第七章 异常和线程

🎉Java异常和并发😍

There is the hentai

1.异常的概念和类型

[TOC]

1.1 异常概念

异常，就是不正常的意思，程序中异常是指：

- 异常：在程序执行的过程中，出现的非正常的情况，最终会导致JVM的非正常停止

> 在Java 等面向对象的变成语言中，异常本身是一个类，产生异常就是创建异常对象并抛出了一个异常对象。

- **在Java中处理异常的方式是中断处理**

> 异常值得并不是语法错误。当语法错误时，编译不通过，不会产生字节码文件，程序根本不能运行

1.2 异常体系

异常机制其实在帮助我们找到程序中的问题，异常的跟类是`Java.lang.Throwable`

其下有两个子类：`Java.lang.Error`,`Java.lang.Exception`,

> 平常所说的异常是指`Java.lang.Exception`。

```
graph LR
A[异常]==>B(Error:不能处理,只能尽量避免)
A[异常]==>C(Exception:由于使用不当导致,可以避免)
```

throwable体系

- Error 指的是错误，用于知识合理的程序不应该试图捕获的**严重问题**，只能事先解决。
- Exception 表示异常，异常产生后程序员可以通过代码方式纠正，是程序继续运行，是必须要处理的。

throwable中的常用方法

- `public void printStackTrace ()` ：打印异常的详细信息。

包含了异常的类型，异常的原因，还包括异常出现的位置，在开发和调试阶段，都要使用.

- `public String getMessage()`:获取发生异常的原因。

1.3 异常分类

所有错误和异常的超类：`Throwable`

主要考虑的异常`Exception`，如果出现异常程序内继续运行

- `Exception`

```
 //编译异常
public static void main(String[] args)  {
       SimpleDateFormat sdf= new SimpleDateFormat("yyyy-MM-dd");
       Date date=sdf.parse("1909-09-09");
       System.out.println(sdf);
   }
```

```
 //解决方法一：
public static void main(String[] args) throws ParseException
    {
        SimpleDateFormat sdf= new SimpleDateFormat("yyyy-MM-dd");
        Date date=sdf.parse("1909-09-09");
        System.out.println(date);
        // Thu Sep 09 00:00:00 CST 1909
    }
```

```
//解决方法二：
public static void main(String[] args) {
        SimpleDateFormat sdf= new SimpleDateFormat("yyyy-MM-dd");
        Date date=null;
        try
        {
            date=sdf.parse("1909-09-09");
        }catch (ParseException e)
        {
            e.printStackTrace();
        }
        System.out.println(date);
        // Thu Sep 09 00:00:00 CST 1909
    }
```

//在异常代码加上 `sout("Hello world");` 程序可以继续运行

```
java.text.ParseException: Unparseable date: "1909-0909"
    at java.text.DateFormat.parse(DateFormat.java:357)
    at Exception.example01.main(example01.java:42)
hello world
```

- Error

```
public static void main(String[] args) {
    int []arr={1,2,3};
    try //可能会出现异常的代码
    {
        System.out.println(arr[3]);
    }catch (Exception e) //异常的处理逻辑
    {
        System.out.println(e);
    }//Exception
    int a[]=new int [1024*1024*1024];    //Error
    //必须修改代码，让创建的数组变小

}
```

1.4 异常过程解析

```
public static void main(String[] args) {
    /*
      定义一个方法，获取数组指定索引的元素
      参数:
      int arr[];
      int index ;
     */
    int arr[]={1,2,3};
    int e=getElement(arr,3);
    System.out.println(e);
    System.out.println("hello");

}
public static int getElement(int []arr,int index)
 {
     int ele=arr[index];
     return ele;
 }
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException Create breakpoint : 3
    at Exception.example01.getElement(example01.java:37)
    at Exception.example01.main(example01.java:31)
```

> 处理流程如下：

JVM 发现异常会做两件事：

- JVM 会根据异常原因产生一个异常对象包含了异常产生的（内容，原因，位置）
- 在getElement方法中，没有异常的处理器try--catch，那么JVM就会把异 常对象抛出给方法调用者main方法来处理这个异常。

Main方法：

- 接收异常对象，main方法没有异常的处理逻辑，继续把对象抛出给main方法的调用者JVM处理

JVM接受到这个异常对象做了两件事：

- 把异常对象（内容，原因，位置）以红色的字体打印在控制台
- JVM会终止当前正在执行的Java程序 --->中断处理

#2.异常的处理

Java 异常处理的5个关键字: try,catch,finally,throw,throws;

2.异常处理

2.1 抛出异常throw

在编写程序时，必须考虑程序出现问题的情况。

在Java中，提供了一个throw 关键字

1. 创建一个异常对象，封装一些提示信息。
2. 需要将这个异常对象通过throw关键字告知给调用者，throw异常对象

throw用在方法内，用来抛出一个异常对象，将这个异常对象传递到调用者出去，并结束当前方法的执行。

使用格式：

```
throw new 异常类名（参数）;
```

例如：

```
throw new NullPointerException("要访问的arr数组不存在");
```

throw关键字

作用： 可以使用throw关键字在指定的方法中抛出指定的异常。 使用格式： throw new xxxException("异常产生的原因"); Notice:

1. throw 关键字必须写在方法的内部
2. throw 关键字后边的new对象必须是Exception或者Exception的子类对象。
3. throw 关键字抛出指定的异常对象，我们就必须处理这个异常对象
    - throw 关键字后边创建的RuntimeException或者是RuntimeException 子类对象，交给JVM处理（打印异常对象，中断程序）
    - throw 关键字后边创建的是编译异常（写代码的时候报错），我们就必须处理这个异常
    - NullPointerException 是一个运行期异常，是RuntimeException的一个子类

```
 // 分析异常产生过程，如何处理异常
   public static void main(String[] args) {
      int arr[]=null;
       int e=getElement(arr,3);
       System.out.println(e);
   }
   public static int getElement(int arr[],int index)
   {

       if(arr==null)
       {
           throw new NullPointerException("输入的数组为空");
           //Exception in thread "main" java.lang.NullPointerException: 输入的数组
为空
       }
       if(index<0||index>arr.length-1)
       {
           throw new ArrayIndexOutOfBoundsException("数组越界");
           // Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
数组越界
       }
       int e=arr[index];
       return e;
   }
```

