**C语言中一种更优雅的异常处理机制**

|  |
| --- |
| 上一篇文章对C语言中的goto语句进行了较深入的阐述，实际上goto语句是面向过程与面向结构化程序语言中，进行异常处理编程的最原始的支持形 式。后来为了更好地、更方便地支持异常处理编程机制，使得程序员在C语言开发的程序中，能写出更高效、更友善的带有异常处理机制的代码模块来。于是，C语 言中出现了一种更优雅的异常处理机制，那就是setjmp()函数与longjmp()函数。  　　实际上，这种异常处理的机制不是C语言中自身的一部分，而是在C标准库中实现的两个非常有技巧的库函数，也许大多数C程序员朋友们对它都很熟悉，而 且，通过使用setjmp()函数与longjmp()函数组合后，而提供的对程序的异常处理机制，以被广泛运用到许多C语言开发的库系统中，如jpg解 析库，加密解密库等等。  　　也许C语言中的这种异常处理机制，较goto语句相比较，它才是真正意义上的、概念上比较彻底的，一种异常处理机制。作风一向比较严谨、喜欢刨根问底的主人公阿愚当然不会放 弃对这种异常处理机制进行全面而深入的研究。下面一起来看看。  **setjmp函数有何作用？**  　　前面刚说了，setjmp是C标准库中提供的一个函数，它的作用是保存程序当前运行的一些状态。它的函数原型如下：  int setjmp( jmp\_buf env );  　　这是MSDN中对它的评论，如下：  　　setjmp函数用于保存程序的运行时的堆栈环境，接下来的其它地方，你可以通过调用longjmp函数来恢复先前被保存的程序堆栈环境。当setjmp和longjmp组合一起使用时，它们能提供一种在程序中实现“非本地局部跳转”（"non-local goto"）的机制。并且这种机制常常被用于来实现，把程序的控制流传递到错误处理模块之中；或者程序中不采用正常的返回（return）语句，或函数的正常调用等方法，而使程序能被恢复到先前的一个调用例程（也即函数）中。  　 　对setjmp函数的调用时，会保存程序当前的堆栈环境到env参数中；接下来调用longjmp时，会根据这个曾经保存的变量来恢复先前的环境，并且 当前的程序控制流，会因此而返回到先前调用setjmp时的程序执行点。此时，在接下来的控制流的例程中，所能访问的所有的变量（除寄存器类型的变量以 外），包含了longjmp函数调用时，所拥有的变量。  　　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，请使用C++提供的异常处理机制。  　　好了，现在已经对setjmp有了很感性的了解，暂且不做过多评论，接着往下看longjmp函数。  **longjmp函数有何作用？**  　　同样，longjmp也是C标准库中提供的一个函数，它的作用是用于恢复程序执行的堆栈环境，它的函数原型如下：  void longjmp( jmp\_buf env, int value );  　　这是MSDN中对它的评论，如下：  　　longjmp函数用于恢复先前程序中调用的setjmp函数时所保存的堆栈环境。setjmp和longjmp组合一起使用时，它们能提供一种在程序中实现“非本地局部跳转”（"non-local goto"）的机制。并且这种机制常常被用于来实现，把程序的控制流传递到错误处理模块，或者不采用正常的返回（return）语句，或函数的正常调用等方法，使程序能被恢复到先前的一个调用例程（也即函数）中。  　 　对setjmp函数的调用时，会保存程序当前的堆栈环境到env参数中；接下来调用longjmp时，会根据这个曾经保存的变量来恢复先前的环境，并且 因此当前的程序控制流，会返回到先前调用setjmp时的执行点。此时，value参数值会被setjmp函数所返回，程序继续得以执行。并且，在接下来 的控制流的例程中，它所能够访问到的所有的变量（除寄存器类型的变量以外），包含了longjmp函数调用时，所拥有的变量；而寄存器类型的变量将不可预 料。setjmp函数返回的值必须是非零值，如果longjmp传送的value参数值为0，那么实际上被setjmp返回的值是1。  　　在调用setjmp的函数返回之前，调用longjmp，否则结果不可预料。  　　在使用longjmp时，请遵守以下规则或限制： 　　· 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。 　　· 不要使用longjmp函数，来实现把控制流，从一个中断处理例程中传出，除非被捕获的异常是一个浮点数异常。在后一种情况下，如果程序通过调用 \_fpreset函数，来首先初始化浮点数包后，它是可以通过longjmp来实现从中断处理例程中返回。 　　· 在C++程序中，小心对setjmp和longjmp的使用，应为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。 把setjmp和longjmp组合起来，原来它这么厉害！ 　　