

基于 JFace Text Framework 构建全功能代码编辑器:

第 1 部分

引言

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

2008 年 3 月 20 日

JFace Text Framework (JFace 文本框架,后面直接简称为 JTF) 是 Eclipse 中重要的框架,是其它开发工具的基石之一,比如 JDT (Java Development Tool) 中的 Java 源代码编辑器就是基于它开发的。相信用过 JDT 的人都会对它的源代码编辑器有深刻印象,因为它的很多功能可以让我们很方便的编辑 Java 源代码,比如语法高亮和内容提示等等。所有这些功能都是在 JTF 架的基础上实现的,所以学会使用这个框架对于开发某种语言的编辑工具是至关重要的。即使不是为了这个目的,学习这个框架也有助于提高对 Eclipse 整体的认识。但是 JTF 本身也确实是一个复杂的框架,非几句话能解释清楚。本系列文章以 JTF 的特性为中心,逐个介绍某个特性的相关概念以及实现方式,并且会给出相应的示例程序。因此随着系列的进行,示例程序会发展成一个全功能的编辑器。本文是此系列文章的第一篇,列举了 JTF 的一些特性,并且简略的介绍了 ANTLR 的相关技术,为后续的文章做好了铺垫。本系列的所有代码都是在 Eclipse 3.3 上调试运行的。

JFace Text Framework (JFace 文本框架,后面直接简称为 JTF) 是 Eclipse 中重要的框架,是其它开发工具的基石之一,比如 JDT (Java Development Tool) 中的 Java 源代码编辑器就是基于它开发的。相信用过 JDT 的人都会对它的源代码编辑器有深刻印象,因为它的很多功能可以让我们很方便的编辑 Java 源代码,比如语法高亮和内容提示等等。所有这些功能都是在 JTF 架的基础上实现的,所以学会使用这个框架对于开发某种语言的编辑工具是至关重要的。即使不是为了这个目的,学习这个框架也有助于提高对 Eclipse 整体的认识。但是 JTF 本身也确实是一个复杂的框架,非几句话能解释清楚。本系列文章以 JTF 的特性为中心,逐个介绍某个特性的相关概念以及实现方式,并且会给出相应的示例程序。因此随着系列的进行,示例程序会发展成一个全功能的编辑器。本文是此系列文章的第一篇,列举了 JTF 的一些特性,并且简略的介绍了 ANTLR 的相关技术,为后续的文章做好了铺垫。本系列的所有代码都是在 Eclipse 3.3 上调试运行的。

JFace 文本框架特性一览

JFace 本身是一个更高级的 UI 库,文本框架只是其中的一部分。JFace 本身是基于 SWT 的,所以文本框架的功能也是受制于 SWT 的,具体的说,是受制于 SWT 中 StyledText 的能力。如果要简单的概括一下 JTF 是什么,可以说 JTF 只不过是把 StyledText 的一些功能包装的很方便让你来用罢了。所以对 JTF 的功能不要感到很神奇,所有的特性都可以在 SWT 中找到它的根据。目前, JTF 支持以下特性:

- Syntax Highlight(语法高亮)
- Double Click (鼠标双击)
- Content Assistant (内容提示)
- Text Decoration (文本装饰)
- Text Hover (文本悬浮帮助)
- Annotation Hover (标注悬浮帮助)
- Quick Assistant (快速帮助)
- Hyperlink (超链接)
- Template (模版)
- Text Formatting (文本格式化)
- Text Folding (文本折叠)

本系列将介绍所有这些特性,至于这些特性的具体概念会在以后的文章中一一介绍,而且还会顺带介绍一些 JTF 本身并不支持的特性,比如 Triple Click (鼠标三击)。由于 JFace 和 SWT 都在不断发展中,所以也有可能提到一些上面没有列出的特性。但是 JTF 本身是一个复杂的框架,所以我不打算一一讲解每一行示例代码的含义,只会关注基本概念和架构以及核心代码,同时为了尽量保持示例代码的简洁,我也不考虑性能等诸多因素。

ANTLR

对于编写自定义的源代码编辑器来说，有一个源代码的解析器是很重要的。现在有很多这样的工具，比如 **JavaCC**, **Antlr**, **bison** 等等，我们将使用 **Antlr** 来构造我们的语言解析器。如果你不熟 **Antlr** 也没有关系，这并不是本文的重点，我会稍微介绍一下 **Antlr** 的基本概念，你只要了解我用 **Antlr** 做了什么事就可以了。

Antlr 是一个解析器生成器，可以根据你书写的文法自动生成解析器代码，并且可以生成语法树。之所以需要一个解析器是因为我们要在本系列中一步步实现一个源代码编辑器，很多情况下我们需要一颗语法树来帮助我们获得一些信息。我们可以通过遍历这棵语法树查询我们想要的信息，然后决定我们的程序逻辑。

我将自定义一种简单的语言，然后实现这种语言的编辑器，可以称这种语言为数学表达式语言，在这种语言里，我们仅支持变量声明和四则运算，并且只支持整数类型。看下面的例子：

清单1. 非常简单的假想语言：数学表达式语言

```
        a = 3;
b = (1 + 2) * 3;
a / 2 + b * 5;
```

从例子中可以看出，我定义的这种语言以分号作为行的结尾，变量声明采用等号作为操作符，变量声明了之后，可以参与到四则运算中，或者直接使用数字常量。这就是全部的语法了。下面是对这种简单语言的一个总结：

- 操作符: +, -, *, /, (,), =
- 变量名: 任意长度，仅能由字母组成
- 每条语句以分号结尾

我为这种语言定义了 **Antlr** 的文法，由于并非本文主题，这里不列举其代码，下面是一个与之相关的文件清单，大家可以在示例代码中找到它们：

表 1. ANTLR 相关文件说明

文件名	说明
Expr.g	语言的ANTLR语法描述
Expr__.g	ANTLR自动从Expr.g生成的词法分析器文法描述
Expr.tokens	ANTLR自动生成的所有符号的列表
ExprLexer.java	语言的词法分析器
ExprParser.java	语言的语法分析器
TokenList.java	符号列表，可以通过它得到指定偏移位置的符号以及得到某个符号的下一个符号
TokenManager.java	符号管理器，维护 IDocument 实例到符号列表的映射，每次 IDocument 实例发生变化时，符号列表缓存将被清空，这样下次会得到一个重新生成的符号列表
TreeManager.java	语法树管理器，维护 IDocument 实例到语法树的映射，每次 IDocument 实例发生变化时，语法树的缓存将被清空，这样下次会得到一个重新生成的语法树
TreeHelper.java	这个方法定义了一些和语法树相关的方法，可以从语法树中抽取一些信息
IExprTokens.java	这个接口定义了语言中可能出现的所有符号
SharedParser.java	为了稍微提高一点性能，维护了词法分析器和语法分析器的单一实例

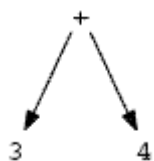
需要强调的是：上述这些类对于实现一个完整的代码编辑器是非常重要的。对于文本来说，我们没有篇幅去仔细了解它们是怎么实现的，只需要明白它们能够做什么就可以了。对于 **ANTLR** 的具体使用和解析器相关的内容，值得花一本书的篇幅来介绍，对此有兴趣的读者可以阅读最后的参考文献。我们不做详细阐述，但是下面会介绍一些**ANTLR**的基本概念：

- 符号(Token): 可以认为符号是一个具有特定意义的词法单元，比如一个英文单词，一个数字都是一个符号，比如语句”a = 3;”，它包含了 6 个符号：a, 空格，等于号，空格，3，分号。当然具体哪些

部分能构成一个符号是由你写的文法来规定的。符号可以有很多属性，比如符号的类型、起始位置、结束位置等等

- 词法分析器（**Lexer**）：词法分析器把一个字符流解析成为一个符号流。词法分析器一般要和一个语法分析器结合起来使用。词法分析器的输出就是语法分析器的输入
- 语法分析器（**Parser**）和语法树（**Tree**）：语法分析器把一个符号流解析成其它的格式，可以是一棵树或者其它什么东西。比如 $3 + 4$ 可以变为如下图所示的结构：

图1. 语法树



加号变成了一颗树的根节点，而3和4成为了它的孩子节点。在这样的树结构里，包含了一定的语义信息。我们通过遍历这颗树，得到数学表达式的结果或者进行其它操作。语法树的好处是可以重复的遍历它而不是每次都去把表达式解析一次。语法树实际上是一种中间格式。

- 文法（**Grammar**）：ANTLR中可以编写很多文法：比如词法分析器的文法，语法解析器的文法，语法规文法等。ANTLR的文法文件的扩展名是“.g”，根据你的文法类型，ANTLR会生成不同的源代码文件

一旦我们拥有了处理我们语言源代码的能力，并能够方便的管理源代码的词法语法信息，就相当于大楼有了牢固的地基一样，事情已经成功了一半了。后面的文章将告诉大家如何完成另外一半的工作。

初始示例代码

为了提供文本编辑和源代码编辑的能力，JTF 提供了 **TextViewer** 和 **SourceViewer**。**TextViewer** 仅仅面向纯文本的编辑，而 **SourceViewer** 提供了一个代码编辑器所需要的特性。代码编辑相对纯文本编辑来说复杂很多，需要很多附加功能，比如前面提到的 JTF 的所有特性，**SourceViewer** 都是做了一定的包装的。为了方便自定义，**SourceViewer** 通过一个 **SourceViewerConfiguration** 来控制这些特性的行为。缺省的实现基本上就和纯文本编辑器一样，所以我们从扩展 **SourceViewer** 以及 **SourceViewerConfiguration** 开始，一步步来完成我们自己的代码编辑器。本文附带的例子是一个起点，虽然有 **ExprViewer** 和 **ExprConfiguration**，但是还没有任何自定义代码。

在初始示例代码里，我们为 **Eclipse** 工具条添加了一个按钮，点击之后会出来一个对话框，我们的自定义编辑器就放在对话框里。之所以没有用 **Eclipse** 标准的 **org.eclipse.ui.editors** 扩展点，是因为 **editor** 本质上也是包装了一个 **SourceViewer**，它为我们隐藏了一些东西。所以为了更清楚的说明问题，我们从一个较为原始的状态开始。

结束语

JTF 是一个比较成熟的文本编辑框架，通过 JTF 我们可以实现自己的代码编辑器。我的目的是对 JTF 的每个特性进行详细的解说，最终能够构建出一个可以达到和 **Java Editor** 相同水平的代码编辑器。本系列的每一篇都会附带一个阶段性的代码示例，以便读者对照代码更好的理解 JTF 的架构和细节。

声明

本文仅代表作者的个人观点，不代表 IBM 的立场。

下载

描述	名字	大小	下载方法
第一小节示例代码	jtf.tutorial.part1.zip	1120KB	HTTP

[→ 关于下载方法的信息](#)

参考资料

- 如果对ANTLR有更多的兴趣，请参考“[ANTLR 项目主页](#)”
- “[SWT 和 JFace，第 1 部分: 简介](#)”（developerWorks 中国，2005 年 5 月）介绍了 JFace 基础知识。

关于作者

马若劫是 Lotus Forms 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验，同时也是国内著名开源项目 LumaQQ 的创立者

第 1 段商标声明。 第 2 段商标声明。 其他公司、产品或服务的名称可能是其他公司的商标或服务标志。

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经 IBM 公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 2 部分

Syntax Highlight

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

2008 年 3 月 20 日

Syntax Highlight(语法高亮)是指把文本的不同内容用不同的颜色, 字体等渲染, 通过这种方式, 用户可以快速发现某种内容, 可以在短时间内对全文的结构有一个大概的了解。本文探讨如何在 **JTF** 中实现语法高亮。

Syntax Highlight(语法高亮)是指把文本的不同内容用不同的颜色, 字体等渲染, 通过这种方式, 用户可以快速发现某种内容, 可以在短时间内对全文的结构有一个大概的了解。本文探讨如何在 **JTF** 中实现语法高亮。

前提

语法高亮是把一段文本中的不同内容使用不同的样式来渲染的功能, 如下图所示:

图1. Java编辑器的语法高亮

```
public interface IColorConstants
{
    /** default color */
    RGB DEFAULT = new RGB(0, 0, 0);

    // number color
    RGB INTEGER = new RGB(0x0, 0x0, 0xFF);

    // variable color
    RGB VARIABLE = new RGB(0x84, 0x00, 0x0);
}
```

图 1 中的 Java 源代码的不同部分被 Java 编辑器渲染成了不同颜色。比如关键字是偏红的, 并且是粗体; 变量是蓝色, 注释是绿色等等。不同的颜色可以让用户迅速的找到想要的内容, 可以说是一个很方便的辅助功能。

在本系列的一部分中, 我提到了 ANTLR 以及使用 ANTLR 产生词法和语法分析器。语法高亮是一个很依赖词法分析器的功能, 因为词法分析器会把字符流解析成符号流, 符号包含了各种属性, 比如类型, 起始位置和结束位置等等, 语法高亮恰恰需要这些信息来渲染文本。我们已经有了些基础类可以帮助我们管理符号列表, 比如 **TokenList**, 它可以根据字符偏移得到对应的符号。这为实现语法高亮提供了良好的基础。

ITokenScanner

JTF 是通过一个 **ITokenScanner** 接口来了解符号信息的, 我们先来看看这个接口的定义:

