

代码


魏家明◎编著

CODE
Structure

结构

程序员必备
打造完美代码

蓝宝书

 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

代码结构

魏家明 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书对如何优化代码结构做了深入的探讨，分为5个部分：编程问题与原则、编程格式与风格、让代码更容易读、如何做代码重构和C语言一些要素。

本书对这些部分做了重点的探讨：代码中存在的各种问题，编程时要遵循的原则，编程时要注重的格式、注释和名字，如何让表达式和控制流简单易读，如何消除代码中的重复冗余，如何切分代码和少写代码等。另外，本书还探讨了C语言设计中的一些要素和常见问题。

本书适合具有一定编程基础的人阅读，读者通过阅读本书，可以规范代码设计，提高设计能力，优化代码结构，提高代码的可读性，从而提高代码的各种特性。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

代码结构/魏家明编著. —北京：电子工业出版社，2016.8
ISBN 978-7-121-29603-1

I. ①代… II. ①魏… III. ①程序语言 IV. ①TP312

中国版本图书馆CIP数据核字（2016）第179694号

策划编辑：牛平月

责任编辑：张 剑

文字编辑：牛平月

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1 092 1/16 印张：17.5 字数：450 千字

版 次：2016 年 8 月第 1 版

印 次：2016 年 8 月第 1 次印刷

印 数：3 000 册 定价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）88254454。

序 言

通过编程语言可以设计两种系统：软件系统和硬件系统。例如，使用 C 语言设计操作系统和应用软件，使用 Verilog 设计 ASIC 芯片和 FPGA 应用。

系统的生命周期包含两个阶段：开发期和运营期。在开发期，公司投入大量的人力和物力用于开发系统，所以总是希望系统能发挥最大的能力；在运营期，为了尽可能地延长系统的运营时间，就需要投入大量的工作进行系统维护。

系统维护一般包括三大类：一是纠正性维护，用来修改系统中存在的错误和缺陷；二是适应性维护，为了让系统适应不断变化的环境，需要对系统进行修改和扩充；三是完善性维护，为了让系统提升性能、扩大功能、拓展应用范围，需要对系统进行扩充和移植。

在这三大类的系统维护工作中，第二类和第三类维护所占的份额最大，约占总维护工作的 80% 左右。所以，系统的可维护性是首要考虑的问题，系统的运营过程就是维护系统生命价值的过程。根据调查表明，系统维护成本已占到系统生命周期成本的 70% 以上。这表明系统维护的难度越来越大，已成为目前系统开发所面临的最大问题。

影响系统维护的方面有很多：系统架构设计不合理、不灵活，难以修改和扩充，编写代码时不注意代码的内在质量，代码可读性和可理解性差，缺乏相关的文档资料，代码和文档不相符合，员工频繁离职导致工作脱节等。

随着系统开发越来越复杂，就需要采取科学的管理方法和优秀的设计技术，严格把控系统设计的质量，使得系统设计按照有条不紊的方式进行，提高代码编写的生产率，提高代码的可靠性、可读性、可理解性和可维护性，从而降低系统开发和维护的成本。

为了设计易于维护的系统，就要设计出通用、灵活、易于维护的系统架构，系统设计模块化，模块高内聚低耦合，遵守编码风格，优化代码结构，提高代码设计的质量，保证代码的可读性、可理解性、可测试性、可维护性、可移植性等内在特性，从而降低系统的维护难度，延长系统运营的生命周期。

作者根据上面这些开发和维护的需求，对实际代码设计进行分析总结，同时阅读多位编程大师的书籍，查找大量的资料，从而完成了本书的编写。作者在本书中讨论了这些内容：代码中存在的各种问题，编程时要遵循的原则，编程时要注重格式、注释和名字，如何让表达式和控制流简单易读，如何消除重复冗余的代码，如何切分代码和少写代码，C 语言的一些要素等。

读者通过阅读本书，可以规范代码设计，提高设计能力，优化代码结构，提高代码的可读性，从而提高代码的各种特性。

GigaDevice Semiconductor (Beijing) Inc.

北京兆易创新科技股份有限公司

副总经理 曹堪宇博士

2016 年 5 月 12 日

前言

这是一本向编程大师致敬的书，也是一本对代码设计思考的书，还是一本对实际代码设计有帮助的书。

Martin Fowler said: “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

有些人花了很大的精力编写代码，但只是满足于代码能工作就行，从未想过让代码有好的结构和可读性，也就不会花费精力调整代码，从而导致代码很糟糕。其实，没什么比糟糕的代码更差劲的了，因为糟糕的代码只会降低代码的可读性，只会拖团队的后腿，只会让系统的成本增加。

系统的成本在于长期维护的成本，这是因为为了尽量降低维护时出现缺陷和错误，就有必要透彻地理解系统在干什么。当系统变得越来越复杂，设计人员就需要花费越来越多的时间来理解系统，但还是可能会有很多的困惑和误解，而代码中存在的各种各样的问题还会加剧设计人员对代码的困惑和误解。

代码设计中存在很多问题，例如命名混乱、名不副实、格式混乱、注释混乱、重复冗余、臃肿庞大、晦涩难懂、过度耦合、滥用变量、嵌套太深、代码混杂、太多警告、过度设计、陈旧腐败等，这些都是在代码结构设计上出了问题，这些问题就像“死气沉沉的沼泽”一样，让设计人员痛苦不堪，艰难地跋涉。

代码设计中存在的各种问题，会影响代码的设计质量，影响代码的灵活性、可读性、可理解性、可维护性、可测试性、可移植性、可重用性等，还会影响设计人员的干劲与活力，甚至影响设计人员间的协作精神。

所以代码设计应该具有良好的代码结构，具有良好的可读性和表达力，能够清晰地表达设计者的意图，同时也是对工作的负责，对团队的尊敬。设计人员在理解代码时花费的时间越少，就越能减少设计缺陷和错误，越能减少维护成本，同时设计者本人也会赢得其他人员的尊敬。

本书针对如何优化代码结构，通过以下 5 个部分进行讨论。

- 1) 编程问题与原则：本部分讨论代码质量、代码问题、人员问题、编程原则和编程之道，因为只有清楚了存在的问题，掌握了编程原则，才有可能写出更好的代码。
- 2) 编程格式与风格：本部分讨论格式优美、注释合理、名字定义等方面，还介绍 Emacs 的使用，这些方面对设计人员非常重要。
- 3) 让代码更容易读：本部分讨论如何让编写出的代码更容易阅读，包括消除警告、精心用变量、表达式易读、控制流易读、设计好函数。
- 4) 如何做代码重构：本部分讨论代码重构的好处和方法，包括消除重复、代码切分、少

写代码、简化代码，还讨论代码生成、代码测试。

5) C 语言一些要素：本部分讨论 C 语言的一些要素，讨论一些容易混淆和出错的地方，还讨论 C 语言的一些好用法。

如果你没有优化代码结构的意愿，那么说再多也没用，当你看见自己混乱的代码时，你也不会感到害臊；当你看见别人优美的代码时，你也不会有任何感觉。

如果你有优化代码结构的意愿，就要注重编码格式、名字定义、代码易读、消除冗余、代码重构等方面的学习，并积极实践这些方法，从一点一滴做起，最后就会获得很好的代码结构。

本书举例主要以 C 语言为主，但也会有 Verilog 和 Perl 的例子。有些人觉得它们是三种完全不同应用的语言，竟然在一本书里描述，可能觉得不可接受，但是代码设计都有共性，都需要注意代码结构，关键在于灵活掌握这些设计原则，那么对任何编程语言你都会游刃有余。

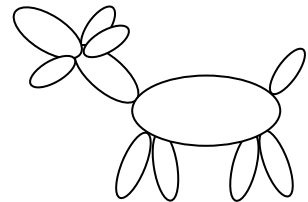
衷心地感谢曹堪宇博士，他在百忙之中为本书题写了序言，他带领我们做系统设计的时候，我深深地被他的精益求精的精神所打动。

衷心地感谢我的朋友燕雪松、张茜歌等人帮我审稿，帮我在书中找到好多的缺陷和错误。

衷心地感谢网上的朋友们，因为我采用了很多的网上资料，感谢你们的无私奉献。

衷心地感谢电子工业出版社的支持，正是由于策划编辑牛平月的密切联系和各位编辑的认真工作，才使得本书得以顺利地与读者见面。

如果您在本书中发现有缺陷或者错误的地方，或者您对本书存有模糊或者疑惑的地方，请通过 QQ 或者邮件与我联系，我的 QQ 号码是 943609120，您的任何反馈都是令人欢迎的。我还建立了一个名为“代码结构”的 QQ 群，群号码是 482433665，欢迎大家加入，共同探讨各种与代码设计有关的各种问题。



魏家明

2016年6月30日

目 录

第一部分 编程问题与原则

第 1 章 美的设计	2	3.18 陈旧腐败	12
1.1 美学观点	2	3.19 停滞不前	12
1.2 代码可读	2	3.20 不可扩充	12
1.3 适用范围	3	3.21 最后总结	12
第 2 章 代码质量	4	第 4 章 人员问题	13
2.1 外在特性	4	4.1 思维定势	13
2.2 内在特性	4	4.2 思维顽固	14
2.3 一个故事	5	4.3 小中见大	14
2.4 提升质量	5	4.4 懒虫心理	14
第 3 章 代码问题	7	4.5 粗枝大叶	15
3.1 最混乱的	7	4.6 巧合编程	15
3.2 命名混乱	8	4.7 应付差事	15
3.3 名实不副	8	4.8 固步自封	15
3.4 格式混乱	9	4.9 疲惫不堪	15
3.5 注释混乱	9	4.10 环境混乱	16
3.6 重复冗余	9	4.11 管理失职	16
3.7 臃肿庞大	9	4.12 个人性格	16
3.8 晦涩难懂	10	第 5 章 编程原则	17
3.9 过度耦合	10	5.1 高内聚低耦合	17
3.10 滥用变量	10	5.2 设计模式	18
3.11 嵌套太深	10	5.3 编码风格	19
3.12 代码混杂	10	5.4 干干净净	20
3.13 不确定性	11	第 6 章 编程之道	21
3.14 太多警告	11	6.1 注重质量	21
3.15 鸡同鸭讲	11	6.2 遵守规则	22
3.16 过度设计	11	6.3 简洁编程	22
3.17 基础不好	11	6.4 整洁编程	23

6.5 快乐编程.....	24	6.10 代码重构.....	26
6.6 团队协作.....	25	6.11 深入学习.....	27
6.7 测试驱动.....	25	6.12 寻求诗意.....	27
6.8 考虑全局.....	25	6.13 程序员节.....	28
6.9 代码切分.....	26		

第二部分 编程格式与风格

第 7 章 使用 Emacs.....	30	9.5 分组成块.....	45
7.1 Emacs 介绍.....	30	9.6 添加空白.....	46
7.2 Emacs 安装.....	31	9.7 书写语句.....	47
7.3 常用快捷键.....	31	9.8 书写表达式.....	47
7.4 作者的“.emacs”.....	32	9.9 Verilog 部分.....	48
7.5 cua-mode.....	33	9.10 保持一致.....	49
7.6 shell buffer.....	34	9.11 代码例子.....	50
第 8 章 快速排序.....	35	第 10 章 注释合理.....	52
8.1 算法简介.....	35	10.1 无用的注释.....	52
8.2 C/C++ 语言.....	35	10.2 有用的注释.....	58
8.3 Java.....	36	10.3 如何写注释.....	62
8.4 Perl.....	36	第 11 章 名字定义.....	65
8.5 Verilog.....	37	11.1 命名方法.....	65
第 9 章 格式优美.....	39	11.2 命名.....	67
9.1 合理安排.....	39	11.3 命名规则.....	73
9.2 横向缩进.....	40	11.4 名字使用.....	77
9.3 纵向对齐.....	43	11.5 SPEC 定义.....	77
9.4 顺序书写.....	44		

第三部分 让代码更容易读

第 12 章 消除警告.....	84	第 13 章 精心用变量.....	89
12.1 不可忽视.....	84	13.1 修改变量的名字.....	89
12.2 警告类型.....	86	13.2 进行变量初始化.....	90
12.3 打开警告.....	87	13.3 减少变量的个数.....	91

13.4	缩小变量作用域	92	15.5	判断中的赋值语句	106
13.5	减少变量的操作	95	15.6	if 语句的逻辑顺序	107
第 14 章	表达式易读	96	15.7	使用 “?:”	108
14.1	糟糕的表达式	96	15.8	使用 switch	108
14.2	使用中间变量	96	15.9	使用 return	109
14.3	使用等价逻辑	98	15.10	选择 for/while	109
14.4	简化判断逻辑	98	15.11	少用 do/while	110
14.5	使用宏定义	99	15.12	少用 goto	112
14.6	使用查找表	99	15.13	语句对比	113
14.7	注意操作符	101	15.14	减少嵌套	114
14.8	简洁的写法	102	15.15	减少代码	117
第 15 章	控制流易读	104	第 16 章	设计好函数	119
15.1	组织直线型代码	104	16.1	不好的函数	119
15.2	判断中的表达式	105	16.2	好的函数	119
15.3	判断中的注意事项	105	16.3	小的函数	120
15.4	判断中的参数顺序	106	16.4	递归调用	121

第四部分 如何做代码重构

第 17 章	代码重构	124	第 20 章	少写代码	145
17.1	为什么重构	124	20.1	合适就刚刚好	145
17.2	重构的好处	124	20.2	保持代码简洁	145
17.3	重构的难题	125	20.3	使用循环解决	146
17.4	实际的例子	125	20.4	熟悉语言特性	147
第 18 章	消除重复	128	20.5	熟悉库函数	147
18.1	代码重复的产生	128	20.6	熟悉系统工具	149
18.2	代码重复的后果	128	第 21 章	简化代码	150
18.3	代码重复的解决	129	21.1	重新设计代码	150
18.4	消除重复的例子	129	21.2	寻找更好算法	152
第 19 章	代码切分	133	第 22 章	代码生成	155
19.1	抽取独立的代码	133	22.1	配置 Linux 的内核	155
19.2	设计通用的代码	135	22.2	生成寄存器的代码	156
19.3	简化已有的接口	137	22.3	生成 Benes 的代码	157
19.4	一次只做一件事	138	第 23 章	代码测试	161
19.5	让函数功能单一	142	23.1	测试中问题	161
19.6	删除无用的代码	144	23.2	测试的原则	162

23.3 设计要更好	162	23.5 测试智能化	164
23.4 提高可读性	162	23.6 定位 Bug	167

第五部分 C 语言一些要素

第 24 章 关键字	170	29.4 复合常量赋值	208
第 25 章 类型	172	29.5 函数中的变长数组	208
25.1 内部数据类型	172	29.6 结构中的灵活数组	209
25.2 新增数据类型	173	29.7 数组作为函数参数	209
25.3 enum	173	29.8 数组和指针	210
25.4 struct	174	第 30 章 指针	212
25.5 union	175	30.1 指针的说明	212
25.6 typedef	175	30.2 啰嗦的指针	213
25.7 复杂的数据类型	177	30.3 void *	214
25.8 Endian 问题	177	30.4 restrict	215
第 26 章 变量	179	30.5 多级指针	216
26.1 声明和定义	179	30.6 数组指针和指针数组	217
26.2 变量分类	179	30.7 函数指针和指针函数	219
26.3 const 变量	183	30.8 malloc	222
26.4 volatile 变量	185	30.9 alloca	223
26.5 混合声明	187	30.10 指针使用中的问题	223
第 27 章 常量	188	第 31 章 语句	225
27.1 常量类型	188	31.1 if	225
27.2 常量定义	189	31.2 switch	225
27.3 常量区分	189	31.3 for 和 while	226
27.4 其他问题	189	31.4 do { ... } while	226
第 28 章 操作	190	31.5 break	227
28.1 操作符表格	190	31.6 return	228
28.2 操作符解释	192	31.7 goto	228
28.3 强制进行类型转换	199	31.8 exit()	229
28.4 运算时的类型转换	199	31.9 复合语句	229
28.5 赋值时的类型转换	203	30.10 空语句	229
第 29 章 数组	206	第 32 章 函数	231
29.1 数组的说明	206	32.1 void	231
29.2 初始化	206	32.2 static	231
29.3 字符串	207	32.3 inline	231

32.4	函数原型	232	35.2	封装函数	250
32.5	参数可变	233	35.3	使用断言	251
32.6	其他讨论	234	第 36 章	内存映像	254
第 33 章	库函数	235	36.1	程序编译后的 section	254
33.1	使用 getopt()	235	36.2	程序运行时的内存空间	255
33.2	使用 qsort()	236	36.3	简单的 malloc.c	255
33.3	文件模式问题	236	第 37 章	汇编语言	258
33.4	返回值的问题	238	37.1	如非必要	258
33.5	控制字符问题	238	37.2	熟悉 ABI	259
33.6	缓冲区问题	239	37.3	汇编例子	259
33.7	可重入问题	240	第 38 章	GCC 特色	261
第 34 章	预处理	242	38.1	MinGW	261
34.1	文件包含	242	38.2	执行过程	262
34.2	宏定义	244	38.3	内嵌汇编	262
34.3	条件编译	248	38.4	__attribute__	264
34.4	其他命令	249	参考文献	266	
34.5	预处理输出	249			
第 35 章	错误处理	250			
35.1	错误检查和处理	250			

编程问题与原则

有些人花了很大的精力编写代码，但只是满足于让代码能工作就行，从没有想过让代码有好的结构和好的可读性，也就不会花费精力调整代码，从而导致代码很糟糕。其实，没什么比糟糕的代码更差劲的了，因为糟糕的代码只会降低代码的可读性，只会拖团队的后腿，让系统的成本增加。

本部分讨论代码质量、代码问题、人员问题、编程原则和编程之道，因为只有清楚了编程中存在的问题，掌握了编程原则，才有可能写出更好的代码。

第 1 章

美的设计

1.1 美学观点

“程序设计是一门艺术”这句话有两个意思：一方面是说，程序设计像艺术设计一样，深不可测，奥妙无穷；另一方面是说，程序员像艺术家一样，也有发挥创造性的无限空间^[14]。

Donald Knuth 认为“计算机科学”不是科学，而是一门艺术。它们的区别在于：艺术是人创造的，而科学不是；艺术是可以无止境提高的，而科学不能；艺术创造需要天赋，而科学不需要。所以 Donald Knuth 把他的 4 卷本巨著命名为《计算机程序设计艺术》(*The Art of Computer Programming*)。

Donald Knuth 不仅是计算机学家、数学家，而且是作家、音乐家、作曲家、管风琴设计师。他的独特的审美感使他拥有广泛的兴趣、多方面的造诣，他的传奇般的创造力也是源于这一点。对于 Donald Knuth 来说，衡量计算机程序是否完整的标准不仅在于它是否能够运行，他认为计算机程序应该是雅致的，甚至可以说是美的。计算机程序设计应该是一门艺术，一个算法应该像一段音乐，而一个好的程序就应该像一部优秀的文学作品。

Bjarne Stroustrup, C++语言发明者，说“我喜欢优雅高效的代码。代码逻辑应当直截了当，让缺陷难以隐藏；应当减少依赖关系，使之便于维护；应当依据分层战略，完善错误处理；应当把性能调至最优，省得引诱别人做没规矩的优化，搞出一堆混乱来”。他特别使用“优雅”这个词来说明“令人愉悦的优美、精致和简单”^[18]。

一个人的美学观点会影响他的程序设计，因为 Knuth 有这么多的艺术爱好，所以他把程序设计看成艺术设计，在程序设计中要体现出程序的美。同样，当 Bjarne Stroustrup 编写优雅且高效的代码的时候，他也是在程序设计中寻求美。

有些人的美学观点是简单和谐、整洁有序；有些人的美学观点是宏大华丽、空洞无味；还有些人的美学观点是乱七八糟、凑合了事。你的美学观点是什么呢？有些人颇为自负，自我感觉良好，以为领悟到了编程的真谛，看到代码可以运行，就洋洋得意，可是却对自己造成的混乱熟视无睹。那堆“可以运行”的程序，就在眼皮底下慢慢腐坏，然后废弃扔掉。

所以程序设计要遵从自己的方法论，要体现自己的奇思妙想，要让设计有更长的生命力，而不是最后沦为豆腐渣工程。程序设计应该像设计艺术作品一样，要寻求美、设计美，要精雕细琢、仔细打磨，要经历痛苦与无奈，还要经历快乐与自得。

1.2 代码可读

在人们眼中，程序员大部分时间是用在编写和调试代码上，但是其实程序员大部分时间是用在阅读和理解代码上，包括自己写的代码，也包括别人写的代码。为什么要花这么多时

间阅读和理解代码呢？因为一个系统不是从零开始设计的，代码有各种来源，有自己设计的，有别人设计的，也有从别的系统里拿过来的。一方面要让自己设计的代码正常工作，另一方面要消化吸收其他的代码。这就要求程序员不停地测试，不停地“Debug”，不停地阅读理解代码。因此代码要具有很好的可读性，才能减少阅读理解的时间。

如果有人对你说：“Hi，过来帮我看一下这个问题出在哪里。”然后他给你展现的是一堆乱七八糟的代码。你心里怎么想，你是不是在想：“这位老兄是在逗我玩吗？竟然好意思拿这样的代码给我看。”在阅读代码的过程中，人们说脏话的频率是衡量代码内在质量的唯一标准，所以代码最重要的读者不是编译器、解释器或者电脑，而是人，好的代码应该能让人快速阅读并理解。

现实总是那么不尽如人意，很多人的代码中都有着一些阻碍代码可读的问题，这些问题包括：命名混乱、名实不副、格式混乱、注释混乱、重复冗余、臃肿庞大、晦涩难懂、过度耦合、滥用变量、嵌套太深、代码混杂、太多警告、过度设计、陈旧腐败等，反正就是各种各样的混乱。这些问题是现实存在的，是显而易见的，但是对于这些问题很多人都毫不在意。

当你意识到你的代码中存在的问题后，你才能想办法规范自己的代码设计，提高代码的可读性，从而提高代码的可理解性、可扩展性、可测试性、可维护性、可移植性，从而你才会有这种感觉：“恢恢乎其于游刃必有余地矣……提刀而立，为之四顾，为之踌躇满志，善刀而藏之。”

1.3 适用范围

本书主要探讨这些问题：编程问题与原则、编程格式与风格、让代码更容易读、如何做代码重构和 C 语言一些要素。

本书举例主要以 C 语言为主，但是并不局限于 C 语言，也会举一些 Perl 和 Verilog 的例子。有些人会觉得这三种语言跨度太大了，包含了软件语言、脚本语言和硬件语言。但是编程语言和编程方法都具有共性，本书所探讨的代码结构也具有共性。例如名字定义、格式优美、消除冗余、代码切分，C 语言可以这么做，Verilog 可以这么做，其他语言同样也可以这么做。

编写代码不只是用来设计软件，还包括用 Verilog 设计硬件电路，也包括用 MatLab 做科学仿真，所以本书很少使用“软件”这个词，而是使用“系统”表示所有可以用代码实现的东西，这样更具有实在的意义；所以本书所说的代码设计是广义的，所讨论的原则是普遍适用的；所以本书的程序员是广义的，只要是编写代码的人，都可以称为程序员。

C 和 Perl 程序构成的基本单位是函数，Verilog 程序构成的基本单位是模块，但是也夹杂着一些函数和任务。本书以讨论函数为主，以讨论模块为辅，因为设计思想是相同的，所以很多提到函数的地方也适用于模块。

本书不只适用于软件开发人员，也适用于所有编写代码的人员。本书讨论的方法简单明了、切实可行，会对你有所帮助，会帮助你规范代码设计，优化代码结构，提高代码的可读性。

第 2 章

代码质量

对于一个完整的系统来说，需求和架构是宏观的，设计和测试是微观的。只有做到宏观和微观的协调统一，才能做出一个完美的系统，否则最后只能是一个烂尾的系统。系统的质量归根结底反映到代码的质量上，代码的质量不仅决定系统的质量，还直接影响系统的成本。代码同时拥有外在和内在的质量特性，我们在设计时需要着重关注它们。

2.1 外在特性

外在特性指的是该系统的用户所能感受到的部分，包括下列内容：^[19]

- 1) 正确性：指系统在规范、设计和实现方面错误的稀少程度。
- 2) 易用性：指系统在学习和使用上的容易程度。
- 3) 高效率：指系统是否能尽可能少地占用资源，包括内存使用和执行时间。
- 4) 可靠性：指系统是否能够可靠地运行，应该有很长的平均无故障时间。
- 5) 完整性：指系统既能确保数据能够正确地访问，又能阻止不正确和未经授权的访问。
- 6) 适应性：指系统在不做修改的情况下，能够在其他应用或者环境中使用的范围。
- 7) 精确性：指系统输出结果的误差程度，尤其当输出结果是数量值的时候。
- 8) 健壮性：指系统在接收无效输入或者处于压力环境时继续正常运行的能力。

在以上这些特性中，有一些特性是相互重叠的，但它们都有不同的含义，并且在不同的场合下，重要性也有所不同。

外在特性是用户关心的唯一特性，用户只会关心系统是否正确运行，只会关心系统是否容易使用，而不会关心代码是否具有很好的可读性，不会关心代码修改起来是否很容易，不会关心代码是否具有很好的结构。

2.2 内在特性

内在特性指的是系统内在的质量特性，包括下列内容：^[19]

- 1) 可读性：指阅读并理解代码的难易程度，尤其是在细节语句的层次上。
- 2) 可理解性：指在系统组织和细节语句的层次上理解整个系统的难易程度。与可读性相比，可理解性对系统提出了更高的内在一致性要求。
- 3) 可维护性：指是否能够很容易地对系统修改缺陷和错误，提高运行性能，增加新的功能。
- 4) 可测试性：指的是可以进行何种程度的单元测试或者系统测试，以及在何种程度上验

证系统是否符合要求。

- 5) 可移植性：指一个系统是为特定用途或者环境而设计的，那么当该系统被用于其他用途或者环境的时候，需要对系统修改的难易程度。
- 6) 可重用性：指系统的某些部分可被应用到其他系统的程度，以及此项工作的难易程度。

内在特性和外在特性并不能完全割裂开来，因为在某些层次上，内在特性会影响某些外在特性。如果一个系统无法从内部理解或者维护，那么其缺陷也是很难修正的，而这又会影响到其正确性和可靠性等外在特性。如果一个系统很刻板，无法根据用户的需要进行修改，那么就会影响到其可用性这一外在特性。

我们需要清楚的是：对于一个系统需要什么样的特性，这些特性间在什么时候会发生什么样的相互作用。让所有的特性都表现得尽善尽美是很困难的，需要根据这些相互竞争的目标寻找出一套最优化的解决方案。

2.3 一个故事

20 世纪 80 年代末，有家公司编写了一个很流行的应用软件，许多专业人士都买来用。但是，后来的发布周期就开始拉长，缺陷总是不能修复，装载时间越来越久，崩溃的概率也越来越大。Martin 至今还记得他在某天沮丧地关掉这个软件之后，从此就不再用它。在那之后不久，该公司就关门大吉了^[18]。

20 年后，Martin 见到那家公司的一位早期雇员，问他当年发生了什么事。他的回答让 Martin 越发地感到恐惧。原来当时他们赶着推出产品，代码写得乱七八糟，特性越多，代码也越来越乱，最后他们再也没有办法管理这些代码。所以，是糟糕的代码毁了这家公司。

在某些公司中，代码质量被认为是次要目标，快速而糟糕（quick and dirty）的编程成了普遍的现象，那些胡乱堆砌劣质代码并能“快速完成工作”的程序员受到公司的重视，而那些编写出高质量代码并在发布之前完成工作的程序员则受到公司的轻视。既然如此，谁还会把代码质量当作他们的头等大事呢？

如果你是一位有经验的程序员，肯定遭遇过这种困境，我们有专用的词来形容这种困境：沼泽（wading）。当我们阅读修改调试那些糟糕代码的时候，我们就如同在杂草丛生、水潭密布、泥沼暗藏、望不到尽头的沼泽地里跋涉，我们拼命地想找到出路，期望有点什么线索能启发我们到底发生了什么事情，但是目光所及，只是越来越多的死气沉沉的代码。

2.4 提升质量

代码的内在质量要靠编程规范来保证，但是编程规范在很多公司里无人问津。很多人一边在为糟糕的代码烦恼，却又一边继续编写着糟糕的代码。程序员之间的相互尊重体现在他们所编写的代码上，他们对工作的负责也体现在这里，而糟糕的代码能体现出对同事的尊重和对工作的负责吗？

我们都希望编写出高质量的代码，通往高质量代码的方法有两种：一种是事后修补，另一种是从开始就关注质量。

前者属于常见的工作方法，就是传统的瀑布式开发方法，它要求进行大量的测试，当你以为已经完工了，到头来你会发现还有许多工作要做，需要反反复复地分析、设计、编码和测试。

后者是一种可以让你对系统有信心、可以长年维护系统的方法，它要求根据系统的外在和内在特性，明确定义出代码质量的目标，在设计和编码中要遵循众多规则，同时不断地进行着代码重构。

第 3 章

代码问题

Martin Fowler 在《重构》中有一句经典的话：“任何一个傻瓜都能写出计算机可以理解的程序，但是只有优秀的程序员才能写出人类容易理解的程序。”有些程序员能够快速编写出可运行的代码，但是代码晦涩难懂让人眩晕，请问这样的代码有人愿意阅读并维护吗？

汪曾祺写过一篇短篇小说《云致秋行状》，其中有这样一段：云致秋经常起草一些向上面汇报的材料，翻翻笔记本，摊开横格纸就写，一写就是十来张。写到后来，写不下去了，就叫我：“老汪，你给我瞧瞧，我这写的是个什么呀？”我一看：哩哩啰啰，噜苏重复，不知所云。他写东西还有个特点，不分段，从第一个字到末一个句号，一气到底，一大篇！经常得由我给他“归置归置”，重新整理一遍。他看了说：“行！你真有两下。”我说：“你写之前得先想想，想清楚再写呀。李笠翁说，要袖手于前，才能疾书于后啊！”他说：“对对对！我这是疾书于前，袖手于后！写到后来，没了辙了！”

本着“各扫门前雪”的原则，笔者本不愿意多说，不愿意对别人的代码指手画脚，但是在现实中总是有很多的不如意代码，于是不得不说出来。这些不如意的代码中经常充斥着各种问题，有些程序员写的代码就和云致秋写的汇报材料一样，“邋邋拉拉，噜苏重复”。有些人觉得作者说的言重了，那么就仔细检查一下自己的代码，看看自己的代码是否存在这些问题，“有则改之，无则加勉”。如果你的代码存在这样的问题，那么就要反思一下，为什么会出现这样的问题？怎样才能把这些问题消除掉呢？怎样才能避免再次出现这样的问题呢？

3.1 最混乱的

国际 C 语言混乱代码大赛 (<http://www.ioccc.org/years.html>, IOCCC, The International Obfuscated C Code Contest) 是一项国际编程赛事，从 1984 年开始，每年举办一次 (1997、1999、2002、2003 和 2006 年例外)。大赛的目的是写出最有创意的最让人难以理解的 C 语言代码，审核基准是“滥用混乱代码的程度，以及创造性的滥用”。

IOCCC 是由 Landon Curt Noll 和 Larry Bassel 在 1984 年开始的，那时他俩受雇于国家半导体 (National Semiconductor) 的 Genix 程序移植事业群，比赛的点子就来自他俩在修正某些写得很烂的代码时所记录的笔记。

每年 IOCCC 的比赛规则会张贴在网站上，比赛规则每年不同，这些规则通常是蓄意书写成文，同时伴随着精巧的漏洞，鼓励参赛者发现这些漏洞并创造性地滥用它们。

作品提交后，要经过几个回合的审核。通过最后一轮审核的作品会被归为特别的一类以示嘉奖，例如“最滥用 C 预处理器”或者“最古怪的行为”，并且发表在 IOCCC 官方网站上。该赛事主办方声明：作品被发表在 IOCCC 网站就是竞赛的锦标。

例子：Compute the factorial function

```
#include<stdarg.h> /*Y=\f*/
#include<stdio.h> /*.\x.f(*/
#define z return/*xx)(\x.*/
#include<stdlib.h> /*f(xx)*/
typedef union w w; union w{
#define L(f,y,x,t) w _ ##\
f(w x,...){va_list argh;w\
y; va_start(arch,x);y= *\
(va_arg(arch,w*));va_end\
(arch);z(t);} w f (w fw)\
{r((w){.c= &_## f}, fw);}
int _; void*p;w(*c)(w,...
#define r(x,y)w*k=malloc\
(sizeof(w)<<1); k[0]= x\
; k[1]=y; z((w) {.p= k})
#define l(f,x,t)w _##f(\
w x,...){z(t);}w f={.p\
=(w[1]){.c= &_##f}}};
);;w a(w f,w x){w*d=f
.p;z((*d->c)(x,d+1)
); }
w a_(w f,w fw){r(f,
fw);}w _(w f){w*k=f.p,r=
a(*k,k[1]); free(k) ;z(r);}
L(F,f,x,(w){._=x._?x._*a_(f),
(w){._=x._-1})._:1})l(F_,f,F(f)
)L( W,(f),/*=\*/x,a(f,a_(x,x))
)l( Y,/*=\*/f,a(W(f),W(f))
)int main(){printf("%."f\n",
a(a(Y,F_),(w){._=10})._
/60./60/24/*d */
); }
```

3.2 命名混乱

存在问题：命名杂乱无章、含义不清或者含义错误，这些名字包括目录、文件、模块、函数、变量、常量等。

代码是否具有很好的可读性，名字也很关键，因为名字可以当作一条小小的注释，尽管空间不算很大，但选择一个好名字可以让它承载更多的信息。

3.3 名实不副

存在问题：代码和文档不相符合，很多都对不上，或者代码根本就没有相应的文档。

程序员最烦的两件事：第一件事是给别人的代码编写文档；第二件事是接手别人的代码却没有相应的文档，或者有文档但很混乱。照理说应该先有文档，然后有代码，但是很多时候不是这样的，文档普遍滞后于代码。不管怎样，都要保持文档和代码的一致性，做到文档的及时更新，做到“师出有名”。

3.4 格式混乱

存在问题：格式混乱，对内不好阅读、理解和维护，对外影响公司形象，让人觉得不专业。

请读者阅读这样一段字序颠倒的文字：“研表究明，汉字的序顺并不定一能影阅响可读性，比如你当看完这句话后，才发这现里的字全是都乱的”，是不是这样呢？

当别人阅读代码时，他们对代码的第一印象就是“代码的格式是否整齐”，格式很差劲的代码能有好的可读性吗？

很多人对代码要求的缩进格式都不遵守，对其他的格式要求又能遵守多少呢？对其他的编码风格要求还能遵守吗？

3.5 注释混乱

存在问题：有些代码根本就没有注释，有些代码虽有注释，但是这些注释乱七八糟、糊弄差事、毫无用处、误导理解、格式混乱、过时失效。

很多程序员感兴趣的是代码而非注释，因为前者更能带来成就感，所以注释往往不受重视，所以更容易出现问题。

3.6 重复冗余

存在问题：相同或相似的代码反复出现，代码拖拉混乱、拖泥带水、啰嗦冗余、不好理解、不便维护，只会导致效率的低下和资源的浪费。

当作者面对这样的代码时，就如同听六字真言一样，在作者的耳边反复地念叨着“唵嘛呢叭咪吽”，作者多么希望这样的代码“千言万语，万语千言，最后汇成一句话”啊！

记得作者小时候上育红班，老师教大家写字，很简单的字，比如“土”、“人”、“口”等，还留作业呢。有几个同学就写了满满的好几页，还跟我们显摆，以图获得老师的夸奖，当然笔者是一个字没写的，这说明笔者很懒，但是像他们这样写字又有多大的意义呢？

3.7 臃肿庞大

存在问题：函数代码有非常多的行（数百到数千），结构散乱，分支太多，臃肿不堪，不做有效的代码切分，到处都是细节代码，阅读理解起来很费劲，真是“牵一发而动全身”。

好代码是精美的瓷器，精心地、分门别类地摆放在壁橱里，想取出任意一件都很方便。而滥代码就像一堆胡乱摆放的层层叠叠的瓷器，像一座小山，有的很完美，有的则残破不堪，想从里面拿出任意一件，都会引起这座瓷器山倒塌崩溃。

对于这样的滥代码，当你要修改一个小缺陷，或者增加一个小功能，都会引发一次代码地震。

3.8 晦涩难懂

存在问题：表达式庞大复杂，控制流绕来绕去，代码逻辑混乱、晦涩难懂，难于阅读、理解和维护。

对于这些晦涩难懂的程序，很费劲地看明白了，过了不久，再看还是很费劲。代码应当让人易于理解，让人用最少的时间看懂。

如果只有你一个人从事某个项目，你可能会想，“我是唯一使用这些代码的人，谁会关心是不是有人理解它呢？”即使这样，编写让人理解的代码也是值得的，因为几个月后你可能要重新看这些代码，那时你自己写的代码看上去也已经很陌生。另外，如果别人要加入到你的项目中呢？如果你的代码要移植到别的项目中呢？这些都是要考虑的。

通常来说，代码行数越少越好，但这也不是一定的，代码行数对于代码的可读性只是相对的，过分地压缩代码的行数反到会影响代码的可读性。

3.9 过度耦合

存在问题：模块间存在过度的耦合，牵扯不开，藕断丝连。

耦合是模块间相互连接的紧密程度，是从功能角度来度量模块间的功能联系。模块间耦合的高低取决于模块间接口的复杂性、调用的方式以及传递的信息。模块间的联系越紧密，其耦合性就越强，模块的独立性则越差。

3.10 滥用变量

存在问题：全局变量随意使用，局部变量胡乱使用，函数参数随便定义，而且不考虑变量作用域。

只有遵循全局变量的使用规则，使用全局变量才能是高效的，但是程序员还是倾向于滥用它们。不管是全局变量，还是局部变量，使用时都要格外小心。

3.11 嵌套太深

存在问题：代码中包含太多层次的嵌套结构，很难分清楚代码的嵌套关系。

很多的 if、for、while 等语句，一层一层地嵌套在一起，阅读的时候经常容易把人搞晕，经常要让人琢磨哪个“{”与哪个“}”相匹配，或者琢磨哪个“begin”与哪个“end”相匹配。

3.12 代码混杂

存在问题：代码层次结构划分不好，高层次的代码和低层次的代码混杂在一起。

代码设计要有层次，通过相邻层次间的接口建立联系，不要跨越多级层次建立直接的联系，否则系统分层就没有多大用处，对于系统的扩充和维护都有很大的影响。

3.13 不确定性

存在问题：因为线程执行存在竞争条件，导致执行存在不确定性。

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就会存在竞态条件。导致竞态条件发生的代码称作临界区，通过在临界区使用适当的同步就可以避免竞态条件。Verilog 代码中的执行不确定性有些是由阻塞赋值引起的，或者是由系统中多个不同时钟引起的。

3.14 太多警告

存在问题：对于编译器报出的大量警告毫不在意。这些警告很不和谐，似“眼中钉、肉中刺”一样。

编译时有太多的警告报告，其实这里面就可能存在着 Bug，但是很多人毫不在意，而且报告错误时，当错误与警告混杂在一起时，在众多的警告中查找错误也很费劲。

对于 C 语言，很多警告其实就是 Bug，是编程上的失误或遗漏，例如没有函数原型，例如变量没有初始值就使用。

对于 Verilog，很多警告其实就是 Bug，是编程上的失误或遗漏，会导致仿真或综合失败，例如敏感列表不全，例如端口宽度不匹配。

3.15 鸡同鸭讲

存在问题：因为口头约定、沟通不够、错误理解，导致代码中出现各种问题。

鸡叫“喔喔”，鸭叫“嘎嘎”，或者换一种说法，就是“驴唇不对马嘴”。设计人员之间沟通不够，导致设计出来的代码对接不上。

3.16 过度设计

存在问题：代码的灵活性和复杂性超出所需^[17]。

有些程序员之所以这么做，是因为他们相信自己知晓系统未来的需求，他们试图今天就把方案设计得更灵活、更复杂，以适应明天的需求。随着过度灵活、过度复杂的代码的堆积，团队成员就得在毫无必要的更庞大、更复杂的代码上继续工作。

3.17 基础不好

存在问题：因为最基础的东西没设计好，导致后续一系列问题，例如测试不全、编程不方便，然后摇摇晃晃进行了好几年，只能勉力维持，却不能推倒重建。

每个系统都或多或少存在不完美的设计，刚开始可能注意不到，到后来就会慢慢凸显出来，只能勉力维持。

例如，有些芯片的 Spec 定义不好，有些寄存器的定义很差劲，导致相应的 Interrupt Handler 代码写起来很费劲。

3.18 陈旧腐败

存在问题：经过长时间的设计之后，代码中包含着很多无用的常量、变量、函数、注释、文件、目录，不做有效地剔除。

很多人认为“存在的就是合理的”，不愿意对代码做大的改动，只想在代码上打补丁，最后成了“百衲衣”，这些陈旧腐败的东西对阅读、修改和维护存在很大的干扰。

这些过时陈旧的代码纯属鸡肋，留着无用，放着碍事，看着闹心，有啥可留恋的，难道删除这些代码感到痛心吗？

3.19 停滞不前

存在问题：代码能 Work，就认为 OK 了！以为完成 99%，其实可能只完成 50% 或者更少！有些人的口头语就是：“以后就改。”但是很多时候“以后等于永不 (Later equals never)”。

3.20 不可扩充

存在问题：设计的东西是一锤子的买卖，难以扩充，然后再“头痛医头，脚痛医脚”。

例子，中国公民的身份证号码，15 位时位数不够，扩充到 18 位后又带来了“X”的问题。

例子，千年虫问题，用 2 位数表示年份，只能表示 1900~1999 的年份。

例子，某公司设计过一款芯片，可是适用范围太小，且不支持一些重要特性，很快就被淘汰。

3.21 最后总结

文艺程序员写代码追求让别人看懂，普通程序员追求让自己看懂，“Idiot”程序员则追求让编译器能看懂。半年后看自己当初写的代码，文艺程序员不知道自己写的但很容易看懂，普通程序员知道自己写的但是不太容易看懂，“Idiot”程序员埋头看了半天后拍着桌子吼道：“这是哪个笨蛋写的程序！”

代码中存在的这些问题会严重降低代码的内在质量，严重影响代码的可读性、可理解性、可维护性、可移植性等，这里面可读性是最重要的，因为它是其他特性的基础。如果你能从你的代码中发现这些问题，并能够改掉这些问题，那多美好啊！关键在于你是否有这个意愿！

最后还有一句话：代码就是程序员的脸面，难道你不洗脸吗？难道你不想以干净整洁的脸面给人看吗？

第 4 章

人员问题

代码中存在着各种各样的问题，但是这些问题都是由程序员造成的，从代码中就可以看出程序员的优点和缺点。同时程序员之间也会面对各种问题：矛盾、吵架、无奈、拖拉、笨重、繁杂、低效、自我、设计拖后腿、测试不充分、错误层出不穷。

对于人的缺点，卡耐基的《人性的弱点》中做了很好的分析和总结，这些在程序员的性格上都有体现，有些还会被放大，例如傲气和自负。所以我们应该克服自己的缺点，努力提高自己。

柳依依说：“其实我爸是个程序员，我大姐叫玲玲，二姐叫玲依，三姐叫依玲，我还有个妹妹叫忆初—00、01、10、11、溢出。”这里会讲一些段子，类比一下，搞笑一下。

4.1 思维定势

一位公安局长在路边同一位老人谈话，这时跑过来一个小孩，急促地对公安局长说：“你爸爸和我爸爸吵起来了！”老人问公安局长：“这孩子是你什么人？”公安局长说：“是我儿子。”请你回答：这两个吵架的人和公安局长是什么关系？

对于这个问题，在 100 名测试者中只有两人答对！后来对一个三口之家问这个问题，父母没答对，孩子却很快答了出来：“局长是个女的，吵架的一个是局长的丈夫，即孩子的爸爸；另一个是局长的爸爸，即孩子的外公。”

为什么那么多成年人对如此简单问题的解答反而不如孩子呢？这就是思维定势：按照成年人的经验，公安局长应该是男的，所以从男局长这个心理定势去推想，自然找不到答案；而小孩子没有这方面的经验，也就没有心理定势的限制，因而一下子就找到了正确答案。

思维定势（Thinking Set），也称“惯性思维”，就是按照积累的思维活动经验教训和已有的思维规律，在反复使用中所形成的比较稳定的、定型化了的思维。在环境不变的情况下，思维定势能够让人应用已掌握的方法迅速地解决问题，但当环境发生变化时，它就会妨碍人们采用新的方法。

思维定势的特点是：思维模式强大的惯性或顽固性。优点是能应用已掌握的方法快速解决问题；缺点是消极的思维定势会束缚创造性思维。

例如，如果给你看两张照片，一张照片上的人英俊、文雅，另一张照片上的人丑陋、粗俗，然后告诉你，这两个人中有一个是全国通缉的罪犯，让你指出谁是罪犯，你大概不会犹豫吧！

先前形成的知识、经验、习惯，都会使人们形成认知的固定倾向，从而影响后来的分析和判断，形成思维定势，即思维总是摆脱不了已有“框框”的束缚，表现出消极的思维定势。

我们在编写代码时，总是在按照我们熟悉的一些套路编写，例如，如何定义变量名和函数名，代码是否有很好的结构，代码是否有优美的格式，这些都在展示着我们的思维定势。通过这些思维定势展现出来的代码在有些地方表现得很好，但在有些地方表现得就很差。另外，编写代码的工作是团队协作，每个人都可以看见大家的代码，代码都摆在那里，有的很好，有的很差，但是很多时候，差的代码还是在那里，不会做任何改动，因为编写差代码的人认为自己的代码很好，他们不会想到向那些好代码学习，这就是他们的思维定势。

思维定势还有一个表现就是“习惯成自然”，当你习惯于乱七八糟的环境的时候，你就不会去清理卫生；当你习惯于乱七八糟的代码时候，你就不会去做代码的调整和优化。

4.2 思维顽固

某程序员退休后决定练习书法，于是花重金购买文房四宝。某日，他饭后突生雅兴、点上檀香、铺纸研墨、定气凝神、挥毫泼墨，郑重地写下了一行字：**Hello world!**

思维还有一个特点，就是思维的顽固性，表现在“没有悟性，思维僵化，固步自封”，与此相对的是，思维的灵活性，就是“充满悟性，灵活多变，一点就通”。编写代码是一个思维创造的过程，如果思维顽固，怎么能编写出好的代码来呢？

“江山易改，本性难移”，“三岁看大，五岁看老”，这样的俗语暗示：我们的人格和思维模式如同被模具定了型，难以改变。这些长期以来的思维方式在你的大脑中形成了一个顽固的封闭循环，一直自动地将你引向不理性行为和消极情绪，阻碍目标的达成和问题的解决，影响你的工作和生活。

“不识庐山真面目，只缘身在此山中”，通过对自己思维的反省和思考，把自己从问题中抽离出来，以一种旁观者的角度重新审视事件本身，问题往往会迎刃而解。

4.3 小中见大

我们都知道“小中见大”这个道理，就是从小处可以看出大的问题或道理。有的人写数十行代码都不注意格式，那么当代码规模变得很庞大时，格式是不是就极其混乱了呢？当习惯成自然，以后想改都难改，这也是编写代码中的一种蝴蝶效应吧。

“一屋不扫，何以扫天下”，其实不管是多大的设计，都需要注意各方面的问题，不要因为小，就不重视。

4.4 懒虫心理

人都有惰性，这是人性的弱点，代码能够运行，工作正常，那就满意了，就不再调整，不再优化。有些人觉得代码能工作了，干啥还要调整、还要优化呢？改出问题咋办？不如就这样吧！

但是有一种“懒虫”值得称赞，他们为了代码的可读性和可维护性，为了减少以后的工作量，不厌其烦地仔细调整代码，这是一劳永逸的做法，因为他们是优秀的程序员。

4.5 粗枝大叶

有个程序员下班前给老婆打电话：“老婆，晚饭我带回来吃，你说买些啥？”老婆说：“买1斤包子吧，如果遇到卖西瓜的，就买一个。”结果：程序员买了一个包子回家。

照理说作为一个程序员，做事应该有条理、有计划、有耐性、还细心，但是有一些人工作很不严谨，丢三落四、马马虎虎，如果真是这样，就要考虑一下自己是否适合编写代码这项工作。

另外，编写代码需要多年的积累，但是有一些人编写代码的质量却与其工作时间的长短不成正比，很多工作十年、二十年的人写出的代码可读性依旧很差，真是不忍一读。

4.6 巧合编程

程序员几乎都有过这样的经历：在调试程序很长时间之后，开始疲惫不堪漫无目的地瞎碰，这里试一下，那里改一点，如果凑巧程序可以运行了，便万事大吉。这种工作方式就是靠巧合编程，最后往往会导致一场灾难，因为一旦出错，就会发生连锁反应，很多错误都暴露出来。

如果鞋里进了沙粒，我们就会马上停止前进，取出沙粒，然后再前进，否则我们的脚丫子会磨出水泡，磨破脚掌，然后跛足而行。同样，对于代码中的 Bug，我们是深究其发生原因呢？还是不出其发生原因，只是寻找避开的方法呢？

4.7 应付差事

有些人存在着“做一天和尚，撞一天钟”、“得过且过”的思想，在这样思想的驱使下，工作只能是应付差事，这样下去，怎么能做到“升职加薪，当上总经理，出任 CEO，迎娶白富美，从此登上人生巅峰”呢？

还有一种状态就是疲于应付，天天出 Bug，天天打补丁，费力不讨好，工作无激情，只能凑合了事。这时候就要检讨自己的状况，为什么会这样呢？为什么当初就不考虑周全呢？

4.8 固步自封

如果你把编写代码作为你的职业，那么你就始终处于学习充电的状态，要丰富自己的知识，要接受前沿的科技。可是如果你对你所用编程语言的掌握还是停留在大学毕业时的水平，那么说明你已经非常落后了。你是选择止步不前，还是选择迈步向前，全靠你自己了。

4.9 疲惫不堪

有个程序员自述经常熬夜的三个弊端：“第一，我的记忆力越来越差；第二，我数数经常会数错；第四，我的记忆力越来越差。”可见疲惫的程序员将“第三”说成“第四”，果然是“数数经常会数错”。

编写代码应该是愉快的过程，需要爆发出思维的火花，不是工作时间越长，就越能编写出好的代码来。但是当工作劳累又辛苦，冗长又乏味，胶着于项目中，遭遇各种问题，旧问

题还没有理清，新问题又接连出现，在这种情况下，没有好好地休息，没有奇妙和乐趣，能写出好的代码吗？

4.10 环境混乱

你的屏幕、机箱、键盘、鼠标是不是落满灰尘，键盘里面是不是布满了灰尘、面包渣、方便面？你的工作桌面是不是乱七八糟，胡乱地摆放着导线、电路板、测量仪器、书本资料、饮料、茶叶、小食品？你的被褥是不是胡乱地摊开着，沙发上是不是乱扔着各种东西，水槽里是不是堆放着要清洗的碗筷餐具，灶台、抽油烟机上是不是布满了油污，冰箱里是不是存放着过期食品？

其实，你的工作桌面或者居住环境也能反映出你的代码状态，如果你的环境又脏又乱又差，凭什么相信你的代码写得很优秀呢？

4.11 管理失职

代码质量好不好，也是与管理工作是否到位有关。管理工作不到位，就会为以后的工作埋下各种的隐患。

公司对编码风格和编码质量没有要求，认为代码只要能工作就好。另外，因为项目紧张，只求快速地完成工作，满足客户要求就行啦，于是就更加忽略代码质量。

如果管理人员自己都不注重代码质量，自己编写的代码就很乱，那么他们对代码质量能提出合理的要求吗？他们能要求别人保证代码质量吗？

如果注重代码质量的程序员参与到不在乎质量的团队中，就会发现自己在团队里很难受，会发现自己编写代码的热情一点一点被耗尽。

有的团队还会指定质量官员，把保证代码质量的责任交给他，但是这其实很荒谬，因为代码质量只能来源于团队中所有人的贡献，而非一个人的责任。

4.12 个人性格

“我们就是一群代码猴子，上蹿下跳，自以为领略了编程的真谛。可惜，当我们抓着几个酸桃子，洋洋自得地坐在树枝上，却对自己造成的混乱熟视无睹。那堆‘可以运行’的乱麻一样的程序，就在我们的眼皮底下慢慢地腐败。”^[18]

有的人充满傲气，自以为是，认为“俺的技术就是牛。”

有的人个性张扬，任性随性，认为“俺就是要这么做。”

有的人事不关己，高高挂起，认为“那是他们的事，与俺无关。”

有的人只求速度，不求质量，认为“让代码运行起来就 OK 啦！”

有的人指责别人，开脱自己，认为“他们的代码乱七八糟，毫无道理。”

有的人拖沓散漫，不负责任，认为“俺的代码就凌乱啰嗦，你能咋地吧！”

因为这些人具有如此鲜明的个人特点，导致与他们的合作很困难，导致团队内部暗流涌动。有句话叫“乌鸦落在猪身上，只看到别人黑，没看到自己黑”，当我们张扬自己的个性时，我们是否想到了自己呢？我们自己在别人的眼中又是什么样子呢？所以我们还是要虚心一些，含蓄一些，温和一些，这样为好。

笔者学习驾照的最大体会就是要遵守交通法规，所以笔者非常注意出行安全，坐车时要系安全带，过马路时不闯红灯。在现实生活中我们会看到很多不遵守交规的现象，例如闯红灯、扔杂物、往外吐痰、斗气路怒、开远光灯、连续并线、酒驾醉驾、不打转向灯、开车打手机，这些不遵守交规的现象导致很多的事故，轻则追尾、剐蹭、受伤，重则车毁人亡、多车连撞，有些当事人对于自己的责任还推诿、耍赖、放横，有些场面真是惨不忍睹啊，既危害自己，又危害别人。

“一切皆有章法”，“家有家法，行有行规”，对于编码人员，也有相应的编码原则，可是又有多少人愿意遵守编码原则呢？如果你不遵守编码原则，就会造成各种代码问题。

5.1 高内聚低耦合

高内聚低耦合（High Cohesion and Low Coupling）是判断系统设计好坏的标准，主要是看模块内部的内聚性是否高，模块之间的耦合度是否低。

内聚是从功能角度来度量模块内部的功能联系，是模块内部各个元素彼此结合的紧密程度。模块的内聚性越强，模块的独立性则越好。

高内聚是指：模块内各元素彼此结合的紧密程度高，模块是由相关性很强的代码组成，只负责一项任务，也就是常说的单一责任原则。

耦合是从功能角度来度量模块间的功能联系，表示模块之间相互连接的紧密程度。模块之间耦合的高低取决于模块间接口的复杂性、调用的方式和传递的信息。模块之间联系越紧密，其耦合性就越强，模块的独立性则越差。

低耦合是指：对于一个完整的系统，模块与模块之间，尽可能地让它们独立地存在。也就是说，让每个模块尽可能地独立完成某个特定的功能，模块与模块之间的接口应该尽量少而简单。如果两个模块之间的关系比较复杂，最好考虑做进一步的模块划分，这样有利于模块的修改和组织。

内聚可以分为以下 7 种，它们之间的内聚度由弱到强排列如下：

- 1) 偶然内聚：当模块内各部分之间没有联系，或者即使有联系，联系也很松散，则称这种内聚为偶然内聚，也称为巧合内聚。
- 2) 逻辑内聚。这种模块把数种相关的功能组合在一起，每次被调用时，由传送给模块的参数来确定该模块应完成哪一种功能。
- 3) 时间内聚：把需要同时执行的动作组合在一起形成的模块为时间内聚模块。
- 4) 过程内聚：指构件或者操作的组合方式，允许在调用前面的构件或操作之后，马上调

用后面的构件或操作，即使两者之间没有数据进行传递。

- 5) 通信内聚：指模块内所有处理元素都在同一个数据结构上操作，或者处理使用相同的输入数据或者产生相同的输出数据，有时称为信息内聚。
- 6) 顺序内聚：指一个模块中各个处理元素都密切相关于同一功能且必须顺序执行，前一功能元素输出就是下一功能元素的输入。
- 7) 功能内聚：共同完成同一功能，缺一不可，模块不可再分割。

耦合可以分为以下 5 种，它们之间的耦合度由高到低排列如下：

- 1) 内容耦合：它有多种耦合方式，一个模块访问另一个模块的内部数据，或者一个模块不通过正常入口而转到另一个模块的内部，或者一个模块有多个入口。
- 2) 公共耦合：当两个或多个模块通过公共数据环境相互作用时，它们之间的耦合称为公共环境耦合。
- 3) 控制耦合：如果两个模块通过参数交换信息，交换的信息有控制信息，那么这种耦合就是控制耦合。
- 4) 特征耦合：当把整个数据结构作为参数传递给被调用模块，而且这个数据结构中的所有元素都被使用，那么这是完全正确的。但是，当把整个数据结构作为参数传递而只使用其中一部分数据元素时，就出现了特征耦合。在这种情况下，被调用模块可以使用的数据多于它实际需要的数据，这可能会导致对数据的访问失去控制，从而给系统犯错误提供机会。
- 5) 数据耦合：如果两个模块通过参数交换信息，而且交换的信息仅仅是数据，那么这种耦合就是数据耦合。

高内聚低耦合对系统设计有什么好处呢？事实上，从短期来看，并没有很明显的好处，甚至短期内会影响系统的开发进度，因为高内聚低耦合的系统对设计人员提出了更高的要求。高内聚低耦合的好处体现在系统持续发展的过程中，高内聚低耦合的系统具有更好的可维护性、可扩展性、可重用性，可以更高效地完成系统的开发维护，可以持续地支持业务的发展，而不会成为业务发展的障碍。

架构设计的目的就是在保持系统内在联系的前提下分解系统，实现系统的高内聚低耦合，降低系统开发的复杂性，而分解系统的基本方法就是分层和分割。但是如何对系统分层和分割，分层和分割到什么样的程度，并不是一件容易的事，这方面有各种各样的分解原则，例如面向方面、面向对象、面向接口、面向服务、依赖注入、关注点分离等设计原则。

5.2 设计模式

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的代码设计经验总结。使用设计模式就是为了可重用代码、让代码更容易被他人理解、保证代码的可靠性。毫无疑问，设计模式是系统设计的基石脉络，如同大厦的结构一样；设计模式能使代码设计真正工程化；设计模式对自己、他人和系统都是多赢的。

可复用面向对象系统一般划分为两大类：工具箱和框架。我们平时开发的具体系统都是应用程序，而 Java 的 API 就属于用于应用程序的工具箱；框架是一组用于构成某种特定可复用设计的相互协作的类，例如 Java 的 EJB 就是用于企业计算的框架。

框架通常定义了应用体系的整体结构类和对象的关系等设计参数，以便于具体应用的实现者能集中精力于应用本身的特定细节。框架主要记录应用中共同的设计决策，框架强调设计复用，因此框架设计中必然要使用设计模式。

设计模式有助于对框架结构的理解，成熟的框架通常使用多种设计模式。如果你熟悉这些设计模式，毫无疑问，你将迅速掌握框架的结构。普通开发者如果突然接触 EJB、Spring 等框架，会觉得特别难学、难掌握，如果转而先掌握设计模式，无疑是得到一把剖析 EJB 或 J2EE 系统的利器。

使用设计模式就是为了提高代码的可复用性，那么怎么才能实现代码复用呢？面向对象有几个原则：开闭原则、里氏代换原则、依赖倒转原则、接口隔离原则、合成/聚合复用原则、最小知识原则。设计模式就是实现了这些原则，从而达到了代码复用、增加可维护性的目的。

设计模式在面向对象编程中广泛使用，似乎只可以用在 C++、C# 和 Java 这样的语言中，对于 C 语言这样的函数式语言，似乎没有什么用处，但是从设计模式本身的概念来讲，C 语言也是有自己的设计模式的，例如 Linux 和 uC-OS，它们都是由 C 语言设计的，也有自己的设计模式，如果要扩展或移植它们，也要遵守它们的设计模式。

5.3 编码风格

假设我们写的是文章而不是程序，我们会很自然地把文章划分为章节和段落。如果把整篇文章写成长长的一段，从头至尾洋洋洒洒如念经般“一气呵成”，虽然这对文章的内容没有影响，但这样的文章有谁会愿意读呢？

这是人人都能理解的道理，可是当文章变成程序的时候，就不是每个人都能想得通的了。不仅仅是初学者，甚至是一些熟练的程序员，也会写出凌乱不堪的代码。许多人一定有过这样的经历：亲手写的程序，一年半载之后自己就完全看不懂了，这时就不得不再花时间把原来的思路看明白。

有人会说：代码是给机器运行的，又不是给人看的，写那么好看有什么用？代码确实是给机器运行的，可是机器看它才需要几分钟，其他时间不都是你在看代码吗？开发一个系统要经历很长的时间，并且常常由多人合作完成，如果你把代码写得连自己都看不明白，怎么与别人交流与合作？

有的人觉得现在就是一个人写程序，并不需要和其他人合作，是不是就不用遵守这些编码风格？有句话叫作“平时不烧香，临时抱佛脚”，忽然间你就遵守了编码风格，可能吗？

编码风格就是要求我们写的代码符合一定的格式，达到信、达、雅的要求。这样在满足功能和性能目标的前提下，能够规范代码的编写，增强代码的内在质量，才能够更好地保证功能的正确，更好地阅读理解，更好地交流合作，更好地整理文档。同一个开发团队内，只有保持良好且一致的编码风格，才能更好地提高开发效率。

编码风格其实就像写文章应该分章节、分段一样简单，只要愿意遵守，你的代码可读性就会大大提高，那么别人在看你的代码时，就能感觉到你那良好的编程修养，即所谓“见字如见人”。

5.4 干干净净

如果我们的环境是干干净净的，那么我们的生活会很愉快，同样如果我们的代码是干干净净的，我们的工作也会很愉快。但是为什么有些人会接受脏乱差的环境呢？为什么有些人会写出乱七八糟的代码呢？

我们要随时删除那些无用的代码，保持设计干净、逻辑清晰。我们写的代码不是张旭和怀素的狂草，也不是毕加索的现代艺术，不是写得乱七八糟别人都看不懂，就显得你有多高深。

干干净净的代码就是要求代码整洁、结构合理、层次清晰、注释明了，没有烂代码、重复代码和冗余代码，合理地建立目录，合理地分配到不同文件中。有些人觉得格式不重要，只要能正确运行就行，何必在这上面花费时间呢？其实整理格式花费的时间并不多，我们的大部分时间都是花在设计和调试上。养成习惯之后，注重格式就会自然而然地完成，对自己、对别人、对项目都有益。

写代码就和写文章一样，刚开始可能粗陋无序、结构杂乱，然后就要仔细斟酌推敲，达到你心目中的样子。不要以为能够运行就足够了，还需要做很多整理。代码的设计不是一开始就是整洁的，要经过不断地调整结构，才能达到好的结果。

作者认识一个大学生（现在已经毕业了），问他们的宿舍干净不，他说不太干净，又问他是不是有的宿舍很干净，他说是的。他说宿舍要那么干净干啥，反正大家都习惯了，还说有的宿舍其实更脏。想一想，大学宿舍其实就那么大的地方，每个同学平时多注意一些，平时稍微整理一下，就会很干净，可是他们却不做，没有办法呀。其实干干净净的代码和干干净净的宿舍一样，关键在于你自己有没有想保持干干净净的心，如果没有这个心，那么不管外人如何说，也没有多大的效果。

当我们阅读写得很糟糕的代码时，我们会感觉很难受，我们的心会非常堵得慌；当我们阅读写得很漂亮的代码时，我们会很受启发，就如同“一缕阳光洒入我们的心扉”。好的代码会很明确地告诉你它在做什么，我们使用它会感到很有趣，并且会鼓励我们把自己的代码写得更好。

虽然我们苦口婆心地劝说让大家遵守编码风格，但又有多少人能认真地遵守呢？又有多少人能认真地去实践呢？

6.1 注重质量

代码设计是非常复杂的，它的成功不但依赖于架构和项目管理，也与代码质量紧密相关。代码质量与其整洁度成正比，整洁的代码在质量上较为可靠，也为后期维护奠定了良好的基础。

在梁肇新的《编程高手箴言》里：“以前所有的 C 语言书中，不太注重格式的问题，写出的程序像一堆垃圾一样。这也导致很多程序员的程序中有很多是废码、垃圾代码，这和那些入门的书非常有关系。因为这些书从不强调代码规范，而真正的商业程序绝对是规范的。你写的程序和他写的程序应该是格式大致相同，否则谁也看不懂。如果写出来的代码大家都看不懂，那绝对是垃圾。如果把那些垃圾‘翻’了半天，勉强才能把里面的‘金子’找出来，那这样的程序不如不要，还不如重新写过，这样思路还会更清楚一点。这是入门首先要注意的事情，即规范的格式是入门的基础。”那么垃圾代码具体体现在什么地方呢？

- 1) 没有好的书写风格，没有好的命名习惯，没有清晰的层次结构，没有好的文件和目录结构，繁琐啰嗦，在可读性和可维护性上没有任何保证；
- 2) 认为代码能正确运行就可以了，就万事大吉了，即使是对于非常糟糕的代码，他们也是自我感觉良好，他们不会在代码上精益求精；
- 3) 编写代码，为了修正错误或为了支持新功能，打了一个又一个补丁，最后成了一件“百衲衣”；
- 4) 编写的代码虽然能够正确地运行，但是他们的代码就像易碎的瓷器一样，重构性差，稍微改动就会导致代码运行不正常。

所有这些都是不负责任的表现，都是懒惰，最后的代码只能是偏离正道、乱七八糟、惨不忍睹、死气沉沉、难以阅读、难以维护，随着混乱的增加，团队生产力持续下降，最后趋于零，整个开发团队深陷沼泽，难以自拔，最后只能像扔垃圾一样扔掉这些垃圾代码。我们是不是很鄙视那些在公共场所乱丢垃圾的人？那么我们为什么能容忍这些给项目带来垃圾代码的人呢？

让代码工作和让代码整洁是两种不同的工作态度。只要代码“能工作”就转移到下一个任务上，然后那个“能工作”的代码就留在了那个所谓的“能工作”的状态，这其实是一种自毁行为。而让代码整洁则是对设计的精益求精，是对代码质量的认真负责。

代码质量要通过一条条的规则来保证，只有在良好的编码规则保证下，不断地改进优化自己的设计，提高工作效率和工作质量，才能编写出优质的代码。

6.2 遵守规则

每个程序员都喜欢按照自己的习惯编写代码，与自己风格相近的代码阅读和理解起来就容易一些；相反，与自己风格相差较大的代码阅读和理解起来就困难一些。另外，那些编码风格随意的代码，通常晦涩凌乱，在工作上既会给开发者本人带来很大的麻烦，也会给合作者和维护者带来很大的麻烦。

如果没有统一的编码规则约束，设计出来的代码就会充斥着各种不同的风格，显得混乱而邋遢，对代码质量产生巨大影响，代码的可读性、可维护性、可移植性会大大地降低。

但是强调遵守编码规则，经常是一个不太受欢迎的话题，因为某些人不能接受别人说他的编码风格不好，不能接受让他采用更好的编码风格，这仿佛触到了他的痛点和自尊一样。不管如何，遵守编码风格是非常有必要的，必须要推行下去。

对于一个开发部门，应该制定出一套大家都认同的编码规则，包括命名约定、格式约定、注释约定、结构约定等。大家都要遵守这些编码规则，而不是成为一个摆设放在那里。只有这样才能编写出整洁的代码，从而达到这样的效果：名字定义清晰，书写格式规范，逻辑组织有序，层次结构清晰。这样的代码用一个字形容，就是“爽”。

德国人非常注重规则和纪律，干什么都十分认真。凡是有明文规定的，德国人都会自觉遵守；凡是明确禁止的，德国人绝不会去碰它。在很多人的眼中，在很多情况下，德国人近乎呆板，缺乏灵活性，甚至有点儿不通人情。但细细想来，这种“不灵活”甚为有益，没有纪律，何来秩序？没有规矩，哪有认真？

德国人也很讲究清洁和整齐，不仅注意保持自己生活的小环境的清洁和整齐，也十分重视大环境的清洁和整齐。在德国，无论是公园、街道，还是影剧院或者其他公共场合，到处都收拾得干干净净，整整齐齐。所以德国人能研究出高精尖的技术，能制造出精密的仪器，是不是与他们的“守纪律，讲整洁”有极大关系呢？

6.3 简洁编程

编写代码有一个“KISS”原则，就是“Keep It Simple and Stupid”，简单来说，就是要把一个系统做得连白痴都会用。这就是编写代码的高层境界，好听的说法也是有的——“简单就是美”。

例如，对于机械工程师来说，减少零件数量、简化产品设计能够大幅度地减少工作量，一个零件在其开发周期中的任务包括零件设计、生成工程图、样品制作、零件试产、零件装配、零件质量和功能验证等，无一不是繁重的任务。所以减少零件数量、简化产品设计对于工程师来说是看得见的实惠，能够让工程师把更多的时间和精力放在提高产品设计质量上。

同样，编写代码也不是一个轻描淡写的过程。在做任何一个设计时，你必须考虑很多因

素，所有设计应当尽可能的简洁，这样产生的系统才是容易理解和容易维护的。但是简洁并不意味着要抛弃有意义的特性，也不是意味着“Quick and Dirty”，简洁意味着更加优雅的设计。所以简洁是要通过许多思考和多次修改才能达到的，而这些努力的回报就是代码错误更少，更容易维护。

如同代数中的因式分解一样，把一个多项式化为几个最简整式的积。编写代码要时时刻刻注重简洁，通过使用合并和分解（合并就是抽取公共代码做成函数，分解就是把大的函数切分成小的函数），消除冗余重复，优化代码结构，获得更加简洁的代码。

CPU 的设计曾经存在着复杂和简洁之争，也就是这两个阵营：CISC 和 RISC。

CISC（复杂指令集）：CPU 指令长且复杂，需要分成多个微指令去执行，因为指令众多，所以开发程序比较容易。但是由于指令复杂，导致执行工作效率较差，处理数据速度较慢。例如 Intel 的 Pentium 系列就是 CISC。

RISC（精简指令集）：CPU 指令短且简单，内部有快速处理指令的电路，使得指令译码与数据处理较快，所以执行效率比 CISC 高，但是程序必须要经过编译器的优化，才能发挥它的最大效率。例如 Cortex 系列、Power PC、MIPS 都是 RISC。

现在的 CPU 设计都是遵循 RISC 设计理念的，因为 RISC 的设计更加简洁。

著名的 Verilog 专家 Cliff Cummings 说：“我是一个简洁编码的狂热分子。”一般来说，代码越短越好。如果我们能够把一段代码整齐清晰地写在一页纸内，那么就很容易理解这段代码的意图。

按照我的观点，那些额外的要消耗掉一二代码行的“begin-end”和近来流行的那种“//end-always”的标示风格都毫无必要地增加了混乱，因为你在发现并识别出重要的细节之前，你的眼睛总是要扫描这些无用的代码。

我把这个现象称为“Where's Waldo”，这个名字来源于同名的儿童玩具书。虽然 Waldo 穿着鲜亮的红白条 T 恤，但是当他处于混乱的环境里，要想发现他就很困难。正如在混乱的环境里难以发现 Waldo 一样，当 RTL 代码格式混乱、注释愚蠢时，那么简单的编码错误都有可能被掩盖掉。

我们还可以从“断舍离”中借鉴一些理念，“断舍离”是山下英子推行的修行哲学“断行、舍行、离行”，通过对日常家居环境的收拾整理，改变意识，脱离物欲和执念，过上自由舒适的生活。断——断绝不需要的东西，舍——舍弃多余的废物，离——脱离对物品的执着。

“断舍离”推行的是一种非常简洁的生活方式，从“断舍离”开始，建立更深刻的洞察、更高远的观点、更宽广的视野，培养“俯瞰力”和“扫除力”，成为具备决断力与行动力的快乐人！

6.4 整洁编程

你设计的代码不是在真空中使用的，其他人也要使用、维护你的代码，这依赖于对你代码的理解。所以，你设计的代码应当能够让别人理解，你设计的代码应该有很好的质量。

代码的质量体现在每一个细节里，包括每一个变量的名字，每一行代码的书写，每一个函数的构造，有一处不整洁就如同“一粒老鼠屎坏了一锅粥”一样弄脏你的代码。下面就是

代码整洁的原则。

- 1) 好的代码需要在一定的原则、模式和实践下保证。好的代码不是一天练成的，需要非常用功，阅读大量的代码，需要仔细琢磨那些好的代码。
- 2) 在遵循一定的原则和模式，得到代码整洁的感觉，这种代码整洁感使得设计人员制订修改计划、按图索骥、重构代码。
- 3) 整洁的代码只做一件事，意图清晰、干净利落、直截了当、力求集中，全神贯注。糟糕的代码想做太多事，意图混乱，目的含糊。
- 4) 整洁代码是以增量式方法开发出来的，这样可以精炼并结构化代码。代码重构是要在完整的测试下保证的。如果没有得到完整的测试，很难做到重构代码。
- 5) 代码可以被读懂，不是因为其中的注释，而应该是由于代码本身的优雅：变量名意义清楚，空行和空格使用得当，逻辑分块清晰，表达式简洁明了。代码能够自我解释，而不用依赖太多的注释。
- 6) 使用注释描述设计意图，但是注释不能代替优秀的代码。
- 7) 代码简单，便于阅读。但要想达到代码简单，你所做的并不简单，简单并不意味着简陋、业余和不足，简单意味着你的技术精华。
- 8) 不要说“稍后我再调整代码”，因为有个原则，“稍后等于永不（Later equals never）”。要随时随地调整代码，让代码始终处于整洁状态，最后达到美好的设计。

我们应该从一开始就编写易读易维护的代码，提高表达力，整理编码格式，整理数据结构，提取公共函数，清除陈腐无用的代码，最终写出优雅高效的代码。

“流水不腐，户枢不蠹”，厨房脏了就擦一下，总比满墙都是油污以后再去清理的代价小得多。有价值的东西，比如回顾、重构、测试，一切有利于团队建设、提高生产力的实践都应该频繁且持续去做，然后日积月累就养成了习惯。

6.5 快乐编程

有个搞自驾游的司机师傅曾经跟笔者说过，“只有人车一体，爱护车，车子才会给你好的回报，你才会驾驭自如”，笔者从他的话语中感受到了驾驶汽车的快乐。因为他们对车的热爱，他们感受到了快乐，所以这些爱车的人会定期洗车、保养、修理、更换配件，会学习知识、遵守交规。

对于热爱编码的人来说，编写代码是一场精妙绝伦、富有挑战、迸发激情、充满享受、精益求精的活动，他们沉浸于编写代码中的乐趣中，在代码中寻求简单、和谐、完美、极致，下面就是优秀程序员所具有的品质。

- 1) 他们拥有强烈的好奇心，对所用语言、算法、数据结构精深地掌握。
- 2) 他们拥有团队协作精神，善于交流，更好地把自己的观点表达出来。
- 3) 他们拥有规范化的代码编写习惯，遵守编码规则。
- 4) 他们仔细研究问题，三思而后行，把握程序设计的每个细节。
- 5) 他们不害怕程序出现 Bug，严格定位并处理掉 Bug。

优秀的程序员都是“懒虫”加“处女座”，为了便于阅读理解，为了便于维护升级，为了系统完美运行，对代码使用优美的格式，做持续不断的重构，寻找最优的结构，同时完善各种设计文档，这就是“懒虫程序员”所做的，因为他们知道，他们做了这些事情之后，他们

