

Cs2Lua

Dreaman

2021年

主要內容

- 第一章 c#与lua
- 第二章 Cs2Lua简介
- 第三章 Slua原理、 API生成与修改
- 第四章 c#翻译到lua
- 第五章 lua运行时
- 第六章 值类型处理
- 第七章 性能问题与优化策略

主要內容

➤ 第一章 c#与lua

1. 语言排名
2. 语言选择

c#与Lua

● 语言排名

● c#

	Nov 2020	Nov 2019	Change	Programming Language	Ratings	Change
1	2	2	▲	C	16.21%	+0.17%
2	3	3	▲	Python	12.12%	+2.27%
3	1	1	▼	Java	11.68%	-4.57%
4	4			C++	7.60%	+1.99%
5	5	5		C#	4.67%	+0.36%
6	6	6		Visual Basic	4.01%	-0.22%
7	7	7		JavaScript	2.03%	+0.10%
8	8	8		PHP	1.79%	+0.07%
9	16	16	▲	R	1.64%	+0.66%
10	9	9	▼	SQL	1.54%	-0.15%
11	14	14	▲	Groovy	1.51%	+0.41%
12	21	21	▲	Perl	1.51%	+0.68%
13	20	20	▲	Go	1.36%	+0.51%
14	10	10	▼	Swift	1.35%	-0.31%
15	11	11	▼	Ruby	1.22%	-0.04%

c#与Lua

• 语言排名

• Lua

25	Rust	0.58%
26	SAS	0.58%
27	Dart	0.54%
28	COBOL	0.53%
29	Scala	0.53%
30	Julia	0.50%
31	PowerShell	0.48%
32	D	0.46%
33	ABAP	0.43%
34	Fortran	0.41%
35	Lisp	0.40%
36	Kotlin	0.38%
37	Lua	0.37%
38	Ada	0.37%
39	VHDL	0.35%
40	Prolog	0.35%

c#与Lua

- 语言排名

- c#长期排名

Programming Language	2020	2015	2010	2005	2000	1995	1990	1985
C	1	2	2	1	1	2	1	1
Java	2	1	1	2	3	29	-	-
Python	3	6	6	7	23	9	-	-
C++	4	3	3	3	2	1	2	9
C#	5	4	5	6	10	-	-	-
JavaScript	6	8	10	10	7	-	-	-
PHP	7	7	4	4	19	-	-	-

