

2

正在创建第一个Xtext语言：

在本章中，我们将使用Xtext开发一个DSL，并学习Xtext语法语言是如何工作的。当我们修改DSL的语法时，我们将看到使用Xtext编程的典型开发工作流程。本章还将简要介绍EMF(Eclipse建模框架)，该框架由Xtext依赖于构建程序的AST（抽象语法树）。

针对实体的DSL

现在我们将实现一个简单的DSL来建模实体，它可以看作是简单的Java类；每个实体都可以有一个超级类型实体(您可以将它视为Java超类)和一些属性(类似于Java字段)。此示例是Xtext文档中找到的域模型示例的变体。

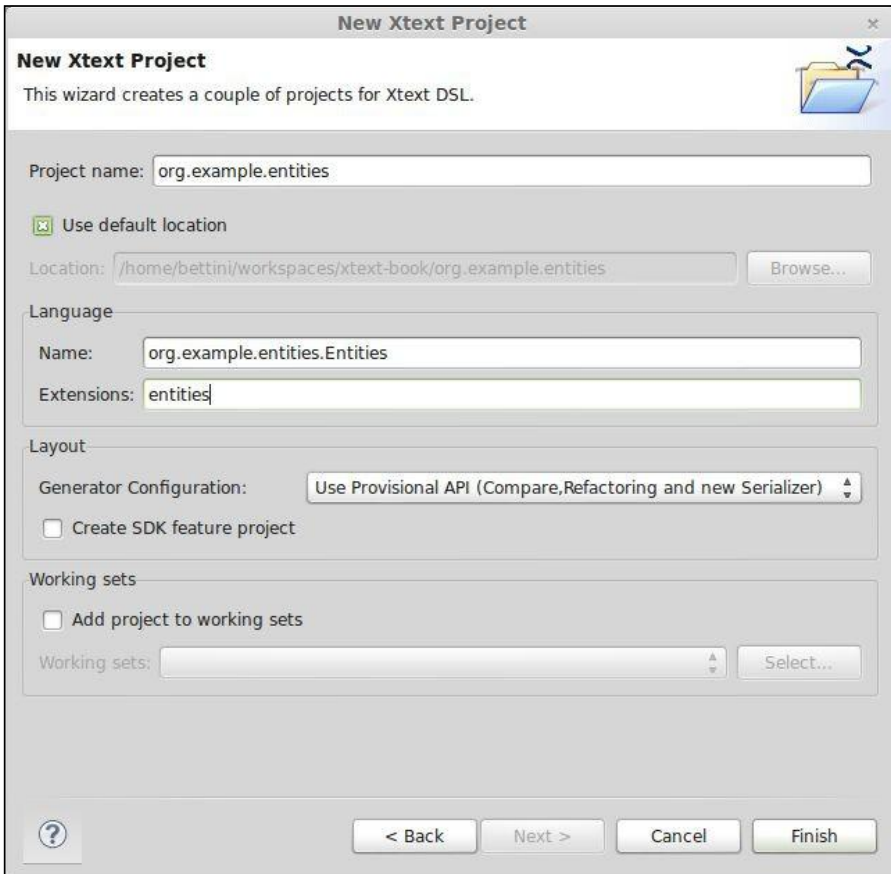
正在创建该项目

首先，我们将使用Xtext项目向导来为DSL创建项目（我们已经在第1章实施DSL的结尾进行了实验）。

1. 启动月食并导航到文件|New|项目。.....在该对话框中进行导航转到Xtext类别，并选择Xtext项目。
2. 在下一个对话框中，您应指定以下名称：
 - 项目名称：org.example.entities
 - 姓名：org.example.entities。各实体
 - **扩展范围：各实体**
 - 取消选中选项创建SDK功能项目(我们将使用
仅在第11章“构建和发布”中创建SDK功能项目)

该向导将创建三个项目，并将打开文件Entities.xtext是语法的定义。

该向导的主对话框显示在如下屏幕截图中：



Xtext项目

Xtext向导生成三个项目，通常，在Xtext中实现的每个DSL都有这三个项目（根据向导中指定的项目名称）。在我们的例子中，我们有：

- org.example.entities是包含语法的主要项目定义和独立于UI的所有运行时组件
- org.example.entities.tests包含单元测试

- `org.example.entities.ui` 包含与UI相关的组件（月蚀编辑器和与用户界面相关的功能）

我们将在第6章中的自定义和单元测试中描述UI功能
第七章，测试。

修改语法

您从第1章实现DSL中记得，*Xtext*由生成默认语法。在本节中，您将了解此生成的语法所包含的内容，我们将修改它以包含实体DSL的语法。生成的语法外观如下：

语法`org.example.entities`。与`org.eclipse.xtext.common`相关的实体。
接线端子

生成实体 “`http://www.example.org/entities/Entities`” 产品型号：
问候语+=问候语*;

问候语：

“你好”，名字是=ID”！ ‘;

第一行声明了语言的名称（和语法）的名称，它也对应于。`xtext`文件的完全限定名（该文件称为`Entities.xtext`，它在包`org.example.entities`中）。

语法声明还声明，它重用了语法终端，它为引用的字符串、数字和注释定义了语法规则，因此在我们的语言中，我们不必定义这些规则。语法终端是*Xtext*库的一部分；在第12章*Xbase*中，我们将看到另一个*Xtext*库语法的示例（*Xbase*语法）。

生成声明为EMF定义了一些生成规则，我们将进行讨论稍后。

在前两个声明之后，将指定语法的实际规则。有关这些规则的完整语法，请参考官方的*Xtext*文档（<http://www.eclipse.org/Xtext/documentation.html>）。目前，我们将编写的所有规则都将有一个名称、一个冒号、该规则所接受的实际语法形式，并以分号终止。

现在我们修改语法如下:

语法org.example.entities。具有以下的实体:

org.eclipse.xtext.common.接线端子

生成实体 “http://www.example.org/entities/Entities” 模型: 实体+=实体*;

工程建设单位:

“实体” 名称=ID (“扩展” 超类型=[实体]) ? ‘{’ 属性+=属性*
,’

;

属性:

键入=[实体]数组? = (“[] ”) ? 名称=ID “; ” ;

每个语法中的第一个规则定义了解析器的起始位置和的类型
DSL模型的根元素, 即抽象语法树 (AST)。

在本示例中, 我们声明实体DSL程序是实体元素的集合。此集合存储在模型对象中, 特别是在一个称为实体的特性中 (稍后将看到, 该集合作为列表实现)。操作符+=暗示了它是一个集合的事实。星形运算符*表示元素的数量 (在本例中, 实体) 是任意的; 特别是, 它可以是任意数字 ≥ 0 。因此, 程序也可以为空, 并且不包含任何实体。



如果我们希望我们的程序至少包含一个实体, 我们应该使用
操作符+而不是*。

实体元素的形状以其自己的规则来表示:

工程建设单位:

“实体” 名称=ID (“扩展” 超类型=[实体]) ? ‘{’ 属性+=属性*
,’

;

首先, 字符串文字 (在Xtext中可以用单引号或双引号来表示) 定义了DSL的关键字。在这个规则中, 我们三个关键字, 即 “实体”、 “扩展”、 “{” 和 “}”。

因此，有效的实体声明句以关键字“实体”开头，后面跟着ID；我们的语法中没有定义ID的规则，因为这是我们从语法终端继承的规则之一。如果您想知道ID是如何定义的，您可以点击+点击Xtext编辑器中的ID，这将带到语法终端，在那里您可以看到一个ID以可选的“^”字符开头，后面是字母(“a”...“z” | “A”..“Z”)、“\$”字符或下划线“_”，后面跟着任意数量的字母、“\$”字符、下划线和数字(“0”..“9”)：

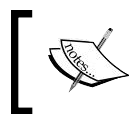
```

'^' ? ( 'a' .. 'z' | 'A' .. 'Z' | '$' | '_' )
( 'a' .. 'z' | 'A' .. 'Z' | '$' | '_' | '0' .. '9' ) * ;

```

如果与现有关键字发生冲突，可选的“^”字符用于转义标识符。解析的ID将被分配给解析的实体模型元素的特征名称。

()吗？运算符声明了一个可选的部分。因此，在ID之后，您可以写入关键字“扩展”和实体的名称。这说明了Xtext的一个强大特性，即交叉引用。事实上，在关键字“扩展”之后，我们想要的不仅是一个名称，而是一个现有实体的名称。这可以用方括号和我们想要引用的类型用语法来表示。Xtext将通过在程序中搜索具有给定名称的该类型的元素（在我们的例子中是一个实体）来自动解析交叉引用。如果它找不到它，它就会自动发出一个错误。请注意，为了使此机制正常工作，所引用的元素必须具有一个称为name的功能。正如我们将在以下一节中看到的，自动代码完成机制也将考虑交叉引用，因此提出了要参考的元素。



默认情况下，交叉引用及其分辨率是基于特征名称和ID确定的。此行为可以自定义，请参见第10章，范围定义。

然后，应使用卷括号“{ }”，可以在其中指定属性元素（记住+=和*的含义）；这些属性元素将存储在相应实体对象的属性特性中。

属性：

键入=[实体]数组? = (“[] ”) ? 名称=ID “; ” ;

属性规则需要存储在类型功能中的实体名称（如前面所述，这将是交叉引用）以及属性的名称；属性还必须以“；”终止。请注意，在该类型之后，可以指定可选的“[]”；在这种情况下，属性的类型被视为数组类型，并且特征数组将为true。这个特性是布尔值，因为我们使用了
?=指定运算符，在该操作符之后，我们指定一个可选零件。

让我们试试这个编辑器吧

在第1章实现DSL的最后，我们看到了如何运行Xtext生成器；您应该遵循同样的步骤，但是不是右键单击。xtext文件并导航到作为|生成Xtext项目运行，我们右键单击。mwe2文件（在我们的示例中是GenerateEntities.mwe2），并导航到“作为|MWE2工作流运行”。（记得接受下载ANTLR生成器的请求，如第1章，实施DSL所述）。

在启动新的月蚀实例之前，必须确保启动配置具有足够的PermGen大小，否则将遇到“内存不足”错误。

您需要在启动时将以下值指定为VM参数

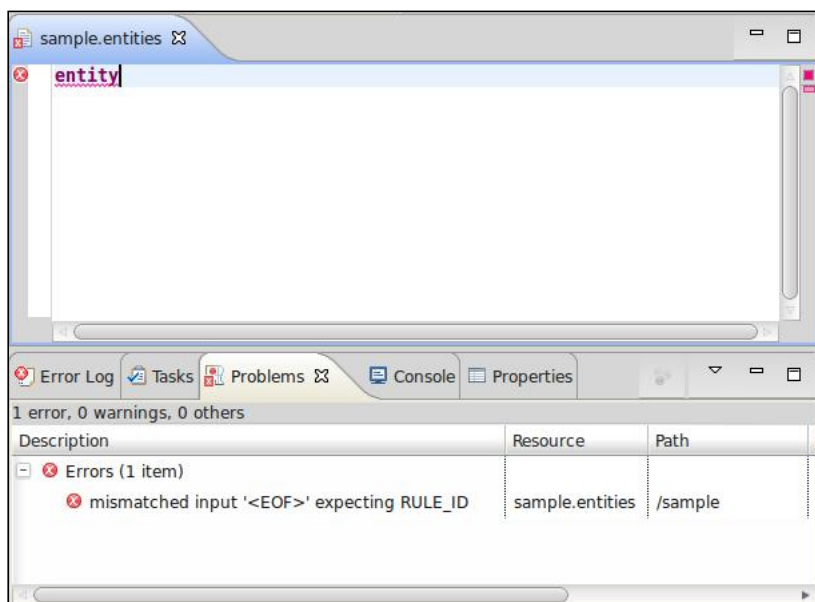
配置：-XX: MaxPermSize=256m。

您还可以简单地使用目录中org.example.entities项目的Xtext中为您创建的启动配置。启动；您可能无法看到该目录，因为默认情况下，工作台会隐藏以点开头的资源，因此请确保在工作区首选项中删除该过滤器。或者，您可以右键单击该项目，并导航到以|方式运行。运行配置。；在对话框中，可以在“月蚀应用程序”下看到“启动运行时日蚀”；选择它，然后单击“运行”。

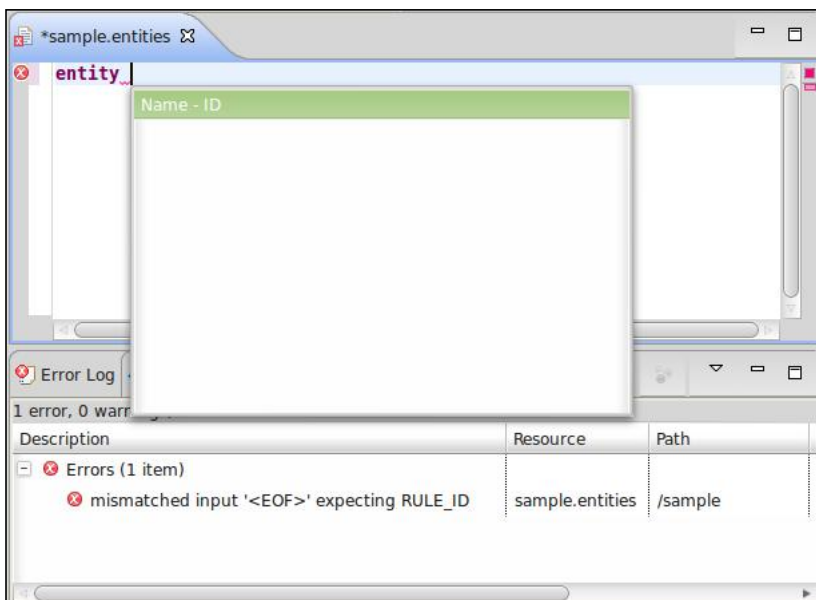
将运行一个新的Eclipse实例，并将出现一个新的工作台；在此实例中，可以使用实体DSL实现。所以让我们创建一个新的通用项目（例如，称为示例）。在此项目中，创建一个新文件；文件的名称并不重要，但文件扩展名必须是实体（请记住，这是我们在Xtext项目向导中选择的扩展名）。一旦创建了该文件，系统也将打开该文件，并且将被要求您将Xtext特性添加到项目中。您应该接受它，以使您的DSL编辑器在月蚀中正常工作。

编辑器为空，但没有错误，因为空程序是有效的实体程序（请记住是如何使用操作符*定义模型规则的）。如果您访问内容辅助程序（使用Ctrl+空格栏），您将不会得到任何建议，而是为您插入实体关键字。这是因为生成的内容辅助程序足够聪明，可以知道，在那个特定的程序上下文中，只有一件有效的事情要做：从关键字实体开始。

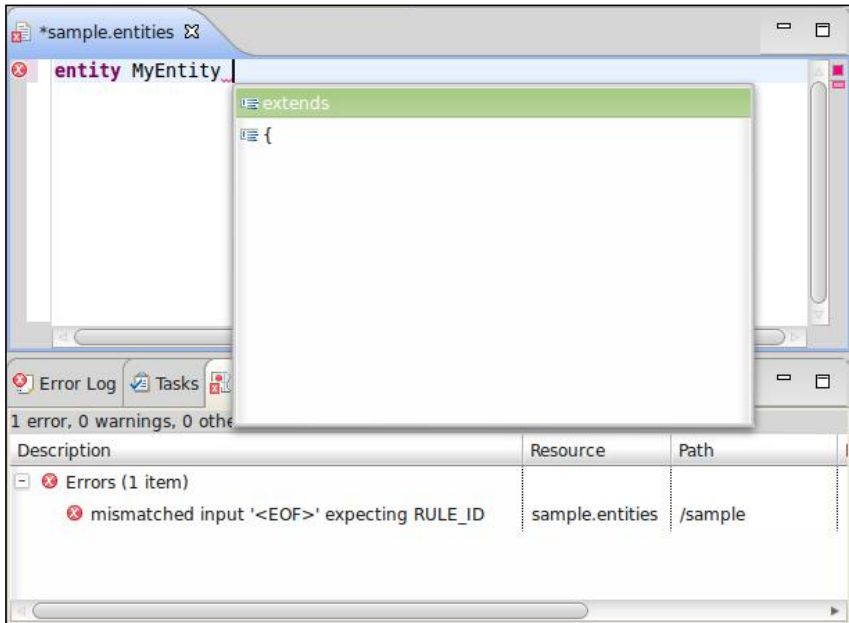
之后出现错误（请参考以下屏幕截图），因为实体定义仍然不完整（您可以看到语法错误告诉您需要使用标识符，而不是文件的末尾）：



如果再次访问内容辅助程序，您将得到需要使用标识符的提示（请参阅以下屏幕截图），因此让我们编写一个标识符：

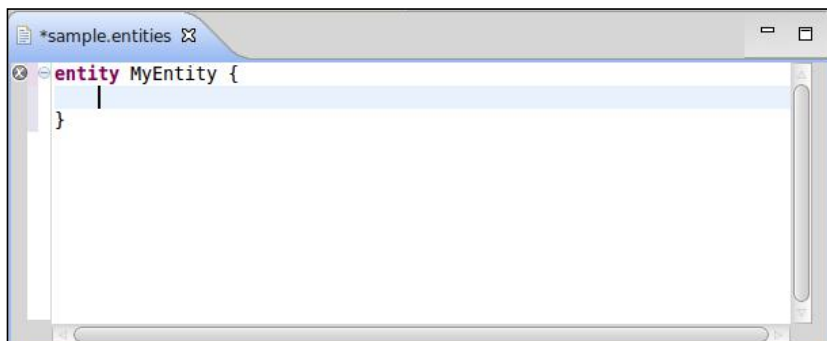


如果在标识符之后访问内容辅助，您将看到得到两个建议（请参考以下屏幕截图）。同样，生成的内容辅助程序知道，在该程序上下文中，您可以继续使用“扩展”规范或打开的卷括号。



如果选择打开的卷括号 {，您将在生成的编辑器中注意到一些有趣的内容（请参见以下屏幕截图）：

- 编辑器会自动插入相应的闭合卷括号
- 在括号之间插入换行线可正确执行缩进，并将光标移到正确位置
- 编辑器左侧的折叠将自动处理
- 错误标记变为灰色，表示当前程序中的问题已解决，但尚未保存（保存文件会使错误标记消失，“问题”视图变为空）



继续对编辑器进行试验：特别是在需要进行实体引用的上下文中（即在扩展关键字之后或声明属性时），您将看到内容辅助程序将为您提供当前程序中定义的所有实体元素。



我们不应该允许一个实体扩展自己；此外，层次结构应该是无循环的。但是，无法在语法中表达这些约束；这些问题必须通过实现自定义验证器（第4章，验证）或自定义范围界定机制（第10章，范围界定）来处理。

我们还想强调的是，所有这些远不易手动实现的功能，都是由Xtext从DSL的语法定义开始的Xtext生成的。

Xtext生成器

Xtext使用MWE2DSL来配置其工件的生成；默认生成的。mwe2文件已经具有良好的默认值，因此，目前我们不会对其进行修改。然而，有趣的是，通过调整这个文件，我们可以请求Xtext生成器生成对其他功能的支持，我们将在本书的后面看到。

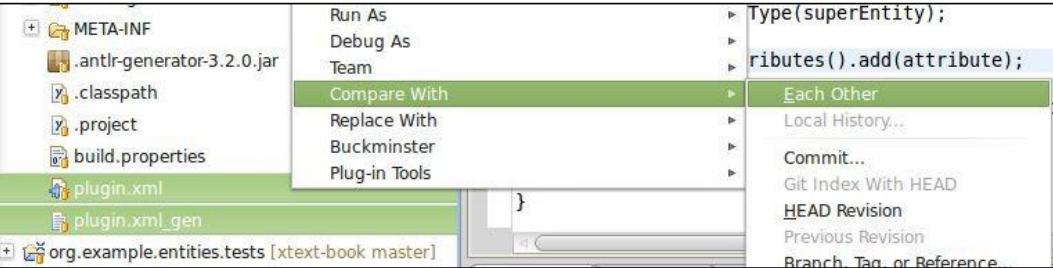
在MWE2 workflow执行期间，Xtext将生成与DSL的UI编辑器相关的项目，但最重要的是，它将从Xtext语法中推导出一个ANTLR规范，包括在解析时创建AST的所有操作。AST节点的类将使用EMF框架生成（如下一节所述）。

每次修改语法（.xtext文件）后，必须运行生成器。整个发电机基础设施依赖于发电缺口模式（1996年）。实际上，代码生成器很好，但是当您必须自定义生成的代码时：后代可能会覆盖您的自定义。生成间隙模式通过分离已生成（并且可以被覆盖）的代码和可以自定义（没有被覆盖的风险）的代码来解决这个问题。在Xtext中，生成的代码被放在源文件夹src-gen中（这适用于所有三个项目）；该源文件夹中的内容不应该被修改，因为在下一代中它将被覆盖。程序员可以安全地修改源文件夹src中的所有内容。

实际上，在第一代版本中，Xtext还将在源文件夹src中生成一些存根类，以帮助程序员找到一个起点。这些类永远不会重新生成，因此可以安全地进行编辑，而不会有被生成器覆盖的风险。一些存根类从Xtext库继承自默认类，而其他存根类继承自src-gen中的类。

src文件夹中生成的大多数存根类实际上都是Xtend类；Xtend编程语言将在下一章中介绍，因此，目前我们将不会看到这些存根类。

前面描述的生成策略有一个例外，它涉及到文件plugin.xml (在运行时和UI插件中)：进一步的Xtext生成将在项目的根目录中生成文件plugin.xml_gen。通过与plugin.xml进行比较，你来检查是否发生了变化。在这种情况下，您应该手动合并差异。这可以通过使用月食轻松地做到：选择两个文件，右键点击并导航到|相互比较。，如下屏幕截图所示：



一般来说，只有在修改.mwe2文件或使用新版本的Xtext（可以引入新特性）时才需要检查plugin.xml和plugin.xml_gen之间的差异。

最后，在运行MWE2工作流后，由于语法已更改，可以引入新的EMF类或修改一些现有的EMF类；因此，需要重新启动正在测试编辑器的其他Eclipse实例。

月蚀建模框架(EMF)

日蚀模型框架(EMF)（斯坦伯格等人，2008年），[http://www.](http://www.eclipse.org/modeling/emf)

[eclipse.org/modeling/emf](http://www.eclipse.org/modeling/emf)为基于结构化数据模型的构建工具和应用程序提供了代码生成设施。大多数以某种方式处理建模的Eclipse项目都是基于EMF的，因为它通过其建模机制简化了复杂软件应用程序的开发。模型规范（元模型）可以在XMI、XML模式、UML、理性玫瑰型或带注释的Java中进行描述。也可以使用在Xtext中实现的Xcore以编程方式指定元模型。通常，一个元模型是用Ecore格式定义的，它基本上是UML类图的一个子集的实现。



