


Alexey Efimov

IntelliJ IDEA插件开发基础

Note:

这篇文章将帮助你了解IntelliJ IDEA插件开发的基础知识，快速开发自己的插件。它讲解了日常插件开发的基本原则、语法描述和插件发布，同时包含了一个插件例子，一步步教你如何去开发插件。当你读完这篇文章后，你可以试用一下文章中开发的插件： [SampleProject.zip](#) (242 Kb)
你需要依据 [如何开始](#) 段落中的步骤1和2建立 *IntelliJ IDEA JDK*。

内容

[约束条件](#)

[介绍](#)

[获取插件](#)

[如何开始](#)

[步骤1](#)

[步骤2](#)

[步骤3](#)

[步骤4](#)

[插件是如何工作的？](#)

[IntelliJ IDEA的组件模型](#)

[加载组件](#)

[卸载组件](#)

[组件容器](#)

[HelloWorld 插件](#)

[创建组件](#)

[创建 Action](#)

[第一个插件](#)

[添加配置](#)

[插件的结构描述](#)

[插件标识](#)

[插件间的依赖关系](#)

[版本和编译版本号](#)

[组件注册](#)

[本地化](#)

[插件发布](#)

[创建归档](#)

[上传插件至插件库](#)

[通报新插件](#)

[总结和常用链接](#)

约束条件

这篇文章假设你正在使用IntelliJ IDEA 5.1或更高版本，或者是高于[EAP#4121](#)的版本。 文章中涉及到的一些IntelliJ IDEA Open API 中特性或类可能在以前的版本中并不存在。

介绍

在IntelliJ IDEA中, *插件*是独立的模块, 它可以通过手动和内置工具的方式加载到IntelliJ IDEA中。

IntelliJ IDEA的插件依据他们的功能可以进行分组, 主要包括:

代码审查和重构 (Inspection Gadgets, Intension Power Pack, Refactor-J, Refactor-X等等)

自定义编辑器 (例如 Images)和工具窗口 (SQLQuery, PsiViewer等)

其他语言支持 (Groovy, JavaScript等)

应用服务器整合(Resin, JBoss, Tomcat等)

版本控制系统整合 (CVS, Clearcase等)

框架和其他技术支持(Hibernate Tools, IdeaSpring, XPathView, Struts等)

外部应用整合 (Jira Browser, JFormdesigner等)

用户界面提升, 各种工具栏和菜单 (TabSwitch, CVS bar等)

自定义编译器 (Native2Ascii)

其他: 包括各种工具插件和游戏(IdeaJad, simpleUML, Tetris, Sokoban等)

目前IntelliJ IDEA为插件开发人员提供了各种特性和功能, 因此IDEA很多的特性都可以被扩展。在IntelliJ IDEA 5.0之前版本, 没有提供对其他语言的支持, 现在IDEA可以以插件的方式支持各种语言, 如JavaScript,CSS, Groovy, SQL等等. 渐渐地, 通过版本 更新, IntelliJ IDEA将为插件开发人员提供特性, 这些将通过 **IntelliJ IDEA Open API** 呈现出来。

下面就是一些比较受欢迎的IntelliJ IDEA插件:

TabSwitch: 小巧, 但非常实用。它允许你在打开的各个文件中进行切换, 就象使用Alt+Table快捷键切换Windows应用一样。

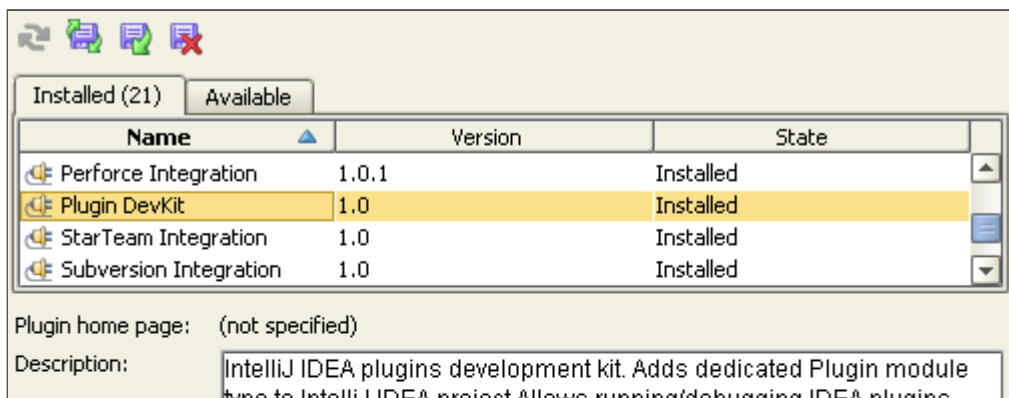
SQL Query:在IDE工具就可以编写并执行数据库查询语句, 有时该工具就象呼吸一样非常必要。

Regex: 运行你在IDEA中校验各种正则表达式。

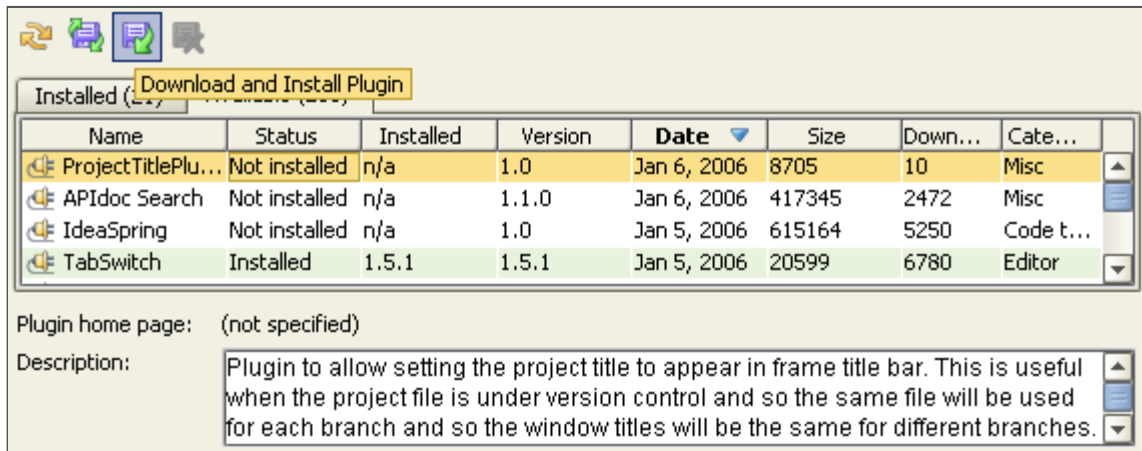
XPathView:在编辑窗口中提供了XML文件的XPath处理能力。

获取插件

已经在插件库中注册过的插件可以点击**File->Settings->Plugins** 菜单, 插件管理器将会添加、更新和删除相关插件。如果你做了相关的操作, 需要你重新启动IDEA使其生效。



