当一个对象没有任何引用指向的时候，它就满足垃圾回收的条件  
当它被垃圾回收的时候，它的finalize() 方法就会被调用。  
finalize() 不是开发人员主动调用的方法，而是由虚拟机JVM调用的。

equals() 用于判断两个对象的内容是否相同

hashCode方法返回一个对象的哈希值

抽象类和接口的区别、

区别1：  
子类只能继承一个抽象类，不能继承多个  
子类可以实现**多个**接口  
区别2：  
抽象类可以定义  
public,protected,package,private  
静态和非静态属性  
final和非final属性  
但是接口中声明的属性，只能是  
public  
静态  
final的  
即便没有显式的声明 **注:**抽象类和接口都可以有实体方法。 接口中的实体方法，叫做[默认方法](https://how2j.cn/k/interface-inheritance/interface-inheritance-default-method/676.html)

与非静态内部类不同，**静态内部类**水晶类的实例化 **不需要一个外部类的实例为基础**，可以直接实例化  
语法：**new 外部类.静态内部类();**  
因为没有一个外部类的实例，所以在静态内部类里面**不可以访问外部类的实例属性和方法**  
除了可以访问外部类的**私有静态成员外**，静态内部类和普通类没什么大的区别

在匿名类中使用外部的局部变量，外部的局部变量必须修饰为final  
为什么要声明为final，其机制比较复杂，请参考第二个Hero代码中的解释  
**注：**在jdk8中，已经不需要强制修饰成final了，如果没有写final，不会报错，因为编译器**偷偷的**帮你加上了看不见的final

接口只能定义default实体方法

getTime() 得到一个long型的整数，这个整数代表 从**1970.1.1 08:00:00:000**开始 每经历一毫秒，增加1

当前日期的毫秒数：new Date().getTime() 和 System.currentTimeMillis() 是一样的。

那么最特殊的一个数字，就是零. 零这个数字，就代表Java中的时间原点，其对应的日期是1970年1月1日 8点0分0秒 。 (为什么是8点，因为中国的太平洋时区是UTC-8，刚好和格林威治时间差8个小时)

throws与throw这两个关键字接近，不过意义不一样，有如下区别：  
1. throws 出现在方法声明上，而throw通常都出现在方法体内。  
2. throws 表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某个异常对象。

可查异常： CheckedException  
可查异常即**必须进行处理的异常**，要么try catch住,要么往外抛，谁调用，谁处理

运行时异常RuntimeException指： **不是必须进行try catch的异常**  
**常见运行时异常:**  
除数不能为0异常:ArithmeticException  
下标越界异常:ArrayIndexOutOfBoundsException  
空指针异常:NullPointerException  
在编写代码的时候，依然可以使用try catch throws进行处理，与可查异常不同之处在于，**即便不进行try catch，也不会有编译错误**  
Java之所以会设计运行时异常的原因之一，是因为下标越界，空指针这些运行时异常**太过于普遍**，如果都需要进行捕捉，代码的可读性就会变得很糟糕。

异常分类：异常分为可查异常，运行时异常和错误3种  
其中，运行时异常和错误又叫非可查异常

错误Error，指的是**系统级别的异常**，通常是内存用光了  
在**默认设置下**，一般java程序启动的时候，最大可以使用16m的内存  
如例不停的给StringBuffer追加字符，很快就把内存使用光了。抛出**OutOfMemoryError**。与运行时异常一样，错误也是不要求强制捕捉的

**文件常用方法：renameTo方法用于对物理文件名称进行修改，但是并不会修改File对象的name属性。**

把流定义在try()里,try,catch或者finally结束的时候，会自动关闭  
这种编写代码的方式叫做 **try-with-resources**， 这是从JDK7开始支持的技术  
所有的流，都实现了一个接口叫做 **AutoCloseable**，任何类实现了这个接口，都可以在try()中进行实例化。 并且在try, catch, finally结束的时候自动关闭，回收相关资源。

字符保存在文件中肯定也是以数字形式保存的，即对应在不同的**棋盘**上的不同的数字  
用**记事本**打开任意文本文件，并且**另存为**，就能够在编码这里看到一个下拉：  
**ANSI**这个**不是ASCII**的意思，而是采用**本地编码**的意思。如果你是中文的操作系统，就会使GBK，如果是英文的就会是ISO-8859-1  
**Unicode** UNICODE原生的编码方式  
**Unicode big endian** 另一个 UNICODE编码方式  
**UTF-8** 最常见的UTF-8编码方式，数字和字母用一个字节， 汉字用3个字节。

FileReader得到的是字符，所以一定是已经把字节**根据某种编码识别成了字符**了  
而FileReader使用的编码方式是Charset.defaultCharset()的返回值，如果是中文的操作系统，就是GBK  
FileReader是不能手动设置编码方式的，为了使用其他的编码方式，只能使用InputStreamReader来代替，像这样：

new InputStreamReader(new FileInputStream(f),Charset.forName("UTF-8"));

