|  |
| --- |
|  |
| Writing Signatures for eyoung |
|  |
|  |
| **eyoung.father@gmail.com** |
| **2014/3/21** |

|  |
| --- |
|  |

目录

[1. 总体结构 2](#_Toc383180046)

[1.1. 语法定义 2](#_Toc383180047)

[1.2. 语法说明 2](#_Toc383180048)

[2. 规则前言 2](#_Toc383180049)

[2.1. 语法定义 2](#_Toc383180050)

[2.2. Prologue Code 3](#_Toc383180051)

[2.3. %output选项 4](#_Toc383180052)

[2.4. %file-init选项 4](#_Toc383180053)

[2.5. %file-finit选项 5](#_Toc383180054)

[2.6. %work-init选项 6](#_Toc383180055)

[2.7. %work-finit选项 7](#_Toc383180056)

[2.8. %event-init选项 8](#_Toc383180057)

[2.9. %event-preprocessor选项 9](#_Toc383180058)

[2.10. %event-finit选项 10](#_Toc383180059)

[2.11. %event选项 11](#_Toc383180060)

[2.12. %import选项 11](#_Toc383180061)

[3. 规则体 13](#_Toc383180062)

[3.1. 语法定义 13](#_Toc383180063)

[3.2. 语法说明 14](#_Toc383180064)

[4. 规则后记 16](#_Toc383180065)

[4.1. 语法定义 16](#_Toc383180066)

[4.2. 语法说明 16](#_Toc383180067)

# 总体结构

## 语法定义

eyoung\_file**:**

prologue\_opt TOKEN\_DPERCENT

signature\_opt TOKEN\_DPERCENT

epilogue\_opt

**;**

## 语法说明

eyoung IPS规则文件从结构上大致分为以“%%”分隔的三个部分：规则前言、规则体和规则后记。这三个部分都是可选的，但是分隔符“%%”是不能省略的。这一点，eyoung IPS规则文件同bison的语法文件类似。

示例：

<prologue>

%%

<signature>

%%

<epilogue>

# 规则前言

规则前言主要用来定义事件、函数入口点、外部文件引用、外部函数声明等等信息。

## 语法定义

prologue\_opt**:**

empty

**|** prologue\_list

**;**

prologue\_list**:**

prologue

**|** prologue\_list prologue

**;**

prologue**:**

TOKEN\_PROLOGUE\_CODE

**|** TOKEN\_OUTPUT TOKEN\_STRING

**|** TOKEN\_IMPORT TOKEN\_STRING

**|** TOKEN\_FILE\_INIT TOKEN\_STRING

**|** TOKEN\_FILE\_FINIT TOKEN\_STRING

**|** TOKEN\_WORK\_INIT TOKEN\_STRING

**|** TOKEN\_WORK\_FINIT TOKEN\_STRING

**|** TOKEN\_EVENT\_INIT TOKEN\_STRING TOKEN\_STRING

**|** TOKEN\_EVENT\_PREPROCESSOR TOKEN\_STRING TOKEN\_STRING

**|** TOKEN\_EVENT\_FINIT TOKEN\_STRING TOKEN\_STRING

**|** TOKEN\_EVENT TOKEN\_STRING TOKEN\_STRING

**;**

empty**:**

**;**

这里，prologue\_opt部分要么是empty，表示当前规则文件不存在prologue部分；要么是一个prologue\_list链。prologue\_list由不少于一个prologue组成，每一个prologue都是一个独立的前言定义。

## Prologue Code

Prologue Code是使用成对的“%{”和“}%”包含的C语言代码，代码的格式与C99标准兼容。Prologue Code中的代码会被完整地拷贝到中间代码.eyc文件中，用于后续的实时编译工作。例如：

**%{**

#include <stdio.h>

#include "myheader.h"

#ifdef something

#undef something

#endif

#define something

*/\**

*\* c style comment*

*\*/*

*//c++ style comment*

**typedef** **struct** my\_struct

**{**

**int** a**;**

**}**my\_struct\_t**;**

**extern** **int** my\_external\_func**(void);**

**static** **int** aaaa**;**

**static** **int** my\_static\_func**(void);**

**}%**

## %output选项

**格式**：

%**output** “*file-name*”

**说明**：

%output选项用来定义当前规则文件经过预处理之后的中间文件名称，中间文件名称使用字符串的形式给出——使用双引号包含的字符序列。默认情况下，中间文件的名称是当前规则文件名称后添加“.c”。例如，如果当前处理的规则文件是http.ey文件，默认会被预处理称http.ey.c文件。如果使用%output “http.eyc”选项，则会将http.ey预处理保存为http.eyc文件。**推荐使用.eyc作为中间文件的扩展名**。

