# 普元Studio开发入门

[目标 4](#_Toc290038505)

[预期读者 4](#_Toc290038506)

[文档导读 4](#_Toc290038507)

[Eclipse入门 6](#_Toc290038508)

[Eclipse是什么？ 6](#_Toc290038509)

[Eclipse平台体系结构 6](#_Toc290038510)

[插件开发入门 9](#_Toc290038511)

[认识插件 9](#_Toc290038512)

[扩展点机制 10](#_Toc290038513)

[开发一个简单的插件 11](#_Toc290038514)

[搭建插件开发环境 11](#_Toc290038515)

[新建一个插件项目 12](#_Toc290038516)

[插件清单编辑器 13](#_Toc290038517)

[添加一个扩展 14](#_Toc290038518)

[运行插件项目 19](#_Toc290038519)

[Studio概览 22](#_Toc290038520)

[总体架构 22](#_Toc290038521)

[核心模块 25](#_Toc290038522)

[资源访问API 25](#_Toc290038523)

[模型管理 26](#_Toc290038524)

[元模型 30](#_Toc290038525)

[UI框架 31](#_Toc290038526)

[资源管理器（Navigator） 33](#_Toc290038527)

[图形化编辑器 37](#_Toc290038528)

[索引 39](#_Toc290038529)

[Studio总体开发过程 41](#_Toc290038530)

[搭建Studio开发环境 41](#_Toc290038531)

[导入源代码 41](#_Toc290038532)

[运行Studio 41](#_Toc290038533)

[实现新功能 42](#_Toc290038534)

[设计 42](#_Toc290038535)

[开发 42](#_Toc290038536)

[调试 42](#_Toc290038537)

[产品集成 42](#_Toc290038538)

[添加到功能部件中 42](#_Toc290038539)

[添加到编译环境中 43](#_Toc290038540)

[基本规范 44](#_Toc290038541)

[插件命名规范 44](#_Toc290038542)

[插件依赖规范 44](#_Toc290038543)

[编码规范 44](#_Toc290038544)

[入门准备 45](#_Toc290038545)

[SWT 45](#_Toc290038546)

[JFace 45](#_Toc290038547)

[Workspace 45](#_Toc290038548)

[EMF 45](#_Toc290038549)

[GEF 45](#_Toc290038550)

[相关书籍 46](#_Toc290038551)

[引入一种新的资源 47](#_Toc290038552)

[设计资源模型 47](#_Toc290038553)

[创建EMF模型 48](#_Toc290038554)

[生成模型代码 53](#_Toc290038555)

[对模型进行测试 55](#_Toc290038556)

[让Studio认识资源 58](#_Toc290038557)

[注册 58](#_Toc290038558)

[解析 60](#_Toc290038559)

[编译 62](#_Toc290038560)

[验证 63](#_Toc290038561)

[索引 64](#_Toc290038562)

[运行Studio 65](#_Toc290038563)

[新建资源文件 67](#_Toc290038564)

[新建向导框架 67](#_Toc290038565)

[实现BPMN新建向导 68](#_Toc290038566)

[注册向导 69](#_Toc290038567)

[注册到资源管理器 72](#_Toc290038568)

[在资源管理器中展现资源 74](#_Toc290038569)

[开始之前 74](#_Toc290038570)

[为节点添加图片 75](#_Toc290038571)

[修改节点文字 76](#_Toc290038572)

[显示子节点 77](#_Toc290038573)

[添加过滤器 78](#_Toc290038574)

[对资源进行编辑 81](#_Toc290038575)

[开始之前 81](#_Toc290038576)

[MVC与GEF 81](#_Toc290038577)

[Draw2D 84](#_Toc290038578)

[图形化BPMN编辑器 85](#_Toc290038579)

[创建Graphical Viewer 86](#_Toc290038580)

[创建RootEditPart 88](#_Toc290038581)

[实现Content EditPart 89](#_Toc290038582)

[实现各个子EditPart 90](#_Toc290038583)

[添加节点间的连线 93](#_Toc290038584)

[注册 94](#_Toc290038585)

[事件处理 97](#_Toc290038586)

[附录 100](#_Toc290038587)

## 目标

本文档是Studio开发的入门性介绍，目标是通过本文档的学习，使得具备一定Eclipse插件开发经验的开发人员能够快速从事Studio相关的开发工作，熟悉Studio的基本架构。

## 预期读者

* 准备从事Studio开发的开发人员
* 想要了解Studio基本开发过程的人

## 文档导读

[Eclipse入门](#_Eclipse入门)章节介绍了Eclipse的基本概念以及Eclipse平台的体系结构，使读者对Eclipse平台有个基本的认识。如果您是Eclipse插件开发的新手，推荐阅读本章节。

[插件开发入门](#_插件开发入门)介绍了插件开发的一些基本概念，并通过一个简单的示例演示了插件开发的基本过程。如果您已经做过插件开发，则可以跳过本章。

[Studio概览](#_Studio概览)介绍了EOS Studio的总体架构，并对Studio中的核心框架和模块进行了简单的介绍，通过阅读本章您可以了解到EOS Studio开发中所涉及到的相关技术。

[Studio总体开发过程](#_Studio总体开发过程)描述了初次接触Studio开发的人员如何搭建Studio开发环境，以及进行Studio开发的一般性过程。

[基本规范](#_基本规范)章节介绍了从事Studio插件开发的一些基本规范。

[入门准备](#_入门准备)列出了从事Eclipse插件开发和Studio开发需要掌握的一些Eclipse基础知识和模块，包括这些模块的一些参考资料。

[引入一种新的资源](#_引入一种新的资源)介绍了如何向Studio中添加一种新的资源，并以BPMN资源为例进行了一般性过程的介绍。

[让Studio认识资源](#_让Studio认识资源)介绍了Studio中是如何管理资源的，包括资源的注册、解析、编译、验证、索引等。主要围绕Studio的模型管理框架进行了介绍。

[新建资源文件](#_新建资源文件)介绍了如何利用Studio的UI框架来快速创建资源新建向导。

[在资源管理器中展现资源](#_在资源管理器中展现资源)介绍了如何在Studio的资源管理器中进行资源的展现，通过本章的介绍使读者对Eclipse的CNF（Common Navigator Framework）框架以及Studio的资源管理器框架有初步的认识。

[对资源进行编辑](#_对资源进行编辑)介绍了如何利用Studio的GEF框架来快速搭建一个图形化编辑器，我们以BPMN图形化编辑器为例，穿插介绍了GEF框架，希望通过阅读本章使您对GEF有基本的认识，为GEF的深入开发做准备。

## Eclipse入门

### Eclipse是什么？

对Java开发人员而言，Eclipse是强大的Java集成开发环境；对C/C++开发人员而言，Eclipse是C/C++集成开发环境；对Python开发人员而言，Eclipse是友好的Python集成开发环境。

Eclipse不止这些，从更广泛意义上来讲，**Eclipse是一个开放源代码的、基于Java的可扩展开发平台**。Eclipse为我们实现了一套灵活的架构和IDE基础设施，它可以作为各种IDE的基础平台，基于该基础平台，我们可以快速构建自己的集成开发环境。

通过在Eclipse基础平台上插入JDT（Java开发工具集）插件，Eclipse就拥有了强大的Java集成开发环境；同样，在Eclipse平台上插入CDT（C/C++开发工具集）插件，您就可以在Eclipse中进行C/C++开发。

Eclipse的这种灵活性源自于它的插件机制，在Eclipse中所有的功能都以插件（Plug-in）的形式提供；插件通过扩展点，允许外部插件对其进行功能扩展，如：工作台（Workbench）插件对外提供了org.eclipse.ui.actionSets扩展点，通过该扩展点，外部插件可以向工作台中添加新的菜单。

### Eclipse平台体系结构

Eclipse平台的底层实现了OSGi规范，每个插件都对应OSGi中的一个Bundle,这使得Eclipse插件具有动态的、可插拔的特性。Eclipse平台体系结构如下图所示：



由上图可知，Eclipse平台主要由平台运行时（Platform Runtime）、工作空间（Workspace）、工作台（Workbench）以及团队支持和帮助系统等组件组成。各组件的主要功能和作用说明如下：

* **平台运行时（Platform Runtime）**

平台运行时定义了扩展点和插件模型。平台运行时负责动态地发现插件并维护平台注册表中有关这些插件及其扩展点的信息。插件是根据平台的用户操作在需要时加载和启动的。运行时是使用OSGi框架实现。

* **工作空间（Workspace）**

定义API，以创建和管理由工具生成的并保存在文件系统中的资源（项目、文件和文件夹）。

* **工作台（Workbench）**

工作台为Eclipse提供用户界面。它是使用标准窗体工具包（SWT）和一个更高级的API（JFace）来构建的。

SWT是Eclipse提供的一个用户界面库，它创建了Java版的本地主机操作系统GUI控件，它依赖于本机实现，因此，SWT具有本机窗口系统的界面外观，它更加快速。

JFace在SWT的基础了又封装了一层，提供了更加高级、粒度更大的界面组件，如：对话框、向导等。JFace的目的不是替换SWT，而是更加容易的在SWT基础至少构建用户界面。

工作台为我们提供了丰富的扩展点，使得我们可以很容易的为工作台添加自己的菜单、工具条、编辑器、视图、透视图等。

* **团队支持（Team Work）**

团对支持组件负责提供版本控制和配置管理支持。我们可以很容易扩展团队支持组件以支持新的版本控制系统。

* **调试支持（Debug）**

对语言的调试支持是任何IDE都应该具有的特性，它在很大程度上提高了开发人员的效率。Eclipse的调试支持组件为我们抽象了一套语言无关的调试模型。基于该模型，我们可以很容易的扩展它，为新语言提供调试支持。

* **帮助（Help）**

Eclipse提供了一套灵活的帮助系统实现，对外提供了许多扩展点，IDE的实现者可以很容易扩展它，以便在自己的系统中加入帮助功能。

EOS Studio就是在Eclipse基础平台之上，通过扩展开发，向用户提供了一组插件，使得Eclipse具有了EOS应用开发的能力。

## 插件开发入门

本章节将介绍插件开发的基本概念和基本过程，并通过开发一个简单的插件让大家对插件开发的整体过程有直观和清晰的认识。具备Eclipse插件开发经验的读者可以跳过本章。

### 认识插件

插件是Eclipse定义新功能的核心机制，是Eclipse可以识别的最小功能单元；类似于一个对象，插件封装了功能的行为和数据。一个插件通常由实现了插件功能的Java类和插件的描述文件MANIFEST.MF构成。一个插件的结构如下所示：

|  |
| --- |
| org.eclipse.core.runtime （插件的名称）  |-- META-INF  |-- MANIFEST.MF //满足OSGi规范的插件描述文件  |-- XXX //插件资源目录：可以包含类和相关资源  |-- plugin.xml //扩展与扩展点描述文件 |

MANIFEST.MF文件描述了插件的元数据信息，包括：标识符、版本、依赖的其它插件、导出的包等信息。EOS Studio中的逻辑流编辑器是通过com.primeton.studio.biz.ui插件提供的，下面列出该插件的MANIFEST.MF文件的内容：

|  |
| --- |
| Manifest-Version: 1.0  **Bundle-ManifestVersion**: 2  **Bundle-Name**: %PluginName  **Bundle-SymbolicName**: com.primeton.studio.biz.ui;*singleton*:=true  **Bundle-Version**: 6.0.0  **Bundle-Vendor**: %PluginProvider  **Bundle-Localization**: plugin  **Require-Bundle**: com.primeton.studio.gef.ui,  com.primeton.studio.biz.core,  com.primeton.studio.workbench.wizard,  com.primeton.studio.navigator,  com.primeton.studio.debug.ui,  com.primeton.emf.core,  com.primeton.studio.ui,  com.primeton.studio.core,  org.eclipse.core.runtime,  org.eclipse.gef,  org.eclipse.emf.ecore,  com.primeton.studio.wsdl.core,  com.primeton.studio.license,  org.eclipse.xsd  Eclipse-LazyStart: true  **Export-Package**: com.primeton.studio.biz.ui,  com.primeton.studio.biz.ui.actions,  com.primeton.studio.biz.ui.conncheck,  com.primeton.studio.biz.ui.editor,  com.primeton.studio.biz.ui.factory,  com.primeton.studio.biz.ui.refactor,  com.primeton.studio.biz.ui.wizard  **Bundle-Activator**: com.primeton.studio.biz.ui.BizUiPlugin |

对MANIFEST.MF文件中的一些重要属性说明如下：

|  |  |
| --- | --- |
| **属性** | **备注** |
| Manifest-Version | MANIFEST.MF文件规格版本号。 |
| **Bundle-SymbolicName** | 插件的唯一标识符。 |
| **Bundle-Version** | 该属性指定插件的版本号。 |
| **Require-Bundle** | 该属性指定插件所依赖的其它外部插件。 |
| **Export-Package** | 该属性指定插件对外暴露的包。 |
| Eclipse-LazyStart | 这是Eclipse在OSGi之上添加的扩展属性，说明该插件是否是懒加载的。 |
| **Bundle-Activator** | 插件的激活器，该类用于启动和停止插件。 |

在安装好的Eclipse中，插件一般位于Eclipse安装目录下的plugins目录中。

### 扩展点机制

前面我们讲过Eclipse的底层是基于OSGi机制实现的，OSGi为我们提供了一种以模块化的方式来开发、部署、运行、管理组件（Bundle）的机制，其主要的特征包括模块化、生命周期管理、松耦合交互等。我们这里的插件就对应到OSGi中的组件（Bundle）。但是，OSGi并没有追求如何让一个组件更容易扩展、更加开放。而Eclipse本身作为一个基础平台，其关键需求就是方便用户扩展，并能够很方便的和Eclipse平台本身做无缝集成。基于此，Eclipse在OSGi之上提供了扩展点的机制。

Eclipse的扩展点机制主要包含三个部分：扩展点（Extension Point）、扩展（Extension）和扩展注册表（Extension Registry）。

扩展点定义了扩展的契约，它通常由XML标记和Java接口组成。程序依据需要在合适的地方引入扩展点，其它插件可以根据扩展点的定义向扩展注册表中注册扩展。Eclipse中通过plugin.xml文件来描述扩展点和扩展信息。

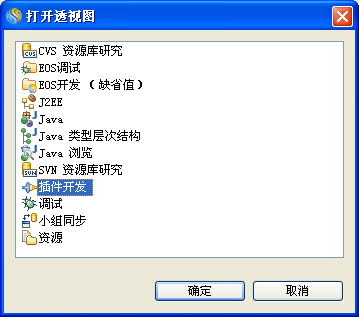
### 开发一个简单的插件

本节我们将动手开发一个简单的Hello World插件，该插件将向Eclipse的工作台（Workbench）中添加一个新的菜单“Hello”，这个菜单将包含“Say Hello”子菜单，点击这个子菜单，将显示一个“Hello World!”对话框。

#### 搭建插件开发环境

在Eclipse中进行插件开发，需要插件开发环境（PDE），您可以从Eclipse网站（http://www.eclipse.org/downloads/）上下载带有插件开发环境的Eclipse版本，如：Eclipse Classic版。EOS Studio也包含了PDE开发环境，我们将以EOS Studio作为我们的插件开发环境。

1. 获取并安装EOS 6.X 版本的Studio；
2. 启动Studio后，选择**窗口🡪打开透视图🡪其他…**菜单项；
3. 在弹出的“打开透视图”对话框中，选择“**插件开发**”，如下所示：



1. 点击确定，系统将切换到插件开发透视图。

#### 新建一个插件项目

一个插件对应一个插件项目，插件开发环境（PDE）为我们提供了新建插件项目向导。新建一个插件项目的步骤如下：

1. 选择**文件🡪新建🡪项目…**菜单项；
2. 系统弹出“新建项目”向导，选择“**插件项目**”，点击“下一步”；
3. 在“插件项目”向导页中，指定项目的名称：com.primeton.studio.hello，点击“下一步”；
4. 在“插件内容”向导页中，可以指定插件的一些基本属性，如：插件标识、插件版本等，如下所示：