清单 1. ITokenScanner 接口

```
public interface ITokenScanner {
    void setRange(IDocument document, int offset, int length);

    IToken nextToken();

    int getTokenOffset();
}
```

```

        int getTokenLength();
    }

```

可见通过这个接口，JTF 可以遍历所有的符号并且得到符号的偏移和长度信息。所以实现这个接口是一个实质性的工作。但是 JTF 怎么知道如何去渲染不同的文本呢？为了解决这个问题，JTF 也提供了一个 **IToken** 接口，并且有一个基本的实现类 **Token**。注意 ANTLR 里面也有 **Token** 这个类，两者的概念类似，只不过用法不同。JTF 的 **Token** 可以附带一些自定义的对象，我们可以把文本的属性，比如前景背景字体样式等等，都放到 **Token** 里面，这样 JTF 就知道该如何渲染文本了。这些文本属性也被 JTF 包装成了一个 **TextAttribute** 类，我们在构造 **Token** 之后的传入 **TextAttribute** 对象就可以了。

既然我们有了 **TokenList** 这样的基础类，实现 **ITokenScanner** 就是很简单的事了，我们看看 **ExprTokenScanner** 的代码：

清单 2. ExprTokenScanner 实现了 ITokenScanner

```

public class ExprTokenScanner implements ITokenScanner {
    /*
     * JFace token type definition
     */

    private static IToken VARIABLE = new Token(
        new TextAttribute(ColorManager.getInstance().getColor(Constants.VARIABLE)));
    private static IToken INTEGER = new Token(
        new TextAttribute(ColorManager.getInstance().getColor(Constants.INTEGER)));
    private static IToken DEFAULT = new Token(
        new TextAttribute(ColorManager.getInstance().getColor(Constants.DEFAULT)));

    // character offset
    private int offset;
    // last token returned by nextToken()
    private CommonToken lastToken;
    // token list
    private TokenList tokenList;

    public ExprTokenScanner() {
        offset = 0;
    }

    public int getTokenLength() {
        return lastToken.getStopIndex() - lastToken.getStartIndex() + 1;
    }

    public int getTokenOffset() {
        return lastToken.getStartIndex();
    }

    public IToken nextToken() {
        if (lastToken == null)
            lastToken = tokenList.getToken(offset);
        else
            lastToken = tokenList.getNextToken(lastToken);

        if (lastToken == null)
            return Token.UNDEFINED;

        switch (lastToken.getType()) {
            case IExprTokens.ID:
                return VARIABLE;
            case IExprTokens.INT:
                return INTEGER;
            case org.antlr.runtime.Token.EOF:
                return Token.EOF;
        }
    }
}

```



```

        default t:
            return DEFAULT;
    }
}

public void setRange(IDocument document, int offset, int length) {
    lastToken = null;
    this.offset = offset;
    tokenList = TokenManager.getTokenList(document);
}
}

```

`getTokenLength` 和 `getTokenOffset` 的实现是相当直接的，因为长度和偏移信息都保存在了 ANTLR 的 `Token` 实现中。`nextToken` 稍微有点长，因为我们需要把 ANTLR 的 `Token` 映射到 JTF 的 `Token`，所有用了一个 `switch` 语句来检查了一下 `Token` 的类型，从而返回相应的 JTF `Token`。返回的 `Token` 已经预先定义好了，并加上了文本属性信息。`setRange` 可能是个不太直观的方法，这个方法有两个目的：第一是将 `ITokenScanner` 和 `IDocument` 分离开，这样一个 `ITokenScanner` 可以为多个 `IDocument` 服务；第二是提高性能，因为每次都从头把文本渲染一遍是个不太经济的做法，所以 `setRange` 有 `offset` 和 `length` 两个参数来控制需要渲染的范围，提高了渲染速度。

PresentationReconciler

有了 `TokenScanner` 之后，剩下的事情就是让 JTF 知道它。上一部分提到了 `SourceViewerConfiguration`，它是 JTF 和外界交互的枢纽。我已经实现了 `ExprConfiguration`，但是它还是空的，现在我们要覆盖父类的 `getPresentationReconciler` 方法：

清单 3. 覆盖父类的 `getPresentationReconciler` 方法

```

public IPresentationReconciler getPresentationReconciler(ISourceViewer sourceViewer)
{
    PresentationReconciler reconciler = new PresentationReconciler();

    DefaultDamagerRepairer dr = new DefaultDamagerRepairer(getTokenScanner());
    reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
    reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

    return reconciler;
}

```

PresentationReconciler，顾名思义，是一个负责 **Presentation** 的类，这里 **Presentation** 指的就是文本的颜色，字体等等。文本渲染是一个不断进行的过程，因为用户可能在不断的修改文本内容。JTF 必须知道用户输入了什么，然后重新检查一下输入的内容，再刷新。所以 JTF 引入了 **Damager** 和 **Repairer** 的概念，**Damager** 负责计算用户的输入对哪些区域造成了破坏，**Repairer** 负责重新渲染被破坏的文本区域。**Damager** 和 **Repairer** 是按照文本类型来组织的，但是在我们的例子中，只有一个缺省文本类型，所以我们只为缺省类型设置了 **Damager** 和 **Repairer**。稍后我们会简单介绍一下文本类型的概念。

幸运的是 JTF 已经为我们提供了 **PresentationReconciler** 以及 **Damager**、**Repairer** 的缺省实现，很少有需要扩展这些类的情况。这里我就不详细解释了。

提示：`ExprConfiguration` 已经在 `JTFDialog` 中和 `ExprViewer` 联系起来了，具体参见 `JTFDialog` 的 `createDialogArea` 方法。

Content Type

你可以把 **Content Type** 翻译成文本类型后者内容类型，总之它是用来描述具有某种特征的文字的。比如在 XML 语言中，标签肯定是由大于号和小于号括起来的，这样你就可以给 XML 的标签定义一种文本类型，更细一点，XML 的结束标签是由 "</" 和 ">" 括起来的，所以也可以为结束标签定义一种单独的文本

类型。上一小节提到了不同的文本类型可以有不同的 **Damager** 和 **Repairer**，实际上 **JTF** 中很多特性都可以针对某个文本类型来做，因此适当的使用文本类型可以做一些更精细的控制。不过，这也得根据你要编辑的文本来具体考虑，像 **XML** 这样的非常结构化的语言，划分文本类型是很容易的。但是对于我这个例子中的数学表达式语言来说，就不太方便划分出很多类型来，你非要划分一下，可能会使得简单问题复杂化了。所以，要具体问题具体分析。

另外一个问题是 **JTF** 如何知道某段文本是什么样的类型呢？这里需要引入一个 **Partition**（分区）的概念，在 **JTF** 中，每个文档都有一个与之联系的 **Document Partitioner**（文档分区器），它会根据你给定的规则来扫描整个文本从而辨别哪块是哪个类型。所以它有一个相同的问题：对于非结构化的语言，划分分区不是特别容易的事情。

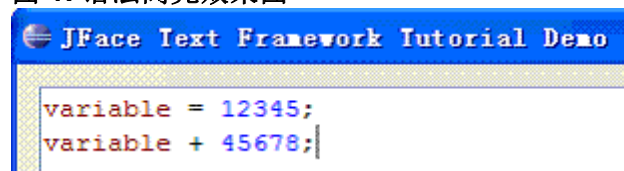
所以当你处理一种文法规则不那么严谨的语言的时候，也可以不定义文本类型，就用缺省的就够了。我的例子就都采用了这样的方式。同时由于 **ANTLR** 生成的词法分析器非常好用，所以也没有必要那么做。

提示：关于分区和扫描规则的内容，读者可以参考 **IDocumentPartitioner** 和 **IRule** 接口以及它们的实现。这里不做详细的解释了。

结束语

语法高亮的部分到这里就完成了，下图是例子的效果：

图 1. 语法高亮效果图



变量的颜色是咖啡色，数字是蓝色，其它都缺省是黑色。有了语法高亮的帮助，整个文档结构都一目了然了。

声明

本文仅代表作者的个人观点，不代表 **IBM** 的立场。

下载

描述	名字	大小	下载方法
第二小节示例代码	jtf.tutorial.part2.zip	1130KB	HTTP

[→ 关于下载方法的信息](#)

参考资料

- 如果对 **ANTLR** 有更多的兴趣，请参考“[ANTLR 项目主页](#)”
- “[SWT 和 JFace，第 1 部分: 简介](#)”（**developerWorks** 中国，2005 年 5 月）介绍了 **JFace** 基础知识。

关于作者

马若劫是 Lotus Forms 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验，同时也是国内著名开源项目 LumaQQ 的创立者

第 1 段商标声明。 第 2 段商标声明。 其他公司、产品或服务的名称可能是其他公司的商标或服务标志。

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经 IBM 公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 3 部分

Double Click 和 Triple Click

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

2008 年 3 月 27 日

Double Click (双击) 和 **Triple Click** (三击) 是方便用户选择文本 (当然不一定非得是选择文本) 的辅助功能。本文介绍在 **JTF** 里面如何自定义双击行为, 以及如何为 **JTF** 添加三击支持。

Double Click (双击) 和 **Triple Click** (三击) 是方便用户选择文本 (当然不一定非得是选择文本) 的辅助功能。本文介绍在 **JTF** 里面如何自定义双击行为, 以及如何为 **JTF** 添加三击支持。

Double Click

在 **Eclipse** 的 **Java** 编辑器中, 用户用鼠标在编辑区域双击时, 可以看到双击处的单词会被选中, 这就是 **JTF** 的 **Double Click** (双击) 特性。双击时触发的动作是可以自定义的, 不一定非要选择一段文本。

与双击相关的接口是 **ITextDoubleClickStrategy**, 它只有一个方法叫做 **doubleClicked**, 只要实现这个接口就可以了, 在例子中, 我添加了 **ExprDoubleClickStrategy** 类:

清单1. ExprDoubleClickStrategy 实现了 ITextDoubleClickStrategy 接口

```
public class ExprDoubleClickStrategy implements ITextDoubleClickStrategy {
    public void doubleClicked(ITextViewer viewer) {
        // get doc
        IDocument doc = viewer.getDocument();

        // get token list
        TokenList tokenList = TokenManager.getTokenList(doc);

        // get double click position
        int offset = viewer.getSelectedRange().x;

        // get token in that offset
        CommonToken token = tokenList.getToken(offset);

        // select whole token if token is not null
        if(token != null && token.getType() != Token.EOF)
        {
            // select double clicked token
            viewer.setSelectedRange(
                token.getStartIndex(), token.getStopIndex() - token.getStartIndex() + 1);
        }
    }
}
```

这个流程非常直接, 得到被点击的位置, 通过位置得到相应的符号, 然后选择整个符号。我们再次利用了 **TokenList** 来得到指定字符偏移处的符号。

和本系列第二部分一样, 有了实现还得让 **JTF** 知道你的实现, 我们再来修改 **ExprConfiguration**, 覆盖一个 **getDoubleClickStrategy** 方法:

清单2. 让 JTF 知道你的 Double Click 实现

```
public ITextDoubleClickStrategy getDoubleClickStrategy(  
    ISourceViewer sourceViewer, String contentType)  
{  
    return new ExprDoubleClickStrategy();  
}
```

只是简单的返回我们实现的 `ITextDoubleClickStrategy` 而已，这样 JTF 就知道我们的双击行为了，注意双击行为也是和文本类型绑定到一起的，但是我们只有一种类型，所以没有利用这个信息。

读者可以尝试本文的例子，双击某个变量，看看是否这个变量被全部选中了。

Triple Click

有了双击，可能自然就会想到三击。但是 JTF 本身是不支持三击行为的。我们需要自己实现，只要模仿双击的机制来做就可以了。

接口

模仿双击的处理方式，我们也添加一个 `ITextTripleClickStrategy` 接口，如下所示：

清单3. 模仿 Double Click，创建 ITextTripleClickStrategy 接口

```
public interface ITextTripleClickStrategy {  
    /**  
     * Invoked when triple clicking detected  
     */  
    public void tripleClicked(ITextViewer viewer);  
}
```

然后我定义了 `ExprTripleClickStrategy`，它实现了 `ITextTripleClickStrategy` 接口。为了简单起见，我不添加具体代码了，只是显示一个对话框表示三击事件被我们捕捉到了。三击的时候具体做什么，读者有兴趣可以自己完成。

配置

下一步就是让 JTF 知道我们的三击策略，不过 `SourceViewerConfiguration` 没有和三击有关的方法，我们可以模仿 `getDoubleClickStrategy` 的形式添加一个 `getTripleClickStrategy` 方法，然后修改 `ExprViewer` 的 `configure` 方法，把三击策略安装上去。其实就是一个哈希表，里面把文本类型和三击策略映射了起来。这些代码都是模仿 `SourceViewer` 中对双击策略的处理方式写的，所以就不一一列举了。

事件的触发

最后一步是触发三击事件，不然你装多少个三击策略也没用。从原理上说，三击事件就是鼠标双击之后又点了一下，我们可以监听双击事件，然后在下一次鼠标单击时检查其与双击事件的时间间隔，如果小于一个阈值，就触发三击事件。所以为了触发三击事件，我们需要给 `ExprViewer` 安装一个鼠标事件监听器：

