



方正北魏楷书

MetaDSL及在Unity游戏开发的应用

dreaman

2021年

目录

- 第一章 初识MetaDSL
 - 1. DSL与MetaDSL
 - 2. 编译器与解释器
 - 3. 实现一个常规的脚本解释器
 - 4. 扩展更多语法
 - 5. 翻译到IL再执行
 - 6. unity3d地形生成DSL
 - 7. 描述+执行的混合模式
- 第二章 MetaDSL的演化与LOP简介
 - 1. MetaDSL的演化
 - 2. LOP思想简介
- 第三章 MetaDSL的语法与语义
 - 1. MetaDSL的语法
 - 2. MetaDSL的语义

目录

- 第四章 基于MetaDSL的剧情脚本与编辑器
 - 1. 剧情演示
 - 2. 剧情脚本的设计
 - 3. 剧情编辑器的设计
- 第五章 一个基于DSL的通用的资源处理环境（简介）
 - 1. 资源处理工具演示
 - 2. 资源处理脚本的设计
 - 3. 资源处理脚本解释器
 - 4. 再谈描述+执行的脚本解释模式
 - 5. 抛砖引玉—还可以做什么呢
- 第六章 LOP与游戏前端架构
 - 1. 游戏前端开发
 - 2. 变化与支持变化
 - 3. LOP是一种方案么

第一章 初识MetaDSL

本课程相关的代码与资源在github上

MetaDSL源码：<http://github.com/dreamanlan/MetaDSL>

各个例子在源码首页下面有链接

第一部分我们先从MetaDSL能做什么开始，先有个直观的感觉，稍后再介绍MetaDSL的细节

如果有条件，参与课程的同学请从github下载并实时演练

DSL与MetaDSL

➤ DSL—Domain Specific Language—领域特定语言

- 我们已经用过的DSL
 - XML
 - TDR协议
 - 历史上有一版Visual Foxpro使用XML来保存
 - Protocol-buffer
 - JSON
 - Google很多开源工程的build系统GYP，就是使用JSON来配置如何编译工程
 - Chromium
 - Node.js
 - YAML
 - Unity的meta文件、prefab、clip等的文本格式
 - 操作符重载
 - 矩阵运算
 - 四元数运算
 - LINQ

DSL与MetaDSL

➤ DSL —Domain Specific Language—领域特定语言

- 《Domain Specific Languages》
 - 作者Martin Fowler，出版于2010
 - DSL分类
 - 外部DSL
 - 内部DSL
 - 语言工作台
 - 使用DSL的主要作用
 - 提高程序员的开发效率
 - 提供了新的抽象层级
 - 便于理解与阅读
 - 提供封装与新的复用方式
 - 促进与领域专家的沟通
 - 使用领域术语
 - 领域专家理解DSL比理解程序代码更容易
 - 提供灵活性
 - DSL作为配置，支持运行时变更
 - DSL作为代码生成工具，支持执行环境变更

DSL与MetaDSL

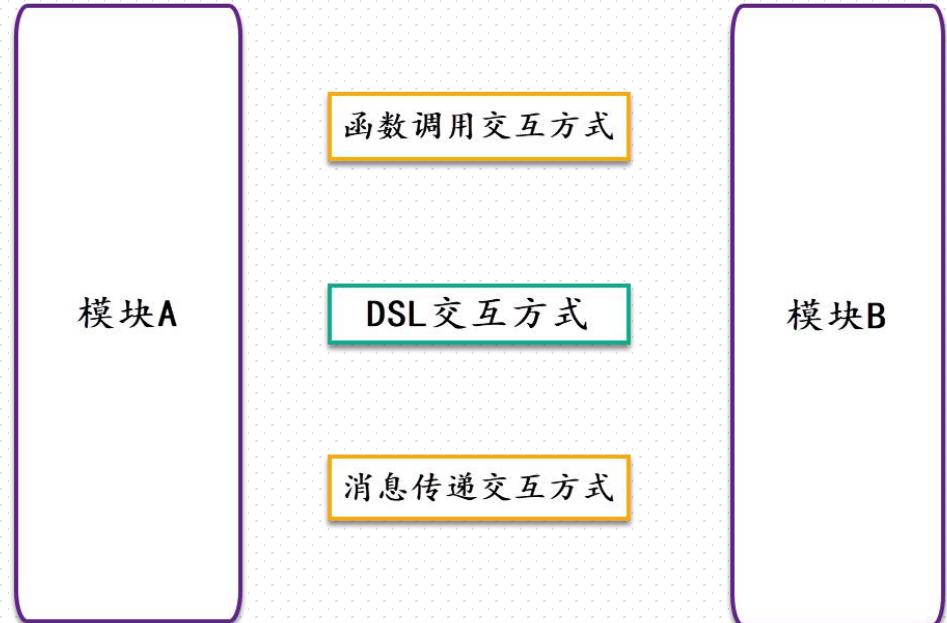
➤ DSL —Domain Specific Language—领域特定语言

- 《Domain Specific Languages》
 - 通常被认为的主要问题
 - 语言噪声
 - 需要学习新的语法
 - 复杂的程序库与API也有大的学习成本
 - 构建成本
 - 编写词法、语法解析
 - 编写解释器
 - 私有语言
 - 找不到合适的开发人员
 - 需要降低学习成本、限制语言应用范围
 - 一叶障目的抽象
 - 因为便利，所以什么都想用
 - 所有成功抽象都会面临的问题

DSL与MetaDSL

➤ DSL—Domain Specific Language—领域特定语言

- 补充一点
 - 除了提供新的抽象，还提供一种新的接口表示
 - 函数调用
 - 编译器检查
 - 耦合比较紧密—静态依赖
 - 受限表达能力
 - 消息传递
 - 缺少检查—通用类型（一种类型表示多种参数实体）
 - 松散耦合—没有静态依赖
 - 受限表达能力
 - DSL
 - 解析与解释器检查
 - 松散耦合—没有静态依赖
 - 图灵完备的表达能力
 - 作为接口表示的方式还能提供架构层面的能力



DSL与MetaDSL

➤ DSL—Domain Specific Language—领域特定语言

- 与通用目标语言（GPL）对比

- 来自《**DSL Engineering-Designing, Implementing and Using Domain-Specific Languages**》，2013

	GPLs	DSLs
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

DSL与MetaDSL

➤ DSL —Domain Specific Language—领域特定语言

- 内部DSL
 - 内部DSL使用宿主语言支持的语法，不用自己编写parser
 - 要求DSL使用者熟悉宿主语言
 - 使用宿主语言的开发环境
 - 使用时通常无法与宿主语言代码隔离
 - 实例：c++ boost库里的spirit库
 - 基于操作符重载形成特定样式的语法
 - 用户通过定义语法得到一个语法parser

我认为严格来说不是完整的DSL语言

DSL与MetaDSL

➤ DSL —Domain Specific Language—领域特定语言

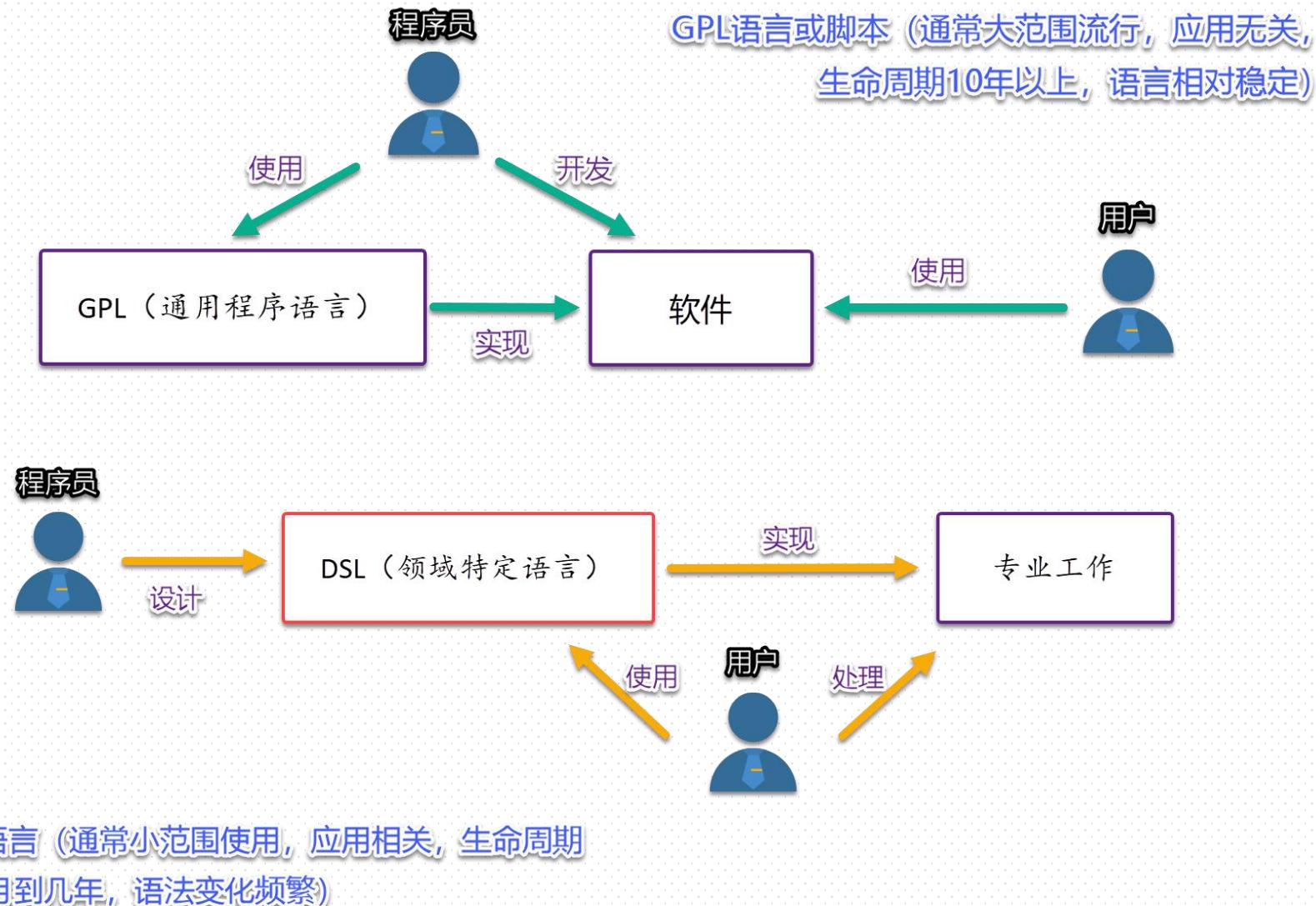
- 外部DSL
 - 词法与语法独立
 - 开发环境独立
 - 需要单独编写parser
 - 实例
 - XML
 - JSON
 - Protocol-Buffer
 - YAML

真正意义的DSL语言，但是语
法的设计与解析还是有较大工
作量，如果语法是逐渐演化设
计的，解析工作量会很可观

DSL与MetaDSL

➤ 什么是MetaDSL

- 一种辅助实现DSL的语言
- 提供了类C语言的语法
 - 语法像xml与json一样灵活
 - 在保持C系语法风格前提下，能适应绝大多数语法结构
- 省去编写DSL语言的词法与语法分析
 - 基于MetaDSL的语义数据直接开始DSL的语义分析或解释器编写



DSL与MetaDSL

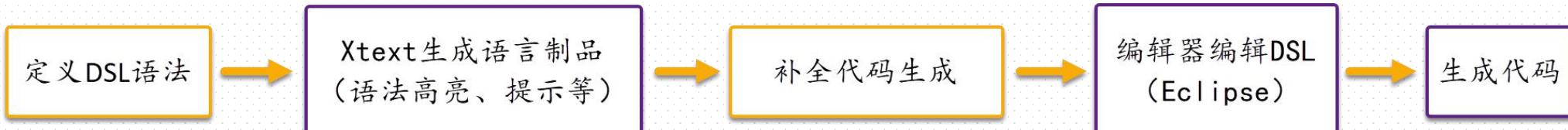
➤ 其他实现DSL语言的工具

- *Eclipse Modeling + Xtext*
 - <https://www.eclipse.org/Xtext/>
 - *Xtext*是一个构建文本化DSL的框架

LANGUAGE ENGINEERING FOR EVERYONE!

Xtext is a framework for development of programming languages and domain-specific languages. With Xtext you define your language using a powerful grammar language. As a result you get a full infrastructure, including **parser**, **linker**, **typechecker**, **compiler** as well as editing support for **Eclipse**, any editor that supports the **Language Server Protocol** and your favorite **web browser**.

Eclipse Xtext开发流程



DSL与MetaDSL

➤ 其他实现DSL语言的工具

- JetBrains MPS---Meta Programming System (元编程系统)
 - <https://www.jetbrains.com/mps/>
 - 通用的语言工作台
 - 直接定义与操作抽象语法
 - 图形化编程

Projectional Editor

Communicate with terminology that everyone in your field understands. Use non-textual notation with projectional editing including math notations, diagrams, and forms.

```
System.out.println(String.valueOf(( $\sum_{i=0}^3 \begin{bmatrix} 1 & k & 0 \\ 0 & 1.0 & 0 \\ 0 & 0 & 1 \end{bmatrix})))$ );  
System.out.println(exp(a + i * b) - exp(a) * (cos(b) + i * sin(b)));  

$$\text{matrix<Double>} s = \begin{bmatrix} 3.0 & \sin(1) & 1 & 1 \\ 2 & 1 & \frac{1.0}{2} & 2 \\ 3 & 7 - \frac{1.0}{2} + 1 & \exp(1) & 3 \\ 0 & 2 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix};$$

```

JetBrains MPS开发流程

定义AST结点规则

定义代码生成规则

编辑器编辑AST
(JetBrains)

生成代码

DSL与MetaDSL

➤ MetaDSL可以做什么？

- 我们曾经用来实现过
 - 特效脚本、解释器与编辑器
 - 技能脚本与解释器（好吧，这个真的是让策划手写的）
 - 战斗条件表达式编译器
 - 服务器配置脚本系统
 - Dotnet DLL文件补丁工具的脚本部分
 - 表格结构描述与读表器代码生成
 - 协议格式描述与协议读写代码生成
 - 帮助文档语言与生成器
 - 游戏内置GM脚本
 - 剧情脚本、解释器与编辑器（本课程会介绍）
 - C#实现的通用脚本解释器与批处理脚本（本课程会介绍）
 - 交互式资源分析与处理脚本（本课程会介绍）
- 还可以做什么？

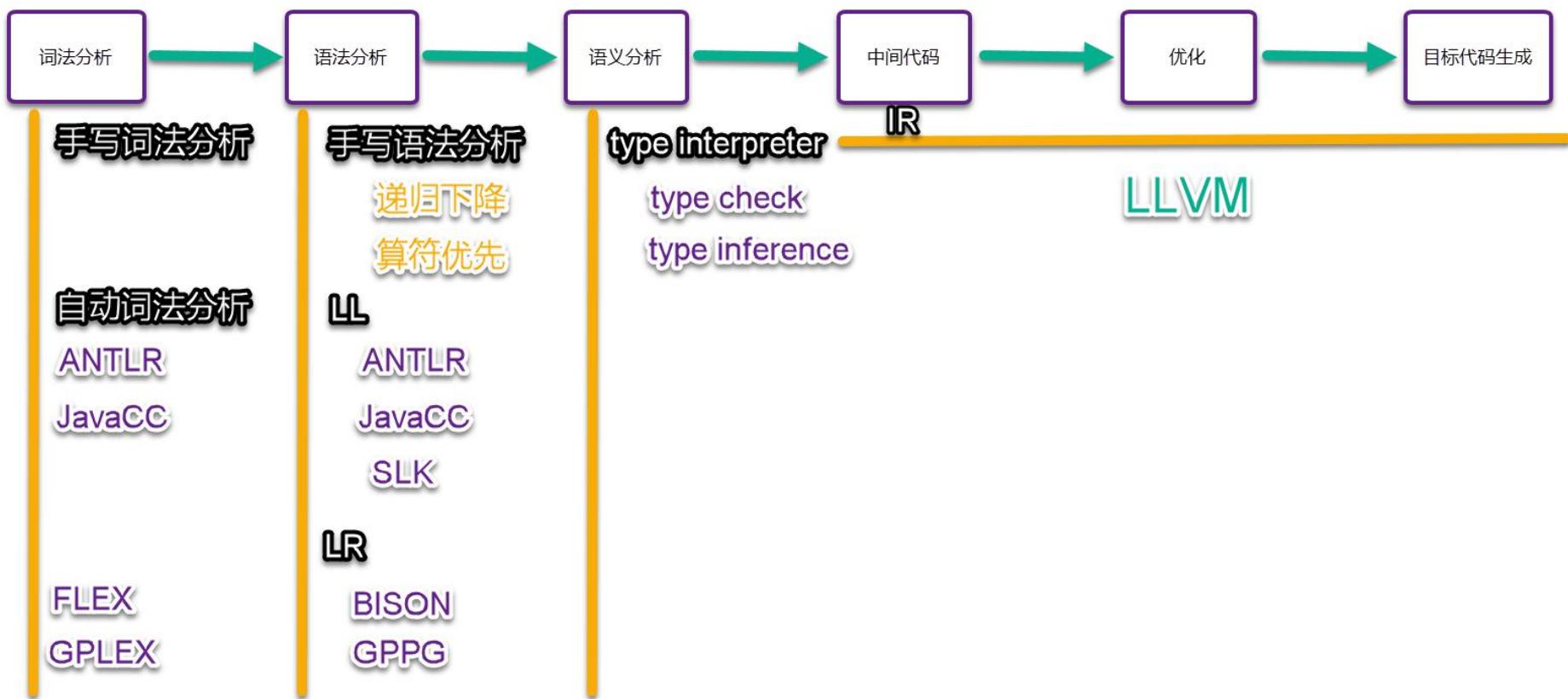
等待大家去挖掘

我们这里先从一个解释器的开发说起

编译器与解释器

➤ 编译器与解释器

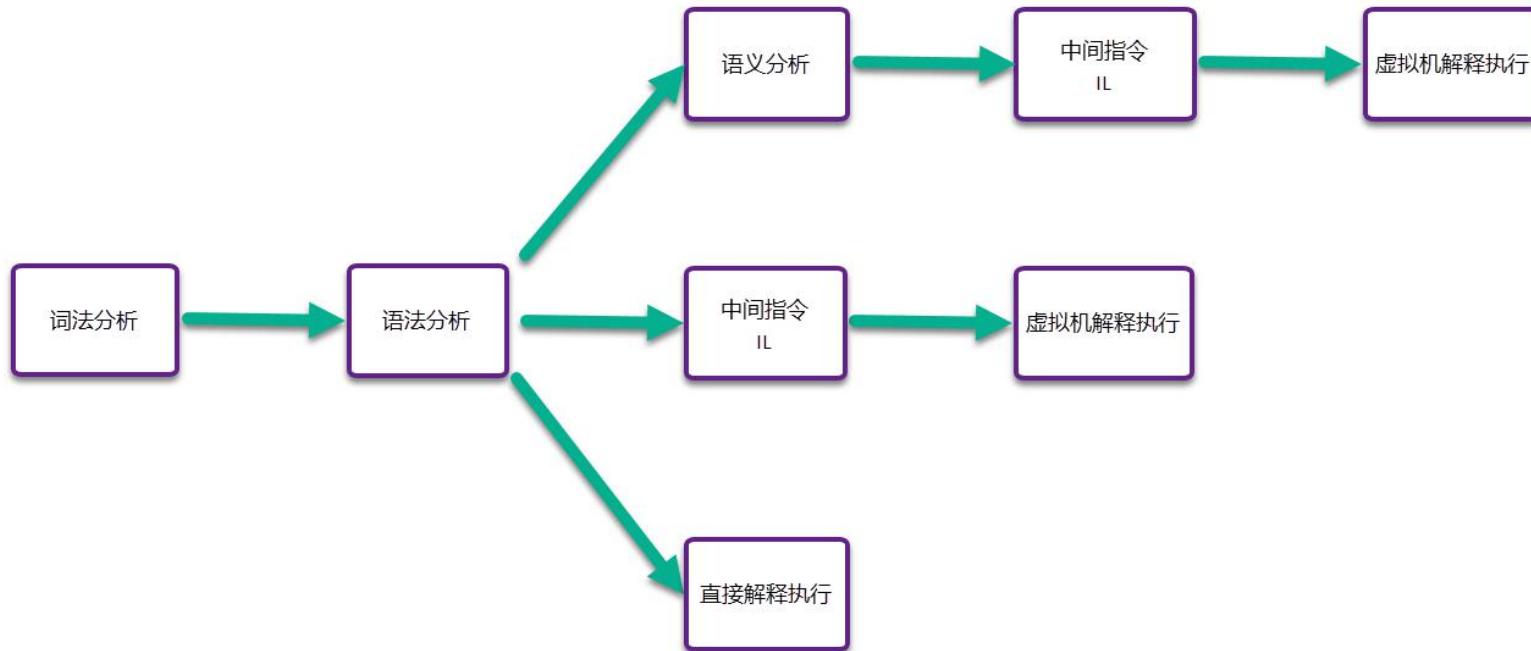
- 编译器
 - 词法分析
 - 语法分析
 - 语义分析
 - 中间代码
 - 优化
 - 目标代码生成



编译器与解释器

➤ 编译器与解释器

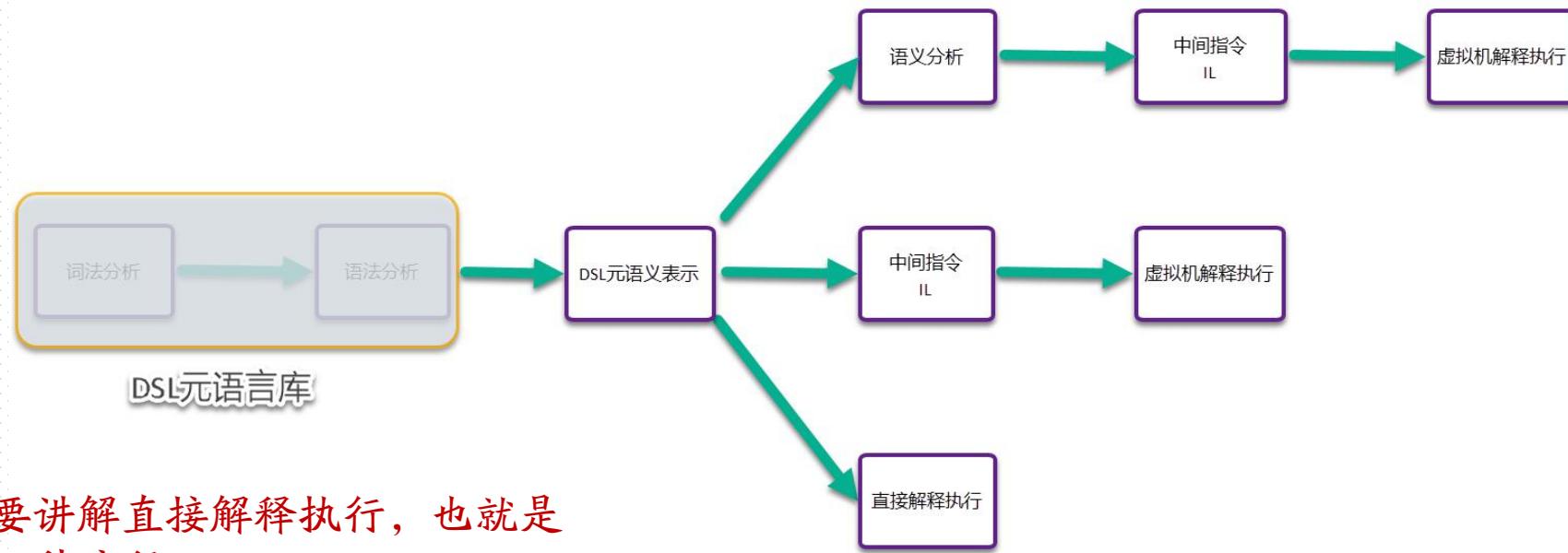
- 解释器
 - 词法分析
 - 语法分析
 - 语义分析
 - 中间指令
 - 解释执行



编译器与解释器

➤ 编译器与解释器

- 基于MetaDSL的解释器
 - 语义分析
 - 中间指令
 - 解释执行



本课程主要讲解直接解释执行，也就是最简单的一种流程

实现一个常规的脚本解释器

➤ 极简的脚本解释器

- 脚本特性
 - 函数定义
 - 内部API调用
- 代码实现
 - <https://github.com/dreamanlan/ScriptInterpreterByDsl0>

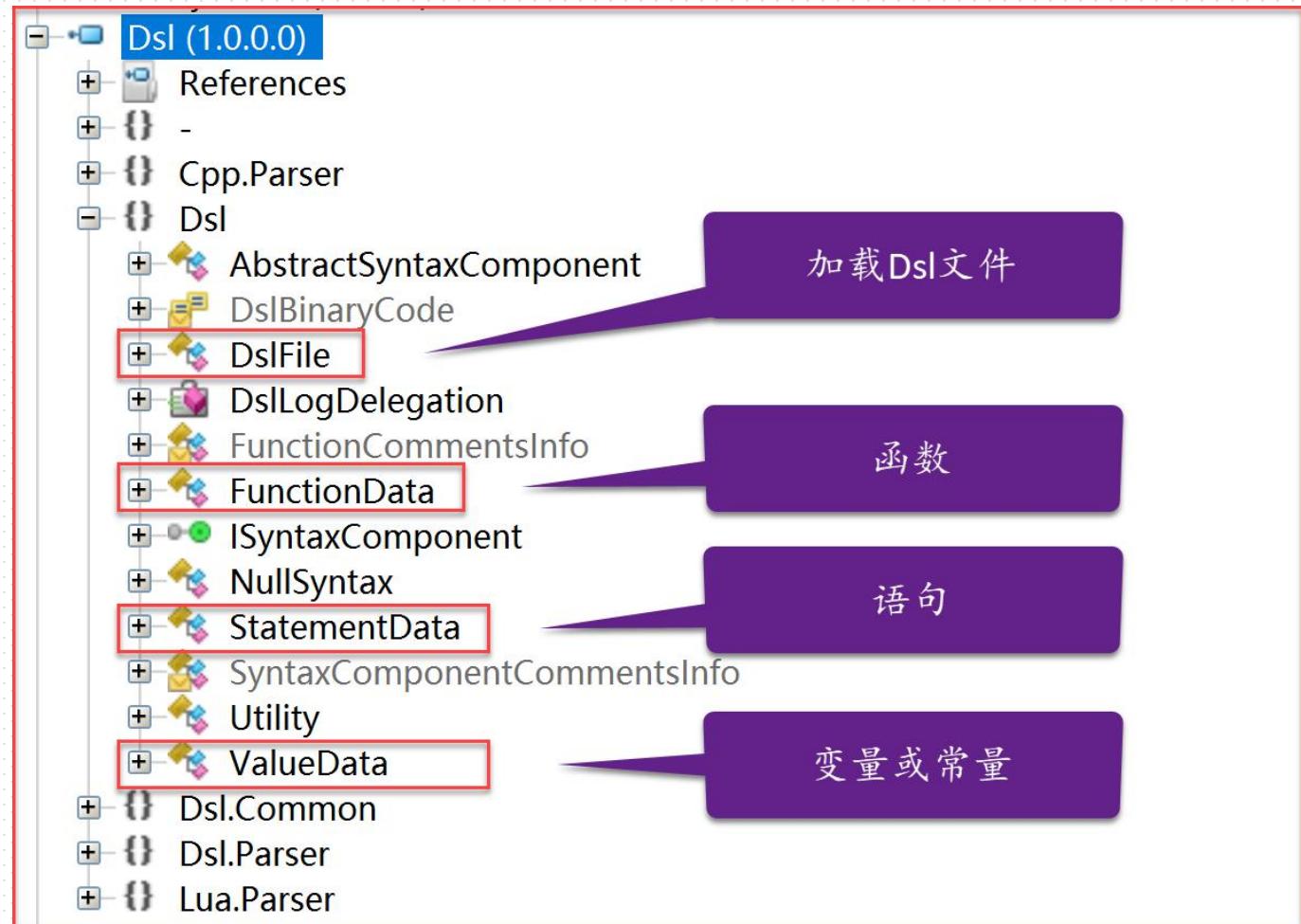
```
static void Main(string[] args)
{
    string script =
@"function(main)
{
    echo('hello world !');
    wait(1000);
    echo('1');
    wait(1000);
    echo('2');
    wait(1000);
    echo('3');
    wait(1000);
    echo('4');
    wait(1000);
    echo('5');
    wait(1000);
    echo('press any key to exit ...');
    readkey();
};";
    Execute(script);
}

private static void Execute(string code)
{
    Interpreter interpreter = new Interpreter();
    interpreter.Register("echo", new ExpressionFactoryHelper<EchoExp>());
    interpreter.Register("wait", new ExpressionFactoryHelper<WaitExp>());
    interpreter.Register("readkey", new ExpressionFactoryHelper<ReadKeyExp>());
    if (interpreter.Parse(code, "testscript")) {
        object v = interpreter.Call("main");
        if (null == v) {
            Console.WriteLine("call result: null");
        }
        else {
            Console.WriteLine("call result: {0}", v);
        }
    }
    else {
        Console.WriteLine("Parser failed.");
    }
}
```

初识MetaDSL库接口

➤ MetaDSL库

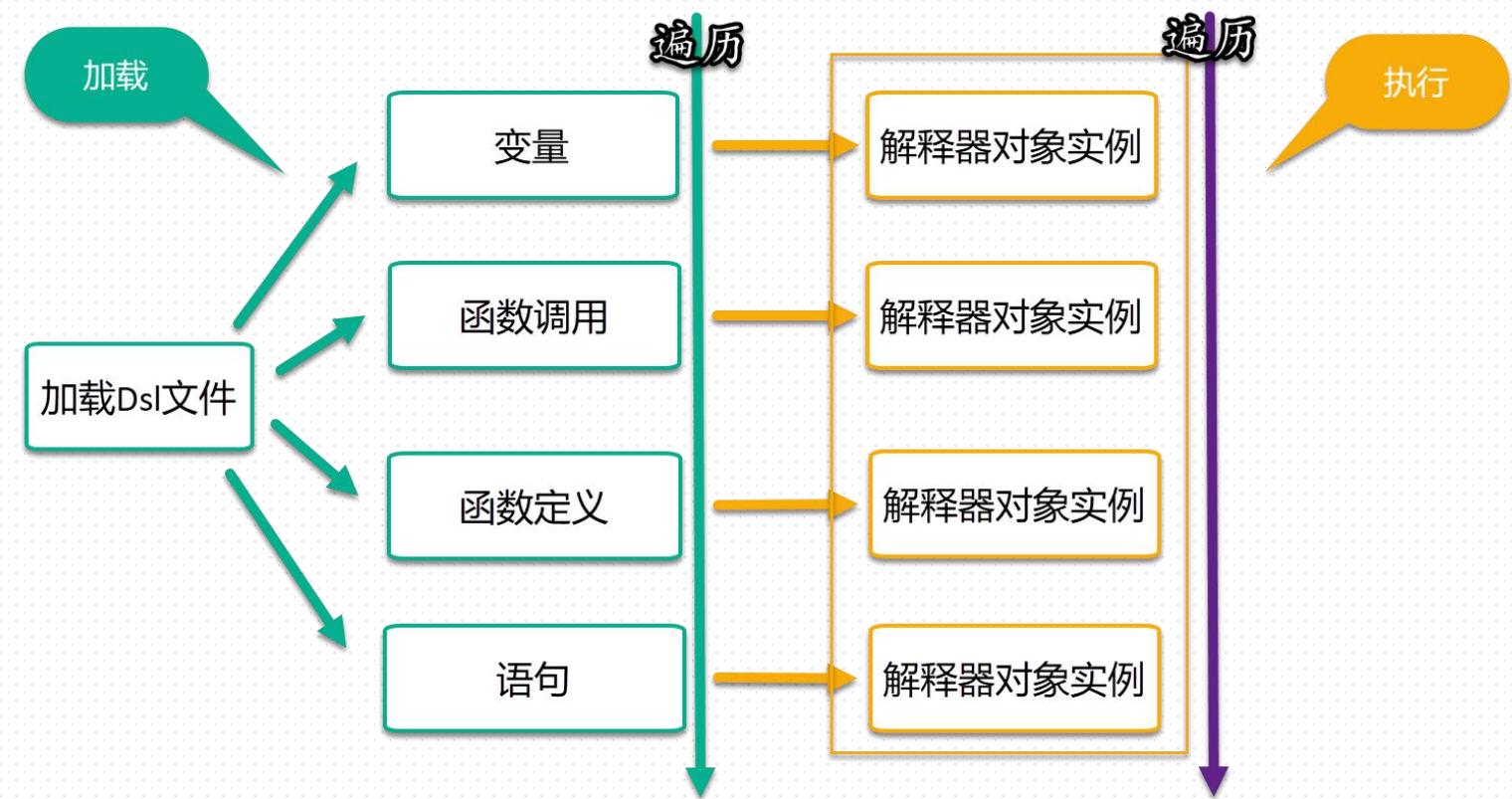
- 核心类
 - 加载dsl文件的
 - DslFile
 - 代表语句的
 - StatementData
 - 代表函数的
 - FunctionData
 - 代表变量或常量的
 - ValueData
- 以上三个类
 - 都继承类
 - AbstractSyntaxComponent
 - 都实现接口
 - ISyntaxComponent