1. 点击“完成”，向导将在您的工作空间中创建一个插件项目。

#### 插件清单编辑器

在工作台左边的“包资源管理器”视图中，可以看到我们刚刚新建的插件项目，选择META-INF目录下的MANIFEST.MF文件，右键选择“打开方式🡪插件清单编辑器”，系统将打开插件清单编辑器，如下所示：



插件清单编辑器主要用来编辑MANIFEST.MF和plugin.xml文件，主要的编辑页说明如下。

“概述”页中显示了插件的基本信息，如：插件标识、版本、激活器等。

“依赖性”页中显示了插件所依赖的其它插件以及导入的包，通过该页可以为插件添加新的插件依赖。

“运行时”页中显示了插件可以供外部访问的包，以及插件的类路径。通常，我们可以在“类路径”中添加插件所引用的第三方类库。

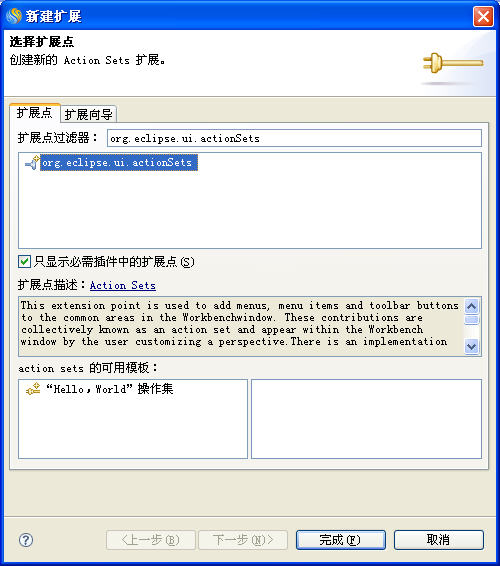
“扩展”页中显示了该插件所提供的扩展，在该页中，我们可以为插件定义新的扩展。

“扩展点”页中显示了该插件定义的扩展点信息。

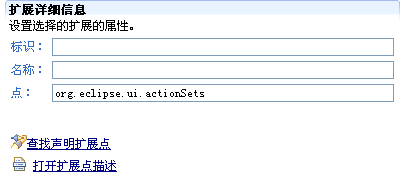
#### 添加一个扩展

Eclipse的工作台（Workbench）插件，为我们提供了org.eclipse.ui.actionSets扩展点，通过该扩展点可以很容易的在工作台中加入自己的菜单。添加一个菜单的步骤如下：

1. **添加org.eclipse.ui.actionSets扩展**
2. 打开插件清单编辑器，选择“扩展”页；
3. 在“所有扩展”列表中，选择“**添加…**”按钮，系统弹出新建扩展向导；
4. 在扩展点过滤器中，输入：org.eclipse.ui.actionSets，如下所示：

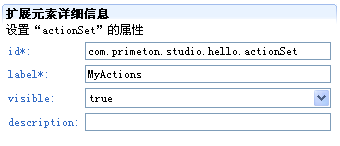


1. 点击“完成”，向导将在“所有扩展”列表中添加一个新的扩展；
2. 选择新建的org.eclipse.ui.actionSets扩展，在右边将显示扩展的详细信息，如下所示：

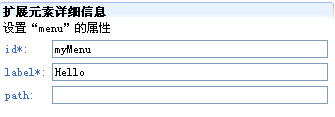


点击“打开扩展点描述”链接，将打开org.eclipse.ui.actionSets扩展点的详细描述信息，这个信息将帮助我们很好的理解该扩展点的作用，以及如何使用这个扩展点。

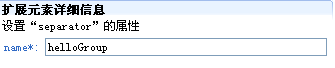
1. **新建actionSet**
2. 在新建的org.eclipse.ui.actionSets扩展上点击右键，选择“**新建🡪actionSet**”菜单项；
3. 编辑actionSet的属性，如下所示：



1. **新建menu**
2. 在新建的actionSet上，右键选择“**新建🡪menu**”菜单项；
3. 编辑menu的属性，如下所示：

****

1. **新建分组（separator）**
2. 在新建的menu上，右键选择“新建🡪separator”菜单项；
3. 编辑separator的属性，如下所示：



1. **新建action**
2. 在新建的“MyActions”actionSet上，右键选择“新建🡪action”菜单项；
3. 设置action的属性，如下所示：

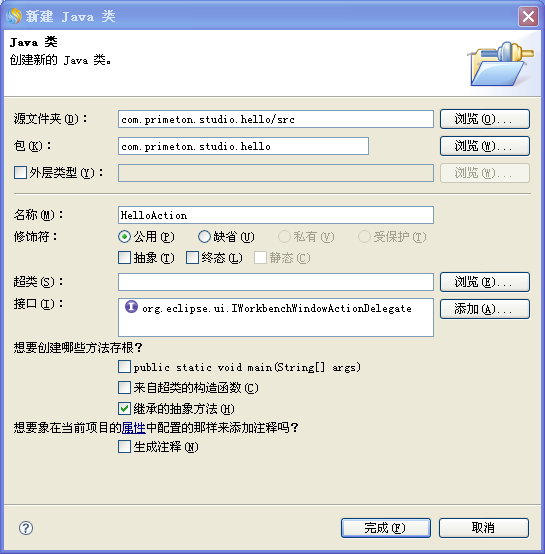


其中label属性指定了action的名称；menubarPath属性指定了该action将出现在菜单条的哪个位置上，这里的值为myMenu/helloGroup，说明了这个action将显示在我们刚才定义的myMenu菜单的helloGroup分组中。

1. 我们需要为action指定一个实现类，根据org.eclipse.ui.actionSets有关action的描述信息，这个实现类需要实现org.eclipse.ui.IWorkbenchWindowActionDelegate接口。点击hello action中的class属性，如下所示：



系统将弹出Java类新建向导，指定包名和类名信息，如下所示：



点击“完成”，系统将创建一个HelloAction类，该类实现了org.eclipse.ui.IWorkbenchWindowActionDelegate接口；

1. HelloAction类创建完毕后，系统将自动回填hello action的class属性值，如下所示：



1. 激活插件的插件清单编辑器，点击“保存”来保存上述配置的扩展信息。
2. **实现action功能**

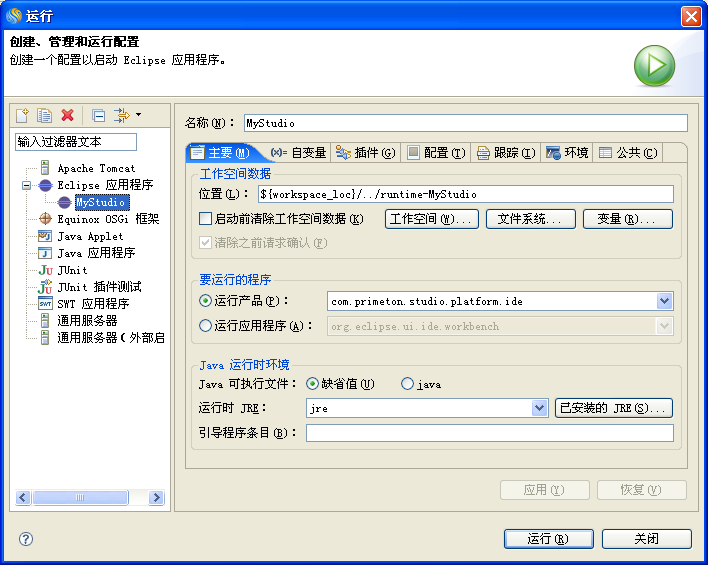
HelloAction类执行时，将运行其中的run()方法，我们编辑HelloAction#run方法，让它弹出一个对话框来显示“Hello World!”信息，HelloAction类如下所示：

|  |
| --- |
| **package** com.primeton.studio.hello;  **import** org.eclipse.jface.action.IAction;  **import** org.eclipse.jface.dialogs.MessageDialog;  **import** org.eclipse.jface.viewers.ISelection;  **import** org.eclipse.ui.IWorkbenchWindow;  **import** org.eclipse.ui.IWorkbenchWindowActionDelegate;  **public** **class** HelloAction **implements** IWorkbenchWindowActionDelegate {    **private** IWorkbenchWindow window;  **public** **void** dispose() {  }  **public** **void** init(IWorkbenchWindow window) {  **this**.window = window;  }  **public** **void** run(IAction action) {  MessageDialog.*openInformation*(  window.getShell(),  "Hello",  "Hello World!\n欢迎进入插件开发的世界!");  }  **public** **void** selectionChanged(IAction action, ISelection selection) {  }  } |

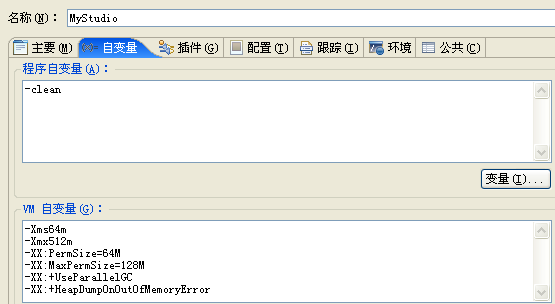
#### 运行插件项目

经过以上步骤后，我们的第一个插件就开发完成了，现在我们来运行这个插件项目，看看它的效果。运行插件的步骤如下：

1. 选择“运行🡪运行…”菜单项，弹出运行配置向导；
2. 在左边的“运行类型”树上，选择“Eclipse 应用程序”，右键选择“新建”菜单，将新建一个Eclipse应用程序运行项；
3. 编辑相关信息，如：名称、工作空间位置、要运行的产品等信息，如下所示：



1. 编辑“自变量”信息，添加“程序自变量”和“VM自变量”，如下所示：

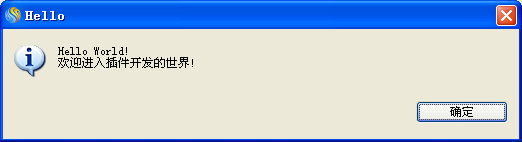


之所以要编辑这些信息，是因为我们要运行的Eclipse应用程序包含了许多插件，为了使我们的Eclipse应用程序平稳运行，我们调整了Java虚拟机的内存。

1. “插件”配置页中，显示了即将要运行的Eclipse应用程序所包含的插件，可以看出，Eclipse自动将我们工作空间中的插件加入到了插件列表中，如下所示：



1. 点击“应用”按钮，保存运行配置信息；
2. 点击“运行”按钮，将启动一个新的Eclipse应用程序，它包含了我们所创建的hello插件。启动完毕后，可以看到新启动的EOS Studio的菜单条中包含了我们所添加的“hello”菜单项，点击“hello”菜单项，将显示如下对话框：



至此，我们的第一个插件已经开发完成了，可以看出开发一个插件还是比较简单的。同时也可以看出由于Eclipse本身的扩展性，使我们可以很容易的扩展它，为它添加新的功能。

## Studio概览

Studio是面向SOA应用的集成开发环境，它能够支持SOA应用的设计、开发、部署、调试、重构。Studio在Eclipse基础平台之上做了大量扩展，一共包含了90多个插件，通过这些插件有效支撑了EOS项目的开发。

在长期的Studio开发中，普元积累和沉淀了一套自有的Studio开发框架，如：可视化编辑器框架、UI框架、资源管理框架等，利用这些框架可以加快Studio的功能开发，提高开发效率。

同时，经过多个定制化项目的积累，Studio目前也具备了一定的开放性。通过Studio提供的众多扩展点，第三方用户可以很容易的为Studio扩展新的功能，实现了Studio的快速定制。

### 总体架构

Studio的总体架构图如下所示：



从上图可以看出，Studio的主要核心模块包括：基础框架、编辑器模块、通用功能模块等。

**Eclipse平台**

Eclipse平台主要包括Eclipse基础平台，以及在基础平台之上的由Eclipse组织提供的JDT、WTP、EMF、GEF等重要模块组成。

Eclipse基础平台主要提供了基于OSGi规范实现的插件机制，以及工作台UI、工作空间资源管理等核心功能。

JDT是Java开发工具集，EOS项目本质上是一个Java项目，因此我们利用JDT来管理和维护我们的Java资源，包括：Java源文件编辑、编译、重构等。

WTP（Web Tools Platform），Web应用工具平台，是用来开发Web应用的插件集。Studio中的JSP/HTML编辑器、数据库资源管理器以及数据集编辑器所用到的数据库操作相关的功能，就是基于WTP插件集进行扩展开发而形成的。

EMF（Eclipse Modeling Framework）插件集是Eclipse MDA的基础，其核心是基于MOF规范实现的Ecore模型。EMF提供了一套强大的代码生成机制，依据模型描述文件（如：XML Schema、带Annotation的Java接口等），EMF可以为模型生成健壮的、设计良好的Java代码，同时，它所生成的Java代码具备天然的事件通知机制，非常适合作为工具（如：各种资源编辑器）的底层模型。Studio中的各种编辑器的底层模型就是通过EMF生成的，这大大简化了模型的开发工作。

GEF（Graphical Editing Framework），图形编辑框架。该框架封装了一套图形编辑相关的API，通过它可以很容易的实现可视化图形编辑器。GEF插件集主要包含了两个部分：Draw2D和GEF；其中Draw2D封装了一套二维图形绘制框架，而GEF则在Draw2D的基础上提供了一套图形编辑（交互）框架，在用户操作与图形之间建立起了沟通的桥梁。Studio中的各种可视化编辑器，如：页面流编辑器、逻辑流编辑器、构件装配等，都是基于GEF实现的。

LTK（Languages ToolKit），是一套独立于特定语言的语言基础框架，Eclipse中各种开发语言（如：Java、C/C++）所提供的重构功能都是基于该框架进行开发的。Studio中所提供的重构功能也是基础该框架进行开发的。

**基础框架**

Studio的基础框架主要包含四个部分：资源访问API、模型管理框架、UI框架和元模型。

资源访问API针对不同的资源存储介质和操作API进行了抽象，提供了一套统一的资源访问API。

模型管理框架提供了一套机制，对EOS中的模型，如：页面流模型、逻辑流模型等，以一种统一的方式进行管理：注册、编译检查、编译、索引等。

UI框架对常见的界面样式和元素进行了抽象，在SWT/JFace之上进行了封装，使得UI的编程工作得以大大简化，提高了UI开发效率。

元模型是模型的模型，即EOS中的各种资源模型，如：页面流模型、逻辑流模型都可以通过Studio的元模型进行描述和展现。在Studio中引入元模型有效的降低了模块间的耦合，同时，Studio的元模型还是一个轻量级的模型，这保证了资源管理器得以快速展现。

**通过功能**

通用功能主要包括资源管理器（Navigator）、组件库视图、索引、重构等模块组成。

资源管理器主要负责提供一个视图，以树形结构层次分明的展现EOS项目的资源组成，EOS的资源管理器是基于Eclipse的CNF（Common Navigator Framework）进行扩展开发的，通过提供新的Navigator Content来告诉CNF如何展现EOS项目。

组件库视图展现了EOS项目可引用的资源，它扩展自资源管理器。

索引是任何IDE都必须具备的功能，可以说是IDE的灵魂！Studio基于Lucene实现了对EOS项目资源的索引。

重构是现代IDE具备的一个重要特性。利用重构，用户可以方便的对既有代码作结构化调整。Studio基于LTK和资源索引提供了对EOS资源的重构功能，包括：重命名、移动等。

**编辑器模块**

编辑器模块主要包含了各种编辑器的实现，特别是图形化编辑器的实现，主要包括编辑器模型和GEF框架两个子模块。

其中，编辑器模型基于EMF抽象出了一套图形化编辑器的底层模型，其核心元素是节点（Node）和连线（Connection），Studio的大部分图形化编辑器，如：页面流编辑器、逻辑流编辑器、数据集编辑器等，它们的底层模型都是同一套模型。

