D 语言编程

参考手册 1.0

(中文版 上册)

张雪平 译著

2008.09

版权声明

前 言

本电子书的制作源自 Digital Mars 官方的 DMD 手册，基于 DMD 1.035。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。DMD 2.0 的手册正在翻译整理中，相对 DMD 1.0 手册，内容会新增并更改了很多，敬请关注！如果您在阅读的过程上发现有任何错漏，欢迎来信指正。

本电子书的下载地址：http://bitworld.ys168.com

QQ 交流群：47514066（满员），51879343

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 OpenOffice.org 提供的 OpenOffice，这是一套很好的办公软件
- 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

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)

目录

第一篇 综 述

第 1 章 D 编程语言 1.0

> "在我看来，大部分“新的”编程语言都可以归结为下面两类：一类是来自具有激进创新精神(radical new paradigm)的学术界；另一类则是那些来自关注 RAD 和 web 的大型公司。或许是应该到让一门新语言是源自编译器实现的实践经验的时候了。" -- Michael
>
> "太好了，正是我所要的，编程时可以用 D 。 -- Segfault

由 Kris Bell、Lars Ivar Igesund、Sean Kelly 以及 Michael Parker 编写的 D 语言书籍《Learn to Tango with D》。现在已经出版了。

第一届《D 编程语言大会》于 2007 年 8 月 23 至 24 日在亚马逊河边的西雅图举办。

D 是一种系统编程语言。它的重点在于整合了 C 和 C++的强大及高性能，同时又具有像现代语言 Ruby 和 Python 一样的程序员生产力。对于其它像质量保证、文档、管理、移植性和可靠性等这些需求，它也给予了特别的关注。

D 语言需要静态录入，然后直接编译成机器代码。它是多模式的(multiparadigm)，即支持多种编程风格：命令式、面向对象以及元式编程。它属于 C 语法家族的成员，其形式上跟 C++ 非常接近。请参考简明的功能列表。

它不受任何公司讨论或任何已有的编程理论所左右。D 编程社区 的需要和贡献是其前进的方向。

目前有两个实现版本，用于 Win32 和 x86 Linux 平台的 Digital Mars DMD 包，和用于多个平台的 GCC D 编译器 包——包括 Windows 和 Mac OS X。

在 dsource，有大量正在成长的 D 源码和项目。更多关于 D 语言的维基、库、工具、媒体文章等等的链接，可以在 d 链接 看到。

本文档有 pdf 格式，另外还有 日语 和 葡萄牙语 翻译版本。 德语书籍《D 编程： 新的编程语言介绍》，除此之外还有日语书籍《D 语言完美指导》。

下面的 D 程序实例演示了一部分功能：

```
#!/usr/bin/dmd -run
/* 支持 sh 风格脚本语法 */

/* D 的 Hello World
编译方法：
    dmd hello.d
或者优化编译：
    dmd -O -inline -release hello.d
*/

import std.stdio;

void main(string[] args)
{
   writefln("Hello World, Reloaded");

// 自动的类型推断以及内建的 foreach
   foreach (argc, argv; args)
```

```
    {
// 面向对象的编程
        auto cl = new CmdLin(argc, argv);
// 改进的类型安全的 printf
        writefln(cl.argnum, cl.suffix, " arg: %s", cl.argv);
// 自动或显示的内存管理
        delete cl;
    }

// 嵌套结构和类
    struct specs
    {
// 所有成员自动被初始化
        int count, allocated;
    }

// 嵌套的函数可以引用
    // 像 args 那样的外层变量
    specs argspecs()
    {
        specs* s = new specs;
// 不需要使用 '->'
        s.count = args.length;              // 使用 .length 来获取数组的长度
        s.allocated = typeof(args).sizeof; // 内建的本地类型特性
        foreach (argv; args)
            s.allocated += argv.length * typeof(argv[0]).sizeof;
        return *s;
    }

// 内建字符串和常见字符串操作
    writefln("argc = %d, " ~ "allocated = %d",
        argspecs().count, argspecs().allocated);
}

class CmdLin
{
    private int _argc;
    private string _argv;

public:
    this(int argc, string argv) // 构造函数
    {
        _argc = argc;
        _argv = argv;
    }

    int argnum()
    {
        return _argc + 1;
```

```
    }

    string argv()
    {
        return _argv;
    }

    string suffix()
    {
        string suffix = "th";
        switch (_argc)
        {
          case 0:
            suffix = "st";
            break;
          case 1:
            suffix = "nd";
            break;
          case 2:
            suffix = "rd";
            break;
          default:
            break;
        }
        return suffix;
    }
}
```

注意：所有的 D 用户都必须同意，在下载并使用 D、或者阅读 D 手册时，他们都能明确地将有关知识产权的任何声明，同发送给 Digital Mars 的任何回执（可能是投递或电邮）中的版权或专利通告进行区分。

第 2 章 概述

2.1 D 是什么？

D 是一种通用的系统和应用编程语言。它比 C++ 更高级，并且同样具备编写高性能代码以及直接编写操作系统 API 和硬件接口的能力。D 很适合于编写从中等规模到那些由团队合作完成、数百万行代码规模的各种程序。D 很容易学习，它为编程者提供了很多便利，并且非常适合具有挑战性的编译器优化技术。

D 不是脚本语言，也不是一种解释型语言。它不需要 VM 宗教、或至高无上的哲学。它是一种供追求实用的程序员（那些想快速、可靠地完成工作，而又要写出的代码易维护、可读性好的人）使用的实用编程语言。

D 是数十年来实现多种语言编译器的经验的积累，是用这些语言构造大型工程的尝试的积累。D 从其它语言（主要是 C++ ）那里获得了灵感，并使用经验和现实世界的实用性来实现它。

2.2 为什么是 D？

真的是什么原因呢？是谁需要另一种编程语言？

自从 C 语言被发明以来，软件工业走过了一段漫长的道路。许多新的概念被加入了 C++ 中，但同时维护了同 C 的向后兼容性，包括兼容了原始设计中的所有的弱点。有很多尝试企图修正这些缺陷，但是兼容性却一直困扰着它。同时，C 和 C++ 都在不断引入新的特性。这些新特性必须被小心的加入到现有的结构中，以免重写旧的代码。最终的结果就是它们变得非常的复杂—— C 标准将近 500 页，C++ 标准大概有 750 页！C++ 实现起来既困难又代价高昂，造成的结果就是各种实现之间都有差别，因此很难写出完全可以移植的 C++ 代码。

C++ 程序员倾向于使用语言孤岛来编程，例如：十分精通语言中的某些特性，同时尽量避开其他特性。尽管代码通常在编译器之间是可移植的，但在程序员之间移植就不那么容易了。C++ 的一个长处是它支持很多根本上不同的编程风格——但从长远来看，互相重复和互相冲突的风格会给开发带来阻碍。

对于象可变大小的数组和字符串连接等这样的功能，C++是在标准库中实现的，而不是在语言核心里面。不在语言核心中实现这些功能造成好些 不太理想的后果。

是否能把 C++ 的能力释放、重新设计并重铸到一门简单、正交并实用的语言中呢？这种语言是否能做到易于正确实现，并使编译器有能力有效地生成高度优化的代码呢？

现代编译器技术已经取得了很大的进步，有些原来用作原始编译技术的补充的语言特性已经可以被忽略。（比如 C 语言中的关键字‘register’，另一个更为微妙的例子是 C 中的宏预处理程序。）我们可以依赖现代编译器的优化技术而不是使用语言特性来获得可以接受的代码质量。

2.2.1 D 的主要目标

- 通过加入已经被证明的能够提高生产力的特性、调整语言特性以避免常见但耗费精力

的 bug 的出现，至少减少 10%的软件开发成本。
- 是代码易于在编译器之间、在机器之间、在操作系统之间移植。
- 支持多种编程范式，也就是至少支持命令式、结构化、面向对象和范型编程范式。
- 对于熟悉 C 或者 C++ 的人来说，学习曲线会更短。
- 提供必要的底层访问能力。
- 要使 D 的编译器从根本上易于实现（相对于 C++ 来说）。
- 要同本机的 C 语言应用程序二进制接口相兼容。
- 语法要做到上下文无关。
- 对编写国际化的应用程序提供便利的支持。
- 同时支持契约式编程和单元测试方法论。
- 能够构建轻量级的、独立的程序。
- 减少创建文档的代价。

2.2.2 从 C/C++ 保留下来的功能

粗看上去 D 就跟 C 和 C++ 一样。这样一来学习以及将代码移植到 D 就非常容易。从 C/C++ 转向 D 也自然很多。程序员不必从头学起。

使用 D 并不意味着程序员会如 Java 或者 Smalltalk 那样被严格的限制在某一个运行时 vm（虚拟机）上。D 没有虚拟机，编译器直接生成可连接的目标文件。D 如同 C 完成的那样被直接跟操作系统打交道。通常那些你熟悉的工具，如 **make** 同样适用于 D 开发。

- D 将很大程度上保留 C/C++ 的**外观**。它将使用相同的代数语法，绝大多数的相同表达式和语句形式，以及总体的结构。
- D 程序既可以采用 C 的 **函数和数据**风格来写， 也可以采用 C++ 的 **面向对象**、**模板元代码**风格来写，甚至三者混合使用。
- **编译/链接/调试**开发模型被继承下来，但是没有什么能阻止 D 程序被编译成为字节码然后解释执行。
- **Exception handling**.越来越多的使用经验显示，异常处理是比 C 传统的使用出错代码/全局 errno 变量模型更为高级的错误处理模型。
- **运行时类型识别**.C++ 部分地实现了这个功能，而 D 更进一步。对运行时类型识别完全支持，具有更好的垃圾回收、更强的调试器支持、更自动持久等等好处。
- D 维持了同 **C 调用协定**的函数连接兼容。这样就能够使 D 程序直接访问操作系统的 API。程序员有关现有 API 和编程范例的知识和经验可以继续在使用 D 时使用，而只需付出极少的努力。
- **运算符重载**。D 支持对运算符的重载，这样就可以用用户定义的类型扩展由基本类型构成的类型系统。
- **模板元码编程**。模板是实现范型编程的一种手段。其他的手段包括有使用宏或者采用协变数据类型。使用宏已经过时了。协变类型很直接，但是低效且缺少类型检查。C++ 模板的问题是它们太复杂，同语言的语法不和谐，还有各种各样的类型转换和重载规则，等等。D 提供了一种简单得多的使用模板的方法。
- **RAII** (资源获得即初始化)。RAII 技术是编写可靠软件的重要方法之一。
- **Down 和 dirty 编程**。D 将保留 down&dirty 编程的能力，而不用采用别的语言编写的外部模块。在进行系统编程时，有时需要将一种指针转换成另一种指针，或者使用

汇编语言。D 的目标不是 *避免* down&dirty 编程，而是减少在进行普通程序设计时对它们的需要。

2.2.3 废弃的功能

- 对 C 的源码级兼容性。保留对 C 的源码级兼容的扩展已经有了（C++ 和 Objective-C）。在这方面的进一步工作受制于大量的遗留代码，已经很难对这些代码进行什么重大的改进了。
- 对 C++ 的链接兼容性。C++ 的运行时对象模型太复杂了——如果要较好的支持它，基本上就是要求 D 编译器变成一个完整的 C++ 编译器了。
- C 预处理程序。宏处理是一种扩展语言的简单方法，它可以给语言加入某些语言本不支持的（对于符号调试器不可见的）特征。条件编译、使用 #include 分层的文本、宏、符号连接等，本质上构成了两种难以区分两种语言的融合体，而不是一种语言。更糟的是（或许是最好的），C 预处理程序是一种十分原始的宏语言。是停下来的时候了，看看预处理程序是用来做什么的，并将这些功能直接设计到语言内部。
- 多重继承。它是一种拥有饱受争议的价值的复杂特征。它很难用一种高效的方式实现，而且在编译器实现它时很容易出现各种错漏。几乎所有的 MI 的功能都能够通过使用单根继承加接口和聚集的方式实现。而那些只有 MI 才能支持的功能并不能弥补它带来的副作用。
- 名字空间。当链接独立开发的代码时，可能会发生名字的冲突，名字空间就是解决这个问题的一种尝试。模块的概念更简单并且工作得更好。
- 标记名字空间。这是 C 的一个糟糕的特征，结构的标记名称位于一个同其它符号不同的符号表中。C++ 试图合并标记名字空间和正常的名字空间，但同时还要维持对遗留 C 代码的向后兼容性。结果是无用的混乱。
- 前向声明。C 编译器在语义上只知道什么东西实在词法上位于当前状态之前的。C++ 进行了一点点扩展，类中的成员可以依赖于它之后声明的类成员。D 更进一步，得到了一个合情合理的结论，前向声明根本就没有存在的必要。函数可以按照一种自然的顺序定义，不用再像 C 那样为了避免前向声明而采用常用的从里到外的顺序定义。
- 包含文件。造成编译器运行缓慢的原因之一是编译每个编译单元时都需要重新解析数量巨大的头文件。包含文件的工作应该采用导入到符号表中的方式来完成。
- 三字节码和双字节码。Unicode 是未来。
- 非虚成员函数。在 C++ 中，由累得设计者决定一个函数是否应该是虚函数。在子类中重写一个函数而忘记在父类中将其更新为虚函数是一个常见的（并且非常难以发现的）编码错误。将所有成员函数设置为虚函数，并由编译器来判断函数是否被重写、并由此将没有被重写的函数转换为非虚函数的做法更为可靠。
- 任意长度的位字段。位字段是一种复杂、低效并且很少用到的特征。
- 支持 16 位计算机。D 从不考虑混合使用远/近指针和其它所有用于声称好的 16 位代码的机制。D 语言的设计假设目标机器至少拥有 32 位的平坦内存空间。D 将能够被毫无困难的移植到 64 位架构上。
- 对编译遍数的互相依赖。在 C++ 中，需要一个符号表和各种的预处理程序命令才能成功的解析一个源文件。这样就使预解析 C++ 源码变得不可能，并且使编写代码分析程序和语法制导的编辑器的过程十分难以正确实现。
- 编译器的复杂性。通过降低实现的复杂度，这就更有可能出现多个的*正确* 实现。
- 让人哑口无言的浮点数。如果要人使用实现现代浮点数的硬件，那一定方便程序员的，而不是提供浮点以支持在机子里让人哑口无言的低得可怜的整体特性。尤其，D 实现必须支持 IEEE 754 算法，而且如果提供了扩展精确度，它也必须支持。

- 符号 < 和 > 的模板重载。这个选择给程序员们、C++实现人员和 C++源码解析工具商，带来了多年都存在的错漏、痛苦和混乱。它也使得没有完完整整地做过 C++编译器，就不可能正确解析 C++ 源代码。D 使用在语法上更为清楚明白的!(和)。

2.2.4 D 适合谁

- 经常使用 lint 或者类似的代码分析工具以期在编译之前减少错漏的程序员。
- 将编译器的警告级别调到最高的人和那些告诉编译器把警告作为错误的人。
- 不得不依靠编程风格规范来避免常见的 C 错漏的编程部门经理们。
- 认定 C++ 面向对象编程所允诺的功能由于 C++ 太复杂而不能达到的人。
- 沉溺于 C++ 强大的表达力但是被显式内存管理和查找指针错漏折磨得精疲力尽的人。
- 需要内建的测试和验证机制的项目。
- 开发百万行规模的程序的团队。
- 认为语言应当提供足够的特征以避免显式处理指针的程序员。
- 编写数值运算程序的程序员。D 拥有众多直接支持数值计算的特征，例如直接提供了复数类型和拥有确定行为的 NaN 和无穷大。（这些都被加进了最新的 C99 标准，但是没有加进 C++ 中。）
- 写的程序中有一半是使用像 Ruby 和 Python 那样的脚本语言，而另一半是使用 C++来提高瓶颈的程序员。D 具有许多 Ruby 和 Python 的生产力功能，这使得使用一种语言编写各个应用程序成为可能。
- D 的词法分析程序和解析程序完全互相独立，并且独立于语义分析程序。这意味着易于编写简单的工具来很好地处理 D 源码而不用编写一个完整的编译器。这还意味着源码可以以记号的形式传递个某个需要它的程序。

2.2.5 D 不适合谁

- 现实一点说，没人会把上百万行的 C 或 C++ 程序用 D 重新写一遍，因为 D 不直接兼容 C/C++ 源代码，D 并不适合于遗留程序。（但是，D 对遗留的 C API 提供了很好的支持。D 并不直接连接到那些做为 C 接口的代码。）
- 作为编程入门语言—— Basic 或者 Java 更适合于初学者。对于中级到高级的程序员来说，D 是他们优秀的第二门语言。
- 语言纯粹主义者。D 是一门实用的语言，它的每个特征都是为这个目的服务的，D 并没有想成为一门“完美”的语言。例如，D 拥有可以基本上避免在日常任务中使用指针的结构和语义。但是，D 仍然支持指针，因为有时我们需要打破这条规则。类似地，D 保留了类型转换，因为有时我们需要重写类型系统。

2.3 D 的主要功能

本节列出了一些更有意义的 D 的功能。

2.3.1 面向对象编程

2.3.1.1 类

D 的面向对象天性来自于类。采用的继承模型时单根继承加接口。类 Object 位于是整个继承体系的最底端，所以所有的类都实现了一个通用的功能集合。类通过引用的方式实例化，所以不需要用于在异常后进行清理工作的复杂代码。

2.3.1.2 操作符重载

类可以通过重载现有的运算符扩展类型系统来支持新类型。例如，创建一个 bignumber 类 ，然后重载 +、-、* 和 / 运算符，这样它就可以使用普通的代数运算语法了。

2.3.2 生产力

2.3.2.1 模块

源文件同模块是一一对应的。D 不是使用#include 来带有声明包含一个文件的文本，而是导入模块。不用担心多次导入一个模块，也不用再把头文件用`#ifndef/#endif` 或者 `#pragma once` 等等。

2.3.2.2 声明 vs 定义

C++ 的函数和类通常需要声明两次——声明位于 .h 头文件中，定义位于 .c 源文件中。这个过程易于出错而且冗长繁琐。显然，应该只需要程序员编写一次，而由编译器提取出声明信息并将它导入到符号表中。这正是 D 所做的。

样例:

```
class ABC
{
    int func() { return 7; }
    static int z = 7;
}
int q;
```

不再需要单独定义成员函数、静态成员、外部声明之类的，也不需要像这样烦人的语法：

```
int ABC::func() { return 7; }
int ABC::z = 7;
extern int q;
```

注意：当然，在 C++ 中，简单的函数如 `{ return 7; }`也可以内嵌写入，但是复杂的就不行了。另外，如果有前向引用的话，就必须保证已经声明了被引用的那个函数一个原型。下面的代码在 C++ 中是不会工作的：

```
class Foo
{
    int foo(Bar *c) { return c->bar(); }
};

class Bar
{
```

```
  public:
    int bar() { return 3; }
};
```

但等价此段的 D 代码就可以正常工作：

```
class Foo
{
    int foo(Bar c) { return c.bar; }
}

class Bar
{
    int bar() { return 3; }
}
```

D 函数是否被内嵌取决于优化设置。

2.3.2.3 模板

D 模板提供了一种简洁的方式来支持提供对在范型编程的支持，同时还提供了偏特化能力。模板类和模板函数都可以使用，只要带上 variadic 型模板参数和元组(Tuple)都行。

2.3.2.4 关联数组(Associative Arrays)

关联数组是索引可以为任意类型的数组，不像普通数组那样必须使用整数作为索引。本质上，关联数组就是散列表。关联数组使构建快速、高效、无错的符号表变得容易了。

2.3.2.5 真正的 typedef

C 和 C++ 的 typedef 实际上是类型 *别名*，因为它不会引入新的类型。D 实现了真正的 typedef：

```
typedef int handle;
```

实实在在地创造了一个新类型 **handle**。D 同样会对 typedef 引入的类型进行类型检查，并且 typedef 也参与函数重载的决策。例如：

```
int foo(int i);
int foo(handle h);
```

2.3.2.6 文档

传统意义上，文档需要做两次——首先是一些注明函数功能的注释，接着是将其重新写成一个单独的 html 或 man 文件。并且随着时间的流逝，它们自然地就会出现不同步的情况——代码升级了，而独立的文档却没有。如果能直接从嵌入在源码的注释中提取并生成所需要的完整的文档，那么不仅会节省一半准备文档的时间，还可能使得保持文档跟源码同步更为容易。Ddoc 是 D 规范的文档生成器。本页面也是使用 Ddoc 生成的。

尽管存在第三方工具为 C++ 完成这个功能，但它们也有很多不足：

- 100% 正确地解析 C++ 是相当困难的。要达到该目标需要一个完整的 C++ 编译器。第三方工具倾向于只正确解析 C++ 的字集，因此它们的使用也就局限在该字集里的源代码。
- 不同编译器支持不同版本的 C++ ,而且有着不同的 C++ 扩展。第三方工具在适应这些不同时是有困难的。
- 第三方工具不可能提供所有常见平台支持的。它们有着跟编译器根本不同的升级周期。
- 将其内建在编译器里，意味着它存在于所有的 D 实现版本中。在任何时候都有一个默认的可以使用，意味着它很可能会被使用到。

2.3.3 函数

如你所愿，D 提供常规的对函数的支持，包括全局函数、重载函数、函数内嵌、成员函数、虚函数、函数指针等等。另外，D 还支持：

2.3.3.1 嵌套函数

函数可以嵌套在其他函数内。这对于代码分解、局部性以及函数闭包技术都具有很高的价值。

2.3.3.2 函数字法

匿名函数可以直接嵌入到表达式中。

2.3.3.3 动态闭包

嵌套函数和类成员函数可以被称为闭包（也被称为委托），它们可使范型编程更为容易并保证类型安全。

2.3.3.4 In、Out 和 Inout 参数

这几个修饰符不只能使函数更为易于理解，还能避免使用指针而不会影响代码的功能，另外这也会提高编译器帮助程序员找到编码问题的可能性。

这些修饰符使 D 能够直接同更多的外部 API 对接。也就无需使用“接口定义语言”之类的东西了。

2.3.4 数组

C 数组有几个可以被纠正的缺点：

- 数组本身并不带有数组结构的信息，它们必须另外存储和传递。一个经典的例子就是传递给 `main(int argc , char *argv [])` 的 argc 和 argv 参数。（在 D 中，main 被声明为 `main(char[][] args )`。）
- 数组不是一等公民。当一个数组 被传递给函数时，他被转换为指针，尽管那个原型令人迷惑地声称它是一个数组。当发生类型转换时，所有的数组类型信息也就丢失了。
- C 数组的大小不可改变。这意味着即使最简单的聚集如堆栈都必须用一个复杂的类构

造。

- C 数组没有边界检查，因为它们根本不知道数组边界是多少。
- 数组声明中的 [] 位于标识符之后。这使得声明如一个指向数组的指针这样的东西都需要复杂难懂语法：

```
int (*array)[3];
```

在 D 中，数组的 [] 位于左侧：

int[3]* array;　　// 声明了一个指向含有 3 个 int 的数组的指针

```
long[] func(int x);      // 声明了一个返回含有 long 数据的数组
```

显然这更易于理解。

D 数组有四种变体：指针、静态数组、动态数组和关联数组。

参看 数组。

2.3.4.1 字符串(Strings)

在 C 和 C++ 中，对字符串的操作是如此的频繁，而又如此的笨拙，以至于最好还是由语言本身来支持它比较好。现代语言都处理字符串连接、复制等等操作，D 语言也提供了这些支持。

2.3.5 资源管理

2.3.5.1 自动内存管理

D 的内存分配完全采用垃圾收集。经验告诉我们，C++ 中的很多复杂特征都是用于处理内存释放的。有了垃圾回收，语言就变得简单多了。

有一种看法认为垃圾收集是给那些懒惰、初级的程序员准备的。我还记得那些对 C++ 的评论，毕竟，没有什么 C++ 能做而 C 不能做的，或者这对汇编来说也一样。

采用垃圾回收可以避免 C 和 C++ 中必需的乏味的、易于出错的内存分配和追踪代码。这不只意味着更少的开发时间和更低的维护费用，还意味着程序运行得更快！

当然，可以在 C++ 中使用垃圾回收器，在我自己的项目中就有使用它。C++ 对垃圾收集程序并不友好，这也造成了 C++ 中垃圾收集的低效。很多运行时库的代码都不能同来垃圾收集程序一同工作。

有关于这个主题的更详细的讨论，请参看 垃圾回收。

2.3.5.2 显式内存分配

尽管 D 是一种采用垃圾收集的语言，还是可以重写某个类的 new 和 delete 操作以采用一个定制的分配器。

2.3.5.3 RAII

RAII 是一种管理资源分配和释放的现代软件开发技术。D 以一种可控的、可预测的方式支持 RAII，它是独立于垃圾回收程序的回收周期的。

2.3.6 性能

2.3.6.1 轻量级聚集

D 支持简单的 C 风格的结构，既保证了对 C 数据结构的兼容性，也是因为有时采用类有杀鸡用牛刀之嫌。

2.3.6.2 内联汇编

设备驱动程序、高性能系统程序、嵌入式系统和某些特殊的代码需要使用汇编语言完成任务。尽管 D 的实现不一定要实现内联汇编，它也仍被定义为语言的一部分。他可以满足绝大多数使用汇编语言的需要，这样就不需要单独的汇编程序或者使用 DLL 了。

许多的 D 实现同时也实现那些类似于 C 的支持 I/O 端口操作、直接访问浮点硬件等内部功能的内函数。

2.3.7 可靠性

现代的语言应该竭尽所能地帮助程序员避免出错。语言提供的帮助有多种形式：从易于使用更为健壮的技术，到有编译器指出明显出错的代码，到运行时检查。

2.3.7.1 契约

契约式编程（由 B. Meyer 发明）是一种用于保证程序正确性的革命性的技术。D 版本的 DbC 包括函数先验条件、函数后验条件、类不变量和断言契约。参看 D 实现的 契约 。

2.3.7.2 单元测试(Unit Tests)

可以给一个类加入单元测试，这样测试程序就能在程序启动时自动运行。这样就能够在每次构建时都验证类是否实现了他所应完成的功能。单元测试构成了源代码的一部分。创建单元测试成为了类开发过程中的自然的一部分，而不是将完成的代码直接抛给测试小组。

单元测试可以使用其它语言完成，但是其结果看起来有一种七拼八凑的感觉，而且你采用的那种语言很可能并不兼容这个概念。单元测试是 D 的一个主要功能。对于库函数来说，单元测试已经被证明是十分有效的。它既可以保证函数工作正常，也可以演示如何使用这些函数。

考虑大量的可以从网上下载的 C++ 库和应用程序代码。其中有“几个”是带有验证测试的？更不要奢望单元测试了？少于 1%？通常的做法是，如果它们能通过编译，我们就假定它是正确的。而且我们不知道编译过程中给出的警告到底是真正的错误还是瞎唠叨。

契约式编程和单元测试使 D 为编写可信赖、健壮的系统程序的最好的语言。单元测试还是我们能够粗略但快速地估计你从未经手的 D 代码片断的质量——如果没有单元测试和契约式编程，每人会干这种事。

2.3.7.3 调试属性和语句

现在调试成为了语言语法的一部分了。可以在编译时决定是否使用这些代码，再也不用使用宏或者预处理命令了。调试语法提供了一种持续的、可移植的、易于理解的识别调试代码的方法，使程序员既能够生成带有调试代码的二进制版本，也能够生成没有调试代码的二进制版本。

2.3.7.4 异常处理(Exception Handling)

D 采用了更为高级的 *try-catch-finally* 模型而不是原来的 try-catch。没有必要只是为了利用析构函数实现 *finally* 语义而构造一个傀儡对象。

2.3.7.5 同步

因为多线程编程已经越来越成为主流，所以 D 提供了构建多线程程序的原语。同步既可以作用在方法上，也可以作用在对象上。

```
synchronized int func() { ... }
```

同步方法一次只允许一个线程执行。

同步语句将在语句块周围插入一个互斥体，控制对象或全局的访问。

2.3.7.6 对健壮性技术的支持

- 使用动态数组而不是指针
- 使用对变量的引用而不是指针
- 使用对对象的引用而不是指针
- 使用垃圾回收而不是显式内存分配
- 内建线程同步原语
- 不再有宏给你的代码来那么一下子
- 使用内联函数而不是宏
- 在很大程度上减少了使用指针的需要
- 整型的大小是明确的
- 不用再担心 char 类型是否有符号了
- 不必再分别在源文件和头文件中重复地写声明了
- 为调试代码提供了显式的解析支持

2.3.7.7 编译时检查

- 更强的类型检查
- 不允许空的";"循环体
- 赋值语句不会返回布尔类型的结果
- 废弃过时的 API

2.3.7.8 运行时检查

- assert() 表达式
- 数组边界检查
- switch 语句中的未定义 case 语句异常
- 内存耗尽异常
- in、out 和类不变量提供了对契约式编程的支持

2.3.8 兼容性

2.3.8.1 运算符优先级和求值规则

D 保留了 C 的运算符和它们的优先级、求值的规则和类型提升规则。这就避免了由于同 C 的语义不同而造成的微妙的难以发现的错漏的出现。

2.3.8.2 直接访问 C API

D 不支拥有同 C 类型对应的类型，它还提供了直接访问 C 函数的能力。完全没有必要编写封装函数和参数变换器，也没有必要逐一地复制聚集类型的成员。

2.3.8.3 支持所有的 C 数据类型

使对 C API 或者现有的 C 库代码的接口成为可能。D 支持结构、联合、枚举、指针和所有的 C99 类型。D 还拥有设置结构成员对齐方式的能力，这样就可以保证同外部导入的数据格式的兼容。

2.3.8.4 操作系统异常处理

D 的异常处理机制将在应用程序中利用底层操作系统提供的异常处理方式。

2.3.8.5 使用现成的工具

D 生成标准的目标文件格式，这样就能够使用标准的汇编程序、链接器、调试器、性能分析工具、可执行程序压缩程序和其他的分析程序，还能够同其他语言编写的代码相链接。

2.3.9 项目管理

2.3.9.1 版本控制

D 对从同一份源码生成多个版本的程序提供了内建的支持。它替代了 C 预处理程序的 #if/#endif 技术。

2.3.9.2 废弃

随着代码不停的演进，一些旧的库代码会被更新、更好的版本代替。同时旧的版本必须可用以支持旧的客户代码，旧的版本可以被标记为*废弃的*。使用废弃功能的代码通常会被认为是非法的，不过可以通过编译器开关来允许这些功能。这样一来负责维护的程序员就可以更为轻松的判断哪里是依赖于已经被废弃的特征的。

2.4 D 程序样例 (sieve.d)

```
/* Sieve of Eratosthenes prime numbers */

bool[8191] flags;

int main()
{   int i, count, prime, k, iter;

   printf("10 iterations\n");
   for (iter = 1; iter <= 10; iter++)
   {   count = 0;
      flags[] = 1;
      for (i = 0; i < flags.length; i++)
      {   if (flags[i])
         {   prime = i + i + 3;
            k = i + prime;
            while (k < flags.length)
            {
               flags[k] = 0;
               k += prime;
            }
            count += 1;
         }
      }
   }
   printf ("\n%d primes", count);
   return 0;
}
```

第 3 章 Win32 平台下的 D

本章描述的是 32 位的 Windows 系统下的 D 实现。当然，D 在 Windows 下所特有的功能通常是不能被移植到其它平台的。

在 D 语言中不用 C 语言中的：

```
#include <windows.h>
```

而是直接使用：

```
import std.c.windows.windows;
```

3.1 调用协定(Calling Conventions)

在 C 中，Windows API 的调用协定是__stdcall。在 D 中，仅需要使用：

```
extern (Windows)
{
       /* ... 函数声明 ... */
}
```

连接属性“Windows”设置了调用协定和名字碎解(name mangling)文法，以保持跟 Windows 系统的兼容性。

对于 C 中可能是 __declspec(dllimport) 或 __declspec(dllexport) 的函数，则需要使用属性 export：

```
export void func(int foo);
```

如果没有函数体，它就被导入。如果有函数体，它就被导出。

3.2 Windows 可执行文件

Windows GUI 程序也可以使用 D 来编写。在\dmd\samples\d\winsamp.d 里，可以找到一个示例。

下面这些是必不可少的：

1. 不使用 main 函数作为程序入口点，而是换成使用 WinMain 函数。
2. WinMain 必须使用下列形式：

```
import std.c.windows.windows;

extern (C) void gc_init();
extern (C) void gc_term();
extern (C) void _minit();
extern (C) void _moduleCtor();
extern (C) void _moduleDtor();
extern (C) void _moduleUnitTests();

extern (Windows)
```

```
int WinMain(HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPSTR lpCmdLine,
            int nCmdShow)
{
    int result;

    gc_init();              // 初始化垃圾回收器
    _minit();               // 初始化模块构造函数表

    try
    {
        _moduleCtor();      // 调用模块构造函数
        _moduleUnitTests(); // 运行单元测试（可选）

        result = myWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);

        _moduleDtor();      // 调用模块析构函数
    }

    catch (Object o)        // 捕捉任何未捕捉过的异常
    {
        MessageBoxA(null, cast(char *)o.toString(), "Error",
                    MB_OK | MB_ICONEXCLAMATION);
        result = 0;         // 失败
    }

    gc_term();              // 运行 finalizers；中止垃圾回收器
    return result;
}

int myWinMain(HINSTANCE hInstance,
              HINSTANCE hPrevInstance,
              LPSTR lpCmdLine,
              int nCmdShow)
{
/* ... 在这里插入用户代码 ... */
}
```

函数 `myWinMain()` 用于放置用户代码，`WinMain` 函数的其余部分是初始化和关闭 D 运行系统的样板文件。

3. `.def`(模块定义文件 - Module Definition File)文件至少包含下面两行：

```
EXETYPE NT
SUBSYSTEM WINDOWS
```

如果没有这些内容，每次应用程序运行时，Win32 都会打开一个文本控制台窗口。

4. 编译器在识别到 `WinMain()` 的存在后，便发出对 __acrtused_dll 和运行库 phobos.lib 的引用。

第 4 章 在 D 中编写 Win32 DLL

DLL (动态连接库) 是 Windows 系统编程的基础之一。D 语言支持创建多种不同类型的 DLL。

关于什么是 DLL 以及它们是怎样工作的这些背景信息，在 Jeffrey Richter 所写的书《高级 Windows》中的第 11 章是必看的内容。

本指导会给大家展示如何使用 D 来创建各种类型的 DLL。

- 带 C 接口的 DLL
- 提供 COM 服务的 DLL
- D 代码调用 DLL 中的 D 代码

4.1 带 C 接口的 DLL

带有 C 接口的 DLL 可以连接到其它任何的代码，只要编写它们的语言支持调用 DLL 中的 C 函数。

跟 C 的方式几乎一样，DLL 也可以使用 D 来创建。`DllMain()` 是不可少的，其形式是：

```
import std.c.windows.windows;
HINSTANCE g_hInst;

extern (C)
{
       void gc_init();
       void gc_term();
       void _minit();
       void _moduleCtor();
       void _moduleUnitTests();
}

extern (Windows)
BOOL DllMain(HINSTANCE hInstance, ULONG ulReason, LPVOID pvReserved)
{
    switch (ulReason)
    {
       case DLL_PROCESS_ATTACH:
          gc_init();                  // 初始化 GC（垃圾回收器）
          _minit();                   // 初始化模块列表
          _moduleCtor();              // 运行模块构造函数
          _moduleUnitTests();         // 运行单元测试
          break;

       case DLL_PROCESS_DETACH:
          gc_term();                  // 关闭 GC
          break;

       case DLL_THREAD_ATTACH:
       case DLL_THREAD_DETACH:
          // 还不支持多线程
          return false;
```

```
    }
    g_hInst=hInstance;
    return true;
}
```

注解:

- 对 _moduleUnitTests() 的调用是可选的。
- 编译器在识别到 DllMain() 的存在后，便会引用 __acrtused_dll 和运行库 phobos.lib。

跟一个 .def (Module Definition File - 模块定义文件) 一起连接，其内容大致为：

```
LIBRARY         MYDLL
DESCRIPTION     'My DLL written in D'

EXETYPE         NT
CODE            PRELOAD DISCARDABLE
DATA            PRELOAD SINGLE

EXPORTS
              DllGetClassObject       @2
              DllCanUnloadNow         @3
              DllRegisterServer       @4
              DllUnregisterServer     @5
```

在 EXPORTS 列表里的函数仅仅是举的示例。请把它们替换成实际从 MYDLL 里导出的函数。另一种方法，就是使用 implib。这里有一个示例，是关于只带有 print() 函数的这样一个简单的 DLL，该函数的功能也仅仅是输出一个字符串：

4.1.1.1 mydll2.d:

```
module mydll;
export void dllprint() { printf("hello dll world\n"); }
```

注意： 在这些实例里，我们使用了 printf 来代替 writefln，目的是尽可能让示例变得简单些。

4.1.1.2 mydll.def:

```
LIBRARY "mydll.dll"
EXETYPE NT
SUBSYSTEM WINDOWS
CODE SHARED EXECUTE
DATA WRITE
```

将上面包含有 DllMain() 的代码放入到文件 dll.d 里面。 使用下面的命令来编译和连接该动态库：

```
C:>dmd -ofmydll.dll mydll2.d dll.d mydll.def
C:>implib/system mydll.lib mydll.dll
```

```
C:>
```

这些操作会创建 mydll.dll 和 mydll.lib。现在轮到使用该动态连接库的程序 test.d：

4.1.1.3 test.d:

```
import mydll;

int main()
{
   mydll.dllprint();
   return 0;
}
```

创建 mydll2.d 一个复本，它不带函数体：

4.1.1.4 mydll.d:

```
export void dllprint();
```

使用下面命令编译并连接：

```
C:>dmd test.d mydll.lib
C:>
```

然后运行：

```
C:>test
hello dll world
C:>
```

4.1.2 内存分配

D 的 DLL 使用垃圾回收式的内存管理。其问题在于：当指向分配数据的指针超越 DLL 的界限时，会发生什么事件？如果该 DLL 带有 C 接口，那么我们你可以断定其原因在于：跟使用其它语言写成的代码进行了连接。而对于那些语言，它们并不了解 D 的内存管理方式。于是，必须使用 C 接口来屏蔽那些 DLL 的调用者，让它们无需了解无关内容。

下面是几种解决此问题的方法：

- 不向此 DLL 的调用者返回指向 D 的 GC（垃圾回收器）所分配的内存的指针。而是让该调用者分配缓存，并且把该 DLL 填充到该缓存。
- 将数据指针限制在 D 的 DLL 内部，这样“垃圾回收器”就不会将它释放掉。建立某种协议，以便调用者在安全释放数据时，可以通过它来通知 D 的 DLL。
- 使用像 VirtualAlloc() 那样的操作系统底层函数来分配用于 DLL 之间数据传输的内存块。
- 在分配用于返回给调用者的数据块时，可以使用 std.c.stdlib.malloc() (或者其它的非“垃圾回收”式分配器)。导出一个函数，让调用者可以用来释放该数据块。

4.2 COM 编程

许多 Windows API 接口都是跟 COM (Common Object Model——公共对象模型) 对象 (也叫

做 OLE 或 ActiveX 对象)有关。一个 COM 对象是这样的：其第一个字段是一个指向 vtbl[]的指针，而在 vtbl[] 里的第三个入口表示的是 QueryInterface()、AddRef() 和 Release()。