以介质是硬盘为例，**字节流和字符流的弊端**：  
在每一次读写的时候，都会访问硬盘。 如果读写的频率比较高的时候，其性能表现不佳。  
为了解决以上弊端，采用缓存流。  
缓存流在读取的时候，**会一次性读较多的数据到缓存中**，以后每一次的读取，都是在缓存中访问，直到缓存中的数据读取完毕，再到硬盘中读取。

缓存流在写入数据的时候，会先把数据写入到缓存区，直到缓存区**达到一定的量**，才把这些数据，**一起写入到硬盘中去**。按照这种操作模式，就不会像字节流，字符流那样**每写一个字节都访问硬盘**，从而减少了IO操作

用数据流的writeUTF()和readUTF() 可以进行数据的**格式化顺序读写**  
如本例，通过DataOutputStream 向文件顺序写出 布尔值，整数和字符串。 然后再通过DataInputStream 顺序读入这些数据。  
**注：** 要用DataInputStream 读取一个文件，这个文件必须是由DataOutputStream 写出的，否则会出现EOFException，因为DataOutputStream 在写出的时候会做一些特殊标记，只有DataInputStream 才能成功的读取。

对象流指的是可以直接**把一个对象以流的形式**传输给其他的介质，比如硬盘  
一个对象以流的形式进行传输，叫做序列化。 该对象所对应的类，必须是实现Serializable接口

import java.io.Serializable;

public class Hero implements Serializable {

    //表示这个类当前的版本，如果有了变化，比如新设计了属性，就应该修改这个版本号

    private static final long serialVersionUID = 1L;

    public String name;

    public float hp;

}

System.out 是常用的在控制台输出数据的  
System.in 可以从控制台输入数据

**集合框架**

**add**有两种用法  
第一种是直接add对象，把对象加在最后面

heros.add(new Hero("hero " + i));

第二种是在指定位置加对象

heros.add(3, specialHero);

通过方法**contains**判断一个对象是否在容器中  
判断标准： 是否是同一个对象，而不是name是否相同

通过get获取指定位置的对象，如果输入的下标越界，一样会报错

**indexOf**用于判断一个对象在ArrayList中所处的位置  
与[contains](https://how2j.cn/k/collection/collection-arraylist-method/685.html#step2454)一样，判断标准是对象是否相同，而非对象的name值是否相等

**remove**用于把对象从ArrayList中删除  
remove可以根据下标删除ArrayList的元素

heros.remove(2);  
也可以根据对象删除

heros.remove(specialHero);

**set**用于替换指定位置的元素

**size** 用于获取ArrayList的大小

**toArray**可以把一个ArrayList对象转换为数组。  
需要注意的是，如果要转换为一个Hero数组，那么需要传递一个Hero数组类型的对象给toArray()，这样toArray方法才知道，你希望转换为哪种类型的数组，否则只能转换为Object数组

**addAll** 把另一个容器所有对象都加进来

不指定泛型的容器，可以存放任何类型的元素  
指定了泛型的容器，只能存放指定类型的元素以及其子类

除了实现了List接口外，LinkedList还实现了**双向链表结构**Deque，可以很方便的在头尾插入删除数据

对于HashMap而言，key是唯一的，不可以重复的。  
所以，以相同的key 把不同的value插入到 Map中会导致旧元素被覆盖，只留下最后插入的元素。  
不过，同一个对象可以作为值插入到map中，只要对应的key不一样

Set中的元素，没有顺序。严格的说，是没有按照元素的插入顺序排列  
HashSet的具体顺序，既不是按照插入顺序，也不是按照hashcode的顺序。

不保证Set的迭代顺序; 确切的说，在不同条件下，元素的顺序都有可能不一样

Collection是 Set List Queue和 Deque的接口  
Queue: 先进先出队列  
Deque: 双向链表  
**注：**Collection和Map之间没有关系，Collection是放一个一个对象的，Map 是放键值对的  
**注：**Deque 继承 Queue,间接的继承了 Collection

Collections是一个类，容器的工具类,就如同Arrays是数组的工具类，常用的方法有：1. **reverse** 使List中的数据顺序发生翻转

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List<Integer> num = **new** ArrayList<Integer>();  **for**(**int** i=1;i<11;i++) {  num.add(i);  }  System.***out***.println("集合中的数据：" + num);  //reverse翻转  Collections.*reverse*(num);  System.***out***.println("翻转后集合中的数据：" + num);  //shuffle洗牌  Collections.*shuffle*(num);  System.***out***.println("混淆后集合中的数据：" + num);  //sort排序  Collections.*sort*(num);  System.***out***.println("排序后集合中的数据：" + num);  //swap交换指定下标的值的位置  Collections.*swap*(num, 0, 9);  System.***out***.println("交换下标为0和9的元素之后集合中的数据：" + num);  //rotate向右滚动指定单位长度  Collections.*rotate*(num, 5);  System.***out***.println("向右滚动5个元素位置之后集合中的数据：" + num);  //把非线程安全的List转换为线程安全的List  List<Integer> synNum = (List<Integer>) Collections.*synchronizedList*(num);  } |