实现一个常规的脚本解释器

➤ 极简的脚本解释器

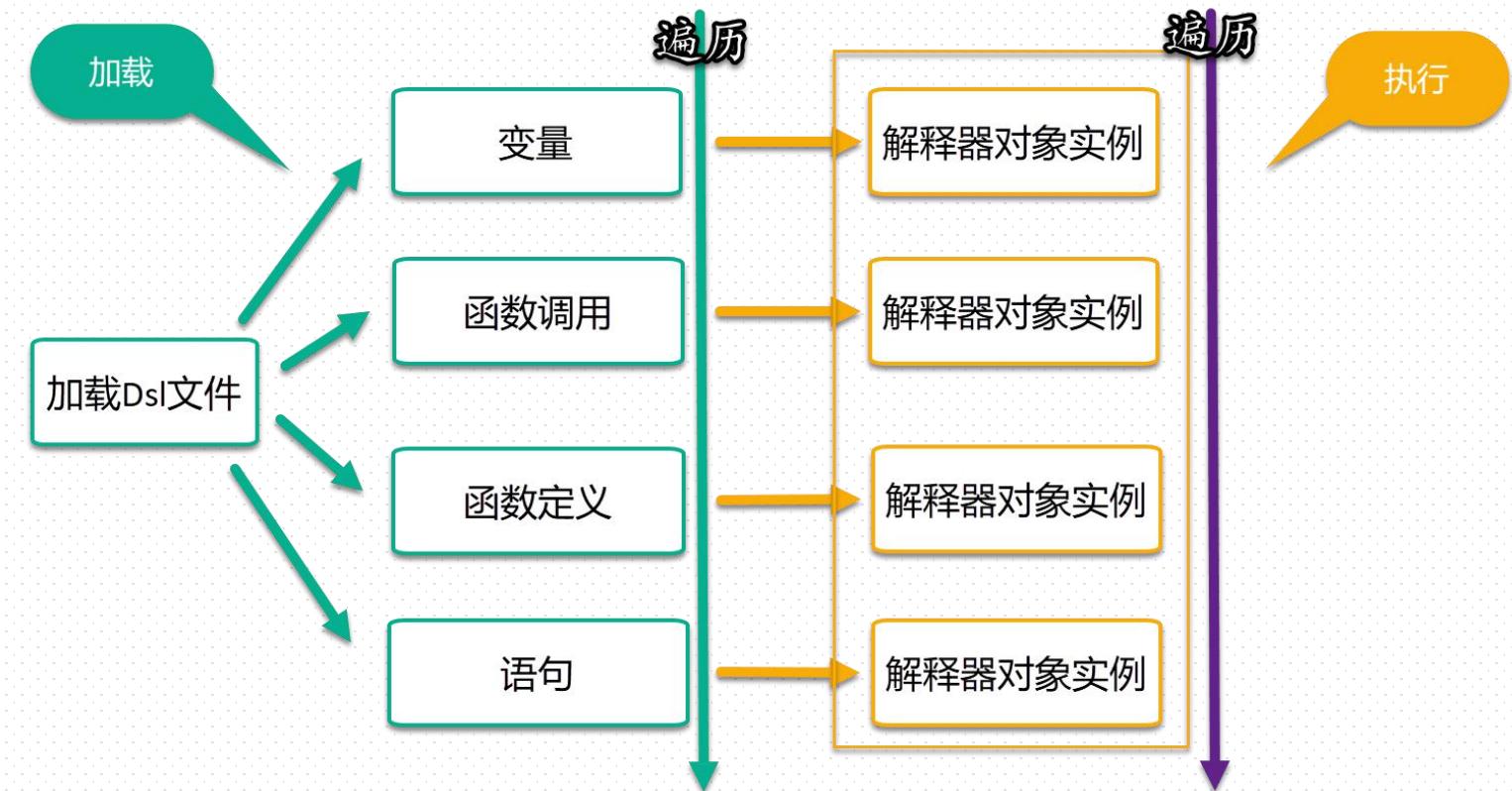
- MetaDSL库加载就提供了基本的语法单位的语义信息，一般能直接对应到脚本的语法元素
 - 变量
 - 函数调用
 - 函数定义
 - 语句
- 解释器要做的事就是对各个语法元素按代码顺序进行处理
 - 遍历
 - 处理每个语法元素，构造运行时数据
 - 执行
- 比较自然的方式是对每个语法元素提供一个对象实例来进行处理



实现一个常规的脚本解释器

➤ 极简的脚本解释器

- 每个语法元素对应一个解释器对象实例
 - 表达式对象
 - 表达式对象的两个职责
 - 加载
 - 执行
- 代码实现
 - <https://github.com/dreamanlan/ScriptInterpreterByDsl0>



实现一个常规的脚本解释器

➤ 与标准脚本语言的差别

- 没有单独的词法/语法解析，直接使用MetaDSL的语法
- 语句与内部函数都是API
 - 每个API实现2个方法，一个用于加载，一个用于执行
 - 思考：如何实现if语句和while语句？表达式呢？
- 脚本语言加载后就是各个API实例组成的一棵树，解释执行的过程是对该树的后序遍历（隐式）并调用每个结点的执行方法
 - 思考：如何改造成加载脚本翻译为中间指令，然后执行时由虚拟机解释执行这些中间指令？

扩展更多语法

➤ 添加if与while语句，普通算术运算表达式

- [IfWhileAndOperator.cs](#)

- 以While为例，图中只贴出了while代码体是多行代码的源码
- 加载过程相当于构造了一个表达式树
- 计算过程就是对每个表达式求值，并按while语句语义执行
- 为了简单，这里没有考虑break/continue{return}语句的情形
- 涉及流程例外跳转的，需要解释器提供控制变量并在相应复合语句的实现进行处理

```
internal sealed class WhileExp : AbstractExpression
{
    protected override object DoCalc()
    {
        object v = 0;
        for ( ; ; ) {
            var condVal = m_Condition.Calc();
            if (CastTo<long>(condVal) != 0) {
                for (int index = 0; index < m_Expressions.Count; ++index) {
                    v = m_Expressions[index].Calc();
                }
            }
            else {
                break;
            }
        }
        return v;
    }

    protected override bool Load(Dsl.FunctionData funcData)
    {
        if (funcData.IsHighOrder) {
            Dsl.ISyntaxComponent cond = funcData.LowerOrderFunction.GetParam(0);
            m_Condition = Interpreter.Load(cond);
            for (int ix = 0; ix < funcData.GetParamNum(); ++ix) {
                IExpression subExp = Interpreter.Load(funcData.GetParam(ix));
                m_Expressions.Add(subExp);
            }
        }
        else {
            //error
            Interpreter.Log("Interpreter error, {0} line {1}", funcData.ToString(false), funcData.GetLine());
        }
        return true;
    }

    protected override bool Load(Dsl.StatementData statementData)...
}

private IExpression m_Condition;
private List<IExpression> m_Expressions = new List<IExpression>();
}
```

翻译到IL再执行

➤ 翻译到IL再执行

- 示例实现
- 简版虚拟机，只支持
 - 过程
 - API
 - 表达式
 - 全局变量
 - 参数
- 虚拟机特点
 - 翻译到IL后会丢失语句的结构信息
 - 操作符一般固化到虚拟机指令
 - 语句不太容易做成API
 - 编译时必须知道语句的涵义，以便翻译为多条IL指令
 - 除非API本身包含编译方法
 - 但会与虚拟机耦合过紧

```
internal interface IApi
{
    long Calc(long[] args);
}

internal enum InsEnum : int
{
    ARG = 0,
    VARSET,
    VAR,
    NEG,
    ADD,
    SUB,
    MUL,
    DIV,
    MOD,
    AND,
    OR,
    NOT,
    GT,
    GE,
    EQ,
    NE,
    LE,
    LT,
    LSHIFT,
    RSHIFT,
    BITAND,
    BITOR,
    BITXOR,
    BITNOT,
    PUSH,
    CALL,
    NUM
}

internal class Instruction
{
    internal InsEnum Opcode;
    internal long Operand;
```

Unity3d地形生成DSL

➤ Unity3d地形生成DSL

- [TerrainProcessor](#)
 - <https://github.com/dreamanlan/TerrainGenByDsl>
- Unity3d地形基于高度图，地形生成就是对各个像素的处理
- 所以可以采取类似像素shader的思路

```
1  input
2  {
3      resettrees(true);
4      rect(11,11,489,489);
5      sampler("img", "Assets/test.bmp");
6  }
7  height
8  {
9      $x=arg(0);
10     $y=arg(1);
11     height = samplered("img", $x, $y)/1000.0+samplegreen("img", $x, $y)/1000.0+sampleblue("img", $x, $y)/1000.0;
12     /*
13     if(height>0.1 && rndint(0,100)<5){
14         addtree(0,$x/128,height,$y/128,0,2,3,0x00ff00,0xff0000);
15     };
16     */
17 }
18 alphamap
19 {
20     $x=arg(0);
21     $y=arg(1);
22     setalpha(0, sin($x/180.0));
23     setalpha(1, cos($y/180.0));
24 }
25 detail
26 {
27     $x=arg(0);
28     $y=arg(1);
29     $layer=arg(2);
30     detail = 0;
31 }
```

描述+执行的混合模式

➤ 描述+执行的混合模式

- 前面的地形处理DSL和之前的脚本解释器不太一样
 - 代码结构分为input、height、alphamap、detail等部分
 - 后面几部分的语句块部分是可以执行的脚本
 - Input的语句块部分不是可以执行的脚本，而是提供给解释器的各种描述信息
- 部分提供描述信息、部分提供执行代码的混合模式正是MetaDSL显著区别于通用脚本语言的特点
 - 描述部分更像是配置数据，类似xml/json的作用
 - 执行部分则更像脚本
 - 解释器可以按照描述信息的指引，组合并调度执行部分的代码，从而实现比单纯执行脚本更灵活的效果

```
1  input
2  {
3      resettrees(true);
4      rect(11,11,489,489);
5      sampler("img", "Assets/test.bmp");
6  }
7  height
8  {
9      $x=arg(0);
10     $y=arg(1);
11     height = samplered("img", $x, $y)/1000.0+samplegreen("img", $x, $y)/1000.0+sampleblue("img", $x, $y)/1000.0;
12     /*
13     if(height>0.1 && rndint(0,100)<5{
14         addtree(0,$x/128,height,$y/128,0,2,3,0x00ff00,0xffff0000);
15     };
16     */
17 }
18 alphamap
19 {
20     $x=arg(0);
21     $y=arg(1);
22     setalpha(0, sin($x/180.0));
23     setalpha(1, cos($y/180.0));
24 }
25 detail
26 {
27     $x=arg(0);
28     $y=arg(1);
29     $layer=arg(2);
30     detail = 0;
31 }
```

描述+执行的混合模式

➤ 描述+执行的混合模式

- 从解释器看，这种模式下要实现2部分解释器
 - 描述部分解释器
 - 执行部分解释器
- 执行部分通常是一个常规的命令式脚本的功能
 - 具有结构化语言的特征
 - 支持顺序、分支、循环语句
 - 支持API
 - 可选择支持过程抽象（视需求复杂程度）
 - 不需要支持数据抽象
 - 不需要支持自定义数据结构
 - 不需要支持类定义
- 也就是说，执行部分解释器其实可以实现为通用的解释器，在这个示例里的DslCalculator就是一个通用的实现，实际上我们项目里的多个DSL都共用了这个执行部分的解释器（稍后在介绍MetaDSL语法语义时我们会继续分析）

描述+执行的混合模式

➤ 描述+执行的混合模式

- 按使用场景，MetaDSL的用法可以分为几类：
 1. 像xml或json一样用作信息描述或配置，此种用法不涉及执行部分解释器
 2. 本例中地形生成这样的描述+执行的混合模式，其中执行部分解释器采用通用解释器
 3. 描述+执行的混合模式，但运行时需要专门的运行模型，此时执行部分解释器不能采用通用解释器，我们后面会介绍的剧情脚本就是此类
 4. 用于实现普通脚本，此种用法不涉及描述部分解释器，只涉及执行部分解释器（可能是通用解释器或专用解释器）



第二章 DSL与MetaDSL

MetaDSL的演化

➤ MetaDSL的演化

前置说明：

这里介绍MetaDSL的设计起源与演化，选择的语法是MetaDSL的一种选择，这里主要是展示语言表达力层面的考虑，大家完全可以设计其他样式的语法（当然也许会有一定难度，我参考的C系语言语法恰好具有一定的递归与自相似性）

MetaDSL的演化

➤ MetaDSL的演化

- 最早源于07年的一个特效编辑器
 - OGRE引擎
 - OGRE引擎里材质、后处理、粒子系统等都使用自定义脚本描述，概念上就是一种DSL
 - 特效从一开始也选择使用一种类似particle system的方式描述
- 采用相对标准的方式实现
 - 定义EBNF语法
 - 使用语法分析生成器生成parser
 - 手写词法分析与其他

MetaDSL的演化

```
// A sparkly purple fountain.
particle_system Examples/PurpleFountain
{
    material      Examples/Flare2
    particle_width 20
    particle_height 40
    cull_each     false
    quota         10000
    billboard_type oriented_self

    // Area emitter.
    emitter Point
    {
        angle          15
        emission_rate 75
        time_to_live   3
        direction      0 1 0
        velocity_min   250
        velocity_max   300
        colour_range_start 0 0 0
        colour_range_end   1 1 1
    }

    // Gravity.
    affector LinearForce
    {
        force_vector    0 -100 0
        force_application add
    }

    // Fader.
    affector ColourFader
    {
        red -0.25
        green -0.25
        blue -0.25
    }
}
```

MetaDSL的演化

➤ MetaDSL的演化

- SLK语法分析程序生成器
 - <http://www.slkpg.org/>
- effect语法规例

```
1 EFFECTS:
2   { EFFECT_STATEMENT }
3
4 EFFECT_STATEMENT:
5   effect NAME \{ { EFFECT_ELEMENT } \}
6
7 EFFECT_ELEMENT:
8   OBJ_TREE_ELEMENT
9   TIMELINE_ELEMENT
10
11 OBJ_TREE_ELEMENT:
12   object NAME \{ { OBJ_ATTR } OBJ_TREE_ELEMENT \}
13
14 OBJ_ATTR:
15   position NUM NUM NUM
16   rotation NUM NUM NUM NUM
17   scale NUM NUM NUM
18   mesh PATH
19   particle PATH
20
21 TIMELINE_ELEMENT:
22   timeline NAME \{ { TRACK_ELEMENT } \}
23
24 TRACK_ELEMENT:
25   track NAME \{ { TRIGER_ELEMENT } \}
26
27 TRIGER_ELEMENT:
28   trigger NUM \{ { ACTION_ELEMENT } \}
29
30 ACTION_ELEMENT:
31   playsound PATH
32   moveobj PATH NUM NUM NUM NUM
33   rotateobj PATH NUM NUM NUM NUM NUM
34   scaleobj PATH NUM NUM NUM NUM
35   changecolor PATH NUM NUM NUM NUM NUM
```

MetaDSL的演化

➤ MetaDSL的演化

- effect文件示例

```
1  effect BOMB
2  {
3      object ROOT
4      {
5          position 1 2 3
6          rotation 0 0 0 1
7          scale 1 1 1
8          mesh "assets/fbx/go1.fbx"
9
10     object NODE1
11     {
12         position 1 2 3
13         rotation 0 0 0 1
14         scale 1 1 1
15         particle "assets/ps/fly_fire.ps"
16
17         object NODE2
18         {
19             position 1 2 3
20             rotation 0 0 0 1
21             scale 1 1 1
22             particle "assets/ps/bome.ps"
23         }
24     }
25 }
26 timeline
27 {
28     track t1
29     {
30         trigger 10
31         {
32             changecolor "ROOT/NODE1" 10 0.5 0.6 0.7 0.3
33         }
34         trigger 15
35         {
36             moveobj "ROOT" 3 5 0 8
37         }
38     }
39 }
40 }
```

MetaDSL的演化

➤ MetaDSL的演化

- 带语义行为的effect语法示例

```
1 // EwaExecute.txt - generated by the SLK parser generator
2
3 // This file can be edited into the SlkAction class.
4
5 public void execute ( int number )
6 {
7     switch ( number ) {
8         case 1: addEffect(); break;
9         case 2: addObject(); break;
10        case 3: setPosition(); break;
11        case 4: setRotation(); break;
12        case 5: setScale(); break;
13        case 6: setMesh(); break;
14        case 7: setParticle(); break;
15        case 8: addTimeline(); break;
16        case 9: addTrack(); break;
17        case 10: addTriger(); break;
18        case 11: addPlaySound(); break;
19        case 12: addMoveObj(); break;
20        case 13: addRotateObj(); break;
21        case 14: addScaleObj(); break;
22        case 15: addChangeColor(); break;
23        case 16: pushName(); break;
24        case 17: pushPath(); break;
25        case 18: pushNum(); break;
26    }
27 }
28 }
```

```
1 EFFECTS:
2     { EFFECT_STATEMENT }
3
4 EFFECT_STATEMENT:
5     effect NAME _action_addEffect \{ { EFFECT_ELEMENT } \}
6
7 EFFECT_ELEMENT:
8     OBJ_TREE_ELEMENT
9     TIMELINE_ELEMENT
10
11 OBJ_TREE_ELEMENT:
12     object NAME _action_addObject \{ { OBJ_ATTR } OBJ_TREE_ELEMENT \}
13
14 OBJ_ATTR:
15     position NUM NUM NUM _action_setPosition
16     rotation NUM NUM NUM NUM _action_setRotation
17     scale NUM NUM NUM _action_setScale
18     mesh PATH _action_setMesh
19     particle PATH _action_setParticle
20
21 TIMELINE_ELEMENT:
22     timeline NAME _action_addTimeline \{ { TRACK_ELEMENT } \}
23
24 TRACK_ELEMENT:
25     track NAME _action_addTrack \{ { TRIGER_ELEMENT } \}
26
27 TRIGER_ELEMENT:
28     triger NUM _action_addTriger \{ { ACTION_ELEMENT } \}
29
30 ACTION_ELEMENT:
31     playsound PATH _action_addPlaySound
32     moveobj PATH NUM NUM NUM NUM _action_addMoveObj
33     rotateobj PATH NUM NUM NUM NUM NUM _action_addRotateObj
34     scaleobj PATH NUM NUM NUM NUM _action_addScaleObj
35     changecolor PATH NUM NUM NUM NUM NUM _action_addChangeColor
36
37 NAME:
38     IDENTIFIER _action_pushName
39
40 PATH:
41     STRING _action_pushPath
42
43 NUM:
44     NUMBER _action_pushNum
```

MetaDSL的演化

➤ MetaDSL的演化

- 特效的组成部分是不断扩充的，新需求时常也会提出对语法的增删改
 - 效果元素或属性都由一个语法关键字引导
 - 修改需要相应修改语法定义
 - 相应的重新生成parser
 - 每个语法单位需要一个对应的语义行为
 - 对应的语义行为也需要修改
 - 语义行为的实现需要添加或修改
 - . . .

是否有办法定义一个通用的语法，不要总是添加新的效果组件就重新定义语言呢？

MetaDSL的演化

➤ MetaDSL的演化

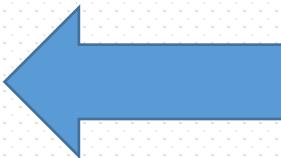
- 通用化的思路
 - 把关键字变成普通标识符，就不再需要相应的语法规则了
 - 去掉关键字后，需要一种通用的结构表示所有相似的语法
- 观察C系语言的语法构造，可以抽象出四类
 - 语句
 - if(exp){}else{}
 - while(exp){}
 - do{}while(exp);
 - 函数定义
 - void funcname(param1,param2,...){}
 - 函数调用
 - funcname(arg1,arg2,...);
 - 变量或标识符
 - break;
 - continue;
 - int a;

MetaDSL的演化

➤ MetaDSL的演化

- effect语法通用化改造之一
 - 去关键字
 - 统一属性与元素

```
1 EFFECTS:  
2   { EFFECT_STATEMENT }  
3  
4 EFFECT_STATEMENT:  
5   NAME ( VALUE ) \{ { ATTR } { EFFECT_STATEMENT } \}  
6  
7 ATTR:  
8   NAME ( VALUE { VALUE } )  
9
```



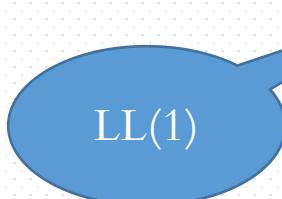
```
1 EFFECTS:  
2   { EFFECT_STATEMENT }  
3  
4 EFFECT_STATEMENT:  
5   effect NAME \{ { EFFECT_ELEMENT } \}  
6  
7 EFFECT_ELEMENT:  
8   OBJ_TREE_ELEMENT  
9   TIMELINE_ELEMENT  
10  
11 OBJ_TREE_ELEMENT:  
12   object NAME \{ { OBJ_ATTR } OBJ_TREE_ELEMENT \}  
13  
14 OBJ_ATTR:  
15   position NUM NUM NUM  
16   rotation NUM NUM NUM NUM  
17   scale NUM NUM NUM  
18   mesh PATH  
19   particle PATH  
20  
21 TIMELINE_ELEMENT:  
22   timeline NAME \{ { TRACK_ELEMENT } \}  
23  
24 TRACK_ELEMENT:  
25   track NAME \{ { TRIGER_ELEMENT } \}  
26  
27 TRIGER_ELEMENT:  
28   trigger NUM \{ { ACTION_ELEMENT } \}  
29  
30 ACTION_ELEMENT:  
31   playsound PATH  
32   moveobj PATH NUM NUM NUM NUM  
33   rotateobj PATH NUM NUM NUM NUM NUM  
34   scaleobj PATH NUM NUM NUM NUM  
35   changecolor PATH NUM NUM NUM NUM NUM
```

MetaDSL的演化

➤ MetaDSL的演化

- effect语法通用化改造之一
 - 去关键字
 - 统一属性与元素

```
1 EFFECTS:  
2   { EFFECT_STATEMENT }  
3  
4 EFFECT_STATEMENT:  
5   NAME ( VALUE ) \{ { ATTR } { EFFECT_STATEMENT } \}  
6  
7 ATTR:  
8   NAME ( VALUE { VALUE } )  
9
```



```
1 EFFECTS:  
2   { EFFECT_STATEMENT }  
3  
4 EFFECT_STATEMENT:  
5   NAME ( VALUE ) \{ { ATTR_OR_ELEMENT } \}  
6  
7 ATTR_OR_ELEMENT:  
8   NAME ( VALUE { VALUE } ) [ \{ { ATTR_OR_ELEMENT } \} ]  
9
```



```
1 EFFECTS:  
2   { EFFECT_ELEMENT }  
3  
4 EFFECT_ELEMENT:  
5   NAME ( VALUE { VALUE } ) [ \{ { EFFECT_ELEMENT } \} ]  
6
```

MetaDSL的演化

➤ MetaDSL的演化

- effect语法通用化改造之一
 - 改造后的effect文件格式
 - 去关键字
 - 统一属性与元素
 - 属性值用括号括起来
 - 正确识别属性开始与结束
 - **增加新属性不需要再修改文法**

```
1  effect(BOMB)
2  {
3      object(ROOT)
4      {
5          position(1 2 3)
6          rotation(0 0 0 1)
7          scale(1 1 1)
8          mesh("assets/fbx/go1.fbx")
9
10     object(NODE1)
11     {
12         position(1 2 3)
13         rotation(0 0 0 1)
14         scale(1 1 1)
15         particle("assets/ps/fly_fire.ps")
16
17     object(NODE2)
18     {
19         position(1 2 3)
20         rotation(0 0 0 1)
21         scale(1 1 1)
22         particle("assets/ps/bome.ps")
23     }
24   }
25 }
26 timeline
27 {
28     track(t1)
29     {
30         trigger(10)
31         {
32             changeColor("ROOT/NODE1" 10 0.5 0.6 0.7 0.3)
33         }
34         trigger(15)
35         {
36             moveObj("ROOT" 3 5 0 8)
37         }
38     }
39 }
40 }
```

MetaDSL的演化

➤ MetaDSL的演化

- effect语法通用化改造之二
 - 属性参数部分可选
 - 允许if/else样式的元素描述

```
1 EFFECTS:  
2   { EFFECT_ELEMENT }  
3  
4 EFFECT_ELEMENT:  
5   EFFECT_ELEMENT_SINGLE { EFFECT_ELEMENT_SINGLE }  
6  
7 EFFECT_ELEMENT_SINGLE:  
8   NAME [ ( VALUE { VALUE } ) ] [ \{ { EFFECT_ELEMENT } \} ]  
9
```

二义性文法
[发生在LL(7)
时]

问题原因是effect元素之
间现在必须要一个分隔
符，以区分是一新的元
素还是由多个单元素构
成的effect元素

```
1 EFFECTS:  
2   { EFFECT_ELEMENT }  
3  
4 EFFECT_ELEMENT:  
5   NAME ( VALUE { VALUE } ) [ \{ { EFFECT_ELEMENT } \} ]  
6
```

```
1 EFFECTS:  
2   { EFFECT_ELEMENT ; }  
3  
4 EFFECT_ELEMENT:  
5   EFFECT_ELEMENT_SINGLE { EFFECT_ELEMENT_SINGLE }  
6  
7 EFFECT_ELEMENT_SINGLE:  
8   NAME [ ( VALUE { VALUE } ) ] [ \{ { EFFECT_ELEMENT ; } \} ]  
9
```

二义性消除并保持
LL(1)

MetaDSL的演化

➤ MetaDSL的演化

- effect语法通用化改造之二
 - 改造后的effect文件格式
 - 属性参数部分可选
 - 允许if/else样式的元素描述
 - onlow部分
 - 无属性参数
 - onlow是object的另一选择路径

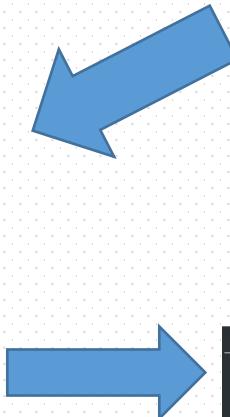
```
1   effect(BOMB)
2   {
3       object(ROOT)
4       {
5           position(1 2 3);
6           rotation(0 0 0 1);
7           scale(1 1 1);
8           mesh("assets/fbx/go1.fbx");
9
10      object(NODE1)
11      {
12          position(1 2 3);
13          rotation(0 0 0 1);
14          scale(1 1 1);
15          particle("assets/ps/fly_fire.ps");
16
17      object(NODE2)
18      {
19          position(1 2 3);
20          rotation(0 0 0 1);
21          scale(1 1 1);
22          particle("assets/ps/bome.ps");
23      }
24      onlow
25      {
26          position(0 0 0);
27          rotation(0 0 0 1);
28          scale(1 1 1);
29      };
30  };
31  };
32  timeline
33  {
34      track(t1)
35      {
36          trigger(10)
37          {
38              changecolor("ROOT/NODE1" 10 0.5 0.6 0.7 0.3);
39          };
40          trigger(15)
41          {
42              moveobj("ROOT" 3 5 0 8);
43          };
44      };
45  };
46 }
```

MetaDSL的演化

➤ MetaDSL的演化

- effect语法通用化改造之三
 - 属性值允许属性或元素样式表示

```
1 EFFECTS:  
2   { EFFECT_ELEMENT ; }  
3  
4 EFFECT_ELEMENT:  
5   EFFECT_ELEMENT_SINGLE { EFFECT_ELEMENT_SINGLE }  
6  
7 EFFECT_ELEMENT_SINGLE:  
8   NAME_OR_VALUE [ ( { EFFECT_ELEMENT } ) ] [ \{ { EFFECT_ELEMENT ; } \} ]  
9
```



属性值直接换成
EFFECT_ELEMENT产生二义性，
和之前类似，也是发生在LL(7)时，
问题原因也类似，在多个属性值
之间需要添加分隔符

```
1 EFFECTS:  
2   { EFFECT_ELEMENT ; }  
3  
4 EFFECT_ELEMENT:  
5   EFFECT_ELEMENT_SINGLE { EFFECT_ELEMENT_SINGLE }  
6  
7 EFFECT_ELEMENT_SINGLE:  
8   NAME [ ( VALUE { VALUE } ) ] [ \{ { EFFECT_ELEMENT ; } \} ]  
9
```

```
1 EFFECTS:  
2   { EFFECT_ELEMENT ; }  
3  
4 EFFECT_ELEMENT:  
5   EFFECT_ELEMENT_SINGLE { EFFECT_ELEMENT_SINGLE }  
6  
7 EFFECT_ELEMENT_SINGLE:  
8   NAME_OR_VALUE [ ( { EFFECT_ELEMENT , } ) ] [ \{ { EFFECT_ELEMENT ; } \} ]  
9
```

回归LL(1)

MetaDSL的演化

➤ MetaDSL的演化

- effect语法通用化改造之三
 - 改造后的effect文件格式
 - 属性值允许属性或元素样式表示
 - 属性值写成属性样式，如changecolor
 - 可以标明参数的涵义
 - 可以实现可选参数效果
 - 现在这个语法表现力应该与xml相当了

```
1  effect(BOMB)
2  {
3      object(ROOT)
4      {
5          position(1, 2, 3);
6          rotation(0, 0, 0, 1);
7          scale(1, 1, 1);
8          mesh("assets/fbx/go1.fbx");
9
10     object(NODE1)
11     {
12         position(1, 2, 3);
13         rotation(0, 0, 0, 1);
14         scale(1, 1, 1);
15         particle("assets/ps/fly_fire.ps");
16
17         object(NODE2)
18         {
19             position(1, 2, 3);
20             rotation(0, 0, 0, 1);
21             scale(1, 1, 1);
22             particle("assets/ps/bome.ps");
23         }
24         onlow
25         {
26             position(0, 0, 0);
27             rotation(0, 0, 0, 1);
28             scale(1, 1, 1);
29         };
30     };
31     timeline
32     {
33         track(t1)
34         {
35             trigger(10)
36             {
37                 changecolor("ROOT/NODE1", duration(10), color(0.5, 0.6, 0.7, 0.3));
38             };
39             trigger(15)
40             {
41                 moveobj("ROOT", duration(3), position(5, 0, 8));
42             };
43         };
44     };
45 };
46 }
```