收获：

以后工作中，首先必须对方法传过来的参数进行合法性校验
如果参数不合法，我们就必须抛出异常的原因，告诉方法的调用者，传递的参数有问题
可以对传递过来的参数index 进行合法性校验
如果index的范围不在数组的索引范围内
那么我们就必须抛出数组索引越界异常，告知方法的调用者”传递的索引超出了数组的使用范围“

2.2 Objects非空判断

还记得我们学习过的一个类Objects吗，曾经提到过它由一些静态的使用方法组成，这些方法是null-save（空指针安全的）那么在它的源码中，对对象为null的值进行了爬出异常操作

- `public static <T>T requireNonNull(T obj)`:查看指定引用对象是不是null.

查看源代码时发现这里对null进行了抛出异常操作

以后对参数进行判断时可以直接使用Objects中的方法判断，简化代码

```
//e.g
       Objects.requireNonNull(arr,"数组为空");
       Objects.requireNonNull(arr);
```

2.3 声明异常 throws

声明异常：将问题表示出来，报告给调用者。如果方法内通过throw抛出了编译异常，而没有捕获处理，那么必须通过throws进行声明，让调用者去处理，所有RuntimeException都无需用throws声明。

声明异常格式：

```
修饰符 返回值类型 方法名（参数） throws 异常类名1，异常类名2...{}
```

```
throws关键字：异常处理的第一种方式，交给别人处理
作用：
      当方法内部抛出异常对象的时候，那么我们就必须处理这个异常对象
      可以使用throws关键字才处理异常对象，会把异常对象声明抛出个方法的调用者处理
      自己不处理，交给鄙人处理，最终交给JVM处理-->中断处理
使用格式：再方法生民时使用
     修饰符 返回值类型 方法名 （参数列表） throws AAAException BBBException...{
              throw new AAAException (”reasonAAA“)
              throw new AAAException (”reasonBBB“)
              .....
     }
```

注意：

1. throws 关键字必须写在方法声明处。
2. throws 关键字后边声明的异常必须时Exception或者是Exception的子类
3. 方法内部如果抛出了多个异常对象，那么throws后面必须也声明多个异常，如果抛出的多个异常对象由父子类关系那么直接声明父子类异常即可。
4. 调用了一个声明处异常的方法，就必须处理异常，要么继续throw，最终交给JVM中断处理，要么用try..catch自己处理异常.

声明异常的代码演示：

```
 //public static void main(String[] args) throws FileNotFoundException,IOException
   //public static void main(String[] args) throws IOException; IOException
extends Exception
   //IOException extends Exception
   public static void main(String[] args) throws Exception {
         readFile("D:\\文档\\阿门Home\\Java异常和线程.md");
   }
    public static void readFile(String fileName) throws
FileNotFoundException,IOException {
      //检查当前文件路径是否存在，如果不是抛异常
      if(!fileName.equals("D:\\文档\\阿门Home\\Java异常和线程.md"))
      {
         throw new FileNotFoundException("DD");
      }
      //检查文件后缀名是否为.txt 若不是则抛出异常
```

```
        if(!fileName.endsWith(".txt"))
        {
            throw new IOException("没有找到该文件");
        }
    }
```

```
graph LR
A[Exception]==>B[IOException]
A[Exception]==>C[RuntimeException]
C[RuntimeException]==>D[NullPointerException]
C[RuntimeException]==>E[OutOfBoundException]
B[IOException]==>F[FilewNotFoundException]
...
```

2.4 捕获异常try…catch

当使用throw后，程序抛出异常，最终导致JVM中断处理，而try…catch 会自己处理异常，可以继续处理后续程序。

格式

```
try{
    //可能产生异常的代码
}catch(异常类名 变量名)//定义一个异常的变量，用来接收try中抛出的异常对象
{
    异常的处理逻辑，即如何处理异常对象，
}
```

注意：一般在工作中，会把异常的信息记录到一个日志中。

1. try中可能会产生多个异常对象，那么可以使用多个catch来捕获处理这些对象
2. 如果try中出现了异常，那么就会执行catch中的异常处理逻辑，执行完毕后继续执行后面的代码，如果try中没有异常，不会执行catch中的处理逻辑，继续（try…catch体之后的代码）执行之后代码

```
public static void main(String[] args) {
        try{
            readFile("D:\\文档\\阿门Home\\Java异常和线程.md");
        }catch (IOException ioe)
        {
            System.out.println("IOEProblem");
        }
       System.out.println("hello world");
   }
    public static void readFile(String fileName) throws IOException {
      if(!fileName.equals("D:\\文档\\阿门Home\\Java异常和线程.md"))
      {
          throw new FileNotFoundException("DD");
```

```
        }

        if(!fileName.endsWith(".txt"))
        {
            throw new IOException("没有找到该文件");
        }
    }
```

```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" ...
IOEProblem
hello world
```

如何获取异常信息：

Throwable (异常和错误的超类)中定义了一些查看方法：

- public String getMessage(): 获取异常的表述简短信息，提示错误原因
- public String toString():获取异常的类型和异常详细描述信息（一般不用）
- public void printStackTrace():打印异常的跟踪站信息并输出到控制台。

