

---

# KNIGHT

---

A Programming Language to protect the Security of Information Flow



2024 5

MADE BY BLANKHEART

# 目 录

1	程序语言 Knight .....	1
1.1	简介 .....	1
1.2	权限类型 .....	1
1.3	EBNF 抽象语法 .....	1
1.4	安全检测与语义 .....	3
1.5	直观语法 .....	4
1.6	Knight 语言对信息流的保护 .....	6
1.7	Knight 语言的实现与演示 .....	8
2	Knight 语言编译器 Trial .....	10
2.1	简介 .....	10
2.2	整体设计 .....	10
2.3	类型检查 .....	12
2.4	KnightAssembly .....	13
2.5	示例 .....	15
3	Knight 语言虚拟机 Steed .....	17
3.1	简介 .....	17
3.2	整体设计 .....	17
3.3	示例 .....	18
4	Knight 语言演示系统 Duke .....	19
4.1	简介 .....	19
4.2	整体设计 .....	19
4.3	支持命令 .....	20
4.4	示例 .....	20

# 1 程序语言 Knight

## 1.1 简介

Knight 语言是一种强类型的命令式程序语言，支持基本的语法结构，包括常量变量、逻辑和算术运算、分支循环、输入输出、递归调用内/外部函数等。最大的特点在于所有的变量和常量的类型为数据类型（整数、小数、字符串、布尔）和权限类型的混合类型。

数据类型会在编译期进行严格的检查，按照确定的类型检查规则进行类型检查和合理的类型转换，确保类型安全。而关于权限类型的检查，Knight 在设计上考虑到程序想要获得的权限和实际被给予的权限不一定相同，所以不仅会在编译期进行检查，保证程序的行为和预定义的权限相一致，还会插入对应的 KnightAssembly 代码（Knight 会被编译为 KnightAssembly，在虚拟机上运行），在运行时对权限进行检查，不符合权限规则的语句在运行时将不会执行被跳过，确保程序运行过程中的信息流安全。

## 1.2 权限类型

Knight 中的程序、常量、变量均有一个权限类型  $P$ ， $P = \{p_1, p_2, p_3, \dots, p_n\}$  是一个由原子权限构成的有限集合（所有常量的权限类型为一个空集）。如定义一个变量  $int\ x < root, use\_x >$ ，变量  $x$  具有数据类型  $int$ ，权限类型  $P_x = \{root, use\_x\}$ 。权限类型可以被理解为如果想要任意访问变量  $x$  的内容，应用程序的权限类型  $P$  这个集合必须包含权限类型  $P_x$ 。关于权限类型的定义、检查、运行规则会在 2.4 节安全检测与语义中进一步说明。

## 1.3 EBNF 抽象语法

### 1.3.1 翻译单元

一个由 Knight 语言编写的单文件就是一个翻译单元，翻译单元的 EBNF 语法如下图 1 所示。每个翻译单元由多个 *Permission*、*Invoke*、*Function* 定义构成，CODE\_EOF 是代码的结尾标记。

```
< Translation_Unit > ::= { < Permission_Definition > | < Invoke_Definition >
                           | < Function_Definition > } CODE_EOF
```

图 1 翻译单元的 EBNF 语法

### 1.3.2 定义

Knight 程序中的定义主要分为 4 种，*Permission*、*Invoke*、*Function*、*LocalVariable*，其 EBNF 语法如下图 2 所示，Knight 语言只支持局部变量所以只有 *LocalVariable* 定义。定义中需要的类型和参数等由 *Type*、*Parameter*、*Parameter* 列表给出。

```
< Permission_Definition > ::= PERMISSION IDENTIFIER { IDENTIFIER }
< Type > ::= (INT | DEC | STR | BOOL) [ "< IDENTIFIER { IDENTIFIER } >" ]
< Invoke_Definition > ::= INVOKE IDENTIFIER : [ < Type > ] IDENTIFIER
                           "(" [ < Parameter_List > ] ")"
< Function_Definition > ::= [ < Type > ] IDENTIFIER ( [ < Parameter_List > ] )
                           "{ " [ < Statement > ] " }
```

```

< Parameter >::= < Type > IDENTIFIER
< Parameter_List >::= < Parameter > {"," < Parameter >}
< LocalVariable_Definition >::= < Type > IDENTIFIER [":" < Expression >] ";"

```

图2 定义的 ENBF 语法

### 1.3.3 语句

Knight 语言语句的 ENBF 语法如下图 3 所示。支持的语句包括 If-else(分支判断)、While(循环)、Return(函数返回)、Exit(退出程序)、Sleep(程序暂停)、Input(输入)、Output(输出)、Assignment(赋值)、Expression(表达式)。

```

< Statement >::= < Compound_Statement > | < LocalVariable_Definition >
| < If_Statement > | < While_Statement > | < Return_Statement >
| < Exit_Statement > | < Sleep_Statement > | < Input_Statement >
| < Output_Statement > | < Assignment_Statement >
| < Expression_Statement >
< Compound_Statement >::= "{"{< Statement >}"}"
< If_Statement >::= IF "(" < Expression > ")" < Statement > [ELSE < Statement >]
< While_Statement >::= WHILE "(" < Expression > ")" < Statement >
< Return_Statement >::= RET [< Expression >] ";"
< Exit_Statement >::= EXIT ";"
< Sleep_Statement >::= SLEEP CONSTANT_INT ";"
< Input_Statement >::= INPUT IDENTIFIER{,IDENTIFIER} ";"
< Output_Statement >::= OUTPUT < Expression > {,< Expression >} ";"
< Assignment_Statement >::= IDENTIFIER "=" < Expression > ";"
< Expression_Statement >::= < Expression > ";"