的工作量会大大减轻。

这是程序员的最高境界：庖丁为文惠君解牛，手之所触，肩之所倚，足之所履，膝之所踣，砉（huā）然响然，奏刀騞（huō）然，莫不中音。合于桑林之舞，乃中经首之会……提刀而立，为之四顾，为之踌躇满志，善刀而藏之。

6.6 团队协作

你的代码不只是代表你自己，也代表着项目组、部门以至于公司的颜面。你要设身处地想一想，你的代码是否可读，是否可维护，是否给你的同事造成了困扰。代码设计是程序员之间彼此协调的工作，“你不是一个人在战斗”，所以要有“与人方便，与己方便”、“人人为我，我为人人”的精神，就是要有团队协作的精神。

团队协作是指在团队的基础上，发挥团队精神、互帮互助以达到团队最大工作效率。对于团队的成员来说，不仅要有个人能力，更需要有在不同位置上各尽所能、与其他成员协调合作的能力。团队强调的是协同工作，所以团队的工作气氛很重要，它直接影响团队的合作能力。团队内部要有互相帮助、取长补短、共同承担的精神，这样才能造就出一个好的团队。

6.7 测试驱动

测试驱动开发（Test-Driven Development, TDD），是一种不同于传统开发流程的新型的开发方法。它要求在编写某个功能的代码之前先编写测试代码，然后再编写能让测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简洁可用和高质量的代码，有很高的灵活性和健壮性，能够快速响应变化，并加速开发过程。

例如盖房子的时候，工人师傅砌墙，会先用桩子拉上线，以使砖能够砌得笔直，因为砌砖的时候都是以这根线为基准的。TDD 就像这样，先写测试代码，就像工人师傅先用桩子拉上线，然后编码的时候以此为基准，只编写符合这个测试的功能代码。

如果工人师傅砌墙时不拉线，而是直接把砖往上砌，砌了一些之后再去看是否笔直，这时候可能会用一根线，量一下砌好的墙是否笔直，如果不直再进行校正，敲敲打打。使用传统的软件开发过程就像这样，先编写代码，编码完成之后才写测试程序，以此检验已写的代码是否正确，如果有错误再修改。

测试驱动开发不是一种测试技术，它是一种分析技术、设计技术，更是一种组织所有开发活动的技术。相对于传统的结构化开发方法，它具有很多优势，可以及时向用户反馈，提高代码的质量，提高开发效率。

6.8 考虑全局

在做任何动作之前，首先做一个清晰、完整的考虑，这样才能产生更好的结果。如果你考虑了，但还是产生错误的结果，那么这种努力也是值得的。

首先，一个系统存在的理由就是：为用户提供价值。你所有的决定都取决于这一点。在指定一个需求、写下一段功能、决定系统平台和开发过程之前，问自己一个问题，“这样做会为系统增加价值吗？”如果答案是“Yes”，就做；如果答案是“No”，就不做。这个原则是最

基本的原则。

其次，清晰的远见是一个系统设计成功的基础。没有清晰的远见，项目开发到最后就变成天天为一个不好的设计做补丁。Booch 说：“只有当你对系统的体系有一个清晰的感觉，你才可能去发现通用的抽象和机制”。这种通用性还会让系统变得更简洁、更可靠。如果你不断地复制、粘贴、修改代码，那么你就不会对系统有一个清晰的认识，而且你将陷入到一个大泥潭中。

再次，成功的系统有很长的生命期，要对未来开放，要保证可扩充性，要在开始就做到最好。你必须让系统能够适应这样和那样的变化。所以，开始就不要把系统设计限定死，总是要问一下自己“如果这样会怎么样”，要考虑到各种各样的可能性，而不能光图省事。

但是，也要知道适可而止的原则，只需要将应用程序必需的功能包含进来，而不要试图添加那些你认为可能都需要的功能。在一个系统中，往往 80%的时间花费在 20%的功能上，所以不能无限地扩充系统的要求和功能。

“站得高，看得远”，只有拥有了全局的观念，才能处理解决现实中遇到的各种各样的问题，才能创造一个完美的系统。

6.9 代码切分

我们要通过分类、归纳和总结的方法，按照高内聚低耦合的原则，把代码划分为模块，形成合理的分层和分块结构。

一般来说，我们应该让每一个模块只做一个功能，隐藏内部实现细节，提供一个干净的接口，这就是高内聚的原则。另一方面，低耦合的原则就是模块与模块之间保持独立的存在，尽量让每个模块独立完成某个特定的子功能，让每个模块的接口尽量简单。

我们要注意提取公共代码或常用代码，形成独立模块，便于使用和移植。如果有可能，就要将它们通用化，这样就可以更灵活地使用它们。

在模块内部，我们也要合理地切分逻辑，让相关的代码紧挨着，组合在一起形成一个个不同的代码块，按照合理的顺序安排这些代码，并针对每个代码块添加注释。这样我们就可以按照每个代码块阅读和理解，就能够更好地阅读和理解整个模块。

鲁迅先生《秋夜》中有这样的语句，“我家门前有两棵树，一棵是枣树，另一棵也是枣树”，读起来似乎很美，还有很多联想的空间，可是想一想如果有 100 棵枣树，鲁迅先生会怎么写呢？难道要“一棵，一棵，又一棵”地写下去吗？写代码也一样，写几行重复的代码可以，可是不应该写数十到数百行重复的代码。

模块内不要存在重复的代码，因为重复的代码会造成代码混乱、维护困难和修改遗漏。当消除重复的代码之后，模块就变得清晰易读，更有表达力。重复的代码总是有共性，如果多次出现相同或相似的表达式/代码块，那么就应该设法把它们抽取并独立出来。

代码的结构并非一开始就是合理的，只有经过不断地调整，才能达到好的结果，但是很多人却存在着“一蹴而就”的想法。

6.10 代码重构

代码重构（Code Refactoring）就是在不改变系统外部行为的前提下，通过调整代码改善代码的内部结构，提高代码的质量和性能，提高系统的可维护性和可扩展性。

有人会问，为什么不在项目开始时多花些时间把设计做好，而要在以后花时间做代码重构呢？要知道一个完美得可以预见未来任何变化的设计，或一个灵活得可以容纳任何扩展的设计是不存在的。系统设计人员对即将着手的项目往往只能从大方向予以把控，而无法知道每个细枝末节。其次永远不变的就是变化，用户往往要在系统成型后，才开始“品头论足”，才能提出新的需求，设计人员毕竟不是先知先觉的神仙，功能的变化导致设计的调整在所难免。所以“测试为先，持续重构”作为良好开发习惯被越来越多的人所采纳，测试和重构就像江河湖海的护堤一样，成为保证代码质量的法宝。

代码重构和测试驱动开发相结合，可以提供一种精益、迭代和训练有素的编码风格，能够在很大程度上提高编码效率。通过不断地代码重构，逐步地提升代码质量，这样才能让系统更好地运行，延长系统的生命周期，拓展系统的应用范围。

6.11 深入学习

程序员的读书历程是这样的：X 语言入门 → X 语言应用实践 → X 语言高阶编程 → X 语言的科学与艺术 → 编程之美 → 编程之道 → 编程之禅 → 颈椎病康复指南。

在编写代码的人中，虽然很多人不是学习计算机软件专业的，他们写的代码也很好，但是他们是不是还欠缺一些东西呢？是不是也要学习一下《结构化程序设计》、《面向对象编程》、《软件工程》、《代码整洁之道》、《代码大全》、《敏捷编程》、《数据结构》、《设计模式》等书呢？反过来，即使计算机软件专业毕业的工程师，他们又是否掌握了这些书的精髓呢？

《C 语言程序设计》是一门普遍要学习的课程，有些人还在继续使用着 C 语言，那么请问现在的水平相比于大学毕业时的水平提高了多少呢？大学学的那些是很基本的东西，浅显易懂，够用就行。如果你在用 GCC 编程，那么你研究过《GNU C Compiler 手册》吗？学习过 C 语言的最新标准 C99 吗？如果你还在用 C 语言，而且还要深入下去，推荐大家有空学习一下《程序员的自我修养》。

每种语言都有基本的东西，也有高深的东西，需要深入研究语言的特性，避免语言的缺陷和陷阱，清楚了语言的要素之后，灵活组合运用它们，才能编写出优美的代码！说到缺陷和陷阱，C 语言有很多缺陷和陷阱，尤其是在操作符和指针上；Verilog 也有一些缺陷和陷阱，也是在操作符上，还有就是阻塞赋值和非阻塞赋值上。当你遭遇一些说不清的问题的时候，其实就有可能是这些缺陷和陷阱在做怪。

还有众多的设计方法学，例如面向对象就是在传播一种编程理念，把数据和方法封装为对象，不把数据和方法割裂开来，更自然地看待问题，而设计模式，与面向对象语言相结合，总结出一些设计原则，可以更方便地进行系统设计。

6.12 寻求诗意

优秀的代码阅读起来是一种享受，像听一首优美的歌，沉浸其中是一个充满快乐的过程，而不是令人烦恼的过程，我们会为代码中的精妙之处赞叹不已。

我们以《梨花又开放》为例，首先歌词充满了真情和诗意，歌词分为三部分，回顾过去、回到现在、重复前两部分，歌词用词工整押韵、朗朗上口、结构对称。

原唱：周峰。作词：丁小齐。作曲：因幡晃（日）

忘不了故乡 年年梨花放
染白了山冈 我的小村庄
妈妈坐在梨树下 纺车嗡嗡响
我爬上梨树枝 闻那梨花香
摇摇洁白的树枝 花雨漫天飞扬
落在妈妈头上 飘在纺车上
给我幸福的故乡 永生难忘
永生永世我不能忘

重返了故乡 梨花又开放
找到了我的梦 我一腔衷肠
小村一切都依然 树下空荡荡
开满梨花的树下 纺车不再响
摇摇洁白的树枝 花雨漫天飞扬
两行滚滚泪水 流在树下
给我血肉的故乡 永生难忘
永生永世我不能忘

摇摇洁白的树枝 花雨漫天飞扬
落在妈妈头上 飘在纺车上
给我幸福的故乡 永生难忘
永生永世我不能忘
摇摇洁白的树枝 花雨漫天飞扬
两行滚滚泪水 流在树下
给我血肉的故乡 永生难忘
永生永世我不能忘 永不能忘

优秀的代码也是如此，要经过设计者的仔细雕琢和精心打磨，要充满设计者的真情实意和奇思妙想，只有这样，代码才会使用、保留、传承、维护下去。

6.13 程序员节

程序员节是一个国际上被众多科技公司和软件企业承认的业内人士节日，日期是在每年的第 256（十六进制为 0x100）天，也就是平年的 9 月 13 日或闰年的 9 月 12 日。因为 256 是程序员都熟知的 8 位数，所以选取每年的第 256 天作为国际程序员节。它是俄罗斯的一个官方节日，很多国家也庆祝这个节日，包括中国、巴西、德国、加拿大、法国、英国和美国等。

每逢这一天，程序员们会穿着白色服装来庆祝，因为白色被选为程序员节的主题色。在红绿蓝 24 位深（RGB）颜色空间里，每种原色可以有 256 种级别（0~255 共 256 个数值）的深浅变化，当三种原色都达到最大深浅值，即为十六进制的 0xFFFFFFFF 时，表示白色。

程序员过了 5 次程序员节后，就会进化为“程序猿”，因为有的程序员已经完全变成了秃顶，已经回归到了原始类人猿。

编程格式与风格

我们有时候不愿意阅读别人的代码，就是因为他们的编码风格很差劲，读起来感觉很不舒服。

本部分讨论 Emacs 的使用和编码风格，编码风格包含格式优美、注释合理、名字定义，这些方面对于设计人员是非常重要的。

每个团队都应该基于行业规范制定出一套编码标准，设计人员都必须遵守这套标准，而不是把标准放在那里成了摆设。

使用 Emacs

“工欲善其事，必先利其器”，选择一个好的编辑器对于编写代码会有很大的帮助，一是使用上方便快捷，二是能够很好地控制编辑模式。

7.1 Emacs 介绍

Emacs 是一个功能强大的文本编辑器，在程序员和其他以技术工作为主的计算机用户中广受欢迎。Emacs 是 Editor MACroS（编辑器宏）的缩写，是由著名的自由软件之父 Richard Stallman 设计的。

Emacs 对于不同类型的文本进入不同的编辑模式，即“主模式”（major mode）。Emacs 对不同类型的文本定义了不同的主模式，包括普通文本文件、各种编程语言的源文件、HTML 文档等。

每种主模式都有特殊的 Emacs Lisp 变量和函数，在这种模式下用户能更方便地处理这一特定类型的文本。例如，各种编程的主模式会对源文件文本中的关键字、注释以不同的字体和颜色加以语法高亮。主模式还可以提供诸如跳转到函数的开头或者结尾这样的命令。

Emacs 还能进一步定义“次模式”（minor mode）。每一个缓冲区（buffer）只能关联一个主模式，却能同时关联多个次模式。例如，C 语言的主模式可以同时定义多个次模式，每个次模式有着不同的缩进风格（indent style）。

Emacs 提供了这些我们常用的特性。

- 1) 支持众多语言的对齐方式，而且可以自动完成对齐，例如 Verilog、C、ASM、Shell-Script、Perl 和 TCL 等。
- 2) 快捷的编辑方式，支持众多的快捷键，不用鼠标就可以完成所有的操作。如果使用 cua-mode，就可以使用 Word 软件中常用的快捷键。
- 3) 支持常规替换和正则替换，尤其是正则替换，你不用编写脚本就可以完成正则替换。
- 4) 对关键字、变量、宏定义和注释等使用不同颜色的显示，看起来更清晰、更方便。
- 5) 支持多 buffer 编辑。
- 6) 启动 Shell，在 Shell 里面输入命令，上下左右、翻页、拷贝粘贴，使用起来很灵活。

Emacs 是一个大宝藏，如果你能把它的基本命令和几种常用编程语言的模式用好，就非常不错。作者在没用 Emacs 之前，还要用手工方式去对齐 C 程序，很费事。后来作者改用 Emacs，刚开始有点不习惯，但是用了一段时间之后发现太好了，真是爱不释手。

7.2 Emacs 安装

假如你在用 Linux, OK, 很简单, 你只要在安装时选择安装 Emacs 即可, 然后把笔者的“.emacs”拷贝到你的根目录里。

如果你在用 Windows, 那么安装与配置要分以下 6 步。

- 1) 下载: 从 <http://ftp.gnu.org/pub/gnu/emacs/windows/> 下载 Emacs 的 Windows 版本, 可以选择 Emacs-24.1-bin-i386.zip。
- 2) 解压: 在 C 盘根目录下新建一个文件夹, 取名 Emacs-24.1 (也可以是其他路径), 把 emacs-24.1-bin-i386.zip 里的文件解压到这个目录下, 这样在 C:/Emacs-24.1/下就有 bin、etc、info、leim、lisp 和 site-lisp 等目录。
- 3) 安装: 双击 bin 文件夹里的 addpm.exe 进行安装, 安装后将在开始菜单生成 Gnu Emacs/Emacs 链接, 单击这个链接便可启动 Emacs。你也可以双击 bin 文件夹里的 runemacs.exe 启动。
- 4) 修改注册表: 用 regedit 打开注册表, 找到 HKEY_LOCAL_MACHINE/SOFTWARE/GNU/Emacs (如果没有, 就手动添加此项), 在此项下添加字符串值, 名称为 HOME, 值为 C:/Emacs-24.1。这样做的目的是让 C:/Emacs-24.1 成为 Emacs 的 HOME 路径。
- 5) 复制笔者的“.emacs”到 C:/Emacs-24.1 或者指定的 Home Directory (笔者机器上的目录是 C:/Users/asus/AppData/Roaming)。你可以按照你自己的要求和习惯修改“.emacs”。
- 6) 执行: 单击开始菜单中的 Gnu Emacs/Emacs。

你可以进一步到网上查找 Emacs 学习教程。你也可以在 Windows 上安装 Cygwin, 里面有 Emacs 和 XEmacs, 只不过这里的 Emacs 是字符接口的, 而 XEmacs 又需要你自己学习如何配置。

7.3 常用快捷键

掌握 Emacs 的快捷键可以说是 Emacs 爱好者的基本功, 也是提高编辑速度和质量所必备的, 但是初学者可能记不住这么多的快捷键, 所以必要时可以查看一下手册, 最常用的快捷键数量也就数十个。实在记不住快捷键, 就用鼠标点菜单, 选择要做的操作, 但不如使用快捷键那么方便。

Emacs 的快捷键都是绑定在 Ctrl、Shift 和 Meta (使用 Alt 或 Esc 键) 上的。例如, C-x 就是 Ctrl+x, S-x 就是 Shift+x, M-x 就是 Alt+x 或 ESC+x (注意: ESC 和 x 不能同时按)。例如, 当要退出 Emacs 时, C-x C-c 表示先按 C-x 再按 C-c, 就可以退出 Emacs。

下面是常用的快捷键, 结合下节介绍的 cua-mode, 可以使用键盘快速地编辑文件和运行命令。如果实在记不住这些命令, 那么没关系, 慢慢来。

- C-x C-f 打开文件, 出现提示时输入文件名;
- C-x C-v 打开一个文件, 取代当前 buffer;
- C-x C-s 保存文件;
- C-x C-w 另存为新文件;
- C-x i 插入文件;

- C-x d 打开目录，出现提示时输入目录名；
- C-x C-c 退出 Emacs；

- C-x 1 返回到只有一个窗口状态；
- C-x 2 把当前窗口拆分为上下两个窗口；
- C-x 3 把当前窗口拆分为左右两个窗口；
- C-x k 关闭当前窗口；
- C-x b 切换窗口，出现提示时输入 buffer 名；
- C-x C-b 显示所有的窗口列表；
- C-x C-c 退出 Emacs。

M-x <command>用于执行 Emacs 的命令，可以使用 Tab 完成命令，或提示有几条命令备选。如果你想执行某个不常用的命令，而且此命令没有绑定到快捷键上，那么就使用它来执行。例如，

- M-x info 打开 Emacs 手册，进一步学习 Emacs 的使用；
- M-x find-file 打开文件，等同于 C-x C-f；
- M-x upcase-region 把选定的 region 或后续区域做大写转换；
- M-x downcase-region 把选定的 region 或后续区域做小写转换；
- M-x doctor 你可以和一个心理医生对话。

7.4 笔者的“.emacs”

这是笔者的“.emacs”，你可以复制到你的 HOME 目录下面。你可以进一步修改它，选择自己喜欢的颜色，定义自己的常用命令快捷键。如果你不喜欢用 cua-mode，那么就把 cua-mode 的几行用“;”注释掉。

```
;;If you use Windows, then use the below line, otherwise use;; to comment it.
;(load-file "c:/emacs-24.1/lisp/emulation/cua-base.el")
(load-file "~/lisp/emulation/cua-base.el")
;;If you use Linux, then use the below line, otherwise use;; to comment it.
;(load-file "cua-mode.el")
(cua-mode t)

(global-set-key [C-home] 'beginning-of-buffer)
(global-set-key [C-end] 'end-of-buffer)
(global-set-key [home] 'beginning-of-line)
(global-set-key [end] 'end-of-line)

(define-key global-map [f3] 'isearch-forward)
(define-key isearch-mode-map [f3] 'isearch-repeat-forward)
(define-key global-map [S-f3] 'isearch-backward)
(define-key isearch-mode-map [S-f3] 'isearch-repeat-backward)
(define-key global-map [f5] 'goto-line)

(define-key global-map [M-f6] 'kill-buffer)
(define-key global-map [S-f6] 'buffer-menu)
(define-key global-map [C-f6] 'switch-to-buffer)

(define-key global-map [f10] 'replace-string)
(define-key global-map [f11] 'replace-regexp)
```

```

(define-key global-map [C-f10] 'query-replace)
(define-key global-map [C-f11] 'query-replace-regexp)

(set-background-color "black")
(set-foreground-color "white")
(set-mouse-color "yellow")
(set-cursor-color "red")

(setq kill-whole-line t)
(global-font-lock-mode t)
(column-number-mode t)
(which-func-mode t)
(show-paren-mode t)
(setq transient-mark-mode t)
(setq require-final-newline t)

(custom-set-faces)
(put 'upcase-region 'disabled nil)
(put 'downcase-region 'disabled nil)

(custom-set-variables
 '(column-number-mode t)
 '(cua-mode t nil (cua-base))
 '(show-paren-mode t))

(defun my-c-mode-common-hook()
  (c-set-style "K&R")
  (setq indent-tabs-mode nil)
  (setq tab-width 4)
  (c-toggle-auto-hungry-state 1)
  (setq c-basic-offset 4)
  (setq c-macro-shrink-window-flag t)
  (setq c-macro-preprocessor "cpp")
  (setq c-macro-cppflags " ")
  (setq c-macro-prompt-flag t)
  (setq hs-minor-mode t)
  (setq abbrev-mode t)
)
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)

;;On Linux
;;You can run shell command in shell-1
;;(shell)
;;(rename-buffer shell-1)

;;On Windows
;;You can run the unix-like command in shell-1
;;(eshell)
;;(rename-buffer shell-1)

```

7.5 cua-mode

如果你不想去记忆 Emacs 的那么多快捷键,而且你想在编辑 Word 文档时和在编辑 Verilog 文件时使用同样快捷键,那么就使用 cua-mode 吧。

cua-mode 是 CUA key bindings (Motif/Windows/Mac GUI) 的模拟。在这个模式里,你可以用 **Shift+<movement>** 键组合来高亮和扩充一个区域 (region),然后用其他键操作这个区域。如果你已经习惯用 Word 软件写文档,那么你可能已经熟悉以下键组合。若不熟悉,那么就打

开一个文件试一试这些键组合。

Shift + Up, Shift + Down, Shift + Left, Shift + Right

Shift + Home, Shift + End, Shift + PgUp, Shift + PgDown

Shift + Ctrl + Home, Shift + Ctrl + End

当你选定一个区域之后，那么就可以用以下的键组合操作这个区域。

Ctrl + c, 复制选定区域到剪贴板。

Ctrl + x, 复制选定区域到剪贴板，并删除它。

Ctrl + v, 把剪贴板的内容粘贴到指定位置。

Ctrl + z, 取消以前的操作（注意：它不是针对区域的操作）。

你也可以对一个区域做常规替换或正则替换操作，选定一个区域后，按 F10 键（或执行 M-x replace-string）或 F11 键（或执行 M-x replace-regexp）。

你也可以对一个区域做大小写转换操作（upcase-region, downcase-region），选定一个区域后，执行 M-x upcase-region 或 M-x downcase-region。

7.6 shell buffer

根据你使用的系统是 Linux 或 Windows，你可以把“.emacs”文件的后几行注释去掉，这样就会有一个名叫“shell-1”的“buffer”，在这里你可以输入“shell”命令。注意：在 Windows 上 eshell 是用来模拟 Linux shell 界面的。

笔者非常喜欢在这里运行命令，而不是在笨拙的 xterm 里面运行。在 shell-1 buffer，可以有以下便利。

- 1) 输入命令之后查看运行输出的信息，你可以按方向键、PageUp、PageDown、Home、End、Ctrl+Home 和 Ctrl+End 等查看信息，可以在这里搜索字符串。你可以在这里复制，然后粘贴到其他地方。当你不需要这些输出信息时，你就选择全部，然后删除。
- 2) 你可以方便地修改命令，只要把光标移动到你要修改的命令位置，修改它，然后再运行。你还可以输入 history，找到你要的命令，去掉前面的数字，然后再运行。

第 8 章

快速排序

现代编程语言都有相似的元素和结构，当我们熟悉一种或数种语言的元素和结构之后，就会很容易地推广到其他语言上了。下面以快速排序为例，看这几种语言的实现，它们的元素和结构上是不是很相似呢？所以虽然本书主要是以 C 语言为例，但是很多原则都可以推广到其他语言上。

8.1 算法简介

快速排序由 Tony Hoare 在 1962 年提出，它的基本思想是：通过一趟排序把要排序的数据分割成独立的两部分，其中一部分的所有数据比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，从而让所有数据变成有序的序列。

设要排序的数组是 $A[0] \cdots A[N-1]$ ，首先任意选取一个数据（通常选用数组的第一个数）作为关键数据，然后将所有比它小的数都放到它前面，所有比它大的数都放到它后面，这个过程称为一趟快速排序。值得注意的是，快速排序不是稳定的排序算法，也就是说，多个具有相同值的数据的相对位置也许会在算法结束时产生变动。下面就是一趟快速排序的算法。

- 1) 设置两个变量 i 、 j ，排序开始的时候： $i=0$ ， $j=N-1$ 。
- 2) 以第一个数组元素作为关键数据，赋值给 key ，即 $key=A[0]$ 。
- 3) 从 j 开始向前搜索，即由后开始向前搜索 ($j--$)，找到第一个小于 key 的值 $A[j]$ ，将 $A[j]$ 和 $A[i]$ 互换。
- 4) 从 i 开始向后搜索，即由前开始向后搜索 ($i++$)，找到第一个大于 key 的 $A[i]$ ，将 $A[i]$ 和 $A[j]$ 互换。
- 5) 重复第 3、4 步，直到 $i=j$ 。

说明：当找到符合条件的值进行交换时，保持 i 和 j 指针位置不变。另外， $i=j$ 这一过程一定正好是 $i++$ 或 $j--$ 完成的时候，此时就让循环结束。

8.2 C/C++语言

```
void quick_sort (int ar[], int low, int high)
{
    if (low >= high)
        return;
    int first = low;
    int last = high;
    int key = ar[first]; //选择第一个数据作为 key
```

```

while (first < last)
{
    while (first < last && ar[last] >= key)
        last--;
    ar[first] = ar[last]; //将比 key 小的数据移到低端
    while (first < last && ar[first] <= key)
        first++;
    ar[last] = ar[first]; //将比 key 大的数据移到高端
}
ar[first] = key; //将 key 移动到确定的位置
quick_sort (ar, low, first - 1);
quick_sort (ar, first + 1, high);
}

```

8.3 Java

```

class Quick
{
    public void sort (int ar[], int low, int high)
    {
        if (low >= high)
            return;
        int first = low;
        int last = high;
        int key = ar[first];
        while (first < last)
        {
            while (first < last && ar[last] >= key)
                last--;
            ar[first] = ar[last];
            while (first < last && ar[first] <= key)
                first++;
            ar[last] = ar[first];
        }
        ar[first] = key;
        sort(ar, low, first - 1);
        sort(ar, first + 1, high);
    }
}

```

8.4 Perl

```

#!/usr/bin/perl
use strict;
use warnings;
sub quick_sort
{
    my $ar = $_[0];
    my $low = $_[1];
    my $high = $_[2];
    return if ($low >= $high);

    my $first = $low;
    my $last = $high;
    my $key = $$ar[$first];
    while ($first < $last)
    {
        while ($first < $last && $$ar[$last] >= $key)

```



```

        {
            $last--;
        }
        $$ar[$first] = $$ar[$last];
        while ($first < $last && $$ar[$first] <= $key)
        {
            $first++;
        }
        $$ar[$last] = $$ar[$first];
    }
    $$ar[$first] = $key;
    quick_sort ($ar, $low, $first - 1);
    quick_sort ($ar, $first + 1, $high);
}
my @num = ();
my $count = 50;
my $i;
for ($i = 0; $i < $count; $i++)
{
    $num[$i] = rand (10000);
}

printf ("Before sort\n");
for ($i = 0; $i < $count; $i++)
{
    printf ("%d %d\n", $i, $num[$i]);
}

quick_sort (\@num, 0, $count - 1);

printf ("\nAfter sort\n");
for ($i = 0; $i < $count; $i++)
{
    printf ("%d %d\n", $i, $num[$i]);
}

```

8.5 Verilog

```

`define TOTAL_NUMBER    40
module test;
    integer ar[0:`TOTAL_NUMBER-1];
    task automatic quick_sort;
        input [31:0] low;
        input [31:0] high;
        reg [31:0] first, last;
        integer key;
        begin: sort_block
            if (low < high) begin
                first = low;
                last = high;
                key = ar[first];
                while (first < last)
                    begin
                        while (first < last && ar[last] >= key)
                            last = last - 1;
                        ar[first] = ar[last];
                        while (first < last && ar[first] <= key)
                            first = first + 1;
                        ar[last] = ar[first];
                    end
            end
        end
    endtask
endmodule

```

```
        end
        ar[first] = key;
        quick_sort (low, first - 1);
        quick_sort (first + 1, high);
    end
endtask

integer i;
initial begin
    for (i = 0; i < `TOTAL_NUMBER; i = i + 1)
        ar[i] = $random;
    $display ("Before sort:");
    for (i = 0; i < `TOTAL_NUMBER; i = i + 1)
        $display ("%d %d", i, ar[i]);
    quick_sort (0, `TOTAL_NUMBER - 1);
    $display ("After sort:");
    for (i = 0; i < `TOTAL_NUMBER; i = i + 1)
        $display ("%d %d", i, ar[i]);
    end
endmodule
```

注意：我在使用 VCS 仿真时，TOTAL_NUMBER 不能定义得太大，可能 VCS 不能支持太深的递归。

我们编写代码的时间其实大部分用在了看代码上。当我们阅读代码时，格式混乱的代码会让我们感觉非常不快，即使代码中充满了奇思妙想；而格式优美的代码会让我们心情愉悦，我们看代码会更容易、更快捷。虽然编码格式不会影响执行速度、内存使用等方面的性能，但是对检查代码、理解代码、修改代码和编码效率很产生影响，会对成员之间的沟通与合作产生影响，所以编码格式非常重要，不可忽视，我们必须严肃对待。

我们在阅读一本好书时，不只是书的内容吸引人，书的格式和编排也很优美，看的时候也很方便，可以从头看到尾，也可以跳来跳去看。好的代码也是如此，不仅代码设计巧妙，而且格式也很优美，具有很好的对齐、留白以及顺序，可以为代码的逻辑结构提供有益的提示，让阅读代码变得更容易。好的代码有三条原则：使用一致的布局，让读者很快就习惯这种风格；让相似的代码看上去相似；把相关的代码行分组，形成代码块。

本章介绍的这些编码格式看着简单，但是应用之后，可以大幅度地提高可读性、可理解性和可维护性，从而可以为大规模的代码重构提供巨大的便利。我们要保持良好的编码格式，就要确定一套编码格式的规则，并且遵守这套规则。

9.1 合理安排

文件内的内容要按照一定的顺序合理安排，相关的内容要放在一起，不相关的内容用空行加以分割，这样程序看起来会更加清晰，不要像一锅粥似的混在一起。

C 语言头文件的内容最好按下面的顺序安排。

- 1) 开头的注释，包含版权信息、版本信息和总体介绍。
- 2) 使用 `#ifndef/define/endif`，用来防止头文件内容被重复包含。
- 3) 如果需要包含别的头文件，放在这里。
- 4) 宏定义—常量，例如 “`#define MAX_VPB 2048`”。
- 5) 宏定义—代码，例如 “`#define MAX_VAL(a,b) ((a) > (b) ? (a) : (b))`”。
- 6) 枚举类型定义，例如 “`enum ANIMAL_ENUM {DOG, HEN, PIG};`”。
- 7) 自定义数据类型声明，例如结构、联合和 `typedef`。
- 8) 全局变量声明，例如 “`extern int gHasInitFlag;`”。
- 9) 函数原型声明，例如 “`int get_max_data (int count, int ar[]);`”。

C 语言代码文件的内容最好按下面的顺序安排。

- 1) 开头的注释，包含版权信息、版本信息和总体介绍。
- 2) 包含头文件，需要包含所用的头文件，以避免出现缺少函数原型的问题。

- 3) 文件自己使用的常量、枚举、自定义数据类型，但最好把它们也放到头文件中。
- 4) 全局变量定义，如果需要，可以初始化它们。
- 5) 静态变量定义，如果只是文件内使用，可以把全局变量定义成静态的，以减少名字冲突。
- 6) 函数定义，函数按照调用关系排列，调用者放到被调用者的前面。如果函数是私有的，可以定义成静态函数，以减少名字冲突。

Verilog 的头文件一般只放宏定义常量，或者放公用的 `function` 和 `task`，不像 C 语言的头文件有那么多内容。Verilog 的代码文件一般是一个文件只有一个模块，也不像 C 语言的代码文件有那么多内容，关键在于每个模块可能会很大，所以要安排好端口声明和内部逻辑的分布。

9.2 横向缩进

现在的编程语言都是通过“{ ... }”或者“`begin ... end`”支持嵌套结构，为了更好地表示语句之间嵌套关系，通常都是使用语句的横向缩进表示这种嵌套关系，横向缩进可以大大提高代码的可读性。下面以 C/C++ 语言的 `if` 和 `for` 语句为例，常用如下缩进格式。

例子 1: 最紧凑格式，“else”与“}”在同一行

```
if (a > b) {
    //omit.....
} else if (a == b) {
    //omit.....
} else {
    //omit.....
}

for (i =0; i < 100; i++) {
    //omit.....
}
```

例子 2: 紧凑格式，“else”另起一行

```
if (a > b) {
    //omit.....
}
else if (a == b) {
    //omit.....
}
else {
    //omit.....
}

for (i =0; i < 100; i++) {
    //omit.....
}
```

例子 3: 非紧凑格式，“{”和“}”与“if/else”保持对齐

```
if (a > b)
{
    //omit.....
}
else if (a == b)
{
    //omit.....
}
```

```

else
{
    //omit.....
}

for (i =0; i < 100; i++)
{
    //omit.....
}

```

例子 4: 非紧凑格式, “{” 和 “}” 相比 “if/else” 向右缩进 2 个空格

```

if (a > b)
{
    //omit.....
}
else if (a == b)
{
    //omit.....
}
else
{
    //omit.....
}

for (i =0; i < 100; i++)
{
    //omit.....
}

```

对于 switch 语句, 常用两种格式: 行尾布局 and 标准缩进。标准缩进格式感觉更好, 因为修改和阅读更方便。

例子 1: case 语句采用行尾布局格式

```

switch (k & STRTOG_Retmask) {
case STRTOG_NoNumber:
case STRTOG_Zero: L[0] = L[1] = 0;
                    break;
case STRTOG_Denormal: L[_1] = bits[0];
                      L[_0] = bits[1];
                      break;
case STRTOG_Normal:
case STRTOG_NaNbits: L[_1] = bits[0];
                     L[_0] = (bits[1] & ~0x100000) | ((exp + 0x3ff + 52) << 20);
                     break;
case STRTOG_Infinite: L[_0] = 0x7ff00000;
                      L[_1] = 0;
                      break;
case STRTOG_NaN: L[_0] = 0x7fffffff;
                 L[_1] = (__ULong)-1;
                 break;
}

```

例子 2: case 语句采用标准缩进格式

```

switch (k & STRTOG_Retmask) {
case STRTOG_NoNumber:
case STRTOG_Zero:
    L[0] = L[1] = 0;
    break;
case STRTOG_Denormal:
    L[_1] = bits[0];

```

```

    L[_0] = bits[1];
    break;
case STRTOG_Normal:
case STRTOG_NaNbits:
    L[_1] = bits[0];
    L[_0] = (bits[1] & ~0x100000) | ((exp + 0x3ff + 52) << 20);
    break;
case STRTOG_Infinite:
    L[_0] = 0x7ff00000;
    L[_1] = 0;
    break;
case STRTOG_NaN:
    L[_0] = 0x7fffffff;
    L[_1] = (__ULong)-1;
}

```

例子：简单的 switch，可以采取更紧凑的方式，这样看着更清晰一些

```

switch (coin)
{
case 1: str = "Cent"; break;
case 5: str = "Nickel"; break;
case 10: str = "Dime"; break;
case 25: str = "Quarter"; break;
}

```

这个 switch 例子说明在使用格式缩进的时候，不要太拘泥于一种形式，不要太死脑筋，如果能获得更好的表现效果，代码小范围内的不一致也是可以接受的。

对于其他语句都有相应的缩进格式。这几种格式并没有好坏之分，依个人偏好选择。不管怎样，关键在于要固定使用一种格式，不要混合使用这几种格式，尤其是要让自己和团队保持一致的缩进格式。

例子：对于函数，一般都采用如下格式，用于函数的“{”和“}”都在单独一行

```

int calculate_summary (int count, int ar[])
{
    int sum = 0;
    for (i =0; i < 100; i++)
        sum += ar[i];
    return sum;
}

```

对于简单 if 和 for 语句，是否也要使用缩进格式呢？是不是写得更紧凑一些更好呢？这种紧凑为一行的语句确实简单，但是从代码清晰的角度看，还是使用两行的较好，而且在单步调试的时候，可以明确地看到语句是否真的执行了。

例子：语句使用两行

```

if (a > b)
    x = y;
for (i =0; i < 100; i++)
    sum += ar[i];

```

例子：语句使用一行

```

if (a > b) x = y;
for (i =0; i < 100; i++) sum += ar[i];

```

通常都用 Tab 来实现代码的缩进，这样可以减少几个空格的输入。在编辑器中，Tab 定义为几个空格是可以设置的，一般设置为 4 个或 8 个，最好设置为 8 个，因为 8 个更符合通常

使用。

有的编辑器很智能，例如 Emacs，按 Tab 键之后，自动实现当前代码行的缩进，很方便。如果选中一块区域，再按 Tab 键，就可以实现这块区域的缩进。

通常编辑器要把 Tab 保存到文件中，当你在一个编辑器中编辑好一个文件，然后换用另一个编辑器，如果两个编辑器中 Tab 的空格设置不一样，就会导致在原来的编辑器中对齐好的格式，在另一个编辑器中发生严重的错位。所以最好的办法是，按 Tab 键实现自动对齐，但是使用空格替换 Tab 键，使得文件中不存在 Tab 键，这样在任何编辑器中都不会发生格式错位的问题。Emacs 就有这样的设置。

如果你没有耐心做手工缩进，就不要使用简单编辑器，要使用 Emacs 这样的智能编辑器！另外，在 Unix/Linux 下有对 C 语言程序自动缩进的工具有 indent，可以支持多种不同的缩进格式。

编辑器中的缩进格式是可以设置的，例如上面的 if 和 for 语句的四种缩进格式，尤其是在 Emacs 这样灵活的编辑器中，这时就需要在公司/部门内使用统一的缩进格式，不能每人都都有自己的缩进格式，否则就会出现格式混乱的问题，所以要有一个统一的初始化设置文件“.emacs”。

当你使用编辑器时，要选用固定宽度的字体，这样可以更好地表现缩进关系，因为某些编辑器可以针对不同的元素（关键字、变量、常量、宏定义、函数等）设置不同的字体，这时代码看起来很不舒服。

上面探讨了 C/C++ 语言的缩进格式，Perl、Verilog 等现代编程语言都有类似的缩进格式，而且 Emacs 支持得都很好。

9.3 纵向对齐

纵向对齐（也就是上下对齐的列）可以让读者更轻松地浏览文本，同样你可以借用“纵向对齐”的方法来提高代码的可读性，这样可以更方便地检查代码。

有些人不喜欢用它，觉得建立和维护对齐的工作量太大，因为在只改动一行的内容时，可能需要改动所有行的对齐。但是，在实际的使用中，纵向对齐并不像有些人担心的那样费时工夫，因为需要纵向对齐的代码并不是太多，而且确实可以获得很好的可读性。

例子：纵向对齐—代码

```
#define bin_index(sz) \
(((unsigned long)(sz) >> 9) == 0) ? ((unsigned long)(sz) >> 3) : \
(((unsigned long)(sz) >> 9) <= 4) ? 56 + ((unsigned long)(sz) >> 6) : \
(((unsigned long)(sz) >> 9) <= 20) ? 91 + ((unsigned long)(sz) >> 9) : \
(((unsigned long)(sz) >> 9) <= 84) ? 110 + ((unsigned long)(sz) >> \
12) : \
(((unsigned long)(sz) >> 9) <= 340) ? 119 + ((unsigned long)(sz) >> \
15) : \
(((unsigned long)(sz) >> 9) <= 1364) ? 124 + ((unsigned long)(sz) >> \
18) : \
126)
```

例子：纵向对齐—注释

```
struct mallinfo {
    int arena; /* total space allocated from system */
    int ordblks; /* number of non-inuse chunks */
}
```

```

int smblks; /* unused -- always zero */
int hblks; /* number of mmapped regions */
int hblkhd; /* total space in mmapped regions */
int usmblks; /* unused -- always zero */
int fsmblks; /* unused -- always zero */
int uordblks; /* total allocated space */
int fordblks; /* total non-inuse space */
int keepcost; /* top-most, releasable (via malloc_trim) space */
};

```

对于 Verilog，在模块定义时纵向对齐端口，在实例化模块时纵向对齐端口和信号，会收到非常好的效果，会大大提高代码的可读性。

例子：纵向对齐—模块实例化

```

cgu                                cgu_i
(
  //Interface with APB bus
  .preset_n      (cgu_preset_n),
  .pclk          (cgu_pclk),
  .psel          (psel_cgu),
  .paddr        (paddr),
  .penable      (penable),
  .pwrite       (pwrite),
  .pwrdata      (pwrdata),
  .prdata       (prdata_cgu),
  //Interface with CCF
  .cgu_pdr      (cgu_pdr),
  .cgu_cnt      (cgu_cnt),
  .cgu_pcr      (cgu_pcr),
  .cgu_lpc      (cgu_lpc),
  .cgu_cst      (cgu_cst),
  .cgu_clr      (cgu_clr),
  .cgu_div      (cgu_div),
  .cgu_msr      (cgu_msr),
  .cgu_rcr      (cgu_rcr),
  .cgu_rst      (cgu_rst)
);

```

9.4 顺序书写

有些情况下，代码书写的顺序并不会影响代码执行的正确性，但是即使是这样也不要随便书写，把相关的代码按照有意义的顺序书写会有很大的好处。代码的书写顺序可以如下：按照先输入后输出的顺序，按照从重要到次要的顺序，或者按照字母顺序。

当对一块代码确定一种书写顺序之后，在代码中的各个位置，都要使用同样的顺序，这样既便于检查书写错误，又便于阅读代码。

例子：这些代码在一个文件中出现多次，都按同样的顺序书写

```

mParaOneZonePtr->vb2vpbPtr[currentOperPtr->VB].ver++;
mParaOneZonePtr->plktPtr[i].vpbArrayPtr[1] = j;
mParaOneZonePtr->plktPtr[i].vpbAmount++;
mParaOneZonePtr->plktPtr[i].emptyVPP = 0;
mParaOneZonePtr->plktPtr[i].seqPageAmount = 0;

```

函数定义时要定义一些形式参数，只要函数定义的形参和函数调用的实参对应上就可以，执行就会正确。但是这并不意味着我们可以随意安排形参的顺序，形参也要按照一定顺序排列，就是要按照形参的主次关系排列，这样可以便于阅读代码。另外，一般函数都是成组出

现，以便于按照不同的功能操作数据，这时就要在这组函数内保持形参顺序的一致性。

例子：常用 string 函数的参数顺序

```
char *strcpy (char *dst, const char *src);
char *strncpy(char *dst, const char *src, size_t n);
char *strcat (char *dst, const char *src);
char *strncat(char *dst, const char *src, size_t n);
int  strcmp (const char *s1, const char *s1);
int  strncmp(const char *s1, const char *s2, size_t n);
char *strchr (const char *s, int c);
char *strrchr(const char *s, int c);
size_t strlen (const char *s);
```

对于 Verilog 的 module, 当有数十个到数百个端口时, 这些端口就要根据功能划分出组来, 然后对每一组按照固定的顺序排列, 只有这样, 你才能轻松地管理这些端口。下面就是推荐的模块端口声明顺序: 异步复位、时钟信号、使能信号、控制信号、地址信号、响应信号、数据信号。

9.5 分组成块

我们会很自然地按照分块的方法来思考问题, 就如同把文章划分成章节和段落一样, 所以我们要按照代码行之间内在的逻辑关系, 把关系紧密的代码行放在一起, 形成一个个代码块, 并在代码块之间用空行加以分割, 这样就可以提高代码的可读性, 可以更方便地阅读代码, 可以一块一块地理解代码。

对于用 C/C++ 和 Perl 等软件语言编写的程序, 因为代码行是按先后顺序串行执行的, 相对来说, 很容易划分出代码块。对于用 Verilog 硬件语言编写的程序, 因为 always 和 assign 语句都是并行执行的, 把它们放到哪里都能正确执行, 即使是胡乱摆放它们, 执行也是正确的, 但是这样会给理解代码带来巨大的困难, 所以更要把它们划分出代码块。

划分代码块不只是适用于执行语句, 也适用于变量定义、函数声明、宏定义声明等, 还有 Verilog 中大量的端口声明和端口连接, 只要它们有内在的逻辑关系, 就要把它们放在一起。

对于多个连续的代码块, 为了方便地区分它们, 作者通常会在它们之间添加这样的一行注释, “//-----”, 因为在 Emacs 编辑器中, 注释和代码的颜色不同, 可以更方便地区分出每一个代码块。

当划分出代码块之后, 就可以在代码块的上方很方便地定义只用于此代码块的临时变量, 这样就可以缩小这些临时变量的作用域, 因为现在的编程语言都支持就近定义临时变量。另外, 当划分代码块之后, 就可以很方便地在代码块的上方添加注释, 说明它们的功能用途和运行机制。

例子：划分代码块, strcmp()

```
int strcmp (const char * s1,const char * s2)
{
    //If s1 or s2 are unaligned, then compare bytes.
    if (!UNALIGNED (s1, s2)) {
        //If s1 and s2 are word-aligned, compare them a word at a time.
        unsigned long *a1 = (unsigned long*)s1;
        unsigned long *a2 = (unsigned long*)s2;
        while (*a1 == *a2) {
            //To get here, *a1 == *a2, thus if we find a null in *a1,
            //then the strings must be equal, so return zero.
```

```

        if (DETECTNULL (*a1))
            return 0;

        a1++;
        a2++;
    }

    //A difference was detected in last few bytes of s1, so search bitwise
    s1 = (char*)a1;
    s2 = (char*)a2;
}

while (*s1 != '\0' && *s1 == *s2) {
    s1++;
    s2++;
}
return (*(unsigned char *) s1) - (*(unsigned char *) s2);
}

```

9.6 添加空白

横向缩进、纵向对齐要添加空格或 Tab，分组成块要添加空行，这些都是有意的，就如同中国画的留白有很好视觉效果和意境一样，空格和空行也会提高代码的视觉效果，可以更好地表达出代码的逻辑结构，从而提高代码的可读性。

基于同样的道理，如果代码看起来还是很拥挤，就可以在表达式和语句上适当地添加空格、空行和换行，让代码看起来更清晰，进一步提高代码的可读性。

- 1) 在代码块之间添加空行，可以让每个代码块清晰地表示出来；
- 2) 在操作符的左右添加空格，可以让每个表达式清晰地表示出来；
- 3) 在函数声明和函数调用时，函数名和括号之间加一个空格，参数后面不加空格，逗号后面加一个空格，如果函数有很多参数，那么就在合适的位置换行；
- 4) 其他不该添加空格、空行和换行的地方，就不要随便添加，否则就会带来不好的效果。

例子：因为使用空格和换行，所以看起来更清晰

```

x=a>b?a:b;
for(i=0;i<100;i++)
    sum+=ar[i];
printf("The results of a%d b%d are a+b%d a-b%d a*b%d
a/d%d\n",a,b,a+b,a-b,a*b,a/b);
//添加空格和换行后
x = a > b ? a : b;
for (i = 0; i < 100; i++)
    sum += ar[i];
printf ("The results of a%d b%d are a+b%d a-b%d a*b%d a/d%d\n",
        a, b, a + b, a - b, a * b, a / b);

```

例子：因为使用换行，所以看起来更清晰

```

if ((prev_vb != vb && prev2_vb != vb) || (prev_vb == vb && (prev_lpn + 1) !=
lpn) || (prev2_vb == vb && (prev2_lpn + 1) != lpn))
    FTL_Fill_Residual_VPP ((flag << 30) | lba, 1, vb);
//添加空格和换行后
if ((prev_vb != vb && prev2_vb != vb)
    || (prev_vb == vb && (prev_lpn + 1) != lpn)
    || (prev2_vb == vb && (prev2_lpn + 1) != lpn))
    FTL_Fill_Residual_VPP ((flag << 30) | lba, 1, vb);

```

有些人的代码确实有很多的空格和换行，可是毫无美感，因为他们不是在有意地寻求代码的表达力，只是很随意地添加了这些空格和换行，但是他们并没有感觉到有什么不妥的地方。

9.7 书写语句

代码是由语句和表达式构成的，本节说一下语句，下节说一下表达式。

在书写所用语言（C、Perl、Verilog 等）的每条语句时，要遵循如下约定。

- 1) 每一个语句都要独立成行，不要把多条语句放在一行，不要把几条很简单的语句放到一行里。因为独立成行更方便阅读、查找错误和单步调试。
- 2) 每一个或一组变量的声明也要独立成行，不要把多个类型相同但没有关联的变量挤到一行。因为独立成行便于对变量添加注释，也便于阅读修改。
- 3) 在同一个代码块之内不加空行，可以表明此代码块具有紧密的关系。
- 4) 在不同的代码块之间添加空行，可以表明每个代码块做不同的功能。
- 5) 保持每行不超过 80 个字符，因为行太长了看着不方便。
- 6) 对于 if、switch、for、while 等语句选择合适的缩进方式。
- 7) 不要使用太深的嵌套，因为阅读起来非常不方便。
- 8) 合理使用 Tab，一个 Tab 最好对应 4 个字符，另外文件内最好不要保存 Tab。

在书写 Verilog 的每条语句时，遵循如下约定可以达到更好的效果：

- 1) 对于 always、for、while 语句，begin 要在它们的下一行。
- 2) 对于 initial、if、else if、else 语句，begin 要与它们在同一行，而且要在 begin 后换行。
- 3) end 要占据单独一行，不要在同一行写 end else 或 end else begin 这种语句。

9.8 书写表达式

首先，要构造合适长度的表达式，不要构造冗长复杂的表达式，因为冗长复杂的表达式不容易理解，不容易“debug”，而且不容易维护。

其次，可以在表达式的合适位置添加括号，因为可以清晰地表示运算的优先级，同时避免错误理解运算符的优先级。但是不要添加过多的括号，因为括号多了，代码看起来也费劲，这个问题可以通过拆分表达式解决。

例子：下面两句，哪一个更简洁呢？不同的人会有不同的观点，笔者倾向于第二句。

```
if ((alpha < beta) && (gamma >= delta)) //verbose
if (alpha < beta && gamma >= delta) //concise
```

在表达式的合适位置添加空格和换行，阅读起来更清晰。

- 1) 对于双目操作符，例如=、+、-、*、/、%、<<、&、&&等，在它们的两边各加一个空格。
- 2) 对于单目操作符，例如++、--、&、*、+、-等，让它们和操作数紧贴在一起，不加空格。
- 3) 访问数组元素和结构成员时（使用[]、.、->），让它们和操作数紧贴在一起，不加空格。

- 4) 对于逗号 (,)，只在逗号后面加空格。
- 5) 对于分号 (;)，只在分号后面加空格。
- 6) 行尾不加空格。
- 7) 在某些位置可以不添加空格，这样看起来更紧凑一些。
- 8) 在某些位置可以多添加一些空格，使得上下几行的某些变量有对应，达到纵向对齐。
- 9) 当表达式很长时，就在适当的位置换行，并添加空格让某些变量有对应，达到纵向对齐。

例子：C 语言代码，合理地使用空格。

```
mustlast = g->must + g->mten - 1;
p->ssize = len / (size_t)2 * (size_t)3 + (size_t)1;
tail = slow (m, rest, stop, es, stopst);
while (*--dp == *--pp && pp != mustfirst);
for (ss = startst; !hard && ss < stopst; ss++) .....
```

例子：Verilog 代码，合理地使用空格。

```
wire [`XA_WIDTH*2-1:0] xa = (val_a * val_b + val_c);
wire [3:0] kpc_num = (cfg_sel == 0 ? 4'd0 :
                    cfg_sel == 1 ? 4'd1 :
                    cfg_sel == 2 ? 4'd3 :
                    cfg_sel == 3 ? 4'd7 :
                    cfg_sel == 4 ? 4'd15 : 4'd15);
```

9.9 Verilog 部分

我们在做 ASIC 或 FPGA 设计的时候，系统中会有大量的 Verilog 模块，模块端口声明和模块实例化的工作是非常繁琐的，好的书写格式不仅可以减轻一些工作，还可以提高代码的可读性。

9.9.1 模块端口声名

在声明模块的端口时，我们要遵循如下约定。

- 1) 要尽量使用 Verilog-2001 标准，这样不仅可以减少代码行，而且便于修改和删除。
- 2) 每行只声明一个端口，这样可以在其上或其后添加一些简短的注释。
- 3) 不要按字母排序，按字母排序的端口声明是最差劲的。
- 4) 不要按输入/输出分组。少量的信号还可以使用，大量的信号就非常差劲，修改和删除非常费劲，理解起来也费劲。
- 5) 要按功能分组，功能组前面可以加注释，功能组之间加空行分割，以便于阅读。
- 6) 在每个功能组内，哪一个最主控，哪一个就越靠前，因为这样符合思维习惯，所以采用以下顺序：异步复位、时钟信号、使能信号、控制信号、地址信号、响应信号、数据信号。
- 7) module 名字后面的括号要另起一行，放置在固定的列数上，而且括号后面不要加空格，这样后续的端口信号依此对齐。

9.9.2 模块实例化

在模块实例化时，我们要遵循如下约定。

- 1) 要使用按端口名字连接的方式，不要使用按端口位置连接的方式，这样可以提高代码的可读性和可修改性，而且便于检查连线是否正确。
- 2) 实例化端口的顺序要与模块端口声明的顺序保持一致，这样便于查找、修改和删除。对于不用的输出端口或 IO 端口，也要把它们列出来，以避免出现“lint warnings”。
- 3) 在实例化大模块时，每个端口要占据一行，“.port_name”要对齐，(signal_name)也要对齐，在括号内不要添加空格。有些人喜欢在括号内信号名的左右两侧加一些空格，这样做其实并不好看。
- 4) 在实例化大量的小模块时，可以采用紧凑的格式，即把多个端口信号放到一行上。例如，对大量 PAD 的实例化。
- 5) 模块实例名字后面的括号要另起一行，放置在固定的列数上，而且括号后面不要加空格，这样后续的端口信号依此对齐。

例子：不好的格式，括号没有另起一行，端口和“(”对齐，信号没有对齐。

```

cgu    cgu_i(
        .preset_n(cgu_preset_n),
        .pclk(cgu_pclk),
        .psel(psel_cgu),
        .paddr(paddr),
        .penable(penable),
        .pwrite(pwrite),
        .pwwdata(pwwdata),
        .prdata(prdata_cgu),
        .....
    );

```

例子：好看的格式，括号另起一行，端口与“(”保持对齐，信号对齐。

```

cgu                cgu_i
(
    .preset_n      (cgu_preset_n),
    .pclk          (cgu_pclk),
    .psel          (psel_cgu),
    .paddr         (paddr),
    .penable       (penable),
    .pwrite        (pwrite),
    .pwwdata       (pwwdata),
    .prdata        (prdata_cgu),
    .....
);

```

9.10 保持一致

有些人喜欢这样的风格，有些人喜欢那样的风格，这属于个人的审美偏好。选择一种风格而非另一种风格，不会对代码的可读性有什么影响，但是如果把两种风格混合在一起，就会真的对代码的可读性有影响。

另外，使用习惯的方式编写代码还受心理因素影响，因为程序员都有着强烈的意识，觉得其他程序员也会遵循习惯的方式。

所以当我们参与到一个项目中，即使我们感觉不习惯这个项目的编码风格，我们还是要遵守已成事实的风格，因为保持编码风格的一致性是很重要的。

9.11 代码例子

下面是 C 语言和 Verilog 代码例子，它们都具有完美的格式，遵守缩进，并添加空白。

例子：strstr 函数，来源于 newlib

```
char *strstr(const char *searchee, const char *lookfor)
{
    if (*searchee == 0) {
        if (*lookfor)
            return (char *)NULL;
        return (char *)searchee;
    }

    while (*searchee) {
        size_t i = 0;
        while (1) {
            if (lookfor[i] == 0)
                return (char *)searchee;
            if (lookfor[i] != searchee[i])
                break;
            i++;
        }
        searchee++;
    }

    return (char *)NULL;
}
```

例子：ccf_gen_div

```
module ccf_gen_div
    #(parameter NUM_WIDTH = 4, DEN_WIDTH = 10)
    (input          reset_n,
     input          clk_in,
     input          enable,
     //numerator
     input [NUM_WIDTH-1:0] numerator,
     //denominator
     input [DEN_WIDTH-1:0] denominator,
     //Duration control, can be n/2
     input [DEN_WIDTH-1:0] duration,
     output         clk_out);

    reg [DEN_WIDTH-1:0] R_count;
    wire [DEN_WIDTH-1:0] diff = (denominator - numerator);
    wire load = (R_count >= diff);
    always @(posedge clk_in or negedge reset_n)
        begin
            if (!reset_n)
                R_count <= 0;
            else if (enable) begin
                if (load)
                    R_count <= R_count - diff;
                else
                    R_count <= R_count + numerator;
            end
            else
                R_count <= 0;
        end
end
```

```
reg R_comp;
wire comp = (enable && R_count >= duration);
always @(posedge clk_in or negedge reset_n)
    begin
        if (!reset_n)
            R_comp <= 0;
        else if (enable)
            R_comp <= comp;
        else
            R_comp <= 0;
    end

    assign clk_out = R_comp;
endmodule
```

第 10 章

注释合理

当你编写代码时，你的脑海中会有很多有价值的信息。但是如果你不对代码添加注释加以说明，那么当别人阅读你的代码时，他们看到的只是干巴巴的代码，很难看懂这些代码。

注释是对代码的解释和补充，体现着设计者的思想。通过阅读注释，读者能够更方便地阅读代码，读者能够更好地思考代码在做什么。但是，我们要注意对代码的可读性起决定因素的不是注释，而是编码风格，编码风格包括含义清晰的名字、优美的编码格式、清晰的代码结构、简单易懂的算法等。所以在写注释前，首先要有好代码，不要为滥代码写注释。

很多人不喜欢写注释，觉得写注释没什么用处，而且浪费时间，这是因为这些人把写代码和写注释分割开来，看成不同的工作，他们只是在代码已经很稳定时，才开始写注释，但是这时写注释要花费更多的时间，因为还要回忆或思考某行代码在干什么。如果你在写代码时就随时把你的想法写出来，虽然这时写出的注释不太完整，但是以后你在浏览修改代码时，可以进一步完善这些注释，这样不是挺好吗？关键在于你要养成随时写注释的习惯。

注释不仅要占用代码行的空间，而且阅读时还要花费一些时间，所以注释应该物有所值，确实能对理解代码有所帮助。所以，我们首先要区分一下无用的注释和有用的注释。

10.1 无用的注释

注释应该让人对代码加深理解，而不应该让人对代码产生疑惑。任何一行注释都必须有意义，对代码没有意义的注释都要摒弃掉。但是现实代码中存在很多无用的注释，更是存在着一些乱七八糟、陈旧过时、误导读者的注释，这些注释对代码有极大的破坏性。

10.1.1 毫无用处的注释

如果代码本身就已经写得清晰明了，从代码就可以迅速地推断出代码的意图，添加没有提供任何新信息的注释就变得毫无意义，这些注释对于读者没有什么帮助。

例子：毫无用处的注释

```
char name[20];           /*The name*/
char version[16];       /*The version */
char licence_name[128]; /*The licence name*/
/*Return the days of the specified month*/
int get_days_of_month (int month)
```

例子：毫无用处的注释

```
form_title = 1; //是否显示 title, 1 表示显示, 0 表示不显示
form_add = 1; //是否允许增加, 1 表示可以, 0 表示不可以
form_select = 0; //是否允许选择, 1 表示可以, 0 表示不可以
form_count = 0; //输入对象的行数
```


例子：毫无用处的注释，因为任何人都明白“addr+1”的含义

```
//Increment addr
addr = addr + 1;
```

10.1.2 循规蹈矩的注释

有的公司要求程序员在每个函数前面都要添加一段注释，解释函数的用途和参数的使用。这只是在循规蹈矩，有些对函数的注释只是把函数和参数的名字做了简单展开，有时从函数和参数的名字，很容易就知道函数的用途和用法，这样的注释其实没有太大的用处，但是程序员有时不得不这样做。

例子：函数说明的常用格式

```
/******
Function:      //函数名称
Description:   //函数功能、性能等的描述
Calls:        //本函数调用的函数清单
Called By:    //调用本函数的函数清单
Input:        //输入参数说明，包括每个参数的作用、取值说明及参数间关系。
Output:       //对输出参数的说明
Return:       //函数返回值的说明
Others:       //其他说明
*****/
```

例子：循规蹈矩的注释

```
/* Function Name:  GetMaxAndMinData
   Description:    Find the max and min value in the array
   input:         nArr is the array, Num is the number of the array
   output:        nMax is the max value, nMin is the min value
   return:        -1/FAIL, 0/SUCCESS */
int GetMaxAndMinData(int nArr[], int nNum, int *nMax, int *nMin)
{
    int tempMax = 0;
    int tempMin = 0;
    if (nNum <= 0)
        return -1;
    for (int i = 0; i < nNum; i++)
    {
        if (nArr[i] > tempMax) tempMax = nArr[i];
        if (nArr[i] < tempMin) tempMin = nArr[i];
    }
    *nMax = tempMax;
    *nMin = tempMin;
    return 0;
}
```

10.1.3 代码混乱的注释

代码设计得混乱、重复、啰嗦、臃肿，变量名和函数名含义不清晰，代码的可读性很差，就添加大量的注释解释代码的功能，解释名字的含义，能说清楚吗？对这样的代码添加注释，可能根本就解释不清，只能增加复杂、零碎、啰嗦。其实这些都是“拐杖式注释”^[20]，本来代码是在跛足而行，然后又用注释给代码添加一支拐杖。

好的名字比好的注释更重要，因为任何用到这个名字的地方都能看到它。好的代码比好的注释更重要，因为读者如果能直接看明白代码，就无须注释，因为“好代码 > 坏代码 + 好

注释”。

例如，`fast_memory_copy` 具有两个用途，复制和填充，但是名字却只有明确的“复制”含义，于是还要解释清楚“`n&0x8000`”的用途。所以应该把这个函数拆分为两个函数：`fast_memory_copy` 和 `fast_memory_set`，这时因为函数功能单一，名实相符，只须简单地解释 `n` 的使用要求即可。

例子：拆分前的 `fast_memory_copy`

```
//fast_memory_copy has two usage: copy or set.
//if (n & 0x8000 == 0), then copy n words from src into dst
//if (n & 0x8000 == 1), then set x words of src into dst. x = n & 0x7FFF;
//n must be the multiple of 8
void fast_memory_copy (void *dst, void *src, int n)
```

例子：拆分为 `fast_memory_copy` 和 `fast_memory_set`

```
//n must be the multiple of 8
void fast_memory_copy (void *dst, void *src, int n)
//n must be the multiple of 8
void fast_memory_set (void *dst, int value, int n)
```

10.1.4 误导理解的注释

有的注释内容混乱、词不达意、毫无关联、不知所云、啰里啰嗦、事无巨细、故意卖萌、呐呐自语，总之就是误理解，对于这样的注释就应该直接删去。

例子：啰里啰嗦的注释

```
/* This is a for loop that prints the words "Number xxx" to the console screen
  1 million times each on its own line. It accomplishes this by starting
  at 0 and incrementing by 1. If the value of the counter equals 1 million
  the for loop stops executing. */
for (i = 0; i < 1000000; i++) printf ("Number %d\n", i);
/* I discussed with Jim from Sales over coffee at the Starbucks on main street
  one day and he told me that Sales Reps receive commission based upon the
  following structure. Did I mention that I ordered the Caramel Latte with
  a double shot of Espresso? */
```

例子：故意卖萌的注释

```
//这个函数为赞美冥王星而创建。
//这段代码非常 Cool，到底是做什么的呢？
//老猪老猪我爱你，就像老鼠爱大米。
```

10.1.5 语句错误的注释

有些人用英语写注释，但是单词错误，语法不对，不符合英语习惯，有的还用“中国式英语”；有些人用中文写注释，但是有错别字，有病句，不符合中文习惯，还使用网络流行词语。这样的注释都显得不伦不类的，读起来很费劲。

例子：相似或相近的单词，容易用错

“adapt、adopt、adept”，“contend、content、context、contest”，“accept、receive”，“bring、take、fetch、carry、get”，“a few、a little”，“look、see、watch”

例子：语法错误

“have been copy”，“is belong to me”，“We should read books may be useful to us”，“We have little time to read some books which we interest”

如果你要用英语做注释，就要符合英语的习惯，单词、语法、断句都要正确。另外，在

每个标点符号后面要加一个空格，这是英语的习惯，但是很多人就是不注意。

如果你在用中文添加注释，不要使用错别字，例如，不要滥用“在、再”，滥用“的、地、得”，“度假”写成“渡假”；不要使用网络流行的词语，例如，“晕”、“哇塞”、“稀饭”、“酱紫”、“然并卵”、“也是醉了”、“喜大普奔”；也不要写错误的句子，例如，“在老师的耐心帮助下，使他的学习成绩提高很快”。

10.1.6 格式混乱的注释

注释要与它所注释的代码保持上下对齐，当代码发生了缩进(开始列发生了前移或后移)，对应的注释也要做相应的移动。另外，还有些人写注释时直接从第一列开始，从不考虑与代码对齐的问题，这样看着也非常不好。

例子：格式混乱的注释，这样的注释会扰乱代码的逻辑结构

```

        //Write the flash address
    if (nandOperPtr->transferFlag == 0 || f_off == 0)
    {
//Note: gNFC_Chip_Select can be placed here safely,
        //because CS# is only changed here.
        gNFC_Chip_Select = flash_cs_value[f_idx];
//Note: If transferFlag is 0, p_off must be EVEN.
        //Calculate the column address
        col_addr = p_off * FLASH_PART_SIZE;
        //Issue the data input command
        FLASH_WriteCMD (FLASH_CMD_MASK | CMD_PROG_80);
        //Issue the address of p_idx plane
        FLASH_WriteAddress_New (col_addr, nandOperPtr->wRowAddr[p_idx], 5);
    }

```

10.1.7 格式繁琐的注释

注释时搞出一些花样，看着花里胡哨的，看起来似乎很美，有用吗？读者关心的是这段文字，而不是这些繁琐花哨的格式。每次修改这样的注释，都要花费一些时间来调整格式。

例子：格式繁琐的注释

```

//=====
//== rtc_clk : pp_rtc_xi differences: ==
//== 1. rtc_clk is down during power-off (pp_resetp_n/resetp_up_n=0) ==
//== 2. rtc_clk becomes active 2 cycle after valid resetp_up_n ==
//== 3. rtc_clk domain registers can be scanned while pp_rtc_xi NOT ==
//== 4. rtc_clk's rtc_reset_n contains all resets while pp_rtc_xi NOT ==
//== Principle: pure software-relative registers use rtc_clk ==
//=====

```

10.1.8 屏蔽代码的注释

当某一段代码不再使用的时候，很多人就会用“//”或者“/* */”把这段代码注释掉。即使这些代码毫无用处，以后不会再用到，它们还会静静地呆在那里，慢慢地在那里腐败发酵变臭。另外，如果一段代码中包含有用“/* */”写的注释，然后你又用“/* */”把这段代码注释掉，编译时就会报告注释嵌套的警告。

例子：屏蔽代码的注释

```

/* This block of code is no longer needed because we
   found out that Y2K was a hoax and our systems did

```

```

    not roll over to 1/1/1900 */
/*char today[40];
get_system_date (today);
if (strcmp (today, "19000101") == 0)
{
    strcpy (today, "20000101");
    char *message = "The date has been fixed for Y2K.";
    printf ("%s\n",message);
} */

```

屏蔽不用的代码，最好使用`#if 0 ... #endif`；如果从两段代码中选择使用一段代码，最好使用`#if 0 ..#else... #endif`，或者`#if 1 ..#else... #endif`。

注意：要定期清理掉无用的代码，因为“`#if 0/1`”写多了，看着既闹心，又分散注意力。

```

例子：使用#if 0 and #if 1
#if 0
int opt_index = 1;
int opt_buffer[128];
void record_fill_info (int val)
{
    opt_buffer[0] = opt_index;
    opt_buffer[opt_index] = val;
    if (opt_index == 127) opt_index = 1;
    else opt_index++;
}
#else
int opt_index = 1;
char opt_buffer[512];
void record_fill_info (char val)
{
    opt_buffer[0] = opt_index;
    opt_buffer[opt_index] = val;
    if (opt_index == 511) opt_index = 1;
    else opt_index++;
}
#endif

```

不要保留那些陈旧过时的代码，如果确实已经没有任何保留价值，该删除的就删除掉吧，“往事不要再提，人生已多风雨，纵然记忆抹不去，爱与恨都还在心里，真的要断了过去，让明天好好继续……。”

10.1.9 陈旧过时的注释

当代码已经发生改变，但是原来的注释还保留着，没有做同步修改，使得代码和注释不一致。注释存在的时间越久，就越偏离所要描述的代码，甚至注释跟代码变得完全不一致。这样的注释会使人产生误解，读者看懂了注释，但是却与代码对不上，你说让人恼火不？

程序员应当负责让注释和代码保持一致，当代码发生了变化，要及时修改或删除注释，保证注释和代码之间是相互对应的。

10.1.10 用作标识的注释

有的人会用注释在代码里做一些标识，用于记录修改的人员和日期，或者一些没什么用处的标识。

例子：记录修改的人员和日期

```
ftl_vpb_count -= 128; //Added by Wei Jiaming on 2015-10-24
power_delay_ms = 25; //Corrected by Wei Jiaming on 2015-09-13
```

例子：随便添加一些标识

```
//Start//////////////////////////////////////
//Stop=====
```

10.1.11 用作日志的注释

有些人在修改代码时，就在文件的头部手工添加或者由工具添加一些注释，记录修改的人员、时间、版本、原因等。

例子：记录修改历史的注释

```
// Revision 1.8 2004/06/09 14:43:05 rayb
// updated copright notices
// Revision 1.7 2002/10/30 16:51:11 jstokes
// removed some redundant code
// Revision 1.6 2002/09/23 13:25:30 jstokes
// cosmetic changes - fix leda errors
// Revision 1.5 2002/09/12 13:15:44 jstokes
// got rid of macro log_n replaced by macro LOG2_SBW, which is calculated by
// tcl script. FC2 had a problem with a bit slice being taken by a nested
// ( x=1 ? b : a ) type statement
// Revision 1.3 2002/07/11 13:14:23 jstokes
// Now outputs byte lane enables for both the bus endianness and little
// endian format. Littly endian is always required for wdata, as this is
// always transformed to little endian.
// Revision 1.2 2002/06/28 09:13:11 jstokes
// Cosmetic changes and some bug fixes
// Revision 1.1 2002/05/23 15:51:16 jstokes
// moved from /module
// Revision 1.1 2002/05/21 08:54:48 jstokes
// Created, Exact same as Michele's version, only name slightly changed.
```

很久以前，修改历史确实应该放在文件头部的注释里，但是现在就不应该这么做了，应由源代码控制系统（CVS 或 SVN）或者问题追踪系统统一管理，因为修改历史记录只会用大量过时且无趣的文本搞乱代码。

10.1.12 不好注释的例子

例子：乱七八糟的注释

```
/*
*****/
/*
*****/
/*<FUNC+>*****
*****/
/* 函数名称: NorFlash_Erase */
/* 功能描述: NorFlash 擦除 */
/* 输入参数: Offset 从哪个偏移位置开始擦除，偏移值从 0 开始，一个偏移值代表 512 字节一个扇区
* Len 擦除的字节数，Len 是 512 的整数倍 */
/* 输出参数: 无 */
/* 返回值: 0 成功 */
/* 操作流程: */
```

```

/* 其他说明:                                     */
/* 修改记录:                                     */
/* ----- */
----- */
/*    2014-08-1    V1.00    wuyouren    创建函数    */
/*<FUNC->***** */
*****/
int NorFlash_Erase(unsigned int Offset, unsigned int Len)
{
    unsigned char *startPtr, *endPtr;
    //BSP_GIntCtrl (0);
    startPtr = (unsigned char *) (0x50000 + (Offset*512)); /* startAddr */
    endPtr = (unsigned char *) (startPtr + Len); /* endAddr */
    for (; startPtr < endPtr; startPtr += 512)
    {
        NorFlash_ExecuteCmd(0xB, (unsigned int *)startPtr, 0x00);
    }
    //BSP_GIntCtrl (1);
    return 0;
}

```

这段代码的注释包含了很多无用的注释，很多地方都是在糊弄差事。

- 1) 格式繁琐，表面上好看，但是有些地方因为没有对齐，导致看起来乱七八糟。
- 2) 函数名本来就透露出函数的功能，再添加注释描述功能，根本就没什么用处。
- 3) 形参 `Offset` 和 `Len` 的含义不明确，导致需要添加额外的注释。
- 4) `startPtr` 和 `endPtr` 后面的注释也是毫无用处，赋值的意思就已经清晰明了。
- 5) 使用注释把过时无用的代码屏蔽掉。

笔者曾经问设计人员，为什么要把注释写成这样？他说这段程序要提供给客户，为了表现代码的细致周详，才写成这样。

```

例子：修改后的注释，更改形参的名字，让含义更清晰，同时删掉无用的“{ }”
//Input: start_sector 是要擦除的开始扇区号，每个扇区有 512 字节。
//      total_bytes 是要擦除的总字节数，应该是 512 字节的整数倍。
//Return: always be 0/SUCCESS
int NorFlash_Erase (unsigned int start_sector, unsigned int total_bytes)
{
    unsigned char *start_ptr, *end_ptr;
    start_ptr = (unsigned char *) (0x50000 + (start_sector*512));
    end_ptr = (unsigned char *) (start_ptr + total_bytes);
    for (; start_ptr < end_ptr; start_ptr += 512)
        NorFlash_ExecuteCmd(0xB, (unsigned int *)start_ptr, 0x00);
    return 0;
}

```

10.2 有用的注释

最好的注释是代码本身，因为如果代码本身清晰明了，就根本不需要额外的注释，所以你要首先写好代码，用代码记录你的设计思想。

最好的注释是程序员认真编写的，使用正确的语法、拼写和格式，字斟句酌，保持简洁，没有闲扯，没有画蛇添足。

最好的注释是站在读者的角度看待问题，去想象他们需要什么样的注释，能够对代码给出清晰明了的解释。

10.2.1 记录版权信息

代码中添加的版权信息主要包括这些内容：文件名、公司、作者、时间、功能描述、创建标识、修改标识、修改描述等。这些内容按照固定的格式放在文件的前面。

例子：记录版权信息

```

/*****
 * COPYRIGHT (c) 2000 Denali Software, Inc. All rights reserved. *
 * ----- *
 * This code is proprietary and confidential information of      *
 * Denali Software. It may not be reproduced, used or transmitted *
 * in any form whatsoever without the express and written      *
 * permission of Denali Software.                               *
 *****/
 * Module: clock_contol.v                                       *
 * Designer: Steven Shrader                                     *
 * Date: 5/19/2003                                             *
 * Documentation:                                              *
 * Version:                                                     *
 *****/
// $Id: denali_mc_clock_control.v,v 1.2 2011/06/02 10:04:53 kyx Exp $
// DENALI VER: 516 CUSTOMER VER: 0 REL:

