文章
Qiao Peng · 一月 14, 2021 阅读大约需 7 分钟

使用类投影安装 Caché 应用程序

您好！ 本文介绍另一种为基于 InterSystems Caché 的解决方案创建安装程序的简单方法。
主题将涵盖只需一项操作即可安装或从 Caché 中完全删除的应用程序。
如果您仍在编写需要执行多个步骤才能安装应用程序的安装说明，是时候将这个过程自动化了。

问题的提出

假设我们为 Caché 开发了一个小型实用程序，之后我们想要将其分发。
当然，最好不要让不必要的配置和安装细节打扰到安装它的用户。
此外，这些说明必须非常全面，而且要面向可能对 Caché 一无所知的用户。如果是 Web 实用程序，安装程序不仅会要求用户将其类导入 Caché，而且至少还要配置 Web 应用程序才能对其进行访问，这是相当大的工作量：

当然，所有这些操作都可以通过编程方式执行。 您只需要了解如何实现。
但即使是这样，我们也需要让用户执行操作，例如在终端中执行一个命令。

通过单次导入操作进行安装

Caché 允许我们在类导入期间执行安装。 这意味着用户只需要使用任一方便的方法导入包含类包的 XML 文件：

将 XML 文件拖放到 Studio 区域。
通过管理门户：系统资源管理器 -> 类 -> 导入。
通过终端：do $system.OBJ.Load("C:\FileToImport.xml","ck")。
我们为安装应用程序而预先准备的代码将在类导入和编译后立即执行。
如果用户要卸载我们的应用程序（删除软件包），我们还可以清理应用程序在安装过程中创建的所有内容。

创建投影

为了扩展 Caché 编译器的行为，或者，在我们的示例中，为了在类的编译或反编译期间执行代码，我们需要在软件包中创建一个投影类。 它是一个扩展了 %Projection.AbstractProjection
的类，并重载了它的两个方法：CreateProjection（在编译过程中执行）和
RemoveProjection（在重新编译或删除类时触发）。

通常，将这个类命名为 Installer 是个好方法。 我们来看一个名为 “ MyPackage ” 的软件包的简单安装程序示例：

```
Class MyPackage.Installer Extends %Projection.AbstractProjection [ CompileAfter = (Class1, Class2) ]
{

Projection Reference As Installer;

/// This method is invoked when a class is compiled.
ClassMethod CreateProjection(cls As %String, ByRef params) As %Status
{
  write !, "Installing..."
}

/// This method is invoked when a class is 'uncompiled'.
ClassMethod RemoveProjection(cls As %String, ByRef params, recompile As %Boolean) As %Status
{
  write !, "Uninstalling..."
}

}
```

这里的行为可以描述为：

- 第一次导入和编译软件包时，只触发 CreateProjection 方法。
- 以后再编译 MyApp.Installer 时，或者导入 “ 新 ” 的安装程序类覆盖 “ 旧 ” 类时，将触发旧类的 RemoveProjection 方法，且 %recompile 参数等于 1，之后调用新类的 CreateProjection 方法。
- 删除软件包（同时删除 MyApp.Installer）时，将只调用 RemoveProjection 方法，参数 recompile = 0。

还需要注意以下几点：

- 类关键字 CompileAfter 应该包括应用程序的类名列表，在执行投影类的方法之前，需要对它们进行编译。

始终建议在此列表中填入应用程序中的所有类，因为如果安装过程中出错，我们不需要执行投影类的代码；

- 两个方法都接受 cls 参数 - 它是顶级类名，在我们的示例中为 MyApp.Installer。这个理念来自于创建投影类的本义 - 通过从派生自 %Projection.AbstractProjection 的类再派生，可以单独为我们的应用程序的任何类制作“安装程序”。只有在这种情况下才会体现出意义，但对于我们的任务来说是多余的；
- CreateProjection 和 RemoveProjection 方法都接受第二个参数 params - 它是一个关联数组，以“参数名称”-“值”对的形式处理有关当前编译设置和当前类的参数值的信息。 通过执行 zwrite params 可以非常容易地探索该参数的内容；
- RemoveProjection 方法接受 recompile 参数，只有删除类时，该参数才等于 0，重新编译时不等于 0。