清单4. 给 ExprViewer 添加三击事件触发机制

```
// How long we can wait for triple click after double click  
public static final long TRIPLE_CLICK_THRESHOLD = 500;  
  
private class TripleClickStrategyConnector extends MouseAdapter {  
    private long doubleClickTime;  
  
    public TripleClickStrategyConnector() {  
        doubleClickTime = 0;  
    }  
}
```

```
@Override
public void mouseDoubleClick(MouseEvent e) {
    doubleClickTime = System.currentTimeMillis();
}

@Override
public void mouseDown(MouseEvent e) {
    // compare time interval with threshold
    if (System.currentTimeMillis() - doubleClickTime <= TRIPLE_CLICK_THRESHOLD) {
        ITextTripleClickStrategy strategy = (ITextTripleClickStrategy) selectContentTypePlugin(
            getSelectedRange().x, tripleClickStrategies);
        if (strategy != null) {
            strategy.tripleClicked(ExprViewer.this);
        }
    }

    // clear double click time to avoid trigger triple click more than once
    doubleClickTime = 0;
}
}
```

上面的代码实现了我所描述的逻辑，如果检测到三击事件，我们通知 `ITextTripleClickStrategy`。到这里为止，我们的三击流程就跑完了。当然不止是三击，四击五击以至 **N** 击，就都可以这样实现了。

结束语

运行本文的例子之后会发现，三击后对话框确实出来了，但是文本选择区域发生了变化。这是因为 `StyledText` 内部做了一些事情，处理了我们的鼠标事件，如果完整的实现 `ITextTripleClickStrategy` 接口并设置我们想要的选择范围，这样的情况就不会出现了。至于三击之后可以做什么，我可以提供一个建议：以语句“`abcde = 12345;`”为例，双击 `abcde` 会选中 `abcde`，三击 `abcde` 则可以选中这条语句，即到分号为止的地方。我们可以从三击位置做一个扫描，直到前后碰到分号为止，有兴趣的读者可以尝试完善这个功能。

声明

本文仅代表作者的个人观点，不代表 IBM 的立场。

下载

描述	名字	大小	下载方法
第三小节示例代码	jtf.tutorial.part3.zip	1130KB	HTTP

[→ 关于下载方法的信息](#)

参考资料

- 如果对 ANTLR 有更多的兴趣，请参考“[ANTLR 项目主页](#)”

- “SWT 和 JFace, 第 1 部分: 简介” (developerWorks 中国, 2005 年 5 月) 介绍了 JFace 基础知识。

关于作者

马若劫是 Lotus Forms 部门的一位软件工程师, 主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验, 同时也是国内著名开源项目 LumaQQ 的创立者

第 1 段商标声明。 第 2 段商标声明。 其他公司、产品或服务的名称可能是其他公司的商标或服务标志。

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经 IBM 公司或原始作者的书面明确许可, 请勿转载。如果您希望转载, 请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 4 部分

Content Assistant

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

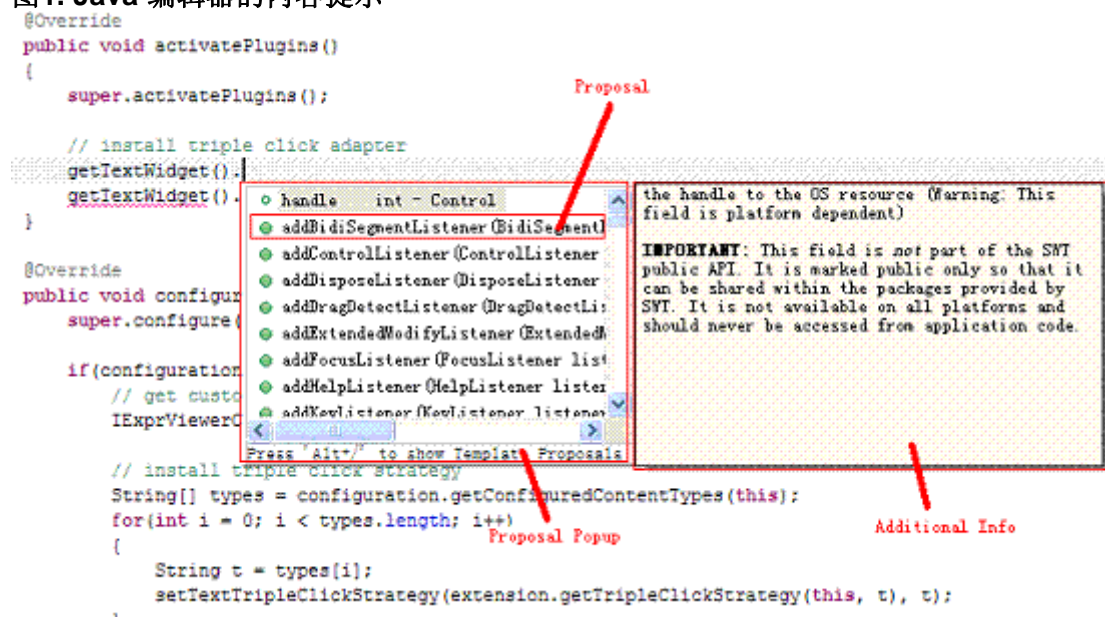
2008 年 3 月 27 日

Content Assistant (内容提示) 可以用来帮助程序员尽快完成源代码。可以说这个功能是一个代码编辑器必须具备的功能, 不然一个编辑器也无法称为代码编辑器了。本文探讨内容提示的相关概念并给出具体示例。

Content Assistant

Content Assistant (内容提示) 可以帮助程序员快速的完成代码, 并且还有代码自动补全的附加功能。这对于一个代码编辑器来说是至关重要的, 也是不少人喜欢用 IDE 编写代码的原因之一。但是这个功能背后却不是那么简单的, 我们先来了解一下 JTF 中和 Content Assistant 相关的概念, 下面是 Eclipse 中 Java 编辑器的内容提示的样子:

图1. Java 编辑器的内容提示



先来介绍一下图 1 中出现的三个概念:

- **Proposal (提议)**: Proposal 代表了一个可能的自动完成选项, 程序员选择之后, 代码会自动填入到编辑器里。
- **Proposal Popup (提议弹出列表)**: Proposal Popup 是用来显示自动完成列表的窗口
- **Additional Info (附加信息)**: 每个提议都可以附带一些帮助信息, 叫做 Additional Info, 它会显示在弹出列表的旁边, 并且当你选择某个 Proposal 的时候自动刷新。

这三个部分都是可以定制的, 只不过有的简单有点麻烦一点。比如我们看到弹出列表的下面有一行提示“Press ‘Alt+’ to show Template Proposals”, 这在标准的弹出列表里面是没有的, JDT 定制了这一部分。

提示: 在弹出列表出现后, 你可能会发现有些键盘事件被弹出列表处理了, 比如你按上下箭头, 它会改变当前被选择的 Proposal。这是因为在列表弹出之前, 内容提示管理器向文本框添加了一个按键校验事件处理器, 截获了这些按键。具体的代码可以参考 ContentAssistant 的内部类 InternalListener。

为示例代码添加内容提示支持

我打算为本文的示例代码添加以下的内容提示支持: 自动提示已经声明的变量名。比如下面的语句:

清单 1. 示例语句

```
a = 3;

b = 4;
```

那么当用户在激活内容提示时，我们将显示出 **a** 和 **b** 供它选择，也就是显示之前声明过的变量。所有的声明过的变量可以通过遍历语法树来得到，我们在 **TreeHelper** 里面有一个 **getVariables**，它会完成这样的功能，如果你生成的语法树不一样，调整这个方法就可以了。注意输入的时候语法必须是正确的，不然语法解析器识别不出这是一个声明语句，也就得不到变量了。

IContentAssistProcessor

第一步，我们要实现 **IContentAssistProcessor** 接口，它就是所有 **Proposal** 的来源。不过这个接口的方法比我们想象的要多一些：

清单 2. IContentAssistProcessor 接口

```
public interface IContentAssistProcessor {
    ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int offset);
    IContextInformation[] computeContextInformation(ITextViewer viewer, int offset);
    char[] getCompletionProposalAutoActivationCharacters();
    char[] getContextInformationAutoActivationCharacters();
    String getErrorMessage();
    IContextInformationValidator getContextInformationValidator();
}
```

这些方法牵涉到了一些概念，我们来一一的解释它们：

- **computeCompletionProposals**：这个就是所有 **Proposal** 的来源了，返回的类型是 **ICompletionProposal** 数组，**ICompletionProposal** 代表的就是单个的自动完成选项。
- **computeContextInformation**；**Context Information**（上下文信息）是个新概念，它在这里表示你选择了某个 **Proposal** 之后，会有一个提示信息弹出来，那个就叫上下文信息。要注意它和上面提到过的 **Additional Info** 是不同的东西。
- **getCompletionProposalAutoActivationCharacters**：这个方法引入了一个 **Auto Activation**（自动激活）的概念，所谓自动激活就是在某种条件下 **Proposal Popup** 自动弹出。这个“某种条件”指的是些字符，比如最常用的应该是“.”号。
- **getContextInformationAutoActivationCharacters**：上下文信息也有自动激活的功能
- **getErrorMessage**：如果内容提示无法找到任何 **Proposal**，它可以返回一个错误信息给用户
- **getContextInformationValidator**：上下文信息是可以进行校验的，如果失败，上下文信息不会被显示

computeCompletionProposals 方法显然是必须实现的，我添加了一个 **ExprContentAssistProcessor** 类，下面是它的实现方式：

清单 3. ExprContentAssistProcessor 实现了 IContentAssistProcessor 接口

```
public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int offset) {
    // get document
    IDocument doc = viewer.getDocument();

    // get tree
    Tree tree = TreeManager.getTree(doc);
    if(tree == null)
        return null;

    // get current selected range
    Point range = viewer.getSelectedRange();

    // get all declared variables
    List<String> variables = TreeHelper.getVariables(tree);

    // create proposals
```

```
List<ICompletionProposal> proposals = new ArrayList<ICompletionProposal>();
for(String var : variables) {
    proposals.add(new CompletionProposal (
        var, range.x, range.y, var.length(), null, var, null, "Add your info here"));
}
return proposals.toArray(new ICompletionProposal [proposals.size()]);
}
```

我们遍历语法树得到了所有的变量，你可以看到整个实现代码在 **ANTLR** 以及一些工具类的帮助下显得非常简洁。注意我们为每个变量创建了一个 **CompletionProposal**，它的构造函数参数非常多，最后一个就是 **Additional Info**，我这里只是填了一些无用的信息作为演示之用。其它的参数涉及自动完成需要的所有信息，比如插入的字符串，在哪里插入，图标等等。

配置

又到了将我们的实现和 **JTF** 连接起来的时间，还是修改 **ExprConfiguration**，要覆盖的方法变成了 **getContentAssistant**：

清单 4. 让 **JTF** 知道我们的内容提示实现

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer)
{
    ContentAssistant assistant = new ContentAssistant();
    assistant.setInformationControlCreator(new IInformationControlCreator() {
        public IInformationControl createInformationControl (Shell parent)
        {
            DefaultInformationControl control = new DefaultInformationControl (parent);
            return control;
        }
    });

    // add assist processor
    IContentAssistProcessor processor = new ExprContentAssistProcessor();
    assistant.setContentAssistProcessor(processor, IDocument.DEFAULT_CONTENT_TYPE);