**示例**：

%output “http.eyc”

## %file-init选项

**格式**：

%**file-init** “*function-name*”

**说明**：

%file-init选项用来向eyoung IPS引擎注册一个构造函数指针，该函数指针类型定义在include/libengine\_type.h中：

***typedef******int******(\*****file\_init\_handle****)(****engine\_t eng****);***

该函数在所有规则被解析完成，且JIT实时编译器完成动态编译和连接之后，由eyoung IPS引擎按照规则文件解析的顺序，依次调用通过%file-init注册的文件初始化函数。**函数返回0表示执行成功，否则表示执行失败。eyoung IPS引擎发现执行失败的初始化函数后，就会停止加载规则。**

**注意**：由于这个初始化函数是在所有规则文件编译连接之后才被执行，所以要求这个函数具有**“外部”属性**，它可以是规则内定义的全局函数，也可以是其通过%import加载的动态链接库中的全局函数，但是无法使用static静态函数！

**示例**：

%{

#**include** “libengine.h”

**extern int** my\_file\_init(engine\_t eng);

}%

%**file-init** “my\_file\_init”

## %file-finit选项

**格式**：

%**file-finit** “*function-name*”

**说明**：

%file-finit选项用来向eyoung IPS引擎注册一个析构函数指针，该函数指针类型定义在include/libengine\_type.h中：

***typedef******int******(\*****file\_finit\_handle****)(****engine\_t eng****);***

当规则卸载时，eyoung IPS引擎按照当初规则加载的顺序，依次调用使用%file-finit注册的析构函数，用来释放规则文件内分配的资源。函数返回0表示执行成功，否则表示执行失败。**执行失败意味着可能会有资源没有被释放从而造成资源的泄露，需要由规则开发人员仔细检查失败的原因。**

**注意**：与%file-init选项类似，%file-finit也要求注册的函数具有**“外部”属性**，无法使用static静态函数！

另外，还可以使用头文件include/libengine\_export.h中定义的**ey\_add\_file\_finit**宏，在%file-init注册的初始化函数中注册析构函数，此时允许注册的函数是规则内定义的static函数。**实际运行中，推荐使用本用法**。

**示例**：

(1)：

%{

#**include** “libengine.h”

**extern** **int** my\_file\_finit(engine\_t eng);

}%

%**file-finit** “my\_file\_finit”

(2)：

**%{**

#**include** “libengine.h”

**static int** my\_file\_finit(engine\_t eng);

**%}**

%**file-init** “my\_file\_init”

**%%**

**%%**

**static int** my\_file\_finit(engine\_t eng)

{

**return** 0；

}

**int** my\_file\_init(engine\_t eng)

{

ey\_add\_file\_finit(eng, my\_file\_finit);

**return** 0;

}

## %work-init选项

**格式**：

%**work-init** “*function-name*”

**说明**：

%work-init选项用来向eyoung IPS引擎注册**一个且最多一个构造**函数指针，该函数指针类型定义在include/libengine\_type.h中：

***typedef******int******(\*****work\_init\_handle****)(****engine\_work\_t \*work****);***

当一个engine\_work\_t对象被创建时，eyoung IPS引擎自动调用这个注册的函数。该函数用来维护work级别的资源的分配和初始化，work的具体概念请参考《Programming Guide for eyoung》。函数返回0，表示初始化成功，否则表示失败。执行失败时，会造成engine\_work\_t对象的创建失败。

**注意**：与%file-init选项类似，%work-init也要求注册的函数具有**“外部”属性**，无法使用static静态函数！

此外，还可以使用头文件include/libengine\_export.h中定义的宏**ey\_set\_userdefine\_work\_init**也可以完成相同功能。此时需要在%file-init注册的初始化函数中注册调用该宏完成注册工作。使用这种方式时允许注册的work-init函数是规则文件内定义的static函数。**实际运行中，推荐使用本用法**。

**示例**：

(1)：

%{

#**include** “libengine.h”

**extern** **int** my\_work\_init(engine\_work\_t \*work);

}%

%**work-init** “my\_work\_init”

(2)：

**%{**

#**include** “libengine.h”

**static int** my\_work\_init(engine\_work\_t \*work);

**%}**

%**file-init** “my\_file\_init”