MetaDSL的演化

➤ MetaDSL的演化

- 最终effect语法（带上语义行为）
 - 为了方便进行语义行为处理，把属性与语句2个部分拆开书写了

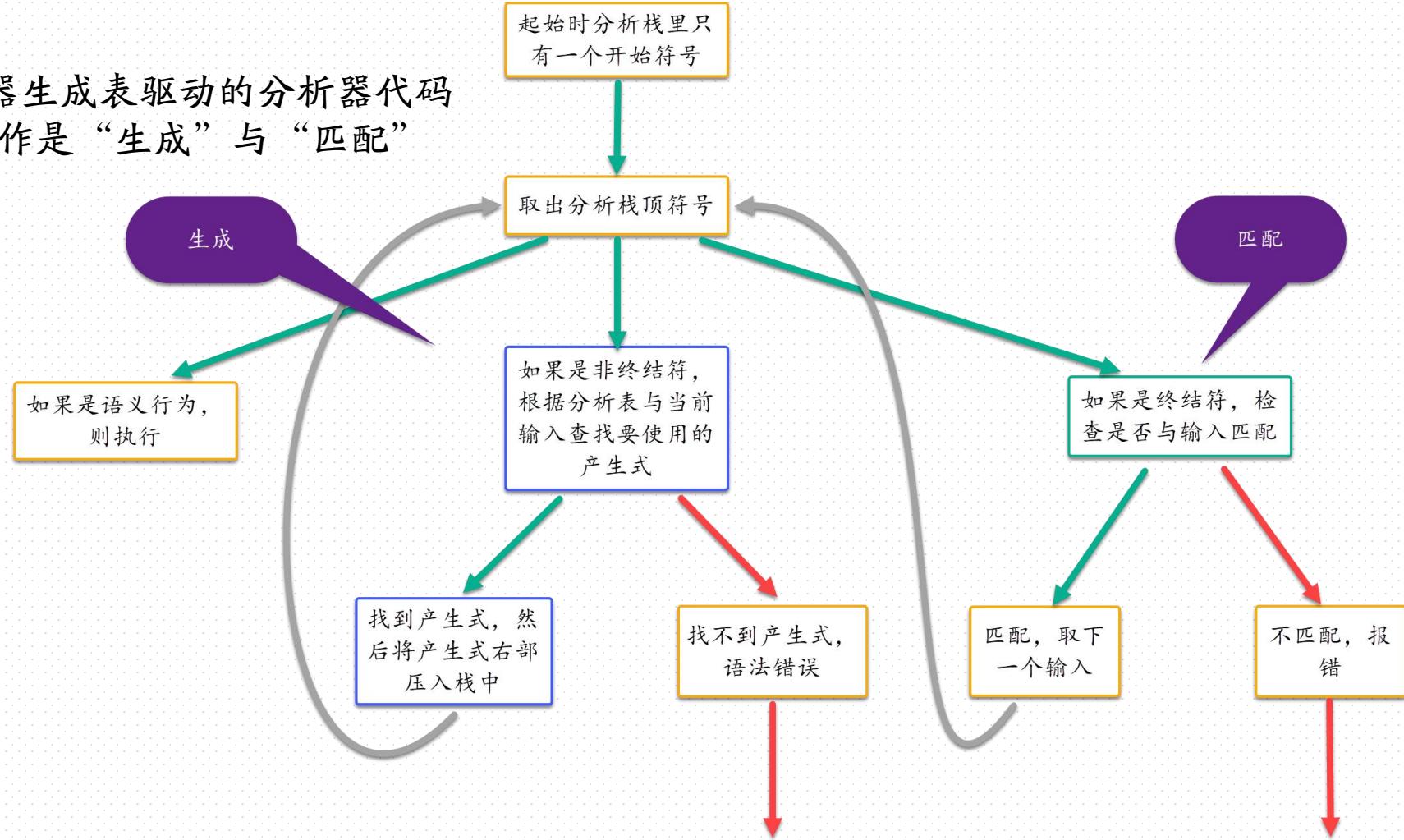
```
1 EFFECTS:  
2     { EFFECT_ELEMENT_DEF ; }  
3  
4 EFFECT_ELEMENT_DEF:  
5     _action_beginEffect EFFECT_ELEMENT _action_endEffect  
6  
7 EFFECT_ELEMENT:  
8     EFFECT_ELEMENT_SINGLE { EFFECT_ELEMENT_SINGLE }  
9  
10 EFFECT_ELEMENT_SINGLE:  
11     NAME_OR_VALUE _action_addSingleEffectElement [ EFFECT_ELEMENT_PARAMS ] [ EFFECT_ELEMENT_STATEMENTS ]  
12  
13 EFFECT_ELEMENT_PARAMS:  
14     ( _action_beginParam { EFFECT_ELEMENT_DEF , } ) _action_endParam  
15  
16 EFFECT_ELEMENT_STATEMENTS:  
17     \{ _action_beginStatement { EFFECT_ELEMENT_DEF ; } \} _action_endStatement  
18  
19 NAME_OR_VALUE:  
20     IDENTIFIER _action_pushId  
21     STRING _action_pushString  
22     NUMBER _action_pushNumber  
23
```

```
1 // DefExecute.txt - generated by the SLK parser generator  
2  
3 // This file can be edited into the SlkAction class.  
4  
5 public void execute ( int number )  
6 {  
7     switch ( number ) {  
8         case 1: beginEffect(); break;  
9         case 2: endEffect(); break;  
10        case 3: addSingleEffectElement(); break;  
11        case 4: beginParam(); break;  
12        case 5: endParam(); break;  
13        case 6: beginStatement(); break;  
14        case 7: endStatement(); break;  
15        case 8: pushId(); break;  
16        case 9: pushString(); break;  
17        case 10: pushNumber(); break;  
18    }  
19 }  
20  
21
```

MetaDSL的演化

➤ MetaDSL的演化

- 生成的Parser代码
 - SLK分析器生成器生成表驱动的分析器代码
 - LL分析的主要操作是“生成”与“匹配”



MetaDSL的演化

➤ MetaDSL的演化

- 生成的Parser代码
 - 符号按值分类
 - 分析表
 - 分析栈

```
1 // DefConstants.cs - generated by the SLK parser generator
2
3 class DefConstants {
4     public const short SEMI_ = 1;
5     public const short LPAREN_ = 2;
6     public const short COMMA_ = 3;
7     public const short RPAREN_ = 4;
8     public const short LBRAKE_ = 5;
9     public const short RBRAKE_ = 6;
10    public const short IDENTIFIER_ = 7;
11    public const short STRING_ = 8;
12    public const short NUMBER_ = 9;
13    public const short END_OF_SLK_INPUT_ = 10;
14
15    public const short NT_EFFECTS_ = 11;
16    public const short NT_EFFECT_ELEMENT_DEF_ = 12;
17    public const short NT_EFFECT_ELEMENT_ = 13;
18    public const short NT_EFFECT_ELEMENT_SINGLE_ = 14;
19    public const short NT_EFFECT_ELEMENT_PARAMS_ = 15;
20    public const short NT_EFFECT_ELEMENT_STATEMENTS_ = 16;
21    public const short NT_NAME_OR_VALUE_ = 17;
22    public const short NT_EFFECT_ELEMENT_DEF_SEMI_STAR_ = 18;
23    public const short NT_EFFECT_ELEMENT_SINGLE_STAR_ = 19;
24    public const short NT_EFFECT_ELEMENT_PARAMS_OPT_ = 20;
25    public const short NT_EFFECT_ELEMENT_STATEMENTS_OPT_ = 21;
26    public const short NT_EFFECT_ELEMENT_DEF_COMMAS_STAR_ = 22;
27    public const short NT_EFFECT_ELEMENT_DEF_SEMI_2_STAR_ = 23;
```

第一个非终结符，比它小的
值是终结符

最后一个非终结符，比它大
的值是语义行为

分析表

```
1 // DefParser.cs - generated by the SLK parser generator
2
3 class DefParser {
4     private static short[] Production = {0,2,11,18 ,4,12,24,13,25 ,3,13,14,19 ,5,14,17,26,20,21
5 ,6,15,2,27,22,4,28 ,6,16,5,29,23,6,30 ,3,17,7,31
6 ,3,17,8,32 ,3,17,9,33 ,4,18,12,1,18 ,1,18 ,3,19,14,19
7 ,1,19 ,2,20,15 ,1,20 ,2,21,16 ,1,21 ,4,22,12,3,22
8 ,1,22 ,4,23,12,1,23 ,1,23 ,0};
9
10    private static int[] Production_row = {0,1,4,9,13,19,26,33,37,41,45,50,52,56,58,61,63
11 ,66,68,73,75,80,0};
12
13    private static short[] Parse = {0,0,15,14,15,5,15,6,15,15,15,17,0,17,13,16,13,17,17
14 ,17,12,12,12,1,1,1,10,10,10,11,19,0,0,18,18,18,21,20
15 ,20,20,2,2,2,3,3,4,4,4,7,8,9,0};
16
17    private static int[] Parse_row = {0,16,34,37,40,3,2,43,20,13,1,10,27,31,0};
18
19    private const short END_OF_SLK_INPUT_ = 10;
20    private const short START_SYMBOL = 11;
21    private const short START_ACTION = 24;
22    private const short END_ACTION = 34;
23
24    public static void
25        parse ( DefAction action,
26                DefToken tokens,
27                DefError error,
28                short start_symbol )
29    {
30        short rhs, lhs;
31        short production_number, entry, symbol, token, new_token;
32        int production_length, top, index, level;
33        short[] stack = new short[512];
34
35        top = 511;
36        stack [ top ] = 0;
37        if ( start_symbol == 0 ) {
38            start_symbol = START_SYMBOL;
39        }
40        if ( top > 0 ) { stack [ --top ] = start_symbol;
41        } else { error.message ("DefParse: stack overflow\n"); return; }
42        token = tokens.get();
43        new_token = token;
44
45        for ( symbol = (stack[top] != 0 ? stack[top++]: (short) 0); symbol != 0; ) {
46            if ( symbol >= START_ACTION ) {
47                action.execute ( symbol - (START_ACTION-1) );
48            } else if ( symbol >= START_SYMBOL ) {
49                entry = 0;
50                level = 1;
51                if ( entry == 0 ) {
52                    index = Parse_row [ symbol - (START_SYMBOL-1) ];
53                    index += token;
54                }
55            }
56        }
57    }
58}
```

开始符号之前是终结符、之后到开始语义行为之间是非终结符

遇到语义行为，执行

遇到非终结符

MetaDSL的演化

➤ MetaDSL的演化

- 生成的Parser代码
 - LL算法又称“生成-匹配”算法，算法里主要的操作是
 - 生成—根据非终结符选择产生式将右部符号压栈
 - 匹配—与输入匹配当前选中的产生式里的终结符
 - LR算法则称“移进-归约”算法，相应的算法主要操作是
 - 移进—将输入移到栈里
 - 归约—根据栈里的符号内容选择产生式并归约到产生式左部

```
56     entry = Parse [ index ];
57 }
58 if ( entry != 0 ) {
59     index = Production_row [ entry ];
60     production_length = Production [ index ] - 1;
61     lhs = Production [ ++index ];
62     if ( lhs == symbol ) {
63         action.predict ( entry );
64         index += production_length;
65         for ( ; production_length-- > 0; --index ) {
66             if ( top > 0 ) { stack [ --top ] = Production [ index ];
67             } else { error.message ( "DefParse: stack overflow\\n" ); return; }
68         }
69     } else {
70         new_token = error.no_entry ( symbol, token, level-1 );
71     }
72 } else {
73     new_token = error.no_entry ( symbol, token, level-1 );
74 }
75 } else if ( symbol > 0 ) {
76     if ( symbol == token ) {
77         token = tokens.get();
78         new_token = token;
79     } else {
80         new_token = error.mismatch ( symbol, token );
81     }
82 } else {
83     error.message ( "\\n parser error: symbol value 0\\n" );
84 }
85 if ( token != new_token ) {
86     if ( new_token != 0 ) {
87         token = new_token;
88     }
89     if ( token != END_OF_SLK_INPUT_ ) {
90         continue;
91     }
92     symbol = (stack[top] != 0 ? stack[top++] : (short) 0);
93 }
94 if ( token != END_OF_SLK_INPUT_ ) {
95     error.input_left ();
96 }
97 }
98 }
99 }
100
101
102
103 
```

匹配，将产生式右边压栈

不匹配，报错

终结符匹配，读下一个词，否则报错

更新当前词，上面非终结符获取产生式时要用到

出栈，进入下一处理循环

语法分析结束，但输入未结束，报错

MetaDSL的演化

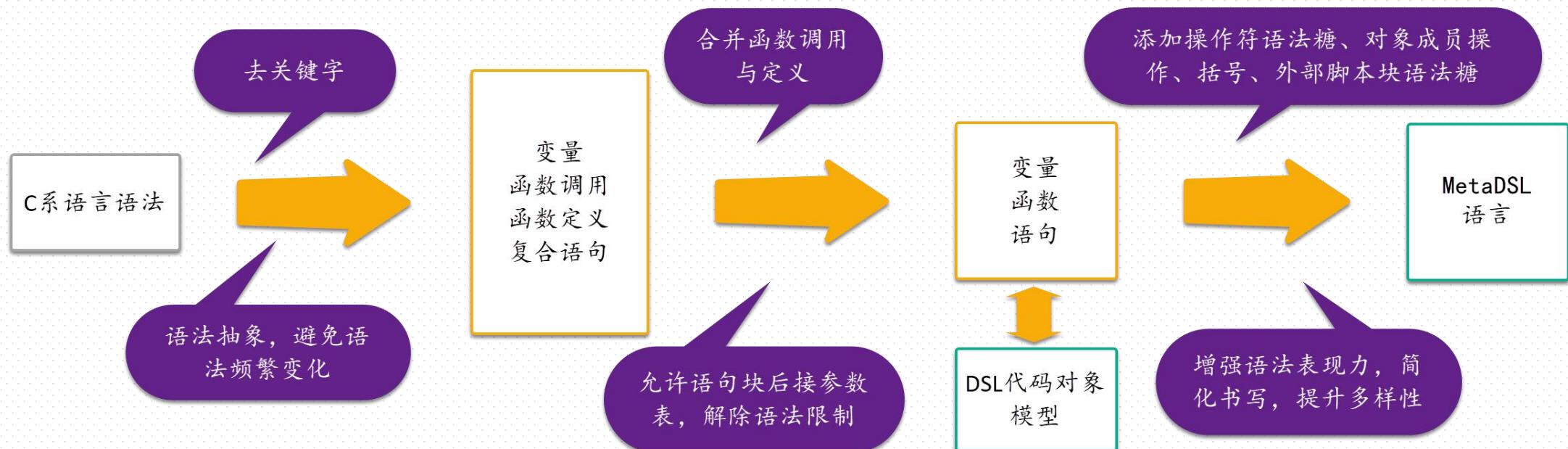
➤ MetaDSL的演化

- 特效的语法部分到此告一段落
- 从特效的语法设计我们可以看到一些通用语法的影子以及通用化语法的方法
- 现在我们回到C系语言语法风格，前面说过C系语言的语法构造，可以抽象出四类
 - 语句
 - 函数定义
 - 函数调用
 - 变量或标识符
- 最后的特效语法其实也是由这些语法组成的
 - 特效元素语法其实是语句（多个语句块部分）与函数定义（一个语句块部分）
 - 特效属性其实是函数调用语法
 - 不带参数的属性其实就是变量的语法
- 我们沿这个思路继续探索就得到了MetaDSL的语法

MetaDSL的演化

➤ MetaDSL的演化

- 我们沿这个思路继续探索就得到了MetaDSL的语法



MetaDSL的演化

➤ MetaDSL的演化

- 抽象化的四种语法结构--递归嵌套结构

- 语句

- 语句 ::= 函数 { 函数 } [分隔符]
 - if(exp){}elseif(exp){}else{};
 - function(name)args(arg1,arg2){};

- 函数 (调用与语句块为可选，但至少需要有一个)

- 函数 ::= [调用] [语句块]
 - 语句块 ::= \{ { 语句 } \}
 - function(name){}
 - while(exp){}

- 调用 (变量与参数表为可选，但至少需要有一个)

- 调用 ::= [变量] [参数表]
 - 参数表 ::= ({ 语句 })

- 变量

- 变量 ::= 任何标识符或常量

- 分隔符(逗号一般用于参数表，分号一般用于语句，但语法上不区分)

- 分隔符 ::= , 或 ;

注意相邻语句间必须有一个分隔符，没有就无法正确区分语句的开始与结束。实践中发现这里书写时很容易犯错的地方，因为C系语言的语句结束是不需要加分号的。这是关键字变成普通标识符后为消除二义性引入的，与C系语言多个表达式语句间需要分号分隔道理是一样的。

请注意，这个语法完全没用关键字！

MetaDSL的演化

➤ MetaDSL的演化

- 这个语法的表达能力已经等同于XML，但相对于C系语言，还是有限制
 - 比如无法表达高阶调用
 - $f(a,b,c)(d,e,f)$
- 支持它需要修改调用语法定义
 - 调用 ::= [变量] { 参数表 }
- 既然参数表可以有高阶的形式，那么语句块部分是否也应该如此？
 - 函数 ::= [调用] { 语句块 }
- 现在完整语法列表如下
 - 语句 ::= 函数 { 函数 } 分隔符
 - 调用 ::= [变量] { 参数表 }
 - 函数 ::= [调用] { 语句块 }
 - 变量 ::= 任何标识符或常量
 - 分隔符 ::= , 或 ;
- ok，这就已经超出C系语言的语法范围了（C系语言的语句块不能有高阶表示）

MetaDSL的演化

➤ MetaDSL的演化

- 我们来看这个语言的一个句子
 - Let's look at a sentence in this language ;
 - 噢？这是一个合法的语句么？
 - ✓ 每个单词是语句的一个函数
 - ✓ 这个函数没有参数表与语句块部分
 - ✓ 语句由多个函数连接而成，最后加上分隔符
 - ✓ 所以确实是合法的
- 再看几个
 - ✓ (x,y,z)from(tuple_list)where(equal(x,1));
 - ✓ (a)(b)(c){d}{e}{f}{g,h,i};
 - ✓ if(a){b;}elseif(c){d;}elseif(e){f;}else{g;};
 - ❑ a(b,c,d){e;f;}(g); 冲突了，看起来是一个函数后面接了一个参数表，而函数后面应该或者是分隔符，或者是另一个函数

MetaDSL的演化

➤ MetaDSL的演化

- 目前为止，我们的语法是从C系语言的结构抽象来的，有一定表达力，也有限制
- 我们来尝试突破刚才的限制
 - 所谓抽象，说白了就是求同存异，也就是忽略一些信息，合并同类项
 - 不过我们已经没有关键字了，所以没法再合并/消除关键字了
 - 那么语法结构呢，我们有四种语法结构
 - 刚才的冲突产生于一个函数后面尝试接一个参数表
 - 参数表是调用的语法
 - 也就是说，函数与调用语法冲突了
 - 那么，**函数和调用可以合并么？**

MetaDSL的演化

➤ MetaDSL的核心语法

- 函数和调用合并后
 - 语句 ::= 函数 { 函数 } 分隔符
 - 函数 ::= [变量] { 参数表 }
 - 变量 ::= 任何标识符或常量
 - 分隔符 ::= , 或 ;
- 其中
 - 参数表 ::= ({ 语句 }) 或 \{ { 语句 } \}
- 这就是MetaDSL的核心语法了
 - 最终语法是三层结构
 - 语句、函数、变量
 - 函数的语法提供灵活性
 - 变量、参数表都是可选的，但至少存在一个
 - 用圆括号括起来的参数列表与用花括号括起来的语句列表是参数表的两种形式
 - 参数表可以是高阶表示

MetaDSL的演化

➤ MetaDSL语法思考

- 为什么语法结构变成三层（语句、函数、变量）了？
 - 可能与我们的世界和语言有关
- 维特根斯坦《逻辑哲学论》
 - 世界是事实的总和
 - 事实是诸事态的存在
 - 事态是诸对象（事物）的一种结合
 - 对象是简单的
 - 对象构成世界的实体。因此它们不可能是组合成的。

世界的三层结构：事实、事态、对象

- 命题的总和就是语言
- 命题符号化事实
- 命题是基本命题的真值函项
- 最简单的命题，即基本命题，断定一个事态的存在
- 基本命题是由名称组成的，是名称的一种联结，是名称的函项
- 名称在命题中代表对象

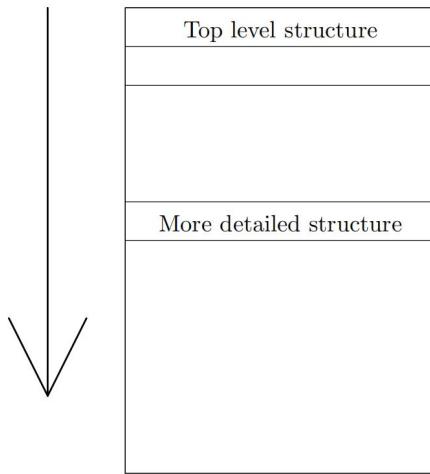
语言的三层结构：命题、基本命题、名称

LOP思想简介

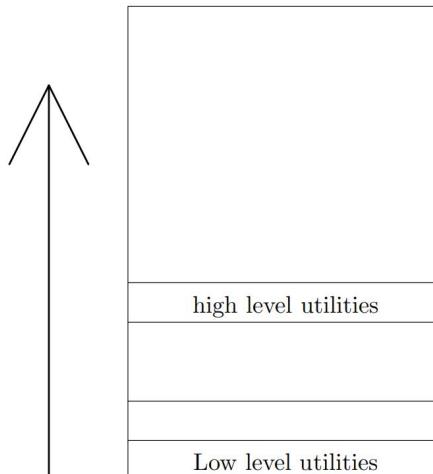
➤ LOP---面向语言编程

- 面向语言编程（LOP）最早提出于1994年[middle-out-t](#)，强调基于语言来提供分层间的抽象，这与操作系统分层虚拟机的观点很相似，每一层抽象都定义一种语言，作为向上一层提供服务的接口
- 四种开发模型

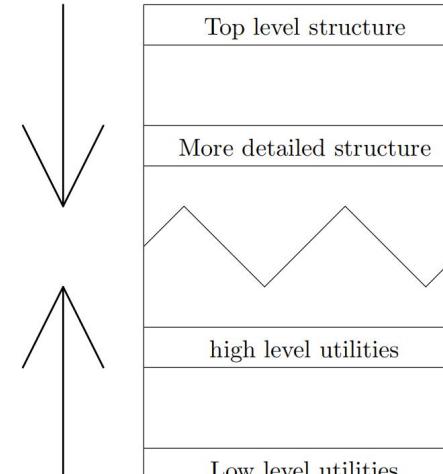
Top Down Development



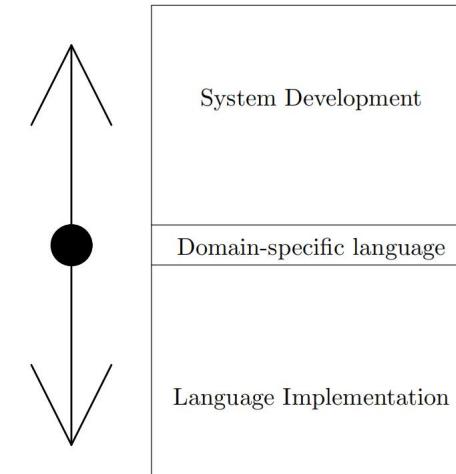
Bottom Up Development



Outside In Development



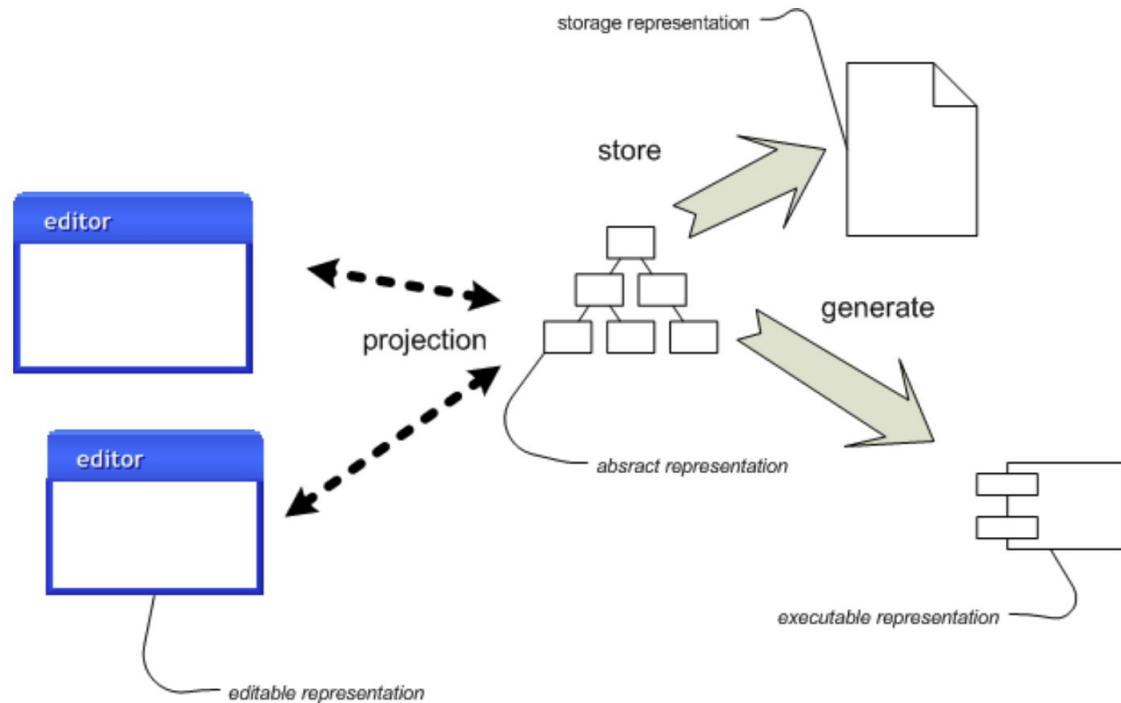
Middle Out Development



LOP思想简介

➤ LOP---面向语言编程

- 2005年[Martin Fowler](#)的[Language Workbenches: The Killer-App for Domain Specific Languages?](#)提出语言工作台可能是DSL语言的一个比较好的形式，并给出了设计DSL的三部分工作
 - Define the abstract syntax, that is the **schema** of the abstract representation.
 - Define an **editor** to let people manipulate the abstract representation through a projection.
 - Define a **generator**. This describes how to translate the abstract representation into an executable representation. In practice the generator defines the semantics of the DSL.
- 微软的design-time工具集（Visual Studio）与意图编程（Intentional programming）
- Unity3d的编辑器功能开发



LOP思想简介

➤ LOP---面向语言编程

- 2018年的这篇文章（[language oriented software engineering](#)）对LOP的开发过程总结为：
 - The language-oriented methodology proceeds in three steps:
 - Design a domain specific language (DSL) for your core application logic.
 - Write your application in the DSL.
 - Build interpreters to execute your DSL programs, mapping your logical constructs to the actual machinery.

