Loxodon Framework

release v1.7.5

MVVM Framework for Unity3D （C# & XLua）

开发者 Clark

要求Unity 5.6.0或者更高版本

LoxodonFramework是一个轻量级的MVVM(Model-View-ViewModel)框架，它是专门为Unity3D游戏开发设计的，参考了WPF和Android的MVVM设计，它提供了视图和视图模型的数据绑定、本地化、一个简单的服务容器、配置文件组件、线程工具组件、应用上下文和玩家上下文，异步线程和协程的任务组件等基本组件，同时还提供了一个UI视图的框架。所有代码都基于面向对象面向接口的思路设计，几乎所有功能都可以自定义。而且在数据绑定部分进行了性能优化，在支持JIT的平台上使用的是委托的方式绑定，在不支持JIT的平台，默认使用的是反射，但是可以通过注入委托函数的方式来优化！

本框架使用C#语言开发，同时也支持使用XLua来开发，XLua插件是一个可选项，如果项目需要热更新，那么只要安装了XLua插件，则可以完全使用Lua来开发游戏。

这个插件兼容 MacOSX,Windows,Linux,UWP,IOS and Android等等，并且完全开源。

已测试的平台：

- **PC/Mac/Linux** (.Net2.0 subset; .Net2.0; .Net4.x; .Net Standard 2.0; IL2CPP)
- **IOS** (.Net2.0 subset; .Net2.0; .Net4.x; .Net Standard 2.0; IL2CPP)
- **Android** (.Net2.0 subset; .Net2.0; .Net4.x; .Net Standard 2.0; IL2CPP)
- **UWP(window10)** (.Net2.0 subset; .Net2.0; .Net4.x; .Net Standard 2.0; IL2CPP)

关键特性

- 支持多平台，高扩展性，面向接口开发;
- 支持C#和Lua开发;
- 支持线程和协程的异步结果和异步任务，采用Future/Promise设计模式;
- 提供了多线程组件，线程切换组件和定时执行器;
- 提供了一个消息系统，支持订阅和发布;
- 提供可加密的配置文件，支持对象存取，可自定义类型转换器，扩展功能;
- 提供了本地化支持，与Android的本地化类似，支持基本数据类型、数组、和U3D的一些值类型;
- 支持全局上下文和玩家上下文;
- 提供了一个服务容器，支持注册和注销服务;
- 提供了AlertDialog、Loading、Toast等通用UI控件，支持自定义外观;
- 提供了UI视图的控制和管理功能；
- 提供数据绑定功能:
  - Field绑定，只支持OneTime的模式，因无法支持改变通知;
  - 属性绑定，支持TwoWay双向绑定，值修改自动通知;
  - 普通字典、列表绑定，不支持改变通知;
  - 支持C#事件绑定;
  - 支持Unity3D的EventBase事件绑定;
  - 支持静态类的属性和Field的绑定;
  - 支持方法绑定（包括静态方法）;
  - 支持命令绑定，通过命令绑定可以方便控制按钮的有效无效状态;
  - 支持可观察属性、字典、列表的绑定，支持改变通知，视图模型修改自动更改UI显示;
  - 支持表达式的绑定;

- 支持类型转换器，可以将图片名称转换为图集中的Sprite;
- 可以自定义扩展更多的绑定类型;

下载

- Unity3d官方商店下载
- Github下载

Lua插件安装（可选）

在本框架中，对于Lua语言的支持是通过插件扩展的方式来支持，它依赖腾讯的XLua项目和Loxodon.Framework.XLua插件，在项目的LoxodonFramework/Docs/XLua目录中可以找到Loxodon.Framework.XLua的插件，它是可选的，只有需要热更新并且使用Lua语言开发的项目才需要安装它。具体安装步骤如下，为避免出错，请严格按以下步骤安装。

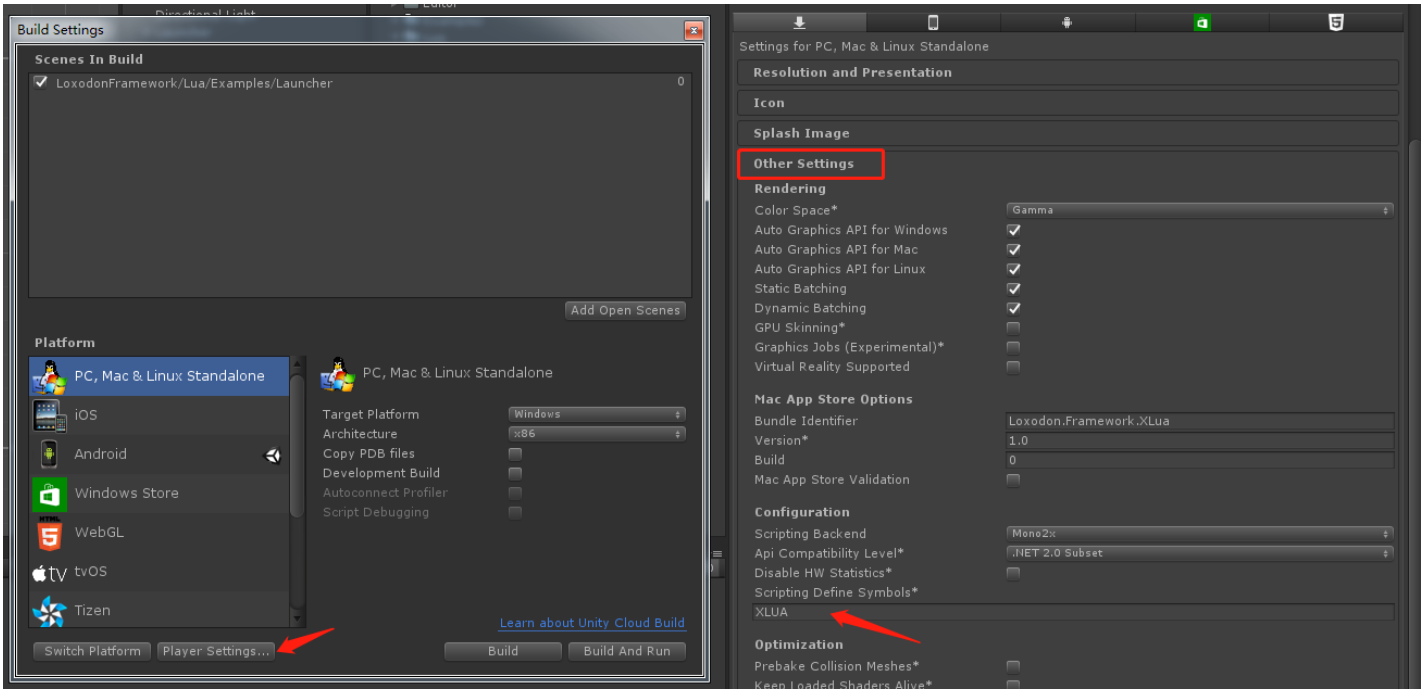
安装XLua

从Xlua的Github仓库下载最新版的XLua，可以使用源码版本Source code.zip或者xlua*v2.x.xx.zip版本（建议使用xlua*v2.x.xx.zip版本，避免和XLua示例类名冲突）。请将下载好的xlua解压缩，拷贝到当前项目中。

XLua下载

配置宏定义

配置Unity3D项目Player Setting/Other Settings/Scripting Define Symbols，添加XLUA的宏定义，为避免将来切换平台时出错，最好将PC、Android、iOS等平台的都配上。



导入Lua插件

在LoxodonFramework/Docs/XLua/目录中，找到Loxodon.Framework.XLua.unitypackage文件，双击导入项目。

如果出现编译错误，请检查是否导入了XLua的Examples目录，这个目录下的InvokeLua.cs文件定义了PropertyChangedEventArgs类，因没有使用命名空间，会导致类名冲突，请删除XLua目录下的Examples文件夹或者给InvokeLua.cs文件中的PropertyChangedEventArgs类添加上命名空间。

查看示例

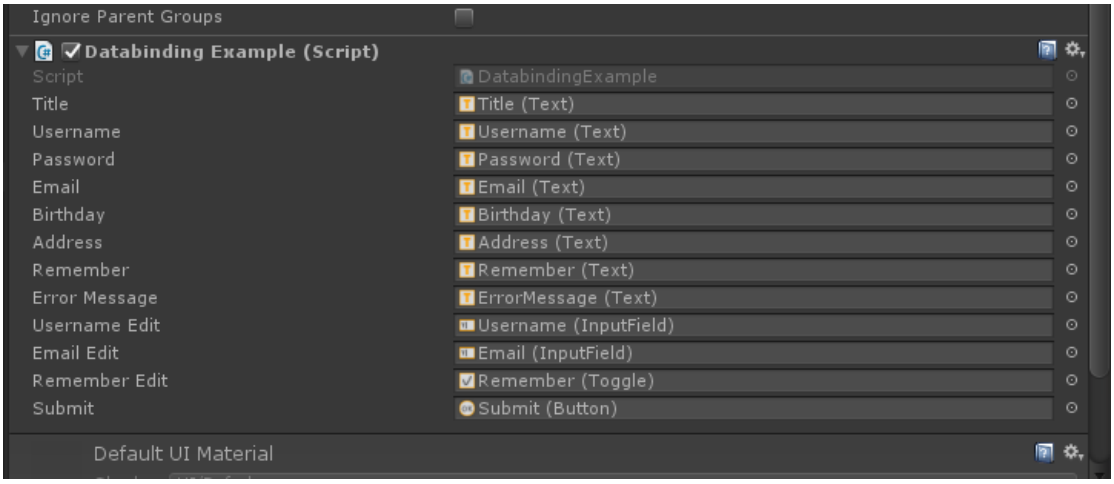
打开LoxodonFramework/Lua/Examples目录，查看示例。

快速入门

创建一个视图，左侧显示一个账号信息，右侧是一个表单，通过提交按钮可以修改左侧的账号信息，现在我们通过框架的视图和数据绑定功能来演示我们是如何做的。界面如下图：

C# 示例

在一个UI视图的根对象上添加视图脚本组件DatabindingExample，并且将UI控件赋值到对应的属性上,这个示例中属性都是通过C#硬编码来定义的，当然你也可以使用动态的属性表VariableArray来动态定义属性，具体可以看Lua的例子，配置好属性后如下图所示。



下面请看代码，我们是如果来定义视图模型和视图脚本的，又是怎么样来绑定视图到视图模型的。

```
/// <summary>
/// 账号子视图模型
/// </summary>
public class AccountViewModel : ObservableObject
{
    private int id;
```

```
    private string username;
    private string password;
    private string email;
    private DateTime birthday;
    private readonly ObservableProperty<string> address = new ObservableProperty<string>();

    public int ID
    {
        get { return this.id; }
        set { this.Set<int>(ref this.id, value, "ID"); }
    }

    public string Username
    {
        get { return this.username; }
        set { this.Set<string>(ref this.username, value, "Username"); }
    }

    public string Password
    {
        get { return this.password; }
        set { this.Set<string>(ref this.password, value, "Password"); }
    }

    public string Email
    {
        get { return this.email; }
        set { this.Set<string>(ref this.email, value, "Email"); }
    }

    public DateTime Birthday
    {
        get { return this.birthday; }
        set { this.Set<DateTime>(ref this.birthday, value, "Birthday"); }
    }

    public ObservableProperty<string> Address
    {
        get { return this.address; }
    }
}


/// <summary>
/// 数据绑定示例的视图模型
/// </summary>
public class DatabindingViewModel : ViewModelBase
{
    private AccountViewModel account;
    private bool remember;
    private string username;
    private string email;
    private ObservableDictionary<string, string> errors = new ObservableDictionary<string, string>();

    public AccountViewModel Account
    {
        get { return this.account; }
        set { this.Set<AccountViewModel>(ref account, value, "Account"); }
    }

    public string Username
    {
        get { return this.username; }
        set { this.Set<string>(ref this.username, value, "Username"); }
    }

    public string Email
```