c#与Lua

- 技术上，选择哪个语言没那么重要
- 经济上，语言很重要

c#与Lua

● 语言选择 Android



iOS



主要內容

➤ 第二章 Cs2Lua简介

1. 技术路线选择
2. 基础架构
3. 基于组件的开发
4. 实际架构
5. Cs2Lua运行时三部分

Cs2Lua 简介

➤ 技术路线选择—反编译vs编译

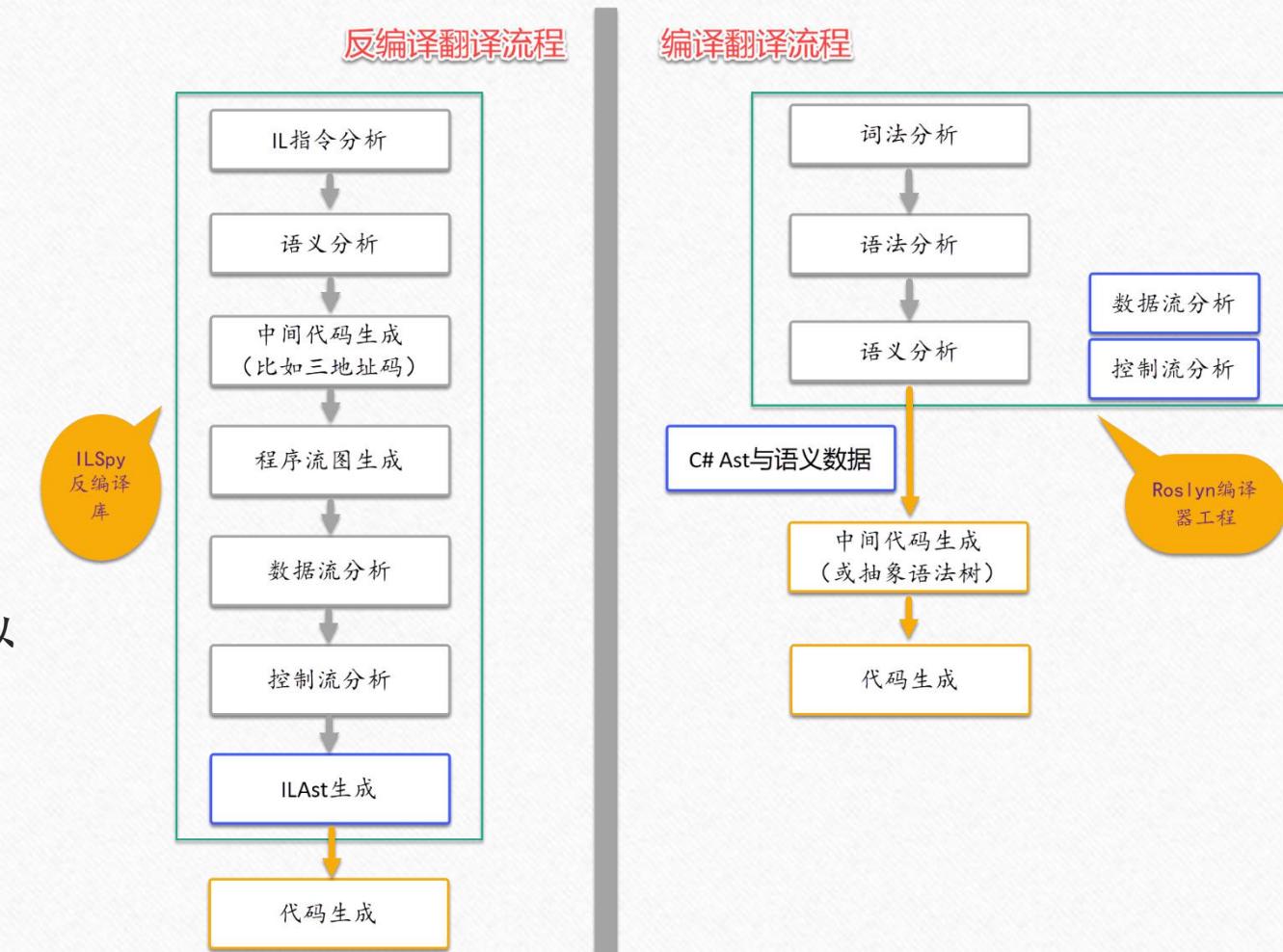
◆ 反编译（输入dll+ pdb）

- 非官方逆向工具库
- 仅基于ILAst翻译，缺少语义信息
- TkLua与Bridge.lua采用

◆ 编译（输入cs）

- 官方开源库Roslyn
- 基于C# Ast+语义信息翻译，可以使用数据流与控制流分析
- Csharp.lua采用

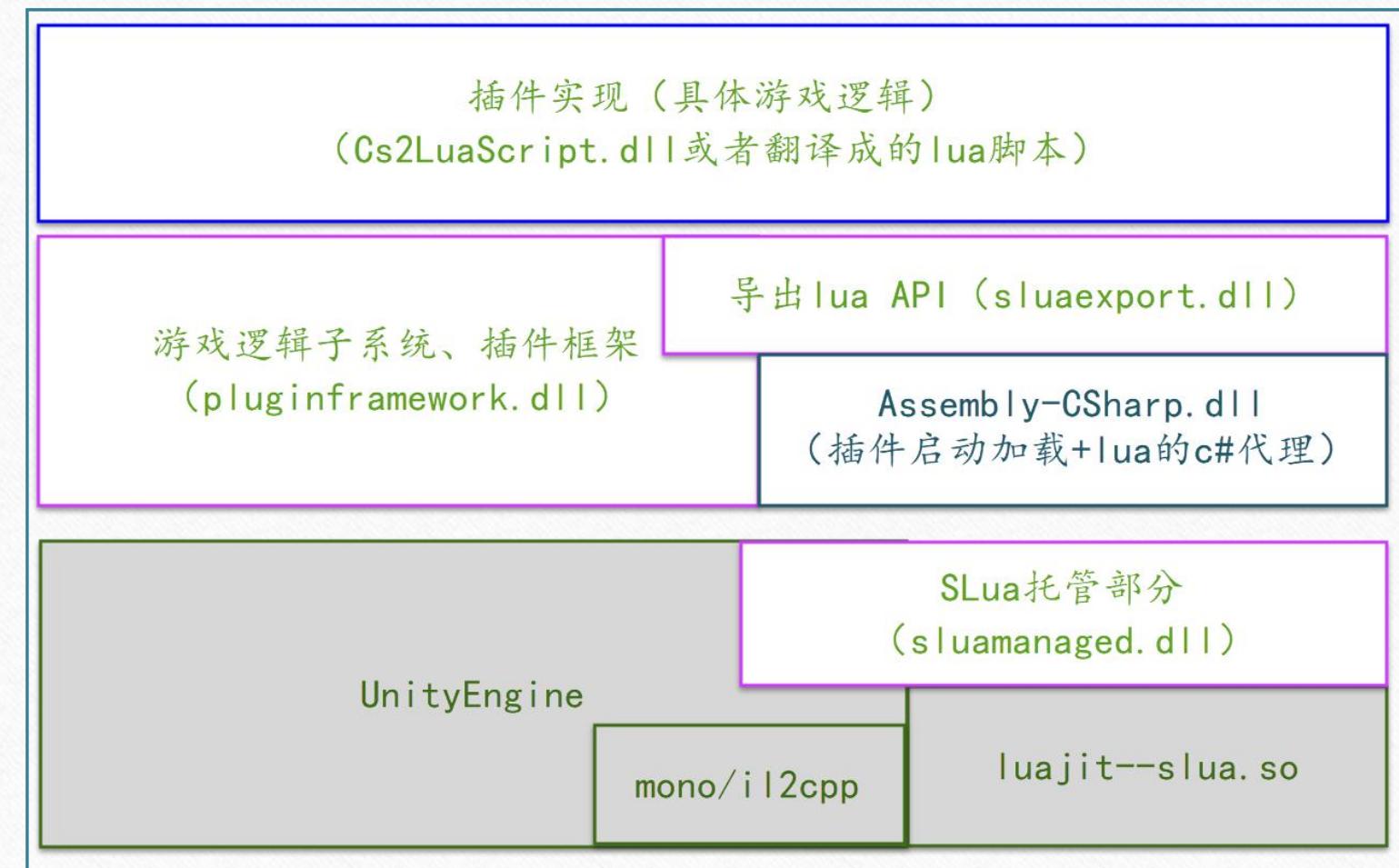
◆ Cs2Lua采用编译方式



Cs2Lua 简介

- 基础架构

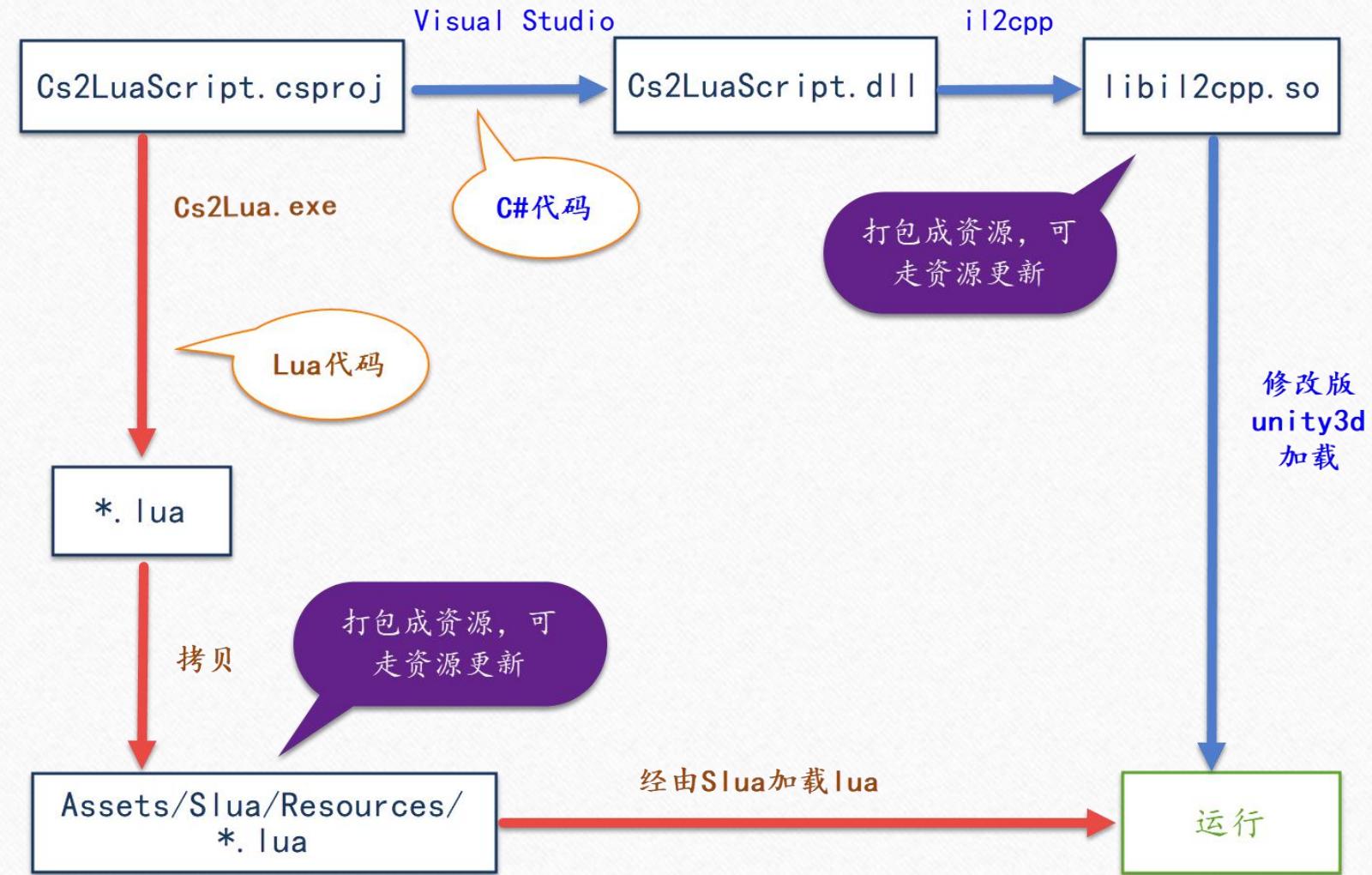
- 模块组织
- 层次结构
- 最初设计



Cs2Lua 简介

基础架构

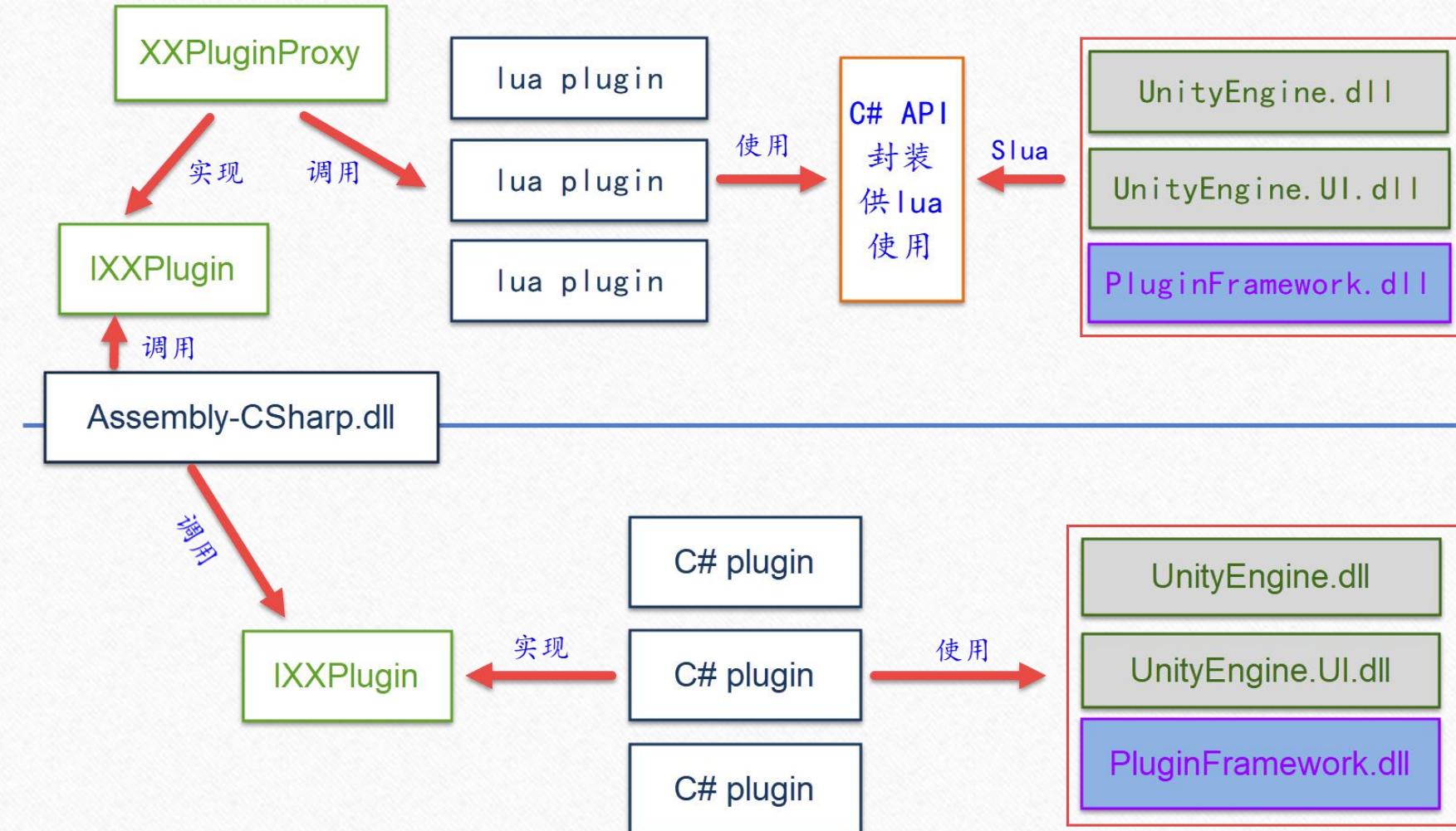
- C#部署方式
- Lua部署方式



Cs2Lua 简介

- 基础架构

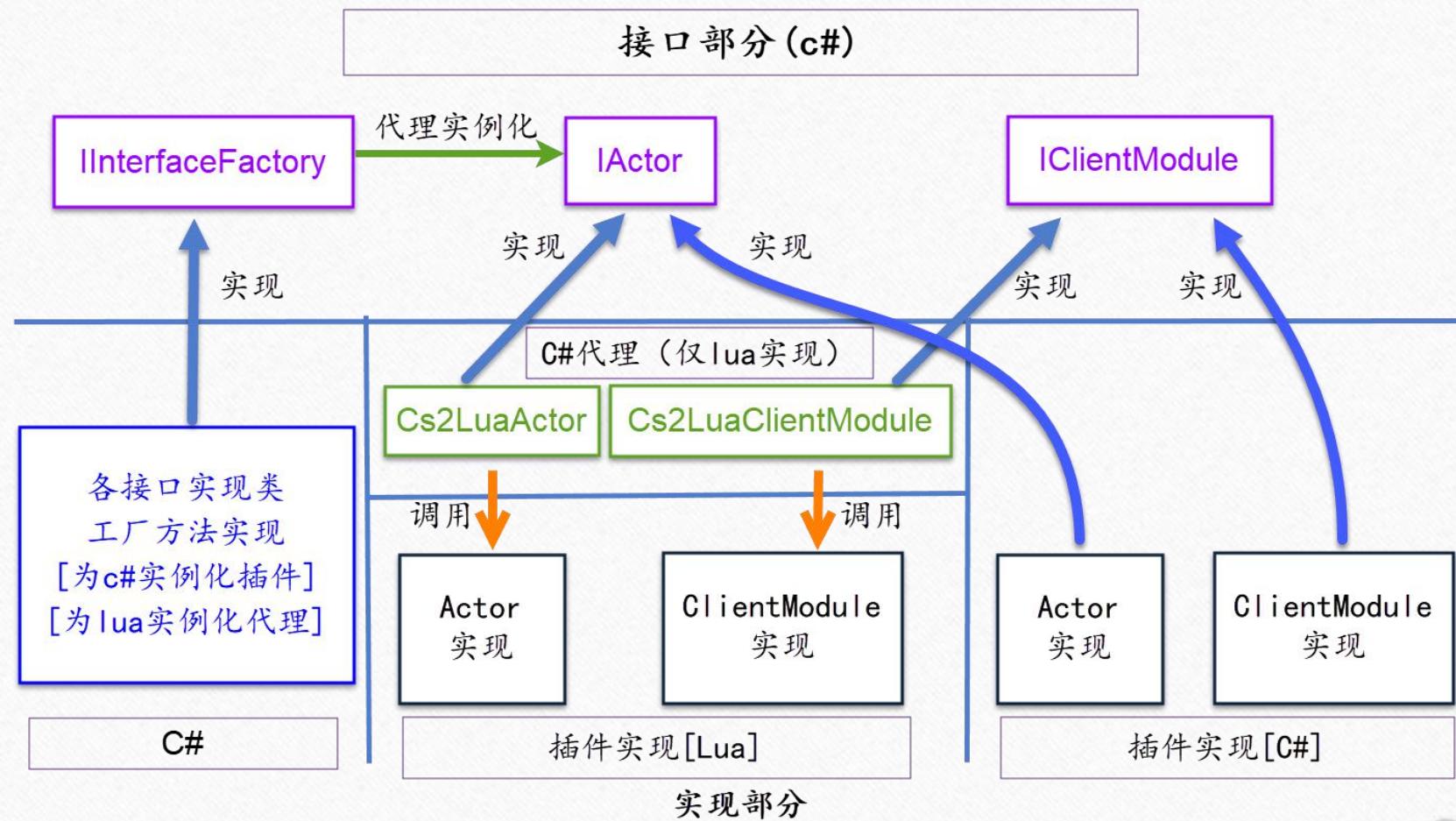
- 插件工作机制



Cs2Lua 简介

- 基础架构

- 基于接口通信的组件
- 理想：组件间交互均通过接口进行
 - 组件是运行时可部署的单位
 - 组件可以使用任一支持的语言实现
 - 任何组件都可以被任一语言实现的组件替代



Cs2Lua 简介

- 基于组件的开发

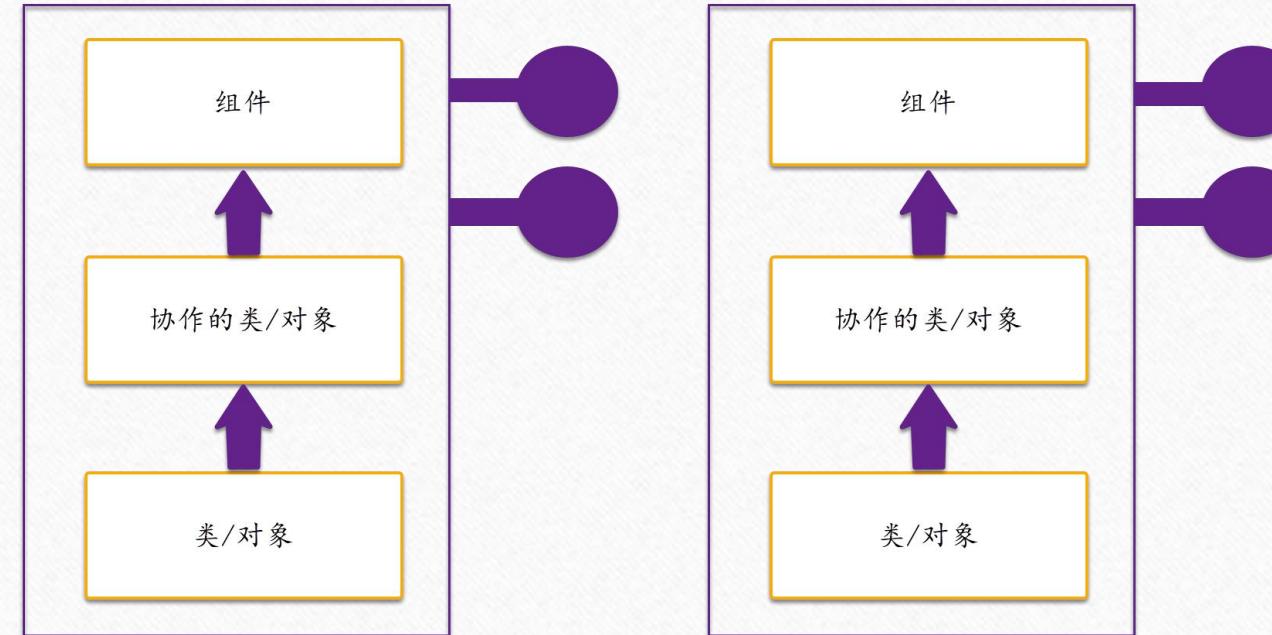
- 微软com组件
 - Office系列
 - Dotnet framework底层

- 优点

- 结构清晰
- 松散耦合
- 可扩展
- 可更新

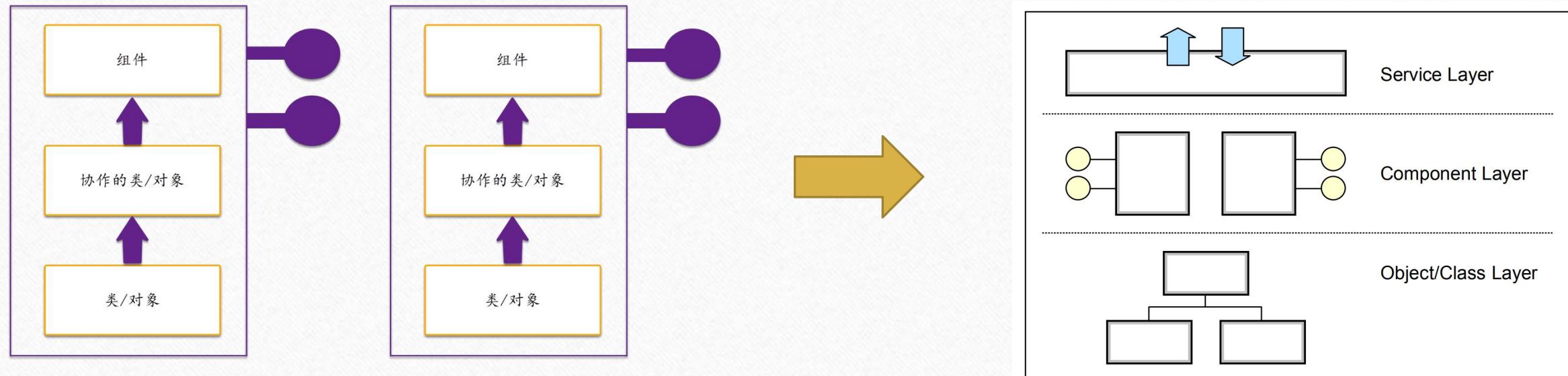
- 缺点

- 开发难度较大
- 需要先进行接口设计，制定组件间交互的协议
- 严格遵循基于接口的开发流程
- 组件间交互方式受到严格限制
 - 不像在同一个工程里同一语言各类间自由引用



