12.异常

1. 异常的概念及使用

1.1 异常的概念

- 异常处理机制允许程序中独立开发的部分能够在运行时就出现的问题进行通信并做出相应的处理，异常使得我们能够将问题的检测与解决问题的过程分开，程序的一部分负责检测问题的出现，然后解决问题的任务传递给程序的另一部分，检测环节无须知道问题的处理模块的所有细节。
- C语言主要通过错误码的形式处理错误，错误码本质就是对错误信息进行分类编号，拿到错误码以后还要去查询错误信息，比较麻烦。异常时抛出一个对象，这个对象可以函数更全面的各种信息。

1.2 异常的抛出和捕获

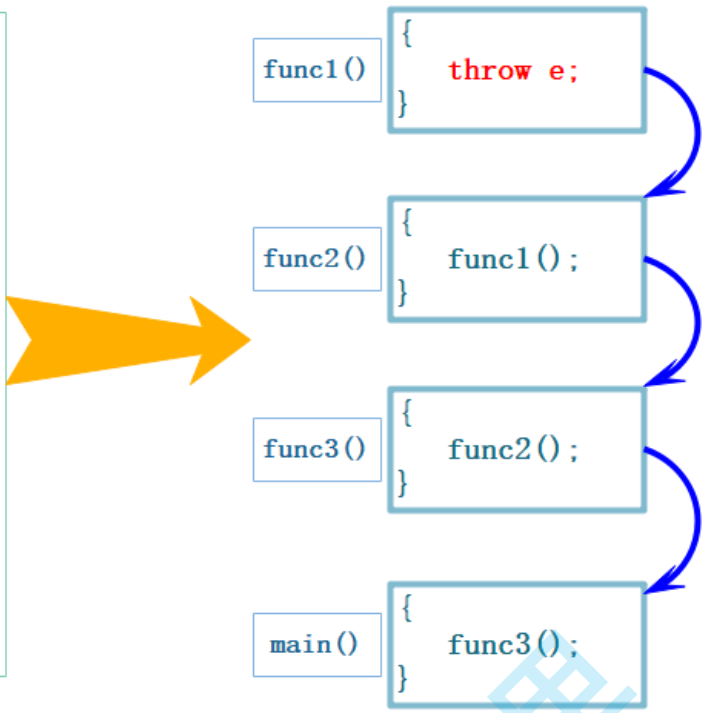
- 程序出现问题时，我们通过抛出(throw)一个对象来引发一个异常，该对象的类型以及当前的调用链决定了应该由哪个catch的处理代码来处理该异常。
- 被选中的处理代码是调用链中与该对象类型匹配且离抛出异常位置最近的那一个。根据抛出对象的类型和内容，程序的抛出异常部分告知异常处理部分到底发生了什么错误。
- 当throw执行时，throw后面的语句将不再被执行。程序的执行从throw位置跳到与之匹配的catch模块，catch可能是同一函数中的一个局部的catch，也可能是调用链中另一个函数中的catch，控制权从throw位置转移到了catch位置。这里还有两个重要的含义：1、沿着调用链的函数可能提早退出。2、一旦程序开始执行异常处理程序，沿着调用链创建的对象都将销毁。
- 抛出异常对象后，会生成一个异常对象的拷贝，因为抛出的异常对象可能是一个局部对象，所以会生成一个拷贝对象，这个拷贝的对象会在catch子句后销毁。（这里的处理类似于函数的传值返回）

1.3 栈展开

- 抛出异常后，程序暂停当前函数的执行，开始寻找与之匹配的catch子句，首先检查throw本身是否在try块内部，如果在则查找匹配的catch语句，如果有匹配的，则跳到catch的地方进行处理。
- 如果当前函数中没有try/catch子句，或者有try/catch子句但是类型不匹配，则退出当前函数，继续在外层调用函数链中查找，上述查找的catch过程被称为栈展开。
- 如果到达main函数，依旧没有找到匹配的catch子句，程序会调用标准库的 `terminate` 函数终止程序。
- 如果找到匹配的catch子句处理后，catch子句代码会继续执行。