    return assistant;
}
```

注意我们第一步实现的只是一个 **Processor**，还不是真正的内容提示管理器，幸运的是 **JTF** 为我们提供了 **ContentAssistant**，我们只要新建一个就可以了。第二行看上去有些不解，稍后我会解释。请注意最后一段，大家可以发现内容提示也是和文本类型绑定到一起的。

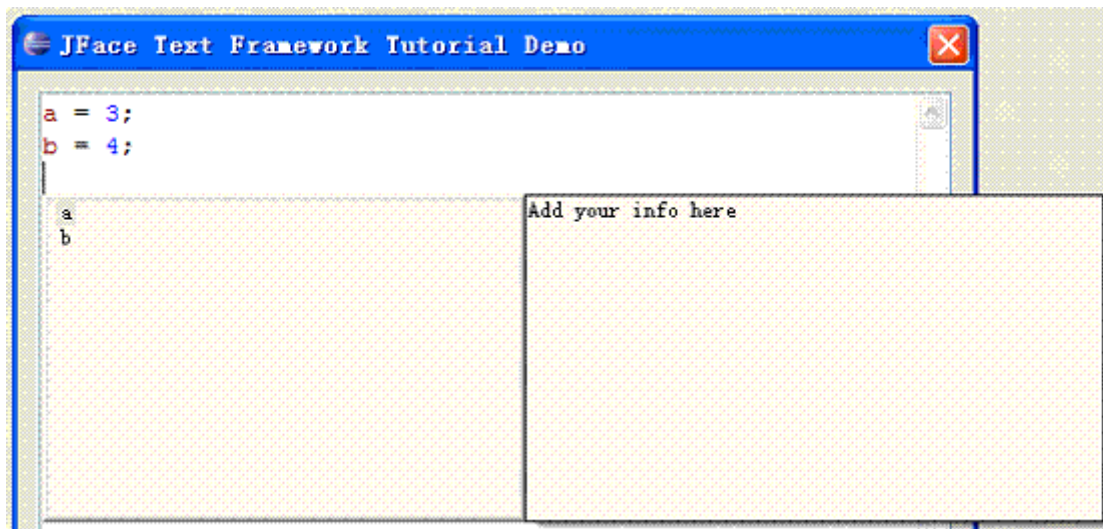
快捷键

用过 **Java** 编辑器的应该知道，内容提示可以用热键进行呼出，这个热键可以在 **Eclipse** 的设置里找到，以 **Eclipse 3.3** 为例，我们在设置中找到 **General->Keys**，然后在 **filter** 中输入 **Content Assist** 即可找到。为了能够让快捷键对我们的编辑器也有效果，需要安装一个 **Handle** 来处理它。这部分内容超出了本文的范围，所以我不详细解释了。大家可以发现 **ExprViewer** 中多了一些成员和方法，比如 **createHandlers** 方法，它们都是为了处理快捷键而准备的。

效果

到这里为止，一个很基本的内容提示就完成了，下图是它的效果：

图 2. 内容提示效果图



Information Control

回过头来看看上一节中我卖的关子：**ContentAssistant** 设置了一个 **InformationControlCreator**。从字面上很好理解，**Information Control**（信息控件）就是用来显示信息的一个控件，而 **InformationControlCreator** 就是创建控件的工厂了。信息控件可以用来显示任何信息，在内容提示的情况下，显示的就是 **Additional Info**。这个控件可以使用任何形式，那么里面的内容也就根据控件的能力可以有不同的变化。比如，你可以用一个浏览器控件来显示信息，这样的话，你的信息可以用 **HTML** 来写。在例子中，我们用的是 **JTF** 的缺省实现：**DefaultInformationControl**，它内部使用的是 **StyledText** 控件。它虽然用的不是浏览器，但是它内部提供了一个信息渲染接口：**InformationPresenter**。如果你使用 **HTMLTextPresenter**，它可以支持你在信息中嵌入 **HTML** 标签。

由于信息控件是一个通用的部件，它被广泛的用在其它需要显示信息的地方，比如我们以后会提到的 **Text Hover**（文本悬浮帮助）。同时由于 **JTF** 使用了一系列的接口来抽象信息控件的功能，因此可以很方便的实现自己的信息控件。

结束语

正如我所说，本文的例子是很基本的，有很多可以提高的地方，这些高级的功能留给有兴趣的读者完成。这里给出一些我能想到的问题以供参考：

- 内容提示只是显示所有的变量，它不会根据用户已经输入的内容来提示。比如有两个变量 **test** 和 **haha**，如果用户输入了“**te**”再激活内容提示，那么我们应该只提示 **test**。这个并非难事，我们有 **TokenList** 来帮助我们得到符号信息。
- 列出的 **Proposal** 没有图标，只有文字，这是一个小问题。学习了本文之后，你能立刻想起来要加个图标应该修改哪里吗？
- 对于 **Proposal Popup**：我们没有定制，可以尝试像 **Java** 编辑器那样给它底部加上些提示
- 对于信息控件，用的是缺省实现。可以尝试使用浏览器，然后使用 **HTML** 显示帮助信息，看上去效果会更好。
- 对于 **IContentAssistProcessor**，我们没有实现其它方法，比如上下文信息，自动激活。

要使内容提示功能达到和 **Java** 编辑器一样的高度，还是要花一些精力的。我一向提倡先了解基本概念，再深入具体细节。希望本文可以作为大家的起点，最终构造出一个专业的内容提示模块。

声明

本文仅代表作者的个人观点，不代表 **IBM** 的立场。

下载

描述	名字	大小	下载方法
第四小节示例代码	jtf.tutorial.part4.zip	1130KB	HTTP

→ [关于下载方法的信息](#)

参考资料

- 如果对 ANTLR 有更多的兴趣，请参考“[ANTLR 项目主页](#)”
- “[SWT 和 JFace，第 1 部分: 简介](#)”（developerWorks 中国，2005 年 5 月）介绍了 JFace 基础知识。

关于作者

马若劫是 Lotus Forms 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验，同时也是国内著名开源项目 LumaQQ 的创立者

第 1 段商标声明。 第 2 段商标声明。 其他公司、产品或服务的名称可能是其他公司的商标或服务标志。

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经 IBM 公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 6 部分: Text Hover 和 Annotation Hover

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

2008 年 4 月 10 日

Text Hover (文本悬浮) 和 **Annotation Hover** (标注悬浮) 是两种提供快速帮助的功能。本文介绍两种悬浮的基本概念和在 JTF 中的实现方式。

Text Hover

Text Hover (文本悬浮) 可以让用户快速的得到某种信息, 而不用打开相对缓慢的 **Eclipse** 帮助系统。**Eclipse** 的 **Java** 编辑器使用了文本悬浮来显示 **Javadoc** 帮助, 极大的方便了程序员。从功能上看, 文本悬浮可以认为是一种增强型的 **Tooltip**, 因为它支持显示更多的内容, 并且可以显示成各种样子。

文本悬浮看起来很酷, 但是原理却很简单。基本上就是三个步骤:

1. 根据鼠标位置得到字符偏移
2. 根据字符偏移得到字符所在单词和上下文信息
3. 根据这个单词以及上下文信息显示相关帮助

最关键的事情在于得到字符所在的单词并得到上下文信息, 这样才能决定该显示什么样的帮助信息。所以这就又回到了词法解析器和语法解析器的领域了, 只好略过不提。不过经过了这么多次的强调之后, 希望你对词法解析器和语法解析器的重要性有了深切的体会。它们是代码编辑器里最重要的基石。

第一步不用我们操心, **StyledText** 已经提供了这样的能力。我们要处理的是第二和第三步。对于文本悬浮, 相关的接口是 **ITextHover**; 对于标注悬浮, 相关的接口是 **IAnnotationHover**。

ITextHover

接下来我打算实现用文本悬浮显示变量值的功能, 比如对下面的代码:

清单1. 示例语言

```
                pa = 4;
b = 4;
a = b + 4;
```

如果你把鼠标移到 **a** 上, 则会显示“8”, 如果移到**b**上, 则会显示“4”, 是不是很酷呢?

修改解析器

这样的功能没有解析器的支持可不好做, 所以我又修改了 **Expr.g** 文件, 使解析器能够保存所有已经声明的变量和它们的值。**SharedParser** 也做了一定的修改, 提供了一个 **getVariableValue** 方法来根据变量名得到值。用 **ANTLR** 完成这些事情确实很简单, 我就不一一列出代码了。

实现 ITextHover

好消息是 **JTF** 已经提供了一个缺省的实现, 叫做 **DefaultTextHover**, 只要继承一下就可以了。下面是具体的代码:

清单2. ExprTextHover 继承 DefaultTextHover

```
public class ExprTextHover extends DefaultTextHover {
```

```

public ExprTextHover(I SourceViewer sourceViewer) {
    super(sourceViewer);
}

@Override
public IRegion getHoverRegion(ITextViewer textViewer, int offset) {
    return new Region(offset, 1);
}

@Override
public String getHoverInfo(ITextViewer textViewer, IRegion hoverRegion) {
    // query super first
    String info = super.getHoverInfo(textViewer, hoverRegion);

    // if null, use our logic
    if(info == null) {
        // get document
        IDocument doc = textViewer.getDocument();

        // get token list
        TokenList list = TokenManager.getTokenList(doc);

        // get token
        Token token = list.getToken(hoverRegion.getOffset());
        if(token == null)
            return null;

        // if token is variable, get variable value
        if(token.getType() == IExprTokens.ID) {
            // parse
            TreeManager.getTree(doc);
            return String.valueOf(SharedParser.getVariableValue(token.getText()));
        } else
            return null;
    } else
        return info;
}
}

```

主要的工作在 `getHoverInfo` 这个方法里面，我先调用了父类的 `getHoverInfo`，这一般来说是推荐的，因为父类的 `getHoverInfo` 会去首先查询相应位置处有没有一个标注，如果有，它会返回标注的信息。所以我在后面检查了父类的返回值，如果不是 `null` 且鼠标下面是一个变量名的话，就从解析器得到变量的值。

配置

最后一步是修改 `ExprConfiguration`，覆盖 `getTextHover` 方法，只是简单的返回我定义的 `ExprTextHover` 即可。

清单3. 让 JTF 知道我们的 Text Hover 实现

```

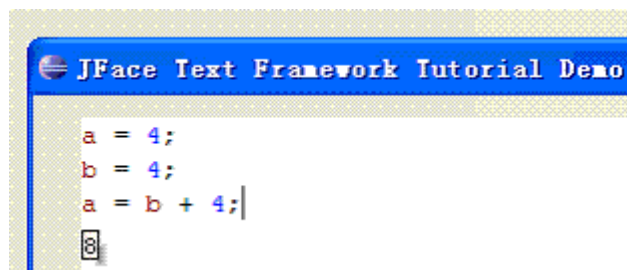
public ITextHover getTextHover(I SourceViewer sourceViewer, String contentType) {
    return new ExprTextHover(sourceViewer);
}

```

效果

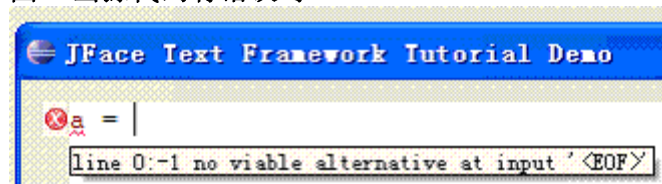
运行后，看看是不是得到了我想要的效果：

图1. 文本悬浮效果图



如我希望的那样，鼠标移动到 **a** 上面时，显示出了“8”。如果源代码中有错误，那么效果如下所示：

图2. 当源代码有错误时



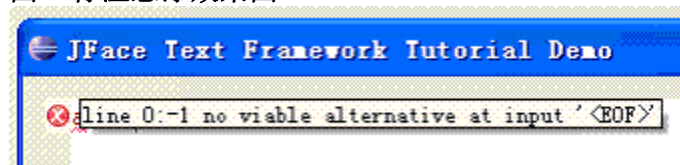
这正是由于我首先调用了父类的 `getHoverInfo` 的效果，父类为我们取出了标注信息。

IAnnotationHover

Annotation Hover（标注悬浮）只能给标尺上的标注提供悬浮帮助。在实现上，它比文本悬浮更简单一些，因为你不需要判断鼠标所在单词，也不需要想办法获取一些上下文信息。**JTF** 也提供了缺省实现：**DefaultAnnotationHover**。我在例子中继承了它，但是什么都没有覆盖。然后在 **ExprConfiguration** 中覆盖 `getAnnotationHover` 即可，和文本悬浮非常类似。

下面是标注悬浮的效果：

图3. 标注悬浮效果图



显示的信息和图2是一样的，因为是同一个标注。如果想返回不同的信息，覆盖 **DefaultAnnotationHover** 的 `getHoverInfo` 方法就可以了。

关于信息显示控件

我曾经在本系列的第四部分提过：**JTF**使用了一些接口来抽象信息显示方面的功能，比如 **IInformationControl** 代表了一个信息显示控件，**IInformationControlCreator** 代表了一个信息显示控件工厂。既然文本悬浮和标注悬浮也是显示信息，那么它们可以不可以定制信息显示控件呢？

答案是绝对没问题，但是你在 **ITextHover** 和 **IAnnotationHover** 找不到与之有关系的方法。是不是想起了什么？那些带有“Extension”字样的接口？Bingo! 看看 **ITextHoverExtension** 和 **IAnnotationHoverExtension** 吧，具体怎么做我就不演示了。

Accessibility

一个要达到产品级的软件，必须要考虑 **Accessibility** 的问题，所谓 **Accessibility**，简单的说，就是让残疾人也能够使用你的软件。**Accessibility** 有一个很基本的要求就是关键的文字都要能被读屏软件读出来。文

本悬浮和标注悬浮是很酷，但是它带来了 **Accessibility** 方面的问题，因为悬浮窗口没有焦点，读屏软件是不会读它们的。我们看看在 **Java** 编辑器中，它如何解决这个问题：

图4. **Java** 编辑器中的文本悬浮

```
*/
boolean canHandleMouseCursor();

