華中科技大學

第10章 异常与断言

许向阳

xuxy@hust.edu.cn

大纲

華中科技大學

10.1 异常处理

10.2 捕获顺序

10.3 函数的异常接口

10.4 异常类型

10.5 异常对象的析构

10.6 断言

异常（Exception）

- 程序运行时，检测出**将要**发生不正常的情况
- 最常见的异常

  除数为0

  数组越界访问

  打开文件时，文件不存在

  无效数据，如输入的年龄为负数

  分配空间时，空间不够导致无法分配

  网络通讯时，网络不通等

10.1 异常处理

- 异常处理即异常错误处理
  - 给出错误提示
  - 让程序沿一条不会出错的路径继续执行
  - 通知用户遇到了何种异常，不得不停止执行
  - 结束前做一些必要的工作

    将内存中的数据写入文件

    关闭打开的文件

    释放动态分配的内存空间等

10.1 异常处理

异常处理的例子

```
#include <fstream>
#include <iostream>
using namespace std;
int main(int argc, char ** argv)
{  ifstream source(argv[1]);      //打开文件
    char line[128];
    if (source.fail( )) {
      cout <<"error opening the file "<<argv[1] <<endl;
      exit(1);
    }
   while(!source.eof())
    { source.getline(line, sizeof(line));    cout <<line <<endl;  }
   source.close();
   return 0;
}
```

10.1 异常处理

异常处理的例子

```
int main(int argc, char ** argv)
{  ifstream source(argv[1]);      //打开文件
    char line[128];
    try {
        if(source.fail( ))   throw  1;
        while(!source.eof())
          { source.getline(line, sizeof(line));
             cout <<line <<endl;  }
         source.close();
   }
   catch (int) {  cout <<"error opening the file "<<argv[1];   }
   return 0;
}
```

10.1 异常处理

- 谁负责检查是否出现了异常？
- 出现了什么异常？

  异常的信息：编号？ 异常提示串？ 什么位置？……
- 谁负责接收报告？

  自己处理？向上级报告？向上级的上级报告？
- 接到报告后如何处理？
- 在执行一段程序的过程中，可能出现多个异常，

  如何处理？

10.1 异常处理

- C++的异常处理机制的基本思想

  将异常的检测与处理分离。

- 用try、throw和catch三个关键字实现

10.1 异常处理


```
int main( )
{  int m,n;
   cout<<"Please input two integers:"; cin>>m>>n;
   try
   {     if (n==0)   throw 0;
         cout<< (m/n)<<endl;
   }
   catch(int) // 仅指出异常的类型，而未定义接收参数
   {     cout<<"Divided by 0!"<<endl;
         return -1;
   }
  return 0;
}
```

在try代码块中包含需要监控的程序段

抛出一个整型异常

catch语句捕获一个整型异常并处理

10.1 异常处理

```
try{
        ..... throw   1 ;       // 抛出int 型的异常
        ..... throw   "error"; // 抛出字符串型的异常
              throw  MyException("Cannot do it!");
}                              // MyException 是一个 类
catch (int  arg1){
        ......          }
catch (const char * arg2){
      //exception handling for  const char * type  }
catch (MyException arg3){
     //exception handling for type MyException  }
catch (...){  ..... }
```

变量arg1用来接收throw抛出的异常值

10.1 异常处理

- 要监控的程序部分包含在**try**代码块中；
- 在**try**块中调用的函数也将被监控；
- 如果**try**块中的程序代码发生了异常错误，就使用**throw**抛出异常；
- **try**块中抛出的异常将被紧跟在**try**语句之后的**catch**语句捕获。
- **try** 与 **catch** 是紧密关联的，不能单独出现**try**，也不能单独出现 **catch**

10.2 捕获顺序

- 在try语句后面可以有一个或多个catch语句；
- 如果 catch语句中指定的数据类型与异常的类型匹配，那么这个catch语句将被执行。

  所有其他的catch语句都将被忽略。
- 当异常信息被捕获时，变量arg将用来接收异常信息的值；
- 如果抛出的异常没有与之类型相匹配的catch语句，异常向上一级传递。
- 如果没有出现异常，则不会执行catch中的语句