```

图3 语句的 EBNF 语法

### 1.3.4 表达式

Knight 语言表达式的 ENBF 语法如下图 4 所示。支持的表达式包括逻辑 and、or、not、?=(等于)、!=、<、<=、>、>=比较运算，算术+、-、\*、/、%，取 Variable/Constant 的值，FunctionCall(内部函数调用/外部函数调用)等。

```

< Expression >::= < LogicOr_Expression >
< LogicOr_Expression >::= < LogicAnd_Expression > {"or" < LogicAnd_Expression >}
< LogicAnd_Expression >::= < Equality_Expression > {"and" < Equality_Expression >}
< Equality_Expression >::= < Relational_Expression > {"("?"|"!=") < Relational_Expression >}
< Relational_Expression >::= < PlusMinus_Expression > {"("<"| "<="| ">"| ">=")
< PlusMinus_Expression >}
< PlusMinus_Expression >::= < MulDiv_Expression > {"("<+"| "<-") < MulDiv_Expression >}
< MulDiv_Expression >::= < Unary_Expression > {"("<*"| "</"| "<%") < Unary_Expression >}
< Unary_Expression >::= ["-|"not"] (< Primary_Expression > | < FunctionCall_Expression >)
< Primary_Expression >::= "(" < Expression > ")" | IDENTIFIER
| CONSTANT_INT | CONSTANT_DEC | CONSTANT_STR | "true" | "false"
< FunctionCall_Expression >::= [IDENTIFIER "." IDENTIFIER "(" [< Expression > {"," < Expression >}] ")"

```

图4 表达式的 ENBF 语法

## 1.4 安全检测与语义

下面我们主要对 Knight 中与权限类型相关的定义、计算、检测等内容进行详细的阐述。

### 1.4.1 定义

- *Permission* 用来定义程序所需权限。例如 *permission USE\_Key, Position*，在编译时的权限检查中会假定程序具有 *USE\_Key, Position* 权限，在程序安装时会向用户请求这些权限。程序的权限类型与变量的权限类型一样均为一个集合，所以所有的 *Permission* 定义的权限都会被加入程序的权限集合 *P* 中。

- *Invoke* 严格来讲是对外部程序中函数的声明，此处统称为定义方便管理。运行时虚拟机会自动将 *Invoke* 定义的函数装载，可以在当前程序中使用 *app.function* 的格式进行调用。

- *Type* 定义由数据类型和后面紧跟的一个可选的权限类型构成。数据类型有 4 种：*int*（整数），*dec*（小数），*str*（字符串），*bool*（布尔）。例如 *str < P1, P2 > s: "Hello"*；定义了一个字符串变量 *s*，数据类型为 *str*，权限类型为 *P1, P2*，初值为 "Hello"；*bool t*；定义了一个布尔变量 *t*，数据类型为 *bool*，权限类型为 *空*。

### 1.4.2 权限类型计算规则

本节将会详细描述权限类型的计算规则，关于数据类型的计算和转换规则会在 3.3.1 数据类型检查小节中详细描述。

- 常量的权限类型为  $\emptyset$ ，变量 *x* 的权限类型为定义变量时定义的权限  $P_x$ 。
- 如果表达式计算的格式为  $e_1 \text{ op } e_2$ ，表达式  $e_1$  的权限类型为  $P_1$ ，表达式  $e_2$  的权限类型为  $P_2$ ，那么计算出的新表达式的权限类型为  $P_1 \cup P_2$ 。
- 如果表达式计算的格式为  $\text{op } e$ ，表达式  $e$  的权限类型为  $P_e$ ，那么计算出的新表达式的权限类型仍然为  $P_e$ 。

### 1.4.3 语句

- *If* 语句的信息流安全检查需要计算出条件表达式 *Expression* 的权限类型  $P_e$ （表达权限类型计算规则将在下一小节中描述）。假设在当前程序中 *Permission* 定义的权限类型为  $P$ ，那么在编译时需要保证  $P_e \subseteq P$ ，否则会报错。而在程序运行时，只有  $P_e \subseteq P$  *If* 语句才会正常按照表达式计算出的布尔值选择分支执行，否则整个 *If* 语句会被跳过不执行。

- *While* 语句的信息流安全检查需要计算出条件表达式 *Expression* 的权限类型  $P_e$ 。假设在当前程序中 *Permission* 定义的权限类型为  $P$ ，那么在编译时需要保证  $P_e \subseteq P$ ，否则会报错。而在程序运行时，只有满足  $P_e \subseteq P$  *While* 语句才会正常按照表达式计算出的布尔值执行循环，否则整个 *While* 语句会被跳过不执行。

- 执行 *Exit* 语句会直接退出当前正在执行的程序。

- *Sleep* 语句可以让程序睡眠 *N* 秒，暂停程序执行。

- *Output* 语句的信息流安全检查需要对于每个表达式，计算出 *Expression* 的权限类型  $P_e$ 。假设在当前程序中 *Permission* 定义的权限类型为  $P$ ，那么在编译时需要保证  $P_e \subseteq P$ ，否则会报错。而在程序运行时，只有满足  $P_e \subseteq P$  *Output* 语句才会正常打印该表达式的值，否则将不打印结果。

- *Assign* 语句的信息流安全检查需要计算出表达式 *Expression* 的权限类型  $P_e$ 。假设在当前程序中 *Permission* 定义的权限类型为  $P$ ，而被赋值的变量 *x* 的权限类型为  $P_x$ ，那么在编译时需要保证  $P_e \subseteq P_x \subseteq P$ ，否则会报错。而在程序运行时，只有满足  $P_x \subseteq P$ （ $P_e \subseteq P_x$  可以在编

译通过的条件下保证), 变量 $x$ 才会被正常赋值, 否则该赋值语句无效。

• **Assign**语句含有的赋值规则还有一个很重要的影响就是在函数调用中, 传入参数和传出返回值都被视作赋值行为, 所以也会遵循赋值规则。编译时, 不符合赋值规则的函数调用、返回行为均无法通过编译。而在运行时, 在传参的过程中, 对于每对形参和实参, 只有符合赋值规则, 才能完成传递。同样, 在传出返回值时, 会计算出一个表达式, 当符合赋值规则时, 该表达式才会被赋值给返回值。如果不符合赋值规则, 被调用函数中不符合赋值规则的形参获得的实参值会是一个历史值(垃圾值), 调用函数获得的返回值也是一个历史值(垃圾值), 可以理解为是一个未定义的行为。虽然当不符合赋值规则时不会影响函数语句本身的执行, 但是函数调用执行完成后必然会获得一个错误的函数返回值。这种设计的缺点是程序可能会计算出莫名其妙的结果, 并且难以 debug; 但好处也是显而易见的, 不会因为不符合赋值规则, 程序就停止执行函数调用而崩溃, 可以在保护信息流安全的同时不影响程序的持续运行。

