The General-Purpose Programming Language

Tripod

参考规范（草案）

戴巧归 著

序

在当今这个计算机行业蓬勃发展的阶段，涌现出了一代又一代经典的编程语言，这些语言毫无疑问在它所产生的那个时代和那个环境大获成功。但是，随着时代的进步，那些编程语言随着时代的进步褪下了光辉的面纱，转而留下了一个有一个难以解决的历史遗留问题。

为除去这些语言的弊病，诞生了大量的“现代化”的编程语言，这些编程语言虽然使得开发者的效率大大提高，但同时也在执行效率和底层安全性方面付出了惨痛的代价。并且语言的易理解程度也在逐渐下降，语言的语法方面也难以统一，造成了深入掌握的困难。这些弊病随着大量的使用而不断地呈现出来，而成为一门现代化编程语言的硬伤。

为了解决经典语言的历史遗留问题，同时也为了解决编程语言现代化上的种种困难，方便各行各业的人快速上手。我决定开发一门简明，优雅，强健，集合大量经典编程语言的精华，并将这些精华融而为一，同时又具有一个统一的语法结构的新式编程语言。而今，这门语言的原始草案已经成型，也基本上达成了最初的设计目标。

这门语言名叫 Tripod，中文译为鼎，取革故鼎新之意。Tripod 同时支持多种编程范式，其中包括有面向过程、面向对象、泛型和函数式。本草案描述了语言各个方面的特性，包括声明，类型，类，对象，函数和模块等等。

文后的附录 A 与 B 简单的描述了语言的词法和文法结构，这两篇是专供编译器设计者供参考的，它在某些方面仍还有些许不足，需要改良，这将在规范的以后版本中做出修正。

致谢

这门语言的灵感来源于 Basic，C\C++，Pascal，Scheme 等编程语言，在此首先向这些语言的设计者 JohnG.Kemeny、ThomasE.Kurtz、Dennis.MacAlistair.Ritchie、Bjarne.Stroustrup、Niklaus.Wirth、Harold.Abelson、Gerald.Jay.Sussman、Julie.Sussman 致以诚挚的谢意，没有他们，也就不会诞生这门语言。

本书大量参考了上述这些编程语言相关的设计文档，对于语言本身无需做出修改的地方，部分或者全部摘录自下列著作中的原文：

- BrianW.Kernighan, DennisM.Ritchie.著，徐宝文，李志译。C 程序设计语言(第 2 版)[M]. 2004.
- SamuelP.HarbisonIII, GuyL.Steele Jr.著，徐波，等译。C 语言参考手册[M]. 2011.
- Jeff.Marshall，《FreeBASIC programmer's guide》，https://www.freebasic.net/wiki/wikka.php?wakka=CatPgProgrammer（2016-12-5）
- Michaël.Van.Canneyt，《Free Pascal Reference guide》，https://www.freepascal.org/docs-html/current/ref/ref.html（September 2017）
- James.Gosling，Bill.Joy，Guy.Steele，Gilad.Bracha，Alex.Buckley，《The Java® Language Specification》https://docs.oracle.com/javase/specs/jls/se7/html/index.html（Java SE 7 Edition 2013-02-28）
- Harold.Abelson，Gerald.Jay.Sussman，Julie.Sussma 著，裘宗燕译。计算机程序的构造和解释[M]. 2004.

在此向这些书籍和文档的作者及译者致以诚挚的谢意，真诚的感谢每个人。

目 录

第 1 章　标　记

本章描述了构建语言源代码的基本词汇单位：词法元素；以及如何构成这种词法元素的方法。

基本的词法元素又被称为词法标记，它们是语言的“单词”：根据编程语言的规则将字符组合成标记。

1.1　字 符 集

字符集，又称为词汇表，是用来构建词法标记的基本单位。语言所用的字符集由 5 部分组成。

一、52 个大小写拉丁字母（下划线被视为字母）。

二、10 个数字

三、空白字符，包括空格，水平制表符（HT），垂直制表符（VT），回车符（CR），换行符（LF），和换页符（FF）等格式控制字符。

四、特殊字符，这些通常是用于算数运算或者某些特殊操作的字符。特殊字符共计 23 种，表 2-1 是对它们的详细描述。

五、中文编程所用字符集，由编译器设计者指定。

所有的空白字符用于在源代码中控制格式，分隔词法标记的作用，除此以外，它们不再具有其它任何特殊作用，都将被忽略。换行符表示一行的结束，为了使代码能够延续到下一行，可以使用单一的下划线（_）字符，下划线字符后只能出现换行符，不能出现其他标记。

语言中的符号分为两种：运算符、分隔符。

一、运算符用来进行表达式运算。

二、分隔符也即传统意义上的标点符号。

下表列出了 Tripod 中预定义的特殊字符，见表 1-1。

1-1 特殊字符

字符	名称	功能	字符	名称	功能	字符	名称	功能
(	左括号	…	**+**	加号	相加	**"**	双引号	字符串
)	右括号	…	**-**	减号	相减	**'**	单引号	字符
[	左中括号	…	*****	乘号	相乘	****	反斜杠	转义、续行
]	右中括号	…	**/**	除号	相除	**<**	小于号	小于
{	左花括号	….	**%**	百分号	取余	**>**	大于号	大于
}	右花括号	…	**&**	和号	位与			
:	冒号	…	**|**	竖杠	位或			
;	分号	…	**^**	折音符	异或			
.	点号	…	**,**	逗号	分隔			

1.2 注 释

注释属于源代码的一部分，在执行时它们会被执行器忽略。注释的存在仅仅用于帮助程序员更好的理解代码，对代码进行备注，标记。编译器会完全无视注释的存在而执行其余代码。

注释有两种书写形式。

一、单行注释。这种注释只能占用一行，随着一行的结束，注释的作用消失。所有从 // 符号开始的文本到行的末尾，所有的内容，包括符号自身，都将被忽略。

二、多行注释。这种注释允许一次跨越多行。所有从 /* 符号开始的文本到 */ 符号结束的内容，包括符号自身，都将被忽略。

注释具有如下的一些性质:

一、注释不能嵌套。

二、/* 和 */ 在以 // 开头的注释中没有特殊含义。

三、// 在以 /* 或 /** 开头的注释中没有特殊含义。

示例

下面的例子展示了使用不同注释的方法。

```
// This is a great programming language.
/* This is a great programming language. */
/**
    This is a great programming language.
    Its multi-paradigm system is very robust.
*/
```

1.3 标识符

标识符用于给常量、变量、函数、类型、标签、模块命名。标识符（或名称）由大小写拉丁字母、数字和下划线的字符序列组成。标识符的第一个字符必须是字母（下划线被视为字母）。

语言的标识符区分大小写，当两个标识符具有相同的拼写时，并且大小写也一致时，这两个标识符被认为是同一个标识符。其他情况下，标识符都是不同的。例如，MyProgram 和 myProgram 是不同的标识符。

标识符不能与关键字，布尔值，空值具有相同的拼写。

示例

下面的例子展示了标识符的一般形式。

```
MyTestProcedure
insertItem
receive_information
_123
liMingNum123
```

1.4 关键字

关键字是特殊的标识符，由语言预先定义，是语言的一部分。程序员无法重定义关键字标识符，也不能把它们当成普通的标识符使用。语言不区分关键字的大小写。例如 loop 和 LOOP 是同一个关键字。

下表列出了语言当前可用的 61 个关键字，见表 1-2。

表 1-2 关键字

Byte	UByte	Single	If	Virt	Else	byVal	class
DByte	UDByte	Double	For	Public	Object	subject	protect
QByte	UQByte	Real	Do	Private	Unifiy	byVar	entity
OByte	UOByte	And	Loop	Protect	Enum	static	private
Int	Const	Or	Pass	Type	variety	byConst	let
Void	Var	AndAlso	Break	Via	nil	goto	
Import	Val	OrElse	Return	ByteBool	null	byRef	
Exit	Boolean	WordBool	QWordBool	LongBool	ref	public	

1.5 字面量

字面量用于表示程序源代码中的一个实际值。字面量包括整数字面量，实数字面量，字符字面量、布尔字面量、nil 字面量、null 字面量和字符串字面量，其中字符字面量属于整数字面量。

1.5.1 整数字面量

十进制

十进制数字（0 1 2 3 4 5 6 7 8 9）。
注意：要获得负值，可以在数字文字前放置“-”符号（operator -（取负））

示例

```
int x = 123456
byte b = -128
```

十六进制

“0x”后跟十六进制数字（0 1 2 3 4 5 6 7 8 9 ABCDEF）。

示例

```
int x = 0x1E240
byte b = 0x80
```

八进制

“0”后跟八进制数字（0 1 2 3 4 5 6 7）。

示例

```
int x = 0361100
byte b = 0200
```

二进制

“0b”后跟八进制数字（0 1）。

示例

```
int x = 0b11110001001000000
byte b = 0b10000000
```

整数大小后缀

如果未给出整数文字后缀，则会自动计算保存文字所需的数字字段大小。指定大小后缀可确保编译器将数字视为特定的整数大小。数字文字前缀的负（-）和正（+）符号不是数字文字的一部分。它们被视为单目表达式。

前缀，后缀和十六进制字母数字都不区分大小写。

具体的整数后缀可参见后文类型章节。

1.5.2 实数字面量

浮点数以十进制数字指定，可以是正数或负数，具有小数部分，也可以携带指数。

示例

```
double x = 123.456
double y = -123E23
```

实数大小后缀

默认情况下，没有指数或后缀的浮点数被视为双精度浮点值。

具体的实数后缀可参见后文类型章节。

1.5.3 字符串字面量

字符串字面量由一对双引号以及双引号中间的字符序列构成。字符序列可以有 0 个或多个字符，这些字符可以用一般形式表示，也可以由转义字符序列表示。字符串字面量用于表示一个字符串。字符串的存储基于一个整数的数组，或者一个指向整数数组的引用，用来存储一个完整字符串的整数数组往往最后一个元素是 0，0 代表了字符串的末尾。字符串的每个字符属于整数类型，基于被赋值的整数类型调整每个字符所占用的字节数。

示例

```
"Hello World!"
```

```
"That's right!"
```

1.5.4 字符字面量

字符字面量由一对单引号以及单引号中间的一个字符构成。字符字面量用于表示一个字符或者一个转义序列。字符字面量属于整数类型，基于被赋值的整数类型调整每个字符所占用的字节数。

示例

```
'\a'
'A'
'\x78F'
```

1.5.5 布尔字面量

布尔字面量用于指定布尔类型的值。布尔类型的值有两种：真（true），假（false）。

1.5.6 nil 字面量和 null 字面量

nil 字面量指示了一个空的引用。null 字面量指示了一个空指针。

1.6 转义序列

转义序列允许在字符串中表示一些非图形字符和不易直接书写的字符文字以及字符串文字中的单引号和双引号。用来表示转义序列的转义形式共有两种种：字符转义码以及数值转义码。

字符转义码用于表示一些常用的特殊字符，如响铃，退格，换行，回车等。这些字符无法直接在字符串中书写。这类字符与各种平台的字符集无关，它的实现是通用的。

下标列出了具体的转义序列信息，见表 1-3。

表 2-3　字符转义码

转义码	转换	转义码	转换
a	警告（即响铃）	v	垂直制表符
b	退格	\	反斜杠
f	换页	'	单引号
n	换行	"	双引号
r	回车	?	问号
t	水平制表符		

数值转义码使用八进制或十六进制数值代表字符集中的一个字符，八进制表示形式只允许出现 3 个数字，十六进制形式允许任意数量的数字，所有出现在数值转义码中的数字都必须是无符号数。十六进制数值转义码总是以\x 开始。

第2章 术 语

本章将描述有关声明，定义以及与它们相关的术语，这些术语将在后文中多次出现，如有需要，请参阅本章。

术语章节只包含通用术语介绍，专用术语详见其他具体章节。

2.1 声 明

在语言中，声明就是预先为某个实体分配一个标识符（名称），并构造其数据结构，限定其类型。

这个名称就是该对象的标识符。声明时可以限定标识符的类型。除变量声明以外，其他声明不分配空间，不关联代码块，只预保留一个标识符的形式与名字。

2.1.1 作用域

作用域，作用域指的是该声明所产生的标识符在程序文本中可见的那部分区域。也即一个标识符在哪个作用域中有效，一旦离开这个作用域，与这个标识符相关的声明和定义就会失效，在作用域外部使用内部的标识符将会出错。如果在外部使用了这个标识符，而没有出错，那么使用的必然是外部作用域中同名的变标识符，而不是内部作用域中的标识符。

在函数定义，类定义，代码块中声明的标识符具有局部作用域，离开代码块后这个标识符便会失效。在模块顶层声明的标识符具有模块作用域，在整个模块中都可用，模块结束后，标识符失效。

2.1.2 作用域树

在语言中，允许类，函数、对象以及标签代码块之间的相互嵌套，这种嵌套的深度没有数量限制，由于这种相互嵌套的关系，导致类，函数，代码块中出现大量重叠的作用域，这样的一整个作用域以及它们的嵌套关系图称为类、函数、标签的作用域树。

作用域树中，每个类，函数，代码块的名字同时也被称为作用域名，作用域树中不可存在同名的作用域。

2.1.3 可见性

在一个作用域中，可能存在统一标识符的不同定义（由于导入，作用域嵌套所产生），这时，新声明的标识符往往会隐藏曾经声明标识符，使得这些标识符在这个作用域中被隐藏，不能直接使用。

一个在作用域中未被隐藏的标识符称为可见标识符，在作用域中被隐藏的标识符称为不可见标识符。声明如果没有由于导入和嵌套的原因被其他声明所覆盖，那么这个声明的标识符总是在自己的作用域内是可见的。

标识符何时由于覆盖而被隐藏，取决于新声明标识符在作用域中的声明位置。

2.1.4 重复声明

在同一作用域中，不能出现同名变量的形式不同的声明，这种形式是非法的。这种形式被认为产生了“冲突”。

2.1.5 预应用

预应用是指在对一个标识符的引用出现标识符的完整定义之前，将标识符完整的定义后置，使得定义之间可以递归调用或者前向调用。

2.2 定 义

定义是指在声明之后，为具体的变量关联代码块或初始化。

简单来说，定义就是伴随着初始化的声明。能够定义的包括有各种类,.对象，函数等等。定义对于函数和类而言，好比创建了一个模具，这个可以根据这个模具来定义一些新的对象或者函数。

2.2.1 对象

对象指的是在计算机中存在一块内存区域，我们可以通过声明后的变量来访问，修改，存取这块区域中的数据。广义上来说，就是存在这么一个东西，程序员可以通过某种方式来操作它，使它完成某些相应的功能，提供所需的数据，执行必要的变更。

变量在声明时可能会创建一个对象，可以通过变量名使用这个对象。

2.2.2 范围

范围是指对象有效的范围，在什么样的范围内对象有效，什么样的范围内对象失效。

范围也称为对象的存储持久性。一个在代码块内创建的对象具有局部范围，这个对象在离开这个代码块后便会失效；一个在模块顶层创建的对象具有模块范围，这个对象在离开这个模块后便会失效；一个动态创建的对象具有动态范围，这个对象除非被手动释放或者由于不再使用而被回收否则不会被销毁。

2.2.3 初始化

初始化是指在声明的同时，对标识符进行赋值或者关联代码块。

创建对象，使其从声明开始就具有默认状态。

对于一个局部范围变量来说，这个标识符所指代的对象在每次创建时都会进行初始化，如未显式指定初始化的形式，则这个对象的初始状态是未定义的；对于一个模块范围变量来说，模块的初始化由模块初始化表达式序列执行，当模块被导入时，或者直接执行时，都会自动执行初始化；对于一个利用动态创建表达式创建的动态范围对象来说，变量声明时即意味着初始化。

初始化的形式有两种：按值初始化和定义初始化。按值初始化用于变量的初始化，定义初始化用于函数、对象、类定义的初始化。

语法

```
DataType variable = initValue
type id : … ;
```

第3章 类 型

对象的类型决定该对象可取值的集合以及可以对该对象执行的操作。

所有的变量都具有一个类型。Tripod 语言融合了大量编程语言的关于类型方面的优良部分，同时它自身也对很多不便的类型系统做了改良。

用户可以声明自己的类型，语言也鼓励用户竟可能的利用语言提供的类型自定义类型。定义类型实际上是定义了一个标识符，可以在源代码中进一步声明变量时用于表示此自定义类型。定义类型发生在声明变量的初始化阶段。

有 12 种主要类型，见表 3-1。

表 3-1 类型及其分类

类型	类型分类		
byte、dbyte、qbyte、obyte（有符号）、int	整数类型	算术类型	简单类型
ubyte、udbyte、uqbyte、uobyte（无符号）			
Boolean、ByteBool、WordBool、LongBool、QWordBool			
enum T			
T*	指针类型		
single、double、real	实数类型		
T[…][]…	数组类型		结构化类型
virt、type T	类类型		
T(…)	函数类型		
unify T	联合类型		
<…>	泛型类型		
variety v	变体类型		
void	void 类型		
var、const、val T	引用类型		
= type	类型定义		

本章将讨论 Tripod 的所有类型。对于每种类型，我们将讨论如何声明这种类型的对象、这种类型的值的范围、这种类型的长度或表示形式方面的任何限制以及在这种类型上所定义的操作。

3.1 整数类型

Tripod 利用 Pascal 扩展了 C 语言的整数类型，并为每个类型指明了具体的内存空间大小，他提供了比其他语言更多的整数类型和操作符。这些变型反映了在绝大多数计算机上所存在的不同字长和各种类型的算术运算符，使得 Tripod 具有跟 C 一样的底层硬件紧密性。Tripod 的整数类型用于表示下面这些数据:

一、有符号或无符号整数值，为它们提供了普通的算术运算。

二、位向量，对它们的运算包括“非”、“与”、“或”以及左移位和右移位运算。

三、布尔值，0 被认为是“假”，所有非零值被认为是“真”，其中整数 1 是最规范的“真”值。布尔类型变量只支持逻辑运算，不支持算术运算，可以将布尔类型转换为整数进行算术运算。字符，它在计算机上用整数编码表示，单个字符占用多大的整数类型，取决于被赋值的变量的整数类型。

四、枚举值，每个枚举值用整数编码表示。

整数类型可以方便的分为四种：符号类型、无符号类型、布尔类型和枚举类型。对于每种类型，都可以使用一组类型限定符声明这种类型的对象。

3.1.1 有符号整数类型

Tripod 向用户提供了四种标准的有符号整数类型，分别用类型限定符 byte、dbyte、qbyte 和 obyte 表示，另外，int 类型是平台通用整数类型，具体大小和范围取决于编译器的编译设置。

下表列出了 Tripod 中预定义的有符号整数类型及其范围和大小，见表 3-2。

表 3-2 有符号整数类型及其分类

类型	位数	字节数	最小值	最大值	后缀
byte	8	1	-128	+127	
dbyte	16	2	-32768	+32767	
qbyte	32	4	-2147483648	+2147483647	l_{opt}
obyte	64	8	-9223372036854775808	+9223372036854775807	ll_{opt}
int	32 or 64	4 or 8	…	…	i_{opt}

int 是 Tripod 用于整数数学和按位运算的主要数据类型。它是数字字面量的默认类型。int 型在声明时，如不指明具体的数据类型大小，它可以是 32 位或 64 位有符号整数数据类型，具体取决于目标平台。如果给出显式位大小，则它可以存储的值可以从-1ll << (bits-1)到(1ll << (bits-1))-1。可选的类型有 byte，等价于 int<8>、dbyte，等价于 int<16>、qbyte，等价于 int<32>、obyte，等价于 int<64>。

语法

byte *variable*
dbyte *variable*
qbyte *variable*
obyte *variable*
int *variable*
int *variable<bits>*

参数

bits
一个数字常量表达式，指示所需整数的位大小。所允许的值是 8，16，32 或 64。

示例

下面是一些典型的有符号整数变量声明的例子。

```
int x = 0x8000000000000000
int y = 0x7FFFFFFFFFFFFFFF
```

```
int<8> x_8 = 0x80
int<8> y_8 = 0x7F
```

