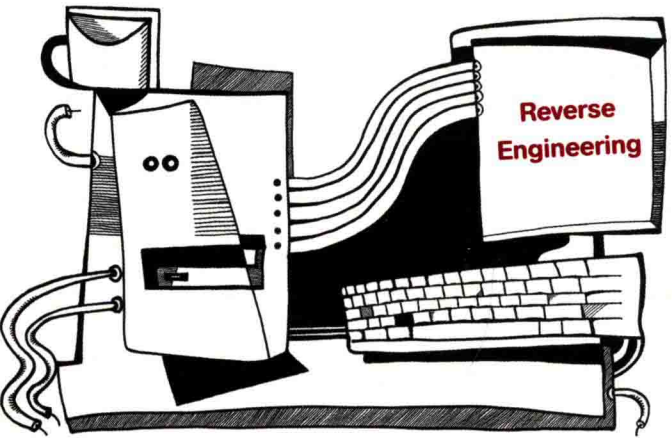


安天 ANTIY

异步图书 www.epubit.com.cn

- 了解逆向工程的权威指南
- 初学者必备的大百科全书
- 安天网络安全工程师培训必读书目

Reverse Engineering for Beginners

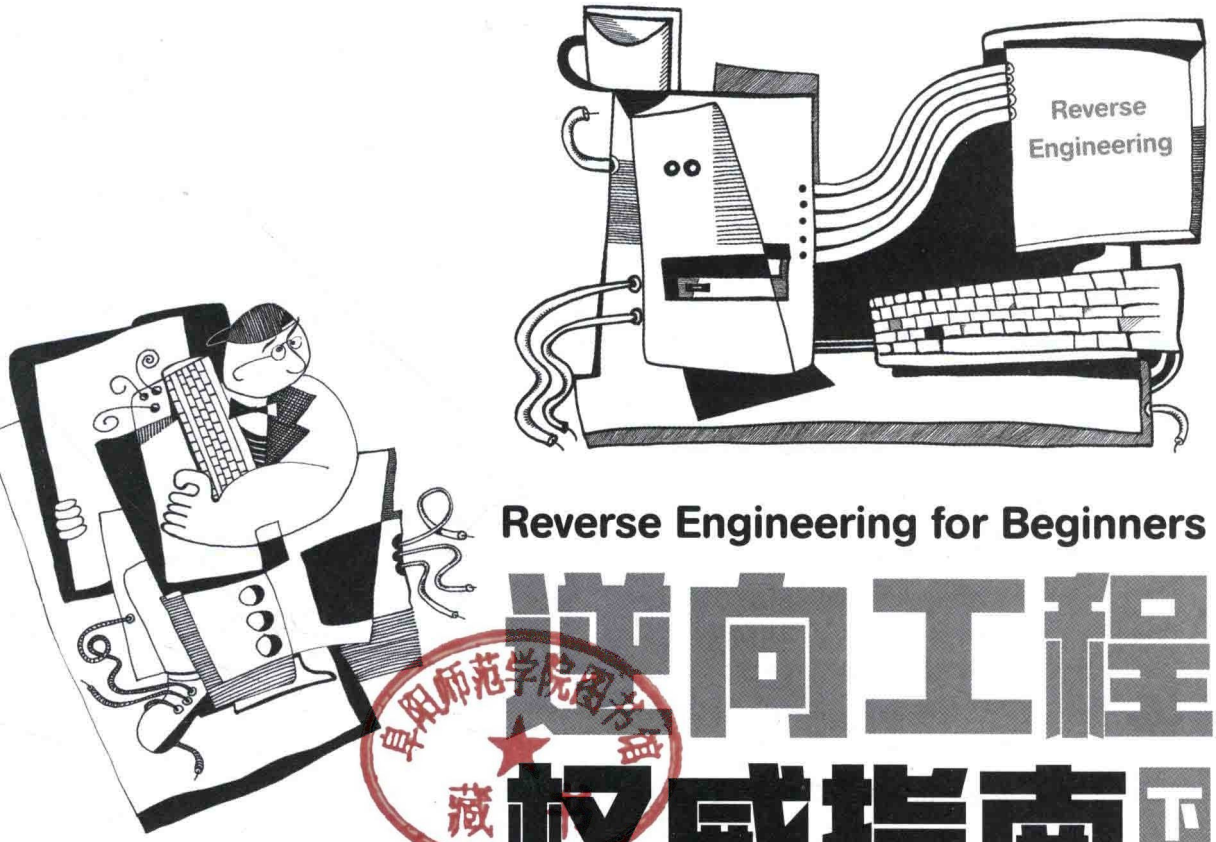
逆向工程权威指南 下册

[乌克兰] Dennis Yurichev◎著

Archer 安天安全研究与应急处理中心◎译

中国工信出版集团

人民邮电出版社 POSTS & TELECOM PRESS



Reverse Engineering for Beginners

逆向工程权威指南（下册）

[乌克兰] Dennis Yurichev◎著
Archer 安天安全研究与应急处理中心◎译

人民邮电出版社
北京

目　录

第 47 章　字符串剪切……485

47.1　x64 下的 MSVC 2013 优化……486
47.2　x64 下采用编辑器 GCC 4.9.1 进行非优化操作……487
47.3　x64 下的 GCC 4.9.1 优化……488
47.4　ARM64：非优化的 GCC（Linaro）4.9……489
47.5　ARM64:优化 GCC（Linaro）4.9……490
47.6　ARM: Keil 6/2013 优化（ARM 模式）……491
47.7　ARM:Keil 6/2013（Thumb 模式）优化……492
47.8　MIPS……493