有三个函数
func1(),func2(),func3(),在func2()中调用
func1(),func3()中调用
func2(),main()中调用
func3()，并在func1()抛出一个异常，在main()用catch语句捕获。

栈展开过程如下：
首先检查throw本身是否在try块内部，如果是再查找匹配的catch语句。如果有匹配的，则处理。
没有则退出当前函数栈，继续在调用函数的栈中进行查找，不断重复上述过程，若到达main函数的栈，依旧没有匹配的，则终止程序




```
double Divide(int a, int b)
{
    try
    {
        // 当b == 0时抛出异常
        if (b == 0)
        {
            string s("Divide by zero condition!");
            throw s;
        }
        else
        {
            return ((double)a / (double)b);
        }
    }
    catch (int errid)
    {
        cout << errid << endl;
    }

    return 0;
}

void Func()
{
    int len, time;
    cin >> len >> time;
```

```
28
29      try
30      {
31          cout << Divide(len, time) << endl;
32      }
33      catch (const char* errmsg)
34      {
35          cout << errmsg << endl;
36      }
37
38      cout <<__FUNCTION__<<":" << __LINE__ << "行执行" << endl;
39  }
40
41  int main()
42  {
43      while (1)
44      {
45          try
46          {
47              Func();
48          }
49          catch (const string& errmsg)
50          {
51              cout << errmsg << endl;
52          }
53      }
54
55      return 0;
56  }
57
```

1.4 查找匹配的处理代码

- 一般情况下抛出对象和catch是类型完全匹配的，如果有多个类型匹配的，就选择离他位置更近的那个。
- 但是也有一些例外，允许从非常量向常量的类型转换，也就是权限缩小；允许数组转换成指向数组元素类型的指针，函数被转换成指向函数的指针；允许从派生类向基类类型的转换，这个点非常实用，实际中继承体系基本都是用这个方式设计的。
- 如果到main函数，异常仍旧没有被匹配就会终止程序，不是发生严重错误的情况下，我们是不期望程序终止的，所以一般main函数中最后都会使用catch(...)，它可以捕获任意类型的异常，但是是不知道异常错误是什么。

```
1  #include<thread>
```

```cpp

// 一般大型项目程序才会使用异常，下面我们模拟设计一个服务的几个模块
// 每个模块的继承都是Exception的派生类，每个模块可以添加自己的数据
// 最后捕获时，我们捕获基类就可以
class Exception
{
public:
    Exception(const string& errmsg, int id)
            :_errmsg(errmsg)
            , _id(id)
    {}

    virtual string what() const
    {
            return _errmsg;
    }

    int getid() const
    {
            return _id;
    }

protected:
    string _errmsg;
    int _id;
};

class SqlException : public Exception
{
public:
    SqlException(const string& errmsg, int id, const string& sql)
            :Exception(errmsg, id)
            , _sql(sql)
    {}

    virtual string what() const
    {
            string str = "SqlException:";
            str += _errmsg;
            str += "->";
            str += _sql;
            return str;
    }
private:
    const string _sql;
};