## 1.5 直观语法

这里我们通过几个程序示例 **Fib.k**、**BMI.k**、**Key.k** 来直观感受一下 Knight 语言的语法和编程风格, 详细的编程语法不在此处赘述。(本文中的程序示例均在 **VSCode** 中完成编辑, 语法高亮由本人自行编写的 **Knight**、**KnightASM** 语言插件提供)

• 示例 **Fib.k** 如图 5 所示。

```
# Fib.k

int Fib(int n)
{
    if(n<=2)ret 1;

    ret Fib(n-1)+Fib(n-2);
}

main()
{
    int i:1;
    while(i<=10)
    {
        output "Fib(",i,"): ",Fib(i);
        i=i+1;
    }
}
```

图 5 Fib.k

- 示例 BMI.k 如图 6 所示。

```
## BMI.k
-- This is a test
-- You can calculate bmi by this program
##
permission Look_BMI
str<Look_BMI> GetBMIResult(dec Height,dec Weight)
{
    dec BMI:Weight/Height/Height;
    str<Look_BMI> result;
    if(BMI<18.5)
        result="Thin";
    else if(BMI<=24.5)
        result="Healthy";
    else
        result="Fat";
    ret "\t"+result;
}

main()
{
    str name;
    int age:0;
    bool sex:false;    # false is woman,true is man
    dec height;
    dec weight;
    output "Please input your:";
    output "-- Name\n","-- Age\n","-- Sex\n","-- Height\n","-- Weight";
    input name,age,sex,height,weight;
    output "Please wait 3 seconds... .";
    sleep 3;
    output GetBMIResult(height,weight);
}
```

图 6 BMI.k

- 示例 Key.k 如图 7 所示。

```
# Key.k

permission USE_KEY

invoke HaveKey:str<USE_KEY> GetKey(str name)

main()
{
    output HaveKey.GetKey("PJD");

    ret;
}
```

图 7 Key.k

## 1.6 Knight 语言对信息流的保护

### 1.6.1 显式流保护

显式流指的是程序中的信息可以通过直接拷贝的方式进行流动,这是信息流安全中最常见的部分。Knight 对信息显式流进行保护的示例 ExplicitFlow.k 如图 8 所示。

```
# ExplicitFlow.k
permission USE_KEY

main()
{
    str<USE_KEY> key:"123456";
    str new_key;
    new_key=key;
    ret;
}
```

图 8 ExplicitFlow.k

ExplicitFlow 中程序的权限类型为{*USE\_KEY*}，定义了一个权限类型为{*USE\_KEY*}的变量 *key*，初始值为"123456"。再定义一个权限类型为空的变量 *new\_key*，当企图将 *key* 的值赋给 *new\_key* 时，编译会报错，阻止信息从高权限等级流向低权限等级。



### 1.6.2 隐式流保护

隐式流指的是可以通过程序中的控制结构来指示或者推测信息，一般情况下泄露的信息量小于显式流。Knight 对信息隐式流进行保护的示例 ImplicitFlow.k 如图 9 所示。

```
# ImplicitFlow.k

permission USE_NUMBER
invoke NumLib:int<USE_NUMBER> GetSectreNumber()

main()
{
    int<USE_NUMBER> number;
    number=NumLib.GetSectreNumber();

    if (number < 0)
        output "Negative";
    else
        output "Positive";

    ret;
}
```

图 9 ImplicitFlow.k

ImplicitFlow 中程序的权限类型为{*USE\_NUMBER*}，GetSecretNumber()是 NumLib 程序中的一个函数，返回一个类型为 $int < USE\_NUMBER >$ 的秘密数字。变量 *number* 是一个权限类型为{*USE\_NUMBER*}的整数，其值从 GetSecretNumber 函数中获取。控制语句中会对 *number* 的正负进行判断并打印字符串信息，有信息泄露的风险。

在编译时，会计算出  $number < 0$  这个表达式的权限类型为{*USE\_NUMBER*}，检查程序的权限类型是否包含该集合，不包含则报错。ImplicitFlow 能够通过编译器对权限的检查，完成编译。而在运行时，该程序可能被用户拒绝给予权限*USE\_NUMBER*，所以在编译检查的同时就插入了对应的 Test 和 Jump 等相关的 KnightAssembly（具体实现会在后续贯穿工具链工程描述的示例程序 Pi 中体现），这样就可以在运行时检查程序是否真的具有需要的权限，如果没有*USE\_NUMBER*，该控制语句就会被跳过，保证信息不会被泄露。

### 1.6.3 权限控制

对于 Output 这种打印语句，不属于显式流或者隐式流的范畴，可以简单归纳在通过权限控制进行保护之中。示例 SecurityAdd.k 如图 10 所示。

```

#SecurityAdd.k
permission USE_A,USE_B

int<USE_A,USE_B> SecurityAdd(int<USE_A> a,int<USE_B> b)
{
    ret a+b;
}

main()
{
    output SecurityAdd(1,1);
}

```

图 10 SecurityAdd.k

SecurityAdd 中程序的权限类型为{*USE\_A*,*USE\_B*},SecurityAdd 函数接收一个类型为 *int* < *USE\_A* > 的整数 *a*, 和一个类型为 *int* < *USE\_B* > 的整数 *b*, 返回值的类型为 *int* < *USE\_A*,*USE\_B* >。

Output 语句直接打印 SecurityAdd(1,1)的值, 两个参数 1 被赋值给 *a* 和 *b* 时均满足赋值规则。Output 语句如果想要打印一个类型为 *int* < *USE\_A*,*USE\_B* > 的值, 需要保证程序的权限包含{*USE\_A*,*USE\_B*}, 否则编译会报错。该程序满足要求, 能够通过编译时的检查。与上个隐式流例子中的 *if* 语句一样, 编译时已经插入了 Test 相关的 KnightAssembly, 所以在运行时如果不能通过权限检查, Output 语句会被跳过, 不打印任何信息, 确保信息流安全。