Studio所封装的GEF框架是图形化编辑器的抽象实现，它是基于上述的编辑器模型实现的，通过扩展抽象实现，我们可以很容易的开发出一个图形化编辑器。

### 核心模块

Studio的核心模块是Studio的精华所在，从事Studio开发的人员必须对这些模块了然于胸。Studio的核心模型主要包括：资源管理、模型管理、UI框架、资源管理器、图形化编辑器和索引等。下面我们对这些模块做一些简单的介绍。

#### 资源访问API

资源访问框架在Java IO、Eclipse资源模型和压缩包资源模型的基础上抽象出了一套统一的资源访问API。利用该框架，可以使我们以统一的方式访问各种形式的资源，如：文件系统中的资源、压缩包中的资源。

利用Studio开发的EOS项目，存在两种编译环境，一种是直接在Eclipse中进行项目编译；另外一种是通过Ant脚本对EOS项目进行编译。我们知道，在Eclipse中通常需要使用Eclipse提供的资源管理API（IResource），在这种情况下，如果我们的编译相关的代码直接使用IResource进行操作的话，由于在Ant环境下无法直接使用Eclipse的资源管理API，那么我们必须再为Ant编译环境重写一套基于Java IO的编译代码。这造成了功能的重复开发，同时给维护工作带来困难。如何解决这个问题？资源管理框架给我们带来了福音，我们的编译代码只需要依赖IResourceDelegate抽象，框架自动识别环境，提供对不同环境的资源访问。

资源访问API的核心接口类以及它们之间的关系如下图所示：



资源访问API内置了三种类型的资源实现：

* Java IO资源访问：JavaResourceDelegate
* Eclipse资源访问：EclipseResourceDelegate
* 压缩包资源访问：ArchiveResourceDelegate

同时，用户可以根据自己的需要实现针对特定类型的资源访问。如：我们可以将EOS项目保存到数据库中，那么只需要利用JDBC协议实现一套IResourceDelegate即可。

关于资源访问API的更详细信息，请参见《资源访问API》文档。资源访问API的相关代码位于com.primeton.studio.runtime插件中，有兴趣的读者请查阅相关代码。

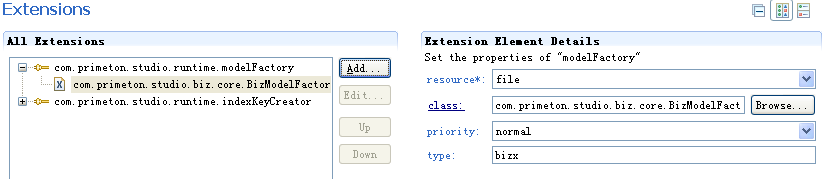
#### 模型管理

模型管理框架提供了一套机制，对EOS项目中的模型，如：页面流模型、逻辑流模型等，以一种统一的方式进行管理：注册、编译检查、编译、索引等。JDT只维护了Java模型和资源，而EOS项目包含了多种模型，如果为每个模型都提供一套管理机制的化，实现起来复杂，不利于统一架构和维护。所以，在Studio中提供了对多种模型进行统一管理的机制，其核心类是IModelFactory，故名思意，它是模型工厂，从这个工厂中，我们可以很方便的得到模型相关的处理器，如：模型编译器（IModelCompiler）、模型验证器（IModelValidator）、模型解析器（IModelParser）等。IModelFactory类以及与它相关的类的结构如下图所示：

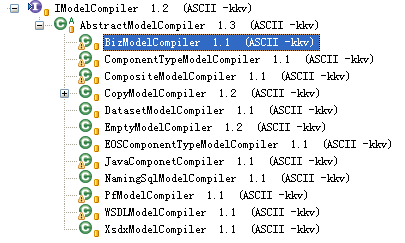


在Studio中，通常一个资源都会有一个对应的IModelFactory。通过IModelManager#getModelFactory方法，我们可以资源的扩展名得到该资源对应的IModelFactory，进而得到资源所对应的模型的解析器、编译器和验证器等。

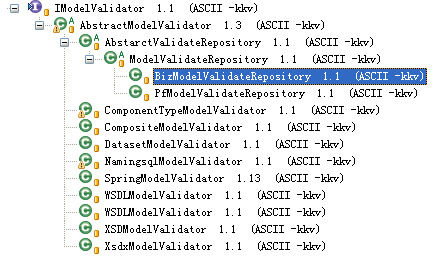
在Studio中引入一种新的模型时，都需要提供该模型的IModelFactory实现，并通过com.primeton.studio.runtime.modelFactory扩展点注册到系统中。RuntimePlugin在启动时，会调用loadModelFactories()方法来加载所有的模型工厂类。如：BizModelFactory是针对逻辑流模型的工厂类，它与.bizx扩展名相对应，在com.primeton.studio.biz.core中注册了该模型工厂，如下所示：



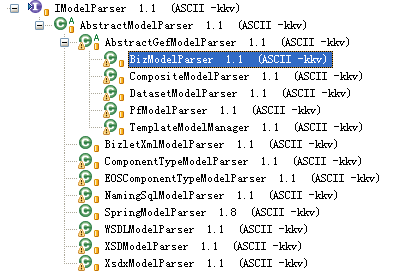
模型编译器负责对模型进行编译，将源文件编译成目标文件。IModelCompiler是模型编译器的接口类，该类的compile方法负责模型的编译工作。EOSProjectBuilder是针对EOS项目的增量编译器，当资源发生变化时，该编译器会被启动用来对资源做编译处理工作，而实际上EOSProjectBuilder并没有对资源做实质的处理工作，它在做完编译预处理后，通过获取资源对应的IModelFactory实例，将资源的编译、验证等工作交由具体的模型工厂来处理。Studio中的各种资源的编译器如下所示：



模型验证器负责对模型进行校验，检查模型是否合法。IModelValidator是模型验证器的接口类。Studio中各种资源的验证器如下所示：



模型解析器负责资源文件的解析和保存工作，将资源文件解析成模型，同时将模型保存到资源文件中。IModelParser是模型解析器的接口类。Studio中的各种资源的模型解析器如下所示：



其中BizModelParser是逻辑流对应的模型解析器。需要强调的是IModelParser中还有一个很重要的方法toElement：

|  |
| --- |
| public IEosElement[] **toElement**(IEosModel model, IProgressMonitor monitor) throws ModelParseException, EosModelException; |

该方法有两个作用：

* 资源管理器（Navigator）中展现该资源时，会调用该方法来获取该资源要展现的子节点，该方法所返回的IEosElement数组中包含的元素就是该资源要展现的子节点。
* EOSProjectBuilder在对资源建索引时，也会调用这个方法来获知要建立哪些索引条目。

#### 元模型

Studio的元模型主要有两个作用：

1. 作为资源管理器的轻量级模型，资源管理器使用IEosElement作为节点的模型。
2. 模块之间通过元模型进行交互，有效降低了模块间的耦合。

Studio的元模型如下所示：



其中，IContribution是构件包节点的模型；IEosModel用来表示一个EOS资源模型，之所以说Studio的元模型是轻量级模型，是因为它只有在需要的时候才会调用解析器去获得真正的EOS资源模型，通过IEosModel#getModel方法可以获得重量级的EOS资源模型；IReference用来表示一种资源引用。Studio的元模型参考了JDT的Java元模型，大家有兴趣的话可以研究一下JDT的元模型，获得更多关于元模型的信息。

Studio的元模型位于com.primeton.studio.runtime插件中，请阅读相关代码获得更多信息。

#### UI框架

UI开发在Studio的开发工作中占了很大比重，如何提升UI的开发效率是个值得考虑的问题。

Eclipse使用SWT而不是Swing来处理UI需求，但是无论哪种UI，都只是提供最底层的功能，对于开发一个Studio而言，这些功能远不足以应付需求，如数据存取，布局，验证等功能，如果直接使用SWT进行开发，就需要编写大量的验证代码和相应的Listener。

Studio的UI框架是一套完整的UI框架，用来支持布局，验证，数据绑定，以及Undo和Redo功能，可以有效提高Studio的开发效率。

UI框架中的核心概念是Object Editor, Property Editor, Validator和IControlFactory；其中，Object Editor，顾名思义是“对象编辑器”，通过UI对一个对象进行编辑，一个对象有多个属性（property），这些属性的编辑通过Property Editor进行编辑，属性的值需要满足一定的规则，这些规则通过Validator进行校验。Object Editor、Property Editor和Validator之间的关系如下所示：



由上图可以看出，一个ObjectEditor对象包含了多个IPropertyEditor，而每个IPropertyEditor又包含了多个IValidator。ObjectEditor的element属性说明了UI要编辑的对象；IPropertyEditor有个重要的方法build，它用来构造编辑该属性所需要的Control对象，通常，IPropertyEditor通过关联一个IControlCreator，将实际Control对象的构建工作委托给具体的IControlCreator实现；那么属性编辑器如何从要编辑的element中获取和设置它所编辑的属性值呢？IPropertyEditor通过属性名称（Property Name）来标识它所要编辑的对象属性，然后通过Introspector来获取该属性所对应的值。

UI框架提供了两种Introspector接口的实现，如下所示：



其中，MapIntrospector用来从Map中获取property name所对应的值，这要求UI所编辑的应该是一个Map类型的对象；PropertyIntrospector利用OGNL来获取属性值，这通常要求UI所编辑的是一个Java Bean对象。我们可以根据需要实现新的Introspector来获取待编辑对象的属性值。

UI框架中还有一个很重要的对象IControlFactory，它用来构造一个具体的Control，通常，我们可以通过IControlFactory封装一个UI片段实现，从而促进某些公共特征的UI重用。IControlFactory相关对象的类图如下所示：



其中IControlFactory的createControl方法用来构造一个具体的Control对象，UI框架提供了IControlFactory的抽象实现AbstractControlFactory，我们如果要提供新的Control Factory只需要扩展自AbstractControlFactory即可。CompoundControlFactory可以包含多个IControlFactory，这通常用于复杂UI的构造，它的一个子类TabbedControlFactory是一个包含了多个Tab页的控件，逻辑流编辑器中的图元属性对话框大量使用了这个TabbedControlFactory。

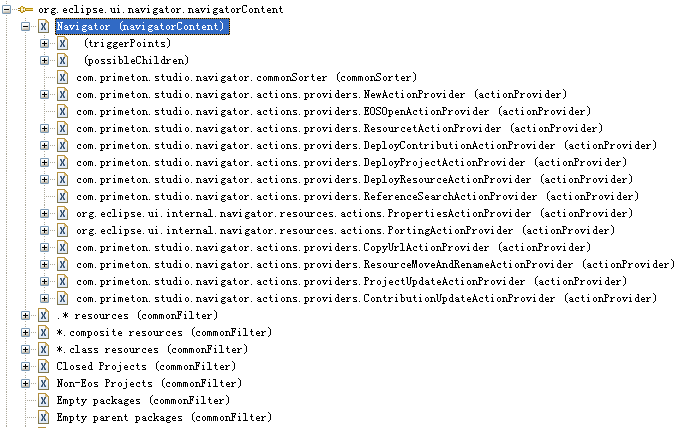
关于UI框架的更多内容请参阅《UI框架》文档。UI框架的源代码位于com.primeton.studio.ui插件中，请阅读相关源代码了解更详细的信息。

#### 资源管理器（Navigator）

资源管理器是开发人员使用最多的功能，它可以帮助开发人员更清楚的了解整个系统的结构，以及物理目录，可以通过各种方式对资源进行管理。一个资源管理器的好坏直接决定了一个IDE是否易用，好用。

Studio的资源管理器是基于Eclipse3.2新增的Common Navigator Framework (CNF)框架，通过扩展点来实现的，CNF对应的插件是org.eclipse.ui.navigator，该插件原是WTP平台上一个用于快速开发各种视图的插件，但在WTP平台的开发过程中，发现该插件的扩展性很好，可以通过各种方式来挂入虚拟结点，物理结点等，可以很容易的将各种系统通过不同的方式进行整合显示。如JDT和WTP都已经提供相应的扩展来支持这种视图。

CNF框架是一个很复杂的框架，提供了大量的扩展点，其中最主要的扩展点是org.eclipse.ui.navigator.navigatorContent，通过这个扩展点可以向资源管理器中提供新的内容，在注册新的navigatorContent时，需要提供content provider和label provider的实现。有关该扩展点的使用请阅读扩展点描述文档。Studio的资源管理器插件名为com.primeton.studio.navigator，在该插件中我们通过navigatorContent扩展点注册了EOS资源的展现，如下所示：



上图中的navigatorContent说明了我们所注册的资源管理器要显示的内容；triggerPoints表明了这个navigatorContent对哪些节点起作用；possibleChildren表明了这个navigatorContent允许显示的内容；commonSorter注册了用来对节点进行排序的排序器；actionProvider用来向资源管理器中添加右键菜单；commonFilter用来向资源管理器中注册过滤器。

Studio的资源管理器的content provider实现类是NavigatorContentProvider，在实现这个content provider时，参考了JDT的Navigator实现，引入了Workbench Adapter机制，通过为每个元素提供一个IWorkbenchAdapter实现，将元素子节点的获取、元素的显示委托给各自的IWorkbenchAdapter，并通过扩展点获取元素的Workbench Adapter，从而实现了解耦，可以看做是策略模式的一种实现。Studio中content provider相关的类图如下所示：



通过org.eclipse.core.runtime.adapters扩展点我们可以为特定的元素提供IAdapterFactory的实现，并通过IAdapterFactory#getAdapter方法获取元素的IWorkbenchAdapter实现。上图中，ContributionWorkbenchAdapter是对应的构件包节点的子节点元素获取器，ModelWorkbenchAdapter是IEosModel元素的孩子节点获取器。

Studio的资源管理器的Label Provider的实现类是NavigatorLabelProvider，其实现原理与Content Provider类似。类图如下所示：



资源管理器的一个很实用的功能是能够对所显示的元素进行绚烂，如：一个资源存在错误时，可以在资源的图片上显示一个错误小图标；团队开发时，一个资源在本地修改后，通常在资源的名称前添加一些特殊符号（>或\*）来说明该资源被修改了。在Studio中这个功能通过ProblemDecorator类来实现，在NavigatorLabelProvider为元素获得要显示的图片或文本后，会调用ProblemDecorator来检查该元素是否存在错误，如果存在错误，则对原图标进行绚烂。ProblemDecorator类会根据元素类型，通过扩展点来获取元素的IProblemAnalyser实例，由IProblemAnalyser来判断指定元素是否存在问题。因此，我们可以为特定的元素注册IProblemAnalyser的实现，Studio已经为我们提供了IProblemAnalyser的默认实现ResourceProblemAnalyser。

资源管理器还需要处理节点的刷新显示功能。当某个编辑器保存资源后，应该在资源管理器中刷新该资源对应的节点，以及所有引用了该资源的节点，这是为了：

1. 更新该资源的孩子节点；
2. 显示该资源是否存在错误；
3. 引用了该资源的其它资源节点是否存在错误；

如：逻辑流A被逻辑流B引用，编辑逻辑流A，为其添加一个输入参数，然后保存逻辑流A，此时B节点应该被刷新，并显示一个错误标记来说明所引用的逻辑流A参数不匹配。一种简便的做法是，只要发生资源保存动作就对资源管理器进行一次完全的刷新，但是，显而易见，这种做法会带来性能问题。好的做法是只刷新相关的节点，Studio中的ResourceMapper用来解决这个问题，该类实现了IResourceChangeListener接口，同时通过一个Map保存了资源与树节点之间的映射关系，当资源发生改变时，通过Map得到资源对应的树节点，并对这些树节点进行刷新。

资源管理器的代码位于Studio的com.primeton.studio.navigator插件中，阅读相关代码获取更详细的信息。

#### 图形化编辑器

EOS的一个重要特性是可以对业务逻辑和页面流转逻辑等功能具有可视化化装配和设计的能力。这在Studio中表现为逻辑流编辑器、页面流编辑器、数据集编辑器等各种图形化编辑器。

Studio基于GEF对图形化编辑器进行了封装，使得编辑器的开发人员不必了解GEF技术的底层细节。Studio的图形化编辑器主要包含两个模块，一个是基于EMF的编辑器的底层模型，另一个是由com.primeton.studio.gef.ui插件提供的图形化编辑器的抽象GEF实现。