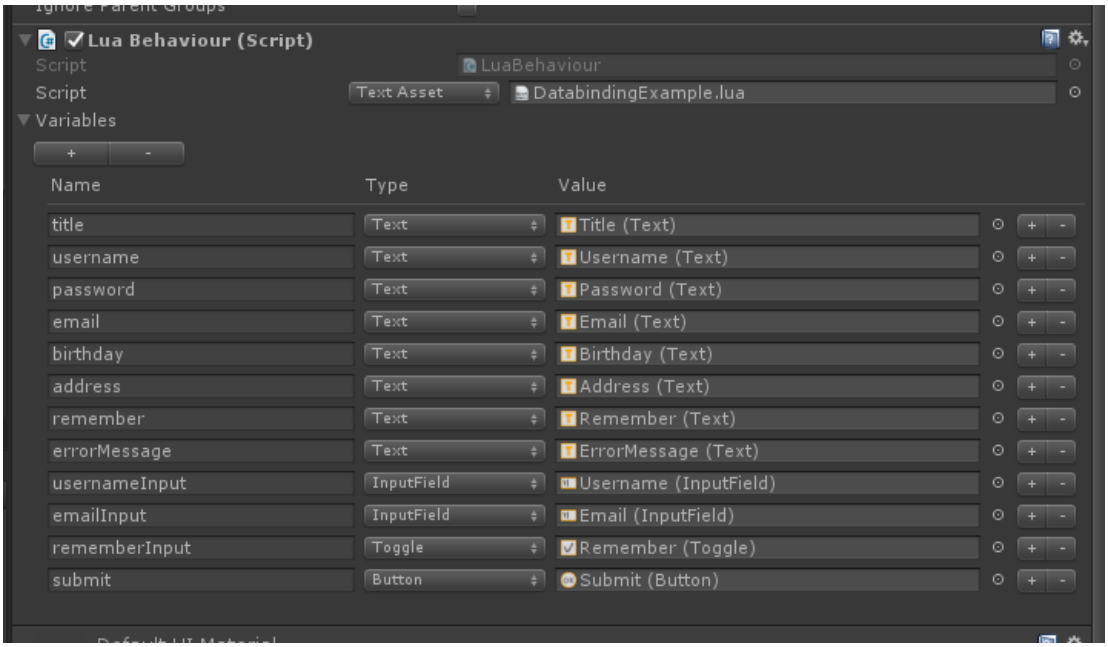
```
    {
        get { return this.email; }
        set { this.Set<string>(ref this.email, value, "Email"); }
    }

    public bool Remember
    {
        get { return this.remember; }
        set { this.Set<bool>(ref this.remember, value, "Remember"); }
    }

    public ObservableDictionary<string, string> Errors
    {
        get { return this.errors; }
        set { this.Set<ObservableDictionary<string, string>>(ref this.errors, value, "Errors"); }
    }

    public void OnUsernameValueChanged(string value)
    {
        Debug.LogFormat("Username ValueChanged:{0}", value);
    }

    public void OnEmailValueChanged(string value)
    {
        Debug.LogFormat("Email ValueChanged:{0}", value);
    }

    public void OnSubmit()
    {
        if (string.IsNullOrEmpty(this.Username) || !Regex.IsMatch(this.Username, "^[a-zA-Z0-9_-]{4,12}$"))
        {
            this.errors["errorMessage"] = "Please enter a valid username.";
            return;
        }

        if (string.IsNullOrEmpty(this.Email) || !Regex.IsMatch(this.Email, @"^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$"))
        {
            this.errors["errorMessage"] = "Please enter a valid email.";
            return;
        }

        this.errors.Clear();
        this.Account.Username = this.Username;
        this.Account.Email = this.Email;
    }
}


/// <summary>
/// 数据绑定示例视图
/// </summary>
public class DatabindingExample : UIView
{
    public Text title;
    public Text username;
    public Text password;
    public Text email;
    public Text birthday;
    public Text address;
    public Text remember;

    public Text errorMessage;

    public InputField usernameEdit;
    public InputField emailEdit;
    public Toggle rememberEdit;
    public Button submit;
```

```
protected override void Awake()
{
    //获得应用上下文
    ApplicationContext context = Context.GetApplicationContext();

    //启动数据绑定服务
    BindingServiceBundle bindingService = new BindingServiceBundle(context.GetContainer());
    bindingService.Start();

    //初始化本地化服务
    CultureInfo cultureInfo = Locale.GetCultureInfo();
    var provider = new DefaultDataProvider("LocalizationTutorials", new XmlDocumentParser())
    Localization.Current = Localization.Create(provider, cultureInfo);

}

protected override void Start()
{
    //创建账号子视图
    AccountViewModel account = new AccountViewModel()
    {
        ID = 1,
        Username = "test",
        Password = "test",
        Email = "yangpc.china@gmail.com",
        Birthday = new DateTime(2000, 3, 3)
    };
    account.Address.Value = "beijing";

    //创建数据绑定视图
    DatabindingViewModel databindingViewModel = new DatabindingViewModel()
    {
        Account = account
    };

    //获得数据绑定上下文
    IBindingContext bindingContext = this.BindingContext();

    //将视图模型赋值到DataContext
    bindingContext.DataContext = databindingViewModel;

    //绑定UI控件到视图模型
    BindingSet<DatabindingExample, DatabindingViewModel> bindingSet;
    bindingSet = this.CreateBindingSet<DatabindingExample, DatabindingViewModel>();

    //绑定左侧视图到账号子视图模型
    bindingSet.Bind(this.username).For(v => v.text).To(vm => vm.Account.Username).OneWay();
    bindingSet.Bind(this.password).For(v => v.text).To(vm => vm.Account.Password).OneWay();
    bindingSet.Bind(this.email).For(v => v.text).To(vm => vm.Account.Email).OneWay();
    bindingSet.Bind(this.remember).For(v => v.text).To(vm => vm.Remember).OneWay();
    bindingSet.Bind(this.birthday).For(v => v.text).ToExpression(vm => string.Format("{0} ({1})",
     vm.Account.Birthday.ToString("yyyy-MM-dd"), (DateTime.Now.Year - vm.Account.Birthday.Year))).OneWay();
    bindingSet.Bind(this.address).For(v => v.text).To(vm => vm.Account.Address).OneWay();

    //绑定右侧表单到视图模型
    bindingSet.Bind(this.errorMessage).For(v => v.text).To(vm => vm.Errors["errorMessage"]).OneWay();
    bindingSet.Bind(this.usernameEdit).For(v => v.text, v => v.onEndEdit).To(vm => vm.Username).TwoWay();
    bindingSet.Bind(this.usernameEdit).For(v => v.onValueChanged).To(vm => vm.OnUsernameValueChanged(""));
    bindingSet.Bind(this.emailEdit).For(v => v.text, v => v.onEndEdit).To(vm => vm.Email).TwoWay();
    bindingSet.Bind(this.emailEdit).For(v => v.onValueChanged).To(vm => vm.OnEmailValueChanged(""));
    bindingSet.Bind(this.rememberEdit).For(v => v.isOn, v => v.onValueChanged).To(vm => vm.Remember).TwoWay();
    bindingSet.Bind(this.submit).For(v => v.onClick).To(vm => vm.OnSubmit());
    bindingSet.Build();

    //绑定标题,标题通过本地化文件配置
    BindingSet<DatabindingExample> staticBindingSet = this.CreateBindingSet<DatabindingExample>();
```

```
        staticBindingSet.Bind(this.title).For(v => v.text).To(() => Res.databinding_tutorials_title).OneTime();
        staticBindingSet.Build();
    }
}
```

Lua 示例

在Lua示例中，LuaBehaviour脚本是一个通用的脚本，它是由框架提供的，我们只需要编写绑定到这个脚本上的Lua脚本即可，如下图中的DatabindingExample.lua。在LuaBehaviour中，为确保通用性，所有的成员属性也是通过VariableArray属性表来动态定义的，如下图所示。



在Lua脚本DatabindingExample.lua中，上图所有的动态属性都被注册到Lua环境中，我们可以通过self对象来访问所有的属性，请看下面的代码。

```
require("framework.System")

local Context = CS.Loxodon.Framework.Contexts.Context
local LuaBindingServiceBundle = CS.Loxodon.Framework.Binding.LuaBindingServiceBundle
local ObservableObject = require("framework.ObservableObject")
local ObservableDictionary = require("framework.ObservableDictionary")

---
--创建一个Account子视图模型
--@module AccountViewModel
local AccountViewModel = class("AccountViewModel",ObservableObject)

function AccountViewModel:ctor(t)
    --执行父类ObservableObject的构造函数，这个重要，否则无法监听数据改变
    AccountViewModel.super.ctor(self,t)

    if not (t and type(t)=="table") then
        self.id = 0
        self.username = ""
        self.Password = ""
        self.email = ""
        self.birthday = os.time({year =1970, month = 00, day =00, hour =00, min =00, sec = 00})
        self.address = ""
    end
end

---
--创建一个数据绑定示例的视图模型
--@module DatabindingViewModel
local DatabindingViewModel = class("DatabindingViewModel",ObservableObject)
```

```
function DatabindingViewModel:ctor(t)
    --执行父类ObservableObject的构造函数，这个重要，否则无法监听数据改变
    DatabindingViewModel.super.ctor(self,t)

    if not (t and type(t)=="table") then
        self.account = Account()
        self.remember = false
        self.username = ""
        self.email = ""
        self.errors = ObservableDictionary()
    end

end

function DatabindingViewModel:submit()
    if #self.username < 1 then
        --注意C#字典类型的使用方式，通过set_Item或者get_Item 访问
        self.errors:set_Item("errorMessage","Please enter a valid username.")
        return
    end

    if #self.email < 1 then
        --注意C#字典类型的使用方式，通过set_Item或者get_Item 访问
        self.errors:set_Item("errorMessage","Please enter a valid email.")
        return
    end

    self.errors:Clear()

    self.account.username = self.username
    self.account.email = self.email
    self.account.remember = self.remember
end

---
--创建一个数据绑定视图,扩展DatabindingExample.cs 对象，这里的target是从C#脚本传过来的
--@module DatabindingExample
local M = class("DatabindingExample",target)

function M:awake()
    local context = Context.GetApplicationContext()
    local container = context:GetContainer()

    --初始化Lua的数据绑定服务，一般建议在游戏的C#启动脚本创建
    local bundle = LuaBindingServiceBundle(container)
    bundle:Start();
end

function M:start()
    --初始化Account子视图模型
    local account = AccountViewModel({
            id = 1,
            username = "test",
            password = "test",
            email = "yangpc.china@gmail.com",
            birthday = os.time({year =2000, month = 03, day =03, hour =00, min =00, sec = 00}),
            address = "beijing",
            remember = true
        })

    --初始化视图模型
    self.viewModel = DatabindingViewModel({
            account = account,
            username = "",
            email = "",
            remember = true,
            errors = ObservableDictionary()
```

```
        })

    self:BindingContext().DataContext = self.viewModel

    --绑定UI控件到视图模型
    local bindingSet = self:CreateBindingSet();

    bindingSet:Bind(self.username):For("text"):To("account.username"):OneWay()
    bindingSet:Bind(self.password):For("text"):To("account.password"):OneWay()
    bindingSet:Bind(self.email):For("text"):To("account.email"):OneWay()
    bindingSet:Bind(self.remember):For("text"):To("account.remember"):OneWay()
    bindingSet:Bind(self.birthday):For("text"):ToExpression(function(vm)
            return os.date("%Y-%m-%d",vm.account.birthday)
        end ,"account.birthday"):OneWay()
    bindingSet:Bind(self.address):For("text"):To("account.address"):OneWay()
    bindingSet:Bind(self.errorMessage):For("text"):To("errors['errorMessage']"):OneWay()
    bindingSet:Bind(self.usernameInput):For("text","onEndEdit"):To("username"):TwoWay()
    bindingSet:Bind(self.emailInput):For("text","onEndEdit"):To("email"):TwoWay()
    bindingSet:Bind(self.rememberInput):For("isOn","onValueChanged"):To("remember"):TwoWay()
    bindingSet:Bind(self.submit):For("onClick"):To("submit"):OneWay()

    bindingSet:Build()
end

return M
```

功能介绍

上下文（Context）

在很多框架中，我们应该经常看到上下文这个概念，它可以说就是与当前代码运行相关的一个环境，你能在上下文中提供了当前运行需要的环境数据或者服务等。在这里，我根据游戏开发的特点，我提供了应用上下文（ApplicationContext）、玩家上下文（PlayerContext），同时也支持开发人员根据自己的需求来创建其他的上下文。

在上下文中，我创建了一个服务容器（有关服务容器的介绍请看下一章节）来存储与当前上下文相关的服务，同时创建了个字典来存储数据。通过上下文的Dispose()，可以释放所有在上下文容器中注册的服务。**但是需要注意的是，服务必须继承System.IDisposable接口，否则不能自动释放。**

- **全局/应用上下文（ApplicationContext）**

  应用上下文是一个全局的上下文，它是单例的，它主要存储全局共享的一些数据和服务。所有的基础服务，比如视图定位服务、资源加载服务，网络连接服务、本地化服务、配置文件服务、Json/Xml解析服务、数据绑定服务等等，这些在整个游戏中都可能使用到的基础服务都应该注册到应用上下文的服务容器当中，可以通过应用上下文来获得。

  ```
  //获得全局的应用上下文
  ApplicationContext context = Context.GetApplicationContext();

  //获得上下文中的服务容器
  IServiceContainer container = context.GetContainer();

  //初始化数据绑定服务，这是一组服务，通过ServiceBundle来初始化并注册到服务容器中
  BindingServiceBundle bundle = new BindingServiceBundle(context.GetContainer());
  bundle.Start();

  //初始化IUIViewLocator，并注册到容器
  container.Register<IUIViewLocator>(new ResourcesViewLocator ());

  //初始化本地化服务，并注册到容器中
  CultureInfo cultureInfo = Locale.GetCultureInfo();
  var dataProvider = new ResourcesDataProvider("LocalizationExamples", new XmlDocumentParser());
  Localization.Current = Localization.Create(dataProvider, cultureInfo);
  container.Register<Localization>(Localization.Current);

  //从全局上下文获得IUIViewLocator服务
  ```

```
IUIViewLocator locator = context.GetService<IUIViewLocator>();

//从全局上下文获得本地化服务
Localization localization = context.GetService<Localization>();
```

- **玩家上下文（PlayerContext）**

  玩家上下文是只跟当前登录的游戏玩家相关的上下文，比如一个游戏玩家Clark登录游戏后，他在游戏中的基本信息和与之相关的服务，都应该存储在玩家上下文中。比如背包服务，它负责拉取和同步玩家的背包数据，缓存了玩家背包中的武器、装备、道具等等，它只与当前玩家有关，当玩家退出登录切换账号时，这些数据都应该被清理和释放。我们使用了玩家上下文来存储这些服务和数值时，只需要调用PlayerContext.Dispose()函数，就可以释放与当前玩家有关的所有数据和服务。