注意这个上下文中的元级别：Ecore模型是一个元模型，因为它是一个描述模型的模型。使用元模型，EMF将为模型生成一组Java类。如果：

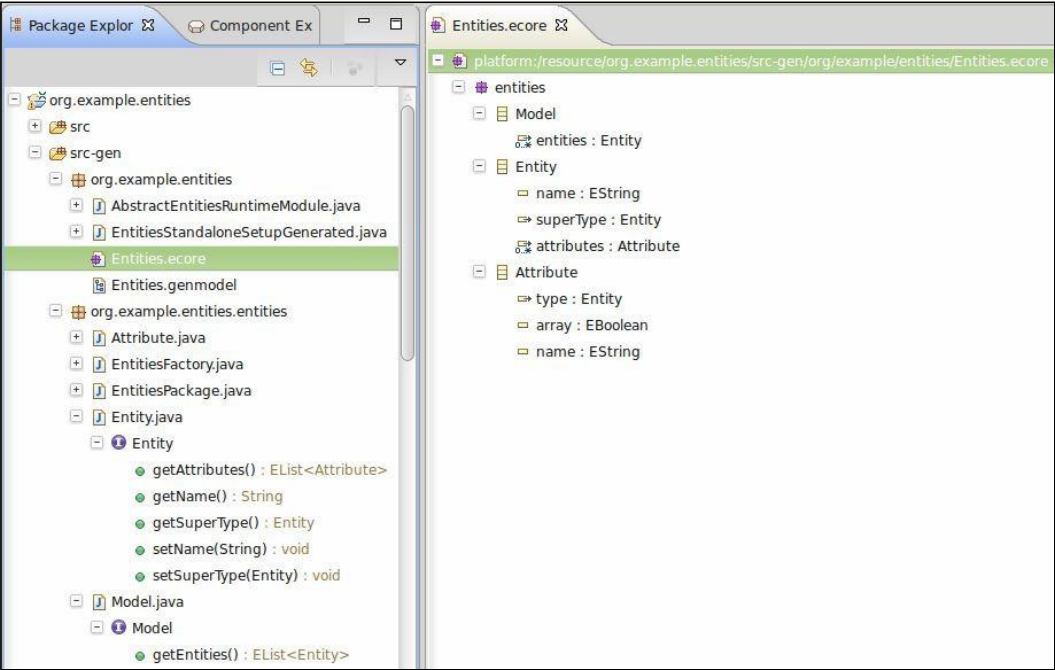
您不熟悉建模技术，您可以将元模型视为定义Java类的一种方法（即层次结构关系、字段、方法签名等）。EMF生成的所有Java类都是EObject的子类，可以看作是java.lang.Object的EMF。类似地，EClass对应于java.lang.Class来处理内省和反射机制。

Xtext依赖于EMF来创建AST抽象语法树，我们在第1章实现DSL中讨论了该树。根据语法规则，Xtext将自动推断出您的语言的EMF元模型。有关元模型推断的所有详细信息，您可以参考Xtext文档。目前，您可以考虑这个简化的场景：对于语法中的每个规则，将为规则中的每个特性使用一个字段（以及getter和设置器）创建一个EMF接口和类。例如，对于实体规则，我们将会有的相应的Java接口（以及相应的实现Java类）：

```
公共接口实体扩展了EObject{
    字符串getName();
    voidsetName(字符串值);
    实体获取超级类型();
    void设置超级类型(实体值); EList<属性
>get属性();
}
```

由于生成了这些Java工件，因此会将它们放置在相应的包中
在src-gen文件夹中。

您可以通过使用默认的EMF Ecore编辑器打开生成的元模型文件Entities.ecore来查看它。虽然您可能不知道EMF中元模型描述的细节信息，但要理解它的含义（参见下面的屏幕截图；在屏幕截图中，还可以看到EMF生成的一些扩展Java接口）：



元模型的推断和相应的EMF代码生成是由Xtext透明和自动地处理。但是，Xtext也可以使用自己维护的现有元模型，如文档中详细说明（我们不会在本书中使用此机制）。

由于DSL程序的模型是作为这些生成的EMF Java类的实例生成的，因此需要了解EMF的基本知识。一旦您必须对DSL执行额外的约束检查并生成代码，您就需要检查这个模型并遍历它。

使用生成的Java类很容易，因为它们遵循约定。特别是，EMF类的实例必须通过静态工厂创建（由EMF生成本身产生），因此无法使用任何构造函数；字段（即特性）可以完成获取器和设置器的初始化。EMF中的集合实现为EList接口（它是标准库列表的扩展）。只要考虑到这些概念，就很容易以编程方式操作程序的模型。例如，此Java代码片段会以编程方式创建一个对应于实体DSL程序的实体模型：

```
导入org.example.entities.entities。属性；导入
org.example.entities.entities。实体工厂；导入
org.example.entities.entities。各实体；
导入org.example.entities.entities。产品型号；
```

公共类实体

```
公共静态空白主（字符串参数）{实体工厂工厂=实体工厂。电力；
```

```
    实体超级实体=factory.createEntity();
    superEntity.setName（“我的超级实体”）；
```

```
    实体实体=factory.createEntity(); entity.setName
    （“我实体”）； entity.setSuperType（超级实
    体）；
```

```
    属性属性=factory.createAttribute();
    attribute.setName（“my属性”）；
    attribute.setArray(false); attribute.setType（超级实
    体）；
```

```
    entity.getAttributes().add（属性）；
```

```
    模型模型=factory.createModel();
    model.getEntities().add（超级实体）；
    model.getEntities().add（实体）；
```

```
}
```

```
}
```

EMF很容易学习，但与任何强大的工具一样，有很多东西需要学习来充分掌握它。正如之前所暗示的，它在日食世界中被广泛使用，因此你可以把它看作是一种投资。值得注意的是，新的Eclipse平台e4使用了基于EMF的应用程序模型，因此，如果您计划基于新的Eclipse4开发RCP应用程序，则必须处理EMF。

对DSL的改进

现在我们有了一个可工作的DSL，我们可以做一些改进和修改到语法中。

在对语法进行每次修改之后，正如我们在Xtext生成器部分中所说的，我们必须运行MWE2工作流，以便Xtext将生成新的ANTLR解析器和更新的EMF类。

首先，在尝试使用编辑器时，你可能会注意到，而

```
我的属性;
```

是我们DSL的有效句子（注意方括号之间的空格）

```
My实体[      ]我的属性;
```

产生一个语法错误。

这并不好，因为空间在DSL中不应该相关(尽管有像Python和Haskell这样的语言中空间确实相关)。

问题是在属性规则中，我们指定了[]，因此方括号之间不允许空格；我们可以如下修改该规则：

```
属性：键入=[实体]（数组？ = “[” ]”）？ 名称=ID “；”；
```

由于我们将两个方括号分成两个单独的标记，因此在编辑器中允许使用方括号之间的空格。实际上，空格会自动被丢弃（除非它们在令牌定义中明确表示）。

我们可以通过允许可选长度来进一步优化DSL中的阵列规格：

```
属性：
```

```
键入=[实体]（数组？ = “[” （长度为=INT）？ ‘’）？ 名称=ID “；”；
```

在我们的语法中没有定义INT的规则：我们从语法结尾继承了这个规则。正如您可以想象的，INT需要一个整数文字，因此我们的模型中的长度特征也将具有一个整数类型。由于长度功能是可选的（注意问号），以下两个属性定义都将是我们的DSL的有效句子：

```
我实体[]a;
```

```
我实体[10]b;
```

如果未指定长度，则长度功能将保留默认的整数值（0）。

正在处理的类型

我们定义属性类型概念的方式在概念上是不正确的，因为数组特性是属性的一部分，而它应该只涉及属性的类型。

然后，我们可以在一个单独的规则中分离属性类型的概念（这也将模型中产生一个新的EMF类）：

属性：

类型=属性类型名称-ID “；”；

属性类型：

实体=[实体](数组? = “[” (长度为=INT)? ‘]’)?；

如果运行MWE2工作流，您会注意到DSL编辑器中没有任何差异，但AST的元模型已更改。例如，考虑我们前面显示的这部分实体实例示例：

```
属性属性=factory.createAttribute();
attribute.setName(“my属性”);
attribute.setArray(false); attribute.setType(超级实
体);
```

这不再是有效的Java代码，必须进行如下更改：

```
属性属性=factory.createAttribute();
attribute.setName(“my属性”);
属性类型属性类型=factory.createAttributeType(); attributeType.setArray(false);
attributeType.setLength(10);
attributeType.setEntity(超级实体);
attribute.setType(属性类型);
```

作为对DSL的进一步增强，我们希望有一些基本类型：目前，只有实体可以用作类型。例如，让我们假设DSL提供了三种基本类型：字符串、int和布尔值。因此，一个基本类型是由其字面表示法来表示。相反，实体类型（到目前为止我们所使用的唯一类型概念）实际上是对现有实体的引用。此外，我们希望能够同时声明基本类型和实体类型的数组。

由于这些原因，数组特性仍然属于属性类型，但我们需要对元素类型进行抽象；因此，我们修改语法如下：

属性类型：

元素类型=元素类型(数组? = “[” (长度为=INT)? ‘’)? ;

元素类型：

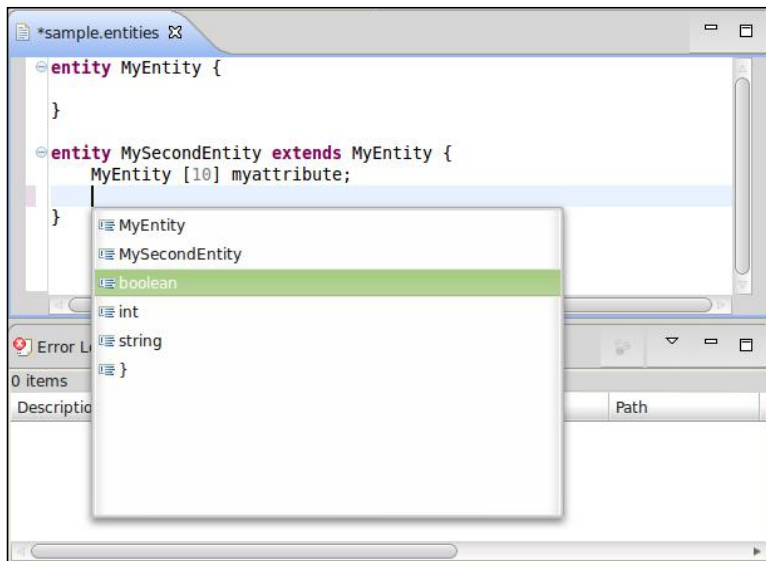
基本类型: |实体类型;

基本类型: type名称=(“字符串” | int “|” 布尔”);

实体类型: 实体=[实体];

正如您所看到的，我们引入了一个规则，元素类型，它又依赖于两个替代规则（互排斥）：基础类型和实体类型。使用管道运算符“|”分隔替代规则。请注意，像元素类型这样的规则基本上委托给其他替代规则，它在生成的EMF类中隐式地引入了继承关系；因此，继承关系的基本类型和实体类型都从元素类型继承。在“基本类型”规则中，字符串特征字名称将包含在程序中输入的相应关键字。

运行MWE2工作流后，可以尝试编辑器，现在也可以使用三种基本类型；此外，在需要类型规范的上下文中，内容辅助（参考以下屏幕截图）将提供所有可能的元素类型替代方案（包括实体类型和基本类型）：



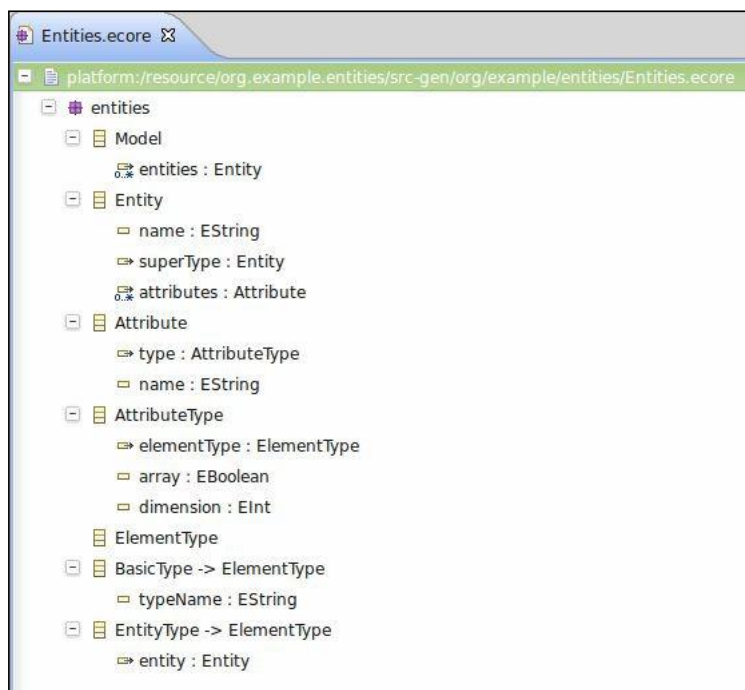


一般来说，最好不要像以前使用的基本类型那样依赖硬编码的替代方法。相反，最好为DSL提供一个具有一些预定义类型和函数的标准库。该技术的应用将参见第10章，范围界定。

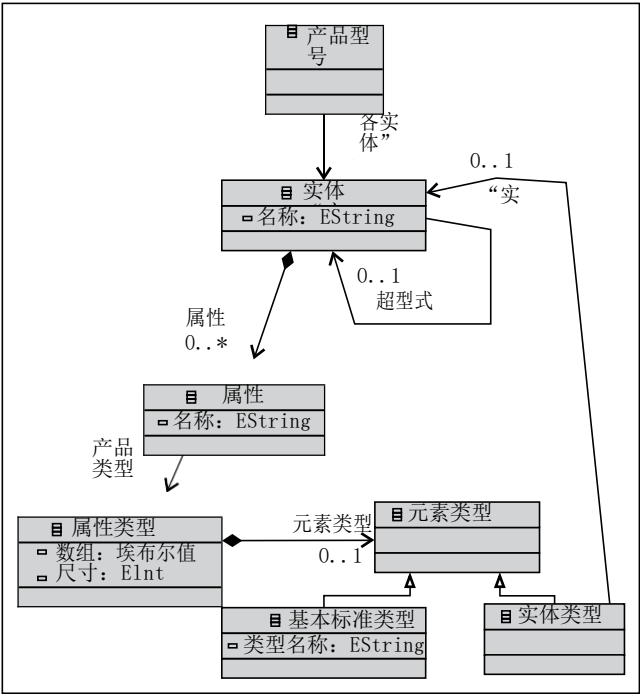
现在我们的EMF模型再次改变了，我们需要改变我们的实体EMF示例类如下：

```
属性属性=factory.createAttribute();
attribute.setName("my属性");
属性类型属性类型=factory.createAttributeType(); attributeType.setArray(false);
attributeType.setLength(10);
实体类型实体类型=factory.createEntityType();
entityType.setEntity(超实体); attributeType.setElementType
(实体类型); attribute.setType(属性类型);
```

如果重新打开生成的Entities.ecore（请参阅以下屏幕截图），您可以看到DSL的AST的当前元模型（请注意继承关系：基本类型和实体类型从元素类型继承）：



前面的EMF元模型也可以用以下类似UML的约定的图形符号来表示（请参见下图）：



我们现在在实体DSL中有足够的特性来开始处理典型的语言实现的额外任务。我们将在接下来的章节中使用这个示例DSL。

产品简介

在本章中，您学习了如何使用Xtext实现一个简单的DSL，您可以看到，从语法定义开始，Xtext会自动为DSL生成许多工件，包括IDE工具。

您还开始学习EMF API，它允许您以编程方式操作表示程序AST的模型。能够以编程方式访问模型对于对已解析的程序执行额外检查以及执行代码生成至关重要，正如我们将在书的其余部分中看到的那样。

在下一章中，我们将介绍新的编程语言Xtend(随Xtext附带，并用Xtext本身实现)：一种类似Java的通用编程语言与Java紧密集成，允许您编写更简单、更干净的程序。我们将在书的其余部分中使用Xtend来实现在Xtext中实现的语言的所有方面。

3

Xtend编程

语言

在本章中，我们将介绍Xtend编程语言，a类似Java的语言，与Java紧密集成。Xtend具有比Java有更简洁的语法，并提供了其他功能，如类型推理、扩展方法和lambda表达式，更不用说多行模板表达式（这在编写代码生成器时非常有用）。在Xtext中实现的DSL的所有方面都可以用Xtend而不是Java来实现，因为它更容易使用，并且允许您编写更具可读性的代码。由于Xtend完全与Java互操作，因此可以重用所有Java库；此外，所有EclipseJDT (Java开发工具)都将与Xtend无缝工作。

对Xtend的介绍

Xtend编程语言附带了非常好的文档，可以在其网站上找到，<http://www.eclipse.org/xtend>. 我们会给出一个答案本章中概述了Xtend，但我们强烈建议您彻底浏览一下Xtend文档。Xtend本身是在Xtext中实现的，它证明了在Xtext中实现的语言的概念。

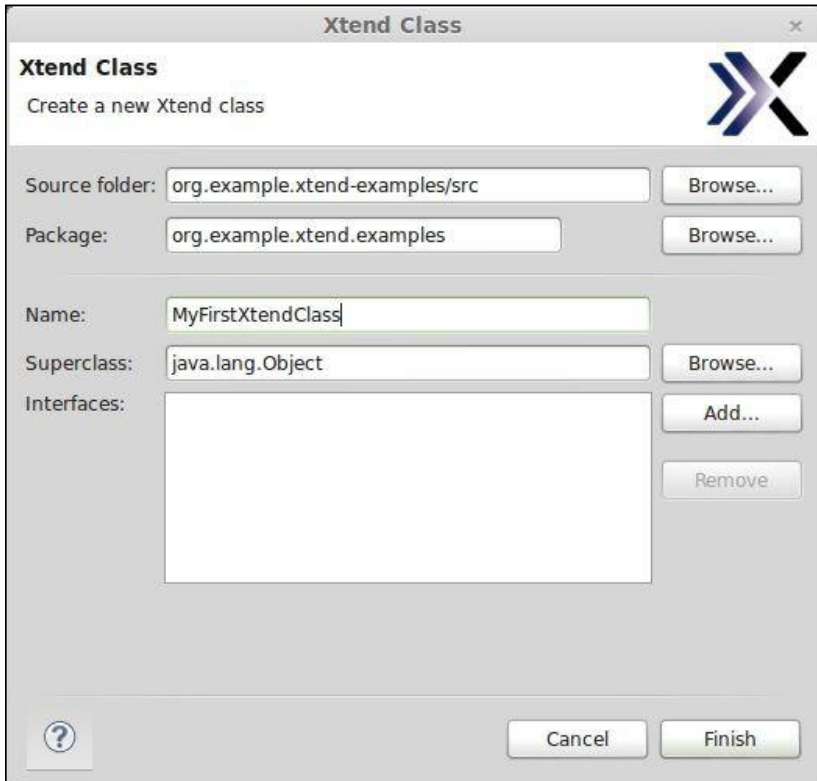
我们将在整本书中使用Xtend来编写DSL实现的所有部分。也就是说，我们将使用它来自定义UI特性、编写测试、实现约束检查，以及为我们将在本书中开发的所有DSL示例编写代码生成器或解释器。特别是，从2.4版本开始，Xtext为DSL项目生成的所有存根类默认都是Xtend类(而不是以前版本中的Java)。

您仍然可以通过自定义MWE2工作流来生成Java存根类，但是在这本书中，我们将始终使用Xtend类。Xtend除了为编写代码生成器提供有用的机制（最重要的是多线模板表达式）外，还提供了强大的特性，使模型访问和遍历非常容易、直接、读取和维护。事实上，除了语法定义之外，在实现DSL时的剩余时间里，您还必须访问AST模型。Xtend程序被翻译成Java，Xtend代码可以访问所有的Java库，因此Xtend和Java可以无缝共存。

在项目中使用Xtend

您可以在EclipseJava项目（包括普通Java和插件项目）中使用Xtend。

在插件项目中，可以右键单击源文件夹并选择新|Xtend类；您将看到此向导类似于标准的新Java类向导，因此可以选择包、类名称、超级类和界面（请参阅以下屏幕截图）：



一旦创建了类，您将会得到一个错误标记，其中有消息“在类路径上没有找到强制性库包‘org.eclipse.xtext.xbase.lib’ 2.4.0或更高版本”。您只需要使用快速修复程序“将Xtend库添加到类路径中”，所需的Xtend包将被添加到项目的依赖项中。

将在您的插件项目xtend-gen中创建一个新的源文件夹，一旦保存一个。xtend文件，与Xtend代码对应的Java代码将自动生成（使用Eclipse的构建基础结构）。就像由Xtext生成器创建的src-gen（如上章所述）一样，程序员不能对其在x文本-gen中的文件进行修改，因为它们将被Xtend编译器覆盖。



文件夹xtend-gen不会自动添加到插件项目的构建源文件夹中，因此，您应该手动将其添加到build.properties文件中（该文件有一个警告标记，编辑器将为您提供一个快速修复程序来添加该文件夹）。仅对插件项目进行要求。

您可以使用相同的步骤在普通Java项目中创建Xtend类（同样，您必须使用快速修复程序将Xtend库添加到类路径中）。当然，在这种情况下，没有build.properties需要调整。



从2.4.0版本开始，Xtext会自动为您的DSL（而不是Java存根类）生成Xtend存根类，因此，运行时和UI插件项目已经被设置为使用Xtend及其库。

但是，这并不适用于测试插件项目；因此，当开始在测试插件项目中编写Xtend类时（我们将在第7章，测试中看到），您需要执行本节中描述的设置步骤。

Xtend——使用更少的“噪音”的更好的Java

Xtend是一种静态类型的语言，它使用Java类型的系统（包括Java泛型）。因此，Xtend和Java是完全可互操作的。

Xtend的大多数语言概念与Java非常相似，即类、接口和方法。此外，Xtend还支持Java注释的大部分特性。Xtend的目标之一是有一个不那么“噪音”的Java版本；事实上，在Java中，一些语言特性是冗余的，只会使程序更加冗长。

让我们用Xtend写一个“你好，世界”程序：

软件包org.example.xtend.examples

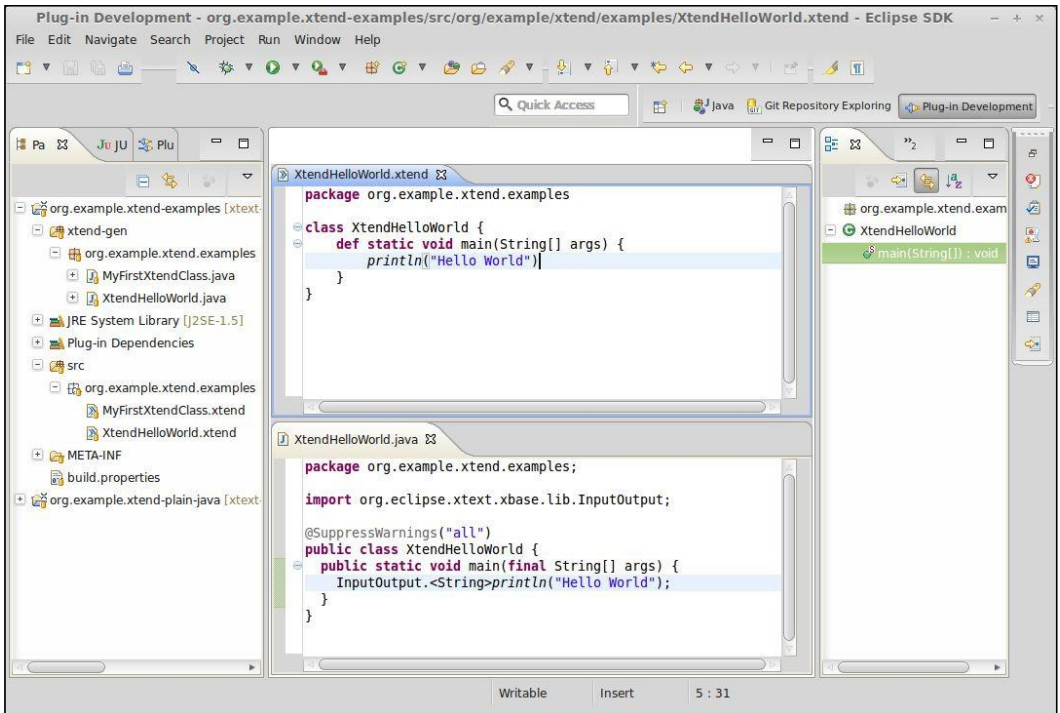
类

```
def静态vaingargs){  
    打印件（“你好，世界”）  
}  
}
```