第 48 章　toupper()函数……495

48.1　x64……495
48.1.1　两个比较操作……495
48.1.2　一个比较操作……496
48.2　ARM……497
48.2.1　ARM64 下的 GCC……497
48.3　总结……498

第 49 章　不正确的反汇编代码……499

49.1　x86 环境下的从一开始错误的反汇编……499
49.2　随机噪音，怎么看起来像反汇编指令？……500

第 50 章　混淆技术……505

50.1　字符串变换……505
50.2　可执行代码……506
50.2.1　插入垃圾代码……506
50.2.2　用多个指令组合代替原来的一个指令……506
50.2.3　始终执行或者从来不会执行的代码……506
50.2.4　把指令序列搞乱……507
50.2.5　使用间接指针……507
50.3　虚拟机以及伪代码……507
50.4　一些其他的事情……507
50.5　练习题……508
50.5.1　练习 1……508

第 51 章　C++……509

51.1　类……509
51.1.1　一个简单的例子……509
51.1.2　类继承……515
51.1.3　封装……518
51.1.4　多重继承……520
51.1.5　虚拟方法……523
51.2　ostream 输出流……526
51.3　引用……527
51.4　STL/标准模板库（Standard Template Library）……528
51.4.1　std::string（字符串）……528
51.4.2　std::list……535
51.4.3　std::vector 标准向量……545
51.4.4　std::map()和 std::set()……552

第 52 章　数组与负数索引……563

第 53 章　16 位的 Windows 程序……566

53.1　例子#1……566
53.2　例子#2……566
53.3　例子#3……567
53.4　例子#4……568
53.5　例子#5……571
53.6　例子#6……574
53.6.1　全局变量……576

第四部分　Java

第 54 章　Java……581

54.1　简介……581
54.2　返回一个值……581
54.3　简单的计算函数……586
54.4　JVM 的内存模型……588
54.5　简单的函数调用……588
54.6　调用函数 beep()（蜂鸣器）……590
54.7　线性同余随机数产生器（PRNG）…591
54.8　条件转移……592
54.9　传递参数……594
54.10　位操作……595

54.11 循环……596
54.12 switch()语句……598
54.13 数组……599
54.13.1 简单的例子……599
54.13.2 数组元素求和……601
54.13.3 输入变量为数组的主函数 main()……601
54.13.4 预设初始值的的数组……602
54.13.5 可变参数函数……604
54.13.6 二维数组……606
54.13.7 三维数组……606
54.13.8 小结……607
54.14 字符串……607
54.14.1 第一个例子……607
54.14.2 第二个例子……608
54.15 异常处理……609
54.16 类……612
54.17 简单的补丁……614
54.17.1 第一个例子……614
54.17.2 第二个例子……616
54.18 总结……618

第五部分 在代码中发现重要而有趣的内容

第 55 章 编译器产生的文件特征……621

55.1 Microsoft Visual C++……621
55.1.1 命名规则……621
55.2 GCC 编译器……621
55.2.1 命名规则……621
55.2.2 Cygwin……621
55.2.3 MinGW……621
55.3 Intel FORTRAN……621
55.4 Watcom 以及 OpenWatcom……622
55.4.1 命名规则……622
55.5 Borland 编译器……622
55.5.1 Dephi 编程语言……622
55.6 其他的已知 DLL 文件……623

第 56 章 Win32 环境下与外部通信……624

56.1 在 Windows API 中最经常使用的函数……624
56.2 tracer:解析指定模块的所有函数……624

第 57 章 字符串……626

57.1 字符串……626
57.1.1 C/C++中的字符串……626
57.1.2 Borland Delphi……626
57.1.3 Unicode 编码……626
57.1.4 Base64……628
57.2 错误/调试信息……629
57.3 可疑的魔数字符串……629

第 58 章 调用宏 assert()（中文称为断言）……630

第 59 章 常数……631

59.1 魔数……631
59.1.1 动态主机配置协议（Dynamic Host Configuration Protocol，DHCP）……632
59.2 寻找常数……632

第 60 章 检索关键指令……633

第 61 章 可疑的代码模型……635

61.1 XOR 异或指令……635
61.2 手写汇编代码……635

第 62 章 魔数与程序调试……637

第 63 章 其他的事情……638

63.1 总则……638
63.2 C++……638
63.3 部分二进制文件的特征……638
63.4 内存“快照”对比……638
63.4.1 Windows 注册表……639
63.4.2 瞬变比较器 Blink-comparator……639

第六部分 操作系统相关

第 64 章 参数的传递方法（调用规范）……643

64.1 cdecl [C Declaration 的缩写]……643
64.2 stdcall [Standard Call 的缩写]……643
64.2.1 带有可变参数的函数……644
64.3 fastcall……644
64.3.1 GCC regparm……645
64.3.2 Watcom/OpenWatcom……645
64.4 thiscall……645
64.5 64 位下的 x86……646
64.5.1 Windows x64……646
64.5.2 64 位下的 Linux……648

