

Activiti 5.16 用户手册

Activiti 5.16 用户手册

1.	简介	1
1.1.	协议	1
1.2.	下载	1
1.3.	源码	1
1.4.	必要的软件	1
1.4.1.	JDK 6+	1
1.4.2.	Eclipse Indigo 和 Juno	1
1.5.	报告问题	1
1.6.	试验性功能	1
1.7.	内部实现类	2
2.	开始学习	3
2.1.	一分钟入门	3
2.2.	安装Activiti	3
2.3.	安装Activiti数据库	4
2.4.	引入Activiti jar和依赖	4
2.5.	下一步	5
3.	配置	6
3.1.	创建ProcessEngine	6
3.2.	ProcessEngineConfiguration bean	7
3.3.	数据库配置	7
3.4.	JNDI数据库配置	8
3.4.1.	使用	9
3.4.2.	配置	10
3.5.	支持的数据库	10
3.6.	创建数据库表	11
3.7.	理解数据库表的命名	12
3.8.	数据库升级	12
3.9.	启用Job执行器	12
3.10.	配置邮件服务器	13
3.11.	配置历史	13
3.12.	为表达式和脚本暴露配置	13
3.13.	配置部署缓存	13
3.14.	日志	13
3.15.	映射诊断上下文	14
3.16.	事件处理	14
3.16.1.	事件监听器实现	15
3.16.2.	配置与安装	15
3.16.3.	在运行阶段添加监听器	16
3.16.4.	为流程定义添加监听器	17
3.16.4.1.	让监听器执行用户定义的逻辑	17
3.16.4.2.	监听抛出BPMN事件	17
3.16.4.3.	流程定义中监听器的注意事项	18
3.16.5.	通过API分发事件	18
3.16.6.	支持的事件类型	19
3.16.7.	附加信息	22
4.	Activiti API	23
4.1.	流程引擎的API和服务	23
4.2.	异常策略	25
4.3.	使用Activiti的服务	25

4.3.1. 发布流程	26
4.3.2. 启动一个流程实例	28
4.3.3. 完成任务	28
4.3.4. 挂起, 激活一个流程	29
4.3.5. 更多知识	29
4.4. 查询API	29
4.5. 表达式	30
4.6. 单元测试	31
4.7. 调试单元测试	32
4.8. web应用中的流程引擎	34
5. Spring集成	36
5.1. ProcessEngineFactoryBean	36
5.2. 事务	36
5.3. 表达式	38
5.4. 资源的自动部署	39
5.5. 单元测试	40
5.6. 基于注解的配置	41
5.7. JPA 和 Hibernate 4.2.x	43
6. 部署	45
6.1. 业务文档	45
6.1.1. 编程式部署	45
6.1.2. 通过Activiti Explorer控制台部署	45
6.2. 外部资源	46
6.2.1. Java类	46
6.2.2. 在流程中使用Spring beans	46
6.2.3. 创建独立应用	47
6.3. 流程定义的版本	47
6.4. 提供流程图片	48
6.5. 自动生成流程图片	49
6.6. 类别	50
7. BPMN 2.0介绍	51
7.1. 啥是BPMN?	51
7.2. 定义一个流程	51
7.3. 快速起步: 10分钟教程	52
7.3.1. 前提	52
7.3.2. 目标	52
7.3.3. 用例	53
7.3.4. 流程图	53
7.3.5. XML内容	53
7.3.6. 启动一个流程实例	54
7.3.7. 任务列表	55
7.3.8. 领取任务	56
7.3.9. 完成任务	57
7.3.10. 结束流程	58
7.3.11. 代码总结	59
7.3.12. 更多思考	60
8. BPMN 2.0结构	61
8.1. 自定义扩展	61
8.2. 事件 (Event)	61

8.2.1. 事件定义	61
8.2.2. 定时器事件定义	62
8.2.3. 错误事件定义	63
8.2.4. 信号事件定义	63
8.2.4.1. 触发信号事件	63
8.2.4.2. 捕获信号事件	64
8.2.4.3. 查询信号事件的订阅	64
8.2.4.4. 信号事件范围	64
8.2.4.5. 信号事件实例	64
8.2.5. 消息事件定义	66
8.2.5.1. 触发消息事件	66
8.2.5.2. 查询消息事件的订阅	67
8.2.5.3. 消息事件实例	67
8.2.6. 开始事件	67
8.2.7. 空开始事件	68
8.2.7.1. 描述	68
8.2.7.2. 图形标记	68
8.2.7.3. XML结构	68
8.2.7.4. 空开始事件的自定义扩展	68
8.2.8. 定时开始事件	69
8.2.8.1. 描述	69
8.2.8.2. 图形标记	69
8.2.8.3. XML内容	69
8.2.9. 消息开始事件	69
8.2.9.1. 描述	69
8.2.9.2. 图形标记	70
8.2.9.3. XML内容	71
8.2.10. 信号开始事件	71
8.2.10.1. 描述	71
8.2.10.2. 图形标记	71
8.2.10.3. XML格式	72
8.2.11. 错误开始事件	72
8.2.11.1. 描述	72
8.2.11.2. 图形标记	72
8.2.11.3. XML内容	72
8.2.12. 结束事件	73
8.2.13. 空结束事件	73
8.2.13.1. 描述	73
8.2.13.2. 图形标记	73
8.2.13.3. XML内容	73
8.2.14. 错误结束事件	73
8.2.14.1. 描述	73
8.2.14.2. 图形标记	73
8.2.14.3. XML内容	74
8.2.15. 取消结束事件	74
8.2.15.1. 描述	74
8.2.15.2. 图形标记	74
8.2.15.3. XML内容	75
8.2.16. 边界事件	75

8.2.17. 定时边界事件	75
8.2.17.1. 描述	75
8.2.17.2. 图形标记	75
8.2.17.3. XML内容	76
8.2.17.4. 边界事件的已知问题	77
8.2.18. 错误边界事件	77
8.2.18.1. 描述	77
8.2.18.2. 图形标记	78
8.2.18.3. XML内容	78
8.2.18.4. 实例	79
8.2.19. 信号边界事件	79
8.2.19.1. 描述	79
8.2.19.2. 图形标记	80
8.2.19.3. XML内容	80
8.2.19.4. 实例	80
8.2.20. 消息边界事件	80
8.2.20.1. 描述	80
8.2.20.2. 图形标记	80
8.2.20.3. XML内容	80
8.2.20.4. 实例	81
8.2.21. 取消边界事件	81
8.2.21.1. 描述	81
8.2.21.2. 图形标记	81
8.2.21.3. XML内容	81
8.2.22. 补偿边界事件	82
8.2.22.1. 描述	82
8.2.22.2. 图形标记	82
8.2.22.3. XML内容	82
8.2.23. 中间捕获事件	83
8.2.24. 定时中间捕获事件	83
8.2.24.1. 描述	83
8.2.24.2. 图形标记	83
8.2.24.3. XML内容	83
8.2.25. 信号中间捕获事件	83
8.2.25.1. 描述	83
8.2.25.2. 图形标记	84
8.2.25.3. XML内容	84
8.2.25.4. 实例	84
8.2.26. 消息中间捕获事件	84
8.2.26.1. 描述	84
8.2.26.2. 图形标记	84
8.2.26.3. XML内容	84
8.2.26.4. 实例	84
8.2.27. 内部触发事件	85
8.2.28. 中间触发空事件	85
8.2.29. 信号中间触发事件	85
8.2.29.1. 描述	85
8.2.29.2. 图形标记	85
8.2.29.3. XML内容	86

8.2.29.4. 实例	86
8.2.30. 补偿中间触发事件	86
8.2.30.1. 描述	86
8.2.30.2. 图形标记	88
8.2.30.3. XML内容	88
8.3. 顺序流	88
8.3.1. 描述	88
8.3.2. 图形标记	88
8.3.3. XML内容	88
8.3.4. 条件顺序流	89
8.3.4.1. 描述	89
8.3.4.2. 图形标记	89
8.3.4.3. XML内容	89
8.3.5. 默认顺序流	90
8.3.5.1. 描述	90
8.3.5.2. 图形标记	90
8.3.5.3. XML内容	90
8.4. 网关	91
8.4.1. 排他网关	91
8.4.1.1. 描述	91
8.4.1.2. 图形标记	92
8.4.1.3. XML内容	92
8.4.2. 并行网关	93
8.4.2.1. 描述	93
8.4.2.2. 图形标记	93
8.4.2.3. XML内容	93
8.4.3. 包含网关	95
8.4.3.1. 描述	95
8.4.3.2. 图形标记	95
8.4.3.3. XML内容	96
8.4.4. 基于事件网关	97
8.4.4.1. 描述	97
8.4.4.2. 图形标记	97
8.4.4.3. XML内容	98
8.4.4.4. 实例	98
8.5. 任务	99
8.5.1. 用户任务	99
8.5.1.1. 描述	99
8.5.1.2. 图形标记	99
8.5.1.3. XML内容	99
8.5.1.4. 持续时间	100
8.5.1.5. 用户分配	100
8.5.1.6. Activiti对任务分配的扩展	101
8.5.2. 脚本任务	102
8.5.2.1. 描述	102
8.5.2.2. 图形标记	102
8.5.2.3. XML内容	103
8.5.2.4. 脚本中的变量	103
8.5.2.5. 脚本结果	104

8.5.3. Java服务任务	104
8.5.3.1. 描述	104
8.5.3.2. 图形标记	104
8.5.3.3. XML内容	104
8.5.3.4. 实现	105
8.5.3.5. 属性注入	106
8.5.3.6. 服务任务结果	108
8.5.3.7. 处理异常	108
8.5.3.8. 在JavaDelegate里使用activiti服务	109
8.5.4. Web Service任务	110
8.5.4.1. 描述	110
8.5.4.2. 图形标记	110
8.5.4.3. XML内容	110
8.5.4.4. Web Service任务IO规范	111
8.5.4.5. Web Service任务数据输入关联	111
8.5.4.6. Web Service任务数据输出关联	112
8.5.5. 业务规则任务	112
8.5.5.1. 描述	112
8.5.5.2. 图形标记	113
8.5.5.3. XML内容	113
8.5.6. 邮件任务	114
8.5.6.1. 邮件服务器配置	114
8.5.6.2. 定义一个邮件任务	114
8.5.6.3. 使用实例	115
8.5.7. Mule任务	117
8.5.7.1. 定义一个mule任务	117
8.5.7.2. 应用实例	117
8.5.8. Camel任务	117
8.5.8.1. 定义camel任务	118
8.5.8.2. 简单Camel调用	118
8.5.8.3. 乒乓实例	119
8.5.8.4. 异步乒乓实例	121
8.5.8.5. 从camel规则中实例化工作流	122
8.5.9. 手工任务	122
8.5.9.1. 描述	122
8.5.9.2. 图形标记	122
8.5.9.3. XML内容	122
8.5.10. Java接收任务	123
8.5.10.1. 描述	123
8.5.10.2. 图形标记	123
8.5.10.3. XML内容	123
8.5.11. Shell任务	123
8.5.11.1. 描述	123
8.5.11.2. 定义shell任务	123
8.5.11.3. 应用实例	124
8.5.12. 执行监听器	124
8.5.12.1. 流程监听器的属性注入	126
8.5.13. 任务监听器	127
8.5.14. 多实例（循环）	128

8.5.14.1. 描述	128
8.5.14.2. 图形标记	129
8.5.14.3. XML内容	129
8.5.14.4. 边界事件和多实例	131
8.5.15. 补偿处理器	131
8.5.15.1. 描述	131
8.5.15.2. 图形标志	132
8.5.15.3. XML内容	132
8.6. 子流程和调用节点	132
8.6.1. 子流程	132
8.6.1.1. 描述	132
8.6.1.2. 图形标记	132
8.6.1.3. XML内容	133
8.6.1.4. 事件子流程	134
8.6.2.1. 描述	134
8.6.2.2. 图像标记	134
8.6.2.3. XML内容	134
8.6.2.4. 实例	135
8.6.2.5. 事务子流程	137
8.6.3.1. 描述	137
8.6.3.2. 图形标记	139
8.6.3.3. XML内容	139
8.6.3.4. 实例	140
8.6.2.6. 调用活动（子流程）	140
8.6.4.1. 描述	140
8.6.4.2. 图形标记	141
8.6.4.3. XML内容	141
8.6.4.4. 传递变量	141
8.6.4.5. 实例	141
8.7. 事务和并发	142
8.7.1. 异步操作	142
8.7.2. 排他任务	143
8.7.2.1. 为什么要使用排他任务？	143
8.7.2.2. 什么是排他job？	145
8.8. 流程实例授权	145
8.9. 数据对象	146
9. 表单	148
9.1. 表单属性	148
9.2. 外置表单的渲染	151
10. JPA	152
10.1. 要求	152
10.2. 配置	152
10.3. 用法	153
10.3.1. 简单例子	153
10.3.2. 查询JPA流程变量	155
10.3.3. 使用Spring beans和JPA结合的高级例子	155
11. 历史	158
11.1. 查询历史	158
11.1.1. HistoricProcessInstanceQuery	158

11.1.2. HistoricVariableInstanceQuery	158
11.1.3. HistoricActivityInstanceQuery	159
11.1.4. HistoricDetailQuery	159
11.1.5. HistoricTaskInstanceQuery	160
11.2. 历史配置	160
11.3. 审计目的的历史	161
12. Eclipse Designer	162
12.1. Installation	162
12.2. Activiti Designer 编辑器的特性	163
12.3. Activiti Designer 的BPMN 特性	170
12.4. Activiti Designer 部署特性	180
12.5. 扩展Activiti Designer	182
12.5.1. 定制画板	182
12.5.1.1. 扩展的设置 (Eclipse/Maven)	183
12.5.1.2. 将扩展应用到Activiti Designer	185
12.5.1.3. 向画板添加形状	188
12.5.1.4. 属性的类型	191
12.5.1.5. 禁用画板中默认形状	195
12.5.2. 校验图形和导出到自定义的输出格式	197
12.5.2.1. 创建ProcessValidator扩展	198
12.5.2.2. 创建ExportMarshaller扩展	199
13. Activiti Explorer	201
13.1. 流程图	201
13.2. 任务	203
13.3. 启动流程实例	203
13.4. 我的流程实例	204
13.5. 管理	205
13.6. 报表	209
13.6.1. 报告数据JSON	211
13.6.2. 实例流程	211
13.6.3. 报告开始表单	213
13.6.4. 流程例子	214
13.7. 修改数据库	214
14. Activiti Modeler	215
14.1. 编辑模型	217
14.2. 导入模型	219
14.3. 把发布的流程定义转换成可编辑的模型	219
14.4. 把模型导出成BPMN XML	220
14.5. 把模型部署到Activiti引擎中	221
15. REST API	223
15.1. 通用Activiti REST原则	223
15.1.1. 安装与认证	223
15.1.2. 使用Tomcat	223
15.1.3. 方法和返回值	224
15.1.4. 错误响应体	225
15.1.5. 请求参数	225
15.1.5.1. URL片段	225
15.1.5.2. Rest URL查询参数	225
15.1.5.3. JSON内容参数	226

15.1.5.4. 分页与排序	226
15.1.5.5. JSON查询变量格式	226
15.1.5.6. 变量格式	227
15.2. 部署	229
15.2.1. 部署列表	229
15.2.2. 获得一个部署	230
15.2.3. 创建新部署	231
15.2.4. 删除部署	231
15.2.5. 列出部署内的资源	231
15.2.6. 获取部署资源	232
15.2.7. 获取部署资源的内容	233
15.3. 流程定义	234
15.3.1. 流程定义列表	234
15.3.2. 获得一个流程定义	236
15.3.3. 更新流程定义的分类	237
15.3.4. 获得一个流程定义的资源内容	237
15.3.5. 获得流程定义的BPMN模型	238
15.3.6. 暂停流程定义	238
15.3.7. 激活流程定义	239
15.3.8. 获得流程定义的所有候选启动者	239
15.3.9. 为流程定义添加一个候选启动者	240
15.3.10. 删除流程定义的候选启动者	241
15.3.11. 获得流程定义的一个候选启动者	242
15.4. 模型	242
15.4.1. 获得模型列表	242
15.4.2. 获得一个模型	244
15.4.3. 更新模型	245
15.4.4. 新建模型	246
15.4.5. 删除模型	246
15.4.6. 获得模型的可编译源码	247
15.4.7. 设置模型的可编辑源码	247
15.4.8. 获得模型的附加可编辑源码	247
15.4.9. 设置模型的附加可编辑源码	248
15.5. 流程实例	248
15.5.1. 获得流程实例	248
15.5.2. 删除流程实例	249
15.5.3. 激活或挂起流程实例	249
15.5.4. 启动流程实例	250
15.5.5. 显示流程实例列表	251
15.5.6. 查询流程实例	253
15.5.7. 获得流程实例的流程图	254
15.5.8. 获得流程实例的参与者	255
15.5.9. 为流程实例添加一个参与者	255
15.5.10. 删除一个流程实例的参与者	256
15.5.11. 列出流程实例的变量	257
15.5.12. 获得流程实例的一个变量	257
15.5.13. 创建(或更新)流程实例变量	258
15.5.14. 更新一个流程实例变量	259
15.5.15. 创建一个新的二进制流程变量	260

15.5.16. 更新一个二进制的流程实例变量	261
15.6. 分支	262
15.6.1. 获取一个分支	262
15.6.2. 对分支执行操作	262
15.6.3. 获得一个分支的所有活动节点	263
15.6.4. 获取分支列表	264
15.6.5. 查询分支	266
15.6.6. 获取分支的变量列表	267
15.6.7. 获得分支的一个变量	268
15.6.8. 新建(或更新) 分支变量	269
15.6.9. 更新分支变量	270
15.6.10. 创建一个二进制变量	271
15.6.11. 更新已经已存在的二进制分支变量	271
15.7. 任务	272
15.7.1. 获取任务	272
15.7.2. 任务列表	273
15.7.3. 查询任务	277
15.7.4. 更新任务	278
15.7.5. 操作任务	279
15.7.6. 删除任务	280
15.7.7. 获得任务的变量	280
15.7.8. 获取任务的一个变量	281
15.7.9. 获取变量的二进制数据	282
15.7.10. 创建任务变量	283
15.7.11. 创建二进制任务变量	284
15.7.12. 更新任务的一个已有变量	285
15.7.13. 更新一个二进制任务变量	286
15.7.14. 删除任务变量	287
15.7.15. 删除任务的所有局部变量	287
15.7.16. 获得任务的所有IdentityLink	288
15.7.17. 获得一个任务的所有组或用户的IdentityLink	288
15.7.18. 获得一个任务的一个IdentityLink	289
15.7.19. 为任务创建一个IdentityLink	289
15.7.20. 删除任务的一个IdentityLink	290
15.7.21. 为任务创建评论	290
15.7.22. 获得任务的所有评论	291
15.7.23. 获得任务的一个评论	292
15.7.24. 删除任务的一条评论	293
15.7.25. 获得任务的所有事件	293
15.7.26. 获得任务的一个事件	294
15.7.27. 为任务创建一个附件, 包含外部资源的链接	294
15.7.28. 为任务创建一个附件, 包含附件文件	295
15.7.29. 获得任务的所有附件	296
15.7.30. 获得任务的一个附件	297
15.7.31. 获取附件的内容	297
15.7.32. 删除任务的一个附件	298
15.8. 历史	298
15.8.1. 获得历史流程实例	298
15.8.2. 历史流程实例列表	299

15.8.3. 查询历史流程实例	301
15.8.4. 删除历史流程实例	302
15.8.5. 获取历史流程实例的IdentityLink	303
15.8.6. 获取历史流程实例变量的二进制数据	303
15.8.7. 为历史流程实例创建一条新评论	303
15.8.8. 获得一个历史流程实例的所有评论	304
15.8.9. 获得历史流程实例的一条评论	305
15.8.10. 删除历史流程实例的一条评论	305
15.8.11. 获得单独历史任务实例	306
15.8.12. 获取历史任务实例	307
15.8.13. 查询历史任务实例	310
15.8.14. 删除历史任务实例	312
15.8.15. 获得历史任务实例的IdentityLink	312
15.8.16. 获取历史任务实例变量的二进制值	312
15.8.17. 获取历史活动实例	313
15.8.18. 查询历史活动实例	314
15.8.19. 列出历史变量实例	315
15.8.20. 查询历史变量实例	316
15.8.21. 获取历史任务实例变量的二进制值	317
15.8.22. 获取历史细节	318
15.8.23. 查询历史细节	319
15.8.24. 获取历史细节变量的二进制数据	320
15.9. 表单	320
15.9.1. 获取表单数据	320
15.9.2. 提交任务表单数据	321
15.10. 数据库表	322
15.10.1. 表列表	322
15.10.2. 获得一张表	323
15.10.3. 获得表的列信息	323
15.10.4. 获得表的行数据	324
15.11. 引擎	325
15.11.1. 获得引擎属性	325
15.11.2. 获得引擎信息	325
15.12. 运行时	326
15.12.1. 接收信号事件	326
15.13. 作业	327
15.13.1. 获取一个作业	327
15.13.2. 删除作业	328
15.13.3. 执行作业	328
15.13.4. 获得作业的异常堆栈	328
15.13.5. 获得作业列表	329
15.14. 用户	331
15.14.1. 获得一个用户	331
15.14.2. 获取用户列表	331
15.14.3. 更新用户	333
15.14.4. 创建用户	333
15.14.5. 删除用户	334
15.14.6. 获取用户图片	334
15.14.7. 更新用户图片	334

15.14.8. 列出用户列表	335
15.14.9. 获取用户信息	335
15.14.10. 更新用户的信息	336
15.14.11. 创建用户信息条目	337
15.14.12. 删除用户的信息	337
15.15. 群组	338
15.15.1. 获得群组	338
15.15.2. 获取群组列表	338
15.15.3. 更新群组	339
15.15.4. 创建群组	340
15.15.5. 删除群组	340
15.15.6. 获取群组的成员	341
15.15.7. 为群组添加一个成员	341
15.15.8. 删除群组的成员	341
15.16. 传统REST – 通用方法	342
15.17. 资源	343
15.17.1. 上传发布	343
15.17.2. 获取发布	343
15.17.3. 获取发布资源	344
15.17.4. 获取发布的一个资源	344
15.17.5. 删除发布	344
15.17.6. 删除发布	344
15.18. 引擎	345
15.18.1. 获取流程引擎	345
15.19. 流程	345
15.19.1. 流程定义列表	345
15.19.2. 获得流程定义表单属性	346
15.19.3. 获得流程定义表单资源	346
15.19.4. 获取流程定义图	347
15.19.5. 启动流程实例	347
15.19.6. 流程实例列表	347
15.19.7. 获得流程实例细节	348
15.19.8. 获得流程实例图	349
15.19.9. 获得流程实例的任务	349
15.19.10. 继续特定流程实例的活动 (receiveTask)	350
15.19.11. 触发特定流程实例的信号	350
15.20. 任务	351
15.20.1. 获得任务简介	351
15.20.2. 任务列表	351
15.20.3. 获取任务	352
15.20.4. 获取任务表单	353
15.20.5. 执行任务操作	353
15.20.6. 表单属性列表	354
15.20.7. 为任务添加一个附件	354
15.20.8. 获得任务附件	355
15.20.9. 为任务添加一个url	355
15.21. 身份	355
15.21.1. 登录	355
15.21.2. 获得用户	356

15.21.3. 列出用户的群组	356
15.21.4. 查询用户	356
15.21.5. 创建用户	357
15.21.6. 为群组添加用户	357
15.21.7. 从群组删除用户	358
15.21.8. 获得用户图片	358
15.21.9. 获得群组	358
15.21.10. 群组用户列表	358
15.21.11. 查询群组	359
15.21.12. 创建群组	359
15.21.13. 为群组添加用户	360
15.21.14. 为群组删除用户	360
15.22. 管理	360
15.22.1. 作业列表	360
15.22.2. 获得作业	361
15.22.3. 执行一个作业	361
15.22.4. 执行多个作业	362
15.22.5. 数据库表列表	362
15.22.6. 获得表元数据	362
15.22.7. 获得表数据	363
16. 集成CDI	364
16.1. 设置activiti-cdi	364
16.1.1. 查找流程引擎	364
16.1.2. 配置Process Engine	364
16.1.3. 发布流程	366
16.2. 基于CDI环境的流程执行	366
16.2.1. 与流程实例进行关联交互	366
16.2.2. 声明式流程控制	367
16.2.3. 在流程中引用bean	367
16.2.4. 使用@BusinessProcessScoped beans	368
16.2.5. 注入流程变量	368
16.2.6. 接收流程事件	368
16.2.7. 更多功能	369
16.3. 已知的问题	369
17. 集成LDAP	370
17.1. 用法	370
17.2. 用例	370
17.3. 配置	370
17.4. 属性	371
17.5. 为Explorer集成LDAP	375
18. 高级功能	376
18.1. 监听流程解析	376
18.2. 支持高并发的UUID id生成器	377
18.3. 多租户	378
18.4. 执行自定义SQL	379
18.5. 使用ProcessEngineConfigurator实现高级流程引擎配置	381
18.6. 启用安全的BPMN 2.0 xml	382
18.7. 事件日志（实验）	382
19. 使用Activiti-Crystalball进行流程仿真（实验）	384

19.1. 介绍	384
19.1.1. 简介	384
19.1.2. CrystalBall是独立的	384
19.2. CrystalBall内部	384
19.3. 历史分析	385
19.3.1. 历史的事件	385
19.3.2. 回放	385
19.3.3. 调试流程引擎	386
19.3.4. 重播	387

第#1#章#简介

1. 1. #协议

Activiti是基于Apache V2协议 [[..../license.txt](#)]发布的。

1. 2. #下载

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

1. 3. #源码

发布包里包含大部分的已经打好jar包的源码。如果想找到并构建完整的源码库，请参考 wiki “构建发布包” [[http://docs.codehaus.org/display/ACT/Developers+Guide#DevelopersGuide-Buildingthedistribution](#)]。

1. 4. #必要的软件

1. 4. 1. #JDK 6+

Activiti需要运行在JDK 6或以上版本上。进入 Oracle Java SE 下载页面 [[http://www.oracle.com/technetwork/java/javase/downloads/index.html](#)] 点击“下载 JDK”按钮。页面上也提供了安装的方法。为了验证是否安装成功，可以在命令行中执行 `java -version`。它将会打印出安装的JDK的版本。

1. 4. 2. #Eclipse Indigo 和 Juno

(译者注：Eclipse 3.7 版本代号 Indigo 靛青，Eclipse 4.2 版本代号 Juno 朱诺)。在Eclipse下载页面 [[http://www.eclipse.org/downloads/](#)] 下载你选择的eclipse发布包。解压下载文件，你就可以通过eclipse目录下的eclipse文件启动它。此外，在该用户指南后面，专门有一章介绍安装eclipse设计器插件。

1. 5. #报告问题

任何一个自觉的开发者都应该看看 如何聪明的提出问题 [[http://www.catb.org/~esr/faqs/smart-questions.html](#)]。

看完之后，你可以在用户论坛 [[http://forums.activiti.org/en/viewforum.php?f=3](#)] 上进行提问和评论，或者在JIRA问题跟踪系统 [[http://jira.codehaus.org/browse/ACT](#)] 中创建问题。

注意

虽然Activiti已经托管在GitHub上了，但是问题不应该提交到GitHub的问题跟踪系统上。如果你想报告一个问题，不要创建一个GitHub的问题，而是应该使用JIRA [[http://jira.codehaus.org/browse/ACT](#)]。

1. 6. #试验性功能

那些标记着 [EXPERIMENTAL] 的章节表示功能尚未稳定。

所有包名中包含`.impl.`的类都是内部实现类，都是不保证稳定的。不过，如果用户指南把哪些类列为配置项，那么它们可以认为是稳定不变的。

1. 7. #内部实现类

在jar包中，所有包名中包含`.impl.`（比如：`org.activiti.engine.impl.pvm.delegate`）的类都是实现类，它们应该被视为流程引擎内部的类。对于这些类和接口都不能够保证其稳定性。

第#2#章#开始学习

2. 1. #一分钟入门

从Activiti网站 [<http://www.activiti.org>] 下载Activiti Explorer的WAR文件后， 可以按照下列步骤以默认配置运行样例。 你需要一个Java 运行环境 [<http://java.sun.com/javase/downloads/index.jsp>] 和 Apache Tomcat [<http://tomcat.apache.org/download-70.cgi>] （其实，任何提供了servlet功能的web容器都可以正常运行。但是我们主要是使用tomcat进行的测试）。

- 把下载的activiti-explorer.war复制到Tomcat的webapps目录下。
- 执行Tomcat的bin目录下的startup.bat或startup.sh启动服务器。
- Tomcat启动后，打开浏览器访问<http://localhost:8080/activiti-explorer>。 使用kermit/kermit登录。

这样就好了！Activiti Explorer默认使用H2内存数据库，如果你想使用其他数据库 请参考这里。

2. 2. #安装Activiti

要安装Activiti你需要一个 Java运行环境 [<http://java.sun.com/javase/downloads/index.jsp>] 和 Apache Tomcat [<http://tomcat.apache.org/download-70.cgi>]。还要确认设置好JAVA_HOME系统变量。不同的操作系统下的设置方法是不同的。

要运行Activiti Explorer和REST web应用，你要从Activiti的下载页下载WAR文件， 复制到Tomcat安装目录下webapps目录下。 默认Explorer应用使用的内存数据库已经包含了示例流程，用户和群组信息。

下面是示例中可以使用的用户：

表 2.1. 示例用户

账号	密码	角色
kermit	kermit	admin
gonzo	gonzo	manager
fozzie	fizzie	user

现在，你可以访问下列web应用：

表 2.2. webapp工具

Webapp名称	URL	描述
Activiti Explorer	http://localhost:8080/activiti-explorer	流程引擎的用户控制台。使用它来启动新流程，分配任务，查看并认领任务，等等。这个

Webapp名称	URL	描述	
		工具也可以用来管理Activiti引擎。	

注意Activiti Explorer演示实例只是一种简单快速展示Activiti的功能的方式。但是并不是说只能使用这种方式使用Activiti。Activiti只是一个jar，可以内嵌到任何Java环境中：swing或者Tomcat，JBoss，WebSphere等等。也可以把Activiti作为一个典型的单独运行的BPM服务器运行。只要java可以做的，Activiti也可以。

2. 3. #安装Activiti数据库

就像在一分钟入门里说过的，Activiti Explorer默认使用H2内存数据库。要让Activiti使用独立运行的H2数据库或者其他数据库，可以修改Activiti Explorer web应用WEB-INF/classes目录下的db.properties。

另外，注意Activiti Explorer自动生成了演示用的默认用户和群组，流程定义，数据模型。要想禁用这个功能，要修改WEB-INF目录下的activiti-standalone-context.xml。可以使用下面的demoDataGenerator bean定义代码完全禁用安装默认数据。从代码中也可以看出，我们可以单独启用或禁用每一项功能。

```
<bean id="demoDataGenerator" class="org.activiti.explorer.demo.DemoDataGenerator">
  <property name="processEngine" ref="processEngine" />
  <property name="createDemoUsersAndGroups" value="false" />
  <property name="createDemoProcessDefinitions" value="false" />
  <property name="createDemoModels" value="false" />
</bean>
```

2. 4. #引入Activiti jar和依赖

为了引用Activiti jar和依赖，我们推荐使用 Maven [<http://maven.apache.org/>]（或Ivy [<http://ant.apache.org/ivy/>]），它简化了我们之间的依赖管理。参考<http://www.activiti.org/community.html#maven.repository> 来为你的项目引入必须的jar包。

如果不使用Maven，你也可以自己把这些jar引入到你的项目中。Activiti下载zip包包含了一个libs目录，包含了所有Activiti的jar包（和源代码jar包）。依赖没有用这种方式发布。Activiti引擎必须的依赖如下所示（通过mvn dependency:tree生成）：

```
org.activiti:activiti-engine:jar:5.12.1
+- org.apache.commons:commons-email:jar:1.2:compile
|   +- javax.mail:mail:jar:1.4.1:compile
|   \- javax.activation:activation:jar:1.1:compile
+- org.apache.commons:commons-lang3:jar:3.1:compile
+- org.mybatis:mybatis:jar:3.1.1:compile
+- org.springframework:spring-beans:jar:3.1.2.RELEASE:compile
|   \- org.springframework:spring-core:jar:3.1.2.RELEASE:compile
|       +- org.springframework:spring-aspects:jar:3.1.2.RELEASE:compile
|           \- commons-logging:commons-logging:jar:1.1.1:compile
\- joda-time:joda-time:jar:2.1:compile
```

注意：只有使用了mail service task才必须引入mail依赖jar。

所有依赖可以在Activiti 源码 [<https://github.com/Activiti/Activiti>]的模块中， 通过mvn dependency:copy-dependencies下载。

2. 5. #下一步

使用Activiti Explorer web应用 是一个熟悉Activiti概念和功能的好办法。但是，Activiti的主要目标是为你自己的应用添加强大的BPM和工作流功能。 下面的章节会帮助你熟悉 如何在你的环境中使用Activiti进行编程：

- 配置章节 会教你如何设置Activiti， 如何获得ProcessEngine类的实例， 它是所有Activiti引擎功能的中心入口。
- API章节会带领你了解建立Activiti API的服务。 这些服务用简便的方法提供了Activiti引擎的强大功能， 它们可以使用在任何Java环境下。
- 对深入了解BPMN 2.0， Activiti引擎中流程的编写结构感兴趣吗？ 请继续浏览BPMN 2.0 章节。

第#3#章#配置

3. 1. #创建ProcessEngine

Activiti流程引擎的配置文件是名为activiti.cfg.xml的XML文件。 注意这与使用Spring方式创建流程引擎 是不一样的。

获得ProcessEngine最简单的办法是 使用org.activiti.engine.ProcessEngines类：

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

它会在classpath下搜索activiti.cfg.xml，并基于这个文件中的配置构建引擎。 下面代码展示了实例配置。 后面的章节会给出配置参数的详细介绍。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/beans.xsd">
    <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
        <property name="jdbcUrl" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
        <property name="jdbcDriver" value="org.h2.Driver" />
        <property name="jdbcUsername" value="sa" />
        <property name="jdbcPassword" value="" />
        <property name="databaseSchemaUpdate" value="true" />
        <property name="jobExecutorActivate" value="false" />
        <property name="mailServerHost" value="mail.my-corp.com" />
        <property name="mailServerPort" value="5025" />
    
</beans>
```

注意配置XML文件其实是一个spring的配置文件。 但不是说Activiti只能用在Spring环境中！ 我们只是利用了Spring的解析和依赖注入功能 来构建引擎。

配置文件中使用的ProcessEngineConfiguration可以通过编程方式创建。 可以配置不同的bean id（比如，第三行）。

```
ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault();
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource);
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource, String beanName);
ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream);
ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream, String beanName);
```

也可以不使用配置文件，基于默认创建配置（参考各种支持类）

```
ProcessEngineConfiguration.createStandaloneProcessEngineConfiguration();
ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration();
```

所有这些ProcessEngineConfiguration.createXXX()方法都返回 ProcessEngineConfiguration，后续可以调整成所需的对象。 在调用buildProcessEngine()后， 就会创建一个ProcessEngine：

```

ProcessEngine processEngine = ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration()
    .setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB_SCHEMA_UPDATE_FALSE)
    .setJdbcUrl("jdbc:h2:mem:my-own-db;DB_CLOSE_DELAY=1000")
    .setJobExecutorActivate(true)
    .buildProcessEngine();

```

3. 2. #ProcessEngineConfiguration bean

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

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

这个bean会用来构建ProcessEngine。 有多个类可以用来定义processEngineConfiguration。 这些类对应不同的环境，并设置了对应的默认值。 最好选择（最）适用于你的环境的类， 这样可以少配置几个引擎的参数。 下面是目前可以使用的类（以后会包含更多）：

- org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration: 单独运行的流程引擎。Activiti会自己处理事务。 默认，数据库只在引擎启动时检测（如果没有Activiti的表或者表结构不正确就会抛出异常）。
- org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration: 单元测试时的辅助类。Activiti会自己控制事务。 默认使用H2内存数据库。数据库表会在引擎启动时创建，关闭时删除。 使用它时，不需要其他配置（除非使用job执行器或邮件功能）。
- org.activiti.spring.SpringProcessEngineConfiguration: 在Spring环境下使用流程引擎。 参考Spring集成章节。
- org.activiti.engine.impl.cfg.JtaProcessEngineConfiguration: 单独运行流程引擎，并使用JTA事务。

3. 3. #数据库配置

Activiti可能使用两种方式配置数据库。 第一种方式是定义数据库配置参数：

- jdbcUrl: 数据库的JDBC URL。
- jdbcDriver: 对应不同数据库类型的驱动。
- jdbcUsername: 连接数据库的用户名。
- jdbcPassword: 连接数据库的密码。

基于JDBC参数配置的数据库连接 会使用默认的MyBatis [http://www.mybatis.org/]连接池。 下面的参数可以用来配置连接池（来自MyBatis参数）：

- jdbcMaxActiveConnections: 连接池中处于被使用状态的连接的最大值。默认为10。
- jdbcMaxIdleConnections: 连接池中处于空闲状态的连接的最大值。
- jdbcMaxCheckoutTime: 连接被取出使用的最长时间，超过时间会被强制回收。 默认为20000（20秒）。

- `jdbcMaxWaitTime`: 这是一个底层配置，让连接池可以在长时间无法获得连接时，打印一条日志，并重新尝试获取一个连接。（避免因为错误配置导致沉默的操作失败）。默认为20000（20秒）。

示例数据库配置：

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

也可以使用`javax.sql.DataSource`。（比如，Apache Commons [http://commons.apache.org/dbcp/]的DBCP）：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" >
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/activiti" />
    <property name="username" value="activiti" />
    <property name="password" value="activiti" />
    <property name="defaultAutoCommit" value="false" />
</bean>

<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <property name="dataSource" ref="dataSource" />
    ...
</bean>
```

注意，Activiti的发布包中没有这些类。你要自己把对应的类（比如，从DBCP里）放到你的classpath下。

无论你使用JDBC还是DataSource的方式，都可以设置下面的配置：

- `databaseType`: 一般不用设置，因为可以自动通过数据库连接的元数据获取。只有自动检测失败时才需要设置。可能的值有：{h2, mysql, oracle, postgres, mssql, db2}。如果没使用默认的H2数据库就必须设置这项。这个配置会决定使用哪些创建/删除脚本和查询语句。参考支持数据库章节 了解支持哪些类型。
- `databaseSchemaUpdate`: 设置流程引擎启动和关闭时如何处理数据库表。
 - `false`（默认）：检查数据库表的版本和依赖库的版本，如果版本不匹配就抛出异常。
 - `true`：构建流程引擎时，执行检查，如果需要就执行更新。如果表不存在，就创建。
 - `create-drop`：构建流程引擎时创建数据库表，关闭流程引擎时删除这些表。

3. 4. #JNDI数据库配置

默认，Activiti的数据库配置会放在web应用的WEB-INF/classes目录下的db.properties文件中。这样做比较繁琐，因为要用户在每次发布时，都修改Activiti源码中的db.properties并重新编译war文件，或者解压缩war文件，修改其中的db.properties。

使用JNDI（Java命名和目录接口）来获取数据库连接，连接是由servlet容器管理的，可以在war部署外边管理配置。与db.properties相比，它也允许对连接进行更多的配置。

3. 4. 1. #使用

要想把Activiti Explorer和Activiti Rest应用从db.properties转换为使用 JNDI数据库配置，需要打开原始的Spring配置文件（activiti-webapp-explorer2/src/main/webapp/WEB-INF/activiti-standalone-context.xml 和activiti-webapp-rest2/src/main/resources/activiti-context.xml），删除“dbProperties”和“dataSource”两个bean，然后添加如下bean：

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

接下来，我们需要添加包含了默认的H2配置的context.xml文件。如果已经有了JNDI配置，会覆盖这些配置。对Activiti Explorer来说，对应的配置文件activiti-webapp-explorer2/src/main/webapp/META-INF/context.xml 如下所示：

```
<Context antiJARLocking="true" path="/activiti-explorer2">
    <Resource auth="Container"
              name="jdbc/activitiDB"
              type="javax.sql.DataSource"
              scope="Shareable"
              description="JDBC DataSource"
              url="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000"
              driverClassName="org.h2.Driver"
              username="sa"
              password=""
              defaultAutoCommit="false"
              initialSize="5"
              maxWait="5000"
              maxActive="120"
              maxIdle="5"/>
</Context>
```

对于Activiti REST应用，添加的activiti-webapp-rest2/src/main/webapp/META-INF/context.xml 如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/activiti-rest2">
    <Resource auth="Container"
              name="jdbc/activitiDB"
              type="javax.sql.DataSource"
              scope="Shareable"
              description="JDBC DataSource"
              url="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=-1"
              driverClassName="org.h2.Driver"
              username="sa"
              password=""
              defaultAutoCommit="false"
              initialSize="5"
              maxWait="5000"
              maxActive="120"
              maxIdle="5"/>
</Context>
```

可选的一步，现在可以删除Activiti Explorer和Activiti Rest两个应用中 不再使用的db.properties文件了。

3. 4. 2. #配置

JNDI数据库配置会因为你使用的Servlet container不同而不同。 下面的配置可以在tomcat中使用，但是对其他容易，请引用你使用的容器的文档。

如果使用tomcat， JNDI资源配置在 \$CATALINA_BASE/conf/[enginename]/[hostname]/[warnname].xml （对于Activiti Explorer来说，通常是在\$CATALINA_BASE/conf/Catalina/localhost/activiti-explorer.war）。 当应用第一次发布时，会把这个文件从war中复制出来。 所以如果这个文件已经存在了，你需要替换它。要想修改JNDI资源让应用连接mysql而不是H2， 可以像下面这样修改：

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/activiti-explorer2">
    <Resource auth="Container"
        name="jdbc/activitiDB"
        type="javax.sql.DataSource"
        description="JDBC DataSource"
        url="jdbc:mysql://localhost:3306/activiti"
        driverClassName="com.mysql.jdbc.Driver"
        username="sa"
        password=""
        defaultAutoCommit="false"
        initialSize="5"
        maxWait="5000"
        maxActive="120"
        maxIdle="5"/>
</Context>
```

3. 5. #支持的数据库

下面列出Activiti使用的数据库类型（大小写敏感）。

表 3.1. 支持的数据库

Activiti数据库类型	JDBC URL实例	备注
h2	jdbc:h2:tcp://localhost/activiti	默认配置的数据库
mysql	jdbc:mysql://localhost:3306/activiti?autoReconnect=true	使用mysql-connector-java驱动 测试
oracle	jdbc:oracle:thin:@localhost:1521:xe	
postgres	jdbc:postgresql://localhost:5432/activiti	
db2	jdbc:db2://localhost:50000/activiti	

Activiti数据库类型	JDBC URL实例	备注
mssql	jdbc:sqlserver://localhost:1433/activiti	

3. 6. #创建数据库表

下面是创建数据库表最简单的办法：

- 把activiti-engine的jar放到classpath下
- 添加对应的数据库驱动
- 把Activiti配置文件 (activiti.cfg.xml) 放到 classpath下， 指向你的数据库（参考数据库配置章节）
- 执行 DbSchemaCreate 类的main方法

不过，一般情况只有数据库管理员才能执行DDL语句。 在生产环境，这也是最明智的选择。 SQL DDL语句可以从Activiti下载页或Activiti发布目录里找到，在database子目录下。 脚本也包含在引擎的jar中(activiti-engine-x.jar)， 在org/activiti/db/create包下 (drop目录里是删除语句)。 SQL文件的命名方式如下

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

其中 db 是 支持的数据库， type 是

- engine: 引擎执行的表。必须。
- identity: 包含用户，群组，用户与组之间的关系的表。 这些表是可选的，只有使用引擎自带的默认身份管理时才需要。
- history: 包含历史和审计信息的表。可选的：历史级别设为none时不会使用。 注意这也引用一些需要把数据保存到历史表中的功能（比如任务的评论）。

MySQL用户需要注意： 版本低于5.6.4的MySQL不支持毫秒精度的timestamp或date类型。 更严重的是，有些版本会在尝试创建这样一列时抛出异常，而有些版本则不会。 在执行自动创建/更新时，引擎会在执行过程中修改DDL。 当使用DDL时，可以选择通用版本和名为mysql55的文件。（它适合所有版本低于5.6.4的情况）。 后一个文件会将列的类型设置为没有毫秒的情况。

总结一下，对于MySQL版本会执行如下操作

- <5.6: 不支持毫秒精度。可以使用DDL文件（包含mysql55的文件）。可以实现自动创建/更新。
- 5.6.0 – 5.6.3: 不支持毫秒精度。无法自动创建/更新。建议更新到新的数据库版本。如果真的需要的话，也可以使用mysql 5.5。
- 5.6.4+: 支持毫秒精度。可以使用DDL文件（默认包含mysql的文件）。可以实现自动创建、更新。

注意对于已经更新了MySQL数据库，而且Activiti表已经创建/更新的情况， 必须手工修改列的类型。

3. 7. #理解数据库表的命名

Activiti的表都以ACT_开头。 第二部分是表示表的用途的两个字母标识。 用途也和服务的API对应。

- ACT_RE_*: 'RE' 表示repository。 这个前缀的表包含了流程定义和流程静态资源（图片，规则，等等）。
- ACT_RU_*: 'RU' 表示runtime。 这些运行时的表，包含流程实例，任务，变量，异步任务，等运行中的数据。 Activiti只在流程实例执行过程中保存这些数据，在流程结束时就会删除这些记录。 这样运行时表可以一直很小速度很快。
- ACT_ID_*: 'ID' 表示identity。 这些表包含身份信息，比如用户，组等等。
- ACT_HI_*: 'HI' 表示history。 这些表包含历史数据，比如历史流程实例，变量，任务等等。
- ACT_GE_*: 通用数据，用于不同场景下。

3. 8. #数据库升级

在执行更新之前要先备份数据库（使用数据库的备份功能）

默认，每次构建流程引擎时都会进行版本检测。这一切都在应用启动或Activiti webapp启动时发生。如果Activiti发现数据库表的版本与依赖库的版本不同，就会抛出异常。

要升级，你要把下面的配置放到activiti.cfg.xml配置文件里：

```
<beans ... >

<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <!-- ... -->
    <property name="databaseSchemaUpdate" value="true" />
    <!-- ... -->
</bean>

</beans>
```

然后，把对应的数据库驱动放到classpath里。升级应用的Activiti依赖。启动一个新版本的Activiti 指向包含旧版本的数据库。将databaseSchemaUpdate设置为true，Activiti会自动将数据库表升级到新版本，当发现依赖和数据库表版本不通过时。

也可以执行更新升级DDL语句。也可以执行数据库脚本，可以在Activiti下载页找到。

3. 9. #启用Job执行器

JobExecutor是管理一系列线程的组件，可以触发定时器（也包含后续的异步消息）。在单元测试场景下，很难使用多线程。因此API允许查询(ManagementService.createJobQuery)和执行job(ManagementService.executeJob)，所以job可以在单元测试中控制。要避免与job执行器冲突，可以关闭它。

默认，JobExecutor在流程引擎启动时就会激活。
活JobExecutor，可以设置

如果不想在流程引擎启动后自动激

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

3. 10. #配置邮件服务器

可以选择配置邮件服务器。Activiti支持在业务流程中发送邮件。想真正的发送一个email，必须配置一个真实的SMTP邮件服务器。参考e-mail任务。

3. 11. #配置历史

可以选择定制历史存储的配置。你可以通过配置影响引擎的历史功能。参考历史配置。

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

3. 12. #为表达式和脚本暴露配置

默认，activiti.cfg.xml和你自己的Spring配置文件中所有bean都可以在表达式和脚本中使用。如果你想限制配置文件中的bean的可见性，可以配置流程引擎配置的beans配置。ProcessEngineConfiguration的beans是一个map。当你指定了这个参数，只有包含这个map中的bean可以在表达式和脚本中使用。通过在map中指定的名称来决定暴露的bean。

3. 13. #配置部署缓存

所有流程定义都被缓存了（解析之后）避免每次使用前都要访问数据库，因为流程定义数据是不会改变的。默认，不会限制这个缓存。如果想限制流程定义缓存，可以添加如下配置

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

这个配置会把默认的hashmap缓存替换成LRU缓存，来提供限制。当然，这个配置的最佳值跟流程定义的总数有关，实际使用中会具体使用多少流程定义也有关。

也可以注入自己的缓存实现。这个bean必须实现org.activiti.engine.impl.persistence.deploy.DeploymentCache接口：

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

有一个类似的配置叫knowledgeBaseCacheLimit和knowledgeBaseCache，它们是配置规则缓存的。只有流程中使用规则任务时才会用到。

3. 14. #日志

从Activiti 5.12开始，SLF4J被用作日志框架，替换了之前使用java.util.logging。所有日志（activiti, spring, mybatis等等）都转发给SLF4J 允许使用你选择的日志实现。

默认activiti-engine依赖中没有提供SLF4J绑定的jar，需要根据你的实际需要使用日志框架。如果没有添加任何实现jar，SLF4J会使用NOP-logger，不使用任何日志，不会发出警告，而且什么日志都不会记录。可以通过<http://www.slf4j.org/codes.html#StaticLoggerBinder>了解这些实现。

使用Maven，比如使用一个依赖（这里使用log4j），注意你还需要添加一个version：

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

activiti-explorer和activiti-rest应用都使用了Log4j绑定。执行所有activiti-*模块的单元测试页使用了Log4j。

特别提醒如果容器classpath中存在commons-logging：为了把spring日志转发给SLF4J，需要使用桥接（参考<http://www.slf4j.org/legacy.html#jc1OverSLF4J>）。如果你的容器提供了commons-logging实现，请参考下面网页：<http://www.slf4j.org/codes.html#release>来确保稳定性。

使用Maven的实例（忽略版本）：

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
</dependency>
```

3. 15. #映射诊断上下文

在5.13中，activiti支持slf4j的MDC功能。如下的基础信息会传递到日志中记录：

- 流程定义Id标记为mdcProcessDefinitionID
- 流程实例Id标记为mdcProcessInstanceId
- 分支Id标记为mdcExecutionId

默认不会记录这些信息。可以配置日志使用期望的格式来显示它们，扩展通常的日志信息。比如，下面的log4j配置定义会让日志显示上面提及的信息：

```
log4j.appender.consoleAppender.layout.ConversionPattern =ProcessDefinitionId=%X{mdcProcessDefinitionID}
executionId=%X{mdcExecutionId} mdcProcessInstanceId=%X{mdcProcessInstanceId} mdcBusinessKey=%X{mdcBusinessKey}
```

当系统进行高风险任务，日志必须严格检查时，这个功能就非常有用，比如要使用日志分析的情况。

3. 16. #事件处理

Activiti 5.15中实现了一种事件机制。它允许在引擎触发事件时获得提醒。参考所有支持的事件类型了解有效的事件。

可以为对应的事件类型注册监听器，在这个类型的任何时间触发时都会收到提醒。你可以添加引擎范围的事件监听器通过配置，添加引擎范围的事件监听器在运行阶段使用API，或添加event-listener到特定流程定义的BPMN XML中。

所有分发的事件，都是org.activiti.engine.delegate.event.ActivitiEvent的子类。事件包含（如果有效）type，executionId，processInstanceId和processDefinitionId。对应的事件会包含事件发生时对应上下文的额外信息，这些额外的载荷可以在支持的所有事件类型中找到。

3.16.1. #事件监听器实现

实现事件监听器的唯一要求是实现org.activiti.engine.delegate.event.ActivitiEventListener。下面是一个实现监听器的例子，它会把所有监听到的事件打印到标准输出中，包括job执行的事件异常：

```
public class MyEventListener implements ActivitiEventListener {

    @Override
    public void onEvent(ActivitiEvent event) {
        switch (event.getType()) {

            case JOB_EXECUTION_SUCCESS:
                System.out.println("A job well done!");
                break;

            case JOB_EXECUTION_FAILURE:
                System.out.println("A job has failed...");
                break;

            default:
                System.out.println("Event received: " + event.getType());
        }
    }

    @Override
    public boolean isFailOnException() {
        // The logic in the onEvent method of this listener is not critical, exceptions
        // can be ignored if logging fails...
        return false;
    }
}
```

isFailOnException()方法决定了当事件分发时，onEvent(..)方法抛出异常时的行为。这里返回的是false，会忽略异常。当返回true时，异常不会忽略，继续向上传播，迅速导致当前命令失败。当事件是一个API调用的一部分时（或其他事务性操作，比如job执行），事务就会回滚。当事件监听器中的行为不是业务性时，建议返回false。

activiti提供了一些基础的实现，实现了事件监听器的常用场景。可以用来作为基类或监听器实现的样例：

- org.activiti.engine.delegate.event BaseEntityEventListener：这个事件监听器的基类可以用来监听实体相关的事件，可以针对某一类型实体，也可以是全部实体。它隐藏了类型检测，并提供了三个需要重写的方法：onCreate(..)，onUpdate(..) 和 onDelete(..)，当实体创建，更新，或删除时调用。对于其他实体相关的事件，会调用 onEntityEvent(..)。

3.16.2. #配置与安装

把事件监听器配置到流程引擎配置中时，会在流程引擎启动时激活，并在引擎启动启动中持续工作着。

eventListeners属性需要org.activiti.engine.delegate.event.ActivitiEventListener的队列。通常，我们可以声明一个内部的bean定义，或使用ref引用已定义的bean。下面的代码，向配置添加了一个事件监听器，任何事件触发时都会提醒它，无论事件是什么类型：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration"
  ...
  <property name="eventListeners">
    <list>
      <bean class="org.activiti.engine.example.MyEventListener" />
    </list>
  </property>
</bean>
```

为了监听特定类型的事件，可以使用`typedEventListeners`属性，它需要一个`map`参数。
 map的key是逗号分隔的事件名（或单独的事件名）。
 map的value
 是`org.activiti.engine.delegate.event.ActivitiEventListener`队列。下面的代码演示了向配置中添加一个事件监听器，可以监听job执行成功或失败：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration"
  ...
  <property name="typedEventListeners">
    <map>
      <entry key="JOB_EXECUTION_SUCCESS, JOB_EXECUTION_FAILURE" >
        <list>
          <bean class="org.activiti.engine.example.MyJobEventListener" />
        </list>
      </entry>
    </map>
  </property>
</bean>
```

分发事件的顺序是由监听器添加时的顺序决定的。首先，会调用所有普通的事件监听器（`eventListeners`属性），按照它们在`list`中的次序。然后，会调用所有对应类型的监听器（`typedEventListeners`属性），如果对应类型的事件被触发了。

3.16.3. #在运行阶段添加监听器

可以通过API（`RuntimeService`）在运行阶段添加或删除额外的事件监听器：

```
/**
 * Adds an event-listener which will be notified of ALL events by the dispatcher.
 * @param listenerToAdd the listener to add
 */
void addEventListener(ActivitiEventListener listenerToAdd);

/**
 * Adds an event-listener which will only be notified when an event occurs, which type is in the given types.
 * @param listenerToAdd the listener to add
 * @param types types of events the listener should be notified for
 */
void addEventListener(ActivitiEventListener listenerToAdd, ActivitiEventType... types);

/**
 * Removes the given listener from this dispatcher. The listener will no longer be notified,
 * regardless of the type(s) it was registered for in the first place.
 * @param listenerToRemove listener to remove
 */
void removeEventListener(ActivitiEventListener listenerToRemove);
```

注意运行期添加的监听器引擎重启后就消失了。

3.16.4. #为流程定义添加监听器

可以为特定流程定义添加监听器。监听器只会监听与这个流程定义相关的事件，以及这个流程定义上发起的所有流程实例的事件。监听器实现可以使用，全类名定义，引用实现了监听器接口的表达式，或配置为抛出一个message/signal/error的BPMN事件。

3.16.4.1. #让监听器执行用户定义的逻辑

下面代码为一个流程定义添加了两个监听器。第一个监听器会接收所有类型的事件，它是通过全类名定义的。第二个监听器只接收作业成功或失败的事件，它使用了定义在流程引擎配置中的beans属性中的一个bean。

```
<process id="testEventListeners">
<extensionElements>
    <activiti:eventListener class="org.activiti.engine.test.MyEventListener" />
    <activiti:eventListener delegateExpression="${testEventListener}" events="JOB_EXECUTION_SUCCESS, JOB_EXECUTION_FAILED" />
</extensionElements>

...
</process>
```

对于实体相关的事件，也可以设置为针对某个流程定义的监听器，实现只监听发生在某个流程定义上的某个类型实体事件。下面的代码演示了如何实现这种功能。可以用于所有实体事件（第一个例子），也可以只监听特定类型的事件（第二个例子）。

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

...
</process>
```

entityType支持的值有：attachment, comment, execution, identity-link, job, process-instance, process-definition, task。

3.16.4.2. #监听抛出BPMN事件

[试验阶段]

另一种处理事件的方法是抛出一个BPMN事件。请注意它只针对与抛出一个activiti事件类型的BPMN事件。比如，抛出一个BPMN事件，在流程实例删除时，会导致一个错误。下面的代码演示了如何在流程实例中抛出一个signal，把signal抛出到外部流程（全局），在流程实例中抛出一个消息事件，在流程实例中抛出一个错误事件。除了使用class或delegateExpression，还使用了throwEvent属性，通过额外属性，指定了抛出事件的类型。

```
<process id="testEventListeners">
<extensionElements>
    <activiti:eventListener throwEvent="signal" signalName="My signal" events="TASK_ASSIGNED" />
</extensionElements>
```

```
</process>
```

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener throwEvent="globalSignal" signalName="My signal" events="TASK_ASSIGNED" />
  </extensionElements>
</process>
```

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener throwEvent="message" messageName="My message" events="TASK_ASSIGNED" />
  </extensionElements>
</process>
```

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener throwEvent="error" errorCode="123" events="TASK_ASSIGNED" />
  </extensionElements>
</process>
```

如果需要声明额外的逻辑，是否抛出BPMN事件，可以扩展activiti提供的监听器类。在子类中重写isValidEvent(ActivitiEvent event)，可以防止抛出BPMN事件。对应的类是org.activiti.engine.test.api.event.SignalThrowingEventListenerTest, org.activiti.engine.impl.bpmn.helper.MessageThrowingEventListener 和 org.activiti.engine.impl.bpmn.helper.ErrorThrowingEventListener.

3.16.4.3. #流程定义中监听器的注意事项

- 事件监听器只能声明在process元素中，作为extensionElements的子元素。监听器不能定义在流程的单个activity下。
- delegateExpression中的表达式无法访问execution上下文，这与其他表达式不同（比如 gateway）。它只能引用定义在流程引擎配置的beans属性中声明的bean，或者使用spring（未使用beans属性）中所有实现了监听器接口的spring-bean。
- 在使用监听器的 class 属性时，只会创建一个实例。记住监听器实现不会依赖成员变量，确认是多线程安全的。
- 当一个非法的事件类型用在events属性或throwEvent中时，流程定义发布时就会抛出异常。（会导致部署失败）。如果class或delegateExecution由问题（类不存在，不存在的bean引用，或代理类没有实现监听器接口），会在流程启动时抛出异常（或在第一个有效的流程定义事件被监听器接收时）。所以要保证引用的类正确的放在classpath下，表达式也要引用一个有效的实例。

3.16.5. #通过API分发事件

我们提供了通过API使用事件机制的方法，允许大家触发定义在引擎中的任何自定义事件。建议（不强制）只触发类型为CUSTOM的ActivitiEvents。可以通过RuntimeService触发事件：

```
/** 
 * Dispatches the given event to any listeners that are registered.
 * @param event event to dispatch.
```

```

*
* @throws ActivitiException if an exception occurs when dispatching the event or when the {@link ActivitiEvent}
* is disabled.
* @throws ActivitiIllegalArgumentException when the given event is not suitable for dispatching.
*/
void dispatchEvent(ActivitiEvent event);

```

3.16.6. #支持的事件类型

下面是引擎中可能出现的所有事件类型。每个类型都对应org.activiti.engine.delegate.event.ActivitiEventType中的一个枚举值。

表 3.2. 支持的事件

事件名称	描述	事件类型
ENGINE_CREATED	监听器监听的流程引擎已经创建完毕，并准备好接受API调用。	org.activiti...ActivitiEvent
ENGINE_CLOSED	监听器监听的流程引擎已经关闭，不再接受API调用。	org.activiti...ActivitiEvent
ENTITY_CREATED	创建了一个新实体。实体包含在事件中。	org.activiti...ActivitiEntityEvent
ENTITY_INITIALIZED	创建了一个新实体，初始化也完成了。如果这个实体的创建会包含子实体的创建，这个事件会在子实体都创建/初始化完成后被触发，这是与ENTITY_CREATED的区别。	org.activiti...ActivitiEntityEvent
ENTITY_UPDATED	更新了已存在的实体。实体包含在事件中。	org.activiti...ActivitiEntityEvent
ENTITY_DELETED	删除了已存在的实体。实体包含在事件中。	org.activiti...ActivitiEntityEvent
ENTITY_SUSPENDED	暂停了已存在的实体。实体包含在事件中。会被ProcessDefinitions, ProcessInstances 和 Tasks抛出。	org.activiti...ActivitiEntityEvent
ENTITY_ACTIVATED	激活了已存在的实体，实体包含在事件中。会被ProcessDefinitions, ProcessInstances 和 Tasks抛出。	org.activiti...ActivitiEntityEvent
JOB_EXECUTION_SUCCESS	作业执行成功。job包含在事件中。	org.activiti...ActivitiEntityEvent
JOB_EXECUTION_FAILURE	作业执行失败。作业和异常信息包含在事件中。	org.activiti...ActivitiEntityEvent and org.activiti...ActivitiExceptionEvent

事件名称	描述	事件类型
JOB_RETRIES_DECREMENTED	因为作业执行失败，导致重试次数减少。作业包含在事件中。	org.activiti...ActivitiEntityEvent
TIMER_FIRED	触发了定时器。job包含在事件中。	org.activiti...ActivitiEntityEvent
JOB_CANCELED	取消了一个作业。事件包含取消的作业。作业可以通过API调用取消，任务完成后对应的边界定时器也会取消，在新流程定义发布时也会取消。	org.activiti...ActivitiEntityEvent
ACTIVITY_STARTED	一个节点开始执行	org.activiti...ActivitiActivityEvent
ACTIVITY_COMPLETED	一个节点成功结束	org.activiti...ActivitiActivityEvent
ACTIVITY_SIGNALLED	一个节点收到了一个信号	org.activiti...ActivitiSignalEvent
ACTIVITY_MESSAGE_RECEIVED	一个节点收到了一个消息。在节点收到消息之前触发。收到后，会触发ACTIVITY_SIGNAL或ACTIVITY_STARTED，这会根据节点的类型（边界事件，事件子流程开始事件）	org.activiti...ActivitiMessageEvent
ACTIVITY_ERROR RECEIVED	一个节点收到了一个错误事件。在节点实际处理错误之前触发。事件的activityId对应着处理错误的节点。这个事件后续会是ACTIVITY_SIGNALLED或ACTIVITY_COMPLETE，如果错误发送成功的话。	org.activiti...ActivitiErrorEvent
UNCAUGHT_BPMN_ERROR	抛出了未捕获的BPMN错误。流程没有提供针对这个错误的处理器。事件的activityId为空。	org.activiti...ActivitiErrorEvent
ACTIVITY_COMPENSATE	一个节点将要被补偿。事件包含了将要执行补偿的节点id。	org.activiti...ActivitiActivityEvent
VARIABLE_CREATED	创建了一个变量。事件包含变量名，变量值和对应的分支或任务（如果存在）。	org.activiti...ActivitiVariableEvent
VARIABLE_UPDATED	更新了一个变量。事件包含变量名，变量值和对应的分支或任务（如果存在）。	org.activiti...ActivitiVariableEvent
VARIABLE_DELETED	删除了一个变量。事件包含变量名，变量值和对应的分支或任务（如果存在）。	org.activiti...ActivitiVariableEvent
TASK_ASSIGNED	任务被分配给了一个人员。事件包含任务。	org.activiti...ActivitiEntityEvent

事件名称	描述	事件类型
TASK_CREATED	创建了新任务。它位于ENTITY_CREATE事件之后。当任务是由流程创建时，这个事件会在TaskListener执行之前被执行。	org.activiti...ActivitiEntityEvent
TASK_COMPLETED	任务被完成了。它会在ENTITY_DELETE事件之前触发。当任务是流程一部分时，事件会在流程继续运行之前，后续事件将是ACTIVITY_COMPLETE，对应着完成任务的节点。	org.activiti...ActivitiEntityEvent
TASK_TIMEOUT	任务已超时，在TIMER_FIRED事件之后，会触发用户任务的超时事件，当这个任务分配了一个定时器的时候。	org.activiti...ActivitiEntityEvent
PROCESS_COMPLETED	流程已结束。在最后一个节点的ACTIVITY_COMPLETED事件之后触发。当流程到达的状态，没有任何后续连线时，流程就会结束。	org.activiti...ActivitiEntityEvent
MEMBERSHIP_CREATED	用户被添加到一个组里。事件包含了用户和组的id。	org.activiti...ActivitiMembershipEvent
MEMBERSHIP_DELETED	用户被从一个组中删除。事件包含了用户和组的id。	org.activiti...ActivitiMembershipEvent
MEMBERSHIPS_DELETED	所有成员被从一个组中删除。在成员删除之前触发这个事件，所以他们都是可以访问的。因为性能方面的考虑，不会为每个成员触发单独的MEMBERSHIP_DELETED事件。	org.activiti...ActivitiMembershipEvent

引擎内部所有ENTITY_*事件都是与实体相关的。下面的列表展示了实体事件与实体的对应关系：

- ENTITY_CREATED, ENTITY_INITIALIZED, ENTITY_DELETED: Attachment, Comment, Deployment, Execution, Group, IdentityLink, Job, Model, ProcessDefinition, ProcessInstance, Task, User.
- ENTITY_UPDATED: Attachment, Deployment, Execution, Group, IdentityLink, Job, Model, ProcessDefinition, ProcessInstance, Task, User.
- ENTITY_SUSPENDED, ENTITY_ACTIVATED: ProcessDefinition, ProcessInstance/Execution, Task.

3. 16. 7. #附加信息

只有同一个流程引擎中的事件会发送给对应的监听器。。的那个你有很多引擎 - 在同一个数据库运行 - 事件只会发送给注册到对应引擎的监听器。其他引擎发生的事件不会发送给这个监听器，无论实际上它们运行在同一个或不同的JVM中。

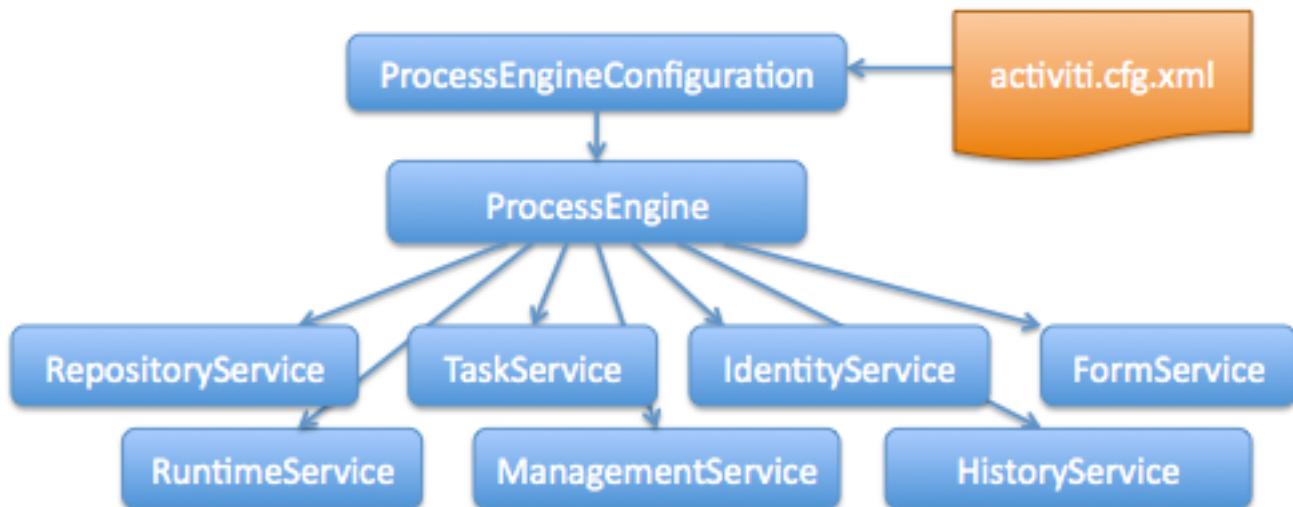
对应的事件类型（对应实体）都包含对应的实体。根据类型或事件，这些实体不能再进行更新（比如，当实例以被删除）。可能的话，使用事件提供的EngineServices来以安全的方式来操作引擎。即使如此，你需要小心的对事件对应的实体进行更新/操作。

没有对应历史的实体事件，因为它们都有运行阶段的对应实体。

第#4#章#Activiti API

4.1. #流程引擎的API和服务

引擎API是与Activiti打交道的最常用方式。 我们从ProcessEngine开始， 创建它的很多种方法都已经在 配置章节中有所涉及。 从ProcessEngine中， 你可以获得很多囊括工作流/BPM方法的服务。 ProcessEngine和服务类都是线程安全的。 你可以在整个服务器中仅保持它们的一个引用就可以了。



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

RuntimeService runtimeService = processEngine.getRuntimeService();
RepositoryService repositoryService = processEngine.getRepositoryService();
TaskService taskService = processEngine.getTaskService();
ManagementService managementService = processEngine.getManagementService();
IdentityService identityService = processEngine.getIdentityService();
HistoryService historyService = processEngine.getHistoryService();
FormService formService = processEngine.getFormService();
```

`ProcessEngines.getDefaultProcessEngine()`会在第一次调用时 初始化并创建一个流程引擎，以后再调用就会返回相同的流程引擎。 使用对应的方法可以创建和关闭所有流程引擎: `ProcessEngines.init()` 和 `ProcessEngines.destroy()`。

ProcessEngines会扫描所有`activiti.cfg.xml` 和 `activiti-context.xml` 文件。 对于`activiti.cfg.xml`文件， 流程引擎会使用Activiti的经典方式构建:

`ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(inputStream).buildProcessEngine()`。对于`activiti-context.xml`文件， 流程引擎会使用Spring方法构建: 先创建一个Spring的环境，然后通过环境获得流程引擎。

所有服务都是无状态的。这意味着可以在多节点集群环境下运行Activiti， 每个节点都指向同一个数据库， 不用担心哪个机器实际执行前端的调用。 无论在哪里执行服务都没有问题。

RepositoryService可能是使用Activiti引擎时最先接触的服务。 它提供了管理和控制发布包和流程定义的操作。 这里不涉及太多细节， 流程定义是BPMN 2.0流程的java实现。 它包含了一个流程每个环节的结构和行为。 发布包是Activiti引擎的打包单位。 一个发布包可以包含多个BPMN 2.0 xml文件和其他资源。 开发者可以自由选择把任意资源包含到发布包

中。既可以把一个单独的BPMN 2.0 xml文件放到发布包里，也可以把整个流程和相关资源都放在一起。（比如，'hr-processes'实例可以包含hr流程相关的任何资源）。可以通过RepositoryService来部署这种发布包。发布一个发布包，意味着把它上传到引擎中，所有流程都会在保存进数据库之前分析解析好。从这点来说，系统知道这个发布包的存在，发布包中包含的流程就已经可以启动了。

除此之外，服务可以

- 查询引擎中的发布包和流程定义。
- 暂停或激活发布包，对应全部和特定流程定义。暂停意味着它们不能再执行任何操作了，激活是对应的反向操作。
- 获得多种资源，像是包含在发布包里的文件，或引擎自动生成的流程图。
- 获得流程定义的pojo版本，可以用来通过java解析流程，而不必通过xml。

正如RepositoryService负责静态信息（比如，不会改变的数据，至少是不怎么改变的），RuntimeService正好是完全相反的。它负责启动一个流程定义的新实例。如上所述，流程定义定义了流程各个节点的结构和行为。流程实例就是这样一个流程定义的实例。对每个流程定义来说，同一时间会有很多实例在执行。RuntimeService也可以用来获取和保存流程变量。这些数据是特定于某个流程实例的，并会被很多流程中的节点使用（比如，一个排他网关常常使用流程变量来决定选择哪条路径继续流程）。Runtimeservice也能查询流程实例和执行。执行对应BPMN 2.0中的'token'。基本上执行指向流程实例当前在哪里。最后，RuntimeService可以在流程实例等待外部触发时使用，这时可以用来继续流程实例。流程实例可以有很多暂停状态，而服务提供了多种方法来'触发'实例，接受外部触发后，流程实例就会继续向下执行。

任务是由系统中真实人员执行的，它是Activiti这类BPMN引擎的核心功能之一。所有与任务有关的功能都包含在TaskService中：

- 查询分配给用户或组的任务
- 创建独立运行任务。这些任务与流程实例无关。
- 手工设置任务的执行者，或者这些用户通过何种方式与任务关联。
- 认领并完成一个任务。认领意味着一个人期望成为任务的执行者，即这个用户会完成这个任务。完成意味着“做这个任务要求的事情”。通常来说会有很多种处理形式。

IdentityService非常简单。它可以管理（创建，更新，删除，查询...）群组和用户。请注意，Activiti执行时并没有对用户进行检查。例如，任务可以分配给任何人，但是引擎不会校验系统中是否存在这个用户。这是Activiti引擎也可以使用外部服务，比如ldap，活动目录，等等。

FormService是一个可选服务。即使不使用它，Activiti也可以完美运行，不会损失任何功能。这个服务提供了启动表单和任务表单两个概念。启动表单会在流程实例启动之前展示给用户，任务表单会在用户完成任务时展示。Activiti支持在BPMN 2.0流程定义中设置这些表单。这个服务以一种简单的方式将数据暴露出来。再次重申，它是可选的，表单也不一定要嵌入到流程定义中。

HistoryService提供了Activiti引擎的所有历史数据。在执行流程时，引擎会保存很多数据（根据配置），比如流程实例启动时间，任务的参与者，完成任务的时间，每个流程实例的执行路径，等等。这个服务主要通过查询功能来获得这些数据。

ManagementService在使用Activiti的定制环境中基本上不会用到。 它可以查询数据库的表和表的元数据。 另外，它提供了查询和管理异步操作的功能。 Activiti的异步操作用途很多，比如定时器，异步操作， 延迟暂停、激活，等等。 后续，会讨论这些功能的更多细节。

可以从javadocs [.. / javadocs/index.html]中获得这些服务和引擎API的更多信息。

4. 2. #异常策略

Activiti中的基础异常为org.activiti.engine.ActivitiException，一个非检查异常。 这个异常可以在任何时候被API抛出，不过特定方法抛出的“特定”的异常都记录在 javadocs [.. / javadocs/index.html]中。 例如，下面的TaskService：

```
/**  
 * Called when the task is successfully executed.  
 * @param taskId the id of the task to complete, cannot be null.  
 * @throws ActivitiObjectNotFoundException when no task exists with the given id.  
 */  
void complete(String taskId);
```

在上面的例子中，当传入一个不存在的任务的id时，就会抛出异常。 同时，javadoc明确指出taskId不能为null，如果传入null， 就会抛出ActivitiIllegalArgumentException。

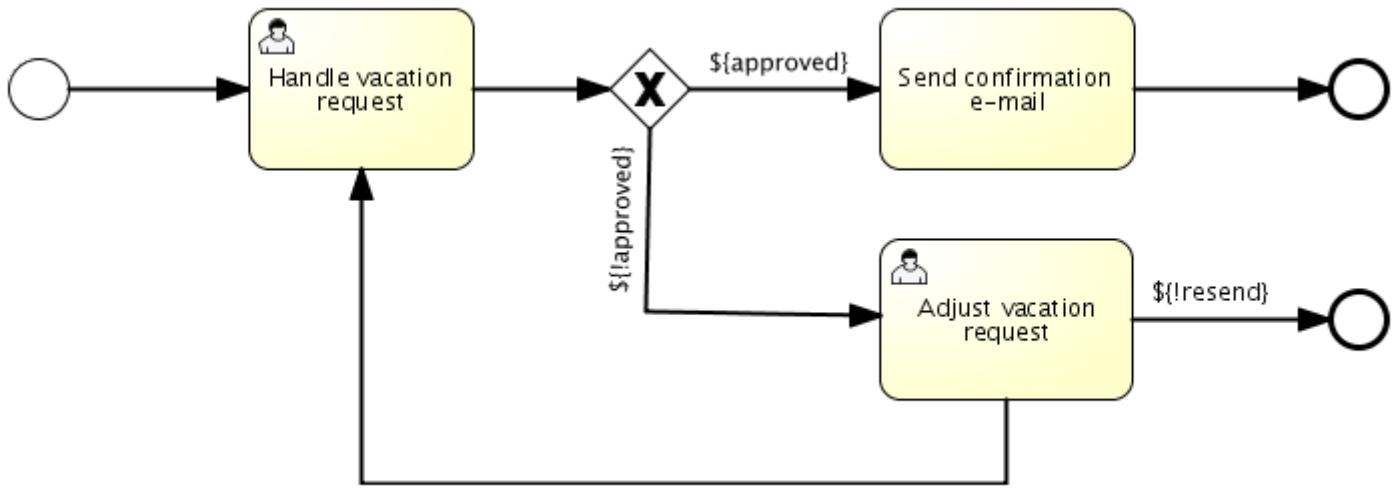
我们希望避免过多的异常继承，下面的子类用于特定的场合。 流程引擎和API调用的其他场合不会使用下面的异常， 它们会抛出一个普通的ActivitiExceptions。

- ActivitiWrongDbException：当Activiti引擎发现数据库版本号和引擎版本号不一致时抛出。
- ActivitiOptimisticLockingException：对同一数据进行并发方法并出现乐观锁时抛出。
- ActivitiClassNotFoundException：当无法找到需要加载的类或在加载类时出现了错误（比如，JavaDelegate，TaskListener等）。
- ActivitiObjectNotFoundException：当请求或操作的对应不存在时抛出。
- ActivitiIllegalArgumentException：这个异常表示调用Activiti API时传入了一个非法的参数，可能是引擎配置中的非法值，或提供了一个非法制，或流程定义中使用的非法值。
- ActivitiTaskAlreadyClaimedException：当任务已经被认领了，再调用taskService.claim(...)就会抛出。

4. 3. #使用Activiti的服务

像上面介绍的那样，要想操作Activiti引擎，需要通过 org.activiti.engine.ProcessEngine实例暴露的服务。 下面的代码假设你已经拥有了一个可以运行的Activiti环境。 你就就可以操作一个org.activiti.engine.ProcessEngine。 如果只想简单尝试一下代码， 可以下载或者cloneActiviti单元测试模板 [https://github.com/Activiti/activiti-unit-test-template]， 导入到IDE中，把testUserguideCode()方法添加到 org.activiti.MyUnitTest中。

这个小例子的最终目标是做一个工作业务流程， 演示公司中简单的请假申请：



4.3.1. #发布流程

任何与“静态”资源有关的数据（比如流程定义）都可以通过 RepositoryService访问。从概念上讲，所以静态数据都是Activiti的资源内容。

在src/test/resources/org/activiti/test目录下创建一个新的xml文件

VacationRequest.bpmn20.xml（如果不使用单元测试模板，你也可以在任何地方创建），内容如下。注意这一章不会解释例子中使用的xml结构。如果有需要可以先阅读bpmn 2.0章来了解这些。

```

<?xml version="1.0" encoding="UTF-8" ?>
<definitions id="definitions"
  targetNamespace="http://activiti.org/bpmn20"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:activiti="http://activiti.org/bpmn">

  <process id="vacationRequest" name="Vacation request">

    <startEvent id="request" activiti:initiator="employeeName">
      <extensionElements>
        <activiti:formProperty id="numberOfDays" name="Number of days" type="long" value="1" required="true"/>
        <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyyy)" datePattern="dd-MM-yyyy hh:mm:ss" value="2010-01-01T00:00:00" />
        <activiti:formProperty id="vacationMotivation" name="Motivation" type="string" />
      </extensionElements>
    </startEvent>
    <sequenceFlow id="flow1" sourceRef="request" targetRef="handleRequest" />

    <userTask id="handleRequest" name="Handle vacation request" >
      <documentation>
        ${employeeName} would like to take ${numberOfDays} day(s) of vacation (Motivation: ${vacationMotivation})
      </documentation>
      <extensionElements>
        <activiti:formProperty id="vacationApproved" name="Do you approve this vacation" type="enum" required="true">
          <activiti:value id="true" name="Approve" />
          <activiti:value id="false" name="Reject" />
        </activiti:formProperty>
        <activiti:formProperty id="managerMotivation" name="Motivation" type="string" />
      </extensionElements>
    </userTask>
  </process>
</definitions>
  
```

```

<potentialOwner>
    <resourceAssignmentExpression>
        <formalExpression>management</formalExpression>
    </resourceAssignmentExpression>
</potentialOwner>
</userTask>
<sequenceFlow id="flow2" sourceRef="handleRequest" targetRef="requestApprovedDecision" />

<exclusiveGateway id="requestApprovedDecision" name="Request approved?" />
<sequenceFlow id="flow3" sourceRef="requestApprovedDecision" targetRef="sendApprovalMail">
    <conditionExpression xsi:type="tFormalExpression">${vacationApproved == 'true'}</conditionExpression>
</sequenceFlow>

<task id="sendApprovalMail" name="Send confirmation e-mail" />
<sequenceFlow id="flow4" sourceRef="sendApprovalMail" targetRef="theEnd1" />
<endEvent id="theEnd1" />

<sequenceFlow id="flow5" sourceRef="requestApprovedDecision" targetRef="adjustVacationRequestTask">
    <conditionExpression xsi:type="tFormalExpression">${vacationApproved == 'false'}</conditionExpression>
</sequenceFlow>

<userTask id="adjustVacationRequestTask" name="Adjust vacation request">
    <documentation>
        Your manager has disapproved your vacation request for ${numberOfDays} days.
        Reason: ${managerMotivation}
    </documentation>
    <extensionElements>
        <activiti:formProperty id="numberOfDays" name="Number of days" value="${numberOfDays}" type="long" required="true" />
        <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyyy)" value="${startDate}" date="true" />
        <activiti:formProperty id="vacationMotivation" name="Motivation" value="${vacationMotivation}" type="string" />
        <activiti:formProperty id="resendRequest" name="Resend vacation request to manager?" type="enum" required="true" />
            <activiti:value id="true" name="Yes" />
            <activiti:value id="false" name="No" />
    </activiti:formProperty>
    </extensionElements>
    <humanPerformer>
        <resourceAssignmentExpression>
            <formalExpression>${employeeName}</formalExpression>
        </resourceAssignmentExpression>
    </humanPerformer>
</userTask>
<sequenceFlow id="flow6" sourceRef="adjustVacationRequestTask" targetRef="resendRequestDecision" />

<exclusiveGateway id="resendRequestDecision" name="Resend request?" />
<sequenceFlow id="flow7" sourceRef="resendRequestDecision" targetRef="handleRequest">
    <conditionExpression xsi:type="tFormalExpression">${resendRequest == 'true'}</conditionExpression>
</sequenceFlow>

<sequenceFlow id="flow8" sourceRef="resendRequestDecision" targetRef="theEnd2" />
<conditionExpression xsi:type="tFormalExpression">${resendRequest == 'false'}</conditionExpression>
</sequenceFlow>
<endEvent id="theEnd2" />

</process>
</definitions>

```

为了让Activiti引擎知道这个流程，我们必须先进行“发布”。发布意味着引擎会把BPMN 2.0 xml解析成可以执行的东西，“发布包”中的所有流程定义都会添加到数据库中。这样，当引擎重启时，它依然可以获得“已发布”的流程：

```
ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
RepositoryService repositoryService = processEngine.getRepositoryService();
repositoryService.createDeployment()
    .addClasspathResource("org/activiti/test/VacationRequest.bpmn20.xml")
    .deploy();

Log.info("Number of process definitions: " + repositoryService.createProcessDefinitionQuery().count());
```

可以阅读发布章来了解更多关于发布的信息。

4.3.2. #启动一个流程实例

把流程定义发布到Activiti引擎后，我们可以基于它发起新流程实例。对每个流程定义，都可以有很多流程实例。流程定义是“蓝图”，流程实例是它的一个运行的执行。

所有与流程运行状态相关的东西都可以通过RuntimeService获得。有很多方法可以启动一个新流程实例。在下面的代码中，我们使用定义在流程定义xml 中的key来启动流程实例。我们也可以在流程实例启动时添加一些流程变量，因为第一个用户任务的表达式需要这些变量。流程变量经常会被用到，因为它们赋予来自同一个流程定义的不同流程实例的特别含义。简单来说，流程变量是区分流程实例的关键。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("employeeName", "Kermit");
variables.put("number0fDays", new Integer(4));
variables.put("vacationMotivation", "I'm really tired!");

RuntimeService runtimeService = processEngine.getRuntimeService();
ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("vacationRequest", variables);

// Verify that we started a new process instance
Log.info("Number of process instances: " + runtimeService.createProcessInstanceQuery().count());
```

4.3.3. #完成任务

流程启动后，第一步就是用户任务。这是必须由系统用户处理的一个环节。通常，用户会有一个“任务列表”，展示了所有必须由整个用户处理的任务。下面的代码展示了对应的查询可能是怎样的：

```
// Fetch all tasks for the management group
TaskService taskService = processEngine.getTaskService();
List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();
for (Task task : tasks) {
    Log.info("Task available: " + task.getName());
}
```

为了让流程实例继续运行，我们需要完成整个任务。对Activiti来说，就是需要complete任务。下面的代码展示了如何做这件事：

```
Task task = tasks.get(0);

Map<String, Object> taskVariables = new HashMap<String, Object>();
taskVariables.put("vacationApproved", "false");
taskVariables.put("managerMotivation", "We have a tight deadline!");
taskService.complete(task.getId(), taskVariables);
```

流程实例会进入到下一个环节。在这里例子中，下一环节允许员工通过表单调整原始的请假申请。员工可以重新提交请假申请，这会使流程重新进入到第一个任务。

4. 3. 4. #挂起，激活一个流程

我们可以挂起一个流程定义。当挂起流程定时时，就不能创建新流程了（会抛出一个异常）。可以通过RepositoryService挂起一个流程：

```
repositoryService.suspendProcessDefinitionByKey("vacationRequest");
try {
    runtimeService.startProcessInstanceByKey("vacationRequest");
} catch (ActivitiException e) {
    e.printStackTrace();
}
```

要想重新激活一个流程定义，可以调用repositoryService.activateProcessDefinitionXXX方法。

也可以挂起一个流程实例。挂起时，流程不能继续执行（比如，完成任务会抛出异常），异步操作（比如定时器）也不会执行。

runtimeService.suspendProcessInstance方法。

用runtimeService.activateProcessInstanceXXX方法。

骨气流程实例可以调
激活流程实例可以调

4. 3. 5. #更多知识

上面章节中我们仅仅覆盖了Activiti功能的表层。未来我们会继续扩展这些章节，以覆盖更多Activiti API。当然，像其他开源项目一样，学习的最好方式是研究代码，阅读javadoc。

4. 4. #查询API

有两种方法可以从引擎中查询数据：查询API和原生查询。查询API提供了完全类型安全的API。你可以为自己的查询条件添加很多条件（所以条件都以AND组合）和精确的排序条件。下面的代码展示了一个例子：

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

有时，你需要更强大的查询，比如使用OR条件或不能使用查询API实现的条件。这时，我们推荐原生查询，它让你可以编写自己的SQL查询。返回类型由你使用的查询对象决定，数据会映射到正确的对象上。比如，任务，流程实例，，执行，等等。因为查询会作用在数据库上，你必须使用数据库中定义的表名和列名；这要求了解内部数据结构，因此使用原生查询时一定要注意。表名可以通过API获得，可以尽量减少对数据库的依赖。

```
List<Task> tasks = taskService.createNativeTaskQuery()
    .sql("SELECT count(*) FROM " + managementService.getTableName(Task.class) + " T WHERE T.NAME_ = #{taskName}")
    .parameter("taskName", "gonzoTask")
    .list();

long count = taskService.createNativeTaskQuery()
    .sql("SELECT count(*) FROM " + managementService.getTableName(Task.class) + " T1, "
        + managementService.getTableName(VariableInstanceEntity.class) + " V1 WHERE V1.TASK_ID_ = T1.ID_")
    .count();
```

4. 5. #表达式

Activiti使用UEL处理表达式。UEL即统一表达式语言，它是EE6规范的一部分（参考 EE6规范 [<http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html>]）。为了在所有运行环境都支持最新UEL的所有功能，我们使用了一个JUEL的修改版本。

表达式可以用在很多场景下，比如Java服务任务，执行监听器，任务监听器和条件流。虽然有两重表达式，值表达式和方法表达式，Activiti进行了抽象，所以两者可以同样使用在需要表达式的场景中。

- Value expression: 解析为值。默认，所有流程变量都可以使用。所有spring bean（spring环境中）也可以使用在表达式中。一些实例：

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

- Method expression: 调用一个方法，使用或不使用参数。当调用一个无参数的方法时，记得在方法名后添加空的括号（以区分值表达式）。传递的参数可以是字符串也可以是表达式，它们会被自动解析。例子：

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

注意这些表达式支持解析原始类型（包括比较），bean，list，数组和map。

在所有流程实例中，表达式中还可以使用一些默认对象：

- execution: DelegateExecution提供外出执行的额外信息。
- task: DelegateTask提供当前任务的额外信息。注意，只对任务监听器的表达式有效。
- authenticatedUserId: 当前登录的用户id。如果没有用户登录，这个变量就不可用。

想要更多具体的使用方式和例子，参考spring中的表达式，Java服务任务，执行监听器，任务监听器或条件流。

4. 6. #单元测试

业务流程是软件项目的一部分，它也应该和普通的业务流程一样进行测试： 使用单元测试。因为Activiti是一个嵌入式的java引擎， 为业务流程编写单元测试和写普通单元测试完全一样。

Activiti支持JUnit 3和4进行单元测试。使用JUnit 3时， 必须集成org.activiti.engine.test.ActivitiTestCase。 它通过保护的成员变量提供ProcessEngine和服务，在测试的setup()中， 默认会使用classpath下的activiti.cfg.xml初始化流程引擎。 想使用不同的配置文件，可以重写getConfigurationResource()方法。 如果配置文件相同的话， 对应的流程引擎会被静态缓存， 就可以用于多个单元测试。

继承了ActivitiTestCase你，可以在测试方法上使用 org.activiti.engine.test.Deployment注解。 测试执行前，与测试类在同一个包下的， 格式为testClassName.testMethod.bpmn20.xml的资源文件，会被部署。 测试结束后，发布包也会被删除，包括所有相关的流程实例，任务，等等。 Deployment注解也可以直接设置资源的位置。 参考Javadocs [.. / javadocs/org/activiti/engine/test/Deployment.html]获得更多信息。

把这些放在一起，JUnit 3测试看起来像这样。

```
public class MyBusinessProcessTest extends ActivitiTestCase {

    @Deployment
    public void testSimpleProcess() {
        runtimeService.startProcessInstanceByKey("simpleProcess");

        Task task = taskService.createTaskQuery().singleResult();
        assertEquals("My Task", task.getName());

        taskService.complete(task.getId());
        assertEquals(0, runtimeService.createProcessInstanceQuery().count());
    }
}
```

要想在使用JUnit 4编写单元测试时获得同样的功能， 可以使用org.activiti.engine.test.ActivitiRule。 通过它，可以通过getter方法获得流程引擎和各种服务。 和 ActivitiTestCase一样（参考上面章节）， 使用这个Rule 也会启用org.activiti.engine.test.Deployment注解（参考上面章节使用和配置的介绍）， 它会在classpath下查找默认的配置文件。 如果配置文件相同的话，对应的流程引擎会被静态缓存， 就可以用于多个单元测试。

下面的代码演示了JUnit 4单元测试并使用了ActivitiRule的例子。

```
public class MyBusinessProcessTest {

    @Rule
    public ActivitiRule activitiRule = new ActivitiRule();

    @Test
    @Deployment
    public void ruleUsageExample() {
        RuntimeService runtimeService = activitiRule.getRuntimeService();
        runtimeService.startProcessInstanceByKey("ruleUsage");
    }
}
```

```

TaskService taskService = activitiRule.getTaskService();
Task task = taskService.createTaskQuery().singleResult();
assertEquals("My Task", task.getName());

taskService.complete(task.getId());
assertEquals(0, runtimeService.createProcessInstanceQuery().count());
}
}

```

4. 7. #调试单元测试

当使用内存数据库H2进行单元测试时，下面的教程会告诉我们 如何在调试环境下更容易的监视Activiti的数据库。 这里的截图都是基于eclipse，这种机制很容易复用到其他IDE下。 IDEs.

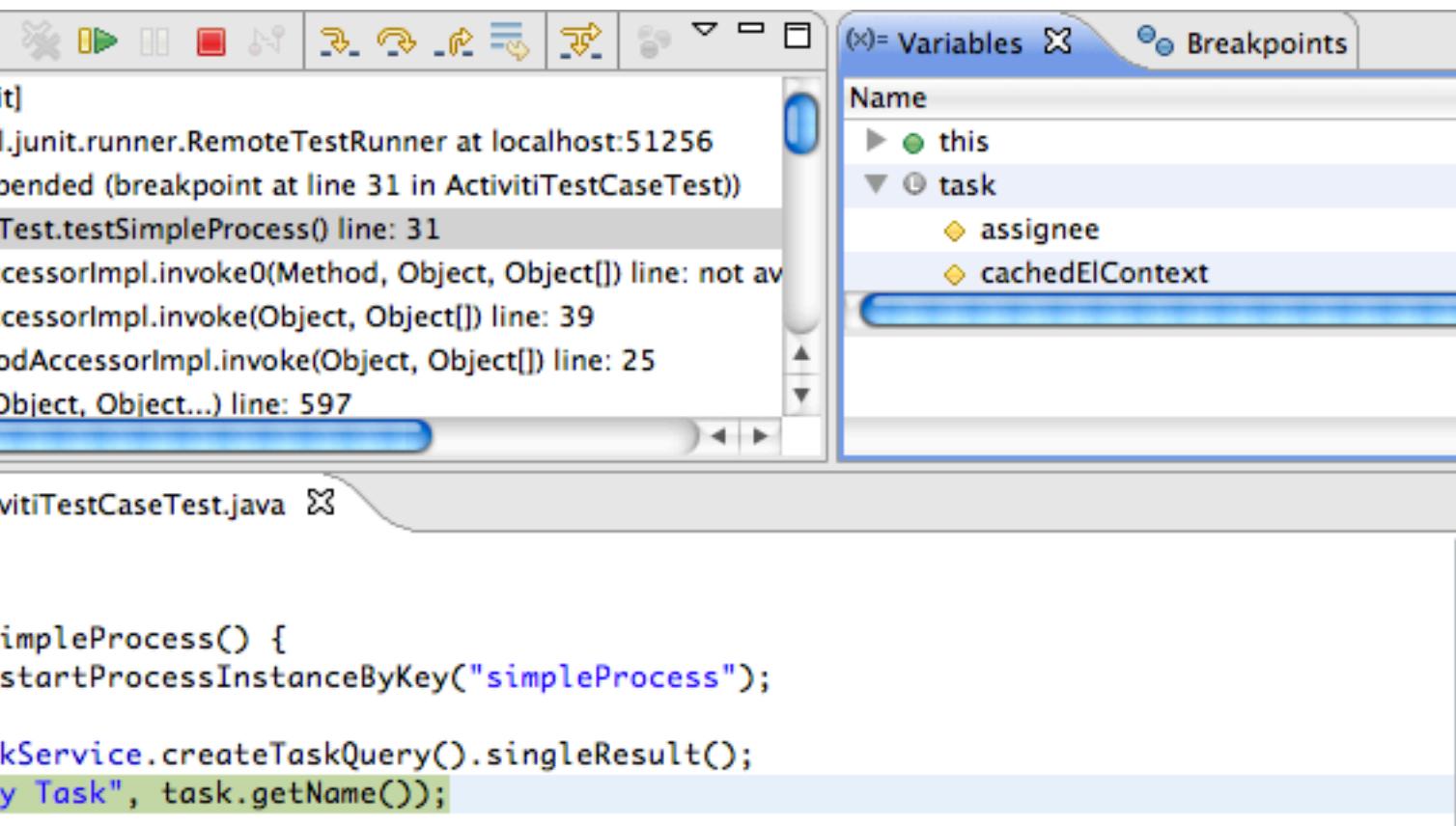
假设我们已经在单元测试里设置了一个断点。 Eclipse里，在代码左侧双击：