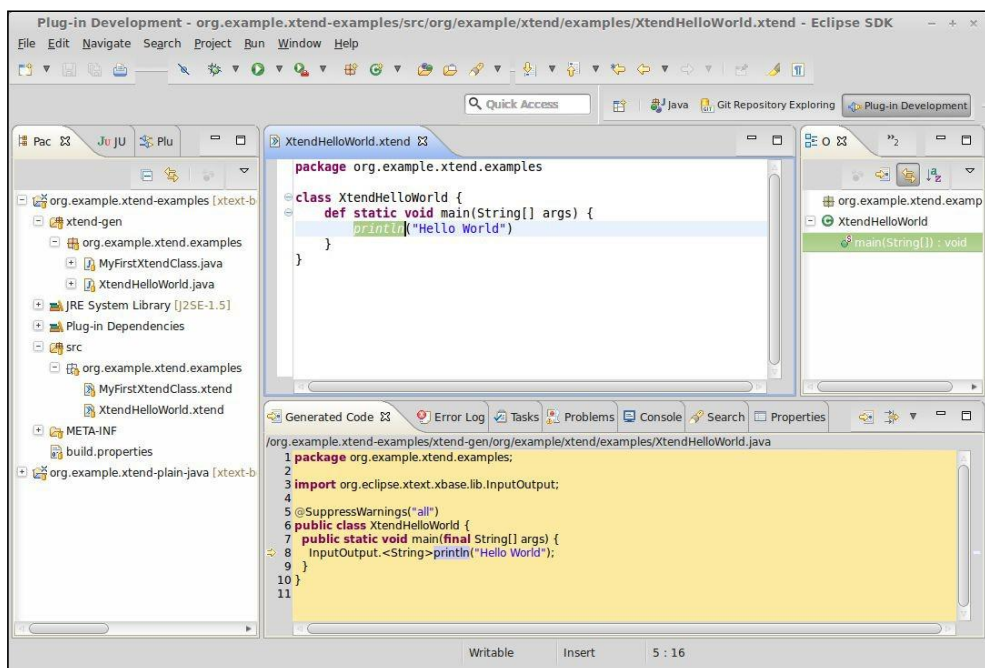
您可以看到它与Java相似，尽管有一些不同。首先，缺少的分号；不是错误：在Xtend中，它们不是必需的（尽管它们可以使用）。所有方法声明都以def或覆盖开始（本章后面说明）。方法在默认情况下是公共的。

请注意，该编辑器的工作原理与JDT提供的编辑器几乎相同

（例如，请参见以下屏幕截图中的Xtend类的大纲视图）。您可能还希望查看与Xtend类对应的xtend-gen文件夹中生成的Java类（请参阅以下屏幕截图）：



虽然通常不需要查看生成的Java代码，但在开始使用Xtend时，查看生成的内容可能很有帮助，特别是在学习Xtend的新构造时。不用手动打开生成的Java文件，您可以打开Xtend生成的代码视图；此视图的内容将显示生成的Java代码，特别是与正在编辑的Xtend文件部分对应的代码（参见下面的屏幕截图）。当保存Xtend文件时，将更新此视图。



下面还有一些Xtend示例：

类

vals=“我的领域”//最终领域
varmyList=新的领英列表<整数>//非最终字段

```
def条（字符串输入）{
    var缓冲区=输入
    缓冲区==||==>0
    //，最后一个表达式是返回表达式
}
```

字段和局部变量使用val（最终字段和变量）
和var（对于非最终字段和变量）。字段在默认情况下是私有的。

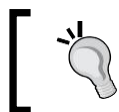
您可能已经注意到，我们编写了`==`，用于比较两个Java字符串。这通常是Java中常见的错误之一，因为您将比较对象引用，而不是它们的实际值。然而，Xtend透明而正确地处理操作符，因此`==`实际上是比较了字符串的值（实际上，它映射到方法等于）。

Xtend为getter和setter方法提供了一些“语法糖”（即编写更容易阅读的语法），因此，您可以简单地写`getName()`，而不是写`o`；同样，与其写`o.setName`（“...”），您可以写`o.name=“.....”`。根据JavaBeans约定（getter方法以开头而不适用于get）的布尔字段。方法调用也有类似的句法糖，因此，当一个方法没有参数时，可以避免使用括号。因此，在前面的代码中，`myList.size`对应于`myList.size()`。

前面的代码还显示了Xtend的其他重要特性：

- 类型推断：当可以推断出变量的类型时（例如，从其初始化表达式中）时，不需要指定它。
- 一切都是一个表达式：在Xtend中没有语句，一切都是一个表达式；在方法体中，最后一个表达式是返回表达式（注意没有返回，尽管可以显式指定返回表达式）。

不需要方法的返回类型的声明：从返回的表达式推断出来（本例中是布尔值）。



编辑器中Xtend元素的文本将提供有关推断类型的信息。

请注意，必须始终指定方法参数的类型。

访问Xtend中类的静态成员（字段和方法）必须使用运算符：`::`来完成，而不是只用于非静态成员（与Java不同。均用于两者），例如：

```
导入java.util.Collections
类静态方法{
    定义静态空主（字符串参数）{参数列表=集合：：
        emptyListSystem：： out.println（列表）
    }
}
```

访问Java类的内部类/接口是通过使用\$来完成的，例如，
给定了这个Java类：


```
公共课程
    我的界面
        公共静态字符串的= “s” ;
    }
}
```

我们可以使用以下语法访问Xtend中的内部接口：

```
我的接口$接口
```


因此，可以访问静态字段：

```
我的接口接口： : s
```

 在Xtend2.4.2中，静态成员也可以通过点运算符访问它们和内部类型。因此，前一行也可以写如下：
MyJavaClass.MyInnerInterface.s。

对Java类的引用，即类型文本(在Java中通过使用
类名后跟为.class)，用typeof(类名)表示，例如：

```
类型(实体) //对应于Java中的Entity.class
```

 在Xtend2.4.2中，类型文本也可以用其简单的名称来指定：您可以简单地写入实体，而不是类型(实体)。

Xtend对方法覆盖更严格：如果子类覆盖方法，
它必须用重写而不是def显式地定义该方法，否则就会引发编译错误。这应避免意外
的方法覆盖（即，您不打算提供超类方法的重写版本）。更重要的是，如果
删除稍后被覆盖的方法，您将希望知道为什么可能永远不会调用您的方法。

在Xtend中（一般情况下，默认情况下，使用默认终端语法使用Xtext实现的任何
DSL中），可以用单引号和双引号指定字符串。这允许程序员根据字符串内容选择
首选格式，以便字符串中的引号不必转义，例如：

```
vals1= “我的 “字符串” ”
vals2= ‘我的 “字符串” ’
```

使用Java中的反斜杠字符仍然可以逃脱。

扩展的方法

扩展方法是一种语法糖机制，它允许您向现有类型添加新方法而不修改它们；与其在方法调用的括号内传递第一个参数，还可以使用第一个参数作为接收器。该方法似乎是参数类型的成员之一。

例如，如果`m`（实体）是一个扩展方法，并且`e`的类型为实体，则可以编写`e.m()`而不是`m(e)`，即使`m`不是在实体中定义的方法。

使用扩展方法通常会导致更可读的代码，因为方法调用是链的；例如，`o.foo()`。
`bar()`而不是嵌套，例如：`bar(foo(o))`。

Xtend提供了几种方法来使方法作为扩展方法使用。

Xtend提供了一个具有多个实用程序类和静态方法的丰富运行时库。这些静态方法在Xtend代码中自动可用，因此您可以将它们全部用作扩展方法。

当然，编辑器还为扩展方法提供了代码完成功能，以便您可以尝试使用代码助手。这些实用程序类的目的是增强标准类型和集合的功能。



扩展方法在Xtend编辑器中以橙色突出显示。

例如，您可以写下：

“我的字符串”。`上上`

以以下代替：

字符串扩展名。第一个上（“我的字符串”）

类似地，您可以对集合使用一些实用程序方法，例如，如下代码中的`addAll`：

```
val列表=新的仲裁列表<字符串>列表。所有  
件（“a”、“b”、“c”）
```

您还可以使用来自现有Java实用程序类中的静态方法（例如，`java.直到。集合`）通过在Xtend源文件中使用静态扩展导入作为扩展方法，例如：

导入静态扩展名`java.util.Collections.*`

在该Xtend文件中，`java.util.Collections`的所有静态方法都将作为扩展方法使用。

在Xtend类中定义的方法可以自动用作扩展方法

在那个类中，例如：

```
类扩展方法{
    def my列表方法（列出<?>列表）{
        //的一些实现
    }

    度(){
        val 列表=新的仲裁列表<字符串>list.myListMethod
    }
}
```

最后，通过将“扩展”关键字添加到字段、局部变量或参数声明中，其实例方法分别成为该类、代码块或方法体中的扩展方法。例如，假设您有此类：

```
类我的列表扩展{

    def a列表方法（列出<?>列表）{
        //的一些实现
    }

    def 其他列表方法（列表<?>列表）{
        //的一些实现
    }
}
```

您希望将其方法作为另一个类C中的扩展方法。然后，可以在C中声明MyList扩展类型的扩展字段（即扩展字段声明），在C的方法中，可以使用MyListExtensins中声明的方法作为扩展方法：

```
C类{

    扩展我的列表扩展e=新的我的列表扩展

    度(){
        val 列表=新的仲裁列表<字符串>list.aListMethod
        list.anotherListMethod
    }
}
```

您可以看到，我的列表扩展的两个方法在C中被用作扩展方法。实际上，方法m中的两个方法调用等价于：

```
e. a列表方法（列表） e. 另一个列表方法（列表）
```

如前所述，您可以通过添加该关键字来实现相同的目标对局部变量的扩展：

```
度() {  
    =新的列表扩展  
    val列表=新的仲裁列表<字符串  
    >list.aListMethod  
    list.anotherListMethod  
}
```

或指向一个参数声明：

```
{val列表=新的Ararray列表<字符串  
    >list.aListMethod list.anotherListMethod  
}
```

当声明具有关键字扩展名的字段时，该字段的名称是可选的。在声明具有关键字扩展名的局部变量时，也是如此。

正如我们前面暗示的，Xtend库类的静态方法在Xtend程序中自动可用。实际上，静态方法可以独立于扩展方法的机制来使用。第一个Xtend实例类中的println方法确实是来自Xtend实用程序类的静态方法。提供了许多处理集合的实用方法；特别是，而不是写作：

```
val列表=新的仲裁列表<字符串>列表。所有  
件（“a”、“b”、“c”）
```

我们可以简单地写道：

```
参数列表=新参数列表（“a”、“b”、“c”）
```

通过使用这些方法，您可以充分利用Xtend的类型推理机制。

隐式变量

您知道，在Java中，这个特殊变量在一个方法中隐式地绑定到调用该方法的对象；在Xtend中也是如此。

然而，Xtend也引入了另一个特殊的变量它。虽然不能声明名称为此的变量或参数，但允许使用其名称这样做。如果在当前程序上下文中存在声明，则该变量的所有成员都可以隐式可用，而不使用。（就像所有成员在实例方法中隐式可用），例如：

```
类It示例{
    deftrans1(字符串){到小写
        //it.toLowerCase
    }

    deftrans2(字符串){
        变量是=
        至小写字母//it.toLowerCase
    }
}
```

这允许您编写更紧凑的代码。

Lambda表达式

lambda表达式（或简称）定义了一个匿名函数。Lambda表达式是可以传递给方法或存储在变量中。

Lambda表达式是典型的函数语言，早在设计之前就有面向对象语言存在了。因此，作为一种语言机制，它们是如此之老，以至于Java从一开始就没有提供它们是很奇怪(它们计划在Java8中提供)。在Java中，可以使用匿名内部类模拟lambda表达式，如下例所示：

```
导入java.util.*;
公共阶级
    公共静态空白主(字符串[]args){List<字符串>字符串=新的
        ArrayList<字符串>();
        ...
        Collections.sort(字符串,          新的比较器<字符串>(){
            公共int比较(左字符串, 右字符串){
                返回left.compareToIgnoreCase(右);
            }
        });
    }
}
```

在前面的例子中，排序算法只需要一个实现比较的函数；因此，将匿名函数传递给其他函数的功能将使事情变得更加容易（Java中的匿名内部类通常用于模拟匿名函数）。

实际上，匿名内部类的大多数使用只使用只有一种方法的接口（或抽象类）（因此，它们也被称为SAM类型-单抽象方法）；这应该更明显地表明，这些内部类的目的是模拟lambda表达式。

Xtend支持lambda表达式：它们使用方括号[]进行声明；参数和实际主体由管道符号|分隔。lambda的主体通过调用其应用方法并传递所需的参数来执行。

下面的代码定义了一个分配给局部变量的lambda表达式，以一个字符串和一个整数作为参数，并返回两者的字符串连接。然后，它计算传递这两个参数的lambda表达式：

```
val l=[字符串, inti|s+i]  
println(l.apply(“s”, 10))
```

Xtend还引入了lambda表达式的类型（函数类型）；参数类型（用括号括起来）与返回类型由符号分隔

=>(当然，在定义lambda表达式类型时，可以充分利用泛型类型)。例如，前面的声明可以使用如下显式类型编写：

```
val(字符串, int)=>字符串l=[字符串, inti|s+i]
```

回想一下，Xtend具有强大的类型推理机制：当上下文提供了足够的信息时，可以省略变量类型声明。在前面的声明中，我们明确表示了lambda表达式的类型，因此lambda表达式的参数类型是冗余的，因为它们可以推断出来：

```
val(字符串, int)=>字符串l=[s, i|s+i]
```


函数类型在声明以lambda表达式作为参数的方法时非常有用（请记住，必须始终指定参数的类型），例如：

```
def执行((字符串, int)=>字符串f){f.应用程序  
    (“s”, 10)  
}
```

然后，我们可以传递一个lambda表达式作为该方法的参数。当我们将一个lambda作为参数传递给该方法时，有足够的信息可以充分推断其参数的类型，这允许我们省略这些声明：

```
执行([s, i|s+i])
```

lambda表达式还捕获当前作用域；在定义时可见的所有最终局部变量和所有参数都可以在lambda表达式的主体中引用。这类似于Java匿名内部类，它可以访问Java周围的最终变量和最终参数。


 在Xtend中，方法参数总是自动为最终的。

实际上，在引擎盖下，Xtend会为lambda表达式生成Java内部类（而且将来它将与Java8兼容）。lambda表达式在定义它的环境上被“关闭”：封闭上下文的引用变量和参数被lambda表达式捕获。因此，lambda表达式通常被称为闭包。

例如，请考虑以下代码：

```
软件包org.example.xtend.examples
类羊肉示例
def静态执行((字符串, int)=>字符串f){f.apply(“s”, 10)
}
def静态vaingargs){
    瓦尔c=“aaa”
    打印n(执行([s, i|i+i+]))//打印s10aaa
}
}
```

您可以看到，lambda表达式在定义时使用了局部变量c，但是该变量的值即使经过计算，也可用。

 在形式上，表达式是一种语言构造，而闭包是一种实现技术。从另一个角度来看，lambda表达式是函数文字定义，而闭包是函数值。然而，在大多数编程语言和文献中，这两个术语经常可互换使用。

虽然函数类型在Java中不可用，但Xtend可以自动执行所需的转换；特别是，Xtend可以自动处理JavaSAM（单一抽象方法）类型：如果Java方法需要SAM类型的实例，在Xtend中，可以通过传递lambda调用该方法（Xtend将执行所有类型检查和转换）。这进一步演示了Xtend是如何与Java紧密集成的。

因此，使用`java.util.Collections.sort`传递内部类的Java示例可以在Xtend中编写如下(当使用lambda时，代码要紧凑得多)：

```
vaw列表=新的列表（“第二”，“第一”，“第三”）集合：：排序
（列表，
    [arg0, arg1|arg0.compareToIgnoreCase(arg1)])
```

再次注意Xtend如何推断lambda参数的类型。

Xtend还支持循环；特别是，在对于循环中，可以使用类型推断并避免写入类型（例如，对于（`s: list`））。然而，如果你熟悉了lambdas，你会发现你往往不会最使用循环

的时间。特别是，在Java中，通常使用循环在集合中找到某个东西；使用Xtendlambdas（以及可以自动用作库的所有实用方法作为扩展方法），您不需要再编写这些循环；您的代码将更具可读性。例如，考虑此Java代码（其中字符串是List<字符串>）：

```
字符串找到=null;
用于（字符串字符串：字符串）{
    如果(string.startsWith(“F”)){找到
        =字符串;
        中断
    }
}
系统.out.println（发现）;
```

使用lambda的相应的Xtend代码如下：

```
打印符（字符串。查找第一([s|。startsWith(“F”)])）
```

在Java版本中，您不能立即理解代码的作用。

Xtend版本几乎可以被解读为一个英语句子（有一些已知的数学符号）：“在集合中找到第一个元素，这样s就以F开头”。

Xtend为lambdas提供了一些额外的句法糖，以使代码更具可读性。

首先，当lambda是方法调用的最后一个参数时，它可以放在 (...) 括号之外（如果调用只需要一个参数，则可以省略()）：

```
集合：： 排序（列表）[arg0,
    arg1|arg0.compareToIgnoreCase(arg1)]
字符串。首先[s|。开始（“F”）]
```

此外，我们前面引入的特殊符号也是lambda表达式中的默认参数名称；因此，如果lambda只有一个参数，则可以避免指定它，而是将其用作隐式参数：

```
字符串。查找首先[it.startsWith（“F”）]
```

因为它的所有成员都可以隐式地可用而不使用”。”，你可以简单地写下以下内容：

```
字符串。找到第一[明星与（“F”）]
```

这甚至更具可读性。如果需要定义一个根本不需要参数的lambda，则需要通过定义一个空参数列表来显式它

|符号。例如，以下代码将发出编译错误，因为lambda（隐式）需要一个参数：

```
val1=[打印件（“你好”）]1。应
用()
```

正确的版本应表述如下：

```
val1=[|打印n（“你好”）]1。应用()
```

在为DSL实现检查和生成代码时，大多数时候您必须检查模型和遍历集合；前面的所有功能将对您在这方面有很大帮助。如果您仍然不相信，让我们试试练习：假设您有以下人员列表（其中Person是具有字符串字段名、姓氏和整数字段年龄的类）：

```
个人列表=新的列表(
    新人（“詹姆斯”，“史密斯”，50岁），
    新人物（“约翰”，“史密斯”，40岁），
    新人（“詹姆斯”、“安德森”，40岁）、新人
    （“约翰”、“安德森”，30岁）、新人（“保
    罗”、“安德森”，30岁）
```

我们想找到前三个名字以J开头的年轻人，我们想把他们印成“姓氏，姓”在同一行用“；”分隔，因此，结果应该是（注意：；必须是分隔符）：

安德森，约翰；史密斯，约翰；安德森，詹姆斯

尝试在Java中这样做：在Xtend(使用lambdas和扩展方法)中，它同样简单
具体情况如下：

```
val结果=个人列表.filter{firstname.startsWith("J")}.  
    按年龄进行分  
    类。采取  
    (3)。  
    地图[姓氏+"", "+的名字"]。  
    加入("；")  
打印结果(结果)
```

多行模板表达式

除了遍历模型之外，在编写代码生成器时，大多数时候您会编写表示生成代码的字符串；不幸的是，对于这个任务，Java并不理想。事实上，在Java中，您不能编写多行字符串文字。

这实际上会导致两个主要问题：如果字符串必须包含换行符，则必须使用特殊字符
\\n；如果为了可读性，要将字符串文字分成几行，则必须将字符串部分与+连接。

如果您只需要生成几行，这可能不是一个大问题；然而，DSL的生成器通常需要生成许多行。

例如，让我们假设您要编写一个用于生成一些Java方法定义的生成器；您可以使用负责生成特定部分的方法编写一个Java类，如下代码所示：

```
公共代码生成器  
    公共字符串生成体（字符串名称，字符串代码）{  
        返回 "+name+" 的 "/*正文*/\\n" +代码；  
    }  
  
    公共字符串生成方法（字符串名称，字符串代码）{  
        返回 "公共void"+名称+ "() { "+ "\\t "+通用主体  
            (名称，代码)+ "}" ；  
    }  
}
```

然后，您可以按如下方式调用它：

```
Java编码生成器发电机=新的Java编码生成器(); 系统。  
out.println(generator.generateMethod(“m”,  
    “System.out.println(你好); 返回;));
```

然而，我们可以发现这种方法的一些缺点：

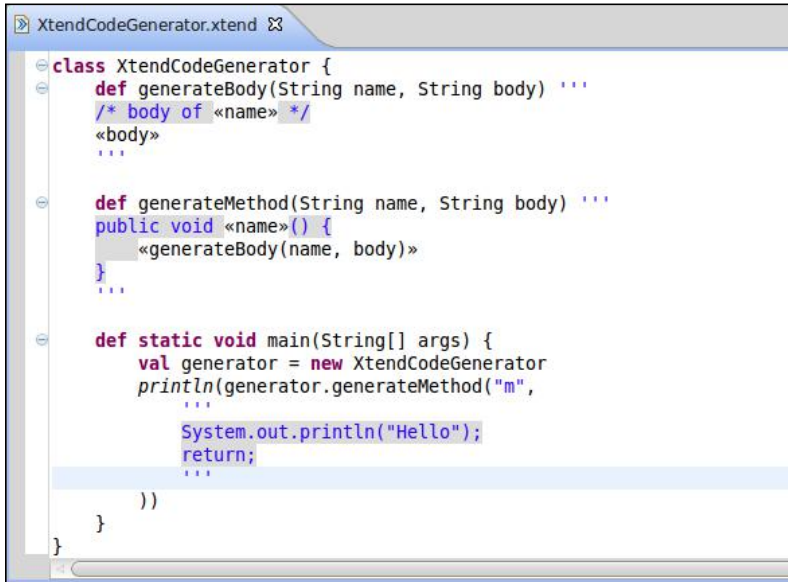
- 最终生成的代码的形状并不能立即被理解（可变零件破坏常定零件）
- 处理代码缩进并不简单
- 纽行和标签字符不容易识别
- 某些经常性字符必须被手动转义(例如，“”报价单)

运行该发电机的结果如下：

```
公共空白m() {  
    m*/System.out.prin  
tln的/*身体 (“你好”) ); 返回; }
```

虽然生成的代码是有效的Java代码，但它的格式不是很好(Java编译器不关心格式化，但生成格式良好的代码会很好，因为程序员可能想要读取或调试它)。这是由于我们在Java代码生成器类中编写方法的方式：代码在这方面是错误的，但在使用Java字符串时，要正确地得到这一点并不容易。