/**
 * Return
 * hover
 * the s
 *
 * Returns whether the provided information control can interact with
 * the mouse cursor. I.e. the hover must implement custom information
 * control management
 *
 * @param sourceviewer the source viewer this hover is registered with
 */
```

可以看到 **Java** 编辑器的文本悬浮窗口下面有一个提示：按 **F2** 之后可以得到焦点。这个功能即方便了查看一些比较长的帮助，又解决了 **Accessibility** 的问题。

我这里不演示具体步骤，但是给有兴趣的读者一些提示：

1. 悬浮窗口下面的提示可以通过 **DefaultInformationControl** 的构造函数传进去，如果你用的其它的信息控件，则看具体情况。
2. 为了处理相应的快捷键，需要定义一个快捷键绑定，快捷键发生的时候调用相应的方法把信息控件变成可设置焦点的。**Eclipse**中有一些代码可以参考，比如 **InformationDispatchAction**，它是 **TextEditorAction** 的内部类。

AbstractInfor...

如果你是一个喜欢刨根问底的人，你可能会问：我虽然实现了 **TextHover** 接口，但是文本编辑器是怎么知道什么时候该调用我这个接口的呢？文本编辑器不是上帝，它当然是不知道的，需要外力帮助它知道。可以看看 **AbstractInformationControlManager** 这个类以及它的子类，会发现它有一个叫做 **TextViewerHoverManager** 的子类，是不是有点明白了呢？原来 **TextViewerHoverManager** 会安装在 **TextViewer** 上，它会监听悬浮事件，然后负责调用我们的实现。

从 **AbstractInformationControlManager** 可以看出一点：**JFace** 把信息显示功能包装成了通用的模块，并不是一定要在文本编辑器这样的场合才可以用悬浮信息窗口。如果你需要在其它地方添加类似的功能，可以继承 **AbstractInformationControlManager**，完成你自己的信息显示功能。

结束语

又给大家留下了可以发挥的地方，我小小的总结一下：

1. 没有提供自定义的信息显示控件
2. 没有实现悬浮窗口的可焦点化
3. 尝试为其它东西实现一个悬浮提示功能，比如工具条？

如果上面这些部分你都能完成的话，相信你就可以做出来非常专业的悬浮帮助系统了。

声明

本文仅代表作者的个人观点，不代表 **IBM** 的立场。

下载

描述	名字	大小	下载方法
第六小节示例代码	jtf.tutorial.part6.zip	1140KB	HTTP

→ [关于下载方法的信息](#)

参考资料

- 如果对 ANTLR 有更多的兴趣，请参考“[ANTLR 项目主页](#)”
- “[SWT 和 JFace，第 1 部分: 简介](#)”（developerWorks 中国，2005 年 5 月）介绍了 JFace 基础知识。

关于作者

马若劫是 Lotus Forms 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验，同时也是国内著名开源项目 LumaQQ 的创立者

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经 IBM 公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 5 部分: Text Decoration

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

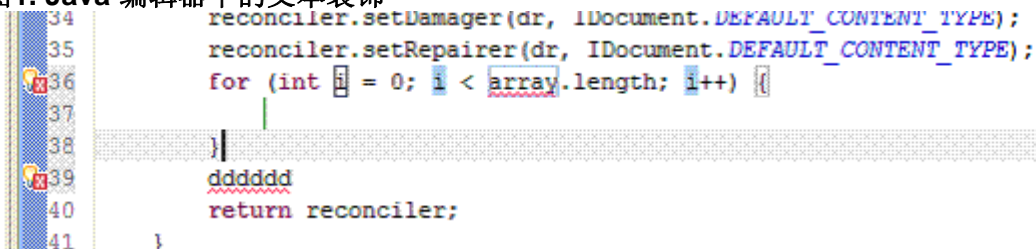
2008 年 4 月 10 日

Text Decoration (文本装饰) 是指对文本进行一些附加的视觉效果处理。本文介绍和文本装饰相关的概念并讨论如何在 JTF 中支持文本装饰。

Text Decoration

对于代码编辑器来说, Text Decoration (文本装饰) 是一个必需的功能。我们先来看看 Java 编辑器中对文本装饰的应用:

图1. Java 编辑器中的文本装饰



```
34 reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
35 reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);
36 for (int i = 0; i < array.length; i++) {
37
38 }
39 ddddd
40 return reconciler;
41 }
```

可以看到 Java 编辑器会把错误的部分用一个红色的波浪线标记出来, 还可以看到对于模版来说, 模版参数周围有一个小矩形, 而且在左边还有相应的错误图标显示。这些都叫做文本装饰。

我们在编辑代码的时候, 出错是不可避免的, 因此有了文本装饰功能之后, 我们可以快速的发现错误位置, 这对代码编辑是非常有用的。下面就来看一看要实现文本装饰功能都需要了解哪些概念。

提示: 模版会在以后的文章中进行介绍

Annotation

Annotation (标注) 这个术语已经使用的很泛滥了, 在很多地方都可以看到, 比如 J2SE 5.0 中也有 Annotation 的概念。放到 JTF 中来讲, 标注指的是和某块文本区域绑定的特定信息, 至于特定信息是什么, 是可以自定义的。可以观察一下 JTF 中 Annotation 这个类, 可以看到它有类型信息, 有附加的文本信息, 同时它还有很多子类, 而且子类也都包含了一些扩展信息。

标注包含类型信息是很重要的, 通过类型, 就可以知道某个标注的目的和作用了, 然后可以在界面上反映出来。图 1 中的红色波浪线, 实际上是因为在那个位置上有一个错误类型的标注。

Ruler

Ruler (标尺) 是显示辅助信息的一块区域, 它一般附着在编辑器的周围。比如图1中那两个表示错误的图标就是显示在 VerticalRuler (垂直标尺) 上的。标尺可以多于一个, 在图 1 中可以看到另外一个用来显示行号的标尺, 需要多少个标尺可以定制的。

IAnnotationPresentation

名称里面带了“Presentation”字样的都和界面渲染有关, 而 IAnnotationPresentation 就是用来渲染标注的。这个接口只是负责在标尺上绘制标注, 图1中表示错误的图标就是出自这里。如果你自己定义的标注没有实现这个接口, 那么就不会出现在标尺上了。

IAnnotationModel

IAnnotationModel 是用来管理标注的。它负责管理某个文档上所有的标注, 并且负责触发相应的标注事件, 比如添加, 删除等等。也许这个接口叫做 IAnnotationManager 更合适些。由于 Eclipse 不断在发展中, 它的 API 时常需要增强, 但是为了不破坏与旧版本的兼容性, 这些扩展一般都是以扩展接口的方式出现的。对于 IAnnotationModel 来说, 你可以发现还有一个 IAnnotationModelExtension 接口, 里面有一

些更强更方便的方法。不光是 `IAnnotationModel`，你在很多地方都可以发现名字是 `Extension` 结尾的接口，所以请记住这种现象，当我们浏览代码的时候，不要以为某个接口就已经是全部了。

IAnnotationAccess

这个接口可以用来访问标注的一些信息。但是在 3.0 之后，标注的信息都可以通过 `Annotation` 类访问到，所以这个接口已经过时了，只是为了向下兼容而保留，我就不罗嗦了。

AnnotationPainter

这是 JTF 中和标注绘制相关的重要类。上面提到的 `IAnnotationPresentation` 负责在标尺上绘制标注，而它是负责在编辑器（也就是 `StyledText`）里面绘制标注的。我们看到的红色波浪线和矩形框就归它负责。

IDrawingStrategy

这个接口是定义在 `AnnotationPainter` 内部的，它是绘画工作真正完成的地方。`AnnotationPainter` 实际上只是调用它来完成绘图操作而已。如果你在 `Hierarchy` 视图中看一下 `IDrawingStrategy`，会发现它有很多实现类。那个红色的波浪线就是 `SquigglesStrategy` 的杰作。

AnnotationPreference

不要以为什么样的标注画成什么样子是规定死了的，实际上它们都是通过 Eclipse 的 Preference 系统来控制的。请打开 Eclipse 的 Preference 设置，找到 `General->Editors->Text Editors->Annotations`，你会发现原来这里都可以控制。而这些选项就被包装在了 `AnnotationPreference` 中。如果你想添加一种标注的配置信息，可以通过程序方式，也可以通过 `org.eclipse.ui.editors.markerAnnotationSpecification` 扩展点。

MarkerAnnotationPreferences

`AnnotationPreference` 包装的只是一种标注的配置信息，要得到全部的信息，可以通过 `MarkerAnnotationPreferences` 类。

实现文本装饰功能

我打算实现一个语法错误检查的功能，如果用户输入的源代码有问题，则把它标记出来。让我们一步步开始这个过程。

SourceViewerDecorationSupport

不得不说和文本装饰相关的概念、接口非常的多，而我列举的只是一些最重要的，还不是全部。这么复杂的功能，要直接写到 `SourceViewer` 里面去不免显的臃肿，所以 `SourceViewer` 通过 `SourceViewerDecorationSupport` 来支持文本装饰功能，避免了把所有代码都塞到一个类里面去。

我在本系列第一部分提过，之所以没有用 Eclipse 标准 `editors` 扩展点来做演示，是因为 `Editor` 本身为我们隐藏了一些东西，文本装饰的支持就是其中之一。现在我们手动添加文本装饰的支持，在 `ExprViewer` 中增加一个 `configureDecorationSupport` 方法并在 `configure` 方法中调用它。代码如下：

清单 1. 在 ExprViewer 中使用 SourceViewerDecorationSupport

```
protected void configureDecorationSupport()
{
    // create support object
    decorationSupport = new SourceViewerDecorationSupport(
        this, null, new DefaultMarkerAnnotationAccess(), ColorManager.getInstance());

    // add other annotation preference
    MarkerAnnotationPreferences prefs = new MarkerAnnotationPreferences();
    MarkerAnnotationPreferences.initializeDefaultValues(
        Activator.getDefault().getPreferenceStore());
    Iterator<Object> e = prefs.getAnnotationPreferences().iterator();
    while(e.hasNext()) {
        // add to support
    }
}
```

```

        decorationSupport.setAnnotationPreference((AnnotationPreference)e.next());
    }

    // install support
    decorationSupport.install(Activator.getDefault().getPreferenceStore());
}

```

我没有增加自定义的标注，所以这个方法并不复杂。如果你有自己的标注类型，那么可以用程序方式或者扩展方式定义自己的标注配置信息，然后调用 **SourceViewerDecorationSupport** 的 **setAnnotationPreference** 方法将你的配置添加到库中

错误检测支持

下一步再一次和 **ANTLR** 扯上了关系，如果没有解析器的支持，要知道源代码中的错误恐怕有点困难。幸好我们已经有了很多基础工具，但是仍然有点不够。我稍微修改了一下 **ANTLR** 的文法，支持把错误保存到一个 **Map** 中。这样解析完成后，我就可以直接从解析器得到错误列表了。同时 **SharedParser** 做了少许增强，以便得到最近一次解析时的错误信息。这些修改就不一一列出了，总之我现在具有了得到错误位置与错误信息的能力。

扩展 Annotation

下一步是扩展 **Annotation** 类，因为 **Annotation** 类并没有实现 **IAnnotationPresentation**。我想让错误也在标尺上显示出来，所以要扩展 **Annotation**。我定义了 **ExprAnnotation**，它的代码的核心是 **paint** 方法，作用是在标尺上画一个背景，然后再画一个错误图标。至于画图的具体细节没有必要关心，熟悉 **SWT** 的 **GC** 使用方式的话应该不是问题，这里我就略过不提了。

Ruler 支持

现在的编辑器还没有一个标尺，所以稍微修改一下 **JTFDialog**，传给 **ExprViewer** 的构造函数一个 **VerticalRuler** 实例即可。

触发语法检查

万事具备，只欠东风。现在只要找到一个合适的时机进行语法检查，并安装我们的标注就可以了。文本改变事件是一个选择，也是最简单的选择，所以我新建了一个 **SyntaxChecker** 类，并把这个类注册为文本事件监听器。让我们看看文本改变时它会做些什么：

提示：注意我们的布局代码也发生了变化，原来是 **viewer.getTextWidget().setLayoutData(...)**，现在是 **viewer.getControl().setLayoutData(...)**。因为安装了标尺之后，**viewer** 内部会对布局做了一些调整，这个时候 **StyledText** 不再是顶层控件，所以要使用 **getControl**。如果不了解的话，界面的布局会出乎意料，恐怕一时半会也不知道问题从何而来。

清单2. SyntaxChecker

```

    public void documentChanged(DocumentEvent event) {
        // get model
        IAnnotationModel model = viewer.getAnnotationModel();
        if(model == null || !(model instanceof IAnnotationModelExtension))
            return;

        // create map contains annotations to be added
        Map<Annotation, Position> toBeAdded = new HashMap<Annotation, Position>();

        // get annotations to be removed
        Annotation[] toBeRemoved = getAnnotations(new String[] {
            "org.eclipse.ui.workbench.texteditor.error"
        });

        // get document
        IDocument doc = event.getDocument();

        // parse it
        TreeManager.getTree(doc);

        // get errors
        Map<Token, String> errors = SharedParser.getLastErrors();
    }

```



```
// add annotation
for(Token token : errors.keySet()) {
    CommonToken ct = (CommonToken)token;
    Annotation anno = new ExprAnnotation("org.eclipse.ui.workbench.texteditor.error",
        errors.get(token));
    Position pos = new Position(
        ct.getStartIndex(), ct.getStopIndex() - ct.getStartIndex() + 1);
    toBeAdded.put(anno, pos);
}