10.2 捕获顺序


```
int main( )
{  int m,n;
    cout<<"Please input two integers:"; cin>>m>>n;
    try
    {  cout<<division(m,n)<<endl;   }
    catch(int)
    {      cout<<"Divided by 0!"<<endl;
           return -1;
    }
   return 0;
}
int division(int x,int y)
{   if (y==0)      throw 0;
    return x/y;
}
```

函数中未处理异常；

向上一级传递异常

10.2 捕获顺序

```
int main( )
{  int m,n;
   cout<<"Please input two integers:"; cin>>m>>n;
   try
   {    if (n==0)   throw 0;
        cout<< (m/n)<<endl;
   }
   catch(const char * arg)
   {    cout<<arg<<endl;
        return -1;
   }
  return 0;
}
```

抛出的异常的值与变量arg类型不配;
异常应向上一级传递，但main已是最后一级，程序终止

10.2 捕获顺序

```
void Xhandler(int test)
{
  try{    if(test) throw test;
          else throw "Value is zero";
  }
  catch(int i) {
    cout << "Caught One!  Ex. #: " << i << '\n';
  }
  catch(const char *str) {
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}
```

每个catch语句所能捕获的异常必须是不同类型

抛出常量串异常，应用 const char * 来捕获

10.2 捕获顺序


```
int main( )
{  int m,n;
    cout<<"Please input two integers:";
    cin>>m>>n;
    try
    {     if (n==0)   throw 0;
          cout<< (m/n)<<endl;
    }
    catch(const  char * arg)   // 抛出的异常信息的值
                               // 与形参变量arg类型不配
    {     cout<<arg<<endl;
          return -1;
    }
     cout<<"here"<<endl;     //  无int 类型的异常处理，
     return 0;               //  引起非正常的程序终止，
                             //  不会显示 here
}
```

10.2 捕获顺序

非正常的程序终止

- 如果抛出的异常没有与之类型相匹配的catch语句，则该异常信息将被传递到调用该程序模块的上一级，它的上级捕获到这个异常信息后进行处理。
- 如果上一级模块仍然未处理，就再传递给其上一级，逐级上传。
- 如果到最高一级还无法处理，那么将发生非正常的程序终止（abnormal program termination）。

自定义运行终止函数

- 如果程序中抛出了一个未被处理的异常信息，
  系统将调用C++标准库中的函数terminate( )，
  默认情况下，terminate( )用abort( ) 终止程序。

- 程序员可以编写自己的终止函数，
  通过set_terminate函数传递给异常处理模块，
  系统在找不到相匹配的异常错误处理模块时调用该函数。

10.2 捕获顺序

```
void myterminate() //自定义的运行终止函数
{
    cout<<"This is my terminater."<<endl;
    //...释放程序中申请的系统资源
    exit(1);
}
int main()
{
    try{
        set_terminate(myterminate);
        //...
        throw "Exception ... ";
    }
    catch(int i){  }
    return 0;
}
```

自定义的运行终止函数

10.2 捕获顺序

捕获所有的异常

```
void Xhandler(int test)
{
  try{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a';   // throw char
    if(test==2) throw 123.23; // throw double
  }
  catch(...) { // 捕获所有的异常
    cout << "Caught One!\n";  }
}
```

10.2 捕获顺序

捕获所有其他的异常

```
void Xhandler(int test)
{  try{
     if(test==0) throw test;   // throw int
     if(test==1) throw 'a';    // throw char
     if(test==2) throw 123.23; // throw double
   }
   catch(int i) { // catch an int exception
         cout << "Caught " << i << '\n';  }
   catch(...) { // 捕获所有其他的异常
         cout << "Caught One!\n";  }
}
```

10.2 捕获顺序

- 没有操作数的 throw 语句，

  只能出现在 catch 语句中；

  向上一级传递这一异常；

- 在catch 中出现含有操作数的throw语句，

  向上一级抛出新的异常；

- 在try 语句中，不能出现无操作数的throw。

10.3 函数的异常接口

- 可由该函数引发的、而其自身又不想捕获或处理的异常；
- 在函数的参数表后面定义异常接口，用throw列出要引发的异常类型；
- … func(…)；// 可引发任何异常
- … func(…) noexcep ； //不引发任何异常
  … func(…) throw( ); //不引发任何异常
  … func(…) throw(void); //不引发任何异常
- … func(…) throw(A, B, C);
  引发 A 类型、B类型、C类型的异常
- … func(…) const throw(A, B, C); (成员函数)

