Breed Rational Application Develop v0.1

author by zhao_nf(ttchgm@gmail.com)

1 系统说明

1.1 系统开发环境

系统采用 python 开发，模版规则引擎采用 jinja2，文档生成器采用自定义开发。

1.2 系统定位

这版的系统代号为 breed，意喻繁殖，属于 Rational Application Develop 工具类型，这个开发工具没有 IDE，为了减少开发难度而没有做 IDE 和控件拖拽，从某种意义上 IDE 是会降低开发效率，增加开发友好度。但为了这个豪华的表现层付出了非常昂贵的代价不是非常值当。

暂时 0.1 版本只支持 html,java,java 泛型和匿名注释的转换工作。

1.3 系统层级结构

系统主要功能实现，

转换程序(translate)：支持部分 java 的 SSH 框架中部分业务定义和业务逻辑代码到 breed 语言的语法。

分析程序(Parser)：分析 breed 程序，生一个 breed 词法集合文件。

生成程序(Make)：根据词法集合文件，分析出要生成的源格式（注意这个格式不是源代码）

模板库(template)：可以让用户根据源格式自定义一套自己的工作环境的对应的生产模版库，以便生成最终代码。

java 代码 ->Translate -> Parser ->Make ->template ->target code。

用户编写的 Breed 源代码 →Parser ->Make ->template ->target code。

系统主要支出如上两种模式。

2 系统架构

2.1 系统模型

系统由翻译层，由 3 个程序组成。一个是 lex.py ，一个是 Inpter.py，一个是 action.py

系统提供部分

{

输入一个 java 代码，

首先把它分析成词单元，然后根据词单元，分拆接口。

分拆出建议代码，然后根据建议代码到分析器

分析器整理好数据结构。

}

用户需要自己来修改 action.py

{

根据分析结果集合来，书写自己的生成代码集合。(当然这里会提供一个基本集合和 breed 的语法对照注释，以方便用户编写)

}

以下是书写层次规范和框架：

根据 breed 语法结构中定义的有：数据结构(DataStruct) ，页面(Page)，业务逻辑(Logic)

关联(Assioc)，检测(Check)，持久化(Persistence)，算法(Algorithm)。

比如一个 SSH 模型的 java 代码（我们不去解析架构内的配置文件的代码），假设有

Model，Control，Services，DAO，View 几个层面。

那么我们首先会解析 Model 层，转换到 DataStruct 层中。

DAO 层中的操作数据的逻辑，我们会转换到 Persistence 中。相应的增加基本的增删改查。

Control 和 Services 层的代码，我们都会归纳到 Check 和 Logic 层中。

View 层的代码，我们会归纳到 Page 层中。

算法层，负责提供基本的算法组件。会以智能提示注释的方式提供给用户，用户可以自行修改代码。

关联层，则是分析过的结果后通过集合关联和单独关联，来优化 breed 代码之用。

如上即是一个转换规范，也是一个书写规范。

2.2 语法支持目标和基本内容

系统的主语法方向有：

数据结构(DataStruct)，页面(Page)，业务逻辑(Logic)
关联(Assioc)，检测(Check)，持久化(Persistence)，算法(Algorithm)

其中有独立文件的微： DataStruct Page Persistence

其中以组件或者内嵌代码的有：Logic， Assioc， Check， Algorithm

系统主要目标：

1、实现页面的定义和集合化简化操作。
2、前端脚本的定义。
3、数据结构的定义。
4、检测 规范和组件化。
5、逻辑 独立化。
6、实现算法的复用和公式化。
7、持久层更易于优化。
8、公式化。

公式化：假设 Logic 层中描述一个例子：

假设投一个保单：

那么进入后端必然是这样：

```
function Regplc( DataStruct.plc )
{
    [
        begintrans
        check DataStruct.plc.Person.Apl
        check DataStruct.plc.Person.Isd
        check DataStruct.plc.plcinfo
        check DataStruct.plc.subject
        for (albt : DataStruct.plc.Lbts){
            check albt
        }
        [ check raise  → {!}  Rollback  : {=}  insert DataStruct.plc into Persistence.plc ]
        commit
    ]
}
```

如上就是检测投保信息和插入一条投保单记录的过程。[ ]代表在这个函数使用一个公式化的逻辑表达方式，如果你写过嵌入式 SQL 的化，基本上一个业务的过程是可以这样写的。

那么有人问了如果我的业务逻辑非常复杂，要分成多表存储或者中间的值需要在投保的时候跟其他的平台进行交互。这种情况如何写？

字段分表的话 这个比较简单。只要在 Persistence 中进行就可以。

```
Persistence Plc

insert into Param.apl into  Persistence.apl
insert into  Param.isd into Persistence.isd
...
for (albt : Param.lbts )
    insert …..
```

这样的语句是一个基本的语法词汇析构，然后具体是生成存储过程还是生成目标代码需要你来编写模版。

比如 Persistence 目录下的 plc
上面一条 Insert 语句对应一个模版语句块
你可以使用@Import 来分拆文件解析，也可以直接在一个文件中写入全部的模版。
当然你也可以在这个目录中写一个关键字 Insert.all。编译器会识别这个关键字，来把所有的 Insert 语句都按这个关键字来进行。

投保的是跟其他平台交互的时候可以通过

函数调用的方式进行。

3 基本定义

系统分成，显示，检测，数据结构 ，逻辑段。四大部分。

系统的包逻辑引用采用 Import 来导入。切只能放在非@ 段中。

3.1 文件关键字

3.1.1 Page

文法

Page ID stat

定义

标注程序是一个页面程序。

3.1.2 Component

文法

Component ID stat

定义

标注程序是一个组件程序。

3.1.3 DataStruct

文法

DataStruct ID stat

定义

标注程序是一个结构说明

3.1.4 Persistence

文法

Persistence

定义

标注程序是一个持久层结构。

3.2 段

文法:

@ID stat @end

定义

所有的程序结构定义都由@段和@end 内来定义。

ID 由如下几种构成

define

此关键字用来定义一些语句。

function

此关键字用来定义函数。

assioc

此关键字用来定义关系。

3.2.1 define 段

文法

@define stat @end

定义

此段主要用来定义数据结构，页面的具体层次信息和结构，

此段也可以用来定义 Interface

interface interfacename{

function 1

….

function n

}

3.2.2 function 段

文法

@function stat @end

定义

我们首先介绍一下@function 中的东西

首先@function 下支持：

function 函数定义，定义格式与 C 语言类似

不过他没有返回值

定义格式如下：

```
function id ( [in] paramid1 , [in] paramid2,[out] paramid3){
}
```

check 过程定义，定义格式与上面一样。

```
Check id ( [in] paramid1 , [in] paramid2,[out] paramid3){
}
```

3.2.3 assioc 关联段

文法：

@assioc stat @end

3.3 全局关键字

Interface

用来生成接口的声明

系统支持单继承。

function

函数有[IN] [OUT] [IN_OUT] 三种参数。

用来生成函数的声明

4 类型化

struct, function , check ,interface 这些都可以以类型化存在。