封装printf函数 | TimerのBlog

yanglianoo.github.io/2023/07/11/封装printf函数

2023年7月11日

1. 可变参数

1.1 可变参数的使用

在实现printf函数之前，我们必须先了解以下在可变参数是如何实现的，在C语言标准库中，可变参数是通过如下几个函数或者宏定义来实现的：

c

```
va_list;
va_start(
ap,v);
va_arg(ap
,type)
va_end(ap
)
```

先举一个在x86架构下使用可变参数的例子：

c

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

void printStrings(int count, ...) {
    va_list args;
    va_start(args, count);

    for (int i = 0; i < count; i++) {
        const char* str = va_arg(args, const
char*);
        printf("%s\n", str);
    }

    va_end(args);
}

int main() {
    printStrings(3, "Hello", "World", "!");
    return 0;
}
```

执行上面的代码就会打印：

plaintext

va_list的使用方法：

- 首先在函数中定义一个具有va_list型的变量，这个变量是指向参数的指针。
- 然后用va_start宏初始化变量刚定义的va_list变量，使其指向第一个可变参数的地址。
- 然后va_arg返回可变参数，va_arg的第二个参数是你要返回的参数的类型（如果多个可变参数，依次调用va_arg获取各个参数）。
- 最后使用va_end宏结束可变参数的获取。

在使用va_list是应该注意以下问题：

- 可变参数的类型和个数完全由代码控制，**它并不能智能地识别不同参数的个数和类型**，正因为如此所以在上面的测试打印代码中我们需要传入一个参数的count值：3
- 如果我们不需要一一详解每个参数，只需要将可变列表拷贝到某个缓冲区，可以用vsprintf函数。
- 因为编译器对可变参数的函数原型检查不够严格，对编程查错不利，不利于我们写出高质量的代码

1.2 可变参数在不同架构下的体现

首先我们来看一下在i386架构下的一个可变参数的实现：

c

```
//将可变参数全部入栈
typedef char* va_list;
//在32位系统栈帧分配的单元大小是4字节(一个参数占4字节)
#define va_start(ap,v) (ap = (va_list)&v )
#define va_arg(ap,t) (*(t*)(ap += sizeof(t *)))
#define va_end(ap) (ap = (va_list)0)   //直接将va_list 置为
空指针
```

在32位x86架构下ABI的规定中函数的参数会被依次入栈，从右往左依次压入栈中，因此可变参数的实现可以以上面的形式实现，va_start指向的是第一个参数的地址，由于参数依次排列在栈中，所以其余的参数可以依次取出来，32位的栈帧的单元大小是4字节。

所以上面的测试代码在i386架构下参数的排列方式如下：参数1就是 count：3

我们来做个测试验证以下：

c

```c
#include "stdio.h"
//将可变参数全部入栈
typedef void* Va_list;
//在64位系统栈帧分配的单元大小是8字节(一个参数占8字节)
#define Va_start(ap,v) (ap = (Va_list)&v )
#define Va_arg(ap,t) (*(t*)(ap += sizeof(t *)))
#define Va_end(ap) (ap = (Va_list)0)   //直接将Va_list 置为
空指针


void print_str(int count, ...) {

#if 1 //i386
    /*这里就是复现上面可变参数的逻辑*/
    void* ap = (void*)&count;
    printf("%d\n", *(int *)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
#else // riscv64
    void* ap = (void*)&fmt;
    printf("%s\n", *(int*)ap);
    ap += sizeof(char**) * 6;
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
#endif
}

int main() {
    int s1 = 3;
    char * s2 = "world";
    char * s3 = "fuck you";
    char * s4 = "riscv";
    print_str(s1,s2,s3,s4);
    return 0;
}
```

编译后用qemu-i386运行：

sh

```sh
gcc -O0 -m32   va_list.c -o
i386.out
qemu-i386 i386.out
```

结果如下：

```
timer@DESKTOP-JI9EVEH:~/quard-star/test/va_list$ qemu-i386 i386.out
3
world
fuck you
riscv
```

可以看见正常答应所有的参数，证明在i386架构下参数是依次排列在栈中的，至于我为什么要打印“fuck you riscv”等下就会知道啦哈哈哈。

所以在32为X86架构下我们可以自己实现一个可变参数宏，但是我们的quard_star是64位的riscv架构，函数参数在栈中的排布是和编译器息息相关的，为了方便测试，我们建立如下文件夹：