  玩家上下文中默认继承了全局上下文的所有服务和属性，所以通过玩家上下文可以获取到所有在全局上下文中的服务和数据，当玩家上下文注册了与全局上下文中Key值相同的服务或者是属性时，它会在玩家上下文中存储，不会覆盖全局上下文中存储的数据，当通过Key访问时，优先返回玩家上下文中的数据，只有在玩家上下文中找不到时才会去全局上下文中查找。

```
//为玩家clark创建一个玩家上下文
PlayerContext playerContext = new PlayerContext("clark");

//获得玩家上下文中的服务容器
IServiceContainer container = playerContext.GetContainer();

//将角色信息存入玩家上下文
playerContext.Set("roleInfo", roleInfo);

//初始化背包服务，注册到玩家上下文的服务容器中
container.Register<IKnapsackService>(new KnapsackService());

//从通过玩家上下文获得在全局上下文注册的IViewLocator服务
IUIViewLocator locator = playerContext.GetService<IUIViewLocator>();

//从通过玩家上下文获得在全局上下文注册的本地化服务
Localization localization = playerContext.GetService<Localization>();

//当用户clark退出登录时，注销玩家上下文，自动注销所有注册在当前玩家上下文中的服务。
playerContext.Dispose();
```

- **其它上下文（Context）** 一般来说，在很多游戏开发中，我们只需要全局上下文和玩家上下文就足以满足要求，但是在某些情况下，我们还需要一个上下文来存储环境数据，比如在MMO游戏中，进入某个特定玩法的副本，那么我就需要为这个副本创建一个专属的上下文，当副本中的战斗结束，退出副本时，则销毁这个副本上下文来释放资源。

```
//创建一个上下文，参数container值为null，在Context内部会自动创建
//参数contextBase值为playerContext，自动继承了playerContext中的服务和属性
Context context = new Context(null,playerContext);

//获得上下文中的服务容器
IServiceContainer container = context.GetContainer();

//注册一个战斗服务到容器中
container.Register<IBattleService>(new BattleService());
```

服务容器

在项目开始时，我曾调研过很多C#的控制反转和依赖注入（IoC/DI）方面的开源项目，开始是想用Zenject来做为服务的容器使用，后来因为考虑到移动项目中，内存和CPU资源都相当宝贵，不想再引入一个这么大的库来消耗内存，也不想因为反射导致的性能损失，而且强制用户使用IoC/DI也不太合适，毕竟不是所有人都喜欢，所以我就自己设计了一个简单的服务容器，来满足服务注册、注销、读取这些最基本的功能。

注意：所有注册的服务，只有继承System.IDisposable接口，实现了Dispose函数，才能在IServiceContainer.Dispose()时自动释放。

- **服务注册器(IServiceRegistry)**

  服务注册负责注册和注销服务，它可以根据服务类型或者名称注册一个服务实例到容器中，也可以注册一个服务工厂到容器中，用户可以根据自己的需求来选择是否需要注册一个服务工厂，是创建一个单态的服务，还是每次都创建一个新的服务实例。

```
IServiceContainer container = ...
IBinder binder = ...
IPathParser pathParser = ...

//注册一个类型为IBinder的服务到容器中,可以通过container.Resolve<IBinder>() 或者
//container.Resolve("IBinder") 来访问这个服务，在容器中默认使用了IBinder.Name做为Key存储。
container.Register<IBinder>(binder);

//注册一个名为parser的IPathParser到容器中
//只能通过container.Resolve("parser")来访问这个服务
container.Register("parser",pathParser);
```

- **服务定位器(IServiceLocator)**

  通过服务定位器可以获得服务，服务定位器可以根据服务名称或者类型来查询服务，当服务以类型的方式注册，则可以通过类型或者类型名来查找服务，当服务以特定的名称为Key注册，则只能通过服务名来查找服务。

```
IServiceContainer container = ...

//IBinder服务在上段代码中，以类型方式注册，所以可以通过类型或者名称方式查询服务
IBinder binder = container.Resolve<IBinder>(); //or container.Resolve("IBinder")

//IPathParser在上段代码中以特定名称"parser"注册，则只能通过名称"parser"来查询服务
IPathParser pathParser = container.Resolve("parser");
```

- **服务Bundle(IServiceBundle)**

  ServiceBundle的作用是将一组相关的服务打包注册和注销，比如我的数据绑定服务，就是通过ServiceBundle.Start()方法一次性注册所有数据绑定有关的服务，当服务不在需要时，又可以通过ServiceBundle.Stop()方法来注销整个模块的所有服务（见下面的代码）。这在某些时候非常有用，比如启动和停止一个模块的所有服务。

```
//初始化数据绑定模块，启动数据绑定服务,注册服务
BindingServiceBundle bundle = new BindingServiceBundle(context.GetContainer());
bundle.Start();

//停止数据绑定模块，注销所有数据绑定相关的服务
bundle.Stop();
```

应用配置（Preference）

Perference可以说就是Unity3d的PlayerPrefs，只是我对PlayerPrefs的功能进行了扩展、补充和标准化。Perference除了可以存储boolean、int、float、string等基本数据类型之外，还可以存储DateTime、Vector2、Vector3、Vector4、Color、Version，以及任何JsonUtility可以序列化的对象类型，甚至你可以自己自定义类型编码解码器（ITypeEncoder）来扩展任何你想存储的类型。Perference支持加密的方式存储数据，并且我实现了两种持久化的方式，第一种是将数据转换为string的方式存储在Unity3D的PlayerPrefs中。第二种是以二进制的方式存储在文件中，一般在项目测试时我都使用文件持久化的方式，因为我可以直接删除Application.persistentDataPath目录下的文件方便的删除配置。

Perference除了扩展以上功能外，我还扩展了配置的作用域，如同前文中的Context一样，同样包括全局的配置和玩家的配置，也同样支持某个局部模块的配置。全局配置可以用来存放当前资源更新的版本，最后登录的用户名等与应用相关的信息；玩家配置可以存在多个（如果在一台机器上有多个账户登录的话），可以存放具体某个玩家在本机的配置信息，如玩家在游戏中背景音乐、音效、画面质量、视距远近的设置等等。

下面跟随我的代码，我们来了解它是如何使用的。

```
//注册一个Preference的工厂，默认是PlayerPrefsPreferencesFactory工厂，只有使用File持久化才需要改为BinaryFilePreferencesFactory工厂
Preferences.Register(new BinaryFilePreferencesFactory());

//获得全局配置，如果不存在则自动创建
Preferences globalPreferences = Preferences.GetGlobalPreferences();

//存储当前资源更新后的数据版本
globalPreferences.SetObject<Version>("DATA_VERSION",dataVersion);

//存储游戏最后成功登录的用户名，下次启动游戏时自动填写在账号输入框中
globalPreferences.SetString("username","clark");
```

```
//数据修改后调用Save函数保存数据
globalPreferences.Save();

//根据key值"clark@zone5"获得配置，如果不存在则自动创建，这里的意思是获得游戏第5区名为clark的用户的配置信息
//在Preferences.GetPreferences()函数中，name只是一个存取的Key，你可以完全按自己的意思组合使用。
Preferences userPreferences Preferences.GetPreferences("clark@zone5");

//设置游戏音乐、音效开关，并保存
userPreferences.SetBool("Music_Enable",true);
userPreferences.SetBool("Sound_Enable",true);
userPreferences.Save();
```

在Preferences中，我虽然已支持了很多种的数据类型，但是总有些特殊需求我是无法满足的，那么你通过ITypeEncoder来扩展自己的类型；并且如果你对配置数据的安全性是有要求的，那么你也可以使用自己的密码来加密数据。

```
/// <summary>
/// 自定义一个类型编码器
/// </summary>
public class ColorTypeEncoder : ITypeEncoder
{
    private int priority = 900; //当一个类型被多个类型编码器支持时，优先级最高的有效(优先级在-999到999之间)

    public int Priority
    {
        get { return this.priority; }
        set { this.priority = value; }
    }

    public bool IsSupport(Type type)
    {
        if (type.Equals(typeof(Color)))
            return true;
        return false;
    }

    //将string类型转回对象类型
    public object Decode(Type type, string value)
    {
        if (string.IsNullOrEmpty(value))
            return null;

        Color color;
        if(ColorUtility.TryParseHtmlString(value,out color))
            return color;

        return null;
    }

    //将对象转换为string来保存，因为PlayerPrefs只支持string类型的数据
    public string Encode(object value)
    {
        return ColorUtility.ToHtmlStringRGBA((Color)value);
    }
}


//默认使用AES128_CBC_PKCS7加密，当然你也可以自己实现IEncryptor接口，定义自己的加密算法。
byte[] iv = Encoding.ASCII.GetBytes("5CyM5tcL3yDFiWlN");
byte[] key = Encoding.ASCII.GetBytes("W8fnmqMynlTJXPM1");

IEncryptor encryptor = new DefaultEncryptor(key, iv);

//序列化和反序列化类
ISerializer serializer = new DefaultSerializer();

//添加自定义的类型编码器
```

```
serializer.AddTypeEncoder(new ColorTypeEncoder());

//注册Preferences工厂
BinaryFilePreferencesFactory factory = new BinaryFilePreferencesFactory(serializer, encryptor);
Preferences.Register(factory);
```

更多的示例请查看教程 Basic Tutorials

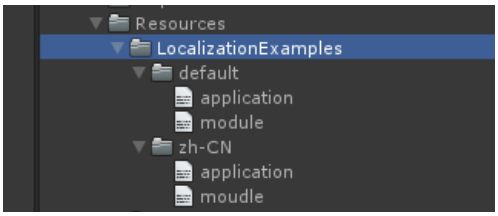
国际化和本地化

国际化和本地化是指软件、应用、游戏等使之能适应目标市场的语言、地区差异以及技术需要等。所以在游戏开发中，为适用不同的市场需求，本地化是必不可少的功能，我参考了Android的本地化设计思路，设计了本框架的本地化模块。本地化模块和前面提到的任何模块一样，它也是可以自定义的，可以自由扩展的，下面我就来介绍一下如何来使用本地化模块。

- **目录结构**

  本地化文件可以放在Resources目录下，通过Unity3D的Resources来访问，也可以放入AssetBundle中，通过AssetBundle来加载，甚至你可以放入任何其他地方，通过自定义的IDataProvider来读取。并且这些方式可以同时存在，后加载的覆盖先加载的。在本框架中，我提供了DefaultDataProvider和AssetBundleDataProvider两个数据提供器分别来加载Resources中和AssetBundle中的本地化数据文件。无论在Resources中还是在AssetBundle，其目录结构和加载规则是一致的。首先必须有一个本地化配置文件的根目录，如下图的LocalizationExamples目录，在根目录下创建各个语言的目录，比如 default、zh、zh-CN、zh-TW、zh-HK、en、en-US、en-CA、en-AU等等（具体可以参考System.Globalization.CultureInfo类的Name和TwoLetterISOLanguageName，如zh-CN是Name，zh是TwoLetterISOLanguageName）。在default目录中的配置必须是最完整的，它是默认语言配置，而且是必须的，而其他目录都是可选的。zh目录是中文目录，zh-CN是中国大陆的配置目录，zh-TW是台湾区的配置目录，zh-HK是中国香港的配置目录。从配置文件的优先级来说（zh-CN|zh-TW|zh-HK) > zh > default，优先级高的配置将覆盖优先级低的配置。

  在每一个配置文件目录中，配置文件建议按业务模块分多个文件配置，不要所有的配置都写入一个文本文件中，如下图所示，所有全局的配置写入application.xml中，而其他的配置则按模块名称来命名配置文件。

- **配置文件的格式**

  配置文件默认只支持XML格式，如有必要也可以通过自定义IDocumentParser来支持其他的格式，如Json格式，二进制格式，或者从SQLite中加载等。

  default版本的application和module如下:

```
<!-- application.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app.name">Loxodon Framework Examples</string>
    <string name="framework.name">LoxodonFramework</string>
    <vector3 name="user.position">(20 , 20.2 , 30)</vector3>
    <color name="color.black">#000000</color>
    <color-array name="button.transition.colors">
        <item>#FFFFFFFF</item>
        <item>#F5F5F5FF</item>
        <item>#C8C8C8FF</item>
        <item>#C8C8C880</item>
    </color-array>
    <datetime name="created">2016-10-27T00:00:00.000</datetime>
</resources>

<!-- module.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="startup.progressbar.tip.loading">Loading...</string>
    <string name="startup.progressbar.tip.unziping">Unziping...</string>
```

```
    <string name="login.failure.tip">Login failure.</string>
    <string name="login.exception.tip">Login exception.</string>
    <string name="login.validation.username.error">Please enter a valid username.</string>
    <string name="login.validation.password.error">Please enter a valid password.</string>
    <string name="login.label.title.text">Sign in</string>
    <string name="login.button.confirm.text">Confirm</string>
    <string name="login.button.cancel.text">Cancel</string>
    <string name="login.label.username.text">Username:</string>
    <string name="login.label.password.text">Password:</string>
    <string name="login.input.username.prompt">Enter username...</string>
    <string name="login.input.password.prompt">Enter password...</string>
</resources>
```

zh-CN版本的application和module如下:

```
<!-- application.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app.name">Loxodon Framework 示例</string>
    <string name="framework.name">LoxodonFramework</string>
    <vector3 name="user.position">(20 , 20.2 , 30)</vector3>
    <color name="color.black">#000000</color>
    <color-array name="button.transition.colors">
        <item>#FFFFFFFF</item>
        <item>#F5F5F5FF</item>
        <item>#C8C8C8FF</item>
        <item>#C8C8C880</item>
    </color-array>
    <datetime name="created">2016-10-27T00:00:00.000</datetime>
</resources>

<!-- module.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="startup.progressbar.tip.loading">加载中...</string>
    <string name="startup.progressbar.tip.unziping">解压中...</string>
    <string name="login.failure.tip">登录失败</string>
    <string name="login.exception.tip">登录异常</string>
    <string name="login.validation.username.error">输入的用户名格式错误</string>
    <string name="login.validation.password.error">输入的密码格式错误</string>
    <string name="login.label.title.text">登录</string>
    <string name="login.button.confirm.text">确认</string>
    <string name="login.button.cancel.text">取消</string>
    <string name="login.label.username.text">用户名:</string>
    <string name="login.label.password.text">密  码:</string>
    <string name="login.input.username.prompt">请输入用户名...</string>
    <string name="login.input.password.prompt">请输入密码...</string>
</resources>
```