64.6 单/双精度数型返回值……649
64.7 修改参数……649
64.8 指针型函数参数……649

第 65 章 线程本地存储 TLS……652

65.1 线性同余发生器（改）……652
65.1.1 Win32 系统……652
65.1.2 Linux 系统……656

第 66 章 系统调用（syscall-s）……658

66.1 Linux……658
66.2 Windows……659

第 67 章 Linux……660

67.1 位置无关的代码……660
67.1.1 Windows……662
67.2 在 Linux 下的 LD_PRELOAD……662

第 68 章 Windows NT……666

68.1 CRT (Win32)……666
68.2 Win32 PE 文件……669
68.2.1 术语……670
68.2.2 基地址……670
68.2.3 子系统……670
68.2.4 操作系统版本……670
68.2.5 段……671
68.2.6 重定向段 Relocations(relocs)……672
68.2.7 导出段和导入段……672
68.2.8 资源段……674
68.2.9 .NET……675
68.2.10 TLS 段……675
68.2.11 工具……675
68.2.12 更进一步……675
68.3 Windows SEH……675
68.3.1 让我们暂时把 MSVC 放在一边……675
68.3.2 让我们重新回到 MSVC……680
68.3.3 Windows x64……693
68.3.4 关于 SEH 的更多信息……697
68.4 Windows NT：临界区段……697

第七部分 常用工具

第 69 章 反汇编工具……703

69.1 IDA……703

第 70 章 调试工具……704

70.1 tracer……704
70.2 OllyDbg……704
70.3 GDB……704

第 71 章 系统调用的跟踪工具……705

71.1 strace/dtruss……705

第 72 章 反编译工具……706

第 73 章 其他工具……707

第八部分 更多范例

第 74 章 修改任务管理器（Vista）……711

74.1 使用 LEA 指令赋值……713

第 75 章 修改彩球游戏……715

第 76 章 扫雷（Windows XP）……717

76.1 练习题……721

第 77 章 人工反编译与 Z3 SMT 求解法……722

77.1 人工反编译……722
77.2 Z3 SMT 求解法……725

第 78 章 加密狗……730

78.1 例 1：PowerPC 平台的 MacOS Classic 程序……730
78.2 例 2: SCO OpenServer……737
解密错误信息……745
78.3 例 3: MS-DOS……747

第 79 章 “QR9”：魔方态加密模型……753

第 80 章 SAP……782

80.1 关闭客户端的网络数据包压缩功能……782
80.2 SAP 6.0 的密码验证函数……793

第 81 章 Oracle RDBMS……797

81.1 V$VERSION 表……797
81.2 X$KSMLRU 表……805
81.3 V$TIMER 表……806

第 82 章 汇编指令与屏显字符……811
82.1 EICAR……811
第 83 章 实例演示……813
83.1 10PRINT CHR$(205.5+RND(1));: GOTO 10……813
83.1.1 Trixter 的 42 字节程序……813
83.1.2 笔者对 Trixter 算法的改进：27 字节……814
83.1.3 从随机地址读取随机数……814
83.1.4 其他……815
83.2 曼德博集合……815
83.2.1 理论……816
83.2.2 demo 程序……820
83.2.3 笔者的改进版……822
第九部分 文件分析
第 84 章 基于 XOR 的文件加密……827
84.1 Norton Guide：单字节 XOR 加密实例……827
信息熵……828
84.2 4 字节 XOR 加密实例……828
84.3 练习题……830
第 85 章 Millenium 游戏的存档文件……831
第 86 章 Oracle 的.SYM 文件……835
第 87 章 Oracle 的.MSDB 文件……842
总结……845
第十部分 其他
第 88 章 npad……849
第 89 章 修改可执行文件……851
89.1 文本字符串……851
89.2 x86 指令……851
第 90 章 编译器内部函数……852
第 91 章 编译器的智能短板……853
第 92 章 OpenMP……854
92.1 MSVC……856
92.2 GCC……858
第 93 章 安腾指令……860
第 94 章 8086 的寻址方式……863
第 95 章 基本块重排……864
95.1 PGO 的优化方式……864
第十一部分 推荐阅读
第 96 章 参考书籍……869
96.1 Windows……869
96.2 C/C++……869
96.3 x86/x86-64……869
96.4 ARM……869
96.5 加密学……869
第 97 章 博客……870
97.1 Windows 平台……870
第 98 章 其他内容……871
第十二部分 练习题
第 99 章 初等难度练习题……875
99.1 练习题 1.4……875
第 100 章 中等难度练习题……876
100.1 练习题 2.1……876
100.1.1 Optimizing MSVC 2010 x86……876
100.1.2 Optimizing MSVC 2012 x64……877
100.2 练习题 2.4……877
100.2.1 Optimizing MSVC 2010……877
100.2.2 GCC 4.4.1……878
100.2.3 Optimizing Keil（ARM mode）……879
100.2.4 Optimizing Keil（Thumb mode）……880
100.2.5 Optimizing GCC 4.9.1（ARM64）……880
100.2.6 Optimizing GCC 4.4.5