Cs2Lua 简介

- 基于组件的开发
 - 往下一步是服务，服务的普及化是微服务

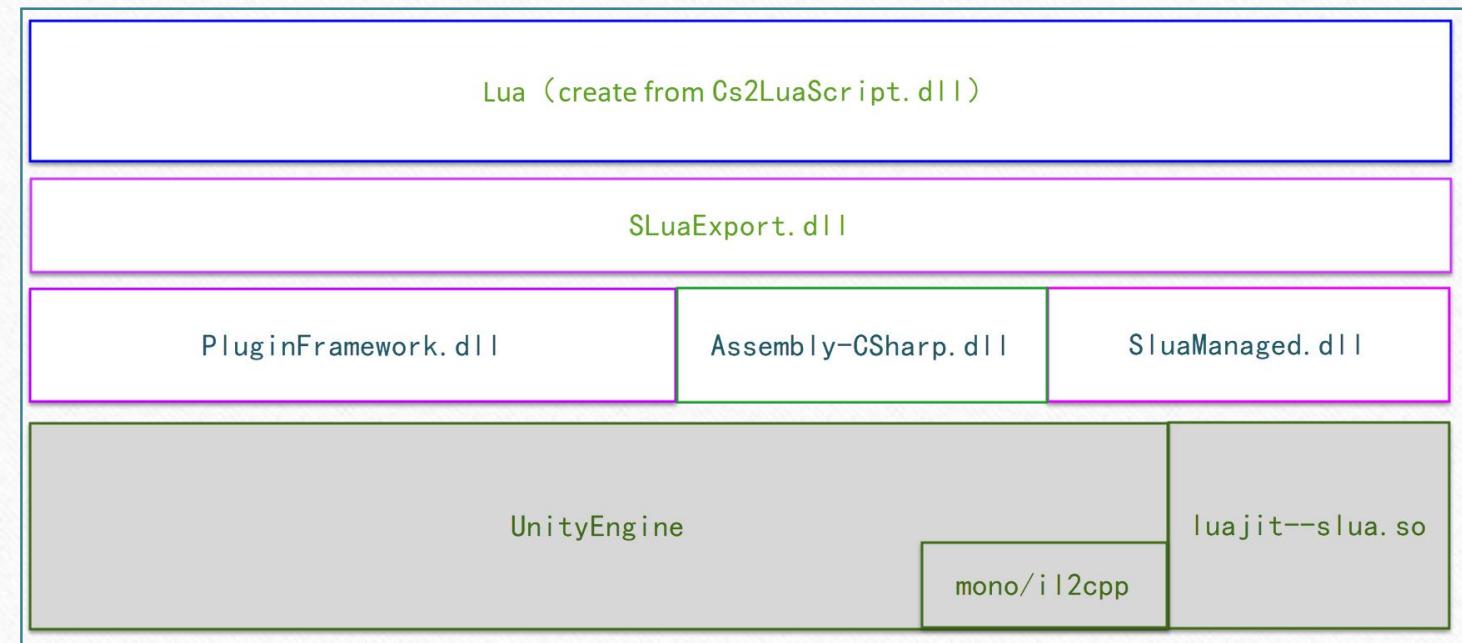


不涉及云游戏的时候，服务与我们关系不大

Cs2Lua 简介

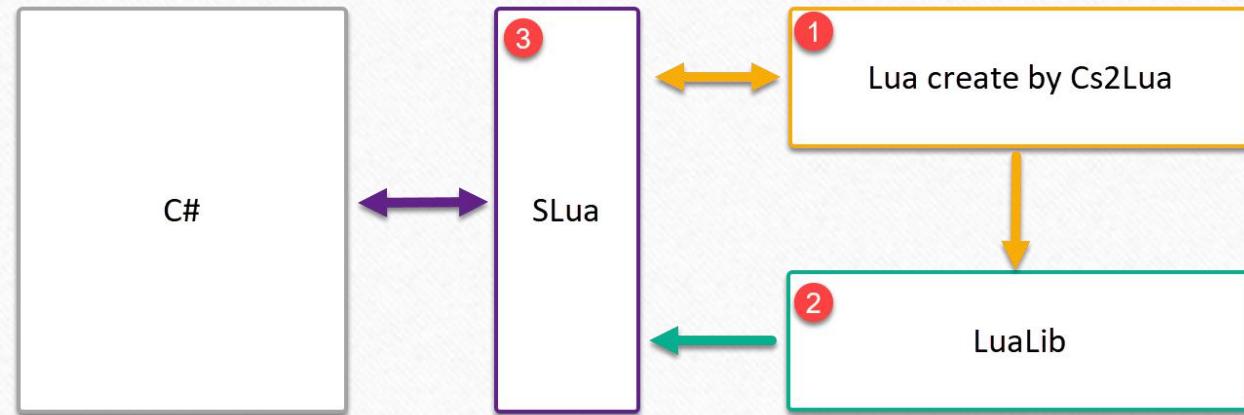
- 实际项目运用

- 保留了插件架构
 - Cs2LuaScript整体可以编译为dll或者翻译为lua
 - Cs2LuaScript整体可以更新
 - 单个c#实现可移植到lua
 - 避免跨语言的组件间的交互
- 没有强制要求基于组件的开发



Cs2Lua 简介

- Cs2Lua运行时三部分
 - Cs2Lua翻译
 - 游戏逻辑
 - 保持语义将c#语法变换到Lua语法
 - 无法变换的委托到LuaLib的函数
 - LuaLib运行时库
 - 辅助实现Lua模拟c#语义
 - 对象模型、操作符处理、位运算、整数运算、基础值方法处理、值类型处理、基本容器、delegate处理、发布-订阅处理
 - SLua
 - 实现c#与Lua的互操作
 - 允许Lua调用c#
 - 允许c#调用Lua
 - 自动生成c# API的Lua封装
 - 我们另外添加：自动生成Lua的c#代理类



主要內容

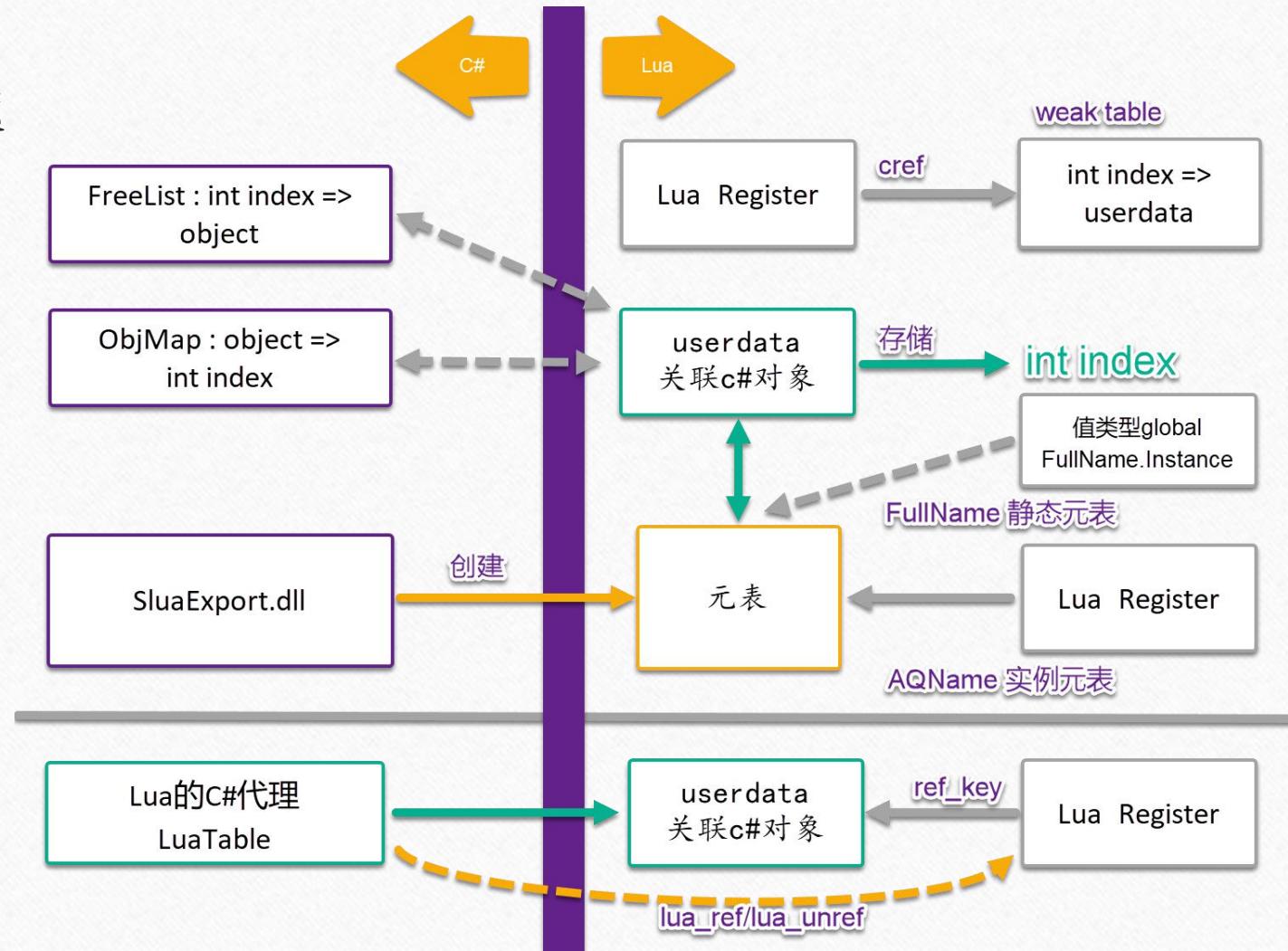
➤ 第三章 Slua原理、 API生成与修改

- 1、Slua的原理
- 2、导出Api限制
- 3、手写lua与翻译对Slua的要求
- 4、主要修改

Slua原理、API生成与修改

- Slua的原理

- 主要解决lua与c#的互操作问题
- c#对象与lua表如何关联



Slua原理、API生成与修改

- 导出Api限制
 - 无法导出泛型类型
 - 无法导出泛型方法
 - 需要按约定提供一个非泛型版本
 - 对c#的限制
 - 不翻译为lua的模块
 - 公有类不能是泛型类
 - 公有方法不能是泛型方法

Slua原理、 API生成与修改

- 导出Api限制
 - 函数重载
 - Slua为方便手写lua，重载函数不进行换名导出
 - 多个版本共用一个名称，运行时根据参数数量与类型来匹配
 - 最后一个参数是变参的版本与普通参数个数相同的版本无法区分
 - 某些类型从lua实参无法区分，如float与int等
 - Cs2Lua不需要手写lua，所以修改为换名导出函数的多个重载版本
 - 避免变参函数
 - 变参不管是在c#还是翻译到lua都存在GC，因为c#的语义是把变参组织成一个数组，这个操作是在调用时处理的
 - Cs2Lua翻译时需要保持与c#语义一致，所以变参翻译后也会构造一个数组(lua table)

Slua原理、 API生成与修改

- 导出Api限制
 - Dotnet framework库里的容器类无法导出
 - c#导出接口里使用非泛型版本容器
 - 继承泛型容器并提供具体类型
 - 翻译到lua的c#代码假设泛型容器类是lua实现的
 - lualib里的容器类实现
 - Cs2Lua翻译时会检查下面2种情形并报错
 - 不能作为参数传给c# dll的方法
 - 不能作为变量接受c# dll方法的返回值
 - 泛型方法无法导出，需要提供一个非泛型版本，泛型参数作为类型为Type的实参传入（只能部分实现泛型版本功能，比如不能new泛型参数的新实例）
 - Cs2Lua在翻译时会寻找泛型方法的对应非泛型版本并生成调用非泛型版本的代码
 - 适应Unity里的GetComponent等泛型方法的非泛型版本约定

Slua原理、 API生成与修改

- 手写lua与翻译对Slua的要求
 - 手写lua要考虑书写简洁与方便理解接口
 - 可以只导出部分符合lua语言特点的c#特性
 - 手写lua时可以选择不使用那些有差异或不支持的特性
 - **手写lua的思维方式是lua脚本的方式**
 - 翻译要求翻译出的lua实现c#代码的语义
 - Lua与c#在很多语言特性上语义差异巨大
 - Cs2Lua需要为此进行大量的相关转换或定制处理
 - **Cs2Lua翻译出的lua的执行机制是c#语言的方式**

Slua原理、API生成与修改

- 主要修改

- 导出接口方法
- 允许继承类用作接口实例
- 重载方法换名分别导出
- 操作符方法导出成静态方法
- 构造函数导出成静态方法
- Lua元表操作只保留 __gc 与 __tostring
- 协程机制修改
- 值类型lua代码挪到lualib
- 实例元表记录到lua全局变量（备用）

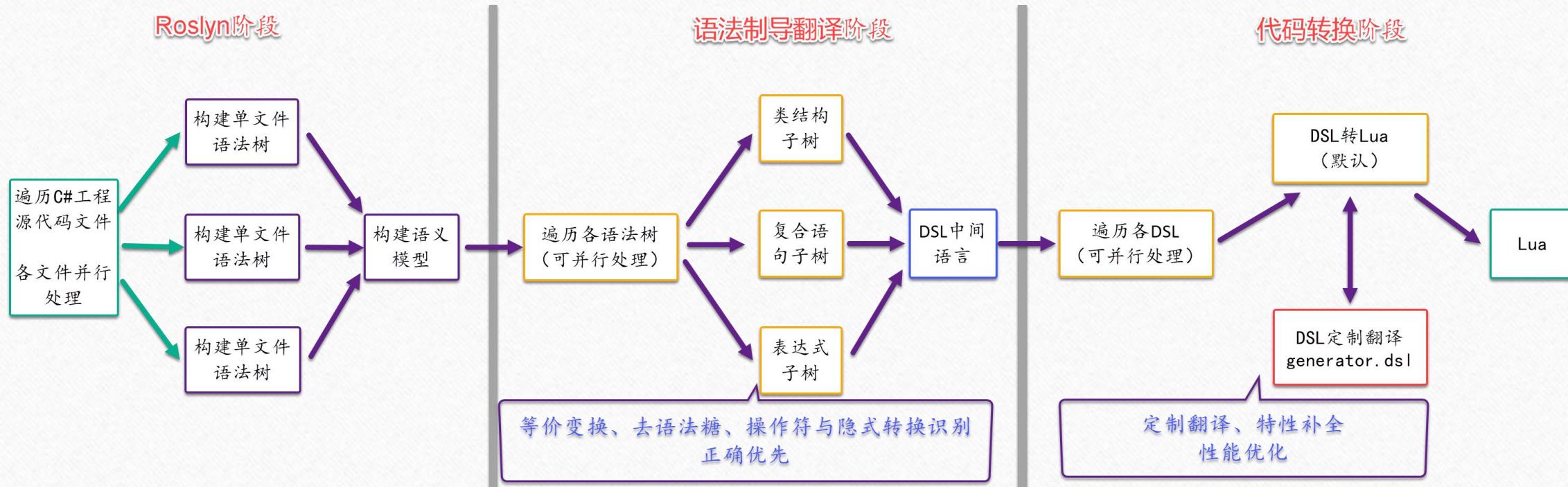
主要內容

➤ 第四章 C#翻译到lua

1. 语法制导与两阶段翻译
2. 支持的特性
3. 不支持的特性
4. 翻译变换
5. 属性控制与定制翻译
6. 代码生成配置
7. 定制翻译脚本
8. 委托到lua运行时的处理