```

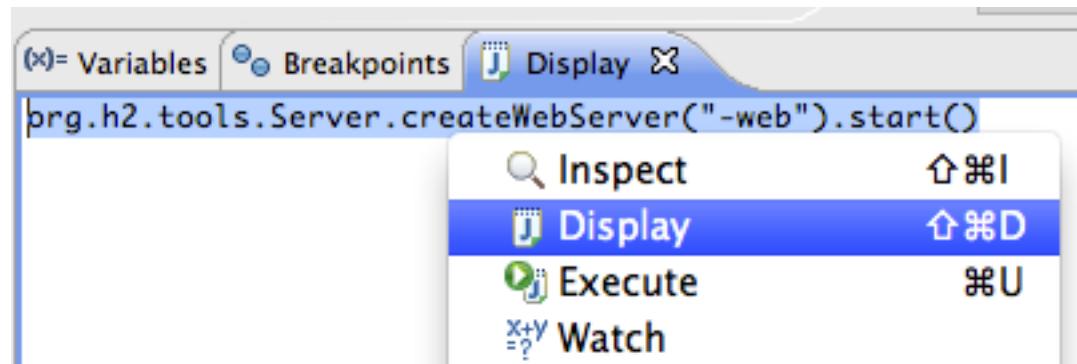
27 public void testSimpleProcess() {
28     runtimeService.startProcessInstanceByKey("simpleProcess");
29
30     Task task = taskService.createTaskQuery().singleResult();
31     assertEquals("My Task", task.getName());
32

```

现在用调试模式运行单元测试（右击单元测试，选择“运行为”和“单元测试”），测试会停在我们的断点上，然后我们就可以监视测试的变量，它们显示在右侧面板里。



要监视Activiti的数据，打开“显示”窗口 （如果找不到，打开“窗口”->“显示视图”->“其他”，选择显示。） 并点击（代码已完成）`org.h2.tools.Server.createWebServer("-web").start()`



选择你点击的行，右击。然后选择“显示”（或者直接快捷方式就不用右击了）



现在打开一个浏览器，打开`http://localhost:8082`，输入内存数据库的JDBC URL（默认为`jdbc:h2:mem:activiti`），点击连接按钮。

The screenshot shows a 'Login' configuration dialog. It has two sections:

- Saved Settings:** A dropdown menu set to "Generic H2 (Embedded)".
- Setting Name:** A text input field containing "Generic H2 (Embedded)" with "Save" and "Remove" buttons next to it.

Below these are four fields for database connection details:

- Driver Class:** `org.h2.Driver`
- JDBC URL:** `jdbc:h2:mem:activiti`
- User Name:** `sa`
- Password:** (empty field)

At the bottom are two buttons: "Connect" and "Test Connection".

你仙子阿可以看到Activiti的数据，通过它们可以了解单元测试时如何以及为什么这样运行的。

jdbc:h2:mem:activiti
ACT_GE_BYTEARRAY
ACT_GE_PROPERTY
ACT_HI_ACTINST
ACT_HI_DETAIL
ACT_HI_PROCINST
ACT_HI_TASKINST
ACT_ID_GROUP
ACT_ID_MEMBERSHIP
ACT_ID_USER
ACT_RE_DEPLOYMENT
ACT_RE_PROCDEF
ACT_RU_EXECUTION
ACT_RU_IDENTITYLINK
ACT_RU_JOB
ACT_RU_TASK
ACT_RU_VARIABLE

Run (Ctrl+Enter) Clear SQL statement:

```
SELECT * FROM ACT_RU_TASK
```

SELECT * FROM ACT_RU_TASK;

ID_	REV_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	NAM
6	1	4	4	simpleProcess:1:3	My

(1 row, 6 ms)

Edit

4. 8. #web应用中的流程引擎

ProcessEngine是线程安全的，可以在多线程下共享。在web应用中，意味着可以在容器启动时创建流程引擎，在容器关闭时关闭流程引擎。

下面代码演示了如何编写一个ServletContextListener 在普通的Servlet环境下初始化和销毁流程引擎：

```
public class ProcessEnginesServletContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent servletContextEvent) {
        ProcessEngines.init();
    }

    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        ProcessEngines.destroy();
    }
}
```

contextInitialized方法会执行ProcessEngines.init()。这会查找classpath下的activiti.cfg.xml文件，根据配置文件创建一个ProcessEngine（比如，多个jar中都包含配置文件）。如果classpath中包含多个配置文件，确认它们有不同的名字。当需要使用流程引擎时，可以通过

ProcessEngines.getDefaultProcessEngine()

或

ProcessEngines.getProcessEngine("myName");

。当然，也可以使用其他方式创建流程引擎，可以参考配置章节中的描述。

ContextListener中的contextDestroyed方法会执行ProcessEngines.destroy(). 这会关闭所有初始化的流程引擎。

第#5#章#Spring集成

虽然没有Spring你也可以使用Activiti，但是我们提供了一些非常不错的集成特性。这一章我们将介绍这些特性。

5. 1. #ProcessEngineFactoryBean

可以把流程引擎（ProcessEngine）作为一个普通的Spring bean进行配置。类 org.activiti.spring.ProcessEngineFactoryBean是集成的切入点。这个bean需要一个流程引擎配置来创建流程引擎。这也意味着在文档的配置这一章的介绍属性的创建和配置对于Spring来说也是一样的。对于Spring集成的配置和流程引擎bean看起来像这样：

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
</bean>

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
    <property name="processEngineConfiguration" ref="processEngineConfiguration" />
</bean>
```

注意现在使用的 processEngineConfiguration bean 是 org.activiti.spring.SpringProcessEngineConfiguration 类。

5. 2. #事务

我们将会一步一步地解释在Spring examples中公布的 SpringTransactionIntegrationTest 下面是我们使用这个例子的Spring配置文件（你可以在SpringTransactionIntegrationTest-context.xml找到它）以下展示的部分包括数据源（dataSource），事务管理器（transactionManager），流程引擎（processEngine）和Activiti引擎服务。

当把数据源（DataSource）传递给 SpringProcessEngineConfiguration（使用“dataSource”属性）之后，Activiti内部使用了一个org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy代理来封装传递进来的数据源（DataSource）。这样做是为了确保从数据源（DataSource）获取的SQL连接能够与Spring的事物结合在一起发挥得更出色。这意味着它不再需要在你的Spring配置中代理数据源（dataSource）了。然而它仍然允许你传递一个TransactionAwareDataSourceProxy到SpringProcessEngineConfiguration中。在这个例子中并不会发生多余的包装。

为了确保在你的Spring配置中申明的一个TransactionAwareDataSourceProxy，你不能把使用它的应用交给Spring事物控制的资源。（例如 DataSourceTransactionManager 和 JPATransactionManager需要非代理的数据源）

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/
                           http://www.springframework.org/schema/context http://www.springframework.org/schema/
                           http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx"/>
```

```

<bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
    <property name="driverClass" value="org.h2.Driver" />
    <property name="url" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    <property name="dataSource" ref="dataSource" />
    <property name="transactionManager" ref="transactionManager" />
    <property name="databaseSchemaUpdate" value="true" />
    <property name="jobExecutorActivate" value="false" />
</bean>

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
    <property name="processEngineConfiguration" ref="processEngineConfiguration" />
</bean>

<bean id="repositoryService" factory-bean="processEngine" factory-method="getRepositoryService" />
<bean id="runtimeService" factory-bean="processEngine" factory-method="getRuntimeService" />
<bean id="taskService" factory-bean="processEngine" factory-method="getTaskService" />
<bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService" />
<bean id="managementService" factory-bean="processEngine" factory-method="getManagementService" />

...

```

Spring配置文件的其余部分包含beans和我们将要在这个特有的例子中的配置：

```

<beans>
    ...
    <tx:annotation-driven transaction-manager="transactionManager"/>

    <bean id="userBean" class="org.activiti.spring.test.UserBean">
        <property name="runtimeService" ref="runtimeService" />
    </bean>

    <bean id="printer" class="org.activiti.spring.test.Printer" />
</beans>

```

首先使用任意的一种Spring创建应用上下文的方式创建其Spring应用上下文。在这个例子中你可以使用类路径下面的XML资源来配置我们的Spring应用上下文：

```

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

```

或者，如果它是一个测试的话：

```

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

```

然后我们就可以得到Activiti的服务beans并且调用该服务上面的方法。ProcessEngineFactoryBean将会对该服务添加一些额外的拦截器，在Activiti服务上面

的方法使用的是 Propagation.REQUIRED事物语义。所以，我们可以使用repositoryService去部署一个流程，如下所示：

```
RepositoryService repositoryService = (RepositoryService) applicationContext.getBean("repositoryService");
String deploymentId = repositoryService
    .createDeployment()
    .addClasspathResource("org/activiti/spring/test/hello.bpmn20.xml")
    .deploy()
    .getId();
```

其他相同的服务也是同样可以这么使用。在这个例子中，Spring的事物将会围绕在userBean.hello()上，并且调用Activiti服务的方法也会加入到这个事物中。

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

这个UserBean看起来像这样。记得在上面Spring bean的配置中我们把repositoryService注入到userBean中。

```
public class UserBean {

    /** 由Spring注入 */
    private RuntimeService runtimeService;

    @Transactional
    public void hello() {
        //这里，你可以在你们的领域模型中做一些事物处理。
        //当在调用Activiti RuntimeService的startProcessInstanceByKey方法时,
        //它将会结合到同一个事物中。
        runtimeService.startProcessInstanceByKey("helloProcess");
    }

    public void setRuntimeService(RuntimeService runtimeService) {
        this.runtimeService = runtimeService;
    }
}
```

5. 3. #表达式

当使用ProcessEngineFactoryBean时候，默认情况下，在BPMN流程中的所有表达式都将会'看见'所有的Spring beans。它可以限制你在表达式中暴露出的beans或者甚至可以在你的配置中使用一个Map不暴露任何beans。下面的例子暴露了一个单例bean (printer)，可以把"printer"当作关键字使用。想要不暴露任何beans，仅仅只需要在SpringProcessEngineConfiguration中传递一个空的list作为'beans'的属性。当不设置'beans'的属性时，在应用上下文中Spring beans都是可以使用的。

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
    <property name="beans">
        <map>
            <entry key="printer" value-ref="printer" />
        </map>
    </property>
</bean>
```

```
<bean id="printer" class="org.activiti.examples.spring.Printer" />
```

现在暴露出来的beans就可以在表达式中使用：例如，在 SpringTransactionIntegrationTest中的 hello.bpmn20.xml展示的是如何使用UEL方法表达式去调用Spring bean的方法：

```
<definitions id="definitions" ...>

<process id="helloProcess">

    <startEvent id="start" />
    <sequenceFlow id="flow1" sourceRef="start" targetRef="print" />

    <serviceTask id="print" activiti:expression="#{printer.printMessage()}" />
    <sequenceFlow id="flow2" sourceRef="print" targetRef="end" />

    <endEvent id="end" />

</process>

</definitions>
```

这里的 Printer 看起来像这样：

```
public class Printer {

    public void printMessage() {
        System.out.println("hello world");
    }
}
```

并且Spring bean的配置（如上文所示）看起来像这样：

```
<beans ...>
    ...
    <bean id="printer" class="org.activiti.examples.spring.Printer" />
</beans>
```

5. 4. #资源的自动部署

Spring的集成也有一个专门用于对资源部署的特性。在流程引擎的配置中，你可以指定一组资源。当流程引擎被创建的时候，所有在这里的资源都将会被自动扫描与部署。在这里有过滤以防止资源重新部署，只有当这个资源真正发生改变的时候，它才会向Activiti使用的数据库创建新的部署。这对于很多用例来说，当Spring容器经常重启的情况下（例如 测试），使用它是非常不错的选择。

这里有一个例子：

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
    <property name="deploymentResources" value="classpath*:./org/activiti/spring/test/autodeployment/autodeploy.*.bpmn20.xml" />
</bean>
```

```

</bean>

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
    <property name="processEngineConfiguration" ref="processEngineConfiguration" />
</bean>

```

默认，上面的配置会把所有匹配的资源发布到Activiti引擎的一个单独发布包下。用来检测防止未修改资源重复发布的机制会作用到整个发布包中。有时候，这可能不是你想要的。比如，如果你发布了很多流程资源，但是只修改了其中某一个单独的流程定义，整个发布包都会被认为变更了，导致整个发布包下的所有流程定义都会被重新发布，结果就是每个流程定义都生成了新版本，虽然其中只有一个流程发生了改变。

为了定制发布方式，你可以为SpringProcessEngineConfiguration指定一个额外的参数`deploymentMode`。这个参数指定了匹配多个资源时的发布处理方式。默认下这个参数支持设置三个值：

- `default`: 把所有资源放在一个单独的发布包中，对这个发布包进行重复检测。这是默认值，如果你没有指定参数值，就会使用它。
- `single-resource`: 为每个单独的资源创建一个发布包，并对这些发布包进行重复检测。你可以单独发布每个流程定义，并在修改流程定义后只创建一个新的流程定义版本。
- `resource-parent-folder`: 把放在同一个上级目录下的资源发布在一个单独的发布包中，并对发布包进行重复检测。当需要多资源需要创建发布包，但是需要根据共同的文件夹来组合一些资源时，可以使用它。

这儿有一个例子来演示将`deploymentMode`参数配置为`single-resource`的情况：

```

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
    <property name="deploymentResources" value="classpath*:activiti/*.bpmn" />
    <property name="deploymentMode" value="single-resource" />
</bean>

```

如果想使用上面三个值之外的参数值，你需要自定义处理发布包的行为。你可以创建一个`SpringProcessEngineConfiguration`的子类，重写`getAutoDeploymentStrategy(String deploymentMode)`方法。这个方法中处理了对应`deploymentMode`的发布策略。

5. 5. #单元测试

当集成Spring时，使用标准的Activiti测试工具类是非常容易的对业务流程进行测试。下面的例子展示了如何在一个典型的基于Spring单元测试测试业务流程：

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:org/activiti/spring/test/junit4/springTypicalUsageTest-context.xml")
public class MyBusinessProcessTest {

    @Autowired
    private RuntimeService runtimeService;

    @Autowired
    private TaskService taskService;
}

```

```

@Autowired
@Rule
public ActivitiRule activitiSpringRule;

@Test
@Deployment
public void simpleProcessTest() {
    runtimeService.startProcessInstanceByKey("simpleProcess");
    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("My Task", task.getName());

    taskService.complete(task.getId());
    assertEquals(0, runtimeService.createProcessInstanceQuery().count());
}

}

```

注意对于这种方式，你需要在Spring配置中（在上文的例子中它是自动注入的）定义一个org.activiti.engine.test.ActivitiRulebean

```

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

```

5. 6. #基于注解的配置

[试验] @EnableActiviti注解相对较新，未来可能会有变更。

除了基于XML的配置以外，还可以选择基于注解的方式来配置Spring环境。这与使用XML的方法非常相似，除了要使用@Bean注解，而且配置是使用java编写的。它已经可以直接用于Activiti-Spring的集成了：

首先介绍（需要Spring 3.0+）的是@EnableActiviti注解。最简单的用法如下所示：

```

@Configuration
@EnableActiviti
public static class SimplestConfiguration {
}

```

它会创建一个Spring环境，并对Activiti流程引擎进行如下配置

- 默认的内存H2数据库，启用数据库自动升级。
 - 一个简单的 DataSourceTransactionManager
 - 一个默认的 SpringJobExecutor
 - 自动扫描 processes/ 目录下的bpmn20.xml文件。
- 在这样一个环境里，可以直接通过注入操作Activiti引擎：

```

@Autowired
private ProcessEngine processEngine;

```

```

@Autowired
private RuntimeService runtimeService;

@Autowired
private TaskService taskService;

@Autowired
private HistoryService historyService;

@Autowired
private RepositoryService repositoryService;

@Autowired
private ManagementService managementService;

@Autowired
private FormService formService;

```

当然， 默认值都可以自定义。比如， 如果配置了DataSource， 它就会代替默认创建的数据库配置。 事务管理器， job执行器和其他组件都与之相同。 比如如下配置：

```

@Configuration
@EnableActiviti
public static class Config {

    @Bean
    public DataSource dataSource() {
        BasicDataSource basicDataSource = new BasicDataSource();
        basicDataSource.setUsername("sa");
        basicDataSource.setUrl("jdbc:h2:mem:anotherDatabase");
        basicDataSource.setDefaultAutoCommit(false);
        basicDataSource.setDriverClassName(org.h2.Driver.class.getName());
        basicDataSource.setPassword("");
        return basicDataSource;
    }

}

```

其他数据库会代替默认的。

下面介绍了更加复杂的配置。注意AbstractActivitiConfigurer用法， 它暴露了流程引擎的配置， 可以用来对它的细节进行详细的配置。

```

@Configuration
@EnableActiviti
@EnableTransactionManagement(proxyTargetClass = true)
class JPAConfiguration {

    @Bean
    public OpenJpaVendorAdapter openJpaVendorAdapter() {
        OpenJpaVendorAdapter openJpaVendorAdapter = new OpenJpaVendorAdapter();
        openJpaVendorAdapter.setDatabasePlatform(H2Dictionary.class.getName());
        return openJpaVendorAdapter;
    }

    @Bean
    public DataSource dataSource() {

```

```

BasicDataSource basicDataSource = new BasicDataSource();
basicDataSource.setUsername("sa");
basicDataSource.setUrl("jdbc:h2:mem:activiti");
basicDataSource.setDefaultAutoCommit(false);
basicDataSource.setDriverClassName(org.h2.Driver.class.getName());
basicDataSource.setPassword("");
return basicDataSource;
}

@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean(
    OpenJpaVendorAdapter openJpaVendorAdapter, DataSource ds) {
    LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
    emf.setPersistenceXmlLocation("classpath:/org/activiti/spring/test/jpa/custom-persistence.xml");
    emf.setJpaVendorAdapter(openJpaVendorAdapter);
    emf.setDataSource(ds);
    return emf;
}

@Bean
public PlatformTransactionManager jpaTransactionManager(
    EntityManagerFactory entityManagerFactory) {
    return new JpaTransactionManager(entityManagerFactory);
}

@Bean
public AbstractActivitiConfigurer abstractActivitiConfigurer(
    final EntityManagerFactory emf,
    final PlatformTransactionManager transactionManager) {

    return new AbstractActivitiConfigurer() {

        @Override
        public void postProcessSpringProcessEngineConfiguration(SpringProcessEngineConfiguration engine) {
            engine.setTransactionManager(transactionManager);
            engine.setJpaEntityManagerFactory(emf);
            engine.setJpaHandleTransaction(false);
            engine.setJobExecutorActivate(false);
            engine.setJpaCloseEntityManager(false);
            engine.setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB_SCHEMA_UPDATE_TRUE);
        }
    };
}

// A random bean
@Bean
public LoanRequestBean loanRequestBean() {
    return new LoanRequestBean();
}
}

```

5. 7. #JPA 和 Hibernate 4.2.x

在Activiti引擎的serviceTask或listener中使用Hibernate 4.2.x JPA时，需要添加Spring ORM这个额外的依赖。Hibernate 4.1.x及以下版本是不需要的。应该添加如下依赖：

```

<dependency>
    <groupId>org.springframework</groupId>

```

```
<artifactId>spring-orm</artifactId>
<version>${org.springframework.version}</version>
</dependency>
```

第#6#章#部署

6. 1. #业务文档

为了部署流程，它们不得不包装在一个业务文档中。一个业务文档是Activiti引擎部署的单元。一个业务文档相当与一个压缩文件，它包含BPMN2.0流程，任务表单，规则和其他任意类型的文件。大体上，业务文档是包含命名资源的容器。

当一个业务文档被部署，它将会自动扫描以 .bpmn20.xml 或者.bpmn作为扩展名的BPMN文件。每个那样的文件都将会被解析并且可能会包含多个流程定义。

注意

业务归档中的Java类将不能够添加到类路径下。为了能够让流程运行，必须把存在于业务归档程中的流程定义使用的所有自定义的类（例如：Java服务任务或者实现事件的监听器）放在activiti引擎的类路径下：

6. 1. 1. #编程式部署

通过一个压缩文件（支持Zip和Bar）部署业务归档，它看起来像这样：

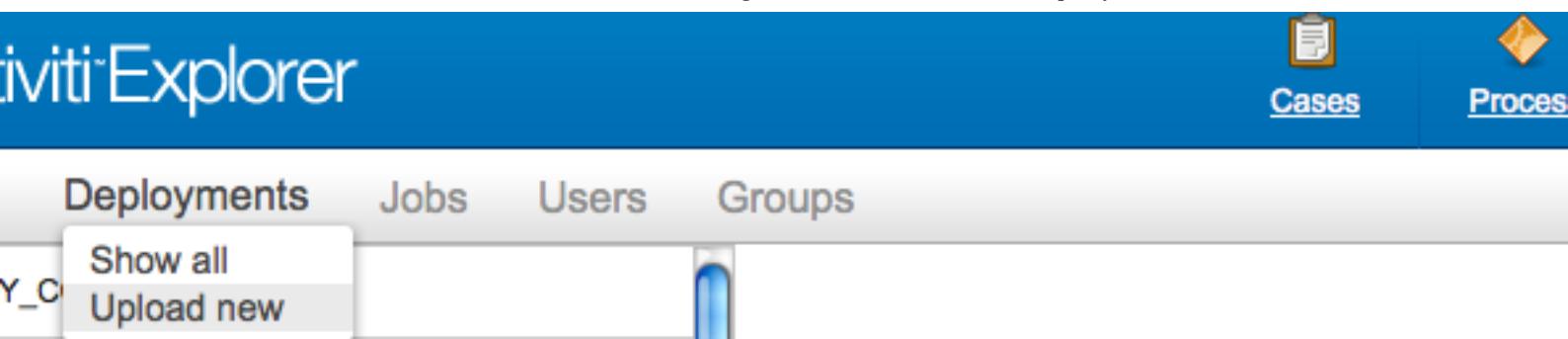
```
String barFileName = "path/to/process-one.bar";
ZipInputStream inputStream = new ZipInputStream(new FileInputStream(barFileName));

repositoryService.createDeployment()
    .name("process-one.bar")
    .addZipInputStream(inputStream)
    .deploy();
```

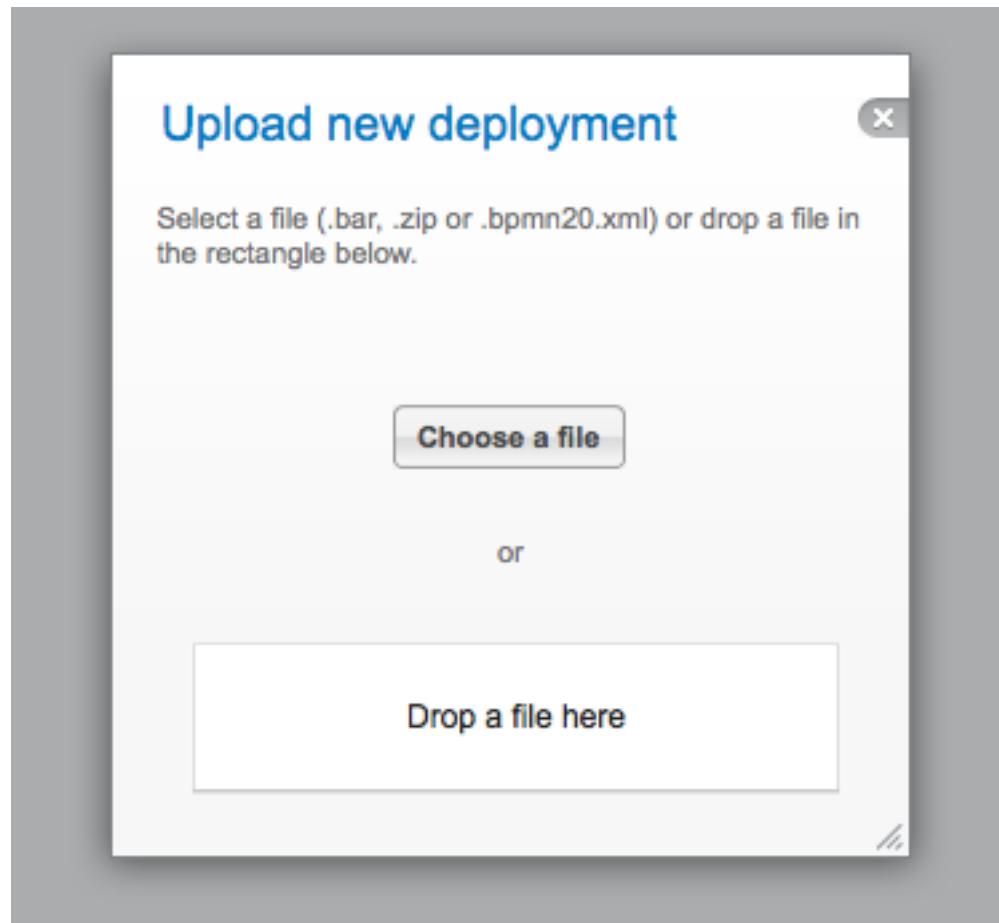
它也可以通过一个独立资源（例如bpmn, xml等）构建部署。详细信息请查看javadocs。

6. 1. 2. #通过Activiti Explorer控制台部署

Activiti web控制台允许你通过web界面的用户接口上传一个bar格式的压缩文件（或者一个bpmn20.xml格式的文件）。选择Management 标签 和 点击 Deployment:



现在将会有个弹出窗口允许你从电脑上面选择一个文件，或者你可以简单的拖拽到指定的区域（如果你的浏览器支持）。



6.2. #外部资源

流程定义保存在Activiti所支持的数据库中。当使用服务任务、执行监听器或者从Activiti配置文件中配置的Spring beans时，流程定义能够引用这些委托类。这些类或者Spring配置文件对于所有流程引擎中可能执行的流程定义必须是可用的。

6.2.1. #Java类

当流程实例被启动的时候，在流程中被使用的所有自定义类（例如：服务任务中使用的JavaDelegates、事件监听器、任务监听器，...）应该存在与流程引擎的类路径下。

然后，在部署业务文档时，这些类不必都存在于类路径下。当使用Ant部署一个新的业务文档时，这意味着你的委托类不必存在与类路径下。

当你使用示例设置并添加你自定义的类，你应该添加包含自定义类的jar包到activiti-explorer控制台或者activiti-rest 的webapp lib文件夹中。以及不要忽略包含你自定义类的依赖关系（如果有）。另外，你还可以包含你自己的依赖添加到你的Tomcat容器的安装目录中的\${tomcat.home}/lib。

6.2.2. #在流程中使用Spring beans

当表达式或者脚本使用Spring beans时，这些beans对于引擎执行流程定义时必须是可用的。如果你将要构建你自己的web应用并且按照Spring集成这一章中描述那样在你的应用上下文配置流程引擎，这个看上去非常的简单。但是要记住，如果你也在使用 Activiti rest web 应用，那么也应该更新 Activiti rest web应用的上下文。你可以把在activiti-rest/lib/activiti-cfg.jar 文件中的activiti.cfg.xml替换成你的Spring上下文配置的activiti-context.xml文件。

6. 2. 3. #创建独立应用

你可以考虑把Activiti rest web 应用加入到你的web应用之中，因此，就仅仅只需要配置一个 ProcessEngine，从而不用确保所有的流程引擎的所有委托类在类路径下面并且是否使用正确的spring配置。

6. 3. #流程定义的版本

BPMN中并没有版本的概念，没有版本也是不错的，因为可执行的BPMN流程作为你开发项目的一部分存在版本控制系统的知识库中（例如 SVN, Git 或者Mercurial）。而在Activiti中，流程定义的版本是在部署时创建的。在部署的时候，流程定义被存储到Activiti使用的数据库之前，Activiti讲会自动给 流程定义 分配一个版本号。

对于业务文档中每一个的流程定义，都会通过下列部署执行初始化属性key, version, name 和 id:

- XML文件中流程定义（流程模型）的 id属性被当做是流程定义的 key属性。
- XML文件中的流程模型的name 属性被当做是流程定义的 name 属性。如果该name属性并没有指定，那么id属性被当做是name。
- 带有特定key的流程定义在第一次部署的时候，将会自动分配版本号为1，对于之后部署相同key的流程定义时候，这次部署的版本号将会设置为比当前最大的版本号大1的值。该 key属性被用来区别不同的流程定义。
- 流程定义中的id属性被设置为 {processDefinitionKey}:{processDefinitionVersion}:{generated-id}，这里的generated-id是一个唯一的数字被添加，用于确保在集群环境中缓存的流程定义的唯一性。

举个流程的例子

```
<definitions id="myDefinitions" >
  <process id="myProcess" name="My important process" >
    ...
  </process>
</definitions>
```

当部署了这个流程定义之后，在数据库中的流程定义看起来像这样：

表 6.1.

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

假设我们现在部署用一个流程的最新版本号（例如 改变用户任务），但是流程定义的id保持不变。 流程定义表将包含以下列表信息：

表 6.2.

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

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

当 `runtimeService.startProcessInstanceByKey("myProcess")` 方法被调用时，它将会使用流程定义版本号为2的，因为这是最新版本的流程定义。可以说每次流程定义创建流程实例时，都会默认使用最新版本的流程定义。

我们应该创建第二个流程，在Activiti中，如下，定义并且部署它，该流程定义会添加到流程定义表中。

```
<definitions id="myNewDefinitions" >
  <process id="myNewProcess" name="My important process" >
    ...
  </process>
</definitions>
```

这个表结构看起来像这样：

表 6.3.

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

注意：为何新流程的key与我们的第一个流程是不同的？尽管流程定义的名称是相同的（当然，我们应该也是可以改变这一点的），Activiti仅仅只考虑id属性判断流程。因此，新的流程定义部署的版本号为1。

6. 4. #提供流程图片

流程定义的流程图可以被添加到部署中，该流程图将会持久化到Activiti所使用的数据库中并且可以通过Activiti的API进行访问。该流程图也可以被用来在Activiti Explorer控制台中的流程中进行显示。

如果在我们的类路径下面有一个流程，`org/activiti/expenseProcess.bpmn20.xml`，该流程定义有一个流程key 'expense'。以下遵循流程定义图片的命名规范（按照这个特地顺序）：

- 如果在部署时一个图片资源已经存在，它是BPMN2.0的XML文件名后面是流程定义的key并且是一个图片的后缀。那么该图片将被使用。在我们的例子中，这应该是 `org/activiti/expenseProcess.expense.png`（或者 `jpg/gif`）。如果你在一个BPMN2.0 XML文件中定义多个流程定义图片，这种方式更有意义。每个流程定义图片的文件名中都将会有一个流程定义key。
- 如果没有这样的图片存在，部署的时候寻找与匹配BPMN2.0 XML 文件的名称的图片资源。在我们的例子中，这应该是 `org/activiti/expenseProcess.png`。注意：这意味着在同一个BPMN2.0 XML文件夹中的每个流程定义都会有相同的流程定义图片。因此，在每一个BPMN2.0 XML文件夹中仅仅只有一个流程定义，这绝对是不会有问题的。

当使用编程式的部署方式：

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

接下来，可以通过API来获取流程定义图片资源：

```
ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
    .processDefinitionKey("expense")
    .singleResult();

String diagramResourceName = processDefinition.getDiagramResourceName();
InputStream imageStream = repositoryService.getResourceAsStream(processDefinition.getDeploymentId(), diagramR
```

6. 5. #自动生成流程图片

在部署的情况下没有提供图片，在上一节中描述，如果流程定义中包含必要的‘图像交换’信息时，Activiti流程引擎竟会自动生成一个图像。

该资源可以按照部署时 提供流程图片完全相同的方式获取。



```
<bpmndi:BPMNShape bpmnElement="sid-52B0C487-F79C-4095-A1A1-1679FAD077C2"
    id="sid-52B0C487-F79C-4095-A1A1-1679FAD077C2_gui" isExpanded="true">
    <omgdc:Bounds height="160.0" width="313.0" x="930.0" y="440.0" />
</bpmndi:BPMNShape>
<bpmndi:BPMNShape bpmnElement="sid-8CA62404-2C30-4F92-8830-2374A967E1CE"
    id="sid-8CA62404-2C30-4F92-8830-2374A967E1CE_gui">
    <omgdc:Bounds height="30.0" width="30.0" x="960.0" y="505.0" />
</bpmndi:BPMNShape>
<bpmndi:BPMNShape bpmnElement="sid-87E17935-3E0C-49F3-9189-F783CA3E1FFD"
    id="sid-87E17935-3E0C-49F3-9189-F783CA3E1FFD_gui">
    <omgdc:Bounds height="80.0" width="100.0" x="1044.0" y="480.0" />
</bpmndi:BPMNShape>
<bpmndi:BPMNShape bpmnElement="sid-CF8AC45C-93A4-4EC5-9028-3289007FC3FE"
    id="sid-CF8AC45C-93A4-4EC5-9028-3289007FC3FE_gui">
    <omgdc:Bounds height="78.0" width="78.0" x="1144.0" y="500.0" />
</bpmndi:BPMNShape>
```



如果，因为某种原因，在部署的时候，并不需要或者不必要生成流程定义图片，那么就需要在流程引擎配置的属性中使用`isCreateDiagramOnDeploy`：

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

现在就不会生成流程定义图片。

6. 6. #类别

部署和流程定义都是用户定义的类别。流程定义类别在BPMN文件中属性的初始化的值<definitions ... targetNamespace="yourCategory" ...>

部署类别是可以通过直接使用API进行指定的看起来想这样：

```
repositoryService
    .createDeployment()
    .category("yourCategory")
    ...
    .deploy();
```

第#7#章#BPMN 2.0介绍

7.1. #啥是BPMN?

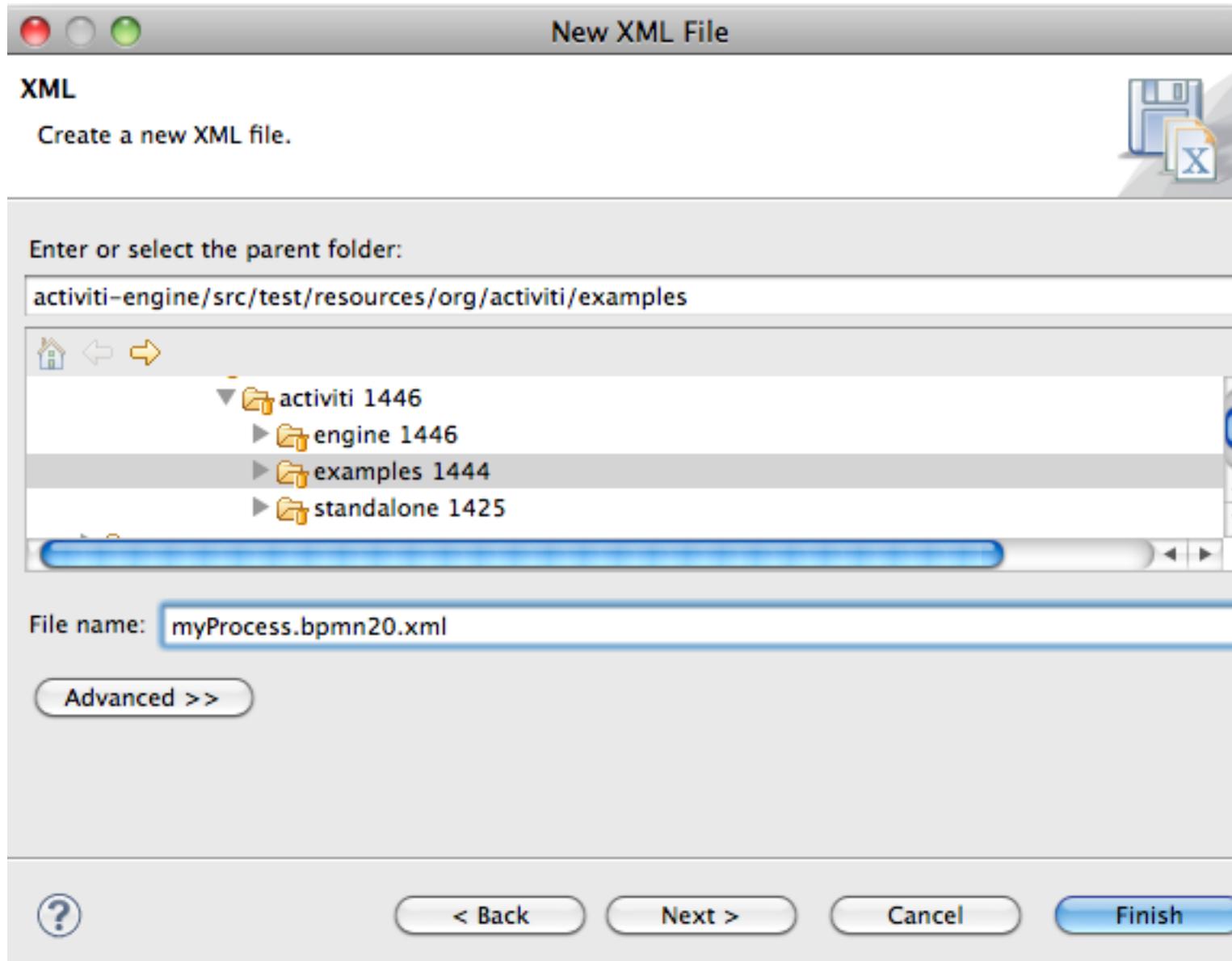
参考我们的FAQ中的BPMN 2.0部分 [<http://activiti.org/faq.html#WhatIsBpmn20>]。

7.2. #定义一个流程

注意

文章假设你在使用Eclipse IDE [<http://eclipse.org/>] 来创建和编辑文件。不过，其中只用到了Eclipse很少的特性。你可以使用喜欢的任何工具来创建包含BPMN 2.0的xml文件。

创建一个新的XML文件（右击任何项目选择“新建”->“其他”->“XML-XML文件”）并命名。确认文件后缀为ends with .bpmn20.xml 或 .bpmn，否则引擎无法发布。



BPMN 2.0根节点是definitions节点。这个元素中，可以定义多个流程定义（不过我们建议每个文件只包含一个流程定义，可以简化开发过程中的维护难度）。一个空的流程定义

看起来像下面这样。注意，definitions元素 最少也要包含xmlns 和 targetNamespace的声明。targetNamespace可以是任意值，它用来对流程实例进行分类。

```
<definitions
    xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
    xmlns:activiti="http://activiti.org/bpmn"
    targetNamespace="Examples">

    <process id="myProcess" name="My First Process">
        ..
    </process>

</definitions>
```

你也可以选择添加线上的BPMN 2.0格式位置， 下面是ecipse中的xml配置。

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

process元素有两个属性：

- id: 这个属性是必须的， 它对应着Activiti ProcessDefinition对象的key属性。 ‘id’可以用 来启动流程定义的流程实例， 通过RuntimeService的startProcessInstanceByKey方法。 这个方法会一直使用最新发布版本的流程定义(译者注：实际中一般都使用这种方式启动流程)。

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

注意，它和startProcessInstanceId方法不同。 这个方法期望使用Activiti引擎在发布时 自动生成的id。, 可以通过调用processDefinition.getId()方法获得这个值。 生成的 id的格式为'key:version'， 最大长度限制为64个字符， 如果你在启动时抛出了一个ActivitiException，说明生成的id太长了， 需要限制流程的key的长度。

- name: 这个属性是可选的， 对应ProcessDefinition的name属性。 引擎自己不会使用这个属性，它可以用来在用户接口显示便于阅读的名称。

7.3. #快速起步：10分钟教程

这张我们会演示一个（非常简单）的业务流程，我们会通过它介绍一些基本的Activiti概念和API。

7.3.1. #前提

教程假设你已经能安装并运行Activiti demo，并且你使用了独立运行的H2服务器。修改db.properties，设置其中的 jdbc.url=jdbc:h2:tcp://localhost/activiti，然后根据H2的文档 [http://www.h2database.com/html/tutorial.html#using_server]启动独立服务器。

7.3.2. #目标

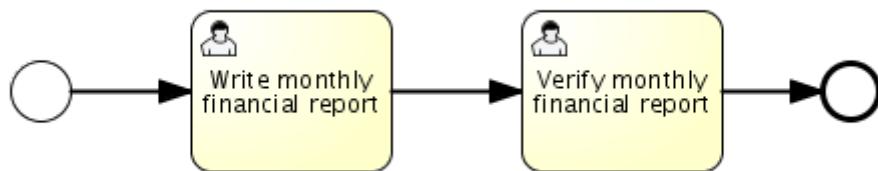
教程的目标是学习Activiti和一些基本的BPMN 2.0概念。 最终结果是一个简单的Java SE程序可以发布流程定义， 通过Activiti引擎API操作流程。 我们也会使用一些Activiti相关的工具。当然，我们在教程中所学的 也可以用于你构建自己的业务流程web应用。

7.3.3. #用例

用例很直接：我们有一个公司，就叫BPMCorp。在BPMCorp中，每个月都要给公司领导一个金融报表。由会计部门负责。当报表完成时，一个上级领导需要审批文档，然后才能发给所有领导。

7.3.4. #流程图

上面描述的业务流程可以用Activiti Designer 进行可视化设计。然后，为了这个教程，我们会手工编写XML，这样可以学到更多知识细节。我们流程的图形化BPMN 2.0标记看起来像这样：



我们看到有空开始事件（左侧圆圈），后面是两个用户任务：“制作月度财报”和“验证月度财报”，最后是空结束事件（右侧粗线圆圈）。

7.3.5. #XML内容

业务流程的XML内容（FinancialReportProcess.bpmn20.xml）如下所示：很容易找到流程的主要元素（点击链接可以了解BPMN 2.0结构的详细信息）：

- （空）开始事件 是我们流程的入口。
- 用户任务是流程中与操作者相关的任务声明。注意第一个任务分配给accountancy组，第二个任务分配给management组。参考用户任务分配章节 了解更多关于用户任务分配人员和群组的问题。
- 当流程达到空结束事件就会结束。
- 这些元素都使用连线连接。这些连线拥有source 和 target属性，定义了连线的方向。

```

<definitions id="definitions"
  targetNamespace="http://activiti.org/bpmn20"
  xmlns:activiti="http://activiti.org/bpmn"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL">

  <process id="financialReport" name="Monthly financial report reminder process">

    <startEvent id="theStart" />

    <sequenceFlow id='flow1' sourceRef='theStart' targetRef='writeReportTask' />

    <userTask id="writeReportTask" name="Write monthly financial report" >
      <documentation>
        Write monthly financial report for publication to shareholders.
      </documentation>
      <potentialOwner>
        <resourceAssignmentExpression>
          <formalExpression>accountancy</formalExpression>
        </resourceAssignmentExpression>
      </potentialOwner>
    </userTask>
  </process>
</definitions>
  
```

```

        </resourceAssignmentExpression>
    </potentialOwner>
</userTask>

<sequenceFlow id='flow2' sourceRef='writeReportTask' targetRef='verifyReportTask' />

<userTask id="verifyReportTask" name="Verify monthly financial report" >
    <documentation>
        Verify monthly financial report composed by the accountancy department.
        This financial report is going to be sent to all the company shareholders.
    </documentation>
    <potentialOwner>
        <resourceAssignmentExpression>
            <formalExpression>management</formalExpression>
        </resourceAssignmentExpression>
    </potentialOwner>
</userTask>

<sequenceFlow id='flow3' sourceRef='verifyReportTask' targetRef='theEnd' />

<endEvent id="theEnd" />

</process>

</definitions>

```

7.3.6. #启动一个流程实例

现在我们创建好了业务流程的流程定义。有了这个流程定义，我们可以创建流程实例了。这时，一个流程实例对应了特定月度财报的创建和审批。所有流程实例都共享同一个流程定义。

为了使用流程定义创建流程实例，首先要发布业务流程，这意味着两方面：

- 流程定义会保存到持久化的数据存储里，是为你的Activiti引擎特别配置。所以部署好你的业务流程，我们就能确认引擎重启后还能找到流程定义。
- BPMN 2.0流程文件会解析成内存对象模型，可以通过Activiti API操作。可以通过发布章节获得关于发布的更多信息。

就像章节里描述的一样，有很多种方式可以进行发布。一种方式是通过下面的API。注意所有与Activiti引擎的交互都是通过services。

```

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

```

现在我们可以启动一个新流程实例，使用我们定义在流程定义里的id（对应XML文件中的process元素）。注意这里的id对于Activiti来说，应该叫做key（译者注：一般在流程模型中使用的ID，在Activiti中都是Key，比如任务ID等...）。

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

这会创建一个流程实例，首先进入开始事件。开始事件之后，它会沿着所有的外出连线（这里只有一条）执行，到达第一个任务（“制作月度财报”）。Activiti会把一个任务保存到数据库里。这时，分配到这个任务的用户或群组会被解析，也会保存到数据库里。需要注意，Activiti引擎会继续执行流程的环节，除非遇到一个等待状态，比如用户任务。在

等待状态下，当前的流程实例的状态会保存到数据库中。直到用户决定完成任务才能改变这个状态。这时，引擎会继续执行，直到遇到下一个等待状态，或流程结束。如果中间引擎重启或崩溃，流程状态也会安全的保存在数据库里。

任务创建之后，`startProcessInstanceByKey`会在到达用户任务这个等待状态之后才会返回。这时，任务分配给了一个组，这意味着这个组是执行这个任务的候选组。

我们现在把所有东西都放在一起，来创建一个简单的java程序。创建一个eclipse项目，把Activiti的jar和依赖放到classpath下。（这些都可以在Activiti发布包的libs目录下找到）。在调用Activiti服务之前，我们必须构造一个ProcessEngine，它可以让我们访问服务。这里我们使用‘单独运行’的配置，这会使用demo安装时的数据库来构建ProcessEngine。

你可以在这里 [[images/FinancialReportProcess.bpmn20.xml](#)] 下载流程定义XML。这个文件包含了上面介绍的XML，也包含了必须的BPMN图像信息以便在Activiti工具中能编辑流程。

```
public static void main(String[] args) {

    // Create Activiti process engine
    ProcessEngine processEngine = ProcessEngineConfiguration
        .createStandaloneProcessEngineConfiguration()
        .buildProcessEngine();

    // Get Activiti services
    RepositoryService repositoryService = processEngine.getRepositoryService();
    RuntimeService runtimeService = processEngine.getRuntimeService();

    // Deploy the process definition
    repositoryService.createDeployment()
        .addClasspathResource("FinancialReportProcess.bpmn20.xml")
        .deploy();

    // Start a process instance
    runtimeService.startProcessInstanceByKey("financialReport");
}
```

7.3.7. #任务列表

我们现在可以通过TaskService来获得任务了，添加以下逻辑：

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

注意我们传入的用户必须是accountancy组的一个成员，要和流程定义中向对应：

```
<potentialOwner>
    <resourceAssignmentExpression>
        <formalExpression>accountancy</formalExpression>
    </resourceAssignmentExpression>
</potentialOwner>
```

我们也可以使用群组名称，通过任务查询API来获得相关的结果。现在可以在代码中添加如下逻辑：

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

因为我们配置的ProcessEngine使用了与demo相同的数据，我们可以登录到Activiti Explorer [<http://localhost:8080/activiti-explorer/>]。默认，accountancy（会计）组里没有任何人。使用kermit/kermit登录，点击组，并创建一个新组。然后点击用户，把组分配给fozzie。现在使用fozzie/fozzie登录，现在我们就可以启动我们的业务流程了，选择Processes页，在‘月度财报’的操作列点击‘启动流程’。

The screenshot shows the Activiti Explorer interface with the 'Processes' tab selected. On the left, there's a sidebar with several items: 'Deployed process definitions', 'Model workspace', 'firstline vs escalated', and 'report reminder process'. The main area displays a process definition titled 'Monthly financial report rendering' with a gear icon, indicating it's a process diagram. Below it, it says 'Version 1' and 'Deployed moments ago'.

和上面介绍的那样，流程会执行到第一个用户任务。因为我们以kermit登录，在启动流程实例之后，就可以看到有了一个新的待领任务。选择任务页来查看这条新任务。注意即使流程被其他人启动，任务还是会被会计组里的所有人作为一个候选任务看到。

The screenshot shows the Activiti Explorer interface with the 'Tasks' tab selected. At the top, there are buttons for 'Queued' (1), 'Involved' (0), and 'Archived' (0). A dropdown menu is open over the 'Accountancy' button, showing the count of tasks for each group: Accountancy (1), Engineering (0), Management (0), Marketing (0), and Sales (0).

7.3.8. #领取任务

现在一个会计要认领这个任务。认领以后，这个用户就会成为任务的执行人，任务会从会计组的其他成员的任务列表中消失。认领任务的代码如下所示：

```
taskService.claim(task.getId(), "fozzie");
```

任务会进入认领任务人的个人任务列表中。

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

在Activiti Explorer UI中，点击认领按钮，会执行相同的操作。任务会移动到登录用户的个人任务列表。你也会看到任务的执行人已经变成当前登陆的用户。

The screenshot shows the Activiti Explorer interface. At the top, there are tabs for 'Tasks', 'Processes', and 'Rules'. Below the tabs, there are buttons for 'My Tasks' (0), 'Queued' (1), 'Involved' (0), and 'Archived' (0). The 'Queued' button is highlighted with a red box. On the left, there's a list item for 'Monthly financial report'. On the right, a detailed view of a task titled 'Write monthly financial' is shown. This task has a due date of '31' (No due date), a priority of 'Medium Priority', and a claimable status. A red box highlights the 'Claim' button. Below the task details, sections for 'People' (No owner) and 'Subtasks' (No subtasks defined) are visible. At the bottom, a 'Complete task' button is shown.

7.3.9. #完成任务

现在会计可以开始进行财报的工作了。报告完成后，他可以完成任务，意味着任务所需的所有工作都完成了。

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

对于Activiti引擎，需要一个外部信息来让流程实例继续执行。任务会把自己从运行库中删除。流程会沿着单独一个外出连线执行，移动到第二个任务（‘审批报告’）。与第一个任务相同的机制会使用到第二个任务上，不同的是任务是分配给 management组。

在demo中，完成任务是通过点击任务列表中的完成按钮。因为Fozzie不是会计，我们先从Activiti Explorer注销然后使用kermit登陆（他是经理）。第二个任务会进入未分配任务列表。

7.3.10. #结束流程

审批任务可以像之前介绍的一样查询和领取。完成第二个任务会让流程执行到结束事件，就会结束流程实例。流程实例和所有相关的运行数据都会从数据库中删除。

登录Activiti Explorer就可以进行验证，可以看到保存流程运行数据的表中已经没有数据了。

The screenshot shows the Activiti Explorer interface. The top navigation bar includes 'Tasks' and 'Processes' buttons. Below the navigation, there are tabs for 'Active Processes', 'Suspended Processes', 'Jobs', 'Users', and 'Groups'. On the left, a sidebar lists categories: 'T (3)', 'I (1)', 'SCR (0)', '(0)', and 'K (0)'. The main content area displays a database table named 'ACT_RU_EXECUTION' with a database icon. A message below the table states 'Table contains no rows.'.

通过程序，你也可以使用historyService判断流程已经结束了。

```
HistoryService historyService = processEngine.getHistoryService();
```

```
HistoricProcessInstance historicProcessInstance =
historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());
```

7.3.11. #代码总结

把上述代码组合在一起，获得的代码如下所示 （这些代码考虑到你可能会在Activiti Explorer UI中启动一些流程实例。这样，它会获得多个任务，而不是一个，所以代码可以一直正常运行）：

```
public class TenMinuteTutorial {

    public static void main(String[] args) {

        // Create Activiti process engine
        ProcessEngine processEngine = ProcessEngineConfiguration
            .createStandaloneProcessEngineConfiguration()
            .buildProcessEngine();

        // Get Activiti services
        RepositoryService repositoryService = processEngine.getRepositoryService();
        RuntimeService runtimeService = processEngine.getRuntimeService();

        // Deploy the process definition
        repositoryService.createDeployment()
            .addClasspathResource("FinancialReportProcess.bpmn20.xml")
            .deploy();

        // Start a process instance
        String procId = runtimeService.startProcessInstanceByKey("financialReport").getId();

        // Get the first task
        TaskService taskService = processEngine.getTaskService();
        List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("accountancy").list();
        for (Task task : tasks) {
            System.out.println("Following task is available for accountancy group: " + task.getName());

            // claim it
            taskService.claim(task.getId(), "fozzie");
        }

        // Verify Fozzie can now retrieve the task
        tasks = taskService.createTaskQuery().taskAssignee("fozzie").list();
        for (Task task : tasks) {
            System.out.println("Task for fozzie: " + task.getName());

            // Complete the task
            taskService.complete(task.getId());
        }

        System.out.println("Number of tasks for fozzie: "
            + taskService.createTaskQuery().taskAssignee("fozzie").count());

        // Retrieve and claim the second task
        tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();
        for (Task task : tasks) {
            System.out.println("Following task is available for accountancy group: " + task.getName());
            taskService.claim(task.getId(), "kermit");
        }
    }
}
```

```
// Completing the second task ends the process
for (Task task : tasks) {
    taskService.complete(task.getId());
}

// verify that the process is actually finished
HistoryService historyService = processEngine.getHistoryService();
HistoricProcessInstance historicProcessInstance =
    historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());
}

}
```

这段代码包含在实例中的一个单元测试中（似的，你可以运行单元测试来测试你的流程。参考单元测试章节来了解更多信息）。

7.3.12. #更多思考

可以看到业务流程相对于现实来说太简单了。 然而，你可以了解Activiti中的BPMN 2.0结构， 你可以考虑对业务流程进行以下方面的加强：

- 定义网关来实现决策环节。 这样，经理可以驳回财报， 重新给会计创建一个任务。
- 考虑使用变量， 这样我们可以保存或引用报告， 把它显示到表单中。
- 在流程最后加入服务任务， 把报告发给每个领导。
- 其他

第#8#章#BPMN 2.0结构

本章介绍Activiti支持的BPMN 2.0结构，以及对BPMN标准的扩展。

8.1. #自定义扩展

BPMN 2.0标准对于各方都是一个好东西。最终用户不用担心会绑死在供应商提供的专有解决方案上。框架，特别是activiti这样的开源框架，可以提供相同功能（甚至是更好的实现），足以和大的供应商媲美。按照BPMN 2.0标准，从大供应商的解决方案迁移到activiti只会经过一个简单而平滑的过程。

标准不好的一点是，它常常是不同公司之间大量讨论和妥协的结果。（而且通常是愿景）。作为开发者去阅读流程定义的BPMN 2.0 xml时，有时会感觉用这种结构和方法去做事太麻烦了。因此activiti把简化开发作为最优先的事情，我们会使用一些被称为‘Activiti BPMN 扩展’的功能。这些扩展是新的结构或方法来简化对应的结构，它们并不属于BPMN 2.0规范。

虽然BPMN 2.0规范清楚的指明了如何开发自定义扩展，但是我们还要确认一下几点：

- 自定义扩展的前提是总有简单的方法转换成标准方法。所以当你决定使用自定义扩展时，不用担心没办法回头。
- 当使用自定义扩展时，总会清楚的指明使用了新的XML元素，属性，等等。比如会使用activiti:命名空间前缀。
- 这些扩展的目标是最终把它们加入到下一版本的BPMN规范中，或者至少可以引起对特定BPMN结构的讨论。

因此无论是是否想要使用自定义扩展，这都取决于你。很多因素会影响决定这个决定（图形编辑器，公司策略，等等）。只是因为我们相信标准里的一些功能可以更简单或更高校，所以才决定提供自定义扩展。请对扩展给予我们（正面或负面）的评价，或者是对自定义扩展的心想法。说不定有一天你的想法就会加入到规范中。

8.2. #事件 (Event)

事件用来表明流程的生命周期中发生了什么事。事件总是画成一个圆圈。在BPMN 2.0中，事件有两大分类：捕获 (catching) 或 触发 (throwing) 事件。

- 捕获 (Catching)：当流程执行到事件，它会等待被触发。触发的类型是由内部图表或XML中的类型声明来决定的。捕获事件与触发事件在显示方面是根据内部图表是否被填充来区分的（白色的）。
- 触发 (Throwing)：当流程执行到事件，会触发一个事件。触发的类型是由内部图表或XML中的类型声明来决定的。触发事件与捕获事件在显示方面是根据内部图表是否被填充来区分的（被填充为黑色）。

8.2.1. #事件定义

事件定义决定了事件的语义。如果没有事件定义，这个事件就做什么特别的事情。没有设置事件定义的开始事件不会在启动流程时做任何事情。如果给开始事件添加了一个事件定义

(比如定时器事件定义) 我们就声明了开始流程的事件 “类型” (这时定时器事件监听器会在某个时间被触发)。

8.2.2. #定时器事件定义

定时器事件是根据指定的时间触发的事件。可以用于 开始事件、中间事件 或 边界事件

定时器定义必须下面介绍的一个元素：

- timeDate。使用 ISO 8601 [http://en.wikipedia.org/wiki/ISO_8601#Dates] 格式指定一个确定的时间，触发事件的时间。示例：

```
<timerEventDefinition>
  <timeDate>2011-03-11T12:13:14</timeDate>
</timerEventDefinition>
```

- timeDuration。指定定时器之前要等待多长时间，timeDuration可以设置为timerEventDefinition的子元素。 使用ISO 8601 [http://en.wikipedia.org/wiki/ISO_8601#Durations] 规定的格式 (由BPMN 2.0规定)。示例 (等待10天)。

```
<timerEventDefinition>
  <timeDuration>P10D</timeDuration>
</timerEventDefinition>
```

- timeCycle。指定重复执行的间隔，可以用来定期启动流程实例，或为超时时间发送多个提醒。timeCycle元素可以使用两种格式。第一种是 ISO 8601 [http://en.wikipedia.org/wiki/ISO_8601#Repeating_intervals] 标准的格式。示例 (重复3次，每次间隔10小时)：

```
<timerEventDefinition>
  <timeCycle>R3/PT10H</timeCycle>
</timerEventDefinition>
```

另外，你可以使用cron表达式指定timeCycle，下面的例子是从整点开始，每5分钟执行一次：

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

请参考教程 [<http://www.quartz-scheduler.org/docs/tutorials/crontrigger.html>] 来了解如何使用cron表达式。

注意：第一个数字表示秒，而不是像通常Unix cron中那样表示分钟。

重复的时间周期能更好的处理相对时间，它可以计算一些特定的时间点 (比如，用户任务的开始时间)，而cron表达式可以处理绝对时间 - 这对定时启动事件特别有用。

你可以在定时器事件定义中使用表达式，这样你就可以通过流程变量来影响那个定时器定义。 流程定义必须包含ISO 8601 (或cron) 格式的字符串，以匹配对应的时间类型。

```
<boundaryEvent id="escalationTimer" cancelActivity="true" attachedToRef="firstLineSupport">
  <timerEventDefinition>
```

```

<timeDuration>${duration}</timeDuration>
</timerEventDefinition>
</boundaryEvent>

```

注意： 只有启用job执行器之后， 定时器才会被触发。 (activiti.cfg.xml中的jobExecutorActivate需要设置为true， 不过， 默认job执行器是关闭的)。

8. 2. 3. #错误事件定义

错误事件是由指定错误触发的。

重要提醒： BPMN错误与Java异常完全不一样。 实际上， 他俩一点儿共同点都没有。 BPMN错误事件是为了对 业务异常建模。 Java异常是要 用特定方式处理。

错误事件定义会引用一个error元素。 下面是一个error元素的例子， 引用了一个错误声明：

```

<endEvent id="myErrorEndEvent">
  <errorEventDefinition errorRef="myError" />
</endEvent>

```

引用相同error元素的错误事件处理器会捕获这个错误。

8. 2. 4. #信号事件定义

信号事件会引用一个已命名的信号。 信号全局范围的事件（广播语义）。 会发送给所有激活的处理器。

信号事件定义使用signalEventDefinition元素。 signalRef属性会引用definitions根节点里定义的signal子元素。 下面是一个流程的实例， 其中会抛出一个信号，并被中间事件捕获。

```

<definitions... >
  <!-- declaration of the signal -->
  <signal id="alertSignal" name="alert" />

  <process id="catchSignal">
    <intermediateThrowEvent id="throwSignalEvent" name="Alert">
      <!-- signal event definition -->
      <signalEventDefinition signalRef="alertSignal" />
    </intermediateThrowEvent>
    ...
    <intermediateCatchEvent id="catchSignalEvent" name="On Alert">
      <!-- signal event definition -->
      <signalEventDefinition signalRef="alertSignal" />
    </intermediateCatchEvent>
    ...
  </process>
</definitions>

```

signalEventDefinition引用相同的signal元素。

8. 2. 4. 1. #触发信号事件

既可以通过bpmn节点由流程实例触发一个信号， 也可以通过API触发。 下面的org.activiti.engine.RuntimeService中的方法 可以用来手工触发一个信号。

```
RuntimeService.signalEventReceived(String signalName);
RuntimeService.signalEventReceived(String signalName, String executionId);
```

signalEventReceived(String signalName); 和 signalEventReceived(String signalName, String executionId); 之间的区别是 第一个方法会把信号发送给全局所有订阅的处理器（广播语义），第二个方法只把信息发送给指定的执行。

8.2.4.2. #捕获信号事件

信号事件可以被中间捕获信号事件或边界信息事件捕获。

8.2.4.3. #查询信号事件的订阅

可以查询所有订阅了特定信号事件的执行：

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

我们可以使用signalEventReceived(String signalName, String executionId)方法 吧信号发送给这些执行。

8.2.4.4. #信号事件范围

默认，信号会在流程引擎范围内进行广播。就是说，你可以在一个流程实例中抛出一个信号事件，其他不同流程定义的流程实例 都可以监听到这个事件。

然而，有时只希望在同一个流程实例中响应这个信号事件。比如一个场景是，流程实例中的同步机制，如果两个或更多活动是互斥的。

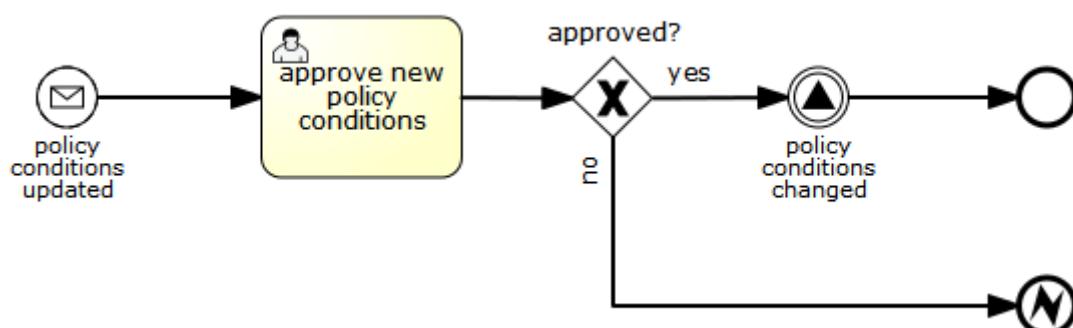
如果想要限制信号事件的范围，可以使用信号事件定义的scope 属性（不是BPMN2.0的标准属性）：

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

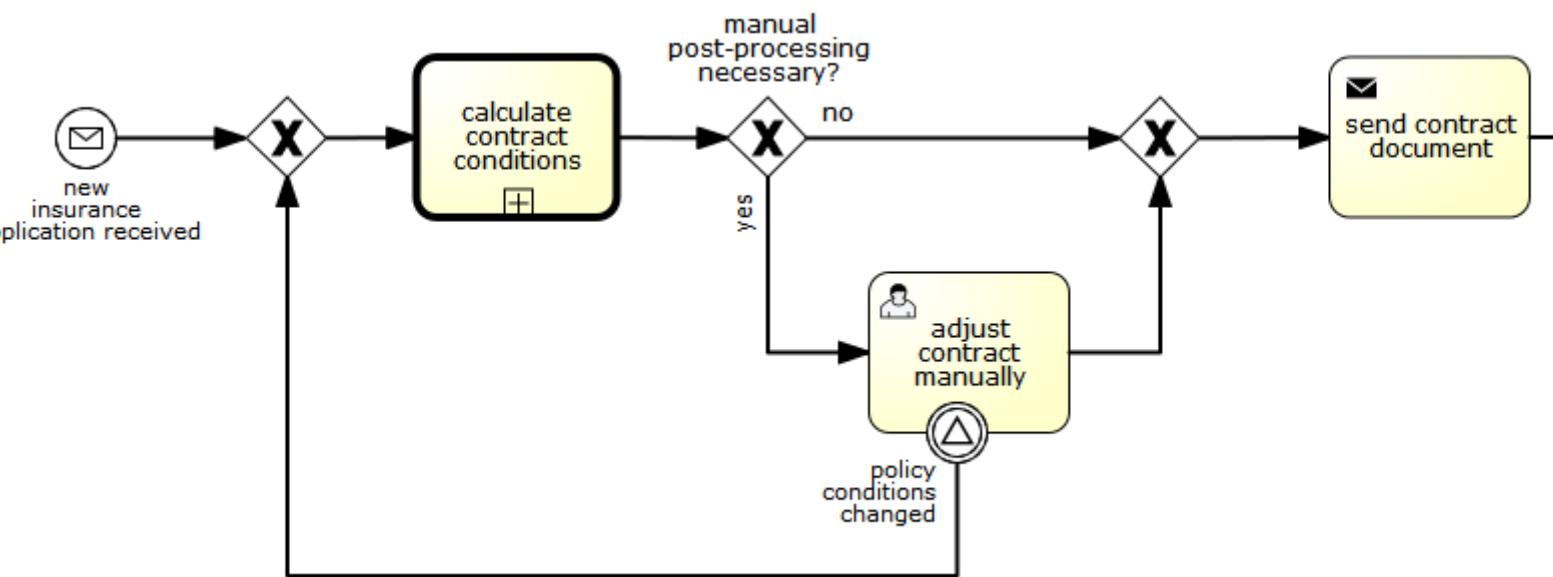
The default value for this is attribute is "global".

8.2.4.5. #信号事件实例

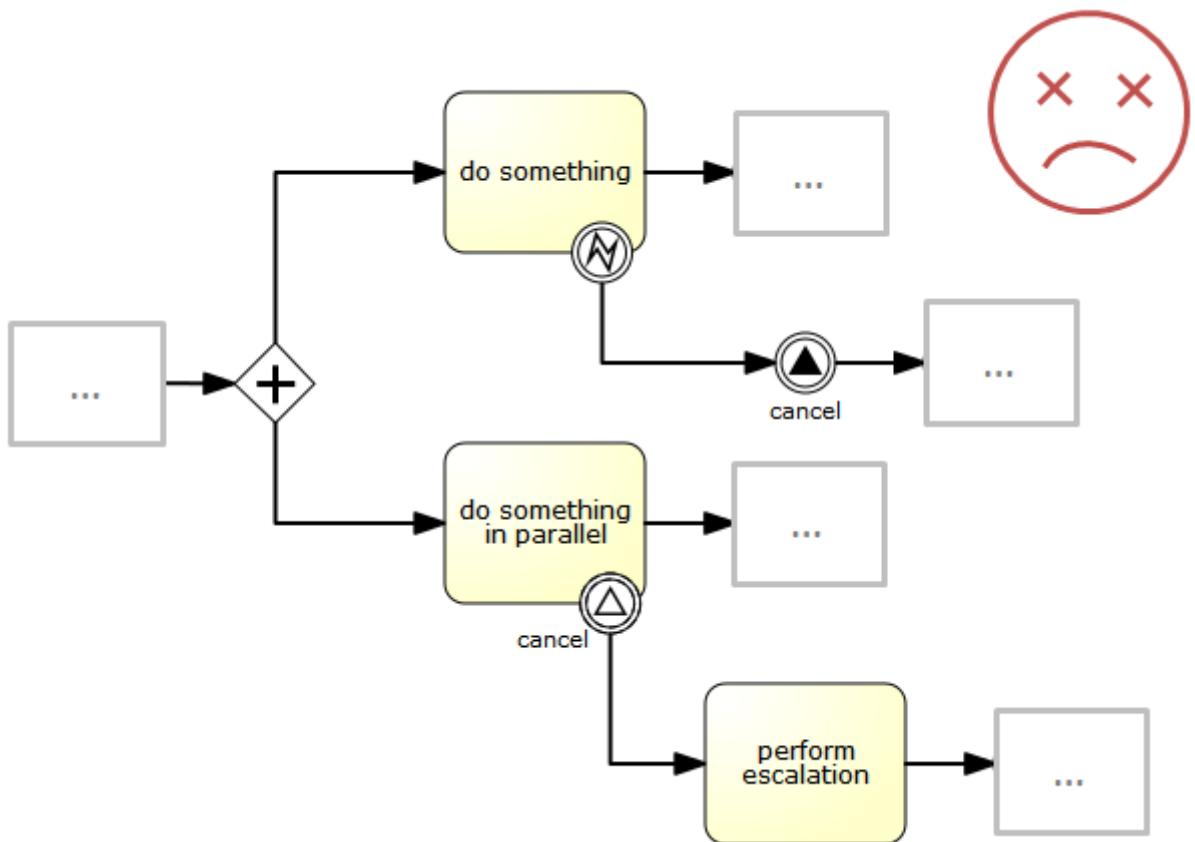
下面是两个不同流程使用信号交互的例子。第一个流程在保险规则更新或改变时启动。在修改被参与者处理时，会触发一个信息，通知规则改变：



这个时间会被所有感兴趣的流程实例捕获。下面是一个订阅这个事件的流程实例。



注意：要了解信号事件是广播给所有 激活的处理器的。 这意味着在上面的例子中，所有流程实例都会接收到这个事件。 这就是我们想要的。然而，有的情况下并不想要这种广播行为。 考虑下面的流程：



上述流程描述的模式activiti并不支持。这种想法是执行“do something”任务时出现的错误，会被边界错误事件捕获， 然后使用信号传播给并发路径上的分支，进而中断“do something in parallel”任务。 目前，activiti实际运行的结果与期望一致。信号会传播给边界事件并中断任务。但是，根据信号的广播含义，它也会传播给所有其他订阅了信号事件的流程实例。 所以，这就不是我们想要的结果。

注意： 信号事件不会执行任何与特定流程实例的联系。 如果你只想把一个信息发给指定的流程实例，需要手工关联，再使用 `signalEventReceived(String signalName, String executionId)` 和对应的 查询机制。

8. 2. 5. #消息事件定义

消息事件会引用一个命名的消息。每个消息都有名称和内容。和信号不同，消息事件总会直接发送给一个接受者。

消息事件定义使用`messageEventDefinition`元素。`messageRef`属性引用了`definitions`根节点下的一个`message`子元素。下面是一个使用两个消息事件的流程例子，开始事件和中间捕获事件分别声明和引用了两个消息事件。

```
<definitions id="definitions"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:activiti="http://activiti.org/bpmn"
  targetNamespace="Examples"
  xmlns:tns="Examples">

  <message id="newInvoice" name="newInvoiceMessage" />
  <message id="payment" name="paymentMessage" />

  <process id="invoiceProcess">

    <startEvent id="messageStart" >
      <messageEventDefinition messageRef="newInvoice" />
    </startEvent>
    ...
    <intermediateCatchEvent id="paymentEvt" >
      <messageEventDefinition messageRef="payment" />
    </intermediateCatchEvent>
    ...
  </process>

</definitions>
```

8. 2. 5. 1. #触发消息事件

作为一个嵌入式的流程引擎，activiti不能真正接收一个消息。这些环境相关，与平台相关的活动比如连接到JMS（Java消息服务）队列或主题或执行WebService或REST请求。这个消息的接收是你要在应用或架构的一层实现的，流程引擎则内嵌其中。

在你的应用接收一个消息之后，你必须决定如何处理它。如果消息应该触发启动一个新流程实例，在下面的RuntimeService的两个方法中选择一个执行：

```
ProcessInstance startProcessInstanceByMessage(String messageName);
ProcessInstance startProcessInstanceByMessage(String messageName, Map<String, Object> processVariables);
ProcessInstance startProcessInstanceByMessage(String messageName, String businessKey, Map<String, Object> proce
```

这些方法允许使用对应的消息系统流程实例。

如果消息需要被运行中的流程实例处理，首先要根据消息找到对应的流程实例（参考下一节）然后触发这个等待中的流程。RuntimeService提供了如下方法可以基于消息事件的订阅来触发流程继续执行：

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

8.2.5.2. #查询消息事件的订阅

Activiti支持消息开始事件和中间消息事件。

- 消息开始事件的情况，消息事件订阅分配给一个特定的 process definition。这个消息订阅可以使用ProcessDefinitionQuery查询到：

```
ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
    .messageEventSubscription("newCallCenterBooking")
    .singleResult();
```

因为同时只能有一个流程定义关联到消息的订阅点，查询总是返回0或一个结果。如果流程定义更新了，那么只有最新版本的流程定义会订阅到消息事件上。

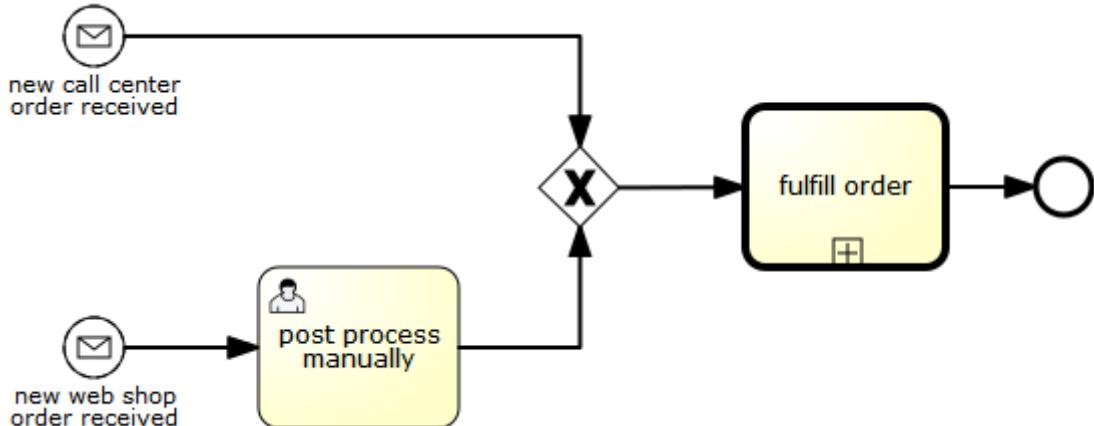
- 中间捕获消息事件的情况，消息事件订阅会分配给特定的执行。这个消息事件订阅可以使用ExecutionQuery查询到：

```
Execution execution = runtimeService.createExecutionQuery()
    .messageEventSubscriptionName("paymentReceived")
    .variableValueEquals("orderId", message.getOrderId())
    .singleResult();
```

这个查询可以调用对应的查询，通常是流程相关的信息（这里，最多只能有一个流程实例对应着orderId）。

8.2.5.3. #消息事件实例

下面是一个使用两个不同消息启动的流程实例：



可以在，流程需要不同的方式来区分开始事件，而后最终会进入同样的路径。

8.2.6. #开始事件

开始事件用来指明流程在哪里开始。开始事件的类型（流程在接收事件时启动，还是在指定时间启动，等等），定义了流程如何启动，这通过事件中不同的小图表来展示。在XML中，这些类型是通过声明不同的子元素来区分的。

开始事件都是捕获事件：最终这些事件都是（一直）等待着，直到对应的触发时机出现。

在开始事件中，可以设置下面的activiti特定属性：

- initiator: 当流程启动时, 把当前登录的用户保存到哪个变量名中。示例如下:

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

登录的用户必须使用IdentityService.setAuthenticatedUserId(String)方法设置, 并像这样包含在try-finally代码中:

```
try {
    identityService.setAuthenticatedUserId("bono");
    runtimeService.startProcessInstanceByKey("someProcessKey");
} finally {
    identityService.setAuthenticatedUserId(null);
}
```

这段代码来自Activiti Explorer, 所以它可以和 ???一起结合使用。

8.2.7. #空开始事件

8.2.7.1. #描述

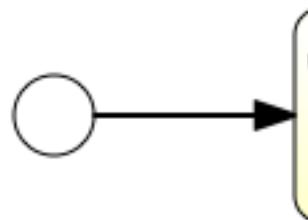
空开始事件技术上意味着没有指定启动流程实例的触发条件。这就是说引擎不能预计什么时候流程实例会启动。空开始事件用于, 当流程实例要通过API启动的场景, 通过调用startProcessInstanceByXXX方法。

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

注意: 子流程都有一个空开始事件。

8.2.7.2. #图形标记

空开始事件显示成一个圆圈, 没有内部图表(没有触发类型)



8.2.7.3. #XML结构

空开始事件的XML结构是普通的开始事件定义, 没有任何子元素 (其他开始事件类型都有一个子元素来声明自己的类型)

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

8.2.7.4. #空开始事件的自定义扩展

formKey: 引用用户在启动新流程实例时需要填写的表单模板, 更多信息可以参考表单章节。实例:

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

8. 2. 8. #定时开始事件

8. 2. 8. 1. #描述

定时开始事件用来在指定的时间创建流程实例。 它可以同时用于只启动一次的流程 和应该在特定时间间隔启动多次的流程。

注意：子流程不能使用定时开始事件。

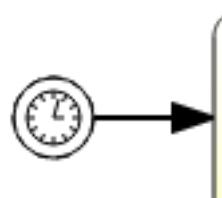
注意：定时开始事件在流程发布后就会开始计算时间。 不需要调用

startProcessInstanceByXXX，虽然也而已调用启动流程的方法， 但是那会导致调用 startProcessInstanceByXXX时启动过多的流程。

注意：当包含定时开始事件的新版本流程部署时， 对应的上一个定时器就会被删除。这是因为通常不希望自动启动旧版本流程的流程实例。

8. 2. 8. 2. #图形标记

定时开始事件显示为了一个圆圈， 内部是一个表。



8. 2. 8. 3. #XML内容

定时开始事件的XML内容是普通开始事件的声明， 包含一个定时定义子元素。 请参考定时定义 查看配合细节。

示例：流程会启动4次， 每次间隔5分钟， 从2011年3月11日， 12:13开始计时。

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

示例：流程会根据选中的时间启动一次。

```
<startEvent id="theStart">
  <timerEventDefinition>
    <timeDate>2011-03-11T12:13:14</timeDate>
  </timerEventDefinition>
</startEvent>
```

8. 2. 9. #消息开始事件

8. 2. 9. 1. #描述

消息开始事件可以用其使用一个命名的消息来启动流程实例。 这样可以帮助我们使用消息名称来选择正确的开始事件。

在发布包含一个或多个消息开始事件的流程定义时，需要考虑下面的条件：

- 消息开始事件的名称在给定流程定义中不能重复。流程定义不能包含多个名称相同的消息开始事件。如果两个或以上消息开始事件应用了相同的事件，或两个或以上消息事件引用的消息名称相同，activiti会在发布流程定义时抛出异常。
- 消息开始事件的名称在所有已发布的流程定义中不能重复。如果一个或多个消息开始事件引用了相同名称的消息，而这个消息开始事件已经部署到不同的流程定义中，activiti就会在发布时抛出一个异常。
- 流程版本：在发布新版本的流程定义时，之前订阅的消息订阅会被取消。如果新版本中没有消息事件也会这样处理。

启动流程实例，消息开始事件可以使用下列RuntimeService中的方法来触发：

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

这里的messageName是messageEventDefinition的messageRef属性引用的message元素的name属性。启动流程实例时，要考虑一下因素：

- 消息开始事件只支持顶级流程。消息开始事件不支持内嵌子流程。
- 如果流程定义有多个消息开始事件，`runtimeService.startProcessInstanceByMessage(...)`会选择对应的开始事件。
- 如果流程定义有多个消息开始事件和一个空开始事件。
`runtimeService.startProcessInstanceByKey(...)`和`runtimeService.startProcessInstanceById(...)`会使用空开始事件启动流程实例。
- 如果流程定义有多个消息开始事件，而且没有空开始事件，
`runtimeService.startProcessInstanceByKey(...)`和`runtimeService.startProcessInstanceById(...)`会抛出异常。
- 如果流程定义只有一个消息开始事件，
`runtimeService.startProcessInstanceByKey(...)`和`runtimeService.startProcessInstanceById(...)`会使用这个消息开始事件启动流程实例。
- 如果流程被调用环节（callActivity）启动，消息开始事件只支持如下情况：
 - 在消息开始事件以外，还有一个单独的空开始事件
 - 流程只有一个消息开始事件，没有空开始事件。

8.2.9.2. #图形标记

消息开始事件是一个圆圈，中间是一个消息事件图标。图标是白色未填充的，来表示捕获（接收）行为。



8.2.9.3. #XML内容

消息开始事件的XML内容时在普通开始事件申请中包含一个 messageEventDefinition子元素：

```

<definitions id="definitions"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:activiti="http://activiti.org/bpmn"
  targetNamespace="Examples"
  xmlns:tns="Examples">

  <message id="newInvoice" name="newInvoiceMessage" />

  <process id="invoiceProcess">

    <startEvent id="messageStart" >
      <messageEventDefinition messageRef="tns:newInvoice" />
    </startEvent>
    ...
  </process>
</definitions>
```

8.2.10. #信号开始事件

8.2.10.1. #描述

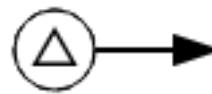
signal开始事件，可以用来通过一个已命名的信号（signal）来启动一个流程实例。信号可以在流程实例内部使用“中间信号抛出事务”触发，也可以通过API(runtimService.signalEventReceivedXXX方法)触发。两种情况下，所有流程实例中拥有相同名称的signalStartEvent都会启动。

注意，在两种情况下，都可以选择同步或异步的方式启动流程实例。

必须向API传入signalName，这是signal元素的name属性值，它会被signalEventDefinition的signalRef属性引用。

8.2.10.2. #图形标记

信号开始事件显示为一个中间包含信号事件图标的圆圈。标记是无填充的，表示捕获（接收）行为。



8.2.10.3. #XML格式

signalStartEvent的XML格式是标准的startEvent声明，其中包含一个signalEventDefinition子元素：

```
<signal id="theSignal" name="The Signal" />

<process id="processWithSignalStart1">
    <startEvent id="theStart">
        <signalEventDefinition id="theSignalEventDefinition" signalRef="theSignal" />
    </startEvent>
    <sequenceFlow id="flow1" sourceRef="theStart" targetRef="theTask" />
    <userTask id="theTask" name="Task in process A" />
    <sequenceFlow id="flow2" sourceRef="theTask" targetRef="theEnd" />
    <endEvent id="theEnd" />
</process>
```

8.2.11. #错误开始事件

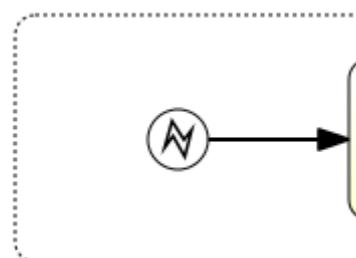
8.2.11.1. #描述

错误开始事件可以用来触发一个事件子流程。 错误开始事件不能用来启动流程实例。

错误开始事件都是中断事件。

8.2.11.2. #图形标记

错误开始事件是一个圆圈，包含一个错误事件标记。标记是白色未填充的，来表示捕获（接收）行为。



8.2.11.3. #XML内容

错误开始事件的XML内容是普通开始事件定义中，包含一个errorEventDefinition子元素。

```
<startEvent id="messageStart" >
    <errorEventDefinition errorRef="someError" />
```

```
</startEvent>
```

8. 2. 12. #结束事件

结束事件表示（子）流程（分支）的结束。 结束事件都是触发事件。 这是说当流程达到结束事件，会触发一个结果。 结果的类型是通过事件的内部黑色图标表示的。 在XML内容中，是通过包含的子元素声明的。

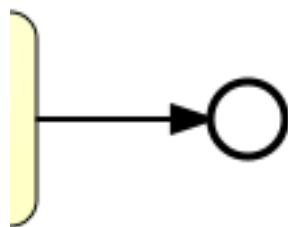
8. 2. 13. #空结束事件

8. 2. 13. 1. #描述

空结束事件意味着到达事件时不会指定抛出的结果。 这样，引擎会直接结束当前执行的分支，不会做其他事情。

8. 2. 13. 2. #图形标记

空结束事件是一个粗边圆圈，内部没有小图表（无结果类型）



8. 2. 13. 3. #XML内容

空结束事件的XML内容是普通结束事件定义，不包含子元素 （其他结束事件类型都会包含声明类型的子元素）。

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

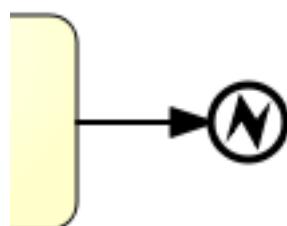
8. 2. 14. #错误结束事件

8. 2. 14. 1. #描述

当流程执行到错误结束事件， 流程的当前分支就会结束，并抛出一个错误。 这个错误可以被对应的中间边界错误事件捕获。 如果找不到匹配的边界错误事件，就会抛出一个异常。

8. 2. 14. 2. #图形标记

错误结束事件是一个标准的结束事件（粗边圆圈），内部有错误图标。 错误图表是全黑的，表示触发语法。



8. 2. 14. 3. #XML内容

错误结束事件的内容是一个错误事件， 子元素为errorEventDefinition。

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

errorRef属性引用定义在流程外部的error元素：

```
<error id="myError" errorCode="123" />
...
<process id="myProcess">
  ...
```

error的errorCode用来查找 匹配的捕获边界错误事件。 如果errorRef与任何error都不匹配， 就会使用errorRef来作为errorCode的缩写。 这是activiti特定的缩写。 更具体的说， 见如下代码：

```
<error id="myError" errorCode="error123" />
...
<process id="myProcess">
  ...
    <endEvent id="myErrorEndEvent">
      <errorEventDefinition errorRef="myError" />
    </endEvent>
```

等同于

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

注意errorRef必须与BPMN 2.0格式相符， 必须是一个合法的QName。

8. 2. 15. #取消结束事件

[EXPERIMENTAL]

8. 2. 15. 1. #描述

取消结束事件只能与BPMN事务子流程结合使用。 当到达取消结束事件时， 会抛出取消事件， 它必须被取消边界事件捕获。 取消边界事件会取消事务，并触发补偿机制。

8. 2. 15. 2. #图形标记

取消结束事件显示为标准的结束事件（粗边圆圈）， 包含一个取消图标。 取消图标是全黑的， 表示触发语法。



8. 2. 15. 3. #XML内容

取消结束事件内容是一个结束事件， 包含cancelEventDefinition子元素。

```
<endEvent id="myCancelEndEvent">
    <cancelEventDefinition />
</endEvent>
```

8. 2. 16. #边界事件

边界事件都是捕获事件，它会附在一个环节上。（边界事件不可能触发事件）。这意味着，当节点运行时，事件会监听对应的触发类型。当事件被捕获，节点就会中断，同时执行事件的后续连线。

所以边界事件的定义方式都一样：

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

边界事件使用如下方式进行定义：

- 唯一标识（流程范围）
- 使用caught属性 引用事件衣服的节点。 注意边界事件和它们附加的节点在同一级别上。（比如，边界事件不是包含在节点内的）。
- 格式为XXXEventDefinition的XML子元素
(比如，TimerEventDefinition, ErrorEventDefinition, 等等) 定义了边界事件的类型。参考对应的边界事件类型，获得更多细节。

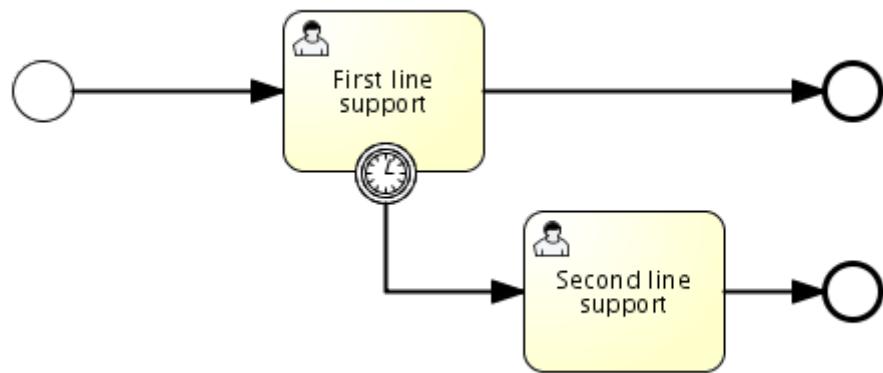
8. 2. 17. #定时边界事件

8. 2. 17. 1. #描述

定时边界事件就是一个暂停等待警告的时钟。当流程执行到绑定了边界事件的环节，会启动一个定时器。当定时器触发时（比如，一定时间之后），环节就会中断，并沿着定时边界事件的外出连线继续执行。

8. 2. 17. 2. #图形标记

定时边界事件是一个标准的边界事件（边界上的一个圆圈），内部是一个定时器小图标。



8.2.17.3. #XML内容

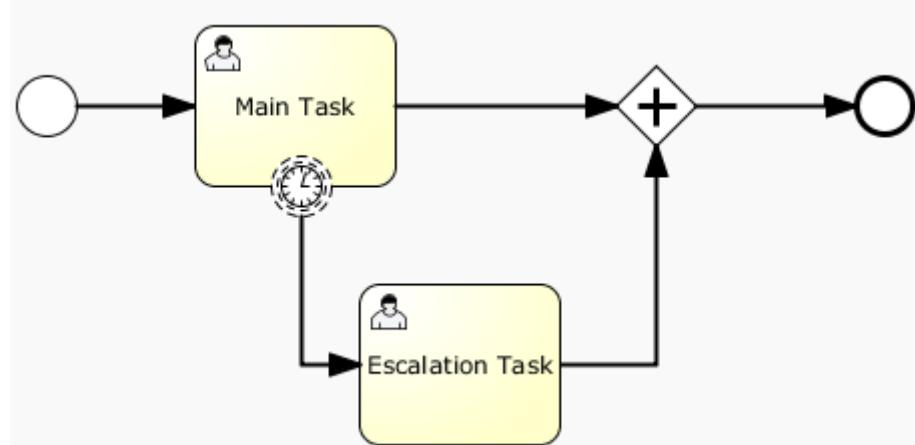
定时器边界任务定义是一个正规的边界事件。 指定类型的子元素是timerEventDefinition元素。

```

<boundaryEvent id="escalationTimer" cancelActivity="true" attachedToRef="firstLineSupport">
  <timerEventDefinition>
    <timeDuration>PT4H</timeDuration>
  </timerEventDefinition>
</boundaryEvent>
  
```

请参考定时事件定义获得更多定时其配置的细节。

在流程图中，可以看到上述例子中的圆圈边线是虚线：



经典场景是发送一个升级邮件，但是不打断正常流程的执行。

因为BPMN 2.0中，中断和非中断的事件还是有区别的。默认是中断事件。 非中断事件的情况，不会中断原始环节，那个环节还停留在原地。 对应的，会创建一个新分支，并沿着事件的流向继续执行。 在XML内容中，要把cancelActivity属性设置为false：

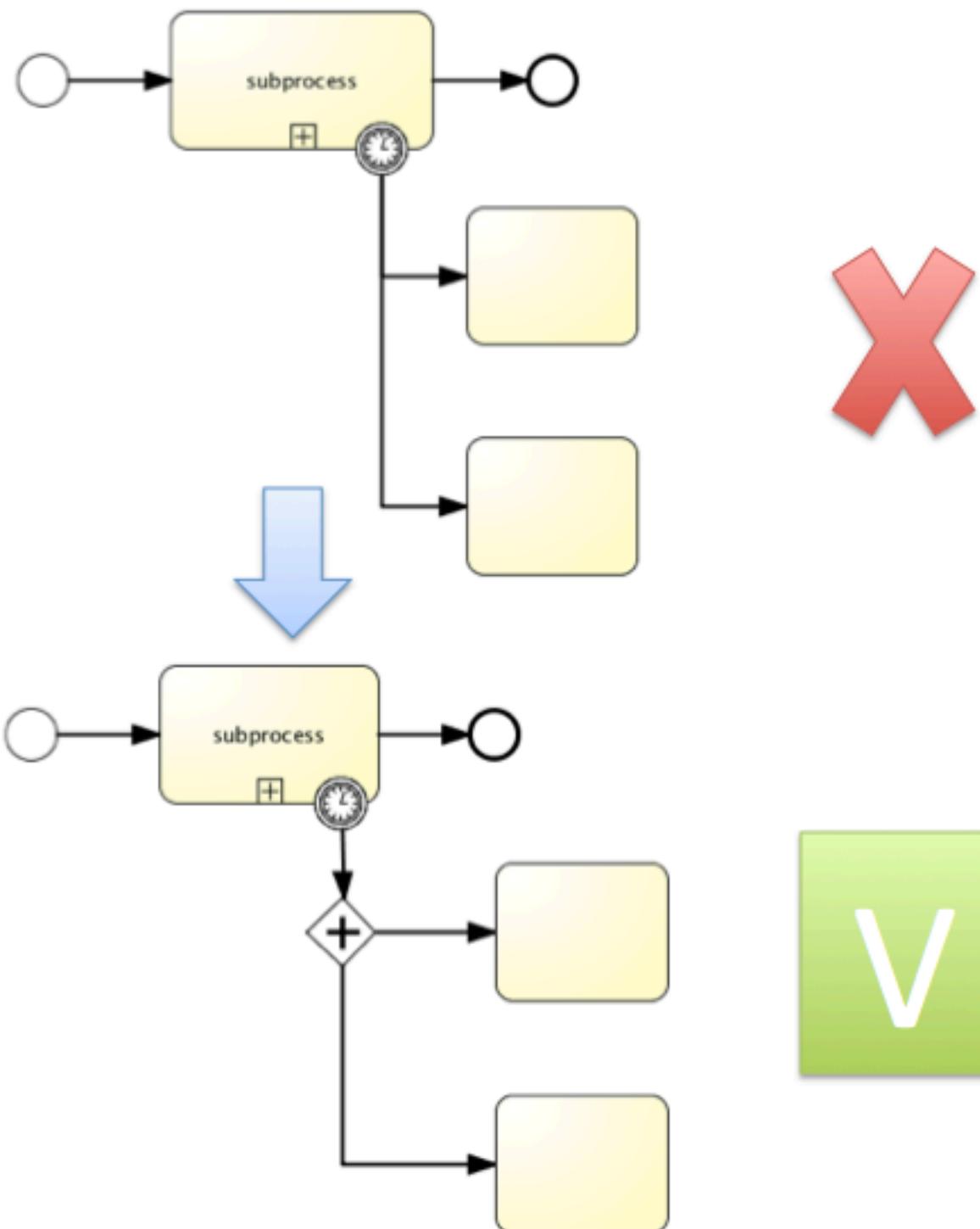
```

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

注意：边界定时事件只能在job执行器启用时使用。 （比如，把activiti.cfg.xml中的jobExecutorActivate 设置为true，因为默认job执行器默认是禁用的）。

8.2.17.4. #边界事件的已知问题

使用边界事件有一个已知的同步问题。 目前，不能在边界事件后面不能有多条外出连线（参考ACT-47 [<http://jira.codehaus.org/browse/ACT-47>]）。 解决这个问题的方法是在一个连线后使用并发网关。



8.2.18. #错误边界事件

8.2.18.1. #描述

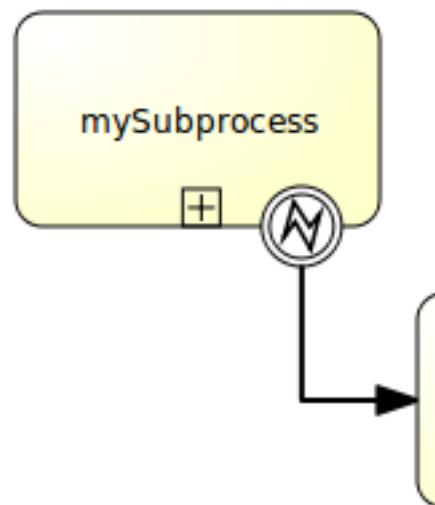
节点边界上的中间捕获错误事件，或简写成边界错误事件，它会捕获节点范围内抛出的错误。

定义一个边界错误事件，大多用于内嵌子流程，或调用节点，对于子流程的情况，它会为所有内部的节点创建一个作用范围。错误是由错误结束事件抛出的。这个错误会传递给上层作用域，直到找到一个错误事件定义向匹配的边界错误事件。

当捕获了错误事件时，边界任务绑定的节点就会销毁，也会销毁内部所有的执行分支（比如，同步节点，内嵌子流程，等等）。流程执行会继续沿着边界事件的外出连线继续执行。

8. 2. 18. 2. #图形标记

边界错误事件显示成一个普通的中间事件（圆圈内部有一个小圆圈）放在节点的标记上，内部有一个错误小图标。错误小图标是白色的，表示它是一个捕获事件。



8. 2. 18. 3. #Xml 内容

边界错误事件定义为普通的边界事件：

```

<boundaryEvent id=" catchError" attachedToRef="mySubProcess">
    <errorEventDefinition errorRef="myError"/>
</boundaryEvent>

```

和错误结束事件一样，errorRef引用了process元素外部的一个错误定义：

```

<error id="myError" errorCode="123" />
...
<process id="myProcess">
...