10.3 函数的异常接口


```
int sum(int a[ ], int t, int s, int c) throw (const char *)
{
    if (s < 0 || s >= t || s + c<0 || s + c>t)
          throw "subscription overflow";
        // 若发出const char *类型的异常，
        // 下面的语句不执行
    int r = 0,x=0;
    for (x = 0; x < c; x++)
        r += a[s+x];
    return r;
}
```

10.3 函数的异常接口

- 异常接口不是函数原型的一部分，不能通过异常接口来定义和区分重载函数；
- **不可意料的异常**：若函数申明不引发任何异常的函数，但函数体又引发的异常，或者引发了未说明的异常；
- 通过**set_unexpected**过程，可以将不可意料的异常处理过程设置为程序自定义的不可意料的异常处理过程。

10.3 函数的异常接口


```
void f1()   // noexcept
{     throw "occur error"; }
int main() {
   try {      f1( );   }
    catch (...) {
        cout << "catch any exception" << endl;
    }
    return 0;
}
```

对于 void f1( ) {…} ， 显示 catch any exception

对于 void f1( ) noexcept {…} ,

编译警告：不引发异常，但又包含了异常。

运行结果：不会显示 catch any exception , 异常终止

10.3 函数的异常接口

- noexcept可以表示throw( )或throw(void)；
- noexcept一般用在不会出现异常的函数后面；
- noexcept可以出现在任何函数的后面，包括constexpr函数和Lambda表达式的参数表后面；
- throw(除void外的类型参数)不应出现在constexpr函数的参数表后面，并且constexpr函数也不能抛出异常，否则不能优化生成常量表达式。

10.4 异常类型

- 程序员可自己创建异常类型；
- 在实际程序中，大多数异常的类型都是类，而不是内置数据类型（标准类型）；
- 创建一个类来描述发生的错误信息，可以帮助异常处理模块处理错误；
- catch可以捕获任意类型的异常，

10.4 异常类型

使用异常类

```
#include <iostream>
using namespace std;
class MyException {
public:
    char str_what[80];
    MyException() { *str_what = 0; }
    MyException(const char *s) {
        strcpy(str_what, s);
    }
};
```

10.4 异常类型

使用异常类

```
int main()
{    int a, b;
     try {
       cout << "Enter numerator and denominator: ";
        cin >> a >> b;
        if(!b)  throw MyException("Cannot divide by 0!");
        else   cout << "Quotient is " << a/b << "\n";
     }
    catch (MyException e) { // catch an error
        cout << e.str_what << "\n";
    }
    return 0;
}
```

10.4 异常类型

使用异常类

```
try {
        throw  MyException("Cannot divide by 0!");
   //    MyException    temp("Cannot divide by 0!");
   //     throw  temp;
}
catch (MyException &e) {  }
//  catch (MyException e) {  }
```

不同的写法，执行过程有差异；
细节等同于函数的参数的传递

10.4 异常类型

- 如果父类A的子类为B，B类异常能被catch(A)、catch(const A)、 catch(volatile A)、 catch(const volatile A) 等捕获。
- 指向可写B类对象的指针异常也能被catch(A*)、catch(const A*)、 catch(volatile A*)、catch(const volatile A*)等捕获。
- 捕获子类对象的catch应放在捕获父类对象的catch前面。
- catch(const volatile void *)能捕获任意指针类型的异常
- catch(...)能捕获任意类型的异常。