HashMap和Hashtable都实现了Map接口，都是键值对保存数据的方式  
区别1：  
HashMap可以存放 null  
Hashtable不能存放null  
区别2：  
HashMap不是[线程安全的类](https://how2j.cn/k/thread/thread-synchronized/355.html" \l "step793)  
Hashtable是线程安全的类

对于HashMap：keySet()可以获取所有的key, values()可以获取所有的value

分析HashMap查找性能卓越的原因：由于底层采用了哈希表（散列表）存储数据，所以HashMap在查找、删除、修改方面的效率都很高

**-----hashcode概念-----**  
所有的对象，都有一个对应的**hashcode（散列值）**

HashSet的数据是不能重复的，相同数据不能保存在一起，到底如何判断是否是重复的呢？  
 根据[HashSet和HashMap的关系](https://how2j.cn/k/collection/collection-hashset/364.html#step825)，我们了解到因为HashSet没有自身的实现， 而是里面封装了一个HashMap，所以本质上就是判断HashMap的key是否重复。

再通过上一步的学习，key是否重复，是由两个步骤判断的：  
hashcode是否一样  
如果hashcode不一样，就是在**不同的坑里(链表)**，一定是不重复的  
如果hashcode一样，就是在**同一个坑里**，还需要进行equals比较  
如果equals一样，则是重复数据  
如果equals不一样，则是不同数据。

假设Hero有三个属性 name,hp,damage  
一个集合中放存放10个Hero,通过Collections.sort对这10个进行排序  
那么**到底是hp小的放前面？还是damage小的放前面？**Collections.sort也无法确定  
所以要指定到底按照哪种属性进行排序  
这里就需要提供一个Comparator给定如何进行两个对象之间的**大小**比较

|  |
| --- |
| //直接调用sort会出现编译错误，因为Hero有各种属性  //到底按照哪种属性进行比较，Collections也不知道，不确定，所以没法排    //Collections.sort(heros);      //引入Comparator，指定比较的算法    Comparator<Hero> c = new Comparator<Hero>() {      @Override      public int compare(Hero h1, Hero h2) {      //按照hp进行排序      if(h1.hp>=h2.hp)       return 1;  //正数表示h1比h2要大                  else      return -1;     }  };    Collections.sort(heros,c);    System.out.println("按照血量排序后的集合：");    System.out.println(heros); |

**class** Hero2 **implements** Comparable<Hero2> {

……

@Override

**public** **int** compareTo(Hero2 hero) {

**if**(damage<hero.damage) {

**return** 1; //降序排序

}**else** {

**return** -1;

}

}

**泛型**

假设容器的泛型是Hero,那么**Hero的子类**APHero,ADHero**都可以放进去**  
和Hero无关的类型Item还是放不进去

ArrayList heroList<? extends Hero> 表示这是一个Hero泛型或者其子类泛型  
heroList 的泛型可能是Hero  
heroList 的泛型可能是APHero  
heroList 的泛型可能是ADHero  
所以 可以确凿的是，**从heroList取出来的对象，一定是可以转型成Hero的**  
但是，不能往里面放东西，因为  
放APHero就不满足<ADHero>  
放ADHero又不满足<APHero>

ArrayList heroList<? super Hero> 表示这是一个Hero泛型或者其父类泛型  
heroList的泛型可能是Hero  
heroList的泛型可能是Object  
**可以往里面插入Hero以及Hero的子类**  
但是取出来有风险，因为不确定取出来是Hero还是Object

泛型通配符? 代表任意泛型  
既然?代表任意泛型，那么换句话说，这个容器什么泛型都有可能  
所以只能以Object的形式取出来  
并且不能往里面放对象，因为不知道到底是一个什么泛型的容器

总结

如果希望只取出，不插入，就使用? extends Hero

如果希望只插入，不取出，就使用? super Hero

如果希望，又能插入，又能取出，就不要用？通配符

既然 子类对象 转 父类对象是可以成功的，那么子类泛型转父类泛型能成功吗？

假设可以转型成功  
引用hs指向了ADHero泛型的容器  
作为Hero泛型的引用hs, 看上去是可以往里面加一个APHero的。  
但是hs这个引用，实际上是指向的一个ADHero泛型的容器  
如果能加进去，就变成了ADHero泛型的容器里放进了APHero，这就矛盾了  
所以子类泛型**不可以**转换为父类泛型，且父类泛型**也不可以**转换为子类泛型