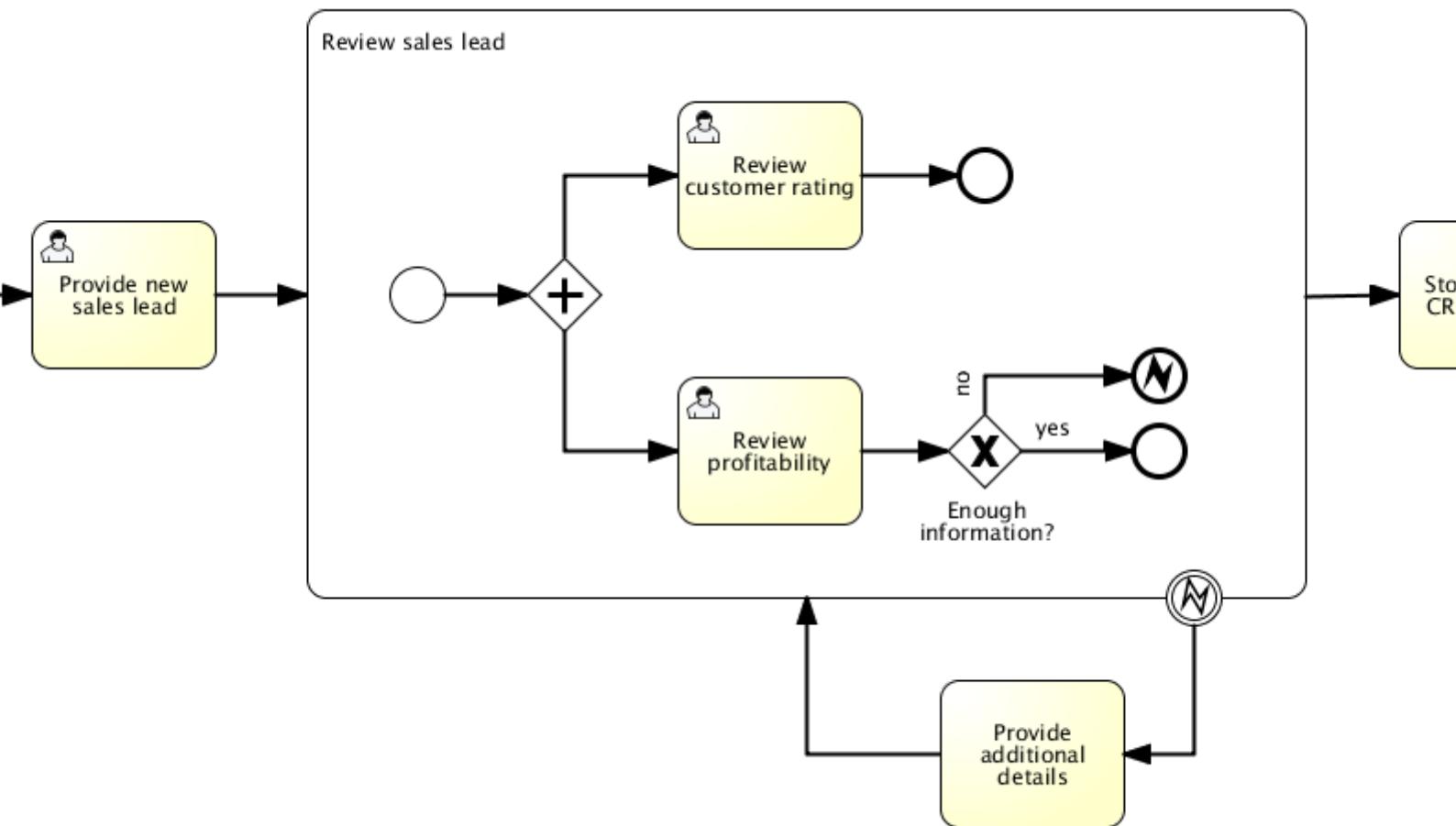
```

errorCode用来匹配捕获的错误：

- 如果没有设置errorRef，边界错误事件会捕获所有错误事件，无论错误的errorCode是什么。
- 如果设置了errorRef，并引用了一个已存的错误，边界事件就只捕获错误代码与之相同的错误。
- 如果设置了errorRef，但是BPMN 2.0中没有定义错误，errorRef就会当做errorCode使用（和错误结束事件的用法类似）。

8.2.18.4. #实例

下面的流程实例演示了如何使用错误结束事件。当完成‘审核盈利’这个用户任务是，如果没有提供足够的信息，就会抛出错误，错误会被子流程的边界任务捕获，所有‘回顾销售’子流程中的所有节点都会销毁。（即使‘审核客户比率’还没有完成），并创建一个‘提供更多信息’的用户任务。



这个流程也放在demo中了。流程XML和单元测试可以在org.activiti.examples.bpmn.event.error包下找到。

8.2.19. #信号边界事件

8.2.19.1. #描述

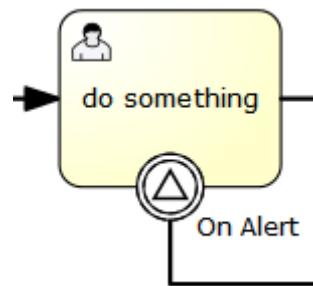
节点边界的中间捕获信号，或简称为边界信号事件，它会捕获信号定义引用的相同信号名的信号。

注意：与其他事件（比如边界错误事件）不同，边界信号事件不只捕获它绑定方位的信号。信号事件是一个全局的范围（广播语义），就是说信号可以在任何地方触发，即便是不同的流程实例。

注意：和其他事件（比如边界错误事件）不同，捕获信号后，不会停止信号的传播。如果你有两个信号边界事件，它们捕获相同的信号事件，两个边界事件都会被触发，即使它们在不同的流程实例中。

8. 2. 19. 2. #图形标记

边界信号事件显示为普通的中间事件（圆圈里有个小圆圈），位置在节点的边缘，内部有一个信号小图标。信号图标是白色的（未填充），来表示捕获的意思。



8. 2. 19. 3. #XML内容

边界信号事件定义为普通的边界事件：

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

8. 2. 19. 4. #实例

参考信号事件定义章节。

8. 2. 20. #消息边界事件

8. 2. 20. 1. #描述

节点边界上的中间捕获消息，或简称边界消息事件，根据引用的消息定义捕获相同消息名称的消息。

8. 2. 20. 2. #图形标记

边界消息事件显示成一个普通的中间事件（圆圈里有个小圆圈），位于节点边缘，内部是一个消息小图标。消息图标是白色（无填充），表示捕获语义。



注意，边界消息事件可能是中断（右侧）或非中断（左侧）的。

8. 2. 20. 3. #XML内容

边界消息事件定义为标准的边界事件：

```
<boundaryEvent id="boundary" attachedToRef="task" cancelActivity="true">
```

```
<messageEventDefinition messageRef="newCustomerMessage"/>
</boundaryEvent>
```

8. 2. 20. 4. #实例

参考消息事件定义章节。

8. 2. 21. #取消边界事件

[EXPERIMENTAL]

8. 2. 21. 1. #描述

在事务性子流程的边界上的中间捕获取消，或简称为边界取消事件 cancel event，当事务取消时触发。当取消边界事件触发时，首先中断当前作用域的所有执行。然后开始补偿事务内的所有激活的补偿边界事件。补偿是同步执行的。例如，离开事务钱，边界事务会等待补偿执行完毕。当补偿完成后，事务子流程会沿着取消边界事务的外出连线继续执行。

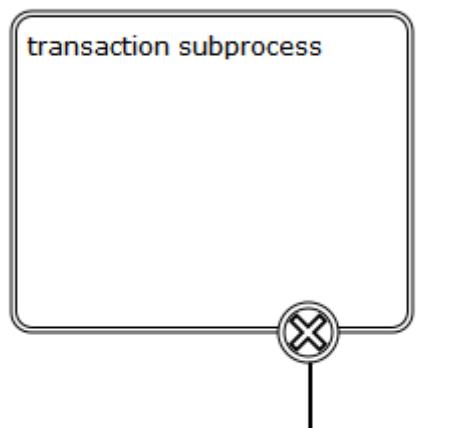
注意：每个事务子流程只能有一个取消边界事件。

注意：如果事务子流程包含内嵌子流程，补偿只会触发已经成功完成的子流程。

注意：如果取消边界子流程对应的事务子流程配置为多实例，如果一个实例触发了取消，就会取消所有实例。instances.

8. 2. 21. 2. #图形标记

取消边界事件显示为一个普通的中间事件（圆圈里套小圆圈），在节点的边缘，内部是一个取消图标。取消图标是白色（无填充），表明是捕获语义。



8. 2. 21. 3. #XML 内容

取消边界事件定义为普通边界事件：

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

因为取消边界事件都是中断的，所以不需要使用cancelActivity属性。

8. 2. 22. #补偿边界事件

[EXPERIMENTAL]

8. 2. 22. 1. #描述

节点边界的中间捕获补偿，或简称为补偿边界事件，可以用来设置一个节点的补偿处理器。

补偿边界事件必须使用直接引用设置唯一的补偿处理器。

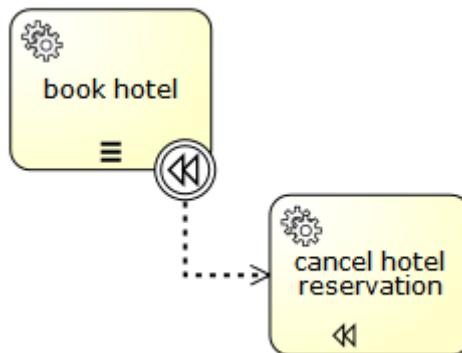
补偿边界事件与其他边界事件的策略不同。其他边界事件（比如信号边界事件）当到达关联的节点就会被激活。离开节点时，就会挂起，对应的事件订阅也会取消。补偿边界事件则不同。补偿边界事件在关联的节点成功完成时激活。当补偿事件触发或对应流程实例结束时，事件订阅才会删除。它遵循如下规则：

- 补偿触发时，补偿边界事件对应的补偿处理器会调用相同次数，根据它对应的节点的成功次数。
- 如果补偿边界事件关联到多实例节点，补偿事件会订阅每个实例。
- 如果补偿边界事件关联的节点中包含循环，补偿事件会在每次节点执行时进行订阅。
- 如果流程实例结束，订阅的补偿事件都会结束。

注意：补偿边界事件不支持内嵌子流程。

8. 2. 22. 2. #图形标记

补偿边界事件显示为标准中间事件（圆圈里套圆圈），位于节点边缘，内部有一个补偿小图标。补偿图标是白色的（无填充），表示捕获语义。另外，下面的图形演示了使用无方向的关联，为边界事件设置补偿处理器。



8. 2. 22. 3. #XML内容

补偿边界事件定义为标准边界事件：

```
<boundaryEvent id="compensateBookHotelEvt" attachedToRef="bookHotel" >
    <compensateEventDefinition />
</boundaryEvent>

<association associationDirection="One" id="a1" sourceRef="compensateBookHotelEvt" targetRef="undoBookHotel" />
```

```
<serviceTask id="undoBookHotel" isForCompensation="true" activiti:class="..." />
```

因为补偿边界事件在节点成功完成后激活， 所以不支持cancelActivity属性。

8. 2. 23. #中间捕获事件

所有中间捕获事件都使用同样的方式定义：

```
<intermediateCatchEvent id="myIntermediateCatchEvent" >
    <XXXEventDefinition/>
</intermediateCatchEvent>
```

中间捕获事件的定义包括

- 唯一标识（流程范围内）
- 一个结构为XXXEventDefinition的XML子元素（比如TimerEventDefinition等）定义了中间捕获事件的类型。参考特定的捕获事件类型，获得更多详情。

8. 2. 24. #定时中间捕获事件

8. 2. 24. 1. #描述

定时中间事件作为一个监听器。当执行到达捕获事件节点，就会启动一个定时器。当定时器触发（比如，一段时间之后），流程就会沿着定时中间事件的外出节点继续执行。

8. 2. 24. 2. #图形标记

定时器中间事件显示成标准中间捕获事件，内部是一个定时器小图标。



8. 2. 24. 3. #XML内容

定时器中间事件定义为标准中间捕获事件。指定类型的子元素为timerEventDefinition元素。

```
<intermediateCatchEvent id="timer">
    <timerEventDefinition>
        <timeDuration>PT5M</timeDuration>
    </timerEventDefinition>
</intermediateCatchEvent>
```

参考定时器事件定义了解配置信息。

8. 2. 25. #信号中间捕获事件

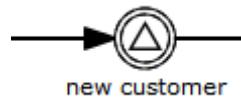
8. 2. 25. 1. #描述

中间捕获信号事件 通过引用信号定义来捕获相同信号名称的信号。

注意：与其他事件（比如错误事件）不同，信号不会在捕获之后被消费。如果你有两个激活的信号边界事件捕获相同的信号事件，两个边界事件都会被触发，即便它们在不同的流程实例中。

8. 2. 25. 2. #图形标记

中间信号捕获事件显示为一个普通的中间事件（圆圈套圆圈），内部有一个信号小图标。信号小图标是白色的（无填充），表示捕获语义。



8. 2. 25. 3. #XML内容

信号中间事件定义为普通的中间捕获事件。对应类型的子元素是signalEventDefinition元素。

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

8. 2. 25. 4. #实例

参考信号事件定义章节。

8. 2. 26. #消息中间捕获事件

8. 2. 26. 1. #描述

一个中间捕获消息事件，捕获特定名称的消息。

8. 2. 26. 2. #图形标记

中间捕获消息事件显示为普通中间事件（圆圈套圆圈），内部是一个消息小图标。消息图标是白色的（无填充），表示捕获语义。



8. 2. 26. 3. #XML内容

消息中间事件定义为标准中间捕获事件。指定类型的子元素是messageEventDefinition元素。

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

8. 2. 26. 4. #实例

参考消息事件定义章节。

8. 2. 27. #内部触发事件

所有内部触发事件的定义都是同样的：

```
<intermediateThrowEvent id="myIntermediateThrowEvent" >
    <XXXEventDefinition/>
</intermediateThrowEvent>
```

内部触发事件定义包含

- 唯一标识（流程范围）
- 使用格式为XXXEventDefinition的XML子元素（比如signalEventDefinition等）定义中间触发事件的类型。参考对应触发事件的类型，了解更多信息。

8. 2. 28. #中间触发空事件

下面的流程图演示了一个空中间触发事件的例子，它通常用于表示流程中的某个状态。



通过添加执行监听器，就可以很好地监控一些KPI。

```
<intermediateThrowEvent id="noneEvent" >
    <extensionElements>
        <activiti:executionListener class="org.activiti.engine.test.bpmn.event.IntermediateNoneEventTest$MyExecutionListener" />
    </extensionElements>
</intermediateThrowEvent>
```

这里你可以添加自己的代码，把事件发送给BAM工具或DWH。引擎不会为这个事件做任何事情，它直接径直通过。

8. 2. 29. #信号中间触发事件

8. 2. 29. 1. #描述

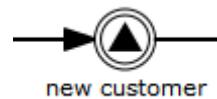
中间触发信号事件为定义的信号抛出一个信号事件。

在activiti中，信号会广播到所有激活的处理器中（比如，所以捕获信号事件）。信号可以通过同步和异步方式发布。

- 默认配置下，信号是同步发送的。就是说，抛出事件的流程实例会等到信号发送给所有捕获流程实例才继续执行。捕获流程实例也会在触发流程实例的同一个事务中执行，意味着如果某个监听流程出现了技术问题（抛出异常），所有相关的实例都会失败。
- 信号也可以异步发送。这时它会在到达抛出信号事件后决定哪些处理器是激活的。对这些激活的处理器，会保存一个异步提醒消息（任务），并发送给jobExecutor。

8. 2. 29. 2. #图形标记

中间信号触发事件显示为普通中间事件（圆圈套圆圈），内部又一个信号小图标。信号图标是黑色的（有填充），表示触发语义。



8. 2. 29. 3. #XML内容

消息中间事件定义为标准中间触发事件。 指定类型的子元素是signalEventDefinition元素。

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

异步信号事件如下所示:

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

8. 2. 29. 4. #实例

参考信号事件定义章节。

8. 2. 30. #补偿中间触发事件

[EXPERIMENTAL]

8. 2. 30. 1. #描述

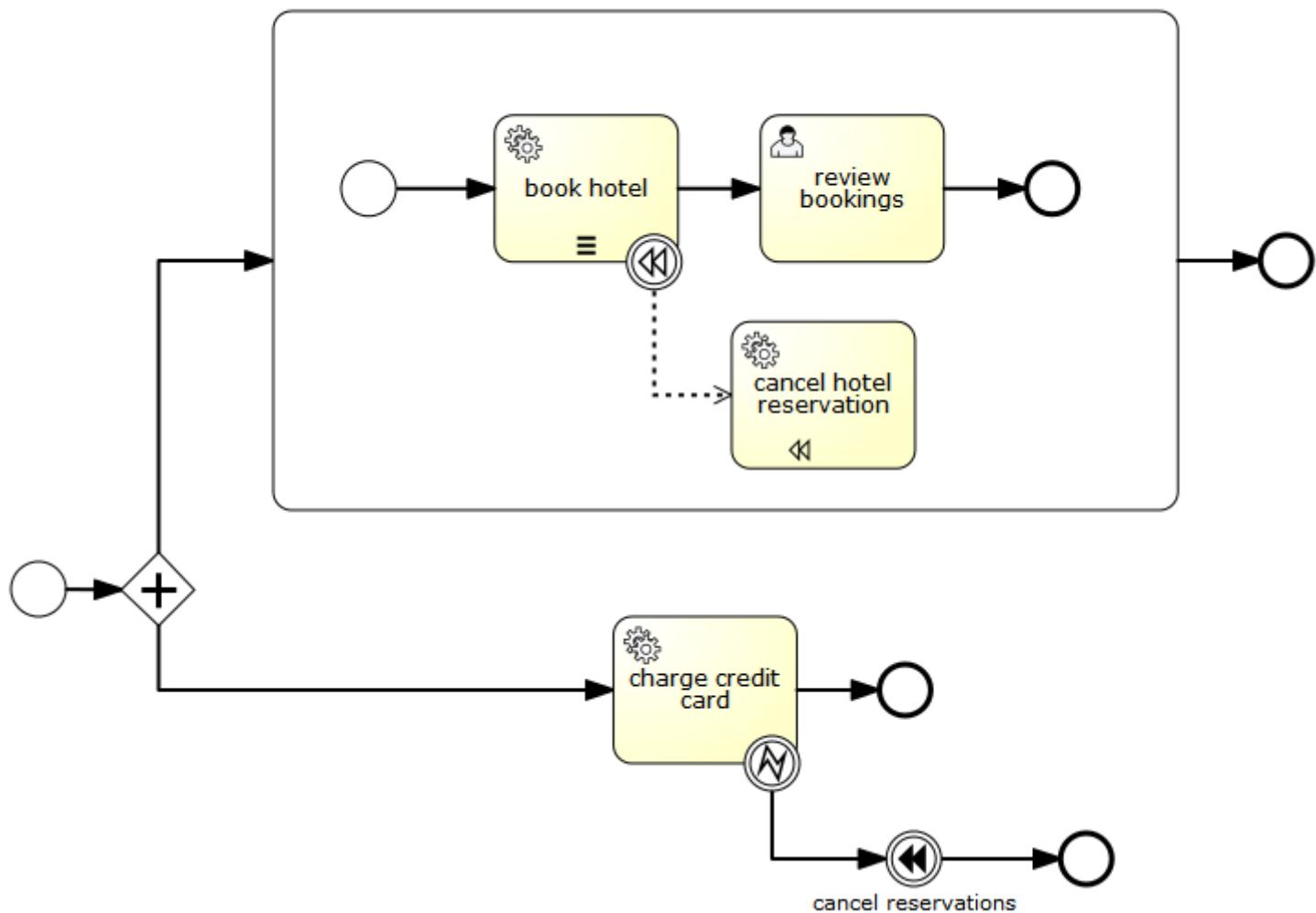
中间触发补偿事件 可以用来触发补偿。

触发补偿: 补偿可以由特定节点或包含补偿事件的作用域触发。 补偿是通过分配给节点的补偿处理器来完成的。

- 当补偿由节点触发，对应的补偿处理器会根据节点成功完成的次数执行相同次数。
- 如果补偿由当前作用域触发，当前作用域的所有节点都会执行补偿，也包含并发分支。
- 补偿的触发是继承式的：如果执行补偿的节点是子流程，补偿会作用到子流程中包含的所有节点。 如果子流程是内嵌节点，补偿会递归触发。 然而，补偿不会传播到流程的上层：如果补偿在子流程中触发，不会传播到子流程范围外。 bpmn规范定义，由节点触发的流程只会作用到“子流程同一级别”。
- activiti的补偿执行次序与流程执行顺序相反。 以为着最后完成的节点会最先执行补偿，诸如此类。
- 中间触发补偿事件可以用来补偿成功完成的事务性子流程。

注意： 如果补偿被一个包含子流程的作用域触发，子流程还包含了关联补偿处理器的节点，补偿只会传播到子流程，如果它已经成功完成了。 如果子流程中的节点也完成了，并关联了

补偿处理器，如果子流程包含的这些节点还没有完成，就不会执行补偿处理器。参考下面实例：



这个流程中，我们有两个并发分支，一些分支内嵌子流程，一个是“使用信用卡”节点。假设两个分支都启动了，第一个分支等待用户完成“审核预定”任务。第二个分支执行“使用信用卡”节点，并发生了一个错误，这导致“取消预定”事件，并触发补偿。这时，并发子流程还没有结束，意味着补偿事件不会传播给子流程，所以“取消旅店预定”这个补偿处理器不会执行。如果用户任务（就是内嵌子流程）在“取消预定”之前完成了，补偿就会传播给内嵌子流程。

流程变量：当补偿内嵌子流程时，用来执行补偿处理器的分支可以访问子流程的本地流程实例，因为这时它是子流程完成的分支。为了实现这个功能，流程变量的快照会分配给分支（为执行子流程而创建的分支）。为此，有以下限制条件：

- 补偿处理器无法访问子流程内部创建的，添加到同步分支的变量。
- 分配给分支的流程变量在继承关系上层的（分配给流程实例的流程变量没有包含在快照中）：补偿触发时，补偿处理器通过它们所在的地方访问这些流程变量。
- 变量快照只用于内嵌子流程，不适用其他节点。

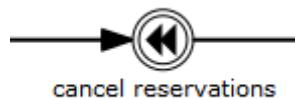
已知限制：

- `waitForCompletion="false"`还不支持。当补偿使用中间触发补偿事件触发时，事件没有等待，在补偿成功结束后。

- 补偿自己由并发分支执行。并发分支的执行顺序与被补偿的节点完成次序相反。未来activiti可能支持选项来顺序执行补偿。
- 补偿不会传播给callActivity调用的子流程实例。

8. 2. 30. 2. #图形标记

中间补偿触发事件显示为标准中间事件（圆圈套圆圈），内部是一个补偿小图标。补偿图标是黑色的（有填充），表示触发语义。



8. 2. 30. 3. #XML内容

补偿中间事件定义为普通的中间触发事件。
是compensateEventDefinition元素。

对应类型的子元素

```
<intermediateThrowEvent id="throwCompensation">
  <compensateEventDefinition />
</intermediateThrowEvent>
```

另外，可选参数activityRef可以用来触发特定作用域/节点的补偿：

```
<intermediateThrowEvent id="throwCompensation">
  <compensateEventDefinition activityRef="bookHotel1" />
</intermediateThrowEvent>
```

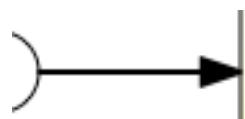
8. 3. #顺序流

8. 3. 1. #描述

顺序流是连接两个流程节点的连线。流程执行完一个节点后，会沿着节点的所有外出顺序流继续执行。就是说，BPMN 2.0默认的行为就是并发的：两个外出顺序流会创造两个单独的，并发流程分支。

8. 3. 2. #图形标记

顺序流显示为从起点到终点的箭头。箭头总是指向终点。



8. 3. 3. #XML内容

顺序流需要流程范围内唯一的id，以及对起点与终点元素的引用。

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

8.3.4. #条件顺序流

8.3.4.1. #描述

可以为顺序流定义一个条件。离开一个BPMN 2.0节点时，默认会计算外出顺序流的条件。如果条件结果为true，就会选择外出顺序流继续执行。当多条顺序流被选中时，就会创建多条分支，流程会继续以并行方式继续执行。

注意：上面的讨论仅涉及BPMN 2.0节点（和事件），不包括网关。网关会用特定的方式处理顺序流中的条件，这与网关类型相关。

8.3.4.2. #图形标记

条件顺序流显示为一个正常的顺序流，不过在起点有一个菱形。条件表达式也会显示在顺序流上。



8.3.4.3. #XML内容

条件顺序流定义为一个正常的顺序流，包含conditionExpression子元素。注意目前只支持tFormalExpressions，如果没有设置xsi:type="", 就会默认值支持目前支持的表达式类型。

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

当前条件表达式只能使用UEL，可以参考表达式章节获取更多信息。使用的表达式需要返回boolean值，否则会在解析表达式时抛出异常。

- 下面的例子引用了流程变量的数据，通过getter调用JavaBean。

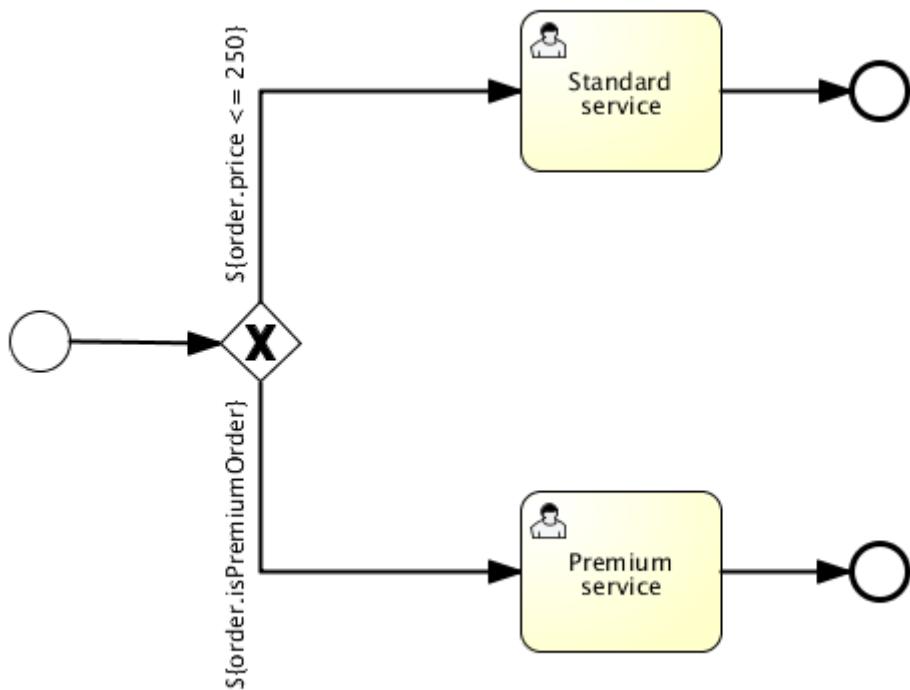
```
<conditionExpression xsi:type="tFormalExpression">
  <![CDATA[{$order.price > 100 && order.price < 250}]]>
</conditionExpression>
```

- 这个例子通过调用方法返回一个boolean值。

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

在activiti发布包中，包含以下流程实例，使用了值和方法表达式
考org.activiti.examples.bpmn.expression包)：

(参



8.3.5. #默认顺序流

8.3.5.1. #描述

所有的BPMN 2.0任务和网关都可以设置一个默认顺序流。只有在节点的其他外出顺序流不能被选中时，才会使用它作为外出顺序流继续执行。默认顺序流的条件设置不会生效。

8.3.5.2. #图形标记

默认顺序流显示为普通顺序流，起点有一个“斜线”标记。



8.3.5.3. #XML内容

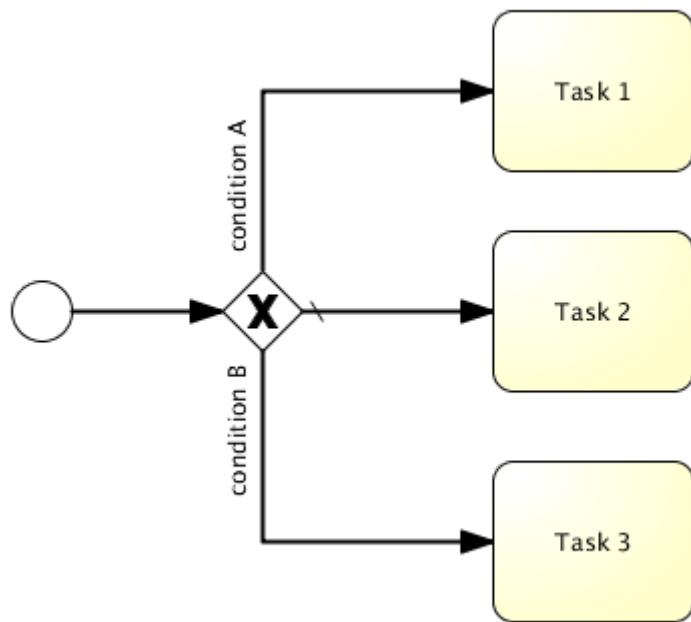
默认顺序流通过对应节点的default属性定义。下面的XML代码演示了排他网关设置了默认顺序流flow 2。只有当conditionA和conditionB都返回false时，才会选择它作为外出连线继续执行。

```

<exclusiveGateway id="exclusiveGw" name="Exclusive Gateway" default="flow2" />
<sequenceFlow id="flow1" sourceRef="exclusiveGw" targetRef="task1">
    <conditionExpression xsi:type="tFormalExpression">$ {conditionA}</conditionExpression>
</sequenceFlow>
<sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="task2"/>
<sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="task3">
    <conditionExpression xsi:type="tFormalExpression">$ {conditionB}</conditionExpression>
</sequenceFlow>

```

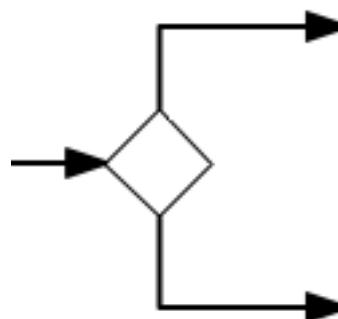
对应下面的图形显示：



8. 4. #网关

网关用来控制流程的流向（或像BPMN 2.0里描述的那样，流程的tokens。） 网关可以消费也可以生成token。

网关显示成菱形图形，内部有有一个小图标。 图标表示网关的类型。



8. 4. 1. #排他网关

8. 4. 1. 1. #描述

排他网关（也叫异或（XOR）网关，或更技术性的叫法 基于数据的排他网关）， 用来在流程中实现决策。 当流程执行到这个网关，所有外出顺序流都会被处理一遍。 其中条件解析为 true 的顺序流（或者没有设置条件，概念上在顺序流上定义了一个' true'）会被选中，让流程继续运行。

注意这里的外出顺序流 与BPMN 2.0通常的概念是不同的。通常情况下，所有条件结果为 true 的顺序流 都会被选中，以并行方式执行，但排他网关只会选择一条顺序流执行。 就是说，虽然多个顺序流的条件结果为true， 那么XML中的第一个顺序流（也只有这一条）会被选中，并用来继续运行流程。 如果没有选中任何顺序流，会抛出一个异常。

8.4.1.2. #图形标记

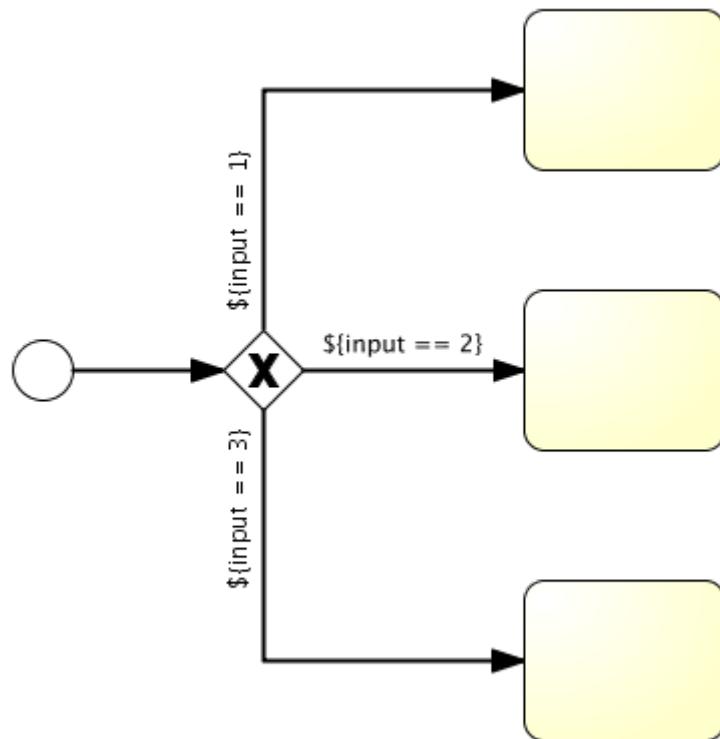
排他网关显示成一个普通网关（比如，菱形图形），内部是一个“X”图标，表示异或（XOR）语义。注意，没有内部图标的网关，默认为排他网关。BPMN 2.0规范不允许在同一个流程定义中同时使用没有X和有X的菱形图形。



8.4.1.3. #XML内容

排他网关的XML内容是很直接的：用一行定义了网关，条件表达式定义在外出顺序流中。参考条件顺序流 获得这些表达式的可用配置。

参考下面模型实例：



它对应的XML内容如下：

```
<exclusiveGateway id="exclusiveGw" name="Exclusive Gateway" />

<sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="theTask1">
    <conditionExpression xsi:type="tFormalExpression">${input == 1}</conditionExpression>
</sequenceFlow>

<sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="theTask2">
    <conditionExpression xsi:type="tFormalExpression">${input == 2}</conditionExpression>
</sequenceFlow>

<sequenceFlow id="flow4" sourceRef="exclusiveGw" targetRef="theTask3">
    <conditionExpression xsi:type="tFormalExpression">${input == 3}</conditionExpression>
</sequenceFlow>
```

</sequenceFlow>

8.4.2. #并行网关

8.4.2.1. #描述

网关也可以表示流程中的并行情况。最简单的并行网关是 并行网关，它允许将流程 分成多条分支，也可以把多条分支 汇聚到一起。 of execution.

并行网关的功能是基于进入和外出的顺序流的：

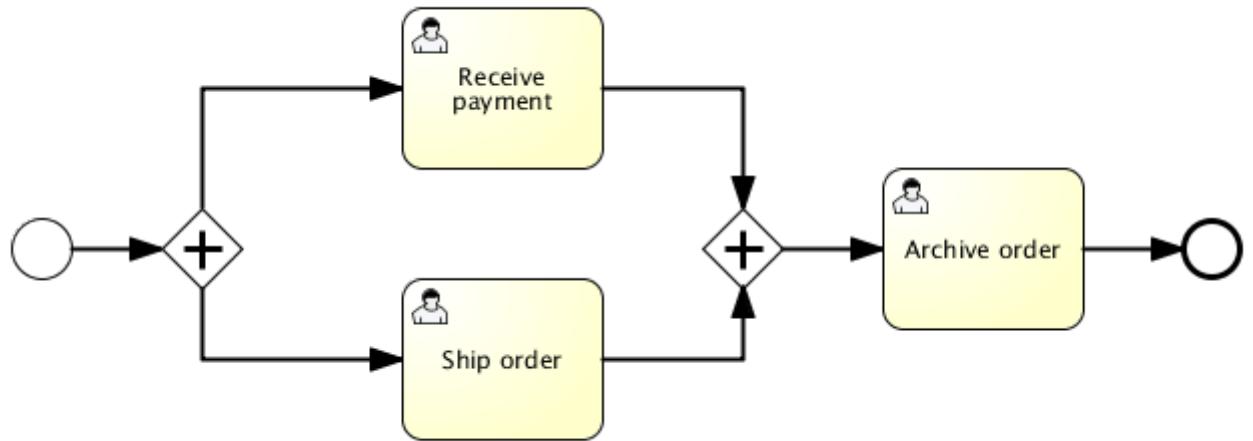
- 分支： 并行后的所有外出顺序流，为每个顺序流都创建一个并发分支。
- 汇聚： 所有到达并行网关，在此等待的进入分支， 直到所有进入顺序流的分支都到达以后， 流程就会通过汇聚网关。

注意，如果同一个并行网关有多个进入和多个外出顺序流， 它就同时具有分支和汇聚功能。这时，网关会先汇聚所有进入的顺序流，然后再切分成多个并行分支。

与其他网关的主要区别是， 并行网关不会解析条件。 即使顺序流中定义了条件，也会被忽略。

8.4.2.2. #图形标记

并行网关显示成一个普通网关（菱形）内部是一个“加号”图标， 表示“与（AND）”语义。



8.4.2.3. #XML内容

定义并行网关只需要一行XML：

<parallelGateway id="myParallelGateway" />

实际发生的行为（分支，聚合，同时分支聚合）， 要根据并行网关的顺序流来决定。

参考如下代码：

```

<startEvent id="theStart" />
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork" />
  
```

```

<parallelGateway id="fork" />
<sequenceFlow sourceRef="fork" targetRef="receivePayment" />
<sequenceFlow sourceRef="fork" targetRef="shipOrder" />

<userTask id="receivePayment" name="Receive Payment" />
<sequenceFlow sourceRef="receivePayment" targetRef="join" />

<userTask id="shipOrder" name="Ship Order" />
<sequenceFlow sourceRef="shipOrder" targetRef="join" />

<parallelGateway id="join" />
<sequenceFlow sourceRef="join" targetRef="archiveOrder" />

<userTask id="archiveOrder" name="Archive Order" />
<sequenceFlow sourceRef="archiveOrder" targetRef="theEnd" />

<endEvent id="theEnd" />

```

上面例子中，流程启动之后，会创建两个任务：

```

ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
TaskQuery query = taskService.createTaskQuery()
    .processInstanceId(pi.getId())
    .orderByTaskName()
    .asc();

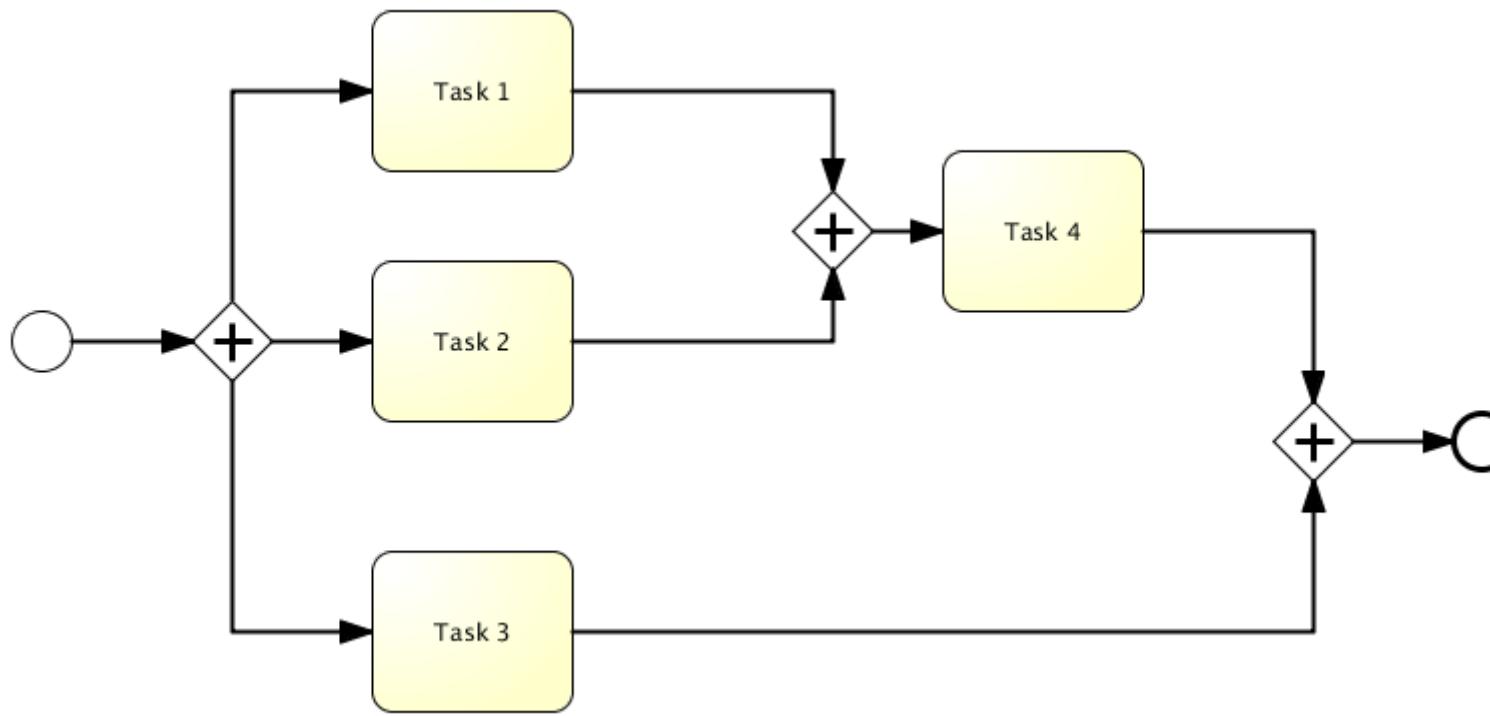
List<Task> tasks = query.list();
assertEquals(2, tasks.size());

Task task1 = tasks.get(0);
assertEquals("Receive Payment", task1.getName());
Task task2 = tasks.get(1);
assertEquals("Ship Order", task2.getName());

```

当两个任务都完成时，第二个并行网关会汇聚两个分支，因为它只有一条外出连线，不会创建并行分支，只会创建归档订单任务。

注意并行网关不需要是“平衡的”（比如，对应并行网关的进入和外出节点数目相等）。并行网关只是等待所有进入顺序流，并为每个外出顺序流创建并发分支，不会受到其他流程节点的影响。所以下面的流程在BPMN 2.0中是合法的：



8.4.3. #包含网关

8.4.3.1. #描述

包含网关可以看做是排他网关和并行网关的结合体。和排他网关一样，你可以在外出顺序流上定义条件，包含网关会解析它们。但是主要的区别是包含网关可以选择多于一条顺序流，这和并行网关一样。

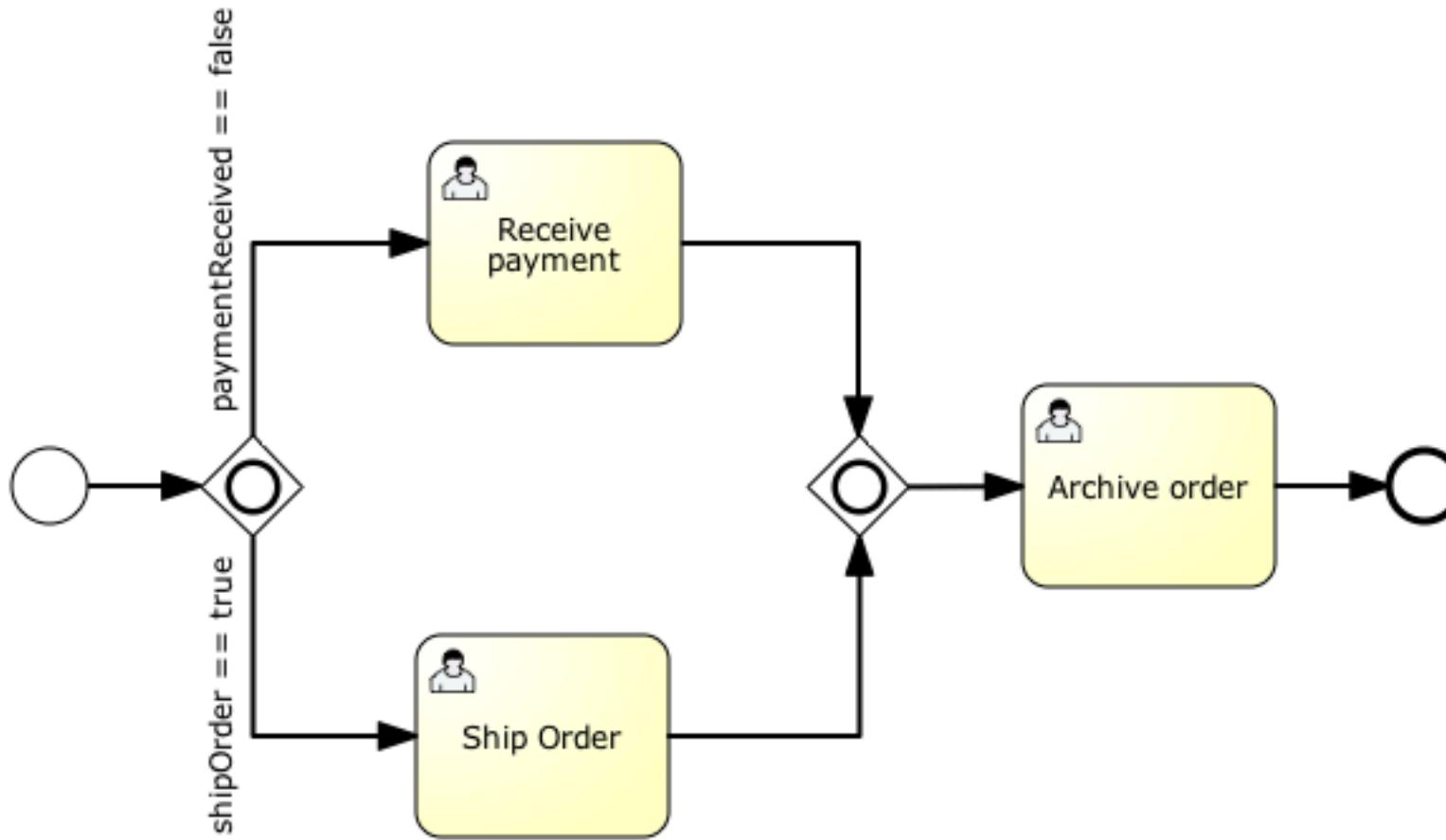
包含网关的功能是基于进入和外出顺序流的：

- 分支：所有外出顺序流的条件都会被解析，结果为true的顺序流会以并行方式继续执行，会为每个顺序流创建一个分支。
- 汇聚：所有并行分支到达包含网关，会进入等待章台，直到每个包含流程token的进入顺序流的分支都到达。这是与并行网关的最大不同。换句话说，包含网关只会等待被选中执行了的进入顺序流。在汇聚之后，流程会穿过包含网关继续执行。

注意，如果同一个包含节点拥有多个进入和外出顺序流，它就会同时含有分支和汇聚功能。这时，网关会先汇聚所有拥有流程token的进入顺序流，再根据条件判断结果为true的外出顺序流，为它们生成多条并行分支。

8.4.3.2. #图形标记

并行网关显示为一个普通网关（菱形），内部包含一个圆圈图标。



8. 4. 3. #XML内容

定义一个包含网关需要一行XML:

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

实际的行为（分支，汇聚或同时分支汇聚），是由连接在包含网关的顺序流决定的。

参考如下代码:

```

<startEvent id="theStart" />
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork" />

<inclusiveGateway id="fork" />
<sequenceFlow sourceRef="fork" targetRef="receivePayment" >
<conditionExpression xsi:type="tFormalExpression">$ {paymentReceived == false}</conditionExpression>
</sequenceFlow>
<sequenceFlow sourceRef="fork" targetRef="shipOrder" >
<conditionExpression xsi:type="tFormalExpression">$ {shipOrder == true}</conditionExpression>
</sequenceFlow>

<userTask id="receivePayment" name="Receive Payment" />
<sequenceFlow sourceRef="receivePayment" targetRef="join" />

<userTask id="shipOrder" name="Ship Order" />
<sequenceFlow sourceRef="shipOrder" targetRef="join" />

<inclusiveGateway id="join" />

```

```

<sequenceFlow sourceRef="join" targetRef="archiveOrder" />

<userTask id="archiveOrder" name="Archive Order" />
<sequenceFlow sourceRef="archiveOrder" targetRef="theEnd" />

<endEvent id="theEnd" />

```

在上面的例子中，流程开始之后，如果流程变量为`paymentReceived == false`和`shipOrder == true`，就会创建两个任务。如果，只有一个流程变量为`true`，就会只创建一个任务。如果没有条件为`true`，就会抛出一个异常。如果想避免异常，可以定义一个默认顺序流。下面的例子中，会创建一个任务，发货任务：

```

HashMap<String, Object> variableMap = new HashMap<String, Object>();
variableMap.put("receivedPayment", true);
variableMap.put("shipOrder", true);
ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
TaskQuery query = taskService.createTaskQuery()
    .processInstanceId(pi.getId())
    .orderByTaskName()
    .asc();

List<Task> tasks = query.list();
assertEquals(1, tasks.size());

Task task = tasks.get(0);
assertEquals("Ship Order", task.getName());

```

当任务完成后，第二个包含网关会汇聚两个分支，因为只有一个外出顺序流，所以不会创建并行分支，只有归档订单任务会被激活。

注意，包含网关不需要“平衡”（比如，对应包含网关的进入和外出数目需要相等）。包含网关会等待所有进入顺序流完成，并为每个外出顺序流创建并行分支，不会受到流程中其他元素的影响。

8. 4. 4. #基于事件网关

8. 4. 4. 1. #描述

基于事件网关允许根据事件判断流向。网关的每个外出顺序流都要连接到一个中间捕获事件。当流程到达一个基于事件网关，网关会进入等待状态：会暂停执行。与此同时，会为每个外出顺序流创建相对的事件订阅。

注意基于事件网关的外出顺序流和普通顺序流不同。这些顺序流不会真的“执行”。相反，它们让流程引擎去决定执行到基于事件网关的流程需要订阅哪些事件。要考虑以下条件：

- 基于事件网关必须有两条或以上外出顺序流。
- 基于事件网关后，只能使用`intermediateCatchEvent`类型。（activiti不支持基于事件网关后连接`ReceiveTask`。）
- 连接到基于事件网关的`intermediateCatchEvent`只能有一条进入顺序流。

8. 4. 4. 2. #图形标记

基于事件网关和其他BPMN网关一样显示成一个菱形，内部包含指定图标。

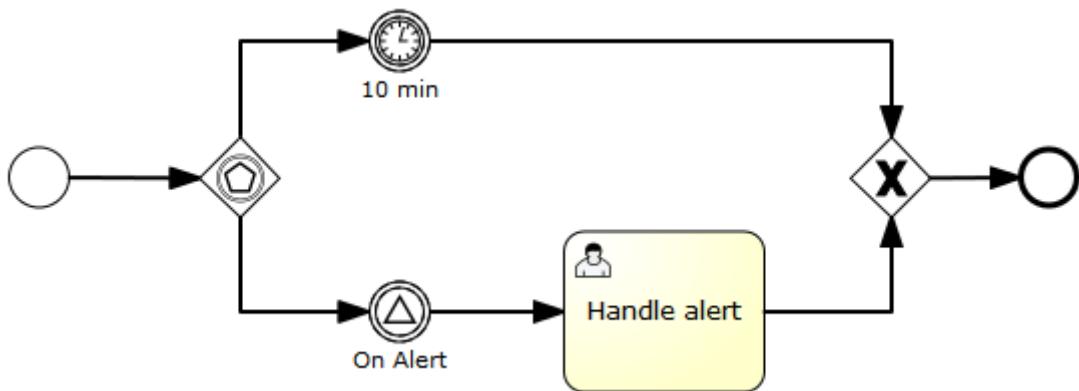


8. 4. 4. 3. #XML内容

用来定义基于事件网关的XML元素是eventBasedGateway。

8. 4. 4. 4. #实例

下面的流程是一个使用基于事件网关的例子。当流程执行到基于事件网关时，流程会暂停执行。与此同时，流程实例会订阅警告信号事件，并创建一个10分钟后触发的定时器。这会产生流程引擎为一个信号事件等待10分钟的效果。如果10分钟内发出信号，定时器就会取消，流程会沿着信号执行。如果信号没有出现，流程会沿着定时器的方向前进，信号订阅会被取消。



```

<definitions id="definitions"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:activiti="http://activiti.org/bpmn"
  targetNamespace="Examples">

  <signal id="alertSignal" name="alert" />

  <process id="catchSignal">
    <startEvent id="start" />
    <sequenceFlow sourceRef="start" targetRef="gw1" />
    <eventBasedGateway id="gw1" />
    <sequenceFlow sourceRef="gw1" targetRef="signalEvent" />
    <sequenceFlow sourceRef="gw1" targetRef="timerEvent" />
    <intermediateCatchEvent id="signalEvent" name="Alert">
      <signalEventDefinition signalRef="alertSignal" />
    </intermediateCatchEvent>
    <intermediateCatchEvent id="timerEvent" name="Alert">
      <timerEventDefinition>
        <timeDuration>PT10M</timeDuration>
      </timerEventDefinition>
    </intermediateCatchEvent>
  </process>
</definitions>
  
```

```

<sequenceFlow sourceRef="timerEvent" targetRef="exGw1" />
<sequenceFlow sourceRef="signalEvent" targetRef="task" />

<userTask id="task" name="Handle alert"/>

<exclusiveGateway id="exGw1" />

<sequenceFlow sourceRef="task" targetRef="exGw1" />
<sequenceFlow sourceRef="exGw1" targetRef="end" />

<endEvent id="end" />
</process>
</definitions>

```

8. 5. # 任务

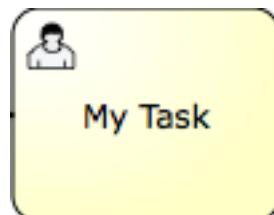
8. 5. 1. #用户任务

8. 5. 1. 1. #描述

用户任务用来设置必须由人员完成的工作。当流程执行到用户任务，会创建一个新任务，并把这个新任务加入到分配人或群组的任务列表中。

8. 5. 1. 2. #图形标记

用户任务显示成一个普通任务（圆角矩形），左上角有一个小用户图标。



8. 5. 1. 3. #XML内容

XML中的用户任务定义如下。id属性是必须的。name属性是可选的。

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

用户任务也可以设置描述。实际上所有BPMN 2.0元素都可以设置描述。添加documentation元素可以定义描述。

添

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

描述文本可以通过标准的java方法来获得：

```
task.getDescription()
```

8.5.1.4. #持续时间

任务可以用一个字段来描述任务的持续时间。可以使用查询API来对持续时间进行搜索，根据在时间之前或之后进行搜索。

我们提供了一个节点扩展，在任务定义中设置一个表达式，这样在任务创建时就可以为它设置初始持续时间。表达式应该是`java.util.Date`, `java.util.String` (ISO8601格式), ISO8601 持续时间 (比如PT50M) 或null。例如：你可以在流程中使用上述格式输入日期，或在前一个服务任务中计算一个时间。这里使用了持续时间，持续时间会基于当前时间进行计算，再通过给定的时间段累加。比如，使用“PT30M”作为持续时间，任务就会从现在开始持续30分钟。

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

任务的持续时间也可以通过`TaskService`修改，或在`TaskListener`中通过传入的`DelegateTask`参数修改。

8.5.1.5. #用户分配

用户任务可以直接分配给一个用户。这可以通过`humanPerformer`元素定义。`humanPerformer`定义需要一个`resourceAssignmentExpression`来实际定义用户。当前，只支持`formalExpressions`。

```
<process ... >

...
<userTask id='theTask' name='important task' >
  <humanPerformer>
    <resourceAssignmentExpression>
      <formalExpression>kermit</formalExpression>
    </resourceAssignmentExpression>
  </humanPerformer>
</userTask>
```

只有一个用户可以坐拥任务的执行者分配给用户。在`activiti`中，用户叫做执行者。拥有执行者的用户不会出现在其他人的任务列表中，只能出现执行者的个人任务列表中。

直接分配给用户的任务可以通过`TaskService`像下面这样获取：

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

任务也可以加入到人员的候选任务列表中。这时，需要使用`potentialOwner`元素。用法和`humanPerformer`元素类似。注意它需要指定表达式中的每个项目是人员还是群组（引擎猜不出来）。

```
<process ... >

...
<userTask id='theTask' name='important task' >
  <potentialOwner>
    <resourceAssignmentExpression>
      <formalExpression>user(kermit), group(management)</formalExpression>
    </resourceAssignmentExpression>
  </potentialOwner>
</userTask>
```

```
</resourceAssignmentExpression>
</potentialOwner>
</userTask>
```

使用potentialOwner元素定义的任务，可以像下面这样获取（使用TaskQuery的发那个发与查询设置了执行者的任务类似）：

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

这会获取所有kermit为候选人的任务，例如：表达式中包含user(kermit)。这也会获得所有分配包含kermit这个成员的群组（比如，group(management)，前提是kermit是这个组的成员，并且使用了activiti的账号组件）。用户所在的群组是在运行阶段获取的，它们可以通过IdentityService进行管理。

如果没有显示指定设置的是用户还是群组，引擎会默认当做群组处理。所以下面的设置与使用group(accountancy)效果一样。

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

8.5.1.6. #Activiti对任务分配的扩展

当分配不复杂时，用户和组的设置非常麻烦。为避免复杂性，可以使用用户任务的自定义扩展。

- assignee属性：这个自定义扩展可以直接把用户任务分配给指定用户。

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

它和使用上面定义的humanPerformer效果完全一样。

- candidateUsers属性：这个自定义扩展可以为任务设置候选人。

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

它和使用上面定义的potentialOwner效果完全一样。注意它不需要像使用potentialOwner通过user(kermit)声明，因为这个属性只能用于人员。

- candidateGroups属性：这个自定义扩展可以为任务设置候选组。

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

它和使用上面定义的potentialOwner效果完全一样。注意它不需要像使用potentialOwner通过group(management)声明，因为这个属性只能用于群组。

- candidateUsers 和 candidateGroups 可以同时设置在同一个用户任务中。

注意：虽然activiti提供了一个账号管理组件，也提供了IdentityService，但是账号组件不会检测设置的用户是否有效。它嵌入到应用中，也允许activiti与其他已存的账户管理方案集成。

如果上面的方式还不满足需求，还可以使用创建事件的任务监听器来实现自定义的分配逻辑：

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

DelegateTask会传递给TaskListener的实现， 通过它可以设置执行人， 候选人和候选组：

```
public class MyAssignmentHandler implements TaskListener {

  public void notify(DelegateTask delegateTask) {
    // Execute custom identity lookups here

    // and then for example call following methods:
    delegateTask.setAssignee("kermit");
    delegateTask.addCandidateUser("fozzie");
    delegateTask.addCandidateGroup("management");
    ...
  }
}
```

使用spring时， 可以使用向上面章节中介绍的自定义分配属性， 使用表达式 把任务监听器设置为spring代理的bean， 让这个监听器监听任务的创建事件。 下面的例子中， 执行者会通过调用ldapService这个spring bean的findManagerOfEmployee方法获得。 流程变量emp会作为参数传递给bean。

```
<userTask id="task" name="My Task" activiti:assignee="${ldapService.findManagerForEmployee(emp)}"/>
```

也可以用来设置候选人和候选组：

```
<userTask id="task" name="My Task" activiti:candidateUsers="${ldapService.findAllSales()}"/>
```

注意方法返回类型只能为String或Collection<String> （对应候选人和候选组）：

```
public class FakeLdapService {

  public String findManagerForEmployee(String employee) {
    return "Kermit The Frog";
  }

  public List<String> findAllSales() {
    return Arrays.asList("kermit", "gonzo", "fozzie");
  }
}
```

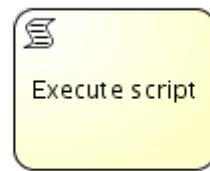
8. 5. 2. #脚本任务

8. 5. 2. 1. #描述

脚本任务时一个自动节点。当流程到达脚本任务， 会执行对应的脚本。

8. 5. 2. 2. #图形标记

脚本任务显示为标准BPMN 2.0任务（圆角矩形）， 左上角有一个脚本小图标。



8. 5. 2. 3. #XML内容

脚本任务定义需要指定script 和scriptFormat。

```
<scriptTask id="theScriptTask" name="Execute script" scriptFormat="groovy">
<script>
sum = 0
for ( i in inputArray ) {
    sum += i
}
</script>
</scriptTask>
```

scriptFormat的值必须兼容 JSR-223 [<http://jcp.org/en/jsr/detail?id=223>]。（java平台的脚本语言）。默认Javascript会包含在JDK中，不需要额外的依赖。如果你想使用其他（JSR-223兼容）的脚本引擎，需要把对应的jar添加到classpath下，并使用合适的名称。比如，activiti单元测试经常使用groovy，因为语法比java简单太多。

注意，groovy脚本引擎放在groovy-all.jar中。在2.0版本之前，脚本引擎是groovy jar的一部分。这样，需要添加如下依赖：

```
<dependency>
<groupId>org.codehaus.groovy</groupId>
<artifactId>groovy-all</artifactId>
<version>2.x.x</version>
</dependency>
```

8. 5. 2. 4. #脚本中的变量

到达脚本任务的流程可以访问的所有流程变量，都可以在脚本中使用。实例中，脚本变量'inputArray'其实是流程变量（整数数组）。

```
<script>
sum = 0
for ( i in inputArray ) {
    sum += i
}
</script>
```

也可以在脚本中设置流程变量，直接调用 execution.setVariable("variableName", variableValue)。默认，不会自动保存变量（注意：activiti 5.12之前存在这个问题）。可以在脚本中自动保存任何变量。（比如上例中的sum），只要把scriptTask 的autoStoreVariables属性设置为true。然而，最佳实践是不要用它，而是显示调用execution.setVariable()，因为一些当前版本的JDK对于一些脚本语言，无法实现自动保

存变量。 参考这里 [http://www.jorambarrez.be/blog/2013/03/25/bug-on-jdk-1-7-0_17-when-using-scripttask-in-activiti/] 获得更多信息。

```
<scriptTask id="script" scriptFormat="JavaScript" activiti:autoStoreVariables="false">
```

参数默认为 false，意思是如果没有为脚本任务定义设置参数，所有声明的变量将只存在于脚本执行的阶段。

如何在脚本中设置变量的例子：

```
<script>
    def scriptVar = "test123"
    execution.setVariable("myVar", scriptVar)
</script>
```

注意：下面这些命名已被占用，不能用作变量名： out, out:print, lang:import, context, elcontext。

8.5.2.5. #脚本结果

脚本任务的返回值可以通过制定流程变量的名称，分配给已存或一个新流程变量，使用脚本任务定义的' activiti:resultVariable' 属性。任何已存的流程变量都会被脚本执行的结果覆盖。如果没有指定返回变量名，脚本的返回值会被忽略。

```
<scriptTask id="theScriptTask" name="Execute script" scriptFormat="juel" activiti:resultVariable="myVar">
    <script>#{echo}</script>
</scriptTask>
```

上例中，脚本的结果（表达式'#{echo}'的值）在脚本完成后，会设置到' myVar' 变量中。

8.5.3. #Java服务任务

8.5.3.1. #描述

java服务任务用来调用外部java类。

8.5.3.2. #图形标记

服务任务显示为圆角矩形，左上角有一个齿轮小图标。



8.5.3.3. #XML内容

有4钟方法来声明java调用逻辑：

- 实现JavaDelegate或ActivityBehavior
- 执行解析代理对象的表达式

- 调用一个方法表达式
- 调用一直值表达式

执行一个在流程执行中调用的类， 需要在' activiti:class' 属性中设置全类名。

```
<serviceTask id="javaService"
    name="My Java Service Task"
    activiti:class="org.activiti.MyJavaDelegate" />
```

参考实现章节 了解更多使用类的信息。

也可以使用表达式调用一个对象。对象必须遵循一些规则，并使用activiti:class属性进行创建。（了解更多）。

```
<serviceTask id="serviceTask" activiti:delegateExpression="${delegateExpressionBean}" />
```

这里， delegateExpressionBean是一个实现了JavaDelegate接口的bean， 它定义在实例的spring容器中。

要指定执行的UEL方法表达式， 需要使用activiti:expression。

```
<serviceTask id="javaService"
    name="My Java Service Task"
    activiti:expression="#{printer.printMessage()}" />
```

方法printMessage（无参数）会调用 名为printer对象的方法。

也可以为表达式中的方法传递参数。

```
<serviceTask id="javaService"
    name="My Java Service Task"
    activiti:expression="#{printer.printMessage(execution, myVar)}" />
```

这会调用名为printer对象上的方法printMessage。 第一个参数是DelegateExecution，在表达式环境中默认名称为execution。 第二个参数传递的是当前流程的名为myVar的变量。

要指定执行的UEL值表达式， 需要使用activiti:expression属性。

```
<serviceTask id="javaService"
    name="My Java Service Task"
    activiti:expression="#{split.ready}" />
```

ready属性的getter方法， getReady（无参数）， 会作用于名为split的bean上。 这个对象会被解析为流程对象和（如果合适）spring环境中的对象。

8. 5. 3. 4. #实现

要在流程执行中实现一个调用的类， 这个类需要实

现org.activiti.engine.delegate.JavaDelegate接口，并在execute方法中提供对应的业务逻辑。 当流程执行到特定阶段， 它会指定方法中定义好的业务逻辑，并按照默认BPMN 2.0中的方式离开节点。

让我们创建一个java类的例子， 它可以流程变量中字符串转换为大写。

这个类需要实
现org.activiti.engine.delegate.JavaDelegate接口，

这要求我们实
现execute(DelegateExecution)方法。 它包含的业务逻辑会被引擎调用。 流程实例信息，如

流程变量和其他信息，可以通过 DelegateExecution [<http://activiti.org/javadocs/org/activiti/engine/delegate/DelegateExecution.html>] 接口访问和操作（点击对应操作的javadoc的链接，获得更多信息）。

```
public class ToUppercase implements JavaDelegate {

    public void execute(DelegateExecution execution) throws Exception {
        String var = (String) execution.getVariable("input");
        var = var.toUpperCase();
        execution.setVariable("input", var);
    }

}
```

注意：serviceTask定义的class只会创建一个java类的实例。所有流程实例都会共享相同的类实例，并调用execute(DelegateExecution)。这意味着，类不能使用任何成员变量，必须是线程安全的，它必须能模拟在不同线程中执行。这也影响着属性注入的处理方式。

流程定义中引用的类（比如，使用activiti:class）不会在部署时实例化。只有当流程第一次执行到使用类的时候，类的实例才会被创建。如果找不到类，会抛出一个ActivitiException。这个原因是部署环境（更确切的是classpath）和真实环境往往是不同的。比如当使用ant或业务归档上传到Activiti Explorer来发布流程 classpath没有包含引用的类。

[内部：非公共实现类] 也可以提供实现org.activiti.engine.impl.pvm.delegate.ActivityBehavior接口的类。实现可以访问更强大的ActivityExecution，它可以影响流程的流向。注意，这不是一个很好的实践，应该尽量避免。所以，建议只有在高级情况下并且你确切知道你要做什么的情况下，再使用ActivityBehavior接口。

8.5.3.5. #属性注入

可以为代理类的属性注入数据。支持如下类型的注入：

- 固定的字符串
- 表达式

如果有效的话，数值会通过代理类的setter方法注入，遵循java bean的命名规范（比如firstName属性对应setFirstName(...)方法）。如果属性没有对应的setter方法，数值会直接注入到私有属性中。一些环境的SecurityManager不允许修改私有属性，所以最好还是把你想要注入的属性暴露出对应的setter方法来。无论流程定义中的数据是什么类型，注入目标的属性类型都应该是 org.activiti.engine.delegate.Expression。

下面代码演示了如何把一个常量注入到属性中。属性注入可以使用' class' 属性。注意我们需要定义一个' extensionElements' XML元素，在声明实际的属性注入之前，这是BPMN 2.0 XML格式要求的。

```
<serviceTask id="javaService"
    name="Java service invocation"
    activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
    <extensionElements>
        <activiti:field name="text" stringValue="Hello World" />
    </extensionElements>
</serviceTask>
```

ToUpperCaseFieldInjected类有一个text属性，类型是org.activiti.engine.delegate.Expression。调用text.getValue(execution)时，会返回定义的字符串Hello World。

也可以使用长文字（比如，内嵌的email），可以使用'activiti:string'子元素：

```
<serviceTask id="javaService"
    name="Java service invocation"
    activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
<extensionElements>
    <activiti:field name="text">
        <activiti:string>
            Hello World
        </activiti:string>
    </activiti:field>
</extensionElements>
</serviceTask>
```

可以使用表达式，实现在运行期动态解析注入的值。这些表达式可以使用流程变量或spring定义的bean（如果使用了spring）。像服务任务实现里说的那样，服务任务中的java类实例会在所有流程实例中共享。为了动态注入属性的值，我们可以在org.activiti.engine.delegate.Expression中使用值和方法表达式，它会使用传递给execute方法的DelegateExecution参数进行解析。

```
<serviceTask id="javaService" name="Java service invocation"
    activiti:class="org.activiti.examples.bpmn.servicetask.ReverseStringsFieldInjected">

<extensionElements>
    <activiti:field name="text1">
        <activiti:expression>${genderBean.getGenderString(gender)}</activiti:expression>
    </activiti:field>
    <activiti:field name="text2">
        <activiti:expression>Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}</activiti:expression>
    </activiti:field>
</ extensionElements>
</ serviceTask>
```

下面的例子中，注入了表达式，并使用在传入的当前DelegateExecution解析它们。完整代码可以参考org.activiti.examples.bpmn.servicetask.JavaServiceTaskTest.testExpressionFieldInjection。

```
public class ReverseStringsFieldInjected implements JavaDelegate {

    private Expression text1;
    private Expression text2;

    public void execute(DelegateExecution execution) {
        String value1 = (String) text1.getValue(execution);
        execution.setVariable("var1", new StringBuffer(value1).reverse().toString());

        String value2 = (String) text2.getValue(execution);
        execution.setVariable("var2", new StringBuffer(value2).reverse().toString());
    }
}
```

另外，你也可以把表达式设置成一个属性，而不是字元素，让XML更简单一些。

```
<activiti:field name="text1" expression="${genderBean.getGenderString(gender)}" />
<activiti:field name="text1" expression="Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}" />
```

因为java类实例会被重用，注入只会发生一次，当服务任务调用第一次的时候。当你的代码中的属性改变了，值也不会重新注入，所以你应该把它们看做是不变的，不用修改它们。

8.5.3.6. #服务任务结果

服务流程返回的结果（使用表达式的服务任务）可以分配给已经存在的或新的流程变量，可以通过指定服务任务定义的' activiti:resultVariable' 属性来实现。指定的路程比那两的值会被服务流程的返回结果覆盖。如果没有指定返回变量名，就会忽略返回结果。

```
<serviceTask id="aMethodExpressionServiceTask"
    activiti:expression="#{myService.doSomething()}"
    activiti:resultVariable="myVar" />
```

在上面的例子中，服务流程的返回值（在' myService' 上调用' doSomething()' 方法的返回值， myService可能是流程变量，也可能是spring的bean），会设置到名为' myVar' 的流程变量里，在服务执行完成之后。

8.5.3.7. #处理异常

执行自定义逻辑时，常常需要捕获对应的业务异常，在流程内部进行处理。activiti提供了不同的方式来处理这个问题。

8.5.3.7.1. #抛出BPMN Errors

可以在服务任务或脚本任务的代码里抛出BPMN error。为了实现这个，要从JavaDelegate，脚本，表达式和代理表达式中抛出名为 BpmnError的特殊ActivitiException。引擎会捕获这个异常，把它转发到对应的错误处理中。比如，边界错误事件或错误事件子流程。

```
public class ThrowBpmnErrorDelegate implements JavaDelegate {

    public void execute(DelegateExecution execution) throws Exception {
        try {
            executeBusinessLogic();
        } catch (BusinessException e) {
            throw new BpmnError("BusinessExceptionOccured");
        }
    }
}
```

构造参数是错误代码，会被用来决定 哪个错误处理器会来响应这个错误。参考边界错误事件 获得更多捕获BPMN error的信息。

这个机制应该只用于业务失败，它应该被流程定义中设置的边界错误事件或错误事件子流程处理。技术上的错误应该使用其他异常类型，通常不会在流程里处理。

8.5.3.7.2. #异常顺序流

[内部，公开实现类] 另一种选择是在一些异常发生时，让路程进入其他路径。下面的代码演示了如何实现。

```

<serviceTask id="javaService"
    name="Java service invocation"
    activiti:class="org.activiti.ThrowsExceptionBehavior">
</serviceTask>

<sequenceFlow id="no-exception" sourceRef="javaService" targetRef="theEnd" />
<sequenceFlow id="exception" sourceRef="javaService" targetRef="fixException" />

```

这里的任务有两个外出顺序流，分别叫exception和 no-exception。异常出现时会使用顺序流的id来决定流向：

```

public class ThrowsExceptionBehavior implements ActivityBehavior {

    public void execute(ActivityExecution execution) throws Exception {
        String var = (String) execution.getVariable("var");

        PvmTransition transition = null;
        try {
            executeLogic(var);
            transition = execution.getActivity().findOutgoingTransition("no-exception");
        } catch (Exception e) {
            transition = execution.getActivity().findOutgoingTransition("exception");
        }
        execution.take(transition);
    }

}

```

8. 5. 3. 8. #在JavaDelegate里使用activiti服务

一些场景下，需要在java服务任务中使用activiti服务（比如，通过RuntimeService启动流程实例，而callActivity不满足你的需求）。

org.activiti.engine.delegate.DelegateExecution允许通过

org.activiti.engine.EngineServices接口直接获得这些服务：

```

public class StartProcessInstanceTestDelegate implements JavaDelegate {

    public void execute(DelegateExecution execution) throws Exception {
        RuntimeService runtimeService = execution.getEngineServices().getRuntimeService();
        runtimeService.startProcessInstanceByKey("myProcess");
    }

}

```

所有activiti服务的API都可以通过这个接口获得。

使用这些API调用出现的所有数据改变，都是在当前事务中的。

在像spring和CDI

这样的依赖注入环境也会起作用，无论是否启用了JTA数据源。

比如，下面的代码

功能与上面的代码一致，

这是RuntimeService是通过依赖注入获得的，而不是通过org.activiti.engine.EngineServices接口。

```

@Component("startProcessInstanceDelegate")
public class StartProcessInstanceTestDelegateWithInjection {

```

```

@.Autowired
private RuntimeService runtimeService;

public void startProcess() {
    runtimeService.startProcessInstanceByKey("oneTaskProcess");
}

}

```

重要技术提示：因为服务调用是在当前事务里，数据的产生或改变，在服务任务执行完之前，还没有提交到数据库。所有API对于数据库数据的操作，意味着未提交的操作在服务任务的API调用中都是不可见的。

8.5.4. #Web Service任务

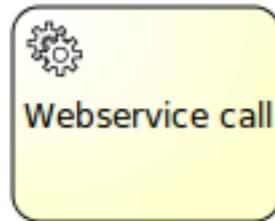
[EXPERIMENTAL]

8.5.4.1. #描述

Web Service任务可以用来同步调用一个外部的Web service。

8.5.4.2. #图形标记

Web Service任务与Java服务任务显示效果一样。



8.5.4.3. #XML内容

要使用Web Service我们需要导入它的操作和类型。可以自动使用import标签来指定Web Service的WSDL：

```

<import importType="http://schemas.xmlsoap.org/wsdl/"
       location="http://localhost:63081/counter?wsdl"
       namespace="http://webservice.activiti.org/" />

```

上面的声明告诉activiti导入WSDL定义，但没有创建item定义和消息。假设我们想调用一个名为'prettyPrint'的方法，我们必须创建为请求和响应信息对应的消息和item定义：

```

<message id="prettyPrintCountRequestMessage" itemRef="tns:prettyPrintCountRequestItem" />
<message id="prettyPrintCountResponseMessage" itemRef="tns:prettyPrintCountResponseItem" />

<itemDefinition id="prettyPrintCountRequestItem" structureRef="counter:prettyPrintCount" />
<itemDefinition id="prettyPrintCountResponseItem" structureRef="counter:prettyPrintCountResponse" />

```

在申请服务任务之前，我们必须定义实际引用Web Service的BPMN接口和操作。基本上，我们定义接口和必要的操作。对每个操作我们都会重用上面定义的信息作为输入和输出。比如，下面定义了'counter'接口和'prettyPrintCountOperation'操作：

```
<interface name="Counter Interface" implementationRef="counter:Counter">
<operation id="prettyPrintCountOperation" name="prettyPrintCount Operation"
implementationRef="counter:prettyPrintCount">
<inMessageRef>tns:prettyPrintCountRequestMessage</inMessageRef>
<outMessageRef>tns:prettyPrintCountResponseMessage</outMessageRef>
</operation>
</interface>
```

然后我们可以定义Web Service任务使用##WebService实现，并引用Web Service操作。

```
<serviceTask id="webService"
name="Web service invocation"
implementation="##WebService"
operationRef="tns:prettyPrintCountOperation">
```

8. 5. 4. #Web Service任务IO规范

除非我们使用简化方式处理数据输入和输出关联（如下所示），每个Web Service任务可以定义任务的输入输出IO规范。配置方式与BPMN 2.0完全兼容，下面格式化后的例子，我们根据之前定义item定义，定义了输入和输出。

```
<iSpecification>
<dataInput itemSubjectRef="tns:prettyPrintCountRequestItem" id="dataInputOfServiceTask" />
<dataOutput itemSubjectRef="tns:prettyPrintCountResponseItem" id="dataOutputOfServiceTask" />
<inputSet>
<dataInputRefs>dataInputOfServiceTask</dataInputRefs>
</inputSet>
<outputSet>
<dataOutputRefs>dataOutputOfServiceTask</dataOutputRefs>
</outputSet>
</iSpecification>
```

8. 5. 4. 5. #Web Service任务数据输入关联

有两种方式指定数据输入关联：

- 使用表达式
- 使用简化方式

要使用表达式指定数据输入关联，我们需要定义来源和目的item，并指定每个item属性之间的对应关系。下面的例子中我们分配了这些item的前缀和后缀：

```
<dataInputAssociation>
<sourceRef>dataInputOfProcess</sourceRef>
<targetRef>dataInputOfServiceTask</targetRef>
<assignment>
<from>${dataInputOfProcess.prefix}</from>
<to>${dataInputOfServiceTask.prefix}</to>
</assignment>
<assignment>
<from>${dataInputOfProcess.suffix}</from>
<to>${dataInputOfServiceTask.suffix}</to>
</assignment>
```

```
</dataInputAssociation>
```

另外，我们可以使用更简单的简化方式。'sourceRef'元素是activiti的变量名，'targetRef'元素是item定义的一个属性。在下面的例子中，我们把'PrefixVariable'变量的值分配给'field'属性，把'SuffixVariable'变量的值分配给'suffix'属性。

```
<dataInputAssociation>
  <sourceRef>PrefixVariable</sourceRef>
  <targetRef>prefix</targetRef>
</dataInputAssociation>
<dataInputAssociation>
  <sourceRef>SuffixVariable</sourceRef>
  <targetRef>suffix</targetRef>
</dataInputAssociation>
```

8. 5. 4. 6. #Web Service任务数据输出关联

有两种方式指定数据输出关联：

- 使用表达式
- 使用简化方式

要使用表达式指定数据输出关联，我们需要定义目的变量和来源表达式。方法和数据输入关联完全一样：

```
<dataOutputAssociation>
  <targetRef>dataOutputOfProcess</targetRef>
  <transformation>${dataOutputOfServiceTask.prettyPrint}</transformation>
</dataOutputAssociation>
```

另外，我们可以使用更简单的简化方式。'sourceRef'元素是item定义的一个属性，'targetRef'元素是activiti的变量名。方法和数据输入关联完全一样：

```
<dataOutputAssociation>
  <sourceRef>prettyPrint</sourceRef>
  <targetRef>OutputVariable</targetRef>
</dataOutputAssociation>
```

8. 5. 5. #业务规则任务

[EXPERIMENTAL]

8. 5. 5. 1. #描述

业务规则用户用来同步执行一个或多个规则。activiti使用drools规则引擎执行业务规则。目前，包含业务规则的.drl文件必须和流程定义一起发布，流程定义里包含了执行这些规则的业务规则任务。意味着流程使用的所有.drl文件都必须打包在流程BAR文件里，比如任务表单。更多使用Drools Expert创建业务规则的信息，请参考JBoss Drools [<http://www.jboss.org/drools/documentation>]的文档。

如果想要使用你的规则任务的实现，比如，因为你想用不同方式使用drools，或你想使用完全不同的规则引擎，你可以使用BusinessRuleTask上的class或表达式属性，它用起来就和ServiceTask [#bpmnJavaServiceTask]一样。

8. 5. 5. 2. #图形标记

业务规则任务使用一个表格小图标进行显示。



8. 5. 5. 3. #XML内容

要执行部署流程定义的BAR文件中的一个或多个业务规则，我们需要定义输入和输出变量。对于输入变量定义，可以使用逗号分隔的一些流程变量。输出变量定义智能包含一个变量名，，它会把执行业务规则后返回的对象保存到对应的流程变量中。注意，结果变量会包含一个对象列表。如果没有指定输出变量名称，默认会使用org.activiti.engine.rules.OUTPUT。

下面的业务规则任务会执行和流程定义一起部署的素有业务规则：

```
<process id="simpleBusinessRuleProcess">

    <startEvent id="theStart" />
    <sequenceFlow sourceRef="theStart" targetRef="businessRuleTask" />

    <businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="${order}"
        activiti:resultVariable="rulesOutput" />

    <sequenceFlow sourceRef="businessRuleTask" targetRef="theEnd" />

    <endEvent id="theEnd" />

</process>
```

业务规则任务也可以配置成只执行部署的. drl文件中的一些规则。这时要设置逗号分隔的规则名。

```
<businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="${order}"
    activiti:rules="rule1, rule2" />
```

这时，只会执行rule1和rule2。

你也可以定义哪些规则不用执行。

```
<businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="${order}"
    activiti:rules="rule1, rule2" exclude="true" />
```

这时除了rule1和rule2以外，所有部署到流程定义同一个BAR文件中的规则都会执行。

像之前提到的，可以用一个选项修改BusinessRuleTask的实现：

```
<businessRuleTask id="businessRuleTask" activiti:class="${MyRuleServiceDelegate}" />
```

注意BusinessRuleTask的功能和ServiceTask一样，但是我们使用BusinessRuleTask的图标来表示 我们在这里要执行业务规则。

8. 5. 6. #邮件任务

activiti强化了业务流程，支持了自动邮件任务，它可以发送邮件给一个或多个参与者，包括支持cc, bcc, HTML内容等等。注意邮件任务不是BPMN 2.0规范定义的官方任务。（它也没有对应的图标）。因此，activiti中邮件任务是用专门的服务任务实现的。

8. 5. 6. 1. #邮件服务器配置

activiti引擎要通过支持SMTP功能的外部邮件服务器发送邮件。为了实际发送邮件，引擎突 知道如何访问邮件服务器。下面的配置可以设置到activiti.cfg.xml配置文件中：

表 8.1. 邮件服务器配置

属性	是否必须	描述
mailServerHost	否	邮件服务器的主机名（比如： mail.mycorp.com）。默认为localhost
mailServerPort	是，如果没有使用默认端口	邮件服务器上的SMTP传输端口。默认为25
mailServerDefaultFrom	否	如果用户没有指定发送邮件的邮件地址，默认设置的发送者的邮件地址。默认为activiti@activiti.org
mailServerUsername	如果服务器需要	一些邮件服务器需要认证才能发送邮件。默认不设置。
mailServerPassword	如果服务器需要	一些邮件服务器需要认证才能发送邮件。默认不设置。
mailServerUseSSL	如果服务器需要	一些邮件服务器需要ssl交互。默认为false。
mailServerUseTLS	如果服务器需要	一些邮件服务器（比如gmail）需要支持TLS。默认为false。

8. 5. 6. 2. #定义一个邮件任务

邮件任务是一个专用的服务任务，这个服务任务的type设置为'mail'。

```
<serviceTask id="sendMail" activiti:type="mail">
```

邮件任务是通过属性注入进行配置的。所有这些属性都可以使用EL表达式，可以在流程执行中解析。下面的属性都可以设置：

表 8.2. 邮件服务器配置

属性	是否必须	描述
to	是	邮件的接受者。可以使用逗号分隔多个接受者
from	否	邮件发送者的地址。如果不提供，会使用默认配置的地址。
subject	否	邮件的主题
cc	否	邮件抄送人。可以使用逗号分隔多个接收者
bcc	否	邮件暗送人。可以使用逗号分隔多个接收者
charset	否	可以修改邮件的字符集，对很多非英语语言是必须设置的。
html	否	作为邮件内容的HTML。
text	否	邮件的内容，在需要使用原始文字（非富文本）的邮件时使用。可以与html一起使用，对于不支持富客户端的邮件客户端。客户端会降级到仅显示文本的方式。
htmlVar	否	使用对应的流程变量作为e-mail的内容。它和html的不同之处是它内容中包含的表达式会在mail任务发送之前被替换掉。
textVar	否	使用对应的流程变量作为e-mail的纯文本内容。它和html的不同之处是它内容中包含的表达式会在mail任务发送之前被替换掉。
ignoreException	否	处理邮件失败时，是否忽略异常，不抛出ActivitiException，默认为false。
exceptionVariableName	否	当设置了ignoreException = true处理email时不抛出异常，可以指定一个变量名来存储失败信息。

8.5.6.3. #使用实例

下面的XML演示了使用邮件任务的例子。

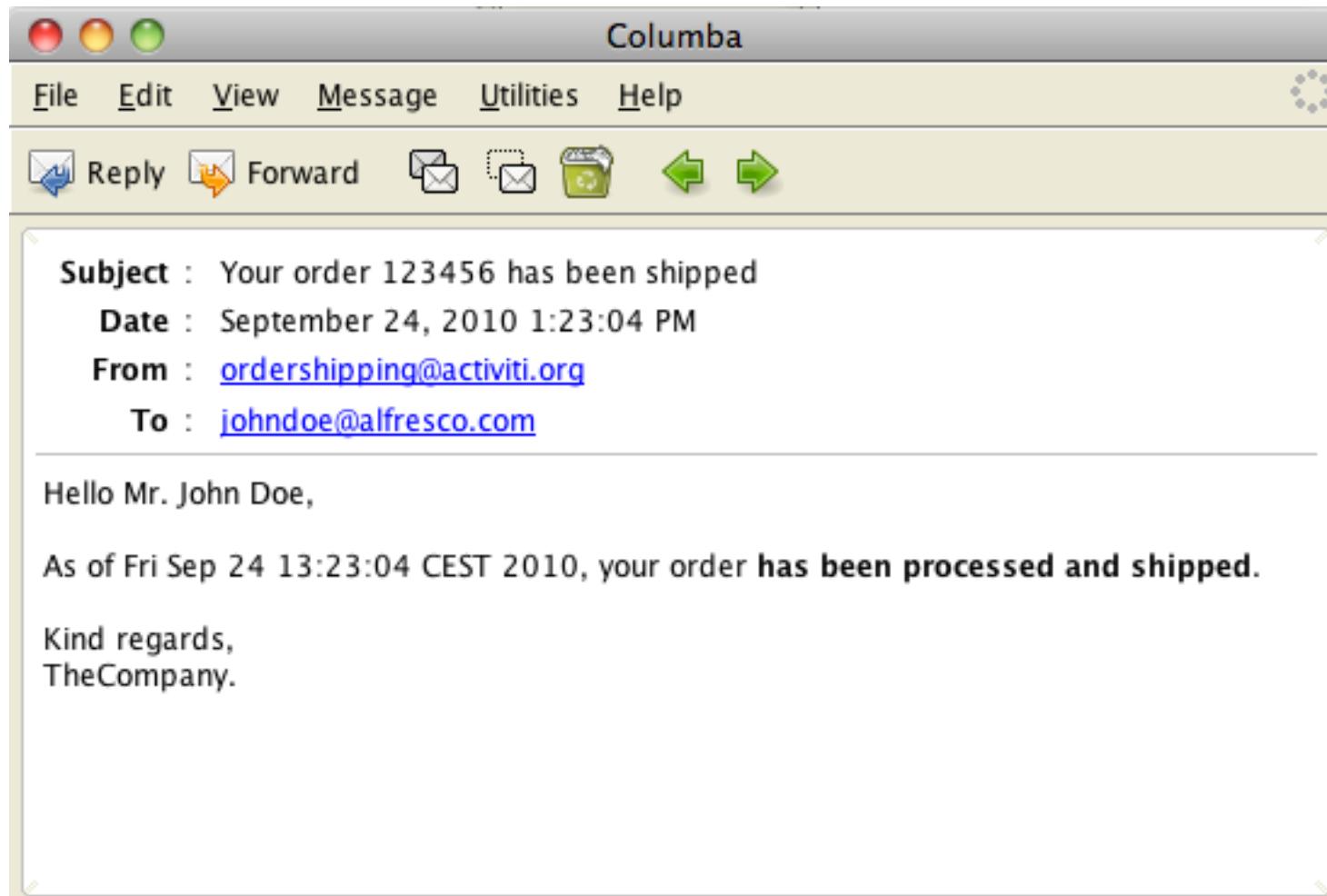
```
<serviceTask id="sendMail" activiti:type="mail">
```

```

<extensionElements>
    <activiti:field name="from" stringValue="order-shipping@thecompany.com" />
    <activiti:field name="to" expression="${recipient}" />
    <activiti:field name="subject" expression="Your order ${orderId} has been shipped" />
    <activiti:field name="html">
        <activiti:expression>
            <![CDATA[
                <html>
                    <body>
                        Hello ${male ? 'Mr.' : 'Mrs.' } ${recipientName},<br/><br/>
                        As of ${now}, your order has been <b>processed and shipped</b>.<br/><br/>
                        Kind regards,<br/>
                        TheCompany.
                    </body>
                </html>
            ]]>
        </activiti:expression>
    </activiti:field>
</extensionElements>
</serviceTask>

```

结果如下：



8. 5. 7. #Mule任务

mule任务可以向mule发送消息，以强化activiti的集成能力。注意mule任务不是BPMN 2.0规范定义的官方任务。（它也没有对应的图标）。因此，activiti中mule任务是用专门的服务任务实现的。

8. 5. 7. 1. #定义一个mule任务

mule任务是一个专用的服务任务，这个服务任务的type设置为'mule'。

```
<serviceTask id="sendMule" activiti:type="mule">
```

mule任务是通过属性注入进行配置的。所有这些属性都可以使用EL表达式，可以在流程执行中解析。下面的属性都可以设置：

表 8.3. Mule服务器配置

属性	是否必须	描述
endpointUrl	是	希望调用的Mule终端
language	是	你要使用解析荷载表达式(payloadExpression) 属性的语言。
payloadExpression	是	作为消息荷载的表达式。
resultVariable	否	将要保存调用结果的变量名称。

8. 5. 7. 2. #应用实例

下面是一个使用mule任务的例子。

```
<extensionElements>
    <activiti:field name="endpointUrl">
        <activiti:string>vm://in</activiti:string>
    </activiti:field>
    <activiti:field name="language">
        <activiti:string>juel</activiti:string>
    </activiti:field>
    <activiti:field name="payloadExpression">
        <activiti:string>"hi"</activiti:string>
    </activiti:field>
    <activiti:field name="resultVariable">
        <activiti:string>theVariable</activiti:string>
    </activiti:field>
</extensionElements>
```

8. 5. 8. #Camel任务

Camel任务可以从Camel发送和介绍消息，由此强化了activiti的集成功能。注意camel任务不是BPMN 2.0规范定义的官方任务。（它也没有对应的图标）。在activiti中，camel任

务时由专用的服务任务实现的。 要使用camel任务功能时，也要记得吧activiti camel包含到项目里。

8.5.8.1. #定义camel任务

camel任务是一个专用的服务任务， 这个服务任务的type设置为' camel'。

```
<serviceTask id="sendCamel" activiti:type="camel">
```

流程定义只需要在服务任务中定义camel类型。 集成逻辑都会代理给camel容器。默认activiti引擎会在spring容器中查找camelContext bean。 camelContext定义了camel容器加载的路由规则。下面的例子中路由规则是从指定的java包下加载的。 但是你也可以通过spring配置直接定义路由规则。

```
<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>org.activiti.camel.route</package>
  </packageScan>
</camelContext>
```

如果想了解更多关于camel路由的信息，可以访问Camel的网站 [http://camel.apache.org/]。 在这里只通过很小的例子演示了基础的概念。 在第一个例子中，我们会通过activiti工作流实现最简单的Camel调用。我们称其为SimpleCamelCall。

如果想定义多个Camel环境bean，并且（或者）想使用不同的bean名称，可以重载CamelTask的定义，如下所示：

```
<serviceTask id="serviceTask1" activiti:type="camel">
  <extensionElements>
    <activiti:field name="camelContext" stringValue="customCamelContext" />
  </extensionElements>
</serviceTask>
```

8.5.8.2. #简单Camel调用

这个例子对应的文件都可以在activiti org.activiti.camel.examples.simpleCamelCall包下找到。我们的目标是简单激活一个特定的camel路由。 首先，我们需要一个Spring环境，它要包含之前介绍的路由。这些文件的目的如下：

```
<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>org.activiti.camel.examples.simpleCamelCall</package>
  </packageScan>
</camelContext>
```

包含名为SimpleCamelCallRoute的路由的类文件，放在PackageScan标签的扫描目录下。 下面就是路由的定义：

```
public class SimpleCamelCallRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        from("activiti:SimpleCamelCallProcess:simpleCall").to("log: org.activiti.camel.examples.SimpleCamelCall");
    }
}
```

这个规则仅仅打印消息体，不会做其他事情。注意终端的格式。它包含三部分：

表 8.4. 终端URL:

部分	说明
终端URL	引用activiti终端
SimpleCamelCallProcess	流程名
simpleCall	流程中的Camel服务

OK，我们的规则已经配置好，也可以让Camel使用了。现在看工作流部分。工作流看起来像这样：

```
<process id="SimpleCamelCallProcess">
    <startEvent id="start"/>
    <sequenceFlow id="flow1" sourceRef="start" targetRef="simpleCall"/>

    <serviceTask id="simpleCall" activiti:type="camel"/>

    <sequenceFlow id="flow2" sourceRef="simpleCall" targetRef="end"/>
    <endEvent id="end"/>
</process>
```

在serviceTask部分，它只注明服务的类型是Camel，目标规则名为simpleCall。这与上面的activiti终端相匹配。初始化流程后，我们会看到一个空的日志。好，我们已经完成了这个最简单的例子了。

8.5.8.3. #乒乓实例

我们的例子成功执行了，但是Camel和Activiti之间没有任何交互，而且这样做也没有任何优势。在这个例子里，我们尝试向Camel发送和接收数据。我们发送一个字符串，camel进行一些处理，然后返回结果。发送部分很简单，我们把变量里的消息发送给camel。这里是我们的调用代码：

```
@Deployment
public void testPingPong() {
    Map<String, Object> variables = new HashMap<String, Object>();

    variables.put("input", "Hello");
    Map<String, String> outputMap = new HashMap<String, String>();
    variables.put("outputMap", outputMap);

    runtimeService.startProcessInstanceByKey("PingPongProcess", variables);
    assertEquals(1, outputMap.size());
    assertNotNull(outputMap.get("outputValue"));
    assertEquals("Hello World", outputMap.get("outputValue"));
```

```
}
```

变量“input”是Camel规则的实际输入，outputMap会记录camel返回的结果。流程应该像是这样：

```
<process id="PingPongProcess">
  <startEvent id="start"/>
  <sequenceFlow id="flow1" sourceRef="start" targetRef="ping"/>
  <serviceTask id="ping" activiti:type="camel"/>
  <sequenceFlow id="flow2" sourceRef="ping" targetRef="saveOutput"/>
  <serviceTask id="saveOutput" activiti:class="org.activiti.examples.pingPong.SaveOutput" />
  <sequenceFlow id="flow3" sourceRef="saveOutput" targetRef="end"/>
  <endEvent id="end"/>
</process>
```

注意，SaveOutput这个serviceTask，会把“Output”变量的值从上下文保存到上面提到的OutputMap中。现在，我们必须了解变量是如何发送给Camel，再返回的。这里就要涉及到camel实际执行的行为了。变量提交给camel的方法是由CamelBehavior控制的。这里我们使用默认的配置，其他的会在后面提及。使用这些代码，我们就可以配置一个期望的camel行为：

```
<serviceTask id="serviceTask1" activiti:type="camel">
  <extensionElements>
    <activiti:field name="camelBehaviorClass" stringValue="org.activiti.camel.impl.CamelBehaviorCamelBodyImpl" />
  </extensionElements>
</serviceTask>
```

如果你没有特别指定一个行为，就会使用org.activiti.camel.impl.CamelBehaviorDefaultImpl。这个行为会把变量复制成名称相同的Camel属性。在返回时，无论选择什么行为，如果camel消息体是一个map，每个元素都会复制成一个变量，否则整个对象会复制到指定名称为“camelBody”的变量中。了解这些后，就可以看看我们第二个例子的camel规则了：

```
@Override
public void configure() throws Exception {
  from("activiti:PingPongProcess:ping").transform().simple("${property.input} World");
}
```

在这个规则中，字符串“world”会被添加到“input”属性的后面，结果会写入消息体。这时可以检查javaServiceTask中的“camelBody”变量，复制到“outputMap”中，并在testcase进行判断。现在这个例子是在默认的行为下运行的，然后我们看一起其他的方案。在启动的所有camel规则中，流程实例id会复制到camel的名为“PROCESS_ID_PROPERTY”的属性中。后续可以用它关联流程实例和camel规则。他也可以在camel规则中直接使用。

Activiti中可以使用三种不同的行为。这些行为可以通过在规则URL中指定对应的环节来实现覆盖。这里有一个在URL中覆盖现存行为的例子：