```
timer@DESKTOP-JI9EVEH:~/quard-star/test/va_list$ ls
a.out     i386.out        riscv32.out        riscv64.out        test.c
build.sh  i386_debug.txt  riscv32_debug.txt  riscv64_debug.txt  va_list.c
```

其中`va_list.c`做了一点修改：

c

```c
#include "stdio.h"

void print_str(const char * fmt,
...) {

#if 1
    void* ap = (void*)&fmt;
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
#else
    void* ap = (void*)&fmt;
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**) * 6;
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
#endif
}

int main() {
    char * s1 = "hello";
    char * s2 = "world";
    char * s3 = "fuck you";
    char * s4 = "riscv";
    print_str(s1,s2,s3,s4);
    return 0;
}
```

test.c先不管，`build.sh`的内容如下：

shell

```
gcc -O0 -m32   va_list.c -o i386.out
riscv64-unknown-elf-gcc -O0 va_list.c -o riscv64.out
riscv32-unknown-elf-gcc -O0   va_list.c -o riscv32.out

objdump -D i386.out > i386_debug.txt
riscv64-unknown-elf-objdump -D riscv64.out >
riscv64_debug.txt
riscv32-unknown-elf-objdump -D riscv32.out >
riscv32_debug.txt

#   qemu-i386 i386.out
#   qemu-riscv64 riscv64.out
#   qemu-riscv32 riscv32.out
```

可以看见会分别把`va_list.c`编译成I386架构、riscv32架构、riscv64架构的可执行程序，然后将可执行程序反汇编输出到各自的.txt文件中。

我们来`qemu-i386 i386.out`，可以看见正常输出没问题：

```
timer@DESKTOP-JI9EVEH:~/quard-star/test/va_list$ qemu-i386 i386.out
hello
world
fuck you
riscv
```

如果riscv64架构下编译器也把函数参数一个个依次排放在栈中，那么执行的结果肯定和i386一样，ok，我们`qemu-riscv64 riscv64.out`运行一下。

```
timer@DESKTOP-JI9EVEH:~/quard-star/test/va_list$ qemu-riscv64 riscv64.out
hello
Segmentation fault
```

可以看见除了第一个参数正常输出了，后面的直接报错了。这是为啥呢，我们来看一下生成的汇编代码，打开`riscv64_debug.txt`，找到`main`函数和`print_str`函数

```
  1024e:   fd043683            ld a3,-48(s0)
  10252:   fd843603            ld a2,-40(s0)
  10256:   fe043583            ld a1,-32(s0)
  1025a:   fe843503            ld a0,-24(s0)
  1025e:   f43ff0ef            jal   101a0 <print_str>
```

可以看见在main函数中将4个函数参数分别放进了a0,a1,a2,a3寄存器，这是riscv的ABI规定的：

寄存器 接口名称 描述 在调用中是否保留?

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero 硬编码 0	—
x1	ra	Return address 返回地址	No
x2	sp	Stack pointer 栈指针	Yes
x3	gp	Global pointer 全局指针	—
x4	tp	Thread pointer 线程指针	—
x5	t0	Temporary/alternate link register 临时寄存器	No /备用链接寄存器
x6–7	t1–2	Temporaries 临时寄存器	No
x8	s0/fp	Saved register/frame pointer 保存寄存器	Yes /帧指针
x9	s1	Saved register 保存寄存器	Yes
x10–11	a0–1	Function arguments/return values 函数参数	No /返回值
x12–17	a2–7	Function arguments 函数参数	No
x18–27	s2–11	Saved registers 保存寄存器	Yes
x28–31	t3–6	Temporaries 临时寄存器	No
f0–7	ft0–7	FP temporaries 浮点临时寄存器	No
f8–9	fs0–1	FP saved registers 浮点保存寄存器	Yes
f10–11	fa0–1	FP arguments/return values 浮点参数/返回值	No
f12–17	fa2–7	FP arguments 浮点参数	No
f18–27	fs2–11	FP saved registers 浮点保存寄存器	Yes
f28–31	ft8–11	FP temporaries 浮点临时寄存器	No

a0~a7用于函数传参，然后我们找到`print_str`函数：