（MIPS）……881
100.3 练习题 2.6……882
100.3.1 Optimizing MSVC 2010……882
100.3.2 Optimizing Keil（ARM mode）……883
100.3.3 Optimizing Keil（Thumb mode）……884
100.3.4 Optimizing GCC 4.9.1（ARM64）……884
100.3.5 Optimizing GCC 4.4.5（MIPS）……885
100.4 练习题 2.13……885
100.4.1 Optimizing MSVC 2012……886
100.4.2 Keil（ARM mode）……886
100.4.3 Keil（Thumb mode）……886
100.4.4 Optimizing GCC 4.9.1（ARM64）……886
100.4.5 Optimizing GCC 4.4.5（MIPS）……887
100.5 练习题 2.14……887
100.5.1 MSVC 2012……887
100.5.2 Keil（ARM mode）……888
100.5.3 GCC 4.6.3 for Raspberry Pi（ARM mode）……888
100.5.4 Optimizing GCC 4.9.1（ARM64）……889
100.5.5 Optimizing GCC 4.4.5（MIPS）……890
100.6 练习题 2.15……891
100.6.1 Optimizing MSVC 2012 x64……892
100.6.2 Optimizing GCC 4.4.6 x64……894
100.6.3 Optimizing GCC 4.8.1 x86……895
100.6.4 Keil（ARM 模式）：面向 Cortex-R4F CPU 的代码……896
100.6.5 Optimizing GCC 4.9.1（ARM64）……897
100.6.6 Optimizing GCC 4.4.5（MIPS）……898
100.7 练习题 2.16……899
100.7.1 Optimizing MSVC 2012 x64……899
100.7.2 Optimizing Keil（ARM mode）……899
100.7.3 Optimizing Keil（Thumb mode）……900
100.7.4 Non-optimizing GCC 4.9.1（ARM64）……900
100.7.5 Optimizing GCC 4.9.1（ARM64）……901
100.7.6 Non-optimizing GCC 4.4.5（MIPS）……904
100.8 练习题 2.17……905
100.9 练习题 2.18……905
100.10 练习题 2.19……905
100.11 练习题 2.20……905

第 101 章 高难度练习题……906

101.1 练习题 3.2……906
101.2 练习题 3.3……906
101.3 练习题 3.4……906
101.4 练习题 3.5……906
101.5 练习题 3.6……907
101.6 练习题 3.8……907

第 102 章 Crackme/Keygenme……908

附录 A x86……909

A.1 数据类型……909
A.2 通用寄存器……909
A.2.1 RAX/EAX/AX/AL……909
A.2.2 RBX/EBX/BX/BL……910
A.2.3 RCX/ECX/CX/CL……910
A.2.4 RDX/EDX/DX/DL……910
A.2.5 RSI/ESI/SI/SIL……910
A.2.6 RDI/EDI/DI/DIL……910
A.2.7 R8/R8D/R8W/R8L……911
A.2.8 R9/R9D/R9W/R9L……911
A.2.9 R10/R10D/R10W/R10L……911
A.2.10 R11/R11D/R11W/R11L……911
A.2.11 R12/R12D/R12W/R12L……911
A.2.12 R13/R13D/R13W/R13L……911
A.2.13 R14/R14D/R14W/R14L……912
A.2.14 R15/R15D/R15W/R15L……912
A.2.15 RSP/ESP/SP/SPL……912
A.2.16 RBP/EBP/BP/BPL……912
A.2.17 RIP/EIP/IP……912
A.2.18 段地址寄存器 CS/DS/ES/SS/FS/GS……913
A.2.19 标识寄存器……913
A.3 FPU 寄存器……913
A.3.1 控制字寄存器（16 位）……914

A.3.2 状态字寄存器（16位）……914
A.3.3 标记字寄存器（16位）……915
A.4 SIMD寄存器……915
A.4.1 MMX寄存器……915
A.4.2 SSE与AVX寄存器……915
A.5 FPU调试寄存器……915
A.5.1 DR6规格……916
A.5.2 DR7规格……916
A.6 指令……917
A.6.1 指令前缀……917
A.6.2 常见指令……917
A.6.3 不常用的汇编指令……922
A.6.4 FPU指令……927
A.6.5 可屏显的汇编指令（32位）……928

附录B ARM……931

B.1 术语……931
B.2 版本差异……931
B.3 32位ARM（AArch32）……931
B.3.1 通用寄存器……931
B.3.2 程序状态寄存器/CPSR……931
B.3.3 VFP（浮点）和NEON寄存器……932
B.4 64位ARM（AArch64）……932
通用寄存器……932
B.5 指令……933
Conditional codes速查表……933

附录C MIPS……934

C.1 寄存器……934
C.1.1 通用寄存器GPR……934
C.1.2 浮点寄存器FPR……934
C.2 指令……934
转移指令……935

附录D 部分GCC库函数……936

附录E 部分MSVC库函数……937

附录F 速查表……938

F.1 IDA……938
F.2 OllyDbg……939
F.3 MSVC选项……939
F.4 GCC……939
F.5 GDB……940

附录G 练习题答案……941

G.1 各章练习……941
第3章……941
第5章……941
第7章……941
第13章……942
第14章……942
第15章……942
第16章……942
第17章……943
第18章……943
第19章……944
第21章……945
第41章……946
第50章……946
G.2 初级练习题……947
G.2.1 练习题1.1……947
G.2.2 练习题1.4……947
G.3 中级练习题……947
G.3.1 练习题2.1……947
G.3.2 练习题2.4……948
G.3.3 练习题2.6……949
G.3.4 练习题2.13……949
G.3.5 练习题2.14……949
G.3.6 练习题2.15……949
G.3.7 练习题2.16……950
G.3.8 练习题2.17……950
G.3.9 练习题2.18……950
G.3.10 练习题2.19……950
G.3.11 练习题2.20……950
G.4 高难度练习题……950
G.4.1 练习题3.2……950
G.4.2 练习题3.3……950
G.4.3 练习题3.4……950
G.4.4 练习题3.5……950
G.4.5 练习题3.6……951
G.4.6 练习题3.8……951
G.5 其他练习题……951
G.5.1 “扫雷（Windows XP）”……951