想要深入理解 COM，Kraig Brockshmidt 的《Inside OLE(深入 OLE)》 一书是不可多得的参考资料。

COM 对象跟 D 接口很相像。任何 COM 对象都可以表达为 D 接口，同时每一个带有接口 X 的 D 对象都可以做为一个 COM 对象 X。这就表示 D 跟使用其它语言实现的 COM 对象是兼容的。

虽然不是严格必须的，Phobos 库还是提供了一个对象，对于 D 的 COM 对象（名叫 ComObject），其做为一个超类是相当有用的。ComObject 提供了 QueryInterface()、AddRef() 和 Release()的默认实现。

Windows COM 对象使用 Windows 调用协定，但它并非 D 所默认的，因此 COM 函数需要拥有属性：extern (Windows)。 那么，就来编写一个 COM 对象吧：

```
import std.c.windows.com;

class MyCOMobject : ComObject
{
    extern (Windows):
        ...
}
```

这段样本代码可以做为 COM 客户端程序和服务端 DLL。

4.3 D 代码调用 DLL 中的 D 代码

让 D 里的 DLL 之间能够彼此互通，就象它们是被静态连接在一起一样，自然是很让人期待的；因为这样两个应用程序就可以被共享，而且不同的 DLL 可以独立进行开发。

潜在的困难在于怎样处理“垃圾回收(gc)”。每一个 EXE 和 DLL 都有它们自己的 gc 实例。尽管这些 gc 在没有彼此干涉的情况下可以共存，但是运行多个 gc 实例就显得多余和低效。这里想要表达的意思是：挑选一个 gc，而让那些 DLL 重定向它们的 gc 来使用它。这里使用的 gc 也就是 EXE 文件中的那个，虽然让每个 DLL 有一个 gc 也是可能的。

这个样例用于展示如何静态装载 DLL，以及如何动态装载/卸载它。

下面开始编写用于 DLL 的代码，mydll.d：

```
/*
* MyDll 演示了如何编写 D 的 DLL。
 */

import std.c.stdio;
import std.c.stdlib;
import std.string;
import std.c.windows.windows;
import std.gc;

HINSTANCE   g_hInst;
```

```
extern (C)
{
       void _minit();
       void _moduleCtor();
       void _moduleDtor();
       void _moduleUnitTests();
}

extern (Windows)
   BOOL DllMain(HINSTANCE hInstance, ULONG ulReason, LPVOID pvReserved)
{
   switch (ulReason)
   {
      case DLL_PROCESS_ATTACH:
         printf("DLL_PROCESS_ATTACH\n");
         break;

      case DLL_PROCESS_DETACH:
         printf("DLL_PROCESS_DETACH\n");
         std.c.stdio._fcloseallp = null; // 这样 stdio 就不会被关闭
         break;

      case DLL_THREAD_ATTACH:
         printf("DLL_THREAD_ATTACH\n");
         return false;

      case DLL_THREAD_DETACH:
         printf("DLL_THREAD_DETACH\n");
         return false;
   }
   g_hInst = hInstance;
   return true;
}

export void MyDLL_Initialize(void* gc)
{
   printf("MyDLL_Initialize()\n");
   std.gc.setGCHandle(gc);
   _minit();
   _moduleCtor();
//  _moduleUnitTests();
}

export void MyDLL_Terminate()
{
   printf("MyDLL_Terminate()\n");
   _moduleDtor();                        //  运行模块析构函数
   std.gc.endGCHandle();
}

static this()
{
   printf("static this for mydll\n");
}
```

```
static ~this()
{
    printf("static ~this for mydll\n");
}

/* ------------------------------------------------------------ */

class MyClass
{
    char[] concat(char[] a, char[] b)
    {
        return a ~ " " ~ b;
    }

    void free(char[] s)
    {
        delete s;
    }
}

export MyClass getMyClass()
{
    return new MyClass();
}
```

DllMain

这个是所有 D 的 DLL 的主要入口点。在 C 的启动代码部分（对于 DMC++，源代码就是 \dm\src\win32\dllstart.c）会调用它。在这里放置 **printf**，以便大家可以跟踪它是怎么样被调用的。注意：在前面的 DllMain 样例代码中初始化代码和终止代码在这里都没有。这是因为初始化会依赖于谁在装载 DLL，同时它是怎么样被装载的（静态还是动态）。在这里没必要做那么多。 The only oddity is the setting of **std.c.stdio._fcloseallp** to null.如果不将它设置成 null，C 运行时会清空并关闭所有的标准 I/O 缓存（像 **stdout**、**stderr** 等等)，同时还切断更多的输出。将其设置为 null，是调用该 DLL 的调用者负责。

MyDLL_Initialize

实际上我们有自己的 DLL 初始化例程，这样在其被调用时可以被很好地控制。在调用者自我初始化后，这些必须被调用：Phobos 运行库和模块构造器（通常是在进入 **main**() 时）。这个函数带有一个实参——指向调用者的 gc 的手柄。我们在后面会看到该如何获得该手柄。不去调用 **gc_init**() 来初始化 DLL 的 gc，而是调用 **std.gc.setGCHandle**()，同时传递指向哪个 gc 被使用的手柄。这一步通知调用者的 gc，要扫描 DLL 的哪个数据区。紧接着调用 **_minit**() 初始化模块表，调用 **_moduleCtor**() 来运行模块构造函数。**_moduleUnitTests**() 是可选的，用于运行 DLL 的单元测试。这个函数被 **导出**，就像一个函数在 DLL 之外的可见情况下被建立那样。

MyDLL_Terminate

相应地，这个函数会中止 DLL，并在卸载之前调用它。它有两个任务：通过

_moduleDtor() 调用 DLL 的模块构造器；通知运行库该 DLL 不会再通过 **std.gc.endGCHandle**()使用调用者的 gc。最后一步很关键，因为 DLL 会被移除内存，这时如果 gc 继续扫描它的数据区，就会引起段出错(segment fault)。

static this, static ~this

这些都是模块静态构造函数和析构函数的例子，在这里每个都带有一个输出，表明它们正在运行以及是什么时候运行的。

MyClass

这个例子是关于一个可以被 DLL 的调用者导出和使用的类。**concat** 成员函数会分配一些 gc 内存，**free** 会释放 gc 内存。

getMyClass

这是个“导出车间(exported factory)”，它为 **MyClass** 分配一个实例，并返回一个对它的引用。

建造(build)动态连接库 `mydll.dll`：

1. **`dmd -c mydll -g`**
   把 `mydll.d` 编译成 `mydll.obj`。**-g** 生成调试信息
2. **`dmd mydll.obj \dmd\lib\gcstub.obj mydll.def -g -L/map`**
   将 `mydll.obj` 连接进名叫 `mydll.dll` 的动态库。`gcstub.obj` 并不是必需的，不过它可以防止 gc 代码被连接进去，因为它不会再被使用。这样可以节省 12Kb。`mydll.def` 是模块定义文件, 其内容是：

   ```
   LIBRARY       MYDLL
   DESCRIPTION   'MyDll demonstration DLL'
   EXETYPE       NT
   CODE          PRELOAD DISCARDABLE
   DATA          PRELOAD SINGLE
   ```

   -g 允许生成调试信息，而 **-L/map** 会生成映像文件 `mydll.map`。
3. **`implib /noi /system mydll.lib mydll.dll`**
   创建导入库 `mydll.lib`，此库适合于跟需要要静态装载 `mydll.dll` 的应用程序进行连接。

下面是文件 `test.d`，用于测试 `mydll.dll`。它有两个版本，一个用于静态绑定到 DLL，另外一个则动态装载它。

```
import std.stdio;
import std.gc;

import mydll;

//version=DYNAMIC_LOAD;

version (DYNAMIC_LOAD)
{
    import std.c.windows.windows;

    alias void function(void*) MyDLL_Initialize_fp;
    alias void function() MyDLL_Terminate_fp;
    alias MyClass function() getMyClass_fp;
```

```
int main()
{   HMODULE h;
    FARPROC fp;
    MyDLL_Initialize_fp mydll_initialize;
    MyDLL_Terminate_fp  mydll_terminate;

    getMyClass_fp  getMyClass;
    MyClass c;

    printf("Start Dynamic Link...\n");

    h = LoadLibraryA("mydll.dll");
    if (h == null)
    {   printf("error loading mydll.dll\n");
        return 1;
    }

    fp = GetProcAddress(h, "D5mydll16MyDLL_InitializeFPvZv");
    if (fp == null)
    {   printf("error loading symbol MyDLL_Initialize()\n");
        return 1;
    }

    mydll_initialize = cast(MyDLL_Initialize_fp) fp;
    (*mydll_initialize)(std.gc.getGCHandle());

    fp = GetProcAddress(h, "D5mydll10getMyClassFZC5mydll7MyClass");
    if (fp == null)
    {   printf("error loading symbol getMyClass()\n");
        return 1;
    }

    getMyClass = cast(getMyClass_fp) fp;
    c = (*getMyClass)();
    foo(c);

    fp = GetProcAddress(h, "D5mydll15MyDLL_TerminateFZv");
    if (fp == null)
    {   printf("error loading symbol MyDLL_Terminate()\n");
        return 1;
    }

    mydll_terminate = cast(MyDLL_Terminate_fp) fp;
    (*mydll_terminate)();

    if (FreeLibrary(h) == FALSE)
    {   printf("error freeing mydll.dll\n");
        return 1;
```

```
        }

        printf("End...\n");
        return 0;
    }
}
else
{   // 静态连接这个 DLL

    int main()
    {
        printf("Start Static Link...\n");
        MyDLL_Initialize(std.gc.getGCHandle());
        foo(getMyClass());
        MyDLL_Terminate();
        printf("End...\n");
        return 0;
    }
}

void foo(MyClass c)
{
    char[] s;

    s = c.concat("Hello", "world!");
    writefln(s);
    c.free(s);
    delete c;
}
```

让我们先来完成静态连接的版本，这个相对简单些。使用下面命令编译并连接：

```
C:>dmd test mydll.lib -g
```

注意，它使用了 mydll.lib（它是mydll.dll 的导入库）来连接。代码很直接，调用 **MyDLL_Initialize()** 来初始化 mydll.lib，同时传递指向 test.exe 的 gc 的手柄。然后，我们使用 DLL，并调用它的函数，就像它是 test.exe 的一部分一样。在 **foo()** 里，gc 内存由 test.exe 和 mydll.dll 两者来分配和释放。当用完该 DLL，就使用 **MyDLL_Terminate()** 来中止它。

像下面那样运行它：

```
C:>test
DLL_PROCESS_ATTACH
Start Static Link...
MyDLL_Initialize()
static this for mydll
Hello world!
MyDLL_Terminate()
static ~this for mydll
End...
C:>
```

动态连接版本的建立略微有点困难。使用下面命令编译并连接：

```
C:>dmd test -version=DYNAMIC_LOAD -g
```

导入库 mydll.lib 并不需要了。通过调用 **LoadLibraryA**() 来装载 DLL，并且第一个导入的函数都可以通过 **GetProcAddress**() 来重新获取。一种比较简单的地获取传递给 **GetProcAddress**() 的修饰名称的方法就是：从生成的 mydll.map（在以 **Export** 开头的下面）文件里复制并粘贴。一旦完成，我们就可以使用 DLL 类的成员函数，就像它们是 test.exe 的一部分。最后，释放带有 **FreeLibrary**() 的 DLL。

像下面那样运行它：

```
C:>test
Start Dynamic Link...
DLL_PROCESS_ATTACH
MyDLL_Initialize()
static this for mydll
Hello world!
MyDLL_Terminate()
static ~this for mydll
DLL_PROCESS_DETACH
End...
C:>
```

第 5 章 在 Windows 平台调试 D

微软的 Windows 调试器\dmd\bin\windbg.exe，虽然是一个 C++ 调试器，也可以用来对 D 程序进行符号调试。 windbg.exe 的版本跟提供的那个不同的话，是不可能跟 D 一起工作的（也可以使用 Jascha Wetzel 的 Windows 版本的 Ddbg。）

要准备一个用于符号调试的程序，可以在编译时使用开关 **-g**：

```
dmd myprogram -g
```

调用调试器：

```
windbg myprogram args...
```

这里的 args... 是传递给 myprogram.exe 的命令行参数。

当调试器启动之后，就会进入到命令行窗口：

```
g _Dmain
```

会执行程序直到入口 main() 处。此时起，按 **F10** 键就会单步调试每一行代码。

基本命令：

F5
: 运行直到遇到断点、出现异常或者程序结尾。

F7
: 继续执行到光标处。

F8
: 单步调试，进入到函数调用。

F10
: 单步调试，跳过函数调用。

关于 **windbg** 的更多详细信息，请查询文件\dmd\bin\windbg.hlp。

第 6 章 将 C 的 .h 文件转换成 D 模块

虽然 D 不能直接编译 C 的源代码，但它却有着简单的到 C 代码的接口，同时可以直接连接 C 目标文件，以及在 DLL 中的 C 函数。到 C 代码的接口一般情况可以在 C 的.h 文件中找到。因此，跟 C 进行连接的关键就是将 C 的 .h 文件转换成 D 模块。实践表明这样的操作有相当的困难，因为有些东西必须要由人来进行判定。下面的指导有助于这样的转换。

6.1 预处理

有时 .h 文件可能变得让人痛苦不堪，因为过多层的宏、#include 文件、#ifdef 块，等等。D 没有象 C 那样的预处理，因此转换的第一步就是使用预处理的最终输出来代替预处理本身。对于 DMC (Digital Mars C/C++ 编译器)，命令：

```
dmc -c program.h -e -l
```

会创建一个文件—— program.lst，此文件就是所有文本预处理后的源文件。

移除所有的 #if、#ifdef、#include，等等语句。

6.2 连接属性

通常，使用下面的形式：

```
extern (C)
{
/* ...文件内容... */
}
```

来启用 C 连接属性。

6.3 类型

小范围的查找替换时，可以仔细地将 C 类型转换成 D 类型。下表所显示的是典型的 32 位 C 代码映射：

映射 C 类型到 D 类型

C 类型	D 类型
long double	real
unsigned long long	ulong
long long	long
unsigned long	uint
long	int
unsigned	uint

unsigned short	ushort
signed char	byte
unsigned char	ubyte
wchar_t	wchar or dchar
bool	bool, byte, int
size_t	size_t
ptrdiff_t	ptrdiff_t

6.4 NULL

NULL 和 ((void*)0) 应使用 null 进行替换。

6.5 数字字法

所有数字后缀 'L' 或 'l' 应该被移除，因为 C 的 long 通常跟 D 的 int 一样大小。类似地，后缀'LL'应该使用单个'L'替换。任何后缀'u'在 D 中的效果一样。

6.6 字符串字法

在大部分情况下，字符串的任何前缀'L'可以被丢弃，因为 D 会显示地将字符串转换成宽字符，如有必要的话。不过，你也可以将：

```
L"string"
```

替换为：

```
"string"w       // 16 位宽字符
"string"d       // 32 位宽字符
```

6.7 宏

如下的宏列表：

```
#define FOO     1
#define BAR     2
#define ABC     3
#define DEF     40
```

可以被替换成：

```
enum
{   FOO = 1,
    BAR = 2,
    ABC = 3,
```

```
    DEF = 40
}
```

或者是：

```
const int FOO = 1;
const int BAR = 2;
const int ABC = 3;
const int DEF = 40;
```

像下面那样的函数风格宏：

```
#define MAX(a,b) ((a) < (b) ? (b) : (a))
```

可以被替换成函数：

```
int MAX(int a, int b) { return (a < b) ? b : a; }
```

不过，如果这些函数使用的是内部静态初始化（在编译时被计算，而不是在运行时），那这种方式就没有作用。想在编译时可以被计算，则可以使用模板：

```
#define GT_DEPTH_SHIFT  (0)
#define GT_SIZE_SHIFT   (8)
#define GT_SCHEME_SHIFT (24)
#define GT_DEPTH_MASK   (0xffU << GT_DEPTH_SHIFT)
#define GT_TEXT         ((0x01) << GT_SCHEME_SHIFT)

/* Macro that constructs a graphtype */
#define GT_CONSTRUCT(depth,scheme,size) \
       ((depth) | (scheme) | ((size) << GT_SIZE_SHIFT))

/* Common graphtypes */
#define GT_TEXT16  GT_CONSTRUCT(4, GT_TEXT, 16)
```

相应的 D 版本应该是：

```
const uint GT_DEPTH_SHIFT  = 0;
const uint GT_SIZE_SHIFT   = 8;
const uint GT_SCHEME_SHIFT = 24;
const uint GT_DEPTH_MASK   = 0xffU << GT_DEPTH_SHIFT;
const uint GT_TEXT         = 0x01 << GT_SCHEME_SHIFT;

// 构造图类型的模板
template GT_CONSTRUCT(uint depth, uint scheme, uint size)
{
// 注意，常量的名字跟那个模板的名字一样
 const uint GT_CONSTRUCT = (depth | scheme | (size << GT_SIZE_SHIFT));
}

// 通常的图类型
const uint GT_TEXT16 = GT_CONSTRUCT!(4, GT_TEXT, 16);
```

6.8 声明列表

D 不允许声明列表使用不同的类型。因此：

```
int *p, q, t[3], *s;
```

应该换成：

```
int* p, s;
int q;
int[3] t;
```

6.9 Void 参数列表

不带参数的函数：

```
int foo(void);
```

在 D 中就是：

```
int foo();
```

6.10 Const 类型修饰

D 的 const 是一种存储类别(storage class)，而不是类型修饰符(type modifier)。因此，去掉所有作为“类型修饰符”的 const：

```
void foo(const int *p, char *const q);
```

变成了：

```
void foo(int* p, char* q);
```

6.11 外部全局 C 变量

无论什么时候，一旦在 D 中声明了一个全局变量，那它实际上也就被定义了。但是，如果在要被连接的 C 目标文件中也定义了该变量，则会产生“重复定义(multiple definition)”错误。要修正这个问题，请使用 extern 存储类别。例如，假设有个叫做 foo.h 的 C 头文件：

```
struct Foo { };
struct Foo bar;
```

它可以替换成 D 模块 foo.d:

```
struct Foo { }
extern (C)
{
    extern Foo bar;
}
```

6.12 Typedef（类型定义）

D 中的 alias 等效于 C 中的 typedef：

```
typedef int foo;
```

变成了：

```
alias int foo;
```

6.13 结构

将下面的声明：

```
typedef struct Foo
{   int a;
    int b;
} Foo, *pFoo, *lpFoo;
```

替换为：

```
struct Foo
{   int a;
    int b;
}
alias Foo* pFoo, lpFoo;
```

6.14 结构成员对齐

默认情况下，一个好的 D 实现会对齐结构成员，其方式跟设计来这样工作的 C 编译器一样。不过，如果 .h 文件使用了 #pragma 来控制对齐，那它们可能会跟 D 的 align 属性重复：

```
#pragma pack(1)
struct Foo
{
    int a;
    int b;
};
#pragma pack()
```

变成了：

```
struct Foo
{
  align (1):
    int a;
    int b;
}
```

6.15 嵌套结构

```
struct Foo
```

```
{
    int a;
    struct Bar
    {
        int c;
    } bar;
};

struct Abc
{
    int a;
    struct
    {
        int c;
    } bar;
};
```

变成了：

```
struct Foo
{
    int a;
    struct Bar
    {
        int c;
    }
    Bar bar;
}

struct Abc
{
    int a;
    struct
    {
        int c;
    }
}
```

6.16 __cdecl, __pascal, __stdcall

```
int __cdecl x;
int __cdecl foo(int a);
int __pascal bar(int b);
int __stdcall abc(int c);
```

变成了：

```
extern (C) int x;
extern (C) int foo(int a);
extern (Pascal) int bar(int b);
```

```
extern (Windows) int abc(int c);
```

6.17 __declspec(dllimport)

```
__declspec(dllimport) int __stdcall foo(int a);
```

变成了：

```
export extern (Windows) int foo(int a);
```

6.18 __fastcall

真是不幸，D 不支持 __fastcall 转换。因此，对于需要转换的块，要么使用 C 写成这样：

```
int __fastcall foo(int a);

int myfoo(int a)
{
    return foo(int a);
}
```

并且使用支持 __fastcall 的 C 编译器进行编译并连接进去；或者编译上面的内容，再使用 obj2asm 反汇编它，接着使用 内联汇编 的方法把它插入到 D 的 myfoo 块里。

第 7 章 D 样式

D 样式(D Style) 指的是方便书写 D 程序的一套样式风格。D 样式并非编译器强行要求的，它纯粹就是一些装饰，是可有可无的。不过，使用 D 样式，可以让其他人在处理你的代码里会更轻松，同时也让你在使用其他人的代码时更容易。D 样式可以做为为你的项目团队定制的项目风格向导的起点。

遵从 Phobos 和其它官方的 D 源代码，请使用下面的指导。

7.1 空白符(White Space)

- 每行一个语句。
- 硬制表符(hardware tabs)是 8 列缩进的。如果它们设置的值不同的话，则尽量避免使用硬制表符。
- 每一个缩进级保持在 4 列。
- 操作符跟操作数之间使用一个空格隔开。
- 使用两个空行把函数体隔开。
- 使用一个空行将函数体中的变量声明跟语句隔开。

7.2 注释(Comments)

- 单行注释请使用“//”：

```
语句;    // 注释
```

```
语句;       // 注释
```

- 多行语句块则使用块注释：

```
/*
* 注释
* 注释
 */
语句;
```

```
语句;
```

- 使用嵌套注释来“注释掉”一块试用代码：

```
/+++++
      /*
* 注释
* 注释
       */
语句;
语句;
 +++++/
```

7.3 命名协定(Naming Conventions)

通用的

通过联结多个单词组成的名字应该让除了第一个单词以外的每一个单词大写首字母。名字不应该使用下划线 '_' 作为开头。

```
int myFunc();
```

模块

模块和包的名字应该全部小写，并且只包含字符：[a..z][0..9][_]。这样可以避免在大小写敏感的文件系统中处理时出现问题。

C 模块

那些做为 C 函数接口的模块应该放置到 "c" 包里，例如：

```
import std.c.stdio;
```

模块名应该全部小写。

类、结构、联合、枚举、模板名

全都大写首字母。

```
class Foo;
class FooAndBar;
```

函数名

函数名不需要大写首字母。

```
int done();
int doneProcessing();
```

常量名

全部大写。

枚举成员名

全部大写。

7.4 无意义的类型别名

像这些：

```
alias void VOID;
alias int INT;
alias int* pint;
```

都应该避免。

7.5 声明风格

由于声明是左关联，所以需要靠左对齐它们：

```
int[] x, y;      // 记住 x 和 y 是相同的类型
int** p, q;      // 记住 p 和 q 是相同的类型
```

以强调它们的关系。请不要使用 C 风格：

```
int []x, y;      // 容易引起混淆，因为 y 也是一个整型数组：int[]
int **p, q;      // 容易引起混淆，因为 q 也是一个二维整型指针：int**
```

7.6 操作符重载

操作符重载是一个很强大的工具，可以扩展语言本身所支持的基本类型。不过，正为其强大，因此有着使代码出现迷乱的可能。特别地，已有的 D 操作符已有很直观的意义，如：'+' 表示 '加'，而 '<<' 表示 '左移'。重载操作符 '+' 时，使用跟 '加' 完全不同的意义，绝对会引起混乱，因此应该避免这样做。

7.7 匈牙利命名法

使用匈牙利命名法指定一个变量的类型是件很糟糕的事。不过，使用这样的方法来命名变量（可以通过其类型来表达）在实际当中却相当有效。

7.8 文档

所有全局声明应该使用 Ddoc 格式来进行文档说明。

7.9 单元测试(Unit Tests)

出于实际意义，所有的函数应该使用单元测试来检验——在要被检验的函数后面立即列出单元测试块。每一种可能的代码至少应该被执行一次，并使用此工具来检验：代码覆盖分析器。

第 8 章 嵌入式文档

D 语言允许将契约和测试代码嵌入到实际的代码当中，这样做有助于保持彼此之间协调。只是还缺少的一样东西，那就是文档；因为一般的注释通常不太适合用于自动提取并格式化成手册。在源代码中嵌入用户文档有相当大的好处，例如：不用两次撰写文档，就可让文档跟代码可以随时保持一致。

下面是一些可行的方法：

- Doxygen 已经部分支持 D 语言
- Java 的 Javadoc，这个可能是最有名的
- C# 的 嵌入式 XML
- 其它的 文档工具

D 支持使用嵌入式文档的目的是：

1. 不仅仅在提取出并处理后看起来也很舒服，作为嵌入的文档看起来也一样舒服。
2. 撰写轻松、自然，即最小程度地依赖于<标识>或其它笨拙的的形式——这些都是不会在最终的文档中出现的。
3. 不会重复编译器由解析代码时就已经知道的信息。
4. 不依赖于嵌入的 HTML，因为这样的话会妨碍其它特殊目的的提取和格式化。
5. 基于已有的 D 注释形式，因此这跟仅对 D 语言代码感兴趣的解析器完全无关。
6. 看起来应该跟代码完全不一样，因此看起来也不会跟代码相互混淆。
7. 应该尽可能让用户可以使用 Doxygen 或其它的文档提取器。

8.1 规范(Specification)

对于嵌入式文档注释形式的说明仅仅是指明了信息是怎么样被展现给编译器的。信息如何被使用以及最终的表现形式都是被实现定义好了的。无论最终的表现形式是 HTML 网页、man 页面、PDF 文件等等中的哪一种，这些都没有被指定为 D 语言的一部分。

8.1.1 处理的阶段

嵌入式文档注释被处理的一系列阶段是：

1. 词法分析——文档注释被标识并附着到特征符。
2. 解析——文档注释关联到特定的声明和组合。
3. 分区——每一个文档注释被分割成一连串的区域。
4. 处理特定的分区。
5. 完成非特定的分区的高亮。
6. 模块的所有分区被组合。
7. 完成宏文本替换，生成最终结果。

8.1.2 词法

嵌入式文档注释有以下几种形式：

1. /** ...*/ 在‘/’开头之后有两个‘*’
2. /++ ...+/ 在‘/’开头之后有两个‘+’

3. /// 三根斜线

下面都是嵌入式文档注释：

```
/// 这是一个单行文档注释。

/** 这个也是。 */

/++ 这个还是。 +/

/**
这是个简要的文档注释。
 */

/**
* 本行领头的那个‘*’不是文档注释的一部分。
 */

/**********************************
紧跟‘/**’之后的其它‘*’并不是
注释文档的一部分。
 */

/++
这是个简要的文档注释。
 +/

/++
+ 本行领头的那个‘+’不是文档注释的一部分。
 +/

/++++++++++++++++++++++++++++++++
紧跟‘/++’之后的其它‘+’并不是
注释文档的一部分。
 +/

/*************** 挨着的‘*’不是分开的 *****************/
```

在注释开头、结尾和左边的‘*’和‘+’会被忽略，它们也不是嵌入式文档的一部分。不符合这些形式的注释都不是文档注释。

8.1.3 解析(Parsing)

每一个文档注释都跟一个声明相关联。如果某文档注释单独在一行里，或者它的左边都只有空白符，那么它就会引用到下一个声明。应用到相同声明的多个文档注释会被连接到一起。没有关联到声明的文档注释会被忽略。在 *模块声明* 之前的文档注释会被应用到整个模块。如果文档注释在同一行里出现在某个的声明右边，那么它就会关联到该声明。

如果有一个声明的文档注释仅仅由标识符 ditto(意□ □ 同前) 组成，那么在同一个声明域内的前一个声明的文档注释对这个声明同样有效。

如果一个声明没有文档注释，那么该声明就不会出现在输出结果中。为了确保它能出现在输出中，请为它设置一个空的声明注释。

```
int a;  /// a 的文档；b 没有文档
int b;

/** c 和 d 的文档*/
/** 更多 c 和 d 的文档*/
int c;
/** ditto */
int d;

/** e 和 f 的文档 */ int e;
int f;  /// ditto

/** g 的文档 */
int g; /// 更多 g 的文档

/// C 和 D 的文档
class C
{
int x;    /// C.x 的文档

/** C.y 和 C.z 的文档*/
   int y;
   int z;    /// ditto
}

/// ditto
class D
{
}
```

8.1.4 分区(Sections)

文档注释是一系列的 *分区*。一个*分区* 是一个名字——在一行里其第一个非空字符应该是‘:’。该名字组成了该分区名。分区名大小写无关。

8.1.4.1 总结(Summary)

第一个分区是 *总结*，并且它没有分区名。直到遇到一个空行或分区名，它都是第一段。尽管总结可以是任意长度，但还是最好保持在一行之内。*总结* 分区是可选的。

8.1.4.2 描述

第二个没名字的分区是 *描述*。它包括了所有跟在 *总结* 之后的所有段落，直到遇到一个分区名或注释结束结尾。

尽管 *描述* 分区是可选的，*描述* 也将不存在，如果没有 *总结* 分区的话。

```
/**********************************
* 简要总结
* myfunc 可以干什么，这些组成了总结分区。
 *
* 大纲描述的第一个段落。
```

```
 *
* 大纲描述的
* 第二个段落。
 */

void myfunc() { }
```

紧跟在无名分区 *总结* 和 *描述* 之后的有名分区。

8.1.5 标准分区

鉴于一致性和预见性，存在有好几个标准分区。但并不是要求它们都出现。

作者(Authors):

列出该声明的作者。

```
/**
 * Authors: Melvin D. Nerd, melvin@mailinator.com
 */
```

错漏(Bugs):

列出任何已知的错漏。

```
/**
 * Bugs:负数值不工作。
 */
```

日期(Date):

指定当前版本的日期。日期格式应该可以被 std.date 所解析。

```
/**
 * Date: March 14, 2003
 */
```

废弃的(Deprecated):

如果某个声明被标记为“deprecated”，那么就在这里提供一个解释和正确的处理方法。

```
/**
 * Deprecated:由函数 bar() 所取代。
 */

deprecated void foo() { ... }
```

示例:

随意的使用格式示例

```
/**
```

```
 * Examples:
 * --------------------
* writefln("3"); // 输出 '3' 到标准输出
 * --------------------
 */
```

历史(History):

修定历史

```
/**
 * History:
*       V1 是原始版本
 *
*       V2 添加了功能 X
 */
```

证书(License):

关于代码授权的任何证书信息。

```
/**
 * License:随意使用于任何目的
 */

void bar() { ... }
```

返回值(Returns):

说明函数的返回值。如果函数返回 **void**，那么就没有必要说明了。

```
/**
* 读取文件。
 * Returns:文件内容。
 */

void[] readFile(char[] filename) { ... }
```

参考(See_Also):

列出其它的符号以及相关条目的 URL

```
/**
 * See_Also:
 *    foo, bar, http://www.digitalmars.com/d/phobos/index.html
 */
```

标准(Standards):

如果本声明跟某项标准兼容，那么就将其描述写到这里。

```
/**
 * Standards:符合 DSPEC-1234
 */
```

抛出(Throws):

列出产生的异常以及在什么环境下可能会出现。

```
/**
* 写入文件。
 * Throws:WriteException 异常，在失败时。
 */

void writeFile(char[] filename) { ... }
```

版本(Version):

指定当前声明的版本。

```
/**
 * Version: 1.6a
 */
```

8.1.6 特殊分区

有些分区具有特殊的意义和语法。

版权(Copyrights):

它包含了版权声明。当给模块声明进行注释时，COPYRIGHT 宏包含了本区的内容。版权分区仅在用于模块声明的注释时才会被特别对待。

```
/** Copyrights:公共领域(Public Domain) */

module foo;
```

参数(Params):

函数参数可以通过将其列在参数分区的方式来进行注释。凡是由一个紧跟着‘=’的标识符所引导的那行，都代表一个新的参数描述。一个描述可以跨越多行。

```
/**********************************
* foo 完成了这个。
 * Params:
 *       x =      这个
*                 而不是那个
 *       y =      是那个
 */

void foo(int x, int y)
{
}
```

宏(Macros):

宏分区的语法规则跟"**Params:**"分区的一样。它是一系列的 *名字=值* 对。*名字* 指的是宏名，而 *值* 代表的是替换文本。

```
/**
 * Macros:
 *       FOO =    现在的时间是给
*                 所有好人的
 *       BAR =    bar
 *       MAGENTA =    <font color=magenta></font>
 */
```

8.2 高亮(Highlighting)

8.2.1.1 嵌入注释

文档注释本身也可以使用 $(DDOC_COMMENT 注释文本) 这样的语法来进行注释。这些注释并不嵌套。

8.2.1.2 嵌入代码

D 代码可以通过使用至少的三个连字符（在内部便可以描述代码分区）方式来实现嵌入：

```
 /++++++++++++++++++++++++
+ 我们的函数。
+ 样例:
  + --------------------------
  +  import std.stdio;
  +
  +  void foo()
  +  {
+        writefln("foo!");   /* 输出字符串 */
  +  }
  + --------------------------
  +/
```

注意，文档注释使用了 /++ ...+/ 这样的形式，以便 /* ...*/ 可以被使用在代码分区里面

8.2.1.3 嵌入 HTML

HTML 可以被嵌入到文档注释里，而且可以被原封不动地传递到 HTML 输出当中去。不过，由于即使 HTML 是预期的嵌入文档注释提取器所输出的格式，但它也不一定是正确的，因此在实际当中最好还是尽量避免使用它。

```
/** 嵌入式 HTML 示例:
 *
 *      <li> <a href="http://www.digitalmars.com">Digital Mars</a> </li>
 *      <li> <a href="http://www.classicempire.com">Empire</a> </li>
 *
 */
```

8.2.1.4 强调

在函数参数的文档注释里或是在跟声明相关联的域名文档注释里的标识符在输出里会被进行

强调。强调的形式可以使用斜体、加粗、链接等等。怎么样强调依赖于具体情况——是函数参数、类型，还是 D 语言关键字等等。为了防止意外的标识符强调，可以使用下划线(_)来引导。在输出时下划线会被剔除掉。

8.2.1.5 字符实体(Character Entities)

有些字符对于文档处理器来说有着特别的意义，为了避免混淆，最好是使用相应的字符实体来将它们替换掉：

单个字符	实体
<	&lt;
>	&gt;
&	&amp;

在代码分区内没必要这样做，如果特殊字符后面紧跟的不是‘#’或字母也同样没必要。

8.3 宏

文档注释处理器包含简单的文本处理器。当在分区文本中出现了 $(*NAME*)，那么它就会被跟 *NAME* 相对应那个替换文本所替换。替换文本然后被递归地检查更多的宏。如果宏递归中遇到使用不带参数或使用相同参数的文本作为最后的宏，那么替换后就变成空文本。那些超出替换文本边界的宏符不会展开。如果宏名未定义，替换文本里面就不会有字符。如希望 $(NAME) 没有被展开地出现在输出里，则 $ 应该使用 $ 进行替换。

宏可以有参数。从该标识符到结束符')'的任何文本都是 $0 参数。替换文本里的 $0 会被参数文本替换。如果在参数文本里存在逗号，那么 $1 就代表第一个逗号之前的参数文本，$2 代表的文本位于第一个逗号和第二个逗号之间，等等，直到到 $9。$+ 代表位于从第一个逗号到反括号 ')' 之间所有的文本。参数文本可以包含嵌套的圆括号、"" 或 " 字符串、 注释、标识。如果有绕行，而且使用了非嵌套的圆括号，那么它们就可以使用实现 (来代表 (，而) 则代表)。

宏定义来自于下列地方（顺序无关）：

1. 预定义宏。
2. 来自 sc.ini 的 DDOCFILE 设置所指定的文件。
3. 来自命令行指定的 *.ddoc 文件。
4. 由 Ddoc 运行时生成的定义。
5. 来自所有的“宏：”分区的定义。

宏重定义会替换前面相同名字的定义。这意味着来自不同地方的宏定义序列组成是有层次的。

以 "D_" 和 "DDOC_" 开头的宏被保留了。

8.3.1 预定义宏

这些是固定在 Ddoc 中的，代表了 Ddoc 格式和加亮表示符所需要的最小定义。这些定义用于

简单的 HTML 。

```
B =     <b>$0</b>
I =     <i>$0</i>
U =     <u>$0</u>
P =     <p>$0</p>
DL =    <dl>$0</dl>
DT =    <dt>$0</dt>
DD =    <dd>$0</dd>
TABLE = <table>$0</table>
TR =    <tr>$0</tr>
TH =    <th>$0</th>
TD =    <td>$0</td>
OL =    <ol>$0</ol>
UL =    <ul>$0</ul>
LI =    <li>$0</li>
BIG =   <big>$0</big>
SMALL = <small>$0</small>
BR =    <br>
LINK =  <a href="$0">$0</a>
LINK2 = <a href="$1">$+</a>

RED =   <font color=red>$0</font>
BLUE =  <font color=blue>$0</font>
GREEN = <font color=green>$0</font>
YELLOW =<font color=yellow>$0</font>
BLACK = <font color=black>$0</font>
WHITE = <font color=white>$0</font>

D_CODE = <pre class="d_code">$0</pre>
D_COMMENT = $(GREEN $0)
D_STRING  = $(RED $0)
D_KEYWORD = $(BLUE $0)
D_PSYMBOL = $(U $0)
D_PARAM   = $(I $0)

DDOC =  <html><head>
       <META http-equiv="content-type" content="text/html; charset=utf-8">
       <title>$(TITLE)</title>
       </head><body>
       <h1>$(TITLE)</h1>
       $(BODY)
       </body></html>

DDOC_COMMENT   = <!-- $0 -->
DDOC_DECL      = $(DT $(BIG $0))
DDOC_DECL_DD   = $(DD $0)
DDOC_DITTO     = $(BR)$0
DDOC_SECTIONS  = $0
DDOC_SUMMARY   = $0$(BR)$(BR)
DDOC_DESCRIPTION = $0$(BR)$(BR)
DDOC_AUTHORS   = $(B Authors:)$(BR)
             $0$(BR)$(BR)
DDOC_BUGS      = $(RED BUGS:)$(BR)
             $0$(BR)$(BR)
DDOC_COPYRIGHT = $(B Copyright:)$(BR)
             $0$(BR)$(BR)
DDOC_DATE      = $(B Date:)$(BR)
             $0$(BR)$(BR)
DDOC_DEPRECATED = $(RED Deprecated:)$(BR)
             $0$(BR)$(BR)
```

```
DDOC_EXAMPLES  = $(B Examples:)$(BR)
              $0$(BR)$(BR)
DDOC_HISTORY   = $(B History:)$(BR)
              $0$(BR)$(BR)
DDOC_LICENSE   = $(B License:)$(BR)
              $0$(BR)$(BR)
DDOC_RETURNS   = $(B Returns:)$(BR)
              $0$(BR)$(BR)
DDOC_SEE_ALSO  = $(B See Also:)$(BR)
              $0$(BR)$(BR)
DDOC_STANDARDS = $(B Standards:)$(BR)
              $0$(BR)$(BR)
DDOC_THROWS    = $(B Throws:)$(BR)
              $0$(BR)$(BR)
DDOC_VERSION   = $(B Version:)$(BR)
              $0$(BR)$(BR)
DDOC_SECTION_H = $(B $0)$(BR)$(BR)
DDOC_SECTION   = $0$(BR)$(BR)
DDOC_MEMBERS   = $(DL $0)
DDOC_MODULE_MEMBERS   = $(DDOC_MEMBERS $0)
DDOC_CLASS_MEMBERS    = $(DDOC_MEMBERS $0)
DDOC_STRUCT_MEMBERS   = $(DDOC_MEMBERS $0)
DDOC_ENUM_MEMBERS     = $(DDOC_MEMBERS $0)
DDOC_TEMPLATE_MEMBERS = $(DDOC_MEMBERS $0)
DDOC_PARAMS    = $(B Params:)$(BR)\n$(TABLE $0)$(BR)
DDOC_PARAM_ROW = $(TR $0)
DDOC_PARAM_ID  = $(TD $0)
DDOC_PARAM_DESC  = $(TD $0)
DDOC_BLANKLINE  = $(BR)$(BR)

DDOC_PSYMBOL    = $(U $0)
DDOC_KEYWORD    = $(B $0)
DDOC_PARAM      = $(I $0)
```

