异常和错误处理（基于 Delphi/VCL）

有人在看了我的"如何将界面代码和功能代码分离（基于 Delphi/VCL）"之后，提到一个问题，就是如何对服务端的类的错误进行处理。

在基于函数的结构中，我们一般使用函数返回值来标明函数是否成功执行，并给出错误类型等信息。于是就会有如下形式的代码：

```
RetVal := SomeFunctionToOpenFile();

if RetVal = E_SUCCESSED then
    ......
else if RetVal = E_FILENOTFOUND then
    ......
else if RetVal = E_FILEFORMATERR then
    ......
else then
    ......
```

使用返回错误代码的方法是非常普遍的，但是使用这样的方法存在 2 个问题：

1、造成冗长、繁杂的分支结构（大量的 if 或 case 语句），使得控制流程变得复杂

2、可能会有没有被处理的错误（函数调用者如果不判断返回值的话）

而异常是对于错误处理的面向对象的解决方案。它可以报告错误，但需要知道的是，并非由于错误而引发了异常，而仅仅是因为使用了 raise。

在 Object Pascal 中，抛出异常使用的是 raise 保留字。在任何时候（即使没有错误发生），raise 都将会导致异常的发生。

异常可以使得代码从异常发生处立刻返回，从而保护其下面的敏感代码不会得到执行。通过异常从函数返回和正常从函数返回（执行到函数末尾或执行了 Exit）对于抛出异常的函数本

身来说是没有什么区别的。区别在于调用者处，通过异常返回后，执行权会被调用者的 try...except 块所捕获（如果它们存在的话）。如果调用者处没有 try...except 块的话，将不会继续执行后续语句，而是返回更上层的调用者，直至找到能够处理该异常的 try...except 块。异常被处理后，将继续执行 try...except 块之后的语句，控制权就被留在了处理异常的这一层。当异常处理程序感觉对异常的处理还不够完整时，需要更上层调用者继续处理，可以重新抛出异常（使用简单的 raise;即可）将控制权交给更上层调用者。

如果根本就没有预设 try...except 块，则最终异常会被最外层的封装整个程序的 VCL 的 try...except 块所捕获。

因此，不会有不被处理的异常，换句话说，也就是不会有不被处理的错误（虽然错误和异常并不能划等号）。这也是异常机制比使用返回错误代码方法的优越之处。另外，异常被抛出后，其控制流程的走向非常清晰明了，不会造成流程失去控制的情况。

举个例子说明异常的工作机制，假设我们要打开某种特定格式的文件：

先定义两个异常类（从 Exception 继承）

```
EFileNotFound = class(Exception);
EFileFormatErr = class(Exception);
```

假设 Form1 上有一个按纽，按下按纽即打开文件：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    try
        ToOpenFile();
    except
        on EFileNotFound do
            ShowMessage('Sorry, I can''t find the file');
        on EFileFormatErr do
            ShowMessage('Sorry, the file is not the one I want');
        on E:Exception do
```

```
            ShowMessage(E.Message);
        end;
    end;
```

以及打开文件的功能函数：

```
procedure ToOpenFile;
var RetVal:Integer;
begin
    //Some code to openfile
    RetVal := -1; //open failed

    if RetVal = 0 then //success
        Exit

    else if RetVal = -1 then
        Raise EFileNotFound.Create('File not found')

    else if RetVal = -2 then
        Raise EFileFormatErr.Create('File format error')

    else //other error
        Raise Exception.Create('Unknown error');
end;
```

程序中 TForm1.Button1Click 调用 ToOpenFile，并预设了对 ToOpenFile 可能抛出的异常处理的 try...except。当然，也可以对 TForm1.Button1Click 的异常处理代码进行简化：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    try
        ToOpenFile();
    except
        ShowMessage('Open file failed');
```

```
    end;
end;
```

使用异常解决了使用返回错误代码方法存在的问题，当然，使用异常也不是没有代价的。异常会增加程序的负担，因此滥用异常也是不可取的。写若干 try...except 和写数以千计的 try...except 之间是有很大区别的。用 Chalie Calverts 的话来说就是："在似乎有用的时候，就应该使用 try...except 块。但是要试着让自己对这种技术的热情不要太过头"。

另外，Object Pascal 引入了独特的 try...finally 结构。前面我说过，通过异常从函数返回和正常从函数返回是没有什么区别的。因此，函数中的栈中的局部对象，会自动得到释放，而堆中的对象则不会。而然，Object Pascal 的对象模型是基于引用的，其存在于堆中，而非栈中。因此，有时我们在通过异常从函数返回之前需要清理一些局域的对象资源。try...finally 正是解决这个问题的。

我改写了以上的 ToOpenFile 的代码，这次让 ToOpenFile 过程中使用了一些资源，并在异常发生后（或者不发生）从函数返回前都会释放这些资源：

```
procedure ToOpenFile;
var RetVal: Integer;
    Stream: TStream;
begin
    //Some code to openfile
    Stream := TStream.Create;
    RetVal := -1; //open failed

    try
        if RetVal = 0 then //success
            Exit
        else if RetVal = -1 then
            Raise EFileNotFound.Create('File not found')
        else if RetVal = -2 then
            Raise EFileFormatErr.Create('File format error')
```

```
        else //other error
            Raise Exception.Create('Unknown error');
    finally
        Stream.Free;
    end;
end;
```

