# Toolkit5.5的Data DLL开发指南

当前的.net版本和原先相比，有了很大的变化。.net core的推出，大有取代传统.net framework的趋势。

## DLL的类型

.net core推出后，.net的DLL有三种格式了，传统的.net framework的dll依旧存在，而.net core有自己的dll体系。两种dll使用各自不同的API，因此不能互通。由于.net core和.net framework存在大量相同的API，所以，为了最大程序的共享代码，微软又推出了.net standard，它支持多种平台，包括.net core和.net framework。和.net framework一样，.net core和.net standard都有各自的版本号。

当前系统使用.net standard 2.0，对应的是.net core 2.2和.net framework4.8.2。

在Toolkit5中，很多基础的dll都是采用.net standard格式。这样可以被两个平台共同使用。如Core，Data，MetaData，AdoData等。为了尽可能的使用.net standard，还从WebApp中剥离了大量可以在.net standard中使用的代码，形成一个WebApp.Common。

## Data DLL的开发

在了解了DLL的类型后，我们在开发中，应该尽量选择.net standard这种dll类型。这样，生成的DLL可以在.net core和.net framework中复用。而且，由于Toolkit5.5的核心的DLL基本都是基于.net standard的，所以引用上不存在任何问题。

Data DLL是我们最常见开发的DLL，主要包含了TableResolver插件和Source插件。建议，无特别情况下，Data DLL应该定义为.net standard的DLL。Core，MetaData，Data，AdoData都可以直接引用。必要时，还可以引用WebApp.Common（诸如IPageMaker，IModule这些基础接口都定义在这里）。

正常情况下，TableResolver和Source都是数据逻辑代码，不会需要Web的东西。我注意到，很多时候，大家喜欢使用WebGlobalVariable.Request.QueryString来获取Url上的参数。为了回避这个现象，我在UpdatingEventArgs类中添加了InputData属性，InputData.QueryString和Request.QueryString完全相同。在OnUpdatingRow函数中可以使用。

当然，如果在TableResolver和Source中使用Session或者Response这些对象，那么就只能用.net core或者.net framework的DLL了，两种类型的Web App使用的这些对象完全不同，无法互通。碰到这种情况，只建议将这样的代码归类到.net core或者.net framework的DLL中，不要图省事，把所有的代码都放过来。

## 有关Session的说明

.net core中，不提倡使用Session。虽然，还是支持Session。因此，Toolkit5.5在.net core版本上完全放弃了Session的使用。以前的SessionGlobal对象，虽然继续存在，但是，它不是存储在Session中，而是存储在缓存中。Toolkit5.5中依旧可以通过WebGlobalVariable访问Session来存取内容，但是不建议这么使用。另外，.net core和.net framework的Session有一个非常大的区别，就是.net core的Session不支持存取对象，只支持简单数据类型。如果要存取对象，必须将对象序列化和反序列化。

基于上述原因，不推荐大家使用Session。如果，需要保存数据进行共享，可以考虑使用缓存（关于缓存的构造和使用参考《Toolkit5.5的缓存》）。

## 关于IUserInfo的说明

.net core中，由于Session发生了重大变化。因此SessionGlobal采用了缓存方式。而IUserInfo这个结构，其实并不存放在SessionGlobal中。它也发生了重大的变化。

.net core中，对UserInfo采用了JWT（Json Web Tokens）的编码方式加密存储。这是个共享协议，理论上，其他支持JWT的Web App可以共享使用。

原先打算JWT放在http的headers中，但是后来发现，浏览器直接敲地址时，无法在headers中添加东西，现有的web架构做不到这一点。只能暂时将JWT存储在cookies中。

每次web请求时，系统会自动尝试从cookie中加载这个JWT，一旦加载成功，将存储在HttpContext.User中。所以，SessionGlobal中User实际上是指向HttpContext.User。BaseGlobalVariable.Info也是指向这里。因此，只要cookie中这个JWT有效，那么用户就会自动登录。现在只有登录的页面，才会向cookie中写入JWT，该cookie有效期8小时，且不会超过当天。Logout时，会清除该cookie。有了这个结构后，不用担心Session过期的问题。