首先下载插件列表，只需点击**Available**标签。如果你是通过代理服务器联到Internet，请点击 **HTTP Proxy Settings**按钮，输入正确的信息。 IntelliJ IDEA会向插件库(<http://plugins.intellij.net>)发出请求，列出适用当前版本IDEA的所有插件。你只需在列表中选择该插件，然后点击下载图标，就可以安装该插件。以这种方式，你可以安装更多的插件。



如何开始插件开发?

假想你现在决定开发一个插件来满足你同事的强烈需求，行，让我们开始吧。

首先需要创建一个项目来包含你的插件。

步骤 1

如果你还没有下载**Plugin Development Package**，请先下载此开发包。

如果你使用EAP版本进行插件，你可以下载EAP的插件开发包，文件如**idea4121-dev.zip**，该文件位于[IntelliJ IDEA EAP Access](#)页面(4121是EAP的编译版本号)。

如果你想下载发布版本的插件开发包，请访问[IntelliJ IDEA Plugin Developers](#)页面，点击**Plugin Development**链接去下载开发包文件，

解压下载的开发包文件至IntelliJ IDEA的安装目录。

注意:

开发包仅包括插件源码和IntelliJ IDEA Open API 的Javadoc文档，所以及时没有该开发包你同样进行插件开发。然后我们推荐你下载并安装该开发包，它会让你的开发更加便捷。

步骤 2

解压插件开发包以后，我们需要创建一个项目来编写我们的插件。

点击**File->New Project**菜单项，输入项目名称，如"idea-plugins"，然后设定项目的存储路径，点击**Next**。

插件开发需要特定类型的Java™ SDK (JDK): IntelliJ IDEA SDK，它是Jdk标准库和IntelliJ IDEA Open API库的整合体，你可以设置该SDK的JavaDoc和Source Code，这些信息可以在插件开发包找到。

点击**Configure**按钮，将会弹处Jdk设置窗口，点击**Add IntelliJ IDEA SDK**按钮（包含IDEA 图标的那个按钮），该按钮如下所示：



选择IntelliJ IDEA的安装目录（实际上IDEA会自动选择），然后点击OK按钮。

接着, 将该SDK的名称改为 "idea", **Sandbox Home**是保存运行IDEA各种配置参数和缓存的目录，IDEA会将插件拷贝到该目录下进行调试，如果可能的话，请更改目录的**sandbox**存储目录。

Name:	idea
IntelliJ IDEA SDK home path:	C:\TOOLS\idea
Sandbox Home:	C:\TOOLS\idea\sandbox

注意:

这里我们推荐你不要使用IntelliJ IDEA的SDK版本号，如果你使用EAP版本的话，这可能引起混乱，如"IDEA 4121"名称可能在更高版本的IDEA的插件开发中引起误解。







点击OK按钮， 你会看到IntelliJ IDEA SDK会被显示在项目可用JDK的列表中。设置JDK你只需做一次，今后你只需从列表中选择即可。

点击Next按钮。

步骤 3

在下一个对话框中，选择Create single-module project选项，然后点击Next按钮。

这是我们需要设定module类型，这里需要选择Plugin Module。

Select module type:	Description:
 Java Module	This module type is designed to e IntelliJ IDEA. It allows you to prop SDK and all necessary deploymer running/debugging instance of In
 Web Module	
 Ejb Module	
 J2EE Application Module	
 J2ME Module	
 Plugin Module	

接着我们需要设定module的名称和路径，如果我们是初学者，就让我们创建传统的HelloWorld插件，这里module名称最好能表达插件所代表的含义。

Module name:
helloWorld
Module content root:
C:\IdeaProjects\idea_plugins\helloWorld

点击Next 按下；在下一个对话框中点击Finish即可。

步骤 4

恭喜你!你已经完成了插件项目的创建，。

在Project工具窗口中展开项目菜单树，找到`plugin.xml`文件（在 `META-INF` 目录下），这是插件的描述文件，包含插件的各种信息，如插件名称、描述、适用的IDEA版本号等等，同时包含component和action描述信息，该描述文件是IntelliJ IDEA加载插件的依据。如果你创建了一个component，而沒有在该描述文件中声明的话，IDEA并不会加载该component，请注意这点。

`plugin.xml`文件包含各种信息，请不要忘记设定。

解下来让我们设定插件的描述文件，让其工作起来。这里我们设定插件名称，描述和其他信息。让我们看一下`since-build`属性，在后面的我们会进行具体描述，现在我们只需设置为Help->About对话框中的编译版本号即可。

META-INF/plugin.xml文件内容如下：

```
<!DOCTYPE idea-plugin PUBLIC
"Plugin/DTD" "http://plugins.intellij.net/plugin.dtd">
<idea-plugin>
  <name>HelloWorld</name>
  <description>This plugin does nothing</description>
  <version>1.0</version>
  <vendor>JetBrains</vendor>
  <idea-version since-build="4121" />
</idea-plugin>
```

现在我们可以开始编写我们第一个插件啦。

插件是如何工作的？

在进行Hello World插件开发之前，我们想向大家介绍一下插件工作的原理。所有的class和interface都来自于IntelliJ IDEA Open API 开发包，你可以在插件开发包中找到。

IntelliJ IDEA component模型

IntelliJ IDEA 包含多种组件模型，这些组件模型都是基于[PicoContainer](#)，组件都包含在这些容器中，但容器有不同的级别。

有三种级别的容器：*application*，*project*，和 *module*。在application级别上可以包含多个*project*级别的容器；而project级别包含多个*module*级别容器。