3.1.2 无符号整数类型

对于每一种有符号整数类型，都存在着一种无符号类型，它占据相同数量的存储空间，但采用了不同的整数编码方式。为了声明无符号类型的整数，可以在对应的有符号类型限定符关键字前面添加前缀 u 构成无符号整数类型限定符。

下表列出了 Tripod 中预定义的无符号整数类型及其范围和大小，见表 3-3。

表 3-3 无符号整数类型及其分类

类型	位数	字节数	最小值	最大值	后缀
ubyte	8	1	0	+255	
udbyte	16	2	0	+65535	
uqbyte	32	4	0	+4294967295	ul_{opt}
uobyte	64	8	0	+18446744073709551615	ull_{opt}

字符类型也是一种整数类型。也就是说，这种类型的值都是整数，可以在整型表达式中使用。

字符类型依赖于具体的整数类型，如果将一个字符赋给一个 ubyte 型变量，那它就是占有一个字节的无符号字符，如果将一个字符赋给一个 udbyte 型变量，那它就是占有两个字节的无符号字符。余下的字符类型依次类推。

在语言中，以 0 位结尾字符的字符数组的概念就是“字符串”。

语法

```
ubyte  variable
udbyte variable
uqbyte variable
uobyte variable
```

示例

下面是一些典型的无符号整数变量声明的例子:

```
ubyte  ubytevar  = 0xFF
udbyte udbytevar = 0xFFFF
uqbyte uqbytevar = 0xFFFFFF
uobyte uobytevar = 0xFFFFFFFF
```

3.1.3 布尔类型

布尔类型按所占字节大小分类,用于表示逻辑运算中的真假。布尔类型的预定义可能值只有 true 和 fasle，这些是可以分配给布尔类型的唯一两个值。当然，任何解析为布尔值的表达式也可以指定为布尔类型。这两个值属于关键字，不可被重新定义。任何返回布尔类型结果的表达式都可以将值赋值给布尔类型对象。

下表列出了 Tripod 中预定义的布尔类型及其范围和大小，见表 3-4。

表 3-4　布尔类型及其分类

类型	位数	字节数	真值	假值	后缀
ByteBool	8	1	true (non 0)	false (0)	…
WordBool	16	2	true (non 0)	false (0)	…
LongBool	32	4	true (non 0)	false (0)	…
QWordBool	64	8	true (non 0)	false (0)	…
Boolean	32 or 64	4 or 8	true (non 0)	false (0)	…

在 Tripod 中，所有的整数都可以用来表示布尔值。0 表示“假”，所有非零值都表示“真”。布尔表达式如果是假，则它的求值结果是 0。如果布尔表达式为真，则它的求值结果为 1。对布尔类型执行向整数类型的强制类型转换，总是能得到布尔类型内部用来代表真假的整数值。

对整数类型执行向布尔类型的强制类型转换，非零为 true，零为 false。布尔类型总是映射为对应的有符号整数类型。

语法

```
ByteBool  variable
WordBool  variable
LongBool  variable
QWordBool variable
Boolean   variable
```

示例

下面是一些典型的布尔类型变量声明的例子:

```
Boolean boolvar = true
boolvar = Boolean(1)
boolvar = Boolean(0)
```

3.1.4　枚举类型

枚举类型是由一种称为枚举常量的标识符所表示的一组整数数值。枚举常量是在定义枚举类型时指定的，它的类型为 int。每个枚举类型都是由一种因编译器而异的整数类型表示的，并且与这种类型兼容。

不过枚举类型与布尔类型一样，都是单独的类型，不能与整数类型混用。

使用指定的枚举类型时，分配的元素必须在枚举标识符列表中按升序排列。

语法

```
Enum EnumType [= (SymbolName [= Expression] [, ...])]
```

参数

EnumType

枚举类型的名字。

SymbolName

枚举常量的名字。

Expression

一个常量表达式用于给某个枚举常量指定值。

示例

下面这个声明：

```
Enum   fish = (trout, carp, halibut)
fish my_fish, your_fish
```

创建了一种枚举类型 fish，它的值包括 trout、carp 和 halibut。它还声明了两个这种枚举类型的变量 my_fish 和 your_fish，它们可以使用下面的语句进行赋值：

```
my_fish = halibut
your_fsih = trout
```

枚举将直接将其常量添加到父命名空间，而不是单独充当命名空间，这种行为类似于 C 语言，因此，枚举常量可能与父命名空间中的其他符号产生冲突。一个枚举类型变量可以作为一个用户定义的类型将传递给可以操作这种类型的运算符或函数。

3.2 指针类型

对于任何类型 T，都可以形成指针类型“指向 T 的指针”。取决于 T 是对象还是函数，指针类型又可以分为对象指针或函数指针。指针类型的值是 T 类型的对象或函数的地址。

指针是变量，它的值是内存中的地址，并且它们被称为“指向”该存储器。指针的声明方式与任何其他变量一样，在类型名称后面加上后缀*用于声明指向某个类型的指针。

语法

```
DataType * variable
```

示例

```
int * ip          // 指向 int 类型对象的指针
byte * bp         // 指向 byte 类型对象的指针
int () * fp       // 指向返回整型对象的函数的指针
```

指针有两个最常用的操作符。一个是取地址运算符@，用于创建指针值；另一个是间接访问运算符*，它对指针变量进行求值，以访问它所指向的对象。

示例

```
int i, j
int * ip
ip = @i
i = 22
j = ip *          // j 现在具有值 22
```

```
ip * = 17          // i 现在具有值 17
```

在上面这个例子中，指针 ip 被赋值为变量 i 的地址（@i）。在这个赋值之后，表达式 ip*所表示的对象与 i 相同。

指针类型的其它操作包括赋值、减法、关系和相等测试、与整数的加法和减法以及整数类型和指针类型之间的相互转换。

指针类型的长度因编译器而异。在有些情况下，指针的长度不一定和任何类型的整数长度存在任何关系，但一般可认为 int 类型至少与任何指针类型一样长。

3.2.1 通用指针

在一般编程中，偶尔需要一种通用的数据指针，可以把它转换为任何对象指针类型。在进行指针求值之前把通用指针转换为适当的类型。

为了声明通用指针，需要使用 void *类型，这个类型代表通用指针类型。通用指针无法用*或下标运算符进行求值，也不能作为加法或减法运算符的操作数。任何指向对象或不完整类型的指针（不指向函数类型）都可以转换为 void *类型，然后再转换回来，而不会发生任何改变。void *类型既不是对象指针，也不是函数指针。

示例

下面是一些指针声明和转换的例子：

```
void   * generic_ptr
int    * int_ptr
byte   * byte_ptr
generic_ptr = int_ptr
int_ptr = generic_ptr
int_ptr = byte_ptr
int_ptr = (int *) byte_ptr
```

通用指针在使用函数原型方面提供了额外的灵活性。当一个函数具有一个可以接受任何类型数据指针的形式参数时，这个形式参数就应该被声明为 void *类型。如果这个形式参数被声明为任何其他指针类型，对应的实际参数必须是相同的类型，因为不同的指针类型是不兼容的。

示例

strcpy 工具函数用于复制字符串，因此需要 byte *类型的实参：

```
byte * strcpy(byte *s1, const byte * s2)
```

但是，memcpy 函数可以接受任何类型的指针，所以使用 void *：

```
void * memcpy(void * s1, const void * s2, size_t n)
```

3.2.2 null 指针和非法指针

Tripod 的每种指针类型都有一个特殊的值，称为 null 指针，它与这种类型的每个合法指针都不相同，它等于 null 指针常量，并且可以转换为其他指针类型的 null 指针。当它用于布尔上下文环境时，它的值是

“假”（不需经过显式类型转换）。Tripod 的 null 指针常量是值为 0 的任何整数表达式（或者转换为 void *之后）。

通常，所有 null 指针的表示形式是所有的位为 0，但这并非强制要求。事实上，不同的指针类型可能采用不同的 null 指针表示形式。如果 null 指针没有被表示为 0，编译器设计者必须花点心思，确保在不同类型的 null 指针和 null 指针常量之间进行正确的转换。

示例

下面这条语句：

```
if ip: i = ip * ;
```

是下面这条语句的常用简便记法：

```
if ip <> null: i = ip * ;
```

作为一种良好的编程风格，当一个指针不再指向一个合法的对象或者函数时，最好把它的值显式置为 null。

用户也可能因为不小心而创建了非法指针，即指针的值不是 null，但并没有指向适当的对象或函数。在声明指针变量之后，如果没有把它初始化为某个合法的指针或 null，常常会导致非法指针。对非法指针的任何使用，包括把它与 null 值进行比较、把它作为实参传递给函数，把它的值赋值给另一个指针、其结果在 Tripod 中是未定义的。

其他可能出现非法指针的情况包括把任何整数值强制转换为指针类型、销毁指针所指向对象的存储空间，或者使用指针运算导致指针所指向的元素超出了数组的边界。试图对非法指针进行求值将会导致运行时错误。

在使用指针运算时，Tripod 要求数组最后一个元素之后的那个位置仍然是一个可定义的地址，尽管对这种地址进行求值可能是非法的。这样，用户可以很方便地使用指针表达式对数组进行遍历。

示例

下面这个循环使用了数组尾部之后的地址，不过它并没有对这个地址进行求值：

```
int Array[N]
int * p = @Array[0]
…
loop if p < @Array[0] :
    p++ … ;
```

这个要求可能会限制编译器在一些采用非连续地址体系结构的目标计算机上的实现，只能把数组的最大长度减少 1 个对象。在这种计算机上，对位于非连续内存位置中的指针进行运算是不可能的。为了执行指针运算，程序员在分配内存时必须保证它是连续的。

3.2.3 null 指针和非法指针

用户不应该假设所有的指针类型（实际上就是所有的地址）具有统一的表示形式。在常见的多字节编址的计算机上，所有的指针都采用简单的字节地址，占据一定的空间（例如 1 个字）。在这些计算机上，指针和整数类型的转换在表示形式上不要修改，并且不会丢失信息。

但是，Tripod 并不要求编译器总是保持这种良好的行为。所以，在指针类型之间进行转换时，用户总是应该使用显式的强制类型转换，并特别需要注意传递给函数的指针实参的类型与函数所期望的类型兼容。void *可以作为通用的对象指针。但是，并不存在通用的函数指针。

3.3 实数类型

构成 Tripod 实数类型的数据类型具有 2 种长度，即单精度和双精度，或者成为 Single 和 Double，以及基于平台可变的 Real 类型。

Real 类型取决于平台，它可以是 Single 或 Double 中的任意一种。所有的实数类型应当遵循有关的 IEEE 标准。单精度实数类型一般用以容纳比较小的实数，单精度实数类型总是能够保持 6 位小数精度。双精度实数类型一般用以容纳比较大的实数，双精度实数类型总是能够保持 15 位小数精度。

real 型应当总是保持与各个平台相兼容，为了使程序尽量能够移植，请尽量使用 real 型作为标准实数类型使用。

语法

```
single    variable
double    variable
real      variable
```

示例

下面是一些实数类型对象的典型声明：

```
double A = 1.985766472453666
single B = 1.9857665
real   C = 3.1415926535247
```

下表列出了 Tripod 中预定义的实数类型及其范围和大小，见表 3-5。

表 3-5 实数类型及其分类

类型	位数	字节数	最小值	最大值	后缀
single	32	4	+/-1.401 298 E-45	+/-3.402 823 E+38	S_{opt}
double	64	8	+/-4.940 656 458 412 465 E-324	+/-1.797 693 134 862 316 E+308	D_{opt}
real	32 or 64	4 or 8	…	…	R_{opt}

3.4 结构化类型

结构化类型是一种可以在一个变量中保存多个值的类型。结构化类型可以嵌套到无限级别。由于结构化类型作用和功能繁多，一章难以详述，故将在后面的章节中讨论每种可能的结构化类型。

3.5 变体类型

variety 数据类型是所有没被显式声明为其他类型变量的数据类型。variety 数据类型并没有类型声明字符。存储在变体类型变量中的值的类型仅在运行时确定：它取决于已分配给变量的内容。几乎任何简单类型都可以分配给变体，算术类型，字符串类型、等等。

结构化类型（如联合，数组，对象和类）与变量以及指针不是分配兼容的。

语法

```
variety variable
```

示例

这意味着以下的声明是有效的：

```
Enum TMyEnum = (One, Two, Three)
variety v
int i
byte b
word w
obyte o
double d
TMyEnum En

v = i
v = b
v = w
v = o
v = d
v = En
```

3.5.1 赋值与表达式中的变体

从上面的定义可以看出，大多数简单类型可以分配给变体。同样，可以将变量分配给简单类型，如果可能，变量的值将转换为分配给的类型，从变体类型转换为普通类型时必须使用强制类型转换，以确保这是用户所想要的行为。

示例

```
variety v
int i

v = "100"
i = int(v)
v = "Something else."
i = int(v)
```

变体保存字符串字面量的方式是保存指向字符串内存位置的地址。所以可以将字符串赋值给变体，但必须通过强制类型转换才能得到内存位置的值。

变体类型由编译器动态缩放变体变量的空间，这可能会影响到设计变体表达式进行计算与求值时的性能，因此应当谨慎使用。如果需要进行大量计算，最好避免使用变体。

3.5.2 变体与函数

变体类型不属于聚合类型，但是由于它的数据长度是会动态改变的，所以变体在作为函数的参数或返回值时，可以使用跟聚合类型一样的处理方式，利用引用或者指针作为参数或返回值进行变体值的传递。

示例

```
variety v, k
int i

variety * vi(variety * v):
    variety y + = v
    i = int(v)
    y = i
    return @y
    ;
```

3.6 void 类型

void 类型不包含任何值或操作。

void 类型用于
一、作为函数的返回类型，表示这个函数不返回任何值，类似于其他编程语言中的过程。
二、用于类型转换表达式中，显式表示丢弃一个值。
三、形成 void *类型，它是一种“通用的”数据指针。
四、在函数声明器中代替形参列表，表示这个函数不接受任何参数。

示例

write_line 的声明使用了 void，它既作为返回类型，又做为形参列表。

```
void write_line(void)
…
write_line()
```

write_line 的声明表示这个函数返回一个值，但这个函数调用把它转换为 void，显式地丢弃了这个返回值。

```
int write_line2(void)
…
void(write_line2())
```

3.7 引用类型

引用类型用来指定一个对象的别名。

引用类型变量总是指向一个对象，这个对象可以为空。引用类型变量总是起到作为对象的别名作用，一旦引用变量引用了一个对象，之后便可以通过变量访问这个对象。

引用类型通过使用 var、val 或 const 引用限定符进行声明，其后为可选的类型限定符，然后是定义表达式。Tripod 的引用是透明的，它是某个对象的简单别名，在使用引用变量时，就相当于使用引用的对象。引用变量一旦引用了一个对象后，之后如果需要引用其他对象，则必须通过强制取引用表达式得到对象的引用。

语法

```
var    [DataType] variable
const  [DataType] variable
val    [DataType] variable
```

示例

```
int i = 0, k = 1
var int vi = i
vi = var(k)
```

3.7.1 引用对象模型

对象既不是可更改的，也不是不可更改的，对象就在对象，本身不具有任何多余的的属性。var、val 与 const 关键字并不作用于对象本身，它们仅仅作用于定义的变量。引用变量声明时并不创建具体的对象，它仅仅预先定义了可以保管这种类型对象的一个名称。

引用类型变量的操作与它引用的类型的对象操作一致，对引用类型变量的操作就是对其所引用对象的操作。

示例

```
var int i
int k = 0
i = k
i = 1
```

首先定义了一个整型引用变量 i，它是一个整型变量的别名，由于 i 尚未引用任何类型的对象，所以对其直接进行赋值，可以使其引用变量 k，之后对 i 赋值将会改变 k 的值。

3.7.2 值引用

利用 val 关键字可以在定义变量时创建值引用变量，值引用变量所指向的对象在被调用之前就必须确定，值引用变量指向对象的内容是无法更改的，并且值引用变量在初始化后不可更改所引用的值。使用值引用变量时，就是直接使用变量所引用对象的值。

值引用类似于其他语言中的常量。

示例

```
val VTAB = '\013'
val BELL = '\007'
val str = "I'm a string"
```

3.7.3 常引用

利用 const 关键字可以在定义变量时创建常引用变量，常引用变量所指向的对象通过引用变量是无法更改它所引用对象的任何内容的，并且常引用变量在初始化引用一个对象后，这个变量不可以更改对所引用对象的引用。

被 const 限定的变量所指向的对象，对象的内容虽然不能通过这个变量进行更改，但允许从这个变量取值，或进一步计算。const 变量所引用的对象可以通过赋值传递给另外的 const 变量和 val 变量，但不能传递给 var 变量。

const 变量所引用的对象可以传递给 const，也可以在运行时传递给尚未引用任何值的 val 引用变量，被 val 所引用后的对象无法再以任何方式修改，这称之为对象的固定，之后通过其他类型的引用都无法修改该对象。

示例

```
type Data:
    int A, B, C
    ;
Data x_Data
x_Data.A = 0
x_Data.B = 1
x_Data.C = 2
const Data cx = x_Data
~~vx.A = 3~~                        // 错误，不可通过常引用变量修改对象
```

3.7.4 变引用

利用 var 关键字可以在定义变量时创建变引用变量，变引用变量所指向的对象通过创建的变量可以通过更改它来更改具体所引用的对象。变引用变量在初始化引用一个对象后，这个变量可以更改对对象的引用。

var 变量所引用的对象可以传递给 const，也可以在运行时传递给尚未引用任何值的 val 引用变量，被 val 所引用后的对象无法再以任何方式修改，这称之为对象的固定，之后通过其他类型的引用都无法修改该对象。

从 var 变量引用的对象传递给 const 变量是可行的。另外，这个过程是不可逆的，也即 const 变量所引用的对象无法再被 var 变量所引用。

如果同时有 var 变量和 const 变量引用了同一个对象，例如先创建了 var 变量，后将其引用传给了 const 变量，那么仍然 var 变量可以修改对象，const 标识符不可以修改对象。

如果一开始就用 const 定义的变量引用对象，则之后不能通过这个变量将对象传给 var 变量。

示例

```
type Data:
    int A, B, C
    ;
Data x_Data
x_Data.A = 0
x_Data.B = 1
x_Data.C = 2

var Data vx = x_Data
vx.A = 1                    // ok!
const Data cx = vx          // ok!
~~cx.A = 2~~                    // invalid!
vx.B = 3                    // ok!
~~var Data vvx = cx~~           // invalid!
val vlx = vx                // ok!
~~vx.A = 7~~                    // invalid!
```

3.7.5 通用引用

在一般编程中，偶尔需要一种通用的引用变量，它可以引用任何类型的对象。定义通用引用的方式是使用引用限定符不加类型限定符的版本声明引用变量。声明时具有类型限定符的引用称为具型引用，没有类型限定符的引用称为无型引用，也称通用引用，通用引用向具型引用转换时需要经过显式的类型转换，而具型引用转换为通用引用则不需经过显式的类型转换。

通用引用在运行时可以动态的根据其类型进行解引用，也可以将其转换为具型类型引用进行解引用，通用引用在作为函数的可变参数以及返回类型时提供了额外的灵活性。当一个函数可以接受可变数量的参数时，所有可变参数都是作为一个通用引用的数组传递给函数。

示例

```
void Action(var int * count; var A ...):
    int i = -1
    count* = 0
    loop if A[i++] <> Nil:
        count++
        ;
    ;
```

上述的函数用于对传递进函数的参数数量进行计数，所有的可变参数都保存在 A 的数组中，通过对特定项的下标解引用来获取特定的参数项。

3.7.5 Nil 引用

每个引用变量都有一个特殊的值，nil。nil 值表示该引用不指向任何对象。当它用于布尔上下文环境时，它的值是“假”，在引用变量创建而未引用一个对象时，引用变量具有默认值 nil，使用 nil 值的引用变量将

