D 语言编程

参考手册

(中文版 中册)

张雪平 译著

2009.05

版权声明

- 本电子书纯为宣传和推广 D 语言而作，不具任何商业目的。
- 本译稿可供大家免费阅读、在网上自由传播，但切勿擅自更改或用作任何形式的商业用途。如有必要，请事先征得译作者的同意。
- 请大家重视知识产权的保护、尊重译作者的劳动成果。
- 本译稿的原稿来自 DMD 官方手册，在翻译过程中部分有参考前人的译作。
- 原稿版权归属 Digital Mars 所有，译作者拥有本译稿所有版权。

前 言

本电子书的制作源自 Digital Mars 官方的 DMD 手册，基于 DMD 2.029。由于原手册是 html 文档，因此在整个版面风格上还不尽统一，等 DMD 的发行版相对稳定后，再来统一调整、美化版面。由于 DMD 2.0 本身还在不断的发展，相应的手册也随时在更新，相对 DMD 1.0 手册，内容的确新增了很多，希望中文手册能够帮助大家更好地学习 DMD 2.0！

拖了那么久，总算是再次更新了一下。如果您在阅读的过程中发现有任何错漏，欢迎来信指正。

快毕业了，又快工作了。希望以后能有更多时间来维护本译稿。

本电子书的下载地址：http://bitworld.ys168.com，http://www.dlang.net

QQ 交流群：47514066（满员），51879343，8399665

感 谢

- 感谢 Digital Mars 无私提供英文原稿
- 感谢 Walter Bright 发明了 D 语言，让我有机会可以翻译这份文档
- 感谢 www.dnaic.com 提供了部分翻译参考，详见：
- http://www.dnaic.com/d/doc/d/index.html
- 感谢 www.OpenOffice.org 提供的 OpenOffice.org，这是一套功能强大免费的办公软件
- 感谢在我身边不断支持我的家人、朋友、同学以及各位热心的网友
- 感谢自己仍保持了热情且继续翻译、更新下去

作者简介

张雪平，西南石油大学研究生院 2006 级，模式识别与智能系统专业

- QQ: 85976988 ICQ: 56-1981-07
- E-Mail: zxpmyth@yahoo.com.cn
- Phone: 13688099543

附言：

本作品的完善需要您无私的帮助与慷慨捐助。

捐款账号：9558804402115568648(中国工商银行)

更新记录

☑2008.10 初始版本
☑2009.05 少量新章节，更多错误更正、补充翻译，改善章节编号

目　录

第四篇 语言参考

第 1 章 词法

在 D 中，词法(lexical)分析独立于语法(syntax)分析和语义(semantic)分析。词法分析器是将源文件分割成特征符。词法描述的是特征符是些什么。D 的词法设计适合高速扫描，它拥有最小的特例集；由于只有一遍翻译，使得编写一个正确的扫描程序相当容易。对于熟悉 C 和 C++ 的人来说，特征符也很容易识别。

1.1 编译的阶段

编译被分为多个阶段。每个阶段都不依赖于后继的阶段。例如，扫描程序不依赖于语义分析程序。这种分离使语法制导编辑器等语言工具相对容易构造。这也使通过将其存储为‘符号’形式来压缩 D 源码成为可能。

1. **源文件字符集(source character set)**
   先检查源文件使用的字符集是什么，然后装载相应的扫描器(scanner)。允许的格式是 ASCII 或 UTF。
2. **脚本行**
   如果第一行以 #! 开头，那么第一行将被忽略。
3. **词法分析(lexical analysis)**
   源文件被分割为特征符序列。特殊特征符会被替换成其它的特征符。特殊特征符序列先会被处理，接着被移除。
4. **语法分析(syntax analysis)**
   特征符序列会被解析为语法树。
5. **语义分析(semantic analysis)**
   通过遍历语法树来声明变量、载入符号表、分配类型并在大体上判断程序的意图。
6. **优化(optimization)**
   这步是可选的——它试着在保持语义等价的同时重写程序，只是生成一个运行速度更快的版本。
7. **代码生成(code generation)**
   根据目标架构选取相应的指令来实现程序的语义。通常的结果是生成一个目标文件，方便用于连接器的输入。

1.2 源文本(Source Text)

D 源文本可以是下面各种形式之一：

- ASCII
- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-32BE
- UTF-32LE

UTF-8 是传统的“7-位 ASCII”的超集(superset)。源文本的开头可以是下列 UTF 字节序标志(BOM)之一：

UTF 字节序标记

格式	BOM(字节序标志)
UTF-8	EF BB BF
UTF-16BE	FE FF
UTF-16LE	FF FE
UTF-32BE	00 00 FE FF
UTF-32LE	FF FE 00 00
ASCII	no BOM

如果源文件不是以 BOM 开头，那么第一个字符必须小于或等于 U0000007F。

在 D 中没有“双连字(digraphs)”或者“三连字(trigraphs)”。

源文本的源表示(source representation)被解码成 Unicode *字符*。这些 *字符* 又被进一步分成： 空白符、行尾符、注释符、特殊特征符序列、特征符，以及跟在前面那些字符之后的文尾符。

应用贪心算法（即：词法分析器每次都尽量生成一个最长的特征符）将源代码文本分割成很多特征符。如：“>>”是一个右移符号，而不是两个大于符号。

1.3 文尾符(End of File)

```
文尾符:
        文件的物理结束符
        \u0000
        \u001A
```

在遇到上述符号之一时就认为文件终止。

1.4 行尾符(End of Line)

```
行尾符:
        \u000D
        \u000A
        \u000D \u000A
        文尾符
```

不允许用反斜线来将一行分为多行，每行的长度也没有限制。

1.5 空白符(White Space)

```
空白:
        空格
        空格 空白符
```

```
空格:
        \u0020
        \u0009
        \u000B
        \u000C
```

1.6 注释(Comments)

```
注释:
        /* 字符 */
        // 字符 行尾符
        嵌套注释块

字符:
        单个字符
        单个字符 多个字符

嵌套注释块:
        /+ 嵌套注释块字符 +/

嵌套注释块字符:
        嵌套注释块单个字符
        嵌套注释块单个字符 嵌套注释块多个字符

嵌套注释块单个字符:
        单个字符
        嵌套注释块
```

D 语言有三种注释：

1. 可以跨越多行，但是不能嵌套的块注释。
2. 在行尾结束的单行注释。
3. 可以跨越多行并且可以嵌套的嵌套注释。

字符串和注释的内容不会被分析成特征符。因此，在一个字符串内注释起始符并不会引导一个注释，并且在注释内的字符串分割符也不会影响对注释结尾符以及嵌套的"/+"注释起始符的识别。为了避免"/+"出现在"/+"注释中，在注释里的注释起始符将会被忽略。

```
a = /+ // +/ 1;          // 解析为 'a = 1;'
a = /+ "+/" +/ 1";       // 解析为 'a = " +/ 1";'
a = /+ /* +/ */ 3;       // 解析为 'a = */ 3;'
```

注释不能被用作连接符，例如：abc/**/def 是两个特征符：abc 和 def，而不是只有一个 abcdef。

1.7 特征符(Tokens)

```
特征符:
        标识符
        单个字符串文字
        字符文字
        整数文字
```

浮点数文字
关键字
/
/=
.
..
...
&
&=
&&
|
|=
||
-
-=
--
+
+=
++
<
<=
<<
<<=
<>
<>=
>
>=
>>=
>>>=
>>
>>>
!
!=
!<>
!<>=
!<
!<=
!>
!>=
(
)
[
]
{
}
?
,

```
        ;
        :
        $
        =
        ==
        *
        *=
        %
        %=
        ^
        ^=
        ~
        ~=
```

1.8 标识符(Identifiers)

标识符:
        标志符起始符
        标志符起始符 多个标志符字符

多个标志符字符:
        单个标志符字符
        单个标志符字符 多个标志符字符

标志符起始符:

        _
        字母
        通用字母

单个标志符字符:
        标志符起始符
        0
        非零数字

标识符的起始符由一个字母、_ 或通用字母组成，紧跟的字符可以是字母、_、数字或通用字母。通用字母在 ISO/IEC 9899:1999(E)的附录 D（即 C99 标准）中有定义。标识符长度任意，并且区分大小写。以'__ (两个下划线)'开头的标识符被保留了。

1.9 字符串字法(String Literals)

字符串文字:
        所见即所得字符串
        另一种所见即所得字符串
        双引号字符串

        十六进制字符串
        分隔串
        特征符字符串

所见即所得字符串:
        r" *多个所见即所得字符* " *后缀*可选的

另一种所见即所得字符串:
        ` *所见即所得字符* ` *后缀*可选的

多个所见即所得字符:
        单个所见即所得字符
        单个所见即所得字符 多个所见即所得字符

所见即所得字符:
        单个字符
        行尾符

双引号字符串:
        " *多个双引号字符* " *后缀*可选的

多个双引号字符:
        单个双引号字符
        单个双引号字符 多个双引号字符

单个双引号字符:
        单个字符
        转义序列
        行尾符

转义序列:
        \'
        \"
        \?
        ****
        \a
        \b
        \f
        \n
        \r
        \t
        \v
        **** *文尾符*
        \x *单个十六进制数字 单个十六进制数字*
        **** *单个八进制数字*
        **** *单个八进制数字 单个八进制数字*
        **** *单个八进制数字 单个八进制数字 单个八进制数字*
        \u *单个十六进制数字 单个十六进制数字 单个十六进制数字 单个十六进制数字*
        \U *单个十六进制数字 单个十六进制数字 单个十六进制数字 单个十六进制数字 单个十六进制数字 单个十六进制数字 单个十六进制数字 单个十六进制数字*
        \& *命名的字符实体* **;**

十六进制字符串:
        x" *多个十六进制串字符* " *后缀*可选的

```
多个十六进制串字符:
        单个十六进制串字符
        单个十六进制串字符 多个十六进制串字符

单个十六进制串字符:
        单个十六进制数字
        空白
        行尾符

后缀:
        c
        w
        d

分隔串:
        q" 分隔符 所见即所得字符 配对分隔符 "

特征符字符串:
        q{ 特征符 }
```

字符串文字可以是一个双引号字符串、一个引起来的所见即所得字符串、一个转义序列、分隔字符串、特征符字符串或者一个十六进制字符串。

1.9.1 所见即所得字符串

所见即所得字符串需要使用‘r"’和‘"’封闭引起来。所有位于‘r"’和‘"’之间的字符都是字符串的一部分，不过对于 *行尾*，它会被当作一个单一的‘\n’字符。在‘r" "’中没有转义序列：

```
r"hello"
r"c:\root\foo.exe"
r"ab\n"                    // 由四个字符组成的字符串：'a'、'b'、'\'、'n'
```

所见即所得字符串还有另外一种形式，即使用反引号‘`’。由于‘`’字符并不是所有的键盘上都有，而且有时在屏幕上难以同另一个常用的字符‘'’区分。因此，还是尽量少用‘`’为好；不过当用在由‘"’引起来的字符串中时，它就变得很有用了。

```
`hello`
`c:\root\foo.exe`
`ab\n`                     // 由四个字符组成的字符串：'a'、'b'、'\'、'n'
```

1.9.2 双引号字符串

双引号字符串指的是用‘""’引起来的字符串。在这种串中，可以嵌入由标准的‘\特征符’构成的嵌入转义序列。*行尾* 则被视作一个单一的‘\n’字符。

```
"hello"
"c:\\root\\foo.exe"
"ab\n"                     // 由三个字符组成的字符串：'a'、'b'和一个换行符
"ab
"                          // 由三个字符组成的字符串：'a'、'b'和一个换行符
```

1.9.3 十六进制字符串

十六进制字符串使用十六进制数据构造字符串。十六进制数据不需要组成有效的 UTF 字符。

```
x"0A"                    // 等同于 "\x0A"
x"00 FBCD 32FD 0A"       // 等同于 "\x00\xFB\xCD\x32\xFD\x0A"
```

空白和换行符都会被忽略，因此格式化十六进制数据就很方便了。十六进制字符的个数必须是 2 的倍数。

相邻的字符串应该用‘~’运算符连接，或者仅仅并列一起即可：

```
"hello " ~ "world" ~ \n // 构成字符串：'h','e','l','l','o',' ',
                        // 'w','o','r','l','d',换行符
```

下面的形式都是等价的：

```
"ab" "c"
r"ab" r"c"
r"a" "bc"
"a" ~ "b" ~ "c"
\x61"bc"
```

可选的 *后缀* 字符产生了特定类型的字符串，而且还需要由上下文来确定。这在字符类型不能清楚地确定时相当有用，例如，基于字符串类型的重载。后缀字符的种类有：

字符串文字后缀字符

后缀	类型
c	char[ ]
w	wchar[ ]
d	dchar[ ]

```
"hello"c          // char[]
"hello"w          // wchar[]
"hello"d          // dchar[]
```

字符串文字是只读的。对字符串文字的写入并不能总是被检测到，而写入的话会引起“未定义的行为(undefined behavior)”。

1.9.4 分隔串

分隔串(Delimited string)使用不同形式的分割符。分割符（无论是字符还是标识符）都必须紧跟在符号"之后，之间不能有任何空格。分割终止符，也必须紧跟在第二个"符号之前，之间不能有任何空格。*嵌套分割符*可以嵌套，并且可以是下例字符之一：

嵌套分隔符

分隔符	配对分隔符
[	]
(	)
<	>
{	}

```
q"(foo(xxx))"    // "foo(xxx)"
q"[foo{]"        // "foo{"
```

如果分隔符是一个标识符，则该标识符后必须立即紧跟一个新行，而且配对分隔符跟该行起始处的标识符相同：

```
writefln(q"EOS
This
is a multi-line
heredoc string
EOS"
);
```

紧跟着起始标识符的换行符不是该字符串的一部分，不过在结束标识符之前的换行符却是该串的一部分。

另外，配对分隔符等同于分隔符：

```
q"/foo]/"        // "foo]"
q"/abc/def/"     // 错误
```

1.9.5 特征符字符串

特征符字符串(Token string) 由符号“**q{**”起始，功能符“**}**”做为结尾。两者之间必须是有效的 D 特征符。**{** 和 **}** 特征符可以嵌套。该字符串由处于特征符字符串括起来的各种字符组成，其中包含注释。

```
q{foo}                 // "foo"
q{/*}*/ }              // "/*}*/ "
q{ foo(q{hello}); }    // " foo(q{hello}); "
q{ @ }                 // 错误，@ 不是一个有效的 D 特征符
q{ __TIME__ }          // 例如：" __TIME__ "，它不会被替换成时间
q{ __EOF__ }           // 错误，因为 __EOF__ 不是一个特征符，它是文尾符
```

1.10 字符字法(Character Literals)

```
字符字法:
        ' 单引号字符 '

单引号字符:
        单个字符
        转义序列
```

字符文字是单个的字符或者由单引号''括起来的转义序列。

1.11 整数字法(Integer Literals)

整数字法:
    整数
    整数 整数后缀

整数:
    十进制数
    二进制数
    八进制数
    十六进制数

整数后缀:
    L
    u
    U
    Lu
    LU
    uL
    UL

十进制数:
    0
    非零数字
    非零数字 多个十进制数字

二进制数:
    0b *多个二进制数字*
    0B *多个二进制数字*

八进制数:
    0 *多个八进制数字*

十六进制数:
    0x *多个十六进制数字*
    0X *多个十六进制数字*

非零数字:
    1
    2
    3
    4
    5
    6
    7
    8
    9

多个十进制数字:
    单个十进制数字

```
        单个十进制数字 多个十进制数字

单个十进制数字:
        0
        非零数字
        _

多个二进制数字:
        单个二进制数字
        单个二进制数字 多个二进制数字

单个二进制数字:
        0
        1
        _

多个八进制数字:
        单个八进制数字
        单个八进制数字 多个八进制数字

单个八进制数字:
        0
        1
        2
        3
        4
        5
        6
        7
        _

多个十六进制数字:
        单个十六进制数字
        单个十六进制数字 多个十六进制数字

单个十六进制数字:
        单个十进制数字
        a
        b
        c
        d
        e
        f
        A
        B
        C
        D
        E
        F
        _
```

整数可以采用十进制、二进制、八进制或者十六进制。

十进制整数是十进制数字的序列。

二进制整数是二进制数字的序列，以‘0b’为前缀。

八进制整数是八进制数字的序列，以‘0’为前缀。

十六进制整数指的是一个以‘0x’为前缀的十六进制数字序列。

整数可以内嵌 '_' 字符，它们会被忽略。嵌入的 '_' 可以用于格式化较长的文字，例如作为千位分隔符：

```
123_456         // 123456
1_2_3_4_5_6_    // 123456
```

整数后可以紧跟着一个 'l' 或者一个 'u' 或者两者都有。

整数的类型按照下述规则判断：

十进制文字类型

十进制范围	类型
0 .. 2_147_483_647	int
2_147_483_648 .. 9_223_372_036_854_775_807L	long
十进制范围，带 L 后缀	**类型**
0L .. 9_223_372_036_854_775_807L	long
十进制范围，带 U 后缀	**类型**
0U .. 4_294_967_296U	uint
4_294_967_296U .. 18_446_744_073_709_551_615UL	ulong
十进制范围，带 UL 后缀	**类型**
0UL .. 18_446_744_073_709_551_615UL	ulong
非十进制范围	**类型**
0x0 .. 0x7FFF_FFFF	int
0x8000_0000 .. 0xFFFF_FFFF	uint
0x1_0000_0000 .. 0x7FFF_FFFF_FFFF_FFFF	long
0x8000_0000_0000_0000 .. 0xFFFF_FFFF_FFFF_FFFF	ulong
非十进制范围，带 L 后缀	**类型**
0x0L .. 0x7FFF_FFFF_FFFF_FFFFL	long
0x8000_0000_0000_0000L .. 0xFFFF_FFFF_FFFF_FFFFL	ulong
非十进制范围，带 U 后缀	**类型**

0x0U .. 0xFFFF_FFFFU	uint
0x1_0000_0000UL .. 0xFFFF_FFFF_FFFF_FFFFUL	ulong
非十进制范围，带 UL 后缀	**类型**
0x0UL .. 0xFFFF_FFFF_FFFF_FFFFUL	ulong

1.12 浮点数字法(Floating Literals)

浮点数字法:
浮点数
浮点数 后缀
整数 虚数后缀
整数 浮点后缀 虚数后缀
整数 实数后缀 虚数后缀

浮点数:
十进制浮点数
十六进制浮点数

十进制浮点数:
多个十进制数字 .
多个十进制数字 .多个十进制数字
多个十进制数字 .多个十进制数字 十进制指数
. 十进制数
. 十进制数 十进制指数
多个十进制数字 十进制指数

十进制指数
e *多个十进制数字*
E *多个十进制数字*
e+ *多个十进制数字*
E+ *多个十进制数字*
e- *多个十进制数字*
E- *多个十进制数字*

十六进制浮点数:
十六进制前缀 多个十六进制数字 .多个十六进制数字 十六进制指数
十六进制前缀 .多个十六进制数字 十六进制指数
十六进制前缀 多个十六进制数字 十六进制指数

十六进制前缀:
0x
0X

十六进制指数:
p *多个十六进制数字*
P *多个十六进制数字*
p+ *多个十六进制数字*
P+ *多个十六进制数字*
p- *多个十六进制数字*
P- *多个十六进制数字*

```
后缀:
        浮点后缀
        实数后缀
        虚数后缀
        浮点后缀 虚数后缀
        实数 虚数后缀

浮点后缀:
        f
        F

实数后缀:
        L

虚数后缀:
        i
```

浮点数可以使用十进制或者十六进制格式，如同标准 C 一样。

十六进制浮点数以 **0x** 开头，阶码以 **p** 或 **P** 开头，后面跟着以 2 为底的阶数。

浮点数可以内嵌 '_' 字符，它们会被忽略。嵌入的‘_’用来格式化冗长的文字以提高可读性，例如可以将它们用作千位分隔符：

```
123_456.567_8           // 123456.5678
1_2_3_4_5_6_._5_6_7_8   // 123456.5678
1_2_3_4_5_6_._5e-6_     // 123456.5e-6
```

不带后缀浮点文字类型是 double。浮点数可以跟随有一个 **f**、**F** 或 **L** 后缀。**f** 或 **F** 后缀说明是 fload；**L** 代表 real。

如果浮点文字后面跟着 **i**，那么它就是一个 *ireal* (虚数) 类型。

示例:

```
0x1.FFFFFFFFFFFFFp1023          // double.max
0x1p-52                         // double.epsilon
1.175494351e-38F                // float.min
6.3i                            // idouble 6.3
6.3fi                           // ifloat 6.3
6.3Li                           // ireal 6.3
```

如果该串文字超出了该类型的表示范围，会被视为错误。如果该串文字取整后可以用该类型的有效位数字表示，就不是错误。

复数文字不是特征符，而是在语义分析时用实数和虚数表达式构造的：

```
4.5 + 6.2i              // 复数
```

1.13 关键字(Keywords)

关键字是保留的标识符：

关键字:

```
        abstract
        alias
        align
        asm
        assert
        auto

        body
        bool
        break
        byte

        case
        cast
        catch
        cdouble
        cent
        cfloat
        char
        class
        const
        continue
        creal

        dchar
        debug
        default
        delegate
        delete
        deprecated
        do
        double

        else
        enum
        export
        extern

        false
        final
        finally
        float
        for
        foreach
        foreach_reverse
        function

        goto
```

```
idouble
if
ifloat
import
in
inout
int
interface
invariant
ireal
is

lazy
long

macro
mixin
module

new
nothrow
null

out
override

package
pragma
private
protected
public
pure

real
ref
return

scope
short
static
struct
super
switch
synchronized

template
this
throw
```

```
        true
        try
        typedef
        typeid
        typeof

        ubyte
        ucent
        uint
        ulong
        union
        unittest
        ushort

        version
        void
        volatile

        wchar
        while
        with

        __FILE__
        __LINE__
        __traits
```

1.14 特殊特征符(Special Tokens)

这些特征符会根据下面的表格替换成其它的特征符：

特殊特征符(Special Tokens)

特殊特征符	替换为...
__DATE__	字符串文字，表示的是编译日期："*mmm dd yyyy*"
__TIME__	字符串文字，表示的是编译时间："*hh:mm:ss*"
__TIMESTAMP__	字符串文字，表示的是编译日期和时间："*www mmm dd hh:mm:ss yyyy*"
__VENDOR__	字符串文字，表示的是编译器服务商，如"Digital Mars D"
__VERSION__	整数，表示的是编译器版本，如：2001

1.15 特殊特征符序列(Special Token Sequences)

特殊特征符序列由词法分析程序处理，它可以出现在其他特征符之间，并且不影响语法分析。

目前只有一个特殊特征符序列，`#line`。

```
特殊特征符序列:
        # line 整数 行尾
```

```
        # line 整数 指定文件 行尾

指定文件:
        " 多个字符 "
```

它会将源代码的行号设置为某一个*整数*，可选地将源文件名设置为 *指定的文件*，紧跟在源文件文本的下一行。与码文件名和行号用于打印调试信息，还被符号调试器用于将生成的代码映射回源代码。

例如：

```
int #line 6 "foo\bar"
x;                          // 这里是文件 foo\bar 的第 6 行
```

注意，*Filespec* 字符串中的反斜杠不会被特殊处理。

第 2 章 模块

```
模块:
        模块声明 多个声明定义
        多个声明定义

多个声明定义:
        单个声明定义
        单个声明定义 多个声明定义

单个声明定义:
        属性指示符
        导入声明
        枚举声明
        类声明
        接口声明
        聚集声明
        单个声明
        构造函数
        析构函数
        不变量
        单元测试
        静态构造函数
        静态析构函数
        Debug 指定
        Version 指定
        Mixin 声明
        ;
```

模块(Module)同源文件是一一对应的。模块名就是去掉路径和扩展名的文件名。

模块自动为它的内容提供一个名字空间。模块跟类有一点相像，不同之处是：

- 每个模块只有一个实例，并且它是静态分配的。
- 模块没有虚表(virtual table)。
- 模块不能继承，它们没有父模块，等等。
- 每个文件只有一个模块。
- 模块的符号可以导入。
- 模块总是在全局作用域内编译，并且不受周围的特征或其它修饰符影响。

多个模块可以组织成一个结构，叫做“*包(package)*”。

模块提供了以下几种保证：

- 模块导入的顺序并不会对语义产生什么影响。
- 一个模块的语义不会被那些导入它的模块所影响。
- 如果一个模块 C 导入了模块 A 和 B，那么任何对 B 的修改都不会隐式地更改在模块 C 里的跟 A 相独立的代码。

2.1 模块声明(Module Declaration)

“*模块声明*”指定了模块的名称和它所属的包。如果不指定，模块名将设定为去掉路径和扩

展名的文件名。

```
模块声明:
        module 模块完全限定名 ;
        module (system) 模块完全限定名 ;

模块完全限定名:
        模块名
        包 .模块名

模块名:
        标识符

包:
        包名
        包 .包名

包名:
        标识符
```

最右边的“*标识符*”是模块所在的“*包*”。包在源文件路径中对应于目录名。包名不能是关键字，因此对应的路径名也同样不能是关键字。

如果出现的话，“*模块声明*”按照语法位于源文件的开头，并且每个源文件只能有一个。

样例:

```
module c.stdio;     // 这是模块 stdio，它位于 c 包中
```

按照惯例，包和模块名都为小写。这是因为包和模块名称同操作系统中的目录名和文件名一一对应，而许多文件系统不区分大小写。把所有的包和模块名称小写将减少在不同文件系统之间迁移项目时的问题。

2.2 系统模块

系统模块(System modules) 指的是使用“(System)”标记为安全的模块，该标记就是出现在*模块声明*里的。系统模块一定要是正确实现了安全内存模式，因为即使它们会被强制，但编译器对此也并不做检查。

2.3 Import 声明

与文本的包含文件不同，D 通过使用“*导入声明*”直接导入符号：

```
导入声明:
        import 导入列表 ;
        static import 导入列表 ;

导入列表:
        导入
```

```
        导入绑定
        导入 , 导入列表

导入:
        模块完全限定名
        模块别名标识符 = 模块完全限定名

导入绑定:
        导入 :导入绑定列表

导入绑定列表:
        导入绑定
        导入绑定 , 导入绑定列表

导入绑定:
        标识符
        标识符 = 标识符

模块别名标识符:
        标识符
```

有好几种形式的“*导入声明*”，包括由粗(generalized)到细(fine-grained)的导入。

在“*导入声明*”里的声明顺序并不重要。

模块完全限定名（位于“*导入声明*”中）必须使用它们所处的那个包的名称来完整修饰。它们就不会被认为是相对于那个导入它们的模块。

2.3.1 基本导入(Basic Imports)

最简单的导入形式就是只做列出要被导入的模块：

```
import std.stdio;  // 导入模块 stdio（自 std 包里）
import foo, bar;   // 导入杠杆 foo 和 bar

void main()
{
writefln("hello!\n");  // 调用 std.stdio.writefln
}
```

基本导入工作原理就是：首先，在当前名字空间里搜索名字。如果没有找到，那么就到所有导入的模块里去查找。如果在这些导入模块中唯一找到一个，则使用它。如果在多个导入模块里找到，则就出现错误。

```
module A;
void foo();
void bar();

module B;
void foo();
void bar();

module C;
import A;
void foo();
void test()
```

```
{ foo(); // C.foo() 被调用，它在搜索导入模块之前被找到
  bar(); // A.bar() 被调用，因为搜索了导入模块
}

module D;
import A;
import B;
void test()
{ foo();   // 错误，是 A.foo() 还是 B.foo() 呢?
  A.foo(); // 正确，调用 A.foo()
  B.foo(); // 正确，调用 B.foo()
}

module E;
import A;
import B;
alias B.foo foo;
void test()
{ foo();   // 调用 B.foo()
  A.foo(); // 调用 A.foo()
  B.foo(); // 调用 B.foo()
}
```

2.3.2 公共导入(Public Imports)

默认情况下，导入是“*private(私有的)*”。即表示，如果 A 导入模块 B，同时 B 又导入模块 C，则 C 的名字是不会被搜索的。一个导入可以特别地声明为“*public（公共的）*”，这时它会处理成这样：对于带有“*导入声明*”的模块，它的任何导入模块也会导入那些公共导入的模块。

```
module A;
void foo() { }

module B;
void bar() { }

module C;
import A;
public import B;
...
foo();  // 调用 A.foo()
bar();  // 调用 B.bar()

module D;
import C;
...
foo();  // 错误，foo() 未定义
bar();  // 正确，调用 B.bar()
```

2.3.3 静态导入(Static Imports)

基本导入对于相对没有几个模块和导入的程序很有效。如果存在大量模块导入，则在各种不同的导入模块里，名字冲突就会出现。防止此种情况发生的一种方式就是使用“静态导入(static imports)”。静态导入要求我们使用带完整修饰的名称来引用模块名。

```
static import std.stdio;

void main()
{
writefln("hello!");             // 错误，writefln 未定义
    std.stdio.writefln("hello!");  // 正确，writefln 被完全限定
}
```

2.3.4 更名导入(Renamed Imports)

可以为某个导入给出本地名称；通过这个本地名称，所有对该模块符号的引用都必须进行限定：

```
import io = std.stdio;

void main()
{
io.writefln("hello!");          // 正确，调用 std.stdio.writefln
    std.stdio.writefln("hello!");  // 错误，std 未定义
    writefln("hello!");            // 错误，writefln 未定义
}
```

更名的导入在处理很长的导入名时很方便。

2.3.5 选择性导入(Selective Imports)

特定的符号可以特别地从一个模块被导入，并被绑定到当前名字空间里。

```
import std.stdio : writefln, foo = writef;

void main()
{
std.stdio.writefln("hello!");  // 错误，std 未定义
    writefln("hello!");            // 正确，writefln 被绑定到当前名字空间
    writef("world");               // 错误，writefln 未定义
    foo("world");                  // 正确，调用 std.stdio.writef()
    fwritefln(stdout, "abc");      // 错误，writefln 未定义
}
```

static 不能跟选择性导入一起使用。

2.3.6 更名的且带选择性的导入

更名的且带选择性的导入被组合的情形：

```
import io = std.stdio : foo = writefln;

void main()
{
```

```
writefln("bar");            // 错误，writefln 未定义
    std.stdio.foo("bar");       // 错误，foo 被绑定到当前名字空间
    std.stdio.writefln("bar"); //  错误，std 未定义
    foo("bar");                 // 错误，foo 被绑定到当前名字空间，
                            // FQN 不需要
    io.writefln("bar");         // 正确，io=std.stdio 将在当前名字空间里的名称 io
                            // 绑定来引用整个模块
    io.foo("bar");              //  错误，foo 被绑定到当前名字空间，
                            // 而 foo 不是 io 的成员
```

2.3.7 模块域运算符

有些时候，有必要重写通常的词法作用域规则以访问被局部名称掩盖的名称。可以用全局作用域运算符‘.’达到这个目的，全局运算符位于标志符之前：

```
int x;

int foo(int x)
{
    if (y)
        return x;               // 返回的是 foo.x，而非全局的 x
    else
        return .x;              // 返回全局的 x
}
```

前导的‘.’意味着在模块作用域级别查找名称。

2.4 静态构造和析构

静态构造函数是用来在 main() 之前运行的初始化模块或类的代码。静态析构函数是在 main() 返回之后执行的代码，通常用来释放系统资源。

在一个模块里可以有多个静态构造函数和静态析构函数。静态构造函数的是以词法顺序运行的，而静态析构函数则是以词法相反的顺序运行。

2.4.1 静态构造的顺序

静态初始化的顺序隐式地由模块内的 *import* 声明的顺序决定。每个模块都会在它所依赖的模块之后调用自己的静态构造函数。除了这条规则以外，模块静态构造函数的执行顺序是不定的。

导入声明中的循环（循环依赖）是允许的，只要不是两个模块都含有静态构造或析构函数就行。如果违反这条规则会在运行时产生异常。

2.4.2 一个模块中静态构造的顺序

在模块内部，静态构造会按照它们出现的词法顺序执行。

2.4.3 静态析构的顺序

静态析构将按照与构造函数相反的顺序执行。只有当模块的静态构造函数成功执行后，才会执行静态析构函数。

2.4.4 单元测试的顺序

单元测试是以出现在模块里的词法顺序运行的。

2.5 Mixin 声明

```
Mixin 声明:
mixin ( 赋值表达式 ) ;
```

赋值表达式 必须在编译时求值成一个常量字符串。该字符串的文本内容必须要可编译成一个有效的*多个声明定义*，而且同样地被编译。

第 3 章 声明

单个声明:
typedef *声明*
alias *声明*
声明

声明:
存储类别 声明
基本类型 多个声明符 **;**
基本类型 单个声明符 函数体
自动声明

多个声明符:
声明符初始值
声明符初始值 **,** *声明符标志符列表*

声明符初始值:
声明符
声明符 **=** *初始值*

声明符标志符列表:
声明符标志符
声明符标志符 **,** *声明符标志符列表*

声明符标志符:
标识符
标识符 **=** *初始值*

基本类型:

```
bool
byte
ubyte
short
ushort
int
uint
long
ulong
char
wchar
dchar
float
double
real
ifloat
idouble
ireal
cfloat
cdouble
creal
void
```

```
.标识符列表
标识符列表
        Typeof
        Typeof .标识符列表

基本类型 2:
        *
        [ ]
[ 表达式 ]
[ 表达式 ..表达式 ]
[ 类型 ]
delegate 多个形参 多个函数属性opt
function 多个形参 多个函数属性opt

声明符:
基本类型 2 声明符 多个声明符后缀可选的
基本类型 2 标识符 多个声明符后缀可选的

多个声明符后缀:
单个声明符后缀
单个声明符后缀 多个声明符后缀

单个声明符后缀:
        [ ]
[ 表达式 ]
[ 类型 ]
        模板参数列表可选的 多个参数 多个成员函数属性可选的

标志符列表:
标识符
标识符 .标识符列表
模板实例
模板实例 .标识符列表

多个存储类别:
        单个存储类别
        单个存储类别 多个存储类别

单个存储类别:
        abstract
        auto
        const
        deprecated
        extern
        final
        invariant
        nothrow
        override
```

```
        pure
        scope
        static
        synchronized

类型:
基本类型
基本类型 声明符2

声明符2:
基本类型2 声明符2
( 声明符2 )
( 声明符2 ) 多个声明符后缀

多个形参:
        ( 形参列表 )
        ( )

形参列表:
单个形参
单个形参 , 形参列表
单个形参 ...
        ...

单个形参:
声明符
声明符 = 默认初始化表达式
InOut 声明符
InOut 声明符 = 默认初始化表达式

InOut:
        in
        out
        ref
        lazy

多个函数属性:
        单个函数属性
        单个函数属性 多个函数属性

单个函数属性:
        nothrow
        pure

多个成员函数属性:
        成员函数属性
        单个成员函数属性 多个成员函数属性

单个成员函数属性:
        const
        invariant
        单个函数属性

默认初始化表达式:
        赋值表达式
```

```
        __FILE__
        __LINE__

初始值：
空初始值
        非空初始值

非空初始值：
赋值表达式
数组初始值
结构初始值

数组初始值：
        [ ]
        [ 多个数组成员初始化 ]

多个数组成员初始化：
        单个数组成员初始化
        单个数组成员初始化 ,
        单个数组成员初始化 , 多个数组成员初始化

单个数组成员初始化：
        非空初始值
        赋值表达式 :非空初始值

结构初始值：
        {  }
        { 多个结构成员初始值 }

多个结构成员初始值：
        单个结构成员初始值
        单个结构成员初始值 ,
        单个结构成员初始值 , 多个结构成员初始值

单个结构成员初始值：
        非空初始值
        标识符 :非空初始值
```

3.1 声明语法

声明语法通常从右向左读：

```
int x;          // x 是一个 int
int* x;         // x 是指向 int 的指针
int** x;        // x 是一个指向指针的指针，而被指向的指针又指向 int
int[] x;        // x 是 int 型的数组
int*[] x;       // x 是一个指针数组，该指针指向 int
int[]* x;       // x 是指向 int 型数组的指针
```

数组读法也是从右至左：

```
int[3] x;        // x 是一个拥有 3 个 int 类型元素的数组
int[3][5] x;     // x 是拥有 5 个元素的数组，每个元素是有 3 个元素的 int 数组
int[3]*[5] x;    // x 是有 5 个元素的数组，每个元素是指向有 3 个元素的 int 的数组的指针
```

指向函数的指针用关键字 **function** 声明：

```
int function(char) x;    // x 为指向带有一个 char 类型参数的函数，
                         // 并且返回值的类型是 int
int function(char)[] x; // x 为一个函数指针数组，
                         // 此函数有一个 char 类型的参数，而返回值类型是 int
```

也可以使用 C 风格的数组声明方式：

```
int x[3];           // x 是一个拥有 3 个 int 类型元素的数组
int x[3][5];        // x 是拥有 3 个元素的数组，每个元素是有 5 个元素的 int 数组
int (*x[5])[3];     // x 是有 5 个元素的数组，每个元素是指向有 3 个元素的 int 的数组的指针
int (*x)(char);     // x 是指向函数的指针，函数有一个 char 类型的参数，
                 // 并且返回值的类型是 int
int (*[] x)(char); // x 是一个函数指针数组，
                 // 此函数有一个 char 类型的参数，而返回值类型是 int
```

在多重声明中，所有被声明的符号拥有相同的类型：

```
int x,y;         // x 和 y 是 int
int* x,y;        // x 和 y 是指向 int 的指针
int x,*y;        // 错误，多重类型
int[] x,y;       // x 和 y 是 int 数组
int x[],y;       // 错误，多重类型
```

3.2 隐式类型接口

自动声明：

多个存储类别 标识符 **=** *赋值表达式* **;**

如果一个声明由“*存储类别*”起始，并且有“*非空初始值*”（由这可以推断类型），则在声明里的类型可以被忽略。

```
static x = 3;       // x 的类型是 int
auto y = 4u;        // y 的类型是 uint
auto s = "string"; // s is type immutable(char)[6]

class C { ... }

auto c = new C();  // c 是类 C 实例的手柄
```

“*非空初始值*”不能包含向前引用（在将来这个限制可能被移除）。隐式推断的类型是在编译时，而在非运行时静态绑定到该声明。

3.3 类型定义(Type Defining)

强类型(Strong types)可以通过 typedef 引入。对于函数重载和调试器来说，在语义上，强类型是类型检查系统可以将其同其他类型区分的类型。

```
typedef int myint;

void foo(int x) { . }
void foo(myint m) { . }


 .
myint b;
foo(b);          // 调用 foo(myint)
```

Typedef 可以指定一个跟基础类型（underlying type）的初始值不同的值：

```
typedef int myint = 7;
myint m;         // 初始化为 7
```

3.4 类型别名(Type Aliasing)

时为类型起一个别名是很方便的，例如可以作为一个冗长复杂类型（比如一个函数指针）的简写形式。在 D 中，可以使用别名声明达到这个目的：

```
alias abc.Foo.bar myint;
```

别名类型在语义上等价于原类型。调试器无法区分它们，在函数重载是也不会区分它们。例如：

```
alias int myint;

void foo(int x) { . }
void foo(myint m) { .}  // 错误，多次定义了函数 foo
```

类型别名等价于 C 中的 typedef。

3.5 Alias 声明

一种符号可以被声明为另一种符号的“*别名(alias)*”。例如：

```
import string;

alias string.strlen mylen;
 ...
int len = mylen("hello");        // 实际上调用的是 string.strlen()
```

下面的别名声明都是有效的：

```
template Foo2(T) { alias T t; }
alias Foo2!(int) t1;
```

```
alias Foo2!(int).t t2;
alias t1.t t3;
alias t2 t4;

t1.t v1;         // v1 的类型是 int
t2 v2;           // v2 的类型是 int
t3 v3;           // v3 的类型是 int
t4 v4;           // v4 的类型是 int
```

别名符号可以将一个较长的符号简记为一个较短的符号，还可以将一个引用从一个符号重定位到另一个符号：

```
version (Win32)
{
    alias win32.foo myfoo;
}
version (linux)
{
    alias linux.bar myfoo;
}
```

别名可以用来从一个模块中‘导入(import)’一个符号到当前作用域里：

```
alias string.strlen strlen;
```

别名还可以‘导入’一系列的重载函数，这样就可以在当前作用域中重载函数：

```
class A {
    int foo(int a) { return 1; }
}

class B : A {
    int foo( int a, uint b ) { return 2; }
}

class C : B {
    int foo( int a ) { return 3; }
    alias B.foo foo;
}

class D : C  {
}


void test()
{
    D b = new D();
    int i;

    i = b.foo(1, 2u);   // 调用 B.foo
    i = b.foo(1);       // 调用 C.foo
}
```

注意： 类型别名有时会同别名声明十分相似：

```
alias foo.bar abc;       // 它是一个类型还是一个符号?
```

在语义分析是将会区分这两种情况。

别名不能用于表达式：

```
struct S { static int i; }
S s;

alias s.i a;     // 非法，s.i 是一个表达式
alias S.i b;     // 正确
b = 4;           // 把 S.i 设置为 4
```

3.6 Extern 声明

带有存储类别 **extern** 的变量声明不会在该模块里分配存储空间。它们在其它某个目标文件里被定义，而且要求跟要连接的名称相匹配。它的基本用法就是用于连接在 C 文件里定义的全局变量。