Application Component 在IntelliJ IDEA程序中，Application Component仅以一个实例存在。创建application component，只需实现[ApplicationComponent](#) 接口，然后在`plugin.xml`文件的`application-components`区域进行声明。

Project components. 在每一个打开项目中，仅包含一个project component实例，因此在IntelliJ IDEA中可能包含多个Project component实例。创建一个project component，只需实现[ProjectComponent](#) 接口，然后在`plugin.xml`文件的`project-components` 区域进行声明即可。

Module components. 在没一个项目的module中，仅有一个module component实例，因此在一个项目中包含多个module-level component实例，实例的数量就是项目中加载的module的数量。创建module component，只需实现[ModuleComponent](#) 接口，然后在`plugin.xml` 文件的`module-components` 区域进行声明。

project component的实例数并不等于IntelliJ IDEA打开的项目数量，而是大于打开的项目数。除了打开的项目包含项目组件，另一个项目是Template Project，它是一个隐藏的项目，IDEA一直加载这个项目。

Component加载

Application Component在IDEA启动时加载。Project和module component在项目启动时共同加载。

一个组件通过调用以下函数来实例化：

1. 首先加载Component的构造函数。
2. 如果组件实现[JDOMExternalizable](#)接口，将会调用 [readExternal](#) 函数。
3. 接下来[initComponent](#) 函数将被调用。
4. 如果是project或module component，[projectOpened](#) 函数将会被调用。如果是module component，[moduleAdded](#) 函数会被调用。

如果希望在component中访问其他component，我们只需在该component的构造函数声明即可，在这种情况下，IntelliJ IDEA自动实例化所依赖的component，因此在构造函数中，所依赖的组件依据被实例化啦。

有依赖关系的component实例化样例

```
package com.intellij.tutorial.helloWorld;
import ...
public class MyComponent implements ApplicationComponent {
    private final MyOtherComponent otherComponent;
    public MyComponent(MyOtherComponent otherComponent) {
        this.otherComponent = otherComponent;
    }
    ...
}
```

Component卸载

Application component在IDEA关闭时被卸载， Project和module component在项目关闭时被卸载。

在卸载component是，以下函数将会被调用：

1. 如果component实现了 [JDOMExternalizable](#) 接口，component的状态将会通过调用[writeExternal](#) 函数保存到xml文件中。
2. 如果是project 或 module component，[projectClosed](#) 将会被调用。
3. 接下来[disposeComponent](#)将会被调用。

Component 容器

前面我们提到有三种不同的容器，application container实现[Application](#) 接口； project container 实现[Project](#)接口； module 容器实现[Module](#)接口。 每一个容器都有自己的函数去获取容器内的component。

从application容器中获取组件例子：

```
Application application = ApplicationManager.getApplication();
MyOtherComponent otherComponent =
    application.getComponent(MyOtherComponent.class);
```

获取application容器是非常容易的，在IDEA中application容器仅有一个(上面的例子就说明啦)。获取project和module容器可能有点困难。通常我们在component构造函数声明或从action的事件处理的context的获取。

将容器以构造参数方式引入到component中

```

import ...
public class MyProjectComponent implements ProjectComponent {
    private final Project project;
    public MyProjectComponent(Project project) {
        this.project = project;
    }
    ...
    public void foo() {
        MyOtherProjectComponent otherProjectComponent =
            project.getComponent(MyOtherProjectComponent.class);
    }
}

```

在这个例子中，组件在构造函数中获取了容器对象，将其保存，然后在component其他方面进行引用。但是还是要注意这一点。Be careful when passing this reference to other components (especially application-level ones). If an application-level component does not release the reference, but saves it inside itself, all the resources used by a project or module will not be unloaded from the memory on the project closing.

一个稍微复杂的例子就是在事件处理中获取容器，请看下面的例子。[DataContext](#)类可以为action提供获取容器的方法，这个class应该引起足够的注意。

在Action的事件处理中获取容器对象：

```

import ...
public class MyAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        DataContext dataContext = e.getDataContext();
        Project project =
            (Project)dataContext.getData(DataConstants.PROJECT);
        Module module =
            (Module)dataContext.getData(DataConstants.MODULE);
    }
}

```

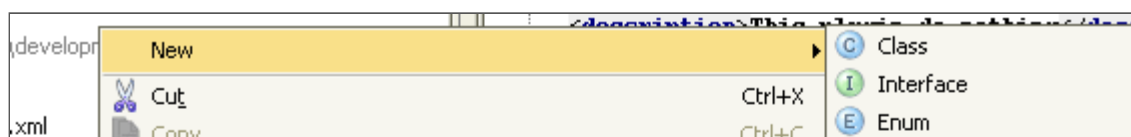
HelloWorld 插件

现在让我们回到我们的HelloWorld插件，我们将增加一个新的component，它将弹处一个对话框显示"Hello World!"文本。

创建 Component

在项目中创建`com.intellij.tutorial.helloWorld` package。在该package点击鼠标右键，在弹出的菜单中点击New->Application Component菜单项，