Ddoc 不支持生成 HTML 代码。它只是格式化成基本的格式化宏，然后（以它们预定义的形式）展开到 HTML 里。如果输出的不是所要的 HTML，那么这些宏需要被重新定义。

基本的格式宏

B	加粗
I	倾斜
U	加上下划线
P	表示一个段落
DL	表示一个定义列表
DT	表示一个定义列表里的定义
DD	表示一个定义的描述
TABLE	表示一个表格

TR	表示一个表格里的行
TH	表示一行里的头条目
TD	表示一行里的数据条目
OL	表示一个有序列表
UL	表示一个无序列表
LI	表示列表里的一个条目
BIG	表示比某字体尺寸大些
SMALL	参数表示比某字体尺寸小些
BR	换行
LINK	生成可点击的链接
LINK2	生成可点击的链接，第一个参数为地址
RED	设置成红色
BLUE	设置成蓝色
GREEN	设置成绿色
YELLOW	设置成黄色
BLACK	设置成黑色
WHITE	设置成白色
D_CODE	表示 D 代码
DDOC	整个模板用于输出

DDOC 的特别之处在于它提供有一个模板文件，而所有生成的文本都会被插入到这个模板文件里（插入到由 Ddoc 生成的宏 **BODY** 所示的地方)。例如，为了使用样式表，**DDOC** 应该重新定义成这样：

```
DDOC =  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
       <html><head>
       <META http-equiv="content-type" content="text/html; charset=utf-8">
       <title>$(TITLE)</title>
       <link rel="stylesheet" type="text/css" href="style.css">
       </head><body>
       <h1>$(TITLE)</h1>
       $(BODY)
       </body></html>
```

DDOC_COMMENT 用于将注释插入到输出文件当中。

D 代码的加亮由下列宏完成：

D 代码格式宏

D_COMMENT	高亮注释

D_STRING	高亮字符串文字
D_KEYWORD	高亮 D 的关键字
D_PSYMBOL	高亮当前的声明名称
D_PARAM	高亮当前函数声明的参数

加亮宏以 **DDOC_** 作为开头。它们控制所表示内容单个部分的格式化。

Ddoc 分区格式化宏

DDOC_DECL	高亮声明。
DDOC_DECL_DD	高亮声明的描述。
DDOC_DITTO	高亮同前的声明。
DDOC_SECTIONS	高亮所有的分区。
DDOC_SUMMARY	高亮总结区。
DDOC_DESCRIPTION	高亮声明描述区。
DDOC_AUTHORS .. DDOC_VERSION	高亮相应的标准区。
DDOC_SECTION_H	高亮非标准区的分区名。
DDOC_SECTION	高亮非标准区的内容。
DDOC_MEMBERS	默认高亮类、结构等的所有成员。
DDOC_MODULE_MEMBERS	高亮模块的所有成员。
DDOC_CLASS_MEMBERS	高亮类的所有成员。
DDOC_STRUCT_MEMBERS	高亮结构的所有成员。
DDOC_ENUM_MEMBERS	高亮枚举的所有成员。
DDOC_TEMPLATE_MEMBERS	高亮模板的所有成员。
DDOC_PARAMS	高亮函数的参数区。
DDOC_PARAM_ROW	高亮“名字=值”函数参数。
DDOC_PARAM_ID	高亮参数名。
DDOC_PARAM_DESC	高亮参数值。
DDOC_PSYMBOL	高亮特殊区在引用的声明名称
DDOC_KEYWORD	高亮 D 的关键字。
DDOC_PARAM	高亮函数参数。
DDOC_BLANKLINE	插入空行。

例如，你可以重新定义 **DDOC_SUMMARY**：

```
DDOC_SUMMARY = $(GREEN $0)
```

现在，整个总结分区都变成绿色的了。

8.3.2 来自于 sc.ini 的 DDOCFILE 的宏定义

宏定义文本文件可以被创建和指定在 sc.ini 里面：

```
DDOCFILE=myproject.ddoc
```

8.3.3 来自在命令行的 .ddoc 文件的宏定义

在 DMD 命令行带有 .ddoc 扩展名的文件名就是要读取和处理的文本文件。

8.3.4 由 Ddoc 生成的宏定义

BODY	设置成生成的文档文本。
TITLE	设置成模块名。
DATETIME	设置成当前的日期和时间。
YEAR	设置成当前的年号。
COPYRIGHT	设置成所有属于模块注释的 **版权：**区的内容。
DOCFILENAME	设置成要生成的输出文件的名字。

8.4 将 Ddoc 用于其它的文档

Ddoc 主要的目的就是用于将嵌入式注释生成文档。但它也可以用于处理其它的文档。这样做的原因就是利用 Ddoc 的宏能力以及 D 代码的语法加亮能力。

如果 .d 源文件是以字符串 "Ddoc" 开头，那么它就被当作一般目的的文档处理，而不是作为 D 代码源文件。从 "Ddoc" 串之后到文件末尾或任何 "宏:"分区整个部分构成了文档。在该文本里没有自动加亮，除了加亮使用“--- lines”嵌入到两行之间的 D 代码。只以宏处理才会被完成。

许多的 D 文档，当然包括本页，就是使用这种方式生成的。这种文档的底部都会有标记，表明是使用 Ddoc 生成的。

8.5 参考

CandyDoc 是一个很好的关于如何使用宏和样式表来定制 Ddoc 结果的实例。

第 9 章 基本原理

关于 D 的各种设计决定的缘由的问题，经常可以被看到。那么，这里略谈几个。

9.1 操作符重载

9.1.1 为什么不把它们命名成 operator+()、operator*() 等等呢？

这种方式是属于 C++ 的，这样使得能够引用由 'operator+' 重载的 '+'。麻烦是有些事情并不十分适合。例如，比较操作符 <、 <=、> 跟 >=。在 C++ 里，这四个都必须要重载以便能够获得完整覆盖(coverage)。在 D 里，必须要定义的仅 cmp() 一个函数就行了，而比较操作则通过语义分析从它那里继承。

operator/() 重载也提供非对称方式（作为成员函数）重载反向的操作。例如：

```
class A
{
   int operator/(int i);                // 重载 (a/i)
   static operator/(int i, A a)         // 重载 (i/a)
}
```

第二个重载完成了反向的重载，不过它不能是虚的(virtual)，如果那样的话就会跟第一个重载形成混乱的不对称。

9.1.2 为什么不大范围允许定义好的操作符重载函数？

1. 因为操作符重载需要参数是对象才能实现，这样它们逻辑上就属于那个对象的成员函数。这样就造成了这种情况：当操作数是不同类型的对象时该如何处理：

   class A { }
   class B { }

   ```
   int opAdd(class A, class B);
   ```

   Should opAdd() be in class A or B?显然通过格式调整的解决方法就是将它放置到第一个操作数的类里。

   class A
   {
   　　int opAdd(class B) { }

   ```
   }
   ```

2. 操作符重载一般需要访问类的私有成员，而让它们成为全局的则会破坏类的面向对象封装。
3. (2) 能够被操作符重载定址，同时自动获得“友好的”访问；不过这种情况比较少见，而且也不符合 D 的简单性要求。

9.1.3 为什么不允许用户可定义的操作符？

这些对于连接新的中缀操作符(infix operation)到各种 unicode 符号可能非常有用。问题在

于，在 D 里这些特征符跟语义分析完全无关的。用户可定义操作符会破坏这点。

9.1.4 为什么不允许用户可定义的操作符优先？

主要的问题是这会影响语法分析，而且语法分析跟 D 里的语义分析是完全无关的。

9.1.5 Why not use operator names like __add__ and __div__ instead of opAdd, opDiv, etc.?

关键字"__"表示的应该是一种专属语言(proprietary language)扩展，而不是语言的基本部分。

9.1.6 为什么不让二元操作符重载成为静态成员呢？这样的话就指定两个参数，并且对于反向操作不会再有任何问题。

这意味着操作符重载不能是虚的(virtual)，而且可能实现成为一个包含了另一个虚函数的外壳，以便完成实际的工作。这个会消失的，就像丑陋的老马(ugly hack)一样。 其次，函数 opCmp() 已经是一个在 Object 里的操作符重载，不过由于一些原因，它需要是虚的(virtual)；而让它跟其它操作符重载完成的方式不对称，则是不必要的混淆。

9.2 特性

9.2.1 为什么 D 在语言核心里会有那样的特性，像使用 T.infinity 就可以给出浮点类型的 infinity(无穷大)，而不是像在 C++ 里那样需要通过库来得到：`std::numeric_limits::infinity`？

我们可以将此问题改述成这样：“如果在已有的语言里存在一种表达它的方式，为什么还要将它内建到语言核心里呢？”关于 T.infinity：

1. 将它内建到语言核心里，意味着该语言就能知道浮点 infinity(无穷大)是什么。如果是位于模板、类型定义(typedef)、cast、常量位模式(const bit patterns)等等里面，那么语言就不知道它是什么，并且在使用不正确时，就不可能给出准确的错误信息。
2. (1) 的副作用就是，在常量合拢(constant fold)以及其它优化方面，不可能高效地使用它。
3. 模板初始化、装载 `#include` 文件等等，所有这些都是非常耗费编译时间和编译内存的。
4. 然而，最糟的是为了获得 infinity(无穷大)，却没了长度，即表明“语言和编译器根本就不了解 IEEE 754 浮点数——因此不能依赖于它。”实际上许多其它方面优秀的 C++ 编译器浮点在做比较时并不能正确处理 NaN。(Digital Mars C++ 可以正确处理。)关于在表达式或库函数里对 NaN 或 Infinity 进行处理，C++98 并没有提及。因此可以肯定它是不工作的。

总结一下，要支持 NaN 和 infinitie 所做的工作并不比让模板返回“位模式(bit pattern)”少得了多少。它必须被内建到编译器的核心逻辑里去，并且必须体现在所有处理浮点数的库代码里。而且它还必须要在标准里。

举例说明，如果 op1 或者 op2 或者两者都是 NaN，那么：

```
(op1 < op2)
```

并不能产生等同于下面的结果：

```
!(op1 >= op2)
```

要求 NaN 是正确实现的。

9.3 为什么使用 static if(0) 而不用 if (0)呢？

几条限制：

1. if (0) 引入了新的域，而 static if(...) 却没有。为什么这个值得注意呢？如果有人想有条件地声明一个新的变量，那这个就有关系了：

   static if (...) int x; else long x;

   ```
   x = 3;
   ```

   对应地：

   if (...) int x; else long x;

   ```
   x = 3;     // error, x is not defined
   ```

2. static if 条件为假的那部分并不要求语义上有效。例如，它可能依赖于一个条件编译的声明或其它的什么东西：

   ```
   static if (...) int x;
   int test()
   {
       static if (...) return x;
       else return 0;
   }
   ```

3. Static if 可以出现在只有声明才被允许的地方：

   ```
   class Foo
   {
           static if (...)
               int x;
   }
   ```

4. Static if 能够声明新的类型别名：

   ```
   static if (0 || is(int T)) T x;
   ```

第 10 章 警告

根据有的人的观点，警告既是一种破碎的语言设计的征兆，也是一种有用的“lint(棉绒)”——就像用于分析代码以及查找潜在麻烦点(trouble spot)的工具。大部分的时间，这些麻烦点将成为合法的代码，好像就是那么回事一样。更为罕见的是，它可能会暗示一种无心错误会成为程序员的一部分。

警告不会被定义成 D 编程语言的一部分。它们存在于编译器服务商的判断力里，而且很大可能因服务商不同而有所不一样。所有的构造（具体实现可能会为它生成一个警告信息）都是合法的 D 代码。

这些都是由 Digital Mars D 编译器在提供了“**-w**”开头时所生成的警告信息。大部分都已经产生了一些激烈的争论，具体就是——它到底应该是一个警告还是一个错误，以及在这些情形下正确的编写 D 代码的方式应该是什么。因为没有一致的看法出现，所以它们都是做为可选的警告信息出现的，同时也作为又一个关于如何编写正确代码的选择。

10.1 警告——将类型为 *type* 的表达式 *expr* 隐式转换到类型 *type* 可能引起数据丢失

D 遵从 C 和 C++ 的关于在表达式里默认整数提升(integral promotion)的规则。这样做的原因就是既为了兼容性，也因为现在的设计的 CPU 在执行这样的语义时都很有效率。这些规则也包括了隐式变窄(narrowing)规则，这里此规则的结果可以转换回一个更小的类型，在这个过程中就可能丢失一些有意义的数位：

```
byte a,b,c;
...
a = b + c;
```

b 和 *c* 都被提升为类型 **int**，使用的就是默认整数提升规则。相加的结果然后也是 **int** 类型。为了赋值给 *a*，那个 **int** 就会隐式转换成一个 **byte**。

它在程序里是不是一个错漏，这要依赖于 *b* 和 *c* 的值是否有可能会引起溢出，以及该举出对于这个算法是否重要。解决这个警告的方法：

1. 插入一个 cast：

```
a = cast(byte)(b + c);
```

   这样就消除了该警告，不过由于类型转换(cast)是一种生硬的工具(blunt instrument)——它会跳过类型检查，因此它就可能掩盖了另外的错漏——如果 *a*、*b* 或者 *c* 的类型在将来改变，或者如果它们的类型是由模板实例设置的，那么就有可能引入这样的错漏。（在泛型代码(generic code)里, `cast(byte)` 很可能是 `cast(typeof(a))`）。
2. 将 *a* 的类型更改至 **int**。这个通常是一种更好的解决办法；只是如果它必须为一个更小的类型，这种办法当然就不可能有效；这样就引导我们去：
3. 执行运行时举出检查：

   byte a,b,c;
   int tmp;
   ...

```
tmp = b + c;
assert(cast(byte)tmp == tmp);
a = cast(byte)tmp;
```

用于保证不会丢失有意义的数位。

一些被提议的语言解决办法：

1. 让编译器自动插入跟选项 3 等效的运行时检查。
2. 添加新的构想(construct)：**implicit_cast**(□ 型) 表□ 式 ，它属于泛型类型转换的一种受限形式，并且只在允许隐式类型转换的地方才工作。
3. 把 **implicit_cast** 实现成一个模板。

10.2 警告——数组的“length”隐藏了在外层作用域里的“length”名字

在一个数组索引表达式或分割的“[ ]”里面，变量 *length* 会被定义，并被设置成该数组的长度。*length* 的作用域是从开始的 '[' 到结尾的 ']'。

如果在外层作用域里，*length* 被声明成一个变量名，那么那个版本就可能被试图用于“[ ]”之间。例如：

```
char[10] a;
int length = 4;
...
return a[length - 1];    // 返回 a[9]，而非 a[3]
```

解决这个警告的方法：

1. 把外面的 *length* 更改为其它的名字。
2. 把“[ ]”里的使用的 *length* 替换成 *a.length*。
3. 如果 *length* 处于全局作用域或者类域，并且它就是试图被使用的那个，那么请使用 *.length* 或者 *this.length* 来消除歧义。

一些被提议的语言解决办法：

1. 把 *length* 作为一个特殊符号或关键字，用以代替隐式将它声明成一个变量。

10.3 警告——在函数最后没有返回

考虑下面的情况：

```
int foo(int k, Collection c)
{
    foreach (int x; c)
    {
        if (x == k)
            return x + 1;
    }
```

```
}
```

同时假定算法的本意是：那个 *x* 总是在 *c* 里被找到，这样 **foreach** 就绝不会因失败执行到该代码的底部。在函数的结束处没有 **return**，因为该点是不可到达的，并且所有的 return 语句都会是死代码(dead code)。这是个完全合法的代码。为了有助于保证此类代码的健壮性(robustness)，D 编译器会通过在函数结束处自动插入一个 *assert(0);* 语句来确保它是正确的。这样，在那个 **foreach** 绝不会失败的假设里，如果有一个错误，它就会在运行时由于明显的断言(assertion)失败而被捕捉到；这样就比让该代码只是离开，从而引起不确定、随意的失败要好很多。

不过对于 D 的此种行为，在其它语言里并不常见，而且有些程序员对于要依赖于它相当地不舒服，有的则希望在编译时而不是在运行时看到这样的错误。他们希望看到东西能被显式表达在那个 **foreach** 不会失败的源码里。因此有这个警告。

解决这个警告的方法：

1. 在函数结尾处放置一个 return 语句，返回某种任意值（毕竟，它绝不会被执行）：

```
int foo(int k, Collection c)
{
    foreach (int x; c)
    {
        if (x == k)
            return x + 1;
    }
    return 0;   // 禁止关于无 return 语句的警告
}
```

   虽然这样做相当普遍，且当程序员无什么经验或很匆忙时才发生，但它也是个极糟的解决办法。当 **foreach** 由于某个错漏失败时这种麻烦才会发生，这时该函数不会检测到这个错漏，而是返回一个未定值，这样就可能引起其它的问题或根本就检测不到问题。这里还有另外一个问题：看到这个的维护员在分析这段代码的行为时，就会迷惑——为什么这里有一个 return 语句，而它又从不会被执行。（这个可以使用注释进行解决，不过注释总是会缺少(missing)、过期或只是简单错误(just plain wrong)。）死代码对于程序维护一直是个容易令人混淆的问题，因此故意地插入它也是个很糟的想法。

2. 更好的解决办法是让 D 语言隐式完成的显现出来——在那里放置一个断言(assert)：

```
int foo(int k, Collection c)
{
    foreach (int x; c)
    {
        if (x == k)
            return x + 1;
    }
    assert(0);
}
```

   现在，如果 **foreach** 失败，那么这个错误就会被检测到。另外，它是自文档说明(self-documenting)的。

3. 另一种解决办法是完成一个带有自定义错误类和更为用户友好信息的 **throw**：

```
int foo(int k, Collection c)
{
    foreach (int x; c)
    {
        if (x == k)
            return x + 1;
    }
    throw new MyErrorClass("fell off the end of foo()");
}
```

10.4 警告——switch 语句没有 default

类似于无 return 语句警告的是在一个 switch 警告里没有 default。请考虑：

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

根据 D 语言语义，这个表示了唯一的可能值 *i* 为 1 和 2。任何其它的值都会在程序里带来一个错漏。为了检测这个错漏，编译器把 switch 实现成了这个样子：

```
switch (i)
{
    case 1: dothis(); break;
    case 2: dothat(); break;
    default: throw new SwitchError();
}
```

这是跟 C 和 C++ 完全不同的行为，在它们里面的 switch 是类似于这样的：

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

使用这个的错漏潜在性是很高的，因为偶而忽略一种情况(case)，或者在程序的一部分添加一个新的值而又忘了为它在另外一部分添加新的情况(case)，这都是常见的错误，虽然在运行时 D 可以捕捉到这个错误，抛出 `std.switcherr.SwitchError` 的一个实例，但是有的人还是宁愿至少有一个警告，这样就可以在编译时捕捉到这种错误。

解决这个警告的方法：

1. 插入一个显式的 default：

```
switch (i)
{
    case 1: dothis(); break;
    case 2: dothat(); break;
    default: assert(0);
}
```

2. 跟缺少了 return 语句的情况一样，让 default 带上具有自定义 error（错误）类的 thrown，然后使用它。

10.5 警告——语句不可到达

考虑下面的代码：

```
int foo(int i)
{
   return i;
   return i + 1;
}
```

第二个 return 语句是不可到达的，即是说，它为死代码。虽然死代码在发行后的代码里属于“劣质风格(poor style)”，但在试着隔离开一个错漏或者体验不同的大小块的代码时，它还是可以合法地大量出现。

解决这个警告的方法：

1. 使用 /+ ...+/ 注释掉死代码：

```
int foo(int i)
{
    return i;
    /+
        return i + 1;
     +/
}
```

2. 把死代码放置到 `version(none)` 条件里面：

```
int foo(int i)
{
    return i;
    version (none)
    {
        return i + 1;
    }
}
```

3. 在建立发行版时，仅启用警告进行编译。

第二篇 高级技术

第 11 章 内存管理

任何有意义的程序都需要分配和释放内存。随着程序复杂性、大小的增长和性能的提高，内存管理技术变得越来越重要。D 提供了多种管理内存的方式。

在 D 中主要有三种分配内存的方法：

1. 静态数据，分配在默认数据段内。
2. 堆栈数据，分配在 CPU 程序堆栈内。
3. 垃圾回收的数据，动态分配于垃圾回收堆上。

本章描述了使用它们的技术，同时还有一些高级话题：

- 字符串（和数组）的写时复制
- 实时
- 平滑操作
- 自由链表
- 引用计数
- 显式类实例分配
- 标记/释放
- RAII (资源获得即初始化)
- 在堆栈上分配类实例
- 在堆栈上分配未初始化的数组
- 中断服务例程

11.1 字符串（和数组）的写时复制

考虑将一个数组传递给函数的情况，可能会修改该数组的值，以及返回修改后的数组。因为数组是通过引用传递的，而不是通过值传递，因此有个关键的问题就是：谁掌控数组的内容？例如，一个将所有字母转换大写的函数：

```
char[] toupper(char[] s)
{
    int i;

    for (i = 0; i < s.length; i++)
    {
        char c = s[i];
        if ('a' <= c && c <= 'z')
            s[i] = c - (cast(char)'a' - 'A');
    }
    return s;
}
```

注意，调用者的那个 s[] 也被修改了。这可能根本就不是所要的，更糟的是，s[] 可能会位于一块只读的内存中。

如果 toupper() 老是去复制 s[]，则对于已经全部是大写的字符串就会消耗不必要的时间和内存。

解决方案就是实现“写时复制”，这个意味着只有在串需要被修改时才会被复制。一些串处

理语言将它作为默认的行为，但是这么做代价巨大。串"abcdeF"将会在函数中被复制 5 次。如果要使用该协议来获得最高的效率，就必须在代码中显示的使用它。这里的 toupper() 被改写了，以便高效地实现的“写时复制”：

```
char[] toupper(char[] s)
{
    int changed;
    int i;

    changed = 0;
    for (i = 0; i < s.length; i++)
    {
        char c = s[i];
        if ('a' <= c && c <= 'z')
        {
            if (!changed)
            {   char[] r = new char[s.length];
                r[] = s;
                s = r;
                changed = 1;
            }
            s[i] = c - (cast(char)'a' - 'A');
        }
    }
    return s;
}
```

在 D Phobos 运行库的数组处理函数都实现了“写时复制”协议。

11.2 实时

实时编程意味着程序必须能够保证一个最大延迟，或者说完成操作的最长时间。在大多数内存分配方案中，包括 malloc/free 和垃圾回收，理论上延迟都是无限制的。保证延迟的最可靠的方法是预先为对时间要求苛刻的部分分配全部的数据。如果没有对内存分配的调用，GC 也不会运行，因此它并不会引起超出最大延迟。

11.3 平滑操作

同实时编程相关的就是对一个程序操作平滑的需求，即要求当垃圾回收器停止所有其他所有活动来进行垃圾回收时，程序不会随意的暂停。这种程序的一个例子就是是互动射击类游戏。如果游戏不规则的乱停，尽管这个对于程序的来说并不致命，但也会让用户恼怒不已。

有几种技术可以用来消除或者减轻这种影响：

- 在运行那些必须平滑运行的代码之前，预先分配所有的数据。
- 在程序执行过程中自己已经暂停的那个时候，手动运行垃圾回收循环。这种例子可以是当程序刚刚显示了供用户输入的提示符而用户暂时没有响应时。这会减少在平滑运

行的代码时将会需要的垃圾回收循环的机率。

- 在执行需要平滑运行的代码之前调用 gc.disable()，在其执行之后调用 gc.enable()。这会使 GC 乐于分配更多的内存，而不去运行回收传递(collection pass)。

11.4 自由链表

自由链表是一种加速访问频繁分配和丢弃这种类型数据的好方法。它的思想很简单——当处理守一个对象时，并不真正释放它们，而是将它们放到一个自由链表上。在分配时，直接从自由链表上取用。

```
class Foo
{
    static Foo freelist;                    // 自由链起始

    static Foo allocate()
    {    Foo f;

        if (freelist)
        {    f = freelist;
            freelist = f.next;
        }
        else
            f = new Foo();
        return f;
    }

    static void deallocate(Foo f)
    {
        f.next = freelist;
        freelist = f;
    }

    Foo next;            // 由 FooFreeList 使用
    ...
}

void test()
{
    Foo f = Foo.allocate();
    ...
    Foo.deallocate(f);
}
```

这种自由链表方法很高效。

- 如果用于多线程环境，需要对 allocate() 和 deallocate() 函数进行同步。
- 当采用 allocate() 从自由链表中分配对象时，Foo 的构造函数不会再次运行，所以分配程序需要重新初始化那些需要初始化成员。
- 不一定要为它实现 RAII，因为就算有对象由于抛出异常的原因没有被传递给 deallocate()，最终也会由 GC 回收。

11.5 引用计数

引用计数要求在每个对象内部保留一个计数域。当有引用指向它时，递增引用计数；当指向它的引用消失时，递减引用计数。当引用计数到 0，就删除该对象。

D 不对引用计数提供任何自动支持，引用计数必须由程序员显式地实现。

Win32 COM 编程 使用成员函数 AddRef() 和 Release() 来维护引用计数。

11.6 显式类实例分配

D 提供了对定制对象实例的分配程序和释放程序的方法。通常情况下，对象实例应该分配在垃圾收集堆上，当垃圾收集程序运行时释放无用的对象实例。对于那些特殊情况，可以使用 *New 声明* 和 *Delete 声明*。例如，使用 C 运行时库的 `malloc` 和 `free`：

```
import std.c.stdlib;
import std.outofmemory;
import std.gc;

class Foo
{
    new(size_t sz)
    {
        void* p;

        p = std.c.stdlib.malloc(sz);
        if (!p)
            throw new OutOfMemoryException();
        std.gc.addRange(p, p + sz);
        return p;
    }

    delete(void* p)
    {
        if (p)
        {   std.gc.removeRange(p);
            std.c.stdlib.free(p);
        }
    }
}
```

new() 的主要功能是：

- new() 不能指定返回类型，但返回类型被定义为 void*。new() 必须返回 void*。
- 如果 new() 无法分配内存，不必返回 null，但必须抛出异常。
- new() 返回的指针必须按照默认的堆齐方式对齐。在 win32 系统中这个是 8。
- 当从 Foo 派生的类调用该分配程序时，需要提供参数 *size*，并且应该大于传递给 Foo 的值。
- 如果不能分配所需的存储空间，不会返回 null。而会产生一个异常。产生什么样的异

常由程序员决定，在该情况下，就会产生 OutOfMemory()异常。

- 当扫描内存收集根指针到垃圾收集堆时，会自动扫描静态数据段和堆栈。但不会扫描 C 的堆。因此，如果 Foo 或它的派生类使用的分配程序含有指向由垃圾收集程序分配的对象的引用的话，就需要以某种方式通知 GC 这个事实。可以使用 gc.addRange() 方法达到这个目的。
- 不必初始化内存，因为编译器会自动在调用 new() 之后插入代码将类实例的成员设为它们的默认值，然后运行构造函数（如果有的话）。

delete() 的主要功能是：

- 已经对参数 p 调用了析构函数（如果有的话），所以它指向的数据将被视为垃圾。
- 指针 p 可以为 null。
- 如果使用 gc.addRange() 通知了 GC，就必须在释放程序中调用 gc.removeRange()。
- 如果定义了 delete()，就必须定义对应的 new()。

如果使用类专用的分配程序和释放程序分配内存，就必须小心地编码以避免内存泄露和悬挂引用的出现。如果涉及到异常，还必须实现 RAII 以避免内存泄露。

对于结构和联合，自定义分配器和释放器也已完成。

11.7 标记/释放

标记/释放等价于在堆栈上分配和释放内存。这会在内存中创建一个‘堆栈’。对象的分配仅仅需要向下移动堆栈指针。指针被作了“标记”，释放整个内存区域时只需要简单地将堆栈指针复位到标记点处即可。

```
import std.c.stdlib;
import std.outofmemory;

class Foo
{
    static void[] buffer;
    static int bufindex;
    static const int bufsize = 100;

    static this()
    {   void *p;

        p = malloc(bufsize);
        if (!p)
            throw new OutOfMemoryException;
        std.gc.addRange(p, p + bufsize);
        buffer = p[0 .. bufsize];
    }

    static ~this()
    {
        if (buffer.length)
        {
            std.gc.removeRange(buffer);
            free(buffer);
            buffer = null;
        }
```

```
    }

    new(size_t sz)
    {   void *p;

        p = &buffer[bufindex];
        bufindex += sz;
        if (bufindex > buffer.length)
            throw new OutOfMemory;
        return p;
    }

    delete(void* p)
    {
        assert(0);
    }

    static int mark()
    {
        return bufindex;
    }

    static void release(int i)
    {
        bufindex = i;
    }
}

void test()
{
    int m = Foo.mark();
    Foo f1 = new Foo;           // 分配
    Foo f2 = new Foo;           // 分配
    ...
    Foo.release(m);             // 释放 f1 和 f2
}
```

在分配时，buffer[] 被作为一个区域加入 gc，所以不需要在 Foo.new() 内部再通过一个单独的调用完成此事。

11.8 RAII (资源获得即初始化)

在使用显式内存分配和释放时，RAII 技术在避免内存泄露方面很有用。把 scope 属性 添加到这样的类里会有帮助。

11.9 在堆栈上分配类实例

在堆栈上分配类实例可用来分配在函数退出时需要释放的临时对象。不过，如果它们：

- 在函数里被分配成局部符号
- 使用 **new** 来进行分配
- 使用不带参数的 **new**
- 有 **scope(域)** 存储类别

那么，它们它们就被分配到栈上。这个比在一个实例上进行 allocate/free 循环更有效率。不过要小心，任何对该对象的引用在函数返回后都将不会再存在。

```
class C { ... }

scope c = new C();       // c 被分配到堆栈上
```

如果类有析构函数，那么就可以确保在类对象不在域中(go out of scope)时该析构函数会被运行，即使因为异常退出了该域。

11.10 在堆栈上分配未初始化的数组

在 D 中的数组总是会被初始化。所以，下面的声明：

```
void foo()
{   byte[1024] buffer;

    fillBuffer(buffer);
    ...
}
```

不会运行得像你想象的那样快，因为 buffer[] 总是会被初始化。如果对程序的剖析表明的确是初始化造成了性能问题，则可以通过使用 *VoidInitializer（空初始化器）*来解决：

```
void foo()
{   byte[1024] buffer = void;

    fillBuffer(buffer);
    ...
}
```

在使用堆栈上的未初始化的数据前，需要认真考虑下面这些警告：

- 垃圾回收器会扫描堆栈上的未初始化数据，以查找所有指向已分配内存的引用。因为未初始化数据是以前的 D 堆栈帧所构成的，所以很有可能其中有些东西会被误认为是引用到 GC 堆里，这样该 GC 内存就不会被释放。这种问题确实会发生，并且相当难以跟踪。
- 一个函数可能会把一个指向它的堆栈帧内部数据的引用传递到函数外面。当分配新的堆栈帧正好覆盖了旧的数据，并且新的堆栈帧没有初始化时，指向旧数据的引用看起来仍是有效的。那么程序行为就会变得不定。如果所有堆栈帧上的数据都被初始化的话，将会大大提高此类错漏以可重复方式暴露出来的可能性。
- 就算用法正确，未初始化数据也会是错漏和麻烦的源头。D 的设计目标之一是通过限制源码中的未定义行为来提高可靠性和可移植性，而未初始化数据是未定义、不可移植、错误缠身及不可预测的行为的万恶之源。因此，这个习语(idiom)应该只有在其它速度优化办法都用尽，并且基准测试显示它真的可以加快整体执行速度的情况下使用。

11.11 中断服务例程

在垃圾回收器完成“回收传递(collection pass)”时，它会暂停所有的运行线程，目的是扫描它们的堆栈和寄存器内容，以便找到对 GC 所分配对象的引用。如果 ISR (中断服务例程) 线程被暂停，程序也会被中止。

因此，ISR 线程不应该被暂停。使用 std.thread 函数创建的线程会被暂停。但使用 C 的 `_beginthread()` 或等效的函数创建的线程不会，而 GC 并不知道它们存在。

想要它顺利工作：

- ISR 线程就不能分配任何使用 GC 的内存。即是说不能使用全局的 `new` 语句。不要调整动态数组的大小，不要往关联数组里添加任何元素。任何对 D 运行库的使用都应该需要检查一下，看看分配 GC 内存的各种可能性；或者最好是，让 ISR 根本就不去调用任何 D 运行库函数。
- ISR 并不能持用对任何 GC 分配内存的唯一引用，否则的话，GC 可能在 ISR 还在使用该内存时就将其释放。解决办法是让被暂停线程中的某一个也持有一个对它的引用，或者该线程在全局数据里存储一个对它的引用。

第 12 章 异常安全编程

异常安全编程指的是这样一种编程方法——即使有段可能产生异常的代码真的产生了异常，该程序的状态也不会被损坏(corrupted)，而且资源也不会出现泄漏(leaked)。使用传统的方法来保证这点正确，经常会导致代码复杂、无吸引力和脆弱不堪。最后，异常安全也常常是错误连连或者简简单单忽视了方便所带来的风险。

12.1 示例一

例如，有一个 Mutex 类型的 m，它需要被获取，并且还有一些语句需要它，可是被释放了：

```
void abc()
{
    Mutex m = new Mutex;
    lock(m);     // 锁定该互斥量
    foo();       // 过程处理
    unlock(m);   // 将该互斥量解锁
}
```

如果 foo() 产生了异常，那么 abc() 就会通过展开的(unwinding)的异常退出，这样 unlock(m) 就绝不会获得调用，而 Mutex 去未被释放。这是此段代码最致命的问题。

RAII（资源获得即初始化）习语跟 try-finally 语句成为了编写异常安全程序的传统方法的中坚。

RAII 被限定用于析构，这样上面的例子就可以得到解决，方法就是通过提供一个带有析构函数的 Lock 类，在退出该域时该析构函数可以被调用：

```
class Lock
{
    Mutex m;

    this(Mutex m)
    {
        this.m = m;
        lock(m);
    }

    ~this()
    {
        unlock(m);
    }
}

void abc()
{
    Mutex m = new Mutex;
    scope L = new Lock(m);
    foo();       // 过程处理
}
```

如果 abc() 是正常退出的，或者是通过来自 foo() 的异常退出的，那么 L 的析构函数就会被调用，并且 mutex 也会被解锁。使用 try-finally 来解决这个相同的问题，就象这样：

```
void abc()
{
    Mutex m = new Mutex;
    lock(m);     // 锁定该互斥量
    try
    {
        foo();  // 过程处理
    }
    finally
    {
        unlock(m);       // 将该互斥量解锁
    }
}
```

两种解决方法都有效，不过也都有不足。RAII 解决方法经常要求创建额外的哑元(dummy)类，还得写很多行的代码，或者使用大量混乱不堪的控制流逻辑。这样去管理那些需要被清理以及在程序里仅出现一次的资源是值得的，不过当仅需要一次完成这个操作时，它就乱套了。try-finally 解决的办法是将展开的代码(unwinding code)从 setup 里分隔出来，而这常常又是一个庞大的分离操作。关系紧密的代码应该被组织到一起。

scope exit 语句是个更简单的方法：

```
void abc()
{
    Mutex m = new Mutex;

    lock(m);     // 锁定该互斥量
    scope(exit) unlock(m);       // 在离开该域时解锁

    foo();       // 过程处理
}
```

在正常执行就要结束那一刻，或者因为有异常产生而离开当该域时，“scope(exit)”语句得到了执行。它把那部分“展开代码”放到了恰到好处的地方，紧挨着需要“展开”的状态创建那个地方。相比 RAII 或 try-finally 解决办法，它需要编写的代码更少，而且不要求创建哑元类。

12.2 示例二

下一个例子是关于一个有着众所周知的事务处理(transaction processing)问题的类：

```
Transaction abc()
{
    Foo f;
    Bar b;
```

```
    f = dofoo();
    b = dobar();

    return Transaction(f, b);
}
```

dofoo() 和 dobar() 都必须成功，否则这个事务就会失败。如果该事务失败的话，这些数据都必须被恢得到 dofoo() 跟 dobar() 都还没有产生时的那个状态。要支持这个，dofoo() 就要有一个“展开”操作——dofoo_undo(Foo f)，它可以恢复创建 Foo 时那个状态。

带上 RAII 方法：

```
class FooX
{
    Foo f;
    bool commit;

    this()
    {
        f = dofoo();
    }

    ~this()
    {
        if (!commit)
            dofoo_undo(f);
    }
}

Transaction abc()
{
    scope f = new FooX();
    Bar b = dobar();
    f.commit = true;
    return Transaction(f.f, b);
}
```

带上 the try-finally 方法：

```
Transaction abc()
{
    Foo f;
    Bar b;

    f = dofoo();
    try
    {
        b = dobar();
        return Transaction(f, b);
    }
    catch (Object o)
    {
        dofoo_undo(f);
        throw o;
    }
```

```
}
```

这些也有效，不过还是有着相同的问题。RAII 方法导致了创建哑元类，以及从 abc() 函数中移出一些逻辑的不便。而 try-finally 方法则过于冗长，即使对于这样一个简单的例子也是；如果要求有两个以上的事务都必须成功，试试去编写看看。这个规模无法想像。

scope(failure) 语句解决办法象这个样子：

```
Transaction abc()
{
    Foo f;
    Bar b;

    f = dofoo();
    scope(failure) dofoo_undo(f);

    b = dobar();

    return Transaction(f, b);
}
```

如果通过一个异常退出该域，则仅会执行 dofoo_undo(f)。“展开代码”是最小的，而且也恰到好处。对于更为复杂的事务，它以一种很自然的方式增长。

```
Transaction abc()
{
    Foo f;
    Bar b;
    Def d;

    f = dofoo();
    scope(failure) dofoo_undo(f);

    b = dobar();
    scope(failure) dobar_unwind(b);

    d = dodef();

    return Transaction(f, b, d);
}
```

12.3 示例三

下一个例子是关于临时性更改一些对象的状态。假设有一个类型数据成员 verbose，它控制着那些记录类型动作的信息是否要被显示。在其中的一个方法里面，verbose 需要被返回“关闭”，因为有一个循环，否则会引起大量信息的输出：

```
class Foo
{
    bool verbose;       // true 表示输出信息；false 则代表不输出
```

```
    ...
    bar()
    {
        auto verbose_save = verbose;
        verbose = false;
        ... lots of code ...
        verbose = verbose_save;
    }
}
```

如果通过异常退出 Foo.bar()，就会产生问题—— verbose 标志状态不会被恢复。这个可以简单地使用 scope(exit) 来修复：

```
class Foo
{
    bool verbose;        // true 表示输出信息；false 则代表不输出
    ...
    bar()
    {
        auto verbose_save = verbose;
        verbose = false;
        scope(exit) verbose = verbose_save;

        ...许多代码...
    }
}
```

如果“...□ 多代□ ...”保持在一定长度之内，这个也能简洁地解决了问题；而且在将来维护程序员在里面插入一个返回语句，而不用去理会在退出之前必须重置 verbose。重置代码应该位于概念上所属的地方，而不是它执行的地方（类似的情况就是在 *For 语句* 里的继续表达式）。不管是因为 return、break、goto、continue，还是因为表达式，造成退出该域，这个都有效。