## 1.7 Knight 语言的实现与演示

### 1.7.1 常见编程语言实现方案

编译器、虚拟机、解释器三种实现方案对比如图 11 所示。越往左侧生成的程序的执行性能越高, 越往右侧工程实现的灵活度越高。

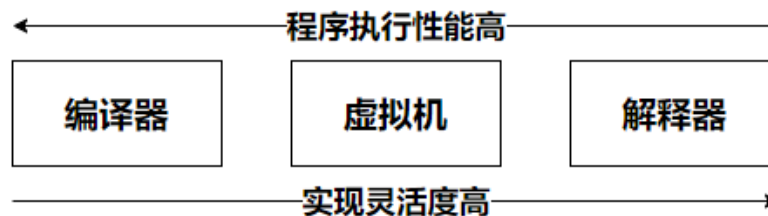


图 11 编程语言实现方式对比

- 编译器：编译器会将高级语言源代码编译为机器代码（此处的编译器概念指的是狭义的编译器，广义的编译器生成的目标代码不一定是机器码）。编译器实现的编程语言，编译后的程序在执行时可以直接运行在芯片之上，在所有实现方式中运行速度最快。缺点在于每次对源代码更改后想要运行都需要重新编译，而且由于目标代码必须是机器码，编译器的编写缺乏一定灵活度。编译器实现的代表语言有 Pascal、Fortran、C、C++、Go、Rust 等。

• **解释器**: 解释器可以直接对高级语言源代码解释执行, 不需要生成中间代码或者机器代码。由于解释器每次执行程序都要进行整个程序的分析工作, 与编译器相比执行速度较慢。但是解释器支持逐行读取和执行源代码 (REPL), 具有更高的灵活性和交互性, 实现也更简单, 很多脚本语言都采用解释器实现。解释器实现的代表语言有 Lua、Perl、Ruby、PHP、Python、JavaScript 等。

• **虚拟机**: 虚拟机是介于编译器与解释器之间的一种实现方式。通常由一个编译器将高级语言代码编译成某种中间代码, 如字节码, 这些中间代码会运行在虚拟机上, 虚拟机这种实现方式包括一个编译器和一个虚拟机。程序的执行速度在编译器实现和解释器实现之间, 中和了两者的优缺点, 代码编写的灵活度大于编译器。虚拟机实现的最著名的语言之一就是 Java, 被编译为 Java 字节码后运行在 Java 的虚拟机 JVM 之上, 包括 Kotlin 和 Scala 均可以编译为 Java 字节码在 JVM 上执行。其他虚拟机实现的语言还有 C#, Erlang 等。

• **转译器**: 还有一些高级语言采用转译器方式实现, 将源代码编译到另一种高级语言, 借由目标语言已经拥有的工具链运行。转译器实现使得程序编译流程更多, 比较麻烦。好处在于可以借用已有的语言的工具链, 目前有很多高级语言采用编译到 JavaScript 的方案。采用这种方式实现的语言有早期的 C++ (通过 Cfront 编译到 C)、TypeScript、CoffeeScript、Dart 等。

### 1.7.2 Knight 语言实现方案

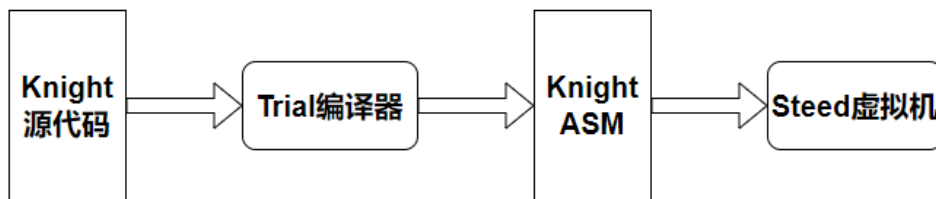


图 12 Knight 编译运行流程

在 Knight 语言的实现中, 基于程序执行性能以及实现灵活度的考虑, 我采用了虚拟机实现方案, 编译流程如图 12 所示。Knight 语言源代码会被 Knight 编译器 Trial 编译为 KnightAssembly, KnightAssembly 会在虚拟机 Steed 上运行。

### 1.7.3 Knight 语言效果演示



图 13 Duke 系统

由于 Knight 语言的设计考虑到程序在实际运行中会被给予的权限与预期不一致, 所以如果想要演示 Knight 语言的完整功能, 需要模拟 Knight 程序的安装与运行, 以及对其他程序中函数的调用, 并且用户能够对 Knight 程序赋予、更改权限。

为此我特意开发了一个使用起来像操作系统的 Duke 系统, 如图 13 所示。通过 Duke 的 Shell 可以执行多种命令, 包括程序的安装、卸载、更改权限等。能够模拟 Knight 程序的执行环境, 对语言权限相关的功能进行直观的演示。

## 2 Knight 语言编译器 Trial

### 2.1 简介

```
blankheart0@blankheart0:~$ trial
Usage: trial [-LUTA] knight_file
----->>> option <<<-----
-L: Compile to lower-case KnightAssembly file (default)
-U: Compile to upper-case KnightAssembly file
-T: Scan the code, print tokens
-A: Parse the code, print abstract syntax tree
```

图 14 Trial 编译器

Trial 是 Knight 语言的编译器，如图 14 所示。Trial 通过词法分析、语法分析、代码生成等阶段，将 Knight 源码编译为 KnightAssembly。代码生成的过程中会对数据类型和权限类型进行严格的检查，插入合适的数据类型转换语句以及运行时权限检查语句，确保 Knight 的类型安全和程序运行时的信息流安全。