输入component名称: `HelloWorldApplicationComponent`。



点击 **OK**按钮， IntelliJ IDEA 将会创建该组件同时会在`plugin.xml`文件中自动注册。

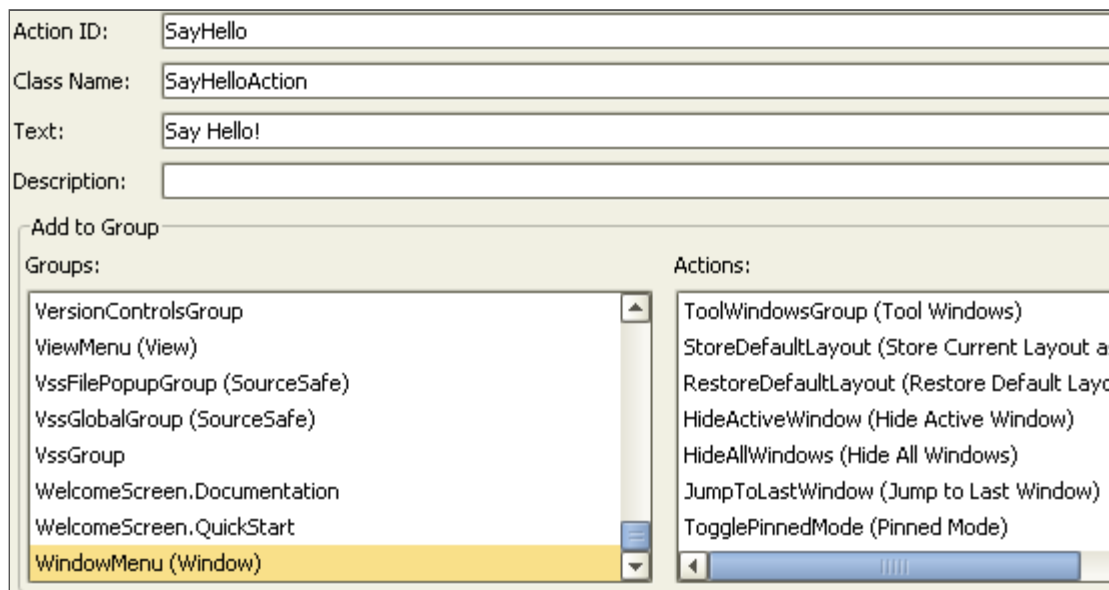
接下来让我们在该组件中添加`sayHello` 函数。我们将使用[Messages](#) 工具类去显示特定信息。

文件: `/com/intellij/tutorial/helloWorld/HelloWorldApplicationComponent.java`

```
package com.intellij.tutorial.helloWorld;
import ...
public class HelloWorldApplicationComponent implements
    ApplicationComponent {
    public void initComponents() {
    }
    public void disposeComponent() {
    }
    public String getComponentName() {
        return "HelloWorldApplicationComponent";
    }
    public void sayHello() {
        // Show dialog with message
        Messages.showMessageDialog(
            "Hello World!",
            "Sample",
            Messages.getInformationIcon()
        );
    }
}
```

创建 Action

我们已经创建了"Hello World" component，接下来我们需要添加一个菜单项来体验其功能。右击 `helloWorld` package，在弹出的菜单中选择**New->Action**菜单项，输入Action的ID和类名，在**Groups** 列表中选择**WindowMenu (Window)**(请参考下面的截屏)。



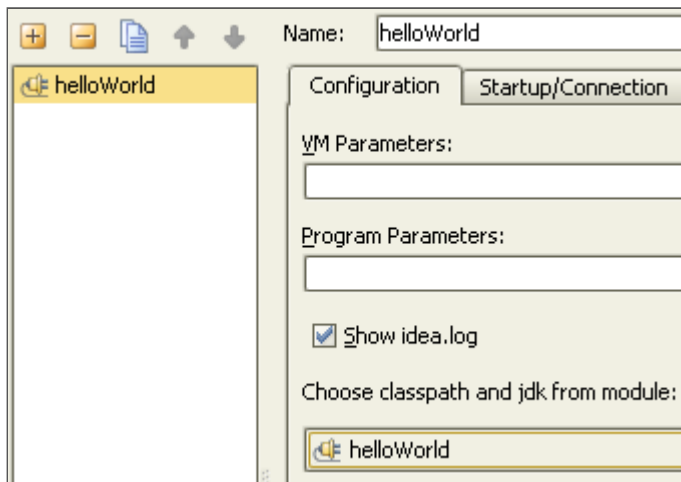
点击OK按钮，IntelliJ IDEA将会创建Action类，同时在`plugin.xml`中进行声明。现在我们需要在Action调用component的`sayHello`。

文件: `/com/intellij/tutorial/helloWorld/SayHelloAction.java`

```
package com.intellij.tutorial.helloWorld;
import ...
public class SayHelloAction extends AnAction {
    public void actionPerformed(AnActionEvent e) {
        Application application =
            ApplicationManager.getApplication();
        HelloWorldApplicationComponent helloWorldComponent =
            application.getComponent(
                HelloWorldApplicationComponent.class);
        helloWorldComponent.sayHello();
    }
}
```

启动第一个插件

接下来我们需要设定配置去运行或调试我们的第一个插件，点击`Run->Edit Configurations`菜单，在`Run/Debug Configuration`对话框中，点击`Plugin` 标签，创建一个新的配置项，然后点击OK按钮。

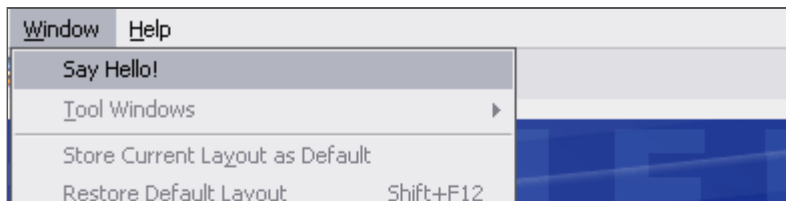


现在我们只需按下`Shift+F9` (或点击 `Run->Debug` 菜单项) 就可以启动该插件。

注意:

为了调试或测试插件，我们需要在IntelliJ IDEA中启动另一个IDEA实例，在这个实例中，插件已经被加载了，我们可以进行各种操作。IntelliJ IDEA 使用 `sandbox` 文件夹保存该实例的各种配置信息，并将相关的插件拷贝到该目录下进行调试和测试。

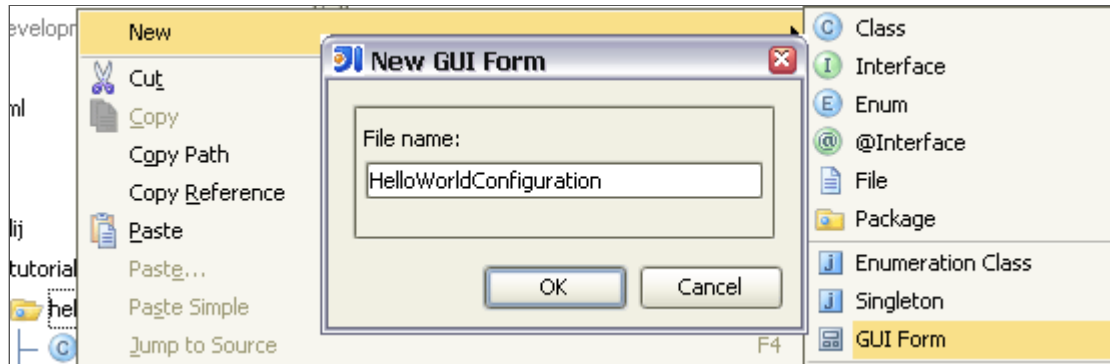
IntelliJ IDEA调试实例启动后，切换到新的窗口，点击`Window`菜单，选择`Say Hello!` 菜单项。