会造成错误。引用变量与其他语言中的指针类似，在很多情况下引用可以和指针可以互换使用，使用引用变量更安全，也更方便。

作为一种良好的编程风格，当一个引用不再指向一个合法的对象时，最好可以把其设为 nil，不过，在执行器能够自动的将所有指向已销毁对象的引用变量置 nil 的情况下，这并不是一种必要的形式。

语言中还存在着默认引用的概念，默认引用使用关键字 ref 定义，ref 具体指代 var、const 还是 val 由具体的平台决定。当引用作为函数的返回类型时，应当使用加 by 前缀的引用名，如 byRef，byVal，byConst 和 byVal，以与引用变量相区分。

最后，严格的来说，引用并不是一种具体的数据类型，它只是对象或实体的一个别名，它并不占据内存空间，每当在引用被调用时，编译器应总是将引用的对象包装成合适的形式直接进行调用。

3.8 类型别名

类型别名是一种为类型赋予另一个名称的方法，但也可用于创建真正的新类型。

语法

```
Declaration = type
```

参数

Declaration
变量，函数、数组等等的声明。

示例

```
int Array[3] = type
Array x = (0, 1, 2)
int Action(void) = type
Action y:
    return 0
    ;
var Array z                 // ok!
var int vArray[3] = type    // invalid!
```

类型别名允许为复杂的类型提供容易记忆的简写形式，还能避免作用域中的名字冲突。

新定义的类型不能与其它类型限定符以及其他 type 定义的类型变量组合使用，type 定义后的类型名总是完整类型名，不需再次组合。

type 只允许定义类型限定符，引用类型所用的限定符是引用类型限定符，二者不能混用。所以 type 定义中不可以使用引用限定符，但是允许在使用 type 类型变量时，在其前加上引用限定符。

使用 type 进行类型变量定义会引入新的类型，新的类型从某种程度上来说可以作为源类型使用，也就是源类型的别名，支持源类型所支持的一切操作。同时，也可针对新类型定义一些新的操作，这些操作对源类型并不会产生影响。

为方便操作，语言鼓励用户定义自己的类型，纵使是简单类型，语言本身也建议能够对它们进行重定义，以使得程序代码具有更好的可理解性。

第4章 变 量

变量指定了具有特定类型空间大小的一个内存位置，变量是这样一个内存位置的代名词。变量必须在使用前显式声明。除非声明变量，否则不会分配内存。

变量在进行声明后，对变量的操作，就是对相应内存位置的操作，而内存中存储这个变量的位置则取决于变量声明的位置和方式。编译器应当透明地处理这些内存位置的分配，尽管这个位置可能会在声明中受到影响。

4.1 声明

变量必须在模块、函数、代码块中可以声明变量的位置声明变量。变量的声明通过变量的声明语句进行声明。声明的同时，可以指定变量的类型限制以及它的初始值。

变量在声明时，可以不指定它的具体类型，对于这种变量，它的类型由编译器运行时动态决定。对于一个未用类型限定符进行声明的变量，则必须在第一次使用时对它进行初始化。

语法

```
DataType variable
```

示例

以下是一些可用的变量声明：

```
int FirstNumber, SecondNumber
FirstNumber = 1
SecondNumber = 2
FirstString = "Hello, world!"
SecondString = "The beautiful world!"
```

4.2 作用域

作用域确定了变量和对象的可见性和访问规则。

变量的作用域是指它在一段程序上下文中的可见性。变量在声明范围之外是不可见的（无法访问）。声明变量的位置和方式决定了它的范围。变量遵循一般标识符的作用域规则。另外，变量在进入它的作用域后会进行初始化：

一、程序启动时，起始模块全局变量初始化一次。

二、模块导入时，被导入的模块中全局变量初始化一次。

三、每次进入函数或代码块时，初始化局部变量。

需要注意的是，局部作用域的非值变量与值变量初始化的行为不同。具有局部作用域的值类型变量的行为类似于全局初始化。

4.3 初始化

默认情况下，变量在声明后不会自动进行初始化。任何假设它们包含 0 或任何其他默认值都是错误的，它们大多都包含有垃圾数据。为了解决这个问题，必须在声明的同时对变量进行初始化。与正常变量的声明区别在于它们的声明包含初始值。

上述规则唯一的例外是引用类型，引用类型变量在声明时如不进行显式的初始化，则该引用变量具有默认的初始值 Nil。任何使用具有 Nil 值的引用都是一种错误。

示例

这个声明：

```
byte* string = "This is an initialized string."
```

变量的值将使用提供的值进行初始化。以下是一种更好的方法：

```
val byte* str = "This is an initialized string."
byte* string = str
```

数组在声明时也可以进行初始化，数组初始化只能初始化一阶数组，或者将高阶数组降阶所得的一阶数组。数组的初始化形式是用圆括号括起初始化元素，元素之间用逗号分隔。初始化元素的数量必须与类型声明中的元素数量完全相同。举个例子：

```
byte* [20] tt[3] = ("ikke", "gij", "hij")
obyte ti = (1, 2, 3)
```

备注

应该强调的是，初始化变量在进入作用域时会被初始化，这与程序启动时初始化的值类型变量不同。对于局部变量的初始化也是如此。无论何时调用函数，代码块，都会对局部变量进行初始化。在上一次调用函数，代码块时所发生的任何更改都将被撤销。

对于初始化指针类型时应当格外小心，不能使用指针更改值类型变量（内存只读区域）的内容。

```
val s = "My String."
byte* sp = s
s[0] = "His String."        // 这是一个错误，不可修改只读内存区域的值
```

第5章 数 组

数组是特殊类型的变量，它们充当许多值或元素的容器。数组可以存储除了 void、不完整类型以及函数类型以外的任何类型的元素，并且它的所有元素共享相同的类型。

数组存储的元素是声明时数组类型的一个值。数组的每个元素都有一个相应的位置，这是一个整数值，范围从数组的下限到其上限（包括）。这些位置用于使用 operator[]访问数组中的各个元素，operator[]获取一个位置并返回对该位置元素的引用。数组中的有效位置大于或等于其下限，并且小于或等于其上限。

语法

```
DataType Array[Item , ...][...]...
```

参数

Array
　　数组变量名。
Item
　　数组一个维度的长度。

示例

一个数组被声明为 int A[3]，它由元素 A[0]、A[1]、A[2]、A[3]组成。

在下面的代码中，声明了一个整型数组（ints）和一个指针数组（ptrs），ptrs 中的每个指针被设置为 ints 数组中对应整数的地址。

```
int ints[10], i = 0
int * ptrs[10]
loop if i < 10:
   ptrs[i] = @ints[i]
   i++
   ;
```

数组所占据的内存大小（使用 SizeOf 运算符多得到的结果）总是等于数组的元素数量乘以每个元素在内存中的大小。

5.1 数组和指针

在 Tripod 中，“T 的数组”和“指向 T 的指针”之间存在密切的关系。首先，当一个数组标识符出现在一个表达式中时，这个标识符的类型就从“T 的数组”转换为“指向 T 的指针”，并且这个标识符的值被转换为指向数组第 1 个元素的指针。这是寻常单目转换规则之一。这个转换规则的唯一例外是当数组标识符作为 SizeOf 或取地址（@）运算符的操作数时。此时，SizeOf 返回整个数组的长度，@返回指向数组的指针（而不是指向数组的第 1 个元素的指针的指针）。

示例

在下面的第 2 行中，a 的值被转换为指向数组第 1 个元素的指针：

```
int a[10]
int * ip = a
```

它和下面的写法完全相同：

```
ip = @a[0]
```

SizeOf(a)的值将是 SizeOf(int) * 11，而不是 SizeOf(int*)，也不是 SizeOf(int) * 10。

其次，数组的下标是根据指针运算定义的。也就是说，表达式 a[i,k]被定义为与((a) + (i*k) +k)*相同，其中 a 根据寻常单目规则转换为@a[0,0]。下标的这个定义意味着任何指针都可以像数组一样用下标进行访问。用户必须保证指针指向了正确的数组元素。

示例

如果 d 的类型是 double，dp 是指向 double 对象的指针，考虑下面这个表达式：

```
d = dp[4]
```

只有当 dp 当前指向一个 double 数组的一个元素时，并且 dp 所指向的元素后面至少还有 4 个元素时，这个定义才是合法的。

5.2 高维数组

高维数组被声明为“具有多个维度索引的数组”。例如，在下面这个声明中：

```
int matrix[4,4,4,4]
```

matrix 被声明为具有 4 个维度的数组，每个维度均有 5 个项。

高维数组 matrix 的元素是按照下面的方式存储的（按递增的地址）：

```
matrix[0,0,0,0] 、 matrix[0,0,0,1] 、 matrix[0,0,0,2] 、 matrix[0,0,0,3] 、
matrix[0,0,0,4] 、… 、
matrix[4,4,4,0] 、 matrix[4,4,4,1] 、 matrix[4,4,4,2] 、 matrix[4,4,4,3] 、
matrix[4,4,4,4]
```

高维数组 matrix 总共包括 5 的 4 次方个元素，且每个元素都是一个 int 类型的变量。中括号内部被逗号分隔的索引项的数目代表了数组的维度数目，每个维度上数的大小代表了这个维度的长度。高维数组的维数从右往左数起，最右边的那一维称为第 0 维，右起第二维称为第 1 维，以此类推。一个 N 维数组的最大维为 N-1 维。一个高维数组的项数是各个维度长度与维数的乘积。

高维数组可以用来构造高阶数组，高阶数组可以数组为单位对数组进行操作。

5.3 高阶数组

高阶数组被声明为“数组的数组”。高阶数组的每个项都是一个指针变量的引用。例如，在下面这个声明中：

```
int Array[4,4][4,4]
```

Array 是一个高阶数组，它有两个阶，最右边的是第 0 阶，左边的是第 1 阶，第一阶数组中存储了第 0 阶整型数组每个数组起始位置的指针。

改变高阶数组所保存的数组：

```
int mArray[4,4]
Array[4,4] = mArray
```

高阶数组中括号的数目代表了阶数，每个中括号内部的数列代表了这个阶的维数和维度长度。高阶数组的阶数从右往左数起，最右边的那一阶称为第 0 阶，右起第二阶称为第 1 阶，以此类推。一个 N 阶数组的最大阶为 N-1 阶。一个 N 阶数组的第 K 阶的每个项总是保存了一个这种形式的 K 阶数组的引用，N，K 总是大于 0。

高维数组是只具有一个阶的特殊高阶数组，以后所称的高阶数组均包括一般的高维数组。

5.4 静态数组

静态数组是在整个程序执行期间具有固定常量大小的数组，静态数组在声明时使用值引用变量或字面量进行声明。这可以允许更快的程序执行，因为已经分配了数组的内存，这与动态数组不同，其元素存储器直到运行时才被分配。

具有自动存储功能的静态数组，将它们的元素分配给程序堆栈，只有当数组在范围内时，指向这些元素的指针仍然有效。在程序执行期间，任何存储类的静态数组都不能调整大小，只有动态数组才能调整大小。静态数组也可以用作用户定义类型中的数据成员，在这种情况下，数组直接作为用户定义类型结构的一部分进行分配。

静态在声明时，声明所采用的索引必须在运行前就可以知道它们的值。尝试访问索引超出声明范围的元素将生成运行时错误（如果启用了范围检查）。

示例

一个静态数组的声明：

```
real Array[100]
```

用于访问数组元素的有效索引介于 0 和 100 之间，其中包含边界 0 和 100。如果数组类型自身就是一个数组，则可以将两个数组合并为一个高阶数组。以下声明：

```
real APoints[3][100]
```

5.5 动态数组

动态数组的声明形式与静态数组相似，唯一的区别是数组的长度是由一个非常量表达式指定的。当执行器遇到这样的声明时，将对长度表达式进行求值，并用它的结果（必须是个正整数）作为数组的长度，创建这个数组。

在创建之后，动态数组就不能再改变长度。数组中的元素只能在已分配的空间内访问。访问这个空间以外的元素将产生未定义的结果。

示例

在下面的代码片段中，a 和 b 都是可变长度的数组，指针 c 是一种指向某个类型的指针。

```
int a_size
…
void f(int b_size):
    int c_size = b_size + a_size
    int a[a_size++]
    int b[7][a_size]
    int c*[5][c_size]
    ;
…
```

5.6 数组边界

在分配了数组的存储空间之后，数组的长度必须是已知的。但是，由于编译器在正常情况下并不会检查下标是否位于被声明的数组的边界之内，因此在声明外部数组（在另一个模块中定义）或者声明作为函数形参的数组时，可以省略数组的维数。

示例

下面这个函数 sum 返回数组 a 中每个数组前 n 个元素的和。它并没有指定数组的边界。

```
int a[][]
…
int sum(int n):
    int i = 0, k = 0, s = 0
    loop if k < SizeOf(a):
        loop if i < n:
            s += a[k][i]
            i++
            ;
        k++
        ;
    return s
    ;
```

这个函数也可有如下的声明方式:

```
int sum(var int a[][], int n)
```

聚合类型作为函数参数传递时，必须传递引用或者指针，数组作为引用传递时，可以省略维度，但不可省略阶数。

可以直接应用于数组的运算符只有 SizeOf 运算符和取地址运算符@。当数组作为 SizeOf 的操作数时，它必须具有边界，其结果是这个数组当前阶所占据的存储单元的数量。对于包含 n 个元素的 T 型的数组，SizeOf 的结果总是等于 n*SizeOf(T)。@运算符的结果是个指向数组的指针。

第6章 联 合

联合被定义为包含了一些成员，但在任一时刻最多只能容纳它的一个成员的数据类型。

从概念上说，联合的所有成员在该联合的存储空间中是重合的。如果这个联合非常大，或者需要一个包含大量联合对象的数组，就可以显著地节省内存。

与类和枚举一样，每个联合类型定义引入了一个新的、与众不同的联合类型标识符。如果联合定义中存在联合标签，它就与这种新的联合类型相关联，并可以在后续的联合类型引用中使用。联合类型也允许使用前向声明和不完整定义，其规则和类类型相同。

联合的成员可以是不属于可变修改类型的任何对象。另外，联合不能包含自身的实例，不过可以包含指向自身的指针和引用。联合成员的命名遵循单一作用域中的命名规范，也就是说，一个联合内部各成员的名称必须各不相同，但它们可以与其他联合的成员同名，并可以与变量、函数和类型使用相同的名称。

语法

```
unify T:
    Declaration …
    ;
```

参数

T
: 新的联合类型名称。

Declaration
: 联合的成员，多个声明项。

示例

假设我们希望一个对象根据情况可以是一个整数或者实数。我们定义一个 datum 联合：

```
unify datum:
    int i
    double d
    ;
```

并且定义了一个该联合类型的变量：

```
datum u
```

为了在这个联合中存储一个整数，可以使用下面的写法：

```
u.i = 5
```

为了在这个联合中存储一个实数，可以使用下面的写法：

```
u.d = 88.9e4
```

只有当一个成员的类型与该联合的最后一次赋值的类型相同时，才能够引用这个成员。Tripod 并没有提

供方法查询联合最后所赋值的是哪个成员，程序员可以记住它或以显式地记录与联合相关的数据标签。数据标签是个与联合相关联的对象，提示当前在联合中所存储的是哪个成员的值。数据标签和联合可以包含在一个公共的结构中。

6.1 联合成员的布局

联合类型的每个成员所分配的存储空间是从该联合的起始地址开始的。一个联合在任一时刻只能包含它的 1 个成员的值。联合类型对象的起始地址总是满足它所包含的任何成员的存储对齐要求。

示例

如果已有下面的联合类型和对象定义：

```
unify U:
    …
    int c
    …
    ;
U object
U * p = @object
```

那么下面这两个表达式具有相同的内容：

```
U* & P.c == P
@(P.c) == int * (P)
```

而且，它们所包含的值总是相同的，不管成员 c 的类型是什么，也不管 c 之前或之后的其他成员类型是什么。

6.2 联合的大小

联合类型对象的长度就是表示它的最大成员的值所需要的存储空间，以及尾部的必要填充空间，以满足适当的边界对其要求。规则是联合将进行填充，直到它所占据的空间等于该类型的数组的一个元素所占的空间。记住，对于任何类型 T（包括联合），包含 n 个元素的 T 类型的数组的大小等于 T 的大小乘以 n。另一种说法是联合的终止地址和起始地址具有相同的对齐要求。也就是说，如果一个联合必须以偶数字节的地址开始，它也必须以偶数字节的地址结束。

注意，联合类型的对齐要求至少与它的对齐要求最严格的那个成员同样严格。

示例

在一台要求所有 double 类型的对象的地址为 8 的倍数的计算机上，下面这个联合的长度将是 16，多余的 6 个字节用来填充，最大的成员长度是 10：

```
unify U:
    double value
    byte name[10] ;
```

第7章 类

Tripod 支持面向对象程序设计，支持面向编程的最基本单位就是类。

类是可由用户创建的特殊变量类型。类实际上只是一个容器，其中包含一堆其他变量，如数组，但与数组不同，类可以包含不同的变量类型（而数组总是包含许多相同类型的变量）。事实上，类甚至可以在其中拥有其他函数或者内部的类!