3.7 typeof

```
Typeof:
typeof ( 表达式 )
       typeof ( return )
```

Typeof 用来获得一个表达式的类型。例如：

```
void func(int i)
{
    typeof(i) j;         // j 的类型是 int
    typeof(3 + 6.0) x;   // x 是 double 型
    typeof(1)* p;        // p 是指向 int 型的指针
    int[typeof(p)] a;    // a 的类型是 int[int*]

    writefln("%d", typeof('c').sizeof); // 输出 1
    double c = cast(typeof(1.0))j;      // 将 j 转换为 double 型
}
```

表达式 不会被计算，只会生成它的类型：

```
void func()
{   int i = 1;
    typeof(++i) j;       // j 被声明为 int ，i 不会自增
    writefln("%d", i);   // 输出 1
}
```

有三种特殊情况：

1. 就算不在成员函数中，**typeof(this)** 也将生成 **this** 在非静态成员函数中的类型。
2. 类似的，**typeof(super)** 也将生成 **super** 在非静态成员函数中的类型。
3. 当处于一个函数的作用域内部时，**typeof(return)** 会产生该函数的返回类型。

```
class A { }

class B : A
{
    typeof(this) x;      // x 被声明为 B 型
    typeof(super) y;     // y 被声明为 A 型
}

struct C
{
    typeof(this) z;      // z 被声明为 C* 型
    typeof(super) q;     // 错误，C 没有父结构
}

typeof(this) r;          // 错误，没有包含在类或者结构中
```

Typeof 是最有用的的地方就是在于编写泛型模板代码。

3.8 Void 初始化

```
空初始值:
       void
```

通常，变量初始化可以使用一个显式的 *初始值*，也可以设置成变量类型的默认值。如果 *初始化值* 为 **void**，实际上变量并没有被初始化。如果它的值在被设置前使用了，则会导致未定义的程序行为。

```
void foo()
{
    int x = void;
    writefln(x);         // 会输出“垃圾”信息
}
```

因此，我们应该只使用 **void** 初始值，做为在优化关键代码时的最后的手段。

第 4 章 类型

4.1 基本数据类型

基本数据类型

关键字	描述	默认初始值（.init）
void	无类型	-
bool	布尔值	false
byte	8 位有符号数	0
ubyte	8 位无符号数	0
short	16 位有符号数	0
ushort	16 位无符号数	0
int	32 位有符号数	0
uint	32 位无符号数	0
long	64 位有符号数	0L
ulong	64 位无符号数	0L
cent	128 位有符号数（预留将来使用）	0
ucent	128 位无符号数（预留将来使用）	0
float	32 位浮点数	float.nan
double	64 位浮点数	double.nan
real	硬件支持的最大的浮点大小（**实现提示：** 80 bits for x86 CPUs) or double size, whichever is larger	real.nan
ifloat	浮点虚数	float.nan * 1.0i
idouble	双精度虚数	double.nan * 1.0i
ireal	实型虚数	real.nan * 1.0i
cfloat	带两个浮点值的复数	float.nan + float.nan * 1.0i
cdouble	双精度复数	double.nan + double.nan * 1.0i
creal	实型复数	real.nan + real.nan * 1.0i
char	8 位无符号 UTF-8 数	0xFF
wchar	16 位无符号 UTF-16 数	0xFFFF
dchar	32 位无符号 UTF-32 数	0x0000FFFF

4.2 派生的数据类型

- 指针(pointer)
- 数组(array)
- 关联数组
- function
- delegate

*字符串*是数组的特例。

4.3 自定义数据类型

- alias
- typedef
- enum
- struct
- union
- class

4.4 基类型(Base Types)

枚举的 *基类型* 就是其所依据的类型：

```
enum E : T { ...}        // T 就是 E 的 基类型
```

类型定义的 *基类型* 就是其所组成的类型：

```
typedef T U;             // T 就是 U 的 基类型
```

4.5 指针转换

D 允许指针到非指针、非指针到指针的转换；但是，决不要对指向由垃圾回收程序所分配的数据的指针进行此类操作。

4.6 隐式转换

隐匿转换用于在需要的时候自动完成类型转换。

类型定义(typedef)或枚举(enum)可以隐式地转换到它的基类型，但在其它环境就需要显示转换。typedef 定义的类型会被隐式转换到它的实际类型。例如：

```
typedef int myint;
int i;
myint m;
i = m;                   // 正确
m = i;                   // 错误
m = cast(myint)i;        // 正确
```

```
m = 3;                    // 正确

enum Foo { E }
Foo f;
i = f;                    // 正确
f = i;                    // 错误
f = cast(Foo)i;           // 正确
f = 0;                    // 错误
f = E;                    // 正确
```

4.7 整数提升

整数提升可以转换下列类型：

整数提升

源类型	目标类型
bool	int
byte	int
ubyte	int
short	int
ushort	int
char	int
wchar	int
dchar	uint

如果类型定义或枚举的基类型时是上面左列中的某种类型，那么也会转换到右列中相应类型。

4.8 常见的算术转换

常见的算术转换会将二元运算符的操作数转换为通用的类型。这个操作数必须已经是算术类型。下面的规则将会按顺序应用：

1. 如果操作数是实型，那么其它的操作数将被转换成实型。
2. 如果操作数是双精度数，那么其它的操作数将被转换成双精度数。
3. 如果操作数是浮点数，那么其它的操作数将被转换成浮点数。
4. 否则才对每个操作数进行整数提升，步骤如下：
    1. 如果两个操作数类型相同，则无需再作转换。
    2. 果两个操作数都是有符号或无符号的，较小的类型会被转换为较大的类型。
    3. 如果有符号的类型比无符号的类型大，无符号的类型会被转换为有符号的类型。
    4. 否则有符号的类型会被转换为无符号的类型。

如果有一到两个操作数类型在经上面的转换后变成了类型定义或枚举，则最终类型是：

1. 如果操作数是相同类型，则结果也就是该类型。
2. 如果其中一个操作数是类型定义或枚举，而另一个的类型是类型定义或枚举的基类

型，则结果就是基类型。

3. 如果两个操作数是不同的类型定义或枚举，但有相同的基类型，则结果就是基类型。

整型值不能隐式地转换成其它在整数提升后不能代表整数位模式的类型。例如：

```
ubyte  u1 = cast(byte)-1;   // 错误，-1 不能表示成 ubyte 类型
ushort u2 = cast(short)-1;  // 错误，-1 不能表示成 ushort 类型
uint   u3 = cast(int)-1;    // 错误，-1 不能表示成 uint 类型
ulong  u4 = cast(ulong)-1;  // 正确，-1 可以表示成 ulong 类型
```

浮点类型不能隐式转换成整数类型。

浮点复数类型不能隐式转换成浮点非复数类型。

浮点虚数类型不能隐式转换成浮点、双精度或实型类型。浮点、双精度或实型类型不能隐式转换成浮点虚数类型

4.9 bool

bool（布尔类型）是一种只有一个字节大小的类型，它仅能存放 true 或 false，即真或假。只能接受布尔类型操作数的操作符有：& | ^ &= |= ^= ! && || ?:. 一个布尔值可以隐式地转换成任何整数类型：false 变成 0，而 true 变成 1。单独的数字 0 和 1 可以隐式地转换成相应的布尔值 false 和 true。测试一个表达式的布尔值即意味着：在算术类型时，测试 0 或!=0；而测试 null 或 !=null 则是表示浮点或引用运算。

4.10 委托(Delegates)

在 D 中没有成员指针(pointers-to-members)，但它支持一个更为有用的概念，即*委托(delegates)*。委托是一个由两部分数据组成的聚集：一个是对象引用；另一个是函数指针。当调用函数时，对象引用就形成 *this* 指针。

委托的声明同函数指针的声明很相像，区别就是使用关键字 **delegate** 来替代 (*)，并且随后紧跟着标志符：

```
int function(int) fp;   // fp 是指向函数的指针
int delegate(int) dg;   // dg 是函数的委托
```

C 风格的声明函数指针的语法也被支持：

```
int (*fp)(int);         // fp 是指向函数的指针
```

委托的初始化同函数指针一样：

```
int func(int);
fp = &func;             // fp 指向 func

class OB
{   int member(int);
}
```

```
OB o;
dg = &o.member;          // dg 是对象 o 和
                      // 成员函数 member 的委托
```

委托不能用静态成员函数或者非成员函数初始化。

委托跟函数指针的调用是等同的：

```
fp(3);           // 调用 func(3)
dg(3);           // 调用 o.member(3)
```

第 5 章 特性

每种类型和表达式都有可以查询的特性(Properties)：

特性示例

表达式	值
int.sizeof	计算得 4
float.nan	计算得浮点数 nan（不是一个数）值
(float).nan	计算得浮点数 nan 值
(3).sizeof	计算得 4（因为 3 是一个 int 类型）
2.sizeof	语法错误，因为“2.”是一个浮点数
int.init	int 的默认初始值
int.mangleof	产生串“i”
int.stringof	产生串“int”
(1+2).stringof	产生串“1 + 2”

所有类型都拥有的特性

特性	描述
.init	初始值
.sizeof	字节大小（等同于 C 中的 sizeof(类型)）
.alignof	对齐大小
.mangleof	字符串所代表的是该类型的“碎解"表示(mangled representation)
.stringof	字符串所代表的是该类型的源表示(source representation)

整数类型的特性

特性	描述
.init	初始值 (0)
.max	最大值
.min	最小值

浮点类型的特性

特性	描述
.init	初始值 (NaN)

.infinity	无穷大
.nan	NaN 值
.dig	换算为十进制表示后的精度
.epsilon	最小递增值 1
.mant_dig	尾数的位数
.max_10_exp	以 $10^{max\_10\_exp}$ 形式可以表示的最大整数值
.max_exp	以 2^{max_exp-1} 形式可以表示的最大整数值
.min_10_exp	以 $10^{min\_10\_exp}$ 形式可以表示成的标准值的最小整数值
.min_exp	以 2^{min_exp-1} 形式可以表示成的标准值的最小整数值
.max	可表示的最大值（除了无穷）
.min	可表示的最小值（除了 0）
.re	实数部分
.im	虚数部分

5.1 .init 特性

.init 产生一个常量表达式，其值为默认初始值。如果应用到某个类型，其值为该类型的默认初始值。如果应用到某个变量或域，其值为这个变量或域的默认初始值。例如：

```
int a;
int b = 1;
typedef int t = 2;
t c;
t d = cast(t)3;

int.init        // 为 0
a.init          // 为 0
b.init          // 为 1
t.init          // 为 2
c.init          // 为 2
d.init          // 为 3

struct Foo
{
    int a;
    int b = 7;
}

Foo.a.init      // 为 0
Foo.b.init      // 为 7
```

5.2 .stringof 特性

.stringof 会生成一个常量串，即其前缀的“源表示(source representation)”。如果被应用到某个类型，那么它就是那种类型的字符串。如果被应用到一个表达式，那么它就是那个表达式的“源表示”。语义分析并不会处理那个表达式。例如：

```
struct Foo { }

enum Enum { RED }

typedef int myint;

void main()
{
    writefln((1+2).stringof);        // "1 + 2"
    writefln(Foo.stringof);          // "Foo"
    writefln(test.Foo.stringof);     // "test.Foo"
    writefln(int.stringof);          // "int"
    writefln((int*[5][]).stringof); // "int*[5][]"
    writefln(Enum.RED.stringof);     // "Enum.RED"
    writefln(test.myint.stringof);   // "test.myint"
    writefln((5).stringof);          // "5"
}
```

5.2.1 类和结构的特性

特性是成员函数，但在语法上被看作是域。特性可以被读写。特性可以通过调用一个不带实参的方法来读取；再通过调用一个带有实参的方法来进行写入，其中的实参就会成为该特性的值。

简单的特性可以表示成：

```
struct Foo
{
    int data() { return m_data; }        // 读取特性

int data(int value) { return m_data = value; } // 写入特性

  private:
    int m_data;
}
```

使用方法：

```
int test()
{
    Foo f;

    f.data = 3;          // 等同于 f.data(3);
    return f.data + 3;   // 等同于 return f.data() + 3;
}
```

如果没有读方法就意味着特性是只写的。如果没有写方法就意味着特性是只读的。可以有多个写方法共存，用正常的重载规则来决定采用哪个函数。

在其他方面，这些方法同其他的方法都是相同的。它们可以是静态的、拥有不同的连接属性(linkage)、被重载、被取地址等等。

注意：目前特性不能作为运算符 *op*=、++ 或 -- 的左值。

5.3 .sizeof 特性

`e.sizeof` 会给出表达式 *e* 的字节大小。

在获取某个成员的大小时，此处不必是 *this* 对象：

```
struct S {
    int a;
    static int foo() {
        return a.sizeof;    // 返回 4
    }
}

void test() {
int x = S.a.sizeof;     // 将 x 设置为 4
}
```

`.sizeof` 应用到一个类对象时，它会返回类引用的大小，而不是类实例的大小。

5.4 .alignof 特性

`.alignof` 会给出表达式或类型的对齐大小。例如，对齐大小为 1，即表示它会以一个字节为边界进行对齐，而 4 则表示它会以 32 位为界进行对齐。

第 6 章 属性

```
属性指示器:
属性 :
属性 声明定义块

属性:
连接属性
对齐属性
    Pragma
    deprecated
保护属性
    static
    final
    override
    abstract
    const
    auto
    scope

单个声明块
单个声明定义
    { }
{ 多个声明定义 }
```

属性指的是用来修改一个或多个声明的方法。一般的形式有：

```
attribute declaration;          仅对该声明有效

attribute:                      对所有声明都有效，直到
                              当前作用域的结尾
    declaration;
    declaration;
    ...

attribute                       对语句块中所有的声明都有效
{
    declaration;
    declaration;
    ...
}
```

对于出现在可选的 else 子句中的声明：

```
attribute
    declaration;
else
    declaration;

attribute                       对语句块中所有的声明都有效
{
    declaration;
    declaration;
```

```
    ...
}
else
{
    declaration;
    declaration;
    ...
}
```

6.1 连接属性(Linkage Attribute)

```
连接属性：
        extern
        extern ( 连接类型 )

连接类型：
        C
        C++
        D
        Windows
        Pascal
        System
```

D 提供了一种方便的调用 C 函数和操作系统 API 函数的方法，因为对这两者的兼容都是必要的。*链接类型* 是大小写敏感的，并且可以由具体实现进行扩展（但它们不能是关键字）。**C** 和 **D** 是必须要有的，其它的则取决于具体的实现。**C++** 被保留以便将来使用。 **System** is the same as **Windows** on Windows platforms, and **C** on other platforms.**实现注意：** 对于 Win32 平台，**Windows** 和 **Pascal** 应该总是存在。

通过下面的方式指定 C 函数调用约定：

```
extern (C):
        int foo();       // 使用 C 协定调用 foo()
```

D 调用协定是：

```
extern (D):
```

或者：

```
extern:
```

Windows API 调用协定是：

```
extern (Windows):
    void *VirtualAlloc(
    void *lpAddress,
    uint dwSize,
    uint flAllocationType,
```

```
    uint flProtect
    );
```

6.2 对齐属性(Align Attribute)

```
对齐属性：
        align
        align ( Integer )
```

指定结构成员如何对齐。如果只有 **align**，则将其设置为默认值，该值跟同阵营里的 C 编译器的默认对齐值是相同的。*Integer* 用于指定一个对齐值；当使用非默认的对齐值时，它会匹配同阵营里的 C 编译器的行为。

匹配同阵营里的 C 编译器的行为可能会产生一些令人惊讶的后果，例如，对于 Digital Mars C++ 则有下列情况：

```
struct S
{    align(4) byte a;     // 放置到偏移 0 处
    align(4) byte b;     // 放置到偏移 1 处
}
```

“*对齐属性*”即表示兼容 C ABI，这个跟跨平台的二进制兼容可不是一回事。看看下面压缩后的结构：

```
align (1) struct S
{    byte a;      // 放置到偏移 0 处
    byte[3] filler1;
    byte b;      // 放置到偏移 4 处
    byte[3] filler2;
}
```

值 1 表示不进行对齐；成员被压缩在一起。

不对使用“*New 表达式*”在不是 `size_t` 的倍数的边界上分配的引用或指针进行对齐操作。垃圾回收器假定 gc 所分配对象的指针和引用是在 `size_t` 字节边界上。如果不是那些，就会导致未定义行为。

在被应用到那些不是结构或结构成员的声明时，“*对齐属性*”会被忽略掉。

6.3 废弃属性(Deprecated Attribute)

我们经常会将库中的某一属性标记为废弃，同时仍然保留它以保持向后兼容。这类声明可以被标记为废弃，这意味着可以通过设置编译器选项，在遇到引用这类声明的代码时产生错误：

```
deprecated
{
        void oldFoo();
}
```

实现注意： 编译器应该有一个选项用来指定是否被废弃的属性可以通过编译。

6.4 保护属性(Protection Attribute)

```
保护属性:
    private
    package
    protected
    public
    export
```

可用的保护属性有：**private**、**package**、**protected**、**public** 或 **export**。

private（私有的）表示只有本类的成员，或者跟该类处于同一模块内的成员和函数，才可以访问该成员。私有成员不能被重写。私有模块成员等价于 C 程序中的 **static** 声明。

package（包）扩展了 private 的访问权限，这样包中的成员就可以访问位于同一包内的其他的模块内的成员。如果一个模块位于嵌套的包，那么这条规则只对最里层的包有效。

protected（受保护的）表示只有本类的成员，或者所有自它派生的类的成员，或者跟该类处于同一模块内的成员和函数，才可以访问该成员。如果通过一个派生类成员函数访问一个被保护的实例成员，则访成员仅能被那些是该成员函数调用的“this”对象实例访问。模块成员进行保护是非法的。

public（公有的）表示在可执行程序中的任何代码都可以访问这个成员。

export（导出）表示可执行程序外的任何代码都可以访问这个成员。导出的意思也就是从 DLL 中导出定义。

6.5 常量属性(Const Attribute)

```
const
```

const 属性声明可以在编译时进行求值的常量。例如：

```
const int foo = 7;

const
{
    double bar = foo + 6;
}
```

6.6 重写属性(Override Attribute)

```
override
```

override 属性用于虚函数。即表示该函数一定会重写(override)在基类里具有相同的名字和参数函数。当某个基类的成员函数的参数变化时，override 属性有助于捕获错误；而且，这也要求所有派生类都必须更新它们的重写函数。

```
class Foo
{
    int bar();
    int abc(int x);
}

class Foo2 : Foo
{
    override
    {
        int bar(char c);        // 错误，Foo 中没有 bar(char)
        int abc(int x);         // 正确
    }
}
```

6.7 静态属性(Static Attribute)

```
static
```

static 属性用于函数和数据。即表示此声明不能应用于一个对象的特定实例，不过可以应用于对象的类型。换句话说，就是不存在 **this** 引用。当用于其他声明时， **static** 会被忽略。

```
class Foo
{
    static int bar() { return 6; }
    int foobar() { return 7; }
}

...

Foo f = new Foo;
Foo.bar();      // 得到 6
Foo.foobar();   // 错误，没有 Foo 的实例
f.bar();        // 得到 6;
f.foobar();     // 得到 7;
```

静态函数决不能是虚的(vitual)。

在整个程序中，静态数据只有一个实例，而不能每个对象都有一个。

static 并没有 C 中那样局限于一个文件的功能。在 D 里，可以使用 **private** 属性来实现它。例如：

```
module foo;
int x = 3;              // x 是全局的
private int y = 4;      // y 仅限于模块 foo 中有效
```

6.8 自动属性(Auto Attribute)

```
auto
```

auto 属性用于局部变量和类声明。

```
auto i = 6.8;   // 声明 i 为 double
```

6.9 域属性(Scope Attribute)

```
scope
```

scope 属性用于局部变量和类声明。对于类声明，**scope** 属性会创建一个 *scope* 类。对于局部变量，**scope** 实现 RAII（资源获得即初始化）协议。即表示对象的析构函数自动被调用，当对它的引用 超出作用域时。即使通过抛出异常退出了该作用域，析构函数还是会被调用，这样 **scope** 就被用来保证清除工作。

如果有不只一个 **scope** 变量超出了同一个点的作用域，那么析构函数会依着跟创建这些变量时相反的顺序被调用。

scope 不能用于全局变量、静态变量、数据成员，以用 ref 或 out 参数。**scope** 数组是不被允许的，并且 **scope** 函数返回值也是不允许的。分配一个 **scope**，而不是初始化，同样不被允许。**基本解释：** 这些限制在将来可能被放松，如果理由充分的话。

6.10 抽象属性(Abstract Attribute)

如果一个类是抽象，那么它就不能被直接实例化。它只能被当作另一个非抽象类的基类进行实例化。

如果类被定义时，内部有一个抽象属性(abstract attribute)，或者在它内部的某些虚成员函数被声明成抽象的，那么这个类就是抽象的。

非虚函数不能被声明成抽象的。

函数声明成抽象的仍然可以拥有函数体。这就是为什么即使它们必须被重写(override)，仍然能够提供“基类功能。”

第 7 章 Pragmas

```
Pragma:
pragma ( 标识符 )
pragma ( 标识符 , 表达式列表 )

表达式列表:
        表达式列表 , 表达式
        表达式
```

Pragma（编译器指令）是一种传递特殊信息给编译器以及往 D 里添加服务商的特定扩展的一种方式。Pragma 以 ‘;’结尾，它们可以影响一条语句、一块语句、一个声明或者一块声明。

Pragmas 可以做为声明：*Pragma 声明块*；或者做为语句：*Pragma 语句*。

```
pragma(ident);              // 单独使用

pragma(ident) declaration; // 影响一个声明

pragma(ident):              // 影响随后的声明
    declaration;
    declaration;

pragma(ident)               // 影响一块声明
{    declaration;
    declaration;
}

pragma(ident) statement;    // 影响一个声明

pragma(ident)               // 影响一块语句
{    statement;
语句;
}
```

pragma 的类型由 *标志符* 指明。*表达式列表* 是由逗号分隔的 *赋值表达式* 列表。*赋值表达式* 必须可以作为表达式解析，但它们的语义取决于具体 pragma 的语义。

7.1 预定义 Pragma

所有的实现必须支持这些指令，不然就忽略它们：

msg

在编译时打印出消息，*赋值表达式* 必须是字符串文字：

```
pragma(msg, "compiling...");
```

lib

在目标文件里插入一个指示，用于连接由“*赋值表达式*”所指定的库。“*赋值表达式*”必须是字符串文字：

```
pragma(lib, "foo.lib");
```

startaddress

放置一个指示块到目标文件里以表明第一个参数所指定的函数将是该程序的开始地址：

```
void foo() { ... }
pragma(startaddress, foo);
```

这个对于应用级编程通常是不会使用到的，但是对于特定的系统工作是有用的。对于应用代码，起始地址由运行库来维护。

7.2 服务商指定的 Pragma

服务商指定的 pragma *标识符* 可以被定义成这样：以服务商的商标名作为前缀，类似于 version 标识符：

```
pragma(DigitalMars_funky_extension) { ... }
```

编译器对于不能识别的 *Pragma*，必须给出一个错误，即使它们由是服务商指定的。这个表示服务商指定的 pragma 应该使用 version 语句封装起来：

```
version (DigitalMars)
{
    pragma(DigitalMars_funky_extension) { ... }
}
```

第 8 章 表达式

C 和 C++ 程序员会发现 D 中的表达式很熟悉，另外还有一些有意思的扩充。

表达式用来计算多个值并返回一个特定类型的值。随后，所得的值可以被用于赋值、测试或被忽略。表达式也可能有副作用。

8.1 求值顺序

下列二元表达式以严格的自左向右的顺序进行求值：

Or 表达式、*Xor 表达式*、*And 表达式*、*Cmp 表达式*、*Shift 表达式*、*Add 表达式*、*Cat 表达式*、*Mul 表达式*、 *逗号表达式*、*OrOr 表达式*、*AndAnd 表达式*

下列二元表达式以实现所定义的顺序进行求值：

赋值表达式、 函数参数

当它不是特定的的时候，依赖于求值顺序就是个错误。例如，下面的就是非法的：

```
i = i++;
c = a + (a = b);
func(++i, ++i);
```

如果编译器能够确定表达式的值是依赖于求值顺序的话，它将报告一个错误（但这不是必须的）。检测这种错误的能力是实现的质量方面的问题。

8.2 表达式

表达式:
        赋值表达式
        赋值表达式 , *表达式*

先计算“,”左面的操作数，然后计算右面的操作数。表达式的类型是右面的操作数的类型，表达式的结果是右面的操作数的结果。

8.3 赋值表达式

赋值表达式:
        条件表达式
        条件表达式 = *赋值表达式*
        条件表达式 += *赋值表达式*
        条件表达式 -= *赋值表达式*
        条件表达式 *= *赋值表达式*
        条件表达式 /= *赋值表达式*
        条件表达式 %= *赋值表达式*
        条件表达式 &= *赋值表达式*
        条件表达式 |= *赋值表达式*
        条件表达式 ^= *赋值表达式*
        条件表达式 ~= *赋值表达式*
        条件表达式 <<= *赋值表达式*

```
        条件表达式 >>= 赋值表达式
        条件表达式 >>>= 赋值表达式
```

右面操作数的类型会被隐式地转换为左面操作数的类型，并赋给左面的操作数。结果的操作数的类型是左值得类型，结果的值是赋值后左值的值。

左面的操作数必须是左值。

8.3.1 赋值运算符表达式

赋值运算符表达式，如：

```
a op= b
```

在语义上等价于：

```
a = a op b
```

差别是操作数 *a* 只计算一次。

8.4 条件表达式

```
条件表达式:
        OrOr 表达式
        OrOr 表达式 ?表达式 :条件表达式
```

第一个表达式会被转换为布尔型，并被计算。如果结果为真，就计算第二个表达式，所得到的结果就是条件表达式的结果。如果结果为假，就计算第三个表达式，所得到的结果就是条件表达式的结果。如果第二个或者第三个表达式是 void 型的，返回的类型就是 void 。否则，第二个和第三个表达式的类型会被隐式地转换为一个它们的公共类型，并成为条件表达式结果的类型。

8.5 OrOr 表达式

```
OrOr 表达式:
        AndAnd 表达式
        OrOr 表达式 || AndAnd 表达式
```

OrOr 表达式 的返回类型是布尔型，除非右边的操作数类型为 void，这时结果类型将是 void。

OrOr 表达式 计算它左面的操作数。 如果左面操作数在转型到布尔型后，结果为真，就不会计算右面的操作数。如果 *OrOr 表达式* 结果的类型是布尔型，则表达式的结果就为真。如果左面操作数为假，就计算右面的操作数。如果 *OrOr 表达式* 是布尔型的，则结果就是右面的操作数转型为布尔型后的结果。

8.6 AndAnd 表达式

```
AndAnd 表达式:
        Or 表达式
        AndAnd 表达式 && Or 表达式
```

AndAnd 表达式 的返回类型是布尔型，除非右边的操作数类型为 void，这时结果类型将是 void。

AndAnd 表达式 计算它左面的操作数。

如果左面操作数在转型到布尔型后，结果为假，就不会计算右面的操作数。如果 *AndAnd 表达式* 结果的类型是布尔型，则表达式的结果就为假。

如果左面操作数为真，就计算右面的操作数。如果 *AndAnd 表达式* 是布尔型的，则结果就是右面的操作数转型为布尔型后的结果。

8.7 Bitwise 表达式

Bit wise（按位）表达式对它们的操作数 bitwise（按位）运算。 它们的操作数必须是正数类型的。首先，会应用默认的正数提升。然后，执行按位运算。

8.7.1 Or 表达式

```
Or 表达式:
        Xor 表达式
        Or 表达式 | Xor 表达式
```

操作数之间执行‘或’运算。

8.7.2 Xor 表达式

```
Xor 表达式:
        And 表达式
        Xor 表达式 ^ And 表达式
```

操作数之间执行‘异或’运算。

8.7.3 And 表达式

```
And 表达式:
        Cmp 表达式
        And 表达式 & Cmp 表达式
```

操作数之间执行‘与’运算。

8.8 比较表达式

```
And 表达式:
```

```
        相等表达式
        同一表达式
        关系表达式
        In 表达式
```

8.9 相等表达式

```
相等表达式:
        移位表达式
        移位表达式 == 移位表达式
        移位表达式 != 移位表达式
        移位表达式 is 移位表达式
        移位表达式 !is 移位表达式
```

相等表达式比较相等运算符 (==) 或不等运算符 (!=) 的两个操作数。结果的类型是布尔型。在比较之前，操作数会通过常用的转换转型为它们的一个公共类型。

如果操作数是整数值或者指针，相等被定义按位精确匹配。结构的相等是指按位精确匹配（检查包括由于对齐而造成的空洞，通常它们会在初始化时被设为零）。浮点数的相等更复杂。-0 和 +0 相等。如果两个操作数都是 NAN，则 == 和 != 运算都会返回假。其他情况下，按位比较相等性。

对于复数来说，相等被定义为等价于：

```
x.re == y.re && x.im == y.im
```

不相等被定义为等价于：

```
x.re != y.re || x.im != y.im
```

对于类和结构对象而言，表达式 (a == b) 被写成 a.opEquals(b)，而 (a != b) 被写成 !a.opEquals(b)。

对于类对象，== 和 != 运算符比较的这些对象的内容。因此，如果跟 null 进行比较则是无效的，因为 null 并没有内容。请使用 is 他 !is 运算符来代替。

```
class C;
C c;
if (c == null)  // 出错
  ...
if (c is null)  // 正确
  ...
```

至于静态和动态数组，相等被定义为数组长度相等，并且对应的元素都相等。

8.9.1 同一表达式

```
同一表达式:
```

```
        移位表达式 is 移位表达式
        移位表达式 !is 移位表达式
```

is 用于比较同一性。如果要比较非同一性，使用 *e1* **!is** *e2*。结果的类型是布尔型。在比较之前，操作数会通过常用的转换转型为它们的一个公共类型。

对于类对象来说，同一性被定义为引用指向同一个对象。可以使用 **is** 比较 null 类对象。

对于结构对象，同一性被定义为在结构里的位是同一的。

对于静态和动态数组来说，同一性被定义为所指向的数组拥有相同的数组元素以及相同的元素数目。

对于其它操作数类型，同一性被定义成相等性。

同一运算符 **is** 不能被重载。

8.10 关系表达式

```
关系表达式:
        移位表达式
        移位表达式 < 移位表达式
        移位表达式 <= 移位表达式
        移位表达式 > 移位表达式
        移位表达式 >= 移位表达式
        移位表达式 !<>= 移位表达式
        移位表达式 !<>移位表达式
        移位表达式 <> 移位表达式
        移位表达式 <>= 移位表达式
        移位表达式 !>移位表达式
        移位表达式 !>= 移位表达式
        移位表达式 !<移位表达式
        移位表达式 !<= 移位表达式
```

首先，会对操作数应用整数提升。关系表达式返回结果的类型是布尔型。

对于类对象来说，Object.cmp() 的结果构成了左操作数，0 构成了右操作数。关系表达式 (o1 op o2) 的结果是：

```
(o1.opCmp(o2) op 0)
```

如果对象为 **null**，进行比较就是错误。

对于静态和动态数组来说，关系 op 的结果是操作符应用到数组中第一个不相等的元素的结果。如果两个数组相等，但是长度不同，较短的数组“小于”较长的数组。

8.10.1 整数比较

如果两个操作数都是整数类型，就会进行整数比较。

整数比较运算符

运算符	关系
<	小于

>	大于
<=	小于等于
>=	大于等于
==	相等
!=	不相等

如果 <、<=、> 或 >= 表达式中一个操作数为有符号数，另一个操作数为无符号数，会被视为错误。可以使用类型转换将两个操作数转型为同是有符号数或同是无符号数。

8.10.2 浮点数比较

如果操作数中有浮点数，则执行浮点数的比较操作。

任何有实用价值的浮点操作都必须支持 NAN 值。尤其关系运算符要支持 NAN 操作数。浮点数的关系运算的结果可以是小于、大于、等于或未定义（未定义的意思是有操作数为 NAN ）。这意味着有 14 可能的比较条件：

浮点数比较运算符

运算符	大于	小于	等于	未定义	异常	关系
==	F	F	T	F	否	相等
!=	T	T	F	T	否	未定义、小于或大于
>	T	F	F	F	是	大于
>=	T	F	T	F	是	大于等于
<	F	T	F	F	是	小于
<=	F	T	T	F	是	小于等于
!<>=	F	F	F	T	否	未定义
<>	T	T	F	F	是	小于或大于
<>=	T	T	T	F	是	小于，等于或大于
!<=	T	F	F	T	否	未定义或大于
!<	T	F	T	T	否	未定义、大于或等于
!>=	F	T	F	T	否	未定义或小于
!>	F	T	T	T	否	未定义、小于或等于
!<>	F	F	T	T	否	未定义或等于

8.10.2.1 注解:

1. 对于浮点数比较运算符，(*a* !*op* *b*) 跟 !(*a* *op* *b*) 是不一样的。

2. “未定义”的意思是有操作数为 NAN 。
3. “异常”的意思是如果有操作数是 NAN ，则产生 *Invalid 异常* 。它并不是个抛出的异常。*Invalid 异常* 可以使用位于 std.c.fenv 里的函数来进行检查。

类比较

对于类对象，关系运算符会比较这些对象的内容。因此，如果跟 `null` 进行比较则是无效的，因为 `null` 并没有内容。

```
class C;
C c;
if (c < null)  // 出错
  ...
```

8.11 In 表达式

```
In 表达式:
        移位表达式 in 移位表达式
```

可以检测一个元素是否在关联数组中：

```
int foo[char[]];
...
if ("hello" in foo)
        ...
```

in 表达式同关系表达式 <, <= 等有相同的优先级。 *In 表达式* 的返回值是 **null**，如果元素不在数组中的话；如果它在数组中，则它是一个指向该元素的指针。

8.12 移位表达式

```
移位表达式:
        求和表达式
        关系表达式 << 求和表达式
        关系表达式 >> 求和表达式
        关系表达式 >>> 求和表达式
```

操作数必须是整数类型，并且会使用常用的整数提升。结果的类型是左操作数提升后的类型。结果的值是左操作数移动右操作数指定的位得到的值。

<< 是左移。>> 是有符号右移（译注：也叫算术右移）。>>> 是无符号右移（译注：也叫逻辑右移）。

如果要移动的位数超过了左操作数的位数，会被认为是非法的：

```
int c;
c << 33;        // 错误
```

8.13 求和表达式

求和表达式:
    求积表达式
    求和表达式 **+** *求积表达式*
    求和表达式 **-** *求积表达式*
    连接表达式

如果操作数是整数类型，会应用整数提升，然后会通过常用的算术转换提升为它们的公共类型。

如果有操作数为浮点类型，另一个操作数会被隐式地转换为浮点类型，然后会通过常用的算术转换提升为它们的公共类型。

如果运算符是 + 或 -，第一个操作数是指针，并且第二个操作数是整数类型，结果的类型就是第一个操作数的类型，结果的值是指针加上（或减去）第二个操作数乘以指针所指类型的大小得到的值。

如果第二个操作数是一个指针，而第一个操作数是一个整数，并且操作符是 +，则会按照上面所说的方式进行指针运算，只不过操作数的顺序反过来。

如果两个操作数都是指针，而且操作符是“+”，那么它就是非法的。对于“-”，这些指针就做减，结果会被操作数所指向的类型的大小整除。如果这些指针指向不同的类型则会出错。

浮点操作数的求和表达式不是可结合的。

8.14 连接表达式

连接表达式:
    求和表达式 **~** *求积表达式*

连接表达式 用于连接数组，最后生成一个动态数组。这些数组必须具有相同元素类型。如果有个操作数是一个数组，而另一个是那个数组元素类型的操作数，则该数会被转换成一个包含它的长度为 1 的数组，然后完成连接操作。

8.15 求积表达式

求积表达式:
    一元表达式
    求积表达式 ***** *一元表达式*
    求积表达式 **/** *一元表达式*
    求积表达式 **%** *一元表达式*

操作数必须是算术类型先会执行整数提升，然后会通过常用的算术转换提升为它们的公共类型。

于整数操作数来说，*、/ 和 % 对应于乘、除和取模运算。对于乘运算，会忽略溢出，结果会简单地截取为整数类型。如果除或者取模运算的右操作数为 0，会抛出一个

DivideByZero 异常。

对于 **%** 操作符的整型操作数，如果操作数为正，则结果符号为正；否则结果符号由具体实现定义。

对于浮点操作数来说，各种运算同对应的 IEEE 754 浮点运算相同。

浮点数的积表达式不是可结合的。

8.16 一元表达式

```
一元表达式:
        后缀表达式
        & 一元表达式
        ++ 一元表达式
        -- 一元表达式
        * 一元表达式
        - 一元表达式
        + 一元表达式
        ! 一元表达式
        ~ 一元表达式
        ( 类型 ) .标识符
        New 表达式
        Delete 表达式
        Cast 表达式
        New 匿名类表达式
```

8.16.1 New 表达式

```
New 表达式:
        New 参数 类型 [ 赋值表达式 ]
        New 参数 类型 ( 参数列表 )
        New 参数 类型
        New 参数 类参数 基类列表可选的 { 多个声明定义 }

New 参数:
        new ( 参数列表 )
        new ( )
        new

类参数:
        class ( 实参列表 )
        class ( )
        class

实参列表:
        赋值表达式
        赋值表达式 , 实参列表
```

New 表达式 用来在垃圾回收堆（默认情况）上分配内存，或者使用类指定的分配器分配内存。

在为多维数组分配内存时，声明按照同读取后缀数组声明顺序相同的顺序读取声明。

```
char[][] foo;    // 字符串动态数组
...
foo = new char[][30];    // 分配 30 个字符串数组
```

上面的分配也可以写成这样：

```
foo = new char[][](30); // 分配 30 个字符串数组
```

要分配嵌套的数组，可以使用多重参数：

```
int[][][] bar;
...
bar = new int[][][](5,20,30);
```

这个等同于：

```
bar = new int[][][5];
foreach (ref a; bar)
{
    a = new int[][20];
    foreach (ref b; a)
    {
        b = new int[30];
    }
}
```

如果有一个 **new (** *实参列表* **)**，那么这些参数在大小参数之后被传递给类或结构特有的分配函数。

如果 *New 表达式* 被用来作为函数里带有 **scope** 储存类别的局部变量的初始值，并且*实参列表*（传递给 **new** 的）为空，那么在堆栈上分配实例，而不是在堆实例进行分配，或者使用类特有的分配器。

8.16.2 Delete 表达式

```
Delete 表达式:
        delete 一元表达式
```

如果 *一元表达式* 是一个类对象引用，而且该类有析构函数，那么该析构函数就会对于该对象实例被调用。

其次，如果 *一元表达式* 是一个类对象引用，或者是一个指向结构实例的指针，并且类或结构已经重载了操作符 delete，那么操作符 delete 就会针对访类对象实例或结构实例被调用。

否则，垃圾回收器被调用来立即释放针对该类实例或结构实例分配的内存。如果垃圾回收没有被用来为该实例分配内存，那么就会导致未定义行为发生。

如果 *一元表达式* 是一个指针或动态数组，那么垃圾回收器会被调用来立即释放内存。如果

垃圾回收没有被用来为该实例分配内存，那么就会导致未定义行为发生。

指针、动态数组或者引用在完成 delete 操作后被设置成 **null**。

如果 *一元表达式* 是在堆栈 h 上分配的变量，则访类析构函数会针对该实例被调用（如果有必要的话）。垃圾回收器以及任何类的重新分配器都不会被调用。

8.16.3 Cast 表达式

Cast 表达式:
    cast (*类型*) *一元表达式*

Cast 表达式 将 *一元表达式* 转换到指定的 *类型*。

```
cast(foo) -p;    // 将 (-p) 转换到类型 foo
(foo) - p;       // 从 foo 减去 p
```

任何对于派生类引用的类型转换完成时，都会执行运行时检查以确保它是真的向下转换的。如果不是那样，则结果就为 **null**。**注意：**这个等同于 C++ 里的 dynamic_cast 操作符。

```
class A { ... }
class B : A { ... }

void test(A a, B b)
{
    B bx = a;           // 错误，需要类型转换
    B bx = cast(B) a;   // 如果 a 不是 B 类型，bx 就为 null
    A ax = b;           // 不需要类型转换
    A ax = cast(A) b;   // 不需要向上类型转换的运行时检查
}
```

如果想要检测一个对象 o 是否是类 B 的一个实例，可以使用类型转换：

```
if (cast(B) o)
{
// o 是 B 的一个实例
}
else
{
// o 不是 B 的一个实例
}
```

虽然将浮点文字从一种类型转换成另一种类型会改变它的类型，但是在内部它仍然被保留了完整的类型，为的是常量折叠(constant folding)。

```
void test()
{
   real a = 3.40483L;
   real b;
b = 3.40483;          // 文字没有被截断成双精度数(double precision)
   assert(a == b);
   assert(a == 3.40483);
   assert(a == 3.40483L);
   assert(a == 3.40483F);
   double d = 3.40483; // 当给变量赋值时截断文字
```

```
    assert(d != a);      // 因此它不再一样
    const double x = 3.40483; // 赋值给常量不会
    assert(x == a);            // 被截断，如果初始值是可见的话
}
```