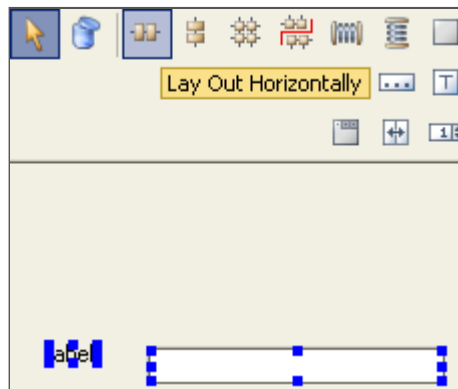
添加插件配置项

到目前为止，我们的插件仅仅显示"Hello World!"， 显的优点幼稚。下面我们希望我们的插件更专业点。 为了做到这点，我们需要为插件创建一个Form，来保存插件的配置信息，这个配置Form包含一个文本输入框。

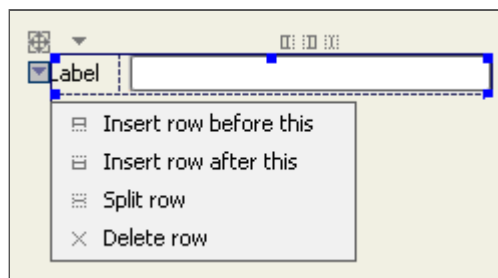
在**helloWorld** package上右击鼠标，在弹出的菜单中选择New->GUIForm项。输入Form名称：**HelloWorldConfiguration**。



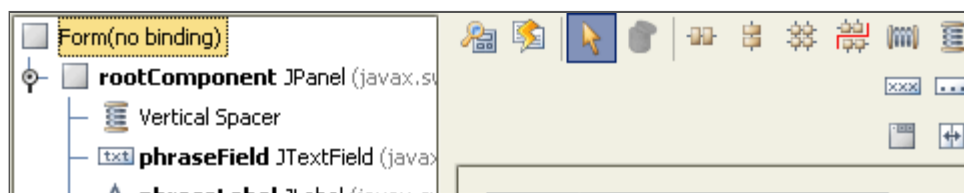
在Form编辑窗口中，拖动 **JLabel** 和 **.JTextField** 元素，放在Form窗口中。将元素按以下屏幕布局，然后选择这些元素，点击**Lay Out Horizontally** 按钮。



IntelliJ IDEA将会创建一个水平面板，包含选定的各个元素。在编辑窗口中选择面板，在面板的左边会出现一个箭头图标，点击该箭头图标，选择 **Insert row after this** 将会创建一个新的格子。



接下来在元素面板中选择 **Vertical Spacer** 元素，拖放至新创建的格子中。



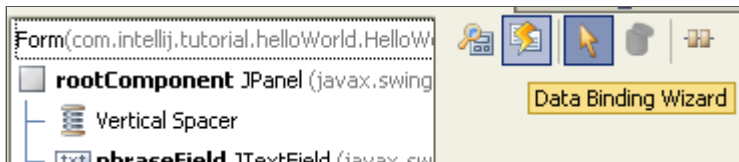
在元素的树行菜单中（在form 编辑窗口的左边), 选择 **JPanel** 元素, 在属性面板中, 设置JPanel的 **binding**属性为 **rootComponent**名称, 对于输入框(JTextField)将其 **binding** 属性设为phraseField名称。对于标签(JLabel) 将其设置为 **phraseLabel**。现在Form已经完成啦, 我们需要将Form绑定到指定的Java类。

在同样的package下创建一个新的类, 命名为**HelloWorldConfigurationForm**。

切换回Form编辑窗口, 将Form的**binding** 属性值设定为该类的名称。

这一切做完后, IntelliJ IDEA 将会在元素树中高亮显示这些元素, 提示你在**HelloWorldConfigurationForm**类去创建对应的字段。 你只需按下**Alt + Enter**快捷键就可以解决这些问题。

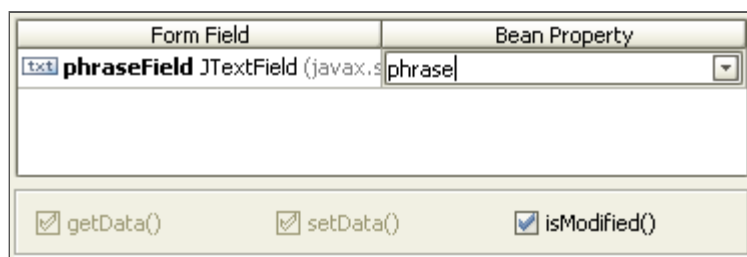
当所有的冲突被解决后, 点击 **Data Binding Wizard** 按钮 (如以下屏幕所示)。



向导将会创建相关的代码来读取和设置Form的各种元素的信息。 选择**Bind to existing bean**选项, 将其设置为 **HelloWorldApplicationComponent** 类, 然后点击**Next**按钮。



在Bean Property 列表项中, 设置类的字段名称: **phrase**, 然后点击**Finish**按钮。IntelliJ IDEA会自动创建相关的代码去读取 **HelloWorldConfigurationForm** 类的字段值(i.e. the **phrase** 字段), 同时会添加**isModified** 函数。



现在我们需要更改**HelloWorldConfigurationForm** 类, 添加一个方法可以获取form的根元素**getRootComponent**。