Xtend提供了多行模板表达式来解决上述所有问题（事实上，Xtend中的所有字符串都是多行的）。使用多行模板表达式用Xtend编写的相应代码生成器如下屏幕截图所示：



```
class XtendCodeGenerator {
    def generateBody(String name, String body) '''
        /* body of «name» */
        «body»
    '''


    def generateMethod(String name, String body) '''
        public void «name»() {
            «generateBody(name, body)»
        }
    '''

    def static void main(String[] args) {
        val generator = new XtendCodeGenerator
        println(generator.generateMethod("m",
            '''
                System.out.println("Hello");
                return;
            ''')
        ))
    }
}
```

在解释代码之前，我们必须首先提到，最终的输出很好格式的格式如下：

```
公共空白m() {
    /*身体的m*/System.out.println
    （“你好”）； 返回；
}
```

模板表达式使用三重单引号定义（这允许我们直接使用双引号而不转义它们）；它们可以跨越多行，表达式中的换行将与最终输出中的换行相对应。变量零件可以使用网格（也称为角引号或法语引号）直接插入到表达式中。注意，在网格之间，可以指定任何表达式，甚至可以调用方法。您也可以使用条件表达式和循环（我们将在本书后面看到一个例子；有关所有详细信息，请参考该文档）。

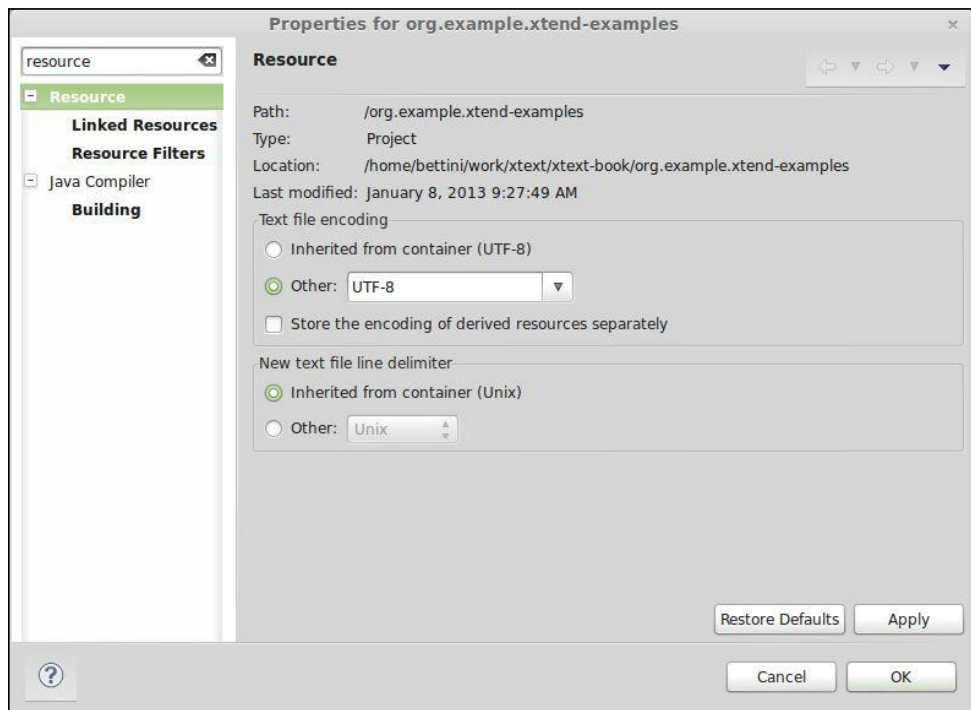
 曲线括号 {} 对于仅包含模板表达式的Xtend方法体是可选的。

模板表达式的另一个重要特性是，可以以一种智能的方式自动处理缩进。正如您从前面的屏幕截图中看到的，Xtend编辑器对多行模板字符串使用了特定的语法着色策略，以便让您了解最终输出中的缩进将是什么样子。



要将鼠标插入Xtend月食编辑器中，可以分别使用键盘快捷键 Ctrl+Shift+< 和 Ctrl+Shift+>。在Mac操作系统上，它们也可以使用 Alt+q() 和 Alt+Q()。或者，可以使用模板表达式中的内容辅助插入一对。

指南的缺点是必须有一致的编码，特别是当您在团队中使用不同操作系统的团队工作时。您应该始终对那些使用Xtend的所有项目使用UTF-8编码，以确保正确的编码存储在项目首选项中（进而保存在版本控制系统中，如Git）。你应该右键单击这个项目，然后选择“属性”。，并在资源属性中，显式地设置编码（请参考以下屏幕截图）。在编写任何Xtend代码之前，必须设置此属性（稍后更改编码将更改所有规则字符，并且必须手动修复）。像Windows这样的系统使用的默认编码在Linux等其他系统中是没有的，而UTF-8随处可用。



其他操作员

除了标准运算符之外，Xtend还有有助于保持代码紧凑。

首先，大多数标准运算符都被扩展到具有预期意义的列表。例如，在执行以下代码时：

```
val l1 = 新地址列表
(“a”) l1 += “b”
= l2 新地址列表(c) = 1 + 打印 (l3)
```

将打印字符串a、b、c。

通常情况下，在调用其方法之前，您必须检查一个对象是否为null，否则您可能需要返回null(或者简单地执行no操作)。正如您在DSL开发中看到的，这是一个相当经常出现的情况。Xtend提供了操作员`?.`，这是标准选择操作符的空安全版本。)。写`o?.m`对应于`if (o != null) o.m`。这在您有级联选择时特别有用，例如，`o?.f?.m`。

猫王们吗？：运算符是处理出现空实例时的默认值的另一个方便的操作符。它有以下语义：`x? : y`返回x，则返回x，否则。

结合这两个运算符，可以轻松地设置默认值，例如：

```
//等价于：如果(o != null)o.toString()其他“默认”结果=o?.toString()：“默认值”
```

with运算符（或双箭头运算符）`=>`将对象绑定到lambda表达式的作用域，以便对其执行某些操作；此运算符的结果就是对象本身。形式上，运算符`=>`是一个二进制运算符，它在左侧接受一个表达式，在右侧使用一个参数的lambda表达式：操作符执行以左侧为的lambda表达式

该参数。结果是应用lambda表达式后的左操作数。例如，该代码包括：

```
返回eINSTANCE.createEntity=>[名称=“MyEntity”]
```

相当于：

```
val 实体=eINSTANCE.createEntity
entity.name=“实体”
返回的实体
```

这个运算符与隐式参数和获取者和设置者的语法糖相结合非常有用，它可以初始化新创建的对象，以便不使用临时变量（再次增加了代码的可读性）。作为演示，请考虑我们在中看到的Java代码片段

章2, 创建你的第一个Xtext语言, 我们用来构建一个实体与
为了方便起见, 我们将在此报告的属性（及其类型）:

```
实体实体=eINSTANCE.createEntity(); entity.setName
    (“我实体”); entity.setSuperType(超级实体);
属性属性=eINSTANCE.createAttribute(); attribute.setName(“my属性”);
属性类型属性类型=eINSTANCE.createAttributeType(); attributeType.setArray(false);
attributeType.setDimension(10);
实体类型实体=eINSTANCE.createEntityType()类型;
entityType.setEntity(超实体); attributeType.setElementType
    (实体类型); attribute.setType(属性类型);
entity.getAttributes().add(属性);
```

这需要许多变量，这是一个巨大的干扰(你能快速得到一个吗知道代码是做什么？)。在Xtend中，我们可以简单地写道：

```
eINSTANCE.createEntity=>[命名
    = “My实体” 超类型=超实体
    属性+=eINSTANCE.createAttribute=>[名称= “my属性”
        键入=eINSTANCE.createAttributeType=>[数组=false]
            尺寸尺寸: =10
            元素类型=          电力。创建实体类型=>[实体=超级实体
        ]
    ]
]
```


多态方法调用

Java中的方法重载解析(在Xtend中的默认情况下)是一种静态机制,这意味着特定的方法会根据参数的静态类型进行选择。当您处理属于a的对象时类层次结构,这个机制很快就显示了它的局限性:您可能会编写通过引用其基类来操作多个多态对象的方法,但是由于静态重载只使用这些引用的静态类型,因此拥有这些方法的多个变体是不够的。对于多态方法调用(也称为多个分派或动态重载),将根据参数的运行时类型进行方法选择。

Xtend为多态方法调用提供了分派方法:在调用时,会根据参数的运行时类型选择标记为分派的重载方法。

回到第2章的实体DSL,创建您的第一个Xtext语言,元素类型是元素类型的基本类型,属性类型具有对元素类型的引用元素类型;要为此引用具有字符串表示,我们可以编写两种分派方法,如下例:

```
def调度类型到字符串(基本类型类型){type.typeName
}
def调度类型到字符串(实体类型类型){type.entity.name
}
```

现在,当我们对引用元素类型调用类型到字符串时,该选择将使用该引用的运行时类型:

```
def到字符串(属性类型、属性类型)
{attributeType.elementType.typeToString
}
```

有了这种机制,您可以消除困扰许多Java程序的级联(和显式类转换)。

增强的交换机表达式

Xtend提供了一个更强大的Java交换机语句版本。首先,与从一个情况下降到下一个情况的Java相比,只对选定的情况被执行(因此,您不必插入显式的断开指令来避免随后的情况块执行);此外,开关可以与任何情况一起使用对象引用。

Xtend开关表达式允许您编写涉及的案例表达式，如下示例所示：

```
def字符串开关示例（实体e，实体专用实体）{
    开关e{
        案例e.name.length>0: “有一个名称”
        案例e.超类型!=null: “具有超级类型”案例特殊实体:
            “特殊实体”默认值: “”
    }
}
```

如果case表达式是布尔表达式（如上例中的前两个示例），则如果case表达式的计算结果为true，则case匹配。如果案例表达式不是布尔类型，则与使用等于方法的主表达式值（上例中的第三例）进行比较。然后计算匹配案例的冒号之后的表达式，这个求值是整个开关表达式的结果。

Xtend开关表达式的另一个有趣的特性是类型保护程序。使用此功能，您可以将类型指定为事例条件，并且只有当开关值是该类型的实例时（正式上，如果符合该类型），事例才会匹配。特别是，如果开关值是一个变量，则该变量将自动转换到案例体中的匹配类型。这允许我们实现实例化和显式强制转换的典型Java级联的更清晰版本。虽然我们可以使用调度方法来实现相同的目标，但具有类型保护功能的交换表达式可以是一个有效的和更紧凑的替代方案。

例如，使用调度方法的代码可以重写如下：

```
字符串（属性类型属性类型）=attributeType.elementType开
关类型
    基本类型：//元素类型是一个基本类型，这里的
        elementType.typeName
    实体类型：//元素类型是一个实体类型，在这里是
        elementType.entity.name
}
```

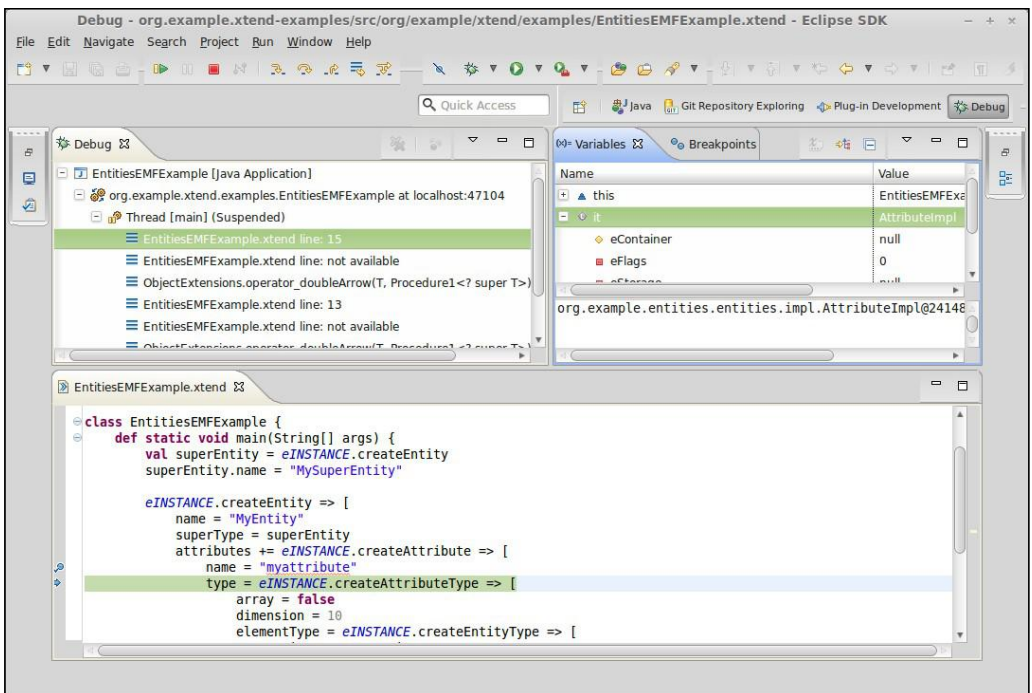
注意整体类型如何自动倒放到案例体中的匹配类型。

根据您的编程场景，您可能需要在调度方法和基于类型的交换表达式之间进行选择；但是，请记住，虽然分派方法可以被覆盖和扩展（即在派生类中，可以为基类中未处理的参数组合提供额外的分派方法），但交换表达式在方法内部，因此它们不允许相同的可扩展性特性。此外，调度案例会根据类型层次结构自动重新排序（大多数具体类型优先），而交换机案例会按指定的顺序进行评估。

正在调试Xtend代码

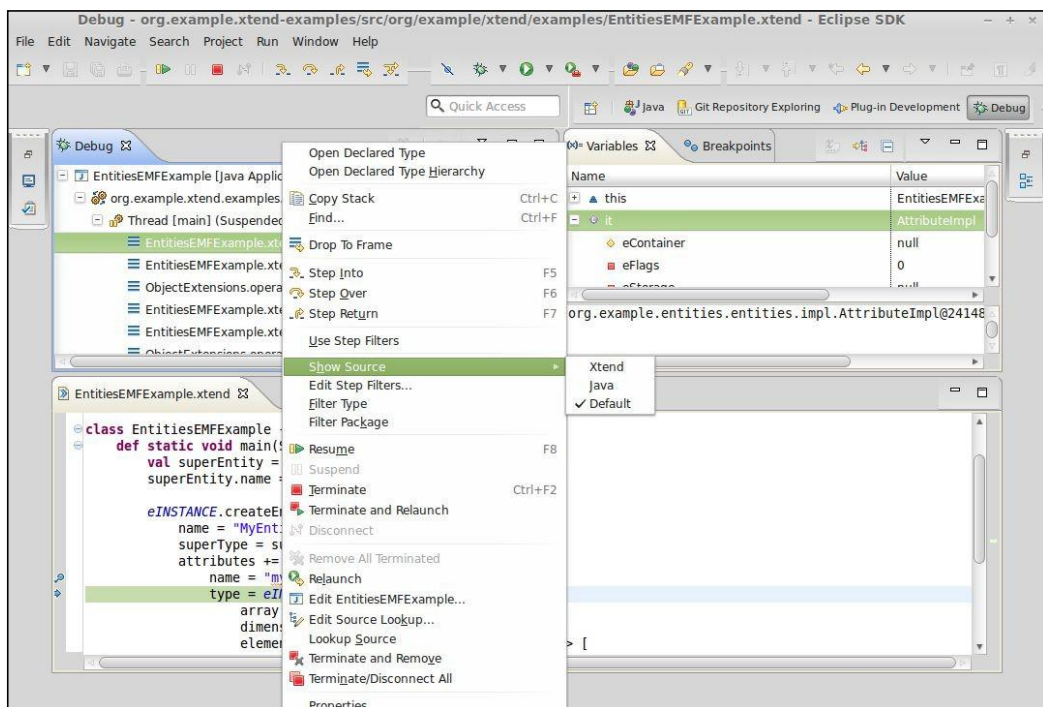
由Xtend生成的Java代码非常干净，易于调试。但是，由于Xtendile代码与EclipseJDT调试器的完全集成，也可以直接对Xtend代码(而不是生成的Java代码)进行调试。

这意味着您可以开始调试一个Java应用程序，它在某个时候调用Xtend生成的Java代码，然后自动完成它来调试原始的Xtend源代码。下一个是屏幕截图显示了Xtend代码的调试会话。请注意，所有JDT调试器视图都可用：此外，可以在变量视图中查看类似的隐式变量：



您还可以直接调试一个Xtend文件（例如，包含一个主方法），因为所有的运行和调试配置启动都可用于Xtend文件。断点可以通过双击来在Xtend文件中插入编辑器中的断点标尺（当前，调试上下文菜单不可用于Xtend文件）。

如果出于任何原因，在调试Xtend代码时需要调试生成的Java代码，则可以右键单击与Xtend文件行对应的元素上的调试视图并选择“显示源”（参见以下屏幕截图）。



产品简介

Xtend提供了许多功能，允许您编写干净和更可读的代码。由于它与Java完全集成，所有的Java库都可以从Xtend内部访问；此外，Xtend本身的IDE工具基本上与JDT的工具基本相同。基于上述所有原因，Xtext促进使用Xtend来开发DSL实现的所有方面。

在下一章中，我们将展示如何使用Xtext增强API使用EMF验证器机制实现DSL的约束检查。

4 验证

在本章中，我们将介绍验证的概念，特别是实现验证的Xtext机制：验证器。通过验证，您可以实现对在分析时无法执行的DSL的附加约束检查时间。Xtext允许您以一种简单和声明性的方式实现这种约束检查；此外，您只需要向Xtext通信可能出现的错误或警告，它将负责在IDE中生成相应的错误标记。在DSL的用户在编辑器中输入信息时，验证将在后台进行，以便立即提供反馈。我们还将展示如何实现与验证过程中生成的错误和警告对应的快速修复，以便帮助用户解决由于验证错误引起的问题。

在Xtext中进行的验证

正如我们在第1章实现DSL中所暗示的那样，解析程序只是编程语言实现的第一阶段。特别是，在解析过程中，不能总是确定程序的总体正确性。正在尝试使用

在语法规则中嵌入额外的约束检查可以使这样的规范更加复杂，或者它可能是不可能的，因为一些额外的静态分析只能在其他程序部分已经被解析的情况下执行。

实际上，最佳实践是在语法和验证中尽可能少地做（我们将在第9章类型检查和第10章范围界定中使用此实践）。这是因为可以提供更好的错误消息，并更准确地检测符合快速修复条件的问题。

验证机制将在本书的所有示例中广泛使用。通常，即使对于小型DSL，也必须实现验证器来执行额外的约束检查。

在Xtext中，这些约束检查使用验证器实现，验证器是从相应的EMF API继承的概念（见Steinberg等人，2008）。在EMF中，可以实现对EMF模型的元素执行约束检查的验证器。因为Xtext使用EMF来表示a的AST

解析程序后，验证器的机制自然扩展到Xtext DSL。特别是，Xtext通过提供一种声明性方式来指定DSL约束的规则，增强了用于验证的EMF API。此外，Xtext附带默认验证器，其中一些默认启用，以执行许多DSL常见的检查（例如，交叉引用检查）。自定义验证器可以由默认的Xtext验证器组成。

默认验证器

让我们回到第2章的实体DSL，创建您的第一个Xtext语言。由于我们在实体语法中表示了交叉引用，因此我们可以在生成的编辑器中看到Xtext链接器/验证器。如果我们输入了一个不正确的引用（例如，一个不存在的超级实体的名称），我们就会得到“无法解析对……的引用”的错误。对交叉参考的检查由Xtext提供的默认验证器执行（交叉参考解析是第10章范围的主要主题）。

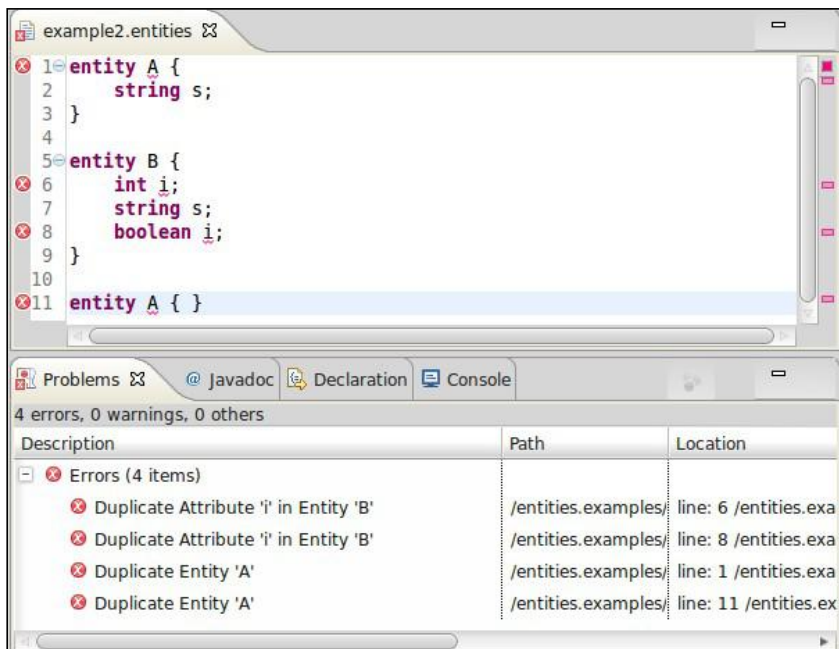
Xtext提供的另一个标准验证器是检查程序中名称是否唯一。此检查可验证每个元素类型的名称是否唯一；例如，您可以具有与实体名称相同的属性，但不能具有两个名称相同的实体。

默认情况下未启用唯一的名称验证器，但可以启用它通过修改MWE2文件，如下代码段所示。在我们的例子中，它是GenerateEntities.mwe2；特别是，我们需要取消注释有关名称统一验证器的组合检查规范：

```
片段=验证。验证片段自动注入{检查=
    "org.eclipse.xtext.validation.名称：产品统一验证器"
}
```

当然，在此之后，您需要运行MWE2工作流。

如果现在尝试在实体DSL编辑器中声明两个名称相同的实体，将会出现错误，如下屏幕截图所示：



值得注意的是，除了元素类型之外，此验证器还考虑了包含关系，例如，允许在两个不同实体中声明的两个属性具有相同的名称（从前面的屏幕截图中可以看到，A和B都具有属性，并且这是允许的）。



从技术上讲，所有可通过名称引用的东西都在命名空间中命名，这由包含关系所暗示。这导致了合格的名称，这将在第10章，范围界定中进行解释。

此验证器的默认行为应适用于大多数DSL。如果您的DSL需要要对名称或一般重复元素有更严格的约束，必须实现自定义名称AreUnique验证器类或简单地禁用名称AreUnique验证器并在自己的验证器中实现这些名称检查（自定义重复名称检查示例见第9章，类型检查）。

自定义验证器

虽然默认验证器可以执行一些常见的验证任务，但根据您希望DSL拥有的语义，必须由您执行对DSL的大多数检查。

这些附加检查可以使用Xtext在运行时插件项目的src文件夹中的验证子包中为您生成的Xtend类来实现。在我们的示例中，这个类被称为实体验证器。请记住，由于此类在src文件夹中，因此它不会被未来的MWE2 workflow执行所覆盖。Xtext通过调用带有@Check注释的每个方法，并将具有兼容运行时类型的所有实例传递给每个实例来执行验证