包含了异常的类型，异常的原因，还包括异常出现的位置，在开发和调试阶段都要

使用printStackTrace；JVM默认调用此方法

```
 System.out.println("IOEProblem");
            System.out.println(ioe.getMessage());//没有找到该文件
            //重写Object类的toStirng方法
            System.out.println(ioe.toString());//java.io.IOException: 没有找到该文
件
            ioe.printStackTrace();//程序中断
//            java.io.IOException: 没有找到该文件
//            at Exception.example01.readFile(example01.java:51)
//            at Exception.example01.main(example01.java:32)
```

2.5 finally代码块

有一些特定的代码块无论异常是否发生，都需要指向，另外，应为异常会引发程序跳转，导致有些语句执行不到。而finally就是解决这个问题的，finally代码块中存放的代码一定会被执行的。

当我们在try语句中带开了一些物理资源（磁盘文件/网络连接/数据库连接等），我们都得在使用完成之后，最终关闭打开的资源。

又如：当学习IO流之后，当打开一个关联文件的资源，最后程序不管结果如何，都需要把这个资源关闭。

finally语法

```
try...catch...finally//自身需要处理异常，最终还得关闭资源
```

> 注意

1. finally不能单独使用，必须和try一起使用。
2. finally 一般用于资源释放（资源回收），无论程序是否出现异常，最后都要释放资源（IO）

参考代码：

上面的代码块hello world放入finally中

```
//Main: :
try{
        readFile("D:\\文档\\阿门Home\\Java异常和线程.md");
         }catch (IOException ioe)
         {
             System.out.println("IOEProblem");
             ioe.printStackTrace();//程序中断
         }finally{
             System.out.println("hello world");
         }
}
```

2.6 多异常的捕获

1. try中可能会产生的异常对象：

```
new ArrayIndexOutOfBoundsException("3");
new IndexOutOfBoundsException("3");
```

try中如果出现异常对象，就会把异常对象抛出给catch处理，抛出的异常对象，会从上到下一次赋值给catch中定义的异常。

多个异常的处理

1. 多个异常分别处理。

   代码：

```
//1.多个异常分别处理
    public static void main(String[] args) {
        int arr[]={1,2,3};
        try{
            System.out.println(arr[3]);
        }catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
```

```
        }
        try{
            List<Integer>list=new ArrayList<>();
            list.add(1); list.add(1); list.add(1);
            System.out.println(list.get(3));
        }catch(IndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
    }
```

> 结果

```
java.lang.ArrayIndexOutOfBoundsException: 3
java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
```

2. 多个异常一次捕获多次处理。

   代码：

```
 //ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException
public static void main(String[] args) {
        try{
            List<Integer>list=new ArrayList<>();
            list.add(1);list.add(1);list.add(1);
            System.out.println(list.get(3));
        }catch(ArrayIndexOutOfBoundsException AIE){
            System.out.println(AIE);
        }catch(IndexOutOfBoundsException IE){
            System.out.println(IE);
        }
    }
//java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
```

> 注意：
>
> > catch里边定义的异常变量，如果有子父类关系，那么子类的异常变量必须写在上边，否则就会报错

3. 多个异常一次捕获，一次处理。

```
public static void main(String[] args) {
        try{
            List<Integer>list=new ArrayList<>();
            list.add(1);list.add(1);list.add(1);
```

```
            System.out.println(list.get(3));
        }catch(Exception e){
            System.out.println(e);
        }
    }
```

运行时抛出的异常可以不处理，即不捕获，也不声明抛出，默认交给虚拟机处理，什么时候不抛出运行时异常了，再处理。

- 如果finally里面有return语句，永远返回finally中的结果，避免该情况。
- 子父类：
- 如果父类抛出了多个异常，子类重写父类方法时，抛出和父类相同的异常/抛出父类的异常子类/不抛出异常。
- 父类没有抛出异常，子类重写该方法时也不可抛出异常。此时子类产生该异常，只能捕获处理，不能声明抛出
- 在try/catch后可以追加finally代码块，其中的代码一定会被执行，通常用于资源回收。

3.自定义异常

> Java提供的异常类，不够我们使用，我么需要自定义一些异常类

格式：

```
public class xxxException extends Exception|RuntimeException {
    - 添加一个空参数的构造方法
    - 添加一个带异常信息的构造方法
}
```

注意：

- 自定义异常一般都是以Exception结尾，用以说明此类是一个异常类
- 自定义异常类必须继承自Exception或RuntimeException
    - 继承自Exception：那么就是一个编译期异常，如果方法内部抛出了编译期异常，必须处理这个异常，要么throw，要么try..catch;
    - 继承自RuntimeException：那么定义的异常类无需处理，交给虚拟机处理。

代码实例：

```
package Exception;
//在Exception包下声明自定义异常类注册异常类
public class RegisterException extends Exception {
    public RegisterException()
    {
        super();
    }
    public RegisterException(String s)
    {
         super(s);
```

```
    }
}
```

实例：

```
package Exception;

import org.w3c.dom.ls.LSOutput;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.*;
import java.util.Date;
import java.util.Objects;
import java.lang.Exception;
public class example01 {
    static String[] usernames={"shx","lt","skks"};
    public static void main(String[] args) throws RegisterException {
     Scanner sc =new Scanner(System.in);
     String username=sc.next();
     checkUserName(username);
    }
    public static void checkUserName(String username) throws
RegisterException {
        for(String name:usernames)
        {
            if(name.equals(username))
            {
                throw new RegisterException("用户名已经被注册了");
            }else continue;
        }
        System.out.println("恭喜你注册成功！");
    }
}
```

问题：

1. 父类中未声明抛出异常的方法，子类重写时允许增加抛出checked类型异常声明吗，为什么？

> 答：在Java中，子类重写方法时允许增加抛出checked类型异常声明。这是因为子类重写方法时可以添加更多的异常检查（checked exceptions）来反映其特定实现的情况。
>
> 当然，在进行此操作时需要遵循以下规则：
>
> 1. 子类添加的异常必须是父类方法中未声明的异常。

> 2. 子类添加的异常必须是checked类型异常。
> 3. 子类抛出的异常不能比父类抛出的异常更通用。
>
> 如果违反任何一条规则，则编译器将无法编译代码。
>
> 需要注意的是，如果父类方法抛出了checked类型异常，则在子类覆盖该方法时，要么不抛出异常，要么只能抛出具有相同或更具体类型的异常。
>
> 总之，子类重写父类方法时，允许添加抛出checked类型异常声明，但需要遵守一定的规则以确保代码健壮性和正确性。

2. java允许自定义unchecked类吗?

