异常

回忆C++的异常

在C++中，我们会更多关注异常的嵌套问题（~~我因为这个还被扣分子~~）。由于C++的异常实际行为和编译器等因素也有关，所以下面只简单回忆下C++里异常的大概样子：

在C++中可以`throw`出一个变量我们计作`valTh`，异常机制会使用`valTh`复制构造出一个`valEx`，然后退出`throw`的那段语句（伴随着析构的过程，这里不讨论析构顺序了，一来析构顺序是依赖编译器的实现的，二来在Java里就完全不用担心析构顺序的问题了——毕竟析构都没了），进入外层的`catch`语句。在`catch`中会选择第一个能捕捉到`valEx`的量`valCa`（这里注意，如果`valEx`是一个继承类，其会其被基类捕捉到），然后捕捉构造`valCa`，并在`catch`部分继续执行程序。如果一个`catch`并没有捕捉到就会打断语句并继续进入外层的`catch`，直到错误被抛到了`main`外

C++中任何东西都可以作为被抛出的对象，这当然也包括基础类型

如果没有办法确定有什么异常会被`catch`，C++允许使用`catch(...)`来捕捉所有类型的异常

Java中的异常简单介绍

这里的内容比较乱，是建立在读者已经了解C++异常机制的基础上的，所以很多东西会很有跳跃性

和C++不同的地方

Java中的异常与C++最大的不同有：

所有的自己编写的异常类应当继承自某个异常类

特别的，最好继承自类型相近的异常类（虽然这种异常类可能不是很好找）

此外，由于所有的异常类实际上都继承自`Throwable`（是的，是继承类不是实现接口，尽管`Throwable`是一个形容词），所以`Throwable`里面会有一些默认的方法，这些方法被继承之后都是可以使用的，我们会在后面的代码过程中看到部分`Throwable`的方法

实际上`Throwable`有两个继承类，分别为`Exception`和`Error`，后者一般在发生系统级错误时被抛出，大多数情况下码农对其是无能为力的。关于`Throwable`的继承树见后

这种单一继承源的编程思路带来了太多好处，比如通用的方法、相对简单的类型检查等等

由于Java`Class`类的反射机制存在，其可以方便地调用类名，所以对于异常来说，最重要的部分就是类名，我们也会在后面看到这一点

异常抛出的严格检查

编译器会检查`try`里面有哪些可能被抛出的异常，并且要求可能被抛出的异常要么<1> 被`catch`、<2> 在方法中被`throws`声明；

在方法声明的时候必须把所有可能抛出的异常使用`throws`声明到（但是如果某种异常不会被抛出也可以被声明）。在C++中其实也存在这么一种声明机制，只是C++中这不是必须的，但是在Java中，编译器如果检查到有某种可能在方法内被抛出的异常没有被声明到的话会报错