RAII 解决方法试着捕捉 verbose 的错误状态做为资源，一种毫无意义的提取。而 try-finally 解决办法要求在两个在概念上有着联系的设置和重置代码之间进行任意大的分离，另外还需要其它不相关的域。

12.4 示例四

这里又另一个关于多步事务的例子，这次是一个电子邮件程序。发送一封电子邮件需要两个操作：

1. 完成 SMTP 发送操作。
2. 复制该电子邮件到“已发送(Sent)”文件夹，对于 POP 它位于本地磁盘里，而对于 IMAP，也处于远端。

那些实际上没有被发送的信息不应该出现在“已发送(Sent)”里，而已发送的信息却必须出现在“已发送(Sent)”里。

操作(1)不是可恢复的，因为它是众所周知的分布计算问题。操作(2)是可恢复的，并且带有一定程度的可靠性(some degree of reliability)。因此我们将该工作分成三步：

1. 把信息复制到“已发送(Sent)”，同时把标题更改为 "[Sending] <Subject>"。这个操作

要求确保在用户的 IMAP 账号里（或者本地磁盘）有一定的空间，以及连接存在并且有效等等。

2. 通过 SMTP 发送信息。
3. 如果发送失败，就从“已发送(Sent)”里删除该信息。如果信息发送成功，就把标题由 "[Sending] <Subject>" 更改为 "<Subject>"。这两个操作都有很高的成功性。如果该文件夹是本地的，成功的可能性非常高。如果文件夹是远程的，成功的可能性仍然比(1)步要高很多，因为它并没有导致任意大数据的传送。

```
class Mailer
{
   void Send(Message msg)
   {
      {
         char[] origTitle = msg.Title();
         scope(exit) msg.SetTitle(origTitle);
         msg.SetTitle("[Sending] " ~ origTitle);
         Copy(msg, "Sent");
      }
      scope(success) SetTitle(msg.ID(), "Sent", msg.Title);
      scope(failure) Remove(msg.ID(), "Sent");
      SmtpSend(msg);  // 最后做可靠性最低的部分
   }
}
```

这个是对复杂问题很好的解决方法。使用 RAII 来改写这个例子，需要需要两个额外的小类：MessageTitleSaver 和 MessageRemover。而使用 try-finally 来重写这个例子，会需要使用嵌套的 try-finally 语句，或者使用额外的变量来跟踪状态的演变。

12.5 示例五

假设给用户发送回应，内容就是一大堆的操作（鼠标变成一个沙漏、窗口标题变红/变斜体……）。使用 scope(exit) 可以轻松完成，同时不需要去了解手工资源的事，不管用于提示的 UI 状态元素是什么：

```
void LongFunction()
{
   State save = UIElement.GetState();
   scope(exit) UIElement.SetState(save);
...许多代码...
}
```

甚至，scope(success) 和 scope(failure) 可以被用来提示，操作是成功了还是有错误发生：

```
void LongFunction()
{
   State save = UIElement.GetState();
   scope(success) UIElement.SetState(save);
   scope(failure) UIElement.SetState(Failed(save));
```

```
...许多代码...
}
```

12.6 什么时候使用 RAII、try-catch-finally 和 Scope

RAII 是用于管理资源，这个不同于管理状态和事务。try-catch 还是需要的，因为 scope 不能捕捉异常。多余的是 try-finally。

12.7 致谢

Andrei Alexandrescu 在 Usenet（新闻组）上参与讨论过这些构造的可用性，并且在 comp.lang.c++.moderated 的一系列帖子里定义了关于 try/catch/finally 的语义，帖子标题就是：A safer/better C++（更安全/更好的 C++）?，此帖子起始于 Dec 6, 2005。D 语言实现这个想法，整个过程经过了轻微地修改语法，还有创建者对于该功能的实验，以及来自 D 程序员社区的有用建议，尤其是 Dawid Ciezarkiewicz 和 Chris Miller。

我很是感激 Scott Meyers，因为它教会了我异常安全编程。

12.8 参考

1. Generic<Programming>:Change the Way You Write Exception-Safe Code Forever 作者 Andrei Alexandrescu 和 Petru Marginean
2. "Item 29: Strive for exception-safe code" in Effective C++ Third Edition, pg.127 作者 Scott Meyers

第 13 章 再谈模板

作者 Walter Bright, http://www.digitalmars.com/d

“我正告诉你们的那些，正是我们所教给我们的在三或四年后将毕业的正在编程的学生的那些内容。说服你不要因为不理解而逃避它，这是我的任务。你们也知道我的那些正在编程的学生并没有理解它……那是因为我也没有理解它。没人能懂。”
-- Richard Deeman

13.1 摘要

C++ 里的模板发展开始于最初的特征符替换，最后进入到语言本身。C++ 模板的许多有用方面是被发现的，而不是被设计的。这样的副作用就是，C++ 模板因为笨拙的语法、大量令人不可思议的规则以及难于完全地实现，常常受到批评。如果我们退一步，看看模板到底能够做些什么、它们都有什么用途，并且重新设计它们，那么它们看起来会是什么样子呢？模板能够变得强大、给人美感、易于说明而且实现直接吗？本章就来看看在 D 编程语言里的又一种模板设计[1]。

13.2 相似性

- 编译时间语义
- 函数模板
- 类模板
- 类型参数
- 值参数
- 模板参数
- 局部显式特例化
- 类型推演
- 隐式函数模板实例化
- SFINAE (Substitution Failure Is Not An Error - 置换失败不是错误)

13.3 实参语法

要记住的第一件事就是使用“< >”来包括形参列表和实参列表。不过“< >”有几个严重的问题。它们跟操作符 <、> 和 >> 让人容易搞混。这意味着像下面那样的表达式：

```
a<b,c>d;
```

和：

```
a<b<c>>d;
```

在语法上都是含糊不清的，不管是对编译器还是对程序员。如果你在并不熟悉的代码里碰到了 `a<b,c>d;`，面对大量的声明和 `.h`，想判断出它是不是一个模板，将是极艰难。为了处理这个，程序员、编写编译器的人以及编写语言标准的人已经花费掉了多少努力呀？

会有更好方式的。通过表明“!”不是被用作二进制操作符，D 解决了这个问题，因此把：

```
a<b,c>
```

替换为：

```
a!(b,c)
```

这样在句法上就不再含糊不清了。由此使得它易于解析、易于生成合理的错误信息，并且在检查代码以确定“是的，a 一定是个模板。”时，变得更为容易。

13.4 模板定义语法

C++ 可以定义两种广义类型(broad type)的模板：类模板和函数模板。每一种模板都是独立地编写，即使它们的关系紧密：

```
template<class T, class U> class Bar { ... };

template<class T, class U> T foo(T t, U u) { ... }

template<class T, class U> static T abc;
```

POD（Plain Old Data——无格式的旧数据，跟 C 中的格式一样）结构将相关的数据声明放置到一起；而类则把相关的数据和函数声明放置到一起，但却没有什么东西用来把要被实例化到一块的模板逻辑组织到一起。在 D 里，我们可以编写：

```
template Foo(T, U)
{
  class Bar { ... }

  T foo(T t, U u) { ... }

  T abc;

  typedef T* Footype;    // 任何声明都可以被模板化
}
```

Foo 为这个要被访问的模板组成了一个名字空间，例如：

```
Foo!(int,char).Bar b;
Foo!(int,char).foo(1,2);
Foo!(int,char).abc = 3;
```

当然，这样可能有点冗长乏味，因此我们可以对某个特定的实例化使用别名：

```
alias Foo!(int,char) f;
f.Bar b;
f.foo(1,2);
f.abc = 3;
```

对于类模板，语法甚至更为简单。 像这样定义了一个类：

```
class Abc
{
    int t;
    ...
}
```

于是仅通过添加一个形参列表，它就变成了一个模板：

```
class Abc(T)
{
    T t;
    ...
}
```

13.5 模板声明、定义和导出

C++ 模板可以有模板声明、模板定义和导出模板三种形式。因为 D 有着真正模块系统，而不是原文式的 #include 文件，因此在 D 里就只有模板定义一种形式。对于模板的声明和导出，都跟 D 不相关的。例如，在模块 A 里有一个模板定义：

```
module A;

template Foo(T)
{
    T bar;
}
```

它可以像这样，从模块 B 里进行访问：

```
module B;

import A;

void test()
{
    A.Foo!(int).bar = 3;
}
```

同样，也可以使用别名来使访问简单些：

```
module B;

import A;
alias A.Foo!(int).bar bar;

void test()
{
    bar = 3;
}
```

13.6 模板参数

C++ 模板参数可以是：

- 类型

- 整数值
- 静态/全局地址
- 模板名

D 模板参数可以是：

- 类型
- 整数值
- 浮点值
- 字符串
- 模板
- 或者任何符号

每一个都可以有默认值，而且类型参数都可以带有（一种限定形式的）约束(constraint)：

```
class B { ... }
interface I { ... }

class Foo(
R,              // R 可以是任何类型
  P:P*,           // P 必须是指针类型
  T:int,          // T 必须是 int 类型
  S:T*,           // S 必须是指向 T 的指针
  C:B,            // C 必须属于类 B 或者是
                // 派生自 B
  U:I,            // U 必须是这样一个类
                // 实现了接口 I 的类
  char[] string = "hello",
// 字符串,
                // 默认是 "hello"
  alias A = B   // A is any symbol
                // (包括模板符号),
                // 默认是 B
  )
{
   ...
}
```

13.7 特例化(Specialization)

局部显式特例化基本上跟 C++ 里的一样，不同的是没有“primary（初级）”模板的概念。所有具有相同名字的模板会在模板实例化时被检查，其形参跟实参最为吻合的那个模板才会被实例化。

```
template Foo(T) ...
template Foo(T:T*) ...
template Foo(T, U:T) ...
template Foo(T, U) ...
template Foo(T, U:int) ...
```

```
Foo!(long)        // 选中 Foo(T)
Foo!(long[])      // 选中 Foo(T), T 是 long[]
Foo!(int*)        // 选中 Foo(T*), 而 T 为 int
Foo!(long,long)   // 选中 Foo(T, T)
Foo!(long,short)  // 选中 Foo(T, U)
Foo!(long,int)    // 选中 Foo(T, U:int)
Foo!(int,int)     // 有歧义：是选 Foo(T, U:T)
                  // 还是选 Foo(T, U:int)
```

13.8 两级名字查询

C++ 有一些非常规的规则用于模板内的名字查询，例如，不去查看基类、不允许在限定域内重新声明模板参数名、不考虑发生在定义点之后的重载（下面的例子就来自 C++98 标准)：

```
int g(double d) { return 1; }

typedef double A;

template<class T> B
{
  typedef int A;
};

template<class T> struct X : B<T>
{
A a;               // a 是 double 类型
int T;             // 错误，T 重得声明了
  int foo()
{  char T;         // 错误，T 重得声明了
return g(1);    // 总是返回 1
  }
};

int g(int i) { return 2; }       // 此定义 X 已看不到了
```

在 D 里，限定域内的名字查询规则跟其它情况没什么两样：

```
int g(double d) { return 1; }

typedef double A;

class B(T)
{
  typedef int A;
}

class X(T) : B!(T)
{
A a;               // a 的类型是 int
  int T;             // 正确，T 被重新声明为 int
  int foo()
{   char T;        // 正确，T 被重新声明为 char
    return g(1);   // 总是返回 2
  }
```

```
};

int g(int i) { return 2; }       // 函数可以被向前引用
```

13.9 模板递归

带有特例化的模板递归，意味着 C++ 模板实际上构成了一种编程语言，尽管有些怪异。假设有一套模板，用于在运行时计算阶乘。类似于 "hello world" 程序那样，阶乘也是模板元编程的标准例子：

```
template<int n> class factorial
{
  public:
    enum
    {
      result = n * factorial<n - 1>::result
    };
};

template<> class factorial<1>
{
  public:
    enum { result = 1 };
};

void test()
{
// 输出 24
  printf("%d\n", factorial<4>::result);
}
```

在 D 里，这种递归也一样工作，而且敲打的内容要少很多：

```
template factorial(int n)
{
  const factorial = n * factorial!(n-1);
}

template factorial(int n : 1)
{
  const factorial = 1;
}

void test()
{
writefln(factorial!(4));  // 输出 24
}
```

也可以通过使用 `static if` 来构造它，只需要在一个模板里面就可以：

```
template factorial(int n)
```

```
{
  static if (n == 1)
    const factorial = 1;
  else
    const factorial = n * factorial!(n-1);
}
```

减少了 13 行代码，变成了更为清晰的 7 行。`static if` 等同于 C++ 的 `#if`。但 `#if` 不能访问模板参数，因此所有的模板条件编译必须使用局部显式特例化来处理。`static if` 使这种构造变得简单多了。

D 可以让这个甚至更简单。值生成模板（像阶乘那样的）是可能的，只是它更简单，只需要编写一个在编译时可以被计算的函数就可以了：

```
int factorial(int n)
{
  if (n == 1)
    return 1;
  else
    return n * factorial(n - 1);
}

static int x = factorial(5);  // x 被静态初始化为 120
```

13.10 SFINAE (Substitution Failure Is Not An Error - 置换失败不是错误)

这个用于确定模板参数类型是否是一个函数，源自“*C++ Templates: The Complete Guide*", Vandevoorde & Josuttis pg. 353:

```
template<U> class IsFunctionT
{
  private:
    typedef char One;
    typedef struct { char a[2]; } Two;
    template static One test(...);
    template static Two test(U (*)[1]);
  public:
    enum {
      Yes = sizeof(IsFunctionT::test(0)) == 1
    };
};

void test()
{
  typedef int (fp)(int);

  assert(IsFunctionT<fp>::Yes == 1);
}
```

模板 `IsFunctionT` 依赖于两个副作用(side effect)得到最后结果。首先，它依赖属于非法 C++ 类型的函数数组。这样，如果 `U` 是一个函数类型，那么第二个 `test` 就不会被选择，

因为那样做的话会引起一个错误(SFINAE (置换失败不是错误))。因此，test 会被选择。如果 U 不是一个函数类型，那么第二个 test 就更为适合 .... 其次，它将决定选择哪个 test，方式就是通过检查返回值的大小，例如使用 sizeof(One) 或 sizeof(Two)。不幸的是，C++ 里的模板元编程似乎常常只依赖于一种副作用，且无法清楚地对所期望的进行编码。

在 D 里，这个可以被写成：

```
template IsFunctionT(T)
{
  static if ( is(T[]) )
    const int IsFunctionT = 0;
  else
    const int IsFunctionT = 1;
}

void test()
{
  alias int fp(int);

  assert(IsFunctionT!(fp) == 1);
}
```

is(T[]) 等同于 SFINAE (置换失败不是错误)。它会试着创建一个数组 T，而如果 T 函数类型，那么它就是函数数组。因此这个是非法类型，T[] 跟 is(T[]) 失败都会返回 false。

虽然可以使用 SFINAE (置换失败不是错误)，但 is 表达式是可以直接测试类型的，因此使用一个模板来询问关于类型的问题是完全没有必要的：

```
void test()
{
  alias int fp(int);

  assert( is(fp == function) );
}
```

13.11 带浮点的模板元编程

让我们转移到那些在 C++ 里无法实现的事情上去。例如，这样一个模板，使用 Babylonian 方法返回实数 x 的平方根：

```
import std.stdio;

template sqrt(real x, real root = x/2, int ntries = 0)
{
  static if (ntries == 5)
// 对于每一次迭代，精确度翻倍，
    // 5 次足够了
    const sqrt = root;
```

```
  else static if (root * root - x == 0)
const sqrt = root;  // 精确匹配
  else
// 再次迭代
    const sqrt = sqrt!(x, (root+x/root)/2, ntries+1);
}

void main()
{
    real x = sqrt!(2);
    writefln("%.20g", x); // 1.4142135623730950487
}
```

在一些运行时的浮点计算里，例如计算 gamma(伽玛) 函数，由于要考虑速度的原因，单独求解平方根是经常需要的。这些模板浮点算法不需要跟它们在运行时那样有效率，需要的是它们必须正确。

更多更为复杂的模板也要构造，Don Clugston 已经编写了一个用于在运行时计算 π 的模板。[2]

再一次，我们可以只使用一个在编译时可以被计算的函数来完成这个：

```
real sqrt(real x)
{
    real root = x / 2;
    for (int ntries = 0; ntries < 5; ntries++)
    {
        if (root * root - x == 0)
            break;
        root = (root + x / root) / 2;
    }
    return root;
}
static y = sqrt(10);    // y 被静态初始化为 3.16228
```

13.12 带字符串的模板元编程

带上字符串甚至可以完成更为有意义的事。这个实例完成的是在编译时把一个整数转换成一个字符串：

```
template decimalDigit(int n)    // [3]
{
  const char[] decimalDigit = "0123456789"[n..n+1];
}

template itoa(long n)
{
  static if (n < 0)
    const char[] itoa = "-" ~ itoa!(-n);
  else static if (n < 10)
    const char[] itoa = decimalDigit!(n);
  else
    const char[] itoa = itoa!(n/10L) ~ decimalDigit!(n%10L);
}
```

```
char[] foo()
{
return itoa!(264);   // 返回 "264"
}
```

该模板会计算字符串文件的 hash(哈稀) 值：

```
template hash(char [] s, uint sofar=0)
{
   static if (s.length == 0)
      const hash = sofar;
   else
      const hash = hash!(s[1 .. length], sofar * 11 + s[0]);
}

uint foo()
{
    return hash!("hello world");
}
```

13.13 正则表达式编译器

对于其它更具有意义的事物，象正则表达式编译器，D 模板又能怎么样应对呢？Eric Niebler 依靠 C++ 的表达式模板能力为其编写了一个。[4] 使用表达式模板就的问题在于，我们局限在仅能使用 C++ 操作符语法以及比较关系(precedence)。因此，使用表达式模板的正则表达式看起来并不像正则表达式，它更多是像 C++ 表达式。Eric Anderton 依靠 D 够解析字符串的模板能力为其写了一个。[5] 这就表明，在字符串内，我们可以使用所期望的正则表达式语法和操作符。

正则表达式编译器通过解析正则字符串参数，从前端一个接一个地获取特征符(token)，接着为每一个特征符谓词实例化自定义模板函数，最后将它们组合成一个可以直接实现正则表达式的函数。对于有语法错误的表达式，它甚至会给出合理的错误信息。

带上用于匹配的字符串参数去调用该函数，就会得到一个匹配字符串的数组：

```
import std.stdio;
import regex;

void main()
{
    auto exp = &regexMatch!(r"[a-z]*\s*\w*");
    writefln("matches: %s", exp("hello    world"));
}
```

接下来的内容是 Eric Anderton 的正则表达式编译器的删剪版本。不过它也足够编译上面的正则表达式，并说明它是怎么样完成的。

```
module regex;
```

```
const int testFail = -1;

/**
* 编译 pattern[] 并扩展成一个自主生成的
* 函数，此函数会接收字符串 str[] 并且应用给它的
* 正则表达式，返回匹配后的数组。
 */

template regexMatch(char[] pattern)
{
  char[][] regexMatch(char[] str)
  {
    char[][] results;
    int n = regexCompile!(pattern).fn(str);
    if (n != testFail && n > 0)
      results ~= str[0..n];
    return results;
  }
}

/*****************************
* 函数 testXxxx() 是由模板自主生成的
* 用来匹配正则表达式的每一个谓词。
 *
 * Params:
 *       char[] str       用于被匹配的输入串
 *
 * Returns:
 *       testFail         没有匹配的
 *       n >= 0           匹配到 n 个字符
 */

// 总是匹配
template testEmpty()
{
  int testEmpty(char[] str) { return 0; }
}

/// 如果 testFirst(str) 跟 testSecond(str) 匹配，则结果为匹配
template testUnion(alias testFirst, alias testSecond)
{
  int testUnion(char[] str)
  {
    int n1 = testFirst(str);
    if (n1 != testFail)
    {
      int n2 = testSecond(str[n1 .. $]);
      if (n2 != testFail)
        return n1 + n2;
    }
    return testFail;
  }
}

/// 如果 str[] 的第一部分匹配 text[]，则结果为匹配
```

```
template testText(char[] text)
{
  int testText(char[] str)
  {
    if (str.length &&

        text.length <= str.length &&

        str[0..text.length] == text
       )
      return text.length;
    return testFail;
  }
}

/// testPredicate(str) 匹配 0 次或更多次，则结果为匹配
template testZeroOrMore(alias testPredicate)
{
  int testZeroOrMore(char[] str)
  {
    if (str.length == 0)
      return 0;
    int n = testPredicate(str);
    if (n != testFail)
    {
      int n2 = testZeroOrMore!(testPredicate)(str[n .. $]);
      if (n2 != testFail)
        return n + n2;
      return n;
    }
    return 0;
  }
}

/// 如果 term1[0] <= str[0] <= term2[0]，则结果为匹配
template testRange(char[] term1, char[] term2)
{
  int testRange(char[] str)
  {
    if (str.length && str[0] >= term1[0]
                   && str[0] <= term2[0])
      return 1;
    return testFail;
  }
}

/// 如果 ch[0]==str[0]，则结果为匹配
template testChar(char[] ch)
{
  int testChar(char[] str)
```

```
  {
    if (str.length && str[0] == ch[0])
      return 1;
    return testFail;
  }
}

/// 如果 str[0] 是一个字(word)字符，则结果为匹配
template testWordChar()
{
  int testWordChar(char[] str)
  {
    if (str.length &&

        (
         (str[0] >= 'a' && str[0] <= 'z') ||
         (str[0] >= 'A' && str[0] <= 'Z') ||
         (str[0] >= '0' && str[0] <= '9') ||
         str[0] == '_'
        )
       )
    {
      return 1;
    }
    return testFail;
  }
}

/*****************************************************/

/**
* 返回 pattern[] 的前面部分，直到
* 结尾或特殊字符。
 */

template parseTextToken(char[] pattern)
{
  static if (pattern.length > 0)
  {
    static if (isSpecial!(pattern))
      const char[] parseTextToken = "";
    else
      const char[] parseTextToken =
          pattern[0..1] ~ parseTextToken!(pattern[1..$]);
  }
  else
    const char[] parseTextToken="";
}

/**
* 解析 pattern[] 直到包含的终止符。
 * Returns:
 *      token[]          终止符之前的所有内容。
 *      consumed         在 pattern[] 里被解析的字符数目
```

```
 */
template parseUntil(char[] pattern,char terminator,bool fuzzy=false)
{
  static if (pattern.length > 0)
  {
    static if (pattern[0] == '\\')
    {
      static if (pattern.length > 1)
      {
        const char[] nextSlice = pattern[2 .. $];
        alias parseUntil!(nextSlice,terminator,fuzzy) next;
        const char[] token = pattern[0 .. 2] ~ next.token;
        const uint consumed = next.consumed+2;
      }
      else
      {
        pragma(msg,"Error: expected character to follow \\");
        static assert(false);
      }
    }
    else static if (pattern[0] == terminator)
    {
      const char[] token="";
      const uint consumed = 1;
    }
    else
    {
      const char[] nextSlice = pattern[1 .. $];
      alias parseUntil!(nextSlice,terminator,fuzzy) next;
      const char[] token = pattern[0..1] ~ next.token;
      const uint consumed = next.consumed+1;
    }
  }
  else static if (fuzzy)
  {
    const char[] token = "";
    const uint consumed = 0;
  }
  else
  {
    pragma(msg,"Error: expected " ~
              terminator ~
              " to terminate group expression");
    static assert(false);
  }
}

/**
```

```
* 解析 character 类的内容。
 * Params:
*   pattern[] = 要编译的剩余模式
* 输出:
*   fn        = 生成的函数
*   consumed = 在 pattern[] 里被解析的字符数目
 */

template regexCompileCharClass2(char[] pattern)
{
  static if (pattern.length > 0)
  {
    static if (pattern.length > 1)
    {
      static if (pattern[1] == '-')
      {
        static if (pattern.length > 2)
        {
          alias testRange!(pattern[0..1], pattern[2..3]) termFn;
          const uint thisConsumed = 3;
          const char[] remaining = pattern[3 .. $];
        }
        else // length is 2
        {
          pragma(msg,
            "Error: expected char following '-' in char class");
          static assert(false);
        }
      }
      else // not '-'
      {
        alias testChar!(pattern[0..1]) termFn;
        const uint thisConsumed = 1;
        const char[] remaining = pattern[1 .. $];
      }
    }
    else
    {
      alias testChar!(pattern[0..1]) termFn;
      const uint thisConsumed = 1;
      const char[] remaining = pattern[1 .. $];
    }
    alias regexCompileCharClassRecurse!(termFn,remaining) recurse;
    alias recurse.fn fn;
    const uint consumed = recurse.consumed + thisConsumed;
  }
  else
  {
    alias testEmpty!() fn;
    const uint consumed = 0;
  }
}
```

```
/**
* 用来递归解析 character 类。
 * Params:
*  termFn = 此点之前生成的函数
*   pattern[] = 要编译的剩余模式
* 输出:
*  fn = 生成函数，包含有 termFn 和
*       被解析的 character 类
*   consumed = 在 pattern[] 里被解析的字符数目
 */

template regexCompileCharClassRecurse(alias termFn,char[] pattern)
{
  static if (pattern.length > 0 && pattern[0] != ']')
  {
    alias regexCompileCharClass2!(pattern) next;
    alias testOr!(termFn,next.fn,pattern) fn;
    const uint consumed = next.consumed;
  }
  else
  {
    alias termFn fn;
    const uint consumed = 0;
  }
}

/**
* character 类开开始。编译它。
 * Params:
*   pattern[] = 要编译的剩余模式
* 输出:
*   fn       = 生成的函数
*   consumed = 在 pattern[] 里被解析的字符数目
 */

template regexCompileCharClass(char[] pattern)
{
  static if (pattern.length > 0)
  {
    static if (pattern[0] == ']')
    {
      alias testEmpty!() fn;
      const uint consumed = 0;
    }
    else
    {
      alias regexCompileCharClass2!(pattern) charClass;
      alias charClass.fn fn;
```

```
    const uint consumed = charClass.consumed;
   }
  }
  else
  {
   pragma(msg,"Error: expected closing ']' for character class");
   static assert(false);
  }
}

/**
* 查找并解析 '*' 后缀。
 * Params:
*  test = 编译正则表达式到此点之前的函数
*   pattern[] = 要编译的剩余模式
* 输出:
*   fn       = 生成的函数
*   consumed = 在 pattern[] 里被解析的字符数目
 */

template regexCompilePredicate(alias test, char[] pattern)
{
  static if (pattern.length > 0 && pattern[0] == '*')
  {
   alias testZeroOrMore!(test) fn;
   const uint consumed = 1;
  }
  else
  {
   alias test fn;
   const uint consumed = 0;
  }
}

/**
* 解析转义序列符(escape sequence)。
 * Params:
*   pattern[] = 要编译的剩余模式
* 输出:
*   fn       = 生成的函数
*   consumed = 在 pattern[] 里被解析的字符数目
 */

template regexCompileEscape(char[] pattern)
{
  static if (pattern.length > 0)
  {
   static if (pattern[0] == 's')
   {
// 空白字符
    alias testRange!("\x00","\x20") fn;
   }
   else static if (pattern[0] == 'w')
   {
```

```
// 字(word)字符
     alias testWordChar!() fn;
    }
    else
    {
     alias testChar!(pattern[0 .. 1]) fn;
    }
    const uint consumed = 1;
  }
  else
  {
   pragma(msg,"Error: expected char following '\\'");
   static assert(false);
  }
}

/**
* 解析并编译由 pattern[] 所代表的正则表达式。
 * Params:
*   pattern[] = 要编译的剩余模式
* 输出:
*   fn        = 生成的函数
 */

template regexCompile(char[] pattern)
{
  static if (pattern.length > 0)
  {
    static if (pattern[0] == '[')
    {
     const char[] charClassToken =
        parseUntil!(pattern[1 .. $],']').token;
     alias regexCompileCharClass!(charClassToken) charClass;
     const char[] token = pattern[0 .. charClass.consumed+2];
     const char[] next = pattern[charClass.consumed+2 .. $];
     alias charClass.fn test;
    }
    else static if (pattern[0] == '\\')
    {
     alias regexCompileEscape!(pattern[1..pattern.length]) escapeSequence;
     const char[] token = pattern[0 .. escapeSequence.consumed+1];
     const char[] next =
        pattern[escapeSequence.consumed+1 .. $];
     alias escapeSequence.fn test;
    }
    else
    {
     const char[] token = parseTextToken!(pattern);
```

```
    static assert(token.length > 0);
    const char[] next = pattern[token.length .. $];
    alias testText!(token) test;
   }

   alias regexCompilePredicate!(test, next) term;
   const char[] remaining = next[term.consumed .. next.length];

   alias regexCompileRecurse!(term,remaining).fn fn;
  }
  else
   alias testEmpty!() fn;
}

template regexCompileRecurse(alias term,char[] pattern)
{
  static if (pattern.length > 0)
  {
   alias regexCompile!(pattern) next;
   alias testUnion!(term.fn, next.fn) fn;
  }
  else
   alias term.fn fn;
}

/// 用于解析的实用函数
template isSpecial(char[] pattern)
{
  static if (
   pattern[0] == '*' ||
   pattern[0] == '+' ||
   pattern[0] == '?' ||
   pattern[0] == '.' ||
   pattern[0] == '[' ||
   pattern[0] == '{' ||
   pattern[0] == '(' ||
   pattern[0] == ')' ||
   pattern[0] == '$' ||
   pattern[0] == '^' ||
   pattern[0] == '\\'
  )
   const isSpecial = true;
  else
   const isSpecial = false;
}
```

13.14 更多模板元编程

1. Tomasz Stachowiak 的编译时间 raytracer。
2. Don Clugston 的编译时间 99 Bottles of Beer。

13.15 参考

[1] D 编程语言，请看 http://www.digitalmars.com/d/

[2] Don Clugston 的 π 计算器，请看
http://trac.dsource.org/projects/ddl/browser/trunk/meta/demo/calcpi.d

[3] Don Clugston 的 decimaldigit 和 itoa，请看
http://trac.dsource.org/projects/ddl/browser/trunk/meta/conv.d

[4] Eric Niebler 的 Boost.Xpressive 正则表达式模板库 位于 http://boost-sandbox.sourceforge.net/libs/xpressive/doc/html/index.html

[5] Eric Anderton 的针对 D 的正则表达式模板库 位于
http://trac.dsource.org/projects/ddl/browser/trunk/meta/regex.d

13.16 致谢

我要真诚地感谢 Don Clugston、Eric Anderton 以及 Matthew Wilson 给以的灵感和帮助。

第 14 章 元组(Tuples)

一个元组(tuple)指的是一种元素序列。这些元素可以是类型、表达式、别名。元组的数目和元素在编译时被固定；它们在运行时都不能被更改。

元组拥有结构和数组两者的特性。像结构那样，元组的元素可以属于不同的类型。像数组那样，元组的元素也可以通过索引来访问。

那么，如何构造一个元组呢？实际上并没有什么元组字法规范。不过，因为 variadic（个数不定）型模板参数能够建立元组，因此我们可以通过定义一个模板来创建它：

```
template Tuple(E...)
{
    alias E Tuple;
}
```

并且，像这样使用它：

```
Tuple!(int, long, float)         // 创建一个带有 3 种类型元素的元组
Tuple!(3, 7, 'c')                // 创建一个带有 3 个表达式元素的元组
Tuple!(int, 8)                   // 创建一个带有 1 种类型和 1 个表达式两个元素的元组
```

为了方便引用某个元组，我们可以使用别名：

```
alias Tuple!(float, float, 3) TP; // TP 现在就是一个带有两个浮点和常数 3 的元组
```

元组可以被当作模板的参数使用，只要它们被“压平(flatten)”后可以放进参数列表。这使得添加一个新元素到一个已有的元组里，或都连接两个元组更为直接：

```
alias Tuple!(TP, 8) TR;  // TR 现在是 float,float,3,8
alias Tuple!(TP, TP) TS; // TS 是 float,float,3,float,float,3
```

元组使用了数组的许多特性。对于初学者，在一个元组里的元素数目可以使用 **.length** 特性获得：

```
TP.length        // 计算得 3
```

元组也可以被索引：

```
TP[1] f = TP[2];         // f 被声明成 float，并且被初始化成 3
```

甚至可以被分割(slice)：

```
alias TP[0..length-1] TQ; // TQ 现在等同于 Tuple!(float, float)
```

的确，**length** 被定义在 [ ] 里。并且有一个限制：用于索引和分割的索引值必须在编译时被计算出来。

```
void foo(int i)
{
    TQ[i] x;             // 错误，i 不是常数
}
```

这些使得生成一个元组的“头(head)”和“尾(tail)”非常简单。头就是 TP[0]，而尾就是 TP[1 .. length]。有了头和尾，再加入一点点的条件编译，那么我们就可以使用模板来实现一些经典的递归算法。例如，这样一个模板，它返回一个元组，而该元组则由带尾(trailing)类型参数 *TL* 组成，同时移除了第一个类型参数 *T*：

```
template Erase(T, TL...)
{
    static if (TL.length == 0)
        // 0 长度的元组，返回它自己
        alias TL Erase;
    else static if (is(T == TL[0]))
        // 跟元组的第一个元素匹配，返回尾部
        alias TL[1 .. length] Erase;
    else
        // 不匹配，返回的操作是将头部连接到递归得到的尾部
        alias Tuple!(TL[0], Erase!(T, TL[1 .. length])) Erase;
}
```

14.1.1 类型元组(Type Tuples)

如果一个元组的元素仅仅是类型，那么它就叫做 *类型元组*（有时也叫类型列表）。因为函数参数列表就是一个类型列表，因此元组类型可以从它们得里获得。有一种方式就是使用一个 *Is 表达式*：

```
int foo(int x, long y);

...
static if (is(foo P == function))
    alias P TP;
// TP 现在等同于 Tuple!(int, long)
```

这个在模板 std.traits.ParameterTypeTuple 里被泛化：

```
import std.traits;

...
alias ParameterTypeTuple!(foo) TP;      // TP 现在是 tuple (int, long)
```

类型元组 可以被用来声明函数：

```
float bar(TP);  // 等同于 float bar(int, long)
```

如果正在完成隐式函数模板实例化，那么表示参数类型的元组类型就可以被推导：

```
int foo(int x, long y);

void Bar(R, P...)(R function(P))
{
    writefln("return type is ", typeid(R));
    writefln("parameter types are ", typeid(P));
```

```
}

...
Bar(&foo);
```

输出

```
return type is int
parameter types are (int,long)
```

类型推导可以被用来创建一个带有任意数目和类型参数的函数：

```
void Abc(P...)(P p)
{
    writefln("parameter types are ", typeid(P));
}

Abc(3, 7L, 6.8);
```

输出

```
parameter types are (int,long,double)
```

关于此方面更多的信息，请看 Variadic 型模板。

14.1.2 表达式元组(Expression Tuples)

如果一个元组的元素仅仅是表达式，那么它就可以称做 *表达式元组*。可以使用元组模板来进行创建：

```
alias Tuple!(3, 7L, 6.8) ET;

...
writefln(ET);             // 输出 376.8
writefln(ET[1]);          // 输出 7
writefln(ET[1..length]); // 输出 76.8
```

它可以用来创建数组数据：

```
alias Tuple!(3, 7, 6) AT;

...
int[] a = [AT];          // 等同于 [3,7,6]
```

结构或类的数据域可以通过使用 **.tupleof** 特性转换成一个元组表达式：

```
struct S { int x; long y; }

void foo(int a, long b)
{
    writefln(a, b);
}

...
S s;
s.x = 7;
s.y = 8;
foo(s.x, s.y);  // 输出 78
```

```
foo(s.tupleof); // 输出 78
s.tupleof[1] = 9;
s.tupleof[0] = 10;
foo(s.tupleof); // 输出 109
s.tupleof[2] = 11;  // 错误，没有 S 的第三个段域(field)
```

通过使用 **typeof**，可以从结构的数据段创建一个类型元组：

```
writefln(typeid(typeof(S.tupleof)));     // 输出 (int,long)
```

这个封装在模板 std.traits.FieldTypeTuple 里面。

14.1.3 循环

由于“头-尾”式的函数编程方式对于元组也是有效的，因此这使得使用循环更为方便。*Foreach 语句* 既可以用于 *类型元组* 也可以用于 *表达式元组*。

```
alias Tuple!(int, long, float) TL;
foreach (i, T; TL)
   writefln("TL[%d] = ", i, typeid(T));

alias Tuple!(3, 7L, 6.8) ET;
foreach (i, E; ET)
   writefln("ET[%d] = ", i, E);
```

输出

```
TL[0] = int
TL[1] = long
TL[2] = float
ET[0] = 3
ET[1] = 7
ET[2] = 6.8
```

14.1.4 元组声明

使用 *类型元组* 声明的变量就变成了 *表达式元组*：

```
alias Tuple!(int, long) TL;

void foo(TL tl)
{
   writefln(tl, tl[1]);
}

foo(1, 6L);      // 输出 166
```

14.1.5 将它们组合到一起

这些能力可以被放置到一起来实现这样一个模板：它能封装函数的参数，返回使用这些参数

调用该函数的委托。

```
import std.stdio;

R delegate() CurryAll(Dummy=void, R, U...)(R function(U) dg, U args)
{
    struct Foo
    {
        typeof(dg) dg_m;
        U args_m;

        R bar()
        {
            return dg_m(args_m);
        }
    }

    Foo* f = new Foo;
    f.dg_m = dg;
    foreach (i, arg; args)
        f.args_m[i] = arg;
    return &f.bar;
}

R delegate() CurryAll(R, U...)(R delegate(U) dg, U args)
{
    struct Foo
    {
        typeof(dg) dg_m;
        U args_m;

        R bar()
        {
            return dg_m(args_m);
        }
    }

    Foo* f = new Foo;
    f.dg_m = dg;
    foreach (i, arg; args)
        f.args_m[i] = arg;
    return &f.bar;
}


void main()
{
    static int plus(int x, int y, int z)
    {
        return x + y + z;
    }

    auto plus_two = CurryAll(&plus, 2, 3, 4);
    writefln("%d", plus_two());
    assert(plus_two() == 9);
```

```
    int minus(int x, int y, int z)
    {
        return x + y + z;
    }

    auto minus_two = CurryAll(&minus, 7, 8, 9);
    writefln("%d", minus_two());
    assert(minus_two() == 24);
}
```

使用参数 *Dummy* 的原因是我们不能重载带有两个相同参数列表的模板。因此我们通过给当中的某一个带上 dummy 参数，以示区别。

14.1.6 未来的方向

- 从函数里返回元组。
- 让元组可以使用像 =、 += 等等那样的操作符。
- 让元组有像 **.init** 那样的特性，以便将该属性传递给每一个元组成员。

第 15 章 C 语言接口

D 的设计就是要在目标系统上能够很好地符合于 C 编译器。D 通过依赖于目标环境上的 C 运行库，来弥补没有自己的 VM(虚拟机) 的不足。将大量现有的 C 的 API 移植到 D 或者用 D 来进行封装，并没有多大意义。直接调用它们不是要方便得多。

如果采用跟 C 编译器相同的数据类型、内存分布(layout)和函数调用/返回序列，那么它就可以实现。

15.1 调用 C 函数

D 可以直接调用 C 函数。不需要封装函数、参数变换，同时也不需要将 C 函数放到单独的 DLL 中。

C 函数必须被声明，而且必须指定调用协定，大部份情况下为 "C" 调用约定，例如：

```
extern (C) int strcmp(char* string1, char* string2);
```

