## 代码生成及转换规则

### 概要说明

核心思想

编译的过程即把一段符号（代码）按一定规则转换为另一段符号（代码的过程）

所以我们可以将此过程视为一个简易的编译过程，即将一段符号（流程图型的描述）

转换为另一段符号（目标代码）。

所以本转换程序大量参考主流编译器的实现思想（简易）

1. 从流程图到最终生成代码的过程非常类似一个外部DSL(领域特定语言的解析)

所以该过程大量参考了编译器的相关流程(简易)是一个典型的管道架构风格

由于有相对完整的理论支撑，所以程序在一定程度上保证了最大的灵活和可维护

并且从图形生成代码有之前这样做的语言的成功先例。

1. 代码转换程序本身的输入是描述页面节点以及节点连线结构的数据结构(JSON)

通过前端的一次转换(Parse)获得需要保留的元数据将其转换为后台相对容易理解

的类抽象语法树型结构(AST)，这里我称之为原始抽象语法树前端将其传给后台

后台会首先对该前端传入的语法树结构进行分析检查是否合理这一过程除了将

前端传过来的JSON转换为后台更容易处理的数据结构外还包含一部分语义分析

语义检查功能（如case子句的条件不可为空等等）。此处转换成功后，会将这个

语法树根据业务规则进行变换多次（趟，当前只有一次）后，

成为一个利于代码生成的AST结构。

1. 执行代码生成逻辑，此处参考老马（Martin Flowler）《领域特定语言》最终一章

无视模型的代码生成部分进行生成。

1. 问题说明:这样设计使用AST讲编译器的前后分开是比较常见的做法，有以下几点好处

1),.若将来业务固定，甚至可以设计一门自己的文本型语言DSL，通过语法分析生成AST

2),也可以通过单独修改生成代码部分适应多个平台比如未来可能会生成到(JAVA)平台

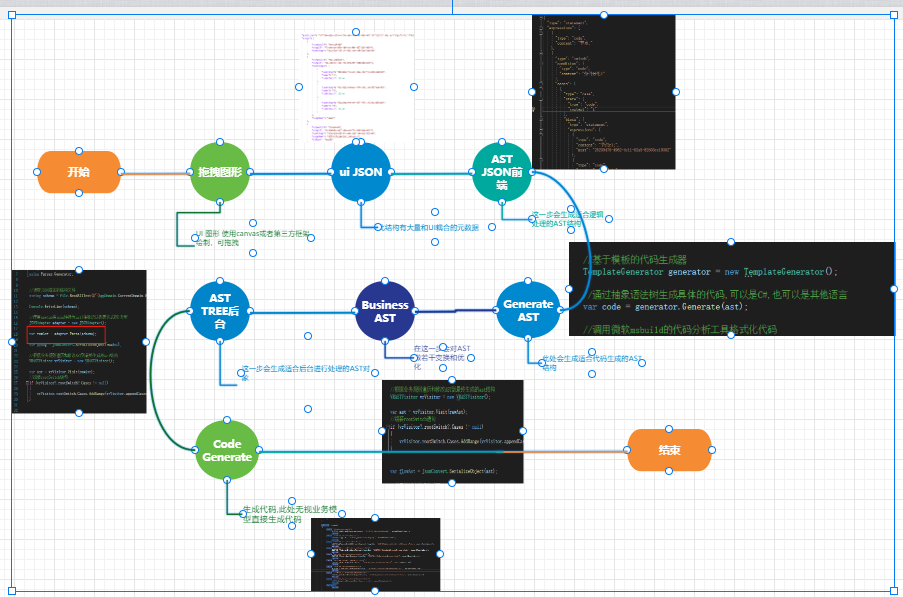
甚至解释执行,这主要归功于AST这个中间表示(IR).