但是有一点遗憾，即使是这样也没有办法保证所有异常都被检查到，我们可以用下面的语句卡出bug（来自《Java编程思想》P268）

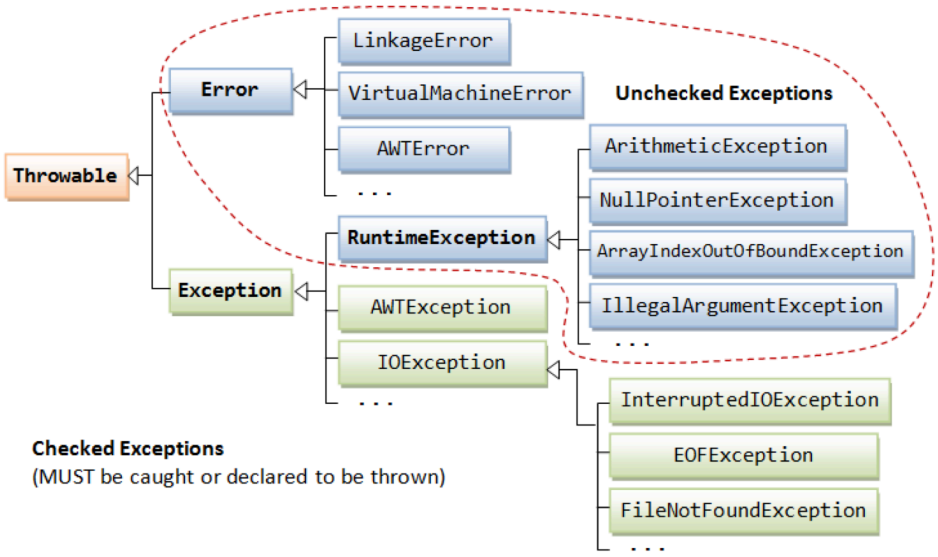
```
class VeryImportantException extends Exception{
    @Override
    public String toString(){
        return "A Very Important Exception";
    }
}

class HoHunException extends Exception{
    public String toString(){
        return "A trivial Exception";
    }
}

public class LostMessage {
    void f() throws VeryImportantException{
        throw new VeryImportantException();
    }
    void dispose() throws HoHunException{
        throw new HoHunException();
    }

    public static void main(String[] args) {
        try{
            LostMessage lm=new LostMessage();
            try {
                lm.f(); // 抛出了异常VeryImportantException，但是会丢失
            }finally {
                lm.dispose(); // 抛出异常HoHunException并且覆盖了VeryImportantException
            }
        }catch (Exception e){ // 捕捉的是VeryImportantException
            System.out.println(e); // "A trivial Exceptions"
            // 这里和后面介绍的try-with-resources不一样，不能通过e.getSuppressed()获得被忽略的
信息
        }
    }
}
```

对于所有的异常，会被分成`unchecked exceptions`和`checked exceptions`，在继承中的关系可以看下图



`checked exceptions`是会被编译器检查的，你必须捕捉(即`catch`到)或声明(即在方法后`throws`到)

而`unchecked exceptions`不会被编译器检查，因此可以不专门捕捉/声明

捕捉所有异常和`finally`关键字

在C++中如果想要捕捉所有异常需要使用`catch(...)`语句，而在Java中就是使用`catch(Exception )`，因为所有的需要处理的异常都继承自`Exception`（见上图），这也是单一继承源的好处。另外在Java中可以使用`finally`关键字，无论是否有异常无论异常是否被`catch`，`finally`里面的内容都会被执行

为了防止程序流程出问题，不要在`finally`里面加上诸如`return`、`break`的语句（见《Java核心技术卷I》P292）

按照《Java编程思想》，`finally`实际上一般用于内存回收、清理已经打开的文件等等

在方法继承中的特殊情况

如果派生类的某一个方法继承自基类的某个方法，由于继承类方法和基类方法的"相同形式"指<相同函数名、相同形参列表、相同的`throws`列表>三者相同，也就是继承类的对应方法不能抛出比基类多或者少的异常

和C++相似的地方

一旦抛出异常，就会终止当前语句的执行，然后在运行栈中上跳寻找第一个能`catch`对应异常的位置。我们下面通过一个例子来回忆异常是什么样的

为了方便叙述，我们先写一个很简单的异常类

```
public class MyException extends Exception {
    public MyException() {
        super();
        System.out.println("MyException : MyException [] [] -> []");
    }
}
```

按照《Java编程思想》，这种异常类大多数情况都够用了。然后写个测试函数

```
import java.util.*;
import java.io.*;
import java.math.*;
import java.time.*;
import java.lang.*;

public class Main {
    public static void g() throws MyException{
        System.out.println("Main : g [] [] -> [void]");
        MyException valG = new MyException();
        throw valG;
    }
    public static void f() throws MyException{
        System.out.println("Main : f [] [] -> [void]");
        try{
            g();
        }catch(MyException valF){
            System.out.println("Cautch in f()");
            throw valF;
        }
    }
    public static void main(String[] args) throws IOException,MyException {
        Scanner cin = new Scanner(System.in);
        try{
            f();
        }catch (MyException valMa){
            System.out.println("Cautch in main");
        }

    }
}
```

可以看一下上面的过程思考可能发生什么过程

在Java中，捕捉的时候是直接相当于做了一次`=`（也就是异常类型指针层面的等于），而不是调用`.clone()`，这和C++中会调用复制构造函数是完全不一样的（实际上也正是这种不一样让Java的异常看起来没那么复杂——至少你不用去研究构造析构顺序这种东西了）。因此上面的代码输出是

```
Main : f [] [] -> [void]
Main : g [] [] -> [void]
MyException : MyException [] [] -> []
Cautch in f()
Cautch in main
```