```
101a8:   fca43c23          sd a0,-40(s0)
101ac:   e40c              sd a1,8(s0)
101ae:   e810              sd a2,16(s0)
101b0:   ec14              sd a3,24(s0)
101b2:   f018              sd a4,32(s0)
```

在`print_str`函数的栈帧中，可以看见编译器将a0放在了栈中一个奇怪的位置，这里的a0中存的就是第一个参数，后面三个参数的值是依次排放的，ok我们来做一下测试：

c

```c
#include "stdio.h"

void print_str(const char * fmt,
...) {

#if 0
    void* ap = (void*)&fmt;
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
#else //riscv64
    void* ap = (void*)&fmt;
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**) * 6;
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
    ap += sizeof(char**);
    printf("%s\n", *(char**)ap);
#endif
}

int main() {
    char * s1 = "hello";
    char * s2 = "world";
    char * s3 = "fuck you";
    char * s4 = "riscv";
    print_str(s1,s2,s3,s4);
    return 0;
}
```

在上面的代码中我在获取第一个参数后，在获取第二个参数时：`ap += sizeof(char**) * 6;`，将偏移量乘了6，那是因为在上面汇编代码中，我们发现a0和a1在栈中的偏移量是48个字节，在64位系统中栈帧每个单元为8个字节，因此乘以六刚刚好可以访问到。运行看一下结果：

```
timer@DESKTOP-JI9EVEH:~/quard-star/test/va_list$ qemu-riscv64 riscv64.out
hello
world
fuck you
riscv
```

正常输出没问题，因此在riscv64下编译器有自己的一套存放参数的方式，我无法自己实现一个类似i386架构下的可变参数宏，不知道写编译器的人做了哪些操作，我去看源码里riscv架构也没定义va_list宏。可以看看这篇博客：

编程参考 - va_list的定义问题_va_list 头文件_夜流冰的博客-CSDN博客

2. 实现printf函数

由于printf函数需要用到可变参数，但是上面的分析中不知道如何在riscv64下实现自己的va_list系列的宏，那我们就只有使用编译器提供的了

先看一下os目录下新增了哪些文件：

sh

```
timer@DESKTOP-JI9EVEH:~/quard-star/os$ ls
Makefile  entry.S  main.c  os.h  os.ld  printf.c  sbi.c
sbi.h
```

`os.h`：用extern声明了两个函数，包含了`<stdarg.h>`头文件，这里有可变参数相关的宏

c

```
#ifndef __OS_H__
#define __OS_H__

#include <stddef.h>
#include <stdarg.h>

/* printf */
extern int  printf(const char* s,
...);
extern void panic(char *s);
extern void
sbi_console_putchar(int ch);

#endif /* __OS_H__ */
```

`printf.c`:

c

```
#include "os.h"
void uart_puts(char *s)
{
        while (*s) {
                sbi_console_putchar(*s++);
        }
}

static int _vsnprintf(char * out, size_t n, const char* s, va_list vl)
{
        int format = 0;
        int longarg = 0;
        size_t pos = 0;
        for (; *s; s++) {
                if (format) {
                        switch(*s) {
                        case 'l': {
                                longarg = 1;
                                break;
                        }
                        case 'p': {
                                longarg = 1;
                                if (out && pos < n) {
                                        out[pos] = '0';
                                }
                                pos++;
                                if (out && pos < n) {
                                        out[pos] = 'x';
                                }
                                pos++;
                        }
                        case 'x': {
                                long num = longarg ? va_arg(vl, long) : va_arg(vl,
int);
                                int hexdigits = 2*(longarg ? sizeof(long) :
sizeof(int))-1;
                                for(int i = hexdigits; i >= 0; i--) {
                                        int d = (num >> (4*i)) & 0xF;
                                        if (out && pos < n) {
                                                out[pos] = (d < 10 ? '0'+d : 'a'+d-
10);
                                        }
                                        pos++;
                                }
                                longarg = 0;
                                format = 0;
                                break;
                        }
                        case 'd': {
                                long num = longarg ? va_arg(vl, long) : va_arg(vl,
int);
                                if (num < 0) {
                                        num = -num;
                                        if (out && pos < n) {
                                                out[pos] = '-';
                                        }
                                        pos++;
                                }
```

