这是道哥的第 017 篇原创

关注+星标公众号，不错过最新文章

一、前言

在 C 标准库中，有两个威力很猛的函数：setjmp 和 longjmp，不知道各位小伙伴在代码中是否使用过？我问了身体的几位同事，一部分人不认识这两个函数，有一部分人知道这个函数，但从来没有使用过。

从知识点范围来看，这两个函数的功能比较单纯，一个简单的示例代码就能说清楚了。但是，我们需要从这个知识点进行发散、思考，在不同的维度上，把这个知识点与这个编程语言中其它类似的知识进行联想、对比；与其他编程语言中类似的概念进行比较；然后再思考这个知识点可以使用在哪些场合，别人是怎么来使用它的。

今天，我们就来掰扯掰扯这两个函数。虽然在一般的程序中使用不上，但是在今后的某个场合，当你需要处理一些比较奇特的程序流程时，也许它们可以给你带来意想不到的效果。

例如：我们会把 setjmp/longjmp 与 goto 语句进行功能上的比较；与 fork 函数从返回值上进行类比；与 Python/Lua 语言中的协程进行使用场景上的比较。

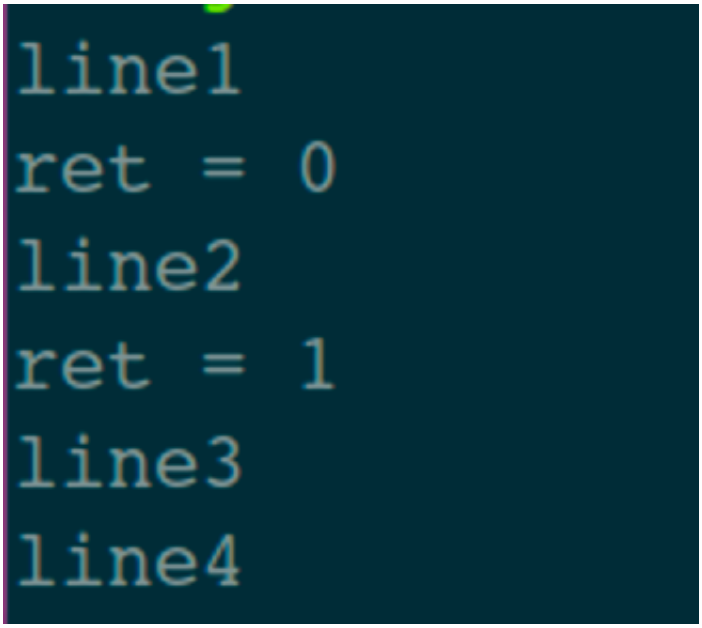
二、函数语法介绍

1. 最简示例

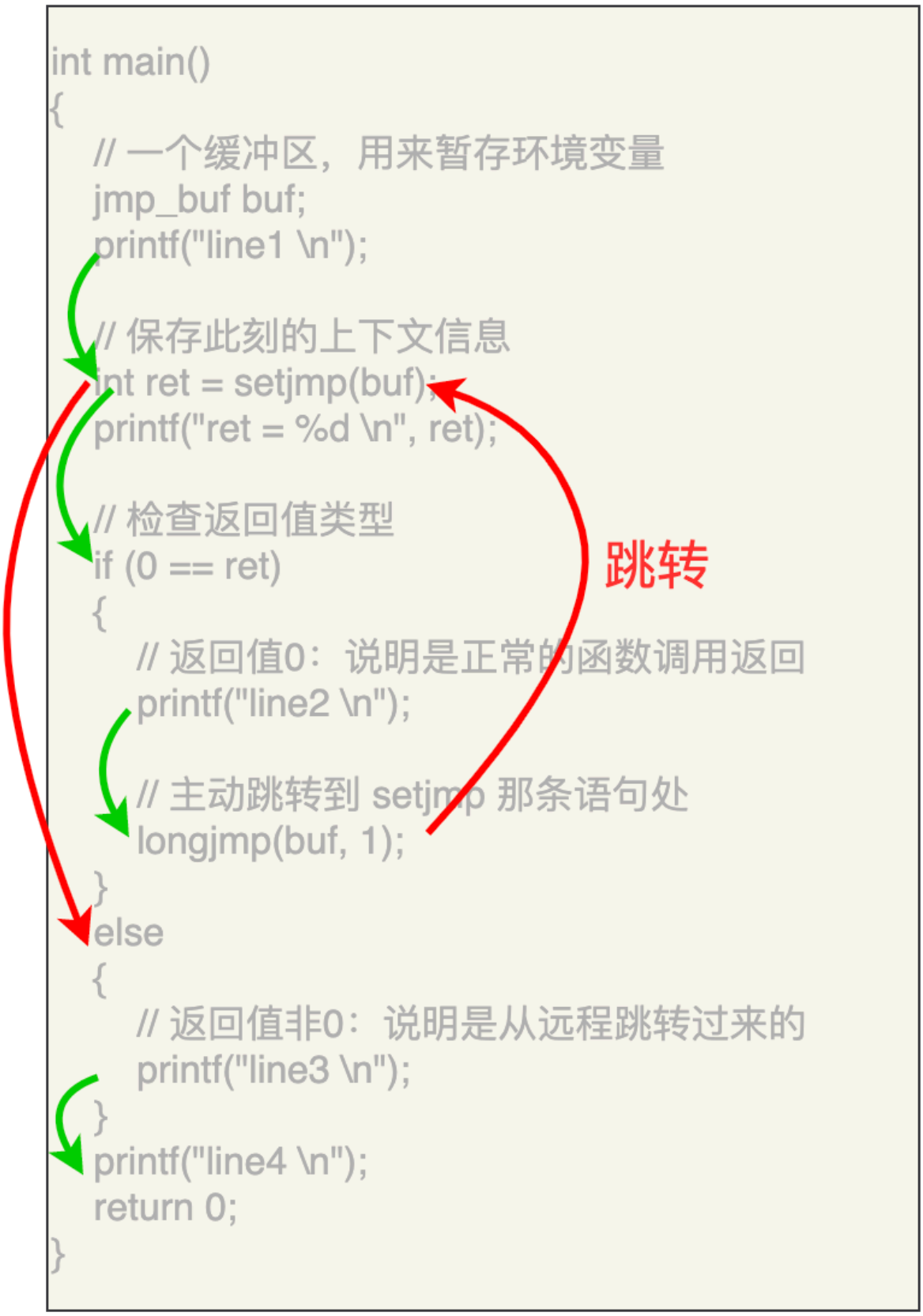
先不讲道理，直接看一下这个最简单的示例代码，看不懂也没关系，混个脸熟:

```
int main()
{
    // 一个缓冲区，用来暂存环境变量
    jmp_buf buf;
    printf("line1 \n");

    // 保存此刻的上下文信息
    int ret = setjmp(buf);
    printf("ret = %d \n", ret);

    // 检查返回值类型
    if (0 == ret)
    {
        // 返回值0：说明是正常的函数调用返回
        printf("line2 \n");

        // 主动跳转到 setjmp 那条语句处
        longjmp(buf, 1);
    }
    else
    {
        // 返回值非0：说明是从远程跳转过来的
        printf("line3 \n");
    }
    printf("line4 \n");
    return 0;
}
```

执行结果：



执行顺序如下(如果不明白就不要深究，看完下面的解释再回过头来看):