Lambda表达式：

|  |
| --- |
| **public** **class** LambdaTest {  **public** **static** **void** main(String[] args) {  Random r = **new** Random();  List<Hero> hero = **new** ArrayList<Hero>();  **for**(**int** i=0; i<10; i++) {  hero.add(**new** Hero("hero" + i, r.nextInt(100), r.nextInt(100)));  }  System.***out***.println("初始化后的集合：");  System.***out***.println(hero);    // System.out.println("使用匿名类的方式，筛选出hp>50 && damage<50的hero：");  // HeroChecker checker = new HeroChecker() {  // @Override  // public boolean test(Hero h) {  // return (h.hp>50 && h.damage<50);  // }  // };  // filter(hero, checker);    System.***out***.println("使用lambda表达式：");  *filter*(hero, (h)->h.hp>80 && h.damage<50);//h是指Hero对象，可以是任取的变量。  // Lambda表达式可以看成是匿名类一点点演变过来  // 把外面的壳子去掉  // 只保留方法参数和方法体  // 参数和方法体之间加上符号 ->  HeroChecker c2 = (Hero h) ->{  **return** h.hp>100 && h.damage<50;  };  // 把return和{}去掉  HeroChecker c3 = (Hero h) ->h.hp>100 && h.damage<50;  // 把 参数类型和圆括号去掉(只有一个参数的时候，才可以去掉圆括号)  HeroChecker c4 = h ->h.hp>100 && h.damage<50;  // 把c4作为参数传递进去  *filter*(hero, c4);  // 直接把表达式传递进去  *filter*(hero, h -> h.hp > 100 && h.damage < 50);  // 与匿名类 概念相比较，Lambda 其实就是匿名方法  // 这是一种把方法作为参数进行传递的编程思想。    // Lambda表达式虽然带来了代码的简洁，但是也有其局限性。  // 1. 可读性差，与啰嗦的但是清晰的匿名类代码结构比较起来，Lambda表达式一旦变得比较长，就难以理解  // 2. 不便于调试，很难在Lambda表达式中增加调试信息，比如日志  // 3. 版本支持，Lambda表达式在JDK8版本中才开始支持，如果系统使用的是以前的版本，考虑系统的稳定性等原因，而不愿意升级，那么就无法使用。  // Lambda比较适合用在简短的业务代码中，并不适合用在复杂的系统中，会加大维护成本。    }    **private** **static** **void** filter(List<Hero> hero, HeroChecker checker) {  **for**(Hero he: hero) {  **if**(checker.test(he)) {  System.***out***.println(he);  }  }  }  }  **public** **interface** HeroChecker {  **public** **boolean** test(Hero h);  } |

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;  import java.util.Random;  /\*  \* 匿名表达式引用静态方法和引用对象方法  \*/  public class LambdaTest1 {  public static void main(String[] args) {  Random r = new Random();  List<Hero> hero = new ArrayList<Hero>();  for(int i=0; i<10; i++) {  hero.add(new Hero("hero-"+i, r.nextInt(100), r.nextInt(100)));    }  System.out.println("初始化之后的集合：");  System.out.println(hero);  // HeroChecker c = new HeroChecker() {  // public boolean test(Hero h) {  // return h.hp>70 && h.damage>70;  // }  // };  //  // System.out.println("使用匿名类过滤：");  // filter (hero, c);    System.out.println("使用Lambda表达式过滤：");  filter(hero, h->h.hp>40 && h.damage>40);//与HeroChecker c无关    System.out.println("在Lambda表达式中使用静态方法过滤：");  filter(hero, h->LambdaTest1.testHero(h));    System.out.println("直接引用静态方法过滤：");  filter(hero, LambdaTest1::testHero);    //引用对象方法：与引用静态方法很类似，只是传递方法的时候，需要一个对象的存在  System.out.println("使用引用对象方法过滤：");  LambdaTest1 lt = new LambdaTest1();  filter(hero, lt::testHero1); //非static方法  }      public static boolean testHero(Hero h) {  return h.hp>50 && h.damage>50;  }    public boolean testHero1(Hero h) {  return h.hp>30 && h.damage>30;  }      private static void filter(List<Hero> hero, HeroChecker checker) {  for(Hero her:hero) {  if(checker.test(her)) {  System.out.println(her);  }  }    }    } |

|  |
| --- |
| /\*  \* 引用容器中的对象方法  \*/  **public** **class** LambdaTest2 {  **public** **static** **void** main(String[] args) {  Random r = **new** Random();  List<Hero> hero = **new** ArrayList<Hero>();  **for** (**int** i = 0; i < 10; i++) {  hero.add(**new** Hero("hero-" + i, r.nextInt(100), r.nextInt(100)));  }  System.***out***.println("初始化之后的集合：");  System.***out***.println(hero);  System.***out***.println("Lambda表达式过滤：");  *filter*(hero, hh->hh.hp>50&&hh.damage<50);    System.***out***.println("Lambda表达式中调用容器中对象的matched方法过滤：");  *filter*(hero, hm->hm.matched());    System.***out***.println("直接引用容器中对象的matched方法过滤：");  *filter*(hero, Hero::matched);    }    **public** **boolean** testHero(Hero h) {  **return** h.hp>70 && h.damage<70;  }    **private** **static** **void** filter(List<Hero> h, HeroChecker hc) {  **for** (Hero he : h) {  **if** (hc.test(he)) {  System.***out***.println(he);  }  }  }      } |

|  |
| --- |
| /\*  \* 引用构造器  \* 有的接口中的方法会返回一个对象，比如java.util.function.Supplier  \* 提供了一个get()方法，返回一个对象  \*/  **public** **class** LambdaTest3 {  **public** **static** **void** main(String[] args) {  Supplier<List> s = **new** Supplier<List>() {  @Override  **public** List get() {  **return** **new** ArrayList();  }    };  //为了调用getList方法，有三种方式：  //第一种：匿名类  List list1 = *getList*(s);    //第二种：Lambda表达式  List list2 = *getList*(()->**new** ArrayList());    //第三种：引用构造器  List list3 = *getList*(ArrayList::**new**);    }  **public** **static** List getList(Supplier<List> s) {  **return** s.get();  }  } |