10.4 异常类型

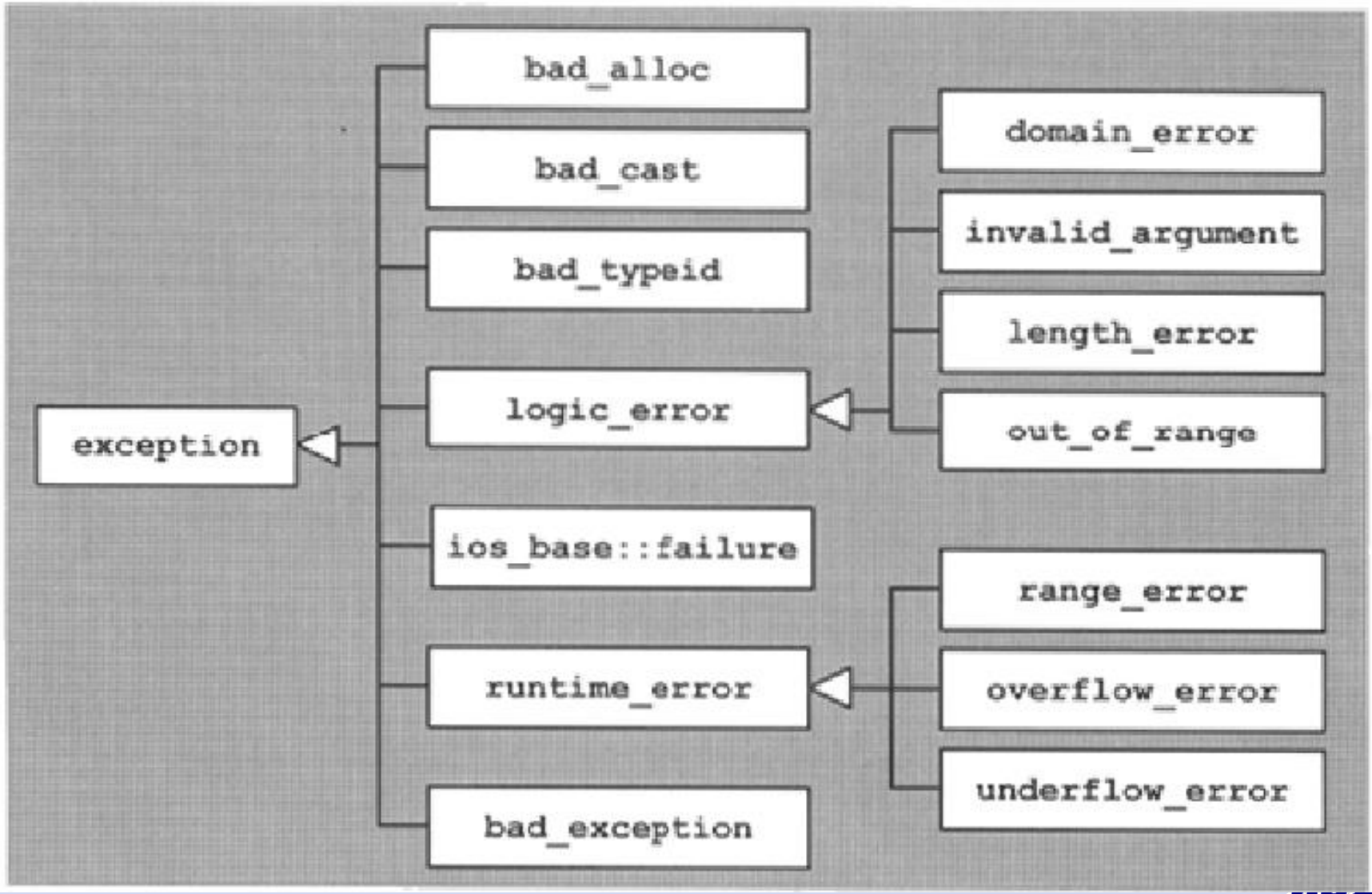
- 如果通过new产生的指针类型的异常，在catch处理后，通常应使用delete释放内存；
- 如果继续传播指针类型的异常，则可以不使用delete；
- 从最内层被调函数抛出异常，
  到外层调用函数的catch 捕获异常，
  函数调用链所有局部对象都会被自动析构，
  使用异常处理机制在一定程度上防止了内存泄漏；
- 调用链中通过new分配的内存不会自动释放；
- 特殊情况在产生异常对象的过程中也会出现内存泄漏情况：未完成构造的对象。