然后就可以在 D 代码中很自然的调用它们了：

```
import std.string;
int myDfunction(char[] s)
{
    return strcmp(std.string.toStringz(s), "foo");
}
```

需要注意以下几点：

- D 知道 C 函数名的变换规则及函数的调用/返回指令序列。
- C 函数不能被另一个同名的 C 函数重载。
- D 中没有如 __cdecl、__far、__stdcall、__declspec 或者如此之类的 C 类型修饰符。这个可以使用一些特性，如 extern (C)，来进行处理。
- D 中没有 const 或 volatile 类型修饰符。想要声明那些使用了这类类型修饰符的 C 函数，则仅需要从声明中移除这些关键字即可。
- D 中的字符串不是以 0 结尾的。关于此点的更多信息，请参见“数据类型兼容性”。但是，D 里的字符串文字是以 0 结尾的。

同样地，C 代码也可以调用 D 函数，只要 D 函数采用了同 C 编译器兼容的特性，多数情况是 extern (C)：

```
// myfunc() 可以被任何 C 函数调用
extern (C)
{
    void myfunc(int a, int b)
    {
        ...
    }
}
```

15.2 存储分配

C 代码通过调用 malloc() 和 free() 显示地管理内存。D 使用 D 垃圾收集程序分配内存，所以不需要显式地释放内存。

D 仍然可以使用 c.stdlib.malloc() 和 c.stdlib.free() 显式地分配和释放内存，可用于链接那些需要 malloc 缓冲的 C 函数之类的情况。

如果指向来即收集程序分配的内存的指针被传递给 C 函数，必须确保这份内存不会在使用它的 C 函数退出之前被内存收集程序回收。要达到这个目的，可以：

- 使用 c.stdlib.malloc() 创建一份数据的副本，把该副本传递给 C 函数。
- 将指针放在堆栈上（作为参数或者自动变量），因为垃圾收集程序会扫描堆栈。
- 将指向它的指针放在静态数据段，因为垃圾收集程序会扫描静态数据段。
- 使用 gc.addRoot() 或者 gc.addRange() 将指针交由垃圾收集程序管理。

就算是指向所分配内存块内部的指针也足以让 GC 知道对象正在使用；也就是，不一定要需要维护指向所分配内存开始处的指针。

垃圾收集程序不会扫描不是通过 D Thread 接口创建的线程的堆栈。也不会扫描其它 DLL 中的数据段等。

15.3 数据类型兼容性

D 类型	C 类型
void	**void**
bit	无等价类型
byte	**signed char**
ubyte	**unsigned char**
char	**char** (在 D 里 chars 是 unsigned)
wchar	**wchar_t** (当 sizeof(wchar_t) 为 2 时)
dchar	**wchar_t** (当 sizeof(wchar_t) 为 4 时)
short	**short**
ushort	**unsigned short**
int	**int**
uint	**unsigned**
long	**long long**

ulong	**unsigned long long**
float	**float**
double	**double**
real	**long double**
ifloat	**float _Imaginary**
idouble	**double _Imaginary**
ireal	**long double _Imaginary**
cfloat	**float _Complex**
cdouble	**double _Complex**
creal	**long double _Complex**
struct	**struct**
union	**union**
enum	**enum**
class	无等价类型
*type******	*type* *****
*type***[***dim***]**	*type***[***dim***]**
*type***[***dim***]***	*type***(*)[***dim***]**
*type***[]**	无等价类型
*type***[***type***]**	无等价类型
type **function(***parameters***)**	*type***(*)(***parameters***)**
type **delegate(***parameters***)**	无等价类型

对于大多数的 32 位 C 编译器来说，上述对应关系是成立的。C 标准并不约束类型的大小，所以使用这种对应关系时要格外小心。

15.4 调用 printf()

主要的问题是 printf 格式指示符如何匹配对应的 D 数据类型。尽管按照设计，printf 只能处理以 0 结尾的字符串，不能处理 D 的 char 动态数组，但事实证明，由于 D 动态数组的结构是{长度，指向数据的指针}，printf 的 `%.*s` 格式工作的很好：

```
void foo(char[] string)
{
   printf("my string is: %.*s\n", string);
}
```

机敏的读者会注意到 printf 格式字符串文字量并不以‘\0’结尾。这是因为如果字符串文字量不是数据结构的初始值，就会在结尾处存储一个辅助的‘\0’。

有一个改进的 D 函数可以用于格式化输出，它就是 `std.stdio.writef()`。

15.5 结构和联合

D 的结构和联合同 C 中的相似。

C 代码通常使用命令行选项或者各种实现提供的 #pragma 指令指定结构的对齐或者紧缩方式。D 支持与 C 编译器规则对应的显式的对齐特征。可以先查看 C 代码是如何对齐的，然后据此显式地设置 D 结构的对齐方式。

D 不支持位域(bit field)。如有必要，则可以使用移位(shift)和屏蔽(mask)操作来进行模拟。

第 16 章 C++ 语言接口

D 不提供到 C++ 的接口。但是因为 D 提供了对 C 的接口，所以可以访问采用 C 链接的 C++ 函数。

D 的类对象同 C++ 的类对象不兼容。

第 17 章 移植性指南

从软件工程的观点来看，应尽量减少代码中那些可以避免的移植性问题。用于减少潜在的移植性问题的技术有：

- 应该将整数和浮点类型的大小视为下界。算法应该能够在相应的类型大小增长后依然运行良好。
- 浮点运算应该可以使用高于保存相应值的变量的精度。浮点算法应该在相应的类型的精度提高后依然运行良好。
- 避免依赖于计算中那些副作用的顺序，因为编译器可能会改变这些顺序。例如：

```
a + b + c
```

  可以按照各种顺序计算：(a + b) + c、a + (b + c)、(a + c) + b、(c + b) + a 等。括号控制运算符的优先级，但括号不能控制求值的顺序。

  函数参数既可能从左到右计算，也可能从右到左计算，这依赖于所采用的调用管理。

  如果可结合的运算符 + 或 * 的操作数是浮点值，表达式的顺序不会被调整。

- 避免依赖于字节序；也就是，不要依赖于 CPU 是“低字节优先”还是“高字节优先”。
- 避免依赖于指针或者引用的大小，它们可不一定同某个整数一样大小。
- 如果不可避免的要依赖于类型的大小，应该在代码中放入一个 `assert` 进行验证：

```
assert(int.sizeof == (int*).sizeof);
```

17.1 32 位平台向 64 平台移植

64 位处理器和操作系统就在我们面前。初步的想法：

- 无论是在 32 位还是在 64 位代码中，整数类型的大小相同。
- 从 32 位迁移到 64 位后，指针和引用的大小将从 4 字节增为 8 字节。
- 使用 **size_t** 作为可以覆盖整个地址空间的那个无符号整数类型的别名。数组索引值的类型应该是 **size_t**。
- 使用 **ptrdiff_t** 作为可以覆盖整个地址空间的那个无符号整数类型的别名。代表两个指针不同点的类型应该是 **ptrdiff_t**。
- **.length**、**.size**、**.sizeof**、**.offsetof** 和 **.alignof** 特性的类型为 **size_t**。

17.2 Endianness

Endianness（字节序）指的是多字节类型存储的顺序。顺序有两种：*big endian（高字节优先）* 和 *little endian（低字节优先）*。编译器依据目标系统的顺序预先定义了 version 标识符：**BigEndian** 或 **LittleEndian**。所有的 x86 系统都是 little endian(低字节优先)。

需要考虑到字节序的情形有：

- 当从一个使用不同字节序格式写成的外部源（如一个文件）读取数据时。
- 当读取或写入象 **long** 或 **double** 那样的多字节类型的单个字节时。

17.3 OS 特有的代码

系统特有的代码可以通过把不同处分离出来单独放入独立模块中的方式来处理。在编译时，导入相应系统所特有的模块。

在处理较小的差异时，可以通过在系统特有的模块内定义一个常量，然后在 *If 语句* 或者 *StaticIf 语句* 中使用该常量的方式来处理。

第 18 章 HTML 中嵌入 D

D 编译器可以提取和编译嵌入在 HTML 文件中的 D 代码。此种能力表明可以编写在具备完全格式化和显示 HTML 能力的浏览器进行显示的 D 代码。

例如，让一个类名实际超链到定义该类的地方是可行的。对于浏览此代码的人来说并没有什么新东西，它只是使用了 HTML 浏览器的一般功能。字符串可以被显示成绿色，注释则显示成红色，而关键字显示成**粗体**，都有可能。甚至在代码里作为一般 HTML 图像标记嵌入图片都是可能的。

在 HTML 里嵌入 D 使得把代码的文档和代码本身一起放置到一个文件里成为可能。这样也就不再需要转换在注释里的文档，以便日后可以被技术作者提取。它的代码和文档可以同时地被维护，不需要重复辛苦。

它的工作原理是很简单。如果到编译器前台的源文件在 .htm 或 .html 里，那么这些代码就会被假定内嵌在 HTML 中。这个源文件然后会被预处理——去除 <code> 和 </code> 外的所有文本。然后，去除所有其它的 HTML 标记，而嵌入的字符编码会被转换成 ASCII。这个处理过程不会去分析 HTML 本身的错误。在原始 HTML 文件里的所有新行会保持在它们在那个预处理文本里相应的位置，因此调试行号也就会保持一致。最终文本就被送给 D 编译器。

这里有一个 D 程序实例——"hello world"，每一个 HTML 文件都可以嵌入它。此文件可以被编译并运行。

```
import std.stdio;

int main()
{
 writefln("hello world");
 return 0;
}
```

第 19 章 D 应用程序接口

符合 D ABI（Application Binary Interface——应用程序二进制接口）的 D 实现可以生成库、DLL 等等，这样由其它实现创建的二进制文件就可以跟 D 进行交互操作了。

这份规范的大部分都处于 TBD（To Be Defined——尚未完成）状态。

19.1 C ABI

本规范中提到的 C ABI 指的是目标系统中的 C 应用程序二进制接口。C 和 D 代码应该能够自由地链接到一起，尤其是，D 代码应该能够访问所有的 C ABI 运行时库。

19.2 基本类型

尚未完成

19.3 结构

符合目标平台上的 C ABI 结构设计。

19.4 类

一个对象的组成如下：

偏移量	内容
0	指向 vtable 的指针
ptrsize	监视器
*ptrsize*2...*	非静态成员

vtable 的组成为：

偏移量	内容
0	指向 ClassInfo 实例的指针
ptrsize...	指向虚拟成员函数的指针

类的定义：

```
class XXXX
{
    ....
};
```

生成下列内容：

- 一个叫做 ClassXXXX 的类实例。
- 名叫 StaticClassXXXX 的类型，它用于定义所有的静态成员的。
- 一个 StaticClassXXXX 型的实例，名叫 StaticXXXX 的，用于静态成员。

19.5 接口

尚未完成

19.6 数组

动态数组的组成为：

偏移量	内容
0	数组维数
size_t	指向数组数据的指针

动态数组的声明：

```
type[] array;
```

静态数组的声明：

```
type[dimension] array;
```

因此，静态数组总是静态地将维数作为类型的一部分，这样它就跟 C 中的实现一样。静态数组和动态数组之间可以方便地相互转换。

19.7 关联数组(Associative Arrays)

关联数组由一个指向“不透明物(opaque)”的指针组成，该“不透明物”为具体实现所定义的类型。目前的实现都包含在 phobos/internal/aaA.d 里。

19.8 引用类型(Reference Types)

D 有引用类型，但它们都是隐式的。例如，类总是通过引用来访问的；这意味着类的实例决不会位于堆栈里，也不会作为函数参数被传递。

当传递一个静态数组到函数时，虽然声明的是静态数组，实际的结果是静态数组的引用。例如：

```
int[3] abc;
```

当将 abc 传递给函数时，会产生下面的隐式转换：

```
void func(int[3] array); // 实际上是 <引用到><数组 array[3]><其类型为 int>
void func(int* p);       // abc 被转换成一个指针
                         // 指向第一个元素
void func(int[] array);  // abc 被转换成一个动态数组
```

19.9 名字碎解(Name Mangling)

D 实现类型安全连接的方式是通过将 D 标识符 *碎解(mangling)* 为包含域(include scope) 和类型信息(type information)。

```
碎解的名字:
_D 限定的名字 类型
_D 限定的名字 M 类型

限定的名字:
符号名
符号名 限定的名字

符号名:
    LName
模板实例名
```

M 表示该符号是一个需要 `this` 指针的函数。

模板实例名有类型以及编码在当中的参数的值：

```
模板实例名:
__T LName 模板参数 Z

模板参数:
单个模板参数
单个模板参数 多个模板参数

单个模板参数:
T 类型
V 类型 值
    S LName

值:
n
数
N 数
e 十六进制浮点数
c 十六进制浮点数 c 十六进制浮点数
A 数 值...

十六进制浮点数:
NAN
    INF
    NINF
N 多个十六进制数字 P 指数
十六进制数字 P 指数

指数:
N 数
数

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:
阿拉伯数字
    A
    B
    C
    D
    E
F
```

n

代表 **null** 参数。

数

代表正数（包括字母）。

N *数*

代表负数

e *十六进制浮点数*

代表浮点型实数和虚数

c *十六进制浮点数* **c** *十六进制浮点数*

代表浮点型复数

宽度 数 **_** *十六进制数字*

宽度 表示一个字符的大小是 1 个字节**(a)**、2 个字节**(w)** 还是 4 个字节**(d)**。*数* 指的是字符串里的字符数。*十六进制数字* 就是字符串的十六进制数据。

A *数 值...*

数组字法。将*值* 重复指定 *数* 那么多次。

```
名字:
名字起始符
名字起始符 多个名字字符

名字起始符:

    _
阿拉伯字母

单个名字字符:
名字起始符
阿拉伯数字

多个名字字符:
单个名字字符
单个名字字符 多个名字字符
```

名字 指的是标准 D 标识符。

```
LName:
数目 名字

数:
阿拉伯数字
```

```
阿拉伯数字 数

阿拉伯数字:
    0
    1
    2
    3
    4
    5
    6
    7
    8
    9
```

LName 指的是这样一个名字：以 *数* 做为开头，同时给出 *名字* 中的字符数。

19.10 类型碎解(Type Mangling)

通过简单的线性方法将类型碎解成这样：

```
类型:
    Const
不变量
Array 类型
Sarray 类型
Aarray 类型
Pointer 类型
Function 类型
Ident 类型
Class 类型
Struct 类型
Enum 类型
Typedef 类型
Delegate 类型
None 类型
Void 类型
Byte 类型
Ubyte 类型
Short 类型
Ushort 类型
Int 类型
Uint 类型
Long 类型
Ulong 类型
Float 类型
Double 类型
Real 类型
Ifloat 类型
Idouble 类型
Ireal 类型
Cfloat 类型
Cdouble 类型
```

```
Creal类型
Bool类型
Char类型
Wchar类型
Dchar类型
类型元组

Const:
    x Type

不变量:
    y Type

Array类型:
A 类型

Sarray类型:
G 数 类型

Aarray类型:
H 类型 值

Pointer类型:
P 类型

Function类型:
调用协定 多个参数 ArgClose 类型

调用协定:
F
    U
    W
    V
    R

多个参数:
单个参数
单个参数 多个参数

单个参数:
类型
J 类型
K 类型
L 类型

ArgClose
X
    Y
    Z

Ident类型:
```

```
    I LName

Class 类型:
    C LName

Struct 类型:
    S LName

Enum 类型:
    E LName

ypedef 类型:
    T LName

Delegate 类型:
D Function 类型

None 类型:
n

Void 类型:
    v

Byte 类型:
g

Ubyte 类:
    h

Short 类型:
s

Ushort 类型:
    t

Int 类型:
i

Uint 类型:
    k

Long 类型:
    l

Ulong 类型:
m

Float 类型:
    f

Double 类型:
    d

Real 类型:
    e
```

```
Ifloat 类型:
    o

Idouble 类型:
    p

Ireal 类型:
    j

Cfloat 类型:
    q

Cdouble 类型:
r

Creal 类型:
    c

Bool 类型:
    b

Char 类型:
    a

Wchar 类型:
    u

Dchar 类型:
    w

类型元组:
B 数 多个参数
```

19.11 函数调用协定

extern (C) 调用协定符合被在主机系统上所支持的 C 编译器使用的 C 协定。适合于 x86 的 extern (D) 调用协定在这里有描述。

19.11.1 寄存器协定

- EAX、ECX、EDX 都是杂合(scratch)寄存器，并且会被函数破坏(destroy)。
- EBX、ESI、EDI、EBP 在跨函数调用时都需要被保护。
- EFLAGS 在跨函数调用时被假定破坏了，除了值必须为向前的方向标志以外。
- FPU 栈在调用一个函数时必须为空。
- FPU 控制字在跨函数调用时必须被保护。
- 浮点返回值被返回在 FPU 栈里。即使不使用它们，也需要调用者将它们清除干净。

19.11.2 返回值(Return Value)

- bool、byte、ubyte、short、ushort、int、uint、指针等类型、对象以及接口都被返回在 EAX 里。
- long 和 ulong 被返回在 EDX 和 EAX 里面，这里 EDX 获得最重要的一半(the most significant half)。
- float、double、real、ifloat、idouble、ireal 都被返回在 ST0 里。
- cfloat、cdouble、creal 都返回在 ST1 和 ST0 里面，这里 ST1 是实数部分，而 ST0 则代表虚数部分。
- 动态数组返回指针到 EDX 里，而长度则返回在 EAX 里。
- 关联数组被返回到 EAX 里，同时“垃圾(garbage)”被返回在 EDX 里。EDX 的值在将来可能会被移除；这里用来跟早期的 AA 实现保持兼容。
- 委托(delegate)返回指向函数的指针到 EDX 里，而返回的内容存放在 EAX 里面。
- 对于 Windows，1、2 和 4 个字节大小的结构被返回在 EAX。
- 对于 Windows，8 个字节的结构被返回在 EDX 和 EAX 里面，这里 EDX 获得最重要的一半。
- 对于其它大小的结构，以及在 Linux 里的结构，它们的返回值都是通过隐藏的指针（作为函数的参数被传递）被保存的。
- 构造函数返回 this 指针到 EAX 里。

19.11.3 形式参数

参数个数固定(non-variadic)型函数的形参：

```
foo(a1, a2, ..., an);
```

会像下面那样被传递：

a1
a2
...
an
hidden
this

这里的 *hidden* 在需要返回一个结构值时才出现，而 *this* 的出现，则需要根据情况而定，是代表成员函数的 this 指针还是代表嵌套函数的上下文指针(context pointer)。

在满足下面的条件时，最后个参数被传递在 EAX 里，而不是被压入到栈里：

- 适合存放在 EAX 里。
- 不是一个 3 字节大小的结构。
- 为一个浮点类型。

参数并不总是被压成多个 4 字节单元，同时向上累加，因此堆栈总是被分配成 4 个字节大小一个单元。首先压入最最重要的第一个。**out** 和 **ref** 传递的是指针。静态数组传递的是指向第一个元素的指针。在 Windows 平台里，real 压入时是一个元素 10 字节大小，而压入 creal 时，一个元素则占用 20 字节大小。在 Linux 平台里，real 压入时是一个元素 12 字节大小，而压入 creal 时，一个元素则占用 12 字节大小。多出的两个字节占据“最重要(most significant)”的位置。

被调用者负责清理堆栈。

参数个数不定(variadic)型函数的形参：

```
void foo(int p1, int p2, int[] p3...)
foo(a1, a2, ..., an);
```

会像下面那样被传递：

p1
p2
a3
hidden
this
可变(variadic)部分被转换成一个动态数组，而其余部分则跟参数固定(non-variadic)型函数一样。

参数个数不定(variadic)型函数的形参：

```
void foo(int p1, int p2, ...)
foo(a1, a2, a3, ..., an);
```

会像下面那样被传递：

an
...
a3
a2
a1
_arguments
hidden
this
要求调用者清除堆栈。**_argptr** 不会被传递，它由被调用者计算。

19.12 异常处理(Exception Handling)

19.12.1 Windows

符合微软 Windows 结构化异常处理协定。

19.12.2 Linux

使用静态地址范围/处理器表。尚未完成

19.13 垃圾回收

对此的接口可以在 `phobos/internal/gc` 里找到。

19.14 运行时间辅助函数

这些可以在 `phobos/internal` 找到。

19.15 模块初始化和终止(Module Initialization and Termination)

尚未完成

19.16 单元测试

尚未完成

19.17 符号调试(Symbolic Debugging)

D 有的一些类型是现有的 C 或 C++ 调试器中是无法表示的。它们就是动态数组、关联数组和委托。把这些类型表示成结构会产生问题，因为结构的函数调用协定跟这些类型是大大不同的；这就使得 C/C++ 调试器会表示成错误的东西。对于这些调试器，它们被表示成符合该类型调用协定的 C 类型。**dmd** 编译器在使用 **-gc** 编译器开关时，只生成 C 符号类型信息。

C 调试器的类型

D 类型	C 的表示
动态数组	unsigned long long
关联数组	void*
delegate	long long

对于可以被修改来接受新类型的调试器，下面的扩展有助于它们完全支持这些类型。

19.17.1 Codeview 调试器扩展

D 通过 **LF_OEM** (0x0015) 来表明使用了 Codeview OEM 通用类型信息。格式为：

Codeview 针对 D 的 OEM 扩展

字段大小	2	2	2	2	2	2
D 类型	**页索引**	**OEM 标识**	**recOEM**	**数字索引值**	**类型索引**	**类型索引**
动态数组	LF_OEM	*OEM*	1	2	*@index*	*@element*
关联数组	LF_OEM	*OEM*	2	2	*@key*	*@element*
delegate	LF_OEM	*OEM*	3	2	*@this*	*@function*

OEM 0x42
index 数组索引的类型索引
key 关键字的类型索引
element 数组元素的类型索引
this 上下文指针的类型索引
function 函数的类型索引
这些扩展可以通过 obj2asm 完整输出。

调试器 Ddbg 支持它们。

19.17.2 Dwarf 调试器扩展

添加了下面页类型：

针对 D 的 Dwarf 扩展

D 类型	标识符	值	格式
动态数组	DW_TAG_darray_type	0x41	DW_AT_type 为元素类型
关联数组	DW_TAG_aarray_type	0x42	DW_AT_type 为元素类型；DW_AT_containing_type 为键的类型
delegate	DW_TAG_delegate_type	0x43	DW_AT_type 为函数类型；DW_AT_containing_type 为 'this' 类型

这些扩展可以通过 dumpobj 完整输出。

调试器 ZeroBUGS 支持它们。

第三篇 附 录

第 20 章 D vs 其它语言

To D, or not to D. -- Willeam NerdSpeare

本表格快速而粗糙的列出了 D 的各种功能， 这些功能都可以被用来跟其它语言进行比较。尽管各个语言的标准库中也提供了很多功能，但这张表格只考虑内建到语言核心中的功能。基本解释。

D 语言功能比较表

功能	D	
垃圾回收	有	
函数		
函数委托	有	
函数重载	有	
out 型函数参数	有	
嵌套函数	有	
函数字法	有	
动态闭包	有	
类型安全的 variadic(个数不定) 型参数	有	
懒式的函数实参求值	有	
数组		
轻型数组	有	
大小可变数组	有	
内建字符串	有	
数组分割	有	
数组边界检查	有	
数组字法	有	
关联数组	有	
强类型定义	有	
字符串 switch	有	
别名	有	
面向对象编程(OOP)		
面向对象	有	

功能	D	
多重继承	无	
接口	有	
运算符重载	有	
模块	有	
动态的类装入	无	
嵌套类	有	
内部（适配器）类	有	
协变式类型返回	有	
特性	有	
性能		
内联汇编	有	
硬件直接访问	有	
轻型对象	有	
显示内存分配控制	有	
虚拟机无关	有	
直接有本地代码生成	有	
泛型编程		
类模板(Class Templates)	有	
函数模板(Function Templates)	有	
隐示的函数模板实例化	有	
局部显式特例化	有	
带值模板参数	有	
模板式模板参数	有	
Variadic 型模板参数	有	
混入(Mixins)	有	
static if	有	
is 表达式	有	
typeof	有	

功能	D	
foreach	有	
隐式类型接口	有	
可靠性		
契约编程	有	
单元测试	有	
静态构造顺序	有	
初始化保证	有	
RAII (自动的析构器)	有	
异常处理	有	
作用域(Scope)守护	有	
try-catch-finally 块	有	
线程同步原语	有	
兼容性		
C 风格语法	有	
枚举类型	有	
支持所有的 C 类型	有	
80 位浮点数	有	
复数和虚数	有	
直接访问 C	有	
使用现有的调试器	有	
结构成员对齐控制	有	
生成标准目标文件	有	
宏文本预处理器	无	
其它		
条件编译	有	
Unicode 源代码	有	
文档注释	有	

20.1 注解

面向对象

这意味着支持类、成员函数、继承和虚函数分派。

内联汇编

许多 C 和 C++ 编译器支持内联汇编，但这并不是语言标准的一部分，并且各种实现的语法和质量都有很大差别。

接口

C++ 对接口的支持是如此不完善，以至于专门发明了 IDL（Interface Description Language - 接口定义语言）弥补这个不足。

模块

许多人指出 C++ 并不真正支持模块，他们是对的。但是 C++ 的名字空间和头文件一起可以实现模块的许多功能。

垃圾回收

Hans-Boehm 垃圾回收器可以成功的用于 C 和 C++，但是这并不是语言标准的一部分。

隐式类型接口

这个指的是可以从初始化那里得到某个声明的类型的能力。

契约编程

作为一个扩展，Digital Mars C++ 编译器支持 契约编程。请把一些用于进行“契约编程”的 C++ 技术 跟 D 进行一个比较。

大小可变数组

C ++ 标准库中有一部分是用于实现变长数组的，但是，它们并不是核心语言的一部分。一个符合标准的独立的 C++（C++98 17.4.1.3）实现并不必须提供这些库。

内建字符串

C++ 标准库中有一部分是用于实现字符串的，但是，它们并不是核心语言的一部分。一个符合标准的独立的 C++（C++98 17.4.1.3）实现并不必须提供这些库。这里有一个关于 C++ 字符串跟 D 内建字符串之间的比较。

强类型定义

强类型定义（Strong typedef）在 C/C++ 中可以通过将一个类型用一个结构封装来模拟。如果要采用这种方式，会使编程工作十分繁琐，所以它并不被人支持。

使用现有的调试器

采用这种方法意味着使用那些能够处理嵌入到常见可执行文件格式中的调试数据的常见调试器。不必编写一个只能用于某种语言的调试程序。

结构成员对齐控制

尽管许多的 C/C++ 编译器包括指定结构对齐方式的编译器指令，但对此并没有一个标准，而不同编译器的做法通常是不兼容的。

关于结构成员对齐，C# 标准 ECMA-334 25.5.8 只提到：“*成员在结构中的顺序是未指定的。出于对齐的目的，可以在结构的开头、中间或者结尾处加入匿名填充。填充值是未定义的。*”因此，尽管 Microsoft 可能进行扩展以支持特定的成员对齐方式，但它们不是 C# 官方标准的一部分。

支持所有的 C 类型

C99 添加了 C++ 部支持的许多新类型。

80 位浮点数

尽管 C 和 C++ 标准规定了 long double 类型，很少有编译器（除了 Digital Mars C/C++）真正实现了 80 位（或更长）的浮点类型。

混入(Mixins)

在不同的程序设计语言中，混入(Mixins)有许多不同的意义。D 的混入批的是将任意的声明序列插入（混入-mixing）到当前作用域中。混入可以在全局、类、结构或者局部级别使用。

C++ 混入

C++ 混入指的是另一套技术。其一同 D 的接口类是同类的东西。其二是指创建一个下述形式的模板：

```
template <class Base> class Mixin : public Base
{
... 混入过程体 ...
}
```

D 的混入与此不同。

Static If

C 和 C++ 中预处理器块 #if 表面上跟 D 中的“static if”是等价的。但实际上它们有着本质的不同—— #if 不能使用程序的任何常量、类型或符号。它仅能使用预处理宏。参看 本例。

Is 表达式

is 表达式 使得可以使用基于类型字符的条件编译。在 C++ 中使用模板参数模式匹配勉强实现了该功能。请参考这个实例来比较一下两种方式的不同。

同 Ada 的比较

James S. Rogers 已经写了一篇《同 Ada 的比较图》。

内部（适配器）类

一个 *嵌套类* 指的是定义在另一个类作用域内的类。一个 *内部类* 指的是能够引用外部类的成员和域的嵌套类；你可以把它们看作是拥有指向包含它们的外部类的‘this’指针的类。

文档注释

文档注释指的是使用一种标准化的方式来从使用了特殊注释符的源码中提取生成文档。

20.2 错误

如果我在这张表中犯了任何的错误，请联系我，我将改正。

第 21 章 核心语言功能 vs 库实现

D 提供有几个功能是内建到语言核心里的，而这些在其它语言（比如 C++）里却是以库的形式实现。

1. 动态数组(Dynamic Arrays)
2. 字符串(Strings)
3. 关联数组(Associative Arrays)
4. 复数

有人把这些当作语言膨胀的证据，而不是什么有用的功能。因此，为什么不把它们都实现成标准库类型呢？

一些简要的最初结论：

1. 它们每一个都被大量使用。即表示可用性方面哪怕是很小的一点提高也是值得去争取的。
2. 成为语言核心中的功能意味着，编译器在没有正确使用类型时，能够给出更多更好的关键错误信息。库实现给出的常常是基于这些实现的细节的糟糕透顶的信息。
3. 库功能不能造就新的语法、新的操作符或者新的特征符。
4. 库实现通常需要花费大量编译时间来处理该实现，一遍又一遍地编译每一个，这样就减慢了编译速度。
5. 有人希望库实现能给最终用户提供机动性。但是只要它们被标准化，标准化成允许它们特殊化的（C++ 标准支持这个）编译器的关键点，它们就会变得跟核心内建功能一样没什么机动性可言。
6. 定义新库类型的能力，尽管在最近几年有很大进步，但仍然有很多是被希望能平滑地集成到已有的语言里。像解决问题不力(Rough edge——粗糙的刀刃)，表达能力不强(clumsy syntax——笨拙的语法)，以及问题考虑不全面(odd corner——残余角落)等这些情况举不胜举。

更多专门的注释：

21.1 动态数组(Dynamic Arrays)

C++ 核心内建有数组。只是表现不是太好。它们并没有得到修正，而是以做为 C++ 标准模板库一部分的形式创建了几种不同的数组类型，每一个都涉及到内建数组，且有着不同的效率。这些包括有：

- `basic_string`
- `vector`
- `valarray`
- `deque`
- `slice_array`
- `gslice_array`
- `mask_array`
- `indirect_array`

修正内建数组意味着不再需要所有那些变形。有一种数组类型就可以包含它们全部，仅需要去学习一个事物，并且让一种数组类型跟其它数组类型一起工作时不会再有问题。

同样地，内建数组让我们可以创建漂亮的语句。起始部分是数组文字，接着是一些数组特有的新的操作符。数组库的实现必须要做很多工作才能重载已有的操作符。索引操作符 `a[i]`，跟 C++ 里的一样。新增的是，数组连接符“~”，数组追加符“~=”，数组分割符“`a[i..j]`”，以及数组向量符“`a[]`”。

连接符“~”和“~=”解决了当仅有一个已有操作符被重载时可能出现的问题。通常，在数组库实现里使用“+”来完成连接。但是，这个就跟不让“+”表示数组向量相加出现矛盾。另外，连接操作跟相加操作根本就不是一回事，它们两个都使用同样的操作符就会产生混乱。

21.2 字符串(Strings)

详细的跟 C++ 的 std::string 的比较。

C++ 当然有支持字符串字母和字符数组形式的内建字符串。只是，它们都因为 C++ 内建数组的薄弱而痛苦不堪。

不过，如果一个字符串不是一个字符数组它会是什么呢？如果内建数组问题得以修正的话，难道就不能顺带解决字符串问题吗？肯定可以。D 语言没有字符串类，起初看起来有点奇怪，不过由于字符串处理实际上就是处理字符数组，因此，如果数组不工作，一个类就没有什么东西可以添加给它。

另外，那个相加操作导致内建字符串字母不是相同的类型，因为没有字符串库类类型。

21.3 关联数组(Associative Arrays)

它的最大好处就是，又一次使得语句漂亮。一个有着 `T` 类型键，并且存储 `int` 类型值的关联数组可以很自然地写成这样：

```
int[T] foo;
```

而不需要：

```
import std.associativeArray;
...
std.associativeArray.AA!(T, int) foo;
```

内建关联数组还提供了拥有关联数组字法的能力，这个可以经常需要的附加功能。

21.4 复数

详细的跟 C++ 的 std::complex 的比较。

最为显著的原因就是跟 C 的虚数和复数浮点类型兼容。其次就是拥有虚数浮点字法的能力。难道不是这个：

```
c = (6 + 2i - 1 + 3i) / 3i;
```

远比下面的更为优越：

```
c = (complex!(double)(6,2) + complex!(double)(-1,3)) / complex!(double)(0,3);
```

? 无庸置疑!

第 22 章 针对 C 程序员的 D 编程

Et tu, D? Then fall, C! -- William Nerdspeare

每个有经验的 C 程序员都积累了一系列的习惯和技术，这几乎成了第二天性。有时候，当学习一门新语言时，这些习惯会因为太令人舒适而使人看不到新语言中等价的方法。所以下面收集了一些常用的 C 技术，以及如何在 D 中完成同样的任务。

因为 C 没有面向对象的特征，所以有关面向对象的论述请参见“针对 C++ 程序员的 D 编程”。

C 的预处理程序在“C 的预处理程序 vs D”中有讨论。

- 获得一个类型的大小
- 获得一个类型的最大值和最小值
- 基本类型
- 特殊的浮点值
- 浮点除法中的余数
- 在浮点比较中处理 NAN
- 断言
- 初始化数组的所有元素
- 遍历整个数组
- 创建可变大小数组
- 字符串连接
- 格式化输出
- 函数的向前引用
- 无参函数
- 带标号的 break 和 continue
- goto 语句
- 结构标记名字空间
- 查找字符串
- 设置结构成员对齐方式
- 匿名结构和联合
- 声明结构类型和变量
- 获得结构成员的偏移量
- 联合的初始化
- 结构的初始化
- 数组的初始化
- 转义字符串字法
- Ascii 字符 vs 宽字符
- 类似枚举的数组
- 创建一个新的 typedef 类型

- 比较结构
- 比较字符串
- 数组排序
- 易失性内存访问
- 字符串字法
- 遍历数据结构
- 无符号右移
- 动态闭包
- Variadic 型函数参数

22.1.1 获得一个类型的大小

22.1.1.1 C 方式

```
sizeof(int)
sizeof(char *)
sizeof(double)
sizeof(struct Foo)
```

22.1.1.2 D 方式

使用大小属性：

```
int.sizeof
(char *).sizeof
double.sizeof
Foo.sizeof
```

22.1.2 获得一个类型的最大值和最小值

22.1.2.1 C 方式

```
#include <limits.h>
#include <math.h>

CHAR_MAX
CHAR_MIN
ULONG_MAX
DBL_MIN
```

22.1.2.2 D 方式

```
char.max
char.min
```

```
ulong.max
double.min
```

22.1.3 基本类型

22.1.3.1 C 类型到 D 类型

```
bool                =>        bit
char                =>        char
signed char         =>        byte
unsigned char       =>        ubyte
short               =>        short
unsigned short      =>        ushort
wchar_t             =>        wchar
int                 =>        int
unsigned            =>        uint
long                =>        int
unsigned long       =>        uint
long long           =>        long
unsigned long long  =>        ulong
float               =>        float
double              =>        double
long double         =>        real
_Imaginary long double =>     ireal
_Complex long double   =>     creal
```

尽管 char 是一个无符号 8 bit 的类型，而 wchar 是一个无符号 16 bit 的类型，它们还是被划为独立的类型以支持重载解析和类型安全。

在 C 中各种整数类型和无符号类型的大小是不同的；不像 D 中那样。

22.1.4 特殊的浮点值

22.1.4.1 C 方式

```
#include <fp.h>

NAN
INFINITY

#include <float.h>

DBL_DIG
DBL_EPSILON
DBL_MANT_DIG
DBL_MAX_10_EXP
DBL_MAX_EXP
DBL_MIN_10_EXP
DBL_MIN_EXP
```

22.1.4.2 D 方式

```
double.nan
```

```
double.infinity
double.dig
double.epsilon
double.mant_dig
double.max_10_exp
double.max_exp
double.min_10_exp
double.min_exp
```

22.1.5 浮点除法中的余数

22.1.5.1 C 方式

```
#include <math.h>

float f = fmodf(x,y);
double d = fmod(x,y);
long double r = fmodl(x,y);
```

22.1.5.2 D 方式

D 支持浮点操作数的求余（‘%’）运算符：

```
float f = x % y;
double d = x % y;
real r = x % y;
```

22.1.6 在浮点比较中处理 NAN

22.1.6.1 C 方式

C 对操作数为 NAN 的比较的结果没有定义，并且很少有 C 编译器对此进行检查（Digital Mars C 编译器是个例外，DM 的编译器检查操作数是否是 NAN）。

```
#include <math.h>

if (isnan(x) || isnan(y))
   result = FALSE;
else
   result = (x < y);
```

22.1.6.2 D 方式

D 的比较和运算符提供对 NAN 参数的完全支持。

```
result = (x < y);         // 如果 x 或 y 是 nan ，则值为 false
```

22.1.7 断言是所有好的防御性编码策略必要的组成部分

22.1.7.1 C 方式

C 不直接支持断言，但是它支持 __FILE__ 和 __LINE__ ，可以以它们为基础使用宏构建断言。事实上，除了断言以外，__FILE__ 和 __LINE__ 没有什么其他的实际用处。

```
#include <assert.h>

assert(e == 0);
```

22.1.7.2 D 方式

D 直接将断言构建在语言里：

```
assert(e == 0);
```

22.1.8 初始化数组的所有元素

22.1.8.1 C 方式

```
#define ARRAY_LENGTH        17
int array[ARRAY_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++)
   array[i] = value;
```

22.1.8.2 D 方式

```
int array[17];
array[] = value;
```

22.1.9 遍历整个数组

22.1.9.1 C 方式

数组长度单独定义的，或者使用笨拙的 sizeof() 表达式来获取长度。

```
#define ARRAY_LENGTH        17
int array[ARRAY_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++)
   func(array[i]);
```

或者：

```
int array[17];
for (i = 0; i < sizeof(array) / sizeof(array[0]); i++)
   func(array[i]);
```

22.1.9.2 D 方式

可以使用“length”属性访问数组的长度：

```
int array[17];
for (i = 0; i < array.length; i++)
   func(array[i]);
```

或者使用更好的方式：

```
int array[17];
foreach (int value; array)
   func(value);
```

22.1.10 创建可变大小数组

22.1.10.1 C 方式

C 不能用数组来处理这个。需要另外创建一个变量保存长度，并显式地管理数组大小：

```
#include <stdlib.h>

int array_length;
int *array;
int *newarray;

newarray = (int *)
   realloc(array, (array_length + 1) * sizeof(int));
if (!newarray)
   error("out of memory");
array = newarray;
array[array_length++] = x;
```

22.1.10.2 D 方式

D 支持动态数组，可以轻易地改变大小。D 支持所有的必备的内存管理。

```
int[] array;

array.length = array.length + 1;
array[array.length - 1] = x;
```