基于EMF的编辑器底层模型的核心概念是：图（Diagram）、节点（Node）和连线（Connection），其中Diagram可以包含多个Node，Node与Node之间通过Connection相互连接，编辑器模型的类图如下所示：



Diagram就是图形化编辑器要编辑的顶层对象，它扩展自Node，而Node运用了Composite模式，形成了树形的层级（父子）关系，所以Diagram可以包含多个Node，TProcess是页面流、逻辑流编辑器要编辑的对象，DataSetDiagram是数据集编辑器所编辑的对象；NodeElement可以通过sourceConnections和targetConnections属性获得它的节点的连入线与连出线集合；同样，Connection可以通过source和target属性获得连线的源节点和目标节点。在Studio中上述模型通过XML Schema进行描述，并利用EMF生成模型代码。

基于编辑器的底层模型和GEF框架，Studio提供了图形化编辑器的抽象实现。AbstractGraphicalEditor是图形化编辑器的基类，当我们需要实现新的图形化编辑器时通常派生自该类，类图如下所示：



AbstractGraphicalEditor通过关联的GraphicalViewer来展现和绘制图形元素，GraphicalViewer通过关联的DiagramEditPartFactory获得某个模型元素所对应的EditPart。

要了解图形化编辑器的细节必须要了解GEF框架，请查阅Eclipse的相关文档获得GEF框架更详细的信息。Studio的GEF框架位于com.primeton.studio.gef.ui插件中，请阅读相关代码获得更多信息。

#### 索引

索引是任何IDE都必须具备的功能，可以说是IDE的灵魂！JDT之所以能够成为主流的Java开发工具，很大程度上取决于它所提供的功能强大又异常复杂的索引机制。IDE的许多功能，如：查找、重构，都离不开底层的索引功能。EOS基于Lucene实现了对EOS项目资源的索引，每个EOS资源对应Lucene中的一个Document，资源的相关属性被表示成Document中的Field。Studio中与索引有关的几个主要类如下所示：



IndexManager用来创建索引，它和IndexBuilderImpl都实现了IndexBuilder接口，IndexManager只是一个代理类，它维护了一个针对不同项目（project）的IndexBuilderImpl实例，并将索引的创建工作委托给具体的IndexBuilderImpl。IndexFinder用来查找索引。IndexManager和IndexFinder都实现了单例模式，在系统中可以很方便的获取它们的实例。

Studio中索引相关的代码位于com.primeton.studio.runtime插件中，请阅读相关代码获取更信息的信息。

## Studio总体开发过程

### 搭建Studio开发环境

进行Studio开发所需要的一般环境如下表所示：

|  |  |
| --- | --- |
| **组件** | **说明** |
| JDK 1.5.22 | Java基础环境 |
| Eclipse 3.2.1 | Eclipse基础平台，需要包含JDT和PDE |
| WTP 1.5.1 | Web Tools Platform |
| EMF 2.2.1 | Eclipse Modeling Framework |
| GEF 3.2.1 | Graphical Editing Framework |

EOS 开发版已经包含了所有上述组件，因此，我们可以安装一个EOS 开发版作为我们的Studio开发环境。

请获取并安装EOS的开发版，推荐使用最新的EOS 6.3。

### 导入源代码

Studio的源代码是最好的学习Studio的途径，在有源代码许可的条件下，可以将Studio的源代码导入到插件开发环境，这将方便后续插件开发的代码跟踪与调试。当然，这也不是必须的。

有两种途径可以导入Studio的源代码，一种是直接从CVS中获取Studio插件项目的源代码；另一种是通过源代码导入向导，将本地磁盘上已经存在的Studio插件项目源代码导入到插件开发环境中。

### 运行Studio

通过在Studio开环境中创建新的“Eclipse应用程序”运行项，我们可以在Studio开发环境中启动并运行一个新的Studio，这个Studio将包含工作空间中的插件项目。通过这个新运行的Studio，可以测试并调试我们的插件项目是否达到了功能要求。关于如何运行一个新的Studio，更详细的信息请参见插件开发入门中的“[运行插件项目](#_运行插件项目)”章节。

### 实现新功能

为Studio添加新功能或扩展已有的Studio功能通常需要进行设计、开发和调试这三个主要步骤。

#### 设计

设计阶段的主要任务：

1. 设计新功能的插件构成。确定为了实现新功能，需要创建哪些新的插件，每个插件的作用，这些新插件之间的依赖关系；
2. 设计插件之间的依赖关系，包括对已有插件的依赖关系；
3. 设计新功能可能的扩展，定义好扩展点。

#### 开发

根据设计阶段确定的插件，在插件开发环境中创建新的插件，建立插件之间的依赖关系，并编码实现新的功能。

#### 调试

通过在插件开发环境中，启动一个新的Studio，并设置合适的断点对新功能进行调试，修正存在的问题。

### 产品集成

#### 添加到功能部件中

功能部件（Feature）是插件的集合，通常将一些功能相关的插件（Plug-in）和段（Fragment）组织成功能部件，功能部件是Eclipse管理插件和段的基本单位，如：Eclipse的自动更新功能就是以功能部件为单位的。com.primeton.studio.features是Studio的功能部件项目，该项目中的feature.xml文件描述了该功能部件的相关信息，如：名称、版本号、所包含的插件和段。PDE提供了功能部件编辑器来编辑该文件。双击features.xml文件可以打开功能部件编辑器，在“插件”编辑页中，可以将需要的插件加入到这个功能部件（feature）中。

#### 添加到编译环境中

Studio提供了编译脚本用来对Studio中所包含的插件进行编译，打包成产品。我们新开发的插件也需要添加到产品中，这可以通过编辑com.primeton.studio.releng插件中的map文件来完成。Map文件描述了需要编译哪些插件和段，这些插件和段位于什么位置等信息。

## 基本规范

### 插件命名规范

1. Studio中的插件必须以com.primeton.studio作为名称前缀；
2. 进行某个新功能开发时，通常需要新建两个插件，假设这个新功能名为XXX，则这两个插件的命名如下：

|  |  |
| --- | --- |
| **插件名称** | **说明** |
| com.primeton.studio.XXX.core | 该插件用来存放新功能的模型代码 |
| com.primeton.studio.XXX.ui | 该插件用来存放于新功能相关的UI代码 |

1. 段的命名

需要为插件引入段时，段的名称=插件的名称+.fragment，如：需要为com.primeton.studio.biz.ui插件引入段，则这个段的名称为：

com.primeton.studio.biz.ui.fragment

### 插件依赖规范

1. Core插件可以依赖其它的Core插件，但不能依赖UI相关的插件；
2. 插件需要引用第三方类库时，考虑到重用性，尽量将这个第三方库以独立插件的形式提供。同时，Studio中的com.primeton.studio.library和com.primeton.studio.server.library两个插件提供了大量的第三方类库，请尽量依赖这两个插件提供的功能。

### 编码规范

请遵循普元统一的[编码规范](#编码规范)！

## 入门准备

### SWT

Standard Widget Toolkit（标准窗体工具包），是Eclipse提供的一个用户界面库。利用SWT创建的用户界面具有本地操作系统外观，以及更好的性能。Eclipse中的用户界面都是通过SWT实现的。

<http://www.eclipse.org/swt/docs.php>

### JFace

JFace在SWT的基础了又封装了一层，提供了更加高级、粒度更大的界面组件，如：对话框、向导等，JFace使得SWT更加易于使用。

<http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/jface.htm>

### Workspace

Workspace定义了一套API用以操作文件系统中的资源，并使资源具有可管理性。通过为资源实现附加属性，使资源更有意义。

<http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/resInt.htm>

### EMF

Eclipse Modeling Framework是Eclipse MDA的基础，其核心是基于MOF规范实现的Ecore模型。同时，EMF提供了一套强大的代码生成机制和模型编辑框架。

<http://www.eclipse.org/modeling/emf/>

### GEF

Graphical Editing Framework（图形编辑框架）是Eclipse封装的一套轻量级的交互式图形编辑框架，基于该框架可以实现美观的、高交互和复杂的图形交互。

<http://help.eclipse.org/helios/topic/org.eclipse.gef.doc.isv/guide.html>

### 相关书籍

1. Eclipse插件开发

<http://product.dangdang.com/product.aspx?product_id=20859426&ref=search-1-pub>

## 引入一种新的资源

图片包含 物体

描述已自动生成

我们将为EOS项目引入BPMN资源，让用户可以用BPMN标准来描述业务流程。BPMN（Business Process Modeling Notation，业务流程建模符号）是一套标准统一的业务流程建模符号。在BPMN之前，每个厂家在业务流程建模方面，都有自己的一套特有标准，这不利于业务流程的表述，也提高了业务人员的学习难度。BPMN规定了一套简单的、易于理解的图形符号，让用户可以清晰的表达业务流程。

我们将为Studio添加BPMN业务建模支持，并将BPMN模型编译成XPDL（XML Process Description Language）。

### 设计资源模型

出于演示目的，我们将简化BPMN模型，仅支持少量的BPMN图元。我们的BPMN模型仅支持单泳道（Swimlane），并用整个Diagram表示一个泳道；另外，我们的BPMN模型还将支持以下元素：

|  |  |  |
| --- | --- | --- |
| 元素 | 图形符号 | 说明 |
| 事件（Event） |  | 仅支持开始事件和结束事件 |
| 活动（Activity） |  | 仅支持任务（Task）活动 |
| Gateway |  |  |
| Sequence Flow |  |  |

在“[图形化编辑器](#_图形化编辑器)”章节的描述中，我们讲过Studio为了简化图形化编辑器开发的工作量，提高开发效率，封装了一套用于图形化编辑的，基于EMF的编辑器底层模型，我们的BPMN编辑器也将扩展自该模型。为此，我们设计了如下的BPMN资源模型：



其中，Process代码流程，它是我们BPMN资源的顶层元素；BPMN中的Event、Activity和Gateway都是节点元素，因此它们扩展自NodeElement，StartEvent和EndEvent是我们将支持的两种事件类型，Task是我们所支持的一种Activity；Sequence Flow是节点之间的连线，因此，它扩展自Connection。

为了利用EMF来生成代码，我们这里将使用XML Schema来描述我们的BPMN模型，

### 创建EMF模型

为了利用EMF来生成模型代码，我们将为我们所设计的BPMN资源模型创建对应的EMF模型。在Studio中创建EMF模型的主要步骤如下：

1. **创建模型插件**

我们将创建一个插件来存放资源相关的模型和代码。根据插件命名规范，我们创建了一个名为的插件，插件新建向导中的相关配置如下图所示：



激活器命名为BPMNCorePlugin，此外，该插件不需要依赖UI，因此，不需要勾选“此插件将对UI进行添加”；

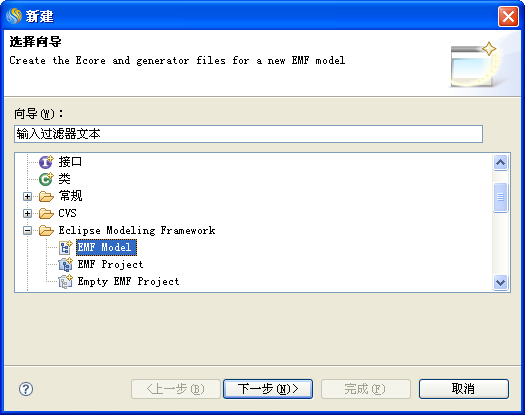
1. **创建XML Schema**
2. 在新建的com.primeton.studio.bpmn.core插件目录下创建schema目录；
3. 在schema目录下新建一个名为bpmn.xsd的文件；
4. 我们的BPMN模型的Schema描述如下所示：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:xsd="http://www.w3.org/2001/XMLSchema"  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" xmlns:eos="http://emf.primeton.com/core"  xmlns:bpmn="http://www.primeton.com/bpmn/1.0"  targetNamespace="http://www.primeton.com/bpmn/1.0"  elementFormDefault="qualified">    <!-- 导入EOS图形化编辑器基础模型 -->  <import namespace="http://emf.primeton.com/core" schemaLocation="AbstractModel.xsd"/>    <!-- 流程图定义 -->  <element name="process" type="bpmn:Process"/>  <complexType name="Process">  <complexContent>  <extension base="eos:Diagram">  </extension>  </complexContent>  </complexType>    <!-- 事件图元定义 -->  <complexType name="Event" abstract="true">  <complexContent>  <extension base="eos:NodeElement">  </extension>  </complexContent>  </complexType>  <complexType name="StartEvent">  <complexContent>  <extension base="bpmn:Event">  </extension>  </complexContent>  </complexType>  <complexType name="EndEvent">  <complexContent>  <extension base="bpmn:Event">  </extension>  </complexContent>  </complexType>    <!-- 活动定义 -->  <complexType name="Activity">  <complexContent>  <extension base="eos:NodeElement">  </extension>  </complexContent>  </complexType>  <complexType name="Task">  <complexContent>  <extension base="bpmn:Activity">  </extension>  </complexContent>  </complexType>    <!-- Gateway -->  <complexType name="Gateway">  <complexContent>  <extension base="eos:NodeElement">  </extension>  </complexContent>  </complexType>    <!-- sequence flow -->  <complexType name="SequenceFlow">  <complexContent>  <extension base="eos:Connection">  </extension>  </complexContent>  </complexType>    </schema> |

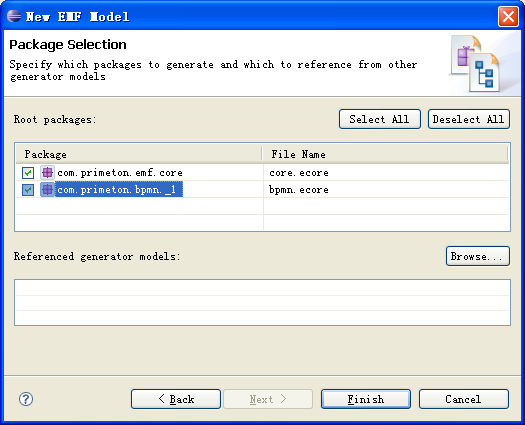
因为我们的BPMN模型是扩展自Studio图形化编辑器所提供的模型，因此bpmn.xsd文件需要导入Studio图形化编辑器的模型描述文件，请将下列的AbstractModel.xsd文件拷贝到schema目录下：

# 

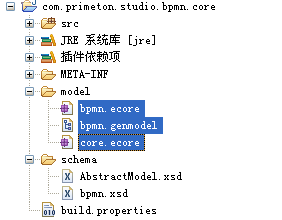
1. **创建EMF模型**
2. 在com.primeton.studio.bpmn.core插件目录下创建一个名为model的目录；
3. 在“包资源管理器”中，选择model目录，右键选择“新建🡪其他…”菜单；
4. 在弹出的新建向导中，选择Eclipse Modeling Framework分类目录下的EMF Model，如下所示：



1. 点击“下一步”，输入文件名为：bpmn.genmodel；
2. 点击“下一步”，选择从“XML Schema”导入模型；
3. 点击“下一步”，浏览工作空间，选择bpmn.xsd文件；
4. 点击“下一步”，选择包列表中列出的两个包，修改com.primeton.bpmn.\_1包对应的文件名为“bpmn.ecore”，如下图所示：



1. 点击“完成”，将在model目录下生成三个文件，如下图所示：



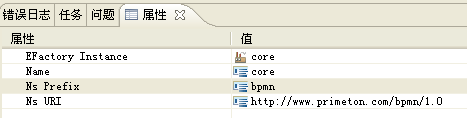
其中，bpmn.ecore文件是我们的BPMN模型对应的Ecore模型，bpmn.genmodel是Ecore模型对应的代码生成模型，其中配置了代码生成的相关信息，core.ecore是Studio图形化编辑器底层模型的Ecore表述。