### 2.2 整体设计

Trial 编译器的整体架构如图 15 所示。

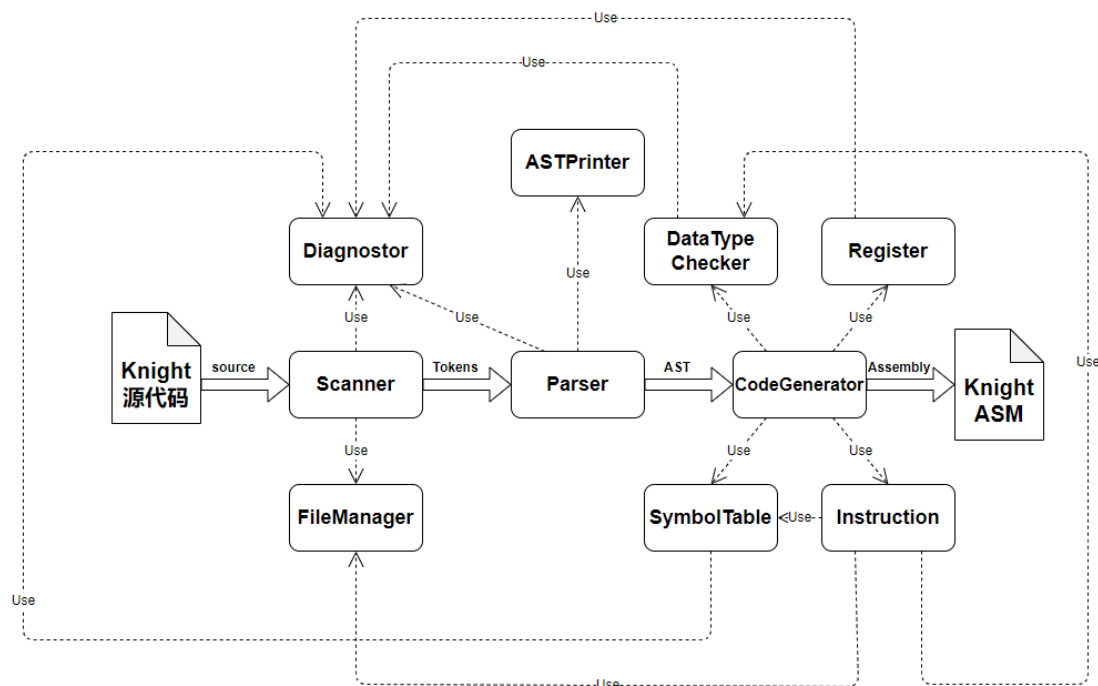


图 15 Trial 编译器架构图

• **Scanner:** 词法分析器。词法分析是编译器处理的第一步，负责将高级语言源代码转化为一系列的词法单元，即 Token。Token 是编译器后续进行语法分析等处理的基础。词法分析器会根据预定义的模式匹配源代码中的各种语法单元，如关键字（permission, if, while, int 等）、标识符（变量名，函数名）、常数（整数、浮点数、字符串、布尔）和运算符（+, -,

`*`, `/`, `==`, `!=` 等)。一旦词法分析器识别出一个有效的词法单元,它就会生成相应的 **Token**,并将这些标记组织成一个 **Token 流**或 **Token 列表**,供后续的语法分析和语义分析使用。同时,词法分析器也会检测并报告一些基本的语法错误,例如非法字符、未结束的字符串等。源代码中的空白字符和注释通常会在词法分析阶段被忽略,在编译的后续阶段就无需考虑这些内容。一些高级的词法分析器可能会在这一阶段构建和管理符号表,记录已经定义的标识符和它们的属性, **Trial** 中的符号表在代码生成阶段构建。**Trial** 中源代码经过 **Scanner** 会生成一个 **Token 列表**,传递给 **Parser** 语法分析器。**Trial** 编译器如果使用 `-T` 选项,会在经过对源代码的词法分析后,打印 **Token 列表**并结束后续编译器流程。

- **Parser**: 语法分析器。语法分析是编译器处理的第二步,负责将词法分析器生成的 **Token 流**转化为语法结构,通常以语法树或其他形式的抽象语法表示。语法分析器会根据预定义的文法规则验证 **Token 流**是否符合源语言的语法结构。如果 **Token 流**与文法规则不匹配,语法分析器会报告语法错误。一旦验证成功,语法分析器会根据语法规则构建一个语法树。语法树是源代码的结构化表示,它反映了源代码的层次结构和操作优先级。在某些编译器中,语法分析器可能会生成一个中间表示(如三地址代码、字节码或 LLVM IR 等),为后续的优化和代码生成做准备。在 **Trial** 中,所有的 AST 类由 **ASTNode** 统一继承而来,与 EBNF 语法描述中的非终结符一一对应。**Parser** 中有所有类型 AST 树的 **Parse** 函数。经过 **Parser** 生成的一整颗 AST 语法树会直接传递给代码生成器 **CodeGenerator** 做树遍历代码生成,不会生成其他中间表示形式进行优化工作,这是 **Trial** 编译器的一点不足。

- **ASTPrinter**: 语法树打印模块。每个 AST 子类都需要在 **ASTPrinter** 中实现一个 **Print** 方法,对自身进行打印,利用缩进和换行,从左至右体现语法树结构(根在最左)。通过观察 **Parse** 出来的语法树,可以协助进行编译器的 **debug**。**Trial** 编译器如果使用 `-A` 选项,会在经过对源代码的词法分析和语法分析后,打印语法树并结束后续编译器流程。

- **CodeGenerator**: 代码生成器。代码生成器是编译器中的关键组件,主要任务是根据经过语法分析器产生的中间表示(如语法树、三地址代码等),在符号表、寄存器管理等其他模块的协助下,生成目标机器的代码(如机器码、汇编码、字节码等)。在生成目标代码之前,代码生成器可能会应用一系列的优化策略,如常量折叠、死代码删除、指令调度等,以提高目标代码的性能和效率。代码生成器需要考虑目标平台的特性和限制,如指令集、寄存器分配、内存模型等,确保生成的目标代码能够在目标平台上正确、高效地执行。与词法分析器和语法分析器一样,代码生成器也可能检测 and 报告一些无法生成目标代码的错误,如类型不匹配、未声明的变量等。代码生成器的设计和实现质量直接影响到编译器的整体性能和生成代码的质量。**Trial** 中的 **CodeGenerator** 实现了每个 AST 子类的 **CodeGen** 方法。**CodeGenerator** 会根据 **Parser** 生成的 AST 做树遍历,与 **DataTypeChecker**、**Register**、**SymbolTable**、**Instruction** 等模块相互配合,生成面向 **Steed** 虚拟机的 **KnightAssembly**。**KnightAssembly** 是一种伪汇编,抽象层级略高于机器,可以在 **Steed** 虚拟机上面作为可执行程序运行。