```

10.2.2 全局观的注释

这种注释放在文件开头，从全局的观点解释一组相关函数或一个模块的用途。

例子：DW_ahb_dmac_busmux.v 的对自身的解释

```

/* 1. Parameterized, parallel one-hot mux that will multiplex several buses
   (controlled by BUS_COUNT parameter) of a particular width (specified by
   MUX_WIDTH parameter).
  2. The module would be able to mux three 12-bit buses, or seven 5-bit buses,
   or any other combination, depending on the parameter values used when the
   module is instantiated.
  3. In order to have a consistent port structure regardless of the number of
   inputs, the input data buses must be concatenated into a single vector.
  4. Note that the select lines must be mutually exclusive.      */

```

10.2.3 解释代码意图

你可能听说过注释只应该说明“为什么”，不应该去说“做什么”和“怎么做”，我们的建议是其实你不用去区分“为什么”、“做什么”和“怎么做”，你可以做任何能帮助读者理解代码的事情。你最好站在读者的立场考虑问题，也就是常说的“换位思考”，如果某段代码确实复杂，就注释加以解释^[20]。例如，解释常量为什么使用这个值，解释相关变量之间的关系，解释代码段的运行机制。

通常每个代码段都要有一段注释，如果代码简单，就做很简单的概述性注释；如果代码复杂，那么就详细解释代码段的工作机制。这样的注释很有用，因为读者在深入研究这段代码的细节之前，从注释就可以知道这段代码的用处。

如果很快就能推断出代码段的含义，通常就没有太大的必要去添加注释。但是对于一些读者，读有注释的代码比读没有注释的代码还是要快速得多，所以可以添加几句概述性的注释。

例子：解释为什么加 1

```

//In burst mode, the bytes are written in consecutive addresses. Need to

```

```
//access the next address to verify that the next byte was properly saved.
addr = addr + 1
```

例子：代码段的注释

```
//DMA transmit fifo data level register
//Write control for 'dmatdler'
assign tx_fifo_depth = `SSI_TX_FIFO_DEPTH - 1'b1;
always @(posedge pclk or negedge presetn)
begin
    if (presetn == 1'b0)
        dmatdler_ir <= {8{1'b0}};
    else if (dmatdler_we == 1'b1
        && byte_en[0] == 1'b1
        && ipwdata[7:0] <= tx_fifo_depth[7:0])
        dmatdler_ir[7:0] <= ipwdata[7:0];
end
```

例子：strlen 的注释

```
size_t strlen (const char *str)
{
    const char *start = str;
    unsigned long *aligned_addr;

    /* Align the pointer, so we can search a word at a time. */
    while (UNALIGNED (str)) {
        if (!*str)
            return str - start;
        str++;
    }

    /* If the string is word-aligned, we can check for the presence of
       a null in each word-sized block. */
    aligned_addr = (unsigned long *)str;
    while (!DETECTNULL (*aligned_addr))
        aligned_addr++;

    /* Once a null is detected, we check each byte in that block for a
       precise position of the null. */
    str = (char *) aligned_addr;
    while (*str)
        str++;
    return str - start;
}
```

注释还可以把晦涩难懂的语句翻译为某种可读的形式，这样便于理解代码。

例子：解释晦涩难懂的语句

```
double __ieee754_log10(double x)
{
    double y,z;
    __int32_t i,k,hx;
    __uint32_t lx;

    EXTRACT_WORDS(hx,lx,x);
    k=0;
    if (hx < 0x00100000) { /* x < 2**(-1022) */
        if (((hx&0x7fffffff)|lx)==0) /* log(+0)=-inf */
            return -two54/zero;
        if (hx<0) return (x-x)/zero; /* log(-#) = NaN */
        k -= 54; x *= two54; /* subnormal number, scale up x */
        GET_HIGH_WORD(hx,x);
    }
}
```



```
}
.....
}
```

10.2.4 记录代码瑕疵

代码始终都在向前演进，在这个过程中肯定会有瑕疵，你可以在要修改的地方使用注释记录当前状态，同时记录代码将来应该怎样做改动。读者通过这种注释就可以知道代码的当前状态和未来改进的方向。

这样的注释常用 **TODO**、**FIXME** 等关键字，用于警示它们与其他的注释不一样。**TODO** 用于标识程序员认为应该做但是还没有做的功能，说明如何实现这个功能。**FIXME** 用于标识代码可能有缺陷或者错误，或者需要提高性能，说明如何修改代码，如何做进一步增强。

例子：记录代码的状态

```
//TODO: The below algorithm is O(nlogn) now and need be optimized to O(n).
//FIXME: The below code is not finished at all, but they can be used now.
```

10.2.5 记录注意事项

注释可以用来记录代码的很多注意事项。

- 1) 记录某段代码的运行效率或准确度，告诉读者当前代码已经是最优的，不要再做无谓的优化。
- 2) 记录当前代码状态，告诉当前代码有点乱，需要等测试完后再做代码调整。
- 3) 记录代码中存在的问题，提醒其他人使用时要注意。
- 4) 说明函数的功能用途和参数的使用范围，可以防止函数被错误使用，做到“未雨绸缪”。
- 5) 举例说明函数的使用方法，这样的方法更加直观，而且不用花费太多的解释。

例子：记录注意事项

```
//Note: lba must be < ftl_max_capacity, otherwise error occurs.
//Note: this function can only be used after FTL_ReadOpen is executed
correctly.
```

例子：记录注意事项

```
// One of the subtleties that might not be obvious that makes this work
// so well is the use of the blocking assignment (=) that allows
// data_out to be built up incrementally. The one-hot select builds up
// into the wide "or" function you'd code by hand.
always @ (selects or data_in) begin
    integer i, j;
    if (!ENABLE) // If not enabled, connect it to 0
        data_out = 0;
    else if (BUS_COUNT == 1) // single port, do nothing, pass the data
        data_out = data_in;
    else begin // let's build the mux
        data_out = 'b0;
        for (i = 0; i < BUS_COUNT; i = i + 1)
            for (j = 0; j < MUX_WIDTH; j = j + 1)
                data_out[j] = data_out[j] |
                    (data_in[MUX_WIDTH*i + j] & selects[i]);
    end
end
```

10.3 如何写注释

注释一定要认真写，因为好的注释不仅是对工作的负责，也是对读者的尊重。

10.3.1 整理代码

首先，我们应该意识到最能说清楚代码设计思想的是代码本身，所以当代码很复杂、很杂乱或很糟糕时，不要为了解释这样的代码加上一大堆注释，应该把代码整理干净，让代码变得清晰易懂，然后再添加注释。

其次，还要保证代码和注释保持一致，在整理代码的同时，不能让注释陈旧过时，也不能让注释与代码产生冲突，更不能让注释根本就是错误的，否则会严重误导读者。

10.3.2 注释位置

一般在代码的如下位置添加注释。

- 1) 文件头，对文件内的函数和模块做一个整体介绍，然后描述其实现功能、关键特性等。
- 2) 函数头，解释函数的用途、怎么样调用、参数的要求、返回值的说明。
- 3) 常量和变量，解释常量和变量的用途，变量的取值范围等。
- 4) 代码块，说明代码的意图，根据需要做简单的说明，或者做详细的说明。
- 5) 根据需要添加一些特殊的注释，例如 Note、Warn、Todo 和 Fixme 等。
- 6) 尽量把注释放在单独的行上，最好不要把注释放在代码行的尾部，可以根据情况灵活掌握。如果注释很短，例如用来解释变量，那么就可以考虑放在行尾。

10.3.3 遣词造句

注释的语句要把代码说清楚，所以要注意如下事项。

- 1) 注释要能够说明设计意图，要做到简明扼要、精确紧凑，避免啰嗦、废话连篇。
- 2) 注释要列出要点。如果一个问题需要用很多语句来描述，那么最好用 1、2、3、……列出每个要点，这样看起来更清晰。
- 3) 可以使用例子说明函数的用法，说明代码块的工作机制。
- 4) 根据情况选用中文或英文注释，但尽量还是使用英文。
- 5) 如果用中文做注释，就要符合中文的习惯，不要用错别字，不要写病句，也不要流行的网络词语。
- 6) 如果用英文做注释，就要符合英文的习惯，单词、语法、断句都要正确。另外，在每个标点符号后面要加一个空格，这是英文的习惯，但是很多人就是不注意。

10.3.4 书写格式

注释的格式要看着舒服，所以要注意如下事项。

- 1) 注释要放在它所注释的代码块附近，在每个代码块或重要代码行的上方添加注释。
- 2) 注释最好不要放在代码行的尾部，但是可以根据情况灵活掌握。
- 3) 注释要使用有效实用的格式，不要把注释搞得繁琐花哨，否则修改起来不方便。

- 4) 注释只有一行或数行时, 最好使用 “//”, 而不要使用 “/* */”, 因为 “//” 用起来更加简洁。
- 5) 注释有多行文字时, 需要大段描述, 最好使用 “/* */”。
- 6) 注释要注重格式, 符合缩进格式, 要与注释的代码保持上下对齐。
- 7) 注释的每行语句不要太长, 不要超过 80 个字符, 要注意换行。

10.3.5 实际例子

例子: newlib 库中的 `__kernel_sin()` 函数

```

/*=====
Copyright (C) 1993 by Sun Inc. All rights reserved.
Developed at SunPro, a Sun Microsystems, Inc. business.
Permission to use, copy, modify, and distribute this
software is freely granted, provided that this notice
is preserved.
=====*/

/* __kernel_sin( x, y, iy)
kernel sin function on [-pi/4, pi/4], pi/4 ~ 0.7854
Input x is assumed to be bounded by ~pi/4 in magnitude.
Input y is the tail of x.
Input iy indicates whether y is 0. (if iy=0, y assume to be 0).

Algorithm
1. Since sin(-x) = -sin(x), we need only to consider positive x.
2. if x < 2^-27 (hx<0x3e400000 0), return x with inexact if x!=0.
3. sin(x) is approximated by a polynomial of degree 13 on
   [0, pi/4]
           3           13
       sin(x) ~ x + S1*x + ... + S6*x
   where
   | sin(x)           2       4       6       8       10       12 |   -58
   |----- - (1+S1*x +S2*x +S3*x +S4*x +S5*x +S6*x )| <= 2
   | x                                     |
4. sin(x+y) = sin(x) + sin'(x')*y
   ~ sin(x) + (1-x*x/2)*y
   For better accuracy, let
           3       2       2       2       2
       r = x *(S2+x *(S3+x *(S4+x *(S5+x *S6))))
   then
           3       2
       sin(x) = x + (S1*x + (x *(r-y/2)+y))
*/
#include "fdlibm.h"

#ifdef __STDC__
static const double
half = 5.000000000000000000000000e-01, /* 0x3FE00000, 0x00000000 */
S1 = -1.6666666666666666666666324348e-01, /* 0xBFC55555, 0x55555549 */
S2 = 8.33333333332248946124e-03, /* 0x3F811111, 0x1110F8A6 */
S3 = -1.98412698298579493134e-04, /* 0xBF2A01A0, 0x19C161D5 */
S4 = 2.75573137070700676789e-06, /* 0x3EC71DE3, 0x57B1FE7D */
S5 = -2.50507602534068634195e-08, /* 0xBE5AE5E6, 0x8A2B9CEB */
S6 = 1.58969099521155010221e-10; /* 0x3DE5D93A, 0x5ACFD57C */

double __kernel_sin(double x, double y, int iy)
{
    double z,r,v;

```

```
__int32_t ix;
GET_HIGH_WORD (ix, x);
ix &= 0x7fffffff;          /* high word of x */
if (ix < 0x3e400000)      /* |x| < 2**27 */
if ((int)x == 0) return x; /* generate inexact */
z = x * x;
v = z * x;
r = S2 + z * (S3 + z * (S4 + z * (S5 + z * S6)));
if (iy==0) return x + v * (S1 + z * r);
else return x - ((z * (half * y - v * r) - y) - v * S1);
}
```

第 11 章

名字定义

父母在给子女取名字的时候，都要仔细斟酌，希望取个好的名字能给子女带来幸福平安，然而取个不好的名字就有可能影响子女的一生，例如“赵狗剩”、“钱乱花”、“孙三臭”、“李四喵”。

代码中的名字也一样，名字可以当作一条小小的注释，尽管空间不算很大，但好的名字可以让它承载更多的信息，可以给代码带来更好的可读性，而不好的名字会给工作带来各种麻烦和不便。所以要定义好模块、函数、常量、变量等名字。

定义好的名字，就要遵循一定的命名规范，给出有意义的、长度合适的中文名字和英文名字。设计、测试、开发和销售等人员都要统一使用这些定义好的名字，以保持名字的一致性，便于大家的交流，也便于在整个系统内保证代码的可读性。

11.1 命名方法

命名方法就是一种命名惯例，并无绝对与强制，目的就是为增强代码的可读性。名字一般是由多个单词连接在一起构成的，按照不同的连接规则，形成不同的命名方法。

有些程序员喜欢全部小写，有些程序员喜欢用下划线，还有些程序员喜欢其他的方法，所以如果要命名一个“fish name”的变量，常用的写法会有 `fishname`、`fish_name`、`FishName` 或者 `fishName`。但是混合使用不同的命名方法会给程序员的阅读和交流带来不便。所以在编写代码时，一旦选用某种命名方法，就应保持命名的一致性。

对于一种语言来说，或者对于某个应用来说，都有其约定俗成的命名方法，所以当我们参与其中的时候，就要遵守这些命名规则，也就是要“入乡随俗”。

11.1.1 帕斯卡命名法

帕斯卡命名法 (Pascal)，源自于 Pascal 语言的命名惯例，也称为“大驼峰命名法”，为驼峰式大小写的子集。

它的命名规则是：单词之间不以空格断开，不以连接号 (-)、下划线 (_) 连接；每个单词的首字母都要大写。

例子：`FirstName`、`LastName`、`PrintEmployeePaychecks()`

11.1.2 骆驼式命名法

骆驼式命名法 (Camel-Case) 来自于 Perl 语言中普遍使用的大小写混合格式，而 Larry Wall 等人所著的畅销书《Programming Perl》的封面图片正是一匹骆驼。

骆驼式命名法是指混合使用大小写字母来构成变量和函数的名字。因为使用此法命名的名字看上去就像骆驼的驼峰一样高低起伏，所以又称为驼峰命名法。

骆驼式命名法分为小驼峰命名法（Lower Camel Case）和大驼峰命名法（Upper Camel Case）。小驼峰法的命名规则是：除第一个单词之外，其他单词的首字母都要大写。大驼峰法的命名规则是：每个单词的首字母都要大写，也就是 Pascal 命名法。

例子：firstName、lastName、printEmployeePaychecks()

骆驼式命名法在许多新的函数库和 Microsoft Windows 这样的环境中使用得相当多。在 Java 中，类一般用大驼峰法命名，方法和变量则多用小驼峰法命名。

11.1.3 匈牙利命名法

匈牙利命名法是一位叫 Charles Simonyi 的匈牙利程序员发明的，他在微软工作了二十多年，于是这种命名法就通过微软的各种产品和文档资料向世界传播开了。

匈牙利命名法的基本原则是：“变量名=属性+类型+对象描述”，其中每个对象的名称都要求有明确含义，可以取对象名字全称或名字的一部分。

属性部分：g_全局变量，c_常量，m_类成员变量，s_静态变量……

类型部分：布尔 b，无符号 u，整型 i，短整型 n，长整型 l，单精度浮点 f，双精度浮点 d，字符 ch，字 w，双字 dw，字符串 sz，数组 a，指针 p，函数 fn，句柄 h……

描述部分：最大 Max，最小 Min，临时变量 Temp，源对象 Src，目的对象 Dest……

例子：pfnDrinkWater，pfn 是类型描述，表示指向函数的指针，DrinkWater 是变量对象描述，所以它表示指向 DrinkWater 函数的函数指针变量。

例子：g_cchInputData，g_是属性描述，表示全局变量，c 和 ch 分别是计数类型和字符类型，一起表示变量类型，InputData 是变量对象描述，所以它表示一个以字符类型进行计数的全局变量。

例子：表单的名称为 form，简写为 frm，当表单变量名称为 SwitchTable 时，那么变量全称应该为 frmSwitchTable，这样就可以很容易从变量名看出 SwitchTable 是一个表单。

匈牙利命名法非常便于记忆，而且使变量名非常清晰易懂，增强了代码的可读性，便于程序员之间的交流。但是匈牙利命名法也有很大的缺点，缺点就在于命名的啰嗦，还有就是类型的复杂性增加了命名的难度，这就要求在使用中扬长避短，合理应用。

11.1.4 下划线连接法

下划线连接法是在 C 语言出现后开始流行起来的，在许多旧的程序和 Linux/Unix 这样的环境下，它的使用非常普遍。下划线连接法就是使用下划线把各个单词连接在一起。变量名和函数名中的单词通常都是小写的，常量名和宏定义名中的单词通常都是大写的。

比起前面的三种命名法，这种名字因为使用了“_”，使得名字变得长了一些，但是实际使用没有太大的问题。在 Verilog 代码中，这种方法也用得非常多。

例子：first_name、last_name、print_employee_paychecks()、MAX_PERSON_NUMBER

例子：Verilog 代码的例子

```
input          otg_utmi_txready,           //UTMI Transmit Data Ready
input [15:0]   otg_utmi_dataain,          //UTMI Receive Data
input          otg_utmi_rxvalidh,         //UTMI Receive Data Valid High
```

```
input          otg_utmi_rxvalid,           //UTMI Receive Data Valid
input          otg_utmi_rxactive,       //UTMI Receive Active
input          otg_utmi_rxerror,        //UTMI Receive Error is detected
input [1:0]    otg_utmi_linestate,      //PHY Line State
```

11.2 命名

我们要把有用的信息装入名字，名字就要起得有意义、易于理解、容易记忆、长度合适、方便书写，这样对编写代码、阅读代码都有很大的好处。

但是我们经常会在代码中看到很多不如意的名字，例如 `tmp`，这样的名字有时会很含糊；`size` 或者 `get`，看上去很合理，但可能只装入了很少的信息；有人竟然选择使用 `ak47` 作为临时变量的名字，可能是真的想不出好的名字。

如果你要选择使用这些简单的名字，那么你就应该给出合理的解释。如果仅仅是因为懒惰而滥用它们，实在是不应该。面对代码中的众多名字，你应该多费一些心思，想出一些更有表达力的名字，提高你的代码可读性，同时你的命名能力也会很快提升。

11.2.1 选择合适的名字

命名时要避免使用模糊、空洞、晦涩、混淆或奇怪的名字，要选择使用专业、具体、合适、准确且通用的名字，要能准确地描述所表示的事物，这样通过名字就能知道它的具体用途，而不用殚精竭虑地研究它的具体用途。

英语是一门丰富的语言，一个名字就可以有很多的同义词选择，要根据情况选择合适的单词。

例子：`send` 和 `receive` 有很多的同义词，通常使用 `send` 和 `receive`

`send` 具有这些同义词：`transmit`、`dispatch`、`deliver`、`forward`、`post`。

`receive` 具有这些同义词：`gain`、`obtain`、`accept`、`get`。

在用 Verilog 设计 UART、SPI 等模块时，就经常成对使用 `send/receive`，或者使用 `transmit/receive`。

在 Windows 系统中，`DispatchMessage()` 用于分发消息，`PostMessage()` 用于寄送消息。

命名相反的事物或操作时，就要使用常用的反义词，这样可以保持名字的一致性。

例子：常用的反义词

`add/remove`, `begin/end`, `first/last`, `next/previous`, `insert/delete`,
`lock/unlock`, `min/max`, `old/new`, `open/close`, `push/pop`, `send/receive`,
`show/hide`, `source/destination`, `start/stop`, `up/down`。

对于 `stack`，有压栈和出栈，就用 `stack_push`、`stack_pop`。

对于单链表，有插入节点、追加节点和删除节点，就用 `list_insert`、`list_append`、`list_delete`。

对于文件，有打开文件和关闭文件，就用 `fopen`、`fclose`。

对于硬件模块，有开始操作和停止操作，就用 `start_operate`、`stop_operate`。

命名时，根据需要组合出合适的名字，使其能够清晰地反映出要命名的事物。

例子：组合出合适的名字

```
total_expense、average_expense、max_expense、min_expense、
total_revenue、average_revenue、max_revenue、min_revenue、
#define MIN_VPB_NUMBER    127
#define MAX_VPB_NUMBER    2047
```

命名函数时，因为它们都是在实现特定的功能，就要使用特定的组合：“模块名+动词+名词”，也就是使用“主谓宾”这种格式。注意：不要使用名词作为函数名，要根据函数内部的操作来选择动词。

例如：使用组合的名字，定义出函数的名字

```
norflash_write_data、norflash_read_data、norflash_erase_data
当要获取链表的 size 的时候，
list_size() //这不是好的函数名字，因为直接使用名词作为函数名
list_get_size() //可以用于表示轻量级访问，函数内部不做什么计算，直接返回 size 值
list_compute_size() //可以用于表示重量级访问，函数内部要做一些计算，然后返回 size 值
```

命名布尔变量时，应该从名字就能反映出这个变量的值只有 `true` 和 `false`。最好不要使用否定的单词，例如 `not`、`never`、`nothing`、`disable`，因为阅读代码时，肯定的名字会比否定的名字更让人明白一些，否定的名字还是要让人绕一下子。

例子：布尔变量

```
found、ready、busy、error、success、pass、fail //常用的布尔变量名
work_done、operation_complete、init_finished //常用的布尔变量名
disable_smart_page = false; //不好
enable_smart_page = true; //更好
if (!spi_not_ready) //不好
if (!spi_busy) //更好
```

11.2.2 避免简单的名字

命名时要选择具体的名字，避免使用简单的名字，例如 `tmp`、`foo`、`bar`、`val`、`a1`、`a2`。简单的名字会给代码带来模糊，降低代码的可读性。

例子：对比下面两个 `strncpy` 的形参名字，是不是第二个更加清晰易懂？

```
char * strncpy (char *a1, char *a2, int n) //a1 和 a2 太简单，很难知道函数的用法
{
    int i;
    for (i = 0; i < n; i++) a1[i] = a2[i];
    return a1;
}
char * strncpy (char *dst, char *src, int n) //dst 和 src 很好，很容易就知道函数的用法
{
    int i;
    for (i = 0; i < n; i++) dst[i] = src[i];
    return dst;
}
```

例子：`tmp` 是一个简单的名字

```
int calculate_average (int count, int *number)
{
    int i, tmp = 0;
    for (i = 0; i < count; i++)
        tmp += number[i];
    return tmp / count;
}
```

这里 `tmp` 并不好，因为它没有反应出变量的功能，如果使用 `sum` 替代 `tmp`，就会更具有表达力。另外，有些人觉得使用 `index` 替换 `i` 也很有意义。其实不是这样的，因为把 `i`、`j`、`k` 用在循环计数和数组索引很符合常规，也很方便阅读，而如果使用 `index`，就显得有些啰嗦。

有的时候，把 *i*、*j*、*k* 用作循环计数和数组索引，不仅容易搞混，而且还容易写错，尤其是 *i* 和 *j* 之间，所以我们可以使用稍长一些的名字。

例如：对于二维数组

```
for (i = 0; i < team_count; i++)
    for (j = 0; j < event_count; j++)
        score[i][j] = 0;
//可以改为
for (ti = 0; ti < team_count; ti++)
    for (ej = 0; ej < event_count; ej++)
        score[ti][ej] = 0;
//或者可以改为
for (team_index = 0; team_index < team_count; team_index++)
    for (event_index = 0; event_index < event_count; event_index++)
        score[team_index][event_index] = 0;
```

有的时候，使用简单的名字也没有什么关系，也会带来很好的效果^[20]，虽然简单的名字没有包含太多的含义。

```
例子：使用简单的 tmp
if (right < left) {
    int tmp =right;
    right = left;
    left = tmp
}
```

在这里 `tmp` 这个名字就很好，因为 `tmp` 就是用作临时存储，作用域就在这三行，没有其他的用处，不会被挪作他用。

11.2.3 避免误解的名字

命名时要避免使用可能会带来误解的名字，要考虑名字是否容易混淆，要考虑这个名字是否有歧义，要做到名实相符。

0 和 O、1 和 l、2 和 z 就很容易混淆，所以命名时要特别注意。例如，不要命名这样的名字，`lba0` 和 `lbaO`，`wen1` 和 `wenl`。但是，有些 Verilog IP 提供商把源文件中的变量做了加扰处理，提供如下样子的代码，可以在一定程度上防止泄露代码的实现细节。

```
例子：因为名字混淆而难以理解
function [data_width-1:0] l01I0101;
    input [data_width*depth-1:0] A;
    input [addr_width-1:0] SEL;
    reg [31:0] 0101II11, 110I1010, 11II0010;
    begin
        l01I0101 = {data_width {1'b0}};
        11II0010 = 0;
        for (0101II11=0; 0101II11<depth; 0101II11=0101II11+1) begin
            if (0101II11 == SEL) begin
                for (110I1010=0; 110I1010 < data_width; 110I1010=110I1010+1) begin
                    l01I0101[110I1010] = A [110I1010 + 11II0010];
                end
            end
        end
        11II0010 = 11II0010 + data_width;
    end
endfunction
```

命名时，不要使用奇怪的名字或组合，否则让人很费解，猜不透意思。

例子：奇怪的名字

`marry_ymd` (`ymd = year、month、day` 的第一个字母) 就很奇怪，直接使用 `marry_day`，这多么明了。

命名时，要让名字有区分，不要让名字相似，否则会造成混淆。

例子：相似的名字

对于 `get_active_account()`、`get_active_accounts()`、`get_active_account_data()`、`get_active_account_info()`，这些容易区分开吗？

本来已经有了 `total_revenue` 和 `average_revenue`，然后又命名 `revenue_total` 和 `revenue_average` 用作别的用途，这些容易区分开吗？

命名时，不要命名容易引起混淆的名字，否则会造成理解上的歧义。

例子：名字有歧义

`filter("year <= 2015")`，`filter` 有歧义，这个语句是把 `year <= 2015` 的数据过滤掉呢，还是抽取出 `year <= 2015` 的数据呢？所以可以改为如下：

```
FilterOut("year <= 2015")    //如果过滤掉满足条件的数据
ExtractData("year <= 2015") //如果抽取出满足条件的数据
```

命名表示范围的变量时，在取名字的时候会有些烦恼，因为常用的范围有两种情况：包含尾部和不包含尾部。所以，对于表示范围的变量，我们可以做出这样的规定：

- 1) 如果包含尾部，就用 `first` 和 `last`，范围内包含 `first` 和 `last`。
- 2) 如果不包含尾部，就用 `begin` 和 `end`，范围内包含 `begin`，但不包含 `end`。

例子：表示范围的变量

```
heap_sort_data (first_index, last_index);    //包含 last_index
norflash_erase_data (begin_addr, end_addr); //不包含 end_addr
```

如果你要对数学上使用的 “[..]”、 “[..)””、“(..]”、“(..)” 做出命名，可以使用 `begin` 和 `end`，然后再加一个 `flag` 变量，用于区分是哪种范围。

11.2.4 避免错误的单词

命名时要正确地使用英文单词，而且不要把单词拼写错误，但是经常看到误用单词和拼错单词的情况，让人“贻笑大方”。命名时不要使用汉语拼音，因为不便于交流，而且会让读者迷糊，所以还是要学好英语。

例子：相似的单词

```
adapt、adopt、adept
contend、content、context、contest
accept、receive
bring、take、fetch、carry、get
ensure、insure
```

例子：拼写错误

把 “`sector`” 写成了 “`secotr`”，这是笔者现实遭遇到的拼写错误，而且还在多个文件中使用 `#define SECOTR_PART_SIZE 0x200`

11.2.5 避免使用关键字

命名时不要使用编程语言的关键字，因为编译时容易出错，所以不要使用 `for`、`while`、`case`、

break 等名字。虽然在 C 语言中 new 和 delete 不是关键字，但是在 C++ 中 new 和 delete 就是关键字，分别用于创建和销毁对象。所以即便使用 C 编程，也尽量不要使用 C++ 的关键字，说不定哪天你写的函数要移植到 C++ 的环境中呢。

从 Verilog 到 SystemVerilog，new 和 delete 也变成了关键字，而且增加了更多的关键字，所以即便使用 Verilog 编程，也尽量不要使用 SystemVerilog 的关键字。

11.2.6 使用缩写的名字

如果原始名字太长，那么在实际使用时，就可以使用缩写或者创造一些缩写。创造一个缩写名字，通常就是取多个英文单词的首字母，然后合在一起就可以了。对于自创的缩写名字，只要在文档里说清楚，不与别的名字产生冲突，大家没有什么异议，你就可以放心地使用了。

使用缩写名字可以便于编写和阅读代码，因为缩写名字看着更加清晰明了。我们看英文文档时，一般都有一个名字对应的表格，用于说明缩写名字和原始名字的对对应关系，而文档的正文一般都是使用缩写。

例子：这些都是约定俗成的缩写。如果使用原始名字或者中文名字，那就是大傻瓜！

NBA	National Basketball Association (美国职业篮球赛)，CCTV 非得叫它为“美职篮”。
FTL	Flash Translation Layer，FTL 是一种软件中间层，用于将闪存模拟成为虚拟块设备。
SATA	Serial Advanced Technology Attachment，串行高级技术附件。
USB	Universal Serial Bus.
UART	Universal Asynchronous Receiver/Transmitter.
CDMA	Code Division Multiple Access.
LCD	Liquid Crystal Display.

例子：这些是常用的或者自创的缩写

doc	document
str	string
calc	calculation
abm	ahb bus matrix
emi	external memory interface
cgu	clock generation unit

11.2.7 使用前缀和后缀

为了增强名字的可读性，通常对一组相关的名字，使用统一的前缀或后缀，这样可以给名字附带更加有用的信息。根据名字的可读性，选择使用前缀或后缀，通常使用前缀的情况多一些。

我们可以用这些名字做一下类比，例如莫言的小说《丰乳肥臀》中七姐妹来弟、招弟、领弟、想弟、盼弟、念弟、求弟；枝条扭曲的树木变种：龙爪槐、龙爪桑、龙爪榆；海洋哺乳动物：海狮、海豹、海象、海豚、海狗、海牛、海獭。

例子：FTL 作为前缀

```
FTL_ReadOpen
FTL_ReadSector
FTL_WriteOpen
FTL_WriteSector
```

例子：Verilog 代码中 dma 和 uhc 的信号。注意：这里两组信号的顺序也是一样的！

```
wire [1:0]      dma_htrans;
wire [2:0]      dma_hburst;
wire           dma_hwrite;
wire [2:0]      dma_hsize;
```

```

wire [31:0] dma_haddr;
wire [31:0] dma_hwdata;
wire dma_hmastlock;
wire dma_hreadyout;
wire [1:0] dma_hresp;
wire [31:0] dma_hrdata;

wire [1:0] uhc_htrans;
wire [2:0] uhc_hburst;
wire uhc_hwrite;
wire [2:0] uhc_hsize;
wire [31:0] uhc_haddr;
wire [31:0] uhc_hwdata;
wire uhc_hmastlock;
wire uhc_hreadyout;
wire [1:0] uhc_hresp;
wire [31:0] uhc_hrdata;

```

如果命名一个需要度量或者有其他属性的名字，那么最好在名字上带上它的度量或属性，这样更能清晰地说明名字所表示的值。

例子：度量时间的单位有天、小时、分钟、秒、毫秒、微秒，对应的英文为 `day`、`hour`、`minute`、`second`、`ms`、`us`，所以命名一个延迟变量时，可以根据需要选择 `delay_days`、`delay_hours`、`delay_minutes`、`delay_seonds`、`delay_ms`、`delay_us`。

例子：度量距离的单位有公里、米、厘米、毫米、微米、像素、英寸，对应的英文为 `mile`、`meter`、`cm`、`mm`、`um`、`pixel`、`inch`，所以命名一个距离变量时，可以根据需要选择 `distance_miles`、`distance_meters`、`distance_cm`、`distance_mm`、`distance_um`、`distance_pixels`、`disatance_inches`。

例子：加密时数据分为明文数据、密文数据、初始数据、密钥数据，那么就可以命名为 `data_plain`、`data_crypt`、`data_init`、`data_key`。

11.2.8 使用合适的长度

有些人认为名字越长，传递的含义越多，就越有表达力，越让人理解。其实不是，根据不同的环境选择合适长度的名字，会更有表达力。如同你去短期度假时，你带的行李会比较少；当你去长期度假时，你带的行李会比较多^[20]

在小的作用域里，使用短的名字；在大的作用域里，使用长的名字，让名字包含足够的信息，以便于更清晰地表达名字的含义，尤其对于全局变量来说。但是，不要使用太长的名字，因为超过 30 个字符就已经很难书写和阅读了。。

有些书（例如《代码整洁之道》）对变量和函数使用了很长名字，但是英语毕竟不是我们的母语，过长的名字反而会降低代码的可读性，所以我们还是不要使用太长的名字，最好能做到“**Min length, Max information**”。

例如，循环变量使用单字母 `i`、`j`、`k` 等，临时变量使用 2~3 个单词，函数名使用 2~4 个单词，全局变量使用 2~4 个单词。

使用合适长度的名字还便于查找名字的使用情况。例如，使用 `grep` 对所有文件查找 `get_work_days` 的使用情况，如果使用一个短名字，可能会返回很多无关的代码行。

11.2.9 使用不同的格式

通过对不同种类的名字使用不同的命名方法，就可以通过名字区分出名字所属的种类。

例子：对不同种类的名字使用不同的命名方法

```
#define MAX_LOG_NUMBER 6000           //对常量使用以_连接的全部大写
int gLogAccessCount = 0;              //对全局变量使用 g 开头的小驼峰法
int gCurrentNumber = 0;
static int sLastLine = 0;            //对静态变量使用 s 开头的小驼峰法
void LogOpenFile (char *file_name);   //对函数使用大驼峰法，函数名要由“主谓宾”
格式构成
void LogWriteFile (char *line_content); //对函数形参和局部变量使用下划线连接法
char *LogReadFile (int line_number);
void LogCloseFile (void);
```

11.3 命名规则

使用统一的命名规则具有很多好处，可以规范代码的设计，保证代码的一致性，便于团队内部交流合作，有助于提高代码的可读性，让你减少需要考虑的事情，关注代码更重要的事务。

11.3.1 C 语言命名规则示例

表 11-1 所示的是方便可行的 C 语言命名规则示例，可以参照实际情况修改出一个更方便的版本。

表 11-1 C 语言命名规则示例

元 素	命 名 方 法	示 例
类型名	下划线连接 全部大写	typedef struct { int num; char name[40]; char sex; float score; } STUDENT;
函数名	帕斯卡	NandFlashEraseTwoPlane NandFlashWriteTwoPlane NandFlashReadTwoPlane
全局变量	小驼峰 以 g 开头	int gFlashType, gFlashCount;
静态变量	小驼峰 以 s 开头	int sCurrentPosition;
局部变量 函数形参	下划线连接 全部小写	int plane_index, plane_offset;

续表

元 素	命 名 方 法	示 例
宏定义—常量 枚举类型常量	下划线连接 全部大写	<pre>#define SIG_DFL ((_sig_func_ptr)0) /* Default action */ #define SIG_IGN ((_sig_func_ptr)1) /* Ignore action */ #define SIG_ERR ((_sig_func_ptr)-1) /* Error return */ typedef enum GETOPT_ORDERING_T { PERMUTE, RETURN_IN_ORDER, REQUIRE_ORDER } GETOPT_ORDERING_T;</pre>
宏定义—代码	下划线连接 全部大写	<pre>#define BUFFER_INSERT(B, P) { \ (B)->next = (P)->next; \ (B)->prev = (P); \ (P)->next = (B); \ (B)->next->prev = (B); \ }</pre>

11.3.2 Verilog 命名规则示例

下表是方便可行的 Verilog 命名规则示例，可以参照实际情况修改出一个更方便的版本。在作者接触的 Verilog 代码中，命名方法普遍使用下划线连接方法，然后根据需要选择合适长度的名字。

表 11-2 Verilog 命名规则示例

元 素	命 名 方 法	示 例
模块名	下划线连接 全部小写 模块的名字要作为子 模块名字的前缀	<pre>tcu tcu_clk_switch tcu_timer tcu_reg tcu_sync_pulse tcu_sync_data</pre>
实例名	下划线连接 全部小写	<p>使用这样的格式: <module_name>_i 或 u_<module_name></p> <p>当需要多个实例化时:</p> <p>可以使用<module_name>_i0、<module_name>_i1 或者使用 u0_<module_name>、u1_<module_name></p> <p>例如, tcu_i、tcu_clk_switch_i 例如, u_tcu、u_tcu_clk_switch</p>
信号名 变量名	下划线连接 全部小写 相关的信号使用同样 的前缀	<pre>wire enable_v3p3_en; wire enable_vcom_en; wire enable_vddh_en; wire enable_vpos_en; wire enable_vneg_en;</pre>

续表

元 素	命 名 方 法	示 例
函数名 任务名	下划线连接 全部小写 相关的 task 使用同样的前缀	task efuse_read_one_bit; task efuse_write_one_bit; task efuse_check_program; task power_sleep_to_active; task power_active_to_sleep;
参数名	下划线连接 全部大写	module tcu #(parameter IS_OST = 0, COUNT_CHANNEL = 6, WIDTH_CHANNEL = 4, WIDTH_COUNTER = 16, SUPPORT_PWM = 1, SUPPORT_WDT = 1)
宏定义—常量	下划线连接 全部大写	`define TCU_TCCR_SELECT 2:0 `define TCU_TCCR_PRESCALE 5:3 `define TCU_TCCR_ONE_SHOT 6 `define TCU_TCCR_WDT_EN 7 `define TCU_TCCR_PWM_EN 8 `define TCU_TCCR_INITL 9 `define TCU_TCCR_SD 10 `define TCU_TCCR_PAUSE 11
宏定义—代码	下划线连接 全部小写	`define max_val(a,b) ((a) > (b) ? (a) : (b)) `define print_info(x) \$display("Info: the value is %d", x);

在 Verilog 中，命名最复杂的就是要面对各种大量的信号名，通过使用特定的前/后缀和有意义的名字缩写，可以简化名字的命名，从而简化表达式的书写，提高代码的可读性。下表就是一些常用的前/后缀和名字缩写。

表 11-3 Verilog 信号命名常用的前后缀和缩写

前/后缀	说 明	示 例
_n, _b	用于低电平有效的信号	tccr_pwm_en_n
_en, _dis	用于表示信号 enable 和 disable	tccr_pwm_en
_clk	时钟信号	ahb_clk, cpu_clk uhc_hclk, uhc_dev_clk
_rst, _reset	用于表示高电平有效的复位信号	
_rst_n, _reset_n	用于表示低电平有效的复位信号	ahb_reset_n, cpu_reset_n uhc_hreset_n, uhc_dev_reset_n
_addr	address	spi_addr, i2c_addr, uart_addr
_ptr	pointer	fifo_rd_ptr, fofo_wr_ptr
_rd	read	
_wr	write	
_din	datain	spi_din
_dout	dataout	spi_dout
_ctrl	control	

续表

前/后缀	说 明	示 例
_cfg	config	
test_, scan_	DFT 信号	test_mode, test_se, test_si, test_so
R_, r_	表示寄存器，会综合出实际的寄存器	R_int_en, R_config 或者 r_int_en, r_config
Rp_, Rr_, Rx_	跨时钟域寄存器，分别对应 pclk、rclk、xclk	Rp_tchr, Rr_tchr, Rx_tchr
R1_, R2_, R3_ _d1, _d2, _d3	延迟寄存器	R1_din, R2_din, R3_din 或者 din_d1, din_d2, din_d3 用于区分两个不同时钟域的延迟寄存器 Rp1_din, Rp2_din, Rp3_din Rr1_din, Rr2_din, Rr3_din
ahb_	Advanced High Performance Bus	
apb_	Advanced Peripheral Bus	
cpu_	CPU	
mem_	Memory	
ram_	RAM	
dmac_	Direct Memory Access Controller	
cgu_	Clock Generation Unit	
intc_	Interrupt Controller	
rtc_	Real Time Controller	
spi_	Serial Peripheral Interface	
i2c_	Inter Integrated Circuit	
uart_	Universal Asynchronous Receiver/Transmitter	
gpio_	General Purpose IO	
nfc_	Nand Flash Controller	
emi_	External Memmory Interface	
kpc_	Key Press Controller	
usb_	Universal Serial Bus	
uhc_	USB Host Controller	
udc_	USB Device Controller	
sdhc_	SD Host Controller	
sddc_	SD Device Controller	

11.4 名字使用

11.4.1 名字规划

设计一个系统时,要根据功能把系统划分为子系统,然后再进一步划分为模块和子模块,也就是划分为树形结构。

然后,根据这个树形结构,对系统、子系统、模块和子模块规定好正规名字和缩写名字,缩写名字可用作前缀。再规划出目录结构,目录结构和树形结构可以有对应关系,也可以没有。

进一步,在编写文档和代码的时候,需要对很多东西命名,包括常量、变量、函数、宏定义、文件,对于 Verilog 来说,还有模块、任务、信号、事件。命名这些东西时要遵循团队的命名规则,对于同一类名字不要混合使用不同的命名方法,尽量保持名字在穿越模块时保持一致,这样可以便于代码的阅读和修改,便于团队内的交流。

只有做好名字规划,才能做到代码内统一,进一步做到全局范围内统一。

11.4.2 代码内统一

在设计代码时,要尽量做到名字统一。在 C 语言中,函数调用时,传递的变量有可能在多级函数调用中使用,要尽量保持一致,因为如果形参在不同级的函数中有不同的名字,阅读时可能还要绕一下。另外,在 C 语言中,一组函数要对形式参数的名字保持一致。

例子:操作 string 的函数组在形参名字上保持一致

```
char* strcpy(char *dst, const char *src)
char* strncpy(char *dst, const char *src, size_t n);
```

在 Verilog 中,信号经常要在多个模块中连接传递,要尽量保持名字的一致,在自己设计的模块内,一般都有所保证,但是在多人设计的系统中,模块之间信号名有时会存在不一致。所以在设计之初,就要定义好模块之间的连接约定。

11.4.3 全局内统一

在嵌入式设计中,设计一个模块,就要有这些文档,应用文档 (Spec)、设计文档、测试文档、软件文档等;还有这些代码,Verilog 设计代码、Verilog 测试代码、C 语言软件代码。

对于这个模块的寄存器,在这么多的文档和代码中,都要使用规范统一的名字,这样才能方便阅读文档、阅读代码、查找问题。

在作者所经历的一个芯片设计中,Spec 的规范化严重滞后,没有给出正式的寄存器名字 (Register name) 和寄存器位名字 (Register bit name),导致在不同地方使用不同的名字,而且有些软件代码就是直接使用地址值 (没有使用相应的宏定义),导致很多设计脱节。

11.5 SPEC 定义

在设计一个硬件模块时,首先要定义出 Spec,说明模块的功能、特性、寄存器、工作机制、硬件连接、软件操作,然后设计 Verilog 代码,同时做仿真测试,保证 Verilog 代码的正

确性，最后集成到系统内，编写软件做集成测试。

软件操作的方便性，与模块结构的设计合理不合理有关，还与模块对软件接口的设计合理不合理有关。软件接口就是模块的寄存器，软件通过读写寄存器控制这个模块。

如果模块的寄存器设计合理，软件操作就很方便，代码编写和阅读起来也很方便，但是实际情况是，有的人把模块的寄存器设计得很怪，导致编写软件代码时不方便，而且编写出来的软件代码理解起来也很费劲。

那么如何把模块的寄存器设计得更合理呢？首先看一下寄存器位类型和寄存器类型。

11.5.1 寄存器类型

对于 32 位总线的系统来说，硬件模块给软件提供的接口一般是多个 32-bit 的寄存器，但是每个 32-bit 的寄存器可能包含多种数据，它们对应于寄存器位，这些寄存器位的操作实际有不同种类型，如表 11-4 所示。

表 11-4 寄存器位的类型

类 型	说 明
RW	普通的寄存器位，可读可写
RO	只读的寄存器位，写忽略
WO	只写的寄存器位，读返回 0
RWC	读写的寄存器位，写 0 或写 1 清除，最好使用写 1 清除
Reserved	保留位，读出值为 0，写入值忽略

硬件模块应该按照功能组织寄存器，每个寄存器中的寄存器位要尽量保持为同一种类型，寄存器位尽量对齐在 4-bit 或者 8-bit 边界上，尽量不要把不同类型的寄存器位混杂在一起，这样可以方便软件操作，寄存器的类型如表 11-5 所示。

表 11-5 寄存器的类型

功 能 类 型	操 作 类 型	说 明
配置模块	RW	用于配置模块的工作特性
中断使能	RW	用于使能模块的一些中断，1-bit 对应一个中断
中断原始	RO	用于表示模块的中断状态，1 表示发生了对应的中断 原始状态是没有处理过的状态
中断状态	RO	用于表示模块的中断状态，1 表示发生了对应的中断 中断状态是已经处理过的状态，就是中断状态 = 中断原始 & 中断使能
中断清除	WO	用于清除模块的中断状态，向对应的位写 0/1 清除对应的中断 最好使用写 1 清除中断，因为软件把中断状态寄存器读回后，把它直接写入这个寄存器，就可以清除相应的中断
其他状态	RO	用于表示模块其他的状态，比如 FIFO 的空满
模块数据	RO/WO/RW	用于读写模块的数据，根据需要可以选择不同的类型
控制模块	WO	用于控制模块的操作 例如开始、暂停、继续、停止

但是现实情况是，有些设计模块中把不同功能的寄存器混杂在一起，有的寄存器中包含着多种不同的操作类型，例如，将配置、中断使能、中断状态、中断清除等都放在一个寄存器中，这样编写出来的软件代码是很奇怪的。

11.5.2 示例模块

下面是一个假想的数据单向发送/接收模块，每次发送或接收 32-bit 数据，支持奇偶校验，支持错误检查。为了获得更好的传送效率，使用一个 32x32-bit 的 FIFO，FIFO 的中断触发级别可以设置为 1、8、16 或者 24。

下面就是此假想模块的寄存器定义，具有这些特点：

- 1) 按照功能组织每个寄存器，每个寄存器中的所有寄存器位都是同一种类型，寄存器位都对齐在 4-bit 边界上，便于软件操作。
- 2) 规范的寄存器和寄存器位的名字，这些英文名字便于在所有文档和代码中统一使用。
- 3) 注意： $INT_MASK_FLAG_REG = INT_RAW_FLAG_REG \& INT_EN_REG$ ，软件通常使用 $INT_MASK_FLAG_REG$ 。

表 11-6 示例模块的寄存器

Name	Descriptions	R/W	Initial Value	Address Offset	Access Size
CTRL_REG	控制寄存器	RW	0x00000000	0x00	32-bit
CFG_REG	配置寄存器	RW	0x00000000	0x04	32-bit
INT_EN_REG_REG	中断使能寄存器	RW	0x00000000	0x08	32-bit
INT_RAW_FLAG_REG	中断原始寄存器	RO	0x00000000	0x0C	32-bit
INT_MASK_FLAG_REG	中断状态寄存器	RO	0x00000000	0x10	32-bit
INT_CLEAR_REG	中断清除寄存器	WO	0x00000000	0x14	32-bit
STAT_REG	状态寄存器	RO	0x00000001	0x18	32-bit
DATA_REG	数据寄存器	RO/WO	0x00000000	0x1C	32-bit

CTRL_REG

Bit	Name	Description	R/W
31:1	Reserved		
0	START	0: 停止操作 1: 开始操作	WO

CFG_REG

Bit	Name	Description	R/W
31:24	Reserved		
23:16	TIMEOUT_NUM	0: 禁止超时检查 1 ~ 255: 允许超时检查，检查时间= TIMEOUT_NUM * 128 Cycles	RW
15:10	Reserved		

续表

Bit	Name	Description	R/W
9:8	FIFO_TRIG_LEVEL	FIFO 触发级别 对发送来说, 如果 FIFO 中数据的个数小于下面的值, 就产生中断 对接收来说, 如果 FIFO 中数据的个数不小于下面的值, 就产生中断 00: 1-byte 01: 8-byte 10: 16-byte 11: 24-byte	RW
7:6	Reserved		
5:4	PARITY_CHECK	奇偶校验类型 00: 不使用校验 01: 使用 Odd 10: 使用 Even 11: 保留	RW
3:1	Reserved		
0	DIRECTION	0: 表示发送 (Send) 1: 表示接收 (Receive)	RW

INT_EN_REG_REG

Bit	Name	Description	R/W
31:2	Reserved		
1	DATA_ERROR	0: 禁止此中断 1: 使能此中断	RW
0	FIFO_TRIG	0: 禁止此中断 1: 使能此中断	RW

INT_RAW_FLAG_REG

Bit	Name	Description	R/W
31:2	Reserved		
1	DATA_ERROR	0: 此中断没发生 1: 此中断已发生	RO
0	FIFO_TRIG	0: 此中断没发生 1: 此中断已发生	RO

INT_MASK_FLAG_REG

Bit	Name	Description	R/W
31:2	Reserved		
1	DATA_ERROR	0: 此中断没发生 1: 此中断已发生	RO
0	FIFO_TRIG	0: 此中断没发生 1: 此中断已发生	RO

INT_CLEAR_REG

Bit	Name	Description	R/W
31:2	Reserved		
1	DATA_ERROR	写 0 忽略，写 1 清除	WO
0	FIFO_TRIG	写 0 忽略，写 1 清除	WO

STAT_REG

Bit	Name	Description	R/W
31:6	Reserved		
5:4	ERROR_TYPE	错误类型 00: 没有错误 01: 奇偶校验错误 10: 超时错误 11: 保留位	RO
3:1	Reserved		
0	READY	0: BUSY 1: READY	RO

DATA_REG

Bit	Name	Description	R/W
31:0	VALUE	当模块配置为发送时，此寄存器为 WO 当模块配置为接收时，此寄存器为 RO	WO RO

让代码更容易读

当我们遇到不喜欢的代码时常常这么想：“这人的代码太差劲了！”这些代码很差劲，可能的原因就是变量使用不当，表达式太复杂，控制流太复杂，或者函数设计得不好。研究表明，我们大多数人通常只能同时考虑3~4件事情，既然如此，我们怎么能在大脑中考虑那么多复杂的东西呢？

本部分探讨的是让代码更容易阅读的方法，包含精心用变量、表达式易读、控制流易读、定义好函数，还包含消除警告。

第 12 章

消除警告

一个男子在路边一根接着一根地抽烟。一个女子走过来对他说：“嘿，你不知道你是在慢性自杀吗？注意看看烟盒上的警告信息。”男子悠然自得地又吸了一口说：“没关系，我是个程序员。”“嗯？这和你是程序员有什么关系？”“我们一点儿也不在乎警告（Warning），我们只在乎错误（Error）。”

这其实是很多程序员的心态，他们对警告毫不在乎，认为警告对程序运行不会有任何影响，但是他们不知道忽略警告其实会产生很多问题。

12.1 不可忽视

很多人根本就不在乎警告，认为警告不会产生任何问题，不会影响程序的正确执行。有些人嫌这些警告很麻烦，所以干脆就在编译选项里面把产生警告的选项关闭，不让编译器产生任何警告。

事实上，任何警告都不可以忽视，一是因为有些警告其实就是 Bug，是编程上的失误或者疏漏；二是因为当错误与警告混杂在一起时，在众多的警告中找到一个错误也要浪费一些时间；三是因为当大量的警告产生时，就会显得代码写得很不认真、很不干净、很不专业；四是因为当大量的警告报告出来，让人看着闹心，谁还会有耐心检查它们呢。

例子：main.c 和 mul.c，编译器会报告没有 mul() 的函数原型

```
//Place in main.c
#include <stdio.h>
//double mul (double x, double y);
main (void)
{
    printf ("%f\n", mul (5, 6));
    return 0;
}
//Place in mul.c
double mul (double x, double y)
{
    return x + y;
}
```

因为没有 mul() 的函数原型，编译器就会直接把 5 和 6 按照 int 传递过去，而 mul() 还按照 double 解释这两个参数，同时编译器还认为 mul() 的返回值为 int，所以导致计算完全错误。如何改正这个错误呢？就是把 mul() 的函数原型写到 main.c 中。

例子：错误，因为误用 “=”，实际要用 “==”

```
int find_key (int ar[], int count, int key)
{
    int i;
    for (i = 0; i < count; i++)
```



```

    if (ar[i] = key) return i;
    return -1;
}

```

编译器会把这个警告报告出来，当把“=”改为“==”，程序就会正常运行。

例子：x1()是正确的，x2()是错误的，因为函数返回的是局部变量的地址

```

char *x1 (void)
{
    char *x = "abcdef";
    return x;
}

char *x2 (void)
{
    char x[] = "abcdef";
    return x;
}

```

编译器会把 x2() 的警告报告出来，如下：

```
x2.c:9: warning: function returns address of local variable
```

例子：类型不匹配

```
void * const ram_BufPtr[8] = {0, 0, 0, 0, 0x288000, 0x288400, 0x288800, 0x288C00};
```

编译器会报告类型不匹配，应该修改为：

```
void * const ram_BufPtr[8] = {(void *)0, (void *)0, (void *)0, (void *)0,
    (void *)0x288000, (void *)0x288400, (void *)0x288800, (void *)0x288C00};
```

下面是编译一个实际程序时报告出来的警告的一部分，足足有 389 个，编译时就会看着警告哗哗地显示出来，是不是很吓人呢？这个程序只有 38 个 C 语言文件和 40 个头文件，规模并不是很大，要是真心花半天时间，也就能把这些警告消除掉。可就是这样，设计者还是让这些警告存在了 3 年，随着程序规模的增大，警告也越来越多，但是设计者根本就毫不在意。最后在项目主管的强烈要求下，设计者才把它们消除掉。

例子：大量警告中的一部分

```

xdifopt.c:973: warning: pointer targets in passing argument 2 of
'ram_GetCtrl' differ in signedness
xdifopt.c:1818: warning: assignment makes pointer from integer without a cast
xdifopt.c:1872: warning: passing argument 1 of 'SYS_AtomModify' makes
integer from pointer without a cast
xdifopt.c:2032: warning: assignment makes pointer from integer without a cast
xdifopt.c:2259: warning: passing argument 2 of 'SYS_MemoryCopy' makes
pointer from integer without a cast
xdifopt.c:2266: warning: assignment makes pointer from integer without a cast
crypt_api.c:419: warning: unused variable 'curSts'
des_app.c:43: warning: implicit declaration of function 'BSP_RegisterISR'
sm2_app.c:101: warning: passing argument 1 of 'mem_cpy' discards qualifiers
from pointer target type
cuaa_api.c:65: warning: passing argument 1 of 'memcpy' discards qualifiers
from pointer target type
instruction.c:162: warning: passing argument 2 of 'Char2char_Little'
discards qualifiers from pointer target type
instruction.c:177: warning: passing argument 1 of 'mem_cpy' discards
qualifiers from pointer target type
instruction.c:177: warning: passing argument 2 of 'mem_cpy' discards
qualifiers from pointer target type
instruction.c:202: warning: unused variable 'Pri_Key'
instruction.c:200: warning: unused variable 'Lc'

```

```

instruction.c:342: warning: unused variable 'Hash_data'
instruction.c:35: warning: implicit declaration of function
'sm2_generation'
instruction.c:43: warning: implicit declaration of function 'sm2_encrypt'
instruction.c:48: warning: implicit declaration of function 'sm2_decrypt'
instruction.c:53: warning: implicit declaration of function 'sm2_sign'
process_instr.c:58: warning: implicit declaration of function 'mem_cpy'
process_instr.c:82: warning: passing argument 2 of 'SchUpdate' from
incompatible pointer type
process_instr.c:86: warning: passing argument 2 of 'SchUpdate' from
incompatible pointer type
process_instr.c:99: warning: passing argument 2 of 'SchUpdate' from
incompatible pointer type
process_instr.c:103: warning: passing argument 2 of 'SchUpdate' from
incompatible pointer type
process_instr.c:52: warning: unused variable 'ilen'
process_instr.c:52: warning: unused variable 'i'

```

上面是 C 语言程序的例子，其实很多人写的 Verilog 程序也存在很多警告，其中一些警告其实就是 Bug，会导致仿真或综合失败，例如敏感列表不全，例如端口宽度不匹配，例如生成 Latch。

所以只要编译器报告出警告，就说明代码可能存在问题，就要有吹毛求疵的精神，消掉这些“眼中钉、肉中刺”一样的警告，从而减少代码的调试时间。

12.2 警告类型

C 语言程序编译时，常见如下的警告类型，而且这些警告必须要修正。

- 1) 赋值时类型不一致，常出现在不同类型的指针赋值上。解决办法是尽量保持类型一致，或者使用强制类型转换。
- 2) 使用没有初值的局部变量。解决办法是检查代码，确保局部变量有初始值。
- 3) 根本没有使用的局部变量。解决办法是删掉这些没有使用的局部变量。
- 4) 宏定义重复定义。解决办法就是只保留一个宏定义，别的宏定义必须删掉。如果重复的宏定义都为同一个值，可能不会出现错误，但是事实上有时重复的宏定义分布在不同的头文件中，它们被定义成不同的值。
- 5) 没有函数原型。解决办法就是保证每个函数都有原型，如果是用户自定义的函数，没有相应的头文件，就要编写自己的头文件。

例子：这个小小的程序就会产生 8 个警告

```

#include <stdio.h>
#define AAA 0x5578
#define AAA 0x5579
char *str = "12345678";
int main (void)
{
    char c = 0x1234;
    int x, y, z;
    int a, b;
    x = 12;
    y = 34;
    if (x = y)
        return 1;
    a == x;
}

```

```

    b = str;
    return 0;
}
//编译输出的警告
x3.c:3:1: warning: "AAA" redefined
x3.c:2:1: warning: this is the location of the previous definition
x3.c: In function main:
x3.c:7: warning: overflow in implicit constant conversion
x3.c:12: warning: suggest parentheses around assignment used as truth value
x3.c:14: warning: statement with no effect
x3.c:15: warning: assignment makes integer from pointer without a cast
x3.c:8: warning: unused variable z
x3.c:7: warning: unused variable c

```

Verilog 程序编译时，常见如下的警告类型，而且这些警告必须修正。

- 1) 敏感列表不全，导致前后仿真不一致。解决办法时尽量使用 `always @(*)`。
- 2) 模块实例化时端口信号的位数不匹配，导致编译失败。解决办法是保证端口位数的匹配。
- 3) `always` 块中 `case` 或 `if` 语句分支不全，导致生成 Latch，而 Latch 在做 DFT 时是不可测的。解决办法是补齐 `case` 或 `if` 语句的分支，或在 `always` 开始处给要修改的变量赋初值。

12.3 打开警告

编译器都支持关闭警告，或者打开警告，而且支持打开警告的级别。编译时，应该尽量选择级别最高的警告选项。

虽然编译器可以检测并报告代码中的常见缺陷，但是编译器并不能发现那些人为制造的缺陷，所以在编写代码时还是要小心谨慎，要注意符合编码规范，避免编写过于简单和极为复杂的代码。

12.3.1 对于 C 语言

如果你在用 GCC，那么可以使用如下的编译选项，以打开最高的警告级别：

```
gcc -W -Wall -pedantic -ansi -std=c99 -o hello hello.c
```

除了编译器的检查，还有一些静态分析工具用于分析 C 语言程序，例如 `pc-lint`、`splint`。`splint` 可以检查发现以下问题：变量声明了但未使用、变量类型不匹配、变量在使用前未定义、不可达代码、死循环、数组越界、内存泄漏等。

```

例子: splint_msg.c
//splint_msg.c
int func_splint_msg1 (void)
{
    int a;
    return 0;
}
int func_splint_msg2 (void)
{
    int* a = (int*)malloc (sizeof(int));
    a = NULL;
    return 0;
}

```

例子：运行 `splint splint_msg.c` 之后，我们来看输出的告警信息：

```
splint_msg.c: (in function func_splint_msg1)
splint_msg.c:4:6: Variable a declared but not used
  A variable is declared but never used. Use /*@unused@*/ in front of
  declaration to suppress message. (Use -varuse to inhibit warning)
splint_msg.c: (in function func_splint_msg2)
splint_msg.c:10:2: Fresh storage a (type int *) not released before assignment:
a = NULL
  A memory leak has been detected. Storage allocated locally is not released
  before the last reference to it is lost. (Use -mustfreefresh to inhibit
  warning)
splint_msg.c:9:37: Fresh storage a created
Finished checking --- 2 code warnings
```

不管 `pc-lint`、`splint` 等静态程序分析工具的功能多么强大，它们对程序的检查也有疏漏的地方，工具的使用并不能提高我们的编程能力，我们更应该通过它们知道各种编码错误和代码隐患，凭借积累的编码知识把错误和隐患消灭在萌芽状态。

12.3.2 对于 Verilog

仿真编译时，打开 `vcs` 或 `ncverilog` 的 `lint` 检查。检查编译的输出，错误一定要修正，但是那些警告也要修正。

综合编译时，检查综合工具的编译输出，检查是否生成了 `Latch`，检查 `always` 敏感列表中的信号是否列全。

另外，还有一些静态检查工具，例如 `Leda` 和 `Spyglass`，它们可以做超强的静态分析，还可以做跨时钟域的检查。

12.3.3 对于 Perl

作者经常使用 `Perl` 编写一些脚本，在脚本开头会固定放置下面这两行，以便于产生警告，检查程序中的错误。

```
use strict;
use warning;
```

例子：使用 `use strict` 和 `use warning`，就会报告出错

```
$string = "hello world";
@array = qw(ABC DEF);
%hash = (A=>1, B=>2);
```

对上面的代码使用这两个 `use` 之后，就要添加 `my`，更改如下。

```
my $string = "hello world";
my @array = qw(ABC DEF);
my %hash = (A=>1, B=>2);
```

第 13 章

精心用变量

对于变量的草率使用会降低代码的可读性。因为，变量的名字不好，就会降低代码的可读性；变量的数目越多，就越难跟踪他们的使用情况；变量的作用域越大，就需要跟踪它们的使用越久；变量的改变越频繁，就越难以跟踪它们的当前值。

13.1 修改变量的名字

变量的名字对于代码的可读性有很大影响，如果变量用了好名字，那么代码就清晰易懂、一目了然，而如果变量用了不好的名字，那么代码就是符号和数字的大杂烩，干巴巴的，毫无趣味。下面的函数用于计算数字 n 之前的素数个数，可以对比一下这里三个局部变量的名字对代码可读性的影响。

例子：变量使用不好的名字

```
int count_prime (int n)
{
    int i, j, k, x;
    int *a = malloc (sizeof(int) * n);
    int m = 0;
    a[m++] = 2;
    x = 0;
    for (i = 3; i <= n; i++)
    {
        k = (int)sqrt (i);
        while (a[x] <= k && x < m)
            x++;
        for (j = 0; j < x; j++)
            if (i % a[j] == 0)
                break;
        if (j == x)
            a[m++] = i;
    }
    free (a);
    return m;
}
```

例子：变量使用好的名字

```
int count_prime (int number)
{
    int i, j, k, stop;
    int *prime = malloc (sizeof(int) * number);
    int count = 0;
    prime[count++] = 2;
    stop = 0;
    for (i = 3; i <= number; i++)
```

```
{
    k = (int)sqrt (i);
    while (prime[stop] <= k && stop < count)
        stop++;
    for (j = 0; j < stop; j++)
        if (i % prime[j] == 0)
            break;
    if (j == stop)
        prime[count++] = i;
}
free (prime);
return count;
}
```

13.2 进行变量初始化

如果没有对变量做初始化，就去读取并使用变量的值，会产生运行错误。下面就是一些常见的变量初始化方面的错误。

- 1) 从来就没有对变量初始化，这时变量的值就是启动程序时变量所在内存区域的值，通常是不定值。
- 2) 变量值已经过期，就是变量在某个地方曾经被初始化，但是该值已经不再有效，需要重新初始化。
- 3) 只初始化了变量的一部分，而其他部分还是不定值。例如，对于一个结构类型的变量，没有对所有的成员做初始化。

例子：错误的初始化

```
i = 0;
j = 0;
while (i < 10) {
    while (j < 20)
        ar[i][j] = i * 20 + j;
}
//需要改为如下
i = 0;
while (i < 10) {
    j = 0;
    while (j < 20)
        ar[i][j] = i * 20 + j;
}
```

下面是对变量初始化的一些建议。

- 1) 对于没有进行初始化就使用的临时变量，编译器会报告出警告，所以要检查并消除所有的警告。
- 2) 定义变量时就对变量做初始化，这是一种最简单的防范初始化错误的方法，但是会导致过度的初始化代码。
- 3) 推迟进行变量的初始化，只在靠近变量第一次使用时的位置进行初始化。这种方法会拉长定义变量和使用变量之间的距离。
- 4) 最好的方法是，在靠近变量第一次使用时的位置定义并初始化变量。
- 5) 有些变量不用初始化，对它们初始化就是浪费代码。第一种情况是，每次使用前肯定有初始化，例如一些全局的数组或结构；第二种情况是，变量在 if、switch 语句的所

有分支中被完全改写。

- 6) 对于代码中没有初始化的全局变量和静态局部变量，它们会被分配到 BSS 区域，操作系统在装载程序运行时，会把 BSS 区域全部初始化为 0。程序可以利用这个特性，如果它们的初始值就是 0，那么就不用对它们做初始化。

例如：不用初始化的情况

```
for (i = 0; i <100; i++)          //for 语句本身就包含了对 i 的初始化
if (a > b) x = a; else x = b;    //不需要对 x 初始化，因为 x 在 if/else 分支都被修改
switch (ch){                    //不需要对 x 初始化，因为 switch 包含 default，而且 x 在所有的分支都被
修改
case 'A': x = 12; break;
case 'B': x = 34; break;
case 'C': x = 56; break;
default: x = 78; break;
}
```

例子：需要初始化的情况

```
if (a > b) x = a; //需要对 x 初始化，因为 x 只在 if 分支被修改
//需要改为如下
x = b;
if (a > b) x = a;
```

例子：需要初始化的情况

```
switch (ch){                    //需要对 x 初始化，因为 switch 没有包含 default
case 'A': x = 12; break;
case 'B': x = 34; break;
case 'C': x = 56; break;
}
//需要改为如下
x = 78;
switch (ch){
case 'A': x = 12; break;
case 'B': x = 34; break;
case 'C': x = 56; break;
}
```

13.3 减少变量的个数

代码中通过引入一些中间变量，拆分大的表达式或者提取公共表达式，就可以提高代码的可读性，因为它们有助于简化复杂的表达式。但是在这里，我们的兴趣是减少不能改进可读性的变量，当消除这种变量后，代码变得更加精炼，而且更加易于理解。下面就是代码中可以去掉的变量。

- 1) 代码中会残留一些根本就没有使用的全局变量、局部变量和函数的参数，尤其是没有使用的局部变量，编译器还会对它们产生编译警告。对于这些无用的变量，可以不用做太多的考虑就把它删掉。
- 2) 没有价值的中间变量，因为它没有用于拆分任何复杂的表达式，也没有做更多的解释说明，而且只用过一次，所以它没有任何价值。
- 3) 有些变量只是用于保存临时结果或者用作标志，通过调整代码，完全可以把这样的变量删除掉，同时代码会变得更紧凑。

例子：x 只用一次，是一个没有价值的中间变量

```
x = sin (PI / 4);
value = x;
//直接就把 x 删掉
value = sin (PI / 4);
```

例子：使用 found_flag，用作发现的标志

```
struct list *delete_name (struct list *head, char *expect_name)
{
    int found_flag = 0;
    struct list *prev = NULL;
    struct list *node = head;
    while (node != NULL) {
        if (strcmp (node->name, expect_name) == 0) {
            found_flag = 1;
            break;
        }
        prev = node;
        node = node->next;
    }
    if (found_flag) {
        if (prev == NULL)
            head = node->next;
        else
            prev->next = node->next;
        free (node);
    }
    return head;
}
```

例子：删掉 found_flag，发现后直接处理

```
struct list *delete_name (struct list *head, char *expect_name)
{
    struct list *prev = NULL;
    struct list *node = head;
    while (node != NULL) {
        if (strcmp (node->name, expect_name) == 0) {
            if (prev == NULL)
                head = node->next;
            else
                prev->next = node->next;
            free (node);
            break;
        }
        prev = node;
        node = node->next;
    }
    return head;
}
```

这里通过把处理 node 的代码移到循环内，发现后就立即处理，这样就把 found_flag 删掉，而且简化了代码。

13.4 缩小变量作用域

变量的作用域是指变量在程序内的可见和可引用的范围。全局变量是在整个程序范围内，静态全局变量是在文件范围内，局部变量的作用域只在函数内或代码块内。

我们都知道要减少全局变量的使用，因为全局变量的作用域是整个程序范围，很难跟踪这些全局变量的使用情况，全局变量有“名字空间污染”的危险，就是因为全局变量定义得太多了，结果与局部变量发生了重名。当“名字空间污染”发生时，就会存在这种导致程序错误的情况：本来要使用或修改这个变量，却不小心使用或修改了另一个同名的变量。所以要尽量少用全局变量^[20]。

很多编程语言都提供了多重作用域，包括全局、模块、类、函数、语句块等作用域。通常越严格的访问控制越好，就是让变量对尽量少的代码可见，为什么要这么做呢？因为我们的大脑在同一时刻只能暂存几个临时记忆块，而变量的作用域越大，变量的个数越多，跟踪它们就越费劲，看代码时就不得不跳来跳去，而如果让变量对尽量少的代码可见，就能够有效地减少需要同时考虑的变量个数。

如何减小变量的作用域呢？尽量少用全局变量，在代码块内定义临时变量，把大文件拆分成小文件，把大模块拆分成小模块，把大函数拆分成小函数，把大的类拆分成小的类。

另外，对于局部变量，尽量要让它只有一个用途，不要让同一个变量在两个不同的位置具有不同的用途，因为让变量有多个用途就是在加大变量的作用域。通常这种情况是由变量的名字引起的，这样的变量通常是一个很简单的名字，例如 `x`、`k`、`foo`、`tmp` 等，我们可以通过重新命名并定义新的变量来解决这个问题。有些人觉得能少用一个变量，就少用一个变量，这样可以节省内存空间，但是现在已经不是 20 世纪 80 年代了，不要为了节省几个变量，就降低代码的可读性。

对于 C 语言，如果全局变量只在一个文件范围内使用，那么可以把它定义成静态全局变量；如果全局变量只在一个函数内使用，那么就可以把它定义成静态局部变量；如果变量只在语句块内使用，就可以把它定义在语句块内。

C 语言有两个标准 C89 和 C99，C89 要求所有的临时变量定义放在函数或语句块的顶端，所以在 C89 的代码里，对于稍微复杂的函数，就要在函数头部声明很多的局部变量。这个要求很令人遗憾，因为当读者看到这些变量时，就要马上考虑这些变量的使用，即使有些变量要过很久之后才会用到。

C99 和 C++ 去掉了这个要求，可以在代码中随时声明变量，也就是可以把代码和变量声明混合在一起，因为混合声明可以缩小变量的作用域，对于提高代码的可读性确实有很大的帮助。对于 Verilog 和 Perl，从一开始就支持混合声明。

例子：很多的局部变量定义在函数头部

```
int load_elf_file (const char *filename, int argc, int argv, int *retcode)
{
    int          elf_exec_fileno;
    int          i, retval;
    uint         k, elf_bss, elf_brk, elf_entry;
    uint         start_code, end_code, end_data;
    struct elf_phdr *elf_ppnt, *elf_phdata;
    struct elfhdr elf_ex;
    struct file   *file;
    .....
}
```

例子：C89 支持的局部变量定义，`temp` 可以定义在语句块内

```
void reverse (int array[], size_t size)
{
    int i, j;
```

```

for (i = 0, j = size - 1; i < j; i++, j--)
{
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

```

例子：C99 支持 i 和 j 这样定义，在 for 中定义 i 和 j

```

void reverse (int array[], size_t size)
{
    for (int i = 0, j = size - 1; i < j; i++, j--)
    {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

```

例子：C 语言的标准函数 strtok(), 使用混合声明，而且 last 是静态局部变量

```

char *strtok(char *s, const char *delim)
{
    static char *last;
    if (s == NULL && (s = last) == NULL)
        return NULL;
    char *tok;
    char *ucdelim;
    char *spanp;
    int c, sc;
    int found = 0;
    .....
}

```

对于 Verilog 模块，应该尽量使用混合声明，就是尽量在使用处的前面定义 wire 和 reg，这是因为模块有时很大，从数百行到数千行，这时如果在模块的头部定义 wire 和 reg，就使得这些 wire 和 reg 的作用域变得非常大，从而使得阅读修改代码非常不方便。另外，在定义一个 wire 时，还可以直接就把对应的表达式赋值给它，这样还可以减少一条 assign 语句。

例子：Verilog 模块混合声明

```

wire sram_wr_sync = sram_cs | sram_wr | sram_eb ;
reg r_sram_wr_sync;
always @(negedge sram_clk or negedge sram_rst_n)
begin
    if (~sram_rst_n)
        r_sram_wr_sync <= #DLY 'h1;
    else if (soft_rst_end)
        r_sram_wr_sync <= #DLY 'h1;
    else
        r_sram_wr_sync <= #DLY sram_wr_sync;
end
wire sram_wr_neg_pls = ~sram_wr_sync & r_sram_wr_sync;

//write control signal
wire sfr_ctrl1_wr = sram_wr_neg_pls & (sram_addr == ADR_CTRL1);
wire sfr_key_wr = sram_wr_neg_pls & (sram_addr == ADR_KEY);
wire sfr_plain_wr = sram_wr_neg_pls & (sram_addr == ADR_PLAIN);

//read control signal
wire sfr_ctrl1_rd = sram_wr & (sram_addr == ADR_CTRL1);

```

```
wire sfr_key_rd  = sram_wr & (sram_addr == ADR_KEY);
wire sfr_crypt_rd = sram_wr & (sram_addr == ADR_CRYPT);
```

也许有人会问，是不是可以像拆分 C 语言函数一样，把大模块拆分成小模块，从而减小变量的个数和作用域？这种方法是可行的，但是大模块有两个特点，一是信号线多，二是逻辑复杂，当拆分成小模块时，逻辑的复杂性并不会降低，反过来连接众多的小模块也是一件很麻烦的事情，信号线需要在模块之间穿来穿去，所以有时候拆分成小模块就有些得不偿失。

13.5 减少变量的操作

通过减少对变量的操作次数，可以大大减少代码量，同时缩小代码的作用域，提高代码的可读性。例如，下面的代码是从键盘输入一个数，如果是 EOF，就返回-2；如果是 Q 或 q，就返回-1；如果是 0~9，就返回对应的整数值；如果是其他，就一直等待继续输入。

例子：ch=getchar()有两处

```
int input_option (void)
{
    int ch;
    ch = getchar ();
    while (ch != EOF) {
        if (ch == 'Q' || ch == 'q')
            return -1;
        else if (ch >= '0' && ch <= '9')
            return (ch - '0');
        ch = getchar ();
    }
    return -2;
}
```

这里有些啰嗦，因为 ch=getchar()有两处，完全可以改成一处，对应的代码如下。

例子：ch=getchar()只有一处

```
int input_option (void)
{
    while (1) {
        int ch = getchar ();
        if (ch == EOF)
            return -2;
        else if (ch == 'Q' || ch == 'q')
            return -1;
        else if (ch >= '0' && ch <= '9')
            return (ch - '0');
    }
}
```