这样的方法。该方法的名称并不重要，但单个参数的类型却很重要。您可以为相同的类型定义尽可能多的带注释的方法；Xtext将全部调用它们。在这些方法中，您可以实现对该元素的语义检查。如果语义检查失败，您可以调用错误方法，这将很快得到解释。

例如，我们希望确保实体的层次结构中没有循环；因此，我们在验证器中编写了以下带注释的方法：

```

类实体验证器扩展抽象实体验证器{@Check
  def选中NoCycleIn实体层次结构（实体实体）{
    如果（entity.superType==为空）
      返回/没有任何需要检查的东西

    虚拟实体=<实体>newHashSet（）=<（实体）
    无功功率=entity.superType
    同时（当前！=null）{
      如果（visitedEntities.contains（电流））{
        错误（“实体 “+current.name+” 、EntitiesPackage：：
          eINSTANCE.entity_SuperType的层次结构中的循环”）
        返回
      }
      visitedEntities.add（电流）电流
      =current.superType
    }
  }
}

```

在前面一种方法中，我们通过记录我们正在访问的所有实体来遍历实体的层次结构（当然，如果实体没有超类型，就没有任何可检查）。如果在此访问中我们发现一个已经访问过的实体，这意味着层次结构包含一个循环，我们发出一个错误。在这种情况下，离开while循环是至关重要的，否则这个循环将永远不会结束（毕竟，我们找到了一个循环，我们将无休止地遍历这个层次结构）。

方法错误(我们参考Xtext文档)有许多重载版本。在本示例中，我们使用了需要以下内容的版本：

- 错误消息（由您提供有意义的信息）。
- 应报告错误的被检查的EObject的EMF特征，即应标记为错误。在此例中，包含错误的特征是超类型特征。

访问类和特性来自为元模型DSL生成的EPackage类（在我们的示例中，实体包）。使用此EPackage，可以通过两种方式获得EMF功能：

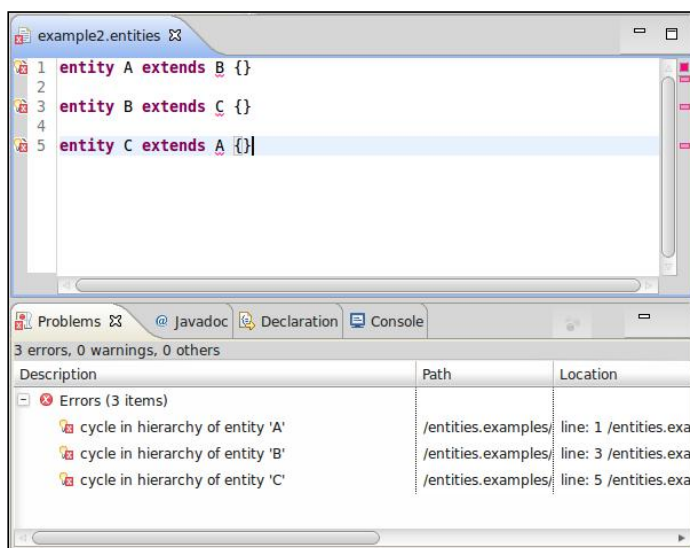
- 使用包的静态实例，然后使用与特征对应的方法（正如我们在前面的代码中所做的）：

```
EntitiesPackage : eINSTANCE.entity_SuperType.
```

- 使用内部接口的静态字段文字：实体包\$文字：：实体
SUPER_TYPE。

在这两种情况下，在Xtend程序中，您可以依赖内容帮助轻松选择功能。

我们现在可以通过定义，在实体DSL编辑器中尝试上面的验证检查在层次结构中包含循环的实体，如下屏幕截图所示：



您可以看到，编辑器中突出显示的元素是关键字扩展后的实体名称，因为这与特征超类型对应。

这三个错误标记还显示，Xtext为程序中实体类型的所有元素调用了我们的@Check注释方法。

使用适当的信息调用错误方法将允许Xtext管理基于Xtext的资源的标记（在重新分析之前清除它们，跟踪脏和保存状态，等等）。标记将出现在IDE中支持的任何地方：在左右编辑器标尺、“问题”视图和包资源管理器中。

如果要发出警告而不是错误（这被认为意味着模型无效），只需调用与错误方法具有相同签名的警告方法。例如，在我们的实体DSL中，我们遵循一个标准约定关于名称：实体的名称应以大写字母开头，而属性的名称应为小写。如果用户不遵守此约定，我们将发出警告；该程序都被认为是有效的。为了实现这一点，我们在实体验证器类中编写以下方法（注意使用从类导入的静态方法作为类的扩展方法）：

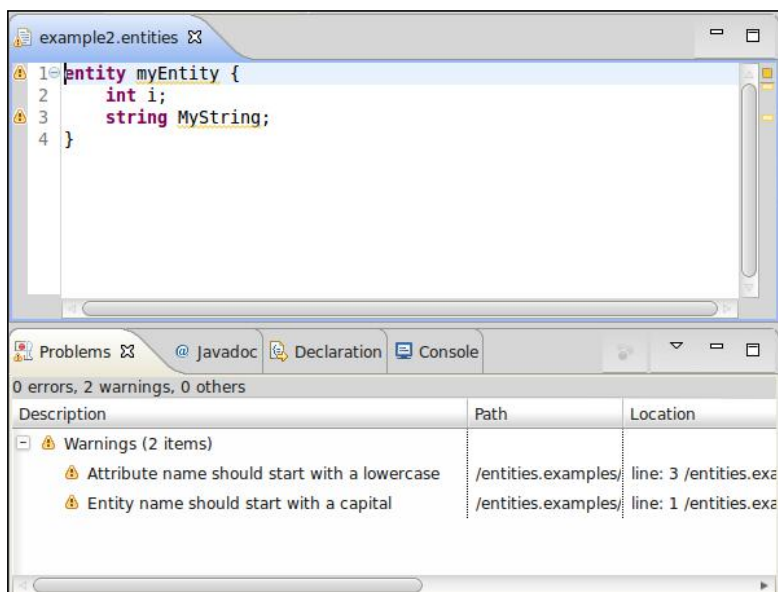
导入静态扩展名java.lang.Character.*

类实体验证器扩展了抽象实体验证器{

```
...
@检查一下
def检查实体名称与资本（实体实体）{
    如果(entity.name.charAt(0)。小写字母)
        警告（“实体名称应以大写开头”，EntitiesPackage: :
            eINSTANCE.entity_Name)
}

@检查一下
定义checkAttributeNameStartsWithLowercase(Attribute地址){
    如果(attr.name.charAt(0)。大写字母)
        警告（“属性名称应以小写开头”，EntitiesPackage: :
            eINSTANCE.attribute_Name)
}
}
```

以下屏幕截图显示了如何创建警告标记，而不是错误标记：



在Xtext2.4中，添加了“信息”严重级别（调用的方法是信息）。在这种情况下，信息标记只显示在编辑器的标尺中，而相应的文件没有被标记。

这只是一个简单验证器实现的示例；在书的其余部分中，我们将看到许多其他执行更复杂约束检查的实现（其中，类型检查，如第8章表达式语言和第9章类型检查所示）。

快速修复程序

正如我们在第1章实现DSL中所说的，快速修复是解决程序中问题的建议。快速修复通常通过错误标记提供的上下文菜单实现，它们在编辑器标尺和“问题”视图中都可用。



由于快速修复与验证紧密相关，我们将在本章中描述它们。此外，它们允许我们熟悉代表程序AST的EMF模型的操作。

在我们的实体DSL中，我们可以为验证器发布的每个警告和错误提供快速修复；此外，正如我们稍后将看到的，我们还可以提供由Xtext默认验证器发布的错误的快速修复。

Xtext提供了一个简单的机制来实现与验证器发出的错误或警告连接的快速修复。Xtext生成器生成一个Xtend存根类到Xtend插件项目；在我们的实体DSL示例中，这个类是org.example.entities.ui.quickfix. 实体快速修复程序提供程序。

快速修复程序由与错误或警告标记相关联的问题代码触发。问题代码只是一个唯一标识问题的字符串。因此，当调用错误或警告方法时，我们必须提供一个表示问题代码的附加参数。在验证器中，这通常是通过定义一个公共字符串常量，其值以DSL的包名称作为前缀，并以问题的合理名称结束。传递快速修复提供程序可以重用的额外问题数据也可能更有意义，以显示对快速修复的更有意义的描述，并实际修复程序。值得注意的是，当验证需要计算一些昂贵的东西时，问题数据非常有用；快速修复可能会避免再次计算它。因此，我们使用方法错误的另一个版本和警告，它需要四个参数，我们修改验证器如下（只显示修改的部分）：

类实体验证器扩展了抽象实体验证器{

公共静态

val HIERARCHY_CYCLE= “org.example.entities. 层次结构周期”；

公共静态val INVALID_ENTITY_NAME= “org.example.entities. 实体名称无效”；

公共静态val INVALID_ATTRIBUTE_NAME= “org.example.entities. 无效的属性名称”；

@检查一下

def选中NoCycleIn实体层次结构（实体实体）{

...

错误（“实体 “+current.name+” 层次结构中的循环、EntitiesPackage：：
eINSTANCE.entity_SuperType、HIERARCHY_CYCLE、
current.superType.name）

...

}

@检查一下

def检查实体名称与资本（实体实体）{

如果(entity.name.charAt(0)。小写字母)

```

        警告(“实体名称应以大写字母开头”，EntitiesPackage：：
            eINSTANCE.entity_Name, INVALID_ENTITY_NAME,
            entity.name)
    }

    @检查一下
    定义checkAttributeNameStartsWithLowercase(Attribute地址){
        如果(attr.name.charAt(0)。大写字母)
            警告(“属性名称应以小写开头”，EntitiesPackage：：
                eINSTANCE.attribute_Name, INVALID_ATTRIBUTE_NAME,
                attr.name)
    }
}

```

问题常量作为第三个参数传递给方法的错误和警告。问题数据是可选的，您可以传递可变数量的问题数据参数。为了实现一个快速修复，我们在实体的@修复提供程序中定义了一个方法，以及对这个快速修复适用的问题代码的引用。该方法的名称并不重要，但参数类型是固定的。

例如，对于关于实体名称的第一个字母必须是大写的警告，我们实现了一个快速修复程序，它会自动将该实体的第一个字母大写：

类实体快速修复提供程序扩展防御快速修复提供程序{@修复程序(实体验证器：：

```

INVALID_ENTITY_NAME)
无效的首字母(发行问题，
                                发行人决议书接受者接受者){
    acceptor.accept(问题，
        “大写第一个字母”，//标签“大写 ‘ “的第一个字母”
        +issue.data.获取(0) + “ ‘ ”，//描述
        “Entity.gif”，//图标
    [
        上下文|
        valxtext文档=context.xtextDocument
        val第一个字母=xtextDocument.get(issue.offset, 1);
        xtextDocument.replace(issue.offset, 1,
                                第一个字母。第一个上);
    ]
    );
}

```

让我们分析一下这段代码的作用。快速修复提供程序方法的第一个参数是表示错误信息的问题对象；这是由Xtext使用传递给验证器中的错误或警告的信息在内部构建的。第二个参数是一个接受体。接受器是一个概念，您将看到在Xtext中许多地方使用的不同类型的接受器；您通常只需要在接受器上调用所接受的方法，并传递一些参数。

传递给接受者接受方法的前三个参数是标签（在此修复程序的快速修复程序弹出窗口中显示）、说明（应显示选择此快速修复程序的效果意味着什么，或者使用户确信他们要应用的修复程序的东西）和图标（如果不需要图标，可以传递空字符串；如何在DSLUI中使用自定义图标）。注意，对于描述，我们使用问题数据的第一个元素（一个数组），因为我们知道在验证器中，我们传递了超类型的名称作为单一的问题数据（在实现快速修复时，您必须与验证器传递的信息一致）。

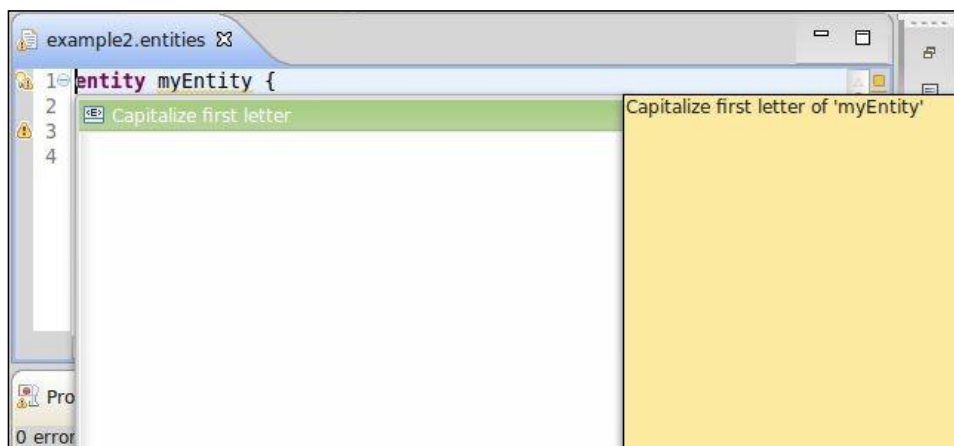
第四个参数是最有趣的一个，因为它实际上是由用户选择时由快速修复程序执行的修改代码。在Java中，第四个参数将是一个内部类，但在Xtend中，我们可以传递一个lambda。快速修复可以根据源文本（文本修改）或模型（语义修改）执行修正。下面两节进行解释。

文字修改

可以指定一个接受I修改上下文类型的单一参数。由于Xtend具有强大的类型推理机制，仅指定参数的名称(Xtend将推断其类型)就足够了。我们也可以没有指定任何参数，因为默认情况下，lambdas自动有一个命名为它的参数。在我们的示例中，为了明确地说明了参数的名称。

在前面的代码中，我们使用在给定的修改上下文中传递的IDocument参数来访问我们要在快速修复中修改的文本。我们已经得到了在问题对象中标记错误/警告的偏移量和长度。我们现在可以使用文档方法获取（偏移、长度）并替换()来执行第一个字母的大写。

这个快速修复程序显示在下面的屏幕截图中。



使用这种策略来实现快速修复有我们需要的缺点
处理编辑器的实际文本。

型号修改情况

另一种策略依赖于程序也可以作为EMF模型：如果我们修改模型，Xtext编辑器将自动更新其内容。因此，我们可以指定一个需要两个参数的lambda：

包含错误的EMF元素（即EObject）和修改
上下文。EObject元素是报告警告的元素。

例如，要取消大写属性的第一个字母，我们可以编写快速修复程序
使用以下策略：

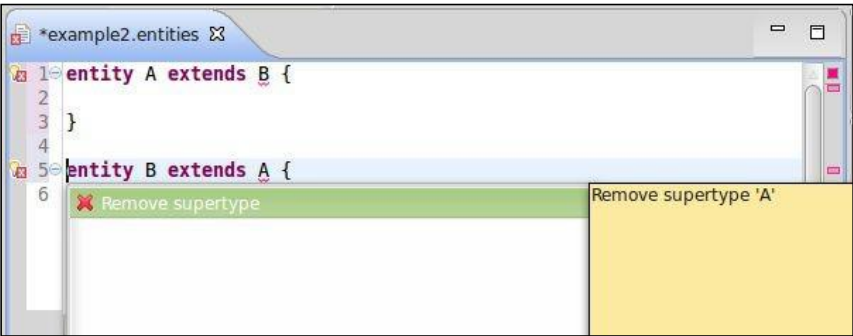
```
@Fix(EntitiesValidator: : INVALID_ATTRIBUTE_NAME)
无效非大写属性(发行问题,
                                发行人决议书接受者接受者) {
    acceptor.accept(问题,
        “不大写第一个字母”, //标签 “不大写 ‘ “的第
        一个字母”
        +issue.data.获取(0) + “ “”, //描述
        “Attribute.gif”, //图标[
        元素, 上下文|
        (元素作为Attribute)。name=issue.data.获取(0)。下下
    ]
    );
}
```

在这种情况下，元素是报告警告的属性对象。因此，我们只需简单地将固定的名称分配给属性的名称。请注意，使用此策略，我们只操作EMF模型，而无需处理编辑器的内容：然后，Xtext将负责更新编辑器的内容。

直接修改程序的EMF模型的能力变得更加复杂
快速修复程序更容易实现。例如，如果我们想实现快速修复程序
要删除层次结构中包含循环的实体的超类型，我们只需要将超类型特性设置为null，如以下代码段所示（您可以在以下屏幕截图中看到快速修复）：

```
@修复程序（实体验证器：： HIERARCHY_CYCLE）
超级类型（问题问题，

                                发行人决议书接受者接受者）{
    acceptor.accept（问题， “删
        除超类型”，
        ‘ ‘ ‘移除超类型 “issue.data.get（0）” ‘ ‘ ‘，
        “delete_obj.gif”，
        [元素，上下文|
            （元素为Entity）。superType=null;
        ]
    );
}
```



注意，语义变化导致模型元素中没有任何超类型，因此源文本扩展也被删除。通过修改程序的文本来实现同样的快速修复程序将需要更多的努力，而且将更容易出错。另一方面，文本修改允许修复语义模型中不存在的内容。例如，我们可以允许一些标点符号在语法中是可选的，以便提供更好的错误消息，如“缺少逗号”，而不是“语法错误”。这种宽大的行为使得解析器能够生成一个模型，并且有一些语义上的基础验证，而不是我们根本没有得到模型的语法错误。此外，语义更改总是包括格式设置，这可能会有其他不必要的副作用（我们将在第6章，自定义设置中处理格式设置）。

针对默认验证器的快速修复程序

如前所述，我们还可以提供由默认的Xtext验证器发布的错误的快速修复程序。您可能已经注意到，如果在实体DSL编辑器中引用了缺少的实体，Xtext已经提出了一些快速修复方法：如果源文件中存在其他实体，则建议更改对现有实体之一的引用。我们可以提供一个额外的快速修复程序，建议自动添加丢失的实体。为了做到这一点，我们必须在快速修复提供程序中为问题

`org.eclipse.xtext.diagnostics.Diagnostic.LINKING_`诊断定义一个方法。我们将缺失的实体添加到当前模型中；为了使事情更有趣，快速修复程序应该在引用缺失实体的实体后面添加缺失的实体。例如，请考虑以下源文件：

```
实体，我的第一个实体；
```

```
    一；
}
```

```
实体，我的其他实体{
}
```

属性定义中引用的FooBar实体没有在程序中定义，我们想在第一实体的定义之后（在其他实体之前）添加它，因为这是包含引用缺失实体的属性定义的实体。

让我们首先展示这个快速修复的代码（我们将使用Xtend特性，设置器，模板字符串，和与操作符，使我们的逻辑更紧凑）：

```
导入静态扩展名org.eclipse.xtext。生态生存周期2。*@修复程序（诊断
```

```
程序：：LINKING_DIAGNOSTIC）
```

无效实体(问题问题,

发行人决议书接受者接受者) {

acceptor.accept(问题, “创建缺

失实体”, “创建缺失实体”,

“Entity.gif”,

[元素, 上下文|

当前有效值实体=元素。获取类型的容器(类型(实体))

瓦尔模型=currentEntity.eContainer作为模型

model.entities.add(

model.entities.indexOf(当前实体)+1,

EntitiesFactory: : eINSTANCE.createEntity()=>[

名称: =context.xtextDocument.get(issue.offset,

issue.length)

]

)

]

);

}

考虑传递给lambda的EObject元素是引用缺失实体的程序元素, 因此, 它不一定是实体(例如, 如果缺少的实体在属性的类型规范中, 如前面的实体程序段, 那么EObject元素是属性类型)。为了得到包含实体, 我们可以提升EMF模型的包含关系

直到我们到达一个实体元素。或者, 我们可以使用许多静态实用程序方法之一, 在这里作为扩展方法导入, 由EcoreUtil2类中的Xtext提供(这是EMF的标准EcoreUtil类的补充)。特别地, 我们使用get容器类型, 它在容器关系中继续前进, 直到它找到指定类型的元素。为了检索根模型元素, 我们可以简单地强制转换找到的实体的容器(因为在我们的实体DSL中, 实体只能包含在模型中)。然后, 将新创建的实体插入所需位置(即当前实体位置之后)。

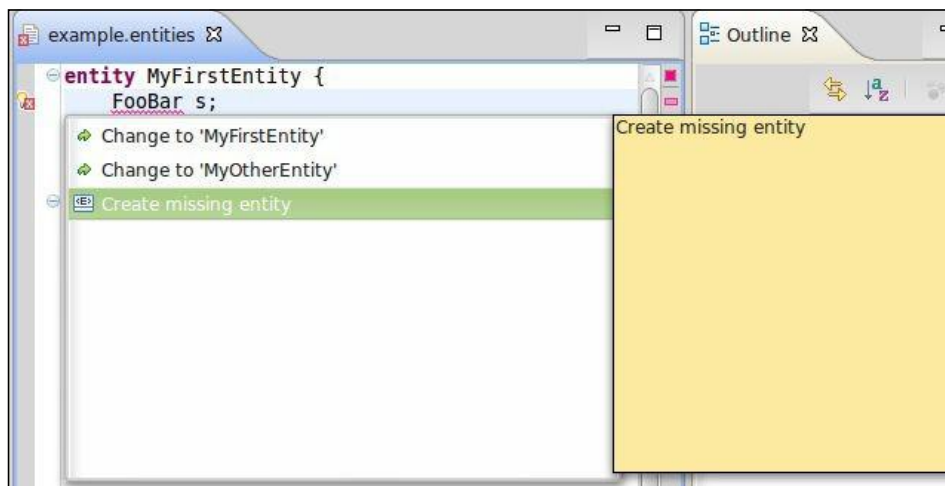


花点时间看看EcoreUtil和EcoreUtil2两个类, 因为它们提供了处理EMF模型时需要的许多有用方法。



若要创建缺少的实体，我们必须知道它的名称。对于此问题（它不是由我们自己的验证器生成的），问题数据不包含关于缺少的元素名称的任何信息。但是，问题偏移量告诉我们文档中缺少元素的名称在哪个位置。因此，可以使用编辑器文档中的此偏移量（长度也包含在问题中）来检索缺少的元素的名称。

现在可以检查此快速修复的作用（参见下面的截图；这还显示了Xtext提供的默认快速修复，建议将引用实体的名称更改为当前源中可用实体的名称）。



产品简介

在本章中，您学习如何使用基于@Check注释方法的Xtext验证器机制实现约束检查。只要通过实现自定义验证器并使用适当的信息调用方法错误或警告，Xtext就会产生错误和警告标记，导致标记文本区域，并在Eclipse中的各种视图中显示标记。

我们还展示了如何实现快速修复程序。由于Xtext会自动将DSL编辑器的内容与AST的EMF模型同步，因此我们可以简单地修改该模型，而无需处理程序的文本表示。

在下一章中，我们将为在Xtend中实现的实体DSL编写一个代码生成器，并依赖其高级特性进行代码生成：从用实体DSL编写的程序开始，我们将生成相应的Java代码。您将看到Xtext会自动将代码生成器集成到Eclipse的构建基础架构中。

5

代码生成

在本章中，您将学习如何使用Xtend编程语言为XtextDSL编写代码生成器。使用我们在前一章中开发的实体DSL，我们将编写一个代码生成器，它将为每个实体声明生成相应的Java类。我们还将看到代码生成器如何通过Xtext自动集成到月蚀构建器基础架构中。最后，DSL实现可以导出为Java独立的命令行编译器。