**%%**

**%%**

**static int** my\_work\_init(engine\_work\_t \*work)

{

**return** 0；

}

**int** my\_file\_init(engine\_t eng)

{

ey\_set\_userdefine\_work\_init(eng, my\_work\_init);

**return** 0;

}

## %work-finit选项

**格式**：

%**work-finit** “*function-name*”

**说明**：

%work-finit选项用来向eyoung IPS引擎注册**一个且最多一个析构**函数指针，该函数指针类型定义在include/libengine\_type.h中：

***typedef******int******(\*****work\_finit\_handle****)(****engine\_work\_t \*work****);***

当一个engine\_work\_t对象被销毁时，eyoung IPS引擎自动调用这个注册的函数。该函数用来维护work级别的资源的释放。具体的概念参考《Programming Guide for eyoung》。函数返回0，表示初始化成功，否则表示失败。**函数执行失败会造成work级的资源泄露，规则开发人员一定要仔细检查原因。**

**注意**：与%file-init选项类似，%work-finit也要求注册的函数具有**“外部”属性**，无法使用static静态函数！

此外，定义在头文件include/libengine\_export.h中的宏**ey\_set\_userdefine\_work\_finit**也可以完成相同功能。此时，需要在%file-init注册的初始化函数中注册调用该宏完成注册工作。这种用法允许注册的函数是规则内定义的static函数。**实际运行中，推荐使用本用法**。

**示例**：

(1)：

%{

#**include** “libengine.h”

**extern** **int** my\_work\_finit(engine\_work\_t \*work);

}%

%**work-init** “my\_work\_finit”

(2)：

**%{**

#**include** “libengine.h”

**static int** my\_work\_finit(engine\_work\_t \*work);

**%}**

%**file-init** “my\_file\_init”

**%%**

**%%**

**static int** my\_work\_finit(engine\_work\_t \*work)

{

**return** 0；

}

**int** my\_file\_init(engine\_t eng)

{

ey\_set\_userdefine\_work\_finit(eng, my\_work\_finit);

**return** 0;

}

## %event-init选项

**格式**：

%**event-init** *“event-name” “function-name”*

**说明**：

%event-init选项用来为名称为“event-name”的事件向eyoung IPS引擎注册**一个且最多一个构造**函数指针，该函数指针类型定义在include/libengine\_type.h中：

***typedef******int******(\*****event\_init\_handle****)(****engine\_work\_event\_t \*event****);***

当一个engine\_work\_event\_t对象被创建时，eyoung IPS引擎自动调用这个注册的函数。该函数用来维护event级别的资源的分配和初始化，event的具体概念请参考《Programming Guide for eyoung》。函数返回0，表示初始化成功，否则表示失败。执行失败时，会造成engine\_work\_event\_t对象的创建失败。

**注意**：与%file-init选项类似，%event-init也要求注册的函数具有**“外部”属性**，无法使用static静态函数！

此外，定义在头文件include/libengine\_export.h中的宏**ey\_set\_userdefine\_event\_init**也可以完成相同功能。此时，需要在%file-init注册的初始化函数中注册调用该宏完成注册工作。这种做法允许注册的函数是规则内定义的static函数。**实际运行中，推荐使用本用法**。

**示例**：

(1)：

%{

#**include** “libengine.h”

**extern** **int** my\_event\_init(engine\_work\_event\_t \*event);

}%

%**event** “my\_ev” “void”

%**event-init** “my\_ev” “my\_event\_init”

(2)：

**%{**

#**include** “libengine.h”

**static int** my\_event\_init(engine\_work\_event\_t \*event);

**%}**

%**event** “my\_ev” “void”

%**file-init** “my\_file\_init”

**%%**

**%%**

**static int** my\_event\_init(engine\_work\_event\_t \*event)

{

**return** 0；

}

**int** my\_file\_init(engine\_t eng)

{

ey\_set\_userdefine\_event\_init(eng, my\_ev, my\_event\_init);

**return** 0;

}

## %event-preprocessor选项

**格式**：

%**event-preprocessor** *“event-name” “function-name”*

**说明**：

engine\_work\_event\_t对象被创建后，可能分多次提交检测。%event-preprocessor选项用来为名称为“event-name”的事件向eyoung IPS引擎注册**一个且最多一个预处理**函数指针，该函数指针类型定义在include/libengine\_type.h中：

***typedef******int******(\*****event\_preprocess\_handle****)(****engine\_work\_event\_t \*event****);***

当一个engine\_work\_event\_t对象实例被提交检测时，eyoung IPS引擎自动调用这个注册的函数完成对象数据格式转换等预处理操作。event的具体概念请参考《Programming Guide for eyoung》。函数返回0，表示预处理成功，否则表示失败。**执行失败时，会造成engine\_work\_event\_t对象的不能被IPS引擎检测。**