把一个值 *v* 类型转换成一个结构 *S*，这样的操作当值不是同类型的结构时，就等同于：

```
S(v)
```

8.17 后缀表达式

```
后缀表达式:
        基本表达式
        后缀表达式 .标识符
        后缀表达式 .New 表达式
        后缀表达式 ++
        后缀表达式 --
        后缀表达式 ( )
        后缀表达式 ( 实参列表 )
        索引表达式
        分割表达式
```

8.18 索引表达式

```
索引表达式:
        后缀表达式 [ 实参列表 ]
```

后缀表达式 会被计算。 如果 *后缀表达式* 是一个静态或者动态类型的数组表达式，那么符号“$”就会被设置成该数组里元素的个数。 如果 *后缀表达式* 是一个 *表达式元组*，那么符号“$”就会被设置成该元组里元素的个数。 对于 *实参列表* 的求值会创建新的声明作用域，而且“$”只出现在该作用域中。

如果 *后缀表达式* 是一个 *表达式元组*，那么 *实参列表* 必须只有一个参数组成，而且必须是静态可求值到一个整型常量。整形常量 *n* 然后选择第 *n* 个表达式（位于 *表达式元组* 里），它就是 *Index 表达式* 的结查。如果 *n* 超出 *±í´ïÊ½ 元组* 的范围，则会出现一个错误。

8.19 分割表达式

```
分割表达式:
        后缀表达式 [ ]
        后缀表达式 [ 赋值表达式 ..赋值表达式 ]
```

后缀表达式 会被计算。如果 *后缀表达式* 的类型为静态或者动态数组，会隐式地声明变量 **length**（以及特殊符号 $），并将数组的长度赋给它。对于 *赋值表达式..赋值表达式* 的求

值会创建新的声明作用域，而且 **length**（以及 $）只出现在该作用域中。

第一个 *赋值表达式* 被规定包含分割块的下界， 而第二个 *赋值表达式* 则超出上界。表达式的结果就是 *后缀表达式* 数组一个分割部分。

如果使用了 **[ ]** 形式，则该分割部分就是整个数组。

分割部分的类型是动态数组，而此动态数组元素的类型就是 *后缀表达式* 的元素类型。

如果 *后缀表达式* 是一个 *表达式元组*，那么分割部分的结果就是一个新的 *表达式元组*（由上界和下界组成），此新的表达式必须静态求值成整型常量。如果这些边界超出范围，就会出一个错误。

8.20 基本表达式(Primary Expressions)

```
基本表达式:
        标识符
        .标识符
        this
        super
        null
        true
        false
        $
        __FILE__
        __LINE__
        数字字法
        字符文字
        多个字符串文字
        数组字法
        关联数组字法
        函数字法
        Assert 表达式
        Mixin 表达式
        Import 表达式
        基本类型 .标识符
        typeid ( 类型 )
        Is 表达式
        ( 表达式 )
        Traits 表达式
```

8.20.1 .标识符

标识符 会在模块作用域内进行查找，而不是在当前词法的嵌套作用域内。

8.20.2 this

在非静态成员函数内，**this** 转换成对调用此函数的对象的引用。如果此对象是一个结构的实例，则么 **this** 将会是该实例的指针。如果成员函数是显式地通过引用 **typeof(this)** 调用的，会生成一个非虚函数调用：

```
class A
{
```

```
    char get() { return 'A'; }

    char foo() { return typeof(this).get(); }
    char bar() { return this.get(); }
}

class B : A
{
    char get() { return 'B'; }
}

void main()
{
    B b = new B();

    b.foo();            // 返回 'A'
    b.bar();            // 返回 'B'
}
```

8.20.3 super

super 等同于 **this**，除了它被类型转换成 **this** 的基类。如果不存在相应的基类，就会被认为是错误。在结构成员函数里使用 **super** 是错误的。（只有 Object 类没有基类。）**super** 不允许出现在结构的成员函数中。如果成员函数是显式地通过引用 **super** 调用的，会生成一个非虚函数调用。

8.20.4 null

关键字 **null** 用于表示指针、函数指针、委托、动态数组、关联数组和类对象的空值。如果它还没有被转换到某个类型，它就会被指定类型 (void *)，而且对于指针、函数指针、委托等等，将其转换成 null 值都是准确的转换。在它被转换到某个类型后，这个转换是隐式的，但不再准确。

8.20.5 true, false

它们是 **bool** 类型，在转换到整数类型时，它们相应地变成 1 和 0。

8.20.6 字符字法(Character Literals)

字符字法指的是单个字符，并且类型是 **char**、**wchar** 或者 **dchar** 中的某一个。如果文字是一个 \u 转义序列，对应类型就是 **wchar**。如果文字是一个 \U 转义序列，对应类型就是 **dchar**。否则，它的类型是能够容纳它的最小的类型。

8.20.7 字符串字法(String Literals)

```
多个字符串文字:
        单个字符串文字
```

多个字符串文字 *单个字符串文字*

字符串文字可以隐式转换成下面类型，它们拥有相同的权重：

immutable(char)*
immutable(wchar)*
immutable(dchar)*
immutable(char)[]
immutable(wchar)[]
immutable(dchar)[]

字符串文字尾部会被追加一个 0，这样的话，它们就可以轻易地传递给期望是 const char* 类型串的 C 或 C++ 函数。 这个 0 不会被包括进字符串文字的 .length 特性。

8.20.8 数组字法

数组字法:
[*实参列表* **]**

数组字法指的是一个处于方括号 [和] 之间，以逗号隔开的 *赋值表达式* 列表。*赋值表达式* 组成了静态数组的元素，数组的长度就是元素的数目。第一个元素的类型代表了所有元素的类型，并且所有元素隐式转换成该类型。如果该类型是一个静态数组，则它就被转换成一个动态数组。

```
[1,2,3];        // 类型是 int[3]，有元素： 1，2 和 3
[1u,2,3];       // 类型是 uint[3]，有元素：1u，2u 和 3u
```

如果在 *实参列表* 里的参数有一个是 *表达式元组*，那么该 *表达式元组* 的元素将被插入进来代替元组而成为参数。

数组文字被分配在内存受控堆上。因此，它们可以安全由函数返回：

```
int[] foo()
{
    return [1, 2, 3];
}
```

当数组文字转换成另一个数组类型时，该数组的每一个元素也会转换成新的元素类型。当数组不是进行文字的类型转换时，该数组会被认定为一个新的类型，而其长度也会被重新计算：

```
import std.stdio;

void main()
{
// 转换数组文字
  const short[] ct = cast(short[]) [cast(byte)1, 1];
writeln(ct);  // 输出 [1 1]

// 转换其它数组表达式
```

```
  short[] rt = cast(short[]) [cast(byte)1, 1].dup;
writeln(rt);  // 输出 [257]
}
```

8.20.9 关联数组字法

关联数组字法:
        [*多个键值对* **]**

多个键值对:
        单个键值对
        单个键值对 **,** *多个键值对*

单个键值对:
        键表达式 **:***值表达式*

键表达式:
        条件表达式

值表达式:
        条件表达式

关联数组字法指的是一个处于方括号 [和] 之间，以逗号隔开的“*键:值*”对列表。此列表不能为空。第一个键的类型代表了所有键的类型，而其它剩余的所有键都会隐式转换成该类型。第一个值的类型代表了所有键的类型，而其它剩余的所有值都会隐式转换成该类型。*关联数组字法* 不能用于静态初始化。

```
[21u:"he",38:"ho",2:"hi"]; // 类型为 char[2][uint]，同时键为 21u, 38u 和 2u
                           // 而值为 "he", "ho" 和 "hi"
```

如果在 *键值对* 里的有一个键或值是一个 *表达式元组*，那么这个 *表达式元组* 的元素将被插入进来代替该元组而成为参数。

8.20.10 函数字法

函数字法:
        function *类型*$_{\text{可选的}}$ *参数属性* $_{\text{可选的}}$ *函数体*
        delegate *类型*$_{\text{可选的}}$ *参数属性* $_{\text{可选的}}$ *函数体*
        参数属性 *函数体*
        函数体

参数属性:
        形式参数
        多个形式参数 *多个函数属性*

有了 *函数字法*，就可以直接将匿名函数和匿名委托嵌入到表达式中。*类型*是函数或委托的

返回类型，如果被忽略的话，则会根据 *返回语句*（位于 *函数体*里）来进行推断。**(** *参数列表* **)** 是传递给函数的参数。如果忽略的话，会被认为是空参数列表 **()**。函数字法的类型是指向函数或者委托的指针。如果忽略关键字 **function** 或者 **delegate**，则默认是 delegate。

例如：

```
int function(char c) fp;         // 声明函数指针

void test()
{
    static int foo(char c) { return 6; }

    fp = &foo;
}
```

等同于：

```
int function(char c) fp;

void test()
{
    fp = function int(char c) { return 6;} ;
}
```

而：

```
int abc(int delegate(long i));

void test()
{   int b = 3;
    int foo(long c) { return 6 + b; }

    abc(&foo);
}
```

等同于：

```
int abc(int delegate(long i));

void test()
{   int b = 3;

    abc( delegate int(long c) { return 6 + b; } );
}
```

而下面的内容则是推断出返回类型为 **int**：

```
int abc(int delegate(long i));

void test()
{   int b = 3;

    abc( (long c) { return 6 + b; } );
}
```

匿名委托的行为就像任意的语句文字。例如，下面的 loop 可以执行任何语句：

```
double test()
{   double d = 7.6;
```

```
    float f = 2.3;

    void loop(int k, int j, void delegate() statement)
    {
        for (int i = k; i < j; i++)
        {
            statement();
        }
    }

    loop(5, 100, { d += 1; } );
    loop(3, 10,  { f += 3; } );

    return d + f;
}
```

与 嵌套函数 相比，**function** 形式类似于静态或者非嵌套函数，而 **delegate** 形式类似于非静态嵌套函数。换句话说，委托字法可以访问它外围函数的堆栈，而函数字法则不能。

8.20.11 断言表达式

```
断言表达式:
        assert ( 赋值表达式 )
        assert ( 赋值表达式 , 赋值表达式 )
```

断言会计算 *表达式*。如果结果为假，会抛出一个 **AssertError** 异常。如果结果为真，不会抛出任何异常。如果 *表达式* 包含程序所依赖的任何副作用，就是一个错误。通过编译时的命令行选项，编译器可以根本不对断言表达式求值。断言表达式的结果的类型是 `void`。断言是 D 支持 契约式编程 的一个基础。

表达式 `assert(0)` 是一个特殊的情况；它表示它是不可达到的代码。如果它是可达到，则会抛出一个 **AssertError** 异常，或者执行过程被中断（在 x86 处理器里，**HLT** 指令可以用来中断执行过程）。编程的优化和代码生成阶段可以假定它是不可达到的代码。

第二个 *表达式*，如果有的话，必须隐式可转换成类型 `char[]`。如果结果为 false，它就会被求值，而字符串结果会被添加到 **AssertError** 的信息里。

```
void main()
{
    assert(0, "an" ~ " error message");
}
```

当编译完成并且执行时，它会产生这样的信息：

```
Error: AssertError Failure test.d(3) an error message
```

8.20.12 Mixin 表达式

Mixin 表达式:
mixin (*赋值表达式*)

赋值表达式 必须在编译时求值成一个常量字符串。该字符串的文本内容必须要可编译成一个有效的*赋值表达式*，而且同样地被编译。

```
int foo(int x)
{
return mixin("x + 1") * 7;  // 跟 ((x + 1) * 7) 一样
}
```

8.20.13 Import 表达式

Import 表达式:
import (*赋值表达式*)

赋值表达式 必须在编译时求值成一个常量字符串。字符串的内容文本内容被认定为一个文件名。文件会被读取，而此文件的实际内容会变成字符串文字。

```
void foo()
{
//输出文件 foo.txt 的内容
   writefln( import("foo.txt") );
}
```

8.20.14 Typeid 表达式

Typeid 表达式:
typeid (*类型*)

返回一个 **TypeInfo**（同 *类型* 相对应）类的实例。

8.20.15 Is 表达式

Is 表达式:
is (*类型*)
is (*类型* :*类型特例*)
is (*类型* == *类型特例*)
is (*类型* *标识符*)
is (*类型* *标识符* :*类型特例*)
is (*类型* *标识符* == *类型特例*)
is (*类型* *标识符* :*类型特例* , *模板参数列表*)
is (*类型* *标识符* == *类型特例* , *模板参数列表*)

类型特例:
类型
typedef
struct
union

```
class
interface
enum
function
delegate
super
const
interface
```

Is 表达式 在编译时进行求值，并且用于检查合法的类型、比较类型的等效性、决定某种类型可否隐式转换到了另一种类型，以及减少类型的字类型。*Is 表达式* 的结果就是一个 int 类型的：如果条件不满足，就为 0；如果满足，则为 1。

类型 指的是要被测试的类型。它必须语句正确，不过不必语义正确。如果语义不正确，则条件就不满足。

标识符 在条件满足的情况下被声明为结果类型的别名。仅能使用 *标识符* 形式的情形就是如果 *Is 表达式* 出现在 *StaticIf 条件* 里。

类型特例(TypeSpecialization) 指的是要跟 *类型(Type)* 进行比较的类型。

Is 表达式 的形式有：

1. **is (** *类型* **)**
   如果 *类型* 语义正确（要求语句正确无关紧要），则条件满足。

```
alias int func(int);     // func 是函数类型的别名
void foo()
{
    if ( is(func[]) )    // 不满足，因为函数的数组
                         // 是不被允许的
        writefln("satisfied");
    else
        writefln("not satisfied");

    if (is([][]))        // 错误，[][] 不是个语句有效的类型
        ...
}
```

2. **is (** *类型* **:***类型特例化* **)**
   如果 *类型* 语义正确，则条件满足；它等同于或者能够被隐式转换成 *类型特列化*。*类型特例* 仅允许是一个 *类型*。

```
alias short bar;
void foo(bar x)
{
    if ( is(bar : int) )          // 满足，因为 short 能够
                                  // 隐式转换到 int
        writefln("satisfied");
    else
```

```
        writefln("not satisfied");
}
```

3. **is (** *类型* **==** *类型特例化* **)**

   如果 *类型* 语义正确，则条件满足；它跟 *类型特例化* 同类型。

   如果 *类型特例* 属于 **typedef**、**struct**、**union**、**class**、**interface**、**enum**、**function**、**delegate**、**const** 或 **invariant** 中的某一个， 那么如果 *类型* 也是其中之一的话，则条件满足。

```
alias short bar;
typedef char foo;
void test(bar x)
{
    if ( is(bar == int) )        // 不满足，因为 short 跟
                                 // int 不同类型
        writefln("satisfied");
    else
        writefln("not satisfied");

if ( is(foo == typedef) ) // 满足，因为 foo 是一个 typedef
        writefln("satisfied");
    else
        writefln("not satisfied");
}
```

4. **is (** *类型* *标识符* **)**

   如果 *类型* 语义正确，则条件满足。这样的话，*标识符* 被声明成为 *类型* 的一个别名。

```
alias short bar;
void foo(bar x)
{
    static if ( is(bar T) )
        alias T S;
    else
        alias long S;
writefln(typeid(S));   // 输出 "short"

if ( is(bar T) )  // 错误，标识符 T 形式仅能
                  // 在 StaticIf条件 里面
        ...
}
```

5. **is (** *类型* *标识符* **:***类型特例* **)**

   如果 *类型* 是 *类型特例*；或者如果 *类型* 是一个类并且 *类型特例* 是它的一个基类或基接口，则条件满足。*标识符* 要被声明成 *类型特例* 的别名，或者声明成导出的类型（如果 *类型特例* 依赖于 *标识符* 的话）。

```
alias int bar;
alias long* abc;
void foo(bar x, abc a)
{
    static if ( is(bar T : int) )
```

```
        alias T S;
    else
        alias long S;

    writefln(typeid(S));         // 输出 "int"

    static if ( is(abc U : U*) )
        U u;

    writefln(typeid(typeof(u)));         // 输出 "long"
}
```

标识符 的类型被决定的方式类似于通过 *模板参数特例化* 来决定的模板参数类型。

6. **is (** *类型* *标识符* == *类型特例* **)**

   如果 *类型* 语义正确，并且等于 *类型特例*，则条件满足。*标识符* 要被声明成 *类型特例* 的别名，或者声明成导出的类型（如果 *类型特例* 依赖于 *标识符* 的话）。

   如果 *类型特例* 属于 **typedef**、**struct**、**union**、**class**、**interface**、**enum**、**function**、**delegate**、**const** 或 **invariant** 中的某一个， 那么如果 *类型* 也是其中之一的话，则条件满足。另外，*标识符* 会被设置成该类型的一个别名：

关键字	*标识符* 的别名类型
typedef	通过 typedef 定义出的 *类型*
struct	*类型*
union	*类型*
class	*类型*
interface	*类型*
super	基类和接口*类型元组*
enum	枚举类型
function	函数参数类型的 *类型元组*
delegate	委托函数类型
return	函数、委托或者函数指针返回类型
const	*类型*
invariant	*类型*

```
alias short bar;
enum E : byte { Emember }
void foo(bar x)
{
static if ( is(bar T == int) ) // 不满足，short 不是 int
       alias T S;
```

```
    alias T U;                          // 错误，T 没有被定义

static if ( is(E V == enum) )  // 满足，E 是一个 enum
       V v;                            // v 被声明为一个字节
}
```

例如，要测试看 X 是否是一个 typedef 以及其基类型是否为 int：

```
typedef int X;

static if (is(X base == typedef))
{
    static assert(is(base == int), "base of typedef X is not int");
}
else
{
    static assert(0, "X is not a typedef");
}
```

7. **is (** *类型 标识符* **:***类型特例化* **,** *模板参数列表* **)**
   is (*类型 标识符* **==** *类型特例化* **,** *模板参数列表* **)**

   更多复杂类型可以进行模式匹配；*模板参数列表* 用于声明基于匹配到的那部分模式的符号，等同于表明模板参数被匹配到了。

```
import std.stdio;

void main()
{
  alias long[char[]] AA;

  static if (is(AA T : T[U], U : const char[]))
  {
    writefln(typeid(T));        // long
    writefln(typeid(U));        // const char[]
  }

  static if (is(AA A : A[B], B : int))
  {
assert(0);  // 不应该匹配，因为 B 不是 int
  }

  static if (is(int[10] W : W[V], int V))
  {
    writefln(typeid(W));        // int
    writefln(V);                // 10
  }

  static if (is(int[10] X : X[Y], int Y : 5))
  {
    assert(0);  // 不应该匹配，Y 应该为 10
  }
}
```

8.21 结合律(Associativity)和交换律(Commutativity)

一种实现可能会根据数学上的结合律和交换律重新排列求值表达式，只要是在该执行的线程内，就不会有什么特别的不同。

此规则不包含在重排浮点表达式时的结合律或交换律。

第 9 章 语句

C 和 C++ 程序员会发现 D 中的语句很熟悉，同时还有一些有意的扩充。

语句:
    ;
非空语句
作用域语句块

无作用域非空语句:
非空语句
块语句

无作用域语句:
    ;
非空语句
块语句

非空或无作用域语句块:
非空语句
作用域语句块

非空语句:
标号语句
表达式语句
声明语句
If 语句
条件语句
While 语句
Do 语句
For 语句
Foreach 语句
Switch 语句
Case 语句
Default 语句
Continue 语句
Break 语句
Return 语句
Goto 语句
With 语句
Synchronized 语句
Try 语句
ScopeGuard 语句
Throw 语句
Asm 语句
Pragma 语句
Mixin 语句
Foreach 范围语句

9.1 作用域(scope)语句

作用域语句:
非空语句
块语句

对于 *非空语句* 或 *块语句* 会引入新的用于局部符号的作用域。

即使会引入一个新的作用域，局部符号的声明也不能掩盖（隐藏）在同一函数里的其它的局部符号声明。

```
void func1(int x)
{   int x;       // 非法，x 掩盖了参数 x

    int y;

    { int y; }  // 非法，y 掩盖了封闭域里的 y

    void delegate() dg;
    dg = { int y; };     // 正确，这个 y 不在同一个函数里

    struct S
    {
        int y;           // 正确，这个 y 是个成员，不是个局部变量
    }

    { int z; }
    { int z; }  // 正确，这个 y 没有掩盖其它的 z

    { int t; }
    { t++;   }  // 非法，t 未定义
}
```

此思想用于避免在复杂函数里由于限定域里的声明无意识情况隐藏了前面的声明而引起的错漏。在一个函数里的那些局部变量名应该全部唯一。

9.2 作用域块语句

作用域块语句:
块语句

作用域块语句会为 *块语句* 引入新的作用域。

9.3 标号语句

语句可以有标号。标号是一种标志符，其后跟随一条语句。

标号语句:

标识符: *无 Scope 语句*

包括空语句在内的任何语句都可以有标号，因此都可以作为 goto 语句的目标。标号语句也可以用作 break 或者 continue 语句的目标。

标号位于独立的名字空间中，不与声明、变量、类型等位于同一名字空间。就算如此，标号也不能同局部声明重名。标号的名字空间是它们所在的函数体。标号的名字空间不嵌套，也就是说，一个语句块中的标号可以在语句块外访问。

9.4 块语句

块语句:
    { }
{ *语句列表* **}**

语句列表:
语句
语句 *语句列表*

语句块是由{ }包括起来的语句序列。其中的语句按照词法顺序执行。

9.5 表达式语句

表达式语句:
表达式 **;**

表达式会被计算。

没有作用的表达式，如 (x + x)，在表达式语句中是非法的。如果需要这个表达式，那么将它的类型转换为 void 就可以让它合法。

```
int x;
x++;                    // 正确
x;                      // 非法
1+1;                    // 非法
cast(void)(x + x);      // 正确
```

9.6 声明语句

声明语句声明并初始化变量。

声明语句:
单个声明

一些声明语句:

```
int a;              // 把 a 声明成类型 int，并把它初始化为 0
struct S { }        // 声明结构 S
alias int myint;
```

9.7 If 语句

If 语句提供了按条件执行语句的方法。

```
If 语句:
        if ( If 条件 ) Then 语句
        if ( If 条件 ) Then 语句 else Else 语句

If 条件:
        表达式
        auto 标识符 = 表达式
        单个声明符 = 表达式

Then 语句:
        作用域语句

Else 语句:
        作用域语句
```

表达式 将被计算，计算的结果必须可以被转换为布尔型。如果它为 true，则转换到 *Then 语句*，否则就转换到 *Else 语句*。

“虚悬的(dangling) else”问题可以通过使用最近的那个 if 语句关联到该 else 来解决。

如果提供有一个 **auto** *标识符*，那么它就会被声明并被初始化成该*表达式*的值和类型。它的作用范围从被初始化开始一直到 *Then 语句* 的结尾。

如果提供有一个 *声明符*，那么它就会被声明并被初始化成该*表达式*的值。它的作用范围从被初始化开始一直到 *Then 语句* 的结尾。

```
import std.regexp;
...
if (auto m = std.regexp.search("abcdef", "b(c)d"))
{
writefln("[%s]", m.pre);       // 输出 [a]
    writefln("[%s]", m.post);      // 输出 [ef]
    writefln("[%s]", m.match(0)); // 输出 [bcd]
    writefln("[%s]", m.match(1)); // 输出 [c]
    writefln("[%s]", m.match(2)); // 输出 []
}
else
{
writefln(m.post);     // 错误，m 未定义
}
writefln(m.pre);          // 错误，m 未定义
```

9.8 While 语句

```
While 语句:
while ( 表达式 ) 作用域语句
```

While 语句实现了简单的循环。 *表达式* 将被计算，计算的结果必须可以被转换为布尔型。如果它的结果为 true，执行*作用域语句*。在执行该 *作用域语句* 后，此 *表达式* 会被再次计算，并且如果为 true 的话，该 *作用域语句* 会被再次执行。这个过程会持续下去，直到 *表达式* 的计算出的结果为 false。

```
int i = 0;
while (i < 10)
{
    foo(i);
    i++;
}
```

Break 语句 将退出循环。*Continue 语句* 将直接转去再次计算 *表达式*。

9.9 Do 语句

Do 语句:
do *作用域语句* **while** **(** *表达式* **)**

do-While 语句实现了简单的循环。 先执行 *作用域语句*。然后 *表达式* 会被计算，同时要求计算的结果必须可以被转换为布尔型。如果结果为 true，则此循环会被重复一次。这个过程会持续下去，直到 *表达式* 的计算出的结果为 false。

```
int i = 0;
do
{
    foo(i);
} while (++i < 10);
```

Break 语句 将退出循环。*Continue 语句* 将直接转去再次计算 *表达式*。

9.10 For 语句

For 语句实现了带有初始化、测试和递增的循环结构。

For 语句:
for **(***初始化 测试; 递增***)** *作用域语句*

初始化:
;
无范围非空语句

测试:
空
表达式

递增:
空
表达式

先执行 *初始化*。然后计算 *测试* ，结果必须可以被转换为布尔型。如果结果为 true，执行语句。在执行语句后，将执行 *递增*。然后再次计算 *测试*，如果结果为 true 再次执行语句。持续这个过程直到 *测试* 的计算结果为 false 。

Break 语句 将退出循环。*Continue 语句* 将直接跳转到 *递增*。

ForStatement 会建立新的作用域。如果 *初始值* 声明了一个变量，变量的作用域持续到语句结束。例如：

```
for (int i = 0; i < 10; i++)
       foo(i);
```

等同于：

```
{    int i;
    for (i = 0; i < 10; i++)
       foo(i);
}
```

函数体不能为空：

```
for (int i = 0; i < 10; i++)
       ;        // 非法
```

正确的写法是：

```
for (int i = 0; i < 10; i++)
{
}
```

初始值 可以忽略。*测试* 也可以被忽略；如果忽略，就假定为 true 。

9.11 Foreach 语句

foreach 语句遍历一个聚集的全部内容。

```
Foreach 语句:
Foreach (Foreach 类型列表; 聚集) 无作用域非空语句

Foreach:
    foreach
    foreach_reverse

Foreach 类型列表:
Foreach 类型
Foreach 类型 , Foreach 类型列表

Foreach 类型:
ref 类型 标识符
类型 标识符
ref 标识符
标识符

聚集:
表达式
```

元组

聚集 会被计算。它的结果必须为静态数组、动态数组、关联数组、结构或类类型的聚集表达式。对于聚集表达式的每个元素执行一次 *无作用域非空语句*。在每次遍历开始时，*Foreach 类型列表* 声明的变量默认为聚集的内容的拷贝（即采用传值方式）。如果变量是 **ref**，这些变量就是聚集内容的引用（即采用传引用方式）。

聚集必须是循环不变量，即在 *无作用域非空语句* 里将聚集的元素不能被添加或移除。

9.11.1 foreach 用于数组

如果聚集表达式是静态或动态数组，可以声明一个或者两个变量。如果声明了一个变量，那么这个变量就被赋予数组元素的 *值*，依着次序地进行。除了下面提到的特殊情况以外，变量的类型必须同数组内容的类型相匹配，。如果声明了两个变量，第一个变量则对应该数组的 *索引*，而第二个对应该数组的 *值*。注意，该 *索引* 必须是 **int**、**uint** 或 **size_t** 类型，而不能是 *ref*，并且它的值就是数组元素的索引值。

```
char[] a;
...
foreach (int i, char c; a)
{
    writefln("a[%d] = '%c'", i, c);
}
```

对于 **foreach**，数组的元素会从索引 0 开始被迭代，一直到该数组的最大值。对于 **foreach_reverse**，则会以相反的顺序访问数组元素。

9.11.2 foreach 用于字符数组

如果聚集表达式是 **char**、**wchar** 或者 **dchar** 的静态或动态数组，则 *值* 的 *类型* 可以是 **char**、**wchar** 或者 **dchar** 中的任何一个。采用这种方式，所有的 UTF 数组都可以被解码为任意的 UTF 类型：

```
char[] a = "\xE2\x89\xA0";      // \u2260 encoded as 3 UTF-8 bytes

foreach (dchar c; a)
{
    writefln("a[] = %x", c);    // 输出 'a[] = 2260'
}

dchar[] b = "\u2260";

foreach (char c; b)
{
    writef("%x, ", c);  // prints 'e2, 89, a0, '
}
```

聚集可以是字符串文字，这样它们就可以被看作 char、wchar 或 dchar 数组来进行访问：

```
void test()
{
    foreach (char c; "ab")
    {
        writefln("'%s'", c);
```

```
    }
    foreach (wchar w; "xy")
    {
        writefln("'%s'", w);
    }
}
```

输出结果：

```
'a'
'b'
'x'
'y'
```

9.11.3 foreach 用于关联数组

如果聚集表达式是关联数组，可以声明一个或者两个变量。如果声明了一个变量，那么这个变量就被赋予数组元素的 *值*，依着次序地进行。变量的类型必须同数组内容的类型匹配。如果声明了两个变量，第一个变量则对应该数组的 *索引*，而第二个对应该数组的 *值*。*索引* 必须同关联数组的索引具有相同的类型。它不能是 *ref*，它的值是数组元素的索引。对于 **foreach**，数组元素的顺序是不明确的。而将 **foreach_reverse** 用于关联数组是非法的。

```
double[char[]] a;       // 索引 类型是 char[]，值 类型是 double
...
foreach (char[] s, double d; a)
{
    writefln("a['%s'] = %g", s, d);
}
```

9.11.4 foreach 用于带有范围的结构和类

可以使用范围(range)来迭代结构和类对象，即表示必须定义下列特性：

Foreach 范围特性

特性	目的
.empty	如果没有更多的元素，则返回真值 true
.next	将范围的左边界向右移动一个位置
.retreat	将范围的右边界向左移动一个位置
.head	返回范围的最左元素
.toe	返回范围的最右元素

原意：

```
foreach (e; range) { ... }
```

翻译后即为：

```
for (auto __r = range; !__r.empty; __r.next)
{   auto e = __r.head;
   ...
}
```

类似地：

```
foreach_reverse (e; range) { ... }
```

翻译后即为：

```
for (auto __r = range; !__r.empty; __r.retreat)
{   auto e = __r.toe;
   ...
}
```

如果 foreach 范围特性不存在，就使用 `opApply` 方法代替。

9.11.5 foreach 用于带有 opApply 的结构和类

如果它是一个结构或类对象，则 **foreach** 会通过特有的 *opApply* 成员函数会定义。**foreach_reverse** 的动作则由特有的 *opApplyReverse* 成员函数进行定义。为了使用相应的 foreach 语句，这些特有函数必须被该类型定义。这些函数拥有的类型：

```
int opApply(int delegate(ref Type [, ...]) dg);

int opApplyReverse(int delegate(ref Type [, ...]) dg);
```

这里的 *类型* 跟在 *标识符* 里的 *Foreach 类型* 声明所使用的 *类型* 相匹配。多重 *Foreach 类型* 跟在传递给 **opApply** 或 **opApplyReverse** 的委托类型里的多重 *类型* 相一致。也可以有多重 **opApply** 和 **opApplyReverse** 函数，通过将 *dg* 的类型跟 *Foreach 语句* 的 *Foreach 类型* 相匹配来进行一个一个的选择。应用函数体会迭代它所聚集的全部元素，并把它们每一个都传递给 *dg* 函数。如果 *dg* 返回 0，那么应用就继续下一个元素。如果 *dg* 返回一个非零值，apply 必须停止迭代并返回该值。否则，在迭代完所有的元素之后，apply 会返回 0 。

例如，假设有一个类，它是一个窗口，拥有两个元素：

```
class Foo
{
   uint array[2];

   int opApply(int delegate(ref uint) dg)
   {   int result = 0;

      for (int i = 0; i < array.length; i++)
      {
         result = dg(array[i]);
         if (result)
            break;
      }
      return result;
   }
```

```
}
```

使用这个的实例可能像这样：

```
void test()
{
    Foo a = new Foo();

    a.array[0] = 73;
    a.array[1] = 82;

    foreach (uint u; a)
    {
        writefln("%d", u);
    }
}
```

输出结果：

```
73
82
```

9.11.6 foreach 用于委托

如果 *聚集* 是一个委托，那么此委托的类型署名就跟用于 **opApply** 的一样。这使得许多不同命名的循环策略可以在相同的类或结构里共存。

9.11.7 foreach 用于元组

如果聚集是一个 tuple，那么就可能会声明一到两个变量。如果声明了一个变量，那么这个变量就被赋予该 Tupe 元素的 *值*，一个接一个地。如果给定了该变量的类型，则它必须跟 Tuple 内容里的类型相匹配。如果没有给定，则该变量的类型会被设置成该 Tuple 元素的类型，它可能从一个迭代变化到另一个迭代。如果声明了两个变量，第一个变量则对应该数组的 *索引*，而第二个对应该数组的 *值*。*索引* 必须是 **int** 或 **uint** 类型，它不能是 *ref*，并且它的值是该 Tuple 元素的索引。

如果该 Tuple 是一个类型列表，那么该 foreach 语句对于每一个类型都会被执行一次，并且该值会成该类型的别名。

```
import std.stdio;
import std.typetuple;    // 用于类型元组

void main()
{
    alias TypeTuple!(int, long, double) TL;

    foreach (T; TL)
    {
        writefln(typeid(T));
```

```
    }
}
```

输出

```
int
long
double
```

9.11.8 foreach 的 ref 参数

ref 可以被用来更新原有的元素：

```
void test()
{
    static uint[2] a = [7, 8];

    foreach (ref uint u; a)
    {
        u++;
    }
    foreach (uint u; a)
    {
        writefln("%d", u);
    }
}
```

输出结果：

```
8
9
```

ref 不能被应用到该索引值。

如果没有指定，在 *Foreach 类型* 里的 *类型* 可以从 *聚集* 的类型进行推断。

9.11.9 foreach 的局限性

在 foreach 迭代所有元素时，聚集本身一定不要被重新调整大小、重新分配、释放、重新赋值或重新析构。

```
int[] a;
int[] b;
foreach (int i; a)
{
    a = null;                       // 错误
    a.length = a.length + 10;       // 错误
    a = b;                          // 错误
}
a = null;                           // 正确
```

9.11.10 foreach 里的 break 和 continue

如果在 foreach 体内有 *Break 语句*，则会退出 foreach，而 *Continue 语句* 将立即开始下一

轮遍历。

9.12 Switch 语句

switch 语句依照 switch 表达式的值来选择一条 case 语句执行。

```
Switch 语句:
        switch ( 表达式 ) 作用域语句

Case 语句:
        case 表达式列表 :语句

Default 语句:
        default: 语句
```

表达式 会被计算。结果的类型 T 必须是整数类型或者 `char[]`、 `wchar[]` 或者 `dchar[]`。所得结果同各个 case 表达式进行比较。如果存在匹配，就转而执行相应的 case 语句。

case 表达式，也就是 *表达式列表*，是由逗号分隔的表达式的列表。

如果没有 case 表达式能够匹配，并且存在 default 语句，就转去执行 default 语句。

如果没有 case 表达式能够匹配，并且不存在 default 语句，那么就抛出异常 `std.switcherr.SwitchError`。之所以这么做，是为了捕获一些常见的错误，如为枚举添加了新值却忘记了为这个值提供对应的 case 语句。这同 C 和 C++ 不太一样。

case 表达式求值后必须是一个常数值或数组，或者运行时初始化的常量以及整型的 invariant 变量。 它们都必须隐式地转换成 switch *表达式* 的类型。

case 表达式的值必须互不相同。const 或 invariant 变量必须要有不同的名字。如果它们共享一个值，则第一个带有该值的 case 语句会获得控制。 而且不能有两个或者更多的 default 语句。

与 switch 相关联的 case 语句和 default 语句可以在语句块中嵌套；它们不必非要位于最外层的块中。例如，下面是合法的：

```
    switch (i)
    {
        case 1:
        {
            case 2:
        }
            break;
    }
```

case 语句会顺序执行后面的 case 子句。break 语句将退出 switch *块语句*。例如：

```
switch (i)
{
```

```
    case 1:
        x = 3;
    case 2:
        x = 4;
        break;

    case 3,4,5:
        x = 5;
        break;
}
```

会将 x 设置为 4，如果 i 为 1 的话。

注意： 同 C 和 C++ 不同的是，可以在 switch 表达式中使用字符串。例如：

```
char[] name;
...
switch (name)
{
    case "fred":
    case "sally":
        ...
}
```

对于如命令行选项处理这样的应用来说，采用这种语法的代码更直接，更清晰并且不易出错。ascii 和 wchar 都可以使用。

实现注意： 编译器的代码生成程序可以认为 case 语句已经按照使用的频率排序，频率高的排在最前面，频率低的排在最后。尽管这不会影响程序的正确性，但对于提高性能来说是有益的。

9.13 Continue 语句

```
Continue 语句:
    continue;
continue 标识符 ;
```

continue 会中当前断其封闭循环语句的迭代，然后开始下一次迭代。 continue 执行它所在最内层的 while、for 或者 do 语句的下次迭代。执行递增子句。

如果 continue 后跟有 *标识符*，则该 *标识符* 必须是它所在的 while、for 或者 do 语句外的标号，continue 会执行所在循环的下次迭代。如果不存在那样的语句，就是错误。

所有被跳过的 finally 子句都会执行，同时会释放所有被跳过的 synchronization 对象。

注意： 如果 finally 子句中有 return、throw 或者 goto 到 finally 子句之外的这样的动作，continue 的目标永远也无法达到。

```
for (i = 0; i < 10; i++)
{
    if (foo(i))
        continue;
    bar();
}
```

9.14 Break 语句

Break 语句:
break;
break *标识符* ;

break 退出它所在的语句。 break 退出它所在最内层的 while、for、do 或者 switch 语句，并在该语句之后恢复执行。

如果 break 后跟有 *标识符*，则该 *标识符* 是它所在的 while、for、do 或者 switch 语句外的标号，break 会退出那条语句。如果不存在那样的语句，就是错误。

所有被跳过的 finally 子句都会执行，同时会释放所有被跳过的 synchronization 对象。

注意： 如果 finally 子句中有 return、throw 或者 goto 到 finally 子句之外的这样的动作，break 的目标永远也无法达到。

```
for (i = 0; i < 10; i++)
{
   if (foo(i))
      break;
}
```

9.15 Return 语句

Return 语句:
return;
return *表达式* ;

return 语句的作用是推出当前的函数并提供一个返回值。 如果函数指定了非 void 的返回类型，就必须给出 *表达式*。*表达式* 会被隐式地转换为函数的返回类型。

如果函数指定了一个不为 void 的返回类型，则至少需要一个 return 语句、throw 语句 或 assert(0) 表达式。

就算函数的返回类形是 **void**，也可以有 *表达式* 存在。*表达式* 会被计算，但是什么也不会返回。

在函数实际返回之前，任何带有 scope 存储特性的对象都会被消除掉，所有的封闭 finally 子句会被执行，所有的 scope(exit) 语句会被执行，所有的 scope(success) 语句会被执行，并且所有的封闭 synchronization 对象会被释放。

如果外围的 finally 子句中有 return、goto 或者 throw 的话，函数就不会正常的返回。

如果存在后验条件（参见契约式编程）,则在计算 *表达式* 之后执行该后验条件，然后函数返回。

```
int foo(int x)
{
   return x + 3;
```

```
}
```

9.16 goto 语句

```
Goto 语句:
goto 标识符 ;
    goto default ;
    goto case ;
goto case 表达式 ;
```

一个 goto 会跳转到以 *标识符* 为标号的语句处。

```
    if (foo)
        goto L1;
    x = 3;
L1:
    x++;
```

第二种形式：`goto default;`，即跳转到最内层 *Switch 语句* 中的 *Default 语句* 处。

第三种形式：`goto case;`，即跳转到最内层 *Switch 语句* 中的下一个 *Case 语句* 处。

第四种形式：`goto case` **表达式;**，即跳转到跟封闭在 *Switch 语句* 里面的匹配该 *Expression* 的 *Case 语句* 处。

```
switch (x)
{
    case 3:
        goto case;
    case 4:
        goto default;
    case 5:
        goto case 4;
    default:
        x = 4;
        break;
}
```

所有的被跳过的 finally 子句都会执行，同时会释放所有的被跳过的同步互斥体。

使用 *Goto 语句* 来跳过初始化是非法的。

9.17 With 语句

with 语句用来简化对同一个对象的重复引用。

```
With 语句:
        with ( 表达式 ) 作用域语句
        with ( 符号 ) 作用域语句
        with ( 模板实例 ) 作用域语句
```

表达式 的结果是对类实例或者结构的引用。在 with 的过程体内，所有的标志符符号都将在

那个类引用的名字空间内查找。*With 语句*

```
with (expression)
{
    ...
    ident;
}
```

在语义上等价于：

```
{
    Object tmp;
    tmp = expression;
    ...
    tmp.ident;
}
```

注意 *表达式* 只会被计算一次。with 语句不会改变 **this** 或 **super** 引用的对象。

对于是一个作用域或者 *模板实例* 的 *符号*，在查找符号时相应的作用域会被搜索。例如：

```
struct Foo
{
    typedef int Y;
}
...
Y y;              // 错误，Y 未定义
with (Foo)
{
    Y y;          // 等同于 Foo.Y y
}
```

9.18 Synchronized 语句

synchronized 语句用来约束在多线程里带有同步互斥访问的语句。

```
Synchronized 语句:
synchronized 作用域语句
synchronized ( 表达式 ) 作用域语句
```

同步(synchronized)通过使用一个互斥量(mutex)只允许每次一个线程去执行“*作用域语句*”。

使用什么样的互斥量则由 *表达式* 来决定。如果没有*表达式*，那么就创建全局互斥量，每一个 synchronized 语句对应一个。不同的 synchronized 语句会有不同的全局互斥量。