// replace annotation one time, this provides a better performance
// than remove/add one by one
((IAnnotationModelExtension)model).replaceAnnotations(toBeRemoved, toBeAdded);
}
```

我只列出了这个最关键的方法。它的基本流程是：

1. 做一些必要的检查
2. 得到之前已经存在的错误标注
3. 解析源代码，得到错误列表
4. 为每个错误创建一个 **ExprAnnotation**，并把标注类型设为错误类型
5. 通过 **replaceAnnotations** 批量刷新标注

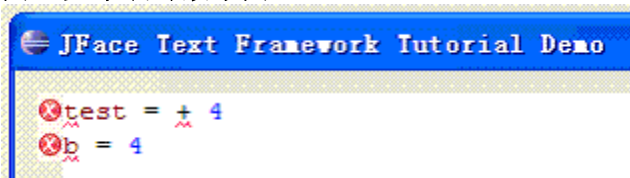
需要解释一下的是第 5 步，为什么要用 **replaceAnnotations**？主要还是出于性能的考虑，因为你每次添加删除一个标注，都会触发一系列的连锁反应，所以如果你一个个的添加删除，速度会非常慢。

replaceAnnotations 解决了这个问题，因为它是一次全部替换。但是这个方法是 **IAnnotationModelExtension** 接口里面的，为了保险起见，需要先检查类型。当然我们这个例子很简单，并没有显著的性能问题。

效果

下面是运行后的效果，假如我没有输入分号，错误的标注就显示出来了。

图2. 文本装饰效果图



结束语

让我来指出还是做的不够的地方吧：

1. 错误检测太过简单，比如不能标注出未声明的变量。要做一个专业的代码编辑器的话，错误检测可不能这样简单。
2. 语法检查是在文本内容发生变化后立刻触发，当源代码越来越多的时候，这很可能造成性能问题。我们可以从很多方面想办法，比如提高语法解析器的效率，或者减少不必要的语法检查次数，可以考虑用定时器，当用户隔一段时间没有输入动作时再进行语法检查。最好的办法是使用 **Reconciler**，关于 **Reconciler** 的概念会在本系列的第 11 部分中提到。
3. 我没有自定义标注类型，用的是 **Eclipse** 自带的类型。那么不妨尝试一下定义自己的标注类型，可以通过 **org.eclipse.ui.editors.annotationTypes** 扩展点。
4. 我也没有自定义标注的渲染方式，用 **IDrawingStrategy** 尝试画一个很酷的标注吧。要分两步走，实现 **IDrawingStrategy**，然后注册你的 **IDrawingStrategy**。
5. 我用的是标准的垂直标尺，尝试实现一个自己的标尺，把标注画到上面。
6. 标注只有界面上的提示，没有任何文字上的信息，用户很难知道到底是什么错误。不过不要担心，这是因为我没有提到 **Text Hover**（文本悬浮帮助），以后的文章将完善这个功能。

这些不足的地方留给有兴趣的读者。

声明

本文仅代表作者的个人观点，不代表 IBM 的立场。

下载

描述	名字	大小	下载方法
第五小节示例代码	jtf.tutorial.part5.zip	1140KB	HTTP

[→ 关于下载方法的信息](#)

参考资料

- 如果对 ANTLR 有更多的兴趣，请参考[“ANTLR 项目主页”](#)
- [“SWT 和 JFace，第 1 部分: 简介”](#)（developerWorks 中国，2005 年 5 月）介绍了 JFace 基础知识。

关于作者

马若劼是 Lotus Forms 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验，同时也是国内著名开源项目 LumaQQ 的创立者

第 1 段商标声明。 第 2 段商标声明。 其他公司、产品或服务的名称可能是其他公司的商标或服务标志。

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经 IBM 公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 7 部分: Quick Assistant

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

2008 年 4 月 24 日

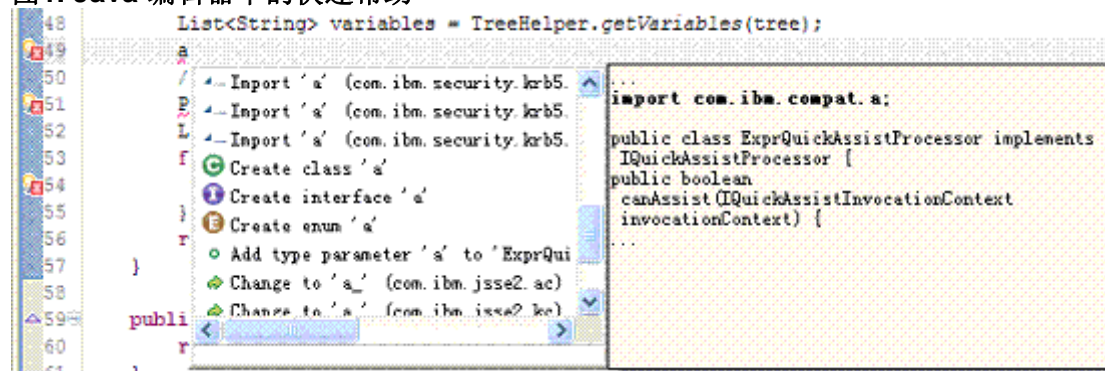
Quick Assistant (快速帮助) 的基本用途是为源代码中的错误提供一些快速的解决方案, 它和 Content Assistant (内容提供) 虽目的不同, 但架构类似。本文介绍如何快速帮助的概念和实现方法。

Quick Assistant

Quick Assistant (快速帮助) 的基本用途是为源代码中的错误提供一些快速的解决方案。快速的意思是指这个方案足够简单或者足够模式化, 可以由程序帮你自动完成。当然快速帮助是无法解决深层次的问题的, 不过一般我们在编写代码的时候, 犯的最多的都是一些小错误, 所以快速帮助是个非常有用的功能。

在 Java 编辑器中, 快速帮助看上去就是下图的样子:

图1. Java 编辑器中的快速帮助



可见, 不管是从名字上, 还是界面上, 快速帮助都非常类似我提过的 Content Assistant (内容提示) 功能。实际上, 它们的架构和实现方式也差不多。

提示: 如果还不了解内容提示的概念, 参见本系列第四部分了解更多信息。

快速帮助是基于 Annotation (标注) 的, 我们已经在本系列第五部分中介绍了如何创建一个标注并显示出来。标注包含一个类型信息, 比如错误或者警告或者只是提示。快速帮助的基本想法就是判断光标所在位置有没有标注, 如果有则检查标注的类型, 如果是你感兴趣的类型, 比如错误, 则触发快速帮助。

由于其和内容提示的相似性, 我就不废话了, 让我们直接看看如何实现快速帮助吧。

实现快速帮助

在目前的例子里, 我已经把语法错误显示出来了。因为快速帮助可以针对标注类型来触发, 所以我打算增加一种错误类型: Undeclared Variable (未声明的变量), 比如在下面的例子中:

清单1. 错误的语法, 变量 b 未声明

```
a = 3;  
a = b;
```

a 被赋值了两次, 第一次是用常量, 第二次是把变量b的值赋给变量a。如果根据目前的解析器文法来说, 这段代码的语法没有问题, 但是语义有问题, 因为 b 没有声明过。

对于这种新的错误类型，我会认为用户也许是敲错了变量名，所以会显示出声明过的变量名列表，如果用户选择了一个，则未声明的变量被替换成声明过的变量。对于原来的错误，我不提供快速帮助，这样大家就可以看出差别了。

实现底层支持

要增加错误类型，又需要修改解析器文法，我已经完成这部分，所有未声明的变量都被保存到了一个列表里面。同样，又增强了 **SharedParser** 以便得到这些未声明的变量。

添加标注类型

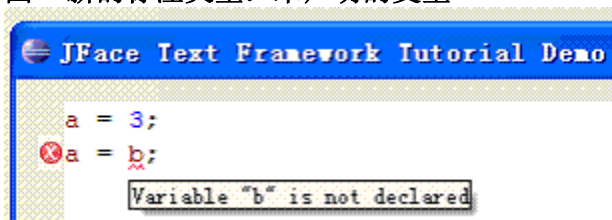
我不能还是使用 `org.eclipse.ui.workbench.texteditor.error` 这样的标注类型，需要有所区别才行。所以我通过 `org.eclipse.ui.editors.annotationTypes` 扩展点添加了一个标注类型，它的父类还是 `org.eclipse.ui.workbench.texteditor.error`，这样的话它就可以继承父类的一些设置，不用我去创建 `AnnotationPreference` 了。扩展的声明如下：

清单2. 错误的语法，变量 **b** 未声明

```
<extension
  point="org.eclipse.ui.editors.annotationTypes">
<type
  name="jtf.tutorial.annotation undeclared.variable"
  super="org.eclipse.ui.workbench.texteditor.error">
</type>
</extension>
```

然后我在 **SyntaxChecker** 中为每个未声明的变量创建了这个类型的标注，因为有了底层支持，这个过程很简单，就不贴出代码了。我们看看实际的效果：

图2. 新的标注类型：未声明的变量



IQuickAssistProcessor

前面两步是把准备工作做完了，现在才轮到真正的接口上场了。**IQuickAssistProcessor** 的角色和 **IContentAssistProcessor** 的角色是完全相同的，只是具体方法不同罢了。下面是 **IQuickAssistProcessor** 的声明：

清单3. IQuickAssistProcessor 接口

```
public interface IQuickAssistProcessor {
    String getErrorMessage();

    boolean canFix(Annotation annotation);

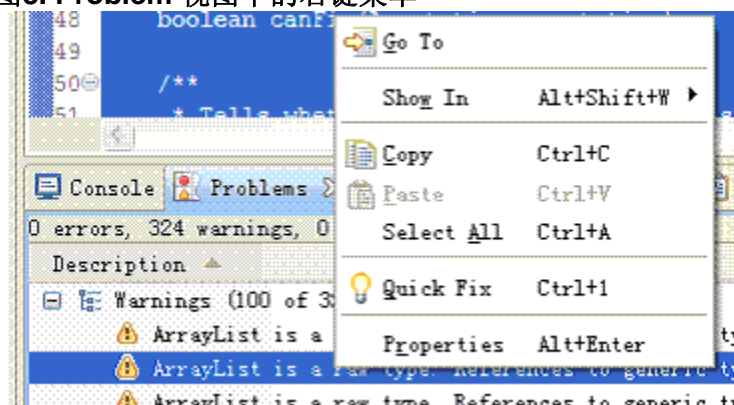
    boolean canAssist(IQuickAssistInvocationContext invocationContext);

    ICompletionProposal[] computeQuickAssistProposals(
        IQuickAssistInvocationContext invocationContext);
}
```

快速帮助比内容提示多了一个上下文相关的接口：**IQuickAssistInvocationContext**。它可以用来为快速帮助提供一些辅助信息。实际上，内容提示也是需要上下文信息的，但是 **JTF** 没有另外做一个接口来包装它，如果你要做一个复杂的内容提示功能，往往需要自己来定义一个上下文信息接口。

`canFix` 方法在内容提示中也没有类似物。这个方法主要是提供给 **Marker** 系统来使用的，比如下图中所示的右键菜单：

图3. Problem 视图中的右键菜单



当你在 **Problem** 视图中右键点击一个错误的时候，弹出菜单中有一个 **Quick Fix** 的菜单项，如果 `canFix()` 返回 `false`，那么这个菜单项就会变灰而不可用了。由于我的例子中没有创建 **Marker**，所以这个方法用处有限。

提示：如果不了解 **Marker** 是什么，可以参考 Eclipse.org 的文章: [Mark My Words](#)

`canAssist` 方法给了你一个检查不同上下文的机会，这样你就可以通过不同的上下文信息激活不同的快速帮助了。`computeQuickAssistProposals` 方法比较简单，除了名字和参数不同，它和内容提示中的 `computeCompletionProposals` 基本类似。

其它的方法都和内容提示类似，来看看例子中是怎么实现这个接口的，我只列出了 `computeQuickAssistProposals`，其它方法的实现都非常简短，在此省略。

清单4. ExprQuickAssistProcessor 实现了 IQuickAssistProcessor 接口

```
public class ExprQuickAssistProcessor implements IQuickAssistProcessor {
    // other code
    ...

    public ICompletionProposal[] computeQuickAssistProposals(
        IQuickAssistInvocationContext invocationContext) {
        // get viewer
        ISourceViewer viewer = invocationContext.getSourceViewer();

        // get annotation
        Annotation anno = getAnnotation(viewer, invocationContext.getOffset());
        if(anno == null || !canFix(anno))
            return null;

        // get doc
        IDocument doc = viewer.getDocument();

        // get tree
        Tree tree = TreeManager.getTree(doc);
        if(tree == null)
            return null;

        // get all declared variables
        List<String> variables = TreeHelper.getVariables(tree);

        // create proposals
        Position pos = viewer.getAnnotationModel().getPosition(anno);
        List<ICompletionProposal> proposals = new ArrayList<ICompletionProposal>();
        for(String var : variables) {
            proposals.add(new CompletionProposal(
                var, pos.getOffset(), pos.getLength(), var.length(),
```



