序服务来处理基于时间的活动，如计时器事件和期限。在 EJB 环境中运行时，进程引擎使用基于 EJB 计时器服务的调度程序。它为所有 RuntimeManager 实例注册此调度程序。

您可能需要使用特定于应用服务器的配置来支持集群操作。

UserGroupCallback 和 UserInfo 实现选择

UserGroupCallback 和 UserInfo 接口所需的实现可能因不同的应用程序而异。这些接口不能直接注入 EJB。您可以使用以下系统属性来选择现有的实现，或使用这些接口的自定义实现流程引擎：

- org.jbpm.ht.callback ：此属性为 UserGroupCallback 接口选择实现：
  - MV EL：默认实施，通常用于测试。
  - LDAP ：基于 LDAP 的实施。此实施需要在 jbpm.usergroup.callback.properties 文件中进行额外的配置。
  - db ：基于数据库的实施。此实施需要在 jbpm.usergroup.callback.properties 文件中进行额外的配置。
  - JAAS ：从容器请求用户信息的实现。
  - props ：基于属性的简单回调。这种实施需要额外属性文件，其中包含所有用户和组。
  - 自定义 ：一个自定义实施。您必须在 org.jbpm.ht.custom.callback 系统属性中提供实现的完全限定类名称。
- org.jbpm.ht.userinfo:此属性为 UserInfo 接口选择实现：
  - LDAP ：基于 LDAP 的实施。此实施需要在 jbpm-user.info.properties 文件中进行额外的配置。
  - db ：基于数据库的实施。此实施需要在 jbpm-user.info.properties 文件中进行额外的配置。
  - props ：基于属性的简单实施。这种实施需要包含所有用户信息的额外属性文件。
  - 自定义 ：一个自定义实施。您必须在 org.jbpm.ht.custom.userinfo 系统属性中提供实现的完全限定域名。

通常，在应用服务器或 JVM 的启动配置中设置系统属性。在使用服务之前，您还可以在代码中设置属性。例如，您可以提供一个自定义 @Startup bean 来配置这些系统属性。

68.4.2. 本地 EJB 接口

以下本地 EJB 服务接口扩展了核心服务：

- org.jbpm.services.ejb.api.DefinitionServiceEJBLocal
- org.jbpm.services.ejb.api.DeploymentServiceEJBLocal
- org.jbpm.services.ejb.api.ProcessServiceEJBLocal
- org.jbpm.services.ejb.api.RuntimeDataServiceEJBLocal
- org.jbpm.services.ejb.api.UserTaskServiceEJBLocal

您必须使用这些接口作为注入点，并使用 @EJB 标注：

使用本地 EJB 服务接口

```
@EJB
private DefinitionServiceEJBLocal bpmn2Service;

@EJB
private DeploymentServiceEJBLocal deploymentService;

@EJB
private ProcessServiceEJBLocal processService;

@EJB
private RuntimeDataServiceEJBLocal runtimeDataService;
```

注入这些接口后，在它们上调用操作方式与核心模块相同。使用本地接口不存在限制。

68.4.3. 远程 EJB 接口

以下专用远程 EJB 接口扩展了核心服务：

- org.jbpm.services.ejb.api.DefinitionServiceEJBRemote
- org.jbpm.services.ejb.api.DeploymentServiceEJBRemote
- org.jbpm.services.ejb.api.ProcessServiceEJBRemote
- org.jbpm.services.ejb.api.RuntimeDataServiceEJBRemote
- org.jbpm.services.ejb.api.UserTaskServiceEJBRemote

您可以以与本地接口相同的方式使用这些接口，但处理自定义类型除外。

您可以通过两种方式定义自定义类型。全局 定义的类型在应用程序类路径上可用，并包含在企业应用程序中。如果您在 部署单元本地 定义类型，则会在项目依赖项中声明该类型（例如，在 KJAR 文件中），并在部署时解析。

全局可用类型不需要任何特殊处理。EJB 容器在处理远程请求时自动提取数据。但是，默认情况下，本地自定义类型对 EJB 容器不可见。

进程引擎 EJB 服务提供了一种使用自定义类型的机制。它们提供以下额外类型：

- org.jbpm.services.ejb.remote.api.RemoteObject: a serializable wrapper class for single-value 参数
- org.jbpm.services.ejb.remote.api.RemoteMap: A dedicated java.util.Map 实现，以简化接受自定义对象输入的服务方法的远程调用。映射的内部实施包含已序列化的内容，以避免在发送时进行额外的序列化。