C#翻译到Lua

语法制导与两阶段翻译



C#翻译到Lua

- 语法制导与两阶段翻译

- 第一步去语法糖与简化语言特性，采用语法制导的翻译方式翻译到DSL
 - 相当于是翻译到一个简化的中间语言，这一步利用Roslyn工程的语法与语义信息，有时涉及多次遍历语法树收集信息，逻辑上比较复杂
 - 通过定义语序更接近C#语法构造的中间语言，一方面减少翻译时需要提前收集信息的次数，另一方面减少目标语言特性对翻译的干扰，便于保证翻译的正确性
 - 同时，翻译不依赖目标语言，也保证了这一步的稳定性，避免频繁的变化产生不可控的修改
- 第二步从DSL转换到Lua，这一步聚主要是添加针对Lua的处理或语法特点
 - 相对于翻译到DSL，这一步相对来说比较简单，基本是对函数或语句的语法变换，更容易支持变化
 - 这一步里引入了可配置翻译与定制翻译机制
 - 由于C#与Lua语言在语义上的差异，存在许多语言特性无法在通用翻译层面保证完全等价或者性能上可接受，支持可配置翻译与定制翻译允许针对特定问题域选取正确的或性能上较优的翻译结果
 - 为了避免陷入过多细节，后续翻译的介绍我们忽略这2个步骤，直接描述C#到Lua的效果，但定制翻译例外，它是在第一步结果上对第二步的定制



C#翻译到Lua

• 支持的特性

- 类、方法、indexer、特性、事件、委托、字段等类定义相关的机制，静态与实例2类
- 基本语句与表达式
- 接口（转换时忽略）
- partial类
- 预处理与属性，转换时处理，目标lua不包含
- 类方法重载（overload非override）
- lambda表达式与lambda函数
- 匿名类、匿名委托等
- 自定义类支持generic（为每个generic实例生成一份代码，generic类本身不生成代码）
- 数组与集合支持
- 枚举支持
- 支持ref/out参数
- 扩展方法

C#翻译到Lua

- 支持的特性

- yield与协程
- 操作符重载
- 部分支持Attribute（目前会生成Cs2Lua_attributes.lua/txt文件，包含了所有自定义代码里用到的attribute，但lualib.lua里未实现自定义属性的访问机制）
- 部分支持自定义struct，语法层面基本完成，拷贝语义需要在lualib.lua里实现

C#翻译到Lua

• 不支持的特性

- 不完全支持类的继承与泛型（可以实现接口），主要是因为lua的元表机制很难完美实现c#对象继承的详细语义同时还保持良好的效率与可理解性
- 忽略异常相关语句（try/catch/finally/filter），简单支持只捕获Exception异常的try/catch/finally
- 不支持指针与内存相关操作fixed/*/&/stackalloc
- 不支持async/wait/lock等异步或并发相关设施
- 不支持label与goto语句
- 不支持checked/unchecked语句
- 不支持linq语法糖（直接调用方法就可以，而且c#的linq支持本来也不如visual basic全，这语法风格与c#有点不搭，放弃了）
- 自定义struct未完全实现拷贝语义（需要在lualib.lua里完善）
- 不支持c# 7.0及以后版本引入的模式匹配相关语法与本地方法

C#翻译到Lua

- 翻译变换

- ref/out参数【假设c#函数定义int f(int a, int b, ref int c, out int d)】

```
r = f(a, b, ref c, out d)
```

```
=>
```

```
r, c, d = f(a, b, c)
```

C#翻译到Lua

- 翻译变换

- 带ref/out参数函数出现在表达式中

```
v1+v2+f(a,b,ref c,out d)
```

=>

```
v1+v2+(function() local r; r,c,d = f(a,b,c); return r; end)()
```

C#翻译到Lua

- 翻译变换

- 复合赋值

```
a+=f(a,b,ref c,out d)
```

```
=>
```

```
a = a + (function() local r; r,c,d = f(a,b,c); return r; end)()
```

C#翻译到Lua

- 翻译变换

- 对象创建

```
var o = new Class(a, b) { m_F = 123, m_F2 = 456 }

=>

local o = (function() local obj; obj = Class(a, b); (function(self) self.m_F = 123; self.m_F2 = 456; end)(obj); return obj; end)()
```

C#翻译到Lua

- 翻译变换

- 循环中的continue实现

```
while(a < b)
{
    ++ a;
    if(a < 10)
        continue;
    if(a > 100)
        break;
    ++ a;
}
```



```
while a < b do
    local isBreak = false
repeat
    a = a + 1
    if a > 2 then
        isBreak = false
        break
    end
    if a > 100 then
        isBreak = true
        break
    end
    a = a + 1
until true
if isBreak then
    break
end
end
```

C#翻译到Lua

- 翻译变换

- switch中的break实现

```
switch(cond)
{
    case 1:
        if(a + b < 12)
            break;
        a += 2;
        b += 4;
        c = a * b;
        break;
    default:
        c = 123;
        break;
    case 2:
        c = 456;
        break;
}
```



```
if cond == 1 then
repeat
    if a + b < 12 then
        break
    end
    a = a + 2
    b = b + 4
    c = a * b
    break
until true
elseif cond == 2 then
repeat
    c = 456
    break
until true
else
repeat
    c = 123
    break
until true
end
```

C#翻译到Lua

- 翻译变换

- Try-catch实现

```
try {
    return a + b;
}
catch(Exception e) {
    UnityEngine.Debug.LogFormat("{0}\n{1}", e.Message, e.StackTrace);
}
finally {
    UnityEngine.Debug.Log("finally");
}
```



```
local mret
local tryret, tryretval
tryret, tryretval = luatry(function()
    mret = a + b
    return 1
end)
if tryret then
    UnityEngine.Debug.Log("finally")
    if tryretval == 1 then
        return mret
    end
end
local handled
handled = false
luacatch(handled, tryretval, (not tryret) and function(e)
    handled = true
    UnityEngine.Debug.Log("{0}\n{1}", e.Message, e.StackTrace)
end)
if not tryret then
    UnityEngine.Debug.Log("finally")
end
```

C#翻译到Lua

• 属性控制与定制翻译

- Cs2Lua.Entry属性

用于指明某个c#类的对应lua文件要生成一个入口方法，具体实现在lualib.lua里的defineentry函数，生成代码会调用defineentry。这个主要用于提供Main函数入口。

- Cs2Lua.Export属性

用于指明某个c#类的构造在转换为lua后生成的供c#端调用的__new_object方法里用于对象构造。

- Cs2Lua.EnableInherit属性

用于允许某个c#类使用继承（此时转换到lua时不会报错），Cs2Lua会采用一种继承实现机制来实现继承，但与c#的继承语义不太一样，此属性用于能确保使用一致继承的情形（就是说语义与c#是一致的），一般继承层次只有2层并且只涉及复用代码与纯虚函数重载的情形是可以的，子类隐藏父类非虚函数的情形要避免使用。

C#翻译到Lua

- 属性控制与定制翻译

- Cs2Lua.Require(string luaModuleName) 属性

用于指出c#代码在转换后需要require某个lua模块，比如lualib里的文件就需要在Program类上标记require。往往用于手写部分lua代码的情形。

```
[Cs2Ds1.Require("lualib_valuetypescript", "lualib_arithmetic", "lualib_container", "lualib_delegate",
    "lualib_linq", "lualib_object", "lualib_event", "cs2luanetworkmessagepools", "gameeventlua")]
[Cs2Ds1.Entry]
public static class Program
{
    [Cs2Ds1.Ignore]
    internal static void InitDll()
    {
        PluginManager.Instance.RegisterObjectFactory("GameController", new ObjectPluginFactory<GameController>());
        PluginManager.Instance.RegisterObjectFactory("Cs2LuaModuleProxy", new ObjectPluginFactory<Cs2LuaModuleProxy>());
        PluginManager.Instance.RegisterObjectFactory("Cs2LuaUiControllerManager", new ObjectPluginFactory<Cs2LuaUiControllerManager>());
    }
}
```

C#翻译到Lua

- 属性控制与定制翻译

- Cs2Lua.Ignore属性

用于在C#代码里标记类或方法不进行翻译处理，使用它们的代码就像它们已经存在一样转换。（这一属性主要用于手动用lua实现一些C#代码的功能）

```
[Cs2Ds1.Ignore]
internal static class Cs2LuaLibrary
{
    internal static bool IsCs2Lua(object o)
    {
        return false;
    }
    internal static string FormatString(string fmt, params object[] args)
    {
        return string.Format(fmt, args);
    }
}
```

C#翻译到Lua

- 属性控制与定制翻译

- Cs2Lua.TranslateTo(string luaModuleName, string targetMethodName) 属性

用于指定某个方法的实现翻译为调用指定lua模块的指定lua函数（要求目标lua函数签名与方法一致，用于手动翻译某个lua方法的情形）。

```
internal static class Cs2LuaType
{
    [Cs2Dsl.TranslateTo("Cs2LuaTypeImpl", "Cs2LuaTypeImpl.GetFullName")]
    internal static string GetFullName(System.Type type)
    {
        return type.AssemblyQualifiedName;
    }
}
```

C#翻译到Lua

- 属性控制与定制翻译
 - Cs2Lua.NeedFuncInfo 属性

用于指定某个方法需要生成函数信息，函数信息是值类型模拟或在函数退出前要进行处理所需要的一个机制。

C#翻译到Lua

- 属性控制与定制翻译

- __DSL__宏
 - Cs2Lua在翻译时会定义这个宏，c#代码里可以使用这个宏来控制哪些代码被翻译到lua或相反
 - 这个宏主要不是为了定制翻译，而是允许在c#代码里针对翻译到Lua与不翻译到Lua写有区别的逻辑
 - 一般来说支撑Cs2Lua的工具类或底层类会需要用到

C#翻译到Lua

• 代码生成配置

- cs2lua.dsl
 - 配置indexer方法是否可翻译为下标访问以及下标是否加1
 - 主要用于使用lua数组实现的c#集合类型
 - 配置多个类的翻译结果合并到一个输出文件
 - 主要用于消息或表格读表器代码的合并
 - 消息有1000多个，表格有500多个
 - 配置为指定函数生成prologue/epilogue代码
 - 主要用于调试与性能分析
- rewriter.dsl
 - 配置翻译允许的泛型类、泛型方法、泛型参数、扩展方法等
 - 标准dotnet的集合类List、Dictionary、HashSet等泛型类型在Cs2Lua里用作内建容器类
 - 配置翻译不允许的类、方法、特性、字段
 - Slua封装API需要生成相应代码，这些封装代码在iOS上是无法热更新的，上线后需要对未导出lua API的unity API限制使用

C#翻译到Lua

• 定制翻译脚本

- Cs2Lua第一阶段翻译会将非结构类语法（类、函数、复合语句）如操作符、特性访问、函数调用等代码翻译为dsl函数调用的形式，第二阶段可以对这些函数调用进行转换，默认是转换为lua的表达式语法或无法精确对应的转换为对lualib里的函数调用
- Cs2Lua允许在generator.dsl脚本里修改默认的转换到lua的行为，一般是对特定类型相关的处理，可以通过定制翻译得到一个性能更合适的版本。我们在值类型的处理里主要借助了这个机制，对特定的值类型或返回值类型的API进行特殊转换，从而避免GC，降低内存并提升性能

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—位运算与条件表达式

```
lshift(v, n)
rshift(v, n)
bitnot(v)
bitand(v1, v2)
bitor(v1, v2)
bitxor(v1, v2)

simplecondexp(cv, tv, fv)
condexp(cv, tc, tf, fc, ff)
condaccess(v, func)
nullcoalescing(v, func)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—类型转换、常量、字符串、CustomData等

```
wrapconst(t, name)
wrapchar(char, intValue)
typecast(obj, t, tk)
typeas(obj, t, tk)
typeis(obj, t, tk)
isequal(v1, v2)
stringisequal(v1, v2)
stringconcat(str1, str2)
cssrtoluastr(str)
luastrtocssstr(str)
luatoobject(symKind, isStatic, symName, arg1, ...)
objecttolua(arg1, ...)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—异常处理、操作符、基础类型方法等

```
luausing(func, ...)
luatry(func, ...)
luacatch(handled, err, func)
luathrow(obj)

invokeoperator(rettype, class, method, ...)
invokeexternoperator(rettype, class, method, ...)
callbasicvalue(obj, isEnum, class, method, ...)
getbasicvalue(obj, isEnum, class, property)
setbasicvalue(obj, isEnum, class, property, value)
callarraystaticmethod(firstArray, secondArray, method, ...)
invokeintegeroperator(op, luaop, opd1, opd2, type1, type2)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—扩展方法与indexer