文件: `/com/intellij/tutorial/helloWorld/HelloWorldConfigurationForm.java`

```
public class HelloWorldConfigurationForm {
    private JPanel rootComponent;
    private JTextField phraseField;
    private JLabel phraseLabel;
    public HelloWorldConfigurationForm() {
        // Enable mnemonic action
        phraseLabel.setLabelFor(phraseField);
    }
}
```

```

// Method returns the root component of the form
public JComponent getRootComponent() {
    return rootComponent;
}
public void setData>HelloWorldApplicationComponent data) {
    phraseField.setText(data.getPhrase());
}
public void getData>HelloWorldApplicationComponent data) {
    data.setPhrase(phraseField.getText());
}
public boolean isModified>HelloWorldApplicationComponent data) {
    return phraseField.getText() != null ?
        !phraseField.getText().equals(data.getPhrase()) :
        data.getPhrase() != null;
}
}

```

现在,我们有了一个图形化的Form, 通过该Form的`phrase` 字段可以设置`HelloWorldApplicationComponent` 类的字段信息。现在我们需要更改`sayHello` 方法, 依据`phrase`字段来显示特定的文本值。

修改HelloWorldApplicationComponent类的sayHello方法

```

public void sayHello() {
    // Show dialog with message
    Messages.showMessageDialog(
        phrase,
        "Sample",
        Messages.getInformationIcon()
    );
}

```

为了将该组件注册到IntelliJ IDEA设置面板中 (通过File->Settings菜单项), 我们需要对插件的component进行调整。现在我们需要让component实现`Configurable` 接口。在下面的代码, 你可以看到该接口的实现。

Configurable 接口在HelloWorldApplicationComponent类中的实现

```

import ...
public class HelloWorldApplicationComponent
implements ApplicationComponent, Configurable {
    private HelloWorldConfigurationForm form;
    private String phrase;
    public void initComponents() {
    }
    public void disposeComponent() {
    }
    public String getComponentName() {
        return "HelloWorldApplicationComponent";
    }
    public void sayHello() {
    }
}

```

```

        // Show dialog with message
        Messages.showMessageDialog(
            phrase,
            "Sample",
            Messages.getInformationIcon()
        );
    }
    public String getDisplayName() {
        // Return name of configuration icon in Settings dialog
        return "HelloWorld";
    }
    public Icon getIcon() {
        return null;
    }
    public String getHelpTopic() {
        return null;
    }
    public JComponent
    createComponent() {
        if (form == null) {
            form = new HelloWorldConfigurationForm();
        }
        return form.getRootComponent();
    }
    public boolean isModified() {
        return form != null && form.isModified(this);
    }
    public void apply() throws ConfigurationException {
        if (form != null) {
            // Get data from form to component
            form.getData(this);
        }
    }
    public void reset() {
        if (form != null) {
            // Reset form data from component
            form.setData(this);
        }
    }
    public void disposeUIResources() {
        form = null;
    }
    public String getPhrase() {
        return phrase;
    }
    public void setPhrase(final String phrase) {
        this.phrase = phrase;
    }

```

```
}  
}
```

为了实现 [Configurable](#) 接口，我们需要了解一下component的Form的工作原理，例如，在[createComponent](#) 方法中我们需要创建一个Form，返回其根元素。当Form被显示时， [apply](#), [reset](#) 和 [isModified](#) 方法将被激活。

当用户点击 OK按钮或Apply按钮， [apply](#) 函数将会被调用，form中的文本框将自动会更新

[HelloWorldApplicationComponent](#) 类的字段。 当form被实例化后，用户点击Cancel按钮， [reset](#) 函数将会被调用，form的文本框中的值将会被重置。Form会定期调用[isModified](#) 函数更新component状态。 如果函数返回 **false**, Apply 按钮将不会生效。

当用户关闭Form时， [disposeUIResources](#) 函数将会被调用。在该方法中，你需要释放Form中涉及的各种资源(在这个例子我们需要将form置为 **null**)。

注意：

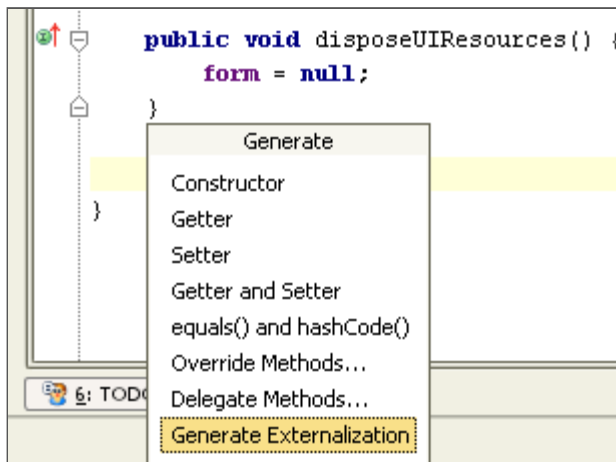
在这个例子中，可能你已经注意到在[createComponent](#) 函数中创建Form，在[disposeUIResources](#) 函数中释放。 在下一次操作中， [createComponent](#) 函数会再次创建Form，这是因为IntelliJ IDEA并未在每次调用createComponent后设置当前Look&Feel，所以保证Form和其他form的Look&Feel一样， 在每次调用 [createComponent](#) 函数时，你需要重新创建Form。

[Configurable](#)接口同时添加了三个函数： [getDisplayName](#) 函数显示该component在设置面板中的名称， [getIcon](#) 函数提供图标， [getHelpTopic](#)返回对应帮助文档中的ID。如果返回值不为空，Help按钮就会呈现。

现在还需要对component进行一次更改，我们希望将component的配置信息保存起来，如保存

[HelloWorldApplicationComponent](#)类的[phrase](#)字段值。为了能够保存和恢复当前的component状态，我们的component还需实现[JDOMExternalizable](#) 接口。

在 IntelliJ IDEA中，我们可以快速产生各种函数保存component的配置信息，打开 [HelloWorldApplicationComponent](#) 类，按下 **Alt+Insert**快捷键，在弹出的菜单中选择Generate Externalization选项。



注意：

整个操作仅当component没有实现[JDOMExternalizable](#) 接口时有效。

执行操作后，IntelliJ IDEA 将会实现[JDOMExternalizable](#) 接口，并添加基本的实现代码，如[readExternal](#)函数和[writeExternal](#)函数。

