反射：

在讲反射之前，看看类的加载机制：将class文件加载到内存

Javac将.java文件编译为二进制的.class文件，程序运行时，需要将此class文件加载到内存中，主要包含如下步骤：装载，链接(校验，准备，解析)，初始化

装载：通过类装载器查找并导入class文件，创建Class对象

链接：把类的二进制数据合并到JRE中

校验：检查载入的class文件的正确性

准备：给类的静态变量分配存储空间

解析：将符号引用转成直接引用

初始化：对类的静态变量，静态代码块执行初始化操作

装载 类加载器会将二进制文件加载到方法区，并将二进制的静态存储结构转化为方法区的运行时的数据结构，在堆中生成一个代表此类的java.lang.Class对象(每个类都对应一个Class对象)

链接/校验：为了保证class文件中的字节流包含的信息符合当前虚拟机要求

链接/准备：**在方法区中为静态变量分配内存**

链接/解析：将符号引用变换为直接引用(将常量池的符号引用转变为直接引用)，String str=”123”，将123替换为常量池中123的地址

初始化：到了初始化阶段才是真正开始执行类中定义的java代码，为静态变量，类变量进行初始化

类初始化的触发条件：

new，获取类的static变量，修改类的static变量，执行静态方法

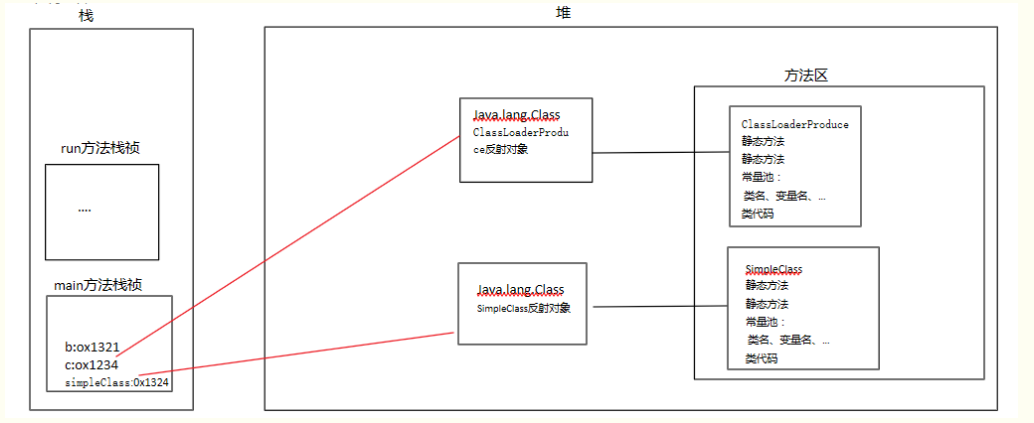
使用java.lang.reflect包的方法对类进行反射调用时

当初始化一个类时，如果其父类还没初始化，则需先初始化父类

当虚拟机启动时，用户指定执行的主类(main方法所在类)

类的加载是用到的时候才会加载到JVM中(按需加载)

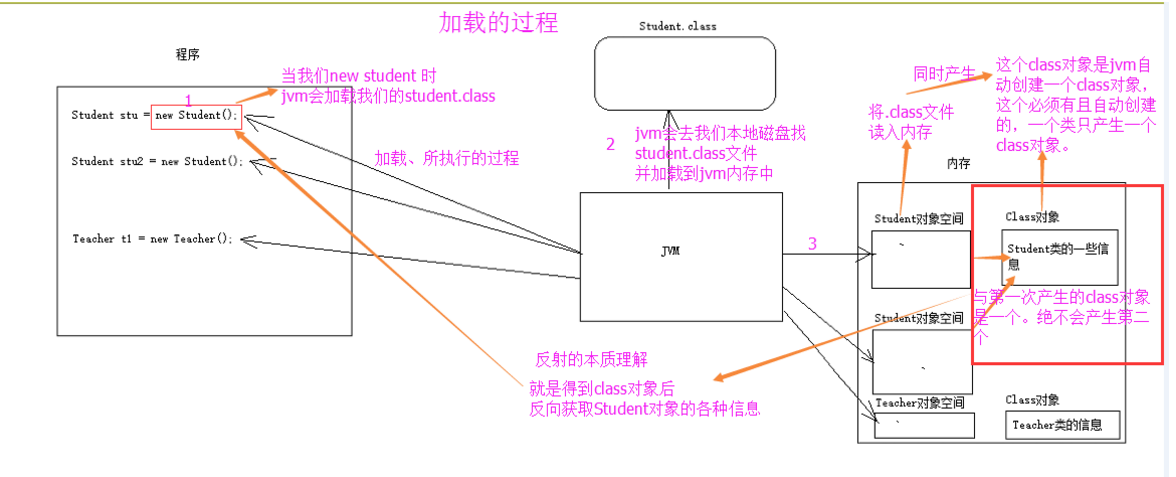
构造函数也是静态方法



Class对象：

当类被装载后，都会在堆中为此类创建一个Class对象，Class对象可以获取类的相关信息(方法，变量等)