- **支持的数值类型**

  默认支持以下所有类型和他们的数组类型，通过自定义类型转换器ITypeConverter，可以支持新的数据类型。

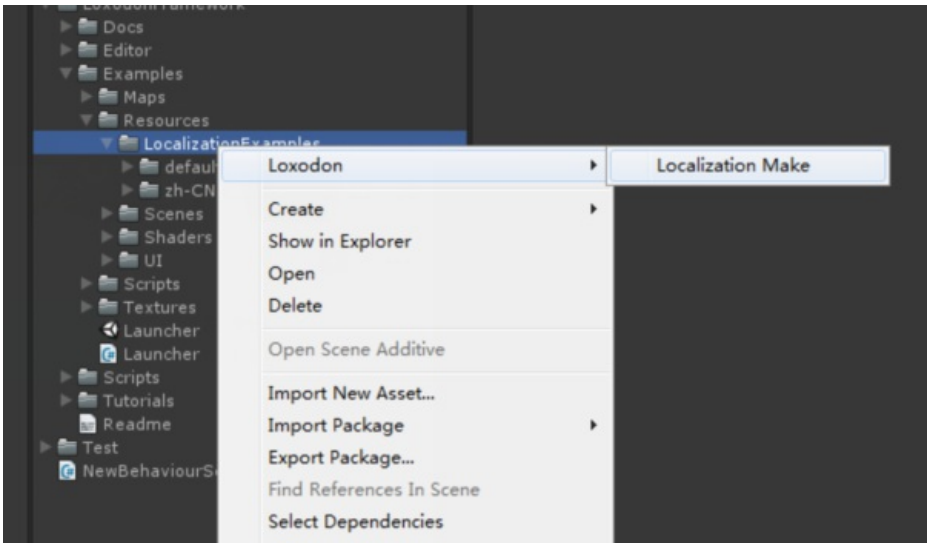
```
string
boolean
sbyte
byte
short
ushort
int
uint
long
ulong
char
float
double
```

```
decimal
datetime
vector2
vector3
vector4
color
```

- **生成C#脚本**

本地化配置的属性，类似Android配置一样，可以生成一个静态类来使用，如果是使用C#版本的MVVM，可以这么使用，这样增加了语言的编译校验机制，避免出错。如果是使用Lua编程，则不建议这么做，直接使用Localization类即可。

在本地化配置的根目录右击，弹出代码生成菜单如下图，点击Localization Make，选择代码目录和文件名，生成C#静态类。



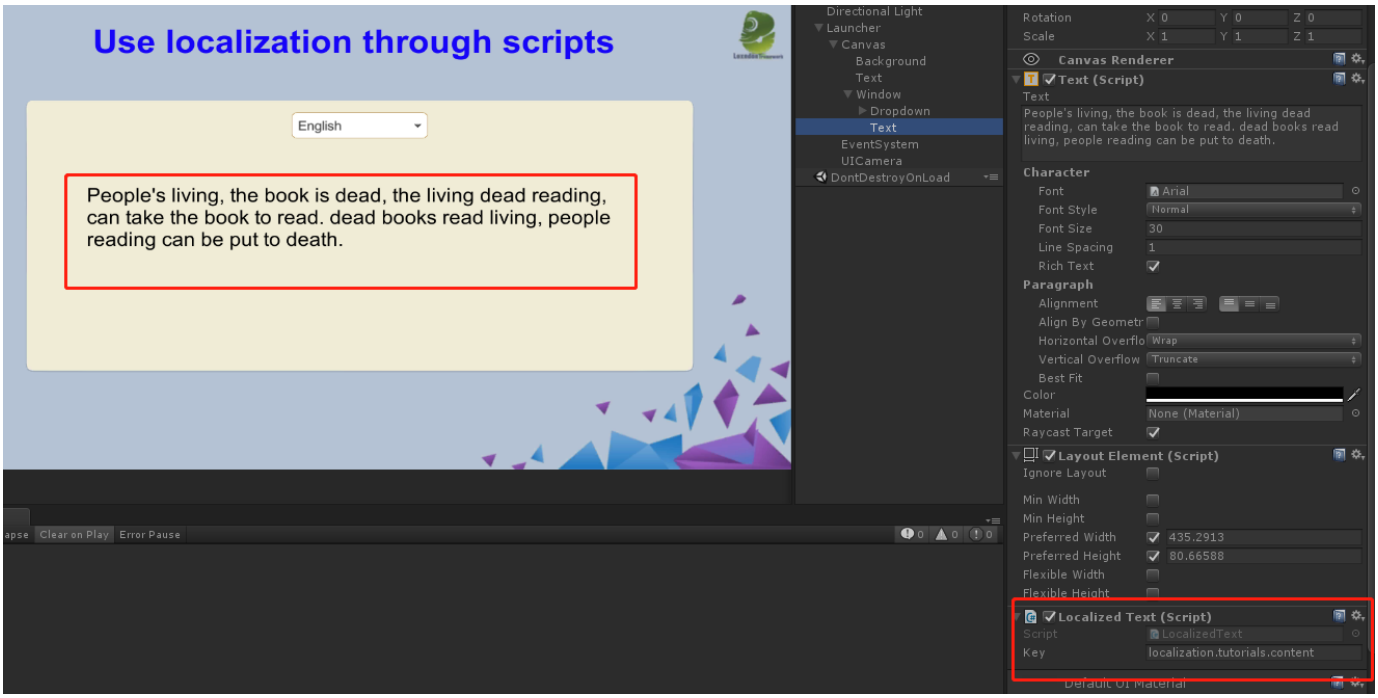

```
public static partial class R
{
    public readonly static V<string> startup_progressbar_tip_loading = new V<string>("startup.progressbar.tip.loading");

    public readonly static V<string> startup_progressbar_tip_unziping = new V<string>("startup.progressbar.tip.unziping");

    public readonly static V<string> login_failure_tip = new V<string>("login.failure.tip");

    public readonly static V<string> login_exception_tip = new V<string>("login.exception.tip");
}
```

- **使用示例**

通过生成的C#代码调用或者通过Localization类调用。

```
Localization localization = Localization.Current

//通过Localization的成员方法调用
string errorMessage = localization.GetText("login.validation.username.error", "Please enter a valid username.");

//通过生成的静态代码调用（比如提前生成C#代码）
string loadingMessage = R.startup_progressbar_tip_loading;
```

配合UI组件使用本地化配置，下面我们模拟一个游戏中语言切换的使用场景，来了解本地化模块的用法。在下图中，红色线框中的英文通过本地化服务来加载和修改，它是通过挂在Text对象上的LocalizedText组件来实现中文和英文切换的。




```
public class LocalizationExample : MonoBehaviour
{
    public Dropdown dropdown;

    private Localization localization;

    void Awake ()
    {
        CultureInfo cultureInfo = Locale.GetCultureInfoByLanguage (SystemLanguage.English);

        //创建一个数据提供器，从LocalizationTutorials目录中加载本地化文件
        var dataProvider = new DefaultDataProvider ("LocalizationTutorials", new XmlDocumentParser ());

        //创建一个本地化服务
        Localization.Current = Localization.Create (dataProvider, cultureInfo);
        this.localization = Localization.Current;

        //监听下拉列表的改变，在英文和中文间切换
        this.dropdown.onValueChanged.AddListener (OnValueChanged);
    }

    void OnValueChanged (int value)
    {
        switch (value) {
        case 0:
            //设置本地化服务当前语言为英文
            this.localization.CultureInfo = Locale.GetCultureInfoByLanguage (SystemLanguage.English);
            break;
        case 1:
            //设置本地化服务当前语言为中文
            this.localization.CultureInfo = Locale.GetCultureInfoByLanguage (SystemLanguage.ChineseSimplified);
            break;
        default:
            //设置本地化服务当前语言为英文
            this.localization.CultureInfo = Locale.GetCultureInfoByLanguage (SystemLanguage.English);
            break;
        }
    }

    void OnDestroy ()
    {
        this.dropdown.onValueChanged.RemoveListener (OnValueChanged);
```

```
    }
}
```

本地化文件配置如下

```
<!-- 英文版 -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app.name">LoxodonFramework</string>
    <string name="databinding.tutorials.title">Databinding Examples</string>
    <string name="localization.tutorials.content">People's living, the book is dead,
        the living dead reading, can take the book to read. dead books read living,
        people reading can be put to death.</string>
</resources>

<!-- 中文版 -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app.name">LoxodonFramework</string>
    <string name="databinding.tutorials.title">数据绑定示例</string>
    <string name="localization.tutorials.content">人是活的，书是死的，活人读死书，可以把书读活。
    死书读活人，可以把人读死。</string>
</resources>
```

更多的示例请查看教程 Localization Tutorials

线程/协程异步结果和异步任务

为了方便协程和线程的异步调用，我根据Future/Promise的设计模式，设计一组异步结果、异步任务，在使用时我们可以通过同步的方式来获得任务的执行结果，也可以通过回调的方式来获得任务的结果，跟随下面的示例，我们来了解异步结果的使用。

- **AsyncResult**

  利用AsyncResult，我们来创建一个可以取消的协程任务，并分别通过同步阻塞的方式和回调的方式来获得执行结果。

```
public class AsyncResultExample : MonoBehaviour
{

    protected IEnumerator Start ()
    {
        //********启动任务，同步方式调用示例***********//
        IAsyncResult<bool> result = StartTask();

        //等待任务完成，result.WaitForDone ()函数返回一个迭代器IEnumerator
        yield return result.WaitForDone ();

        if(r.Exception !=null)
        {
            Debug.LogFormat("任务执行失败：{0}",r.Exception);
        }
        else
        {
            Debug.LogFormat("任务执行成功 result = {0}",r.Result);
        }


        //********启动任务，回调方式调用示例***********//
        result = StartTask();
        result.Callbackable().OnCallback((r) =>
        {
            if(r.Exception !=null)
            {
                Debug.LogFormat("任务执行失败：{0}",r.Exception);
            }
            else
            {
```

```
                Debug.LogFormat("任务执行成功 result = {0}",r.Result);
            }
        });

    }

    //创建一个任务
    public IAsyncResult<bool> StartTask()
    {
        //创建一个异步结果，参数cancelable = true，支持取消操作
        AsyncResult<bool> result = new AsyncResult<bool> (true);

        //启动任务
        this.StartCoroutine (DoTask (result));

        return result;
    }

    /// <summary>
    /// 模拟一个任务
    /// </summary>
    /// <returns>The task.</returns>
    /// <param name="promise">Promise.</param>
    protected IEnumerator DoTask (IPromise<bool> promise)
    {
        for (int i = 0; i < 20; i++) {
            //如果外部调用了AsyncResult.Cancel()函数，则这里的IsCancellationRequested = true，请求取消任务
            if (promise.IsCancellationRequested) {
                promise.SetCancelled ();
                yield break;
            }
            yield return new WaitForSeconds (0.5f);
        }

        //执行完成必须设置结果
        promise.SetResult (true);
    }
}
```

- **ProgressResult**

  ProgressResult与AsyncResult功能类似，只是增加了任务进度，下面我来看示例。

```
/// <summary>
/// 任务进度
/// </summary>
public class Progress
{
    public int bytes;
    public int TotalBytes;

    public int Percentage { get { return (bytes * 100) / TotalBytes; } }
}

public class ProgressResultExample : MonoBehaviour
{
    protected void Start()
    {
        //开始一个任务
        IProgressResult<Progress, string> result = StartTask();

        //打印任务进度
        result.Callbackable().OnProgressCallback(progress =>
        {
            Debug.LogFormat("Percentage: {0}% ", progress.Percentage);
        });
```

```
        //监听任务结果
        result.Callbackable().OnCallback(r =>
        {
            Debug.LogFormat("IsDone:{0} Result:{1}", r.IsDone, r.Result);
        });
    }

    public IProgressResult<Progress, string> StartTask()
    {
        ProgressResult<Progress, string> result = new ProgressResult<Progress, string>(true);

        this.StartCoroutine(DoTask(result));

        return result;
    }

    /// <summary>
    /// 模拟一个有进度的任务
    /// </summary>
    /// <returns>The task.</returns>
    /// <param name="promise">Promise.</param>
    protected IEnumerator DoTask(IProgressPromise<Progress, string> promise)
    {
        int n = 50;
        Progress progress = new Progress();
        progress.TotalBytes = n;
        progress.bytes = 0;
        StringBuilder buf = new StringBuilder();
        for (int i = 0; i < n; i++)
        {
            /* If the task is cancelled, then stop the task */
            if (promise.IsCancellationRequested)
            {
                promise.SetCancelled();
                yield break;
            }

            progress.bytes += 1;
            buf.Append(" ").Append(i);
            promise.UpdateProgress(progress);/* update the progress of task. */
            yield return new WaitForSeconds(0.01f);
        }

        //执行完成必须设置结果
        promise.SetResult(buf.ToString());
    }

}
```