```
from("activiti:asyncCamelProcess:serviceTaskAsync2?copyVariablesToProperties=true").
```

下面的表格提供了三种camel行为的概述：

表 8.5. 已有的camel行为：

行为	URL	描述
CamelBehaviorDefaultImpl	copyVariablesToProperties	把Activiti变量复制为Camel属性
CamelBehaviorCamelBodyImpl	copyCamelBodyToBody	只把名为"camelBody"Activiti变量复制成camel的消息体
CamelBehaviorBodyAsMapImpl	copyVariablesToBodyAsMap	把activiti的所有变量复制到一个map里，作为Camel的消息体

上面的表格解释和activiti变量如何传递给camel。下面的表格解释和camel的变量如何返回给activiti。它只能配置在规则URL中。

表 8.6. 已有的camel行为：

Url	描述
默认	如果Camel消息体是一个map，把每个元素复制成activiti的变量，否则把整个camel消息体作为activiti的"camelBody"变量。
copyVariablesFromProperties	将Camel属性以相同名称复制为Activiti变量
copyCamelBodyToBodyAsString	和默认一样，但是如果camel消息体不是map时，先把它转换成字符串，再设置为"camelBody"。
copyVariablesFromHeader	额外把camel头部以相同名称复制成Activiti变量

例子的源码放在activiti-camel模块的org.activiti.camel.examples.pingPong包下。

8.5.8.4. #异步乒乓实例

之前的例子都是同步的。流程会等到camel规则返回之后才会停止。一些情况下，我们需要activiti工作流继续运行。这时camelServiceTask的异步功能就特别有用。你可以通过设置camelServiceTask的async属性来启用这个功能。

```
<serviceTask id="serviceAsyncPing" activiti:type="camel" activiti:async="true"/>
```

通过设置这个功能，camel规则会被activiti的jobExecutor异步执行。当你在camel规则中定义了一个队列，activiti流程会在camelServiceTask执行时继续运行。camel规则会以完全异步的方式执行。如果你想在什么地方等待camelServiceTask的返回值，你可以使用一个receiveTask。

```
<receiveTask id="receiveAsyncPing" name="Wait State" />
```

流程实例会等到接收一个signal，比如来自camel。在camel中你可以发送一个signal给流程实例，通过对称的activiti终端发送消息。

```
from("activiti:asyncPingProcess:serviceAsyncPing").to("activiti:asyncPingProcess:receiveAsyncPing");
```

对于一个常用的终端，会使用冒号分隔的三个部分：

- 常量字符串“activiti”
- 流程名称
- 接收任务名

8.5.8.5. #从camel规则中实例化工作流

之前的所有例子中，activiti工作流会先启动，然后在流程中启动camel规则。也可以使用另外一种方法。在已经启动的camel规则中启动一个工作流。这会触发一个receiveTask十分类似，除了最后的部分。这是一个实例规则：

```
from("direct:start").to("activiti:camelProcess");
```

我们看到url有两个部分，第一个部分是常量字符串“activiti”，第二部分是流程的名称。很明显，流程应该已经部署完成，并且是可以启动的。

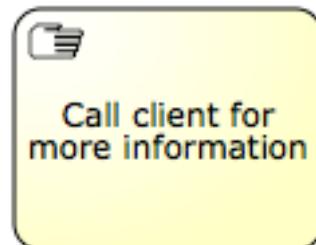
8.5.9. #手工任务

8.5.9.1. #描述

手工任务定义了BPM引擎外部的任务。用来表示工作需要某人完成，而引擎不需要知道，也没有对应的系统和UI接口。对于引擎，手工任务是直接通过的活动，流程到达它之后会自动向下执行。

8.5.9.2. #图形标记

手工任务显示为一个圆角矩形，左上角是一个手型小图标。



8.5.9.3. #XML内容

```
<manualTask id="myManualTask" name="Call client for more information" />
```

8. 5. 10. #Java接收任务

8. 5. 10. 1. #描述

接收任务是一个简单任务，它会等待对应消息的到达。当前，我们只实现了这个任务的java语义。当流程达到接收任务，流程状态会保存到存储里。意味着流程会等待在这个等待状态，直到引擎接收了一个特定的消息，这会触发流程穿过接收任务继续执行。

8. 5. 10. 2. #图形标记

接收任务显示为一个任务（圆角矩形），右上角有一个消息小标记。消息是白色的（黑色图标表示发送语义）



8. 5. 10. 3. #XML内容

```
<receiveTask id="waitForSignal" name="waitForSignal" />
```

要在接收任务等待的流程实例继续执行，可以调用runtimeService.signal(executionId)，传递接收任务上流程的id。下面的代码演示了实际是如何工作的：

```
ProcessInstance pi = runtimeService.startProcessInstanceByKey("receiveTask");
Execution execution = runtimeService.createExecutionQuery()
    .processInstanceId(pi.getId())
    .activityId("waitForSignal")
    .singleResult();
assertNotNull(execution);

runtimeService.signal(execution.getId());
```

8. 5. 11. #Shell任务

8. 5. 11. 1. #描述

shell任务可以执行shell脚本和命令。注意shell任务不是BPMN 2.0规范定义的官方任务。（它也没有对应的图标）。

8. 5. 11. 2. #定义shell任务

shell任务是一个专用的服务任务，这个服务任务的type设置为'shell'。

```
<serviceTask id="shellEcho" activiti:type="shell">
```

shell任务使用属性注入进行配置。所有属性都可以包含EL表达式，会在流程执行过程中解析。可以配置以下属性：

表 8.7. Shell任务参数配置

属性	是否必须	类型	描述	默认值
command	是	String	执行的shell命令	
arg0-5	否	String	参数0至5	
wait	否	true/false	是否需要等待到 shell进程结束	true
redirectError	否	true/false	把标准错误打印到 标准流中	false
cleanEnv	否	true/false	shell进行不继承 当前环境	false
outputVariable	否	String	保存输出的变量名	不会记录输出结果
errorCodeVariable	否	String	包含结果错误代码 的变量名	不会注册错误级别的变量名
directory	否	String	shell进程的默认 目录	当前目录

8.5.11.3. #应用实例

下面的代码演示了使用shell任务的实例。它会执行shell脚本“cmd /c echo EchoTest”，等到它结束，再把输出结果保存到resultVar中。

```
<serviceTask id="shellEcho" activiti:type="shell" >
  <extensionElements>
    <activiti:field name="command" stringValue="cmd" />
    <activiti:field name="arg1" stringValue="/c" />
    <activiti:field name="arg2" stringValue="echo" />
    <activiti:field name="arg3" stringValue="EchoTest" />
    <activiti:field name="wait" stringValue="true" />
    <activiti:field name="outputVariable" stringValue="resultVar" />
  </extensionElements>
</serviceTask>
```

8.5.12. #执行监听器

兼容性提醒：在发布5.3后，我们发现执行监听器，开API。这些类在org.activiti.engine.impl...的子包，org.activiti.engine.impl.pvm.delegate.ExecutionListener, org.activiti.engine.impl.pvm.delegate.TaskListener and org.activiti.engine.impl.pvm.el.Expression已经废弃了。从现在开始，应该使用org.activiti.engine.delegate.ExecutionListener, org.activiti.engine.delegate.TaskListener 和 org.activiti.engine.delegate.Expression。在新的公开API中，删除了ExecutionListener.getEventSource()。因为已经设置了废弃编译警告，所以已存的代码应该可以正常运行。但是要考虑切换到新的公共API接口（包名中没有.impl.）。

任务监听器，表达式还是非公包名中有一个impl。

执行监听器可以执行外部java代码或执行表达式，当流程定义中发生了某个事件。可以捕获的事件有：

- 流程实例的启动和结束。
- 选中一条连线。
- 节点的开始和结束。
- 网关的开始和结束。
- 中间事件的开始和结束。
- 开始时间结束或结束事件开始。

下面的流程定义包含了3个流程监听器：

```
<process id="executionListenersProcess">

    <extensionElements>
        <activiti:executionListener class="org.activiti.examples.bpmn.executionlistener.ExampleExecutionListenerOne" />
    </extensionElements>

    <startEvent id="theStart" />
    <sequenceFlow sourceRef="theStart" targetRef="firstTask" />

    <userTask id="firstTask" />
    <sequenceFlow sourceRef="firstTask" targetRef="secondTask" />
    <extensionElements>
        <activiti:executionListener class="org.activiti.examples.bpmn.executionlistener.ExampleExecutionListenerTwo" />
    </extensionElements>
    </sequenceFlow>

    <userTask id="secondTask" />
    <extensionElements>
        <activiti:executionListener expression="#{myPojo.myMethod(execution.event)}" event="end" />
    </extensionElements>
    </userTask>
    <sequenceFlow sourceRef="secondTask" targetRef="thirdTask" />

    <userTask id="thirdTask" />
    <sequenceFlow sourceRef="thirdTask" targetRef="theEnd" />

    <endEvent id="theEnd" />

</process>
```

第一个流程监听器监听流程开始。监听器是一个外部java类（像是ExampleExecutionListenerOne），需要实现org.activiti.engine.delegate.ExecutionListener接口。当事件发生时（这里是end事件），会调用notify(ExecutionListenerExecution execution)方法。

```
public class ExampleExecutionListenerOne implements ExecutionListener {

    public void notify(ExecutionListenerExecution execution) throws Exception {
        execution.setVariable("variableSetInExecutionListener", "theValue");
        execution.setVariable("eventReceived", execution.getEventName());
    }
}
```

也可以使用实现org.activiti.engine.delegate.JavaDelegate接口的代理类。代理类可以在结构中重用，比如serviceTask的代理。

第二个流程监听器在连线执行时调用。注意这个listener元素不能定义event，因为连线只能触发take事件。为连线定义的监听器的event属性会被忽略。

最后一个流程监听器在节点secondTask结束时调用。这里使用expression代替class来在事件触发时执行/调用。

```
<activiti:executionListener expression="${myPojo.myMethod(execution.eventName)}" event="end" />
```

和其他表达式一样，流程变量可以处理和使用。因为流程实现对象有一个保存事件名称的属性，可以在方法中使用execution.eventName获的事件名称。

流程监听器也支持使用delegateExpression，和服务任务相同。

```
<activiti:executionListener event="start" delegateExpression="${myExecutionListenerBean}" />
```

在activiti 5.12中，我们也介绍了新的流程监听器，org.activiti.engine.impl.bpmn.listener.ScriptExecutionListener。这个脚本流程监听器可以为某个流程监听事件执行一段脚本。

```
<activiti:executionListener event="start" class="org.activiti.engine.impl.bpmn.listener.ScriptExecutionListener">
  <activiti:field name="script">
    <activiti:string>
      def bar = "BAR"; // local variable
      foo = "FOO"; // pushes variable to execution context
      execution.setVariable("var1", "test"); // test access to execution instance
      bar // implicit return value
    </activiti:string>
  </activiti:field>
  <activiti:field name="language" stringValue="groovy" />
  <activiti:field name="resultVariable" stringValue="myVar" />
<activiti:executionListener>
```

8.5.12.1. #流程监听器的属性注入

使用流程监听器时，可以配置class属性，可以使用属性注入。这和使用服务任务属性注入相同，参考它可以获得属性注入的很多信息。

下面的代码演示了使用了属性注入的流程监听器的流程的简单例子。

```
<process id="executionListenersProcess">
  <extensionElements>
    <activiti:executionListener class="org.activiti.examples.bpmn.executionListener.ExampleFieldInjectedExecutionListener">
      <activiti:field name="fixedValue" stringValue="Yes, I am" />
      <activiti:field name="dynamicValue" expression="${myVar}" />
    </activiti:executionListener>
  </extensionElements>

  <startEvent id="theStart" />
  <sequenceFlow sourceRef="theStart" targetRef="firstTask" />

  <userTask id="firstTask" />
  <sequenceFlow sourceRef="firstTask" targetRef="theEnd" />

  <endEvent id="theEnd" />
</process>
```

```

public class ExampleFieldInjectedExecutionListener implements ExecutionListener {

    private Expression fixedValue;

    private Expression dynamicValue;

    public void notify(ExecutionListenerExecution execution) throws Exception {
        execution.setVariable("var", fixedValue.getValue(execution).toString() + dynamicValue.getValue(execution).t
    }
}

```

ExampleFieldInjectedExecutionListener类串联了两个注入的属性。（一个是固定的，一个是动态的），把他们保存到流程变量' var' 中。

```

@Deployment(resources = {"org/activiti/examples/bpmn/executionListener/ExecutionListenersFieldInjectionProcess.bpmn20.xml"})
public void testExecutionListenerFieldInjection() {
    Map<String, Object> variables = new HashMap<String, Object>();
    variables.put("myVar", "listening!");

    ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("executionListenersProcess", variables);

    Object varSetByListener = runtimeService.getVariable(processInstance.getId(), "var");
    assertNotNull(varSetByListener);
    assertTrue(varSetByListener instanceof String);

    // Result is a concatenation of fixed injected field and injected expression
    assertEquals("Yes, I am listening!", varSetByListener);
}

```

8. 5. 13. #任务监听器

任务监听器可以在发生对应的任务相关事件时执行自定义java逻辑 或表达式。

任务监听器只能添加到流程定义中的用户任务中。注意它必须定义在BPMN 2.0 extensionElements的子元素中，并使用activiti命名空间，因为任务监听器是activiti独有的结构。

```

<userTask id="myTask" name="My Task" >
    <extensionElements>
        <activiti:taskListener event="create" class="org.activiti.MyTaskCreateListener" />
    </extensionElements>
</userTask>

```

任务监听器支持以下属性：

- event（必选）：任务监听器会被调用的任务类型。可能的类型为：
 - create：任务创建并设置所有属性后触发。
 - assignment：任务分配给一些人时触发。当流程到达userTask，assignment事件会在create事件之前发生。这样的顺序似乎不自然，但是原因很简单：当获得create时间时，我们想获得任务的所有属性，包括执行人。

- **complete:** 当任务完成，并尚未从运行数据中删除时触发。
- **delete:** 只在任务删除之前发生。注意在通过completeTask正常完成时，也会执行。
- **class:** 必须调用的代理类。这个类必须实现org.activiti.engine.delegate.TaskListener接口。

```
public class MyTaskCreateListener implements TaskListener {
    public void notify(DelegateTask delegateTask) {
        // Custom logic goes here
    }
}
```

可以使用属性注入把流程变量或执行传递给代理类。注意代理类的实例是在部署时创建的（和activiti中其他类代理的情况一样），这意味着所有流程实例都会共享同一个实例。

- **expression:**（无法同时与class属性一起使用）：指定事件发生时执行的表达式。可以把DelegateTask对象和事件名称（使用task.eventName）作为参数传递给调用的对象。

```
<activiti:taskListener event="create" expression="${myObject.callMethod(task, task.eventName)}" />
```

- **delegateExpression**可以指定一个表达式，解析一个实现了TaskListener接口的对象，这与服务任务一致。

```
<activiti:taskListener event="create" delegateExpression="${myTaskListenerBean}" />
```

- 在activiti 5.12中，我们也介绍了新的任务监听器，org.activiti.engine.impl.bpmn.listener.ScriptTaskListener。脚本任务监听器可以为任务监听器事件执行脚本。

```
<activiti:taskListener event="complete" class="org.activiti.engine.impl.bpmn.listener.ScriptTaskListener">
<activiti:field name="script">
<activiti:string>
    def bar = "BAR"; // local variable
    foo = "FOO"; // pushes variable to execution context
    task.setOwner("kermit"); // test access to task instance
    bar // implicit return value
</activiti:string>
</activiti:field>
<activiti:field name="language" stringValue="groovy" />
<activiti:field name="resultVariable" stringValue="myVar" />
<activiti:taskListener>
```

8.5.14. #多实例（循环）

8.5.14.1. #描述

多实例节点是在业务流程中定义重复环节的一个方法。从开发角度讲，多实例和循环是一样的：它可以根据给定的集合，为每个元素执行一个环节甚至一个完整的子流程，既可以顺序依次执行也可以并发同步执行。

多实例是在一个普通的节点上添加了额外的属性定义（所以叫做‘多实例特性’），这样运行时节点就会执行多次。下面的节点都可以成为一个多实例节点：

- User Task
- Script Task
- Java Service Task
- Web Service Task
- Business Rule Task
- Email Task
- Manual Task
- Receive Task
- (Embedded) Sub-Process
- Call Activity

网关和事件 不能设置多实例。

根据规范的要求，每个上级流程为每个实例创建分支时都要提供如下变量：

- nrOfInstances: 实例总数
 - nrOfActiveInstances: 当前活动的，比如，还没完成的，实例数量。 对于顺序执行的多实例，值一直为1。
 - nrOfCompletedInstances: 已经完成实例的数目。
- 可以通过`execution.getVariable(x)`方法获得这些变量。

另外，每个创建的分支都会有分支级别的本地变量（比如，其他实例不可见， 不会保存到流程实例级别）：

- loopCounter: 表示特定实例的在循环的索引值。可以使用activiti 的`elementIndexVariable`属性修改loopCounter的变量名。

8. 5. 14. 2. #图形标记

如果节点是多实例的，会在节点底部显示三条短线。 三条竖线表示实例会并行执行。 三条横线表示顺序执行。



8. 5. 14. 3. #Xml 内容

要把一个节点设置为多实例，节点`xml`元素必须设置一个`multiInstanceLoopCharacteristics`子元素。

```
<multiInstanceLoopCharacteristics isSequential="false|true">
...
</multiInstanceLoopCharacteristics>
```

isSequential属性表示节点是进行 顺序执行还是并行执行。

实例的数量会在进入节点时计算一次。
有一些方法配置它。一种方法是使
用loopCardinality子元素直接指定一个数字。

```
<multiInstanceLoopCharacteristics isSequential="false|true">
  <loopCardinality>5</loopCardinality>
</multiInstanceLoopCharacteristics>
```

也可以使用结果为整数的表达式：

```
<multiInstanceLoopCharacteristics isSequential="false|true">
  <loopCardinality>${nrOfOrders-nrOfCancellations}</loopCardinality>
</multiInstanceLoopCharacteristics>
```

另一个定义实例数目的方法是，通过loopDataInputRef子元素，设置一个类型为集合的流程变量名。 对于集合中的每个元素，都会创建一个实例。 也可以通过inputDataItem子元素指定集合。 下面的代码演示了这些配置：

```
<userTask id="miTasks" name="My Task ${loopCounter}" activiti:assignee="${assignee}">
  <multiInstanceLoopCharacteristics isSequential="false">
    <loopDataInputRef>assigneeList</loopDataInputRef>
    <inputDataItem name="assignee" />
  </multiInstanceLoopCharacteristics>
</userTask>
```

假设assigneeList变量包含这些值[kermit, gonzo, fozie]。 在上面代码中，三个用户任务会同时创建。每个分支都会拥有一个用名为assignee的流程变量，这个变量会包含集合中的对应元素，在例子中会用来设置用户任务的分配者。

loopDataInputRef和inputDataItem的缺点是1) 名字不好记， 2) 根据BPMN 2.0格式定义，它们不能包含表达式。activiti通过在 multiInstanceCharacteristics中设置 collection和 elementVariable属性解决了这个问题：

```
<userTask id="miTasks" name="My Task" activiti:assignee="${assignee}">
  <multiInstanceLoopCharacteristics isSequential="true"
    activiti:collection="${myService.resolveUsersForTask()}" activiti:elementVariable="assignee" >
  </multiInstanceLoopCharacteristics>
</userTask>
```

多实例节点在所有实例都完成时才会结束。也可以指定一个表达式在每个实例结束时执行。如果表达式返回true，所有其他的实例都会销毁，多实例节点也会结束，流程会继续执行。这个表达式必须定义在completionCondition子元素中。

```
<userTask id="miTasks" name="My Task" activiti:assignee="${assignee}">
  <multiInstanceLoopCharacteristics isSequential="false"
    activiti:collection="assigneeList" activiti:elementVariable="assignee" >
      <completionCondition>${nrOfCompletedInstances/nrOfInstances} >= 0.6 </completionCondition>
```

```

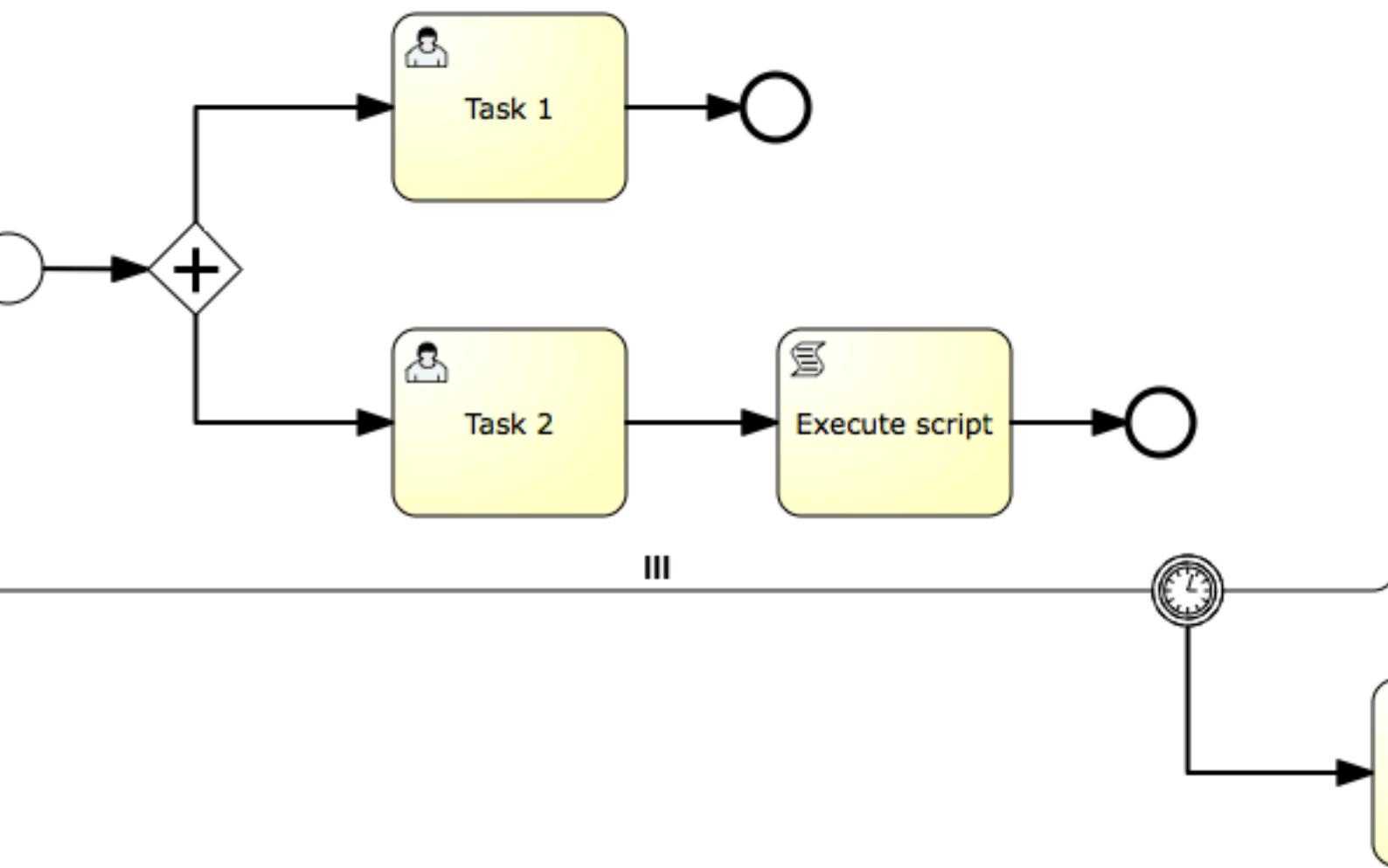
</multiInstanceLoopCharacteristics>
</userTask>

```

在这里例子中，会为assigneeList集合的每个元素创建一个并行的实例。当60%的任务完成时，其他任务就会删除，流程继续执行。

8. 5. 14. 4. #边界事件和多实例

因为多实例是一个普通节点，它也可以在边缘使用边界事件。对于中断型边界事件，当捕获事件时，所有激活的实例都会销毁。参考以下多实例子流程：



这里，子流程的所有实例都会在定时器触发时销毁，无论有多少实例，也不管内部哪个节点没有完成。

8. 5. 15. #补偿处理器

8. 5. 15. 1. #描述

[EXPERIMENTAL]

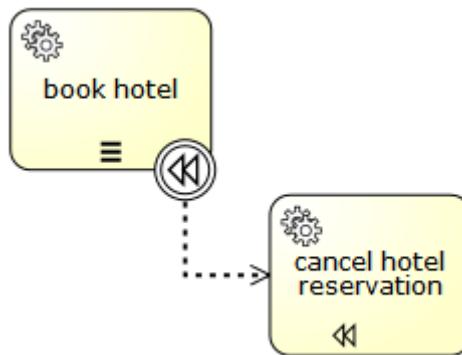
如果一个节点用来补偿另一个节点的业务，它可以声明为一个补偿处理器。补偿处理器不包含普通的流，只在补偿事件触发时执行。

补偿处理器不能包含进入和外出顺序流。

补偿处理器必须使用直接关联分配给一个补偿边界事件。

8. 5. 15. 2. #图形标志

如果节点是补偿处理器，补偿事件图标会显示在中间底部区域。下面的流程图显示了一个服务任务，附加了一个补偿边界事件，并分配了一个补偿处理器。注意“cancel hotel reservation”服务任务中间底部区域显示的补偿处理器图标。



8. 5. 15. 3. #XML内容

为了声明作为补偿处理器的节点，我们需要把isForCompensation设置为true：

```

<serviceTask id="undoBookHotel" isForCompensation="true" activiti:class="...">
</serviceTask>
  
```

8. 6. #子流程和调用节点

8. 6. 1. #子流程

8. 6. 1. 1. #描述

子流程（Sub-process）是一个包含其他节点，网关，事件等等的节点。它自己就是一个流程，同时是更大流程的一部分。子流程是完全定义在父流程里的（这就是为什么叫做内嵌子流程）。

子流程有两种主要场景：

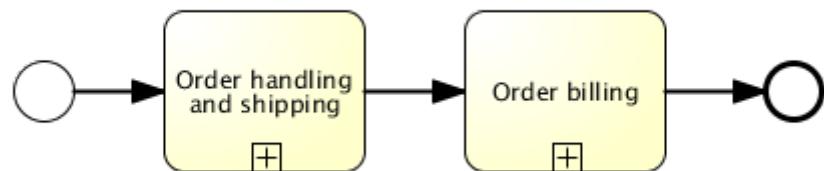
- 子流程可以使用继承式建模。很多建模工具的子流程可以折叠，把子流程的内部细节隐藏，显示一个高级别的端对端的业务流程总览。
- 子流程会创建一个新的事件作用域。子流程运行过程中抛出的事件，可以被子流程边缘定义的边界事件捕获，这样就可以创建一个仅限于这个子流程的事件作用范围。

使用子流程要考虑如下限制：

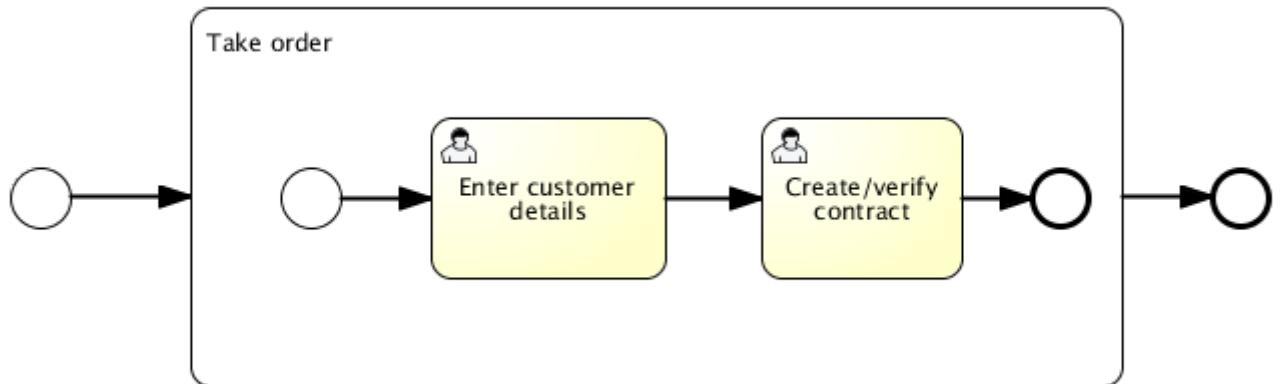
- 子流程只能包含一个空开始事件，不能使用其他类型的开始事件。子流程必须至少有一个结束节点。注意，BPMN 2.0规范允许忽略子流程的开始和结束节点，但是当前activiti的实现并不支持。
- 顺序流不能跨越子流程的边界。

8. 6. 1. 2. #图形标记

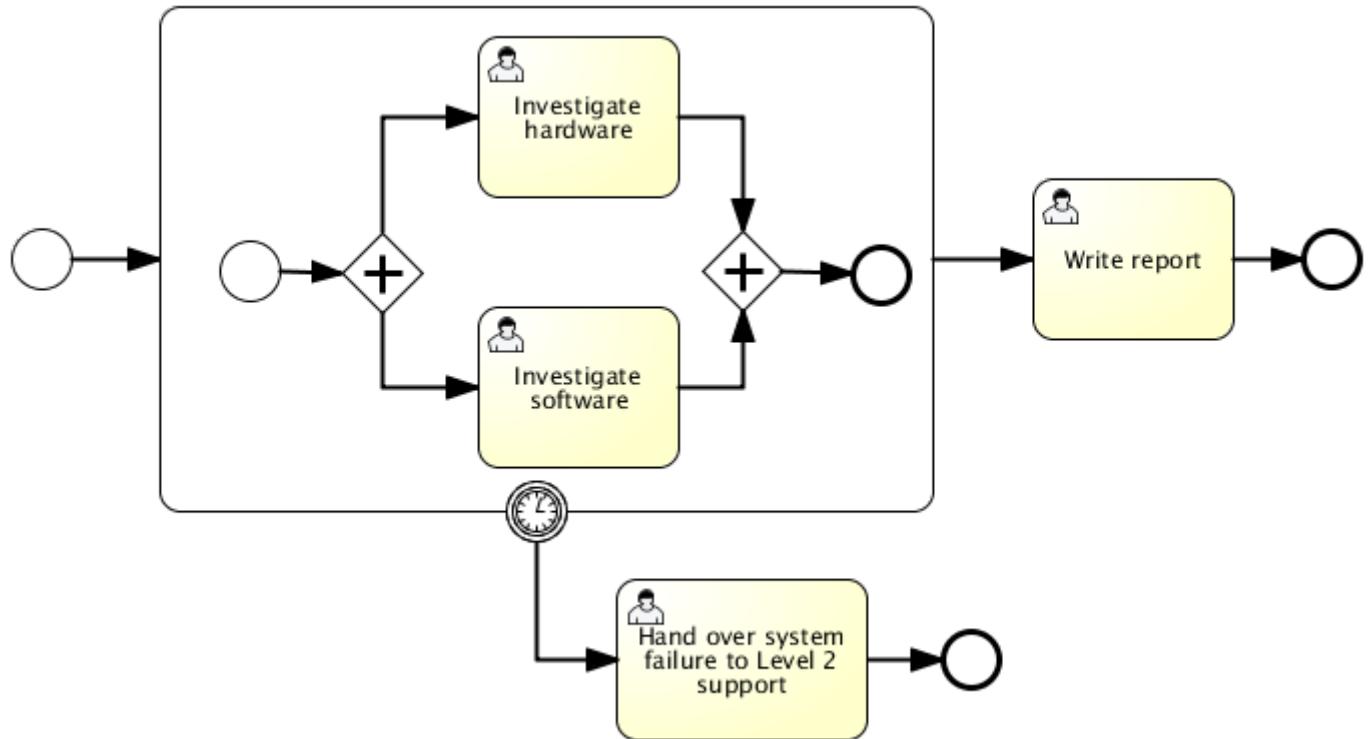
子流程显示为标准的节点，圆角矩形。这时子流程是折叠的，只显示名称和一个加号标记，展示了高级别的流程总览：



这时子流程是展开的，子流程的步骤都显示在子流程边界内：



使用子流程的主要原因，是定义对应事件的作用域。下面流程模型演示了这个功能：调查软件/调查引荐任务需要同步执行，两个任务需要在同时完成，在二线支持解决之前。这里，定时器的作用域（比如，节点需要及时完成）是由子流程限制的。



8.6.1.3. #XML内容

子流程定义为subprocess元素。所有节点，网关，事件，等等。它是子流程的一部分，需要放在这个元素里。

```

<subProcess id="subProcess">
    <startEvent id="subProcessStart" />
    ...
    other Sub-Process elements ...
    <endEvent id="subProcessEnd" />
</subProcess>

```

8. 6. 2. #事件子流程

8. 6. 2. 1. #描述

事件子流程是BPMN 2.0中的新元素。事件子流程是由事件触发的子流程。事件子流程可以添加到流程级别或任意子流程级别。用于触发事件子流程的事件是使用开始事件配置的。为此，事件子流程是不支持空开始事件的。事件子流程可以被消息事件，错误事件，信号事件，定时器事件，或补偿事件触发。开始事件的订阅在包含事件子流程的作用域（流程实例或子流程）创建时就会创建。当作用域销毁也会删除订阅。

事件子流程可以是中断的或非中断的。一个中断的子流程会取消当前作用域内的所有流程。非中断事件子流程会创建那一个新的同步分支。中断事件子流程只会被每个激活状态的宿主触发一次，非中断事件子流程可以触发多次。子流程是否是终端的，配置使用事件子流程的开始事件配置。

事件子流程不能有任何进入和外出流程。当事件触发一个事件子流程时，输入顺序流是没有意义的。当事件子流程结束时，无论当前作用域已经结束了（中断事件子流程的情况），或为非中断子流程生成同步分支会结束。

当前的限制：

- activiti只支持中断事件子流程。
- activiti只支持使用错误开始事件或消息开始事件的事件子流程。

8. 6. 2. 2. #图像标记

事件子流程可以显示为边框为虚线的内嵌子流程。



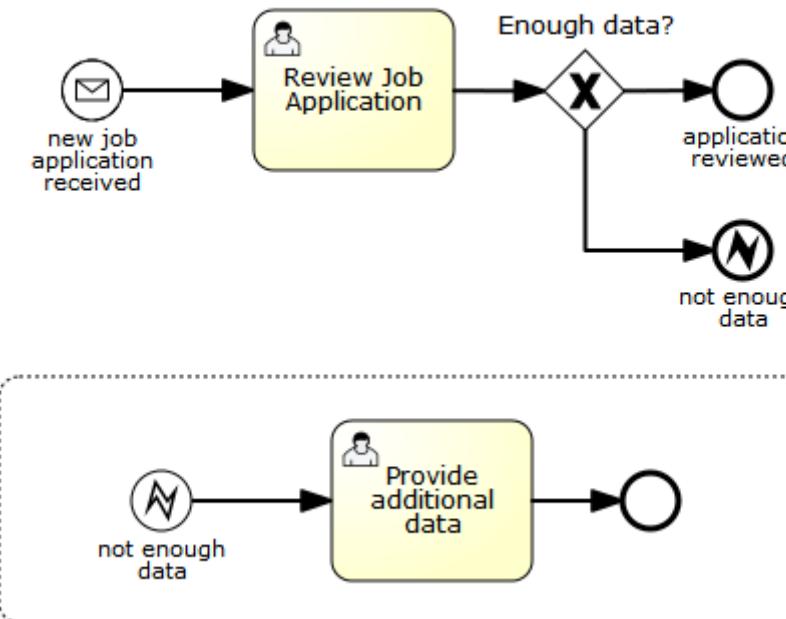
8. 6. 2. 3. #XML内容

事件子流程的XML内容与内嵌子流程是一样的。另外，要把triggeredByEvent属性设置为true：

```
<subProcess id="eventSubProcess" triggeredByEvent="true">
  ...
</subProcess>
```

8. 6. 2. 4. #实例

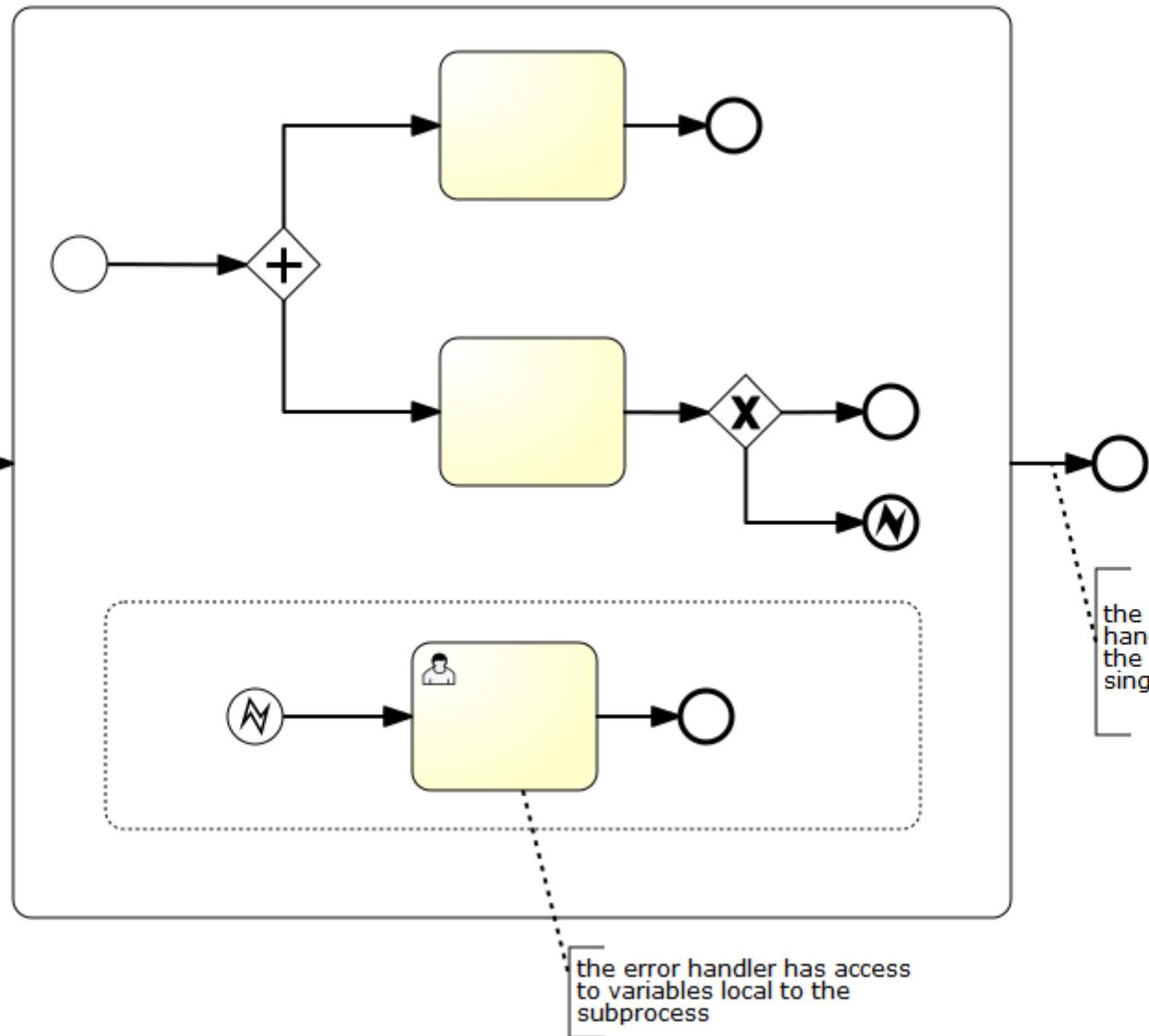
下面是一个使用错误开始事件触发的事件子流程的实例。事件子流程是放在“流程级别”的，意思是，作用于流程实例：



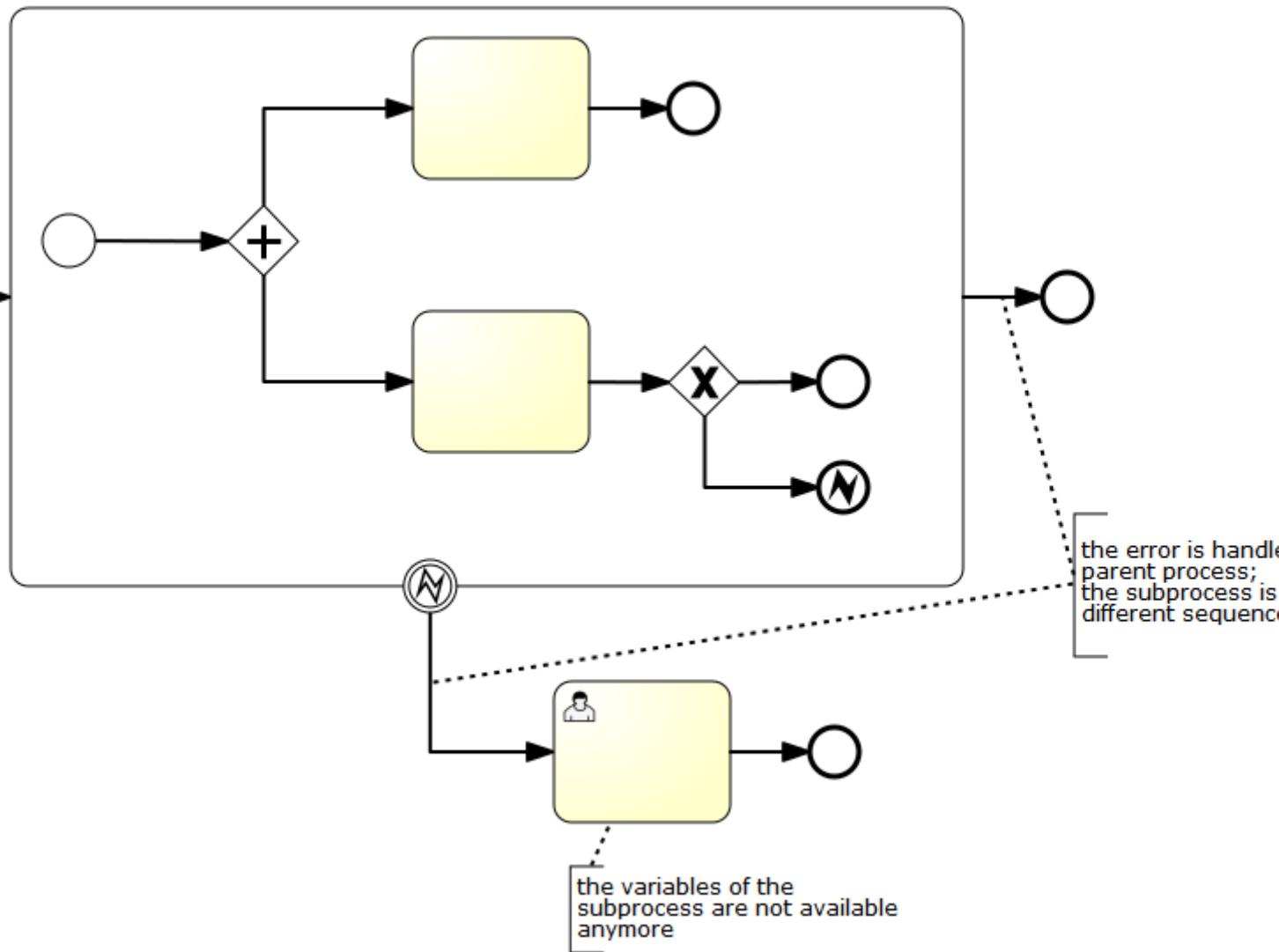
事件子流程的XML如下所示：

```
<subProcess id="eventSubProcess" triggeredByEvent="true">
  <startEvent id="catchError">
    <errorEventDefinition errorRef="error" />
  </startEvent>
  <sequenceFlow id="flow2" sourceRef="catchError" targetRef="taskAfterErrorCatch" />
  <userTask id="taskAfterErrorCatch" name="Provide additional data" />
</subProcess>
```

如上面所述，事件子流程也可以添加成内嵌子流程。如果添加为内嵌子流程，它其实是边界事件的一种替代方案。考虑下面两个流程图。两种情况内嵌子流程会抛出一个错误事件。两种情况错误都会被捕获并使用一个用户任务处理。



相对于：



两种场景都会执行相同任务。然而，两种建模方式是不同的：

- 内嵌子流程是使用与执行作用域宿主相同的流程执行的。意思是内嵌子流程可以访问它作用域内的内部变量。当使用边界事件时，执行内嵌子流程的流程会删除，并生成一个流程根据边界事件的顺序流继续执行。这意味着内嵌子流程创建的变量不再起作用了。
- 当使用事件子流程时，事件是完全由它添加的子流程处理的。当使用边界事件时，事件由父流程处理。

这两个不同点可以帮助我们决定是使用边界事件还是内嵌事件子流程来解决特定的流程建模/实现问题。

8.6.3. #事务子流程

[EXPERIMENTAL]

8.6.3.1. #描述

事务子流程是内嵌子流程，可以用来把多个流程放到一个事务里。事务是一个逻辑单元，可以把一些单独的节点放在一起，这样它们就可以一起成功或一起失败。

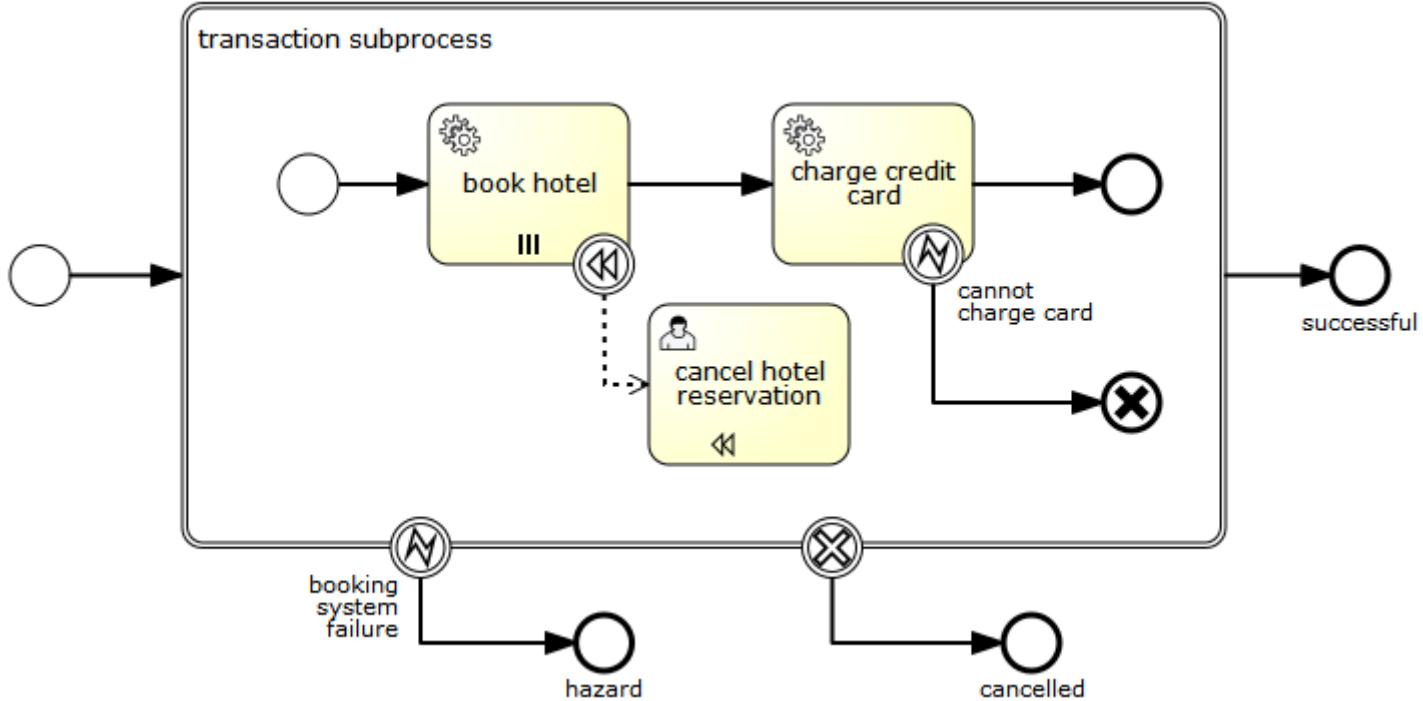
事务可能的结果： 事务可以有三种可能的结果：

- 事务成功，如果没有取消也没有因为问题终结。如果事务子流程是成功的，就会使用外出顺序流继续执行。如果流程后来抛出了一个补偿事件，成功的事务可能被补偿。

注意：和普通内嵌子流程一样，事务可能在成功后， 使用中间补偿事件进行补偿。

- 事务取消，如果流程到达取消结束事件。这时，所有流程都会终结和删除。触发补偿的一个单独的流程，会通过取消边界事件继续执行。在补偿完成之后，事务子流程会使用取消边界事务的外出顺序流向执行。
- 事务被问题结束，如果跑出了一个错误事件，而且没有在事务子流程中捕获。（如果错误被事务子流程的边界事件处理了，也会这样应用。）这时，不会执行补偿。

下面的图形演示了三种不同的结果：



与ACID事务的关系：一定不要把bpmn事务子流程与技术（ACID）事务相混淆。bpmn事务子流程不是技术事务领域的东西。要理解activiti中的事务管理，请参考 并发与事务。bpmn事务和技术事务有以下不同点：

- ACID事务一般是短期的，bpmn事务可能持续几小时，几天，甚至几个月才能完成。（考虑事务中包含的节点可能有用户任务，一般人员响应的时间比应用时间要长。或者，或者，在其他情况下，bpmn事务可能要等待发生一些事务事件，就像要根据某种次序执行。）这种操作通常要相比更新数据库的一条数据，或把一条信息保存到事务性队列中，消耗更长的时间来完成。
- 因为不能在整个业务节点的过程中保持一个技术性的事务，所以bpmn事务一般要跨越多个ACID事务。
- 因为bpmn事务会跨越多个ACID事务，所以会丧失ACID的特性。比如，考虑上述例子。假设“约定旅店”和“刷信用卡”操作在单独的ACID事务中执行。也假设“预定旅店”节点已经成功了。现在我们处于一个中间不稳定状态，因为我们预定了酒店，但是还没有刷信用卡。现在，在一个ACID事务中，我们要依次执行不同的操作，也会有一个中间不稳定状态。不同的是，这个中间状态对事务的外部是可见的。比如，如果通过外部预定服务进行了预定，其他使用相同预定服务的部分就可以看到旅店被预定了。这意味着实现业务事务时，我们完全失去了隔离属性（注：我们也经常放弃隔离性，来为ACID事务获得更高的并发，但是我们可以完全控制，中间不稳定状态也只持续很短的时间）。

- bpmn业务事务也不能使用通常的方式回滚。因为它跨越了多个事务，bpmn事务取消时一些ACID事务可能已经提交了。这时，它们不能被回滚了。

因为bpmn事务实际上运行时间很长，缺乏隔离性和回滚机制都需要被区别对待。实际上，这里也没有更好的办法在特定领域处理这些问题：

- 使用补偿执行回滚。如果事务范围抛出了取消事件，会影响已经执行成功的节点，并使用补偿处理器执行补偿。
- 隔离性的缺乏通常使用特定领域的解决方法来解决。比如，上面的例子中，一个旅店房间可能会展示给第二个客户，在我们确认第一个客户付费之前。虽然这可能与业务预期不符，预定服务可能选择允许一些过度的预约。
- 另外，因为事务会因为风险而中断，预定服务必须处理这种情况，已经预定了旅店，但是一直没有付款的情况。（因为事务被中断了）。这时预定服务需要选择一个策略，在旅店房间预定超过最大允许时间后，如果还没有付款，预定就会取消。

综上所述：ACID处理的是通常问题（回滚，隔离级别和启发式结果），在实现业务事务时，我们需要找到特定领域的解决方案来处理这些问题。

目前的限制：

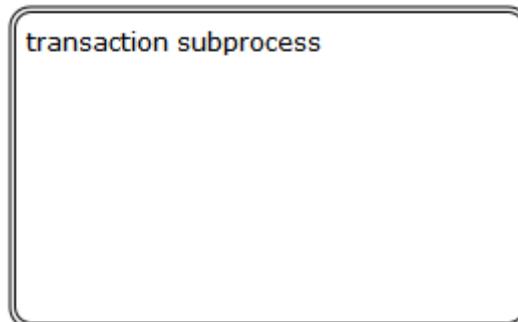
- bpmn规范要求流程引擎能根据底层事务的协议处理事件，比如如果底层协议触发了取消事件，事务就会取消。作为内嵌引擎，activiti目前不支持这项功能。（对此造成的后果，可以参考下面的一致性讨论）。

ACID事务顶层的一致性和优化并发：bpmn事务保证一致性，要么所有节点都成功，或者一些节点成功，对其他成功的节点进行补偿。无论哪种方式，都会有一致性的结果。不过要讨论一些activiti内部的情况，bpmn事务的一致性模型是叠加在流程的一致性模型之上的。

activiti执行流程是事务性的。并发使用了乐观锁。在activiti中，bpmn错误，取消和补偿事件都建立在同样的acid事务与乐观锁之上。比如，取消结束事件只能触发它实际到达的补偿。如果之前服务任务抛出了未声明的异常。或者，补偿处理器的效果无法提交，如果底层的acid事务的参与者把事务设置成必须回滚。或者当两个并发流程到达了取消结束事件，可能会触发两次补偿，并因为乐观锁异常失败。所有这些都说明activiti中实现bpmn事务时，相同的规则也作用于普通的流程和子流程。所以为了保证一致性，重要的是使用一种方式考虑实现乐观事务性的执行模型。

8. 6. 3. 2. #图形标记

事务子流程显示为内嵌子流程，使用双线边框。



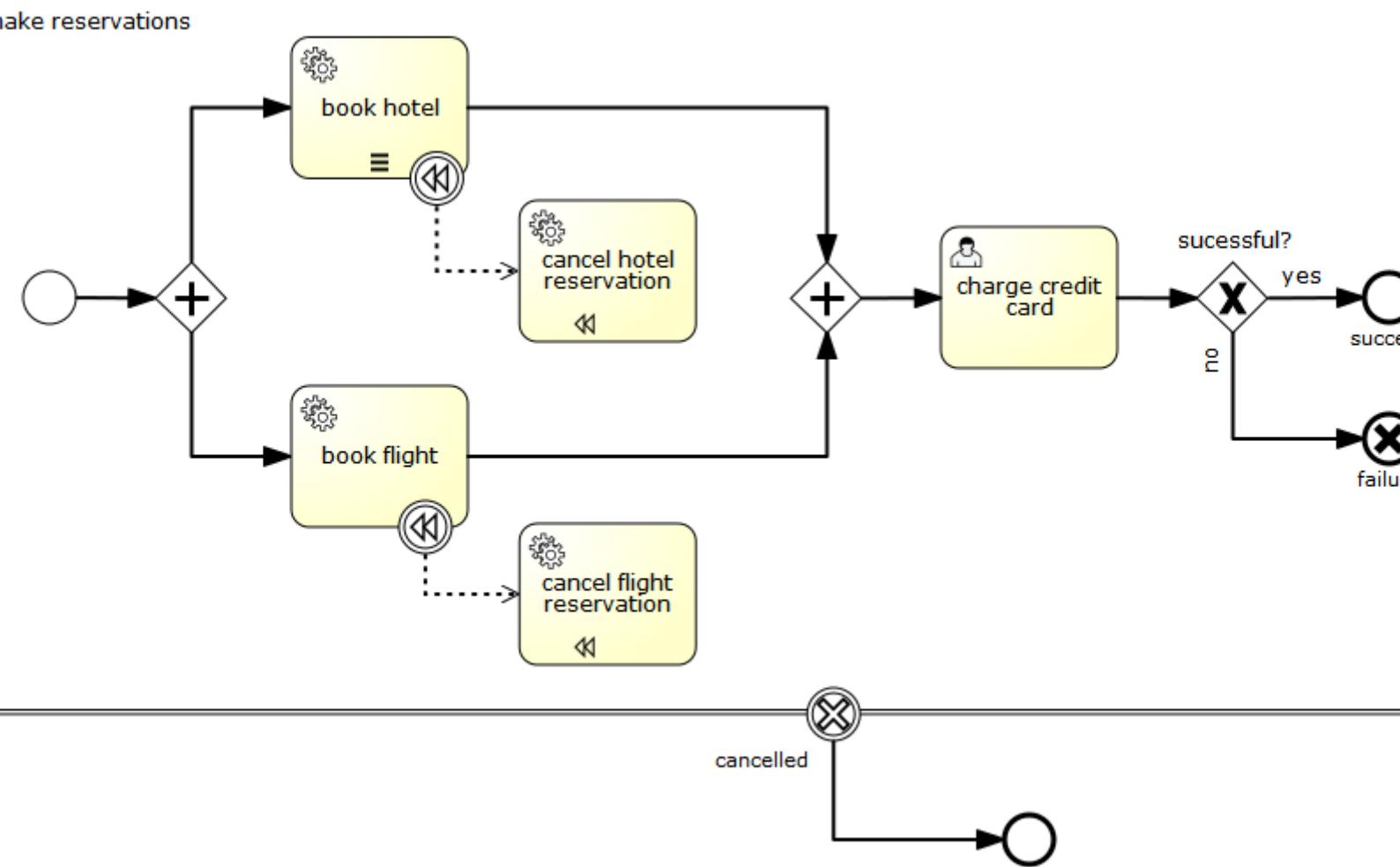
8. 6. 3. 3. #XML内容

事务子流程使用transaction标签：

```
<transaction id="myTransaction" >
...
</transaction>
```

8. 6. 3. 4. #实例

下面是事务子流程的实例：



8. 6. 4. #调用活动（子流程）

8. 6. 4. 1. #描述

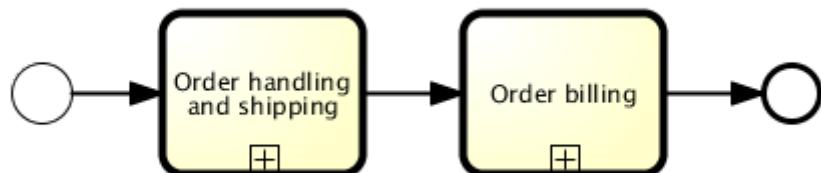
bpmn 2.0区分了普通子流程，也叫做内嵌子流程，和调用节点，看起来很相似。上概念上讲，当流程抵达及诶单时，两者都会调用子流程。

不同点是调用节点引用流程定义外部的一个流程，子流程会内嵌到原始的流程定义中。使用调用节点的主要场景是需要重用流程定义，这个流程定义需要被很多其他流程定义调用的时候。

当流程执行到调用节点，会创建一个新分支，它是到达调用节点的流程的分支。这个分支会用来执行子流程，默认创建并行子流程，就像一个普通的流程。上级流程会等待子流程完成，然后才会继续向下执行。

8. 6. 4. 2. #图形标记

调用节点显示与子流程相同，不过是粗边框（无论是折叠和展开的）。根据不同的建模工具，调用节点也可以展开，但是显示为折叠的子流程。



8. 6. 4. 3. #XML内容

A call activity is a regular activity, that requires a calledElement that references a process definition by its key. In practice, this means that the id of the process is used in the calledElement.

```
<callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess" />
```

注意，子流程的流程定义是在执行阶段解析的。就是说子流程可以与调用的流程分开部署，如果需要的话。

8. 6. 4. 4. #传递变量

可以把流程变量传递给子流程，反之亦然。数据会复制给子流程，当它启动的时候，并在它结束的时候复制回主流程。

```

<callActivity id="callSubProcess" calledElement="checkCreditProcess" >
  <extensionElements>
    <activiti:in source="someVariableInMainProcess" target="nameOfVariableInSubProcess" />
    <activiti:out source="someVariableInSubProcess" target="nameOfVariableInMainProcess" />
  </extensionElements>
</callActivity>
  
```

我们使用activiti扩展来简化BPMN标准元素调用dataInputAssociation和dataOutputAssociation，这只能在你使用BPMN 2.0标准方式声明流程变量才管用。

这里也可以使用表达式：

```

<callActivity id="callSubProcess" calledElement="checkCreditProcess" >
  <extensionElements>
    <activiti:in sourceExpression="${x+5}" target="y" />
    <activiti:out source="${y+5}" target="z" />
  </extensionElements>
</callActivity>
  
```

最后 $z = y + 5 = x + 5 + 5$

8. 6. 4. 5. #实例

下面的流程图演示了简单订单处理。先判断客户端信用，这可能与很多其他流程相同。检查信用阶段这里设计成调用节点。



流程看起来像下面这样：

```

<startEvent id="theStart" />
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="receiveOrder" />

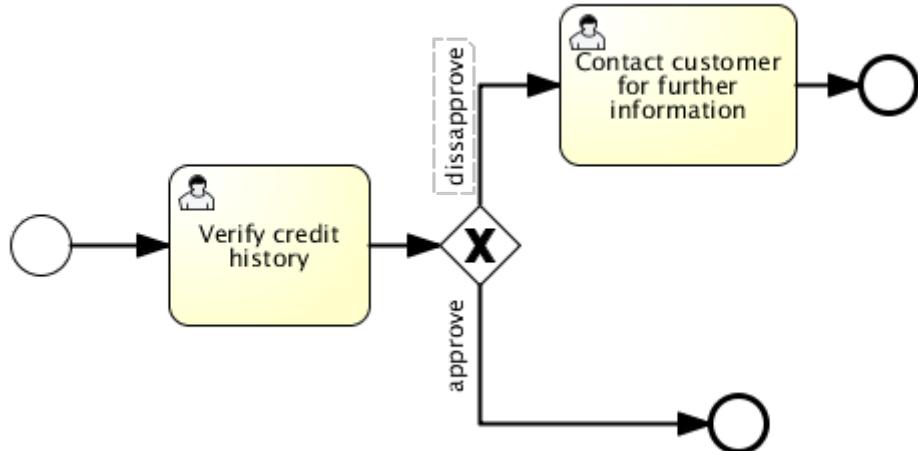
<manualTask id="receiveOrder" name="Receive Order" />
<sequenceFlow id="flow2" sourceRef="receiveOrder" targetRef="callCheckCreditProcess" />

<callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess" />
<sequenceFlow id="flow3" sourceRef="callCheckCreditProcess" targetRef="prepareAndShipTask" />

<userTask id="prepareAndShipTask" name="Prepare and Ship" />
<sequenceFlow id="flow4" sourceRef="prepareAndShipTask" targetRef="end" />

<endEvent id="end" />
  
```

子流程看起来像下面这样：



子流程的流程定义没有什么特别的。 它也可以单独使用，不用其他流程调用。

8. 7. #事务和并发

8. 7. 1. #异步操作

activiti通过事务方式执行流程，可以根据你的需求定制。现在开始看一下Activiti通常是如何处理事务的。如果触发了activiti的操作（比如，开始流程，完成任务，触发流程继续执行），activiti会推进流程，直到每个分支都进入等待状态。更抽象的说，它会流程图执行深度优先搜索，如果每个分支都遇到等待状态，就会返回。等待状态是“稍后”需要执行任务，就是说activiti会把当前状态保存到数据库中，然后等待下一次触发。触发可能来自外部，比如用户任务或接收到一个消息，也可能来自activiti本身，比如我们设置了定时器事件。下面图片展示了这种操作：

我们可以看到包含用户任务，服务任务和定时器事件的流程。完成用户任务，和校验地址是在同一个工作单元中，所以它们的成功和失败是原子性的。意味着如果服务任务抛出异常，我们要回滚当前事务，这样流程会退回到用户任务，用户任务就依然在数据库里。这就是activiti默认的行为。在（1）中应用或客户端线程完成任务。这会执行服务，流程推进，直到遇到一个等待状态，这里就是定时器（2）。然后它会返回给调用者（3），并提交事务（如果事务是由activiti开启的）。

有的时候，这不是我们想要的。有时我们需要自己控制流程中事务的边界，这样就能把业务逻辑包裹在一起。这就需要使用异步执行了。参考下面的流程（判断）：

这次我们完成了用户任务，生成一个发票，把发票发送给客户。这次生成发票不在同一个工作单元内了，所以我们不想对用户任务进行回滚，如果生成发票出错了。所以，我们想让activiti实现的是完成用户任务（1），提交事务，返回给调用者应用。然后在后台的线程中，异步执行生成发票。后台线程就是activiti的job执行器（其实是一个线程池）周期对数据库的job进行扫描。所以后面的场景，当我们到达“generate invoice”任务，我们为activiti创建一个稍后执行的job“消息”，并把它保存到数据库。job会被job执行器获取并执行。我们也会给本地job执行器一个提醒，告诉它有一个新job，来增加性能。

要想使用这个特性，我们要使用activiti:async="true"扩展。例子中，服务任务看起来就是这样：

```
<serviceTask id="service1" name="Generate Invoice" activiti:class="my.custom.Delegate" activiti:async="true" />
```

activiti:async可以使用到如下bpmn任务类型中：task，serviceTask，scriptTask，businessRuleTask，sendTask，receiveTask，userTask，subProcess，callActivity

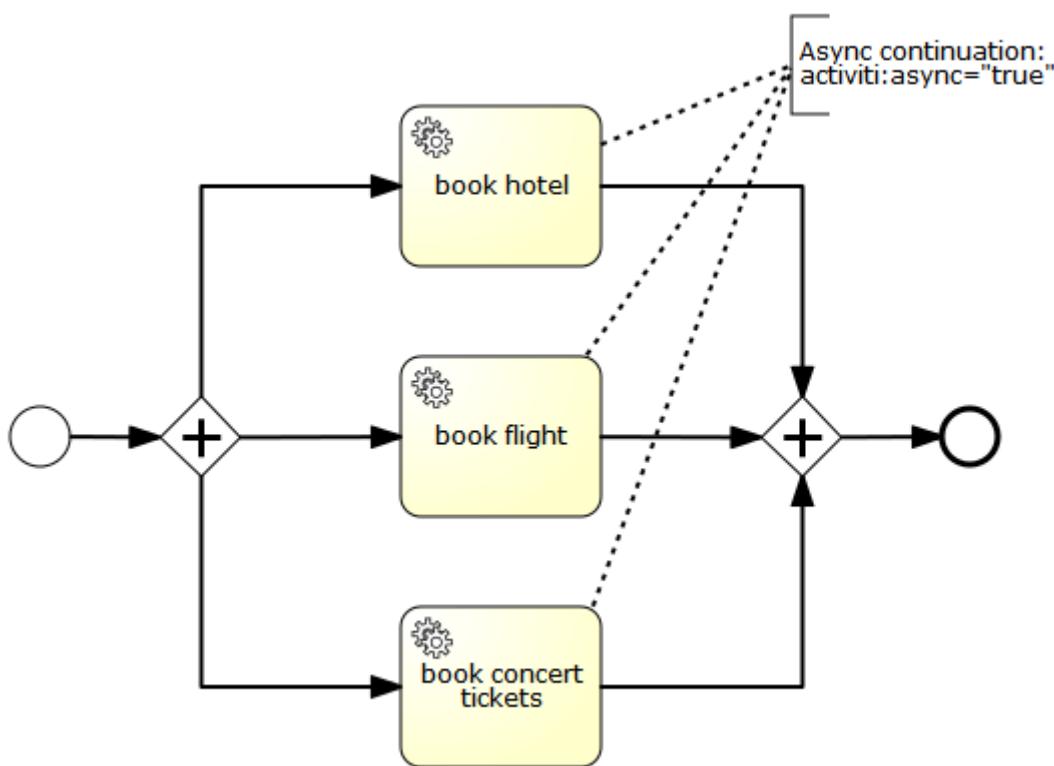
对于userTask，receiveTask和其他等待装填，异步执行的作用是让开始流程监听器运行在一个单独的线程/事务中。

8.7.2. #排他任务

从activiti 5.9开始，JobExecutor能保证同一个流程实例中的job不会并发执行。为啥呢？

8.7.2.1. #为什么要使用排他任务？

参考如下流程定义：



我们有一个并行网关，后面有三个服务任务，它们都设置为异步执行。这样会添加三个job到数据库里。一旦job进入数据库，它就可以被JobExecutor执行了。JobExecutor会获取job，把它们代理到工作线程的线程池中，会在那里真正执行job。就是说，使用异步执行，你可以吧任务分配给这个线程池（在集群环境，可能会使用多个线程池）。这通常是个好事情。然而它也会产生问题：一致性。考虑一下服务任务后的汇聚。当服务任务完成后，我们到达并发汇聚节点，需要决定是等待其他分支，还是继续向下执行。就是说，对每个到达并行汇聚的分支，我们都需要判断是继续还是等待其他分支的一个或多个分支。

为什么这就是问题了呢？因为服务任务配置成使用异步执行，可能相关的job都在同一时间被获取，被JobExecutor分配给不同的工作线程执行。结果是三个单独的服务执行使用的事物在到达并发汇聚时可能重叠。如果出现了这个问题，这些事物是互相不可见的，其他事物同时到达了相同的并发汇聚，假设它们都在等待其他分支。然而，每个事物都假设它们在等待其他分支，所以没有分支会越过并发汇聚继续执行，流程实例会一直在等待状态，无法继续执行。

activiti是如何解决这个问题的？activiti使用了乐观锁。当我们基于判断的数据看起来不是最新的时（因为其他事务可能在我们提交之前进行了修改，我们会在每个事务里增加数据库同一行的版本）。这时，第一个提交的事务会成功，其他会因为乐观锁异常导致失败。这就解决了我们上面讨论的流程的问题：如果多个分支同步到达并行汇聚，它们会假设它们都在登录，并增加它们父流程的版本号（流程实例）然后尝试提交。第一个分支会成功提交，其他分支会因为乐观锁导致失败。因为流程是被job触发的，activiti会尝试在等待一段时间后尝试执行同一个job，想这段时间可以同步网关的状态。

这是一个很好的解决方案吗？像我们看到的一样，乐观锁允许activiti避免非一致性。它确定我们不会“堵在汇聚网关”，意思是：或者所有分支都通过网关，或者数据库中的job正在尝试通过。然而，虽然这是一个对于持久性和一致性的完美解决方案，但对于上层来说不一定是期望的行为：

- activiti只会对同一个job重试估计次数（默认配置为3）。之后，job还会在数据库里，但是不会再重试了。意味着这个操作必须手工执行job的触发。

- 如果job有非事务方面的效果，它不会因为失败的事务回滚。比如，如果“预定演唱会门票”服务没有与activiti共享事务，重试job可能导致我们预定了过多门票。

在activiti中，我们推荐了新的概念，并已经在jbpm4中实现了，叫做“排他job”。

8.7.2.2. #什么是排他job?

对于一个流程实例，排他任务不能同时执行两个。考虑上面的流程：如果我们把服务任务申请为排他任务，JobExecutor会保证对应的job不会并发执行。相反，它会保证无论什么时候获取一个流程实例的排他任务，都会把同一个流程实例的其他任务都取出来，放在同一个工作线程中执行。它保证job是顺序执行的。

如何启用这个特性？从activiti 5.9开始，排他任务已经是默认配置了。所以异步执行和定时器事件默认都是排他任务。另外，如果你想把job设置为非排他，可以使用activiti:exclusive="false"进行配置。比如，下面的服务任务就是异步但是非排他的。

```
<serviceTask id="service" activiti:expression="${myService.performBooking(hotel, dates)}" activiti:async="true"
```

这是一个好方案吗？有一些人认为我们这是否是一个好方案。他们的结论会帮你在并发和性能问题方面节省时间。这个问题上需要考虑两件事情：

- 如果你是专家并且知道自己在做什么时（理解“为什么排他任务”这章的内容），也可以关闭这个功能，否则，对于大多数使用异步执行和定时器的用户来说，这个功能是没问题的。
- 它也没有性能问题，在高负载的情况下性能是个问题。高负载意味着JobExecutor的所有工作线程都一直在忙碌着。使用排他任务，activiti可以简单的分布不同的负载。排他任务意味着同一个流程实例的异步执行会由相同的线程顺序执行。但是要考虑：如果你有多个流程实例时。所有其他流程实例的job也会分配给其他线程同步执行。意味着虽然activiti不会同时执行一个流程实例的排他job，但是还会同步执行多个流程实例的一步执行。通过一个总体的预测，在大多数场景下，它都会让单独的实例运行的更迅速。而且，对于同一流程实例中的job，需要用到的数据也会利用执行的集群节点的缓存。如果任务没有在同一个节点执行，数据就必须每次从数据库重新读取了。

8.8. #流程实例授权

默认所有人在部署的流程定义上启动一个新流程实例。通过流程初始化授权功能定义的用户和组，web客户端可以限制哪些用户可以启动一个新流程实例。注意：activiti引擎不会校验授权定义。这个功能只是为减轻web客户端开发者实现校验规则的难度。设置方法与用户任务用户分配类似。用户或组可以使用<activiti:potentialStarter>标签分配为流程的默认启动者。下面是一个例子：

```
<process id="potentialStarter">
  <extensionElements>
    <activiti:potentialStarter>
      <resourceAssignmentExpression>
        <formalExpression>group2, group(group3), user(user3)</formalExpression>
      </resourceAssignmentExpression>
    </activiti:potentialStarter>
  </extensionElements>
  <startEvent id="theStart"/>
  ...

```

上面的XML中，user(user3)是直接引用了用户user3，group(group3)是引用了组group3。如果没显示设置，默认认为是群组。
也可以使用<process>标签的属性，<activiti:candidateStarterUsers>和<activiti:candidateStarterGroups>。下面是一个例子：

```
<process id="potentialStarter" activiti:candidateStarterUsers="user1, user2"
         activiti:candidateStarterGroups="group1">
    ...

```

可以同时使用这两个属性。

定义流程初始化授权后，开发者可以使用如下方法获得授权定义。这些代码可以获得给定的用户可以启动哪些流程定义：

```
processDefinitions = repositoryService.createProcessDefinitionQuery().startableByUser("userxxx").list();
```

也可以获得指定流程定义设置的潜在启动者对应的IdentityLink。

```
identityLinks = repositoryService.getIdentityLinksForProcessDefinition("processDefinitionId");
```

下面例子演示了如何获得可以启动给定流程的用户列表：

```
List<User> authorizedUsers = identityService().createUserQuery().potentialStarter("processDefinitionId")
```

相同的方式，获得可以启动给定流程配置的群组：

```
List<Group> authorizedGroups = identityService().createGroupQuery().potentialStarter("processDefinitionId")
```

8.9. #数据对象

[试验功能]

BPMN提供了一种功能，可以在流程定义或子流程中定义数据对象。根据BPMN规范，流程定义可以包含复杂XML结构，可以导入XSD定义。对于Activiti来说，作为Activiti首次支持的数据对象，可以支持如下的XSD类型：

- ```
<dataObject id="dObj1" name="StringTest" itemSubjectRef="xsd:string"/>
```

- ```
<dataObject id="dObj2" name="BooleanTest" itemSubjectRef="xsd:boolean"/>
```

- ```
<dataObject id="dObj3" name="DateTest" itemSubjectRef="xsd:datetime"/>
```
- ```
<dataObject id="dObj4" name="DoubleTest" itemSubjectRef="xsd:double"/>
```
- ```
<dataObject id="dObj5" name="IntegerTest" itemSubjectRef="xsd:int"/>
```
- ```
<dataObject id="dObj6" name="LongTest" itemSubjectRef="xsd:long"/>
```

数据对象定义会自动转换为流程变量，名称与' name' 属性对应。除了数据对象的定义之外，activiti也支持使用扩展元素来为这个变量赋予默认值。下面的BPMN片段就是对应的例子：

```
<process id="dataObjectScope" name="Data Object Scope" isExecutable="true">
  <dataObject id="dObj123" name="StringTest123" itemSubjectRef="xsd:string">
    <extensionElements>
      <activiti:value>Testing123</activiti:value>
    </extensionElements>
  </dataObject>
```

第#9#章#表单

Activiti提供了一种方便而且灵活的方式在业务流程中以手工方式添加表单。我们对表单的支持有2种方式：通过表单属性对内置表单进行渲染和外置表单进行渲染。

9. 1. #表单属性

业务流程相关联的所有信息要么是包含自身的流程变量，要么是通过流程变量的引用。Activiti支持存储复杂的Java对象作为流程变量，如 序列化对象， Jpa实体对象或者整个XML文档作为字符串。

用户是在启动一个流程和完成用户任务时，与流程进行交互的。表单需要某个UI技术渲染之后才能够与用户进行交互。为了能够使用不同UI技术变得容易，流程定义包含一个对流程变量中复杂的Java类型对象到一个' properties' 的Map<String, String>类型的转换逻辑。

使用Activiti API的方法查看公开的属性信息。然后，任意UI技术都能够在这些属性上面构建一个表单。该属性专门（并且更多局限性）为流程变量提供了一个视图。表单所需要显示的属性可以从下面例子中的返回值FormData中获取。

```
StartFormData FormService.getStartFormData(String processDefinitionId)
```

or

```
TaskFormData FormService.getTaskFormData(String taskId)
```

在默认情况下，内置的表单引擎，' sees' 这些变量就像对待流程变量一样。如果任务表单属性和流程变量是一对一的关系，那么任务表单属性就不需要进行申明了，例如，下面的申明：

```
<startEvent id="start" />
```

当执行到开始事件时，所有的流程变量都是可用的，但

```
formService.getStartFormData(String processDefinitionId).getFormProperties()
```

会是一个空值，因为没有定义具体的映射。

在上面的实例中，所有被提交的属性都将会作为流程变量被存储在Activiti使用的数据库中。这意味着在一个表单中新添加一个简单的input输入字段，也会作为一个新的变量被存储。

属性来自于流程变量，但不一定非要作为流程变量存储起来，例如，一个流程变量可能是JPA实体如类Address。在某种UI技术中使用的表单属性StreetName可能会关联到一个表达式#{address.street}。

类似的，用户提交的表单属性应该作为流程变量进行存储或者使用UEL值表达式将其作为流程变量的一个嵌套属性进行存储，例如#{address.street}。

同样的，提交的表单属性默认的行为是作为流程变量进行存储，除非一个 formProperty 申明了其他的规则。

类型转换同样也可以应用于表单数据和流程变量之间处理的一部分。

例如：

```
<userTask id="task">
  <extensionElements>
    <activiti:formProperty id="room" />
    <activiti:formProperty id="duration" type="long"/>
    <activiti:formProperty id="speaker" variable="SpeakerName" writable="false" />
    <activiti:formProperty id="street" expression="#{address.street}" required="true" />
  </extensionElements>
</userTask>
```

- 表单属性 room 将会被映射为String类型流程变量 room。
- 表单属性 duration 将会被映射为 java.lang.Long类型流程变量 duration。
- 表单属性 speaker 将会被映射为流程变量 SpeakerName。它只能够在TaskFormData对象中使用。如果 属性speaker提交，将会抛出一个ActivitiException的异常。类似的，将其属性设置为readable="false"，该属性就会在FormData进行排除，但是在提交后仍然会对其进行处理。
- 表单属性street将会映射为Java Bean address的属性street 作为String类型的流程变量。当提交的表单属性并没有提供并且 required="true" 时，那么就会抛出一个异常。

表单数据也可以作为FormData的一部分提供类型元数据。该FormData可以从StartFormData
FormService.getStartFormData(String processDefinitionId) 和 TaskFormData
FormService.getTaskFormData(String taskId) 方法的返回值中获取。

我们支持以下的几种表单属性类型：

- string (org.activiti.engine.impl.form.StringFormType)
- long (org.activiti.engine.impl.form.LongFormType)
- enum (org.activiti.engine.impl.form.EnumFormType)
- date (org.activiti.engine.impl.form.DateFormType)
- boolean (org.activiti.engine.impl.form.BooleanFormType)

对于申明每一个表单属性，以下的FormProperty信息可以通过List<FormProperty>
formService.getStartFormData(String processDefinitionId).getFormProperties() 和 List<FormProperty>
formService.getTaskFormData(String taskId).getFormProperties() 获取。

```
public interface FormProperty {
  /**
   * the key used to submit the property in {@link FormService#submitStartFormData(String, java.util.Map)}
   * or {@link FormService#submitTaskFormData(String, java.util.Map)} */
  String getId();
  /** the display label */
  String getName();
  /** one of the types defined in this interface like e.g. {@link #TYPE_STRING} */
  FormType getType();
  /** optional value that should be used to display in this property */
  String getValue();
```

```

/** is this property read to be displayed in the form and made accessible with the methods
 * {@link FormService#getStartFormData(String)} and {@link FormService#getTaskFormData(String)}. */
boolean isReadable();
/** is this property expected when a user submits the form? */
boolean isWritable();
/** is this property a required input field */
booleanisRequired();
}

```

例子：

```

<startEvent id="start">
<extensionElements>
<activiti:formProperty id="speaker"
    name="Speaker"
    variable="SpeakerName"
    type="string" />

<activiti:formProperty id="start"
    type="date"
    datePattern="dd-MMM-yyyy" />

<activiti:formProperty id="direction" type="enum">
    <activiti:value id="left" name="Go Left" />
    <activiti:value id="right" name="Go Right" />
    <activiti:value id="up" name="Go Up" />
    <activiti:value id="down" name="Go Down" />
</activiti:formProperty>

</extensionElements>
</startEvent>

```

所有的表单属性的信息都是可以通过API进行访问的。可以通过

`formProperty.getType().getName()`获取类型的名称。

甚至可以通过

`formProperty.getType().getInformation("datePattern")`获取日期的匹配方式。通过

`formProperty.getType().getInformation("values")`可以获取到枚举值。

Activiti控制台支持表单属性并且可以根据表单定义对表单进行渲染。例如下面的XML片段

```

<startEvent ... >
<extensionElements>
    <activiti:formProperty id="numberOfDays" name="Number of days" value="${numberOfDays}" type="long" required="true" />
    <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyy)" value="${startDate}" datePattern="dd-MM-yy" />
    <activiti:formProperty id="vacationMotivation" name="Motivation" value="${vacationMotivation}" type="string" />
</extensionElements>
</userTask>

```

当使用 Activiti控制台时，它将会被渲染成流程的启动表单。

The screenshot shows a web-based form titled "Vacation request". At the top right, it says "Version 1" and "Deployed one hour ago". The form has three input fields: "Number of days*" (with a red asterisk), "First day of holiday (dd-MM-yyyy)*" (with a red asterisk and a calendar icon), and "Motivation". Below the form are two buttons: "Start process" and "Cancel".

9.2. #外置表单的渲染

该API同样也允许你执行Activiti流程引擎之外的方式渲染你自己的任务表单。这些步骤说明你可以用你自己的方式对任务表单进行渲染。

本质上，所有需要渲染的表单属性都是通过2个服务方法中的一个进行装配的： StartFormData
FormService.getStartFormData(String processDefinitionId) 和 TaskFormdata
FormService.getTaskFormData(String taskId).

表单属性可以通过 ProcessInstance FormService.submitStartFormData(String processDefinitionId, Map<String, String> properties) and void FormService.submitStartFormData(String taskId, Map<String, String> properties) 2种方式进行提交。

想要了解更多表单属性如何映射为流程变量，可以查看 第 9.1 节 “表单属性”

你可以将任何表单模版资源放进你要部署的业务文档之中（如果你想要将它们按照流程的版本进行存储）。它将会在部署中作为一种可用的资源。你可以使用String
ProcessDefinition.getDeploymentId() 和 InputStream RepositoryService.getResourceAsStream(String deploymentId, String resourceName); 获取部署上去的表单模版。这样就可以获取你的表单模版定义文件，那么你就可以在你自己的应用中渲染/显示你的表单。

你也可以使用该功能获取任务表单之外的其他的部署资源用于其他的目的。

属性<userTask activiti:formKey="..."通过API String FormService.getStartFormData(String processDefinitionId).getFormKey() 和 String FormService.getTaskFormData(String taskId).getFormKey() 暴露出来的。你可以使用这个存储你部署的模版中的全名（例如org/activiti/example/form/my-custom-form.xml），但是这并不是必须的。例如，你可以在表单属性中存储一个通用的key，然后运用一种算法或者换转去得到你实际使用的模版。当你想要通过不同UI技术渲染不能的表单，这可能更加方便，例如，使用正常屏幕大小的web应用程序的表单， 移动手机小屏幕的表单和甚至可能是IM表单和email表单模版。

第#10#章#JPA

你可以使用JPA实体作为流程变量，并且可以这样做：

- 基于流程变量更新已有的JPA实体，它可以在用户任务的表单中填写或者由服务任务生成。
- 重用已有的领域模型不需要编写显示的服务获取实体或者更新实体的值。
- 根据已有实体的属性做出判断（网关即分支聚合）。
- ...

10.1. #要求

只支持符合以下要求的实体：

- 实体应该使用JPA注解进行配置，我们支持字段和属性访问两种方式。@MappedSuperclass 也能够被使用。
- 实体中应该有一个使用@Id注解的主键，不支持复合主键（@EmbeddedId 和 @IdClass）。Id 字段/属性能够使用JPA规范支持的任意类型：原生态数据类型和他们的包装类型（boolean 除外），String, BigInteger, BigDecimal, java.util.Date 和 java.sql.Date.

10.2. #配置

为了能够使用JPA的实体，引擎必须有一个对EntityManagerFactory的引用。这可以通过配置引用或者提供一个持久化单元名称。作为变量的JPA实体将会被自动检测并进行相应的处理

下面例子中的配置是使用jpaPersistenceUnitName：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">

    <!-- 数据库的配置 -->
    <property name="databaseSchemaUpdate" value="true" />
    <property name="jdbcUrl" value="jdbc:h2:mem:JpaVariableTest;DB_CLOSE_DELAY=1000" />

    <property name="jpaPersistenceUnitName" value="activiti-jpa-pu" />
    <property name="jpaHandleTransaction" value="true" />
    <property name="jpaCloseEntityManager" value="true" />

    <!-- job executor configurations -->
    <property name="jobExecutorActivate" value="false" />

    <!-- mail server configurations -->
    <property name="mailServerPort" value="5025" />
</bean>
```

接下来例子中的配置提供了一个我们自定义的EntityManagerFactory（在这个例子中，使用了OpenJPA 实体管理器）。注意该代码片段仅仅包含与例子相关的beans，去掉了其他beans。OpenJPA实体管理的完整并可以使用的例子可以在activiti-spring-examples(/activiti-spring/src/test/java/org/activiti/spring/test/jpa/JPASpringTest.java)中找到。

```

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitManager" ref="pum"/>
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter">
            <property name="databasePlatform" value="org.apache.openjpa.jdbc.sql.H2Dictionary" />
        </bean>
    </property>
</bean>

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    <property name="dataSource" ref="dataSource" />
    <property name="transactionManager" ref="transactionManager" />
    <property name="databaseSchemaUpdate" value="true" />
    <property name="jpaEntityManagerFactory" ref="entityManagerFactory" />
    <property name="jpaHandleTransaction" value="true" />
    <property name="jpaCloseEntityManager" value="true" />
    <property name="jobExecutorActivate" value="false" />
</bean>

```

同样的配置也可以在编程式创建一个引擎时完成，例如：

```

ProcessEngine processEngine = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResourceDefault()
    .setJpaPersistenceUnitName("activiti-pu")
    .buildProcessEngine();

```

配置属性：

- jpaPersistenceUnitName: 使用持久化单元的名称（要确保该持久化单元在类路径下是可用的）。根据该规范，默认的路径是/META-INF/persistence.xml）。要么使用 jpaEntityManagerFactory 或者 jpaPersistenceUnitName。
- jpaEntityManagerFactory: 一个实现了 javax.persistence.EntityManagerFactory 的 bean 的引用。它将被用来加载实体并且刷新更新。要么使用 jpaEntityManagerFactory 或者 jpaPersistenceUnitName。
- jpaHandleTransaction: 在被使用的 EntityManager 实例上，该标记表示流程引擎是否需要开始和提交/回滚事物。当使用 Java 事物 API (JTA) 时，设置为 false。
- jpaCloseEntityManager: 该标记表示流程引擎是否应该关闭从 EntityManagerFactory 获取的 EntityManager 的实例。当 EntityManager 是由容器管理的时候需要设置为 false (例如 当使用并不是单一事物作用域的扩展持久化上下文的时候)。

10.3. #用法

10.3.1. #简单例子

使用 JPA 变量的例子可以在 JPAVariableTest [http://svn.codehaus.org/activiti/activiti/trunk/modules/activiti-engine/src/test/java/org/activiti/standalone/jpa/JPAVariableTest.java] 中找到。我们将会一步一步的解释 JPAVariableTest.testUpdateJPAEntityValues。

首先，我们需要创建一个基于 META-INF/persistence.xml 的 EntityManagerFactory 作为我们的持久化单元。它包含持久化单元中所有的类和一些供应商特定的配置。

我们将使用一个简单的实体作为测试，其中包含有一个id和String类型的value属性，这也将被持久化。在允许测试之前，我们创建一个实体并且保存它。

```
@Entity(name = "JPA_ENTITY_FIELD")
public class FieldAccessJPAAEntity {

    @Id
    @Column(name = "ID_")
    private Long id;

    private String value;

    public FieldAccessJPAAEntity() {
        // Empty constructor needed for JPA
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

我们启动一个新的流程实例，添加一个实体作为变量。至于其他的变量，它们将会被存储在流程引擎的持久化数据库中。下一次获取该变量的时候，它将会根据该类和存储Id从EntityManager中加载。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("entityToUpdate", entityToUpdate);

ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("UpdateJPAValuesProcess", variables)
```

在我们的流程定义中的第一个节点是一个服务任务，它将会调用entityToUpdate上的setValue方法，它其实就是我们之前在启动流程实例时候设置的JPA变量并且它将会从当前流程引擎的上下文关联的EntityManager中加载。

```
<serviceTask id='theTask' name='updateJPAAEntityTask' activiti:expression="${entityToUpdate.setValue('updatedValue')}">
```

当完成服务任务时，流程实例将会停留在流程定义中定义的用户任务环节上。这时我们就可以查看该流程实例。与此同时，EntityManager已经被刷新了并且改变的实体已经被保存进数据

库中。当我们获取entityToUpdate的变量value时，该实体将会被再次加载并且我们获取该实体属性的值将会是updatedValue。

```
// Servicetask in process 'UpdateJPValuesProcess' should have set value on entityToUpdate.
Object updatedEntity = runtimeService.getVariable(processInstance.getId(), "entityToUpdate");
assertTrue(updatedEntity instanceof FieldAccessJPAAEntity);
assertEquals("updatedValue", ((FieldAccessJPAAEntity)updatedEntity).getValue());
```

10.3.2. #查询JPA流程变量

你可以查询某一JPA实体作为变量的ProcessInstances 和 Executions 。注意，在ProcessInstanceQuery 和 ExecutionQuery查询中仅仅variableValueEquals(name, entity) 支持JPA实体变量。方法 variableValueNotEquals, variableValueGreaterThanOrEqual, variableValueLessThan 和 variableValueLessThanOrEqual并不被支持并且传递JPA实体值的时候会抛出一个ActivitiException。

```
ProcessInstance result = runtimeService.createProcessInstanceQuery().variableValueEquals("entityToQuery", enti
```

10.3.3. #使用Spring beans和JPA结合的高级例子

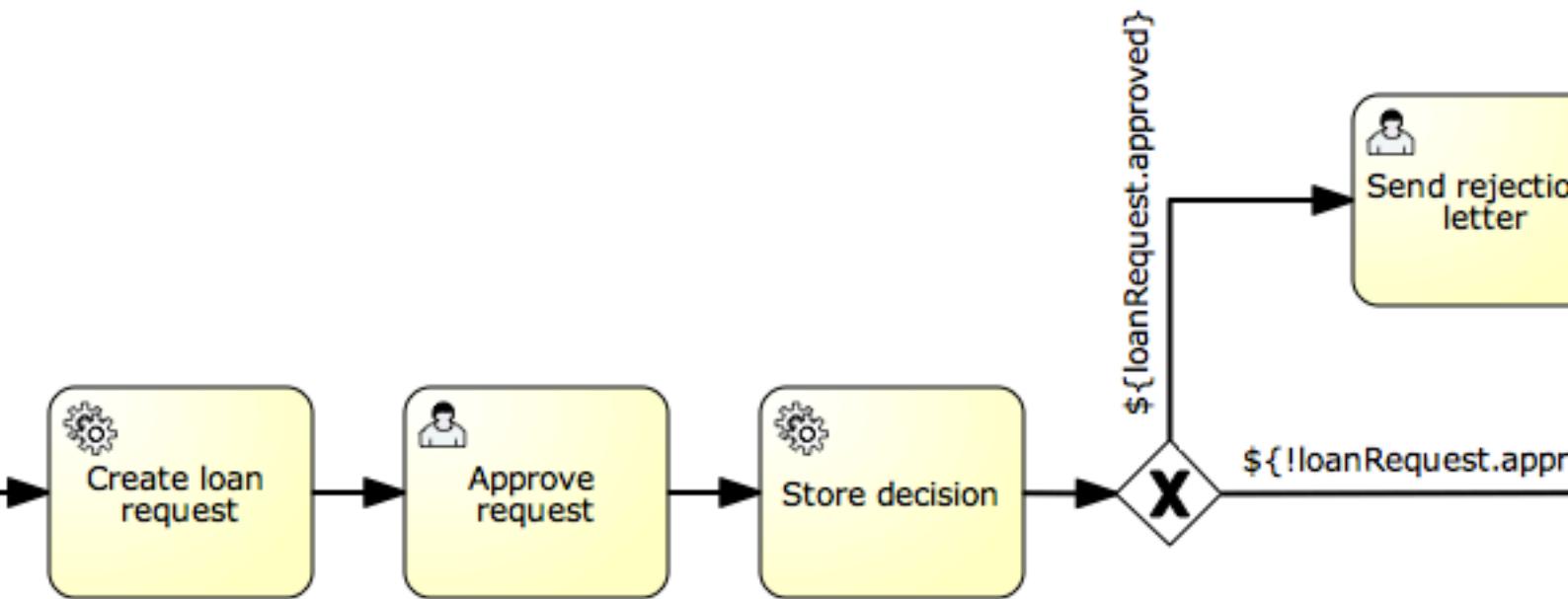
一个更加高级的例子，JPASpringTest，可以在activiti-spring-examples中找到。它描述了如下简单的使用情况：

- 已经存在了一个使用JPA实体的Spring-bean，它用来存储贷款申请。
- 使用Activiti，我们可以通过已经存在的bean获取已经使用的实体，并使用它作为变量用于我们的流程中。

按照下面的步骤定义流程：

- 服务任务，创建一个新的贷款申请，使用已经存在的LoanRequestBean 接受启动流程时候的变量（例如 可以来自流程启动时候的表单）。 使用activiti:resultVariable（它作为一个变量对表达式返回的结果进行存储）将创建出来的实体作为变量进行存储。
- 用户任务，允许经理查看贷款申请，并填入审批意见（同意/不同意），审批意见将作为一个boolean变量approvedByManager进行存储
- 服务任务，更新贷款申请实体因此该实体与流程保持同步
- 根据贷款申请实体变量approved的值，将利用唯一网关（BPMN2规范）自动决定下一步该选择那一条路径：当申请批准，流程结束。否则，一个额外的任务将会使用（发送拒绝信），所以这样就是可以让拒绝信手动通知客户。

请注意该流程并不包含任何表单，因为它仅仅被用于单元测试。



```

<?xml version="1.0" encoding="UTF-8"?>
<definitions id="taskAssigneeExample"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:activiti="http://activiti.org/bpmn"
  targetNamespace="org.activiti.examples">

  <process id="LoanRequestProcess" name="Process creating and handling loan request">
    <startEvent id='theStart' />
    <sequenceFlow id='flow1' sourceRef='theStart' targetRef='createLoanRequest' />

    <serviceTask id='createLoanRequest' name='Create loan request'
      activiti:expression="${loanRequestBean.newLoanRequest(customerName, amount)}"
      activiti:resultVariable="loanRequest"/>
    <sequenceFlow id='flow2' sourceRef='createLoanRequest' targetRef='approveTask' />

    <userTask id="approveTask" name="Approve request" />
    <sequenceFlow id='flow3' sourceRef='approveTask' targetRef='approveOrDissapprove' />

    <serviceTask id='approveOrDissapprove' name='Store decision'
      activiti:expression="${loanRequest.setApproved(approvedByManager)}" />
    <sequenceFlow id='flow4' sourceRef='approveOrDissapprove' targetRef='exclusiveGw' />

    <exclusiveGateway id="exclusiveGw" name="Exclusive Gateway approval" />
    <sequenceFlow id="endFlow1" sourceRef="exclusiveGw" targetRef="theEnd">
      <conditionExpression xsi:type="tFormalExpression">${loanRequest.approved}</conditionExpression>
    </sequenceFlow>
    <sequenceFlow id="endFlow2" sourceRef="exclusiveGw" targetRef="sendRejectionLetter">
      <conditionExpression xsi:type="tFormalExpression">${!loanRequest.approved}</conditionExpression>
    </sequenceFlow>

    <userTask id="sendRejectionLetter" name="Send rejection letter" />
    <sequenceFlow id='flow5' sourceRef='sendRejectionLetter' targetRef='theOtherEnd' />

  <endEvent id='theEnd' />
  <endEvent id='theOtherEnd' />
</process>