2. 函数说明

首先来看下这个 2 个函数的签名：

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int value);
```

它们都在头文件 `setjmp.h` 中进行声明，维基百科的解释如下：

> setjmp: Sets up the local jmp_buf buffer and initializes it for the jump. This routine saves the program's calling environment in the environment buffer specified by the env argument for later use by longjmp. If the return is from a direct invocation, setjmp returns 0. If the return is from a call to longjmp, setjmp returns a nonzero value。
> longjmp：Restores the context of the environment buffer env that was saved by invocation of the setjmp routine in the same invocation of the program. Invoking longjmp from a nested signal handler is undefined. The value specified by value is passed from longjmp to setjmp. After longjmp is completed, program execution continues as if the corresponding invocation of setjmp had just returned. If the value passed to longjmp is 0, setjmp will behave as if it had returned 1; otherwise, it will behave as if it had returned value。

下面我再用自己的理解把上面这段英文解释一下：

setjmp 函数

> 1. 功能：把执行这个函数时的各种上下文信息保存起来，主要就是一些寄存器的值；
> 2. 参数：用来保存上下文信息的缓冲区，相当于把当前的上下文信息拍一个快照保存起来；
> 3. 返回值：有 2 种返回值，如果是直接调用 setjmp 函数时，返回值是 0；如果是调用 longjmp 函数跳转过来时，返回值是非 0；这里可以与创建进程的函数 fork 进行一下类比。