语法

```
type myClass < myGenericType > = class | unify | virt:
    DeclarationList
    …
    Initializationlist
    …
    ;
```

或

```
class | unify | virt myClass < myGenericType > :
    DeclarationList
    …
    Initializationlist
    …
    ;
```

参数

myClass
: 新定义的类类型。

DeclarationList
: 类类型中的变量、函数、类的声明列表。

Initializationlist
: 类类型中相关成员的初始化。

myGenericType
: 泛型类型参数。

示例

```
type Point = class:
    int x, y
    void Point(int x, y):
        Point.x = x
        Point.y = y
        ;
```

总共有三种形式的类，分别为 unify、class 和 virt，联合也属于类的一种。class 类的所有成员默认具有 public 的访问权限，virt 类的所有成员默认不具有任何访问权限。

除 virt 类以外，其他两种类类型都可以进行实例化。

7.1 成员

存储在类内的不同变量、函数和类被称为“成员”，或更一般地称之为项。成员可以是几乎任何类型的变量，包括数字类型，字符串，指针，枚举甚至数组。在类中创建变量的方式与正常创建变量的方式一样。可以通过（.）运算符访问类成员。

示例

```
type myClass = class
    int someVar
    ;
myClass myObj
myObj.someVar = 23
```

定义了一个新的类型名称叫 myClass，有一个整型的数据成员 someVar，然后创建了一个这种类型的对象 myObj，之后通过点运算符再为对象中的这个成员 someVar 赋值。

7.2 权限对象

访问权限限制访问类成员的某些代码部分，Tripod 中通过权限对象来实现访问权限的功能。

类的所有成员（包括成员数据，过程，常量等）属于三种不同的分类之一，每种分类都有自己的规则，规定可以访问或引用代码的位置，这些规则称为访问权限。有 public，protect 和 private 三种访问权限，它们分别在块定义的专用定义块中定义。

访问权限对象可以进行前置声明。访问权限作用于以类视角所见的所有成员，而不是以 public、protect、private 对象视角所见的成员，但是在使用上，必须指明具体的权限对象。

语法

```
public myPubObject:
    …
    ;
protect myProObject: ;
private myPriObject: ;
```

参数

myPubObject、*myProObject*、*myPriObject*

具有 public 属性、protect 属性、private 属性成员的对象。

示例

```
type Point = class:
    private Data:
```

```
        int useCount = 0
        int x, y
        ;
    void move(int dx, dy):
        Data.x += dx
        Data.y += dy
        Data.useCount++
        ;
    ;
```

public 所属成员可以从任何地方被引用；例如，可以从成员程序或模块级代码或过程访问它们；protect 所属成员只能从声明它们的 class 的成员函数或派生的 class 的成员函数中访问，外部代码无法访问它们；private 所属成员只能从声明它们的类型或类的成员过程访问，外部代码或派生类型或类的成员过程无法访问它们。

这三种代码块所形成的程序实体就是一个可定义的具有访问属性的对象，它将类内部的成员按访问权限，所属进行分组，归类。在类的多继承中用以确保类成员的秩序。

权限对象也允许嵌套。

备注

与函数和指针类型一样,有时需要类的前向定义。类前向定义使用类的名称与类定义相关的关键字 Class，Class、Virt。如下例所示：

```
type TClassB = class
type TClassA = class:
    TClassB B
    ;
type TClassB = class:
    TClassA A
    ;
```

7.3 实例化

Tripod 不具有像其他编程语言用于类实例化的构造函数用于创建对象。Tripod 中，只要是一个能够显示定义的类，一定是可以进行实例化的。

语法

```
TClass myObj
TClass +
```

参数

TClass
: 待实例化的类类型。

myObj
: 实例化后的对象名。

+

动态实例化运算符。

示例

```
ClassType ClassVar                    // 静态创建形式
var ClassVar = ClassType +
var ClassType ClassVar +              // 上一种形式的简写形式
ClassType * ClassVar = ClassType +
ClassType * ClassVar +                // 实例化后赋值给指针
```

类的实例化有两种形式：一种是静态实例化，静态实例化的对象具有局部范围，实例化的语法与一般的变量声明一样；另一种是动态实例化，动态实例化后的对象具有动态作用域，动态实例化采用的是动态创建的语法。

类的动态实例化的语法不光可以用来实例化类，也可以用来实例化任何可以定义对象的类型。

备注

利用引用动态创建的引用对象和指向这种类型对象的指针的关系如下：

```
var ClassVar = ClassType +
ClassType * xClassVar = @ClassVar
ClassVar = *xClassVar
```

利用引用类型动态创建的对象使用形式更一般形式的变量没有区别，所以将引用对象传递给指针对象时需要进行取地址操作。

类的动态创建表达式的结果就是这样类的一个对象。如果类的动态创建失败，那么引用类型变量的值为 Nil，指针类型变量的值为 Null。

7.4 对象销毁

利用静态声明语句创建的具有局部作用域的对象不需要手动进行销毁，对于使用动态创建语句新建的对象必须使用动态销毁语句进行销毁类实例（在编译器不支持自动垃圾收集功能的前提下）。

语法

myObj -

参数

myObj
待销毁的对象。

-
动态销毁运算符。

示例

```
var TComponent A +

A.Name = "My Component."
A-
```

类实例的动态销毁不会删除或禁止对实例的引用，为了确保在销毁对象后不出现野指针或者错误引用，应当手动的置空引用或空指针。类的动态销毁的结果取决于最后是怎样的方式完成的类销毁，如果类实例经过手动销毁成功销毁，将会返回一个大于 0 的值，如果在手动销毁前已自动销毁，将返回 0 值，如果手动销毁失败，则返回一个小于 0 的值。

7.5 成员函数

类的成员函数具有类类型所具有的的完全的访问权限。

7.5.1 声明和定义

声明和定义成员函数。

成员函数的声明与普通的模块级函数一致。除了它们在 Type 中声明并在其外部定义，Tripod 同时允许在类内部进行函数声明在外部进行函数定义以及直接在类内部进行函数定义。

外部定义的成员函数，必须与类内部相应的成员函数相匹配。

示例

下述的示例函数 f 采用了内部定义的形式，函数 g 采用了外部定义的形式。

```
class foo:
    int i
    void f(int n):
        print(n)
        ;
    int g() ;
    ;

void foo.g():
    return 420
    ;
```

7.5.2 调用

调用成员函数。

成员过程就像成员数据一样被引用，也就是说，它们的名称前缀是对象实例的名称或成员访问运算符（.）。类的成员函数调用与对象也没有什么不同。

示例

以下是有效的成员函数调用：

```
var TAnObject AnObject +

AnObject.AAction
```

7.5.3 类名

所有的成员函数都具有一个额外的参数，这个参数并不经过显式的声明。调用它们时，使用实例的名称和解引用运算符（.）来的引用实例自身的一个成员，允许成员函数直接访问该实例。

编译器添加的附加参数就是类的名字，由于它是引用，因此对类名的任何修改实际上是对调用时传递给成员函数的实例的修改。使用类名就像任何其他变量一样，即可以将它传递给采用相同类型对象的函数，调用其他成员函数并使用（.）访问成员数据等。

然而，大多数时候，明确地使用它是不必要的；成员函数可以引用它们直接通过名称传递的实例的其他成员，而不必使用类名对其进行限定。除非当函数内部的参数或实际参数名隐藏了类的成员时，才需要使用类名限定符来直接得到类成员的引用。在这种情况下，类名是引用这些隐藏成员名称的唯一方法。

示例

以下示例使用类名来引用名称被函数参数和局部变量隐藏的成员数据：

```
type foo = class:
    void f(int i)
    void g()
    int i = 420
    ;

void foo.f(int i):
            // 函数参数 i 隐藏了成员变量 i，所以必须显示限定 i 的引用。
    print(foo.i)
    ;
void foo.g():
    int i
            // 函数局部变量 i 隐藏了成员变量 i，所以必须显示限定 i 的引用。
    print(foo.i)
    ;
```

7.5.4 访问权限

成员函数与一般的模块级函数不同，成员函数具有相关类所指定的完全的访问权限，它们可以具有 private，protect 以及 public 性质的访问权限。

7.6 继承

继承将一个或多个类型合成为一个更大的类型，新类型具有原类型的所有成员。

unify 类型与 class 是实类型，virt 是虚类型，含有虚成员的实类型是半虚类型，不含有虚成员的类型是纯实类型。纯实类型不能被继承，但可以实例化，虚类型和半虚类型虽然可以被继承，但是不能进行实例化。

继承通过在类中使用 via 语句实现，via 语句可以一次继承多个类型。继承语句在类型中放置的位置是任意的，只需使其保持在使用超类成员或者扩展超类之前即可。被继承的类称为基类或超类，继承后的类称为子类或派生类。如果不需对类进行扩展，可以直接在类关键字后加上继承的类型。

示例

```
type Point = virt:
    int x = 0, y = 0
    void move(int  dx, dy):
        x += dx
        y += dy
        ;
    ;

type SlowPoint = class(Point):
    int xLimit, yLimit
    via Point:
        void move(int dx, dy):
            Point.move(limit(dx, xLimit), limit(dy, yLimit))
        ;
    int limit(int d, limit):
        if d > limit :
            return limit ;
        Else if d < limit :
            return -limit ;
        Else Do: return d ;
        ;
    ;
```

SlowPoint 通过 via 语句继承了 Point 类，并重写了 Point 类中的 move 方法。该方法限制了点在每次调用方法时可以移动的距离。当为类的实例调用该方法时，将始终调用类中的重写定义，即使对象的引用是从基类型引用变量中获取的。

7.6.1 扩展

扩展是指子类为超类添加更多的成员，重载、重写超类型中的函数，并为子类与超类建立更多的联系。

扩展通过在类型中使用 via 语句实现，via 语句可以一次扩展一个或多个类型，但并不推荐同时扩展多个类型，这会使代码结构变得混乱。

扩展后子类中的超类超类部分权限对象与原来的超类一致，via 语句不会更改访问属性。via 语句所扩展的类必须具体到扩展位置与原位置一致。via 不可扩展 private 权限对象。

示例

```
type Point = class:
    public Action:
        int x = 0, y = 0
        void move(int dx, dy):
            x += dx
            y += dy
```

```
            ;
        ;
    ;

type SlowPoint = class:
    int xLimit, yLimit
    via Point.Action:
        int z
        void move(int dx, dy):
            Point.move(limit(dx, xLimit), limit(dy, yLimit))
        ;
    int limit(int d, limit):
        if d > limit :
            return limit
            ; Else if d < limit :
                        return -limit
                        ; Else return d
            ;
```

利用 via 语句扩展了 Point 中的 Action 对象，重写了它的 move 函数，并为 Point 类添加了整型成员变量 z。

7.6.2 派生

派生是指在子类继承父类后，子类为自身额外添加更多的成员。

随着继承、扩展和派生，子类所依赖的父类数量越来越庞大，整体结构类似于一棵正向生长的树，这称之为类的派生树，类的派生树遵循作用域树的规则。

7.7 虚类

虚类是一种特殊的类类型，虚类不能直接实例化，但可以作为其他类的基类。

虚类内部可以包含成员的完整定义也可以包含不完整定义，这些不完整定义转而由派生类进行定义，虚类可以继承虚类，以扩展虚类。凡继承虚类的实类型都必须实现虚类中所有未实现的虚类函数。

7.8 类函数

每个类类型中都存在着一个特殊的函数（包括虚函数），这个函数与类名同名，可以具有不同的返回类型，可以进行重载、重写，在类实例化后，可以通过实例名调用函数，这个函数被称为类的类函数。

每个类型都默认提供一个主调函数的实现，默认的类函数什么也不做，它不接受任何参数，返回类型为空，每当显式声明了一个类函数的时候，这个默认的类函数就会被替代，如果需要一个类似默认主调函数功

能的函数，就必须再进行显式的定义。类函数定义时可以省略类函数的返回值，这样定义的类函数返回这个类的一个对象，返回指针还是引用取决于被赋值的变量。

类的类函数位于类顶层，始终对外公开，无论它具有何种权限，都可以通过直接或间接实例化的对象进行访问。

直接实例化：指类可以直接通过声明或动态创建的方式进行实例化。

间接实例化：必须依附于可以实例化的类型，通过依附的类型进行实例化。

示例

下例使用类函数模拟了 scheme 语言中常用的 cons 函数，与 scheme 不同的是，cons 是一个类，它可以直接读取内部的变量，而不用利用函数进行间接求值。

```
type cons = class:
    var left, right
    cons(var l, r):
        var cons i +
        i.left = l _ i.right = r
        return i
        ;
    ;
```

7.9 嵌套定义

类定义中可以包含更多的内部类定义，内部类的使用方式与一般类无异，内部类可以具有自身的访问权限，具有自身的类函数，并且也可以被重载，重写。内部类的作用域规则遵循作用域的一般规则。

实类型如果有虚的内部类，就不可以进行实例化，但是可以被继承。

第8章 泛 型

泛型是用于生成其他类型的模板。这些可以是类，对象，接口甚至函数，数组，记录。它是一个来自C ++的概念，它深深地融入了语言。

创建和使用泛型是一个具有两阶段的过程。

一、每个泛型的具体定义都是一种新的类型：它是一个代码模板。

二、泛型类型特化：它定义了第二种类型，它是泛型类型的特定实现。

8.1 泛型定义

泛型附着于一般的类定义中，它包含额外的类型占位符列表。

语法

```
<T [via Supers][; …]>
```

参数

T
    类型占位符。

Supers
    额外的类型限定。

示例

以下是有效的泛型类定义：

```
type TList<_T> = class:
    via TObject
    int TCompareFunc(const _T Item1, Item2) = type
    _T data

    void Add(_T item):
        data = item
        ;
    void Sort(TCompareFunc compare):
        if compare(data, 20) <= 0 :
            halt(1)
            ;
        ;
    ;
```

对于类，对象和函数类型，泛型类型声明后面应该是类定义。泛型类型声明很像普通类型声明，除了存

在尚未知的类型。未知类型列在占位符列表中，在泛型特化之前它们是未知的。

这个声明和实现有一些值得注意的事情：

一、假设存在一个占位符_T。当泛型特化时，它将被类型标识符替换。标识符_T 不能用于除类型占位符之外的任何其他内容。这意味着以下内容无效：

```
void TList.Sort(TCompareFunc compare):
    int _t
    …
    ;
```

二、在泛型类的定义中，除了泛型类型可以是未知的以外，其他所有标识符的类型都必须是已知的，而不应该等到泛型类特化时才能确定。

8.2 泛型特化

一旦定义了泛型类型，它就可以用于生成其他类型。这就像定义新类型一样，模板占位符用实际的类型定义填充。

示例

这是一个非常简单的定义。鉴于上一节中的 TList 声明，以下是有效的类型定义：

```
TList<Pointer> TPointerList = type
TList<Integer> TIntegerList = type
```

除了泛型以外，特化中的所有其他类型必须是已知的。

```
type TMyFirstType<T1> = class
    via TMyObject
    ;
type TMySecondType<T2> = class
    via TMyOtherObject
    ;
```

接下来的这个泛型特化是非法的，因为 TMyFirstType 是泛型不是具体的类型：

```
TMySecondType<TMyFirstType> TMySpecialType = type
```

同时，下例是允许的：

```
TMyFirstType<Atype> TA = type
TMySecondType<TA> TB = type
```

因为 TA 在被 TB 定义时，已经特化完成，成为了一个完整类型。

8.3 泛型限制

模板列表可以具有类型的额外说明符，这对于对象类型特别有用，如果模板类型必须从某个类下降，那么可以在模板列表中指定。可以为类和虚类型限制指定多个类型标识符。如果指定了类，则用于模板的类型必须等于或派生自指示的类型。

示例

下例演示了泛型的限制，为了特化这个泛型类，必须使用派生或扩展了 TComponent 类型的类，或者 Tcomponent 类型自身。

```
import sysutils, classes
type A = class
type B = class
type TList<_T via A, int> = class:
    via TObject
    int TCompareFunc(const _T Item1, Item2) = type
    _T data

    void Add(_T item):
        data = item
        ;
    void Sort(TCompareFunc compare):
        if compare(data, 20) <= 0 :
            halt(1)
            ;
        ;
    ;
```

给与上述的定义，则有如下声明:

```
TList<A> TAList
```

下述的定义是不允许的，因为泛型定义不接受与 A、C 类型不相关的类型。

~~TList<B> TBList~~

8.4 泛型重载

Tripod 允许对泛型类进行重载。这意味着可以使用不同的模板类型列表声明相同的泛型类。

示例

以下声明是可能的:

```
type TTest<T> = class(TObject)
    T TObj
    ;
type TTest<T; S> = class(TObject)
    T TObj
    S SObj
    ;
```

泛型类型能否重载取决于两个方面：

一、泛型参数的数量是否相同，具有不同数量参数的泛型类型总是能够进行重载。

二、如果泛型参数数量相同，则看是否存在泛型类型限制，被限制的类型不能与其他泛型类型所限制的类型存在交集，如不存在，则可进行重载。

8.5 泛型兼容

泛型类被特化后，相当于产生了一种新的，不同的类型。如果使用相同的模板类型，则这些类型是赋值兼容的。具有与参数相同类型的泛型类的每个特化都是一个新的，不同的类型，但如果用于特化它们的模板类型相等，则这些类型是赋值兼容的。

8.6 重载和泛型

重载和泛型密切相关，泛型能够大大增强重载函数的表达能力。

示例

想象一下一个具有如下定义的泛型类：

```
type TMyClass<T> = class(TObject):
    T Add(T A, B): Add = A + B
        ;
    ;
```

扩展它，然后对其进行重载，使它能够处理 Matrix 类型：

```
type TMyMatrixClass = class:
    via TMyClass<Matrix>:
        Matrix Add(Matrix A, B): … ;
        ;
    ;
```

第9章 对 象

Tripod 支持面向对象程序设计，并且 Tripod 支持原始的对象类型。

对象应当被视为一个特殊的类，这个类不可用于定义其他对象，可以不经实例化直接访问成员，但可以定义指向对象的引用和指针。该类包含了对象定义中声明的所有字段，以及指向与对象类型关联的方法的指针。

语法

```
type NewObj = object(SuperObject, …):
    …
    ;
```

参数

NewObj
: 新建对象的标识符。

SuperObject
: 所继承的父对象。

示例

以下是对象的有效定义：

```
type TObj = object:
    private Info:
        byte*[] Caption
        ;
    public Action:
        void init()
        void done()
        void SetCaption(byte* AValue)
        byte* GetCaption()
        ;
    ;
```

声明一个对象就像声明一个类一样，内部像类一样也可以包含函数。对象也能继承和扩展，只不过只能从其他的对象继承和扩展，而不能继承类。这意味着可以使用这些字段和方法，就好像它们包含在声明为“子”对象的对象中一样。

对象内部也可以具有跟类一样的访问权限对象。对象默认具有 public 的访问权限。

对象支持虚方法和虚变量的声明。如果对象内部包含虚对象或者虚方法，则这个对象不能直接使用，需要透过子对象间接使用。

语法

```
Declaration = virt
```

参数

Declaration
变量或函数的声明。

示例

下例声明了一个具有虚成员变量的对象:

```
type TObj = object:
   private Info:
      byte*[] Caption = virt
      ;
```

上述示例中的对象具有一个虚成员变量，它具有私有的访问权限，无法被重载、重写、定义，所以任何继承这个对象的对象都将变为虚对象，这可能是一种潜在的错误。除非您清楚的知道自己在做些什么，否则请不要把虚成员引入对象的 private 部分。

虚成员的这种声明方式也可以引入类中，使得原本的实类变成虚类，这样的类不可以直接进行实例化。只能通过子类或派生类进行实例化。

第 10 章　转换约定

将给定类型变量的数据传输到另一种类型的变量需要转换。

当一个运算符的几个操作数类型不同时，就需要通过一些规则把它们转换为具有某种共同的类型。一般来说，自动转换是指把“比较窄的”操作数转换为“比较宽的”操作数，并且不丢失信息的转换。

例如，在计算表达式 f+i 时，将整型变量 i 的值自动转换为实数型（这里的变量 f 为实数型）。不允许使用无意义的表达式，例如，不允许把 float 类型的表达式作为下标。

针对可能导致信息丢失的表达式，编译器可能会给出警告信息，比如把较长的整型赋给较短的整型变量，把实数型值赋值给整型变量，等等，但这些表达式并不非法。

10.1 类型兼容

在 Tripod 中，如果两种类型属于同种类型，或者“足够接近”，在许多情况下可以看成是相同的，这两种类型就是兼容的。

对于两种兼容的类型，它们可以是相同的类型，也可以是具有某些共同属性的引用、函数或数组。后面的小节将讨论这些特殊的规则。

每两个兼容类型具有一个相关联的合成类型，它是由两种兼容类型所产生的公共类型。它有点类似于使用寻常双目转换对两个整数类型进行组合，产生一个公共的结果类型，以便在一些算术操作符中使用。我们将在描述类型兼容性规则时描述由两个兼容类型所产生的合成类型。

10.1.1　等价类型

只有当两种算术类型是同一类型时，它们才是兼容的。

如果两种类型是相同的，或者确切的说，这两个类型的本质、本源相同，那么它们必然是兼容的，并且它们的合成类型也属于同一种类型。在 Tripod 中，任何限定符都可能会改变类型。例如，const int 和 int 类型并不兼容，当然更不属于相同的类型。

通过 type 可以创建一个新类型，这个新类型形式上与源类型不同，但是本质上相当于源类型的别名，支持源类型所支持的运算。

示例

在这些声明之后，p 和 q 的类型是相同的。x 和 y 的类型是相同的，但它们都和 u 的类型不同。类型 TS 和 class S 是相同的，u、w 和 y 的类型也是相同的。

```
byte* p, q
class A: int a, b ;
A x,y
class S: int a, b ;
S TS = type
S w
```

```
TS y
```

10.1.2 枚举兼容

每个枚举类型定义产生了一种新的整数类型。Tripod 要求每种枚举类型与编译器所定义的表示它的整数类型保持兼容。对于同一个程序中的不同枚举类型，它们的兼容整数类型可能不同。合成类型是枚举类型。同一个模块中所定义的两个不同的枚举类型是不兼容的。

示例

在下面的声明中，E1 和 E2 的类型是不兼容的。但 E1 和 E3 的类型是兼容的，因为它们属于同种类型。由于枚举类型一般被当做整数类型，因此不同枚举类型的值可以混合使用，而不必考虑类型兼容性。

```
enum e: a, b;
e E1, E2
enum ex: c, d;
ex E3
```