### 生成模型代码

我们将对BPMN的Ecore模型和代码生成模型进行配置，并生成最终的BPMN模型代码。

编辑Ecore模型的主要步骤如下：

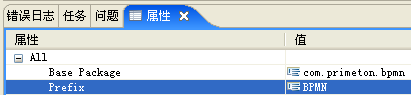
1. 双击bpmn.ecore文件，将打开Ecore模型编辑器；
2. 在编辑器中，展开BPMN的Ecore模型，选择名为\_1的package，右键选择“Show Properties View”菜单，将显示属性视图；
3. 编辑“\_1”package的各个属性值，如下所示：



1. 保存bpmn.ecore文件；

编辑BPMN代码生成模型的主要步骤如下：

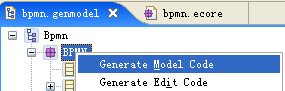
1. 双击bpmn.genmodel文件，系统将代码生成编辑器；
2. 在编辑器中，选择名为“-1”的pacakge，在属性视图中，修改它的前缀（Prefix）为BPMN，如下图所示：



1. 保存bpmn.genmodel文件。

生成模型代码：

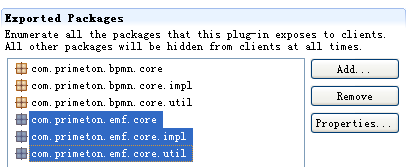
1. 在bpmn.genmodel的编辑器中，选择“BPMN”包节点；
2. 右键选择“Generate Model Code”菜单，如下所示：



EMF将为我们生成基于EMF的BPMN模型代码。你会发现生成的代码存在编译错误，因此你需要完成下列步骤：

1. 因为我们的BPMN模型代码扩展自Studio图形编辑器的模型代码，该代码位于com.primeton.studio.server.library插件中，因此com.primeton.studio.bpmn.core插件需要添加对com.primeton.studio.server.library(这快写错了)插件的依赖。
2. 该插件不需要对外提供扩展，因此在插件清单编辑器中，删除“扩展”中EMF所添加的扩展。
3. 删除不必要的导出包。

在插件清单编辑器中，选择“运行时”页，在“导出的包”中，删除下图所示的三个包：



至此，我们已经为我们的BPMN资源生成了模型代码。EMF为每个模型元素都生成了接口和实现类，接口位于com.primeton.bpmn.core包中，实现类位于com.primeton.bpmn.core.impl包中；同时，EMF在com.primeton.bpmn.core.util包中生成了一些工具类，这些工具类主要用来辅助资源存储的。另外，所生成的代码中，还有两个很重要的类，一个是BPMNFactory，另一个是BPMNPackage，其中，BPMNFactory是一个工厂类，用来创建（生产）模型元素，BPMNPackage中保存了模型的元数据信息。有兴趣的同学，可以进一步学习一下EMF相关的内容。

### 对模型进行测试

我们可以写一个简单的测试类对BPMN模型进行序列化和反序列化测试，如下所示：

|  |
| --- |
| **import** java.io.ByteArrayOutputStream;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** org.eclipse.emf.common.util.URI;  **import** org.eclipse.emf.ecore.resource.Resource;  **import** com.primeton.bpmn.core.BPMNFactory;  **import** com.primeton.bpmn.core.BPMNPackage;  **import** com.primeton.bpmn.core.DocumentRoot;  **import** com.primeton.bpmn.core.EndEvent;  **import** com.primeton.bpmn.core.Process;  **import** com.primeton.bpmn.core.SequenceFlow;  **import** com.primeton.bpmn.core.StartEvent;  **import** com.primeton.bpmn.core.impl.BPMNFactoryImpl;  **import** com.primeton.bpmn.core.util.BPMNResourceFactoryImpl;  **public** **class** BPMNModelTester {    **public** BPMNModelTester() {  }    **public** **void** test() **throws** Exception {  save(); //保存模型  load(); //加载模型  }    **private** **void** load() **throws** Exception {  Resource resource = (**new** BPMNResourceFactoryImpl()).createResource(  URI.*createURI*(BPMNPackage.*eNS\_URI*));  FileInputStream input = **new** FileInputStream("c:\\test.bpmn");  resource.load(input, **null**);  DocumentRoot documentRoot = (DocumentRoot)resource.getContents().get(0);  Process process = documentRoot.getProcess();  System.*out*.println(process.getName());  }    **private** **void** save() **throws** Exception {  ByteArrayOutputStream outStream = **new** ByteArrayOutputStream();  Resource resource = (**new** BPMNResourceFactoryImpl()).createResource(  URI.*createURI*(BPMNPackage.*eNS\_URI*));  BPMNFactory factory = **new** BPMNFactoryImpl();  DocumentRoot documentRoot = factory.createDocumentRoot();  Process process = createProcess();    documentRoot.getMixed().add(BPMNPackage.Literals.*DOCUMENT\_ROOT\_\_PROCESS*, process);  resource.getContents().add(documentRoot);  resource.save(outStream, **null**);    FileOutputStream file = **new** FileOutputStream("c:\\test.bpmn");  file.write(outStream.toByteArray());  file.close();  }    **private** Process createProcess() {  BPMNFactory factory = BPMNFactory.*eINSTANCE*;    // 创建一个流程  Process process = factory.createProcess();  process.setName("MyProcess");    // 创建Start Event  StartEvent start = factory.createStartEvent();  start.setId("1");  start.setName("开始");    // 创建一个Task  Task task = factory.createTask();  task.setId("2");  task.setName("任务");    // 创建一个End Event  EndEvent end = factory.createEndEvent();  end.setId("3");  end.setName("结束");    // 创建Start Event到Task的连线  SequenceFlow startToTask = factory.createSequenceFlow();  start.getSourceConnections().add(startToTask);  task.getTargetConnections().add(startToTask);    // 创建Task到End Event的连线  SequenceFlow taskToEnd = factory.createSequenceFlow();  task.getSourceConnections().add(taskToEnd);  end.getTargetConnections().add(taskToEnd);    process.getNodes().add(start);  process.getNodes().add(task);  process.getNodes().add(end);  **return** process;  }    **public** **static** **void** main(String[] args) **throws** Exception {  BPMNModelTester tester = **new** BPMNModelTester();  tester.test();  }  } |

其中，save()方法测试了模型的序列化，load()方法测试了模型的反序列化，而createProcess()方法则说明了如何利用生成的EMF代码来创建一个BPMN模型。

## 让Studio认识资源

本章我们将讲述如何让Studio认识新引入的资源，并对资源进行管理，包括：资源的注册、解析、编译、验证和索引。

### 注册

在Studio中，要想让Studio认识你的资源，需要为资源注册一个IModelFactory实例，关于IModelFactory的相关信息请参阅“[模型管理](#_模型管理)”。为BPMN模型提供model factory的主要步骤如下：

1. **创建BPMNModelFactory类**

该类必须实行IModelFactory接口。Studio已经为我们提供了抽象的IModelFactory实现AbstractModelFactory，通常只需要扩展自该类即可。如果是图形化编辑器模型，Studio为我们提供了AbstractGefModelFacotry的实现，因此，我们的BPMNModelFactory将扩展自AbstractGefModelFactory。

1. **添加插件依赖**

为了BPMNModelFacotry类能顺利编译通过，我们需要为com.primeton.studio.bpmn.core插件添加插件依赖，所依赖的插件如下：

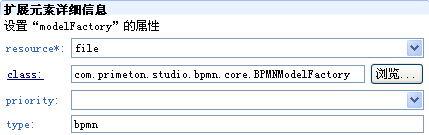
com.primeton.studio.runtime

com.primeton.studio.core

com.primeton.emf.core

1. **注册modelFactory扩展**

Studio为model factory提供了注册扩展点：com.primeton.studio.runtime.modelFactory，通过这个扩展点我们可以将BPMN模型工厂注册到Studio的模型管理器中，进行统一管理。打开com.primeton.studio.bpmn.core的插件清单编辑器，在“扩展”中通过com.primeton.studio.runtime.modelFactory扩展点，添加BPMNModelFactory扩展，配置信息如下：



其中，type属性说明了BPMNModelFactory所管理的资源的扩展名，这里值为bpmn。

1. **实现BPMNModelFactory**

通过BPMNModelFactory，Studio模型管理器可以获得后缀名为bpmn资源的模型解析器、模型编译器和模型验证器等相关信息，该类的实现如下所示：

|  |
| --- |
| **public** **class** BPMNModelFactory **extends** AbstractGefModelFactory {    **private** BPMNModelParser modelParser = **new** BPMNModelParser();    **private** BPMNModelCompiler modelCompiler = **new** BPMNModelCompiler();    **public** BPMNModelFactory() {  **super**();    //通过优先级设置节点的显示顺序  **this**.setPriority(OrderConstant.*ORDER\_CONFIGURATION*+1);  }  **public** Object createModel(IProgressMonitor arg0) {  **return** BPMNPackage.*eINSTANCE*.getBPMNFactory().createProcess();  }  **public** String getBundleName() {  **return** **null**;  }  **public** String getCategory() {  **return** "BPMN";  }  **public** IModelCompiler getCompiler(Object arg0, IProgressMonitor arg1) {  **return** **this**.modelCompiler;  }  **public** IModelParser getModelParser(IFileDelegate arg0, IProgressMonitor arg1) {  **return** **this**.modelParser;  }  **public** IModelValidator getValidator(Object arg0, IProgressMonitor arg1) {  **return** **null**;  }  **public** String getDisplayName() {  **return** **null**;  }  **public** String getName() {  **return** "bpmn";  }  } |

这里的BPMNModelParser和BPMNModelCompiler我们将在后续实现。

### 解析

模型解析器用来对模型进行序列化和反序列化操作，其主要方法是save和parse。我们将为BPMN模型实现BPMNModelParser类，该类是IModelParser的一个实例。BPMNModelParser的实现如下：

|  |
| --- |
| **public** **class** BPMNModelParser **extends** AbstractGefModelParser {    **static** {  Map registry = EPackage.Registry.*INSTANCE*;  String bpmnURI = BPMNPackage.*eNS\_URI*;  registry.put(bpmnURI, BPMNPackage.*eINSTANCE*);  registry.put(CorePackage.*eNS\_URI*, CorePackage.*eINSTANCE*);  }  @Override  **public** Resource createResource() {  ResourceFactoryImpl resourceFactory = **new** BPMNResourceFactoryImpl();  **return** resourceFactory.createResource(URI.*createURI*(BPMNPackage.*eNS\_URI*));  }  @Override  **public** Object parse(InputStream inStream, IProgressMonitor monitor) **throws** ModelParseException {  **try** {  Resource resource = **this**.createResource();  **if**(inStream.available() == 0){  **return** **null**;  }  resource.load(inStream, getOptions());    List contents = resource.getContents();  **if**(contents!=**null** && contents.size()>0) {  DocumentRoot documentRoot = (DocumentRoot)contents.get(0);  **return** documentRoot.getProcess();  }  } **catch** (Exception e) {  **throw** **new** ModelParseException(e);  }    **return** **null**;  }  @Override  **public** **void** save(IFileDelegate file, Object object) **throws** IOException {  ByteArrayOutputStream outStream = **new** ByteArrayOutputStream();  Resource resource = **this**.createResource();  BPMNFactory factory = **new** BPMNFactoryImpl();  DocumentRootImpl documentRoot = (DocumentRootImpl) factory.createDocumentRoot();  documentRoot.getMixed().add(BPMNPackage.Literals.*DOCUMENT\_ROOT\_\_PROCESS*, object);  resource.getContents().add(documentRoot);  resource.save(outStream, getOptions());  ResourceHelper.*write*(file, outStream.toByteArray());  }  } |

该类扩展自AbstractGefModelParser，静态初始化方法中，我们向EMF模型管理器中注册了BPMN模型和图形化编辑器模型的URI所对应的EPackage；save()方法中实现了BPMN模型的保存，parse方法负责解析输入流获得BPMN模型（Process实例）。

IModelParser还有一个重要的方法toElement：

|  |
| --- |
| **public** IEosElement[] toElement(IEosModel model,  IProgressMonitor monitor)  **throws** ModelParseException, EosModelException; |

该方法有两个主要的用途：

* 生成轻量级模型

在资源管理器中显示资源的子节点时，会调用该方法获取IEosModel所关联的重量级模型，并根据重量级模型生成对应的轻量级模型，用于资源管理器的节点展现。

* 创建索引

EOS项目编译器在编译EOS项目时，会调用该方法，获取IReference元素列表，并用于索引的创建。

### 编译

模型编译器用来对模型进行编译处理。当后缀名为bpmn的资源文件发生变化时，Studio的增量编译器（EOSProjectBuilder）将通过BPMNModelFactory获取该资源的编译器，并调用其中的编译方法对资源进行编译处理。

BPMN模型通常可以映射成XPDL或BPEL4WS模型，这两种模型都是业务流程的执行模型，可以根据需要将我们的BPMN模型映射成其中的一种。

我们的BPMN模型编译器名为BPMNModelCompiler，该类是IModelCompiler的实例，Studio已经为我们提供了抽象实现AbstractModelCompiler，我们只需要扩展自该类，并实现doCompile方法即可。出于演示目的，我们这里只对模型进行了拷贝，并没有映射到XPDL，如下所示：

|  |
| --- |
| **public** **class** BPMNModelCompiler **extends** AbstractModelCompiler {  @Override  **protected** IResourceDelegate[] doCompile(  IFileDelegate fileDelegate,  **int** buildKind,  Object object,  **int** validateState,  IProgressMonitor progressMonitor,  IMessageCaller messageCaller) {  IFileDelegate outputFile = ResourceHelper.*getOutputFile*(fileDelegate, "xpdl");  IFolderDelegate parentFolder = outputFile.getParent();  **if** (!parentFolder.exists()) {  parentFolder.create();  }  ResourceHelper.*copy*(fileDelegate, outputFile);  **return** **new** IResourceDelegate[] {  outputFile  };  }  } |

doCompile方法的主要参数说明如下：

* fileDelegate：要编译的源文件；
* buildKind：编译的类型，是增量编译还是全编译；
* object：BPMN模型对象
* validateState：模型校验结果，可以根据它来决定是否要生成输出文件；
* 返回值IResourceDelegate[]：编译输出文件集。

根据我们的编译器实现，BPMN模型编译成功后，将在输出目录下生成一个同名的，后缀名为xpdl的文件。

### 验证

验证的目的是为了检查模型中存在的错误。如果存在错误，可以在问题视图中显示错误信息，提示用户错误的原因，以便用户及时更正存在的错误。

BPMNModelValidator类用于对BPMN模型进行校验，该类是IModelValidator的实例。Studio已经为我们提供了IModelValidator的抽象实现AbstractModelValidator，因此，BPMNModelValidator将扩展自该类，我们只需要实现validate方法即可，如下所示：

|  |
| --- |
| **public** **class** BPMNModelValidator **extends** AbstractModelValidator {  **public** **boolean** validate(  IFileDelegate fileDelegate,  Object object,  ModelChangeEvent modelChangeEvent,  IMessageCaller messageCaller,  IProgressMonitor progressMonitor) {  **return** **true**;  }  } |

Validate方法的参数说明如下：

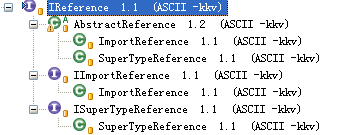
* fileDelegate：待验证的模型文件；
* object: 待验证的模型；
* modelChangeEvent：ModelChangeEvent描述了引起模型验证的原因，其中包含了验证的上下文信息；
* messageCaller：IMessageCaller的实例，在Eclipse环境下，该实例将向“问题”视图中输出错误信息；
* 返回值是布尔类型：true表示模型验证通过；

出于演示目的，我们的validate方法只是简单的返回true来说明模型没有错误，您可以根据需要对模型元素进行遍历，检查存在的错误。

### 索引

索引表明了资源之间的引用关系。如果允许我们的BPMN模型调用逻辑流、运算逻辑等资源，则我们需要为BPMN模型建立对逻辑流等资源的引用关系，并通过索引来查询这种引用关系。当所引用的逻辑流等资源发生变化时，studio的模型管理器将通过反向索引来查询该逻辑流被哪些资源所引用，并调用这些资源的验证器进行模型校验。