如果有*表达式*，那么它就必须求值成一个对象或一个*接口*的实例，此时，它是实际是经过类型转换而成为实现该*接口*的对象实例的。使用的互斥量被指定到了该对象实例，然后被所有引用该实例的同步语句共享。

要是 *作用域语句* 由于异常、goto 或 return 被终止了，同步便会被释放掉。

样例:

```
synchronized { ... }
```

这实现一个标准的临界区。

9.19 Try 语句

异常处理由 try-catch-finally 语句完成。

```
Try 语句:
        try 作用域语句 多个 Catch
        try 作用域语句 多个 Catch Finally 语句
        try 作用域语句 Finally 语句

多个 Catch:
        最后的 Catch
        Catch
        Catch 多个 Catch

最后的 Catch:
        catch 无作用域非空语句

Catch:
        catch ( Catch 参数 ) 无作用域非空语句

Catch 参数:
        基本类型 标识符

Finally 语句:
        finally 无作用域非空语句
```

Catch 参数 声明类型为 T 的变量 v，这里的 T 是 Object 或继承自 Object。v 由抛出的表达式进行初始化，如果 T 具有跟抛出的表达式相同的类型或是其基类的话。如果异常对象的类型为 T 或者派生自 T，则该 catch 子句就会被执行。

如果只给定了类型 T 而没有变量 v，那么该 catch 依然会被执行。

如果有 *Catch 参数* 类型 T1 隐藏了后面类型为 T2 的 *Catch*，则会产生一个错误；例如，如果 T1 跟 T2 同类型或是 T2 的基类，则这就是一个错误。

最后的 Catch 会捕捉所有的异常。

Finally 语句 总是会被执行，无论是否 **try** *作用域语句* 由于 goto、break、continue、return、exception 或失败而退出。

如果一个异常在 *Finally 语句* 里产生了，而在执行 *Finally 语句* 之前没有被捕捉到，则原来的所有异常会被新的异常所替换：

```
import std.stdio;

int main()
{
    try
    {
```

```
        try
        {
            throw new Exception("first");
        }
        finally
        {
            writefln("finally");
            throw new Exception("second");
        }
    }
    catch(Exception e)
    {
        writefln("catch %s", e.msg);
    }
    writefln("done");
    return 0;
}
```

输出：

```
finally
catch second
done
```

Finally 语句 不可能因为 goto、break、continue 或 return 而退出；也不可以因为 goto 而进入。

Finally 语句 不可能包含任何的 *Catches*。在将来的版本中这个限制可以被放宽。

9.20 Throw 语句

抛出一个异常。

```
Throw 语句:
        throw 表达式 ;
```

表达式 被计算，结果必须是 Object 引用类型。该 Object 的引用被作为异常抛出。

```
throw new Exception("message");
```

9.21 作用域守护语句(Scope Guard Statement)

```
作用域守护语句:
        scope(exit) 非空或无作用域块语句
        scope(success) 非空或无作用域块语句
        scope(failure) 非空或无作用域块语句
```

作用域守护语句 在当前作用域的结束处执行 *非空或无作用域块语句*，而不是在 *作用域守护语句* 所在的那个地方执行。**scope(exit)** 会在正常退出该作用域时或在由于异常展开而退出时执行 *非空或无作用域块语句*。**scope(failure)** 会在由于异常展开退出该作用域时执行 *非空或无作用域块语句*。**scope(success)** 则会在正常退出该作用域时执行 *非空或无作用域块语句*。

如果在一个作用域内有多重 *作用域守护语句*，那么它们执行的顺序则跟它们在词法上出现的顺序相反。如果有 scope 实例在作用域结束处被析构，则它们会以词法呈现的相反顺序跟 *作用域守护语句* 交织在一起。

```
writef("1");
{
    writef("2");
    scope(exit) writef("3");
    scope(exit) writef("4");
    writef("5");
}
writefln();
```

输出：

```
12543

{
    scope(exit) writef("1");
    scope(success) writef("2");
    scope(exit) writef("3");
    scope(success) writef("4");
}
writefln();
```

输出：

```
4321

class Foo
{
    this() { writef("0"); }
    ~this() { writef("1"); }
}

try
{
    scope(exit) writef("2");
    scope(success) writef("3");
    scope Foo f = new Foo();
    scope(failure) writef("4");
    throw new Exception("msg");
    scope(exit) writef("5");
    scope(success) writef("6");
    scope(failure) writef("7");
}
catch (Exception e)
{
}
writefln();
```

输出：

```
0412
```

scope(exit) 或 **scope(success)** 语句不可能由于 throw、goto、break、continue 或 return 退出；也不可能由于 goto 而进入。

9.22 Asm 语句

asm 语句用来支持内联汇编：

```
Asm 语句：
        asm { }
        asm { Asm 指令列表 }

Asm 指令列表：
        Asm 指令 ;
        Asm 指令 ; Asm 指令列表
```

asm 语句使你可以直接使用汇编语言指令。这样不必费力地采用外部的汇编程序就可以获得对 CPU 特殊特征的直接访问能力。D 编译器会替你管理函数调用约定、堆栈设置等等恼人的事情。

指令的格式，当然，高度依赖于目标 CPU 的指令集，所以是由 实现定义 的。但是，格式需要遵循下述约定：

- 必须使用同 D 语言相同的记号。
- 注释的形式必须同 D 语言的相通。
- Asm 指令以‘;’结尾，而不是以行尾结尾。

这些规则保证 D 的源代码可以独立于语法或语义分析而被记号化。

例如，对于 Intel Pentium 处理器：

```
int x = 3;
asm
{
    mov EAX,x;          // 将 x 载入寄存器 EAX
}
```

内联汇编可以用来直接访问硬件：

```
int gethardware()
{
    asm
    {
            mov EAX, dword ptr 0x1234;
    }
}
```

对于某些 D 实现，如将 D 翻译为 C 的翻译器来说，内联汇编没有什么意义，也就不需要

实现它。version 语句可以用来应付这种情况：

```
version (D_InlineAsm_X86)
{
    asm
    {
        ...
    }
}
else
{
/* ... 某种环境 ... */
}
```

9.23 Pragma 语句

Pragma 语句:
Pragma 无作用域语句

9.24 Mixin 语句

Mixin 语句:
mixin (*赋值表达式*) ;

赋值表达式 必须在编译时求值成一个常量字符串。该字符串的文本内容必须要可编译成一个有效的*语句列表*，而且同样地被编译。

```
import std.stdio;

void main()
{
    int j;
    mixin("
        int x = 3;
        for (int i = 0; i < 3; i++)
            writefln(x + i, ++j);
        ");     // 正确

    const char[] s = "int y;";
mixin(s);  // 正确
    y = 4;      // 正确，mixin 声明的 y

    char[] t = "y = 3;";
mixin(t);  // 错误，t 在编译时是不可计算的

mixin("y =") 4; // 错误，字符串必须是完整语句

mixin("y =" ~ "4;");  // 正确
}
```

9.25 Foreach 范围语句

foreach 范围语句(foreach range statement)会在特定范围内循环执行。

```
Foreach 范围语句:
Foreach (Foreach 类型; Lwr 表达式 ..Upr 表达式 ) Scope 语句

Lwr 表达式:
表达式

Upr 表达式:
表达式
```

Foreach 类型 要声明一个变量，它可以是显示类型，也可以是从 *Lwr 表达式* 和 *Upr 表达式* 推断出的类型。*Scope 语句* 然后会被执行 *n* 次；这里的 *n* 指的是由“*Upr 表达式* - *Lwr 表达式*”计算出的结果。如果 *Upr 表达式* 小于或等于 *Lwr 表达式*，则 *Scope 语句* 就会执行 0 次。如果 *Foreach* 为 **foreach**，则该变量会被设置为 *Lwr 表达式*，并在每一次迭代后递增。如果 *Foreach* 为 **foreach_reverse**，则该变量会被设置为 *Upr 表达式*，并在每一次迭代之前递减。*Lwr 表达式* 和 *Upr 表达式* 都是每次仅计算一次，而不管 *Scope 语句* 会被执行多少次。

```
import std.stdio;

int foo()
{
    writefln("foo");
    return 10;
}

void main()
{
    foreach (i; 0 .. foo())
    {
        writef(i);
    }
}
```

输出

```
foo0123456789
```

第 10 章 数组

有四种数组(arrays)：

数组种类

语法	描述
*type**	指向数据的指针
type[*integer*]	静态数组
type[]	动态数组
type[*type*]	关联数组

10.1 指针(Pointers)

```
int* p;
```

它们是简单的指向数据的指针，等价于 C 语言的指针。这些指针的用于提供与 C 的接口，以及用于一些特定的系统工作。由于它没有相关联的长度特性，所以对于在编译或运行时进行越界检查这类工作就没有办法。大多数传统的指针用法可以通过使用动态数组、`out` 和 `ref` 参数以及引用类型来代替。

10.2 静态数组(Static Arrays)

```
int[3] s;
```

这个类似于 C 语言的数组。静态数组与众不同的地方就是其长度是在编译时固定的。

静态数组总的大小不能超过 16Mb。对于这样大的数组可以使用动态数组来代替。

维数为 0 的静态数组是允许的，只是不会为它分配空间。它可以用作做动态长度的结构的最后一个成员，或者用作模板扩展的退化情况(degenerate case)。

10.3 动态数组(Dynamic Arrays)

```
int[] a;
```

动态数组由数组长度和指向数据的指针组成。多个动态数组可能共享数组中的全部或者部分数据。

10.4 数组声明(Array Declarations)

有两种声明数组的方式：前缀式和后缀式。前缀形式是首选的方法，尤其对于一些非普通(non-trivial)类型。

10.4.1.1 前缀数组声明

前缀声明出现在被声明的标志符之前，并从右至左读，于是有：

```
int[] a;          // int 型动态数组
int[4][3] b;      // 含有 3 个元素的数组，每个元素是含 4 个 int 的数组
int[][5] c;       // 含有 5 个元素的数组，每个元素是 int 的动态数组
int*[]*[3] d;     // 含有 3 个元素的数组，每个元素是指向含指向 int 的动态数组的指针
int[]* e;         // 所指元素为动态数组的指针
```

10.4.1.2 后缀数组声明

后缀声明出现在被声明的标志符之后，并且从左至右读。下面的每一组内的声明都是等价的：

```
// int 型动态数组
int[] a;
int a[];

// 含有 3 个元素的数组，每个元素是含 4 个 int 的数组
int[4][3] b;
int[4] b[3];
int b[3][4];

// 含有 5 个元素的数组，每个元素是 int 的动态数组
int[][5] c;
int[] c[5];
int c[5][];

// 含有 3 个元素的数组，每个元素是指向含指向 int 的动态数组的指针
int*[]*[3] d;
int*[]* d[3];
int* (*d[3])[];

// 所指元素为动态数组的指针
int[]* e;
int (*e)[];
```

基本解释： 后缀形式同 C 和 C++ 采用的形式完全一样，提供这种支持会给用惯了这种形式的程序员提供一条容易的移植的途径。

10.5 用法(Usage)

对数组的操作可以分为两大类——对数组引用的操作和对数组内容的操作。C 只让操作符影响该手柄。在 D 中，这两种操作都有。

数组名其实就是数组的手柄，如 p、s 或者 a：

```
int* p;
int[3] s;
int[] a;
```

```
int* q;
int[3] t;
int[] b;

p = q;          // p 和 q 指向同一个内容。
p = s;          // p 指向数组 s 的第一个元素
p = a;          // p 指向数组 a 的第一个元素

s = ...;        // 错误，因为 s 是一个编译时
              // 静态数组引用。

a = p;          // 错误，因为由 p 所指向的数组，其长度
              // 未知
a = s;          // 被初始化为指向数组 s
a = b;          // a 和 b 指向同一个数组
```

10.6 分割(Slicing)

分割(Slicing) 一个数组表示的是指定它的一个子数组。一个数组分割并不会复制数据，它仅仅是该数组的又一个引用。例如：

```
int[10] a;      // 声明一个含有 10 个 int 数组
int[] b;

b = a[1..3];    // a[1..3] 是含有 2 个元素的数组，它的内容是
              // a[1] 和 a[2]
foo(b[1]);      // 等同于 foo(0)
a[2] = 3;
foo(b[1]);      // 等同于 foo(3)
```

简写方式 [] 代表整个数组。例如，下面那些赋给 b 的值：

```
int[10] a;
int[] b;

b = a;
b = a[];
b = a[0 .. a.length];
```

在语义上都是等价的。

分割不只是在引用数组中的某个部分时很方便，还可以把指针转换成带有边界检查的数组：

```
int* p;
int[] b = p[0..8];
```

10.7 数组复制(Array Copying)

当分割运算符作为赋值表达式左值出现时，意味着赋值的目标是数组的内容而不是对数组的引用。当左值是一个分割，而右值是一个数组或者对应类型的指针时，就会发生复制。

```
int[3] s;
int[3] t;

```

```
s[] = t;                // t[3] 的 3 个元素被复制到 s[3] 中
s[] = t[];              // t[3] 的 3 个元素被复制到 s[3] 中
s[1..2] = t[0..1];      // 等价于 s[1] = t[0]
s[0..2] = t[1..3];      // 等价于 s[0] = t[1], s[1] = t[2]
s[0..4] = t[0..4];      // 错误，s 中只有 3 个元素
s[0..2] = t;            // 错误，左值和右值的长度不同
```

重叠复制(Overlapping copies)会被视为错误：

```
s[0..2] = s[1..3];      // 错误，重叠复制
s[1..3] = s[0..2];      // 错误，重叠复制
```

相对于 C 的串行语义(serial semantics)来说，禁止重叠复制可以让编译器进行更为大胆的并行代码优化。

10.8 数组赋值(Array Setting)

如果一个分割运算符作为赋值表达式的左值出现，并且右值的类型同数组元素的类型相同，那么作为左值的数组的内容将被设置为右值的值。

```
int[3] s;
int* p;

s[] = 3;                // 等价于 s[0] = 3, s[1] = 3, s[2] = 3
p[0..2] = 3;            // 等价于 p[0] = 3, p[1] = 3
```

10.9 数组连接(Array Concatenation)

二元运算符“~”是 *连接* 运算符。它用来连接数组：

```
int[] a;
int[] b;
int[] c;

a = b ~ c;      // 创建一个新的数组，方法就是将
             // b 和 c 两个数组连接起来
```

许多语言重载了“+”运算符完成连接操作。这会产生令人费解的结果：

```
"10" + 3
```

它的值等于 13 还是字符串 "103"？这并不是一目了然的，语言设计者不得不抖擞精神，小心地编写规则来避免模棱两可的情况——而这些规则难以正确实现、容易被忽视、忘记。让“+”仍然代表相加，而使用另一个运算符来表示数组的连接，则要好得多。

类似地，“~=”运算符则表示追加, 例如：

```
a ~= b;         // a 变为 a 和 b 的连接的结果
```

连接操作总是会为操作数创建一个复本，即使其中有一个操作数是长度为 0 的数组也是一样，因此：

```
a = b;                  // a 引用了 b
a = b ~ c[0..0];        // a 引用到 b 的一个复本
```

10.10 数组操作(Array Operations)

许多数组操作，也即向量操作，可以在更高层次上进行表达，而仅仅是使用使用循环。例如，有下面的循环：

```
T[] a, b;
...
for (size_t i = 0; i < a.length; i++)
    a[i] = b[i] + 4;
```

是想将 a 的每个元素都赋值成 b 所对应的元素再加上 4。这个也可以表达成向量的形式：

```
T[] a, b;
...
a[] = b[] + 4;
```

向量操作可以通过分片操作符来表现，可以是操作符 =、+=、-=、*=、/=、%=、^=、&= 或 |= 的左值。右值可以是一个表达式，而其组成既可以是一个做为左值的具有相同长度和类型的数组分割，也可以是一个左值元素类型的表达式，甚至任意组合。支持向量操作的操作符有二元操作符：+、-、*、/、%、^、& 和 |， 以及一元操作符：- 和 ~。

左值分割和任意右值分割都不能出现交错。向量赋值运算符是从右到左进行计算的，而其它的二元运算符是从左至右进行计算的。所有操作数会被刚好计算一次，即使该数组分割内部只有零个元素。

数组元素计算的顺序是由具体实现来定义的，甚至可能并行完成。应用程序不必依赖于此顺序。

实现注意：大部分平常的向量运算都希望能利用在目标计算机里提供的任何向量数学指令。

10.11 指针运算(Pointer Arithmetic)

```
int[3] abc;                     // 含有 3 个 int 型元素的静态数组
int[] def = [ 1, 2, 3 ];        // 含有 3 个 int 型元素的动态数组

void dibb(int* array)
{
       array[2];               // 跟 *(array + 2) 相同
       *(array + 2);           // 获得第 3 个元素
}

void diss(int[] array)
{
       array[2];               // 正确
       *(array + 2);           // 错误，数组不是一个指针
}

void ditt(int[3] array)
```

```
{
       array[2];                // 正确
       *(array + 2);            // 错误，数组不是一个指针
}
```

10.12 矩形数组(Rectangular Arrays)

有经验的 FORTRAN 数值计算程序员都知道，对于像矩阵运算(matrix operations)的这样的事情，使用多维“矩形”数组相比通过指向指针的指针（这个源自语义“指向数组的指针的数组”）来访问它们要快很多。例如，D 的语法：

```
double[][] matrix;
```

就将变量 matrix 声明为一个指向数组的指针的数组。（动态数组被实现为指向数组数据的指针。）因为数组可以有不同的大小（可以动态设置大小），所以有时它被称作“锯齿(jagged)”数组。对于代码优化来说更为糟糕的是，数组的行有时会互相指向！幸运的是，D 的静态数组，使用相同的语法，被实现为固定的矩形分布：

```
double[3][3] matrix;
```

它声明了一个 3 行 3 列的矩形矩阵，在内存中连续分布。在其他语言中，它被称为多维数组并且声明类似于：

```
double matrix[3,3];
```

10.13 数组长度(Array Length)

在静态或动态数组的 [ ] 中，变量 **length** 被隐式的声明并设定为数组的长度。也可以使用符号“**$**”。

```
int[4] foo;
int[]  bar = foo;
int*   p = &foo[0];

// 下面这些表达式是等价的：
bar[]
bar[0 .. 4]
bar[0 .. length]
bar[0 .. $]
bar[0 .. bar.length]

p[0 .. length]          // 'length' 未定义，因为 p 不是数组
bar[0]+length           // 'length' 未定义，因为它不在 [ ] 内

bar[length-1]   // 得到数组内的最后一个元素
```

10.14 数组特性(Array Properties)

静态数组的特性(properties)有：

静态数组特性

特性	描述
.sizeof	返回以字节为单位的数组的大小。
.length	返回数组中元素的个数。对于静态数组来说这是一个定值。它的类型为 **size_t**。
.ptr	返回指向数组第一个元素的指针。
.dup	创建一个同样大小的动态数组并将原数组的内容复制到新数组中。
.idup	创建一个同样大小的动态数组并将原数组的内容复制到新数组中。复制后类型会变成 invariant。*仅限于 D 2.0*
.reverse	将数组中的元素按原来的逆序排列。返回数组。
.sort	将数组中的元素按照适当的方法排序。返回数组。

动态数组的特性有：

动态数组特性

特性	描述
.sizeof	返回动态数组引用的大小，在 32 位平台上是 8。
.length	获得/设置数组中元素的个数。它的类型为 **size_t**。
.ptr	返回指向数组第一个元素的指针。
.dup	创建一个同样大小的动态数组并将原数组的内容复制到新数组中。
.idup	创建一个同样大小的动态数组并将原数组的内容复制到新数组中。复制后类型会变成 invariant。*仅限于 D 2.0*
.reverse	将数组中的元素按原来的逆序排列。返回数组。
.sort	将数组中的元素按照适当的方法排序。返回数组。

对于工作在类对象数组上的 **.sort** 特性，该类的定义里必须定义函数：`int opCmp(Object)`。它用于决定类对象的顺序。注意：该参数是 `Object` 类型，而不是该类的类型。

对于工作在结构或联合数组上的 **.sort** 特性，该类的定义里必须定义函数：`int opCmp(S)` 或者 `int opCmp(S*)`。类型 `S` 是结构或者联合类型。此函数会决定排列次序。

示例:

```
p.length        // 错误，对于指针来说 length 是未知的
s.length        // 编译时间常量 3
a.length        // 运行时值

p.dup           // 错误，长度未知，无法复制
s.dup           // 创建一个含有 3 个元素的数组，
```

```
                // 将元素复制到它内
a.dup             // 创建一个含有 a.length 个元素的数组，
                // 同时把 a 的元素复制到它里面
```

10.14.1 设置动态数组的长度

动态数组的 **.length** 特性可以设置为运算符“=”的左值：

```
array.length = 7;
```

这会造成数组被在适当的位置被重新分配，现有的内容被原封不动的复制到新的数组中。如果新的数组比原数组短，将只保留新数组能够容纳的那些内容。如果新数组比原数组长，新数组长出的部分将会使用元素的默认初始值填充。

为了达到最高的效率，运行时总是试图适当地调整数组的大小以避免额外的复制工作。如果数组不是由 new 运算符或者上一个 resize 操作分配的，并且新数组比原数组大，那么运行时将总是进行复制。

这意味着如果有一个数组进行了分割，紧跟着该数组又被重置大小，那么改变大小后的数组可能会同那个分割有重叠，即：

```
char[] a = new char[20];
char[] b = a[0..10];
char[] c = a[10..20];

b.length = 15;  // 总是会被根据实际情况的重置大小，因为它是位于 a[] 上的一个分割，
                // 而 a[] 的长度大于 15
b[11] = 'x';    // a[15] 和 c[5] 也受到影响

a.length = 1;
a.length = 20;  // 内存分布没有改变

c.length = 12;  // 总是会进行复制，因为 c[] 不在
                //  gc 分配的块的开始处
c[5] = 'y';     // 不影响 a[] 和 b[] 的内容

a.length = 25;  // 可能会也可能不会进行复制
a[3] = 'z';     // 可能会也可能不会影响到 b[3] ，它仍然同原来的
                //  a[3] 有重叠
```

为保证复制的行为，请使用 .dup 特性来确保可以被重新调整大小的数组的唯一性。

这些讨论也同样适用于使用 ~ 和 ~= 运算符进行的连接操作。

重置动态数组的大小是一个相对昂贵的操作。所以，尽管下面这个用于填充数组的方法结果正确：

```
int[] array;
while (1)
{   c = getinput();
```

```
    if (!c)
      break;
    array.length = array.length + 1;
    array[array.length - 1] = c;
}
```

它的效率也不会高。有一种更为实际的方法可以尽量减少重新调整大小操作的数量：

```
int[] array;
array.length = 100;         // 估计的大小
for (i = 0; 1; i++)
{    c = getinput();
     if (!c)
       break;
     if (i == array.length)
       array.length = array.length * 2;
     array[i] = c;
}
array.length = i;
```

选择一个好的猜测值是一门艺术，但是通常你能找到一个适用于 99% 的情况的值。例如，当从控制台采集用户输入时——长度不太可能超过 80 。

10.14.2 函数做为数组特性

如果函数的第一个参数是数组，那么此函数可以被当作该数组的一个特性来进行调用：

```
int[] array;
void foo(int[] a, int x);

foo(array, 3);
array.foo(3);    // 表示同样的意思
```

10.15 数组边界检查(Array Bounds Checking)

如果使用小于 0 或大于等于数组长度的数作为数组下标是一个错误。若下标越界，如果是由运行时检测到，会抛出一个 ArrayBoundsError 异常；如果是在编译时检测到，会产生一个错误。程序不应该依赖于数组越界异常检查，例如，下面这个程序的方法是错误的：

```
try
{
    for (i = 0; ; i++)
    {
        array[i] = 5;
    }
}
catch (ArrayBoundsError)
{
// 终止循环
}
```

此循环的正确写法是：

```
for (i = 0; i < array.length; i++)
```

```
{
    array[i] = 5;
}
```

实现注意： 编译器应该在编译时尝试检查数组越界错误，例如：

```
int[3] foo;
int x = foo[3];          // 错误，越界了
```

应该可以在编译时通过一个选项决定是否插入动态数组越界检查代码。

10.16 数组初始化(Array Initialization)

10.16.1 默认初始化(Default Initialization)

- 指针被初始化为 **null**。
- 静态数组的内容被初始化为数组元素类型的默认初始值。
- 动态数组被初始化为拥有 0 个元素。
- 关联数组被初始化为拥有 0 个元素。

空的初始化(Void Initialization)

当数组的 *初始值* 为 **void** 时就会发生空的(void)初始化。没有进行初始化，即表示此时数组的内容将为未定义。这个做为效率优化是最有用的。空的初始化是一种高级技术，应该只在经剖析表明它有必要时才使用它。

10.16.2 静态数组的静态初始化(Static Initialization)

静态初始化由包含在 [ ] 里面的数组元素值列表实现。这些值可以可选地在它们的前面放置一个索引值和“:”。如果没有提供索引值，则该索引值会被设置成前一个索引值加 1，或者如果是第一个值，则索引值会为 0。

```
int[3] a = [ 1:2, 3 ];          // a[0] = 0, a[1] = 2, a[2] = 3
```

当数组下标识为枚举类型时，这种写法是最方便的：

```
enum Color { red, blue, green };

int value[Color.max + 1] = [ Color.blue:6, Color.green:2, Color.red:5 ];
```

当这些数组出现在全局域里时，它们都是静态的。否则，就需要使用 **const** 或 **static** 存储类别来对它们进行标记，让它们成为静态数组。

10.17 特殊数组类型

10.17.1 字符串

一个字符串指的就是由多个字符组成的一个数组。而字符串字法指的仅仅是一种写成字符数组的简单的方法。字符串文字是不可变的（只读）。

```
char[] str1 = "abc";                    // 错误，"abc" 不是可变量(mutable)
char[] str2 = "abc".dup;                // 正确，做一个可变的复本
invariant(char)[] str3 = "abc";         // 正确
invariant(char)[] str4 = str1;          // 错误，str4 不是可变量
invariant(char)[] str4 = str1;          // 正确，做一个不可变的(invariant)复本
```

名字 string 是 invariant(char)[] 的别名，因此上面的声明可以等价地写成：

```
char[] str1 = "abc";                    // 错误，"abc" 不是可变量(mutable)
char[] str2 = "abc".dup;                // 正确，做一个可变的复本
string str3 = "abc";          // 正确
string str4 = str1;           // 错误，str4 不是可变量
string str5 = str1.idup;      // 正确，做一个不可变的(invariant)复本
```

char[]字符串采用 UTF-8 格式。wchar[]字符串采用 UTF-16 格式。dchar[]字符串采用 UTF-32 格式。

字符串可以复制、比较、连接和追加：

```
str1 = str2;
if (str1 < str3) ...
func(str3 ~ str4);
str4 ~= str1;
```

使用显而易见的语义。所有生成的临时对象都会被垃圾回收器回收（或者使用 alloca()）。而且，这对于所有的数组都适用，而不仅仅是对字符串数组。

可以使用指向 char 的指针：

```
char* p = &str[3];        // 指向第 4 个元素
char* p = str;            // 指向第 1 个元素
```

但是，因为 D 中的字符串数组不是以 0 终止的，所以如果要把一个字符串指针传递给 C 时，应该加上终止符 0：

```
str ~= "\0";
```

或者使用函数 std.string.toStringz。

字符串的类型在编译的语义分析阶段决定。类型是下列之一：char[]、wchar[]、dchar[]，并且按照隐式转换规则决定。如果有两个可用的等价的转换，会被认为是一个错误。为了避免出现这种模棱两可的局面，应该使用类型转换，或者使用后缀 **c**、**w** 或 **d**：

```
cast(wchar [])"abc"       // 这是个 wchar 型的字符数组
"abc"w                    // 这个也是
```

如果需要的话，字符串文字会隐式地在 char、wchar 和 dchar 之间转换。

```
char c;
wchar w;
dchar d;

c = 'b';                // c 被赋值为 char 字符 'b'
w = 'b';                // w 被赋值为 wchar 字符 'b'
w = 'bc';               // 错误——只能使用一个 wchar 字符
w = "b"[0];             // w 被赋值为 wchar 字符 'b'
w = \r[0];              // w 被赋值为 wchar 字符“回车”
d = 'd';                // d 被赋值为 dchar 字符 'd'
```

10.17.1.1 C 的 printf() 和 字符串

printf() 是一个 C 函数，它不是 D 的一部分。**printf()** 输出以 0 终止的 C 字符串。有两种用 **printf()** 输出 D 字符串的方法。第一种方法是加一个终止符 0，并将结果转换为 char* :

```
str ~= "\0";
printf("the string is '%s'\n", cast(char*)str);
```

或者：

```
import std.string;
printf("the string is '%s'\n", std.string.toStringz(str));
```

字符串文字后面已经添加有一个 0，因此可以直接使用它们：

```
printf("the string is '%s'\n", cast(char*)"string literal");
```

因此，为什么传递给 printf 的第一个字符串文字不需要进行类型转换呢？第一个参数被原型化(prototyped)成了一个 char*，而一个字符串文字可以被隐式转换到类型 char*。但是传递给 printf 的余下实参是个数不定型(variadic)的（由“...”指定），而且一个字符串文字是被当作一个（长度,指针）的组合传递给个数不定型形参的。

第二种方法使用精度指示符。D 数组的布局是：数组长度、字符串，所以下面这种方法可以工作：

```
printf("the string is '%.*s'\n", str);
```

最好的方式是使用 std.stdio.writefln，它可以处理 D 字符串：

```
import std.stdio;
writefln("the string is '%s'", str);
```

10.17.2 隐式转换

指针 *T** 可以被隐式的转换为下面的内容：

- void*

静态数组 *T*[*dim*] 可以被隐式地转换为下列之一：

- *T*[]
- *U*[]
- void[]

动态数组 *T*[] 可以被隐式地转换为下列之一：

- *U*[]
- void[]

这里的 *U* 是 *T* 的基类。

第 11 章 关联数组

关联数组有一个索引，该索引不一定要是一个整数，也可以是一个很稀疏的值(sparsely populated)。一个关联数组的索引叫做 *关键字(key)*，而它的类型叫做 *关键字类型(KeyType)*。

在关联数组的声明中，*关键字类型* 的类型声明位于 [] 内：

```
int[char[]] b;           // 整型关联数组 b
                    // 通过一个字符数组来建立索引。
                    // 其 关键字类型 是 char[]
b["hello"] = 3;          // 设定关键字 "hello" 对应的值为 3
func(b["hello"]);        // 把 3 做为参数传递给 func()
```

关联数组里的特定关键字可以通过 remove 函数被移除：

```
b.remove("hello");
```

如果该关键字在关联数组里，则 *In 表达式* 会产生一个指向此关联字所关联的值的指针；如果不在，则为 **null**：

```
int* p;
p = ("hello" in b);
if (p != null)
       ...
```

关键字类型 不能是函数或者 void。

11.1 使用类做为关键字类型

类也可以被用作 *关键字类型*。为了这个可以工作，类定义必须重写(override) `Object` 类的下列成员函数：

- `hash_t toHash()`
- `bool opEquals(Object)`
- `int opCmp(Object)`

`hash_t` is an alias to an integral type.

注意，`opCmp` 和 `opEquals` 的参数都是 `Object` 类型，而不是定义的那个类类型。

例如：

```
class Foo
{
   int a, b;

   hash_t toHash() { return a + b; }

   bool opEquals(Object o)
   {   Foo f = cast(Foo) o;
       return f && a == foo.a && b == foo.b;
   }
```

```
    int opCmp(Object o)
    {   Foo f = cast(Foo) o;
        if (!f)
            return -1;
        if (a == foo.a)
            return b - foo.b;
        return a - foo.a;
    }
}
```

这些实现可以使用 opEquals 或者 opCmp 或者两者。小心以防 opEquals 和 opCmp 的结果在它们的类对象不同时彼此混淆，或者小心相反的情况。

11.2 使用结构或联合做为关键字类型

如果 *关键字类型* 是一个结构或联合类型，则会使用默认机制基于结构值内的二元数据来计算 hash（哈稀值）和它的比较值。也可以通过提供下面函数做为结构成员实现自定义机制：

```
const hash_t toHash();
const bool opEquals(ref const KeyType s);
const int opCmp(ref const KeyType s);
```

例如：

```
import std.string;

struct MyString
{
    string str;


    const hash_t toHash()
    {   hash_t hash;
        foreach (char c; s)
            hash = (hash * 9) + c;
        return hash;
    }

    const bool opEquals(ref const MyString s)
    {
        return std.string.cmp(this.str, s.str) == 0;
    }

    const int opCmp(ref const MyString s)
    {
        return std.string.cmp(this.str, s.str);
    }
}
```

这些实现可以使用 opEquals 或者 opCmp 或者两者。请特别注意以防在 opEquals 和 opCmp 的结构/联合对象不同时，它们的结果可能彼此混淆，或者是相反的情况。

11.3 特性

关联数组的特性有：

关联数组特性

特性	描述
.sizeof	返回指向关联数组的引用的大小；通常值是 8 。
.length	返回关联数组中值的个数。与动态数组不同的是，它是只读的。
.keys	返回动态数组，数组的元素就是关联数组的那些关键字。
.values	返回动态数组，数组的元素就是关联数组的那些对应值。
.rehash	适当地重新组织关联数组以提高查找效率。例如，程序已经载入了符号表并将开始进行查找事，rehash 会提高效率。返回指向新数组的引用。

11.4 关联数组示例：单词统计

```
import std.file;          // D 文件 I/O
import std.stdio;

int main (string[] args)
{
   int word_total;
   int line_total;
   int char_total;
   int[char[]] dictionary;

   writefln("   lines   words   bytes file");
for (int i = 1; i < args.length; ++i)      // 程序的形式参数
   {
      char[] input;             // 输入缓冲区
      int w_cnt, l_cnt, c_cnt; // word、line、char 计数器
      int inword;
      int wstart;

      // 将文件读入到 input[]
      input = cast(char[])std.file.read(args[i]);

      foreach (j, char c; input)
      {
         if (c == '\n')
               ++l_cnt;
         if (c >= '0' && c <= '9')
         {
         }
         else if (c >= 'a' && c <= 'z' ||
```

```
                c >= 'A' && c <= 'Z')
        {
            if (!inword)
            {
                wstart = j;
                inword = 1;
                ++w_cnt;
            }
        }
        else if (inword)
        {
            char[] word = input[wstart .. j];
            dictionary[word]++;         // 递增 word 的计数器
            inword = 0;
        }
        ++c_cnt;
    }
    if (inword)
    {
        char[] word = input[wstart .. input.length];
        dictionary[word]++;
    }
    writefln("%8d%8d%8d %s", l_cnt, w_cnt, c_cnt, args[i]);
    line_total += l_cnt;
    word_total += w_cnt;
    char_total += c_cnt;
}

if (args.length > 2)
{
    writef("--------------------------------------\n%8d%8d%8d total",
           line_total, word_total, char_total);
}

writefln("--------------------------------------");
foreach (word; dictionary.keys.sort)
{
    writefln("%3d %s", dictionary[word], word);
}
return 0;
}
```

第 12 章 结构&联合

尽管类是引用类型，而结构是值类型。任何 C 结构都可以被准确的表示为 D 结构。按照 C++ 说法，D 结构是一种 POD (普通旧数据) 类型，且带有无关紧要的构造函数和析构函数。结构和联合用于表示简单的数据聚集，或者用作这样一种方式——在硬件上的描绘数据结构或者描绘一种外部类型。外部类型可以被操作系统的 API 定义，或者被文件格式定义。面向对象功能提供有类数据类型。

一个结构被定义成不带身份标识；即是说，它的实现是自由的，可以很方便进行结构的位复制。

结构、类比较表

功能	struct	class	C 结构	C++ 结构	C++ 类
值类型	X		X	X	X
引用类型		X			
数据成员	X	X	X	X	X
隐藏成员	X	X		X	X
静态成员	X	X		X	X
默认成员初始值	X	X			
位域			X	X	X
非虚成员函数	X	X		X	X
虚成员函数		X		X	X
构造函数	X	X		X	X
postblit/复制构造函数	X			X	X
析构函数	X	X		X	X
RAII	X	X		X	X
赋值重载	X			X	X
字法	X				
操作符重载	X	X		X	X
继承		X		X	X
不变量	X	X			
单元测试	X	X			
同步		X			
可参数化	X	X		X	X
对齐控制	X	X			

成员保护	X	X		X	X
默认公有	X	X	X	X	
标记名字空间			X	X	X
匿名	X		X	X	X
静态构造函数	X	X			
静态析构函数	X	X			
常量/不变量	X	X			
inner nesting	YES	YES			

```
聚集声明:
        标记 标识符 结构体
        标记 标识符 ;

标记:
        struct
        union

结构体:
        { }
        { 多个结构体声明 }

多个结构体声明:
        单个结构体声明
        单个结构体声明 多个结构体声明

单个结构体声明:
        单个声明
        静态构造函数
        静态析构函数
        不变量
        单元测试
        结构分配器
        结构释放器
        结构构造函数
        结构 Postblit
        结构析构函数
        AliasThis


结构分配器:
        类分配器

结构释放器:
        类释放器
```

它们同在 C 中的结构基本一样，除了以下一些例外：

- 没有位域
- 可以显式指定对齐
- 没有单独的标记名字空间——标记名并入当前的作用域
- 像下面的声明：

```
struct ABC x;
```

是不允许的，正确方法是：

```
ABC x;
```

- 匿名结构/联合可以作为其他结构/联合的成员存在
- 可以为成员提供默认初始值
- 可以拥有成员函数和静态成员

12.1 结构的静态初始化

如果不提供初始值，静态结构成员会被默认地初始化为成员类型的初始值。如果提供了静态初始值，成员会按照这样的语法初始化：成员名，冒号，表达式。成员可以按照任意顺序初始化。静态结构的初始值要求必须在编译期可求值。没有出现在初始化列表中的成员会被默认值初始化。

```
struct X { int a; int b; int c; int d = 7;}
static X x = { a:1, b:2};              // c 被设置为 0, d 为 7
static X z = { c:4, b:5, a:2 , d:5};  // z.a = 2, z.b = 5, z.c = 4, z.d = 5
```

C 风格的初始化，依赖于结构声明中的成员的顺序，也是被支持的：

```
static X q = { 1, 2 };    // q.a = 1, q.b = 2, q.c = 0, q.d = 7
```

结构文字也可以用来初始化静态结构，不过它们在编译期必须是可求值的。

```
static X q = S( 1, 2+3 );   // q.a = 1, q.b = 5, q.c = 0, q.d = 7
```

静态初始值语法也可以被用来初始化非静态变量，同时假定未给出成员名字。该初始化值无须要求在编译期可求值。

```
void test(int i)
{
   S s = { 1, i };   // q.a = 1, q.b = i, q.c = 0, q.d = 7
}
```

12.2 联合的静态初始化

联合需要显式地初始化。

```
union U { int a; double b; }
static U u = { b : 5.0 };               // u.b = 5.0
```

如果联合的其他成员占用的存储空间更大，剩下的部分被初始化为零。

12.3 结构的动态初始化

结构可以被来自于另一个相同类型值动态初始化：

```
struct S { int a; }
S t;       // 默认初始化
t.a = 3;
S s = t;  // s.a 被设置为 3
```

如果重写(override)了该结构的 opCall，同时使用一个不同类型的值来初始化此结构，那么 opCall 操作就会被调用：

```
struct S
{   int a;

    static S opCall(int v)
    {   S s;
        s.a = v;
        return s;
    }

    static S opCall(S v)
    {   S s;
        s.a = v.a + 1;
        return s;
    }
}

S s = 3;         // 把 s.a 设置为 3
S t = s;         // 把 t.a 设置成 3，而 S.opCall(s) 没有被调用
```

12.4 结构字法

结构字法由结构的名字跟上带括号的参数列表组成：

```
struct S { int x; float y; }

int foo(S s) { return s.x; }

foo( S(1, 2) );   // 将域 x 设置为 1，域 y 设置为 2
```

结构字法在句法等同于函数调用。如果一个结构有一个名叫 opCall 的成员函数，那么此结构的结构文字就是可用的。如果参数多于了结构里域的个数，那么这样就会产生错误。如果参数个数比域个数更少，则多余的域则会使用它们各自的默认初始值进行初始化。如果结构里有匿名联合，那么就只有匿名联合的第一个成员可以使用结构文字进行初始化，而且其余的非重迭域会被默认初始化。

12.5 结构特性

结构特性

.sizeof	以字节为单位的结构大小
.alignof	大小边界结构需要进行排列
.tupleof	获得域的元组类型

12.6 结构域特性

结构域特性

.offsetof	相对结构开始处的域偏移字节数

12.7 常量与不变量结构

结构声明可能使用存储类别 const 或 invariant。它等效于将结构的每个成员声明为 const 或 invariant。

```
const struct S { int a; int b = 2; }

void main()
{
S s = S(3);     // 把 s.a 初始化成 3
    S t;            // 把 t.a 初始化成 0
    t = s;          // 正确，t.a 现在为 3
    t.a = 4;        // 错误，t.a 是常量
}
```