10.1.3 数组兼容

对于两个具有相似限定形式的数组，只有当它们的元素类型兼容时，它们才是兼容的。

如果数组指明了阶数、阶上指明了维度，则它们的阶数和长度、维度也必须相等。但是，如果只有一个数组指定了所有阶的常量维度长度，或者两者都未指定常量长度（必须保证阶数和维数相同），则这两种类型也是兼容。

两个兼容数组类型的合成类型就是具有合成元素类型的数组类型，并且具有和那两个数组相同的类型限定形式。如果任一源类型在定义时指定了数组维度的长度，则合成类型也是这个长度。否则，它的长度就是未指定的。如果数组的长度未指定，并且使用它们的上下文环境要求它们兼容，则它们运行时长度必须相同，否则结果就是未定义的。

示例

下面这些数组的兼容情况如注释所示。

```
int A[3,3,3][ , ]          // 与  B, C     兼容
int B[3,3,3][2,2]          // 与  A, C     兼容
int C[ , , ][ , ]          // 与  A, B, D  兼容
int D[4,4,4][ , ]
const int E[,,][,]         // 不与任何类型 兼容
```

E 与其他类型不兼容，因为 E 是一个引用类型，不是数组类型，这里的声明仅仅创建了一个可以引用这种类型对象的别名。

10.1.4 函数兼容

为了使两个函数保持兼容，必须满足下面这些条件：

一、函数的返回类型必须是兼容的。

二、形参的数量以及省略号的使用必须一致。

三、对应的形参必须具有兼容的类型。

形参的名称并不需要一致。合成类型是满足形参为合成形参类型、以相同的方式使用省略号并且具有

兼容返回类型的函数。

10.1.5 对象、类和联合兼容

在每个对象、类或联合类型定义中，它的类型标识符引入了一种新的对象、类或联合类型。它们并不与同一个模块中所定义的其他对象、类或联合类型相同，也不与它们兼容。

10.1.6 指针兼容

对于两个具有相似限定形式的指针，当它们指向兼容的类型时，它们就是兼容的。两个兼容的指针类型的合成类型是指向合成类型的指针（采用相似的限定形式）。

10.2 转换

Tripod 允许在几种情况下把一种类型的值转换为另一种类型：

一、类型转换表达式可以把一个值显式转换为另一种类型。

二、为了执行算术或逻辑运算，操作数的值可以隐式地转换为其他类型。

三、一种类型的对象可能会赋值给另一种类型的地址单元，这会引起隐式类型转换。

四、函数的实际参数在函数调用之前可能会隐式地转换为另一种类型。

五、函数的返回值可能会在函数返回之前隐式地转换为另一种类型。

一个特定的类型并不能够转换为任何类型，而是存在一定的限制。例如，在赋值时可能出现的所有转换与强制类型转换时可能发生的所有转换并不相同。

10.2.1 表示形式的变化

当一个值从一种类型转换为另一种类型时，可能会发生表示形式的变化，也可能不发生变化。例如，当两个类型具有不同的长度时，就必定出现表示形式的变化。当整数转换为实数表示形式时，即使整数和实数具有相同的长度，也必定会出现表示形式的变化。但是，当一个 int 类型的值转换为 uobyte 类型时，就并不一定会发生表示形式的变化。

一些表示形式的变化是很简单的，只是抛弃多余多余的位或者用 0 填充相应的位。其他改变可以是复杂的，例如，在整数和浮点数表示之间的转换。对于下节中讨论的每一种转换，我们都描述了可能需要的表示变化。

10.2.2 细微的转换

把一个值从一种类型转换为另一种相同（或兼容）的类型总是可行的。在这种情况下，不会发生表示形式的改变。大多数编译器拒绝把结构或联合类型转换为自身类型，因为一般不允许对结构或联合进行转换。

10.2.3 转换为算术类型

标量类型（算术类型和指针）可以转换为整数类型。

对于整数类型，规则取决于以下排名：

在 32 位：

byte < **ubyte** < **dbyte** < **udbyte** < **qbyte** < **int** < **uqbyte** < **obyte** < **uobyte**

在 64 位：

byte < ubyte < dbyte < udbyte < qbyte < uqbyte < obyte < int < uobyte

首先，根据上面的排名，所有小于 int 的类型都被转换为 int 类型，然后，如果类型具有不同的大小，则转换较小的类型以匹配较大的类型。接下来，如果参数具有符号，则将转换为无符号类型以匹配另一个类型。

最后，int 类型将被替换：

在 32 位：

qbyte

在 64 位：

obyte

另一方面，如果其中一个类型是 Single 或 Double，则两个参数都被转换或提升为 Double。

10.2.4 数字数据的转换

赋值表达式或变量作为参数传递给函数或作为函数返回的结果时，将隐式发生类型转换。通过强制类型转换可以使得转换强制进行。

int 型到 int 型、有符号类型、无符号类型的转换：

- 任何整数类型到较小的整数类型：保留最低有效位
- 任何整数类型到更大的整数类型：符号扩展以填充最高有效位

整数类型转换为单精度或双精度实数类型：

- 可能会失去精确度

双精度转换为单精度实数：

- 可能会丢失精度
- 如果浮点数的值超出目标类型的范围，则结果是未定义的。

10.2.5 转换为对象、类和联合类型

非兼容的对象、类和联合类型之间不允许进行转换。

10.2.6 转换为枚举类型

它与转换为算术类型的规则相同。一些允许的转换，如在枚举类型和浮点类型之间，可能是不好的编程风格的特征。

10.2.6 转换为指针类型

一般而言，指针和整数类型可以转换为指针类型。在一些特殊的情况下，数组或函数也可以转换为指针。

任何类型的 null 指针都可以转换为其他指针类型，并且仍然可以被当做 null 指针。在转换之后，null 指针的表示形式可能会发生变化。

对于任意的类型 S 和 D，一个“指向 S 的指针”类型的值可以转换为“指向 D 的指针”。对象指针不能被转换为函数指针，反之亦然，但是，转换所产生的结果指针的行为可能受到表示形式改变以及编译器所施

加的对齐限制的影响。

整数常量 0 或任何值为零的整数常量，或者转换为 void *类型的这种常量，都可以表示 null 指针常量，并且总是可以转换为其他指针类型。这种转换的结果是与其他任何合法指针不同的 null 指针。不同指针类型的 null 指针可具有不同的内部表示形式。nuil 指针并不一定要把它的所有位都设置为 0。

除了常量 0 之外的整数也可以转换为指针类型，但转换结果是不可移植的。进行这种转换的意图是指针被认为是一种无符号整数类型（具有与指针类型相同的长度），并根据标准的整数转换规则把源类型转换为目标类型。

“T 数组”类型的表达式被转换为“指向 T 的指针”类型的值,这个指针指向这个数组的第 1 个元素。这是寻常单目转换的一种。

通过对函数替代指针，“返回 T 的因数”（例如，函数指定者）类型的表达式转换为“指向返回 T 的函数的指针”类型的值。这通常发生在单目转换部分中。

10.2.7 转换为引用类型

要转换为引用类型，直接对引用变量赋值即可，如果需要更改引用变量所引用的对象，则需要进行强制取引用。

10.2.8 转换为数组和函数类型

Troup 不允许把任何类型转换为数组或函数类型。

10.2.9 转换为 void 类型

任何值都可以转换为 void 类型。当然，这种转换的结果无法用来表示任何东西。这种转换可能发生在表达式的值将被丢弃的场合，例如在表达式语句中。

10.3 寻常转换

本章前面所讨论的所有转换都可以通过类型转换显式地进行，而不会出现错误。注意，Tripod 并不允许函数指针显式地直接转换为对象指针(或反之)，尽管这种转换可以通过一种适当的整数类型来进行。这个限制反映了对象指针和函数指针在表示形式上可能具有很大的区别。

类型限定符的存在与否并不会彭响类型转换的有效性，有些类型转换可以绕过类型限定符。在赋值时允许进行的转换具有更强的限制性。

一个指针对象转换为 void *然后再转换回原来的类型后将保持它的原值。

10.3.1 赋值转换

在简单的赋值表达式中，赋值操作符左边和右边的表达式的类型应该是相同的。如果不是，就会试图把赋值符右边的表达式的值转换为左边的类型。

10.3.2 单目转换

单目转换在执行一个操作之前确定是否并如何对它的唯一操作数执行转换。这种转换的目的是尽量减少运算符必须处理的算术类型的数量。对干单目运算符!、-、+、~和*，它们的操作数会自动进行转换。对于双目的<<和>>操作符，每个操作数会单独进行转换。

10.3.3 双目转换

当两个值必须一起进行操作时，它们首先根据双目转换规则转换为一种公共类型，它一般也就是结果的类型。在绝大多数双目运算符中，转换将作用于两个操作数，单目转换和双目转换合称为算术转换。

对两个操作数执行双目转换的运算符首先独立地对每个操作数执行单目转换，加宽绞短的值，并把数组和函数转换为指针。在此之后，如果任一操作数不是算术类型或者两个操作数属于相同的算术类型，则不再执行进一步的转换。

10.3.4 默认函数实参转换

如果表达式作为函数调用的实参，必须将参数表达式的类型转换为相应的形式参数的类型。函数的形式参数的声明类型以及返回值的类型受一些类型调整的影响，其方式与函数实参的转换相同。

示例

下述示例简单的演示了使用强制类型转换，类函数和赋值转换的功能：

```
type UDT = class:
    int I
    int operator cast():
        return I
        ;

    int operator = ():
        I = 0
        ;
    void UDT(int I_):
        I = I_
        ;
    ;

int J = 12
UDT u1, u2
u1(J)                           // 显式调用类函数
u2 = J                          // 隐式调用重载的赋值运算符

u1.I = 34
J = UDT(u1)                     // 显式调用重载后的强制类型转换 cast()
```

第 11 章　表达式

利用语言提供的运算符、关键字与特殊类型的变量（操作数、字面量等）组成公式，对具有某种类型的变量进行各种运算从而得到某种结果，这种公式被称为表达式。

程序中的大部分工作都是通过计算表达式来完成的，无论是因为它们的副作用，例如变量的赋值，还是对它们求值。表达式通过组合可以构成更大的表达式，可以在较大的表达式中用作参数或操作数，或者影响语句中的执行顺序，或者两者兼有。

表达式通常由两个组件构成：运算符及其操作数。通常运算符是双目的，即它需要 2 个操作数。双目运算符总是出现在操作数之间（如 `X/Y`）。有时一个运算符是单目的，即它只需要一个参数。单目运算符总是出现在操作数之前，如`-X`，特殊的也能出现在操作数之后如 `X*`。

本章将介绍 Tripod 中有关表达式的概念，以及各种运算符和表达式的用法。

11.1 概览

Tripod 所提供的运算符非常丰富，大多借鉴于 C 系语言，在用法与形式上与 C 系语言有很多相通的地方，部分细节来源于 Basic 或 Pascal。Tripod 自身几乎没有增加任何额外的运算符，但它为某些运算符提供了新的功能，用以满足多样化的需求。

本节所描述关于运算符的详细语法完整指定了 Tripod 中所有运算符的功能、性质、语法、优先级和结合性。为了先对这些信息有一个大概的认识，表 11-1 提供了运算符各种性质的大纲，按照优先级从高到低排列，并列出了这些运算符的结合性。

表 11-1 运算符

标记	运算符	类型	优先级	结合性
Algebra、Literal	简单标记	基本	17	N/A
Array[…][]…	下标	后缀	16	从左到右
Function(…)()…	函数调用	后缀	16	从左到右
.	解引用	后缀	16	从左到右
类型限定符(表达式)	强制类型转换	前缀	16	从右到左
*	指针求值	前缀	15	从右到左
++、--	增值、减值	后置	14	从左到右
~	按位非	前置	13	从右到左
-、+	算术负、正号	前置	13	从右到左
Not	逻辑非	前置	13	从右到左
++、--	增值、减值	前置	13	从左到右
@	取地址	前置	13	从右到左
*、/、\、%	乘，除，整除，取模	双目	12	从左到右
+、-	加减	双目	11	从左到右
<<、>>、>>>	左右移位	双目	10	从左到右

标记	运算符	类型	优先级	结合性
<、>、<=、>=	关系	双目	9	从左到右
==、<>	相等、不相等	双目	8	从左到右
&	按位与	双目	7	从左到右
^	按位异或	双目	6	从左到右
|	按位或	双目	5	从左到右
And、AndAlso	逻辑与	双目	4	从左到右
Or、OrElse	逻辑或	双目	3	从左到右
=、+=、-=、*=、/=、\=、%=、<<=、>>=、&=、^=、|=、>>>=	赋值	双目	2	从右到左
+、-	动态创建、销毁	对象	N/A	从左到右

11.1.1 优先级

当在单个表达式中发生多个操作时，以预定按顺序求值和解析每个操作。这称为操作顺序或运算符优先级。优先级指示了在混合有一个或多个不同种类的运算符的表达式中，不同的运算符与操作数进行运算的优先次序。优先级高的运算符较优先级低的运算符相比，高优先级的运算符会优先对操作数进行运算。

11.1.2 结合性

结合性指示了在同一优先级的运算符结合的多个表达式之间，运算的方向，这称之为运算符的结合性。从左到右表示运算从左到右执行，从右到左表示运算从右到左执行。

通常，二元运算符（如+，^）和一元后缀运算符（如（），*）从左到右进行求值，并且一元前缀运算符（如 Not，@）从右到左进行求值。

具有“N/A”相关性的运算符可能表示没有具体的优先级和结合性指定，需要手动进行指定。像 Cast 这样的成员函数运算符总是第一个被求值的，因为它们的语法需要括号。赋值运算符总是最后被求值的。括号可用于覆盖运算符优先级。括号内的操作在其他操作之前执行。在括号内使用正常的运算符优先级。

运算符优先级和结合性规则决定了表达式的含义。为了避免表达式可能会产生的副作用，以及副作用对表达式造成的不利影响，同时也为了能够更好地利用副作用，表达式的求值顺序必须严格遵照运算符的优先级和结合性顺序进行求值。

11.2 运算符

预定义的运算符分为以下几类:

一、赋值运算符

将一个操作数的值赋值给另一个操作数的运算符。

二、算术运算符

对其操作数执行数学计算并返回结果的运算符。

三、条件运算符

比较操作数之间关系的运算符。

四、逻辑运算符

对其操作数执行逻辑计算并返回结果的运算符。

五、索引运算符

根据索引值返回对变量或对象的引用的运算符。

六、增值运算符

对算术类型对象进行增值减值操作。

七、指针运算符

使用指针和地址的运算符。

八、类型或类运算符

提供对 Type 或 Class 成员的访问的操作员。

九、内存运算符

为内存分配内存和构造对象的运算符。

下面的小节将单独介绍 Tripod 的各种运算符。

11.2.1 赋值运算符

赋值运算符将一个值赋给它的操作数。

赋值运算符根据第二个或右侧操作数的值执行对第一个或左侧操作数的赋值。大多数赋值运算符都是组合运算符，因为它们首先对两个操作数执行数学或按位运算，然后将结果赋给左侧操作数。

11.2.1.1 =[>]（赋值）

将值赋给一个变量。

语法

```
void operator let   (var T1 lhs; var T2 rhs)
void operator =[>]  (var T1 lhs; var T2 rhs)
```

用法

```
lhs = rhs
```

或

```
lhs => rhs
```

或，类似 Basic

```
[ let ] lhs = rhs
```

或

```
[ let ] lhs => rhs
```

参数

lhs

将被赋值的变量。

T1

任何数字，布尔和指针类型。

rhs

被赋给 lhs 的值。

T2

任何可转换为 T2 的类型。

该运算符将其右侧操作数（rhs）的值赋给其左侧操作数（lhs）。右侧操作数必须可隐式转换为左侧类型

（T1）（用于将布尔值转换为整数，false 或 true 布尔值将变为内部实际的整数值）。

请不要与 operator == (Equal)相混淆。这个运算符是起比较作用的，而不是用来进行赋值。

替代符号‘=>’可以用于代替‘=’的赋值。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int i
i = 420

if i == 69 :
    print("Error: i should equal 420")
    ;
    print("All is good.")
```

11.2.1.2 &=（按位与运算、赋值）

将操作数先按位与再进行赋值。

语法

```
int operator &= (var T1 lhs; var T2 rhs)
```

用法

```
lhs &= rhs
```

参数

lhs
将被赋值的变量。

T1
任何整数类型。

rhs
用 *lhs* 执行按位与的值。

T2
任何整数类型。

这个运算符将两个操作数先进行与运算，得到结果后，将结果赋值给左侧的变量（用于将布尔值转换为整数，false 或 true 布尔值将变为内部实际的整数值）。它在功能上等同于：

```
lhs = lhs & rhs
```

&=比较其操作数的每个位，lhs&rhs，如果两个位都是 1，则第一个操作数中的相应位 lhs 设置为 1。否则设置为 0。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
ubyte numeric_value1, numeric_value2
numeric_value1 = 15
numeric_value2 = 30
```

```
numeric_value1 &= numeric_value2          // Result:    14
print(numeric_value1)
```

11.2.1.3 |=（按位或运算、赋值）

执行按位或并将结果赋值给变量。

语法

```
int operator |= (var T1 lhs; var T2 rhs)
```

用法

```
lhs |= rhs
```

参数

lhs
将被赋值的变量。
T1
任何整数类型。
rhs
用 *lhs* 执行按位或的值。
T2
任何整数类型。

此运算符将两个操作数进行按位或运算并将结果赋值给变量（用于将布尔值转换为整数，false 或 true 布尔值将变为内部实际的整数值）。它在功能上等同于：

```
lhs = lhs | rhs
```

|=比较其操作数的每个位，lhs|rhs，如果任意一个位是 1，则第一个操作数中的相应位 lhs 设置为 1。否则设置为 0。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
ubyte a = 0b00110011
ubyte b = 0b01010101
a |= b
// Result:          a = 0b01110111
```

11.2.1.4 ^=（按位异或运算、赋值）

执行按位异或并将结果赋值给变量。

语法

```
void operator ^= (var T1 lhs; var T2 rhs)
```

用法

```
lhs ^= rhs
```

参数

lhs
将被赋值的变量。
T1
任何整数类型。
rhs
用 *lhs* 执行按位或的值。
T2
任何整数类型。

此运算符按位运行或将结果赋值给变量（用于将布尔值转换为整数，false 或 true 布尔值将变为内部实际的整数值）。它在功能上等同于：

lhs = lhs ^ rhs

^=比较其操作数的每个位，lhs^rhs，如果两个位相同（均为 1 或 0），则第一个操作数中的相应位 lhs 设置为 0。否则设置为 1。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
ubyte a = 0b00110011
ubyte b = 0b01010101
a ^= b
// Result:          a = 0b01100110
```

11.2.1.5 <<=（按位左移位运算、赋值）

向左移位并为变量赋值。

语法

```
void operator <<= (var int lhs; var int rhs)
void operator <<= (var uqbyte lhs; var uqbyte rhs)
void operator <<= (var obyte lhs; var obyte rhs)
void operator <<= (var uobyte lhs; var uobyte rhs)
```

用法

```
lhs <<= rhs
```

参数

lhs
将被赋值的变量。
rhs
用 *lhs* 执行左移位后的值。

该运算符将其左侧（lhs）参数中的位向左移动其右侧（rhs）参数指定的位数，并将结果赋给 lhs。它在功能上等同于：

lhs = lhs << rhs

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int i
i = 0b00000011
i <<= 3
// Result: 0b00011000
```

11.2.1.6 >>=（按位有符号右移位运算、赋值）

有符号向右移位并为变量赋值。

语法

```
void operator >>= (var int lhs; var int rhs)
void operator >>= (var uqbyte lhs; var uqbyte rhs)
void operator >>= (var obyte lhs; var obyte rhs)
void operator >>= (var uobyte lhs; var uobyte rhs)
```

用法