> 答：Java允许自定义未检查异常类。在Java中，未检查异常通常是RuntimeException或其子类。通过扩展RuntimeException或其子类，您可以创建自己的未检查异常类。
>
> 例如，您可以创建一个称为MyException的未检查异常类，如下所示：
>
> ```
> public class MyException extends RuntimeException {
>        public MyException(String message) {
>           super(message);
>        }
> }
> ```
>
> 然后，在您的代码中，您可以像任何其他未检查异常一样使用MyException：
>
> ```
> throw new MyException("Something went wrong");
> ```

3. 出现异常时，java会沿方法调用栈寻找最近一个专门捕获处理与此异常类型相同的程序，而非能处理此异常类型的程序吗?

> 是的，在Java中，当发生异常时，程序会沿着调用方法的栈向上查找，寻找最近的专门捕获和处理此异常类型的程序，而不是能够处理该异常类型的程序。在方法中抛出异常时，可以使用try-catch语句来捕获并处理异常。
>
> 如果没有相应的catch块来处理异常，则该异常将被传递到调用该方法的方法中，如果该方法也没有相应的catch块来处理异常，则该异常将继续向上传递，直到找到包含相应的catch块来处理该异常为止。如果在整个调用堆栈中都没有找到任何一个处理该异常的catch块，则程序将会终止执行，并打印异常信息。
>
> 需要注意的是，Java要求catch块按照异常的子类相对于父类的顺序排列。这种排列方式可以确保兼容性，即在多态的情况下，能够正确地识别出异常的类型并进行处理。
>
> 总之，Java会沿着方法调用栈向上查找最近的专门捕获处理与此异常类型相同的程序，以确保异常能够得到适当的处理。

4. 所有可能出现的异常都必须被显式捕获处理，否则无法通过编译吗?

> 在Java中，并不是所有可能出现的异常都必须被显式捕获处理，否则无法通过编译。只有checked类型的异常（也称为受检查异常）才需要显式捕获处理或者在方法签名上抛出异常。
>
> 对于unchecked类型的异常（也称为未受检查异常），则不需要强制进行异常处理，但是建议在代码中进行适当的异常处理。当然，如果未能正确处理这些异常，程序运行时可能会引起一些问题，例如异常信息的输出和程序的崩溃等。
>
> 此外，在Java 7及更高版本中，还可以使用try-with-resources语句来自动关闭资源，同时避免了由于忘记关闭资源而导致的潜在问题。
>
> 总之，在Java中，只有checked类型的异常需要强制进行异常处理，而对于unchecked类型的异常，则不需要强制进行异常处理，但是需要在代码中进行适当的异常处理来确保程序的健壮性和可靠性。

5. 可以不在出现异常的方法内捕获处理异常吗?

> 1. 在该方法的调用者中捕获和处理异常。
> 2. 将异常传递给更高层次的调用者；如果在整个调用堆栈中都没有处理该异常，则程序将终止执行，将异常信息输出到控制台或记录在日志文件中。
>
> 这三种方式都可以用于处理异常，选择哪种方式取决于具体情况。通常，在处理异常时，建议将代码与业务逻辑分开，并遵循异常传递机制（也称为异常转发机制）。
>
> 需要注意的是，在使用try-catch语句和处理异常时，应该尽可能地减小try块的范围，只让可能抛出异常的语句放在其中，以避免不必要的捕获和处理异常。
>
> 总之，在Java中，不需要在出现异常的方法内捕获和处理异常，可以将异常传递给更高层次的调用者进行处理。但是，应该注意良好的异常处理实践，确保程序的健壮性和可靠性。

6. 与外部资源互交时，可能出现Unchecked类型异常吗?

> 在Java中，当与外部资源（如文件、网络和数据库等）进行互交时，可能会出现未检查异常（Unchecked Exception），比如NullPointerException、IllegalArgumentException等等。这些异常通常是由于资源访问或数据传输的问题导致的。
>
> 例如，在使用Java IO类读取文件时，可能会发生FileNotFoundException或IOException等未检查异常。同样，在使用Java Socket类进行网络编程时，也可能会出现连接超时或网络故障等未检查异常。
>
> 需要注意的是，虽然这些未检查异常在Java编程中很常见，但它们并不是所有与外部资源互交时都可能出现的异常。某些异常，如SecurityException、IOException等可能是checked类型异常，需要在代码中显式捕获和处理。
>
> 总之，在Java中，与外部资源互交时可能会出现未检查异常，开发者应该在代码中进行适当的异常处理来确保程序的健壮性和可靠性。

7. 抽象方法可以抛出异常吗?

> 在Java中，抽象方法是不允许声明抛出异常的。这是因为抽象方法必须在具体的子类中实现，而子类可以覆盖父类中的异常声明。如果在抽象方法中声明了异常，则子类可能需要增加相同的异常声明或者更具体的异常声明，这样就会导致冲突和不一致。
>
> 具体地说，如果一个接口或抽象类中的方法具有异常声明，则任何实现该接口或继承该抽象类的具体子类都必须捕获并处理这些异常。但是，对于抽象方法来说，它只是一个声明，在具体的子类中进行实现

> 时，可以选择增加相同的异常声明、更具体的异常声明或者不声明异常。
>
> 因此，在Java中，抽象方法不允许声明抛出异常，否则会导致子类无法实现该方法或者需要增加冲突的异常声明，从而违反了Java语言的继承和多态机制。
>
> 总之，在Java中，抽象方法不能声明抛出异常，如果需要在具体的子类中抛出异常，则需要在实现该方法的子类中添加合适的异常声明。

8. 方法声明抛出异常会影响方法签名吗?