也就是，`try`里面可能会`throw`一些东西并阻断代码继续运行，然后被抛出的异常类会在上层的`catch`的某一处捕捉

`Exception`和`Throwable`的介绍

构造器

`Excrption`的构造器有四个比较重要的构造器，见下

```
package java.lang;
public class Exception extends Throwable { // 实际上Exception的类就只有下面这五个构造函器，没有其他方法属性之类的
    static final long serialVersionUID = -3387516993124229948L; // 不太懂这是干嘛的
    public Exception() { // 默认构造器
        super();
    }
    public Exception(String message) { // 使用message:String的构造器
        super(message);
    }
    public Exception(String message, Throwable cause) { // 使用message:String和cause:Throwable的构造器
        super(message, cause);
    }
    public Exception(Throwable cause) { // 使用cause:Throwable的构造器
        // 对应的方法：Throwable的getCause方法
        super(cause);
    }
    protected Exception(String message, Throwable cause,
                        boolean enableSuppression,
                        boolean writableStackTrace) { // 暂时不介绍
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

`cause`是用来构造链式异常的。如果栈底方法抛出了方法被栈中捕捉了，然后产生了新的异常，希望把两个(甚至过多)异常都传上去，就可以使用这种构造方法，把异常链传递上去

`message`的作用会在后面见到

`Throwable`的构造器：

```
public class Throwable implements Serializable {
    private String detailMessage;
    public Throwable() {
        fillInStackTrace();
    }
    public Throwable(String message) {
        fillInStackTrace();
        detailMessage = message;
    }
    public Throwable(String message, Throwable cause) {
        fillInStackTrace();
        detailMessage = message;
        this.cause = cause;
    }
    public Throwable(Throwable cause) { // 实际上public Exception(Throwable cause)会调用到
这里来
        fillInStackTrace();
        detailMessage = (cause==null ? null : cause.toString());
        this.cause = cause;
    }
    protected Throwable(String message, Throwable cause,
                        boolean enableSuppression,
                        boolean writableStackTrace) { // 这个就不管了
        ...
    }
}
```

`Exception`的`.printStackTrace()`

我们先试下`.getStackTrace()`的方法会产生什么效果。使用的还是之前的那个例子：

```
import java.io.IOException;
import java.util.Scanner;

public class MyException extends Exception {
    public MyException(String message) {
        super(message);
        System.out.println("MyException : MyException [] [] -> []");
    }
    public static void g() throws MyException{
        System.out.println("Main : g [] [] -> [void]");
        MyException valG = new MyException("This is Message");
        throw valG;
    }
```

```
    public static void f() throws MyException{
        System.out.println("Main : f [] [] -> [void]");
        try{
            g();
        }catch(MyException valF){
            System.out.println("Cautch in f()");
            throw valF;
        }
    }
    public static void main(String[] args) throws IOException,MyException {
        Scanner cin = new Scanner(System.in);
        try{
            f();
        }catch (MyException valMa){
            valMa.printStackTrace(System.err);
        }
    }
}
```

得到的输出是：

```
Main : f [] [] -> [void]
Main : g [] [] -> [void]
MyException : MyException [] [] -> []
Cautch in f()
MyException: This is Message
    at MyException.g(MyException.java:11)
    at MyException.f(MyException.java:17)
    at MyException.main(MyException.java:26)
```

最后四行是`err`流的内容，也就是`printStackTrace()`输出的内容。可以看出，`printStackTrace()`会输出类名、构造时的`message`和栈轨迹

关于`printStackTrace()`的具体实现，我们可以使用IDEA来查看，这里就不展示了

`Throwable`的方法

```
public class Throwable implements Serializable {
    public String getMessage() {
        return detailMessage;
    }
    public String getLocalizedMessage() {
        return getMessage();
    }
    public synchronized Throwable getCause() {
        return (cause==this ? null : cause);
    }
```