```
lhs >>= rhs
```

参数

lhs
将被赋值的变量。
rhs
用 *lhs* 执行右移位后的值。

该运算符将其左侧（lhs）参数中的位向右移动其右侧（rhs）参数指定的位数（空出来的位用 1 填充），并将结果赋给 lhs。它在功能上等同于：

```
lhs = lhs >> rhs
```

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int i
i = 0b10011000
i >>= 3
// Result: 0b11110011
```

11.2.1.7 >>>=（按位无符号右移位运算、赋值）

有符号向右移位并为变量赋值。

语法

```
void operator >>>= (var int lhs; var int rhs)
void operator >>>= (var uqbyte lhs; var uqbyte rhs)
void operator >>>= (var obyte lhs; var obyte rhs)
void operator >>>= (var uobyte lhs; var uobyte rhs)
```

用法

lhs >>>= rhs

参数

lhs
将被赋值的变量。
rhs
用 *lhs* 执行右移位后的值。

该运算符将其左侧（lhs）参数中的位向右移动其右侧（rhs）参数指定的位数（空出来的位用 0 填充），并将结果赋给 lhs。它在功能上等同于：
lhs = lhs >>> rhs
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int i
i = 0b10011000
i >>= 3
// Result: 0b00010011
```

11.2.1.8 +=（加法运算、赋值）

相加后为变量赋值。

语法

void operator += (**var** *T1 lhs*; **var** *T2 rhs*)

用法

lhs += rhs

参数

lhs
将被赋值的变量。
T1
任何算术类型。
rhs
将被加到 lhs 中的值。
T2
任何算术类型。

此运算符将两个操作数相加，并将结果赋给 lhs。它在功能上等同于：
lhs = lhs + rhs
对于算术类型，右侧表达式（rhs）将转换成左侧类型（T1）。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
double n = 6
n += 1
// Result:  7
```

11.2.1.9 -=（减法运算、赋值）

相减后为变量赋值。

语法

```
void operator -= (var T1 lhs; var T2 rhs)
```

用法

```
lhs -= rhs
```

参数

lhs
将被赋值的变量。
T1
任何算术类型。
rhs
将被加到 lhs 中的值。
T2
任何算术类型。

此运算符将两个操作数相减，并将结果赋给 lhs。它在功能上等同于:
lhs = lhs - rhs
对于算术类型，右侧表达式（rhs）将转换成左侧类型（T1）。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
double n = 6
n -= 2.2
// Result:  3.8
```

11.2.1.10 *=（乘法运算、赋值）

相乘后为变量赋值。

语法

```
void operator *= (var T1 lhs; var T2 rhs)
```

用法

```
lhs *= rhs
```

参数

lhs
将被赋值的变量。
T1
任何算术类型。
rhs
将被乘到 lhs 中的值。
T2
任何算术类型。

此运算符将两个操作数相乘，并将结果赋给 lhs。它在功能上等同于:
lhs = lhs - rhs
对于算术类型，右侧表达式（rhs）将转换成左侧类型（T1）。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
double n = 6
n *= 2
// Result:  12
```

11.2.1.11 /=（除法运算、赋值）

相除后为变量赋值。

语法

void operator /= (**var** *T1 lhs*; **var** *T2 rhs*)

用法

lhs /= rhs

参数

lhs
将被赋值的变量。
T1
任何算术类型。
rhs
将被除去 lhs 中的值。
T2
任何算术类型。

此运算符将两个操作数相除，并将结果赋给 lhs（这个运算符执行实数类型的除法运算，除数不可以为 0）。它在功能上等同于:
lhs = lhs / rhs
对于算术类型，右侧表达式（rhs）将转换成左侧类型（T1）。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
double n = 6
n /= 2.2
// Result:  2.7272727272
```

11.2.1.12 %=（取模运算、赋值）

取模后为变量赋值。

语法

```
void operator %= (var T1 lhs; var T2 rhs)
```

用法

```
lhs %= rhs
```

参数

lhs
将被赋值的变量。
T1
任何整数类型。
rhs
将被取模到 lhs 中的值。
T2
任何整数类型。

此运算符将两个操作数取模，并将结果赋给 lhs（这个运算符执行整数类型的除法运算）。它在功能上等同于：

```
lhs = lhs % rhs
```

对于算术类型，右侧表达式（rhs）将转换成左侧类型（T1）。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int n = 11
n /= 3
// Result:  2
```

11.2.1.13 \=（整除运算、赋值）

整除后为变量赋值。

语法

```
void operator %= (var T1 lhs; var T2 rhs)
```

用法

```
lhs \= rhs
```

参数

lhs
将被赋值的变量。
T1
任何算术类型。
rhs
将被整除到 lhs 中的值。
T2
任何算术类型。

此运算符将两个操作数整除，并将结果赋给 lhs（这个运算符执行整数类型的除法运算）。它在功能上等同于：

```
lhs = lhs \ rhs
```

对于算术类型，右侧表达式（rhs）将转换成左侧类型（T1）。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
double n = 6
n \= 2.2
// Result:  3
```

11.2.2 算术运算符

算术运算符用于数学表达式的计算。
算术运算符使用其操作数的值进行数学运算并返回结果。

11.2.2.1 +（加法运算）

求两个表达式的和。

语法

```
T operator + (var T lhs; var T rhs)
```

用法

```
res = lhs + rhs
```

参数

lhs
加法运算符左侧的表达式。
rhs
加法运算符右侧的表达式。
T
任何算术类型。

返回

返回两个表达式的和。

当左侧和右侧表达式是数值时，Operator+（加法运算）返回两个值的总和。如果 T 中存在一个指针类型，则执行指针类型的加法运算。

此运算符不会修改操作数。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
Single n
n = 4.75 + 5.25
```

11.2.2.2 -（减法运算）

求两个表达式的差。

语法

```
T operator - (var T lhs; var T rhs)
```

用法

```
res = lhs - rhs
```

参数

lhs

减法运算符左侧的表达式。

rhs

减法运算符右侧的表达式。

T

任何算术类型。

返回

返回两个表达式的差。

当左侧和右侧表达式是数值时，Operator+（减法运算）返回两个值的差。如果 T 中存在一个指针类型，则执行指针类型的减法运算。

此运算符不会修改操作数。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
Single n
n = 4 - 5
```

11.2.2.3 *（乘法运算）

求两个表达式的积。

语法

T **operator *** (**var** *T lhs*; **var** *T rhs*)

用法

*res = lhs * rhs*

参数

lhs
乘法运算符左侧的表达式。
rhs
乘法运算符右侧的表达式。
T
任何算术类型。

返回

返回两个表达式的积。

Operator *（乘法运算）返回两个值的积。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
Double n
n = 4 * 5
```

11.2.2.4 /（除法运算）

求两个表达式的商。

语法

T **operator /** (**var** *T lhs*; **var** *T rhs*)

用法

res = lhs / rhs

参数

lhs
除法运算符左侧的表达式。
rhs
除法运算符右侧的表达式。

T

任何算术类型。

返回

返回两个表达式的商。

Operator /（除法运算）返回两个值的商，与整数除法不同，浮点除零是可以安全执行的，商将保持表示无穷大的特殊值，将其转换为字符串返回类似“Inf”或“INF”的内容，精确文本是特定于平台的。

此运算符不会修改操作数。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
Double n
n = 6 / 2.3
```

11.2.2.5 %（取模运算）

求两个表达式的余数。

语法

```
T operator % (var T lhs; var T rhs)
```

用法

```
res = lhs % rhs
```

参数

lhs

取模运算符左侧的表达式。

rhs

取模运算符右侧的表达式。

T

任何整数类型。

返回

返回两个表达式的余数。

Operator %（取模运算）返回两个整数表达式的余数。通过向上或向下舍入将浮点数值转换为整数。

此运算符不会修改操作数。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int n
n = 47 % 2.3
n = 5 % 3
```

11.2.2.6 \（整除运算）

求两个整数表达式的商。

语法

```
T operator \ (var T lhs; var T rhs)
```

用法

```
res = lhs \ rhs
```

参数

lhs
整除运算符左侧的表达式。

rhs
整除运算符右侧的表达式。

T
任何整数类型。

返回

返回两个整数表达式的商。

Operator \（整除运算）返回两个整数表达式的商。通过向上或向下舍入将浮点数值转换为整数。如果除数是 0 会引发除 0 错误。

此运算符不会修改操作数。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int n
n = 47 \ 5
n = 7 \ 2.4
```

11.2.2.7 -（负数运算）

改变算术表达式的符号。

语法

```
T operator - (var T rhs)
```

用法

```
res = - rhs
```

参数

rhs
要取负的右侧表达式。

T
任何算术类型。

返回

返回表达式的相反数。

Operator -（负数运算）是前缀单目运算符，返回表达式的相反数。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
obyte n
n = -5
n = -n
```

11.2.2.8 +（正数运算）

形式的表示一个无符号数。

语法

T **operator +** (**var** *T rhs*)

用法

res = + *rhs*

参数

rhs
要取正的右侧表达式。
T
任何算术类型。

返回

右侧表达式的值。

Operator +（整数运算）是前缀单目运算符，不会改变表达式的值，用来显式标明一个无符号数。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
obyte n
n = +5
n = -5-(+5)
```

11.2.2.9 <<（左移位运算）

将数值表达式的位向左移动。

语法

T **operator <<** (**var** *T lhs;* **var** *T rhs*)

用法

res = lhs << rhs

参数

lhs
左移位运算符左侧的表达式。
rhs
左移位运算符右侧的表达式。
T
任何整数类型。

返回

lhs 左移位 rhs 指定的位数所得的值。

Operator <<（左移位运算）将左侧表达式（lhs）中的所有位移位左侧表达式（rhs）指定的位数。
此运算符不会修改操作数。
如果结果太大而无法容纳在结果的数据类型中，则丢弃最左边的位（“移出”）。
对于 rhs 小于零，或大于或等于结果数据类型中位数的值，此操作的结果未定义。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
byte n
n = 0b01010101
n << 3
```

11.2.2.10 >>（有符号右移位运算）

将数值表达式的位向右移动，保留符号位。

语法

T **operator >>** (**var** *T lhs;* **var** *T rhs*)

用法

res = lhs >> rhs

参数

lhs
有符号右移位运算符左侧的表达式。
rhs
有符号右移位运算符右侧的表达式。

T

任何整数类型。

返回

lhs 有符号右移位 rhs 指定的位数所得的值。

Operator >>（有符号右移位运算）将左侧表达式（lhs）中的所有位移位左侧表达式（rhs）指定的位数。如果是负数，空出来的位用 1 填充。

此运算符不会修改操作数。

对于 rhs 小于零，或大于或等于结果数据类型中位数的值，此操作的结果未定义。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
byte n
n = 0b11010101
n >> 3
```

11.2.2.10 >>>（无符号右移位运算）

将数值表达式的位向右移动，不保留符号位。

语法

T **operator >>>** (**var** *T lhs;* **var** *T rhs*)

用法

res = lhs >>> rhs

参数

lhs

无符号右移位运算符左侧的表达式。

rhs

无符号右移位运算符右侧的表达式。

T

任何整数类型。

返回

lhs 无符号右移位 rhs 指定的位数所得的值。

Operator >>>（无符号右移位运算）将左侧表达式（lhs）中的所有位移位左侧表达式（rhs）指定的位数。即使是负数，空出来的位也用 0 填充。

此运算符不会修改操作数。

对于 rhs 小于零，或大于或等于结果数据类型中位数的值，此操作的结果未定义。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
byte n
```

```
n = 0b11010101
n >> 3
```

11.2.3 关系运算符

比较关系。

关系运算符执行两个操作数的值之间的比较。每个运算符返回一个布尔结果，如果关系为真，则返回 true（1），否则返回 false（0）。

11.2.3.1 ==（等于）

比较两个操作数的相等关系。

语法

boolean **operator ==** (**var** *T lhs;* **var** *T rhs*)

用法

res = lhs == rhs

参数

lhs
等于运算符左侧的表达式。

rhs
等于运算符右侧的表达式。

T
任何算术类型。

返回

如果表达式相等则返回 true（1），如果不相等则返回 false（0）。

Operator ==（等于运算）是一个双目运算符，它将两个表达式进行相等性比较并返回结果。返回标准的布尔值，true（1），false（0）。

此运算符不会修改操作数。

不要在条件表达式中与赋值号（=）相混淆。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int i = 0
i = 420

if i = 60 :
   print("serious error: i should equal 420")
   ;
```

11.2.3.2 <>（不等于）

比较两个操作数的不等关系。

语法

```
boolean operator <> (var T lhs; var T rhs)
```

用法

```
res = lhs <> rhs
```

参数

lhs
不等于运算符左侧的表达式。
rhs
不等于运算符右侧的表达式。
T
任何算术类型。

返回

如果表达式不相等则返回 true（1），如果相等则返回 false（0）。

Operator <>（等于运算）是一个双目运算符，它将两个表达式进行相等性比较并返回结果。返回标准的布尔值，true（1），false（0）。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int i = 10, j = i

if i <> j :
   print("error: "; i; " does not equal "; j)
   ;
```

11.2.3.3 <（小于）

比较两个操作数的小于关系。

语法

```
boolean operator < (var T lhs; var T rhs)
```

用法

```
res = lhs < rhs
```

参数

lhs
小于运算符左侧的表达式。
rhs
小于运算符右侧的表达式。

T
任何算术类型。

返回

如果左侧表达式小于右侧表达式则返回 true（1），如果大于等于则返回 false（0）。

Operator <>（不等于）是一个双目运算符，它将两个表达式进行小于比较并返回结果。返回标准的布尔值，true（1），false（0）。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int i = 10, j = i

if i <> j :
   print("error: "; i; " does not equal "; j)
   ;
```

11.2.3.4 >（大于）

比较两个操作数的大于关系。

语法

```
boolean operator > (var T lhs; var T rhs)
```

用法

```
res = lhs > rhs
```

参数

lhs
大于运算符左侧的表达式。
rhs
大于运算符右侧的表达式。
T
任何算术类型。

返回

如果左侧表达式大于右侧表达式则返回 true（1），如果小于等于则返回 false（0）。

Operator <>（大于）是一个双目运算符，它将两个表达式进行大于比较并返回结果。返回标准的布尔值，true（1），false（0）。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
if (420 > 69):
```

```
print("420 > 69 is true.")
;
```

11.2.3.5 <=（小于等于）

比较两个操作数的小于等于关系。

语法

```
boolean operator <= (var T lhs; var T rhs)
```

用法

```
res = lhs <= rhs
```

参数

lhs
小于等于运算符左侧的表达式。
rhs
小于等于运算符右侧的表达式。
T
任何算术类型。

返回

如果左侧表达式小于等于右侧表达式则返回 true（1），如果大于则返回 false（0）。

Operator <=（小于等于）是一个双目运算符，它将两个表达式进行小于等于比较并返回结果。返回标准的布尔值，true（1），false（0）。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
if (69 <= 420):
    print("69 <= 420 is true.")
    ;
```

11.2.3.6 >=（大于等于）

比较两个操作数的大于等于关系。

语法

```
boolean operator >= (var T lhs; var T rhs)
```

用法

```
res = lhs >= rhs
```

参数

lhs
大于等于运算符左侧的表达式。
rhs
大于等于运算符右侧的表达式。
T
任何算术类型。

返回

如果左侧表达式大于等于右侧表达式则返回 true（1），如果小于则返回 false（0）。

Operator >=（大于等于）是一个双目运算符，它将两个表达式进行大于等于比较并返回结果。返回标准的布尔值，true（1），false（0）。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
if (420 >= 69):
    print("420 >= 69 is true.")
    ;
```

11.2.4 逻辑运算符

对其操作数执行按位计算并返回结果的运算符。
逻辑运算符对其操作数的值执行逻辑运算，并返回结果。这些运算符同时也是位运算符，因为通过对其操作数的每个位执行逻辑运算可以找到结果。

11.2.4.1 And（逻辑与）

返回两个操作数的按位与的结果。

语法

```
boolean operator And (var T lhs; var T rhs)
```

用法

```
res = lhs And rhs
```

参数

lhs
逻辑与运算符左侧的表达式。
rhs
逻辑与运算符右侧的表达式。
T
任何算术类型。

返回

返回两个操作数的按位与的值。

此运算符返回按其操作数按位与的值，这是一个逻辑运算，它产生一个值，其位设置取决于操作数的位数。下面的真值表演示了逻辑与运算操作的所有组合：

lhs Bit	Rhs Bit	结果
0	0	0
1	0	0
0	1	0
1	1	1

此运算符不会修改操作数。执行短路运算。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.4.2 Or（逻辑或）

返回两个操作数的按位或的结果。

语法

```
boolean operator Or (var T lhs; var T rhs)
```

用法

```
res = lhs Or rhs
```

参数

lhs
逻辑或运算符左侧的表达式。
rhs
逻辑或运算符右侧的表达式。
T
任何算术类型。

返回

返回两个操作数的按位或的值。

此运算符返回按其操作数按位或的值，这是一个逻辑运算，它产生一个值，其位设置取决于操作数的位数。下面的真值表演示了逻辑或运算的所有组合：

lhs Bit	Rhs Bit	结果
0	0	0
1	0	1
0	1	1
1	1	1

此运算符不会修改操作数。执行短路运算。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.4.3 Not（逻辑非）

返回操作数的按位取反的结果。

语法

boolean **operator Not** (**var** *T rhs*)

用法

res = *lhs* Or *rhs*

参数

rhs
逻辑或运算符右侧的表达式。
T
任何算术类型。

返回

返回操作数的按位取反的值。

此运算符返回按其操作数按位取反的值，这是一个逻辑运算，它产生一个值，其位设置取决于操作数的位数。下面的真值表演示了逻辑非运算的所有组合：

Rhs Bit	结果
1	0
0	1

此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.4.4 AndAlso（逻辑与，非短路）

返回两个操作数的按位与的结果。

语法

boolean **operator AndAlso** (**var** *T lhs*; **var** *T rhs*)

用法

res = *lhs* AndAlso *rhs*

参数

lhs
逻辑与运算符左侧的表达式。
rhs
逻辑与运算符右侧的表达式。
T
任何算术类型。

返回

返回两个操作数的按位与的值。

此运算符返回按其操作数按位与的值，这是一个逻辑运算，它产生一个值，其位设置取决于操作数的位数。下面的真值表演示了逻辑与运算操作的所有组合：

lhs Bit	Rhs Bit	结果
0	0	0
1	0	0
0	1	0
1	1	1

不执行短路运算始终求值两个表达式。
此运算符不会修改操作数。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.4.5 OrElse（逻辑或，非短路）

返回两个操作数的按位或的结果。

语法

```
boolean operator OrElse (var T lhs; var T rhs)
```

用法

```
res = lhs OrElse rhs
```

参数

lhs
逻辑或运算符左侧的表达式。
rhs
逻辑或运算符右侧的表达式。
T
任何算术类型。

返回

返回两个操作数的按位或的值。

此运算符返回按其操作数按位或的值，这是一个逻辑运算，它产生一个值，其位设置取决于操作数的位数。下面的真值表演示了逻辑或运算的所有组合：

lhs Bit	Rhs Bit	结果
0	0	0
1	0	1
0	1	1
1	1	1

不执行短路运算始终求值两个表达式。

此运算符不会修改操作数。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.5 索引运算符

运算符基于一个索引返回一个引用。

索引运算符根据其第二个或右侧操作数的值返回对某个内存的引用。此操作数用作从第一个或左侧操作数表示的某些内存的开头的索引或偏移量。

语法

```
T operator [] (var T* lhs; var int rhs ...)
```

用法

```
res = lhs OrElse rhs
```

参数

lhs

基址。

rhs

偏移。

T

任何数据类型。

*T**

类型 T 的指针。

此运算符返回对基址中内存距离值的引用。它基本上是“(lhs + rhs[0] + rhs[1] + …)*”的简写，因为引用可以被认为是一个指针，其值为存储位置“(lhs + rhs[0] + rhs[1] + …)”，并且隐含地取消引用；两者都完全一样。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