Class类和Class对象：每个类都对应一个Class对象，Class对象就好比是Class类的实例，所有的Class对象都源于这个Class类



Class类也是一种类，和class关键字不同

获取类对应的Class对象的方式(如果没有，会将.classs文件加载到内存中)

Public class Person{}

获取Class对象的方式：

Class cl=Class.forName(“Xxx.Xxx.Person”);

Class cl=new Person().getClass();

Class cl=Person.class；Class字面常量，更简单，更安全，通过字面常量获取Class对象的引用不会自动初始化类(字面常量触发的是加载阶段)，即类装载后就可以获取到Class对象，而通过forName获取还要初始化类，执行java代码因此执行forName时需要捕获运行的异常，并且字面常量不仅可以应用在类上，还以应用在接口，数组以及基本数据类型(int.class)，

使用Class.forName加载Class对象时，不仅会将class文件加载到内存，而且还会对class进行初始化，而使用字面常量会延迟初始化加载，初始化加载延迟到对静态方法或非final静态域的首次引用

字面常量只会装载而不会初始化

Class.forName为什么会初始化类？

Class.forName是Class的一个静态方法，而所有类都属于Class，所以调用Class.forName实际上就是调用的自己的静态方法

获取Class对象和类的初始化是两回事：Class对象的创建在类被加载的时候就会创建，而类的初始化是在类加载后执行的

泛化的Class对象引用：

Class<Person> cl=Person.class：cl表示返回的是Person的Class对象

Class<?> cl=Person.class：cl表示任意Class对象的引用

Class<? extends Person> cl=Man.class(Man继承于Person)：cl引用的是Person的子类的Class对象

Instanceof和isInstance：

Instanceof是判断某个对象是否是某类的实例(if(obj instanceof Person))

IsInstance是判断对象的类型(if(Person.class.isInstance(obj)))

Instanceof和isInstance的结果都相同，只是使用上不同

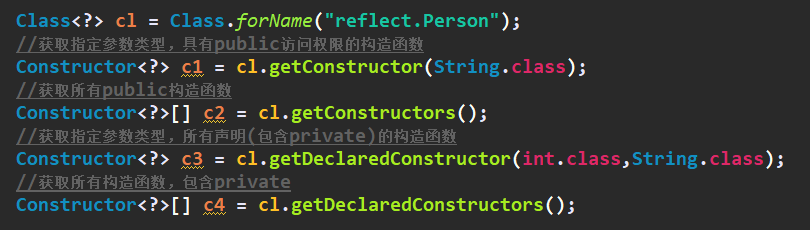
IsInstance是Class类的一个方法，而且是一个Native方法

Instanceof是关键字

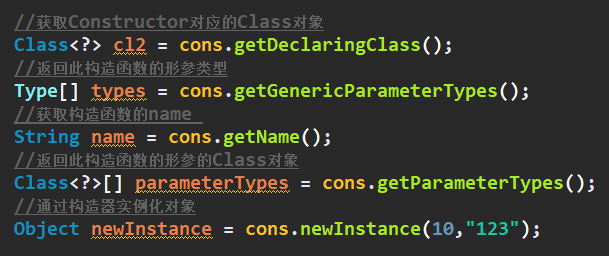
反射机制：在运行状态，对于任意一个类都能够知道这个类所有属性和方法，对于任意一个对象都能够调用他的任意属性和方法。

在java的反射中用的是Class类和java.lang.reflect类库下的类

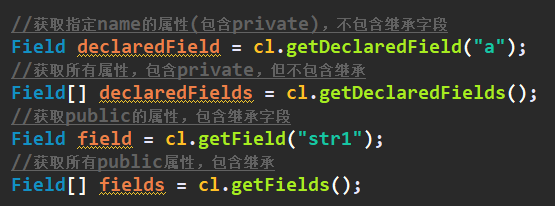
Constructor类是java.lang.reflect下的一个类，反映的是Class对象所对应的构造方法，通过Class对象获取Constructor的方式