作者见过很多类似的代码，在进入 while 循环之前做了一些准备工作，然后在 while 的尾部又把这些准备工作重复了一遍，有些教科书上也写着这样的代码，真是误人子弟。这里的 ch=getchar()很简单，看起来没有什么问题，但是设想一下，如果按同样的方法实现其他复杂的操作，相当于把 ch=getchar()换成多行代码，在 while 循环的外边和在 while 循环的尾部，就存在两处大的重复，你说烦不烦呢？

所以通过减少对变量的操作次数，可以大大地减少代码量，同时缩小代码的作用域，提高代码的可读性。当你注重小的结构调整，可以带来小的收益，可以积少成多，集腋成裘，进一步，就可以做出大的结构调整，从而获得更大的收益。

第 14 章

表达式易读

每人手里都会有几张银行卡，上面的卡号是不是一般都是 4 位一组？这样我们在电话银行或电脑里输入时，或读写时都会很方便。但是有的银行卡号就很变态，是“6 位+13 位”表示，用起来一点都不方便，经常搞错。这是一个很小的细节，银行难道就不注意吗？

对于表达式也一样，代码中的表达式越长，就越难以理解。所以为了便于理解表达式，就不要构造太长的表达式，就要尽量把不易理解的长表达式拆分成容易理解的短表达式，或者换一些方法重写表达式。

14.1 糟糕的表达式

作者曾经在某些代码中见过一些很长的表达式，下面是最长的一个表达式。这样的代码在可读性、可理解性、可维护性上都很差劲，作者深深地被这样的代码“所折服”。

例子：Verilog 的超长表达式，都放在一行内，共 1061 个字符

```
wire wdat_oper_byte_last_but1_lch = ((o_cpu_dd_acc == 1'b1 &&
(i_nvm_toggle_mode_ena == 1'b1 || i_nvm_sync_mode_ena == 1'b1)) ||
((nvm_wr_rd_extra_2b_dec == 1'b0 || nvm_wparity_ena == 1'b1) &&
(((rw_data_cnt_eq4 == 1'b1 && i_l_nls_cnt_eq0 == 1'b0) || (rw_data_cnt_eq6 ==
1'b1 && i_l_nls_cnt_eq0 == 1'b1)) && i_nvm_sync_mode_ena == 1'b1) ||
(((nls_cnt_all_eq0 == 1'b1 && rw_data_cnt_eq4 == 1'b1) || (nls_cnt_all_eq0 ==
1'b0 && rw_data_cnt_eq2 == 1'b1)) && i_nvm_toggle_mode_ena == 1'b1) ||
(rw_data_cnt_eq0 == 1'b1 && i_nvm_toggle_mode_ena == 1'b0 && i_nvm_sync_mode_ena
== 1'b0)) || (nvm_wr_rd_extra_2b_dec == 1'b1 && nvm_wparity_ena == 1'b0 &&
(((rw_data_cnt_eq2 == 1'b1 && i_l_nls_cnt_eq0 == 1'b0) || (rw_data_cnt_eq4 ==
1'b1 && i_l_nls_cnt_eq0 == 1'b1)) && i_nvm_sync_mode_ena == 1'b1) ||
(((nls_cnt_all_eq0 == 1'b1 && rw_data_cnt_eq2 == 1'b1) || (nls_cnt_all_eq0 ==
1'b0 && rw_data_cnt_eq0 == 1'b1)) && i_nvm_toggle_mode_ena == 1'b1) ||
(rw_data_cnt_eq7fe == 1'b1 && i_nvm_toggle_mode_ena == 1'b0 &&
i_nvm_sync_mode_ena == 1'b0)))) && i_two_ch == 1'b0;
```

14.2 使用中间变量

有些人的代码可读性很差，或者是因为臃肿庞大，或者是因为短小紧凑。有人会问，代码臃肿庞大肯定不好，难道代码短小紧凑还不好吗？一般来说短小紧凑的代码是很好，但是要有一个度，如果过于强调代码的短小紧凑，那么实际代码的表达力可能很差，很难把代码的设计意图表现出来。所以说，如果想让代码具有很好的可读性，就要想办法提高代码的表达力。

为了提高代码的表达力，可以根据需要引入不同用途的中间变量，就是把执行过程打散

成一系列具有良好命名的中间值。使用中间变量的方法可按如下考虑^[20]。

- 1) 拆分大的表达式。如果大的表达式难以理解，就要考虑把大的表达式拆分为多个子表达式。为了清晰地表示每个子表达式的含义，就要引入额外的临时变量，每个变量的名字都要能充分地表示出子表达式的含义。这样的变量可以称为“解释变量”，可以提高代码的可读性，让代码文档化。我们可以对一个超长表达式引入多个解释变量，每个解释变量对应一个子表达式，当超长表达式拆分成多个子表达式后，就可以逐步理解每个子表达式，这时代码的复杂度就会大大地降低。
- 2) 提取公共表达式。如果代码中有多处使用相同的表达式，就要提取公共表达式，即使你可以很清楚地看出这种表达式的含义，还是要尽量地提取。对于提取出的公共表达式，也要引入额外的临时变量，这个变量的名字要能充分地表示出公共表达式的含义。这种变量可以称为“总结变量”，因为使用较短且清晰的名字代替公共表达式，可以更清楚地表达代码的设计意图。
- 3) 提取公共的子表达式。当多个表达式之间有公共的子表达式，那么同样要把这个公共的子表达式提取出来，增加一个中间变量，然后这些表达式都使用这个中间变量。这样有个好处就是提高代码的可读性，减少代码的维护量。

使用中间变量还有一个好处就是可以提高程序的运行速度，减少代码维护量。如果原来的表达式有好几处使用，那么在程序运行时，这些地方都要计算这个表达式的值，而且如果要修改这个表达式，这几处都要修改，难免会发生遗漏。但是当使用中间变量后，在程序运行时，只需要在这一处计算这个表达式的值，其他位置因为是使用中间变量，可以直接使用它，无须再计算；另外如果要修改这个表达式，只须修改这一处即可。

例子：使用中间变量

```
FLASH_WriteCMD (FLASH_ADDR_MASK | FLASH_ADDR_LATCH_CNT | (col_addr &
0xFFFF));
//====After using address
address = (col_addr & 0xFFFF);
FLASH_WriteCMD (FLASH_ADDR_MASK | FLASH_ADDR_LATCH_CNT | address);
```

这里使用了 1 个中间变量：address。

例子：使用中间变量

```
for (i = idle_index[lkpID] / 32; i < IDLE_GROUP_AMOUNT; i++) {
    x = idle_block[lkpID][i];
    if (x == 0) continue;
    for (j = (i == idle_index[lkpID] / 32 ? idle_index[lkpID] % 32 : 0); j
< 32; j++)
        if (x & (1 << j)) {
            flag = 1;
            break;
        }
    if (flag) break;
}
//====After using start_i and start_j
start_i = idle_index[lkpID] / 32;
start_j = idle_index[lkpID] % 32;
for (i = start_i; i < IDLE_GROUP_AMOUNT; i++) {
    x = idle_block[lkpID][i];
    if (x == 0) continue;
    for (j = (i == start_i ? start_j : 0); j < 32; j++)
        if (x & (1 << j)) {
            flag = 1;
        }
}
```

```

        break;
    }
    if (flag) break;
}

```

这里使用了 2 个中间变量：start_i、start_j。

例子：使用中间变量，Verilog 代码

```

wire [8:0] addr_tcfr = (channel_base + `TCU_CH_OFFSET * c_number +
`TCU_CHOF_TCFR);
wire sel_tcfr = (psel && penable && paddr[8:0] == addr_tcfr);
wire wr_tcfr = (sel_tcfr && pwrite);
wire [8:0] addr_tchr = (channel_base + `TCU_CH_OFFSET * c_number +
`TCU_CHOF_TCHR);
wire sel_tchr = (psel && penable && paddr[8:0] == addr_tchr);
wire wr_tchr = (sel_tchr && pwrite);

```

这里使用了 4 个中间变量：addr_tcfr、sel_tcfr、addr_tchr、sel_tchr。

14.3 使用等价逻辑

对于一个逻辑表达式，你可以使用两种等价的方式表达它们，这就是德摩根定律。

```

!(a && b && c) <=> !a || !b || !c
!(a || b || c) <=> !a && !b && !c

```

对于有的表达式，你可以用德摩根定律把它换一种方式表示出来，使之更简洁，更具有可读性。

例子：使用德摩根定律

```

if (!(mParaInOneZonePtr->plktPtr[slot].mapFlag == FTL_PLKT_MAP_ALL_PAGE
&& mParaInOneZonePtr->plktPtr[slot].emptyVPP > 6
&& mParaInOneZonePtr->plktPtr[slot].emptyVPP != vppAmountOfVPB))
//After using De Morgan Law
if (mParaInOneZonePtr->plktPtr[slot].mapFlag != FTL_PLKT_MAP_ALL_PAGE
|| mParaInOneZonePtr->plktPtr[slot].emptyVPP <= 6
|| mParaInOneZonePtr->plktPtr[slot].emptyVPP == vppAmountOfVPB)

```

14.4 简化判断逻辑

在很多编程语言中，赋值操作可以写在 if 或 while 等的判断条件里面。有些人很推崇使用这种方式，因为他们觉得这种方式用起来很方便，而且代码看起来更紧凑一些。但是从实际使用效果上看，这种方式可能会被滥用，从而实现了复杂的逻辑。

另外，当判断条件的表达式很复杂时，就要使用中间变量的方法简化判断表达式，不要在判断条件中塞一堆东西。

例子：看着费劲，因为判断中包含有赋值操作，还包含一层“()”

```

if (((node = find_name (name)) != NULL) && node->salary < 8192)

```

例子：看着方便，因为把赋值操作从判断中抽取出来，减少一层“()”

```

node = find_name (name);
if (node != NULL && node->salary < 8192)

```

这说明不要为了减少代码行数，就把表达式搞得非常复杂，否则会让人读起来很困惑。

14.5 使用宏定义

代码中出现原始数字通常是很坏的现象，因为这些原始数字不容易维护和修改，所以应该使用良好命名的常量来隐藏它们。另外，如果常量之间有运算关系，定义时也要保留这种关系，不要自己计算这些值。

代码中有时会出现多个复杂的表达式，这些复杂的表达式又长又相似，这时可以把复杂的表达式定义成宏，然后使用宏简化表达式的书写。通过使用宏定义，就可以简化代码的书写，提高代码的可读性。

例子：使用宏定义常量，保持常量之间的运算关系

```
#define KPC_DRV_BASE      0x00028000
#define KPC_DRV_CMD      (KPC_DRV_BASE + (0x0000 << 2))
#define KPC_DRV_DATA0    (KPC_DRV_BASE + (0x0001 << 2))
#define KPC_DRV_DATA1    (KPC_DRV_BASE + (0x0002 << 2))
```

例如，在嵌入式系统中，经常要访问内存映射寄存器，这些寄存器必须按照 `volatile` 形式的指针访问，否则就会出错，所以就用如下的形式访问。

例子：使用 `volatile` 形式的指针

```
if ((*volatile unsigned int *)SPI_STAT_REG) & 0x0009)
    *(volatile unsigned int *)SPI_CTRL_REG = 0x0002;
```

但是这种形式很啰嗦，而且书写容易出错，可以通过宏定义简化书写。

例子：使用宏定义简化代码的书写

```
#define REG32(reg) (*(volatile unsigned int *)(reg))
if (REG32(SPI_STAT_REG) & 0x0009)
    REG32(SPI_CTRL_REG) = 0x0002;
//Use REG32 to make other expression simple
#define ost_set_tccr_one_shot(x, val) \
    (REG32(OST_TCCR(x)) &= (~(1<< 6)) | (((val) & 1) << 6))
#define ost_set_tccr_pre(x, val) \
    (REG32(OST_TCCR(x)) &= (~(7<< 3)) | (((val) & 7) << 3))
#define ost_set_tccr_sel(x, val) \
    (REG32(OST_TCCR(x)) &= (~(7<< 0)) | (((val) & 7) << 0))
```

因为使用宏定义可能会造成一些麻烦或错误，所以现在已经不鼓励使用宏定义，可以改用 `const`、`enum` 定义常量，改用 `inline` 函数代替函数宏（Function-like Macro）。

在 Verilog 中，宏定义（``define`）常被用来定义常数，较少用来定义函数宏，但是因为宏定义在 Verilog 代码中是全局的，可能会导致名字重定义，所以要仔细检查，另外如果只是在模块内使用的常量，建议使用 `localparam`。

14.6 使用查找表

我们看一下 C 语言 `ctype` 相关函数的实现，`ctype` 相关的函数用于检查字符所属类别，包含：`isalpha`，`isalnum`，`isascii`，`isctrl`，`isdigit`，`isgraph`，`islower`，`isprint`，`ispunct`，`isspace`，`isupper`，`isxdigit`。这组函数很简单，我们可能认为它们是按如下方式的实现的，就是通过判断语句实现的。

例子：`ctype` 函数的简单实现，或者使用宏定义实现

```
int isdigit (int c)
```

```

{
    return c >= '0' && c <= '9';
}

int iscntrl (int a)
{
    return a >= 0 && a <= 30;
}

int isspace (int a)
{
    return a == ' ' || a == '\t' || a == '\n' || a == '\v' || a == '\f' ||
a=='\r';
}

```

但是，实际上这些库函数不是通过判断语句实现的，而是按如下的查找表方式实现的，这样的实现是不是更加简洁呢？而且执行起来更加快捷。例如，如果 `isspace` 按上面的判断方式实现，最多时需要执行 6 次判断，而使用查找表，直接就返回结果。

例子：`ctype` 函数的查找表实现

```

//定义字符属性类别
#define _U      0x01    /* upper */
#define _L      0x02    /* lower */
#define _D      0x04    /* digit */
#define _C      0x08    /* cntrl */
#define _P      0x10    /* punct */
#define _S      0x20    /* white space (space/lf/tab) */
#define _X      0x40    /* hex digit */
#define _SP     0x80    /* hard space (0x20) */

//定义 256 元素的查找表
unsigned char _ctype_lookup[] = {
    _C,_C,_C,_C,_C,_C,_C,_C,          /* 0-7 */
    _C,_C|_S,_C|_S,_C|_S,_C|_S,_C|_S,_C|_S,_C|_S, /* 8-15 */
    _C,_C,_C,_C,_C,_C,_C,_C,          /* 16-23 */
    _C,_C,_C,_C,_C,_C,_C,_C,          /* 24-31 */
    _S|_SP,_P,_P,_P,_P,_P,_P,_P,      /* 32-39 */
    _P,_P,_P,_P,_P,_P,_P,_P,          /* 40-47 */
    _D,_D,_D,_D,_D,_D,_D,_D,          /* 48-55 */
    _D,_D,_P,_P,_P,_P,_P,_P,          /* 56-63 */
    _P,_U|_X,_U|_X,_U|_X,_U|_X,_U|_X,_U|_X,_U, /* 64-71 */
    _U,_U,_U,_U,_U,_U,_U,_U,          /* 72-79 */
    _U,_U,_U,_U,_U,_U,_U,_U,          /* 80-87 */
    _U,_U,_U,_P,_P,_P,_P,_P,          /* 88-95 */
    _P,_L|_X,_L|_X,_L|_X,_L|_X,_L|_X,_L|_X,_L, /* 96-103 */
    _L,_L,_L,_L,_L,_L,_L,_L,          /* 104-111 */
    _L,_L,_L,_L,_L,_L,_L,_L,          /* 112-119 */
    _L,_L,_L,_P,_P,_P,_P,_C,          /* 120-127 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* 128-143 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* 144-159 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* 160-175 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* 176-191 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* 192-207 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* 208-223 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /* 224-239 */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; /* 240-255 */
//或用函数实现
int isdigit (int c) {return (_ctype_lookup[c] & _N); }
int iscntrl (int c) {return (_ctype_lookup[c] & _C); }
int isspace (int c) {return (_ctype_lookup[c] & _S); }

```



```
//或用宏定义实现
#define isdigit(__c) (_ctype_lookup(__c) & _N)
#define iscntrl(__c) (_ctype_lookup(__c) & _C)
#define isspace(__c) (_ctype_lookup(__c) & _S)
```

我们编写别的代码的时候，如果能够建立类似的查找表，那是最好的了，可以节省好多代码，而且维护起来更加方便。

14.7 注意操作符

C、Verilog、Perl 等语言都有丰富的操作符，它们都有不同的优先级。当一个复杂表达式有多种运算符参与时，我们经常会被供应商绕晕，这时候的解决办法就是添加括号，通过括号明确表示运算的优先级。但是括号多了也麻烦，因为看代码时为了匹配左右括号，也经常绕晕。所以最好的解决办法还是通过增加中间变量，减少括号，降低表达式的复杂度。

C 语言的操作符除了多，还有个特点就是容易混淆，例如，=和==之间误用，逻辑操作符(&&、||、!)和位操作符(&、|、~、^)之间就容易搞混，前++/--和后++/--容易搞混，前++/--、后++/--与*（指针取值）一起用的时候也容易搞混，所以一定要搞明白这些容易混淆的地方。

另外，尽量不要把前++/--、后++/--与*（指针取值）一起用，把指针取值操作放到一个解释变量中就很好，也就增加几行代码罢了。

例子：经典的 strcpy()函数是按照如此写法，++和*一起使用

```
char *strcpy (char *des, const char *src)
{
    char *r = des;
    while ((*des++ = *src++) != '\0');
    return r;
}
```

例子：修改后的 strcpy()函数，++和*分开使用

```
char *strcpy (char *des, const char *src)
{
    char *r = des;
    while (1) {
        char tmp = *src;
        *des = tmp;
        des++;
        src++;
        if (tmp == '\0') break;
    }
    return r;
}
```

表面上看修改后的 strcpy()函数看着啰嗦，但是编译后的汇编代码其实是一样的。对于 strcpy 这么简单的代码，可以把++和*一起使用，达到更紧凑的效果，但是当指针的操作很复杂时，就不要把++和*一起使用，因为容易把人搞糊涂，在后面的“指针”章中就有一些对应的例子。

Verilog 的操作符比 C 语言的操作符要少一些，简单一些；Perl 的操作符比 C 语言的操作符更多一些，更复杂一些，虽然没有指针操作，但是有别的复杂操作。不管怎样，很多操作是相同或类似的，所以我们可以从 C 语言的操作符借鉴很多东西。

14.8 简洁的写法

有些时候，通过换一种写法，减少代码的书写量，就可以达到简化表达式的目的，从而提高代码的可读性。这就需要我们熟悉最新的语言标准，例如 C 语言的 C89 和 C99 标准，Verilog 的 Verilog-1995 和 Verilog-2001 标准。我们更需要学习了解 C99 和 Verilog-2001 标准。

例如，C 语言数组的初始化，有时可能数组的元素大部分都是 0，如果使用 C89 标准，就需要写很多 0，但是如果使用 C99 标准，只需要写特定的非零值。

例子：C 语言数组的初始化，C89 和 C99 标准

```
int a[15] = {0, 0, 29, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 29}; //C89
//is changed to
int a[15] = {[2] = 29, [9] = 7, [14] = 28}; //C99
```

例子：相比于“*”形式的访问，使用“[]”形式的访问可以减少书写，获得更好的可读性

```
if ((* (src_ptr + 0) == INTF_CMD_TAG1) && (* (src_ptr + 1) == INTF_CMD_TAG2)
    && (* (src_ptr + 2) == INTF_CMD_TAG3) && (* (src_ptr + 3) == INTF_CMD_TAG4)
    && (* (src_ptr + 4) == INTF_CMD_TAG5) && (* (src_ptr + 5) == INTF_CMD_TAG6))
//is changed to
if (src_ptr[0] == INTF_CMD_TAG1 && src_ptr[1] == INTF_CMD_TAG2
    && src_ptr[2] == INTF_CMD_TAG3 && src_ptr[3] == INTF_CMD_TAG4
    && src_ptr[4] == INTF_CMD_TAG5 && src_ptr[5] == INTF_CMD_TAG6)
```

Verilog 中的连接操作有两种格式 {a, a, a} 和 {3 {a}}，但是很多人只是停留在 {a, a, a} 上，很少使用 {3 {a}}，导致某些代码很啰嗦。例如，下面要把 wen[3:0] 扩展成 ram_wen[31:0]，可以有如下两种写法。

例子：啰嗦的写法

```
wire [31:0] ram_wen =
    {wen[3], wen[3], wen[3], wen[3], wen[3], wen[3], wen[3],
     wen[2], wen[2], wen[2], wen[2], wen[2], wen[2], wen[2],
     wen[1], wen[1], wen[1], wen[1], wen[1], wen[1], wen[1],
     wen[0], wen[0], wen[0], wen[0], wen[0], wen[0], wen[0], wen[0]};
```

例子：紧凑的写法

```
wire [31:0] ram_wen = {{8{wen[3]}}, {8{wen[2]}}, {8{wen[1]}}, {8{wen[0]}}};
```

使用紧凑的写法还有一个好处就是书写不容易出错，因为在啰嗦的写法中，如果某一个 wen 的下标写错了，就不容易看出来，但是在紧凑的写法中，如果某一个 wen 的下标写错了，很容易看出来，因为书写的代码量少了。

例子：啰嗦的写法，检查出 wen 的下标错误，需要扫描很多代码

```
wire [31:0] ram_wen =
    {wen[3], wen[3], wen[3], wen[3], wen[3], wen[3], wen[3], wen[3],
     wen[2], wen[2], wen[2], wen[2], wen[3], wen[2], wen[2], wen[2],
     wen[1], wen[1], wen[1], wen[1], wen[1], wen[1], wen[1], wen[1],
     wen[0], wen[0], wen[0], wen[0], wen[0], wen[0], wen[0], wen[0]};
```

例子：紧凑的写法，检查出 wen 的下标错误，只须扫描较少的代码

```
wire [31:0] ram_wen = {{8{wen[3]}}, {8{wen[3]}}, {8{wen[1]}}, {8{wen[0]}}};
```

例如，我们要从 32-bit 的 vector 按照 index 抽取出 4-bit，按照 Verilog-1995 标准书写出来的代码显得很啰嗦，但是在使用 Verilog-2001 标准增加的“+.”后，可以获得非常简洁的表达式。

例子：当 `vector` 的位长很大时，代码就很啰嗦，而且不好移植。

```
always @(*)
begin
  case (index)
    0: value = vector[3:0];
    1: value = vector[7:4];
    2: value = vector[11:8];
    3: value = vector[15:12];
    4: value = vector[19:16];
    5: value = vector[23:20];
    6: value = vector[27:24];
    7: value = vector[31:28];
  endcase
end
```

例子：使用 `part-select` 之后，代码就很简洁，而且这是可以综合的。

```
always @(*)
begin
  value = vector[index*4 +: 4];
end
```

第 15 章

控制流易读

如果代码中没有条件、循环或者其他的控制语句，那么代码的可读性会非常好，但是实际的代码不会是这个样子，条件、循环或者其他的控制语句可能会让代码变得混乱。本章就讨论如何让控制流更容易读，关键思想就是把条件、循环或者其他控制语句变得越自然越好，不用让读者停下来重读你的代码。

15.1 组织直线型代码

直线型代码是指执行顺序相关的代码，代码中的语句具有相互依赖的关系。直线型代码是最容易理解的代码，例如下面的代码就是按照先后顺序处理数据的。

例子：直线型代码

```
input_data (data);
process_data (data, result);
output_result (result);
```

虽然有些代码也是直线型代码，但是代码中前后相连的语句没有依赖关系，这样的顺序对于代码的可读性也有很大的影响，因为阅读时读者的目光需要跳来跳去。对于这样的代码，可以依据就近原则，把相关的代码调整到一起，这样就可以按照自上而下的顺序阅读了。相关的代码是指代码处理同一批的数据、执行同一批的任务或者执行具有某种相互依赖的关系。

例子：直线型代码，不好

```
input_data (data0);
input_data (data1);
input_data (data2);

process_data (data0, result0);
process_data (data1, result1);
process_data (data2, result2);

output_result (result0);
output_result (result1);
output_result (result2);
```

例子：直线型代码，调整

```
input_data (data0);
process_data (data0, result0);
output_result (result0);

input_data (data1);
process_data (data1, result1);
output_result (result1);

input_data (data2);
```

```
process_data (data2, result2);
output_result (result2);
```

进一步，这样的代码可以使用循环实现。

```
例子：使用循环实现
for (i = 0; i < 3; i++) {
    input_data (data[i]);
    process_data (data[i], result[i]);
    output_result (result[i]);
}
```

另外，当你把相关的代码组织在一起之后，就会发现它们之间有很强的联系，而与它们前后的语句没有太多的联系。在这种情况下，你就可以把它们独立出来做成一个独立的函数。

15.2 判断中的表达式

为了便于阅读代码，尽量不要在判断中使用复杂的表达式，我们可以通过下面的方法简化判断中的表达式：

- 1) 可以通过引入中间变量，降低表达式的难度。
- 2) 使用德摩根定律把表达式修改为等价逻辑。
- 3) 使用括号让表达式的逻辑更清晰一些。

这些方法已经在上一章“表达式易读”中做了介绍，这里就不多说了。

15.3 判断中的注意事项

为了提高代码的可读性，对布尔变量、整型变量和指针变量做比较时，应该选择能够和变量类型匹配的操作，虽然使用其他操作也是等价的。

如果是布尔变量，应该直接使用或使用“!”，不要把它与 1 和 0 比较。

```
if (ready == 0)
if (ready != 0)
//应该使用下面的
if (ready)
if (!ready)
```

如果整数与 0 比较，应该使用==或!=与 0 比较，不要直接使用或使用“!”。

```
if (number)
if (!number)
while (i) { printf ("index %d", i); i--;}
//应该使用下面的
if (number == 0)
if (number != 0)
while (i != 0) { printf ("index %d", i); i--;}
while (i > 0) { printf ("index %d", i); i--;} //这个也可以
```

如果检查指针变量，应该使用==或!=与 NULL 比较。不要把它和 0 比较，也不要直接使用或者使用“!”，否则会让人误以为它是一个整型变量。

```
if (p == 0)
if (p != 0)
if (p)
if (!p)
```

```
//应该使用下面的
if (p == NULL)
if (p != NULL)
```

如果是 float/double 变量，不要使用 “==” 或 “!=” 与期望数值比较，因为它们有精度限制。

```
if (value == expect)
//应该使用下面的
#define DELTA 0.000001
if (value >= (expect - DELTA) && value <= (expect + DELTA))
```

15.4 判断中的参数顺序

为了便于阅读代码，对于 if、while、for 中判断 true/false 的表达式，表达式的左侧应该是不断变化的值，右侧则应该是用来比较的值，右侧更倾向于是一个常量或者不变量。

这个原则和英语的语法一致，也和汉语的语法一致，因为我们会很自然地说“如果你的工资大于 9000 元”，而不会说“如果 9000 元小于你的工资”。同样，对应的代码“if (salary > 9000)” 就比 “if (9000 < salary)” 更加具有可读性。

例子：按自然说话的顺序

```
while (word_received < word_expected) //好
while (word_expected > word_received) //不好
for (i = 0; i < MAX_WORDS; i++) //好
for (i = 0; MAX_WORDS > i; i++) //不好
```

15.5 判断中的赋值语句

在 C 语言中，“=” 用于表示赋值，“==” 用于表示判断相等，而“=” 也可以放在 if、while、for 中的判断条件中^[Dustin Boswell]。

例子：“==” 误写成 “=”

```
fopen (file_name, "rb");
if (fp = NULL) { .....}
```

这是程序员的笔误，但是对于这样的代码编译也是可以通过的。实际应该为：

```
fopen (file_name, "rb");
if (fp == NULL) { .....}
```

所以为了避免出现这样的 Bug，很多人就把参与比较的这两个参数交换了一下位置，代码改为如下的样子。

例子：交换顺序，以防止 “==” 误写成 “=”

```
if (NULL == fp) { .....}
```

这时候如果把 “==” 误写成 “=”，那么对于 if (NULL = fp)，编译就会报告错误。虽然这种写法可以有效地避免书写错误，但是这种写法读起来很不自然，相当别扭，而且这种写法只可用在变量和常量之间。现在，编译器已经能对 if (fp = NULL) 这样的代码给出警告，所以 if (NULL == fp) 这种写法已经过时，不要再这么写了，只要检查编译器的警告，就可以很容易地发现 “==” 误写为 “=”。

有人可能想进一步简写代码，把上面的两行代码合为下面的一行，因为赋值也可以放在

判断条件中。

例子：判断中使用赋值，总是报告警告

```
if (fp = fopen (file_name, "rb"))
```

这是正确的代码，但是编译器总是会报告出警告，怎么避免呢？这时只要在赋值表达式外面再加一级“()”，编译器就不会再给出警告。

例子：判断中使用赋值，增加一级“()”，消除编译警告

```
if ((fp = fopen (file_name, "rb")))
```

不管怎样，为了便于阅读代码，同时减少书写错误，最好不要在判断语句中使用赋值，虽然判断语句中可以使用赋值。

15.6 if 语句的逻辑顺序

为了便于阅读代码，对于 `if (cond) {...} else { ... }` 语句的判断逻辑，可以优先处理某个分支。

- 1) 先处理正逻辑的情况，后处理负逻辑的情况，也就是要符合人的思维方式。
- 2) 先处理常见的情况，后处理不常见的情况，也就是要符合人的思维方式。
- 3) 先处理简单的情况，后处理复杂的情况，也就是“先易后难”的原则。
- 4) 先处理可疑、危险或错误的情况，后处理正确的情况，也就是“排除异常”的原则。

例子：对正逻辑的情况先做处理

```
if (!status_ready) {
    //do status_not_ready...
}
else {
    //do status_is_ready...
}
```

根据实际情况把代码修改如下，先处理正逻辑的情况。

```
if (status_ready) {
    //do status_is_ready...
}
else {
    //do status_not_ready...
}
```

例子：对 `fp == NULL` 的情况先做处理

```
fp = fopen (file_name, "rb");
if (fp == NULL) {
    status == ERROR_OPEN_FILE;
}
else {
    fread (buffer, length, 1, fp);
    //process_buffer .....
}
```

按照这样的方式编写 `if/else` 代码，不仅可以获得更好的可读性，而且在把这样的代码转为函数时，通过使用 `return` 可以很方便地减少多层嵌套，提高代码的可读性，这部分内容要在“减少嵌套”节中讨论。

15.7 使用 “?:”

众所周知，`x = cond ? a : b` 是 `if (cond) { x = a; } else { x = b; }` 的简写。有的人认为使用 “?:” 就只须写一行代码，可以提高可读性，有的人则认为 “?:” 会造成阅读上的困难，而且很难用调试器来调试。其实，我们只要根据情况选择使用 “?:” 或者 `if/else` 即可。如果表达式比较简单，就可以使用 “?:”，因为这样代码会更加紧凑而且容易阅读。

例子：“?:” 用于简单的表达式

```
if (x > y) return x - y;
else      return 0;
//将代码修改如下更好
return (x > y) ? x - y : 0;
```

例子：“?:” 可以获得更好的表达效果，`newlib` 的 `mprec.c` 中的代码

```
z = xa > xa0 ? *--xa : 0;
d0 = Exp_1 | y << k - Ebits | z >> Ebits + 16 - k;
w = xa > xa0 ? *--xa : 0;
y = xa > xa0 ? *--xa : 0;
d1 = z << k + 16 - Ebits | w << k - Ebits | y >> 16 + Ebits - k;
```

有些时候，连着使用 “?:”，会收到非常好的效果，可读性更好，而且代码也简单。

例子：连着使用 “?:”，`newlib` 中的 `fstab.c`

```
f->fs_type = (__hasmntopt (m, FSTAB_RW) ? FSTAB_RW :
             __hasmntopt (m, FSTAB_RQ) ? FSTAB_RQ :
             __hasmntopt (m, FSTAB_RO) ? FSTAB_RO :
             __hasmntopt (m, FSTAB_SW) ? FSTAB_SW :
             __hasmntopt (m, FSTAB_XX) ? FSTAB_XX : "??");
```

如果表达式复杂，最好不用 “?:”，因为代码会变得很难阅读，因为已经没有必要把代码都挤到一个表达式里面。

例子：使用 `if` 语句更好

```
result = ((scale && (y = dword0(rv) & Exp_mask) <= 2*P*Exp_msk1) ? (0xffffffff
& (0xffffffff << (2*P+1-(y>>Exp_shift)))) : 0xffffffff);
//将代码修改如下更好
if (scale && (y = dword0(rv) & Exp_mask) <= 2*P*Exp_msk1)
    result = (0xffffffff & (0xffffffff << (2*P+1-(y>>Exp_shift))));
else
    result = 0xffffffff;
```

15.8 使用 `switch`

当要比较的值在某个范围之内，而且要与这个范围内的值依次比较，如果使用 `if` 语句，要写一连串的 `if` 语句，这时就可以使用 `switch` 语句。相比于 `if` 语句，`switch` 语句更加规整，像一个表格一样。使用 `switch` 要注意如下情况。

- 1) 要按照字母顺序或者数字顺序排列各种情况，或者把正常的情况放在前面。
- 2) 简化每种情况处理的代码，因为短小精悍的代码会使 `switch` 的结构更加清晰。
- 3) 要在 `switch` 的每个分支结束处添加 `break`，除非是特意的 `follow` 执行。
- 4) 虽然 `default` 是可选的，但是最好也加上，用于处理缺省的情况或者错误的情况。

例子：使用 `switch`

```
switch(grade)
```



```

{
case 'A':
    printf("85-100\n");
    break;
case 'B':
    printf("70-84\n");
    break;
case 'C':
    printf("60-69\n");
    break;
case 'D':
    printf("0 -60\n");
    break;
default:
    printf("error!\n");
    break;
}

```

使用 `switch`，还可以获得更快的执行速度，因为编译器首先会对所有 `case` 建立一个跳转表 `jump_table`，然后计算 `grade` 与“A”的差值 `index`，如果 `index` 在 0~3 之间，就跳转到 `jump_table[index]`，否则跳转到 `default` 执行。

15.9 使用 `return`

有些人觉得函数中不应该出现多个 `return` 语句，这是不对的，因为从函数中提前返回会收到很好的可读性。

例子：使用 `return`

```

char * dirname (char *path)
{
    char *p;
    if( path == NULL || *path == '\0' )
        return ".";
    p = path + strlen(path) - 1;
    while( *p == '/' ) {
        if( p == path )
            return path;
        *p-- = '\0';
    }
    while( p >= path && *p != '/' )
        p--;
    return
        p < path ? "." :
        p == path ? "/" :
        (*p = '\0', path);
}

```

但是有的时候，确实不能简单地使用 `return`，因为在 `return` 之前要做一些清理工作，例如需要关闭文件，或者释放 `malloc` 分配的内存，这时候最好有一个统一的 `return` 出口。

15.10 选择 `for/while`

`for` 和 `while` 都用于循环，`while` 循环更通用一些，所以要根据情况选择 `for` 或者 `while` 循环。当循环是对某一个变量进行迭代，而且这个变量是在做递增或递减操作时，使用 `for` 循环

会获得更好的表现效果。

```
例子：while 和 for 的对比，for 循环更加简洁
i = 0;
while (i < 100) {
    printf ("%d\n", i);
    i++;
}

for (i = 0; i < 100; i++)
    printf ("%d\n", i);
```

如果循环执行 count 次，根据情况可以有多种写法，但还是有常用的且更容易理解的写法。

```
例子：几种 for 循环的写法
//循环在 0 ~ count-1 之间，常用
for (i = 0; i < count; i++)           //升序，常用
for (i = 0; i <= count - 1; i++)     //升序，不常用
for (i = count - 1; i >= 0; i--)     //降序，常用
//循环在 1 ~ count 之间，不常用
for (i = 1; i <= count; i++)         //升序，常用
for (i = 1; i < count + 1; i++)     //升序，不常用
for (i = count; i >= 1; i--)        //降序，常用
```

可以看出，常用的写法就是在判断条件中不做加减法运算。

15.11 少用 do/while

do/while 循环是 while 循环的变体，在检查条件是否为真之前，就先执行一次循环体，然后才检查条件是否为真，如果条件为真的话，就会重复这个循环。

```
例子：使用 do/while 循环
FILE * _tmpfile_r (struct _reent *ptr)
{
    FILE *fp;
    int e;
    char *f;
    char buf[L_tmpnam];
    int fd;
    do {
        if ((f = _tmpnam_r (ptr, buf)) == NULL)
            return NULL;
        fd = _open_r (ptr, f, O_RDWR | O_CREAT | O_EXCL | O_BINARY,
                    S_IRUSR | S_IWUSR);
    } while (fd < 0 && ptr->_errno == EEXIST);
    if (fd < 0)
        return NULL;
    fp = _fdopen_r (ptr, fd, "wb+");
    e = ptr->_errno;
    if (!fp)
        _close_r (ptr, fd);
    _CAST_VOID _remove_r (ptr, f);
    ptr->_errno = e;
    return fp;
}
```

通常来说，判断条件应该位于它们所“保护”的代码前面，这也是 if、while、for 语句的

工作方式，我们在看这样的代码时，从前向后只看一遍即可。但是 do/while 这种循环很奇怪，因为它的判断条件放在了结尾处，显得很不自然，很多人在看这样的代码时，就不得不看两遍。

有些人推崇 do/while 循环，因为 do/while 至少要执行一次，但是为了这个原因，把代码写成不自然的形式，会导致代码的可读性降低。其实，所有的 do/while 循环都可以很方便地改为 while 循环，从而获得更好的可读性。

例子：使用 while 循环

```
FILE * _tmpfile_r (struct _reent *ptr)
{
    FILE *fp;
    int e;
    char *f;
    char buf[L_tmpnam];
    int fd;
    while (1) {
        if ((f = _tmpnam_r (ptr, buf)) == NULL)
            return NULL;
        fd = _open_r (ptr, f, O_RDWR | O_CREAT | O_EXCL | O_BINARY,
                    S_IRUSR | S_IWUSR);
        if (!(fd < 0 && ptr->_errno == EEXIST))
            break;
    }
    .....
}
```

在这里，使用 while(1)死循环，当条件不满足时就跳出循环，是不是这样能让代码更加自然呢？

下面是另一个例子，可以对比一下可读性，而且 while 循环减少了一个临时变量 j 的使用。

例子：修改前，使用 do/while

```
check_ptr = NULL;
j = SDIF_SCAN_ADDR;
src_ptr = SYS_RootDir_Check_ptr;
do
{
    if((write_addr >= src_ptr[1]) && (write_addr < src_ptr[2]))
    {
        j = 0;
        cur_cmd = SDIF_SCAN_ADDR;
        sdif_State |= SDIF_WRITE_DIR;
    }
    else if((write_addr >= src_ptr[5]) && (write_addr < src_ptr[7]))
    {
        check_ptr = src_ptr;
        sdif_State |= SDIF_SCAN_FAT;
    }
    if(j)
    {
        if(src_ptr[3])
            j = 0;
        else
            src_ptr += 8;
    }
} while(j);
```

例子：修改后，使用 while 循环

```
check_ptr = NULL;
src_ptr = SYS_RootDir_Check_ptr;
while (1)
```

```

{
    if((write_addr >= src_ptr[1]) && (write_addr < src_ptr[2]))
    {
        cur_cmd = SDIF_SCAN_ADDR;
        sdif_State |= SDIF_WRITE_DIR;
        break;
    }
    else if((write_addr >= src_ptr[5]) && (write_addr < src_ptr[7]))
    {
        check_ptr = src_ptr;
        sdif_State |= SDIF_SCAN_FAT;
    }
    if(src_ptr[3])
        break;
    else
        src_ptr += 8;
}

```

当然 do/while 还是有很大用处的，它广泛用于宏定义中，可以防止错误的宏扩展，在后面的章节还会做详细的讨论。

例子：使用 do/while 定义宏

```
#define set_task_state(task, value)    do { (task)->state = (value); } while (0)
```

15.12 少用 goto

goto 语句是对程序结构影响很大的语句，因为滥用 goto 会使程序难以理解，把程序变成垃圾代码。研究表明，任何程序都可以用顺序、分支和重复结构表示出来，这就是结构化程序设计方法，所以去掉 goto 后，可使程序结构清晰，更加便于理解。

但是也有人认为，goto 使用起来比较灵活，而且有些情况确实能提高程序的效率，若完全删去 goto，有些情况反而会使程序过于复杂，增加一些不必要的计算量。

所以在 C/C++ 中保留了 goto，但被建议不用或少用。对于一些更新的高级编程语言，Java 虽然把 goto 当作关键字，但是不提供 goto 的功能；C# 初始不支持 goto，但是后来还是加上了对 goto 的支持。

使用 goto 一个好处就是可以保证代码存在唯一的出口，避免过于庞大的 if/for/while 等嵌套，所以我们可以从一些 C 语言程序中看到 goto，尤其是在 Linux 内核和大量的库函数中。

例子：newlib 的 puts_r.c，有 3 处使用了 goto

```

int _puts_r (struct _reent *ptr, const char * s)
{
    int result = EOF;
    const char *p = s;
    FILE *fp;
    _REENT_SMALL_CHECK_INIT (ptr);
    fp = _stdout_r (ptr);
    _newlib_flockfile_start (fp);
    ORIENT (fp, -1);
    /* Make sure we can write. */
    if (cantwrite (ptr, fp))
        goto err;
    while (*p)
    {
        if (__sputc_r (ptr, *p++, fp) == EOF)
            goto err;
    }
}

```

```

    }
    if (__putc_r (ptr, '\n', fp) == EOF)
        goto err;
    result = '\n';
err:
    _newlib_flockfile_end (fp);
    return result;
}

```

如果只允许这样的 goto 出现,那么代码就不会出现任何问题。可怕的是,胡乱地使用 goto,随意地跳来跳去,作者就遭遇过这样的代码,竟然使用 goto 从 for 循环的外部跳到循环的内部,设计者就是要使用循环内部的一部分代码,但是这样做是非常危险的。

15.13 语句对比

前面介绍的都是 C 语言相关的控制流方面的处理,表 15-1 是一个对比表格,可以把前面介绍的思想映射到 Perl、Verilog 语言上,当然也可以映射到其他语言上。

表 15-1 C 语言、Perl、Verilog 控制语句的对比

语 句	Perl	Verilog
if/else	对于简单语句,需要使用{ },对于 for、while 等语句也是如此。 例如: 只能写成 if (a > b) { x = a;} else {x = b;} 但是 if 有另一种形式,如下 x = b; x = a if (a > b);	对于简单语句,可以不用 begin/end,对于 for、while 等语句也是如此。 例如: if (a > b) begin x = a; end else begin x = b;end 简写为 if (a > b) x = a; else x = b;
?:	相同	相同
switch	没有,可以使用一些替代方法实现类似的功能,还有 Switch package 可用	使用 case/casex/casexz,功能更强。 没有 break,还可以使用简写形式(..)。 可能存在误用的情况:一是不同 case 对 x/z/?的处理不一样,二是仿真和综合的结果可能会不一样
goto	更强	没有
for	相同	相同
while	相同	相同
do/while	do/while 相同 还有 do/until,但是与 do/while 相反,当 until 的条件为 true 时就退出循环	没有 do/while
其他循环	foreach,用于处理数组元素	repeat,指定循环次数,用于实现简单的循环 forever,实现死循环,常用于生成时钟
break	使用 last 实现 break 功能,而且 last 更强大,可以跳出多层循环	没有 break,使用 disable 实现
continue	使用 next 开始下一轮循环	没有 continue,使用 disable 实现
return	相同	没有 return,使用 disable 实现

15.14 减少嵌套

多级嵌套发生在混合使用 `if`、`for`、`while` 一层套一层的时候。嵌套很深的代码难以理解，因为每个嵌套层次都在读者的“思维栈”上增加了一个层次。当在多级嵌套的尾部看到多个“`}`”的时候，很难搞清楚这些“`}`”是与前边哪一级的“`{`”相匹配。

在很多现实代码中，很多的多级嵌套是由代码书写问题造成的。例如，有的公司就要求对于 `if/for/while` 语句，必须要使用“`{ }`”，他们觉得这样美观，还会减少错误。其实这是不应该的，这不仅增加了一级嵌套，看的时候还很啰嗦，啰嗦也会导致代码出现问题。

对于这些嵌套，有时换一种写法，就可以消除一层嵌套。减少嵌套的方法如下。

- 1) 如果 `if` 语句简单，分支上只有一条语句，那么就不用啰嗦的“`{ }`”。
- 2) 如果 `if` 语句简单，有 `if` 和 `else` 多个分支，而且都是对同一个变量赋值，可以使用“`?:`”替换。
- 3) 对于 `for/while` 循环，可以使用 `break` 和 `continue`，以减少嵌套级数。
- 4) 对于有些嵌套，在判断之后尽快处理，然后执行 `return` 返回或执行 `break` 跳出循环，以减少嵌套级数。
- 5) 对于有些嵌套，可以通过把内层嵌套转化为子函数，以减少主代码的嵌套级数，然后再减少子函数内的嵌套。

例子：去掉啰嗦的“`{ }`”，消除一层嵌套

```
if (write_addr == PAY_ADDRESS)
    curr_state |= PAY_APP;
else
{
    if (write_addr == USB_ADDRESS)
        curr_state |= USB_APP;
    else if (write_addr == MMC_ADDRESS)
        curr_state |= MMC_APP;
}
//=====改为如下
if (write_addr == PAY_ADDRESS)
    curr_state |= PAY_APP;
else if (write_addr == USB_ADDRESS)
    curr_state |= USB_APP;
else if (write_addr == MMC_ADDRESS)
    curr_state |= MMC_APP;
```

例子：去掉啰嗦的“`{ }`”，消除一层嵌套

```
if (salary <= 3000) {
    income_tax = 0;
}
else if (salary <= 5000) {
    income_tax = (salary - 3000) * 0.05;
}
else if (salary <= 7000) {
    income_tax = (salary - 5000) * 0.10;
}
else if (salary <= 9000) {
    income_tax = (salary - 7000) * 0.15;
}
else {
```

```

    income_tax = (salary - 9000) * 0.20;
}
//=====改为如下
if (salary <=3000)
    income_tax = 0;
else if (salary <= 5000)
    income_tax = (salary - 3000) * 0.05;
else if (salary <= 7000)
    income_tax = (salary - 5000) * 0.10;
else if (salary <= 9000)
    income_tax = (salary - 7000) * 0.15;
else
    income_tax = (salary - 9000) * 0.20;

```

例子：使用“?:”会获得更好的效果

```

income_tax = (salary <= 3000 ? 0 :
             salary <= 5000 ? (salary - 3000) * 0.05 :
             salary <= 7000 ? (salary - 5000) * 0.10 :
             salary <= 9000 ? (salary - 7000) * 0.15 :
             (salary - 9000) * 0.20 );

```

例子：使用 break 消掉一层嵌套

```

for (j = 0; j < total_count; j++)
{
    if (vpb_type == 0) {
        break;
    }
    else {
        //Very many codes
    }
}
//=====改为如下
for (j = 0; j < total_count; j++)
{
    if (vpb_type == 0)
        break;
    //Very many codes
}

```

例子：使用 continue，消掉一层嵌套

```

struct list *node = head;
double total_salary = 0;
int i;
while (node != NULL) {
    for (i = 0; i < check_count; i++) {
        if (strcmp (node->name, check_name[i]) == 0) {
            total_salary += node->salary;
            node->bonus = node->salary * 0.2;
        }
    }
    node = node->next;
}
//=====改为如下
struct list *node = head;
double total_salary = 0;
int i;
while (node != NULL) {
    for (i = 0; i < check_count; i++) {
        if (strcmp (node->name, check_name[i]) != 0)
            continue;
        total_salary += node->salary;
    }
}

```

```

        node->bonus = node->salary * 0.2;
    }
    node = node->next;
}

```

例子：判断之后尽快处理，减少多层嵌套

```

int check_person (char *name, char *addr, char *phone)
{
    int ret_code;
    ret_code = OK;
    if (socket_read (name) != OK) {
        ret_code = ERROR_NAME;
    }
    else {
        process_name (name);
        if (socket_read (addr) != OK) {
            ret_code = ERROR_ADDR;
        }
        else {
            process_addr (addr);
            if (socket_read (phone) != OK) {
                ret_code = ERROR_PHONE;
            }
            else {
                process_phone (phone);
                //other processing .....
            }
        }
    }
    return ret_code;
}

```

//=====改为如下

```

int check_person (char *name, char *addr, char *phone)
{
    if (socket_read (name) != OK)
        return ERROR_NAME;
    process_name (name);

    if (socket_read (addr) != OK)
        return ERROR_ADDR;
    process_addr (addr);

    if (socket_read (phone) != OK)
        return ERROR_PHONE;
    process_phone (phone);
    //other processing .....

    return OK;
}

```

例子：这里的嵌套足足有9级，为了区分哪级嵌套，还用“}for...”或“}//...”做了标注

```

for(i=0; i<flashLayout.vbPerZonePtr[zoneID]; i++)
{
    .....
    if(currentVB == 0xFFFF)
    {
        for(;j<vpbAmountPerZone;j++)
        {
            if(0 == ((emptyBlockPtr[(j >> 3)] >> (j % 8)) & 0x1))
            {
                .....
            }
        }
    }
}

```



```

    }
  }
}
else
{
  if(mParaInOneZonePtr->vb2vpbPtr[i].type == MAPPING_TYPE_PAGE)
  {
    if((mParaInOneZonePtr->vb2vpbPtr[i].value < LOOKUP_TABLE_FOR_
VB_AMOUNT
      && plktPtr[mParaInOneZonePtr->vb2vpbPtr[i].value].vb == i))
    {
      ; //its a real page mapping table
    }
    else
    {
      .....
      for(k = gFTLSysInfo.flashLayout.vppAmountOfVPB-1; k >= 0; k--)
      {
        .....
        if(xbuf[0] != 0xFFFFFFFF)
        {
          m = 0;
          if(((k > 0) && (seqPageAmount == lpn)
            && (lpn == k) ) || ((k == 0) && (lpn == 0)))
          {
            m = 0x1;
          }
          if(m == 0x1)
          {
            .....
          }
          else
          {
            for(k=0;k<CACHED_PAGE_LOOKUP_TABLE_FOR_VB_AMOUNT;
k++)
              {
                if(mParaInOneZonePtr->plktPtr[k].type == 0)
                {
                  .....
                }
              }
            } //for
          }
        } //if(mParaInOneZonePtr->vb2vpbPtr[i].type == MAPPING_TYPE_
PAGE)
      } //else
    } //if((UINT16)mParaInOneZonePtr->vb2vpbPtr[i] == 0xFFFF)
  } //for(i=0; i<flashLayout.vbPerZonePtr[zoneID];i++)
}

```

对于这样的代码，作者使用前面介绍的方法，最后把它们的嵌套层数从 9 级降到 5 级。

15.15 减少代码

使用简单的方法实现同样的功能，就会减少代码的行数，提高代码的可读性，而且这样的代码更加易于测试和维护，这是非常明显的道理。下面的方法就把代码改用数组实现，从而减少代码。

例子：Perl 代码，需要多个 if/else 分支

```
if ($name eq "John Smith") {
    $from = "US";
} elsif ($name eq "Rose Black") {
    $from = "England";
} elsif ($name eq "Wei Jiaming") {
    $from = "China";
}
```

例子：通过使用 hash，减少代码

```
my %name_nation = {"John Smith" => "US",
                  "Rose Black" => "England",
                  "Wei Jiaming" => "China"};
my $from = $name_nation{$name};
```

虽然通过简单地添加新的姓名和国家，上面的代码就可以很方便地修改，但是其实仔细想一想，这段代码是不是可以分两部分实现，姓名和国家拿出来放到另一个文本文件中，让代码保持相对独立。

例子：文本文件，name_nation.txt，使用逗号分割姓名和国家

```
John Smith,US
Rose Black,England
Wei Jiaming,China
```

例子：Perl 代码，处理 name_nation.txt

```
my %name_nation;
read_name_nation();
my $from = $name_nation{$name};

sub read_name_nation
{
    my $f = "name_nation.txt";
    open (FILE_S, "< $f") || die ("Error: can not open $f\n");
    while (my $line = <FILE_S>) {
        chomp ($line);
        my @a = split ("", $line);
        next if (scalar (@a) != 2);
        $name_nation{$a[0]} = $a[1];
    }
    close (FILE_S);
}
```

因为这里的 Perl 代码很简单，是文本文件，使用者也可以修改。但是如果按照同样的想法实现为 C 语言代码，然后编译生成可执行文件，把可执行文件和 name_nation.txt 提供给使用者，使用者就不能修改代码，这样就可以保证代码内容保密而且不外泄。

这样的代码有个好处，就是把代码和数据分割开来，让代码保持相对独立，减少代码的修改，让数据可以动态地修改，不只是程序员可以修改，其他使用者都可以修改。

函数是为了实现某个功能做成的一个相对独立的结构。函数具有这些好处：提高运算性能，节约内存空间，避免代码重复，防止程序膨胀，降低设计复杂度，实现分层和分块设计，隐藏设计细节，实现设计抽象，便于程序设计、调试、维护和移植等。

首先我们要知道什么是不好的函数，然后我们才能设计出好的函数。

16.1 不好的函数

不好的函数具有如下这些特征^[19]。

- 1) 函数没有单一的执行目的，做了好多事，不符合让函数功能单一的原则。
- 2) 函数名字很差劲，模糊、无意义或者表述不清，没有清晰地说明函数的功能是什么。
- 3) 函数使用过多的全局变量，导致函数的独立性很差。
- 4) 函数使用结构类型作为参数或返回值，加大了函数调用的开销。
- 5) 函数使用过多的参数，参数的上限是 7 个，超过 7 个参数的函数就已经很难理解和使用。
- 6) 函数的参数顺序混乱，使用起来不方便。
- 7) 函数的参数没有完全被使用，有些参数放在那里就是摆设。
- 8) 函数的代码行非常多，臃肿庞大。
- 9) 函数没有使用说明和注意事项。

16.2 好的函数

好的函数具有如下这些特征。

- 1) 函数具有单一的执行目的，具有很强的内聚性，符合让函数功能单一的原则。
- 2) 函数名字的意义明确，能够清晰地说明函数的功能是什么。
- 3) 函数没有使用全局变量，使得函数的独立性很好。
- 4) 函数没有使用结构类型作为参数和返回值，通过指针和数组传递大的数据。
- 5) 函数的参数数量合理，没有使用过多的参数，参数数量不会超过 7 个。
- 6) 函数的参数顺序合理，使用起来很方便。参数顺序可以按照输入和输出顺序排列，或者按照先主后次的顺序排列，而且在相关的一组函数内保持参数顺序的一致性。
- 7) 函数的参数完全被使用，没有无用的参数。
- 8) 函数的代码行合理，平均要小于 100 行，大的函数也不要超过 150 行。

9) 函数具有明确的使用说明和注意事项

```
例子: newlib 的 basename()
#include <libgen.h>
#include <string.h>
char* basename (char *path)
{
    if (path == NULL || *path == '\0')
        return ".";
    char *p = path + strlen (path) - 1;
    while (*p == '/') {
        if (p == path)
            return path;
        *p-- = '\0';
    }
    while (p >= path && *p != '/')
        p--;
    return p + 1;
}
```

16.3 小的函数

有的人不愿意为一个简单的功能编写一个小函数，他们觉得没有必要去实现一些小函数，觉得实现小函数会降低性能，于是就在代码中散落了很多短小但又重复的代码。但是实际经验告诉我们，实现小函数也是非常必要的，这些小函数也有很多益处，同样可以提高代码的可读性和可修改性。

```
例子: 小函数
void Nor_Wait_Finish (void)
{
    while(REG32(NOR_STATUS) & 0x1) ;
}

void Nor_Block_Erase (int addr)
{
    REG32(addr) = 0x0;
    REG32(NOR_MODE) = NOR_BLOCK_ERASE;
    Nor_Wait_Finish();
}

void Nor_Page_Erase (int addr)
{
    REG32(addr) = 0x0;
    REG32(NOR_MODE) = NOR_PAGE_ERASE;
    Nor_Wait_Finish();
}

void Nor_Page_Write (int addr, int *buf)
{
    for (int i = 0; i < 128; i++)
        REG32(addr + i * 4) = buf[i];
    REG32(NOR_MODE) = NOR_PAGE_WRITE;
    Nor_Wait_Finish();
}
```

如果觉得调用小函数的开销过大，那么可以使用 `inline` 函数实现，或者使用宏实现。在使用宏实现时，注意确保宏定义展开后是正确的，确保不会把带有副作用的表达式用作参数。

```

例子：ARM926 CP15 相关的宏定义，只列出一部分
#define cp15_read_register(crn, crm, opcode2)          \
    ({ unsigned int _dst_;                             \
      asm volatile("mrc\tp15, 0, %0, c\"#crn\", c\"#crm\", \"#opcode2 :\"=r\" \
(_dst_)); \
      _dst_;})

#define cp15_write_register(crn, crm, opcode2, src)   \
    ({                                               \
      asm volatile("mcr\tp15, 0, %0, c\"#crn\", c\"#crm\", \"#opcode2 :\"=r\" \
(src)); \
    })

#define cp15_read_idcode()          cp15_read_register(0, 0, 0)
#define cp15_read_cache_type()     cp15_read_register(0, 0, 1)
#define cp15_read_tcm_status()     cp15_read_register(0, 0, 2)

#define cp15_invalidate_icache_dcachec() cp15_write_register(7, 7, 0, 0)
#define cp15_invalidate_icache()       cp15_write_register(7, 5, 0, 0)
#define cp15_invalidate_dcachec()     cp15_write_register(7, 6, 0, 0)
#define cp15_drain_write_buffer()     cp15_write_register(7, 10, 4, 0)
#define cp15_wait_for_int()          cp15_write_register(7, 0, 4, 0)

#define cp15_enable_icache()          \
    ({                                 \
      int val;                         \
      val = cp15_read_control();       \
      val |= 0x1000;                  \
      cp15_write_control(val);        \
    })

#define cp15_disable_icache()        \
    ({                                 \
      int val;                         \
      val = cp15_read_control();       \
      val &= ~0x1000;                 \
      cp15_write_control(val);        \
    })

```

16.4 递归调用

递归是指函数直接或间接调用自身的一种方法，通常是把一个大型复杂的问题转化为一个与原问题相似的规模较小的问题来求解。通过递归，只用少量的代码就可描述出解题过程，可以大大地减少代码量。最著名的递归是快速排序，可以查看“快速排序”章的算法介绍和代码。

编写递归函数需要注意如下事项。

- 1) 确认递归能够正常停止，防止出现无穷的递归。
- 2) 要把递归限制在一个函数内，防止出现间接调用的情况。
- 3) 要留心栈空间的使用，防止栈空间的溢出。
- 4) 不要用递归解决简单的问题，例如阶乘或者斐波那契数列，否则就很愚蠢。

我们可以通过使用安全计数器，来防止无穷递归和栈溢出的情况，一旦安全计数器超出安全上限，就会报告出错。

如何做代码重构

“破帽遮颜过闹市，漏船载酒泛中流”，这就是一些代码的状态，代码修修补补还可以运行，但是代码内部已经出现了很多的问题，结构混乱，重复冗余，可读性很差，还有很多缺陷。

现实情况有时是这样的，代码本来就已经很混乱，然后为了解决问题，你还要在原来的代码上改来改去，这种做法就是乱上添乱！你往代码中塞进很多的语句，搞到“可以工作”的状态，让系统勉强工作。你需要反思一下，你什么时候能把“可以工作”的引号去掉呢？

本部分讨论代码重构的好处和方法，包括消除重复、代码切分、少写代码、简化代码，还讨论代码生成、代码测试。

17.1 为什么重构

代码重构就是在不改变系统功能的情况下，改变系统的实现方式。为什么要做代码重构？为什么不投入精力来增加系统的功能，而是仅仅去改变系统的实现方式，这不是在浪费人力物力吗？

当一个精心设计的系统出来时，它具有良好的架构和优秀的编码。但是随着时间的发展，需要修改原有的缺陷和错误，需要增加新的功能和需求。为了适应这些变更，就需要修改原来的架构，于是代码的架构就可能慢慢地变得千疮百孔、不堪重负，最后成为一种制约而放弃。

重构就能最大限度地避免这样一种现象，当系统发展到一定阶段后使用重构，不改变系统的外部功能，只对内部的结构进行重新规划和整理。通过不断地调整系统的结构，使系统对于需求的变更始终具有较强的适应能力。

17.2 重构的好处

17.2.1 改进代码设计

当系统达到一定的规模时，必然包含大量的逻辑和复杂性，管理这些复杂性的手段就是要加以合理的组织，这时就要进行设计和重构。设计和重构是相辅相成、彼此互补的，通过重构，你仍然要做预先的设计，但是不必是最优的设计，只需要一个合理的解决方案就够了。所以重构可以带来更简单的设计，同时又不损失灵活性，从而降低设计的难度。

“不积跬步，何以至千里”，代码的重构可以先从“表面层次”上开始，把代码调整成更好的形式，让代码更容易被人理解。例如，清晰优美的形式、简明扼要的注释、含义清晰的命名，然后进一步消除冗余代码、抽取或合并代码、设计通用代码等。

17.2.2 发现代码缺陷

让代码能工作和让代码保持整洁是两种工作态度，很多人把精力用在让代码能工作，认为能工作就万事大吉了，从没有想过认真整理一下代码，优化代码结构，结果很多代码就停留在重复、冗余、臃肿、混乱的状态。如果没有重构，代码就会逐渐腐败变质，越来越像断线的风筝、脱缰的野马而失去控制。

“温故而知新”，代码重构就逼迫你加深对原设计的理解。很多程序员都有写下代码后，

却发生对自己的代码不甚理解的情景，这不是让人很尴尬和害怕吗？当你发生这样的情形时，通过代码重构就可以加深对原设计的理解，发现其中的问题和隐患，把它们消灭在萌芽状态，构建出更好的代码。

17.2.3 提高编程效率

当你发现解决一个问题变得异常复杂时，可能不是问题本身造成的，而是你用错了方法，拙劣的设计会导致臃肿的代码。

提高可读性、改善设计、优化结构、减少缺陷都是为了稳住阵脚。良好的设计是成功的一半，停下来通过重构改进设计，或许会减缓前进的速度，但它带来的后发优势却是不可低估的。

17.2.4 提高系统性能

程序设计需要考虑很多方面：代码空间、数据空间、运行效率、运行可靠性，代码的可读性和可维护性，既要平衡地考虑这些方面，又要重点考虑其他方面。

有些程序员过于关注程序的性能：由于太在乎细小的优化，他们编写出的程序过于精妙，难以维护。而另外一些程序员很少关注程序的性能：他们编写出的程序有着清晰漂亮的结构，但性能极低，以至于毫无用处。优秀的程序员把程序的效率纳入整体考虑中：性能只是系统中的众多问题之一，但有时候也很重要^[21]。

所以，通过代码重构和测试驱动开发二者相结合，首先编写出可调（tunable）的系统，然后调整以获得更好性能的系统。通过性能监测工具，找出系统中的性能热点（hot spot）所在的代码，然后优化它们，直到把性能提高到满意为止。同时，对系统各方面的评估，调整系统的代码，从而设计出空间、时间、效率上都满意的系统。

17.3 重构的难题

重构有很大的好处，可以给我们的工作带来唾手可得的改变，但是我们还要注意它的局限性。

- 1) 重构会修改接口。如果你对接口的修改不会对其他人带来影响，那么你的修改是可控的，而且不会带来太大的问题，否则，你对接口的修改就会带来很大的问题。所以要特别注意保证在系统内同步地修改接口。
- 2) 难以通过重构完成对系统的迁移。如果你想从一个系统构造出另一个系统，通过重构也许很困难，因为可能需要另起炉灶，重新设计系统。
- 3) 重构要选择时机。只有当代码在大部分情况下能够正常运行时，才能做代码重构。另外，如果项目已接近最后期限，也应该避免做代码重构。

17.4 实际的例子

笔者曾经用了一年的时间对一个 TF 卡系统中的 FTL 代码做重构，通过一步一步的修改代码和运行测试，最后设计出灵活可靠、运行稳定的 FTL 代码。

17.4.1 FTL 介绍

Nand Flash 是一种存储介质，由若干个块组成，而每块又包含若干页。Nand Flash 的特殊物理结构使得它无法像磁盘一样在原位置更新数据，因此需要在文件系统和物理层之间加入一个闪存转换层（Flash Translation Layer, FTL），解决文件系统的逻辑扇区地址与 Nand Flash 物理页地址之间的映射问题。

FTL 的主要功能包括：地址映射、坏块管理、垃圾回收、写入均衡、断电恢复。通过写入均衡算法，均衡管理每个块的写入和擦除次数，从而延长 Nand Flash 的使用寿命。

17.4.2 TF 卡系统

这个系统的硬件包括 MCU、Nand Flash 和基板，通过 MCP 方式把它们封装成 TF 卡。MCU 具有 SD 接口和 Nand Flash 接口，通过 SD 接口与手机或读卡器通信，通过 Nand Flash 接口存取 Nand Flash 中的数据。

MCU 内部还包含 32 位的 CPU、40KB 的 SRAM、4KB 的 Buffer、ECC 模块，另外还有一些特殊的模块。

FTL 代码是 MCU 固件（Firmware）的一部分，用于实现上面介绍的 FTL 功能。FTL 代码内部也分为两个层，一个是 Sector 层，负责 LBA 到 PA 的转换，另一个是 Flash 层，负责对 Nand Flash 的访问。

17.4.3 代码原始的状态

笔者在开始接手 FTL 代码时，代码中存在着如下问题。

- 1) 混杂大量无用的代码。原来的 Flash 层代码支持 3 种操作 Nand Flash 的方式：Normal、TwoPlane、InterleaveTwoPlane。表面上看来支持的方式灵活多样，但是实际上却没有太大的用处，InterleaveTwoPlane 的代码根本就没有使用过，已经没有维护的必要，Normal 的代码也是多余的，所有的读写操作都可以通过 TwoPlane 方式实现。
- 2) 混杂着没有测试的功能。MCU 提供一个 PPC_FDD 功能，但是没有仔细测试过，就扔在那里了。
- 3) 运行不稳定，有读写数据错的问题，有下电后再上电就出错的问题，有格式化后访问速度降低的问题
- 4) 存在一些重复的代码。其中有两处几乎完全相同的代码，代码的行数有 100 多行。
- 5) 有些函数非常大，可读性很差，阅读理解起来很费劲。
- 6) 编译时会有大量警告产生，很多警告的原因是类型不匹配和没有函数原型。
- 7) 只能支持 8GB 的 Nand Flash，其他的容量都不支持。
- 8) 与其他厂商的 TF 卡相比，读写速度还是要差一些。

17.4.4 代码半年后状态

笔者在同事的帮助下，逐步做了如下的修改，让系统能稳定工作。

- 1) 因为芯片本身的限制，难以支持 InterleaveTwoPlane，所以直接就把 InterleaveTwoPlane 相关的代码删除。
- 2) 通过软件代码绕过一个硬件 Bug，修改了几个软件缺陷，调整了一些宏定义参数，这

样解决运行不稳定和读写数据错的问题。

- 3) 通过提取公共代码, 抽取出一个大的函数, 从而去掉两大段重复的代码, 只须维护这个大的函数即可。
- 4) 为了提高 Nand Flash 的读写速度, 增加了 SMART_PAGE 功能。
- 5) 为了延长 Nand Flash 的使用寿命, 增加了 IDLE_BLOCK 功能。
- 6) 通过宏定义可以支持 8GB 和 4GB Nand Flash。
- 7) 消除编译时产生的警告, 让代码更清洁。
- 8) Flash 层的代码从 2800 行变到 1500 行。

17.4.5 代码现在的状态

在代码稳定运行数月后, 又提出了新的需求, 需要支持 16GB Nand Flash, 就是要支持 2 个 8GB 的 Nand Flash 芯片。

初始对 16GB 设想的解决方案是通过使用 2 个 Zone 的方式, 代码改动量不大, 尝试了一下, 可以工作, 但是有如下问题。

- 1) 因为管理 FTL 需要 SRAM 空间, 1 个 Zone 时需要 8KB, 2 个 Zone 时就需要 16KB。虽然 SRAM 有 40KB, 但是当 1 个 Zone 时就只剩下 14KB, 这 14KB 刚刚能满足现有的应用要求, 如果使用 2 个 Zone, 那么 SRAM 空间就所剩无几, 别的应用就不能再运行了。
- 2) 因为上电时需要重新建立映射表, 1 个 Zone 时需要 700ms, 2 个 Zone 时就需要 1400ms。1400ms 的时间过长, 在某些手机上不能满足要求。
- 3) 16GB 的方式如果还是通过宏定义的方式支持, 就需要对 4GB、8GB、16GB 编译 3 种不同 Bin 文件, 使用上容易出问题。

考虑到以上这些问题, 以及代码中还存在的的一些问题, 同时为了支持 Nand Flash 的多种连接方式 (4GB、8GB、16GB、32GB), 为了让代码更加灵活, 使用更加方便, 笔者对代码做了较大的改动。

- 1) 通过自动检测识别所连接的 Nand Flash 的类型、个数、方式, 不再使用宏定义方式, 这样只要编译出一个 Bin 文件即可。
- 2) 还是使用 1 个 Zone 的方式, 这样管理 FTL 只需要 8KB 的 SRAM 空间。
- 3) Flash 层的函数名和参数保持不变, 但是内部代码做了很大的更改, 大的函数做了拆分, 代码内部简单直接, 没有绕来绕去的代码, 阅读理解起来更加方便。
- 4) FTL 代码原来存在一些跨级函数调用, 破坏了函数的层次结构。现在保持代码设计的层级结构, 不做跨级函数调用, 不直接使用低层次函数, 但是对性能几乎没有什么影响。
- 5) 删除 PPC_FDD 相关的代码, 因为没有维护的必要。
- 6) 删除 Normal 相关的代码, 因为不再使用。
- 7) 把其他几个大的函数做了拆分, 进一步提高代码的可读性。
- 8) 仔细阅读代码, 修改了几个潜在的 Bug, 修改差劲的代码, 修改注释, 消除编译警告。
- 9) Flash 层的代码从 1500 行变到 900 行。

通过几个月的运行测试, 证明这些改动没有问题, 使用起来更加方便灵活, 而且大大地提高了代码的可读性, 便于未来的修改、扩充和移植。

第 18 章

消除重复

有一位老板曾经夸耀他公司一个工程师的编程能力：“他只要把一份模版代码拷过去，稍加修改，就可以完成一个新的模块。”我们很惊讶，一是惊讶于这位工程师具有如此清晰的思路，二是惊讶于除了这位工程师，还有谁能维护这样的代码，三是惊讶于这位老板竟然能说出这样的话。

其实，我们经常做着重复代码的事，把几行或一整段的代码从这里复制到那里，然后稍加修改，就变成了一份新的代码。但是通过复制、粘贴和修改代码，能产生容易理解、维护、重用的代码吗？这只是产生代码重复的一个例子，其实产生代码重复的原因是各种各样的，混乱的系统中总是充斥着各种各样的重复代码。

关于代码重复最著名的单词是 Kent Beck 的 “Once And Only Once”，就是说系统中代码的任何一个片断应当只出现一次，不管是一个算法、一个常量集合，还是用于阅读的文档，都应该只有一份。

18.1 代码重复的产生

代码重复产生的另外一个原因就是做得太多，XP（极限编程）有一个基本原则叫作 “You Arent Gonna Need It”，意思是说“只实现你真正需要的东西，不要去实现你预期需要的东西。”如果你现在去实现你认为将来需要的东西，它们并不一定就是你以后真正需要的东西，因为在现在的环境中你可能无法理解将来的东西究竟是什么样子，你还要浪费大量的时间去构造这些将来是否必需的功能，然后当你真正实现的时候就可能产生代码重复。

这样看来，产生代码重复的一个原因就是太懒惰，从而去复制、粘贴和修改代码，最后“差强人意”；另一个原因就是太勤快，从而做得太多，最后“过犹不及”；再一个原因就是沟通不够，如果有两个函数虽然命名不同但却拥有相同或相似的功能，往往是因为开发团队内部协调沟通不够造成的。不管怎样，只要代码中出现了重复，就说明代码的质量很差劲。

18.2 代码重复的后果

代码重复可能是系统设计中一切邪恶的根源，因为它会导致很多问题。

- 1) 当你修改代码时，需要修改很多重复的地方，一不小心就会遗漏。
- 2) 你不能很好地对代码进行性能优化，因为你需要修改很多地方。
- 3) 某些地方的代码会过期，导致系统内部不一致，或者不同系统之间的不一致。
- 4) 代码的责任会四处散开，导致代码难以理解，意味着额外的工作、风险和复杂度。

18.3 代码重复的解决

我们知道代码重复有很大的危害，那么在设计时就应该避免产生代码重复，还有就是既然产生了代码重复，那就尽力把它们消除掉。

代码设计有一个重要的原则：不要重复自己（Don't Repeat Yourself, DRY 原则），就是不要写重复的代码。DRY 原则要求在整个系统中每一个小的知识块只能发生一次，且每个知识块必须有一个单一、明确、权威的表征。

当你正在构建一个大型的系统时，你通常会被整体复杂性搞得不知所措。解决复杂性的最基本策略是将系统分成若干个容易处理的部分。你要将系统划分成子系统，每个子系统包含一些特定的功能。子系统往下再划分为模块，这样复杂性将被降低到单一职责（Single Responsibility）。模块就是构成系统功能的最小知识块，你只需要保证这些模块的功能不重复即可。

代码出现了重复，就意味着在抽象层次上出现了遗漏，就要想办法消掉重复，关键在于你如何能找到重复代码。代码设计的复杂性可能会使重复的代码表现为相似性而非完全的重复，有些相似性不能一眼就能看得出来，需要其他的重构步骤和一定的先见之明，才能看出来。

最明显的代码重复就是相同的代码，这些都是复制粘贴的产物，是很死板的重复；再一种代码重复是相似的代码，这些可能是粘贴修改的产物；还有一种代码重复是使用同样的算法但是编码不一样的代码，这些很可能是由不同人员实现的，由于人员之间的沟通不够而没有实现代码的共享。

找到重复的代码之后，下一步就要进行适当的抽象，然后创建通用函数，你可以使用，别人也可以使用。因为消除了代码重复，提升了抽象层次，所以错误会越来越来少，编码会越来越快。

但是，我们还要关注消除代码重复的尺度，大段的重复代码肯定值得去消除，那么几句的重复代码是否也值得去消除呢？代码重复消除的基本方法是建立单独的函数，如果系统是由太多的小函数构成，那么函数调用的开销就会相应地增加，同时也增加了维护的工作量。所以要掌握好消除代码重复的尺度，不要走向极端。

Martin Fowler 的《Refactoring》书中讲了很多用来处理代码重复的方法，主要是面向对象程序设计，如果有这方面需要可以读一下。

18.4 消除重复的例子

18.4.1 消除简单的代码重复

我们看下面的例子，代码重复是显而易见的。有些人可能觉得奇怪，难道真的有人会编写出这样重复的代码吗？事实就是如此，确实有人会编写出这样重复的代码。

例子：消除重复之前^[Andrew Hunt]

```
annuity = 0.0;
if (city == BEIJING) {
    rate = BJ_RATE;
    factor = base * BJ_RATE;
    total = total_bonus (factor) * 2.0 + total_extra (factor) * 1.20;
}
else if {city == GUANGZHOU || city == SHANGHAI) {
    rate = (city == GUANGZHOU ? GZ_RATE : SH_RATE);
    factor = base * rate;
```

```

total = total_bonus (factor) * 2.0 + total_extra (factor) * 1.20;
if (city == GUANGZHOU)
    annuity = 2000.0;
}
else {
    rate = 1;
    factor = base;
    total = total_bonus (factor) * 2.0 + total_extra (factor) * 1.20;
}