- **AsyncTask**

异步任务是对一个线程任务或者一个协程任务的封装，将一个迭代器IEnumerator传入AsyncTask可以创建一个协程任务，或者将一个委托函数传入，可以创建一个后台线程执行的任务。根据任务执行过程，将一个任务拆分成执行前、执行成功后/执行失败后、执行结束几个阶段，在每一个阶段中都可以通过一个委托回调来注册自己的代码块。下面的示例中，我们来看看怎么创建一个协程任务。

```
public class AsyncTaskExample : MonoBehaviour
{
    protected IEnumerator Start()
    {
        AsyncTask task = new AsyncTask(DoTask(), true);

        /* 开始任务 */
        task.OnPreExecute(() =>
        {
            //任务执行前调用
```

```
            Debug.Log("The task has started.");
        }).OnPostExecute(() =>
        {
            //任务成功执行后调用
            Debug.Log("The task has completed.");/* only execute successfully */
        }).OnError((e) =>
        {
            //任务执行失败调用
            Debug.LogFormat("An error occurred:{0}", e);
        }).OnFinish(() =>
        {
            //任务执行完毕，无论成功失败，都会调用
            Debug.Log("The task has been finished.");/* completed or error or canceled*/
        }).Start();

        //等待任务结束
        yield return task.WaitForDone();

        Debug.LogFormat("IsDone:{0} IsCanceled:{1} Exception:{2}", task.IsDone, task.IsCancelled, task.Exception);
    }

    /// <summary>
    /// 模拟一个任务的执行
    /// </summary>
    /// <returns>The task.</returns>
    /// <param name="promise">Promise.</param>
    protected IEnumerator DoTask()
    {
        int n = 10;
        for (int i = 0; i < n; i++)
        {
            yield return new WaitForSeconds(0.5f);
        }
    }
}
```

- **ProgressTask**

ProgressTask与AsyncTask功能类似，只是增加了任务进度，同样ProgressTask既可以创建一个协程任务，也可以创建一个后台线程的任务。

```
public class ProgressTaskExample : MonoBehaviour
{
    protected IEnumerator Start()
    {
        //创建一个任务，这个任务将在一个后台线程中执行
        ProgressTask<float, string> task = new ProgressTask<float, string>(
                new Action<IProgressPromise<float, string>>(DoTask), false, true);

        /* 开始一个任务 */
        task.OnPreExecute(() =>
        {
            //在任务执行前调用
            Debug.Log("The task has started.");
        }).OnPostExecute((result) =>
        {
            //在任务成功执行后调用
            Debug.LogFormat("The task has completed. result:{0}", result);/* only execute successfully */
        }).OnProgressUpdate((progress) =>
        {
            //任务执行的进度
            Debug.LogFormat("The current progress:{0}%", (int)(progress * 100));
        }).OnError((e) =>
        {
            //在任务执行失败后调用
            Debug.LogFormat("An error occurred:{0}", e);
        }).OnFinish(() =>
        {
```

```
        `
            //任务执行完毕，无论成功失败，都会调用
            Debug.Log("The task has been finished.");/* completed or error or canceled*/
        }).Start();

        yield return task.WaitForDone();

        Debug.LogFormat("IsDone:{0} IsCanceled:{1} Exception:{2}", task.IsDone, task.IsCancelled, task.Exception);
    }

    /// <summary>
    /// 模拟一个任务，这不是一个迭代器，这将会在一个后台线程中执行
    /// </summary>
    /// <returns>The task.</returns>
    /// <param name="promise">Promise.</param>
    protected void DoTask(IProgressPromise<float, string> promise)
    {
        try
        {
            int n = 50;
            float progress = 0f;
            StringBuilder buf = new StringBuilder();
            for (int i = 0; i < n; i++)
            {
                /* If the task is cancelled, then stop the task */
                if (promise.IsCancellationRequested)
                {
                    promise.SetCancelled();
                    break;
                }

                progress = i / (float)n;
                buf.Append(" ").Append(i);
                promise.UpdateProgress(progress);/* update the progress of task. */
                Thread.Sleep(200);
            }
            promise.UpdateProgress(1f);
            promise.SetResult(buf.ToString()); /* update the result. */
        }
        catch (System.Exception e)
        {
            promise.SetException(e);
        }
    }
}
```

更多的示例请查看教程 Basic Tutorials

线程/协程执行器

在Unity3d逻辑脚本的开发中，是不支持多线程的，所有的UnityEngine.Object对象，都只能在主线程中访问和修改，但是在游戏开发过程中，我们很难避免会使用到多线程编程，比如通过Socket连接从网络上接受数据，通过多线程下载资源，一些纯计CPU计算的逻辑切入到后台线程去运算等等。这里我就面临一个线程切换的问题。所以在我的框架中，我设计了一个执行器配合前文中的任务结果来使用，它能够很方便的将任务切换到主线程执行，也能很方便的开启一个线程任务。

- **执行器(Executors)**

```
public class ExecutorExample : MonoBehaviour
{

    IEnumerator Start()
    {
        //在后台线程中异步运行一个任务
        Executors.RunAsync(() =>
        {
            Debug.LogFormat("RunAsync ");
        });
```

```
        //在后台线程中异步运行一个任务
        Executors.RunAsync(() =>
        {
            //睡眠1000毫秒
            Thread.Sleep(1000);

            //从后台线程切换到主线程中，
            //waitForExecution = true，当前函数直到主线程执行完后才返回
            Executors.RunOnMainThread(() =>
            {
                Debug.LogFormat("RunOnMainThread Time:{0} frame:{1}", Time.time, Time.frameCount);
            }, true);
        });

        //运行一个协程任务
        IAsyncResult result = Executors.RunOnCoroutine(DoRun());

        //等待任务完成
        yield return result.WaitForDone();
    }

    IEnumerator DoRun()
    {
        for (int i = 0; i < 10; i++)
        {
            Debug.LogFormat("i = {0}", i);
            yield return null;
        }
    }
}
```

- **定时任务执行器(IScheduledExecutor)**

  在本框架中提供了一个线程的定时任务执行器(ThreadScheduledExecutor)和一个Unity3D协程的定时任务执行器(CoroutineScheduledExecutor)，下面我们以线程的定时任务执行器为例，来介绍它的用法。

```
//创建并启动一个线程的定时任务执行器
var scheduled = new ThreadScheduledExecutor();
scheduled.Start();

//延时1000毫秒后执行，以固定频率，每隔2000毫秒，打印一句“This is a test.”
IAsyncResult result = scheduled.ScheduleAtFixedRate(() =>
{
    Debug.Log("This is a test.");
}, 1000, 2000);
```

- **可拦截的迭代器(InterceptableEnumerator)**

  在Unity3D的协程中，如果发生异常，是无法捕获到异常的，所有很多时候无法知道一个协程是否正常执行结束，出现错误也不方便查找原因，根据Unity3D协程其本质是一个迭代器的原理，我设计了一个可以在协程执行过程中注入代码块，捕获异常的可拦截迭代器。使用InterceptableEnumerator对原迭代器进行包装，就可以捕获到协程代码执行异常，并且无论协程是否正常结束，都可在协程退出前插入一个代码块。在我的Executors中，我就是利用InterceptableEnumerator来确保任务正常结束的，无论协程执行成功或者异常我都能通过注册的Finally语句块来设置AsyncResult的结果。

  InterceptableEnumerator支持条件语句块，可以在外部插入一个条件语句块，控制协程逻辑或中止协程。异常语句块，可以捕获到协程异常，Finally语句块，确保协程结束一定会调用这个语句块。下面我们来看看示例。

```
/// <summary>
/// 这是一个迭代器的包装函数
/// </summary>
protected static InterceptableEnumerator WrapEnumerator(IEnumerator routine, IPromise promise)
{
    InterceptableEnumerator enumerator;
    if(routine is InterceptableEnumerator)
```

```
        enumerator = (InterceptableEnumerator)routine;
    else
        enumerator = new InterceptableEnumerator(routine);

    //注册一个条件语句块，如果任务取消，IsCancellationRequested = true，则结束任务
    enumerator.RegisterConditionBlock(() => !(promise.IsCancellationRequested));

    //注册一个异常捕获语句块，如果协程执行错误，则将异常赋值到任务结果，并打印错误
    enumerator.RegisterCatchBlock(e =>
    {
        if (promise != null)
            promise.SetException(e);

        if (log.IsErrorEnabled)
            log.Error(e);
    });

    //注册一个Finally语句块，确保任务能够正常结束退出
    enumerator.RegisterFinallyBlock(() =>
    {
        if (promise != null && !promise.IsDone)
        {
            if (promise.GetType().IsSubclassOfGenericTypeDefinition(typeof(IPromise<>)))
                promise.SetException(new Exception("No value given the Result"));
            else
                promise.SetResult();
        }
    });
    return enumerator;
}
```

更多的示例请查看教程 Basic Tutorials

消息系统(Messenger)

Messenger用于应用模块间的通讯，它提供了消息订阅和发布的功能。Messenger支持按消息类型订阅和发布消息，也支持按channel来订阅和发布消息。

```
public class MessengerExample : MonoBehaviour
{
    private IDisposable subscription;
    private IDisposable chatroomSubscription;
    private void Start()
    {
        //获得默认的Messenger
        Messenger messenger = Messenger.Default;

        //订阅一个消息,确保subscription是成员变量，否则subscription被GC回收时会自动退订消息
        subscription = messenger.Subscribe((PropertyChangedMessage<string> message) =>
        {
            Debug.LogFormat("Received Message:{0}", message);
        });

        //发布一个属性名改变的消息
        messenger.Publish(new PropertyChangedMessage<string>("clark", "tom", "Name"));

        //订阅聊天频道"chatroom1"的消息
        chatroomSubscription = messenger.Subscribe("chatroom1", (string message) =>
         {
             Debug.LogFormat("Received Message:{0}", message);
         });

        //向聊天频道"chatroom1"发布一条消息
        messenger.Publish("chatroom1", "hello!");
    }
```

```
    private void OnDestroy()
    {
        if (this.subscription != null)
        {
            //退订消息
            this.subscription.Dispose();
            this.subscription = null;
        }

        if (this.chatroomSubscription != null)
        {
            //退订消息
            this.chatroomSubscription.Dispose();
            this.chatroomSubscription = null;
        }
    }
}
```

更多的示例请查看教程 Basic Tutorials

可观察的对象(Observables)

ObservableObject、ObservableList、ObservableDictionary，在MVVM框架的数据绑定中是必不可少的，它们分别实现了INotifyPropertyChanged和INotifyCollectionChanged接口，当对象的属性改变或者集合中Item变化时，我们能通过监听PropertyChanged和CollectionChanged事件可以收到属性改变和集合改变的通知，在数据绑定功能中，只有实现了这两个接口的对象在属性或者集合变化时，会自动通知UI视图改变，否则只能在初始绑定时给UI控件赋值一次，绑定之后改变视图模型的数值，无法通知UI控件修改。

下面我们看看ObservableDictionary的使用示例，当我们需要创建一个自定义的ListView控件时，我们需要了解其原理。

```
public class ObservableDictionaryExample : MonoBehaviour
{
    private ObservableDictionary<int, Item> dict;

    protected void Start()
    {
#if UNITY_IOS
        //在IOS中，泛型类型的字典，需要提供IEqualityComparer<TKey>，否则可能JIT异常
        this.dict = new ObservableDictionary<int, Item>(new IntEqualityComparer());
#else
        this.dict = new ObservableDictionary<int, Item>();
#endif
        dict.CollectionChanged += OnCollectionChanged;

        //添加Item
        dict.Add(1, new Item() { Title = "title1", IconPath = "xxx/xxx/icon1.png", Content = "this is a test." });
        dict.Add(2, new Item() { Title = "title2", IconPath = "xxx/xxx/icon2.png", Content = "this is a test." });

        //删除Item
        dict.Remove(1);

        //清除字典
        dict.Clear();
    }

    protected void OnDestroy()
    {
        if (this.dict != null)
        {
            this.dict.CollectionChanged -= OnCollectionChanged;
            this.dict = null;
        }
    }

    //集合改变事件
    protected void OnCollectionChanged(object sender, NotifyCollectionChangedEventArgs eventArgs)
    {
```

```
        switch (eventArgs.Action)
        {
            case NotifyCollectionChangedAction.Add:
                foreach (KeyValuePair<int, Item> kv in eventArgs.NewItems)
                {
                    Debug.LogFormat("ADD key:{0} item:{1}", kv.Key, kv.Value);
                }
                break;
            case NotifyCollectionChangedAction.Remove:
                foreach (KeyValuePair<int, Item> kv in eventArgs.OldItems)
                {
                    Debug.LogFormat("REMOVE key:{0} item:{1}", kv.Key, kv.Value);
                }
                break;
            case NotifyCollectionChangedAction.Replace:
                foreach (KeyValuePair<int, Item> kv in eventArgs.OldItems)
                {
                    Debug.LogFormat("REPLACE before key:{0} item:{1}", kv.Key, kv.Value);
                }
                foreach (KeyValuePair<int, Item> kv in eventArgs.NewItems)
                {
                    Debug.LogFormat("REPLACE after key:{0} item:{1}", kv.Key, kv.Value);
                }
                break;
            case NotifyCollectionChangedAction.Reset:
                Debug.LogFormat("RESET");
                break;
            case NotifyCollectionChangedAction.Move:
                break;
        }
    }
}
```

更多的示例请查看教程 Basic Tutorials

数据绑定(Databinding)

数据绑定是MVVM的关键技术，它用于将视图与视图模型进行绑定连接，视图和视图模型的连接可以是双向的，也可以是单向的，视图模型数据的改变可以通过数据绑定功能自动通知视图改变，同样视图的改变也可以通知视图模型数值进行改变。除了数值的连接外，数据绑定还可以支持事件、方法、命令的绑定。数据绑定在框架中是以一个服务模块的方式存在，它由很多的功能组件组成，如数据绑定上下文、类型转换器、表达式解析器、路径解析器、对象和方法代理、属性和Field的访问器等。数据绑定服务是可选的，只有在使用到框架的视图模块，且使用MVVM的方式来开发UI时，它是必要的。当然你也可以不使用本框架的视图模块，而仅仅使用数据绑定服务。

