# Toolkit5.5 Razor简介

## 背景

Toolkit5的Razor来自于移植共享软件RazorEngine，而Toolkit5.5在移植RazorEngine时，发现提供的版本，无法正常运行，因此改移植共享软件RazorLight。

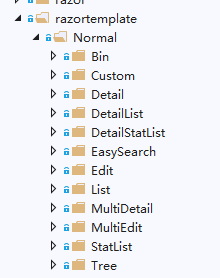
两个版本的不同，导致Razor文件存在一些差异（差异请参看《Razor升级需要注意的问题》）。由于新的版本引入，解决了以前Razor不能调试的难题。同时，归功于VS2019的编辑器或者新的Razor框架（不确定到底是谁），编辑Razor文件也变得更加容易，编辑器提供的报错信息相当准确。

Toolkit5的早期版本中，为每种模型都提供了一个RazorPageMaker，如SingleRazorPageMaker，MasterDetailRazorPageMaker等等。在实践中，发现这种处理，在模型扩展上存在诸多不便。在Toolkit5后期，提供了新的RazorPageTemplatePageMaker和RazorModuleTemplatePageMaker，新版的Toolkit5.5采用了这个设计方案，放弃了原有的的方案。也就是说，在Toolkit5.5的最新版本中，将不再有SingleRazorPageMaker，MasterDetailRazorPageMaker等标签（.net framework的Toolkit5.5版本，可能继续存在）。

Toolkit5中，有很大的需求希望能够存在自己的模板库，但是由于技术所限，没有做到。因此，都以修改razortemplate的现有文件作为解决方案。Toolkit5.5正式引入模板库，允许自行开发并应用自己的模板库，而无须修改原有文件。

## Razor模板库

Toolkit5.5已经从架构到实现上完成了模板库的处理。以后，可以自行设计适合自己的模板库，而无须覆盖原有的Toolkit提供的标准模板库。



Normal目录是Toolkit5.5提供的标准的razor模板库。如果，对此模板不满意，可以在razortemplate下，建立一个新的目录，目录名随意。子目录结构务必和Normal下的子目录结构一致。

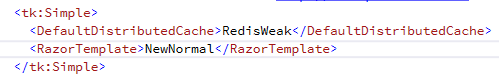
每个子目录下，凡是符合**\*main.cshtml**和**\*template.cshtml**命名的必须严格保留。其他的文件随意，根据自己的实际需要，可以多，也可以少。

关于Bin目录，css.cshtml和js.cshtml是被共享调用的，所以根据自己的需要保留或者放弃。TraceManager.cshtml、FactoryDetail.cshtml和ConfigDetail.cshtml是TracePage使用的，不需要重新构造。ErrorPage.cshtml是出现ErrorPageException后显示的界面，如果觉得显示风格不一致可以重写。Exception.cshtml是有例外后显示的错误提示，如果觉得显示风格不一致可以重写，通常情况下，这个是开发时看的，一般不需要重写。

目录构造完毕后，我们需要修改一个配置，让系统不使用Normal，而是使用新的目录

在新的Default.xml中，我们加入<RazorTemplate>新的目录名</RazorTemplate>就可以了。

如下图：



不写这个配置，或配置中写Normal都是调用Tk5.5提供的标准Razor模板库

## PageTemplate

### 简介

Toolkit5开始，使用razor模板来展示既定的页面。如果用户需要扩展，可以编写自己的razor文件，定义一些section来重载模板中的既有内容。

PageTemplate就是用来处理这个问题的。它提供一个页面的模板入口文件名，默认的RazorData（用来更改界面上的一些UI显示），和使用的Razor Engine（即使用的基类）。

系统定义了IPageTemplate来完成上述功能。

### IPageTemplate

public interface IPageTemplate

{

string GetTemplateFile(ISource source, IInputData input, OutputData outputData);

object GetDefaultPageData(ISource source, IInputData input, OutputData outputData);

string GetEngineName(ISource source, IInputData input, OutputData outputData);

}

IPageTemplate接口定义的三个方法

* GetTemplateFile，返回模板入口文件名称
* GetDefaultPageData，返回默认的PageData
* GetEngineName，返回使用的RazorEngine，系统当前有三个，分别是Toolkit，List和MultiEdit。本质上，就是三个基类。正常情况下，使用Toolkit即可。一般使用常量RazorUtil.TOOLKIT\_ENGINE\_NAME