10.4 异常类型

```
class A
{
   char s[20];
public:
   A(const char* t) {
       strcpy(s, t);
      cout << "construct A " << endl; }
   ~A() { cout << s<<" de construct A ..."
                  << endl; }
};
void f1()
{
   A  a("f1");
   throw "exception";
   cout << "this will not occur" << endl;
}
```

```
void f2()
{   A  a("f2");
   f1();
   cout << "f2 over" << endl;
}

int main()
{   try { f2(); }
   catch (...) {
      cout << "catch processing" << endl;}
   return 0;
}
```

10.4 异常类型


exception
bad_alloc
bad_cast
bad_typeid
logic_error
ios_base::failure
runtime_error
bad_exception
domain_error
invalid_argument
length_error
out_of_range
range_error
overflow_error
underflow_error

10.4 异常类型

标准异常的名字	抛出异常的主体	对应的头文件
bad_alloc	new	< new >
bad_cast	dynamic_cast	<typeinfo>
bad_typeid	typeid	<typeinfo>
bad_exception	exception specification	<exception>

10.4 异常类型

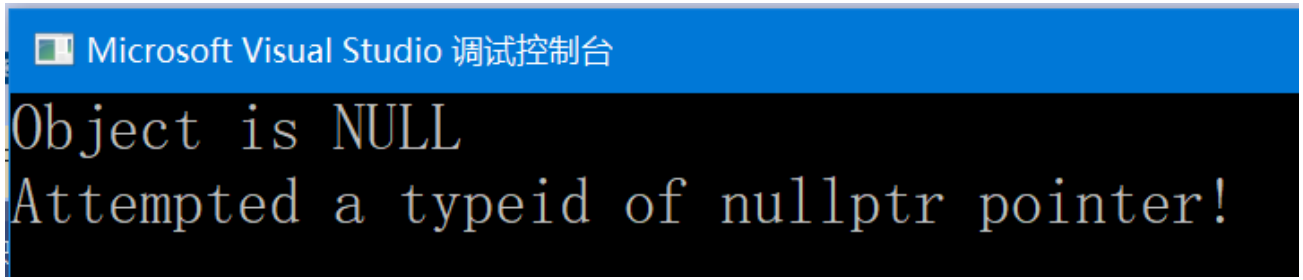

```
class bad_typeid : public exception
{
public:
   bad_typeid();
   bad_typeid(const char * _Message = "bad typeid");
   bad_typeid(const bad_typeid &);
   virtual ~bad_typeid();

   bad_typeid& operator=(const bad_typeid&);
   const char* what() const;
};

// vcruntime_typeinfo.h
```

10.4 异常类型


```
#include <typeinfo>
#include <iostream>
using namespace std;
class A {
public:    virtual ~A();
};

int main() {
    A* a = NULL;

    try {   // 在有虚函数时，typeid 抛出类型为 bad_typeid异常
        cout << typeid(*a).name() << endl;  // Error condition
    }
    catch (bad_typeid) {   // 仅说明要捕获的异常类型
        cout << "Object is NULL" << endl;
    }
}
```

10.4 异常类型


```
A* a = NULL;

try {   // 在有虚函数时，typeid 抛出类型为 bad_typeid异常
        cout << typeid(*a).name() << endl;  // Error condition
}
catch (bad_typeid &bt) {
        cout << "Object is NULL" << endl;
        cout << bt.what() << endl;
}
```



10.4 异常类型


```
#include <iostream>
#include <new>     //需要包含该头文件
using namespace std;
...
   try {
     p = new int[32]; // 为整型数组申请动态存储单元
   }
   catch (bad_alloc xa) {
     cout << "Allocation failure.分配内存失败。\n";
     return 1;
   }
...
```

10.4 异常类型

- 将程序中正常处理的代码（描述问题的算法）与异常处理代码分离开来，提高了程序的可读性。

- 提供了一种更规则的处理异常的风格，便于软件项目组人员之间的合作。
  - 通常情况下，类的创建者监控代码段，从类中抛出异常。
  - 类的使用者捕获到异常并处理。

10.4 异常类型

- 在异常发生时，能够撤销对象，并自动调用析构函数进行善后处理，释放对象所占用的系统资源。

发生异常时资源释放

```
class Y {
        int* p;
        void init() ;
public:
        Y(int s) { p = new int[s] ; init( ) ; }
        ~Y( ) { delete[] p; }
        / / ...
};
```