22.1.11 字符串连接

22.1.11.1 C 方式

有几个难题需要解决，如什么时候可以释放内存、如何处理空指针、得到字符串的长度以及内存分配：

```
#include <string.h>

char *s1;
char *s2;
char *s;

// 连接 s1 和 s2，并将结果存入 s
free(s);
s = (char *)malloc((s1 ? strlen(s1) : 0) +
                (s2 ? strlen(s2) : 0) + 1);
if (!s)
   error("out of memory");
if (s1)
   strcpy(s, s1);
else
   *s = 0;
if (s2)
   strcpy(s + strlen(s), s2);

// 追加 "hello" 到 s
char hello[] = "hello";
char *news;
size_t lens = s ? strlen(s) : 0;
news = (char *)
   realloc(s, (lens + sizeof(hello) + 1) * sizeof(char));
if (!news)
   error("out of memory");
s = news;
memcpy(s + lens, hello, sizeof(hello));
```

22.1.11.2 D 方式

D 为 char 和 wchar 数组分别重载了‘~’和‘~=’运算符用于连接和追加：

```
char[] s1;
char[] s2;
char[] s;

s = s1 ~ s2;
s ~= "hello";
```

22.1.12 格式化输出

22.1.12.1 C 方式

printf() 是通用的格式化打印方法：

```
#include <stdio.h>

printf("Calling all cars %d times!\n", ntimes);
```

22.1.12.2 D 方式

我们还能说什么呢？printf() 规则：

```
printf("Calling all cars %d times!\n", ntimes);
```

writefln() 在类型紧要(type-aware)和类型安全(type-safe)方面比 printf() 有提高：

```
import std.stdio;

writefln("Calling all cars %s times!", ntimes);
```

22.1.13 函数的向前引用

22.1.13.1 C 方式

不能向前引用函数。因此，如果要调用源文件中尚未出现的函数，就必须在调用之前插入函数声明。

```
void forwardfunc();

void myfunc()
{
   forwardfunc();
}

void forwardfunc()
{
   ...
}
```

22.1.13.2 D 方式

程序被看作一个整体，所以没有必要编写前向声明，而且这也是不允许的！D 避免了编写前向函数声明的繁琐和由于重复编写前向函数声明而造成的错误。函数可以按照任何顺序定义。

```
void myfunc()
{
   forwardfunc();
}

void forwardfunc()
{
```

```
   ...
}
```

22.1.14 无参函数

22.1.14.1 C 方式

```
void function(void);
```

22.1.14.2 D 方式

D 是强类型语言，所以没有必要显式地说明一个函数没有参数，只需在声明时不写参数即可。

```
void function()
{
   ...
}
```

22.1.15 带标号的 break 和 continue

22.1.15.1 C 方式

break 和 continue 只用于嵌套中最内层的循环或 switch 结构，所以必须使用 goto 实现多层的 break：

```
   for (i = 0; i < 10; i++)
   {
      for (j = 0; j < 10; j++)
      {
         if (j == 3)
            goto Louter;
         if (j == 4)
            goto L2;
      }
    L2:
      ;
   }
Louter:
   ;
```

22.1.15.2 D 方式

break 和 continue 语句后可以带有标号。该标号是循环或 switch 结构外围的，break 用于退出该循环。

```
Louter:
```

```
   for (i = 0; i < 10; i++)
   {
      for (j = 0; j < 10; j++)
      {
         if (j == 3)
            break Louter;
         if (j == 4)
            continue Louter;
      }
   }
// 中断 Louter 跳转到这里
```

22.1.16 goto 语句

22.1.16.1 C 方式

饱受批评的 goto 语句是专业 C 程序员的一个重要工具。有时，这是对控制流语句的必要补充。

22.1.16.2 D 方式

许多 C 方式的 goto 语句可以使用 D 中的标号 break 和 continue 语句替代。但是 D 对于实际的程序员来说是一门实际的语言，他们知道什么时候应该打破规则。所以，D 当然支持 goto！

22.1.17 结构标记名字空间

22.1.17.1 C 方式

每次都要将 struct 关键字写在结构类型名之前简直是烦人透顶，所以习惯的用法是：

```
typedef struct ABC { ... } ABC;
```

22.1.17.2 D 方式

结构标记名字不再位于单独的名字空间，它们同普通的名字共享一个名字空间。因此：

```
struct ABC { ... }
```

22.1.18 查找字符串

22.1.18.1 C 方式

给定一个字符串，将其同一系列可能的值逐个比较，如果匹配就施行某种动作。该方法的典型应用要数命令行参数处理。

```
#include <string.h>
void dostring(char *s)
{
   enum Strings { Hello, Goodbye, Maybe, Max };
   static char *table[] = { "hello", "goodbye", "maybe" };
   int i;

   for (i = 0; i < Max; i++)
   {
      if (strcmp(s, table[i]) == 0)
         break;
   }
   switch (i)
   {
      case Hello:   ...
      case Goodbye: ...
      case Maybe:   ...
default:      ...
   }
}
```

该方法的问题是需要维护三个并行的数据结构：枚举、表和 switch-case 结构。如果有很多的值，维护这三种数据结构之间的对应关系就不那么容易了，所以这种情形就成了孕育错漏的温床。 另外，如果值的数目很大，相对于简单的线性查找，采用二叉查找或者散列表会极大地提升性能。但是它们需要更多时间进行编码，并且调试难度也更大。这个从未完成过也是很平常的。

22.1.18.2 D 方式

D 扩展了 switch 语句的概念，现在它能像处理数字一样处理字符串。这样，编写字符串查找的代码就变得直接多了：

```
void dostring(char[] s)
{
   switch (s)
   {
      case "hello":   ...
      case "goodbye": ...
      case "maybe":   ...
      default:        ...
   }
}
```

添加新的 case 子句也变得容易起来。可以由编译器为其生成一种快速的查找方案，这样也就避免了由于手工编码而消耗的时间及引入的 bug 。

22.1.19 设置结构成员对齐方式

22.1.19.1 C 方式

这是使用命令行选项完成的，而且该效果会影响整个程序，并且如果有模块或者库没有重新编译，结果会是悲剧性的。为了解决这个问题，需要用到 #pragma:

```
#pragma pack(1)
struct ABC
{
   ...
};
#pragma pack()
```

但是，无论在理论上还是实际上，#pragma 在编译器之间都是不可移植的。

22.1.19.2 D 方式

很显然，设置对齐的主要目的是使数据可移植，因此需要一种表述结构的可移植的方式。

```
struct ABC
{
int z;                // z 按照默认方式对齐

align (1) int x;        // x 按字节对齐
 align (4)
 {
...                   // {} 中的声明按照 dword 对齐
 }
 align (2):// 从现在开始按照 word 对齐

int y;                // y 按照字(word)对齐
}
```

22.1.20 匿名结构和联合

有时，有必要控制嵌套在结构或联合内部的结构的分布。

22.1.20.1 C 方式

C 不允许出现匿名的结构或联合，这意味着需要使用傀儡标记名和傀儡成员：

```
struct Foo
{
   int i;
   union Bar
   {
      struct Abc { int x; long y; } _abc;
```

```
     char *p;
   } _bar;
};

#define x _bar._abc.x
#define y _bar._abc.y
#define p _bar.p

struct Foo f;

f.i;
f.x;
f.y;
f.p;
```

这样做不仅笨拙，由于使用了宏，还使符号调试器无法理解程序究竟做了什么，并且宏还占据了全局作用域而不是结构作用域。

22.1.20.2 D 方式

匿名结构和联合用来以一种更自然的方式控制分布：

```
struct Foo
{
   int i;
   union
   {
      struct { int x; long y; }
      char* p;
   }
}

Foo f;

f.i;
f.x;
f.y;
f.p;
```

22.1.21 声明结构类型和变量

22.1.21.1 C 方式

可以在一条以分号结尾的语句中完成：

```
struct Foo { int x; int y; } foo;
```

Or to separate the two:

```
struct Foo { int x; int y; };   // 注意结尾处的‘;’
struct Foo foo;
```

22.1.21.2 D 方式

结构的定义和声明不能在一条语句中完成：

```
struct Foo { int x; int y; }     // 注意结尾处没有‘;’
Foo foo;
```

这意味着结尾的‘;’可以去掉，免得还要区分 struct {} 和函数及语句块的 {} 之间在分号用法上的不同。

22.1.22 获得结构成员的偏移量

22.1.22.1 C 方式

自然，又要使用宏：

```
#include <stddef>
struct Foo { int x; int y; };

off = offsetof(Foo, y);
```

22.1.22.2 D 方式

偏移量只是另一个属性：

```
struct Foo { int x; int y; }

off = Foo.y.offsetof;
```

22.1.23 联合的初始化

22.1.23.1 C 方式

联合的初始化采用“首个成员”规则：

```
union U { int a; long b; };
union U x = { 5 };                  // 初始化成员'a'为 5
```

为联合添加成员或者重新排列成员的结果对任何的初始化语句来说都是灾难性的。

22.1.23.2 D 方式

在 D 中，初始化那个成员是显式地指定的：

```
union U { int a; long b; }
U x = { a:5 };
```

还避免了误解和维护问题。

22.1.24 结构的初始化

22.1.24.1 C 方式

成员按照它们在 {} 内的顺序初始化：

```
struct S { int a; int b; };
struct S x = { 5, 3 };
```

对于小结构来说，这不是什么问题，但当成员的个数变得很大时，小心地排列初始值以同声明它们的顺序对应变得很繁琐。而且，如果新加了或者重新排列了成员的话，所有的初始化语句都需要进行适当地修改。这可是错漏的雷区。

22.1.24.2 D 方式

可以显式地初始化成员：

```
struct S { int a; int b; }
S x = { b:3, a:5 };
```

这样意义明确，并且不依赖于位置。

22.1.25 数组的初始化

22.1.25.1 C 方式

C 初始化数组时依赖于位置：

```
int a[3] = { 3,2,2 };
```

嵌套的数组可能有、也可能没有 {}：

```
int b[3][2] = { 2,3, {6,5}, 3,4 };
```

22.1.25.2 D 方式

D 也依赖于位置，但是还可以使用索引。下面的语句都产生同样的结果：

```
int[3] a = [ 3, 2, 0 ];
int[3] a = [ 3, 2 ];            // 未提供的初始值被视为 0 ，如同 C 一样
int[3] a = [ 2:0, 0:3, 1:2 ];
int[3] a = [ 2:0, 0:3, 2 ];     // 如果未提供，索引就是
                                // 前一个加一。
```

如果数组的下标为枚举的话，这会很方便，而且枚举的顺序可以变更，也可以加入新的枚举值：

```
enum color { black, red, green }
int[3] c = [ black:3, green:2, red:5 ];
```

必须显式地初始化嵌套数组：

```
int[2][3] b = [ [2,3], [6,5], [3,4] ];

int[2][3] b = [[2,6,3],[3,5,4]];            // 出错
```

22.1.26 转义字符串字法

22.1.26.1 C 方式

C 在 DOS 文件系统中会遇到问题，因为字符串中的‘\’是转义符。如果要使用文件 c:\root\file.c：

```
char file[] = "c:\\root\\file.c";
```

如果使用这则表达式的话，会让人很难高兴起来。考虑匹配引号字符串的转义序列：

```
/"[^\\]*(\\.[^\\]*)*"/
```

在 C 中，令人恐怖的表示如下：

```
char quoteString[] = "\"[^\\\\]*(\\\\.[^\\\\]*)*\"";
```

22.1.26.2 D 方式

字符串本身是 WYSIWYG（所见即所得）的。转义字符位于另外的字符串中。所以：

```
char[] file = `c:\root\file.c`;
char[] quoteString = \"  r"[^\\]*(\\.[^\\]*)*"  \";
```

著名的“hello world”字符串变为了：

```
char[] hello = "hello world" \n;
```

22.1.27 Ascii 字符 vs 宽字符

现代的程序设计工作需要语言以一种简单的方法支持 wchar 字符串，这样你的程序就可以实现国际化。

22.1.27.1 C 方式

C 使用 wchar_t 并在字符串前添加 L 前缀：

```
#include <wchar.h>
char foo_ascii[] = "hello";
wchar_t foo_wchar[] = L"hello";
```

如果代码需要同时兼容 ascii 和 wchar 的化，情况会变得更糟。需要使用宏来屏蔽 ascii 和 wchar 字符串的差别：

```
#include <tchar.h>
tchar string[] = TEXT("hello");
```

22.1.27.2 D 方式

字符串的类型由语义分析决定，所以没有必要用宏调用将字符串包裹起来：

```
char[] foo_ascii = "hello";         // 字符串被当 ascii
wchar[] foo_wchar = "hello";        // 字符串被当 wchar
```

22.1.28 类似枚举的数组

22.1.28.1 C 方式

请考虑：

```
enum COLORS { red, blue, green, max };
char *cstring[max] = {"red", "blue", "green" };
```

当项的数目较小时，很容易保证其正确。但是假设它变得相当庞大。那么当加入新的项时就会很难保证其正确性。

22.1.28.2 D 方式

```
enum COLORS { red, blue, green }

char[][COLORS.max + 1] cstring =
[
   COLORS.red   : "red",
   COLORS.blue  : "blue",
   COLORS.green : "green",
];
```

虽不完美，但却更好。

22.1.29 创建一个新的 typedef 类型

22.1.29.1 C 方式

C 中的 typedef 很弱，也就是说，他们并不真正引入一个类型。编译器并不区分 typedef 类型和它底层的类型。

```
typedef void *Handle;
void foo(void *);
void bar(Handle);

Handle h;
```

```
foo(h);                    // 未捕获的编码错漏
bar(h);                    // 正确
```

C 的解决方案是创建一个傀儡结构，目的是获得新类型才有的类型检查和重载能力。

```
struct Handle__ { void *value; }
typedef struct Handle__ *Handle;
void foo(void *);
void bar(Handle);

Handle h;
foo(h);                    // 语法错误
bar(h);                    // 正确
```

如果要给这个类型定一个默认值，需要定义一个宏，一个命名规范，然后时刻遵守这个规范：

```
#define HANDLE_INIT ((Handle)-1)

Handle h = HANDLE_INIT;
h = func();
if (h != HANDLE_INIT)
    ...
```

对于采用结构的那种解决方案，事情甚至变得更复杂：

```
struct Handle__ HANDLE_INIT;

void init_handle()       // 在开始处调用这个函数
{
    HANDLE_INIT.value = (void *)-1;
}

Handle h = HANDLE_INIT;
h = func();
if (memcmp(&h,&HANDLE_INIT,sizeof(Handle)) != 0)
    ...
```

需要记住四个名字： `Handle, HANDLE_INIT, struct Handle__, value.`

22.1.29.2 D 方式

不需要上面那样的习惯构造。只需要写成：

```
typedef void* Handle;
void foo(void*);
void bar(Handle);

Handle h;
foo(h);
bar(h);
```

为了处理默认值，可以给 typedef 添加一个初始值，可以使用 `.init` 属性访问这个初始

值：

```
typedef void* Handle = cast(void*)(-1);
Handle h;
h = func();
if (h != Handle.init)
    ...
```

仅需要记住一个名字： Handle.

22.1.30 比较结构

22.1.30.1 C 方式

尽管 C 为结构赋值定义了一种简单、便捷的方法：

```
struct A x, y;
...
x = y;
```

却不支持结构之间的比较。因此，如果要比较两个结构实例之间的相等性：

```
#include <string.h>

struct A x, y;
...
if (memcmp(&x, &y, sizeof(struct A)) == 0)
    ...
```

请注意这种方法的笨拙，而且在类型检查上得不到语言的任何支持。

在 memcmp() 里潜在着一个很龌龊的错漏。结构的分布中，由于对齐的原因，可能会有“空洞”。C 不保证这些空洞中为何值，所以两个不同的结构实例可能拥有所有成员的值都对应相等，但比较的结果却由于空洞中的垃圾的存在而为“不等”。

22.1.30.2 D 方式

D 的方式直接而显然：

```
A x, y;
...
if (x == y)
    ...
```

22.1.31 比较字符串

22.1.31.1 C 方式

库函数 strcmp() 用于这个目的：

```
char string[] = "hello";
```

```
if (strcmp(string, "betty") == 0)        // 字符串匹配吗?
   ...
```

C 的字符串以‘\0’结尾，所以由于需要不停地检测结尾的‘\0’，C 的方式在效率上先天不足。

22.1.31.2 D 方式

为什么不用 == 运算符呢？

```
char[] string = "hello";

if (string == "betty")
   ...
```

D 的字符串另外保存有长度。因此，字符串比较的实现可以比 C 的版本快得多（它们之间的差异就如同 C 的 memcmp() 同 strcmp() 之间的差异一样）。

D 还支持字符串的比较运算符：

```
char[] string = "hello";

if (string < "betty")
   ...
```

这对于排序/查找是很有用的。

22.1.32 数组排序

22.1.32.1 C 方式

尽管许多的 C 程序员不厌其烦地一遍一遍实现着冒泡排序，C 中正确的方法却是使用 qsort()：

```
int compare(const void *p1, const void *p2)
{
   type *t1 = (type *)p1;
   type *t2 = (type *)p2;

   return *t1 - *t2;
}

type array[10];
...
qsort(array, sizeof(array)/sizeof(array[0]),
      sizeof(array[0]), compare);
```

必须为每种类型编写一个 compare() 函数，而这些工作极易出错。

22.1.32.2 D 方式

这恐怕是最容易的排序方式了：

```
type[] array;
...
array.sort;       // 适当地为数组排序
```

22.1.33 易失性内存访问

22.1.33.1 C 方式

如果要访问易失性内存，如共享内存或者内存映射 I/O，需要一个易失性的指针：

```
volatile int *p = address;

i = *p;
```

22.1.33.2 D 方式

D 有一种易失性语句，而不是一种类型修饰符：

```
int* p = address;

volatile { i = *p; }
```

22.1.34 字符串字法

22.1.34.1 C 方式

C 的字符串文字不能跨越多行，所以需要用‘\’将文本块分割为多行：

```
"This text spans\n\
multiple\n\
lines\n"
```

如果有很多的文本的话，这种做法是很繁琐的

22.1.34.2 D 方式

字符串文字量可以跨越多行，如下所示：

```
"This text spans
multiple
lines
"
```

所以可以简单用剪切/粘贴将成块的文字插入到 D 源码中。

22.1.35 遍历数据结构

22.1.35.1 C 方式

考虑一个遍历递归数据结构的函数。在这个例子中，有一个简单的字符串符号表。数据结构为一个二叉树数组。代码需要穷举这个结构以找到其中的特定的字符串，并检测它是否是唯一的实例。

为了完成这项工作，需要一个辅助函数 membersearchx 递归地遍历整棵树。该辅助函数需要读写树外部的一些上下文，所以创建了一个 struct Paramblock 用指针指向它的以提高效率。

```
struct Symbol
{
   char *id;
   struct Symbol *left;
   struct Symbol *right;
};

struct Paramblock
{
   char *id;
   struct Symbol *sm;
};

static void membersearchx(struct Paramblock *p, struct Symbol *s)
{
   while (s)
   {
      if (strcmp(p->id,s->id) == 0)
      {
         if (p->sm)
            error("ambiguous member %s\n",p->id);
         p->sm = s;
      }

      if (s->left)
         membersearchx(p,s->left);
      s = s->right;
   }
}

struct Symbol *symbol_membersearch(Symbol *table[], int tablemax, char *id)
{
   struct Paramblock pb;
   int i;

   pb.id = id;
   pb.sm = NULL;
   for (i = 0; i < tablemax; i++)
```

```
  {
    membersearchx(pb, table[i]);
  }
  return pb.sm;
}
```

22.1.35.2 D 方式

这是同一个算法的 D 版本，代码量大大少于上一个版本。因为嵌套函数可以访问外围函数的变量，所以就不需要 Paramblock 或者处理它的簿记工作的细节了。嵌套的辅助函数完全处于使用它的函数的内部，提高了局部性和可维护性。

这两个版本的性能没什么差别。

```
class Symbol
{   char[] id;
   Symbol left;
   Symbol right;
}

Symbol symbol_membersearch(Symbol[] table, char[] id)
{   Symbol sm;

   void membersearchx(Symbol s)
   {
      while (s)
      {
         if (id == s.id)
         {
            if (sm)
               error("ambiguous member %s\n", id);
            sm = s;
         }

         if (s.left)
            membersearchx(s.left);
         s = s.right;
      }
   }

   for (int i = 0; i < table.length; i++)
   {
      membersearchx(table[i]);
   }
   return sm;
}
```

22.1.36 无符号右移

22.1.36.1 C 方式

如果左操作数是有符号整数类型，右移运算符 >> 和 >>= 表示有符号右移；如果左操作数是无符号整数类型，右移运算符 >> 和 >>= 表示无符号右移。如果要对 int 施行无符号右移，必须使用类型转换：

```
int i, j;
...
j = (unsigned)i >> 3;
```

如果 i 是一个 int，那么这个可以工作得很好。但如果 i 是一个使用 typedef 创建的类型，

```
myint i, j;
...
j = (unsigned)i >> 3;
```

并且 myint 碰巧是一个 long int，那么这个类型转换会悄无声息地丢掉最重要的那些 bit，给出一个不正确的结果。

22.1.36.2 D 方式

D 的右移运算符 >> 和 >>= 的行为同它们在 C 中的行为相同。但是 D 还支持显式右移运算符 >>> 和 >>>=，无论左操作数是否有符号，都会执行无符号右移。因此，

```
myint i, j;
...
j = i >>> 3;
```

避免了不安全的类型转换并且对于任何整数类型都能如你所愿的工作。

22.1.37 动态闭包

22.1.37.1 C 方式

考虑一下一个可再次使用的容器类型。为了可以重用，它需要支持这样一种方式——可以应用任意代码到该容器的每一个元素。通过创建一个 *apply* 函数来完成这个，该函数接受一个函数指针——传递的是容器内容里的每一个元素。

同时还需要一个相关指针，这里使用 void *p 表示。这里的例子是关于一个存放整型数组简单的容器类，以及那个计算这些整数最大值的容器的使用者。

```
void apply(void *p, int *array, int dim, void (*fp)(void *, int))
{
    for (int i = 0; i < dim; i++)
        fp(p, array[i]);
```

```
}

struct Collection
{
    int array[10];
};

void comp_max(void *p, int i)
{
    int *pmax = (int *)p;

    if (i > *pmax)
        *pmax = i;
}

void func(struct Collection *c)
{
    int max = INT_MIN;

    apply(&max, c->array, sizeof(c->array)/sizeof(c->array[0]), comp_max);
}
```

虽然这个也工作，但不够灵活。

22.1.37.2 D 方式

D 版本则利用 *委托(delegates)* 来为 *apply* 函数传送相关信息，而 *嵌套的函数* 则用于捕捉相关信息同时还提高方位性。

```
class Collection
{
    int[10] array;

    void apply(void delegate(int) fp)
    {
        for (int i = 0; i < array.length; i++)
            fp(array[i]);
    }
}

void func(Collection c)
{
    int max = int.min;

    void comp_max(int i)
    {
        if (i > max)
            max = i;
    }

    c.apply(comp_max);
}
```

指针没有了，类型转换以及泛型指针也不存在。D 版本完全是类型安全的。在 D 里另一种可行方法是利用 *函数字法*：

```
void func(Collection c)
```

```
{
    int max = int.min;

    c.apply(delegate(int i) { if (i > max) max = i; } );
}
```

不相关的函数名就不再需要创建。

22.1.38 个数可变型函数参数

它的任务是编写带有可变化数目参数的函数，例如参数总计那样的函数。

22.1.38.1 C 方式

```
#include <stdio.h>
#include <stdarg.h>

int sum(int dim, ...)
{   int i;
    int s = 0;
    va_list ap;

    va_start(ap, dim);
    for (i = 0; i < dim; i++)
        s += va_arg(ap, int);
    va_end(ap);
    return s;
}

int main()
{
    int i;

    i = sum(3, 8,7,6);
    printf("sum = %d\n", i);

    return 0;
}
```

对于这个有两个问题。每一个是 sum 函数需要知道提供了多少个实参。它不需要显式给出，而是通过跟编写的实际实参数目同步得到。第二个问题是没有什么方式来检查所提供的参数的类型，它们实际是整型，而非双精度、字符串、结构等等。

22.1.38.2 D 方式

紧跟的数组参数声明表示紧跟后面的参数会被收集一起组成一个数组。这些实参的类型会根据数组类型来进行检查，而实参的数目则变成了数组的属性：

```
import std.stdio;
```

```
int sum(int[] values ...)
{
    int s = 0;

    foreach (int x; values)
        s += x;
    return s;
}

int main()
{
    int i;

    i = sum(8,7,6);
    writefln("sum = %d", i);

    return 0;
}
```

第 23 章 针对 C++ 程序员的 D 编程

每个有经验的 C 程序员都积累了一系列的习惯和技术，这几乎成了第二天性。有时候，当学习一门新语言时，这些习惯会因为太令人舒适而使人看不到新语言中等价的方法。所以下面收集了一些常用的 C 技术，以及如何在 D 中完成同样的任务。

请参考：针对 C 程序员的 D 编程

23.1 定义构造函数

23.1.1 C++ 方式

构造函数跟类同名：

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

23.1.2 D 方式

构造函数用 this 关键字定义：

```
class Foo
{
       this(int x) { }
}
```

这个反映了它们在 D 里是怎么使用的。

23.2 基类初始化

23.2.1 C++ 方式

基类构造函数通过参数初始化列表语法调用。

```
class A { A() {... } };
class B : A
{
    B(int x)
       : A()            //   调用基类的构造函数
    {  ...
    }
};
```

23.2.2 D 方式

基类构造函数通过 super 语法来调用：

```
class A { this() { ... } }
class B : A
{
    this(int x)
    {  ...
       super();         // 调用基类的构造函数
       ...
    }
}
```

D 的方式优于 C++ 的地方在于可以灵活的在派生类的构造函数中的任何地方调用基类构造函数。D 还可以让一个构造函数调用另一个构造函数：

```
class A
{       int a;
       int b;
       this() { a = 7; b = foo(); }
       this(int x)
       {
          this();
          a = x;
```

```
        }
}
```

也可以在调用构造函数之前初始化成员，所以上面的例子等价于：

```
class A
{       int a = 7;
        int b;
        this() { b = foo(); }
        this(int x)
        {
            this();
            a = x;
        }
}
```

23.3 比较结构

23.3.1 C++ 方式

尽管 C++ 为结构赋值定义了一种简单、便捷的方法：

```
struct A x, y;
...
x = y;
```

却不支持结构之间的比较。因此，如果要比较两个结构实例之间的相等性：

```
#include <string.h>

struct A x, y;

inline bool operator==(const A& x, const A& y)
{
    return (memcmp(&x, &y, sizeof(struct A)) == 0);
}
...
if (x == y)
    ...
```

注意对于每个需要比较的结构来说，都要进行运算符重载，并且对运算符的重载会抛弃所有的语言提供的类型检查。C++ 的方式还有另一个问题，它不会检查 (x == y) 真正会发生什么，你不得不察看每一个被重载的 operator==() 以确定它们都做了些什么。

在 operator==() 的 memcmp() 实现里潜在着一个很龌龊的错漏。结构的分布中，由于对齐的原因，可能会有“空洞”。C++ 不保证这些空洞中为何值，所以两个不同的结构实例可能拥有所有成员的值都对应相等，但比较的结果却由于空洞中的垃圾的存在而为“不等”。

为了解决这个问题，operator==() 可以实现来进行成员方式（memberwise）的比较。不幸的是，这是不可靠的，因为 (1) 如果一个成员被加入到结构定义中，程序员可能会忘记同时把它加到 operator==() 中，(2) 对于浮点数的 nan 值来说，就算它们按位比较相等，比较的结果也是不等。

在 C++ 中没有可靠的解决方案。

23.3.2 D 方式

D 的方式直接而显然：

```
A x, y;
...
if (x == y)
    ...
```

23.4 创造新的 typedef 类型

23.4.1 C++ 方式

C++ 中的 typedef 很弱，也就是说，他们并不真正引入一个类型。编译器并不区分 typedef 类型和它底层的类型。

```
#define HANDLE_INIT     ((Handle)(-1))
typedef void *Handle;
void foo(void *);
void bar(Handle);

Handle h = HANDLE_INIT;
foo(h);                 // 未捕获的编码错漏
bar(h);                 // 正确
```

C++ 的解决方案是创建一个傀儡结构，目的是获得新类型才有的类型检查和重载能力。

```
#define HANDLE_INIT     ((void *)(-1))
struct Handle
{   void *ptr;

// 默认初始值
    Handle() { ptr = HANDLE_INIT; }

    Handle(int i) { ptr = (void *)i; }

// 转换为底层的类型
    operator void*() { return ptr; }
};
void bar(Handle);

Handle h;
bar(h);
h = func();
if (h != HANDLE_INIT)
    ...
```

23.4.2 D 方式

不需要上面那样的习惯构造。只需要写成：

```
typedef void* Handle = cast(void*)-1;
void bar(Handle);

Handle h;
bar(h);
h = func();
if (h != Handle.init)
    ...
```

注意，可以给 typedef 提供一个默认的初始值作为新类型的初始值。

23.5 友元

23.5.1 C++ 方式

有时两个类关系很紧密，它们之间不是继承关系，但是它们需要互相访问对方的私有成员。这个是使用 `friend` 声明来完成的：

```
class A
{
   private:
       int a;

   public:
       int foo(B *j);
       friend class B;
       friend int abc(A *);
};

class B
{
   private:
       int b;

   public:
       int bar(A *j);
       friend class A;
};

int A::foo(B *j) { return j->b; }
int B::bar(A *j) { return j->a; }

int abc(A *p) { return p->a; }
```

23.5.2 D 方式

在 D 中，对于凡是同一个模块的成员，友元访问都是隐式存在的。这样做是有道理的，因为关系紧密地类应该位于同一个模块中，所以隐式地赋予位于同一个模块中的其他类友元访问权限是很自然的：

```
module X;

class A
{
   private:
      static int a;

   public:
      int foo(B j) { return j.b; }
}

class B
{
   private:
      static int b;

   public:
      int bar(A j) { return j.a; }
}

int abc(A p) { return p.a; }
```

private 禁止其他模块中访问这些成员。

23.6 运算符重载

23.6.1 C++ 方式

假设有一个结构代表了一种新的算术类型，将其的运算符重载以使其可以和整数比较是很方便的：

```
struct A
{
       int operator <  (int i);
       int operator <= (int i);
       int operator >  (int i);
       int operator >= (int i);
};

int operator <  (int i, A &a) { return a >  i; }
int operator <= (int i, A &a) { return a >= i; }
int operator >  (int i, A &a) { return a <  i; }
int operator >= (int i, A &a) { return a <= i; }
```

所有的 8 个函数缺一不可。

23.6.2 D 方式

D 认识到比较运算符在根本上互相之间是有联系的。所以只用一个函数是必需的：

```
struct A
{
       int opCmp(int i);
}
```

编译器按照 cmp 函数自动解释 <、<=、> 和 >= 运算符，同时处理左操作数不是对象引用的情况。

类似这样的明智的规则也适用于其他的运算符重载，这就使得 D 中的运算符重载不像在 C++ 中那样繁琐且易于出错。只需要少得多的代码，就可以达到相同的效果。

23.7 using 声明名字空间

23.7.1 C++ 方式

在 C++ 里的 *using-声明* 用来从一个名字空间作用域将名字引入当前的作用域：

```
namespace foo
{
    int x;
}
using foo::x;
```

23.7.2 D 方式

D 用模块来代替名字空间和 #include 文件，用别名声明来代替 using 声明：

```
/** Module foo.d **/
module foo;
int x;

/** Another module **/
import foo;
alias foo.x x;
```

别名比简单的 using 声明灵活得多。别名可以用来重命名符号，引用模板成员，引用嵌套类类型等。

23.8 RAII (资源获得即初始化)

23.8.1 C++ 方式

在 C++ 中，资源如内存等，都需要显式的处理。因为当退出当前作用域时会自动调用析构函数，RAII 可以通过将资源释放代码放进析构函数中实现：

```
class File
{   Handle *h;

    ~File()
    {
```

```
        h->release();
    }
};
```

23.8.2 D 方式

大多数的资源释放问题都是简单的跟踪并释放内存。在 D 中这是由垃圾收集程序自动完成的。除了内存外，用得最普遍的资源要数信号量和锁了，在 D 中可用 `synchronized` 声明和语句自动管理。

几乎没留下什么 RAII 问题供 *scope* 类处理。Scope 类退出其作用域时，会调用它们的析构函数。

```
scope class File
{   Handle h;

    ~this()
    {
        h.release();
    }
}

void test()
{
    if (...)
    {   scope f = new File();
        ...
} // 在大括号结束处，f.~this() 会得到运行，即使
     // 通过抛出的异常，退出了 scope
}
```

23.9 特性

23.9.1 C++ 方式

人们常常会定义一个域，同时为它提供面向对象的 get 和 set 函数：

```
class Abc
{
  public:
    void setProperty(int newproperty) { property = newproperty; }
    int getProperty() { return property; }

  private:
    int property;
};
```

```
Abc a;
a.setProperty(3);
int x = a.getProperty();
```

所有这些都不过是增加了击键的次数而已，并且还会使代码变得不易阅读，因为其中充满了 getProperty() 和 setProperty() 调用。

23.9.2 D 方式

属性可以使用正常的域语法 get 和 set，然后 get 和 set 会被编译器用方法调用取代。

```
class Abc
{
// 设置
   void property(int newproperty) { myprop = newproperty; }

// 获取
   int property() { return myprop; }

  private:
   int myprop;
}
```

像这样使用它：

```
Abc a;
a.property = 3;          // 等同于 a.property(3)
int x = a.property;      // 等同于 int x = a.property()
```

因此，在 D 中属性可以被看作一个简单的域名。开始时，属性可以只是一个简单的域名，但是如果后来需要将读取和设置行为改变为函数调用，只需要改动类的定义就够了。这样就避免了定义 get 和 set 时敲入冗长的代码，仅仅是为了‘谨防’日后派生类有可能需要重写(override)它们。这也是一种定义接口类的方法，这些类没有数据域，只在语法上表现得好像它们作了实际工作。

23.10 递归模板(Recursive Templates)

23.10.1 C++ 方式

种使用模板的高级方式是递归的扩展它们，依靠特化来终止递归。用来计算阶乘的模板可能会是这样：

```
template<int n> class factorial
{
   public:
      enum { result = n * factorial<n - 1>::result };
};

template<> class factorial<1>
{
   public:
```

```
        enum { result = 1 };
};

void test()
{
printf("%d\n", factorial<4>::result); // 输出 24
}
```

23.10.2 D 方式

D 的版本与之相似，但是简单一点，利用了将单一模板成员提升到外围的名字空间的能力：

```
template factorial(int n)
{
    enum { factorial = n * .factorial!(n-1) }
}

template factorial(int n : 1)
{
    enum { factorial = 1 }
}

void test()
{
    writefln("%d", factorial!(4));       // 输出 24
}
```

23.11 元模板

问题：为一个有符号整数类型（大小至少有 *n 位*）创建一个 typedef

23.11.1 C++ 方式

这个例子改编自 Carlo Pescio 博士在 Template Metaprogramming:Make parameterized integers portable with this novel technique 里所写的一个实例，并被简化了。

在 C++ 里没办法完成基于一个在模板形参里的表达式结果的条件编译，因此，所有的控制流都源自模板实参的模式匹配，防止各种显式的模板特例化。更糟的是，没有什么方式来完成基于像“小于或等于”那种关系的模板特例化，因此，这个例子使用了一种更聪明的技术——模板被递归扩展，每一次模板值参都加 1，直到匹配到一个特例。如果没有匹配，最后结果就是无用的递归编译器栈溢出或内部错误，或者奇怪的语法错误。

还需要一个预处理器宏弥补模板 typedef 的缺失。

```
#include <limits.h>

template< int nbits > struct Integer
{
```

```
    typedef Integer< nbits + 1 > :: int_type int_type ;
} ;

struct Integer< 8 >
{
    typedef signed char int_type ;
} ;

struct Integer< 16 >
{
    typedef short int_type ;
} ;

struct Integer< 32 >
{
    typedef int int_type ;
} ;

struct Integer< 64 >
{
    typedef long long int_type ;
} ;

// 如果需要的大小不被支持的话，元程序
// 就会把计数器加大，直到有“内部错误(internal error)”发生，
// 或者达到了 INT_MAX。INT_MAX
// 特例化没有定义 int_type，因此
// 编译错误就总是会产生
struct Integer< INT_MAX >
{
} ;

// 让语句漂亮点
#define Integer( nbits ) Integer< nbits > :: int_type

#include <stdio.h>

int main()
{
    Integer( 8 ) i ;
    Integer( 16 ) j ;
    Integer( 29 ) k ;
    Integer( 64 ) l ;
    printf("%d %d %d %d\n",
        sizeof(i), sizeof(j), sizeof(k), sizeof(l));
    return 0 ;
}
```

23.11.2 C++ Boost 方式

这个版本使用了 C++ Boost 库。它由 David Abrahams 提供。

```
#include <boost/mpl/if.hpp>
#include <boost/mpl/assert.hpp>

template <int nbits> struct Integer
    : mpl::if_c<(nbits <= 8), signed char
```

```
    , mpl::if_c<(nbits <= 16), short
    , mpl::if_c<(nbits <= 32), long
    , long long>::type >::type >
{
    BOOST_MPL_ASSERT_RELATION(nbits, <=, 64);
}

#include <stdio.h>

int main()
{
    Integer< 8 > i ;
    Integer< 16 > j ;
    Integer< 29 > k ;
    Integer< 64 > l ;
    printf("%d %d %d %d\n",
        sizeof(i), sizeof(j), sizeof(k), sizeof(l));
    return 0 ;
}
```

23.11.3 D 方式

D 版本也可以使用递归模板来编写，不过还有更好的方式。不像 C++ 例子，这个很容易就能想像出它是怎样进行的。它编译很快，而且如果失败的话，也能给出很有判断力的编译时信息。

```
import std.stdio;

template Integer(int nbits)
{
    static if (nbits <= 8)
        alias byte Integer;
    else static if (nbits <= 16)
        alias short Integer;
    else static if (nbits <= 32)
        alias int Integer;
    else static if (nbits <= 64)
        alias long Integer;
    else
        static assert(0);
}

int main()
{
    Integer!(8) i ;
    Integer!(16) j ;
    Integer!(29) k ;
    Integer!(64) l ;
    writefln("%d %d %d %d",
        i.sizeof, j.sizeof, k.sizeof, l.sizeof);
```

```
    return 0;
}
```

23.12 类型特征

类型特征(Type Trait)是又一个能够在编译时找出类型属性的术语。

23.12.1 C++ 方式

下面的模板来自 C++ Templates:The Complete Guide, David Vandevoorde, Nicolai M. Josuttis 第 353 页，它用于判断模板实参类型是否是个函数：

```
template<typename T> class IsFunctionT
{
    private:
        typedef char One;
        typedef struct { char a[2]; } Two;
        template<typename U> static One test(...);
        template<typename U> static Two test(U (*)[1]);
    public:
        enum { Yes = sizeof(IsFunctionT<T>::test<T>(0)) == 1 };
};

void test()
{
    typedef int (fp)(int);

    assert(IsFunctionT<fp>::Yes == 1);
}
```

此模板依赖于 SFINAE (置换失败不是错误) 原则。这个为什么会有效是一个相当高级的模板话题。

23.12.2 D 方式

SFINAE (Substitution Failure Is Not An Error - 置换失败不是错误) 在 D 里可以在不求助于模板参数模式匹配的情况下完成：

```
template IsFunctionT(T)
{
    static if ( is(T[]) )
        const int IsFunctionT = 0;
    else
        const int IsFunctionT = 1;
}

void test()
{
    typedef int fp(int);

    assert(IsFunctionT!(fp) == 1);
}
```