**注意**：与%file-init选项类似，%event-preprocessor也要求注册的函数具有**“外部”属性**，无法使用static静态函数！

此外，定义在头文件include/libengine\_export.h中的宏**ey\_set\_userdefine\_event\_preprocessor**也可以完成相同功能。此时，需要在%file-init注册的初始化函数中注册调用该宏完成注册工作。这种做法允许注册的函数是规则文件内定义的static函数。**实际运行中，推荐使用本用法**。

**示例**：

(1)：

%{

#**include** “libengine.h”

**extern** **int** my\_event\_preprocessor(engine\_work\_event\_t \*event);

}%

%**event** “my\_ev” “void”

%**event-preprocessor** “my\_ev” “my\_event\_preprocessor”

(2)：

**%{**

#**include** “libengine.h”

**static int** my\_event\_preprocessor(engine\_work\_event\_t \*event);

**%}**

%**event** “my\_ev” “void”

%**file-init** “my\_file\_init”

**%%**

**%%**

**static int** my\_event\_preprocessor(engine\_work\_event\_t \*event)

{

**return** 0；

}

**int** my\_file\_init(engine\_t eng)

{

ey\_set\_userdefine\_event\_preprocessor(eng,

my\_ev, my\_event\_preprocessor);

**return** 0;

}

## %event-finit选项

**格式**：

**%event-finit** *“event-name” “function-name”*

**说明**：

%event-finit选项用来为名称为“event-name”的事件向eyoung IPS引擎注册**一个且最多一个析构**函数指针，该函数指针类型定义在include/libengine\_type.h中：

***typedef******int******(\*****event\_finit\_handle****)(****engine\_work\_event\_t \*event****);***

当一个engine\_work\_event\_t对象被销毁时，eyoung IPS引擎自动调用这个注册的函数。该函数用来维护event级别的资源的释放，event的具体概念请参考《Programming Guide for eyoung》。函数返回0，表示释放成功，否则表示失败。**执行失败时，会造成engine\_work\_event\_t级资源泄露，规则开发人员必须检查这样的问题。**

**注意**：与%file-init选项类似，%event-finit也要求注册的函数具有**“外部”属性**，无法使用static静态函数！

此外，定义在头文件include/libengine\_export.h中的宏**ey\_set\_userdefine\_event\_finit**也可以完成相同功能。此时，需要在%file-init注册的初始化函数中注册调用该宏完成注册工作。这种用法允许注册的函数是规则内定义的static函数。**实际运行中，推荐使用本用法**。

**示例**：

(1)：

%{

#**include** “libengine.h”

**extern** **int** my\_event\_finit(engine\_work\_event\_t \*event);

}%

%**event** “my\_ev” “void”

%**event-finit** “my\_ev” “my\_event\_finit”

(2)：

**%{**

#**include** “libengine.h”

**static int** my\_event\_finit(engine\_work\_event\_t \*event);

**%}**

%**event** “my\_ev” “void”

%**file-init** “my\_file\_init”

**%%**

**%%**

**static int** my\_event\_finit(engine\_work\_event\_t \*event)

{

**return** 0；

}

**int** my\_file\_init(engine\_t eng)

{

ey\_set\_userdefine\_event\_finit(eng, my\_ev, my\_event\_finit);

**return** 0;

}

## %event选项

**格式**：

%**event** *“event-name” “event-type”*

**说明**：

《Programming Guide for eyoung》中: “**事件本质上是由LR语法分析过程中产生的非终结符，事件的名称就是非终结符的名称，事件的类型就是非终结符的类型**”。所以，事件的定义逻辑上是在协议分析器的设计过程中完成的，这里只是通过%event选项形式化地告知eyoung IPS引擎，包括事件的名称、事件的类型等关键属性，而且**这些属性必须与协议分析器的设计相匹配**。

%event选项的执行有两种效果，首先是在eyoung IPS引擎内部分配并初始化了一个ey\_event\_t的对象，用来记录与该事件相关的信息；其次，会在规则转换的中间代码中添加如下代码，用来定义一个在规则中可以直接使用的、与事件名称同名的**指针类型**：