longjmp 函数

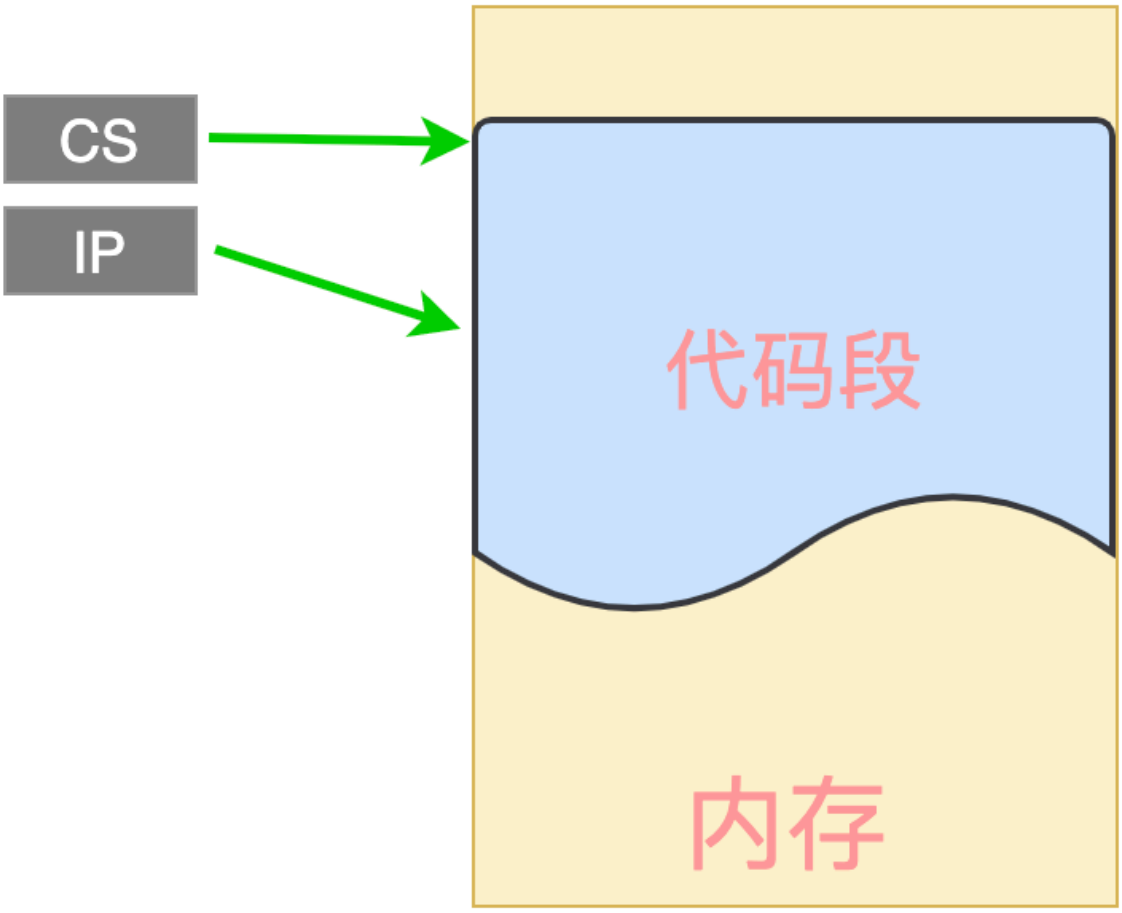
> 1. 功能：跳转到参数 env 缓冲区中保存的上下文(快照)中去执行；
> 2. 参数：env 参数指定跳转到哪个上下文中(快照)去执行， value 用来给 setjmp 函数提供返回判断信息，也就是说：调用 longjmp 函数时，这个参数 value 将会作为 setjmp 函数的返回值；
> 3. 返回值：没有返回值。因为在调用这个函数时，就直接跳转到其他地方的代码去执行了，不会再回来了。

小结：这 2 个函数是配合使用的，用来实现程序的跳转。

3. setjmp：保存上下文信息

我们知道，C 代码在编译成二进制文件之后，在执行时被加载到内存中，CPU 按照顺序到代码段取出每一条指令来执行。在 CPU 中有很多个寄存器，用来保存当前的执行环境，比如：代码段寄存器CS、指令偏移量寄存器IP，当然了还有其他很多其它寄存器，我们把这个执行环境称作上下文。

CPU 在获取下一条执行指令时，通过 CS 和 IP 这 2 个寄存器就能获取到需要执行的指令，如下图：



补充一下知识点：

1. 上图中，把代码段寄存器 CS 当做一个基地址来看待了，也就是说：CS 指向代码段在内存中的开始地址，IP 寄存器代表下一个要执行的指令地址距离这个基地址的偏移量。因此每次取指令时，只需要把这 2 个寄存器中的值相加，就得到了指令的地址；
2. 其实，在 x86 平台上，代码段寄存器 CS 并不是一个基地址，而是一个选择子。在操作系统的某个地方有一个表格，这个表格里存储了代码段真正的开始地址，而 CS 寄存器中 只是存储了一个索引值，这个索引值指向这个表格中的某个表项，这里涉及到虚拟内存的相关知识了；
3. IP 寄存器在获取一条指令之后，自动往下移动到下一个指令的开始位置，至于移动多少个字节，那就要看当前取出的这条指令占用了多少个字节。

CPU 是一个大傻瓜，它没有任何的想法，我们让它干什么，它就干什么。比如取指令：我们只要设置 CS 和 IP 寄存器，CPU 就用这 2 个寄存器里的值去获取指令。如果把这 2 个寄存器设置为一个错误的值，CPU 也会傻不拉几的去取指令，只不过在执行时就会崩溃。

我们可以简单的把这些寄存器信息理解为上下文信息，CPU 就根据这些上下文信息来执行。因此，C 语言为我们准备了 setjmp 这个库函数来把当前的上下文信息保存起来，暂时存储到一个缓冲区中。

保存的目的是什么？为了在以后可以恢复到当前这个地方继续执行。

还有一个更简单的例子：服务器中的快照。快照的作用是什么？当服务器出现错误时，可以恢复到某个快照！

4. longjmp: 实现跳转