```

为了简化代码，增加一个 `rate_lookup` 数组，除 BEIJING、GUANGZHOU、SHANGHAI 之外，其他元素的值都是 1。这种方法使我们能很容易就适应未来的变化。

例子：消除重复之后

```

rate = rate_lookup[city];
factor = base * rate;
total = total_bonus (factor) * 2.0 + total_extra (factor) * 1.20;
annuity = 0.0;
if (city == GUANGZHOU)
    annuity = 2000.0;

```

18.4.2 获取文本行的简化

笔者在用 Perl 编写一个脚本时，脚本中有多处对文本文件进行处理的代码。处理文本是以行为单位进行处理，需要把以“#”或者“//”开始的注释去掉，需要跳过空行，所以初始时的代码如下。

例子：获取行的初始代码

```

while (my $line = <FILE_S>)
{
    chomp ($line);
    $line =~ s/(\\\/|\\#).*$//g;
    $line =~ s/^\s+//g;
    next if ($line =~ /^$/);
    .....
}

```

后来，笔者发现文本文件中的行太长，所以就支持使用“\”作为续行，同时增加了一个函数 `process_line()`，用于替换一部分重复的代码。

例子：获取行增加处理续行的功能

```

while (my $line = <FILE_S>)
{
    process_line ($prev, $line);
    next if ($line =~ /\$/);
    next if ($line =~ /^$/);
    .....
}

sub process_line (\$\$)
{
    my $prev = ${$_[0]};
    my $line = ${$_[1]};
    chomp ($line);
    $line =~ s/(\\\/|\\#).*$//g;
    $line =~ s/^\s+//g;
    $line = $prev.$line;
    if ($line =~ /(.*)\$/ ) {

```

```

        $prev = $1;
    } else {
        $prev = "";
    }
    ${_${0}} = $prev;
    ${_${1}} = $line;
}

```

因为笔者的脚本需要对多个不同的文件处理,存在多个不同的 `while` 循环,但是这些 `while` 循环的头部都是一样的,都是获取一行然后处理,所以笔者在想是否可以进一步简化,同时笔者看着 `$prev` 变量也很别扭。所以笔者仔细研究了一下,最后增加了一个函数,并简化 `while` 循环开头的获取行操作。

例子: 获取行独立成一个函数

```

while (my $line = fetch_whole_line (\*FILE_S))
{
    .....
}

sub fetch_whole_line
{
    my $fh = $_[0];
    my $line;
    my $prev = "";
    while ($line = <$fh>)
    {
        chomp ($line);
        $line =~ s/(\\\/|\\#).*$//g;
        $line =~ s/^\s+//g;
        $line = $prev.$line;
        if ($line =~ /(.*)\$/ ) {
            #The line is tailed with \, continue next part
            $prev = $1;
        } elsif ($line =~ /^$/) {
            #The line is empty, continue next line
        } else {
            last;
        }
    }
    return $line;
}

```

18.4.3 SD 命令的处理

在笔者参与的一个 TF 卡系统中,SD 接口接收命令后就要做相应的处理。在它接收的命令中,有 4 条这样的命令:

```

CMD_17, single_block_read
CMD_18, multiple_block_read
CMD_24, single_block_write
CMD_25, multiple_block_write

```

软件需要对它们做相应的处理,对应的代码是如下的样子。

例子: SD 命令处理代码的样子,初始样子

```

switch(cmd)
{
case 17:
    //Have 150 lines

```

```

        .....
        break;
case 18:
    //Have 150 lines
    .....
    break;
case 24:
    //Have 120 lines
    .....
    break;
case 25:
    //Have 120 lines
    .....
    break;
}

```

注意：//Have xxx lines 是笔者统计的代码行数。

笔者仔细地看了一下代码，发现 CMD_17 和 CMD_18 的代码很相似，而 CMD_24 和 CMD_25 的代码近乎相同，所以笔者就和设计者说了一下，设计者于是修改如下。

例子：SD 命令处理代码的样子，修改后

```

switch(cmd)
{
case 17:
    .....
    break;
case 18:
    .....
    break;
case 24:
case 25:
    if(cmdIndex == 25)
        gSD_StopTransfer = 0;
    .....
    break;
}

```

设计者只是对 CMD_24 和 CMD_25 的代码做了合并，可以看出 CMD_24 和 CMD_25 的代码都有 120 行，只是有 2 行不一样，但是在这之前设计者还是让这样重复的代码一直存在着。

设计者没有对 CMD_17 和 CMD_18 做进一步处理，估计是嫌麻烦，因为不同点的地方多了几处。其实作者希望设计者最终能把代码改成如下的样子。

例子：SD 命令处理代码的最终样子

```

switch(cmd)
{
case 17:
case 18:
    if(cmdIndex == 18)
        .....
        if(cmdIndex == 18)
            .....
    break;
case 24:
case 25:
    if(cmdIndex == 25)
        gSD_StopTransfer = 0;
    .....
    break;
}

```


“众里寻他千百度，蓦然回首，那人却在灯火阑珊处”，我们为了优化代码的结构，提高代码的可读性、可维护性和可移植性，我们还要不断努力。

我们写代码时，开始时的代码可能冗长又复杂，有很多的层次和嵌套循环，有很多重复代码，函数名也是很随意起的，但是我们需要进一步打磨代码，拆分函数、修改名称、消除重复，然后会获得更好的代码。

19.1 抽取独立的代码

国家行政机关有国务院、部委、省厅、市局、县分局、镇办事处等级别的划分，系统设计也具有不同的层次级别，越高级的层次就越抽象越概略，越低级的层次就越具体越细节。系统设计要划分出很好的层次级别，函数也要设计出很好的层次，既不能让层次跨越得太大，也不能让低层次的代码和高层次的代码混杂在一起。

如果函数过于庞大，做各种各样的事，有太多不同层次的抽象，有太多的嵌套层次，函数内部会充斥着很多的细节代码或者低层次代码，从而充斥着大量不相关的问题，没有充分表达函数的意图，会让读者陷入到这些细节中，而不是专注于函数本身的功能。

所以尽量不要把函数搞得太大，要把函数中的代码保持在一个抽象层次上，要把任务切分成子任务并独立出来，这样阅读时就会关注函数的本身功能，不会被细节分心，从而提高代码的可读性，同时让代码更加强壮。

我们要积极主动地发现并抽取独立的代码，可以按照如下的步骤检查大的函数^[20]。

- 1) 首先，确定一个函数的高层次目标是什么？这个目标应该从函数名就可以知道。
- 2) 其次，检查每一行/段代码的目标是什么？是直接为函数的高层次目标服务吗？是不是处于同一抽象级别上呢？
- 3) 最后，如果确定足够的行数在解决独立的子问题，那么就把这段代码独立到子函数中，而且要给这个子函数命名一个很好的名字。

通过这种方法，可以把大函数拆分为多个小函数，因为小函数的名字含义清晰，完成功能单一，代码短小紧凑，便于代码的修改和维护。大函数通过调用这些小函数完成功能，大函数的代码就都处在同一层次上，大函数不用在那些细枝末节上纠缠。所以使用这种方法可以大大地提高代码的可读性。

虽然这是一个很简单的方法，而且可以从根本上改进我们的代码，但是很多程序员并没有使用这个方法，导致一些函数庞大臃肿，为什么呢？

例子：函数中混杂的低层次代码

```
UINT32 end_offset, f_idx, f_off, p_idx, p_off;
```

```

UINT32 col_addr;
end_offset = nand_oper_ptr->end_offset;
f_idx = end_offset / FTL_PARTS_PER_CS;
f_off = end_offset % FTL_PARTS_PER_CS;
p_idx = f_off / FTL_PARTS_PER_PP;
p_off = f_off % FTL_PARTS_PER_PP;

```

例子：抽取出来成为独立的函数

```

void flash_calculate_factor (UINT32 offset, UINT32 *f_idx, UINT32 *f_off,
                             UINT32 *p_idx, UINT32 *p_off)
{
    *f_idx = offset / FTL_PARTS_PER_CS;
    *f_off = offset % FTL_PARTS_PER_CS;
    *p_idx = (*f_off) / FTL_PARTS_PER_PP;
    *p_off = (*f_off) % FTL_PARTS_PER_PP;
}

//call it
end_offset = nand_oper_ptr->end_offset;
flash_calculate_factor (end_offset, &f_idx, &f_off, &p_idx, &p_off);

```

这段代码的可读性就好多了，因为读者可以关注于高层次的目标，不必为低层次的代码分心。另外，`flash_calculate_factor()`很容易进行单独的测试，并且它是以后可以重用的函数。这就是为什么它是一个独立的子问题，因为它完全是自包含的，不用知道其他函数是如何使用它的。

函数中还有一些代码在那里只是为了支持其他代码，例如，为函数准备输入数据或者为函数的输出数据做后期处理，这些“粘附”代码常常和函数的目标没有太大关系，所以最好把这些代码也抽取到独立函数中。

注意：过多的小函数对代码的可读性很不利，因为读者需要关注更多的东西，在查看这些小函数时要跳来跳去。如果增加小函数对代码的可读性没有好处，而且这些小函数没有在别的地方使用，那么就没有必要增加这些小函数。

对于 Verilog 的模块也一样，可以把相对独立的功能独立出来。

例子：这是在设计 `arm_wrapper` 时抽取来的模块 `arm_dbg_syn`

```

module arm_dbg_syn
(
    input        FREE_CLK,
    input        HRESETn,
    input        DBGGEN,
    input        TCK,
    input        TMS,
    input        TDI,
    output       RTCK,
    output       DBGTCKEN,
    output reg   DBGTMS,
    output reg   DBGTDI
);
    wire latch_input;
    reg R_TCK_p1, R_TCK_p2, R_TCK_p3;
    always @(posedge FREE_CLK or negedge HRESETn)
        begin
            if (!HRESETn) begin
                R_TCK_p1 <= 0;
                R_TCK_p2 <= 0;
            end
        end

```

```

        R_TCK_p3 <= 0;
    end
    else if (DBGGEN) begin
        R_TCK_p1 <= TCK;
        R_TCK_p2 <= R_TCK_p1;
        R_TCK_p3 <= R_TCK_p2;
    end
end

assign RTCK = R_TCK_p3;
assign DBGTKEN = ~R_TCK_p2 && R_TCK_p3;
assign latch_input = R_TCK_p2 && ~R_TCK_p3;

always @(posedge FREE_CLK or negedge HRESETn)
begin
    if (!HRESETn) begin
        DBGTMS <= 1'b1;
        DBGTDI <= 1'b0;
    end
    else if (DBGGEN && latch_input) begin
        DBGTMS <= TMS;
        DBGTDI <= TDI;
    end
end
end
endmodule

```

19.2 设计通用的代码

编程语言通常都会有很多的库函数，例如 C 语言和 Perl 都有很多库函数，但是当编程语言没有某项复合功能时，就需要你自己编写一个函数。当你编写这个函数的时候，就要尽量考虑把它设计成一个通用独立的函数。

对这样通用独立的函数，你可以很方便地添加新功能、改进可读性、增加错误处理。经过一段时间之后，你就会有一组不错的通用函数。这些函数常常存放在一个专门的目录（例如 lib/）中，形成一个函数库，这样就可以很方便地在多个项目中重用它们。

一个系统的代码是由通用代码、辅助代码和专有代码组成，通用代码和辅助代码用来解决一般性问题，专有代码用于解决系统的专有问题，也就是让你的系统与众不同的核心部分。如果你能从你的系统中拆分出越多越好的通用代码，你的系统就会越小而且越易于维护，你就可以更加专注于系统本身。

例如，要实现读入文本文件到缓冲区的功能，C 语言库函数没有这项功能，所以就自己编写了一个。

```

例子：read_file, 读入文件到缓冲区
#include <stdio.h>
#include <stdlib.h>
char *read_file (char *file_name)
{
    FILE *fp = fopen (file_name, "r"); //Open as TEXT mode
    if (fp == NULL) {
        fprintf (stderr, "Error: open file %s\n", file_name);
        exit (1);
    }
    //Get the length of file
    fseek (fp, 0, SEEK_END);

```

```

int size = ftell (fp);
fseek (fp, 0, SEEK_SET);
//Allocate the buffer, +1 is used for the tail '\0'
char *buf = malloc (size + 1);
if (buf == NULL) {
    fprintf (stderr, "Error: malloc %d bytes\n", size);
    exit (1);
}
int pos = fread (buf, 1, size, fp);
//Set the tail '\0'. Note: Here pos is used instead of size, because on
Windows
// the file mode has TEXT and BINARY mode, and size maybe is not same
as pos.
buf[pos] = '\0';
return buf;
}

```

例如，作者使用 Perl 编写脚本时，需要一个与 C 语言的 `atoi()` 类似功能的函数，能转化十进制数和十六进制数（以 `0x` 开头），所以就自己编写了一个。

例子：Perl `atoi()`，可以转换十进制和十六进制数

```

sub atoi
{
    my $raw = $_[0];
    my $a = lc ($_[0]);
    my $multiply;
    my $start;
    my ($i, $v, $x);
    if (length ($a) > 2 && (substr ($a, 0, 2) eq "0x")) {
        $start = 2;
        $multiply = 16;
    } elsif (substr ($a, 0, 1) ge "0" && substr ($a, 0, 1) le "9") {
        $start = 0;
        $multiply = 10;
    } else {
        printf ("Error atoi -1: %s\n", $raw);
        exit (1);
    }
    $v = 0;
    for ($i = $start; $i < length ($a); $i++) {
        $v *= $multiply;
        $x = substr ($a, $i, 1);
        if ($multiply == 10 && !($x ge "0" && $x le "9")
            || $multiply == 16 && !(($x ge "0" && $x le "9")
                || ($x ge "a" && $x le "f"))) {
            printf ("Error atoi -2: %s\n", $raw);
            exit (1);
        }
        if ($x le "9") {
            $v += ord ($x) - ord("0");
        } else {
            $v += ord ($x) - ord("a") + 10;
        }
    }
    return $v;
}

```

例如，这是作者常用的一个跨时钟域同步 Pulse 的模块，它只是作者常用的多个同步模块中的一个。

例子: Verilog sync_pulse

```

module sync_pulse
  (input  in_rst_n,
   input  in_clk,
   input  in_pulse,
   input  out_rst_n,
   input  out_clk,
   output out_pulse);
  reg R_in_change;
  always @(posedge in_clk or negedge in_rst_n)
    begin
      if (!in_rst_n) R_in_change <= 0;
      else if (in_pulse) R_in_change <= ~R_in_change;
    end
  reg [2:0] R_out_change;
  always @(posedge out_clk or negedge out_rst_n)
    begin
      if (!out_rst_n) R_out_change <= 0;
      else R_out_change <= {R_out_change[1:0], R_in_change};
    end
  assign out_pulse = R_out_change[1] ^ R_out_change[2];
endmodule

```

19.3 简化已有的接口

人们都喜欢使用接口简洁的函数，这种函数参数少、设置不复杂、花费工夫少，因为这种函数让你的代码看起来简单且优雅。但是如果你所用的接口并不简洁，你是继续使用这样的接口呢，还是创建自己的简洁“包装”函数呢？通过对原来函数的重新包装，你可以得到简洁的接口，可以方便你的使用。

对于 Verilog 的模块也是同样的道理。作者在做 ASIC 设计中，需要使用大量的 IP 模块，这些模块的功能设计得大而全，具有复杂的端口和参数，直接使用会很复杂，而且不需要这么多的功能。所以作者就在这种模块的外面又加了一层，配置好参数，实例化必要的小模块，只把需要的端口连接出来，隐藏不需要使用的端口。

例子: ARM926 的 wrapper

```

module arm_wrapper
  (//-----
   //Interface with ATPG
   input          test_mode,
   input          test_se,
   //-----
   //Interface with CGU and INTC
   input          FREE_CLK,
   input          HRESETn,
   input          nIRQ,
   input          nFIQ,
   //-----
   //Interface with PAD
   input          nTRST,
   input          TCK,
   input          TMS,
   input          TDI,
   output         TDO,
   output         nTDOEN,
   output         RTCK,

```

```

//-----
//Interface with PAD and TCU/WDT/OST
input          DBGEN,
output        DBGACK,
//-----
//ARM AHB instruction fetch
input          IHCLKEN,
input          IHREADY,
input [1:0]    IHRESP,
input [31:0]   IHRDATA,
output        IHLOCK,
output [1:0]   IHTRANS,
output        IHWRITE,
output [2:0]   IHBURST,
output [2:0]   IHSIZE,
output [31:0]  IHADDR,
//-----
//ARM AHB data access
input          DHCLKEN,
input          DHREADY,
input [1:0]    DHRESP,
input [31:0]   DHRDATA,
output        DHLOCK,
output [1:0]   DHTRANS,
output        DHWRITE,
output [2:0]   DHBURST,
output [2:0]   DHSIZE,
output [31:0]  DHADDR,
output [31:0]  DHWDATA);
.....
ARM926EJS      uARM926EJS
  (.....);
arm_dbg_syn    arm_dbg_syn_i
  (.....);
arm_clock_ctrl arm_clock_ctrl_i
  (.....);
endmodule

```

另外，`arm_dbg_syn` 和 `arm_clock_ctrl` 两个辅助模块，它们的功能独立，所以就把它俩抽取出来做成单独的模块。

19.4 一次只做一件事

磁盘碎片指的是硬盘读写过程中产生的不连续存储的文件，磁盘碎片会显著地降低硬盘的运行速度，这是因为读取文件时需要在多个碎片之间跳转，增加了等待盘片旋转到指定扇区的时间和磁头切换磁道所需的寻道时间。通过执行整理碎片的软件，就可以把这些磁盘碎片收集在一起，再把它们作为一个连续的整体存放在硬盘上，从而提高读取文件的速度。

一段长的代码可能包含很多操作，如初始化、申请缓冲区、读入数据、处理数据、写出数据、释放缓冲区等操作，当这些操作像一团乱麻纠缠在一起时，它们就是同时在做几件事，很难说清每个任务是从哪里开始、到哪里结束的，导致代码的可读性大大降低。所以代码应该按照“一次只做一件事”的模式组织，就像给磁盘整理碎片一样，也要给混乱的代码“整理一下碎片”。

如何按照“一次只做一件事”的模式组织代码呢？首先列出代码中的所有任务（这里的

任务没有很严格的定义), 然后把这些任务拆分到不同的代码段中或者不同的函数中。

通过这样整理代码, 代码就会变得整洁合理、层次分明、条理清晰, 就可以一块一块地理解, 代码的逻辑性和可读性得到了很大的提高。

这种整理代码的方式, 不管是对 C 语言和 Perl 这样的串行执行语言, 还是对 Verilog 这样的并行语言, 都很有效果, 都可以提高代码的可读性, 对 Verilog 的代码尤其有效。

例子: 纠缠在一起的 Verilog 代码

```

module JTAG_TAP
  (input TCK, TMS, TRST,
   output CLOCKDR, CLOCKIR, ENABLE, RESET, SEL, SHIFTD, SHIFDIR, UPDATEDR,
   UPDATEIR);
  reg [3:0] R_tap_state, next_state;
  reg      R_reset, R_shiftd, R_shifdir, R_enable;
  assign SEL = R_tap_state[3];
  assign SHIFDIR = ~R_shiftd; //Active high
  always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_reset <= 1'b0;
    else R_reset <= ~(R_tap_state == `JTAP_RESET);
  end
  assign SHIFTD = ~R_shiftd; //Active high
  always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_shiftd <= 1'b0;
    else R_shiftd <= ~(R_tap_state == `JTAP_SftIR);
  end
  assign RESET = R_reset; //Active low
  always @(posedge TCK or negedge TRST) begin
    if (~TRST) R_tap_state <= `JTAP_RESET;
    else R_tap_state <= next_state;
  end
  assign ENABLE = ~R_enable; //TDO output enable, active low
  assign UPDATEDR = ~TCK & (R_tap_state == `JTAP_UpdDR);
  always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_shiftd <= 1'b0;
    else R_shiftd <= ~(R_tap_state == `JTAP_SftDR);
  end
  assign CLOCKDR = TCK | ~((R_tap_state == `JTAP_CapDR) || (R_tap_state ==
`JTAP_SftDR));
  always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_enable <= 1'b0;
    else R_enable <= ((R_tap_state == `JTAP_SftIR) || (R_tap_state ==
`JTAP_SftDR));
  end
  assign CLOCKIR = TCK | ~((R_tap_state == `JTAP_CapIR) || (R_tap_state ==
`JTAP_SftIR));
  assign UPDATEIR = ~TCK & (R_tap_state == `JTAP_UpdIR);
  always @(TMS or R_tap_state) begin
    next_state = R_tap_state;
    if (TMS) begin
      case (R_tap_state)
        `JTAP_RESET: next_state = R_tap_state;
        `JTAP_IDLE: next_state = `JTAP_SelDR;
        `JTAP_SelDR: next_state = `JTAP_SelIR;
        `JTAP_CapDR: next_state = `JTAP_Ex1DR;
        `JTAP_SftDR: next_state = `JTAP_Ex1DR;
        `JTAP_Ex1DR: next_state = `JTAP_UpdDR;
        `JTAP_PusDR: next_state = `JTAP_Ex2DR;
        `JTAP_Ex2DR: next_state = `JTAP_UpdDR;
        `JTAP_UpdDR: next_state = `JTAP_SelDR;
      end
    end
  end
endmodule

```

```

`JTAP_SelIR: next_state = `JTAP_RESET;
`JTAP_CapIR: next_state = `JTAP_Ex1IR;
`JTAP_SftIR: next_state = `JTAP_Ex1IR;
`JTAP_Ex1IR: next_state = `JTAP_UpdIR;
`JTAP_PusIR: next_state = `JTAP_Ex2IR;
`JTAP_Ex2IR: next_state = `JTAP_UpdIR;
`JTAP_UpdIR: next_state = `JTAP_SelDR;
    endcase
end
else begin
    case (R_tap_state)
        `JTAP_RESET: next_state = `JTAP_IDLE;
        `JTAP_IDLE: next_state = R_tap_state;
        `JTAP_SelDR: next_state = `JTAP_CapDR;
        `JTAP_CapDR: next_state = `JTAP_SftDR;
        `JTAP_SftDR: next_state = R_tap_state;
        `JTAP_Ex1DR: next_state = `JTAP_PusDR;
        `JTAP_PusDR: next_state = R_tap_state;
        `JTAP_Ex2DR: next_state = `JTAP_SftDR;
        `JTAP_UpdDR: next_state = `JTAP_IDLE;
        `JTAP_SelIR: next_state = `JTAP_CapIR;
        `JTAP_CapIR: next_state = `JTAP_SftIR;
        `JTAP_SftIR: next_state = R_tap_state;
        `JTAP_Ex1IR: next_state = `JTAP_PusIR;
        `JTAP_PusIR: next_state = R_tap_state;
        `JTAP_Ex2IR: next_state = `JTAP_SftIR;
        `JTAP_UpdIR: next_state = `JTAP_IDLE;
    endcase
end
end
endmodule

```

例子：重新整理的代码，同时适当地添加一些空行

```

module JTAG_TAP
    (input TCK, TMS, TRST,
     output CLOCKDR, CLOCKIR, ENABLE, RESET, SEL, SHIFDR, SHIFIR, UPDATEDR,
     UPDATEIR);

    reg [3:0] R_tap_state, next_state;
    always @(TMS or R_tap_state) begin
        next_state = R_tap_state;
        if (TMS) begin
            case (R_tap_state)
                `JTAP_RESET: next_state = R_tap_state;
                `JTAP_IDLE: next_state = `JTAP_SelDR;

                `JTAP_SelDR: next_state = `JTAP_SelIR;
                `JTAP_CapDR: next_state = `JTAP_Ex1DR;
                `JTAP_SftDR: next_state = `JTAP_Ex1DR;
                `JTAP_Ex1DR: next_state = `JTAP_UpdDR;
                `JTAP_PusDR: next_state = `JTAP_Ex2DR;
                `JTAP_Ex2DR: next_state = `JTAP_UpdDR;
                `JTAP_UpdDR: next_state = `JTAP_SelDR;

                `JTAP_SelIR: next_state = `JTAP_RESET;
                `JTAP_CapIR: next_state = `JTAP_Ex1IR;
                `JTAP_SftIR: next_state = `JTAP_Ex1IR;
                `JTAP_Ex1IR: next_state = `JTAP_UpdIR;
                `JTAP_PusIR: next_state = `JTAP_Ex2IR;
                `JTAP_Ex2IR: next_state = `JTAP_UpdIR;
                `JTAP_UpdIR: next_state = `JTAP_SelDR;
            endcase
        end
    end
endmodule

```



```

        endcase
    end
    else begin
        case (R_tap_state)
            `JTAP_RESET: next_state = `JTAP_IDLE;
            `JTAP_IDLE: next_state = R_tap_state;

            `JTAP_SelDR: next_state = `JTAP_CapDR;
            `JTAP_CapDR: next_state = `JTAP_SftDR;
            `JTAP_SftDR: next_state = R_tap_state;
            `JTAP_Ex1DR: next_state = `JTAP_PusDR;
            `JTAP_PusDR: next_state = R_tap_state;
            `JTAP_Ex2DR: next_state = `JTAP_SftDR;
            `JTAP_UpdDR: next_state = `JTAP_IDLE;

            `JTAP_SelIR: next_state = `JTAP_CapIR;
            `JTAP_CapIR: next_state = `JTAP_SftIR;
            `JTAP_SftIR: next_state = R_tap_state;
            `JTAP_Ex1IR: next_state = `JTAP_PusIR;
            `JTAP_PusIR: next_state = R_tap_state;
            `JTAP_Ex2IR: next_state = `JTAP_SftIR;
            `JTAP_UpdIR: next_state = `JTAP_IDLE;
        endcase
    end
end

always @(posedge TCK or negedge TRST) begin
    if (~TRST) R_tap_state <= `JTAP_RESET;
    else R_tap_state <= next_state;
end

reg R_reset;
always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_reset <= 1'b0;
    else R_reset <= ~(R_tap_state == `JTAP_RESET);
end
assign RESET = R_reset; //Active low

reg R_shiftir;
always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_shiftir <= 1'b0;
    else R_shiftir <= ~(R_tap_state == `JTAP_SftIR);
end
assign SHIFTIR = ~R_shiftir; //Active high

reg R_shiftdr;
always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_shiftdr <= 1'b0;
    else R_shiftdr <= ~(R_tap_state == `JTAP_SftDR);
end
assign SHIFTD R = ~R_shiftdr; //Active high

reg R_enable;
always @(negedge TCK or negedge TRST) begin
    if (~TRST) R_enable <= 1'b0;
    else R_enable <= ((R_tap_state == `JTAP_SftIR) || (R_tap_state ==
`JTAP_SftDR));
end
assign ENABLE = ~R_enable; //TDO output enable, active low

assign SEL = R_tap_state[3];

```

```

        assign CLOCKDR = TCK | ~((R_tap_state == `JTAP_CapDR) || (R_tap_state ==
`JTAP_SftDR));
        assign CLOCKIR = TCK | ~((R_tap_state == `JTAP_CapIR) || (R_tap_state ==
`JTAP_SftIR));

        assign UPDATEDR = ~TCK & (R_tap_state == `JTAP_UpdDR);
        assign UPDATEIR = ~TCK & (R_tap_state == `JTAP_UpdIR);
    endmodule

```

19.5 让函数功能单一

在设计函数的时候，有些人总是经不住诱惑，让函数多一些功能，多做一些事，觉得应该“能者多劳”。但是实际上应该尽量让函数功能单一，就是让函数干好一件事，不要给函数赋予太多的功能，这也是面向对象设计中的单一职责原则。

如果一个函数包含太多的功能，代码就会变得混乱不堪，很难维护和管理。所以当一个大函数包含太多的功能时，就要考虑把它拆分成多个小函数，每个小函数都有单一明确的功能。

另外，当实现一个功能的时候，不要把这个功能分散到多个函数内进行处理，应该将这个功能都放到一个函数内处理。当发现逻辑碎片产生时，应当及时加以清理，将分散的代码合并。

所以，“让函数功能单一”具有两种意思：一是不要往函数里塞各种东西，就是不要“太拥挤”；二是不要把一个功能分散到多个函数，就是不要“太分散”。

所以，当定义一个函数的时候，首先要给函数起个意义清晰的名字，然后编写代码实现函数的功能，最后还要检查是否做到名实相符，就是要检查名字和功能是否相对应。另外，注意函数不要有太多的参数，因为太多的参数会让函数变得低内聚高耦合，会让函数的编写变复杂，也会让函数的使用变复杂。

在一个嵌入式系统中，为了实现更快速的内存复制和内存填充，设计者没有使用库函数 `memcpy` 和 `memset`，另外重写了一个函数 `fast_memory_copy`。这个函数可以每次复制或填充 8 个 word，下面就是对应的代码。

```

例子：原来的 fast_memory_copy
void fast_memory_copy (void *dst, void *src, int n)
{
    if (n <= 0) return;
    int i;
    if (n & 0x8000)
    {
        int *p_d = dst;
        int tmp = (int)src;
        n &= ~0x8000;
        for (i = 0; i < n; i += 8)
        {
            p_d[i+0] = tmp;
            p_d[i+1] = tmp;
            p_d[i+2] = tmp;
            p_d[i+3] = tmp;
            p_d[i+4] = tmp;
            p_d[i+5] = tmp;
            p_d[i+6] = tmp;
        }
    }
}

```

```

        p_d[i+7] = tmp;
    }
}
else
{
    int *p_d = dst;
    int *p_s = src;
    for (i = 0; i < n; i += 8)
    {
        p_d[i+0] = p_s[i+0];
        p_d[i+1] = p_s[i+1];
        p_d[i+2] = p_s[i+2];
        p_d[i+3] = p_s[i+3];
        p_d[i+4] = p_s[i+4];
        p_d[i+5] = p_s[i+5];
        p_d[i+6] = p_s[i+6];
        p_d[i+7] = p_s[i+7];
    }
}
}

```

这个函数有这些问题：一是函数功能不单一，包含复制和填充两项功能；二是函数名不好，`fast_memory_copy` 让人误解；三是使用不便，通过 `if (n & 0x8000)` 判断是复制还是填充。也许有人不相信，这确实是现实存在的代码。

所以需要把这个函数拆分成两个函数 `fast_memory_set` 和 `fast_memory_copy`，代码如下所示。

例子：修改后的 `fast_memory_set` 和 `fast_memory_copy`

```

void fast_memory_set (void *dst, int val, int n)
{
    if (n <= 0) return;
    int *p_d = dst;
    int i;
    for (i = 0; i < n; i += 8)
    {
        p_d[i+0] = val;
        p_d[i+1] = val;
        p_d[i+2] = val;
        p_d[i+3] = val;
        p_d[i+4] = val;
        p_d[i+5] = val;
        p_d[i+6] = val;
        p_d[i+7] = val;
    }
}

void fast_memory_copy (void *dst, void *src, int n)
{
    if (n <= 0) return;
    int *p_d = dst;
    int *p_s = src;
    int i;
    for (i = 0; i < n; i += 8)
    {
        p_d[i+0] = p_s[i+0];
        p_d[i+1] = p_s[i+1];
        p_d[i+2] = p_s[i+2];
        p_d[i+3] = p_s[i+3];
        p_d[i+4] = p_s[i+4];
        p_d[i+5] = p_s[i+5];
    }
}

```

```
    p_d[i+6] = p_s[i+6];  
    p_d[i+7] = p_s[i+7];  
  }  
}
```

19.6 删除无用的代码

无用的代码主要有两种表现形式，一种是用注释或者“`#if 0`”标注的无用代码，不会编译到可执行文件中，所以不会增加代码的长度；另一种是编译到可执行文件中但是从不执行的代码，所以会增加代码的长度，因为永远执行不到，所以就是死代码。

总体来说，无用的代码包含：无用的注释、注释掉的代码、无用的全局变量、无用的局部变量、从不调用的函数、从不执行的代码，这些都会给代码带来混淆，都是应该删掉的废物。

对于这些无用的代码，都要做到定期检查清理，保持代码的整洁干净，否则代码的可读性会降低，代码会因此腐败变臭。

第 20 章

少写代码

你所写的每一行代码都需要测试和维护，通过重用代码或者减少功能，以此让你的代码保持精简节约，可以节省开发和维护的时间。对于一个程序员来说，因为“懒惰是程序员的美德”，所以知道什么时候不写代码是一个必须要学的技巧。

20.1 合适就刚刚好

有这样一个故事，可能是杜撰的：某企业引进一条香皂包装生产线，结果发现经常有空盒流过。厂长请一个博士花了 200 万设计出了自动分检系统。某乡镇企业也遇到同样问题，农民工花 90 元买一大电扇放在生产线旁，有空盒经过便被吹走。

当你要设计一个系统的时候，你要分析系统的需求，然后让设计满足要求即可。不是所有的程序都要求运行得快，要求 100% 正确，或者要求能处理所有的情况，只要满足需求、满足使用即可。如果你仔细检查系统的需求，有时你可以把它削减为一个简单的问题，只需要很少的代码。但是，现实存在两种情况：高估设计和低估设计。

高估设计，最后的结果可能是“过犹不及”，就是事情做得过头，就跟做得不够一样。因为程序员会倾向于实现大而全的功能，甚至实现一些很酷的功能，结果很多功能没有完成，或者没有用到，还让代码变得更加复杂。

低估设计，最后的结果可能是“差强人意”，就是大体上能使人满意，勉强还行。因为程序员还倾向于低估系统的需求，低估实现系统所要的时间，结果实现的系统不满足需求，或者仓促地实现了一个很粗糙的系统。

20.2 保持代码简洁

随着项目的进展，系统中会增加越来越多的函数、文件、目录，函数之间的调用关系很难记住，跟踪 Bug 也很费劲。系统变得越来越庞大，已经没有一个人能全部理解，使用这些代码变得很费劲，修改代码、增加新功能也变得很痛苦^[20]。

最好的方法就是把代码变得越简洁、越轻量越好。

- 1) 减少无用的代码和无用的功能，以保持代码干净整洁。
- 2) 创建更多更好的通用代码，减少重复的代码，要尽量重用代码。
- 3) 让你的项目保持处于分开的子项目状态。

有些程序员一旦写出代码，就不愿删除它们，因为觉得它们代表着很多的实际工作，删除它们就意味着之前所花的时间是浪费。千万不要这么想！删除它们意味着减少以后的工作

量，可以提高代码的可读性，可以提高系统的运行性能。删除独立的无用函数很简单，但是删除有用函数中的无用代码可能就有些费劲，因为你要仔细甄别，要避免把有用的代码删除。

为什么重用代码很重要？因为程序员的效率并不高，程序员平均下来也就每天 10 行代码，注意这 10 行代码是指最终产品里的代码，这些代码代表着大量的设计、测试、修改、优化和文档。这就是重用代码的好处，不仅节省时间，而且减少错误。

ASIC/FPGA 设计中的众多 IP 模块，就是通用代码的最好例子，它们都与系统保持独立，单独进行设计、仿真和测试。为了保证 IP 的灵活性和通用性，这些 IP 一般都做成静态配置或者动态配置，这样在不同的 ASIC 设计中根据需要使用不同的配置。例如，某 DMAC IP 的通道数可以选择 1~8，每个通道的 FIFO 大小选择 16/32/64/128 等。

20.3 使用循环解决

凡是能用循环解决的问题，就应该考虑用循环解决，因为循环能减少代码量，减少书写的错误，让代码更加紧凑。例如，对于下面的数组初始化，如果一个一个地初始化，这么多行，不容易检查出书写错误，但是用循环初始化，很容易就看出来错没错。

例子：数组初始化

```
//Idiot init
ar[0] = 0;
ar[1] = 1;
.....
ar[99] = 99;
//Loop init
for (i = 0; i < 100; i++)
    ar[i] = i;
```

对于 C 语言，这是大家都明白的道理，但是对于 Verilog，有些人在设计用于 ASIC 或 FPGA 的模块时，就停留在一个一个的书写上了，因为他们担心使用循环的模块不能综合出实际的电路。其实他们无须担心，因为现在的综合工具很强大，只要循环次数是常量，for 循环就是可以综合的。当综合工具遇到循环语句时，就会把它们展开成若干条顺序执行的语句，然后再综合出实际的电路。注意：为了使用 for 循环，必须要把信号定义成向量或数组。

例子：DMAH_NUM_CHANNELS 是一个宏定义的参数，可以根据配置修改。最好的办法就是用 for 循环，若一条一条地书写，根本就不可行。

```
always @(src_hs_pol or dma_last_src or ch_en_int)
begin
    integer i;
    for (i = 0; i < `DMAH_NUM_CHANNELS; i = i + 1)
        if(src_hs_pol[i])
            dma_lst_src_pol[i] = ~dma_last_src[i] && ch_en_int[i];
        else
            dma_lst_src_pol[i] = dma_last_src[i] && ch_en_int[i];
end
```

例子：使用 for 循环实现优先级解码器。这里 NUMBER 怎么变化都没关系，代码也不需要像用 casez 一样需要修改。

```
parameter NUMBER = 8;
localparam WIDTH=$clog2(NUMBER);
reg [NUMBER-1:0] intc_src;
reg [WIDTH:0]    intc_number;
reg              flag;
```

```

always @(*)
begin
    intc_number = (1'b1 << WIDTH);
    flag = 1;
    for (i = 0; flag && (i < NUMBER); i = i + 1)
        begin: intc_number_block
            if (intc_src[i] == 1) begin
                intc_number = i;
                flag = 0;
            end
        end
    end
end
end

```

当你对一个模块要实例化多次的时候，可以使用 `generate` 语句。

例子：使用 Verilog-2001 提供的 `generate` 语句

```

generate
for (i = 0; i <= 14; i = i + 1)
    begin: gen_pad_DDR_a
        PDDR0S x (.PAD(px_DDR_A[i]), .I(pp_DDR_A_o[i]), .OEN(1'b0), .SSEL(pp_
DDR_A_ds));
    end
endgenerate

```

20.4 熟悉语言特性

每种语言都有特有的语言特性、最新的语言标准和丰富的库函数，我们好好地掌握这些东西，深入到语言内部，这样才能更好地编写代码。

C 语言既具有高级语言的特点，又具有汇编语言的特点。它既可以用于编写系统程序，也可以用于编写应用程序。例如，C 语言可以把变量声明为 `volatile`，这个在嵌入式软件中经常用到，但是很多人没有用过，或者用了却不知道为什么要用；C 语言标准有 C89 和 C99，但是很多人还没有用上 C99 的混合声明。

Perl 是一种脚本语言，一般被称为“实用报表提取语言”，它吸取了很多语言的特性，包含正则表达式、强有力的函数、强有力的数据类型等。如果你用 Perl 编写一个处理文本的程序，你可以把代码写得像 C 语言的语句一样，但是这样的代码非常低效，因为你没有充分发挥 Perl 的特性，你必须灵活运用正则表达式和强有力的数据类型等特性，你才能编写出高效的代码。

Verilog 作为一种硬件描述语言，它也是优美、丰富而又简洁的语言，只有合理灵活地运用，才能充分表达你的设计意图，才能设计出强壮、简洁的代码。但是 Verilog 也有一些容易混淆的地方，若不注意就会写出错误的语句，如 `task` 和 `function` 的差别、阻塞赋值和非阻塞赋值、敏感列表不全等。

Verilog 标准有 Verilog-1995 和 Verilog-2001。Verilog-2001 增加了 `generate` 语句、`always@(*)`、`+:`、新的端口声明方式等新特性，可以编写出更加简洁的代码。

20.5 熟悉库函数

有的程序员费了半天劲解决了一个问题，还挺高兴的，但是却不知道库函数可以轻松解决这个问题，你说让人沮丧不？所以你需要了解库函数，以方便使用它们，这一点很重要。

每隔一段时间，你最好花时间浏览一下所有的库函数，你无须记录这些库函数的太多细节，只须在你要用某项功能的时候，能想到哪些库函数能提供这项功能，然后查找手册并使用它们，这个就叫做“有备无患”。

例如，要编写一个处理数据脚本，它可以接收如下的命令行：

```
gen_cfg.pl -src sm4_enc_config.txt -src sm4_sbox.txt -dst sm4_enc.hex
```

如果你不知道有库函数能够处理命令行，那么就要自己实现，但是如果你知道有库函数能够很好地处理命令行，那么你就肯定会直接使用库函数。对比下面的代码，就知道使用库函数的好处了，因为 20 行代码直接就被一条 `GetOptions(...)` 代替。

例子：自己编写代码处理命令行

```
#!/usr/bin/perl
use strict;
use warnings;
my @src_file = ();
my $dst_file = "";
my $flag = "";
for (my $i = 0; $i < scalar (@ARGV); $i++) {
    if ($ARGV[$i] eq "-src") {
        $flag = "src";
        continue;
    }
    elsif if ($ARGV[$i] eq "-dst") {
        $flag = "dst";
        continue;
    }
    else {
        if ($flag eq "src") {
            push (@src_file, $ARGV[$i]);
        } elsif ($flag eq "dst") {
            $dst_file = $ARGV[$i];
        } else {
            printf ("Error: %s\n", $ARGV[$i]);
            exit (1);
        }
    }
}

if (scalar (@src_file) == 0 || $dst_file eq "") {
    show_help ();
    exit (1);
}
#other codes.....
```

例子：使用 `GetOptions` 处理命令行

```
#!/usr/bin/perl
use strict;
use warnings;
use Getopt::Long;
my @src_file = ();
my $dst_file = "";
GetOptions ("src=s" => \@src_file, "dst=s" => \$dst_file);
if (scalar (@src_file) == 0 || $dst_file eq "") {
    show_help ();
    exit (1);
}
#other codes.....
```


Perl 提供 `GetOptions()`，C 语言有 `getopt()`和 `getopt_long()`，Bash 也有 `getopts`，而且用法都差不多。

20.6 熟悉系统工具

很多人在 Linux 上工作，除了简单的几条命令，很多命令都不知道怎么使用，Shell 和 Perl 脚本也不会写，根本就没有发挥 Linux 的特性。

有的人说，“我只是在这上面编写某语言程序，我为什么还要熟悉这些东西？”你熟悉了你的运行环境后，就可以使用编写 Shell 或 Perl 脚本做一些辅助性的工作，例如进行数据处理，编写自动测试环境，这些脚本不仅可以加速你的工作，还能够规范你的工作。

即使你不在 Linux 下工作，你也可以在 Windows 上安装 Cygwin 和 ActivePerl，然后可以做类似于 Linux 下的工作。

最常用的 Shell 有 `bash`、`sh`、`ksh`、和 `csk`，你最好使用 `bash`，因为 Bash 是 Linux 预设的 Shell，与 `sh` 和 `ksh` 兼容，而且介绍 `bash` 使用的书更多一些。

例子：在当前目录的 Verilog 文件中过滤出含有 `tcu_wake_n` 的行，并且包含对应的行号

```
grep -n tcu_wake_n *.v
```

例子：批量把所有的 `tcu_wake_n` 替换为 `gpio_wake_n`

```
for f in `find . -name '*.v'`
do
    sed -e 's/tcu_wake_n/gpio_wake_n/g' $f >$f.tmp
    mv -f $f.tmp $f
done
```

例子：统计一个文件中的单词个数

```
cat xxx.txt | tr '!"?:\[\]\{\}(),.\t\n' ' ' | tr 'A-Z' 'a-z' | tr -s ' ' | tr
' ' '\n' | sort | uniq -c | sort -rn
//下面的命令更清晰一些，因为使用了字符组
cat xxx.txt | tr '[:punct:]' ' ' | tr '[:space:]' ' ' | tr 'A-Z' 'a-z' | tr
-s ' ' | tr ' ' '\n' | sort | uniq -c | sort -rn
```

第 21 章

简化代码

为了让代码简单，必须要思考代码复杂在什么地方，想一下是不是原来就搞复杂了？也许本来就没有那么复杂，想一下怎样能让代码更简单一些？通常情况下，你需要跳出原来的圈圈，需要重新深入地考虑一下问题，考虑是否可用一种完全不同的方案来简化代码。

当你把一件复杂的事情向别人描述时，那些小细节很容易就会让他们迷惑。但是当你把一个想法用“自然语言”向别人描述时，很多普通人也能理解它。所以你需要把一个想法提炼出核心内容，不再关注那些小细节，这样不仅可以帮助他人理解这个想法，而且也帮助你自己想得更清晰。当你的代码看着别扭，不伦不类，自己都说不明白时，你就可以用自然语言或伪代码重新描述一下，重新整理一下思路，寻找更加优美整洁的解决方案，然后再用代码实现，这时你的代码就会变得更简单、更易读、更具有表达力。

Antoine de Saint-Exupéry 是法国作家、飞行员，也是童话《小王子》的作者，他曾经说过：“设计者确定其设计已经达到完美的标准不是不能再增加任何东西，而是不能再减少任何东西。”程序员应该使用该标准来检验自己完成的程序。简单的程序通常比具有相同功能的复杂程序更可靠、更安全、更健壮、更高效，而且更易于维护^[21]。

但是，简单又可靠的算法不是凭空从天上掉下来的，你需要多阅读书籍，多从网上查找资料，多和同事交流想法，说不定就能让你脑洞大开，让你想出更简单的算法。有一些讨论算法的书，讨论数据组织、数据排序、数据搜索和性能优化等问题，例如《数据结构》、《算法基础》、《编程珠玑》。当笔者阅读《编程珠玑》时会觉得脑洞大开，噢，原来可以这么想，真是简单又奇妙。

21.1 重新设计代码

作者在研究 FTL 代码时，当操作 Nand Flash 的时候，需要把操作的地址发给 Nand Flash，Nand Flash 可以接受地址字节的个数为 1、2、3 和 5，在每个 cycle 可以向 Nand Flash 写一个地址字节。当字节个数是 1 和 2 时，地址字节只是 column 地址，当是 3 时，地址字节只是 row 地址；当是 5 时，地址字节就包含 column 和 row 地址。原始代码在处理 5 个字节的时候，在调用 FLASH_WriteAddress 时要把 row 地址拆成两部分，分别放到 addr1 和 addr2，然后在这个函数内又通过移位恢复，所以从代码上讲绕来绕去，理解起来还费事。

例子：原始代码

```
void FLASH_WriteAddress (UINT32 addr1, UINT8 addr2, UINT8 cycle)
{
    UINT32 Address;
    switch(cycle)
    {
```

```

case 1:
    Address = (addr1 & 0xFF);
    FLASH_WriteCMD (FLASH_ADR_MASK | Address);
    break;
case 2:
    //column address
    Address = (addr1 & 0xFFFF);
    FLASH_WriteCMD (FLASH_ADR_MASK | FLASH_ADDR_LATCH_CNT_0 | Address);
    break;
case 5:
    //column address
    Address = (addr1 & 0xFFFF);
    FLASH_WriteCMD (FLASH_ADR_MASK | FLASH_ADDR_LATCH_CNT_0 | Address);
    addr1 >>= 16;
    //row address
    addr1 += (addr2 << 16);
    /* Following Issue Page & Block Address. */
case 3:
    // Page & Block Address
    Address = (addr1 & 0xFFFFFFFF);
    FLASH_WriteCMD (FLASH_ADR_MASK | FLASH_ADDR_LATCH_CNT_1 | Address);
    break;
default:
    break;
}
}
//函数调用
FLASH_WriteAddress (col_addr, 0, 1);
FLASH_WriteAddress (nandOperPtr->rRowAddr[1], 0, 3);
FLASH_WriteAddress (((nandOperPtr->wRowAddr[0] << 16) | colAddr),
(nandOperPtr->wRowAddr[0] >> 16), 5); //这里通过移位把 wRowAddr[0]拆成两部分,折腾

```

针对上面的情况，作者把 `addr1` 和 `addr2` 改为 `col_addr` 和 `row_addr`，当字节个数是 1 和 2 时，只使用 `col_addr`；当是 3 时，只使用 `row_addr`；当是 5 时，`col_addr` 和 `row_addr` 都用。这样就不需要移位，而且用起来更加直接。

例子：修改代码

```

void FLASH_WriteAddress (UINT32 col_addr, UINT32 row_addr, UINT32 cycle)
{
    UINT32 Address;
    switch (cycle)
    {
    case 1:
        //One address
        Address = (col_addr & 0xFF);
        FLASH_WriteCMD (FLASH_ADR_MASK | Address);
        break;
    case 2:
        //Column address
        Address = (col_addr & 0xFFFF);
        FLASH_WriteCMD (FLASH_ADR_MASK | FLASH_ADDR_LATCH_CNT_0 | Address);
        break;
    case 5:
        //Column address
        Address = (col_addr & 0xFFFF);
        FLASH_WriteCMD (FLASH_ADR_MASK | FLASH_ADDR_LATCH_CNT_0 | Address);
        //Continue sending the row address
    case 3:
        //Row address (page and block address)
        Address = (row_addr & 0xFFFFFFFF);

```

```

        FLASH_WriteCMD (FLASH_ADR_MASK | FLASH_ADDR_LATCH_CNT_1 | Address);
        break;
    default:
        break;
    }
}

//函数调用，使用很直接
FLASH_WriteAddress (col_addr, 0, 1);
FLASH_WriteAddress (0, nandOperPtr->rRowAddr[0], 3);
FLASH_WriteAddress (col_addr, nandOperPtr->rRowAddr[p_idx], 5);

```

另外，从修改的代码上看，case 2 和 case 5 还可以合并一下，只不过当 cycle = 2 时，就不能继续执行下面的代码，所以可以进一步改为如下，但是好像没有太大的必要。

```

case 2:
case 5:
    //Column address
    Address = (col_addr & 0xFFFF);
    FLASH_WriteCMD (FLASH_ADR_MASK | FLASH_ADDR_LATCH_CNT_0 | Address);
    if (cycle == 2)
        break;
    //Continue sending the row address

```

21.2 寻找更好算法

我们有时会受困于初始的思维，于是设计出来的代码臃肿庞大、难以理解。反过来，如果我们能换一种思路，说不定有更好的表示方法呢。

如何判断两个矩形是否重叠？最简单的方法就是检测一个矩形的 4 个顶点是否落在另一个矩形内。

例子：使用检测顶点的方法

```

int rectanle_overlap (struct rect *a, struct rect *b)
{
    //a 的左上角落在 b 内
    if (a->left >= b->left && a->left <= b->right
        && a->top >= b->top && a->top <= b->bottom)
        return 1;
    //a 的右上角落在 b 内
    if (a->right >= b->left && a->right <= b->right
        && a->top >= b->top && a->top <= b->bottom)
        return 1;
    //a 的左下角落在 b 内
    if (a->left >= b->left && a->left <= b->right
        && a->bottom >= b->top && a->bottom <= b->bottom)
        return 1;
    //a 的右下角落在 b 内
    if (a->right >= b->left && a->right <= b->right
        && a->bottom >= b->top && a->bottom <= b->bottom)
        return 1;
    //a 与 b 重叠，但是 a 的 4 个都不在 b 内 -- 1
    if (a->left >= b->left && a->right <= b->right
        && a->top < b->top && a->bottom > b->bottom)
        return 1;
    //a 与 b 重叠，但是 a 的 4 个都不在 b 内 -- 2
    if (a->left < b->left && a->right > b->right

```

```

        && a->top >= b->top && a->bottom <= b->bottom)
        return 1;
//a 完全覆盖 b
if (a->left < b->left && a->right > b->right
    && a->top < b->top && a->bottom > b->bottom)
    return 1;

//=====
//下面考虑 b 的 4 个角落在 a 的情况
//.....
}

```

我们可以看出这种方法需要考虑的情况太多了，可能还会出现遗漏的情况。

我们换一个思路，判断两个圆是否有重叠很简单， r_1 和 r_2 分别是两个圆的半径， d 是两个圆心的距离，当且仅当 $r_1 + r_2 \leq d$ 时，两个圆有重叠部分。我们可以把矩形看成特殊的圆，一个矩形存在垂直方向和水平方向两个半径，基于该思路，对应的代码如下。

例子：使用判断距离的方法

```

int rectanle_overlap (struct rect *a, struct rect *b)
{
    int a_width = a->right - a->left;
    int a_height = a->bottom - a->top;
    float a_center_x = a->left + a_width/2.0;
    float a_center_y = a->top + a_height/2.0;

    int b_width = b->right - b->left;
    int b_height = b->bottom - b->top;
    float b_center_x = b->left + b_width/2.0;
    float b_center_y = b->top + b_height/2.0;

    float vert_dist = fabs(a->center_y - b->center_x);
    float hori_dist = fabs(a->center_y - b->center_y);
    float vert_thres = (a_height + b_height) / 2.0;
    float hori_thres = (a_width + b_width) / 2.0;

    if(vert_dist > vert_thres || hori_dist > hori_thres)
        return 0;

    return 1;
}

```

我们可以看出代码量减少了很多，肯定不会出现遗漏的情况，但是这里需要计算中心点和长宽值，而且还用到了浮点运算，可能会出现运算精度的问题，还是有些复杂。

我们再换一种思路，要检测矩形重叠，就是要把矩形不重叠的情况排除掉，那么两个矩形不重叠的情况是什么呢？就是一个矩形在另一个矩形的上/下/左/右。

例子：检测不发生重叠的方法

```

int rectanle_overlap (struct rect *a, struct rect *b)
{
    int left = max (a->left, a->left);
    int right = min (a->right, a->right);
    int top = max (a->top, a->top);
    int bottom = min (a->bottom, a->bottom);
    if (left > right || top > bottom)
        return 0;
}

```

```
    else
        return 1;
}
```

这种方法的判断还是多了一些，因为使用了 `max` 和 `min` 操作。

例子：检测不发生重叠的方法

```
int rectanle_overlap (struct rect *a, struct rect *b)
{
    if(a->right < b->left) return 0;
    if(a->left > b->right) return 0;
    if(a->top < b->bottom) return 0;
    if(a->bottom > b->top) return 0;
    return 1;
}
```

这是最简单的方法，简单直观，易于维护，不易出错。

现在有一些代码生成工具，通过配置选项要求，然后生成代码框架，生成 HTML、PHP、Java、C#、C、C++ 等代码，生成的代码可以直接使用，或者修改之后再使用，从而可以大大地提高工作效率。

同样的道理，我们在编写代码的过程中，如果遇到繁琐、易错、但是有规律可循的东西，就可以通过编写工具脚本，自动生成规范的代码。例如，某 C 语言程序需要一个参数数组，这个数组如果手工书写会很费劲，因为参数非常有规律，那么就可以编写一个脚本，生成这个数组并写入到一个文件中，这就是最简单的代码生成。

22.1 配置 Linux 的内核

Linux 作为一个自由软件，在广大爱好者的支持下，内核版本不断更新，新内核修订旧内核的 Bug，并增加了许多新的特性。如果用户想要使用这些新特性，或者想根据自己的系统量身定制一个更高效稳定的内核，就需要重新配置并编译内核。

为了正确合理地设置内核编译配置选项，只编译系统需要功能的代码，一般要从下面四个方面考虑。

- 1) 自己定制编译的内核运行更快，因为具有更少的代码。
- 2) 系统将拥有更多的内存，因为内核部分将不会被交换到虚拟内存中。
- 3) 不需要的功能编译进入内核可能会增加被系统攻击者利用的漏洞。
- 4) 将某种功能编译为模块方式会比编译到内核的方式速度要慢一些。

配置内核可以根据需要与爱好使用下面命令中的一个。

- 1) `make config`: 基于文本的最为传统的配置界面，不推荐使用；
- 2) `make menuconfig`: 基于文本选单的配置界面，字符终端下推荐使用；
- 3) `make xconfig`: 基于图形窗口模式的配置界面，X Window 下推荐使用；
- 4) `make oldconfig`: 如果只想在原来内核配置的基础上修改一些小地方，会省去不少麻烦。

选择相应的配置时，有三种选择，它们分别代表的含义如下。

- 1) Y——将该功能编译进内核；
- 2) N——不将该功能编译进内核；
- 3) M——将该功能编译成可以在需要时动态插入到内核中的模块。

在编译内核的过程中，最烦杂的事情就是配置工作，因为不清楚到底该如何选取这些选项。实际在配置时，大部分选项都可以使用缺省值，只有小部分选项需要根据用户不同的需

要选择。选择的原则是将与内核其他部分关系较远且不经常使用的部分功能代码编译成为可加载模块，有利于减小内核的长度，减小内核消耗的内存，简化在该功能相应的环境改变时对内核的影响；不需要的功能就不要选；与内核关系紧密而且经常使用的部分功能代码直接编译到内核中。

我们自己设计系统，如果也做成这样的灵活配置，就可以参考 Linux 内核配置的方式来实现，以方便使用。很多 Verilog IP 就是做成窗口形式配置，选好设置之后，就可以生成相应的设计代码、验证代码、综合脚本，还是很方便的。

22.2 生成寄存器的代码

在一个非常大的 Verilog 模块中，里面有很多软件读写寄存器（足足有 300 多个），而且这些寄存器都是 32-bit 的。

逐个编写这些寄存器的 Verilog 代码是很繁琐的，因为这些寄存器通过 `we_n[3:0]` 支持 byte 写，`we_n[x]` 等于 0 对应的 `byte[x]` 才能被写入，因为有些寄存器复位后的初始值不是 0，有些寄存器还有保留位。

例子：常规写法，`rcu0_reg0` 的最高 4-bit 保留，初始值为 0

```
reg [27:0] rcu0_reg0;
always @(posedge clk or negedge reset_n)
begin
    if (!reset_n)
        rcu0_reg0 <= 28'h000_0000;
    else if (addr == 32'h0000_0000) begin
        if (~we_n[0]) rcu0_reg0[0 +: 8] <= bdata[0 +: 8];
        if (~we_n[1]) rcu0_reg0[8 +: 8] <= bdata[8 +: 8];
        if (~we_n[2]) rcu0_reg0[16 +: 8] <= bdata[16 +: 8];
        if (~we_n[3]) rcu0_reg0[27:24] <= bdata[27:24];
    end
end
end
```

300 多个寄存器这样写下来，很容易出错，所以笔者建议设计者使用一种新的方法。新的方法是这样的，建立一个通用的小模块，这个模块通过参数传递寄存器的基地址、使用位和初始值。这个模块的 `R_data` 是 32-bit，有人会觉得是不是浪费电路呢？不会的，因为综合时，只有 `used_bits` 中为 1 的位才会产生真正的寄存器，为 0 的位会被优化掉。

当小模块建立之后，就可以一个一个地实例化对应的寄存器，注意每个寄存器都是 32-bit 的。

例子：使用小模块写法

```
module reg32
#(parameter base_addr = 0, used_bits = 32'hFFFF_FFFF, init_val =
32'h0000_0000)
(input      reset_n,
 input      clk,
 input [31:0] addr,
 input [3:0] we_n,
 input [31:0] d,
 output [31:0] q);
reg [31:0] R_data;
wire [31:0] wr_bits = { {8{we_n[3]}}, {8{we_n[2]}}, {8{we_n[1]}},
{8{we_n[0]}} };
always @(posedge clk or negedge reset_n)
```



```

begin
    if (!reset_n)
        R_data <= init_val;
    else if (we_n != 4'hF && addr == base_addr)
        R_data <= used_bits & ((R_data & wr_bits) | (d & ~wr_bits));
    end
    assign q = R_data;
endmodule

wire [31:0] rcu0_reg0;
wire [31:0] rcu0_reg1;
wire [31:0] rcu0_reg2;
.....
reg32 #(32'h0000_0000, 32'h0FFF_FFFF, 32'h0000_0000) u_rcu0_reg0
    (.reset_n(breset_n), .clk(bclk), .we_n(bwe_n),
     .addr(baddr), .d(bwdata), .q(rcu0_reg0));
reg32 #(32'h0000_0004, 32'h0000_FFFF, 32'h0000_1234) u_rcu0_reg1
    (.reset_n(breset_n), .clk(bclk), .we_n(bwe_n),
     .addr(baddr), .d(bwdata), .q(rcu0_reg1));
reg32 #(32'h0000_0008, 32'hFFFF_0000, 32'hFFFF_0000) u_rcu1_reg2
    (.reset_n(breset_n), .clk(bclk), .we_n(bwe_n),
     .addr(baddr), .d(bwdata), .q(rcu1_reg2));
.....

```

这时代码是不是清爽了很多？是不是少写了好多行的代码呢？此方法消除了大量的代码重复。即使是这样，其实还是有进一步的优化空间，减少代码的书写量。我们再仔细研究一下，会发现实例化也是很有规律的，就是寄存器名字、基地址、使用位和初始值在改变，那么是不是可以生成这些实例化的代码呢？所以可以编写一个寄存器配置文件。

例子：寄存器配置文件

```

rcu0_reg0    32'h0000_0000    32'h0FFF_FFFF    32'h0000_0000
rcu0_reg1    32'h0000_0004    32'h0000_FFFF    32'h0000_1234
rcu1_reg2    32'h0000_0008    32'hFFFF_0000    32'hFFFF_0000
.....

```

然后通过一个简单的脚本，处理这个寄存器配置文件，直接生成上面的 Verilog 代码，从而减少代码的书写量，而且维护起来更加容易。

22.3 生成 Benes 的代码

Benes 是一种可重排无阻塞的开关网络，能实现输入端到输出端的所有置换，在通信领域得到广泛的应用。这种网络的特点就是可以通过具体的寻径路由算法，根据全通道排序的要求，实时改变各级节点开关的状态（直通或交叉），从而避免路径的冲突。

Benes 是一种多级互连网络，Benes_{N×N} 有 N 个输入和 N 个输出，共 $(2n-1)$ 级，其中 $n=\log_2 N$ 。Benes_{2×2} 是 Benes 网络的基本节点，它的结构很简单，只有两种状态：直通或交叉。Benes_{4×4} 是由 6 个节点组成的，Benes_{8×8} 是由 20 个节点组成的。Benes_{8×8} 的结构如图 22-1 所示，可以看出节点之间的连接关系很具有规律。

Benes 节点对应的 Verilog 代码如下，其中 i 是 2-bit 的输入， o 是 2-bit 的输出， c 是选择（0：直通，1：交叉）。

例子：Benes 节点

```

module BI
    (input [1:0] i,

```

```
input      c,
output [1:0] o);
assign o = !c ? {i[1], i[0]} : {i[0], i[1]};
endmodule
```

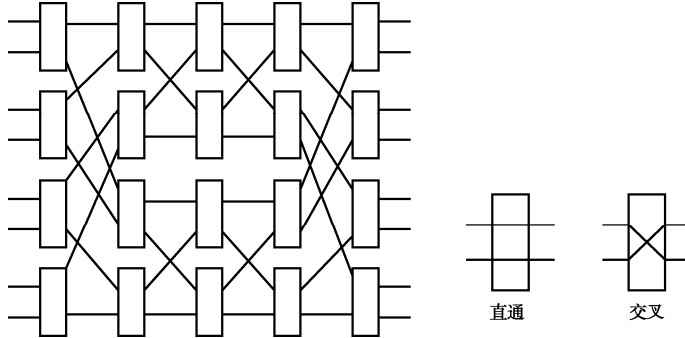


图 22-1 Benes_8x8 的结构

基于这个BI模块,你可以手工写出4x4、8x8、16x16、.....、128x128等Benes网络,Benes_4x4还能很容易地写出来,但是如果进一步手工写出其他Benes网络,你会发现非常费劲,虽然是很简单的连接,但是很容易出错。

对于任意一个Benes_NxN,它的连接关系都是固定的,而且是有规律的,既然如此,为什么不编写脚本自动生成它呢?于是作者就用Perl编写了一个脚本,指定N的值,就可以直接生成对应的Benes_NxN的代码。

```
例子: gen_benes_net.pl
#!/usr/bin/perl
use strict;
use warnings;
use Getopt::Long;
my $number = 8;
GetOptions ("number=s" => \$number);
if ($number eq "") {
    printf ("Usage: gen_benes_net.pl -number <N>\n");
    exit (1);
}
my $module = "Benes_${number}x${number}";
my $file = "$module.v";
open (FILE, "> $file") || die ("Error: can not open $file\n");
my $n = log2 ($number);
my $level = 2 * $n - 1;
my @ar = ([]);
printf (FILE "module $module\n"
        ." (input [%d-1:0] i, input [%d-1:0] c, output [%d-1:0] o);\n",
        $number, $number * $level / 2, $number);
printf (FILE "    wire w[%d-1:0][%d-1:0];\n", $number, $level-1);
my $x;
for ($x = 1; $x < $level; $x++) {
    generate_level ($x);
}
print_connect ();
printf (FILE "endmodule\n");
close (FILE);

sub log2
{
    my $x = ($_[0] - 1);
```

```

my $sum = 0;
while ($x > 0) {
    $sum = $sum + 1;
    $x = $x >> 1;
}
return $sum;
}

sub generate_level
{
    my $nl = $_[0];
    my $pl = $nl - 1;
    my $number_div_2 = int ($number / 2);
    my $level_div_2 = int ($level / 2);
    my $tail = 0;
    my $loop = $nl;
    if ($nl > $level_div_2) {
        $tail = 1;
        $loop = $level - $nl;
    }
    $loop = 2 ** ($loop - 1);
    my $group = $number / $loop;
    my ($i, $j, $c, $m);
    for ($j = 0; $j < $loop; $j++) {
        my $begin = $group * $j;
        for ($i = 0; $i < $group; $i++) {
            if (!$tail) {
                if (($i & 1) == 0) {
                    $c = $i >> 1;
                }
                else {
                    $c = ($i >> 1) + $group / 2;
                }
            }
            else {
                if ($i < ($group / 2)) {
                    $c = $i * 2;
                }
                else {
                    $c = ($i - $group / 2) * 2 + 1;
                }
            }
            $m = $i + $begin;
            $ar[$c + $begin][$nl] = "w[{$m}][{$pl}]";
        }
    }
}

sub print_connect
{
    my ($i, $j);
    printf (FILE "\n");
    for ($i = 0; $i < $number; $i++) {
        printf (FILE " //");
        for ($j = 1; $j < $level; $j++) {
            printf (FILE " %s", $ar[$i][$j]);
        }
        printf (FILE "\n");
    }
    printf (FILE "\n");
    for ($j = 0; $j < $level; $j++) {

```

```

printf (FILE " //Level %d\n", $level);
for ($i = 0; $i < $number / 2; $i++) {
    my ($si, $sc, $so);
    $si = sprintf ("%s, %s", $ar[$i*2+1][$j], $ar[$i*2][$j]) if
($j != 0);
    $si = sprintf ("{i[%d], i[%d]}", $i*2+1, $i*2) if ($j == 0);
    $so = sprintf ("{w[%d][%d], w[%d][%d]}", $i*2+1, $j, $i*2, $j);
    $so = sprintf ("{o[%d], o[%d]}", $i*2+1, $i*2) if ($j == ($level
- 1));
    $sc = sprintf ("c[%d]", $j * ($number / 2) + $i);
    printf (FILE " BI u_bi_%d_%d (.i(%s), .c(%s), .o(%s));\n",
        $i, $j, $si, $sc, $so);
}
printf (FILE "\n");
}
for ($j = 0; $j < ($level - 1); $j++) {
    for ($i = 0; $i < $number; $i++) {
        printf (FILE " wire w_%d_%d = w[%d][%d];\n", $i, $j, $i, $j);
    }
}
printf (FILE "\n");
}

```

运行如下的命令，分别生成 Benes_8×8 和 Benes_128×128。

```

gen_benes_net.pl -number 8
gen_benes_net.pl -number 128

```

自动生成的 Benes_8×8 的代码如下，是不是很有规律？但是如果用手工连接，是不是很容易出错呢？

例子：自动生成的 Benes_8×8

```

module Benes_8x8
(input [8-1:0] i, input [20-1:0] c, output [8-1:0] o);
wire w[8-1:0][4-1:0];
//level 0
BI u_bi_0_0 (.i({i[1], i[0]}), .c(c[0]), .o({w[1][0], w[0][0]}));
BI u_bi_1_0 (.i({i[3], i[2]}), .c(c[1]), .o({w[3][0], w[2][0]}));
BI u_bi_2_0 (.i({i[5], i[4]}), .c(c[2]), .o({w[5][0], w[4][0]}));
BI u_bi_3_0 (.i({i[7], i[6]}), .c(c[3]), .o({w[7][0], w[6][0]}));
//level 1
BI u_bi_0_1 (.i({w[2][0], w[0][0]}), .c(c[4]), .o({w[1][1], w[0][1]}));
BI u_bi_1_1 (.i({w[6][0], w[4][0]}), .c(c[5]), .o({w[3][1], w[2][1]}));
BI u_bi_2_1 (.i({w[3][0], w[1][0]}), .c(c[6]), .o({w[5][1], w[4][1]}));
BI u_bi_3_1 (.i({w[7][0], w[5][0]}), .c(c[7]), .o({w[7][1], w[6][1]}));
//level 2
BI u_bi_0_2 (.i({w[2][1], w[0][1]}), .c(c[8]), .o({w[1][2], w[0][2]}));
BI u_bi_1_2 (.i({w[3][1], w[1][1]}), .c(c[9]), .o({w[3][2], w[2][2]}));
BI u_bi_2_2 (.i({w[6][1], w[4][1]}), .c(c[10]), .o({w[5][2], w[4][2]}));
BI u_bi_3_2 (.i({w[7][1], w[5][1]}), .c(c[11]), .o({w[7][2], w[6][2]}));
//level 3
BI u_bi_0_3 (.i({w[2][2], w[0][2]}), .c(c[12]), .o({w[1][3], w[0][3]}));
BI u_bi_1_3 (.i({w[3][2], w[1][2]}), .c(c[13]), .o({w[3][3], w[2][3]}));
BI u_bi_2_3 (.i({w[6][2], w[4][2]}), .c(c[14]), .o({w[5][3], w[4][3]}));
BI u_bi_3_3 (.i({w[7][2], w[5][2]}), .c(c[15]), .o({w[7][3], w[6][3]}));
//level 4
BI u_bi_0_4 (.i({w[4][3], w[0][3]}), .c(c[16]), .o({o[1], o[0]}));
BI u_bi_1_4 (.i({w[5][3], w[1][3]}), .c(c[17]), .o({o[3], o[2]}));
BI u_bi_2_4 (.i({w[6][3], w[2][3]}), .c(c[18]), .o({o[5], o[4]}));
BI u_bi_3_4 (.i({w[7][3], w[3][3]}), .c(c[19]), .o({o[7], o[6]}));
endmodule

```

任何代码都要经过测试才能发布使用,就算是几行简单的 Shell 脚本,也要经过测试检验。测试是检查系统的正确性、完整性、安全性的过程。软件代码需要经过完整的软件测试,然后才能发布,公开销售或者免费下载; Verilog 代码需要经过完整的仿真验证和 FPGA 测试,才能投入使用或者生产芯片。

测试也是代码重构的基础,保证设计代码平稳地向前演进。测试还起到了设计代码文档的作用,可以通过阅读和运行测试代码,快速地理解设计代码,所以测试代码也要具有良好的表达力和可读性。

23.1 测试中问题

笔者在某公司工作的时候,有个测试人员离开了公司。他写的测试代码很难看懂,杂乱无章、重复冗余,设计人员接手后整理起来也很费劲,于是笔者决定重写测试代码。只是针对其中的某一个小程序,原来的测试人员写了 600 多行代码,而笔者在与设计人员沟通后,只写了 100 多行代码,而且清晰易懂。

设计代码只有经过测试代码的严格测试,质量才能有所保证,测试代码有其特殊性,但是很多地方同样要遵循设计代码的质量标准,需要代码整洁、简单精练,这样才能可扩展、可维护、可复用,才能更有力地保障设计代码。

但是,很多人认为测试代码不用遵循设计代码的质量标准,导致测试代码包含着各种各样的问题,总结如下。

- 1) 测试代码不符合编码规范,名字定义不清,代码缺少注释,代码可读性很差。
- 2) 测试代码不符合分层和分块的原则,冗长、啰嗦、重复,结构混乱不清,函数臃肿庞大,充满太多的细节。
- 3) 测试代码没有随着设计代码同步演进,导致测试代码陈旧腐败。
- 4) 测试代码没有做到可扩展、可维护、可复用。
- 5) 增加新测试不是很容易,很多人倾向于使用复制、粘贴和修改。
- 6) 测试失败的信息不是很有帮助,没有为进一步的调试提供足够的信息。
- 7) 测试的覆盖度不够,只是对整个系统的一部分做了测试。
- 8) 测试没有做成自动化,不能方便地进行自动测试和回归测试。
- 9) 有些人将用于模块测试的代码用过之后就丢弃不用,没有把它们整理好然后提交。
- 10) 有些人测试只求速度,不求质量,破坏规矩,只要测试代码还能工作,就认为 OK 啦,结果测试代码越来越脏乱,越来越难以修改,最后只能被丢弃掉。

23.2 测试的原则

测试代码和设计代码要有一样的要求，但是测试代码也有其独特的一面，如果要进行完美的测试，那么就要遵循如下规则：

- 1) 编码优美：测试代码要遵循编码规范，要有优美的编码风格，要有良好的结构。
- 2) 独立灵活：测试应该相互独立而且灵活，应该能单独运行每个测试。
- 3) 运行快速：测试应该足够快，要既能快速运行，又能并行运行，这样才能尽早地发现错误。
- 4) 可以重复：任何人在任意一台机器上都可以很容易地运行测试。
- 5) 同步编写：测试应当提早编写，与设计代码同步进行编写。
- 6) 和谐通用：建立通用测试平台，方便使用和维护，可以运行一个、多个或所有的测试。
- 7) 自我检测：测试要有自我检测机制，自己检查结果正确与否，不要手工检查结果。
- 8) 规范测试：对 Verilog 不要使用被测模块内部的信号，也不要使用过多的层次引用。
- 9) 回归测试：任何对设计的改动都有可能引入错误，所以要随时进行回归测试。
- 10) 覆盖全面：要采用直接测试和随机测试相结合的方法，要考虑所有的边界条件。
- 11) 不断重构：要不断地对测试代码调整，精益求精，你需要巨大的耐心。
- 12) 积极沟通：测试人员要与设计人员积极地沟通，这样才能写出好的测试代码。不要放过任何出错的测试，也不要放过任何奇怪的现象。
- 13) 测试管理：不要认为模块测试是那种用来确保模块运行的用过即扔的小测试，要丰富完善，还要提交到 CVS 或 SVN 内。

23.3 设计要更好

如果你要为你自己设计的代码编写测试，那么你就会在编写代码时尽量写得更容易测试，这样的编程方式意味着会产生更好的设计代码。对测试友好的设计往往会有良好的代码组织，不同部分做不同的事情。

可测性差的设计代码具有这些特征：使用大量的全局变量，对其他的代码有很大的依赖（高耦合，有过多的状态和设置），代码有不确定性行为（存在竞争条件，导致线程之间控制不好），代码结构性很差，没有很好的抽象和封装，代码层次混乱。

可测性好的设计代码具有这些特征：函数/模块只做一件事情，对其他的代码有很小的依赖（低耦合），函数/模块的接口简单、定义明确，代码具有很好的模块化结构，具有很好的抽象和封装，代码没有不确定性行为。

23.4 提高可读性

测试代码的可读性和设计代码的可读性是同样重要的，其他程序员经常会把测试代码看成非正式的文档，因为它记录了设计代码是如何工作和使用的。因此如果测试代码很容易阅读，那么其他人员就会对设计代码的行为有很好的理解，同时也可以很方便地修改和增加测试。