**typedef** event-type \*event-name;

**示例：**

**%event** “response\_list” “void”

**%event** “response\_header\_server” “http\_response\_header\_t \*”

在中间代码.eyc文件中，会有如下定义：

**typedef** **void** \*response\_list**;**

**typedef** http\_response\_header\_t \*response\_header\_server**;**

## %import选项

**格式**：

**%import** “*dynamic-library-name*”

**说明：**

《Programming Guide for eyoung》的“二进制规则库”一节，介绍了eyoung IPS引擎提供的一种动态加载外部动态链接库的机制。这种机制就是通过%import选项将外部动态链接库引入到当前的程序运行镜像中。

%import的执行会导致四个作用：（1）eyoung IPS引擎使用GNU libdl的功能将dynamic-library-name指示的库动态加载到当前进程运行镜像中；（2）eyoung IPS引擎使用libelf的功能，将动态库中的eyoung扩展段.eyoung\_type和.eyoung\_ident的内容读取到预处理之后的源文件中；（3）如果动态链接库文件中包含.eyoung\_init段，则读取次段中包含的构造函数入口地址并执行，完成.so文件加载之后的第一次初始化操作；（4）如果动态链接库文件中包含.eyoung\_finit段，则读取其中的函数入口地址，当规则卸载、动态库卸载的时候由eyoung IPS引擎执行该析构函数，完成.so文件卸载时的资源释放工作。

对于第（2）点，以test/export\_test.c为例

#include <stdio.h>

#include "ey\_export.h"

**int** a**=**1**;**

**int** foo**(void** **\***link**,** **void** **\***event**)**

**{**

printf**("call foo, a=%d\n",** **++**a**);**

**return** 1**;**

**}**

**int** bar**(void** **\***link**,** **void** **\***event**)**

**{**

printf**("call bar, a=%d\n",** **++**a**);**

**return** 0**;**

**}**

**int** test\_init**(void** **\***eng**)**

**{**

printf**("call init, a=%d\n",** a**++);**

**}**

**int** test\_exit**(void** **\***eng**)**

**{**

printf**("call finit, a=%d\n",** a**--);**

**}**

**struct** s

**{**

**int** a**;**

**int** b**;**

**};**

**EY\_EXPORT\_IDENT(**a**,** **"extern int a;");**

**EY\_EXPORT\_IDENT(**foo**,** **"int foo(void \*link, void \*event);");**

**EY\_EXPORT\_IDENT(**bar**,** **"int bar(void \*link, void \*event);");**

**EY\_EXPORT\_TYPE(**s**,** **"struct s{int a; int b;};");**

**EY\_EXPORT\_INIT(**test\_init**);**

**EY\_EXPORT\_FINIT(**test\_exit**);**

加载之后，将会在预处理之后的中间文件中有如下记录：

**struct s{int a; int b;};**

**extern int a;**

**int foo(void \*link, void \*event);**

**int bar(void \*link, void \*event);**

同时，test\_init会被执行。

# 规则体

## 语法定义

signature\_opt**:**

empty

**|** signatures

**;**

signatures**:**

signature

**|** signatures signature

**;**

signature**:**

signature\_lhs TOKEN\_COLON signature\_pipe\_list TOKEN\_SEMICOLON

**;**

signature\_lhs**:**

TOKEN\_INT

**;**

signature\_pipe\_list**:**

signature\_rhs\_list

**|** signature\_pipe\_list TOKEN\_PIPE signature\_rhs\_list

**;**

signature\_rhs\_list**:**

signature\_rhs

**|** signature\_rhs\_list signature\_rhs

**;**

signature\_rhs**:**

rhs\_name rhs\_condition\_opt rhs\_cluster\_opt rhs\_action\_opt

**;**

rhs\_name**:**

TOKEN\_ID

**;**

rhs\_condition\_opt**:**

empty

**|** TOKEN\_RHS\_CONDITION

**;**

rhs\_cluster\_opt**:**

empty

**|** TOKEN\_SLASH TOKEN\_ID TOKEN\_COLON TOKEN\_STRING

**;**

rhs\_action\_opt**:**

empty

**|** TOKEN\_RHS\_ACTION

**;**

## 语法说明

1. signature

规则体由若干条规则（signature）组成。每条规则的基本结构是：

*id* ***:*** *signature\_pipe\_list ;*

* 标点符号冒号“：”和分号“；”不能省略。
* id是一个十进制整数，是规则在eyoung IPS引擎中唯一的索引值，在同一个engine\_t对象中，id不允许重复，否则会被视为规则解析阶段的语法错误。
* signature\_pipe\_list的定义见2。