```
callexternextension(callerClass, method, ...)  
getexternstaticindexer(callerClass, class, name, argCount, ...)  
getexterninstanceindexer(callerClass, obj, class, name, argCount, ...)  
setexternstaticindexer(callerClass, class, name, argCount, toplevel, ...)  
setexterninstanceindexer(callerClass, obj, class, name, argCount, toplevel, ...)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—基于接口调用

```
callexterninterfacereturnstruct(obj, intf, method, ...)  
getexterninterfacestructmember(obj, intf, name, getmethodname)  
callinterface(obj, intf, method, ...)  
getinterface(obj, intf, name, getmethodname)  
setinterface(obj, intf, name, setmethodname, val)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—delegation

```
wrapdelegation(handlers)
delegationwrap(handler)
delegationcomparewithnil(isstatic, t, k, symKind, beequal)
delegationset(isstatic, t, k, symKind, handler)
delegationadd(isstatic, t, k, symKind, handler)
delegationremove(isstatic, t, k, symKind, handler)
externdelegationcomparewithnil(isstatic, t, k, symKind, beequal)
externdelegationset(isstatic, t, k, symKind, handler)
externdelegationadd(isstatic, t, k, symKind, handler)
externdelegationremove(isstatic, t, k, symKind, handler)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—容器与迭代器

```
wraparray(arr, size, classObj, typeKind)
newarraydim0(classObj, typeKind, defVal)
newarraydim1(classObj, typeKind, defVal, size1)
newarraydim2(classObj, typeKind, defVal, size1, size2)
newarraydim3(classObj, typeKind, defVal, size1, size2, size3)
wrapanonymousobject(dict)
wrapparams(arr, elementType, elementTypeKind)
newdictionary(t, typeargs, typekinds, ctor, dict, ...)
newlist(t, typeargs, typekinds, ctor, list, ...)
newcollection(t, typeargs, typekinds, ctor, coll, ...)
newexterndictionary(t, typeargs, typekinds, ctor, dict, ...)
newexternlist(t, typeargs, typekinds, ctor, list, ...)
newexterncollection(t, typeargs, typekinds, ctor, coll, ...)

newiterator(funcInfo, exp)
getiterator(iterInfo)
recycleiterator(funcInfo, iterInfo)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—对象定义与实例创建

```
defineclass(base, fullName, typeName, class, obj_methods, obj_build, is_value_type)
buildbaseobj(obj, class, baseClass, baseCtor, ...)
builddexternbaseobj(obj, class, baseClass, baseCtor, ...)
defineentry(class)

newstruct(class, typeargs, typekinds, ctor, initializer, ...)
newexternstruct(class, typeargs, typekinds, ctor, initializer, ...)
newobject(class, typeargs, typekinds, ctor, initializer, ...)
newexternobject(class, typeargs, typekinds, ctor, initializer, ...)
newtypeparamobject(t)
defaultvalue(t, typename, isExtern)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—协程方法与LINQ

```
wrapenumerable(func)
wrapyield(yieldVal, isEnumerableOrEnumerator, isUnityYield)
LINQ.exec(lua_table_from_cs_linq)
```

C#翻译到Lua

- 在第二阶段或委托到lua运行时的处理—值类型处理

```
wrapoutstruct(v, classObj)
wrapoutexternstruct(v, classObj)
wrapstruct(v, classObj)
wrapexternstruct(v, classObj)
wrapstructargument(arr, argType, argOperKind, argSymKind, class, callerClass)
wrapexternstructargument(arr, argType, argOperKind, argSymKind, class, callerClass)
wrapstructarguments(arr, argType, argOperKind, argSymKind, class, callerClass)
wrapexternstructarguments(arr, argType, argOperKind, argSymKind, class, callerClass)
getexternstaticstructmember(symKind, class, member)
getexterninstancestructmember(symKind, obj, class, member)
callexterndelegationreturnstruct(funcobj, funcobjname, ...)
callexternextensionreturnstruct(class, member, ...)
callexternstaticreturnstruct(class, member, ...)
callexterninstancereturnstruct(obj, class, member, ...)
getstaticindexerstruct(isExtern, elementType, callerClass, class, name, argCount, ...)
getinstanceindexerstruct(isExtern, elementType, callerClass, obj, class, name, argCount, ...)
setstaticindexerstruct(isExtern, elementType, callerClass, class, name, argCount, ...)
setinstanceindexerstruct(isExtern, elementType, callerClass, obj, class, name, argCount, ...)
arraygetstruct(arrIsExtern, arrSymKind, elementType, arr, argCount, ...)
arraysetstruct(arrIsExtern, arrSymKind, elementType, arr, argCount, ...)
invokeexternoperatorreturnstruct(rettype, class, method, ...)

recycleandkeepstructvalue(fieldType, oldVal, newVal)
luainititalize()
luafinalize(funcInfo)
```

主要內容

➤ 第五章 lua运行时

- 1、交叉引用处理
- 2、class的模拟
- 3、容器迭代器
- 4、delegate
- 5、异常处理
- 6、Unity协程支持

Lua运行时

● 交叉引用处理

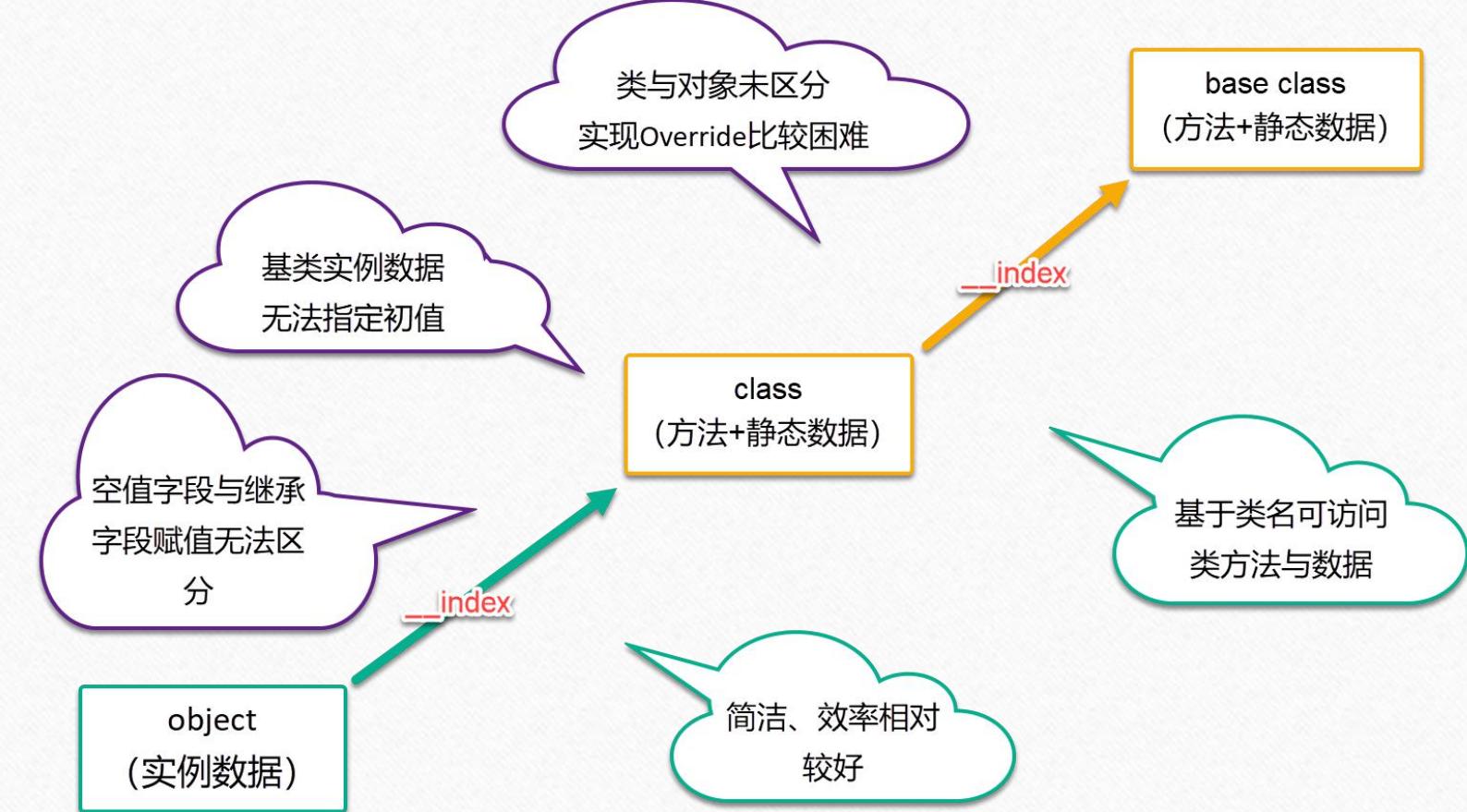
- 主要策略—延迟绑定、惰性执行
 - 类加载时避免引用其它类的代码被执行
 - 纯结构性代码
 - 初始类全部关联临时元表
 - 临时元表的元方法（`__call`、`__index`、`__newindex`）首先将类元表置为空，然后调用类定义方法，构造类的正式结构与设置正式元表，最后再次发起触发元方法的操作
 - 类似于jit的机制，在第一次执行时编译并修改函数入口到native入口

```
function settempmetatable(class)
    setmetatable(
        class,
    {
        index = function(tb, key)
            setmetatable(class, nil)
            class.__define_class()
            return tb[key]
        end,
        newindex = function(tb, key, val)
            setmetatable(class, nil)
            class.__define_class()
            tb[key] = val
        end,
        call = function(...)
            setmetatable(class, nil)
            class.__define_class()
            return class...
        end
    })
    rawset(class, "__cs2lua_predefined", true)
end
```