但是当测试代码存在各种问题，代码量非常多、可读性又非常差的时候，修改测试代码

将是一场噩梦，导致设计人员不敢修改设计代码，因为设计人员不想纠结于测试代码中。即使设计人员修改设计代码，但是也不敢增加新的测试代码，这样过一段时间之后，测试的功能越来越少，设计和测试之间脱节，最后也就对测试失去了信心。

所以应该想办法让测试代码具有很高的可读性，易于修改和增加新的测试，让其他人员也习惯并喜欢上测试代码。这样他们在修改设计代码之前，能够对测试代码的破坏做出分析，能够觉得很容易地修改和增加测试。如果测试代码的可读性很好，大家就会很容易写出新的测试，也会愿意写出更多的测试。另外，如果设计代码写得很容易测试，那么设计和测试会相辅相成，达到双赢的目的。

作为一条普遍的测试原则，你应当“对使用者隐去不必要的细节，以便于突出更重要的细节”，从而更方便地使用，测试代码应该简单、精悍、富有表达力，阅读测试代码的人能够很快地就搞清楚情况，不至于被细节误导或吓到。另外，还应该让出错信息具有可读性，便于出错时定位分析出错原因。

例子：通过宏定义隐去测试的细节

```
//Enable usage: 0/write disable, 1/ write enable.
#define CHECK_MEMORY_WRITE(flag_addr, mem_addr, w_data, enable) \
    ({ U32 exp; \
      REG32(flag_addr) = 0x33; \
      REG32(mem_addr) = w_data; \
      if (enable) exp = 0x33; else exp = 0x34; \
      if (flag_addr == DLM_SPL0_NMI || flag_addr == DLM_SPL1_NMI) \
          CHECK_FLAG_USE_FINITE_LOOP (flag_addr, exp); \
      else \
          CHECK_FLAG_USE_REG32 (flag_addr, exp); \
      REG32(flag_addr) = 0; \
    })

//Enable usage: 0/read disable, 1/read enable, 2/read_enable and check
read_data
#define CHECK_MEMORY_READ(flag_addr, mem_addr, r_data, enable) \
    ({ U32 exp; U32 xx; \
      REG32(flag_addr) = 0x33; \
      xx = REG32(mem_addr); \
      if (enable) { exp = 0x33; \
        if (enable == 2 && xx != r_data) Error (); } \
      else exp = 0x34; \
      if (flag_addr == DLM_SPL0_NMI || flag_addr == DLM_SPL1_NMI) \
          CHECK_FLAG_USE_FINITE_LOOP (flag_addr, exp); \
      else \
          CHECK_FLAG_USE_REG32 (flag_addr, exp); \
      REG32(flag_addr) = 0; \
    })

//Times usage: 0/no exception occur, >= 1/ the times of occurring exceptions
#define CHECK_MEMORY_EXECUTE(flag_addr, function, times) \
    ({ U32 exp; \
      REG32(flag_addr) = 0x33; \
      function; \
      exp = 0x33 + times; \
      CHECK_FLAG_USE_REG32 (flag_addr, exp); \
      REG32(flag_addr) = 0; \
    })
```

例子：使用宏定义简化测试代码的编写

```
void test_check_memory_codeid_secure_region (void)
```

```

{
    //Factory code ID region
    CHECK_MEMORY_WRITE (DLM_SPL0_NMI, ID_DATA_START + 0x000, 0x12345678, 0);
    CHECK_MEMORY_READ (DLM_SPL0_NMI, ID_DATA_START + 0x100, 0x12345678, 1);
    .....
    ....
    //Secure arithmetic region
    Page_Erase(SECURE_START_ADDRESS);
    Page_Write(SECURE_START_ADDRESS);
    CHECK_MEMORY_READ (DLM_SPL0_NMI, SECURE_START_ADDRESS, 0x12345678, 0);
    CHECK_MEMORY_EXECUTE (DLM_SPL0_Trap_TLB_Misc, Secure_program_test2(),
1);
}

```

要为测试函数起个好名字，不要怕名字太长，因为别的代码不会调用测试函数，所以那些要避免使用长函数名的理由在这里并不适用。测试函数名字的作用就像是注释，要具有完整的描述性，要明确地描述要测试的内容。另外，如果测试失败了，测试框架要能够输出测试失败函数的名字，这样就可以一目了然地定位问题。

例子：测试函数的名字

```

test_kpc_press_irq
test_kpc_press_fiq
test_kpc_press_mk
test_kpc_press_wfi
test_kpc_press_wakeup

```

如果你在使用一个测试框架，可能就已经有命名测试函数的规范。对于测试代码的辅助函数，可以按照常规函数一样命名。

23.5 测试智能化

如果想要测试智能化，就要搭建一个灵活的测试环境，既能够维护原有的测试，又能够快速构建新的测试，能够很方便地运行一个、一组或所有测试，这就需要对系统有很好的理解，同时又能够编写出很好的测试脚本。

我们在做定向测试的时候，很多时候测试代码基本上都是一样的，变化的只是几个参数，如果我们只是简单地使用复制粘贴修改，那么我们得到的就是一堆难以维护的代码。但是通过编写通用框架和对应的处理脚本，就可以做到定向测试的智能化。

在 SOC Verilog 仿真测试中，要编写很多的 C 语言程序测试文件，每个 C 语言程序都要编译生成可执行文件，然后由仿真器装载到 SRAM 中，再由 CPU 运行执行。如果一个一个地写出这样的测试文件，会非常累的，而且很难修改和维护。但是通过编写通用的测试程序，就可以通过脚本根据测试参数生成多个不同的测试程序。

例如，test_kpc_press_wakeup.c 就是一个通用的测试程序，用于测试 KPC 模块的 WAKEUP 功能，X_CURRENT、X_AFTER_SLEEP、X_AFTER_WAKEUP、X_ROW、X_COL 是要改变的参数。如果不考虑 smart_相关的注释，那么这只是一个定向测试。但是通过脚本处理，生成 test_kpc_press_wakeup_smart_0.c ~ test_kpc_press_wakeup_smart_5.c，在这几个文件中这些可变参数是由对应的 smart_verify_x 决定的。对于 KPC 模块，另外有几个通用的测试程序，分别测试不同的功能。

例子：test_kpc_press_wakeup.c


```

//smart_begin
//smart_head      X_CURRENT X_AFTER_SLEEP X_AFTER_WAKEUP X_ROW X_COL X_DEB
//smart_every_0   CGU_SLOW_PD  0           1           4           4           3
//smart_every_1   CGU_SLOW_PD  0           3           5           5           3
//smart_every_2   CGU_SLOW_PE  0           1           7           7           4
//smart_every_3   CGU_SLOW_PE  0           3           8           8           5
//smart_every_4   CGU_NORMAL   0           1           7           7           3
//smart_every_5   CGU_NORMAL   0           3           8           8           3
//smart_end

#include <t_kpc.h>
#ifdef SMART_TEST
#else //NO_SMART_TEST
#define X_CURRENT      CGU_NORMAL
#define X_AFTER_SLEEP  0
#define X_AFTER_WAKEUP 7
#define X_ROW          8
#define X_COL           8
#define X_DEB          9
#endif

volatile unsigned int kpc_dataaa;
volatile unsigned int kpc_datab;
volatile int kpc_flag = 0;
void kpc_handler (struct pt_regs *regs)
{
    show ("Enter kpc handler", 0);
    kpc_dataaa = REG32(KPC_DATAAA);
    kpc_datab = REG32(KPC_DATAB);
    //kpc interrupt is cleared by writing 0.
    REG32(KPC_STAT) = 0;
    kpc_flag = 1;
}

int test_kpc_press_wakeup (void)
{
    int i;
    int val;
    int kpc_exp0, kpc_exp1;
    int cst, exp;

    if (MODE == 1)
        cg_u_enter_mode (4, 0);

    show ("Change to state", 1, X_CURRENT);
    REG32(CGU_PCR) = X_CURRENT;
    if (X_CURRENT == CGU_SLOW_PE || X_CURRENT == CGU_NORMAL)
        exp = (CGU_ST_DIV_READY | CGU_ST_PLL_LOCK | CGU_ST_CGU_READY);
    else
        exp = (CGU_ST_DIV_READY | CGU_ST_CGU_READY);
    cst = REG32(CGU_CST);
    while ((cst & exp) != exp)
        cst = REG32(CGU_CST);

    //set gpio
    gpio_use_for_kpc (X_ROW, X_COL);
    //set exception handler
    intc_enable_source (1 << INTC_NUM_KPC);
    set_exp (TP_IRQ, INTC_NUM_KPC, kpc_handler);
    //config kpc
    val = ((X_DEB << 8) | ((X_COL - 1) << 4) | (X_ROW - 1));
}

```

```

REG32(KPC_CFG) = val;
REG32(KPC_DRV_DATA0) = val;
REG32(KPC_DRV_CMD) = KPC_CMD_MATRIX;
//enable kpc
REG32(KPC_CTRL) = 1;

unsigned long long x;
x = 0x0000000001000000ULL;
kpc_exp0 = x & 0xFFFFFFFF;
kpc_exp1 = x >> 32;
show ("Enter sleep", 0);
kpc_flag = 0;
REG32(KPC_DRV_DATA0) = kpc_exp0;
REG32(KPC_DRV_DATA1) = kpc_exp1;
REG32(KPC_DRV_CMD) = KPC_CMD_PRESS;
cgu_enter_sleep (X_AFTER_SLEEP, X_AFTER_WAKEUP, &kpc_flag);
if (kpc_dataa != kpc_exp0 || kpc_datab != kpc_exp1)
{
    show ("Single key press is error!", 4,
        kpc_exp0, kpc_exp1, kpc_dataa, kpc_datab);
    return 1;
}
show ("Exit sleep", 0);
return 0;
}

```

例子：通过脚本生成的 test_kpc_press_wakeup_smart_0.c

```

#define SMART_TEST
#define X_CURRENT          CGU_SLOW_PD
#define X_AFTER_SLEEP      0
#define X_AFTER_WAKEUP     1
#define X_ROW              4
#define X_COL              4
#define X_DEB              3
#include <test_kpc_press_wakeup.c>

```

以上是通过测试框架生成多个测试程序的例子，框架的参数是用宏定义实现的。换个思路，如果框架的参数是用一个结构数组实现的，那么就可以把多条参数放到一个结构数组内，然后对这个结构数组使用索引，一条一条地测试，当然 test_kpc_press_wakeup 需要做一下改动，把参数改成使用结构数组的形式。

例子：使用结构数组实现循环测试。

```

smart_array[] = {
    {CGU_SLOW_PD, 0, 1, 4, 4, 3},
    {CGU_SLOW_PD, 0, 3, 5, 5, 3},
    {CGU_SLOW_PE, 0, 1, 7, 7, 4},
    {CGU_SLOW_PE, 0, 3, 8, 8, 5},
    {CGU_NORMAL, 0, 1, 7, 7, 3},
    {CGU_NORMAL, 0, 3, 8, 8, 3} };
for (i = 0; i < sizeof (smart_array)/sizeof(smart_array[0]); i++)
    test_kpc_press_wakeup(i);

```

只要你有基于框架测试的想法，不管是仿真测试、软件测试还是系统测试，你就可以想到搭建通用的测试框架。进一步，当你能搭建出测试框架时，你就能实现随机测试。但是要注意框架所用的参数不能是完全随机的，而是在受约束的条件下生成的参数。

23.6 定位 Bug

如果你对一个程序员说：你的代码有 Bug。他的第一反应就是：你的环境有问题吧，你真的会使用吗？但是如果你委婉地说：你这个程序和预期的有点不一致，是不是我的使用方法有问题？他本能地会想：是不是又出 Bug 了！这就是程序员的自尊与本能。

大家都知道 Debug 的困难，都经历过 Debug 的痛苦，连续几天才定位出一个 Bug，这样的 Bug 还算是简单的，有的 Bug 可能需要几个月才能定位出来，可以用一句话来形容，“路漫漫其修远兮，吾将上下而求索”。定位 Bug，就要全身心投入进去，脑袋中要闪现出一个个英雄，要像黄继光、邱少云、小英雄雨来、放牛娃王二小一样，坚持下去。只有你投入进去，不错过任何蛛丝马迹，你才能定位到 Bug。

但是 Debug 光有全身心投入也是不行的，还需要专业技巧，这就是优秀的测试人员和普通测试人员之间的区别。设计人员描述了一个花费几小时都不能解决的问题，优秀的测试人员寻问设计人员几个问题之后，就会有明确的测试目标，并且很快就会找到错误的位置和原因。优秀的测试人员永远不会忘记，不管系统的行为乍看起来是多么的神秘莫测，其背后总有合乎逻辑的解释^[21]。

当你定位那些 Bug 之后，你就会恍然大悟，原来是这么回事呀！怎么绕了那么多弯路呢？怎么会犯这么低级的错误呢？真是“吃一堑，长一智”啊。因为你经历了，你才知道怎么回事，以后才会小心避免，对于自己或者别人的类似错误，你就会一看便知，因为它们已经变成了你的直觉。

特斯拉说：“直觉是可以超越知识的。毋庸置疑，我们头脑中存在着一些神奇的神经纤维，它们可以帮助我们发现真理，这是逻辑推理或者其他任何主观努力都做不到的。”但是，这些直觉需要我们的投入和积累，不是平白无故产生的。

下面是笔者帮助同事定位代码遇到的 Bug，觉得这个 Bug 很低级，照理说程序员不应该犯这种低级的错误。

```
例子：错误的 bignum_add_ch
typedef struct BIGNUM_St {
    unsigned int length;
    unsigned char *data;
} BIGNUM;
void bignum_add_ch (BIGNUM *a, BIGNUM *b, unsigned char c)
{
    int i, flag;
    i = a->length - 1;
    flag = c;
    while ((flag != 0) && (i > 0)) {
        a->data[i] = b->data[i] + flag;
        flag = (a->data[i] + c > 0xff) ? 1 : 0;
        i--;
    }
    for (; i >= 0; i--)
        a->data[i] = b->data[i];
}
```

首先说一下错误现象，测试数据使用的是随机数，如果用伪随机数，测试就不会出错；如果用真随机数，测试一百多次通常会错一次，但是错误的时间不是固定的。同事之前还捕捉到了一组错误的数。作者在测试期间也发现了一个规律，就是当最低字节为 0xFF 或 0xFE

时必然出错。

错误最后集中在众多代码中一个很小的函数 `bignum_add_ch` 上，这个函数的功能是一个大数加上一个字符，然后赋值给另一个大数。大数的数据结构如下，`data` 指向分配的空间，空间长度为 `length`，`data[0]` 为最高字节，`data[length-1]` 为最低字节，所以这个加法要从 `length-1` 递减到 0 进行。

笔者不用随机数据作为测试数据，改用那组固定错误的的数据，`b` 的最低字节为 `0xFF`，`c` 的值为 `0x01`，进一步调试，终于发现原因。当第一次循环时，`a->data[0]` 的值应该为 `0x00`，`flag` 的值应该为 1，但是实际的结果是 `a->data[0]=0x00`、`flag=0`，所以出现了错误。于是笔者修改了代码，使用了 `tmp` 变量，结果运行正确，以后再没有出现“测试一百多次通常会错一次”的现象。

例子：修改后的 `bignum_add_ch`

```
void bignum_add_ch (BIGNUM *a, BIGNUM *b, unsigned char c)
{
    int i, flag;
    unsigned int tmp;
    i = a->length - 1;
    flag = c;
    while ((flag != 0) && (i >= 0)) {
        tmp = b->data[i] + flag;
        a->data[i] = tmp;
        flag = (tmp > 0xff) ? 1 : 0;
        i--;
    }
    for (; i >= 0; i--)
        a->data[i] = b->data[i];
}
```

其实笔者在浏览相关的代码时，就已经发现 `bignum_add_ch` 函数写得有些奇怪，但是笔者当时还没有真正明白它的操作，所以没有确定这里是否有错误。只是真正定位到这个函数时，经过同事检查，终于确定出错的原因。

反过来，仔细看一下，这段代码是不是非常简单？这种错误是不是很低级？可就是出错了，你说让人恼火不？笔者感觉设计者是在做巧合编程，做过了一些测试，恰好完全正确，然后就不管了。如果设计者能够仔细检查代码，能够消除代码中的奇怪地方，就不会出现这种低级的错误，笔者和同事也用不着花费一天的时间用来 `Debug`。

C 语言一些要素

汉语有很多容易混淆的地方，稍不小心，就会写出错别字。例如，有一天笔者就在电视上看到“犯罪份子”，实际应该写为“犯罪分子”。例如，乱用“在、再”，乱用“的、地、得”，“潦草”误作“撩草”、“急躁”误作“急燥”、“国籍”误作“国藉”、“相形见绌”误作“相形见拙”、“川流不息”误作“穿流不息”，“好高骛远”误作“好高骛远”、“趋之若鹜”误作“趋之若鹭”，“九霄云外”误作“九宵云外”。另外，汉语还有各种方言，会有交流不畅的情况，所以才有《胡建人怎样学习普通发》。

每种编程语言的发展也不是一蹴而就的，同样有容易混淆的地方，同样也有一些陷阱和缺陷。这点在 C 语言上最能体现，C 语言虽然功能强大高效，使用灵活自由，便于移植，但是 C 语言是一门发展较早的结构化语言，所以也拥有最多的容易混淆的地方，这点在操作符上最为体现，例如，`i++`和`++i`、`&`和`&&`、`#ifdef`和`#if`的区别。C 语言也拥有最多的陷阱和缺陷，要不怎么会有《C 专家编程》、《C 陷阱和缺陷》和《C 和指针》这几本书呢？所以作者每次阅读这些 C 语言的书时，总会有新的发现，总是感到很欣喜。

学习好 C 语言对于学习好其他语言也有很大的帮助，Verilog、Perl、C++、Java、C#等就从 C 语言借鉴了很多东西，有些甚至是完全相同。

本部分讨论 C 语言的一些要素、容易混淆和出错的地方、C99 的一些好的用法，另外还简单讨论一下 GCC 和汇编语言。

第 24 章

关键字

对于 C 语言的关键字，你要做到“了然于胸”的境地。表 24-1 就是 C 语言关键字的速查表，可以知道这些关键字的简单说明。

表 24-1 C 语言关键字的简单说明

关键字	说明
auto	用于定义临时变量，现在没有什么用处
register	用于定义临时变量，现在没有什么用处
char	用于定义整数类型的变量，分别有不同的字节数，通常对应的字节数为 1、2、4、4 还可以定义 long long 类型，它的字节数为 8
short	
int	
long	
signed unsigned	用于定义整数类型的变量，分别对应符号数和无符号数
float double	用于定义浮点数类型的变量，分别对应单精度和双精度，字节数分别为 4 和 8
void	用于声明函数和定义指针 如果函数没有返回值，那么应声明为 void 如果函数没有参数，那么应声明为 void 如果用于指针 (void *)，就声明为无类型的指针
enum	用于定义枚举类型
struct	用于定义结构类型，也可以定义位域
union	用于定义联合类型
typedef	用于定义新的数据类型，方便书写和理解
sizeof	用于返回参数所用的字节数
const	用于说明变量为只读类型
volatile	用于说明变量为易失类型
extern	用于说明函数和全局变量具有外部链接属性。在声明函数原型时，可以不用它。
static	用于说明函数、全局变量和局部变量具有静态属性， 限定静态函数和静态全局变量只能在它们所在的文件内使用。 限定静态局部变量具有静态属性

续表

关 键 字	说 明
if else	用于条件判断
switch case default	用于实现多分支 case 用于每个分支 default 用于缺省的分支
break	用于循环语句时，跳出循环，结束整个循环 用于 switch 语句时，跳出 switch，结束当前分支
continue	用于循环语句，提前结束本次循环
for	for 循环，例如 for(i = 0; i < 100; i++)
while	while 循环，检查条件在开始处
do { } while	do{ }while 循环，检查条件在结尾处，最好少用
goto	用于跳转。最好不用，如果要用，就要合理使用
return	用于从函数返回 对于 void 类型函数，return 不能有返回值 对于有返回值的函数，return 必须有返回值
_Bool	C99 新增，用于定义布尔类型的变量，它的值只有 0 和 1
_Complex _Imaginary	C99 新增，用于定义复数类型的变量
inline	用于把函数定义为内联函数
restrict	用于限定和约束指针
asm __asm__	用于内嵌汇编
__attribute__	GCC 的一大特色就是 __attribute__ 机制。 __attribute__ 可以用来设置函数、变量和类型的属性。

25.1 内部数据类型

对于常见的 32 位 CPU 的编译器，表 25-1 是内部数据类型的取值范围。对于 16 位和 64 位 CPU 编译器，有些类型的取值范围可能与下表不相同，需要检查头文件 `limits.h`，或者使用 `sizeof` 检查一下字节个数。

表 25-1 内部数据类型的取值范围（32 位 CPU）

类 型	字节数	最 小 值	最 大 值	说 明
signed char	1	-256	255	简写为 char
unsigned char	1	0	511	
signed short	2	-32768	32767	简写为 short
unsigned short	2	0	65535	
signed int	4	-2^{31}	$2^{31}-1$	简写为 int signed long 等同于 int
unsigned int	4	0	$2^{32}-1$	unsigned long 等同于 unsigned int
signed long long	8	-2^{63}	$2^{63}-1$	C99 增加，简写为 <code>__int64</code>
unsigned long long	8	0	$2^{64}-1$	C99 增加，简写为 <code>unsigned __int64</code>
float	4	-3.40E+38	+3.40E+38	精度为 6~7 位有效数字
double	8	-1.79E+308	+1.79E+308	精度为 15~16 位有效数字
long double	10/12/16			字节数、范围和精度由编译器实现决定

对于 32 位和 64 位编译器，下表列出了某些类型的字节数差异，所以要想让程序跨平台运行，要特别关注这些差异。

表 25-2 32 位和 64 位编译器的某些类型的字节数差异

类 型	32 位	64 位
char *	4	8
long	4	8
size_t	4	8
time_t	4	8
long long	8	8

例子：打印内部数据类型的字节数
`#include <stdio.h>`


```

int main (void)
{
    printf ("char=%d\n", sizeof(char));
    printf ("short=%d\n", sizeof(short));
    printf ("int=%d\n", sizeof(int));
    printf ("long=%d\n", sizeof(long));
    printf ("long long=%d\n", sizeof(long long));
    printf ("float=%d\n", sizeof(float));
    printf ("double=%d\n", sizeof(double));
    printf ("long double=%d\n", sizeof(long double));
    return 0;
}

```

为了便于程序的移植，也便于代码的统一，最好不要直接使用表格中的类型，而是使用 `typedef` 定义出一套统一的类型，再进一步定义出其他数据类型。编译器已经为我们做了这个工作，就是使用标准的头文件 `stdint.h`。

例子： `stdint.h` 中一部分

```

typedef signed char      int8_t;
typedef unsigned char    uint8_t;
typedef short            int16_t;
typedef unsigned short   uint16_t;
typedef int              int32_t;
typedef unsigned         uint32_t;
typedef long long        int64_t;
typedef unsigned long long uint64_t;

```

除非想要非常节省内存，通常来说使用 `int`、`char` 和 `double` 这三种类型就足够了，不要使用过多的类型，对于 `int` 和 `char`，根据实际场合选择 `signed` 和 `unsigned`。在运算时最好是同种类型的变量，因为不同类型的变量在一起运算可能会出现一些意想不到的问题。

25.2 新增数据类型

C99 增加了 64 位整数：

`long long int`，也可以用 `_int64` 表示。

`unsigned long long int`，也可以用 `unsigned _int64` 表示。

C99 增加了这三种内部数据类型：

`_Bool`，用于表示布尔类型；

`_Complex`，用于表示复数类型；

`_Imaginary`，用于表示复数的虚部。

25.3 enum

`enum` 用于定义枚举类型，定义时要列举出所有可能的取值，声明为枚举类型变量取值不要超出定义的范围。

例子：枚举类型

```

enum WEEKDAY {SUN, MON, TUE, WED, THU, FRI, SAT};
enum WEEKDAY today;

```

`enum` 和 `#define` 都用于定义一些常量，那么用哪一个好一些呢？`#define` 是编译前的预处理命令，预处理只是对使用宏的地方做简单的替换，并不做语法检查，语法是否错误还要在编译时检查。另外，宏定义名不会添加到目标文件的符号列表中，所以不利于程序的调试。但是使用 `enum` 就没有这些问题，所以最好使用 `enum` 定义常量。

下面看一下预处理对 `enum` 和 `#define` 输出的结果，对 `#define` 做了宏替换，而对 `enum` 则做了保留，这里使用 GCC 编译器。

```
例子: enum.c
#if USE_DEFINE
#define ZERO 0
#define ONE 1
#define TWO 1
#else
enum TAGS {ZERO, ONE, TWO};
#endif
int main (void)
{
    printf ("%d\n", ONE);
    return 0;
}
```

```
例子: enum.i, 使用宏定义的预处理输出
#gcc -E -P -DUSE_DEFINE enum.c -o enum.i
int main (void)
{
    printf ("%d\n", 1);
    return 0;
}
```

```
例子: enum.i, 使用 enum 的预处理输出
#gcc -E -P enum.c -o enum.i
enum TAGS {ZERO, ONE, TWO};
int main (void)
{
    printf ("%d\n", ONE);
    return 0;
}
```

25.4 struct

`struct` 用于定义结构类型，它是一种构造类型。结构是由若干成员组成的，成员既可以是基本类型，也可以是其他构造类型。

```
例子: 结构类型
struct EMPLOYEE {
    int id,
    char name[40],
    char sex,
    int salary
};
struct EMPLOYEE employee;
employee.id = 100;
strcpy (employee.name, "Han Meimei");
//两个结构之间可以直接赋值，不用挨个成员赋值
struct EMPLOYEE manager, engineer;
manager = engineer;
```

例子：结构的初始化可以有两种方式

```
struct EMPLOYEE employee = {100, "Han Meimei", 1, 9999}; //C89 的常规方式
struct EMPLOYEE employee = //C99 的指定成员方式, 可以初始化指定的成员
{.id = 100, .name = "Han Meimei", .sex = 1, .salary = 9999};
```

struct 的大小是由成员的顺序和对齐方式决定的, 成员需要按自己的类型对齐, 所以成员之间可能会存在空缺, 例如, EMPLOYEE 的大小可能是 52 字节, 因为在 sex 和 salary 之间有 3 个字节的空缺。为了保证代码在不同系统上的可移植性, 应该使用 sizeof(struct EMPLOYEE) 计算大小, 不要手工计算大小。

struct 也用于定义位域, 就是可以把成员定义成位长的形式, 让位长满足使用即可, 主要是为了节省内存空间, 或者是为了方便地访问某些位, 以及与某些设备寄存器相匹配。成员之间可以有空位域, 就是只有位长, 没有名字, 这是为了成员的位对齐。虽然使用位域能够节省内存空间, 但是会增加代码的执行时间, 因为代码中要增加对位域的抽取和拼接的操作。

例子：位域

```
struct box_props {
    unsigned int opaque: 1;
    unsigned int fill_color: 3;
    unsigned int : 4;
    unsigned int show_border: 1;
    unsigned int border_color: 3;
    unsigned int border_style: 2;
    unsigned int: 2;
};
struct box_props box = {1, 1, 1, 2, 2};
```

可移植性的程序应该避免使用位域, 因为位域在不同的编译器中可能有不同的结果。例如, 位域的成员可能是从左向右分配的, 也可能是从右向左分配的, 不同的编译器可能有不同的分配方式。

25.5 union

union 用于定义联合类型, 它是一种构造类型。联合可以有多个不同数据类型的成员, 这些成员共享同一块内存, 以达到节省空间的目的。联合的长度是由成员中最长的长度决定的。

union 和 struct 都是由多个不同数据类型的成员组成的, 但在任何同一时刻, union 只存放一个被选中的成员, 而 struct 的所有成员都存在。对 union 的一个成员赋值, 就会对其他成员改写, 原来成员的值就不存在, 而对 struct 的各个成员赋值是互不影响的。

例子：联合类型

```
union FOO {int i; char c; double d;};
union FOO foo;
foo.i = 0x12345678;
foo.c = 0xAB;
foo.d = 8765.4321;
```

25.6 typedef

typedef 用来为一种数据类型定义一个新名字, 这里的数据类型包括内部数据类型 (int、

char 等) 和自定义的数据类型 (struct、union 等)。

使用 typedef 目的有两个：一是为类型定义一个意义明确的新名字，二是简化一些比较复杂的数据类型声明。

```
例子: typedef
typedef unsigned int my_size_t;
void measure (my_size_t *psz);
my_size_t array[4];
my_size_t len = getlength();

typedef char Line[81];
Line text; //same as char text[81];
getline (text);
```

```
例子; typedef
typedef struct EMPLOYEE_T {
    int id;
    char name [40];
    char sex;
    int salary;
} EMPLOYEE_T;
EMPLOYEE_T employee;
//或者 struct 后面的名字也可以不要
typedef struct {
    int id;
    char name [40];
    char sex;
    int salary;
} EMPLOYEE_T;
EMPLOYEE_T employee;
```

```
例子; typedef 定义单链表
typedef struct tag_node {
    char name[40];
    int salary;
    struct tag_node *next;
} *NODE_T;
```

使用宏定义也可以定义数据类型，那么哪个更好呢？答案是使用 typedef 更好。例如在下面的例子中，s1 和 s2 都是一样的，都是 char *，而 s3 和 s4 却是不一样的，s3 是 char *，而 s4 是 char。

```
例子: typedef 和 define 的对比
typedef char *pStr1;
#define pStr2 char*
pStr1 s1, s2;
pStr2 s3, s4;
```

编译器会对下面的代码报告一个错误，你知道哪个语句是错的吗？

```
例子: typedef 的陷阱
typedef char *pStr;
char string[4] = "abc";
const char *p1 = string;
const pStr p2 = string;
p1++;
p2++;
```

答案是 p2++ 出错了。const char *p1 是对 p1 指向的数据 (*p1) 加上只读限制，而 p1 本

身没有只读限制,所以 p1++ 没有错误。对于 const pStr p2, 虽然表面上 const pStr p2 与 const char *p2 是一样的, 其实它们两个是不一样的。const pStr p2 和 const long x 本质上是类似的, 都是对变量进行只读限制, 只不过此处变量 p2 的数据类型是我们自己定义的而不是系统固有的类型而已。因此, const pStr p2 的含义是: 数据类型为 char * 的变量 p2 被限定为只读, 因此 p2++ 错误。

这个问题再一次提醒我们: typedef 和 #define 不同, typedef 不是简单的文本替换。

25.7 复杂的数据类型

例子: 使用 typedef 定义函数指针

```
typedef int (*P_OPERATE_DATA)(int type, int offset, void *buffer);
P_OPERATE_DATA func_operate_data;
func_operate_data = socket_operate_data;
int x_type = 0x12;
int x_offset = 0x00;
char x_buffer[256];
(*func_operate_data)(type, offset, buffer);
```

定义和分析复杂的数据类型, 就要考虑操作符的优先级, 先找到名字, 然后从名字开始分析操作符的优先级。对于 P_OPERATE_DATA, 因为括号把它和 * 包在一起, 成为 (*P_OPERATE_DATA), 使得 “*” 的优先级最高, 所以首先 P_OPERATE_DATA 是一个指针, 然后在括号外面定义了函数, 这个函数有 3 个参数且返回值为整数。所以 P_OPERATE_DATA 是一个指针, 是一个指向函数的指针, 简称为函数指针。

25.8 Endian 问题

在各种计算机体系结构中, 因为对字节、字等的存储机制有所不同, 所以引发了计算机通信领域中一个很重要的问题, 即通信双方交流的信息单元 (比特、字节、字、双字等) 应该以什么样的顺序进行传送。如果不达成一致的规则, 通信双方将无法进行正确的编码/译码, 从而导致通信失败。所以, 通常采用 big-endian 和 little-endian 两种字节存储机制, 描述多字节数中各个字节的存储顺序。

Endian 这个词来源于 Jonathan Swift 在 1726 年写的讽刺小说《格列佛游记》。当格列佛在小人国畅游时, 小人国里的小人因为非常小 (身高 6 英寸), 所以总会碰到一些意想不到的问题。有一次因为对水煮蛋是该从大的一端 (Big-End) 剥开还是从小的一端 (Little-End) 剥开的争论而引发了一场战争, 并形成了两支截然对立的队伍。Swift 把支持从大的一端剥开的人称为 Big-Endians, 把支持从小的一端剥开的人称为 Little-Endians, 后缀 ian 表明的就是支持某种观点的人。1980 年, Danny Cohen 在其著名的论文《On Holy Wars and a Plea for Peace》中, 为了平息一场关于在消息中字节该以什么样的顺序进行传送的争论而引用了该词。该文中, Cohen 非常形象贴切地把支持从一个消息序列的最高位开始传送的那伙人叫做 Big-Endians, 把支持从最低位开始传送的那伙人相对地叫做 Little-Endians。此后 Endian 这个词便随着这篇论文而被广为采用。

不同的 CPU 有不同的字节序类型, 字节序是指整数在内存中保存的顺序。最常见的有两种。

- 1) **Little-endian**: 称为小端格式, 把低序字节存储在起始地址 (低位编址)。Little-endian 是最符合人的思维的字节序, 地址低位存储值的低位, 地址高位存储值的高位。为什么最符合人的思维呢? 因为从人的第一观感来说, 低位值小, 就应该放在内存的低地址, 高位值大, 就应该放在内存的高地址。
- 2) **Big-endian**: 称为大端格式, 把高序字节存储在起始地址 (高位编址)。Big-endian 是最直观的字节序, 地址低位存储值的高位, 地址高位存储值的低位。为什么说看着直观呢? 因为只需要把内存地址从左到右按照由低到高的顺序写出, 把值按照通常的高位到低位的顺序写出, 两者对照, 一个字节一个字节地填充进去。

如果把 0x12345678 写到从 0x4000 开始的内存中, 那么这个值在 Little-endian 和 Big-endian 下按表 25-3 所示的不同方式存放。

表 25-3 C 语言操作符

内存地址	Little-endian	Big-endian
0x4000	0x78	0x12
0x4001	0x56	0x34
0x4002	0x34	0x56
0x4003	0x12	0x78

对这些地址按不同的类型访问时, 会有不同的结果。

```
When read them as Little-endian
*(char *)0x4000 --> 0x78
*(char *)0x4001 --> 0x56
*(char *)0x4002 --> 0x34
*(char *)0x4003 --> 0x12
*(short *)0x4000 --> 0x5678
*(short *)0x4002 --> 0x1234
*(int *)0x4000 --> 0x12345678;
When read them as Big-endian
*(char *)0x4000 --> 0x12
*(char *)0x4001 --> 0x34
*(char *)0x4002 --> 0x56
*(char *)0x4003 --> 0x78
*(short *)0x4000 --> 0x1234
*(short *)0x4002 --> 0x5678
*(int *)0x4000 --> 0x12345678;
```

为什么要关注字节序的问题呢? 如果你写的程序只在单机环境下运行, 并且不和别人的程序打交道, 那么你完全可以忽略字节序的存在。但是, 如果你的程序要跟别人的程序进行交互呢? 例如, 你的程序向别人的程序传送 0x12345678, 你们俩的程序采用不同 Endian, 而且传送时又没有任何转换, 那么对方接收的就是 0x78563412。所以, 你们俩的程序要做好约定, 传送时采用哪种 Endian, 程序该怎么做转换。

所有的网络协议都是采用 Big-endian 的方式来传输数据的, 所以有时我们也会把 Big-endian 的方式称之为网络字节序。当两台采用不同字节序的主机通信时, 在发送数据之前必须把字节序转换为网络字节序后再进行传输。ANSI C 中提供了下面四个转换字节序的函数, 它们的函数原型定义在 `netinet/in.h`。

```
hton(unsigned short), hton(unsigned long) //将本地字节序转换为网络字节序
ntohl(unsigned long), ntohs(unsigned short) //将网络字节序转换为本地字节序
```

26.1 声明和定义

变量和函数都有定义和声明的区别，这里说一下。声明（Declaration）描述对象的名称和类型，定义（Definition）则导致为对象分配存储空间。它们的特性如下。

- 1) 同一个对象可以有許多声明，但只能有一个定义，而且声明必须要与定义保持一致。
- 2) 如果一个对象被定义了多次，那么编译或链接就会报告有错误。
- 3) 如果使用了声明的变量或函数，但是没有定义它们，那么链接就会报告错误。
- 4) 对于局部变量，只能定义局部变量，不能像全局变量一样再声明它们。
- 5) 声明时要使用 `extern`，但是对于函数原型可以省略 `extern`。声明函数时，参数名有没有都是可以的，但是建议带上参数名。
- 6) `extern` 的重要用途和多文件的程序有关。当程序有多个文件时，需要分别编译，然后再链接在一起。

例子：定义变量和函数

```
int bar, foo;
int calc (int a, int b) { return (a + b) / (a - b); }
double add_id (int i, double d) { return i + d; }
```

例子：声明变量和函数

```
extern int bar, foo;
extern int calc (int, int); //或者 extern int calc (int a, int b);
double add_id (int, double); //或者 double add_id (int i, double d);
```

26.2 变量分类

26.2.1 全局变量

全局变量（Global Variables）是定义在函数外部的变量，顾名思义，它们是全球存在的。全局变量使用起来很方便，但是也存在着一些特性。

- 1) 全局变量从程序开始时运行，就一直在内存中存在，要在全局数据区中为它们分配空间。
- 2) 全局变量会占用更多的内存，因为它们不会被释放，不能挪作他用。
- 3) 通常使用全局变量不用考虑空间的问题，除非全局变量要求的空间太大，不能满足。
- 4) 全局变量便于在多个函数之间共享数据，可以减少函数调用时传递参数的开销，所以

运行速度会快一些。

- 5) 如果全局变量有初始值，那么程序运行时按初始值初始化，否则就被初始化为 0。
- 6) 全局变量的作用域是从定义处或声明处开始，一直到文件结束。
- 7) 当全局变量与局部变量重名的时候，起作用的是局部变量，全局变量被屏蔽掉。
- 8) 对于变量的名字空间污染，这个问题在小心使用全局变量时是可以避免的。
- 9) 当局部变量满足要求的时候，使用全局变量会让函数的通用性变差，当大量使用全局变量时，可能会引起程序错误，因为在程序的其他地方可能会意外地修改了全局变量的值。

总之，全局变量可以使用，但是不要过度使用，同时应该让全局变量的名字易于理解且不能太短，以避免名字空间的污染。另外，还要避免使用巨大空间的全局变量。

26.2.2 局部变量

局部变量 (Local variables) 是定义在函数内部的变量，是相对于全局变量而言的。局部变量具有如下特性：

- 1) 局部变量不只是可以定义在函数的开头处，也可以定义在复合语句块内，所以局部变量就是定义在“{ }”内的变量。局部变量脱离它所定义的“{ }”，也就不存在了。
- 2) 局部变量是在堆栈区分配，一般堆栈都是向下生长的。当程序运行到定义局部变量的语句时，才会给该局部变量分配内存。
- 3) 局部变量在定义时，可以有初始值，执行到定义局部变量的语句时，才会做相应的初始化。如果局部变量没有初始值，那么它的值是不定的。
- 4) 当局部变量因为“{ }”嵌套发生了重定义，以最近定义的局部变量为准，但是应该避免在嵌套的“{ }”中出现相同名字的临时变量。
- 5) 当全局变量与局部变量重名时，在定义局部变量的“{ }”内，局部变量起作用。
- 6) C99 支持混合声明，就是可以在代码中定义局部变量，这样可以就近使用，缩小局部变量的作用域。另外，还可以使用 (for int i = 0; i < 100; i++) 这样的定义，i 的作用域只在这个循环内有效。
- 7) 很多人习惯在函数的开头定义所有的局部变量，但是当函数太大且变量太多时，就会影响代码的可读性，所以可以考虑在复合语句内定义变量，还可以考虑使用混合声明。

我们可以在函数内定义大的数组，但是尽量不要在函数内初始化大的常数数组，例如下面的代码：

```
void fetch_full_data (int number, char *buffer)
{
    int ar[512] = {0x12345678, 0xABCDDDEAD, ....., 0x55AA55AA};
    .....
}
```

这样做就会导致每次调用函数时，函数都要初始化这个数组，因为 ar[512] 是在堆栈上生成的，退出就不存在了。编译器对这个数组实现初始化的方法是：编译器把这个数组的值放在内存的某个区域（一般是只读数据区），然后调用 memcpy 从这里复制这些常数来初始化 ar[512]。

26.2.3 静态全局变量

静态变量是在变量定义时添加 static 属性的变量，分为静态全局变量和静态局部变量。使

用静态变量，可以减少名字空间污染，使得程序更容易维护。

静态全局变量，也称为文件内静态变量，就是在全局变量前加上关键字 `static`。它的作用域是从定义处开始到文件的结束。多个文件可以定义多个同名的静态全局变量，这些变量的名字不会产生冲突。其他文件还可以定义同名的全局变量，只要编译时不存在名字冲突就行，因为 `extern` 和 `static` 只能选择其一。

```
例子：使用静态全局变量
static int seq_num = 0;
int get_seq_num (void)
{
    return seq_num++;
}
```

26.2.4 静态局部变量

静态局部变量，也称为函数内静态变量，就是在局部变量前加上 `static` 属性。它的作用域是从定义处开始到它所在的“{ }”结束。

通常，在函数体内定义一个变量，每当程序运行到该语句时都会给该局部变量分配栈空间，当程序退出函数时，系统就会收回栈内存，局部变量也就相应地失效。

但是，有时候我们需要保存每次函数执行后的变量值，用于下次函数调用。通常的做法是定义一个全局变量来实现，但这样一来，全局变量已经不属于函数本身，不受函数的单独控制，会给程序的维护带来不便。静态局部变量正好可以解决这个问题，它保存在全局数据区内，而不是保存在栈中，每次的值都能够保持到下一次调用，直到再次赋新值。

虽然静态局部变量始终驻留在全局数据区内，但是它的作用域是局部的，只在定义它的“{ }”中可见，当它脱离了它所在的“{ }”时，它的作用域也就随之结束。

有的书上说静态局部变量是在函数执行到那条定义语句时才做的初始化，这其实是错误的。静态局部变量是在程序开始执行时就做初始化，虽然它的定义语句是在函数内。如果静态局部变量有初始值，那么就把它初始化为初始值；如果没有初始值，就把它初始化为 0。

```
例子：使用静态局部变量
int get_seq_num (void)
{
    static int seq_num = 0;
    return seq_num++;
}
```

26.2.5 对比表格

表 26-1 是这几种变量分类的对比表格，可以对它们做一下快速的比较和分析，根据设计的需要，选择最合适的一种使用。

表 26-1 变量分类的对比

分 类	位 置	作 用 域	初 始 化	说 明
全局变量	在函数外边定义	从定义或声明处到文件的结束	在执行 main()之前执行初始化。如果有初始值，就把它初始化为初始值；如果没有，就把它初始化为 0。	全局变量分为定义和声明。定义时不用 extern，声明时要用 extern。一个全局变量只能有一处定义，但是可以有多处声明。 注意：它是在全局数据区分配
局部变量	在“{ }”定义	从定义处到它所属的“}”	执行到定义的语句时，如果有初始值，就把它初始化；如果没有初始值，就不做初始化，它的值是不定的。	注意：函数参数与局部变量类似，它的作用域是整个函数，它的初值是在函数调用时的实参值。 注意：它是在栈上分配
静态全局变量	全局变量上加 static	从定义或声明处到文件的结束	与全局变量相同	注意：它是在全局数据区分配
静态局部变量	局部变量上加 static	从定义处到它所属的“}”	与全局变量相同	注意：它是在全局数据区分配

26.2.6 代码例子

例子：各种变量类型

```
#include <stdio.h>
int g_data_0 = 0x12345678; //这是一个全局变量，有初始值
int g_data_8; //这是一个全局变量，没有初始值
int static_0 = 0x87654321; //这是一个静态全局变量，有初始值
int static_8; //这是一个静态全局变量，没有初始值
int process_data (int flag, int number) //这两个是函数参数
{
    static int l_count = 8; //这是一个静态局部变量，有初始值
    static int l_position; //这是一个静态局部变量，没有初始值
    int i, j, k; //这些都是局部变量，没有初始值
    int value; //这是一个局部变量，没有初始值

    if (flag == 1) {
        int middle = 0; //这是一个局部变量，有初始值,只在 if(flag == 1){ }内有效
        value = .....
        .....
    }
    else if (flag == 2) {
        for (int i = 0; i < number; i++) //i 是局部变量，只在 for 循环内有效
            .....
        value = .....
        .....
    }

    int sum1, sum2; //这两个是混合声明的局部变量，没有初始值
    for (i = 0; i < number; i++)
    {
        sum1 = .....;
        sum2 = .....;
    }
    value = value + sum1 + sum2;
```

```

    return value;
}

int main (int argc, char *argv[])
{
    int result;           //这是局部变量, 没有初始值
    result = process_data (1, 100, "ABCDEF");
    return 0;
}

```

26.3 const 变量

const 推出的初始目的, 一是为了取代#define 定义的常量, 消除#define 的缺点, 同时继承#define 的优点, 二是用于约束变量, 使得变量只能读, 不能写。

26.3.1 const 和#define

const 定义的常量属于全局变量, 也要占据全局数据空间, 只不过占据的是只读空间, 对应的 section 通常是 “.rodata”, 在嵌入式系统中这个空间可以是 ROM 或者 Nor Flash 区域, 所以每个 const 定义的常量是有地址的。

当代码使用 const 定义的常量时, 汇编代码从只读空间的对应地址中取出来它的值, 然后再使用; 当代码使用#define 定义的常量时, 汇编代码就把它直接放到指令中, 当作指令中的立即数使用。

例子: const 和#define 的区别

```

#define D_PI 3.1415926
double ddd = D_PI;           //直接放到指令中, 就是所谓的立即寻址
const double C_PI = 3.1415926; //把它的值放到只读空间
double ccc = C_PI;           //从只读空间的对应地址中取出它的值, 然后赋给 ccc

```

const 定义的常量的主要好处如下。

- 1) const 定义的常量具有不可变性。例如: 定义 const int MAX=100, 那么 MAX++会产生错误。
- 2) const 和#define 一样用于定义常量, 可以避免意义模糊数字的出现, 可以很方便地进行参数的调整和修改。
- 3) const 定义的常量可以指定不同的类型, 而用#define 定义的常量是没有类型的, 所以 const 定义的常量便于编译器进行类型检查, 提前消除一些编写错误。
- 4) const 定义的常量放到只读空间内, 而不是当作立即数放到代码中, 这样可以节省代码空间, 减少程序运行时的内存分配。

26.3.2 const 的说明

使用 const 可以保护被修饰的东西, 防止意外的修改, 增强代码的健壮性和可靠性, 使用 const 还对阅读理解代码有一定帮助。例如 char *strcpy (char * dst, const char *src), 编译器知道不能对 src 指向的内容做任何改动, 如果代码中对 src 指向的数据有更改, 那么编译就会报告出错误。

通常, 当用 const 修饰普通的变量时, const 只修饰其后的变量, 这时 const 放在类型前还

是类型后并没有区别。例如，`const int x` 和 `int const x` 是没有区别的，都是修饰 `x` 为 `const`。

但是，当 `const` 和指针一起使用时，就让人很困惑，因为 `const` 出现的位置不同，解释也不一样，所以此时不能随便放置 `const`。

对于指针变量 `px`，要知道两点：指针自己的内容是 `px`，指针指向的内容是 `*px`。在分析带有 `const` 约束的指针时，要从右向左分析，例如，`const char *px`，可以理解为 `const char (*px)`，即 `*px` 为 `const`，而 `px` 则是可变的。进一步，我们用下面的例子进行全面的分析。

```

例子：const 限定的内容
//====const 在前面
const int x;           //x 是 const
const char *px;       // *px 是 const, px 可变
const char* const px; //px 和 *px 都是 const
//====const 在后面
int const x;          //x 是 const
char const *px;       // *px 是 const, px 可变
char* const px;       //px 是 const, *px 可变
char const* const px; //px 和 *px 都是 const
    
```

表 26-2 所示的是 `const` 和 “*” 使用的进一步说明。

表 26-2 const 约束指针的说明

代码例子	说明
指针自己的内容可以改变 指针指向的内容不能改变	<pre>int x = 1; int y = 2; const int *px = &x; px = &y; //正确, 因为指针自己的内容可以改变 *px = 3; //错误, 因为指针指向的内容不能改变</pre>
指针自己的内容不能改变 指针指向的内容可以改变	<pre>int x = 1; int y = 2; int* const px = &x; px = &y; //错误, 因为指针自己的内容不能改变 *px = 3; //正确, 因为指针指向的内容可以改变</pre>
指针自己的内容不能改变 指针指向的内容不能改变	<pre>int x = 1; int y = 2; const int* const px = &x; px = &y; //错误, 因为指针自己的内容不能改变 *px = 3; //错误, 因为指针指向的内容不能改变</pre>
是否需要初始值?	<pre>const int *px; 是允许的, 因为指针自己的内容可以改变, 可以没有初始值 int* const px; 是不允许的, 因为指针自己的内容不能改变, 必须要有初始值 const int* const px; 是不允许的, 因为指针自己的内容不能改变, 必须要有初始值</pre>

下面是笔者实际遭遇的例子，`buf_ptr` 里面存的是 8 个 `void *` 数据，希望放到 “.rodata section”，但是编译后发现 `buf_ptr` 被放到了 “.data section”，仔细研究了一下，发现把定义改成 “`void * const buf_ptr[8]`” 后，`buf_ptr` 就被放到了 “.rodata section”，就正确了。

```

例子: const void * vs. void * const
const void * buf_ptr[8] = {(void *)0, (void *)0, (void *)0, (void *)0,
    (void *)0x288000, (void *)0x288400, (void *)0x288800, (void *)0x288C00};
需要改为:
void * const buf_ptr[8] = {(void *)0, (void *)0, (void *)0, (void *)0,
    (void *)0x288000, (void *)0x288400, (void *)0x288800, (void *)0x288C00};
    
```

26.4 volatile 变量

volatile 的表面意思是“易挥发的”，深层意思是“直接存取原始内存地址”，用来修饰被不同线程访问和修改的变量，确保访问 volatile 变量的指令不会被编译器优化掉。如果没有 volatile，就会导致这样的结果：要么无法编写多线程程序，要么不能对程序做优化。

例子：连续写同一个变量

```
PERI_REG[2]=0x12;
PERI_REG[2]=0x34;
PERI_REG[2]=0x56;
PERI_REG[2]=0x78;
```

如果不对 PERI_REG 加 volatile 修饰，那么编译器就会把这四条语句进行优化，认为只保留 PERI_REG[2]=0x78 即可，对于普通的内存，这么做没有任何问题。但是，如果 PERI_REG 对应的是某硬件模块的一个寄存器地址，连续写 4 次，就是要让这个硬件模块执行 4 次不同的操作。编译器把它优化为 PERI_REG[2]=0x78，那就完全错了。

所以，为了避免优化错误，或者在这里使用内嵌汇编，又或者把 PERI_REG 声明为“volatile char *”，而使用 volatile 是最方便的方法。

编译器对于 volatile 变量的读写不做任何优化，当读 volatile 变量时，每次都要读这个变量的值，而不能使用保存在 CPU 寄存器里的备份；当写 volatile 变量时，每次都要写这个变量的值，而不能把多次写合并为一次写。

26.4.1 volatile 的场合

volatile 变量用处很大，需要使用 volatile 的场合有很多。

- 1) 当访问硬件模块的内存映射寄存器时，要按照 volatile 方式访问。
- 2) 中断服务程序和主程序都要使用的全局变量，它们要定义为 volatile。
- 3) 在多线程应用中被多个线程共享的变量，它们要定义为 volatile，还要使用一些同步机制。

编写嵌入式系统的程序员经常要同硬件、中断、RTOS 打交道，这些都要使用 volatile 变量，但是在笔者接触的设计人员中，有些人并不清楚为什么要使用 volatile 变量。

26.4.2 volatile 的说明

对 volatile 变量的解释和对 const 变量的解释是一样的，根据 volatile 位置的不同，会有不同的解释。

例子：volatile 限定的内容

```
//====volatile 在前面
volatile int x;           //x 是 volatile
volatile char *px;       // *px 是 volatile, px 可变
volatile char* volatile px; //px 和 *px 都是 volatile
//====volatile 在后面, 与上面的声明对等
int volatile x;          //x 是 volatile
char volatile *px;       // *px 是 volatile, px 可变
char* volatile px;       //px 是 volatile, *px 可变
```

```
char volatile* volatile px; //px 和*px 都是 volatile
```

另外，volatile 可以和 const 一起使用，例如 const volatile char *px，可以用于描述只读的状态寄存器，因为它的状态可能会发生改变，所以是 volatile，因为程序不能修改它，所以是 const。

26.4.3 volatile 的例子

有时我们会写下面类似的循环让程序延迟一段时间，例如

```
for (int i = 0; i < 100000; i++);
```

当你用-O0 编译的时候，这个 for 循环还能起到延迟一段时间的作用，但是当你用-O1、-O2、-O3 编译的时候，就会发现这个循环完全被优化掉了，延迟也就没了。修改这个问题的方法就是把 i 定义成 volatile，这段代码就不会被优化掉，就会有期望的延迟发生。

```
for (volatile int i = 0; i < 100000; i++);
```

在笔者参与的设计系统中，为了方便地访问硬件模块的寄存器，专门定义了如下三个宏。

```
#define REG32(x) (*(volatile unsigned int*)(x))
#define REG16(x) (*(volatile unsigned short*)(x))
#define REG8(x) (*(volatile unsigned char*)(x))
```

下面的代码是笔者为测试某 SOC 中的 DMAC 所写的代码，这里使用 volatile 的地方有两处：一个是把 dmac_flag 定义成 volatile，因为它是主程序和中断 handler 都要使用的全局变量；另一个是使用 REG32 访问 DMAC 的寄存器。

例子：DMAC 测试程序

```
#include <t_dmac.h>
#define X_CNT 16
#define X_RCH 0
volatile int dmac_flag = 0;
void dmac_handler (struct pt_regs *regs)
{
    int val = REG32(DMA_STATUSTFR);
    REG32(DMA_CLEARFR) = val;
    if (val & (1 << X_RCH))
        dmac_flag++;
}

int test_dmac (void)
{
    int i, val, exp;
    int r_llp, r_buffer, rx_data, rx_total;
    int t_llp, t_buffer, tx_data, tx_total;

    intc_enable_multi_source (1, INTC_NUM_DMACH);
    set_exp (TP_IRQ, INTC_NUM_DMACH, dmac_handler);

    //=====
    dma_enable ();
    tx_total = X_CNT;
    t_buffer = get_memory (tx_total * 4, MEM_ALIGN_BYTE_4,
        MEM_CACHEABLE_NOT);
    tx_data = t_buffer;
    for (i = 0; i < tx_total; i++)
        REG32(tx_data + i * 4) = i;
    rx_total = X_CNT;
```

```

r_buffer = get_memory (rx_total * 4, MEM_ALIGN_BYTE_4,
MEM_CACHEABLE_NOT);
rx_data = r_buffer;
r_llp = get_memory (32, MEM_ALIGN_BYTE_4, MEM_CACHEABLE_NOT);

//=====
dma_init_channel (X_RCH, r_llp, tx_data, rx_data,
                 rx_total, DMA_C_TT_FC_M2M_DMAC,
                 DMA_C_TR_WIDTH_WORD, DMA_C_TR_WIDTH_WORD,
                 DMA_C_ADDR_INC, DMA_C_ADDR_INC,
                 DMA_C_MSIZE_8, DMA_C_MSIZE_8, 0, 0,
                 DMA_C_RELOAD_DIS, DMA_C_RELOAD_DIS, 1);
dma_use_channel (1, X_RCH);

dmac_flag = 0;
while (dmac_flag == 0);

//=====
for (i = 0; i < tx_total; i++) {
    if (REG32(tx_data + i * 4) != REG32(rx_data + i * 4)) {
        show ("Error", 3, i, REG32(tx_data + i * 4), REG32(rx_data + i * 4));
        return 1;
    }
}
return 0;
}

```

26.5 混合声明

在 C89 中,局部变量的定义必须在函数的开头或者在复合语句的开头,也就是必须在“{ ”的后面,在代码中间定义临时变量是不允许的。

在 C99 中,这个约束去掉了,临时变量可以在代码中间定义,也就是可以分散声明变量。这种方式对于大函数很有好处,可以在要使用位置的前面定义变量,这样就可以缩短局部变量的作用域,从而提高代码的可读性。

在 C99 中,还可以在 for 循环的初始化处定义一个或多个临时变量,这种局部变量的使用范围就限制在这个 for 循环内。增加这种定义局部变量的方法很有好处,因为控制 for 循环的变量通常只在循环中使用,在循环初始化时把变量局部化,可以避免不必要的影响。

例子: 在 for 循环的初始化处定义变量

```

main (void)
{
    int i = -99;
    for (int i = 0; i < 10; i++) printf ("%d\n", i); //打印 0~9
    printf ("%d\n", i); //打印-99
    return 0;
}

```

常量是程序不能改变的固定值，可以是任何基本数据类型的值。使用常量可以便于程序的阅读、修改和移植。

27.1 常量类型

常量类型是和 C 语言的内部数据类型相对应的，分为整数常量和浮点数常量，另外把 char 常量称为字符常量，把多个 char 常量称为字符串常量，如表 27-1 所示。

表 27-1 常量类型的说明和举例

类 型	说 明	举 例																												
字符	<p>字符常量是 char 类型常量的另一种表现形式。</p> <p>可以是放在单引号内的单个字符；还可以是放在单引号内的反斜线字符常量，下面是一些编码。</p> <table border="1" data-bbox="307 1042 790 1316"> <tr> <td>\b</td> <td>退格</td> <td>\0</td> <td>0</td> </tr> <tr> <td>\f</td> <td>换页</td> <td>\\</td> <td>反斜线</td> </tr> <tr> <td>\n</td> <td>换行</td> <td>\v</td> <td>垂直制表</td> </tr> <tr> <td>\r</td> <td>回车</td> <td>\a</td> <td>报警</td> </tr> <tr> <td>\t</td> <td>水平制表</td> <td>\?</td> <td>问号</td> </tr> <tr> <td>\"</td> <td>双引号</td> <td>\N</td> <td>八进制常量</td> </tr> <tr> <td>'</td> <td>单引号</td> <td>\xN</td> <td>十六进制常量</td> </tr> </table>	\b	退格	\0	0	\f	换页	\\	反斜线	\n	换行	\v	垂直制表	\r	回车	\a	报警	\t	水平制表	\?	问号	\"	双引号	\N	八进制常量	'	单引号	\xN	十六进制常量	<p>'A', '0', '#'</p> <p>'\t', '\0'</p> <p>用八进制的形式书写为：</p> <pre>#define VTAB '013' //ASCII vertical tab #define BELL '\007' //ASCII bell character</pre> <p>用十六进制的形式书写为：</p> <pre>#define VTAB '\xb' //ASCII vertical tab #define BELL '\x7' //ASCII bell character</pre>
\b	退格	\0	0																											
\f	换页	\\	反斜线																											
\n	换行	\v	垂直制表																											
\r	回车	\a	报警																											
\t	水平制表	\?	问号																											
\"	双引号	\N	八进制常量																											
'	单引号	\xN	十六进制常量																											
整数	<p>整数常量可以按照十进制、八进制、十六进制表示</p> <p>无符号数要在结尾带 u 或 U</p> <p>32 位可以在结尾带 l 或 L</p> <p>64 位必须在结尾带 ll 或 LL</p> <p>所以存在如下这些组合：</p> <p>32 位有符号数可以在结尾带 l 或 L</p> <p>32 位无符号数可以在结尾带 ul 或 UL</p> <p>64 位有符号数必须在结尾带 ll 或 LL</p> <p>64 位无符号数必须在结尾带 ull 或 ULL</p>	<p>1234, 0123456, 0x1234ABCD</p> <p>12345678L</p> <p>12345678UL</p> <p>0x12345678ABCDEF55LL</p> <p>0xABCDEF5587654321ULL</p>																												

续表

类 型	说 明	举 例
浮点数	浮点数常量可以按照常规法或科学记数法表示 float 必须在结尾带 f 或 F double 必须在结尾没有标志 long double 必须在结尾带 l 或 L	1234.5678F //float 1234.5678 //double 1234.5678L //long double -12.34e12 //double
字符串	字符串常量是放在双引号内的多个字符 注意：编程时可以将多个字符串常量连接起来	"I am a programmer" "Hello, World!" "" //空字符串 "Hello, " "world" 等价于 "Hello, world"

27.2 常量定义

为了便于代码的阅读和修改，要把代码中的常量定义成标识符，定义常量的方法有三种，`#define`、`enum` 和 `const`，可以根据需要选择合适的一种使用。

例子：定义常量的方法

```
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
const double TWO_SQRURE_ROOT = 1.414;
#define PI 3.1415926
```

27.3 常量区分

有些人常常不能把 0、`\0`、`'0'`、`NULL` 区别开来，这里说明一下。

- 1) 0 是一个整数常量，它的值就是 0。
- 2) `\0` 是一个字符常量，它的值就是 0，在操作字符串时用它来表示结尾。
- 3) `'0'` 是一个字符常量，它的值是 48。
- 4) `NULL` 是一个指针常量，它的值也是 0，但是它被定义为“`#define NULL ((void *)0)`”，`NULL` 指针是一个无类型指针，定义成这样，主要是为了在给指针变量赋值的时候，或者在和指针类型变量比较的时候，不会产生“数据类型不匹配的”警告。

另外，字符常量与仅包含一个字符的字符串之间，例如 `'x'` 与 `"x"`，有什么区别呢？前者是一个整数，其值是字母 x 在 ASCII 字符集中对应的数值；后者是一个字符串，包含一个字符（即字母 x）和一个结束符 `\0`。

27.4 其他问题

C 语言中的个有特性：相邻的字符串常量自动合并成一个字符串，这样可以便于长字符串的书写。但是由此可能带来书写上的问题，下面是一个字符串常量数组，对应 5 条命令。

```
char *file_cmd[] = {"open", "close", "save", "save as", "exit"};
```

如果不小心把 `"close"` 后面的逗号忘写了，编译器不会报告出错误，因为语法是正确的，编译器会把 `"close" "save"` 连接在一起，当作 `"closesave"`，只有 4 条命令，导致代码执行错误。

另外注意：十进制数和八进制数的区别，不要把它们搞混了，例如 1234 是十进制数，01234 是八进制数。

C 语言拥有丰富的操作符，具有复杂的优先级，是 C 语言的难点之一，如果不注意使用，可能就会构造出难懂或者错误的表达式，所以要仔细研究一下这些操作符。

28.1 操作符表格

复杂表达式的求值顺序是由 3 个因素决定的^[1]：操作符的优先级、操作符的结合性和操作符是否控制求值的顺序。两个相连的操作符哪个先执行取决于它们的优先级，如果两者的优先级相同，那么它们的执行顺序由它们的结合性决定。简单来说，结合性就是一串操作是从左向右依次执行还是从右向左依次执行。最后，还有 4 个操作符（&&、||、?:、逗号），它们可以对整个表达式的求值顺序施加控制，或者保证某个子表达式的求值先于其他子表达式的求值，或者某个子表达式被完全跳过不再求值。

每个操作符的所有属性都列在表 28-1 中。用法示例说明操作符是否要求操作数为左值。lexp 表示左值表达式，rexp 表示右值表达式。左值意味着一个位置，既能获得它的值；也能修改它的值；而右值意味着一个值，只能获得它的值。所以在需要使用右值的地方也可以使用左值，但是在需要使用左值的地方不能使用右值。L-R 表示结合性是从左向右，R-L 表示结合性是从右向左。

表 28-1 C 语言操作符

优先级	操作符	描 述	用 法 示 例	结 果 类 型	结 合 性	是否控制求值顺序
1	()	聚组	(表达式)	与表达式相同	N/A	否
1	()	函数调用	rexp(rexp, ..., rexp)	rexp	L-R	否
1	[]	数组下标引用	rexp[exp]	lexp	L-R	否
1	.	访问结构成员	lexp.member_name	lexp	L-R	否
1	->	访问结构指针成员	rexp->member_name	lexp	L-R	否
1	++	自加（后缀）	lexp++	rexp	L-R	否
1	--	自减（后缀）	lexp--	rexp	L-R	否
2	!	逻辑反	!rexp	rexp	R-L	否
2	~	按位取反	~rexp	rexp	R-L	否
2	+	单目，表示正号	+rexp	rexp	R-L	否

续表

优先级	操作符	描述	用法示例	结果类型	结合性	是否控制 求值顺序
2	-	单目, 表示负号	-rexp	rexp	R-L	否
2	++	自加 (前缀)	++lexp	rexp	R-L	否
2	--	自减 (前缀)	--lexp	rexp	R-L	否
2	*	单目, 间接访问	*rexp	lexp	R-L	否
2	&	单目, 取变量地址	&lexp	rexp	R-L	否
2	sizeof	计算所需空间	sizeof(rexp) or sizeof (类型)	rexp	R-L	否
3	(类型)	强制类型转换	(类型) rexp	rexp	R-L	否
4	*	乘法	rexp * rexp	rexp	L-R	否
4	/	除法	rexp / rexp	rexp	L-R	否
4	%	取模	rexp % rexp	rexp	L-R	否
5	+	加法	rexp + rexp	rexp	L-R	否
5	-	减法	rexp - rexp	rexp	L-R	否
6	<<	左移位	rexp << rexp	rexp	L-R	否
6	>>	右移位	rexp >> rexp	rexp	L-R	否
7	>	大于	rexp > rexp	rexp	L-R	否
7	>=	大于等于	rexp >= rexp	rexp	L-R	否
7	<	小于	rexp < rexp	rexp	L-R	否
7	<=	小于等于	rexp <= rexp	rexp	L-R	否
8	==	等于	rexp == rexp	rexp	L-R	否
8	!=	不等于	rexp != rexp	rexp	L-R	否
9	&	位与	rexp & rexp	rexp	L-R	否
10	^	位异或	rexp ^ rexp	rexp	L-R	否
11		位或	rexp rexp	rexp	L-R	否
12	&&	逻辑与	rexp && rexp	rexp	L-R	是
13		逻辑或	rexp rexp	rexp	L-R	是
14	?:	条件操作	rexp ? rexp : rexp	rexp	N/A	是
15	=	赋值	lexp = rexp	rexp	R-L	否
15	+= -= *= /= %= ^= = <<= >>=	复合赋值	lexp x= rexp	rexp	R-L	否
16	,	逗号	rexp, rexp	rexp	L-R	是

28.2 操作符解释

28.2.1 优先级

当表达式使用多个操作符并且它们的优先级不同时，优先级高的先执行，优先级低的后执行。但是因为 C 语言拥有太多的操作符，所以经常会在优先级上出问题。下表就是操作符优先级存在的常见问题^[12]。

表 28-2 操作符优先级存在的常见问题

优先级问题	表 达 式	误以为的结果	实际的结果
“.” 高于 “*” <-用于消除这个问题	*p.f	认为是(*p).f p 所指对象的成员 f	实际是*(p.f)。 取 p 的成员 f，然后取其指向的内容
“[]” 高于 “*”	int *ap[]	认为是 int (*ap[]) ap 是个指向 int 数组的指针	实际是 int *(ap[]) ap 是个元素为 int 指针的数组
“()” 高于 “*”	int *fp()	认为是 int (*fp()) fp 是个函数指针，所指函数返回 int	实际是 int *(fp()) fp 是个函数，返回值是 int *
比较操作高于位操作	val & mask != 0	认为是(val & mask) != 0	实际是 val & (mask != 0)
比较操作高于赋值	c=getchar() !=EOF	认为是(c=getchar()) !=EOF	实际是 c=(getchar() !=EOF)
算数操作高于移位操作	msb << 4 + lsb	认为是(msb << 4) + lsb	实际是 msb << (4 + lsb)
逗号是优先级最低的	i = 1, 2	认为是 i = (1, 2)	实际是(i = 1), 2

例子：理解下面的优先级

```
val = x + y * z + a / b + ~c - d << e;
val = (x + (y * z) - (a / b) + (~c) - d) << e; //等价
val = x + (y * z) - (a / b) + (~c) - (d << e); //不等价，但是很多人认为是这个
if (x & 0xFE > 0x78)
if (x & (0xFE > 0x78)) //等价，进一步化简，就变为 if (x & 1)
if ((x & 0xFE) > 0x78) //不等价
```

移位操作比算术操作的优先级低，但是很多人经常误以为移位操作比算术运算的优先级高，结果导致代码出错。

位操作（&、|、^）比逻辑操作（&&、||）的优先级高一点，但是比算术操作、移位操作、比较操作的优先级都低，但是很多人经常误以为位操作的优先级很高，结果导致代码错误。

为了避免构造出错误或者难以理解的表达式，有一个方法是使用括号，明确地指出操作的优先级，另一个就是不要构造太长的表达式，通过添加中间变量，降低表达式的长度。

另外，只有掌握了 C 语言的优先级，我们才能轻松地解读和定义复杂的声明，例如下面这几个声明。

例子：复杂的声明

```
int (*bar1(void))(int); //bar1 是一个函数，返回值一个指针，指针指向返回值为 int
的函数
int (*bar2(void))[10]; //bar2 是一个函数，返回值一个指针，指针指向返回值为 int[10]
的数组
int (*bar3[20])(int); //bar3 是一个数组，元素为函数指针
```

对这几个声明的解读是不是有些费劲呢？那么如何降低声明的复杂度呢？如同不要构造长表达式一样，也不要构造过长的声明，可以通过使用 `typedef` 来降低声明的复杂度。

例子：简化的声明

```
typedef int (*func_p)(int); //func_p 为返回值为 int 的函数指针
func_p bar1(void); //bar1 是一个函数，返回值为 func_p
typedef int (*ar_p)[10]; //ar_p 是指向 int[10] 的数组指针
ar_p bar2(void); //bar2 是一个函数，返回值为 ar_p
func_p bar3[20]; //bar3 是一个数组，元素为 func_p
```

28.2.2 结合性

结合性是在多个操作符的优先级相同时使用的规则，每个操作符都有固定的结合性，从左向右结合，或者从右向左结合。

除了后缀++和--，所有的单目操作、sizeof、强制转换、赋值操作、复合赋值的结合性都是从右向左依次执行（R-L）。

例子：从右向左操作，这样的代码很少见，所以最好不要这样写

```
++ ++x
-- ~ ++x
! ~ -x
~ - sizeof(x)
x = y = z; //相当于(x = (y = z));
x += y += z; //相当于(x += (y += z));
```

对于其他的操作，包含后缀++和--，它们的结合性都是从左向右依次执行（L-R）。

例子：从左向右操作，这样的代码更常见

```
employee.name[2]
node->next->name
x++ ++
x + y - z
x * y / z
x & y & z
```

28.2.3 求值顺序

对于这四种操作（&&、||、?:、逗号），它们可以对整个表达式的求值顺序施加控制，或者保证某个子表达式的求值先于其他子表达式的求值，或者某个子表达式被完全跳过不再求值。

对于其他的操作，就没有这个要求了，没有规定子表达式之间的求值顺序，编译器可以实现任意的求值顺序。例如，对于 $z = f1() + f2()$ ，编译器并不能保证 $f1()$ 比 $f2()$ 先执行， $f1()$ 和 $f2()$ 哪个先做都可以。

另外，表达式的求值顺序并非完全由操作符决定，下面这样的语句就很危险，在不同的编译器可能会有不同的结果。

例子：危险的语句

```
y = x - ++x;
```

这里++x 肯定会先执行，但是减法操作的左操作数 x 是++x 之前的值呢，还是之后的值呢？C 语言标准规定类似这样的表达式的值是未定义的，所以一定不要写这样的表达式。

例子：危险的语句， $y[i]$ 中 i 的值也是难以确定的

```
i = 0;
while ( i < n)
    y[i] = x[++i];
例子：危险的语句，有的编译器打印 9，有的编译器打印 12
int i = 3;
printf ("%d", (i++)*(i++));
```

28.2.4 易混淆的

下面是容易混淆的操作符，因为理解错误、区分不开，可能会导致书写上的错误，也可能导致误用的情况，如表 28-3 所示。

表 28-3 易混淆的操作符

操作符	操作符	说明
&& !	& ~	误用逻辑操作和位操作
=	==	误用赋值和等于
*		*具有多个用途，使用时可能会出现问题
&		&具有多个用途，使用时可能会出现问题
<=	<<=	误用小于等于和左移复合赋值

这些易混淆的操作符很像汉字“做、作”、“像、象”、“的、地、得”等，汉字用错了，还能从字面上看出是什么意思，但是操作符用错了，就会导致严重的代码错误。

28.2.5 左值右值

下面的 char *p 恰好指向一个 int 类型的数据，要想对其做自增运算，下面的代码正确吗？

```
例子：char *p;
((int *)p)++;
```

答案是错误的。首先要明白左值和右值的概念，左值就是可以出现在“=”号的左边，可以被赋值；右值是只能出现在“=”号的右端，不能被赋值也不能自增。由于(int *)p 是经过类型转换得到的，所产生的结果是一个右值，所以不能对其进行修改和自增操作。

28.2.6 赋值方式

最好不要在表达式里放置赋值语句，也不要而在 while 和 if 的判断中放置赋值语句。

```
例子：赋值语句
r = s + (t = u - v) / 3;
if ((ch == getchar ()) == EOF)
    break;
//最好改为如下
t = u - v;
r = s + t / 3;
ch == getchar ();
if (ch == EOF)
    break;
```

如果表达式类似于“x = x + 5; y = y * z”;，可以使用复合赋值，简化书写为“x += 5; y *= z;”，这样更方便书写。注意：可以把 i = i + j 写成 i += j，但不要写成 i =+ j，虽然后者是写

错的，但是也可以编译通过。注意： $i * j + k$ 等价于 $i = i * (j + k)$ ，而不是 $i = i * j + k$ 。

如果出现这样的情况“ $z = a + b; y = a + b; x = a + b;$ ”，那么可以使用多重赋值，简化为“ $x = y = z = a + b;$ ”，但是实际情况不多，使用的意义也不大。如果 x 、 y 、 z 的位长或类型不一样，那么多重赋值的结果可能会发生错误，例如， x 和 z 是 `int` 型， y 是 `char` 型，那么就会出现错误。又例如，对于“`int x; float y; y = x = 12.5;`”， y 的值将是 12.0，而不是 12.5。

28.2.7 ++和--

有些人觉得把++和--结合到其他表达式中，代码看起来会更紧凑，而且会产生更短更快的汇编代码。

```
例子：strcpy 使用++和*
char* strcpy (char* des, const char* src)
{
    char *r = des;
    while ((*des++ = *src++) != '\0');
    return r;
}
```

对于有些编译器，可能会对这样的 `strcpy()` 产生很好的汇编代码，但这不是一定的，对于别的编译器，这样写可能根本就不会产生什么优化的代码，与分开写的效果是一样的。

另外，有些合并是毫无用处的，对于下面的代码改动，根本就不会产生更好的代码，而且还降低了代码的可读性。

```
例子：毫无意义的改动
a = a + b;
b++;
改为
a = a + b++;
```

这种写法只是减少了一行代码，看起来更紧凑一些，但是并不能生成更好的汇编代码，所以为了保证代码的可读性，还是要尽量单独使用++和--。

28.2.8 &和*

注意：`&`有两个用途，在双目操作时用于表示位与，在单目操作时用于取变量的地址，不要搞错了。

注意：`*`有多个用途，在描述类型时用于说明指针；在双目操作时用于表示乘法；在单目操作时用于指针的间接寻址，就是返回指针指向的内容。例如，`*p`就用于返回指针 `p` 指向地址中存的内容。