### 具体实现

IPageTemplate采用代码插件的形态进行注册。我们举两个例子，看其实现。

先以单表的编辑页面为例：

[InstancePlugIn]

[PageTemplate(DefaultModelCreator = "DataSetEdit", Author = "YJC", CreateDate = "2017-04-10",

Description = "Edit页面模板")]

internal class EditPageTemplate : IPageTemplate

{

public static readonly IPageTemplate Instance = new EditPageTemplate();

private EditPageTemplate()

{

}

#region IPageTemplate 成员

public string GetTemplateFile(ISource source, IInputData input, OutputData outputData)

{

return WebRazorUtil.GetTemplateFile("Edit/template.cshtml");

}

public object GetDefaultPageData(ISource source, IInputData input, OutputData outputData)

{

return new NormalEditData();

}

public string GetEngineName(ISource source, IInputData input, OutputData outputData)

{

return RazorUtil.TOOLKIT\_ENGINE\_NAME;

}

#endregion IPageTemplate 成员

}

GetDefaultPageData和GetEngineName返回都是中规中矩。

GetTemplateFile最初的实现是return “^Normal/Edit/template.cshtml”;，这是基于razortemplate的模板地址路径。由于系统支持的模板库，所以使用WebRazorUtil.GetTemplateFile来得到具体的配置路径。这也就是为什么要求模板库的子目录和文件名必须一致的原因。

EditPageTemplate类是代码插件，所以定义了PageTemplateAttribute，这个Attribute和正常的插件Attribute有稍许不同，多了DefaultModelCreator属性。它的作用是解决系统存在DataSet和Object两种结果集的问题。在Toolkit5的早期版本中，两种结果集就需要两个模板库来解决，造成大量的代码浪费，同时也造成更新困难。这个ModelCreator尝试将两种数据结构进行整合，这样同一类型页面，由于不同的ModelCreator，使用相同的模板即可。（当然，最新的Vue让数据和UI脱钩，这个问题就会完全解决）

EditPageTemplate类的定义是典型的单例模式，系统采用实例插件的方式进行注册，因此，在类上标注了InstancePlugInAttribute，同时在类中声明了名称为Instance的静态只读变量（实例插件要求变量必须是静态，名称为Instance或者INSTANCE，是否只读看各人喜好）。

下面，在看看单表List的实现：

[InstancePlugIn]

[PageTemplate(DefaultModelCreator = "DataSetList", Author = "YJC", CreateDate = "2017-04-10",

Description = "List页面模板")]

internal class ListPageTemplate : IPageTemplate

{

public static readonly IPageTemplate Instance = new ListPageTemplate();

private ListPageTemplate()

{

}

#region IPageTemplate 成员

public string GetTemplateFile(ISource source, IInputData input, OutputData outputData)

{

if (input.QueryString["GetData"] == "Page" || input.IsPost)

return WebRazorUtil.GetTemplateFile("List/listmain.cshtml");

else

return WebRazorUtil.GetTemplateFile("List/template.cshtml");

}

public object GetDefaultPageData(ISource source, IInputData input, OutputData outputData)

{

return new NormalListData();

}

public string GetEngineName(ISource source, IInputData input, OutputData outputData)

{

return RazorUtil.LIST\_ENGINE\_NAME;