数据绑定服务是一个基础组件，我们可以在游戏初始化脚本中启动数据绑定服务，并且将所有的组件注册到全局上下文的服务容器中。如果有朋友想使用第三方的IoC组件，如Autofac、Zenject等，那么需要参考BindingServiceBundle的代码，将OnStart函数中初始化的所有类用其他的容器来创建。

```
//获得全局上下文
ApplicationContext context = Context.GetApplicationContext();

//初始化数据绑定服务
BindingServiceBundle bindingService = new BindingServiceBundle(context.GetContainer());
bindingService.Start();
```

如果安装了Lua插件，使用Lua编写游戏时，数据绑定服务初始化如下，LuaBindingServiceBundle中增加了有关对Lua对象支持的组件。

```
//获得全局上下文
ApplicationContext context = Context.GetApplicationContext();

//初始化数据绑定服务
LuaBindingServiceBundle bundle = new LuaBindingServiceBundle(context.GetContainer());
bundle.Start();
```

数据绑定示例

```
//创建一个数据绑定集合，泛型参数DatabindingExample是视图，AccountViewModel是视图模型
BindingSet<DatabindingExample, AccountViewModel> bindingSet = this.CreateBindingSet<DatabindingExample, AccountViewModel>();

//绑定Text.text属性到Account.Username上，OneWay是单向,将Account.Username的值赋值到UI控件
bindingSet.Bind(this.username).For(v => v.text).To(vm => vm.Account.Username).OneWay();

//绑定InputField.text到Username属性，双向绑定，修改Username，自动更新InputField控件，修改InputField自动更新Username属性
bindingSet.Bind(this.usernameEdit).For(v => v.text, v => v.onEndEdit).To(vm => vm.Username).TwoWay();

//绑定Button到视图模型的OnSubmit方法，方向属性无效
bindingSet.Bind(this.submit).For(v => v.onClick).To(vm => vm.OnSubmit());

bindingSet.Build();
```

绑定模式

- **OneWay**(View <-- ViewModel)

  单向绑定，只能视图模型修改视图中UI控件的值，ViewModel必须继承了INotifyPropertyChanged接口，并且属性值变化时会触发PropertyChanged事件，否则效果与OneTime一致，只有初始化绑定赋值一次。如Field则只能首次有效。

- **TwoWay**(View <--> ViewModel)

  双向绑定，视图控件修改，会自动修改视图模型，视图模型修改会自动修改视图控件。ViewModel必须支持PropertyChanged事件，UI控件必须支持onEndEdit事件，并且绑定了onEndEdit事件。

- **OneTime**(View <-- ViewModel)

  只赋值一次，只有在绑定关系初始化的时候将ViewModel的值赋值到视图控件上。

- **OneWayToSource**(View --> ViewModel)

  单向绑定，方向与OneWay相反，只能视图UI控件赋值到视图模型的属性。

类型转换器(IConverter)

通常情况下，基本数据类型，当视图控件的字段类型与视图模型字段类型不一致时会自动转换，除非是无法自动转换的情况下才需要自定义类型转换器来支持。但是通过视图模型中保存的图片路径、图片名称或者图集精灵的名称，来修改视图控件上的图片或者图集精灵时，则必须通过类型转换器来转换。

```
//加载一个精灵图集
Dictionary<string, Sprite> sprites = new Dictionary<string, Sprite>();
foreach (var sprite in Resources.LoadAll<Sprite>("EquipTextures"))
{
    if (sprite != null)
        sprites.Add(sprite.name, sprite);
}

//创建一个支持精灵名称到Sprite的转换器
var spriteConverter = new SpriteConverter(sprites);

//获得转换器注册服务，它在数据绑定服务启动时会自动创建并注入上下文容器中
IConverterRegistry converterRegistry = context.GetContainer().Resolve<IConverterRegistry>();

//注册精灵转换器
converterRegistry.Register("spriteConverter",spriteConverter);

//通过视图模型Icon，修改精灵名称，通过spriteConverter转换为对应的Sprite，赋值到图片的sprite属性上。
bindingSet.Bind(this.image).For(v => v.sprite).To(vm => vm.Icon).WithConversion("spriteConverter").OneWay();
```

请查看示例 ListView And Sprite Databinding Tutorials

绑定类型

- **属性和Field绑定**

属性和Field绑定很简单，直接见示例

```
//C#，单向绑定
bindingSet.Bind(this.username).For(v => v.text).To(vm => vm.Account.Username).OneWay();

//C#，双向绑定，双向绑定时视图对象必须支持视图改变的事件，如“onEndEdit”，必须在For函数中配置
bindingSet.Bind(this.usernameEdit).For(v => v.text, v => v.onEndEdit).To(vm => vm.Username).TwoWay();

//C#，非拉姆达表达式的方式
bindingSet.Bind (this.username).For ("text").To ("Account.Username").OneWay ();

--Lua，非拉姆达表达式参数的版本
bindingSet:Bind(self.username):For("text"):To("account.username"):OneWay()
bindingSet:Bind(self.errorMessage):For("text"):To("errors['errorMessage']"):OneWay()
```

- **表达式绑定**

表达式绑定只支持视图模型的一个或者多个属性，通过表达式转换为某个类型的值赋值到视图UI控件上，只能是OneTime或者OneWay的类型。表达式绑定函数，支持拉姆达表达式参数和string参数两种配置方式，C#代码只支持拉姆达表达式参数的方法，代码会自动分析表达式关注的视图模型的一个或者多个属性，自动监听这些属性的改变；Lua代码只支持使用string参数版本的方法，无法自动分析使用了视图模型的哪些属性，需要在参数中配置表达式所使用到的属性。

```
//C#代码，使用拉姆达表达式为参数的ToExpression方法，自动分析监听视图模型的Price属性
bindingSet.Bind(this.price).For(v => v.text).ToExpression(vm => string.Format("${0:0.00}", vm.Price)).OneWay();

--Lua代码，使用string参数版本的ToExpression方法，需要手动配置price属性,如果表达式使用了vm的多个属性，则在"price"后继续配置
bindingSet:Bind(self.price):For("text"):ToExpression(function(vm)
    return string.format(tostring("%0.2f"), vm.price)
end ,"price"):OneWay()
```

- **方法绑定**

方法绑定与属性绑定类似，也支持拉姆达表达式和字符串参数两个版本，方法绑定要确保控件的事件参数类型与视图模型被绑定方法的参数类型一致，否则可能导致绑定失败。

```
//C#，拉姆达表达式方式的绑定，Button.onClick 与视图模型的成员OnSubmit方法绑定
bindingSet.Bind(this.submit).For(v => v.onClick).To(vm => vm.OnSubmit());

//C#，拉姆达表达式方式的绑定，方法带参数，绑定时随便填写一个默认参数就行，此代码并不会被调用，只用来解析绑定关系
bindingSet.Bind(this.emailEdit).For(v => v.onValueChanged).To(vm => vm.OnEmailValueChanged(""));

--Lua，通过字符串参数绑定，Button.onClick 与视图模型的成员submit方法绑定
bindingSet:Bind(self.submit):For("onClick"):To("submit"):OneWay()
```

- **命令和交互请求绑定**

命令是对视图模型方法的一个包装，一般UI按钮onClick的绑定，既可以绑定到视图模型的一个方法，也可以绑定到视图模型的一个命令。但是建议绑定到命令上，命令不但可以响应按钮的点击事件，还能控制按钮的可点击状态，可以在按钮按下后立即使按钮置灰，在按钮事件响应完成后，重新恢复按钮状态。

交互请求(InteractionRequest)交互请求往往都和命令配对使用，命令响应UI的点击事件，处理点击逻辑，交互请求向控制层发生消息控制UI的创建、修改和销毁。

```
//C#，绑定控制层的OnOpenAlert函数到交互请求AlertDialogRequest上
bindingSet.Bind().For(v => this.OnOpenAlert(null, null)).To(vm => vm.AlertDialogRequest);

//绑定Button的onClick事件到OpenAlertDialog命令上
bindingSet.Bind(this.openAlert).For(v => v.onClick).To(vm => vm.OpenAlertDialog);
```

- **集合的绑定**

字典和列表的绑定跟属性/Field绑定基本差不多，见下面的代码

```
//C#，绑定一个Text.text属性到一个字典ObservableDictionary中key ="errorMessage" 对应的对象
bindingSet.Bind(this.errorMessage).For(v => v.text).To(vm => vm.Errors["errorMessage"]).OneWay();
```

- **静态类绑定**

  静态类绑定和视图模型绑定唯一区别就是，静态类绑定创建的是静态绑定集，静态绑定集不需要视图模型对象。

```
//C#，创建一个静态类的绑定集
BindingSet<DatabindingExample> staticBindingSet = this.CreateBindingSet<DatabindingExample>();

//绑定标题到类Res的一个静态变量databinding_tutorials_title
staticBindingSet.Bind(this.title).For(v => v.text).To(() => Res.databinding_tutorials_title).OneWay();
```

Scope Key

在某些视图中，可能需要动态创建绑定关系，动态的移除绑定关系，这里我们提供了一种可以批量的移除绑定关系的方式，那就是Scope Key。

```
//C#,
string scopeKey = "editKey";
bindingSet.Bind(this.username).For(v => v.text).To(vm => vm.Account.Username).WithScopeKey(scopeKey).OneWay();
bindingSet.Bind(this.submit).For(v => v.onClick).To(vm => vm.OnSubmit()).WithScopeKey(scopeKey);

//通过Scope Key移除绑定
this.ClearBindings(scopeKey); //or this.BindingContext().Clear(scopeKey)
```

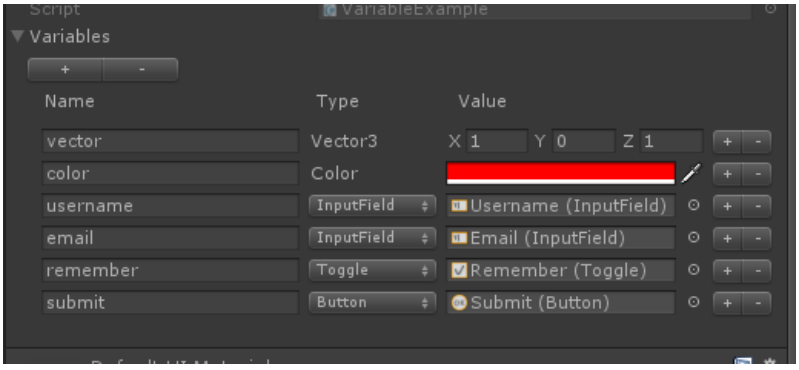
绑定的生命周期

一般来说数据绑定都在视图创建函数中来初始化，通过BindingSet来配置视图控件和视图模型之间的绑定关系，当调用BindingSet的Build函数时，Binder会创建BindingSet中所有的绑定关系对，被创建的绑定对会保存在当前视图的BindingContext中。BindingContext在首次调用时自动创建，同时自动生成了一个BindingContextLifecycle脚本，挂在当前视图对象上，由它来控制BindingContext的生命周期，当视图销毁时，BindingContext会随之销毁，存放在BindingContext中的绑定关系对也会随之销毁。

UI框架

动态变量集(Variables)

在UI的开发过程中，视图脚本往往需要访问、控制UI界面上的UI控件，通常来说，我们要么通过Transform.Find来查找，要么在View脚本中定义一个属性，在编辑UI界面时将控件拖放到这个属性上。第一种方式效率不高，第二种方式新增、删除都要重新改脚本属性，不是那么灵活。 在这里，我提供了第三中方式，VariableArray，这是一个动态的变量集，可以方便的新增和删除，又可以像一个成员属性一样使用。而且它不但支持所有的基本数据类型，还支持Unity组件类型、值类型。




```
//C#，访问变量
Color color = this.variables.Get<Color>("color");
InputField usernameInput = this.variables.Get<InputField>("username");
InputField emailInput = this.variables.Get<InputField>("email");

--Lua，可以直接通过self来访问变量，跟当前Lua表中的成员属性一样
printf("vector:%s",self.vector:ToString())
printf("color:%s",self.color:ToString())
printf("username:%s",self.username.text)
```

```
printf("email:%s",self.email.text)
```

UI视图定位器(IUIViewLocator)

UI视图定位器是一个查询和加载UI视图的服务，它提供了同步和异步加载UI视图的服务。根据项目的不同，可以自定义实现它的功能，你可以从Resources中加载视图，也可以从一个AssetBundle中加载视图，或者两者都支持。

```
//C#，创建一个默认的视图定位器，它支持从Resources中加载视图，如果要从AssetBundle中加载，需要自己实现
IUIViewLocator locator = new DefaultUIViewLocator()

//通过UI视图定位器，根据一个UI路径名加载一个Loading的窗口视图
var window = locator.LoadWindow<LoadingWindow>("UI/Loading");
window.Show();
```

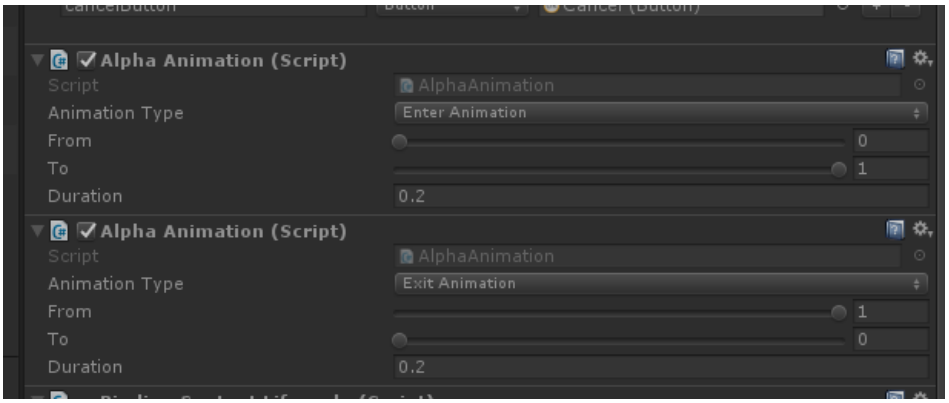
UI视图动画(Animations)