```
        null, var, null, "Add your info here"));  
    }  
    return proposals.toArray(new ICompletionProposal[proposals.size()]);  
}  
}
```

这段代码和内容提示中的很相似，不同点是这里会去取得当前光标处的标注，然后用 `canFix` 方法检查标注是否可以被修正。最后，还是返回一个 `Proposal` 的数组。

配置

最后一步，几乎是把 `ExprConfiguration` 的 `getContentAssistant` 方法复制了一遍，只不过那些接口换成了快速帮助的。大家可以对比一下 `getQuickAssistAssistant` 和 `getContentAssistant` 有什么不同。提醒大家注意一下：快速帮助并没有和文本类型绑定在一起。这是合理的，因为快速帮助一般是用来修正错误的，在一个错误的源代码里面，可能很难知道某个地方到底应该是什么样的文本类型，因此根据文本类型来注册快速帮助功能是有问题的。除了不同点，看到更多的应该是共同点，比如相似的 `Assistant/Processor` 结构，相似的信息显示控件。注意，又是信息显示控件，到现在为止，已经有内容提示，快速帮助，文本悬浮和标注悬浮都使用了它。

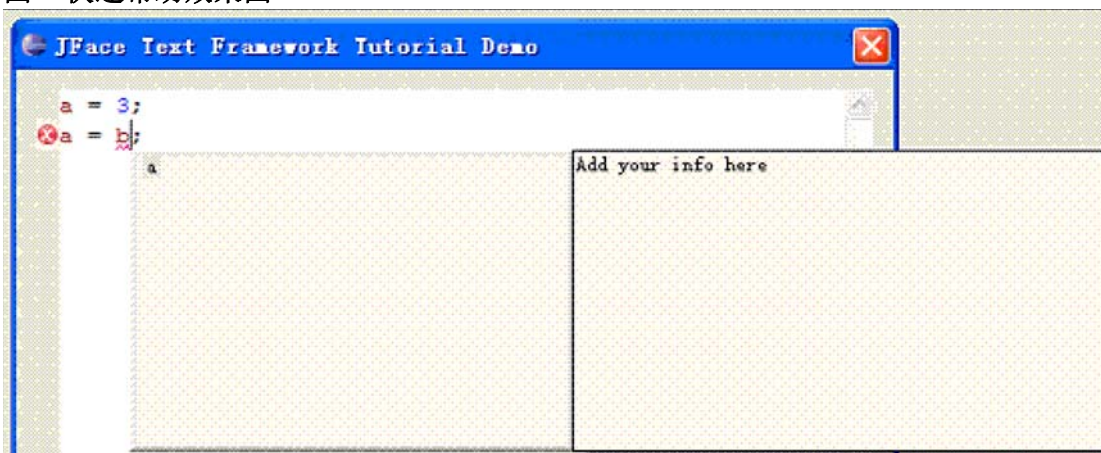
快捷键处理

在 `Eclipse` 里面，触发快速帮助的缺省快捷键是 `Ctrl+1`，和内容提示一样，需要给快速帮助安装一个快捷键处理器，由于我已经在演示内容提示的时候打完了基础，现在只要改改 `ExprViewer` 的 `createHandlers`，添加一个 `handler` 就可以了。

效果

当你将光标放在未声明变量上，再按 `Ctrl+1` 时，可以看到弹出的框里面列出了所有声明过的变量，选择一个之后，未声明的变量会被替换。如下图所示：

图4. 快速帮助效果图



读者可以尝试修正一下其它类型的错误，不会有反应，因为 `ExprQuickAssistProcessor` 里面只接受“未声明变量”这个标注类型

结束语

我没有实现 `IQuickAssistInvocationContext` 接口，如果是像 `Java` 编辑器这样复杂的应用，那就很可能需要了，大家可以想想，如何利用好这个接口。本文还提到了在 `Problem` 视图中有 `Quick Fix` 的菜单项，大家不妨尝试将编辑器中的错误添加到 `Problem` 视图中，然后使用快速帮助功能。

声明

本文仅代表作者的个人观点，不代表 IBM 的立场。

下载

描述	名字	大小	下载方法
第七小节示例代码	jtf.tutorial.part7.zip	1140KB	HTTP

→ [关于下载方法的信息](#)

参考资料

- 如果对 ANTLR 有更多的兴趣，请参考“[ANTLR 项目主页](#)”
- “[SWT 和 JFace，第 1 部分: 简介](#)”（developerWorks 中国，2005 年 5 月）介绍了 JFace 基础知识。

关于作者

马若劼是 Lotus Forms 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验，同时也是国内著名开源项目 LumaQQ 的创立者

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经 IBM 公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 8 部分: Hyperlink

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

2008 年 4 月 24 日

Hyperlink (超链接) 一般用来在编辑器中实现快速的代码定位, 当然它并不局限与此, 你可以自定义超链接的行为。本文介绍超链接的概念和在 JTF 中的实现方式。

Hyperlink

Hyperlink (超链接) 在 Java 编辑器中用来进行快速的代码定位, 当你按住 **Ctrl** 键并把鼠标指向一个函数名的时候, 函数名会显示为超链接, 点击之后代码会跳转到函数的声明处。这个功能使得 Eclipse 浏览代码很方便, 这次我就来介绍如何在自己的编辑器中添加超链接功能。

超链接的定位

编辑器不会知道哪块区域应该显示为超链接, 这是通过 `IHyperlinkDetector` 接口实现的。这里牵涉到语义方面的内容, 因为你必须要能知道鼠标下面到底是个什么。在解析器那一层需要实现这样的支持。

超链接的渲染

JTF 是如何显示超链接的? 可能你会想到标注, 缺省的实现不是这样。这里要介绍另外一个接口: `IHyperlinkPresenter`。JTF 缺省的时候是用 `StyledText` 的 `StyleRange` 实现的, 其实就是把超链接的那块文字置为蓝色且带下划线。但是因为有了这么一个接口, 你可以把超链接弄成任何样子。

实现超链接

本文要实现的超链接功能是: 点击某个变量名, 编辑器会选中声明该变量的那条语句。

底层支持

底层需要支持两个功能: 判断某个位置是一个变量, 以及得到变量声明的语句范围。由于我的例子很简单, 判断是不是变量也非常简单, 只要符号类型是一个 `ID` 类型就行了。得到变量声明的语句范围需要检查语法树, 因为变量声明的子树以等号为根节点, 所以找到对应的根节点就行了。然后从等号开始得到子树的最左和最右节点, 从而计算出整个子树的字符范围。这些代码已经添加到了 `TreeHelper` 中, 具体请参看 `getVariableDeclaration` 和 `getTreeRange` 方法。

实现 `IHyperlink`

我实现了一个 `VariableHyperlink` 来封装超链接信息, 它最重要的方法是 `open()`, 因为它会在你点击超链接后被调用:

清单1. `VariableHyperlink` 的 `open` 方法

```
public void open() {
    // get doc
    IDocument doc = viewer.getDocument();

    // get tree
    Tree tree = TreeManager.getTree(doc);

    // get variable declaration range
    Point range = TreeHelper.getVariableDeclaration(tree, variable);
}
```

```
// select text
if(range != null) {
    viewer.setSelectedRange(range.x, range.y);
    viewer.revealRange(range.x, range.y);
}
```

我用到了刚才提到的 **TreeHelper** 中的方法来得到声明语句的范围，剩下的事情就比较直接了，选择这个范围并确保其在编辑器中可见。

实现 **IHyperlinkDetector**

ExprHyperlinkDetector 完成了发现变量的功能，然后把变量信息包装在 **VariableHyperlink** 中。在之前的文章中，我已经不止一次的使用了 **TokenList** 来得到某个偏移处的符号，**ExprHyperlinkDetector** 也依赖于 **TokenList**，所以这里不详细解释了。

配置

覆盖 **ExprConfiguration** 的 **getHyperlinkDetectors**，让它返回 **ExprHyperlinkDetector**。它的返回值是一个数组，所以你可以安装多个 **IHyperlinkDetector** 实例。对于这个简单的小例子，并无必要使用多个 **IHyperlinkDetector** 实例。

结束语

超链接是 **JTF** 中相对简单的一个功能了，但是不要忘了，我是没有实现全部的功能的：

1. 是不是一定需要按住 **Ctrl** 键才能激活超链接功能呢？不是，请看看 **SourceViewerConfiguration** 的 **getHyperlinkStateMask** 方法。
2. 试试看使用 **IHyperlinkPresenter** 实现一个自定义的超链接样式，这里牵涉到 **HyperlinkManager** 这个类的使用。这个功能较为复杂一些，需要这样做的时候也不多，没有时间的读者不妨看看 **DefaultHyperlinkPresenter** 类的实现。
3. 例子还是有 **bug** 的，比如如果一个变量声明了两次，则只会选择第一条声明语句。可以考虑显示一个浮动窗口来选择跳转到哪个，你想到了什么解决方案呢？

声明

本文仅代表作者的个人观点，不代表IBM的立场。

下载

描述	名字	大小	下载方法
第八小节示例代码	jtf.tutorial.part8.zip	1140KB	HTTP

→ [关于下载方法的信息](#)

参考资料

- 如果对 **ANTLR** 有更多的兴趣，请参考“[ANTLR 项目主页](#)”
- “[SWT 和 JFace，第 1 部分: 简介](#)”（**developerWorks 中国**，2005 年 5 月）介绍了 **JFace** 基础知

识。

关于作者

马若劫是 **Lotus Forms** 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 **Eclipse** 和 **Java** 方面有多年研发经验，同时也是国内著名开源项目 **LumaQQ** 的创立者

IBM 公司保留在 **developerWorks** 网站上发表的内容的著作权。未经IBM公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。

基于 JFace Text Framework 构建全功能代码编辑器:

第 9 部分: Template

级别: 中级

马若劼 (maruojie@cn.ibm.com), 软件工程师, IBM 中国软件开发中心

2008 年 5 月 08 日

Template (模版) 是可以用来快速添加某种固定形式的代码, 提高代码编辑的速度。模版和 JTF 的其它特性有或多或少的联系, 比如内容提示, 比如标注。本文介绍模版的相关概念, 并给出一个简单的实现。

Template

Template (模版) 可以用来快速添加某种固定形式的代码, 中间还可以插入参数。对于 Java 编辑器来说, 你可以在 Eclipse 的设置中找到相应的属性页, 路径是 **General->Java->Editor->Templates**。这个属性页是 Eclipse 标准的模版属性页, 因为它做的比较完善, 所以一般不需要自己写一个。

仔细探索一下这个属性页, 尝试编辑一下模版, 你可能会发现很多不了解的概念, 下面我会一一解释

模版属性

模版包含一些基本属性:

- **Name** (名称): 这个模版的名字, 必须唯一
- **Context** (上下文): 上下文是指这个模版可以在什么位置或者什么情况下出现, 它相当于对模版做了一个粗略的分类。这个属性可以方便你在某些时候隐藏掉不需要显示的模版。
- **Description** (描述): 一段详细介绍模版功能的文字, 还记得内容提示的时候可以通过 **InformationControl** 显示一些帮助信息吗? 这个属性很适合显示在那里。
- **Auto Insert** (自动插入): 当用户的输入只可能定位到一个模版的时候, 是否自动把这个模版插入到编辑器中。不然还是提示用户选择一个模版插入, 即使只有一个模版。

模版的定义

模版可以通过扩展的方式定义, 扩展点是 **org.eclipse.ui.editors.templates**, 也可以通过属性页手动添加, 也可以通过程序方式添加。通过扩展点添加的模版, 可以认为是“静态模版”, 即缺省就存在的, 后面两种方式则更灵活一些, 但是稍微麻烦一点。

模版的存储

模版是需要持久化的, 它最终会被存放到一个 XML 文件里面。我们并不需要知道这个 XML 文件在哪里, 格式是什么, 这些事情被 **TemplateStore** 封装了, 我们直接用它就行。在装载的模版的过程中, 既要装载通过扩展点定义的模版, 又要装载用户手动添加的模版, **TemplateStore** 只能装载通过扩展点定义的模版, 所以一般是使用 **TemplateStore** 的子类 **ContributionTemplateStore**, 它提供了装载用户自定义模版的能力。

模版参数

模版中可以嵌入参数, 即我们看到的 **\${arg}** 的形式。**JTF** 缺省定义了一些参数, 程序员也可以自己定义参数。参数大致可以分为两种: 自动解析式和输入式。自动解析的参数有 **time**, **date** 等等, 这类参数在插入模版到编辑器的时候, 会自动替换成相应的时间, 日期等等。输入式的参数则相当于一个占位符, 用户通过键盘输入替换掉参数内容。后面的例子中不会演示自定义参数, 有兴趣的读者可以看看 **TemplateVariable**, **GlobalTemplateVariables** 和 **TemplateVariableResolver** 这些类。

实现模版功能

下面我来实现模版功能, 然后能让模版在内容提示中出现。

定义模版上下文和缺省模版

首先通过扩展方式添加一个模版上下文类型，并且添加一个缺省的模版：

清单 1. 通过扩展点定义模版上下文

```
<extension
  point="org.eclipse.ui.editors.templates">
  <contextType
    class="jtf.tutorial.template.ExprTemplateContextType"
    id="jtf.tutorial.template.contextType"
    name="Expr Template">
  </contextType>
  <template
    autoinsert="true"
    contextTypeId="jtf.tutorial.template.contextType"
    description="Declare a variable"
    id="jtf.tutorial.template.variableDeclaration"
    name="variableDeclaration">
    <pattern>
      ${variable} = ${integer}
    </pattern>
  </template>
</extension>
```

通过扩展方式还是很简单的，缺省的模版叫做“**variableDeclaration**”，它的信息都放在扩展定义里了，注意正确的方式应该是把一些字符串放到资源文件中，为了简单我没有这样做。

模版上下文类型需要一个实现类，在里面你可以管理你的模版参数，下面是 **ExprTemplateContextType** 的代码：

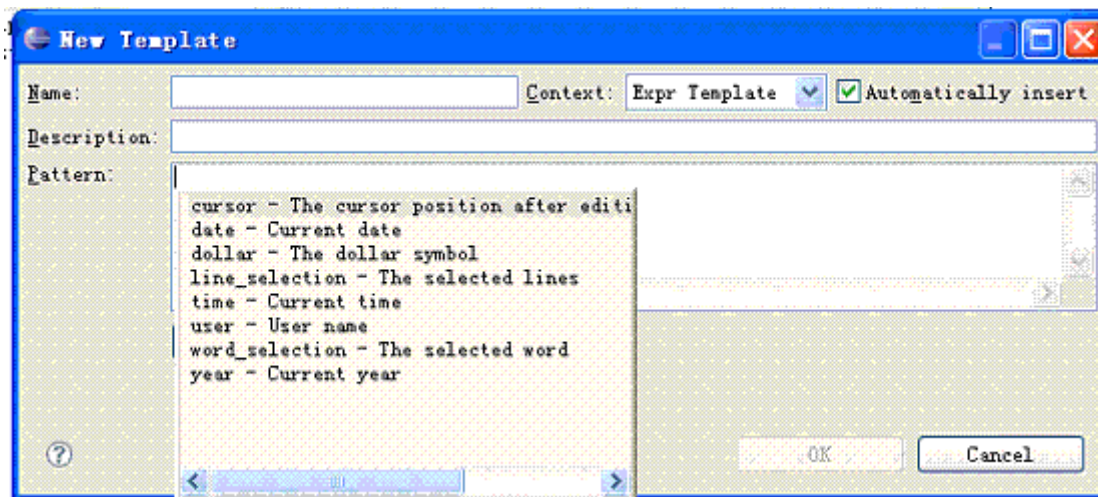
清单2. ExprTemplateContextType 实现