Lambda 聚合操作

|  |
| --- |
| **public** **class** AggregateTest {  **public** **static** **void** main(String[] args) {  Random r = **new** Random();  List<Hero> hero = **new** ArrayList<Hero>();  **for** (**int** i = 0; i < 10; i++) {  hero.add(**new** Hero("hero-" + i, r.nextInt(100), r.nextInt(100)));  }  System.***out***.println("初始化之后的集合：");  System.***out***.println(hero);    System.***out***.println("查询条件：hp>50 && damage<50");  System.***out***.println("传统方式找出满足条件的数据：");  **for**(Hero h:hero) {  **if**(h.hp>50 && h.damage<50) {  System.***out***.println(h);  }  }    // 管道源：在这个例子里，源是一个List    // 中间操作： 每个中间操作，又会返回一个Stream，比如.filter()又返回一个Stream, 中间操作是“懒”操作，并不会真正进行遍历。    // 结束操作：当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。  // 结束操作不会返回Stream，但是会返回int、float、String、 Collection或者像forEach，什么都不返回, 结束操作才进行真正的遍历行为，  // 在遍历的时候，才会去进行中间操作的相关判断    System.***out***.println("聚合操作方式找出满足条件的数据：");  hero  .stream() //注意这里不用分号  .filter(hh-> hh.hp>80 && hh.damage<80)  .forEach(h-> System.***out***.println(h.name));    //管道源是集合  System.***out***.println("管道源是集合：");  hero  .stream()  .forEach(h-> System.***out***.println(h.name));    //管道源是数组  System.***out***.println("管道源是数组：");  Hero hs[] = hero.toArray(**new** Hero[hero.size()]);  Arrays.*stream*(hs) //或者Streams.of(hs)  .forEach(h-> System.***out***.println(h.name));    // 中间操作比较多，主要分两类  // 对元素进行筛选 和 转换为其他形式的流  // 对元素进行筛选：  // filter 匹配  // distinct 去除重复(根据equals判断)  // sorted 自然排序  // sorted(Comparator<T>) 指定排序  // limit 保留  // skip 忽略  // 转换为其他形式的流  // mapToDouble 转换为double的流  // map 转换为任意类型的流    hero.add(hero.get(0)); //制造一个重复数据  System.***out***.println("满足条件hp>80 && damage<80的数据");  hero  .stream()  .filter(h-> h.hp>80 && h.damage<80)  .forEach(h->System.***out***.println(h));    System.***out***.println("去除重复的数据，重复的判断标准是equals");  hero  .stream()  .distinct()  .forEach(h-> System.***out***.println(h));    System.***out***.println("让hero按照hp排序");  hero  .stream()  .sorted((h1,h2)-> h1.hp>=h2.hp ? 1 : -1)  .forEach(h-> System.***out***.println(h));    System.***out***.println("只保留前三个hero对象");//并不影响hero集合  hero  .stream()  .limit(3)  .forEach(h-> System.***out***.println(h));    System.***out***.println("忽略前三个hero对象");  hero  .stream()  .skip(3)  .forEach(h-> System.***out***.println(h));    System.***out***.println("hp转换为double的stream");  hero  .stream()  .mapToDouble(Hero::getHp)  .forEach(h->System.***out***.println(h));    System.***out***.println("转换为任意类型的stream");  hero  .stream()  .map((h)-> h.name + "\_" + h.hp + "\_" + h.damage)  .forEach(h-> System.***out***.println(h));    // 结束操作才真正进行遍历行为，前面的中间操作也在这个时候，才真正的执行。  // 常见结束操作如下：  // forEach() 遍历每个元素  // toArray() 转换为数组  // min(Comparator<T>) 取最小的元素  // max(Comparator<T>) 取最大的元素  // count() 总数  // findFirst() 第一个元素  System.***out***.println("遍历集合中的每个元素");  hero  .stream()  .forEach(h->System.***out***.println(h));    System.***out***.println("返回一个数组");  Object[] hs1 = hero  .stream()  .toArray();  System.***out***.println(Arrays.*toString*(hs));    System.***out***.println("返回damage最小的那个英雄");  Hero minDamageHero =  hero  .stream()  .min((h1,h2)-> h1.damage - h2.damage)  .get();  System.***out***.println(minDamageHero);    System.***out***.println("返回damage最高的那个英雄");  Hero maxDamageHero =  hero  .stream()  .max((h1,h2)-> h1.damage - h2.damage)  .get();  System.***out***.println(maxDamageHero);    System.***out***.println("计算流中数据的总数");  **long** count = hero  .stream()  .count();  System.***out***.println(count);    System.***out***.println("返回第一个元素");  Hero firstHero =  hero  .stream()  .findFirst()  .get();  System.***out***.println(firstHero);    }  } |

任何一个类，都会有一个Class对象于这个类对应，在这个Class对象中，保存着实例化该类时所需要的基本信息，A.class 其实返回的是一个类A的Class对象