12.8 结构构造函数

结构构造函数:
this(***形参列表*** **)** *函数体*

结构构造函数被用来初始化结构的实例。*形参列表* 可以为空。没有使用构造函数实例化的结构实例会被默认初始化成它们的 .init 值。

12.9 结构 Postblit

结构 Postblit:
this(this) *函数体*

复制构造(Copy construction) 被定义为从相同类型的另一个记录来初始化此记录实例。复制构造可以分成两个部分：

1. 位块传送各个域，即，按位复制
2. 在结构里运行 *postblit*

第一个部分由语言自动完成；如果该结构定义了一个 postblit 函数，第二个部分也算是完成的。postblit 只访问到目标结构对象，而不是源对象。它的工作就是根据需要“修复”目标，如复制引用数据，递增引用计数等等。实例：

```
struct S
{
int[] a;     // 数组为 this 实例私有
   this(this)
   {
      a = a.dup;
   }
   ~this()
   {
      delete a;
   }
}
```

12.10 结构析构函数

结构析构函数:
~this() *函数体*

析构函数在一个对象失去作用时会被调用。它们的目标就是要释放结构对象所热的人资源。

12.11 赋值重载

在“复制构造”使用另一个相同类型的对象来初始化一个对象时，赋值(assignment)就会被定义成复制一个对象的内容到另一个已初始化的对象上去，例如：

```
struct S { ... }
S s;       // s 的默认构造过程
S t = s;   // t 由 s 复制构造而来
t = s;     // t 被赋值成 s
```

结构赋值 t=s 的语义定义等同于：

```
t = S.opAssign(s);
```

此处，opAssign 是 S 的一个成员函数：

```
S* opAssign(S s)
{    ... 将 *this 位复制成 tmp ...
... 将 s 位复制成 *this ...
... 调用 tmp 的析构函数 ...
    return this;
}
```

尽管编译器会根据需要生成默认的 opAssign，也还是可以附加一个自定义的 opAssign。自定义的 opAssign 仍然必须实现相等语义，不过可能会更高效。

自定义 opAssign 可能更高效的一个原因就是，该结构有可能会引用到局部缓冲区：

```
struct S
{
    int[] buf;
    int a;

    S* opAssign(ref const S s)
    {
        a = s.a;
        return this;
    }

    this(this)
    {
        buf = buf.dup;
    }

    ~this()
    {
        delete buf;
    }
}
```

这里，S 有一个临时的工作空间 buf[]。一般的 postblit 会毫无意义的将其释放然后重新分配。自定义的 opAssign 会反复使用已有的存储类别。

12.12 嵌套结构

*嵌套结构(nested struct)*是这样一种结构——声明所在的作用域位于一个函数或拥有将本地函数做为模板参数别名的模板结构里面。 嵌套结构有成员函数。 它可以（通过一个添加的隐藏字段）访问其封闭作用域的上下文(context)。

```
void foo()
{   int i = 7;
    struct SS
    {
        int x,y;
        int bar() { return x + i + 1; }
    }
    SS s;
    s.x = 3;
    s.bar(); // 返回 11
}
```

嵌套结构不能用作字段(field)或数组的元素类型：

```
void foo()
{   int i = 7;
    struct SS
    {
```

```
        int x,y;
        int bar() { return x + i + 1; }
    }
    struct DD
    {
        SS s;     // 出错，不能是字段
    }
    SS[3] a;      // 出错，不能是数组元素
    SS[] a;       // 出错，不能是数组元素
}
```

结构可以通过 static 属性来禁止被嵌套，只不过这样就无法访问其所在封闭作用域内的变量了。

```
void foo()
{   int i = 7;
    static struct SS
    {
        int x,y;
        int bar()
        {
           return i;    // 错误，SS 不是一个嵌套结构
        }
    }
}
```

如果模板化的结构拥有一个本地函数，其参数是别名化传递的，则它就会成为一个嵌套结构：

```
struct A(alias F)
{
    int fun(int i) { return F(i); }
}

A!(F) makeA(alias F)() {return A!(F)(); }

void main()
{
    int x = 40;
    int fun(int i) { return x + i; }
    A!(fun) a = makeA!(fun)();
    a.fun(2);
}
```

第 13 章 类

D 的面向对象的特性都来源于类。类层次里的顶层是 Object 类。Object 定义了每个派生类拥有的最小功能集合，并为这些功能提供了默认的实现。

类是程序员定义的类型。作为面向对象语言，D 支持对象，支持封装、继承和多态。D 类支持单继承编程范式，并且支持接口。类对象只能通过引用具现化。

类可以被导出，这意味着它的名字和非私有成员将会暴露给外部的 DLL 或者 EXE 。

类声明的定义：

类声明:
        class *标识符* *基类列表*_{可选的} *类过程体*

基类列表:
        : *父类*
        : *父类* *多个接口类*
        : *单个接口类*

父类:
        标识符
        保护属性 *标识符*

多个接口类:
        单个接口类
        单个接口类 *多个接口类*

单个接口类:
        标识符
        保护属性 *标识符*

保护属性:
        private
        package
        public
        export

类过程体:
        { }
        { *多个类过程体声明* **}**

多个类过程体声明:
        单个类过程体声明
        单个类过程体声明 *多个类过程体声明*

单个类过程体声明:
        单个声明
        构造函数
        析构函数
        静态构造函数
        静态析构函数
        不变量
        单元测试

```
        类分配器
        类释放器
```

类的组成：

- 父类
- 接口
- 动态域
- 静态域
- 类型
- 成员函数
    - 静态成员函数
    - 虚函数(Virtual Functions)
    - 同步函数(Synchronized Functions)
    - 构造函数
    - 析构函数
    - 静态构造函数
    - 静态析构函数
    - 类不变量(Class Invariants)
    - 单元测试
    - 类分配器(Class Allocators)
    - 类释放器(Class Deallocators)
    - Alias This

一个类被定义成：

```
class Foo
{
... 成员 ...
}
```

注意在标志类定义结束的‘ }’之后没有‘;’。同样不能像下面这样声明变量：

```
class Foo { } var;
```

正确方法是：

```
class Foo { }
Foo var;
```

13.1 域(Field)

类成员总是通过‘.’运算符访问。并没有像 C++ 里的 ::或者 -> 操作符

D 编译器有权利重新排列类中各个域的顺序，这样就允许编译器按照实现定义的方式将它们压缩以优化程序。考虑函数中的局部变量的域——编译器将其中一些分配到寄存器中，其他的按照最理想的分布被保存到堆栈帧中。这就给了代码设计者重排代码以加强可读性的自

由，而不必强迫代码设计者依据机器的优化规则排列相关的域。结构/联合提供了显式控制域分布的能力，但这不是类的分内之事。

13.2 域特性(Field Properties)

.offsetof 特性给出的是起始自类实例的域的字节偏移量。**.offsetof** 仅能用于那些产生域自己类型的表达式，而不能用于 class 类型：

```
class Foo
{
    int x;
}
...
void test(Foo foo)
{
    size_t o;

o = Foo.x.offsetof;    // 错误，Foo.x 需要一个 'this' 引用
    o = foo.x.offsetof;    // 正确
}
```

13.3 类特性

特性 **.tupleof** 会返回一个类里所有域的 *表达式元组*，不过隐藏域和基类里的域除外。

```
class Foo { int x; long y; }
void test(Foo foo)
{
    foo.tupleof[0] = 1;          // 把 foo.x 设置为 1
    foo.tupleof[1] = 2;          // 把 foo.y 设置为 2
    foreach (x; foo.tupleof)
       writef(x);               // 输出 12
}
```

.__vptr 和 **.__monitor** 特性会分别访问类对象的 vtbl[] 和 monitor，不过不应在用户代码里使用。

13.4 父类(Super Class)

所有的类都从父类继承。如果没有指定，它就从 Object 继承。Object 是 D 类层次体系的最顶层。

13.5 成员函数(Member Functions)

非静态成员函数有一个额外的隐藏参数，名叫 *this*，通过它可以访问该类对象的其它成员。

13.6 同步函数(Synchronized Functions)

同步的(synchronized)类成员函数拥有存储类别 `synchronized`。如果一个静态成员函数在类的 *classinfo* 对象上是同步的，则表示会有一个 monitor 被用于该类的所有静态同步成员

函数。对于非静态同步函数，使用的 monitor 则属于该类对象的一部分。例如：

```
class Foo
{
synchronized void bar() { ...语句... }
}
```

等同于（而 monitors 也一样）：

```
class Foo
{
   void bar()
   {
synchronized (this) { ...语句... }
   }
}
```

结构没有同步成员函数。

13.7 构造函数

```
构造函数:
       this 参数 函数体
```

所有的成员都被初始化为它对应类型的默认初始值，通常整数被初始化为 0，浮点数被初始化为 NAN。这就防止了因为在某个构造函数中忽略了初始化某个成员而造成那已发现的错误。在类定义中，可以使用静态的初始值代替默认值：

```
class Abc
{
    int a;       // a 的默认初始值为 0
    long b = 7; // b 的默认初始值为 7
    float f;     // f 的默认初始值为 NAN
}
```

静态初始化在调用构造函数之前完成。

构造函数是名为 **this** 的函数，它没有返回值：

```
class Foo
{
    this(int x)          // 声明 Foo 的构造函数
    {   ...
    }
    this()
    {   ...
    }
}
```

基类的构造通过用 **super** 调用基类的构造函数来完成：

```
class A { this(int y) { } }

class B : A
{
    int j;
    this()
    {
        ...
        super(3);        // 调用基类构造函数 A.this(3)
        ...
    }
}
```

构造函数也能调用同一个类中的其他构造函数以共享通用的初始化代码（这个叫做委托构造函数）：

```
class C
{
    int j;
    this()
    {
        ...
    }
    this(int i)
    {
        this();
        j = i;
    }
}
```

如果构造函数中没有通过 **this** 或 **super** 调用构造函数，并且基类有构造函数，编译器将在构造函数的开始处自动插入一个 **super()**。

如果类没有构造函数，但是基类有构造函数，那么默认的构造函数的形式为：

```
this() { }
```

这会由编译器隐式生成。

类对象的构造十分灵活，但是有一些限制：

1. 构造函数互相调用是非法的：

```
this() { this(1); }
this(int i) { this(); } // 非法，构造函数循环调用
```

2. 如果一个构造函数中调用了构造函数，那么这个构造函数的任何执行路径中都只能调用一次构造函数：

```
this()  { a || super(); }        // 非法

this() { (a) ? this(1) : super(); }      // 正确

this()
{
    for (...)
```

```
    {
        super();          // 非法，位于循环内
    }
}
```

3. 在构造函数出现之前显式或者隐式引用 **this** 都是非法的。
4. 不能在标号后调用构造函数（这样做的目的是使检查 goto 的前导条件容易完成）。

类对象的实例使用 *New 表达式* 创建：

```
A a = new A(3);
```

在这个过程中按照下面的步骤执行：

1. 为对象分配存储空间。如果失败，不会返回 **null**，会抛出一个 **OutOfMemoryException** 异常。因此，不再需要编写冗长而乏味的 null 引用防卫代码。
2. 使用类定义中的值静态初始化“原始数据”。指向 vtbl[]（指向虚函数的指针数组）的指针被赋值。这保证了调用构造函数的类是已经完全成型的。这个操作等价于使用 memcpy() 将对象的静态版本拷贝到新分配的对象的空间，但是更高级的编译器将会对这种方法进行优化。
3. 如果为类定义了构造函数，匹配调用参数列表的构造函数被调用。
4. 如果打开了类不变量的检查，在构造函数调用后调用类不变量。

13.8 析构函数

析构函数:
~this() *函数体*

当对象被垃圾回收器删除时将调用析构函数。语法如下：

```
class Foo
{
    ~this()          // Foo 的析构函数
    {
    }
}
```

每个类只能有一个析构函数，析构函数没有参数，没有特征。它总是虚函数。

析此构函数的作用是要能释放对象所持有的所有资源。

程序可以显式的通知垃圾回收程序不在引用一个对象（使用 delete 表达式），然后垃圾回收器会立即调用析构函数，并将对象占用的内存放回自由存储区。析构函数决不会被调用两次。

当析构函数运行结束时会自动调用父类的析构函数。不能显式调用父类的析构函数。

垃圾回收器不能保证对所有的无引用对象调用析构函数。而且垃圾回收器也不能保证调用的

相对顺序。即表示当垃圾回收器调用一个对象的析构函数时，而该对象的类包含的成员有对垃圾回收对象的引用，那么这些引用都可能不再有效。这意味着析构函数不能引用子对象。这条件规则并不适用于自动(auto)对象或那些使用 *Delete 表达式* 删除的对象，原因就是，它们的析构函数不是由垃圾回收器来运行的，即是说所有引用都是有效的。

从数据段中引用的对象不会被 gc 回收。

13.9 静态构造函数

静态构造函数:
    static this() *函数体*

静态构造函数在 `main()` 函数获得控制前执行初始化。静态构造函数用来初始化其值不能再编译时求出的静态类成员。

其他语言中的静态构造函数被设计为可以使用成员进行初始化。这样做的问题是无法精确控制代码执行的顺序，例如：

```
class Foo
{
   static int a = b + 1;
   static int b = a * 2;
}
```

最终 a 和 b 都是什么值？初始化按照什么顺序执行？在初始化前 a 和 b 都是什么值？这是一个编译是错误吗？抑或它是一个运行是错误？还有一种不那么明显的令人迷惑的情况是单独一个初始化是静态的还是动态的。

D 让这一切变得简单。所有的成员初始化都必须在编译时可确定，因此就没有求值顺序依赖问题，也不可能读取一个未被初始化的值。动态初始化由静态构造函数执行，采用语法 `static this()` 实现。

```
class Foo
{
   static int a;                // 默认初始化为 0
   static int b = 1;
   static int c = b + a;        // 错误，不是常量初始值

   static this()                // 静态构造函数
   {
      a = b + 1;               // a 被设置为 2
      b = a * 2;               // b 被设置为 4
   }
}
```

`static this()` 会由启动代码在调用 `main()` 之前调用。如果它正常返回（没有抛出异常），静态析构函数就会被加到会在程序终止时被调用的函数列表中。静态构造函数的参数列表为空。

模块里的静态构造函数会以它们出现在词法里的顺序执行。直接或间接导入的模块的所有静态构造函数则在为些导入者的静态构造函数之前执行。

静态构造函数里的 **static** 并不是属性，因此它必须紧挨在 **this** 前面:

```
class Foo
{
    static this() { ...}          // 一个静态构造函数
    static private this() { ...} // 不是静态构造函数
    static
    {
        this() { ...}             // 不是静态构造函数
    }
    static:
        this() { ...}             // 不是静态构造函数
}
```

13.10 静态析构函数

静态析构函数:
static ~this() *函数体*

静态析构函数定义为具有语法形式 `static ~this()` 的特殊静态函数。

```
class Foo
{
    static ~this()                // 静态析构函数
    {
    }
}
```

静态析构函数在程序终止的时候被调用，但这只发生在静态构造函数成功执行完成时。静态析构函数的参数列表为空。静态析构函数按照静态构造函数调用的逆顺序调用。

静态析构函数里的 **static** 并不是属性，因此它必须紧挨在 **~this** 前面:

```
class Foo
{
    static ~this() { ...}         // 静态析构函数
    static private ~this() { ...} // 不是静态析构函数
    static
    {
        ~this() { ...}            // 不是静态析构函数
    }
    static:
        ~this() { ...}            // 不是静态析构函数
}
```

13.11 类不变量(Class Invariants)

不变量:
invariant() *块语句*

类不变量(class invariants)用来指定类必须总是为真的这一特质（除了在执行成员函数时）。例如，代表日期的类可能有一个不变量：day 必须位于 1..31 之间，hour 必须位于 0..23 之间：

```
class Date
{
    int day;
    int hour;

    invariant()
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24);
    }
}
```

类不变量就是一种契约，是必须为真的断言。当类的构造函数执行完成时、在类的析构函数开始执行时、在 public 或 exported 成员函数执行之前，或者在 public 或 exported 成员函数完成时，将检查不变量。

不变量中的代码不应该调用任何非静态公有类成员函数，无论直接还是间接。如果那样做的话就会造成堆栈溢出，因为不变量将被无限递归调用。

由于不变量是在公共(public)成员或导出(exported)成员的开始部分被调用的，因此，这些成员不应该从构造函数里进行调用。

```
class Foo
{
    public void f() { }
    private void g() { }

    invariant()
    {
        f();  // 错误，不能在 invariant（不变量）中调用公有成员函数
        g();  // 正确，g() 不是 public（公有的）
    }
}
```

可以使用 `assert()` 检查类对象的不变量，例如：

```
Date mydate;
...
assert(mydate);          // 检查类 Date 的 invariant（不变量） 的内容
```

如果不变量检查失败，将抛出一个 `AssertError` 异常。类不变量会被继承，也就是，任何类的不变量都隐式地包含其基类的不变量。

每个类只能有一个 *不变量*。

当编译生成发布版时，不会生成不变量检查代码，这样程序就会以最高速度运行。

13.12 单元测试

单元测试:

```
        unittest 函数体
```

单元测试(Unit Tests)是用来测试一个类是否正常工作的一系列测试用例。理想情况下，单元测试应该在每次编译时运行一遍。保证单元测试得到运行，并且随同类代码一起维护的最好的方法就是：将测试代码同类的实现代码放置到一起。

类可以有一个特殊的成员函数，叫做：

```
unittest
{
...测试代码...
}
```

编译器开关，如 **-unittest**（对于 **dmd**），会使得单元测试代码被编译并被合并到最后的可执行文件中去。在静态初始化运行完成之后并且在调用 `main()` 函数之前，单元测试代码会被运行。

例如，假定一个类 Sum 用来计算两个值得和：

```
class Sum
{
    int add(int x, int y) { return x + y; }

    unittest
    {
        Sum sum = new Sum;
        assert(sum.add(3,4) == 7);
        assert(sum.add(-2,0) == -2);
    }
}
```

13.13 类分配器(Class Allocators)

```
类分配器:
        new 参数 函数体
```

具有下面形式的类成员函数：

```
new(uint size)
{
    ...
}
```

叫做类分配器。如果第一个参数的类型是 uint，类分配器可以有任意数量的参数。可以为类定义多个类分配器，通过通用的函数重载解析规则选择合适的函数。当执行一个 new 表达式：

```
new Foo;
```

并且当 Foo 是拥有分配器的类时，分配器将被调用，第一个参数被设置为分配一个实例所需的以字节为单位的内存大小。分配器必须分配内存并返回一个 void* 指针。如果分配失败，它不必返回一个 **null**，但是必须抛出一个异常。如果分配器有多于一个的参数，余下的参数将在 **new**（在 *New 表达式里*）之后的括号中出现。

```
class Foo
{
    this(char[] a) { ... }

    new(uint size, int x, int y)
    {
        ...
    }
}

...

new(1,2) Foo(a);         // 调用 new(Foo.sizeof,1,2)
```

如果没有指定类分配器，派生类将继承基类的类分配器。

如果实例是创建在堆栈上，则类分配器不会被调用。

另见 显式类实例分配。

13.14 类释放器(Class Deallocators)

类释放器:
delete *形式参数 函数体*

具有下面形式的类成员函数：

```
delete(void *p)
{
    ...
}
```

叫做类释放器。类释放器有且仅有一个类型为 void* 的参数。一个类只能有一个类释放器。当执行一个 delete 表达式：

```
delete f;
```

且 f 是拥有释放器的一个类的实例时，如果类有析构函数，会调用析构函数，然后释放器被调用，一个指向类实例的指针被传递给释放器。释放内存是释放器的责任。

如果不特别指定，派生类会继承基类所有的释放器。

如果实例是创建在堆栈上，则类分配器不会被调用。

另见 显式类实例分配。

13.15 Alias This

AliasThis:
alias *标识符* **this;**

AliasThis 声明会命名另一个类或结构成员，所有未定义的反查(lookup)都会被转向那里。 *标识符* 命名该成员。

类或结构会隐式转换成 *AliasThis* 成员。

每一个类或结构仅有一个 *AliasThis*。

```
struct S
{   int x;
    alias x this;
}

int foo(int i) { return i * 2; }

void test()
{
    S s;
    s.x = 7;
    int i = -s;    // i == -7
    i = s + 8;     // i == 15
    i = s + s;     // i == 14
    i = 9 + s;     // i == 16
    i = foo(s);    // 隐式转换成 int
}
```

13.16 域类(Scope Classes)

一个 scope 类指的是带有 **scope** 属性的类，像这样：

```
scope class Foo { ... }
```

scope 特征是继承的，这样任何从一个 scope 类派生得到的类也带有 scope。

一个 scope 类引用仅能做为一个函数局部变量。它必须声明成 **scope**：

```
scope class Foo { ... }

void func()
{
    Foo f;       // 错误，scope 类的引用必须是 scope
    scope Foo g = new Foo();     // 正确
}
```

当一个 scope 类引用超出作用域时，它的析构函数（如有的话）会被自动调用。即使通过抛出的异常退出了该域，这条原则也成立。

13.17 最终类(Final Classes)

最终类(Final classes)不能分出子类：

```
final class A { }
```

```
class B :A { }  // 错误，类 A 是最终的(final)
```

13.18 嵌套类(Nested Classes)

*嵌套类*指的是声明在一个函数或另一个类作用域里的类。嵌套类可以访问嵌套它的类和函数里的变量和其它符号：

```
class Outer
{
    int m;

    class Inner
    {
        int foo()
        {
            return m;    // 正确访问 Outer 的成员
        }
    }
}

void func()
{   int m;

    class Inner
    {
        int foo()
        {
            return m;    // 正确访问 func() 的局部变量 m
        }
    }
}
```

如果嵌套类有 **static** 属性，那么它就不能访问封闭域里的变量——它们对于堆栈是局部的或者需要的是 **this**：

```
class Outer
{
    int m;
    static int n;

    static class Inner
    {
        int foo()
        {
            return m;    // 错误，Inner 是静态的，并且 m 需要 this
            return n;    // 正确，n 是静态的
        }
    }
}

void func()
{   int m;
    static int n;
```

```
    static class Inner
    {
        int foo()
        {
            return m;    // 错误，Inner 是静态的，并且 m 对于堆栈是局部的
            return n;    // 正确，n 是静态的
        }
    }
}
```

非静态见嵌套类的通过包含一个额外的隐藏成员（叫做上下文相关指针）发挥作用，而隐藏成员指的是：封闭函数的帧指针（如果它是嵌套在一个函数内部的话）；封闭的类实例的 **this** 指针（如果它是嵌套在一个类内部的话）。

当非静态嵌套类被实例化时，上下文相关指针会在该类的构造函数被调用之前会分配，因此构造函数可以完全访问那些封闭变量。非静态嵌套类只有在必需的上下文相关指针信息可用时才会被实例化：

```
class Outer
{
    class Inner { }

    static class SInner { }
}

void func()
{
    class Nested { }

    Outer o = new Outer;         // 正确
    Outer.Inner oi = new Outer.Inner;    // 错误，对于 Outer 没有'this'
    Outer.SInner os = new Outer.SInner; // 正确

    Nested n = new Nested;       // 正确
}
```

由于非静态嵌套类可以访问它的封闭函数的栈变量，因此一旦封闭函数退出，则该访问也就变得无效。

```
class Base
{
    int foo() { return 1; }
}

Base func()
{   int m = 3;

    class Nested : Base
    {
        int foo() { return m; }
    }
```

```
    Base b = new Nested;

    assert(b.foo() == 3);        // 正确，func() 还存在
    return b;
}

int test()
{
    Base b = func();
    return b.foo();              // 错误，func().m 未定义
}
```

如果需要的这种功能话，则让其工作的方式就是为嵌套类的构造函数里的所需要的变量制做一个复本：

```
class Base
{
    int foo() { return 1; }
}

Base func()
{   int m = 3;

    class Nested : Base
    {   int m_;

        this() { m_ = m; }
        int foo() { return m_; }
    }

    Base b = new Nested;

    assert(b.foo() == 3);        // 正确，func() 还存在
    return b;
}

int test()
{
    Base b = func();
    return b.foo();              // 正确，正在使用缓存的 func().m 复本
}
```

this 可以被用于内层类实例的创建，方式就是在它的 *New 表达式* 前面添加前缀：

```
class Outer
{   int a;

    class Inner
    {
        int foo()
        {
            return a;
        }
    }
}
```

```
int bar()
{
    Outer o = new Outer;
    o.a = 3;
    Outer.Inner oi = o.new Inner;
    return oi.foo();     // 返回 3
}
```

这里 **o** 将 *this* 提供给 **Outer** 的外层类实例。

嵌套类里使用的 **.outer** 特性会给出指向它的封闭类的 **this** 指针。如果封闭的上下文不是一个类，则 **.outer** 给出出指向它的指针，而类型为 **void***。

```
class Outer
{
    class Inner
    {
        Outer foo()
        {
            return this.outer;
        }
    }

    void bar()
    {
        Inner i = new Inner;
        assert(this == i.foo());
    }
}

void test()
{
    Outer o = new Outer;
    o.bar();
}
```

13.18.1 匿名嵌套类(Anonymous Nested Classes)

匿名嵌套类要使用 *New 匿名类表达式* 定义和实例化：

New 匿名类表达式:
new (Peren 参数列表)$_{可选的}$ **class** (Peren 参数列表)$_{可选的}$ *父类*$_{可选的}$ *多个接口类*$_{可选的}$ *类过程体*

Peren 参数列表:
(*参数列表***)**

这个等同于：

```
class 标识符 :父类 接口类
      类过程体
```

```
new (参数列表) 标识符 (参数列表);
```

这里 *标识符* 指的是为匿名嵌套类生成的名字。

13.18.2 常量与不变量类

如果一个*类声明* 拥有一个 `const` 或 `invariant` 存储类别，那么它表示的就是该类的每一个成员都会被声明成该存储类别。如果一个基类时 const 或 invariant，那么所有由它导出的类也是 const 或 invariant。

第 14 章 接口

```
接口声明：
        interface 标识符 接口体
        interface 标识符 :上层接口 接口体

上层接口
        标识符
        标识符 , 上层接口

接口体：
        { 多个声明定义 }
```

接口(Interface)描述的是继承自某个接口的类所必须实现的一系列函数。一个接口的实现类可以被转换成对该接口的引用。

有些操作系统对象，如 Win32 里的 COM/OLE/ActiveX，拥有特定的接口。跟 COM/OLE/ActiveX 相兼容的 D 接口叫做 *COM 接口*。

C++ 接口 是接口的又种形式，表示的是跟 C++ 二进制兼容。

接口不能由类派生，只能派生自其它接口。类不能多次派生自一个接口。

```
interface D
{
    void foo();
}

class A : D, D  // 错误，多重接口
{
}
```

不允许创建接口的实例。

```
interface D
{
    void foo();
}

...

D d = new D();           // 错误，不能创建接口的实例
```

接口成员函数不能有实现。

```
interface D
{
    void bar() { }       // 错误，不允许实现
}
```

继承接口的类必须实现接口中的所有函数：

```
interface D
{
    void foo();
}
```

```
class A : D
{
    void foo() { }       // 正确，提供了实现
}

class B : D
{
    int foo() { }        // 错误，没有提供 void foo() 实现
}
```

接口可以被继承，其中的函数可以被重写：

```
interface D
{
    int foo();
}

class A : D
{
    int foo() { return 1; }
}

class B : A
{
    int foo() { return 2; }
}

...

B b = new B();
b.foo();                 // 返回 2
D d = cast(D) b;         // 正确，因为 B 继承了 A 的 D 实现
d.foo();                 // 返回 2;
```

接口可以在派生类中重新实现：

```
interface D
{
    int foo();
}

class A : D
{
    int foo() { return 1; }
}

class B : A, D
{
    int foo() { return 2; }
}

...

B b = new B();
```

```
b.foo();                    // 返回 2
D d = cast(D) b;
d.foo();                    // 返回 2
A a = cast(A) b;
D d2 = cast(D) a;
d2.foo();                   // 返回 2，尽管它是 A 的 D，不是 B 的 D
```

如果类要重新实现接口，必须重新实现接口的所有函数，不能从父类中继承：

```
interface D
{
    int foo();
}

class A : D
{
    int foo() { return 1; }
}

class B : A, D
{
}                // 错误，接口 D 没有 foo()
```

14.1 常量与不变量接口

如果一个接口拥有 const 或 immutable 存储类别，那么它的所有成员也会是 const 或 immutable。这个储存类别不会被继承。

14.2 COM 接口

接口的一个变体是 COM 接口。按照设计，COM 接口被为直接映射到 Windows COM 对象。任何 COM 对象都由一个 COM 接口表示，任何带有 COM 接口的 D 对象都可以被外部 COM 客户端使用。

按定义，COM 接口从 std.c.windows.com.IUnknown 接口派生。COM 接口与普通 D 接口的不同之处在于：

- 它从 std.c.windows.com.IUnknown 接口派生。
- 它不能成为 *Delete 表达式* 的参数。
- 引用不能向上类型转换为封闭的类对象，也不能向下类型转换为从它派生的接口。为了达到这个目的，必须按照标准的 COM 风格为接口实现一个合适的 QueryInterface() 方法。
- 由 COM 接口导出的类叫做 COM 类。
- COM 类的成员函数的默认连接属性(linkage)为：extern(System)。
- vtbl[] 的第一个成员函数并非指向 InterfaceInfo 的指针，而是第一个虚函数的指针。

14.3 C++ 接口

C++ 接口指的就是用来声明 C++ 连接属性的接口：

```
extern (C++) interface Ifoo
{
    void foo();
    void bar();
}
```

上面所表示的是对应于下面的 C++ 声明：

```
class Ifoo
{
    virtual void foo();
    virtual void bar();
};
```

所有由 C++ 接口所派生的接口也是 C++ 接口。C++ 接口与 D 接口的不同之处在于：

- 它不能成为 *Delete 表达式* 的参数。
- 引用不能向上类型转换为封闭的类对象，也不能向下类型转换为从它派生的接口。
- C++ 调用协定是接口的成员函数的默认转换，而 D 调用协定不是。
- `vtbl[]` 的第一个成员函数并非指向 `Interface` 的指针，而是第一个虚函数的指针。

第 15 章 枚举 — 列举类型

```
枚举声明：
        enum 枚举标记 枚举体
        enum 枚举体
        enum 枚举标记 :枚举基类型 枚举体
        enum :枚举基类型 枚举体

枚举标记：
        标识符

枚举基类型：
        类型

枚举体：
        ;
        { 多个枚举成员 }

多个枚举成员：
        单个枚举成员
        单个枚举成员 ,
        单个枚举成员 , 多个枚举成员

单个枚举成员：
        标识符
        标识符 = 赋值表达式
        类型 标识符 = 赋值表达式
```

枚举声明被用来定义一组常量。定义形式有两种：

1. 命名枚举——它有一个*枚举标记*。
2. 匿名枚举——它没有*枚举标记*。

15.1 命名枚举

命名枚举常被用来声明几个有关联的常量，并且使用一个统一的类型来组合它们。*多个枚举成员* 被声明在*枚举标记*的作用域内。*枚举标记* 会声明出一个新的类型，而所有的那些 *枚举成员* 都属于该类型。

下面的例子就定义了一种新的类型 `X` ——它拥有值 `X.A=0`、`X.B=1`、`X.C=2`：

```
enum X { A, B, C }      // 命名枚举
```

如果 *枚举基类型* 没有被显示设置，并且第一个 *枚举成员* 有初始值，那么该初始值的类型就为枚举的类型。否则，默认设置其类型为 `int`。

命名枚举成员不可能有自个的 *类型*。

命名枚举成员可以被隐式转换成它的 *枚举基类型*，不过 *枚举基类型* 不会被隐式转换成枚举类型。

一个 *枚举成员* 值由初始化过程设定。如果没有初始化过程，则它会被设定成前一个 *枚举*

成员 的值 +1。如果它是第一个 *枚举成员*，那么它的值就为 0。

枚举必须至少拥有一个成员。

15.1.1 枚举默认的初始化过程

枚举类型的特性 .init 就是该枚举的第一个成员的值。它也是枚举类型的默认初始化过程。

```
enum X { A=3, B, C }
X x;             // ax 被初始化为 3
```

15.1.2 枚举的特性

枚举特性仅对命令枚举才有效。

.init	第一个枚举成员的值
.min	枚举的最小值
.max	枚举的最大值
.sizeof	存储枚举值所需要的存储器大小

例如：

```
enum X { A=3, B, C }
X.min     // 为 X.A
X.max     // 为 X.C
X.sizeof  // 等同于 int.sizeof
```

命名枚举的 *枚举基类型* 必须支持比较操作，目的就是为了能够计算特性 .max 和 .min。

15.2 匿名枚举

如果不指定 enum *标识符*，则枚举称为 *匿名枚举*，并且 *多个枚举成员* 在 *枚举声明* 出现的作用域内被声明。没有新类型被创建；*多个枚举成员* 的类型是 *枚举基类型*。

枚举基类型 是枚举的底层类型。

如果忽略，则 *枚举成员* 可以有不同的类型。类型首先由下面这些给定：

1. 为 *类型*，如果存在的话，
2. 为 *赋值表达式* 的类型，如果存在的话。
3. 为前一个 *枚举成员* 的类型，如果存在的话。
4. int

```
enum { A, B, C }         // 匿名枚举
```

定义了常量 A=0、B=1、C=2，所有类型都为 int。

枚举必须至少拥有一个成员。

一个 *枚举成员* 值由初始化过程设定。如果没有初始化过程，则它会被设定成前一个 *枚举成员* 的值 + 1。如果它是第一个 *枚举成员*，那么它的值就为 0。

```
enum { A, B = 5+7, C, D = 8+C, E }
```

设置 A=0、B=12、C=13、D=21 和 E=22，所有的类型都为 int。

```
enum : long { A = 3, B }
```

设置 A=3、B=4，所有类型都为 long。

```
enum : string
{
    A = "hello",
    B = "betty",
C     // 错误，不能把 1 加到 "betty" 上
}

enum
{
A = 1.2f,   // A 为 1.2f，类型是 float
    B,          // B 为 2.2f，类型为 float
    int C = 3,  // C 为 3，类型为 int
    D           // D 为 4，类型为 int
}
```

15.2.1 显著常量(Manifest Constants)

如果一个匿名枚举只有一个成员，则“{ }”可以省略掉。

```
enum i = 4;      // i 为 4，类型为 int
enum long l = 3; // l 为 3，类型为 long
```

这种声明不是左值，即表示不能获取它们的地址的。

第 16 章 常量与不变量

在检查数据结构或接口时，如果能轻易地识别出哪些数据是不希望被更改的、哪些数据可能会被更改或者谁可能会更改这些数据，将会是很有意义的。借助语言的录入系统，我们可以完成这样的要求。数据可以被标记为 const 或 invariant，而默认是可以被更改的，或者说是可变的(*mutable*)。

invariant(不变量) 用于那些不需要变更的数据上。不变数据值一旦被构建，则在整个程序的执行期间都将操持一样。不变数据可以被放在 ROM (Read Only Memory－只读内存)里，也可以被放置在由硬件标记为只读的内存页里。由于不变数据是不会变更的，因此，就有了很多优化程序的机会，而且还可以进行函数式编程(functional style programming)。

const(常量) 用于那些不能被指向它们的常量引用所更改的数据上。不过，这些数据却有可能被指向它们的其它引用所更改。常量主要用于通过接口传递数据（保证不要去修改它们）的情形。

不变量跟常量都具有*传递性(transitive)*，即凡是通过一个不变引用所能到达的数据也是不变的；对于常量也是一样。

16.1 Invariant 存储类别

最简单的 invariant 声明就是把它当作一个存储类别来处理。它可以用来声明那些一目了然的常量。

```
invariant int x = 3;     // x 被设置为 3
x = 4;                   // 错误，x 是不变量
char[x] s;               // x 是一个拥有有 3 个 char 型元素的数组
```

数据类型也可以从初始值进行推断：

```
invariant y = 4;         // y 的类型是 int
y = 5;                   // 错误，y 是不变量
```

如果不存在初始值，那么该不变量也可以通过相应的构造函数来进行初始化：

```
invariant int z;
void test()
{
    z = 3;               // 错误，z 是不变量
}
static this()
{
z = 3;     // 正确，可以设置没有静态初始值的
             // 不变量
}
```

对于声明在非本地(non-local)的不变量的初始值，则要求它们在编译时是可计算的：

```
int foo(int f) { return f * 3; }
int i = 5;
invariant x = 3 * 4;       // 正确，为 12
invariant y = i + 1;       // 错误，在编译时不能进行计算
```

```
invariant z = foo(2) + 1; // 正确，foo(2) 可以在编译时计算，结果为 7
```

对于声明为非静态(non-static)的局部不变量的初始值，则要求它们在运行时也是可计算的：

```
int foo(int f)
{
invariant x = f + 1;  // 在编译时进行计算
  x = 3;                 // 错误，x 是不变量
}
```

因为不变量具有传递性(transitive)，因此被一个不变引用的数据也是不变的。

```
invariant char[] s = "foo";
s[0] = 'a';  // 错误，s 引用的是不变数据
s = "bar";   // 错误，s 是不变量
```

invariant 声明可以做为左值(lvalues)，即是说可以获得它们的地址，并且它们占用存储(occupy storage)。

16.2 Const 存储类别

const 声明大部分情况下跟 invariant 声明是一样的，不同之处在于：

- 所有被 const 声明引用的任何数据都不能由该 const 声明更改，但是有可能被其它指向该数据的引用所更改。
- 一个 const 声明的类型本身是常量(const)。

16.3 Invariant 类型

其值确定不会被更改的数据可以被定义成 invariant 类型。关键字 invariant 还可以被用作一个*类型构造器(type constructor)*：

```
invariant(char)[] s = "hello";
```

invariant 作用到了圆括号里面的类型上。因此，虽然 s 可以被赋予新值，但 s[] 的内容却不能被变更：

```
s[0] = 'b';  // 错误，s[] 是不变量
s = null;    // 正确，s 本身不是不变量
```

“不变性”是可传递的，即表示它对于由 invariant 类型所引用的任何数据都有效：

```
invariant(char*)** p = ...;
p = ...;         // 正确，p 不是不变量
*p = ...;        // 正确，*p 不是不变量
**p = ...;       // 错误，**p 是不变量
***p = ...;      // 错误，***p 是不变量
```

用作储存类别的 invariant 等同于将 invariant 作为一个类型构造器用于整个类型的声明：

```
invariant int x = 3;   // x 的类型是 invariant(int)
invariant(int) y = 3;  // y 是不变量
```

16.4 建立不变数据

第一种方式就是使用已经是不变量的文字量，如字符串文字。字符串文字总是不变的(invariant)。

```
auto s = "hello";   // s 为 invariant(char)[5]
char[] p = "world"; // 出错，因为不能隐式地将不变量
                    // 转换成可变的(mutable)
```

第二种方式就是将数据经类型转换(cast)成不变量此时，就需要程序员来确保没有其它可变引用指向访相同的数据。

```
char[] s = ...;
invariant(char)[] p = cast(invariant)s;      // 未定义行为
invariant(char)[] p = cast(invariant)s.dup; // 正确，单独引用
```

特性 .idup 是一种很方便的创建一个数组的不变量复本的方式：

```
auto p = s.idup;
p[0] = ...;        // 错误，p[] 是不变量
```

16.5 使用类型转换来移除不变性

invariant 类型可以使用类型转换(cast)来移除：

```
invariant int* p = ...;
int* q = cast(int*)p;
```

但并不表示，我们就可以更改该数据：

```
*q = 3; // 跃然编译器允许，但结果却属于“未定义行为”
```

将“invariant 正确性”的检验移除掉，这在有些情形下是很有必要的，这样的情形有：静态录入不正确或不可调整，如在引用位于库里的代码，它们就是不能更改的。类型转换(cast)总是一种生硬而有效的手段，在使用它来转换掉“invariant 正确性”的检验时，我们必须负责确保数据的不变性，因为编译器对此已无能为力了。

16.6 不变成员函数

不变成员函数(invariant member functions) 暗示该对象以及由 this 引用所指向的所有东西都是不变的。像这样声明它们：

```
struct S
{   int x;

   invariant void foo()
```

```
    {
        x = 4;       // 错误，x 是不变量
        this.x = 4; // 错误，x 是不变量
    }
```

const 和 invariant 函数属性也可以出现在参数列表的反括号后面：

```
struct S
{
    void bar() invariant
    {
    }
}
```

16.7 Const 类型

const 类型很像 invariant 类型，不同之处在于 const 形式上是一个数据的只读*视图(view)*。而该相同数据的其它别名则可能在任何时候更改它。

16.8 常量成员函数

常量成员函数(const member functions)指的是那些不允许通过该成员函数的 this 引用更改该对象的其它任何部分的函数。

16.9 隐式转换

可变(mutable)和不变(invariant)类型都可以被隐式转换成常量。可变类型不能隐式转换成不变类型，反之也是一样。

16.10 D 的不变量和常量跟 C++ 的常量进行比较

比较常量、不变量

功能	D	C++98
关键字 const	有	有
关键字 invariant	有	无

常量符号	函数式：
	`// 指向 const int 的 const 指针的指针`
	`const(int*)* p;`
		后缀式：
		`// 指向 const int 的 const 指针的指针`
		`const int *const *p;`

功能	D	C++98

可递常量	有：
	`// 指向 const int 的 const 指针的 const 指针`
	`const int** p;`
	`**p = 3; // 错误`
		无：
		`// 指向 int 的指针的 const 指针`
		`int** const p;`
		`**p = 3;  // 正确`