- **DataTypeChecker**: 数据类型检查器。类型检查器负责检查源代码中的类型错误和类型不匹配。类型检查是编译器中的一个重要步骤,能够确保程序在运行时不会因为类型错误而崩溃产生不可预期的行为。类型检查器会检查表达式、函数参数、返回值等的类型,确保它们之间的类型是兼容的,以避免在运行时出现类型错误。当发现类型错误或不匹配时,类型检查器会生成错误信息,指出出现错误的位置以及可能的原因,帮助开发者及时修复错误。在某些情况下,类型检查器会自动进行类型转换,将一种类型的值转换为另一种类型,以满足运算或函数调用的要求。类型检查器的设计和实现需要考虑编程语言的类型系统和语义规则,以及编译器的整体架构。一个有效和高效的类型检查器能够大大提高编程语言的健壮性和开发效率。**Knight** 语言是一种强类型语言,本身设计的重点也在于语言的类型系统,所以 **Trial**

中对于数据类型和权限类型的检查至关重要。DataTypeChecker 实现了对于数据类型的检查、运算和转换，而权限类型的检查、运算和转换融合在了 Instruction 模块封装的各种指令之中，更多细节会在下文 3.3.1 节的数据类型检查部分描述。

- **Register:** 寄存器管理器，实现了对于寄存器的分配和回收。寄存器管理器是编译器优化的重要组成部分，它负责在编译器的代码生成阶段管理和分配寄存器，以最优化地利用计算机硬件的寄存器资源为目标。寄存器是计算机中速度最快的存储设备，直接参与到 CPU 指令的执行过程中，因此有效的寄存器分配能显著提高程序的性能。Trial 编译器中由于生成面向虚拟机的汇编指令，所以此处的寄存器并不是真实的物理寄存器。寄存器分配算法使用的是一种简单的贪心分配策略，会有一些 Load/Store 冗余。

- **SymbolTable:** 符号表。符号表是编译器中的核心数据结构，用于存储和管理程序中的标识符信息，包括变量、函数、类型等的名称、类型和作用域。符号表支持编译器进行标识符的解析、类型检查、作用域管理、存储分配和代码生成，是编译过程中关键的组件之一，直接影响编译器的正确性、效率和优化能力。Trial 中的符号表包括变量表和函数表，变量表的核心是一个哈希表的数组，能够实现变量的多重作用域。函数表中存储着函数的信息，而每个函数中均有一个变量表。因为 Knight 语言只支持局部变量，所以没有全局变量表。

- **Instruction:** KnightAssembly 指令集。Instruction 中封装着所有的 KnightAssembly 指令，会被 CodeGenerator 使用，并调用其他的模块获取需要的程序信息，通过 FileManager 模块将真正的汇编代码写入输出文件中。

- **FileManager:** 文件管理器，对输入和输出文件的所有操作进行封装，便于其他模块的使用。Scanner 会使用 FileManager 读取输入文件中的 Knight 源代码，然后进行词法分析。Instruction 会使用 FileManager 向输出文件中写入 KnightAssembly，完成整个代码的生成工作。

- **Diagnostor:** 错误诊断器。错误诊断器是提高编译器使用体验的重要部分，它负责识别、报告和处理源代码中的错误。当编译器在编译时遇到语法错误、类型错误或其他编译错误时，错误诊断器会生成相应的错误信息，并指出错误发生的位置以及可能的原因。这有助于开发人员快速定位和修复代码中的问题，提高代码的质量和稳定性。Trial 整个编译流程中所有模块发现的错误均由 Diagnostor 进行报告，Diagnostor 中有不同的错误类型，能够准确报告错误的类型以及错误代码的位置，清晰易懂。

## 2.3 类型检查

Knight 语言是一种强类型语言，本身设计的重点也在于语言的类型系统，所以 Trial 中对于数据类型和权限类型的检查和转换是 Knight 语言实现的重点之一。下面会分数据类型与权限类型进行讨论。

### 2.3.1 数据类型检查

由于在 Trial 的设计中，所有的运算都在寄存器中完成，所以类型的检查、运算和转换也全部基于寄存器。以下以 Add 指令为例，给出数据类型的运算和转换规则：

*Add + < Support Commutative law >*

*int + int -> int*

*int + dec -> (dec)int + dec -> dec + dec -> dec*

*- int + str -> error*

*- int + bool -> error*

```

dec + int -> dec + (dec)int -> dec + dec -> dec
dec + dec -> dec
- dec + str -> error
- dec + bool -> error

- str + int -> error
- str + dec -> error
  str + str -> str
- str + bool -> error

- bool + int -> error
- bool + dec -> error
- bool + str -> error
- bool + bool -> error

```

类型前面的括号,代表将当前数据类型转换成括号内的新类型,会插入相应的 Convert 指令。

### 2.3.2 权限类型检查

对于权限类型的检查严格按照 ENBF 中对各种语句权限类型规则的描述,此处不再赘述规则。Trial 会在编译时按照 permission 预先定义的权限对程序中的所有变量、常量的权限类型进行检查,不符合权限类型规则的语句将会报错。在静态检查的同时插入 Test 语句和对应的跳转语句,未来在 Steed 虚拟机上执行程序时,会执行 Test 语句,在运行时进行程序的权限检查,不通过权限检查的语句会经过跳转语句被跳过。这种设计是为了应对程序预期权限与实际获得权限不一致的情况,比较符合现实,通过静态与动态类型检查结合,提供对权限类型的双重保障。

## 2.4 KnightAssembly

Knight 编译器 Trial 的输出为 KnightAssembly,是一种面向 Knight 虚拟机 Steed 的汇编代码,抽象等级略高于机器代码。整个 KnightAssembly 指令集如下表 1 所示:

表 1 KnightAssembly 指令集

Opcode	Operand 1	Operand 2	功能
perm	permission	0/1	定义程序权限
test	register_R	permissions	检测程序权限
cvt	type	register_R	类型转换
var	type	variable	定义变量
load	register_R	constant/variable	加载数据
store	variable	register_R	存储数据

invoke	application	function	声明外部函数
func	function		声明内部函数
trans	register_X/R/Y	register_X/R/Y	寄存器值转移
call	function		函数调用
ret			函数返回
exit			程序退出
sleep	N		程序暂停
push	register_R		保存寄存器至栈
pop	register_R		从栈恢复寄存器
input	variable		输入
output	register_R/endline		输出
lable	N		标签
jmp	N		无条件跳转
jmpt	N	register_R	条件真跳转
jmpf	N	register_R	条件假跳转
or	register_R	register_R	或
and	register_R	register_R	与
equ	register_R	register_R	等于
nequ	register_R	register_R	不等于
les	register_R	register_R	小于
lequ	register_R	register_R	小于等于
gre	register_R	register_R	大于
gequ	register_R	register_R	大于等于
add	register_R	register_R	加
sub	register_R	register_R	减
mul	register_R	register_R	乘
div	register_R	register_R	除
mod	register_R	register_R	模
neg	register_R		负
not	register_R		非