贴一个小代码演示一下：

public class Test {

public static void main(String[] args) {

// 得到类名（不包含包名）

System.out.println( String.class.getSimpleName() ); // String

// 得到类的全名称（包含所在的包名）

System.out.println( String.class.getName() ); // java.lang.String

System.out.println( Test.class.getName() ); // Test

}

}

获取Calss对象的方式主要有三种：

第一种：通过Class.forName(“类的全名称”)获取，用这个方法，最常见的应该是应用于JDBC注册驱动的时候用到的

try {

Class.forName("oracle.jdbc.driver.OracleDriver");// 注册驱动

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

第二种：通过已经实例化的对象获取，getClass()方法获取

第三种：通过类名.class获取

public class Test {

public static void main(String[] args) {

// TODO Auto-generated method stub

Test test = new Test();

System.out.println(test.getClass());// class Test

System.out.println(Test.class);// class Test

}

}

**多线程**

创建多线程有3种方式，分别是[继承线程类](https://how2j.cn/k/thread/thread-start/353.html" \l "step778),[实现Runnable接口](https://how2j.cn/k/thread/thread-start/353.html#step779),[匿名类](https://how2j.cn/k/thread/thread-start/353.html#step780)

首先要理解进程(Processor)和线程(Thread)的区别  
**进程：**启动一个LOL.exe就叫一个进程。 接着又启动一个DOTA.exe，这叫两个进程。  
**线程：**线程是在进程内部同时做的事情，比如在LOL里，有很多事情要同时做，比如"盖伦” 击杀“提莫”，**同时**“赏金猎人”又在击杀“盲僧”，这就是由多线程来实现的。

|  |
| --- |
| /\*  \* 此处代码演示的是不使用多线程的情况：  \* 只有在盖伦杀掉提莫后，赏金猎人才开始杀盲僧  \*/  **public** **class** ThreadTest {  **public** **static** **void** main(String[] args) {  Hero gareen = **new** Hero("盖伦", 1260, 130);    Hero teemo = **new** Hero("提莫", 1000, 150);    Hero bh = **new** Hero("赏金猎人", 600, 200);    Hero leesin = **new** Hero("盲僧", 800, 180);    **while**(!teemo.isDead()) {  gareen.attackHero(teemo);  }    **while**(!leesin.isDead()) {  bh.attackHero(leesin);  }  }  }  **class** Hero{  **public** String name;  **public** **float** hp;  **public** **int** damage;    **public** Hero(String name, **float** hp, **int** damage) {  **this**.name = name;  **this**.hp = hp;  **this**.damage = damage;  }    **public** **void** attackHero(Hero h) {  **try** {  //每次攻击间隔1000毫秒  Thread.*sleep*(1000);  }**catch**(InterruptedException e) {  e.printStackTrace();  }  h.hp -= damage;    **if**(h.isDead()) {  System.***out***.println(h.name + "被击杀！");  }**else** {  String str = String.*format*("%s正在攻击%s，%s的血量变成了%.0f%n", **this**.name, h.name, h.name, h.hp);  System.***out***.print(str);  }    }      **public** **boolean** isDead() {  **return** 0>=hp ? **true** : **false**;  }  } |

创建多线程

|  |
| --- |
| /\* 创建多线程方法二：继承线程类Thread  \* 使用多线程，就可以做到盖伦在攻击提莫的同时，赏金猎人也在攻击盲僧  \* 设计一个类KillThread 继承Thread，并且重写run方法  \* 启动线程办法： 实例化一个KillThread对象，并且调用其start方法  \* 就可以观察到 赏金猎人攻击盲僧的同时，盖伦也在攻击提莫  \*/  **public** **class** MultiThread1 {  **public** **static** **void** main(String[] args) {  Hero gareen = **new** Hero("盖伦", 1260, 130);    Hero teemo = **new** Hero("提莫", 1000, 150);    Hero bh = **new** Hero("赏金猎人", 600, 200);    Hero leesin = **new** Hero("盲僧", 800, 180);    MeanwhileThread mt1 = **new** MeanwhileThread(gareen, teemo);  mt1.start();    MeanwhileThread mt2 = **new** MeanwhileThread(bh, leesin);  mt2.start();  }  }  **class** MeanwhileThread **extends** Thread{  **private** Hero h1;  **private** Hero h2;    **public** MeanwhileThread(Hero h1, Hero h2) {  **this**.h1 = h1;  **this**.h2 = h2;  }    //重写Thread的run方法  // @Override  **public** **void** run() {  **while**(!h2.isDead()) {  h1.attackHero(h2);  }  }  } |