A safe variant

```
class Z {
        vector<int> p;
        void init() ;
public:
        Z(int s) : p(s) { init( ) ; }
        / / ...
};
```

10.6 断言

- 断言（assert）是一个带有整型参数的用于调试程序的函数，在运行时检查断言；
- 实参为真：程序继续执行；
- 实参为假：

  输出断言表达式、所在程序的文件名、行号，

  调用abort()终止程序的执行。
- static_assert定义的断言在编译时检查，为假时终止编译运行。

10.6 断言


```
#include <assert.h>
class  SET {
    int* elem, used, card;
public:      SET(int card);
    virtual int has(int)const;
    virtual SET& push (int);                    //插入一个元素
    virtual ~SET( ) noexcept { if (elem) { delete elem;  elem = 0; } };
};
SET::SET(int c) {
    card = c;
    elem = new int[c];
    assert(elem);            //当elem非空时继续执行
    used = 0;
}
```

10.6 断言


```
int SET::has(int v) const {
    for (int k = 0; k < used; k++)
        if (elem[k] == v) return 1;
    return 0;
}
SET& SET::push(int v) {
    assert(!has(v));          //当集合中无元素v时继续执行
    assert(used < card);      //当集合能增加元素时继续执行
    elem[used++] = v;
    return *this;
}
```

10.6 断言


```
void  main(void)
{
    static_assert(sizeof(int)==4);   // 静态断言
      //VS2019采用x86编译模式时为真，不终止编译运行
    SET  s(2);                       //定义集合只能存放两个元素
    s.push(1).push(2);               //存放第1，2个元素
    s.push(3);
          //因不能存放元素3，断言为假，程序被终止
}
```

总结

- 什么是异常？怎样抛出异常？怎样捕获异常？
- 什么是断言？编译时断言和运行时断言有何差别？
- try …… catch 关联在一起使用
- throw 可间接在一个 try 块中，即 含有throw语句的函数，或者其更上一级的调用函数在 try 块中。
- 一个异常若没有被任何 catch 所捕获，则引起程序异常终止；
- 多个catch 的运行规则，按顺序检查是否匹配、匹配一个后，后面的catch 不再检查。

实战演练

- 在编写一个网页时，需要用户填写多个信息。
- 对有些项是必填的有要求，对有些项信息的正确性也有要求。
- 最后有提交按钮。
- 请问是在点击提交按钮时，检测正确性合适，还是在每一项输入后，检查正确性合适？
- 如何使用C++的异常处理机制，按用户再次输入信息？

实时验证（每项输入后检查）

•优点：

- 即时反馈，用户体验好
- 用户能立即知道问题所在
- 避免在提交时发现多个错误时的挫败感

•缺点：

- 实现复杂度较高
- 可能干扰用户输入流程
- 需要更频繁的验证调用

提交时验证（点击提交按钮时检查）

•优点：

- 实现简单直接
- 不干扰用户输入过程
- 可以一次性显示所有错误

•缺点：

- 用户体验较差（需要填完所有内容才知道错误）
- 可能让用户重复修改同一问题

实战演练

最佳实践：混合策略

实时验证：用于格式明显的字段

（邮箱、电话、密码强度）

提交时验证：用于业务逻辑相关的字段

（用户名唯一性、数据一致性）

实战演练

- 使用分层异常处理

  ValidationException：字段级错误

  FormValidationException：表单级错误

  BusinessRuleException：业务逻辑错误

- 提供友好的重新输入机制

  实时验证：立即提示，原地重新输入

  提交验证：显示所有错误，允许选择性修改

- 考虑使用验证框架

  对于复杂项目，可以考虑使用现有的验证库，

  或者自己封装一个灵活的验证器系统

这样设计既保证代码的健壮性，又提供良好的用户体验。

实战演练

華中科技大學

```
// 自定义验证异常类
class ValidationException : public std::exception {
private:
    std::string field_name;
    std::string error_message;

public:
    ValidationException(const std::string& field, const std::string& msg)
        : field_name(field), error_message(msg) { }

    const char* what() const noexcept override {
        return error_message.c_str();
    }

    const std::string& getFieldName() const { return field_name; }
    const std::string& getErrorMessage() const { return error_message; }
};
```