第三章 MetaDSL的语法与语义

MetaDSL的语法

➤ MetaDSL核心语法review

- 核心语法
 - 语句、函数、变量
 - 定义
 - 语句 ::= 函数 { 函数 } 分隔符
 - 函数 ::= [变量] { 参数表 }
 - 变量 ::= 任何标识符或常量
 - 分隔符 ::= , 或 ;
 - 其中
 - 参数表 ::= ({ 语句 }) 或 \{ { 语句 } \}

MetaDSL的语法

➤ MetaDSL的语法

- 单有核心语法不是很方便
- 实际MetaDSL的实现增加了许多语法糖与核心语法的变体
 - c#操作符（除as/is）与c++操作符合集
 - 操作符相当于函数调用的中缀写法，语义上就是函数
 - 单参数函数的对象成员写法（语义上也是函数，obj是函数名，member是参数）
 - obj.member、obj[member]、obj->member
 - obj.*member、obj->*member
 - obj?.member、obj?.*member
 - obj::member
 - 函数参数的括号
 - ()、[]、{ }
 - .()、.[]、.{ }、?()、?[]、?{ }
 - (: :)、[: :]、<: :>
 - (% %)、[% %]、{ % %}
 - 外部脚本块作为单一语句块参数(在代码生成里有特别的用处)
 - { : : }

MetaDSL的语法

➤ MetaDSL的语法

- 全部支持运算符见词法分析源码里的getOperatorToken方法
 - <https://github.com/dreamanlan/MetaDSL/blob/master/DslParser/DslToken.cs>
- 完整语法链接：
 - <https://github.com/dreamanlan/MetaDSL/blob/master/DslParser/dsl.txt>
 - 这个是语法生成器SLK的语法定义文件，基于EBNF，其中_action_开头的是语义行为
 - 这个语法定义是LL(1)的，已经进行了消除左递归与提取左因子处理，所以看起来不是太直观

MetaDSL的语法

➤ MetaDSL的语法 • 表达能力

描述数据

DSL嵌套元素表示:

```
Root
{
    Level1(l1attr1(l1val1), l1attr2(l1val2))
    {
        Level2(l2attr1(l2val1), l2attr2(l2val2));
    };
}
```

XML嵌套元素表示:

```
<Root>
    <Level1 l1attr1="l1val1" l1attr2="l1val2">
        <Level2 l2attr1="l2val1" l2attr2="l2val2" />
    </Level1>
</Root>
```

DSL数组表示:

```
array(1,2,3,4,5);
或
[1,2,3,4,5];
```

Json数组表示:

```
[1,2,3,4,5]
```

描述逻辑

DSL对象表示:

```
object
{
    key1(val1);
    key2(val2);
}
或
{
    key1 => val1,
    key2 => val2
};
```

Json对象表示:

```
{
    key1 : val1,
    key2 : val2
}
```

DSL高阶函数调用表示:

```
func(arg1, arg2)(arg11, arg22);
```

lisp高阶函数调用表示:

```
((func arg1 arg2) arg11 arg22)
```

C语句表示:

```
if(a > b){
    print(a, b);
}else{
    print(b, a);
};
```

C语句表示:

```
if(a>b){
    print(a, b);
}else{
    print(b, a);
};
```

MetaDSL的语法

➤ MetaDSL的语法

- 乔姆斯基文法与自动机
 - 0型文法（短语文法） \Leftrightarrow 图灵机 \Leftrightarrow 递归函数
 - 1型文法（上下文相关文法） \Leftrightarrow 线性有界自动机
 - 2型文法（上下文无关文法） \Leftrightarrow 下推自动机
 - 3型文法（左/右线性文法[正规文法]） \Leftrightarrow 有限状态自动机 \Leftrightarrow 正则表达式
- 文法的识别能力 $0 \rightarrow 3$ 递减
 - 这里的识别能力实际上是指识别一个语言不符合文法的能力
 - 并不是指符合文法的语言的范围，范围正好是相反的
 - 甚至类比程序语言的抽象概念，能力越弱的越抽象
 - 在类型系统里，约束最少的类型最抽象
 - 子类型是在基类型基础上添加更多约束形成的
 - 子类型实例是基类型实例的子集
 - 从这个意义上讲，2型文法比1型文法更抽象
 - 符合1型文法的语言也符合规则更少的2型文法
 - 也就是说2型文法其实可以识别符合1型文法的语言
 - 但2型文法不能拒绝符合2型文法但不符合1型文法的语言
 - 其它类似

MetaDSL的语法

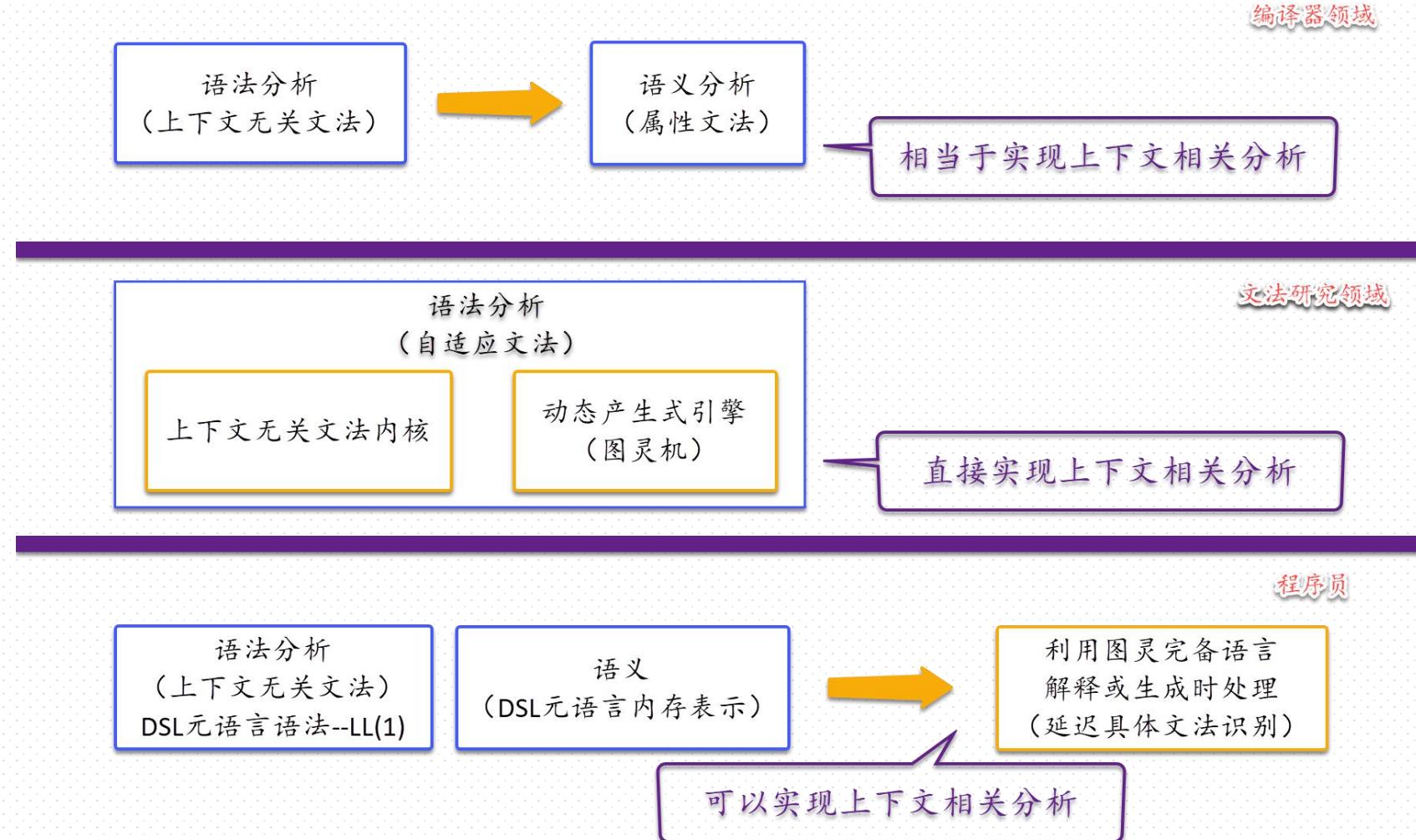
➤ MetaDSL的语法

- MetaDSL文法思考
 - 静态文法
 - CFG（上下文无关文法）
 - 属性文法（一般认为是语义处理，也可以认为是通过扩展CFG提供了上下文相关的解析能力）
 - 适应性文法/动态文法（通过在解析时修改文法来提供更强大的解析能力）
 - MetaDSL文法是实际DSL语言文法的抽象，类似抽象类型与具体类型的关系
 - 相当于参数化实际文法的关键字
 - 关键字=>普通标识符=>多个产生式合并为一个
 - 也表现了一定的适应性
 - 语法分析时进行抽象文法的处理
 - 在语义行为里或更后期进行具体文法的识别与处理
 - 处理逻辑使用图灵完备的语言，理论上可以更加灵活

MetaDSL的语法

➤ MetaDSL的语法

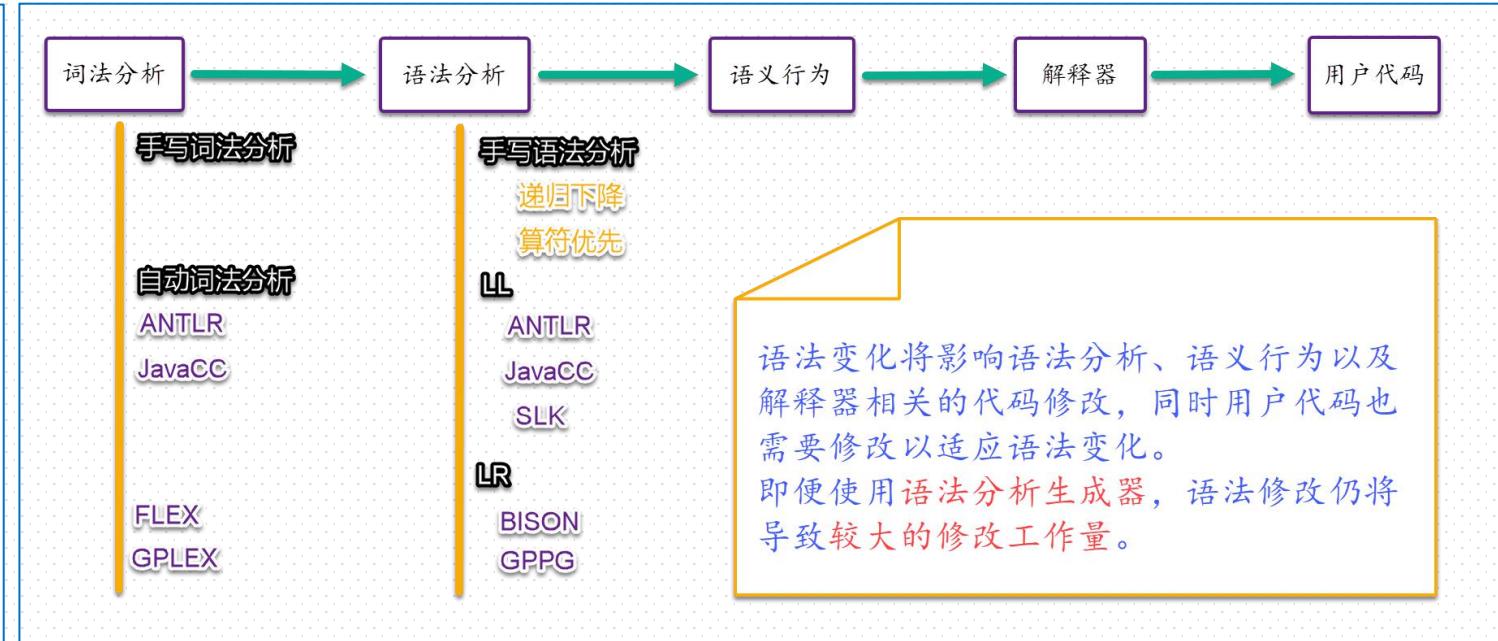
- MetaDSL文法思考



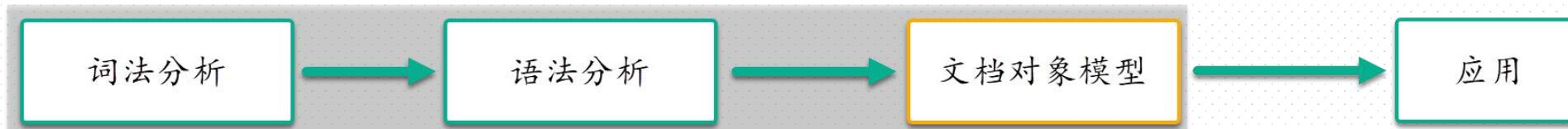
MetaDSL的语义

➤ MetaDSL作为DSL语言 开发工具

- ◆ 省去编写DSL语言的语法分析
- ◆ 避免DSL语言的语法变更
- ◆ 借鉴XML/JSON思想
 - ◆ 文档对象模型



Xml: 元素、属性、子元素

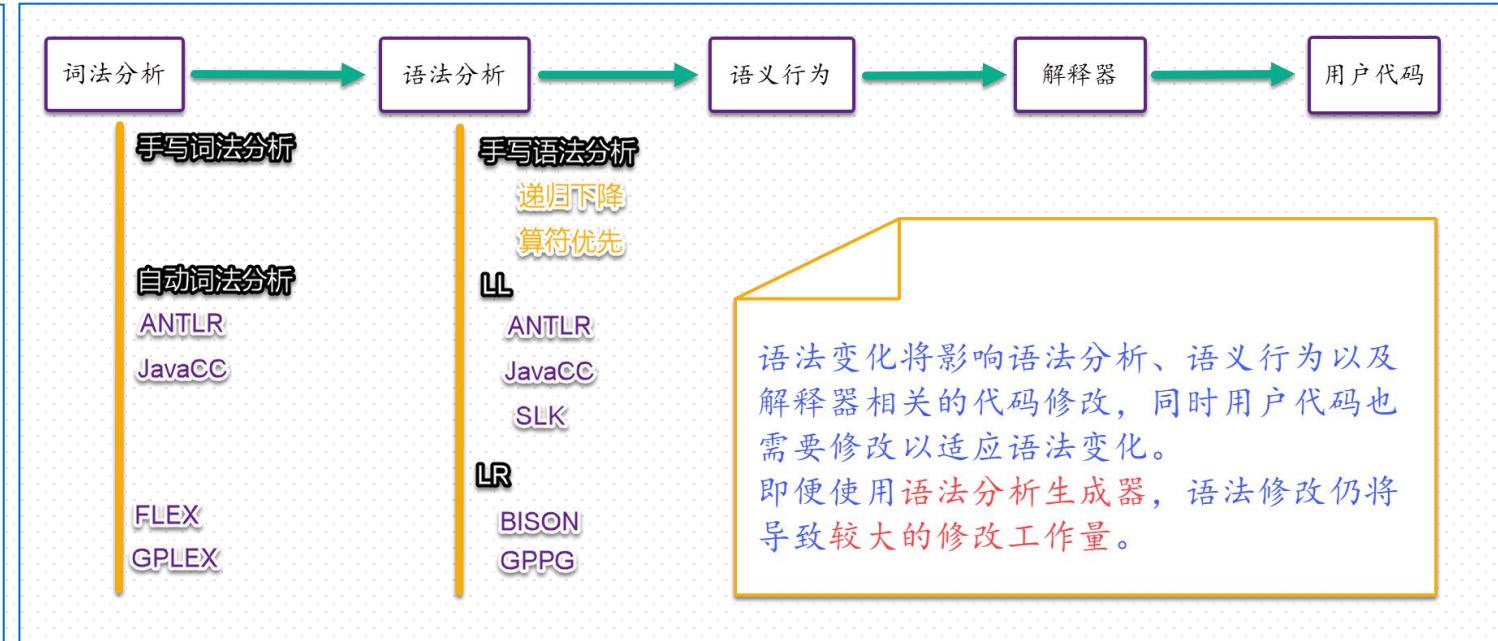


Json: 数组、对象

MetaDSL的语义

➤ MetaDSL作为DSL语言 开发工具

- ◆ 省去编写DSL语言的语法分析
- ◆ 避免DSL语言的语法变更
- ◆ 借鉴XML/JSON思想
 - ◆ 文档对象模型
- ◆ 代码对象模型



MetaDSL: 变量、函数、语句

MetaDSL的语义

➤ MetaDSL的语义

- MetaDSL使用流程



MetaDSL的语义

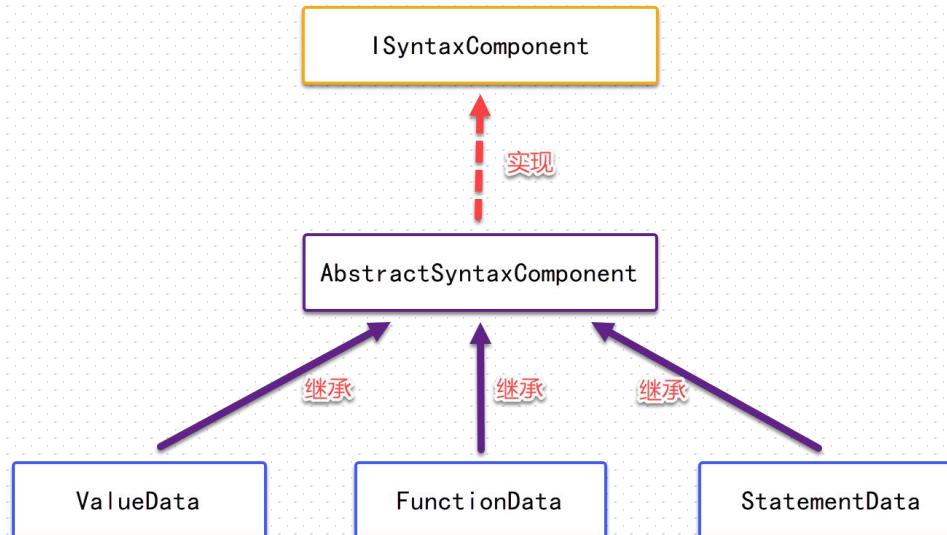
➤ MetaDSL的语义

- 这里的语义与传统语言语义不太一样，因为MetaDSL是作为语言基础提供的
 - 语法解析后得到的内存表示我们称为语义
 - 容易误认为是抽象语法树
 - MetaDSL的语义与核心语法对应
 - 语句、函数、变量
- MetaDSL的所有语法在解析后都对应到三个语义class，表示了三个核心语法的信息
 - StatementData
 - FunctionData
 - ValueData
- 三种语义类都从一个接口派生ISyntaxCompoent
 - <https://github.com/dreamanlan/MetaDSL/blob/master/Dsl.cs>
- 另外有一个加载类用于从文件读取DSL信息
 - DslFile

MetaDSL的语义

➤ MetaDSL的内存表示与使用

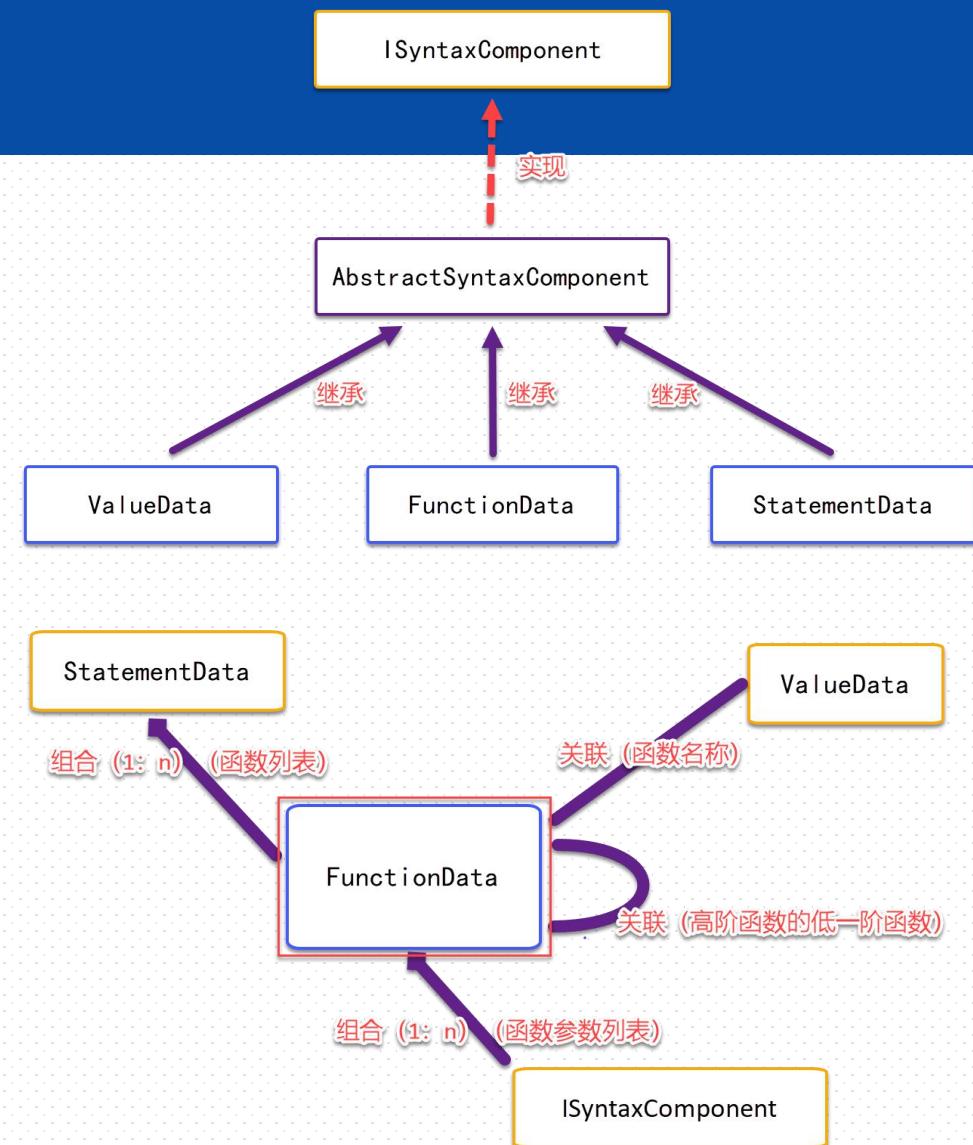
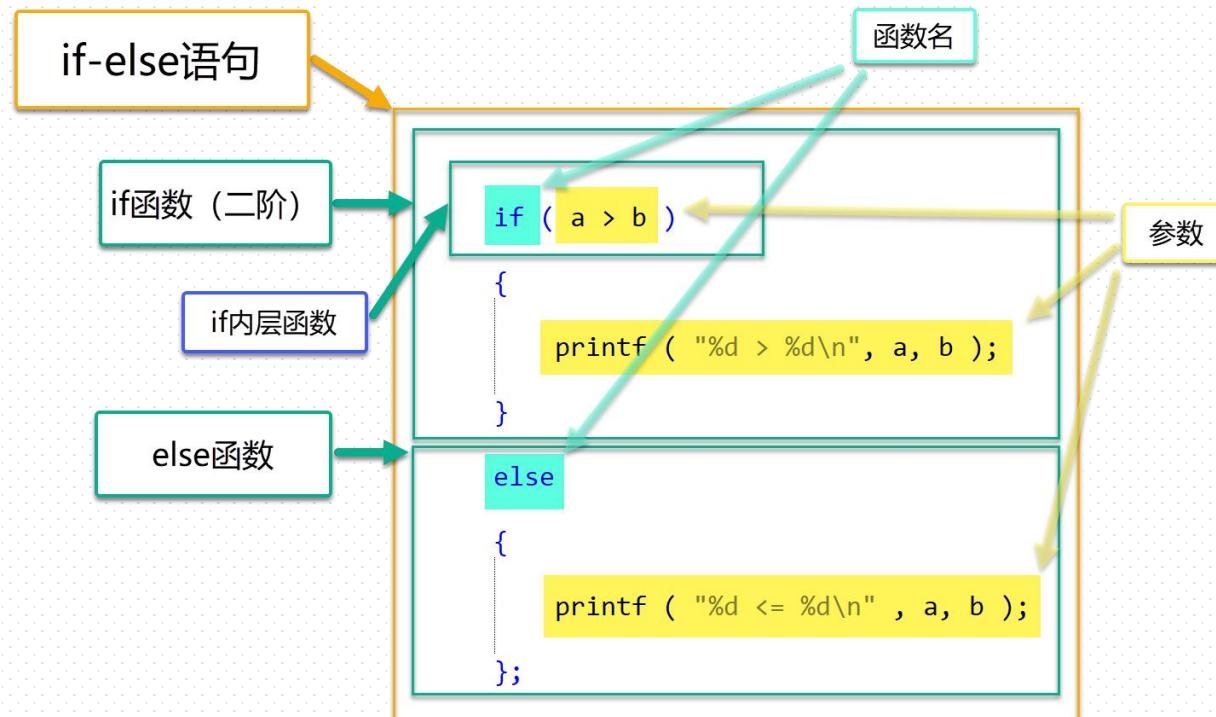
- ◆ 加载MetaDSL得到与核心语法对应的代码对象模型
 - 语句、函数、变量



MetaDSL的语义

➤ MetaDSL的内存表示与使用

- ◆ 加载MetaDSL得到与核心语法对应的代码对象模型
 - 语句、函数、变量



MetaDSL的语义

➤ MetaDSL的内存表示与使用

◆ 加载MetaDSL得到与核心语法对应的代码对象模型

- 语句、函数、变量
- 可解释性
 - DSL代码对象模型面向表示
 - DSL解释器的开发是基于DSL开发的一部分

◆ 通用脚本语言（如Lua）通常不提供语法内存表示

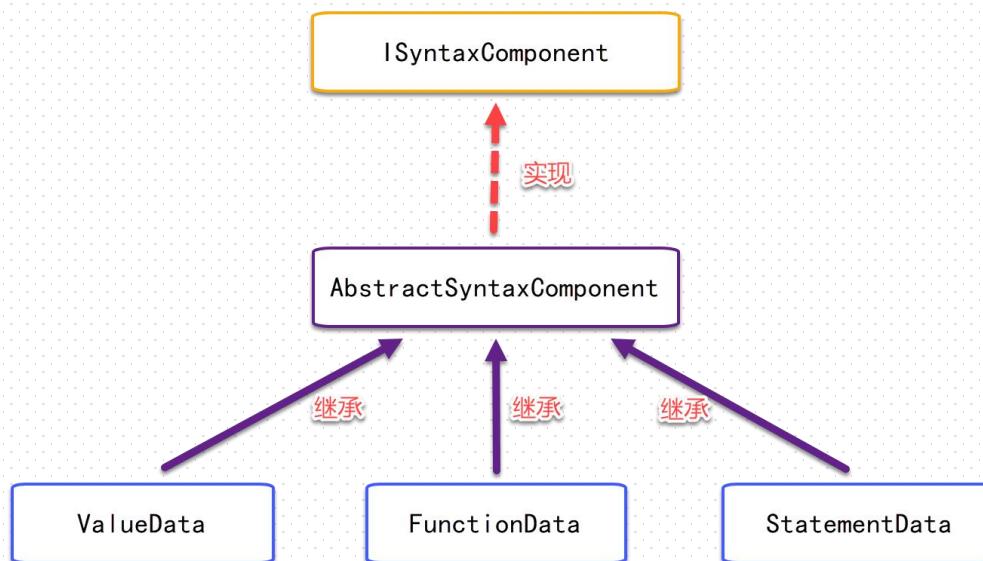
- 嵌入脚本虚拟机
- 访问运行时数据（而非语法内存表示）
- GPL是语义明确的语言，面向执行而非表示



MetaDSL的语义

➤ MetaDSL的语义

- ISyntaxComponent定义了所有语法组件的抽象接口
 - IsValid() 语法是否有效
 - GetId() 获取id
 - GetIdType() 获取id的类型
 - GetLine() 源码里的行号（不一定有效）
 - ToScriptString(bool) 转换为源码字符串



```
/// <summary> 基于函数样式的脚本化数据解析工具。可以用作DSL元语言。
public interface ISyntaxComponent
{
    bool IsValid();
    string GetId();
    int GetIdType();
    int GetLine();
    string ToScriptString(bool includeComment);

    string CalcFirstComment();
    string CalcLastComment();
    void CopyComments(ISyntaxComponent other);
    void CopyFirstComments(ISyntaxComponent other);
    void CopyLastComments(ISyntaxComponent other);
    List<string> FirstComments { get; }
    bool FirstCommentOnNewLine { get; set; }
    List<string> LastComments { get; }
    bool LastCommentOnNewLine { get; set; }
}
```

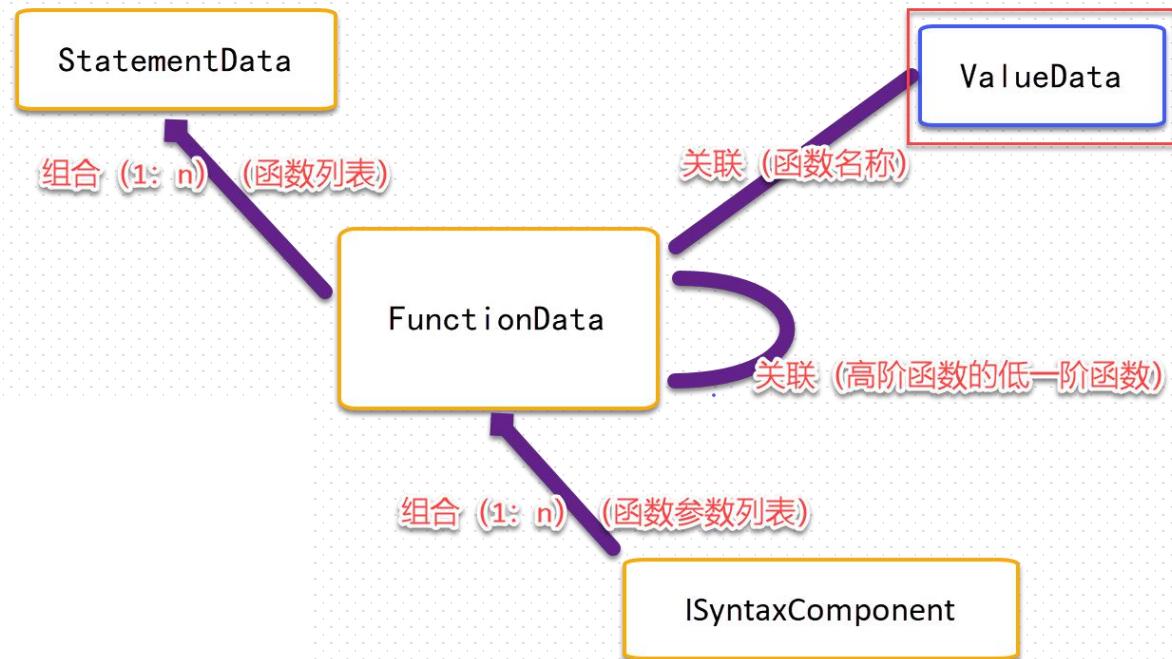
MetaDSL的语义

➤ MetaDSL的语义

- 变量--ValueData
 - 常量值
 - 函数名称
- MetaDSL里只有三种类型
 - 标识符
 - 数字
 - 字符串

```
public const int ID_TOKEN = 0;  
public const int NUM_TOKEN = 1;  
public const int STRING_TOKEN = 2;  
public const int MAX_TYPE = 2;
```

```
public abstract bool IsValid();  
public abstract string GetId();  
public abstract int GetIdType();  
public abstract int GetLine();  
public abstract string ToScriptString(bool includeComment);
```



MetaDSL的语义

➤ MetaDSL的语义

- 变量——ValueData
 - 变量的数据
 - 数值或名称（都用字符串记录）
 - 类型
 - 如果DSL是从文本加载，还保留行号信息



```
/// <summary> 用于描述变量、常量与无参命令语句。可能会出现在函数调用参数表与函数语句列表中。
public class ValueData : AbstractSyntaxComponent
{
    public override bool IsValid()...
    public override string GetId()...
    public override int GetIdType()...
    public override int GetLine()...
    public override string ToScriptString(bool includeComment)...

    public bool HaveId()...
    public void SetId(string id)...
    public void SetType(int _type)...
    public void SetLine(int line)...
    public bool IsId()...
    public bool IsNumber()...
    public bool IsString()...
    public void Clear()...
    public void CopyFrom(ValueData other)...

    public ValueData()...
    public ValueData(string val)...
    public ValueData(string val, int _type)...

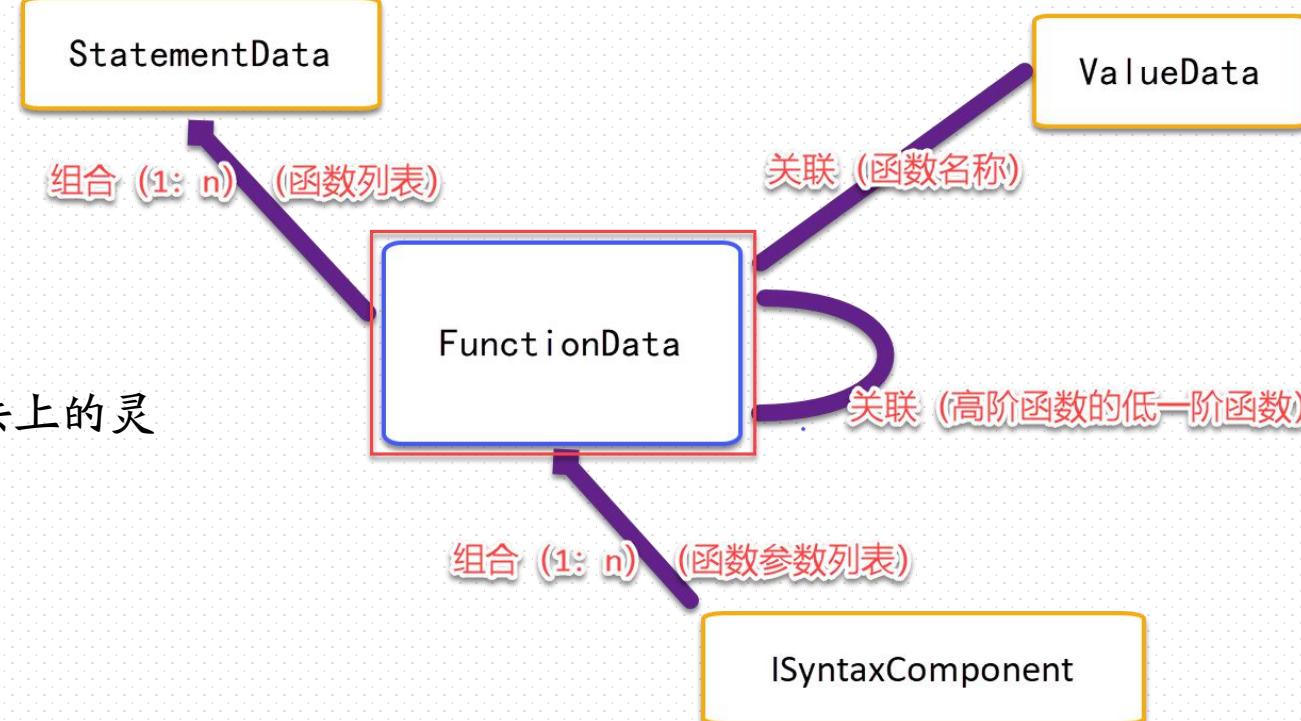
    private int m_Type = ID_TOKEN;
    private string m_Id = string.Empty;
    private int m_Line = -1;

    public static ValueData NullValue...
    private static ValueData s_Instance = new ValueData();
}
```

MetaDSL的语义

➤ MetaDSL的语义

- 函数——FunctionData
 - 函数名
 - 高阶函数表示
 - 参数列表
- 语句由函数列表构成
- 函数是最复杂与灵活的，语法上的灵活变化基本上都是由函数提供