现在已经对setjmp和longjmp都有了很感性的了解，接下来，看一个示例，并从这个示例展开分析，示例代码如下（来源于MSDN）：  /\* FPRESET.C: This program uses signal to set up a \* routine for handling floating-point errors. \*/  #include <stdio.h> #include <signal.h> #include <setjmp.h> #include <stdlib.h> #include <float.h> #include <math.h> #include <string.h>  jmp\_buf mark; /\* Address for long jump to jump to地址远跳的位置 \*/ int fperr; /\* Global error number 全局错误值 \*/  void \_\_cdecl fphandler( int sig, int num ); /\* Prototypes原型 \*/ void fpcheck( void );  void main( void ) { double n1, n2, r; int jmpret; /\* Unmask all floating-point exceptions. 揭开所有浮点异常。 \*/ \_control87( 0, \_MCW\_EM );  /\* Set up floating-point error handler. The compiler will generate a warning because it expects signal-handling functions to take only one argument.  设置浮点错误处理程序。编译器将生成一个警告,因为它预计信号处理函数只有//一个参数。 \*/  if( signal( SIGFPE, fphandler ) == SIG\_ERR )  { fprintf( stderr, "Couldn't set SIGFPE/n" ); abort(); //异常终止一个进程 }  /\* Save stack environment for return in case of error. First time through, jmpret is 0, so true conditional is executed.If an error occurs, jmpret will be set to -1 and false conditional will be executed.  为返回的错误保存堆栈环境。第一次调用setjmp返回0,所以此时jmpret是0, 正确的条件被执行。如果出现错误,jmpret将设置为1,错误的条件将被执行 \*/  // 注意，下面这条语句的作用是，保存程序当前运行的状态 jmpret = setjmp( mark ); if( jmpret == 0 ) { printf( "Test for invalid operation - " ); printf( "enter two numbers: " ); scanf( "%lf %lf", &n1, &n2 );  // 注意，下面这条语句可能出现异常， // 如果从终端输入的第2个变量是0值的话 r = n1 / n2; /\* This won't be reached if error occurs. 如果发生错误就不会达到这里（出现错误，该句不会被运行）。 \*/ printf( "/n/n%4.3g / %4.3g = %4.3g/n", n1, n2, r );  r = n1 \* n2; /\* This won't be reached if error occurs. \*/ printf( "/n/n%4.3g \* %4.3g = %4.3g/n", n1, n2, r ); } else fpcheck(); } /\* fphandler handles SIGFPE (floating-point error) interrupt. Note that this prototype accepts two arguments and that the prototype for signal in the run-time library expects a signal handler to have only one argument.  fphandler处理SIGFPE(浮点错误)中断。注意,这个原型接受两个参数,信号在运行时库的原型预计信号处理器只有一个参数。 \* \* The second argument in this signal handler allows processing of  \_FPE\_INVALID, \_FPE\_OVERFLOW, \_FPE\_UNDERFLOW, and   \_FPE\_ZERODIVIDE, all of which are Microsoft-specific symbols that augment the information provided by SIGFPE. The compiler will generate a warning, which is harmless and expected.  第二个参数的信号处理程序可以处理\_FPE\_INVALID, \_FPE\_OVERFLOW, \_FPE\_UNDERFLOW, and   \_FPE\_ZERODIVIDE.  