单步执行以上代码，可以看出，即使当 RetVal 的值为 0 时，执行 Exit 后，仍然会执行 finally 中的代码，然后再从函数返回。由此保证了局部资源的正确释放。

try...except 和 try...finally 的用途和使用场合是不同的，而很多初学者会将它们混淆。以下是笔者的一些个人认识：try...except 一般用于调用者处捕获所调用的函数所抛出的异常并进行处理。而 try...finally 一般用于抛出异常的函数本身进行一些资源清理工作。

面向对象编程提供了"异常"这种错误处理的方案。善而用之，会对我们的工作有好处，可以显著改善所编写代码的质量。

再谈异常——谈 C++与 Object Pascal 中的构造函数与异常

我们知道，类的构造函数是没有返回值的，如果构造函数构造对象失败，不可能依靠返回错误代码。那么，在程序中如何标识构造函数的失败呢？最"标准"的方法就是：抛出一个异常。

构造函数失败，意味着对象的构造失败，那么抛出异常之后，这个"半死不活"的对象会被如何处理呢？这就是本文的主题。

在 C++中，构造函数抛出异常后，析构函数不会被调用。这是合理的，因为此时对象并没有被完整构造。也就是说，如果构造函数已经做了一些诸如分配内存、打开文件等操作的话，那么类需要有自己的成员来记住做过哪些动作。在 C++中，经典的解决方案是使用 STL 的标准类 auto_ptr，这在每一本经典 C++著作中都有介绍，我在这里就不多说了。在这里，我想再介绍一种"非常规"的方式，其思想就是避免在构造函数中抛出异常。我们可以在类中增加一个 Init(); 以及 UnInit();成员函数用于进行容易产生错误的资源分配工作，而真正的构造函数中先将所有成员置为 NULL，然后调用 Init(); 并判断其返回值（或者捕捉 Init()抛出的异常），如果 I

nit();失败了，则在构造函数中调用 UnInit(); 并设置一个标志位表明构造失败。UnInit()中按照成员是否为 NULL 进行资源的释放工作。示例代码如下：

```
class A
{
private:
    char* str;
    int failed;
public:
    A();
    ~A();
    int Init();
    int UnInit();
    int Failed();
};

A::A()
{
    str = NULL;
    try
    {
        Init();
        failed = 0;
    }
    catch(...)
    {
        failed = 1;
        UnInit();
    }
}
```

```
A::~A()
{
    UnInit();
}

int A::Init()
{
    str = new char[10];
    strcpy(str, "ABCDEFGHI");
    throw 10;
    return 1;
}

int A::UnInit()
{
    if (!str)
    {
        delete []str;
        str = NULL;
    }

    printf("Free Resource\n");
    return 1;
}

int A::Failed()
{
    return failed;
}

int main(int argc, char* argv[])
{
```

```
    A* a = new A;
    if ( a->Failed() )
        printf("failed\n");
    else
        printf("succeeded\n");

    delete a;
    getchar();
    return 0;
}
```

你会发现，在 int A::Init()中包含了 throw 10;的代码（产生一个异常，模拟错误的发生），执行结果是：

```
Free Resource
failed
Free Resource
```

虽然 UnInit();被调用了两次，但是由于 UnInit();中做了判断（if (!str)），因此不会发生错误。而如果没有发生异常（去掉 int A::Init()中的 throw 10;代码），执行结果是：

```
Succeeded
Free Resource
```

和正常的流程没有任何区别。

在 Object Pascal（Delphi/VCL）中，这个问题就变得非常的简单了，因为 OP 对构造函数的异常的处理与 C++不同，在 Create 时抛出异常后，编译器会自动调用析构函数 Destroy，并且会判断哪些资源被分配了，实行自动回收。因此，其代码也变得非常简洁，如下：

```
type A = class
private
    str : PChar;
public
    constructor Create();
```

```
    destructor Destroy(); override;
end;

constructor A.Create();
begin
    str := StrAlloc(10);
    StrCopy(str, 'ABCDEFGHI');
    raise Exception.Create('error');
end;

destructor A.Destroy();
begin
    StrDispose(str);
    WriteLn('Free Resource');
end;

var oa : A;
    i : integer;
begin
    try
        oa := A.Create();
        WriteLn('Succeeded');
        oa.Free();
    except
        oa := nil;
        WriteLn('Failed');
    end;

    Read(i);
end.
```

在这段代码中，如果构造函数抛出异常（即 Create 中含有 raise Exception.Create ('error');），执行的结果是：

Free Resource

Failed

此时的“Free Resource”输出是由编译器自动调用析构函数所产生的。而如果构造函数正常返回（即不抛出异常），则执行结果是：

Succeeded

Free Resource

此时的“Free Resource”输出是由 oa.Free()的调用产生的。

综上，C++与 Object Pascal 对于构造函数抛出异常后的不同处理方式，其实正是两种语言的设计思想的体现。C++秉承 C 的风格，注重效率，一切交给程序员来掌握，编译器不作多余动作。Object Pascal 继承 Pascal 的风格，注重程序的美学意义（不可否认，Pascal 代码是全世界最优美的代码），编译器帮助程序员完成复杂的工作。两种语言都有存在的理由，都有存在的必要！而掌握它们之间的差别，能让你更好地控制它们，达到自由的理想王国。