代码生成的简介

在DSL中编写的程序经过解析和验证后，您可能需要使用解析的EMF模型，即该程序的AST。通常，您可能希望用其他语言生成代码（例如，Java代码）、配置文件、XML、文本文件等等。在所有这些情况下，您都需要编写一个代码生成器。

由于解析程序是EMF模型，您确实可以使用任何处理代码生成的EMF框架。当然，您也可以使用普通Java来生成代码，毕竟，正如我们在前面的章节中看到的，您拥有可以访问EMF模型的所有JavaAPI。

然而，在这本书中，我们将使用Xtend(在第三章Xtend编程语言中介绍)来编写代码生成器，因为它非常适合这个任务。此外，Xtext会自动将代码生成器集成到Eclipse构建基础设施中，这也大大简化了编写代码生成器的任务。我们所要处理的就是生成所需的输出（例如，Java源代码、XML等等）。

在Xtend中编写代码生成器

Xtext会自动创建一个生成器存根。在我们的示例中，存根是在程序包org.example.entities.generator: 中创建的

```
类实体生成器实现了I生成器{
    覆盖空白o生成(资源资源, IFileSystemAccessfsa) {
        //TODO实现了我
    }
}
```

在编写实际代码之前，让我们记住Xtext是一个框架，因此，整个控制流是由框架而不是程序员决定的（这也被称为好莱坞原则：“不要叫我们，我们会叫你”）。这意味着您不必运行生成器；DSLXtext编辑器已经集成在Eclipse的自动构建基础结构中，并且

当写入DSL的源代码发生变化时，生成器将被自动调用（事实上，如果它的一个依赖关系发生了变化，它也会被调用，我们将在后面的章节中看到）。请注意，必须实现的方法只接受一个EMF资源，其中包含程序的EMF模型表示。这意味着如果调用此生成方法，则相应的源程序已被解析和验证：只有在源程序没有任何验证错误时才会被调用。您甚至不必担心将生成代码的物理基本路径位置：它隐藏在传递的IFileSystemAccess文件中。您只需要指定创建生成文件的相对路径及其内容作为字符串（实际上，它可以是任何java.lang.CharSequence）。

首先，我们需要决定我们想要从DSL程序中生成什么。对于实体DSL，为每个实体生成一个Java类可能是有意义的。因此，我们需要检索所传递的资源中的所有实体对象。下面是如何使用一行Xtend代码来实现这一点：

```
resource.allContents. 可更换的材料.过滤器（类型（实体））
```

然后，我们遍历这些实体，并使用实体的名称为每个实体生成一个Java文件。由于我们的DSL中没有显式的包，所以我们选择在包实体中生成所有的Java类。因此，Java文件路径将为：

```
实体/ “+entity.name+” 。java “
```

请记住，我们不必关心基本路径（它将被配置为已传递的IFileSystemAccess文件）。总结一下，我们将得到：

```
覆盖空白o生成(资源资源, IFileSystemAccessfsa) {
  为(e: res.allContents。可更换的材料。过滤器（类型（实体））) {fsa.generateFile (
    “实体/ “+e.name+” 。java”，即编译)
  }
}
```

现在我们必须实现编译方法，它必须返回一个字符串和将存储在生成的文件中的内容。我们使用Xtend多线模板表达式来实现这种方法；该方法见如下截图：

```
def compile(Entity entity) {
  ...
  package entities;

  public class «entity.name» «IF entity.superType != null»extends «entity.superType.name» «ENDIF»{
    «FOR attribute : entity.attributes»
    private «attribute.type.compile» «attribute.name»;
    «ENDFOR»

    «FOR attribute : entity.attributes»
    public «attribute.type.compile» get«attribute.name.toFirstUpper»() {
      return «attribute.name»;
    }

    public void set«attribute.name.toFirstUpper»(«attribute.type.compile» _arg) {
      this.«attribute.name» = _arg;
    }

    «ENDFOR»
  }
  ...
}
```

请注意，Xtend编辑器还会显示最终结果字符串的选项卡缩进。Xtend巧妙地忽略了循环构造的缩进，这只会使Xtend代码的可读性；这些缩进和换行符不会成为结果字符串的一部分。

我们基本上使用存储在实体对象中的信息为Java类编写一个模板。只有在实体具有超类型时，我们才会生成扩展部分。这是使用模板中的IF条件来实现的。然后，我们使用模板循环构造遍历实体的属性两次：第一次生成Java字段，第二次生成获取器和设置器方法。在两个单独的迭代中这样做允许我们生成Java类开始时的所有字段。我们使用第一个上扩展方法来正确地生成获取器和设置器方法的名称。



请注意，IF和FOR（带大写字母）分别用于在模板表达式中指定条件和循环。

我们将属性类型的编译委托给另一种方法：

```
def编译（属性类型、属性类型）
  {attributeType.elementType.typeToString+
    如果(attributeType.array) “[ ]”
    其他的 “”
  }

def调度类型到字符串（基本类型类型）
  {if(type.typeName== “string”) “String” 其他
   type.typeName
  }

def调度类型到字符串（实体类型类型）{type.entity.name
}
```

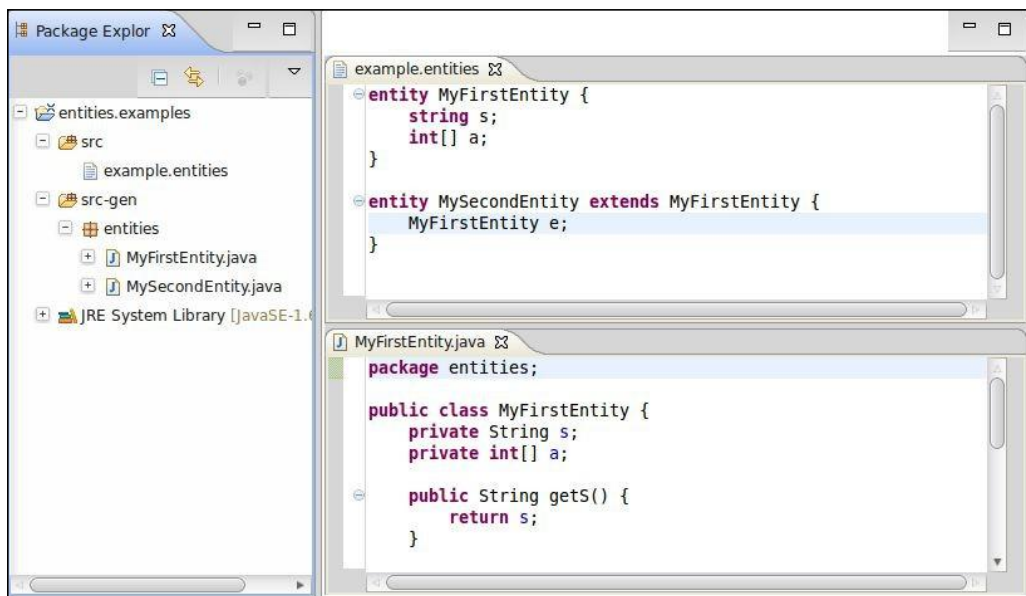
注意到我们的DSL的基本类型文本已经对应于Java的基本类型；唯一的例外是“字符串”，它在Java中对应于“字符串”。为了简化这个示例，我们在代码生成期间不考虑属性类型的长度特性。

与日蚀月食构建机制的集成

是时候看到我们的生成器的操作了：启动Eclipse，在工作区中创建一个Java项目，然后在src文件夹中创建一个新的。entuns文件（记住接受将Xtext特性添加到项目，否则生成器将不会运行）。继续添加一个或多个具有某些属性的实体。请注意，保存文件后就会自动创建src-gen文件夹。此时，您还应该通过将“生成路径|用作源文件夹”，将此生成的文件夹添加到项目源文件夹中。探索src-gen文件夹的内容，将找到。entes文件中每个实体生成的Java类。您可以在以下屏幕截图中看到一个示例。

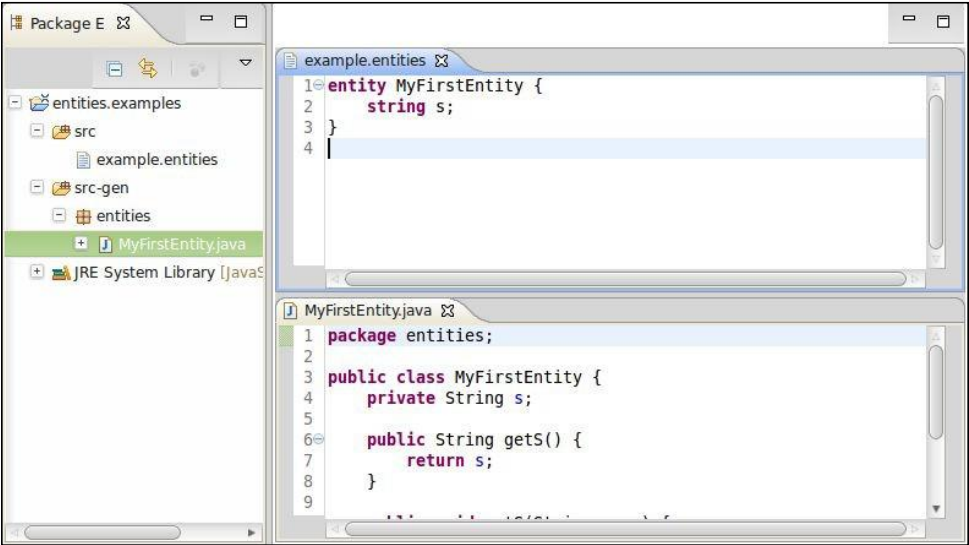


请注意，一个代码生成器只是在创建文本。其他组件必须理解这一点，例如，Java编译器。这就是为什么我们需要将src-gen文件夹添加到项目源文件夹中：通过这种方式，EclipseJava编译器会自动编译生成的Java源代码。



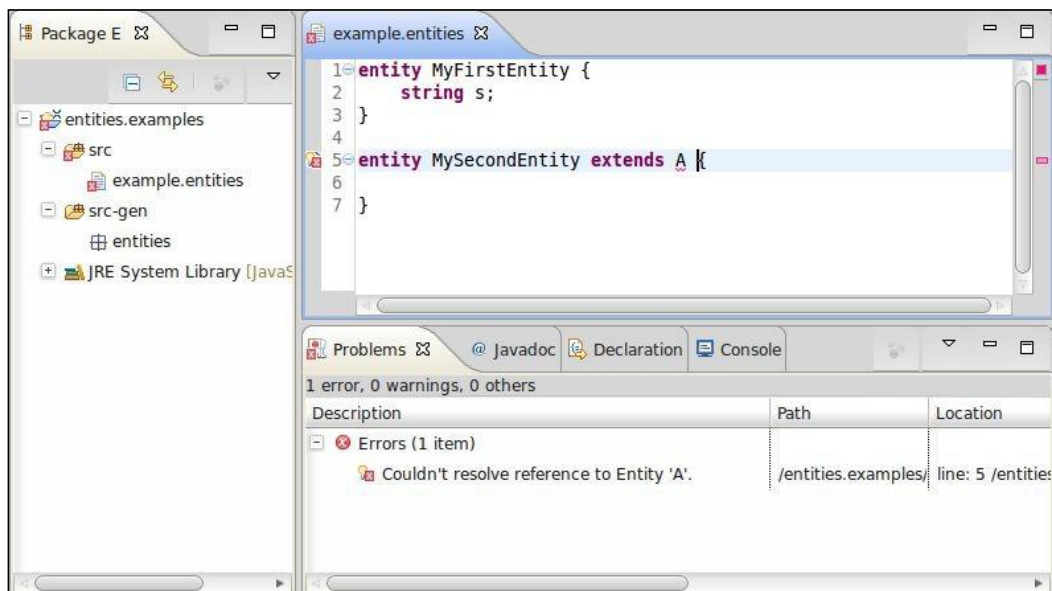
进行一些更改，并注意在保存更改时会重新生成Java文件。删除实体并观察相应的Java文件已被删除。您的DSL生成器已完全集成在蚀月蚀构建系统中。

例如，在下面的屏幕截图中，我们可以看到从第一实体删除字段后生成的Java文件是如何更改的，以及以前是如何更改的
删除后生成的Java文件将自动删除
输入文件中相应的实体定义：



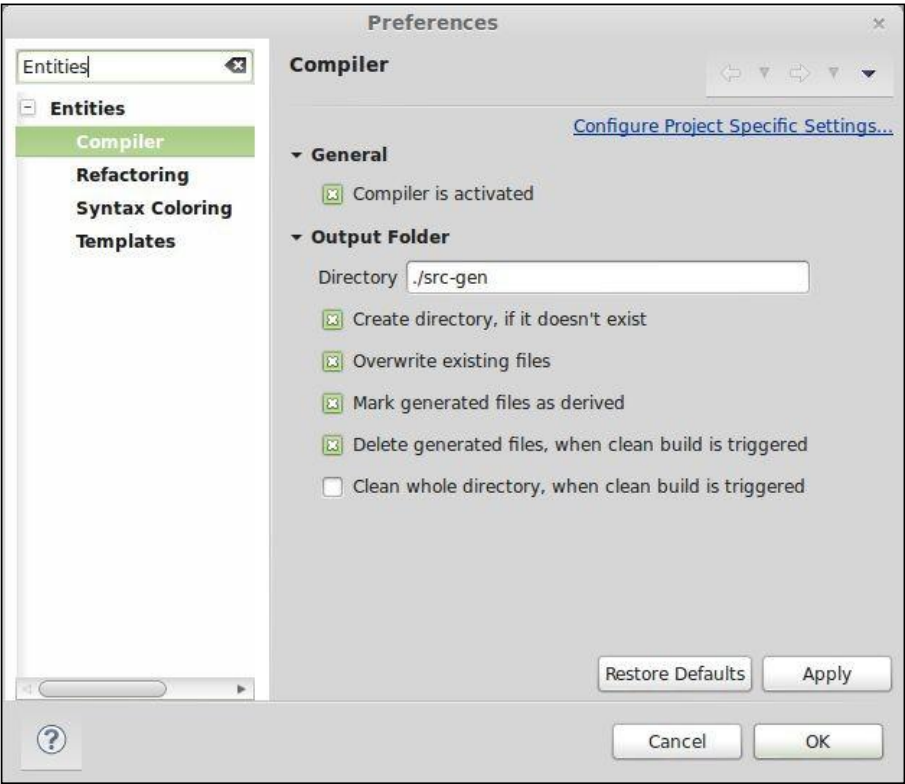
正如我们前面所暗示的，当输入文件更改时，只要该文件不包含验证错误，Xtext将自动调用生成器。Xtext还会自动跟踪生成的文件和原始输入文件之间的关联(如在本例中，我们可以从一个输入文件生成几个输出文件)。当发现DSL源文件无效(即存在验证错误)时，从该文件生成的一切都将自动删除，与包含错误的输入程序元素无关。

例如，在下面的屏幕截图中，我们显示如果我们修改example.entities文件引入错误会发生什么：即使错误与我的第一个实体无关，它相应生成的Java文件仍然被删除：



与Eclipse构建基础设施的集成是可定制的，但是默认行为，包括自动删除与无效源文件对应的生成文件，应该适合大多数情况。事实上，代码生成器往往是为完整和有效而编写的并且通常会为不完整的模型抛出异常或产生完整的垃圾。

在运行MWE2工作流时，Xtext还会为DSL创建首选项页。其中一个首选项页面与代码生成有关。在新的Eclipse实例中，您可以检查创建的Xtext（转到Window|首选项）：您将看到实体DSL的专用部分具有典型配置（例如，语法突出显示颜色和字体以及代码生成首选项），请参阅以下屏幕截图：



独立的命令行编译器

我们已经知道Xtext项目向导为DSL创建了项目，分离了单独项目（.ui项目）中与用户界面相关的特性；运行时项目不依赖于Eclipse用户界面。因此，我们可以创建一个由简单Java类组成的命令行应用程序。只需将以下各行添加到MWE2工作流文件中：

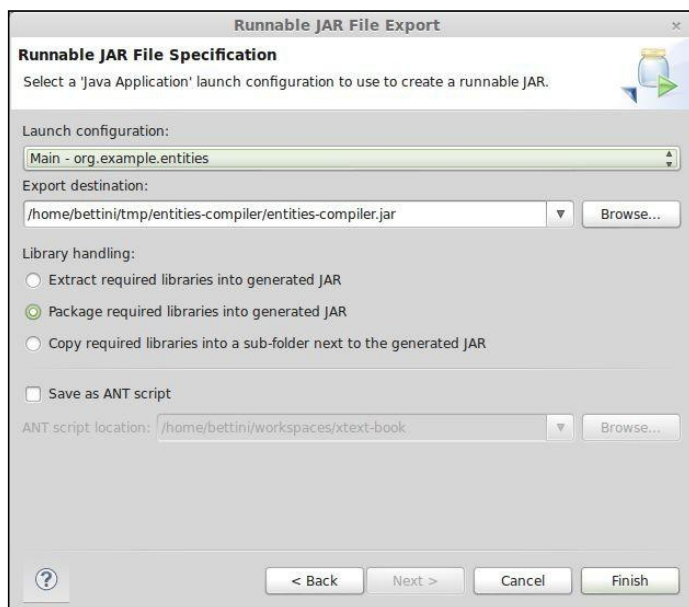
```
//生成器API
片段=生成器。生成器片段{生成JavaMain=true
}
```

如果现在运行工作流，您将在org.example.entities.generator包中的项目的src文件夹中找到一个主类。正如您从第2章“创建第一个Xtext语言”中回忆起的那样，生成到src文件夹中的文件只生成一次（如果它们不存在），因此您可以安全地添加/修改主类的逻辑。这此时不是必需的；我们将在生成时使用类。你不必担心不理解主班在这里所做的所有事情，这将在后面的章节中被揭示。

现在，只要知道生成的主类的主方法接受它就足够了
要为其进行分析、验证和生成代码的文件的命令行参数。

最后，您还可以为独立的实体编译器导出一个JAR文件；
在月蚀中，以下步骤给出了一种简单的方法：

1. 运行主机。将Java文件作为Java应用程序（右键单击并选择“运行”|Java应用程序）。在控制台视图中，您可以看到应用程序以错误“中止：没有提供EMF资源的路径！”因为您没有指定任何命令行参数（但这不是我们创建此启动配置的原因）。
2. 从文件菜单中选择导出。。|Java|可运行的JAR文件，然后单击“下一步”。
3. 选择在步骤1中创建的启动配置，指定导出的JAR文件（例如，entities-compiler.jar）的路径，然后在库处理部分中，选择将必需的库打包到生成的JAR中
（请参见下面的屏幕截图），然后单击“完成”。



生成的JAR文件可以在由输出路径表示的目录中找到。请注意，此JAR文件相当大（近20MB），因为除了项目的类文件外，它还包含所有所需的JAR（例如，Xtext和EMFJAR文件）。这意味着您的JAR文件是独立的，不需要任何进一步的库。现在可以从命令行尝试独立的编译器，只需进入创建jar文件的目录，使用Java运行它，给出.etuts文件的路径：

指向。antines文件>的java-jarentities-compiler.jar<路径

如果给定文件包含错误，您将看到结果报告的错误；否则，应看到“代码生成完成”消息。在后一种情况下，您将在src-gen文件夹中找到所有生成的Java文件。



这演示了Xtext可以生成一个独立的基于命令行的编译器。

产品简介

从DSL源生成代码是开发DSL时的典型任务。Xtend提供了许多有趣的特性，这使得编写代码生成器变得非常容易。

Xtext会自动将代码生成器集成到月蚀构建基础架构中，以便构建能够在文件保存时逐步进行，就像在EclipseJDT中一样。您还可以获得一个命令行独立编译器，以便您的DSL程序可以不用Eclipse地编译。

在下一章中，您将了解依赖性注入框架谷歌Guice，Xtext严重依赖于它来定制它的所有特性。特别是，您还将看到如何为DSL自定义运行时和IDE概念。

6

自定义设置

在本章中，我们将描述定制Xtext组件的主要机制：谷歌Guice，一个依赖性注入框架。使用谷歌Guice，我们可以轻松地、一致地将特定组件的自定义实现注入到Xtext中。在第一部分中，我们将简要地展示一些使用谷歌吉他的Java示例。然后，我们将展示Xtext如何使用这个依赖关系注入框架。特别是，您将学习如何自定义运行时和UI方面。

依赖性注入

依赖性注入模式（Fowler，2004）用于以一致的方式将实现类注入到类层次结构中。当类将特定任务委托给其他类时，这很有用：消息被简单地转发到字段中引用的对象（抽象实际行为）。

让我们考虑一个可能的场景：一个从处理器类和日志器类的实际实现中抽象的服务类(尽管在本节中我们显示Java代码，所有注入原则自然也适用于Xtend)。以下是一个可能的实现方式：

```
公开课服务{
```

```
    私人日志记录器，私人日志记录器；
```

```
    专用处理器处理器；
```

```
    公共无效执行（字符串命令）{logger.log（“执  
        行”+命令）； processor.process（命令）；  
        logger.log（“执行”+命令）；
```

```
    }
```

```
}
```

公开课日志记录员{

```
    公共空白日志（字符串消息）{系统.out.println  
        （“日志：”+消息）；  
    }  
}
```

公共接口处理器{

```
    公共无效程序（对象o）；  
}
```

公共类处理器实现处理器{

```
    私人日志记录器，私人日志记录器；  
  
    公共无效进程(对象o){logger.log（“处理”）；  
        System.out.println（“处理”+o+”...”）；  
    }  
}
```

这些类正确地实现细节中抽象出来，但是正确地初始化字段的问题仍然存在。如果我们初始化构造函数中的字段，那么用户仍然需要硬编码实际的实现类名。还要注意，日志记录器在两个独立的类中使用，因此，如果我们有一个自定义的日志记录器，我们必须确保所有的实例都使用正确的实例。

这些问题可以通过使用依赖关系注入来解决这些问题。使用依赖项注入，硬编码的依赖项将被删除。此外，我们将能够轻松地、一致地在整个代码中切换实现类。虽然同样的目标也可以通过实现工厂方法或抽象的工厂模式来手动实现 (Gamma等人, 1995)，但是使用依赖注入框架，更容易保持所需的一致性，程序员需要编写更少的代码。Xtext使用了依赖性注入框架谷歌Guice，<http://code.google.com/p/google-guice/>。我们参考这个框架提供的所有功能参考谷歌Guice文档；在本节中，我们只是简要描述了它的主要特性。

您可以用@Inject注释来注释您希望Guice注入的字段。google.inject.Inject)：

公开课服务{

```
    输入专用处理器处理器；  
  
    公共无效执行（字符串命令）{logger.log（“执  
        行”+命令）； processor.process（命令）；
```

```

        logger.log ( “执行” +命令 );
    }
}

```

公共类处理器实现处理器输入专用日志器记录器：

```

公共无效进程 (对象o) {logger.log ( “处理” ); 系统。
    out.println( “处理” +o+ ” ... ” );
}
}

```

从注入请求到实例的映射是在Guice模块中指定的，该模块是从抽象模块派生出来的类。方法配置实现为使用简单直观的API指定绑定。

您只需要为接口、抽象类和自定义类指定绑定。这意味着您不需要为日志记录器指定一个绑定，因为它是一个具体的类；相反，您需要为接口指定一个绑定处理器。以下是我们场景的Guice模块示例：

```

公共类标准模块扩展了抽象模块
    受保护的无效配置 () {bind(处理器.class)。to(处理器输入。类)；
    }
}