> 在Java中，方法声明抛出异常会影响方法签名。一个方法的签名由方法名和参数列表组成，如果两个方法在方法名称和参数列表上完全一致，但是其中一个方法声明了抛出异常，而另一个没有声明抛出异常，则它们的方法签名也是不同的。

4.多线程

4.1并发和并行

> - 并发：指的是两个或者多个线程在同一个时间段内发生（一个CPU交替执行多个任务）。

> - 并行：指的是两个或者多个线程在同一时刻发生。（多核CPU同时执行）

4.2线程和进程

- 进程是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以（拥有）同时运行多个进程；进程是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序就是一个进程的创建，运行，消亡的过程。
- 线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中可以有多个线程，这个应用程序一可以称之为多线程程序。

> 总结：一个程序至少有一个进程，一个进程中可以有多个线程。

实例电脑管家：

> 点击功能（病毒查撒谎，清理垃圾，电脑加速）执行后
>
> 就会开启一条应用程序到CPU的执行路径，CPU就可以通过这个路径执行功能，这个路径有一个名字，叫做线程，CPU在多个线程之间高速切换

4.3 线程调度

- 分时调度

  所有线程轮流使用CPU，平均每个线程专用CPU时间

- 抢占式调度

  让优先级高的线程优先使用CPU，如果线程优先级相同，则会随机选择一个（线程随机性），Java使用抢占式调度

  - 设置线程优先级：可以在任务管理其中设置（win11 中详细信息）

4.4 主线程

- 主线程：执行主方法（main方法）的线程。
- 单线程程序：Java程序中只有一个线程
    - 执行从main方法开始，从上到下一次执行

4.5 创建线程类

4.5.1.Java.lang.Thread

> 创建线程的方法：

- 创建一个继承自Thread的子类
- 在子类中重写run方法，设置线程任务

> 调用线程的步骤：

- 创建上述Thead的子类对象
- 调用Thread 类中的start方法，执行run方法
- start方法会开辟新的栈空间，从而进行多线程处理，多个线程之间互不影响，而仅调用run方法则是单线程

> 代码：

```
package Exception;

public class MyThread01 extends Thread {
    @Override
    public void run() {
        for(int i=2;i<30;i++)
        {
            System.out.println("to1"+i);
        }
    }
    @Override
    public synchronized void start() {
        super.start();
    }
}
```

> 分析：main线程，thread01和thread02并发，抢夺线程执行

```
public class ThreadTest {
    public static void main(String[] args) {
        MyThread01 mythread01=new MyThread01();
        MyThread02 mythread02 = new MyThread02();
        mythread02.start();
        mythread01.start();
        for(int i=1;i<=30;i++)
        {
            System.out.println("main"+i);
```

```
        }
    }
}
```

4.5.2.Runnable Interface

- 创建Runnable 方法的实现类RunnableImpl
- 在主函数中创建一个runnableImpl对象重写run方法，调用Thread中的构造函数

  Thread (Runable) 创建一个新线程
- 最后用start方法卡其线程

代码：

RunnableImpl:

```
public class RunableImpl implements Runnable{
    @Override
    public void run() {
        for(int i=1;i<30;i++)
        {
            System.out.println("run-->"+i);
        }
    }
}
```

Test:

```
public class ThreadTest {
    public static void main(String[] args) throws InterruptedException {
        RunableImpl run=new RunableImpl();
        Thread t=new Thread(run);
        t.start();
        for(int i=0;i<30;i++)
        {
            System.out.println("main"+i);
        }
    }
}
```

4.5.3 Runnable 接口的好处

- 避免了单一继承的局限性

  一个类只能继承一个父类，而Runnable接口可以继承其他的类，实现其他的接口

- 增强了程序的拓展性，降低了程序的耦合度

  实现Runnable接口的方法，吧设置线程任务和开启新线程进行了分离。

  可以通过传入不同Runnable接口来创建不同的线程；

4.5.4 匿名内部类创建线程

- 没有名字，写在其它类内部
- 作用：简化代码，把继承，重写，创建子类对象一步完成。

代码：

```
public class ThreadTest {
    public static void main(String[] args) throws InterruptedException {
        new Thread(){
            @Override
            public void run() {
                for(int i=0;i<30;i++)
                {
                    System.out.println("my-->"+i);
                }
            }
        }.start();
        new Thread( new Runnable(){
            @Override
            public void run() {
                for(int i=0;i<30;i++)
                {
                    System.out.println("thread-->"+i);
                }
            }
        }).start();


    }
}
```

4.6 Thread类中的方法

- 获取线程的名称：
  - 使用Thread类中的getName（）方法返回线程名称
  - 可以获取到当前从正在执行线程，currentThread（）返回当前正在执行线程名称.