}

#endregion IPageTemplate 成员

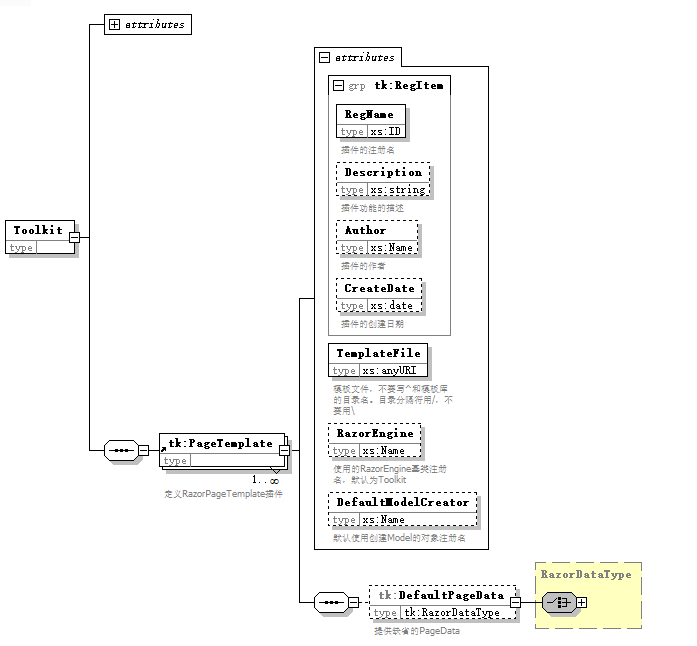
}

其他地方都不解释了，这里着重解释一下GetTemplateFile方法。在List页面中，分页和查询都是返回部分html的，而不是全部。在分页是Url参数会添加GetData=Page的参数，而查询是通过Post方式提交的。因此，这里的逻辑的代码就是，当上述两种情况出现是，使用listmain.cshtml渲染结果，否则就是用template.cshtml获取整个页面。

### Xml配置代码插件

IPageTemplate通过代码插件的方式进行注册。同时，为了方便开发，也提供了Xml配置的方式进行插件注册。

PageTemplate的配置插件schema是v5/ RazorPageTemplate.xsd，格式如下：



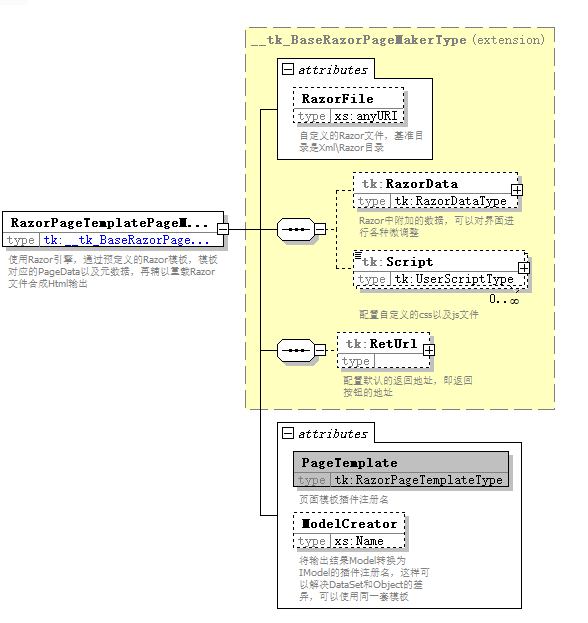
这个定义还是比较一目了然的，和对应的实现方法，属性相匹配。

通过这个schema配置出来的Xml必须放在Xml的**RazorConfigTemplate**目录下，并以**PageTemplate.xml**结尾，才能被系统识别。可以在RazorConfigTemplate建立任意级子目录，以方便归档。

### RazorPageTemplatePageMaker

RazorPageTemplatePageMaker是根据已有的IPageTemplate注册名，使用Razor引擎来展示页面的PageMaker。

它的schema定义如下：



PageTemplate属性是必须填入的，它就是IPageTemplate的插件注册名。

ModelCreator属性用来重载IPageTemplate定义的默认的ModelCreator。

RazorFile属性是用户自己定义的razor文件，可以重载部分section。

RazorData节点是用户用来根据需求，自行定义对应页面的PageData。

Script节点是用户添加额外引用的css或者js

RetUrl节点是重载返回按钮的默认返回地址

## ModuleTemplate

### 简介

PageTemplate只能实现展示一个预制的模板页面。实际中，一个功能往往多个模板，还有其他相应的PageMaker组成。我们以单表功能为例：Insert/Update页面在Get时，使用注册名为Edit的PageTemplate，Post时，使用PostPageMaker；Detail页面使用注册名为Detail的PageTemplate；List页面使用注册名为List的PageTemplate；Delete功能使用PostPageMaker。

ModuleTemplate就是用来解决模块化的配置问题，将若干PageTemplate和PageMaker整合起来。方便整合的功能配置。和Toolkit5的各个有针对性的PageMaker不同，这个ModuleTemplate相对通用，因此会少了一些针对性的定义，但是可以大多数兼容，且扩展方便。