## 2.5 示例

这里我们使用一个简单并贯穿后文的程序 Pi 来说明编译流程。

- 程序 Pi.k 如图 16 所示。

```
# Pi.k
permission SECRET

main()
{
    dec normal_pi:3.14;
    dec<SECRET> secret_pi:3.1415926;

    output "Normal Pi:",normal_pi;
    output "Secret Pi:",secret_pi;
}
```

图 16 Pi.k

- 执行命令 `trial -T Pi.k` 查看经过词法分析生成的 Tokens，如图 17 所示。

Line	Type	Lexeme
3	PERMISSION	permission
3	IDENTIFIER	SECRET
5	IDENTIFIER	main
5	LEFT_PAREN	(
5	RIGHT_PAREN	)
6	LEFT_BRACE	{
7	DEC	dec
7	IDENTIFIER	normal_pi
7	COLON	:
7	CONSTANT_DEC	3.14
7	SEMICOLON	;
8	DEC	dec
8	LESS	<
8	IDENTIFIER	SECRET
8	GREATER	>
8	IDENTIFIER	secret_pi
8	COLON	:
8	CONSTANT_DEC	3.1415926
8	SEMICOLON	;
10	OUTPUT	output
10	CONSTANT_STR	"Normal Pi:"
10	COMMA	,
10	IDENTIFIER	normal_pi
10	SEMICOLON	;
11	OUTPUT	output
11	CONSTANT_STR	"Secret Pi:"
11	COMMA	,
11	IDENTIFIER	secret_pi
11	SEMICOLON	;
12	RIGHT_BRACE	}
13	CODE_EOF	

图 17 Pi Token

- 执行命令 `trial -A Pi.k` 查看经过语法分析生成的语法树，如图 18 所示。



图 18 Pi AST（不完整）

- 执行命令 `trial Pi.k` 编译出 KnightAssembly 文件 `Pi.ks`，如图 19 所示。

```

perm    SECRET,0
func    main
var      dec,normal_pi(0)
load    r0,3.14
store   normal_pi(0),r0
var      dec,secret_pi(0)
load    r0,3.1415926
test    r1,<SECRET>
jmpf    0,r1
store   secret_pi(0),r0
lable   0
load    r0,"Normal Pi:"
output  r0
load    r0,normal_pi(0)
output  r0
output  endlne
load    r0,"Secret Pi:"
output  r0
load    r0,secret_pi(0)
test    r1,<SECRET>
jmpf    1,r1
output  r0
lable   1
output  endlne

```

图 19 Pi.ks

可以看到编译器在 `secret_pi` 的初始化赋值以及 `output` 语句前都插入了 `test` 语句，用于运行时检查程序的权限类型的集合是否包含这些权限，如果检查结果为 `false`，则会通过下方的 `jmpf` 指令跳过赋值和 `output`，保障程序的信息流安全。

### 3 Knight 语言虚拟机 Steed

#### 3.1 简介

```
blankheart0@blankheart0:~$ steed
Usage: steed KnightAssembly_file
```

图 20 Steed 虚拟机

Steed 是 Knight 语言的虚拟机，如图 20 所示。Steed 能够对 KnightAssembly 文件中的汇编指令逐行读取分析并存储，完成读取后会运行存储的指令。Steed 目前只接受由 Trial 编译器编译出的汇编文件，可以保证 KnightAssembly 语法的正确性，不会检查语法错误，从而加快执行效率。

#### 3.2 整体设计

Steed 虚拟机的整体架构如图 21 所示。

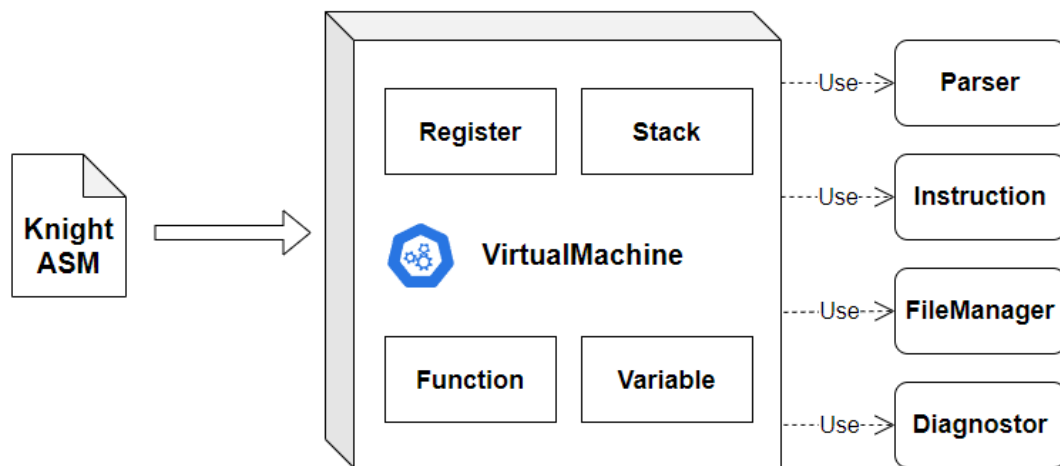


图 21 Steed 虚拟机架构图

- **VirtualMachine:** Steed 虚拟机的本体。内部有 Register、Stack、Function、Variable 等数据结构，存储着虚拟机运行过程中的所有数据和指令。Register 是寄存器变量，Function 是程序的函数存储着读取到的所有 Instruction，Variable 是函数中的变量。Stack 分为寄存器栈和上下文栈，前者用于寄存器的压入和弹出，后者用于保存程序执行的上下文，配合函数的嵌套调用。VirtualMachine 中还有一个 pc 指针（图中未标出），指向需要执行的指令，不同指令的运行会以不同的方式改变 pc 的指向，从而模拟出真实计算机的运行效果。

