列。非本地跳转是通过 setjmp 和 longjmp 函数来提供的。

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

返回：setjmp 返回 0，longjmp 返回非零。

setjmp 函数在 env 缓冲区中保存当前调用环境，以供后面的 longjmp 使用，并返回 0。调用环境包括程序计数器、栈指针和通用目的寄存器。出于某种超出本书描述范围的原因，setjmp 返回的值不能被赋值给变量：

```
rc = setjmp(env);  /* Wrong! */
```

不过它可以安全地用在 switch 或条件语句的测试中[62]。

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

从不返回。

longjmp 函数从 env 缓冲区中恢复调用环境，然后触发一个从最近一次初始化 env 的 setjmp 调用的返回。然后 setjmp 返回，并带有非零的返回值 retval。

第一眼看过去，setjmp 和 longjmp 之间的相互关系令人迷惑。setjmp 函数只被调用一次，但返回多次：一次是当第一次调用 setjmp，而调用环境保存在缓冲区 env 中时，一次是为每个相应的 longjmp 调用。另一方面，longjmp 函数被调用一次，但从不返回。

非本地跳转的一个重要应用就是允许从一个深层嵌套的函数调用中立即返回，通常是由检测到某个错误情况引起的。如果在一个深层嵌套的函数调用中发现了一个错误情况，我们可以使用非本地跳转直接返回到一个普通的本地化的错误处理程序，而不是费力地解开调用栈。

图 8-43 展示了一个示例，说明这可能是如何工作的。main 函数首先调用 setjmp 以保存当前的调用环境，然后调用函数 foo，foo 依次调用函数 bar。如果 foo 或者 bar 遇到一个错误，它们立即通过一次 longjmp 调用从 setjmp 返回。setjmp 的非零返回值指明了错误类型，随后可以被解码，且在代码中的某个位置进行处理。

code/ecf/setjmp.c

```
1  #include "csapp.h"
2
3  jmp_buf buf;
4
5  int error1 = 0;
6  int error2 = 1;
7
8  void foo(void), bar(void);
9
```

图 8-43　非本地跳转的示例。本示例表明了使用非本地跳转来从深层嵌套的函数调用中的错误情况恢复，而不需要解开整个栈的基本框架