七、异常

正常情况意外的事件，具有不确定性。遇到异常后程序不能正常向下执行。例如：用户输入错误、除数为0，需要的文件不存在或打不开、数组下标越界、传入参数为空或不符合指定范围。

非运行时异常

1. 隐式声明抛出：异常类型是RuntimeException或是其子类，程序方法可以对异常不作任何声明抛出或处理，直接交给调用该方法的地方处理，程序能编译通过，不会对可能产生异常的代码行给出提示。
2. 显式异常抛出：如果main中某方法处发生异常，main不负责异常处理，由调用main方法的地方去处理异常，而调用main方法的是Java虚拟机，因此由Java虚拟机进行默认处理

```
public static void main(String args[]) throws IOException
```

3. try-catch-finally

```
try {
    statements
} catch(ExceptionType1 ExceptionObject) {
    Exception Handling
}
finally {
    Finally Handing
}
// 多catch情况：异常类由上到下 排布规则：由子到父，由具体到抽象，或为并列关系
```

自定义例外

1. throw new 异常类：异常对象大部分由系统产生；在编程时也可以主动产生异常

```
//throw new 异常类
static void throwOne() throws SelfGenerateException {
    int a = {
        // 如果a为1就认为在特定应用下存在异常，改变执行路径，抛出异常
        throw new SelfGenerateException("a为1");
    } 1;
    if(a==1)
}
```

八、Java常用类库和工具

Java类库概述

String

1. String

- String不是简单类型，而是一个类，它被用来表示字符序列，
- **Stirng对象是不可变的，在String类中每一个看起来会修改String对象内容的方法，实质都是创建了一个全新的String对象。**
- String作为参数传递的特点（传引用）

```
}
// 尽管是传引用，但通过形参引用并未改变实参指向的字符串内容
public class Str {
    public void changePara(String s) {
        s = s + "a";
    }
    public void invoke() {
        String s = "b";
        changePara(s);
        System.out.println(s)   // b
    }
    public static void main(String[] str) {
        Str s = new Str();
        s.invoke();
    }
```

- 字符串比较

```
// 引用比较
public class StringEqualTest {
    public static void main(String[] args) {
        String s  = new String("Hello");
        String s = new String("Hello");
        if(s==t) {
            System.out.println("相等")
        } else {
            System.out.println("不相等")    //不相等
        }
    }
}
// 值比较
public class StringEqualTest1 {
    public static void main(String args[]) {
        String s = new String("Hello");
        String t = new String("Hello");
        if(s.equals(t)) {
            System.out.println("相等")  // 相等
        } else {
            System.out.println("不相等")
```

```
        }
    }
}
```

- 编码：将Unicode字符集转换为本地字符集的过程
- 解码：将本地字符集转换为Unicode字符集的过程

StringBuffer

1. StringBuffer与String的区别：Strings are constant;their values cannot be changed after they are created. String buffers support mutalbe strings.

```
public class Str {
    public void changePara(StringBuffer s) {
        s = s.append("a");
    }
    public void invoke() {
        StringBuffer s = new StringBuffer("b");
        changePara(s);
        System.out.println(s);  //ba
    }
    public static void main(String[] str) {
        Str s = new Str();
        s.invoke();
    }
}
```

2. StringBuffer与String之间的转化

    - StringBuffer对象的值是可变的，对字符串的增加、插入、修改、删除等操作比String高效（不需要多次创建新的对象）

```
String st = new StringBuffer().append("a").append("b").toString();
```

3. 字符串使用小结

- String、StringBuffer、StringBuilder相同点
    1. 内部实现基于字符数组，封装了对字符串处理的各种操作
    2. 可自动检测数组越界等运行时异常
- 不同点
    1. String内部实现基于常量字符数组，因此内容不可变；SringBuffer、StringBuilder基于普通字符数组，内容可变，数组大小可根据字符串的实际长度自动扩容
    2. 性能方面，对于字符串的处理，相对来说StringBuilder>StringBuffer>String
    3. StringBuffer线程安全：StringBuilder非线程安全

系统类与时间类

- System类： 输入输出流、垃圾回收、系统属性、安全等其他方法
- Runtime类：每一个Java应用程序都有一个Runtime类的实例，允许应用程序与其运行的环境进行交互。要得到该类的实例，不能用new的方法产生，只能调用该类的getRuntime方法。该类的主要作用：启动进程、得到内存参数、System类许多方法的实现者
- Date类：一个存在于java.util包中。一个存在于java.sql包中，是前者的子类，用来描述数据库中的时间字段
- Calendar类：是对时间操作的主要类。要得到其对象引用，不能使用new，而要调用其静态方法getInstance，之后利用相应的对象方法