MetaDSL的语义

➤ MetaDSL的语义

- 函数——FunctionData
 - FunctionData的数据
 - 函数名
 - 参数列表
 - 参数类型
 - 如果是高阶函数，则函数名为空
 - LowerOrderFunction是低一阶的函数
 - 递归直到最终的非高阶函数

```
/// <summary> 函数数据，可能出现在函数头、参数表中。</summary>
public class FunctionData : AbstractSyntaxComponent
{
    public enum ParamClassEnum...
    public override bool IsValid()...
    public override string GetId()...
    public override int GetIdType()...
    public override int GetLine()...
    public override string ToScriptString(bool includeComment)...

    public string CalcComment()...
    public List<string> Comments...

    public List<ISyntaxComponent> Params...
    public bool IsHighOrder...
    public ValueData Name...
    public FunctionData LowerOrderFunction...
    public FunctionData ThisOrLowerOrderCall...
    public FunctionData ThisOrLowerOrderBody...
    public FunctionData ThisOrLowerOrderScript...
    public bool HaveLowerOrderParam()...
    public bool HaveLowerOrderStatement()...
    public bool HaveLowerOrderExternScript()...
    public bool HaveId()...
    public void SetParamClass(int type)...
    public int GetParamClass()...
    public bool HaveParamOrStatement()...
    public bool HaveParam()...
    public bool HaveStatement()...
    public bool HaveExternScript()...
    public int GetParamNum()...
    public void SetParam(int index, ISyntaxComponent data)...
    public ISyntaxComponent GetParam(int index)...
    public string GetParamId(int index)...
    public void ClearParams()...
    public void AddParam(string id)...
    public void AddParam(string id, int type)...
    public void AddParam(ValueData param)...
    public void AddParam(FunctionData param)...
    public void AddParam(StatementData param)...
    public void AddParam(ISyntaxComponent param)...
    public void Clear()...
    public void CopyFrom(FunctionData other)...
    private void PrepareParams()...
    internal override SyntaxComponentCommentsInfo GetCommentsInfo()...
    private FunctionCommentsInfo GetFunctionCommentsInfo()...

    private bool m_IsHighOrder = false;
    private ValueData m_Name = null;
    private FunctionData m_LowerOrderFunction = null;
    private List<ISyntaxComponent> m_Params = null;
    private int m_ParamClass = (int)ParamClassEnum.PARAM_CLASS_NOTHING;

    private FunctionCommentsInfo m_CommentsInfo = null;

    public static FunctionData NullFunction...
    private static FunctionData s_Instance = new FunctionData();
}
```

MetaDSL的语义

➤ MetaDSL的语义

- 函数--FunctionData
 - 参数类型

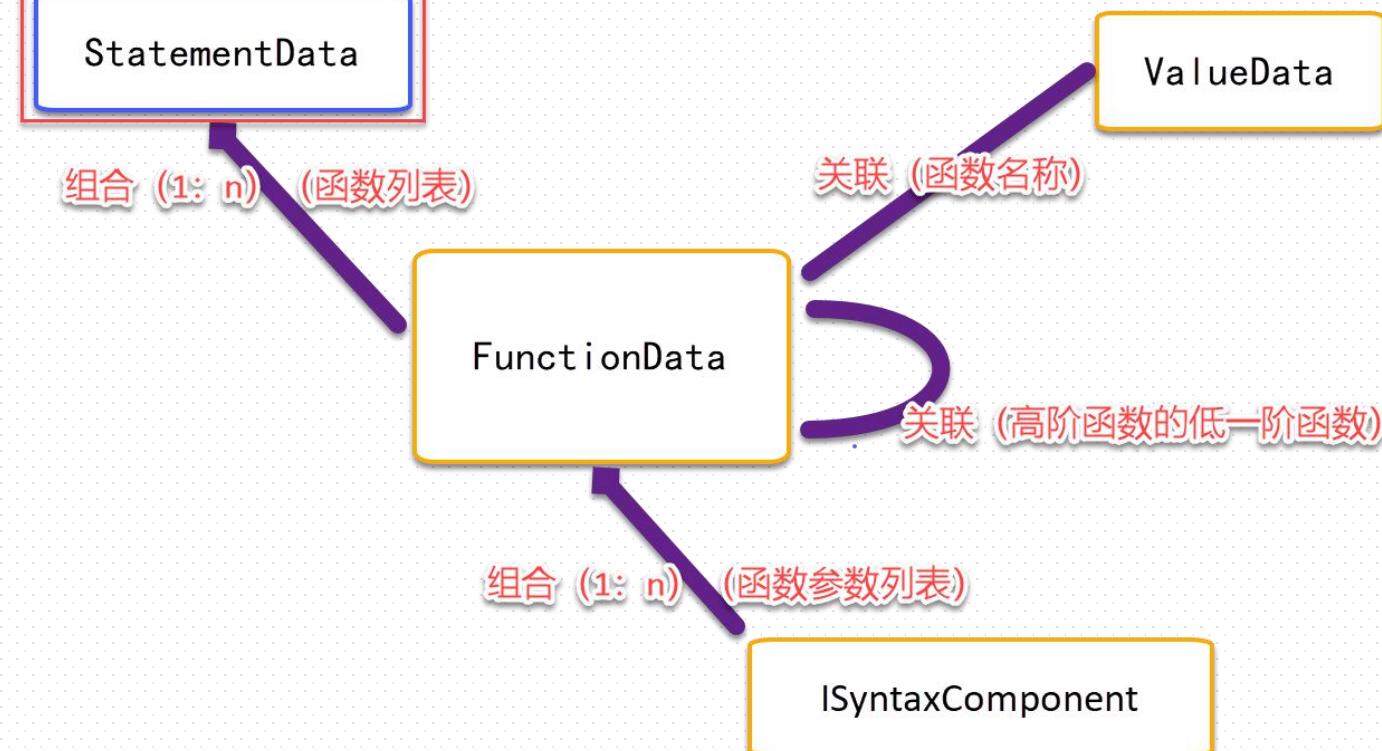
- FunctionData有一个ParamClassEnum，指明了前面提到的各类括号语法或者标明函数名是操作符

```
public enum ParamClassEnum
{
    PARAM_CLASS_MIN = 0,
    PARAM_CLASS_NOTHING = PARAM_CLASS_MIN,
    PARAM_CLASS_PARENTHESIS,
    PARAM_CLASS_BRACKET,
    PARAM_CLASS_PERIOD,
    PARAM_CLASS_PERIOD_PARENTHESIS,
    PARAM_CLASS_PERIOD_BRACKET,
    PARAM_CLASS_PERIOD_BRACE,
    PARAM_CLASS_QUESTION_PERIOD,
    PARAM_CLASS_QUESTION_PARENTHESIS,
    PARAM_CLASS_QUESTION_BRACKET,
    PARAM_CLASS_QUESTION_BRACE,
    PARAM_CLASS_POINTER,
    PARAM_CLASS_STATEMENT,
    PARAM_CLASS_EXTERN_SCRIPT,
    PARAM_CLASS_PARENTHESIS_COLON,
    PARAM_CLASS_BRACKET_COLON,
    PARAM_CLASS_ANGLE_BRACKET_COLON,
    PARAM_CLASS_PARENTHESIS_PERCENT,
    PARAM_CLASS_BRACKET_PERCENT,
    PARAM_CLASS_BRACE_PERCENT,
    PARAM_CLASS_ANGLE_BRACKET_PERCENT,
    PARAM_CLASS_COLON_COLON,
    PARAM_CLASS_COLON_COLON_PARENTHESIS,
    PARAM_CLASS_COLON_COLON_BRACKET,
    PARAM_CLASS_COLON_COLON_BRACE,
    PARAM_CLASS_PERIOD_STAR,
    PARAM_CLASS_QUESTION_PERIOD_STAR,
    PARAM_CLASS_POINTER_STAR,
    PARAM_CLASS_OPERATOR,
    PARAM_CLASS_TERNARY_OPERATOR,
    PARAM_CLASS_MAX,
    PARAM_CLASS_WRAP_INFIX_CALL_MASK = 0x20,
    PARAM_CLASS_UNMASK = 0x1F,
}
```

MetaDSL的语义

➤ MetaDSL的语义

- 语句——StatementData
- 语句在结构上比较简单，就是一个函数列表



MetaDSL的语义

➤ MetaDSL的语义

- 语句——StatementData
 - 语句的数据就是一个函数列表
 - 主要接口都是添加与获取函数的，提供了访问第一、二、三与最后一个函数的便捷方式



```
/// <summary> 语句数据，由多个函数项连接而成。
public class StatementData : AbstractSyntaxComponent
{
    public override bool IsValid()...
    public override string GetId()...
    public override int GetIdType()...
    public override int GetLine()...
    public override string ToScriptString(bool includeComment)...}

    public int GetFunctionNum()...
    public void SetFunction(int index, FunctionData funcData)...
    public FunctionData GetFunction(int index)...
    public string GetFunctionId(int index)...
    public void AddFunction(FunctionData funcData)...
    public List<FunctionData> Functions...
    public FunctionData First...
    public FunctionData Second...
    public FunctionData Third...
    public FunctionData Last...
    public void Clear()...
    public void CopyFrom(StatementData other)...
    internal override SyntaxComponentCommentsInfo GetCommentsInfo()...

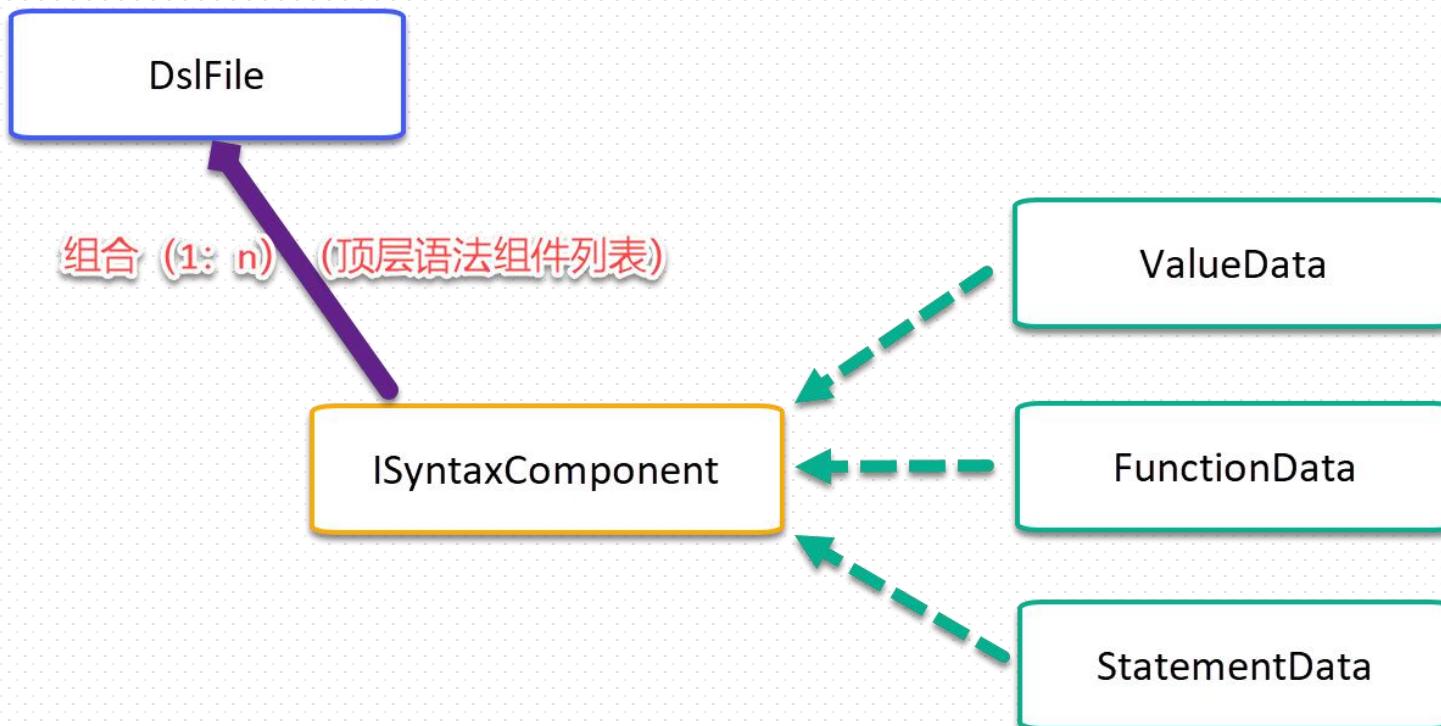
    private List<FunctionData> m_Functions = new List<FunctionData>();
    private SyntaxComponentCommentsInfo m_CommentsInfo = null;

    public static StatementData NullStatementData...
    private static StatementData s_NullStatementData = new StatementData();
}
```

MetaDSL的语义

➤ MetaDSL的语义

- MetaDSL文件—DslFile
 - 顶层语言结构信息列表



MetaDSL的语义

➤ MetaDSL的语义

- MetaDSL文件—DslFile

- 主要提供功能

- 顶层语言结构信息列表
 - 加载与解析
 - 保存（全功能版本包含）

```
public class DslFile
{
    public List<ISyntaxComponent> DslInfos...
    public void AddDslInfo(ISyntaxComponent data)...

    public bool Load(string file, DslLogDelegation logCallback)...
    public bool LoadFromString(string content, string resourceName, DslLogDelegation logCallback)...
    public void Save(string file)...
    public string ToScriptString()...

    public void LoadBinaryFile(string file)...
    public void LoadBinaryCode(byte[] binaryCode)...
    public void SaveBinaryFile(string file)...

    public bool LoadLua(string file, DslLogDelegation logCallback)...
    public bool LoadLuaFromString(string content, string resourceName, DslLogDelegation logCallback)...

    public bool LoadCpp(string file, DslLogDelegation logCallback)...
    public bool LoadCppFromString(string content, string resourceName, DslLogDelegation logCallback)...

    private void WriteInt(Stream s, int val)...
    private void Write7BitEncodedInt(Stream s, int val)...
    private int ReadInt(byte[] bytes, int pos)...
    private int Read7BitEncodedInt(byte[] bytes, int pos, out int byteCount)...

    private class MyStringComparer : IComparer<string>
    {
        public int Compare(string x, string y)...
    }

    private byte[] mBuffer = null;
    private MyStringComparer mStringComparer = null;
    private List<ISyntaxComponent> mDslInfos = new List<ISyntaxComponent>();

    public static byte[] BinaryIdentity...
    public static bool DontLoadComments...
    public static bool IsBinaryDsl(byte[] data, int start)...

    public const string c_BinaryIdentity = "BDSL";

    private static byte[] sBinaryIdentity = null;
    private static bool sDontLoadComments = false;
};
```

MetaDSL的语义

➤ MetaDSL的语义

- 加载与解析一个DSL的常用方式
 - 日志回调
 - Load/LoadFromString
 - Save
 - LoadBinaryFile/LoadBinaryCode
 - SaveBinaryFile
- DSL库分为全版本与发布版本
 - 全版本包含输出脚本的功能，包括文本与二进制2种格式

```
1  DslLogDelegation logCallback = (string msg) => {
2      Console.WriteLine("{0}", msg);
3  };
4  //DslFile.DontLoadComments = true;
5  DslFile file = new DslFile();
6  file.Load("test.txt", logCallback);
7  #if FULL_VERSION
8      file.Save("copy.txt");
9      file.SaveBinaryFile("binary.txt");
10 #endif
11 file.DslInfos.Clear();
12 var code = File.ReadAllBytes("binary.txt");
13 file.LoadBinaryCode(code);
14 #if FULL_VERSION
15     file.Save("unbinary.txt");
16 #endif
```

MetaDSL的语义

➤ MetaDSL的语义

- 加载后如何遍历语义信息
 - 每个DSL文件包含若干DslInfo
 - DslInfo是一个ISyntaxComponent
 - 根据其类型分别按语句、函数、变量三类处理

```
1  private static void LoadDsl()
2  {
3      var dslFile = new Dsl.DslFile();
4      if (dslFile.Load(file, msg => { Console.WriteLine(msg); })) {
5          foreach (var info in dslFile.DslInfos) {
6              ReadConfig(info);
7          }
8      }
9  }
10 private static void ReadConfig(Dsl.ISyntaxComponent cfgInfo)
11 {
12     Dsl.ValueData vd = cfgInfo as Dsl.ValueData;
13     Dsl.FunctionData fd = cfgInfo as Dsl.FunctionData;
14     Dsl.StatementData sd = cfgInfo as Dsl.StatementData;
15     if(null!=sd){
16         ReadStatementData(sd);
17     }
18     else if(null!=fd){
19         ReadFunctionData(fd);
20     }
21     else if(null!=vd){
22         ReadValueData(vd);
23     }
24 }
25 private static void ReadValueData(Dsl.ValueData vd)
26 {
27 }
28 private static void ReadFunctionData(Dsl.FunctionData fd)
29 {
30 }
31 private static void ReadStatementData(Dsl.StatementData sd)
32 {
33 }
```

MetaDSL的语义

➤ MetaDSL的语义

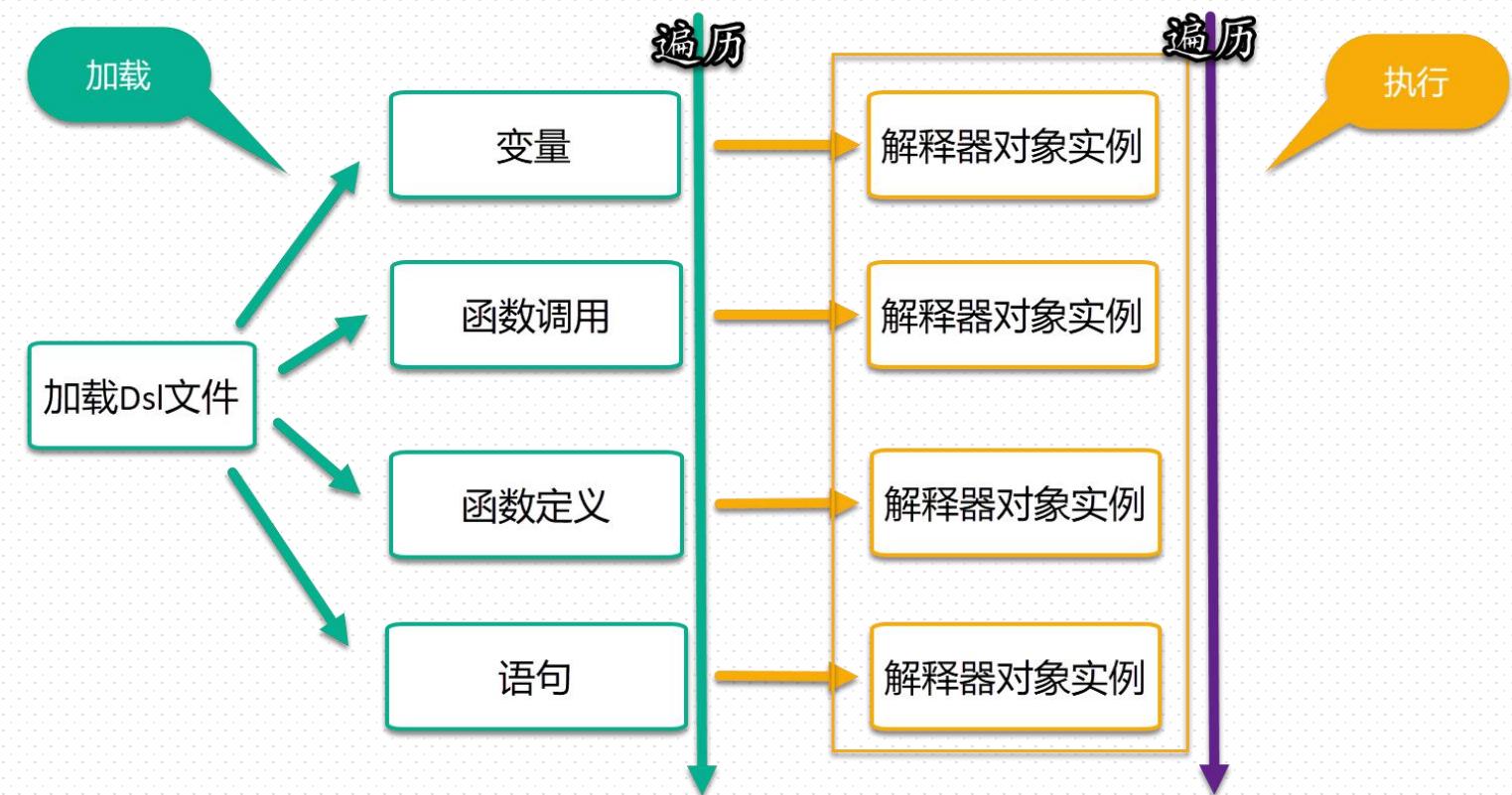
- 加载后如何遍历语义信息
 - 语句、函数、变量三者间也是互相嵌套的关系，在具体加载时也可能互相调用

```
1     private static void ReadValueData(Dsl.ValueData vd)
2     {
3         string id = vd.GetId();
4         vd.GetIdType();
5     }
6
7     private static void ReadFunctionData(Dsl.FunctionData fd)
8     {
9         if(fd.IsHighOrder){
10            ReadFunctionData(fd.LowerOrderFunction);
11        }
12        else{
13            ReadValueData(fd.Name);
14        }
15        if(fd.HaveParam()){
16            //处理参数表
17            foreach(var p in fd.Params){
18                ReadParam(p);
19            }
20        }
21        else if(fd.HaveStatement()){
22            //处理语句表
23            foreach(var p in fd.Params){
24                ReadStatement(p);
25            }
26        }
27    }
28    private static void ReadStatementData(Dsl.StatementData sd)
29    {
30        foreach(var f in sd.Functions){
31            ReadFunctionData(f);
32        }
33    }
```

MetaDSL的语义

➤ MetaDSL的语义

- 加载实例解析
 - DslCalculator
 - <https://github.com/dreamanlan/BatchCommand>



MetaDSL的语义

➤ MetaDSL的语义

- 加载实例解析
 - DslCalculator
 - <https://github.com/dreamanlan/BatchCommand>
- 表达式API接口
 - 加载DSL
 - 执行计算
- 表达式工厂
 - 创建表达式实例
 - 通用工厂

```
public interface IExpression
{
    CalculatorValue Calc();
    bool Load(Dsl.ISyntaxComponent dsl, DslCalculator calculator);
}

public interface IExpressionFactory
{
    IExpression Create();
}

public sealed class ExpressionFactoryHelper<T> : IExpressionFactory where T : IExpression, new()
{
    public IExpression Create()
    {
        return new T();
    }
}
```

MetaDSL的语义

➤ MetaDSL的语义

- 三个层次惯用法
 - 接口
 - 定义抽象
 - 抽象类
 - 实现公共逻辑
 - 简化具体类实现
 - 具体类
 - 具体功能实现
- 抽象API基类
 - 实现dsl加载的模板过程
 - 提供更简化的加载接口

```
public abstract class AbstractExpression : IExpression
{
    public CalculatorValue Calc()
    {
        CalculatorValue ret = CalculatorValue.NullObject;
        try {
            ret = DoCalc();
        }
        catch (Exception ex) {
            var msg = string.Format("calc:[{0}]", ToString());
            throw new Exception(msg, ex);
        }
        return ret;
    }

    public bool Load(Dsl.ISyntaxComponent dsl, DslCalculator calculator)
    {
        m_Calculator = calculator;
        m_Dsl = dsl;
        Dsl.ValueData valueData = dsl as Dsl.ValueData;
        if (null != valueData) {
            return Load(valueData);
        }
        else {
            Dsl.FunctionData funcData = dsl as Dsl.FunctionData;
            if (null != funcData) {
                if (funcData.HaveParam()) {
                    var callData = funcData;
                    bool ret = Load(callData);
                    if (!ret) {
                        int num = callData.GetParamNum();
                        List<IExpression> args = new List<IExpression>();
                        for (int i = 0; i < num; i++) {
                            IExpression arg = Load(callData.GetParam(i));
                            args.Add(arg);
                        }
                        callData.Arguments = args;
                    }
                }
            }
        }
    }
}
```

MetaDSL的语义

➤ MetaDSL的语义

- 抽象API基类
 - 普通函数调用样式API加载
 - Load(Ilist<Iexpression> exps)
 - 语句类API加载，按三类语法类型分别重载处理
 - Load(Dsl.ValueData vd)
 - Load(Dsl.FunctionData fd)
 - Load(Dsl.StatementData sd)

```
        for (int ix = 0; ix < num; ++ix) {
            Dsl.ISyntaxComponent param = callData.GetParam(ix);
            args.Add(calculator.Load(param));
        }
        return Load(args);
    }
    return ret;
}
else {
    return Load(funcData);
}
}
else {
    Dsl.StatementData statementData = dsl as Dsl.StatementData;
    if (null != statementData) {
        return Load(statementData);
    }
}
return false;
}
public override string ToString()
{
    return string.Format("{0} line:{1}", base.ToString(), m_Dsl.GetLine());
}
protected virtual bool Load(Dsl.ValueData valData) { return false; }
protected virtual bool Load(IList<IExpression> exps) { return false; }
protected virtual bool Load(Dsl.FunctionData funcData) { return false; }
protected virtual bool Load(Dsl.StatementData statementData) { return false; }
protected abstract CalculatorValue DoCalc();

protected DslCalculator Calculator
{
    get { return m_Calculator; }
}

private DslCalculator m_Calculator = null;
private Dsl.ISyntaxComponent m_Dsl = null;
```

MetaDSL的语义

➤ MetaDSL的语义

- 最常见的函数API基类
 - 实现加载过程
 - 子类直接实现计算过程即可

```
public abstract class SimpleExpressionBase : AbstractExpression
{
    protected override CalculatorValue DoCalc()
    {
        var operands = Calculator.NewCalculatorValueList();
        for (int i = 0; i < m_Exps.Count; ++i) {
            var v = m_Exps[i].Calc();
            operands.Add(v);
        }
        var r = OnCalc(operands);
        Calculator.RecycleCalculatorValueList(operands);
        return r;
    }
    protected override bool Load(IList<IExpression> exps)
    {
        m_Exps = exps;
        return true;
    }
    protected abstract CalculatorValue OnCalc(IList<CalculatorValue> operands);

    private IList<IExpression> m_Exps = null;
}
```

MetaDSL的语义

➤ MetaDSL的语义

- [BatchCommand 演示](#)
- C#+DSL实现
- Windows/Mac同源码批处理
- DslCalculator通用DSL脚本解释
 - 描述+执行用法的可重用执行部分
 - 已用于前面的地形生成
 - 将用于后面的资源处理

```
C:\WINDOWS\system32\cmd.exe
paused=False
plugindir=C:\Code\Client.\Tools\Batch/.../proj.unity/Assets/Plugins/game
SESSIONNAME=Console
LOCALAPPDATA=C:\Users\dreamanlan\AppData\Local
CommonProgramW6432=C:\Program Files\Common Files
monopath=/Library/Frameworks/Mono.framework/Commands/mono
LOGONSERVER=\GM-NORTHVPN
USERPROFILE=C:\Users\dreamanlan
SystemRoot=C:\WINDOWS
pdb2mdb=C:\Code\Client.\Tools\Batch/.../Tools/lib/mono/4.0/pdb2mdb.exe
USERDOMAIN_ROAMINGPROFILE=TENCENT
APPDATA=C:\Users\dreamanlan\AppData\Roaming
HOMEDRIVE=C:
suffix=Debug
PSModulePath=C:\Program Files\WindowsPowerShell\Modules;C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
libdir=C:\Code\Client.\Tools\Batch/.../CsLibrary/ExternLibrary
USERNAME=dreamanlan
PROCESSOR_ARCHITEW6432=AMD64
FPS_BROWSER_APP_PROFILE_STRING=Internet Explorer
PROCESSOR_ARCHITECTURE=x86
unityhome=C:\unity20180405\Editor
OS=Windows_NT
ComSpec=C:\WINDOWS\system32\cmd.exe
SystemDrive=C:
windir=C:\WINDOWS
ALLUSERSPROFILE=C:\ProgramData
delete directory C:\Code\Product.\Tools\Batch/.../Client/proj.unity/Assets/StandaloneData
create directory C:\Code\Product.\Tools\Batch/.../Client/proj.unity/Assets/StandaloneData
defaultCodePage:936
```

MetaDSL的语义

➤ MetaDSL的语义

- DslCalculator通用DSL脚本解释
 - 描述+执行用法的可重用执行部分
 - 已用于前面的地形生成
 - 将用于后面的资源处理



MetaDSL的语义

```
94     deletefiles("%clientdir%/Tools/JceProtoCodeGenerator/bin/Debug/DataProto/", "*.cs");
95     command{
96         win{: JceProtoCodeGenerator.exe :};
97         unix{: %monopath% JceProtoCodeGenerator.exe :};
98         error(99);
99     };
100
101    copyfile("%clientdir%/Tools/JceProtoCodeGenerator/bin/Debug/DataProto/Message.cs", "%clientdir%/CsLibrary/App/PluginFramework/Network/Message.cs");
102    copyfile("%clientdir%/Tools/JceProtoCodeGenerator/bin/Debug/DataProto/MessageEnum.cs", "%clientdir%/CsLibrary/App/PluginFramework/Network/MessageEnum.cs");
103
104    copyfile("%clientdir%/Tools/JceProtoCodeGenerator/bin/Debug/DataProto/Cs2LuaMessage.cs", "%clientdir%/Cs2LuaScript/Network/Cs2LuaMessage.cs");
105    copyfile("%clientdir%/Tools/JceProtoCodeGenerator/bin/Debug/DataProto/Cs2LuaMessageEnum.cs", "%clientdir%/Cs2LuaScript/Network/Cs2LuaMessageEnum.cs");
106
107    *****
108    * dll编译部分
109    *****
110
111    cd(clientdir);
112    command{
113        win
114        {: %xbuild% /version
115        :};
116
117        unix
118        {: %monopath% %xbuild% /version
119        :};
120    };
121
122
123    copyfile("%clientdir%/CsLibrary/ExternLibrary/FairyGUI_%suffix%.dll", "%clientdir%/CsLibrary/ExternLibrary/FairyGUI.dll");
124
125    if(osplatform() == "Unix"){
126        copyfile("%clientdir%/CsLibrary/ExternLibrary/MapFileForUnity_iOS.dll", "%clientdir%/CsLibrary/ExternLibrary/MapFileForUnity.dll");
127    }else{
128        copyfile("%clientdir%/CsLibrary/ExternLibrary/MapFileForUnity_PcAndroid.dll", "%clientdir%/CsLibrary/ExternLibrary/MapFileForUnity.dll");
129    };
130
131    createdir(logdir);
```

第四章 基于MetaDSL的剧情脚本与编辑器

本部分介绍DSL在实际项目里的一个应用，因为是实际产品的实现，细节会比较复杂，这里的执行模型与通常的脚本和我们前面介绍的DslCalculator不一样，我会结合实际代码演示与讲解，希望能讲明白。

这个应用也可以认为是前面LOP里语言工作台的一个实例，我们定义了一个剧情DSL，实现一个编辑器来编辑这个DSL的抽象表示，然后生成DSL，用于存储与执行。

开发背景

➤ Unity游戏常用的剧情工具

- Flux/uSequence插件
 - Timeline+Track+Event的概念，基于时间轴组织
 - 通用工具，不是专为剧情开发
 - 需要借助unity的各种编辑功能，比如动画要用Animation编辑，要求对unity编辑器比较熟悉
 - 与游戏逻辑的结合较弱，比如实现NPC走到目标点后再继续剧情
 - 编辑操作比较琐碎，编辑界面无法呈现剧情的主体结构
 - 2017年前我们试用了一小段时间，策划同学觉得不可用，放弃
- Timeline
 - Unity 2017版本引入，当时官方制作了一个电影片断，效果惊人
 - Timeline+Track+Clip的概念，基于时间轴组织
 - Clip有多种类型，比较好的集成了动画、特效与位置变化的编辑，支持融合与自动插值
 - 与Cinemachine插件紧密配合很好，可以制作专业级的镜头与动画表现效果
 - 2017年后我们项目比较关键的动画片断基本都改用Timeline制作