```

通过传递模块，可以使用静态方法Guice.createInjector创建喷射器。然后，您可以使用喷射器来创建实例：

```

喷射器=Guice.createInjector (新的备用模块 ())；维修服务=injector.getInstance (服务。类)；service.execute ( “第一个命令” )；

```

注入字段的初始化将由谷歌Guice自动完成。

值得注意的是，该框架还能够初始化（注入）私有字段

（就像在我们的例子中那样）。使用依赖项注入的类的实例只能通过注入器创建；创建新实例不会触发注入，因此用@Inject注释的所有字段都将为空。



在使用Xtext实现DSL时，您将永远不必手动创建一个新的喷油器。事实上，Xtext生成实用程序类以轻松获得注入器，例如，在使用JUnit测试DSL时，我们将在第7章，测试中看到。有关更多细节，我们也可以参考2012年的科恩莱因。本节所示的例子旨在介绍谷歌Guice的主要功能。

如果我们需要不同的绑定配置，我们需要只定义另一个模块。例如，让我们假设我们已经为日志记录和处理定义了额外的派生实现。下面是一个示例，其中记录器和处理器被绑定到自定义实现：

```
公共类定制模块扩展抽象模块{@Override
    受保护的空白配置() {bind(记录器.class).to(定制记录器.类); 绑定(处理
        器).class).to(高级处理器).类);
    }
}
```

使用使用此模块获得的喷油器创建实例将确保始终使用正确的类（例如，定制记录器类将由服务和处理器同时使用）。

您可以从同一应用程序中的不同注入器创建实例，例如：

```
executeService(Guice.createInjector(新标准模块())));
executeService(Guice.createInjector(新定制模块())));

void执行服务(喷油器){
    服务服务=injector.getInstance(服务.类); service.execute(“第一个命令”); service.execute(“第二个命令”);
}
```



可以以许多不同的方式请求注入，例如向构造函数注入参数、使用命名实例、接口的默认实现的规范、setter方法等等。在这本书中，我们将主要使用注入的字段。

当类实例化时，注入字段仅实例化一次。在某些情况下，这并不理想；您可能需要决定何时需要实例化元素，或者可能需要从方法体中创建多个实例。在这种情况下，您可以注入一个`com.google.inject`，而不是注入需要的类型C的实例。提供程序<C>实例，它有一个生成getC的实例。

例如：

公共类日志记录器

专用提供程序<实用程序>实用程序提供程序；

```
公共无效日志（字符串消息）{System.out.println（“LOG： ” +
    消息+ “-” +
    utilityProvider.get().m（））；
}
```

每次我们使用注入的提供程序类创建一个新的实用程序实例时。好处是，要注入实用工具的自定义实现，您不需要提供自定义提供程序：您只需在Guice模块中绑定实用工具类，一切都将按预期工作：

公共类定制模块扩展抽象模块{@Override

```
受保护的空白配置() {bind(记录器.class).to(定制记录器.类)；绑定(处理器
    器).class).to(高级处理器).类)；绑定(实用工具.class).to(自定义
    实用程序.类)；
}
```



必须记住的是，一旦类依赖于注入，它们的实例就只能通过注入器创建，否则所有注入的元素都将为空。一般来说，一旦在框架中使用了依赖性注入，该框架的所有类都必须依赖于注入。

Xtext中的谷歌家伙

所有的Xtext组件都依赖于谷歌Guice依赖注入，甚至是Xtext为DSL生成的类。这意味着在你的课堂上，如果你需要的话要使用Xtext中的类，只需声明这种类型的字段@输入对象的注释。

注入机制允许DSL开发人员自定义Xtext框架的每个组件。

在运行MWE2工作流时，Xtext会同时生成一个完全配置的模块以及从生成的模块中派生出来的空模块。自定义设置将被添加到空存根模块中。不应触摸生成的模块。Xtext生成一个运行时模块，定义相关的界面部分的配置和一个特定于在EclipseIDE中使用的配置。Guice提供了一种组成Xtext使用的模块的机制：UI项目中的模块使用运行时项目中的模块并覆盖一些绑定。

让我们考虑一下实体DSL示例；您可以在运行时项目的src目录中找到类实体运行时模块，它从src-gen目录中继承了抽象实体运行时模块。同样地，在UI项目中，您可以在src目录中找到实体UiMidile类，它继承自src-gen目录中的抽象实体UiModule。

src-gen中的Guice模块已经配置了在MWE2工作流中生成的存根类的绑定。因此，如果要使用存根类自定义方面，则不必指定任何特定的绑定。生成的存根类涉及到程序员通常需要的典型方面

例如，在运行时项目中自定义验证和生成（如我们在前面的章节中看到的），以及UI项目中的标签和大纲（如我们将在下一节中看到）。如果您需要自定义未被任何生成的存根类覆盖的方面，那么您将需要自己编写一个类，然后在src文件夹中的Guice模块中指定类的绑定。

我们将在“其他自定义”部分中看到此场景的一个示例。

通过实现配置方法，如我们在上一节中看到的那样，可以指定这些Guice模块类中的绑定。但是，Xtext提供了一个定义绑定的增强API：Xtext反射性地搜索具有特定签名的方法，以找到Guice绑定。因此，假设要将基类绑定到派生的自定义类，而不是编写以下内容：

公共类实体运行时模块扩展...

受保护的配置() {bind(基础类.class)。to(自定义类.类)；

...

}


您可以仅在模块中定义一个具有特定签名的方法，如下所示：

```
公共类<吗? 扩展基类>双基类() {
    返回自定义类。类别
}
```

请记住，这些方法被反射地调用，因此它们的签名必须遵循预期的约定。我们参考模块API的完整描述参考官方Xtext文档；通常，您将在本书中看到的绑定方法将具有前面的形状，特别是，方法的名称必须以“绑定”开头，后面是我们要提供绑定的类或接口的名称。

重要的是要理解这些绑定方法不一定需要覆盖模块基类中的方法；您还可以使自己的类（与Xtext框架类无关）成为此注入机制的参与者（只要您遵循前面关于方法签名的约定）。一个重要的注意事项是，在超级模块类中定义的绑定必须被具有相同签名的方法覆盖，否则结果是错误地尝试提供同一类/接口的多个/冲突绑定。

在本章的其余部分中，我们将展示IDE和运行时概念的定制示例。对于大多数这些自定义设置，我们将修改在运行MWE2工作流时Xtext生成的相应的Xtend存根类。正如前面所暗示的，在这些情况下，我们将不需要编写一个自定义的吉他绑定。我们还将展示一个没有自动生成的存根类的自定义示例。

 从2.4.0版本开始，由Xtext生成的所有存根类都是Xtend类。唯一的例外是Guice模块类，它们是Java类。

IDE概念的自定义

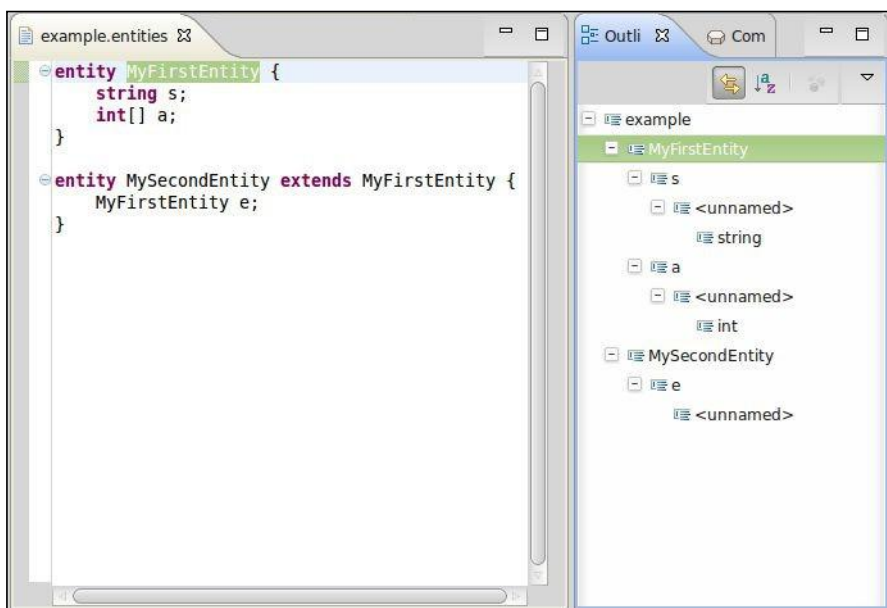
在本节中，我们将展示您可能要自定义的DSL的IDE的典型概念。Xtext也显示了它在此上下文中的可用性，因为，正如您将看到的，它减少了定制工作。

标签

XtextUI类分别使用ILabel提供程序接口，通过其getText和getImage的方法获取文本标签和图标。ILabel提供程序是基于月食JFace的查看器的一个标准组件。您可以在“大纲”视图和内容帮助提案弹出窗口中看到ILabelProvider界面（以及其他许多地方）。

Xtext为所有DSL提供了标签提供程序的默认实现，如果在相应的对象类中找到，则最好使用名称功能生成EMF模型对象的合理表示。

在编辑实体文件时，您可以在“大纲”视图中看到，请参阅屏幕截图如下：



但是，您肯定要自定义DSL的某些元素的表示。

可以在UI中找到DSL的标签提供商根类

子程序包 `ui.labeling` 中的插件项目。此根类扩展了基类 `DefaultEObjectLabel` 提供程序。在实体DSL中，该类称为实体标签提供程序。

这个类使用了一个多态调度器机制(类似于第3章Xtend编程语言中描述的Xtend的调度方法)，这也在Xtext中的许多其他地方使用。因此，与其实现getText和getImage方法，您可以简单地定义几种版本的文本和图像获取方法，并将您想要提供的表示类型的EObject对象作为参数。然后，Xtext将根据要表示的元素的运行时类型来搜索此类方法。

例如，对于我们的实体DSL，我们可以更改属性的文本表示，以便显示它们的名称和更好地表示类型(例如，“名称: type”)。然后，我们定义了一个方法文本，将属性作为参数，并返回一个字符串：

类实体标签提供程序扩展... {@Inject扩展类型表示

```
def文本(属性a) {a.name+
    如果(a.type!=空)
        “: ”+a.type.representation
    其他的 “”
}
```

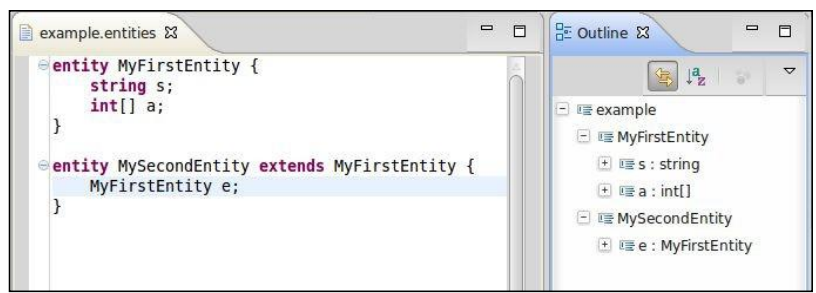
为了获得属性类型元素的表示，我们使用注入的扩展Type表示(它类似于我们在第5章代码生成器中生成属性类型生成Java代码的方式)，特别是它的方法表示。



请记住，标签提供程序例如用于大纲视图，它在编辑器内容更改时刷新，其内容可能包含错误；因此，您必须准备好处理不完整的模型，而某些功能可能仍为空。这就是为什么在访问这些特性之前，您应该始终检查它们是否为空。

请注意，我们注入类型类型的扩展字段，而不是在字段声明中创建一个新的实例。虽然没有严格需要为这个类使用注入，但我们决定依赖它，因为在将来，我们可能希望能够为该类提供一个不同的实现。使用注入而不是新注入的另一点是，另一个类可能依赖于注入（或者它在将来也可以这样做）。使用注入可以为未来和非预期的定制开门。

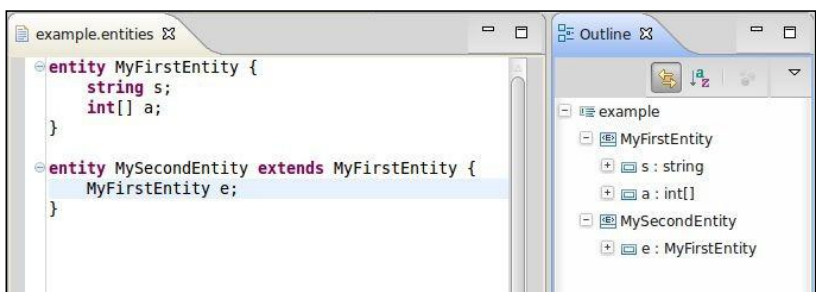
大纲如下截图所示：



我们可以通过使用图像来进一步丰富实体和属性的标签。为此，我们创建一个目录，其中放置要使用的图标图像文件。为了从受益于Xtext对图像的默认处理中获益，我们调用目录图标，并将两个gif图像放在那里，Entity.gif和属性.gif（分别用于实体和属性）。然后，我们在实体标签提供程序中定义了两个图像方法，其中我们只需要返回图像文件的名称(Xtext将为我们做其余的工作)：

```
类实体标签提供程序扩展了DefaultEObjectLabelProvider{  
    ... 如以前一样  
    def图像(实体e) { “Entity.gif” }  
  
    def图像(属性a) { “Attribute.gif” }  
}
```

您可以通过重新启动月食来查看结果，如下屏幕截图所示：



现在，实体和属性标签看起来更好了。



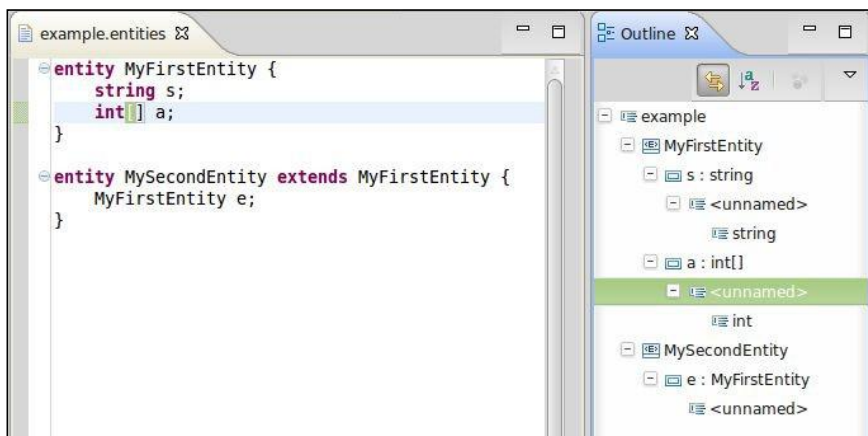
如果您计划为DSL导出插件，以便其他人可以在其月食中安装它们（见第11章，构建和发布），您必须确保图标目录被添加到build.properties文件中，否则该目录将不会被导出。UI插件的build.properties文件的bin.includes部分应该如下所示：

```
bin.includes=元器件-输入值/, \
    .,
    \plugin.xml,
    \icons/
```

“大纲”视图

默认的大纲视图具有很好的功能。特别是，它提供了工具栏按钮，以保持大纲视图选择与编辑器中当前选择的元素同步。此外，它还提供了一个按钮来排序的元素
树形树按字母顺序排列。

默认情况下，树状结构是使用DSL的元模型的包含关系来构建的；这种策略在某些情况下并不是最优的。例如，属性定义还包含属性类型元素，即a带有子元素的结构化定义（例如，元素类型、数组和长度）。如果展开属性元素，则这反映在大纲中（请参见以下屏幕截图）。



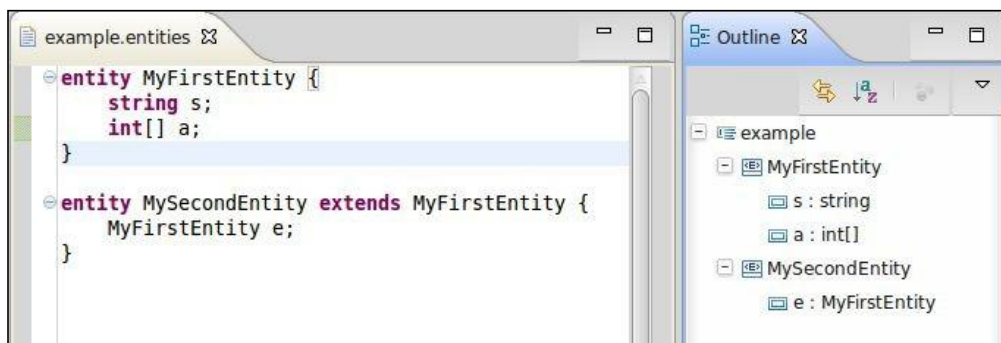
这显示了不必要的元素（如基本类型名称），因为它们显示在属性的标签中，以及不能用名称表示的其他元素（如数组特性）。

我们可以通过使用src文件夹org.example.entities中生成的存根类实体大纲树提供程序来影响大纲树的结构。ui.outline.同样，在这个类中，自定义设置是使用多态调度机制以声明式方式指定的。官方文档详细说明了可以定制的所有功能。

在我们的示例中，我们只是想确保属性的节点是“叶”节点，也就是说，它们不能被进一步扩展，并且它们没有子节点。为了实现这一点，我们只需要定义一个名为_isLeaf的方法（注意下划线），并使用元素类型的参数，返回true。因此，在我们的例子中，我们写道：

```
类实体大纲树提供程序扩展
    默认的大纲树提供程序{
    def _isLeaf(属性a) {true}
}
```

让我们重新启动Eclipse，现在可以看到属性节点不再公开子节点（请参见以下屏幕截图）。



除了定义叶节点外，您还可以通过定义方法_createChildren，将大纲节点的类型和模型元素的类型作为参数，来指定树中特定节点的子节点。这对于定义大纲树的实际根元素非常有用。可以看到，默认情况下，树的根于源文件的一个节点。在本示例中，最好有一个具有许多根节点的树，每个根节点代表一个实体。大纲树的根始终由DefaultRoot节点类型的节点表示；根节点实际上不可见，它只是将在树中以根形式显示的所有节点的容器。

因此，我们定义了以下方法（我们的实体模型是由a模型元素）：

```
公共类实体大纲树提供程序。{
    ... 如以前一样
    defvoid_createChildren(文档根节点在线节点，
                           模型模型) {
        model.entities.每个[实体|
            创建节点（外线节点，实体）；
        ]
    }
}
```

这样，在构建大纲树时，我们为每个实体创建一个（根）节点，而不是源文件的一个根节点（方法创建节点是Xtext基类的一部分）；结果可以在以下屏幕截图中看到：

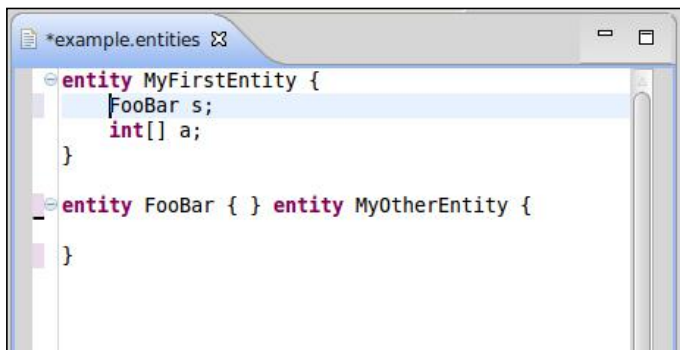


自定义其他方面

在第8章的一种表达式语言中，我们将展示如何自定义内容助手；对于简单的实体DSL，没有必要这样做，因为默认的实现已经做得很好。

自定义格式设置

如果您尝试应用我们在第4章验证中实现的快速修复程序之一，您可能会注意到在EMF模型更改后，编辑器立即反映这一更改；但是，结果的文本表示格式不佳（如下屏幕截图所示，我们应用了添加缺少实体FooBar的快速修复程序）。



一般来说，表示AST的EMF模型不包含有关文本表示的任何信息，即所有空白字符都不是EMF模型的一部分（毕竟，AST是实际程序的抽象）。

Xtext在另一个称为节点模型的内存模型中跟踪这些信息。节点模型包含语法信息，即文本文档中的偏移量和长度。但是，当我们手动更改EMF模型时，我们不提供任何格式化指令，而Xtext使用默认的格式化程序获取已修改或添加的模型部件的文本表示形式。

实际上，在DSL的编辑器中，Xtext已经生成了格式化源的菜单；由于Eclipse编辑器(例如JDT编辑器)的标准，您可以从编辑器的上下文菜单或使用键组合Ctrl+Shift+F访问“格式”菜单。

默认的格式化程序是一个白色的格式化程序，您可以在实体DSL编辑器中测试它：这个格式化程序只是用一个空格分隔程序的所有令牌。通常，您将希望更改此默认行为。

如果提供自定义格式化程序，这不仅在调用格式菜单时使用，而且在手动修改AST模型后需要更新编辑器内容时（例如，执行语义修改的快速修复）时使用。

Xtext生成器已经创建了一个存根类，用于自定义DSL的格式。格式化器存根可以在主插件项目中找到，对于我们的实体DSL，这个类是 `org.example.entities.formatting`。实体的格式化器。

要实现的方法是配置格式，它提供一个格式配置对象作为参数。使用此对象，我们可以声明地声明要应用于特定令牌的格式化操作（例如，之前、之后、前后等）执行行行、缩进、删除空格等等）。

对于实体DSL，我们决定执行以下格式设置：

- 在实体左卷括号“{”后面插入一行换行
- 在实体右卷括号“}”之后插入两行包装，以便实体用一条空行分隔
- 对实体之间的大括号进行缩进令牌（以便使属性在实体内进行缩进）
- 在每个属性声明之后插入一行换行符（即，在“；”之后）
- 删除属性声明的“；”之前可能的空格

格式配置方法名称是描述性的（例如，设置输入、设置包装等）。要指定要应用格式的令牌，我们可以依赖DSL的语法访问类（由Xtext生成）；在我们的示例中，它被称为“实体语法访问”。使用此类，我们可以访问与规则相关的令牌（例如，获取属性属性Access、获取属性访问等）和关键字（例如，卷括号、分号，等等）。对于后者，月蚀内容帮助将帮助您选择要使用的正确方法。

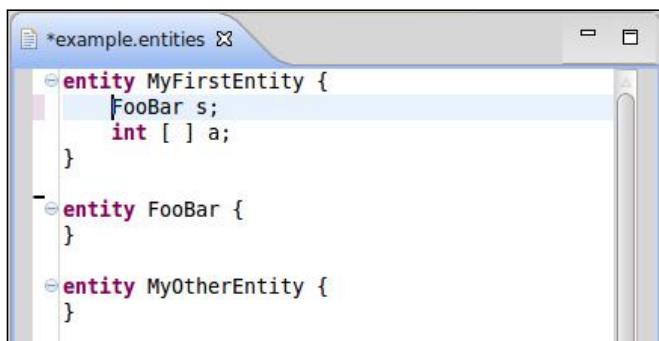
总结一下，要实现我们刚刚绘制的格式化，我们需要实现配置格式化方法如下：

类实体格式化器扩展了对物质的抽象声明

```
覆盖受保护的无效配置格式化（格式化配置c）{
    实体的//格式设置
    例如=，获取实体访问()
    {c.setIndentation(e.getLeftCurlyBracketKeyword_3(), 之间的//缩进
        e.getRightCurlyBracketKeyword_5())
    {c.setLinewrap()。after(e.getLeftCurlyBracketKeyword_3())后的//换行符
```

```
//出现后的两个换行符}c.setLinewrap (2) 。
after(e.getRightCurlyBracketKeyword_5 ())
针对属性的//格式设置
vala=g. 获取属性访问()
//换行符后; c.setLinewrap (1) 。
after(a.getSemicolonKeyword_2 ())
//删除; c.setNoSpace()。before(a.getSemicolonKeyword_2
()) 之前可能的空格
}
}
```

如果您现在重新启动月食，并尝试应用我们的快速修复程序，您将会看到添加的实体格式良好，如下屏幕截图所示：



此外，您也可以通过选择在编辑器中设置整个源从上下文菜单或使用键组合Ctrl+Shift+F设置格式。

其他自定义设置

到目前为止，您看到的所有自定义都是基于对生成存根类的修改，并在src-gen目录下模块中生成的Guice绑定。

然而，由于Xtext在任何地方都依赖于注入，因此即使没有生成存根类，也可以为任何机制生成自定义实现。



如果在Eclipse中安装了XtextSDK，则可以检查Xtext的源。您应该学会通过导航到这些源来检查这些源，看看注入了什么以及如何使用它。然后，您就准备好提供一个自定义实现并注入它。您可以使用“月蚀导航”菜单。特别是，要快速打开一个Java文件（即使是来自库中的源文件），请使用Ctrl+Shift+T（打开类型...）。这只适用于Java类，所以如果你想快速打开另一个源文件（例如，Xtend类或Xtext语法文件），请使用Ctrl+Shift+R（打开资源...）。两个对话框都有一个文本字段，如果开始键入，可用将很快显示元素。Eclipse到处都支持CamelCase，因此您可以键需输入复合名称的大写字母即可快速获得所需的元素；例如，要打开实体运行时模块Java类，请使用打开类型。菜单和只需数字“ERM”以看到过滤的结果。

作为一个例子，我们将展示如何自定义存储生成文件的输出目录（如我们在第5章代码生成中看到的，默认值为src-gen）。当然，用户可以使用Xtext为DSL生成的“属性”对话框修改此输出目录（参见第5章代码生成），但我们希望定制实体DSL定制默认输出目录，以便其成为实体生成。

使用注入的I输出配置提供程序在Xtext实例内部检索默认输出目录。如果您查看此课程（参见前面的提示），您会看到：

导入com.google.inject。实施：

@实现者（输出配置提供程序）。类）

公共接口I输出配置提供程序{设置<输出配置>获取输出配置();

...