参考文献……952

第 47 章　字符串剪切

我们经常需要去除字符串里的开始符号或者结束符号。

本章介绍的程序，专门用于去除字符串的结尾部分的回车和换行字符 CR/LF。

```
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
        char c;
        size_t str_len;

        // work as long as \r or \n is at the end of string
        // stop if some other character there or its an empty string'
        // (at start or due to our operation)
        for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
        {
                if (c=='\r' || c=='\n')
                        s[str_len-1]=0;
                else
                        break;
        };
        return s;
};

int main()
{
        // test

        // strdup() is used to copy text string into data segment,
        // because it will crash on Linux otherwise,
        // where text strings are allocated in constant data segment,
        // and not modifiable.

        printf ("[%s]\n", str_trim (strdup("")));
        printf ("[%s]\n", str_trim (strdup("\n")));
        printf ("[%s]\n", str_trim (strdup("\r")));
        printf ("[%s]\n", str_trim (strdup("\n\r")));
        printf ("[%s]\n", str_trim (strdup("\r\n")));
        printf ("[%s]\n", str_trim (strdup("test1\r\n")));
        printf ("[%s]\n", str_trim (strdup("test2\n\r")));
        printf ("[%s]\n", str_trim (strdup("test3\n\r\n\r")));
        printf ("[%s]\n", str_trim (strdup("test4\n")));
        printf ("[%s]\n", str_trim (strdup("test5\r")));
        printf ("[%s]\n", str_trim (strdup("test6\r\r\r")));
};
```

输入参数总能正常返回并退出。当你需要对串进行批量处理时会非常方便，就像这里的 main()函数一样。

在该程序循环体的 for()语句里有两个判断条件表达式：一个条件表达式是 str_len>0（字符串的长度大于零）；另外一个是 c=s[str_len-1]（意思是取出的值不为 0、不是终止符）。循环判断语句“str_len>0 && (c=s[str_len-1])”实际上利用了所谓“逻辑短路”的执行特性，因此可以这样书写第二个判断表达式（可以参考 Yur13:p.1.3.8）。C/C++编译器自左至右的逐一检测判断条件。因为逻辑操作符是“&&”（与），所以

一旦第一个条件表达式的值为假，计算机就不用再判断（执行）第二个条件判断表达式。实际上，第二个条件表达式是一种只能在相应的条件下才可以运行的语句。笔者综合利用了第一个条件表达式、逻辑操作符“&&”的逻辑含义以及“逻辑短路”的特性，为第二个条件判断表达式限定了执行条件。

47.1 x64 下的 MSVC 2013 优化

指令清单 47.1 x64 下的 MSVC 2013 优化

```
s$ = 8
str_trim PROC

; RCX is the first function argument and it always holds pointer to the string

; this is strlen() function inlined right here:
; set RAX to 0xFFFFFFFFFFFFFFFF (-1)
        or      rax, -1
$LL14@str_trim:
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT $LL14@str_trim
; is string length zero? exit then
        test    eax, eax
$LN18@str_trim:
        je      SHORT $LN15@str_trim
; RAX holds string length
; here is probably disassembler (or assembler printing routine) error,
; LEA RDX... should be here instead of LEA EDX...
        lea     edx, DWORD PTR [rax-1]
; idle instruction: EAX will be reset at the next instructions execution'
        mov     eax, edx
; load character at address s[str_len-1]
        movzx   eax, BYTE PTR [rdx+rcx]
; save also pointer to the last character to R8
        lea     r8, QWORD PTR [rdx+rcx]
        cmp     al, 13 ; is it '\r'?
        je      SHORT $LN2@str_trim
        cmp     al, 10 ; is it '\n'?
        jne     SHORT $LN15@str_trim
$LN2@str_trim:
; store 0 to that place
        mov     BYTE PTR [r8], 0
        mov     eax, edx
; check character for 0, but conditional jump is above...
        test    edx, edx
        jmp     SHORT $LN18@str_trim
$LN15@str_trim:
; return "s"
        mov     rax, rcx
        ret     0
str_trim ENDP
```

第一个特征就是 MSVC 编译器对字符串长度函数 strlen()进行了内联（inline）式的展开和嵌入处理。编译器认为，内联处理后的执行效率会比常规的函数调用（call）的效率更高 。有关内联函数可以参考本书的第 43 章。

内嵌处理之后，strlen()函数的第一个指令是：OR RAX,0xffffffffffffffff。我们不清楚为何 MSVC 采用 OR（或）指令，而没有采用 MOV RAX, 0xffffffffffffffff 指令直接赋值。当然，这两条指令执行的是相同操作：将所有位设置为 1。因此这个数值就是赋值为−1。可以参考本书的第 30 章。

你也可能会问，为什么 strlen()函数会用到−1 这个数。当然是出于优化的目的。这里我们列出了 MSVC

的编译代码。

指令清单 47.2　x64 下的 MSVC 2013 的内嵌函数 strlen()

```
; RCX = pointer to the input string
; RAX = current string length
        or      rax, -1
label:
        inc     rax
        cmp     BYTE PTR [rcx+rax], 0
        jne     SHORT label
; RAX = string length
```

如果把这个变量的初始值设置为 0，那么是否可以把代码压缩得更短一些呢？我们可以来试试。

指令清单 47.3　我们的 strlen()字符串长度函数版本

```
; RCX = pointer to the input string
; RAX = current string length
        xor     rax, rax
label:
        cmp     byte ptr [rcx+rax], 0
        jz      exit
        inc     rax
        jmp     label
exit:
; RAX = string length
```