    代码：

```
public class ThreadTest {
    public static void main(String[] args) {
        MyThread01 mythread01=new MyThread01();
        MyThread02 mythread02 = new MyThread02();
        mythread02.start();
        mythread01.start();
        new MyThread01().start();
        Thread t=Thread.currentThread();
        System.out.println("t=="+t.getName());
    }
}
```

- 设置线程的名称：
  - 使用Thread类中的setName方法
- 使得当前执行的线程以指定的毫秒数暂停一次（频率）

  public static void sleep(long millis)

  调用：Thread.sleep()

4.7线程安全问题

- 单线程程序不会出现线程安全问题
- 多线程，没有访问共享数据，也不会产生线程安全问题
- 多线程，访问共享数据，有可能产生线程安全问题

4.7.1 线程同步技术

- 同步代码块：使用synchronized关键字

  代码RunnableImpl：

```
public class RunableImpl implements Runnable{
    private  int ticket = 100;//线程共享资源
    Object obj =new Object();//锁对象
    @Override
    public void run() {
         while(ticket>0)
         {
             synchronized (obj)
             {
                 if(ticket>0)
                 {

System.out.println(Thread.currentThread().getName()+"buy one ticket --
>"+ticket);
                     ticket--;
                 }else
                 {
```

```
                    System.out.println("fail to do so");
                }
            }
        }

    }
}
```

代码Main:

```
public class ThreadTest {
 public static void main(String[] args) throws InterruptedException {
        Runnable r = new RunableImpl();
        Thread t0 = new Thread(r);
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t0.start();
        t1.start();
        t2.start();
    }
   }
```

同步技术的原理:使用了锁对象,这个锁对象叫做同步锁,也叫做对象所,也叫对象监视器,同步中的线程,没有执行完毕不会释放锁,同步外的对象没有锁不能进行同步

- 使用同步方法

  原理:

  - 定义一个同步方法会把方法内部代码锁住
  - 只让一个线程执行该方法
  - 锁对象如果时静态同步方法,锁对象时本类对象的类文件Runable.getClass();

    否则就是this.

  代码:

```
private static int ticket = 100;
  Object obj =new Object();
  @Override
  public void run() {
       while(ticket>0)
       {
           payTicket();
       }
  }
  public static synchronized  void payTicket()
  {
```

```
            if(ticket>0)
            {
                System.out.println(Thread.currentThread().getName()+"buy one
    ticket -->"+ticket);
                ticket--;
            }else
            {
                System.out.println("fail to do so");
            }
        }
```

4.7.2 Lock(接口)锁

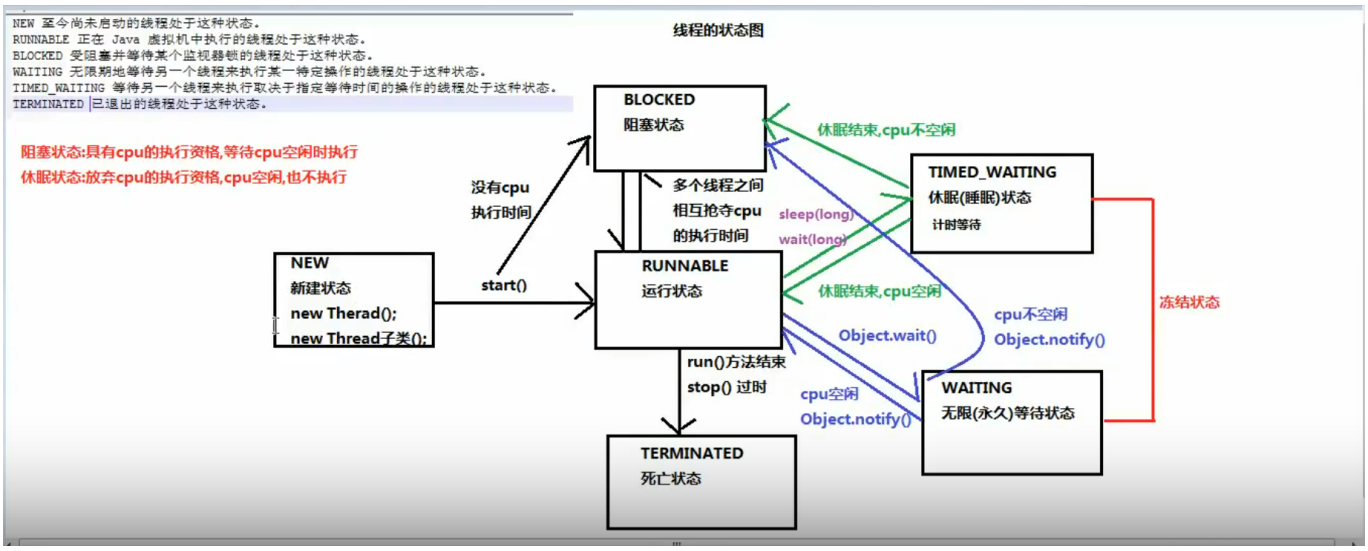
- lock/unlock 获取锁,释放锁
  - 成员内部创建lock实现类例如系统提供的ReentrantLock();
  - 在可能会出现线程安全问题的代码块前使用lock获取锁.
  - 在可能会出现线程安全问题的代码块后使用unlock释放锁.
- unlock可以放入finally代码块中提高程序运行效率

代码:

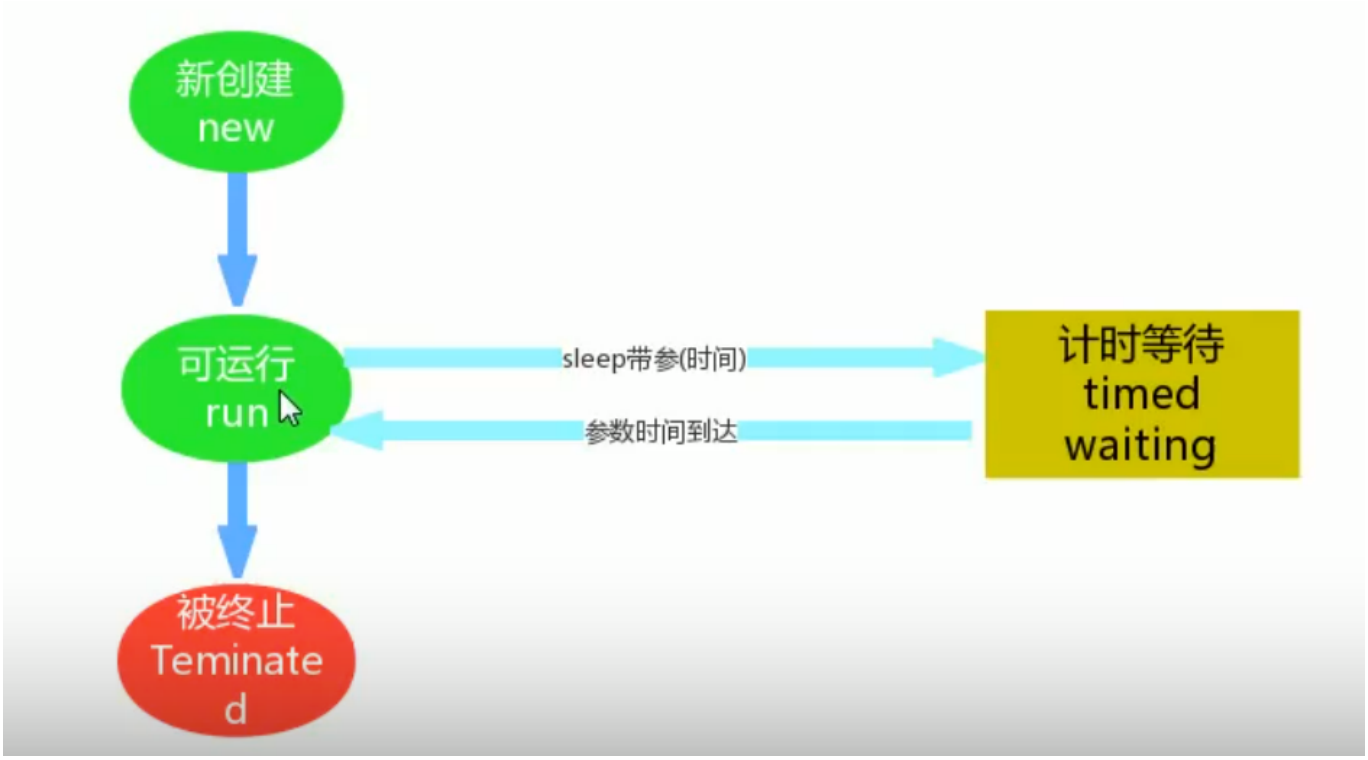
```
public class RunableImpl implements Runnable{
    private static int ticket = 100;
   Lock l=new ReentrantLock();//多态
    @Override
    public void run() {
        l.lock();
         while(ticket>0)
         {
             if(ticket>0)
             {
                 System.out.println(Thread.currentThread().getName()+"buy one
ticket -->"+ticket);
                 ticket--;
             }else
             {
                 System.out.println("fail to do so");
             }
         }
        l.unlock();
    }

}
```

5.线程状态

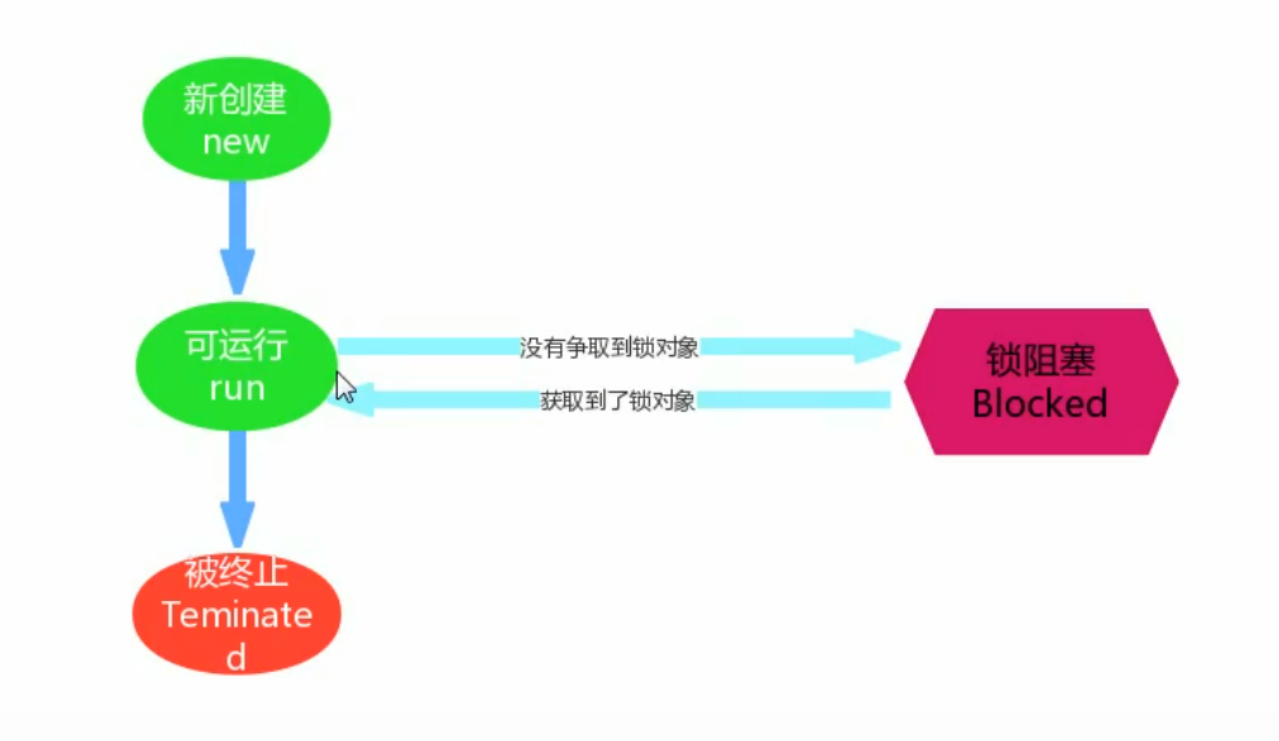
5.1线程状态概述



- Timed Waiting计时等待状态



- Blocked锁阻塞状态



- Waiting无限等待状态

5.2 等待唤醒案例-线程间的通信

- 等待对象和唤醒对象必须在同一个线程执行,运用同步方法
- 等待(wait)和唤醒(notify)只能由同一个锁对象调用
- 等待唤醒之后会居胥执行wait之后的代码