```

```
</process>  
</definitions>
```

虽然上面的例子非常的简单，但是它却展示了JPA结合Spring和参数化方法表达式的强大优势。这样所有的流程就不需要自定义java代码（当然，除了Spring bean之外） 并且大幅度的加快了流程部署。

第#11#章#历史

历史是一个组件，它可以捕获发生在进程执行中的信息并永久的保存，与运行时数据不同的是，当流程实例运行完成之后它还会存在于数据库中。

有5个历史实体对象：

- `HistoricProcessInstances` 包含当前和已经结束的流程实例信息。
- `HistoricVariableInstances` 包含最新的流程变量或任务变量。
- `HistoricActivityInstances` 包含一个活动(流程上的节点)的执行信息。
- `HistoricTaskInstances` 包含关于当前和过去的(已完成或已删除)任务实例信息。
- `HistoricDetails` 包含历史流程实例、活动实例、任务实例的各种信息。

由于数据库中保存着历史信息以及正在运行的流程实例信息，就要考虑怎样尽量少的对运行中的流程实例数据进行访问的方式查询这些表来保证执行的性能。

稍后，这个信息体现在Activiti Explorer。同时它也是报告将生成的信息。

11.1. #查询历史

在API中，提供了对这5种实体的查询方法。类`HistoryService` 提供了以下几种方法
`createHistoricProcessInstanceQuery()`,
`createHistoricVariableInstanceQuery()`,
`createHistoricActivityInstanceQuery()`,
`createHistoricDetailQuery()` 和
`createHistoricTaskInstanceQuery()`。

下面是一些API中查询历史信息的例子。这些方法的详细描述可以在 `ZAjavadocs [.. / javadocs/index.html]` 中找到，在包`org.activiti.engine.history` 中

11.1.1. #HistoricProcessInstanceQuery

流程实例。

获取流程定义ID是'XXX'、已经结束、花费时间最长（持续时间最长）的10个`HistoricProcessInstances`

```
historyService.createHistoricProcessInstanceQuery()
    .finished()
    .processDefinitionId("XXX")
    .orderByProcessInstanceDuration().desc()
    .listPage(0, 10);
```

11.1.2. #HistoricVariableInstanceQuery

在ID为'xxx'、已经结束的流程实例中查询所有`HistoricVariableInstances`，并按变量名排序

```
historyService.createHistoricVariableInstanceQuery()
    .processInstanceId("XXX")
    .orderByVariableName.desc()
```

```
.list();
```

11.1.3. #HistoricActivityInstanceQuery

获取所有已经结束的流程定义ID为' XXX' 并且类型是' serviceTask' 中的最后一个 HistoricActivityInstance

```
historyService.createHistoricActivityInstanceQuery()
    .activityType("serviceTask")
    .processDefinitionId("XXX")
    .finished()
    .orderByHistoricActivityInstanceEndTime().desc()
    .listPage(0, 1);
```

11.1.4. #HistoricDetailQuery

下个例子， 获取所有id为123的流程实例中产量的可变更新信息。这个查询只会返回 HistoricVariableUpdates。注意一些变量名可能包含多个 HistoricVariableUpdate 实体，每次流程运行时会更新变量。你可以用 orderByTime (变量被更新的时间) 或者 orderByVariableRevision (运行更新时变量的版本) 来排序查询。

```
historyService.createHistoricDetailQuery()
    .variableUpdates()
    .processInstanceId("123")
    .orderByVariableName().asc()
    .list()
```

这个例子获取所有流程实例ID为123的流程中，提交任务或者启动流程时的form-properties 。这个查询只会返回 HistoricFormPropertiess 。

```
historyService.createHistoricDetailQuery()
    .formProperties()
    .processInstanceId("123")
    .orderByVariableName().asc()
    .list()
```

最后这个例子获取所有在执行ID为123的任务时的变量更新。 返回全部在任务中设置的变量 (任务局部变量) HistoricVariableUpdates，不是流程实例变量。

```
historyService.createHistoricDetailQuery()
    .variableUpdates()
    .taskId("123")
    .orderByVariableName().asc()
    .list()
```

任务局部变量可以用 TaskService 设置或者使用 DelegateTask，在TaskListener里设置：

```
taskService.setVariableLocal("123", "myVariable", "Variable value");
```

```
public void notify(DelegateTask delegateTask) {
    delegateTask.setVariableLocal("myVariable", "Variable value");
}
```

11.1.5. #HistoricTaskInstanceQuery

获取所有任务中10个花费时间最长（持续时间最长）并已经结束的 HistoricTaskInstances。

```
historyService.createHistoricTaskInstanceQuery()
    .finished()
    .orderByHistoricTaskInstanceDuration().desc()
    .listPage(0, 10);
```

获取删除原因包含“无效”，最后分配给用户“kermit”的 HistoricTaskInstances。

```
historyService.createHistoricTaskInstanceQuery()
    .finished()
    .taskDeleteReasonLike("%invalid%")
    .taskAssignee("kermit")
    .listPage(0, 10);
```

11.2. #历史配置

历史级别可以用编写代码的方法配置，
org.activiti.engine.impl.history.HistoryLevel
在ProcessEngineConfiguration中的常量 HISTORY_*）：

用枚举类型
(或者在5.11之前定义

```
ProcessEngine processEngine = ProcessEngineConfiguration
    .createProcessEngineConfigurationFromResourceDefault()
    .setHistory(HistoryLevel.AUDIT.getKey())
    .buildProcessEngine();
```

级别可以在配置文件 activiti.cfg.xml 或者在 spring-context 中配置：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
    <property name="history" value="audit" />
    ...
</bean>
```

历史信息级别可以配置成以下几种：

- none：忽略所有历史存档。这是流程执行时性能最好的状态，但没有任何历史信息可用。
- activity：保存所有流程实例信息和活动实例信息。在流程实例结束时，最后一个流程实例中的最新的变量值将赋值给历史变量。不会保存过程中的详细信息。
- audit：这个是默认值。它保存所有流程实例信息，活动信息，保证所有的变量和提交的表单属性保持同步。这样所有用户交互信息都是可追溯的，可以用来审计。
- full：这是最高级别的历史信息存档，同样也是最慢的。这个级别存储发生在审核以及所有其它细节的信息，主要是更新流程变量。

在Activiti 5.11之前，历史级别都存在数据库中（表 ACT_GE_PROPERTY，属性名为 historyLevel）。从Activiti 5.11开始，这个值不再用，并且从数据库中忽略或者删除掉。这个配置可以在两次启动间修改，因为是在启动之前修改的，所以不会抛出异常。

11.3. #审计目的的历史

当配置在 audit 级别之上。所有通过 `FormService.submitStartFormData(String processDefinitionId, Map<String, String> properties)` 和 `FormService.submitTaskFormData(String taskId, Map<String, String> properties)` 方法提交的属性都会被记录。

表单属性可以通过API查询，如下：

```
historyService
    .createHistoricDetailQuery()
    .formProperties()
    ...
    .list();
```

上面的例子只有类型为 `HistoricFormProperty` 的详细信息会被查询出来。

如果你在调用 `IdentityService.setAuthenticatedUserId(String)` 提交之前设置了认证用户，那么提交表单的用户将被保存在历史信息中并可以在开始表单中使用 `HistoricProcessInstance.getStartUserId()` 获取，在任务表单中用 `HistoricActivityInstance.getAssignee()` 获取。

第#12#章#Eclipse Designer

Activiti 同时还有个Eclipse 插件，Activiti Eclipse Designer ，可用于图形化建模、测试、部署 BPMN 2.0的流程。

12. 1. #Installation

以下安装指南在 Eclipse Indigo [<http://www.eclipse.org/downloads/>]. 须注意Eclipse 的Helio 是支持NOT 的.

打开 Help -> Install New Software. 在如下面板中，点击 Add 按钮，然后填入下列字段：

- Name: Activiti BPMN 2.0 designer
- Location: <http://activiti.org/designer/update/>

Install

Available Software

Select a site or enter the location of a site.



Work with: type or select a site

Find more software by working with the ["Available Software Sites"](#) preferences.

Add Repository

Name: Activiti BPMN 2.0 designer

Location:

Show only the latest versions of available software Hide items that are already installed

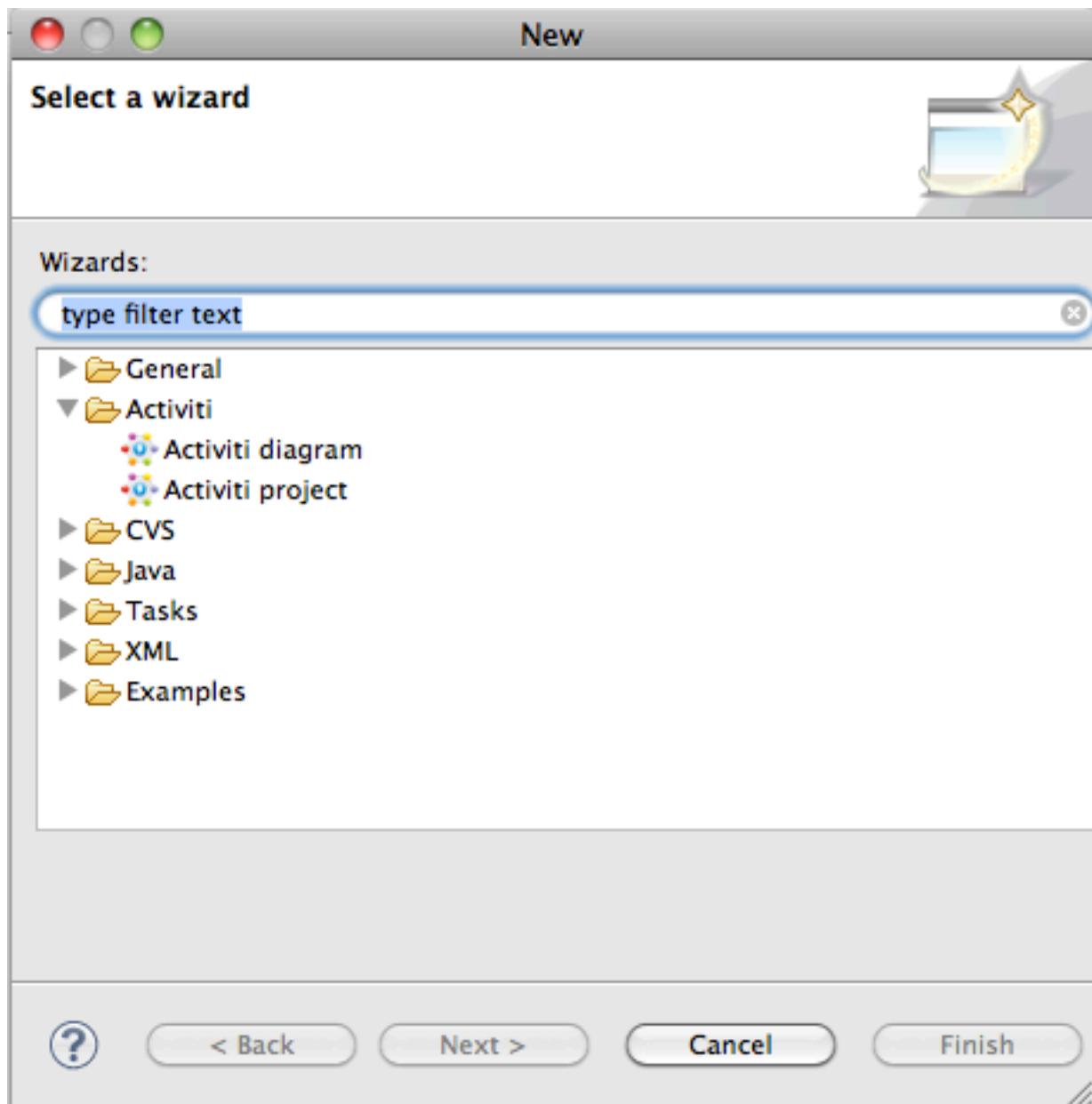
Group items by category [What is already installed?](#)

Contact all update sites during install to find required software

务必选中“Contact all update sites..”，因为它会检查所有当前安装所需要的插件并可以被Eclipse下载。

12.2. #Activiti Designer 编辑器的特性

- 创建Activiti 项目和图形.



- 当创建一个ACTIVITI表格的时候，Activiti Designer会建立一个BPMN的文档. 打开 Activiti Diagram 编辑器时会出现一个图示的画布和调色板, 同一文件在用XML编辑器打开后, 会显示BPMN2.0程定义XML元素. 因此, 同一个文件能同时使用图形图以及BPMN 2.0XML. 注意Activiti的5.9 BPMN扩展尚未作为部署文件来支持程定义. 因此, Activiti Designer 的“创建部署文件”功能生成一个BAR文件, BAR里面有一个含.bpmn内容的.bpmn20.xml文件. 您也可以做一个快速的文件重命名. 另外, 你也可直接用Activiti Designer 打开.bpmn20.xml文件.

The screenshot shows the Eclipse Designer interface with a project named "MyProcess".

Diagram View:

```

graph LR
    Start(( )) -- to usertask --> UserTask[User Task]
    UserTask -- ending --> End(( ))
  
```

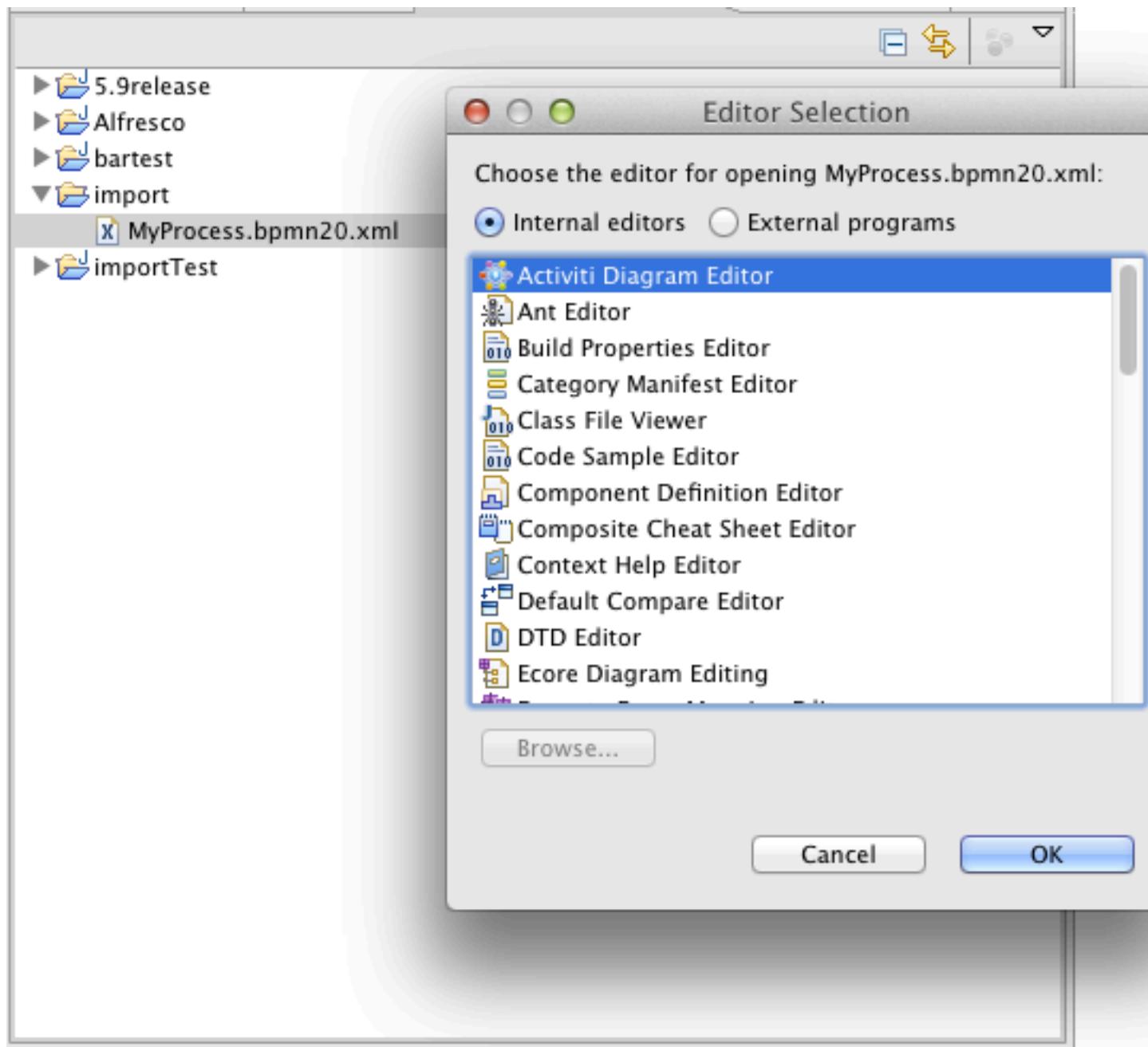
XML View:

```

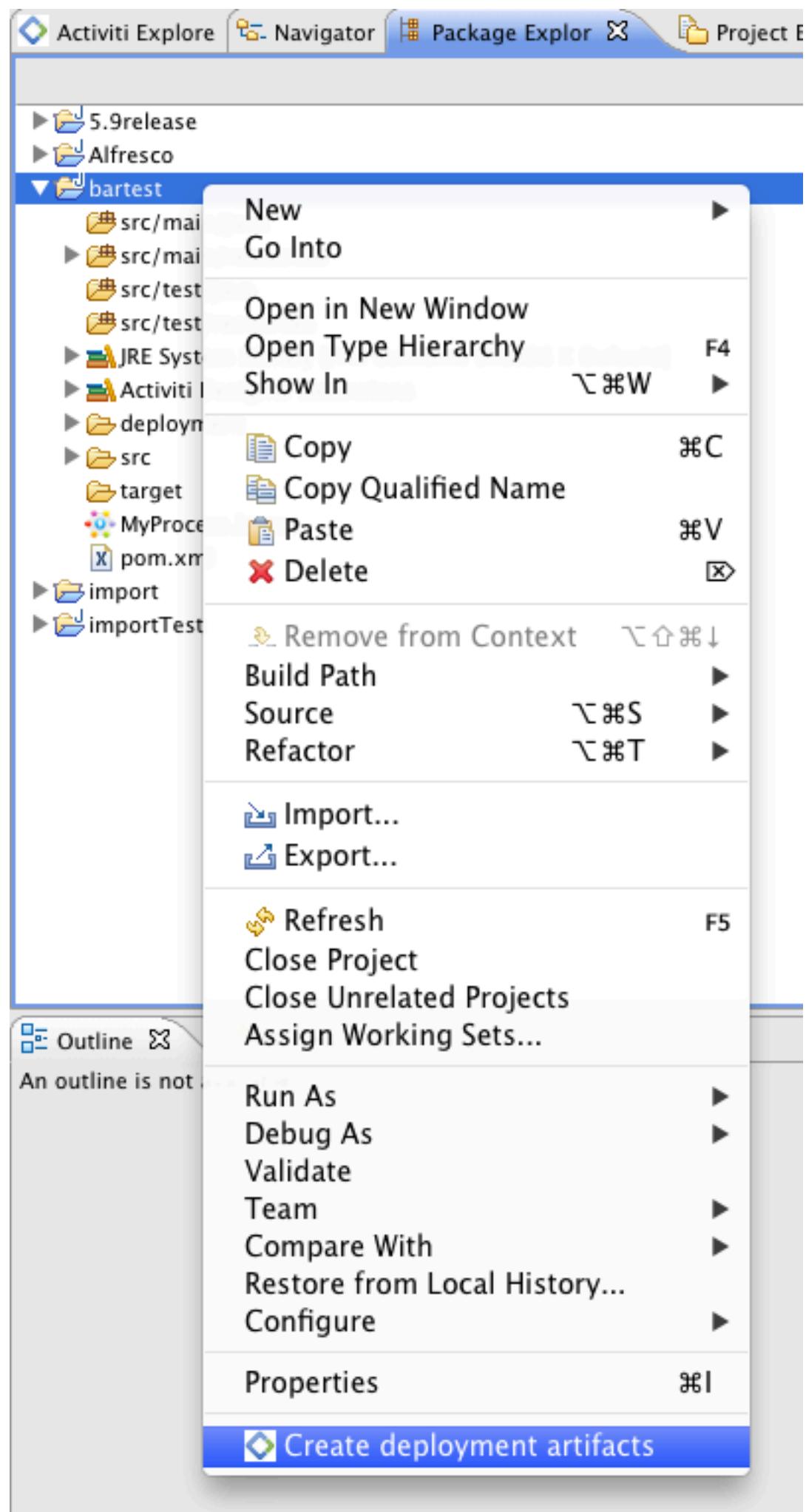
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  <process id="process1" name="process1">
    <startEvent id="startevent1" name="Start"></startEvent>
    <userTask id="usertask1" name="User Task" activiti:assignee="<!-->">
      <extensionElements>
        <activiti:formProperty id="name" name="Name" type="string"/>
      </extensionElements>
    </userTask>
    <endEvent id="endevent1" name="End"></endEvent>
    <sequenceFlow id="flow1" name="to usertask" sourceRef="startevent1" targetRef="usertask1"/>
    <sequenceFlow id="flow2" name="ending" sourceRef="usertask1" targetRef="endevent1"/>
  </process>

```

- BPMN 2.0 XML文件可以导入Activiti Designer，并创建一个图表，只需复制BPMN 2.0 XML进你的项目，再用Activiti Designer打开它。Activiti Designer用BPMN DI的文件创建图表信息。如你有一个BPMN2.0的XML文件而无BPMN DI信息，那也无图可创建。



- 对于部署 BAR 文件以及可选的 JAR 文件,可以通过右击 package explorer视图中的 Activiti项目,然后选择弹出菜单底问的 Create deployment artifacts 选项 由Activiti Designer来创建. 更多关于Designer的部署功能,见 部署 一节.



- 生成单元测试 (在package explorer 视图下右击BPMN2.0的XML文件, 然后选择 generate unit test) 生成的单元测试是配置为运行有H2数据 库上的Activiti配置. 你现在就可以运行单元测试来测试你的流程定义了.

Activiti Explorer Navigator Package Explorer Project Explorer

5.9release
Alfresco
bartest
src/main/java
src/main/resources
src/test/java
src/test/resources
JRE System Library [JVM Contents (MacOS X Default)]
Activiti Designer Extensions
deployment
src
target

MyProcess.bpm
pom.xml
import
importTest

New

- Open F3
- Open With ►
- Show In ⌘W ►

- Copy ⌘C
- Copy Qualified Name
- Paste ⌘V
- Delete ✖

- Remove from Context ⌘↓
- Mark as Landmark ⌘↑

- Build Path ►
- Refactor ⌘T ►

- Import... ↴
- Export... ↴

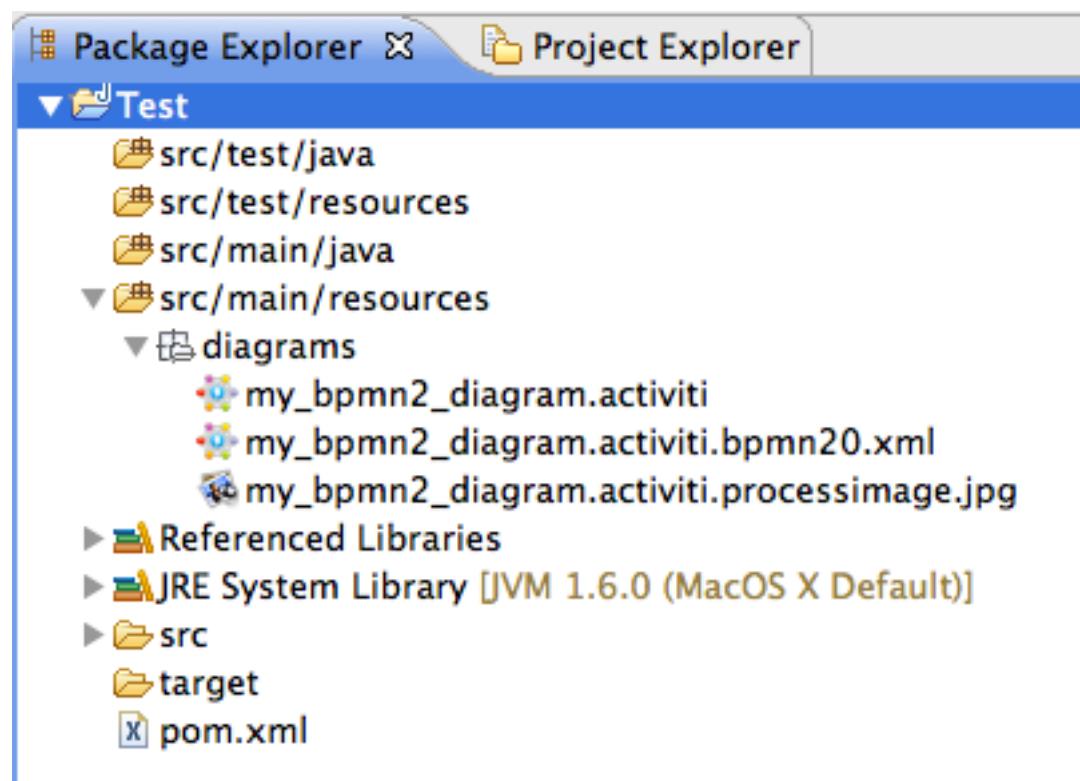
- Refresh F5
- Assign Working Sets...

- Validate
- Run As ►
- Debug As ►
- Team ►
- Compare With ►
- Replace With ►
- Activiti ►
- Source ►

An outline is not available

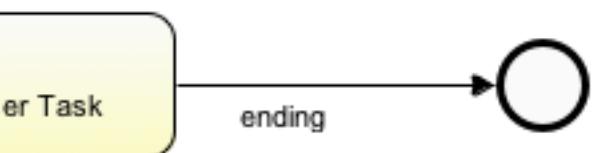
Generate unit tes

- Activiti 项目可以做为Maven 项目来生成. 但是需要配置依赖关系 需要运行 mvn eclipse:eclipse 这样Maven的依赖会像预期的那样被配置. 注意对于流程设计来讲是不需要Maven的依赖. 只是在运行单元测试时需要.

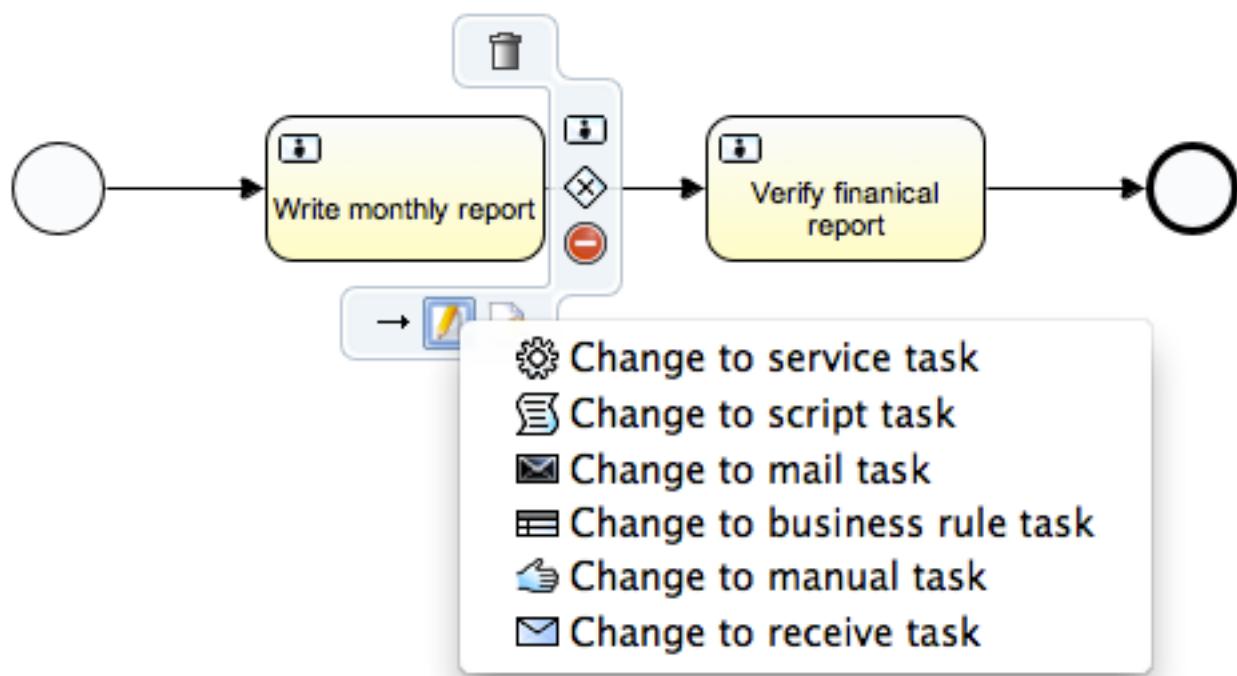


12.3. #Activiti Designer 的BPMN 特性

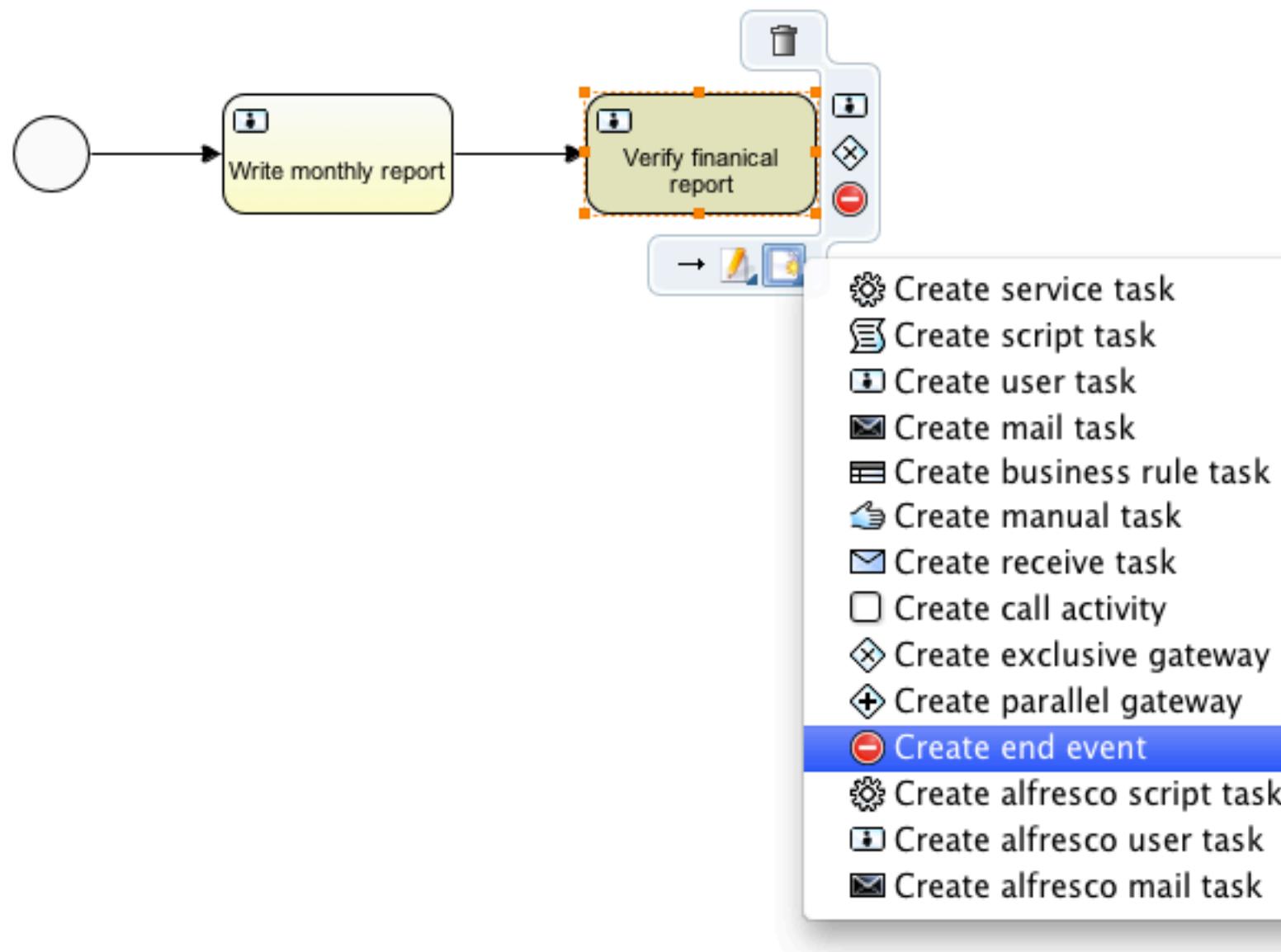
- 支持开始没有事件, 启动错误事件、定时器启动事件, 最后没有一个错误事件, 事件, 结束序列流, 并行网关, 网关, 网关, 独家包容事件网关, 嵌入式子流程、事件子流程、调用活动, 游泳池, 车道, 脚本任务、用户任务、服务任务, 邮件任务, 手动任务, 业务规则任务, 获得任务, 定时器边界事件、错误边界事件、信号边界事件、定时器捕获事件, 信号捕捉事件、信号投掷项目, 和四个Alfresco特定的元素(用户、脚本、邮件任务和开始事件).



- 可以很快地改变一个任务的类型,只需鼠标悬停在该元素上,然后选择新的任务类型.



- 可以快速地添加一个新元素,只需鼠标悬停某个元素上,然后选择一个新的元素类型.



- 支持对Java 服务任务的Java 类、表达式以及代理表达式的配置. 此外, 也可以配置字段扩展.

General Type: Java class Expression Delegate expression

Main config

Listeners

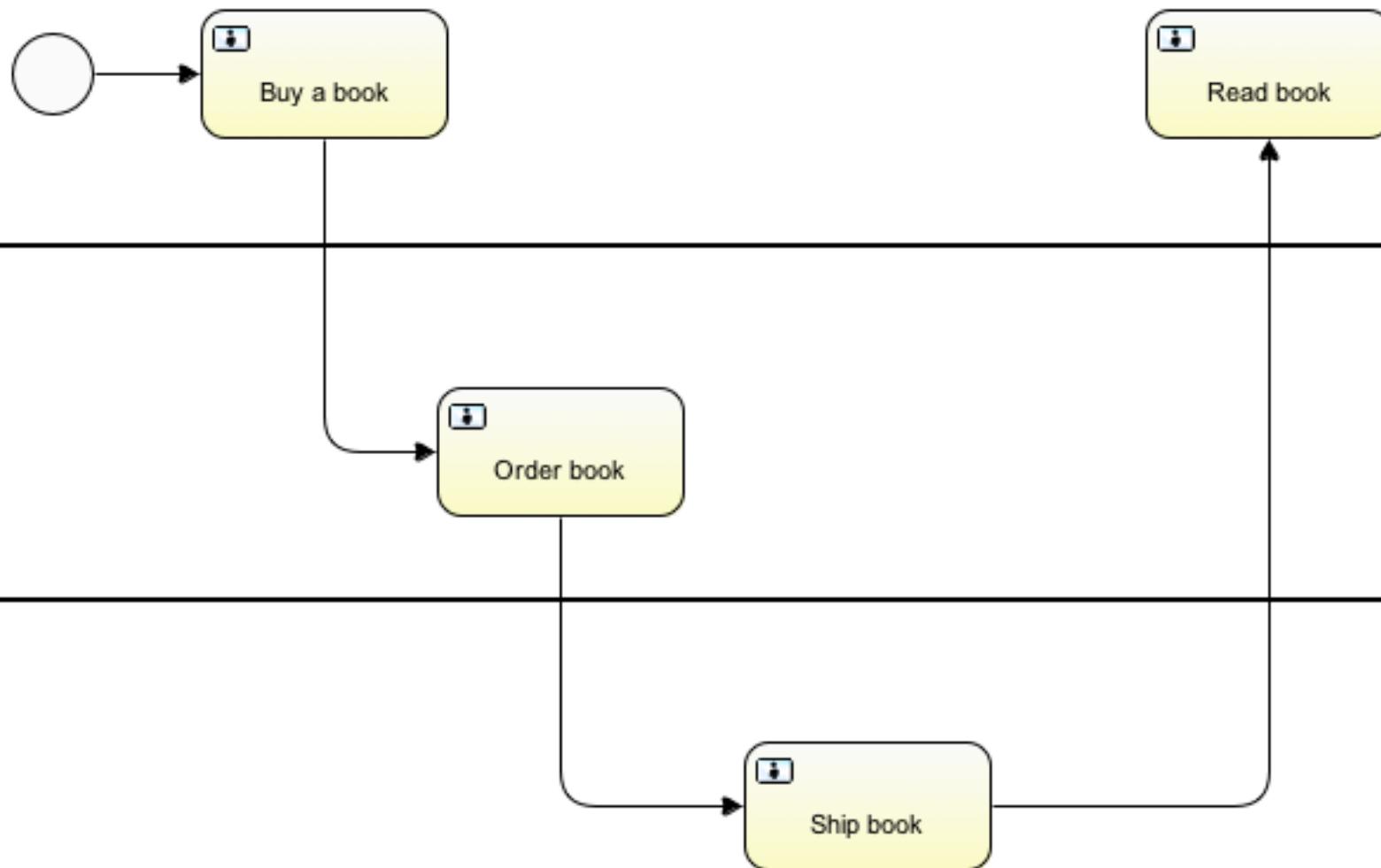
Type: Expression: \${printer.printMessage}

Result variable:

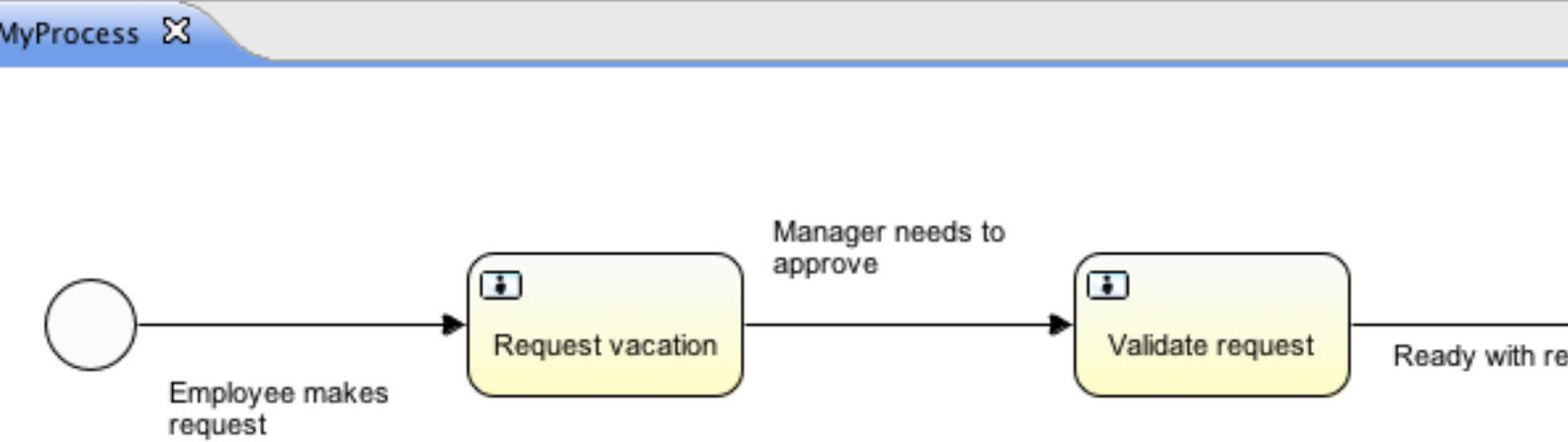
Fields:

Field name	String value / Expression
message	hello

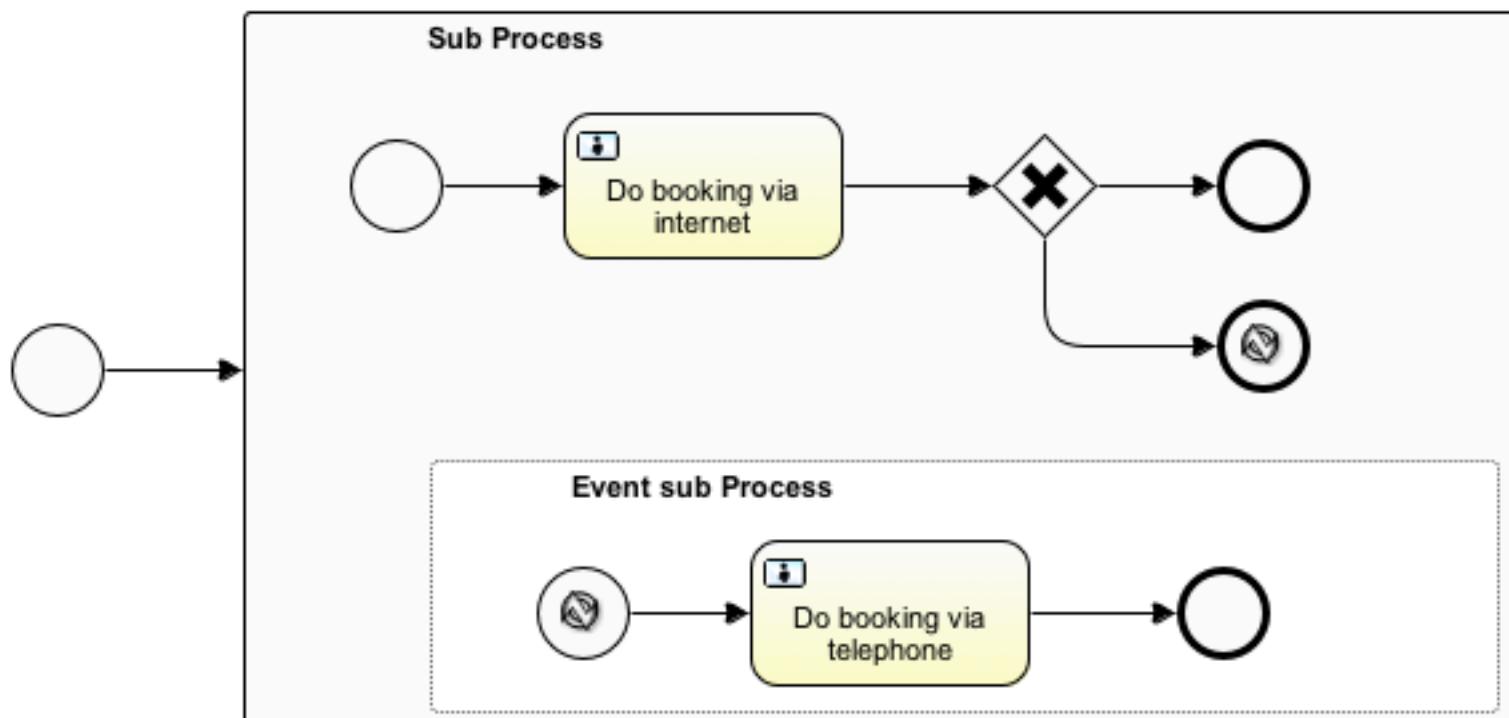
- 支持pools 和 lanes, 因为Activiti能根据不同的程定义读取不同的存储池. 它使仅用一个存储池看起来意义重大. 如果您使用多个池, 要注意储存池的图序列流动, 它可能会在部署AD引擎的过程中发生问题. 当然, 只要你想, 你可以添加尽可能多的POOL, LANES.



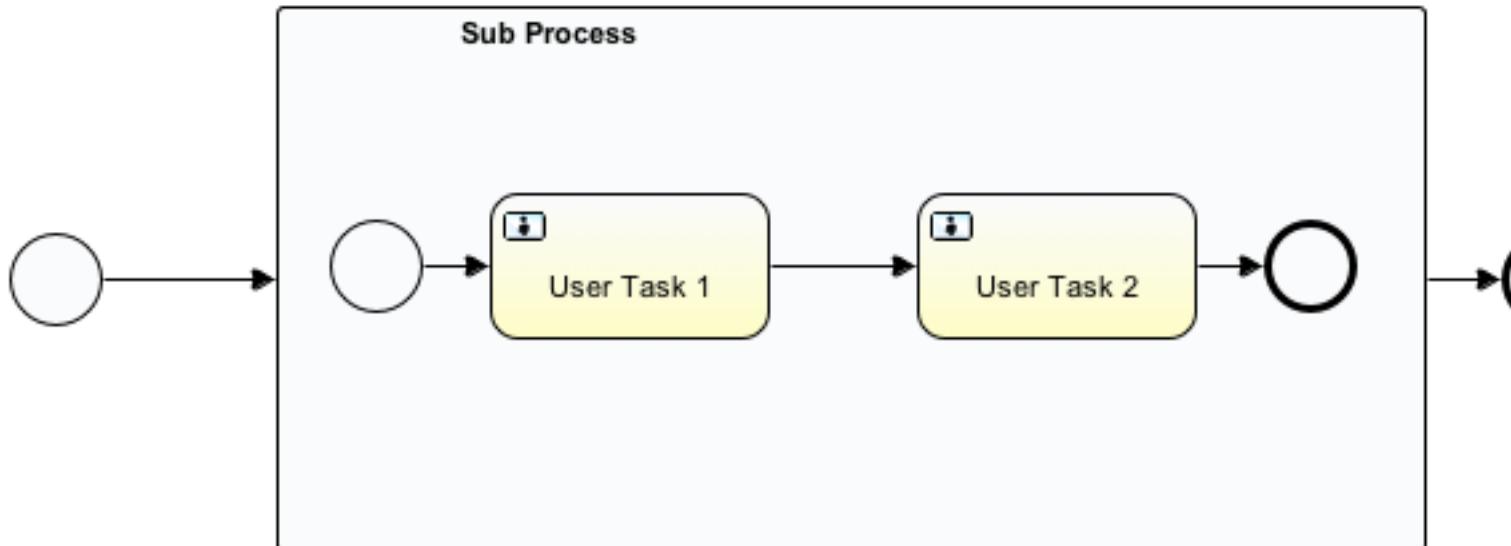
- 您可以通过填写名称属性添加标签序列流. 你可以自己定位标签的位置保存为BPMN 2 XML DI部分信息



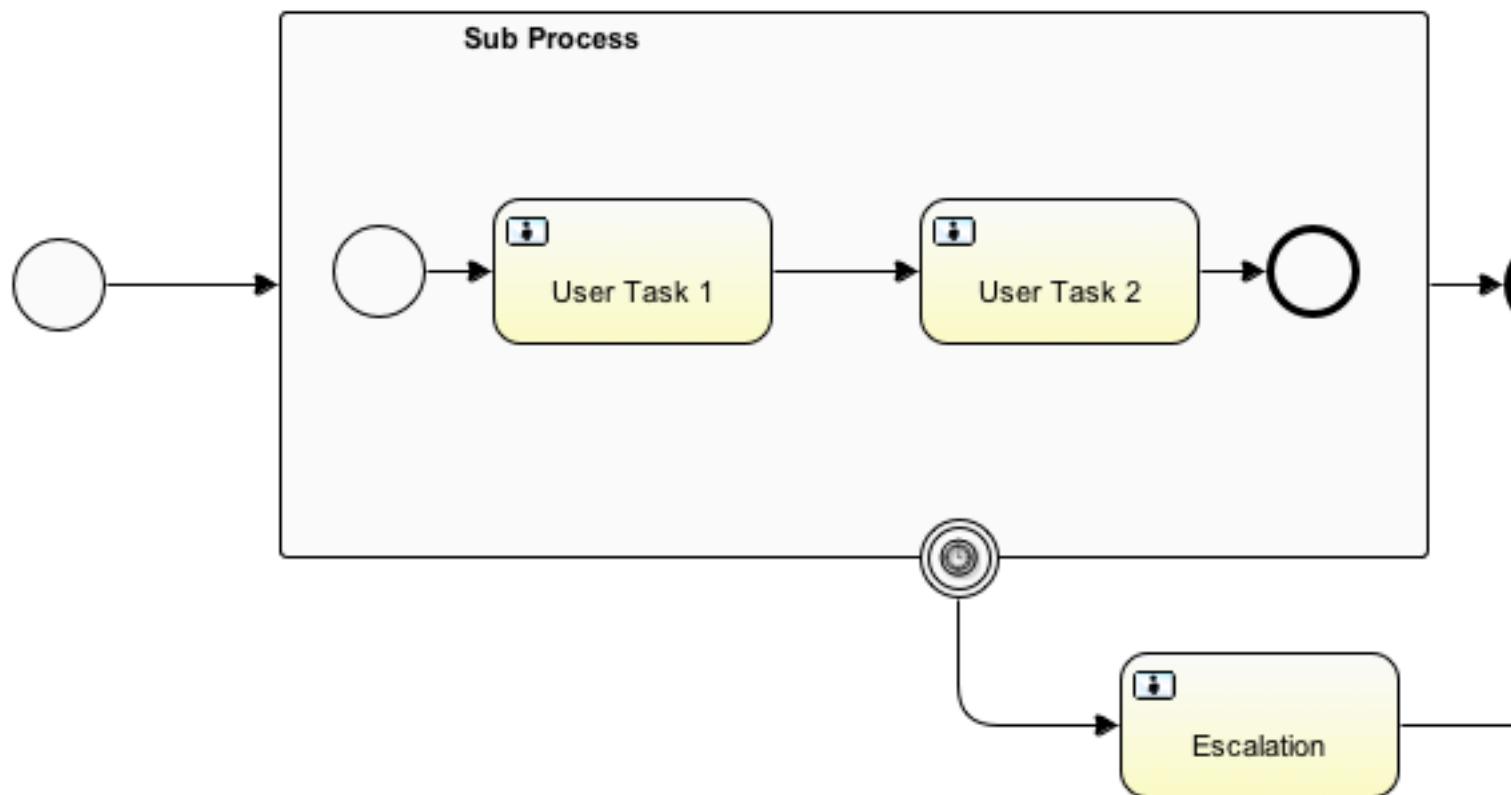
- 支持子流程事件.



- 支持嵌入式子流程. 你还可以添加一个嵌入式子流程进另一个嵌入式子流程.



- 支持任务和嵌入子过程上的定时器边界事件. 尽管, 定时器边界事件最大的意义在于可以在 Activiti Designer 中 的用户任务或嵌入子过程上使用.



- 支持附加的Activiti 的扩展, 如邮件任务、用户任务候选者配置以及脚本任务的配置.



Screenshot of the Eclipse Designer interface showing the configuration of a Mail Task.

The top navigation bar includes: Problems, Properties, Ant, Error Log, and Console.

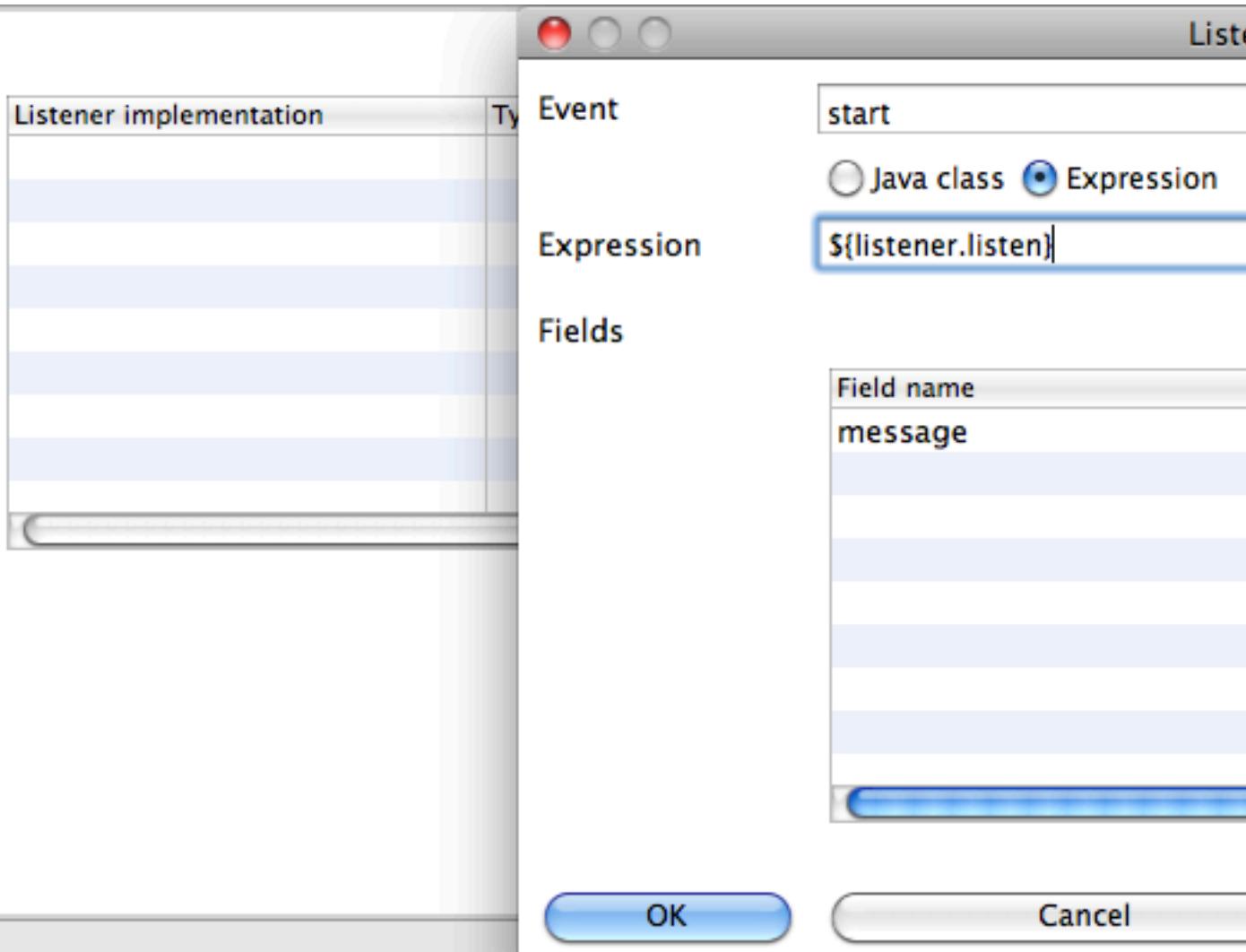
The left sidebar shows categories: General (selected), Main config, Listeners, and Multi instance.

The configuration fields are:

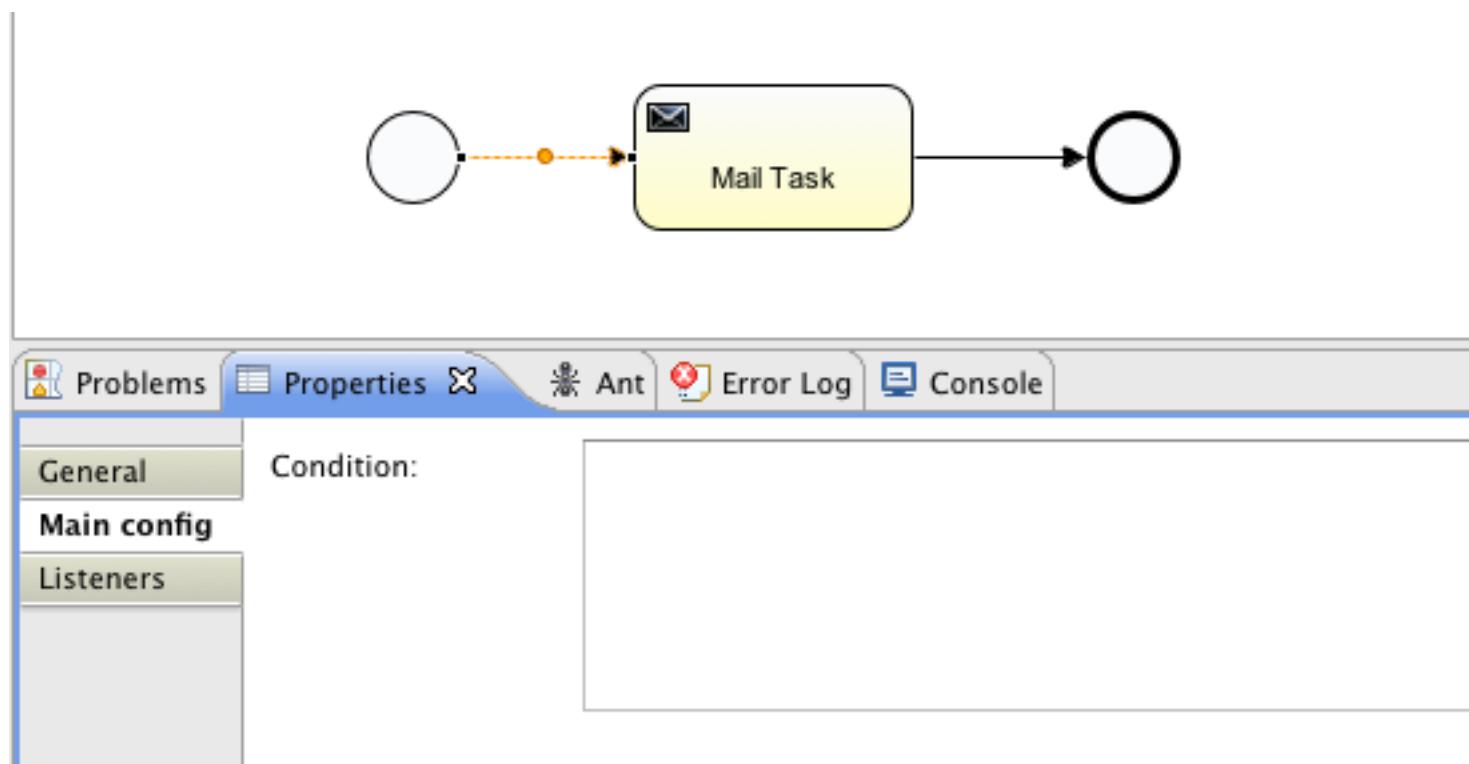
- To: [Text input field]
- From: [Text input field]
- Subject: [Text input field]
- Cc: [Text input field]
- Bcc: [Text input field]
- Html text: [Large text area]
- Non-HTML text: [Large text area]

- 支持Activiti 执行监听和任务监听. 可以给执行监听添加字段扩展.

rs:

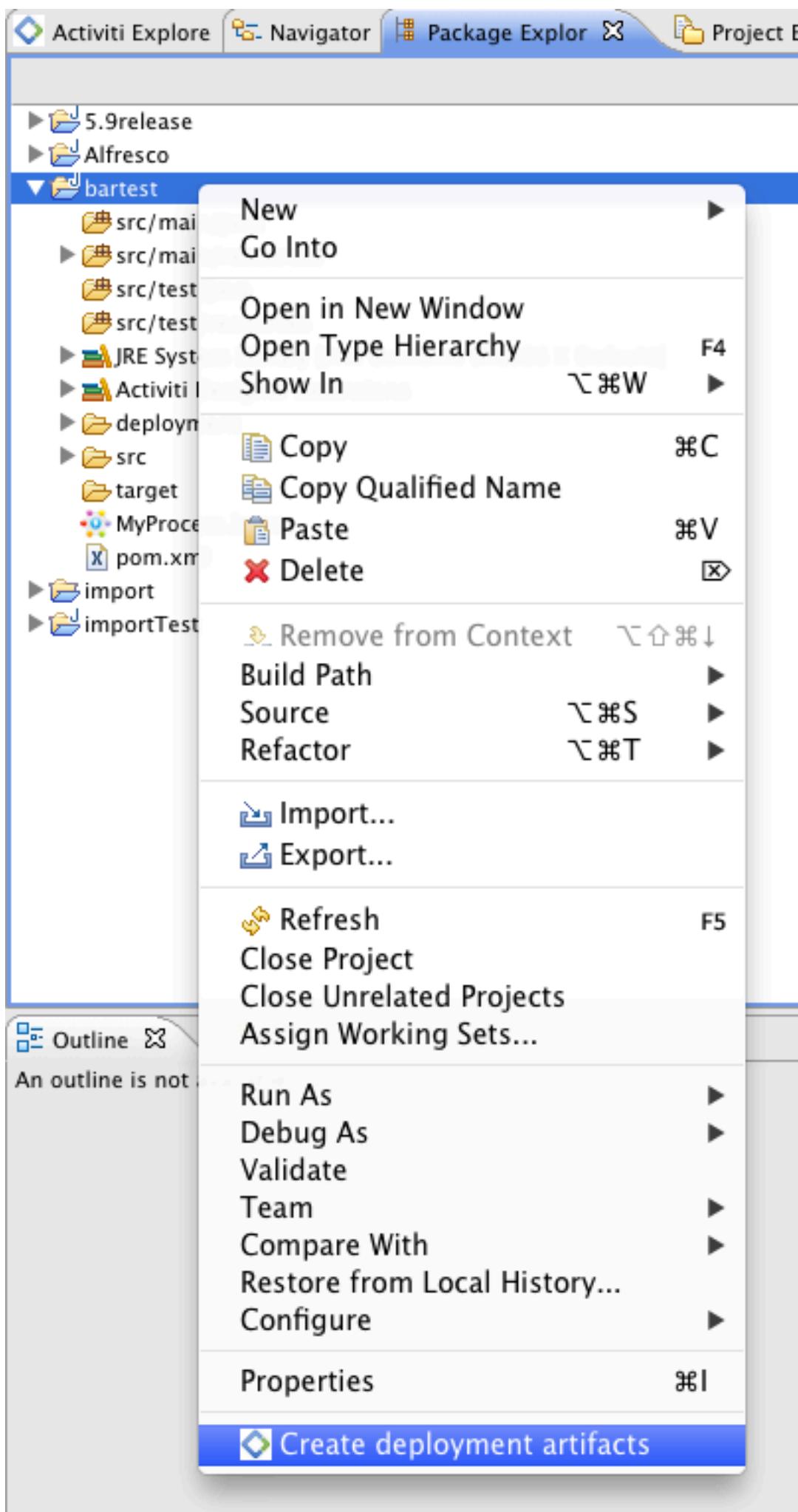


- 支持顺序流上的条件.

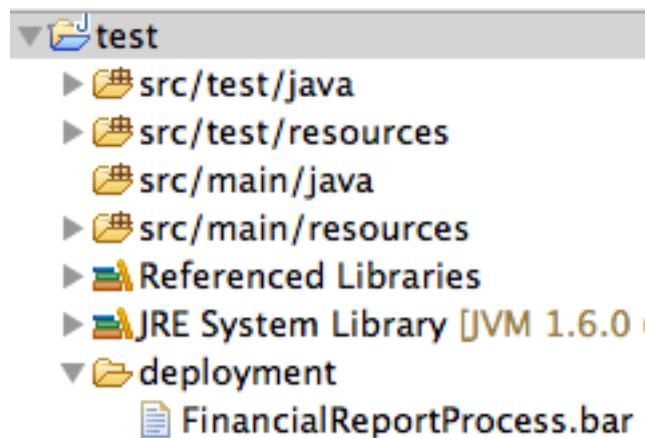


12. 4. #Activiti Designer 部署特性

将流程定义和任务表单部署到Activiti引擎并不难. 要准备一个BAR 文件, 其内含有流程定义的BPMN 2.0的XML 文件, 还可以有任务表单以及可以在Activiti Explorer中浏览的流程图片. 在Activiti Designer中创建BAR文件是非常简单的. 完成流程实现后, 在package explorer视图下右击你的Activiti项目, 然后选择弹出菜单底部的 Create deployment artifacts 选项.



接下来, deployment目录会被创建, 其内包含有BAR文件, 你的Activiti项目中的Java类的JAR文件也可以在此.



现在就可以使用Activiti Explorer中的部署标签将这个文件部署到Activiti引擎上了, 相信你已经准备好了.

当你的项目中含有Java 类时, 部署要稍微麻烦一点. 这种情况下, Activiti Designer 中的 Create deployment artifacts 同时会生成包含编译了的类的JAR 文件. 必须将这个JAR文件部署到你的Activiti Tomcat安装路径下的activiti-XXX/WEB-INF/lib路径下. 以使得这些类在Activiti引擎的类路径下是可用的.

12.5. #扩展Activiti Designer

你可以对Activiti Designer提供的默认功能进行扩展. 这一节介绍有哪些扩展可用, 如何使用这些扩展, 并提供了一些使用的例子. 当默认的功能不能满足要求, 需要额外功能时, 或者业务流程建模时有领域 所特有的需求时, 扩展Activiti Designer就变得很有用了. Activiti Designer的扩展分为两种不同的类型, 扩展画板和扩展输出格式. 每种形式都有其特有的方式和不同的专业技术.

注意

扩展Activiti Designer 需要有技术知识以及专业的Java 编程的知识. 根据你要创建的扩展类型, 你可能也要熟悉Maven、Eclipse 、OSGI、Eclipse扩展以及SWT.

12.5.1. #定制画板

流程建模时, 可以定制显示给用户的画板. 画板是那些可在流程图形的画布上拖拽的形状的集合, 它显示在画布右边. 正如你在默认画板所看到的. 默认的形状被分组到了事件、分支等区隔(称为“抽屉”) 中. Activiti Designer 中有两种内置的选择来定制画板中的抽屉和形状:

- 将你自己的形状/节点添加到现有的或新的抽屉内.
- 禁用Activiti Designer提供的任何或所有的默认BPMN 2.0的形状, 除了连接工具和选择工具.

要想定制画板, 需要创建一个JAR 文件, 并将其添加到Activiti Designer 特定的安装目录下 (后面有更多关于 如何操作). 这样的 JAR 文件称为 扩展. 通过编写包含在扩展中的类, Activiti Designer就能知道你 想要的定制. 为了使其运行, 类必须实现某些接口. 你需要将一个集成了这些接口以及一些继承用的基础 类的类库添加到你项目的类路径下.

你可以在Activiti Designer管理的源码下找到下面列出的代码实例. 查看Activiti源码 projects/designer 目录下的examples/money-tasks 目录.

注意

可以使用你所偏爱的工具来设置你的项目, 然后使用你选择的构建工具来构建JAR文件. 下面的说明, 设置假设使用的是Eclipse Helios , Maven (3. x) 作为构建构建工具, 但任何其它的设置也都能让你创建同样的结果.

12.5.1.1. #扩展的设置 (Eclipse/Maven)

下载并解压 Eclipse [http://www.eclipse.org/downloads] (使用最新版本) 和最新版本 (3. x) 的Apache Maven [http://maven.apache.org/download.html]. 如果你使用2. x版本的 Maven, 在构建项目时会运行出错, 所以确保你的版本是最新的. 我们假设你能熟练使用基本特性和Eclipse的Java编辑器. 由你决定是使用Eclipse 的Maven 特性还是在命令行上运行 Maven 命令.

在Eclipse里创建一个新项目. 这是一个普通的项目类型. 在项目的根路径下创建 pom.xml 文件用来包含Maven 项目的设置. 同时创建 src/main/java 和src/main/resources文件, 这是Maven 对于Java 源文件和资源的约定. 打开pom.xml 文件并添加下列代码:

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.acme</groupId>
    <artifactId>money-tasks</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
    <name>Acme Corporation Money Tasks</name>
    ...
</project>
```

正如你所看到的, 这只是一个定义了项目groupId, artifactId 和 version的基本的pom.xml 文件. 我们将创建一个定制, 其中包含我们money业务中一个的自定义节点.

通过在pom.xml 文件内引入依赖, 就可以将集成的类库添加到你项目的依赖中:

```
<dependencies>
    <dependency>
        <groupId>org.activiti.designer</groupId>
        <artifactId>org.activiti.designer.integration</artifactId>
        <version>5.12.0</version> <!-- Use the current Activiti Designer version -->
        <scope>compile</scope>
    </dependency>
</dependencies>
...
<repositories>
    <repository>
        <id>Activiti</id>
        <url>https://maven.alfresco.com/nexus/content/groups/public/</url>
    </repository>
```

```
</repositories>
```

最后, 在 `pom.xml` 文件中添加对 `maven-compiler-plugin` 的配置, Java 源代码级别最低是 1.5 (看下面的代码片段). 这在使用注解时需要。也可以包含让 Maven 生成 JAR 的 `MANIFEST.MF` 文件的命令。这不是必需的, 但这样你可以使用清单文件的一个特定的属性作为扩展的名称 (这个名称可以显示在 Designer 的某处, 主要是为以后如果 Designer 中有几个扩展的时候使用)。如果你想这样做, 将以下代码片段包含进 `pom.xml` 文件:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
        <showDeprecation>true</showDeprecation>
        <showWarnings>true</showWarnings>
        <optimize>true</optimize>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.3.1</version>
      <configuration>
        <archive>
          <index>true</index>
          <manifest>
            <addClasspath>false</addClasspath>
            <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
          </manifest>
          <manifestEntries>
            <ActivitiDesigner-Extension-Name>Acme Money</ActivitiDesigner-Extension-Name>
          </manifestEntries>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

扩展的名称是由 `ActivitiDesigner-Extension-Name` 属性描述的。现在剩下唯一要做的就是让根据 `pom.xml` 件中的指令来设置项目了。所以打开命令窗口, 转到 Eclipse 工作空间下你的项目的根文件夹。然后执行以下 Maven 命令:

```
mvn eclipse:eclipse
```

等待构建完成。刷新项目 (使用项目上下文菜单 (右击), 选择 Refresh)。此时, `src/main/java` 和 `src/main/resources` 文件夹应该已经是 Eclipse 项目下的资源文件夹了。

注意

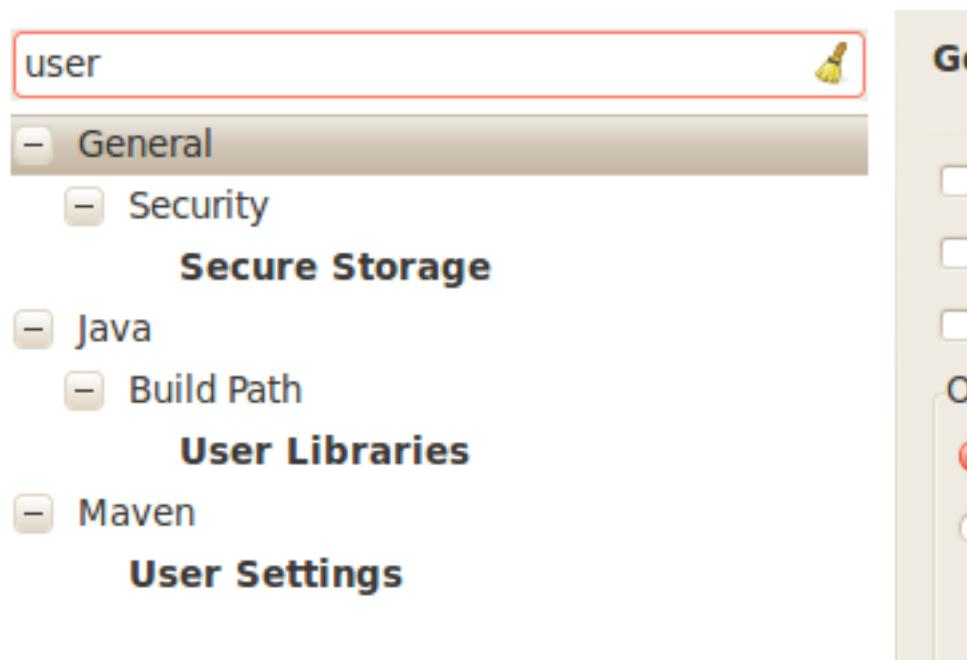
当然你也可以使用 `m2eclipse` [<http://www.eclipse.org/m2e>] 插件, 只要从项目的上下文菜单 (右击) 就能启用 Maven 依赖管理。然后在项目上下文菜单中选择 `Maven > Update project configuration` 这也能设置源文件夹。

设置就这样。现在就可以开始创建 Activiti Designer 的定制了!

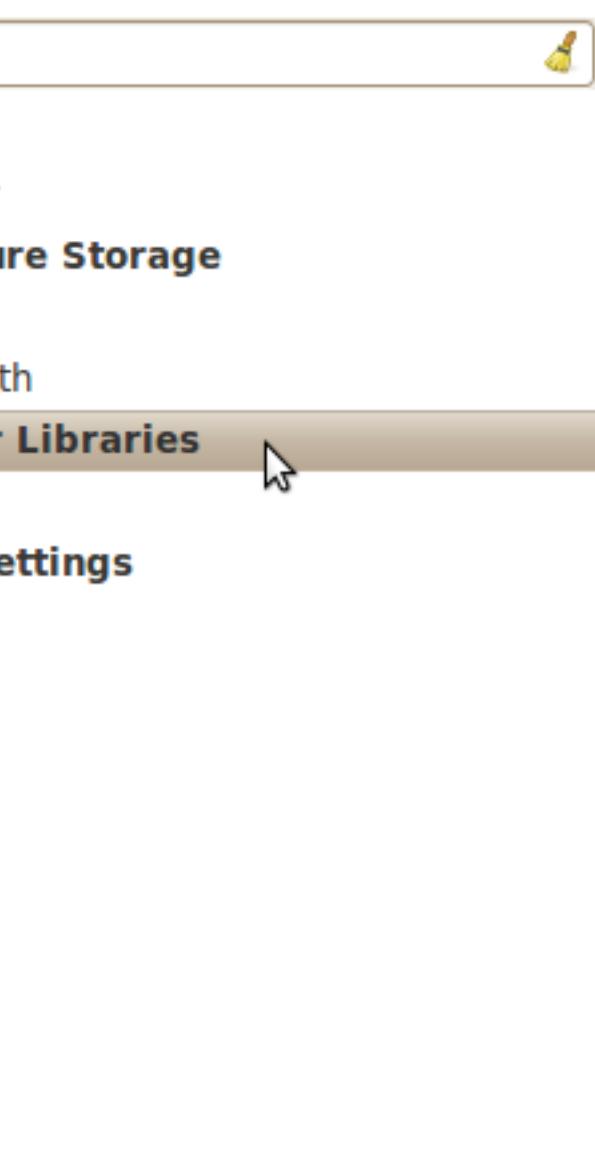
12.5.1.2. #将扩展应用到Activiti Designer

你可能想要知道怎样才能将扩展添加到Activiti Designer 以便定制被应用. 这就是操作步骤:

- 一旦创建完扩展的JAR (例如, 利用Maven执行项目内的mvn的安装程序进行构建), 需要 将其放在计算机中安装Activiti Designer 的地方;
- 将扩展存储到硬盘中一个可以的地方, 记住这个地方. 注意: 该位置必须是Activiti Designer外Eclipse workspace以外的路径, workspace中存储扩展将导致用户得到一个弹出错误消息, 无法扩展;
- 启动Activiti Designer, 选择菜单栏中的 Window > Preferences
- 在preferences窗口中, 输入 user关键字. 在Java 部分中你应该能看到访问Eclipse 中的User Libraries一项.



- 选择User Libraries, 在右侧显示的树形视图中你可以添加类库. 你会看到用来向 Activiti Designer添加扩展的默认分组(根据安装的Eclipse, 可能也会看到另外其它的分组).



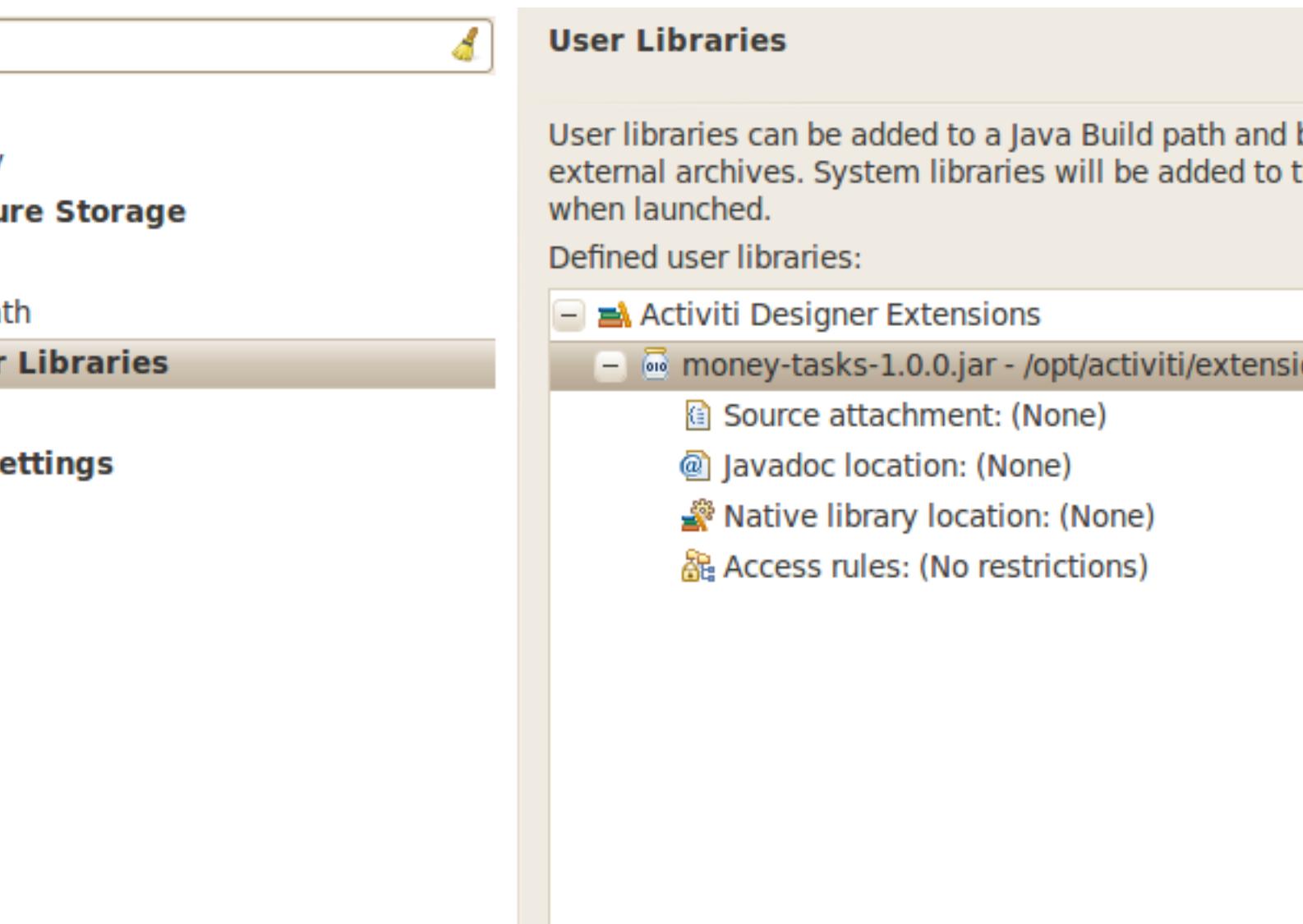
User Libraries

User libraries can be added to a Java Build path and be external archives. System libraries will be added to the when launched.

Defined user libraries:

Activiti Designer Extensions

- 选择Activiti Designer Extensions 分组, 点击Add JARs... 按钮. 导航到存储扩展的文件夹, 选择你想要添加的扩展文件. 完成后, 这个扩展就作为 Activiti Designer Extensions 分组的一部分显示在preference 窗口内, 如下显示.



- 点击OK 按钮并关闭preferences对话框. Activiti Designer Extensions 分组自动添加到了你新创建的Activiti项目. 在Navigator 或Package Explorer 视图中你可以看到此用户类库作为一项出现在项目树中。如果工作空间内已经存在Activiti 的项目了，你也会 看到新扩展显示在该分组内. 例子如下所示.



此时在打开图形的画板中将有来自新扩展的形状（或禁掉的形状，取决于扩展内的定制）. 如果 已经打开了图形, 将其关闭后重新打开看看画板中有什么变化.

12.5.1.3. #向画板添加形状

根据你的项目的设置, 现在你就能够很容易在画板中添加形状了. 每个要添加的形状都是由 JAR文件中的类来描述的. 注意这些类并不是Activiti 引擎在运行时使用的类. 在扩展内描述的那些属性可以设置给每个Activiti Designer 内的形状, you can also define the runtime characteristics that should be used by the engine when a process instance reaches the node in the process. The runtime characteristics can use any of the options that Activiti supports for regular ServiceTasks. See 这一节 for more details.

一个形状的类就是添加了一些注解的普通Java 接口, 但你不必自己实现这个接口. 只需继承类 (目前, 必须是直接继承这个类, 中间不要有抽象类) . 在这个类的Javadoc中你可以找到关于它所提供的 默认设置以及何时需要重写它所实现了的方法的说明. 重写让你做一些诸如为画板和画布上的形状提供 图标 (可以是不同的) 、指定你想让节点 (活动、事件、分支) 拥有的基本形状的事情.

```
/***
 * @author John Doe
 * @version 1
 * @since 1.0.0
 */
public class AcmeMoneyTask extends AbstractCustomServiceTask {
...
}
```

需要实现 `getName()` 方法来决定画板中节点使用的名称. 也可以将节点放进 一个属于它们自己的抽屉, 并为其提供一个图标. 重写 `AbstractCustomServiceTask.` 中恰当的方法. 如果你想要提供一个图标, 确保图标在JAR中`src/main/resources` 包 内, 大小为16x16 像素, 格式为JPEG 或PNG. 提供的路径是相对于那个文件夹的.

给图形添加属性是通过向此类添加成员变量, 并使用 `@Property` 注解对其 进行注解来完成的, 如下:

```
@Property(type = PropertyType.TEXT, displayName = "Account Number")
@Help(displayHelpShort = "Provide an account number", displayHelpLong = HELP_ACCOUNT_NUMBER_LONG)
private String accountNumber;
```

有几个.PropertyType 值你可以使用, 在这一节. 做详细介绍. 通过将`required` 属性设置为true可以使字段成为必填项. 如果用户不填写该字段, 就会有消息和红色背景显示.

如果要确保类中这些属性是它们在property 窗口出现的顺序, 需要指定 `@Property` 注解的`order`属性.

正如你所看到了, 有个 `@Help` 注解用来在填写字段时给用户提供一些引导. 也可以将`@Help` 注解用于类本身——这个信息会显示在呈现给用户的属性表的上方.

下面列出了对`MoneyTask.` 的详细阐述。添加了一个`comments` 字段, 你会看到该节点包含了一个图标.

```
/***
 * @author John Doe
 * @version 1
 * @since 1.0.0

```

```

/*
@Runtime(javaDelegateClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
@Help(displayHelpShort = "Creates a new account", displayHelpLong = "Creates a new account using the account number")
public class AcmeMoneyTask extends AbstractCustomServiceTask {

    private static final String HELP_ACCOUNT_NUMBER_LONG = "Provide a number that is suitable as an account number"

    @Property(type = PropertyType.TEXT, displayName = "Account Number", required = true)
    @Help(displayHelpShort = "Provide an account number", displayHelpLong = HELP_ACCOUNT_NUMBER_LONG)
    private String accountNumber;

    @Property(type = PropertyType.MULTILINE_TEXT, displayName = "Comments")
    @Help(displayHelpShort = "Provide comments", displayHelpLong = "You can add comments to the node to provide additional information")
    private String comments;

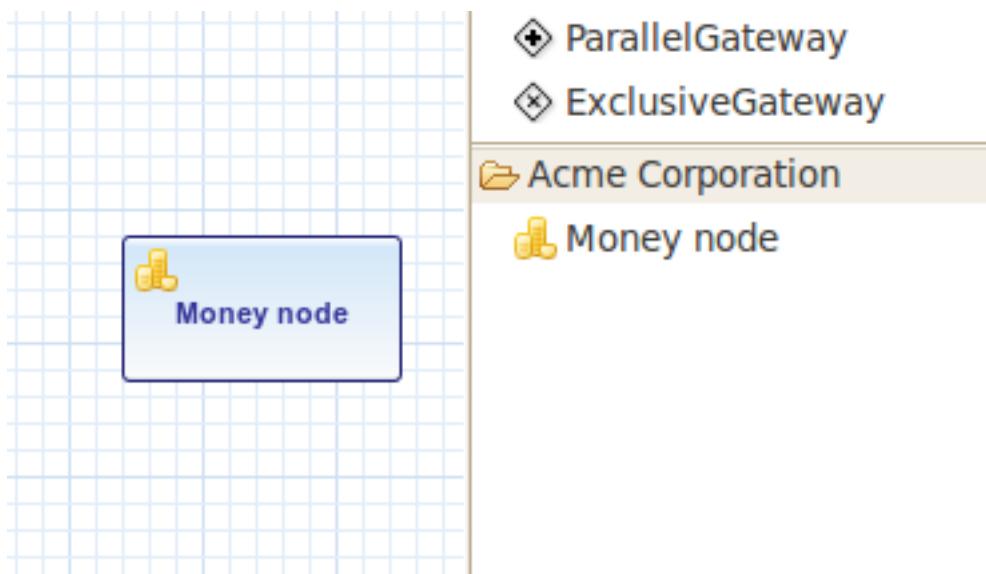
    /*
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.servicetask.AbstractCustomServiceTask#contributeToPaletteDrawer()
     */
    @Override
    public String contributeToPaletteDrawer() {
        return "Acme Corporation";
    }

    @Override
    public String getName() {
        return "Money node";
    }

    /*
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.servicetask.AbstractCustomServiceTask#getSmallIconPath()
     */
    @Override
    public String getSmallIconPath() {
        return "icons/coins.png";
    }
}

```

如果拿这个形状来扩展Activiti Designer, 画板和对应的节点看起来会如此:



money任务的属性窗口显示如下. 注意对于accountNumber 字段的必填信息.

Items Error Log

servicetask3

Money node

ount

ount using the account number specified

We need to review this procedure.

This field is required

Bob thinks it's better to pass the request to accounting first.

在使用过程变量的属性字段创建图表时, 用户可输入静态文本或用表达式. 如 (e.g. "This little piggy went to \${piggyLocation}"). 一般来说, 这适用于文本字段, 用户可以自由用于任何文本. 如果希望用户使用表达式然后你申请运行期的行为给你的CustomServiceTask (使用 @Runtime), 保在此种受拖类型中使用Expression 以便表达式在运行时能做正确解析. 运行时行为的更多信息, 可以在 this section中找到.

每个属性右边的按钮给出了对于字段的帮助. 点击按钮显示弹出框, 如下显示的.

ber specified

ew this procedure.

etter to pass the request to accounting first.

Help for

Provide a

Provide a
an account

12.5.1.3.1. #配置运行时执行自定义服务任务

你的字段设置及扩展适用于设计者, 用户在建模过程中可以配置服务任务的属性, 在大多数情况下, 你会在Activiti的执行进程时想要使用用户配置的属性. 要做到这一点, 当进程达到CustomServiceTask时, 必须指示出Activiti的其中的一类实例.

有一个特殊的注解来指定运行时CustomServiceTask的特点, @Runtime标注. 如例:

```
@Runtime(javaDelegateClass = "org.acme.runtime.AcmeMoneyJavaDelegation")
```

你的 CustomServiceTask 将使一个正常的ServiceTask在BPMN建模过程中输出。 Activiti的使several ways来定义ServiceTask运行时的特征。因此, @Runtime标注可以采取三个属性之一, 它可直接匹配Activiti的选项, 如下:

- javaDelegateClass映射到activiti:class 在BPMN输出。实现JavaDelegate一类的完全限定类名.
- expression映射到activiti:expression 在BPMN输出. 指定表达式的方法被执行, 就如Spring Bean的方法. 使用此选项时, 您应该您应该not 指定任何 @Property 注解字段. 欲知详情, 请参下文.
- javaDelegateExpression 映射到 activiti:delegateExpression 在BPMN中输出. 指定表达式JavaDelegate的类名.

如果你的成员提供Activiti的注入运行时类, 用户的属性值将被注入进去. 名称应与成员在CustomServiceTask里的名称相匹配, 欲了解更多信息, 请参考用户手册中的this part. 请注意, 因为5.11.0版本的设计者可以用Expression界面给动态字段值. 这意味着, 此设计属性的价值必须包含一个表达式, 这个表达式将在实施JavaDelegate 类别时被注入Expression

注意

你可在你CustomServiceTask的成员中使用@Property注解, 如果你使用@Runtime's expression属性运行, 将无法正常工作. 原因是Activiti会试图解决method 而不是一个类别. 因此, 任何成员打上@Property 都会被设计者忽略, 如果你使用的expression @Runtime标注. 设计师不会使它们作为节点的属性窗格中的可编辑字段, 在进程的BPMN的属性将不会产生输出.

注意

注意运行时类不应该在您的扩展JAR, 因为它依赖于Activiti库. Activiti需要在运行时能够找到它, 所以它需要在Activiti引擎的类路径.

设计师的源代码树中包含不同选择的项目配置@Runtime的例子。看一看money-tasks项目的一些出发点, 它涉及到以钱为代表的项目类别的例子.

12.5.1.4. #属性的类型

本节描述属性的类型, 通过CustomServiceTask的type 属性设置为一个 PropertyType值, 可以将其用于自定义服务任务.

12.5.1.4.1. #.PropertyType.TEXT

如下所示, 创建一个单行文本域. 可以是必填项, 其验证信息以提示框的方式显示. 通过将域的背景颜色改为淡红色来显示验证失败.



This field is required

12.5.1.4.2. #.PropertyType.MULTILINE_TEXT

如下所示, 创建多行文本域 (高度固定为80个像素). 可以是必填项, 其验证信息以提示框的方式显示. 通过将域的背景颜色改为淡红色来显示验证失败.

check the balance before approving this.



This field is required

12.5.1.4.3. #.PropertyType.PERIOD

创建结构化的编辑器, 使用微调控件编辑每个单位数量来确定一段时间. 结果如下所示. 可以是必填项 (理解为不是所有值都为0, 时间至少要有1部分是非零值), 其验证信息以提示框的方式显示. 通过将域的背景颜色改为淡红色来显示验证失败. 该字段的值是以格式为 1y 2mo 3w 4d 5h 6m 7s 的字符串进行存储的, 表示1年, 2个月, 3个星期, 4天, 5小时, 6分钟, 7秒. 整个字符串都将被存储, 即使有的部分是0.

y ,	<input type="text" value="4"/>	mo ,	<input type="text" value="0"/>	w ,	<input type="text" value="4"/>	d ,	<input type="text" value="0"/>	h ,	<input type="text" value="0"/>	m ,	<input type="text" value="0"/>
-----	--------------------------------	------	--------------------------------	-----	--------------------------------	-----	--------------------------------	-----	--------------------------------	-----	--------------------------------

y ,	<input type="text" value="0"/>	mo ,	<input type="text" value="0"/>	w ,	<input type="text" value="0"/>	d ,	<input type="text" value="0"/>	h ,	<input type="text" value="0"/>	m ,	<input type="text" value="0"/>
-----	--------------------------------	------	--------------------------------	-----	--------------------------------	-----	--------------------------------	-----	--------------------------------	-----	--------------------------------

12.5.1.4.4. #.PropertyType.BOOLEAN_CHOICE

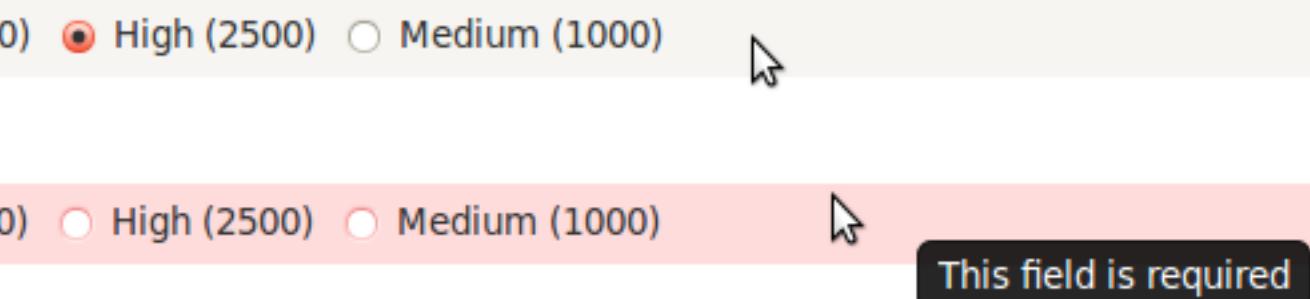
创建单个复选框来控制布尔逻辑或切换选择. 注意你可以指定Property注解的 required 属性, 但它不会被计算, 因为那会使用用户不得选择, 不管 是不是选中了选择框. 图形存储的值是 java.lang.Boolean.toString(boolean), 其结果是"true" 或"false".

12.5.1.4.5. #.PropertyType.RADIO_CHOICE

如下所示, 创建一组单选按钮. 选择任一单选按钮与选择任一其它单选按钮都是排斥的 (即, 只允许选择一个). 可以是必填项, 其验证信息以提示框的方式显示. 通过将这一组的背景颜色改为淡红色来显示验证失败.

这个属性类型要求注解的类的成员还要有 @PropertyItems 注解 (例子, 如下). 利用这个额外的注解, 你可以指定以字符串数组来提供的项目列表. 通过 为每个项目添加两个数组项来制定这些项目; 第一个是要显示的标题; 第二个是要存储的值.

```
@Property(type = PropertyType.RADIO_CHOICE, displayName = "Withdrawl limit", required = true)
@Help(displayHelpShort = "The maximum daily withdrawl amount ", displayHelpLong = "Choose the maximum daily amo
@PropertyItems({ LIMIT_LOW_LABEL, LIMIT_LOW_VALUE, LIMIT_MEDIUM_LABEL, LIMIT_MEDIUM_VALUE, LIMIT_HIGH_LABEL, LI
private String withdrawlLimit;
```

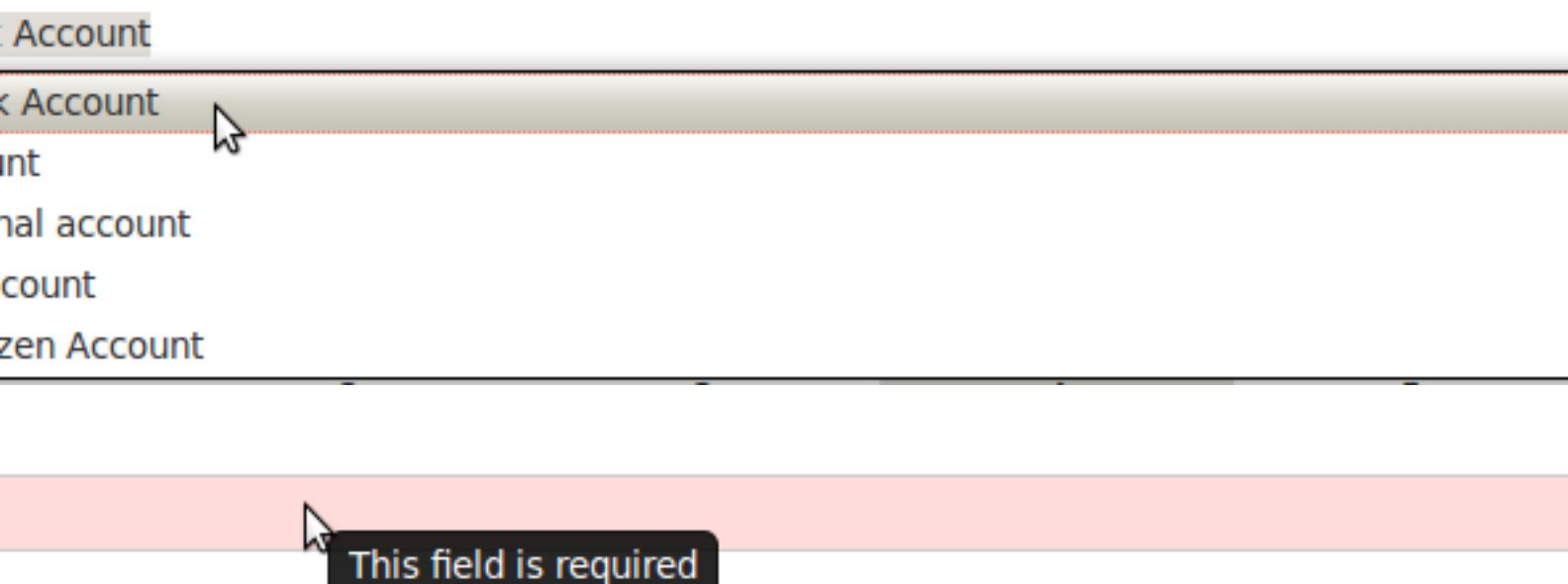


12.5.1.4.6. #.PropertyType.COMBOBOX_CHOICE

如下所示, 创建带有固定选项的下拉列表框. 可以是必填项, 其验证信息以提示框的方式显示. 通过将下拉列表框的背景颜色改为淡红色来显示验证失败.