开发背景

➤ Unity游戏常用的剧情工具

- Timeline
 - 主要由美术同学使用，资源主要以静态资源为主，对策划同学制作动画难度过高（专业性）
 - 动画中使用玩家或游戏NPC时比较麻烦，需要使用替代资源编辑，运行时再替换为游戏里的对象
 - 制作周期长，通常以周为单位预估单个动画的制作时间
 - 修改非常麻烦，基本上相当于重新制作
 - 与游戏结合仍然较弱，动画过程中要插入交互非常困难（比如剧情对话选项与QTE）
- 自制剧情工具
 - 消息触发+消息处理与命令队列的概念，类似电影剧本的表述方式
 - 定制编辑器，保证基础概念映射到编辑器，在编辑器里能反映剧情的主要结构
 - 基于命令的表述隐藏了细节，策划不需要逐帧编辑，命令参数的编辑可以按策划需求定制
 - 与游戏逻辑结合紧密，基于游戏里的对象与功能组合，方便实现交互功能
 - 可以方便调用Timeline集成动画片断播放（此时Timeline更像是一个动画资源）
 - 制作快速，以小时为单位制作剧情
 - 2018年下半年开始成为项目主要剧情制作方式

开发背景

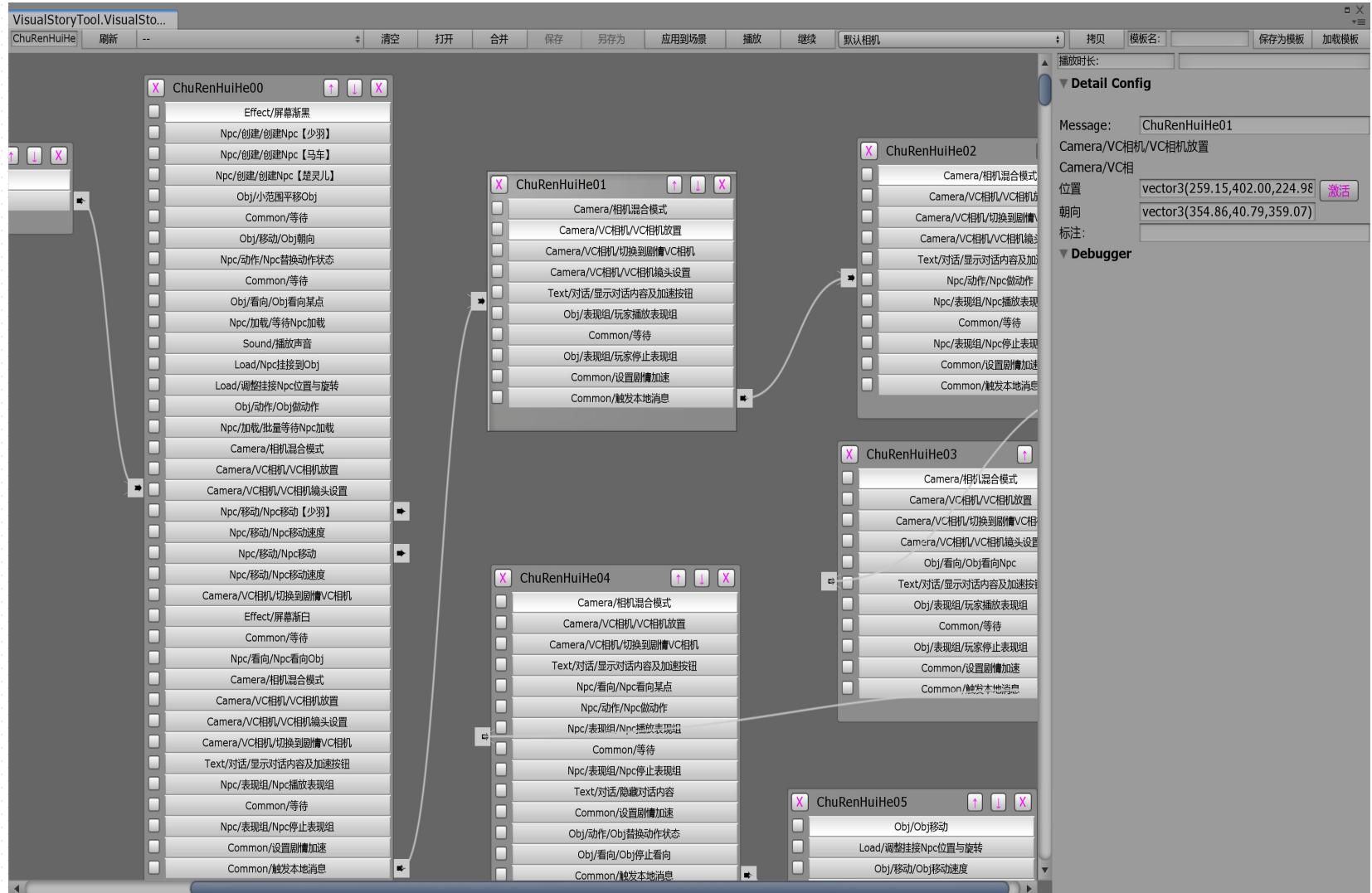
➤ Unity游戏常用的剧情工具

- MetaDSL的作用
 - 基于脚本的思路确保先建立概念模型（先使用脚本描述，再实现），并在编辑器开发前作为与策划沟通的素材，有助于建立概念层的一致性
 - 实现的剧情编辑器比较像语言工作台，DSL作为编辑器与剧情系统中间的纽带，保证了概念对应，约束了需求变化的方向（通过扩充新的命令来支持新需求）
 - 基于语言与解释器思路实现命令队列，提供了类似语言图灵完备性的参考，高层设计相对变得简单
 - 脚本语言方便实现逻辑热更新，语言+API的思路区分了不变与可变，有利于提升可扩展性
 - 剧情脚本作为可解析文本（开发时为文本，转换为运行时的二进制），方便批量修改与编辑

剧情演示

➤ 剧情演示

- 剧情编辑器
- 剧情脚本
- 剧情演示



剧情脚本的设计

➤ 剧情脚本的设计

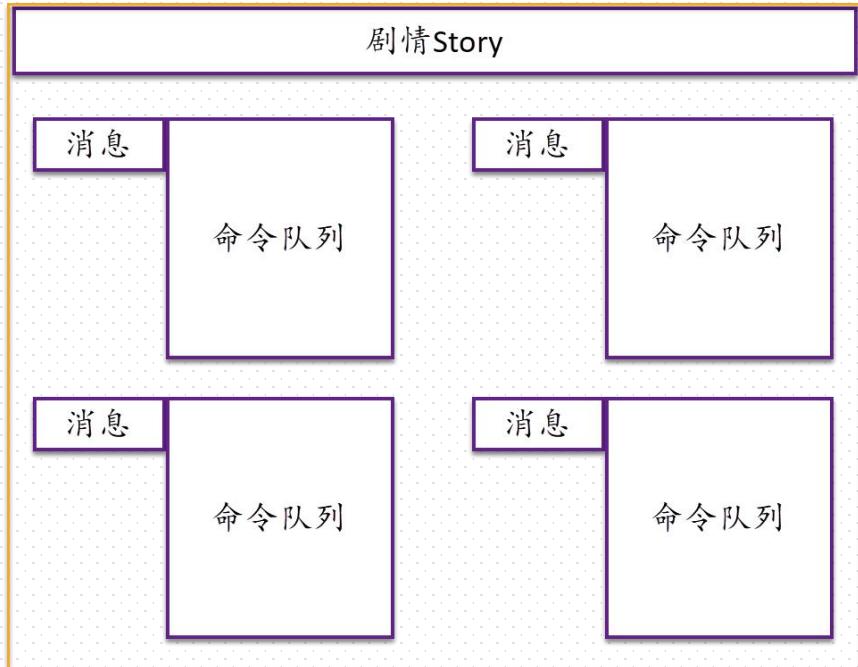
- 本部分将实现一个专用运行模型
 - 基于命令队列的解释器
 - 可以实现类似协程运行的并发非阻塞效果
 - 特别适合于多线描述带等待的逻辑
 - 电影剧本的效果
 - 思想最早见于《星际争霸》脚本编辑器



剧情脚本的设计

➤ 剧情脚本的设计

- 概念
 - Story
 - 消息
 - 消息处理



```
1 story(visual_main,"ChuNanCun")
2 {
3     local
4     {
5         @prefab1(0);
6         @prefab2(0);
7         @prefab3(0);
8         @prefab4(0);
9         @prefab5(0);
10    };
11    onmessage("ChuNanCun")
12    comments
13    {
14        pos(-504.00,-20252.00);
15    }
16    body
17    {
18        EnterDramaState();
19        SetBrainBlend(1,10.000);
20        VCA_SetPosAndDir(0,vector3(165.10,422.69,256.25),vector3(21.29,223.76,0.00));
21        SwitchToStoryCameraVCA(0);
22        storywait(4000);
23        ShowNpcMingPian(10101,62,1);
24        storywait(4000);
25        HideNpcMingPian();
26        storywait(2000);
27        storylocalmessage("end_ChuNanCun");
28    };
}
```

剧情脚本的设计

➤ 剧情脚本的设计

- 特点

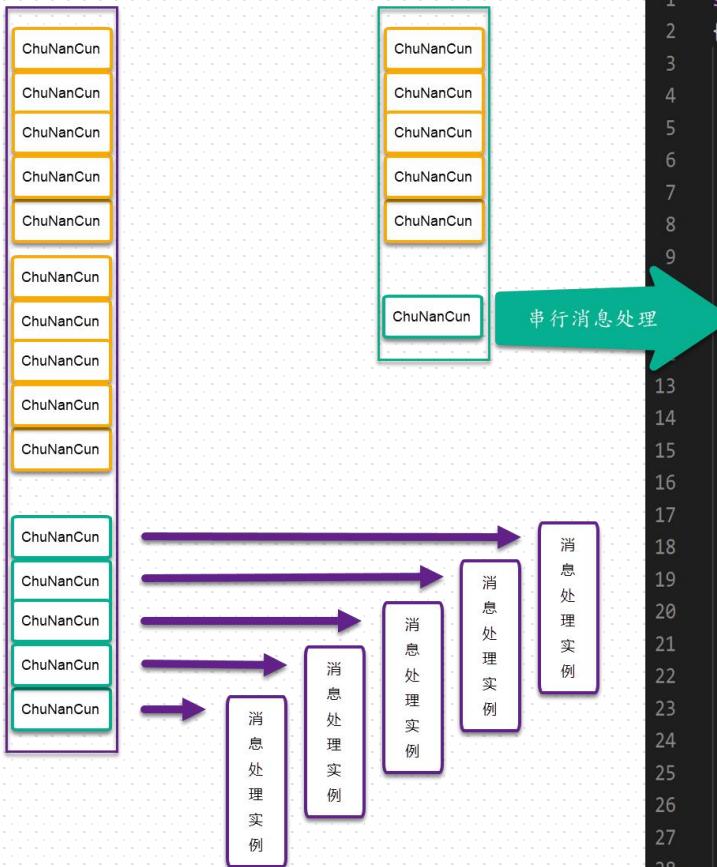
- 消息驱动，每个消息对应一个处理
 - Story里的每个消息处理有2个消息队列
 - 串行消息队列
 - 并行消息队列
 - 消息的触发基本没有开销
 - 触发消息只是把消息放到队列里
- 消息处理可并发
 - 同一个Story里可以有多个消息处理同时处于执行状态
 - 每个串行消息队列对应的消息处理执行一次处理一个消息，后续消息等前面消息处理完成后再处理
 - 并行消息队列里的每个消息额外实例化一个处理，只要不超过处理实例的上限（16个）就可以同时执行处理
- 消息处理跨多个tick执行

```
1 story(visual_main,"ChuNanCun")
2 {
3     local
4     {
5         @prefab1(0);
6         @prefab2(0);
7         @prefab3(0);
8         @prefab4(0);
9         @prefab5(0);
10    };
11    onmessage("ChuNanCun")
12    comments
13    {
14        pos(-504.00,-20252.00);
15    }
16    body
17    {
18        EnterDramaState();
19        SetBrainBlend(1,10.000);
20        VCA_SetPosAndDir(0,vector3(165.10,422.69,256.25),vector3(21.29,223.76,0.00));
21        SwitchToStoryCameraVCA(0);
22        storywait(4000);
23        ShowNpcMingPian(10101,62,1);
24        storywait(4000);
25        HideNpcMingPian();
26        storywait(2000);
27        storylocalmessage("end_ChuNanCun");
28    };
}
```

剧情脚本的设计

➤ 剧情脚本的设计

- 串行与并行消息处理解释

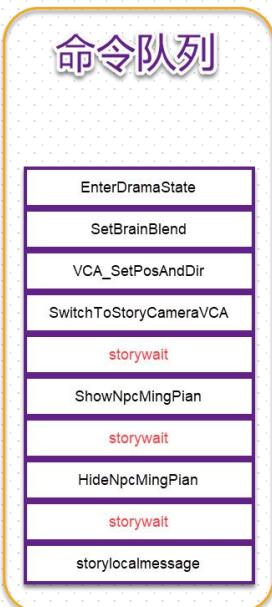


```
1 story(visual_main,"ChuNanCun")
2 {
3     local
4     {
5         @prefab1(0);
6         @prefab2(0);
7         @prefab3(0);
8         @prefab4(0);
9         @prefab5(0);
10    };
11    onmessage("ChuNanCun")
12    comments
13    {
14        pos(-504.00,-20252.00);
15    }
16    body
17    {
18        EnterDramaState();
19        SetBrainBlend(1,10.000);
20        VCA_SetPosAndDir(0,vector3(165.10,422.69,256.25),vector3(21.29,223.76,0.00));
21        SwitchToStoryCameraVCA(0);
22        storywait(4000);
23        ShowNpcMingPian(10101,62,1);
24        storywait(4000);
25        HideNpcMingPian();
26        storywait(2000);
27        storylocalmessage("end_ChuNanCun");
28    };
}
```

剧情脚本的设计

➤ 剧情脚本的设计

- 执行模型
 - 消息处理是一个命令队列
 - 命令队列队首的命令每个tick执行
 - 命令执行结果指示命令是否完成
 - 命令完成从队列里移除
 - 命令自己决定是单tick执行还是跨多个tick执行



消息处理代码的运行时表示

```
11  onmessage("ChuNanCun")
12  comments
13  {
14      pos(-504.00,-20252.00);
15  }
16  body
17  {
18      EnterDramaState();
19      SetBrainBlend(1,10.000);
20      VCA_SetPosAndDir(0,vector3(165.10,422.69,256.25),vector3(21.29,223.76,0.00));
21      SwitchToStoryCameraVCA(0);
22      storywait(4000);
23      ShowNpcMingPlan(10101,62,1);
24      storywait(4000);
25      HideNpcMingPlan();
26      storywait(2000);
27      storylocalmessage("end_ChuNanCun");
28  };
```

剧情脚本的设计

➤ 剧情脚本的设计

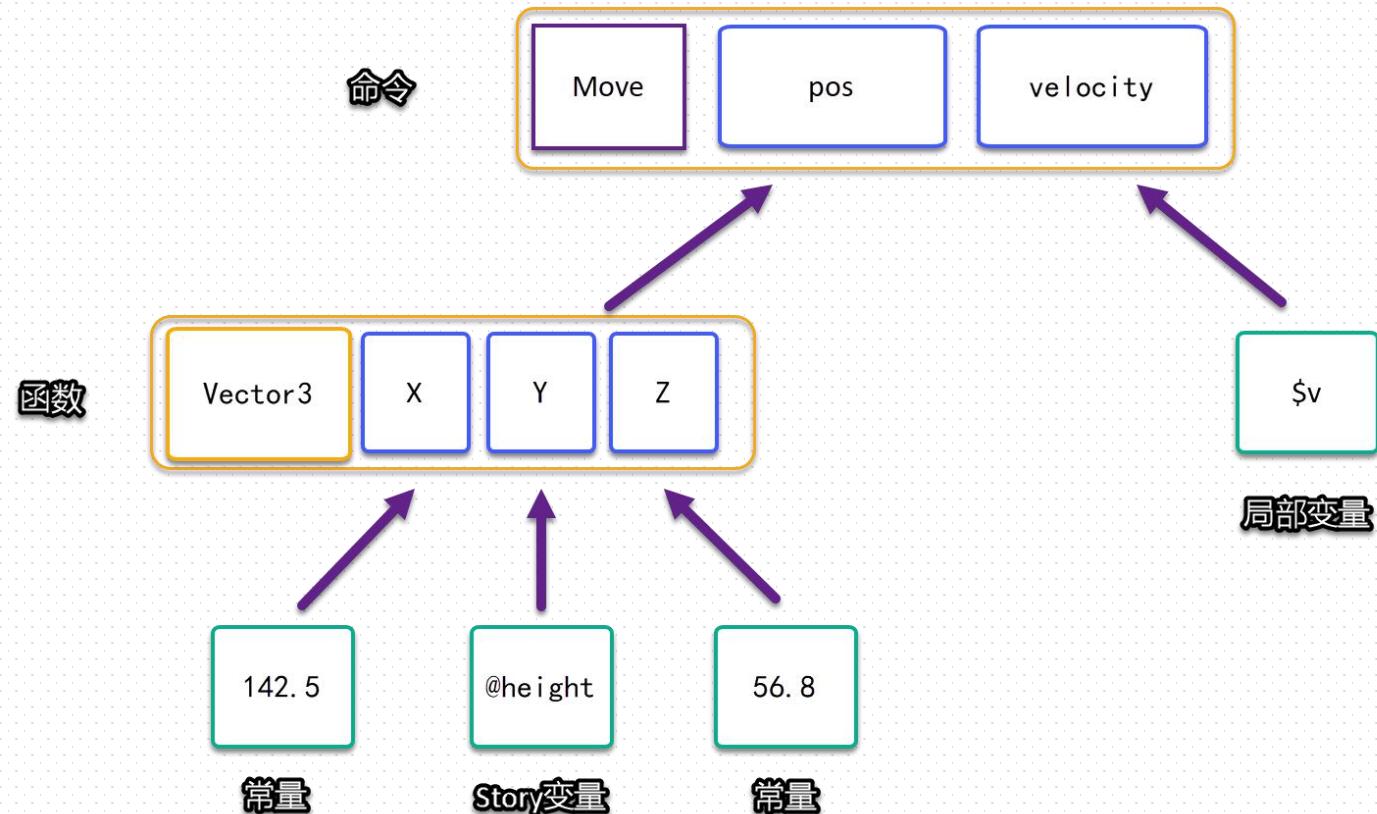
- 执行模型
 - 复合语句命令包含语句块，也就是命令列表，如if/loop等
 - 这些命令自身在消息处理的命令队列里是一条命令
 - 这些命令同时又带有命令队列，用来保存与执行构成语句块的命令
 - 这种多级命令队列的结构在运行时构成了命令队列栈，每个Tick总是栈顶的命令队列得到执行
 - 这也是剧情脚本的执行模型与传统脚本的执行模型的一个比较大的差别
 - 传统的脚本只有一个栈主要用于局部变量与参数，剧情脚本则既有变量栈还有一个命令队列栈，并且二者不是一一对应的



剧情脚本的设计

➤ 剧情脚本的设计

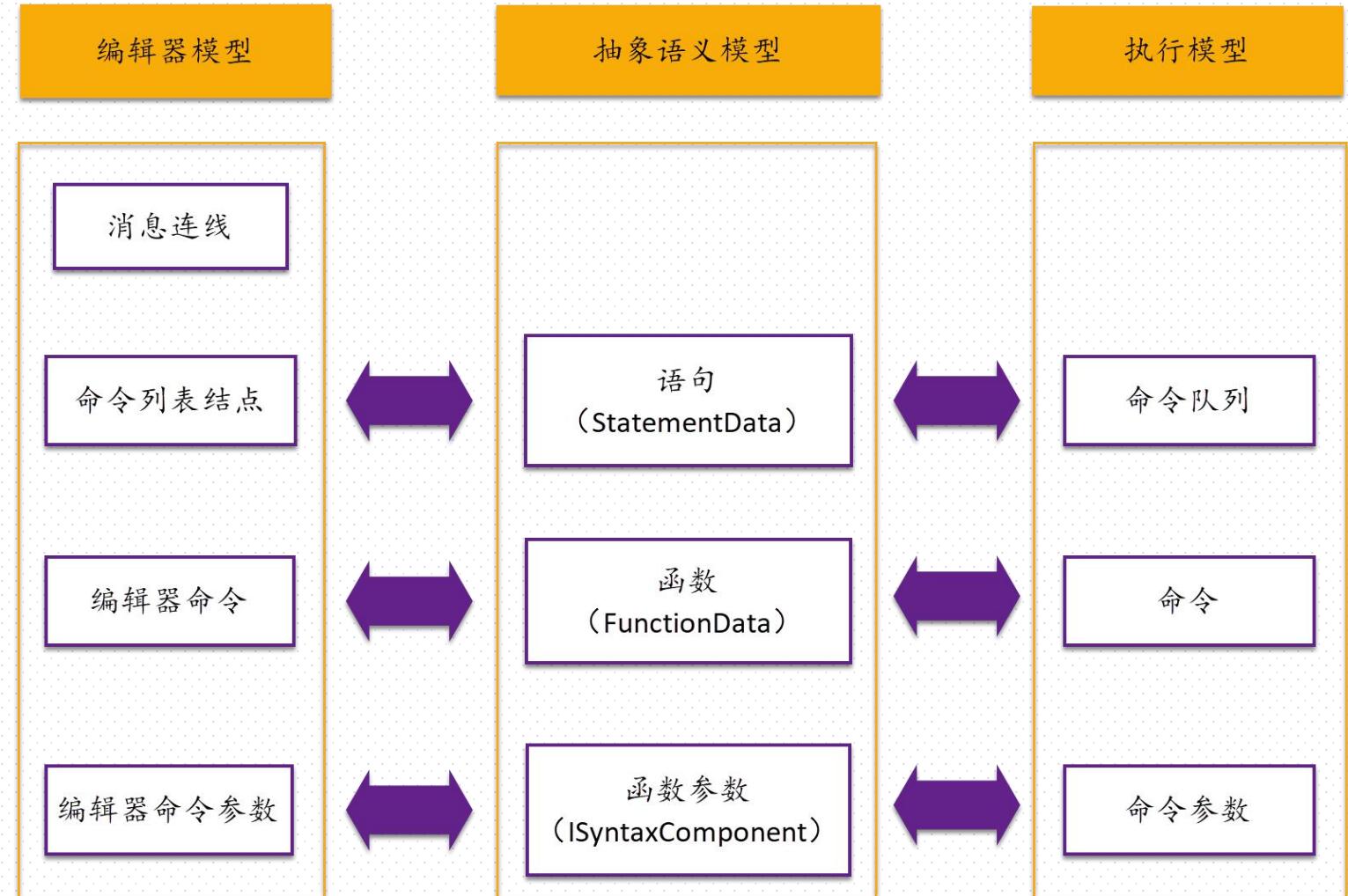
- 执行模型
 - 命令的参数可以是常量、变量或者一个函数调用或计算表达式
 - 命令参数在加载时构成一棵表达式树，运行时按后序遍历求值
 - 参数的求值在命令首次执行时完成
 - 命令跨tick执行时，参数值不会变化



剧情编辑器的设计

➤ 剧情编辑器的设计

- 概念上与脚本保持对应
 - 每个消息处理表示为一个可视图上的结点
 - 结点里面是各个命令
 - 消息触发为从命令到消息处理结点的连线

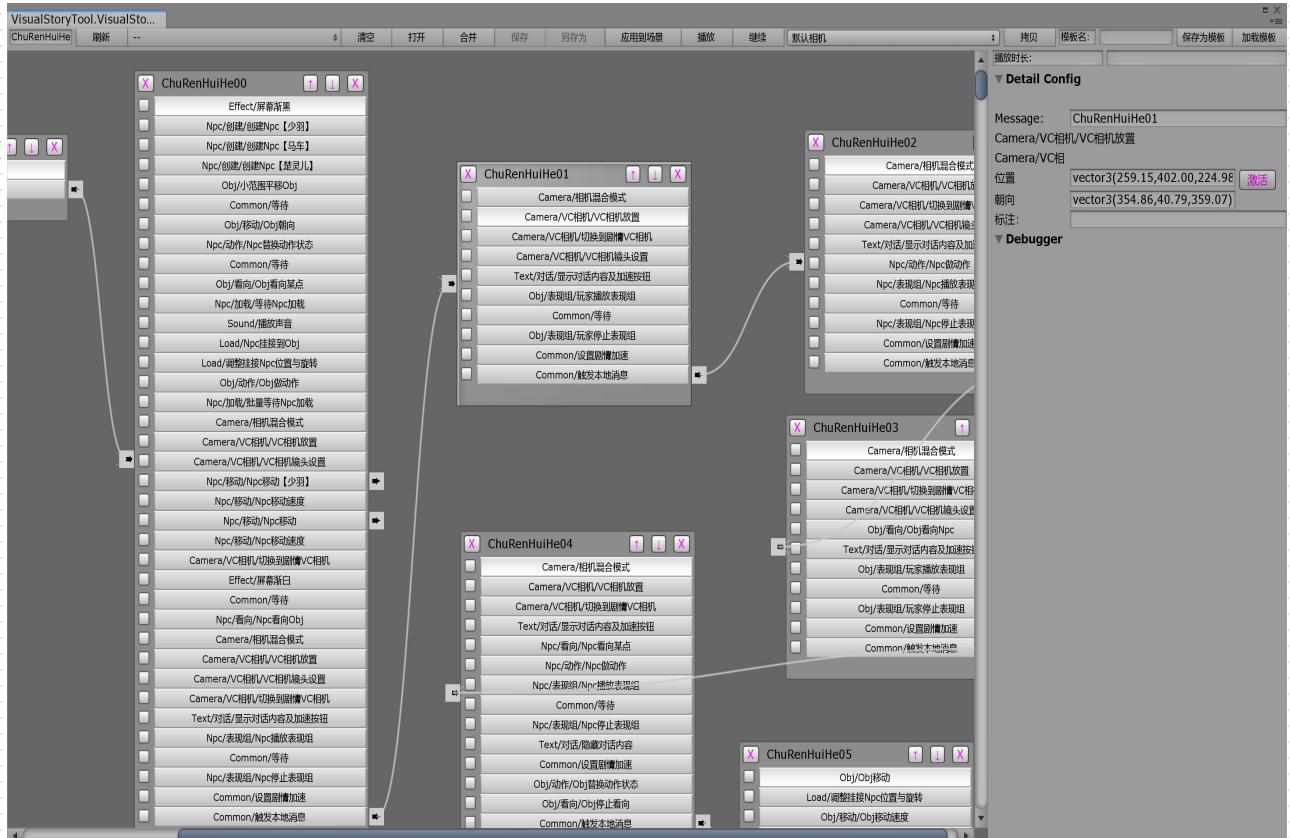


剧情编辑器的设计

➤ 剧情编辑器的设计

- 简化

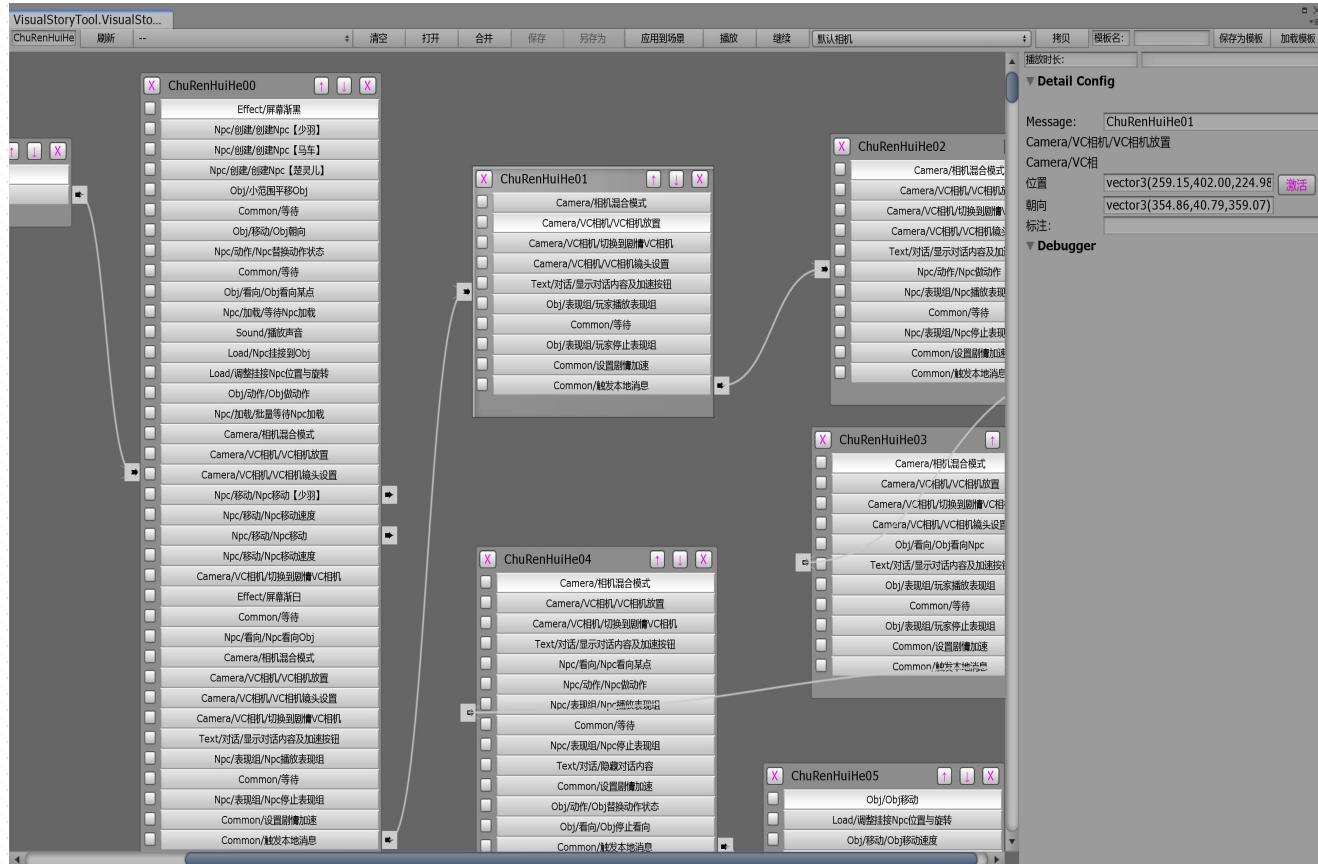
- 消息处理里只有一级命令列表
 - 不支持分支/循环命令，相关逻辑应封装到自定义命令里
- 默认不支持变量与赋值（减小副作用）



剧情编辑器的设计

➤ 剧情编辑器的设计

- 编辑方式
 - 采用属性窗口
 - 每个命令通过属性窗口设置
 - 命令参数支持配置编辑特性
 - 编辑特性为可扩展的编辑器代码，通过为不同类型的参数提供编辑特性实现绝大多数命令的灵活编辑



剧情编辑器的设计

➤ 剧情编辑器的设计

- 编辑器配置
 - [VisualStory.dsl](#)
 - 每个命令使用一个dsl文件描述
 - 名称
 - 说明
 - 参数
 - 初始值
 - 语义类别（决定了编辑样式）
 - 是否可选（位置参数还是命名参数）

```
defapi(SetBrainBlend, "Camera/相机混合模式"){
    blend(int, "混合模式"){
        semantic(popup);
        option("Cut",0);
        option("EaseInOut",1);
        option("EaseIn",2);
        option("EaseOut",3);
        option("HardIn",4);
        option("HardOut",5);
        option("Linear",6);
        initval(0);
        camera(true);
    };
    time(float, "过渡时间"){
        initval(2.0);
        camera(true);
    };
};
```

语义类别

```
defapi(NpcSetTransform, "Load/调整挂接Npc位置与旋转"){
    unitId(int, "UnitID"){
        initval(1001);
    };
    worldOrLocal(int, "1-世界坐标 0-本地坐标"){
        initval(1);
    };
    $position(vector3, "位置"){
        initval(vector3(0,0,0));
        optional(true);
    };
    $rotation(vector3, "朝向"){
        initval(vector3(0,0,0));
        optional(true);
    };
    $scale(vector3, "缩放"){
        initval(vector3(1,1,1));
        optional(true);
    };
};
```