代码:

```
package Concurr;
public class WaitandNotify  {
   private static  Object object=new Object();//锁对象
    public static void main(String[] args) {
         new Thread(new Runnable() {
            @Override
            public void run() {
                while(true)
```

```
                {
                    synchronized (object)
                    {
                        System.out.println("I want have a meal~");
                        try {
                            object.wait();
                            System.out.println("it takes so delocious by
customer");
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {

                while(true)
                {
                    try{
                        Thread.sleep(5000);
                    }catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    synchronized (object)
                    {
                        System.out.println("it takes 5 minutes to make a meal" );
                        object.notify();
                    }
                }
            }
        }).start();
    }

}
```

- 带参数的`wait()`函数,和`notifyall()`
  - 进入`TimeWaiting`状态有两种方式:
    - 使用`sleep(long millis)`方法,在时间结束值后,线程睡醒进入到`Runnable/Blocked`状态
    - 使用`wait(long millis)`方法,在时间结束之后,还没有被`notify()`唤醒,就会自动醒来
  - 唤醒的方法:
    - `notify`:如果有多个等待线程,随机唤醒一个线程
    - `notifyall`唤醒所有线程;

6.线程池

- 线程池:容器–>集合（ArrayList,HashSet,LinkedList,HashMap）
  - 当程序第一次启动的时候，创建多个线程，保存到一个集合中

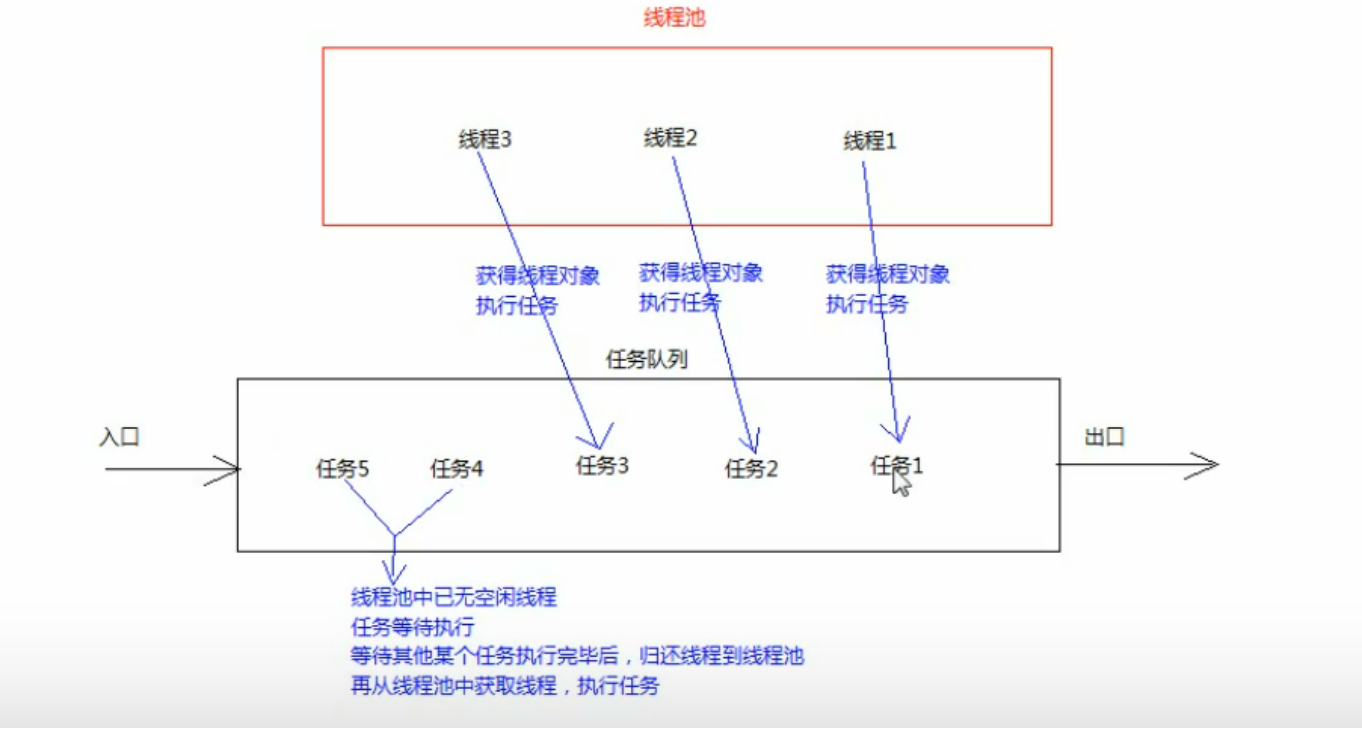
- 当我们需要使用线程的时候，可以从集合中取出线程来使用

  `Thread t=list.remove(0);`返回的是被移除的元素

  `Thread t=linkedlist.removeFirst();`当我们使用完线程，需要把线程归还给线程池

- `list.add(t)/linkedlist.addLast(t);`

> JDK1.5之后JDK内置了线程池，我们可以直接使用



线程池的好处（空间换时间）：

- 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可以执行多个任务
- 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

线程池的工厂类用来生成线程池：

```
import java.util.concurrent.ThreadPoolExecutor;
//Executors 类中的静态方法：
static ExecutorService newFixedThreadPool(int nThreads)
参数：
    int nThreads:创建线程池中包含的线程数量
返回值：
        ExecutorService接口，可以返回ExecutorService接口的实现类，用ExecutorService接口接收（面向接口编程）
```