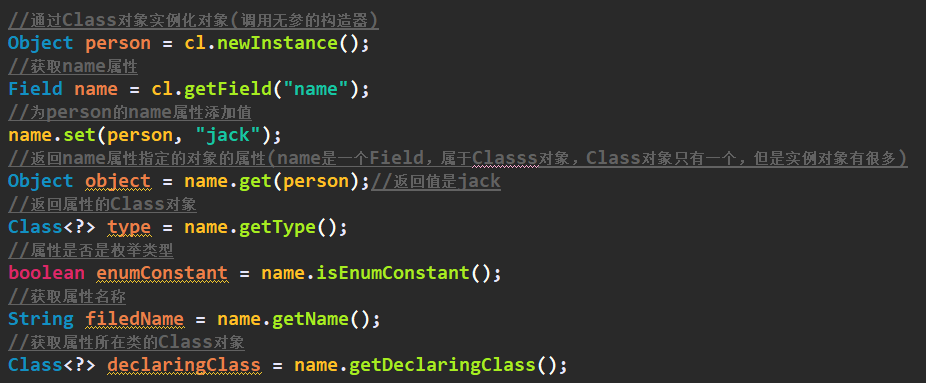
Constructor常用的方法



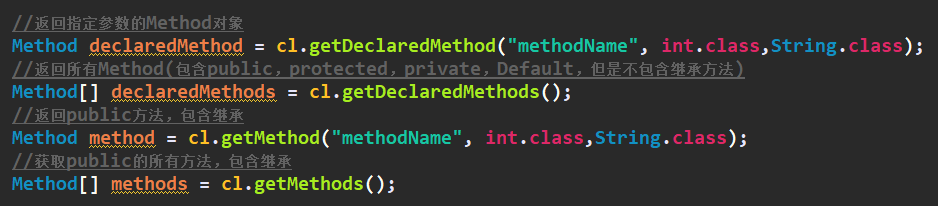
Field类是java.lang.reflect下的包，Filed是类的属性，通过Class对象获取Filed对象的方法



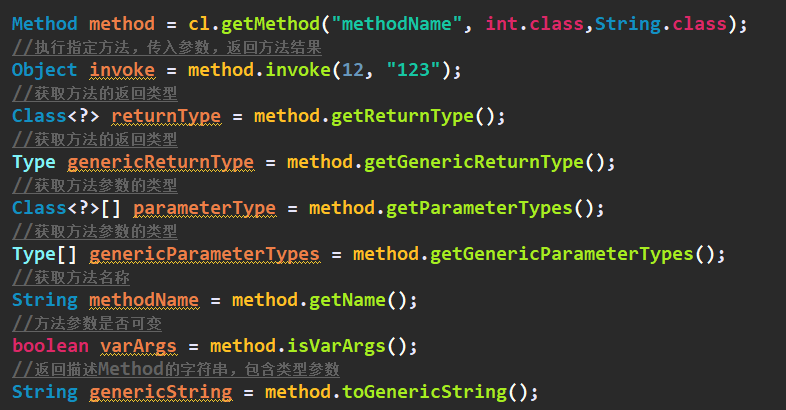
Filed常用方法



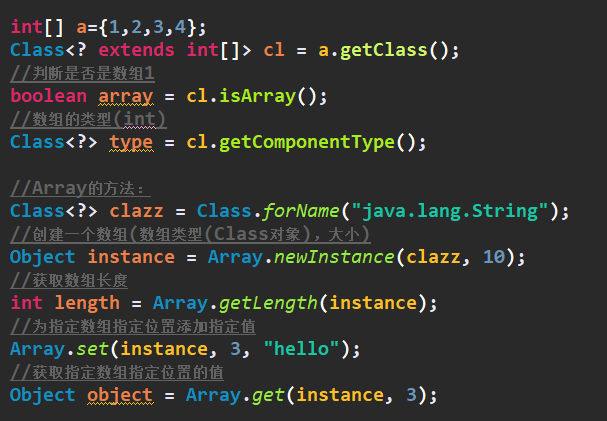
Method类是java.lang.reflect下的类，表示Class对象的方法，通过Class对象获取Method



