# Toolkit5.5和Toolkit5的异同点

Toolkit5.5的开发背景是基于最新的.net core，让Toolkit可以借助于.net core在多平台上运行。并且考虑到以前的一贯性，打算继续支持.net framework。

Toolkit5是基于.net framework4开发的，使用的C#1.0-4.0的语言特性。开发出的Web应用只能在IIS上跑。

Toolkit5.5基于.net standard2.0，.net core2.2和.net framework4.8.2开发的，部分新代码使用了C#6.0-8.0的语言特性。基于.net core的版本不仅可以在Windows（可以挂在IIS上，也可以单独跑）上，还可以基于linux运行，在Mac，Ubuntu，CentOS测试过上都可以运行。

## Toolkit5.5和Toolkit5的相同点

Toolkit5.5在设计之初，并没有打算对Toolkit5进行伤筋动骨的大改。只考虑尽快在.net core上可以跑Toolkit。所以，只要Toolkit5的代码能编译通过，基本上拿来就用，没有做什么改动，除非发现BUG。

Toolkit5中的Core，Core.Extension，MetaData，Data，AdoData的代码，90%以上被成功复制。因此，基于现存这些DLL的代码，基本上拿来就能编译通过。

在升级Toolkit5.5之前，曾经对Core中读写Xml，Json的核心模块做过深度改写，支持了读写List，Dictionary类型的数据，这块5和5.5一致。之所以增加这块，主要是满足JWT代码的支持。同时，也意外的解决了Toolkit5原先不能读写Json数组嵌套的问题。

Toolkit5.5的DataXml，没有变化。从结构到定义和Toolkit5完全一致。Toolkit5.5的ModuleXml在结构上和Toolkit5的结构完全一致，但是由于Razor引擎是重新开发的，所以涉及到Razor的PageMaker需要改动。其他诸如Decoder的Xml，没有变化，直接复制就行。

## Toolkit5.5和Toolkit5的不同点

### Web端完全不同

.net core是基于跨平台的。因此，微软放弃了原有的IIS，而重新设计了一个轻量级的web server来支持跨平台。这个server从公开的API来看，相当轻量，不过web server应该具有的功能，它通过各种插件都能支持。而且大量的引入了异步编程。

因为这个原因，.net core的web端和.net framework基于IIS的web端是完全不同的。原先IIS传统的Application，Session，Request，Response，Server五个对象在.net core上只有局部对应。

所以，基于.net core的web端和基于.net framework的web端是完全不同的。当然，经过分析，Toolkit提供了方法，两边的ISource，IPageMaker，IRedirector等结构还是完全一致的。WebPage也是高度相似。但是涉及到Request，Response这些具体的web server对象，就无法相同了。

### Url构造不同

由于web端完全不同，所以url的解析上也是不同的。.net core的web server放弃了原先.xxx这种通过扩展名的解析方式，完全靠url上。Toolkit5后期推出的.c方式的url解析方式和现有Toolkit5.5的类似，如果掌握了这种方式，对Toolkit5.5的Url构造应该比较容易理解。（具体参考《Toolkit5.5 Url规则》）

### Razor不同

Toolkit5使用基于RazorEngine的共享软件进行移植开发，Toolkit5.5使用基于RazorLight的共享软件进行移植开发。两个引擎不同，导致Razor文件存在微小的差别（具体差别可以参考《Razor升级需要注意的问题》）。RazorLight中还大量的采用了异步编程，而RazorEngine基本上都是同步的。

新版Razor最大的优点在于可以在集成开发环境中进行调试，此外编辑也非常方便，重复利用了集成环境的优势。

### 引入JWT

由于.net core对Session的弱化，Toolkit5.5放弃了在Session中存储用户信息的方式，而引入Json Web Tokens来标识用户。限于前端的技术，暂时，JWT存储在cookie中，而不是存在http header中。JWT属于标准协议，可以和其他非.net语言，如Java，PHP交互，对于实现单点登录也有好处。

此外，SessionGlobal不在存储在Session中，而是以缓存的形式进行存储。

### 支持DistributeCache

Toolkit 5.5通过支持微软提供的IDistributeCache从而达到支持分布式缓存的结构。这种改变，主要在Toolkit内核中，在外部，只要选用适当的基类，就能切换到分布式缓存上。（具体参看《Toolkit5.5的缓存》）

当前系统支持的分布式缓存是Redis，随着Toolkit5.5的逐步完善，会支持SQL Server上的分布式缓存。

### 权限初始化的修改

Toolkit4引入了IInitialization和IWebInitialization接口后，权限的初始化，都是在IWebInitialization接口的SessionStart方法中完成的。由于.net core引入后，不再有SessionStart和SessionEnd事件接入，原有的初始化变得不合适了。加上SessionGlobal也不在存放在Session中，因此，Toolkit5.5修改了这个初始化过程。

Toolkit5中，正式将权限确定为登录，功能，数据，操作，记录列五大权限系统。其中，只有登录和功能需要在登录时进行初始化。原有的数据权限的初始化，由于数据权限改为配置插件，这块初始化被分解。在Toolkit5.5中，为了方便管理，将需要初始化的登录和功能设计的接口进行了统一，定义了一个新的接口

public interface IAppRightBuilder

{

ILogOnRight CreateLogOnRight();

IFunctionRight CreateFunctionRight();

IMenuScriptBuilder CreateScriptBuilder();

