用 setjmp 和 longjmp 实现多线程

文/欧承祖

setjmp 和 longjmp 介绍

通常来说，setjmp/longjmp 被用来实现「异常处理机制」，因为longjmp 可以跨越多个层级的函数调用来恢复程序或线程的状态。

函数 int setjmp(jmp_buf env) 创建本地的 jmp_buf 缓冲区并且初始化，用于将来跳转回此处。这个子程序保存程序的调用环境于 env 参数所指的缓冲区，env 将被 longjmp 使用。如果是从 setjmp 直接调用返回，setjmp 返回值为 0。如果是从 longjmp 恢复的程序调用环境返回，setjmp 返回非零值。

函数 void longjmp(jmp_buf env, int value) 恢复 env 所指的缓冲区中的程序调用环境上下文，env 所指缓冲区的内容是由 setjmp 子程序调用所保存。value 的值从 longjmp 传递给 setjmp。longjmp 完成后，程序从对应的 setjmp 调用处继续执行，如同 setjmp 调用刚刚完成。如果 value 传递给 longjmp 零值，setjmp 的返回值为1；否则，setjmp 的返回值为 value。

setjmp 保存当前的环境（即程序的状态）到平台相关的一个数据结构 (jmp_buf)，该数据结构在随后程序执行的某一点可被 longjmp 用于恢复程序的状态到 setjmp 调用所保存到 jmp_buf 时的原样。这一过程可以认为是「跳转」回 setjmp 所保存的程序执行状态。setjmp 的返回值指出控制是正常到达该点还是通过调用 longjmp 恢复到该点。因此有编程的惯用法: if(setjmp(x)){/* handle longjmp(x) */}。

这是一个使用 setjmp/longjmp 的简单例子:

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

static jmp_buf buf;

void second(void) {
    printf("second\n");         // 打印
    longjmp(buf,1);             // 跳回setjmp的调用处 - 使 setjmp返回值为1
}

void first(void) {
    second();
    printf("first\n");          // 不可能执行到此行
}

int main() {
    if ( ! setjmp(buf) ) {
        first();                // 进入此行前，setjmp返回0
    } else {                    // 当longjmp跳转回，setjmp返回1，因此进入此行
        printf("main\n");       // 打印
    }

    return 0;
}
```

程序的输出结果为：

```
second
main
```

用 setjmp 和 longjmp 实现协程

除了异常处理机制之外，setjmp/longjmp 还可以被用来实现「合作式多任务（cooperative multitasking）」。下面来看这一段代码：

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

jmp_buf trampoline, jb0, jb1;

void switch_thread(jmp_buf old, jmp_buf new) {
    if (!setjmp(old)) {
        longjmp(new, 1);
        /* NEVER REACHED */
    }
}

void thread1() {
    char x[4096];
    x[4096-1] = 1;
    int count = 100;

    while (1) {
        printf("Thread 1; Count: %d\n", count);
        if (--count == 0) count = 100;
        switch_thread(jb1, jb0);
    }
}

void thread0() {
    int count = 0;

    while (1) {
        printf("Thread 0; Count: %d\n", count);
        if (++count == 100) count = 0;
        switch_thread(jb0, jb1);
    }
}

int main()
{
    if (!setjmp(trampoline)) {
        if (!setjmp(jb0)) {
            longjmp(trampoline, 1);
            /* NEVER REACHED */
        } else {
            thread0();
        }
    } else {
        thread1();
    }

    return 0;
}
```

这段代码的输出结果是：

```
Thread 1; Count: 100
Thread 0; Count: 0
Thread 1; Count: 99
Thread 0; Count: 1
...
```

即线程 1 和线程 2 循环往复地执行。

上面代码 `thread1` 函数中的 `x[4096]` 是显式保留的、供主函数运行的空间。因为即便堆栈益处也不会有异常抛出，所以事前显式地预留多余的堆栈空间是有必要的。虽然 `setjmp/longjmp` 可以被用来实现一种协程，但这种实现方式不仅考验程序员的能力，而且可能在不同环境中会出现难以预料的错误。不建议在生产环境中使用。

参考资料

- https://en.wikipedia.org/wiki/Setjmp.h#Cooperative_multitasking
- https://zh.wikipedia.org/wiki/%E5%8D%8F%E7%A8%8B
- https://en.wikipedia.org/wiki/Coroutine#Implementations_for_C
- https://gist.github.com/markwinter/83e19f35eca56124d889