Method常用方法



Java.lang.reflect包下有一个可以动态操作数组的类Array，它提供动态创建数组和访问数组的方法



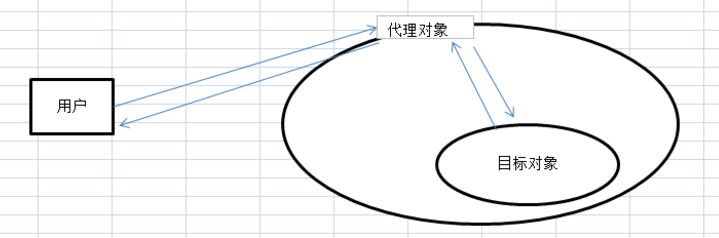
通过反射可以获取到Class的所有东西，继承的类，接口，方法(方法名，参数，返回值，修饰符等)，属性(属性名，属性类型，修饰符等)，构造函数(方法名，参数，修饰符)等等。

反射就是根据Class对象创建一个实例对象

Integer类型还可以：Class cl=Integer.TYPE;

代理：

代理模式是java常见的设计模式，当想在某个对象上进行修改或添加别的功能时，但是又不能直接操作目标对象时，可以通过创建代理对象来访问目标对象，代理对象在目标对象上进行添加功能

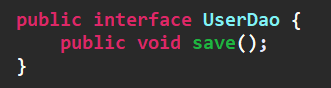


代理模式分为：静态代理，JDK动态代理，Cglib动态代理

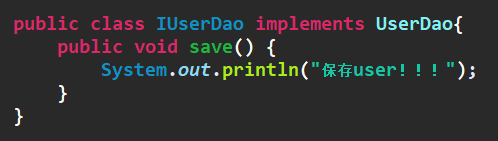
静态代理：静态代理要求代理对象和目标对象必须实现相同的接口或父类

实现过程：

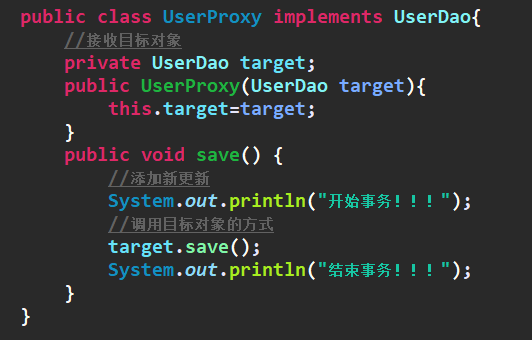
目标对象和代理对象实现的接口或父类



目标对象



代理对象



其实静态代理不需要目标对象和代理对象都实现接口和方法，只是实现后，感觉操作的就是目标方法

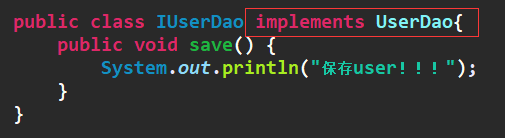
静态代理实际上就是方法与方法之间的调用

静态代理可以在不修改目标对象的前提下对目标功能进行扩展，但是静态代理对后期维护很不方便

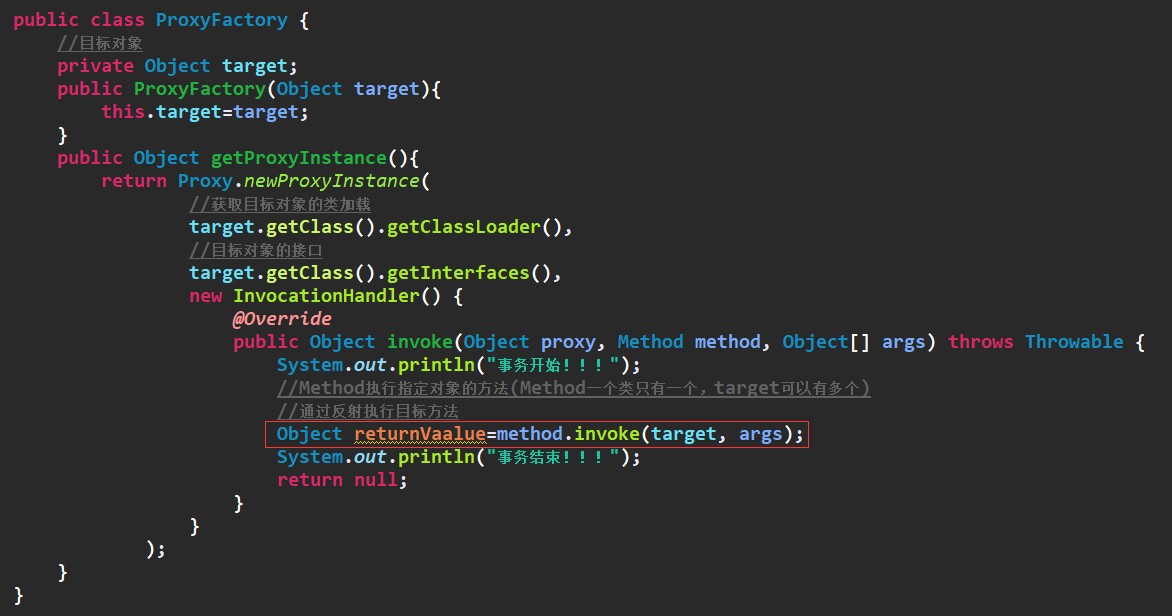
动态代理：动态代理是程序在运行过程中创建的，动态代理不需要代理对象实现接口，动态代理是通过JDK的API生成代理对象的，代理类所在的包：java.lang.reflect.Proxy