类型转换掉常量性	有：
	`// 指向 const int 的指针`
	`const(int)* p;`
	`int* q = cast(int*)p; // 正确`
		有：
		`// 指向 const int 的指针`
		`const int* p;`
		`int* q =`
		`const_cast<int*>p; // 正确`

对类型转换去掉常量性后的数据修改	无：
	`// 指向 const int 的指针`
	`const(int)* p;`
	`int* q = cast(int*)p;`
	`*q = 3;  // 未定义行为`
		有：
		`// 指向 const int 的指针`
		`const int* p;`
		`int* q = const_cast<int*>p;`
		`*q = 3;  // 正确`

顶级常量的重载	有：
	`void foo(int x);`
	`void foo(const int x);  // 正确`
		无：
		`void foo(int x);`
		`void foo(const int x);  // 出错`

常量跟可变量混用	有：
	`void foo(const int* x, int* y)`
	`{`
	`  bar(*x); // bar(3)`
	`  *y = 4;`
	`  bar(*x); // bar(4)`
	`}`
	`...`
	`int i = 3;`
	`foo(&i, &i);`
		有：
		`void foo(const int* x, int* y)`
		`{`
		`  bar(*x); // bar(3)`
		`  *y = 4;`
		`  bar(*x); // bar(4)`
		`}`
		`...`
		`int i = 3;`
		`foo(&i, &i);`

不变量跟可变量的混用	无：
	`void foo(invariant int* x, int* y)`
	`{`
	`  bar(*x); // bar(3)`
	`  *y = 4;  // 未定义行为`
	`  bar(*x); // bar(??)`
		无 invariant

功能	D	C++98

	`}`
	`...`
	`int i = 3;`
	`foo(cast(invariant)&i, &i);`	

字符串文字类型	invariant(char)[]	const char*
字符串文字到非常量的隐式转换	不允许	允许，但已丢弃

第 17 章 函数

```
函数体：
        块语句
        语句体
        In 语句 语句体
        Out 语句 语句体
        In 语句 Out 语句 语句体
        Out 语句 In 语句 语句体

In 语句：
        in 语句块

Out 语句：
        out 语句块
        out ( 标识符 ) 语句块

语句块：
        body 语句块
```

17.1 纯函数(Pure Functions)

纯函数(Pure Functions)指的是那些对相同的参数会产生同一结果的函数。概括一下，要求一个纯函数：

- 其参数都为 invariant 或者可以隐式地转换到 invariant
- 不会读取或写入任何全局可变状态

纯函数会抛出异常并通过 *New 表达式* 分配内存。

```
int x;
invariant int y;
const int* pz;

pure int foo(int i,      // 正确，可隐式转换成 invariant
           char* p,          // 错误，不是 invariant
           const char* q,    // 错误，不是 invariant
           invariant int* s // 正确，它是 invariant
          )
{
x = i;     // 错误，正在修改全局状态
    i = x;     // 错误，正在读取可变全局状态
    i = y;     // 正确，正在读取 invariant 全局状态
    i = *pz;   // 错误，正在读取 const 全局状态
    return i;
}
```

17.2 Nothrow 函数

Nothrow 函数不会抛出派生自类 *Exception* 的任何异常。

17.3 Ref 函数

Ref 函数允许函数通过引用返回。这点等同于 ref 函数参数。

```
ref int foo()
{   auto p = new int;
    return *p;
}
...
foo() = 3;  // 引用返回可以是左值
```

17.4 虚函数(Virtual Functions)

虚函数指的是这样一类函数：它们通过被称作 vtbl[] 的函数指针表间接进行调用，而非直接调用。所有的非静态、非私有、非模板成员函数都是虚函数。这听起来也许有些低效，但是因为 D 编译器在生成代码时知道所有类层次结构，所以所有未被重写的函数都可以被优化成非虚函数。事实上，C++ 程序员倾向于“在不确定时，就将其声明为虚函数”；而 D 采用的方式则是“全都声明成虚函数，除非我们可以证明它们可以是非虚函数”，如此，综合的结果就是可以产生更多更直接的函数调用。这也可以大大减少由于没有将会被重写(voerride)的函数声明为虚函数，而引起的错漏。

因为带有“非 D 连接属性(linkage)”的函数不能是虚函数，因此它们不能被重写。

因为模板成员函数不可能是虚的，因此不能被重写。

标记为 `final` 的函数不能在派生类中被重写，除非它们同时又是 `private` 类型。例如：

```
class A
{
    int def() { ... }
    final int foo() { ... }
    final private int bar() { ... }
    private int abc() { ... }
}

class B : A
{
    int def() { ...}    // 正确，重写 A.def
    int foo() { ...}    // 错误，A.foo 是 final 的
    int bar() { ...}    // 正确，A.bar 是 final private 的，但不是 virtual 的
    int abc() { ...}    // 正确，A.abc 不是 virtual 的，B.abc 是 virtual 的
}

void test(A a)
{
    a.def();    // 调用 B.def
    a.foo();    // 调用 A.foo
    a.bar();    // 调用 A.bar
    a.abc();    // 调用 A.abc
}
```

```
void func()
{    B b = new B();
    test(b);
}
```

允许“协变返回类型(covariant return types)”，即表示在派生类里的重写函数可以返回这样一种类型——此类型派生自被重写函数所返回的类型：

```
class A { }
class B : A { }

class Foo
{
    A test() { return null; }
}

class Bar : Foo
{
    B test() { return null; }    // 重写，并且同 Foo.test() 是协变的
}
```

虚函数都有调用 *this* 引用的隐藏参数，它引用的是函数被调用的那个类对象。

17.5 函数继承(inheritance)和重写(overidding)

派生类中的函数将重写基类中拥有相同函数名以及相同参数类型的函数：

```
class A
{
    int foo(int x) { ... }
}

class B : A
{
    override int foo(int x) { ... }
}

void test()
{
    B b = new B();
    bar(b);
}

void bar(A a)
{
    a.foo(1);    // 调用 B.foo(int)
}
```

而且，在进行重载解析的时候，基类中对应的那些函数都不在考虑之列：

```
class A
{
    int foo(int x) { ... }
    int foo(long y) { ... }
}

class B : A
```

```
{
    override int foo(long x) { ... }
}

void test()
{
    B b = new B();
    b.foo(1);    // 调用 B.foo(long)，因为不会考虑 A.foo(int)
    A a = b;
    a.foo(1);    // 产生运行时错误（实际调用 A.foo(int)）
}
```

要在重载解析过程中让基类的那些函数有效，请使用“*别名声明*”：

```
class A
{
    int foo(int x) { ... }
    int foo(long y) { ... }
}

class B : A
{
    alias A.foo foo;
    override int foo(long x) { ... }
}

void test()
{
    B b = new B();
    bar(b);
}

void bar(A a)
{
    a.foo(1);            // 调用 A.foo(int)
    B b = new B();
    b.foo(1);            // 调用 A.foo(int)
}
```

如果未使用“*Alias 声明*”，那么导出类的函数就会完全重写所有在基类里相同名字的函数，即使在基类里的函数参数类型可能不同，也一样会重写。如果通过隐式转换基类调用了其它的函数，那么就会产生一个 std.HiddenFuncError 异常：

```
import std.hiddenfunc;

class A
{
     void set(long i) { }
     void set(int i)  { }
}
class B : A
{
     void set(long i) { }
```

```
}

void foo(A a)
{   int i;
    try
    {
a.set(3);   // 错误，会抛出运行时异常
                // 因为 A.set(int) 从 B 是不应该可访问的
    }
    catch (HiddenFuncError o)
    {
        i = 1;
    }
    assert(i == 1);
}

void main()
{
    foo(new B);
}
```

如果在你的程序里抛出了一个 HiddenFuncError 异常，则使用重载和重写的话需要在相关类里重新检查一遍。

至于考虑到重载这一问题，如果隐藏函数跟所有其它虚函数脱节(disjoint)的话，HiddenFuncError 异常是不会被抛出。

函数参数的默认值不会被继承：

```
class A
{
    void foo(int x = 5) { ... }
}

class B : A
{
    void foo(int x = 7) { ... }
}

class C : B
{
    void foo(int x) { ... }
}


void test()
{
    A a = new A();
    a.foo();            // 调用 A.foo(5)

    B b = new B();
    b.foo();            // 调用 B.foo(7)

    C c = new C();
    c.foo();            // 错误，C.foo 需要一个参数
}
```

17.6 内联函数(Inline Functions)

D 中没有 inline 关键字。编译器决定是否将一个函数内联，就像 register 关键字不再同编译器是否将变量存在寄存器中相关一样。（也没有 register 关键字。）

17.7 函数重载(Function Overloading)

函数的重载是基于函数的形式参数跟实际参数是有多匹配。通常是*最好*匹配的函数会被选择。匹配的层次为：

1. 无匹配
2. 带有隐式转换的匹配
3. 带有转换成常量的匹配
4. 精确匹配

每一个形式参数（包括 `this` 指针）都会跟函数相应的实际参数进行比较，以决定该形式参数的匹配层次。函数的匹配层次是其形式参数的“*最坏*”匹配层次。

文件(Literal)是不会匹配 `ref` o 或 `out` 实际参数。

如果有两个或更多参数拥有相同的匹配层次，那么就使用“*部分排序*”来试着找出更好的匹配。部分排序(Partial ordering)会找出最为适合特定化的函数。如果没有函数适合特定化，则会产生一个“含糊”错误。部分排序区分函数 `f()` 和 `g()`，方法就是通过取得 `f()` 的实参类型；利用这些类型的默认值构造一个形参列表，并试图把它们跟 `g()` 进行匹配。如果成功，那么 `g()` 至少会跟 `f()` 一样跟特殊化。例如：

```
class A { }
class B : A { }
class C : B { }
void foo(A);
void foo(B);

void test()
{
  C c;
/* foo(A) 和 foo(B) 都使用隐式转换规则匹配。
* 应用部分排序规则，
* foo(B) 不能使用 A 来进行调用，而 foo(A) 可以使用 B 来被调用。
* 因此，foo(B) 会被更多次被特殊化和选择。
   */
foo(c); // 调用 foo(B)
}
```

带有不定个数参数函数相比非不定个数参数的函数会更少被考虑特殊化。

使用非 D 连接属性(linkage)定义的函数不能被重载。因为名字碎解不会把实际参数考虑在里面。

17.8 重载集

在相同作用域声明的函数其重载是彼此互斥的，它们也叫做“ *重载集*”。重载典型的实例就是在模块层定义的函数：

```
module A;
void foo() { }
void foo(long i) { }
```

A.foo() 和 A.foo(long) 组成了一个重载集。不同模块可以使用相同名字定义函数：

```
module B;
class C { }
void foo(C) { }
void foo(int i) { }
```

而且 A 和 B 可以被第三个模块导入。两个重载集 A.foo 和 B.foo 都可以被找到。foo 的实例可以基于它在一个重载集里的准确匹配来进行选择：

```
import A;
import B;

void bar(C c)
{
foo();    // 调用 A.foo()
   foo(1L);  // 调用 A.foo(long)
   foo(c);   // 调用 B.foo(C)
   foo(1,2); // 错误，没有任何 foo 可以匹配
   foo(1);   // 错误，匹配到 A.foo(long) 和 B.foo(int)
   A.foo(1); // 调用 A.foo(long)
}
```

即使 B.foo(int) 相比 A.foo(long) 有着更好的匹配用于 foo(1)，它还是一个错误；因为两个匹配处于不同的重载集里。

重载集可以使用 alias 声明进行合并：

```
import A;
import B;

alias A.foo foo;
alias B.foo foo;

void bar(C c)
{
foo();    // 调用 A.foo()
   foo(1L);  // 调用 A.foo(long)
   foo(c);   // 调用 B.foo(C)
   foo(1,2); // 错误，没有任何 foo 可以匹配
   foo(1);   // 调用 B.foo(int)
   A.foo(1); // 调用 A.foo(long)
}
```

17.8.1 函数形式参数(Function Parameters)

实参存储类别有 **in**、**out**、**ref**、**lazy**、**final**、**const**、**invariant** 或 **scope**。例如：

```
int foo(in int x, out int y, ref int z, int q);
```

x 为 **in**，y 为 **out**，z 为 **ref**，而 q 为空。

存储类别 **in** 等同于 **const scope**。

如果没有指定存储类型，实际参数就会变成形参的一个可变复本。

- 函数声明清楚的表示了哪些是函数的输入，哪些是函数的输出。
- 不再需要单独使用一种叫 IDL 语言。
- 可以给编译器提供更多的信息，从而可以提供更好的错误检查并生成更好的代码。

out 参数被设为对应类型的默认值。例如：

```
void foo(out int x)
{
// 在 foo() 的开始，x 被设置为 0
}

int a = 3;
foo(a);
// a 现在为 0


void abc(out int x)
{
    x = 2;
}

int y = 3;
abc(y);
// y 现在为 2


void def(ref int x)
{
    x += 1;
}

int z = 3;
def(z);
// z 现在为 4
```

对于通过引用传递的动态数组和对象参数，in/out/ref 只会作用到该引用上而不会是那些内容里。

懒式参数被求值的时间不是以函数被调用时，而是参数在函数内被求值时进行的。因此，懒式参数会被计算 0 次或多次。懒式参数不能是一个左值。

```
void dotimes(int n, lazy void exp)
{
    while (n--)
        exp();
}
```

```
void test()
{   int x;
    dotimes(3, writefln(x++));
}
```

控制台输出：

```
0
1
2
```

void 类型的懒式参数不能接受一个任何类型的参数。

17.8.2 函数默认参数

函数参数声明可以拥有默认值：

```
void foo(int x, int y = 3)
{
   ...
}
...
foo(4);   // 等同于 foo(4, 3);
```

默认参数是在函数声明的环境里被求值的。如果给定了一个参数的默认值，则所有后继的参数也必须要拥有默认值。

17.9 参数可变型函数

那些带有可变数目参数的函数就叫做参数可变型(variadic)函数。一个参数可变型函数可以有下面三种形式：

1. C-风格的参数可变型函数
2. 带有类型信息的参数可变型函数
3. 类型安全的参数可变型函数

C-风格的参数可变型函数

C-风格的参数可变型函数被声明时在所必需的函数参数里带有“...”参数。它有一个非 D 连接属性(linkage)，如 extern (C)：

```
extern (C) int foo(int x, int y, ...);

foo(3, 4);       // 正确
foo(3, 4, 6.8); // 正确，参数可变型函数
foo(2);          // 错误，y 参数必不可少
```

必须至少声明一个固定型参数。

```
extern (C) int def(...); // 错误，至少要有一个参数
```

C-风格的参数可变型函数符合 C 对于参数可变型函数的调用协定，而且对于调用像

printf 那样的 C 库函数最有用。这些参数可变型函数的实现会声明一个特殊的局部变量：**_argptr**，此变量是一个指向可变参数里的第一个的 void* 型指针。要访问这些参数，**_argptr** 就必须被转换成预期参数类型的指针：

```
foo(3, 4, 5);   // 第一个可变型参数是 5

int foo(int x, int y, ...)
{   int z;

    z = *cast(int*)_argptr;      // z 被设置为 5
}
```

为了避开不同 CPU 结构上的各种奇特的堆栈分布带来的麻烦，请使用 **std.c.stdarg** 访问那些可变(variadic)参数：

```
import std.c.stdarg;
```

17.9.1 D-风格的参数可变型函数

带有参数和类型信息的参数可变型函数声明方式：在必需的函数参数之后带上参数“...”。它有 D 连接属性(linkage)，因此不需要声明任何非可变型参数：

```
int abc(char c, ...);   // 一个必需的参数： c
int def(...);           // 正确
```

参数可变型函数会声明一个特殊的局部变量：**_argptr**，此变量是一个指向第一个可变参数的 void* 型指针。要访问这些参数，**_argptr** 就必须被转换成预期参数类型的指针：

```
foo(3, 4, 5);   // 第一个可变型参数是 5

int foo(int x, int y, ...)
{   int z;

    z = *cast(int*)_argptr;      // z 被设置为 5
}
```

另一个名叫 **_arguments** 而类型为 TypeInfo[] 的隐藏参数也会被传递给该函数。**_arguments** 会给出参数的数目以及它们的类型，这样可以创建类型安全的参数可变型函数。

```
import std.stdio;

class Foo { int x = 3; }
class Bar { long y = 4; }

void printargs(int x, ...)
{
    writefln("%d arguments", _arguments.length);
    for (int i = 0; i < _arguments.length; i++)
    {   _arguments[i].print();
```

```
        if (_arguments[i] == typeid(int))
        {
            int j = *cast(int *)_argptr;
            _argptr += int.sizeof;
            writefln("\t%d", j);
        }
        else if (_arguments[i] == typeid(long))
        {
            long j = *cast(long *)_argptr;
            _argptr += long.sizeof;
            writefln("\t%d", j);
        }
        else if (_arguments[i] == typeid(double))
        {
            double d = *cast(double *)_argptr;
            _argptr += double.sizeof;
            writefln("\t%g", d);
        }
        else if (_arguments[i] == typeid(Foo))
        {
            Foo f = *cast(Foo*)_argptr;
            _argptr += Foo.sizeof;
            writefln("\t%X", f);
        }
        else if (_arguments[i] == typeid(Bar))
        {
            Bar b = *cast(Bar*)_argptr;
            _argptr += Bar.sizeof;
            writefln("\t%X", b);
        }
        else
            assert(0);
    }
}

void main()
{
    Foo f = new Foo();
    Bar b = new Bar();

    writefln("%X", f);
    printargs(1, 2, 3L, 4.5, f, b);
}
```

其输出为：

```
00870FE0
5 arguments
int
        2
long
        3
double
```

```
       4.5
Foo
       00870FE0
Bar
       00870FD0
```

为了避开不同 CPU 结构上的各种奇特的堆栈分布带来的麻烦，请使用 **std.stdarg** 访问那些可变(variadic)参数：

```
import std.stdio;
import std.stdarg;

void foo(int x, ...)
{
   writefln("%d arguments", _arguments.length);
   for (int i = 0; i < _arguments.length; i++)
   {   _arguments[i].print();

       if (_arguments[i] == typeid(int))
       {
          int j = va_arg!(int)(_argptr);
          writefln("\t%d", j);
       }
       else if (_arguments[i] == typeid(long))
       {
          long j = va_arg!(long)(_argptr);
          writefln("\t%d", j);
       }
       else if (_arguments[i] == typeid(double))
       {
          double d = va_arg!(double)(_argptr);
          writefln("\t%g", d);
       }
       else if (_arguments[i] == typeid(FOO))
       {
          FOO f = va_arg!(FOO)(_argptr);
          writefln("\t%X", f);
       }
       else
          assert(0);
   }
}
```

17.9.2 类型安全的参数可变型函数

使用类型安全的参数可变型函数的情形是：参数的可变部分被用于创建一个数组或一个类对象。

对于数组：

```
int test()
{
return sum(1, 2, 3) + sum(); // 返回 6+0
}

int func()
{
    int[3] ii = [4, 5, 6];
    return sum(ii);              // 返回 15
}

int sum(int[] ar ...)
{
    int s;
    foreach (int x; ar)
        s += x;
    return s;
}
```

对于静态数组：

```
int test()
{
    return sum(2, 3);   // 错误，对于数组需要 3 个值
    return sum(1, 2, 3); // 返回 6
}

int func()
{
    int[3] ii = [4, 5, 6];
    int[] jj = ii;
    return sum(ii);              // 返回 15
    return sum(jj);              // 错误，类型不匹配
}

int sum(int[3] ar ...)
{
    int s;
    foreach (int x; ar)
        s += x;
    return s;
}
```

对于类对象：

```
class Foo
{
    int x;
    char[] s;

    this(int x, char[] s)
    {
        this.x = x;
        this.s = s;
    }
}
```

```
void test(int x, Foo f ...);

...

Foo g = new Foo(3, "abc");
test(1, g);              // 正确，因为 g 是 Foo 的实例
test(1, 4, "def");       // 正确
test(1, 5);              // 错误，不能匹配 Foo 的构造函数
```

实现可能在堆栈上构造对象或数组实例。因此，在参数可变型返回之后引用该实例就会出现一个错误：

```
Foo test(Foo f ...)
{
    return f;    // 错误，在返回之后 f 实例内容无效
}

int[] test(int[] a ...)
{
    return a;            // 错误，在返回之后数组内容无效
    return a[0..1];      // 错误，在返回之后数组内容无效
    return a.dup;        // 正确，因为做了复本
}
```

对于其它类型，参数会自己建立，就像下面的：

```
int test(int i ...)
{
    return i;
}

...
test(3);         // 返回 3
test(3, 4);      // 错误，太多参数
int[] x;
test(x);         // 错误，类型不匹配
```

17.9.3 懒式参数可变型函数

如果可变型参数是一个不带参数的委托数组：

```
void foo(int delegate()[] dgs ...);
```

那么跟该委托的类型不匹配的每一个参数会被转换成一个委托。

```
int delegate() dg;
foo(1, 3+x, dg, cast(int delegate())null);
```

同等于：

```
foo( { return 1; }, { return 3+x; }, dg, null );
```

17.10 局部变量

如果使用一个未被赋值过的局部变量，会被视为错误。编译器的实现未必总能够检测到这些情况。其他语言的编译器有时会为此发出警告，但是因为这种情况几乎总是意味着存在错漏，所以应该把它处理为错误。

如果声明一个变量但却从未使用，会被视为错误。死变量，如同过时的死代码一样，都会使维护者迷惑。

如果声明的一个变量掩盖了统一函数中的另一个变量，会被视为错误：

```
void func(int x)
{   int x;              错误，掩盖了前面定义的 x
    double y;
    ...
    {   char y;         错误，掩盖了前面定义的 y
        int z;
    }
    {   wchar z;        合法，上一个 z 超出了作用域
    }
}
```

因为这种情况看起来不合理，在实践中出现这种情况时不是一个错漏至少看起来也像一个错漏。

如果返回一个局部变量的地址或引用，会被视为错误。

如果局部变量同标签同名，会被视为错误。

17.11 嵌套函数

函数可以被嵌套在其它函数内部：

```
int bar(int a)
{
   int foo(int b)
   {
      int abc() { return 1; }

      return b + abc();
   }
   return foo(a);
}

void test()
{
   int i = bar(3);     // i 被赋值为 4
}
```

嵌套函数只在其名字处在作用域中时才能被访问。

```
void foo()
{
  void A()
  {
```

```
B();    // 错误，B() 被向前引用了
   C(); // 错误，C 未定义
   }
   void B()
   {
      A();      // 正确，在作用域内
      void C()
      {
         void D()
         {
A(); // 正确
   B(); // 正确
            C();       // 正确
            D();       // 正确
         }
      }
   }
A(); // 正确
   B(); // 正确
   C(); // 错误，C 未定义
}
```

和：

```
int bar(int a)
{
    int foo(int b) { return b + 1; }
    int abc(int b) { return foo(b); }    // 正确
    return foo(a);
}

void test()
{
    int i = bar(3);      // 正确
    int j = bar.foo(3); // 错误，bar.foo 不可见
}
```

嵌套函数可以访问由词法上封闭函数所定义的变量和其它符号。这里的“访问”包含既可以读，也可以写两种能力。

```
int bar(int a)
{   int c = 3;

    int foo(int b)
    {
       b += c;          // 4 被加到 b 上
       c++;             // bar.c 现在为 5
       return b + c;    // 返回 12
    }
    c = 4;
```

```
    int i = foo(a);      // i 被设置为 12
    return i + c;        // 返回 17
}

void test()
{
    int i = bar(3);      // i 被赋值为 17
}
```

这种访问能力能够跨越多重嵌套：

```
int bar(int a)
{   int c = 3;

    int foo(int b)
    {
        int abc()
        {
            return c;   // 访问 bar.c
        }
        return b + c + abc();
    }
    return foo(3);
}
```

静态嵌套函数不能访问外围函数的任何堆栈变量，但能访问静态变量。这种行为同静态成员函数类似。

```
int bar(int a)
{   int c;
    static int d;

    static int foo(int b)
    {
        b = d;           // 正确
        b = c;           // 错误，foo() 不能访问 bar() 的堆栈帧
        return b + 1;
    }
    return foo(a);
}
```

函数可以嵌套在成员函数内：

```
struct Foo
{   int a;

    int bar()
    {   int c;

        int foo()
        {
            return c + a;
        }
        return 0;
    }
```

```
}
```

嵌套结构和嵌套类的成员函数不能访问外围函数的堆栈变量，但是能访问其他的符号：

```
void test()
{   int j;
    static int s;

    struct Foo
    {   int a;

        int bar()
        {   int c = s;          // 正确，s 是静态的
            int d = j;          // 错误，不能访问 test() 的堆栈帧

            int foo()
            {
                int e = s;      // 正确，s 是静态的
                int f = j;      // 错误，不能访问 test() 的堆栈帧
                return c + a;   // 正确，bar() 的堆栈帧是可访问的，
                                // 通过指向 Foo.bar() 的 this 指针
                                // 可以访问 Foo 的成员
            }

            return 0;
        }
    }
}
```

嵌套函数总是使用 D 函数连接属性(linkage)类型。

同模块级的声明不同，函数作用域的声明会按照声明的顺序处理。这意味着连个嵌套函数不能互相调用：

```
void test()
{
    void foo() { bar(); }       // 错误，bar 没有被定义
    void bar() { foo(); }       // 正确
}
```

解决的方法是使用委托：

```
void test()
{
    void delegate() fp;
    void foo() { fp(); }
    void bar() { foo(); }
    fp = &bar;
}
```

未来的方向 这个限制可能会被删除。

17.11.1 委托、函数指针和闭包

函数指针可以指向一个静态嵌套函数：

```
int function() fp;

void test()
{   static int a = 7;
    static int foo() { return a + 3; }

    fp = &foo;
}

void bar()
{
    test();
    int i = fp();       // i 被设置为 10
}
```

委托可以用非静态嵌套函数赋值：

```
int delegate() dg;

void test()
{   int a = 7;
    int foo() { return a + 3; }

    dg = &foo;
    int i = dg();       // i 被设置为 10
}
```

被嵌套函数引用的栈变量仍然是有效的，即使是在函数退出后（这点跟 D 1.0 有所不同）。这就叫*闭包(closure)*。栈变量的返回地址不过不是闭包，而是一个错误。

```
int* bar()
{   int b;
    test();
    int i = dg();       // 正确，test.a 在一个闭包内，且还存在
    return &b;          // 错误，bar.b 在 bar() 退出后不再有效
}
```

非静态嵌套函数的委托包括两块数据：指向外围函数堆栈帧的指针（叫做 *帧指针*）和函数的地址。与此类似，结构/类的非静态成员函数委托由 *this* 指针和成员函数的地址组成。这两种形式的委托可以互相转换，实际上它们具有相同的类型：

```
struct Foo
{   int a = 7;
    int bar() { return a; }
}

int foo(int delegate() dg)
{
    return dg() + 1;
}

void test()
{
    int x = 27;
```

```
    int abc() { return x; }
    Foo f;
    int i;

    i = foo(&abc);      // i 被设置为 28
    i = foo(&f.bar);    // i 被设置为 8
}
```

环境和函数的结合被称作 *动态闭包*。

委托的 **.ptr** 特性会返回 *帧指针* 值，而类型为 `void*`。

委托的 **.funcptr** 属性会返回 *帧指针* 值，类型为函数类型。

未来的方向 函数指针和委托或能被合并成一种通用的语法并且彼此可以互相更改。

17.11.2 匿名函数和匿名委托

参看 函数字法。

17.12 main() 函数

对于控制台程序，`main()` 提供入口点。它会在所有的模块初始化，以及所有的单元测试都被运行以后被调用。在它返回以后，所有的模块析构函数会得到运行。`main()` 必须使用下列形式之一进行声明：

```
void main() { ... }
void main(char[][] args) { ... }
int main() { ... }
int main(char[][] args) { ... }
```

17.13 编译时函数执行

有些函数的子集可以在编译时执行。这个在常量合拢(constant folding)需要包括递归(recursion)和循环时很有用。为了在编译时可以被执行，此函数必须符号下面的标准(criteria)：

1. 函数形参都必须是：
    - 整数文字
    - 浮点文字
    - 字符文字
    - 字符串
    - 数组文字，即所有成员都是这个列表的条目
    - 关联数组文字，即所有成员都是这个列表的条目
    - 结构文字，即所有成员都是这个列表的条目
    - const 变量使用这个列表的一个成员进行初始化
2. 函数参数不能是参数可变型或者是 **lazy** 型

3. 函数不能被嵌套或同步(synchronized)
4. 函数不能是非静态(non-static)成员，即它不能有 `this` 指针
5. 在函数里的表达式不能：
    - 抛出异常
    - 使用指针、委托、非常量数组或者类
    - 引用任何全局状态(state)或变量
    - 引用任何局部变量
    - 进行 new 或 delete 操作
    - 调用任何在编译时不可执行的函数
6. 不允许下列语句类型：
    - synchronized 语句
    - throw 语句
    - with 语句
    - scope 语句
    - try-catch-finally 语句
    - 带标号的 break 和 continue 语句
7. 作为特殊情况，下列特性可以在编译时执行：

`.dup`
`.length`
`.keys`
`.values`

为了在编译时执行，函数必须出现在它必须那样执行的环境里，比如：

- 静态变量的初始化
- 静态数组的维
- 用于模板值参(value parameter)的形式参数

```
template eval( A... )
{
   const typeof(A[0]) eval = A[0];
}

int square(int i) { return i * i; }

void foo()
{
static j = square(3);      // 编译时
  writefln(j);
writefln(square(4));       // 运行时
  writefln(eval!(square(5))); // 编译时
}
```

在编译时执行函数会比在运行时的执行花费更长的时间。如果函数进行无限循环状态，那么它就会在编译时挂起(hang)（而不会在运行时挂起）。

在编译时执行的函数在下列场景下会给出跟运行时不同的结果：

- 浮点计算可能会以比运行时更高的精度进行
- 依赖于实现所定义的求值顺序
- 使用未初始化的变量

这些场景都跟不同优化设置影响会影响结果那样的场景一样。

17.13.1 字符串混入和编译时的函数执行

所有在编译时执行的函数都必须是在运行时可执行的。一个函数的编译时求值所完成的是跟在运行时运行函数一样的。即表示函数的语义不能依赖于函数的编译时得到的值。例如：

```
int foo(char[] s)
{
    return mixin(s);
}

const int x = foo("1");
```

它就是非法的，因为不能为 foo() 生成运行时代码。函数模板用于实现此类事情会是种比较合适的方法。

第 18 章 操作符重载

操作符重载就是重新解释作用于特定结构或者类的一元或二元操作符的含义，被重新解释的操作符是作为结构或类的成员实现的。不需要使用额外的语法。

18.1 一元操作符重载

可重载的一元操作符

op	*opfunc*
-e	opNeg
+e	opPos
~e	opCom
***e*	opStar
e++	opPostInc
e--	opPostDec
cast(*类型*)*e*	opCast

假设可被重载的操作符为 *op*，对应的类或结构成员函数名为 *opfunc*，语法：

```
op a
```

这里的 *a* 是一个类或结构对象引用，可以被解释成这下面所写的那样：

```
a.opfunc()
```

18.1.1 重载 ++*e* 和 --*e*

因为按照定义 ++*e* 在语义上等价于 (*e* += 1)，表达式 ++*e* 会被重写为 (*e* += 1)，然后按照这种形式应用操作符重载。--*e* 的情形与之类似。

18.1.2 示例

```
1. class A { int opNeg(); }
   A a;
   -a;      // 等价于 a.opNeg();

2. class A { int opNeg(int i); }
```

```
A a;
-a;     // 等价于 a.opNeg(), 这是个错误
```

重载 cast(*类型*)*e*

成员函数 *e*.**opCast()** 会被调用，并且 **opCast()** 的返回值被隐式地转换为 *类型*。因为不能基于返回值重载，所以每个结构或者类只能有一个 **opCast** 函数。重载转型操作符并不影响隐式转型，只能用于显式转型。

```
struct A
{
    int opCast() { return 28; }
}

void test()
{
    A a;

long i = cast(long)a;    // i 被设置为 28L
    void* p = cast(void*)a; // 错误，不能隐式地
                          // 从 int 转换成 void*
    int j = a;               // 错误，不能隐式地
                          // 从 A 转换到 int
}
```

18.2 二元操作符重载

可重载的二元操作符

op	可否交换？	*opfunc*	*opfunc_r*
+	是	opAdd	opAdd_r
-	否	opSub	opSub_r
*	是	opMul	opMul_r
/	否	opDiv	opDiv_r
%	否	opMod	opMod_r
&	是	opAnd	opAnd_r
|	是	opOr	opOr_r
^	是	opXor	opXor_r
<<	否	opShl	opShl_r
>>	否	opShr	opShr_r
>>>	否	opUShr	opUShr_r

~	否	opCat	opCat_r
==	是	opEquals	-
!=	是	opEquals	-
<	是	opCmp	-
<=	是	opCmp	-
>	是	opCmp	-
>=	是	opCmp	-
=	否	opAssign	-
+=	否	opAddAssign	-
-=	否	opSubAssign	-
*=	否	opMulAssign	-
/=	否	opDivAssign	-
%=	否	opModAssign	-
&=	否	opAndAssign	-
|=	否	opOrAssign	-
^=	否	opXorAssign	-
<<=	否	opShlAssign	-
>>=	否	opShrAssign	-
>>>=	否	opUShrAssign	-
~=	否	opCatAssign	-
in	否	opIn	opIn_r

给定一个二元可重载操作符 *op* 和它对应的名为 *opfunc* 和 *opfunc_r* 的类或结构成员函数，则下面的语法：

```
a op b
```

会按顺序应用下面的规则，来决定使用那种形式：

1. 表达式被重写为下面两种形式：

```
a.opfunc(b)
b.opfunc_r(a)
```

   如果 *a.opfunc* 或 *b.opfunc_r* 函数两者之一存在，就应用重载解析从中选出一个最合适的形式。如果函数存在，但是参数却不匹配，则被视为错误。
2. 如果操作符是可交换的，会尝试下面的形式：

```
a.opfunc_r(b)
b.opfunc(a)
```

3. 如果 *a* 或者 *b* 是结构或者类的引用，就是错误。

示例

```
1. class A { int opAdd(int i); }
   A a;
   a + 1;  // 等同于 a.opAdd(1)
   1 + a;  // 等同于 a.opAdd(1)

2. class B { int opDiv_r(int i); }
   B b;
   1 / b;  // 等同于 b.opDiv_r(1)

3. class A { int opAdd(int i); }
   class B { int opAdd_r(A a); }
   A a;
   B b;
   a + 1;  // 等同于 a.opAdd(1)
   a + b;  // 等同于 b.opAdd_r(a)
   b + a;  // 等同于 b.opAdd_r(a)

4. class A { int opAdd(B b);  int opAdd_r(B b); }
   class B { }
   A a;
   B b;
   a + b;  // 等同于 a.opAdd(b)
   b + a;  // 等同于 a.opAdd_r(b)

5. class A { int opAdd(B b);  int opAdd_r(B b); }
   class B { int opAdd_r(A a); }
   A a;
   B b;
   a + b;  // 歧义：是 a.opAdd(b) 还是 b.opAdd_r(a) ?
   b + a;  // 等同于 a.opAdd_r(b)
```

重载 == 和 !=

这两个操作符都使用 **opEquals**() 函数。表达式 (a == b) 被重写为 a.**opEquals**(b)，而 (a != b) 被重写为 !a.**opEquals**(b).

成员函数 **opEquals**() 是 Object 的一部分，定义为：

```
bool opEquals(Object o);
```

这样所有的类对象都有默认的 **opEquals**()。不过每一个要使用 == 或 != 的类定义应该要重写 opEquals。重写函数的参数必须是 Object 类型，而不能是该类的类型。

结构和联合（此后就称它作结构）可以提供成员函数：

```
bool opEquals(S s)
```

或者：

```
bool opEquals(S* s)
```

这里的 S 是结构名，用于定义如果确定相等性。

如果一个结构没有声明有函数 **opEquals**，那么就用两个结构的内容的位比较来确定是相等还是不等。

注意： 对于类对象的引用跟 **null** 比较操作应该是这样的：

```
if (a is null)
```

而不是这样：

```
if (a == null)
```

后面会被转换成：

```
if (a.opEquals(null))
```

如果 **opEquals**() 是一个虚函数，那么它就会失败。

18.2.1 重载 <, <=, > 和 >=

这些比较操作符都使用 **opEquals**() 函数。表达式 (a *op* b) 被重写为 (a.**opCmp**(b) *op* 0)。可交换的运算被重写为 (0 *op* b.**opCmp**(a))

成员函数 **opCmp**()是 Object 的一部分，定义为：

```
int opCmp(Object o);
```

这样所有的类对象都有默认的 **opCmp**()。

结构的 **opCmp** 操作跟结构的 **opEquals** 操作很类似：

```
struct Pair
{
    int a, b;
    int opCmp(Pair rhs)
    {
        if (a!=rhs.a) return a-rhs.a;
        return b-rhs.b;
    }
}
```

如果结构没有声明 **opCmp**() 函数，试图比较两个结构的大小就是一个错误。

18.2.1.1 基本原理

同时拥有 **opEquals** 和 **opCmp** 两个函数的原因为：

- 有时，测试相等性会比测试小于或大于要高效。
- 让 **opCmp** 定义在 Object，可以使得关联数组跟类一起工作。
- 对于某些对象来说，测试小于或者等于根本就没有意义。这就是为什么

Object.**opCmp** 会抛出一个运行时错误的原因。在每一个比较操作都很重要的类里，**opCmp** 必须被重写(overrid)。

对于类定义，**opEquals** 和 **opCmp** 的参数必须是 Object 类型，而不某个类的类型，目的是为了能够正确重写 Object.**opEquals** 和 Object.**opCmp** 函数。

18.3 函数调用操作符重载 *f*()

函数调用操作符，()，可以通过声明名为 **opCall** 的函数来重载：

```
struct F
{
    int opCall();
    int opCall(int x, int y, int z);
}

void test()
{    F f;
    int i;

    i = f();              // 等同于 i = f.opCall();
    i = f(3,4,5);         // 等同于 i = f.opCall(3,4,5);
}
```

采用这种方法，结构或对象可以表现得像函数一样。

18.4 数组操作符重载

18.4.1 重载索引操作符 *a*[*i*]

数组索引操作符，[]，可以通过声明名为 **opIndex** 的函数来重载，该函数可以有一个或多个参数。至于对数组赋值，可以通过声明名为 **opIndexAssign** 的函数来重载，该函数可以有两个或多个参数。第一个参数是赋值表达式的右值。

```
struct A
{
    int opIndex(size_t i1, size_t i2, size_t i3);
    int opIndexAssign(int value, size_t i1, size_t i2);
}

void test()
{    A a;
    int i;

    i = a[5,6,7];         // 等同于 i = a.opIndex(5,6,7);
    a[i,3] = 7;           // 等同于 a.opIndexAssign(7,i,3);
}
```

采用这种方法，结构或对象可以表现得像数组一样。

注意： To use array index overloading with the *op*=, ++, or -- operators, have the opIndex function return a reference type. This reference is then used as the lvalue for those operators.