```
                                long digits = 1;
                                for (long nn = num; nn /= 10; digits++);
                                for (int i = digits-1; i >= 0; i--) {
                                        if (out && pos + i < n) {
                                                out[pos + i] = '0' + (num % 10);
                                        }
                                        num /= 10;
                                }
                                pos += digits;
                                longarg = 0;
                                format = 0;
                                break;
                        }
                        case 's': {
                                const char* s2 = va_arg(vl, const char*);
                                while (*s2) {
                                        if (out && pos < n) {
                                                out[pos] = *s2;
                                        }
                                        pos++;
                                        s2++;
                                }
                                longarg = 0;
                                format = 0;
                                break;
                        }
                        case 'c': {
                                if (out && pos < n) {
                                        out[pos] = (char)va_arg(vl,int);
                                }
                                pos++;
                                longarg = 0;
                                format = 0;
                                break;
                        }
                        default:
                                break;
                        }
                } else if (*s == '%') {
                        format = 1;
                } else {
                        if (out && pos < n) {
                                out[pos] = *s;
                        }
                        pos++;
                }
        }
        if (out && pos < n) {
                out[pos] = 0;
        } else if (out && n) {
                out[n-1] = 0;
        }
        return pos;
}

static char out_buf[1000]; // buffer for _vprintf()
static int _vprintf(const char* s, va_list vl)
{
        int res = _vsnprintf(NULL, -1, s, vl);
        if (res+1 >= sizeof(out_buf)) {
```

```
                uart_puts("error: output string size overflow\n");
                while(1) {}
        }
        _vsnprintf(out_buf, res + 1, s, vl);
        uart_puts(out_buf);
        return res;
}

int printf(const char* s, ...)
{
        int res = 0;
        va_list vl;
        va_start(vl, s);
        res = _vprintf(s, vl);
        va_end(vl);
        return res;
}

void panic(char *s)
{
        printf("panic: ");
        printf(s);
        printf("\n");
        while(1){};
}
```

上面这段代码抄的是plctlab/riscv-operating-system-mooc: 《从头写一个RISC-V OS》课程配套的资源 (github.com)里的代码，简单说一下实现`printf`函数的逻辑，

一般来说printf函数有多个参数，其中第一个参数为一个字符串，后面的可变参数是为了对应到如`%d,%s,%c`等格式化性质，例如：

c

```
printf("arg1:%d
arg2:%s",5,"hello")
```

所以printf函数的定义形式为：

c

```
int printf(const char*
s, ...)
```

根据上面的代码来分析，首先定义printf函数:

c

```
int printf(const char*
s, ...)
{
        int res = 0;
        va_list vl;
        va_start(vl, s);
        res =
_vprintf(s, vl);
        va_end(vl);
        return res;
}
```

这里的res实际上是第一个参数的字符串的长度，核心的函数为`_vprintf`，代码中将第一个参数的指针即`va_list vl;`和第一个参数实际的值传给了_vprintf

c

```
static char out_buf[1000]; // buffer for _vprintf()
static int _vprintf(const char* s, va_list vl)
{
        int res = _vsnprintf(NULL, -1, s, vl);
        if (res+1 >= sizeof(out_buf)) {
                uart_puts("error: output string size
overflow\n");
                while(1) {}
        }
        _vsnprintf(out_buf, res + 1, s, vl);
        uart_puts(out_buf);
        return res;
}
```

在_vprintf函数中首先先执行可一段代码：

c

```
int res = _vsnprintf(NULL, -1,
s, vl);
```

```c
static int _vsnprintf(char * out, size_t n, const char* s, va_list vl)
{
    int format = 0;
    int longarg = 0;
    size_t pos = 0;
    for (; *s; s++) {
        if (format) {…
        } else if (*s == '%') {
            format = 1;
        } else {
            if (out && pos < n) {
                out[pos] = *s;
            }
            pos++;
        }
        }
    if (out && pos < n) {
        out[pos] = 0;
    } else if (out && n) {
        out[n-1] = 0;
    }
    return pos;
}
```

分析一下发现这里会返回printf函数第一个参数的长度，比如：`printf("arg:%s","hello")`，那么res的值就是`"arg:%s"`的长度即6。然后判断一下是否超过最大长度。然后再调用：

c

```
_vsnprintf(out_buf, res + 1,
s, vl);
```

`_vsnprintf`就是用来对字符串进行格式化的，通过判断第一个参数里`%`的个数来确定可变参数的个数，详细的实现可以看上面代码。

`main.c`:比较简单，就是调用printf函数来输出

c