```
public class ExprTemplateContextType extends TemplateContextType {
    public ExprTemplateContextType() {
        addGlobalResolvers();
    }

    /**
     * We add support for global variables
     */
    private void addGlobalResolvers() {
        addResolver(new GlobalTemplateVariables.Cursor());
        addResolver(new GlobalTemplateVariables.WordSelection());
        addResolver(new GlobalTemplateVariables.LineSelection());
        addResolver(new GlobalTemplateVariables.Dollar());
        addResolver(new GlobalTemplateVariables.Date());
        addResolver(new GlobalTemplateVariables.Year());
        addResolver(new GlobalTemplateVariables.Time());
        addResolver(new GlobalTemplateVariables.User());
    }
}
```

我添加了很多全局参数，所以我可以在编辑模版时看到它们，如下图所示：

图1. 全局参数支持



TemplateStore

我们需要一些方法，可以得到 **TemplateStore**，一般来讲，可以把这些方法放到插件的入口类中，因为一个插件只需要一个 **TemplateStore**。所以在本例中，我放到了 **jtf.tutorial.Activator** 里面。你可以看到两个新增的方法：**getTemplateStore**和**getContextTypeRegistry**。我使用了 **ContributionTemplateStore**，因为我需要装载用户自定义的模版。**ContextTypeRegistry** 的作用很好理解，因为同时可能存在多种模版类型，**Eclipse**需要把每类模版的相关信息管理起来，因此把模版上下文注册到了 **ContextTypeRegistry** 中。

添加属性页

因为我要支持用户自定义模版，所以需要添加一个模版属性页。只要继承 **TemplatePreferencePage** 并添加 **org.eclipse.ui.preferencePages** 扩展即可。在此不赘述了。

添加到内容提示

到上一步为止，设置对话框中也出现了我的属性页，用户可以自定义模版了。但是模版没有任何用武之地，还需要将模版添加到内容提示中才有意义。内容提示是由一个个 **Proposal** 组成的，对于模版，也有对应的 **Proposal** 实现：**TemplateProposal**。所以，接下来修改 **ExprContentAssistProcessor**，插入这么一段代码：

清单3. 修改 **ExprContentAssistProcessor** 以支持模版

```
// get template context type
TemplateContextType contextType = Activator.getDefault().
    getContextTypeRegistry().getContextType(Activator.EXPR_CONTEXT_TYPE);

// create template context
TemplateContext context = new DocumentTemplateContext(contextType, doc, offset, 0);

// get all template
List<ICompletionProposal> proposals = new ArrayList<ICompletionProposal>();
Template[] templates = Activator.getDefault().
    getTemplateStore().getTemplates(Activator.EXPR_CONTEXT_TYPE);
for(Template t : templates) {
    proposals.add(new TemplateProposal(t, context, new Region(offset, 0), null));
}
```

这是一个非常简化的版本，正常情况下，你需要判断光标之前有没有其它字符，如果有，只应该显示以光标之前字符串开头的模版，当你定义了多个模版上下文类型的时候，逻辑就要更复杂了。即便我已经省略了很多东西，创建 **TemplateProposal** 似乎还是比普通的 **Proposal** 麻烦一些，需要得到 **TemplateContext**，而 **TemplateContext** 需要 **TemplateContextType**。

效果

我们的缺省模版现在可以出现在内容提示里了，选择之后，模版的内容就被插入到了编辑器中：

图 2. 内容提示中的模版

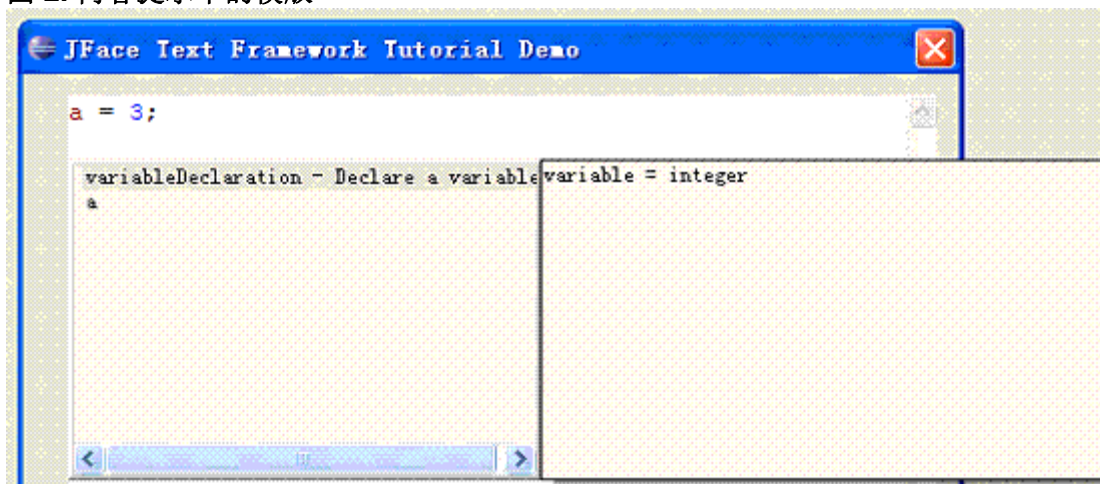
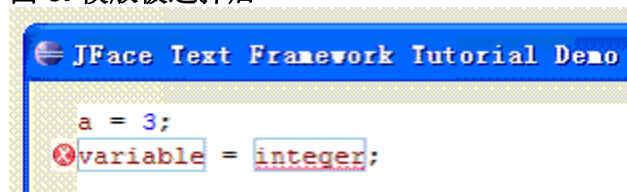


图 3. 模版被选择后

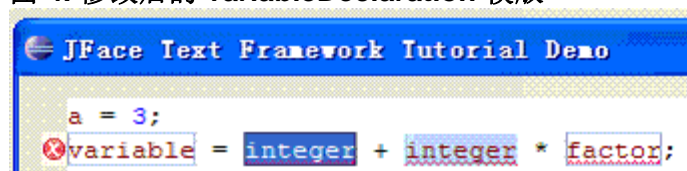


Linked Model

有趣的事情还没有完，当模版被插入之后，可以看到每个参数的周围都有一个小框，你可以用 **Tab** 键在各个参数之间切换，当你按下回车键的时候，小框就消失了，也无法用 **Tab** 键导航了。

我在以前的文章中说过，这些框其实是标注，所以并不神奇。神奇的是可以用 **Tab** 键在参数之间导航，而且还有更神奇的，请看下图：

图 4. 修改后的 variableDeclaration 模版



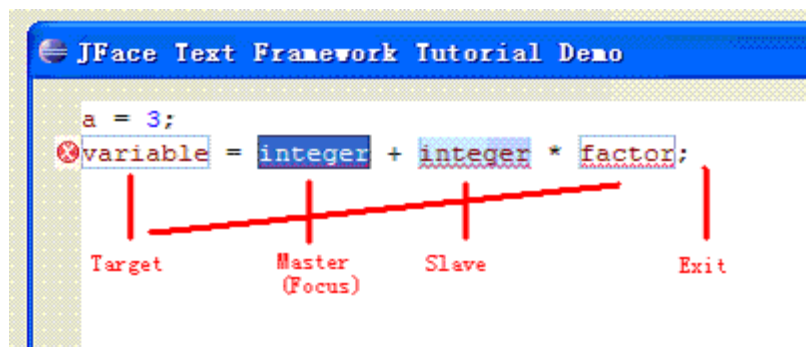
我把 `variableDeclaration` 模版修改了一下，变成了“`${variable} = ${integer} + ${integer} * ${factor};`”。注意第二个参数和第三个参数名字是相同的。当你编辑第二个参数的时候，第三个参数周围不是一个空心矩形框，而是一个实心的蓝色背景，并且当你修改了第二个参数之后，第三个参数也跟着变了。

这些看上去很有趣的功能叫做 **Linked Model**（链式模型），其实就是把一些相关联的标注管理了起来，之所以叫链式模型，我个人认为可能有两个原因：

1. 可以用 **Tab** 键在它们之间按先后顺序切换，好像一个链表一样
2. 第二个参数改变会导致第三个参数也改变，也就是说第三个标注好像链接到了第二个标注上一样

为了实现这样一些功能，**JTF** 到底创建了多少种标注呢？看上去只有两种或者三种，实际上有四种，如下图所示：

图5. 标注类型



普通的空心矩形框表示的是 **Target**（目标）类型的标注，当一个 **Target** 类型的标注拥有焦点时，就成为了 **Master**（主）类型标注，也叫做**Focus**（焦点）类型标注。对于第三个参数，它和第二个参数名称相同，所以 **JTF** 为它创建了一个 **Slave**（从）类型标注。注意最后一个标注，我不是画线画的不准，而是在那个肉眼看不到的地方，还存在一个 **Exit**（退出）类型标注。这些词不是我发明的，如果你看看 **LinkedPositionAnnotations** 这个类就知道了。

将这些标注联系起来的管理方式，就叫做链式模型，由于它即牵涉到标注的管理，又牵涉到界面的绘制，所以在实现上采用了 **MVC** 的模式。**LinkedModeModel** 是模型部分，**LinkedModeUI** 是界面和控制部分。这两个类不是全部，还有很多其它的相关类，甚至是内部类，最有必要了解的是 **LinkedModeUI.IExitPolicy** 这个内部接口。一般来说，当你按下回车的时候，你会从模版编辑中退出来，这就是由 **IExitPolicy** 来判断的。所以，你可以定制退出的行为。**Eclipse** 里面有没有定制退出行为的例子呢？有的，在 **Java** 编辑器里，输入引号之后，编辑器会自动帮你插入另一个引号，如果你再输入一个引号，你不会看到三个引号，而是光标移到了自动插入的引号之后，假如你不继续输入引号，还是按回车，光标也会移到自动插入的引号之后。这就是自定义退出行为的例子，**Java** 编辑器会检查你输入的是什么引号，当你再按下它的时候，你就退出链式编辑状态了，**Java** 编辑器的这个功能同样适用于各种括号。这个功能，可以称为 **Auto Completion**（自动补全），我觉得还不能说它是**JTF**的标准特性之一，它只是链式模型的一个有趣的应用。

所以，这种用 **Tab** 键在很多个标注之间导航的功能，不是模版的专利，你可以用在任何你想用的地方。比如：插入一个函数时，可以用链式模型管理函数的参数，就好像 **Java** 编辑器那样。至于如何做，**TemplateProposal** 已经给了你一个不错的例子。

提示：如果你想要看看证据的话，浏览一下 **CompilationUnitEditor** 中的内部类 **BracketInserter** 和 **ExitPolicy**，这个类在 **org.eclipse.jdt.ui** 中。

结束语

在实现模版的过程中，我也顺便提及了一些相关概念，因为模版和它的特性有着微妙的联系。下面是一些值得思考的问题：

1. 试试看自定义模版参数
2. 自定义 **TemplateStore**？没有什么不可以。
3. 尝试使用多个模版上下文类型
4. 想想看在什么场合可以使用链式模型这个有趣的功能呢

声明

本文仅代表作者的个人观点，不代表IBM的立场。

下载

描述	名字	大小	下载方法
----	----	----	------

第九小节示例代码

jtf.tutorial.part9.zip

1140KB

HTTP

[→ 关于下载方法的信息](#)

参考资料

- 如果对ANTLR有更多的兴趣，请参考“[ANTLR 项目主页](#)”
- “[SWT 和 JFace，第 1 部分: 简介](#)”（developerWorks 中国，2005 年 5 月）介绍了JFace基础知识。

关于作者

马若劫是 Lotus Forms 部门的一位软件工程师，主要从事电子表单技术的研发工作。他在 Eclipse 和 Java 方面有多年研发经验，同时也是国内著名开源项目 LumaQQ 的创立者

IBM 公司保留在 developerWorks 网站上发表的内容的著作权。未经IBM公司或原始作者的书面明确许可，请勿转载。如果您希望转载，请通过 [提交转载请求表单](#) 联系我们的编辑团队。