根据一个UI视图打开、关闭、获得焦点、失去焦点的过程，视图动画可以分为入场动画、出场动画、激活动画、钝化动画。继承UIAnimation或者IAnimation，使用DoTween、iTween等，可以创建自己满意的UI动画。

在框架中UIView支持入场动画和出场动画，当打开一个视图或者隐藏一个视图时会可以播放动画。而Window除了支持入场动画和出场动画，还支持激活动画和钝化动画，并且自动控制播放，当一个Window获得焦点时播放激活动画，当失去焦点是播放钝化动画。

如下所示，在Examples中，我创建了一个渐隐渐显的动画，将他们挂在一个Window视图上，并设置为入场动画和出场动画，当窗口打开时逐渐显现，当窗口关闭时慢慢消失。

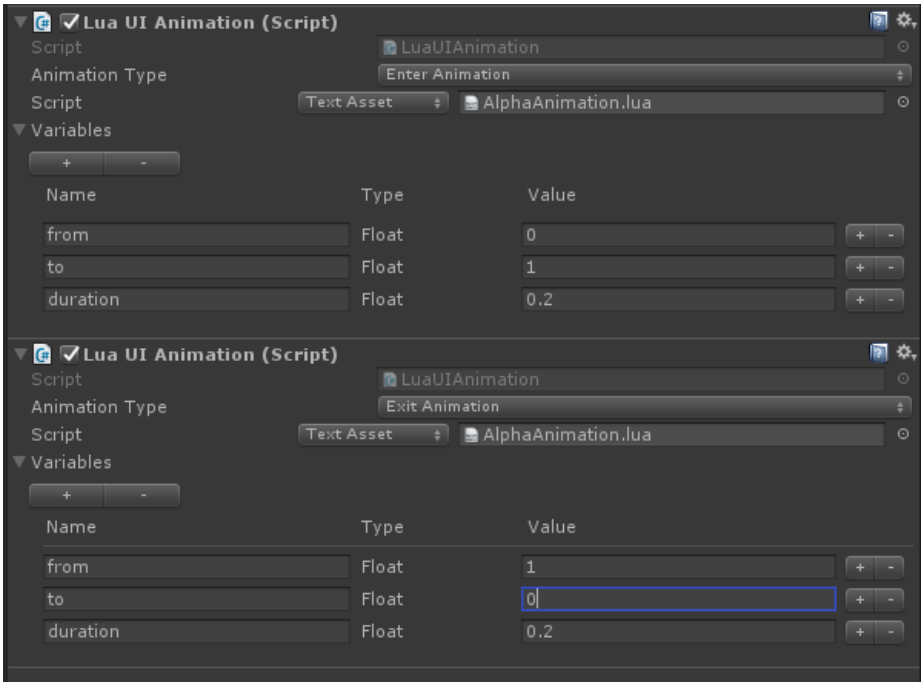
自定义一个C#的渐隐渐显动画




```
public class AlphaAnimation : UIAnimation
{
    [Range (0f, 1f)]
    public float from = 1f;
    [Range (0f, 1f)]
    public float to = 1f;

    public float duration = 2f;

    private IUIView view;

    void OnEnable ()
    {
        this.view = this.GetComponent<IUIView> ();
        switch (this.AnimationType) {
        case AnimationType.EnterAnimation:
            this.view.EnterAnimation = this;
            break;
        case AnimationType.ExitAnimation:
            this.view.ExitAnimation = this;
            break;
        case AnimationType.ActivationAnimation:
            if (this.view is IWindowView)
                (this.view as IWindowView).ActivationAnimation = this;
```

```
            break;
        case AnimationType.PassivationAnimation:
            if (this.view is IWindowView)
                (this.view as IWindowView).PassivationAnimation = this;
            break;
        }

        if (this.AnimationType == AnimationType.ActivationAnimation || this.AnimationType == AnimationType.EnterAnimation) {
            this.view.CanvasGroup.alpha = from;
        }
    }

    public override IAnimation Play ()
    {
        this.view.CanvasGroup.DOFade (this.to, this.duration).OnStart (this.OnStart).OnComplete (this.OnEnd).Play ();
        return this;
    }
}
```

使用DoTween自定义一个Lua的动画




```
require("framework.System")

---
--模块
--@module AlphaAnimation
local M=class("AlphaAnimation",target)

function M:play(view,startCallback,endCallback)
    view.CanvasGroup:DOFade(self.to, self.duration)
        :OnStart(function() startCallback() end)
        :OnComplete(function() endCallback() end)
        :Play()
end

return M
```

UI控件

UGUI虽然为我们提供了丰富的UI控件库，但是在某些时候，仍然无法满足我们的要求，比如我们需要一个性能优越的ListView，这时候我们就需要自定义自己的UI控件。在本框架中，我提供了一些常用的UI控件，比如AlertDialog、Loading、Toast等，在Examples/Resources/UI目录下，你能找到默认

的视图界面，参考这些界面可以重新定义界面外观，修改静态类的ViewName属性可以重新制定视图的加载路径。

下面以AlertDialog为例来介绍它们的用法


```
//对话框视图默认目录路径是UI/AlertDialog，可以通过如下方式修改视图路径
AlertDialog.ViewName = "Your view directory/AlertDialog";

//C#，打开一个对话框窗口
AlertDialog.ShowMessage("This is a dialog test.", "Interation Example", "Yes", null, "No", true,
result =>
{
    Debug.LogFormat("Result:{0}",result);
});
```

视图、窗口和窗口管理器

- **视图(IView/IUIView)**

  视图通俗的讲就是展现给用户所看到的UI界面、图像、动画等。在本框架中，根据游戏视图层的特点，将其分成两大类，场景视图和UI视图。UI视图对应的是IUIView接口，而场景视图对应的是IView接口。

- **视图组(IViewGroup/IUIViewGroup)**

  视图组是一个视图的集合，也可以说是视图容器，它有多个视图组成，在视图组中可以添加、删除子视图。同时视图组本身也是一个视图，它同样可以做为其他视图组的子视图。

  在UI开发中，我们经常会发现一个UI界面可以划分很多的区域，比如Top栏，左边栏，右边栏，Bottom栏，内容区域等等，并且有些部分在多个UI界面之间是可以共享使用的。根据这些特点，我就可以将不同的区域分别做成不同的视图，在最后界面显示时，通过视图组装配成完整的视图，这样既有助于提高代码的重复利用，又大大降低了代码的耦合性和复杂性。**重点说一下，我们可以用这种设计思路来设计游戏的新手引导系统，只有界面需要显示引导时，才将引导界面动态插入到当前的界面中。新手引导的逻辑与正常游戏逻辑完全分离，避免造成引导逻辑和游戏逻辑的高度耦合。**

  同样，在游戏场景视图中，我们也可以将复杂视图拆分成大大小小的视图组和子视图，并且在游戏过程中，动态的添加和删除子视图。比如一个游戏角色，就是场景中的一个子视图，当角色进入视野时添加视图，当从视野消失时，删除视图。

  以王者荣耀日常活动界面为例，可以拆分为顶菜单栏、左侧菜单栏和内容区域，菜单栏视图可以复用，每次只需要改变内容区域的视图即可。

- **窗口(IWindow)**

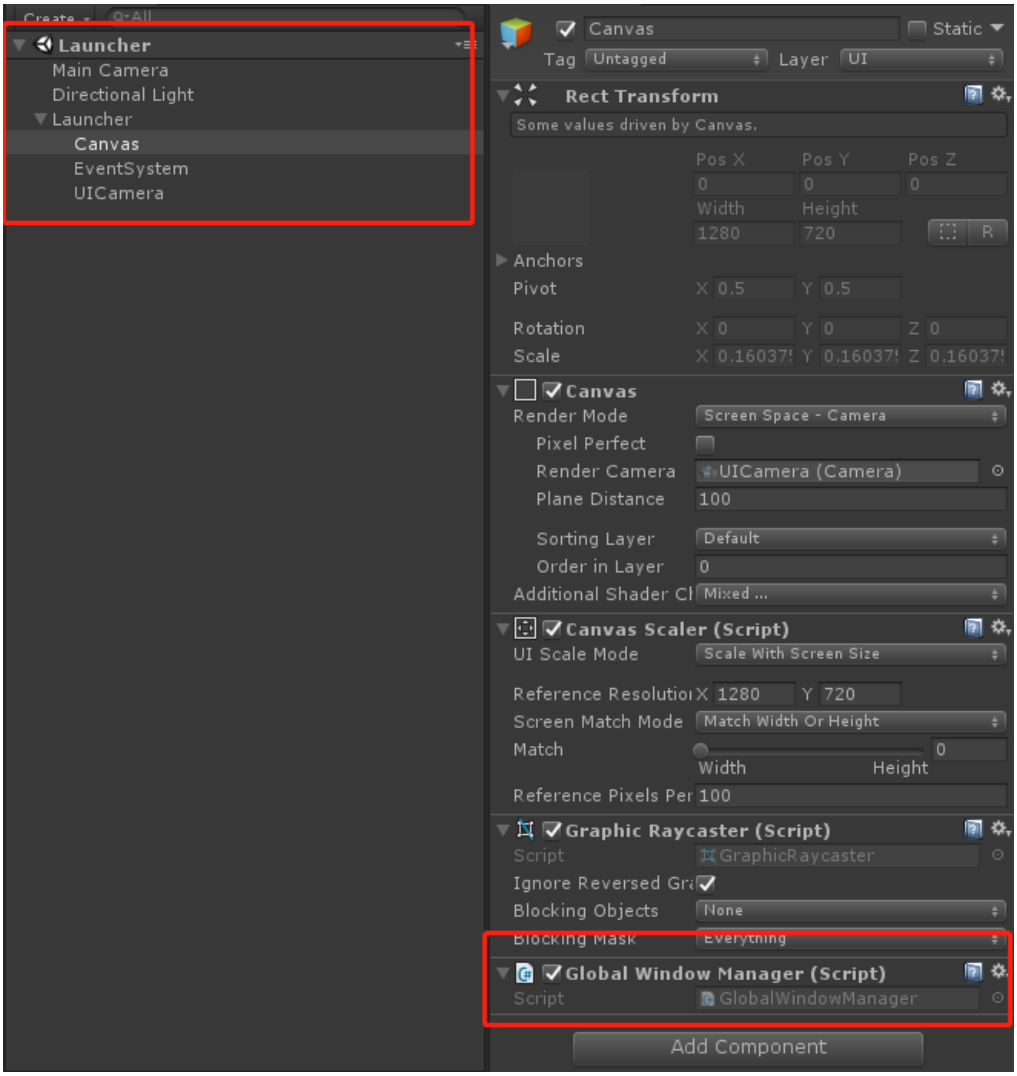
  Window是一个UI界面视图的根容器(IUIViewGroup、IUIView)，同时也是一个控制器，它负责创建、销毁、显示、隐藏窗口视图，负责管理视图、视图模型的生命周期，负责创建子窗口、与子窗口交互等。

```
//C#，创建窗口
public class ExampleWindow : Window
{
    public Text progressBarText;
    public Slider progressBarSlider;
    public Text tipText;
    public Button button;

    protected override void OnCreate(IBundle bundle)
    {
        BindingSet<ExampleWindow, ExampleViewModel> bindingSet = this.CreateBindingSet(new ExampleViewModel());

        bindingSet.Bind(this.progressBarSlider).For("value", "onValueChanged").To("ProgressBar.Progress").TwoWay();
        bindingSet.Bind(this.progressBarSlider.gameObject).For(v => v.activeSelf).To(vm => vm.ProgressBar.Enable).OneWay();
        bindingSet.Bind(this.progressBarText).For(v => v.text).ToExpression(
            vm => string.Format("{0}%", Mathf.FloorToInt(vm.ProgressBar.Progress * 100f)))
        .OneWay();
        bindingSet.Bind(this.tipText).For(v => v.text).To(vm => vm.ProgressBar.Tip).OneWay();
        bindingSet.Bind(this.button).For(v => v.onClick).To(vm => vm.Click).OneWay();
        binding,bound to the onClick event and interactable property.
        bindingSet.Build();
    }

    protected override void OnDismiss()
    {
    }
}

--Lua,创建窗口
require("framework.System")

local ExampleViewModel = require("LuaUI.Startup.ExampleViewModel")

---
--模块
--@module ExampleWindow
```

```
local M=class("ExampleWindow",target)

function M:onCreate(bundle)
    self.viewModel = ExampleViewModel()

    self:BindingContext().DataContext = self.viewModel

    local bindingSet = self:CreateBindingSet()

    bindingSet:Bind(self.progressBarSlider):For("value", "onValueChanged"):To("progressBar.progress"):TwoWay()
    bindingSet:Bind(self.progressBarSlider.gameObject):For("activeSelf"):To("progressBar.enable"):OneWay()
    bindingSet:Bind(self.progressBarText):For("text"):ToExpression(
        function(vm) return string.format("%0.2f%%",vm.progressBar.progress * 100) end,
    "progressBar.progress"):OneWay()
    bindingSet:Bind(self.tipText):For("text"):To("progressBar.tip"):OneWay()
    bindingSet:Bind(self.button):For("onClick"):To("command"):OneWay()
    bindingSet:Build()
end

return M
```

- **窗口容器和窗口管理器(WindowContainer、IWindowManager)**

  窗口管理器是一个管理窗口的容器，游戏启动时首先需要创建一个全局的窗口管理器GlobalWindowManager，将它挂在最外层的根Canvas上（见下图），在这个根Canvas下创建编辑其他的窗口视图。



