将对 setjmp 与 longjmp 的具体使用方法和适用的场合，进行一个非常全面的阐述。

另外请特别注意，setjmp 函数与 longjmp 函数总是组合起来使用，它们是紧密相关的一对操作，只有将它们结合起来使用，才能达到程序控制流有效转移的目的，才能按照程序员的预先设计的意图，去实现对程序中可能出现的异常进行集中处理。

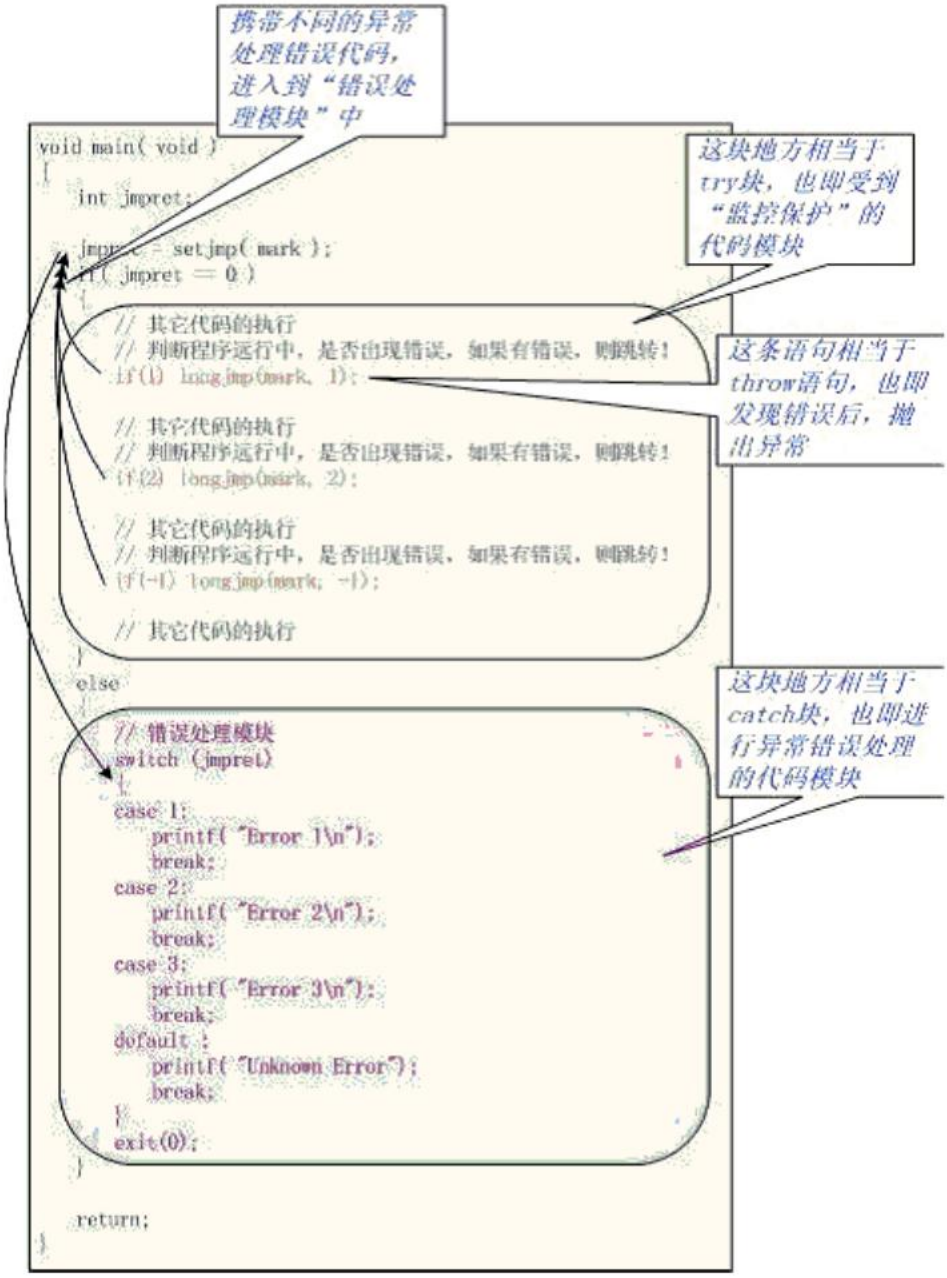
与 goto 语句的作用类似，它能实现本地的跳转

这种情况容易理解，不过还是列举出一个示例程序吧！如下：

```
void main( void )
{
int jmpret;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(1) longjmp(mark, 1);

// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(2) longjmp(mark, 2);

// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(-1) longjmp(mark, -1);

// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
```

```
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}
exit(0);
}

return;
}
```

上面的例程非常地简单，其中程序中使用到了异常处理的机制，这使得程序的代码非常紧凑、清晰，易于理解。在程序运行过程中，当异常情况出现后，控制流是进行了一个本地跳转（进入到异常处理的代码模块，是在同一个函数的内部），这种情况其实也可以用 goto 语句来予以很好的实现，但是，显然 setjmp 与 longjmp 的方式，更为严谨一些，也更为友善。程序的执行流如图 17-1 所示。



setjmp 与 longjmp 相结合，实现程序的非本地的跳转

呵呵！这就是 goto 语句所不能实现的。也正因为如此，所以才说在 C 语言中，setjmp 与 longjmp 相结合的方式，它提供了真正意义上的异常处理机制。其实上一篇文章中的那个例程，已经演示了 longjmp 函数的非本地跳转的场景。这里为了更清晰演示本地跳转与非本地跳转，这两者之间的区别，我们在上面刚才的那个例程基础上，进行很小的一点改动，代码如下：

```
void Func1()
{
```

```
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(1) longjmp(mark, 1);
}

void Func2()
{
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(2) longjmp(mark, 2);
}

void Func3()
{
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(-1) longjmp(mark, -1);
}

void main( void )
{
int jmpret;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行

// 下面的这些函数执行过程中，有可能出现异常
Func1();

Func2();

Func3();

// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
```

```
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}
exit(0);
}

return;
}
```