实现步骤

目标对象



创建动态代理

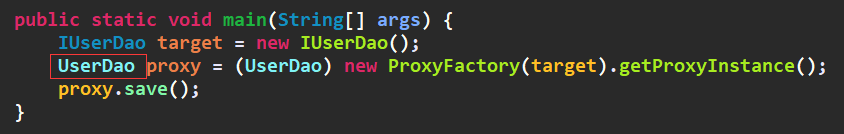


InvocationHandler：事件处理，执行目标方法时，会触发此事件

每一个代理对象都会有一个与之关联的InvocationHandler，调动代理对象的方法时，会执行InvocationHandler的invoke方法去执行目标对象的方法

Proxy.*newProxyInstance* ：创建代理对象

测试



动态代理的目标对象必须实现接口，这样创建的代理对象才可以和目标对象相关联

动态代理是代理对象通过反射来调用目标对象的方法

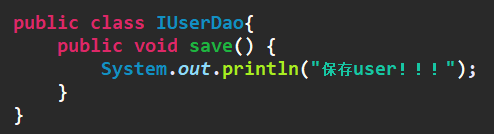
代理对象不需要实现任何接口，但是目标对象一定要实现接口

否则会抛：java.lang.ClassCastException: com.sun.proxy.$Proxy0 cannot be cast to target对象的类路径

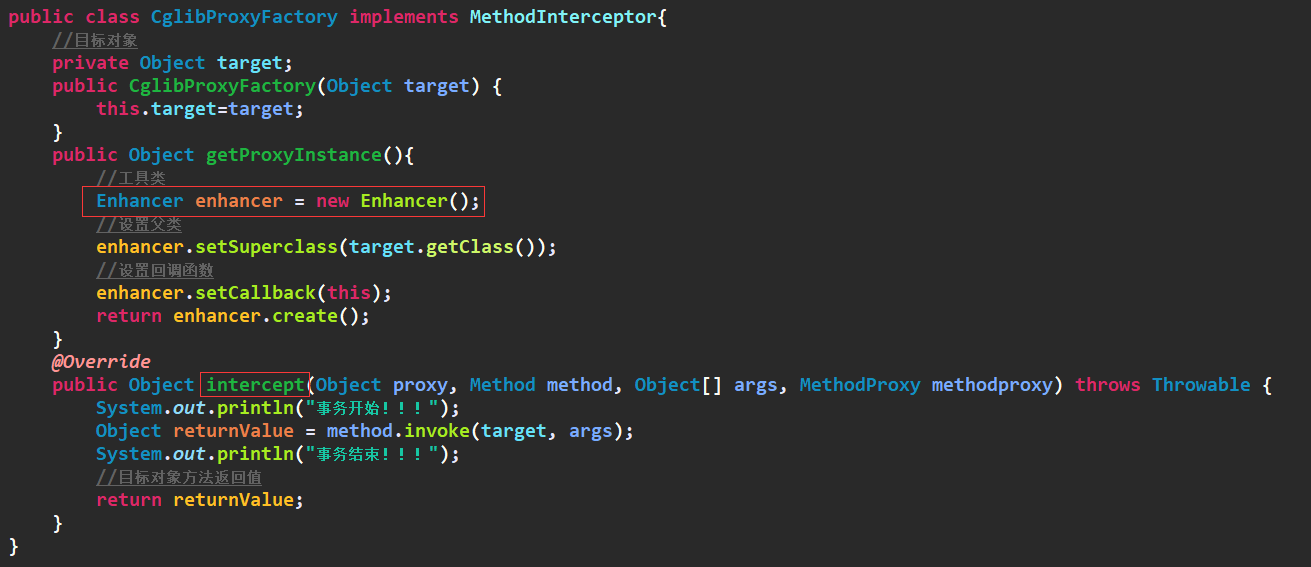
Cglib代理：Cglib代理不需要代理对象实现任何接口，Cglib是在内存中构建一个子类对象从而实现对目标对象功能的扩展，Cglib常用在AOP的框架中