引用关系通过IReference接口进行描述，存在两种类型的引用关系，一种是超类引用（ISuperTypeReference），另一种是导入引用（IImportReference），通常我们都是有导入引用，IReference的层次关系如下图所示：

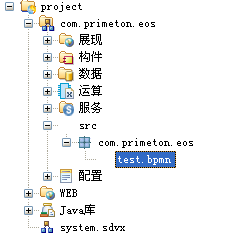


Studio中是通过IModelParser的toElement方法来建立资源的引用关系的。我们可以重载BPMNModelParser的toElement方法来建立BPMN模型对逻辑流等资源的引用关系。

### 运行Studio

至此，我们的BPMN资源已经可以被Studio认识了，Studio可以对后缀名为bpmn的资源进行统一管理了。请参见“[运行插件项目](#_运行插件项目)”章节，在插件开发环境中启动被运行一个新的Studio。

在新运行的Studio中创建一个EOS项目，可以看到构件包节点下多了一个名为“src”的节点，这就是我们的BPMN资源对应的资源分类节点。我们可以将“[对模型进行测试](#_对模型进行测试)”章节中生成的test.bpmn资源拷贝到该节点的com.primeton.eos包下，可以看到资源管理器在com.primeton.eos包下显示出了test.bpmn资源，如下所示：



现在显示的“src”节点和“test.bpmn”节点还未经修饰，我们将在“在资源管理中展现资源”章节介绍如何为这两个节点添加图片和修改显示的文字。在下一章，我们将介绍如何在Studio中新建一个BPMN资源。

## 新建资源文件

Studio通常都需要为所管理的资源提供新建向导，我们将在本章为BPMN资源提供新建向导。

### 新建向导框架

Studio已经在JFace向导框架的基础上封装了一套资源新建向导框架，如下图所示：



AbstractNewWizard是核心类，它持有IWorkbench引用，它的selection属性保存了用户在向导的资源管理器中所选择的树节点对象；AbstractCommonWizard有一个向导页CommonWizardPage，它派生自AbstractResourceWizardPage，这个向导页中有一个ProjectViewer属性，它是一个资源管理器对象，用来让用户选择新建资源将位于哪个项目的哪个包下，ObjectEditor是UI框架中的一个对象，用来编辑对象属性的。

在Studio中，绝大多数的资源新建向导都是扩展自AbstractCommonWizard。

向导框架的代码位于com.primeton.studio.workbench.wizard插件中，请大家阅读相关代码了解更详细的信息。

### 实现BPMN新建向导

我们的BPMN资源新建向导将继承自AbstractCommonWizard。在实现BPMN新建向导之前，我们需要为BPMN资源新建一个UI插件（com.primeton.studio.bpmn.ui），如下所示：



为UI插件添加插件依赖：com.primeton.studio.ui和com.primeton.studio.workbench.wizard。

我们将为BPMN资源实现一个名为NewBPMNWizard的向导，主要步骤如下：

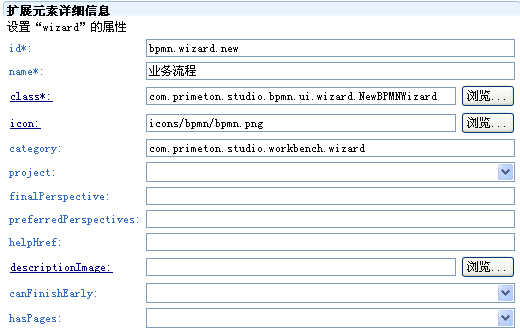
1. 在插件com.primeton.studio.bpmn.ui插件中，新建包com.primeton.studio.bpmn.ui.wizard；
2. 在包com.primeton.studio.bpmn.ui.wizard下创建NewBPMNWizard类，该类扩展自AbstractCommonWizard；
3. 实现NewBPMNWizard类，代码如下所示：

|  |
| --- |
| **public** **class** NewBPMNWizard **extends** AbstractCommonWizard {    **public** NewBPMNWizard() {  **this**.setWindowTitle("新建业务流程");  }  @Override  **protected** **void** updatePageMessage(CommonWizardPage commonWizardPage) {  commonWizardPage.setTitle("新建业务流程");  commonWizardPage.setDescription("在指定的项目中创建一个新的业务流程");  }  @Override  **protected** String getType() {  **return** "bpmn"; //待新建的资源的扩展名  }  } |

由于我们的BPMN新建向导比较简单，AbstractCommonWizard类所提供的功能已经够用了，我们只是对向导进行了一些简单配置。如果需要为向导添加一个新的向导页，可以重载AbstractCommonWizard的addPages方法。

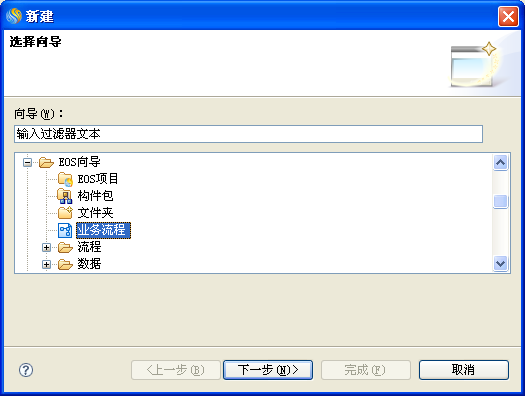
### 注册向导

Eclipse提供了org.eclipse.ui.newWizards扩展点，允许用户为Eclipse添加新建资源向导。我们将通过这个扩展点，将我们的BPMN资源新建向导挂接到Eclipse的向导管理框架中。可以为org.eclipse.ui.newWizards扩展点添加三类子元素：category、wizard和primaryWizard，其中category是为Eclipse的向导管理器添加一个新的分类节点；primaryWizard是注册一个主向导，这个主向导将显示在Eclipse的“文件🡪新建”菜单中；wizard是注册一个普通向导。我们这里将新建一个wizard元素，该元素的配置信息如下：



其中的category属性表明，这个BPMN资源新建向导将显示在“EOS向导”分类目录下。

运行Studio，可以看到我们的“业务流程”新建向导已经在Eclipse的新建向导管理器中列出来了，如下所示：



点击“下一步”将进行新建业务流程向导，如下所示：



填写相关信息后，点击“完成”按钮，向导将在指定目录下创建一个名为mybpmn.bpmn的文件。那么向导是如何创建出一个BPMN资源的呢？仔细查看AbstractCommonWizard#doFinish方法，如下所示：

|  |
| --- |
| IFileDelegate fileDelegate = **new** EclipseFileDelegate(file);  Object model = RuntimeHelper.*createModel*(fileDelegate);  IModelParser parser = RuntimeHelper.*getModelParser*(fileDelegate);  **if** (**null** != parser) {  parser.save(fileDelegate, model);  } |

我们发现，在向导结束时，会调用RuntimeHelper的createModel方法来创建一个新的BPMN模型，而RuntimeHelper#createModel方法会根据资源的扩展名从模型管理器（IModelManager）中获取IModelFactory实例，再调用IModelFactory#createModel方法来创建一个模型；在创建了模型之后，会取得模型解析器来保存模型。

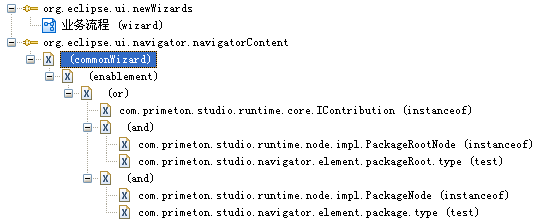
### 注册到资源管理器

为了方便用户创建资源，通常需要将资源新建向导挂接到资源管理器中。我们希望我们的BPMN新建向导能够在资源管理器的“构件包”、“包”等节点的右键菜单中进行调用。

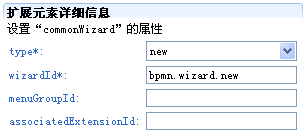
Eclipse的CNF框架已经为我们提供了相关的扩展点，允许我们将新建向导挂接到资源管理器的右键菜单中，这个扩展点为：org.eclipse.ui.navigator.navigatorContent，它的功能很复杂，您可以通过阅读该扩展点的描述获得更详细的信息。

主要步骤如下：

1. 由于需要使用CNF提供的扩展点，因此，我们的com.primeton.studio.bpmn.ui插件需要添加对org.eclipse.ui.navigator插件的依赖；
2. 添加org.eclipse.ui.navigator.navigatorContent扩展，配置信息如下：



其中commonWizard元素的属性设置如下：



Type属性用来标识向导的类型；wizardId属性标识了所引用的向导id，这个id与我们之前通过org.eclipse.ui.newWizards扩展点注册的业务流程新建向导的id相同。

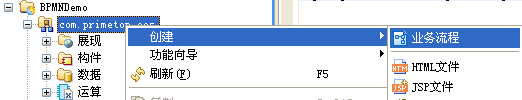
enablement元素用来配置一个条件表达式，当用户在资源管理器中选择一个节点并触发右键显示时，CNF框架会根据用户所选择的节点元素对表达式进行求值，只有条件为真时，这个向导才会在右键菜单中显示。

Instanceof用来做实例判断；

test用来对所选择的元素做属性测试。这里com.primeton.studio.navigator.element.packageRoot.type测试的值（value）为bpmn；com.primeton.studio.navigator.element.package.type测试的值为bpmn。

我们配置的条件表达式表明：业务流程新建向导允许出现在构件包节点、流程节点和包节点的右键菜单中。

1. 运行Studio中，可以看到构件包的右键菜单中多了一个“创建🡪业务流程”菜单项，如下所示：



## 在资源管理器中展现资源

资源管理器中，现在已经能够展现BPMN资源了，但是表现力还不够，我们希望节点显示更有已有的名字，为节点添加图片修饰使其更加直观，同时，我们希望BPMN资源节点能够显示BPMN模型中的一些子节点。我们将在本章指导大家如何做到上述这些。

### 开始之前

请阅读“[资源管理器](#_资源管理器（Navigator）)”章节获得资源管理器的一些基本知识。

Studio资源管理器的节点与模型的对应关系如下：



IContribution

CategoryNode

PackageRootNode

PackageNode

PackageRootNode

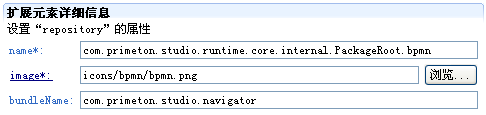
EosModel

前面我们讲过，每个节点都有对应的IWorkbenchAdapter，通过IWorkbenchAdapter可以获得该节点要显示的文字和图片。IContribution对应的Workbench Adapter是ContributionWorkbenchAdapter，查看它的getChildren方法，我们就可以知道构件包（IContribution）节点下包含了哪些子节点；通过getLabel方法获得节点要显示的文字；通过getImageDescriptor方法获得节点要显示的图片。

### 为节点添加图片

我们的BPMN资源的根节点（对应的类型为PackageRootNode），目前没有显示图片，不够直观，这里我们将为该节点添加图片。

PackageRootNode对应的Workbench Adapter是PackageRootNodeWorkbenchAdapter，通过查看它的getImageDescriptor实现，我们可以通过com.primeton.studio.ui插件提供的imageRepository扩展点来为节点注册一张图片。打开com.primeton.studio.bpmn.ui插件的插件清单编辑器，通过com.primeton.studio.ui.imageRepository扩展点为节点注册一张图片，如下所示：

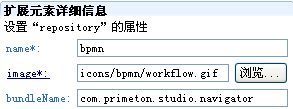


根据PackageRootNodeWorkbenchAdapter#getImageDescriptor的实现：

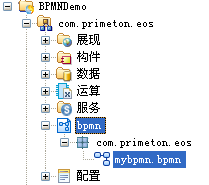
|  |
| --- |
| ……  ImageRepository.*getImageDescriptor*(NavigatorPlugin.*PLUGIN\_ID*, packageRoot.getClass().getName() + "." + packageRootNode.getType());  …… |

这里的name必须是节点对象的类全名加上节点的类型（type），这里的packageRootNode#getType方法取的是BPMNModelFactory#getName的返回值，所以这里的name是：com.primeton.studio.runtime.core.internal.PackageRoot.bpmn；image是所注册的图片；同样，根据getImageDescriptor方法的实现，bundleName的值应该是资源管理器插件的标识符：com.primeton.studio.navigator。

同样的，我们可以为BPMN资源的模型节点（对应的类型为EosModel）添加图片。EosModel节点类型对应的Workbench Adapter是ModelWorkbenchAdapter，查看该类的getImageDescriptor方法实现，我们为BPMN资源的模型节点的图片进行了如下配置：



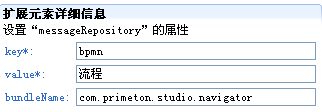
运行Studio，可以看到BPMN资源相关的节点已经显示了我们所注册的图片，如下所示：



### 修改节点文字

我们的BPMN资源的根节点（PackageRootNode），目前显示的是“bpmn”文字，我们希望该节点显示成“流程”。

通过查看它的PackageRootNodeWorkbenchAdapter#getLabel实现，我们可以通过com.primeton.studio.ui插件提供的messageRepository扩展点来为节点注册一个文字描述。打开com.primeton.studio.bpmn.ui插件的插件清单编辑器，通过com.primeton.studio.ui.messageRepository扩展点为节点注册一个文字描述，如下所示：



其中，key值为bpmn，根据PackageRootNodeWorkenchAdapter#getLabel的实现，key值对应到BPMNModelFactory#getName方法的返回值，所以这里是bpmn；value就是节点要显示的文字；同样，根据getLabel方法的实现，bundleName的值应该是资源管理器插件的标识符：com.primeton.studio.navigator。

运行Studio，可以发现BPMN资源的根节点的名称已经显示成了“流程”。

### 显示子节点

我们将在资源管理器的BPMN模型下面显示出事件（Event）和任务（Task）子节点。

负责处理BPMN模型节点的是ModelWorkbenchAdapter，查看它的getChildren方法实现，可以发现最终是调用的IEosModel的getChildren方法来获取模型的子节点的，而IEosModel获取子节点的代码（见EosModel#doLoadChildren方法）如下：

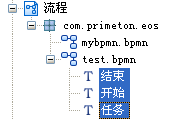
|  |
| --- |
| IEosElement[] elements = RuntimeHelper.*toElement*(**this**);  **if** (**null** != **this**.modelCacheStrategy) {  **this**.modelCacheStrategy.updateCache();  }  **return** elements; |

从上面的代码可以发现是通过RuntimeHelper的toElement方法来获取的孩子节点，跟踪代码进一步发现EosModel获取子节点最终是通过获取对应的模型解析器（IModelParser），并调用其中的toElement方法来获取的子节点。因此，我们需要重写BPMNModelParser的toElement方法，如下所示：

|  |
| --- |
| @Override  **public** IEosElement[] toElement(IEosModel eosModel, IProgressMonitor monitor) **throws** ModelParseException, EosModelException {  Object model = eosModel.getModel();  **if**(model == **null**) {  **return** IEosElementConstant.*NO\_CHILDREN*;  }    // 子节点列表  List<IEosElement> elements = **new** ArrayList<IEosElement>();    IResourceDelegate resourceDelegate = eosModel.getResource();  Process process = (Process)model;  List<NodeElement> nodes = process.getNodes();  **for**(Iterator<NodeElement> iter = nodes.iterator(); iter.hasNext();) {  NodeElement node = iter.next();  **if**(node **instanceof** Event || node **instanceof** Task) {  Type event = **new** Type(resourceDelegate, eosModel);  event.setName(node.getName());  event.setModel(node);  elements.add(event);  }  }  **return** elements.toArray(**new** IEosElement[elements.size()]);  } |

该方法首先通过eosModel#getModel方法获得轻量级模型（IEosModel）所关联的重量级模型（这里是Process），然后遍历Process的各个图元（NodeElement），如果是事件（Event）或者任务（Task），则用元模型中的Type来表示这个子节点，最后返回子节点列表。

运行Studio，可以看到我们在“对模型进行测试”章节中生成的test.bpmn文件列出了子节点，如下图所示：



(引core依赖pross用core的)