Lua运行时

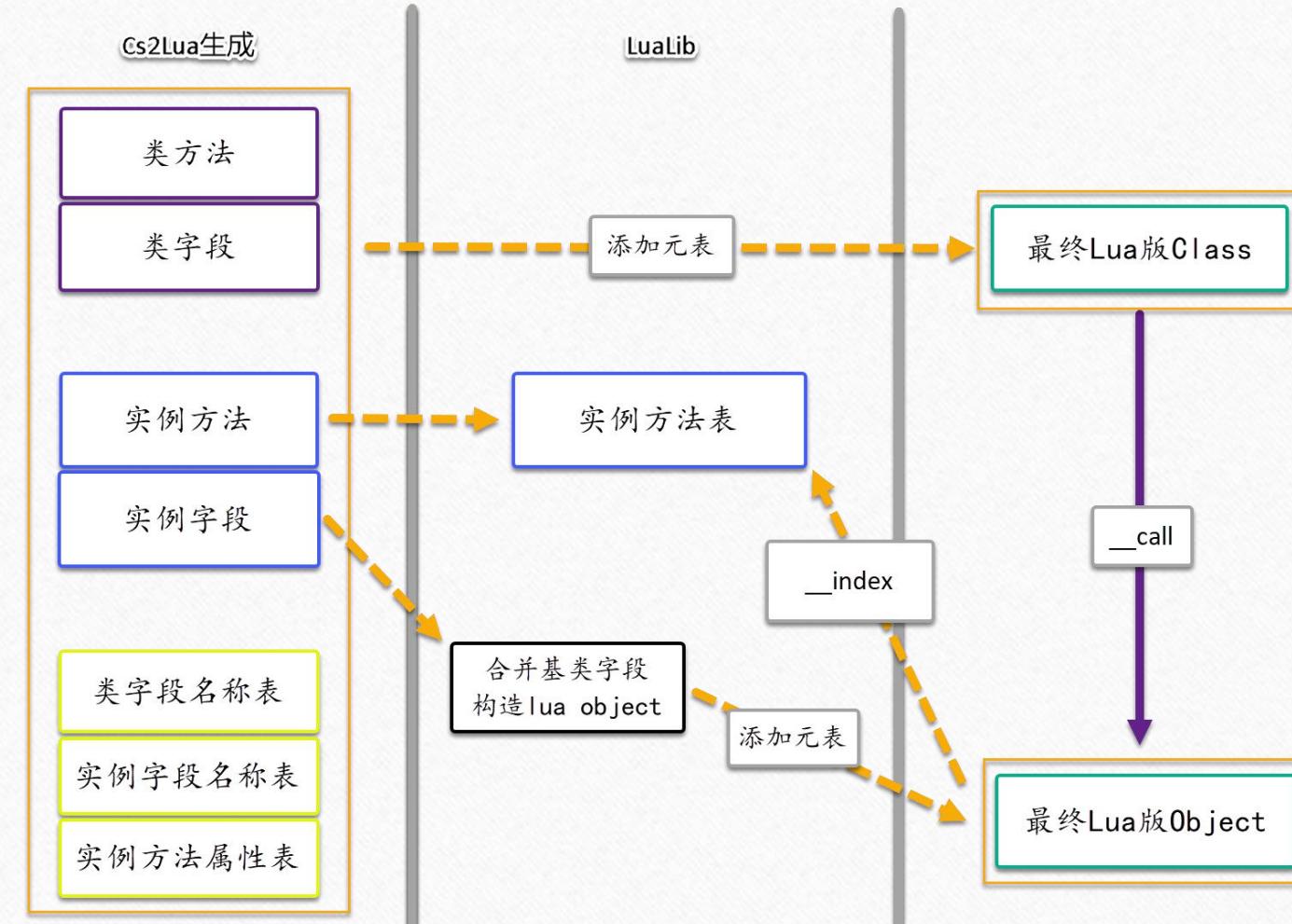
- 手写Lua常用对象模型

- lua作者推荐
- 基于原型的对象模型，类似 javascript
- 不能严格表达面向对象特征



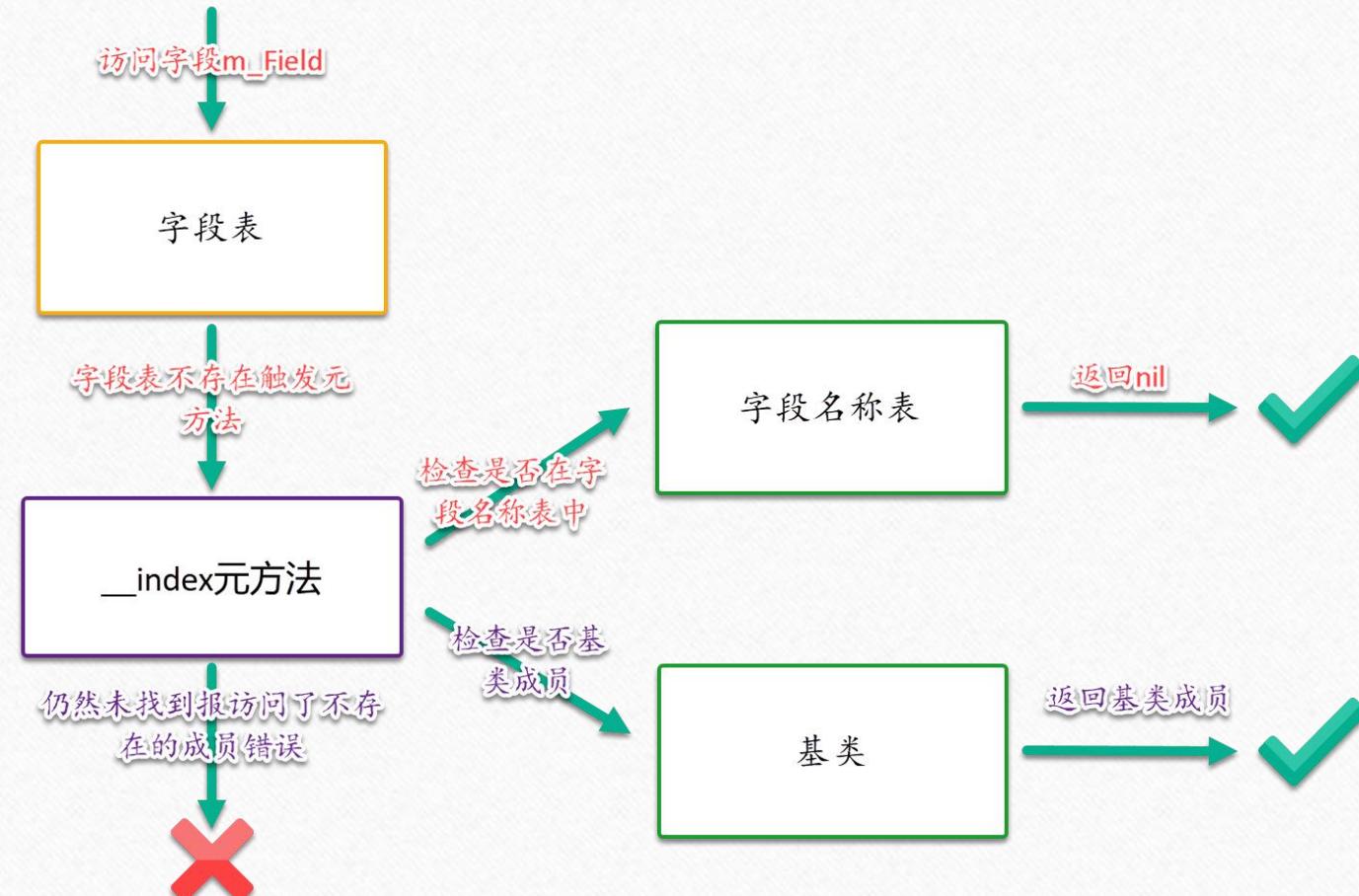
Lua运行时

- class的模拟
 - 翻译生成素材
 - 类table，包含
 - 类方法
 - 类字段
 - 实例方法
 - 实例字段
 - 只读的类信息表，用于运行时判断LuaLib里的
 - defineclass负责将素材转变成最终class
 - 主要工作是合并基类数据、构造与设置元表



Lua运行时

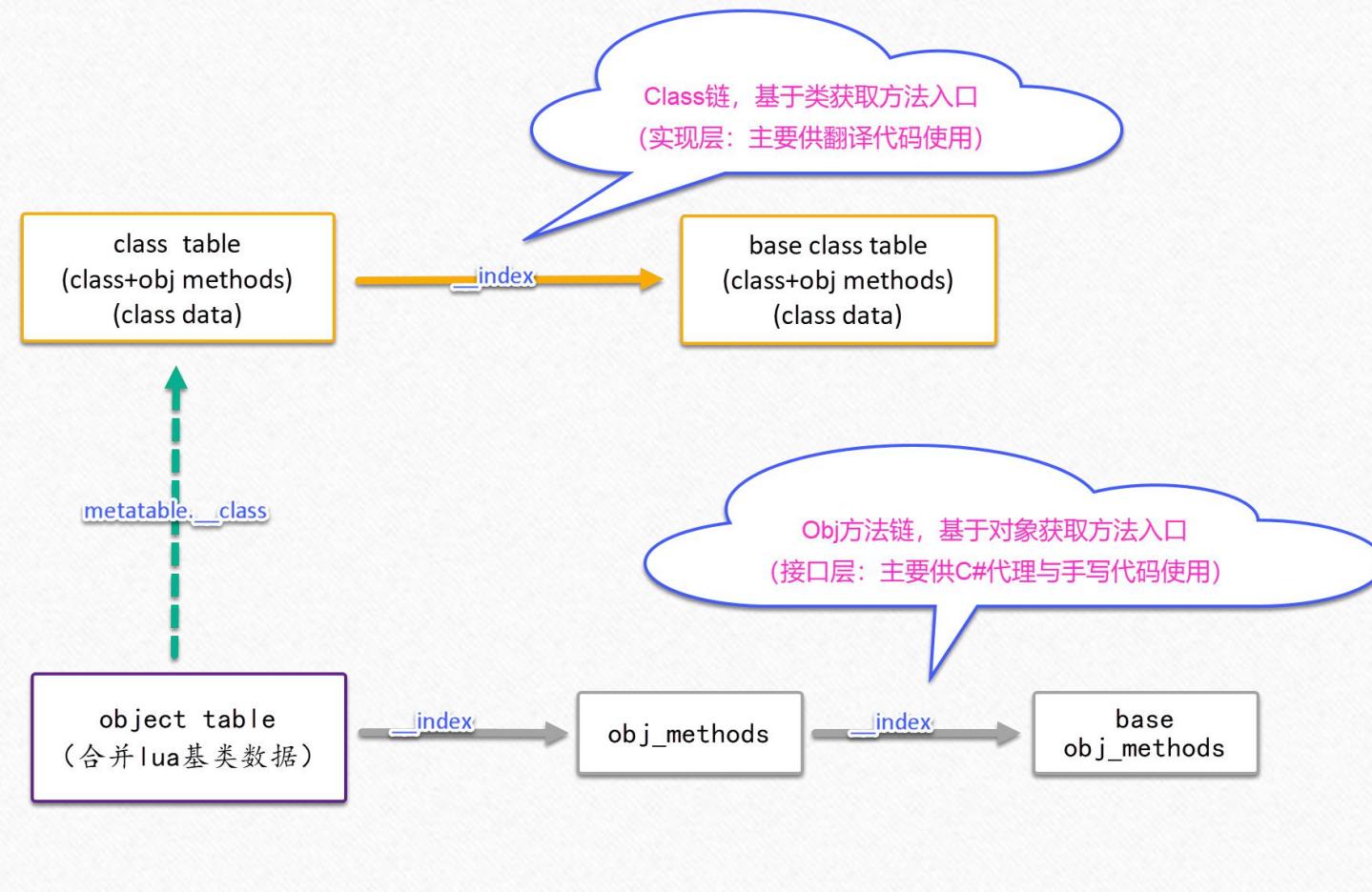
- class的模拟
 - 字段名称表的作用
 - Lua table在保存key/value时，如果值赋为nil，表示删除这个key/value
 - 那么一个类的某个字段为nil如何表示呢？
 - 访问不存在的字段时会触发元方法__index，但是如何知道是一次错误的访问，还是访问值为nil的字段呢
 - 一般__index里还用来模拟继承，此时通常是查询基类是否有这个名字的成员，这也会引起混淆



Lua运行时

- class的模拟

- C#对象按偏移访问数据
- Lua对象按名称访问数据
 - 不能使用基类子对象
 - 任何时候this都是当前对象
 - 不能有重名数据
 - 不能有重名方法 (new修饰)
- 继承
 - lua基类数据合并到子类对象
 - 无lua基类子对象
 - C#基类创建基类子对象
 - c#继承无法实现虚函数机制



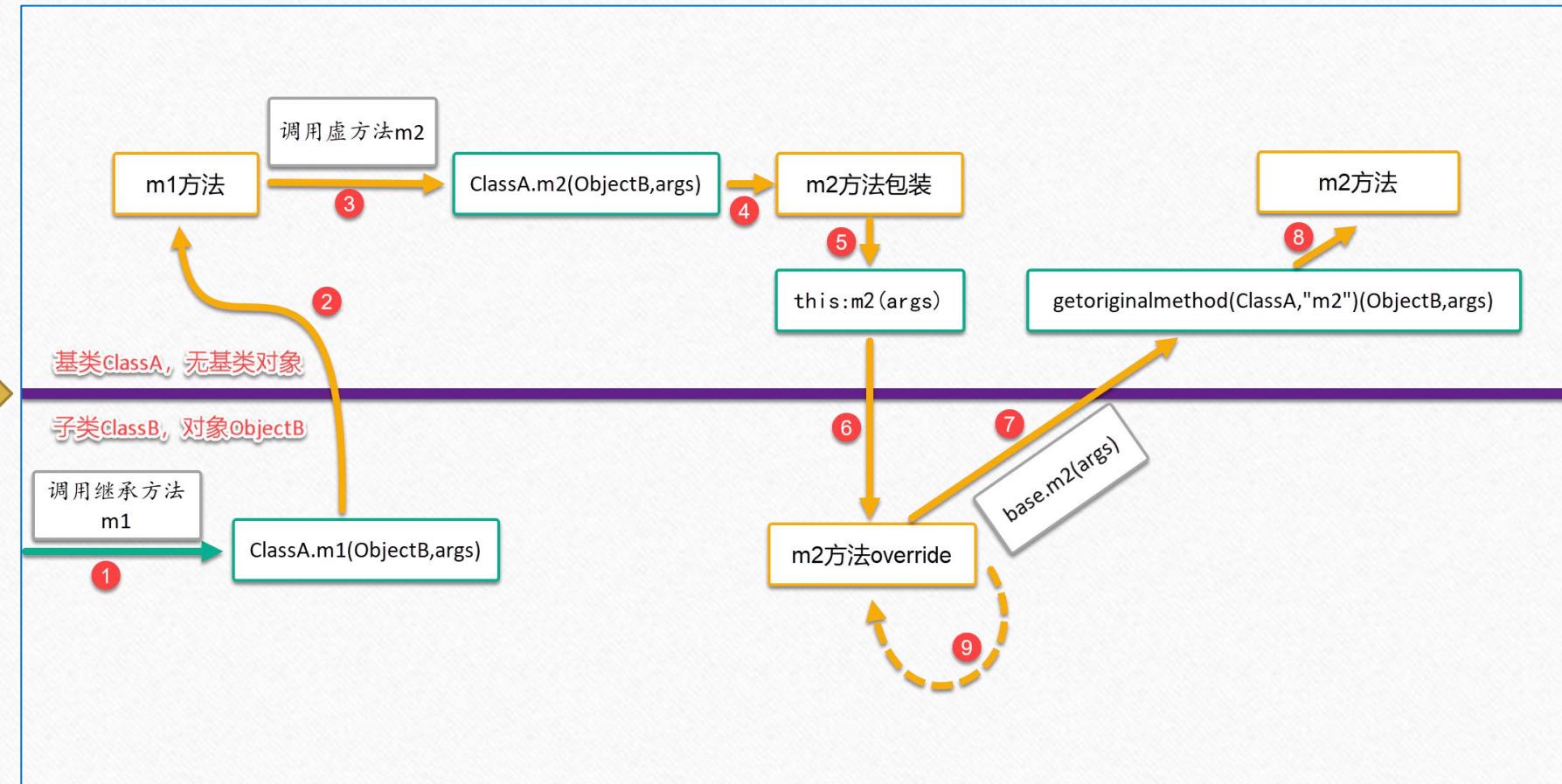
Lua运行时

- class的模拟
 - 虚函数语义实现

```
class ClassA
{
    public void m1()
    {
        m2();
    }
    public virtual void m2()
    {
        ...
    }
}

class ClassB
{
    public override void m2()
    {
        base.m2();
        ...
    }
}

var obj = new ClassB();
obj.m1();
```



Lua运行时

- 容器迭代器

- 数组与 IList 的 foreach 翻译为按下标访问的 for 语句
- 字典的 foreach 语句翻译为 Lua 的 for … in 语句， C# 里是 Ienumerator 接口， lua 是一个迭代器函数，这里需要进行适配
- 由于迭代器函数每次循环时要创建，这里会产生 GC
 - 通过在字典对象上记录迭代器实例来重用迭代器函数
 - 另外迭代器函数重用需要可以重置迭代到初始位置

Lua运行时

- 容器迭代器

- 由于迭代器函数每次循环时要创建，这里会产生GC

- 翻译时借助3个函数完成分配、使用与回收
 - newiterator
 - getiterator
 - Recycleiterator

```
local _foreach_415_8_418_9 = newiterator(__cs2lua_func_info, this.mMasterFollowComDic, Dictionary, Dictionary, true);
for pair in getiterator(_foreach_415_8_418_9) do
    if pair == __cs2lua_nil then
        pair = nil;
    end;
    pair.Value:Tick();
end;
recycleiterator(__cs2lua_func_info, _foreach_415_8_418_9);
```