格式胡类

- 格式化数字：NumberFormat和DecimalFormat
- 格式化日期：SimpleDateFormat对象的format方法是将Date转为String，而parse方法是将String转为Date

九、线程的概念

线程与进程的概念

1. 一个进程就是一个执行中的程序，而每一个进程都有自己独立的一块内存空间、一组系统资源，每一个进程的内部数据和状态都是完全独立的。

2. Java程序执行中的单个顺序的流控制称为线程，多线程则指一个进程中可以同时运行多个不同的线程，执行不同的任务。

3. 线程和进程的区别：同类的多个线程共享一块内存空间和一组系统资源，而线程本身的数据通常只有微处理器的寄存器数据，以及一个供程序执行时使用的堆栈。所以系统在产生一个线程，或者在各个线程之间切换时，负担要比进程小的多。

4. 为什么需要多线程？提高CPU利用率

Thread类

public class Thread extends Object implements Runnable

```
public class MyThread extends Thread {
    public static void main(String args[]) {
        Thread a = new MyThread();  //(1)
        a.start();  //(2)
        System.out.println("This is main thread."); //(3)
    }
    public void run() {
        System.out.println("This is another thread.");  //(4)
    }
}
```

两种创建线程方法的比较

- 方法一：直接继承**Thread**类。（由于已经继承了Thread，不能再继承其他类了）

```
public MyThread extends Thread {
    // 创建及启动线程
    MyThread t = new MyThread();
    t.start();
    public void run()    { //线程体逻辑
    }
}
```

- 方法二：实现**Runnable**接口。当一个类已继承了另一个类时，就只能用实现Runnable接口的方式来创建线程；另外，使用此方法的更多原因是多个线程共享某个对象的资源

```
public MyThread implements Runnable {
    // 创建及启动线程
    MyThread t = new MyThread();
    Thread t1 = new Thread(t);
    t1.start();
    public void run()    {    // 线程体逻辑
    }
}
```

线程的控制与调度

1. 线程的生命周期和状态的改变：
    - 创建 start()
    - 可运行 <--> 运行
    - 阻塞 sleep() wait() join()
    - 死亡 完成或意外终止
2. 线程调度和优先级
    - 线程调度： 当有多个线程处于就绪状态时，线程调度程序根据调度策略安排线程的执行
    - 调度策略：
        - 抢占式调度
        - 时间片轮转调度
    - 线程优先级

线程同步机制

1. 线程安全问题：多线程同时访问共享资源（变量），导致线程安全问题。

```
public void push(char c) {
    data[index] = c;    //(1)
    index++;    //(2)
}
```

2. 对象监视器与synchronized

    - synchronized代码块：监视器就是指定的对象。
    - synchronized方法：监视器就是this对象。
    - synchronized静态方法：监视器就是相应的Class对象。

3. 死锁问题：如果多个线程都处于等待状态，彼此需要对方所占用的监视器所有权，就构成死锁，Java既不能发现死锁也不能避免死锁。

    - 可能发生死锁的代码执行中不一定会死锁，因为线程之间的执行存在很大的随机性。
    - 线程方法suspend()、resume()、stop()由于存在引起死锁的可能，因而逐渐不用。

线程的同步通信

```
// 只有存钱后且账上有钱才能取钱，并且只有取钱后才能存钱
class Account {
    volatile private int value;
    // 布尔标志
    volatile private boolean isMoney = false;
    // put设为同步方法
    synchronized void put(int i) {
        if(isMoney) {
            try { wait(); } // 线程等待
            catch(Exception e) {}
        }
        value = value + i;
        System.out.println("存入"+i+"掌上金额"+value);
        isMoney = true; // 设置标志
        notify();   // 唤醒等待资源的线程
    }
    synchronized void get(int i) {  // 同步方法
        is(!isMoney) {
            try{ wait(); }
            catch(Exception e) {}
        }
        if(value>i)
            value = value - i;
        else {
            i = value;
            value = 0;
        }
        System.out.println("取走"+i+"账上金额"+value);
        isMoney = false;
        notify();
        return i;
    }
}
```

多线程应用场景