### 添加过滤器

CNF框架提供了过滤器扩展点：org.eclipse.ui.navigator.navigatorContent#commonFilter，允许我们通过这个扩展点向资源管理器注册一个新的过滤器来显示或隐藏某些节点。

我们将向资源管理器注册一个过滤器来隐藏BPMN模型中的事件（Event）子节点，主要步骤描述如下：

1. **实现事件过滤器**

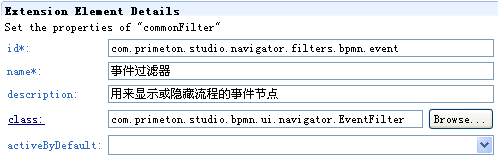
资源管理器的过滤器必须扩展自org.eclipse.jface.viewers.ViewFilter类。因此，我们在com.primeton.studio.bpmn.ui插件中，新建一个com.primeton.studio.bpmn.ui.navigator.EventFilter类，该类如下：

|  |
| --- |
| **public** **class** EventFilter **extends** ViewerFilter {  @Override  **public** **boolean** select(Viewer viewer, Object parentElement, Object element) {  **if**(element **instanceof** IType) {  Object model = ((IType)element).getModel();  **if**(model **instanceof** Event)  **return** **false**;  }    **return** **true**;  }  } |

资源管理器在获得子节点后，将调用所注册的过滤器的select方法来判断是否显示指定的节点。select方法返回true，表示允许显示该节点；返回false，表示不显示该节点。

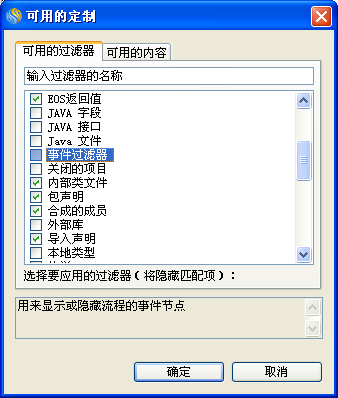
1. **注册过滤器**

通过org.eclipse.ui.navigator.navigatorContent扩展点来注册EventFilter。navigatorContent扩展点通过commonFilter子元素注册过滤器，我们的事件过滤器扩展配置如下：



Id是该过滤器在系统中的唯一标识符，必须以com.primeton.studio.navigator开头；class是过滤器的实现类，必须派生自ViewerFilter；activeByDefault用来配置该过滤器是否默认激活。

运行Studio，在资源管理器的过滤器列表中可以看到我们注册的事件过滤器，如下所示：



勾选“事件过滤器”，点击确定，资源管理器将刷新显示。可以看到，我们的test.bpmn资源的开始事件和结束事件子节点被过滤掉了。

## 对资源进行编辑

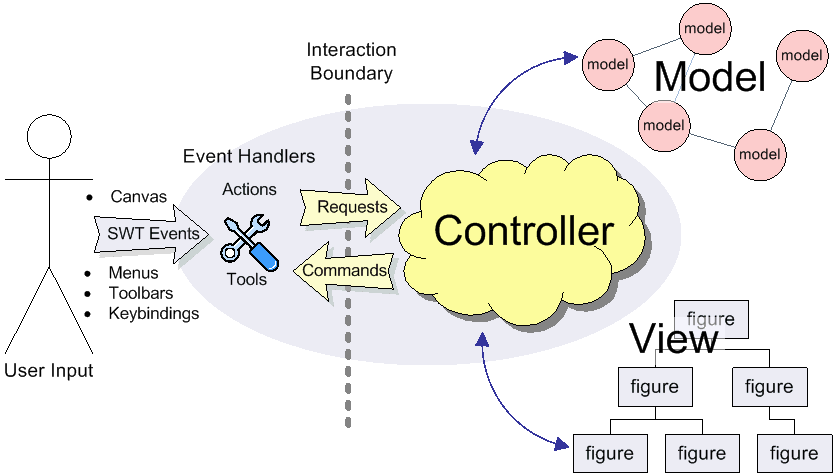
本章我们将利用Studio的GEF框架来实现一个图形化的BPMN编辑器，实现对BPMN的简单编辑。您将学习到GEF框架，以及Studio的GEF框架的相关内容。

### 开始之前

#### MVC与GEF

MVC（Model-View-Controller），模型-视图-控制器，是一种常见的软件架构模式，它把系统分成三个基本部分：模型（Model）、视图（View）和控制器（Controller）。MVC模式解决了模型层与视图层之间的耦合关系，使它们可以独立的进行变化，并通过控制器建立二者之间的关联。

Eclipse的GEF框架就是基于MVC模式进行设计的，如下图所示：



用户的输入事件被转换成请求（Request），并通过Command来修改模型，模型改变后发出事件通知，Controller收到事件通知进行视图的刷新。

在Studio中，模型都是基于EMF自动生成的，EMF所生成的模型代码具有事件通知功能，如：为图元设置名称的方法如下：

|  |
| --- |
| public void setName(String newName) {  String oldName = name;  name = newName;  if (eNotificationRequired())  eNotify(new ENotificationImpl(this,  Notification.SET, //事件类型  CorePackage.ELEMENT\_\_NAME, //对应的feature  oldName, name));  } |

设置名称时，通过eNotify方法来发出事件通知。

MVC中的视图（View）在GEF中有两种实现方式，一种是基于图形的，通过Draw2D中的IFigure进行展现；另一种是树形展现结构的，利用树形结构来展现模型。

MVC中的控制器（Controller）对应到GEF中是EditPart，其中GraphicalEditPart通过创建一个图形（IFigure）来展现模型，该类的主要方法如下：

|  |
| --- |
| Object getModel();  List getChildren();  **void** activate();  **void** deactivate();  Command getCommand(Request request);  EditPolicy getEditPolicy(Object key);  IFigure getFigure(); |

其中，getModel方法用来获取EditPart所关联的模型；getChildren用来获取模型的子节点列表；activate方法将在图元被激活时调用；deactivate方法将在图元变得不可用时调用；getCommand方法将根据一个请求从所注册的EditPolicy中生成一个Command，并通过Command来对模型进行操作；getEditPolicy方法获取图元的编辑策略；getFigure方法获取图元要展现的图形。

在Studio的GEF框架中，图元（NodeElement）对应的EditPart实现类是NodeEditPartEx，该类实现了EMF框架中的事件监听接口（org.eclipse.emf.common.notify.Adapter），并通过EditPart#activate方法将监听器注册到它所关联的模型中，如下所示：

|  |
| --- |
| **public** **void** activate() {  **if** (!isActive()) {  hookIntoModel(getCastedModel());  **super**.activate();  }  }    **protected** **void** hookIntoModel(EObject model) {  **if**(model != **null**) {  model.eAdapters().add(**this**);//注册到模型变化监听器列表中  }  } |

通过deactivate方法从模型中移除事件监听器，如下所示：

|  |
| --- |
| **public** **void** deactivate() {  **if** (isActive()) {  unhookFromModel(getCastedModel());  **super**.deactivate();  }  }  **protected** **void** unhookFromModel(EObject model) {  **if**(model != **null**) {  model.eAdapters().remove(**this**);//移除监听器  }  } |

NodeEditPartEx实现了Adapter事件监听器的notifyChanged方法，当模型发生变化时，该方法被调用，进行视图的刷新，如下所示：

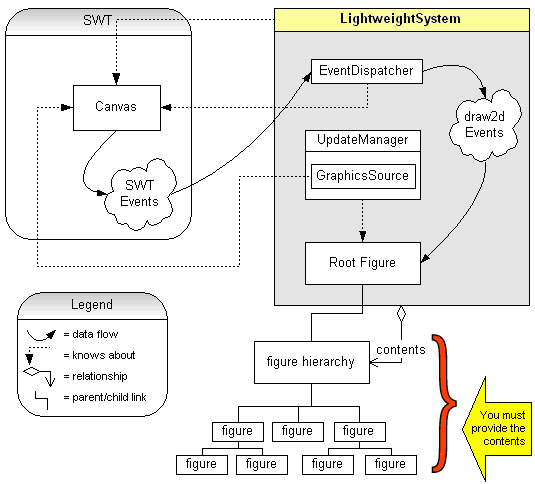
|  |
| --- |
| **public** **void** notifyChanged(Notification notification) {  **int** type = notification.getEventType();  **int** featureId = notification.getFeatureID(CorePackage.**class**);  **switch**(type) {  **case** Notification.*SET*:  **switch**(featureId) {  **case** CorePackage.*LOCATION\_\_X*:  **case** CorePackage.*LOCATION\_\_Y*:  **case** CorePackage.*NODE\_ELEMENT\_\_LOCATION*:  **case** CorePackage.*NODE\_ELEMENT\_\_SIZE*:  refreshVisuals();//刷新视图  **break**;  }  ……  }  } |

对于GEF的详细信息请阅读GEF帮助文档中《GEF Programmer’s Guide》。

#### Draw2D

（Draw2D简介，相关的类说明）

Draw2D是GEF框架的一部分，它提供了在SWT的Canvas上进行图形绘制和布局的能力，GEF通过Draw2D实现了GEF中的视图（View）。Draw2D的原理图如下所示：



LightWeightSystem是Draw2D的核心部件，它是连接SWT和Draw2D的桥梁。它包含三个主要组成部分：RootFigure是LightWeightSystem中所有图形的根，即：其它的图形都是直接或间接的放在RootFigure里的；EventDispatcher是事件分派器，它把画布（Canvas）上产生的各种事件分派给RootFigure，并通过RootFigure分派给最终的图形；UpdateManager用来重绘图形，当Canvas被要求重绘时，LightWeightSystem会调用它的performUpdate方法。

Draw2D中的图形（figure）都要实现IFigure（org.eclipse.draw2d.IFigure）接口。IFigure通过复合模式实现了图形的嵌套，即：一个图形可以包含多个子图形，从而实现对复杂图形的支持；每个图形都有位置和大小，这通过图形的bounds属性进行描述；IFigure通过paint方法实现对图形的绘制，paintFigure用来绘制主图形，paintClientArea用来绘制图形所包含的子图形，paintBorder用来绘制图形的边框。Draw2D提供了很多缺省图形，主要有三类：形状（Shape，如：矩形、圆等）、控件（Widget，如：标签、按钮、滚动条等）、图层（Layer）。

图形可以包含子图形，那么子图形在主图形中是如何被放置和排列的呢？这是通过主图形的布局管理器（LayoutManager）来实现的。每个LayoutManager都包含某种算法，该算法将根据与每个子图形关联的约束（Constraint）对象，计算出子图形最终的位置和大小。Draw2D内置了几种常见的布局管理器，如：流式布局（FlowLayout）、XY布局（XYLayout）等。

Draw2D中还有一个重要的元素：连线（Connection），它也是一种图形，用来在两个图形之间建立连线。连线是由多个点组成的，绘制时通过在点之间创建线段来组成最终的连线。与Connection相关的几个重要概念有：锚点（Anchor）、路由（Router）和Locator，其中Anchor用来控制连线端点在图形上的位置，最常见的是ChopBoxAnchor，它以图形的中心点作为假想连接点，然后计算这条假想连线与图形边缘的交汇点作为实际的锚点；Router负责控制连线的外观和操作方式，没有设置路由时，会采用两点间的直线连接，曼哈顿路由（ManhattanConnectionRouter）是一种横平竖直的连接方式，仿佛曼哈顿的街道一样；Locator的作用是定位图形。

### 图形化BPMN编辑器

由于Studio对GEF的封装，所以我们在Studio中实现一个图形化的编辑器还是相对容易的。AbstractGraphicalEditor是Studio图形化编辑器的基类，Studio中的许多图形化编辑器，如：逻辑流编辑器、页面流编辑器、数据集编辑器等，都扩展自该类。

同样，我们的图形化BPMN编辑器也将扩展AbstractGraphicalEditor。在com.primeton.studio.bpmn.ui插件中，创建类：com.primeton.studio.bpmn.ui.editor.BPMNEditor，如下所示：

|  |
| --- |
| **public** **class** BPMNEditor **extends** AbstractGraphicalEditor {  @Override  **protected** AbstractPropertyMenuManager createMenuManager() {  **return** **null**;  }  @Override  **protected** String getPaletteHelpContextId() {  **return** **null**;  }  @Override  **public** String getPluginId() {  **return** BPMNUiPlugin.*PLUGIN\_ID*;  }  @Override  **protected** ContextMenuProvider createContextMenuProvider() {  **return** **new** AbstractContextMenuProvider(){};  }  } |

其中，getPluginId方法用来获取编辑器所在的插件Id，Studio的GEF框架会利用这个Id来获取插件所注册的元素，如：选用板元素、EditPartFactory等。createContextMenuProvider方法用来创建编辑器的上下文菜单（右键菜单）。为了帮助大家更好的理解图形化编辑器的一些原理，下面我们分别介绍与图形化编辑器密切相关的几个主题。

#### 创建Graphical Viewer

基于GEF的图形化编辑器都会关联一个GraphicalViewer，顾名思义，GraphicalViewer是一个图形查看器，用来显示图形元素。它扩展自EditPartViewer，如下所示：



每个GraphicalViewer都会关联一个EditPartFactory，用来创建模型对应的EditPart；EditDomain是GEF应用程序的状态收集器，通过EditDomain我们可以获得编辑器的命令栈（Command Stack）和活动工具（Active Tool）等与编辑器当前状态相关的信息；ScrollingGraphicalViewer是一种常见的GraphicalViewer实现，这个Viewer带有滚动条；GraphicalViewer通过关联的FigureCanvas来实际显示图形。

Studio的AbstractGraphicalEditor，通过超类GraphicalEditor的createGraphicalViewer方法来创建GraphicalViewer，如下所示：

|  |
| --- |
| **protected** **void** createGraphicalViewer(Composite parent) {  // 创建GraphicalViewer  GraphicalViewer viewer = **new** ScrollingGraphicalViewer();  // 创建FigureCanvas  viewer.createControl(parent);  setGraphicalViewer(viewer);  // 配置GraphicalViewer  configureGraphicalViewer();  hookGraphicalViewer();  // 初始化GraphicalViewer  initializeGraphicalViewer();  } |

其中的configureGraphicalViewer和initializeGraphicalViewer是两个重要的方法，用来对视图进行配置并初始化这个视图，Studio的AbstractGraphicalEditor重载了这两个方法。

AbstractGraphicalEditor通过重载configureGraphicalViewer来实现：

* 为GraphicalViewer创建并设置Root EditPart；
* 通过扩展机制获取注册的EditPartFactory，并设置到GraphicalViewer中；
* 创建并设置GraphicalViewer的上下文菜单；

代码如下所示：

|  |
| --- |
| GraphicalViewer viewer = …;  //创建Root EditPart  ScalableFreeformRootEditPartEx root = getScalableFreeformRootEditPartEx();  viewer.setRootEditPart(root);  ……  //根据插件id获得对应的EditPartFactory  DiagramEditPartFactory factory =  ExtensionsReader.*getInstance*().getEditPartFactory(getPluginId());  factory.setEditor(**this**);  viewer.setEditPartFactory(factory);  ……  **this**.fContextMenuProvider = (AbstractContextMenuProvider) createContextMenuProvider();  **this**.fContextMenuProvider.iniContextMenuProvider(viewer, getActionRegistry());  //设置上下文菜单  viewer.setContextMenu(**this**.fContextMenuProvider);  …… |

关于Root EditPart和EditPartFactory将在后续章节详细描述。

#### 创建RootEditPart

RootEditPart是一种特殊的EditPart，是EditPartViewer的根元素，它是EditPartViewer和EditPartViewer要展现的实际内容之间的桥梁，它不与任何的模型元素相对应。RootEditPart的接口如下所示：

|  |
| --- |
| EditPart getContents();  /\*\*  \* Returns the root's EditPartViewer.  \*/  EditPartViewer getViewer();  /\*\*  \* Sets the <i>contents</i> EditPart. A RootEditPart only has a single child, called its <i>contents</i>.  \*/  **void** setContents(EditPart editpart);  /\*\*  \* Sets the root's EditPartViewer.  \*/  **void** setViewer(EditPartViewer viewer); |