- 实际分配与回收的是一个table，{iter, enumer, dict}，迭代器函数是其中一个元素，其它元素用来重置迭代器到初始位置和关联迭代器对象池

Lua运行时

容器迭代器

- Lua迭代器与c#迭代器还有一个语义差异是Lua迭代器返回nil就代表结束，这样如果容器里有空值就会提前迭代结束
 - 定义一个__Cs2Lua_nil变量来表示容器里的nil元素，然后在for…in里判断恢复成nil即可

```
local meta = getmetatable(exp)
if meta and rawget(meta, "__cs2lua_defined") then
    if meta.cachedIterers and meta.cachedIterers[exp] and #(meta.cachedIterers[exp])>0 then
        local iterInfo = table.remove(meta.cachedIterers[exp], 1)
        iterInfo[2]:Reset()
        if funcInfo then
            table.insert(funcInfo.iter_list, iterInfo)
        end
        return iterInfo
    else
        local enumerator = exp:GetEnumerator()
        local f = function()
            if enumerator:MoveNext() then
                local v = enumerator.Current
                if v == nil then
                    v = __cs2lua_nil
                end
                return v
            else
                return nil
            end
        end
        local iterInfo = {f, enumerator, exp}
        if funcInfo then
            table.insert(funcInfo.iter_list, iterInfo)
        end
        return iterInfo
    end
end
end
```

重置迭代器

Lua迭代器函数与闭包引用C#迭代器

迭代器信息三元组

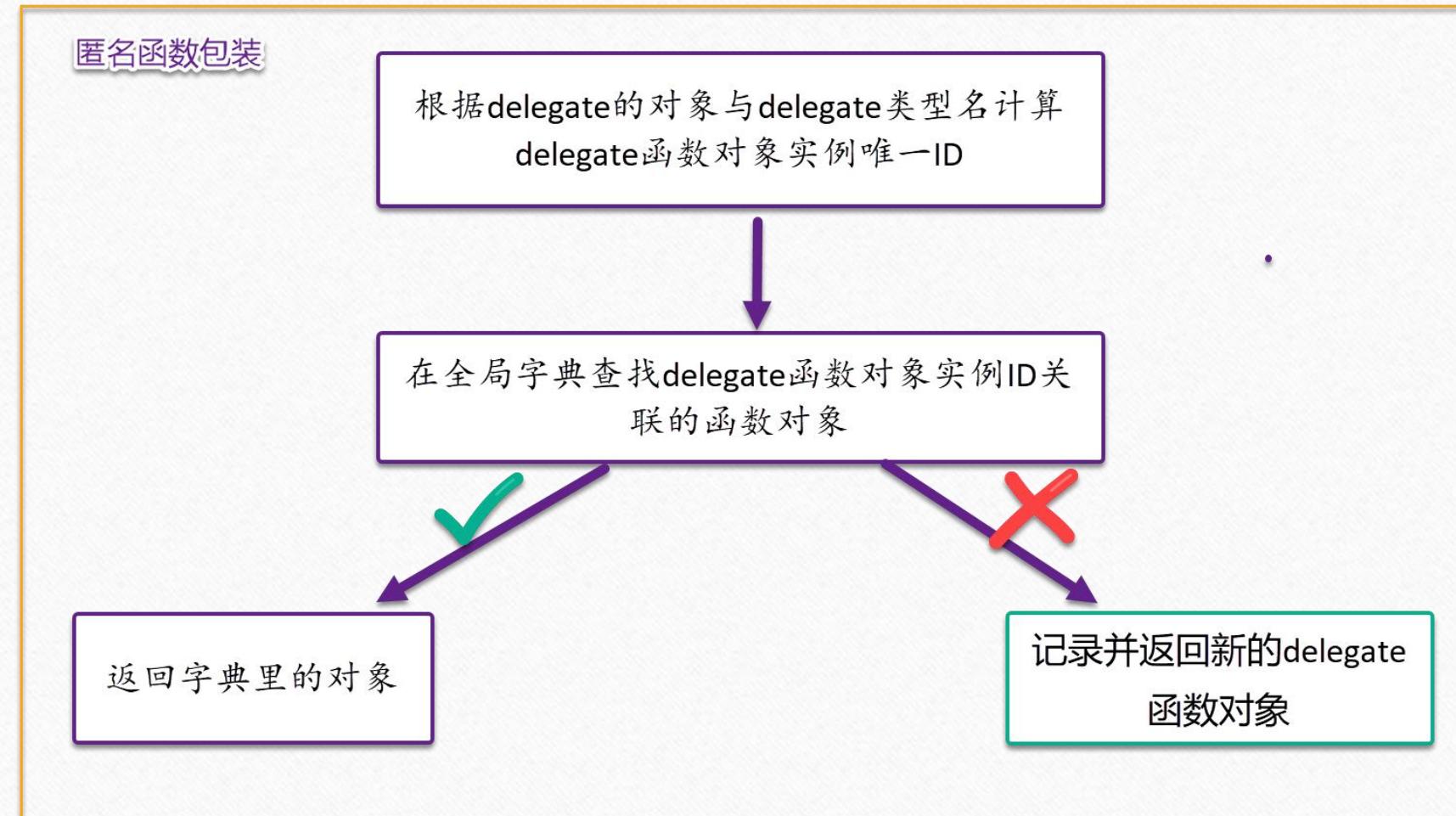
Lua运行时

- delegate

- delegate在Lua里是函数对象，与c#语义上的差异主要在移除delegate时，c#能根据delegate关联的对象与方法来判断应该移除哪个delegate，但lua函数对象每次生成都会是新的
- Cs2Lua采取一些小技巧来保证同样的delegate包装代码在多个地方执行返回同一个函数对象实例，这样在翻译时delegate的安装与移除就可以采用一致的样式（因为移除时参数可能是一个变量，其值才是delegate，实际上在翻译时无法识别要移除的delegate是哪一个）
- 思路
 - 将delegate的函数对象再包装在一层匿名函数代码内，然后执行匿名函数来返回delegate函数对象
 - 这样匿名函数代码里就可以实现不同地方返回同一个delegate函数对象的目的

Lua运行时

- delegate
 - 过程



Lua运行时

- delegate

- 实现

```
local delegateObj;
delegateObj = (function()
    local obj = this;
    local funcKey = calcdelegationkey("ActionWithMissionInfo:Invoke", obj);
    return builddelegationonce(funcKey, getdelegation(funcKey) or function...
        return obj:Invoke(...);
    end);
end)();
```

- 条件运算短路机制避免不必要的函数对象创建

- getdelegation(funcKey)

- 在全局表里检查是否存在函数对象是则返回，否则返回nil

- builddelegationonce(funcKey, funcObj)

- 如果全局表里存在funcKey关联的函数对象就返回该对象
 - 否则返回funcObj

- or条件表达式仅当getdelegation(funcKey)返回nil才会对后续函数对象代码进行计算

Lua运行时

● 异常处理

- luatry+匿名函数对象
 - 函数对象存在GC
- 只支持一个catch，且使用Exception类型捕获
- luacatch也使用函数对象，为避免在未发生异常时也产生GC，利用了条件表达式短路机制
 - 类似delegation的短路机制，但这里需要使用and条件，让luacatch可以根据传入的参数是否有效来判断是否执行catch代码

```
try {
    return a + b;
}
catch(Exception e) {
    UnityEngine.Debug.LogFormat("{0}\n{1}", e.Message, e.StackTrace);
}
finally {
    UnityEngine.Debug.Log("finally");
}

local mret
local tryret, tryretval
tryret, tryretval = luatry(function()
    mret = a + b
    return 1
end)
if tryret then
    UnityEngine.Debug.Log("finally")
    if tryretval == 1 then
        return mret
    end
end
local handled
handled = false
luacatch(handled, tryretval, (not tryret) and function(e)
    handled = true
    UnityEngine.Debug.Log("{0}\n{1}", e.Message, e.StackTrace)
end)
if not tryret then
    UnityEngine.Debug.Log("finally")
end
```

Lua运行时

- Unity协程支持
 - Xlua的协程实现比较巧妙，我们也借鉴过来，现在默认采用此方式
 - 利用lua coroutine yield/resume的特点
 - 每次MoveNext就调用一次resume，这样执行到yield是一次迭代
 - StopCoroutine会让coroutine保持在挂起状态，但看来没有什么影响，lua可以正常处理协程GC

主要内容

➤ 第六章 值类型处理

- 1、c#里的struct特性与作用
- 2、lua不支持struct，普通类与struct的差异
- 3、GC问题
- 4、解决方案

值类型处理

• C#里的struct特性与作用

- 栈上分配，每个变量占用独立的空间
- 赋值与传参是数据的深拷贝（除引用类型成员外）
- 每次赋值都产生一个新的值类型对象
- 不会导致GC

C# struct

栈上分配

赋值与传参产生新实例

无GC

值类型通常可提升性能

Lua 模拟

堆上分配

赋值与传参产生新实例

每个新实例都需要GC

GC性能影响

所以在C#里值类型是一种减少GC提高性能的途径

值类型处理

- lua不支持struct，普通类与struct的差异
 - lua里使用table模拟对象，语义上是引用类型
 - 在堆上分配，会导致GC
 - 要模拟c#值类型语义，赋值与传参时需要构造新对象并深拷贝原对象数据
 - 深拷贝没有很好的通用实现办法