说到跳转，脑袋中立刻跳出的概念就是 goto 语句，我发现很多教程都对 goto 语句很有意见，认为在代码中应该尽量不要使用它。这样的观点出发点是好的：如果 goto 使用太多，会影响对代码执行顺序的理解。

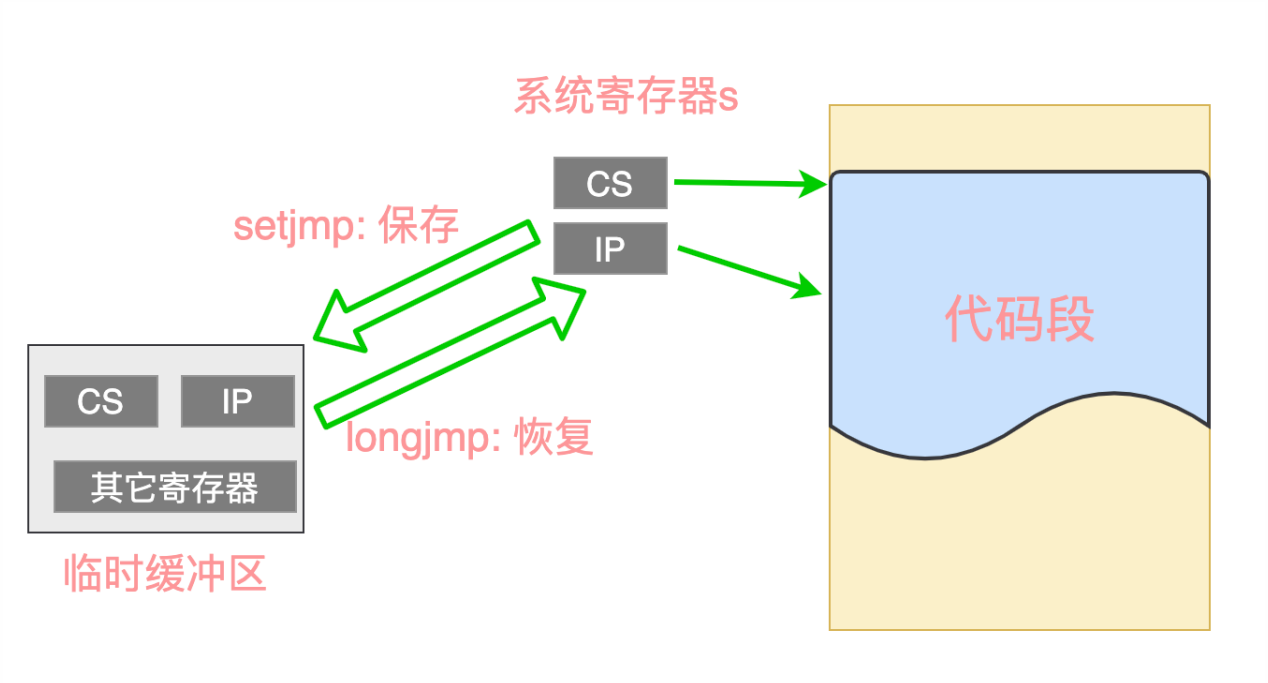
但是如果看一下 Linux 内核的代码，可以发现很多的 goto 语句。还是那句话：在代码维护和执行效率上要寻找一个平衡点。

跳转改变了程序的执行序列，goto 语句只能在函数内部进行跳转，如果是跨函数它就无能为力了。

因此，C 语言中为我们提供了 longjmp 函数来实现远程跳转，从它的名字就可以额看出来，也就是说可以跨函数跳转。

从 CPU 的角度看，所谓的跳转就是把上下文中的各种寄存器设置为某个时刻的快照，很显然，上面的 setjmp 函数中，已经把那个时刻的上下文信息(快照)存储到一个临时缓冲区中了，如果要跳转到那个地方去接着执行，直接告诉 CPU 就行了。

怎么告诉 CPU 呢？ 就是把临时缓冲区中的这些寄存器信息覆盖掉 CPU 中使用的那些寄存器即可。



5. setjmp：返回类型和返回值

在某些需要多进程的程序中，我们经常使用 fork 函数来从当前的进程中"孵化"一个新的进程，这个新进程从 fork 这个函数的下一条语句开始执行。

对于主进程来说，调用 fork 函数之后返回，也是继续执行下一条语句，那么如何来区分是主进程还是新进程呢？ fork 函数提供了一个返回值给我们来进行区分：

> fork 函数返回 0：代表这是新进程；
> fork 函数返回非 0：代表是原来的主进程，返回数值是新进程的进程号。

类似的，setjmp 函数也有不同的返回类型。也许用返回类型来表述不太准确，可以这样理解：从 setjmp 函数返回，一共有 2 个场景：

> 1. 主动调用 setjmp 时：返回 0，主动调用的目的是为了保存上下文，建立快照。
> 2. 通过 longjmp 跳转过来时：返回非 0，此时的返回值是由 longjmp 的第二个参数来指定的。

根据以上这 2 种不同的值，我们就可以进行不同的分支处理了。当通过 longjmp 跳转返回的时候，可以根据实际场景，返回不同的非 0 值。有过 Python、Lua 等脚本语言编程经验的小伙伴，是不是想到了 yield/resume 函数？它们在参数、返回值上的外在表现是一样的！

小结：到这里，基本上把 setjmp/longjmp 这 2 个函数的使用方法讲完了，不知道我描述的是否足够清楚。此时，再看一下文章开头的示例代码，应该一目了然了。

三、利用 setjmp/longjmp 实现异常捕获