3),关于最后是否使用无视模式的代码生成还是未来使用有模型的代码生成这个还有

灵活变更的空间

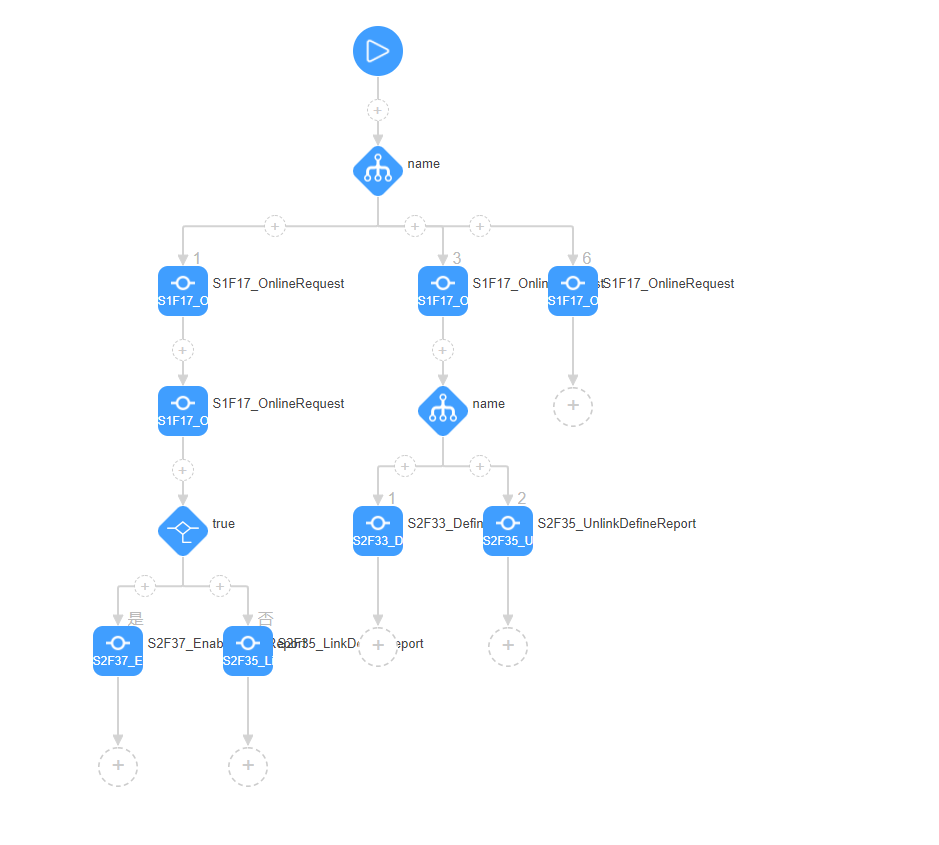
## 详细说明

[引擎架构:](结构示意图.pdf)



图形界面拖拽->ui JSON-> AST JSON-> AST TREE(后端)-> Bussiness AST Tree(后端)->Generate Code AST(后端)->Generate Code(后端) 管道风格。

结构图示:



为了充分说明引擎的功能，我们在这里举了一个相对复杂的例子

来说明代码生成的策略

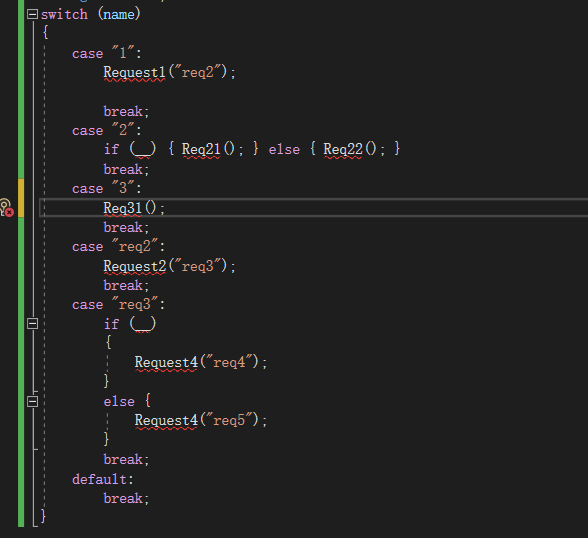
规则有两条:

1. 所有if switch下的第一个子节点和start后直到第一个分支（含）之前的所有节点我们称之为实节点

2.其他节点均为虚节点

3.此处的虚节点和实际节点为我们在系统中引入的概念,所有的实际节点将会直接生成对应的代码，所有的虚拟节点均生成case结构,自上而下形成一种类链表的结构来处理包和请求的收发

上图的结构会生成如下代码块(不看具体节点信息只看结构):



改进问题1)是否引入模型来做代码生成

2)图灵完毕问题

3)无论前端拖拽还是后台的代码生成是否有更好的模型比如合理利用状态机和async await

是否可抽象出一个文本的前端语言类似graphviz（一种DSL）更适合这一领域使得用户可以快速生成代码又不失图形化的优势。

参考 <http://magjac.com/graphviz-visual-editor>

1. 后端是否可以生成到不仅C#的多个平台甚至解释执行
2. 无论对生成的代码做调试路径点亮还是目标优化可读性优化是否还有空间

参考资料:

1. Martin Flowler《领域特定语言》.
2. 编译原理龙书.
3. SICP(计算机程序的构造和解释)
4. .networkflow core（该引擎的实现附带了一个文本方式的DSL实现）
5. graphviz语言一种图形描述的DSL

注意： :红字部分是术语可以参看相应文章。