微软的特定符号,增强我所提供的资料。编译器将生成一个警告,这是无害的预期。  \*/ void fphandler( int sig, int num ) { /\* Set global for outside check since we don't want to do I/O in the handler.  设置全局外部检查,因为我们不想做I / O的处理程序。 \*/ fperr = num; /\* Initialize floating-point package. 初始化浮点包\*/ \_fpreset(); //重新设置浮点运算系统  /\* Restore calling environment and jump back to setjmp. Return -1 so that setjmp will return false for conditional test.  恢复调用环境和跳回setjmp。返回1,setjmp将返回错误的条件测试。 \*/ // 注意，下面这条语句的作用是，恢复先前setjmp所保存的程序状态 longjmp( mark, -1 ); } void fpcheck( void ) { char fpstr[30]; switch( fperr ) { case \_FPE\_INVALID: strcpy( fpstr, "Invalid number" ); break; case \_FPE\_OVERFLOW: strcpy( fpstr, "Overflow" );  break; case \_FPE\_UNDERFLOW: strcpy( fpstr, "Underflow" ); break; case \_FPE\_ZERODIVIDE: strcpy( fpstr, "Divide by zero" ); break; default: strcpy( fpstr, "Other floating point error" ); break; } printf( "Error %d: %s/n", fperr, fpstr ); }  程序的运行结果如下： Test for invalid operation - enter two numbers: 1 2  1 / 2 = 0.5  1 \* 2 = 2  　　上面的程序运行结果正常。另外程序的运行结果还有一种情况，如下： Test for invalid operation - enter two numbers: 1 0 Error 131: Divide by zero  呵呵！程序运行过程中出现了异常（被0除），并且这种异常被程序预先定义的异常处理模块所捕获了。厉害吧！可千万别轻视，这可以C语言编写的程序。  **分析setjmp和longjmp**  　　现在，来分析上面的程序的执行过程。当然，这里主要分析在异常出现的情况下，程序运行的控制转移流程。由于文章篇幅有限，分析时，我们简化不相关的代码，这样更也易理解控制流的执行过程。如下图所示。  　　呵呵！现在是否对程序的执行流程一目了然，其中最关键的就是setjjmp和longjmp函数的调用处理。我们分别来分析之。  　　当程序运行到第②步时，调用setjmp函数，这个函数会保存程序当前运行的一些状态信息，主要是一些系统寄存器的值，如ss，cs，eip， eax，ebx，ecx，edx，eflags等寄存器，其中尤其重要的是eip的值，因为它相当于保存了一个程序运行的执行点。这些信息被保存到 mark变量中，这是一个C标准库中所定义的特殊结构体类型的变量。  　　调用setjmp函数保存程序状态之后，该函数返回0值，于是接下来程序执行到第③步和第④步中。在第④步中语句执行时，如果变量n2为0值，于是便 引发了一个浮点数计算异常，，导致控制流转入fphandler函数中，也即进入到第⑤步。  　　然后运行到第⑥步，调用longjmp函数，这个函数内部会从先前的setjmp所保存的程序状态，也即mark变量中，来恢复到以前的系统寄存器的 值。于是便进入到了第⑦步，注意，这非常有点意思，实际上，通过longjmp函数的调用后，程序控制流（尤其是eip的值）再次戏剧性地进入到了 setjmp函数的处理内部中，但是这一次setjmp返回的值是longjmp函数调用时，所传入的第2个参数，也即-1，因此程序接下来进入到了第⑧ 步的执行之中。  **总结**  　　与goto语句不同，在C语言中，setjmp()与longjmp()的组合调用，为程序员提供了一种更优雅的异常处理机制。它具有如下特点：  　　 （1） goto只能实现本地跳转，而setjmp()与longjmp()的组合运用，能有效的实现程序控制流的非本地（远程）跳转；  　　 （2） 与goto语句不同，setjmp()与longjmp()的组合运用，提供了真正意义上的异常处理机制。例如，它能有效定义受监控保护的模块区域（类似 于C++中try关键字所定义的区域）；同时它也能有效地定义异常处理模块（类似于C++中catch关键字所定义的区域）；还有，它能在程序执行过程 中，通过longjmp函数的调用，方便地抛出异常（类似于C++中throw关键字）。  　　现在，相信大家已经对在C语言中提供的这种异常处理机制有了很全面地了解。但是我们还没有深入它研究它，下一篇文章中继续探讨吧！go！ |