实战演练


```
// 表单数据类
struct UserForm {
    std::string username;
    std::string email;
    std::string phone;
    int age;

    void display() const {
        std::cout << "=== 表单数据 ===" << std::endl;
        std::cout << "用户名: " << username << std::endl;
        std::cout << "邮箱: " << email << std::endl;
        std::cout << "电话: " << phone << std::endl;
        std::cout << "年龄: " << age << std::endl;
    }
};
```

实战演练


```
// 验证工具类
class FormValidator {
public:
    // 用户名验证：3-20个字符，只能包含字母数字
    static void validateUsername(const std::string& username) {
        if (username.length() < 3 || username.length() > 20) {
            throw ValidationException("用户名", "长度在3-20个字符之间");
        }
        std::regex pattern("^[a-zA-Z0-9]+$");
        if (!std::regex_match(username, pattern)) {
            throw ValidationException("用户名", "只能包含字母和数字");
        }
    }

    static void validateEmail(const std::string& email) {.....}
    ......
};
```

实战演练


```
// 验证工具类
class FormValidator {
 // 提交时整体验证
        static void validateForm(const UserForm& form) {
            std::vector<std::string> errors;
            try {  validateUsername(form.username);  }
            catch (const ValidationException& e)
           { errors.push_back(e.getErrorMessage()); }

            try { validateEmail(form.email); }
            catch (const ValidationException& e)
           {  errors.push_back(e.getErrorMessage());  }
           ……
        }
};
```

实战演练


```
class InputHandler {    // 输入处理类 - 支持重新输入
public:
    static std::string getValidatedString(const std::string& prompt,
                const std::function<void(const std::string&)>& validator) {
        std::string input;
        while (true) {
                std::cout << prompt;          std::getline(std::cin, input);
                try {
                    validator(input);
                    return  input;  // 验证成功，返回输入
                }
                catch (const ValidationException& e) {
                    std::cout << "❌ 错误: " << e.what() << std::endl;
                    std::cout << "请重新输入..." << std::endl;
                }
        }
    }
};
```

实战演练


```
// 模拟实时验证（每项输入后立即验证）
UserForm collectFormWithRealTimeValidation() {
    UserForm form;

    std::cout << "=== 用户注册（实时验证）===" << std::endl;

    // 用户名 - 实时验证
    form.username = InputHandler::getValidatedString(
        "请输入用户名: ",     FormValidator::validateUsername  );

    // 邮箱 - 实时验证
    form.email = InputHandler::getValidatedString(
         "请输入邮箱: ",     FormValidator::validateEmail  );
}
```

实战演练----总结

- ValidationException：继承自std::exception，
  用于表示验证错误，包含字段名和错误信息。
- UserForm：数据类，存储用户填写的信息。
- FormValidator：提供静态方法验证各个字段和整个表单。
- InputHandler：处理用户输入，支持重新输入直到有效。