我们没能成功。因为我们不得不加入了一个额外的指令：JMP 跳转指令。

在使用“－1”作为初始值之后，MSVC 2013 随即在加载字符的指令之前分配了一个 INC 指令。如果第一个字符是 0（终止符），那么 RAX 就直接为 0，返回的字符串长度还会是 0。

函数中的其余部分还是很好懂的。当然在程序的最后有另外一个技巧。除去那些内联之后的 strlen()函数的展开代码，整个函数就只有 3 个条件指令。其实，从道理上讲这里应该有 4 个转移指令：第 4 个应当位于函数的结尾部分，用于检查当前字符是不是 0。然后这段代码使用了一个跳转到“$LN18@str_trim”标签的无条件转移指令，而这个标签后面的第一个指令就是条件转移指令 JE。编译器用这种指令组用来判断输入的字符串是不是空字符串（当前字符是不是终止符），而且直接就是在 strlen()执行结束后。所以这里使用 JE 指令有两个目的。这也许杀鸡用宰牛刀，但是无论如何，MSVC 就是这样做的。

要想提示程序性能，就应当尽量脱离条件转移指令进行程序作业。有关详情请参阅本书第 33 章。

47.2　x64 下采用编辑器 GCC 4.9.1 进行非优化操作

```
str_trim:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-24], rdi
; for() first part begins here
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax
        call    strlen
        mov     QWORD PTR [rbp-8], rax    ; str_len
; for() first part ends here
        jmp     .L2
; for() body begins here
.L5:
        cmp     BYTE PTR [rbp-9], 13      ; c=='\r'?
        je      .L3
        cmp     BYTE PTR [rbp-9], 10      ; c=='\n'?
        jne     .L4
.L3:
```

```
        mov     rax, QWORD PTR [rbp-8]    ; str_len
        lea     rdx, [rax-1]              ; EDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]   ; s
        add     rax, rdx                  ; RAX=s+str_len-1
        mov     BYTE PTR [rax], 0         ; s[str_len-1]=0
; for() body ends here
; for() third part begins here
        sub     QWORD PTR [rbp-8], 1      ; str_len--
; for() third part ends here
.L2:
; for() second part begins here
        cmp     QWORD PTR [rbp-8], 0      ; str_len==0?
        je      .L4                       ; exit then
; check second clause, and load "c"
        mov     rax, QWORD PTR [rbp-8]    ; RAX=str_len
        lea     rdx, [rax-1]              ; RDX=str_len-1
        mov     rax, QWORD PTR [rbp-24]   ; RAX=s
        add     rax, rdx                  ; RAX=s+str_len-1
        movzx   eax, BYTE PTR [rax]       ; AL=s[str_len-1]
        mov     BYTE PTR [rbp-9], al      ; store loaded char into "c"
        cmp     BYTE PTR [rbp-9], 0       ; is it zero?
        jne     .L5                       ; yes? exit then
; for() second part ends here
.L4:
; return "s"
        mov     rax, QWORD PTR [rbp-24]
        leave
        ret
```

笔者在程序中增加了注释。执行完长度计算函数 strlen()后，控制权将传递给标号为 L2 的语句。接着注意检查两个条件表达式。如果第一判断条件表达式为真，也就是说如果长度为 0（str_len 的值为 0），那么计算机将不再检测第二个条件判断表达式。这种特性又称为“逻辑短路”。

概括地说，这个函数的执行流程如下：

- 运行 for()语句的第一部分，也就是调用 strlen()函数的循环初始化指令。
- 跳转到标号 L2；检测循环条件是否成立。
- 跳转到标号 L5，进入循环体；
- 再执行 for()语句，如果条件不成立，则直接退出。
- 执行 for()语句的第三部分，将变量 str_len 递减。
- 再次跳转到标号 L2，检测循环条件是否成立、进入循环……周而复始，直到循环条件不成立。
- 跳转到 L4 标号，准备退出。
- 制备返回值，即变量 s。

47.3 x64 下的 GCC 4.9.1 优化

```
str_trim:
        push    rbx
        mov     rbx, rdi
; RBX will always be "s"
        call    strlen
; check for str_len==0 and exit if its so'
        test    rax, rax
        je      .L9
        lea     rdx, [rax-1]
; RDX will always contain str_len-1 value, not str_len
; so RDX is more like buffer index variable
        lea     rsi, [rbx+rdx]        ; RSI=s+str_len-1
        movzx   ecx, BYTE PTR [rsi]   ; load character
        test    cl, cl
```

```
        je      .L9                     ; exit if its zero'
        cmp     cl, 10
        je      .L4
        cmp     cl, 13                  ; exit if its not' '\n' and not '\r'
        jne     .L9
.L4:
; this is weird instruction. we need RSI=s-1 here.
; its possible to get it by' MOV RSI, EBX / DEC RSI
; but this is two instructions instead of one
        sub     rsi, rax
; RSI = s+str_len-1-str_len = s-1
; main loop begin
.L12:
        test    rdx, rdx
; store zero at address s-1+str_len-1+1 = s-1+str_len = s+str_len-1
        mov     BYTE PTR [rsi+1+rdx], 0
; check for str_len-1==0. exit if so.
        je      .L9
        sub     rdx, 1                  ; equivalent to str_len--
; load next character at address s+str_len-1
        movzx   ecx, BYTE PTR [rbx+rdx]
        test    cl, cl                  ; is it zero? exit then
        je      .L9
        cmp     cl, 10                  ; is it '\n'?
        je      .L12
        cmp     cl, 13                  ; is it '\r'?
        je      .L12
.L9:
; return "s"
        mov     rax, rbx
        pop     rbx
        ret
```