18.4.2 重载分割操作符 *a*[] and *a*[*i* .. *j*]

重载切片操作符意味着重载例，如 `a[]` and `a[i .. j]` 这样的表达式。这个可通过声明一个名叫 **opSlice** 的函数来完成。给一个分片(slice)的赋值可以通过声明 **opSliceAssign** 来完成。

```
class A
{
    int opSlice();                                  // 重载 a[]
    int opSlice(size_t x, size_t y);                // 重载 a[i .. j]

    int opSliceAssign(int v);                       // 重载 a[] = v
    int opSliceAssign(int v, size_t x, size_t y); // overloads a[i .. j] = v
}

void test()
{   A a = new A();
    int i;
    int v;

    i = a[];            // 等同于 i = a.opSlice();
    i = a[3..4];        // 等同于 i = a.opSlice(3,4);

    a[] = v;            // 等同于 a.opSliceAssign(v);
    a[3..4] = v;        // 等同于 a.opSliceAssign(v,3,4);
}
```

18.5 重载赋值操作符

赋值操作符 = 可以被重载，如果左值是一个结构聚集，并且 `opAssign` 是该聚集的成员函数的话。

赋值操作符不能对那些可以被隐式转换成左值类型的右值进行重载。而且，不允许下列参数形式用于 `opAssign`：

```
opAssign(...)
opAssign(T)
opAssign(T, ...)
opAssign(T ...)
opAssign(T, U = defaultValue, etc.)
```

这里的 *T* 跟该聚集类型 *A* 相同，它会隐式转换成 *A*；或者如果 *A* 是一个结构，而 *T* 是一个可以隐式转换到 *A* 的类型指针。

18.6 转递(Forwarding)

让结构或类拥有成员函数 `opDot`，则可以实现转递(Forwarding)在结构作用域里不能找到的任意名字，一直可以转递到 `opDot` 函数的所返回的类型那里。换句话，即：

```
struct T {
```

```
  ...
  S opDot() { ... }
}
T t;
...
t.m
```

可以改写成这样：

```
t.opDot().m
```

如果 m 不是结构 T 的成员的话。

像 .sizeof、.init、.offsetof、.alignof、.mangleof 和 .stringof 这样一些成员是不能转递到 opDot 的。

```
struct S {
    int a, b, c;
}

struct T {
    S s;
    int b = 7;

    S* opDot() {
return &s;    // 转递到成员 s
    }
}

void main() {
    T t;
    t.a = 4;
    t.b = 5;
    t.c = 6;
    assert(t.a == 4);
assert(t.b == 5);    // T.b 重写了 S.b
    assert(t.c == 6);
    assert(t.s.b == 0);
assert(t.sizeof == 4*4); // 是 T 的 sizeof，而非 S 的 sizeof
}
```

18.7 未来的发展方向

操作符 !&& || ?: 以及少数其它的符号可能不再支持重载。

第 19 章 模板

> 我想我可以肯定没人理解了模板机制。 -- Richard Deyman

模板(Template)是 D 实现泛型编程(generic programming)的方法。模板通过 *模板声明* 进行定义：

模板声明:
    template *模板标识符* **(** *模板参数列表* **)** *约束*可选的
        { 多个声明定义 **}**

模板标识符:
    标识符

模板参数列表
    模板参数
    模板参数 **,** *模板参数列表*

模板参数:
    模板类型参数
    模板值参数
    模板别名参数
    模板元组参数
    模板 This 参数

无论模板是否被最终实例化，*模板声明* 的过程体在语法上必须是正确的。语义分析延迟到模板实例化时进行。模板有自己的作用域，模板过程体中可以包括类、结构、类型、枚举、变量、函数或者其它模板。

模板参数可以是类型、具体值、符号或者元组(Tuple)。其类型可以使用任何类型。值参数必须是整数型，在特化时它们必须被解析为整数常量。符号可以是任何非局部符号。元组(Tuple)是一种由 0 个或更多的类型、值或符号组成的序列。

模板参数特化将值或类型约束为 *模板参数* 可以接受的值或类型。

模板参数默认值用在不提供 *模板参数* 时。

19.1 显示模板实例化(Explicit Template Instantiation)

模板显示实例化的方式：

模板实例:
    模板标识符 **!(***模板参数列表* **)**

模板标识符:
    标识符

模板参数列表:
    模板参数
    模板参数 **,** *模板参数列表*

模板参数:
    类型

```
        赋值表达式
        符号
```

一旦被实例化，则位于该模板内的声明，也叫做模板成员(template member)，就处于该 *模板实例* 的作用域内：

```
template TFoo(T) { alias T* t; }
...
TFoo!(int).t x; // 声明 x 为 int* 类型
```

模板实例可以有别名：

```
template TFoo(T) { alias T* t; }
alias TFoo!(int) abc;
abc.t x;         // 声明 x 为 int* 类型
```

带有相同 *模板参数列表* 的 *模板声明* 的多重实例，在隐示转换之前，都会引用到相同的实例。例如：

```
template TFoo(T) { T f; }
alias TFoo!(int) a;
alias TFoo!(int) b;
...
a.f = 3;
assert(b.f == 3);        // a 和 b 引用相同的 TFoo 实例
```

即使 *模板实例* 完成于不同的模块中，这条规则也成立。

即使模板实参会被隐式转换成相同的模板形参类型，它们仍然会引用到不同的实例：

```
struct TFoo(int x) { }
static assert(is(TFoo!(3) == TFoo!(2 + 1)));    // 3 和 2+1 都是类型为 int 的 3
static assert(!is(TFoo!(3) == TFoo!(3u)));      // 3u 和 3 是不同的类型
```

如果声明了拥有多个相同 *模板标识符* 的模板，并且它们参数个数不同或者采用不同的特化，那么它们不同。

例如，一个简单的泛型复制函数可以是这个样子：

```
template TCopy(T)
{
    void copy(out T to, T from)
    {
        to = from;
    }
}
```

在使用模板之前，必须先使用具体的类型将其实例化：

```
int i;
TCopy!(int).copy(i, 3);
```

19.2 实例化作用域(Instantiation Scope)

模板实例 总是在声明 *板声明* 的作用域内执行，另外声明的模板参数被看作它们所推出的类型的别名。

例如：

module a

```
template TFoo(T) { void bar() { func(); } }
```

module b

```
import a;

void func() { }
alias TFoo!(int) f;      // 错误：func 没有在模块 a 内定义
```

和：

module a

```
template TFoo(T) { void bar() { func(1); } }
void func(double d) { }
```

module b

```
import a;

void func(int i) { }
alias TFoo!(int) f;
...
f.bar();         // 将调用 a.func(double)
```

模板参数 的特化和默认值将在 *模板声明* 的作用域内被求值。

19.3 参数推导(Argument Deduction)

采用比较对应的模板形参（template parameter）和模板实参（template argument）的方法，模板实例中的模板参数的类型被推倒出来。

对于每个模板参数，按照下面的顺序逐条应用规则直到每个参数的类型都被推倒出来：

1. 如果参数没有指定一个特化，参数的类型被设为指定的模板实参。
2. 如果类型特化依赖于一个类型参数，这个参数的类型就被设为与那个类型实参对应的类型。
3. 如果在检查了所有类型实参之后还有类型参数没有被分配类型，它们就会被分配给在 *模板参数列表* 中位于相同位置的模板实参。
4. 如果应用上述规则之后，还不能做到每个模板参数都精确的对应唯一一个类型，那么就被视为错误。

例如：

```
template TFoo(T) { }
alias TFoo!(int) Foo1;          // (1) T 被推导为 int
alias TFoo!(char*) Foo2;        // (1) T 被推导为 char*

template TBar(T : T*) { }
alias TBar!(char*) Foo3;        // (2) T 被推导为 char

template TAbc(D, U : D[]) { }
alias TAbc!(int, int[]) Bar1;   // (2) D 被推导为 int, U 被推导为 int[]
alias TAbc!(char, int[]) Bar2;  // (4) 错误，D 既是 char 又是 int

template TDef(D : E*, E) { }
alias TDef!(int*, int) Bar3;    // (1) E 是 int
                            // (3) D 是 int*
```

从一个特例化进行的推演可以为多个参数提供值：

```
template Foo(T: T[U], U)
{
    ...
}

Foo!(int[long])  // 实例化 Foo：将 T 设置成 int，而 U 设置成 long
```

当考虑匹配时，一个类被认为可以匹配任何父类或接口：

```
class A { }
class B : A { }

template TFoo(T : A) { }
alias TFoo!(B) Foo4;            // (3) T 是 B

template TBar(T : U*, U : A) { }
alias TBar!(B*, B) Foo5;        // (2) T 是 B*
                            // (3) U 是 B
```

19.4 模板类型参数(Template Type Parameters)

```
模板类型参数:
        标识符
        标识符 模板类型参数特化
        标识符 模板类型参数默认值
        标识符 模板类型参数特化 模板类型参数默认值

模板类型参数特化:
        : 类型

模板类型参数默认值:
        = 类型
```

19.4.1 特例化(Specialization)

模板可以通过在模板参数之后指定一个“:”和一个特化类型来将模板特化为使用某些指定的实参类型。例如：

```
template TFoo(T)         { ... } // #1
template TFoo(T : T[])   { ... } // #2
template TFoo(T : char)  { ... } // #3
template TFoo(T,U,V)     { ... } // #4

alias TFoo!(int) foo1;          // 实例化 #1
alias TFoo!(double[]) foo2;     // 实例化 #2 ，其中 T 为 double
alias TFoo!(char) foo3;         // 实例化 #3
alias TFoo!(char, int) fooe;    // 错误，实参个数不匹配
alias TFoo!(char, int, int) foo4; // 实例化 #4
```

当进行模板实例化时，会挑选匹配 *模板参数列表* 的特化度最高的模板。决定那个模板更为特化的方式同 C++ 处理偏序规则的方式相同。如果结果是模棱两可的，就是错误。

19.5 模板 This 参数(Template This Parameters)

模板 This 参数:
    this *模板类型参数*

模板 This 参数 常被用在成员函数模板里，用于提取 *this* 引用的类型。

```
import std.stdio;

struct S
{
    const void foo(this T)(int i)
    {
        writeln(typeid(T));
    }
}

void main()
{
    const(S) s;
    (&s).foo(1);
    S s2;
    s2.foo(2);
    invariant(S) s3;
    s3.foo(3);
}
```

输出

```
const(S)
S
invariant(S)
```

19.6 模板值参数(Template Value Parameters)

模板值参数:
单个声明
单个声明 模板值参数特化
单个声明 模板值参默认值
单个声明 模板值参数特化 模板值参默认值

模板值参数特化:
: *条件表达式*

模板值参默认值:
= __FILE__
= __LINE__
= *条件表达式*

__FILE__ 和 __LINE__ 会在实例化处扩展出源文件名和行号。

模板值参类型可以是任何在编译时可以被静态初始化的类型，并且值参也可以是任何在编译时可以被计算的表达式。这些类型包括整数、浮点数类型，以及字符串。

```
template foo(string s)
{
    string bar() { return s ~ " betty"; }
}

void main()
{
writefln("%s", foo!("hello").bar()); // 输出： hello betty
}
```

这个模板例子中，指定了一个为 10 的值参数：

```
template foo(U : int, int T : 10)
{
    U x = T;
}

void main()
{
    assert(foo!(int, 10).x == 10);
}
```

19.7 模板别名参数(Template Alias Parameters)

模板别名参数:
alias *标识符 模板别名参数特例化*$_{可选}$ *模板别名参数默认值*$_{可选}$

模板别名参数特例化:
: *类型*

```
模板别名参数默认值:
= 类型
```

别名参数使模板能够使用任何 D 符号来完成参数化，这些符号包括全局名称、局部名称、类型定义名称、模板名称、模板名以及模板实例名称。文字量也可以用作 alias 形参的实参。

- 全局名

```
int x;

template Foo(alias X)
{
    static int* p = &X;
}

void test()
{
    alias Foo!(x) bar;
    *bar.p = 3;          // 把 x 设置为 3
    static int y;
    alias Foo!(y) abc;
    *abc.p = 3;          // 把 y 设置为 3
}
```

- 类型名

```
class Foo
{
    static int p;
}

template Bar(alias T)
{
    alias T.p q;
}

void test()
{
    alias Bar!(Foo) bar;
    bar.q = 3;  // 把 Foo.p 设置为 3
}
```

- 模块名

```
import std.string;

template Foo(alias X)
{
       alias X.toString y;
}

void test()
{
    alias Foo!(std.string) bar;
    bar.y(3);   // 调用 std.string.toString(3)
}
```

- 模板名

```
int x;

template Foo(alias X)
{
    static int* p = &X;
}

template Bar(alias T)
{
    alias T!(x) abc;
}

void test()
{
    alias Bar!(Foo) bar;
    *bar.abc.p = 3;     // 把 x 设置为 3
}
```

- 模板别名

```
int x;

template Foo(alias X)
{
    static int* p = &X;
}

template Bar(alias T)
{
    alias T.p q;
}

void test()
{
    alias Foo!(x) foo;
    alias Bar!(foo) bar;
    *bar.q = 3;         // 把 x 设置为 3
}
```

- 文字量

```
template Foo(alias X, alias Y)
{
    static int i = X;
    static string s = Y;
}

void test()
{
    alias Foo!(3, "bar") foo;
writeln(foo.i, foo.s);  // 输出 3bar
}
```

19.8 模板元组参数(Template Tuple Parameters)

模板元组参数:
    标识符 ...

如果在 *模板参数列表* 里的最后一个模板参数被声明为 *模板元组参数*，那么它会跟任意的模板尾部参数进行匹配。参数的序列组成了一个 *元组(Tuple)*。一个 *元组(Tuple)* 不是一种类型、表达式或者符号。它是一种由类型、表达式或符号的任意混合的序列。

一个它的元素完全由类型组成的 *元组(Tuple)* 叫做 *类型元组(TypeTuple)*。一个它的元素完全由表达式组成的 *元组(Tuple)* 叫做 *表达式元组(ExpressionTuple)*。

一个 *元组(Tuple)* 可被用做参数列表并实例化别一个模板，或者被用作一个函数的参数列表。

```
template Print(A ...)
{
    void print()
    {
        writefln("args are ", A);
    }
}

template Write(A ...)
{
    void write(A a)      // A 是一个 类型元组
                       // a 是一个 表达式元组
    {
        writefln("args are ", a);
    }
}

void main()
{
    Print!(1,'a',6.8).print();                    // 输出结果：args 是 1a6.8
    Write!(int, char, double).write(1, 'a', 6.8); // prints:args 是 1a6.8
}
```

模板元组(template tuples)可以由一个隐式实例化的函数模板的尾部参数的类型进行推导：

```
template Foo(T, R...)
{
    void Foo(T t, R r)
    {
        writefln(t);
        static if (r.length)     // 如果有更多的实参
            Foo(r);              // 处理其余的实参
    }
}

void main()
{
    Foo(1, 'a', 6.8);
}
```

输出：

```
1
a
6.8
```

Tuple 可以由一个委托或函数的传递为函数形参的参数列表的类型进行推导：

```
import std.stdio;

/* R 是 return 类型
* A 是第一个实参类型
* U 是剩余形参类型的 类型元组
 */
R delegate(U) Curry(R, A, U...)(R delegate(A, U) dg, A arg)
{
    struct Foo
    {
        typeof(dg) dg_m;
        typeof(arg) arg_m;

        R bar(U u)
        {
            return dg_m(arg_m, u);
        }
    }

    Foo* f = new Foo;
    f.dg_m = dg;
    f.arg_m = arg;
    return &f.bar;
}

void main()
{
    int plus(int x, int y, int z)
    {
        return x + y + z;
    }

    auto plus_two = Curry(&plus, 2);
    writefln("%d", plus_two(6, 8));     // 输出 16
}
```

在一个 *元组(Tuple)* 里的元素数目可以使用 **.length** 特性获得。第 *n* 个元素可以通过使用 [*n*] 来索引该 *元组(Tuple)* 得到，而对于子元组(sub tuples)则可以通过分割语法(slicing syntax)来创建。

元组(Tuple) 是静态的编译时实体，因此没有办法来动态更改、添加或删除元素。

如果一个带有元组(Tuple)参数的模板跟一个不带元组(Tuple)参数的模板都完全匹配于某一个模板实例，则不带有 *模板元组参数* 的模板会选择。

19.9 模板参数默认值

尾端（最右边）的模板参数可以被给定默认值：

```
template Foo(T, U = int) { ... }
Foo!(uint,long); // 实例化 Foo: T 替换为 uint , U 替换为 long
Foo!(uint);      // 实例化 Foo: T 替换为 uint , U 替换为 int

template Foo(T, U = T*) { ... }
Foo!(uint);      // 实例化 Foo: T 替换为 uint , U 替换为 uint*
```

19.10 隐式模板特性

如果模板有且只有一个成员，并且这个成员和模板同名的话，这个成员就被认为引用的是一个模板实例：

```
template Foo(T)
{
    T Foo;       // 声明变量 Foo 为类型 T
}

void test()
{
    Foo!(int) = 6;       // 代替 Foo!(int).Foo
}
```

19.11 类模板(Class Template)

类模板声明:
class *标识符* **(** *模板参数列表* **)** [*父类* {, *接口类* }] *类过程体*

如果一个模板声明且仅声明了一个成员，并且那个成员是一个同模板同名的类：

```
template Bar(T)
{
    class Bar
    {
        T member;
    }
}
```

则同下面的声明语义等价，称作 *类模板声明*：

```
class Bar(T)
{
    T member;
}
```

19.12 结构、联合以及接口模板

跟类模板类似，结构(struct)、联合(union)以及接口(interface)都可以通过提供一个参数列表来传递进模板。

19.13 函数模板(Function Template)

如果一个模板声明且仅声明了一个成员，并且那个成员是一个同模板同名的函数：

函数模板声明:
类型 标识符 **(** *模板参数列表* **)** **(** *函数参数列表* **)** *函数体*

一个用于计算类型为 *T* 的平方的函数模板就是：

```
T Square(T)(T t)
{
    return t * t;
}
```

函数模板可以通过使用 !(*TemplateArgumentList*) 进行显式的实例化：

```
writefln("The square of %s is %s", 3, Square!(int)(3));
```

或者进行隐式实例化，而此时的 *模板参数列表* 则由函数形参的类型进行推导：

```
writefln("The square of %s is %s", 3, Square(3));  // T 被推导为 int
```

那些要被隐式推导的函数模板类型参数不可能进行特例化：

```
void Foo(T : T*)(T t) { ... }

int x,y;
Foo!(int*)(x);    // 正确，T 不是推导自函数形参
Foo(&y);          // 错误，T 进行了特例化
```

不能被隐式推导的模板形参可以有默认值：

```
void Foo(T, U=T*)(T t) { U p; ... }

int x;
Foo(&x);     // T 是 int，U 是 int*
```

函数模板可以有自己的返回类型——它推导自此函数的第一个 *return 语句*：

```
auto Square(T)(T t)
{
    return t * t;
}
```

如果有不只一个的 return 语句，那么这些 return 语句表达式的类型必须要匹配。如果没有 return 语句，那么该函数模板的返回类型就为 `void`。

19.14 递归模板(Recursive Templates)

可以组合模板的各种特性来产生一些有趣的效果，例如在编译时对非平凡函数求值。例如，可以写一个计算阶乘的模板：

```
template factorial(int n : 1)
{
    enum { factorial = 1 }
}

template factorial(int n)
{
    enum { factorial = n* factorial!(n-1) }
}

void test()
{
    writefln("%s", factorial!(4));      // 输出 24
}
```

19.15 模板约束(Template Constraint)

约束:
if (***约束表达式* **)

约束表达式:
表达式

约束(Constraint) 被用来往一个模板增加额外的关于匹配参数的约束，要求它们不在 *模板参数列表* 可能的范围内。*约束表达式* 在编译时进行计算，并返回一个被转换成布尔值的结果。如果该结果为真，则该模板匹配；否则该模板不匹配。

例如，下列函数模板仅在 N 为奇数值匹配：

```
void foo(int N)()
       if (N & 1)
{
    ...
}
...
foo!(3)();  // 正确，匹配
foo!(4)();  // 错误，不匹配
```

19.16 限制(Limitations)

模板不能用来往类里添加非静态成员或虚函数。例如：

```
class Foo
{
    template TBar(T)
    {
        T xx;                   // 变成为 Foo 的静态成员
        int func(T) { ...}      // 非虚的

        static T yy;                            // 正确
        static int func(T t, int y) { ...}      // 正确
    }
}
```

不能在函数内部声明模板。

模板不能添加函数到接口里：

```
interface TestInterface { void tpl(T)(); }   // 错误
```

第 20 章 模板混入

一个 *模板混入* 指的是从一个 *模版声明* 的过程体内提取一个任意的声明集合，并将它们插入到当前的上下文中。

```
模板混入:
        mixin 模板标识符 ;
        mixin 模板标识符 混入标识符 ;
        mixin 模板标识符 !(模板参数列表 ) ;
        mixin 模板标识符 !(模板参数列表 ) 混入标识符 ;

混入标识符:
        标识符
```

模版混入 可以出现在模块、类、结构、联合的声明列表中，并且可以做为语句。*模板标识符* 是一个 *模版声明*。如果 *模版声明* 没有参数，就可以使用不带 !(*模版参数列表*) 的混入形式。

不像模板具现化，模板混入的过程体在混入所在的作用域内计算，而不是在定义模板声明的地方。这类似于使用剪切和粘贴将模版的过程体插入混入的位置。这对注入参数化的‘样板文件’是有用的，同时对创建模板化嵌套函数也很有用，而正常情况下是不可能具现化嵌套函数的。

```
template Foo()
{
    int x = 5;
}

mixin Foo;

struct Bar
{
    mixin Foo;
}

void test()
{
    writefln("x = %d", x);              // 输出 5
    {   Bar b;
        int x = 3;

        writefln("b.x = %d", b.x);      // 输出 5
        writefln("x = %d", x);          // 输出 3
        {
            mixin Foo;
            writefln("x = %d", x);      // 输出 5
            x = 4;
            writefln("x = %d", x);      // 输出 4
        }
        writefln("x = %d", x);          // 输出 3
    }
    writefln("x = %d", x);              // 输出 5
}
```

混入可以被参数化：

```
template Foo(T)
{
    T x = 5;
}

mixin Foo!(int);                    // 创建类型为 int 的 x
```

混入可以可以为类添加虚函数：

```
template Foo()
{
    void func() { writefln("Foo.func()"); }
}

class Bar
{
    mixin Foo;
}

class Code : Bar
{
    void func() { writefln("Code.func()"); }
}

void test()
{
    Bar b = new Bar();
    b.func();            // 调用 Foo.func()

    b = new Code();
    b.func();            // 调用 Code.func()
}
```

混入在它们出现的地方被求值，而不是在模板声明的地方：

```
int y = 3;

template Foo()
{
    int abc() { return y; }
}

void test()
{
    int y = 8;
mixin Foo;       // 使用的是局部的 y ，而不是全局的 y
    assert(abc() == 8);
}
```

混入可以使用别名参数来参数化符号：

```
template Foo(alias b)
```

```
{
    int abc() { return b; }
}

void test()
{
    int y = 8;
    mixin Foo!(y);
    assert(abc() == 8);
}
```

这个例子使用了一个混入来为任意语句实现一个泛型 Duff's Device（在这里，那个语句采用粗体表示）。在生成一个嵌套函数的同时也生成了一个委托文字量，他们会通过编译器内联：

```
template duffs_device(alias id1, alias id2, alias s)
{
    void duff_loop()
    {
        if (id1 < id2)
        {
            typeof(id1) n = (id2 - id1 + 7) / 8;
            switch ((id2 - id1) % 8)
            {
                case 0:         do {  s();
                case 7:               s();
                case 6:               s();
                case 5:               s();
                case 4:               s();
                case 3:               s();
                case 2:               s();
                case 1:               s();
                                } while (--n > 0);
            }
        }
    }
}

void foo() { writefln("foo"); }

void test()
{
    int i = 1;
    int j = 11;

    mixin duffs_device!(i, j, delegate { foo(); } );
    duff_loop();         // 执行 foo() 10 次
}
```

20.1 混入作用域

混入中的声明被‘导入’到周围的作用域中。如果混入和其周围的作用域中有相同的名字，周围的作用域中的声明将覆盖混入中的那个声明：

```
int x = 3;

template Foo()
{
    int x = 5;
    int y = 5;
}

mixin Foo;
int y = 3;

void test()
{
    writefln("x = %d", x);       // 输出 3
    writefln("y = %d", y);       // 输出 3
}
```

如果两个不同的混入被放入同一个作用域，并且他们中定义了同名的声明，就会出现模棱两可的错误：

```
template Foo()
{
    int x = 5;
    void func(int x) { }
}

template Bar()
{
    int x = 4;
    void func() { }
}

mixin Foo;
mixin Bar;

void test()
{
    writefln("x = %d", x);       // 错误，x 有歧义
    func();              // 错误，func 有歧义
}
```

对 **func()** 的调用有歧义，因为 Foo.func 和 Bar.func 在不同的作用域里。

如果一个混入有 *混入标识符*，它就可以用来消除歧义：

```
int x = 6;

template Foo()
{
    int x = 5;
    int y = 7;
    void func() { }
}
```

```
template Bar()
{
    int x = 4;
    void func() { }
}

mixin Foo F;
mixin Bar B;

void test()
{
    writefln("y = %d", y);       // 输出 7
    writefln("x = %d", x);       // 输出 6
    writefln("F.x = %d", F.x);   // 输出 5
    writefln("B.x = %d", B.x);   // 输出 4
    F.func();                    // 调用 Foo.func
    B.func();                    // 调用 Bar.func
}
```

别名声明可以被用来将声明在不同混入里的函数重载在一起：

```
template Foo()
{
    void func(int x) {  }
}

template Bar()
{
    void func() {  }
}

mixin Foo!() F;
mixin Bar!() B;

alias F.func func;
alias B.func func;

void main()
{
    func();     // 调用 B.func
    func(1);    // 调用 F.func
}
```

一个混入有它自己的作用域，即使一个声明被另一个封闭的声明所重写也一样：

```
int x = 4;

template Foo()
{
    int x = 5;
    int bar() { return x; }
}

mixin Foo;

void test()
{
    writefln("x = %d", x);              // 输出 4
    writefln("bar() = %d", bar());      // 输出 5
}
```

```
}
```

第 21 章 契约编程

契约(Contract)是一项用于减少大型项目成本的突破性技术。契约由先验条件(precondition)、后验条件(postcondition)、错误和不变量(invariant)等概念组成。契约也可以添加到 C++ 里，且无需对语言加以改造，但结果是笨拙且不一致。
在语言内部支持契约的目的是：

1. 让契约有一个一致的外在形式
2. 提供工具支持
3. 使编译器能够根据从契约中收集的信息生成更好的代码
4. 易于管理并强制实行契约
5. 处理契约继承

契约的思想很简单——仅要求表达式的值必须为真。如若不然，契约就被违反，那么按照定义，程序中就一定有错漏。契约构成了程序规格说明的一部分，只不过是从文档中挪到了代码中。就像每个程序员所知道的那样，文档通常是不完整的、过时的、错误的或者是不存在的。将契约用到代码中，使它对该程序也具有了可验证性。

21.1 断言契约(Assert Contract)

断言(assert)最基本的契约。断言(assert)在代码内插入一个可检查的表达式，该表达式在正常的情况下一定是真值：

```
assert(expression);
```

C 程序员会有似曾相识的感觉。但是与 C 不同的是，函数体内的 `assert` 会抛出 `AssertError`，这个异常可以被捕获并处理。在该段代码必须要允许其他代码的恣意使用时，捕获契约违例就变得非常有用；当它要求必须必须是失败可证时，它同样也是有力的调试工具。

21.2 先验和后验契约(Pre and Post Contracts)

先验契约用于指定一条语句在执行前的先验条件。最典型的用法可能要数函数参数的有效性验证了。后验契约用于检验语句的结果。最典型的用法要数验证函数返回值得合法性以及它的任何副作用。语法如下：

```
in
{
```

```
...契约先验条件...
}
out (result)
{
...契约后验条件...
}
body
{
...代码...
}
```

按照定义，如果契约先验条件被违反，则过程体（body）将受到错误的参数。这将抛出一个 AssertError 异常。如果契约后验条件被违反，则意味着过程体中有一个错漏。这将抛出一个 AssertError 异常。

in 或 out 子句都可以被省略。如果过程体带有 out 子句，则变量 result 将被声明，同时被赋予该函数的返回值。例如，我们实现一个求平方根的函数：

```
long square_root(long x)
   in
   {
      assert(x >= 0);
   }
   out (result)
   {
      assert((result * result) <= x && (result+1) * (result+1) >= x);
   }
   body
   {
      return cast(long)std.math.sqrt(cast(real)x);
   }
```

in 和 out 中的断言叫做 契约。其中可以出现任何 D 的语句或者表达式，但是必须要保证这些语句没有副作用，并且最终发行版中的代码不依赖于 这些代码的作用。在构建发行版的程序时，这些代码将不会包括在其中。

如果函数返回 void，即没有结果，那么 out 子句中自然也就不用声明 result。在这种情况下，使用：

```
void func()
  out
  {
      ...契约...
  }
  body
  {
      ...
  }
```

在 out 语句中，*result* 被初始化并设为函数的返回值。

21.3 In，Out 和继承

如果派生类的函数重写了父类中的一个函数，那么它只须满足基类函数的一条 `in` 契约。重写函数然后就变成了一个 *放宽* `in` 契约的过程。

一个不带 `in` 契约的函数表示允许函数参数是任何值。即暗示：如果在继承层次里的任何函数没有 `in` 契约，那么关于重写它的函数的 `in` 契约就没有有用的效果。

反过来，所有的 `out` 契约都必须满足，所以重写函数 *收紧* 了 `out` 契约。

21.4 类不变量(Class Invariants)

类不变量(class invariants)用来指定类必须总是为真的这一特质（除了在执行成员函数时）。在 类 一节有关于它们的介绍。

21.5 参考

契约推荐书目
让 Java 支持契约

第 22 章 条件编译

条件编译 指的就是选择哪些代码需要编译、哪些代码不需要编译的处理过程。（在 C 和 C++ 中，条件编译的实现方法是使用预处理块：`#if/#else/#endif`。）

```
条件声明:
    条件
    条件 else 条件
    条件 : 声明

条件编译声明块:
    单个声明
    { 多个声明 }
    { }

多个声明:
    单个声明
    单个声明 多个声明

条件语句:
    条件 无域非空语句
    条件 无域非空语句 else 无域非空语句
```

如果 *条件* 被满足，那么紧跟的 *条件编译声明块* 或 *语句* 就会被编译。 如果不满足，则可选的 `else` 之后的 *条件编译声明块* 或 *语句* 就会被编译。

任何 *条件编译声明块* 或 *语句* 只有语法正确才能被编译。

不会有新的域被引入，即使 *条件编译声明块* 或 *语句* 被 { } 所封闭。

条件声明 和 *条件语句* 可以嵌套使用。

StaticAssert 可以用来处理在编译有错误条件编译分支时出现的错误。

条件 有以下几种形式：

```
条件:
    Version 条件
    Debug 条件
    StaticIf 条件
```

22.1 Version 条件

版本条件可以实现在一个源文件中放置多个版本的模块。

```
Version 条件:
        version ( 整数 )
        version ( 标识符 )
```

Version 条件 满足条件是 *整数* 大小或等于当前的 *Version 级别*，或者是 *标识符* 跟 *Version 标识符* 相匹配。

Version 级别 和 *Version 标识符* 可以在命令行通过打开 **-version** 开关来设置，或者在模块

中使用 *Version 指定* 来进行设置，当然他们也可以由编译器事先定义的。

版本标识符有自己独特的名字空间，他们不会跟模块中的调试器标识符或其它的符号相冲突。在某个模块中定义的版本标识符并不会对其它导入的模块产生影响。

```
int k;
version (Demo)  // 编译 demo 版本的代码块
{   int i;
    int k;       // 错误，k 已经被定义

    i = 3;
}
x = i;           // 使用前面声明的 i

version (X86)
{
... // 实现自定义的内嵌汇编版本
}
else
{
... // 使用默认但很慢的版本
}
```

22.1.1 Version 指定

```
Version指定
version = 标识符 ;
version = 整数 ;
```

“版本指定(version specification)”使得组织一组功能到一个主版本下变得更为直接，例如：

```
version (ProfessionalEdition)
{
    version = FeatureA;
    version = FeatureB;
    version = FeatureC;
}
version (HomeEdition)
{
    version = FeatureA;
}
...
version (FeatureB)
{
... 实现功能 B ...
}
```

Version 标识符或级别不能被向前引用：

```
version (Foo)
{
    int x;
```

```
}
version = Foo;  // 错误，Foo 已经被使用了
```

Version 指定 只能出现在模块作用域里。

尽管“调试条件”和“版本条件”在表面上具有同样的效果，但它们的目的是完全不的。“调试条件”用于添加在制作发行版本时可以移除的调试代码。版本语句为的是辅助移植和多发行版本。

这里的示例所展示的就是 *完整* 版本和 *演示* 版本的联合：

```
class Foo
{
    int a, b;

    version(full)
    {
        int extrafunctionality()
        {
            ...
            return 1;           // 支持额外的功能
        }
    }
else // 演示
    {
        int extrafunctionality()
        {
            return 0;           // 不支持额外的功能
        }
    }
}
```

建立各种不同版本的方法就是通过传递参数给 version：

```
version(n) // 如果 version 级别 >= n，那么就加入此版本代码
{
... 版本 代码 ...
}

version(标识符) // 添加版本代码，如果
                    // version 关键字为该 标识符 的话
{
... 版本 代码 ...
}
```

在命令行可能的设置是：`-version=`*`n`* 和 `-version=`*`identifier`*.

22.1.2 预定义的版本

有几个环境版本标识符和标识符名字空间被事先定义成统一的格式。版本标识符并不会跟代码中的其它标识符冲突，因为它们处于不同的名字空间。预定义的版本标识符是全局的，即它们对所有正在编译和导入的模块都有效。

预定义的版本标识符

版本标识符	描述
DigitalMars	Digital Mars 是编译器提供商
X86	Intel 和 AMD 32 位处理器
X86_64	Intel 和 AMD 64 位处理器
Windows	Microsoft Windows 系统
Win32	Microsoft 32 位 Windows 系统
Win64	Microsoft 64 位 Windows 系统
linux	所有 linux 系统
Posix	所有 linux 系统
LittleEndian	字节序，低位优先
BigEndian	字节序，高位优先
D_Coverage	生成代码覆盖分析 指令（命令行开关 **-cov**）
D_Ddoc	Ddoc documentation (command line switch **-D**) is being generated
D_InlineAsm_X86	X86 的内嵌汇编已实现
D_InlineAsm_X86_64	Inline assembler for X86-64 is implemented
D_LP64	Pointers are 64 bits (command line switch **-m64**)
D_PIC	生成位置无关代码（命令行开关 **-fPIC**）
unittest	启用单元测试（命令行开关 **-unittest**）
D_Version2	它是 D 的第二版编译器
none	从未定义过的；用于仅禁用一部分代码
all	总是定义过；使用情形跟 **none** 相反

如果新的实现出现了，或是那些标识的确很有意义，则也会被加入。

D 语言无可避免地会随着时间演变。因此，以“D_”打头的版本标志符构成的名字空间被保留作为 D 语言规范或者新特征的标志的名字空间。

另外，上面所列出的这些预定义版本标识符不能设置在命令行或版本语句中。（这样可以避免像 **Windows** 和 **linux** 两都同时被设置这样的情况发生）

特定编译器提供商的版本号可以是被预定义，要求是前缀加上提供商商标的标志符，如这样：

```
version(DigitalMars_funky_extension)
{
    ...
```

```
}
```

一定要把正确的版本号用于正确的目的。例如，在使用特定于提供商的特征时使用该提供商的标识符；在使用特定于操作系统的特征时使用该操作系统的标志符，等等。

22.2 Debug 条件

通常开发人员会构建两种程序，一个发行版，一个调试版。调试版通常包括额外的错误检测代码、测试套件、友好的输出代码等。调试语句的语句体采用条件编译。这是在 D 语言中实现 C 语言中的 `#ifdef DEBUG` / `#endif` 的一种方式。

```
Debug 条件:
    debug
debug ( 整数 )
debug ( 标识符 )
```

debug 条件满足的情形是在编译器参数中使用了 **-debug** 开关。

debug (*整数* **)** 条件满足的情形是 调试级别 >= 该*整数*。

debug (*标识符* **)** 条件满足的情形是 调试标识符跟该 *标识符* 匹配。

```
class Foo
{
       int a, b;
   debug:
       int flag;
}
```

22.2.1 Debug 指定

```
Debug 指定
debug = 标识符 ;
debug = 整数 ;
```

“调试标识符”和“调试级别”可以通过命令行开关 **-debug** 进行设置，也可以使用 *调试指定*。

“调试指定”仅对它所在的模块有效，并不会影响任何导入的模块。“调试标识符”有它们自己的名字空间，跟“版本标识符”和其它的符号无关。

向前引用一个“调试指定”是不合法的：

```
debug (foo) writefln("Foo");
debug = foo;     // 错误，foo 在设置之前已被使用
```

Debug 指定 只能出现在模块作用域里。

建立各种不同调试版的方法就是通过传递参数给 debug：

```
debug(整数) { }     // 如果 debug 级别 >= 该整数，那么就加进调试代码
debug(标识符) { }     // 如果 debug 关键字为该 标识符，那么就加进调试代码
```

在命令行可能的设置是：-debug=*n* 和 -debug=*identifier*.