它的主要方法是setContents，用来设置GraphicalViewer真正要展现的内容（contents），RootEditPart只有唯一的一个孩子。

常见的RootEditPart实现有：

* ScalableRootEditPart
* ScalableFreeformRootEditPart

Studio的图形化编辑器使用的是ScalableFreeformRootEditPart。

#### 实现Content EditPart

Content EditPart通常是模型的根元素所对应的EditPart，它被放置在RootEditPart中，作为RootEditPart的唯一孩子节点。在我们的BPMN模型中，Process是根元素，它是我们的BPMN编辑器的content。

图形化编辑器（AbstractGraphicalEditor）在初始化Graphical Viewer（initializeGraphicalViewer）时会设置viewer要展现的内容（content），如下所示：

|  |
| --- |
| **protected** **void** initializeGraphicalViewer() {  **super**.initializeGraphicalViewer();  GraphicalViewer viewer = getGraphicalViewer();  viewer.setContents(getModel());  ……  } |

编辑器的Viewer会关联一个EditPartFactory，用来获取模型所对应的EditPart，为此，我们在com.primeton.studio.bpmn.ui插件中，新建com.primeton.studio.bpmn.ui.editor.BPMNEditPartFactory类，该类扩展自DiagramEditPartFactory，如下所示：

|  |
| --- |
| **public** **class** BPMNEditPartFactory **extends** DiagramEditPartFactory {  } |

我们暂且不对该类做任何扩展。BPMNEditPartFacotry在编辑器的configureGraphicalViewer方法中被设置到Graphical Viewer中，根据AbstractGraphicalEditor的实现，我们需要通过com.primeton.studio.gef.ui.editPartFactory扩展点为图形化编辑器注册EditPart工厂类，如下所示：



GEF框架在获取Process对象的EditPart时，会调用BPMNEditPartFactory的createEditPart方法来创建Process对应的EditPart，由于我们的BPMN模型扩展自Studio图形化编辑器的底层模型，所以Process对象所对应的EditPart是DiagramEditPart，请查看DiagramEditPart类获得更详细的信息。

#### 实现各个子EditPart

下面我们分别为BPMN模型的子元素创建EditPart。Studio的GEF框架已经为节点元素提供了EditPart的默认实现：NodeEditPartEx，BPMN图元的EditPart将扩展自该类。这些EditPart都位于com.primeton.studio.bpmn.ui.editor.editpart包下。

1. **创建StartEventEditPart**

|  |
| --- |
| **public** **class** StartEventEditPart **extends** NodeEditPartEx {  **public** StartEventEditPart(AbstractGraphicalEditor editor) {  **super**(editor);  }  @Override  **protected** IFigure createFigure() {  Ellipse figure = **new** Ellipse();  figure.setLineWidth(1);  **return** figure;  }  } |

1. **创建EndEventEditPart**

|  |
| --- |
| **public** **class** EndEventEditPart **extends** NodeEditPartEx {  **public** EndEventEditPart(AbstractGraphicalEditor editor) {  **super**(editor);  }    @Override  **protected** IFigure createFigure() {  Ellipse figure = **new** Ellipse();  figure.setLineWidth(3);  **return** figure;  }  } |

1. **创建TaskEditPart**

|  |
| --- |
| **public** **class** TaskEditPart **extends** NodeEditPartEx {  **public** TaskEditPart(AbstractGraphicalEditor editor) {  **super**(editor);  }  @Override  **protected** IFigure createFigure() {  **return** **new** TaskFigure();  }  } |

TaskFigure的实现如下：

|  |
| --- |
| **public** **class** TaskFigure **extends** Figure {  @Override  **protected** **void** paintFigure(Graphics graphics) {  Rectangle rect = **this**.getBounds().getCopy();  graphics.drawRoundRectangle(rect.crop(**new** Insets(1,1,1,1)), 5, 5);  }  } |

1. **创建GatewayEditPart**

|  |
| --- |
| **public** **class** GatewayEditPart **extends** NodeEditPartEx {  **public** GatewayEditPart(AbstractGraphicalEditor editor) {  **super**(editor);  }  @Override  **protected** IFigure createFigure() {  **return** **new** GatewayFigure();  }  } |

GatewayFigure的实现如下：

|  |
| --- |
| **public** **class** GatewayFigure **extends** Figure {    **private** **static** **final** PointList *points* = **new** PointList(**new** **int**[] {  15, 0,  30, 15,  15, 30,  0, 15 });  @Override  **protected** **void** paintFigure(Graphics graphics) {  Rectangle bounds = **this**.getBounds();  graphics.translate(bounds.getLocation());  graphics.drawPolygon(*points*);  }  } |

修改BPMNEditPartFactory，并重载createNodeEditPart方法，如下所示：

|  |
| --- |
| @Override  **public** NodeEditPart createNodeEditPart(AbstractGraphicalEditor editor,Element element){  **if**(element **instanceof** StartEvent) {  **return** **new** StartEventEditPart(editor);  } **else** **if**(element **instanceof** EndEvent) {  **return** **new** EndEventEditPart(editor);  } **else** **if**(element **instanceof** Task) {  **return** **new** TaskEditPart(editor);  } **else** **if**(element **instanceof** Gateway) {  **return** **new** GatewayEditPart(editor);  }    **return** **super**.createNodeEditPart(editor, element);  } |

#### 添加节点间的连线

连线是图形化编辑器中较复杂的一类对象，它在GEF框架中对应的编辑对象是ConnectionEditPart。一条连线由两部分确定：源（Source）节点和目标（Target）节点。

从连线本身的图形角度来看，一条连线的本质是由多个点组成。这些点又分为三大类：起点、中间点和终点，在GEF框架中，起点和终点由ConnectionAnchor负责计算，具体是由ConnectionAnchor的getLocation方法负责计算。中间点由与连线关联的ConnectionRouter负责计算。

在添加节点的EditPart时，会调用节点的refresh方法来为连线设置源连接线和目标连接线，AbstractGraphicalEditPart的refresh方法如下所示：

|  |
| --- |
| **public** **void** refresh() {  **super**.refresh();  refreshSourceConnections();  refreshTargetConnections();  } |

refreshSourceConnections方法负责计算以该节点作为起点的连线，该方法的主要步骤如下：

* 调用节点的getModelSourceConnections方法来获得连线对应的模型

例如，我们的NodeEditPartEx的getModelSourceConnections方法，会通过调用节点对象（NodeElement）的getSourceConnections方法来获取以该节点作为起点的连线集合。

* 创建连线对应的EditPart

获得连线模型后，通过EditPartFactory来创建连线模型对应的EditPart，在Studio的GEF框架中，一般的Connection对象对应的EditPart是ConnectionEditPart。

* 设置连线的源节点

通过调用ConnectionEditPart的setSource方法来设置连线的源节点，这会导致ConnectionEditPart的刷新，它的refresh方法如下：

|  |
| --- |
| **public** **void** refresh() {  refreshSourceAnchor();  refreshTargetAnchor();  **super**.refresh();  } |

从上面的方法可以看出，连线的EditPart会通过调用refreshSourceAnchor和refreshTargetAnchor来设置节点的源锚点和目标锚点。

refreshTargetConnections的过程与此类似。

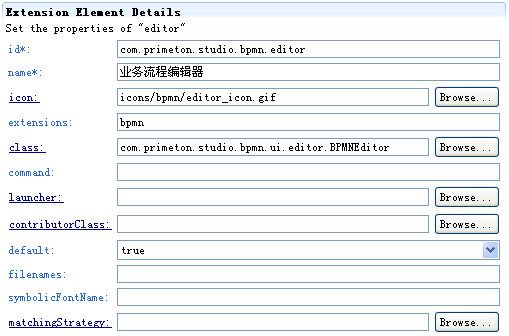
BPMN模型中的SequenceFlow扩展自Studio图形化编辑器底层模型的Connection对象，同时BPMNEditPartFactory扩展自DiagramEditPartFactory，而后者已经为我们提供了Connection对象的EditPart创建方法，我们的SequenceFlow可以直接ConnectionEditPart。

#### 注册

至此，BPMN图形化编辑器的各个部分都已经实现了，现在我们只需要通过Studio的GEF框架提供的扩展点进行相关的注册就可以为BPMN模型提供图形化编辑器了。

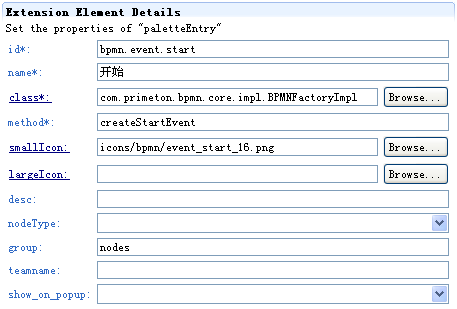
1. **注册编辑器**

通过Eclipse提供的org.eclipse.ui.editors扩展点，可以为资源文件注册一个新的编辑器，这里我们为BPMN资源注册一个新的编辑器，配置如下：



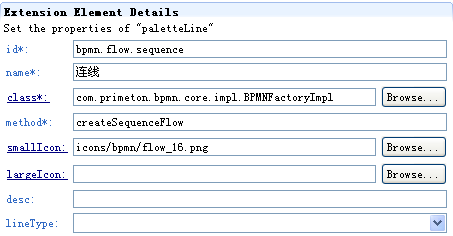
1. **注册选项板元素**

Studio的GEF提供了com.primeton.studio.gef.ui.palette扩展点，通过它我们可以为图形化编辑器注册选项板中的元素。节点对应的palette类型是paletteEntry，连线对应的palette类型是paletteLine。为开始节点的选项板元素配置的信息如下所示：



请参照上述配置，添加“结束”、“任务”、“关口”等选项板元素；

连线的选项板元素配置如下：

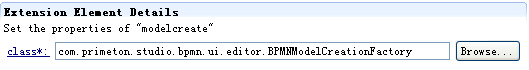


BPMN图形化编辑器在初始化palette viewer时，会通过palette扩展点来初始化选项板。

选项板的Creation 类型的Tool会关联一个CreationFactory实例，当点击选项板中的某个元素时，会调用该实例的getNewObject方法来创建一个模型实例。为此，我们创建了BPMNModelCreationFactory类，如下所示：

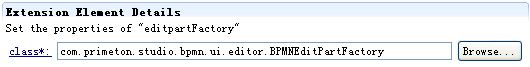
|  |
| --- |
| **public** **class** BPMNModelCreationFactory **extends** ModelCreationFactory {  **public** BPMNModelCreationFactory() {  }  **public** BPMNModelCreationFactory(ModelCreationFactory factory) {  **super**(factory);  }  @Override  **public** Object clone() **throws** CloneNotSupportedException {  **return** **new** BPMNModelCreationFactory(**this**);  }  } |

通过com.primeton.studio.gef.ui.modelcreation扩展点注册BPMNModelCreationFactory，如下所示：

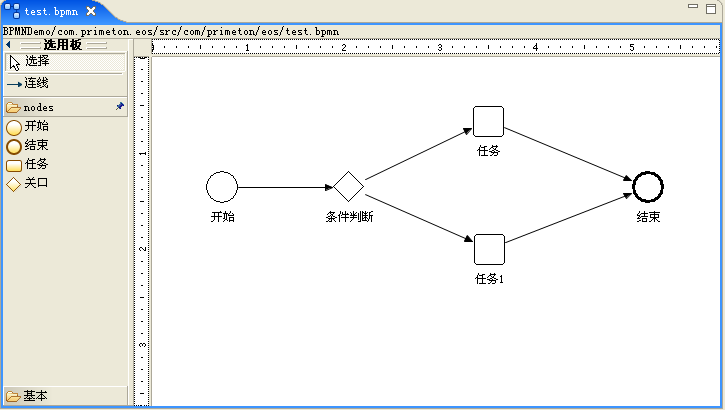


1. **注册EditPart创建工厂**

通过com.primeton.studio.gef.ui.editPartFactory扩展点为BPMN编辑器注册EditPart创建工厂，如下所示：



运行Studio，资源管理器中双击BPMN资源文件，系统将会使用我们注册的BPMN图形化编辑器来打开BPMN资源文件，如下所示：



#### 事件处理

事件处理也是GEF框架中不得不提的一个话题，事件（鼠标、键盘等）如何通过画布（Canvas）传递到具体的图形？如何得到响应？本节我们将解答这些问题。

GEF中，事件处理的一般步骤描述如下：

1. **EditDomain**

画布（Canvas）产生的事件被分发到图形视图（GraphicalViewer）所关联的EditDomain中，EditDomain中定义了一系列的事件处理方法：mouseXXX、keyXXX等用来处理鼠标和键盘产生的事件。EditDomain再将事件传递到当前编辑器的活动工具（Tool）中。

1. **Tool**

Tool（org.eclipse.gef.Tool）用来处理具体的事件，选项板中的每个元素都关联一个具体的Tool，如：图元选择工具对应到SelectionTool，图元创建工具对应到CreationTool。每种Tool都对事件做出不同的响应，从而达到细分事件类型的目的。EditDomain有个activeTool属性，用来保存当前的活动工具，它是状态（State）模式的一种实现，当我们点击选项板中的一个元素时会相应的更新activeTool属性。

Tool负责将事件转换成一种处理请求（Request），如：当我们移动一个图元时，SelectionTool会通过关联的DragEditPartsTracker来创建一个ChangeBoundsRequest，如下所示：

|  |
| --- |
| **protected** Request createTargetRequest() {  **if** (isCloneActive())  **return** **new** ChangeBoundsRequest(*REQ\_CLONE*);  **else**  **return** **new** ChangeBoundsRequest(*REQ\_MOVE*);  } |

1. **EditPart**

请求（Request）创建好以后，会调用目标EditPart的getCommand(Request)方法来获取能够处理该请求的Command，如下所示（AbstractEditPart#getCommand）：

|  |
| --- |
| **public** Command getCommand(Request request) {  Command command = **null**;  EditPolicyIterator i = getEditPolicyIterator();  **while** (i.hasNext()) {  **if** (command != **null**)  command = command.chain(i.next().  getCommand(request));  **else**  command = i.next().  getCommand(request);  }  **return** command;  } |

从上面的代码可以看出，EditPart会委托它的EditPolicy来产生具体的Command。

1. **EditPolicy**

EditPolicy的主要职责是处理Request，产生对应的Command。每个EditPolicy负责处理特定类型的Request，一个EditPart可以安装多个EditPolicy，NodeEditPartEx的createEditPolicies方法如下：

|  |
| --- |
| **protected** **void** createEditPolicies() {  // allow removal of the associated model element  installEditPolicy(EditPolicy.*COMPONENT\_ROLE*, **new** NodeComponentEditPolicy(editor.getPluginId()));  installEditPolicy(TooltipEditPolicy.*TOOLTIP\_ROLE*,**new** TooltipEditPolicy());  // allow the creation of connections and  // and the reconnection of connections between node instances  installEditPolicy(EditPolicy.*GRAPHICAL\_NODE\_ROLE*, **new** GefGraphicalNodeEditPolicy(editor));  } |

其中，NodeComponentEditPolicy用来处理图元的删除；TooltipEditPolicy用来处理鼠标悬停在图元上时显示一个提示框。

1. **Command**

鼠标或键盘产生的事件如果涉及到对模型的修改，一般都会通过EditPolicy来产生Command（org.eclipse.gef.commands.Command），Command的主要方法如下：

* + execute();

执行模型修改动作；

* + undo();

撤消所做的更改；

* + undo();

重做所做的更改；

每个图形化编辑器都会有一个命令执行栈（CommandStack），它负责Command的执行，并以栈的形式记录了执行过的命令以方便回退。

## 附录

1. 编码规范

<http://product.primeton.com/download/attachments/2818061/%E7%BC%96%E7%A0%81%E8%A7%84%E8%8C%83.doc>

2．Ui依赖插件

图片包含 屏幕截图

描述已自动生成