示例

```
int array[4] = (0, 1, 2, 3, 4)
int* iptr = @array[0]
int i = 0
loop if i < 4:
    print(iptr[i])
    ;
```

11.2.6 增值运算符

增值运算符使对象自身增加 1。

单目运算符++和—分别用于把它们的操作数的值增加 1 或减少 1，同时把操作数修改后的值作为结果。它们是具有副作用的操作（这两个运算符还存在后缀形式）。

11.2.6.1 ++（前缀自增）

返回操作数自增 1 后的值。

语法

```
T operator ++ (var T rhs)
```

用法

```
res = ++ rhs
```

参数

rhs
自增运算符右侧的对象。

T
任何算术类型。

返回

返回操作数自增 1 后的值。

如果操作数是个指针，则遵循指针运算的规则。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.6.2 --（前缀自减）

返回操作数自减 1 后的值。

语法

```
T operator -- (var T rhs)
```

用法

```
res = -- rhs
```

参数

rhs
自减运算符右侧的对象。

T
任何算术类型。

返回

返回操作数自减 1 后的值。

如果操作数是个指针，则遵循指针运算的规则。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.6.3 ++（后缀自增）

返回操作数自增 1 前的对象。

语法

T **operator ++** ()

用法

res = rhs ++

参数

rhs
自增运算符右侧的对象。
T
任何算术类型。

返回

返回操作数自增 1 前的对象。

如果操作数是个指针，则遵循指针运算的规则。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.6.4 --（后缀自减）

返回操作数自减 1 前的对象。

语法

T **operator --** ()

用法

res = rhs --

参数

rhs
自减运算符右侧的对象。
T
任何算术类型。

返回

返回操作数自减 1 前的对象。

如果操作数是个指针，则遵循指针运算的规则。
可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.7 指针运算符

指针运算符应用于指针操作。
指针运算符提供了检索其操作数的内存中的地址，以及使用或取消引用该内存的功能。

11.2.7.1 @（取地址）

返回字符串文字，变量或对象的地址。

语法

```
T* operator @(var T rhs)
```

用法

```
res = @rhs
```

参数

rhs
地址运算符右侧的对象。

T
任何数据类型。

返回

返回右操作数的内存地址。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.7.2 *（指针求值）

解引用一个指针。

语法

```
T operator *()
```

用法

```
res = rhs*
```

参数

rhs
指针求值运算符左侧的对象。

T
任何数据类型。

返回

返回特定内存地址的值或对象。

可以使用适当的语法将此运算符作为成员运算符重载为用户定义的类型。

11.2.8 类型运算符

用以对对象或类的运算。
这些运算符在给定对象或其内存地址的情况下返回对对象成员的引用。

语法

```
U operator .(var T lhs)
U operator .(var T* lhs)
```

用法

```
res = lhs . rhs
```

参数

lhs
一个对象。

T
任何数据类型。

rhs
访问成员的名字。

U
rhs 所引用的类型。

返回

返回 rhs 所指定的成员。

这个运算符不可以被重载。

11.2.9 内存运算符

内存运算符用以操作内存中的对象。
内存运算符提供了一种动态分配和释放变量和对象的方法。

11.2.9.1 +（动态创建）

在内存中分配和构造对象。

语法

```
T operator +(T)
T* operator +(T)
```

用法

```
res = T+
```

参数

T
任何数据类型。

返回

如果接受动态创建对象的是一个引用,则返回引用;如果接受动态创建对象的是一个指针,则返回指针。动态创建用于在运行时创建对象，与预先声明的对象不同，动态创建的对象称为动态对象，动态对象在离开

动态对象的作用域后，动态对象的初始标识符虽然已经失效，但是动态对象仍在内存中，并未被销毁。动态创建的对象可以跨作用域使用。

动态创建表达式所用的语法结构是在可以动态创建的对象类型名后添加一个加号，如果动态创建成功，则返回对象的引用，失败则返回 nil。对于指针来说，创建失败返回 null。

这个运算符可以重载已进行自定义的构建。

11.2.9.1 -（动态销毁）

销毁动态创建的对象。

语法

```
int operator -()
```

用法

```
res = lhs-
```

参数

lhs

要销毁的对象。

返回

如果接受动态创建对象的是一个引用，则返回引用；如果接受动态创建对象的是一个指针，则返回指针。如果动态对象销毁成功则返回大于 0 的值，失败则返回小于 0 的值，在已经被回收后再次销毁返回 0。

这个运算符可以被重载已进行自定义的销毁过程。

11.3 运算符重载

Tripod 支持运算符重载。这意味着可以在自定义类型上定义某些运算符的动作，从而允许在数学表达式中使用这些类型。定义操作符的操作非常类似于函数或过程的定义，只有对可能的定义有一些限制，如后面所示。

从本质上讲，运算符重载是一种强大的符号工具；不过运算符重载提供的功能不多，通过常规函数调用也可以获得相同的结果。使用运算符重载时，请务必记住某些隐式规则可能会产生一些意外结果。

定义运算符的方式与定义函数非常相似。

语法

```
T operator OP ():
    …
;
```

具体定义运算符的方式可以参考之前的运算符定义。

第 12 章　语　句

每个算法的核心是它所采取的动作。这些动作容纳在语句或者模块中。语句组合可以形成语句块，goto 语句可以执行向语句块的跳转。

12.1 语句规则

Tripod 的语句采用分号和换行符作为分隔符分隔语句，在任何括号的内部将忽略换行符，在以冒号展开的复合语句中不可以使用分号作为分隔符。Tripod 支持类 C 语言中的花括号复合表达式，在花括号中，分号起到语句分隔符作用，而不是语句结束符的作用。

示例

```
a = b
{b = c; d = e}
x = y
```

在 Tripod 中，有一类被称为连接器的特殊符号，这些符号可以无视换行符的分隔功能，将跨行的词法元素结合在一起。根据连接器的连接方式不同，又区分为三种连接器类型：一种是双向连接器，这种连接器能够无视它前面和后面的 0 个或多个换行符；另一种是前向连接器，这种连接器能够无视它前面的 0 个或多个换行符；最后一种是后向连接器，在其他语言中又称为续行符，这种连接器能够无视它后面的 0 个或多个换行符。

Tripod 中的续行符是行末下划线（_），与 Basic 相同。

展开语句由冒号双向连接器（:）开始，由分号前向连接器（;）结束，中间包含语句序列。展开表达式一般用于函数或类的定义。复合语句由左花括号（{）和右花括号（}）以及中间的语句序列构成，复合表达式是展开语句的另一种形式，二者功能相同。

目前，所有的逻辑运算符（And、AndAlso、Or、OrAlso、Not）都是双向连接器，展开表达式的开头部分冒号（:）和逗号（,）也是双向连接器；展开表达式的结尾分号号（;）是前向连接器；续行符（_）是后向连接器；左花括号（{）是双向连接器，右花括号（}）是前向连接器。

在任何括号包围表达式中，换行符被视为空白字符。

12.2 简单语句

简单的语句不能在单独的语句中分解。

12.2.1　表达式语句

表达式语句由单独的一个表达式构成，用分隔符分隔分隔连续的表达式，使这些表达式成为单独的语句。这种语句的执行方式是对表达式进行求值，然后丢弃它所产生（如果有）的结果。

只有当表达式的求值具有副作用时，表达式语句才是有用的，例如把一个值赋值给一个变量或者执行输入或输出。这种表达式通常是赋值表达式，增值或减值表达式或者函数调用。

示例

```
speed = distance / time
++event_count
print("Again?")
pattern &= mask
if x < y : ++x , ++y
```

对于没有副作用并且求值结果将被丢弃的表达式，编译器可以不对它进行求值，或者不对它的一部分进行求值。

12.2.2 跳转语句

Tripod 支持多种跳转语句，每种跳转语句都有它自己专用的功能。

12.2.2.1 pass、break 和 exit 语句

pass 和 break 语句用于更改循环内部的控制流（break 语句还可以从 block 块中退出）。使用这两种语句可以实现与 goto 语句相同的功能，并被认为是更好的编程风格。

语法

```
pass
break
```

break 语句的执行导致最接近外层的 loop 语句或 block 语句终止。程序的执行立即跳转到被终止的语句之后的位置。如果 break 语句并不位于任何 loop 语句或 block 语句内部，将会出现错误。

pass 语句终止最接近的 loop、block 语句的执行。程序的执行立即转移到语句体的最后，受影响的迭代语句的执行从循环测试，如果是 block 语句，则跳转到 block 的入口位置重新开始。如果 pass 语句并位于任何 loop、block 语句内部，将会出现错误。

exit 语句终止所有层级的条件，迭代，块复合语句，直接跳转到嵌套语句的末尾。

示例

利用 break 终止循环的功能，下例用输入字符填充一个数组，当数组填满或者输入用尽时停止填充：

```
import io
byte array[100]
int i, c

…
loop if i < 100 :
    c = getchar()
    if c == EOF : break ;
    array[i] = c
    i++
    ;
```

示例

下例演示了 exit 退出层层嵌套复合语句的能力:

```
if A:
    loop if B:
        loop for C:
            exit
            ;
        ;
    ;
// E: Exit 将会跳转到这个位置
```

12.2.2.2 return 语句

return 语句用于终止当前函数，并可以返回一个值。

语法

```
return expression
return
```

return 语句的执行导致当前函数终止执行。程序的控制转移到函数的调用者或者紧随函数调用点之后的位置。

如果 return 语句中未出现表达式,函数的返回值默认是函数体中最后一个计算的表达式的值。如果 return 语句中出现了表达式，则函数的返回类型不可以是 void，否则这条语句就是非法的。返回表达式将进行转换，就像赋值给函数的返回类型一样。如果不能进行这种转换，则 return 语句是非法的。

12.2.2.3 goto 语句

goto 语句可以把控制流程转移到函数中的任何地方。

语法

```
goto mySubject
```

goto 关键字后面的代码块名必须与当前函数中某个 subject 名相同。执行 goto 语句使程序的控制流立即跳转到函数中该块的起始位置，接下来执行块中的内容，如果块内部没有 break 语句，则 goto 语句在执行完代码块后控制会回到原来 goto 语句的下一条语句。

使用 goto 语句时，必须牢记以下内容:

一、必须在与 goto 语句相同的块或子块中定义代码块。

二、从循环外部跳到循环内部反之亦然会产生奇怪的效果。

三、Tripod 的 goto 语句是会返回的，不同于其他编程语言中的 goto 语句一去不返。

12.3 结构语句

结构化语句可以分解成组成结构语句的关键结构部分和可以重复、有条件或者顺序执行的较小的简单语句部分。接下来的小节将分别阐述结构化语句的详细构造和功能。

12.3.1 展开语句

冒号（:）和分号（;）以及所包围的语句序列称之为展开语句，展开语句功能上将一系列的语句进行分组，顺序执行这些语句。形式上，展开语句内部以换行符作为语句之间的分隔符。展开语句每一个冒号总是对应有一个分号结束。

展开语句在特殊结构语句的形式上相较复合语句还有一些特殊的用法，这将在之后的小节详述。

12.3.2 条件语句

条件语句共有三种形式：if、for 以及 do 语句。

12.3.3.1 if 语句

语法

```
if expression :
    …
    ;
```

或

```
if expression :
    ; Else …
```

if 关键字和冒号之间的表达式必须能得到布尔类型的结果，或者能够自动转换为布尔类型的结果。如果表达式求值为 true，则执行冒号内部的语句；如果表达式求值为 false 则执行 Else 中的语句。Else 前有个逗号，逗号和 Else 可省略。

示例

```
int signum(int x):
    if x > 0:
        return 1;
        Else if x < 0:
                return -1 ;
                Else return 0
                ;
```

12.3.3.2 for 语句

语法

```
for expression :
    …
    ;
```

或

```
for expression :
    ; Else …
```

for 关键字和冒号之间的表达式必须能得到布尔类型的结果，或者能够自动转换为布尔类型的结果。如果表达式求值为 false，则执行冒号内部的语句；如果表达式求值为 true 则执行 Else 中的语句。Else 前有个逗号，逗号和 Else 可省略。

示例

```
int times = 61
for times < 61:
    times --
    ; Else Do :
        print("Times OK!")
        ;
```

12.3.3.3 do 语句

语法

```
do : … ;
```

do 语句不进行条件判断，直接执行 do 语句内部的语句。

12.3.4 迭代语句

在条件语句前加上关键字 loop 便可以构成迭代式语句，迭代式语句用于控制内部语句的重复执行。

12.3.4.1 loop if 语句

语法

```
loop if expression :
    …
    ;
```

或

```
loop if expression :
    ; Else …
```

if 关键字和冒号之间的表达式必须能得到布尔类型的结果，或者能够自动转换为布尔类型的结果。如果表达式求值为 true，则执行内部的语句一次，然后会跳到条件判断部分进行下一次判断和执行。迭代语句中的 Else 子句只有当第一次条件判断失败时，执行一次，如果第一次条件判断成功，则 Else 将永远不会执行。

Else 前有个逗号，逗号和 Else 可省略。

示例

```
int i = 2
loop if i <= 100:
    println("i = "; i)
    i += 2
    ;
```

```
int x = 1
loop if x > 10e-3:
    x /= 2
    ;
```

12.3.4.2 loop for 语句

语法

```
loop for expression :
    …
    ;
```

或

```
loop for expression :
    ; Else …
```

for 关键字和冒号之间的表达式必须能得到布尔类型的结果，或者能够自动转换为布尔类型的结果。如果表达式求值为 false，则执行内部的语句一次，然后会跳到条件判断部分进行下一次判断和执行。迭代语句中的 Else 子句只有当第一次条件判断失败时，执行一次，如果第一次条件判断成功，则 Else 将永远不会执行。loop for 语句表示直到某个条件成立时才退出循环。

Else 前有个逗号，逗号和 Else 可省略。

示例

```
int pow(int base, exponent):
    int res = 1
    loop for exponent <= 0:
        if exponent % 2 <> 0:
            res *= base ;
        base *= base
        ;
    return res
    ;
```

12.3.4.2 loop do 语句

语法

```
loop do:
    …
    ;
```

loop do 语句表示执行内部语句的无限循环，直到内部执行除 pass 以外任何形式的跳转语句才能退出循环。

备注

所有条件语句与迭代语句在 Tripod 语言内部实际上都是作为表达式实现的，只是单独提取出来可以起

到跟其他语言中这类语句相似的效果而已。

这些表达式的结果往往是这些语句中最后一个计算的表达式的结果。

12.3.5 子结构语句

语法

```
subject mySubject:
    …
    ;
```

块语句用来定义一个代码块，goto 语句可以跳转到这个代码块的位置，如果这个代码块内部未执行 break 或者 exit，goto 语句将会自动归位，回到 goto 语句的下一条语句。如果这个代码块内部包含这两种跳转语句，则 goto 进入后当控制流遇到这两个语句时，将会自动跳出。

subject 中的 break 语句和 exit 语句不影响以正常方式进入语句的控制流。

12.3.6 静态块语句

语法

```
static myStatic:
    …
    ;
```

静态块语句用来指定一些静态类型的变量部分，这部分由同种函数，对象共享，修改一次，将会影响到其他所有使用静态块的对象，条件语句和迭代语句也能使用静态块。

第 13 章　函　数

函数是程序中可以在任意位置，任意次数执行的代码块。这些代码块被称为函数体。

本章将讨论函数的用法，并描述函数的声明和定义、形式参数和返回类型以及函数调用的细节，以及函数的重载方面的问题。

13.1　函数声明

函数声明将函数名标识符与代码块相关联。然后可以使用这个函数名来调用这个函数。代码块后的定义实现了该代码块中的操作。

这个代码块可以选择是否返回一个结果。函数的结果类型可以是任何先前声明的类型。

示例

```
void DoSomething(const String para):
    println("Got paramenter: "; para)
    println("Parameter in upper case: ", Upper(para))
    ;
```

13.2　返回类型

函数返回指的是函数调用在完成时，函数具有一个值的能力，以便可以在表达式中使用该值。

函数可以被定义为返回任何类型的值，函数可以返回聚合对象，这可能会由于平台的限制而无法实现，这时候可以使用返回聚合对象的指针或者聚合对象的引用。

示例

函数的值可以通过三种方式返回:

```
// 函数结束后，最后一个计算的表达式的值
int myFunction():
    12
    ;
// 函数变量作为函数的返回值。
// 一个依序执行的函数，如果设置了函数变量，则函数执行结束后，函数的返回值就是函数变量的值
// 一个使用 return 语句而不返回任何值的函数，返回后函数的值是函数变量的值。
int myFunction():
    myFunction = 12
    return
    ;
```

```
// 通过 return 语句显式返回一个值，这会覆盖函数变量的值。
int myFunction():
    return 12
    ;
```

函数再返回时可以返回函数变量、声明或定义，用来返回一个函数。无论函数以怎样的方式被返回，函数返回函数的类型要与外部函数的返回类型一致，且必须省略内部函数的类型限定符。

示例

```
// 返回一个函数变量，这个函数在执行完外层函数后后可以进行连环调用
int myFunction():
    int innerFunction():
        return 12
        ;
    return innerFunction
    ;
// 返回一个函数声明，这个函数必须在它声明的作用域内实现
// 这种情况下，接受参数为空的函数必须显式写出 void
int myFunction():
    return innerFunction(void)
    int innerFunction(void):
        return 12
        ;
    ;
// 返回一个函数定义，这个函数不可以匿名，纵使它的名字并没有意义
var int myFunction():
    return innerFunction(void):
        return int +
        ;
    ;
```

13.3 形式参数

函数可以在调用时以变量和对象的形式传递信息。在函数声明和调用的上下文中，这些变量和对象称为参数。

参数必须在该函数的形式参数列表中声明，参数列表是过程在引用传递给它的参数时将使用的一个或多个名称和类型的列表。参数列表用括号括起来。这些参数称为形式参数，这些参数会被隐式的在函数内部进行声明，将外部变量或对象传入时，会设置函数内部这些隐式声明的参数。参数可以像任何其他变量或对象一样使用。

形式参数被看成是一个指定（或重写）类型的局部变量，并且复制了对应的实际参数传递给函数的值。对形参进行赋值是允许的，但这种赋值只会修改局部的参数值，而不是修改调用函数中的实际参数。

在函数调用时，形式参数必须和实际参数的类型保持一致，或者是可以自动转换的相互兼容的类型，Tripod 内部实现是一门强类型的语言，虽然它存在着部分弱类型的功能。

示例

```
void Procedure(String s; int n):
    print("The parameters have the values: "; s; " and "; n)
    ;

// 简单类型对象做为形式参数
void Procedure(int param = 1):
    param *= 2
    print("The parameter 'param' = "; param)
    ;
// 数组类型对象作为形式参数，数组在声明时可以指定具体的长度，也可以不指定长度
// 指定长度则要求传入的数组必须匹配指定的长度
// 数组的阶数必须和实际数组相匹配
// 数组只可以从最后阶开始向前省略维度长度，且要求每阶的所有维度均省略长度
real Average(int Array[5][]):
    int i = 0, item_count = 0, count = 0
    int j = 0
    real
    loop if i < 5:
        item_count = LengthOf(Array[i])
        loop if j < count:
            sum += Array[i][j]
            j++
            ;
        count += item_count
        item_count = 0
        i++
        ;
    return sum / count
    ;
```

多个形式参数之间使用分号分隔每个类型的形式参数，使用时按参数声明的顺序书写。

13.4 参数传递

Tripod 支持两种主要的函数参数传递方式：传值调用和传引用调用。所有的参数都可以有自己的默认值，对于传递函数来说，形式参数函数可能会有自己的定义。

默认情况下，函数参数总是按值进行传递。

13.4.1 按值传递

按值传递的参数实际上不会将其自身对象之间传递给函数，它会将自身拷贝自对象中相应对象处，传递的是原始参数的副本。

按值传递参数的行为允许函数修改这个副本，而保持原始变量和对象不变。

对于函数参数，可以指定它们的默认值，如果函数调用省略了参数，则将默认参数传递给函数。

示例

```
val myConst = 20
void myRealFunc(int i = myConst):
    println("Function received: "; i)
    ;
```

13.4.2 按引用传递

与按值传递参数的方式不同，通过引用传递给函数的参数本身不会复制，它会传递变量或对象的别名，通过这个别名可以直接修改外部的变量或对象。

引用类似于变量或对象的别名。无论何时引用一个引用参数，都指的是以它为别名的实际变量或对象。换句话说，您可以将函数的引用参数视为传递给它的真实对象；对引用参数所做的任何更改实际上都是对它所代表的参数的更改。

要指定通过引用传递参数，需要在参数声明前加上引用限定符，引用限定符指定了引用的实际参数是否可更改。

对于 const 常引用函数来说，这意味着这个所引用的这个函数不能修改外部对象。

示例

```
byVar Procedure(var int param):
    // 将 param 的引用传递进来，处理后又传递出去
    param *= 2
    ;
```

对于返回引用的函数，如果返回的是原始类型，则必须在返回类型前加上返回引用限定符，如果是聚合类型，则默认以 **byRef** 类型返回。返回引用限定符与一般的引用限定符不同，一般的引用限定符只声明名字，返回引用限定符则明确定义函数要返回一个引用。

13.5 函数重载

函数重载意味着可以多次定义同名的函数，但每次都使用不同的形式参数列表。参数列表必须至少在其元素类型之一中有所不同。当编译器遇到函数调用时，它将查看函数参数以决定它应该调用哪个定义的函数。当必须为不同类型定义相同的功能时，以及函数的连续调用时，这可能非常用。

示例

```
…
void Dec(var obyte i, decrement)
void Dec(var obyte i)
void Dec(var byte i; var obyte decrement)
void Dec(var byte i)
…
```

当编译器遇到对 Dec 函数的调用时，它将首先搜索它应该使用哪个函数。因此，它检查函数调用中的参数，并查看是否存在与指定参数列表匹配的函数定义。如果编译器找到这样的函数，则会向该函数插入一个调用。如果未找到此类函数，则会生成编译器错误。

13.6 前置声明

函数可以只声明而不定义,。函数的定义可以在模块之后进行，这种声明方式称之为前置声明。该函数可以在前置声明之后使用，就像它已经实现一样。

示例

```
void First(obyte n)
void Second():
    println("In second. Calling first...")
    First(0)
    ;
void First(obyte n)
    println("First received: "; n)
    ;
```

函数只可以进行前置声明一次。在其他模块导入另一个模块时，这个模块不可以再声明这个函数，但可以重新定义这个函数。

前置声明对于函数递归非常有用.