GCC 的实现方式更为复杂。在循环体执行前的代码只执行一次，而且它还会检查结束符是不是回车和换行 CR/LF。这难道不是多此一举吗？

一般来说，实现主循环体的流程是这样的：

① 循环开始，检查 CR/LF 结束符，进行判断。

② 保存零字符。

但是，GCC 编译器会将这两步逆序执行。因此，第一步肯定不会是保存零字符，而是进行下述判断：

① 看看第一个字符是不是 CR/LF,如果不是的话，就会退出。

② 循环开始，保存零字符。

③ 根据检查字符是不是 CR/LF 来决定程序的执行。

这样处理之后，主循环体就小了很多，更适用于目前的 CPU 了。这种代码的中间变量不是 str_len，而是 str_len-1。或许是因为后者更适用于用作缓冲区型数据的索引标号（数组下标）。很明显，GCC 注意到了，str_len-1 使用了两次。因此最好的办法是分配一个变量，其值总是比目前的字符串的长度小 1，然后再将其递减（按照变量 str_len 的递减方式递减）。

47.4　ARM64：非优化的 GCC（Linaro）4.9

它将生成近乎“一一对应”的汇编指令。具体代码如下所示。

指令清单 47.4　非优化的 GCC（Linaro[1]）4.9

```
str_trim:
        stp     x29, x30, [sp, -48]!
```

① Linaro 是一家开源的基于 ARM 操作平台的组织，由多家业内公司联合成立。其开发了 ARM 开发工具、Linux 内核以及 Linux 发行版（包括 Android 及 Ubuntu）的主要自动建构系统。

```
        add     x29, sp, 0
        str     x0, [x29,24] ; copy input argument into local stack
        ldr     x0, [x29,24] ; s
        bl      strlen
        str     x0, [x29,40] ; str_len variable in local stack
        b       .L2
; main loop begin
.L5:
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 13          ; is it '\r'?
        beq     .L3
        ldrb    w0, [x29,39]
; W0=c
        cmp     w0, 10          ; is it '\n'?
        bne     .L4             ; goto exit if it is not
.L3:
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
        strb    wzr, [x0]       ; write byte at s+str_len-1
; decrement str_len:
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        str     x0, [x29,40]
; save X0 (or str_len-1) to local stack
.L2:
        ldr     x0, [x29,40]
; str_len==0?
        cmp     x0, xzr
; goto exit then
        beq     .L4
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
; load byte at address s+str_len-1 to W0
        ldrb    w0, [x0]
        strb    w0, [x29,39] ; store loaded byte to "c"
        ldrb    w0, [x29,39] ; reload it
; is it zero byte?
        cmp     w0, wzr
; goto exit, if its zero or to L5 if its not''
        bne     .L5
.L4:
; return s
        ldr     x0, [x29,24]
        ldp     x29, x30, [sp], 48
        ret
```

47.5 ARM64:优化 GCC（Linaro）4.9

编译器采用了更为高级的优化技术。在程序开始时，首先调入第一个字符，并和十进制数 10 进行比对

（也就是换行 LF 的数值）。此后，后续字符相继被调入主循环。这种处理方法和本书第 47 章第 3 节中的例子类似。

指令清单 47.5　GCC（Linaro）4.9 的优化

```
str_trim:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     x19, x0
; X19 will always hold value of "s"
        bl      strlen
; X0=str_len
        cbz     x0, .L9         ; goto L9 (exit) if str_len==0
        sub     x1, x0, #1
; X1=X0-1=str_len-1
        add     x3, x19, x1
; X3=X19+X1=s+str_len-1
        ldrb    w2, [x19,x1]    ; load byte at address X19+X1=s+str_len-1
; W2=loaded character
        cbz     w2, .L9         ; is it zero? jump to exit then
        cmp     w2, 10          ; is it '\n'?
        bne     .L15
.L12:
; main loop body. loaded character is always 10 or 13 at this moment!
        sub     x2, x1, x0
; X2=X1-X0=str_len-1-str_len=-1
        add     x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
        strb    wzr, [x2,1]     ; store zero byte at address s+str_len-2+1=s+str_len-1
        cbz     x1, .L9         ; str_len-1==0? goto exit, if so
        sub     x1, x1, #1      ; str_len--
        ldrb    w2, [x19,x1]    ; load next character at address X19+X1=s+str_len-1
        cmp     w2, 10          ; is it '\n'?
        cbz     w2, .L9         ; jump to exit, if its zero'
        beq     .L12            ; jump to begin loop, if its' '\n'
.L15:
        cmp     w2, 13          ; is it '\r'?
        beq     .L12            ; yes, jump to the loop body begin
.L9:
; return "s"
        mov     x0, x19
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 32
        ret
```