实战演练----总结

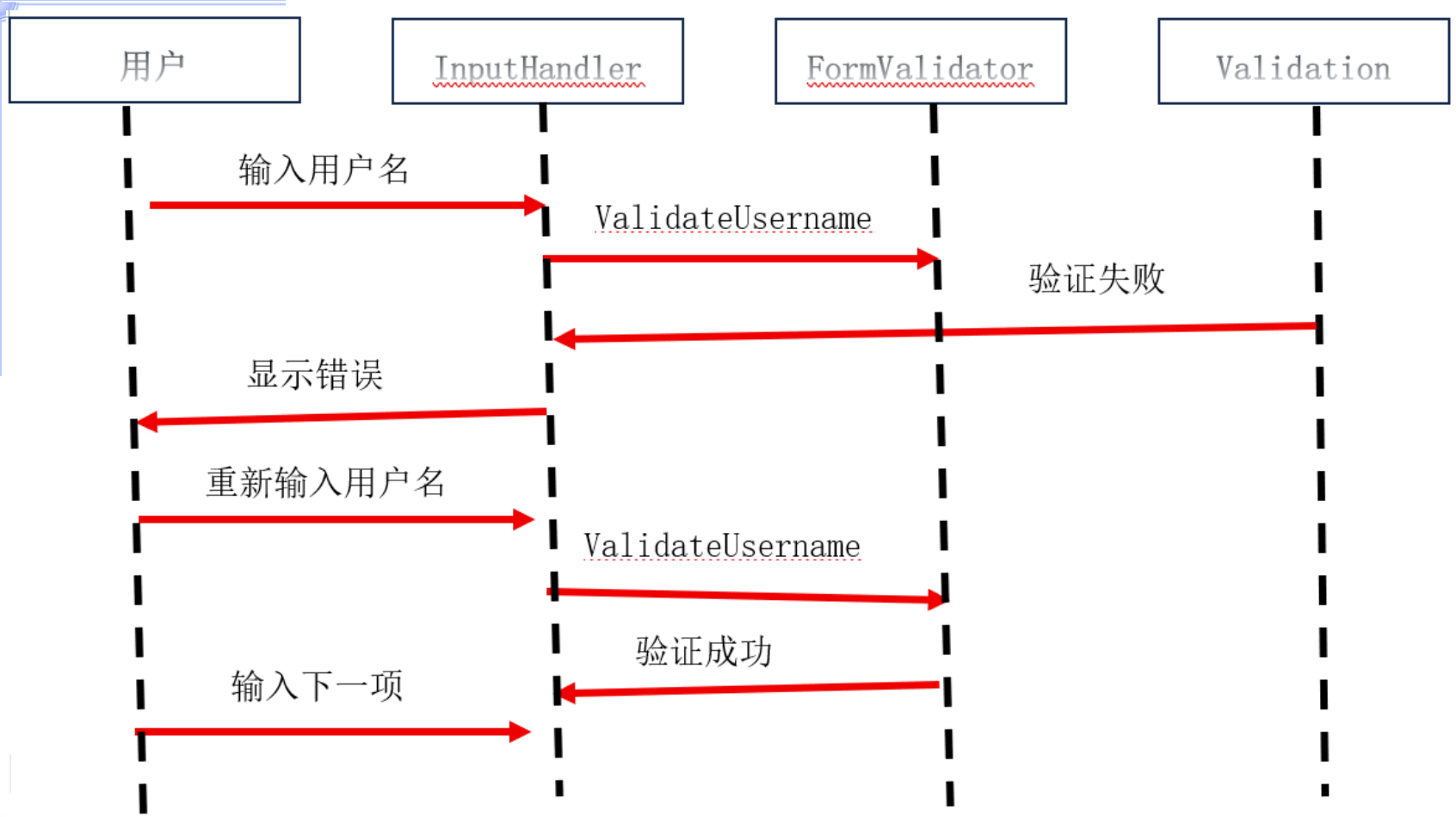
- ValidationException 被 FormValidator 和 InputHandler 使用（抛出和捕获）。
- FormValidator 提供的方法被 InputHandler 使用。
- InputHandler 使用 std::function 来接收验证器，这些验证器通常由 FormValidator 提供。
- UserForm 是数据容器，在 InputHandler 中创建并填充，然后被 FormValidator 验证。

实战演练----总结

工作流程：

1. 程序启动，收集用户输入。
2. 对于每个字段，InputHandler 使用相应的验证函数（来自 FormValidator 或自定义 Lambda）来验证输入。
3. 如果验证失败，抛出 ValidationException，InputHandler 捕获并显示错误，然后重新输入。
4. 所有字段都输入成功后，可以选择在提交时再次整体验证（确保数据一致性，但基础验证已经通过）。
5. 最终，得到一个有效的 UserForm 对象。

实战演练----总结


用户
InputHandler
FormValidator
Validation
输入用户名
ValidateUsername
验证失败
显示错误
重新输入用户名
ValidateUsername
验证成功
输入下一项

实战演练----总结

UML 类图

类、继承关系、依赖关系、成员属性和方法

练习

- 多个catch 的运行规则，按顺序检查是否匹配、匹配一个后，后面的catch 不再检查。

如果有 catch(…)、catch(const void *)、catch(int *) 3个异常处理程序，应该如何摆放它们的位置？

```
catch(int *)  { …..}
catch(const void *) { …….}
catch(…)  { ……}
```

设有异常处理顺序：

```
catch(void * ) { ……}
catch(const  int  *)  {……}
```

什么样的异常会被下面一个catch 语句所捕获？