位置参数

命名参数

剧情编辑器的设计

➤ 剧情编辑器的设计

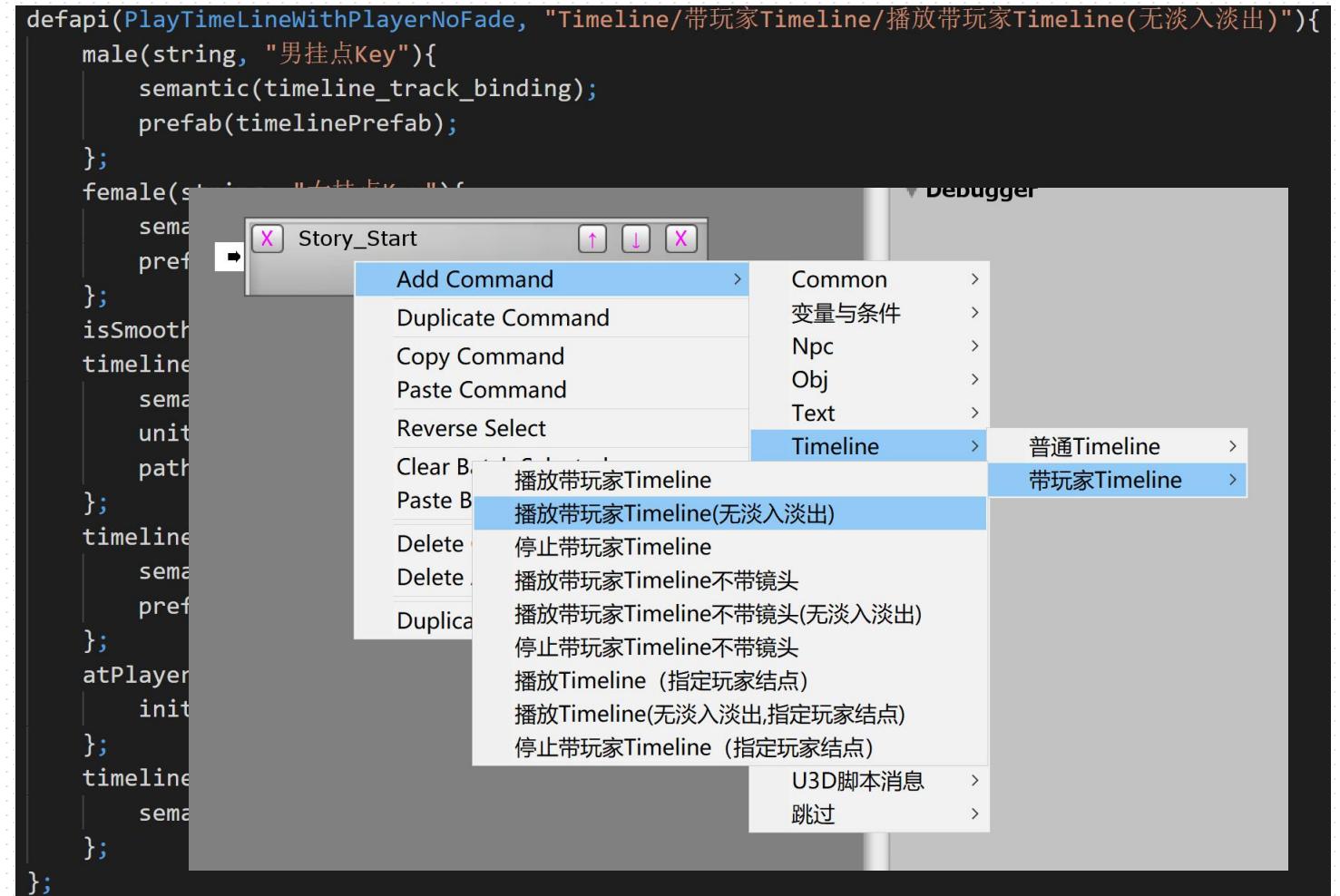
- 编辑器配置
 - 编辑器支持的命令列表从配置读取，构建操作菜单

```
defapi(PlayTimeLineWithPlayerNoFade, "Timeline/带玩家Timeline/播放带玩家Timeline(无淡入淡出)"){  
    male(string, "男挂点Key"){  
        semantic(timeline_track_binding);  
        prefab(timelinePrefab);  
    };  
    female(string, "女挂点Key"){  
        semantic(timeline_track_binding);  
        prefab(timelinePrefab);  
    };  
    isSmoothCamera(bool, "使用镜头融合");  
    timelinePrefab(string, "Timeline Prefab"){  
        semantic(timeline_prefab);  
        unitytype("GameObject");  
        path("LogicScenes/");  
    };  
    timelineNode(string, "TimeLine结点名称"){  
        semantic(timeline_node);  
        prefab(timelinePrefab);  
    };  
    atPlayerPos(int, "在玩家位置播放"){  
        initval(0);  
    };  
    timelineEndMsg(string, "TimeLine结束事件"){  
        semantic(timeline_end_msg);  
    };  
};
```

剧情编辑器的设计

➤ 剧情编辑器的设计

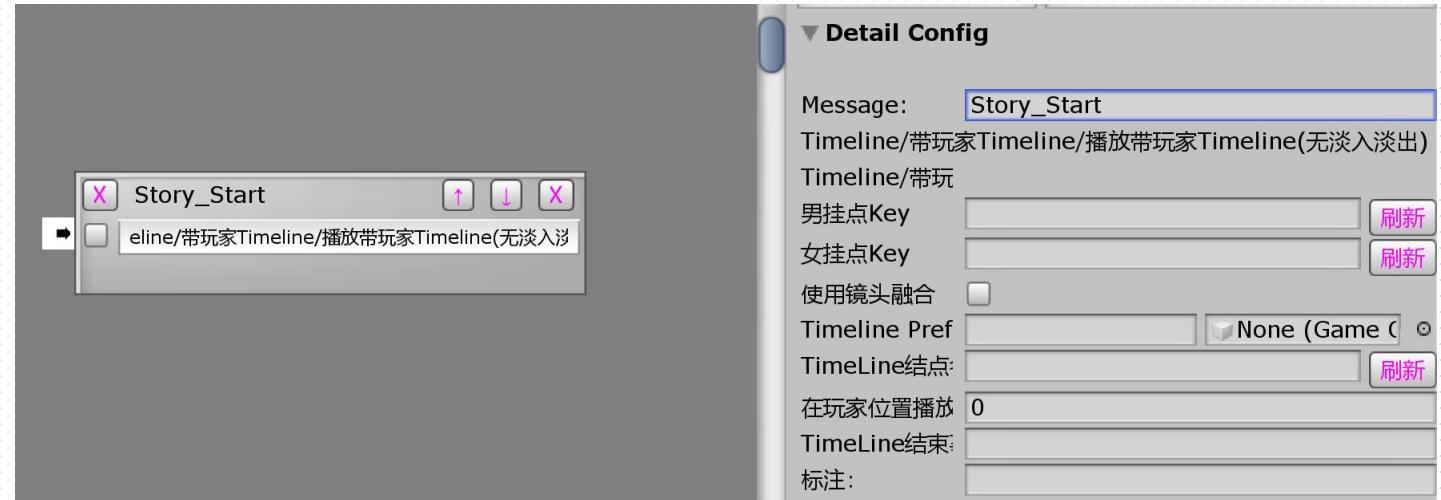
- 编辑器配置
 - 编辑器支持的命令列表从配置读取，构建操作菜单



剧情编辑器的设计

➤ 剧情编辑器的设计

- 编辑器配置
 - 命令以配置为模板创建实例，然后在编辑器里显示
 - 玩家交互编辑会修改或添加其它的数据



剧情编辑器的设计

➤ 剧情编辑器的设计

- [NodeInfo.cs](#)
- 参数编辑接口

```
public interface IParamDrawer
{
    void SetValue(string val);
    string GetValue();
    bool Inspect();
    bool Update();
}
```

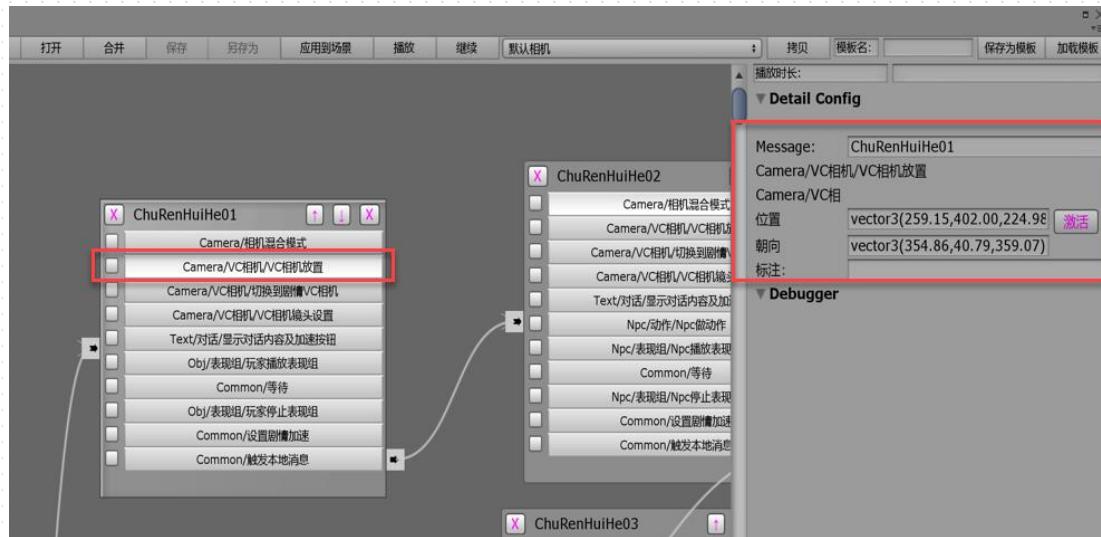
编辑功能，玩家在属性面板上的操作同步给参数数据与场景对象

同步更新功能，玩家对场景对象的修改同步到参数数据，进而同步到属性面板

剧情编辑器的设计

➤ 剧情编辑器的设计

- [NodeInfoEditor.cs](#)
- 命令编辑的流程



获取当前选中的命令信息

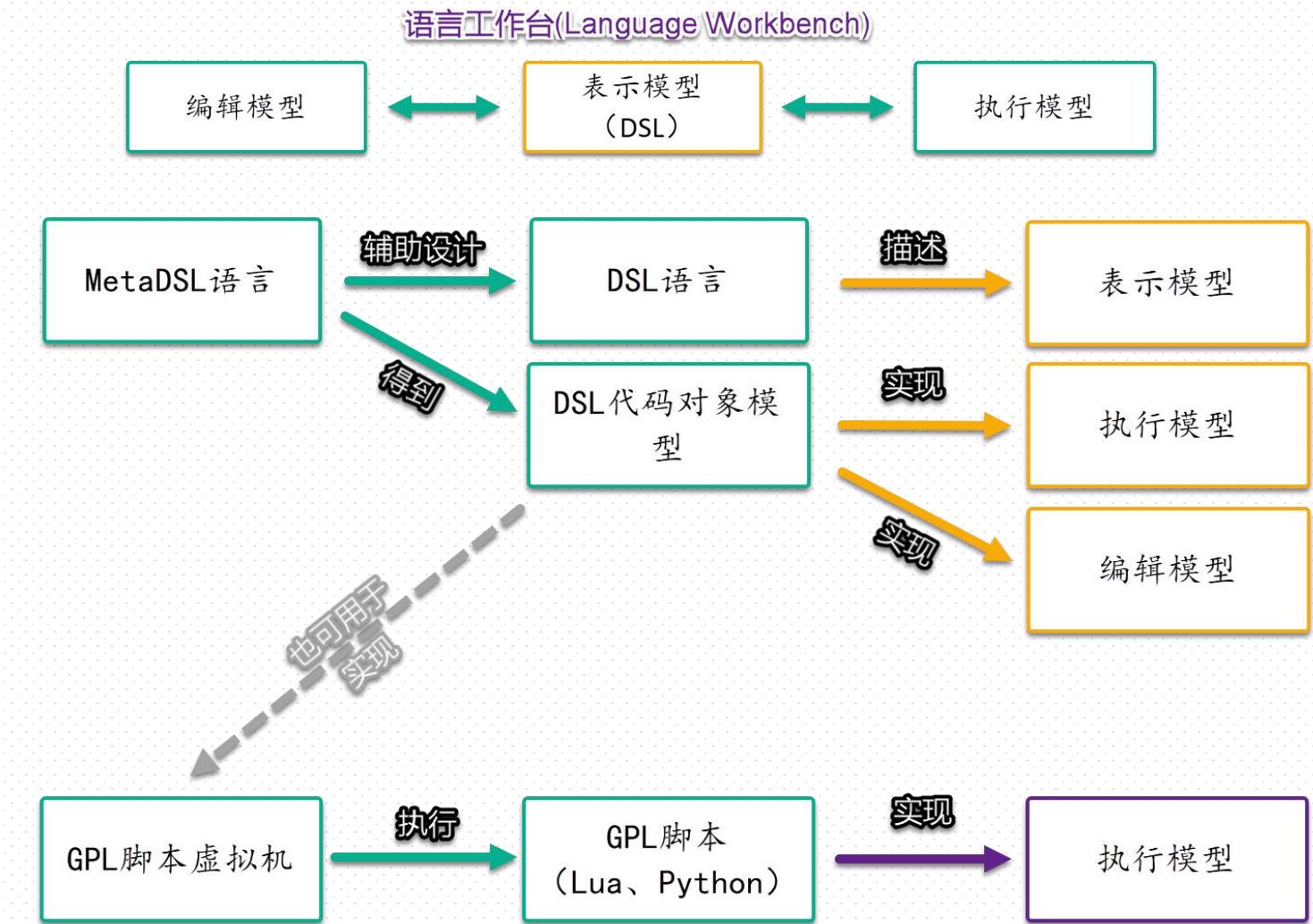
遍历命令的每个参数信息

调用参数信息的Drawer的方法
Inspect来进行交互式编辑

剧情编辑器设计

➤ 用Lua或其他GPL脚本实现 有啥差别

- DSL描述了业务模型
 - 开发与沟通围绕业务模型展开
 - 保证需求一致
 - 满足需求
 - 适应变化
- GPL脚本是开发语言，主要还是用于编写逻辑
 - 热更新
 - 扩展现有逻辑



第五章 一个基于DSL的通用的资源处理环境（简介）

资源处理工具演示

➤ 资源处理工具演示

- 资源检查
- 资源处理
- 数据分析
- 批量处理与自动化

ResourceEditWindow

允许SaveAndReimport 跳过特殊设置DB数据里的资源

强制SaveAndReimport 结果保存包含引用数据

分析 保存 加载 内存: 加载/刷新 批量转换 耗时: 清空 记录 保存 加载 拷贝 命令 加载 批处理

刷新列表 1.Find/1.Project Resources/Find Fbx Mesh 收集资源 处理选中资源 编辑器同步选中 创建场景 保存 加载 拷贝 命令 加载 批处理

triangleCount: 1000
filter:
notfilter:
meshfilter:
meshnotfilter:
pathwidth: 240

```
input("*.fbx")
{
    int("triangleCount", 1000);
    stringlist("filter", "");
    stringlist("notfilter", "");
}
```

全选 全不选 Total count (1398) Go to page (28) 1 Prev Next Refresh 升序 降序 Total value (5383292)

<input type="checkbox"/> 1.	Assets/ResourceAB/Character/Brc	mesh:Che vertex:3692 triangle:2054
<input type="checkbox"/> 2.	Assets/ResourceAB/Character/Brc	mesh:b_CaoChe_LOD vertex:2232 triangle:1208
<input type="checkbox"/> 3.	Assets/ResourceAB/Character/Brc	mesh:damen vertex:2331 triangle:1844
<input type="checkbox"/> 4.	Assets/ResourceAB/Character/Brc	mesh:B_DaMen_LOD vertex:1847 triangle:1034
<input type="checkbox"/> 5.	Assets/ResourceAB/Character/Brc	mesh:b_LuanJianCong_09_lod vertex:2504 triangle:1879
<input type="checkbox"/> 6.	Assets/ResourceAB/Character/Brc	mesh:CharMesh vertex:4244 triangle:3818
<input type="checkbox"/> 7.	Assets/ResourceAB/Character/Brc	mesh:QingTongZun vertex:2847 triangle:2053
<input type="checkbox"/> 8.	Assets/ResourceAB/Character/Brc	mesh:b_QingTongZun_lod vertex:1681 triangle:1223

资源处理脚本的设计

➤ 资源处理脚本的设计

- 资源检查的一般流程
 1. 确定检查什么类型的资源
 2. 遍历该类型的资源
 3. 遍历过程中，逐一进行检查
 4. 输出检查结果
- 不管什么类型的资源，检查的流程是相似的，不同的是每类资源的检查规则与具体的检查方法
 - 也就是说如果检查规则与方法是可以配置的，那资源检查就是通用的系统
- 这有点像是在一个资源集合上查找不满足指定条件的元素列表
 - SQL？
 - 但资源库不是关系数据库，并不能像SQL语句的条件一样进行集合操作，而是像程序代码一样对每个资源都需要一个逻辑来计算是否满足条件
 - 也就是说对每类资源，有一个检查逻辑，对资源集合里的每一个资源运行一次
 - 这有点像shader了（一套逻辑对每个顶点或像素运行一次）

资源处理脚本的设计

➤ 资源处理脚本的设计

- 特性
 - 作为资源处理工具的配置项
 - 菜单与描述
 - 用于构建工具支持的各类检查/处理列表
 - 这个列表自动根据配置项构建
 - 选项与可交互部分
 - 执行部分
 - 可以批量处理与自动化
- 概念
 - 目标资源对象
 - 交互选项
 - 过滤逻辑
 - 对每个资源执行一次
 - 排序、分组、统计
 - 处理

```
1 input("*.fbx")
2 {
3     int("triangleCount", 1000);
4     int("componentCount", 3);
5     stringlist("filter", "");
6     stringlist("notfilter", "");
7     stringlist("uvfilter", "");
8     float("pathwidth", 240){range(20,4096);};
9     feature("source", "project");
10    feature("menu", "1.Project Resources/Check Fbx Mesh");
11    feature("description", "just so so");
12 }
13 filter
14 {
15     if(stringcontains(assetpath, filter) && stringnotcontains(assetpath, notfilter)){
16         var(0) = loadasset(assetpath);
17         var(1) = collectmeshinfo(var(0), importer);
18         //unloadasset(var(0));
19         order = var(1).triangleCount;
20         if(var(1).triangleCount >= triangleCount){
21             var(2) = calcmeshvertexcomponentcount(var(0),true);
22             looplist(var(2)){
23                 $key = $$._Key;
24                 $val = $$._Value;
25                 if($val >= componentCount && stringcontains($key, uvfilter)){
26                     var(3) = newitem();
27                     var(3).AssetPath = assetpath;
28                     var(3).Info = format("skinned:{0},mesh:{1},vertex:{2},triangle:{3},vertex components:{4} {5}",
29                                         var(1).skinnedMeshCount, var(1).meshFilterCount, var(1).vertexCount, var(1).triangleCount, $val, $key
30                                         );
31                     var(3).Order = $val;
32                     var(3).Value = 0;
33                 };
34             };
35         };
36     };
37     0;
38 }
```

资源处理脚本解释器

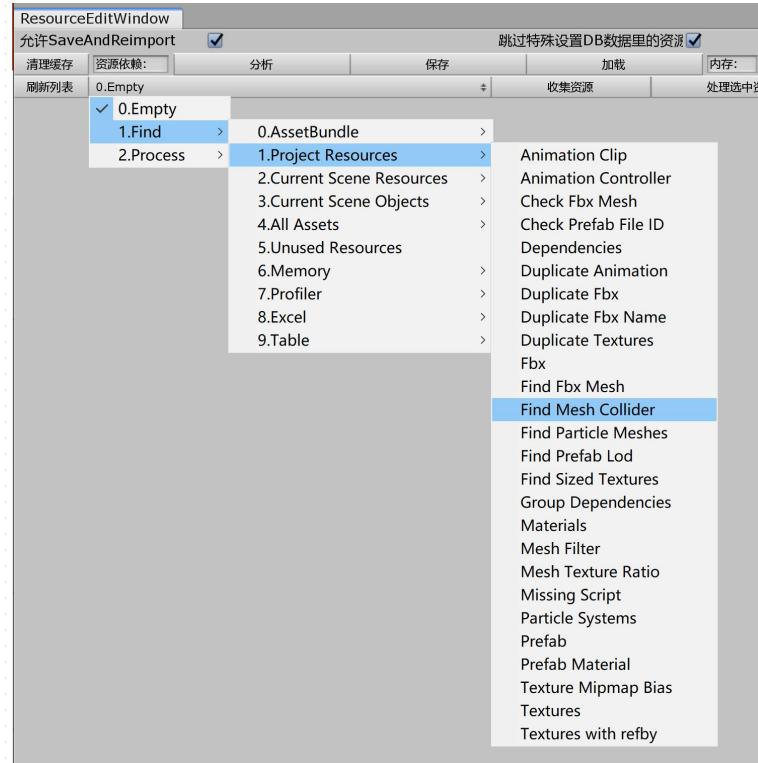
➤ 资源处理脚本解释器

- 菜单构建

- 资源处理工具读取各个处理脚本里的菜单信息来构建菜单

```
1  input("*.prefab")
2  {
3      float("pathwidth",240){range(20,4096);};
4      feature("source", "project");
5      feature("menu", "1.Project Resources/Find Mesh Collider");
6      feature("description", "just so so");
7  }
8  filter
9  {
10     object = loadasset(assetpath);
11     var(0) = getcomponent(object, "MeshCollider");
12     if(isnull(var(0))){
13         0;
14     }else{
15         1;
16     };
17 }
```

在处理工具菜单上的信息，
工作读取每个脚本的相关信
息来构建菜单



资源处理脚本解释器

➤ 资源处理脚本解释器

- 交互UI构建
 - 资源处理脚本的input部分会指明一些参数，资源工具根据这里的信息来构建交互UI
 - 得益于unity传统编辑器GUI的机制，比较容易自动生成与处理交互UI

```
1  input("*.tga","*.png","*.jpg")
2  {
3      int("maxSize",256){
4          |    range(1,1024);
5      };
6      string("prop","");
7          |    multiple(["readable","mipmap"],[1,2]);
8      };
9      string("filter","");
10     string("notfilter","");
11     string("style", "itemlist"){
12         |    popup(["itemlist", "groupelist"]);
13     };
14     int("duptype",1){
15         |    toggle(["md5","guid"],[1,2]);
16     };
17     float("pathwidth",240){range(20,4096);}
18     feature("source", "project");
19     feature("menu", "1.Project Resources/Duplicate Textures");
20     feature("description", "just so so");
21 }
22 filter
23 {
24     var(0) = loadasset(assetpath);
```

脚本查询或处理逻辑会用到的一些参数，这里的描述信息指明了参数类型与UI显示相关的内容

资源处理脚本解释器

➤ 资源处理脚本解释器

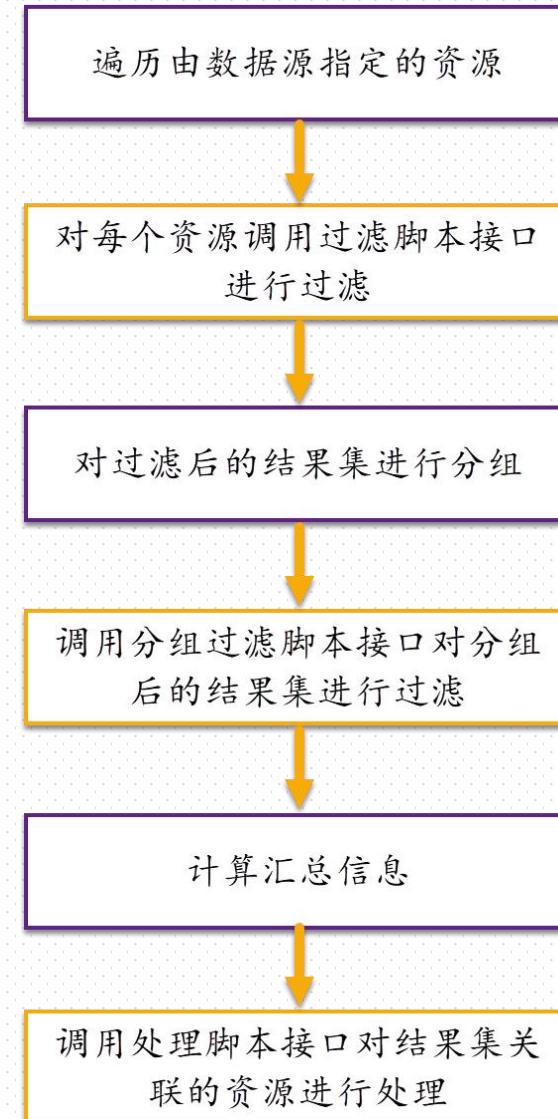
- 数据源
 - 数据源指明资源的来源，比如可能是unity工程里的资源文件或者是场景里的资源对象
 - 资源处理工具对不同的数据源需要采用不一样的遍历方式
 - 文件资源：按目录与扩展遍历
 - 场景资源：按场景依赖的资源集遍历
 - 场景对象：按场景层次结构遍历
 - 数据表：按记录顺序遍历
 - 内存数据：按内存数据集遍历

资源处理脚本解释器

➤ 资源处理脚本解释器

- 查询与处理的流程

1. 根据数据源选择不同的遍历方法
2. 对遍历的每一个资源，调用filter处理脚本，将符合条件的结果添加到结果集
3. 如果配置了分组，则对结果集进行分组与统计计算
4. 如果脚本里配置了group，对分组的每一条结果调用group脚本进行分组后过滤，将符合的结果添加到分组结果集
5. 计算总计信息
6. 调用process脚本对结果集关联的资源进行处理



再谈描述+执行的脚本解释模式

➤ 再谈描述+执行的脚本解释模式

- DSL作为领域特定语言，往往与业务系统有很深的联系，这是与通用语言很大的不同
 - DSL使用的是业务的语言
 - 语义上使用业务的核心概念
 - 甚至DSL语言就是业务的核心概念（在LOP方法里尤其如此）
 - 比如资源脚本里的菜单、数据源、过滤、分组和处理
- 我们可以这样看DSL的作用
 - 概念上作为业务流程与核心概念的有机组成（语言）
 - 确定语言的元素
 - 可配置、数据驱动
 - 运行时作为概念的实现支撑，提供胶水把API组合起来实现业务功能
 - 可以扩展
 - 可以热更新
 - 这两部分正是描述+执行
 - 描述部分与业务关系紧密，业务会依赖描述部分（实际上业务解释描述部分），通常相对稳定
 - 执行部分与业务关系耦合较为松散，业务不依赖执行部分，执行部分的部分API依赖业务系统

再谈描述+执行的脚本解释模式

➤ 再谈描述+执行的脚本解释模式

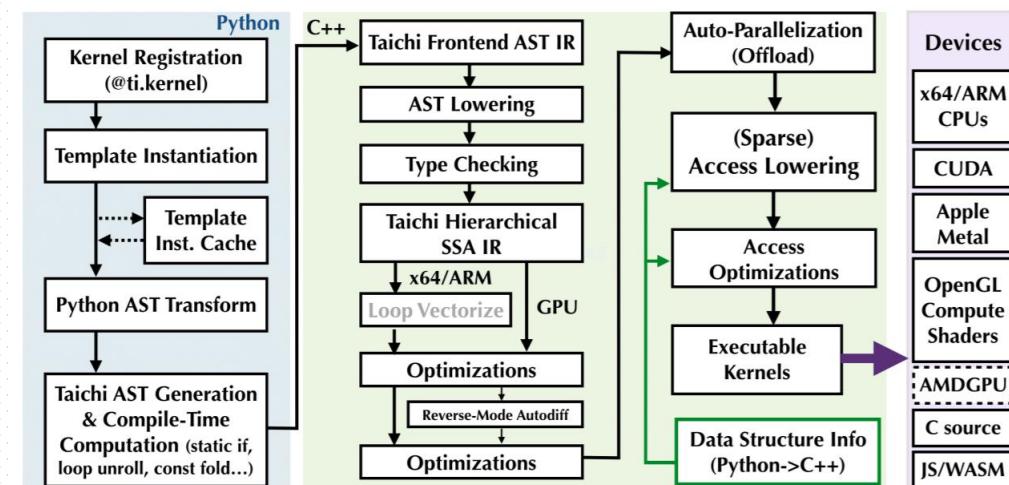
- 或者说
 - 描述部分构成了DSL应用的框架与核心概念
 - 执行部分构成了DSL应用的组件或API，嵌入在描述部分里
- 实际使用的特点
 - 描述部分对每个DSL应用都不相同，由业务决定
 - 执行部分的解释器可以通用（`DslCalculator`），每个DSL应用会有部分特定的API需要编写
 - 设计的重点是描述部分



抛砖引玉—还可以做什么呢

➤ 抛砖引玉—还可以做什么呢

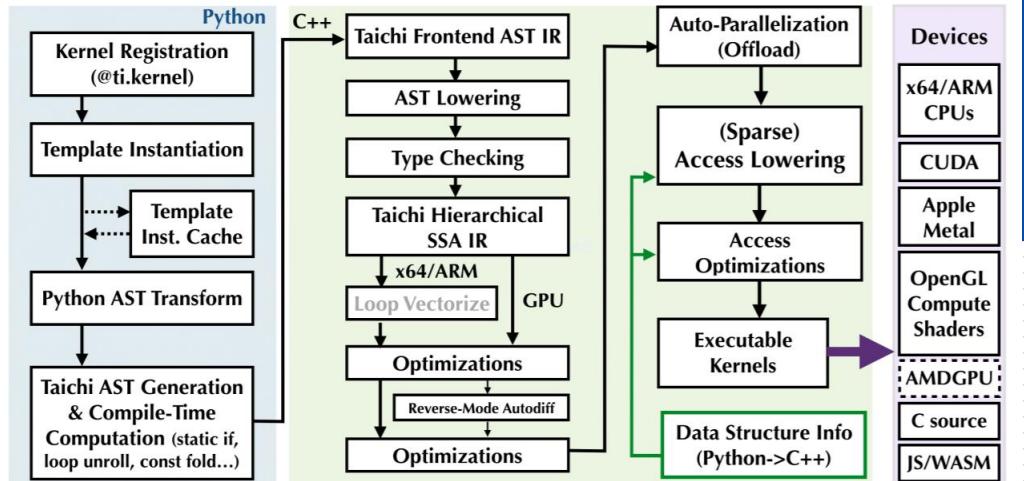
- 目前都是讲的解释器样式的，如果想要利用GPU执行呢？
 - 可以把解释器改成代码生成器，生成gpu上执行的shader
 - 目前很火的太极编程语言，其实前端是把python当DSL使用了



抛砖引玉—还可以做什么呢

➤ 抛砖引玉—还可以做什么呢

- 目前都是讲的解释器样式的，如果想要利用GPU执行呢？
 - 可以把解释器改成代码生成器，生成gpu上执行的shader
 - 目前很火的太极编程语言，其实前端是把python当DSL使用了
 - 利用了python库将python源码翻译为AST的功能
 - 利用python的decorator功能，为指定函数指定decorator
 - 这个decorator获取该函数的AST，然后生成太极的AST，再交给太极的C++部分继续处理（当然，太极的后端才是它核心的部分）



```
def do_compile(self):
    src = remove_indent(oinspect.getsource(self.func))
    tree = ast.parse(src)

    func_body = tree.body[0]
    func_body.decorator_list = []

    if impl.get_runtime().print_preprocessed:
        import astor
        print('Before preprocessing:')
        print(astor.to_source(tree.body[0], indent_with=' '))

    visitor = ASTTransformer(is_kernel=False, func=self)
    visitor.visit(tree)
    ast.fix_missing_locations(tree)

    if impl.get_runtime().print_preprocessed:
        import astor
        print('After preprocessing:')
        print(astor.to_source(tree.body[0], indent_with=' '))

    ast.increment_lineno(tree, oinspect.getsourcelines(self.func)[1] - 1)

    local_vars = {}
    global_vars = _get_global_vars(self.func)

    exec(
        compile(tree,
                filename=oinspect.getsourcefile(self.func),
                mode='exec'), global_vars, local_vars)
    self.compiled = local_vars[self.func.__name__]
```