这个属性类型要求被注解的类的成员还要有 @PropertyItems 注解 (例子, 如下). 利用这个额外的注解, 你可以指定以字符串数组来提供的项目列表. 通过 为每个项目添加两个数组项来制定这些项目; 第一个是要显示的标题; 第二个是要存储的值.

```
@Property(type = PropertyType.COMBOBOX_CHOICE, displayName = "Account type", required = true)
@Help(displayHelpShort = "The type of account", displayHelpLong = "Choose a type of account from the list of op
@PropertyItems({ ACCOUNT_TYPE_SAVINGS_LABEL, ACCOUNT_TYPE_SAVINGS_VALUE, ACCOUNT_TYPE_JUNIOR_LABEL, ACCOUNT_TYP
ACCOUNT_TYPE_JOINT_VALUE, ACCOUNT_TYPE_TRANSACTIONAL_LABEL, ACCOUNT_TYPE_TRANSACTIONAL_VALUE, ACCOUNT_TYPE_ST
ACCOUNT_TYPE_SENIOR_LABEL, ACCOUNT_TYPE_SENIOR_VALUE })
private String accountType;
```



12.5.1.4.7. #.PropertyType.DATE_PICKER

如下所示, 创建日期选择控件. 可以是必填项, 其验证信息以提示框的方式显示 (注意, 使用的控件自动设置为选中系统日期, 所以该值很少为空. 通过将此控件的 背景颜色改为淡红色来显示验证失败.)

这个属性类型要求注解的类成员还要有@DatePickerProperty 注解 (示例, 见下文). 利用这个额外的注解, 你可以指定图形中用于存储日期时间的模式, 以及你想要显示的日期选择器的类型. 这两个属性都是可选的, 并且如果不指定它们都会使用默认值 (它们是 DatePickerProperty 注解中的静态变量). dateTimePattern属性用来向 SimpleDateFormat 类提供模式. 在使用swtStyle 属性时, 必须指定SWT的 DateTime 控件所支持的一个整数, 因为正是使用这个控件来渲染这个属性类型的.

```
@Property(type = PropertyType.DATE_PICKER, displayName = "Expiry date", required = true)
@Help(displayHelpShort = "The date the account expires ", displayHelpLong = "Choose the date when the account will expire")
@DatePickerProperty(dateTimePattern = "MM-dd-yyyy", swtStyle = 32)
private String expiryDate;
```

Sun	Mon	Tue	Wed	Thu
5	26	27	28	29
2	3	4	5	6
9	10	11	12	13
6	17	18	19	20
3	24	25	26	27
0	31	1	2	3

12.5.1.4.8. #.PropertyType. DATA_GRID

如下所示, 创建数据表格控件. 数据表格允许用户输入任意多行数据, 并为每一行的一组固定列输入值 (每个行和列交叉处称为单元格) . 由用户来决定对行的添加、删除.

这个属性类型要求注解的类成员还要有 `@DataGridProperty` 注解 (示例, 见下文). 利用这个附加注解, 可以指定一些数据网格所特有的属性. 需要使用 `itemClass` 属性引用一个类来决定哪些列要加入到表格. Activiti Designer 希望成员变量是List类型. 习惯上, 可以将 `itemClass` 属性表示的类作为泛型类型来使用. 例如, 如果你要在表格中编辑一个购物单, 你会在 `GroceryListItem` 类上定义表格的列. 在你的 `CustomServiceTask` 内, 会想要它是像这样:

```
@Property(type = PropertyType.DATA_GRID, displayName = "Grocery List")
@DataGridProperty(itemClass = GroceryListItem.class)
private List<GroceryListItem> groceryList;
```

除了使用了数据表格, "itemClass"类使用的注解与你用来指定`CustomServiceTask`, 的域的注解是一样的. 具体的, 目前支持 `TEXT`, `MULTILINE_TEXT` 以及 `PERIOD` 你会注意到表格会为每个域创建一个单行的文本控件, 不管其 `.PropertyType`. 是什么类型. 这是为了保持表格的图形吸引力和可读性. 例如, 如果你想要以正常的显示模式显示 `PERIOD`类型的 `.PropertyType`, 你能想象的到在不搞乱屏幕的情况下它是永远也不会适合于 单元格的 (译注. 言外之意, 必然会搞乱屏幕), 对于 `MULTILINE_TEXT` 和 `PERIOD`, 添加在每个域上的双击机制都会弹出一个大的 `.PropertyType` 编辑器. 用户点击OK后, 值被存储到域内, 因此它能在表格中被看到.

必填属性的处理类似于处理 `TEXT` 类型的普通域, 整个表格在任意域失去焦点时 被校验. 如果校验失败, 数据表格中特定单元格的文本控件的背景颜色会变为浅红.

默认, 该组件允许用户添加行, 但不允许决定这些行的顺序. 如果想要对此允许, 必须将 `orderable` 属性设置为 `true` , 这就能让每行末尾的按钮在表格内上下移动.

注意

此刻, 这个属性类型还没被正确注入在运行时的类中.



12.5.1.5. #禁用画板中默认形状

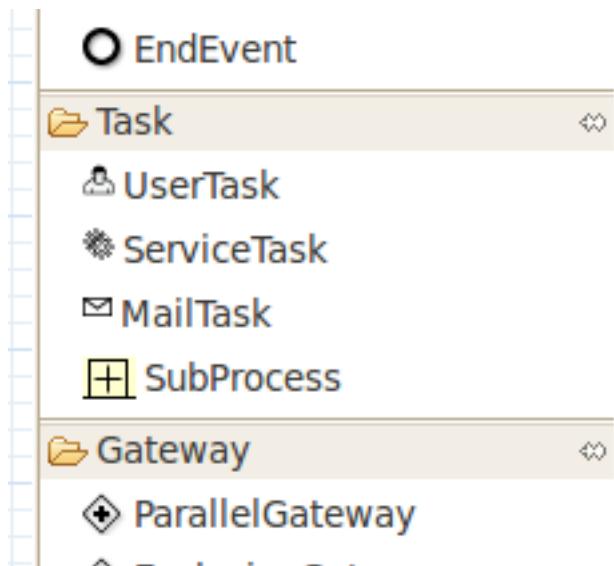
这个定制需要在扩展中包含一个实现了 `DefaultPaletteCustomizer` 接口的类. 不要直接实现这个接口, 需要创建 `AbstractDefaultPaletteCustomizer` 这个基类的子类. 目前, 这个类没提供任何功能, 但 `DefaultPaletteCustomizer` 接口今后的版本会提供更多的能力让这个基类拥有一些合理的默认行为, 所以最好继承, 这样你的 扩展会兼容将来的发布.

继承 AbstractDefaultPaletteCustomizer 类需要你实现方法 disablePaletteEntries(), 此方法必须返回一个 PaletteEntry 的列表. 对于每个默认的形状, 通过将它对应的 PaletteEntry 值添加到你的列表中 可以将其禁掉. 注意如果你移除了默认集合中的形状, 抽屉内没有剩余的形状了, 那么那个抽屉整个就会 从画板中被移除. 如果你希望禁掉所有默认形状, 只需将PaletteEntry.ALL 添加到你 的结果中. 如例子, 以下代码禁掉了画板中的手工任务和脚本任务形状.

```
public class MyPaletteCustomizer extends AbstractDefaultPaletteCustomizer {

    /*
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.palette.DefaultPaletteCustomizer#disablePaletteEntries()
     */
    @Override
    public List<PaletteEntry> disablePaletteEntries() {
        List<PaletteEntry> result = new ArrayList<PaletteEntry>();
        result.add(PaletteEntry.MANUAL_TASK);
        result.add(PaletteEntry.SCRIPT_TASK);
        return result;
    }
}
```

应用这个扩展的结果显示为如下图片. 正如你看到的, 手工任务和脚本任务形状不再在 Tasks 抽屉中了.



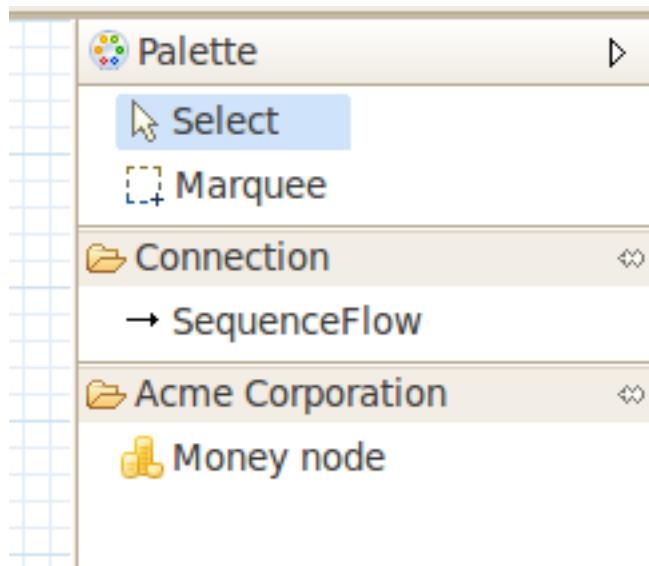
要想禁掉所有默认的形状, 可以使用类似于如下的代码.

```
public class MyPaletteCustomizer extends AbstractDefaultPaletteCustomizer {

    /*
     * (non-Javadoc)
     *
     * @see org.activiti.designer.integration.palette.DefaultPaletteCustomizer#disablePaletteEntries()
     */
    @Override
    public List<PaletteEntry> disablePaletteEntries() {
        List<PaletteEntry> result = new ArrayList<PaletteEntry>();
        result.add(PaletteEntry.ALL);
        return result;
    }
}
```



结果看起来就像这样（注意默认形状的那些抽屉不再显示在画板中了）：



12.5.2. #校验图形和导出到自定义的输出格式

除了定制画板，你还可以给Activiti Designer创建能执行校验、将来自图形的信息保存到Eclipse工作空间内的自定义资源的扩展。对此有内置扩展点，本节说明如何使用这些扩展点。

注意

最近重新使用了ExportMarshaller。我们仍然在开发校验相关的功能。下面的文档介绍的是老方案，一旦新功能完成，就会更新。

Activiti Designer允许编写校验图形的扩展。默认工具中已经存在BPMN 构造的校验了，但如果你想要对更多项进行校验，比如建模约定或 CustomServiceTask属性中的值，这时你就可以添加自己的校验了。这样的扩展被称为Process Validators（流程校验器）。

你也可以让Activiti Designer在保存图形时公布一些额外的格式。这样的扩展被称为 Export Marshallsers（输出装配器），每次用户保存时都会由Activiti Designer 自动调用。通过在Eclipse 设置对话框来启用或禁用这一个行为，这会对扩展进行检测。Designer会确保在保存图片时，根据用户的配置来调用ExportMarshaller。

通常，会想将ProcessValidator和ExportMarshaller 进行联合。假如你有几个CustomServiceTask在用，其属性你想要在生成的流程中使用。然而，在流程产生出来之前，你想首先校验其中一些值。结合ProcessValidator 和 ExportMarshaller是完成这的最好方法，Activiti Designer 能够 将你的扩展无缝隙地插入到这一工具内。

要创建 ProcessValidator和 ExportMarshaller，需要创建与扩展画板所不同的扩展。原因很简单：在你的代码中，需要访问更多由集成类库提供的API。特别是，你会使用Eclipse本身中的类。所以要开始，必须先创建一个Eclipse插件（利用 Eclipse对PDE的支持来完成），然后将它打包在一个自定义的Eclipse产品或特性中。解释 开发Eclipse插件的所涉及的所有详细信息超出了本用户指南的范畴，所以以下说明只局限于 扩展Activiti Designer功能。

你的模块必须依赖以下类库：

- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.activiti.designer.eclipse
- org.activiti.designer.libs
- org.activiti.designer.util

另外，Designer中也可以使用org.apache.commons.lang，你可以根据需要在自己的扩展中使用。

ProcessValidator和ExportMarshaller都是通过继承基础类来创建的。这些基础类从其超类AbstractDiagramWorker那里继承了一些很有用的方法。利用这些方法可以创建显示在Eclipse中problems视图内帮助用户找出问题或要点的消息、警告以及错误标记。也可以通过Resources和InputStreams获得图形的内容。这些信息保存在DiagramWorkerContext，由AbstractDiagramWorker提供。

把调用clearMarkers()作为在ProcessValidator或ExportMarshaller内的第一件要做的事可能会是个好主意；这将清理掉所有之前操作人员的标记（标记自动链接到操作人员，清除一个操作人员的标记不会影响到其它标记）。例如：

```
// Clear markers for this diagram first
clearMarkersForDiagram();
```

你还应该利用进度监控将进度情况报告给用户（在DiagramWorkerContext中），因为校验和/或装配行为都会占用一些时间，在此期间用户被迫等待。报告进度的情况需要一些关于如何使用Eclipse特性的知识。仔细看一下这篇 [<http://www.eclipse.org/articles/Article-Progress-Monitors/article.html>] 深入解释了概念和用法的文章。

12.5.2.1. #创建ProcessValidator扩展

注意

需要复审！

在plugin.xml文件内给org.activiti.designer.eclipse.extension.validation.ProcessValidator扩展点创建扩展。需要为此扩展点创建AbstractProcessValidator类的子类。

```
<?eclipse version="3.6"?>
<plugin>
  <extension
    point="org.activiti.designer.eclipse.extension.validation.ProcessValidator">
    <ProcessValidator
      class="org.acme.validation.AcmeProcessValidator">
      </ProcessValidator>
    </extension>
  </plugin>
```

```
public class AcmeProcessValidator extends AbstractProcessValidator {
```

必须实现几个方法。最重要的是，实现getValidatorId()，为你的校验器返回一个全局唯一的ID。这让你能在ExportMarshaller内对其进行调用，甚至可以让其他人在他们的ExportMarshaller内调

用你的校验器. 实现 `getValidatorName()`, 返回校验器的逻辑名. 这个名称在对话框中显示给用户. 在`getFormatName()`中, 可以返回校验器通常校验的图形的类型.

校验本身是在`validateDiagram()`方法内进行的. 以此来看, 根据你的具体功能在这里放什么样的代码. 但通常在开始你会取得图形的流程节点, 这样你就可以循环遍历它们, 收集、对比并校验数据. 本代码片段展示给你如何进行这些操作:

```
final EList<EObject> contents = getResourceForDiagram(diagram).getContents();
for (final EObject object : contents) {
    if (object instanceof StartEvent ) {
        // Perform some validations for StartEvents
    }
    // Other node types and validations
}
```

在你完成校验后, 不要忘记调用`addProblemToDiagram()`和/或 `addWarningToDiagram()`, 等等. 务必最后返回正确的布尔类型的结果值来表明 你考虑的校验是成功还是失败. 这个结果可以被调用`ExportMarshaller` 使用来决定下一步操作.

12.5.2.2. #创建ExportMarshaller扩展

在`plugin.xml`文件内给 `org.activiti.designer.eclipse.extension.ExportMarshaller` 扩展点创建扩展. 需要为此扩展点创建`AbstractExportMarshaller` 类的子类. 这个抽象基类提供了在编组到你自己格式时所使用的几个有用的方法, 但最重要的是它允许你 将资源保存到工作空间内, 还允许调用校验器.

在Designer的`examples`目录下有一个示例。这个例子演示了如何使用基础类中的方法实现基本功能, 比如访问图形的`InputStream`, 使用`BpmnModel`, 并将资源保存到工作区中。

```
<?eclipse version="3.6"?>
<plugin>
    <extension
        point="org.activiti.designer.eclipse.extension.ExportMarshaller">
        <ExportMarshaller
            class="org.acme.export.AcmeExportMarshaller">
        </ExportMarshaller>
    </extension>
</plugin>
```

```
public class AcmeExportMarshaller extends AbstractExportMarshaller {
```

需要你来实现某些方法, 如`getMarshallerName()`及`getFormatName()`. 这些方法用来向用户显示选项以及在进度对话框中显示信息, 因此要确保你所描述的能表达出你实现的功能.

大部分你的工作是在 `doMarshalDiagram()`方法完成的。

如果想先执行某个校验, 可以直接在你的装配器内调用校验器. 从校验器接收到一个布尔类型的结果值, 这样你就能知道校验是否成功. 大多数情况下, 如果校验无效是不会对图形进行编组的, 但你也可以选择继续, 甚至校验失败后, 创建一个不同的资源。

一旦获得了所有你所需要的数据, 就可以调用`saveResource()`方法来创建包含着 你的数据的文件了. 可以在单个`ExportMarshaller`内多次调用`saveResource()`, 所以装配器可用来创建多个输出文件.

利用AbstractDiagramWorker类中saveResource()方法可以给输出的资源构造文件名。有几个你已经解析过了的有用的变量，允许你来创建像_original-filename_my-format-name.xml 的文件名。 Javadocs 内有对这些变量的描述，ExportMarshaller接口中也做了定义。 你也可以使用resolvePlaceholders()处理一个string（比如，一个路径），如果你想自己解析占位符。 getURIRelativeToDiagram()会帮你做这些事情。

你应该利用进度监控将进度情况报告给用户这篇文章 [<http://www.eclipse.org/articles/Article-Progress-Monitors/article.html>] 描述了如何来完成.

第#13#章#Activiti Explorer

Activiti Explorer，我习惯称之为Activiti控制台，后面也这么翻译。Activiti控制台是一个web应用程序，当我们从Activiti的官方网站下载Activiti的压缩zip文件时候，Activiti控制台在\${Activiti_home}/wars文件夹下面。该控制台的目的并不是创建一个完善的web应用程序，仅仅是为客户端用户准备的应用程序，但是却能够练习和展示Activiti的功能。正如这样，控制台仅仅只是一个Demo，可能有人会使用该控制台集成到他们自己的系统之中。另外，对于该控制台，我们使用了一个内存数据库，也很容易换成你自己的数据库（查看WEB-INF文件夹下面的applicationContext.xml文件）。

随后，登录进该控制台，你将会看见四个比较大的图标按钮用于显示主要功能。



- Tasks: 任务管理功能。这里，如果你是办理人，你可以看见运行中流程实例的自己的待办任务，或者你可以拾取组任务。控制台涉及的功能，子任务的工作，不同角色的人，等等... 控制台也可以允许创建一个独立的任务，该任务并没有关联任何流程实例。
- Process: 显示部署的流程定义列表，并且可以启动一个新的流程实例。
- Reporting: 生成报表和显示之前保存历史的结果数据。查看报表这一节可以获取更多的信息。
- Manage: 当登录的用户具有超级管理员权限才能够看见。用于管理Activiti的流程引擎：管理用于和组，执行和查看停止的jobs，查看数据库和部署新的流程定义。

13. 1. #流程图

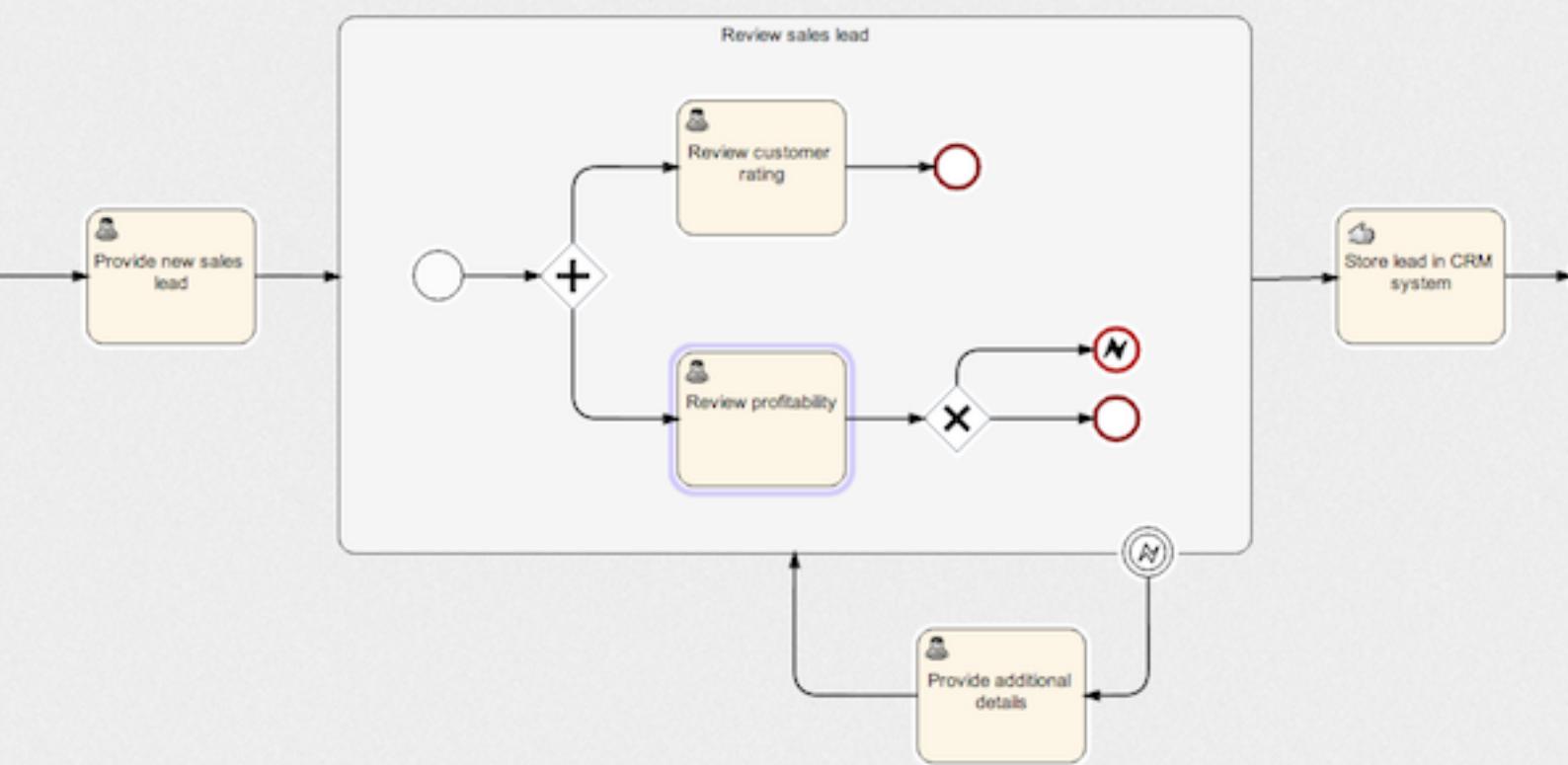
控制台包含的功能，使用Raphaël [http://raphaeljs.com/] Javascript框架自动生成一张流程图。当流程定义XML包含的BPMN注入信息时。该流程图才能够生成。当流程定义XML中并没有BPMN注入信息但是部署的时候包含一张流程图，那么该图片也将会被显示。

New sales lead

Deployed moments ago

completel

36 >



当你并不想使用Javascript生成流程图，你可以在ui.properties文件中禁用它。

```
activiti.ui.jsdiagram = false
```

除了在控制台上面显示流程图，控制台也会很容易的包含你想要查看的流程图。下面的URL将会显示流程定义图片，根据留存定义的ID：

```
http://localhost:8080/activiti-explorer/diagram-viewer/index.html?processDefinitionId=reviewSaledLead:1:36
```

它也可以显示当前流程实例的状态，通过添加一个processInstanceId的请求参数，如下：

```
http://localhost:8080/activiti-explorer/diagram-viewer/index.html?processDefinitionId=reviewSaledLead:1:36&proce
```

13. 2. #任务

The screenshot shows the Activiti Explorer application window. At the top, there's a blue header bar with the Activiti Explorer logo on the left and a 'Tasks' button with a document icon on the right. Below the header is a navigation bar with five items: 'Inbox 1', 'My Tasks 0', 'Queued 0', 'Involved 0', and 'Archived 0'. The 'Inbox' item has a blue background and white text, indicating it is the active tab.

- Inbox: 显示登录用户需要办理的所有任务列表。
- My tasks: 显示登录用户任务拥有者的任务列表。当你创建一个独立的任务，你可以自动化操作该任务。
- Queued: 显示不用的组任务列表，并且登录用户在该组中。这里的所有任务都必须先拾取然后才能够完成。
- Involved: 显示登录用户被参与的任务（即不是办理人和任务拥有者）。
- 归档包含已经完成的（历史的）任务。

13. 3. #启动流程实例

在流程定义选项卡，允许你查看Activiti流程引擎部署的所有流程定义。你可以使用页面顶部右边的按钮启动一个新的流程实例。如果该流程定义有一个启动表单，那么在启动流程实例之前就先显示表单。

ed process definitions

Model workspace

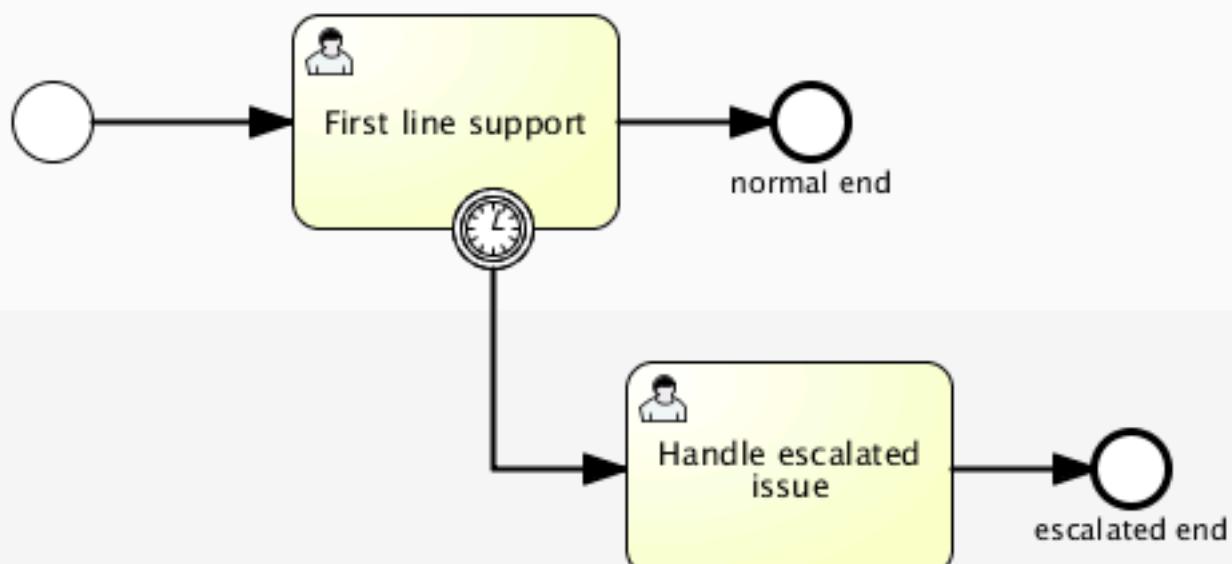
Start pro



Helpdesk process

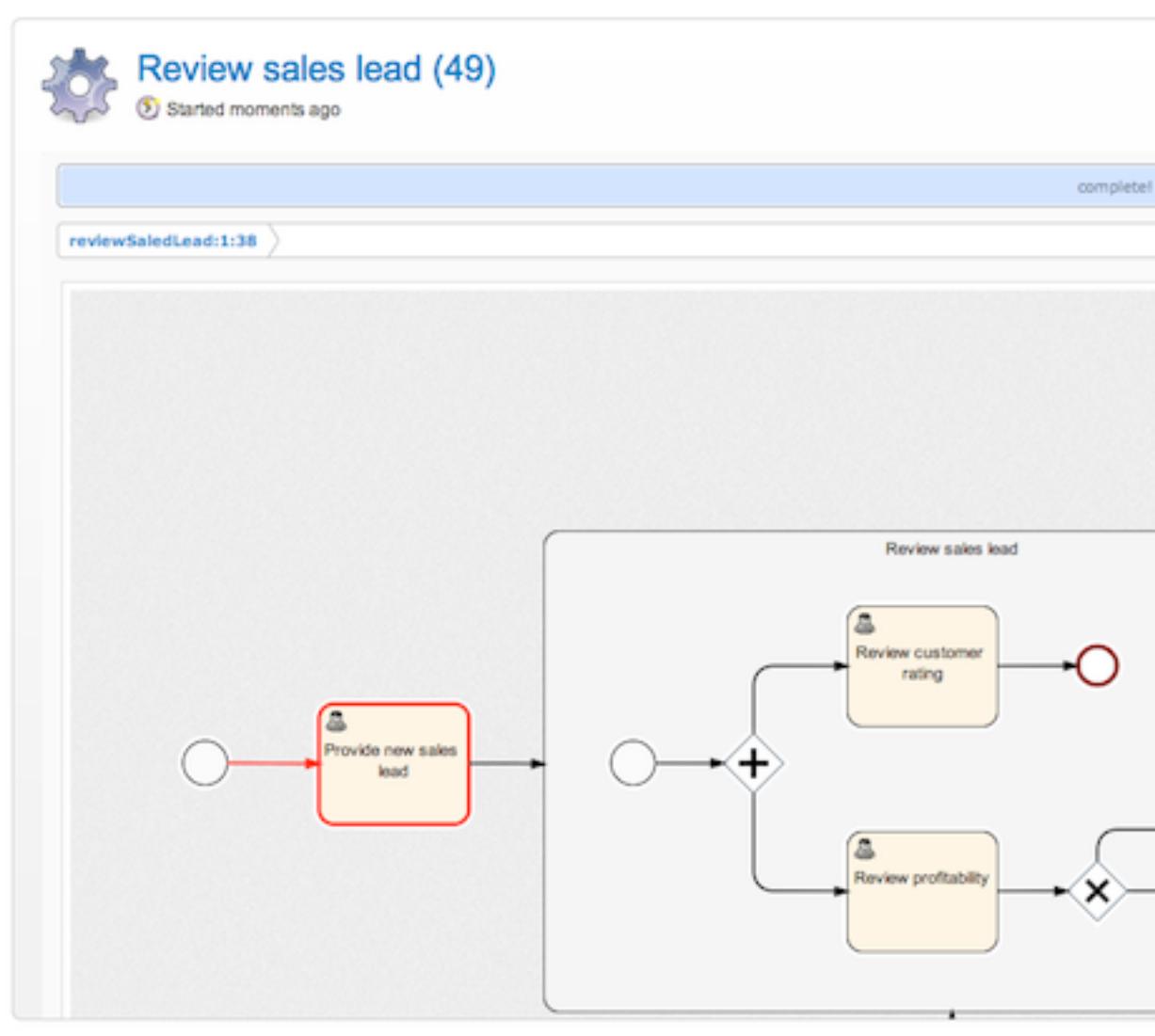
 Version 1  Deployed 5 minutes ago

Process Diagram



13. 4. #我的流程实例

在我的流程 选项卡，显示当前登录用户未完成的用户任务的所有流程实例。这也很直观的显示了流程实例的当前活动和存储的流程变量。



13. 5. #管理

在管理功能中，只有当登录用户只权限组admin中的成员时，该功能才会显示。当点击 Manage 图标按钮，提供以下选项列表

- 数据库：在数据库中显示Activiti有关内容. 当开发流程或者排除故障等问题的时候是非常有用的。

ACT_HI_DETAIL (8)

ACT_HI_PROCINST (5)

ACT_HI_TASKINST (12)

ACT_ID_GROUP (8)

ACT_ID_INFO (6)

ACT_ID_MEMBERSHIP (12)

ACT_ID_USER (3)

ACT_RE_DEPLOYMENT (2)

ACT_RE_PROCDEF (6)

ACT_RU_EXECUTION (8)

ACT_RU_IDENTITYLINK (2)

ACT_RU_JOB (1)

ACT_RU_TASK (8)

ACT_RU_VARIABLE (11)

 ACT_RU_VARIABLE

ID_	REV_	TYPE_	NAME_	EXECUTION_ID_
145	1	string	initiator	144
149	1	string	initiator	148
155	1	string	customerName	148
156	1	null	potentialProfit	148
157	1	string	details	148
169	1	string	employeeName	168
173	1	long	numberOfDays	168

- 部署： 显示当前流程引擎的部署，并且可以看见部署的内容（流程定义，流程图，业务规则，等等...）

atabase

Deployments

Jobs

Users

Groups

activiti-engine-examples.bar

myProcess.bpmn20.xml

**activiti-engine-examples** Deployed 2 hours ago

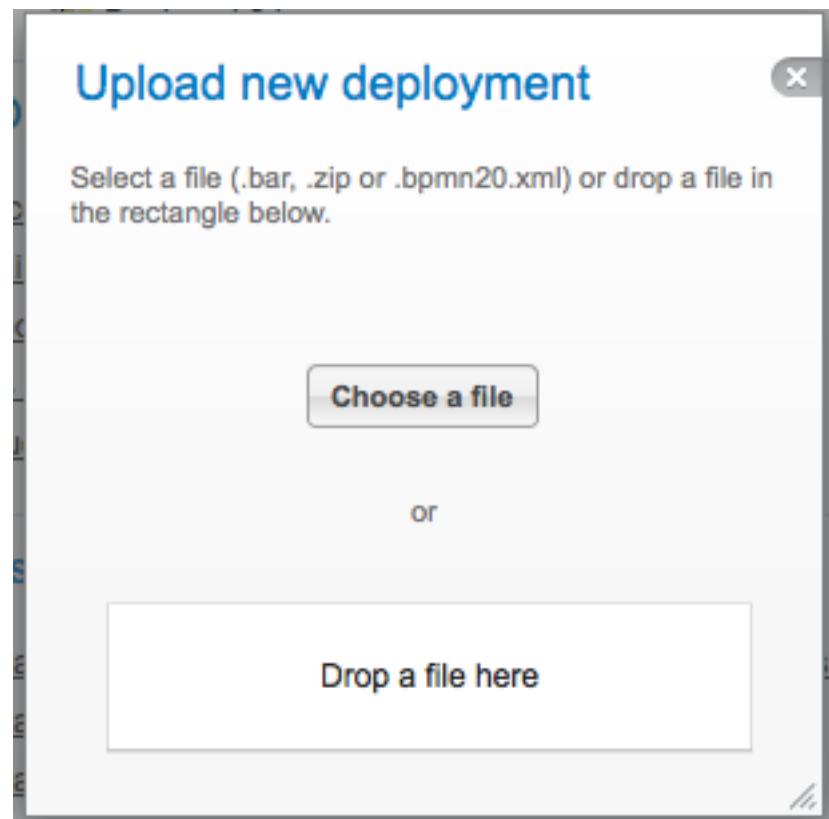
Process Definitions

[Expense process](#)[Fix system failure](#)[Helpdesk process](#)[Review sales lead](#)[Vacation request](#)

Resources

[org/activiti/examples/adhoc/Expense_process.adhoc_Expense](#)[org/activiti/examples/adhoc/Expense_process.bpmn20.xml](#)[org/activiti/examples/bpmn/event/error/reviewSalesLead.bpmn](#)

当你点击 部署 按钮时，你也可以上传新的部署。从自己的计算机中选择一个业务文档或者一个BPMN20.XML文件，或者简单的拖拽到指定的区域就可以部署一个新的业务流程。



- Jobs（作业）：在左边显示当前的作业（定时器，等等）并且运行手动执行他们（例如在截止时间之前触发定时器）。如果作业执行失败（例如邮件服务器不能正常工作），那么就会显示所有的异常。

The interface shows a navigation bar with tabs: Database, Deployments, **Jobs**, Users, Groups. On the left, a sidebar lists three timer jobs: Timer job 215 (highlighted), Timer job 191, and Timer job 203. On the right, a detailed view for Timer job 215 is shown, featuring a green icon with a white greater-than sign, the job name, its due time ('Due 4 minutes from now'), and a status message ('Job hasn't been executed yet.').

- 用户和组 管理用户和组：创建，修改和删除用户和组。关联用户和组，这样他们就会有更多的权限或者他们能够看见在任务分配给特地的组。


Kermit the Frog

Details



Id: kermit

First name: Kermit

Last name: the Frog

Email: kermit@localhost

Groups

ID	NAME	TYPE
accountancy	Accountancy	ass

13. 6. #报表

控制台附带了一些报表例子并且有能力很轻松的在系统中添加新的报表。 报表功能是位于主功能中的' 报表' 按钮。



重要: 如果要让报表工作, 控制台需要配置历史的级别不能够没有。这默认的配置是满足这一要求的。

目前, 该报表选项卡会显示2个子选项卡:

- 生成报表: 显示系统中已知的报表列表。允许运行生成的报表。

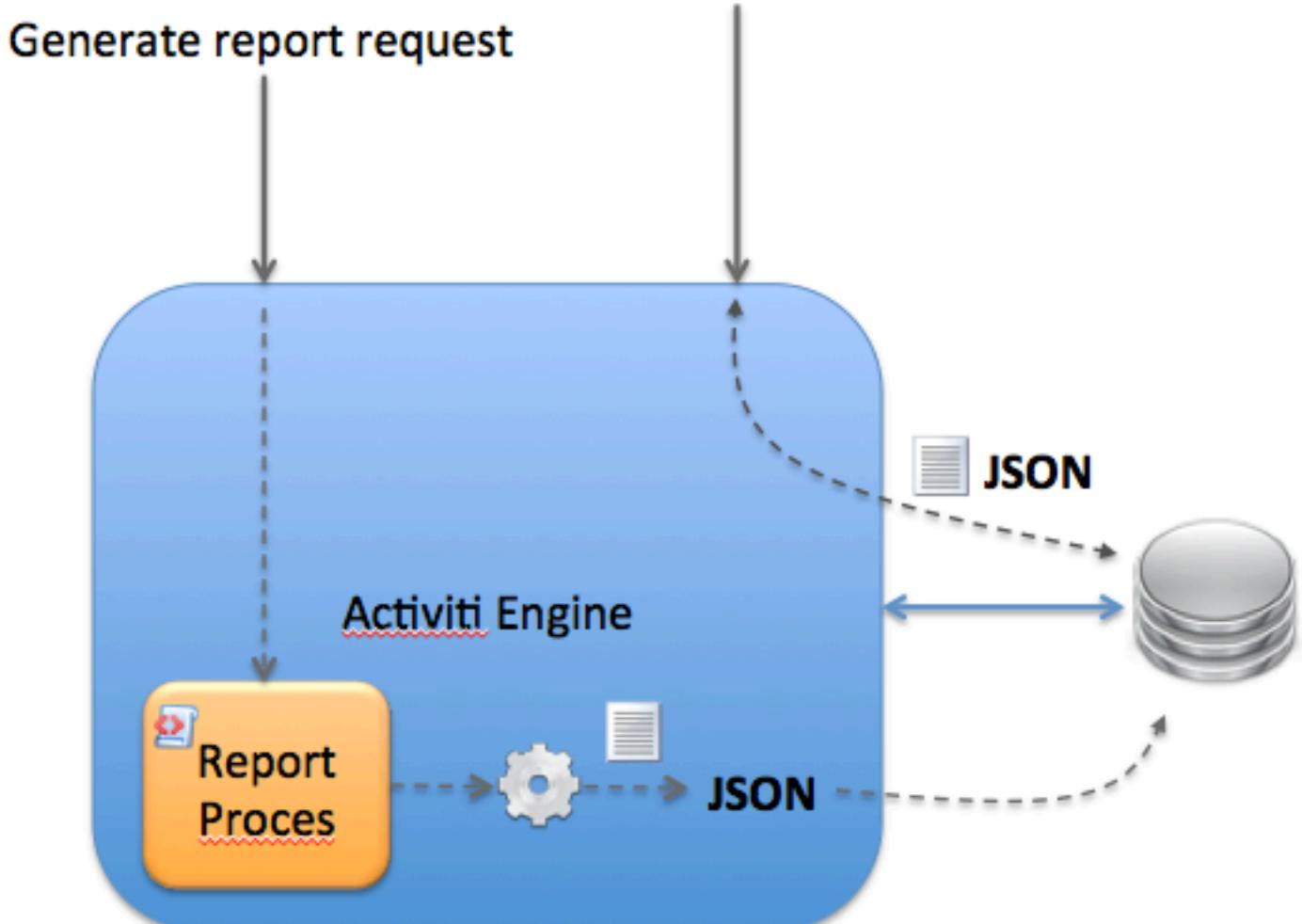
- 保存报表： 显示之前保存的所有报表列表。注意，这里仅仅显示的是个人保存的报表，并且不能看见其他人保存的报表。

流程的数据被用于生成报表中的列表和图标。第一次看上去可能会很奇怪，使用流程生成报表数据有几个优势。

- 该流程能够直接访问Activiti流程引擎的内部。他直接可以使用流程引擎访问数据库。
- 作业执行器能够用于任何其他的流程。这意味着你能够异步生存流程或者仅仅异步执行某些步骤。这也意味着你可以使用定时器，例如在某些时间点上面生成报表数据。
- 可以用已知的工具和已知的概念创建一个新的报表。同时，没有新的概念，服务或者应用被需要。部署或者上传一个新的报表与部署一个新的流程是一样的。
- 它可以使用BPMN2.0结构。这意味着所有的东西，比如并行网关，可以实现基于数据或用户请求输入生成分支。

生成报表数据的流程定义需要把' activiti-report' 设置为分类，这样就能在Explorer的报表列表中显示出来。报表流程可繁可简。能够看到报表的唯一要求是，流程会创建一个名为reportData的流程变量。这个变量必须是json对象的二进制数组。这个变量必须保存到Activiti的历史表中（所以要求引擎必须启用历史功能）所以可以在后面报表保存时获取。

Get saved report request



13.6.1. #报告数据JSON

报表流程必须生成一个变量reportData，这是一个要展现给用户的JSON数据。这个json看起来像这样：

```
{
  "title": "My Report",
  "datasets": [
    {
      "type" : "lineChart",
      "description" : "My first chart",
      "xaxis" : "Year",
      "yaxis" : "Total sales"
      "data" :
      {
        "2010" : 50,
        "2011" : 33,
        "2012" : 17,
        "2013" : 87,
      }
    }
  ]
}
```

json数据会在Explorer中获取，并用来生成图表或列表。 json的元素为：

- title: 这个报表的标题
- datasets: 是数据集的数组，对应报表中不同的图表和列表。
- type每个数据集都有一个类型。这个类型会用来决定如何渲染数据。当前支持的值有： pieChart, lineChart, barChart 和 list.
- description: 每个图表可以在报表中显示一个可选的描述。
- x- 和 yaxis: 只对 lineChart类型起作用。这个可选参数可以修改图表坐标系的名称。
- data: 这是实际的数据。数据是一个key-value格式的json对象。

13.6.2. #实例流程

下面的例子演示了一个“流程实例总览”报表。流程本身非常简单，只包含一个脚本任务（除了开始和结束）使用javascript生成json数据集。虽然所有Explorer中的例子都使用javascript，它们也可以使用java服务任务。执行流程最后的结果就是reportData变量，保存着数据。

重要提示：下面的例子只能运行在JDK 7+环境中。因为使用了javascript引擎（Rhino），如果运行在老JDK版本中会无法实现一些结果，来像下面一样编写脚本。参考下面的一个Jdk 6+兼容的例子。

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn"
  xmlns:bpmdi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DO"
  xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema"
```

```

expressionLanguage="http://www.w3.org/1999/XPath"
targetNamespace="activiti-report"

<process id="process-instance-overview-report" name="Process Instance Overview" isExecutable="true">

    <startEvent id="startevent1" name="Start" />
    <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="generateDataset" />

    <scriptTask id="generateDataset" name="Execute script" scriptFormat="JavaScript" activiti:autoStoreVariables="true">
        <script><![CDATA[

            importPackage(java.sql);
            importPackage(java.lang);
            importPackage(org.activiti.explorer.reporting);

            var result = ReportingUtil.executeSelectSqlQuery("SELECT PD.NAME_, PD.VERSION_ , count(*) FROM ACT_HI_PROCINST PD WHERE PD.PROCESS_INSTANCE_ID = ?");

            var reportData = {};
            reportData.datasets = [];

            var dataset = {};
            dataset.type = "pieChart";
            dataset.description = "Process instance overview (" + new java.util.Date() + ")";
            dataset.data = {};

            while (result.next()) { // process results one row at a time
                var name = result.getString(1);
                var version = result.getLong(2);
                var count = result.getLong(3);
                dataset.data[name + " (" + version + ")"] = count;
            }
            reportData.datasets.push(dataset);

            execution.setVariable("reportData", new java.lang.String(JSON.stringify(reportData)).getBytes("UTF-8"));
        ]]></script>
    </scriptTask>
    <sequenceFlow id="flow3" sourceRef="generateDataset" targetRef="theEnd" />

    <endEvent id="theEnd" />

</process>

</definitions>

```

除了流程xml顶部的标准xml，主要区别是targetNamespace设置为 activiti-report， 把分类设置为与部署的流程定义一样的名称。

脚本的第一行只是进行一些导入，避免每次使用时，都要写包名。 第一个有意义的代码是使用ReportingUtil读取activiti数据库。 返回结果是一个JDBC 结果集。 查询语句下面， javascript创建了使用的json。 json是符合上面描述的需求的。

最后一行脚本有一点儿奇怪。首先我们需要吧json对象转换成字符串， 使用javascript函数JSON.stringify()。 字符串需要保存为二进制数组类型的变量。这是一个技术问题： 二进制数组的大小是无限的，但是字符串的长度有限制。这就是为什么javascript字符串 必须转换成一个java字符串，以获得转换成二进制的功能。

兼容JDK 6（以及更高版本）的同一个流程有一些区别。 原生json功能无法使用，因此提供了一些帮助类（ReportData 和 Dataset）：

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn"
  xmlns:bpmdi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DO"
  xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema"
  expressionLanguage="http://www.w3.org/1999/XPath"
  targetNamespace="activiti-report">

  <process id="process-instance-overview-report" name="Process Instance Overview" isExecutable="true">

    <startEvent id="startevent1" name="Start" />
    <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="generateDataset" />

    <scriptTask id="generateDataset" name="Execute script" scriptFormat="js" activiti:autoStoreVariables="false">
      <script><![CDATA[

        importPackage(java.sql);
        importPackage(java.lang);
        importPackage(org.activiti.explorer.reporting);

        var result = ReportingUtil.executeSelectSqlQuery("SELECT PD.NAME_, PD.VERSION_ , count(*) FROM ACT_HI_PROCINST PD WHERE PD.PROCESS_INSTANCE_ID = ? AND PD.END_TIME IS NOT NULL ORDER BY PD.VERSION_ DESC LIMIT 1");
        var reportData = new ReportData();
        var dataset = reportData.newDataset();
        dataset.type = "pieChart";
        dataset.description = "Process instance overview (" + new java.util.Date() + ")"

        while (result.next()) { // process results one row at a time
          var name = result.getString(1);
          var version = result.getLong(2);
          var count = result.getLong(3);
          dataset.add(name + " (" + version + ")", count);
        }

        execution.setVariable("reportData", reportData.toBytes());
      ]]></script>
    </scriptTask>
    <sequenceFlow id="flow3" sourceRef="generateDataset" targetRef="theEnd" />

    <endEvent id="theEnd" />

  </process>
</definitions>

```

13.6.3. #报告开始表单

因为报表是使用普通流程来生成的，所以表单功能也可以使用。直接在开始事件里加一个开始表单，Explorer就会在生成报表之前 把它展示给用户。

```
<startEvent id="startevent1" name="Start">
```

```

<extensionElements>
    <activiti:formProperty id="processDefinition" name="Select process definition" type="processDefinition" required="true">
        <activiti:formProperty id="chartType" name="Chart type" type="enum" required="true">
            <activiti:value id="pieChart" name="Pie chart" />
            <activiti:value id="barChart" name="Bar chart" />
        </activiti:formProperty>
    </extensionElements>
</startEvent>

```

这会为用户渲染一个普通的表单：

The screenshot shows the Activiti Explorer interface. At the top, there's a title bar with the application name. Below it is a code snippet showing XML configuration for a start event. The main area contains a form titled "Task duration report". The form has two dropdown menus: one for "Select process definition" set to "Create timers process (v1)" and another for "Chart type" set to "Pie chart". At the bottom of the form is a prominent blue "Generate report" button.

表单属性会在启动流程时提交，然后它们就可以像普通的流程变量一样使用，脚本中可以使用它们来生成数据：

```
var processDefinition = execution.getVariable("processDefinition");
```

13. 6. 4. #流程例子

对于默认的，控制台包含4个报表例子：

- Employee productivity(员工的工作效率)： 报表演示使用折线图和开始表单。 报表的脚本也比其他例子要复杂，因为数据会在脚本中先进行解释，再保存到报表数据中。
- Helpdesk (一线与升级)： 使用饼图进行展示，结合两个不同的数据库查询结果。
- Process instance overview (流程实例总览)： 使用多个数据集的报表实例。 报表包含使用相同数据的饼图和列表视图，展示多种数据集可以用来在一个页面中生成不同图表。
- Task duration (任务持续时间)： 另一个使用开始表单的例子，会使用对应的变量来动态生成SQL查询语句。

13. 7. #修改数据库

如果修改控制台例子所用的数据库，改变属性文件apps/apache-tomcat-6.x/webapps/activiti-explorer/WEB-INF/classes/db.properties。 同样，在类路径下放上合适的数据库驱动（Tomcat 共享类库或者在 apps/apache-tomcat-6.x/webapps/activiti-explorer/WEB-INF/lib/中）

第#14#章#Activiti Modeler

Activiti Modeler是一个BPMN web建模组件，它是Activiti Explorer web应用的一部分。Modeler是Signavio核心组件 [<http://code.google.com/p/signavio-core-components/>]项目的一个分支。新版Activiti Modeler的初始开发是由KIS BPM [<http://kisbpn.com>]捐献给Activiti项目的。和之前Activiti Modeler (Signavio核心组件)主要的区别是新Modeler是作为Activiti项目的一部分来维护和开发的。Activiti Modeler的目标是支持所有BPMN元素和Activiti引擎支持的扩展。

当你运行Activiti Explorer使用默认配置时，模型工作台中会有一个示例流程。



Tasks



Processes



Manage

Model workspace

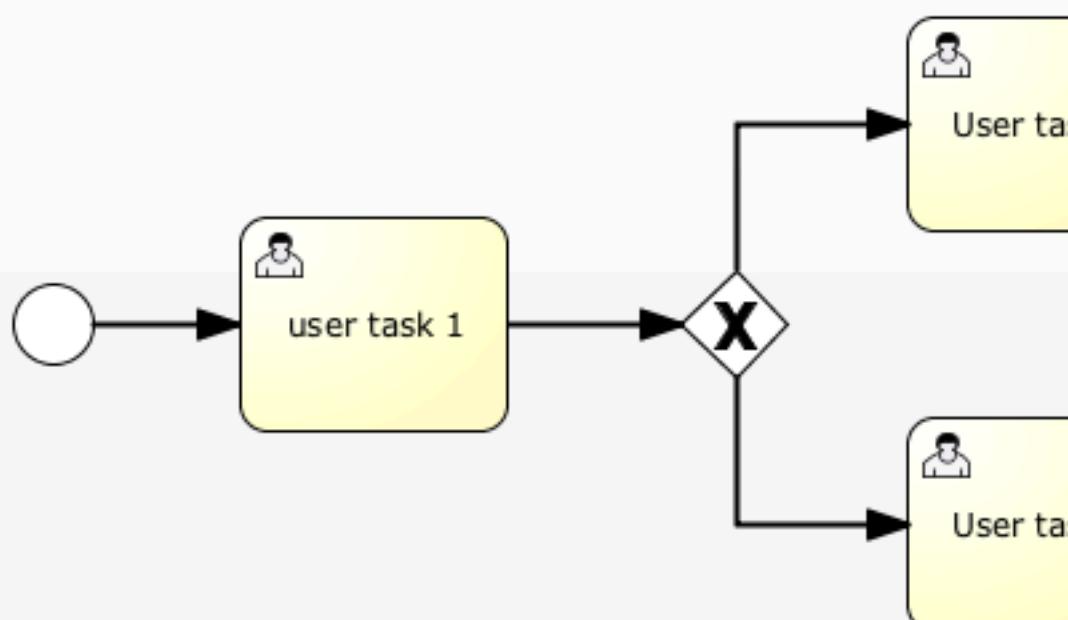
New model



Demo model

Version 1

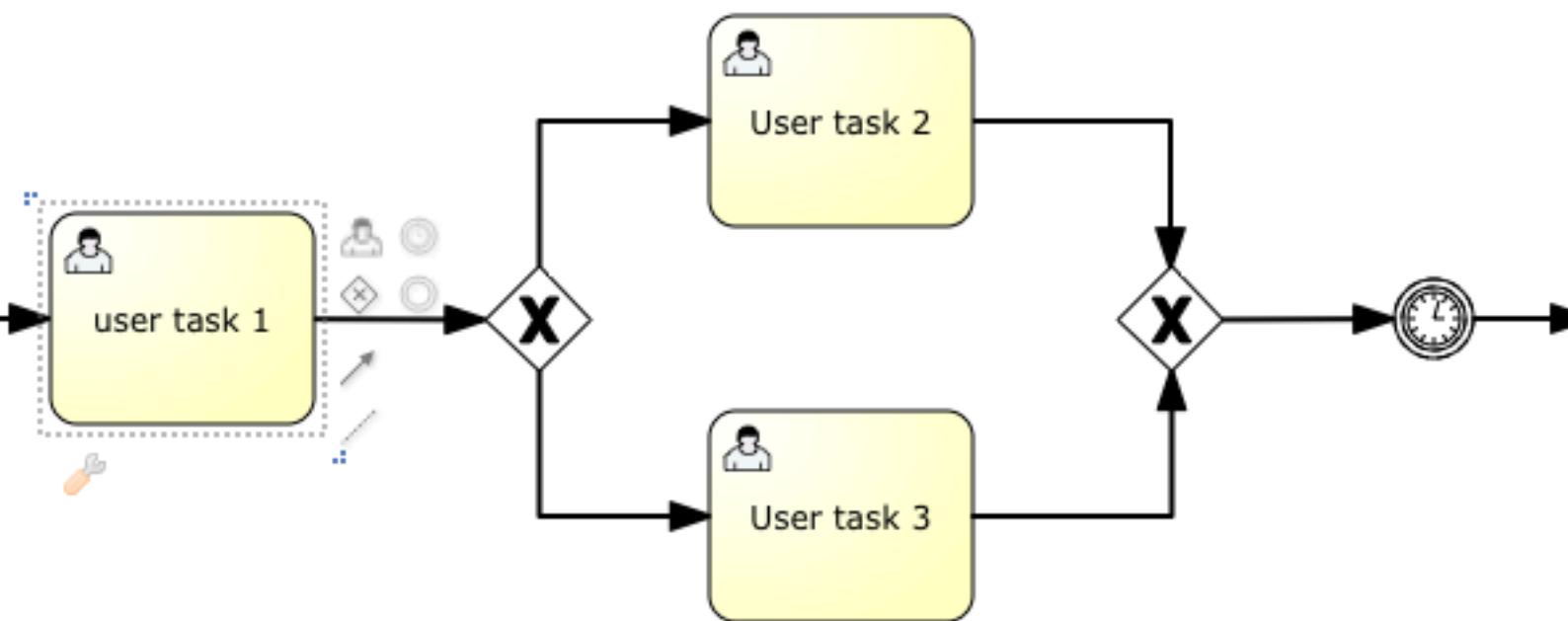
Process Diagram



14. 1. #编辑模型

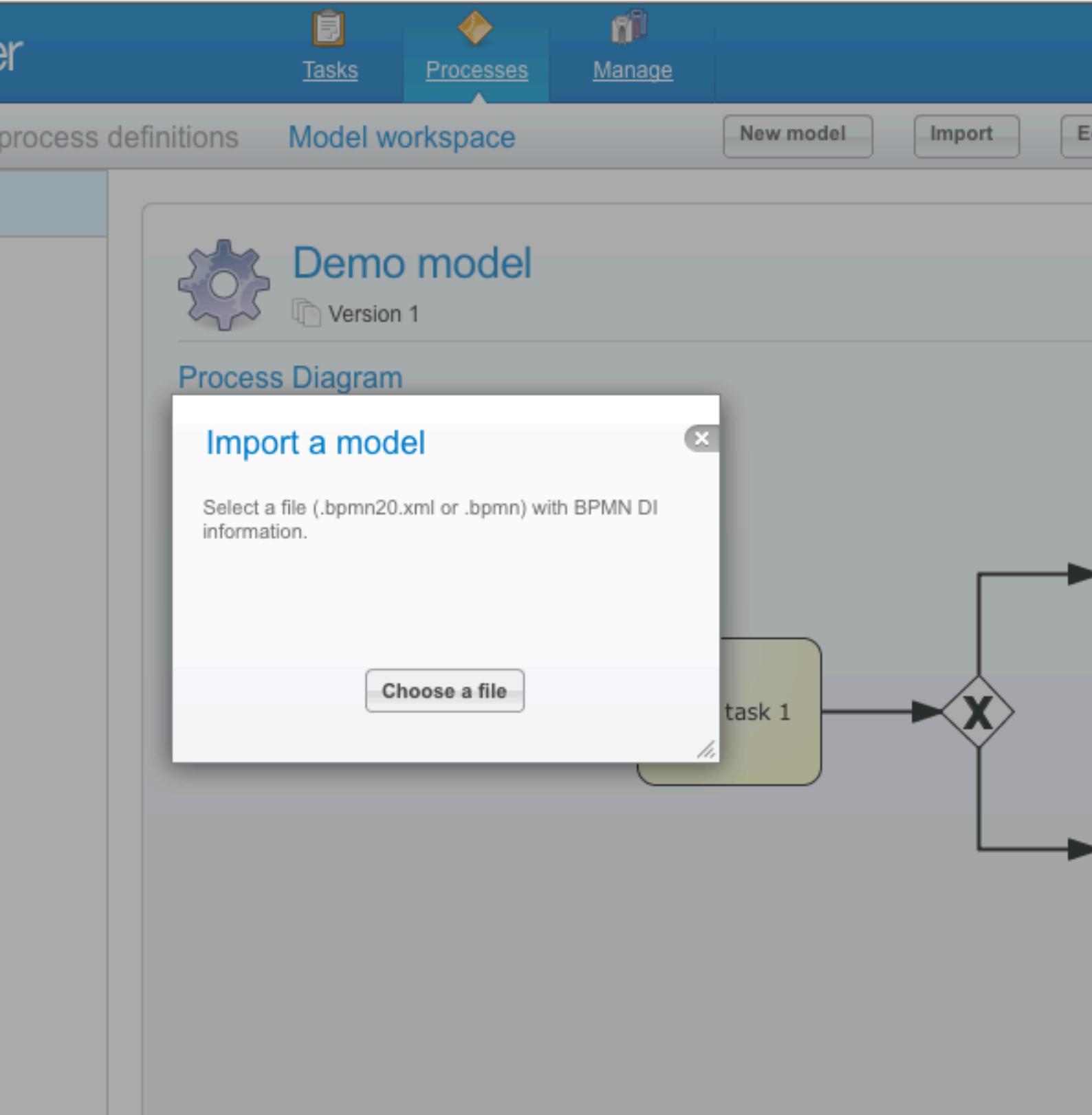
点击模型工作区的编辑按钮，会打开Modeler。 屏幕左侧是BPMN元素工具面板，也可以使用Activiti的扩展组件。 你可以在需要时把新元素拖拽到画布中。 屏幕右侧是选中额元素的和苏醒。 例子截屏中选中了一个用户任务，你可以填写用户任务的属性，比如分配，表单属性和持续时间。 点击屏幕右上方的关闭按钮就可以返回Activiti Explorer。

by KIS BPM



14. 2. #导入模型

你也可以把模型导入到模型工作台中，然后就可以在Activiti Modeler中进行编辑。点击导入按钮，选择.bpmn或.bpmn20.xml文件。注意BPMN XML文件必须包含BPMN DI（坐标）信息。



14. 3. #把发布的流程定义转换成可编辑的模型

发布的流程定义可以转换成模型，然后就可以在Activiti Modeler中编辑了。注意流程定义必须包含BPMN DI（坐标）信息。



Tasks



Processes



Manage

definitions Model workspace



process1



Version 1



Deployed 25 minutes ago

Process Diagram

Convert process1 for editing

Are you sure you want to create a new editor model from this process definition?

Cancel

Convert

14. 4. #把模型导出成BPMN XML

模型工作区中的模型可以导出成BPMN XML文件。选择模型操作选项中的导出选项。



Tasks



Processes



Manage

definitions

Model workspace

New model

Import



Demo model

Version 1

Opening process.bpmn20.xml

You have chosen to open: process.bpmn20.xml

which is a: XML Document

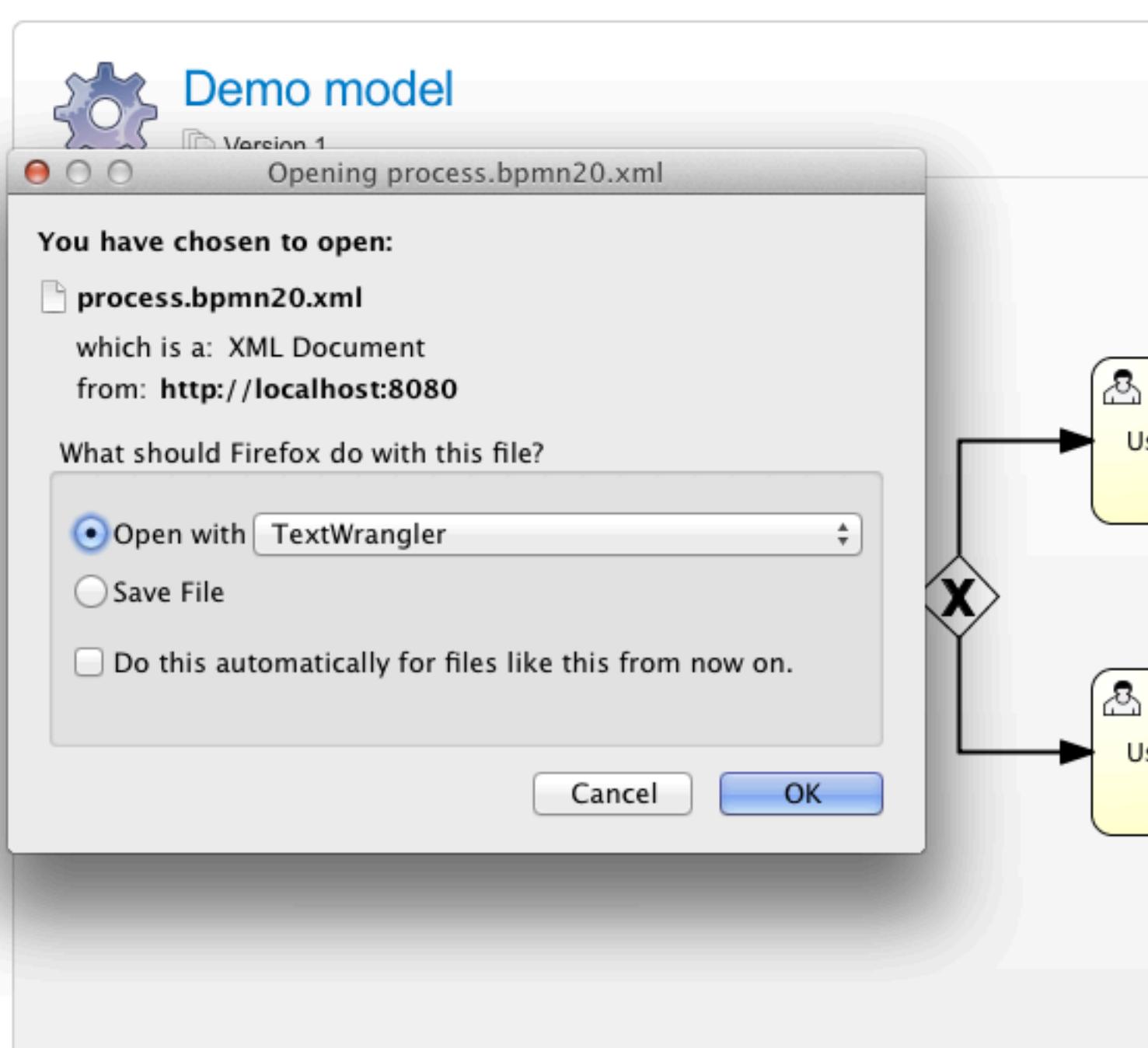
from: <http://localhost:8080>

What should Firefox do with this file?

 Open with Save File Do this automatically for files like this from now on.

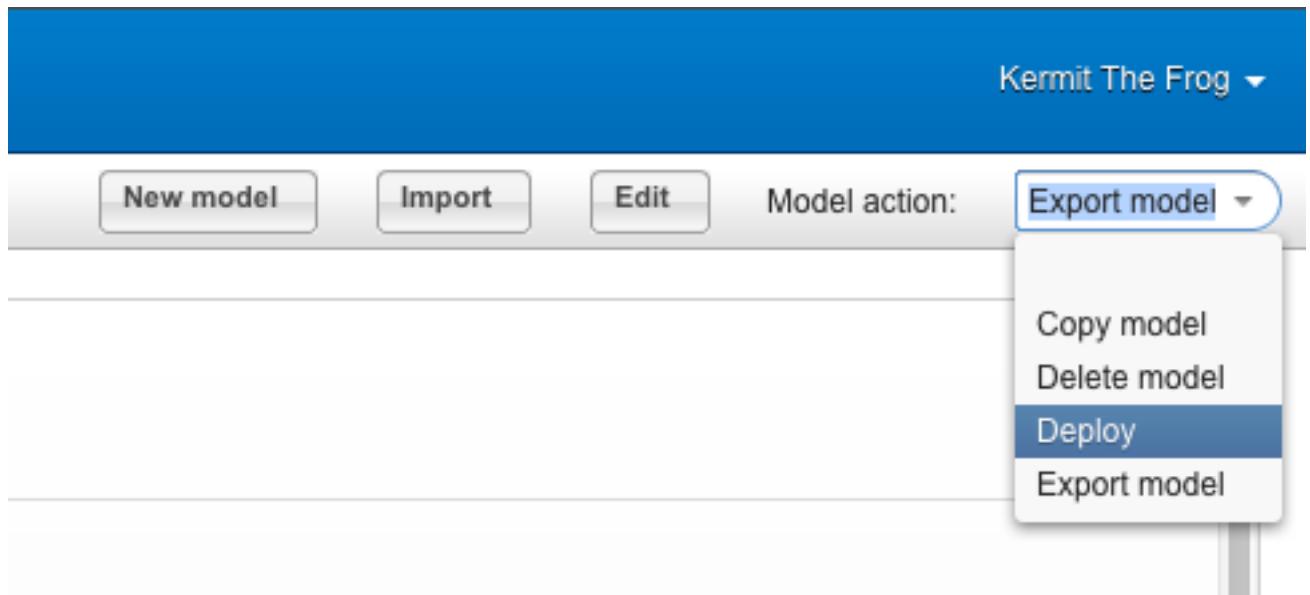
Cancel

OK



14.5. #把模型部署到Activiti引擎中

在模型设置好所有运行所需的属性之后，它就可以发布到Activiti引擎里。选择模型操作选项中的发布选项。



第#15#章#REST API

15.1. #通用Activiti REST原则

15.1.1. #安装与认证

activiti包含了一个activiti引擎的REST API，把activiti-rest.war部署到像Apache Tomcat这样的servlet容器就可以使用。不过，它也可以使用在其他web应用中，把activiti-rest的依赖都加入classpath，添加servlet，并映射到你的应用中。

默认情况下，activiti引擎会连接内存数据库H2。你可以修改WEB-INF/classes目录下的db.properties来修改数据库设置。REST API使用JSON格式（<http://www.json.org>），它是基于Restlet（<http://www.restlet.org>）开发的。

默认所有REST资源都需要先使用有效的activiti用户认证后才能使用。会使用Basic HTTP认证，所以你要一直在请求的HTTP头中包含一个Authorization: Basic ...=属性，或在请求url中包含用户名和密码（比如：<http://username:password@localhost...>）。

建议把Basic认证与HTTPS一起使用。

可以从删除对应资源的认证，或添加额外的授权给一个认证的用户（比如，把用户加入一个组，让它可以执行URL Y）。可以使用org.activiti.rest.common.filter.RestAuthenticator的实现，它有两个方法：

- boolean requestRequiresAuthentication(Request request)：在请求认证检查之前调用（通过头部传递合法的账号和密码）。如果方法返回true，这个方法就需要认证才能访问。如果返回false，无论请求是否认证都可以访问。如果返回false，就不会为这个方法调用isRequestAuthorized。
- boolean isRequestAuthorized(Request request)：在用户已经通过activiti账号管理认证后，但是在请求实际之前调用。可以用来检查认证用户是否可以访问对应请求。如果返回true，会允许请求执行。如果返回true，请求不会执行，客户端会收到对应的错误。

自定义的RestAuthenticator应该设置到RestletServlet

的org.activiti.rest.service.application.ActivitiRestServicesApplication中。最简单的方法是创建ActivitiRestServicesApplication的子类，并在servlet-mapping中设置自定义的类名：

```
<!-- Restlet adapter -->
<servlet>
    <servlet-name>RestletServlet</servlet-name>
    <servlet-class>org.restlet.ext.servlet.ServerServlet</servlet-class>
    <init-param>
        <!-- Application class name -->
        <param-name>org.restlet.application</param-name>
        <param-value>com.my.company.CustomActivitiRestServicesApplication</param-value>
    </init-param>
</servlet>
```

15.1.2. #使用Tomcat

根据Tomcat的默认安全配置 [<http://tomcat.apache.org/tomcat-7.0-doc/security-howto.html>]，默认转码的前斜线（%2F和%5C）都不允许使用（返回400）。这可能对部署资

源和它们的数据URL造成影响，URL可能会包含转移的前斜线。当出现预期外的400问题，设置下面这个系统参数：-Dorg.apache.tomcat.util.buf.UDecoder.ALLOW_ENCODED_SLASH=true。

15.1.3. #方法和返回值

表 15.1. HTTP方法和对应操作

方法	操作
GET	获得一个资源或获得多个资源
POST	创建一个新资源。当请求结果太复杂，无法放到GET请求的URL中，也用来查询资源。
PUT	更新已有资源的属性。也用来调用现存资源的功能。
DELETE	删除现存资源。

表 15.2. HTTP方法响应代码

响应	描述
200 - Ok	操作成功，响应返回（GET和PUT请求）。
201 - Created	操作成功，实体已创建，并返回到响应体中（POST请求）。
204 - No content	操作成功，实体已删除，不会返回响应体（DELETE请求）。
401 - Unauthorized	操作失败。操作要求设置Authentication头部。如果请求中已经设置了头部，对应的凭证是无效的或者用户不允许执行这个操作。
403 - Forbidden	禁止操作，不要重试。这不是认证和授权的问题，这是禁止操作。比如：删除一个执行中流程的任务是不允许的，无论用户或流程任务的状态。
404 - Not found	操作失败。找不到请求的资源。
405 - Method not allowed	操作失败。使用的资源方法不允许调用。比如：想更新（PUT）已部署的资源会返回405结果。
409 - Conflict	操作失败。更新其他操作应更新的资源，会导致更新不合法。也可以表示一个结合中新创建的资源的id已经存在了。
415 - Unsupported Media Type	操作失败。请求体包含了不支持的媒体类型。当请求体的JSON中包含未知的属性或值时，也会返回这个响应，一般是因为无法处理的错误格式或类型。
500 - Internal server error	操作失败。执行操作时出现了预期外的异常。响应体中包含错误的细节。

media-type和HTTP响应永远都是application/json，除非需要使用二进制内容（比如：发布资源数据），会使用内容的media-type。

15.1.4. #错误响应体

当发生错误时（包括客户端和服务端，4XX和5XX状态码），响应体会包含描述发生的错误的描述。下面是任务不存在时出现的404状态：

```
{
  "statusCode" : 404,
  "errorMessage" : "Could not find a task with id '444'."
}
```

15.1.5. #请求参数

15.1.5.1. #URL片段

url上的参数（比如http://host/activiti-rest/service/repository/deployments/{deploymentId}上的deploymentId）都需要转义（参考URL编码或百分号编码 [https://en.wikipedia.org/wiki/Percent-encoding]来解决特殊字符问题）。大多数框架都有这种内置功能，但我们要把这个问题考虑在内。特别是对于可能包含前斜线（比如，部署资源）的情况，这就是必须的。

15.1.5.2. #Rest URL查询参数

设置在URL中的查询参数（比如，http://host/activiti-rest/service/deployments?name=Deployment中的name参数）可以使用以下类型，它们对应着以下REST-API文档：

表 15.3. URL查询参数类型

类型	格式
String	纯文本参数。可以包含任何URL允许的合法的字符。对于XXXLike参数，字符串应该包含通配符%（还要通过url编码）。这可以指定模糊搜索的意图。比如'Tas%'匹配所有以'Tas'开头的值。
Integer	整数参数。只能包含非小数的整数值，在-2.147.483.648与2.147.483.647之间。
Long	长整形参数。只能包含非小数的长整形值，在-9.223.372.036.854.775.808与9.223.372.036.854.775.807之间。
Boolean	布尔类型参数。可以是true或false。如果使用了其他值，会返回'405 - Bad request'响应。
Date	日期类型。使用ISO-8601日期格式（参考维基百科上的ISO-8601 [http://en.wikipedia.org/wiki/ISO_8601]），用于时间和日期组件（比如2013-04-03T23:45Z）。

15.1.5.3. #JSON内容参数

表 15.4. JSON参数类型

类型	格式
String	纯文本参数。可以包含任何URL允许的合法的字符。对于XXXLike参数，字符串应该包含通配符%（还要通过url编码）。这可以指定模糊搜索的意图。比如'Tas%'匹配所有以'Tas'开头的值。
Integer	整数参数，JSON数字。只能包含非小数的整数值，在-2.147.483.648与2.147.483.647之间。
Long	长整形参数，JSON数字。只能包含非小数的长整形值，在-9.223.372.036.854.775.808与9.223.372.036.854.775.807之间。
Boolean	布尔类型参数，JSON布尔。可以是true或false。如果使用了其他值，会返回'405 - Bad request'响应。
Date	日期类型，JSON文本。使用ISO-8601日期格式（参考维基百科上的ISO-8601 [http://en.wikipedia.org/wiki/ISO_8601]），用于时间和日期组件（比如2013-04-03T23:45Z）。

15.1.5.4. #分页与排序

分页与排序参数可以添加到URL的query-string中（比如http://host/activiti-rest/service/deployments?sort=name中的name参数）。

表 15.5. 查询JSON参数

参数	默认值	描述
sort	根据查询实现而不同	查询的名称，对于不同的查询实现，默认值也不同。
order	asc	排序的方式，可以为'asc'或'desc'。
start	0	分页查询开始的值，默认从0开始。
size	10	分页查询每页显示的记录数。默认为10。

15.1.5.5. #JSON查询变量格式

```
{
  "name" : "variableName",
  "value" : "variableValue",
  "operation" : "equals",
  "type" : "string"
}
```

表 15.6. 查询JSON参数

参数	是否必须	描述
name	否	查询包含的变量名称。在一些查询中使用'equals'查询对应资源的所有值时，可以为空。
value	是	查询包含的变量值，要包含给定类型的正确格式。
operation	是	查询使用的参数，可以是以下值： equals, notEquals, equalsIgnoreCase, notEqualsIgnoreCase, lessThan, greaterThan, lessThanOrEquals, greaterThanOrEquals 和 like。
type	否	使用的变量的类型。如果省略，类型会根据value参数决定。所以JSON文本值都会认为是string类型，JSON布尔对应boolean，JSON数字根据数字的长度对应long或integer。在不确定时，建议使用精确的类型。下面列出了不稳定支持的其他类型。

表 15.7. 默认查询JSON类型

类型名称	描述
string	对应java.lang.String。
short	对应java.lang.Integer。
integer	对应java.lang.Integer。
long	对应java.lang.Long。
double	对应java.lang.Double。
boolean	对应java.lang.Boolean。
date	对应java.util.Date。 JSON字符串会使用ISO-8601格式进行转换。

15.1.5.6. #变量格式

在使用变量时（执行，流程和任务），REST-api使用一些通用原则和JSON格式实现读写。变量的JSON格式看起来如下所示：

```
{
  "name" : "variableName",
  "value" : "variableValue",
  "valueUrl" : "http://...",
```

```

    "scope" : "local",
    "type" : "string"
}

```

表 15.8. 变量JSON属性

参数	是否必须	描述
name	是	变量名称。
value	否	变量值。写入变量时，如果没有设置value，会认为值是null。
valueUrl	否	当读取变量的类型为binary或serializable时，这个属性会指向获取原始二进制数据的URL。
scope	否	变量的范围。如果为'local'，变量会对应到请求的资源。如果为'global'，变量会定义到请求资源的上级（或上级树的任何上级）。当写入变量，没有设置scope时，假设使用global。
type	否	变量类型。参考下面的表格对类型的描述。当写入变量，没有设置类型时，会根据请求的JSON属性来推断它的类型，限制为string, double, integer和boolean。如果不确定会用到的类型，建议还是要设置一个类型。

表 15.9. 变量类型

类型名	描述
string	对应java.lang.String。写入时使用JSON文本。
integer	对应java.lang.Integer。写入时，先使用JSON数进行字转换，如果失败就使用JSON文本。
short	对应java.lang.Short。写入时，先使用JSON数字进行转换，如果失败就使用JSON文本。
long	对应java.lang.Long。写入时，先使用JSON数字进行转换，如果失败就使用JSON文本。
double	对应java.lang.Double。写入时，先使用JSON数字进行转换，如果失败就使用JSON文本。
boolean	对应java.lang.Boolean。写入时，使用JSON布尔进行转换。
date	对应java.util.Date。写入时，使用ISO-8601日期格式转换为JSON文本。

类型名	描述
binary	二进制变量，对应字节数组。value属性为null，valueUrl包含指向原始二进制流的URL。
serializable	可序列化的Java对象序列化后的结果。和binary类型相同，value属性为null，valueUrl包含指向原始二进制流的URL。所有可以序列化变量（上面不包含的类型）都会使用这个类型。

可以通过自定义JSON格式来支持更多变量类型（无论是简单数值或复杂/内嵌JSON对象）。

扩展org.activiti.rest.api.RestResponseFactory的initializeVariableConverters()方法，你可以添加自定义的org.activiti.rest.api.service.engine.variable.RestVariableConverter类来支持POJO与对应REST数据的相互转换。实际的JSON转换是通过Jackson实现的。

15. 2. #部署

如果使用tomcat，请参考tomcat用法。

15.2.1. #部署列表

GET repository/deployments

表 15.10. URL查询参数

参数	是否必须	值	描述
name	否	String	只返回指定名称的部署。
nameLike	否	String	只返回名称与指定值相似的部署。
category	否	String	只返回指定分类的部署。
categoryNotEquals	否	String	只返回与指定分类不同的部署。
tenantId	否	String	只返回指定tenantId的部署。
tenantIdLike	否	String	只返回与指定tenantId匹配的部署。
withoutTenantId	否	Boolean	如果为 true, 只返回没有设置tenantId的部署。如果为 false, 忽略 withoutTenantId 参数。
sort	否	'id' (默认), 'name', 'deployTime' 或 'tenantId'	排序属性, 可以与 tenantId一起使用。

通用的分页和排序查询参数都可以使用。

表 15.11. REST响应码

响应码	描述
200	表示请求成功。

成功响应体:

```
{
  "data": [
    {
      "id": "10",
      "name": "activiti-examples.bar",
      "deploymentTime": "2010-10-13T14:54:26.750+02:00",
      "category": "examples",
      "url": "http://localhost:8081/service/repository/deployments/10",
      "tenantId": null
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.2.2. #获得一个部署

```
GET repository/deployments/{deploymentId}
```

表 15.12. 获得一个部署 – URL参数

参数	是否必须	值	描述
deploymentId	是	String	获取部署的id。

表 15.13. 获得一个部署 – 响应码

响应码	描述
200	表示找到了部署，并返回。
404	表示找不到请求的部署。

成功响应体:

```
{
  "id": "10",
  "name": "activiti-examples.bar",
  "deploymentTime": "2010-10-13T14:54:26.750+02:00",
  "category": "examples",
  "url": "http://localhost:8081/service/repository/deployments/10",
  "tenantId": null
}
```

15.2.3. #创建新部署

POST repository/deployments

请求体:

请求体包含的数据类型应该是multipart/form-data。请求里应该只包含一个文件，其他额外的任务都会被忽略。 部署的名称就是文件域的名称。如果需要在一个部署中包含多个资源，把这些文件压缩成zip包，并要确认文件名是以.bar或.zip结尾。

可以在请求体中传递一个额外的参数（表单域） tenantId。字段的值会用来设置部署的租户id。

表 15.14. 创建新部署 - 响应码

响应码	描述
201	表示部署已经创建了。
400	表示请求内没有内容，或部署不支持内容的mimeType。在状态描述中包含更新信息。

成功响应体:

```
{
  "id": "10",
  "name": "activiti-examples.bar",
  "deploymentTime": "2010-10-13T14:54:26.750+02:00",
  "category": null,
  "url": "http://localhost:8081/service/repository/deployments/10",
  "tenantId" : "myTenant"
}
```

15.2.4. #删除部署

DELETE repository/deployments/{deploymentId}

表 15.15. 删除部署 - URL参数

参数	是否必须	值	描述
deploymentId	是	String	删除的部署id。

表 15.16. 删除部署 - 响应码

响应码	描述
204	表示找到了部署并已经删除。响应体为空。
404	表示没有找到请求的部署。

15.2.5. #列出部署内的资源

GET repository/deployments/{deploymentId}/resources

表 15.17. 列出部署内的资源 – URL参数

参数	是否必须	值	描述
deploymentId	是	String	获取资源的部署id。

表 15.18. 列出部署内的资源 – 响应码

响应码	描述
200	表示找到了部署，并返回了资源列表。
404	表示找不到请求的部署。

成功响应体：

```
[
  [
    {
      "id": "diagrams/my-process.bpmn20.xml",
      "url": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resources/diagrams%2Fmy-process.bpmn20.xml",
      "dataUrl": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resourcedata/diagrams%2Fmy-process.bpmn20.xml",
      "mediaType": "text/xml",
      "type": "processDefinition"
    },
    [
      {
        "id": "image.png",
        "url": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resources/image.png",
        "dataUrl": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resourcedata/image.png",
        "mediaType": "image/png",
        "type": "resource"
      }
    ]
  ]
]
```

- mediaType：包含资源的media-type。这是使用（可插拔的）MediaTypeResolver处理的，默认已经支持了一些有限的mime-type映射。
- type：资源类型，可能值为：
 - resource：原始资源。
 - processDefinition：包含一个或多个流程定义的资源。它会被发布器处理。
 - processImage：展示一个已发布流程定义的图形布局。

结果json中的dataUrl属性包含了用来获取二进制资源的真实URL。

15.2.6. #获取部署资源

```
GET repository/deployments/{deploymentId}/resources/{resourceId}
```

表 15.19. 获取部署资源 – URL参数

参数	是否必须	值	描述
deploymentId	是	String	部署ID是请求资源的一部分。

参数	是否必须	值	描述
resourceId	是	String	获取资源的ID 确保URL对资源ID进行编码的情况下，包含斜杠，例如：使用'diagrams%2Fmy-process.bpmn20.xml'代替'diagrams/Fmy-process.bpmn20.xml'。

表 15.20. 获取部署资源 - 响应码

响应码	描述
200	表示部署和资源都已经找到并且部署的资源已经成功返回。
404	表示请求的部署并没有找到或者目前的部署对象并没有该资源ID。状态描述还包含一些额外信息。

成功响应体：

```
{
  "id": "diagrams/my-process.bpmn20.xml",
  "url": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resources/diagrams%2Fmy-process.bpmn20.xml",
  "dataUrl": "http://localhost:8081/activiti-rest/service/repository/deployments/10/resourcedata/diagrams%2Fmy-process.bpmn20.xml",
  "mediaType": "text/xml",
  "type": "processDefinition"
}
```

- **mediaType:** 包含资源的media-type。这是使用（可插拔的）MediaTypeResolver处理的，默认已经支持了一些有限的mime-type映射。
- **type:** 资源类型，可能的值
 - **resource:** 原始资源。
 - **processDefinition:** 包含一个或多个流程定义的资源。它会被发布器处理。
 - **processImage:** 展示一个已发布流程定义的图形布局。

15.2.7. #获取部署资源的内容

```
GET repository/deployments/{deploymentId}/resourcedata/{resourceId}
```

表 15.21. 获取部署资源的内容 - URL参数

参数	是否必须	值	描述
deploymentId	是	String	部署ID是请求资源的一部分。

参数	是否必须	值	描述
resourceId	是	String	资源ID获取数据 确保URL对资源ID进行编码的情况下，包含斜杠，例如：使用' diagrams %2Fmy-process. bpmn20. xml' 代替 'diagrams/Fmy-process. bpmn20. xml'

表 15.22. 获取部署资源的内容 – 响应码

响应码	描述
200	表示部署和资源都已经找到并且部署的资源已经成功返回。
404	表示请求的部署并没有找到或者目前的部署对象并没有该资源ID。状态描述还包含一些额外信息。

成功响应体：

根据请求的资源响应体将包含二进制的资源内容。响应体的content-type的'mimeType'属性将会和资源的返回类型相同。同样，响应头设置content-disposition，允许浏览器下载该文件而不是去显示它。

15.3. #流程定义

15.3.1. #流程定义列表

```
GET repository/process-definitions
```

表 15.23. 流程定义列表 – URL参数

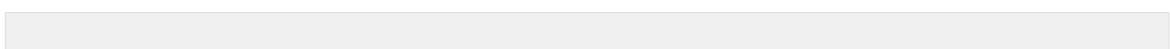
参数	是否必须	值	描述
version	否	integer	只返回给定版本的流程定义。
name	否	String	只返回给定名称的流程定义。
nameLike	否	String	只返回与给定名称匹配的流程定义。
key	否	String	只返回给定key的流程定义。
keyLike	否	String	只返回与给定key匹配的流程定义。

参数	是否必须	值	描述
resourceName	否	String	只返回给定资源名称的流程定义。
resourceNameLike	否	String	只返回与给定资源名称匹配的流程定义。
category	否	String	只返回给定分类的流程定义。
categoryLike	否	String	只返回与给定分类匹配的流程定义。
categoryNotEquals	否	String	只返回非给定分类的流程定义。
deploymentId	否	String	只返回包含在与给定id一致的部署中的流程定义。
startableByUser	否	String	只返回给定用户可以启动的流程定义。
latest	否	Boolean	只返回最新的流程定义版本。只能与'key'或'keyLike'参数一起使用，如果使用了其他参数会返回400的响应。
suspended	否	Boolean	如果为true，只返回挂起的流程定义。如果为false，只返回活动的（未挂起）的流程定义。
sort	否	'name' (默认), 'id', 'key', 'category', 'deploymentId'	排序的属性，可以使用
也可以使用通用的分页与排序查询参数。			

表 15.24. 流程定义列表 - 响应码

响应码	描述
200	表示请求成功，流程定义已返回。
400	表示传递的参数格式错误，或'latest'与'key'，'keyLike'之外的其他参数一起使用了。状态信息包含更多信息。

成功响应体：



```
{
  "data": [
    {
      "id" : "oneTaskProcess:1:4",
      "url" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
      "version" : 1,
      "key" : "oneTaskProcess",
      "category" : "Examples",
      "suspended" : false,
      "name" : "The One Task Process",
      "description" : "This is a process for testing purposes",
      "deploymentId" : "2",
      "deploymentUrl" : "http://localhost:8081/repository/deployments/2",
      "graphicalNotationDefined" : true,
      "resource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.xml",
      "diagramResource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.png",
      "startFormDefined" : false
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "name",
  "order": "asc",
  "size": 1
}
```

- graphicalNotationDefined: 表示流程定义包含图形信息 (BPMN DI)。
- resource: 包含实际部署的BPMN 2.0 xml。
- diagramResource: 包含流程的图形内容, 如果没有图形就返回null。

15.3.2. #获得一个流程定义

GET repository/process-definitions/{processDefinitionId}

表 15.25. 获得一个流程定义 – URL参数

参数	是否必须	值	描述
processDefinitionId	是	String	希望获取的流程定义的 id。

表 15.26. 获得一个流程定义 – 响应码

响应码	描述
200	表示找到了流程定义, 并返回了结果。
404	表示找不到请求的流程定义。

成功响应体:

```
{
  "id" : "oneTaskProcess:1:4",
  "url" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
```

```

"version" : 1,
"key" : "oneTaskProcess",
"category" : "Examples",
"suspended" : false,
"name" : "The One Task Process",
"description" : "This is a process for testing purposes",
"deploymentId" : "2",
"deploymentUrl" : "http://localhost:8081/repository/deployments/2",
"graphicalNotationDefined" : true,
"resource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.xml",
"diagramResource" : "http://localhost:8182/repository/deployments/2/resources/testProcess.png",
"startFormDefined" : false
}

```

- graphicalNotationDefined: 表示流程定义包含图形信息 (BPMN DI)。
- resource: 包含实际部署的BPMN 2.0 xml。
- diagramResource: 包含流程的图形内容, 如果没有图形就返回null。

15.3.3. #更新流程定义的分类

```
PUT repository/process-definitions/{processDefinitionId}
```

请求的JSON:

```
{
  "category" : "updatedcategory"
}
```

表 15.27. 更新流程定义的分类 – 响应码

响应码	描述
200	表示已经修改了流程的分类。
400	表示请求体中没有传递分类。
404	表示找不到请求的流程定义。

成功响应体: 参考repository/process-definitions/{processDefinitionId}.

15.3.4. #获得一个流程定义的资源内容

```
GET repository/process-definitions/{processDefinitionId}/resourcedata
```

表 15.28. 获得一个流程定义的资源内容 – URL参数

参数	是否必须	值	描述
processDefinitionId	是	String	期望获得资源数据的流程定义的id。

响应:

与 GET repository/deployment/{deploymentId}/resourcedata/{resourceId} 的响应码和响应体完全一致。

15.3.5. #获得流程定义的BPMN模型

```
GET repository/process-definitions/{processDefinitionId}/model
```

表 15.29. 获得流程定义的BPMN模型 – URL参数

参数	是否必须	值	描述
processDefinitionId	是	String	期望获得模型的流程定义的id。

表 15.30. 获得流程定义的BPMN模型 – 响应码

响应码	描述
200	表示已找到流程定义，并返回了模型。
404	表示找不到请求的流程定义。

响应体： 响应体是一个转换为JSON格式的org.activiti.bpmn.model.BpmnModel，包含整个流程定义模型。

```
{
  "processes": [
    {
      "id": "oneTaskProcess",
      "xmlRowNumber": 7,
      "xmlColumnNumber": 60,
      "extensionElements": {

      },
      "name": "The One Task Process",
      "executable": true,
      "documentation": "One task process description",

      ...
    ],
    ...
  ]
}
```

15.3.6. #暂停流程定义

```
PUT repository/process-definitions/{processDefinitionId}
```

请求JSON体：

```
{
  "action": "suspend",
  "includeProcessInstances": "false",
  "date": "2013-04-15T00:42:12Z"
}
```

表 15.31. 暂停流程定义 - 请求的JSON参数

参数	描述	是否必须
action	执行的动作。 activate 或 suspend。	是
includeProcessInstances	是否把流程定义下正在运行的流程时也暂停或激活。如果忽略，就不改变流程实例的状态。	否
date	执行暂停或激活的日期（ISO-8601格式）。如果忽略，会立即执行暂停或激活。	否

表 15.32. 暂停流程定义 - 响应码

响应码	描述
200	表示暂停流程成功。
404	表示找不到请求的流程定义。
409	表示请求的流程定义已经暂停了。

成功响应体：参考 `repository/process-definitions/{processDefinitionId}` 的响应。

15.3.7. #激活流程定义

```
PUT repository/process-definitions/{processDefinitionId}
```

请求JSON体：

```
{
  "action" : "activate",
  "includeProcessInstances" : "true",
  "date" : "2013-04-15T00:42:12Z"
}
```

参考暂停流程定义的 JSON参数。

表 15.33. 激活流程定义 - 响应码

响应码	描述
200	表示激活流程成功。
404	表示找不到请求的流程定义
409	表示请求的流程定义已经激活了。

成功响应体：参考 `repository/process-definitions/{processDefinitionId}` 的响应。

15.3.8. #获得流程定义的所有候选启动者

```
GET repository/process-definitions/{processDefinitionId}/identitylinks
```

表 15.34. 获得流程定义的所有候选启动者 – URL参数

参数	是否必须	值	描述
processDefinitionId	是	String	期望获得IdentityLink的流程定义id。

表 15.35. 获得流程定义的所有候选启动者 – 响应码

响应码	描述
200	表示已找到流程定义，并返回了请求的IdentityLink。
404	表示找不到请求的流程定义。

成功响应体：

```
[
  {
    "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/groups/admin",
    "user": null,
    "group": "admin",
    "type": "candidate"
  },
  {
    "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
    "user": "kermit",
    "group": null,
    "type": "candidate"
  }
]
```

15.3.9. #为流程定义添加一个候选启动者

```
POST repository/process-definitions/{processDefinitionId}/identitylinks
```

表 15.36. 为流程定义添加一个候选启动者 – URL参数

参数	是否必填	数据	描述
processDefinitionId	是	String	流程定义的id。

请求体（用户）：

```
{
  "user": "kermit"
}
```

请求体（组）：

```
{
  "groupId": "sales"
}
```

```
}
```

表 15.37. 为流程定义添加一个候选启动者 - 响应码

响应码	描述
201	表示找到了流程定义，并创建了IdentityLink。
404	表示找不到请求的流程定义。

成功响应体：

```
{
  "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
  "user": "kermit",
  "group": null,
  "type": "candidate"
}
```

15.3.10. #删除流程定义的候选启动者

```
DELETE repository/process-definitions/{processDefinitionId}/identitylinks/{family}/{identityId}
```

表 15.38. 删除流程定义的候选启动者 - URL参数

参数	是否必填	数据	描述
processDefinitionId	是	String	流程定义的id。
family	是	String	users 或 groups，依赖 IdentityLink 的类型。
identityId	是	String	需要删除的候选创建者的身份的userId 或 groupId。

表 15.39. 删除流程定义的候选启动者 - 响应码

响应码	描述
204	表示找到了流程定义，并删除了IdentityLink。 响应体为空。
404	表示找不到请求的流程定义，或者在流程定义中找不到与url匹配的IdentityLink。

成功响应体：

```
{
  "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
  "user": "kermit",
  "group": null,
  "type": "candidate"
}
```

15.3.11. #获得流程定义的一个候选启动者

```
GET repository/process-definitions/{processDefinitionId}/identitylinks/{family}/{identityId}
```

表 15.40. 获得流程定义的一个候选启动者 – URL参数

参数	是否必填	数据	描述
processDefinitionId	是	String	流程定义的id。
family	是	String	users 或 groups, 依赖 IdentityLink 的类型。
identityId	是	String	用来获得候选启动者的身份的userId 或 groupId。

表 15.41. 获得流程定义的一个候选启动者 – 响应码

响应码	描述
200	表示找到了流程定义，并返回了IdentityLink。
404	表示找不到请求的流程定义，或者在流程定义中找不到与url匹配的IdentityLink。

成功响应体：

```
{
  "url": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4/identitylinks/users/kermit",
  "user": "kermit",
  "group": null,
  "type": "candidate"
}
```

15.4. #模型

15.4.1. #获得模型列表

```
GET repository/models
```

表 15.42. 获得模型列表 – URL参数

参数	是否必须	值	描述
id	否	String	指返回指定id的模型。
category	否	String	只返回指定分类的模型。
categoryLike	否	String	只返回与给定分类匹配的模型。使用%作为通配符。

参数	是否必须	值	描述
categoryNotEquals	否	String	只返回非指定分类的模型。
name	否	String	只返回指定名称的模型。
nameLike	否	String	只返回与指定名称匹配的模型。使用%作为通配符。
key	否	String	只返回指定key的模型。
deploymentId	否	String	只返回包含在指定部署包中的模型。
version	否	Integer	只返回指定版本的模型。
latestVersion	否	Boolean	如果为 true, 只返回最新版本的模型。最好与key一起使用。如果为 false, 就会返回所有版本。
deployed	否	Boolean	如果为 true, 只返回已部署的模型。如果为 false, 只返回未部署的模型(deploymentId为null)。
tenantId	否	String	只返回指定tenantId的模型。
tenantIdLike	否	String	只返回与指定tenantId匹配的模型。
withoutTenantId	否	Boolean	如果为 true, 只返回没有设置tenantId的模型。如果为 false, 会忽略 withoutTenantId 参数。
sort	否	'id' (默认), 'category', 'createTime', 'key', 'lastUpdateTime', 'name', 'version' 或 'tenantId'	排序的字段, 和'order'一起使用。
可以使用通用的 分页和排序查询参数。			

表 15.43. 获得模型列表 - 响应码

响应码	描述
200	表示请求成功，并返回了模型
400	表示传递的参数格式错误。状态信息中包含更多信息。

成功响应体：

```
{
  "data": [
    {
      "name": "Model name",
      "key": "Model key",
      "category": "Model category",
      "version": 2,
      "metaInfo": "Model metainfo",
      "deploymentId": "7",
      "id": "10",
      "url": "http://localhost:8182/repository/models/10",
      "createTime": "2013-06-12T14:31:08.612+0000",
      "lastUpdateTime": "2013-06-12T14:31:08.612+0000",
      "deploymentUrl": "http://localhost:8182/repository/deployments/7",
      "tenantId": null
    },
    ...
  ],
  "total": 2,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 2
}
```

15.4.2. #获得一个模型