```
#include "os.h"
void os_main()
{
   printf("hello timer
os!");
}
```

`Makefile`:

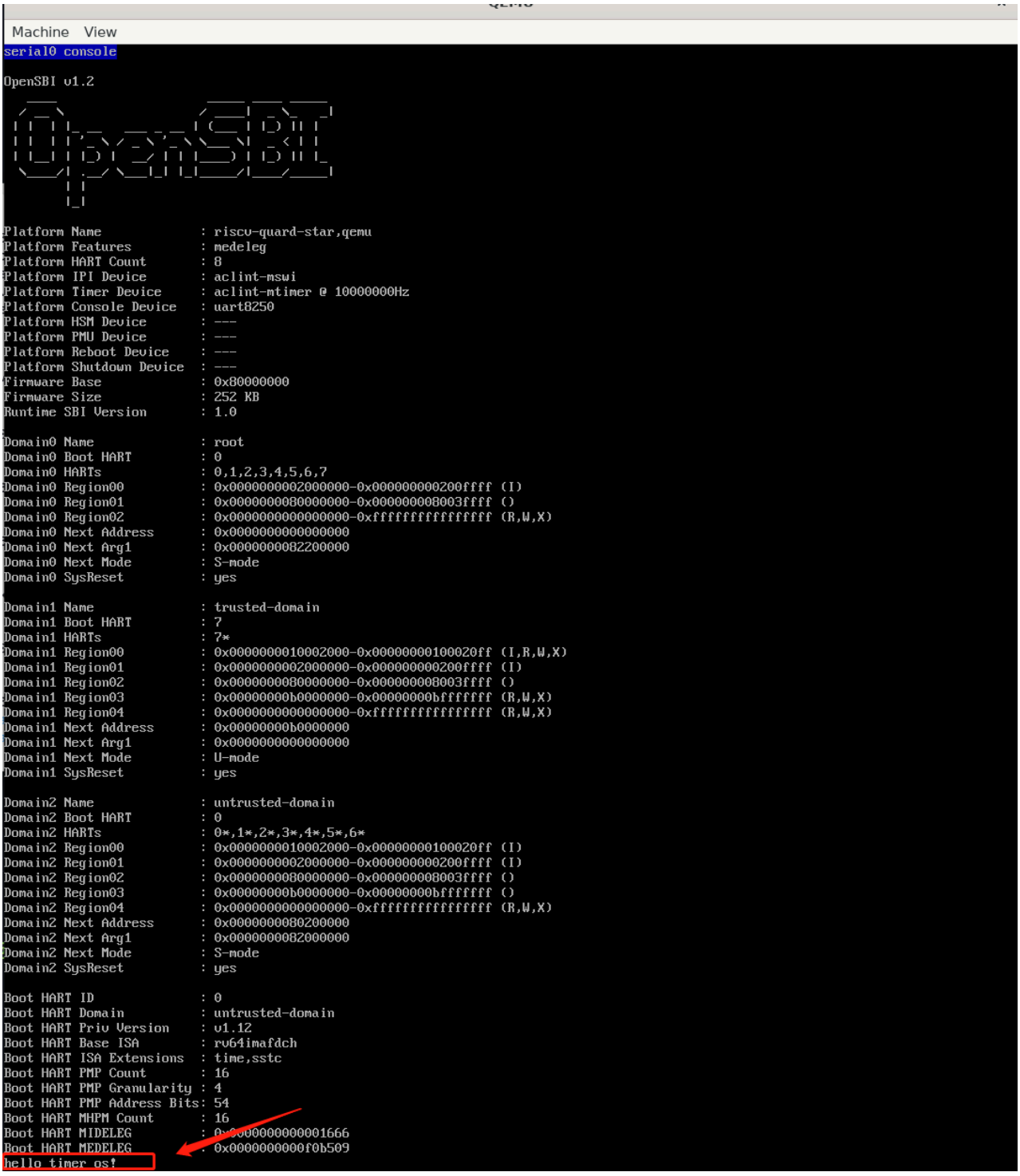
plaintext

```
SRCS_C = \

sbi.c \

main.c \

printf.c \
#加一行
```

3. 测试

c

```
timer@DESKTOP-JI9EVEH:~/quard-star$
./build.sh
timer@DESKTOP-JI9EVEH:~/quard-star$
./run.sh
```



成功打印!

如果发现编译错误，在makefile里加一个编译命令：

makefile

```
CFLAGS = -nostdlib -fno-builtin -
mcmodel=medany
```

参考链接

文章作者: Timer
文章链接: https://yanglianoo.github.io/2023/07/11/封装printf函数/

相关推荐