在很多代码中，++、--经常和*一起使用，操作指针和从指针指向的变量取值。但是尽量不要把*和++、--等操作一起使用，因为会造成很多理解上的困难。例如，“`char ch[] = "abcdef"; char *cp = ch;`”，那么对于如表 28-4 所示的这些操作，你能都解释出来吗？这些操作是不是看着很眼晕呢？如果你有同样的感觉，你就不要写这种表达式。

在某些情况下，使用指针运算代替数组索引，可能会产生稍短的代码。

```
例子：使用指针运算代替数组索引
for (i = 0; i <100; i++) {
    a = ar[i];
    .....
```

```

}
//改为指针方式
p = ar;
for (i = 0; i <100; i++) {
    a = *p++;
    .....
}

```

表 28-4 复杂的*表达式的解释

表达式	等价于	结果	说 明
*cp + 1	(*cp) + 1	'b'	先取出 cp 指向的内容'a'，然后计算'a+1
*(cp + 1)		'b'	取出 (cp+1) 指向的内容'b'
++cp	*(++cp)	'b'	先执行++cp，然后取出 cp 指向的内容'b'
*cp++	*(cp++)	'a'	先获得 cp 的值 P，计算 cp++，然后取出 P 指向的内容'a' 或者，先获得 cp 的值 P，取出 P 指向的内容'a'，然后计算 cp++
++*cp	++(*cp)	'b'	先取出 cp 指向的内容'a'，然后计算'a+1，最后把'b'存回 cp 指向的地址
(*cp)++		'a'	先取出 cp 指向的内容'a'，然后计算'a+1，最后把'b'存回 cp 指向的位置
++*++cp	++>(*++cp)	'c'	先执行++cp，然后取出 cp 指向的内容'b'，再执行'b+1，最后把'c'存回 cp 指向的位置
++*cp++	++*(cp++)	'b'	先获得 cp 的值 P，计算 cp++，然后取出 P 指向的内容'a'，计算'a+1，最后把'b'存回 P 指向的位置

这只是简单的程序，看着还算清晰，可能就节省两三条汇编指令。当数组的索引是由很复杂的表达式计算出来时，使用指针就会降低代码的可读性。不要为了一点点运行时的效率，付出很大的代价：程序难以编写在前，难以维护在后。对于现在优化的编译器、高速的处理器和巨大的内存，实在没有使用指针运算的必要。

另外，你想一下是使用 ar[i]方便，还是*p++方便，是*(p+i)方便，还是 p[i]方便。笔者认为还是 ar[i]和 p[i]使用方便吧！因为“[]”比“*”的优先级高，使用“[]”就可以减少括号的使用。所以对于数组或者指向多个数据的指针，要尽量使用 ar[i]和 p[i]的方式；对于只是指向一个数据的指针，你可以选择使用使用*p 或者 p[0]，就看你的意愿了。但是尽量不要使用*(p+i)形式，因为看着就啰嗦。

28.2.9 “.” 和 “->”

“.” 和 “->” 用于引用结构和联合中的各个元素，“.” 直接用于结构和联合类型的变量，“->” 则用于指向结构和联合类型的指针。

```

例子：
struct EMPLOYEE employee;
struct EMPLOYEE *p = &employee;
employee.id = 100;
p->id = 100;
//其实访问 p->id 还有一个等价的写法，可以写成如下的样子，“()”是必须的，因为“*”比“.”的优先级低。
//很少有人会写成这样，因为啰嗦。
(*p).id = 100;

```


28.2.10 “?:”

“?:”是个强有力的操作,常用于简化 if/else 语句。对于 $x = \text{cond} ? a : b$, 如果 cond 为 TRUE, 那么只计算 a 的值, 否则只计算 b 的值, 这也是与 if/else 语句相符的。

例子: 使用“?:”可以得到更清晰的语句

```
if (cond1)      x = a;
else if (cond2) x = b;
else if (cond3) x = c;
else if (cond4) x = d;
else if (cond5) x = e;
else           x = f;
//等价于下面的写法, 书写“?:”时, 尽量一行写一个 cond, 看着更清晰
x = (cond1 ? a :
     cond2 ? b :
     cond3 ? c :
     cond4 ? d :
     cond5 ? e : f);
```

28.2.11 算术操作

注意: “%”要求操作数只能是整数, 否则编译无法通过。如果参数是浮点数, 可以使用 fmod()。

注意: 当“/”和“%”用于负操作数时, 在 C89 中其结果是不定的, 例如, $-9/7=-1$ 、 $-9\%7=-2$, 或者 $-9/7=-2$ 、 $-9\%7=5$ 。但是, 在 C99 中, 除法的结果总是向零截取, 所以只有 $-9/7=-1$, $-9\%7=-2$ 。

28.2.12 逻辑操作和位操作

逻辑操作包含 &&、||、!, 位操作包含 &、|、~、^, 但是经常出现错误的使用。

逻辑操作的说明如下。

- 1) 逻辑操作结果的值只有两个值, 0 或者 1。
- 2) 对于“&&”, 只有当它的两个操作数都不是 0 时, 结果才等于 1, 否则如果任意一个操作数为 0, 结果就等于 0。
- 3) 对于“||”, 只有当它的两个操作数都是 0 时, 结果才等于 0, 否则如果任意一个操作数不为 0, 结果就等于 1。
- 4) 对于“!”, 只有当它的操作数是 0 时, 结果等于 1, 否则当操作数不为 0 时, 结果等于 0。
- 5) 对于“&&”和“||”, 支持短路求值。对于 $a\&\&b$, 如果计算发现 a 的值等于 0, 就不再计算 b 的值。对于 $a||b$, 如果计算发现 a 的值等于 1, 就不再计算 b 的值。

对于位操作, 我们只要知道是按位一位一位地操作, & (与)、| (或)、~ (取反)、^ (异或)。另外位操作没有短路求值, 因为根本不可能。

对于下面的代码, 把“&&”写成“&”, 会有什么后果呢? [Andrew Koenig]

例子: &&误写成&

```
int i = 0;
while (i < tabsize && tab[i] != x)
    i++;
//写成如下
```

```
int i = 0;
while (i < tabsize & tab[i] != x)
    i++;
```

使用“&”的代码可能会侥幸地工作，因为如下：

- 1) $i < \text{tabsize}$ 和 $\text{tab}[i] \neq x$ 的求值结果为 0 或者 1，没有其他值。因为只要 a 和 b 的取值只有 0 和 1，那么 $a \& b$ 其实和 $a \& \& b$ 是等价的。
- 2) 当使用“&&”时，因为“&&”支持短路求值，所以当 $i = \text{tabsize}$ 时，就不会访问 $\text{tab}[i]$ 。但是，当使用“&”时，“&”不支持短路求值，当 $i = \text{tabsize}$ 时，就还要访问 $\text{tab}[i]$ ，以检查 $\text{tab}[i] \neq x$ ，因为仅仅是读取它的值，一般不会有有什么危险。但是，也说不定当 $i = \text{tabsize}$ 时访问 $\text{tab}[i]$ ，会产生什么异常呢！

如果有人给你提出这个问题：假设两个整型变量，如何不用额外的临时变量，就可以交换这两个变量的值？这只是一个小技巧罢了！

例子：不用临时变量交换两个整数

```
int a = 1234;
int b = 5678;
a ^= b;
b ^= a;
a ^= b;
```

28.2.13 比较操作

比较操作包含： $>$ 、 $>=$ 、 $<$ 、 $<=$ 、 $==$ 、 $!=$ ，它们的运算结果只有两个值，0 或者 1。

注意：不要把“=”和“==”搞错了，把赋值的地方错写成“==”，把比较的地方错写成“=”。

注意：不要写 $i < j < k$ ，虽然它也是合法的，但是它实际等价于 $(i < j) < k$ ，并不是判断 j 是否在 i 和 k 之间。

28.2.14 sizeof

使用 `sizeof` 具有很大的好处：第一，`sizeof` 可以帮助计算数组元素的个数，因为在修改代码过程中，数组元素的个数可能会改变；第二，`sizeof` 可以帮助确定要分配内存的字节数，使得程序可以更方便地跨系统使用，因为在不同的系统上，同一种类型或同一个结构的字节数就有可能不同。

例子：使用 `sizeof`，不太灵活

```
int ar[] = {11, 22, 44, 77, 99};
int count = sizeof (ar) / sizeof (int);
//改为 double
double ar[] = {11, 22, 44, 77, 99};
int count = sizeof (ar) / sizeof (double);
```

本处的代码不太灵活，因为如果 `ar` 的类型变了，`sizeof` 也需要做相应的改变。

例子：使用 `sizeof`，更加灵活

```
int ar[] = {11, 22, 44, 77, 99};
int count = sizeof (ar) / sizeof (ar[0]);
for (i = 0; i < count; i++) ar[i] *= 100;
```

使用 `sizeof(ar[0])`后，可以保证代码有更大的灵活性，因为即使 `ar` 的类型发生改变，也不

用修改代码。

```
例子：使用 sizeof，计算动态分配的字节数
p = malloc (100 * sizeof (struct EMPLOYEE));
```

28.3 强制进行类型转换

自动转换是在源类型和目标类型兼容时自动做的转换，而强制类型转换是把表达式的运算结果强制转换成类型说明符所表示的类型，操作为：`(type)(expression)`。例如：`(float)a` 就把 `a` 转换为浮点型，`(int)(x+y)` 就把 `x+y` 的结果转换为整型。使用强制转换时要注意以下问题。

- 1) `type` 和 `expression` 都必须加括号（对于单个变量，就可以不加括号），如果把 `(int)(x+y)` 写成 `(int)x+y`，则成了把 `x` 转换成 `int` 型之后再与 `y` 相加。
- 2) 无论是强制转换还是自动转换，都只是为了本次运算的需要而对变量的数据类型进行的临时性转换，而不会改变该变量定义时的数据类型。

```
例子：强制转换
int main (void)
{
    float f=6.45, g = -6.45;
    printf("f=%d, f=%f\n", (int)f, f);      //f=6, f=6.450000
    printf("g=%d, g=%f\n", (int)g, g);     //g=-6, g=-6.450000
}
```

浮点数强制转换为整数，相当于把小数部分直接删去，只保留整数部分。另外，这里还有一个自动转换，就是把 `float f` 和 `g` 传递给 `printf()` 时，它们被自动转换为 `double`。

要特别注意 `char` 和 `unsigned char` 转换成 `int` 时的值，可以尝试下面的程序，看是否与你想的一样。

```
例子：打印 char 和 unsigned char 向 int 的强制转换
#include <stdio.h>
int main (void)
{
    int i;
    for (i = 0; i < 256; i++) {
        char sc = i;
        unsigned char uc = i;
        printf ("%d  %x  %x  %x  %x\n",
                i, (int)sc, (unsigned int)sc, (int)uc, (unsigned int)uc);
    }
    return 0;
}
```

28.4 运算时的类型转换

在表达式中混用不同类型的常量和变量时，它们全都转换成同一种类型。编译器把所有的操作数转换成尺寸最大的操作数类型，称为类型提升 (Type promotion)。首先，所有的 `signed` 和 `unsigned char/short` 都提升为 `int`，这一过程称为整数提升 (Integral promotion, ANSI C 标准)。一旦完成了整数提升，所有其他的变化随着操作进行，按以下算法进行自动转换^[赫伯特·希尔特]。

例子：自动转换算法

```
//double <-- float <-- unsigned long <-- long <-- unsigned int
if one operand is double, then convert another operand to double
else if one operand is float, then convert another operand to float
else if one operand is unsigned long, then convert another operand to unsigned
long
else if one operand is long, then convert another operand to long
else if one operand is unsigned int, then convert another operand to unsigned
int
```

还有一个特例：如果一个操作数是 long，而另一个是 unsigned int 常量，同时 unsigned int 常量不能用 long 表示，则两个操作数都转换为 unsigned long。

对表达式实行各条转换规则之后，每对操作数的类型完全一致，操作结果的类型与每对操作数的类型相同。

例子：类型转换的典型例子

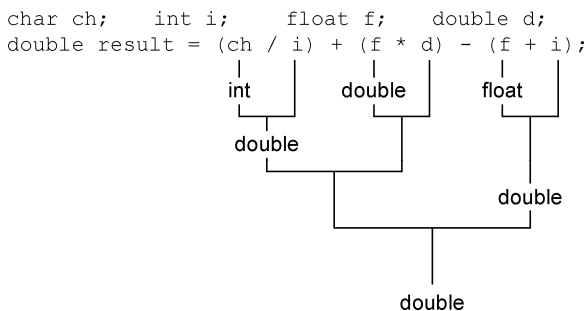


图 28-1 类型转换的典型例子

当表达式中混有各种不同的类型的时候，就会发生自动类型转换，有时会导致最后的运算结果不你所期望的，这时你可能会埋怨编译器不好。但是通常不是编译器不好，而是你没有正确理解 C 语言的类型转换规则。尤其是当有符号数和无符号数在一起的时候最容易出问题，所以对无符号数的使用有如下的建议^[12]：

- 1) 尽量不要在代码中使用无符号数类型，以免增加不必要的复杂性。不要仅仅因为要表达的数据不存在负值（如年龄、工资），就用无符号数来表示它们。
- 2) 尽量使用像 int 这样的有符号数类型，这样在涉及提升混合类型的复杂细节时，不必担心边界的情况，例如，与-1 比较。只有在使用位域和二进制掩码时，才可以使用无符号数。
- 3) 应该在表达式中使用强制转换，使操作数均为有符号数或无符号数，这样就不必由编译器来选择结果类型。

例子：这个循环有什么问题？

```
unsigned int i;
for (i= 100; i >= 0; i--) { ..... }
```

这是个死循环，因为 i 是无符号数，而无符号数始终是大于或等于 0 时，所以 i>=0 是始终为 TRUE，从而导致死循环。

28.4.1 代码例子—A

例子：判断下面的运行结果

```
#include <stdio.h>
```

```
int main (void)
{
    int i = -2;
    unsigned int j = 1;
    if (i + j < 0)
        printf ("less\n");
    else if (i + j == 0)
        printf ("equal\n");
    else
        printf ("large\n");
    return 0;
}
```

运行结果为: large

因为 `j` 是 `unsigned int`, `i` 就要从 `int` 转换为 `unsigned int`, 转换后的值为 `0xFFFFFFFFE`, 相加后的值为 `0xFFFFFFFF`, 大于 0。

例子: `sizeof` 问题, 为什么结果为 0?

```
int result = -1 < sizeof (char);
printf ("%d\n", result);
```

因为 `sizeof` 返回值的类型为 `unsigned int`, 那么比较就要在两个 `unsigned int` 之间进行, `-1` 从 `int` 转为 `unsigned int` 之后值为 `0xFFFFFFFF`, 所以结果为 0。

例子: `strlen` 问题, 为什么判断总为 `true`?

```
if (strlen (str1) - strlen (str2) >= 0) ...
```

因为 `strlen` 的返回值是 `size_t` 类型, 也就是 `unsigned int`, 那么两个 `unsigned int` 相减不可能为负数, 所以结果始终为 `true`。修改它的方法有两个, 如下所示。

```
if ((int)(strlen (str1)) - (int)(strlen (str2)) >= 0) //这个啰嗦, 因为使用强制转换
if (strlen (str1) > strlen (str2)) ... //这个更好, 直接比较
```

28.4.2 代码例子—B

例子: 判断一下下面的运行结果

```
#include <stdio.h>
int cmp_si_uc (int a, unsigned char b) { return a < b; }
int cmp_si_us (int a, unsigned short b) { return a < b; }
int cmp_si_ui (int a, unsigned int b) { return a < b; }
int main (void)
{
    printf ("%d %d %d\n", cmp_si_uc (-1, 1), cmp_si_us (-1, 1), cmp_si_ui (-1, 1));
    return 0;
}
```

运行结果为: 1 1 0

对于前两个比较, `unsigned char` 和 `unsigned short` 在做提升时, 都提升为 `int`, 那么这时比较就是在两个 `int` 之间进行, 那么很自然 `-1 < 1` 的结果为 1。

对于后一个比较, 比较是在 `int` 和 `unsigned int` 之间进行, `int` 要转换为 `unsigned int`, 那么这时比较就是在两个 `unsigned int` 之间进行, 很自然 `0xFFFFFFFF < 1` 的结果为 0。

28.4.3 代码例子—C

例子: 判断下面的运行结果

```
#include <stdio.h>
int main (void)
{
    unsigned char a0 = 6; int b0 = -20; int c0 = (a0 + b0 > 7);
    unsigned char a1 = 6; char b1 = -20; int c1 = (a1 + b1 > 7);
    unsigned int a2 = 6; char b2 = -20; int c2 = (a2 + b2 > 7);
    unsigned int a3 = 6; int b3 = -20; int c3 = (a3 + b3 > 7);
    printf ("%d %d %d %d\n", c0, c1, c2, c3);
    return 0;
}
运行结果为: 0 0 1 1
```

对应的解释如下。

- 1) 对于“`unsigned char a0 = 6; int b0 = -20; int c0 = (a0 + b0 > 7);`”，因为 `a0` 要提升为 `int` 类型的 6，`b0` 恰好为 `int` 类型的 -20，`a0+b0` 是在两个 `int` 类型之间进行，结果也为 `int` 类型的 -14，所以等于 0。
- 2) 对于“`unsigned char a1 = 6; char b1 = -20; int c1 = (a1 + b1 > 7);`”，因为 `a1` 要提升为 `int` 类型的 6，`b1` 要提升为 `int` 类型的 -20，`a1+b1` 是在两个 `int` 类型之间进行，结果也为 `int` 类型的 -14，所以等于 0。
- 3) 对于“`unsigned int a2 = 6; char b2 = -20; int c2 = (a2 + b2 > 7);`”，因为 `a2` 保持为 `unsigned int` 类型的 6，`b2` 要提升为 `unsigned int` 类型的 `0xFFFFFEC`，`a2+b2` 是在两个 `unsigned int` 类型之间进行，结果也为 `unsigned int` 类型，值为 `0xFFFFFFFF2`，所以等于 1。
- 4) 对于“`unsigned int a3 = 6; int b3 = -20; int c3 = (a3 + b3 > 7);`”，因为 `a3` 保持为 `unsigned int` 类型的 6，`b3` 要转换为 `unsigned int` 类型的 `0xFFFFFEC`，`a3+b3` 是在两个 `unsigned int` 类型之间进行，结果也为 `unsigned int` 类型，值为 `0xFFFFFFFF2`，所以等于 1。
- 5) 注意：-20 和 4294967276 的十六进制值都是 `0xFFFFFEC`，按照 `int` 类型和 `unsigned int` 类型解释就有不同的值。

28.4.4 代码例子—D

例子：判断一下下面的运行结果

```
char sc = -1;
printf ("%d %d %d\n", sc == -1, sc == -1U, sc == 0xFF);
unsigned char uc = -1;
printf ("%d %d %d\n", uc == -1, uc == -1U, uc == 0xFF);
```

对应运算结果用表 28-5 说明。

表 28-5 字符比较的运算表格

运 算	结果	左侧提升为	右侧值	说 明
<code>sc == -1</code>	1	<code>0xFFFFFFFF</code>	<code>0xFFFFFFFF</code>	<code>sc</code> 从 <code>char</code> 提升为 <code>int</code> ，还要保持为 -1
<code>sc == -1U</code>	1	<code>0xFFFFFFFF</code>	<code>0xFFFFFFFF</code>	<code>sc</code> 从 <code>char</code> 提升为 <code>int</code> ，再转为 <code>unsigned int</code> 因为 -1U 是 <code>unsigned</code> ，要求两个操作数都为 <code>unsigned</code>
<code>sc == 0xFF</code>	0	<code>0xFFFFFFFF</code>	<code>0x000000FF</code>	<code>sc</code> 从 <code>char</code> 提升为 <code>int</code>
<code>uc == -1</code>	0	<code>0x000000FF</code>	<code>0xFFFFFFFF</code>	<code>uc</code> 从 <code>unsigned char</code> 提升为 <code>int</code>
<code>uc == -1U</code>	0	<code>0x000000FF</code>	<code>0xFFFFFFFF</code>	<code>uc</code> 从 <code>unsigned char</code> 提升为 <code>int</code> ，再转为 <code>unsigned int</code> 因为 -1U 是 <code>unsigned</code> ，要求两个操作数都为 <code>unsigned</code>
<code>uc == 0xFF</code>	1	<code>0x000000FF</code>	<code>0x000000FF</code>	<code>uc</code> 从 <code>unsigned char</code> 提升为 <code>int</code>

虽然 `uc` 被赋值为 `-1`，但是 `uc` 只是取了 `-1` 的低 8 位，高 24 位直接舍去，所以 `uc` 的值是 `0xFF`。在运算时，常数缺省都是按照 32 位符号数处理，`-1` 是 `0xFFFFFFFF`，`0xFF` 是 `0x000000FF`，虽然 `-1U` 也是 `0xFFFFFFFF`，但是它是无符号数。

28.4.5 代码例子—E

例子：判断一下下面的运行结果

```
#include <stdio.h>
main (void)
{
    char c = 0x78;
    int i = c + c;
    printf ("%x\n", i);
    i = (int)c + c;
    printf ("%x\n", i);

    int k = 0x76543210;
    long long ll = k + k;
    printf ("%llx\n", ll);
    ll = (long long)k + k;
    printf ("%llx\n", ll);
    return 0;
}
运行结果为: f0    f0    ffffffffeca86420    eca86420
```

对应的解释如下。

- 1) 因为 `c` 自动转成 `int` 类型后的值为 `0x78`，强制转换 `(int)c` 后的值也为 `0x78`，所以 `i=c+c` 和 `i=(int)c+c` 的值是一样的，都是 `0xF0`。
- 2) 因为 `k+k` 是在两个 `int` 类型之间做运算，运算结果超出了 `int` 类型的最大值，发生了溢出，变成了负数，然后在做赋值转换时，从 32 位扩充为 64 位，要做 `sign_extend`，所以结果为 `0xffffffffeca86420`。
- 3) 因为 `(long long)k` 把 `k` 强制转换为 `long long` 类型，转换后的值为 `0x0000000076543210`，然后 `(long long)k+k` 是在两个 `long long` 之间做运算，`k` 要做自动转换，转换后的值也是 `0x0000000076543210`，最后的结果就为 `0x00000000eca86420`，打印时高位的 0 没有被打印。

28.5 赋值时的类型转换

当赋值运算符两边的运算对象类型不同时，将要发生类型转换，转换的规则是：把赋值运算符右侧表达式的类型转换为左侧变量的类型，如表 28-6 所示。

关于这个表格的进一步解释如下。

- 1) 这个表格把 `int` 类型当作 32 位，所以没有把 `long` 类型列出来。这个表格也没有 `long long` 类型，可以参照上面推导出来。
- 2) 当赋值的右侧和左侧都是同样长度的整型变量时，原值照赋，内部的存储方式不变，但外部值却可能改变，也就是数值的解释发生变化。
- 3) 当赋值的右侧和左侧都是不同长度的整型变量时，如果左侧的长度小于右侧的长度，就做截断处理，高位舍弃。

表 28-6 赋值中的类型转换

源类型	目标类型	转换	说明
char	char	原值照赋	不管 signed 还是 unsigned
short	short	原值照赋	不管 signed 还是 unsigned
int	int	原值照赋	不管 signed 还是 unsigned
short	char	只保留低 8 位	不管 signed 还是 unsigned, 丢失高 8 位
int	char	只保留低 8 位	不管 signed 还是 unsigned, 丢失高 24 位
int	short	只保留低 16 位	不管 signed 还是 unsigned, 丢失高 16 位
signed char	signed short	转换时做 sign_extend	
signed char	unsigned short	转换时做 sign_extend	
unsigned char	signed short	转换时做 zero_extend	
unsigned char	unsigned short	转换时做 zero_extend	
signed char	signed int	转换时做 sign_extend	
signed char	unsigned int	转换时做 sign_extend	
unsigned char	signed int	转换时做 zero_extend	
unsigned char	unsigned int	转换时做 zero_extend	
signed short	signed int	转换时做 sign_extend	
signed short	unsigned int	转换时做 sign_extend	
unsigned short	signed int	转换时做 zero_extend	
unsigned short	unsigned int	转换时做 zero_extend	
int	float, double	改为浮点数形式	数值保持不变
float, double	int	只保留整数部分	舍弃掉浮点数的小数部分, 或者超出表示范围
float	double	改为另一种浮点数形式	
double	float	改为另一种浮点数形式	精度会降低, 或者超出表示范围

4) 当赋值的右侧和左侧都是不同长度的整型变量时, 如果左侧的长度大于右侧的长度, 就要根据右侧是 signed 类型还是 unsigned 类型, 做相应的 sign_extend 或者 zero_extend, 这样可以保持右侧的正负。注意: 有的编译器可能把右侧始终当做正数处理, 只做 zero_extend, 所以要特别注意。

5) 当赋值的右侧和左侧是在 int 类型和 float/double 类型之间进行, 就做相应的转换。

6) 当赋值的右侧和左侧是在 float 类型和 double 类型之间进行, 就做相应的转换。

当 sign_extend 和 zero_extend 发生在从短整型向长整型的赋值时, 扩展的长度由左侧的长度决定, 扩展的符号由右侧的符号位决定, 如果右侧是 signed, 就做 sign_extend, 否则就做 zero_extend。

sign_extend 需要把短整型的符号位填充到长整型高位上, 以保持符号位不变, 例如下面的 si0=sc0, si0 的高 24 位都要填充为 sc0 的第 7 位; zero_extend 需要用 0 填充到长整型高位上, 以保持还是无符号数, 例如下面的 ui0=uc0, ui0 的高 24 位都要填充为 0。

例子: char 到 int 的赋值转换

```
#include <stdio.h>
int main (void)
```



```
{
    signed char    sc0 = -1,    sc1 = 1;
    unsigned char uc0 = 0xFF,  uc1 = 1;
    signed int     si0, si1;
    unsigned int   ui0, ui1;
    si0 = sc0; si1 = sc1; printf ("%08x %08x\n", si0, si1);
    ui0 = sc0; ui1 = sc1; printf ("%08x %08x\n", ui0, ui1);
    si0 = uc0; si1 = sc1; printf ("%08x %08x\n", si0, si1);
    ui0 = uc0; ui1 = sc1; printf ("%08x %08x\n", ui0, ui1);
    return 0;
}
```

运行结果

```
FFFFFFFF 00000001
FFFFFFFF 00000001
000000FF 00000001
000000FF 00000001
```

C 语言赋值时的类型转换可能会让人感到不精密和不严格，因为不管表达式的值怎样，系统都会自动把它转为赋值运算符左部变量的类型。转变后数据可能有所不同，不加注意就可能带来错误。这确实是个缺点，也遭到许多人批评。但不应忘记的是：C 语言最初是为了替代汇编语言而设计的，所以类型转换比较随意。

用强制类型转换是一个好习惯，这样至少从程序上可以看出想干什么。另外，写代码时要做到“心里有数”，不去使用太多的数据类型，同时知道可能会发生的类型转换。

数组是相同数据类型的元素按一定顺序排列的集合，就是把有限个类型相同的变量用一个名字命名，然后用编号区分它们的变量的集合。组成数组的各个变量称为元素，元素的下标是从 0 开始计数的。数组存放在连续内存中，最低地址对应首元素，最高地址对应末元素。数组可以是一维的，也可以是多维的。

数组和指针联系非常紧密，下一章探讨指针。

29.1 数组的说明

数组的维数可以是多维的。

例子：下面依次是是一维、二维、三维数组

```
int a[10];
int data[3][4];
double param[100][4][10];
```

数组元素的索引是从 0 开始的，例如对于一维数组 `a[10]`，它的元素就为 `a[0]~a[9]`。

数组元素在内存中是按顺序依次存放的，先低后高，先低维后高维。

例子：`int data[3][4]` 的元素在内存中的存放顺序由低到高

```
data[0][0], data[0][1], data[0][2], data[0][3],
data[1][0], data[1][1], data[1][2], data[1][3],
data[2][0], data[2][1], data[2][2], data[2][3]
```

二维数组可以看作由一维数组的嵌套而构成的，因为如果一维数组的每个元素又都是 1 个一维数组，那么就组成了 1 个二维数组。

根据这样的分析，1 个二维数组也可以分解为多个一维数组。C 语言允许这种分解，例如有二维数组 `x[3][4]`，就可分解为 3 个一维数组，其数组名分别为 `x[0]`、`x[1]`、`x[2]`，对这 3 个一维数组不须另作说明即可使用。对于这 3 个一维数组，每个都有 4 个元素，例如：一维数组 `x[0]` 的元素为 `x[0][0]`、`x[0][1]`、`x[0][2]`、`x[0][3]`。

基于同样的道理，多维数组可以看作由一维数组的多级嵌套而构成的。

29.2 初始化

例子：一维数组的初始化

```
int a[10]; //数组没有初值
int a[10] = {1, 2, 3, 4, 5}; //只初始化前 5 个
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; //全部初始化，维数没有省略
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; //全部初始化，维数可以省略
```

二维数组的初始化可以按行分段赋值，也可按行连续赋值。

例子：二维数组的初始化

```
//按行分段赋值可写为
int b[5][3]= {{80, 75, 92},
              {61, 65, 71},
              {59, 63, 70},
              {85, 87, 90},
              {76, 77, 85}};

//按行连续赋值可写为
int b[5][3] = {80, 75, 92,
              61, 65, 71,
              59, 63, 70,
              85, 87, 90,
              76, 77, 85};

//对于多维数组，如果初始化所有的元素，那么最高维可以省略
int b[][3]= {{80, 75, 92},
            {61, 65, 71},
            {59, 63, 70},
            {85, 87, 90},
            {76, 77, 85}};

//如果只初始化一部分，那么其他元素会被初始化为 0
int b[5][3]= {{80, 75, 92},
            {61, 65, 71}};

//b[0][2], b[1][2], b[2][2], b[3][2], b[4][2]这 5 个数没有初值
int b[5][3]= {{80, 75},
            {61, 65},
            {59, 63},
            {85, 87},
            {76, 77}};
```

例子：C99 允许对指定的数组元素赋初值，这种方法对于稀疏数组很有效

```
int a[10] = {[0]=1, [6]=7, [9]=10}; //只初始化 0、6、9 这三个元素
```

29.3 字符串

字符串 (string) 为一维字符型数组，并以空字符 ('\0') 结尾。字符串应用很广泛，还有专门的一组函数用来操作字符串，函数原型在 `string.h` 中。

对于字符数组的初始化，C 语言允许用字符串常量对数组进行初始化赋值。如果用字符串常量赋值，那么编译器会在结尾自动添加 '\0'。因为自动添加 '\0'，所以可不用指定数组的长度，而由编译器自行处理。注意：如果用单个字符初始化字符串，必须以 '\0' 结尾。

例子：下面这三种对字符串初始化的方式是等价的

```
char str[]={ 'C', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0' };
char str[]={ "C program" }; //注意：结尾自动添加 '\0'
char str[]="C program"; //注意：结尾自动添加 '\0'
```

注意：`char str1[]="C program";`和 `char *str2="C program";`是不一样的。

`str1` 是一个 10 个字符的数组，它的内容是可以修改的。

`str2` 是一个指向一个字符串的指针，这个字符串在只读区域，是不能修改的。

`str1` 和 `str2` 可以认为没有什么区别，但是不能往 `str2` 写东西。

```
strcpy (str1, "abc"); //是可以的。
```

```
strcpy (str2, "abc"); //是错误的。
```

字符串数组就是二维字符型数组，这个在实际中也常用。只用高维作为索引的时候，就当作字符串使用。

例子：字符串数组

```
char str_ar[20][80];
for (i = 0; i < 20; i++)
    fgets (str_ar[i], 80, stdin);
```

字符串常量数组是一维数组，每个元素是字符串常量。注意：这些常量存放在只读空间。

例子：字符串常量数组

```
char *file_cmd[] = {"open", "close", "save", "save as", "exit"};
char cmd[20];
gets (cmd);
for (i = 0; i < sizeof (file_cmd) / sizeof (file_cmd[0]); i++)
    if (strncmd (cmd, file_cmd[i]) == 0) break;
```

29.4 复合常量赋值

C99 中允许使用复合常量赋值，它可以是指定对象类型的数组、结构或联合。使用复合常量赋值时，在括号内指定类型名，后面跟着包在“{ }”的初始化列表；当类型名是数组时，一定不要指定数组的最高维。

```
double *fp = (double[]){1.0, 2.0, 3.0};
```

这里建立一个 double 指针 fp，它指向 1 个 double 类型的一维数组，里面有 3 个 double 常数。

在函数外建立的复合赋值存在于程序的整个生命周期，在函数内建立的复合赋值是一个局部对象，随着函数的结束而被破坏。

29.5 函数中的变长数组

在 C99 标准出现之前，声明数组时在“[]”中只能使用整数常量表达式，而 C99 做了很大增强，允许数组“[]”中的值是整型变量或是整型表达式，这就是变长数组 (Variable Length Array, VLA)。

例子：C99 支持的变长数组，n 是在运行时动态确定的变量

```
int n;
scanf ("%d", &n);
int array[n];
```

注意：变长数组不是说数组的长度会随时变化，而是说用整型变量或表达式定义的数组，在变长数组的生存期内它的长度同样是固定的。

通常来说，sizeof 通常返回的是整型常量，其值等于数据类型或数据对象的字节数，但是当 sizeof 用于变长数组时，就只能在运行时求值。这种改变是必须的，因为变长数组的大小只有运行时才知道。

例子：对变长数组使用 sizeof

```
#include <stdio.h>
void fun (int n)
```

```

{
    int vla[n];
    printf ("vla takes %d bytes\n", sizeof (vla));
}
int main (int argc, char *argv[])
{
    fun (10);
    fun (20);
    return 0;
}

```

运行结果
vla takes 40 bytes
vla takes 80 bytes

例子：变长二维数组

```

void f (int dim1, int dim2)
{
    int matrix [dim1][dim2];
    .....
}

```

29.6 结构中的灵活数组

C99 中，当结构的成员个数大于 1 时，就允许结构中最后的成员是一个未知大小的数组，这个最后的成员就称为灵活数组。它允许结构中包含一个可变大小的数组，但是由 `sizeof` 返回这种结构的大小不包括这个灵活数组。

如果使用灵活数组成员的结构，那么就应该使用 `malloc()` 进行动态分配，并且根据需要确定灵活数组的大小。

例子：这里为灵活数组 `energy` 分配了 20 个 `int` 类型的空间

```

struct mystruct {
    int id;
    char name[20];
    int energy[]; //flexible array
};
struct mystruct *p;
p = malloc (sizeof(struct mystruct) + 20 * sizeof(int));
for (i = 0; i < 20; i++) p->energy[i] = i;

```

如果不使用灵活数组，就要按下面的常规方式处理。

例子：常规方式处理，需要使用两次 `malloc()`。

```

struct mystruct {
    int id;
    char name[20];
    int *energy;
};
struct mystruct *p;
p = malloc (sizeof(struct mystruct));
p->energy = malloc (20 * sizeof(int));
for (i = 0; i < 20; i++) p->energy[i] = i;

```

29.7 数组作为函数参数

当把数组作为函数的参数时，可以写成几种等价的方式。

对于一维数组，可以写维数，也可以不写维数，也可以写成指针形式。

```
例子：使用一维数组作为参数
void func1 (int ar[10]);
void func1 (int ar[]);
void func1 (int *ar);
```

对于多维数组，可以写所有的维数，也可以不写最高维的维数，也可以写成数组指针的形式。

```
例子：使用多维数组作为参数
void func2 (int ar[10][15]);
void func2 (int ar[][15]);
void func2 (int (*ar)[15]);
void func3 (int ar[10][15][20]);
void func3 (int ar[][15][20]);
void func3 (int (*ar)[15][20]);
```

注意：(*ar)的括号是必需的，去掉括号就具有另外的含义，就会导致编译错误。

有人会问为什么多维数组最高维可以省略不写，其他维必须要写呢？这是因为当用下标引用数组元素的时候，要用数组的维数计算元素在内存中的实际位置，在计算时最高维实际没有用上，只有其他维参与了运算。例如，对于 `int ar[10][15][20]`，现在要操作 `ar[7][8][9] = 1997`，那么 `ar[7][8][9]` 相对于 `ar[0][0][0]` 的偏移量为 $(7 * 15 * 20 + 8 * 20 + 9) * 4$ ，乘以 4 是因为每个 `int` 类型占 4 个字节。

29.8 数组和指针

数组和指针可以认为是相同的：一是在函数声明，传递数组和指针是相同的，例如，在上一节的“数组作为函数参数”中，使用数组和指针都是可以的，因为传递数组实际上传递的是数组首元素的地址；二是在访问它们的元素时，既可以使用 “[]”，也可以使用 “*” 的方式，因为数组名可以认为是一个指针常量值，例如，对于 “`int a[10]; int *p=a;`”，访问第三个元素，可以用多种方式：`a[3]`、`p[3]`、`*(a+3)`和`*(p+3)`。

在其他地方，就不要认为数组和指针是相同的。例如，在一个文件中定义了一个全局变量 `int value[100]`，在另一个文件中要使用它，那么必须使用 `extern int value[100]`（也可以使用 `extern int value[]`，但是不能对它使用 `sizeof`）；但是，如果使用 `extern int *value`，编译可以通过，但运行就完全错误。

为什么错误呢？对于 `extern int value[100]` 和 `extern int *value`，虽然访问 `value` 中的元素是一样的，都可以使用 `value[x]`，但这只是表面上的，生成的汇编代码是完全不一样的。我们可以看下面的例子，比较一下对应汇编代码的区别。

```
例子：数组和指针的区别
extern int datx[100];
extern int *daty;
int get_datx (int i) { return datx[i]; }
int get_daty (int i) { return daty[i]; }

使用 gcc -O2 -S -o a39.s a39.c
_get_datx:
    movl    4(%esp), %eax          //eax 对应参数 i
    //这里直接使用 datx 的值(就是数组首元素的地址)，计算要读取元素的地址
```

```
    movl    _datx(,%eax,4), %eax
    ret
_get_daty:
    movl    _daty, %eax           //这里要从变量 daty 中取出指针本身的值
    movl    4(%esp), %edx        //edx 对应参数 i
    movl    (%eax,%edx,4), %eax  //然后计算要读取元素的地址
    ret
```

这是因为 `daty` 本身始终位于同一个地址，但它的内容可以是不同的值，可以指向不同地址的 `int` 型变量；而 `datx` 是一个数组的地址，而且是不能改变的，总是表示 100 个存放 `int` 的内存空间，里面的内容可以不同。

第 30 章

指针

指针是存放内存地址的变量，正确理解并使用指针是非常重要的，其原因有四个：第一，指针为函数提供修改调用参数的手段；第二，指针可以改善函数的执行效率，通过指针可以编写出更高效紧凑的程序；第三，通过指针使用 `malloc()`和 `free()`实现动态分配内存；第四，指针为动态数据结构提供支持，例如二叉树和链表。

指针是 C 语言中最强的特性之一，也是最危险的特性之一。例如，含有无效值的指针可以使系统瘫痪，不正确的使用指针容易引入难以排除的错误。

30.1 指针的说明

例子：指针和变量

```
int x = 5;           //定义一个整数变量 x
int *px;            //定义一个整数指针 px
px = &x;            //把整数 x 的地址赋值给 px
(*px)++;           //px 指向的整数变量加 1，也可以使用 px[0]++;
printf ("x=%d\n", x); //现在 x 的值是 6
```

例子：函数调用，通过指针修改变量

```
#include <stdio.h>
void inc (int *val)
{
    (*val)++;
}
int main (void)
{
    int a = 3;
    inc (&a);
    printf ("%d" , a); //现在 a 的值是 4
    return 0;
}
```

例子：指针和结构

```
struct EMPLOYEE employee = {100, "Steve Jobs", 0, 3456};
struct EMPLOYEE *p;
p = &employee;
printf ("id %d name%s sex %s salary %d\n",
        p->id, p->name, p->sex ? "female" : "male", p->salary);
```

指针可以进行的算术操作只有加法和减法，指针+1 后指向下一个元素，指针-1 后指向上一个元素。指针还可以进行比较操作。

例子：指针的算术和比较操作

```
char *p, *q;
```



```
p++;    q--;
p += 5; q -= 7;
if (p > q) printf ("p is larger than q!\n");
```

30.2 啰嗦的指针

指针和数组是密切相关的，不带下标的一维数组的名字就是指向数组第一个元素的地址，也就是一个常量指针，所以访问数组和指针的元素有两种方式，既可以使用“*”的形式（数组索引），也可以使用“[]”的形式（指针指向）。

从代码的可读性和方便性来讲，使用“[]”的方式更加方便，因为使用“*”的方式显得啰嗦。但是，还是有很多人继续使用“*”的方式，他们只是觉得因为声明为指针，就得按照“*”的方式访问，有些顽固了。

例子：对比“[]”和“*”的访问方式，一维数组

```
//定义一个整数数组
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p;           //定义一个整数指针 p
//下面两句等价
p = a;           //方便
p = &a[0];       //啰嗦
//下面两句等价
p = a + 5;       //方便
p = &a[5];       //啰嗦

//下面三句等价
p = a;
for (i = 0; i < 10; i++) p[i]++;           //方便
for (i = 0; i < 10; i++) (*(p+i))++;       //啰嗦
for (p = &a[0]; p < &a[10]; p++) (*p)++;   //啰嗦

//四种访问数组元素的方式
p = a;
for (i = 0; i < 10; i++) printf ("%d\n", a[i]);           //方便
for (i = 0; i < 10; i++) printf ("%d\n", *(a+i));       //啰嗦
for (i = 0; i < 10; i++) printf ("%d\n", p[i]);         //方便
for (i = 0; i < 10; i++) printf ("%d\n", *(p+i));       //啰嗦
```

例子：对比“[]”和“*”的访问方式，二维数组

```
//定义一个整数数组
int a[10][20];
//几种引用数组元素的方式
for (i = 0; i < 10; i++)
    for (j = 0; j < 20; j++)
        printf ("%d\n", a[i][j]);           //方便
for (i = 0; i < 10; i++)
    for (j = 0; j < 20; j++)
        printf ("%d\n", (*(a + i) + j));     //啰嗦
for (i = 0; i < 10; i++)
    for (j = 0; j < 20; j++) {
        int *p= a[i];
        printf ("%d\n", *(p + j));         //啰嗦
    }
for (i = 0; i < 10; i++)
```

```
for (j = 0; j < 20; j++)
    printf ("%d\n", *(a[i] + j));           //啰嗦
```

当指针指向的元素是 `struct` 或 `union` 类型，如果要引用内部成员时，应该使用“->”，这是众所周知的，而没有使用`(*p).name` 这样的形式（这里括号是必须的，因为“*”比“.”的优先级低）。

例子：简写方式

```
struct EMPLOYEE *p = employee;
printf ("%d%s\n", p->id, p->name);        //方便
printf ("%d%s\n", (*p).id, (*p).name);   //啰嗦
```

30.3 void *

如果指针 `p1` 和 `p2` 的类型相同，那么 `p1` 和 `p2` 之间可以相互赋值；如果 `p1` 和 `p2` 指向不同的数据类型，那么对于 `p1` 和 `p2` 之间的相互赋值，必须要使用强制类型转换，否则就会报告警告。

例子：`float *p1; int *p2; p1 = p2;`

编译会有警告报出“assignment from incompatible pointer type”，所以必须改为

```
p1=(float *)p2;
```

注意：对于不同类型指针之间的强制转换，必须要知道转换后的运算行为，不要随意转换。

`void *`则不同，任何类型的指针都可以直接赋值给它，它也可以直接赋值给任何类型的指针，无须进行强制类型转换。`void`的字面意思是“无类型”，`void *`则为“无类型指针”，`void *`可以指向任何类型的数据。

```
例子：void *p1; int *p2; p1 = p2;
例子：void *p1; int *p2; p2 = p1;
例子：int *p2 = NULL; //Because #define NULL ((void *)0)
```

这就是内存分配和释放函数为什么要使用 `void *`的原因，因为可以减少强制转换。

```
void *malloc (size_t num_bytes);
void free (void *ptr);
```

例子：使用 `malloc` 和 `free`

```
#include <string.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
int main(void)
{
    char *str;
    str = malloc (10);
    if(str == NULL){
        perror ("malloc");
        exit (1);
    }
    strcpy (str, "Hello");
    printf ("String is %s\n", str);
    free (str);
    return 0;
}
```

注意：在 ANSI C 标准中，不允许对 void 指针进行算术运算，例如 pvoid++或 pvoid+=1 等就是禁止的，但是在 GNU 中是允许的，因为在缺省情况下，GNU 认为 void *与 char *一样，所以 sizeof(*pvoid) == sizeof(char)。

```
void *ptr1, *ptr2;   ptrdiff_t diff;
ptr2 = ptr1 + 5;
diff = ptr2 - ptr1;
*(size_t*)(ptr1 - sizeof (size_t)) = 100;
```

void*指针可以用作泛型，假设你要交换两个变量，如果是交换两个 int 类型变量，就写成如下。

```
void swap_int (int *a1, int *a2)
{
    int tmp = *a1;
    *a1 = *a2;
    *a2 = tmp;
}
```

如果要交换两个 double 类型变量，还要写一个

```
void swap_double (double *a1, double *a2)
```

为了达到通用的效果，就可以使用 void *，但是因为不能从 void *获得交换大小的信息，就添加一个额外的参数 size。

```
void swap (void *a1, void *a2, size_t size)
```

有了这样一个函数，想要交换两个变量，就可以：

```
swap (&ai, &bi, sizeof(int));
swap (&af, &bf, sizeof(double));
```

30.4 restrict

restrict 是 C99 标准引入的关键字，它只可以用于限定和约束指针，表明指针是访问数据对象的唯一且初始的方式。restrict 告诉编译器，对于指针所指向内存中内容的所有修改，都必须通过该指针来修改，而不能通过其他变量或指针来修改。这样做的好处是，能帮助编译器进行更好的优化代码，生成更有效率的汇编代码。例如，int *restrict ptr，这里 ptr 指向的内存单元只能被 ptr 访问，不能通过其他指针访问。通过使用 restrict，编译器可以更好地优化你的代码。

在下面的例子中，restar 指向由 malloc()分配的内存，restar 也是访问这块内存的唯一且初始的方式，par 就不是。使用了关键字 restrict，编译器就可以放心地进行优化了。

例子：使用 restrict 的好处

```
int ar[10];
int *restrict restar=(int *)malloc (10 * sizeof(int));
int *par=ar;
for (n = 0; n < 10; n++)
{
    par[n] += 5;
    restar[n] += 5;
    ar[n] *= 2;
    par[n] += 3;
    restar[n] += 3;
}
```

因为 `restar` 是访问这块分配内存的唯一且初始的方式，那么编译器可以将上面对 `restar` 的操作优化为：`restar[n]+=8`。

但是 `par` 并不是访问数组 `ar` 的唯一方式，因此不能优化成这样：`par[n]+=8`，因为在 `par[n]+=3` 前，就有 `ar[n]*=2`。

C 语言库函数中有两个函数用于把内存中的内容从一个位置复制到另一个位置。在 C99 标准下，它们的原型如下。

```
void *memcpy (void * restrict dst, const void * restrict src, size_t n);
void *memmove (void *dst, const void *src, size_t n);
```

这两个函数都从 `src` 复制 `n` 字节的数据到 `dst`，而且都返回 `dst`。两者之间的差别是由关键字 `restrict` 造成，`memcpy` 可以认为 `dst` 和 `src` 没有重叠；`memmove` 就不能这样认为，它必须考虑 `dst` 和 `src` 的重叠情况。如果 `dst` 和 `src` 的区域存在重叠，例如 `dst` 的头部和 `src` 的尾部重叠，那么 `memcpy` 就会错误地完成操作，但是 `memmove` 可以正常完成操作。所以程序员在使用 `memcpy` 时，必须确保没有重叠的情况，否则就要使用 `memmove`。

`restrict` 告诉编译器可以自由地做一些有关优化的假定，另一方面，`restrict` 告诉程序员在使用库函数的时候，必须要使用满足 `restrict` 要求的参数。

30.5 多级指针

`int **p2` 可以理解为 `int *(*p2)`，是指向指针的指针，这里是指向整数指针的指针，访问时 `(*p2)` 的内容是指向整数的指针，`(**p2)` 的内容是一个整数。

```
例子：多级指针
main (void)
{
    int x = 10;
    int *p = &x;
    int **p2 = &p;
    printf ("%d %d %d\n", x, *p, **p2); //打印的都是 x 的值
    printf ("%p %p %p\n", &x, p, *p2); //打印的都是 x 的地址
    printf ("%p %p\n", &p, p2); //打印的都是 p 的地址
    return 0;
}
```

我们知道指针和一维数组可以认为是等价的，例如

```
int a[10];
int *pa = a; //pa 和 a 都是指向同一块内存，操作 pa[i] 和 a[i] 是一样的
```

但是千万不能得出这样的结论：二维数组和二级指针是等价的，三维数组和三级指针是等价的。因为这样的结论都是错的，例如

```
int a2[6][10];
int **p2 = a2; //This is ERROR
int a3[3][6][10];
int ***p3 = a3; //This is ERROR
```

首先是数据类型不匹配，其次即使你做了强制转换让编译通过了，实际运行操作 `p2` 和 `p3` 也不对。因为在访问 `p2[i][j]` 时，实际是先从 `p2[i]` 取出一个值 `x`，这个 `x` 的类型是 “`int *`”，然后再访问 `x[j]`。

笔者不知道别人有何想法，笔者之前就以为 p2 和 a2、p3 和 a3 是等价的。那么如何在多维数组和指针之间建立关系呢？可以通过下面介绍的数组指针或者指针数组。

30.6 数组指针和指针数组

数组指针和指针数组的含义非常容易混淆，一是因为中文的缩写导致含义有些不清楚，二是因为它们的定义区别就在于有没有括号的问题上，例如对于“char *p[10]”和“char (*p)[10]”，这两个容易分清楚吗？

在区分这两个概念时，注意结尾的两个字是关键。指针数组强调的是数组，而数组指针强调的是指针。

30.6.1 指针数组

指针数组的英文意思是“The array which is composed of pointers”，就是由指针构成的数组，数组所有的元素都是指针，例如 char *p[10]。

指针数组不仅可以是一维数组，也可以是多维数组，形式如下：

```
type *pointer_array[c1][c2]...[cn];
```

注意：不能在*pointer_array上加括号，就是要求“[]”比“*”的优先级高，数组元素的类型是 type*。

注意：不能将二维数组的数组名赋给指针数组，因为两者的类型不一致，例如

```
int a2[6][10];
int *px[6];
px = a2; //This is ERROR
```

指针数组最常用的场合就是说明一个字符串数组，即说明一个数组，它的每个元素都是一个字符串。其实 main()正规写法如下，只不过如果不用 argv 和 env，就可以写成 main(void)。

```
int main (int argc, char *argv[], char *env[]);
int main (int argc, char **argv, char **env);
```

这里 argv 和 env 就是指针数组或者二级指针，因为一维的指针数组和二级指针是相匹配的。

30.6.2 数组指针

数组指针的英文意思是“The pointer which points to one array”，就是指向数组的指针，指针指向的是数组的首地址，例如 char (*p)[10]。

数组指针不仅可以指向一维数组，也可以指向多维数组，形式如下

```
type (*pointer_array)[constant1][constant2]...[constantn];
```

注意：必须在*pointer_array上加括号，就是要求“*”比“[]”的优先级高。你的感觉可能有点别扭，但是你想一下，整数指针指向的是整数，结构指针指向的是结构，那么“数组指针指向的是数组”是不是就清楚了呢？

注意：可以将多维数组的数组名赋给数组指针，因为二者类型是相匹配的，但是要保证数组指针中的维数和多维数组中的维数匹配。

```
int a2[6][10];
int (*py)[10];
```

```

py = a2; //This is CORRECT
进一步
int a3[5][6][10];
int (*pz)[6][10];
pz = a3; //This is CORRECT

```

数组指针可以直接初始化为一个常量数组。

```

int (*ap)[2] = {{1, 2}, {3, 4}, {5, 6}};
相当于
int ar[3][2] = {{1, 2}, {3, 4}, {5, 6}};
int (*ap)[2] = ar;

```

还可以对数组指针分配内存，下面就相当于分配了一个 `int [10][15]` 大小的数组。

```

int (*p)[15];
p = malloc (15 * sizeof(int) * 10);

```

`char *p[10]` 和 `char (*p)[10]` 竟然有这么大区别，这是由操作符的优先级决定的。在 `char *p[10]` 中 “[]” 比 “*” 优先级高，所以 `p` 是具有 10 个元素的数组，数组元素是指针 (`char *`)；在 `char (*p)[10]` 中，因为 “()” 的原因 “*” 比 “[]” 优先级高，所以 `p` 是指针，`p` 指向的是数组 (`char [10]`)。

30.6.3 与二维数组建立联系

对于一个二维数组，例如 `int a2[6][10]`，可以有两种方法与它建立联系，一个是通过指针数组，另一个是通过数组指针。

例子：使用指针数组与数组建立联系。

```

int a2[6][10];
int *px[6] = {a2[0], a2[1], a2[2], a2[3], a2[4], a2[5]};
int **p2 = px;

```

`a2` 是二维数组，`a2[0] ~ a2[5]` 就是一维数组；`px` 是指针数组，它的元素类型是 `int *`，正好与 `a2[0] ~ a2[5]` 匹配，所以可以把 `a2[0] ~ a2[5]` 赋值给 `px` 的元素。进一步，因为 `px` 是指针数组，`p2` 是二级指针，它们之间是匹配的，所以可以有 `p2=px`。

这样就可以按照 `p2[i][j]` 或者 `px[i][j]` 访问 `a2` 的元素了。

例子：使用数组指针与数组建立联系

```

int a2[6][10];
int (*py)[10];
py = a2;

```

如何解读 `int (*py)[10]` 呢？首先 `py` 是个指针，指向的内容是 `int [10]`（就是有 10 个整数的一维数组）。`a2` 是二维数组，但是也可以认为 `a2` 是数组的数组（`a2` 是一维数组，它的元素是有 10 个整数的一维数组，就是 `int [10]`）。所以，`py` 和 `a2` 可以相匹配的，可以直接赋值 `py = a2`，这时 `py[0]` 是 `a2[0]`，`py[1]` 是 `a2[1]`。

这样就可以按照 `py[i][j]` 访问 `a2` 了。

对比一下，`px[i][j]` 和 `py[i][j]`，表面上是一样的，但是 `px` 和 `py` 是不同的类型，其实访问机制是不一样的，编译生成出来的汇编代码也是不一样的。

30.7 函数指针和指针函数

函数指针和指针函数的含义非常容易混淆，一是因为中文的缩写导致含义有些不清楚，二是因为它们的定义区别就在于有没有括号的问题上，例如对于“char *f(void)”和“char (*f)(void)”，这两个容易分清楚吗？

在区分这两个概念时，注意结尾的两个字是关键。函数指针强调的是指针，而指针函数强调的是函数。

30.7.1 指针函数

指针函数的英文意思是“The function which returns one pointer”，就是指返回值是指针的函数，即本质上就是一个函数。我们知道函数返回值都有类型（若无返回值，则为 void），只不过指针函数返回值的类型是某种类型的指针。

我们常用的 malloc() 就是指针函数，它的原型如下。

```
void *malloc (size_t num_bytes);
```

30.7.2 函数指针

函数指针的英文意思是“The pointer which points to one function”，就是指向函数的指针，所以函数指针就是指针，只不过该指针指向的是函数。正如用指针变量可以指向整型、字符型、数组一样，这里是指向函数的开始地址。通过指针变量可以访问其他类型变量，同样通过函数指针可以调用函数。

函数指针的声明方法为：

```
return_type (*func_pointer) ([function parameter]);
```

例如：使用函数指针

```
int func (int x);           //声明一个函数
int (*fp)(int x);         //定义一个函数指针
fp = func;                 //把 func 函数的首地址赋给指针 fp
(*fp)(456);               //通过函数指针调用 func。
```

使用函数指针的说明如下。

- 1) 在定义函数指针时，它的形参和返回值必须和实际函数的形参和返回值一样，否则编译出错。
- 2) (*fp)的括号不能省，括号改变了运算符的优先级。如果没有括号，就变成是在声明一个函数原型，返回值是一个指针。
- 3) 把函数赋值给函数指针时，函数 func 不能带括号，更不能带参数。赋值以后，指针 fp 就指向函数 func 的首地址。这里使用“fp = &func;”也是可以的。
- 4) 通过 fp 调用函数，也可以使用“fp(456);”，但是前提是必须要有“int (*fp)(int x);”声明，因为编译器可以识别出这是在通过函数指针调用函数。如果没有“int (*fp)(int x);”，比如说忘记包含必要的头文件，但是调用函数使用“fp(456);”，这时编译可能会有警告，但是执行会完全错误，笔者遭遇过这样的错误。所以最好还是使用“(*fp)(456);”。

例子：使用函数指针

```
#include <stdio.h>
int max_val (int x, int y)
{
    return x > y ? x : y;
}
int main (void)
{
    int (*ptr) (int, int);
    int a, b, c;
    ptr = max_val;
    scanf ("%d %d", &a, &b);
    c = (*ptr)(a,b);
    printf ("a=%d, b=%d, max=%d", a, b, c);
    return 0;
}
```

30.7.3 使用 typedef 定义函数指针

通过 typedef 定义函数指针类型，可以简化代码编写，而且便于阅读。

例子：使用 typedef 定义函数指针类型

```
#include<stdio.h>
typedef void (*FUNC_P)(void);
void TriangleFunc (void) { printf ("Triangle Func\n"); }
void RectangleFunc (void) { printf ("Rectangle Func\n"); }
void CircleFunc (void) { printf ("Circle Func\n"); }
int main (void)
{
    FUNC_P fp;
    fp = TriangleFunc;
    (*fp)();
    fp = RectangleFunc;
    (*fp)();
    fp = CircleFunc;
    (*fp)();
    return 0;
}
```

下一步可以看一下函数指针数组，顾名思义，它是一个数组，数组的元素都是函数指针。

例子；fp[]是一个有3个元素的函数指针数组

```
#include<stdio.h>
typedef void(*FUNC_P)(void);
void TriangleFunc (void) { printf ("Triangle Func\n"); }
void RectangleFunc (void) { printf ("Rectangle Func\n"); }
void CircleFunc (void) { printf ("Circle Func\n"); }
FUNC_P fp[] = {TriangleFunc, RectangleFunc, CircleFunc};
int main (void)
{
    int option;
    while (1) {
        scanf ("%d", &option);
        printf ("option is %d\n", option);
        if (option < 0 || option >= (sizeof(fp) / sizeof(fp[0])))
            continue;
        (*fp[option])();
        fflush (0);
    }
}
```


30.7.4 signal()

最有名的使用函数指针和 typedef 的例子是 signal(): 第一, signal() 的返回值和第二个参数都是函数指针; 第二, 如果不使用 typedef, signal() 的原型看起来非常复杂。

```
void (*signal(int signum, void (*handler))(int))(int);
```

但是使用 typedef 之后, 是不是一看就明白了? 这里 sighandler_t 定义成函数指针类型, 要求函数的参数是 int, 函数的返回值为 void。

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

signal() 的第一个参数是指定的信号值。第二个参数是信号值的处理方式, 可以忽略该信号 (使用 SIG_IGN), 可以采用按系统默认方式处理该信号 (使用 SIG_DFL), 也可以使用自己定义的信号处理函数。信号处理函数的参数要求是信号值, 这样使得一个信号处理函数就可以处理多个不同的信号。

如果 signal() 调用成功, 返回最后一次设置的 handler 值; 如果失败, 则返回 SIG_ERR。

例子: signal() 使用的演示程序

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void sigroutine (int num)
{
    switch (num) {
        case SIGHUP:
            printf ("Get a signal -- SIGHUP ");
            break;
        case SIGINT:
            printf ("Get a signal -- SIGINT ");
            break;
        case SIGQUIT:
            printf ("Get a signal -- SIGQUIT ");
            break;
    }
}

main (void)
{
    printf ("process id is %d\n", getpid ());
    signal (SIGHUP, sigroutine);
    signal (SIGINT, sigroutine);
    signal (SIGQUIT, sigroutine);
    while (1);
}
```

其中信号 SIGINT 由按下 Ctrl-C 发出, 信号 SIGQUIT 由按下 Ctrl-\ 发出。

该程序执行的结果如下所示。

```
#!/sig_test
process id is 463
Get a signal -SIGINT
Get a signal -SIGQUIT
```

30.8 malloc

全局变量和静态局部变量在编译时就分配好了，而局部变量（不包含静态局部变量）使用堆栈空间，全局变量和局部变量在运行中都不能增减。然而，程序在运行中可能需要数量可变的内存，可能要使用复杂的数据结构（如链表或二叉树），这类结构本身是动态的，要随着需求的变化而变化。为了实现这样的数据结构，就要求程序能够根据需要分配和释放内存，动态分配内存就是程序在运行中管理内存的方法。

动态分配函数从堆（Heap）中取得内存，堆是系统的自由内存区，通常都有很大的内存空间。动态分配的核心函数是 `malloc()` 和 `free()`，其他几个函数不太常用。这些函数协同工作，在 Heap 中建立并维护一个可用内存表。

例子：`malloc()` 和 `free()`

```
char *p = malloc (50 * sizeof(int));
if (p == NULL) {
    fprintf (stderr, "Error: malloc\n");
    exit (1);
}
.....
.....
free (p);
```

注意：对 `malloc()` 分配的内存检查是否为 `NULL`，如果为 `NULL`，就做相应的错误处理。

注意：对 `malloc()` 分配的内存必须由 `free()` 释放，而且只能释放一次。

使用动态内存分配最容易出现的问题就是操作越界，不管是上边界，还是下边界，或者把程序的可用内存表破坏，或者把程序别的数据破坏掉，这些都会导致程序运行错误。

操作越界原因有很多，如下所示。

- 1) 例如，应该使用 `sizeof` 计算 `struct` 的大小，但是自己手工计算 `struct` 的大小，结果自己计算出来的值是错误的。
- 2) 例如，需要 50 个整数空间，应该写为 `malloc(50 * sizeof(int))`，但是写为 `malloc(50)`。
- 3) 例如，操作字符串时没有算上末尾 `\0`，导致错误。

```
str2 = malloc (strlen(str1)); strcpy (str2, str1); //错误
str2 = malloc (strlen(str1)+1); strcpy (str2, str1); //正确
```

- 4) 例如，程序需要检查 `malloc()` 的返回结果，因为内存分配失败时会返回 `NULL`，但是程序没有检查。

```
int *p = malloc (sizeof (int) * 50); //需要检查 p 是否为 NULL
for (int i = 0; i < 50; i++) p[i] = i;
```

- 5) 例如，访问越界，超过了上界。

```
int *p = malloc (sizeof (int) * 50);
for (int i = 0; i <= 50; i++) p[i] = i;
```

- 6) 例如，访问越界，超过了下界。

```
int *p = malloc (sizeof (int) * 50);
i = 49;
while (i >= 0) { --i; p[i] = i; }
```

30.9 alloca

`alloca()` 也是一个内存分配函数，与 `malloc` 类似，但是有一个重要的区别，`alloca` 是在栈 (stack) 上申请内存空间。函数返回时，通过 `alloca()` 申请的内存被自然释放掉，所以千万不能调用 `free()`。

`alloca()` 包含在头文件 `malloc.h` 中，实际定义如下。

```
#define alloca(x) __builtin_alloca((x))
```

这说明它是由编译器处理的，直接就生成合适的汇编代码，所以在生成的汇编代码中根本就不会找到对 `alloca()` 的调用。

```
例子：使用 alloca()
void process_data (int length)
{
    int *p = (int *)alloca(sizeof(int)*length);
    .....
    //此时千万不能调用 free(p)，否则会导致错误
}
```

因为函数返回时，通过 `alloca()` 申请的内存被自然释放掉，所以只能在函数内使用，不能把它像 `malloc()` 分配的内存那样长久使用。

因为 C99 支持变长数组 (VLA)，所以可以使用变长数组代替原来使用 `alloca()` 做的工作。

30.10 指针使用中的问题

如果使用指针不注意，就会带来很多问题，例如“`malloc`”节列出的访问越界问题，另外还有下面这些问题。

使用未初始化或没有分配内存的指针。

```
例子：这里 p 是一个任意值
int *pi;
*pi = 1234;
char *ps;
strcpy (ps, "Ni zai gan sha ne?");
```

本来需要指针的地方却没有使用变量的地址，而是直接使用变量本身。例如某些函数的参数需要传递的是变量地址，但是错误地把变量传递进去。

```
例子：该用地址的地方没用地址
int x;
scanf ("%d", x);
应该改为
scanf ("%d", &x);
```

返回非法指针。

```
例子：返回局部变量
char *copy_string (char *src)
{
    char buffer[100];
    strcpy (buffer, src);
```

```
    return buffer;  
}
```

内存已经释放，但是还继续使用。

```
例子：使用非法指针  
free (p);  
.....  
strcpy (p, "Hello");
```

内存使用完，没有释放，导致内存泄露

```
例子：内存泄露  
p = malloc (100);  
strcpy (p, "Hello");  
.....  
p = malloc (200);    //原来的 100 个字节并没有释放  
strcpy (p, ...);
```

31.1 if

例子：else 语句总是与最近的 if 语句匹配

```
if(cond1)
    if (cond2)
    else {
        .....
    }
```

这里 else 是和 if (cond2) 匹配，而不是和 if (cond1) 匹配，为了更加清晰全面一些，最好写成如下的样子。

例子：else 语句总是与最近的 if 语句匹配

```
if(cond1) {
    if (cond2)
    else {
        .....
    }
}
```

另外，如果 if/else 语句简单，且都是对同一个变量赋值，可以改用“?:”。

31.2 switch

switch 是多分支语句，可以写出更结构化的代码。switch 具有如下特性：

- 1) switch 只能测试相等，if 则能测试关系和逻辑表达式。
- 2) 各个 case 的常数必须不同，如果是字符常数，会自动提升为整数。
- 3) 可以有 default，当与各个 case 都不相等时，就会执行 default 处的语句。
- 4) 当变量和某个 case 比较相等时，就会执行相应的语句。
- 5) 注意要在每个 case 的结尾添加 break，以便跳出 switch。
- 6) 如果要在某个 case 的结尾继续执行它下面的语句，就不要添加 break，但是要心里有数。

当变量是 char 类型时，要注意常量的取值，例如下面的 0xFF 是不对的，要写成-1。

```
char sc;
switch (sc)
{
    case 'A':      break;
    case 0xFF:    break;
```

```

        //case -1:      break;
    }

```

31.3 for 和 while

for 和 while 循环格式为:

```

for (initialization; condition; increment)
    statement
while (condition)
    statement

```

在 for 循环内三项都规整的时候, 可以获得更好的表现效果。

例子: while 和 for 的对比

```

i = 0;
while (i < 100) {
    printf ("%d\n", i);
    i++;
}

```

```

for (i = 0; i < 100; i++)
    printf ("%d\n", i);

```

```

Node *node = list;
while (node != null) {
    print_data (node->data);
    node = node->next;
}

```

```

for (Node *node = list; node != null; node = node->next)
    print_data (node->data);

```

for (initialization; condition; increment)中的三项不是必需的, 任意一项没有都可以, 甚至这三项都没有也可以。例如, for(;;)实现的死循环, 它和 while(1)的效果是一样的。

从使用上讲, 如果使用 for 循环, 最好这三项都是全的。如果这三项不全, 这样的 for 循环看着怪怪的, 所以最好改用 while 循环, 因为 while 循环更自由一些。

有些人非常推崇紧凑的代码, 例如下面 while 中的判断条件处就把所有的事情做了。笔者感觉这样并不太好, 因为过分紧凑的代码可能会影响代码的可读性, 所以最好不要那么紧凑, 分开写更好一些。

例子: 分开写语句

```

while ((ch = getchar ()) != EOF && ch != '\n') ;
最好改为如下
while (1) {
    ch = getchar ();
    if (ch == EOF || ch == '\n')
        break;
}

```

31.4 do {...} while

这个循环的格式如下, 注意: while(condition)必须以分号为结尾。

```

do {

```

```
.....
} while (condition);
```

`do {...} while` 至少要执行一次循环体，有些人就认为这是一个很好的特性。但是从阅读效果上看，`do {...} while` 其实看着怪怪的，把 `do {...} while` 改成 `while` 循环可以获得更好的表现效果，所以最好少用 `do {...} while`。

当然，`do {...} while` 还有很大用处，用于定义包含代码的函数宏，这个在“预处理”章节介绍，就是在利用这个“至少要执行一次循环体”的特性。

31.5 break

`break` 语句用于跳出 `for/while` 循环或者用于终止 `switch` 的执行，但是曾经有个程序员错误地理解了 `break`，从而写出了错误的代码。

例子：正确的代码，使用多个 `if/else`

```
while (1) {
    int ch = getchar ();
    if (ch == 'A') {
        //.....
    }
    else if (ch == 'B') {
        if (COND1) continue;
        //.....
    }
    else if (ch == 'C') {
        if (COND2) break;
        //.....
    }
    else {
        //.....
    }
}
```

他觉得 `if` 语句看着不规整，就把 `if/else` 改为 `switch`，代码如下。

例子：错误的代码，使用 `switch`，错误理解 `break`

```
while (1) {
    int ch = getchar ();
    switch (ch) {
        case 'A':
            //.....
            break;
        case 'B':
            if (COND1) continue;
            //.....
            break;
        case 'C':
            if (COND2) break;
            //.....
            break;
        default:
            //.....
            break;
    }
}
```

这个代码有错误，因为在原来的代码中“if (COND2) break;”能够在 COND2 为 TRUE 时跳出 while(1)循环，而现在的代码中“if (COND2) break;”只能在 COND2 为 TRUE 时结束 switch 的执行，不能跳出 while(1)循环。所以如果要用 switch，代码就要更改如下。

例子：正确的代码，使用 switch，增加 flag

```
while (1) {
    int flag = 0;
    int ch = getchar ();
    switch (ch) {
        case 'A':
            //.....
            break;
        case 'B':
            if (COND1) continue;
            //.....
            break;
        case 'C':
            if (COND2) {
                flag = 1;
                break;
            }
            //.....
            break;
        default:
            //.....
            break;
    }
    if (flag) break;
}
```

有人会问，这里的 continue 会不会有问题呢？答案是不会的，因为 switch 只与 break 有语法上的关系，与 continue 没关系，所以 continue 还是会在 while(1)上起作用。

31.6 return

函数不一定要在函数尾部返回，在中间返回可能会获得更好的效果和可读性。

如果函数需要返回值，那么一定要用 return 返回值，否则编译报告警号。

如果函数不需要返回值（void 类型），那么就不要用 return 返回值，否则编译报告警号。

如果函数不需要返回值（void 类型），那么就不要在末尾添加 return，因为看着啰嗦。

31.7 goto

最好不用 goto，如果非要用 goto，就要知道为什么用，要做到心里有数，不要胡乱使用。

goto 很适合跳出多层嵌套循环，或者跳到一个集中处理的地方，goto 在库函数中使用得很多，因为库函数需要特别优化的代码。

31.8 exit()

exit()是一个函数，用于退出程序执行。例如，发现程序运行错了，就可以执行 exit()马上退出。

31.9 复合语句

复合语句是放在“{.....}”之间一组语句，当作一个语句处理且逻辑上相关联，通常用于放在 if/else/for/while 等处。另外，还可以在复合语句内定义局部变量，这样可以缩小变量的作用域。

```
例子：复合语句
if ( a > b) {
    a = 3;
    y = a + 2;
    x = a + 6;
}
```

逗号(,)可以把多个表达式串在一起当作一条语句使用，最后一个表达式的值是这些表达式的最后值。

```
例子：逗号操作，等价于上面的操作
if ( a > b)
    a = 3, y = a + 2, x = a + 6;
//或者
if ( a > b)
    x = (a = 3, y = a + 2, a + 6);
```

使用这样的逗号是节省一些代码行，但是使用的人并不多，可能有些人已经忘了学过这个。

30.10 空语句

表达式要以“;”结尾，例如“x= a + b;”。但是即使没有表达式，“;”也是一个语句，称为空语句，通常空语句用于语句占位。

```
例子：空语句
if (a >= b) ;
else {
    .....
}

for (i = 0; i < 100; i++);
```

为了更加明显地显示空语句，有人把“;”放在单独的一行。

```
例子：空语句
if (a >= b)
    ;
else {
    .....
}
```

```
for (i = 0; i < 100; i++)  
    ;
```

如果为了获得更好的表现效果，最好不用空语句，而是使用包含一句注释的空块。

```
if (a >= b){  
    /*Do nothing*/  
}  
else {  
    .....  
}  
  
for (i = 0; i < 100; i++)  
    { /*Do nothing, just for delaying some time. */ }
```

空语句还会带来代码问题，多写一个分号就可能导致代码错误。

例子：多写了一个分号，但是编译还能通过

```
//本来应该写成这个的样子  
if ( a >= b)  
    data[x] = a - b;  
//但是不小心写成下面的样子，多写了一个;  
//导致不管判断为真假，下面的语句始终执行  
if ( a >= b);  
    data[x] = a - b;
```

32.1 void

当 `void` 用在函数上时，用于说明函数没有返回值或者没有参数。`void` 的使用规则如下。

- 1) 如果函数没有返回值，就要把它声明为 `void`。例如，“`void some_func (int param);`”。
- 2) 如果函数没有参数，就要把它声明为 `void`。例如，写成“`int some_func (void);`”，不要写成“`int some_func ();`”，因为这是非常古老的写法，会导致编译器对函数原型检查不全。
- 3) 如果函数没有返回值，也没有参数，就要把它声明为“`void some_func (void);`”。
- 4) 如果函数的某个参数可以接受任意类型的指针，那么应把这个参数声明为 `void *`。

32.2 static

当一个程序由多个源文件组成时，C 语言根据函数能否被其他源文件中的函数调用，将函数分为内部函数和外部函数。

如果在一个源文件中定义的函数，只能被本文件中的函数调用，而不能被其他文件中的函数调用，这种函数称为内部函数，也称为静态函数。其他文件中可以定义与静态函数相同名字的函数，不会发生名字上的冲突。

静态函数和静态全局变量定义一样，都可以用来减少名字污染的问题。只要在函数的返回类型前加上 `static`，函数即被定义为静态函数。

32.3 inline

`inline` 用来定义内联函数，主要是用来替代 C 语言中类似函数的宏定义，具体的原因如下。

- 1) C 语言中广泛使用 `#define` 定义类似函数的宏定义。因为这种宏定义在形式和使用上像一个函数，但是没有参数压栈、出栈这些函数调用开销，因此执行效率很高。
- 2) 这种宏定义只是在形式上类似于一个函数，但是在使用它时，只是通过预处理器做简单替换，不能对它做有效的参数检查，还存在替换错误和参数副作用的问题，所以使用宏定义存在着一些隐患和局限性。
- 3) `inline` 推出的目的，就是为了取代这种类似函数的宏定义，既消除了宏定义的缺点，又能很好地保留宏定义的优点。
- 4) 使用 `inline` 定义的内联函数，如同宏定义一样，没有调用的开销，执行效率也很高。

5) 编译器会对内联函数做有效的参数检查, 保证调用正确, 没有宏定义的隐患和局限性。

注意: 只有当函数内容非常简单的时候, 才把函数定义成内联函数。这是因为, 内联函数的代码会在任何调用它的地方展开, 如果函数太复杂, 代码膨胀带来的坏处很可能会大于效率提高带来的好处。