22.3 Static If 条件

StaticIf 条件:
static if (*赋值表达式* **)**

赋值表达式 会被隐式地转换成布尔类型，并且在编译时就会被计算。如果计算值为 **true**，那么条件得到满足。如果计算值为 **false**，那么条件不会得到满足。

如果 *赋值表达式* 没有被隐式地转换成布尔类型或在编译时没有被计算，那么就会出错。

StaticIf 条件 可以出现的地方有：模块、类、模板、结构、联合或者函数域。在函数域里，在 *赋值表达式* 中所引用的符号只要是可以被在该点的表达式正常引用的都行。

```
const int i = 3;
int j = 4;

static if (i == 3)       // 正确，在模板域里
   int x;

class C
{   const int k = 5;

   static if (i == 3)  // 正确
      int x;
   else
      long x;

   static if (j == 3)  // 错误，j 不是一个常量
      int y;

   static if (k == 5)  // 正确，k 在当前域里
      int z;
}

template INT(int i)
{
   static if (i == 32)
      alias int INT;
   else static if (i == 16)
      alias short INT;
   else
      static assert(0);       // 不支持
}

INT!(32) a;     // a 是一个 int
INT!(16) b;     // b 是一个 short
INT!(17) c;     // 错误，静态断言错误
```

StaticIf 条件 的条件跟 *IfStatement* 的条件是不同的，区别在于下面几个方面：

1. 它可以用于有条件地把编译各种声明，而不仅仅局限于语句。
2. 它不会引用新的作用域，即使在要被条件编译的语句中使用了符号 **{}**。
3. 对于那些不满足条件的，里面那些要被条件编译的代码仅需要语法正确就行。而不一定要求语义上也正确。
4. 它必须在编译时可计算。

22.4 静态断言(Static Assert)

静态断言：

static assert (*赋值表达式* **);**
static assert (*赋值表达式* **,** *赋值表达式* **);**

赋值表达式 在编译时被计算，并被转换成布尔值。如果值为 true，则静态断言就被忽略。如果值为 false，则会出现一个错误诊断，然后编译失败。

不像 *断言表达式*，*静态断言* 总是被编译器检查和计算，除非它们所处的位置条件不满足。

```
void foo()
{
    if (0)
    {
        assert(0);         // 绝不出错
        static assert(0); // 总是出错
    }
    version (BAR)
    {
    }
    else
    {
        static assert(0); // 当没有定义版本 BAR 时出错
    }
}
```

StaticAssert 在想提醒注意代码中所不支持的条件编译时相应有用。

第二个可选项 *赋值表达式* 可以用于提供额外的信息，如文本字符串——它位会被随同错误诊断一起被输出。

第 23 章 特征

traits（特征）是语言扩展，它可以使得程序在编译时获得编译器内部的信息。即众所周知的编译时反射(reflection)。它的实现使用了特定、易于扩展的语法（类似于 Pragmas），以便可以根据需要添加新的能力。

```
Traits 表达式:
__traits ( Traits 关键字 , 多个 Traits 参数 )

Traits 关键字:
    isAbstractClass
    isArithmetic
    isAssociativeArray
    isFinalClass
    isFloating
    isIntegral
    isScalar
    isStaticArray
    isUnsigned
    isVirtualFunction
    isAbstractFunction
    isFinalFunction
    hasMember
    getMember
    getVirtualFunctions
    classInstanceSize
    allMembers
    derivedMembers
    isSame
编译

多个 Traits 参数:
单个 Traits 参数
单个 Traits 参数 , 单个 Traits 参数

单个 Traits 参数:
赋值表达式
类型
```

23.1 isArithmetic

如果参数都是数学类型，或数学类型的表达式，那么就返回 **true**。否则，返回 **false**。如果没有参数，则返回 **false**。

```
import std.stdio;

void main()
{
    int i;
    writefln(__traits(isArithmetic, int));
    writefln(__traits(isArithmetic, i, i+1, int));
```

```
    writefln(__traits(isArithmetic));
    writefln(__traits(isArithmetic, int*));
}
```

输出

```
true
true
false
false
```

23.2 isFloating

跟 **isArithmetic** 基本一样， 只是它是用于浮点类型的（包括虚数和复数类型）。

23.3 isIntegral

跟 **isArithmetic** 基本一样， 只是它是用于整数类型的（包括字符类型）。

23.4 isScalar

跟 **isArithmetic** 基本一样， 只是它是用于数量(scalar)类型的。

23.5 isUnsigned

跟 **isArithmetic** 基本一样， 只是它是用于无符号(unsigned)类型的。

23.6 isStaticArray

跟 **isArithmetic** 基本一样， 只是它是用于静态数组类型的。

23.7 isAssociativeArray

跟 **isArithmetic** 基本一样， 只是它是用于关联数组类型的。

23.8 isAbstractClass

如果参数都是抽象类，或抽象类表达式，那么就返回 **true**。否则，返回 **false**。如果没有参数，则返回 **false**。

```
import std.stdio;

abstract class C { int foo(); }

void main()
{
    C c;
```

```
    writefln(__traits(isAbstractClass, C));
    writefln(__traits(isAbstractClass, c, C));
    writefln(__traits(isAbstractClass));
    writefln(__traits(isAbstractClass, int*));
}
```

输出

```
true
true
false
false
```

23.9 isFinalClass

跟 **isAbstractClass** 基本一样， 只是它是用于最终类(final classes)的。

23.10 isVirtualFunction

接受一个参数。如果该参数是一个虚函数，则返回 **true**；否则返回 **false**。

```
import std.stdio;

struct S
{
  void bar() { }
}

class C
{
  void bar() { }
}

void main()
{
    writefln(__traits(isVirtualFunction, C.bar));  // true
    writefln(__traits(isVirtualFunction, S.bar));  // false
}
```

23.11 isAbstractFunction

接受一个参数。如果该参数是一个抽象函数，则返回 **true**；否则返回 **false**。

```
import std.stdio;

struct S
{
  void bar() { }
}

class C
{
  void bar() { }
}

class AC
```

```
{
  abstract void foo();
}

void main()
{
    writefln(__traits(isAbstractFunction, C.bar));   // false
    writefln(__traits(isAbstractFunction, S.bar));   // false
    writefln(__traits(isAbstractFunction, AC.foo));  // true
}
```

23.12 isFinalFunction

接受一个参数。如果该参数是一个最终函数(final function)，则返回 **true**；否则返回 **false**。

```
import std.stdio;

struct S
{
  void bar() { }
}

class C
{
  void bar() { }
  final void foo();
}

final class FC
{
  void foo();
}

void main()
{
    writefln(__traits(isFinalFunction, C.bar));   // false
    writefln(__traits(isFinalFunction, S.bar));   // false
    writefln(__traits(isFinalFunction, C.foo));   // true
    writefln(__traits(isFinalFunction, FC.foo));  // true
}
```

23.13 hasMember

第一个参数是一个带有成员的类型，或者是其类型为带有成员的一个表达式。第二个参数是一个字符串。如果字符串是该类型的有效特性，出返回 **true**；否则返回 **false**。

```
import std.stdio;

struct S
{
    int m;
```

```
}

void main()
{   S s;

   writefln(__traits(hasMember, S, "m")); // true
   writefln(__traits(hasMember, s, "m")); // true
   writefln(__traits(hasMember, S, "y")); // false
   writefln(__traits(hasMember, int, "sizeof")); // true
}
```

23.14 getMember

接受两个参数，第二个必须是一个字符串。结果为一个表达式，其形式为：第一个参数，紧接一个‘.’，然后是第二个参数；此种形式组成一个标识符。

```
import std.stdio;

struct S
{
   int mx;
   static int my;
}

void main()
{ S s;

__traits(getMember, s, "mx") = 1;  // 等同于 s.mx=1;
  writefln(__traits(getMember, s, "m" ~ "x")); // 1

__traits(getMember, S, "mx") = 1;  // 错误，对于 S.mx 没有 this
  __traits(getMember, S, "my") = 2;  // 正确
}
```

23.15 getVirtualFunctions

第一个参数是一个类类型或类类型的表达式。第二个参数为一个字符串，它跟该类的某个函数名匹配。结果是该函数的虚重载数组。

```
import std.stdio;

class D
{
   this() { }
   ~this() { }
   void foo() { }
   int foo(int) { return 2; }
}

void main()
{
   D d = new D();

   foreach (t; __traits(getVirtualFunctions, D, "foo"))
      writefln(typeid(typeof(t)));
```

```
    alias typeof(__traits(getVirtualFunctions, D, "foo")) b;
    foreach (t; b)
        writefln(typeid(t));

    auto i = __traits(getVirtualFunctions, d, "foo")[1](1);
    writefln(i);
}
```

输出

```
void()
int()
void()
int()
2
```

23.16 classInstanceSize

接受一个单一参数，其求值后或者为类类型，或者为类类型的表达式。结果类型为 size_t，其值为在该类类型的运行实例里字节的数目。它基于一个类的静态类型，而非多态类型。

23.17 allMembers

接受一个单一参数，其求值后或者为类型，或者为类型表达式。会返回字符串字法数组，每一个都是该类型的成员名；而该类型由基类（如果该类是一个类型）的所有成员构成：没有重复名。不包括内建特性。

```
import std.stdio;

class D
{
    this() { }
    ~this() { }
    void foo() { }
    int foo(int) { return 0; }
}

void main()
{
    auto a = __traits(allMembers, D);
    writefln(a);
    // [_ctor,_dtor,foo,print,toString,toHash,opCmp,opEquals]
}
```

出现在结果里的字符串的顺序是未定义的。

23.18 derivedMembers

接受一个单一参数，其求值后或者为类型，或者为类型表达式。返回一个字符串文字数组，当中的每一个元素是该类型的成员名。没有重复名。不包括基类成员名字。不包括内建特性。

```
import std.stdio;

class D
{
    this() { }
    ~this() { }
    void foo() { }
    int foo(int) { return 0; }
}

void main()
{
    auto a = __traits(derivedMembers, D);
    writefln(a);        // [_ctor,_dtor,foo]
}
```

出现在结果里的字符串的顺序是未定义的。

23.19 isSame

接受两个参数，如果它们是相同的符号则返回布尔值 **true**；否则返回 **false**。

```
import std.stdio;

struct S { }

int foo();
int bar();

void main()
{
    writefln(__traits(isSame, foo, foo)); // true
    writefln(__traits(isSame, foo, bar)); // false
    writefln(__traits(isSame, foo, S));   // false
    writefln(__traits(isSame, S, S));     // true
    writefln(__traits(isSame, std, S));   // false
    writefln(__traits(isSame, std, std)); // true
}
```

如果两个参数是由可以求出相同值的文字或枚举组成的表达式，则返回 true。

23.20 编译

如果所参数可编译（语义正确），则返回布尔值 **true**。参数可以是符号、类型或语义正确的表达式。参数不能是语句或声明。

如果没有参数，则结果为 **false**。

```
import std.stdio;
```

```
struct S
{
    static int s1;
    int s2;
}

int foo();
int bar();

void main()
{
    writefln(__traits(compiles));                       // false
    writefln(__traits(compiles, foo));                  // true
    writefln(__traits(compiles, foo + 1));              // true
    writefln(__traits(compiles, &foo + 1));             // false
    writefln(__traits(compiles, typeof(1)));            // true
    writefln(__traits(compiles, S.s1));                 // true
    writefln(__traits(compiles, S.s3));                 // false
    writefln(__traits(compiles, 1,2,3,int,long,std));   // true
    writefln(__traits(compiles, 3[1]));                 // false
    writefln(__traits(compiles, 1,2,3,int,long,3[1])); // false
}
```

这个是有用的，因为：

- 在范型代码里，可以给出更好的错误信息，有时是很难去跟踪编译器的。
- 相比模板部分特例化所允许的可以实现更好的 grained 特例化。

第 24 章 D 中的错误处理

我来了，我编码了，我崩溃了。-- Julius C'ster

所有的程序都要应付错误。错误是不在程序正常操作范围内的异常情况：常见的错误情况有：

- 内存耗尽。
- 磁盘空间耗尽。
- 文件名无效。
- 试图写只读文件。
- 试图读不存在的文件。
- 请求不支持的系统服务。

24.1 错误处理问题

C 语言检测报告错误的传统方法并没形成传统，每个函数都有自己的方法，包括：

- 返回 NULL 指针。
- 返回 0 值。
- 返回非零的错误代码。
- 需要检查 errno。
- 如果上述检查失败了，需要调用一个函数。

为了处理可能出现的错误，必须为每个函数添加繁琐的错误处理代码。如果发生了错误，必须有错误恢复代码，并且需要有代码将错误以某种友好的方式告诉给用户。如果不能在发生错误的局部环境处理错误，就必须显式地将错误传播给它的调用者。如果要显示错误类型，需要将一大堆 errno 值转换为合适的文字。这些代码可能占用整个项目编码时间的一大部分，而且如果运行时系统加入了一个新的 errno 值，旧有的代码就不能显示有意义的信息。

功能良好的错误处理代码很容易将清晰而简洁的实现搞得一团糟。

更糟的是，功能良好的错误处理代码本身就很容易出错，还总是整个项目中测试最不充分的部分（因此充满错误），并且还总是被简单地省略。最终的结果就如同“蓝屏死机”那样，程序由于无法处理一些未预料到的错误而失败。

并不值得为“快速而脏的(quick and dirty)”程序编写错误处理代码，因此使用这类程序就好像使用没有锯罩的多功能台式电锯。

我们所需要的是如下的错误处理哲学和方法学：

- 标准化——如果用法一致，它就更有用。
- 就算程序员无法查出出现了什么错误，也要使程序以合适的方式终止。
- 在无需改动旧代码的前提下就可以加入新的错误类型，从而重用旧的代码。
- 不会由于粗心大意而忽略某些错误。
- 重写“快速而脏的”程序直到能够正确处理错误。
- 可以很容易地写出清晰的错误处理代码。

24.2 D 的错误处理解决方案

先让我们观察一下并对错误作出以下假设：

- 错误不属于正常的程序流。错误是异常的、不常见的、不该出现的。
- 因为错误不常见，所以错误处理代码的性能并不十分重要。
- 程序的正常逻辑流的性能很关键。
- 所有的错误必须采用统一的方式处理，不论是显式地编写代码处理它们还是采用系统默认的处理方式。
- 相比于必须从错误里进行恢复的代码，负责检测错误的代码了解更多关于错误的信息。

解决方案是使用异常处理来报告错误。所有的错误都是从抽象类 Error 派生的对象。Error 类有一个叫做 toString() 的纯虚函数，它返回一个 char[] 类型的人类可读的错误描述。

如果代码检测到了一个错误，如“内存耗尽”，则抛出 Error 对象，其中包含消息“内存耗尽”。函数调用堆栈会被展开，并查找 Error 的处理程序。随着堆栈的展开，相应的 Finally 块 会被执行。如果找到了错误处理程序，就在该处恢复程序的执行。否则，会运行默认的错误处理程序，该程序会显示错误消息并终止程序。

这个解决方案是否能满足我们的要求？

标准化——如果用法一致，它就更有用。

这正是 D 的方法，D 运行时库和示例都采用这种方式。

就算程序员无法查出出现了什么错误，也要使程序以合适的方式终止。

如果没有相应的错误处理程序，程序会执行默认的错误处理程序，该处理程序打印出适当的消息，并使程序优雅的推出。

在无需改动旧代码的前提下就可以加入新的错误类型，从而重用旧的代码。

旧的代码可以决定是捕获所有的错误，或者是只捕获特定的错误并向上传播剩余的错误。无论那种情况，都不必为每个错误代码关联一个消息了，编译器会自动为其提供正确的消息。

不会由于粗心大意而忽略某些错误。

异常会采用这种或那种方式处理。不会出现使用 NULL 指针表示出错，但后面的代码却试图使用该 NULL 指针的情况。

重写“快速而脏的”程序直到能够正确处理错误。

“快速而脏的”代码根本不需要包含任何错误处理代码，也不需要检查错误。错误会被默认的处理程序捕获，显示适当的消息，然后程序会优雅的终止。

可以很容易地写出清晰的错误处理代码。

try/catch/finally 语句比那些无穷无尽的 if (出错) goto 处理程序; 语句要清晰得多。

这个解决方案是否符合我们对于的错误的假设？

错误不属于正常的程序流。错误是异常的、不常见的、不该出现的。

D 的异常处理机制完全符合上述假定。

因为错误不常见，所以错误处理代码的性能并不十分重要。

异常处理堆栈的展开相对较慢。

程序的正常逻辑流的性能很关键。

因为正常的代码流不必检查每个函数调用的返回值是否代表出错，使用异常处理来处理错误确实会使程序运行得更快。

所有的错误必须采用统一的方式处理，不论是显式地编写代码处理它们还是采用系统默认的处理方式。

如果某个错误没有对应的处理程序，就会由运行时库中的默认处理程序处理。如果一个错误被忽略了，那一定是程序员特意添加了代码用于忽略这个错误，因此这一定是在程序员意料之中的。

相比于必须从错误里进行恢复的代码，负责检测错误的代码了解更多关于错误的信息。

不需要将错误代码翻译为人类可读的字符串，错误检测代码而不是错误恢复代码负责生成正确的字符串。这种做法也使不同的程序出现相同的错误时会输出相同的消息。

使用异常来处理错误会导致其它的问题——如何写出异常安全的程序。这里有关于怎么做的说明。

第 25 章 垃圾回收

D 是一种全面采用垃圾回收(Garbage Collection)的语言。这意味着它从来不用释放内存。只需要按需分配，然后，由垃圾回收程序周期性地将所有未使用的内存返还到可用内存池。

C 和 C++ 程序员习惯于显式的管理内存分配和释放，很可能会对垃圾回收的好处和功效产生疑虑。通过在新项目中一开始就紧紧采用垃圾回收来进行编写，以及使用垃圾回收来转换现有项目的体验，实际情况表明：

- 采用垃圾回收的程序更快。这或许有些违反常理，但这是有原因的：
  - 引用计数是解决显式内存分配问题的常用解决方案。赋值时的递增和递减操作代码通常是造成程序缓慢的源头之一。将其隐藏在智能指针类之后并不能提高速度。 （无论如何，引用计数也不是全面的解决方案，因为循环引用从不会被删除。）
  - 使用析构函数来释放对象所获得的资源。对于大多数类来说，该资源就是所分配的内存。采用垃圾回收，则大多数的析构函数此时已变成了空的，而且完全可以抛弃。
  - 当对象在堆栈上分配时，那些负责释放内存的析构函数就变得十分重要。对于这些函数来说，必须确立一种机制以使异常发生时，每一个函数帧中的析构函数都会被调用以释放对象持有的内存。如果析构函数变得不相关，就没有设置特殊的堆栈帧用于处理异常，这样运行也会更快。
  - 整个用于管理内存的代码是会增强一点点。但是程序越大，它在缓存中的比例就会越小；而分页越多，它却会运行的更慢。
  - 垃圾回收只会在内存变得紧张时才会运行。当内存尚且宽裕时，程序将全速运行，不会在释放内存上花费任何时间。
  - 相对于过去缓慢的垃圾回收程序，现在的垃圾回收程序已先进很多。经过更新换代，“复制回收程序”已在很大程度上克服了早期的标记和清除算法低效的缺点。
  - 现在的垃圾回收程序实现的是堆压缩技术(heap compaction)。堆压缩技术可以减少程序正在引用的页的数量，即意味着内存访问命中率将更高，而交换也就会更少。
  - 采用垃圾回收的程序不会因为内存泄漏而崩溃。
- 垃圾回收程序回收不被使用的内存，因此不会有“内存泄露”问题；内存泄露会使长期运行的应用程序逐渐耗尽内存直至使系统崩溃。采用 GC 的程序拥有更长期的稳定性。
- 采用垃圾回收的程序有着更少的“难以发现的指针错漏”。 这是因为没有指向已经释放的内存的悬空指针。由于没有了显式的内存管理代码，所以也就不可能再有这样的错漏。
- 采用垃圾回收的程序的开发和调试更快，因为不用开发、调试、测试或维护显式的释放代码。
- 采用垃圾回收的程序会明显更小，因为没用用于管理内存释放的代码，也不需要处理内存释放异常的代码。

垃圾回收也不是“万精油”。它有着以下不足：

- 内存回收何时运行是不可预测的，所以程序可能会出现意外暂停。
- 内存回收时所花费的时间是没有上界的。尽管在实践中它通常运行得很快，但还是无

从保证。
- 除了回收程序以外的所有线程在回收操作进行过程中都会停止运行。
- 垃圾回收器也会遗留下一些内存，而显示释放器(explicit deallocator)不会有这样的情况。在实践中，这不是什么大问题，因为显式释放器通常都会泄露一些内存，致使它们最终耗尽所有内存；另一个理由就是显式释放器通常会不是把释放的内存交还给操作系统，相反，它们会将其返回给自己的内部内存池。
- 垃圾回收本应该被实现成操作系统的一个基本内核服务。但是因为现实并非如此，所以这就造成了采用垃圾回收的程序被迫带着它们的“垃圾回收实现”到处跑。尽管这个实现可以被做成一个共享 DLL，但它仍然是程序的一部分。

这些限制可以通过采用 内存管理 中介绍的技术来缓解。

25.1 垃圾回收如何工作

GC 的工作方式：

1. 在 GC 所分配内存中寻找所有的“root”指针。
2. 在 GC 所分配内存中递归扫描所有被“root”所指向的内存，查找更多指针。
3. 释放所有由 GC 所分配的但没有活动指针所指向的内存。
4. 可能的话，就通过复制分配的对象（叫做复制回收器）压缩余下被占用的内存。

25.2 外部代码同垃圾回收对象的协作

垃圾回收器找寻 root 的地方有：

1. 它的静态数据段
2. 每个线程的堆栈和寄存器
3. 所有由 std.gc.addRoot() 或 std.gc.addRange() 添加的 root

如果一个对象唯一的 root 不在它们中，则回收器将会错过它，并且会释放它占有的内存。

为了避免这种情况的发生，则需要：

- 在回收器要扫描 root 的区域之内，维持一个 root 对应该对象。
- 为使用 std.gc.addRoot() 或 std.gc.addRange() 的对象添加一个 root。
- 使用外部代码的存储分配程序重新分配一个对象或者使用 C 运行时库的 malloc/free 。

25.3 指针和垃圾回收器

D 中的指针可以被粗略地分为两类：指向垃圾回收内存的指针以及其它指针。后者包括由调用 C 的 malloc() 所创建的指针、从 C 库中获得的指针、指向静态数据的指针、指向堆栈上对象的指针等等。对于这些指针，所有在 C 里合法的指针操作都用得上。

但是，对于指向垃圾回收内存的指针和引用，对指针的操作多了一些限制。这些限制其实是微不足到的，但它却可以让垃圾回收程序的设计灵活许多。

未定义行为(Undefined behavior)：

- 将指针同其它的值进行异或运算，就如同在 C 中在链表中用异或保存指针那种小技巧那样。
- 不要使用异或技巧交换两个指针的值。
- 使用类型转换和其他的小技巧将指针存储在非指针变量内。

```
void* p;
...
int x = cast(int)p;    // 错误：未定义行为
```

垃圾回收器在构建 root 时，不会扫描非指针类型。

- 利用特定的指针对齐的特点在低位或者高位存储 flag：

```
p = cast(void*)(cast(int)p | 1);  // 错误：未定义行为
```

- 将可能指向垃圾回收堆的值转换为指针：

```
p = cast(void*)12345678;    // error:未定义行为
```

复制垃圾回收器可能会更改这个值。

- 不要将不同于 null 的“魔数”存入指针。
- 不要将指针值写入到磁盘然后再从磁盘读入到内存。
- 不要使用指针值来计算 hash（哈希）函数。采用复制技术的垃圾回收程序可能会任意地在内存中移动对象，这会使 hash 值失效。
- 不要依赖指针的顺序：

```
if (p1 < p2)              // 错误：未定义行为
    ...
```

因此，再一次提醒，垃圾回收程序可能会在内存中移动对象。

- 不要给指针加上或者减去一个偏移量，不然结果有可能会导致指针指向垃圾回收程序原来为对象分配的范围之外。

```
char* p = new char[10];
char* q = p + 6;          // 正确
q = p + 11;               // 错误：未定义行为
q = p - 1;                // 错误：未定义行为
```

- 如果这些指针可能指向 GC 堆的话，请不要位移(misalign)它们，例如：

```
align (1) struct Foo
{   byte b;
    char* p;     // 位移(misaligned)指针
}
```

如果底层硬件支持位移指针，而且该指针从未用来指向该 GC 堆，则也可以使用它们。

- 不要使用字节挨字节的内存复制方式来复制指针值。这样可能会导致中间条件，到时就没有合法的指针了；而且如果 GC 在此种条件下暂停线程的话，那么它就会搞乱内存。大部分的 memcpy() 实现是可以工作的，因为它的内部实现完成的是以排列块大于或等于一个指针大小的方式进行的复制，不过由于 C 标准并不担保此种实现，

因此在使用 memcpy() 时应特别小心。

可靠且可以做的事情：

- 使用联合共享指针的存储空间：

```
union U { void* ptr; int value }
```

- 如果存在指向垃圾回收对象对象内部的指针，就不必维护指向对象开始处的指针。

```
char[] p = new char[10];
char[] q = p[3..6];
// q 完全可以用来指向整个对象，不需要同时
// 保留 p 。
```

对于大部分任务我们可以避免使用指针。D 提供的许多功能可以让大家不再显式地使用指针，比例：引用对象、动态数组和垃圾回收。提供指针的目的是为了能成功地同 C API 衔接以及完成一些底层的工作。

25.4 使用垃圾回收器

垃圾回收程序并不能解决所有的内存释放问题。例如，如果保留了一个指向一大块数据的指针，那么就算它再也用不到了，垃圾回收程序也无法回收这块空间。为了解决这个问题，一个不错的实际操作就是：在某个对象不再使用时将指向它的引用或指针设置为 null。

这个建议只适用于静态引用或者嵌入到其他对象内的引用。它对于那些存储在堆栈上的引用没什么意义，因为回收程序并不扫描栈顶之上的部分，并且新的堆栈帧总会被初始化。

25.5 参考

- Wikipedia
- GC 问与答
- 统一处理器——垃圾回收器技术
- 垃圾回收：自动的动态内存管理算法

第 26 章 浮点

26.1 浮点运算中间值

在许多计算机上，使用较高精度的运算并不比使用较低精度的运算耗费的时间长，所以为内部的临时变量采用机器允许的最高精度对于数值运算是有意义的。这里采用的哲学是不强求语言为统一而采用各种硬件的最低精度，从而充分利用目标硬件的最佳性能。

对于浮点运算表达式的中间值来说，可能会使用高于表达式类型所要求的精度。操作数的类型只决定最低精度而不是最高精度。**实现注意：** 例如，在 Intel x86 机器上，计算的中间步骤最好（但不是必需）使用硬件提供的全部 80 位精度。

如果临时变量和公共子表达式的使用量很大的话，优化后的代码很可能会得到比未优化代码更精确的解。

算法应该以计算的最小精度为基础。如果实际的精度更高，它们也不应该退化或者失败。同扩展类型不同，float 或者 double 类型应该用于：

- 减少大型数组的内存消耗
- 在速度比正确性更为重要时
- 保持跟 C 数据和函数参数的兼容性

26.2 浮点常量合拢(Constant Folding)

不管操作数的类型是什么，在 **实数** 或更大精度的情况下都会完成浮点常量合拢。遵从的规则就是 IEEE 754 规则，同时也会使用四舍五入原则。

浮点常量被内在地以实现里至少的 **实数** 精度进行表示，不管该常量的类型。对于常量合拢也可以使用额外的精度。结果精度的处理会在编译过程中尽可能晚地进行。例如：

```
const float f = 0.2f;
writefln(f - 0.2);
```

会输出 0。非常量静态值在编译时不能被传播，因此：

```
static float f = 0.2f;
writefln(f - 0.2);
```

会输出 2.98023e-09。在需要那些不会被舍入影响的特定浮点位模式时，也可以使用十六进制浮点数常量。查找 0.2f 的十六进制值：

```
import std.stdio;

void main()
{
    writefln("%a", 0.2f);
}
```

结果就是 0x1.99999ap-3。使用十六进制常量：

```
const float f = 0x1.99999ap-3f;
writefln(f - 0.2);
```

输出 2.98023e-09。

不同的编译器设置、优化设置以及内嵌设置都会影响常量合拢(constant folding)的优化，因此浮点计算的结果可能会由于这些设置而有所不同。

26.3 负数和虚数类型

在现有的语言中，为了将复数类型添加到现存类型体系中可谓费尽周折：模板、结构、运算符重载等等，并且最终的结果几乎都是失败。失败的原因可能是由于复数运算的语义很微妙，是由于编译器不理解程序员想要做什么，因而无法对语义的实现进行优化。

所有这些的目的都只是为了避免加入一个新类型。添加一个新类型意味着编译器可以使所有的复数语义工作“正常”。然后程序员就可以依靠复数的正确（至少是稳定）的实现。

伴随而来的是对虚数的需求。虚数类型是我们可以避开一些微妙的语义问题，并且由于不用处理隐含的 0 实部，可以提高运算的性能。

虚数文字量有一个 i 作为后缀：

```
ireal j = 1.3i;
```

复数文字量没有自身特殊的语法，只需写成实数类型和虚数类型相加即可：

```
cdouble cd = 3.6 + 4i;
creal c = 4.5 + 2i;
```

复数、实数和虚数有两个特性：

```
.re      实数部分（对于虚数则为 0）
.im      虚数部分（对于实数则为 0）
```

例如：

```
cd.re           是 4.5 double
cd.im           是 2 double
c.re            是 4.5 real
c.im            是 2 real
j.im            是 1.3 real
j.re            是 0 real
```

26.4 取整控制

IEEE 754 浮点数算术包括了设置四种不同的取整模式的能力。这些都可以通过在 std.c.fenv 里的函数访问到。

26.5 异常标志

IEEE 754 浮点数算术可以为计算中发生的事件设立标志：

FE_INVALID
FE_DENORMAL
FE_DIVBYZERO
FE_OVERFLOW
FE_UNDERFLOW
FE_INEXACT
这些标志可以使用在 std.c.fenv 里的函数进行设置/重置。

26.6 浮点数比较

除了常用的 <、<=、>、>=、== 和 != 比较运算符外，D 另外提供了专用于浮点数的运算符。它们是 !<>=、<>、<>=、!<=、!<、!>= !> !<> ，并符合 C 扩展 NCEG 的语义。 参看浮点数比较。

26.7 浮点转换

为了减少长度（即它们的运行时计算时间），具体实现可能在浮点计算时进行转换。因为浮点数学并没有严格遵照数学规则，所以有些转换是无效的，尽管这样，有些编程语言还是允许这样做。

浮点表达式的下列转换不被允许，因为根据 IEEE 规则，它们会导致不同的结果。

不允许的浮点转换

转换	注释
$x + 0 \rightarrow x$	如果 *x* 为 -0，则无效
$x - 0 \rightarrow x$	如果 *x* 为 ±0，而四舍五入后趋近于 -∞，则无效
$-x \leftrightarrow 0 - x$	如果 *x* 为 +0，则无效
$x - x \rightarrow 0$	如果 *x* 为 NaN 或 ±∞，则无效
$x - y \leftrightarrow -(y - x)$	因为 (1-1=+0)，但 -(1-1)=-0，所以也无效
$x * 0 \rightarrow 0$	如果 *x* 为 NaN 或 ±∞，则无效
$x / c \leftrightarrow x * (1/c)$	如果(1/*c*) 结果为一个*确切*值，则无效
$x != x \rightarrow$ false	如果 *x* 为 NaN，则无效
$x == x \rightarrow$ true	如果 *x* 为 NaN，则无效
$x\ !op\ y \leftrightarrow\ !(x\ op\ y)$	如果 *x* 或 *y* 为 NaN，则无效

当然，那些会导致副作用的转换也是无效的。

第 27 章 D 的 x86 内联汇编

D，作为一种系统程序设计语言，提供了内联汇编的功能。对于同一个处理器家族来说，D 的内联汇编的实现是标准化了的，例如，Intel Pentium 上的 Win32 D 编译器的内联汇编的语法同 Intel Pentium 上的 Linux D 编译器的语法是一样的。

但是，不同的 D 实现，可以依内存模型、函数调用/返回约定，参数传递约定等的不同而自由实现内联汇编。

本文描述了内联汇编的 x86 实现。

```
Asm 指令:
        标识符 :Asm 指令
        align 整数表达式
        even
        naked
        db 多个操作数
        ds 多个操作数
        di 多个操作数
        dl 多个操作数
        df 多个操作数
        dd 多个操作数
        de 多个操作数
        操作码
        操作码 多个操作数

多个操作数
        单个操作数
        单个操作数 , 多个操作数
```

27.1 标号

汇编指令可以向其他语句一样带有标号。它们可以作为 goto 语句的目标。例如：

```
void *pc;
asm
{
    call L1             ;
 L1:                     ;
    pop EBX             ;
    mov pc[EBP],EBX     ;       // pc 现在指向 L1 处的代码
}
```

27.2 align *整数表达式*

汇编器使用 NOP 指令进行填充，使下一条指令对齐到 *整数表达式* 边界上。*整数表达式* 的值必须是 2 的幂。

使循环代码对齐可以使得执行速度得到可观的提升。

27.3 even

汇编器使用 NOP 指令进行填充，使下一条指令对齐到偶数边界上。

27.4 naked

禁止编译器生成函数的建帧和退帧指令。这就意味着责任落到了使用内联汇编的程序员的头上，因此这种用法主要用于那些全部用内联汇编编写的函数。

27.5 db, ds, di, dl, df, dd, de

这些伪操作用于直接向代码中插入原始数据。**db** 用于字节，**ds** 用于 16 位字，**di** 用于 32 位字，**dl** 用于 64 位字，**df** 用于 32 位浮点型，**dd** 用于 64 位双精度型，而 **de** 用于 80 位扩展实数型。它们都可应用于多个操作数。如果有操作数为字符串文字，汇编器就认为存在一个隐含的 *length* 操作数，在这里的 *length* 表示字符串中有多少了字符。每个操作数会额外使用一个字符。例如：

```
asm
{
db 5,6,0x83;   // 插入字节 0x05、0x06 和 0x83 到代码里
   ds 0x1234;      // 插入字节 0x34、0x12
   di 0x1234;      // 插入字节 0x34、0x12、0x00、0x00
   dl 0x1234;      // 插入字节 0x34、0x12、0x00、0x00、0x00、0x00、0x00、0x00
   df 1.234;       // 插入 float 1.234
   dd 1.234;       // 插入 double 1.234
   de 1.234;       // 插入 extended 1.234
   db "abc";       // 插入 byte 0x61、0x62、and 0x63
   db "abc";       // 插入 byte 0x61、0x00、0x62、0x00、0x63、0x00
}
```

27.6 操作码

本文末尾列出了支持的操作码。

支持下面的寄存器。寄存器名都是大写的。

AL, **AH**, **AX**, **EAX**
BL, **BH**, **BX**, **EBX**
CL, **CH**, **CX**, **ECX**
DL, **DH**, **DX**, **EDX**
BP, **EBP**
SP, **ESP**
DI, **EDI**
SI, **ESI**
ES, **CS**, **SS**, **DS**, **GS**, **FS**

CR0, **CR2**, **CR3**, **CR4**
DR0, **DR1**, **DR2**, **DR3**, **DR6**, **DR7**
TR3, **TR4**, **TR5**, **TR6**, **TR7**
ST
ST(0), **ST(1)**, **ST(2)**, **ST(3)**, **ST(4)**, **ST(5)**, **ST(6)**, **ST(7)**
MM0, **MM1**, **MM2**, **MM3**, **MM4**, **MM5**, **MM6**, **MM7**
XMM0, **XMM1**, **XMM2**, **XMM3**, **XMM4**, **XMM5**, **XMM6**, **XMM7**

27.6.1 特殊情况

lock, **rep**, **repe**, **repne**, **repnz**, **repz**

这些前缀指令不能同它们修饰的指令位于同一语句，它们必须单独写成一条指令。例如：

```
asm
{
    rep   ;
    movsb ;
}
```

pause

内联汇编不支持该操作码，使用

```
{
    rep  ;
    nop  ;
}
```

代替，效果是相同的。

浮点运算

使用指令的两操作数形式：

```
fdiv ST(1);      // 错误
fmul ST;         // 错误
fdiv ST,ST(1);   // 正确
fmul ST,ST(0);   // 正确
```

27.7 多个操作数

```
单个操作数：
Asm 表达式

Asm 表达式：
Asm 逻辑或表达式
Asm 逻辑或表达式 ?Asm 表达式 :Asm 表达式

Asm 逻辑或表达式：
Asm 逻辑与表达式
Asm 逻辑与表达式 || Asm 逻辑与表达式
```

```
Asm 逻辑与表达式:
    Asm 或表达式
    Asm 或表达式 && Asm 或表达式

Asm 或表达式:
    Asm 异或表达式
    Asm 异或表达式 | Asm 异或表达式

Asm 异或表达式:
    Asm 与表达式
    Asm 与表达式 ^ Asm 与表达式

Asm 与表达式:
    Asm 相等表达式
    Asm 相等表达式 & Asm 相等表达式

Asm 相等表达式:
    Asm 关系表达式
    Asm 关系表达式 == Asm 关系表达式
    Asm 关系表达式 != Asm 关系表达式

Asm 关系表达式:
    Asm 移位表达式
    Asm 移位表达式 < Asm 移位表达式
    Asm 移位表达式 <= Asm 移位表达式
    Asm 移位表达式 > Asm 移位表达式
    Asm 移位表达式 >= Asm 移位表达式

Asm 移位表达式:
    Asm 和表达式
    Asm 和表达式 << Asm 和表达式
    Asm 和表达式 >> Asm 和表达式
    Asm 和表达式 >>> Asm 和表达式

Asm 和表达式:
    Asm 积表达式
    Asm 积表达式 + Asm 积表达式
    Asm 积表达式 - Asm 积表达式

Asm 积表达式:
    Asm 括号表达式
    Asm 括号表达式 * Asm 括号表达式
    Asm 括号表达式 / Asm 括号表达式
    Asm 括号表达式 % Asm 括号表达式

Asm 括号表达式:
    Asm 一元表达式
    Asm 括号表达式 [ Asm 表达式 ]
```

```
Asm一元表达式:
Asm类型前缀 Asm表达式
offsetof Asm表达式
seg Asm表达式
+ Asm一元表达式
- Asm一元表达式
! Asm一元表达式
~ Asm一元表达式
Asm基本表达式

Asm基本表达式
整数常量
浮点数常量
__LOCAL_SIZE
    $
寄存器
点标识符

点标识符
标识符
标识符 .点标识符
```

操作数的语法基本遵从了 Intel CPU 文档的约定。具体来说，就是右边的操作数是源操作数，左边的操作数是目的操作数。同 Intel 存在不同之处主要是为了同 D 语言的记号识别器和简单解析的目标兼容。

seg 表示装载符号为 in 的段号。对于 flat 模式代码可能不适合。相反，从相关段寄存器里完成一个 move。

27.7.1 操作类型

```
Asm类型前缀:
       near ptr
       far ptr
       byte ptr
       short ptr
       int ptr
       word ptr
       dword ptr
       qword ptr
       float ptr
       double ptr
       real ptr
```

对于操作数大小模棱两可的情况，如同：

```
add     [EAX],3         ;
```

可以使用 *Asm 类型前缀* 来消除歧义：

```
add     byte ptr [EAX],3        ;
add     int ptr [EAX],7         ;
```

对于 flat 模式代码，**far ptr** 是不适合的。

27.7.2 结构/联合/类 成员偏移量

假设指向聚集的指针位于一个寄存器中，如果要访问聚集的成员，应使用成员的限定名：

```
struct Foo { int a,b,c; }
int bar(Foo *f)
{
    asm
    {    mov      EBX,f            ;
         mov      EAX,Foo.b[EBX]   ;
    }
}
```

27.7.3 堆栈变量

堆栈变量（函数的局部变量，在堆栈上面分配空间）的访问需要通过由 EBP 索引后的变量的名称来完成:

```
int foo(int x)
{
    asm
    {
        mov EAX,x[EBP]   ;   // 将参数 x 装载进 EAX
        mov EAX,x        ;   // 完成同样的工作
    }
}
```

如果 [EBP] 被省略，则它就被假定为局部变量。如果使用了 **naked**，则不再保留。

27.7.4 特殊符号

$

代表下一条指令的开始地址。所以，

```
jmp      $  ;
```

会跳转到 jmp 后的那条指令处。**$** 仅能做为 jmp 或调用指令的目标。

__LOCAL_SIZE

它的值会被局部堆栈帧中的局部字节数替代。当使用 **naked** 并且手动制定堆栈结构时，这会很方便。

27.8 支持的操作码

aaa	aad	aam	aas	adc

add	addpd	addps	addsd	addss
and	andnpd	andnps	andpd	andps
arpl	bound	bsf	bsr	bswap
bt	btc	btr	bts	call
cbw	cdq	clc	cld	clflush
cli	clts	cmc	cmova	cmovae
cmovb	cmovbe	cmovc	cmove	cmovg
cmovge	cmovl	cmovle	cmovna	cmovnae
cmovnb	cmovnbe	cmovnc	cmovne	cmovng
cmovnge	cmovnl	cmovnle	cmovno	cmovnp
cmovns	cmovnz	cmovo	cmovp	cmovpe
cmovpo	cmovs	cmovz	cmp	cmppd
cmpps	cmps	cmpsb	cmpsd	cmpss
cmpsw	cmpxch8b	cmpxchg	comisd	comiss
cpuid	cvtdq2pd	cvtdq2ps	cvtpd2dq	cvtpd2pi
cvtpd2ps	cvtpi2pd	cvtpi2ps	cvtps2dq	cvtps2pd
cvtps2pi	cvtsd2si	cvtsd2ss	cvtsi2sd	cvtsi2ss
cvtss2sd	cvtss2si	cvttpd2dq	cvttpd2pi	cvttps2dq
cvttps2pi	cvttsd2si	cvttss2si	cwd	cwde
da	daa	das	db	dd
de	dec	df	di	div
divpd	divps	divsd	divss	dl
dq	ds	dt	dw	emms
enter	f2xm1	fabs	fadd	faddp
fbld	fbstp	fchs	fclex	fcmovb
fcmovbe	fcmove	fcmovnb	fcmovnbe	fcmovne
fcmovnu	fcmovu	fcom	fcomi	fcomip
fcomp	fcompp	fcos	fdecstp	fdisi
fdiv	fdivp	fdivr	fdivrp	feni
ffree	fiadd	ficom	ficomp	fidiv
fidivr	fild	fimul	fincstp	finit
fist	fistp	fisub	fisubr	fld
fld1	fldcw	fldenv	fldl2e	fldl2t
fldlg2	fldln2	fldpi	fldz	fmul
fmulp	fnclex	fndisi	fneni	fninit

fnop	fnsave	fnstcw	fnstenv	fnstsw
fpatan	fprem	fprem1	fptan	frndint
frstor	fsave	fscale	fsetpm	fsin
fsincos	fsqrt	fst	fstcw	fstenv
fstp	fstsw	fsub	fsubp	fsubr
fsubrp	ftst	fucom	fucomi	fucomip
fucomp	fucompp	fwait	fxam	fxch
fxrstor	fxsave	fxtract	fyl2x	fyl2xp1
hlt	idiv	imul	in	inc
ins	insb	insd	insw	int
into	invd	invlpg	iret	iretd
ja	jae	jb	jbe	jc
jcxz	je	jecxz	jg	jge
jl	jle	jmp	jna	jnae
jnb	jnbe	jnc	jne	jng
jnge	jnl	jnle	jno	jnp
jns	jnz	jo	jp	jpe
jpo	js	jz	lahf	lar
ldmxcsr	lds	lea	leave	les
lfence	lfs	lgdt	lgs	lidt
lldt	lmsw	lock	lods	lodsb
lodsd	lodsw	loop	loope	loopne
loopnz	loopz	lsl	lss	ltr
maskmovdqu	maskmovq	maxpd	maxps	maxsd
maxss	mfence	minpd	minps	minsd
minss	mov	movapd	movaps	movd
movdq2q	movdqa	movdqu	movhlps	movhpd
movhps	movlhps	movlpd	movlps	movmskpd
movmskps	movntdq	movnti	movntpd	movntps
movntq	movq	movq2dq	movs	movsb
movsd	movss	movsw	movsx	movupd

movups	movzx	mul	mulpd	mulps
mulsd	mulss	neg	nop	not
or	orpd	orps	out	outs
outsb	outsd	outsw	packssdw	packsswb
packuswb	paddb	paddd	paddq	paddsb
paddsw	paddusb	paddusw	paddw	pand
pandn	pavgb	pavgw	pcmpeqb	pcmpeqd
pcmpeqw	pcmpgtb	pcmpgtd	pcmpgtw	pextrw
pinsrw	pmaddwd	pmaxsw	pmaxub	pminsw
pminub	pmovmskb	pmulhuw	pmulhw	pmullw
pmuludq	pop	popa	popad	popf
popfd	por	prefetchnta	prefetcht0	prefetcht1
prefetcht2	psadbw	pshufd	pshufhw	pshuflw
pshufw	pslld	pslldq	psllq	psllw
psrad	psraw	psrld	psrldq	psrlq
psrlw	psubb	psubd	psubq	psubsb
psubsw	psubusb	psubusw	psubw	punpckhbw
punpckhdq	punpckhqdq	punpckhwd	punpcklbw	punpckldq
punpcklqdq	punpcklwd	push	pusha	pushad
pushf	pushfd	pxor	rcl	rcpps
rcpss	rcr	rdmsr	rdpmc	rdtsc
rep	repe	repne	repnz	repz
ret	retf	rol	ror	rsm
rsqrtps	rsqrtss	sahf	sal	sar
sbb	scas	scasb	scasd	scasw
seta	setae	setb	setbe	setc
sete	setg	setge	setl	setle
setna	setnae	setnb	setnbe	setnc
setne	setng	setnge	setnl	setnle
setno	setnp	setns	setnz	seto
setp	setpe	setpo	sets	setz
sfence	sgdt	shl	shld	shr
shrd	shufpd	shufps	sidt	sldt
smsw	sqrtpd	sqrtps	sqrtsd	sqrtss

stc	std	sti	stmxcsr	stos
stosb	stosd	stosw	str	sub
subpd	subps	subsd	subss	sysenter
sysexit	test	ucomisd	ucomiss	ud2
unpckhpd	unpckhps	unpcklpd	unpcklps	verr
verw	wait	wbinvd	wrmsr	xadd
xchg	xlat	xlatb	xor	xorpd
xorps				

27.8.1 支持的 Pentium 4 (Prescott) 的操作码

addsubpd	addsubps	fisttp	haddpd	haddps
hsubpd	hsubps	lddqu	监视器 (monitor)	movddup
movshdup	movsldup	mwait		

27.8.2 支持的 AMD 操作码

pavgusb	pf2id	pfacc	pfadd	pfcmpeq
pfcmpge	pfcmpgt	pfmax	pfmin	pfmul
pfnacc	pfpnacc	pfrcp	pfrcpit1	pfrcpit2
pfrsqit1	pfrsqrt	pfsub	pfsubr	pi2fd
pmulhrw	pswapd			