既然 C 函数库给我们提供了这个工具，那就肯定存在一定的使用场景。异常捕获在一些高级语言中(Java/C++)，直接在语法层面进行了支持，一般就是 try-catch 语句，但是在 C 语言中需要自己去实现。

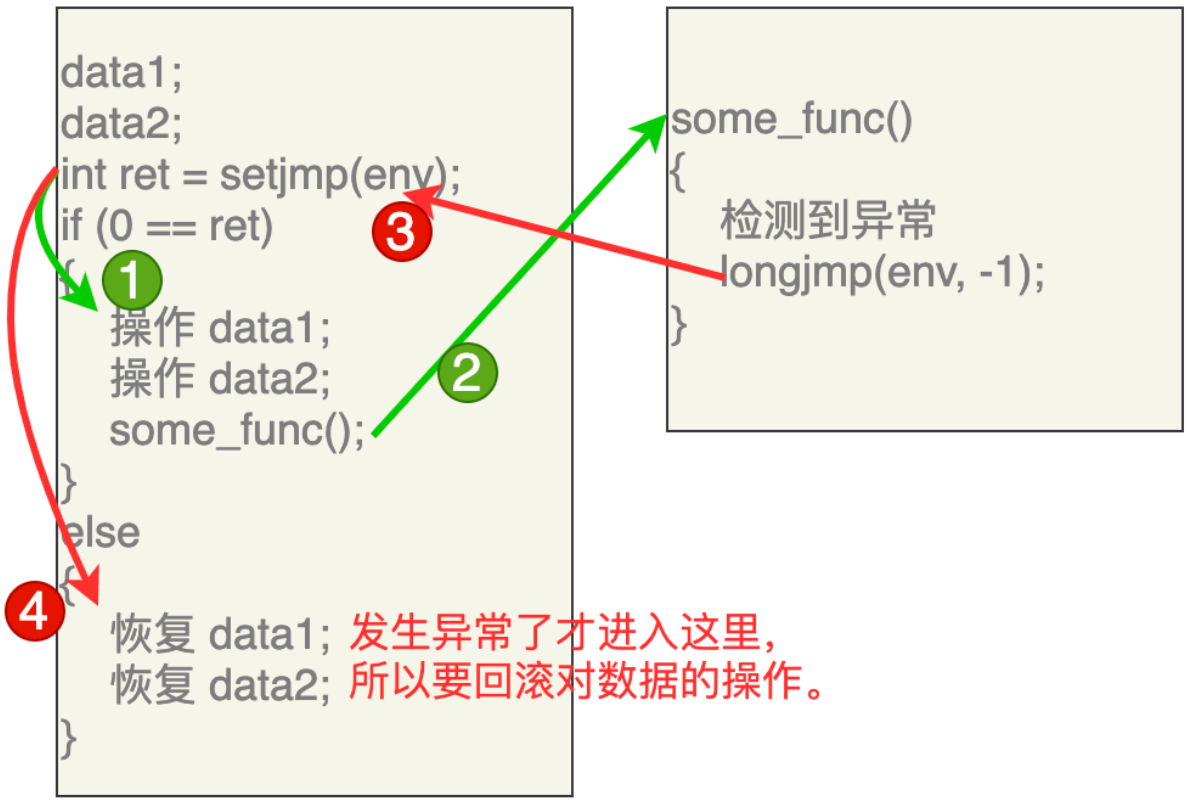
我们来演示一个最简单的异常捕获模型，代码一共 56 行：

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

typedef int     BOOL;
#define TRUE    1
#define FALSE   0

// 枚举：错误代码
typedef enum _ErrorCode_ {
    ERR_OK = 100,          // 没有错误
    ERR_DIV_BY_ZERO = -1   // 除数为 0
} ErrorCode;

// 保存上下文的缓冲区
jmp_buf gExcptBuf;

// 可能发生异常的函数
typedef int (*pf)(int, int);
int my_div(int a, int b)
{
    if (0 == b)
    {
        // 发生异常，跳转到函数执行之前的位置
        // 第2个参数是异常代码
        longjmp(gExcptBuf, ERR_DIV_BY_ZERO);
    }
```

```
    // 没有异常，返回正确结果
    return a / b;
}

// 在这个函数中执行可能会出现异常的函数
int try(pf func, int a, int b)
{
    // 保存上下文，如果发生异常，将会跳入这里
    int ret = setjmp(gExcptBuf);
    if (0 == ret)
    {
        // 调用可能发生异常的哈数
        func(a, b);
        // 没有发生异常
        return ERR_OK;
    }
    else
    {
        // 发生了异常，ret 中是异常代码
        return ret;
    }
}

int main()
{
    int ret = try(my_div, 8, 0);     // 会发生异常
    // int ret = try(my_div, 8, 2);  // 不会发生异常
    if (ERR_OK == ret)
    {
        printf("try ok ! \n");
    }
    else
    {
        printf("try excepton. error = %d \n", ret);
    }

    return 0;
}
```