"Generate Externalization"操作产生的结果：

```
package com.intellij.tutorial.helloWorld;  
import ...  
public class HelloWorldApplicationComponent
```

```

    implements ApplicationComponent, Configurable,
    JDOMExternalizable {
        ...
        public void readExternal(Element element)
        throws InvalidDataException {
            DefaultJDOMExternalizer.readExternal(this, element);
        }
        public void writeExternal(Element element)
        throws WriteExternalException {
            DefaultJDOMExternalizer.writeExternal(this, element);
        }
    }
}

```

从以上的例子我们可以看出，[DefaultJDOMExternalizer](#) 工具类可以保存和恢复component的状态。这个类可以保存指定对象的所有public字段值。现在，phrase字段是private，这个字段会被忽略，所以为了保存phrase值，我们需要将phrase字段的private改为public，同时需要给phrase设定默认值。

将HelloWorldApplicationComponent类的phrase字段设为public

```

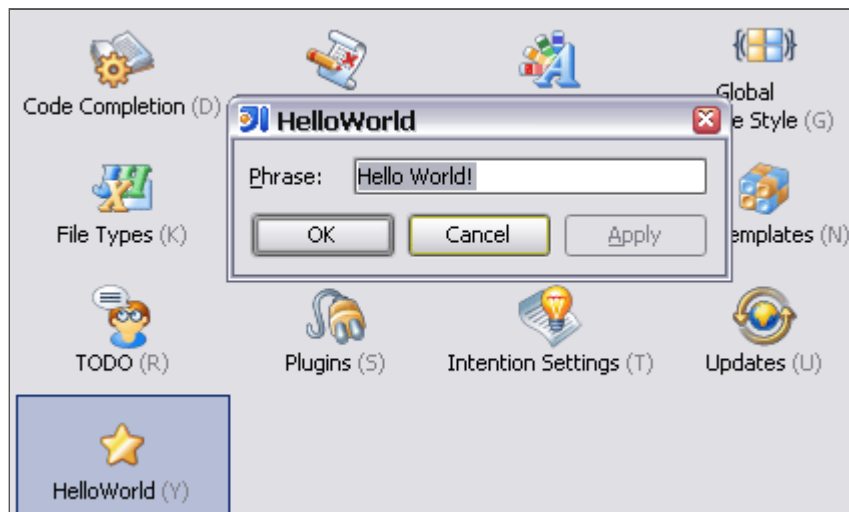
package com.intellij.tutorial.helloWorld;
import ...
public class HelloWorldApplicationComponent
implements ApplicationComponent, Configurable,
JDOMExternalizable {
    ...
    public String phrase = "Hello World!";
    ...
}

```

注意:

如果不可能将字段设为public，而又要保存这些值，你可以只需使用[JDOM](#) 模型保存和恢复这些字段值 (或使用[JDOMExternalizerUtil](#) 类)，而不必使用[DefaultJDOMExternalizer](#) 类。[JDOMExternalizerUtil](#) 类可以让你使用[writeField](#)函数和 [readField](#)函数写入和读取[JDOM](#)模型的属性值。

现在已经可以启动插件啦，请按下Shift + F9，切换到新的IntelliJ IDEA 实例窗口，点击 **File->Settings** 菜单项。



注意:

目前我们还没有设置component的图标(`getIcon` 函数返回值`null`), IntelliJ IDEA 会自动显示默认的yellow star图标。

在Form输入新的字符串, 而不是"Hello World!", 例如, "Caramba! Corrida!", 保存更改。回到主界面, 点击 **Window->Say Hello** 菜单项。

如果你想验证IDEA是否真正保存component信息, 你可以重新启动IDEA的debug实例, 再点击 **Window->Say Hello**菜单项, 将会呈现以下对话框。



插件描述

我们已经创建和启动了插件, 但是我们还没有对`plugin.xml`进行任何编辑, IntelliJ IDEA帮我们完成了这一切。在某些情况下, 我们可能需要手动编辑该文件, 这也是我们要在这部分讲述的。

插件标识

除了设置插件的名称, 你可能还需要设置插件的标识, 只需设置 `id`元素即可。和插件名称不一样, 标识符不能包含空格, 标识符可以用于插件之间的依赖关系。

插件之间的依赖关系

如果你的插件要使用其他插件的API, 可以在 `plugin.xml` 文件中进行声明。默认的情况下, 插件之间的API是不能互相访问的。

插件之间依赖关系可以通过插件描述文件中的`depends`元素进行声明, 你只需设置依赖插件的ID即可。你可以设置多个`depends`元素设定多个依赖关系。

插件之间依赖关系样例

```
<idea-plugin>
  ...
  <depends>MyOtherPluginId</depends>
  <name>MyPlugin</name>
  ...
</idea-plugin>
```

在上面的例子中, `MyPlugin` 插件使用`MyOtherPluginId`插件的API, 所以进行以上设置。

插件版本和编译版本

插件的**版本号**可以告诉IDEA当前插件是否是最新的, 是否需要更新, 这也是为何每一个插件都需要设定版本号, 这样用户可以轻松更新插件。

编译版本号可以告知当前插件适用的IDEA版本，你可以通过 `since-build`元素设定。

`until-build` 属性可以设置插件适用的IDEA最高版本号。

插件版本号 和 编译版本号样例：

```
<idea-plugin>
    ...
    <name>MyPlugin</name>
    <version>1.0</version>
    <idea-version
        since-build="4081"
        until-build="4109"/>
    ...
</idea-plugin>
```

组件注册

Component可以在`application-components`, `projectcomponents`, 和 `module-components`元素中进行声明。在注册一个组件时，component实现类是必不可少的，而component接口是可选的。

component 接口注册样例

```
<component>
    <interface-class>org.MyComponent</interface-class>
    <implementation-class>org.impl.MyComponentImpl</implementation-class>
</component>
```

在以上注册样例中，你可以忽略component的实现代码，仅使用component的即可。 `org.MyComponent`接口 (或类) 并不需要extend或implement [ApplicationComponent](#) 接口，它只是一个普通的接口或类，该接口的实现代码只需实现 [ApplicationComponent](#)接口即可。