```
GET repository/models/{modelId}
```

表 15.44. 获得一个模型 - URL参数

参数	是否必须	值	描述
modelId	是	String	希望获得的模型id。

表 15.45. 获得一个模型 - 响应码

响应码	描述
200	表示找到了模型并成功返回了。
404	表示找不到请求的模型。

成功响应体：

```
{
  "id":"5",
  "url":"http://localhost:8182/repository/models/5",
  "name":"Model name",
  "key":"Model key",
  "category":"Model category",
  "version":2,
  "metaInfo":"Model metainfo",
  "deploymentId":"2",
  "deploymentUrl":"http://localhost:8182/repository/deployments/2",
  "createTime":"2013-06-12T12:31:19.861+0000",
  "lastUpdateTime":"2013-06-12T12:31:19.861+0000",
  "tenantId":null
}
```

15.4.3. #更新模型

PUT repository/models/{modelId}

请求体:

```
{
  "name":"Model name",
  "key":"Model key",
  "category":"Model category",
  "version":2,
  "metaInfo":"Model metainfo",
  "deploymentId":"2",
  "tenantId":"updatedTenant"
}
```

所有请求参数都是可选的。比如，你可以只在请求体的JSON对象中包含' name' 属性，只更新模型的名称，这样其他的字段都不会受到影响。如果显示包含了一个属性，并设置为null，模型值会被更新为null。比如: {"metaInfo" : null} 会清空模型的metaInfo。

表 15.46. 更新模型 - 响应码

响应码	描述
200	表示找到了模型，并成功更新。
404	表示找不到请求的模型。

成功响应体:

```
{
  "id":"5",
  "url":"http://localhost:8182/repository/models/5",
  "name":"Model name",
  "key":"Model key",
  "category":"Model category",
  "version":2,
  "metaInfo":"Model metainfo",
  "deploymentId":"2",
  "deploymentUrl":"http://localhost:8182/repository/deployments/2",
  "createTime":"2013-06-12T12:31:19.861+0000",
```

```

    "lastUpdateTime": "2013-06-12T12:31:19.861+0000",
    "tenantId": "updatedTenant"
}

```

15.4.4. #新建模型

POST repository/models

请求体:

```
{
  "name": "Model name",
  "key": "Model key",
  "category": "Model category",
  "version": 1,
  "metaInfo": "Model metainfo",
  "deploymentId": "2",
  "tenantId": "tenant"
}
```

All request values are optional. For example, you can only include the 'name' attribute in the request body JSON-object, only setting the name of the model, leaving all other fields null.

表 15.47. 新建模型 - 响应码

响应码	描述
201	表示成功创建了模型。

成功响应体:

```
{
  "id": "5",
  "url": "http://localhost:8182/repository/models/5",
  "name": "Model name",
  "key": "Model key",
  "category": "Model category",
  "version": 1,
  "metaInfo": "Model metainfo",
  "deploymentId": "2",
  "deploymentUrl": "http://localhost:8182/repository/deployments/2",
  "createTime": "2013-06-12T12:31:19.861+0000",
  "lastUpdateTime": "2013-06-12T12:31:19.861+0000",
  "tenantId": "tenant"
}
```

15.4.5. #删除模型

DELETE repository/models/{modelId}

表 15.48. 删除模型 - URL参数

参数	是否必须	值	描述
modelId	是	String	希望删除的模型id。

表 15.49. 删除模型 - 响应码

响应码	描述
204	表示找到了模型并成功删除。响应体为空。
404	表示找不到请求的模型。

15.4.6. #获得模型的可编译源码

```
GET repository/models/{modelId}/source
```

表 15.50. 获得模型的可编译源码 - URL参数

参数	是否必须	值	描述
modelId	是	String	模型id。

表 15.51. 获得模型的可编译源码 - 响应码

响应码	描述
200	表示找到了模型并返回了源码。
404	表示找不到请求的模型。

成功响应体： 响应体包含了模型的原始可编译源码。无论源码的内容是什么，响应的content-type都设置为application/octet-stream。

15.4.7. #设置模型的可编辑源码

```
PUT repository/models/{modelId}/source
```

表 15.52. 设置模型的可编辑源码 - URL参数

参数	是否必填	数据	描述
modelId	是	String	模型的id。

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。

表 15.53. 设置模型的可编辑源码 - 响应码

响应码	描述
200	表示找到了模型并更新了源码。
404	表示找不到请求的模型。

成功响应体： 响应体包含了模型的原始可编译源码。无论源码的内容是什么，响应的content-type都设置为application/octet-stream。

15.4.8. #获得模型的附加可编辑源码

```
GET repository/models/{modelId}/source-extra
```

表 15.54. 获得模型的附加可编辑源码 - URL参数

参数	是否必须	值	描述
modelId	是	String	模型id

表 15.55. 获得模型的附加可编辑源码 - 响应码

响应码	描述
200	表示找到了模型并返回了源码。
404	表示找不到请求的模型。

成功响应体： 响应体包含了模型的原始可编译源码。无论附加源码的内容是什么，响应的content-type都设置为application/octet-stream。

15.4.9. #设置模型的附加可编辑源码

```
PUT repository/models/{modelId}/source-extra
```

表 15.56. 设置模型的附加可编辑源码 - URL参数

参数	是否必填	数据	描述
modelId	是	String	模型id

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容

表 15.57. 设置模型的附加可编辑源码 - 响应码

响应码	描述
200	表示找到了模型并更新了附加源码。
404	表示找不到请求的模型。

成功响应体： 响应体包含了模型的原始可编译源码。无论附加源码的内容是什么，响应的content-type都设置为application/octet-stream。

15.5. #流程实例

15.5.1. #获得流程实例

```
GET runtime/process-instances/{processInstanceId}
```

表 15.58. 获得流程实例 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	流程实例id

表 15.59. 获得流程实例 - 响应码

响应码	描述
200	表示找到了流程实例并成功返回。
404	表示找不到请求的流程实例

成功响应体：

```
{
  "id": "7",
  "url": "http://localhost:8182/runtime/process-instances/7",
  "businessKey": "myBusinessKey",
  "suspended": false,
  "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
  "activityId": "processTask",
  "tenantId": null
}
```

15.5.2. #删除流程实例

```
DELETE runtime/process-instances/{processInstanceId}
```

表 15.60. 删除流程实例 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	希望删除的流程实例id

表 15.61. 删除流程实例 - 响应码

响应码	描述
204	表示找到了流程实例并已删除。响应内容为空。
404	表示找不到请求的流程实例

15.5.3. #激活或挂起流程实例

```
PUT runtime/process-instances/{processInstanceId}
```

表 15.62. 激活或挂起流程实例 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	希望激活或挂起的流程实例id

请求响应体（挂起）：

```
{
  "action": "suspend"
}
```

请求响应体（激活）：

```
{
  "action": "activate"
}
```

表 15.63. 激活或挂起流程实例 - 响应码

响应码	描述
200	表示找到了流程实例并执行了对应操作。
400	表示提供的操作不合法。
404	表示找不到请求的流程实例
409	表示无法执行请求的流程实例操作，因为流程实例已经激活或挂起了。

15.5.4. #启动流程实例

POST runtime/process-instances

请求体（使用流程定义id启动）：

```
{
  "processDefinitionId": "oneTaskProcess:1:158",
  "businessKey": "myBusinessKey",
  "variables": [
    {
      "name": "myVar",
      "value": "This is a variable",
    },
    ...
  ]
}
```

请求体（使用流程定义key启动）：

```
{
  "processDefinitionKey": "oneTaskProcess",
  "businessKey": "myBusinessKey",
  "tenantId": "tenant1",
  "variables": [
    {
      "name": "myVar",
      "value": "This is a variable",
    },
    ...
  ]
}
```

请求体（使用message启动）：

```
{
```

```

"message": "newOrderMessage",
"businessKey": "myBusinessKey",
"tenantId": "tenant1",
"variables": [
    {
        "name": "myVar",
        "value": "This is a variable",
    },
    ...
]
}

```

请求体中只能使用processDefinitionId, processDefinitionKey或message三者之一。参数businessKey, variables和tenantId都是可选的。可以在REST变量章节了解更多关于变量格式的信息。注意忽略变量作用域，流程变量总是local。

表 15.64. 启动流程实例 - 响应码

响应码	描述
201	表示成功启动了流程实例。
400	表示要么找不到流程定义（基于id或key），要么指定的message不会启动流程，要么传递了非法的变量。状态描述中包含了错误相关的额外信息。

成功响应体：

```

{
    "id": "7",
    "url": "http://localhost:8182/runtime/process-instances/7",
    "businessKey": "myBusinessKey",
    "suspended": false,
    "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
    "activityId": "processTask",
    "tenantId": null
}

```

15.5.5. #显示流程实例列表

GET runtime/process-instances

表 15.65. 显示流程实例列表 - URL参数

参数	是否必须	值	描述
id	否	String	只返回指定id的流程实例。
processDefinitionKey	否	String	只返回指定流程定义key的流程实例。
processDefinitionId	否	String	只返回指定流程定义id的流程实例。

参数	是否必须	值	描述
businessKey	否	String	只返回指定 businessKey 的流程实例。
involvedUser	否	String	只返回指定用户参与过的流程实例。
suspended	否	Boolean	如果为 true, 只返回挂起的流程实例。如果为 false, 只返回未挂起(激活)的流程实例。
superProcessInstanceId	否	String	只返回指定上级流程实例id的流程实例(对应 call-activity)。
subProcessInstanceId	否	String	只返回指定子流程id的流程实例(对方call-activity)。
excludeSubprocesses	否	Boolean	只返回非子流程的流程实例。
includeProcessVariables	否	Boolean	表示结果中包含流程变量。
tenantId	否	String	只返回指定tenantId的流程实例。
tenantIdLike	否	String	只返回与指定tenantId匹配的流程实例。
withoutTenantId	否	Boolean	如果为 true, 只返回未设置tenantId的流程实例。如果为 false, 会忽略 withoutTenantId 参数。
sort	否	String	排序字段, 应该为id(默认), processDefinitionId, 或 processDefinitionKey 三者之一。
可以使用通用的分页和排序查询参数。			

表 15.66. 显示流程实例列表 - 响应码

响应码	描述
200	表示请求成功, 并返回了流程实例。

响应码	描述
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体:

```
{
  "data": [
    {
      "id": "7",
      "url": "http://localhost:8182/runtime/process-instances/7",
      "businessKey": "myBusinessKey",
      "suspended": false,
      "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
      "activityId": "processTask",
      "tenantId": null
    },
    ...
  ],
  "total": 2,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 2
}
```

15.5.6. #查询流程实例

POST query/process-instances

请求体:

```
{
  "processDefinitionKey": "oneTaskProcess",
  "variables": [
    [
      {
        "name": "myVariable",
        "value": 1234,
        "operation": "equals",
        "type": "long"
      },
      ...
    ],
    ...
  ]
}
```

请求体可以包含所有用于显示流程实例列表中的查询参数。除此之外，查询条件中也可以使用变量列表，格式在此。

可以使用通用的 分页和排序查询参数。

表 15.67. 查询流程实例 - 响应码

响应码	描述
200	表示请求成功，并返回了流程实例。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id": "7",
      "url": "http://localhost:8182/runtime/process-instances/7",
      "businessKey": "myBusinessKey",
      "suspended": false,
      "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
      "activityId": "processTask",
      "tenantId": null
    },
    ...
  ],
  "total": 2,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 2
}
```

15.5.7. #获得流程实例的流程图

```
GET runtime/process-instances/{processInstanceId}
```

表 15.68. 获得流程实例的流程图 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	希望获得流程图的流程实例id。

表 15.69. 获得流程实例的流程图 - 响应码

响应码	描述
200	表示找到了流程实例，并返回了流程图。
400	表示找不到请求的流程实例，流程不包含挺信息(BPMN:DI)，所以没有创建图片。
404	表示找不到请求的流程实例

成功响应体：

```
[{"id": "7", "url": "http://localhost:8182/runtime/process-instances/7", "businessKey": "myBusinessKey", "suspended": false, "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4", "activityId": "processTask", "tenantId": null}, {"id": "8", "url": "http://localhost:8182/runtime/process-instances/8", "businessKey": "myBusinessKey", "suspended": false, "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/processOne%3A1%3A4", "activityId": "processTask", "tenantId": null}], {"total": 2, "start": 0, "sort": "id", "order": "asc", "size": 2}
```

```
{
  "id":"7",
  "url":"http://localhost:8182/runtime/process-instances/7",
  "businessKey":"myBusinessKey",
  "suspended":false,
  "processDefinitionUrl":"http://localhost:8182/repository/process-definitions/processOne%3A1%3A4",
  "activityId":"processTask"
}
```

15.5.8. #获得流程实例的参与者

GET runtime/process-instances/{processInstanceId}/identitylinks

表 15.70. 获得流程实例的参与者 – URL参数

参数	是否必须	值	描述
processInstanceId	是	String	关联的流程实例id。

表 15.71. 获得流程实例的参与者 – 响应码

响应码	描述
200	表示找到了流程实例，并返回了IdentityLink。
404	表示找不到请求的流程实例

成功响应体：

```
[
  {
    "url":"http://localhost:8182/runtime/process-instances/5/identitylinks/users/john/customType",
    "user":"john",
    "group":null,
    "type":"customType"
  },
  {
    "url":"http://localhost:8182/runtime/process-instances/5/identitylinks/users/paul/candidate",
    "user":"paul",
    "group":null,
    "type":"candidate"
  }
]
```

注意groupId总是null，因为只有用户才能实际参与到流程实例中。

15.5.9. #为流程实例添加一个参与者

POST runtime/process-instances/{processInstanceId}/identitylinks

表 15.72. 为流程实例添加一个参与者 – URL参数

参数	是否必须	值	描述
processInstanceId	是	String	关联的流程实例id。

请求体：

```
{
  "userId": "kermit",
  "type": "participant"
}
```

userId 和 type 都是必填项。

表 15.73. 为流程实例添加一个参与者 - 响应码

响应码	描述
201	表示找到了流程实例，并创建了关联。
400	表示请求体没有包含userId或type。
404	表示找不到请求的流程实例

成功响应体：

```
{
  "url": "http://localhost:8182/runtime/process-instances/5/identitylinks/users/john/customType",
  "user": "john",
  "group": null,
  "type": "customType"
}
```

注意groupId总是null，因为只有用户才能实际参与到流程实例中。

15.5.10. #删除一个流程实例的参与者

```
DELETE runtime/process-instances/{processInstanceId}/identitylinks/users/{userId}/{type}
```

表 15.74. 删除一个流程实例的参与者 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	流程实例id
userId	是	String	要删除关联的用户id
type	是	String	删除的关联类型

表 15.75. 删除一个流程实例的参与者 - 响应码

响应码	描述
204	表示找到了流程实例，并删除了关联。响应体为空。
404	表示找不到请求的流程实例，或找不到期望删除的关联。响应状态包含了错误的详细信息。

成功响应体：

```
{
  "url": "http://localhost:8182/runtime/process-instances/5/identitylinks/users/john/customType",
```

```

    "user":"john",
    "group":null,
    "type":"customType"
}

```

注意groupId总是null，因为只有用户才能实际参与到流程实例中。

15.5.11. #列出流程实例的变量

```
GET runtime/process-instances/{processInstanceId}/variables
```

表 15.76. 列出流程实例的变量 – URL参数

参数	是否必须	值	描述
processInstanceId	是	String	变量对应的流程实例id

表 15.77. 列出流程实例的变量 – 响应码

响应码	描述
200	表示找到了流程实例，并返回了变量。
404	表示找不到请求的流程实例

成功响应体：

```
[
  {
    "name":"intProcVar",
    "type":"integer",
    "value":123,
    "scope":"local"
  },
  {
    "name":"byteArrayProcVar",
    "type":"binary",
    "value":null,
    "valueUrl":"http://localhost:8182/runtime/process-instances/5/variables/byteArrayProcVar/data",
    "scope":"local"
  },
  ...
]
```

当变量为二进制或序列化类型时，valueUrl给出了获得原始数据的URL。如果是普通变量，变量值就会直接包含在响应中。注意只会返回local作用域的变量，因为流程实例变量没有global作用域。

15.5.12. #获得流程实例的一个变量

```
GET runtime/process-instances/{processInstanceId}/variables/{variableName}
```

表 15.78. 获得流程实例的一个变量 – URL参数

参数	是否必须	值	描述
processInstanceId	是	String	变量对应的流程实例id

参数	是否必须	值	描述
variableName	是	String	获取变量的名称

表 15.79. 获得流程实例的一个变量 - 响应码

响应码	描述
200	表示找到了流程实例和变量，并返回了变量。
400	表示请求体不完全，或包含非法值。状态描述包含对应错误的详细信息。
404	表示找不到请求的流程实例，或流程实例中不包含指定名称的变量。状态描述中包含对应错误的详细信息。

成功响应体：

```
{
  "name": "intProcVar",
  "type": "integer",
  "value": 123,
  "scope": "local"
}
```

当变量为二进制或序列化类型时，valueUrl给出了获得原始数据的URL。如果是普通变量，变量值就会直接包含在响应中。注意只会返回local作用域的变量，因为流程实例变量没有global作用域。

15.5.13. #创建（或更新）流程实例变量

```
POST runtime/process-instances/{processInstanceId}/variables
```

```
PUT runtime/process-instances/{processInstanceId}/variables
```

使用POST时，会创建所有传递的变量。如果流程实例中已经存在了其中一个变量，就会返回一个错误（409 - CONFLICT）。使用PUT时，流程实例中不存在的变量会被创建，已存在的变量会被更新，不会有任何错误。

表 15.80. 创建（或更新）流程实例变量 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	变量对应的流程实例id

请求体：

```
[
  {
    "name": "intProcVar",
    "type": "integer",
    "value": 123
  },
]
```

```
[ ... ]
```

请求体的数组中可以包含任意多个变量。关于变量格式的更多信息可以参考REST变量章节。
注意此处忽略作用域，流程实例只能设置local作用域。

表 15.81. 创建（或更新）流程实例变量 – 响应码

响应码	描述
201	表示找到了流程实例，并创建了变量。
400	表示请求体不完整，或包含非法值。状态描述包含对应错误的详细信息。
404	表示找不到请求的流程实例
409	表示找到了流程实例，但是已经存在一个相同名称的变量（只在使用POST方法时抛出）。可以使用更新方法替代。

成功响应体：

```
[
  {
    "name": "intProcVar",
    "type": "integer",
    "value": 123,
    "scope": "local"
  },
  ...
]
```

15.5.14. #更新一个流程实例变量

```
PUT runtime/process-instances/{processInstanceId}/variables/{variableName}
```

表 15.82. 更新一个流程实例变量 – URL参数

参数	是否必须	值	描述
processInstanceId	是	String	变量对应的流程实例id
variableName	是	String	希望获得的变量名称

请求体：

```
{
  "name": "intProcVar",
  "type": "integer",
  "value": 123
}
```

请求体的数组中可以包含任意多个变量。关于变量格式的更多信息可以参考REST变量章节。
注意此处忽略作用域，流程实例只能设置local作用域。

表 15.83. 更新一个流程实例变量 - 响应码

响应码	描述
200	表示找到了流程实例和变量，并更新了变量。
404	表示找不到请求的流程实例，或找不到给定名称的流程实例变量。状态描述包含对应错误的详细信息。

成功响应体：

```
{
  "name": "intProcVar",
  "type": "integer",
  "value": 123,
  "scope": "local"
}
```

当变量为二进制或序列化类型时，valueUrl给出了获得原始数据的URL。如果是普通变量，变量值就会直接包含在响应中。注意只会返回local作用域的变量，因为流程实例变量没有global作用域。

15.5.15. #创建一个新的二进制流程变量

```
POST runtime/process-instances/{processInstanceId}/variables
```

表 15.84. 创建一个新的二进制流程变量 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	创建新变量对应的流程实例id

请求体：请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- name：必须的变量名称。
- type：创建的变量类型。如果忽略，会假设使用binary，请求的二进制数据会当做二进制数据组保存起来。

成功响应体：

```
{
  "name": "binaryVariable",
  "scope": "local",
  "type": "binary",
  "value": null,
  "valueUrl": "http://.../runtime/process-instances/123/variables/binaryVariable/data"
}
```

表 15.85. 创建一个新的二进制流程变量 - 响应码

响应码	描述
201	表示成功创建了变量，并返回了结果。
400	表示没有提供希望创建的变量名称。状态消息包含详细信息。
404	表示找不到请求的流程实例。
409	表示流程实例中已经包含了给定名称的变量。可以使用PUT方法来更新变量。
415	表示序列化数据包含的对象的类并不在运行Activiti引擎的JVM中，所以无法反序列化。

15.5.16. #更新一个二进制的流程实例变量

```
PUT runtime/process-instances/{processInstanceId}/variables
```

表 15.86. 更新一个二进制的流程实例变量 - URL参数

参数	是否必须	值	描述
processInstanceId	是	String	创建新变量对应的流程实例id

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- name: 必须的变量名称。
- type: 创建的变量类型。如果忽略，会假设使用binary，请求的二进制数据会当做二进制数组保存起来。

成功响应体：

```
{
  "name" : "binaryVariable",
  "scope" : "local",
  "type" : "binary",
  "value" : null,
  "valueUrl" : "http://.../runtime/process-instances/123/variables/binaryVariable/data"
}
```

表 15.87. 更新一个二进制的流程实例变量 - 响应码

响应码	描述
200	表示成功更新了变量，并返回了结果。
400	表示未提供希望更新的变量名称。状态消息包含了详细信息。
404	表示找不到请求的流程实例id，或找不到指定名称的流程实例变量。

响应码	描述
415	表示序列化数据包含的对象的类并不在运行 Activiti 引擎的 JVM 中，所以无法反序列化。

15.6. #分支

15.6.1. #获取一个分支

```
GET runtime/executions/{executionId}
```

表 15.88. 获取一个分支 - URL参数

参数	是否必须	值	描述
executionId	是	String	获取分支的id

表 15.89. 获取一个分支 - 响应码

响应码	描述
200	表示找到了分支，并成功返回。
404	表示找不到分支

成功响应体：

```
{
  "id": "5",
  "url": "http://localhost:8182/runtime/executions/5",
  "parentId": null,
  "parentUrl": null,
  "processInstanceId": "5",
  "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
  "suspended": false,
  "activityId": null,
  "tenantId": null
}
```

15.6.2. #对分支执行操作

```
PUT runtime/executions/{executionId}
```

表 15.90. 对分支执行操作 - URL参数

参数	是否必须	值	描述
executionId	是	String	希望执行操作的分支id

请求体（继续执行分支）：

```
{
  "action": "signal"
}
```

请求体（分支接收了信号事件）：

```
{
  "action": "signalEventReceived",
  "signalName": "mySignal"
  "variables": [ ... ]
}
```

提醒分支接收了一个信号事件，要使用一个`signalName`参数。还可以传递`variables`参数，它会在执行操作之前设置到分支中。

请求体（分支接收了消息事件）：

```
{
  "action": "messageEventReceived",
  "messageName": "myMessage"
  "variables": [ ... ]
}
```

提醒分支接收了一个消息事件，要使用一个`messageName`参数。还可以传递`variables`参数，它会在执行操作之前设置到分支中。

表 15.91. 对分支执行操作 - 响应码

响应码	描述
200	表示找到了分支，并执行了操作。
204	表示找到了分支，执行了操作，并且操作导致分支结束了。
400	表示请求的操作不合法，请求中缺少必须的参数，或传递了非法的变量。状态描述中包含了错误相关的详细信息。
404	表示找不到分支

成功响应体（当操作没有导致分支结束的情况）：

```
{
  "id": "5",
  "url": "http://localhost:8182/runtime/executions/5",
  "parentId": null,
  "parentUrl": null,
  "processInstanceId": "5",
  "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
  "suspended": false,
  "activityId": null,
  "tenantId": null
}
```

15.6.3. #获得一个分支的所有活动节点

```
GET runtime/executions/{executionId}/activities
```

返回分支以及子分支当前所有活动的节点（递归所有下级）。

表 15.92. 获得一个分支的所有活动节点 - URL参数

参数	是否必须	值	描述
executionId	是	String	获取节点对应的分支id。

表 15.93. 获得一个分支的所有活动节点 - 响应码

响应码	描述
200	表示找到了分支，并返回了节点。
404	表示找不到分支

成功响应体：

```
[
  "userTaskForManager",
  "receiveTask"
]
```

15.6.4. #获取分支列表

```
GET runtime/executions
```

表 15.94. 获取分支列表 - URL参数

参数	是否必须	值	描述
id	否	String	只返回指定id的分支。
activityId	否	String	只返回指定节点id的分支。
processDefinitionKey	否	String	只返回指定流程定义key的分支。
processDefinitionId	否	String	只返回指定流程定义id的分支。
processInstanceId	否	String	只返回作为指定流程实例id一部分的分支。
messageEventSubscriptionName	否	String	只返回订阅了指定名称消息的分支。
signalEventSubscriptionName	否	String	只返回订阅了指定名称信号的分支。
parentId	否	String	只返回指定分支直接下级的分支。
tenantId	否	String	只返回指定tenantId的分支。

参数	是否必须	值	描述
tenantIdLike	否	String	只返回与指定tenantId匹配的分支。
withoutTenantId	否	Boolean	如果为 true, 只返回未设置tenantId的分支。 如果为 false, 会忽略withoutTenantId 参数。
sort	否	String	排序字段, 应该和processInstanceId (默认), processDefinitionId, processDefinitionKey或tenantId一起使用。
可以使用通用的 分页和排序查询参数。			

表 15.95. 获取分支列表 - 响应码

响应码	描述
200	表示请求成功, 并返回了分支。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体:

```
{
  "data": [
    {
      "id": "5",
      "url": "http://localhost:8182/runtime/executions/5",
      "parentId": null,
      "parentUrl": null,
      "processInstanceId": "5",
      "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
      "suspended": false,
      "activityId": null,
      "tenantId": null
    },
    {
      "id": "7",
      "url": "http://localhost:8182/runtime/executions/7",
      "parentId": "5",
      "parentUrl": "http://localhost:8182/runtime/executions/5",
      "processInstanceId": "5",
      "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
      "suspended": false,
      "activityId": "processTask",
      "tenantId": null
    }
  ]
}
```

```

    "total":2,
    "start":0,
    "sort":"processInstanceId",
    "order":"asc",
    "size":2
}

```

15.6.5. #查询分支

POST query/executions

请求体:

```

{
  "processDefinitionKey": "oneTaskProcess",
  "variables": [
    {
      "name": "myVariable",
      "value": 1234,
      "operation": "equals",
      "type": "long"
    },
    ...
  ],
  "processInstanceVariables": [
    {
      "name": "processVariable",
      "value": "some string",
      "operation": "equals",
      "type": "string"
    },
    ...
  ],
  ...
}

```

请求体可以包含在获取分支列表中可以使用的查询条件。除此之外，也可以在查询中提供variables和processInstanceVariables列表，关于变量的格式可以参考此处。

可以使用通用的 分页和排序查询参数。

表 15.96. 查询分支 - 响应码

响应码	描述
200	表示请求成功，并返回了分支。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体:

```
{
  "data": [

```

```
{
  "id":"5",
  "url":"http://localhost:8182/runtime/executions/5",
  "parentId":null,
  "parentUrl":null,
  "processInstanceId":"5",
  "processInstanceUrl":"http://localhost:8182/runtime/process-instances/5",
  "suspended":false,
  "activityId":null,
  "tenantId":null
},
{
  "id":"7",
  "url":"http://localhost:8182/runtime/executions/7",
  "parentId":"5",
  "parentUrl":"http://localhost:8182/runtime/executions/5",
  "processInstanceId":"5",
  "processInstanceUrl":"http://localhost:8182/runtime/process-instances/5",
  "suspended":false,
  "activityId":"processTask",
  "tenantId":null
],
"total":2,
"start":0,
"sort":"processInstanceId",
"order":"asc",
"size":2
}
```

15.6.6. #获取分支的变量列表

GET runtime/executions/{executionId}/variables?scope={scope}

表 15.97. 获取分支的变量列表 – URL参数

参数	是否必须	值	描述
executionId	是	String	变量对应的分支id。
scope	否	String	local 或 global。如果忽略，会返回local和global作用域下的所有变量。

表 15.98. 获取分支的变量列表 – 响应码

响应码	描述
200	表示找到了分支，并返回了变量。
404	表示找不到请求的分支。

成功响应体：

```
[
{
```

```

    "name": "intProcVar",
    "type": "integer",
    "value": 123,
    "scope": "global"
},
{
    "name": "byteArrayProcVar",
    "type": "binary",
    "value": null,
    "valueUrl": "http://localhost:8182/runtime/process-instances/5/variables/byteArrayProcVar/data",
    "scope": "local"
},
...
]

```

当变量为二进制或序列化类型时，valueUrl给出了获得原始数据的URL。如果是普通变量，变量值就会直接包含在响应中。

15.6.7. #获得分支的一个变量

```
GET runtime/executions/{executionId}/variables/{variableName}?scope={scope}
```

表 15.99. 获得分支的一个变量 – URL参数

参数	是否必须	值	描述
executionId	是	String	变量对应的分支id
variableName	是	String	获取的变量名称。
scope	否	String	local 或 global。如果忽略，返回local变量（如果存在）。如果不存在局部变量，返回global变量（如果存在）。

表 15.100. 获得分支的一个变量 – 响应码

响应码	描述
200	表示找到了分支和变量，并返回了变量。
400	表示请求体不完全，或包含非法数值。状态描述中包含了错误相关的详细信息。
404	表示找不到请求的分支，或分支在请求作用域中不包含指定名称的变量（如果忽略scope参数，既不存在local变量也不存在global变量）。状态描述中包含了错误相关的详细信息。

成功响应体：

```
{
    "name": "intProcVar",
```

```

    "type":"integer",
    "value":123,
    "scope":"local"
}

```

当变量为二进制或序列化类型时，valueUrl给出了获得原始数据的URL。如果是普通变量，变量值就会直接包含在响应中。

15.6.8. #新建（或更新）分支变量

```
POST runtime/executions/{executionId}/variables
```

```
PUT runtime/executions/{executionId}/variables
```

使用POST时，会创建所有传递的变量。如果流程实例中已经存在了其中一个变量，就会返回一个错误（409 – CONFLICT）。使用PUT时，流程实例中不存在的变量会被创建，已存在的变量会被更新，不会有任何错误。

表 15.101. 新建（或更新）分支变量 – URL参数

参数	是否必须	值	描述
executionId	是	String	变量对应的分支id

请求体：

```

[
  {
    "name":"intProcVar"
    "type":"integer"
    "value":123,
    "scope":"local"
  },
  ...
]

```

注意你只能提供作用域相同的变量。如果请求体数组中包含了不同作用域的变量，请求会返回一个错误（400 – BAD REQUEST）。请求体数据中可以传递任意个数的变量。关于变量格式的详细信息可以参考REST变量章节。注意，如果忽略了作用域，只有local作用域的比那两可以设置到流程实例中。

表 15.102. 新建（或更新）分支变量 – 响应码

响应码	描述
201	表示找到了分支，并成功创建了变量。
400	表示请求体不完全或包含了非法数据。状态描述中包含了错误相关的详细信息。
404	表示找不到请求的分支。
409	表示找到了流程实例，但是已经存在一个相同名称的变量（只在使用POST方法时抛出）。可以使用更新方法替代。

成功响应体:

```
[
  {
    "name": "intProcVar",
    "type": "integer",
    "value": 123,
    "scope": "local"
  },
  ...
]
```

15.6.9. #更新分支变量

```
PUT runtime/executions/{executionId}/variables/{variableName}
```

表 15.103. 更新分支变量 – URL参数

参数	是否必须	值	描述
executionId	是	String	希望更新的变量对应的分支id。
variableName	是	String	希望更新的变量名称。

请求体:

```
{
  "name": "intProcVar",
  "type": "integer",
  "value": 123,
  "scope": "global"
}
```

关于变量格式的详细信息可以参考REST变量章节。

表 15.104. 更新分支变量 – 响应码

响应码	描述
200	表示找到了分支和变量，并成功更新了变量。
404	表示找不到请求的分支，或分支不包含指定名称的变量。状态描述中包含了错误相关的详细信息。

成功响应体:

```
{
  "name": "intProcVar",
  "type": "integer",
  "value": 123,
  "scope": "global"
}
```

```
}
```

当变量为二进制或序列化类型时，valueUrl给出了获得原始数据的URL。如果是普通变量，变量值就会直接包含在响应中。

15.6.10. #创建一个二进制变量

```
POST runtime/executions/{executionId}/variables
```

表 15.105. 创建一个二进制变量 – URL参数

参数	是否必须	值	描述
executionId	是	String	希望创建的新变量对应的分支id。

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- name: 必须的变量名称。
- type: 创建的变量类型。如果忽略，会假设使用binary，请求的二进制数据会当做二进制数组保存起来。

成功响应体：

```
{
  "name" : "binaryVariable",
  "scope" : "local",
  "type" : "binary",
  "value" : null,
  "valueUrl" : "http://.../runtime/executions/123/variables/binaryVariable/data"
}
```

表 15.106. 创建一个二进制变量 – 响应码

响应码	描述
201	表示成功创建了变量，并返回了结果。
400	表示没有提供希望创建的变量名称。状态信息包含了详细信息。
404	表示找不到请求的分支。
409	表示分支已经拥有了一个与指定名称相关的变量。使用PUT方法来代替更新分支变量。
415	表示序列化数据包含的对象的类并不在运行Activiti引擎的JVM中，所以无法反序列化。

15.6.11. #更新已经存在的二进制分支变量

```
PUT runtime/executions/{executionId}/variables/{variableName}
```

表 15.107. 更新已经存在的二进制分支变量 - URL参数

参数	是否必须	值	描述
executionId	是	String	希望更新的变量对应的分支id。
variableName	是	String	希望更新的变量名称。

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- name: 必须的变量名称。
- type: 创建的变量类型。如果忽略，会假设使用binary，请求的二进制数据会当做二进制数组保存起来。
- scope: 创建的变量作用于。如果忽略，假设是local。

成功响应体：

```
{
  "name" : "binaryVariable",
  "scope" : "local",
  "type" : "binary",
  "value" : null,
  "valueUrl" : "http://.../runtime/executions/123/variables/binaryVariable/data"
}
```

表 15.108. 更新已经存在的二进制分支变量 - 响应码

响应码	描述
200	表示变量已成功更新，并返回了结果。
400	表示没有提供希望更新的变量名称。状态信息包含了详细信息。
404	表示没有找到分支，或分支不包含指定名称的变量。
415	表示序列化数据包含的对象的类并不在运行Activiti引擎的JVM中，所以无法反序列化。

15.7. #任务

15.7.1. #获取任务

```
GET runtime/tasks/{taskId}
```

表 15.109. 获取任务 - URL参数

参数	是否必须	值	描述
taskId	是	String	希望获得的任务id。

表 15.110. 获取任务 - 响应码

响应码	描述
200	表示找到了任务并返回。
404	表示找不到任务。

成功响应体:

```
{
  "assignee": "kermit",
  "createTime": "2013-04-17T10:17:43.902+0000",
  "delegationState": "pending",
  "description": "Task description",
  "dueDate": "2013-04-17T10:17:43.902+0000",
  "execution": "http://localhost:8182/runtime/executions/5",
  "id": "8",
  "name": "My task",
  "owner": "owner",
  "parentTask": "http://localhost:8182/runtime/tasks/9",
  "priority": 50,
  "processDefinition": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
  "processInstance": "http://localhost:8182/runtime/process-instances/5",
  "suspended": false,
  "taskDefinitionKey": "theTask",
  "url": "http://localhost:8182/runtime/tasks/8",
  "tenantId": null
}
```

- delegationState: 任务的代理状态。可以为null, "pending" 或 "resolved"。

15.7.2. #任务列表

GET runtime/tasks

表 15.111. 任务列表 - URL参数

参数	是否必须	值	描述
name	否	String	只返回指定的名称。
nameLike	否	String	只返回与指定名称匹配的任务。
description	否	String	只返回指定描述的任务。
priority	否	Integer	只返回指定优先级的任务。
minimumPriority	否	Integer	只返回比指定优先级大的任务。
maximumPriority	否	Integer	只返回比指定优先级小的任务。

参数	是否必须	值	描述
assignee	否	String	只返回分配给指定用户的任务。
assigneeLike	否	String	只返回负责人与指定值匹配的任务。
owner	否	String	只返回原拥有人为指定用户的任务。
ownerLike	否	String	只返回原拥有人与指定值匹配的任务。
unassigned	否	Boolean	只返回没有分配给任何人的任务。如果传递false，这个值就会被忽略。
delegationState	否	String	只返回指定代理状态的任务。可选值为pending和resolved。
candidateUser	否	String	只返回可以被指定用户领取的任务。这包含将用户设置为直接候选人和用户作为候选群组一员的情况。
candidateGroup	否	String	只返回可以被指定群组中用户领取的任务。
candidateGroups	否	String	只返回可以被指定群组列表中用户领取的任务。数值使用逗号分隔。
involvedUser	否	String	只返回指定用户参与过的任务。
taskDefinitionKey	否	String	只返回指定任务定义id的任务。
taskDefinitionKeyLike	否	String	只返回任务定义id与指定值匹配的任务。
processInstanceId	否	String	只返回作为指定id的流程实例的一部分的任务。
processInstanceBusinessKey	否	String	只返回作为指定key的流程实例的一部分的任务。

参数	是否必须	值	描述
processInstanceBusinessKeyLike	否	String	只返回业务key与指定值匹配的流程实例的一部分的任务。
processDefinitionKey	否	String	只返回作为指定流程定义key的流程实例的一部分的任务。
processDefinitionKeyLike	否	String	只返回指定流程定义key与指定值匹配的流程实例的一部分的任务。
processDefinitionName	否	String	只返回作为指定流程定义名称的流程实例的一部分的任务。
processDefinitionNameLike	否	String	只返回流程定义名称与指定值匹配的流程实例的一部分的任务。
executionId	否	String	只返回作为指定id分支的一部分的任务。
createdOn	否	ISO Date	只返回指定创建时间的任务。
createdBefore	否	ISO Date	只返回在指定时间之前创建的任务。
createdAfter	否	ISO Date	只返回在指定时间之后创建的任务。
dueOn	否	ISO Date	只返回指定持续时间的任务。
dueBefore	否	ISO Date	只返回持续时间在指定时间之前的任务。
dueAfter	否	ISO Date	只返回持续时间在指定时间之后的任务。
withoutDueDate	否	boolean	只返回没有设置持续时间的任务。如果值为false就会忽略这个属性。
excludeSubTasks	否	Boolean	只返回非子任务的任务。
active	否	Boolean	如果为 true, 只返回未挂起的任务（作为未挂起流程的一部分，或者

参数	是否必须	值	描述
			不属于任何流程）。如果为false，只返回作为挂起流程一部分的任务。
includeTaskLocalVariables	否	Boolean	Indication to include task local variables in the result.
includeProcessVariables	否	Boolean	表示在结果中包含变量。
tenantId	否	String	只返回指定tenantId的任务。
tenantIdLike	否	String	只返回与指定tenantId匹配的任务。
withoutTenantId	否	Boolean	如果为 true，只返回未设置tenantId的任务。 如果为 false，会忽略withoutTenantId 参数。
candidateOrAssigned	否	String	选择已经被领取，或分配给某个用户，或者可以被用户领取（候选用户或组）。
可以使用通用的 分页和排序查询参数。			

表 15.112. 任务列表 – 响应码

响应码	描述
200	表示请求成功，并返回任务。
400	表示传递的参数格式错误，或' delegationState' 使用了不合法的数据 (' pending' 和 ' resolved' 以外的数据)。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "assignee": "kermit",
      "createTime": "2013-04-17T10:17:43.902+0000",
      "delegationState": "pending",
      "description": "Task description",
      "dueDate": "2013-04-17T10:17:43.902+0000",
      "execution": "http://localhost:8182/runtime/executions/5",
      "id": "8",
      "lastUpdate": "2013-04-17T10:17:43.902+0000"
    }
  ]
}
```

```

    "name" : "My task",
    "owner" : "owner",
    "parentTask" : "http://localhost:8182/runtime/tasks/9",
    "priority" : 50,
    "processDefinition" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
    "processInstance" : "http://localhost:8182/runtime/process-instances/5",
    "suspended" : false,
    "taskDefinitionKey" : "theTask",
    "url" : "http://localhost:8182/runtime/tasks/8",
    "tenantId" : null
  }
],
"total": 1,
"start": 0,
"sort": "name",
"order": "asc",
"size": 1
}

```

15.7.3. #查询任务

POST query/tasks

请求体:

```

{
  "name" : "My task",
  "description" : "The task description",
  ...
  "taskVariables" : [
    {
      "name" : "myVariable",
      "value" : 1234,
      "operation" : "equals",
      "type" : "long"
    }
  ],
  "processInstanceVariables" : [
    {
      ...
    }
  ]
}

```

此处所有被支持的JSON参数都和获得任务集合完全一样（除了candidateGroupIn，它只能使用在POST任务查询REST服务中），只是使用JSON体参数的方式替代URL参数，这样就可以使用更加高级的查询方式，并能预防请求uri过长导致的问题。除此之外，可以基于任务和流程变量进行查询。taskVariables 和 processInstanceVariables 都可以包含 此处描述的json数组。

表 15.113. 查询任务 – 响应码

响应码	描述
200	表示请求成功，并返回任务。

响应码	描述
400	表示传递了格式错误的参数，或' delegationState' 传递了非法数据 (' pending' 和 ' resolved' 之外的值)。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "assignee": "kermit",
      "createTime": "2013-04-17T10:17:43.902+0000",
      "delegationState": "pending",
      "description": "Task description",
      "dueDate": "2013-04-17T10:17:43.902+0000",
      "execution": "http://localhost:8182/runtime/executions/5",
      "id": "8",
      "name": "My task",
      "owner": "owner",
      "parentTask": "http://localhost:8182/runtime/tasks/9",
      "priority": 50,
      "processDefinition": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
      "processInstance": "http://localhost:8182/runtime/process-instances/5",
      "suspended": false,
      "taskDefinitionKey": "theTask",
      "url": "http://localhost:8182/runtime/tasks/8",
      "tenantId": null
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "name",
  "order": "asc",
  "size": 1
}
```

15.7.4. #更新任务

PUT runtime/tasks/{taskId}

请求JSON体：

```
{
  "assignee": "assignee",
  "delegationState": "resolved",
  "description": "New task description",
  "dueDate": "2013-04-17T13:06:02.438+02:00",
  "name": "New task name",
  "owner": "owner",
  "parentTaskId": "3",
  "priority": 20
}
```

所有请求参数都是可选的。比如，你可以在请求体的JSON对象中只包含'assignee'属性，只更新任务的负责人，其他字段都不填。当包含的字段值为null时，任务的对应属性会被更新为null。比如：{"dueDate": null}会清空任务的持续时间。

表 15.114. 更新任务 - 响应码

响应码	描述
200	表示成功更新了任务。
404	表示找不到任务。
409	表示请求的任务正在被更新。

成功响应体：参考runtime/tasks/{taskId}的响应。

15.7.5. #操作任务

POST runtime/tasks/{taskId}

完成任务 - JSON体：

```
{
  "action": "complete",
  "variables": ...
}
```

完成任务。可以使用variables参数传递可选的variable数组。关于变量格式的详细信息可以参考REST变量章节。注意，此处忽略变量作用域，变量会设置到上级作用域，除非本地作用域包含了同名变量。这与TaskService.completeTask(taskId, variables) 的行为是相同的。

认领任务 - JSON体：

```
{
  "action": "claim",
  "assignee": "userWhoClaims"
}
```

根据指定的assignee认领任务。如果assignee为null，任务的执行人会变成空，又可以重新认领了。

代理任务 - JSON体：

```
{
  "action": "delegate",
  "assignee": "userToDelegateTo"
}
```

指定assignee代理任务。assignee是必填项。

处理任务 - JSON体：

```
{
```

```

    "action" : "resolve"
}

```

处理任务代理。任务会返回给任务的原负责人（如果存在）。

表 15.115. 操作任务 – 响应码

响应码	描述
200	表示操作成功执行。
400	当请求包含了非法数据或当操作需要assignee参数时，却没有传。
404	表示找不到任务。
409	表示因为冲突导致无法执行操作。可能任务正在被更新，或者，在'claim'认清任务时，任务已经被其他用户认领了。

成功响应体：参考runtime/tasks/{taskId}的响应。

15.7.6. #删除任务

```
DELETE runtime/tasks/{taskId}?cascadeHistory={cascadeHistory}&deleteReason={deleteReason}
```

表 15.116. >删除任务 – URL参数

参数	是否必须	值	描述
taskId	是	String	希望删除的任务id。
cascadeHistory	False	Boolean	删除任务时是否删除对应的任务历史（如果存在）。如果没有设置这个参数，默认为false。
deleteReason	False	String	删除任务的原因。cascadeHistory为true时，忽略此参数。

表 15.117. >删除任务 – 响应码

响应码	描述
204	表示找到任务，并成功删除。响应体为空。
403	表示无法删除任务，因为它是流程的一部分。
404	表示找不到任务。

15.7.7. #获得任务的变量

```
GET runtime/tasks/{taskId}/variables?scope={scope}
```

表 15.118. 获得任务的变量 - URL参数

参数	是否必须	值	描述
taskId	是	String	变量对应的任务id。
scope	False	String	返回的变量作用于。如果为 'local'，只返回任务本身的变量。如果为 'global'，只返回任务上级分支的变量。如果不指定这个变量，会返回所有局部和全局的变量。

表 15.119. 获得任务的变量 - 响应码

响应码	描述
200	表示找到了任务并返回了请求的变量。
404	表示找不到任务。

成功响应体：

```
[
  {
    "name" : "doubleTaskVar",
    "scope" : "local",
    "type" : "double",
    "value" : 99.99
  },
  {
    "name" : "stringProcVar",
    "scope" : "global",
    "type" : "string",
    "value" : "This is a ProcVariable"
  },
  ...
]
```

返回JSON数组型的变量。对响应的详细介绍可以参考REST变量章节。

15.7.8. #获取任务的一个变量

```
GET runtime/tasks/{taskId}/variables/{variableName}?scope={scope}
```

表 15.120. 获取任务的一个变量 - URL参数

参数	是否必须	值	描述
taskId	是	String	获取变量对应的任务id。
variableName	是	String	获取变量对应的名称。

参数	是否必须	值	描述
scope	False	String	返回的变量作用于。如果为 'local'，只返回任务本身的变量。如果为 'global'，只返回任务上级分支的变量。如果不指定这个变量，会返回所有局部和全局的变量。

表 15.121. 获取任务的一个变量 - 响应码

响应码	描述
200	表示找到了任务并返回了请求的变量。
404	表示找不到任务，或者任务不包含指定名称的变量（在指定作用域下）。状态信息包含了详细信息。

成功响应体：

```
{
  "name" : "myTaskVariable",
  "scope" : "local",
  "type" : "string",
  "value" : "Hello my friend"
}
```

对响应的详细介绍可以参考REST变量章节。

15.7.9. #获取变量的二进制数据

```
GET runtime/tasks/{taskId}/variables/{variableName}/data?scope={scope}
```

表 15.122. 获取变量的二进制数据 - URL参数

参数	是否必须	值	描述
taskId	是	String	获取变量数据对应的任务id。
variableName	是	String	获取数据对应的变量名称。只能使用 binary 和 serializable 类型的变量。如果使用了其他类型的变量，会返回 404。
scope	False	String	返回的变量作用于。如果为 'local'，只返回任务本身的变量。如果为 'global'，只返回任务上

参数	是否必须	值	描述
			级分支的变量。如果不指定这个变量，会返回所有局部和全局的变量。

表 15.123. 获取变量的二进制数据 - 响应码

响应码	描述
200	表示找到了任务并返回了请求的变量。
404	表示找不到任务，或者任务不包含指定名称的变量（在指定作用域下），或变量的二进制流不可用。状态信息包含了详细信息。

成功响应体： 响应体包含了变量的二进制值。当类型为 `binary` 时，无论请求的 `accept-type` 头部设置了什么值，响应的 `content-type` 都为 `application/octet-stream`。当类型为 `serializable` 时， `content-type` 为 `application/x-java-serialized-object`。

15.7.10. #创建任务变量

```
POST runtime/tasks/{taskId}/variables
```

表 15.124. 创建任务变量 - URL参数

参数	是否必须	值	描述
taskId	是	String	创建新变量对应的任务 id。

创建简单（非二进制）变量的请求体：

```
[
  {
    "name" : "myTaskVariable",
    "scope" : "local",
    "type" : "string",
    "value" : "Hello my friend"
  },
  {
    ...
  }
]
```

请求体应该是包含一个或多个 JSON 对象的数组，对应应该创建的变量。

- `name`: 必须的变量名称。
- `scope`: 创建的变量的作用域。如果忽略，假设为 `local`。
- `type`: 创建的变量的类型。如果忽略，转换为对应的 JSON 的类型 (`string`, `boolean`, `integer` 或 `double`)。
- `value`: 变量值。

关于变量格式的详细信息可以参考REST变量章节。

成功响应体：

```
[
  {
    "name" : "myTaskVariable",
    "scope" : "local",
    "type" : "string",
    "value" : "Hello my friend"
  },
  {
    ...
  }
]
```

表 15.125. 创建任务变量 – 响应码

响应码	描述
201	表示创建了变量，并返回了结果。
400	表示没有传变量名，或尝试使用global作用域为独立任务（没有关联到流程）创建变量，或请求中的变量为空，或请求没有包含变量数组。状态信息包含了详细信息。
404	表示找不到任务。
409	表示任务已经存在指定名称的变量了。可以使用PUT方法来更新任务变量。

15.7.11. #创建二进制任务变量

```
POST runtime/tasks/{taskId}/variables
```

表 15.126. 创建二进制任务变量 – URL参数

参数	是否必须	值	描述
taskId	是	String	创建新变量对应的任务id。

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- name: 必须的变量名称。
- scope: 创建的变量的作用域。如果忽略，假设使用 local。
- type: 创建的变量类型。如果忽略，会假设使用binary，请求的二进制数据会当做二进制数组保存起来。

成功响应体：

```
{
  "name" : "binaryVariable",
  "scope" : "local",
  "type" : "binary",
  "value" : null,
  "valueUrl" : "http://.../runtime/tasks/123/variables/binaryVariable/data"
}
```

表 15.127. 创建二进制任务变量 - 响应码

响应码	描述
201	表示创建了变量，并返回了结果。
400	表示没有传变量名，或尝试使用global作用域为独立任务（没有关联到流程）创建变量。状态信息包含了详细信息。
404	表示找不到任务。
409	表示任务已经存在指定名称的变量了。可以使用PUT方法来更新任务变量。
415	表示序列化数据包含的对象的类并不在运行Activiti引擎的JVM中，所以无法反序列化。

15.7.12. #更新任务的一个已有变量

```
PUT runtime/tasks/{taskId}/variables/{variableName}
```

表 15.128. 更新任务的一个已有变量 - URL参数

参数	是否必须	值	描述
taskId	是	String	希望更新的变量对应的任务id。
variableName	是	String	希望更新的变量名称。

更新简单（非二进制）变量的请求体：

```
{
  "name" : "myTaskVariable",
  "scope" : "local",
  "type" : "string",
  "value" : "Hello my friend"
}
```

- name: 必须的变量名称。
- scope: 更新的变量的作用域。如果忽略，假设为local。
- type: 更新的变量的类型。如果忽略，转换为对应的JSON的类型 (string, boolean, integer 或 double)。
- value: 变量值。

关于变量格式的详细信息可以参考REST变量章节。

成功响应体：

```
{
  "name" : "myTaskVariable",
  "scope" : "local",
  "type" : "string",
  "value" : "Hello my friend"
}
```

表 15.129. 更新任务的一个已有变量 - 响应码

响应码	描述
200	表示成功更新了变量并返回了结果。
400	表示没有传变量名，或尝试使用global作用域为独立任务（没有关联到流程）创建变量。状态信息包含了详细信息。
404	表示找不到任务，或任务在指定作用域不包含指定名称的变量。状态信息包含错误的详细信息。

15.7.13. #更新一个二进制任务变量

```
PUT runtime/tasks/{taskId}/variables/{variableName}
```

表 15.130. 更新一个二进制任务变量 - URL参数

参数	是否必须	值	描述
taskId	是	String	希望更新的变量对应的任务id。
variableName	是	String	希望更新的变量名称。

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- name: 必须的变量名称。
- scope: 创建的变量的作用域。如果忽略，假设使用 local。
- type: 创建的变量类型。如果忽略，会假设使用binary，请求的二进制数据会当做二进制数组保存起来。

成功响应体：

```
{
  "name" : "binaryVariable",
  "scope" : "local",
  "type" : "binary",
  "value" : null,
  "valueUrl" : "http://.../runtime/tasks/123/variables/binaryVariable/data"
```

}

表 15.131. 更新一个二进制任务变量 - 响应码

响应码	描述
200	表示更新了变量，并返回了结果。
400	表示没有传变量名，或尝试使用global作用域为独立任务（没有关联到流程）创建变量。状态信息包含了详细信息。
404	表示找不到任务，或任务在指定作用域不包含指定名称的变量。
415	表示序列化数据包含的对象的类并不在运行Activiti引擎的JVM中，所以无法反序列化。

15.7.14. #删除任务变量

```
DELETE runtime/tasks/{taskId}/variables/{variableName}?scope={scope}
```

表 15.132. 删除任务变量 - URL参数

参数	是否必须	值	描述
taskId	是	String	希望删除的变量对应的任务id。
variableName	是	String	希望删除的变量名称。
scope	否	String	希望删除的变量的作用域。可以是local 或 global。如果忽略，假设为local。

表 15.133. 删除任务变量 - 响应码

响应码	描述
204	表示找到了任务变量，并成功删除。响应体为空。
404	表示找不到任务，或任务不包含指定名称的变量。状态信息包含错误的详细信息。

15.7.15. #删除任务的所有局部变量

```
DELETE runtime/tasks/{taskId}/variables
```

表 15.134. 删除任务的所有局部变量 - URL参数

参数	是否必须	值	描述
taskId	是	String	希望删除的变量对应的任务id。

表 15.135. 删除任务的所有局部变量 - 响应码

响应码	描述
204	表示已经删除了任务的所有局部变量。响应体为空。
404	表示找不到任务。

15.7.16. #获得任务的所有IdentityLink

```
GET runtime/tasks/{taskId}/identitylinks
```

表 15.136. 获得任务的所有IdentityLink - URL参数

参数	是否必须	值	描述
taskId	是	String	希望获得IdentityLink对应的任务id。

表 15.137. 获得任务的所有IdentityLink - 响应码

响应码	描述
200	表示找到了任务，并返回了IdentityLink。
404	表示找不到任务。

成功响应体：

```
[
  [
    {
      "userId" : "kermit",
      "groupId" : null,
      "type" : "candidate",
      "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/users/kermit/candidate"
    },
    [
      {
        "userId" : null,
        "groupId" : "sales",
        "type" : "candidate",
        "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/groups/sales/candidate"
      },
      ...
    ]
]
```

15.7.17. #获得一个任务的所有组或用户的IdentityLink

```
GET runtime/tasks/{taskId}/identitylinks/users
GET runtime/tasks/{taskId}/identitylinks/groups
```

返回对应于用户或组的IdentityLink。响应体与状态码与获得一个任务的所有IdentityLink完全一样。

15.7.18. #获得一个任务的一个IdentityLink

```
GET runtime/tasks/{taskId}/identitylinks/{family}/{identityId}/{type}
```

表 15.138. 获得一个任务的所有组或用户的IdentityLink – URL参数

参数	是否必填	数据	描述
taskId	是	String	任务的id。
family	是	String	groups 或 users, 对应期望获得哪种 IdentityLink。
identityId	是	String	IdentityLink的id。
type	是	String	IdentityLink的类型。

表 15.139. 获得一个任务的所有组或用户的IdentityLink – 响应码

响应码	描述
200	表示找到了任务和IdentityLink，并成功返回。
404	表示找不到任务，或任务不包含请求的 IdentityLink。状态包含了错误的详细信息。

成功响应体：

```
{
  "userId" : null,
  "groupId" : "sales",
  "type" : "candidate",
  "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/groups/sales/candidate"
}
```

15.7.19. #为任务创建一个IdentityLink

```
POST runtime/tasks/{taskId}/identitylinks
```

表 15.140. 为任务创建一个IdentityLink – URL参数

参数	是否必填	数据	描述
taskId	是	String	任务的id。

请求体（用户）：

```
{
  "userId" : "kermit",
  "type" : "candidate",
}
```

请求体（组）：

```
{
  "groupId" : "sales",
  "type" : "candidate",
}
```

表 15.141. 为任务创建一个IdentityLink - 响应码

响应码	描述
201	表示找到了任务，并创建了IdentityLink。
404	表示找不到任务，或任务不包含请求的IdentityLink。状态包含了错误的详细信息。

成功响应体：

```
{
  "userId" : null,
  "groupId" : "sales",
  "type" : "candidate",
  "url" : "http://localhost:8081/activiti-rest/service/runtime/tasks/100/identitylinks/groups/sales/candidate"
}
```

15.7.20. #删除任务的一个IdentityLink

```
DELETE runtime/tasks/{taskId}/identitylinks/{family}/{identityId}/{type}
```

表 15.142. 删除任务的一个IdentityLink - URL参数

参数	是否必填	数据	描述
taskId	是	String	任务的id。
family	是	String	groups 或 users，对应IdentityLink的种类。
identityId	是	String	IdentityLink的id。
type	是	String	IdentityLink的类型。

表 15.143. 删除任务的一个IdentityLink - 响应码

响应码	描述
204	表示找到了任务和IdentityLink，并成功删除了IdentityLink。响应体为空。
404	表示找不到任务，或任务不包含请求的IdentityLink。状态包含了错误的详细信息。

15.7.21. #为任务创建评论

```
POST runtime/tasks/{taskId}/comments
```

表 15.144. 为任务创建评论 - URL参数

参数	是否必须	值	描述
taskId	是	String	创建评论对应的任务id。

请求体:

```
{
  "message" : "This is a comment on the task.",
  "saveProcessInstanceId" : true
}
```

saveProcessInstanceId参数是可选的，如果为 true 会为评论保存任务的流程实例id。

成功响应体:

```
{
  "id" : "123",
  "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
  "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100/co",
  "message" : "This is a comment on the task.",
  "author" : "kermit",
  "time" : "2014-07-13T13:13:52.232+08:00"
  "taskId" : "101",
  "processInstanceId" : "100"
}
```

表 15.145. 为任务创建评论 - 响应码

响应码	描述
201	表示创建了评论，并返回了结果。
400	表示请求不包含评论。
404	表示找不到任务。

15.7.22. #获得任务的所有评论

```
GET runtime/tasks/{taskId}/comments
```

表 15.146. 获得任务的所有评论 - URL参数

参数	是否必须	值	描述
taskId	是	String	获取评论对应的任务id。

成功响应体:

```
[
  {
    "id" : "123",
    "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
    "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100",
    "message" : "This is a comment on the task.",
    "author" : "kermit",
    "time" : "2014-07-13T13:13:52.232+08:00"
    "taskId" : "101",
    "processInstanceId" : "100"
  },
  {
    "id" : "456",
    "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/456",
    "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100",
    "message" : "This is another comment on the task.",
    "author" : "gonzo",
    "time" : "2014-07-13T13:13:52.232+08:00"
    "taskId" : "101",
    "processInstanceId" : "100"
  }
]
```

表 15.147. 获得任务的所有评论 - 响应码

响应码	描述
200	表示找到了任务，并返回了评论。
404	表示找不到任务。

15.7.23. #获得任务的一个评论

```
GET runtime/tasks/{taskId}/comments/{commentId}
```

表 15.148. 获得任务的一个评论 - URL参数

参数	是否必须	值	描述
taskId	是	String	获取评论对应的任务id。
commentId	是	String	评论的id。

成功响应体：

```
{
  "id" : "123",
  "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
  "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100",
  "message" : "This is a comment on the task.",
  "author" : "kermit",
  "time" : "2014-07-13T13:13:52.232+08:00"
  "taskId" : "101",
  "processInstanceId" : "100"
}
```

表 15.149. 获得任务的一个评论 - 响应码

响应码	描述
200	表示找到了任务和评论，并返回了评论。
404	表示找不到任务，或任务不包含指定id的评论。

15.7.24. #删除任务的一条评论

```
DELETE runtime/tasks/{taskId}/comments/{commentId}
```

表 15.150. 删除任务的一条评论 - URL参数

参数	是否必须	值	描述
taskId	是	String	删除评论对应的任务id。
commentId	是	String	评论的id。

表 15.151. 删除任务的一条评论 - 响应码

响应码	描述
204	表示找到了任务和评论，并删除了评论。响应体为空。
404	表示找不到任务，或任务不包含id的评论。

15.7.25. #获得任务的所有事件

```
GET runtime/tasks/{taskId}/events
```

表 15.152. 获得任务的所有事件 - URL参数

参数	是否必须	值	描述
taskId	是	String	获得事件对应的任务id。

成功响应体：

```
[
  {
    "action" : "AddUserLink",
    "id" : "4",
    "message" : [ "gonzo", "contributor" ],
    "taskUrl" : "http://localhost:8182/runtime/tasks/2",
    "time" : "2013-05-17T11:50:50.000+0000",
    "url" : "http://localhost:8182/runtime/tasks/2/events/4",
    "userId" : null
  },
  ...
]
```

]

表 15.153. 获得任务的所有事件 - 响应码

响应码	描述
200	表示找到了任务，并返回了事件。
404	表示找不到任务。

15.7.26. #获得任务的一个事件

GET runtime/tasks/{taskId}/events/{eventId}

表 15.154. 获得任务的一个事件 - URL参数

参数	是否必须	值	描述
taskId	是	String	获得事件对应的任务id。
eventId	是	String	事件的id。

成功响应体：

```
{
  "action": "AddUserLink",
  "id": "4",
  "message": [ "gonzo", "contributor" ],
  "taskUrl": "http://localhost:8182/runtime/tasks/2",
  "time": "2013-05-17T11:50:50.000+0000",
  "url": "http://localhost:8182/runtime/tasks/2/events/4",
  "userId": null
}
```

表 15.155. 获得任务的一个事件 - 响应码

响应码	描述
200	表示找到了任务和事件，并返回了事件。
404	表示找不到任务，或任务不包含对应id的事件。

15.7.27. #为任务创建一个附件，包含外部资源的链接

POST runtime/tasks/{taskId}/attachments

表 15.156. 为任务创建一个附件，包含外部资源的链接 - URL参数

参数	是否必须	值	描述
taskId	是	String	创建附件对应的任务id。

请求体：

```
{
  "name": "Simple attachment",
  "description": "Simple attachment description",
  "type": "simpleType",
  "externalUrl": "http://activiti.org"
}
```

创建附件只有name是必填的。

成功响应体：

```
{
  "id": "3",
  "url": "http://localhost:8182/runtime/tasks/2/attachments/3",
  "name": "Simple attachment",
  "description": "Simple attachment description",
  "type": "simpleType",
  "taskUrl": "http://localhost:8182/runtime/tasks/2",
  "processInstanceUrl": null,
  "externalUrl": "http://activiti.org",
  "contentUrl": null
}
```

表 15.157. 为任务创建一个附件，包含外部资源的链接 - 响应码

响应码	描述
201	表示创建了附件，并返回了结果。
400	表示请求中缺少了附件名称。
404	表示找不到任务。

15.7.28. #为任务创建一个附件，包含附件文件

```
POST runtime/tasks/{taskId}/attachments
```

表 15.158. 为任务创建一个附件，包含附件文件 - URL参数

参数	是否必须	值	描述
taskId	是	String	创建附件对应的任务id。

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- name: 必须的变量名称。
- description: 附件的描述，可选。
- type: 创建的变量类型。如果忽略，会假设使用binary，请求的二进制数据会当做二进制数据组保存起来。

成功响应体：

```
{
  "id":"5",
  "url":"http://localhost:8182/runtime/tasks/2/attachments/5",
  "name":"Binary attachment",
  "description":"Binary attachment description",
  "type":"binaryType",
  "taskUrl":"http://localhost:8182/runtime/tasks/2",
  "processInstanceUrl":null,
  "externalUrl":null,
  "contentUrl":"http://localhost:8182/runtime/tasks/2/attachments/5/content"
}
```

表 15.159. 为任务创建一个附件，包含附件文件 - 响应码

响应码	描述
201	表示创建了附件，并返回了结果。
400	表示请求中缺少附件名称，或请求中未包含文件。错误信息中包含了详细信息。
404	表示找不到任务。

15.7.29. #获得任务的所有附件

```
GET runtime/tasks/{taskId}/attachments
```

表 15.160. 获得任务的所有附件 - URL参数

参数	是否必须	值	描述
taskId	是	String	获取附件对应的任务id。

成功响应体：

```
[
  {
    "id":"3",
    "url":"http://localhost:8182/runtime/tasks/2/attachments/3",
    "name":"Simple attachment",
    "description":"Simple attachment description",
    "type":"simpleType",
    "taskUrl":"http://localhost:8182/runtime/tasks/2",
    "processInstanceUrl":null,
    "externalUrl":"http://activiti.org",
    "contentUrl":null
  },
  {
    "id":"5",
    "url":"http://localhost:8182/runtime/tasks/2/attachments/5",
    "name":"Binary attachment",
    "description":"Binary attachment description",
    "type":"binaryType",
    "taskUrl":"http://localhost:8182/runtime/tasks/2",
    "processInstanceUrl":null,
    "externalUrl":null,
    "contentUrl":null
  }
]
```

```

    "contentUrl":"http://localhost:8182/runtime/tasks/2/attachments/5/content"
}
]

```

表 15.161. 获得任务的所有附件 - 响应码

响应码	描述
200	表示找到了任务，并返回了附件。
404	表示找不到任务。

15.7.30. #获得任务的一个附件

```
GET runtime/tasks/{taskId}/attachments/{attachmentId}
```

表 15.162. 获得任务的一个附件 - URL参数

参数	是否必须	值	描述
taskId	是	String	获取附件对应的任务id。
attachmentId	是	String	附件的id。

成功响应体：

```

{
  "id":"5",
  "url":"http://localhost:8182/runtime/tasks/2/attachments/5",
  "name":"Binary attachment",
  "description":"Binary attachment description",
  "type":"binaryType",
  "taskUrl":"http://localhost:8182/runtime/tasks/2",
  "processInstanceUrl":null,
  "externalUrl":null,
  "contentUrl":"http://localhost:8182/runtime/tasks/2/attachments/5/content"
}

```

- externalUrl - contentUrl：如果附件是一个外部资源链接，externalUrl包含外部内容的URL。如果附件内容保存在Activiti引擎中，contentUrl会包含获取二进制流内容的URL。
- type：可以是任何有效值。包含一个格式合法的media-type时（比如application/xml, text/plain），二进制HTTP响应的content-type会被设置为对应值。

表 15.163. 获得任务的一个附件 - 响应码

响应码	描述
200	表示找到了任务和附件，并返回了附件。
404	表示找不到任务，或任务不包含对应id的附件。

15.7.31. #获取附件的内容

```
GET runtime/tasks/{taskId}/attachment/{attachmentId}/content
```

表 15.164. 获取附件的内容 – URL参数

参数	是否必须	值	描述
taskId	是	String	获取附件数据对应的任务id。
attachmentId	是	String	附件的id, 当附件指向外部URL, 而不是Activiti中的内容, 就会返回404。

表 15.165. 获取附件的内容 – 响应码

响应码	描述
200	表示找到了任务和附件, 并返回了请求的内容。
404	表示找不到任务, 或任务不包含对应id的任务, 或附件不包含二进制流。状态信息包含了详细信息。

成功响应体： 响应体包含了二进制内容。默认，响应的content-type设置为application/octet-stream，除非附件类型包含了合法的Content-Type。

15.7.32. #删除任务的一个附件

```
DELETE runtime/tasks/{taskId}/attachments/{attachmentId}
```

表 15.166. 删除任务的一个附件 – URL参数

参数	是否必须	值	描述
taskId	是	String	希望删除附件对应的任务id。
attachmentId	是	String	附件的id。

表 15.167. 删除任务的一个附件 – 响应码

响应码	描述
204	表示找到了任务和附件, 并删除了附件。响应体为空。
404	表示找不到任务, 或任务不包含对应id的附件。

15.8. #历史

15.8.1. #获得历史流程实例

```
GET history/historic-process-instances/{processInstanceId}
```

表 15.168. 获得历史流程实例 - 响应码

响应码	描述
200	表示找到了历史流程实例。
404	表示找不到历史流程实例。

成功响应体：

```
{
  "data": [
    {
      "id" : "5",
      "businessKey" : "myKey",
      "processDefinitionId" : "oneTaskProcess%3A1%3A4",
      "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
      "startTime" : "2013-04-17T10:17:43.902+0000",
      "endTime" : "2013-04-18T14:06:32.715+0000",
      "durationInMillis" : 86400056,
      "startUserId" : "kermit",
      "startActivityId" : "startEvent",
      "endActivityId" : "endEvent",
      "deleteReason" : null,
      "superProcessInstanceId" : "3",
      "url" : "http://localhost:8182/history/historic-process-instances/5",
      "variables": null,
      "tenantId":null
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "name",
  "order": "asc",
  "size": 1
}
```

15.8.2. #历史流程实例列表

```
GET history/historic-process-instances
```

表 15.169. 历史流程实例列表 - URL参数

参数	是否必填	数据	描述
processInstanceId	否	String	历史流程实例id。
processDefinitionKey	否	String	历史流程实例的流程定义key。
processDefinitionId	否	String	历史流程实例的流程定义id。
businessKey	否	String	历史流程实例的businessKey。
involvedUser	否	String	历史流程实例的参与者。

参数	是否必填	数据	描述
finished	否	Boolean	表示历史流程实例是否结束了。
superProcessInstanceId	否	String	历史流程实例的上级流程实例id。
excludeSubprocesses	否	Boolean	只返回非子流程的历史流程实例。
finishedAfter	否	Date	只返回指定时间之后结束的历史流程实例。
finishedBefore	否	Date	只返回指定时间之前结束的历史流程实例。
startedAfter	否	Date	只返回指定时间之后开始的历史流程实例。
startedBefore	否	Date	只返回指定时间之前开始的历史流程实例。
startedBy	否	String	只返回由指定用户启动的历史流程实例。
includeProcessVariables	否	Boolean	表示是否应该返回历史流程实例变量。
tenantId	否	String	只返回指定tenantId的实例。
tenantIdLike	否	String	只返回与指定tenantId匹配的实例。
withoutTenantId	否	Boolean	如果为 true, 只返回未设置tenantId的实例。 如果为 false, 会忽略 withoutTenantId 参数。
可以使用通用的 分页和排序查询参数。			

表 15.170. 历史流程实例列表 - 响应码

响应码	描述
200	表示成功返回了历史流程实例。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
```

```
{
  "id" : "5",
  "businessKey" : "myKey",
  "processDefinitionId" : "oneTaskProcess%3A1%3A4",
  "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
  "startTime" : "2013-04-17T10:17:43.902+0000",
  "endTime" : "2013-04-18T14:06:32.715+0000",
  "durationInMillis" : 86400056,
  "startUserId" : "kermit",
  "startActivityId" : "startEvent",
  "endActivityId" : "endEvent",
  "deleteReason" : null,
  "superProcessInstanceId" : "3",
  "url" : "http://localhost:8182/history/historic-process-instances/5",
  "variables": [
    {
      "name": "test",
      "variableScope": "local",
      "value": "myTest"
    }
  ],
  "tenantId":null
},
],
"total": 1,
"start": 0,
"sort": "name",
"order": "asc",
"size": 1
}
```

15.8.3. #查询历史流程实例

POST query/historic-process-instances

请求体:

```
{
  "processDefinitionId" : "oneTaskProcess%3A1%3A4",
  ...

  "variables" : [
    {
      "name" : "myVariable",
      "value" : 1234,
      "operation" : "equals",
      "type" : "long"
    }
  ]
}
```

所有支持的JSON参数字段和获得历史流程实例集合完全一样，但是传递的是JSON参数，而不是URL参数，这样可以支持更高级的参数，同时避免请求uri过长。除此之外，查询支持基于流程变量查询。 variables属性是一个json数组，包含此处描述的格式。

表 15.171. 查询历史流程实例 - 响应码

响应码	描述
200	表示请求成功，并返回结果。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id": "5",
      "businessKey": "myKey",
      "processDefinitionId": "oneTaskProcess%3A1%3A4",
      "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
      "startTime": "2013-04-17T10:17:43.902+0000",
      "endTime": "2013-04-18T14:06:32.715+0000",
      "durationInMillis": 86400056,
      "startUserId": "kermit",
      "startActivityId": "startEvent",
      "endActivityId": "endEvent",
      "deleteReason": null,
      "superProcessInstanceId": "3",
      "url": "http://localhost:8182/history/historic-process-instances/5",
      "variables": [
        {
          "name": "test",
          "variableScope": "local",
          "value": "myTest"
        }
      ],
      "tenantId": null
    },
    {
      "total": 1,
      "start": 0,
      "sort": "name",
      "order": "asc",
      "size": 1
    }
  ]
}
```

15.8.4. #删除历史流程实例

```
DELETE history/historic-process-instances/{processInstanceId}
```

表 15.172. 响应码

响应码	描述
200	表示成功删除了历史流程实例。
404	表示找不到历史流程实例。

15.8.5. #获取历史流程实例的IdentityLink

```
GET history/historic-process-instance/{processInstanceId}/identitylinks
```

表 15.173. 响应码

响应码	描述
200	表示请求成功，并返回了IdentityLink。
404	表示找不到历史流程实例。

成功响应体：

```
[
  {
    "type" : "participant",
    "userId" : "kermit",
    "groupId" : null,
    "taskId" : null,
    "taskUrl" : null,
    "processInstanceId" : "5",
    "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/5"
  }
]
```

15.8.6. #获取历史流程实例变量的二进制数据

```
GET history/historic-process-instances/{processInstanceId}/variables/{variableName}/data
```

表 15.174. 获取历史流程实例变量的二进制数据 – 响应码

响应码	描述
200	表示找到了历史流程实例，并返回了请求的变量数据。
404	表示找不到请求的历史流程实例，或流程实例不包含指定名称的变量，或变量不是二进制流。状态信息包含了详细信息。

成功响应体： 响应体包含了变量的二进制值。当类型为 `binary` 时，无论请求的 `accept-type` 头部设置了什么值，响应的 `content-type` 都为 `application/octet-stream`。当类型为 `serializable` 时，`content-type` 为 `application/x-java-serialized-object`。

15.8.7. #为历史流程实例创建一条新评论

```
POST history/historic-process-instances/{processInstanceId}/comments
```

表 15.175. 为历史流程实例创建一条新评论 – URL参数

参数	必须	值	描述
<code>processInstanceId</code>	是	<code>String</code>	需要创建评论的流程实例id。

请求体:

```
{
  "message" : "This is a comment.",
  "saveProcessInstanceId" : true
}
```

saveProcessInstanceId参数是可选的，如果为 true 会为评论保存流程实例id

成功响应体:

```
{
  "id" : "123",
  "taskUrl" : "http://localhost:8081/activiti-rest/service/runtime/tasks/101/comments/123",
  "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100/co",
  "message" : "This is a comment on the task.",
  "author" : "kermit",
  "time" : "2014-07-13T13:13:52.232+08:00",
  "taskId" : "101",
  "processInstanceId" : "100"
}
```

表 15.176. 为历史流程实例创建一条新评论 – 响应码

响应码	描述
201	表示创建了评论，并返回了结果。
400	表示请求中未包含评论。
404	表示找不到请求的历史流程实例。

15.8.8. #获得一个历史流程实例的所有评论

```
GET history/historic-process-instances/{processInstanceId}/comments
```

表 15.177. 获得流程实例的所有评论 – URL参数

参数	必填	值	描述
processInstanceId	是	String	获取评论对应的流程实例id。

成功响应体:

```
[
  {
    "id" : "123",
    "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100/co",
    "message" : "This is a comment on the task.",
    "author" : "kermit",
    "time" : "2014-07-13T13:13:52.232+08:00",
    "processInstanceId" : "100"
  },
]
```

```
{
  "id" : "456",
  "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100",
  "message" : "This is another comment.",
  "author" : "gonzo",
  "time" : "2014-07-14T15:16:52.232+08:00",
  "processInstanceId" : "100"
}
]
```

表 15.178. 获得流程实例的所有评论 - 响应码

响应码	描述
200	表示找到了流程实例，并返回了评论。
404	表示找不到请求的流程实例。

15.8.9. #获得历史流程实例的一条评论

```
GET history/historic-process-instances/{processInstanceId}/comments/{commentId}
```

表 15.179. 获得历史流程的一条评论 - URL参数

参数	必填	值	描述
processInstanceId	是	String	获得评论对应的历史流程实例。
commentId	Yes	String	评论的id。

成功响应体：

```
{
  "id" : "123",
  "processInstanceUrl" : "http://localhost:8081/activiti-rest/service/history/historic-process-instances/100/comment/123",
  "message" : "This is another comment.",
  "author" : "gonzo",
  "time" : "2014-07-14T15:16:52.232+08:00",
  "processInstanceId" : "100"
}
```

表 15.180. 获得历史流程的一条评论 - 响应码

响应码	描述
200	表示找到了历史流程实例和评论，并返回了评论。
404	表示找不到请求的历史流程实例，或历史流程实例不包含指定id的评论。

15.8.10. #删除历史流程实例的一条评论

```
DELETE history/historic-process-instances/{processInstanceId}/comments/{commentId}
```

表 15.181. 删除历史流程实例的一条评论 - URL参数

参数	必填	值	描述
processInstanceId	是	String	需要删除的评论对应的历史流程实例的id。
commentId	Yes	String	评论的id。

表 15.182. 删除历史流程实例的一条评论 - 响应码

响应码	描述
204	表示找到了历史流程实例，并删除了评论。响应体为空。
404	表示找不到请求的历史流程实例，或历史流程实例不包含指定id的评论。

15.8.11. #获得单独历史任务实例

```
GET history/historic-task-instances/{taskId}
```

表 15.183. 获得单独历史任务实例 - 响应码

响应码	描述
200	表示找到了历史任务实例。
404	表示找不到历史任务实例。

成功响应体：

```
{
  "id" : "5",
  "processDefinitionId" : "oneTaskProcess%3A1%3A4",
  "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
  "processInstanceId" : "3",
  "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
  "executionId" : "4",
  "name" : "My task name",
  "description" : "My task description",
  "deleteReason" : null,
  "owner" : "kermit",
  "assignee" : "fozzie",
  "startTime" : "2013-04-17T10:17:43.902+0000",
  "endTime" : "2013-04-18T14:06:32.715+0000",
  "durationInMillis" : 86400056,
  "workTimeInMillis" : 234890,
  "claimTime" : "2013-04-18T11:01:54.715+0000",
  "taskDefinitionKey" : "taskKey",
  "formKey" : null,
  "priority" : 50,
  "dueDate" : "2013-04-20T12:11:13.134+0000",
  "parentTaskId" : null,
  "url" : "http://localhost:8182/history/historic-task-instances/5",
  "variables": null,
```

```

    "tenantId":null
}

```

15.8.12. #获取历史任务实例

```
GET history/historic-task-instances
```

表 15.184. 获取历史任务实例 – URL参数

参数	是否必填	数据	描述
taskId	否	String	历史任务实例id。
processInstanceId	否	String	历史任务实例的流程实例id。
processDefinitionKey	否	String	历史任务实例的流程定义key。
processDefinitionKeyLike	否	String	历史任务实例的流程定义key与指定值匹配。
processDefinitionId	否	String	历史任务实例的流程定义id。
processDefinitionName	否	String	历史任务实例的流程定义名称。
processDefinitionNameLike	否	String	历史任务实例的流程定义名称与指定值匹配。
processBusinessKey	否	String	历史任务实例的流程实例businessKey。
processBusinessKeyLike	否	String	历史任务实例的流程实例业务key与指定值匹配。
executionId	否	String	历史任务实例的分支id。
taskDefinitionKey	否	String	流程的任务部分的流程定义key。
taskName	否	String	历史任务实例的任务名称。
taskNameLike	否	String	对任务名称使用'like'查询历史任务实例。
taskDescription	否	String	历史任务实例的任务描述。
taskDescriptionLike	否	String	对任务描述使用'like'查询历史任务实例。

参数	是否必填	数据	描述
taskDefinitionKey	否	String	历史任务实例对应的流程定义的任务定义key。
taskDeleteReason	否	String	历史任务实例的删除任务原因。
taskDeleteReasonLike	否	String	对删除任务原因使用'like'查询历史任务实例。
taskAssignee	否	String	历史任务实例的负责人。
taskAssigneeLike	否	String	对负责人使用'like'查询历史任务实例。
taskOwner	否	String	历史任务实例的原拥有者。
taskOwnerLike	否	String	对原拥有者使用'like'查询历史任务实例。
taskInvolvedUser	否	String	历史任务实例的参与者。
taskPriority	否	String	历史任务实例的优先级。
finished	否	Boolean	表示是否历史任务实例已经结束了。
processFinished	否	Boolean	表示历史任务实例的流程实例是否已经结束了。
parentTaskId	否	String	历史任务实例的可能的上级任务id。
dueDate	否	Date	只返回指定持续时间的历史任务实例。
dueDateAfter	否	Date	只返回指定持续时间之后的历史任务实例。
dueDateBefore	否	Date	只返回指定持续时间之前的历史任务实例。
withoutDueDate	否	Boolean	只返回没有设置持续时间的历史任务实例。当设置为false时会忽略这个参数。

参数	是否必填	数据	描述
taskCompletedOn	否	Date	只返回在指定时间完成的历史任务实例。
taskCompletedAfter	否	Date	只返回在指定时间之后完成的历史任务实例。
taskCompletedBefore	否	Date	只返回在指定时间之前完成的历史任务实例。
taskCreatedOn	否	Date	只返回指定创建时间的历史任务实例。
taskCreatedBefore	否	Date	只返回在指定时间前创建的历史任务实例。
taskCreatedAfter	否	Date	只返回在指定时间后创建的历史任务实例。
includeTaskLocalVariables	否	Boolean	表示是否应该返回历史任务实例局部变量。
includeProcessVariables	否	Boolean	表示是否应该返回历史任务实例全局变量。
tenantId	否	String	只返回指定tenantId的历史任务。
tenantIdLike	否	String	只返回与指定tenantId匹配的历史任务。
withoutTenantId	否	Boolean	如果为 true，只返回未设置tenantId的历史任务。如果为 false，会忽略 withoutTenantId 参数。
可以使用通用的 分页和排序查询参数。			

表 15.185. 获取历史任务实例 - 响应码

响应码	描述
200	表示可以查询的历史任务实例。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id" : "5",
      "name": "Test Task"
    }
  ]
}
```

```

"processDefinitionId" : "oneTaskProcess%3A1%3A4",
"processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
"processInstanceId" : "3",
"processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
"executionId" : "4",
"name" : "My task name",
"description" : "My task description",
"deleteReason" : null,
"owner" : "kermit",
"assignee" : "fozzie",
"startTime" : "2013-04-17T10:17:43.902+0000",
"endTime" : "2013-04-18T14:06:32.715+0000",
"durationInMillis" : 86400056,
"workTimeInMillis" : 234890,
"claimTime" : "2013-04-18T11:01:54.715+0000",
"taskDefinitionKey" : "taskKey",
"formKey" : null,
"priority" : 50,
"dueDate" : "2013-04-20T12:11:13.134+0000",
"parentTaskId" : null,
"url" : "http://localhost:8182/history/historic-task-instances/5",
"taskVariables": [
  {
    "name": "test",
    "variableScope": "local",
    "value": "myTest"
  }
],
"processVariables": [
  {
    "name": "processTest",
    "variableScope": "global",
    "value": "myProcessTest"
  }
],
"tenantId":null
},
],
"total": 1,
"start": 0,
"sort": "name",
"order": "asc",
"size": 1
}

```

15.8.13. #查询历史任务实例

POST query/historic-task-instances

查询历史任务实例 - 请求体:

```
{
  "processDefinitionId" : "oneTaskProcess%3A1%3A4",
  ...
  "variables" : [
    {
      "name" : "myVariable",
      "value" : 1234,
    }
  ]
}
```

```

        "operation" : "equals",
        "type" : "long"
    }
]
}

```

所有支持的JSON参数字段和获得历史任务实例集合完全一样，但是传递的是JSON参数，而不是URL参数，这样可以支持更高级的参数，同时避免请求uri过长。除此之外，查询支持基于流程变量查询。`taskVariables`和`processVariables`属性是一个json数组，包含此处描述的格式。

表 15.186. 查询历史任务实例 – 响应码

响应码	描述
200	表示请求成功，并返回任务。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id" : "5",
      "processDefinitionId" : "oneTaskProcess%3A1%3A4",
      "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
      "processInstanceId" : "3",
      "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
      "executionId" : "4",
      "name" : "My task name",
      "description" : "My task description",
      "deleteReason" : null,
      "owner" : "kermit",
      "assignee" : "fozzie",
      "startTime" : "2013-04-17T10:17:43.902+0000",
      "endTime" : "2013-04-18T14:06:32.715+0000",
      "durationInMillis" : 86400056,
      "workTimeInMillis" : 234890,
      "claimTime" : "2013-04-18T11:01:54.715+0000",
      "taskDefinitionKey" : "taskKey",
      "formKey" : null,
      "priority" : 50,
      "dueDate" : "2013-04-20T12:11:13.134+0000",
      "parentTaskId" : null,
      "url" : "http://localhost:8182/history/historic-task-instances/5",
      "taskVariables": [
        {
          "name": "test",
          "variableScope": "local",
          "value": "myTest"
        }
      ],
      "processVariables": [
        {
          "name": "processTest",
          "variableScope": "global",
          "value": "myProcessTest"
        }
      ]
    }
  ]
}
```

```

        }
    ]
}
],
"total": 1,
"start": 0,
"sort": "name",
"order": "asc",
"size": 1
}

```

15.8.14. #删除历史任务实例

```
DELETE history/historic-task-instances/{taskId}
```

表 15.187. 响应码

响应码	描述
200	表示似乎删除了历史任务实例。
404	表示找不到历史任务实例。

15.8.15. #获得历史任务实例的IdentityLink

```
GET history/historic-task-instance/{taskId}/identitylinks
```

表 15.188. 响应码

响应码	描述
200	表示请求成功，并返回了IdentityLink。
404	表示找不到任务实例。

成功响应体：

```

[
{
  "type" : "assignee",
  "userId" : "kermit",
  "groupId" : null,
  "taskId" : "6",
  "taskUrl" : "http://localhost:8182/history/historic-task-instances/5",
  "processInstanceId" : null,
  "processInstanceUrl" : null
}
]

```

15.8.16. #获取历史任务实例变量的二进制值

```
GET history/historic-task-instances/{taskId}/variables/{variableName}/data
```

表 15.189. 获取历史任务实例变量的二进制值 - 响应码

响应码	描述
200	表示找到了任务实例，并返回了请求的变量数据。
404	表示找不到任务实例，或任务实例不包含指定名称的变量，或变量没有有效的二进制流。状态信息包含了详细信息。

成功响应体： 响应体包含了变量的二进制值。当类型为 binary时，无论请求的accept-type头部设置了什么值，响应的content-type都为application/octet-stream。当类型为 serializable时， content-type为application/x-java-serialized-object。

15.8.17. #获取历史活动实例

```
GET history/historic-activity-instances
```

表 15.190. 获取历史活动实例 - URL参数

参数	是否必填	数据	描述
activityId	否	String	活动实例id。
activityInstanceId	否	String	历史活动实例id。
activityName	否	String	历史活动实例的名称。
activityType	否	String	历史活动实例的元素类型。
executionId	否	String	历史活动实例的分支id。
finished	否	Boolean	表示历史活动实例是否完成。
taskAssignee	否	String	历史活动实例的负责人。
processInstanceId	否	String	历史活动实例的流程实例id。
processDefinitionId	否	String	历史活动实例的流程定义id。
tenantId	否	String	只返回指定tenantId的实例。
tenantIdLike	否	String	只返回与指定tenantId匹配的实例
withoutTenantId	否	Boolean	如果为 true，只返回未设置tenantId的历史。 如果为 false，会忽略withoutTenantId 参数。

参数	是否必填	数据	描述
可以使用通用的 分页和排序查询参数。			

表 15.191. 获取历史活动实例 - 响应码

响应码	描述
200	表示反悔了查询的历史活动实例。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id": "5",
      "activityId": "4",
      "activityName": "My user task",
      "activityType": "userTask",
      "processDefinitionId": "oneTaskProcess%3A1%3A4",
      "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
      "processInstanceId": "3",
      "processInstanceUrl": "http://localhost:8182/history/historic-process-instances/3",
      "executionId": "4",
      "taskId": "4",
      "calledProcessInstanceId": null,
      "assignee": "fozzie",
      "startTime": "2013-04-17T10:17:43.902+0000",
      "endTime": "2013-04-18T14:06:32.715+0000",
      "durationInMillis": 86400056,
      "tenantId": null
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "name",
  "order": "asc",
  "size": 1
}
```

15.8.18. #查询历史活动实例

POST query/historic-activity-instances

请求体：

```
{
  "processDefinitionId": "oneTaskProcess%3A1%3A4"
}
```

所有支持的JSON参数字段和获得历史任务实例集合完全一样，但是传递的是JSON参数，而不是URL参数，这样可以支持更高级的参数，同时避免请求uri过长。

表 15.192. 查询历史活动实例 - 响应码

响应码	描述
200	表示请求成功，并返回了活动。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id" : "5",
      "activityId" : "4",
      "activityName" : "My user task",
      "activityType" : "userTask",
      "processDefinitionId" : "oneTaskProcess%3A1%3A4",
      "processDefinitionUrl" : "http://localhost:8182/repository/process-definitions/oneTaskProcess%3A1%3A4",
      "processInstanceId" : "3",
      "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/3",
      "executionId" : "4",
      "taskId" : "4",
      "calledProcessInstanceId" : null,
      "assignee" : "fozzie",
      "startTime" : "2013-04-17T10:17:43.902+0000",
      "endTime" : "2013-04-18T14:06:32.715+0000",
      "durationInMillis" : 86400056,
      "tenantId":null
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "name",
  "order": "asc",
  "size": 1
}
```

15.8.19. #列出历史变量实例

```
GET history/historic-variable-instances
```

表 15.193. 列出历史变量实例 - URL参数

参数	是否必填	数据	描述
processInstanceId	否	String	历史变量实例的流程实例id。
taskId	否	String	历史变量实例的任务id。
excludeTaskVariables	否	Boolean	表示从结果中排除任务变量。

参数	是否必填	数据	描述
variableName	否	String	历史变量实例的变量名称。
variableNameLike	否	String	对变量名称使用'like'操作历史变量实例。
可以使用通用的 分页和排序查询参数。			

表 15.194. 列出历史变量实例 - 响应码

响应码	描述
200	表示获得了历史变量实例。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id": "14",
      "processInstanceId": "5",
      "processInstanceUrl": "http://localhost:8182/history/historic-process-instances/5",
      "taskId": "6",
      "variable": {
        "name": "myVariable",
        "variableScope": "global",
        "value": "test"
      }
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "name",
  "order": "asc",
  "size": 1
}
```

15.8.20. #查询历史变量实例

```
POST query/historic-variable-instances
```

请求体：

```
{
  "processDefinitionId": "oneTaskProcess%3A1%3A4",
  ...
  "variables": [
```

```
{
  "variables": [
    {
      "name": "myVariable",
      "value": 1234,
      "operation": "equals",
      "type": "long"
    }
  ]
}
```

所有支持的JSON参数字段和获得历史变量实例集合完全一样，但是传递的是JSON参数，而不是URL参数，这样可以支持更高级的参数，同时避免请求uri过长。除此之外，查询支持基于流程变量查询。 variables属性是一个json数组，包含此处描述的格式。

表 15.195. 查询历史变量实例 - 响应码

响应码	描述
200	表示请求成功，并返回任务。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id": "14",
      "processInstanceId": "5",
      "processInstanceUrl": "http://localhost:8182/history/historic-process-instances/5",
      "taskId": "6",
      "variable": {
        "name": "myVariable",
        "variableScope": "global",
        "value": "test"
      }
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "name",
  "order": "asc",
  "size": 1
}
```

15.8.21. #获取历史任务实例变量的二进制值

```
GET history/historic-variable-instances/{varInstanceId}/data
```

表 15.196. 获取历史任务实例变量的二进制值 - 响应码

响应码	描述
200	表示找到了变量实例，并返回了请求的变量数据。

响应码	描述
404	表示找不到变量实例，或找不到对应名称的变量实例，或变量不包含合法的二进制流。状态信息包含了详细信息。

成功响应体： 响应体包含了变量的二进制值。当类型为 binary时，无论请求的accept-type头部设置了什么值，响应的content-type都为application/octet-stream。当类型为 serializable时， content-type为application/x-java-serialized-object。

15.8.22. #获取历史细节

```
GET history/historic-detail
```

表 15.197. 获取历史细节 – URL参数

参数	是否必填	数据	描述
id	否	String	历史细节的id。
processInstanceId	否	String	历史细节的流程实例id。
executionId	否	String	历史细节的分支id。
activityInstanceId	否	String	历史细节的活动实例id。
taskId	否	String	历史细节的任务id。
selectOnlyFormProperties	否	Boolean	表示结果中只返回FormProperties。
selectOnlyVariableUpdates	否	Boolean	表示结果中只返回变量更新信息。
可以使用通用的 分页和排序查询参数。			

表 15.198. 获取历史细节 – 响应码

响应码	描述
200	表示返回了查询的历史细节。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体：

```
{
  "data": [
    {
      "id" : "26",
      "processInstanceId" : "5",
      "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/5",
      "variables": [
        {
          "name": "var1",
          "value": "value1"
        }
      ]
    }
  ]
}
```

```

    "executionId" : "6",
    "activityInstanceId", "10",
    "taskId" : "6",
    "taskUrl" : "http://localhost:8182/history/historic-task-instances/6",
    "time" : "2013-04-17T10:17:43.902+0000",
    "detailType" : "variableUpdate",
    "revision" : 2,
    "variable" : {
        "name" : "myVariable",
        "variableScope", "global",
        "value" : "test"
    },
    "propertyId", null,
    "propertyValue", null
}
],
"total": 1,
"start": 0,
"sort": "name",
"order": "asc",
"size": 1
}

```

15.8.23. #查询历史细节

POST query/historic-detail

请求体:

```
{
    "processInstanceId" : "5",
}
```

所有支持的JSON参数字段和获得历史变量实例集合完全一样，但是传递的是JSON参数，而不是URL参数，这样可以支持更高级的参数，同时避免请求uri过长。

表 15.199. 查询历史细节 - 响应码

响应码	描述
200	表示请求成功，并返回了历史细节。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

成功响应体:

```
{
    "data": [
        {
            "id" : "26",
            "processInstanceId" : "5",
            "processInstanceUrl" : "http://localhost:8182/history/historic-process-instances/5",
            "executionId" : "6",
            "activityInstanceId", "10",
            "taskId" : "6",

```

```

    "taskUrl" : "http://localhost:8182/history/historic-task-instances/6",
    "time" : "2013-04-17T10:17:43.902+0000",
    "detailType" : "variableUpdate",
    "revision" : 2,
    "variable" : {
        "name" : "myVariable",
        "variableScope", "global",
        "value" : "test"
    },
    "propertyId", null,
    "propertyValue", null
}
],
"total": 1,
"start": 0,
"sort": "name",
"order": "asc",
"size": 1
}

```

15.8.24. #获取历史细节变量的二进制数据

GET history/historic-detail/{detailId}/data

表 15.200. 获取历史细节变量的二进制数据 – 响应码

响应码	描述
200	表示找到了历史细节实例，并返回了请求的变量数据。
404	表示找不到历史细节实例，或历史细节实例不包含指定名称的变量，或变量不包含合法的二进制流。状态信息包含了详细信息。

成功响应体： 响应体包含了变量的二进制值。当类型为 binary时，无论请求的accept-type头部设置了什么值，响应的content-type都为application/octet-stream。当类型为 serializable时， content-type为application/x-java-serialized-object。

15.9. #表单

15.9.1. #获取表单数据

GET form/form-data

表 15.201. 获取表单数据 – URL参数

参数	是否必填	数据	描述
taskId	是（如果没有processDefinitionId）	String	获取表单数据需要对应的任务id。
processDefinitionId	是（如果没有taskId）	String	获取startEvent表单数据需要对应的流程定义id。

表 15.202. 获取表单数据 - 响应码

响应码	描述
200	表示返回了查询的表单数据。
404	表示找不到表单数据。

成功响应体：

```
{
  "data": [
    {
      "formKey": null,
      "deploymentId": "2",
      "processDefinitionId": "3",
      "processDefinitionUrl": "http://localhost:8182/repository/process-definition/3",
      "taskId": "6",
      "taskUrl": "http://localhost:8182/runtime/task/6",
      "formProperties": [
        {
          "id": "room",
          "name": "Room",
          "type": "string",
          "value": null,
          "readable": true,
          "writable": true,
          "required": true,
          "datePattern": null,
          "enumValues": [
            {
              "id": "normal",
              "name": "Normal bed"
            },
            {
              "id": "kingsize",
              "name": "Kingsize bed"
            }
          ]
        }
      ],
      "total": 1,
      "start": 0,
      "sort": "name",
      "order": "asc",
      "size": 1
    }
  ]
}
```

15.9.2. #提交任务表单数据

POST form/form-data

任务表单的请求体：

```
{
```

```

"taskId" : "5",
"properties" : [
  {
    "id" : "room",
    "value" : "normal"
  }
]
}

```

startEvent表单的请求体:

```

{
  "processDefinitionId" : "5",
  "businessKey" : "myKey", (optional)
  "properties" : [
    {
      "id" : "room",
      "value" : "normal"
    }
  ]
}

```

表 15.203. 提交任务表单数据 - 响应码

响应码	描述
200	表示请求成功，并提交了表单数据。
400	表示传递了错误格式的参数。状态信息包含了详细信息。

startEvent表单数据的成功响应体（任务表单数据没有响应）：

```

{
  "id" : "5",
  "url" : "http://localhost:8182/history/historic-process-instances/5",
  "businessKey" : "myKey",
  "suspended" : false,
  "processDefinitionId" : "3",
  "processDefinitionUrl" : "http://localhost:8182/repository/process-definition/3",
  "activityId" : "myTask"
}

```

15.10. #数据库表

15.10.1. #表列表

```
GET management/tables
```

表 15.204. 表列表 - 响应码

响应码	描述
200	表示请求成功。

成功响应体:

```
[
  {
    "name": "ACT_RU_VARIABLE",
    "url": "http://localhost:8182/management/tables/ACT_RU_VARIABLE",
    "count": 4528
  },
  {
    "name": "ACT_RU_EVENT_SUBSCR",
    "url": "http://localhost:8182/management/tables/ACT_RU_EVENT_SUBSCR",
    "count": 3
  },
  ...
]
```

15.10.2. #获得一张表

```
GET management/tables/{tableName}
```

表 15.205. 获得一张表 – URL参数

参数	是否必须	值	描述
tableName	是	String	获取表的名称。

成功响应体:

```
{
  "name": "ACT_RE_PROCDEF",
  "url": "http://localhost:8182/management/tables/ACT_RE_PROCDEF",
  "count": 60
}
```

表 15.206. 获得一张表 – 响应码

响应码	描述
200	表示表存在，并返回了表的记录数。
404	表示表不存在。

15.10.3. #获得表的列信息

```
GET management/tables/{tableName}/columns
```

表 15.207. 获得表的列信息 – URL参数

参数	是否必须	值	描述
tableName	是	String	获取表的名称。

成功响应体:

```
{
  "tableName": "ACT_RU_VARIABLE",
  "columnNames": [
    "ID_",
    "REV_",
    "TYPE_",
    "NAME_",
    ...
  ],
  "columnTypes": [
    "VARCHAR",
    "INTEGER",
    "VARCHAR",
    "VARCHAR",
    ...
  ]
}
```

表 15.208. 获得表的列信息 – 响应码

响应码	描述
200	表示表存在，并返回了表的列信息。
404	表示表不存在。

15.10.4. #获得表的行数据

```
GET management/tables/{tableName}/data
```

表 15.209. 获得表的行数据 – URL参数

参数	是否必须	值	描述
tableName	是	String	获取表的名称。

表 15.210. 获得表的行数据 – URL参数

参数	是否必须	值	描述
start	否	Integer	从哪一行开始获取。默认为0。
size	否	Integer	获取行数，从start开始。默认为10。
orderAscendingColumn	否	String	对结果行进行排序的字段，正序。
orderDescendingColumn	否	String	对结果行进行排序的字段，倒序。

成功响应体：

```
{
```

```

"total":3,
"start":0,
"sort":null,
"order":null,
"size":3,

"data":[
{
    "TASK_ID_":"2",
    "NAME_":"var1",
    "REV_":1,
    "TEXT_":"123",
    "LONG_":123,
    "ID_":"3",
    "TYPE_":"integer"
},
...
]
}

```

表 15.211. 获得表的行数据 - 响应码

响应码	描述
200	表示表存在，并返回了行数据。
404	表示表不存在。

15.11. #引擎

15.11.1. #获得引擎属性

```
GET management/properties
```

返回引擎内部使用的只读属性。

成功响应体：

```
{
    "next.dbid":"101",
    "schema.history":"create(5.14)",
    "schema.version":"5.14"
}
```

表 15.212. 获得引擎属性 - 响应码

响应码	描述
200	表示返回了属性。

15.11.2. #获得引擎信息

```
GET management/engine
```

获得REST服务使用的引擎的只读信息。

成功响应体:

```
{
  "name":"default",
  "version":"5.14",
  "resourceUrl":"file:///activiti/activiti.cfg.xml",
  "exception":null
}
```

表 15.213. 获得引擎信息 - 响应码

响应码	描述
200	表示返回了引擎信息。

15.12. #运行时

15.12.1. #接收信号事件

POST runtime/signals

提醒引擎，接收了一个信号事件，不会特别针对某个流程。

JSON体:

```
{
  "signalName": "My Signal",
  "tenantId" : "execute",
  "async": true,
  "variables": [
    {"name": "testVar", "value": "This is a string"},
    ...
  ]
}
```

表 15.214. 接收信号事件 - JSON体参数

参数	描述	必填
signalName	signal的名称	是
tenantId	信号事件应该执行在的tenantId	否
async	如果为 true，处理信号应该是异步的。返回码为 202 – Accepted 表示请求已接受，但尚未执行。如果为 false，会立即处理信号，结果为 (200 – OK) 会在成功完成后返回。如果忽略，默认为 false。	否
variables	变量数组（通用的参数格式）用来向信号传递载荷。不能把	否

参数	描述	必填
	async 设置为 true, 它会导致返回错误。	

成功响应体:

表 15.215. 接收信号事件 - 响应码

响应码	描述
200	表示已经处理了信号, 没有发生错误。
202	表示信号处理已经进入一个异步作业的队列, 准备执行了。
400	信号没有处理。缺少信号名, 或同时使用了变量和异步, 这是不允许的。响应体包含了错误的额外信息。

15.13. #作业

15.13.1. #获取一个作业

```
GET management/jobs/{jobId}
```

表 15.216. 获取一个作业 - URL参数

参数	是否必须	值	描述
jobId	是	String	获取的作业id。

成功响应体:

```
{
  "id":"8",
  "url":"http://localhost:8182/management/jobs/8",
  "processInstanceId":"5",
  "processInstanceUrl":"http://localhost:8182/runtime/process-instances/5",
  "processDefinitionId":"timerProcess:1:4",
  "processDefinitionUrl":"http://localhost:8182/repository/process-definitions/timerProcess%3A1%3A4",
  "executionId":"7",
  "executionUrl":"http://localhost:8182/runtime/executions/7",
  "retries":3,
  "exceptionMessage":null,
  "dueDate":"2013-06-04T22:05:05.474+0000",
  "tenantId":null
}
```

表 15.217. 获取一个作业 - 响应码

响应码	描述
200	表示作业存在, 并成功返回。

响应码	描述
404	表示作业不存在。

15.13.2. #删除作业

```
DELETE management/jobs/{jobId}
```

表 15.218. 删除作业 – URL参数

参数	是否必须	值	描述
jobId	是	String	期望删除的作业id。.