实现步骤：先要导入cglib.jar

目标对象

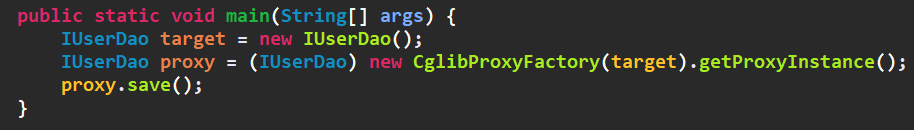


生成代理对象



实现MethodInterceptor接口

测试



Cglib是将目标对象作为父类，所以目标类不能是final/static，代理对象作为子类来实现代理的，通过Enchancer类来创建代理对象，当执行代理的方法时，会执行intercept方法去执行目标对象的方法

JDK动态代理/Cglib代理：

JDK动态代理目标对象必须实现接口，而Cglib代理不需要实现任何接口，目标类不能是final/static

Jdk通过反射来执行目标方法，Cglib是通过创建目标类的子类作为代理对象

jdk动态代理是由java内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。

JDK动态代理比Cglib代理效率高(低版本的Cglib高)

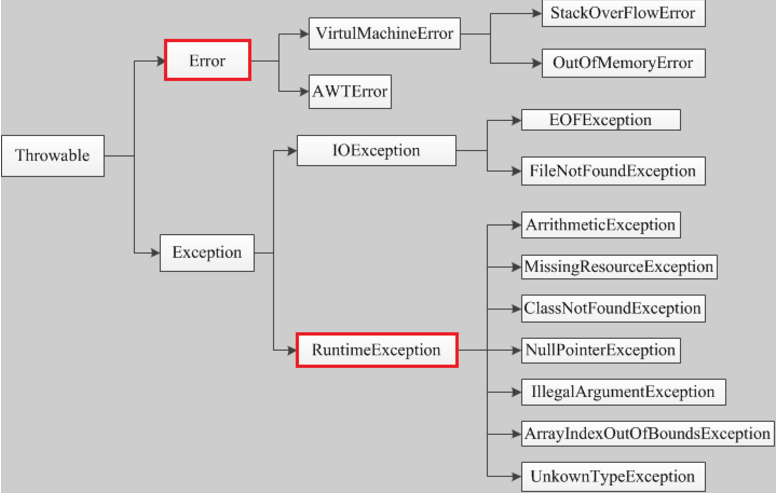
在Spring AOP中，两者方式都支持，可以在配置文件中设置采用哪种代理方式

JDK是通过实现共同接口将代理对象和目标对象相关联起来的，而Cgllib代理是通过父子类将代理对象和目标对象相关联的

异常：

异常处理机制能让程序在异常发生时，按照代码的预先设定的异常处理逻辑，针对性地处理异常，让程序尽最大可能恢复正常并继续执行，且保持代码的清晰。

Java程序中产生了异常，就会用一个对应类型的异常对象来封装异常，JRE就会试图寻找异常处理程序来处理异常。



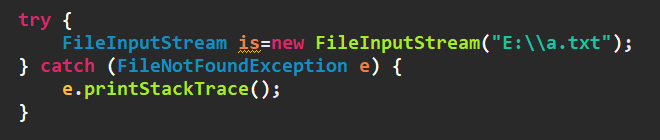
Error类的异常代表JVM本身异常(内存溢出等)，错误不能被程序员通过代码处理，所以Error很少出现

Exception代表程序在运行期间发生了不期望发生的事件，可以通过java异常处理机制处理

异常分为检查异常和非检查异常

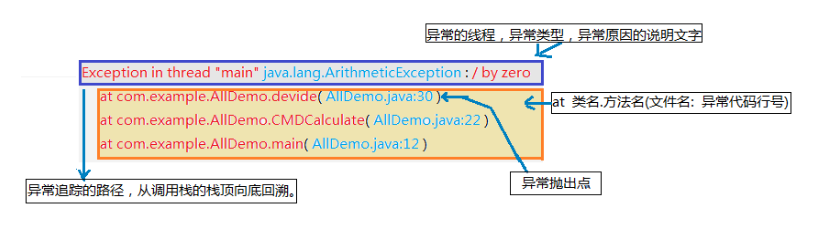
非检查异常：Error和RuntimeException属于非检查异常，javac在编译程序的时候不会提示出现异常，不会要求程序捕获异常，比如数组的下标越界，在编译的时候，不会检查是否越界，也不会要求程序捕获异常(运行是异常)