抛砖引玉—还可以做什么呢

➤ 抛砖引玉—还可以做什么呢

- PCG—程序化内容生成
 - 刚开始介绍的地形生成其实是个简单PCG

抛砖引玉—还可以做什么呢

➤ 抛砖引玉—还可以做什么呢

- 专用语言—翻译到C++或结合llvm可以做到高性能
 - Unity的burst编译器相当于是使用C#作为DSL，然后结合JobSystem做优化

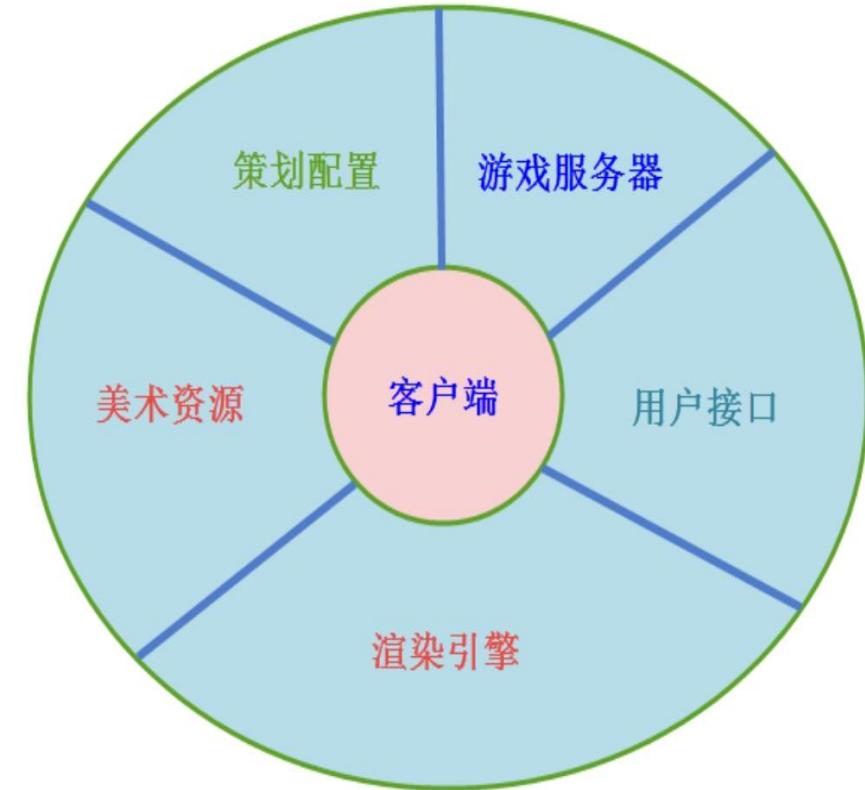
第六章 LOP与游戏前端架构

关于架构的简单思考与展望

LOP与游戏前端架构

➤ 游戏前端开发

- 前端开发通常会涉及
 - 策划表格
 - 美术资源
 - 游戏引擎
 - 游戏服务器（网络）
 - 用户接口（UI）
- 可以理解为
 - 前端负责集成所有游戏相关的素材并通过用户接口给玩家提供体验
 - 前端的特点就是杂
 - 技术杂
 - 功能杂
 - 数据杂



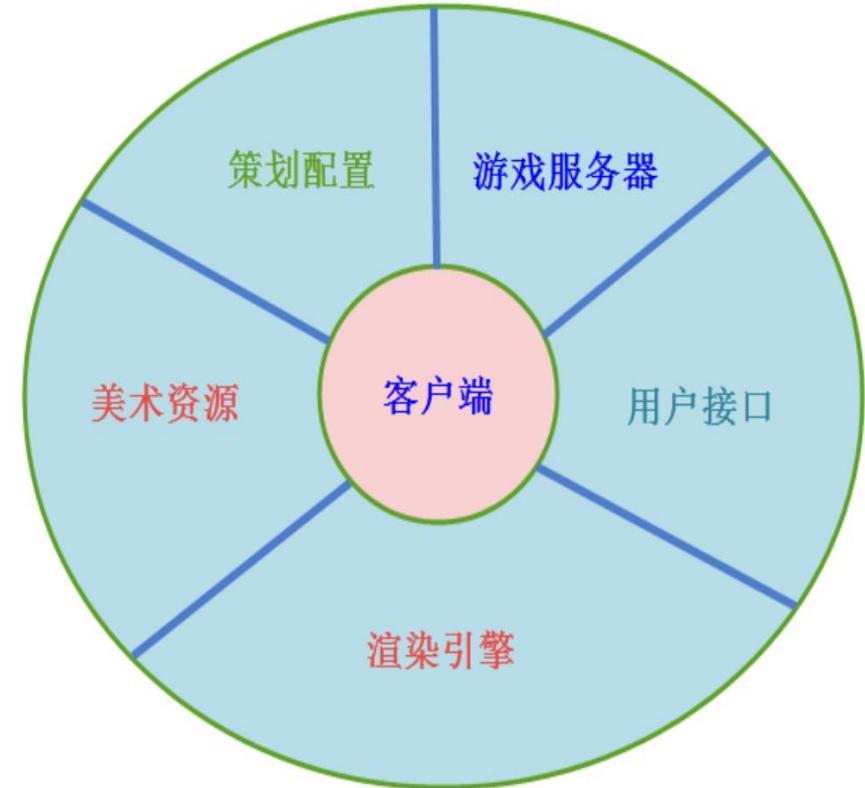
LOP与游戏前端架构

➤ 游戏前端开发

- 那么
- 前端开发最担心的又是什么？

变化

前端开发中相对不会变化的又是什么呢？



LOP与游戏前端架构

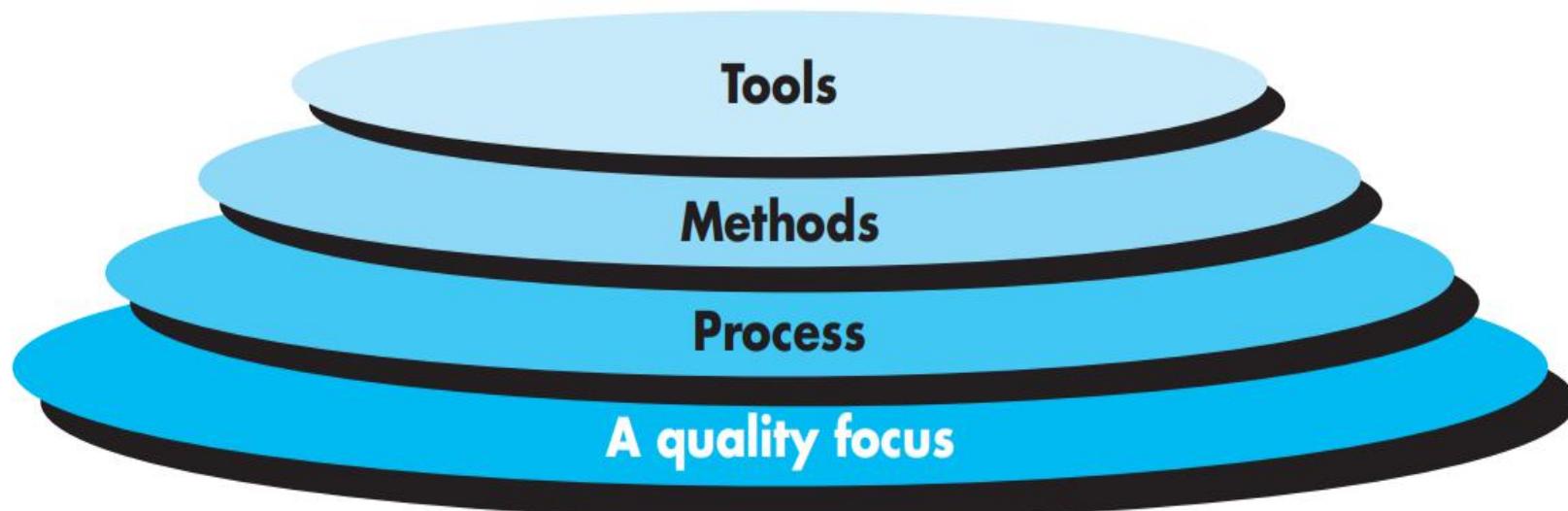
➤ 支持变化重要么？

- 我们先来复习一下软件工程的一些原则

LOP与游戏前端架构

➤ 支持变化重要么？

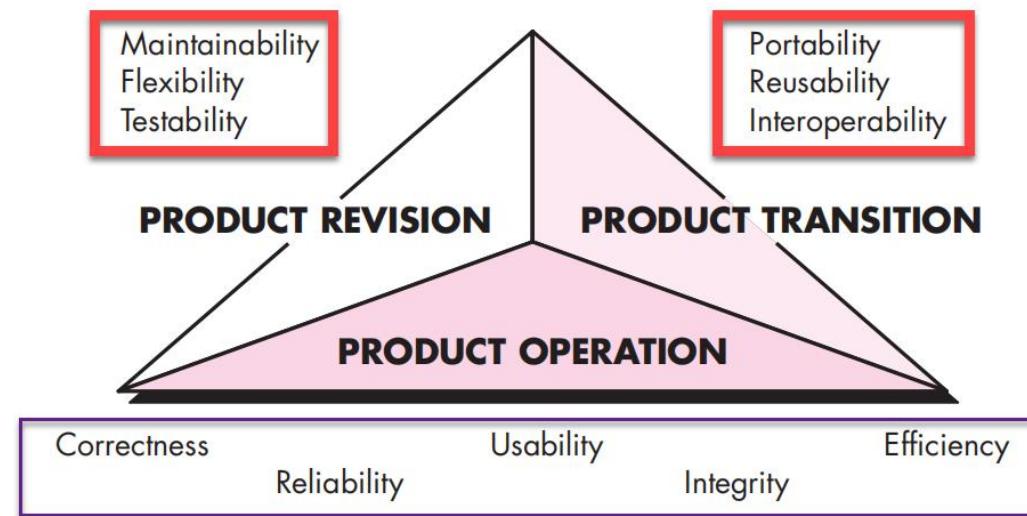
- 软件工程——一种层次化技术
 - 《Software Engineering - A Practitioner's approach》
 - 根基是对软件质量的关注点



LOP与游戏前端架构

➤ 支持变化重要么？

- 软件质量的McCall模型里，三个维度中有两个维度与应对变化相关



- 在《架构整洁之道》里，甚至把支持变化看得比正确性更重要

LOP与游戏前端架构

➤ 易修改性 (Changeability)

- 可维护性
 - 如何进行局部性的修改并使影响最小
- 可扩展性
 - 不影响用户的情况下，使用改进版本的组件替换已有组件
 - 支持新特性或禁用旧特性
- 结构重组
 - 支持重新组织系统间的组件与组件间的关系
- 可移植性
 - 适用于多种硬件平台、用户界面、操作系统、编程语言或编译器

➤ 如何支持变化—启发性原理

- 一些比较重要的启发性原理（《面向模式的软件架构》第一卷）
 - ✓ 抽象（abstraction）
 - ✓ 封装（Encapsulation）
 - ✓ 信息隐藏（Information Hiding）
 - 模块化（Modularization）
 - 关注点分离（Separation of Concerns）
 - ✓ 耦合和内聚（Coupling and Cohesion）
 - ✓ 充分性、完整性和原始性（Sufficiency, Completeness and Primitiveness）
 - ✓ 策略和实现分离（Separation of Policy and Implementation）
 - ✓ 接口和实现分离（Separation of Interface and Implementation）
 - 单一引用点（Single Point of Reference）
 - 分而治之（Divide-and-Conquer）

➤ 如何支持变化—设计原则

- Robert c.martin
 - 《敏捷软件开发》 (**Agile Software Development – Principles,Patterns, and Practices**)
 - 《架构整洁之道》 (**Clean Architecture - A Craftsman's Guide to Software Structure and Design**)
- SOLID设计原则
 - SRP (单一职责原则)
 - 一个软件模块只应服务于一类用户，不要服务几类不同的用户
 - OCP (开闭原则)
 - 一个软件模块应该对扩展开放（允许扩展），而对修改封闭（拒绝修改） LSP (里氏替换原则)
 - LSP (里氏替换原则)
 - 子类型必须要能够替换他们的基类型（is-a关系）
 - ISP (接口隔离原则)
 - 不应该强迫用户依赖他们不用的方法
 - DIP (依赖倒置原则)
 - 高层模块不应依赖于低层模块，二者都应依赖抽象
 - 抽象不应依赖于细节，细节应该依赖抽象

➤ 如何支持变化—与顶层分解相关的设计原则

- SOLID设计原则（忽略类与接口详细设计原则）
 - SRP（单一职责原则）
 - 一个软件模块只应服务于一类用户，不要服务几类不同的用户
 - OCP（开闭原则）
 - 一个软件模块应该对扩展开放（允许扩展），而对修改封闭（拒绝修改）
 - DSL有助于在设计初期识别可扩展与可替换的部分，提升扩展性，控制破坏性修改
 - LSP（里氏替换原则）
 - 子类型必须要能够替换他们的基类型（is-a关系）
 - ISP（接口隔离原则）
 - 不应该强迫用户依赖他们不用的方法
 - DIP（依赖倒置原则）
 - 这条很关键
 - 高层模块不应依赖于低层模块，二者都应依赖抽象
 - 抽象不应依赖于细节，细节应该依赖抽象

➤ 如何支持变化—依赖原则

- REP (重用/发布等价原则)
 - 软件复用的最小粒度应等同于其发布的最小粒度
- CCP (共同闭包原则)
 - 将会同时修改的类放到同一包中，将不会同时修改的类放到不同包中
- CRP (共同复用原则)
 - 不要强迫用户依赖他们不需要的东西，如果重用了包中的一个类，那么就要重用包中所有类
- ADP (无环依赖原则)
 - 在包的依赖关系图中不允许存在环
- SDP (稳定依赖原则)
 - 依赖关系要指向稳定的方向，亦即不稳定的依赖稳定的
- SAP (稳定抽象原则)
 - 包的抽象程序应与其稳定性一致，越稳定的越抽象

➤ 如何支持变化—与顶层分解相关的依赖原则（忽略模块细分的原则）

- REP (重用/发布等价原则)
 - 软件复用的最小粒度应等同于其发布的最小粒度
- CCP (共同闭包原则)
 - 将会同时修改的类放到同一包中，将不会同时修改的类放到不同包中
- CRP (共同复用原则)
 - 不要强迫用户依赖他们不需要的东西，如果重用了包中的一个类，那么就要重用包中所有类
- ADP (无环依赖原则)
 - 在包的依赖关系图中不允许存在环
- SDP (稳定依赖原则)
 - 依赖关系要指向稳定的方向，亦即不稳定的依赖稳定的
- SAP (稳定抽象原则)
 - 包的抽象程序应与其稳定性一致，越稳定的越抽象

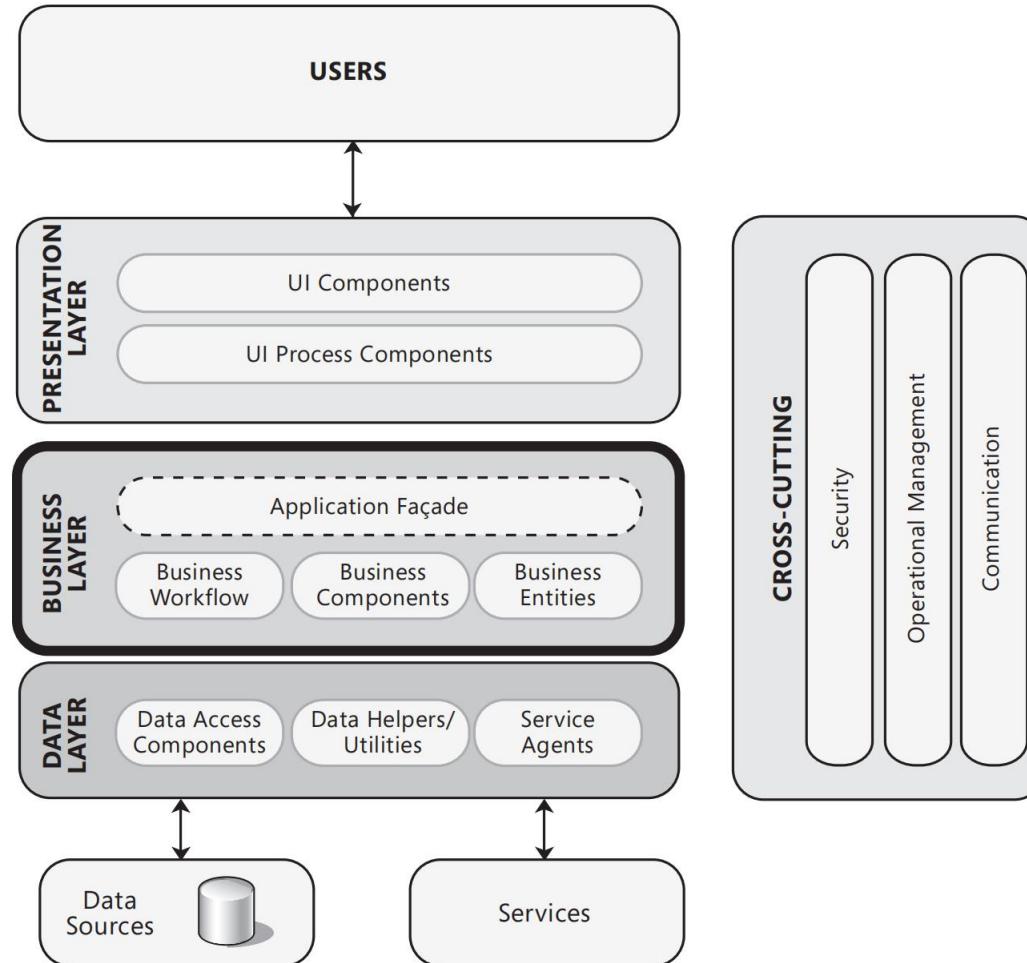
➤ 什么是软件架构

- 《软件工程—实践者的研究方法》与《Microsoft Application Architecture Guide》
 - 程序或计算系统的软件架构是指系统的一个或者多个结构，它包括软件构件、构件的外部可见属性以及它们之间的相互关系
 - 软件架构关注构件接口的公共方面，构件的私有细节属于构件的实现内容，不属于架构
- 《企业应用架构模式》
 - 最高层次的系统分解
 - 系统中不易改变的决定
- 《面向模式的软件架构》
 - 软件架构是对子系统、软件系统组件以及它们之间相互关系的描述。子系统和组件一般定义在不同的视图内，以显示软件系统的相关功能属性和非功能属性。
 - 架构四视图
 - 逻辑视图
 - 进程视图
 - 物理视图
 - 开发视图

LOP与游戏前端架构

➤ 软件架构实例

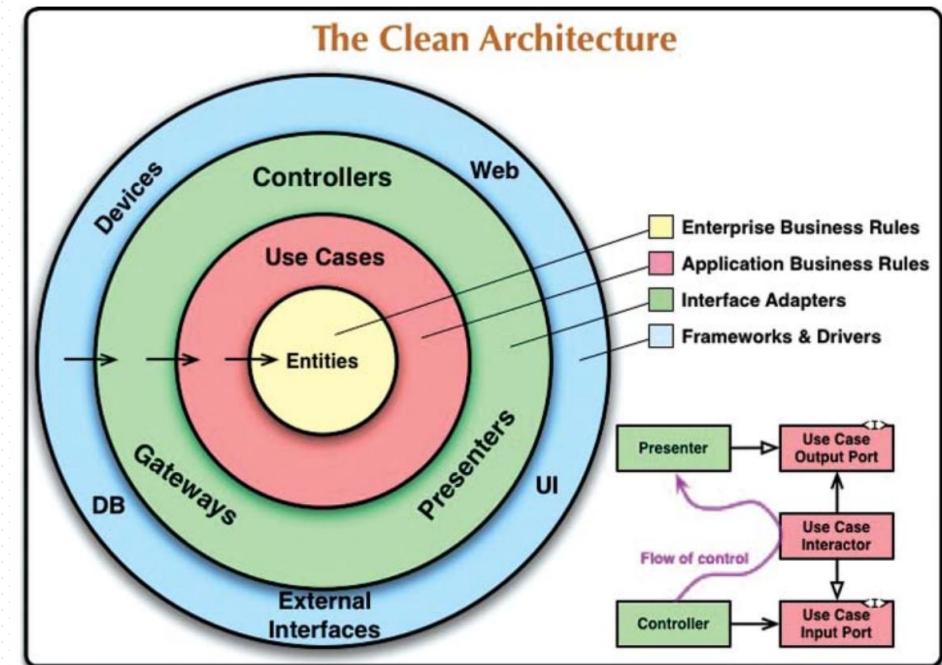
- 微软应用架构
 - 经典三层架构
 - 组件化
 - 配合dotnet框架



LOP与游戏前端架构

➤ 支持变化小结

- 区分变化与不变，并封装与隔离变化的部分
 - 接口不变，实现会变
 - 接口分离
 - 业务不变，实现会变
 - 策略分离
 - 抽象不变，具体会变
 - 抽象、封装与信息隐藏
 - DIP依赖倒置原则
 - 稳定依赖原则与稳定抽象原则
 - 《架构整洁之道》里，列出了几个实现细节的例子
 - 数据库
 - WEB
 - 应用程序框架
 - I/O
 - GUI
 - OS
 - 思考
 - 游戏引擎是实现细节么？



LOP与游戏前端架构

➤ LOP会是一个方案么？

- 回到游戏前端，相对来说
 - 什么是不易变的?
 - 我们能控制的，所谓业务
 - 比如三层架构的中间层与clean架构的里面二层
 - 游戏前端子系统：场景管理、资源管理、游戏对象、动作系统、战斗系统、剧情系统等
 - 什么是易变的?
 - 我们不太能控制的：UI交互、具体玩法逻辑、后台交互协议、数据表，包括引擎
 - 前端如何应对变化?
 - 解耦合
 - 与引擎
 - 与资源
 - 与表格
 - 与网络
 - 与UI

LOP与游戏前端架构

➤ LOP会是一个方案么？

- 语言如何提供解耦合?
 - 回顾一下LOP的开发方法
 - 应用层
 - 语言层
 - 解释器层
 - 底层服务（API）

我们类比一下计算机的硬件与软件，硬件相当于服务与解释器，它可以解释机器码，软件是领域层，语言层就是机器码，我们随时可以把软件完全抹去再重装，不会影响硬件。反过来，我们随时可以更换硬件，把软件装到新的硬件上即可，不影响软件的功能。

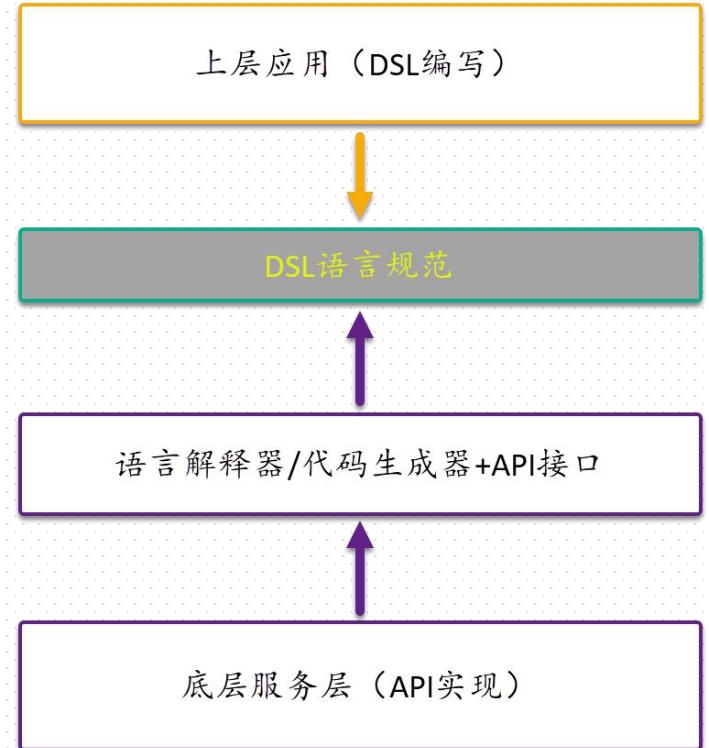
- The language-oriented methodology proceeds in three steps:
 - Design a domain specific language (DSL) for your core application logic.
 - Write your application in the DSL.
 - Build interpreters to execute your DSL programs, mapping your logical constructs to the actual machinery.

LOP与游戏前端架构

➤ LOP会是一个方案么？

- 语言如何提供解耦合?
 - 回顾一下LOP的开发方法
 - 应用层
 - 语言层
 - 解释器层
 - 底层服务（API）
 - 标准模型里上层主要使用DSL编写
 - 上层需要具有完备的领域模型，业务逻辑可以由DSL完整描述
 - 语言解释器的性能可能会是瓶颈，此时使用代码生成器将DSL翻译为高性能的GPL可能是更合适的选择，但这会损失一定的运行时的灵活性

DSL应用标准模型



- The language-oriented methodology proceeds in three steps:
 - Design a domain specific language (DSL) for your core application logic.
 - Write your application in the DSL.
 - Build interpreters to execute your DSL programs, mapping your logical constructs to the actual machinery.

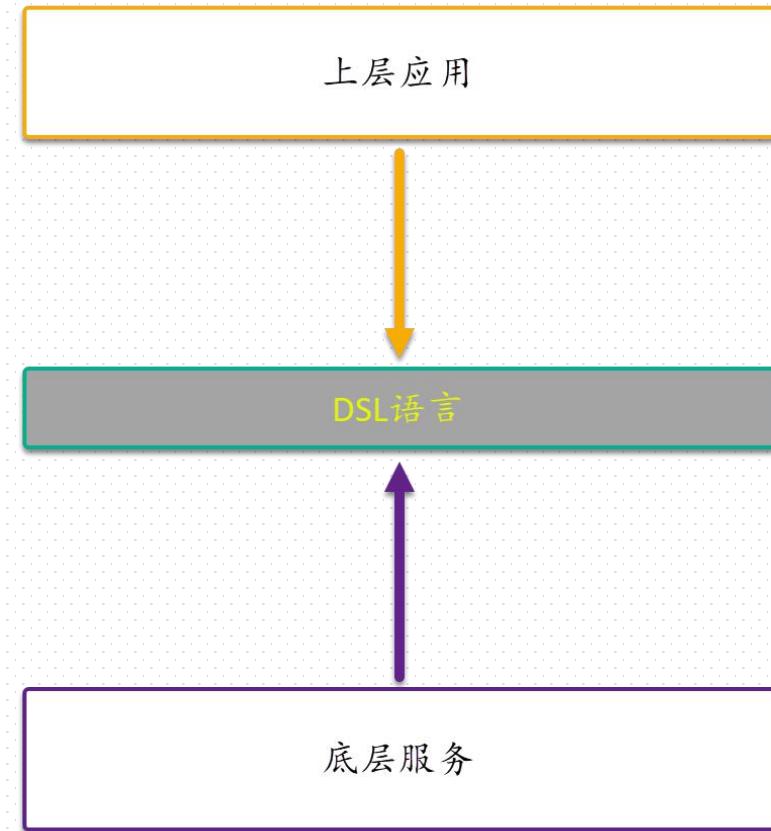
LOP与游戏前端架构

➤ LOP会是一个方案么？

- 语言在某种意义上也是一种接口
 - 接口是一个共享边界，软件的两个或多个组件通过它交换信息

an interface is a shared boundary across which two or more separate components of a computer system exchange information

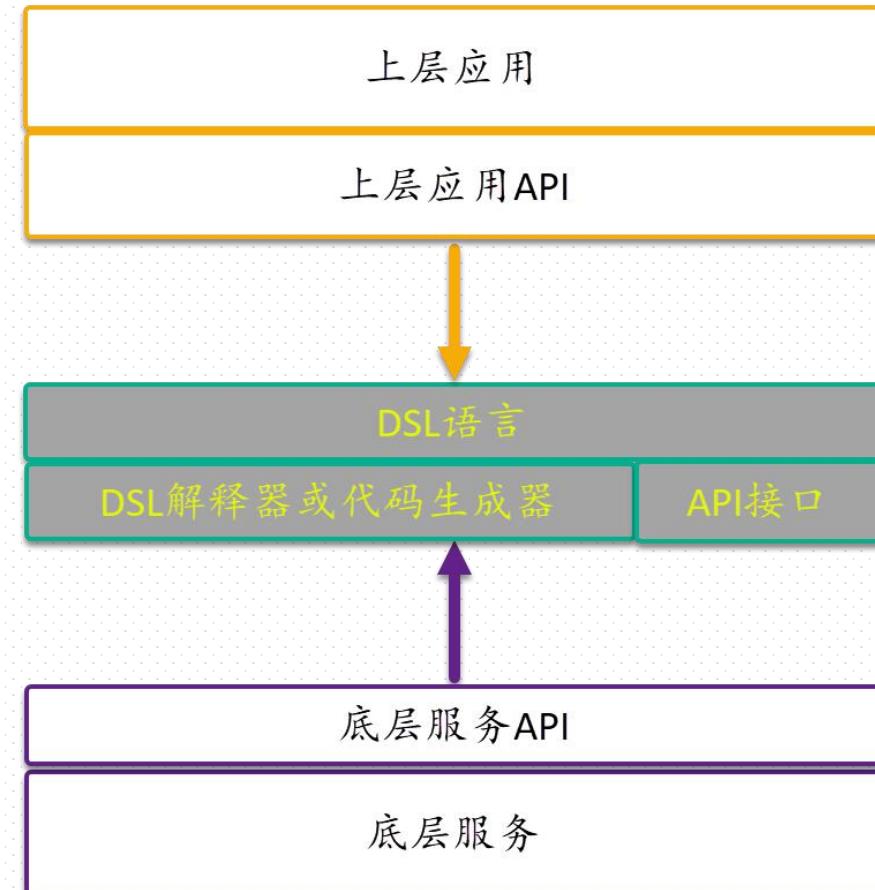
- 语言提供的依赖隔离几乎是阻断性的
 - 语言隔离的2层，互相间不再直接依赖
 - 2层都依赖语言



LOP与游戏前端架构

➤ LOP会是一个方案么？

- 所以还有一种DSL应用模型
 - DSL语言主要用作接口
 - 上层与下层各自提供API
 - DSL脚本通过两层的API把上下两层衔接起来
- 多层架构下的LOP
 - 在LOP方法里，一个多层次架构，底层对上层提供的服务，就是一个新的语言
 - 一个系统里可能存在多种不同复杂度的语言，每一个都定义了它的解释器所能提供的服务



LOP与游戏前端架构

➤ LOP会是一个方案么？

- 各种AAS
 - IAAS---基础架构（硬件）即服务
 - 基础架构与基础架构的使用方解耦
 - PAAS---平台即服务
 - 平台与平台的使用方解耦
 - SAAS---软件即服务
 - 软件与软件使用方解耦
 - FAAS---函数即服务
 - 函数与函数使用方解耦

LOP与游戏前端架构

➤ LOP会是一个方案么？

- 微软roslyn编译器工程
 - CAAS---编译器即服务
- 那么DSL呢?
 - LAAS---语言即服务?

Q&A

THANK YOU