}

同时，将其升格为代码插件。系统提供了一个注册名为Empty的插件，它产生空的权限。如果不配置，默认就是这个权限在起作用。

当自行开发的AppRightBuilder需要注册时，可以在Application.xml中的tk:Application的AppRightBuilder属性中定义。如原先SimpleRight中，就实现了一个注册名为Simple的插件，代码如下：

[InstancePlugIn, AlwaysCache]

[AppRightBuilder(Author = "YJC", CreateDate = "2019-10-11", Description = "基于传统UR\_USERS的简单权限")]

internal class SimpleAppRightBuilder : IAppRightBuilder

{

public static readonly IAppRightBuilder Instance = new SimpleAppRightBuilder();

private SimpleAppRightBuilder()

{

}

public IFunctionRight CreateFunctionRight()

{

return new UserFunctionRight();

}

public ILogOnRight CreateLogOnRight()

{

return new UserLogOnRight();

}

public IMenuScriptBuilder CreateScriptBuilder()

{

return new BootstrapMenuBuilder();

}

}

可以看到，这个插件类，通过私有化构造函数，变成了单例模式。同时定义了名称为Instance的静态变量，并用了InstancePlugInAttribute声明。那么，系统根据这个定义，将Instance变量作为实例插件进行注册。

在Application.xml中定义如下：

<tk:Application AppRightBuilder="Simple">

### DbProvider从AdoData中剥离

.net standard的核心库中不再有System.Data.SqlClient.dll。因此，要使用SqlClient，必须从Nuget获取。为了降低代码的耦合度，AdoData不在支持各种DbProvider，如果需要，可以从Nuget中获取相应的Client（这些Client通常都是.net standard类库），来编写。

Toolkit分别提供了支持Sql Server，注册名为SQL Server的Toolkit.SqlClient和支持MySql，注册名为MySql的Toolkit.MySql来支持这两种数据库。原有的Odbc和OleDb这两种DbProvider不再支持。

由于微软最新推出了Microsoft.Data.SqlClient来替换使用了十多年的System.Data.SqlClient。因此，Toolkit已经开发了基于Microsoft.Data.SqlClient的Toolkit.MsSqlClient，同样支持Sql Server，但是注册名为MS SQL Server。

### Evaluator剥离

Evaluator是Toolkit5引入的，通过表达式树的方式来计算字符串。原先存在于Toolkit.Core.Extension中。在Toolkit5.5中，Core.Extension完全并入Core，但是这块代码有一定的特殊性，.net standard不支持部分API。为此，这块代码被剥离出来。在.net core和.net framework中，封闭都有一个Toolkit.Evaluator，不同的是一个基于.net core 2.2，一个基于.net framework 4.8.2。

在使用上，和Toolkit5相同，无区别。

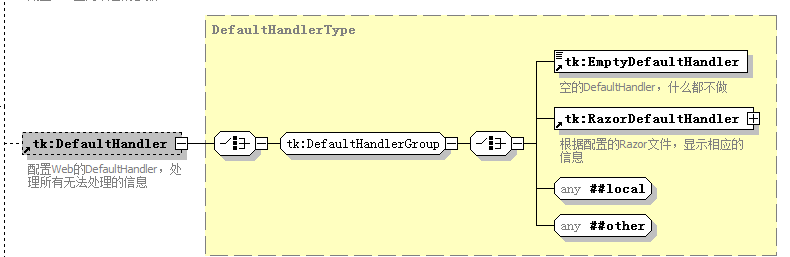
### 取消Default.aspx

Toolkit5及以前的版本，由于使用IIS，所以启动的地址一直都是Default.aspx。但是.net core不再有类似的机构，因此，Toolkit5.5做了如下定义：

如果未登录，直接跳转到Application.xml配置的StartupPath。如果已登录，直接跳转到Application.xml配置的MainPath，并在Url上添加HomePath路径（这个是Toolkit传统的方式）。如果首页不是采用MainPath+HomePath的架构，而直接一个页面，那么MainPath设置为空，将直接跳转到HomePath的配置。

此外，在Toolkit5.5的早期Demo版本中，如果出错或者敲了错误的url，那么界面上都会显示hello, world。这块在release的版本中已经不存在了。碰到无法处理的url，系统将根据配置的DefaultHandler进行处理。默认为空的DefaultHandler，即什么都不做。

DefaultHandler可以在Default.xml中配置，其Xml Schema如下：



RazorDefaultHandler可以显示提示信息，如果觉得上述都不合适，可以自行开发。

### 不支持虚拟目录

Toolkit5及以前版本都是在IIS上部署的，因此，可以配置在虚拟目录下运行。而Toolkit5.5，虽然Windows的发布版本可以配置在IIS上，但是不支持配置在虚拟目录下，只支持配置在网站上。

同时，如果使用dotnet run启动的.net core网站，虽然可以用nginx反向代理到虚拟目录下，但是由于系统内部无法侦测到这个信息，将导致url彻底紊乱。因此，特别规定，**不支持虚拟目录，只支持根节点的网站部署**。

### Web工程

和Toolkit5不一样，Toolkit5.5的.net core的Web工程，需要在csproj文件中添加

<PropertyGroup>

<PreserveCompilationContext>true</PreserveCompilationContext>

<MvcRazorExcludeRefAssembliesFromPublish>false</MvcRazorExcludeRefAssembliesFromPublish>

</PropertyGroup>

这两个参数非常重要，都是razor使用的，没有这些参数，在集成环境中跑razor，一点问题都没有。但是如果publish以后，razor就无法编译，报各式各样的编译错误。

适当的时候，将开发基于.net core的Toolkit工程模板，这样可以在集成环境中直接new这样的工程。