```
    public synchronized Throwable initCause(Throwable cause) {
        if (this.cause != this)
            throw new IllegalStateException("Can't overwrite cause with " +
                                            Objects.toString(cause, "a null"), this);
        if (cause == this)
            throw new IllegalArgumentException("Self-causation not permitted", this);
        this.cause = cause;
        return this;
    }
    public String toString() {
        String s = getClass().getName();
        String message = getLocalizedMessage();
        return (message != null) ? (s + ": " + message) : s;
    }

    public void printStackTrace() {printStackTrace(System.err);}
    public void printStackTrace(PrintStream s) {printStackTrace(new
WrappedPrintStream(s));}
    private void printStackTrace(PrintStreamOrWriter s) {...}
    public void printStackTrace(PrintWriter s) {printStackTrace(new
WrappedPrintWriter(s));}
    public StackTraceElement[] getStackTrace() {return getOurStackTrace().clone();}
}
```

主要是信息方法，直接看源代码就可以看懂，这里不介绍了

自己的`Exception`

可以选择某一个相似的异常类继承，当然也可以自己写或者继承自`Exception`，只要其继承链上有`Throwable`就行

`getSuppressed()`方法

直接看英文doc的注释：Returns an array containing all of the exceptions that were suppressed, typically by the try-with-resources statement, in order to deliver this exception. If no exceptions were suppressed or suppression is disabled, an empty array is returned. This method is thread-safe. Writes to the returned array do not affect future calls to this method.

`try-with-resources`

这是JDK7之后的写法：

```
try(OutputStream os=new ObjectOutputStream(new BufferedOutputStream(new
FileOutputStream("Bird.txt")))){
    Main bird=new Main();
    ((ObjectOutputStream)os).writeObject(bird);
    throw new FileNotFoundException();
}catch (FileNotFoundException e){
    System.out.println(e.getClass().getName());
    System.out.println(Arrays.toString(e.getSuppressed())); // 返回被覆盖的close产生的异常
的信息
}
```

可以在`try`里面放一系列`Closable`的内容，在`try`块执行将要完成 或者`try`块内将要抛出某异常前 会自动调用这些`Closable`的`.close()`方法

`try`块执行将要完成时调用`close()`这个说得通，那"`try`块内将要抛出某异常前"是什么情况呢？这是发生在`try`内部产生了异常，并且`.close()`也产生了异常的情况，此时`catch`只能检测到`try`块内抛出的异常，并不能查出`.close()`发生的异常，此时需要`.getSuppressed()`获得被覆盖的`close()`产生的异常（见上面的代码）

Java内置异常类

经常使用的`extends Exception`

~~可能也不那么经常~~

以下`extends Exception`

- `IOException`：

  继承类：

  - `FileNotFoundException`，即文件找不到的异常
  - `EOFException`，即碰到文件末尾的异常
  - `ObjectStreamException`，`Object`流异常：
    - `NotSerializableException`：输出一个
- `RuntimeException`：

  继承类：

  - `ClassCastException`，类型转换错误，多发生在类型转换错误的情况下。如果`t instanceof T`是`false`，那么`(T)t`就会导致这种`Excption`被跑出来
  - `IndexOutOfBoundsException`，数组越界错误，`int []a=new int[3];a[3]=2;`会导致这种异常
  - `NullPointerException`，使用一个`null`的`.`运算符会导致这种问题，比如`Scanner cin;...;cin.next();`会抛出这种异常，因为`cin`是`null`但是调用了它的方法
  - `ArithmeticException`，除零之类的问题发生时被抛出
  - `IllegalArgumentException`，传参问题发生时被抛出，它也有一些继承类

        - `NumberFormatException`，比如使用`Integer.parseInt(String str)`的时候发现`str`传入的不能构成一个`int`，就会抛出这个异常
        - `IllegalFormatException`，没见过

      主要是了解下`IllegalArgumentException`里面可能有啥，感觉就像是些参数的问题

    - `EmptyStackException`，从空栈中`poll`或者`peek`元素会产生的异常
    - `IllegalStateException`，无效状态异常，不懂

  可以大致上认为，如果一个异常是在运行时产生，并且不是`IO`类的，就是`RuntimeException`

- `AWTException`：和Windows的AWT有关的异常
- `InterruptedException`，在线程中断里面会涉及
- `CloneNotSupportedException`，克隆不被支持

经常使用的`extends Error`

我翻了下源码，好像没有什么特殊的`Error`，只要看到`Error`在某个类的名字就可以认为是继承自`Error`