@实现由Guice注释告诉注入机制接口的默认实现。因此，我们需要做的是创建一个默认实现的子类（即输出配置提供程序），并为接口I输出配置提供程序提供一个自定义绑定。

我们需要覆盖的方法是获取输出配置；如果我们看看它的默认实现，我们会看到：

```
公共集<输出配置>获取输出配置() {输出配置默认输出=新
    输出配置(IFileSystemAccess)。DEFAULT_OUTPUT);
    defaultOutput.setDescription (“输出文件夹”);
    defaultOutput.setOutputDirectory (“./src-gen”);
```

```
defaultOutput.setOverrideExistingResources(true);
defaultOutput.setCreateOutputDirectory(真);
defaultOutput.setCleanUpDerivedResources(true);
defaultOutput.setSetDerivedProperty(真);
返回newHashSet(默认输出);
}
```

当然，有趣的部分是呼吁设置输出目录。我们可以简单地在我们覆盖的方法版本中复制整个方法主体，并更改这一行；但是，我们更喜欢：

1. 调用超级实现并将结果存储在一个局部变量中（它是一个集合）。
2. 取该集合的第一个元素。
3. 调用设置输出目录与我们所需的输出目录。
4. 返回存储在本地变量中的集合。

我们没有定义一个Java子类，而不是定义一个Xtend子类。前面的操作可以使用Xtent“带操作符”轻松实现(参见第3章Xtend编程语言)；此外，我们使用隐式静态扩展从Xtend默认库中得到的方法(特别是head，它返回集合的第一个元素)和setter方法的语法糖。我们的自定义Xtend类如下：

```
类实体输出配置提供程序正在扩展
输出配置提供程序{

    公共valENTITIES_GEN=" "。/实体-代"

    覆盖获取输出配置() {
        super.getOutputConfigurations()=>[
            head.outputDirectory=ENTITIES_GEN
        ]
    }
}
```

注意，我们为输出目录使用公共常量，因为以后在其他类中可能需要它。

我们在主插件项目中创建了这个类，因为这个概念不仅仅是一个UI概念（它也在框架的其他部分中使用）；此外，由于它处理生成功能，我们在生成器子包中创建它。

现在，我们必须在实体运行时模块类中绑定我们的实现：

公共类实体运行时模块正在扩展

抽象实体运行时模块{

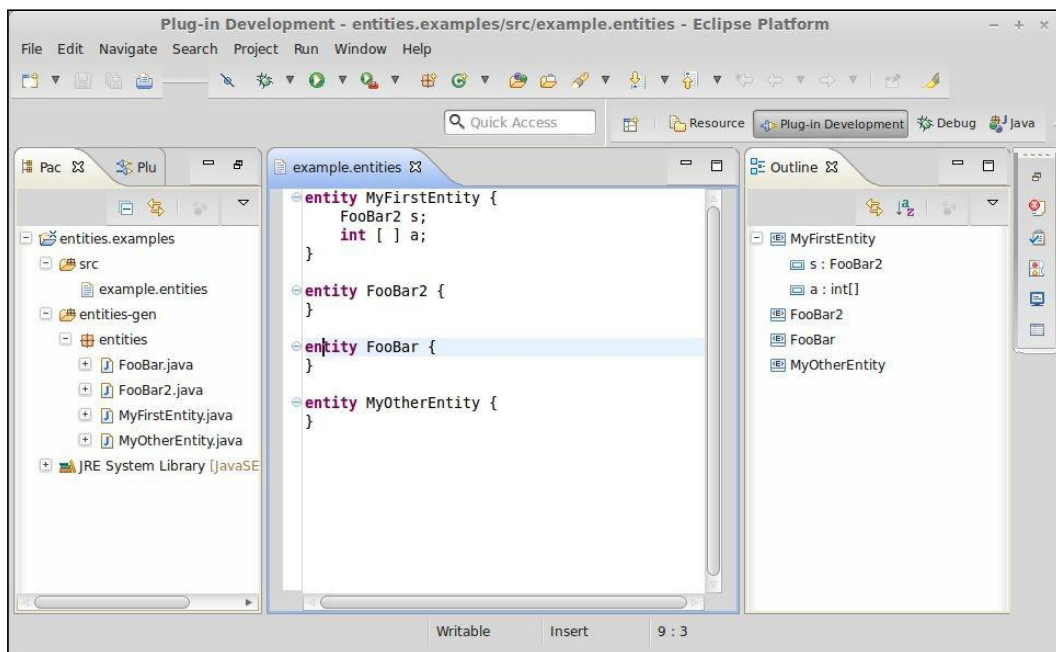
公共类<吗？扩展I输出配置提供程序>bindI输出配置提供程序(){

返回实体输出配置提供程序。**类别**

}

}

如果我们现在重新启动Eclipse，我们可以验证Java代码是否被生成成实体-gen，而不是src-gen，如下屏幕截图所示。如果以前使用同一项目，src-gen目录可能仍存在前几代目录；需要手动删除它并将新实体-gen设置为源文件夹。



产品简介

在本章中，我们介绍了Xtext所依赖的谷歌Guice依赖注入框架。您现在应该知道，在整个框架中一致地注入自定义实现是多么容易。您还学习了如何为DSL自定义一些基本的运行时和IDE概念。

下一章介绍如何对在Xtext中实现的语言执行单元测试。测试驱动的开发是一种重要的编程技术，它将使您的实现更可靠，更能适应库的变化，并允许您快速编程。

7

正在进行的测试

在本章中，您将学习如何通过使用Junit框架和Xtext提供的附加实用程序类来测试DSL实现。这样，DSL实现将有一套可以自动运行的测试。我们将使用前面几章中开发的实体DSL来展示测试在Xtext中实现的DSL的运行特性和UI特性的典型技术。

测试工作简介

编写自动化测试是编写软件时的一种基本技术/方法。它将帮助您编写高质量的软件，其中大部分方面（可能是所有方面）以某种方式以自动和连续的方式验证。虽然成功的测试并不能保证软件没有错误，但自动化测试是专业编程的必要条件（参见贝克2002, Martin2002, 2008, 2011）。

测试还将记录您的代码，无论是框架、库还是应用程序；测试是一种文档形式，不会对实现本身有过时的风险。Javadoc注释可能不会与文档中的代码保持同步，如果不持续更新，手册往往会过时，而如果测试不是最新的，它们就会失败。

测试驱动开发(TDD)方法甚至在编写生产代码之前就促进了测试的编写。在开发DSL时，您可以通过不必先编写测试来放松这种方法。但是，一旦向DSL实现添加到新功能，就应该立即编写测试。必须从一开始就考虑到这一点，因此，您不应该尝试编写DSL的完整语法，而是逐步进行；编写一些规则来解析一个最小的程序，并立即编写测试用于解析一些测试输入程序。只有当这些测试通过时，您才应该继续实现语法的其他部分。

此外，如果已经可以使用DSL的当前版本实现一些验证规则，那么您也应该为当前验证器检查编写测试。

理想情况下，不必运行Eclipse来手动检查DSL的当前实现是否按预期工作。然后，使用测试将使开发速度更快。

测试的数量将随着实现的增长而增加，并且在每次添加新功能或修改现有功能时都应执行测试。您会看到，由于测试将自动运行，一遍又一遍地执行它们除了触发执行之外不需要额外的努力（想想，如果您应该手动检查添加或修改的内容没有破坏一些东西）。

这也意味着您不会害怕触摸实现中的某些东西；在做了一些更改后，只要运行整个测试套件并检查是否损坏了一些问题。如果某些测试失败，您只需要检查是否实际预期失败（以防修复测试）或者是否必须修复您的修改。



值得注意的是，使用版本控制系统(如Git)对于很容易回到已知状态至关重要；仅仅尝试代码并使用测试查找错误并不意味着您可以很容易地跟踪。

您甚至不会害怕将实现移植到所使用的框架的新版本中。例如，当Xtext发布新版本时，很可能某些API发生了更改，并且可能不再使用新版本构建DSL实现。当然，需要运行MWE2工作流。但是在源代码再次编译后，测试套件会告诉您DSL的行为是否仍然相同。特别是，如果某些测试失败，您可以立即了解需要更改哪些部件以符合Xtext的新版本。

此外，如果您的实现依赖于一个可靠的测试套件，贡献者将更容易为DSL提供补丁程序和增强功能；他们可以自己运行这个测试套件，或者他们可以为特定的bug修复程序或新特性添加进一步的测试。主要开发人员也将很容易通过运行测试来决定是否接受这些贡献。

最后，但并非最不重要的是，你会发现从一开始就编写测试将迫使你编写模块化代码（否则你将无法轻易地测试它），这将使编程更加有趣。

Xtext和Xtend本身是用测试驱动的方法开发的。

六月4号公路

Junit是Java最流行的单元测试框架，它与EclipseJDT一起发布。特别是，这本书中的例子是基于Junit版本4。

要实现Junit测试，您只需要编写一个使用`@org.junit.Test`注释的类。我们将把这些方法称为简单的测试方法。这样的Java(或Xtend)类可以使用“Junit测试”启动配置在Eclipse中执行；用`@Test`注释的所有方法都将由Junit执行。在测试方法您可以使用Junit提供的断言方法来实现测试。对于例如，`arsertEquals`（预期，实际）检查两个参数是否相等；`arsertTrue`（表达式）检查传递的表达式计算值是否为true。如果断言失败，Junit将记录此类失败；特别是在月蚀中，Junit视图将向您提供有关失败测试的报告。理想情况下，测试不应失败（您应该在Junit视图中看到绿色条）。



所有的测试方法都可以由Junit按任何顺序执行，因此，您不应编写依赖于另一个方法的测试方法；所有的测试方法都应相互独立执行。

如果使用`@Befrore`注释方法，该方法将在该类中的每个测试方法之前执行，因此，它可以用于为该类中的所有测试方法准备通用设置。类似地，在每个测试方法之后，使用`@After`注释的方法将被执行（即使它失败），因此，它可以用于清理环境。将执行使用`@BeforeClass`注释的静态方法在所有测试方法开始前只有一次（`@AfterClass`具有互补的直观功能）。

ISetup接口

运行测试意味着，除了实现DSL之外，我们还需要引导该环境来支持EMF和Xtext。这是通过适当的实现来完成的。我们需要根据运行测试的方式进行不同的配置；无论是否存在Eclipse，以及是否存在EclipseUI。设置环境的方式就完全不同了。当Eclipse存在时，因为许多服务是共享的，并且已经是Eclipse环境的一部分。当为非日蚀使用设置环境（也称为独立）时，必须配置一些东西，例如创建Guice注入器和注册EMF所需的信息。ISetup接口中的方法`createInjectorAndDoEMFRegistration`正好可以做到这一点。

除了创建注入器之外，此方法还执行EMF全局注册表的所有初始化，以便在调用该方法之后，可以完全使用加载和存储语言模型的EMF API，即使不运行Eclipse。Xtext生成这个接口的实现，以您的DSL命名，可以在运行时插件项目找到。对于我们的实体DSL，它被称为实体标准设置。



名称“独立”表示了Eclipse外部运行时必须使用这个类。因此，在Eclipse内部运行时永远不能调用前面的方法（否则EMF注册表将不一致）。

在普通Java应用程序中，设置DSL的典型步骤（例如，实体DSL）如下所示：

```
喷油器喷射器=新实体标准设置()。创建注入器和远程管理器注册
(); Xtext资源集资源集=
    injector.getInstance(Xtext资源集。类); resourceSet.addLoadOption
    (Xtext资源。OPTION_RESOLVE_ALL, 布尔值。真实的);
资源资源=resourceSet.getResource(Uri.
    createURI(“/path/to/my.entities”), 真的);
型号=(型号) resource.getContents().get(0);
```

这个独立的设置类对于JUnit测试也特别有用，它可以在没有Eclipse实例的情况下运行。这将加快测试的执行速度。当然，在这些测试中，您将无法测试UI功能。

正如我们将在本章中看到的，Xtext提供了许多用于测试的实用工具类，它们不需要我们显式地设置运行时环境。但是，如果您需要调整生成的独立编译器，或者需要以特定的方式为单元测试设置环境，那么了解设置类的存在是很重要的。

为您的DSL实施测试

Xtext高度促进使用单元测试，这反映在默认情况下，MWE2工作流生成一个特定的测试DSL的插件项目上。事实上，测试通常应该驻留在一个单独的项目中，因为它们不应该作为DSL实现的一部分进行部署。这个附加项目以tests后缀结束，因此，对于我们的实体DSL，它是org.example.entities.tests。测试内容插件项目对测试所需的Xtext实用程序包具有所需的依赖性。

我们将使用Xtend来编写JUnit测试。

在测试项目的src-gen目录中，您可以找到无头测试和UI测试的注入器提供程序。您可以使用这些提供程序轻松地编写JUnit测试类，而不必担心注入机制的设置。使用喷油器提供程序的JUnit测试通常会具有以下形状（以实体DSL为例）：

```
@运行与（类型（Xtext运行器））@注入与（实体注入器提供
程序）类型）类MyTest{
    @输入MyClass
    ...
}
```

正如在前面的代码中所暗示的，在这个类中，您可以依赖于注入；我们使用了@InjectWith，并声明必须使用实体注入器提供程序来创建注入器。实体注入器提供程序将透明地为独立环境提供正确的配置。在本章后面将看到，当我们要测试UI功能时，我们将使用EntitiesUiInjectorProvider（请注意该名称中的“Ui”）。

正在测试解析器

您可能要编写的第一个测试是涉及解析的测试。

这反映了这样一个事实，即语法是您在实现DSL时必须写的第一件事。在开始测试之前，您不应该尝试编写完整的语法：您应该只编写几条规则，然后很快编写测试，以检查这些规则是否真的按照您的预期解析了一个输入测试程序。

好的是，您不必将测试输入存储在文件中（尽管您可以这样做）；传递给解析器的输入可以是字符串，而且由于我们使用Xtend，我们可以使用多行字符串。

Xtext测试框架提供了解析助手类来轻松地解析字符串。注入机制将自动告诉此类使用DSL的解析器来解析输入字符串。要解析字符串，我们注入的一个实例

ParseHelper<T>，其中T是DSL模型中的根类的类型——在实体示例中，此类称为模型。方法ParseHelper.parse将在解析给它的输入字符串后返回一个T的实例。

通过注入ParseHelper类作为扩展，我们可以直接在要解析的字符串上使用它的方法。因此，我们可以写道：

```
@运行与（类型（Xtext运行器））@注入与（类型（实体注入
器提供程序））类型实体参数测试{

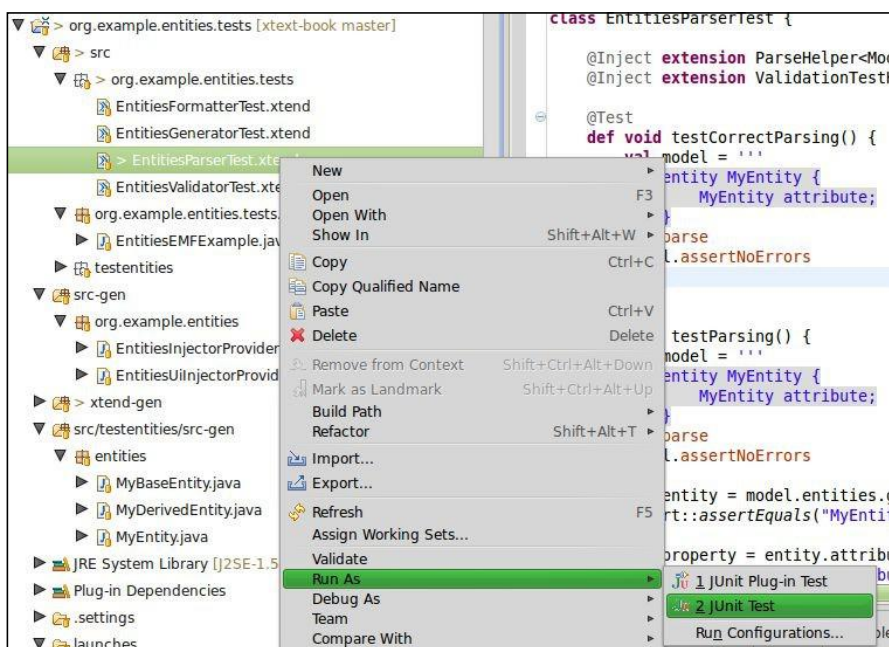
    @对象扩展解析帮助程序<模型>@测试
    定义无效测试分析() {
        val模型= “” 实体我实体{
            My实体属性;
        }
        ‘’ ‘。解析器

        val实体=model.entities.get(0) 断言：： 断言等(“实
        体”， entity.name)

        val属性=entity.attributes.get(0) 断言：： 断言相等(“属
        性”， attribute.name); 断言：： assert相等(“我的实体”，
        (attribute.type.elementType作为实体类型)。
        entity.name);
    }
    ...
}
```

在这个测试中，我们解析输入，并测试预期的结构是否是由于解析而构建的。这些测试在实体DSL中没有增加多少价值，但在更复杂的DSL中，您确实希望测试分析的EMF模型的结构如您所期望的（我们将在第8章表达式语言中看到一个示例）。

现在可以运行测试：右键单击Xtend文件，然后选择以JUnit方式测试运行，如下屏幕截图所示。测试应该通过，你应该在垃圾视图中看到绿色的条。



请注意，即使输入字符串包含语法错误，解析方法也会返回一个EMF模型；因此，如果您想确保输入字符串的解析没有任何语法错误，您必须显式检查。为此，您可以使用另一个实用工具类，验证测试帮助程序。此类提供了许多采用EObject参数的断言方法。您可以使用扩展字段，只需对解析的EMF对象调用断言错误。或者，如果不需要EMF对象，但只需要检查是否没有解析错误，则只需根据解析结果调用它，例如：

类实体参数测试 {

 <模型>@对象扩展验证测试帮助程序

...

 @测试结果

 定义无效测试修正解析() { “

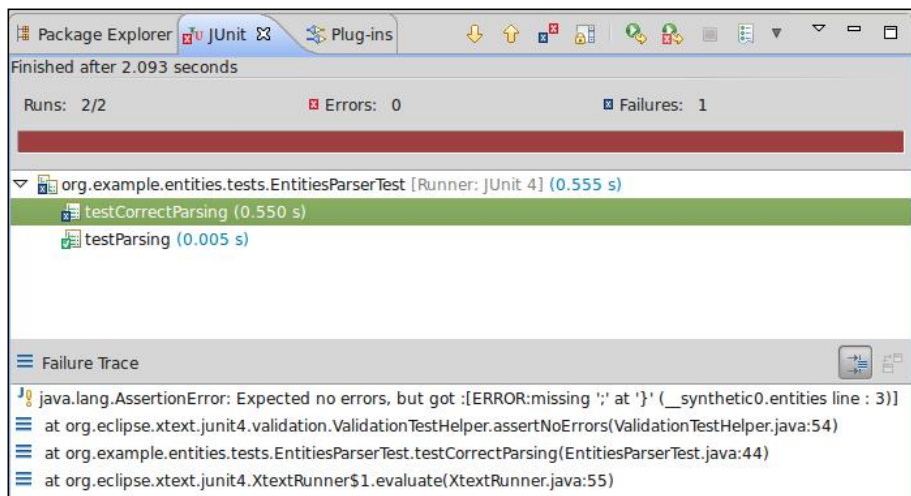
 实体My实体属性

 }

 “ ‘.parse.assertNoErrors

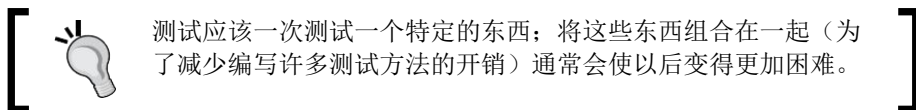
 }

如果再次尝试运行测试，此新测试将失败，如下屏幕截图所示：



报告的错误应该足够清楚：我们忘记了在我们的输入程序中添加终止“；”，因此我们可以修复它并再次运行测试；这一次绿色条应该回来。

现在可以编写其他@Test方法来测试DSL的各种特性（请参见示例的来源）。根据DSL的复杂性，您可能需要编写其中的许多内容。



测试应该一次测试一个特定的东西；将这些东西组合在一起（为了减少编写许多测试方法的开销）通常会使以后变得更加困难。

记住，在实现DSL时应该遵循此方法，而不是在实现所有方法之后。如果您严格执行此操作，您将不必启动Eclipse来手动检查您是否正确实现了功能，您将注意到这种方法将让您快速编程。

理想情况下，您应该从具有单一规则的语法开始，特别是当语法包含非标准终端时。第一个任务是编写一个只解析所有终端的语法。为此编写一个测试，以确保在继续之前没有重叠的终端；如果未添加终端，则不需要这样做标准终端。在此之后，在每一轮开发/测试中添加尽可能少的规则，直到语法完成。