1. signature\_pipe\_list

pipe\_list是由竖线“|”分隔的子规则（每一条子规则称为signature\_rhs\_list，定义见下文），子规则间是“OR”的关系，即竖线两侧的子规则有一条命中即意味着整个signature的命中。例如：

*1* ***:*** *pipe0* ***|*** *pipe1* ***|*** *pipe2* ***;***

表示id为1的规则由三个子规则组成，分别是pipe0、pipe1和pipe2，语义上是pipe0 ***OR*** pipe1 ***OR*** pipe2

1. signature\_rhs\_list

不少于一个signature\_rhs组成

1. signature\_rhs

signature\_rhs是基本的检测单元，其格式是：

*<ev-name> [****(****<C-Format expr>****)****] [****/****<pp-name>****:”****<pp-signature>****”****] [****{****C-Format Code****}****]*

* 方括号表示可选内容
* 加大加粗的标点符号不能省略，否则会导致规则解析时的语法错误。
* <ev-name>是一个经过%event定义过的事件名称。
* <C-Format expr>被称为Condition是一个C99兼容的标量表达式。Condition可选，不写Condition时表示True。
* <pp-name>是注册的预处理器的名称，关于预处理器请参考《Programming Guide for eyoung》的介绍。
* <pp-signature>被称为Pre-Condition，是由<pp-name>预处理器负责定义和解释的预处理规则字符串。Pre-Condition可选，不写表示True
* <C-Format Code>被称为Action，是一个C99兼容的代码段。这段代码需要返回一个0/1型的返回值。Action可选，不写Action暗含返回值是True。
* signature\_rhs的求值方式可以用下边的伪代码表示：

**if**(Pre-Condition.Calc\_Value() == False)

**return** RHS-NOT-MATCH;

**if**(Condition. Calc\_Value() == False)

**return** RHS-NOT-MATCH;

**if**(Action.Run() == False)

**return** RHS-NOT-MATCH;

**return** RHS-MATCH

* Condition的表达式最终会被翻译成一个隐含的函数调用，例如

1: my\_event(β);

其中的Condition经过eyoung IPS引擎的解析后，会在中间代码文件中添加一个名为\_condition\_1\_0\_0的全局函数：

int \_condition\_1\_0\_0(engine\_work\_t \*\_WORK\_,

engine\_work\_event\_t \*\_THIS\_)

{

return (β);

}

该函数名称中”1\_0\_0”中的1是规则ID，；中间的0表示规则右侧用竖线“|”分隔的rhs\_pipe出现的顺序，从0开始记；最后一个0表示同一个rhs\_pipe内的事件出现的顺序，从0开始记。

由于翻译后的函数被添加了两个形参，分别是\_WORK\_和\_THIS\_，分别用来指代当前被检测的engine\_work\_t对象和engine\_work\_event\_t对象。所以，在β中可以直接使用\_WORK\_和\_THIS\_，看似好像是关键字一样。

* 与condition类似地，Action最终也会被翻译成一个隐含的函数，例如：

1: my\_event1{β1} my\_event2{δ1}

| my\_event1{β2} my\_event2{δ2}

;

其中的Condition经过eyoung IPS引擎的解析后，会在中间代码文件中添加：

int \_action\_1\_0\_0(engine\_work\_t \*\_WORK\_,

engine\_work\_event\_t \*\_THIS\_)

{

β1

}

int \_action\_1\_0\_1(engine\_work\_t \*\_WORK\_,

engine\_work\_event\_t \*\_THIS\_)

{

δ1

}

int \_action\_1\_1\_0(engine\_work\_t \*\_WORK\_,

engine\_work\_event\_t \*\_THIS\_)

{

β2

}

int \_action\_1\_1\_1(engine\_work\_t \*\_WORK\_,

engine\_work\_event\_t \*\_THIS\_)

{

δ2

}

这里编号的同Condition函数的编号规则一致。同样地，在action中也允许直接使用\_WORK\_和\_THIS\_，看似好像是关键字一样。

* 所有的Action和Condition函数，经过上述的转换后，都会被JIT实时编译器编译成机器指令并添加到当前进程的运行镜像中。eyoung IPS引擎在检测过程中，会直接找到编译之后的action和condition函数的入口地址并执行它们。

# 规则后记

## 语法定义

epilogue\_opt**:**

empty

**|** TOKEN\_EPILOGUE\_CODE

**;**

## 语法说明

规则后记就是一段可选的、C99兼容的代码段，所有的后记部分，都会在规则解析阶段被eyoung IPS引擎拷贝到预处理文件中。