由于JWT的特殊性，因此，IUserInfo定义的Data1和Data2属性不要附加任何数据。即使附加也没用，在存取过程中一定会丢失。如果需要这些额外数据，请通过Session或者缓存来保存。

## 关于ISupportTenant接口

Toolkit5定义了Tenant.dll，其中有个重要的接口ISupportTenant，存储租户的Id。只有引用了这个DLL，才能检测是否有TenantId的存在。

由于租户的作用越来越重要，因此，Toolkit5.5将这个接口移入了Core，同时移入的还有TenantUserInfo。同时，对IUserInfo进行了扩展，支持IsSupportTenant和GetTenantValue两个扩展方法。

由于这个接口的提升，后续有关Tenant的支持，会比较方便。

## 工作线程

从Toolkit4开始，就提供了这个功能。工作线程是一个特殊的线程，它只有一个线程，线程的优先级还非常低（即它不会抢占其他线程的工作，不搞喧宾夺主的事情）。如果有任务进入，它就做任务，任务做完，它便挂起。如果同时有很多任务进入，那就需要排队，它会把队列里面所有的任务都处理完成后，再挂起。

由于工作线程工作的特殊性，请不要在任务中使用Session，UserInfo这些与web线程上下文有关的对象，会报错。尽可能做一些耗时，但不重要的任务。

有了这么一个线程后，Toolkit有些事情都是让工作线程来处理的，比如错误文件的写入，Log插件的执行等等。

这里，介绍一下工作线程的使用，有意愿的开发可以使用这个线程，将不重要的任务交给它，同时也能加快响应速度。

要使用工作线程，首先Application.xml的定义中，有个UseWorkThread的属性不能设为false，不设或者设为true，都会启动工作线程。

有了工作线程后，在需要的时候，我们就可以调用了。调用时，我们可能是void的过程，或者有返回值的函数。

对于过程，必须简单，只要执行就行，不需要管理后续的。因此BaseGlobalVariable提供了一个简单的方法：

public IAsyncResult BeginInvoke(Delegate method, params object[] args)

第一个参数是调用的方法，用Delegate包装，后面是这个方法的参数列表。

比如Toolkit在记录错误日志时，就是这样调用：

BaseGlobalVariable.Current.BeginInvoke(new Action<string, string>(FileUtil.VerifySaveFile),

fileName, content);

对于函数，因为有返回值，所以还必须有一个异步的回调用来处理返回值。这里除了调用函数外，还要提供一个回调函数。相比于函数，这个要复杂一些。BaseGlobalVariable还有两个方法，用来处理这个问题：

public IAsyncResult BeginInvoke(AsyncCallback callback, Delegate method,

object[] args, object asyncState)

public object EndInvoke(IAsyncResult result)

public delegate void AsyncCallback(IAsyncResult ar);

第一个函数BeginInvoke比过程使用的BeginInvoke要复杂的多，它第一个参数是回调的Delegate，第二，第三个参数和前面BeginInvoke一样。最后一个参数随意，它会出现在IAsyncResult的属性中，所以根据需要设定，通常都是null。

EndInvoke是在回调中调用，用来获取返回值的。

下面的例子用来验证这个过程：

internal static class WorkThreadTest

{

public static int Add(int a, int b)

{

return a + b;

}

public static void ProcessResult(IAsyncResult ar)

{

var result = BaseGlobalVariable.Current.EndInvoke(ar);

int i = result.Value<int>();

Console.WriteLine(i);

}

public static void TestWorkThread()

{

BaseGlobalVariable.Current.BeginInvoke(ProcessResult, new Func<int, int, int>(Add),

new object[] { 10, 20 }, null);

}

}