```
例子: inline 函数
inline int compute (int q)
{
    return q + 5;
}
//这个 for 循环调用 compute 函数
for (i = 0; i < 100; i++)
    ar[i] = compute (i);
//编译器对内联函数处理后, 代码就相当于如下, 消除了函数调用的开销
for (i = 0; i < 100; i++)
    ar[i] = i + 5;
```

32.4 函数原型

为什么需要函数原型呢? 读者可以尝试下面的代码, 注意要把它们放在两个文件中。当 main.c 中没有 add()的原型时, 执行的结果是错误的, 但是当 main.c 中添加了 add()的原型时 (就是把那个注释去掉), 执行的结果就是正确的。

```
例子: main.c 和 add.c, 演示为什么需要函数原型
//Place in main.c
#include <stdio.h>
//double add (double x, double y);
main (void)
{
    printf ("%f\n", add (1, 2));
    return 0;
}
//Place in add.c
double add (double x, double y)
{
    return x + y;
}
```

当一个函数被调用时, 编译器如果无法看到它的函数原型, 那么编译器就假定这个函数返回一个整数值, 同时对传递给函数的实参进行提升, char 和 short 的实参会被转换为 int 类型, float 的实参会被转换为 double 类型。

对于上面的 main()调用 add(), 当没有函数原型时, 编译器就会认为 add()返回值为 int 类型, 同时直接把 int 1 和 2 传递过去; 但是当有函数原型时, add()返回值为 double 类型, 还要把 int 1 和 2 转换为 double 后再传递过去。

函数原型说明一个函数能接受什么样的参数, 返回什么样的结果。编译器会根据函数原型对函数的调用和定义进行检查, 检查形参和实参的数量和类型是否符合, 是否需要对象实参和返回值进行自动转换。如果编译器检查出不符合的地方, 就会报告出错或警告。

例子: 使用函数原型, 检查错误

```

int process_data (int number, double coefficient, char *buffer); //function
prototype

int res = process_data (100); //Error: not enough arguments
passed
int res = process_data (num, coeff, 899); //Error: wrong type of
arguments used
int res = process_data (num, coeff, buf, flag); //Error: too many arguments
passed
//错误的函数定义, 这里的参数与上面的原型不匹配
int process_data (double coefficient, int number, char *buffer)
{
    .....
}

```

例子, 使用函数原型, 编译正确

```

int process_data (int number, double coefficient, char *buffer);
//正确的函数定义
int process_data (double coefficient, int number, char *buffer)
{
    .....
}
//正确的函数调用, 这里 coeff 是 int 类型, 编译器会把它自动转换为 double 类型。
int coeff = 2558;
int res = process_data (num, coeff, buf);

```

函数原型要放在头文件中, 可以便于 C 文件包含使用, 还可以便于查找函数的用法。在函数原型中可以不用把形参的名字写出来, 但是为了便于查看函数的用途和用法, 最好把形参的名字写出来。

例子: 如果函数原型的参数没有名字, 看着会很不方便

```
int process_data (int, double, char *);
```

总之, 函数原型提供了一种检查函数是否被正确引用的机制, 可以检查出很多书写错误, 所以不要图省事不用, 不管是使用自己写的函数, 还是使用库函数, 都要包含合适的头文件。

32.5 参数可变

如果我们希望我们编写的函数能像 `printf` 那样可以接受任意数量的参数, 那么我们就需要使用定义在 `stdarg.h` 中的宏, 这里 `va` 是“variable-argument”的意思。

```

void va_start( va_list arg_ptr, prev_param );
type va_arg( va_list arg_ptr, type );
void va_end( va_list arg_ptr );

```

- 1) `va_list`: 为了访问变长参数列表中的参数, 必须声明一个 `va_list` 类型的对象。它是用来保存宏 `va_start`、`va_arg` 和 `va_end` 所需信息的一种类型。
- 2) `va_start`: 初始化声明为 `va_list` 的对象, 结果用于 `va_arg` 和 `va_end`。它的第一个参数就是声明为 `va_list` 的对象, 第二个参数应该是函数变参之前的参数名。
- 3) `va_arg`: 每次调用 `va_arg` 都会返回一个指定类型 (由第二个参数指定) 的数值, 同时修改用 `va_list` 声明的对象, 使该对象指向参数列表中的下一个参数。
- 4) `va_end`: 为了能够从使用 `va_start` 的函数正常返回, 必须调用这个宏。

注意：在 C 语言中，当调用一个没有原型的函数时，编译器会对每个参数执行默认的实参提升（default argument promotions）。此规则同样适用于可变参数的函数，对于那些对应于可变参数的实参（也就是对应于“...”的实参），也要执行同样默认的实参提升。默认的实参提升工作规则如下。

- 1) float 类型的实际参数将提升到 double 类型。
 - 2) signed/unsigned char 和 short 类型的实际参数提升到 int。
 - 3) 如果 int 不能存储原值，则提升到 unsigned int。
- 所以对于变参函数，va_arg 不能使用 float、char 和 short 作为第二个参数。

例子：可变参数的函数

```
#include <stdio.h>
#include <stdarg.h>
void place_into_array (int ar[], int from, int num, ...)
{
    int i;
    va_list arg_list;
    va_start (arg_list, num);
    for (i = 0; i < num; i++)
        ar[from + i] = va_arg (arg_list, int);
    va_end (arg_list);
}
int main (void)
{
    int ar[100];
    place_into_array (ar, 0, 5, 0, 1, 2, 3, 4, 5);
    place_into_array (ar, 5, 3, 10, 11, 12);
    place_into_array (ar, 8, 4, 100, 101, 102, 103);
    for (int i = 0; i < 12; i++)
        printf ("%d\n", ar[i]);
    return 0;
}
```

32.6 其他讨论

C 语言函数的参数和返回值可以是结构类型，但是尽量不要这样使用，因为如果函数使用很大的结构作为参数或返回值，那么函数调用的开销就会太大。

尽量通过参数向函数传递信息，尽量不用全局变量传递信息。当不得不使用全局变量时，可以考虑把某些相关的全局变量和使用它们的函数放到一个文件中，然后把这些全局变量定义为静态全局变量。

函数的接口要清晰，不要像 getchar() 一样，返回值既用作正常数据（0x00 ~ 0xFF），又用作是否结束的标志（结束时返回 EOF，-1）。

如果函数没有参数，函数调用也一定要加括号，否则函数什么也不做。例如，把“func();”写成了“func;”，结果“func;”就成了一条什么也不做的语句。

如果程序执行正确，main 函数就要返回一个 0，否则返回一个其他值。因为在 Linux/Unix 中，每种 Shell 都有检测程序执行是否正确的方法，在 Bash 中，可以通过检查“\$?” 查询一个程序的运行状态。

C 语言库函数是随编译器一起提供给用户的一组通用函数。库函数极大地方便了用户，同时也补充了 C 语言本身的不足。事实上，在编写 C 语言程序时，应当尽可能多地使用库函数，这样既可以提高程序的运行效率，又可以提高代码质量。

C 语言从 1970 年代开始发展并不断完善，库函数也在不断发展并完善。在库函数的发展过程中，即使有些库函数存在缺陷，但还是要保留下来，这样就会看到有些库函数的功能是重复的，所以我们在使用过程中要尽量选择那些更好用更安全的函数。

下面就探讨几个关于库函数的常见问题，建议读者没事多翻一翻库函数手册，总是会有一些发现。

33.1 使用 getopt()

当我们要实现一个功能时，首先要思考是否有库函数可以使用，别费了半天劲实现了这个功能，最后发现有库函数就可以实现这个功能，而且比我们做得还好，我们的心情会怎样呢？

例如，我们要编写一段处理命令行的代码，就是处理 `main()` 的 `argc` 和 `argv` 参数，我们可以自己编写，也不是太难。但是有现成的库函数可用，可以使用 `getopt()` 或者 `getopt_long()`，`getopt_long()` 要复杂一些，但是更加灵活。下面我们看一下 `getopt()` 的使用。

```
int getopt (int argc, char * const argv[ ], const char *optstring);
```

`getopt()` 用来分析命令行参数。`argc` 和 `argv` 是从 `main()` 传递过来的参数，`optstring` 是要处理的选项字符串。此函数依次扫描 `argv`，检查是否与 `optstring` 中的字母匹配，有匹配就会返回相应的字母。如果 `optstring` 里的字母后接着冒号“:”，则表示还有相关的参数，全局变量 `optarg` 就会指向此额外参数。如果 `getopt()` 找不到匹配的字母，就会打印出错误信息，并将全局变量 `optopt` 设为“?” 字符；如果不希望 `getopt()` 打印出错误信息，则只要把全局变量 `opterr` 设为 0 即可。

例子：使用 `getopt()`，可以处理“-a xxx -b -c -d -e”这样的命令行

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
    int ch;
    opterr = 0;
    while ((ch = getopt (argc, argv, "a:bcde")) != -1)
    {
        switch (ch)
        {
            case 'a':
```

```

        printf ("option a: '%s'\n", optarg);
        break;
    case 'b':
        printf ("option b :b\n");
        break;
    default:
        printf ("other option :%c\n", ch);
    }
}
printf ("optopt +%c\n", optopt);
return 0;
}

```

33.2 使用 qsort()

qsort()的原型在 `stdlib.h` 里，它一共有 4 个参数，没有返回值。

```

void qsort (void *base, size_t num, size_t width,
            int (*cmp)(const void*, const void*));

```

`base` 是要排序的数组或指针，`num` 是要排序的元素个数，`width` 是每个元素的大小，`cmp` 是比较函数的指针。

比较函数定义是这样的：“`int cmp (const void *a, const void *b);`”，返回值必须是 `int` 类型，两个参数的类型必须都是 `const void *`。如果是对数据进行升序排序，那么如果 `a` 比 `b` 大则返回一个正值，`a` 比 `b` 小则返回负值，相等返回 `0`；如果是对数据进行降序排序，返回值就正好相反。

例子：对 `int` 进行快速排序

```

#include <stdio.h>
#include <stdlib.h>
int cmp (const void *a, const void *b)
{
    return (*(int *)a - *(int *)b);    //注意这里使用的是减法
    //return (*(int *)b - *(int *)a);    //如果想要降序排序，就用此句代码
}
int main (void)
{
    int s[100], i;
    for (i = 0; i < 100; i++)
        s[i] = rand ();
    qsort(s, 100, sizeof(s[0]), cmp);
    for(i = 0; i < 100; i++)
        printf("%d ", s[i]);
    return(0);
}

```

33.3 文件模式问题

有一天晚上，笔者的侄子在 QQ 上问笔者一个问题。他说他已经折腾了一天，还是找不到问题的原因，他的同事也找不到问题的原因，生成的图像错位。于是他先发过来一个截图，后发过来一段相关的代码。下面就是与错误相关的代码。

例子：文件打开时的模式问题

```

#define BLEND_WIDTH    256

```



```

#define BLEND_HEIGHT 1024
void generate_blend()
{
    uchar Left_Blend_Model[BLEND_WIDTH*BLEND_HEIGHT];
    IplImage* img = cvCreateImage(cvSize(BLEND_WIDTH,BLEND_HEIGHT),
    IPL_DEPTH_8U,1);
    uchar* data = (uchar*)img->imageData;
    int h,w,i,j;
    uchar Alpha;
    FILE* fp;
    memset(Left_Blend_Model,100,sizeof(uchar)*BLEND_WIDTH*BLEND_HEIGHT);
    for(h=0;h<BLEND_HEIGHT;h++)
        for(w=0;w<BLEND_WIDTH;w++)
            if(h>w)
                Left_Blend_Model[h*img->widthStep+w] = 3;//w;

    fp = fopen("blend_left_256_1024.bin", "w");
    fwrite(Left_Blend_Model, sizeof(uchar), BLEND_WIDTH*BLEND_HEIGHT, fp);
    fclose(fp);

    fp = fopen("blend_left_256_1024.bin", "r");
    fread(data, sizeof(uchar), BLEND_WIDTH*BLEND_HEIGHT, fp);
    fclose(fp);

    cvNamedWindow("image_window",1);
    cvShowImage("image_window",img);
    cvWaitKey(0);
    cvDestroyWindow("image_window");
    cvReleaseImage(&img);
}

```

经确认，该程序是在 Windows 上写的，于是笔者让他修改两处，把程序中打开文件的两处代码改为如下样子。于是他按照笔者说的修改后，结果就运行正确。

```

fp = fopen("blend_left_256_1024.bin", "wb");
fp = fopen("blend_left_256_1024.bin", "rb");

```

他之前写的代码是按照文本模式打开的文件，笔者只是让他改为用二进制模式打开文件，就是在 mode 的参数中带上“b”。因为在 Windows 上文件有二进制模式和文本模式的区别。

- 1) 在 Windows 系统中，文本模式下，文件以“\r\n”代表换行。如果用文本模式打开文件，并用 fputs、fwrite、fprintf 等写函数写入换行符“\n”时，这些写函数会自动在“\n”前面加上“\r”，即实际写入文件的是“\r\n”。
- 2) 在 Windows 系统中，二进制模式下，这些写函数就不会对要写的内容进行处理，写出去什么东西，就是什么东西。
- 3) 在 Unix/Linux 系统中，文本模式和二进制模式是没有区别的，因为在文本模式下，文件就是以“\n”代表换行的，所以就不存在自动转换为“\r\n”的问题。

其实在 Windows 上编写文件读写的程序时，就是需要注意二进制和文本模式的区别。如果他之前仔细阅读过库函数手册，像这种问题根本就是一个小的问题，几分钟就能解决掉，而不用折腾一整天。

后来，笔者的同事也碰到了同样的问题，他把 Linux 下的一个数十行的小程序挪到 Windows 下编译运行，发现写出来的文件多出来一些字节，笔者用几秒钟就把它搞定了。其实，如果你有平时的积累，这些都是非常细小的问题。

33.4 返回值的问题

例子：这段代码有什么问题，这个例子在多本书中出现

```
char ch;
while (ch = getchar() != EOF) {
    ...putchar (ch)
}
```

如果输入来自于键盘，这段代码没有问题；如果输入来自于一个文本文件的重定向，这段代码也没有问题；如果输入来自于一个二进制文件的重定向，这段代码就有问题了。

`getchar()`的返回值是 `int` 类型，取值为 `0x00~0xFF` 和 `-1`，`0x00~0xFF` 是正常的字符输入，`-1` 是 EOF 的值，表示结束。所以不能把 `getchar()` 的返回值赋给 `char` 类型的变量，返回值超出了 `char` 类型的范围。

当输入来自于键盘或者来自于一个文本文件的重定向时，`ch` 的值都是在 `0x00~0x7F` 之间。当输入结束时，`getchar()` 返回 EOF，EOF 的值是 `-1`，赋给 `ch` 时要转换为 8bit 的 `-1`，然后 `ch` 和 EOF 比较，`ch` 又转换为 32bit 的 `-1`，二者相等，可以正常退出循环。

但是当输入来自于一个二进制文件的重定向是，`getchar()` 返回的字节可能包含 `0xFF`。当 `0xFF` 赋给 `ch` 后，变成了 8bit 的 `-1`，然后 `ch` 和 EOF 比较，`ch` 又转换为 32bit 的 `-1`，二者相等，于是提前退出循环。这是因为 `0xFF` 和 EOF (`-1`) 赋给 `ch` 后，转换之后的值没有差别，从而导致错误。

例子：正确的修改办法就是把 `ch` 改为 `int` 类型，然后就会正常运行

```
int ch;
while (ch = getchar() != EOF) {
    ...putchar (ch)
}
```

另外，如果使用 “`unsigned char ch;`”，也会有问题，会出现永远退不出循环的情况，读者可以自行分析一下。

像 `getchar()` 这样，在返回值中既包含正常的输入值，又包含表示是否结束的状态值，看起来很方便，而且这种方法在 C 语言中很常用，但是这种方法违背了软件工程的设计原则。更安全的设计方法是让函数返回两个值：一个用于状态值，另一个用于输入值，只有当状态有效时，才使用输入值。

33.5 控制字符问题

我们经常使用 `scanf` 和 `printf` 做数据的输入和输出，它们拥有非常多的格式控制字符，很难记得住，使用时还是要查看一下库函数手册。

`scanf` 和 `printf` 依靠格式串中的控制字符进行数据的输入和输出，但是 C 语言编译器不会检测格式串中格式控制的数量是否和输入/输出项的数量相匹配，也不会检测格式控制是否适合要输入/输出项的数据类型，这些完全由程序员保证。

`scanf` 和 `printf` 的控制字符基本一样，于是就给人一种错觉，就是它们的控制字符是等价的。其实这是不对的，有些地方还是有区别的。例如它们对 `float` 和 `double` 参数使用的控制字符。

例子：scanf 和 printf 对 float 和 double 参数使用的控制字符

```
float f;
double d;
scanf ("%f", &f);
scanf ("%lf", &d);
printf ("%f", f);
printf ("%f", d);
```

对于 scanf，“%f”用于 float 变量，“%lf”用于 double 变量，因为调用 scanf() 时，传递的是变量地址，scanf 需要根据“%f”和“%lf”来区分是 float 还是 double 变量，然后把相应的数值写进变量里。

对于 printf，“%f”既用于 float 变量，也用于 double 变量，因为调用 printf() 时，float 变量自动提升为 double 类型。

有人会问为什么调用 printf 会把 float 变量自动提升为 double 类型呢？我们已经包含了 stdio.h。先看一下 printf 的函数原型。

```
extern int printf(const char*, ...);
```

你注意到“...”了吗？“...”表示是函数的变参，可以有任意个参数。在函数调用时，因为变参位置上的实参没有形参可以对应，所以就只能按缺省的处理（包括 C 语言发展历史的原因），就需要把 float 变量自动提升为 double 类型。

对于 printf 和 scanf 的格式控制，现在已经做了很多扩充，不是你大学时学习的那一点儿，所以如果想要好好地使用它们，更加灵活地进行输入和输出，就要仔细研究一下库函数手册。

例如，如何让程序根据输出的结果自动实现输出的合理宽度，而不是使用这样的代码

```
printf ("%5d", data);
```

可以使用下面的代码，其中“*”和 width 匹配，可以由 width 动态控制输出宽度。

```
printf ("%*d", width, data);
```

33.6 缓冲区问题

例子：缓冲区可能溢出

```
char buffer[18];
int value;
sprintf (buffer, "The result is %d", value);
```

对于上面的代码，当 value = 1 时，这段代码可以正常工作，但是当 value=123456 时，这段代码就不能正确地工作，因为 buffer 太小，sprintf 写它的时候会溢出，会把别的数据破坏掉，程序可能会跑飞。解决办法就是让 buffer 大一些，这样就不会错了。

更好的解决办法是使用 sprintf，当然还要 buffer 大一些。

例子：缓冲区不会溢出

```
char buffer[40];
int value;
sprintf (buffer, sizeof (buffer), "The result is %d", value);
```

对比一下 sprintf 和 sprintf 的区别：

```
int sprintf (char *str, const char *format, ...);
int sprintf (char *str, size_t size, const char *format, ...);
```

sprintf 多了一个参数 size，就是这个 size 可以保证缓冲区不会溢出。

- 1) 如果格式化后的字符串长度小于 `size`，则将此字符串全部复制到 `str` 中，并在其后添加一个字符串结束符 (`\0`)，返回值为写入的字符串长度。
- 2) 如果格式化后的字符串长度不小于 `size`，则只将其中的 (`size-1`) 个字符复制到 `str` 中，并给其后添加一个字符串结束符 (`\0`)，返回值为写入的字符串长度。

C 语言库函数还有一些可能会出现这样问题的函数，例如 `strcpy()`、`get()`，它们分别有对应的替代函数 `strncpy()`、`fgets()`。

例子：最好使用更安全的函数

```
char * strcpy (char*, const char*);           //可能溢出
char * strncpy (char*, const char*, size_t); //更加安全
char * gets (char *buf);                     //可能溢出
char * fgets (char *buf, int size, FILE* fp); //更加安全
```

33.7 可重入问题

当我们编写中断处理程序或者多线程的程序时，如果使用到了库函数，就要注意考虑使用可重入的库函数。所谓可重入函数，就是只要输入数据相同，就应该产生相同的输出。例如，`strcpy()`就是可重入函数，但是 `strtok()`就不是可重入函数。

```
char * strtok (char *str, const char *delim);
```

`strtok()`用来把字符串分割成一个个片段。`str` 指向欲分割的字符串，`delim` 则为分割字符串中包含的所有字符。当 `strtok()`在 `str` 中发现 `delim` 中包含的分割字符时，就会将该字符改为`\0`。第一次调用时要向 `str` 传入要分割的字符串，以后的调用就要向 `str` 传入 `NULL`。每次调用成功则返回指向被分割出片段的指针。

例子：使用 `strtok`

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char sentence[] = "This is a sentence with 7 tokens";
    printf ("%s\n", sentence);
    char *tokenPtr = strtok (sentence, " ");
    int i = 0;
    while (tokenPtr != NULL) {
        printf ("%d: %s\n", i, tokenPtr);
        tokenPtr = strtok (NULL, " ");
        i++;
    }
    return 0;
}
```

`strtok()`不可重入的原因是：在这个函数内部定义了一个 `static` 变量，这个变量记录了每次分割的位置，所以第一次调用时要向 `str` 传入要分割的字符串，以后的调用就要向 `str` 传入 `NULL`。

在 Linux 上 `strtok()`有对应的可重入 `strtok_r()`，但是不知道为什么在 Windows 的 MinGW 上没有。

例子：使用 `strtok_r()`

```
#include <stdio.h>
```

```

#include <string.h>
int main(void)
{
    char sentence[] = "This is a sentence with 7 tokens";
    printf ("%s\n", sentence);
    char *lastPtr = NULL;
    char *tokenPtr = strtok_r (sentence, " ", &lastPtr);
    int i = 0;
    while (tokenPtr != NULL) {
        printf ("%d: %s\n", i, tokenPtr);
        tokenPtr = strtok_r (NULL, " ", &lastPtr);
        i++;
    }
    return 0;
}

```

例子: `strtok_r()`的实现, 如果在 Windows 使用, 可以使用这段代码

```

char *strtok_r (char *str, const char *delim, char **save_ptr)
{
    char *token;
    if (str == NULL) str = *save_ptr;

    //Scan leading delimiters.
    str += strspn(str, delim);
    if (*str == '\0')
        return NULL;

    //Find the end of the token.
    token = str;
    str = strpbrk(token, delim);
    if (str == NULL)
        //This token finishes the string
        *save_ptr = strchr(token, '\0');
    else {
        //Terminate the token and make *SAVE_PTR point past it
        *str = '\0';
        *save_ptr = str + 1;
    }
    return token;
}

```

另外还有一些函数使用了静态局部变量, 例如 `tmpfile()`、`asctime()`、`ctime()`、`gmtime()`、`localtime()`, 使用这些函数时也要注意。

C 语言提供几种预处理功能，包括文件包含、宏定义、条件编译等预处理命令。预处理是在编译之前做的处理，而且预处理不做语法检查。

34.1 文件包含

C 程序通常由头文件(.h)和代码文件(.c)组成，头文件用于保存程序的声明(declaration)，而代码文件用于保存程序的实现(implementation)。在整个程序中，头文件不是最重要的部分，但它是 C 程序中不可缺少的组成部分。

头文件中通常要包含程序范围内使用的东西，包括：宏定义、类型定义、全局变量声明、函数原型声明。

34.1.1 #include

#include 用于包含头文件，有两种形式：`#include <...>`和`#include "..."`，它们决定了对指定头文件的搜索方式。`#include <...>`用于包含编译器或系统提供的标准头文件，`#include "..."`用于包含自己定义的头文件。

例子：包含头文件

```
#include <stdio.h>
#include <stdlib.h>
#include "ftl_def.h"
#include "ftl_func.h"
```

注意：`#include` 应该只包含文件名，不要把目录名也带上，否则不好维护。编译器有专门的选项用于指定搜索路径，通常是“`-I <Path1> -I <Path2> -I <Path-n>`”。

34.1.2 自定义头文件

编写代码时要根据需要设计自己的头文件，把公共的部分按照一定的格式放到头文件里。编写头文件要按照一定的格式，可以参照下面的例子。

例子：常用头文件的格式

```
//Copyright .....
#ifndef MY_HEADER_H           //用于防止 my_header.h 被重复包含
#define MY_HEADER_H
#include <...>                 //包含标准库的头文件
#include "... "               //包含非标准库的头文件

//宏定义常量和代码
```

```

#define MACRO_NAME1 0x24
#define MACRO_NAME2 0x89
#define MACRO_FUNC1 ({...})

//自定义类型
typedef struct {
.....
} SELF_DEFINE_TYPE;

//全局变量声明
extern int gGlobalVar1;
extern int gGlobalVar2;

//函数原型
void Function1 (...);
void Function2 (...);
#endif

```

34.1.3 标准头文件

例子：常用的标准头文件

```

#include <assert.h> //断言
#include <ctype.h> //字符处理
#include <errno.h> //定义错误码
#include <float.h> //浮点数处理
#include <limits.h> //定义各种数据类型最值常量
#include <locale.h> //定义本地化函数
#include <math.h> //定义数学函数
#include <stdio.h> //定义输入/输出函数
#include <stdlib.h> //定义杂项函数及内存分配函数
#include <string.h> //字符串处理
#include <time.h> //定义关于时间的函数
#include <stdint.h> //整型环境

```

34.1.4 total_header.h

对于一个包含多个 C 文件的程序，每个 C 文件都要包含所用的头文件，所以我们经常看到每个 C 文件头部都有一堆 `#include "....."`，而且每个 C 文件使用的头文件还各不相同，感觉很繁琐。

那么有没有方便的方法解决这个问题呢？对于不太大的程序，例如嵌入式程序，只有几十个 C 文件时，就可以使用这样的方法：编写一个大的头文件，例如 `total_header.h`，里面包含所有要使用的头文件，然后在每个 C 文件中，只需要使用 `#include "total_header.h"`。

这个方法只适用于文件不多的简单程序，对于文件很多的复杂程序，还是要使用原来的方法。这是因为编译 C 程序时，通常进行分别编译，然后链接生成可执行文件。当只有几个 C 文件发生改变时，就只须编译这几个文件，然后再链接生成可执行文件，这样可以节省编译时间。但是当一个头文件发生改变时，就需要编译所有包含它的 C 文件，如果使用 `total_header.h`，所有的 C 文件都要重新编译，很浪费时间。

但是对于文件不多的程序，头文件一般不会做修改，或者改动的频率并不大，所以重新编译所有 C 文件的问题应该不会频繁，或者编译时间不会那么长。

使用 `total_header.h` 方法还有一个好处，就是能够减少宏定义、变量和函数原型上的冲突，一旦有冲突，马上就会报告出来。

34.2 宏定义

34.2.1 说明

使用宏定义可以减少代码错误，提高代码的可读性、可修改性、通用性和一致性。

- 1) `#define` 用于定义一个宏，`#undef` 用于取消之前用 `#define` 定义的宏。
- 2) 宏定义常用于定义常量和函数宏。
- 3) 宏定义通常在文件的最开头。
- 4) 宏定义的名字一般使用大写。
- 5) 宏定义的末尾不能加分号。
- 6) 宏定义的名字和括号之间不能有空格，参数不存在类型。
- 7) 预处理只对使用宏的地方做展开处理，不会对表达式求解。
- 8) 如果宏定义不严格，那么宏替换就容易出问题，或者编译不过，或者代码错误。
- 9) 宏替换会让代码变长，会提高运行效率，而函数调用不会让代码变长，但有额外的开销。

34.2.2 常量宏

例子： `stdio.h` 中定义的一些常量

```
#define _IOFBF    0x0000 /* full buffered */
#define _IOLBF    0x0040 /* line buffered */
#define _IONBF    0x0004 /* not buffered */
#define _IOMYBUF  0x0008 /* stdio malloc()'d buffer */
#define _IOEOF    0x0010 /* EOF reached on read */
#define _IOERR    0x0020 /* I/O error from system */
#define _IOSTRG   0x0040 /* Strange or no file descriptor */
#define SEEK_SET  0
#define SEEK_CUR  1
#define SEEK_END  2
#define EOF       (-1)
```

34.2.3 函数宏

函数宏（Function-like Macro）通常包含一段代码，以实现某些功能。宏名之后可以不带参数，也可以带参数。

例子：不带参数的函数宏

```
#define ENABLE_ALL_INTERRUPT() \
    ({int xx = __cpu_mfsr (CPU_SR_INT_MASK); \
     xx |= 0xffffffff; \
     __cpu_mtsr (xx, CPU_SR_INT_MASK); })
#define DISABLE_ALL_INTERRUPT() \
    ({int xx = __cpu_mfsr (CPU_SR_INT_MASK); \
     xx &= 0x00000000; \
     __cpu_mtsr (xx, CPU_SR_INT_MASK); })
```

通常函数宏都带有几个参数，当编译器遇到带有参数的函数宏时，与之相关的参数就被

实际参数替换。如果函数宏写得不严格，那么宏替换时就会出问题，扩展出错误的代码，或者语法错误，或者代码错误。

对于宏定义中参数，每个都要有“()”括起来，否则很容易出错。

例子：有问题的宏

```
#define MAX(a, b) (a > b ? a : b)
result = MAX (x, y);
//这个问题没有问题，预处理后为 result = (x > y ? x : y)
result = MAX (x & 2, y & 3);
//这个问题有问题，预处理后为“result = (x & 2 > y & 3 ? x & 2 : y & 3);”，因为“>”比“&”优先级高
```

例子：改正后的宏

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
result = MAX (x & 2, y & 3);
//现在没有问题，预处理后 result = ((x & 2) > (y & 3) ? (x & 2) : (y & 3));
```

当宏定义包含多个语句的时候，就要想办法把它们合理地包起来，不能简单地使用“{}”。

例子：有问题的宏

```
#define OPERATE_DATA(x, y, a, b) {(x) = (a) + (b); (y) = (a) - (b);}
if (result > 55)
    OPERATE_DATA (h, k, m, n);
else
    OPERATE_DATA (h, k, p, q);
//预处理后为如下，编译会发生错误，因为多了个分号 (;)，导致语法错误
if (result > 55)
    {(h) = (m) + (n); (k) = (m) - (n);};
else
    {(h) = (p) + (q); (k) = (p) - (q);};
```

我们可以使用两种方法修改这种错误：一是使用 `do {...} while(0)`，因为条件等于 0，代码只会执行一次，而且 `while(0)` 从语法上要求必须要加分号 (;)；二是使用 `{...}`，它是 GCC 的扩展，使用起来更方便。

例子：改正后的宏

```
#define OPERATE_DATA(x, y, a, b) do {(x) = (a) + (b); (y) = (a) - (b);} while (0)
#define OPERATE_DATA(x, y, a, b) ({(x) = (a) + (b); (y) = (a) - (b);})
```

不管是使用 `do {...} while(0)`，还是使用 `{...}`，都可以在“{ }”中定义局部变量。通过引入中间变量，可以降低编写宏的难度。

例子：宏定义中使用局部变量

```
#define DISABLE_ONE_INTERRUPT(val) \
    ({int xx = __cpu_mfsr (CPU_SR_INT_MASK); \
     xx &= ~(1 << (val)); \
     __cpu_mtsr (xx, CPU_SR_INT_MASK); })
```

在使用宏定义的时候，参数要尽量简单，不能带有副作用，也不能是函数调用，否则可能会导致代码多次执行，或者代码执行错误。

例子：宏替换中的参数带有副作用

```
#define P0(x) ((x) ^ ((x) << 9))
result = P0 (a++);
//预处理后为如下，result 不对，而且 a 多加了一次
result = ((a++) ^ ((a++) << 9));

result = P0 (fetch_data(12));
```

```
//预处理后为如下, fetch_data 执行了两次, 增加了执行时间,
//另外如果 fetch_data 返回的结果不一样, result 也是错误。
result = ((fetch_data(12)) ^ ((fetch_data(12)) << 9));

//所以要把它们分别修改为如下
result = P0 (a);
a++;
tmp = fetch_data (12);
result = P0 (tmp);
```

34.2.4 变参宏

C99 对宏定义做了增强, 支持可变参数的宏定义, 通过 `__VA_ARGS__` 实现, 但实际使用起来可能很受限。

```
例子: 使用变参宏
#include <stdio.h>
#define debug_output(format, ...) printf (format, __VA_ARGS__)
int main (void)
{
    debug_output ("%d\n", 5);
    debug_output ("%d %d\n", 5, 6);
    debug_output ("%d %d %d\n", 5, 6, 7);
    return 0;
}
```

34.2.5 #和##

在宏定义中, #用于把其后的串变成用双引号包围的串, ##用于把两个标记 (token) 拼接在一起, 形成一个新标记。

```
例子: #和##的使用
#include <stdio.h>
#define STR(x) #x
#define CONCAT(a, b) a##b
int main (void)
{
    printf ("%s\n", STR (I am Han Meimei));
    printf ("%d\n", CONCAT (123, 456));
    printf ("%s\n", STR (CONCAT (123, 456)));
    return 0;
}
//运行结果
I am Han Meimei
123456
123456
```

```
例子: 使用#define 定义输出语句 TRACE(), 以方便程序的调试。
#define TRACE(var, fmt) printf("TRACE: " #var " = " #fmt "\n", var)
TRACE(result, %d); //对应于: printf("TRACE: result = %d\n", result);
```

```
例子: 填充结构
#define FILL(a) {a, #a}
typedef enum IDD {OPEN, SAVE, CLOSE} IDD;
typedef struct CMD {IDD id; const char * cmd;} CMD;
CMD cmd[] = {FILL(OPEN), FILL(SAVE), FILL(CLOSE)};
相当于:
```

```
CMD cmd[] = {{OPEN, "OPEN"}, {SAVE, "SAVE"}, {CLOSE, "CLOSE"}};
```

34.2.6 转换宏

例子：使用#或##，但是不能产生希望的预处理

```
#include <limits.h> //因为 limits.h 中定义了 INT_MAX，是整型的最大值
#define A          (2)
#define STR(s)     #s
#define CONS(a,b)  int(a##e##b)

printf("int max: %s", STR(INT_MAX));
//输出错误，因为这一行展开后为
printf("int max: %s", "INT_MAX");

printf("%s", CONS(A, A));
//编译错误，因为这一行展开后为
printf("%s", int(AeA));
```

INT_MAX 和 A 没有被按期望地展开，直接就被使用，所以导致错误。解决这个问题方法很简单，就是多加一层转换宏。转换宏的用意就是把所有宏的参数在这层里全部展开，然后转换宏里的那一个宏（_STR 或_CONS）就能得到正确的宏参数。

例子：使用#或##，加上一层转换宏，就能产生希望的预处理

```
#define A          (2)
#define _STR(s)    #s
#define STR(s)     _STR(s)           //转换宏
#define _CONS(a, b) int(a##e##b)
#define CONS(a, b) _CONS(a, b)      //转换宏

printf("int max: %s", STR(INT_MAX));
//这一行展开后为，因为 STR(INT_MAX) --> _STR(0x7FFFFFFF) --> "0x7FFFFFFF"
printf("int max: %s", "0x7FFFFFFF");

printf("%d", CONS(A, A));
//这一行展开后为，因为 CONS(A, A) --> _CONS((2), (2)) --> int((2)e(2))
printf("%d", int((2)e(2)));
```

例子：使用#或##，生成变量名

```
#define _CURRENT1(type, var, line) type var##line
#define _CURRENT0(type, line)      _CURRENT1(type, current, line)
#define CURRENT(type)              _CURRENT0(type, __LINE__)
例：CURRENT(static int);
扩展为：static int current70;           //70 表示该行行号
说明：每次只能解开当前层的宏，所以__LINE__在第二层才能被解开。
第一层：CURRENT(static int); --> _CURRENT0(static int, __LINE__);
第二层：                --> _CURRENT1(static int, current, 70);
第三层：                --> static int current70;
```

34.2.7 预定义宏

C 语言编译器都会预定义宏，表 34-1 所列的宏是这些常用的宏。

编写程序时可以把编译程序的日期和时间（__DATE__、__TIME__）记录到一个字符串里，用作程序的版本，在程序实际运行时，就可以查看当前版本的编译日期和时间。

表 34-1 预定义宏

宏定义名	说 明
<code>__FILE__</code>	含有当前被编译文件的文件名，它是一个字符串
<code>__LINE__</code>	含有当前被编译代码行的行号，它是一个整数
<code>__TIME__</code>	含有源文件被编译时的时间，形式为“时:分:秒”的字符串，例如“Aug 10 2015”
<code>__DATE__</code>	含有源文件被编译时的日期，形式为“月/日/年”的字符串，例如“11:29:36”
<code>__STDC__</code>	如果编译器是标准的，那么 <code>__STDC__</code> 的值是 1，否则为其他值。它是一个整数
<code>__func__</code>	含有当前被编译函数的函数名，它只能在函数内使用。它是一个字符串

编写程序时还可以使用`__FILE__`、`__LINE__`，用来记录调试信息，例如常用的 `assert()` 实际就被定义成如下

```
#define assert(e) ((e) ? (void)0 : _assert(#e, __FILE__, __LINE__))
```

34.3 条件编译

`#if` `#elif` `#else` `#endif`，这些命令允许根据常数表达式的结果有条件地包含部分代码。这些条件编译命令的说明如下：

- 1) `#ifxxx`和`#ifdefxxx`是有区别的。对于`#if xxx`，只有当`xxx`定义了而且值为 1 时，判断才为 `true`，否则为 `false`；对于`#ifdefxxx`，只要`xxx`定义了，不管有值没值，不管是什么值，判断就为 `true`，否则为 `false`。
- 2) 判断条件可以使用比较和逻辑等操作符，例如`#if (MAX_LINE >= 250) || (MAX_PAGE >= 12)`。
- 3) 可以使用 `defined(xxx)`，用于检测某个宏名是否存在，所以`#ifdefxxx`和`#if defined (xxx)`是等价的。使用 `defined(xxx)`更加灵活，可以与其他判断组成逻辑表达式。
- 4) `#if 0` 或 `#if 1` 常用于屏蔽某些暂时不用的代码。

例子：条件编译

```
#if 0
#else
#endif

#if (USE_MMC == 1)
#elif (USE_USB == 1)
#else
#endif

#if (USE_MMC || USE_USB)
#else
#endif

#if USE_DEBUG
#else
#endif

#ifdef USE_DEBUG
#else
#endif
```

```

#ifndef USE_DEBUG
#else
#endif

#if defined(USE_DEBUG)
#elif defined(USE_ASSERT)
#else
#endif

```

34.4 其他命令

34.4.1 #line

`#line` 用于改变 `__FILE__` 和 `__LINE__` 的内容。`#line` 主要用于调试和特殊应用，通常不会使用它。`#line` 的格式如下：

```
#line number "filename"
```

34.1.2 #error

`#error` 用于强制编译器停止编译，主要用于调试程序。当遇到 `#error` 时，对应的错误信息就会显示出来。`#error` 的格式如下

```
#error error-message
```

注意：`error-message` 不需要使用双引号括起来。

34.1.3 #pragma

`#pragma` 是编译器实现时定义的命令，它允许向编译器传入各种指令，完成一些特定的操作。不同编译器对 `#pragma` 支持不同，有的可能根本就不支持，所以要根据使用的编译器，查找相应的用户手册。

34.5 预处理输出

通常程序员不会关心编译器的预处理过程，但是某些时候可能需要查看一下预处理的内容。对于 GCC，使用下面的命令把 `hello.c` 的预处理内容输出到 `hello.i` 中

```
gcc -E -P -o hello.i hello.c
```

`gcc` 实际是调用 `cpp` 进行预处理，可以查看 `cpp` 的手册，进一步了解预处理的细节。

笔者曾经有几年非常热衷于户外活动，虽然没有超强的体力，但还是参加过两次河北小五台山的活动，在第二次南中西三台连穿活动中，从早晨 6:00 走到晚上 9:00，15 个小时的连续行走、上山、下山，即使是感觉要崩溃了，我们还是要坚持走下去。

户外活动最害怕的就是各种突发事件，想一想笔者那些年的户外活动，其实还算安全顺利，但是也经历过山坡落石的情况，现在想起来还是后怕。虽然这几年笔者不怎么参加户外活动，但还是比较关心户外活动的各种新闻，每年总要有一些户外事故发生，因为大雨、风雪、严寒、炎热、雷击、透支、脱水、大意等引发受伤、死亡、失踪等事故。

程序运行也是这样，也会碰到各种问题和错误。程序运行时产生的错误有两种：一种是代码本身就存在编写缺陷或错误，测试不全，没有覆盖到；另一个是代码的强壮性不够，结果在不理想的情况下运行遭遇各种事件。例如，因为大家过年春节回家集中买票，导致购票网站阻塞或崩溃。

35.1 错误检查和处理

错误处理是编写代码时必须要做的事情，代码可能有缺陷，输入可能异常，输出可能丢失，数据可能错误，设备可能失效，当这些错误出现时，程序员必须要保证代码能检查出来错误并能正确地处理掉它。

添加错误检查 and 处理的代码，就会让代码膨胀，就会让程序被错误检查 and 处理的代码所占据。所谓占据，并不是说错误检查 and 处理是代码的全部，而是说到处都是凌乱的错误检查 and 处理的代码，无法看明白代码实际要做的事情。错误检查 and 处理很重要，但是它搞乱了代码逻辑，就是错误的做法。

在 Java 中，通过让发生错误时产生异常，把错误处理代码从程序执行的主路径上分离出来。但是，C 语言不能像 Java 那样产生异常，不能有太好的手段，只能使用常规的错误检查 and 处理的手段，所以要尽量处理好代码的结构，尽量保证代码的可读性。

错误检查 and 处理代码的最大问题是会带来多层嵌套，所以应该使用之前介绍的消除嵌套的方法，减少嵌套的层数。

35.2 封装函数

C 语言中最常见的错误检查 and 处理就是检查 `malloc()` 的返回值是否为 `NULL`，如果是 `NULL`，就打印提示信息，并退出执行。

例子：检测 malloc() 的返回值

```
p = malloc (10 * sizeof (struct EMPLOYEE));
if (p == NULL) {
    fprintf (stderr, "Error: malloc for EMPLOYEE\n");
    exit (0);
}
```

如果程序在不同部分都要分配内存空间，就要重复复制这样的代码，对于小的程序可以做，对于大的程序就有点啰嗦。所以我们可以把 malloc() 封装一下，包含详细的错误处理，下面就是相应的代码。

例子：使用 MALLOC()

```
void *malloc_wrapper (size_t size, char *file, int line, const char *func)
{
    void *p = malloc (size);
    if (p == NULL) {
        fprintf (stderr, "Error: malloc %d bytes in %s %d %s()\n",
                size, file, line, func);
        exit (0);
    }
    return p;
}
#define MALLOC(size) malloc_wrapper (size, __FILE__, __LINE__, __func__)

int main (void)
{
    struct EMPLOYEE *employee = MALLOC (10 * sizeof (struct EMPLOYEE));
    .....
    struct MANAGER *manager = MALLOC (2 * sizeof (struct MANAGER));
    .....
    return 0;
}
```

35.3 使用断言

assert 称为断言，用来保证代码的正确性，用来检查程序执行中是否出现了非法的数据。如果出现了非法的数据，就终止程序的执行以免导致严重后果，同时也便于查找错误。assert 是在 assert.h 中定义，通常是如下的样子。

例子：assert 的定义

```
#ifdef NDEBUG
//If not debugging, assert does nothing.
#define assert(expr) ((void)0)
#else
void _assert (const char*, const char*, int);
#define assert(expr) ((expr) ? (void)0 : _assert(#expr, __FILE__,
__LINE__))
#endif
```

当 NDEBUG 不定义的时候，就使能 assert，否则就禁止 assert。assert 检查表达式 expr 的真假，如果其值为假，就打印出来 expr、文件名和行号，然后执行 abort，让系统杀掉自己并生成 coredump（是否生成 coredump，取决于系统配置）；如果其值为真，则继续向下执行。

下面是使用 assert 的几个原则。

1) 程序一般分为 Debug 和 Release 两个版本，Debug 用于内部开发，Release 发行给用户

使用。

- 2) `assert` 应该只在 `Debug` 中起作用，用于检查不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，前者是不应该出现的，例如编程时的笔误，后者是必然存在的并且是一定要做出处理的。
- 3) 在编写函数时，要进行反复考查，并且问自己“我打算做哪些假定”，一旦确定好假定，就使用 `assert` 对假定进行检查。
- 4) 每个 `assert` 应该只检查一个条件，因为当同时检查多个条件时，如果检查失败，无法直观地判断是哪个条件失败。
- 5) 不要滥用 `assert`，千万不能在 `assert` 内执行带有功能的表达式，因为 `assert` 只在 `NDEBUG` 不定义时有效。如果真的在 `assert` 内执行了功能，那么当程序在关闭 `assert` 时，执行就会完全错误。
- 6) 可以使用 `assert` 做防御性编程：在函数的入口处，使用 `assert` 检查输入参数的正确性；在函数退出之前，使用 `assert` 检查输出数据的正确性。

例子：应该只检查一个条件

```
int reset_buffer_size (int size)
{
    assert (size >= 0 && size <= MAX_BUFFER_SIZE);
    .....
}
```

应该改为如下

```
int reset_buffer_size (int size)
{
    assert (size >= 0);
    assert (size <= MAX_BUFFER_SIZE);
    .....
}
```

例子：不能检查带有功能的表达式

```
assert (set_state() == WORK)
assert (count++ < 100);
//应该改为如下
state = set_state();
assert (state == WORK)
assert (count < 100);
count++;
```

例子：使用 `assert` 的方式

```
#include <stdio.h>
#include <assert.h>
int main( void )
{
    FILE *fp;
    //以只写的方式打开一个文件，不管文件存在不存在，都要创建这个文件
    fp = fopen ("test.txt", "w");
    assert (fp); //所以这里不会出错
    fclose (fp);

    //以只读的方式打开一个文件，如果不存在，打开文件就失败
    fp = fopen ("noexitfile.txt", "r");
    assert (fp); //所以这里出错
    fclose (fp); //程序永远都执行不到这里来
    return 0;
}
```


运行结果如下:

```
Assertion failed: fp, file a41.c, line 13
```

```
This application has requested the Runtime to terminate it in an unusual way.  
Please contact the application's support team for more information.
```

例子: newlib 的 malloc.c 中的代码

```
static void munmap_chunk(mchunkptr p)
{
    INTERNAL_SIZE_T size = chunksize(p);

    assert (chunk_is_mmapped(p));
    assert (!((char*)p >= sbrk_base && (char*)p < sbrk_base + sbrked_mem));
    assert ((n_mmaps > 0));
    assert (((p->prev_size + size) & (malloc_getpagesize-1)) == 0);

    n_mmaps--;
    mmapped_mem -= (size + p->prev_size);
    int ret = munmap((char *)p - p->prev_size, size + p->prev_size);
    //munmap returns non-zero on failure
    assert (ret == 0);
}
```

第 36 章

内存映像

内存映像是针对进程说的，程序是静态的，进程是动态的。内存映像是指操作系统把程序转化为进程的过程中，是如何安排各个内存空间的。

36.1 程序编译后的 section

```
例子: hello.c
#include <stdio.h>
int gInitVardata = 0x12345678;
int gNoInitVardata;

void func1 (void)
{
    printf ("I am func1\n");
}

int main (void)
{
    printf ("Hello, World\n");
    printf ("%d %d\n", gInitVardata, gNoInitVardata);
    func1 ();
    return 0;
}
```

程序编译后生成的可执行文件是由多个 section 组成的。

```
例子: hello 的 section, 使用 MinGW
#gcc -O2 -o hello.exe hello.c
#objdump -hr hello.exe
hello.exe:      file format pei-i386
Sections:
Idx Name          Size      VMA          LMA          File off  Algn
  0 .text          00007384  00401000  00401000  00000400  2**4
                CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data          0000002c  00409000  00409000  00007800  2**2
                CONTENTS, ALLOC, LOAD, DATA
  2 .rdata         00000740  0040a000  0040a000  00007a00  2**5
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .eh_frame     000015a0  0040b000  0040b000  00008200  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .bss          00000a34  0040d000  0040d000  00000000  2**5
                ALLOC
  5 .idata         00000688  0040e000  0040e000  00009800  2**2
                CONTENTS, ALLOC, LOAD, DATA
  6 .CRT           00000018  0040f000  0040f000  0000a000  2**2
                CONTENTS, ALLOC, LOAD, DATA
  7 .tls           00000020  00410000  00410000  0000a200  2**2
```

36.2 程序运行时的内存空间

程序运行时的内存空间可以分为以下 6 个部分（实际的空间安排可能不是这样的顺序）。

- 1) 程序代码区 (Text): 用于存放程序的代码。
- 2) 只读数据区 (Rodata): 用于存放程序的常量。
- 3) 数据区 (Data): 这是静态数据区, 初始化的全局变量和静态局部变量放在 Data 区, 这里的初始化指的是程序内对变量的赋值初始化。
- 4) 数据区 (BSS): 这是静态数据区, 未初始化的全局变量和静态局部变量放在 BSS 区, 这里的未初始化指的是程序内没有对变量的赋值初始化。在操作系统装载程序时, BSS 区域会被初始化为 0。
- 5) 堆 (Heap): 这是动态内存区, 由程序员负责管理, 通过调用 malloc()和 free()进行内存的分配和释放。
- 6) 栈 (Stack): 它也称为堆栈区, 用来存放函数的返回地址、函数参数值、非静态局部变量和 CPU 的当前状态等。其操作方式类似于数据结构中的栈, 也就是后进先出 LIFO。

可执行文件中的各个 section 根据他们各自的属性被装载到各个区域内, 可以看出 Data 区域并不是连续的。注意: 因为要链接 CRT0 (C 程序执行时启动代码, 包括一些初始化的操作), 要添加一些特殊处理的代码, 所以生成了 “.eh_fram”、“.idata”、“.CRT” 和 “.tls section”。

表 36-1 Hello 的内存映像

Section	开始地址	内存空间	对应程序的内容
.text	0x00401000	Text	程序代码
.data	0x00409000	Data	int gInitVardata=0x12345678;
.rdata	0x0040a000	Rodata	"Hello, World\n" "%d %d\n"
.eh_fram	0x0040b000	Rodata	
.bss	0x0040d000	BSS	int gNoInitVardata;
.idata	0x0040e000	Data	包含所有外部函数地址, 用于调用动态链接库
.CRT	0x0040f000	Data	
.tls	0x00410000	Data	

如果你想更加透彻地了解程序装载和运行的机制, 建议阅读《程序员的自我修养》这本书。

36.3 简单的 malloc.c

在作者参与设计的一个嵌入式系统中, 需要使用 malloc()和 free()动态分配内存。但是因为系统中的内存有限, 在使用 newlib 库提供的 malloc()和 free()时, 会出现这种情况: 还有足够的内存空间, 但是形成了多个碎片, 导致有内存却不能分配。所以就根据实际情况, 设计了一个很简单的 malloc.c, 以实现 malloc()和 free(), 实际运行起来很稳定, 恰恰满足“刚刚

好”的原则。

在这个系统中，可以用于动态分配的区域是位于 BSS 和 Stack 之间的 SRAM 空间。在第一次申请时，直接就把这块区域交给 malloc()和 free()管理。因为函数调用时临时变量是在 Stack 空间内分配的，所以这里为 Stack 预留了 2KB 的空间用于临时变量。

例子：简单的 malloc.c

```
#define MEM_LINK_SIZE      sizeof(struct mem_link)
#define POINTER_TO_INTEGER unsigned int

struct mem_link {
    struct mem_link *next;
    unsigned short flag;
    unsigned short length;
};
struct mem_link *mem_head = NULL;

int get_useful_memory (unsigned int end_addr)
{
    unsigned int sp_addr, total;
    //Note: sp_addr is the current pointer of STACK.
    //      2048B is saved for temporary variables.
    asm volatile ("move %0, $sp":"=r"(sp_addr));
    total = sp_addr - end_addr - 2048;
    return total;
}

void init_mem_link (void)
{
    //Add some gap (16B)。Note: &_end is the end address of BSS.
    mem_head = (struct mem_link *) ((unsigned int)&_end + 16);
    mem_head->next = NULL;
    mem_head->flag = 0;
    mem_head->length = get_useful_memory ((unsigned int)mem_head) -
MEM_LINK_SIZE;
}

void *malloc (unsigned int size)
{
    struct mem_link *p;
    struct mem_link *q;

    if (mem_head == 0)
        init_mem_link ();

    //Adjust the allocation size
    size = (size + 3 ) / 4 * 4;

    for (p = mem_head; p != NULL; p = p->next)
        if (p->flag == 0 && p->length >= size)
            break;

    if (p == NULL)
        return NULL;

    //Split this block
    if (p->length >= size + 32) {
        q = (struct mem_link *)((POINTER_TO_INTEGER)p + MEM_LINK_SIZE + size);
        q->next = p->next;
        q->flag = 0;
    }
}
```

```

        q->length = p->length - MEM_LINK_SIZE - size;
        p->next = q;
        p->length = size;
    }

    p->flag = 1;
    return (void *)((POINTER_TO_INTEGER)p + MEM_LINK_SIZE);
}

void *calloc (unsigned int COUNT, unsigned int ELTSIZE)
{
    int *p;
    int size, i;
    size = (COUNT * ELTSIZE + 3) / 4 * 4;
    p = malloc (size);
    for (i = 0; i < size / 4; i++)
        p[i] = 0;
    return p;
}

void free (void *x)
{
    struct mem_link *p;
    struct mem_link *q;
    int found;

    //Change the block to IDLE
    p = (struct mem_link *)((POINTER_TO_INTEGER)x - MEM_LINK_SIZE);
    p->flag = 0;

    //Combine the continuous IDLE blocks
    while (1) {
        found = 0;
        for (p = mem_head; p != NULL; p = p->next) {
            q = p->next;
            if (q != NULL && p->flag == 0 && q->flag == 0) {
                found = 1;
                break;
            }
        }

        if (found) {
            p->next = q->next;
            p->length += q->length + MEM_LINK_SIZE;
        }
        else {
            //There is no continuous IDLE blocks.
            break;
        }
    }
}

```

注意：这里没有实现 `realloc()`，因为系统中没用它分配内存。

汇编语言是低级语言，使用助记符代替机器指令的操作码，使用地址符号或标号代替指令或操作数的地址。汇编语言不是结构化的语言，用它编写出来的程序就像面条一样，有着纠缠不清的跳转、调用和标号，难以阅读、维护和改进。

37.1 如非必要

如果没有必要，就不要使用汇编语言编写代码，因为如下原因。

- 1) 使用汇编语言编写出代码的可读性、可维护性都很差，几十行汇编代码就可能很复杂，而且并不是每个人都能读懂汇编语言。
- 2) 使用汇编语言编写的程序在不同平台之间不可移植，因为它们有不同的指令集。
- 3) C 语言编译器对 C 程序的优化通常都做得很好，完全可以生成与汇编语言相媲美的代码。
- 4) 现在有的 CPU 编译器已经做到支持完全的 C 语言设计，几乎不用了解汇编语言，就可以设计出一个完整的嵌入式系统，例如 Cortex-M3。

当然，因为编译器并不是做得那么完美，不能做得那么大而全，有些地方还是需要使用汇编语言编写代码。

- 1) 中断向量表 (Interrupt Vector Table) 和中断处理代码 (Interrupt Handler)，但是在中断处理代码中可以调用 C 语言编写的函数，以简化中断处理的编写。
- 2) 每种 CPU 都有一些特殊的指令，编译器并不能生成这些指令。
- 3) 对执行性能要求极高的地方，需要手工仔细地编写汇编代码。

如要用汇编语言编写代码，也可以寻找一些更灵巧的方法，从使用方便到使用复杂依次如下。

- 1) 有些编译器已经把特殊的指令做成可以直接“inline”的内建函数 (Built-in Function)，使用起来很方便。例如，某 CPU 就提供了这样的一组函数：__cpu_mfsr()、__cpu_mtsr()、__cpu_isb()、__cpu_dsb()、__cpu_setgie_en()、__cpu_setgie_dis()等。
- 2) 使用嵌入式汇编。例如，GCC 的内嵌汇编更加灵活，可以传递变量，免去寄存器分配的麻烦，在下一章有 GCC 内嵌汇编的介绍。
- 3) 如果要编写纯粹的汇编代码，就要充分发挥汇编器的特性，简化代码的编写，提高代码的可读性。现在的汇编器做得也很灵活，支持伪指令 (把多条机器指令用一条伪指令表示)，可以使用#define 定义的常量，也可以定义代码宏 (macro)。

37.2 熟悉 ABI

ABI (Application Binary Interface, 应用程序二进制接口) 描述了应用程序和操作系统之间、应用程序和库之间、应用的组成部分之间的接口。ABI 涵盖如下细节。

- 1) 数据类型的大小、布局和对齐。
- 2) 函数调用约定, 控制着函数调用如何传递参数、如何接受返回值。例如, 是所有的参数都通过栈传递, 还是部分参数通过寄存器传递? 哪个寄存器对应哪个函数参数? 通过栈传递的第一个函数参数是最先“push”到栈上还是最后?
- 3) 系统调用的编码和应用程序如何向操作系统进行系统调用。
- 4) 目标文件的二进制格式、程序库等。

ABI 不同于 API, API 定义了源代码和运行库之间的接口, 同样的代码可以在支持这个 API 的任何系统中编译, 而 ABI 则允许编译好的目标代码在使用兼容 ABI 的系统中无须改动就能运行。ABI 掩盖了各种细节, 一个完整的 ABI 允许在支持它的操作系统上的程序不经修改, 就可以在其他支持此 ABI 的操作系统上运行, 例如 Intel 的二进制兼容标准 (Intel Binary Compatibility Standard, iBCS)。

通常编写 C 语言程序, 对 ABI 不需要太多的了解, 但是如果深入研究下去, 想跟踪定位指令级的软硬件 Bug, 想要编写汇编代码, 那么就需要对 ABI 有深刻的了解。

37.3 汇编例子

作者在参与某嵌入式系统设计过程中, 原来的系统有 1600 行的汇编代码, 包括中断向量表、初始执行代码、中断处理代码、性能关键的函数。通过仔细研究, 把中断处理代码和某些函数改成用 C 语言实现, 提高了代码可读性, 同时还没有降低性能, 最后只保留了 200 行必要的汇编代码。下面就是这个系统的中断向量表和初始执行代码。

例子: 中断向量表和初始执行代码

```
.macro int_vector_body number
    .align 4
    li    $r0, #\number
    la    $r1, _common_int_handler
    jr    $r1
.endm

.section .cpu_init, "ax"
.align 4
interrupt_vector:
    .align 4
    li    $r1, _begin_boot
    jr    $r1
    nop
    int_vector_body #0
    int_vector_body #1
    int_vector_body #2
    int_vector_body #3
    int_vector_body #4
    int_vector_body #5
    int_vector_body #6
    int_vector_body #7
```

```
int_vector_body #8
int_vector_body #9
int_vector_body #10
int_vector_body #11
int_vector_body #12
int_vector_body #13
int_vector_body #14
int_vector_body #15

.func _begin_boot
.type _begin_boot, @function
_begin_boot:
mfsr    $r0, $sys_itype_reg
andi   $r0, $r0, 0xf
addi   $r0, $r0, #-2
beqz   $r0, nmi_handler    //Check whether it is NMI
li     $r0, 0xF4007        //Set IVB vector size: 16 bytes
mtsr   $r0, $sys_ivb_reg
isb
la     $sp, _STACK_TOP    //Set SP for stack space top
la     $gp, _SDA_BASE_    //Set GP for small data access
la     $r0, _stack
dsb
la     $r0, _start
jr     $r0
nop
.end
```

这里的 `la` 就是伪指令，等价于两条机器指令：`sethi $rx,hi20(var)`和 `ori $rx,$rx,lo12(var)`。这里的 `li` 也是一条伪指令，类似于 `la`。

这里使用了宏（`macro ... endm`），通过它简化了中断向量表的编写，看着更加规整。中断向量表只是提供中断发生时的入口，通过它跳转到 `_common_int_handler`，然后再由它调用 C 语言编写的中断处理函数。

第 38 章

GCC 特色

GCC (GNU Compiler Collection), 是由 GNU 开发的编译器套件。GCC 是以 GPL 许可证发行的自由软件, 也是 GNU 计划的关键部分。GCC 原本作为 GNU 操作系统的官方编译器, 现已被大多数类 Unix 操作系统 (如 Linux、BSD、Mac OS X 等) 采纳为标准的编译器, GCC 同样适用于微软的 Windows。

GCC 的原名为 GNU C Compiler, 因为它原来只能处理 C 语言。但是 GCC 发展得很快, 可以处理 C++, 后来又经扩展支持更多编程语言, 例如 Fortran、Pascal、Objective-C、Java、Ada, 所以改名为 GNU Compiler Collection。

38.1 MinGW

如果你是在 Linux 下工作, 那么你可以直接使用 GCC 编译器, 如果你是在 Windows 下工作, 那么你可以安装 Cygwin 或者 MinGW。笔者以前经常使用 Cygwin, 最近在自己机器上安装时, 发现安装包太大, 浪费太多的时间, 而且笔者也不需要那么多东西, 所以这次笔者就选择安装了 MinGW。

38.1.1 简单介绍

MinGW 是 Minimalist GNU on Windows 的简称, 是用自由软件生成的在 Windows 下运行的编译环境。

MinGW 并不是一个单纯的 C/C++ 编译器, 而是一套 GNU 工具集合。除了 GCC, MinGW 还包含其他的开发工具, 例如 gawk、bison、make 等。所以, MinGW 是为那些使用 Windows 的人提供一套符合 GNU 的工作环境, 我们可以像在 Linux 下一样使用 GNU 程序开发工具。

GCC 是 MinGW 的核心所在, GCC 是一套支持众多计算机程序语言的编译系统, 而且在语言标准的实现上是最接近于标准的, 并且 GCC 几乎可以移植到目前所有可用的计算机平台。

GCC 本身并没有拥有 IDE 界面, 你可以选用你喜欢的文本编辑器来编辑你的源代码, 然后使用 make 等工具进行软件项目的编译、链接、打包、发布。

38.1.2 下载安装

首先, 你要下载 MinGW, 可以从 Sourceforge (<http://sourceforge.net/projects/mingw/files/>) 下载。

其次, 你要配置环境变量, 要根据实际的安装目录进行调整, 在“系统属性→高级→环

境变量→系统变量”设置如下内容。

1) 在 PATH 的值中加入 “C:\MinGW\bin”，这是寻找 GCC 编译器的路径。如果 PATH 中还有其他内容，需要使用英文状态下的分号加以分割。

2) 新建 LIBRARY_PATH 变量，在其值中加入 “C:\MinGW\lib”，这是标准库存放的路径。

3) 新建 C_INCLUDE_PATH 变量，在其值中加入 “C:\MinGW\include”，这是 include 查找头文件的路径。

然后，你要检查安装是否正确，你可以在 cmd 控制台里，输入 “gcc -v”。如果已经成功安装，就会显示 gcc 的版本信息。

38.2 执行过程

虽然我们称 GCC 是 C 语言的编译器，但是由 C 语言文件生成可执行文件的过程不只是编译，而是要经过 4 个相互关联的步骤：预处理(Preprocess)、编译(Compile)、汇编(Assembly)和链接(Link)。下面就是这 4 个步骤的解释。

1) 预处理：gcc 调用 cpp 进行预处理，对源文件中的包含头文件、宏定义、条件编译等进行分析，生成 “.i” 为后缀的文件。

2) 编译：gcc 调用 cc1 进行编译，处理 “.i” 文件，生成 “.s” 为后缀的汇编文件，可以使用一些优化选项，例如 -O2、-fomit-frame-pointer 等。

3) 汇编：gcc 调用 as 进行汇编，处理 “.s” 或 “.S” 文件，生成 “.o” 为后缀的目标文件，“.S” 文件是手工编写的汇编文件。

4) 链接：gcc 调用 ld 进行链接，所有的目标文件被安排在可执行程序中的恰当位置，同时，所调用到的库函数也从各自所在的库中取出并链接到合适的位置。

例子：gcc 编译 hello.c

```
#include <stdio.h>
int main (void)
{
    printf("Hello, World\n");
    return 0;
}
```

预编译：gcc -E -P hello.c -o hello.i

编译：gcc -S hello.i -o hello.s

汇编：gcc -O2 -c hello.s -o hello.o

链接：gcc -o hello.exe hello.o

执行：./hello.exe

你也可以把上面这 4 个步骤简化为 1 步，直接生成可执行文件

```
gcc -O2 -o hello.exe hello.c
```

上面是最简单的执行过程，gcc 有 100 多个的编译选项可用，具体使用时请查看 gcc 手册。如果要对程序跟踪调试，在编译时就要带上 -g ~ -g3 选项，然后使用 gdb 进行跟踪调试。

38.3 内嵌汇编

GCC 支持的内嵌汇编功能非常强大，可以把表达式作为汇编指令的操作数，不用关心寄

寄存器是如何使用的，不用关心计算结果是如何传回变量的，只要写出表达式与汇编指令操作数之间的对应关系，GCC 就会自动地插入代码从而完成必要的操作。

内嵌汇编从语法上分为 4 部分：

```
__asm__ __volatile__ (instruction template : output operand list
                      : input operand list [: clobber list]);
```

内嵌汇编的语法详细解释如下（下面的例子中都是使用的 i386 汇编）。

- 1) `__asm__` 简写为 `asm`，表示后面的代码为内嵌汇编；`__volatile__` 简写为 `volatile`，用于告诉编译器，严禁将此处的汇编语句与其他的语句做重组优化。
- 2) 每部分使用 ":" 隔开，指令模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，就要用 ":" 隔开，相应部分内容为空。例如：`__asm__ __volatile__ ("cli" ::: "memory")`。
- 3) 指令模板 (instruction template)：由多条汇编语句序列组成，语句之间使用 ";"、"\n" 或 "\n\t" 分开。操作数使用占位符来对应表达式或变量，操作数占位符最多 10 个，名称如下：`%0`、`%1`、...、`%9`。
- 4) 输出操作数部分 (output operand list)：用来描述输出操作数，不同的操作数描述符之间用逗号隔开，每个操作数描述符由限定字符串和变量组成。每个输出操作数的限定字符串必须包含 "="，以表示它是一个输出操作数。描述符字符串表示对该变量的限定条件，GCC 根据这些条件决定如何分配寄存器，如何产生必要的代码，以处理指令操作数与变量之间的联系。例如：`__asm__ __volatile__ ("pushfl ; popl %0 ; cli" : "=g" (x))`。
- 5) 输入操作数部分 (input operand list)：用来描述输入操作数，不同的操作数描述符之间使用逗号隔开，每个操作数描述符由限定字符串和表达式或变量组成。例如，"`__asm__ __volatile__ ("lidt %0" : "m" (real_mode_idt));`"。
- 6) 破坏部分 (clobber list)：用于通知编译器我们使用了哪些寄存器或内存，由逗号分隔开的字符串组成，每个字符串描述一种情况，一般是寄存器名，或者使用 "memory"。如果这里写的是一些寄存器，那么这些寄存器就不要在输入或输出部分使用。
- 7) 限定字符：用来指示编译器如何处理表达式与指令操作数之间的关系，通常有很多种，而且是与 CPU 密切相关的，这里就不列举了。

使用带有操作数的内嵌汇编，就要写汇编指令模板，然后把表达式与指令的操作数关联起来，同时说明 GCC 对这些操作有哪些限定条件。

例子：带有操作数的内嵌汇编

```
__asm__ __volatile__ ("movl %1,%0" : "=r" (result) : "r" (input));
```

下面是这条内嵌汇编指令的解释说明：

- 1) "`movl %1,%0`" 是指令模板，`%0` 和 `%1` 代表指令的操作数，称为占位符，内嵌汇编靠它们把表达式与指令的操作数关联起来。
- 2) 指令模板后面用小括号括起来的是表达式，本例中有两个：`result` 和 `input`，它们按照出现的顺序分别与指令操作数 `%0`、`%1` 对应。
- 3) `result` 前面的限定字符串是 "`=r`"，其中 "=" 表示 `result` 是输出操作数，"`r`" 表示需要将 `result` 与某个通用寄存器相关联，先将操作数的值读入寄存器，然后在指令中使用相

应寄存器，而不是 `result` 本身，当指令执行完后再把寄存器中的值存入 `result`。`input` 前面的“r”表示该表达式需要先放入某个寄存器，然后在指令中使用该寄存器参加运算。

- 4) C 表达式或者变量与寄存器的关系由 GCC 自动处理，我们只需使用限定字符串指导 GCC 如何处理即可。限定字符必须与指令对操作数的要求相匹配，否则产生的汇编代码将会有错，读者可以将上例中的两个“r”，都改为“m”（m 表示操作数放在内存，而不是寄存器中），编译后得到的结果是：`movl input, result`。很明显这是一条非法指令，因此限定字符串必须与指令对操作数的要求相匹配。例如指令 `movl` 允许寄存器到寄存器，立即数到寄存器等操作，但是不允许内存到内存的操作，因此两个操作数不能同时使用“m”作为限定字符。

对于各种 CPU，例如 i386、ARM、AVR、MIPS，GCC 都有相应的内嵌汇编支持，但是都有各自的限定字符，请根据需要查看 GCC 的手册。

38.4 __attribute__

GCC 的一大特色就是“__attribute__”机制，“__attribute__”可以用来设置函数、变量和类型的属性。通过“__attribute__”机制，可以设置很多属性，下面介绍的只是一小部分。

__attribute__ 语法格式为：`__attribute__((attribute-list))`。例如，函数声明中的 `__attribute__((noreturn))`，就是告诉编译器这个函数不会返回给调用者，以便编译器在优化时去掉不必要的函数返回代码。

38.4.1 section

gcc 编译后的二进制文件为 ELF 格式，函数会默认地链接到 ELF 文件的 `text section` 中，变量则会链接到 `bss` 和 `data section` 中。如果想把函数或变量放到特定的 `section` 中，就可以使用 `section` 属性。在嵌入式系统中，经常要用到 `section` 属性，以便于把函数或变量安排到合适的 `section`。

例子：使用 `section` 属性

```
int __attribute__((section("TEST"))) test_func1 (int a, int b)
{
    return a + b;
}
```

34.4.2 always_inline

gcc 有个 `inline` 关键字，可以用它定义内联函数。编译器会把 `inline` 函数在调用处展开，以减少函数调用的开销，加快代码的执行速度。但是 gcc 会根据代码的大小，决定是否展开 `inline` 函数，所以就有可能发现 `inline` 函数没有被展开的情况。但是，通过使用 `always_inline` 属性，可以强制对 `inline` 函数展开。

例子：使用 `always_inline` 属性

```
inline void __attribute__((always_inline)) test_func2 (int a, int b)
{
    return a + b;
}
```

}

38.4.3 packed

`packed` 属性用于修饰 `struct`，告诉编译器取消 `struct` 的元素在编译过程中的优化对齐，按照实际占用字节数进行对齐。

例子：使用 `packed` 属性

```
#include <stdio.h>
struct test0 { char a; int b;};
struct test1 { char a; int b;} __attribute__((packed));
int main (void)
{
    printf("%d, %d\n", sizeof(struct test0), sizeof(struct test1));
    return 0;
}
```

`struct test0` 照理来说应该有 $1+4=5$ 字节的大小，但是 `gcc` 编译出来的大小是 8，也就是说 `char a` 后面有 3 字节的空洞，以便让 `int b` 对齐在 4 字节上。但是当加上 `packed` 修饰后，就会按照实际的类型大小来计算，所以 `struct test1` 的大小就是 5。

38.4.4 aligned

`aligned` 属性用于修饰变量、类型和函数的对齐位置

例子：使用 `aligned` 属性

```
int fruit __attribute__((aligned(8))) = 0;
struct test3 {int a;} __attribute__((aligned(8)));
void __attribute__((aligned(2))) test_func2 (void)
{
    printf("hello, world.\n");
}
```

参考文献

- [1] Cliff Cummings. Papers. <http://www.sunburst-design.com>
- [2] 百度百科. <http://baike.baidu.com>.
- [3] 浅析计算机软件可维护性方法. 百度文库.
- [4] 汪曾祺. 汪曾祺文集. 江苏文艺出版社, 1994.
- [5] 山下英子著,吴倩译. 断舍离. 广西科学技术出版社,2013.
- [6] Sriranga Veeraraghav 著,前导译. UNIX Shell 编程 24 学时教程. 机械工业出版社,1999.
- [7] J. Bhasker 著,夏宇闻、甘伟译. Verilog HDL 入门. 北京航空航天大学出版社,2008.
- [8] 魏家明. Verilog 编程艺术. 电子工业出版社,2014.
- [9] 赫伯特·希尔特著,王子恢等译. C 语言大全 (第四版). 电子工业出版社,2001.
- [10] Andrew Koenig 著,高巍译. C 陷阱与缺陷. 人民邮电出版社,2008.
- [11] Kenneth A. Reek 著,徐波译. C 和指针. 人民邮电出版社,2008.
- [12] Peter Van Der Linden 著,徐波译. C 专家编程. 人民邮电出版社,2008.
- [13] K. N. King 著,吕秀锋、黄倩译. C 语言程序设计现代方法. 人民邮电出版社,2010.
- [14] 梁肇新. 编程高手箴言. 电子工业出版社,2003.
- [15] 俞甲子,石凡,潘爱民. 程序员的自我修养. 电子工业出版社,2009.
- [16] Andrew Hunt,David Thomas 著,马维达译. 程序员修炼之道,电子工业出版社,2011.
- [17] Joshua Kerievsky 著,杨光、刘基诚译. 重构与模式. 人民邮电出版社,2013.
- [18] Robert C. Martin 著,韩磊译. 代码整洁之道. 人民邮电出版社,2010.
- [19] Steve McConnell 著,金戈、汤凌等译. 代码大全 (第二版). 电子工业出版社,2006.
- [20] Dustin Boswell 等著,尹哲、郑秀文译. 编写可读代码的艺术. 机械工业出版社,2012.
- [21] Jon Bentley 著,黄倩、钱丽艳译. 编程珠玑 (第二版). 人民邮电出版社,2015.
- [22] Brian W.Kernighan 等著,高博、徐章宁译. 编程格调. 人民邮电出版社,2015.

《代码结构》

读者调查表

尊敬的读者：

欢迎您参加读者调查活动，请对我们的图书提出真诚的意见，您的建议将是我们创造精品的动力源泉。

1. 您可以登录 <http://yydz.phei.com.cn>，进入“客户留言”栏目，或者直接发送邮件给本书编辑，将您对本书的意见和建议反馈给我们。

2. 您可以填写下表后寄给我们。

姓名：_____ 性别： 男 女 年龄：_____ 职业：_____

电话：_____ E-mail：_____

通信地址：_____ 邮编：_____

1. 影响您购买本书的因素（可多选）：

- 封面封底 价格 内容简介、前言和目录 书评广告 出版物名声
作者名声 正文内容 其他 _____

2. 您对本书的满意度：

- 从技术角度 很满意 比较满意 一般 较不满意 不满意
从文字角度 很满意 比较满意 一般 较不满意 不满意
从排版、封面设计角度 很满意 比较满意 一般 较不满意
不满意

3. 您最喜欢书中的哪篇（或章、节）？请说明理由。

4. 您最不喜欢书中的哪篇（或章、节）？请说明理由。

5. 您希望本书在哪些方面进行改进？

6. 您感兴趣或希望增加的图书选题有：

邮寄地址：北京市海淀区万寿路 173 信箱电子信息出版分社 牛平月 收 邮编：100036

编辑电话：(010) 88254454 E-mail: niupy@phei.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为，歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396; (010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市海淀区万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036