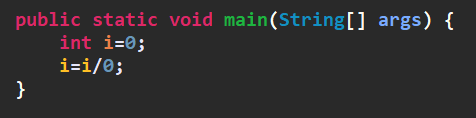
检查异常：除Error和RuntimeException之外的异常，javac在编译的时候要求程序捕获异常(try catch/throws)，程序可能被运行在各种未知的情况下，而程序员无法干预用户如何使用程序，因此需要提前捕获异常，比如程序在通过IO流读取用户输入的路径时，可能输入的路径不正确，因此需要对读取路径进行捕获异常(IOException)

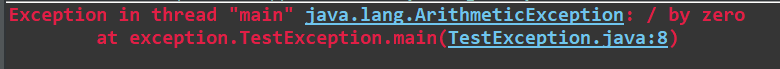


检查异常和非检查异常的区别在于非检查异常可以通过程序员尽可能的降低出错的几率(如数组下标越界)，而检查异常出现错误的几率很大，且程序员不能很好的控制

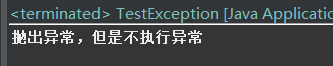
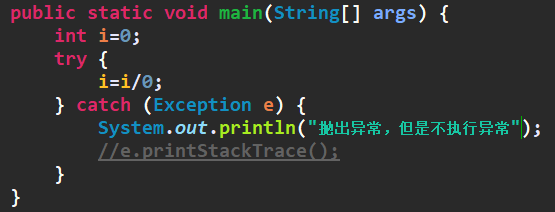
异常是在执行某个函数时引发的，而函数又是层级调用，形成调用栈的，因为，只要一个函数发生了异常，那么调用此异常的函数都会被异常影响，这些被影响的函数以异常信息输出，就形成了异常追踪栈







By/zero属于非检查异常，程序员在未捕获异常时，异常最终会抛给JRE



程序员也可以手动为非检查异常添加异常捕获，这里只是捕获到异常，但是并未处理异常

e.printStackTrace表示是否在栈中打印异常

程序捕获到异常后，可以终止程序的运行，也可以继续运行，不管此异常

异常的处理方式：try..catch..finally和throws

try{

可能会出现异常的代码

如果出现异常，则去匹配catch块中的异常

} catch (SQLException e){

Catch语句块可以有多个，但是每个catch块只能处理一种特点的异常或其子类异常，只有当try语句块中发送SQL的异常时，

才会执行当前catch语句块中的内容，否则到下一个catch语句块中判断

e.printStackTrace()：表示是否打印异常信息

} catch (Exception e){

//…..

}finally{

}

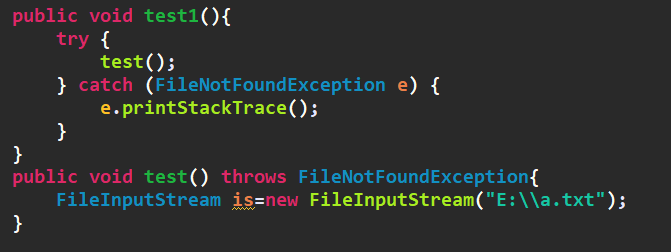
try..catch..finally中的局部变量不能共享

每一个catch块只能处理一个异常，try发生异常时，会顺序往下查询异常是否配置

异常发生后，异常后的代码都不会再执行，而去执行catch语句块中的内容

throws：

try..catch..finally是在异常发生的地方就处理了，而throws是将错误交由上一级的函数处理，自己不处理，交由调用者处理



throws/throw：

抛出异常的3种方式：throws，throw，系统自动抛出

系统自动抛出属于运行时异常(非检查异常) Int a=10/0;

Throw是手动抛出异常

String s=”123”;

If(“123”.equals(s)){

throw new NumberFormatException();

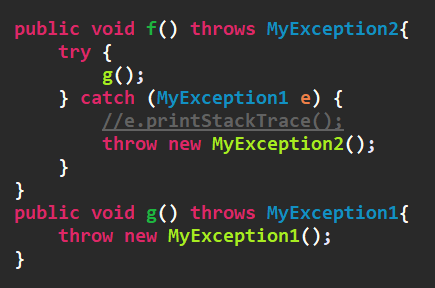
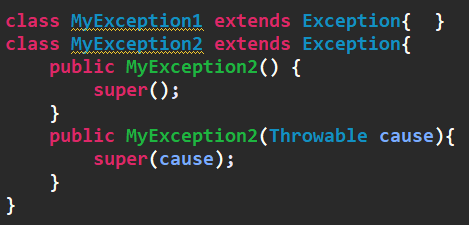
}else{}