C# struct

栈上分配

赋值与传参产生新实例

无GC

值类型通常可提升性能

Lua 模拟

堆上分配

赋值与传参产生新实例

每个新实例都需要GC

GC性能影响

所以在Lua里模拟值类型是一种有很多GC并且不利于性能的机制

我们约定Cs2LuaScript工程里不要定义新的值类型

但是可能需要用到c#里的值类型，或者c# API的参数或返回值会用到值类型

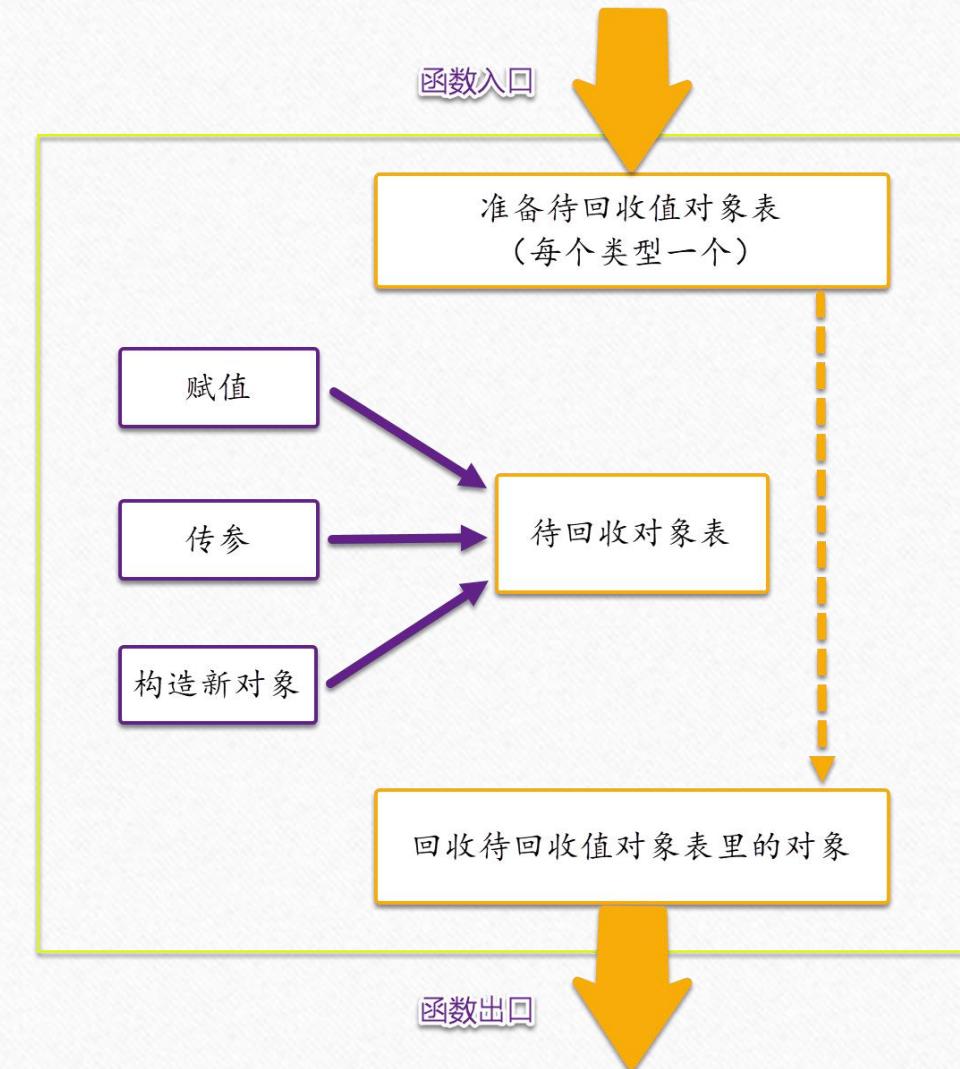
值类型处理

- GC问题
 1. Lua里模拟值类型赋值操作与传参时需要构建新的对象
 2. c#里的Api的返回值如果是值类型，每次调用都会产生一个lua里的对象包装

值类型处理

• 解决方案

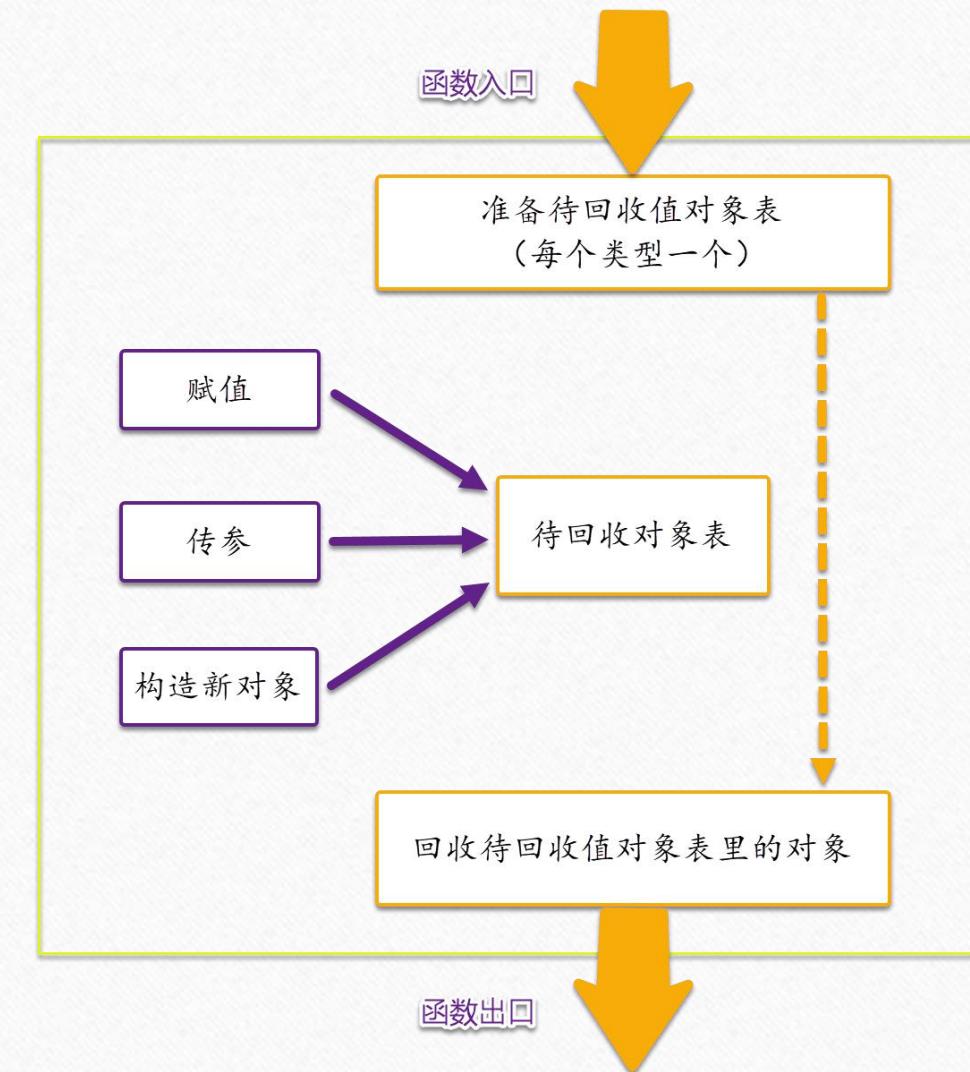
- Lua模拟值类型赋值与传参时构建新对象
 - 在函数进入时为每个值类型准备一个待回收记录表
 - 分配新的值类型对象时记录到待回收记录表
 - 函数返回前回收记录表里的值类型对象



值类型处理

解决方案

- Lua模拟值类型赋值与传参时构建新对象
 - 值类型返回值在返回前从当前函数的待回收对象表移到调用方函数的待回收对象表中
 - 返回语句转换时识别处理
 - 被lambda函数捕获的变量不能回收
 - 基于lambda函数数据流分析获取捕获变量
 - 每次对捕获变量赋值后处理
 - 对象的值类型字段不能回收
 - 赋值给数组或集合的值类型不能回收
 - 定制赋值处理



值类型处理

• 解决方案

- Lua调用c#的API如果返回值是值类型，slua在API包装代码里每次都会创建一个新的关联了值类型元表的userdata
 - 这些需要在c#端为这些API编写一个非值类型返回值的版本，比如使用out参数分别返回值类型的数据成员
 - 然后在generator.dsl里定制这些API的翻译，改为调用非值类型返回值版本，并重新构造值类型对象（使用对象池并记录到待回收对象表中）

值类型处理

● 解决方案

```
script(callexterninstancereturnstruct)args($funcData, $funcOpts, $sb, $indent)
{
    //callexterninstancereturnstruct(obj, class, member, ...)
    $class = getargument($funcData, 1);
    $member = getargument($funcData, 2);

    if($class=="UnityEngine.Camera"){
        if($member=="ViewportPointToRay__Vector3"){
            usefunc("call_camera_viewportpointtoray_ray_v3","(funcInfo, obj, ...)", $funcData, $funcOpts, $sb, $indent, [1,2], "__cs2lua_func_info")
            {
                local pt,eye = ...
                local _,ox,oy,oz,dx,dy,dz = Utility.ViewportPointToRayV3(obj, pt, Slua.out, Slua.out, Slua.out, Slua.out, Slua.out, Slua.out)
                local ori = UnityEngine.Vector3.New(ox,oy,oz)
                local dir = UnityEngine.Vector3.New(dx,dy,dz)
                table.insert(funcInfo.v3_list, ori)
                table.insert(funcInfo.v3_list, dir)
                return UnityEngine.Ray.ctor__Vector3__Vector3(ori,dir)
            };
            return(true);
        }
    }
}
```

值类型处理

• 解决方案

- 前面提到的值类型操作，都需要在generator.dsl里为每类值类型操作针对每个具体的值类型进行定制翻译

- Out参数
- 复制新对象
- 值类型传参
- 外部值类型特性
- 外部值类型返回值
- 值类型indexer
- 值类型数组

```
wrapoutstruct(v, classObj)
wrapoutexternstruct(v, classObj)
wrapstruct(v, classObj)
wrapexternstruct(v, classObj)
wrapstructargument(arr, argType, argOperKind, argSymKind, class, callerClass)
wrapexternstructargument(arr, argType, argOperKind, argSymKind, class, callerClass)
wrapstructarguments(arr, argType, argOperKind, argSymKind, class, callerClass)
wrapexternstructarguments(arr, argType, argOperKind, argSymKind, class, callerClass)
getexternstaticstructmember(symKind, class, member)
getexterninstancestructmember(symKind, obj, class, member)
calleeexterndelegationreturnstruct(funcobj, funcobjname, ...)
calleeexternextensionreturnstruct(class, member, ...)
calleeexternstaticreturnstruct(class, member, ...)
calleeexterninstancereturnstruct(obj, class, member, ...)
getstaticindexerstruct(isExtern, elementType, callerClass, class, name, argCount, ...)
getinstanceindexerstruct(isExtern, elementType, callerClass, obj, class, name, argCount, ...)
setstaticindexerstruct(isExtern, elementType, callerClass, class, name, argCount, ...)
setinstanceindexerstruct(isExtern, elementType, callerClass, obj, class, name, argCount, ...)
arraygetstruct(arrIsExtern, arrSymKind, elementType, arr, argCount, ...)
arraysetstruct(arrIsExtern, arrSymKind, elementType, arr, argCount, ...)
invokeexternoperatorreturnstruct(rettype, class, method, ...)

recycleandkeepstructvalue(fieldType, oldVal, newVal)
luainititalize()
luafinalize(funcInfo)
```

值类型处理

- 解决方案
 - 复制新对象的定制翻译脚本 (DSL)

```
script(wrapexternstruct)args($funcData, $funcOpts, $sb, $indent)
{
    //wrapexternstruct(v, classObj)
    $varName = getargument($funcData, 0);
    $classObj = getargument($funcData, 1);

    if($classObj=="UnityEngine.Vector2"){
        usefunc("wrap_vector2","(funcInfo, v)", $funcData, $funcOpts, $sb, $indent, [1], "__cs2lua_func_info")
        {
            local obj = UnityEngine.Vector2.New(v.x,v.y)
            table.insert(funcInfo.v2_list, obj)
            return obj
        };
        return(true);
    }
    elseif($classObj=="UnityEngine.Vector3"){
        usefunc("wrap_vector3","(funcInfo, v)", $funcData, $funcOpts, $sb, $indent, [1], "__cs2lua_func_info")
        {
            local obj = UnityEngine.Vector3.New(v.x,v.y,v.z)
            table.insert(funcInfo.v3_list, obj)
            return obj
        };
        return(true);
    }
}
```

主要內容

➤第七章 性能问题与优化策略

- 1、函数对象
- 2、表
- 3、字符串
- 4、异常处理与using语句
- 5、柔性可用

性能问题与优化策略

• 函数对象

- lua里每个函数对象至少占20字节
- 函数对象在Cs2Lua里主要用在各种表达式与语句的翻译里，通常是复合赋值或带有ref/out参数的函数调用需要使用函数对象包装
- Cs2Lua在第二步翻译时移除了部分简单用法下的函数对象（见后面），其它情形下的函数对象，需要在性能分析发现热点时通过手动拆分表达式来处理
- 在生成的lua代码里搜索 “(function())” 可以找到翻译使用的匿名函数包装，除delegate外，其它都可以通过拆分表达式消除（非性能热点可以不处理）
- 在Tick/Update等方法里的函数对象应该尽量消除（可以通过luaprofiler分析GC来初步锁定范围）

性能问题与优化策略

• 表

- Lua里每个空表至少占40字节
- 表主要用在对象模拟上
- 减少表的数量对内存影响很大
 - 数据表
 - 每一行是一个对象，有些表行数上万
 - 我们前端表格多达500多张
 - 表格数据放在lua里内存占用很高
 - Chief做了2个优化
 - 优化一：表格对象模型采用原始lua table，去掉继承等特性支持（节省150MB）
 - 优化二：表格数据挪到c#端，lua端通过导出api访问（节省60MB）
- 集合类的构造操作会使用lua table来记录泛型参数信息，Cs2Lua采用与delegation类似的条件表达式短路来避免每次执行都创建lua table

性能问题与优化策略

• 字符串

- Lua里的字符串传到c# API中时，在c#里会构造一个新的c# string实例，这会产生比较大的GC
- 优化
 - 常量字符串添加lua/c#对应表
 - Cs2Lua翻译时会将调用外部API的常量字符串参数缓存到常量字符串表，luastrtocssstr(luastr)
 - 同时提供了一个反向的函数cssrtoluastr(csstr)，Cs2Lua目前没有用到，可能会在定制翻译里用到
- 在开发中应尽量避免跨lua/c#的字符串参数

性能问题与优化策略

- 异常处理

- 异常try块使用函数对象封装才能使用xpcall来执行，但函数对象每次执行都会构造一个新函数对象
- 优化
 - 将try语句块单独拆分到一个函数里
 - 根据数据流分析确定要捕获与返回的变量
 - 根据控制流分析决定是否要在函数结尾添加返回语句

```
AutoDo = function(this, delta)
local __method_ret_47_4_60_5;
local result;
result = false;
local __try_ret_50_8_58_9, __try_retval_50_8_58_9;
__try_ret_50_8_58_9, __try_retval_50_8_58_9, result = luatry(this.__try_func_50_8_58_9, this, delta, result);
ScriptTimeProfiler.EndFrameSample();
__method_ret_47_4_60_5 = result;
return __method_ret_47_4_60_5;
end,
__try_func_50_8_58_9 = function(this, delta, result)
ScriptTimeProfiler.BeginFrameSample(23);
result = this:AutoDo_Impl(delta);
return 0, result;
end,
```

性能问题与优化策略

- 柔性可用

- Cs2Lua在翻译时优先保证正确性，因此在很多语法翻译时采用了函数对象来保证在所有情形下语义一致
- 实际使用时，存在一些简单写法情形可以避免使用函数对象，Cs2Lua在翻译第二阶段对可能简化的情况进行移除函数对象的优化。
 - 类似TryGetValue这样带有out参数返回值又用于判断的，经常会在条件表达式里，这些函数调用在lua里不能直接出现在条件表达式中，必须包装成函数对象，但对于函数调用出现在表达式第一个的情形，函数调用提到条件表达式前面
 - if/while/do-while语句的条件部分可进行类似处理

```
if (m_PageView.TryGetValue(chengePage, out view) && view != null)
{
    view.visible = true;
}
```



```
local __invoke_1187_32_1187_76;
__invoke_1187_32_1187_76,view = this.m_PageView:TryGetValue(chengePage, __cs2lua_out);
if (__invoke_1187_32_1187_76 and (not isequal(view, nil))) then
    view:set_visible(true);
end
```

性能问题与优化策略

- 柔性可用

- 实际使用时，存在一些简单写法情形可以避免使用函数对象，我在翻译第二阶段对可能简化的情况进行移除函数对象的优化
 - 条件表达式如果结果是常量的，可以用一个lua函数来模拟

--tv与fv都是常量时使用

```
function simplecondexp(cv, tv, fv)
    if cv then
        return tv
    else
        return fv
    end
end
```

性能问题与优化策略

• 柔性可用

- 实际使用时，存在一些简单写法情形可以避免使用函数对象，我在翻译第二阶段对可能简化的情况进行移除函数对象的优化
 - 条件表达式是给变量或字段赋值的，可以转变成普通的if/else语句，然后在if/else里分别赋值

```
if (!isMoveing)
{
    wantZ = dir == 0 ? curZ - diffZ % 2 + 2f : curZ + diffZ % 2 - 2f;
}
else
{
    wantZ = dir == 0 ? curZ + 1f : curZ - 1f;
}
```



```
if (not isMoveing) then
| if (dir == 0) then wantZ = ((curZ - (diffZ % 2)) + 2.00000000) else wantZ = ((curZ + (diffZ % 2)) - 2.00000000) end;
else
| if (dir == 0) then wantZ = (curZ + 1.00000000) else wantZ = (curZ - 1.00000000) end;
end;
```

性能问题与优化策略

- 柔性可用

- 实际使用时，存在一些简单写法情形可以避免使用函数对象，我在翻译第二阶段对可能简化的情况进行移除函数对象的优化

- 自增/自减操作

- 类似复合赋值操作，lua里需要使用匿名函数对象包装执行并返回相应计算结果，实际代码中自增/自减出现在表达式中的情况比复合赋值高很多
- 对条件表达式里的简单的自增/自减操作，可以提取到条件表达式外，从而移除函数对象包装
- if/while/do-while语句的条件部分可进行类似处理

```
if (_m_IntVal++ > 0) {  
}  
if (++_m_IntVal > 0) {  
}
```



```
--  
local __unary_212_12_212_22 = this.m_IntVal; this.m_IntVal = (this.m_IntVal + 1);  
if (__unary_212_12_212_22 > 0) then  
end;  
--  
this.m_IntVal = (this.m_IntVal + 1);  
if (this.m_IntVal > 0) then  
end;
```

Q&A