- **Parser:** Steed 的 Parser 模块与传统编译器中的语法分析器有所不同。主要功能是识别出每行 KnightAssembly 的操作码和操作数，通过不同的操作码对应的 parse 函数，分析出操作数，并将其组装成 Instruction，存储至虚拟机的函数之中。

- **Instruction:** 每种 Instruction 都是一个子类，都在 Instruction 模块中实现了自己的 Excute 方法。Function 存储着 Instruction 的指针数组，利用 C++ 的多态机制，调用 Instruction 指针指向的 Excute 方法即可完成指令的执行，每条指令的执行会改变虚拟机的状态。

- **FileManager:** 文件管理器，对 KnightAssembly 输入文件的操作进行封装，便于对文件中指令的读取和分析。

- **Diagnositor:** 错误诊断器，对程序执行过程中出现的所有错误进行报告并终止程序运行。

### 3.3 示例

继续之前的例子 Pi，我们在 Steed 上运行经过 Trial 编译的 Pi.ks。

- 执行命令 `steed Pi.ks` 运行程序，运行结果如下图 22 所示。



```
blankheart0@blankheart0:~/Knight/test/unit/PaperExample$ steed Pi.ks
Normal Pi:3.14
Secret Pi:
```

图 22 Pi.ks 运行结果

可以看到，只输出了变量 `normal_pi` 的值，而变量 `secret_pi` 没有被输出，这样由于实际程序还没有被给予 SECRET 权限，赋值语句和输出语句被跳过。

## 4 Knight 语言演示系统 Duke

### 4.1 简介



图 23 Duke 系统

Duke 是用于演示 Knight 语言功能的演示系统，操作界面如图 23 所示。使用形式和操作系统的 Shell 相同，支持多种命令，重点实现了应用程序的安装、卸载、权限的修改，用来模拟 Knight 程序在真实操作系统权限机制下的运行效果，与用户实际的使用场景一致。

### 4.2 整体设计

Duke 系统的整体架构如图 24 所示。

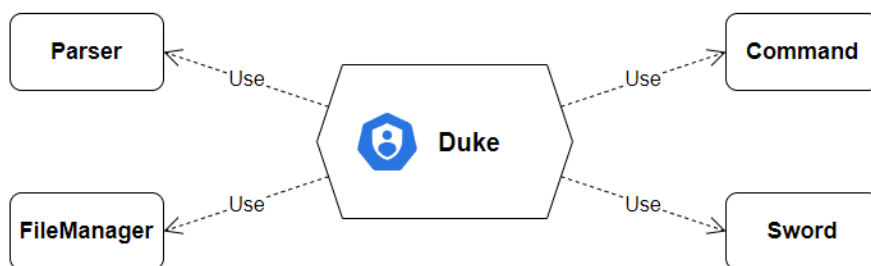


图 24 Duke 系统架构图

- **Parser:** Duke 中 Parser 的功能是识别和分析出用户输入的命令，通过不同的命令对应的 parse 函数，分析出命令的参数，并将其组装成 Command，在 Duke 中完成执行。
- **FileManager:** 文件管理器，对输入和输出文件的操作进行封装。在执行 install 指令安装应用程序或执行 chperm 指令改变应用程序权限时，便于对文件进行操作。
- **Command:** 每种 Command 都是一个子类，都在 Command 模块中实现了自己的 Excute 方法。与 Steed 中的 Instruction 指针一样，借助 C++ 的多态机制，调用 Command 指针指向的 Excute 方法即可完成用户命令的执行。
- **Sword:** 宝剑模块，提供一个飞剑动画，会在启动 duke 和使用 draw（拔剑）命令时播放。

### 4.3 支持命令

Duke 系统现阶段支持的所有命令如下表 2 所示：

表 2 Duke 系统支持命令

Command	Arg	功能
byebye		退出系统
command		显示所有支持的命令和用法
history		显示历史执行命令
echo	something	回声（显示 something）
again		再次执行上一条命令
clear		清屏
draw		拔剑（显示一个飞剑动画）
list	[-pkg -app]	查看当前系统中的文件
install	package	从安装包安装应用程序
uninstall	application	卸载应用程序
chperm	application	修改应用程序权限
run	application	执行应用程序

### 4.4 示例

这里我们继续使用程序 Pi，对 Pi.ks 进行安装和执行。

- 将之前 Trial 编译器编译得到的 Pi.ks 复制至 dukeEnv 文件夹，如图 25 所示。

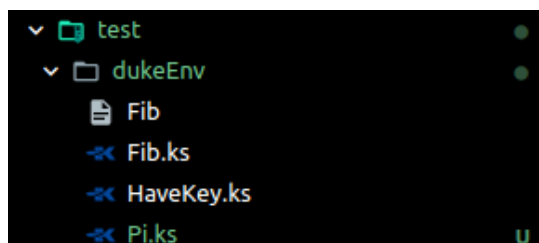


图 25 dukeEnv 文件夹

- 在该文件夹下启动 duke，执行命令 `list -pkg` 查看当前可安装的包，如图 26 所示。

```
Command >> list -pkg
Fib.ks
HaveKey.ks
Pi.ks
```

图 26 安装包

- 执行命令 `install Pi.ks`，给予应用程序权限 SECRET，如图 27 所示。

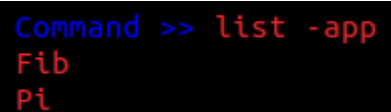
```
Command >> install Pi.ks
Installing ... ..

Require permission: SECRET, yes or no? (y/n)
y

Install Done :)
```

图 27 安装程序

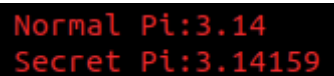
- 执行命令 `list -app` 查看当前已安装的应用程序，如图 28 所示。



```
Command >> list -app
Fib
Pi
```

图 28 已安装程序

- 执行命令 `run Pi` 运行该程序，可以看到 `normal_pi` 和 `secret_pi` 均被输出，如图 29 所示。



```
Normal Pi:3.14
Secret Pi:3.14159
```

图 29 运行程序