这种实现不包括在发送数据时通常不使用的 java.util.Map 的一些方法。

这些特殊对象使用 ObjectInputStream 对象执行大量序列化。它们消除了在 EJB 客户端/容器中对数据进行序列化的需求。因为不需要序列化，所以不需要与 EJB 容器共享自定义数据模型。

以下示例代码适用于本地类型和远程 EJB 服务：

使用带有远程 EJB 服务的本地类型

```
// Start a process with custom types via remote EJB

Map<String, Object> parameters = new RemoteMap();
Person person = new org.jbpm.test.Person("john", 25, true);
parameters.put("person", person);

Long processInstanceId = processService.startProcess(deploymentUnit.getIdentifier(),
"custom-data-project.work-on-custom-data", parameters);

// Fetch task data and complete a task with custom types via remote EJB
Map<String, Object> data = userTaskService.getTaskInputContentByTaskId(taskId);

Person fromTaskPerson = data.get("_person");
fromTaskPerson.setName("John Doe");

RemoteMap outcome = new RemoteMap();
outcome.put("person_", fromTaskPerson);

userTaskService.complete(taskId, "john", outcome);
```

类似地，您可以使用 RemoteObject 类将事件发送到进程实例：

```
// Send an event with a custom type via remote EJB
Person person = new org.jbpm.test.Person("john", 25, true);

RemoteObject myObject = new RemoteObject(person);

processService.signalProcessInstance(processInstanceId, "MySignal", myObject);
```

68.4.4. 远程 EJB 客户端

远程客户端支持通过实施 ClientServiceFactory 接口来提供，该界面是针对应用服务器特定代码的教导：

ClientServiceonnectionFactoryy 接口的 定义

```
/**
 * Generic service factory used for remote lookups that are usually container specific.
 *
 */
public interface ClientServiceFactory {

 /**
  * Returns unique name of given factory implementation
  * @return
  */
 String getName();

 /**
  * Returns remote view of given service interface from selected application
  * @param application application identifier on the container
  * @param serviceInterface remote service interface to be found
  * @return
  * @throws NamingException
  */
 <T> T getService(String application, Class<T> serviceInterface) throws NamingException;
}
```

您可以使用 ServiceLoader 机制动态注册实现。默认情况下，红帽 JBoss EAP 中仅提供一个实施。

每个 ClientServiceonnectionFactory y 实施都必须提供名称。此名称用于在客户端 registry 中注册。按名称查找实施。

以下代码获取默认的 Red Hat JBoss EAP 远程客户端：

获取默认的 Red Hat JBoss EAP 远程客户端

```
// Retrieve a valid client service factory
ClientServiceFactory factory = ServiceFactoryProvider.getProvider("JBoss");
```

```
// Set the application variable to the module name
String application = "sample-war-ejb-app";

// Retrieve the required service from the factory
DeploymentServiceEJBRemote deploymentService = factory.getService(application,
DeploymentServiceEJBRemote.class);
```

检索服务后，您可以使用其方法。

使用 Red Hat JBoss EAP 和远程客户端时，您可以添加以下 Maven 依赖项来引入所有 EJB 客户端库：

```
<dependency>
  <groupId>org.jboss.as</groupId>
  <artifactId>jboss-as-ejb-client-bom</artifactId>
  <version>7.4.14.Final</version> <!-- use the valid version for the server you run on -->
  <optional>true</optional>
  <type>pom</type>
</dependency>
```

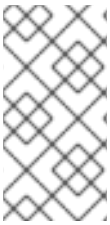
68.5. 与 OSGI 集成

所有核心进程引擎 JAR 文件和核心依赖项都是启用 OSGi。以下额外的进程引擎 JAR 文件也启用了 OSGI：

- jbpm-flow
- jbpm-flow-builder
- jbpm-bpmn2

基于 OSGi 的 JAR 文件包含 META-INF 目录中的 MANIFEST.MF 文件。这些文件包含所需依赖项等数据。您可以将此类 JAR 文件添加到 OSGi 环境。

有关 OSGi 基础架构的更多信息，请参阅 OSGI 文档。



注意

支持与 OSGi 框架集成已被弃用。它不会收到任何新的增强功能或功能，并将在以后的版本中被删除。

附录 A. 版本信息

文档最新更新于 2024 年 3 月 14 日星期四。

附录 B. 联系信息

Red Hat Process Automation Manager 文档团队 ：brms-docs@redhat.com