像这样一个发现某个类型是否是一个函数的任务根本就不需要模板，也不需要借口试图创建非法的函数类型数组。*Is 表达式* 可以直接测试它：

```
void test()
{
    alias int fp(int);

    assert( is(fp == function) );
}
```

第 24 章 C 预处理器 vs D

回到 C 被发明的那个时候，编译器技术是很初级的。在前台里安装一个文本宏预处理器是一种添加大量强大功能的直接而容易的方式。程序规模和复杂度的增加已经证明伴随这些功能有着许多的天生的毛病。D 没有预处理器，但 D 提供了更为灵活的方式来解决这个同样的问题。

24.1 头文件

24.1.1 C 预处理方式

C 和 C++ 大量依赖于头文件的文本包含。这个经常导致编译器必须为每一个源文件一遍又一遍地重复编译几十个上千行的代码，这显然是增长编译时间的源头。头文件通常的用途不外乎是完成符号的，而非文本内容的插入。可以使用“import”语句来完成。符号包含表示编译器只装载一个已经编译好的符号表。对宏“包裹器(wrapper)”禁止多重“#inclusion”、恶心的“#pragma once”语法的需要，以及对用于预编译头文件的让人难于理解且极其脆弱的语法的需要，对于 D 来说，都是不必和无关的。

```
#include <stdio.h>
```

24.1.2 D 方式

D 使用符号导入(import)：

```
import std.c.stdio;
```

24.2 #pragma once

24.2.1 C 预处理方式

C 头文件经常需要被保护，以防被包含(#include)多次。为了这个目的，头文件将包含这样一行：

```
#pragma once
```

或者，更方便移植的版本：

```
#ifndef __STDIO_INCLUDE
#define __STDIO_INCLUDE
... header file contents
#endif
```

24.2.2 D 方式

完全没有必要，因为 D 完成的是导入文件的符号包含；它们只会被导入一次，无论导入声明出现了多少次。

24.3 #pragma pack

24.3.1 C 预处理方式

在 C 里这个用来调整结构的对齐。

24.3.2 D 方式

对于 D 类，是没有必要调整对齐的（实际上，编译器是自由重新调整了数据段，以求获得优化的排列，很像编译器会重新调整在堆栈帧里局部变量）。对于那些映像到外部定义的数据结构上的 D 结构，是需要的，并且可以使用下面的方式来处理：

```
struct Foo
{
       align (4):       // 使用 4 字节对齐
       ...
}
```

24.4 宏

预处理宏为 C 添加了强大的功能和灵活性。不过它们也有不足：

- 宏没有域的概念；它们从定义点开始直到源码的最后这一段范围都有效。它们在 .h 文件、嵌套的代码等等之间任意穿行(cut a swath)。在包含了几十个上千行的宏定义时，想要避免宏扩展不出错将是困难重重的。
- 调试器并不知道宏。在调试一个带在符号数据的程序时，使用只知道宏扩展，而不知道宏本身的调试器是不行的。
- 宏使得“特征化(tokenize)”源代码变得不可能，因为前面的宏变化更改会任意地重新生成新的“特征符(token)”。
- 基于纯文本内容的宏导致了随意和不一致的使用格式，同时使得使用宏的代码易于出

错。（在 C++ 里有些使用模板解决这个问题的尝试。）

- 宏仍然被用来弥补语言表达能力的不足，如头文件相关的“包裹器(wrapper)”。

下面例举的是宏的一些常见使用方法，以及它们在 D 里所对应的功能：

1. 定义文字常量：

C 预处理方式

```
#define VALUE   5
```

D 方式

```
const int VALUE = 5;
```

2. 创建一个值或标志列表：

C 预处理方式

```
int flags:
#define FLAG_X  0x1
#define FLAG_Y  0x2
#define FLAG_Z  0x4
...
flags |= FLAG_X;
```

D 方式

```
enum FLAGS { X = 0x1, Y = 0x2, Z = 0x4 };
FLAGS flags;
...
flags |= FLAGS.X;
```

3. 区分 ascii 字符和 wchar 字符：

C 预处理方式

```
#if UNICODE
    #define dchar       wchar_t
    #define TEXT(s)     L##s
#else
    #define dchar       char
    #define TEXT(s)     s
#endif

...
dchar h[] = TEXT("hello");
```

D 方式

```
dchar[] h = "hello";
```

D 的优化器会内嵌(inline)访函数，并且会在编译时转换该字符串常量。

4. 对以往编译器的支持：

C 预处理方式

```
#if PROTOTYPES
#define P(p)    p
#else
#define P(p)    ()
#endif
int func P((int x, int y));
```

D 方式

通过让 D 编译器打开放源码，它会很大程度上避免语句向后兼容的问题。

5. 类型别名：

C 预处理方式

```
#define INT     int
```

D 方式

```
alias int INT;
```

6. 对于声明和定义使用一个头文件：

C 预处理方式

```
#define EXTERN extern
#include "declarations.h"
#undef EXTERN
#define EXTERN
```

```
#include "declarations.h"
```

在 declarations.h 里：

```
EXTERN int foo;
```

D 方式

声明和定义是相同的，因此没有必要搞乱类的存储类别(storage class)来从同一个源码里生成声明和定义。

7. 轻型的内嵌函数：

C 预处理方式

```
#define X(i)    ((i) = (i) / 3)
```

D 方式

```
int X(ref int i) { return i = i / 3; }
```

编译器优化器会内嵌它；不会有效率损失。

8. “断言”函数的文件和行号信息：

C 预处理方式

```
#define assert(e)       ((e) || _assert(__LINE__, __FILE__))
```

D 方式

assert() 是一个基本内建的表达式。让编译器知道 assert()，也就使得优化器可以知道那些像 _assert()（函数绝不会返回的）那样的东西。

9. 设置函数调用协定：

C 预处理方式

```
#ifndef _CRTAPI1
#define _CRTAPI1 __cdecl
#endif
#ifndef _CRTAPI2
#define _CRTAPI2 __cdecl
#endif
```

```
int _CRTAPI2 func();
```

D 方式

调用协定可以以块的形式被特列化，因此没有必要为每一个函数进行更改：

```
extern (Windows)
{
    int onefunc();
    int anotherfunc();
}
```

10.隐藏古怪的 __near 或 __far 指针：

C 预处理方式

```
#define LPSTR   char FAR *
```

D 方式

D 不支持 16 位代码、混合的指针大小，以及不同类型的指针，因此这个问题跟 D 无关。

11.简单的泛型编程：

C 预处理方式

基于文本替换，选择使用哪个函数呢：

```
#ifdef UNICODE
int getValueW(wchar_t *p);
#define getValue getValueW
#else
int getValueA(char *p);
#define getValue getValueA
#endif
```

D 方式

D 支持属于其它符号的 *别名* 的符号声明：

```
version (UNICODE)
{
    int getValueW(wchar[] p);
    alias getValueW getValue;
}
else
{
    int getValueA(char[] p);
    alias getValueA getValue;
}
```

24.5 条件编译

24.5.1 C 预处理方式

条件编译是 C 预处理器的一个强大功能，但它有不足：

- 预处理器没有域的概念。#if/#endif 能够以一种完全无结构和紊乱的方式跟代码交叉，

这使得事情难于理解。
- 条件编译会引起可能跟在程序中使用的标记符相冲突的“宏-宏(macros - macros)”。
- “#if 表达式”是以跟 C 表达式略微不同的方式进行求值的。
- 预处理器语言在概念上是跟 C 根本不一样的，例如：空格和行终止符对于预处理器都是有意义的，而在 C 里却没有。

24.5.2 D 方式

D 支持条件编译：

1. 把特定的功能版本分割成单独的模块。
2. 调用语句用于启用/禁用调试功能、额外的输出等等。
3. version 语句用于处理由单套源码生成的程序的多个版本。
4. if (0) 语句。
5. 嵌套注释 /+ +/ 可以用来注释掉多块代码。

24.6 代码分解(Code Factoring)

24.6.1 C 预处理方式

让代码的重复部分在多个地方可以执行，这个在函数里很普遍的。性能上的考虑没有将它分解出来成为一个单独的函数，因此使用宏来实现它。例如，考虑一下这个来自一个字节代码解释器的片段：

```
unsigned char *ip;      // 字节代码指令指针
int *stack;
int spi;                // 栈指针
...
#define pop()           (stack[--spi])
#define push(i)         (stack[spi++] = (i))
while (1)
{
    switch (*ip++)
    {
        case ADD:
            op1 = pop();
            op2 = pop();
            result = op1 + op2;
            push(result);
            break;

        case SUB:
        ...
    }
}
```

这个受着大量的问题的困扰：

1. 宏必须计算表达式而且不能声明任何变量。想想扩展它们来检查栈的溢出/下溢的难度吧。

2. 宏存在于语义符号表之外，因此它们留存在域中，甚至还位于声明它们的函数的外面。
3. 宏的参数是文本方式传递的，而不是通过值传递，即表示宏实现需要相当小心，以防多次使用宏参数，并且要使用括号“()”来保护它。
4. 宏对于调试器是不可见的，调试器只知道扩展的表达式。

24.6.2 D 方式

D 使用嵌套函数整洁地处理了这个问题：

```
ubyte* ip;               // 字节代码指令指针
int[] stack;             // 操作数栈
int spi;                 // 栈指针
...

int pop()         { return stack[--spi]; }
void push(int i) { stack[spi++] = i; }

while (1)
{
    switch (*ip++)
    {
        case ADD:
            op1 = pop();
            op2 = pop();
            push(op1 + op2);
            break;

        case SUB:
        ...
    }
}
```

处理的问题有：

1. 嵌套函数拥有 D 函数完整的表达能力。数组讯问已经是边界检查的（可以通过编译时的开关调整）。
2. 嵌套函数名跟其它任何名字一个是被限定在域里的。
3. 由于参数通过值来传递，因此需要考虑一下参数表达式里的副作用。
4. 嵌套函数对于调试器是可见的。

另外，嵌套函数可以被实现内嵌，这样就使得它拥有跟 C 的宏版本所展现的一样的高性能。

24.7 #error 和 Static Asserts

Static asserts(静态断言)是用户定义的在编译时执行的检查；如果检查失败，则编译就会出现

一个错误，然后失败。

24.7.1 C 预处理方式

第一种方式是使用 `#error` 处理块：

```
#if FOO || BAR
    ... code to compile ...
#else
#error "there must be either FOO or BAR"
#endif
```

这个在预处理的表达式（如：仅仅是整数常量表达式，无 cast，无 `sizeof`，无符号常量等等）里有着先天的局限性。

这些问题可以通过定义 `static_assert` 宏（感谢 M. Wilson）在一定程度上绕开：

```
#define static_assert(_x) do { typedef int ai[(_x) ? 1 : 0]; } while(0)
```

并且像这样使用它：

```
void foo(T t)
{
    static_assert(sizeof(T) < 4);
    ...
}
```

这个是有效，不过如果条件求值得 false 的话，会引起编译时语义错误。这个技术的局限性在于有时会跟编译器的错误信息混淆，同时对于在函数体外使用 `static_assert` 也能为力。

24.7.2 D 方式

D 有“static assert(静态断言)”，它可以用在任何声明或语句可以使用的地方。例如：

```
version (FOO)
{
    class Bar
    {
        const int x = 5;
        static assert(Bar.x == 5 || Bar.x == 6);

        void foo(T t)
        {
            static assert(T.sizeof < 4);
            ...
        }
    }
}
else version (BAR)
{
    ...
}
else
{
```

```
    static assert(0);    // 不支持的版本
}
```

24.8 模板混入

D 的“模板混入(template mixins)”表面看起来就象使用 C 的预处理器来插入多块代码，并且在实例化它们的那个域里解析它们。但是“混入”的好处远胜于宏：

1. “混入”是在传递带有语言语法集合的解析后的声明树里的进行替换；而宏是在没有任何组织的随意的预处理特征符里的进行替换。
2. “混入”在同一种语言里。宏是独立且独特的位于 C++ 之上的语言，它有着自己的表达式规则，自己的类型，自己独特的符号表，自己的域规则等等。
3. “混入”基于部分的特殊化规则进行选择，而宏没有重载。
4. “混入”创建了一个域，而宏没有。
5. “混入”跟语法解析工具兼容，而宏不是。
6. “混入”的语义信息和符号表都可以被传递给调试器，而在转换过程中被丢失。
7. “混入”有重写(override)冲突解决规则，而宏只有冲突。
8. “混入”会自动创建唯一的标识，而这个需要使用一个标准的算法；宏必须手工完成它，同时特征符都杂乱不堪。
9. 带有副作用的“混入”的值参只被计算一次；而宏的值参在每一个被用来扩展的地方都需要被计算一下（常导致怪异的错漏）。
10. “混入”实参替换不需要使用圆括号将它“保护”起来，以避免操作符优先顺序重组。
11. “混入”可以像一般的 D 代码那样输入，即允许可以任意长度；而多行宏则必须使用反斜线进行行分割，而且不能使用“//”来做为行尾注释，等等。
12. “混入”可以定义其它的“混入”。宏不参创建其它的宏。

第 25 章 D 字符串 vs C++ 字符串

为什么 D 把字符串内建到语言核心中而不是像 C++ 那样将其完全放到库中呢？出发点是什么？有什么改进呢？

25.1 连接运算符

C++ 字符串依赖于对现有运算符的重载。显而易见的选择是用 += 和 + 作为连接运算符。但如果一个人仅仅查看代码，他会看到 + 并认为是“相加”操作。他需要查看相关类型（而类型通常被多重的 typedef 所埋葬）才能知道这是一个字符串类型，而这个操作不是字符串相加而是连接字符串。

另外，如果操作数是一个 float 数组，那么‘+’是应该被重载为向量相加呢，还是数组的连接？

在 D 中，引入了一个新的二元运算符‘~’作为连接运算符，这样就避免了上面的问题。连接运算符用于数组（字符串是数组的一个子集）。‘~=’是相应的追加运算符。如果操作数是 float 数组，‘~’会执行连接操作，‘+’会执行向量加法。加入一个新的运算符使得我们可以正交而一致地处理数组。（在 D 中，字符串就是字符串的数组，而不是某个特殊的类型。）

25.2 同 C 字符串语法的互用性

只有在其中一个操作数是可重载的时候才能真正采用运算符重载。所以 C++ 字符串不能完全处理任意的包含字符串的表达式。请考虑：

```
const char abc[5] = "world";
string str = "hello" + abc;
```

上面的代码不能通过编译。但是当语言核心支持字符串时情况就不同了：

```
const char[5] abc = "world";
char[] str = "hello" ~ abc;
```

25.3 对 C 字符串语法的延续

在 C++ 中有三种方法可以获得字符串的长度：

```
const char abc[] = "world";     :       sizeof(abc)/sizeof(abc[0])-1
                                :       strlen(abc)
string str;                     :       str.length()
```

这种不一致性使得编写泛型模版十分困难。看看 D：

```
char[5] abc = "world";  :       abc.length
char[] str              :       str.length
```

25.4 检查空数组

C++ 字符串使用一个函数检测一个字符串是否为空：

```
string str;
if (str.empty())
       // 字符串为空
```

在 D 中，空数组的长度为零：

```
char[] str;
if (!str.length)
       // 字符串为空
```

25.5 改变现有数组的大小

C++ 通过 resize() 成员函数处理这种情况：

```
string str;
str.resize(newsize);
```

D 利用了 str 是字符串这一知识，改变它的大小仅仅需要改变 length 特性：

```
char[] str;
str.length = newsize;
```

25.6 分割字符串

C++ 使用特殊的构造函数分割现有的字符串：

```
string s1 = "hello world";
string s2(s1, 6, 5);            // s2 是 "world"
```

D 支持数组分割语法，这在 C++ 中是不可能的：

```
char[] s1 = "hello world";
char[] s2 = s1[6 ..11]; // s2 为 "world"
```

当然，D 中所有的数组都支持分割，而不仅仅是字符串。

25.7 复制字符串

C++ 使用 replace 函数复制字符串：

```
string s1 = "hello world";
string s2 = "goodbye      ";
s2.replace(8, 5, s1, 6, 5);     // s2 is "goodbye world"
```

D 将分割语法作为左值：

```
char[] s1 = "hello world";
char[] s2 = "goodbye      ".dup;
s2[8..13] = s1[6..11];          // s2 为 "goodbye world"
```

需要 .dup 是因为，在 D 里字符串文字是只读的；而 .dup 会创建一个可写入的复本。

25.8 转换为 C 字符串

这是为兼容 C API 所需要的。在 C++ 中，需要使用 c_str() 成员函数：

```
void foo(const char *);
string s1;
foo(s1.c_str());
```

在 D 里，字符串可以使用 .ptr 特性将其转换成 char*：

```
void foo(char*);
char[] s1;
foo(s1.ptr);
```

为了让这个可以工作，此处的 foo 期望一个以 0 作为终止的字符串，s1 必须要求是以 0 作为终止。 另一种方法，就是使用函数 std.string.toStringz 来确保它：

```
void foo(char*);
char[] s1;
foo(std.string.toStringz(s1));
```

25.9 数组边界检查(Array Bounds Checking)

在 C++ 中，没有针对[]的边界检查。在 D 中，数组边界检查默认是打开的，并且可以在结束调试工作后通过编译器的选项关掉。

25.10 字符串 Switch 语句

这在 C++ 中是不可能的，可没有办法同过扩展库来完成。在 D 中，你可以使用最自然的语法形式：

```
switch (str)
{
   case "hello":
   case "world":
      ...
}
```

其中 str 可以是任何字符串：定长字符串数组，如 char[10] ；或者动态字符串，如 char[] 。当然，一个高质量的实现可以基于 case 中的字符串采用各种策略高效地实现这个功能。

25.11 填充字符串

在 C++ 中，这个功能由 replace() 成员函数实现：

```
string str = "hello";
```

```
str.replace(1,2,2,'?');          // str 是 "h??lo"
```

在 D 中，以一种很自然的方式来使用数组分割语法：

```
char[5] str = "hello";
str[1..3] = '?';                 // str 是 "h??lo"
```

25.12 值传(Value) vs 引用(Reference)

C++ 字符串，如 STLport 所实现的，靠的就是值传(value)，而且是以‘\0’结尾。[虽然后者是一种实现决择(implementation choice)，但是 STLport 似乎是最为流行的实现。]此种情况，再加上没有垃圾回收，就会有这样一些的后果。首先，所创建的任何字符串都必须保留它自己一份数据拷贝。必须跟踪字符串数据的“拥有者”，因为如果拥有者被删除的话，所有的引用都会失效。如果有人试图通过将字符串处理为值类型以避免悬挂引用的话，就会为大量的内存分配，数据复制及内存释放付出代价。其次，以 0 结尾就意味着字符串不能引用其它字符串。在数据段、堆栈等处的字符串是不能引用的。

D 字符串是引用类型，并且内存是由垃圾收集程序管理的。这就意味着只需要复制引用而不用复制字符串数据。D 字符串可以引用静态数据段、堆栈、其它字符串、对象、文件缓冲等处的数据。没有必要跟踪字符串数据的“拥有者”。

一个显然的问题是如果多个 D 字符串指向同一个字符串数据，修改数据会造成什么后果？答案是所有的引用都会指向修改后的数据。它造成的后果不同于 C++ 的后果——如果 C++ 允许“写时复制”协定的话，就可以避免这种差异。而如果 C++ 采用“写时复制”技术的话，就可以避免这种差异。

因为 D 字符串是引用而且采用垃圾收集，所以对于那些主要是处理字符串的程序，如 lzw 压缩程序，既可以节省内存，又可以提高速度。

25.13 基准评测

让我们来看一个小工具——wordcount，它用于统计文本文件中每个单词的出现频率。在 D 中，它是像这个样子：

```
import std.file;
import std.stdio;

int main (char[][] args)
{
    int w_total;
    int l_total;
    int c_total;
    int[char[]] dictionary;

    writefln("   lines   words   bytes file");
    for (int i = 1; i < args.length; ++i)
    {
```

```
    char[] input;
    int w_cnt, l_cnt, c_cnt;
    int inword;
    int wstart;

    input = cast(char[])std.file.read(args[i]);

    for (int j = 0; j < input.length; j++)
    {   char c;

        c = input[j];
        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]++;
            inword = 0;
        }
        ++c_cnt;
    }
    if (inword)
    {   char[] w = input[wstart .. input.length];
        dictionary[w]++;
    }
    writefln("%8s%8s%8s %s", l_cnt, w_cnt, c_cnt, args[i]);
    l_total += l_cnt;
    w_total += w_cnt;
    c_total += c_cnt;
}

if (args.length > 2)
{
    writefln("--------------------------------------%8s%8s%8s total",
        l_total, w_total, c_total);
}

writefln("--------------------------------------");

foreach (char[] word1; dictionary.keys.sort)
```

```
    {
        writefln("%3d %s", dictionary[word1], word1);
    }
    return 0;
}
```

(另一种实现 方法，它使用了缓冲文件 I/O 来处理大型文件。)

两个人使用 C++ 采用 C++ 标准模板库分别实现了两个程序：wccpp1 和 wccpp2。输入文件 alice30.txt 的内容是《爱丽丝漫游仙境》。D 编译器 dmd 和 C++ 编译器 dmc，都使用相同的优化程序和相同的代码生成程序。这样我们就可以集中精力比较语言语义的效率而不用管优化和代码生成的先进程度。测试运行在一台 Win XP 机器上。 dmc 使用 STLport 来实现模版。

程序	编译	编译用时	运行	运行用时
D wc	dmd wc -O -release	0.0719	wc alice30.txt >log	0.0326
C++ wccpp1	dmc wccpp1 -o -I\dm\stlport\stlport	2.1917	wccpp1 alice30.txt >log	0.0944
C++ wccpp2	dmc wccpp2 -o -I\dm\stlport\stlport	2.0463	wccpp2 alice30.txt >log	0.1012

下面的测试运行在 linux，再次将 D 编译器(gdc) 和 C++ 编译器 (**g++**) 进行了比较，他们都使用共同的优化器和代码生成器。该系统是 Pentium III 800MHz，运行有 RedHat Linux 8.0 和 gcc 3.4.2。linux 版本的 Digital Mars D 编译器 (**dmd**) 也包含在比较中。

程序	编译	编译用时	运行	运行用时
D wc	gdc -O2 -frelease -o wc wc.d	0.326	wc alice30.txt > /dev/null	0.041
D wc	dmd wc -O -release	0.235	wc alice30.txt > /dev/null	0.041
C++ wccpp1	g++ -O2 -o wccpp1 wccpp1.cc	2.874	wccpp1 alice30.txt > /dev/null	0.086
C++ wccpp2	g++ -O2 -o wccpp2 wccpp2.cc	2.886	wccpp2 alice30.txt > /dev/null	0.095

下面 gdc 同 g++ 的比较使用的环境是：一台 PowerMac G5 2x2.0GHz，运行 MacOS X 10.3.5 and gcc 3.4.2。 (计时不是那么准确。)

程序	编译	编译用时	运行	运行用时
D wc	gdc -O2 -frelease -o wc wc.d	0.28	wc alice30.txt > /dev/null	0.03
C++ wccpp1	g++ -O2 -o wccpp1 wccpp1.cc	1.90	wccpp1 alice30.txt > /dev/null	0.07
C++ wccpp2	g++ -O2 -o wccpp2 wccpp2.cc	1.88	wccpp2 alice30.txt > /dev/null	0.08

25.13.1.1 wccpp2 by Allan Odgaard

```
#include <algorithm>
#include <cstdio>
#include <fstream>
#include <iterator>
#include <map>
#include <vector>

bool isWordStartChar (char c)   { return isalpha(c); }
bool isWordEndChar (char c)     { return !isalnum(c); }

int main (int argc, char const* argv[])
{
   using namespace std;
   printf("Lines Words Bytes File:\n");

   map<string, int> dict;
   int tLines = 0, tWords = 0, tBytes = 0;
   for(int i = 1; i < argc; i++)
   {
      ifstream file(argv[i]);
      istreambuf_iterator<char> from(file.rdbuf()), to;
      vector<char> v(from, to);
      vector<char>::iterator first = v.begin(), last = v.end(), bow, eow;

      int numLines = count(first, last, '\n');
      int numWords = 0;
      int numBytes = last - first;

      for(eow = first; eow != last; )
      {
         bow = find_if(eow, last, isWordStartChar);
         eow = find_if(bow, last, isWordEndChar);
         if(bow != eow)
            ++dict[string(bow, eow)], ++numWords;
      }

      printf("%5d %5d %5d %s\n", numLines, numWords, numBytes, argv[i]);

      tLines += numLines;
      tWords += numWords;
      tBytes += numBytes;
   }

   if(argc > 2)
         printf("-----------------------\n%5d %5d %5d\n", tLines, tWords,
tBytes);
   printf("-----------------------\n\n");

   for(map<string, int>::const_iterator it = dict.begin(); it != dict.end(); +
+it)
         printf("%5d %s\n", it->second, it->first.c_str());

   return 0;
}
```

第 26 章 D 的复数类型和 C++ 的 std::complex

D 的复数是怎么样跟 C++ 的 std::complex 类比较的?

26.1 语句美感

在 C++ 里，复数类型是：

```
complex<float>
complex<double>
complex<long double>
```

C++ 没有明确的虚数类型 D 有 3 复数类型和 3 种虚数类型：

```
cfloat
cdouble
creal
ifloat
idouble
ireal
```

C++ 复数类型可以结合算术符号一起使用，不过因为没有虚数类型，因此只有通过构造语法来创建虚数：

```
complex<long double> a = 5;              // a = 5 + 0i
complex<long double> b(0,7);             // b = 0 + 7i
c = a + b + complex<long double>(0,7);   // c = 5 + 14i
```

在 D 里，虚数字法有'i'后缀可以使用。对应的代码也就变得更自然：

```
creal a = 5;             // a = 5 + 0i
ireal b = 7i;            // b = 7i
c = a + b + 7i;          // c = 5 + 14i
```

更多包含有变量的表达式：

```
c = (6 + 2i - 1 + 3i) / 3i;
```

在 C++ 里，这个变成了：

```
c = (complex<double>(6,2) + complex<double>(-1,3)) / complex<double>(0,3);
```

在往 C++ 里加入了虚数类后，它又是这个样子：

```
c = (6 + imaginary<double>(2) - 1 + imaginary<double>(3)) /
imaginary<double>(3);
```

换句话说，虚数 *nn* 可以只使用 *nn*i 来表示，而不需要它成构造函数调用形式 complex<long double>(0,*nn*)。

26.2 效率

在 C++ 里由于没有虚数类型，意味着关于虚数的操作符会在 0 实数部分引起大量附加的计算。例如，在 D 里将两个虚数相加实际就只是一个加法操作：

```
ireal a, b, c;
c = a + b;
```

在 C++ 里，这个有两个相加操作，因为实数部分也需要相加：

```
c.re = a.re + b.re;
c.im = a.im + b.im;
```

乘法就更糟，因为需要完成 4 个乘法、2 个加法，而不仅仅是一个乘法：

```
c.re = a.re * b.re - a.im * b.im;
c.im = a.im * b.re + a.re * b.im;
```

除法最糟——D 只有一个除法操作，而 C++ 的复数除法实现却一般包含了一个比较操作、3 个除法、3 个乘法 和 3 个加法：

```
if (fabs(b.re) < fabs(b.im))
{
    r = b.re / b.im;
    den = b.im + r * b.re;
    c.re = (a.re * r + a.im) / den;
    c.im = (a.im * r - a.re) / den;
}
else
{
    r = b.im / b.re;
    den = b.re + r * b.im;
    c.re = (a.re + r * a.im) / den;
    c.im = (a.im - r * a.re) / den;
}
```

在 C++ 里为了避免这些效率问题，我们可以使用一个双精度数来模拟虚数。例如，在 D 里有：

```
cdouble c;
idouble im;
c *= im;
```

则在 C++ 里，可以被写成：

```
complex<double> c;
double im;
c = complex<double>(-c.imag() * im, c.real() * im);
```

不过，把复数做为带有算术操作符的库类型的长处就不复存在。

26.3 语义

更为糟糕的是，没有虚数类型会引起不易察觉的错误答案。引用一下 Kahan 教授的描述：

> “在实现复数函数 SQRT 和 LOG （这些在 Fortran 和 现在发行的 C/C++ 编译器库里是必须的）时，使用的方式由于忽视了在 IEEE 754 算术运算里的 0.0 符号，从而只要复数变量 Z 为负的实数值，都会跟象 SQRT(CONJ(Z)) = CONJ(SQRT(Z)) 和 LOG(CONJ(Z)) = CONJ(LOG(Z)) 那样的标识符冲突，这样便发生一连串的问题。如果复数算术运算还使用 pairs (x, y) 来操作，而不是对有着实数和虚数变量的 x + i*y 进行直观求和，这种怪异的事就不可避免。pairs 对于复数算术运算是*不适合的*；复数算术运算需要的是虚数类型。”

语义问题：

- 对于公式 (1 - infinity*i) * i，它应该产生 (infinity + i) 这样的结构。可是，如果第二个因数是 (0 + i) 而不仅仅是那上 i，那么结果将是 (infinity + NaN*i)，有一个假的 NaN 生成了。
- 明确的虚数类型预留了 0 标记，这个对于包含有分支切割(branch cut)的计算是很有必要的。

C99 标准的附录 G 有关于处理此问题的建议。可是，这些建议并不是 C++98 标准的一部分，因此不能广泛的依赖于它。

26.4 参考

How Java's Floating-Point Hurts Everyone Everywhere W. Kahan 和 Joseph D. Darcy 教授

The Numerical Analyst as Computer Science Curmudgeon 作者 W. Kahan 教授

"Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit" 作者 W. Kahan，
《The State of the Art in Numerical Analysis》 第 7 章，(1987) 由 M. Powell and A. Iserles for Oxford U.P. 出版

第 27 章 契约编程(D vs C++)

许多人给我写信说，D 的契约编程(DbC)添加的东西 C++ 也支持。并且他们继续使用在 C++ 里进行 DbC 的技术证明他们的观点。

回顾一下：DbC 是什么，在 D 里它是怎样完成的；并总结出各种不同的 C++ DbC 技术都能够干什么，都是很有意义的。

Digital Mars C++ 通过添加对 C++ 的扩展来实现对 DbC 的支持，不过在这里并不想谈及它们，因为它们不是标准 C++ 的一部分，同时也不被其它的 C++ 编译器所支持。

27.1 在 D 里的契约编程

在 D 的契约编程文档里有着更为完整的说明。小结一下，D 里的 DbC 有下列特性：

1. *assert（断言）* 是基本的契约。
2. 当断言契约失败时，它会抛出一个异常。这个异常可以被捕捉和处理，也允许它终止程序。
3. 类可以有“*类 invariant*”，它会在每一个公共类成员函数的入口和出口，每一个构造函数的出口，以及析构函数的入口这些地方被检查。
4. 对象引用的断言契约会检查该对象的“类 invariant”。
5. 类 invariant 是继承的，即表示一个派生类 invariant 会隐式调用基类 invariant。
6. 函数可以有 *preconditions* 和 *postconditions*。
7. 对于类继承层次里的成员函数，派生类函数的 precondition 是跟它所重写的(override)所有函数的 precondition“或(OR)”运算到一起的。而 postcondition 是“与(AND)”运算到一起的。
8. 通过编译器开关，DbC 代码可以被启用，也可以被撤离编译后代码。
9. 有没有启用 DbC 检查，代码在语义上都是一样的。

27.2 在 C++ 里的契约编程

27.2.1 `assert（□ 言）` 宏

C++ 有基本的 `assert` 宏，它可以测试它的实参，并且如果它失败，就中断程序。`assert` 可以使用 `NDEBUG` 宏来打开和关闭。

`assert` 并不知道“类常量”，而且在它失败时也不会抛出异常。它只是在输出一个信息之后就中断程序。`assert` 依赖于宏文本处理器来工作。

`assert` 位于在标准 C++ 里显式支持 DbC 开始和结束的地方。

27.2.2 类不变量(Class Invariants)

假设在 D 里有一个“类 invariant”

```
class A
{
    invariant() { ...contracts... }
```

```
    this() { ...}        // 构造函数
    ~this() { ...}       // 析构函数

    void foo() { ...}    // 公共成员函数
}

class B : A
{
    invariant() { ...contracts... }
    ...
}
```

在 C++ 里去完成同样的效果（感谢 Bob Bell 提供了这个）：

```
template
inline void check_invariant(T& iX)
{
#ifdef DBC
    iX.invariant();
#endif
}

// A.h:

class A {
    public:
#ifdef DBG
      virtual void invariant() { ...contracts... }
#endif
      void foo();
};

// A.cpp:

void A::foo()
{
    check_invariant(*this);
    ...
    check_invariant(*this);
}

// B.h:

#include "A.h"

class B : public A {
    public:
#ifdef DBG
       virtual void invariant()
       {   ...contracts...
          A::invariant();
       }
#endif
      void bar();
```

```
};

// B.cpp:

void B::barG()
{
    check_invariant(*this);
    ...
    check_invariant(*this);
}
```

对于 A::foo() 有着附加的复杂性。在该函数的每一个正常出口，invariant() 都应该被调用。于是像下面那样的代码：

```
int A::foo()
{
    ...
    if (...)
        return bar();
    return 3;
}
```

就需要改写为：

```
int A::foo()
{
    int result;
    check_invariant(*this);
    ...
    if (...)
    {
        result = bar();
        check_invariant(*this);
        return result;
    }
    check_invariant(*this);
    return 3;
}
```

或者重新编写该函数，以便它只有一个出口点。一种省事的方法就是使用 RAII 技术：

```
int A::foo()
{
#if DBC
    struct Sentry {
      Sentry(A& iA) : mA(iA) { check_invariants(iA); }
      ~Sentry() { check_invariants(mA); }
      A& mA;
    } sentry(*this);
#endif
    ...
    if (...)
        return bar();
    return 3;
}
```

#if DBC 仍然在那里，因为如果 check_invariants 没有编译出什么内容来，有些编译器就不可能优化好所有的事情。

27.3 Preconditions 和 Postconditions

假设在 D 里有下列代码：

```
void foo()
   in { ...preconditions... }
   out { ...postconditions... }
   body
   {
       ...实现...
   }
```

在 C++ 里，这个可以使用嵌套的 Sentry 结构来处理：

```
void foo()
{
   struct Sentry
   {   Sentry() { ...preconditions... }
      ~Sentry() { ...postconditions... }
   } sentry;
...实现...
}
```

如果 preconditions 和 postconditions 仅由 `assert` 宏构成，则所有的都不需要放置在 `#ifdef` 对里，因为如果 `assert` 被关掉的话，一个好的 C++ 编译器会优化好所有的事情。

不过，假设 `foo()` 是排序一个数组，这样 postcondition 需要贯穿该数组，并检验它是否真的被排序了。现在，所有的东西都需要放置在 `#ifdef` 块里：

```
void foo()
{
#ifdef DBC
   struct Sentry
   {   Sentry() { ...preconditions... }
      ~Sentry() { ...postconditions... }
   } sentry;
#endif
...实现...
}
```

（我们可以利用 C++ 规则，即：模板只在被使用来避开 `#ifdef` 时才进行实例化，实例化方法就是把该条件放入到被 `assert` 所引用的那个模板函数里。）

让我们给那个需要在 postcondition 里被检查的 `foo()` 函数添加一个返回值。在 D 里：

```
int foo()
```

```
   in { ...preconditions... }
   out (result) { ...postconditions... }
   body
   {
       ...实现...
       if (...)
           return bar();
       return 3;
   }
```

在 C++ 里：

```
int foo()
{
#ifdef DBC
   struct Sentry
   {   int result;
       Sentry() { ...preconditions... }
       ~Sentry() { ...postconditions... }
   } sentry;
#endif
...实现...
   if (...)
   {   int i = bar();
#ifdef DBC
       sentry.result = i;
#endif
       return i;
   }
#ifdef DBC
   sentry.result = 3;
#endif
   return 3;
}
```

现在 `foo()` 添加一对参数。在 D 里：

```
int foo(int a, int b)
   in { ...preconditions... }
   out (result) { ...postconditions... }
   body
   {
       ...实现...
       if (...)
           return bar();
       return 3;
   }
```

在 C++ 里：

```
int foo(int a, int b)
{
#ifdef DBC
   struct Sentry
```

```
    {   int a, b;
        int result;
        Sentry(int a, int b)
        {   this->a = a;
            this->b = b;
            ...preconditions...
        }
        ~Sentry() { ...postconditions... }
    } sentry(a,b);
#endif
...实现...
        if (...)
        {   int i = bar();
#ifdef DBC
            sentry.result = i;
#endif
            return i;
        }
#ifdef DBC
        sentry.result = 3;
#endif
        return 3;
}
```

27.4 成员函数的 Preconditions 和 Postconditions

假设在 D 里使用多态函数的 preconditions 和 postconditions：

```
class A
{
    void foo()
        in { ...Apreconditions... }
        out { ...Apostconditions... }
        body
        {
            ...实现...
        }
}

class B : A
{
    void foo()
        in { ...Bpreconditions... }
        out { ...Bpostconditions... }
        body
        {
            ...实现...
        }
}
```

调用 B.foo() 的语义是：

- Apreconditions 或 Bpreconditions 必须被满足。
- Apostconditions 和 Bpostconditions 必须被满足。

在 C++ 里来实现它：

```
class A
{
protected:
    #if DBC
    int foo_preconditions() { ...Apreconditions... }
    void foo_postconditions() { ...Apostconditions... }
    #else
    int foo_preconditions() { return 1; }
    void foo_postconditions() { }
    #endif

    void foo_internal()
    {
        ...实现...
    }

public:
    virtual void foo()
    {
        foo_preconditions();
        foo_internal();
        foo_postconditions();
    }
};

class B : A
{
protected:
    #if DBC
    int foo_preconditions() { ...Bpreconditions... }
    void foo_postconditions() { ...Bpostconditions... }
    #else
    int foo_preconditions() { return 1; }
    void foo_postconditions() { }
    #endif

    void foo_internal()
    {
        ...实现...
    }

public:
    virtual void foo()
    {
        assert(foo_preconditions() || A::foo_preconditions());
        foo_internal();
        A::foo_postconditions();
        foo_postconditions();
```

```
    }
};
```

这里发生了一些有趣的事。preconditions 不再使用 `assert` 来完成，因为结果需要被“或(OR)”运算到一起。给大家一个练习，往“类 invariant”里添加一些内容，给 `foo()` 添加函数返回值，并给添加参数到 `foo()`。

27.5 总结

这些 C++ 技术的确有效。不过，除了 `assert` 以外，它们都未被标准化，因此项目之间也就变更各不一样。此外，它们需要大量单调的对某种特定约定的支持，从而给代码造成很大的混乱。也许这就是什么在实际当中很少被看到的原因。

通过在语言里添加对 DbC 的支持，D 提供了一个轻松的方式来使用 DbC，并且保证它正确。由于使用了语言标准化的方式，它就可以随意在项目之间使用。

27.6 参考

在《面向对象的软件构造》的第 C.11 章 介绍了契约编程的理论和原理
作者 Bertrand Meyer, Prentice Hall

在《C++ 语言编程特别版》的 24.3.7.1 到 24.3.7.3 章 讨论了在 C++ 里的契约编程
作者 Bjarne Stroustrup, Addison-Wesley

第 28 章 Lisp vs. Java... D?

