[Liferay中的ActionRequest与RenderRequest 2](#_Toc1772507503)

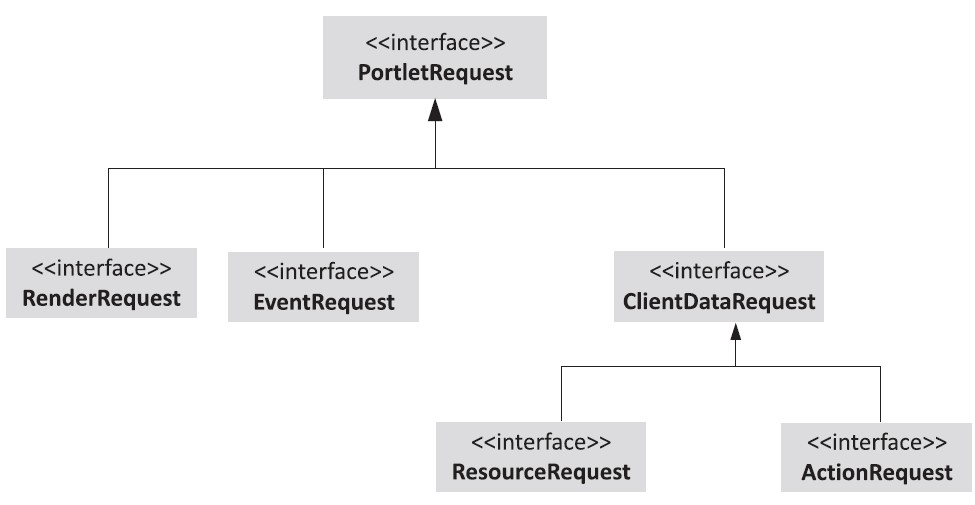
[如何理解依赖注入 3](#_Toc1134484936)

# Liferay中的ActionRequest与RenderRequest

## ActionRequest与RenderRequest简介

首先ActionRequest和RenderRequest都是PortletRequest的子类。但是他们两个又是不同的两个对象。

## PortletRequest的整个对象结构如下图：



**ActionRequest**：*ActionRequest的作用域是portlet在动作处理阶段*，在此阶段中，portlet不知道应该怎么显示他自己，不知道是以html还是vml等内容呈现，不知道portlet的窗口模式是最大化，还是最小化。

**RenderRequest**：**RenderRequest的作用域是portlet的渲染阶段**，在此阶段中portlet知道他自己要怎么显示，显示哪些内容，portlet的窗口状态是什么，是否允许显示等。

简单的说就是**RenderRequest负责显示，ActionRequest负责处理。**

在Liferay中的使用

在Liferay中什么时候使用RenderRequest和ActionRequest的呢？

当我们在portlet中使用标签<portlet:renderURL>和<defineObjects>时，一般后台调用就是对应的doview/doedit等方法，里面的request为RenderRequest。

当我们在Portlet中使用标签<portlet:actionURL>时，后台对应的方法为processAction或者我们在actionURL里面定义的name的方法名，里面的request为ActionRequest。

那我们什么时候使用<portlet:renderURL>和什么时候使用<portlet:actionURL>呢？

比如我现在有一个portlet，此portlet功能为默认显示一个新闻列表，在此页面上有新闻的搜索查询，有新闻的新增、修改等功能。

上面的默认显示新闻列表，这个时候是走的后台的doview方法，里面的request为RenderRequest。在此列表上有一个根据关键词查询新闻的功能，希望查询后还是在此页面显示，这个时候我们一般是将这个查询的请求的地址使用<portlet:renderURL>,在doview里面接收参数，进行查询的处理逻辑后显示。

当我点击新建新闻的按钮时，需要跳转到新增页面，这个时候这个新增的按钮触发时请求的页面，我们一般对此地址使用<portlet:actionURL>。在此新增页面上执行表单提交时，一般也是使用<portlet:actionURL>。

总结一下：

当在portlet的doview、doedit、dohelp等窗口模式下的页面，执行完动作后不跳转其他页面，需要返回这些页面或者需要重新渲染这些页面的时候，使用<portlet:renderURL>。

当我们在portlet中的doview、doedit、dohelp等窗口模式下**需要跳转到其他页面的时候使用**<portlet:actionURL>，在其他页面中**执行表单的处理、跳转等操作时**，也使用<portlet:actionURL>。

# 如何理解依赖注入

举个例子，组件A中有类ClassA，组件B中有接口IB和其对应的实现类B1和B2。

那么，现在ClassA需要利用IB接口来做一些事情，例如：

public class ClassA {  
 public void DoSomething() {  
 IB b = ???  
 b.DoWork();  
 }  
}

现在的问题来了，IB b = ??? 中这三个???要写什么代码？是要写成 IB b = new B1()，还是要写成IB b = new B2() ?

不管是哪一种，都会让ClassA强依赖于IB的实现。

在上面这种方案中，ClassA通过new一个B1或B2来实现对IB的依赖的获取，换句话说，ClassA在主动获取依赖。

这样的设计会让ClassA很难扩展，那我们要改良设计：使用依赖注入。上面说到了，问题出在new这里，也就是依赖是Class去主动获取的，那我们就要解决这个问题：不要去主动获取对IB的依赖（通过new），而让这个依赖从ClassA的外面“注入”进来。注入有多种方式，比较常用的一种是通过构造函数注入，那么，我们要把ClassA改成：

public class ClassA {  
 private IB b;  
  
 public ClassA(IB b) {  
 this.b = b;  
 }  
  
 public DoSomething() {  
 this.b.DoWork();  
 }  
}

可以看到，通过把IB这个依赖从构造函数中“注”进来后，ClassA就不依赖IB的实现了。还可以发现，这个重构过程中，我们是把"ClassA主动获取对IB的依赖”变成“把对IB的依赖从外部注入到ClassA中”，依赖的方向反转了，所以，依赖注入又称“控制反转”。

上面是依赖注入的所有内容。****依赖注入不涉及任何反射、IoC框架（如Unity, Autofac）等内容。要注意这点。****