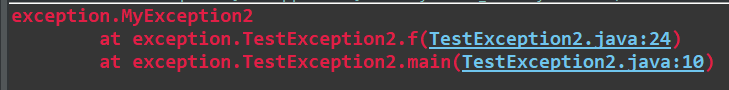
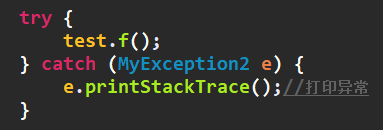
throws是声明方法可能会抛出的异常类型

public void fun() throws Execption1,Exception2{ }

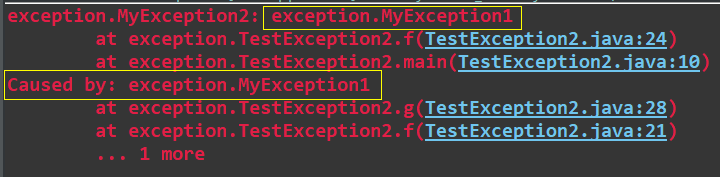
throws是声明方法可能出现的异常，而throw是抛出一个实实在在的异常

异常的链化





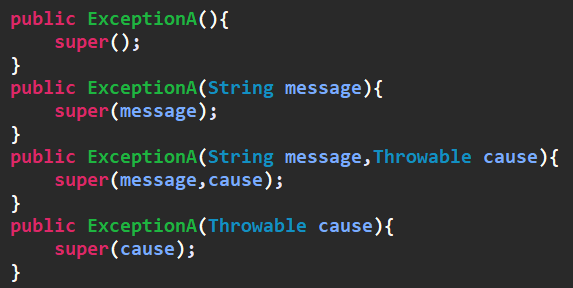
程序是在执行方法g的时候执行，f方法调用g，g发生异常，将异常交由调用者f执行，f交给main执行，在main中打印异常，但是只能知道异常在由f抛出的，但是即使上异常是由g抛出的



将g抛出的异常传递给f抛出的异常，这样就形成了链，在打印异常的时候，就会打印出来了

如果不添加g的异常，那么f抛出的异常将会把g抛出的异常给覆盖掉(向上抛异常，异常可能会被覆盖)

自定义异常：

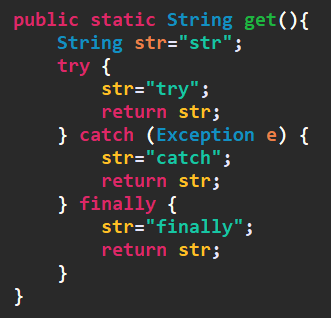
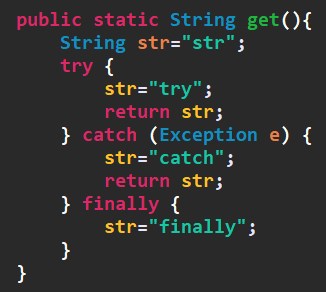


自定义异常应该包含这4个构造函数

message表示打印的异常信息

cause表示其他异常信息，在异常的链化时会用到

finally和return



第一个返回值是try，第二个返回值是finally，finally和return的执行顺序是，在执行return之前执行finally，finally执行完之后再去执行return，所以第一个返回值是try，虽然在finally中对str进行了修改，但是try中已经确定了返回值是try，而第二个返回值是finally，在执行try中的return之前执行了finally，在finally中对str进行了修改，并且return str，那么就没必须再去执行一次return了(一个方法中return只执行一次，并且第一出现就执行，但是finally是特殊)

即便在try块中有return，break，continue等改变执行流程的语句finally也会执行

当程序出错时，需要立即终止，使用非检测异常，不需调用代码显示捕获和处理(try..catch)，而且代码简洁明了

调用代码需要进一步处理和恢复时，使用检查异常，如SQLExecption时，抛出异常，需要关闭数据库连接操作等

ex.printStacktrace()：只是将错误输出到控制台，让程序员修改，这是适合在开发阶段，在实际运行时，应该将错误信息作为日志输出，可以自定义一个将错误输出到日志的异常类，当程序抛出异常时，不是ex.printStacktrace()而是 throw new logException(message，cause);在自定义的logException类中将错误输出到日志

方法A调用方法B，B抛出异常，在B中通过throw new Exception将异常抛给A，则在打印异常信息的时候，只用在A中打印，B中就不用打印了，避免重复打印