### IModuleTemplate

IModuleTemplate的定义和IPageTemplate相比要复杂很多，接口定义如下：

public interface IModuleTemplate

{

IEnumerable<PageTemplateInfo> PageTemplates { get; }

IEnumerable<PageMakerInfo> CreatePageMakers(IPageData pageData);

void SetPageData(ISource source, IInputData input, OutputData outputData, object pageData);

void SetPageMaker(ISource source, IInputData input, OutputData outputData, IPageMaker pageMaker);

}

public class PageTemplateInfo

{

public PageTemplateInfo(Func<ISource, IInputData, OutputData, bool> function, string pageTemplate)

{

TkDebug.AssertArgumentNull(function, "function", null);

TkDebug.AssertArgumentNullOrEmpty(pageTemplate, "pageTemplate", null);

Function = function;

PageTemplate = pageTemplate;

}

public Func<ISource, IInputData, OutputData, bool> Function { get; private set; }

public string PageTemplate { get; private set; }

public string ModelCreator { get; set; }

}

public class PageMakerInfo

{

public PageMakerInfo(Func<ISource, IInputData, OutputData, bool> function,

IPageMaker pageMaker)

{

TkDebug.AssertArgumentNull(function, "function", null);

TkDebug.AssertArgumentNull(pageMaker, "pageMaker", null);

Function = function;

PageMaker = pageMaker;

}

public Func<ISource, IInputData, OutputData, bool> Function { get; private set; }

public IPageMaker PageMaker { get; private set; }

}

IModuleTemplate的PageTemplates属性，要求提供一组PageTemplate的集合；CreatePageMakers要求提供一组PageMaker的集合。SetPageData方法，允许对RazorData提供一些调整，比如设置Dialog模式。SetPageMaker允许对PageMaker使用前，调整部分属性的值。

### 具体实现

ModuleTemplate的代码实现比较复杂，而且由于一些共通性，所以采用了继承的方式。

BaseModuleTemplate定义了基础的架构，实现IModuleTemplate接口。

public abstract class BaseModuleTemplate : IModuleTemplate

{

private bool fInitPageMaker;

private bool fInitPageTemplate;

protected BaseModuleTemplate()

{

Templates = new List<PageTemplateInfo>();

PageMakers = new List<PageMakerInfo>();

}

#region IModuleTemplate 成员

public IEnumerable<PageTemplateInfo> PageTemplates

{

get

{

InternalInitPageTemplate();

return Templates;

}

}

public IEnumerable<PageMakerInfo> CreatePageMakers(IPageData pageData)

{

InternalInitPageMakers();

return PageMakers;

}

public virtual void SetPageData(ISource source, IInputData input,

OutputData outputData, object pageData)

{

}

public virtual void SetPageMaker(ISource source, IInputData input,

OutputData outputData, IPageMaker pageMaker)

{

if (input.IsPost && (input.Style.Style == PageStyle.Update))

{

PostPageMaker maker = pageMaker as PostPageMaker;

if (maker != null)

maker.UseRetUrlFirst = true;

}

}

#endregion IModuleTemplate 成员

public List<PageTemplateInfo> Templates { get; private set; }

public List<PageMakerInfo> PageMakers { get; private set; }

private void InternalInitPageTemplate()

{

if (!fInitPageTemplate)

{

InitPageTemplate();

fInitPageTemplate = true;

}

}

private void InternalInitPageMakers()

{

if (!fInitPageMaker)

{

InitPageMakers();

fInitPageMaker = true;

}

}

protected abstract void InitPageMakers();

protected abstract void InitPageTemplate();

protected void AddPageTemplate(PageTemplateInfo pageTemplateInfo)

{

TkDebug.AssertArgumentNull(pageTemplateInfo, "pageTemplateInfo", this);

Templates.Add(pageTemplateInfo);

}

protected void AddPageMaker(PageMakerInfo pageMakerInfo)

{

TkDebug.AssertArgumentNull(pageMakerInfo, "pageMakerInfo", this);

PageMakers.Add(pageMakerInfo);

}

}

BaseEditModuleTemplate对一个标准增删改查的模块进行了抽象。

public abstract class BaseEditModuleTemplate : BaseModuleTemplate

{

protected BaseEditModuleTemplate(string editTemplate, string detailTemplate,

string listTemplate)

{

TkDebug.AssertArgumentNullOrEmpty(editTemplate, "editTemplate", null);