代码就不需要详细说明了，直接看代码中的注释即可明白。这个代码仅仅是示意性的，在生产代码中肯定需要更完善的包装才能使用。

有一点需要注意：setjmp/longjmp 仅仅是改变了程序的执行顺序，应用程序自己的一些数据如果需要回滚的话，需要我们自己手动处理。



四、利用 setjmp/longjmp 实现协程

1. 什么是协程

在 C 程序中，如果需要并发执行的序列一般都是用线程来实现的，那么什么是协程呢？维基百科对于协程的解释是：

> **协程**（英语：coroutine）是计算机程序的一类组件，推广了协作式多任务的子程序，允许执行被挂起与被恢复。相对子例程而言，协程更为一般和灵活，但在实践中使用没有子例程那样广泛。协程更适合于用来实现彼此熟悉的程序组件，如协作式多任务、异常处理、事件循环、迭代器、无限列表和管道。

更详细的信息在这个页面 协程，网页中具体描述了协程与线程、生成器的比较，各种语言中的实现机制。

我们用生产者和消费者来简单体会一下协程和线程的区别：

2. 线程中的生产者和消费者

1. 生产者和消费者是 2 个并行执行的序列，通常用 2 个线程来执行；
2. 生产者在生产商品时，消费者处于等待状态(阻塞)。生产完成后，通过信号量通知消费者去消费商品；
3. 消费者在消费商品时，生产者处于等待状态(阻塞)。消费结束后，通过信号量通知生产者继续生

> 产商品。

3. 协程中的生产者和消费者

> 1. 生产者和消费者在同一个执行序列中执行，通过执行序列的跳转来交替执行；
> 2. 生产者在生产商品之后，放弃 CPU，让消费者执行；
> 3. 消费者在消费商品之后，放弃 CPU，让生产者执行；

4. C 语言中的协程实现

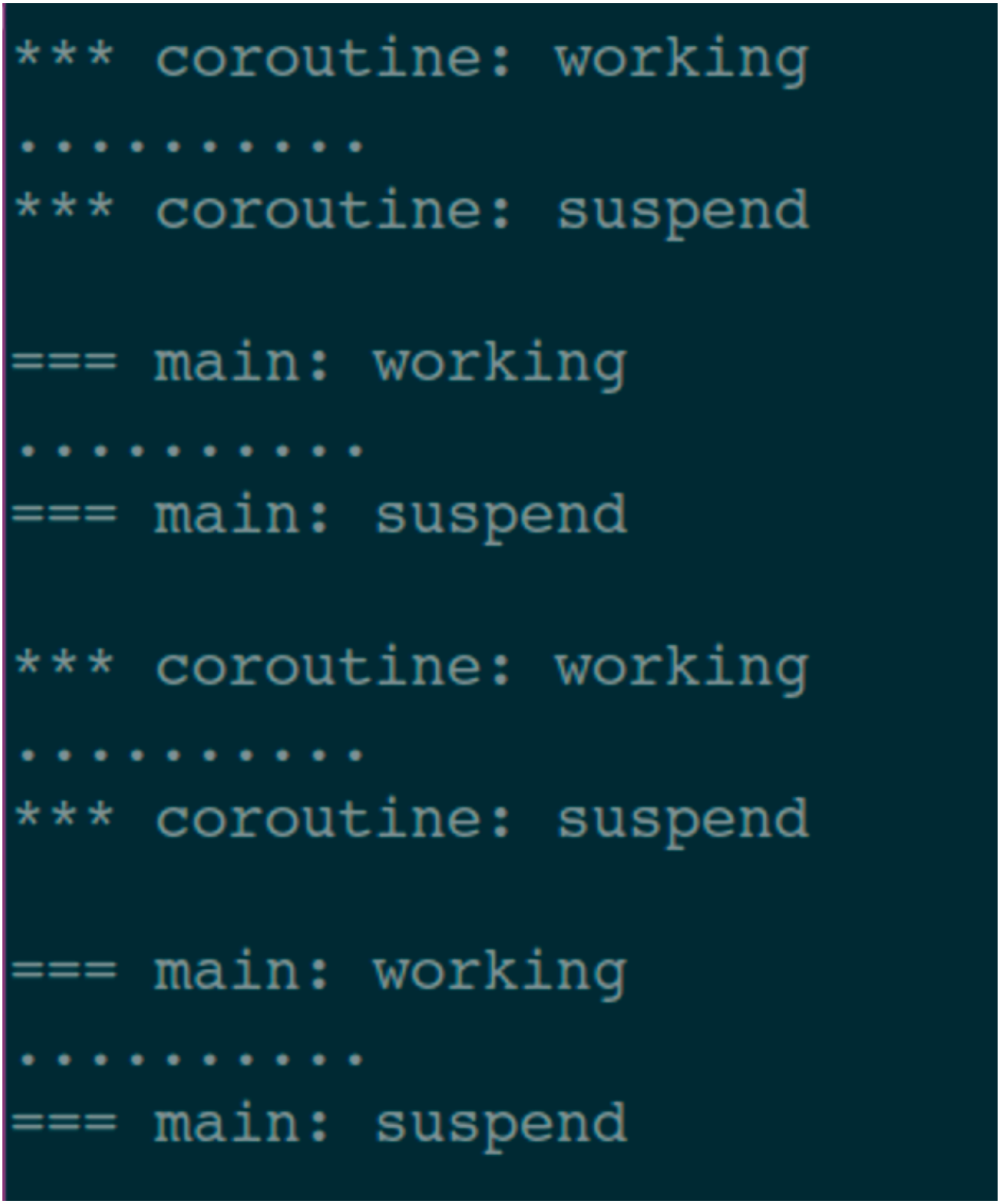
这里给出一个最最简单的模型，通过 setjmp/longjmp 来实现协程的机制，主要是目的是来理解协程的执行序列，没有解决参数和返回值的传递问题。