回顾一下，这与C++中提供的异常处理模型是不是很相近。异常的传递是可以跨越一个或多个函数。这的确为C程序员提供了一种较完善的异常处理编程的机制或手段。

setjmp和longjmp使用时，需要特别注意的事情

1、setjmp与longjmp结合使用时，它们必须有严格的先后执行顺序，也即先调用setjmp函数，之后再调用longjmp函数，以恢复到先前被保存的“程序执行点”。否则，如果在setjmp调用之前，执行longjmp函数，将导致程序的执行流变的不可预测，很容易导致程序崩溃而退出。请看示例程序，代码如下：

```
class Test
{
public:
Test()
~Test()
}obj;

//注意，上面声明了一个全局变量obj

void main( void )
{
int jmpret;

// 注意，这里将会导致程序崩溃，无条件退出
Func1();
while(1);
```

```
jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行

// 下面的这些函数执行过程中，有可能出现异常
Func1();

Func2();

Func3();

// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}
exit(0);
}

return;
}
```

上面的程序运行结果，如下：

```
构造对象
Press any key to continue
```

的确，上面程序崩溃了，由于在Func1()函数内，调用了longjmp，但此时程序还没有调用setjmp来保存一个程序执行点。因此，程序的执行流变的不可预测。这样导致的程序后果是非常严重的，例如说，上面的程序中，有一个对象

被构造了，但程序崩溃退出时，它的析构函数并没有被系统来调用，得以清除一些必要的资源。所以这样的程序是非常危险的。（另外请注意，上面的程序是一个 C++程序，所以大家演示并测试这个例程时，把源文件的扩展名改为 xxx.cpp）。

2、除了要求先调用 setjmp 函数，之后再调用 longjmp 函数（也即 longjmp 必须有对应的 setjmp 函数）之外。另外，还有一个很重要的规则，那就是 longjmp 的调用是有一定域范围要求的。这未免太抽象了，还是先看一个示例，如下：

```
int Sub_Func()
{
int jmpret, be_modify;

be_modify = 0;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}

//注意这一语句，程序有条件地退出
if (be_modify==0) exit(0);
}

return jmpret;
}
```

```
void main( void )
{
Sub_Func();

// 注意，虽然 longjmp 的调用是在 setjmp 之后，但是它超出了 setjmp 的作用范围。
longjmp(mark, 1);
}
```

如果你运行或调试（单步跟踪）一下上面程序，发现它真是挺神奇的，居然 longjmp 执行时，程序还能够返回到 setjmp 的执行点，程序正常退出。但是这就说明了上面的这个例程的没有问题吗？我们对这个程序小改一下，如下：

```
int Sub_Func()
{
// 注意，这里改动了一点
int be_modify, jmpret;

be_modify = 0;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}
```

```
//注意这一语句，程序有条件地退出
if (be_modify==0) exit(0);
}

return jmpret;
}

void main( void )
{
Sub_Func();

// 注意，虽然 longjmp 的调用是在 setjmp 之后，但是它超出了 setjmp 的作用范围。
longjmp(mark, 1);
}
```

运行或调试（单步跟踪）上面的程序，发现它崩溃了，为什么？这就是因为，“在调用 setjmp 的函数返回之前，调用 longjmp，否则结果不可预料”（这在上一篇文章中已经提到过，MSDN 中做了特别的说明）。为什么这样做会导致不可预料？其实仔细想想，原因也很简单，那就是因为，当 setjmp 函数调用时，它保存的程序执行点环境，只应该在当前的函数作用域以内（或以后）才会有效。如果函数返回到了上层（或更上层）的函数环境中，那么 setjmp 保存的程序的环境也将会无效，因为堆栈中的数据此时将可能发生覆盖，所以当然会导致不可预料的执行后果。

3、不要假象寄存器类型的变量将总会保持不变。在调用 longjmp 之后，通过 setjmp 所返回的控制流中，例程中寄存器类型的变量将不会被恢复。（MSDN 中做了特别的说明，上一篇文章中，这也已经提到过）。寄存器类型的变量，是指为了提高程序的运行效率，变量不被保存在内存中，而是直接被保存在寄存器中。寄存器类型的变量一般都是临时变量，在 C 语言中，通过 register 定义，或直接嵌入汇编代码的程序。这种类型的变量一般很少采用，所以在使用 setjmp 和 longjmp 时，基本上不用考虑到这一点。

4、MSDN 中还做了特别的说明，“在 C++程序中，小心对 setjmp 和 longjmp 的使用，因为 setjmp 和 longjmp 并不能很好地支持 C++中面向对象的语义。因此在 C++程序中，使用 C++提供的异常处理机制将会更加安全。”虽然说 C++能非常好的兼容 C，但是这并非是 100%的完全兼容。例如，这里就是一个很好的例子，在 C++ 程序中，它不能很好地与 setjmp 和 longjmp 和平共处。在后面的一些文章中，有关专门讨论 C++如何兼容支持 C 语言中的异常处理机制时，会做详细深入的研究，这里暂且跳过。