TkDebug.AssertArgumentNullOrEmpty(detailTemplate, "detailTemplate", null);

TkDebug.AssertArgumentNullOrEmpty(listTemplate, "listTemplate", null);

EditTemplate = editTemplate;

DetailTemplate = detailTemplate;

ListTemplate = listTemplate;

}

public string EditTemplate { get; private set; }

public string ListTemplate { get; private set; }

public string DetailTemplate { get; private set; }

protected override void InitPageTemplate()

{

Templates.Add(new PageTemplateInfo(

InternalRazorUtil.IsStyle(PageStyle.List), ListTemplate));

Templates.Add(new PageTemplateInfo(

InternalRazorUtil.IsStyle(PageStyle.Detail), DetailTemplate));

Templates.Add(new PageTemplateInfo(

InternalRazorUtil.IsEditStyle(false), EditTemplate));

}

protected override void InitPageMakers()

{

PageMakers.Add(new PageMakerInfo(InternalRazorUtil.IsEditStyle(true),

CreateEditPostPageMaker()));

PageMakers.Add(new PageMakerInfo(InternalRazorUtil.IsStyle(PageStyle.Delete),

CreateDeletePostPageMaker()));

}

protected virtual PostPageMaker CreateDeletePostPageMaker()

{

return new PostPageMaker(ContentDataType.Json, PageStyle.Custom,

new CustomUrlConfig(false, false, "ListRefresh"));

}

protected virtual PostPageMaker CreateEditPostPageMaker()

{

return new PostPageMaker(ContentDataType.Json, PageStyle.List, null);

}

}

SingleModuleTemplate定义了单表ModuleTemplate。

[ModuleTemplate(Author = "YJC", CreateDate = "2017-04-21",

Description = "定义单表增删改查的模板")]

internal class SingleModuleTemplate : BaseEditModuleTemplate

{

public SingleModuleTemplate()

: base("Edit", "Detail", "List")