这样的好处是我们不需要再去访问`org.impl.MyComponentImpl`实现类(例如，它是package 可见)，通过接口就可以很好满足你的要求。

在前面我们已经知道如何从容器中获取component，这很容易，但大多数情况下这会让代码很笨重。这也是为何许多component提供特定的静态函数去从容器中获取component实例。

component静态函数去获取component实例的样例

```
package com.intellij.tutorial.helloWorld;
import com.intellij.openapi.project.Project;
...
public abstract class MyProjectComponent {
    public static MyProjectComponent getInstance(Project project) {
        return project.getComponent(MyProjectComponent.class);
    }
    public abstract void foo();
    ...
}
```

在上述的例子中，component的实现类并不是public的，在其他的package中并不能访问该component的实现类，对

component的访问可以通过interface或abstract class进行操作。

注意:

当仅有一个类，且被声明时(无论在`interface-class` 或 `implementation-class`声明)，都被视为component的实现类。

Project component实现[JDOMExternalizable](#)接口后，默认会将component的配置信息保存项目文件(IRP)中，但更改特定的命令可以将component的信息保存到其他位置。

将project component配置信息保存到IWS文件中的样例

```
<component>
  <implementation-class>org.MyProjectComponent</implementation-class>
  <option name="workspace" value="true" />
</component>
```

最后还有一点，在引用接口或实现类时，名称中(xml元素的文本值或属性值)不要包含空格和换行符，在加载插件时会引起错误。

错误的组件注册样例

```
<component>
  <interface-class>
    org.MyComponent
  </interface-class>
  <implementation-class>
    org.impl.MyComponentImpl
  </implementation-class>
</component>
```

本地化

在plugin.xml文件中你可以设置插件的本地化，只需设置resource bundle即可，如下：

文件: plugin.xml

```
<idea-plugin>
  <name>MyPlugin</name>
  <description>My description</description>
  <resource-bundle>org.MyBundle</resource-bundle>
  ...
</idea-plugin>
```

文件: /org/MyBundle_en.properties:

```
plugin.MyPlugin.description=My description
...
```

The property key is based on the plugin name with the "plugin" prefix and tag name suffix. The `descriptionn` tag is available for localization.

插件发布

在总结这篇文章之前，我还想告诉你如何上传插件至插件库(<http://plugins.intellij.net>)。

创建归档文件

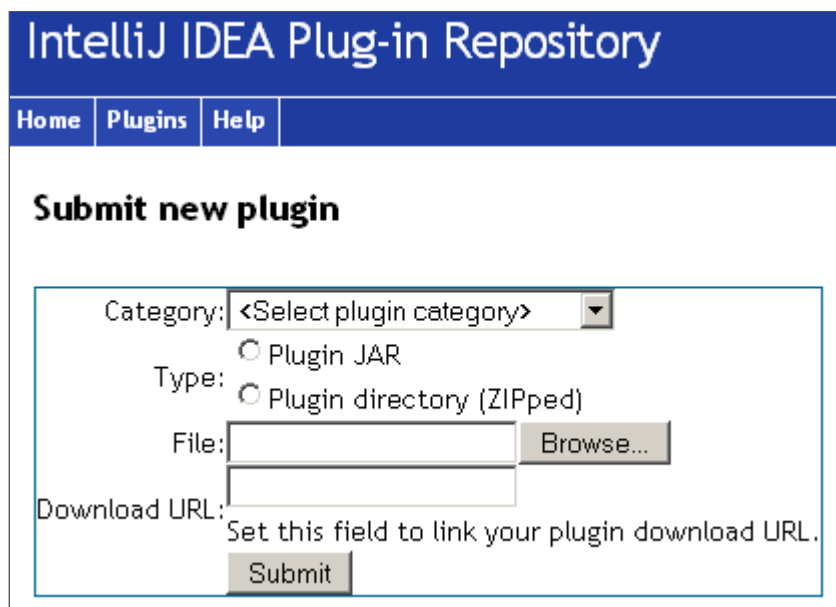
在上传插件之前，你需要为插件创建一个jar或zip文件。在Project工具窗口中，右击module名称，在弹出的对话框中选择 **Prepare Plugin Module '<name>' For Deployment** 菜单项。

注意：

如果模块没有依赖其他开发包，jar文件会被创建。如果依赖其他开发包，zip文件就会被创建，包含其他开发包的jar文件。

上传插件至插件库

上传一个新的插件或者更新插件，请访问<http://plugins.intellij.net> 站点。在这个站点，点击 **Plugins** 标签，然后点击 **Add Plugin** 链接，依据以下屏幕你可以完成插件上传。



The screenshot shows the 'IntelliJ IDEA Plug-in Repository' website. At the top is a navigation bar with 'Home', 'Plugins', and 'Help' links. Below this is a section titled 'Submit new plugin'. Inside this section is a form with the following fields and controls:

- Category:** A dropdown menu with the text '<Select plugin category>'.
- Type:** Two radio buttons: 'Plugin JAR' (selected) and 'Plugin directory (ZIPped)'.
- File:** A text input field followed by a 'Browse...' button.
- Download URL:** A text input field with a note below it: 'Set this field to link your plugin download URL.'
- Submit:** A button at the bottom of the form.

注意：

插件上传后，可能需要一些时间才能在plugin manager出现。因为Server要重新编译插件库中所有的插件，通常这个时间不到1分钟。

通报新版本的插件

当一个插件被创建和更新，插件的作者可以在[Plugins](#)论坛发布消息，告知IntelliJ IDEA用户你插件的功能和特性，更多的人会使用你的插件，同时你也得到更到的反馈。

总结和常用链接

到目前位置，我们讲解IntelliJ IDEA的各种component模型，讲解了[plugin.xml](#)文件中的各项设置，编写了第一个插件，设置配置已经如何去发布等等。

当然，我们不能在一篇文章中告诉你插件开发中的所有知识，但是我们希望这篇文章能够带领你进入IntelliJ IDEA的插件开发行列，扩张IDEA的功能。

[IntelliJ IDEA Plugins](#) - IntelliJ IDEA Plugins Home Page.

[IntelliJ.org TWiki](#) - Wiki site about IntelliJ IDEA and its plugins. Some plugins have its own pages on this site.

[onBoard](#) - online magazine for developers. Here you can find articles written by IntelliJ IDEA developers and other JetBrains staff.