|  |
| --- |
| /\* 创建多线程方法二：实现Runnable接口  \* 创建类Battle，实现Runnable接口  \* 启动的时候，首先创建一个Battle对象，然后再根据该battle对象创建一个线程对象，并启动  \* Battle battle1 = new Battle(gareen,teemo);  \* new Thread(battle1).start();  \* battle1 对象实现了Runnable接口，所以有run方法，但是直接调用run方法，并不会启动一个新的线程。  \* 必须借助一个线程对象的start()方法，才会启动一个新的线程。  \* 所以，在创建Thread对象的时候，把battle1作为构造方法的参数传递进去，这个线程启动的时候，就会去执行battle1.run()方法了。  \*/  **public** **class** MultiThread2 {  **public** **static** **void** main(String[] args) {  Hero gareen = **new** Hero("盖伦", 1260, 130);    Hero teemo = **new** Hero("提莫", 1000, 150);    Hero bh = **new** Hero("赏金猎人", 600, 200);    Hero leesin = **new** Hero("盲僧", 800, 180);    Battle bat1 = **new** Battle(gareen, teemo);  **new** Thread(bat1).start();    Battle bat2 = **new** Battle(bh, leesin);  **new** Thread(bat2).start();  }  }  **class** Battle **implements** Runnable{  **private** Hero h1;  **private** Hero h2;    **public** Battle(Hero h1, Hero h2) {  **this**.h1 = h1;  **this**.h2 = h2;  }    **public** **void** run() {  **while**(!h2.isDead()) {  h1.attackHero(h2);  }  }  } |

|  |
| --- |
| /\* 创建多线程方法三：匿名类  \* 使用匿名类，继承Thread,重写run方法，直接在run方法中写业务代码  \* 匿名类的一个好处是可以很方便的访问外部的局部变量。  \* 前提是外部的局部变量需要被声明为final。(JDK7以后就不需要了)  \*/  **public** **class** MultiThread3 {  **public** **static** **void** main(String[] args) {  Hero gareen = **new** Hero("盖伦", 1260, 130);    Hero teemo = **new** Hero("提莫", 1000, 150);    Hero bh = **new** Hero("赏金猎人", 600, 200);    Hero leesin = **new** Hero("盲僧", 800, 180);    //匿名类  Thread th1 =**new** Thread() {  **public** **void** run() {  //匿名类中用到外部的局部变量teemo，必须把teemo声明为final  //但是再JDK7以后，就不是必须加fianl了。  **while**(!teemo.isDead()) {  gareen.attackHero(teemo);  }  }  };  th1.start();    Thread th2 = **new** Thread() {  **public** **void** run() {  **while**(!leesin.isDead()) {  bh.attackHero(leesin);  }  }  };  th2.start();    }  }  //把上述3种方式再整理一下：  //1. 继承Thread类  //2. 实现Runnable接口  //3. 匿名类的方式  //注： 启动线程是start()方法，run()并不能启动一个新的线程 |

window是大小写不敏感的系统，linux是大小写敏感的系统

其实Join方法实现是通过 wait（小提示：Object 提供的方法）进行线程控制的。 当main线程调用t.join时候，main线程会获得线程对象t的锁（wait 意味着拿到该对象的锁),调用该对象的wait(等待时间)，直到该对象唤醒main线程，比如退出后， 线程对象t退出后，会在native方法中调用线程对象的natifyAll方法，然后执行main线程的后续部分代码。

**常见线程方法**

| **关键字** | **简介** | **示例代码** |
| --- | --- | --- |
| sleep | 当前线程暂停 | [示例代码](https://how2j.cn/k/thread/thread-methods/354.html#step781) |
| join | 加入到当前线程中 | [示例代码](https://how2j.cn/k/thread/thread-methods/354.html#step782) |
| setPriority | 线程优先级 | [示例代码](https://how2j.cn/k/thread/thread-methods/354.html#step783) |
| yield | 临时暂停 | [示例代码](https://how2j.cn/k/thread/thread-methods/354.html#step784) |
| setDaemon | 守护线程 | [示例代码](https://how2j.cn/k/thread/thread-methods/354.html#step2403) |
|  |  |  |

**线程安全**

同步问题

如果一个类，其**方法都是有synchronized修饰的**，那么该类就叫做**线程安全的类**  
同一时间，只有一个线程能够进入 **这种类的一个实例** 的去修改数据，进而保证了这个实例中的数据的安全(不会同时被多线程修改而变成脏数据)  
比如StringBuffer和StringBuilder的区别  
StringBuffer的方法都是有synchronized修饰的，StringBuffer就叫做线程安全的类  
而StringBuilder就不是线程安全的类

|  |
| --- |
| package Thread;  import java.text.SimpleDateFormat;  import java.util.Date;  /\*  \* synchronized表示当前线程，独占 对象 someObject  \* 当前线程独占 了对象someObject，如果有其他线程试图占  \* 有对象someObject，就会等待，直到当前线程释放对someObject的占用。  \* someObject 又叫同步对象，所有的对象，都可以作为同步对象  \* 为了达到同步的效果，必须使用同一个同步对象  \*/  public class SynchronizedTest {  public static void main(String[] args) {    final Object someObject = new Object();    Thread t1 = new Thread() {  public void run() {  try {  System.out.println(now() + " t1线程开始运行");  System.out.println(now() + this.getName() + " 试图占有someObject对象");    synchronized(someObject) {  System.out.println(now() + this.getName() + " 占有someObject对象");  Thread.sleep(4000);  System.out.println(now() + this.getName() + " 释放someObject对象");  }  System.out.println(now() + " t1线程结束");  }catch(InterruptedException e) {  e.printStackTrace();  }  }  };  t1.setName(" t1");  t1.start();    Thread t2 = new Thread() {  public void run() {  try {  System.out.println(now() + " t2线程开始运行");  System.out.println(now() + this.getName() + " 试图占有someObject对象");    synchronized(someObject) {  System.out.println(now() + this.getName() + " 占有someObject对象");  Thread.sleep(5000);  System.out.println(now() + this.getName() + " 释放someObject对象");  }  System.out.println(now() + " t2线程结束");  }catch(InterruptedException e) {  e.printStackTrace();  }  }  };  t2.setName(" t2");  t2.start();    }    public static String now() {  return new SimpleDateFormat("HH:mm:ss").format(new Date());  }    } |