13.7 函数调用

函数调用有三种方式：连续调用，连环调用和常规调用。常规调用方式与其它编程语言无异，一个函数将被单独地进行调用；连环连续调用使得一个函数可以连续调用多次，省略繁琐的函数名的书写，连续调用函数的返回值是最后一次函数成功执行后的返回值；连环调用表明一个函数内部的返回函数依序调用，吸收参数表，连环调用函数的返回值是最后一次函数成功执行后的返回值。

示例

```
// 一个函数
int meFunction(int x):
    return myFunction(int y):
        return ourFunction(int z):
            return z
            ;
        ;
    ;
// 常规调用
meFunction(0)
// 连续调用
meFunction(0; 1; 2)     // 等价于 meFunction(0) meFunction(1) meFunction(2)
// 连环调用
meFuntion(0)(1)(2)      // 等价于 meFunction(0) myFunction(1) ourFunction(3)
```

第 14 章　程序　模块

Tripod 程序的基本组成单位是模块，每个程序由不同的模块组成，程序可能只来自一个模块，也可能有成百上千个。模块可以被编译成可执行代码或者解释执行。模块是包含 Tripod 语言语句的文本文件，文本文件的编码并不强制指定，由编译器设计者自定。

本章将介绍一个完整的 Tripod 程序的构成，模块的使用，以及作用域的规则。

14.1　模块函数

模块函数是指在模块中，存在一个跟模块名同名的函数，这个函数可以被重载，具体调用哪个重载函数取决于编译器设计的约定，模块函数是一个可执行模块的入口点。Tripod 不具有一个固定的程序入口函数，相反，任何被编译器最先调用的模块的模块函数就是程序的入口点。

模块函数可以位于模块顶层，也可以位于模块对象和模块类型内部。模块对象与模块类型都是与模块名同名的对象或类型，它们可以接受泛型，这取决于编译器对模块的执行约定。

示例

```
// Moudle main.t
void main(int args, const byte * argv[]):
    return
    ;
// Module Entry.t
class Entry<T>:
    void Entry(int args, const byte * argv[]):
        return
        ;
    ;
// Moudule Init.t
object Init<T>:
    void Init(int args, const byte argv[][]):
        return
        ;
    ;
```

请注意：main 函数不是程序的唯一的入口点，其他模块函数均可作为程序的入口点，这与 C 系语言有很大的区别，请特别留意。

14.2　导入模块

一个模块可以导入其他模块，通过导入语句实现。

一个模块可以通过导入语句导入另一个模块，被导入模块内部的所有内容将会展开到当前模块中。被导入模块将会依序执行内部的语句会依序进行初始化。

所有的模块按照目录层级组织起来，为了导入一个确定的模块，可以使用目录名层级加上模块的名称进行导入。默认情况下，不携带目录名，直接使用模块名导入的是当前目录下的模块。

如果最后的模块名用星号替代，意味着导入这个目录下的所有模块。

语法

```
import Directory. Module
```

示例

```
import io.*
import net.socket
```

import 语句不具有遗留性。假设存在这样 A, B, C 三个模块，B 模块导入 A 模块，A 模块中的所有数据被导入进了 B 模块中；C 模块导入 B 模块，B 模块中的所有数据被导入进了 C 模块中；一旦发生了这种情况，C 模块对于 A 模块中的所有数据均是不可见的，C 模块只能见到 B 模块中的数据，import 表达式不会使 A 模块中的数据遗留到 B 模块中。

import 语句必须位于模块中所有语句之前。

14.3 模块执行

一个模块在被编译器调用执行时，模块中的所有语句将会执行一次，这称之为模块初始化。如果模块函数位于模块顶层，则在模块初始化后直接执行这个模块函数，开始程序真正的执行流程；如果模块函数位于模块类型之内，则会首先对这个类型进行实例化，这称之为模块类型初始化，最后调用模块函数，程序开始运行；如果模块函数位于模块对象之类，则在模块对象初始化后，调用模块函数。

示例

```
// Module Entry.t
class Entry<T>:
    void Entry(int args, const byte * argv[]):
        println("Do it!")
        return
        ;
    println("Module initialization completed!")
    println("The program begins to run officially!")
    ;
//  Compiler directive: Tripod Entry.t Entry<int>
//  首先 Entry 类会进行隐式实例化，之后将调用 Entry 函数
//  Entry 函数执行后立即返回，模块退出，程序退出
```

14.4 代码块

模块中所有的复合语句和块语句都会构成代码块，一个代码块可以由各种程序实体组成。块可以以某

些方式嵌套，即，过程或函数声明本身可以具有块以及子块。

这意味着程序的代码块可以无限制的嵌套，迭代语句内部甚至可以声明变量，函数，在每次函数重复执行时，这些变量和函数都可以进行初始化。

14.5 作用域

一个标识符从它的声明点开始，直到声明发生的块的结束。标识符的有效范围也称之为标识符的作用域。标识符具体的作用域取决于它的定义方式。

同一个作用域中，不允许具有同名标识符的不同声明和定义。

导入语句可能使得一个作用域中出现同名不同定义的标识符，这时需要通过指定标识符在模块或目录中的完整名字来进行区分，或者区分后进行简单引用。

13.4.1 块作用域

在块中声明的标识符，具有块作用域，它的有效范围持续到当前块的末尾。如果块包含第二个块，其中标识符被重新声明，则在该块内，第二个声明将是有效的。离开内部区块后，第一个声明再次有效。

示例

```
// Module Demo.t
real x
void newDeclaration:
    int x
    x = 1.234                 // 出错，无法将宽类型自动转换成窄类型
    x = 10                    // 成功
    ;
x = 2.468                     // x 再次可用
```

13.4.2 模块作用域

模块顶层的所有标识符从声明点开始有效，直到模块结束。模块顶层标识符在模块内部的类型，对象，函数内都是可用的。

利用 import 语句导入的模块中的标识符，对于当前模块也是有效的。

附录 A

附录 A 提供了 Tripod 语言的词法元素表，用正则表达式描述。

附录 B 提供了 Tripod 语言的文法表，使用 BNF 描述。

（仍有不完善的地方，仅供参考）

语言中的终结符使用小写字母开头的名称并用粗体进行表示。语言中的非终结符使用大写字母开头的名称表示。非终结符的定义由非终结符的名称和后面的一个冒号引入，如果它具有几种候选形式，则分在不同的行显示。

定义的可选成分使用下标 $_{opt}$ 表示。

“one of”表示下面一行或多行中的每个符号都可以是候选定义。

```
D                       =  [0-9]
L                       =  [a-zA-Z]
H                       =  [a-fA-F0-9]
E                       =  ([Ee][+-]?{D}+)
FS                      =  (s|S|d|D|r|R)
LS                      =  ((u|U)|(u|U)?(l|L|ll|LL))
```

上表演示了词法元素的基本构成单位，包括数字，字符以及字面量的后缀。

```
comment                 =  "/*"
                        =  "//"[^\n]
```

注释部分提供注释的起始标记，余下的需要依靠语义分析器进行处理。

```
identifier              =  {L}({L}|{D})*
```

类型名和变量名都依赖于标识符，当编译器读到一个标识符的时候，需要检查它是一个类型还是一个函数或者是一个变量，这交由语义分析器完成。

```
literal                 =  0[xX]{H}+{IS}?
                           0[0-7]*{IS}?
                           [1-9]{D}*{IS}?
                           '(\\.|[^\\'\n])+'
                           {D}+{E}{FS}?
                           {D}*"."{D}+{E}?{FS}?
                           {D}+"."{D}*{E}?{FS}?
                           \"(\\.|[^\\"\n])*\"
```

字面量包括整数，实数和字符串。

附录 B

[B.1……完整类型]

声明引用类型是最完整的类型限定符，它包括引用限定符，一般的类型限定符，之所以称它们为限定符的原因是，它们并不对类型进行指定的作用，一个变量不需要类型也可以进行赋值，初始化，函数不需要返回类型也可以进行定义，类似于 basic，python 等语言。

```
Decl-Reference-Type:
        Reference-Type Decl-Type_opt
Decl-Type:
        Type Aggergation-Or-Pointer-Or-Operator-Type-Seq_opt
```

示例

```
var int
const double
val Array *
```

[B.2……类型]

Type 中包括的有简单类型，泛型类型，抽象类型，实体类型和一些辅助的类型关键字。

```
Type:
        Simple-Type
        Generic-Typ
        Abstract-Type
        Entity-Type
        type
        entity
        virt
```

示例

```
// entity 关键字用来定义一般的实体类型对象，即不可以其为类型，定义对象的对象，如 object。
// entity 对这些实体对象起着一个默认初始化的作用。
int x = entity
real function() = entity
entity Box = object

// type 关键字用于声明一个新定义的类型，或者建立一个类型。
int [5] intArray = type
intArray x
```

```
type ArrayList<T> = class:
    …
;
```

[B.3……聚合或指针或操作符类型]

```
Aggergation-Or-Pointer-Or-Operator-Type-Seq:
        Aggergation-Or-Pointer-Type
        Aggergation-Or-Pointer-Type-Seq Aggergation-Or-Pointer-Type
Aggergation-Or-Pointer-Or-Operator-Type:
        Aggregation-Type
        Pointer-Type
        Operator-Type
Aggregation-Type:
        Array-Type
        Action-Type
Pointer-Type:
        *
```

示例

```
// 聚合类型的声明方式意味着指针、数组和函数可以把索引声明和函数参数表在声明时前置。
// 对于前置的数组索引和前置的函数参数表，必须按相应的语法格式使用。
int***[3,3](int i, j) arrayFunction

// 上述声明的理解方式如下:
arrayFunction                                   的类型:    int***[3,3](int i, j)
(int i,j)arrayFunction                          的类型:    int***[3,3]
[3,3](int i, j) arrayFunction                   的类型:    int***
***[3,3](int i, j) arrayFunction                的类型:    int
```

[B.4……原始类型]

```
Simple-Type;
        byte
        dbyte
        qbyte
        obyte
        int
        ubyte
        udbyte
        uqbyte
        uobyte
        single
        double
```

```
real
byteBool
wordBool
longBool
qwordBool
variety
enum
void
```

[B.5……泛型类型]

```
Generic-Type:
        Type-Name Actual-Type-Parameter opt
Actual-Type-Parameter:
        < Actual-Types >
Actual-Types:
        Actual-Type
        Actual-Types , Actual-Type
Actual-Type:
        Decl-Type
```

[B.6……数组类型]

```
Array-Type:
        [ Array-Type-Parameters opt ]
Array-Type-Parameters:
        Array-Type-Parameter
        Array-Type-Parameters , Array-Type-Parameter
Array-Type-Parameter:
        Constant-Expr
```

[B.7……函数类型]

```
Action-Type:
        ( Action-Type-Parameter-List opt )
Action-Type-Parameter-List:
        Action-Type-Parameter
Action-Type-Parameter-List ; Action-Type-Parameter
Action-Type-Parameter:
        Declaration
```

[B.8……抽象和实体类型]

```
Abstract-Type:
        unify
        class Super opt
Entity-Type:
        object Super opt
Super:
        ( Decl-Types )
```

[B.9……引用类型]

```
Reference-Type:
        var
        ref
        const
        val
        byVar
        byRef
        byConst
        byVal
```

[B.10……声明]

```
Declaration:
        Decl-Reference-Type opt Init-Declarators
Init-Declarators:
        Init-Declarator
        Init-Declarators , Init-Declarator
Init-Declarator:
        Declarator Iniitalizer opt
```

示例

```
// 这种声明方式允许不使用类型限定符，直接定义一个变量，函数。
x = 7
getCode(int key):
    return key * 7
    ;
getCode(x)
```

[B.11……声明器]

```
Declarator:
        identifier
        Operator-Action-id
```

```
        Generic-Type
        Declarator Aggregation-Type
        Declarator Type-Parameter
        Declarator . Declarator
Operator-Action-id:
        Operator Operator-ID
Operator:
        operator
Operator-ID:one of
        +    -    *    /    %    ^    &    |    ~
        Not    =    <    >    +=    -=    *=    /=    %=
        ^=    &=    |=    >>>    <<    >>    >>>=    >>=    <<=    ==    <>
        <=    >=    And    Or    ++    --    []
Type-Parameter
        < Template-Type-Parameter >
Template-Type-Parameter:
        Template-Type
        Template-Type-Parameter ; Template-Type
Template-Type:
        identifier Implement_opt
Implement:
        via Types
Types:
        Type
        Types , Type
```

[B.12……初始化器]

值初始化用来初始化一个变量的值或者某个类型，定义初始化用来初始化一个函数或者类型的定义。双重初始化的形式来源于 pascal。

```
Iniitalizer:
        Value-Initializer Extended-Initializer_opt
Value-Initializer:
        = Value-Initializer-Expr
Value-Initializer-Exprs:
        Value-Initializer-Expr
        Value-Initializer-Exprs , Value-Initializer-Expr
Value-Initializer-Expr:
        Assignment-Expr
        ( Value-Initializer-Exprs )
        Type
Extended-Initializer:
        Expand-Statement
```

[B.13……原始表达式]

原始表达式是构成一个表达式的最基本的语法单元，它包括数组、函数和指针，以及它们相互嵌套产生的类型所构成的表达式。

```
Primary-Expr:
        Algebra
        ( Expression )
        literal
Postfix-Expr:
        Primary-Expr
        Postfix-Expr Array-Access
Postfix-Expr Action-Invocation
        Decl-Reference-Type ( Argument-Expr )
        Postfix-Expr . Algebra
Array-Access:
        [ Argument-Exprs ]
Action-Invocation:
        ( Argument-Exprs-List )
Argument-Exprs-List:
        Argument-Exprs
        Argument-Exprs-List ; Argument-Exprs
Argument-Exprs:
        Argument-Expr
        Argument-Exprs , Argument-Expr
Argument-Expr:
        Assignment-Expr
Prefix-Expr:
        Postfix-Expr
Array-Access Prefix-Expr
        Action-Invocation Prefix-Expr
        * Prefix-Expr
```

示例

```
// 函数数组的前缀声明方式决定了原始表达式的使用方式
// 之前声明的一个函数
int***[3,3](int i, j) arrayFunction

// 它的调用方式
(4, 5)arrayFunction
```

[B.14……基础表达式]

```
Rear-Expr:
        Postfix-Expr
        Rear-Expr ++
        Rear-Expr --
Front-Expr:
        Rear-Expr
        + Front-Expr
        - Front-Expr
        ++ Front-Expr
        -- Front-Expr
        not Front-Expr
        ~ Front-Expr
        @ Front-Expr
```

[B.15……算术表达式]

```
Mul-Expr:
        Front-Expr
        Mul-Expr * Front-Expr
        Mul-Expr / Front-Expr
        Mul-Expr \ Front-Expr
        Mul-Expr % Front-Expr
Add-Expr:
        Mul-Expr
        Add-Expr + Mul-Expr
        Add-Expr - Mul-Expr
Shift-Expr:
        Add-Expr
        Shift-Expr >> Add-Expr
        Shift-Expr << Add-Expr
        Shift-Expr >>> Add-Expr
Rel-Expr:
        Shift-Expr
        Rel-Expr < Shift-Expr
        Rel-Expr > Shift-Expr
        Rel-Expr >= Shift-Expr
        Rel-Expr <= Shift-Expr
Equ-Expr:
        Rel-Expr
        Equ-Expr == Rel-Expr
        Equ-Expr <> Rel-Expr
```

[B.16……位运算表达式]

```
Bit-And-Expr:
        Equ-Expr
        Bit-And-Expr & Equ-Expr
Bit-Xor-Expr:
        Bit-And-Expr ^ Bit-Xor-Expr
Bit-Or-Expr:
        Bit-Xor-Expr | Bit-Or-Expr
```

[B.17……逻辑运算表达式]

```
Log-And-Expr:
        Bit-Or-Expr
Log-And-Expr and Bit-Or-Expr
Log-Or-Expr:
        Log-And-Expr
        Log-Or-Expr or Log-And-Expr
```

[B.18……表达式]

```
Expression:
        Assignment-Expr
        Decl-Type +
        Algebra -
Assignment-Expr:
        Log-Or-Expr
        letopt Assignment-Expr Assignment-Op Log-Or-Expr
Assignment-Op: one of
        =  =>  *=  /=  \=  %=   +=  -=  >>=  <<=  >>>=   &=  ^=  |=
Constant-Expr:
        Assignment-Expr
```

[B.19……语句]

```
Expand-Statement:
        : Statement-Seq ;
Statement-Seq:
        Statement
        Statement-Seq LineTerminator Statement
Statement:
        Block-Statement
        Expr-Statement
```

Selection-Statement
Iteration-Statement
Jump-Statement
Decl-Statement
Import-Statement

Block-Statement:
Block-Identifier Expand-Statement

Block-Identifier:
Subject
Static
Public
Protect
Private

Expr-Statement:
Expression$_{opt}$

Iteration-Statement:
loop *Selection-Statement*

Selection-Statement:
if *Expression Expand-Statement Else-Clause*$_{opt}$

Else-Clause:
else *Statement*

Jump-Statement:
goto *Prefix-Expr*
return *Expression*
return *Init-Declarator*
return
pass
break
exit

Decl-Statement:
Declaration

Import-Statement:
import *Directory-Module-Name*

Directory-Module-Name
identifer
Directory-Module-Name . *identifier*
Directory-Module-Name . *