作者 Lionello Lunesu, Oct. 18th 2006

两周前，Josh Stern 贴了一个 有趣的链接 到 digitalmars.D newsgroup。该链接指向的是一个描述编程挑战的页面，该页面最初用于这样一项研究——38 个 C、 C++ 和 Java 程序员被要求去编写同一个程序的不同版本，用于比较 Java 和 C++ 的效率。

许多其它的程序员也参与了这个挑战。在“一个这样的比较”里，当对开发时间和代码行进行比较时，Lisp 被证明比 Java 和 C++ 都要胜出一筹，将记录保持在了 2 个小时左右，同时只有 45 个非注释和非空白行。对于 C++，最短的开发时间是 3 个小时，最短的程序有 107 行代码。

如果有空闲时间，我想我应该自己去试试。虽然我自己是一个 C++ 程序员（到现在已经有 10 年了），不过我认为这是个很好的机会，可以测试一下我在 D 编程语言 方面的能力。

我并没有看过其它的实现，仅仅是阅读过“初始问题描述”。从阅读该描述到最后完成程序，我花了 **1 个小时又 15 分钟**。程序长度只有 55 行（请记住，最短的 C++ 程序有 107 行），没有包括空白行、注释、断言、单元测试、契约，而且括号置于同一行（跟 Lisp 一样）。

该程序基本上必须完成这些：从字典文件里读取所有的单词；对于每一个字母都有一个对应的数字（就像电话上的字母）；使用这个映射，从另一个方本文件里读取电话号码，并且打印对应该电话号码的所有可能单词的合并。更为详细的信息请看上面的链接。

是什么使得在 D 完成这个任务变更这么简单呢？刚好碰巧在我手边有我所需要每一个构造。使用过的优点有：

- 断言、契约、单元测试
- 有着简单分割功能的内建数组和字符串
- 内建的关键数组
- 垃圾回收
- 类型推演
- 嵌套函数
- 从函数返回一个数组（通过引用）

在实际编写过程中，我并没有停下来想任何事情。一旦我相到了该字典的实际容器，程序就自动往前发展了，这表明“获得的”Lisp 版本是以类似的方式存储字典的。

28.1 编译

想要自己测试这个程序，首先需要下载文件 dictionary.txt 和 input.txt。然后，下载最新的 D 编译器。要编译的话，就复制粘贴这段 D 代码到新建的文件 phoneno.d，然后使用下面的命令编译它：

```
dmd -run phoneno.d

// 由 Lionello Lunesu 创建，可用于公众域。
// 此文件修改自它原来的版本。
// 它被格式化过，以适合你的屏幕。
module phoneno;      // 可选的
```

```
import std.stdio;   // writefln
import std.ctype;   // isdigit
import std.stream;  // 缓冲文件

// 只是为了增强可读性(想像一下这个：char[][][char[]])
alias char[] string;
alias string[] stringarray;

/// 从字符串里去掉非数字字符 (COW)
string stripNonDigit( in string line )
{
    string ret;
    foreach(uint i, c; line) {
// 错误：位于 C:\dmd\src\phobos\std\ctype.d(37) 的 std.ctype.isdigit
        // 跟位于 C:\dmd\src\phobos\std\stream.d(2924) 的 std.stream.isdigit 冲突
        if (!std.ctype.isdigit(c)) {
            if (!ret)
                ret = line[0..i];
        }
        else if (ret)
            ret ~= c;
    }
    return ret?ret:line;
}

unittest {
    assert( stripNonDigit("asdf") == ""  );
    assert( stripNonDigit("\'13-=2 4kop") ==  "1324"  );
}

/// 把一个单词转换成一个数字，忽略掉所有的非数字字母字符
string wordToNum( in string word )
{
// 用于当前任务的翻译表
const char[256] TRANSLATE =
    "                                "  // 0
    "                0123456789      "  // 32
    " 5763049961785188123476223 9    "  // 64
    " 5763049961785188123476223 9    "
    "                                "
    "                                "
    "                                "
    "                                ";
    string ret;
    foreach(c; cast(ubyte[])word)
        if (TRANSLATE[c] != ' ')
            ret ~= TRANSLATE[c];
    return ret;
}

unittest {
```

```
// 使用来自任务描述的任务测试 wordToNum。
 assert( "01112223334455666777888999" ==
   wordToNum("E | J N Q | R W X | D S Y | F T | A M | C I V | B K U | L O P | G
H Z"));
 assert( "01112223334455666777888999" ==
   wordToNum("e | j n q | r w x | d s y | f t | a m | c i v | b k u | l o p | g
h z"));
 assert( "0123456789" ==
   wordToNum("0 |   1   |   2   |   3   |  4  |  5  |   6   |   7   |   8   |
9"));
}

void main( string[] args )
{
// 此关联数组把数字映射到一个单词数组。
    stringarray[string]    num2words;

    foreach(string word; new BufferedFile("dictionary.txt" ) )
num2words[ wordToNum(word) ] ~= word.dup;         // 必须是 dup

/// 为给定数字查找所有的 alternatives
    /// (应该剔除了所有的非数字字符)
    stringarray _FindWords( string numbers, bool digitok )
    in {
        assert(numbers.length >  0);
    }
    out(result) {
        foreach (a; result)
            assert( wordToNum(a) == numbers );
    }
    body {
        stringarray ret;
        bool foundword = false;
        for (uint t=1; t<=numbers.length; ++t) {
            auto alternatives = numbers[0..t] in num2words;
            if (!alternatives)
                continue;
            foundword = true;
            if (numbers.length >  t) {
// 跟其余所有的 alternatives 一起合并成当前所有的 alternatives
                // (下一片可以使用数字开头)
                foreach (a2; _FindWords( numbers[t..$], true     ) )
                    foreach(a1; *alternatives)
                      ret ~= a1 ~ " " ~ a2;
            }
            else
ret ~= *alternatives;    // 添加这些 alternatives
        }
// 试着保留一个数字，仅仅在如果我们被允许以及没有被找到其它的
        // alternatives 的情况下才那样做
        // 测试 "ret.length" 比测试 "foundword" 更有意义
        // 不过其它的实现似乎恰就是那样做的。
        if (digitok && !foundword) { //ret.length == 0
```

```
            if(numbers.length >  1) {
// 使用来自余下的所有 altenatives 合并成一个数字
                // (下一个不能使用一个数字做开头)
                foreach (a; _FindWords( numbers[1..$], false ) )
                    ret ~= numbers[0..1] ~ " " ~ a;
            }
            else
                ret ~= numbers[0..1];    // just append this digit
        }
        return ret;
    }

/// (这个函数被内嵌在原始程序里)
    /// 为给定的电话号码查找所有的 alternatives
    /// 返回：字符串数组
    stringarray FindWords( string phone_number )
    {
        if (!phone_number.length)
            return null;
// 从电话号码里剔除所有的非数字字符，
        // 并且把它传递给该递归函数（允许数字打头）
        return _FindWords( stripNonDigit(phone_number), true );
    }

// 读取电话号码
    foreach(string phone; new BufferedFile("input.txt"   ) )
        foreach(alternative; FindWords( phone ) )
            writefln(phone, ": ", alternative );
}
```

第 29 章 问与答

相同的问题总是突然出现，因此明显要做的事就是准备一个“问与答(FAQ)”。

29.1.1 为什么名叫 D?

最开始的名字其实是 Mars 编程语言。不过我的朋友们一直都把它叫做 D，后来连我自己都开始把它称作 D 了。把 D 做为 C 的后继的想法至少可以追溯到 1998 年前后，请看这里。

29.1.2 我在什么地访才能获得 D 编译器？

就在这里。

29.1.3 D 有 linux 的移植版本吗？

有的，D 编译器包含有 linux 版本。

29.1.4 D 有 GNU 版本吗？

David Friedman 已经把 D 的前台跟 GCC 集成到一块了。

29.1.5 怎么样我才能为 CPU X 编写自己的 D 编译器？

Burton Radons 已经编写了一个后台。你可以将它作为指导。

29.1.6 我能在什么地方获得适合 D 的 GUI 库？

因为 D 可以调用 C 函数，因此所有带 C 接口的 GUI 库都可以通过 D 来访问。大量 D 的 GUI 库及移植可以在“可用的 GUI 库”找到。

29.1.7 我能在什么地方获得适合 D 的 IDE 库？

试试 Elephant、Poseidon 或者 LEDS。

29.1.8 关于模板怎么样了？

D 现在支持高级模板。

29.1.9 为什么强调实现轻松？

难道让语言的用户轻松使用不重要吗？不对，它真的很重要。不过一门“物件”语言对所有人来说都是没用的。语言实现得越简单，就会有越多丰富(robust)的实现。在 C 的鼎胜时期，有超过 30 多种不同的商业 C 编译器用于 IBM PC。但并没有多少转换到 C++。再看看当今市场上的 C++ 编译器，每一个都发展了多少年？至少 10 年？程序员等了多年为的就是

要 C++ 的各个部分在被描述后能够尽快得以实现。如果 C++ 不是那么流行的话，无疑像多继承、模板等等所有这些复杂的功能可能早就被实现了。

我觉得如果一门实现得更简单，那么它也就更容易让人理解。难道这样不是更好——花时间学习编写更好的程序，而不是去学习什么语言诀窍？如果一门语言能够拥有 C++ 语言 90% 的能力，而又仅有它 10%的复杂度，我认为这种“生意”还是值得做的。

29.1.10 为什么 D 中还有 printf?

printf 并不是 D 的一部分，它是 C 的标准运行库（可以被 D 访问）的一部分。D 的标准运行库有 **std.stdio.writefln**，它跟 **printf** 一样强大，不过更容易使用。

29.1.11 D 会开源吗？

D 的前台是开源的，而且源码跟随编译器一起。运行库不是全部开源。有的，David Friedman 已经把 D 的前台跟 GCC 一起集成，创建了 **gdc**——一个 D 的完全开源实现版本。

29.1.12 为什么强调 switch 语句？

许多人要求在 switch 语句里两个 case 之间加上一个 break，而 C 默认使用的就是这种情况，却常常导致很多的程序错漏。

D 没有那样做的原因等同于“整型提升(integral promotion)”规则要求跟操作符优先规则保持一样的原因——为的是让看上去跟 C 中一样的代码具有同样的操作效果。如果语义上有一点轻微的不同，都会产生令人难受的细微错漏(frustratingly subtle bugs)。

29.1.13 为什么我该使用 D 来代替 Java？

D 的目标跟 Java 是完全不一样的——不管理念，还是实际。 See this comparison.

Java 设置的目的是写一次，可以运行在任何地方。D 的目标是编写高效的本地系统应用程序。虽然 D 和 Java 都有着相同的看法——“垃圾回收”很好，而多继承却很糟，而不同的设计目标意味着语言有着给人不同的感觉。

29.1.14 难道 C++ 使用 STL 也能不支持字符串等等吗？

在 C++ 标准库里存在着一些机制用于处理：字符串、动态数组、关联数组、边界检查数组以及复数。

当然，所有这些都可以使用库，同时带上某种编码规则等等来完成。不过你也可以在 C 里使用面向对象编程（我见过有完成的)。通过最简单的 BASIC 翻译器就可以支持的象字符串那样的东西，却要求使用一个庞大而又复杂的基础结构来支持，这真的就合适宜吗？在

STL 里，仅字符串类型的实现，就有超过两千多行的代码，使用了模板的每一个高级功能。你会有多少自信它们都能正确地工作？如果没有正确工作，你要怎样去修复它呢？在你使用时有了错误，面对那些令人难以理解的错误信息，你该怎么办？你要怎么样才能确保正确使用它呢（如没有内存泄漏等等）？

D 的字符串实现简单直接。关于如何使用它，并没会么什么可怀疑的；也不用当心内存泄漏；错误信息也很明确；并且关于它是否跟预期的那样工作，检查起来也不费劲。

29.1.15 难道不能在 C++ 里使用第三方库来完成“垃圾回收”吗？

不是那样的，我自己都有使用一个。它并不是语言的一部分，而且需要一些搅乱语言才能让它工作。在 C++ 里使用 gc 并不符合标准或者对于 C++ 程序员也是极少数情况。象 D 语言一样将其构建到语言里，让它对于每天的编程事务更有实际意义。

GC 的实现并不难，除非你想构建一个更高级的版本。不过更高级的版本就跟构建一个更为优化的版本一样——即使使用简单基本的版本，语言仍然 100%的正确工作。通过多种实现在生成代码的质量上进行比较，比强调特定功能的哪些方面是否根本实现，能更好的服务于编程社区。

29.1.16 难道不能在 C++ 里使用第三方库来完成“单元测试”吗？

当然可以。找一个来试试，然后比较一下在 D 是怎么做的。非常明显，内建到语言里是多么大的提高。

29.1.17 为什么在可移植的语言里会有汇编语句？

汇编语句允许直接将汇编代码插入到 D 函数里。汇编代码显然不具有可移植性。不过，D 还是想能成为开发系统程序的有用语言。系统程序几乎总是对它里面的系统依赖代码很敏感，内嵌汇编就有些不一样了。内嵌汇编对于很多东西总是很有用，像访问特定的 CPU 指令、访问标识位、特定的计算状态以某块代码的超级优化。

以 C 编译器有内嵌汇编器以前，我使用外部的汇编器。同样的痛苦在于：又有许多许多版本的汇编器；服务商继续更改汇编的语法；在不同版本里有着许多不同的错漏；甚至命令行语法也不断被更改。这些意味着用户并不能可靠地重新建立任何汇编器所需要的代码。内嵌的汇编器可以提供可靠性或一致性。

29.1.18 80 位实型数的关键是什么？

更高的精度可以使得完全更为准确的浮点计算，尤其是在将大量小实数相加的时候。Kahan 教授（是他设计了 Intel 的浮点单元）有一篇关于这个题目的论文。

29.1.19 在 D 里我怎么样才能完成匿名结构/联合？

```
import std.stdio;

struct Foo
{
    union { int a; int b; }
    struct { int c; int d; }
}

void main()
{
    writefln(
      "Foo.sizeof = %d, a.offset = %d, b.offset = %d, c.offset = %d, d.offset =
%d",
      Foo.sizeof,
      Foo.a.offsetof,
      Foo.b.offsetof,
      Foo.c.offsetof,
      Foo.d.offsetof);
}
```

29.1.20 怎么样我才能让 printf() 处理字符串?

在 C 里， 打印(printf)一个字符串通常的方法是使用 **%s** 格式：

```
char s[8];
strcpy(s, "foo");
printf("string = '%s'\n", s);
```

在 D 里试着这样做，像这样：

```
char[] s;
s = "foo";
printf("string = '%s'\n", s);
```

通常会打印一些不相关的信息，或者出现访问冲突。原因就是在 C 里，字符串是使用字符 0 来中止的。**%s** 格式会执行打印，直到遇到'0'。在 D 里，字符串不是由 0 来中止的，其大小是由一个独立的长度值来决定。因此，打印字符串应该使用 **%.*s** 格式：

```
char[] s;
s = "foo";
printf("string = '%.*s'\n", s);
```

这正是所期望的。请记住，printf 的 **%.*s** 会执行打印，直到满足其长度或遇到 0，因此在内部带有 0 的字符串只会打印到第一个 0 那个地方。

当然，更为简单的解决办法就是使用 **std.stdio.writefln**，它可以很好的处理 D 字符串。

29.1.21 为什么将浮点值初始化成 NaN，而不是 0？

浮点值如果没有显式进行初始化，那么它就会被初始化成 NaN（不是一个数）：

```
double d;       // d 被设置为 double.nan
```

NaN 有一个很有趣的特性，无论什么时候在计算过程中，NaN 被用作了操作数，那么结果都将是 NaN。因此，NaN 会进行传播，无论什么时候计算过程使用了它，那么它都会出现在输入结果里。这暗示着如果在输出结果里的出现了 NaN，那么毫无疑问地表明使用了一个未被初始化的值。

如果使用 0.0 做为浮点值的默认初始化值，它的影响可能在输出结果里会被轻易地忽视；这样，如果该默认初始值是无意识给的，那么这个错漏就可能不会被发现。

默认初始化值并不意味着是有用的值，它意味着会产生错漏。Nan 就是很好的例子。

不过，编译器真的能够为检测那些被使用但又未被初始化的变量并产生错误信息吗？大部分时候是可以的，不过并不是一直都行，是否有效依赖于编译器内部数据流分析情况。因此，依赖于这些是不可移植和不可靠的。

由于 CPU 设计的方式限定，对于整数并没有 NaN 值，因此 D 就使用 0 来代替。它没有象 NaN 所有的那样的错误检测好处，不过至少由于未知的默认初始化导致的错误是一致的，而且这样更利于调试。

29.1.22 为什么不支持重载分配操作符？

Overloading of the assignment operator for structs is supported in D 2.0.

29.1.23 在我的键盘里没有符号'~'？

在 PC 键盘上，按下[Alt]键以及数字键盘上的 1、2、6（依着顺序）。这样就会产生'~'字符了。

29.1.24 我可以连接使用其它编译器创建的 C 目标文件吗？

DMD 产生 OMF (Microsoft Object Module Format——微软目标模块格式) 目标文件，而其它的编译器，像 VC++，产生的是 COFF 目标文件。这样设计 DMD 的输出是想迎合 DMC —— Digital Mars C 编译器，它产生的也是 OMF 格式。

DMD 使用的 OMF 格式是微软基于早期的 Intel 所设计的一种格式所设计出来的。在某个关键时候，微软由于有了自己定义的关于 COFF 的不同版本，而将其抛弃了。

使用相同的目标格式并不表示任何以该种格式存在的 C 库都可以成功连接和运行。还需要很多的兼容要考虑——例如调用习惯、名字毁损(name mangling)、编译器帮助函数以及隐藏的关于工作方式的假定。如果让 DMD 产生微软的 COFF 输出文件，那么仍然没有多少机会让它们能够成功地跟使用 VC 设计和测试的目标文件一起工作。在微软的编译器生成 OMF 时，后面存在很多的问题。

不同的文件格式使得区分那些无需测试是否可以跟 DMD 一起工作的库文件件很有用。如果它们不是，就可以产生很多怪异的问题，即使可能成功成功将它们连接到一下。这个实际上需要专业人士才能实现让由某个服务商的编译器创建的二进制文件可以跟其它的服务商的编译器输出一起工作。

即是说，linux 版本的 DMD 产生的目标文件是 ELF 格式——这个是 linux 里的标准，并且它专门设计来跟标准的 linux C 编译器(gcc)一起工作。

有一个成功使用已有 C 库的例子——这些库是以 DLL 形式提供，并且使用一般的 C ABI 接口来封装。这里的可连接部分叫做“导入库”，并且微软的 COFF 格式导入库可以成功地转换成 DMD 的 OMF，使用的是 coff2omf 工具。

29.1.25 为什么不支持带有 /foo/g 语法的正则表达式？

主要有两个原因：

1. /foo/g 语法会使得很难将词法器(lexer)跟解析器(parser)分开，因为'/'是分割特征符。
2. 已以有 3 种字符串类型；增加规则表达式就会再添加 3 个。这样会增加更多的东西到编译器、调试信息以及库里，这样做不值得。

29.1.26 为什么使用 C++ 来写 D 的前台，而不是使用 D?

前台使用 C++ 为的是可以跟已有的 gcc 和 dmd 后台都有接口。即表示可以轻松地跟其它已有的后台（有可能是使用 C++ 来编写的）建立接口。DMDScript 的 D 实现，其性能要好于 C++ 版本 的实现；这就表明使用 100% 的 D 编写专业品质的编译器并没有什么问题。

29.1.27 为什么不是所有 Digital Mars 的程序都被转换到 D？

将一个复杂的、调试好的、可工作的应用程序从一种语言转换成另外一种语言并没有多少好处。不过新的 Digital Mars 应用程序会使用 D 来实现。

29.1.28 什么时候我该使用 foreach 循环而不使用 for 循环？

难道仅仅是为了性能或可靠性吗？

通过使用 foreach，你就可以让编译器来决定优化，而不需要自己费力去弄它了。例如，是指针还是索引更好呢？我该不该缓存终止条件呢？我该不该倒着循环呢？对于这些问题的回答都是不容易的，而且对于不同机子也可能不一样。如寄存器分配，就让编译器来进行优化吧。

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

或者：

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

或者：

```
for (T* p = &foo[0]; p < &foo[length]; p++)
```

或者：

```
T* pend = &foo[length];
for (T* p = &foo[0]; p < pend; ++p)
```

或者：

```
T* pend = &foo[length];
T* p = &foo[0];
if (p < pend)
{
       do
       {
       ...
       } while (++p < pend);
}
```

并且，是使用 size_t 还是 int 呢？

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

让编译器来选吧！

```
foreach (v; foo)
       ...
```

注意，我们甚至不需要知道类型 T 需要些什么，这样可以避免在 T 改变时出现错漏。我甚至不需要知道 foo 是否就是个数组，或关联数组，或结构，或类集（collection class）。这也可以避免常见的“越界错漏（fencepost bug）”。

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

而且它也可以避免在 foo 函数被调用时创建临时变量。

使用 for 循环唯一的原因就是如果你的循环不太适合这种方便的形式，例如，你想在循环过程是更改终止条件。

29.1.29 为什么 D 没有像到 C 那样到 C++ 的接口？

要 D 跟 C++ 之间有个接口，其复杂程序跟编写一个 C++ 编译器差不多，而这正好违背了让 D 成为更为容易实现的语言的初衷。对于那些有着 C++ 代码（前提是这些代码都可以工作）的人来说，他们已被 C++ 缠上了（他们也不会转移到其它任何语言）。

要 D 代码去调用那些随意的 C++ 代码（假设它们是不可修改的），还需要解决大量的问题。下面的列表尽管不是全部，不过也显示出了一些难于解决的问题。

1. D 源码是 unicode 格式，而 C++ 的是带有代码页的 ASCII 格式。或者不是那样。这个不太明确。这会影响字符串的内容。
2. std::string 无法处理多字节的 UTF。
3. C++ 有标识名空间。而 D 没有。可能会发生更名的情况。
4. C++ 代码经常依赖于特定编译器的扩展。
5. C++ 有名字空间。而 D 有的是模块。两者之间并没有明显的映射关系。
6. C++ 将源码当成一个巨大的文件（在预处理以后）。而 D 却将源码当作具有层次关系的模块和包。
7. 枚举名字确定范围规则的操作不一样。
8. C++ 代码，尽管几十年来都试图使用内建一个来代替宏功能，实际情况却是比以往更加依赖于层层相叠的宏。在 D 中没有对应的特征符过期(token pasting) 或 字符串化(stringizing)功能。
9. 宏名有超越 #include 文件的全局域功能，而对于巨型源码文件却是局部的。
10. C++ 有着随意的多继承和虚拟基类。而 D 没有。
11. C++ 没有区分 in、 out 和 ref（例如：inout）参数。
12. C++ 名字毁损（name mangling）因编译器不同而不一样。
13. C++ 会产生这样的异常：可以是任意类型，但就不是 Object 的后代。
14. C++ 基于 const 和 volatile 重载。而 D 没有。
15. C++ 重载操作符的方式有着明显的不同——例如，操作符[]()用于左值和右值的重载就是基于 const 重载和代理类。
16. C++ 重载像"<"（完全跟">"无关）那样的操作符。
17. C++ 重载间接符（操作符*）
18. C++ 没有区分类和结构对象。
19. vtbl[] 的定位和排列在 C++ 和 D 之间是不同的。
20. RTTI 完成的方式完全不一样。C++ 没有 classinfo.
21. D 不允许分配重载。
22. D 的结构对象没有构造函数和析构函数。
23. D 不是两阶段探查，同时也不是 Koenig (ADL)探查。
24. C++ 使用“友元”系统来关联类，而 D 使用包和模块。
25. C++ 类的设计倾向于解决显式的内存分配问题，而 D 不是。
26. D 的模板系统很不一样。
27. C++ 有“异常规范”。
28. C++ 有全局性的操作符重载。
29. C++ 的名字毁损(name mangling)依赖于是类型修饰符的 const 和 volatile。因此，D 没有 const 和 volatile 这样的类型修饰符，也没有直接的方式可以从 D 类型推断 C++ 里“毁损的标识符(mangled identifier)”。

重点就是语言功能影响代码设计。C++ 的设计并不适合 D。即使你可以找到某种方式来自动适应两者，最终结果可能是个“诱人的”组合，就像右边是 honda（本田汽车）,而左边是 camaro（卡迈隆）那一样。

29.1.30 为什么 D 不使用引用计算(reference counting)来处理“垃圾回收”？

引用计算有它的长处，不过也有其严重的不足：

- 循环的数据结构不能被解脱。

- 每一个指针复本都会被要求增加或减少——包括在简单地传递一个引用到函数。
- 在多线程应用程序里，增加和减少必须同步。
- 异常处理器（finally 块）必须被插入进来处理所有的减少情况，以便不会有泄漏。而对比于断言(assertion)，就不会有像“zero overhead 异常”这类事情出现。
- 除了支持对于非对象数据的任意分配的引用计算外，还为了支持分割(slicing)和内部指针，那么就必须为每一个要被引用计算的分配单元，分配一个单独的“wrapper”对象。这样实际上就使得需要分配的数目翻倍。
- 该 wrapper 对象表示所有的指针都需要“双重引用”到要存取的数据。
- 修正编译器，使程序员远离这些痛苦，将会使得完全跟 C 接口变得更为困难。
- 引用计算可以将堆分段，从而会像 gc 那样消耗掉更多的内存（gc 很显然地会消耗掉更多的内存）。
- 引用计算不会估量潜在的问题，它只是减少它们。

被提交的 C++ shared_ptr<> 实现了引用计算，它也品尝着所有这些错误。在 shared_ptr<> vs mark/sweep 过程中，我也没有看到可以得高分的，不过如果 shared_ptr<> 最后被证明在关于性能和内存占用两方面都彻底失败的话， 我也丝毫不会感到意外。

即是说，D 可能在未来有选择地支持某种形式的引用计算，因为它在管理如文件处理那样的分散资源时表现得更好些。

29.1.31 “垃圾回收”功能很慢并且是非确定的，难道不是那样的吗？

的确，不过**所有的**动态内存管理都很慢而且是非确定的(non-deterministic)，包括 malloc/free。如果你跟那些实现做实时软件的人交谈，你就会知道他们正好是不使用 malloc/free 的，因为它们都不确定。他们是先预分配所有的数据。不过，使用 GC 来代替 malloc 可以启用高级语言构造（特别像更为强大的数组语法），这样可以大大地减少那些都需要去做的内存分配的数目。这也就意味着 GC 实际上比显示管理来得更快。

第 30 章 有名字符实体

这些都是被 D 语言所支持的字符实体(entity)名字。

注意： 并不是所有在 **符号** 列中的字符都可以在所有的浏览器正确显示。

有名字符实体

名字	值	符号
quot	34	"
amp	38	&
lt	60	<
gt	62	>
OElig	338	Œ
oelig	339	œ
Scaron	352	Š
scaron	353	š
Yuml	376	Ÿ
circ	710	ˆ
tilde	732	˜
ensp	8194	
emsp	8195	
thinsp	8201	
zwnj	8204	
zwj	8205	
lrm	8206	
rlm	8207	
ndash	8211	–
mdash	8212	—
lsquo	8216	‘
rsquo	8217	’
sbquo	8218	,
ldquo	8220	“
rdquo	8221	”
bdquo	8222	„
dagger	8224	†

Dagger	8225	‡
permil	8240	‰
lsaquo	8249	‹
rsaquo	8250	›
euro	8364	€
Latin-1 (ISO-8859-1) 实体		
nbsp	160	
iexcl	161	¡
cent	162	¢
pound	163	£
curren	164	¤
yen	165	¥
brvbar	166	¦
sect	167	§
uml	168	¨
copy	169	©
ordf	170	ª
laquo	171	«
not	172	¬
shy	173	
reg	174	®
macr	175	¯
deg	176	°
plusmn	177	±
sup2	178	²
sup3	179	³
acute	180	´
micro	181	µ
para	182	¶
middot	183	·

cedil	184	¸
sup1	185	¹
ordm	186	º
raquo	187	»
frac14	188	¼
frac12	189	½
frac34	190	¾
iquest	191	¿
Agrave	192	À
Aacute	193	Á
Acirc	194	Â
Atilde	195	Ã
Auml	196	Ä
Aring	197	Å
AElig	198	Æ
Ccedil	199	Ç
Egrave	200	È
Eacute	201	É
Ecirc	202	Ê
Euml	203	Ë
Igrave	204	Ì
Iacute	205	Í
Icirc	206	Î
Iuml	207	Ï
ETH	208	Đ
Ntilde	209	Ñ
Ograve	210	Ò
Oacute	211	Ó
Ocirc	212	Ô
Otilde	213	Õ
Ouml	214	Ö
times	215	×
Oslash	216	Ø
Ugrave	217	Ù
Uacute	218	Ú

Ucirc	219	Û
Uuml	220	Ü
Yacute	221	Ý
THORN	222	Þ
szlig	223	ß
agrave	224	à
aacute	225	á
acirc	226	â
atilde	227	ã
auml	228	ä
aring	229	å
aelig	230	æ
ccedil	231	ç
egrave	232	è
eacute	233	é
ecirc	234	ê
euml	235	ë
igrave	236	ì
iacute	237	í
icirc	238	î
iuml	239	ï
eth	240	ð
ntilde	241	ñ
ograve	242	ò
oacute	243	ó
ocirc	244	ô
otilde	245	õ
ouml	246	ö
divide	247	÷
oslash	248	ø
ugrave	249	ù
uacute	250	ú

ucirc	251	û
uuml	252	ü
yacute	253	ý
thorn	254	þ
yuml	255	ÿ
符号和希腊字母实体		
fnof	402	*f*
阿拉伯字母	913	Α
Beta	914	Β
Gamma	915	Γ
Delta	916	Δ
Epsilon	917	Ε
Zeta	918	Ζ
Eta	919	Η
Theta	920	Θ
Iota	921	Ι
Kappa	922	Κ
Lambda	923	Λ
Mu	924	Μ
Nu	925	Ν
Xi	926	Ξ
Omicron	927	Ο
Pi	928	Π
Rho	929	Ρ
Sigma	931	Σ
Tau	932	Τ
Upsilon	933	Υ
Phi	934	Φ
Chi	935	Χ
Psi	936	Ψ
Omega	937	Ω
alpha	945	α
beta	946	β
gamma	947	γ

delta	948	δ
epsilon	949	ε
zeta	950	ζ
eta	951	η
theta	952	θ
iota	953	ι
kappa	954	κ
lambda	955	λ
mu	956	μ
nu	957	ν
xi	958	ξ
omicron	959	ο
pi	960	π
rho	961	ρ
sigmaf	962	ς
sigma	963	σ
tau	964	τ
upsilon	965	υ
phi	966	φ
chi	967	χ
psi	968	ψ
omega	969	ω
thetasym	977	□
upsih	978	□
piv	982	□
bull	8226	•
hellip	8230	…
prime	8242	′
Prime	8243	″
oline	8254	‾
frasl	8260	⁄

weierp	8472	℘
image	8465	ℑ
real	8476	ℜ
trade	8482	™
alefsym	8501	ℵ
larr	8592	←
uarr	8593	↑
rarr	8594	→
darr	8595	↓
harr	8596	↔
crarr	8629	↵
lArr	8656	⇐
uArr	8657	
rArr	8658	⇒
dArr	8659	
hArr	8660	⇔
forall	8704	∀
part	8706	∂
exist	8707	∃
空	8709	∅
nabla	8711	∇
isin	8712	∈
notin	8713	∉
ni	8715	∋
prod	8719	∏
sum	8721	∑
minus	8722	−
lowast	8727	∗
radic	8730	√
prop	8733	∝
infin	8734	∞
ang	8736	∠

and	8743	∧
或	8744	∨
cap	8745	∩
cup	8746	∪
int	8747	∫
there4	8756	∴
sim	8764	∼
cong	8773	≅
asymp	8776	≈
ne	8800	≠
equiv	8801	≡
le	8804	≤
ge	8805	≥
sub	8834	⊂
sup	8835	⊃
nsub	8836	⊄
sube	8838	⊆
supe	8839	⊇
oplus	8853	⊕
otimes	8855	⊗
perp	8869	⊥
sdot	8901	⋅
lceil	8968	⌈
rceil	8969	⌉
lfloor	8970	⌊
rfloor	8971	⌋
lang	9001	〈
rang	9002	〉
loz	9674	◊

spades	9824	♠
clubs	9827	♣
hearts	9829	♥
diams	9830	◆

第 31 章 D 的常用链接

31.1 维基(Wiki)

- 语言：Wiki4D
- 运行库：Phobos
- 文档：DocWiki
- 书籍：Wikibook
- Dprogramming 的维基
- 日语：D wiki
- 百科全书(Encyclopedia)：Wikipedia
- Harmonia
- Dee 语言
- Tcler 的维基：wiki.tcl.tk
- LinuxQuestions.org

31.2 工具

- D-mode：支持 D 的 emacs 模式
- akide 是一个使用 D 编写的开源 IDE。支持 D 语法加亮、D 项目向导以及 DMD 编译器。
- Zeus for Windows 带有预定义的 D 编译器 (*) 和关键字，并且有一个 ctags 程序 (xtags.exe) 可以支持 D 语言。(*) 对于可以在 Zeus 里工作的 D 编译器，用户需要从 www.digitalmars.com/d/ 下载和安装。如果已经安装了，则它会自动启用。
- 程序编辑器：通过自定义来支持 D。
- Bud 工具用于建立(build) D 可执行文件和库文件。
- DStress：语言一致性(conformance)测试套件。
- Ragel State Machine Compiler（Ragel 状态机编译器） 可以把一定数量的状态机从正则语言编译成可执行的 D 代码。
- CanDyDoc 用于展示我们怎么样使用宏和样式表来自定义 Ddoc 结果。

31.3 库(Libraries)

- SynSoft 的 D 页面 提供了一定数量的免费 D 库。SynSoft 是 Synesis 软件 的出版商，它提供有免费的 D、Java、.NET、Perl 和 Python 库。SynSoft 已经为 D 的标准库贡献了好几个模块，而且目前正在处理标准模板库——DTL。
- Mango 用于服务器端的编程。
- D 的 MySQL 在 Linux 下的绑定。
- **D** graphical **U**ser **I**nterface (DUI——D 的图形化用户接口) 工具集。
- 简单的 URL 装载库，作者 Burton Radons。需要安装 DIG 库，虽然它不使用它，只使用 digc。带有文档。它拥有的功能：
  - urlopen:以流的方式打开一个 URL（http、file 和 ftp 格式都支持）。
  - urlread:打开一个 URL，并读取它的内容。
  - urllistdir:列举一个目录，并返回一个 URLStat 数组（支持文件和 ftp 两种格

式）。
    - urlencode, urldecode：编码和解码 URL。上面的函数都要求编码后的 URL。
- D Win32 COM 库。
- Sam McCall 的分类 字符串库 和 文档。
- DFL D 的 Form（外形）库
- GLee D (OpenGL 2.0)。

31.4 游戏

- Empire, Wargame of the Century——帝国，世纪的战争游戏
- 很老的校园追击动作游戏—— A7Xpg。
- TUMIKI Fighters 游戏。
- Torus Trooper 游戏。

31.5 媒体

- Linux Journal: The D Programming Language
- Golem: Programmiersprache D 1.0 veröffentlicht
- Heise Online: Eleganter programmieren: D ist da
- OSnews: The D Programming Language
- Slashdot: The D Programming Language, Version 1.0
- Digg:Digital Mars - D 编程语言
- Digg: The D Programming Language, Version 1.0
- Internetnews.com: When Is D Better Than C? When It's a Language.
- Bitwise magazine:The Great OOP Debate
- Bitwise 杂志 12 期，May 2006：Bitwise 采访 Walter Bright 和 Dermot Hogan 提问：Is the D language C 'Done Right'?。
- Windows Developer Network 秋季 2003 期 比较 **D** 同 C、C++、C# 和 Java。
- Dr. Dobb's Portal D's the Real Deal -- Finally
- Dr. Dobb's January 2005 有一个关于 **D** 的封面故事
- Dr. Dobb's February 2002 封面故事 The D Programming Language
- Slashdot: C, Objective-C, C++...D Future Or failure?
- Slashdot: The D Language Progresses
- Slashdot: The D Programming Language
- Digg: The Next Generation Programming Language: D
- Bruce Eckel's OO Programming Newsletter #30: New programming language: D
- OSnews April 19, 2004: A, B, C, ...D 编程语言

31.6 比较和基准(Benchmarks)

- 语言流行度排行
- 计算机语言比拼基准
- 99 Bottles of Beer（99 瓶啤酒）

31.7 Forums(论坛), Blogs(博客), Journals(期刊)

- D 期刊。
- Mike Parker 的 D 博客。
- Lars Ivar Igesund 的 D 博客.

31.8 D 的咨询(Consultants)、专业服务(Professional Services)

- Igesund Enterprise Software - 系统设计、开发和集成服务
- OptiCode Windows 和 Linux 软件开发的专业服务。

31.9 杂项

- dsource.org 是一个关于 D 编程语言的开源社团。它的目标就是提供一个环境帮助开源开发，它包括有论坛、源码控制、错漏追踪(bug-tracking)以及发行包(distribution)。
- DMedia 专注于使用 D 来进行多媒体开发。它将包括 2d 和 3d 图形、声明编程以及更多的东西。
- 在你的社团里检验一个 D 用户的组。
- AFB 的 D 语言
- 为什么 D 不是 Java? 作者：Daniel Yokomiso。
- Burton Radons 做了一个 D 的 linux 移植。这个对于所有想建造一个新的用于 D 的支持不同处理器的代码生成器的人，是一个非常有用的指导。
- minddrome networks。
- Pavel 的 DedicateD 站点。有大量带有源码的 D 项目、FAQ、D 新闻等等。
- The Pragmatic Programmers
- Gnu D 编译器，位于 SourceForge。
- The D Journal (就快来了！)
- The Code Moon
- OpenD 项目。
- Open Directory:Computers:Programming:Languages:D。
- DDevil 的 语言比拼 D 的基准。
- Dprogramming.com
- DML，HTML 中嵌入 D
- Felix 写道：“这个压缩包包含有一个用于 DMD、DMC、BCC 编译器的 Windows shell。它也可以扩展到所有其它的编译器。配置文件语法相当严格，但很有效。有种 Kriate 的 shell，只是更为高级。”
- Justin's D Stuff
- MKoD - D 编程语言 (MKoD == Magikal Kingdom of Dreams)
  一直扩展中，目前该站点包含有项目（比如：象 Financial Pkg）、实例（以普通的 D 代码方式，或者是 D 跟 Windows API 的混合方式写成）、由 C 转换到 D 的 Win32 API（比如：ODBC32 和 WinCon）、技术参考（D 的数据类型/大小图表）、一般信息（比如：怎么样安装 D 以及 D 编程 101）、以及 D 站点标语(banner)。
- Ares
- Elephant IDE
- D DBI 用于 D 语言的数据库独立接口
- Harmonia
- Pelcis Place：D 教程。

- 未排序的 D 链接。
- Walter Bright 的 SDWest 2004 关于 D 的介绍。
- rulesPlayer - MPlayer 的 Windows 前端

31.10 日语

- 被翻译成日语的 Digital Mars D 站点
- D 语言的一个教程。
- D 的维基站点
- 使用 using Direct3D9 in the D language。
- Windows 头文件。
- TUMIKI Fighters 游戏的 D 实现。

31.11 德语

- Programmiersprache D, Deutsches D Buch, 作者：Manfred Hansen

31.12 投稿

如果你有任何 **D** 代码、文档、图像或者对 **D** 程序员有意义的网页，请把这些链接电邮到 Digital Mars。