表 15.219. 删除作业 – 响应码

响应码	描述
204	表示找到了作业，并成功删除。响应体为空。
404	表示找不到作业。

15.13.3. #执行作业

```
POST management/jobs/{jobId}
```

请求JSON体：

```
{
  "action" : "execute"
}
```

表 15.220. 执行作业 – 请求的JSON参数

参数	描述	是否必填
action	执行的操作。只支持execute。	是

表 15.221. 执行作业 – 响应码

响应码	描述
204	表示成功执行了操作。响应体为空。
404	表示找不到作业。
500	表示执行作业时出现了异常。状态描述包含了错误的详细信息。如果需要可以后续获取错误堆栈。

15.13.4. #获得作业的异常堆栈

```
GET management/jobs/{jobId}/exception-stacktrace
```

表 15.222. 获得作业的异常堆栈 - URL参数

参数	描述	是否必填
jobId	获取堆栈的作业id。	是

表 15.223. 获得作业的异常堆栈 - 响应码

响应码	描述
200	表示找到了作业，并返回了堆栈。响应包含了原始堆栈，Content-Type永远是text/plain。
404	表示找不到作业，或作业不包含错误堆栈。状态描述包含错误的详细信息。

15.13.5. #获得作业列表

```
GET management/jobs
```

表 15.224. 获得作业列表 - URL参数

参数	描述	类型
id	只返回指定id的作业。	String
processInstanceId	只返回指定id流程一部分的作业。	String
executionId	只返回指定id分支一部分的作业。	String
processDefinitionId	只返回指定流程定义id的作业。	String
withRetriesLeft	如果为 true，只返回尝试剩下的。如果为false，会忽略此参数。	Boolean
executable	如果为 true，只返回可执行的作业。如果为false，会忽略此参数。	Boolean
timersOnly	如果为 true，只返回类型为定时器的作业。如果为false，会忽略此参数。不能与'messagesOnly'一起使用。	Boolean
messagesOnly	如果为 true，只返回类型为消息的作业。如果为false，会忽略此参数。不能与'timersOnly'一起使用。	Boolean
withException	如果为 true，只返回执行时出现了异常的作业。如果为false，会忽略此参数。	Boolean

参数	描述	类型
dueBefore	只返回在指定时间前到期的作业。如果使用了这个参数，就不会返回没有设置持续时间的作业。	Date
dueAfter	只返回在指定时间后到期的作业。如果使用了这个参数，就不会返回没有设置持续时间的作业。	Date
exceptionMessage	只返回指定异常信息的作业	String
tenantId	只返回指定tenantId的作业。	String
tenantIdLike	只返回与指定tenantId匹配的作业。	String
withoutTenantId	如果为 true，只返回未设置 tenantId 的作业。如果为 false，会忽略 withoutTenantId 参数。	Boolean
sort	对结果进行排序的字段，可以是 id, dueDate, executionId, processInstanceId, retries 或 tenantId 其中之一。	String
可以使用通用的 分页和排序查询参数。		

成功响应体：

```
{
  "data": [
    {
      "id": "13",
      "url": "http://localhost:8182/management/jobs/13",
      "processInstanceId": "5",
      "processInstanceUrl": "http://localhost:8182/runtime/process-instances/5",
      "processDefinitionId": "timerProcess:1:4",
      "processDefinitionUrl": "http://localhost:8182/repository/process-definitions/timerProcess%3A1%3A4",
      "executionId": "12",
      "executionUrl": "http://localhost:8182/runtime/executions/12",
      "retries": 0,
      "exceptionMessage": "Can't find scripting engine for 'unexistinglanguage'",
      "dueDate": "2013-06-07T10:00:24.653+0000"
    },
    ...
  ],
  "total": 2,
  "start": 0,
  "sort": "id",
  "order": "asc",
}
```

```

    "size":2
}

```

表 15.225. 获得作业列表 - 响应码

响应码	描述
200	表示返回了作业。
400	表示在url参数中使用了非法的值，或参数中同时使用了'messagesOnly' 和 'timersOnly'。状态描述包含了错误的详细信息。

15.14. #用户

15.14.1. #获得一个用户

```
GET identity/users/{userId}
```

表 15.226. 获得一个用户 - URL参数

参数	是否必须	值	描述
userId	是	String	获取用户的id。

成功响应体：

```

{
  "id":"testuser",
  "firstName":"Fred",
  "lastName":"McDonald",
  "url":"http://localhost:8182/identity/users/testuser",
  "email":"no-reply@activiti.org"
}

```

表 15.227. 获得一个用户 - 响应码

响应码	描述
200	表示用户存在，并成功返回。
404	表示用户不存在。

15.14.2. #获取用户列表

```
GET identity/users
```

表 15.228. 获取用户列表 - URL参数

参数	描述	类型
id	只返回指定id的用户。	String
firstName	只返回指定firstname的用户。	String

参数	描述	类型
lastName	只返回指定lastname的用户。	String
email	只返回指定email的用户。	String
firstNameLike	只返回firstname与指定值匹配的用户。使用%通配符。	String
lastNameLike	只返回lastname与指定值匹配的用户。使用%通配符。	String
emailLike	只返回email与指定值匹配的用户。使用%通配符。	String
memberOfGroup	只返回指定组成员的用户。	String
potentialStarter	只返回指定流程定义id的默认启动人。	String
sort	结果排序的字段，应该是id, firstName, lastname 或 email其中之一。	String
可以使用通用的 分页和排序查询参数。		

成功响应体：

```
{
  "data": [
    {
      "id": "anotherUser",
      "firstName": "Tijs",
      "lastName": "Barrez",
      "url": "http://localhost:8182/identity/users/anotherUser",
      "email": "no-reply@alfresco.org"
    },
    {
      "id": "kermit",
      "firstName": "Kermit",
      "lastName": "the Frog",
      "url": "http://localhost:8182/identity/users/kermit",
      "email": null
    },
    {
      "id": "testuser",
      "firstName": "Fred",
      "lastName": "McDonald",
      "url": "http://localhost:8182/identity/users/testuser",
      "email": "no-reply@activiti.org"
    }
  ],
  "total": 3,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 3
}
```

```
}
```

表 15.229. 获取用户列表 - 响应码

响应码	描述
200	表示成功返回了请求的用户。

15.14.3. #更新用户

```
PUT identity/users/{userId}
```

请求JSON体:

```
{
  "firstName": "Tijs",
  "lastName": "Barrez",
  "email": "no-reply@alfresco.org",
  "password": "pass123"
}
```

所有请求值都是可选的。比如，你可以在请求体JSON对象中只包含'firstName'属性，只更新用户的firstName，其他值都不受影响。当包含的属性设置为null，用户的属性会被更新为null，比如: {"firstName": null}会清空用户的firstName。

表 15.230. 更新用户 - 响应码

响应码	描述
200	表示成功更新了用户。
404	表示找不到用户。
409	表示请求的用户被其他地方更新了。

成功响应体: 参考 `identity/users/{userId}` 的响应。

15.14.4. #创建用户

```
POST identity/users
```

请求JSON体:

```
{
  "id": "tijs",
  "firstName": "Tijs",
  "lastName": "Barrez",
  "email": "no-reply@alfresco.org",
  "password": "pass123"
}
```

表 15.231. 创建用户 - 响应码

响应码	描述
201	表示成功创建了用户。

响应码	描述
400	表示没有传用户的id。

成功响应体：参考 `identity/users/{userId}` 的响应。

15.14.5. #删除用户

```
DELETE identity/users/{userId}
```

表 15.232. 删除用户 – URL参数

参数	是否必填	数据	描述
userId	是	String	期望删除的用户id。

表 15.233. 删除用户 – 响应码

响应码	描述
204	表示找到了用户，并成功删除了。响应体为空。
404	表示找不到用户。

15.14.6. #获取用户图片

```
GET identity/users/{userId}/picture
```

表 15.234. 获取用户图片 – URL参数

参数	是否必填	数据	描述
userId	是	String	期望获得图片的用户id。

响应体：响应体包含了演示图片数据，展示用户的图片。响应的Content-Type对应着创建图片时设置的mimeType。

表 15.235. 获取用户图片 – 响应码

响应码	描述
200	表示找到了用户和图片，图片在响应体中返回。
404	表示找不到用户，或用户没有图片。状态描述包含错误的详细信息。

15.14.7. #更新用户图片

```
PUT identity/users/{userId}/picture
```

表 15.236. 更新用户图片 – URL参数

参数	是否必填	数据	描述
userId	是	String	获得图片对应的用户id。

请求体： 请求应该是multipart/form-data类型。应该只有一个文件区域，包含源码的二进制内容。除此之外，需要提供以下表单域：

- mimeType: 上传的图片的mime-type。如果省略，默认会使用 image/jpeg 作为图片的mime-type。

表 15.237. 更新用户图片 – 响应码

响应码	描述
200	表示找到了用户，并更新了图片。响应体为空。
404	表示找不到用户。

15.14.8. #列出用户列表

```
PUT identity/users/{userId}/info
```

表 15.238. 列出用户列表 – URL参数

参数	是否必填	数据	描述
userId	是	String	获取信息的用户id。

响应体：

```
[
  {
    "key": "key1",
    "url": "http://localhost:8182/identity/users/testuser/info/key1"
  },
  {
    "key": "key2",
    "url": "http://localhost:8182/identity/users/testuser/info/key2"
  }
]
```

表 15.239. 列出用户列表 – 响应码

响应码	描述
200	表示找到了用户，并返回了信息列表（key和url）。
404	表示找不到用户。

15.14.9. #获取用户信息

```
GET identity/users/{userId}/info/{key}
```

表 15.240. 获取用户信息 – URL参数

参数	是否必填	数据	描述
userId	是	String	获取信息的用户id。

参数	是否必填	数据	描述
key	是	String	希望获取的用户信息的key。

响应体:

```
{
  "key": "key1",
  "value": "Value 1",
  "url": "http://localhost:8182/identity/users/testuser/info/key1"
}
```

表 15.241. 获取用户信息 - 响应码

响应码	描述
200	表示找到了用户和指定key的用户信息。
404	表示找不到用户，并且没有指定key的信息。状态描述中包含了错误相关的详细信息。

15.14.10. #更新用户的信息

```
PUT identity/users/{userId}/info/{key}
```

表 15.242. 更新用户的信息 - URL参数

参数	是否必填	数据	描述
userId	是	String	期望更新的信息对应的用户id。
key	是	String	期望更新的用户信息的key。

请求体:

```
{
  "value": "The updated value"
}
```

响应体:

```
{
  "key": "key1",
  "value": "The updated value",
  "url": "http://localhost:8182/identity/users/testuser/info/key1"
}
```

表 15.243. 更新用户的信息 - 响应码

响应码	描述
200	表示找到了用户，并更新了信息。
400	表示请求体中缺少数据。
404	表示找到用户，或找不到指定key的用户信息。状态描述中包含了错误相关的详细信息。

15.14.11. #创建用户信息条目

```
POST identity/users/{userId}/info
```

表 15.244. 创建用户信息条目 - URL参数

参数	是否必填	数据	描述
userId	是	String	期望创建信息的用户id。

请求体:

```
{
  "key": "key1",
  "value": "The value"
}
```

响应体:

```
{
  "key": "key1",
  "value": "The value",
  "url": "http://localhost:8182/identity/users/testuser/info/key1"
}
```

表 15.245. 创建用户信息条目 - 响应码

响应码	描述
201	表示找到了用户，并创建了信息。
400	表示请求体中缺少key或value。状态描述中包含了错误相关的详细信息。
404	表示找不到用户。
409	表示用户已经有了一条指定key的信息条目，可以更新资源实例（PUT）。

15.14.12. #删除用户的信息

```
DELETE identity/users/{userId}/info/{key}
```

表 15.246. 删除用户的信息 – URL参数

参数	是否必填	数据	描述
userId	是	String	希望删除信息的用户id。
key	是	String	期望删除的用户信息的key。

表 15.247. 删除用户的信息 – 响应码

响应码	描述
204	表示找到了用户和信息，并删除了指定key的条目。响应体为空。
404	表示找不到用户，或用户不包含指定key的信息。状态描述中包含了错误相关的详细信息。

15.15. #群组

15.15.1. #获得群组

```
GET identity/groups/{groupId}
```

表 15.248. 获得群组 – URL参数

参数	是否必须	值	描述
groupId	是	String	希望获得的群组id。

成功响应体:

```
{
  "id": "testgroup",
  "url": "http://localhost:8182/identity/groups/testgroup",
  "name": "Test group",
  "type": "Test type"
}
```

表 15.249. 获得群组 – 响应码

响应码	描述
200	表示群组存在，并成功返回。
404	表示群组不存在。

15.15.2. #获取群组列表

```
GET identity/groups
```

表 15.250. 获取群组列表 – URL参数

参数	描述	类型
id	只返回指定id的群组。	String
name	只返回指定名称的群组。	String
type	只返回指定类型的群组。	String
nameLike	只返回名称与指定值匹配的群组 使用%作为通配符。	String
member	只返回成员与指定用户ing相同的群组。	String
potentialStarter	只返回成员作为指定id流程定义的潜在启动者的劝阻。	String
sort	结果排序的字段。应该是 id, name 或 type其中之一。	String
可以使用通用的 分页和排序查询参数。		

成功响应体：

```
{
  "data": [
    {
      "id": "testgroup",
      "url": "http://localhost:8182/identity/groups/testgroup",
      "name": "Test group",
      "type": "Test type"
    },
    ...
  ],
  "total": 3,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 3
}
```

表 15.251. 获取群组列表 – 响应码

响应码	描述
200	表示返回了请求的群组。

15.15.3. #更新群组

```
PUT identity/groups/{groupId}
```

请求JSON体：

```
{
  "name": "Test group",
  "type": "Test type"
}
```

所有请求值都是可选的。比如，你可以在请求体JSON对象中只包含' name' 属性，只更新群组的名称，其他属性都不会受到影响。如果把一个属性设置为null，群组的数据就会更新为null。

表 15.252. 更新群组 – 响应码

响应码	描述
200	表示成功更新了群组。
404	表示找不到请求的群组。
409	表示请求的群组正在被其他地方更新。

成功响应体：参考identity/groups/{groupId}的响应。

15.15.4. #创建群组

```
POST identity/groups
```

请求JSON体：

```
{
  "id": "testgroup",
  "name": "Test group",
  "type": "Test type"
}
```

表 15.253. 创建群组 – 响应码

响应码	描述
201	表示创建了群组。
400	表示丢失了群组的id。

成功响应体：参考 identity/groups/{groupId}的响应。

15.15.5. #删除群组

```
DELETE identity/groups/{groupId}
```

表 15.254. 删除群组 – URL参数

参数	是否必填	数据	描述
groupId	是	String	期望删除的群组id。

表 15.255. 删除群组 – 响应码

响应码	描述
204	表示找到了群组，并成功删除了。响应体为空。

响应码	描述
404	表示找不到请求的群组。

15.15.6. #获取群组的成员

identity/groups/members不允许使用GET。使用 identity/users?memberOfGroup=sales URL来获得某个群组下的所有成员。

15.15.7. #为群组添加一个成员

```
POST identity/groups/{groupId}/members
```

表 15.256. 为群组添加一个成员 - URL参数

参数	是否必填	数据	描述
groupId	是	String	期望添加成员的群组id。

请求JSON体：

```
{
  "userId": "kermit"
}
```

表 15.257. 为群组添加一个成员 - 响应码

响应码	描述
201	表示找到了群组，并添加了成员。
404	表示请求体中未包含userId。
404	表示找不到请求的群组。
409	表示请求的用户已经是群组的一员了。

响应体：

```
{
  "userId": "kermit",
  "groupId": "sales",
  "url": "http://localhost:8182/identity/groups/sales/members/kermit"
}
```

15.15.8. #删除群组的成员

```
DELETE identity/groups/{groupId}/members/{userId}
```

表 15.258. 删除群组的成员 - URL参数

参数	是否必填	数据	描述
groupId	是	String	期望删除成员的群组id。
userId	是	String	期望删除的用户id。

表 15.259. 删除群组的成员 - 响应码

响应码	描述
204	表示找到了群组，并删除了成员。响应体为空。
404	表示找不到请求的群组，或用户不是群组的成员。状态描述中包含了错误相关的详细信息。

响应体：

```
{
  "userId": "kermit",
  "groupId": "sales",
  "url": "http://localhost:8182/identity/groups/sales/members/kermit"
}
```

15.16. #传统REST - 通用方法

下面的章节中包含了传统的REST-api，它们在5.14版本中已经废弃了。REST url不会在未来删除，但是也不会维护了。未来的所有改进都会放到新REST API中。

Activiti包含了一套引擎的REST API，可以把activiti-rest.war部署到像Apache Tomcat一样的Servlet容器里。默认Activiti引擎会连接一个单独运行的h2数据库。你可以修改WEB-INF/classes目录下的db.properties文件来修改数据库配置。REST API使用JSON格式(<http://www.json.org>)，内部使用Restlet构建(<http://www.restlet.org>)。

每个REST API调用都会使用单独的校验级别，你必须登录到系统中，来调用REST API（除了/login服务）。认证是通过Basic HTTP认证实现的，如果你登录为管理员（比如kermit），你应该可以访问下面介绍的所有接口。

API遵守着通常的REST API约定，对读取操作使用GET，对创建对象使用POST，对更新和执行操作使用PUT，对删除对象使用DELETE。在执行应先过很多次对象的操作时，使用POST来执行这些操作来保证传递的参数不受限制。使用POST的原因是HTTP DELETE方法不允许使用请求体。使用DELETE的调用，理论上，它的请求体会被代理剔除。为了保证不发生这种事情，我们使用了POST，虽然PUT也可以更新多个对象，为了保持一致。

所有rest调用都使用“application/json”作为Content-Type（除了上传请求使用“multipart/form-data”）。

执行REST调用的url基础地址是<http://localhost:8080/activiti-rest/service/>。比如，列出引擎中流程定义的方法，在你的浏览器中应该是：<http://localhost:8080/activiti-rest/service/process-definitions>

你就也可以使用curl来通过REST API执行查询，比如：

```
curl --basic --user kermit:kermit http://localhost:8080/activiti-rest/service/tasks?assignee=kermit
```

请参考下面的描述，来了解哪些REST API是目前可用的。请参考“API”章节作为，“一行的注释”，来了解核心API的功能，实现REST API的调用。

15.17. #资源

15.17.1. #上传发布

上传并安装发布.bpmn20.xml, .bpmn, .bar 或 .zip格式，使用普通的“html表单上传”(enctype=multipart/form-data) 换句话说不是json请求。如果发布操作成功，发布信息会包含在返回的响应中。

- 请求： POST /deployment

- API：

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().createDeployment().name(f...)
```

- 响应：

```
{
  "id": "10",
  "name": "activiti-examples.bar",
  "deploymentTime": "2010-10-13T14:54:26.750+02:00",
  "category": "examples"
}
```

15.17.2. #获取发布

返回分页发布列表，可以通过“id”，“name”或“deploymentTime”排序。

- 请求： GET /deployments?start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}

- API：

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().createDeploymentQuery().li...
```

- 响应：

```
{
  "data": [
    {
      "id": "10",
      "name": "activiti-examples.bar",
      "deploymentTime": "2010-10-13T14:54:26.750+02:00",
      "category": "examples"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.17.3. #获取发布资源

返回发布的所有资源。 实例: /deployment/10/resources

- 请求: GET /deployment/{deploymentId}/resources

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().getDeploymentResourceNames()
```

- 响应:

```
{
  "id": "10",
  "name": "activiti-examples.bar",
  "deploymentTime": "2010-10-13T14:54:26.750+02:00",
  "category": "examples",
  "resources": ["resource1", "resource2"]
}
```

15.17.4. #获取发布的一个资源

获取发布的一个资源。 实例: /deployment/10/resource/org/activiti/examples/bpmn/usertask/FinancialReportProcess.bpmn20.xml

- 请求: GET /deployment/{deploymentId}/resource/{resourceName}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().getResourceAsStream(deploymentId, resourceName)
```

- 响应:

比如, 一个.bpmn20.xml文件, 一个图片或其他发布资源包含的文件类型。

15.17.5. #删除发布

删除一个发布。

- 请求: DELETE /deployment/{deploymentId}?cascade={cascade?}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().deleteDeployment(deploymentId, cascade)
```

- 响应:

```
{
  "success": true
}
```

15.17.6. #删除发布

删除多个发布。

- 请求: POST /deployments/delete?cascade={cascade?}

```
{
  "deploymentIds": [ "10", "11" ]
}
```

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().deleteDeployment(deploymentId, cascade);
```

- 响应:

```
{
  "success": true
}
```

15.18. #引擎

15.18.1. #获取流程引擎

返回流程引擎安装细节。如果启动时出现了问题，
的“exception”属性中。

错误的细节会包含在响应

- 请求: GET /process-engine

- API: ProcessEngines.getProcessEngine(configuredProcessEngineName)

- 响应:

```
{
  "name": "default",
  "resourceUrl": "jar:file:\\"<path-to-deployment>\activiti-cfg.jar!\activiti.properties",
  "exception": null,
  "version": "5.11"
}
```

15.19. #流程

15.19.1. #流程定义列表

返回发布的流程定义的信息，可以通过“id”，“name”，“version”或“deploymentId”排序。BPMN2.0 XML的流程图的名字包含在“resourceName”属性中，结合“deploymentId”属性，可以通过上面的获取发布资源REST API调用获得。

- 分页请求: GET /process-definitions?
start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRepositoryService().createProcessDefinitionQuery();
```

- 分页响应:

```
{
  "data": [
    {
      "id": "financialReport:1",
      "key": "financialReport",
      "version": 1,
      "name": "Monthly financial report",
      "resourceName": "org/activiti/examples/bpmn/usertask/FinancialReportProcess.bpmn20.xml",
      "diagramResourceName": "org/activiti/examples/bpmn/usertask/FinancialReportProcess.png",
      "deploymentId": "10",
      "startFormResourceKey": null,
      "isGraphicNotationDefined": true
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.19.2. #获得流程定义表单属性

返回流程定义表单属性。

- 请求: GET /process-definition/{processDefinitionId}/properties

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getFormService().getStartFormData(processDefinitionId)
```

- 响应:

```

        "data": [
{
  "id": "fullName",
  "name": "Full name",
  "value": "${name}",
  "type": "String",
  "required": false,
  "readable": true,
  "writeable": true
}
]
```

15.19.3. #获得流程定义表单资源

返回流程定义表单。

- 请求: GET /process-definition/{processDefinitionId}/form

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().getRenderedStartFormById(processDefinitionId)
```

- 响应:

```
<user-defined-response>
```

15.19.4. #获取流程定义图

如果存在，就返回一个流程定义的PNG图。

- 请求： GET /process-definition/{processDefinitionId}/diagram
- API： repositoryService.getResourceAsStream(processDefinition.getDeploymentId(), processDefinition.getDiagramResourceName());
- 响应：

```
流程定义的PNG图。
```

15.19.5. #启动流程实例

根据流程定义创建流程实例，返回新创建流程实例的细节信息。可以使用请求体对象传递额外的变量（通过表单）。换句话说，就是“processDefinitionId”属性旁边的属性。

注意如果提交的值为true（不是“true”），会被当做Boolean，即使没有使用描述符。这与数字的处理方式相同。比如，123会当做整数。“123”会当做字符串（除非使用描述符定义）。

- 请求： POST /process-instance

```
{
  "processDefinitionId": "financialReport:1:1700",
  "businessKey": "order-4711"
}
```

- API：
ProcessEngines.getProcessEngine(configuredProcessEngineName).getRuntimeService().startProcessInstanceById(processDefinitionId, businessKey) [, variables])
- 响应：

```
{
  "id": "217",
  "processDefinitionId": "financialReport:1:1700",
  "businessKey": "order-4711",
  "processInstanceId": "217"
}
```

15.19.6. #流程实例列表

返回活动的流程实例细节，可以通过“id”，“startTime”，“businessKey”或“processDefinitionId”排序。你可以使用“processDefinitionId” and “businessKey”进行查询。

- 分页请求： GET /process-instances? start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}&businessKey={businessKey}&processDefinitionId={processDefinitionId}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getHistoryService().createHistoricProcessInstance
```

- 分页响应:

```
{
  "data": [
    {
      "id": "2",
      "processDefinitionId": "financialReport:1",
      "businessKey": "55",
      "startTime": "2010-10-13T14:54:26.750+02:00",
      "startUserId": "kermit"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.19.7. #获得流程实例细节

返回指定流程实例的所有细节。可以是运行中或已结束的流程实例。

- 请求: GET /process-instance/{processInstanceId}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getHistoryService().createHistoricProcessInstance
```

- 响应:

```
{
  "id": "2",
  "processDefinitionId": "financialReport:1",
  "businessKey": "55",
  "startTime": "2010-10-13T14:54:26.750+02:00",
  "startActivityId": "startFinancialAnalysis",
  "startUserId": "kermit",
  "completed": false,
  "tasks": [
    {
      "taskId": "3",
      "taskName": "Analyze report",
      "owner": null,
      "assignee": "Kermit",
      "startTime": "2010-10-13T14:53:26.750+02:00",
      "completed": false
    }
  ],
  "activities": [
    {
      "activityId": "4",
      "activityName": "Get report",
      "activityType": "ServiceTask",
      "startTime": "2010-10-13T14:53:25.750+02:00",
      "completed": false
    }
  ]
}
```

```

        "completed": true,
        "startTime": "2010-10-13T14:53:25.950+02:00",
        "duration": 200
    }
],
"variables": [
    {
        "variableName": "reportName",
        "variableValue": "classified.pdf",
    }
]
"historyVariables": [
    {
        "variableName": "reportName",
        "variableValue": "classified.pdf",
        "variableType": "String",
        "revision": 1,
        "time": "2010-10-13T14:53:26.750+02:00"
    }
]
}

```

15.19.8. #获得流程实例图

返回高亮的活动流程PNG图。如果流程定义未包含DI信息就会返回404。

- 请求: GET /process-instance/{processInstanceId}/diagram
- API: ProcessDiagramGenerator.generateDiagram(pde, getRuntimeService().getActiveActivityIds(processInstanceId));
"png",
- 响应:

返回高亮的活动流程PNG图。

15.19.9. #获得流程实例的任务

返回运行中流程实例的任务列表。

- 请求: GET /process-instance/{processInstanceId}/tasks
- API: taskService.createTaskQuery().processInstanceId(processInstanceId);
- 分页响应:

```
{
    "data": [
        {
            "id": "127",
            "name": "Handle vacation request",
            "description": "Vacation request by Kermit",
            "delegationState": "pending",
            "dueDate": "2010-10-13T14:54:26.750+02:00",
            "priority": 50,
            "assignee": null,
            "executionId": "118",
        }
    ]
}
```

```

    "formResourceKey": "org/activiti/examples/taskforms/approve.form",
    "owner": "Kermit",
    "parentTaskId": "120",
    "processDefinitionId": "financialReport:1",
    "processInstanceId": "123",
    "taskDefinitionKey": "125"
  }
],
"total": 1,
"start": 0,
"sort": "id",
"order": "asc",
"size": 1
}

```

15.19.10. #继续特定流程实例的活动 (receiveTask)

继续一个活动分支 (receiveTask)。

- 请求: POST /process-instance/{processInstanceId}/signal

```

{
  "activityId": "receiveTask",
  "variable1": "value",
  "variable2": "value"
}

```

- API: runtimeService.signal(execution.getId(), variables);
- 响应:

```

{
  "success": true
}

```

15.19.11. #触发特定流程实例的信号

向特定流程实例发送一个信号，会触发所有订阅的信号事件。可以通过请求体发送额外的变量。如果你不想发送任何变量，可以使用空的请求体。

- 请求: POST /process-instance/{processInstanceId}/event/{signalName}

```

{
  "variable1": "value",
  "variable2": "value"
}

```

- API: runtimeService.signalEventReceived(signalName, execution.getId() [, variables]);
- 响应:

```

{
  "success": true
}

```

}

15. 20. #任务

15. 20. 1. #获得任务简介

为特定用户返回任务简介：分配给用户的任务数量，用户可以认领的未分配任务的数量， 用户作为成员的群组可以认领的未分配任务。

- 请求： GET /tasks-summary?user={userId}

- API：

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createTaskQuery().xxx().count()
```

- 响应：

```
{
  "assigned": {
    "total": 0
  },
  "unassigned": {
    "total": 1,
    "groups": [
      {
        "accountancy": 1,
        "sales": 0,
        "engineering": 0,
        "management": 0
      }
    ]
  }
}
```

15. 20. 2. #任务列表

返回分页的任务列表，可以同构“id”，“name”，“description”，“priority”，“assignee”，“executionId”或“processInstanceId”排序。列表必须基于特定角色的用户：负责人（分配给用户的任务列表）或候选人（用户可以领取的任务列表）或候选群组（用户作为成员的群组可以认领的任务列表）。如果任务通过“formResourceKey”属性指定了一个表单。任务的表单可以通过获取任务表单REST API获得。

- 分页请求： GET /tasks?[assignee={userId} | candidate={userId} | candidate-group={groupId}]&start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}

- 实例：

```
curl --basic --user kermit:kermit http://localhost:8080/activiti-rest/service/tasks?assignee=kermit
```

- API：

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createTaskQuery().xxx().listPage()
```

- 分页响应：

```
{
  "data": [
    {
      "id": "127",
      "name": "Handle vacation request",
      "description": "Vacation request by Kermit",
      "delegationState": "pending",
      "dueDate": "2010-10-13T14:54:26.750+02:00",
      "priority": 50,
      "assignee": null,
      "executionId": "118",
      "formResourceKey": "org/activiti/examples/taskforms/approve.form",
      "owner": "Kermit",
      "parentTaskId": "120",
      "processDefinitionId": "financialReport:1",
      "processInstanceId": "123",
      "taskDefinitionKey": "125"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.20.3. #获取任务

通过任务id获取任务的细节。

- 请求: GET /task/{taskId}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createTaskQuery().taskId(taskId).
```

- 响应:

```
{
  "id": "127",
  "name": "Handle vacation request",
  "description": "Vacation request by Kermit",
  "delegationState": "pending",
  "dueDate": "2010-10-13T14:54:26.750+02:00",
  "priority": 50,
  "assignee": null,
  "executionId": "118",
  "formResourceKey": "org/activiti/examples/taskforms/approve.form",
  "owner": "Kermit",
  "parentTaskId": "120",
  "processDefinitionId": "financialReport:1",
  "processInstanceId": "123",
  "taskDefinitionKey": "125",
  "subTaskList": [
    {
      "id": "129",
      "name": "Analyze request",
      "description": "Analyze request",
      "delegationState": "pending",
      "dueDate": "2010-10-13T14:54:26.750+02:00",
      "priority": 50
    }
  ]
}
```

```

    "priority": 50,
    "assignee": null,
    "executionId": "118",
    "owner": "Kermit",
    "parentTaskId": "127"
  }
],
"identityLinkList" : [
  {
    "type": "candidate",
    "userId": "Fozzie",
    "groupId": null
  }
],
"attachmentList" : [
  {
    "id": "130",
    "name": "vacation_request.xls",
    "description": "Vacation request",
    "type": "application/pdf",
    "url": null
  }
]
}

```

15.20.4. #获取任务表单

获取任务表单。

- 请求: GET /task/{taskId}/form

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().getRenderedTaskForm(taskId)
```

- 响应:

```
<user-defined-response>
```

15.20.5. #执行任务操作

对任务执行操作（认领，释放，分配或完成）。对于“完成”操作的额外变量（通过表单）可以通过请求体传递。要从表单读取更多变量，可以参考启动流程实例章节。

- 请求: PUT /task/{taskId}#[claim|unclaim|complete|assign] 认领和释放不需要JSON体，对于完成你可以提供一些变量，分配必须使用userId。例如，完成操作的请求体:

```
{
  "variableName1": "variableValue1",
  "variableName2": "variableValue2"
}
```

例如，分配操作的请求体:

```
{
  "userId": "newAssignee"
}
```

- API: ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().xxx(taskId ...)

- 响应:

```
{
  "success": true
}
```

15.20.6. #表单属性列表

返回运行中任务的表单的属性列表，由流程定义。

- 请求: GET /form/{taskId}/properties

- API:

ProcessEngines.getProcessEngine(configuredProcessEngineName).getFormService().getTaskFormData(taskId).getFormProperties()

- 响应:

```
{
  "data": [
    {
      "id": "userName",
      "name": "User",
      "value": "foobar",
      "type": "string",
      "required": "true",
      "readable": "true",
      "writable": "true"
    }
  ]
}
```

15.20.7. #为任务添加一个附件

为任务实例添加一个附件。

- 请求: PUT /task/{taskId}/attachment

```
{}
```

- API:

ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createAttachment(...)

- 响应:

```
{
  "id": "130",
  "name": "vacation_request.xls",
  "description": "Vacation request",
  "type": "application/pdf",
  "url": null
}
```

15. 20. 8. #获得任务附件

返回任务附件

- 请求: GET /attachment/{attachmentId}
- API: taskService.getAttachment(attachmentId);
- 响应:

附件。

15. 20. 9. #为任务添加一个url

为任务实例添加一个url

- 请求: PUT /task/{taskId}/url

{}

- API:

ProcessEngines.getProcessEngine(configuredProcessEngineName).getTaskService().createAttachment(...)

- 响应:

```
{
  "id": "130",
  "name": "google.com",
  "description": "Good search sitet",
  "type": null,
  "url": "http://www.google.com"
}
```

15. 21. #身份

15. 21. 1. #登录

认证一个用户。如果用户和密码不匹配，会返回403。如果认证成功，会返回200响应状态。

- 请求: POST /login

```
{
  "userId": "kermit",
  "password": "kermit"
}
```

- API:

ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().checkPassword(userId, password)

- 响应:

```
{
  "success": true
}
```

15.21.2. #获得用户

返回用户的细节。

- 请求: GET /user/{userId}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createUserQuery().userId(userId)
```

- 响应:

```
{
  "id": "kermit",
  "firstName": "Kermit",
  "lastName": "the Frog",
  "email": "kermit@server.com"
}
```

15.21.3. #列出用户的群组

返回用户对应的群组分页列表，可以使用“id”，“name”或“type”排序。

- 分页请求: GET /user/{userId}/groups?

```
start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}
```

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().xxx(userId)
```

- 分页响应:

```
{
  "data": [
    {
      "id": "admin",
      "name": "System administrator",
      "type": "security-role"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.21.4. #查询用户

返回用户列表，通过查询的文本搜索firstname和lastname。

- 分页请求: GET /users?searchText=ker

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createUserQuery().userFirstName("Kermit")
ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createUserQuery().userLastName("the Frog")
```

- 响应:

```
{
  data: [
    {
      "id": "kermit",
      "firstName": "Kermit",
      "lastName": "the Frog",
      "email": "kermit@server.com"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.21.5. #创建用户

创建一个新用户。

- 请求: PUT /user

```
{
  "id": "kermit",
  "firstName": "Kermit",
  "lastName": "the Frog",
  "email": "kermit@server.com",
  "password": "kermit"
}
```

- API: ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().newUser();
ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().saveUser();

- 响应:

```
{
  "success": true
}
```

15.21.6. #为群组添加用户

为群组添加用户，通过这个REST服务。

- 请求: POST /user/{userId}/groups

```
["management", "sales"]
```

- API: identityService().createMembership(userId, groupId);
- 响应:

```
{
  "success": true
}
```

15.21.7. #从群组删除用户

从群组删除用户。

- 请求: DELETE /user/{userId}/groups/{groupId}
- API: identityService().deleteMembership(userId, groupId);
- 响应:

```
{
  "success": true
}
```

15.21.8. #获得用户图片

返回用户图片

- 请求: GET /user/{userId}/picture
- API: identityService.getUserPicture(userId);
- 响应:

```
The user picture.
```

15.21.9. #获得群组

返回群组细节。

- 请求: GET /group/{groupId}
- API:
ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createGroupQuery().groupId(groupId)
- 响应:

```
{
  "id": "admin",
  "name": "System administrator",
  "type": "security-role"
}
```

15.21.10. #群组用户列表

返回一个群组的用户细节，可以通过“id”，“firstName”，“lastName”或“email”排序。

- 分页请求: GET /group/{groupId}/users

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getIdentityService().createUserQuery().memberOfGr...
```

- 分页响应:

```
{
  data: [
    {
      "id": "kermit",
      "firstName": "Kermit",
      "lastName": "the Frog",
      "email": "kermit@server.com"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.21.11. #查询群组

返回群组列表，通过查询的文本搜索id或name。

- 分页请求: GET /groups?searchText=ad

- API: identityService.createGroupQuery().list()

- 响应:

```
{
  data: [
    {
      "id": "admin",
      "name": "System administrator",
      "type": "security-role"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.21.12. #创建群组

创建新群组。

- 请求: PUT /group

```
{
```

```

    "id": "admin",
    "name": "System administrator",
    "type": "security-role"
}

```

- API: identityService.newGroup(); identityService.saveGroup();
- 响应:

```
{
  "success": true
}
```

15. 21. 13. #为群组添加用户

为群组添加用户。

- 请求: POST /group/{groupId}/users

```
["kermit", "fozzie"]
```

- API: identityService().createMembership(userId, groupId);
- 响应:

```
{
  "success": true
}
```

15. 21. 14. #为群组删除用户

为群组删除用户。

- 请求: DELETE /group/{groupId}/users/{userId}
- API: identityService().deleteMembership(userId, groupId);
- 响应:

```
{
  "success": true
}
```

15. 22. #管理

15. 22. 1. #作业列表

返回分页的作业列表，可以通过“id”， “process-instance-id”， “execution-id”， “due-date”， “retries” 或一些自定义的属性id排序。列表可以通过 process instance id, due date 或作业是否重试，是否可执行，只是消息或定时器来查询。

- 分页请求: GET /management/jobs?process-instance={processInstanceId}&with-retries-left={withRetriesLeft=false}&executable={executable=false}&only-timers={onlyTimers=false}&only-messages={onlyMessage=false}&dueDate-lt={iso8601Date}&dueDate-lte={iso8601Date}&dueDate-ht={iso8601Date}&dueDate-hlte={iso8601Date}&start={start=0}&size={size=10}&sort={sort=id}&order={order=asc}
- API: ProcessEngines.getProcessEngine(configuredProcessEngineName).createJobQuery().xxx().listPage()
- 分页响应:

```
{
  "data": [
    {
      "id": "212",
      "executionId": "211",
      "retries": -1,
      "processInstanceId": "210",
      "dueDate": null,
      "assignee": null,
      "exceptionMessage": "Can't find scripting engine for 'groovy'"
    }
  ],
  "total": 1,
  "start": 0,
  "sort": "id",
  "order": "asc",
  "size": 1
}
```

15.22.2. #获得作业

返回作业的细节。

- 请求: GET /management/job({jobId})
- API: ProcessEngines.getProcessEngine(configuredProcessEngineName).createJobQuery().id(jobId).singleResult()
- 响应:

```
{
  "id": "212",
  "executionId": "211",
  "retries": -1,
  "processInstanceId": "210",
  "dueDate": null,
  "assignee": null,
  "exceptionMessage": "Can't find scripting engine for 'groovy'",
  "stacktrace": "org.activiti.engine.ActivitiException: Can't find scripting engine for 'groovy'\n\tat ..."
}
```

15.22.3. #执行一个作业

执行一个作业。

- 请求: PUT /management/job/{jobId}/execute

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().executeJob(jobId)
```

- 响应:

```
{
  "success": true
}
```

15.22.4. #执行多个作业

执行多个作业。

- 请求: POST /management/jobs/execute

```
{
  "jobIds": [ "212" ]
}
```

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().executeJob(jobId)
```

- 响应:

```
{
  "success": true
}
```

15.22.5. #数据库表列表

返回引擎中的所有数据库表的元数据。

- 请求: GET /management/tables

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().getTableCount()
```

- 响应:

```
{
  "data": [
    {
      "tableName": "ACT_GE_PROPERTY",
      "noOfResults": 2
    }
  ]
}
```

15.22.6. #获得表元数据

返回一个数据库表的元数据。

- 请求: GET /management/table/{tableName}

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().getTableMetaData(tableName)
```

- 响应:

```
{
  "tableName": "ACT_GE_PROPERTY",
  "columnNames": ["REV_", "NAME_", "VALUE_"],
  "columnNames": ["class java.lang.Integer", "class java.lang.String", "class java.lang.String"]
}
```

15.22.7. #获得表数据

返回分页的数据库表数据列表。

- 分页请求: GET /management/table/{tableName}/data

- API:

```
ProcessEngines.getProcessEngine(configuredProcessEngineName).getManagementService().createTablePageQuery().tab
```

- 分页响应:

```
{
  "data": [
    {
      "NAME_": "schema.version",
      "REV_": "1",
      "VALUE_": "5.10"
    },
    {
      "NAME_": "next.dbid",
      "REV_": "4",
      "VALUE_": "310"
    }
  ],
  "total": 2,
  "start": 0,
  "sort": "NAME_",
  "order": "asc",
  "size": 2
}
```

第#16#章#集成CDI

activiti-cdi模块提供activiti的可配置型和cdi扩展。 activiti-cdi最突出的特性有：

- 支持@BusinessProcessScoped beans（绑定到流程实例的cdi bean），
- 流程为cdi bean支持自定义EL处理器，
- 使用注解为流程实例提供声明式控制，
- Activiti可以挂接在cdi事件总线上，
- 支持Java EE和Java SE，支持Spring，
- 支持单元测试。

```
<dependency>
<groupId>org.activiti</groupId>
<artifactId>activiti-cdi</artifactId>
<version>5.x</version>
</dependency>
```

16.1. #设置activiti-cdi

Activiti cdi可以安装在不同环境中。这里，我们会根据配置项一一讲解。

16.1.1. #查找流程引擎

cdi扩展需要访问到ProcessEngine。为实现此功能，使用org.activiti.cdi.spi.ProcessEngineLookup接口在运行期进行查找。 cdi模块使用默认的名为org.activiti.cdi.impl.LocalProcessEngineLookup的实现，它使用ProcessEngines这个工具类来查找ProcessEngine。默认配置下，使用ProcessEngines#NAME_DEFAULT来查找ProcessEngine。这个类可能是使用了自定义名称的子类。 注意：需要把activiti.cfg.xml放在classpath下。

Activiti cdi使用java.util.ServiceLoader SPI处理org.activiti.cdi.spi.ProcessEngineLookup的实例。为了提供接口的自定义实现，我们需要创建一个文本文件，名为 META-INF/services/org.activiti.cdi.spi.ProcessEngineLookup，在文件中我们需要指定实现的全类名。

注意

如果你没有提供自定义的org.activiti.cdi.spi.ProcessEngineLookup实现，activiti会使用默认的LocalProcessEngineLookup实现。这时，你所需要做的就是把activiti.cfg.xml放到classpath下（看下一章）。

16.1.2. #配置Process Engine

实际的配置依赖于选用的ProcessEngineLookup策略（参考上章）。这里，我们主要结合LocalProcessEngineLookup讨论可用的配置，这要求我们在classpath下提供一个spring的activiti.cfg.xml。

Activiti提供了不同的ProcessEngineConfiguration实现，主要是依赖实际使用的事务管理策略。 activiti-cdi模块对事务的要求不严格，意味着任何事务管理策略都可以使用（即

便是spring事务抽象层）。简单来讲，cdi模块提供两种自定义ProcessEngineConfiguration实现：

- org.activiti.cdi.CdiJtaProcessEngineConfiguration: activiti的JtaProcessEngineConfiguration的子类，用于在activiti使用JTA管理的事务环境。
- org.activiti.cdi.CdiStandaloneProcessEngineConfiguration: activiti的StandaloneProcessEngineConfiguration的子类，用于在activiti使用简单JDBC事务环境。

下面是JBoss 7下的activiti.cfg.xml文件的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- lookup the JTA-Transaction manager -->
    <bean id="transactionManager" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:jboss/TransactionManager"/>
        <property name="resourceRef" value="true" />
    </bean>

    <!-- process engine configuration -->
    <bean id="processEngineConfiguration"
          class="org.activiti.cdi.CdiJtaProcessEngineConfiguration">
        <!-- lookup the default Jboss datasource -->
        <property name="dataSourceJndiName" value="java:jboss/datasources/ExampleDS" />
        <property name="databaseType" value="h2" />
        <property name="transactionManager" ref="transactionManager" />
        <!-- using externally managed transactions -->
        <property name="transactionsExternallyManaged" value="true" />
        <property name="databaseSchemaUpdate" value="true" />
    </bean>
</beans>
```

这是Glassfish 3.1.1下的例子（假设已经配置了名为jdbc/activiti的datasource）：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- lookup the JTA-Transaction manager -->
    <bean id="transactionManager" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:appserver/TransactionManager"/>
        <property name="resourceRef" value="true" />
    </bean>

    <!-- process engine configuration -->
    <bean id="processEngineConfiguration"
          class="org.activiti.cdi.CdiJtaProcessEngineConfiguration">
        <property name="dataSourceJndiName" value="jdbc/activiti" />
        <property name="transactionManager" ref="transactionManager" />
        <!-- using externally managed transactions -->
        <property name="transactionsExternallyManaged" value="true" />
        <property name="databaseSchemaUpdate" value="true" />
    </bean>
```

```
</beans>
```

注意上面的额配置需要“spring-context”模块：

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>3.0.3.RELEASE</version>
</dependency>
```

在Java SE环境下的配置和创建ProcessEngine章节中提供的例子一样，
使用
“CdiStandaloneProcessEngineConfiguration”
替换
“StandaloneProcessEngineConfiguration”。

16.1.3. #发布流程

可以使用标准的activiti-api发布流程（RepositoryService）。另外，activiti-cdi提供自动发布 classpath下processes.xml中列出的流程的方式。下面是一个processes.xml文件的例子：

```
<?xml version="1.0" encoding="utf-8"?>
<!-- list the processes to be deployed -->
<processes>
<process resource="diagrams/myProcess.bpmn20.xml" />
<process resource="diagrams/myOtherProcess.bpmn20.xml" />
</processes>
```

16.2. #基于CDI环境的流程执行

这一章，我们简短了解activiti cdi扩展使用的基于环境的流程执行模型。BPMN业务流程通常是一个长时间运行的操作，包含了用户和系统任务的操作。运行过程中，流程会分成多个单独的工作单元，由用户和应用逻辑执行。在activiti-cdi中，流程实例可以分配到cdi环境中，关联展现成一个工作单元。这是非常有用的，如果工作单元太复杂，比如如果实现的用户任务是不同形式的复杂顺序，可以在这个操作中保持“non-process-scoped”状态。默认配置下，流程实例分配到“broadest”激活环境，就会启动交互，如果交互环境没有激活，就会返回到请求中。

16.2.1. #与流程实例进行关联交互

处理@BusinessProcessScoped beans，或注入流程变量时，我们实现了激活的cdi环境与流程实例的关联。Activiti-cdi提供了org.activiti.cdi.BusinessProcess bean来控制关联，特别是：

- startProcessBy*(...)方法，对应activiti的RuntimeService中的相关方法，允许启动和随后向关联的业务流程，
- resumeProcessById(String processInstanceId)，允许通过提供的id来关联流程实例，
- resumeTaskById(String taskId)，允许通过提供的id来关联任务（扩展情况下，也关联相应的流程实例），

一个工作单元（比如用户任务）完成后，completeTask()方法可以调用来解除流程实例和会话/请求的关联。这会通知activiti当前任务已经完成，并让流程实例继续执行。

注意，BusinessProcess bean是@Named bean，意思是导出的方法可以通过 表达式语言调用，比如在JSF页面中。下面的JSF 2 代码启动一个新的交互， 把它分配给一个用户任务实例， id作为一个请求参数传递（比如pageName.jsf?taskId=XX）：

```
<f:metadata>
<f:viewParam name="taskId" />
<f:event type="preRenderView" listener="#{businessProcess.startTask(taskId, true)}" />
</f:metadata>
```

16.2.2. #声明式流程控制

Activiti-cdi允许通过注解声明启动流程实例和完成任务。

@org.activiti.cdi.annotation.StartProcess注解允许 通过“key”或“name”启动流程实例。 注意流程实例会在注解的方法返回之后启动。比如：

```
@StartProcess("authorizeBusinessTripRequest")
public String submitRequest(BusinessTripRequest request) {
    // do some work
    return "success";
}
```

根据activiti的配置，注解方法的代码和启动流程实例 会在同一个事务中执行。
@org.activiti.cdi.annotation.CompleteTask事务的使用方式相同：

```
@CompleteTask(endConversation=false)
public String authorizeBusinessTrip() {
    // do some work
    return "success";
}
```

@CompleteTask注解可以结束当前会话。 默认行为会在activiti返回后结束会话。可以禁用结束会话的功能， 实例可以参考上述代码。

16.2.3. #在流程中引用bean

Activiti-cdi使用自定义解析器把CDI bean暴露到activiti E1中。这就可以在流程中引用这些bean：

```
<userTask id="authorizeBusinessTrip" name="Authorize Business Trip"
    activiti:assignee="#{authorizingManager.account.username}" />
```

“authorizingManager”可以是生产者方法提供的bean：

```
@Inject @ProcessVariable Object businessTripRequesterUsername;

@Produces
@Named
public Employee authorizingManager() {
    TypedQuery<Employee> query = entityManager.createQuery("SELECT e FROM Employee e WHERE e.account.username='"
        + businessTripRequesterUsername + "'", Employee.class);
    Employee employee = query.getSingleResult();
    return employee.getManager();
```

```
}
```

你可以使用同样的方法在服务任务中调用EJB的业务方法，
使
用activiti:expression="myEjb.method()"扩展。 注意，这要求在MyEjb类中使用@Named注解。

16.2.4. #使用@BusinessProcessScoped beans

使用activiti-cdi，bean的生命周期可以绑定到流程实例上。为了扩展，可以提供一个自定义的环境实现，命名为BusinessProcessContext。BusinessProcessScoped bean的实例会作为流程变量保存到当前流程实例中。BusinessProcessScoped bean需要是PassivationCapable（比如序列化）。下面是使用流程作用域bean的例子：

```
@Named
@BusinessProcessScoped
public class BusinessTripRequest implements Serializable {
    private static final long serialVersionUID = 1L;
    private String startDate;
    private String endDate;
    // ...
}
```

有时，我们需要使用流程作用域bean，没有与流程实例关联，比如启动流程之前。如果当前流程实例没有激活，BusinessProcessScoped bean实例会暂时保存在局部作用域里（比如，会话或请求，依赖环境。如果作用域后来与业务流程实例关联了，bean实例会刷新到流程实例里。）

16.2.5. #注入流程变量

流程变量可以实现用于注入。Activiti-CDI支持以下方式：

- @BusinessProcessScoped使用@Inject [附加修饰] 类型 属性名实现类型安全的注入
- 使用@ProcessVariable(name?)修饰符实现对类型不安全的流程变量的注入：

```
@Inject @ProcessVariable Object accountNumber;
@Inject @ProcessVariable("accountNumber") Object account
```

为了通过EL引用流程变量，我们可以简单实用如下方式：

- @Named @BusinessProcessScoped beans可以直接引用，
- 其他流程变量可以使用ProcessVariables bean来使用：

```
# {processVariables['accountNumber']}
```

16.2.6. #接收流程事件

[EXPERIMENTAL]

Activiti可以挂在CDI的事件总线上。这样我们可以使用标准CDI事件机制来监听流程事件。为了启用activiti的CDI事件支持，需要在配置中启用对应的解析监听器：

```
<property name="postBpmnParseHandlers">
<list>
<bean class="org.activiti.cdi.impl.event.CdiEventSupportBpmnParseHandler" />
```

```
</list>
</property>
```

现在activiti已经配置成使用CDI事件总线发布事件。下面给出了如何在CDI bean中处理事件的方式。在CDI，我们可以使用@Observes注解声明特定的事件监听器。事件监听是类型安全的。流程事件类型是org.activiti.cdi.BusinessProcessEvent。下面是一个简单事件监听方法的例子：

```
public void onProcessEvent (@Observes BusinessProcessEvent businessProcessEvent) {
    // handle event
}
```

监听器可以监听所有事件。如果想限制监听器接收的事件类型，我们可以添加修饰注解：

- **@BusinessProcess:** 限制指定流程定义的事件。比如：@Observes @BusinessProcess("billingProcess") BusinessProcessEvent evt
- **@StartActivity:** 限制指定环节的事件。比如：@Observes @StartActivity("shipGoods") BusinessProcessEvent evt 在进入id为"shipGoods"的环节时会触发。
- **@EndActivity:** 限制指定环节的事件。比如：@Observes @EndActivity("shipGoods") BusinessProcessEvent evt 在离开id为"shipGoods"的环节时会触发
- **@TakeTransition:** 限制指定连线的事件。

修饰命名可以自由组合。比如，为了接收"shipmentProcess"流程中所有离开"shipGoods"环节的事件，我们可以编写如下监听方法：

```
public void beforeShippingGoods (@Observes @BusinessProcess("shipmentProcess") @EndActivity("shipGoods") BusinessProcessEvent evt) {
    // handle event
}
```

默认配置下，事件监听器是同步调用，并在同一个事务环境中。CDI事务性监听器（只在JavaEE / EJB环境下有效），可以控制监听器什么时候处理事件，比如，我们可以保证监听器只在事件中的事务成功之后才处理：

```
public void onShipmentSucceeded (@Observes (during=TransactionPhase.AFTER_SUCCESS) @BusinessProcess("shipmentProcess") BusinessProcessEvent evt) {
    // send email to customer.
}
```

16.2.7. #更多功能

- 流程引擎和服务都可以注入：@Inject ProcessEngine, RepositoryService, TaskService, ...
- 当前流程实例和任务可以注入：@Inject ProcessInstance, Task,
- 当前业务标识可以注入：@Inject @BusinessKey String businessKey,
- 当前流程实例id可以注入：@Inject @ProcessInstanceId String pid,

16.3. #已知的问题

虽然activiti-cdi已经使用了SPI，并设计为“可移植扩展”，但是只在Weld下测试过。

第#17#章#集成LDAP

企业通常已经在LDAP系统各种保存了用户和群组信息。自从5.14版本开始，Activiti提供了一种解决方案，通过简单的配置就可以告知activiti如何连接LDAP。

在Activiti 5.14之前，Activiti就已经可以集成LDAP了。然后，5.14的配置简单了很多。不过，配置LDAP的“老”办法依然有效。更确切的说，简化的配置其实是基于“老”方法的封装。

17.1. #用法

要想在你的项目中集成LDAP，在pom.xml中添加如下依赖：

```
<dependency>
    <groupId>org.activiti</groupId>
    <artifactId>activiti-ldap</artifactId>
    <version>latest.version</version>
</dependency>
```

17.2. #用例

集成LDAP目前有两大用例：

- 通过IdentityService进行认证。比如，使用Activiti Explorer 通过LDAP登录。
- 获得用户的组。这在查询用户可以看到哪些任务时非常重要。（比如，任务分配给一个候选组）。

17.3. #配置

集成LDAP是通过向流程引擎配置章节中的configurators注入 org.activiti.ldap.LDAPConfigurator的实例来实现的。这个类是高度可扩展的：如果默认的实现不符合用例的话，可以很容易的重写方法，很多依赖的bean都是可插拔的。

这时一个实例配置（注意：当然，通过代码创建引擎时，是非常简单的）。现在不用担心所有参数，我们会在下一章详细讨论。

```
<bean id="processEngineConfiguration" class="... SomeProcessEngineConfigurationClass">
    ...
    <property name="configurators">
        <list>
            <bean class="org.activiti.ldap.LDAPConfigurator">
                <!-- Server connection params -->
                <property name="server" value="ldap://localhost" />
                <property name="port" value="33389" />
                <property name="user" value="uid=admin, ou=users, o=activiti" />
                <property name="password" value="pass" />
                <!-- Query params -->
            
        
    

```

```

<property name="baseDn" value="o=activiti" />
<property name="queryUserById" value="(&(objectClass/inetOrgPerson)(uid={0}))" />
<property name="queryUserByFullNameLike" value="(&(objectClass/inetOrgPerson)(|({0}={1}*)({2}=*))" />
<property name="queryGroupsForUser" value="(&(objectClass=groupOfUniqueNames)(uniqueMember={0}))" />

<!-- Attribute config -->
<property name="userIdAttribute" value="uid" />
<property name="userFirstNameAttribute" value="cn" />
<property name="userLastNameAttribute" value="sn" />

<property name="groupIdAttribute" value="cn" />
<property name="groupNameAttribute" value="cn" />

</bean>
</list>
</property>
</bean>

```

17.4. #属性

下面是org.activiti.ldap.LDAPConfigurator可以配置的属性：

表 17.1. LDAP配置属性

属性名	描述	类型	默认值
server	LDAP服务器地址。比如'ldap://localhost:33389'	String	
port	LDAP运行的端口	int	
user	连接LDAP使用的账号	String	
password	连接LDAP使用的密码	String	
initialContextFactory	连接LDAP使用的InitialContextFactory名称	String	com.sun.jndi.ldap.LdapCtxFactory
securityAuthentication	连接LDAP时设置的'java.naming.security.authentication'属性值	String	simple
customConnectionParameters	可以设置那些没有对应setter的连接参数。参考 http://docs.oracle.com/javase/tutorial/jndi/ldap/jndi.html 中的自定义属性。这些属性用来配置连接池，特定的安全设置，等等。所有提供的参数都会用来创建LDAP连接。	Map<String, String>	

属性名	描述	类型	默认值
baseDn	搜索用户和组的基“显著名称”(DN)	String	
userBaseDn	搜索用户基于' distinguished name'(DN)。如果没有提供，会使用baseDn(参考上面)	String	
groupBaseDn	搜索群组基于' distinguished name'(DN)。如果没有提供，会使用baseDn(参考上面)	String	
searchTimeLimit	搜索LDAP的超时时间，单位毫秒。	long	一小时
queryUserById	使用用户id搜索用户的查询语句。比如: (&(objectClass=inetOrgPerson)(uid={0})) 这里，LDAP中所有包含'inetOrgPerson'类的匹配'uid'属性的值都会返回。如例子中所示，{0}会被用户id替换。如果只设置一个查询无法满足特定的LDAP设置，可以选择使用LDAPQueryBuilder，这样就会提供比单纯使用查询增加更多功能。	string	
queryUserByFullNameLike	使用全名搜索用户的查询语句。比如：(&(objectClass=inetOrgPerson)(({0}={1}*)({2}={3}*)) 这里，LDAP中所有包含'inetOrgPerson'类的匹配first name和last name的值都会返回。注意{0}会替换为firstNameAttribute(如上所示)，{1}和{3}是搜索内容，{2}是lastNameAttribute。如果只设置一个查询无	string	

属性名	描述	类型	默认值
	法满足特定的LDAP设置， 可以选择使用 LDAPQueryBuilder， 这样就会提供比单纯使用查询增加更多功能。		
queryGroupsForUser	使用搜索指定用户的组的查询语句。 比如： (&(objectClass=groupOfUniqueNames)(uniqueMember={0})) 这里， LDAP中所有包含' groupOfUniqueNames' 类的 提供的DN（匹配用户的DN） 是' uniqueMember' 的记录都会返回。 像例子中演示的那样， {0}会替换为用户id。 如果只设置一个查询无法满足特定的LDAP设置， 可以选择使用 LDAPQueryBuilder， 这样就会提供比单纯使用查询增加更多功能。	string	
userIdAttribute	匹配用户id的属性名。这个属性用来在查找用户对象时 关联LDAP对象与Activiti用户对象之间的关系。	string	
userFirstNameAttribute	匹配first name的属性名。 这个属性用来在查找用户对象时 关联LDAP对象与Activiti用户对象之间的关系。	string	
userLastNameAttribute	匹配last name的属性名。 这个属性用来在查找用户对象时 关联LDAP对象与Activiti用户对象之间的关系。	string	
groupIdAttribute	匹配组id的属性名。 这个属性用来在查找组对象时 关联LDAP对象与Activiti组对象之间的关系。	string	

属性名	描述	类型	默认值
groupNameAttribute	匹配组名的属性名。 这个属性用来在查找组对象时 关联LDAP对象与 Activiti组对象之间的关系。	String	
groupTypeAttribute	匹配组名的属性类型。这个属性用来在查找组对象时 关联LDAP对象与 Activiti组对象之间的关系。	String	

下列属性用在希望修改默认行为 或修改组缓存的情况:

表 17.2. 高级属性

属性名	描述	类型	默认值
ldapUserManagerFactory	设置 LDAPUserManagerFactory 的实例的自定义实例，如果默认实现不满足需求。	LDAPUserManagerFactory	
ldapGroupManagerFactory	设置 LDAPGroupManagerFactory 的实例的自定义实例，如果默认实现不满足需求。	LDAPGroupManagerFactory	
ldapMemberShipManagerFactory	设置 LDAPMembershipManagerFactory 的实例的自定义实例，如果默认实现不满足需求。 注意它不常用，因为正常情况下LDAP会自己管理关联关系。	LDAPMembershipManagerFactory	
ldapQueryBuilder	设置自定义查询构造器，如果默认实现不满足需求。 LDAPQueryBuilder实例用在LDAPUserManager 和LDAPGroupManager 中， 执行对LDAP的查询。 默认实现会使用配置的 queryGroupsForUser 和 queryUserById 属性。	org.activiti.ldap.LDAPQueryBuilder 的实例	
groupCacheSize	组缓存的大小。 这是一个LRU缓存，用来缓存用户的组， 可以避免每	int	-1

属性名	描述	类型	默认值
	次查询用户的组时，都要访问LDAP。如果值小于0，就不会创建缓存。默认为-1，所以不会进行缓存。		
groupCacheExpirationTime	设置组缓存的过期时间，单位为毫秒。当获取特定用户的组时，并且组缓存也启用了，组会保存到缓存中，并使用这个属性设置的时间。例如，当组在00:00被获取，过期时间为30分钟，那么所有在00:30之后进行的查询都不会使用缓存，而是再次去LDAP查询。因此，所以在00:00 - 00:30 进行的查询都会使用缓存。	long	1小时

注意，在使用活动目录（AD）时：Activiti论坛中的人们反映对于活动目录（AD），'InitialDirContext' 需要设置为Context.REFERRAL。可以通过上面描述的customConnectionParameters传递。

17.5. #为Explorer集成LDAP

- 把上面的LDAP配置添加到activiti-standalone-context.xml中。
- 把activiti-ldap jar放到WEB-INF/lib目录下
- 删除demoDataGenerator bean，因为它会尝试插入数据（集成LDAP不允许这么做）
- 将下面的配置添加到activiti-ui.context的explorerApp bean中：

```

<property name="adminGroups">
  <list>
    <value>admin</value>
  </list>
</property>
<property name="userGroups">
  <list>
    <value>user</value>
  </list>
</property>

```

请使用你自己的配置替换其中的值。需要用到的数据是组的id（通过groupIdAttribute配置）。上述配置会让'admin'组下的所有用户都成为Activiti Explorer的管理员，用户组也一样。所有不匹配的组都会当做“分配”组，这样任务就可以分配给他们。

第#18#章#高级功能

下面内容将介绍使用Activiti的高级用例，它会超越BPMN 2.0流程的范畴。因此，对于Activiti的明确目标和经验有利于理解这里的内容。

18.1. #监听流程解析

bpmn 2.0 xml文件需要被解析为Activiti内部模型，然后才能在Activiti引擎中运行。解析过程发生在发布流程或在内存中找不到对应流程的时候，这时会从数据库查询对应的xml。

对于每个流程，BpmnParser类都会创建一个新的BpmnParse实例。这个实例会作为解析过程中的容器来使用。解析过程很简单：对于每个BPMN 2.0元素，引擎中都会有一个对应的org.activiti.engine.parse.BpmnParseHandler实例。这样，解析器会保存一个BPMN 2.0元素与BpmnParseHandler实例的映射。默认，Activiti使用BpmnParseHandler来处理所有支持的元素，也使用它来提供执行监听器，以支持流程历史。

可以向Activiti引擎中添加自定义的org.activiti.engine.parse.BpmnParseHandler实例。经常看到的用例是把执行监听器添加到对应的环节，来处理一些事件队列。Activiti在内部就是这样进行历史处理的。要想添加这种自定义处理器，需要为Activiti添加如下配置：

```
<property name="preBpmnParseHandlers">
    <list>
        <bean class="org.activiti.parsing.MyFirstBpmnParseHandler" />
    </list>
</property>

<property name="postBpmnParseHandlers">
    <list>
        <bean class="org.activiti.parsing.MySecondBpmnParseHandler" />
        <bean class="org.activiti.parsing.MyThirdBpmnParseHandler" />
    </list>
</property>
```

配置到preBpmnParseHandlers的BpmnParseHandler实例会添加在默认处理器的前面。与之类似，postBpmnParseHandlers会加在后面。当自定义处理器内部逻辑对处理顺序有要求时就很重要了。

org.activiti.engine.parse.BpmnParseHandler是一个很简单的接口：

```
public interface BpmnParseHandler {
    Collection<Class<? extends BaseElement>> getHandledTypes();
    void parse(BpmnParse bpmnParse, BaseElement element);
}
```

getHandledTypes()方法会翻译这个解析器处理的所有类型的集合。它们都是BaseElement的子类，返回集合的泛型限制也说明了这一点。你也可以继承AbstractBpmnParseHandler类并重写getHandledType()方法，这样就只需要返回一个类型，而不是一个集合。这个类也包含了需要默认解析处理器所需要的帮助方法。BpmnParseHandler实例只有在解析器访问到这个方

法返回的类型时才会被调用。 在下面的例子中，当BPMN 2.0 xml包含process元素时，就会执行executeParse方法中的逻辑（这是一个已经完成类型转换的方法，它替换了BpmnParseHandler接口中的parse方法。）

```
public class TestBPMNParseHandler extends AbstractBpmnParseHandler<Process> {

    protected Class<? extends BaseElement> getHandledType() {
        return Process.class;
    }

    protected void executeParse(BpmnParse bpmnParse, Process element) {
        ...
    }
}
```

重要提示：在编写自定义解析处理器时，不要使用任何解析BPMN 2.0结构的内部类。这会很难找到问题。安全的方法是实现BpmnParseHandler接口或集成内部抽象类org.activiti.engine.impl.bpmn.parser.handler.AbstractBpmnParseHandler。

可以（但不常用）替换默认的BpmnParseHandler实例 把解析BPMN 2.0元素解析为Activiti内部模型。可以通过下面的代码来实现：

```
<property name="customDefaultBpmnParseHandlers">
    <list>
        ...
    </list>
</property>
```

举个简单的例子，强行把所有服务任务都设置为异步的：

```
public class CustomUserTaskBpmnParseHandler extends ServiceTaskParseHandler {

    protected void executeParse(BpmnParse bpmnParse, ServiceTask serviceTask) {

        // Do the regular stuff
        super.executeParse(bpmnParse, serviceTask);

        // Make always async
        ActivityImpl activity = findActivity(bpmnParse, serviceTask.getId());
        activity.setAsync(true);
    }
}
```

18.2. #支持高并发的UUID id生成器

在一些（非常）高并发的场景，默认的id生成器可能因为无法很快的获取新id区域而导致异常。所有流程引擎都有一个id生成器。默认的id生成器会在数据库划取一块id范围，这样其他引擎就不能使用相同范围的id。在引擎启动期间，当默认的id生成器发现已经越过id范围时，就会启动一个新事务来获得新范围。在（非常）极限的情况下，这

会在非常高负载的情况下导致问题。对于大部分情况，默认id生成就足够了。默认的org.activiti.engine.impl.db.DbIdGenerator也有一个idBlockSize属性，可以配置获取id范围的大小，这样可以改变获取id的行为。

另一个可以选用的默认id生成器是org.activiti.engine.impl.persistence.StrongUuidGenerator，它会在本地生成一个唯一的UUID [http://en.wikipedia.org/wiki/Universally_unique_identifier]，把它作为所有实体的标识。因为生成UUID不需要访问数据库，所以它在高并发环境下的表现比较好。要注意默认id生成器的性能（无论好坏）都依赖于运行硬件。

UUID生成器可以像下面这样配置到activiti中：

```
<property name="idGenerator">
    <bean class="org.activiti.engine.impl.persistence.StrongUuidGenerator" />
</property>
```

使用UUID id生成器需要以下依赖：

```
<dependency>
    <groupId>com.fasterxml.uuid</groupId>
    <artifactId>java-uuid-generator</artifactId>
    <version>3.1.3</version>
</dependency>
```

18.3. #多租户

多租户通常是在软件需要为多个不同组织服务时产生的概念。关键是数据分片，组织不能看到其他组织的数据。这种场景下，组织（或部门，或小组，或。。。）就叫做租户。

注意它和安装多个实例是从基本上不同的，这是一个activiti流程引擎实例为每个组织分别运行（对应不同的数据库表）。虽然activiti是轻量级的，运行流程引擎不会消耗很多资源，但是它还是增加了复杂性，并需要更多维护工作。但是，对一些场景，可能也是正确的解决方案。

activiti的多租户主要围绕着数据分片来实现。很重要的一点是，Activiti没有强行校验多租户的规则。这意味着不会校验，查询和使用数据时用户是否使用了正确的租户。这应该由activiti引擎的调用者一层负责完成。activiti只确认租户信息会被保存，并在查询流程数据时会被用到。

在向activiti流程引擎发布流程定义时，可能需要传递一个租户标识。它是一个字符串（比如，UUID，部门ID，等等），限制在256字符内，租户的唯一标识。

```
repositoryService.createDeployment()
    .addClassPathResource(...)
    .tenantId("myTenantId")
    .deploy();
```

通过部署传递租户id有以下含义：

- 所有包含在部署中的流程定义都会继承部署的tenantId。
- 所有从这些流程定义发起的流程实例，都会继承流程定义的tenantId。

- 所有流程实例运行阶段创建的任务都会继承流程实例的tenantId。单独运行的task也可以包含tenantId。
- 所有流程实例运行阶段创建的分支都会继承流程实例的tenantId。
- 触发一个信号抛出事件（在流程本身或通过API）可以通过tenantId实现。信号指挥在租户环境下执行：比如，如果有多个信号捕获事件，并且名字相同，实际只有正确的tenantId下的事件会被调用。
- 所有作业（定时器，异步调用）会集成tenantId，或者来自流程定义（比如定时开始事件），或流程实例（运行期创建的作业，比如异步调用）。这样其实潜在的可以支持为一些租户指定不同优先级的自定义jobExecutor。
- 所有历史实体（历史流程实例，任务和节点）会从它们对应的运行状态集成tenantId。
- 作为单独的一部分，model也可以设置tenantId（model用来存储Activiti modeler设计的bpnn 2.0模型）。

为了确实为流程数据使用tenantId，所有的查询API都可以通过tenantId进行查询。比如（可以使用其他的实体的对应查询实现替换）：

```
runtimeService.createProcessInstanceQuery()
    .processInstanceId("myProcessId")
    .processDefinitionKey("myProcessDefinitionKey")
    .variableValueEquals("myVar", "someValue")
    .list()
```

查询API也允许对tenantId使用like语法，也可以过滤未设置tenantId的实体。

重要的实现细节：因为数据库的限制（特指：处理null的唯一校验）默认的表示未设置租户的tenantId的值是空字符串。（流程定义key，流程定义version，tenantId）的组合应该是唯一的（有一个数据库约束校验这个规则）。也要注意tenantId不应设置为null，它会影响一些数据库（Oracle）的查询，它把空字符串当做null处理。（这也是为什么.withoutTenantId查询会检查空字符串或null）。这意味着相同的流程定义（流程定义key相同）可以部署到不同的租户下，可以拥有各自的副本。当不使用租户时也不会影响使用。

注意上面介绍的所有内容都不会影响Activiti在集群环境下运行。

[试验项目] 可以通过调用repositoryService的changeDeploymentTenantId(String deploymentId, String newTenantId)修改tenantId。它会修改之前继承的所有tenantId。当我们从非多租户环境向多租户环境下切换时，就会非常实用了。参考方法的javadoc获得更多细节信息。

18.4. #执行自定义SQL

Activiti API允许使用高级API操作数据库。比如，在查询数据方面，查询API和Native Query API是非常强大的。然而，对于某些情况，它们还是不够轻便。下面的章节描述了如何使用完全自定义的SQL语句（select, insert, update和delete）可以执行在Activiti的数据存储之上，但是完全又是配置在流程引擎中的（比如使用事务）。

为了使用自定义SQL，activiti引擎使用下层框架的功能，MyBatis。使用自定义SQL的第一件事，要创建MyBatis映射类。可以阅读 MyBatis用户手册 [http://mybatis.github.io/]

[mybatis-3/java-api.html]了解更多信息。比如，假设不需要全部的任务数据，只需要其中的一小部分。可以使用Mapper实现，如下：

```
public interface MyTestMapper {
    @Select("SELECT ID_ as id, NAME_ as name, CREATE_TIME_ as createTime FROM ACT_RU_TASK")
    List<Map<String, Object>> selectTasks();
}
```

这个mapper需要像下面这样设置到流程引擎配置中：

```
...
<property name="customMybatisMappers">
    <set>
        <value>org.activiti.standalone.cfg.MyTestMapper</value>
    </set>
</property>
...
```

注意，这是一个接口。MyBatis框架会在运行阶段为它创建一个实例。还要注意返回值是没有类型的，但是map的list（和对应的行列对应）。如果需要也可以使用MyBatis映射。

为了执行上面的查询，可以使用managementService.executeCustomSql方法。这个方法需要一个CustomSqlExecution实体。只是一个封装类，隐藏了引擎的内部实现所需执行的信息。

不幸的是，java泛型让它有点儿不易阅读。下面的两个泛型是mapper类和返回类型类。然而，真实的落实是简单调用mapper方法并返回结果（如果有对应的话）。

```
CustomSqlExecution<MyTestMapper, List<Map<String, Object>>> customSqlExecution =
    new AbstractCustomSqlExecution<MyTestMapper, List<Map<String, Object>>>(MyTestMapper.class) {

    public List<Map<String, Object>> execute(MyTestMapper customMapper) {
        return customMapper.selectTasks();
    }

};

List<Map<String, Object>> results = managementService.executeCustomSql(customSqlExecution);
```

上面list中的Map只包含id, name 和 create time，不是全部的任务对象。

使用上面的方式可以执行任何SQL。另一个更复杂的例子：

```
@Select({
    "SELECT task.ID_ as taskId, variable.LONG_ as variableValue FROM ACT_RU_VARIABLE variable",
    "inner join ACT_RU_TASK task on variable.TASK_ID_ = task.ID_",
    "where variable.NAME_ = #{variableName}"
})
List<Map<String, Object>> selectTaskWithSpecificVariable(String variableName);
```

使用这种方法，任务表会与变量表关联。只会获得对应名称的变量，任务id和对应的数字值会被返回。

18.5. #使用ProcessEngineConfigurator实现高级流程引擎配置

可以使用ProcessEngineConfigurator实现一种高级的扩展流程引擎的配置。想法是创建一个org.activiti.engine.cfg.ProcessEngineConfigurator接口的实现，注入到流程引擎配置里：

```
<bean id="processEngineConfiguration" class="...SomeProcessEngineConfigurationClass">
    ...
    <property name="configurators">
        <list>
            <bean class="com.mycompany.MyConfigurator">
                ...
            </bean>
        </list>
    </property>
    ...
</bean>
```

实现这个接口需要实现两个方法。configure方法，把ProcessEngineConfiguration作为参数。可以通过这种方法添加自定义配置，这个方法会被保证调用到，在流程创建之前，但在所有默认配置执行之前。另一个方法是getPriority方法，允许对configurator进行排序，如果一些configurator依赖其他的时候。

一个configurator的实例是LDAP集成，这个configurator用来替换默认的user和group管理器类，使用处理LDAP用户存储的类。所以基本上一个configurator允许很大程度上修改或增强流程引擎，对非常高级的场景是很有用的。另一个例子是使用自定义的版本替换流程定义缓存：

```
public class ProcessDefinitionCacheConfigurator extends AbstractProcessEngineConfigurator {
    public void configure(ProcessEngineConfigurationImpl processEngineConfiguration) {
        MyCache myCache = new MyCache();
        processEngineConfiguration.setProcessDefinitionCache(enterpriseProcessDefinitionCache);
    }
}
```

流程引擎配置器也可以通过ServiceLoader [http://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html] 自动从classpath中加载。就是说，放在jar中的configurator实现必须放在classpath下，并在jar的META-INF/services目录下包含一个org.activiti.engine.cfg.ProcessEngineConfigurator文件。文件的内容是自定义实现的全类名。当流程引擎启动时，日志会显示找到了哪些configurator：

```
INFO org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl - Found 1 auto-discoverable Process Engine Configuration
INFO org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl - Found 1 Process Engine Configurators in the classpath
INFO org.activiti.engine.impl.cfg.ProcessEngineConfigurationImpl - class org.activiti.MyCustomConfigurator
```

注意，这种ServiceLoader的方式在某些环境下可能无法正常运行。 可以使用ProcessEngineConfiguration的enableConfiguratorServiceLoader属性来禁用这个功能。（默认为true）。

18.6. #启用安全的BPMN 2.0 xml

大多数情况下，BPMN 2.0流程发布到Activiti引擎是在严格的控制下的，比如开发团队。然后，一些情况下，可能需要把比较随意的BPMN 2.0 xml上传到引擎。这种情况，要考虑恶意用户会攻击服务器，参考这里 [<http://www.jorambarrez.be/blog/2013/02/19/uploading-a-funny-xml-can-bring-down-your-server/>]。

为了避免上面链接描述的攻击，可以在引擎配置中设置：enableSafeBpmnXml：

```
<property name="enableSafeBpmnXml" value="true"/>
```

默认这个功能没有开启！这样做的原因是它需要使用StaxSource [<http://download.java.net/jdk7/archive/b123/docs/api/javax/xml/transform/stax/StAXSource.html>]类。不幸的是，一些平台（比如，JDK 6，JBoss，等等）不能用这个类（因为老的xml解析实现）所以不能启用安全BPMN 2.0 xml。

如果Activiti运行的平台支持这项功能，请打开这个功能。

18.7. #事件日志（实验）

在Activiti 5.16版本中，添加了一种（实验性）事件日志机制。这种日志机制构建在通用目的下的Activiti引擎的事件机制，默认是禁用的。它的意图是，由引擎产生的事件会被捕获，包含所有事件数据（和其他信息）的map会被创建出来，并提供给org.activiti.engine.impl.event.logger.EventFlusher，它会把数据刷新到其他地方。默认，会使用一个简单地基于数据库的事件处理器/刷新器，它会使用jackson把map转换为JSON，并把它保存到数据库中的EventLogEntryEntity实体。默认会创建这个数据库日志表（名叫ACT_EVT_LOG）。如果没有使用事件日志，可以删除这个表。

如果想启用数据库日志：

```
processEngineConfiguration.setEnableDatabaseEventLogging(true);
```

或者在运行阶段：

```
databaseEventLogger = new EventLogger(processEngineConfiguration.getClock());
runtimeService.addEventListener(databaseEventLogger);
```

EventLogger类可以继承。特别是，createEventFlusher()方法需要返回一个org.activiti.engine.impl.event.logger.EventFlusher接口的实例，在不想用默认的数据日志时。managementService.getEventLogEntries(startLogNr, size)；可以获取Activiti的EventLogEntryEntity实例。

很容易看到，现在可以使用大数据的NoSQL存储，比如MongoDb，Elastic Search等等来存储JSON。也可以看到这里使用的类（org.activiti.engine.impl.event.logger.EventLogger/EventFlusher和很多EventHandler类）是可插拔的，可以切换成你自己的应用场景（比如，不在数据库中存储JSON，而是把他们放到队列或大数据存储中。）

注意，事件日志机制是Activiti传统历史管理器的附加品。虽然所有数据都在数据库表中，但是它并没有为查询优化，所以不容易获取。真实的使用场景是审计跟踪，和把它们放到大数据存储中。

第#19#章#使用Activiti-Crystalball进行流程仿真（实验）

19.1. #介绍

19.1.1. #简介

activiti-crystalball (CrystalBall) 是Activiti业务流程管理平台的仿真引擎。CrystalBall可以用用户模拟一下场景：

- 决策支持 - 对于生产流程（比如，我们是否应该向系统添加更多资料以达到截止日期？）
- 优化和验证 - 测试修改并验证它们的影响。
- 培训 - 模拟器可以用来在使用前培训员工。
-

19.1.2. #CrystalBall是独立的

不需要：

- 创建单独的模拟模型和引擎。
- 为模拟创建不同的报告。
- 为模拟引擎准备很多数据。

CrystalBall模拟器是基于activiti的。所以很容易复制数据，启动模拟器，可以从历史中重播流程行为。

19.2. #CrystalBall内部

CrystalBall是一个离散事件模拟器 [http://en.wikipedia.org/wiki/Discrete_event_simulation]。最简单的实现是org.activiti.crystalball.simulator.SimpleSimulationRun。

```
init();

SimulationEvent event = removeSimulationEvent();

while (!simulationEnd(event)) {
    executeEvent(event);
    event = removeSimulationEvent();
}

close();
```

SimulationRun可以执行由不同源生成的模拟事件（参考PlayBack [#crb-playback]）。

19.3. #历史分析

模拟器可以使用的用例之一是分析历史。生产环境没有提供任何重复和调试bug的机会。这就是为什么基本不可能把流程引擎恢复到生产环境出现问题时完全一样的状态。问题的原因不是硬件，因为

- 时间 - 流程实例可能执行好几个月。
- 并发 - 流程实例会和其他实例一起运行，问题可能只产生于并发执行的情况。
- 用户 - 很多用户可以参与到流程实例中。它会把流程实例影响到出现问题的状态。

模拟器可以更好的暴露以上的问题。模拟过程是虚拟的，不会依赖真实环境。Activiti流程引擎本身是虚拟的。不需要创建虚拟流程引擎，为模拟环境使用。并发场景也是原生的。用户行为都会记录日志，并可以从日志重现，或者根据需要进行预测和生成。

分析历史的最好办法是重现一次。真实环境很难实现重现，但是模拟器就可以实现。

19.3.1. #历史的事件

重现历史子重要的事情是记录影响状态的事件。可以说我们的流程是由用户事件驱动的（比如，领取，完成任务。。。）这种情况下，我们可以使用两种事件源：

- 流程实例 - 当前只支持原始的 activiti-crystalball [http://gro-mar.github.io/activiti-crystalball/] 项目。
- ActivitiEvent日志。基本上，我们可以向引擎添加想要记录日志的 ActivitiEventListener。事件日志可以保存下来，用于后续的分析。基本的实现是 org.activiti.crystalball.simulator.delegate.event.impl.InMemoryRecordActivitiEventListener

```
@Override
public void onEvent(ActivitiEvent event) {
    Collection<SimulationEvent> simulationEvents = transform(event);
    store(simulationEvents);
}
```

事件会被保存，我们可以对历史进行重现。

19.3.2. #回放

回放的好处是可以一遍一遍播放，直到我们完全理解发生了什么。Crystalball模拟器基于真实数据，真实用户行为，这是crystalball的优势。

理解回放工作的最好方法是一步一步解释，
基于JUnit测试的一个例子
org.activiti.crystalball.simulator.delegate.event.PlaybackRunTest。测试的模拟流程是一个最简单的例子：

```
<process id="theSimplestProcess" name="Without task Process">
  <documentation>This is a process for testing purposes</documentation>

  <startEvent id="theStart"/>
  <sequenceFlow id="flow1" sourceRef="theStart" targetRef="theEnd"/>
  <endEvent id="theEnd"/>

</process>
```

流程已经发布了，可以用于真实和模拟运行。

- 记录事件 -

```
// get process engine with record listener to log events
ProcessEngine processEngine = (new RecordableProcessEngineFactory(THE_SIMPLEST_PROCESS, listener))
    .getObject();

// start process instance with variables
Map<String, Object> variables = new HashMap<String, Object>();
variables.put(TEST_VARIABLE, TEST_VALUE);
processEngine.getRuntimeService().startProcessInstanceByKey(SIMPLEST_PROCESS, BUSINESS_KEY, variables);

// check process engine status - there should be one process instance in the history
checkStatus(processEngine);

// close and destroy process engine
EventRecorderTestUtils.closeProcessEngine(processEngine, listener);
ProcessEngines.destroy();
```

上面的代码片段会在startProcessInstanceByKey方法调用后
ActivitiEventType.ENTITY_CREATED。

记录

- 开始运行模拟

```
final SimpleSimulationRun.Builder builder = new SimpleSimulationRun.Builder();
// init simulation run
// get process engine factory - the only difference from RecordableProcessEngineFactory that log listener
DefaultSimulationProcessEngineFactory simulationProcessEngineFactory = new DefaultSimulationProcessEngineFactory();
// configure simulation run
builder.processEngine(simulationProcessEngineFactory)
    // set playback event calendar from recorded events
    .eventCalendar(new PlaybackEventCalendarFactory(new SimulationEventComparator(), listener.getSimulationEventComparator()))
    // set handlers for simulation events
    .customEventHandlerMap(EventRecorderTestUtils.getHandlers());
SimpleSimulationRun simRun = builder.build();

simRun.execute(new NoExecutionVariableScope());

// check the status - the same method which was used in record events method
checkStatus(simulationProcessEngineFactory.getObject());

// close and destroy process engine
simRun.getProcessEngine().close();
ProcessEngines.destroy();
```

更高级的回放例子在

org.activiti.crystalball.simulator.delegate.event.PlaybackProcessStartTest。

19.3.3. #调试流程引擎

回放限制我们执行所有模拟事件（比如，开始流程，完成任务）一次性。 调试器允许我们把自行拆分成更小的步骤，在步骤之间观察流程引擎的状态。

SimpleSimulationRun实现了SimulationDebugger接口。SimulationDebugger可以一步一步执行模拟事件， 可以模型特定时间的执行。

```

/**
 * Allows to run simulation in debug mode
 */
public interface SimulationDebugger {
    /**
     * initialize simulation run
     * @param execution - variable scope to transfer variables from and to simulation run
     */
    void init(VariableScope execution);

    /**
     * step one simulation event forward
     */
    void step();

    /**
     * continue in the simulation run
     */
    void runContinue();

    /**
     * execute simulation run till simulationTime
     */
    void runTo(long simulationTime);

    /**
     * execute simulation run till simulation event of the specific type
     */
    void runTo(String simulationEventType);

    /**
     * close simulation run
     */
    void close();
}

```

可以执行SimpleSimulationRunTest来实际观察流程引擎调试器的运行。

19.3.4. #重播

回放需要创建另一个流程引擎实例。回放不会影响真实环境，换句话说，它需要模拟环境配置。重播工作在真实的流程引擎之上。重播在运行的流程引擎中执行模拟事件。结论是重播是实时运行的。实时意味着会被立即执行。

下面的例子演示了如何重播一个流程实例。相同的技术可以用于回放一个流程实例。
 (ReplyRunTest) 测试的第一部分初始化流程引擎，启动一个流程实例，完成流程实例的任务。

```

ProcessEngine processEngine = initProcessEngine();

TaskService taskService = processEngine.getTaskService();
RuntimeService runtimeService = processEngine.getRuntimeService();

Map<String, Object> variables = new HashMap<String, Object>();
variables.put(TEST_VARIABLE, TEST_VALUE);
ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(USER_TASK_PROCESS, BUSINESS_KEY,
variables);

Task task = taskService.createTaskQuery().taskDefinitionKey("userTask").singleResult();

```

```
TimeUnit.MILLISECONDS.sleep(50);
taskService.complete(task.getId());
```

使用的流程引擎是基础的InMemoryStandaloneProcessEngine，配置了

- InMemoryRecordActivitiEventListener （已在回放中使用过）记录Activiti事件，并转换为模拟事件。
- UserTaskExecutionListener – 当创建新用户任务时，新任务会重播流程实例，把任务完成事件放到事件日历中。

测试的下一个部分，在原始流程相同的引擎上启动模拟调试器。重播事件处理器使用StartReplayProcessEventHandler替换StartProcessEventHandler。

StartReplayProcessEventHandler获取流程实例id来重播，在流程实例启动的初始位置处理。StartProcessEventHandler在开始阶段，会创建一个新流程实例，包含一个变量。变量名为_replay.processInstanceId。变量用来保存重播的流程实例id。与SimpleSimulationRun不同，ReplaySimulationRun不会：

- 创建和关闭流程引擎 实例。
- 修改模拟时间。（无法修改实际时间）

```
final SimulationDebugger simRun = new ReplaySimulationRun(processEngine,
getReplayHandlers(processInstance.getId()));
```

现在可以开始重播流程实例了。刚开始，这里没有运行的流程实例。只有一个已完成的，在历史中的流程实例。在初始化后，会在事件日历中添加一个模拟事件 – 用来启动流程实例，重播已经完成的流程实例。

```
simRun.init();

// original process is finished - there should not be any running process instance/task
assertEquals(0, runtimeService.createProcessInstanceQuery().processDefinitionKey(USER_TASK_PROCESS).count());
assertEquals(0, taskService.createTaskQuery().taskDefinitionKey("userTask").count());

simRun.step();

// replay process was started
assertEquals(1, runtimeService.createProcessInstanceQuery().processDefinitionKey(USER_TASK_PROCESS).count());
// there should be one task
assertEquals(1, taskService.createTaskQuery().taskDefinitionKey("userTask").count());
```

任务创建时，UserTaskExecutionListener会创建一个新模拟事件来结束用户任务。

```
simRun.step();

// userTask was completed - replay process was finished
assertEquals(0, runtimeService.createProcessInstanceQuery().processDefinitionKey(USER_TASK_PROCESS).count());
assertEquals(0, taskService.createTaskQuery().taskDefinitionKey("userTask").count());
```

模拟结束了，我们可以继续启动另一个流程实例，或者其他事件。现在我们可以关闭simRun和流程引擎。

```
simRun.close();
processEngine.close();
```

```
ProcessEngines.destroy();
```