类 %Projection.AbstractProjection 还有其他方法，我们可以重新定义这些方法，但我们的任务并不需要这样做。

一个示例

让我们更深入地了解为我们的实用程序创建 Web 应用程序的任务，并创建一个简单示例。
假设我们的实用程序是一个 REST 应用程序，在浏览器中打开时，它只发送一个响应“I am installed!”。
要创建这样的应用程序，我们需要创建一个描述它的类：

```
Class MyPackage.REST Extends %CSP.REST
{

XData UrlMap
{
<Routes>
 <Route Url="/" Method="GET" Call="Index"/>
Routes>
}

ClassMethod Index() As %Status
{
 write "I am installed!"
 return $$$OK
}

}
```

创建并编译该类后，我们需要将其注册为 Web 应用程序入口点。 我在本文的顶部图示了如何进行配置。
在执行所有这些步骤后，最好通过访问 http://localhost:57772/myWebApp/
来检查我们的应用程序是否正常工作（注意以下几点：1. 末尾的斜线是必需的；2. 端口 57772
在您的系统中可能有所不同。 它将匹配您的管理门户端口）。

当然，所有这些步骤都可以通过 Web 应用程序创建方法 CreateProjection 以及删除方法 RemoveProjection
中的一些代码来自动执行。 在这种情况下，我们的投影类如下所示：

```
Class MyPackage.Installer Extends %Projection.AbstractProjection [ CompileAfter = MyPackage.REST ]
{

Projection Reference As Installer;

Parameter WebAppName As %String = "/myWebApp";

Parameter DispatchClass As %String = "MyPackage.REST";

ClassMethod CreateProjection(cls As %String, ByRef params) As %Status
{
 set currentNamespace = $Namespace
 write !, "Changing namespace to %SYS..."
```

```
 znspace "%SYS" // we need to change the namespace to %SYS, as Security.Applications class exists only there
 write !, "Configuring WEB application..."
 set cspProperties("AutheEnabled") = $$$AutheUnauthenticated // public application
 set cspProperties("NameSpace") = currentNamespace // web-application for the namespace we import classes to
 set cspProperties("Description") = "A test WEB application." // web-application description
 set cspProperties("IsNameSpaceDefault") = $$$NO // this application is not the default application for the
namespace
 set cspProperties("DispatchClass") = ..#DispatchClass // the class we created before that handles the requests
 return ##class(Security.Applications).Create(..#WebAppName, .cspProperties)
}

ClassMethod RemoveProjection(cls As %String, ByRef params, recompile As %Boolean) As %Status
{
 write !, "Changing namespace to %SYS..."
 znspace "%SYS"
 write !, "Deleting WEB application..."
 return ##class(Security.Applications).Delete(..#WebAppName)
}

}
```

在此示例中，每次编译 MyPackage.Installer 类都将创建一个 Web 应用程序，每次“反编译”都将其删除。最好在创建或删除应用程序之前检查该应用程序是否存在（例如，使用 ##class(Security.Applications).Exists(“Name”)），但是为了使本示例简单起见，这个作业就留给阅读本文的读者来完成了。

在创建 MyPackage.REST 和 MyPackage.Installer 类之后，我们可以将这些类导出为一个 XML 文件，并将该文件分享给所有有需要的人。导入此 XML 的用户将自动设置 Web 应用程序，然后可以开始在浏览器中使用。

结果

与 InterSystems 社区上介绍的使用 %Installer 类部署应用程序的方法不同，这种方法有以下优点：

1. 使用“纯”Caché ObjectScript。至于 %Installer，需要用特定标签来填充 xData 块，大量文档对此进行了介绍。
2. 安装我们的应用程序的方法是在类编译后立即执行的，我们不需要手动执行；
3. 如果类（包）被删除，将自动执行删除我们的应用程序的方法，这不能通过使用 %Installer 来实现。

我的项目中已经使用这种应用程序安装方法 - Caché WEB Terminal、Caché Class Explorer 和 Caché Visual Editor。您可以在此处找到 Installer 类的示例。

顺便提一下，开发者社区还有一个帖子介绍了投影的功能，作者是 John Murray。

另外值得一提的是 Package Manager 项目，该项目旨在让 InterSystems 数据平台的第三方应用程序只需通过一个命令或一次点击即可安装，就像类似 npm 的包管理器一样。

#对象数据模型 #终端 #编译器 #部署 #Caché

源 URL:

https://cn.community.intersystems.com/post/%E4%BD%BF%E7%94%A8%E7%B1%BB%E6%8A%95%E5%BD%B1%E5%AE%89%E8%A3%85-cach%C3%A9-%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F