47.6　ARM: Keil 6/2013 优化（ARM 模式）

这里我们会再次看到，编译器分配了 ARM 模式下的条件指令，使整个代码更为紧凑。

指令清单 47.6　Keil 6/2013 优化（ARM 模式）

```
str_trim PROC
        PUSH    {r4,lr}
; R0=s
        MOV     r4,r0
; R4=s
        BL      strlen          ; strlen() takes "s" value from R0
; R0=str_len
        MOV     r3,#0
; R3 will always hold 0
|L0.16|
```

```
        CMP     r0,#0           ; str_len==0?
        ADDNE   r2,r4,r0        ; (if str_len!=0) R2=R4+R0=s+str_len
        LDRBNE  r1,[r2,#-1]     ; (if str_len!=0) R1=load byte at address R2-1=s+str_len-1
        CMPNE   r1,#0           ; (if str_len!=0) compare loaded byte against 0
        BEQ     |L0.56|        ; jump to exit if str_len==0 or loaded byte is 0
        CMP     r1,#0xd         ; is loaded byte '\r'?
        CMPNE   r1,#0xa         ; (if loaded byte is not '\r') is loaded byte '\r'?
        SUBEQ   r0,r0,#1        ; (if loaded byte is '\r' or '\n') R0-- or str_len--
        STRBEQ  r3,[r2,#-1]     ; (if loaded byte is '\r' or '\n') store R3 (zero) at
    address R2-1=s+str_len-1
        BEQ     |L0.16|        ; jump to loop begin if loaded byte was '\r' or '\n'
|L0.56|
; return "s"
        MOV     r0,r4
        POP     {r4,pc}
        ENDP
```

47.7 ARM:Keil 6/2013（Thumb 模式）优化

在 Thumb 模式指令集的条件执行指令比 ARM 模式指令集的少，因此这种代码更接近 x86 的指令。但是在程序的 22 和 23 行处的偏移量 0x20 和 0x1f，会令多数人感到匪夷所思。为什么 Keil 6 编译器会分配这些指令？老实说，很难讲。也许这就是 Keil 6 优化进程的诡异之处。不管这种代码多么令人费解，整个程序的功能确实忠实于我们的源代码。

指令清单 47.7 Keil 6/2013（Thumb 模式）优化

```
1     str_trim PROC
1           PUSH    {r4,lr}
2         MOVS r4,r0
4     ; R4=s
5           BL          strlen          ; strlen() takes "s" value from R0
6     ; R0=str_len
7           MOVS    r3,#0
8     ; R3 will always hold 0
9           B       |L0.24|
10    |L0.12|
11          CMP     r1,#0xd         ; is loaded byte '\r'?
12          BEQ     |L0.20|
13          CMP     r1,#0xa         ; is loaded byte '\n'?
14          BNE     |L0.38|         ; jump to exit, if no
15    |L0.20|
16          SUBS    r0,r0,#1        ; R0-- or str_len--
17          STRB    r3,[r2,#0x1f]   ; store 0 at address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1
18    |L0.24|
19          CMP     r0,#0           ; str_len==0?
20          BEQ     |L0.38|         ; yes? jump to exit
21          ADDS    r2,r4,r0        ; R2=R4+R0=s+str_len
22          SUBS    r2,r2,#0x20     ; R2=R2-0x20=s+str_len-0x20
23          LDRB    r1,[r2,#0x1f]   ; load byte at
24       address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 to R1
25          CMP     r1,#0           ; is loaded byte 0?
26          BNE     |L0.12|         ; jump to loop begin, if its not 0'
27    |L0.38|
28    ; return "s"
29          MOVS    r0,r4
30          POP     {r4,pc}
31          ENDP
```

47.8　MIPS

指令清单 47.8　(IDA)GCC 4.4.5 优化

```
str_trim:
; IDA is not aware of local variable names, we gave them manually:
saved_GP          = -0x10
saved_S0          = -8
saved_RA          = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+saved_RA($sp)
                sw      $s0, 0x20+saved_S0($sp)
                sw      $gp, 0x20+saved_GP($sp)
; call strlen(). input string address is still in $a0, strlen() will take it from there:
                lw      $t9, (strlen & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jalr    $t9
; input string address is still in $a0, put it to $s0:
                move    $s0, $a0 ; branch delay slot
; result of strlen() (i.e, length of string) is in $v0 now
; jump to exit if $v0==0 (i.e., if length of string is 0):
                beqz    $v0, exit
                or      $at, $zero ; branch delay slot, NOP
                addiu   $a1, $v0, -1
; $a1 = $v0-1 = str_len-1
                addu    $a1, $s0, $a1
; $a1 = input string address + $a1 = s+strlen-1
; load byte at address $a1:
                lb      $a0, 0($a1)
                or      $at, $zero ; load delay slot, NOP
; loaded byte is zero? jump to exit if its so':
                beqz    $a0, exit
                or      $at, $zero ; branch delay slot, NOP
                addiu   $v1, $v0, -2
; $v1 = str_len-2
                addu    $v1, $s0, $v1
; $v1 = $s0+$v1 = s+str_len-2
                li      $a2, 0xD
; skip loop body:
                b       loc_6C
                li      $a3, 0xA ; branch delay slot
loc_5C:
; load next byte from memory to $a0:
                lb      $a0, 0($v1)
                move    $a1, $v1
; $a1=s+str_len-2
; jump to exit if loaded byte is zero:
                beqz    $a0, exit
; decrement str_len:
                addiu   $v1, -1     ; branch delay slot
loc_6C:
; at this moment, $a0=loaded byte, $a2=0xD (CR symbol) and $a3=0xA (LF symbol)
; loaded byte is CR? jump to loc_7C then:
                beq     $a0, $a2, loc_7C
                addiu   $v0, -1     ; branch delay slot
; loaded byte is LF? jump to exit if its not LF':
                bne     $a0, $a3, exit
                or      $at, $zero  ; branch delay slot, NOP
loc_7C:
; loaded byte is CR at this moment
```