```

```cpp
class CacheException : public Exception
{
public:
    CacheException(const string& errmsg, int id)
            :Exception(errmsg, id)
    {}

    virtual string what() const
    {
            string str = "CacheException:";
            str += _errmsg;
            return str;
    }
};

class HttpException : public Exception
{
public:
    HttpException(const string& errmsg, int id, const string& type)
            :Exception(errmsg, id)
            , _type(type)
    {}

    virtual string what() const
    {
            string str = "HttpException:";
            str += _type;
            str += ":";
            str += _errmsg;
            return str;
    }

private:
    const string _type;
};

void SQLMgr()
{
    if (rand() % 7 == 0)
    {
            throw SqlException("权限不足", 100, "select * from name = '张三'");
    }
    else
    {
            cout << "SQLMgr 调用成功" << endl;
    }
}
```

```
96
97  void CacheMgr()
98  {
99      if (rand() % 5 == 0)
100     {
101             throw CacheException("权限不足", 100);
102     }
103     else if (rand() % 6 == 0)
104     {
105             throw CacheException("数据不存在", 101);
106     }
107     else
108     {
109             cout << "CacheMgr 调用成功" << endl;
110     }
111
112     SQLMgr();
113 }
114
115 void HttpServer()
116 {
117     if (rand() % 3 == 0)
118     {
119             throw HttpException("请求资源不存在", 100, "get");
120     }
121     else if (rand() % 4 == 0)
122     {
123             throw HttpException("权限不足", 101, "post");
124     }
125     else
126     {
127             cout << "HttpServer调用成功" << endl;
128     }
129
130     CacheMgr();
131 }
132
133
134 int main()
135 {
136     srand(time(0));
137
138     while (1)
139     {
140             this_thread::sleep_for(chrono::seconds(1));
141
142             try
```

```
143             {
144                     HttpServer();
145             }
146             catch (const Exception& e) // 这里捕获基类，基类对象和派生类对象都可以被捕获
147             {
148                     cout << e.what() << endl;
149             }
150             catch (...)
151             {
152                     cout << "Unkown Exception" << endl;
153             }
154     }
155
156     return 0;
157 }
```

1.5 异常重新抛出

有时catch到一个异常对象后，需要对错误进行分类，其中的某种异常错误需要进行特殊的处理，其他错误则重新抛出异常给外层调用链处理。捕获异常后需要重新抛出，直接`throw;`就可以把捕获的对象直接抛出。

```
1  // 下面程序模拟展示了聊天时发送消息，发送失败补货异常，但是可能在
2  // 电梯地下室等场景手机信号不好，则需要多次尝试，如果多次尝试都发
3  // 送不出去，则就需要捕获异常再重新抛出，其次如果不是网络差导致的
4  // 错误，捕获后也要重新抛出。
5  void _SeedMsg(const string& s)
6  {
7      if (rand() % 2 == 0)
8      {
9          throw HttpException("网络不稳定，发送失败", 102, "put");
10     }
11     else if (rand() % 7 == 0)
12     {
13         throw HttpException("你已经不是对象的好友，发送失败", 103, "put");
14     }
15     else
16     {
17         cout << "发送成功" << endl;
18     }
19 }
20
21 void SendMsg(const string& s)
22 {
```

```
23      // 发送消息失败，则再重试3次
24      for (size_t i = 0; i < 4; i++)
25      {
26          try
27          {
28              _SeedMsg(s);
29              break;
30          }
31          catch (const Exception& e)
32          {
33              // 捕获异常，if中是102号错误，网络不稳定，则重新发送
34              // 捕获异常，else中不是102号错误，则将异常重新抛出
35
36              if (e.getid() == 102)
37              {
38                  // 重试三次以后否失败了，则说明网络太差了，重新抛出异常
39                  if (i == 3)
40                          throw;
41
42                  cout << "开始第" << i + 1 << "重试" << endl;
43              }
44              else
45              {
46                  throw;
47              }
48          }
49      }
50  }
51
52  int main()
53  {
54      srand(time(0));
55
56      string str;
57      while (cin >> str)
58      {
59          try
60          {
61              SendMsg(str);
62          }
63          catch (const Exception& e)
64          {
65              cout << e.what() << endl << endl;
66          }
67          catch (...)
68          {
69              cout << "Unkown Exception" << endl;
```

```
        }
    }

    return 0;
}
```