如果想深入研究 C 语言中的协程实现，可以看一下达夫设备这个概念，其中利用 goto 和 switch 语句来实现分支跳转，其中使用的语法比较怪异、但是合法。

```
typedef int     BOOL;
#define TRUE    1
#define FALSE   0

// 用来存储主程和协程的上下文的数据结构
typedef struct _Context_ {
    jmp_buf mainBuf;
    jmp_buf coBuf;
} Context;

// 上下文全局变量
Context gCtx;

// 恢复
#define resume() \
    if (0 == setjmp(gCtx.mainBuf)) \
    { \
        longjmp(gCtx.coBuf, 1); \
    }

// 挂起
#define yield() \
    if (0 == setjmp(gCtx.coBuf)) \
    { \
        longjmp(gCtx.mainBuf, 1); \
    }

// 在协程中执行的函数
void coroutine_function(void *arg)
{
    while (TRUE)  // 死循环
    {
        printf("\n*** coroutine: working \n");
        // 模拟耗时操作
        for (int i = 0; i < 10; ++i)
```

```
        {
            fprintf(stderr, ".");
            usleep(1000 * 200);
        }
        printf("\n*** coroutine: suspend \n");

        // 让出 CPU
        yield();
    }
}

// 启动一个协程
// 参数1：func 在协程中执行的函数
// 参数2：func 需要的参数
typedef void (*pf)(void *);
BOOL start_coroutine(pf func, void *arg)
{
    // 保存主程的跳转点
    if (0 == setjmp(gCtx.mainBuf))
    {
        func(arg); // 调用函数
        return TRUE;
    }

    return FALSE;
}

int main()
{
    // 启动一个协程
    start_coroutine(coroutine_function, NULL);

    while (TRUE) // 死循环
    {
        printf("\n=== main: working \n");

        // 模拟耗时操作
        for (int i = 0; i < 10; ++i)
        {
            fprintf(stderr, ".");
            usleep(1000 * 200);
        }

        printf("\n=== main: suspend \n");

        // 放弃 CPU，让协程执行
        resume();
    }

    return 0;
}
```

打印信息如下：



五、总结

这篇文章的重点是介绍 setjmp/longjmp 的语法和使用场景，在某些需求场景中，能达到事半功倍的效果。

当然，你还可以发挥想象力，通过执行序列的跳转来实现更加花哨的功能，一切皆有可能！

不吹嘘，不炒作，不浮夸，认真写好每一篇文章！

欢迎转发、分享给身边的技术朋友，道哥在此表示衷心的感谢！
转发的推荐语已经帮您想好了：

道哥总结的这篇总结文章，写得很用心，对我的技术提升很有帮助。好东西，要分享！

最后，祝您：面对代码，永无bug；面对生活，春暖花开！

【原创声明】

> 作者：道哥(公众号: IOT物联网小镇)
> 知乎：道哥
> B站：道哥分享
> 掘金：道哥分享
> CSDN：道哥分享

我会把十多年嵌入式开发中的项目实战经验进行输出总结！
长按下图二维码关注，关注+星标公众号，每篇文章都有干货。

推荐阅读

C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
一步步分析-如何用C实现面向对象编程
原来gdb的底层调试原理这么简单
关于加密、证书的那些事
深入LUA脚本语言，让你彻底明白调试原理