窗口容器既是一个窗口管理器，又是一个窗口，在窗口容器中可以添加、删除子窗口、管理子窗口，也可以像一个普通窗口一样显示、隐藏。拿我们的MMO游戏来说，一般会创建一个名为"Main"的主窗口容器和一个"Battle"的窗口容器，在主界面打开的所有窗口视图都会放入到Main容器中，但是当进入某个战斗副本时，会将Main容器隐藏，将"Battle"容器显示出来，战斗副本中所有UI窗口都会用Battle容器来管理，退出副本时，

只需要关闭Battle容器，设置Main容器可见，就可以轻松恢复Main容器中窗口的层级关系。

```
//C#，创建一个MAIN容器，默认会在全局窗口管理器中创建
WindowContainer winContainer = WindowContainer.Create("MAIN");
IUIViewLocator locator = context.GetService<IUIViewLocator>();

//在MAIN容器中打开一个窗口
StartupWindow window = locator.LoadWindow<StartupWindow>(winContainer, "UI/Startup/Startup");
ITransition transition = window.Show()
```

交互请求(InteractionRequest)

交互请求(InteractionRequest)在MVVM框架的使用中，我认为是最难理解，最复杂和最绕的地方，而且在网上很多的MVVM示例中，也没有讲到这部分，为什么我们需要交互请求呢？交互请求解决了什么问题？引入交互请求主要目的是为了视图模型(ViewModel)和视图(View)解耦，**在视图模型中，我们不应该创建、引用和直接控制视图，因为那是控制层的工作，不应该是视图模型层的工作，视图层可以依赖视图模型层，但是反之则不允许，切记**。在一个按钮(Button)的点击事件中，往往会触发视图的创建或者销毁，而在MVVM中，按钮点击事件一般都会绑定到视图模型层的一个命令（ICommand）上，即绑定到视图模型的一个成员方法上，在这个方法中往往除了视图无关的逻辑外，还包含了控制视图的创建、打开、销毁的逻辑，前文中提到，这些逻辑会造成对视图层引用和依赖，这是不允许的，所以我们就引入了交互请求(InteractionRequest)的概念，通过交互请求，将视图控制的逻辑发回到控制层中处理（在本框架中就是View、Window脚本，它们既是视图层又是控制层，见前面章节中MVVM架构图）。

请看下面的代码示例，使用交互请求来打开一个警告对话窗，同时在对话窗关闭时，收到用户选择的结果。

```
public class InteractionExampleViewModel : ViewModelBase
{
    private InteractionRequest<DialogNotification> alertDialogRequest;

    private SimpleCommand openAlertDialog;

    public InteractionExampleViewModel()
    {
        //创建一个交互请求，这个交互请求的作用就是向控制层(InteractionExample)发送一个打开对话窗的通知
        this.alertDialogRequest = new InteractionRequest<DialogNotification>(this);

        //创建一个打响应按钮事件的命令
        this.openAlertDialog = new SimpleCommand(Click);
    }

    public IInteractionRequest AlertDialogRequest { get { return this.alertDialogRequest; } }

    public ICommand OpenAlertDialog { get { return this.openAlertDialog; } }

    public void Click()
    {
        //设置命令的Enable为false，通过数据绑定解耦，间接将视图层按钮设置为不可点击状态
        this.openAlertDialog.Enabled = false;

        //创建一个对话框通知
        DialogNotification notification = new DialogNotification("Interation Example", "This is a dialog test.", "Yes", "No", true

        //创建一个回调函数，此回调函数会在AlertDialog对话框关闭时调用
        Action<DialogNotification> callback = n =>
        {
            //设置命令的Enable为true，通过绑定会自动恢复按钮的点击状态
            this.openAlertDialog.Enabled = true;

            if (n.DialogResult == AlertDialog.BUTTON_POSITIVE)
            {
                //对话框Yes按钮被按下
                Debug.LogFormat("Click: Yes");
            }
            else if (n.DialogResult == AlertDialog.BUTTON_NEGATIVE)
            {
                //对话框No按钮被按下
                Debug.LogFormat("Click: No");
            }
```

```
        };

        //交互请求向View层OnOpenAlert函数发送通知
        this.alertDialogRequest.Raise(notification, callback);
    }
}

public class InteractionExample : WindowView
{
    public Button openAlert;
    protected override void Start()
    {
        InteractionExampleViewModel viewModel = new InteractionExampleViewModel();
        this.SetDataContext(viewModel);

        //创建一个bindingSet
        BindingSet<InteractionExample, InteractionExampleViewModel> bindingSet;
        bindingSet = this.CreateBindingSet<InteractionExample, InteractionExampleViewModel>();

        //绑定本视图的OnOpenAlert函数到视图模型的交互请求AlertDialogRequest，当交互请求触发时，自动调用OnOpenAlert函数
        bindingSet.Bind().For(v => this.OnOpenAlert(null, null)).To(vm => vm.AlertDialogRequest);

        //绑定按钮的onClick事件到视图模型的OpenAlertDialog命令上
        bindingSet.Bind(this.openAlert).For(v => v.onClick).To(vm => vm.OpenAlertDialog);

        bindingSet.Build();
    }

    //创建和打开对话框的函数，通过交互请求触发
    private void OnOpenAlert(object sender, InteractionEventArgs args)
    {
        //收到视图模型层交互请求alertDialogRequest发来的通知

        //得到通知数据
        DialogNotification notification = args.Context as DialogNotification;

        //得到AlertDialog窗口关闭时的回调函数
        var callback = args.Callback;

        if (notification == null)
            return;

        //创建一个对话窗
        AlertDialog.ShowMessage(notification.Message, notification.Title, notification.ConfirmButtonText, null,
            notification.CancelButtonText,
            notification.CanceledOnTouchOutside,
            (result) =>
            {
                //将对话窗按钮事件响应结果赋值到notification，传递到视图模型层使用
                notification.DialogResult = result;

                //对话窗关闭时，调用交互请求中设置的回调函数，通知视图模型层处理后续逻辑
                if (callback != null)
                    callback();
            });
    }
}
```

请查看示例 Interaction Tutorials

Lua

模块与继承

利用lua的原表继承，在lua开发中模拟了类(模块)和继承的概念，通过System模块的class函数，可以定义模块、继承模块，继承C#类，扩展C#实例，

以面向对象的思路编写lua代码。

通过下面的代码示例，我们来看看如何定义模块和继承模块

```
-- 定义一个名为 Animal 的基类
local Animal = class("Animal")

-- Animal类的构造函数，通过Animal()来创建Animal类的实例，同时会调用这个构造函数
function Animal:ctor(...)
end

-- 为Animal定义一个walk()的方法
function Animal:walk()
    print("animal walk")
end

-- 为Animal定义一个run()方法
function Animal:run()
    print("animal run")
end

-- 定义一个名为Cat的类，它继承了Animal类
local Cat = class("Cat",Animal)

-- Cat类的构造函数
function Cat:ctor()
    -- 重载了构造函数，会覆盖父类构造函数，通过如下显示的调用父类构造函数
    Cat.super.ctor(self)
    self.age = 5
end
```

要在lua继承一个C#类，那么这个类必须要能通过new关键字来实例化。比如MonoBehaviour脚本类，无法通过new关键字来实例化，是无法在lua中继承的。在class函数中，第一个参数是类名，第二个参数必须是C#类的实例化函数，看如下代码。

```
-- 定义一个继承C#类ResourcesViewLocator的模块，推荐模块的变量名默认都使用M
local M = class("LuaResourcesViewLocator",function(...)
    return CS.Loxodon.Framework.Examples.ResourcesViewLocator(...)
end)

function M:LoadView(name)

    --代码省略

end

return M
```

MonoBehaviour脚本无法被继承，但是它的实例可以被lua扩展，使用class函数，我们可以为它添加新的属性和方法，与C#类继承不同，class第二个参数是一个C#类的实例。请看lua示例中，C#脚本LuaLauncher的扩展代码。

"target"对象是在C#脚本LuaLauncher中，在初始化lua脚本环境时将自己的实例注入到lua环境的，在本框架所有的扩展脚本中，统一使用"target"的变量名,请在游戏逻辑开发中遵循这一规则。

C#代码，LuaLauncher脚本中初始化lua执行环境的部分。

```
var luaEnv = LuaEnvironment.LuaEnv;
scriptEnv = luaEnv.NewTable();

LuaTable meta = luaEnv.NewTable();
meta.Set("__index", luaEnv.Global);
scriptEnv.SetMetaTable(meta);
meta.Dispose();

//将this注入到lua环境表中，这里请统一使用target变量名
scriptEnv.Set("target", this);
```

```
string scriptText = "";
if(script.Type == ScriptReferenceType.TextAsset)
    scriptText = script.Text.text;
else
    scriptText = string.Format("return require(\"{0}\")", script.Filename);

object[] result = luaEnv.DoString(scriptText, string.Format("{0}({1})", "Launcher", this.name), scriptEnv);

if (result.Length != 1 || !(result[0] is LuaTable))
    throw new Exception();

metatable = (LuaTable)result[0];

onAwake = metatable.Get<Action<MonoBehaviour>>("awake");
onEnable = metatable.Get<Action<MonoBehaviour>>("enable");
onDisable = metatable.Get<Action<MonoBehaviour>>("disable");
onStart = metatable.Get<Action<MonoBehaviour>>("start");
onDestroy = metatable.Get<Action<MonoBehaviour>>("destroy");
```

通过lua扩展LuaLauncher脚本的功能，awake、enable、disable、start、destroy函数都可以在lua中实现，在C#中调用。

```
require("framework.System")

local WindowContainer = CS.Loxodon.Framework.Views.WindowContainer
local Context = CS.Loxodon.Framework.Contexts.Context
---
--Launcher 模块，参数target是约定的，请不要修改。
--@module Launcher
local M=class("Launcher",target)

function M:start()
    -- 获得应用上下文，一个游戏建议创建应用上下文和玩家上下文。
    -- 全局的服务都放入应用上下文中，如账号服务，网络组件，配置服务等基础组件和服务
    -- 只与某个玩家相关的如背包服务、装备服务、角色服务都放入玩家上下文，当登出游戏可以统一释放
    local context = Context.GetApplicationContext()

    -- 从应用上下文获得一个视图定位器
    local locator = context:GetService("IUIViewLocator")

    -- 创建一个名为MAIN的窗口容器
    local winContainer = WindowContainer.Create("MAIN")

    -- 通过视图定位器，加载一个启动窗口视图
    local window = locator:LoadWindow(winContainer, "LuaUI/Startup/Startup")
    window:Create() --创建窗口
    local transition = window:Show() --显示窗口，返回一个transition对象，窗口显示一般会有窗口动画，所以是一个持续过程的操作
    transition:OnStateChanged(function(w,state) print("Window:"..w.Name.." State:"..state:ToString()) end) --监听显示窗口过程的窗口状态
    transition:OnFinish(function() print("OnFinished")  end) --监听窗口显示完成事件

    print("lua start...")
end

return M
```

Lua的ObserableObject

Lua的Table要满足MVVM数据绑定的要求，在属性改变时能够触发属性修改的通知，那么就必须继承ObserableObject对象。它与C#的ObserableObject功能类似，只是为了适应Lua开发，用Lua语言重新实现的一个版本。在Lua中定义的视图模型和子视图模型，都必须继承这个类。下面请看示例

```
require("framework.System")

local ObservableObject = require("framework.ObservableObject")
```

```
---
--创建一个Account视图模型
--@module AccountViewModel
local M = class("AccountViewModel",ObservableObject)

function M:ctor(t)
    --执行父类ObservableObject的构造函数，这个重要，否则无法监听数据改变
    Account.super.ctor(self)

    self.id = 0
    self.username = ""
    self.Password = ""
    self.email = ""
    self.birthday = os.time({year =1970, month = 00, day =00, hour =00, min =00, sec = 00})
    self.address = ""

    if t and type(t)=="table" then
        for k,v in pairs(t) do self[k] = v end
    end
end

return M
```

Lua中使用Unity的协程

XLua为我们提供了一个在lua中创建迭代器(IEnumerator)的函数util.cs_generator()。通过这个函数的可以将一个lua方法包装成一个C#的IEnumerator，然后在C#中放入协程执行。

下面的doLoad函数模拟了一个加载任务，执行了一个从1到50的循环，利用lua协程的yield方法，每个次循环睡眠0.1秒。

```
---
-- 模拟一个加载任务
function M:doLoad(promise)
    print("task start")

    for i = 1, 50 do
        --如果有取消请求，即调用了ProgressResult的Cancel()函数，则终止任务
        if promise.IsCancellationRequested then
            break
        end

        promise:UpdateProgress(i/50) --更新任务进度

        --这里coroutine.yield中可以不传入参数，则表示是每帧执行一次，
        --也可以传入所有继承了YieldInstruction的参数，如:UnityEngine.WaitForSeconds(0.1)
        --还可以传入一个IEnumerator对象，如：AsyncResult.WaitForDone()
        coroutine.yield(CS.UnityEngine.WaitForSeconds(0.1))--等待0.1秒
    end
    promise:UpdateProgress(1)
    promise:SetResult() --设置任务执行完成
    print("task end")
end
```

使用XLua的函数util.cs_generator将doLoad包装成IEnumerator放入Executors.RunOnCoroutineNoReturn中执行。

```
local result = ProgressResult(true)
Executors.RunOnCoroutineNoReturn(util.cs_generator(function() self:doLoad(result) end))
```

联系方式

邮箱: yangpc.china@gmail.com
网站: https://cocowolf.github.io/loxodon-framework/
QQ群: 622321589