1.6 异常安全问题

- 异常抛出后，后面的代码就不再执行，前面申请了资源(内存、锁等)，后面进行释放，但是中间可能会抛异常就会导致资源没有释放，这里由于异常就引发了资源泄漏，产生安全性的问题。中间我们需要捕获异常，释放资源后面再重新抛出，当然后面智能指针章节讲的RAII方式解决这种问题是更好的。
- 其次析构函数中，如果抛出异常也要谨慎处理，比如析构函数要释放10个资源，释放到第5个时抛出异常，则也需要捕获处理，否则后面的5个资源就没释放，也资源泄漏了。《Effctive C++》第8个条款也专门讲了这个问题，别让异常逃离析构函数。

```
double Divide(int a, int b)
{
    // 当b == 0时抛出异常
    if (b == 0)
    {
            throw "Division by zero condition!";
    }
    return (double)a / (double)b;
}

void Func()
{
    // 这里可以看到如果发生除0错误抛出异常，另外下面的array没有得到释放。
    // 所以这里捕获异常后并不处理异常，异常还是交给外层处理，这里捕获了再
    // 重新抛出去。
    int* array = new int[10];

    try
    {
            int len, time;
            cin >> len >> time;
            cout << Divide(len, time) << endl;
    }
    catch (...)
    {
            // 捕获异常释放内存
            cout << "delete []" << array << endl;
            delete[] array;
```

```
29
30              throw; // 异常重新抛出，捕获到什么抛出什么
31      }
32
33      cout << "delete []" << array << endl;
34      delete[] array;
35  }
36
37  int main()
38  {
39      try
40      {
41              Func();
42      }
43      catch (const char* errmsg)
44      {
45              cout << errmsg << endl;
46      }
47      catch (const exception& e)
48      {
49              cout << e.what() << endl;
50      }
51      catch (...)
52      {
53              cout << "Unkown Exception" << endl;
54      }
55
56      return 0;
57  }
```


1.7 异常规范

- 对于用户和编译器而言，预先知道某个程序会不会抛出异常大有裨益，知道某个函数是否会抛出异常有助于简化调用函数的代码。
- C++98中函数参数列表的后面接throw()，表示函数不抛异常，函数参数列表的后面接throw(类型1,类型2...)表示可能会抛出多种类型的异常，可能会抛出的类型用逗号分割。
- C++98的方式这种方式过于复杂，实践中并不好用，C++11中进行了简化，函数参数列表后面加noexcept表示不会抛出异常，啥都不加表示可能会抛出异常。
- 编译器并不会在编译时检查noexcept，也就是说如果一个函数用noexcept修饰了，但是同时又包含了throw语句或者调用的函数可能会抛出异常，编译器还是会顺利编译通过的(有些编译器可能会报个警告)。但是一个声明了noexcept的函数抛出了异常，程序会调用 `terminate` 终止程序。
- noexcept(expression)还可以作为一个运算符去检测一个表达式是否会抛出异常，可能会则返回false，不会就返回true。

```cpp
// C++98
// 这里表示这个函数只会抛出bad_alloc的异常
void* operator new (std::size_t size) throw (std::bad_alloc);
// 这里表示这个函数不会抛出异常
void* operator delete (std::size_t size, void* ptr) throw();

// C++11
size_type size() const noexcept;
iterator begin() noexcept;
const_iterator begin() const noexcept;

double Divide(int a, int b) noexcept
{
    // 当b == 0时抛出异常
    if (b == 0)
    {
        throw "Division by zero condition!";
    }
    return (double)a / (double)b;
}

int main()
{
    try
    {
        int len, time;
        cin >> len >> time;
        cout << Divide(len, time) << endl;
    }
    catch (const char* errmsg)
    {
        cout << errmsg << endl;
    }
    catch (...)
    {
        cout << "Unkown Exception" << endl;
    }

    int i = 0;
    cout << noexcept(Divide(1,2)) << endl;
    cout << noexcept(Divide(1,0)) << endl;
    cout << noexcept(++i) << endl;

    return 0;
}
```

2. 标准库的异常

- https://legacy.cplusplus.com/reference/exception/exception/
- C++标准库也定义了一套自己的一套异常继承体系库，基类是exception，所以我们日常写程序，需要在主函数捕获exception即可，要获取异常信息，调用what函数，what是一个虚函数，派生类可以重写。