{

}

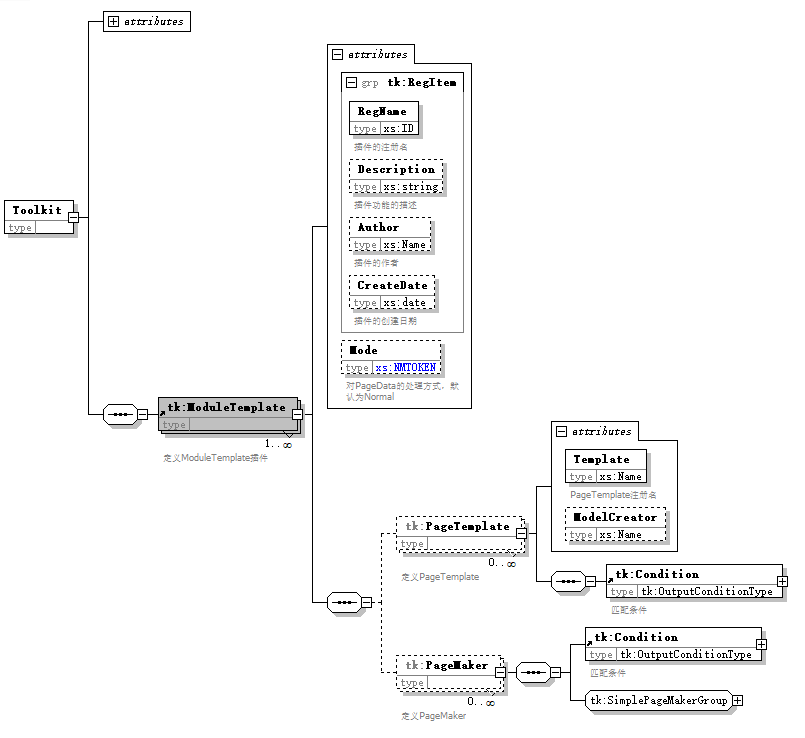
}

这里，我们看到，SingleModuleTemplate也是一个插件。

### Xml配置代码插件

IModuleTemplate和IPageTemplate类似，既可以通过代码插件的方式进行注册。同时，也提供了Xml配置的方式进行插件注册。

ModuleTemplate的配置插件schema是v5/ RazorModuleTemplate.xsd，格式如下：

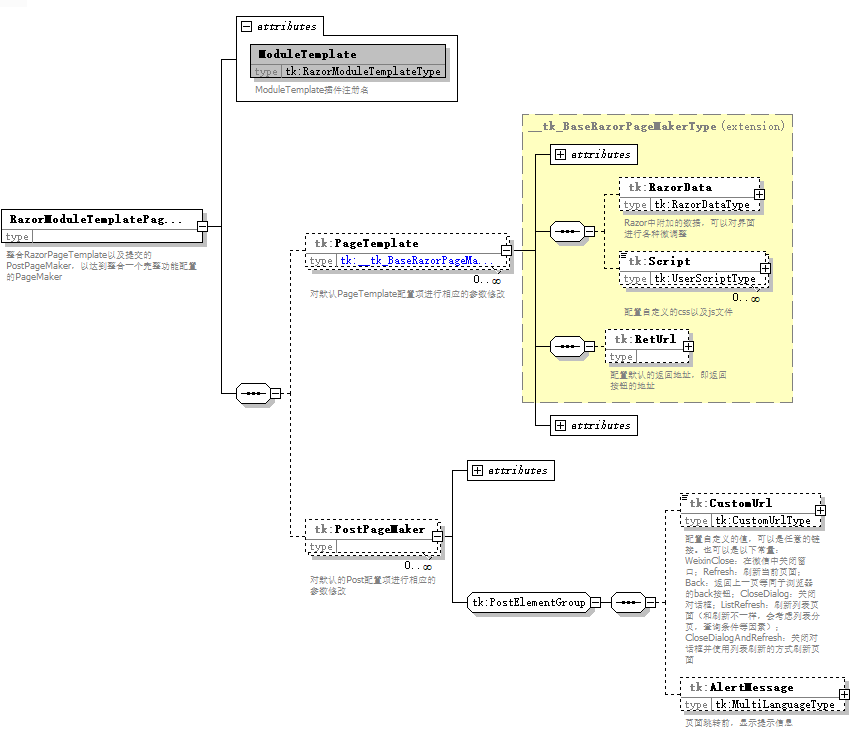


通过这个schema配置出来的Xml必须放在Xml的**RazorConfigTemplate**目录下，并以**ModuleTemplate.xml**结尾，才能被系统识别。可以在RazorConfigTemplate建立任意级子目录，以方便归档。

### RazorModuleTemplatePageMaker

RazorModuleTemplatePageMaker是根据已有的IModuleTemplate注册名，根据注册的条件调度具体的PageMaker。

它的schema定义如下：



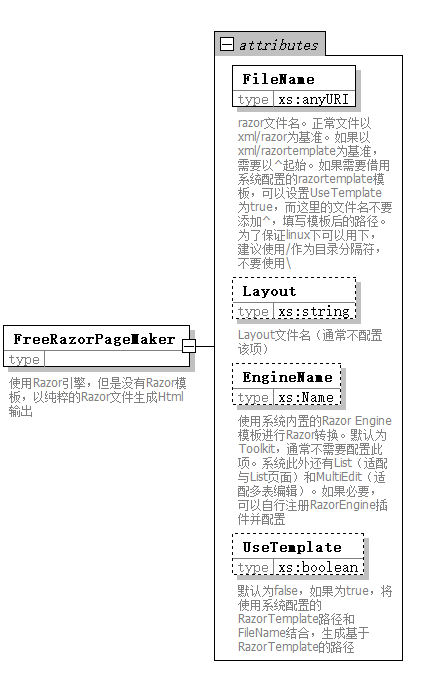
ModuleTemplate属性是必须填入的，它就是IModuleTemplate的插件注册名。

PageTemplate节点可以重载某个PageTemplate的定义。

PostPageMaker节点可以重载某个PostPageMaker的定义。

## FreeRazorPageMaker

FreeRazorPageMaker针对无模板的Razor文件。它的schema定义如下：



这里面的注释已经比较详细的说明了每个属性。

## 小结

新版的Razor中，只有FreeRazorPageMaker、RazorPageTemplatePageMaker、RazorModuleTemplatePageMaker。和Toolkit5相比，PageMaker大幅减少。同时，采用Xml配置插件的方式，可以方便的扩充自己的PageTemplate和ModuleTemplate。

由于新版的设计，使用的不够多，尚不清楚它存在的缺陷。将根据使用的具体反馈，对相关细节进行调整。