|  |
| --- |
| **package** Thread;  **public** **class** SynchronizedTest1 {  **public** **static** **void** main(String[] args) {  **final** Object someObject = **new** Object();  Hero gareen = **new** Hero("盖伦", 1000, 100);    **int** n=1000;  Thread[] addThread = **new** Thread[n];  Thread[] reduceThread = **new** Thread[n];    **for**(**int** i = 0; i < n; i++) {  Thread t1 = **new** Thread() {  **public** **void** run() {  **synchronized**(someObject) {  System.***out***.println("t1线程add, gareen.hp++");  gareen.recover();  }  **try** {  Thread.*sleep*(100);  }**catch**(InterruptedException e) {  e.printStackTrace();  }  }  };  t1.start();  addThread[i] = t1;  }    **for**(**int** i = 0; i < n; i++) {  Thread t2 = **new** Thread() {  **public** **void** run() {  **synchronized**(someObject) {  System.***out***.println("t2线程hurt, gareen.hp--");  gareen.hurt();  }  **try** {  Thread.*sleep*(100);  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }    };  t2.start();  reduceThread[i] = t2;  }  **for**(Thread t: addThread) {  **try** {  t.join();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  **for**(Thread t : reduceThread) {  **try** {  t.join();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.printf("%d 个增加hp线程和 %d 个减少hp线程结束后%n 盖伦的hp变成 %.0f ", n, n, gareen.hp);    }  } |

|  |
| --- |
| **package** Thread;  **public** **class** synchronizedTest2 {  **public** **static** **void** main(String[] args) {  **final** Hero teemo = **new** Hero("提莫", 800, 150);    **int** n = 1000;  Thread[] addThread = **new** Thread[n];  Thread[] reduceThread = **new** Thread[n];  **for**(**int** i = 0; i < n; i++) {  Thread t1 = **new** Thread() {  **public** **void** run() {  **synchronized**(teemo) {  System.***out***.println("t1线程addHp, teemo.hp++");  teemo.recover();  }  **try** {  Thread.*sleep*(100);  } **catch**(InterruptedException e) {  e.printStackTrace();  }    }  };  t1.start();  addThread[i] = t1;  }    **for**(**int** i = 0; i < n; i++) {  Thread t2 = **new** Thread() {  **public** **void** run() {  //使用teemo作为synchronized  //在方法hurt中有synchronized(this)  System.***out***.println("t2线程hurtHp, teemo.hp--");  teemo.hurt();  **try** {  Thread.*sleep*(100);  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  };  t2.start();  reduceThread[i] = t2;  }    **for**(Thread t1 : addThread) {  **try** {  t1.join();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  **for**(Thread t2 : reduceThread) {  **try** {  t2.join();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.printf("%d 个增加hp线程和%d 个减少hp线程结束后%n 提莫的血量是 %.0f%n", n, n, teemo.hp);  }  } |

|  |
| --- |
| **package** Thread;  **public** **class** SynchronozedTest3 {  **public** **static** **void** main(String[] args) {  **final** Hero bh = **new** Hero("赏金猎人", 600, 180);  **int** n = 1000;    Thread[] addT = **new** Thread[n];  Thread[] reduceT = **new** Thread[n];  **for**(**int** i=0; i<n; i++) {  Thread t1 = **new** Thread() {  **public** **void** run() {  //recover自带synchronized  bh.recover();  **try** {  Thread.*sleep*(100);  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  };  t1.start();  addT[i] = t1;  }  **for**(**int** i=0; i<n; i++) {  Thread t2 = **new** Thread() {  **public** **void** run() {  //hurt自带synchronized  bh.hurt();  **try** {  Thread.*sleep*(100);  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  };  t2.start();  reduceT[i] = t2;  }    **for**(Thread t1 : addT) {  **try** {  t1.join();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  **for**(Thread t2 : reduceT) {  **try** {  t2.join();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }    System.***out***.printf("%d 个增加hp线程和%d 个减少hp线程结束后%n 提莫的血量是 %.0f%n", n, n, bh.hp);  }  } |

|  |
| --- |
| **package** Thread;  /\*  \* 此处代码演示的是不使用多线程的情况：  \* 只有在盖伦杀掉提莫后，赏金猎人才开始杀盲僧  \*/  **public** **class** ThreadTest {  **public** **static** **void** main(String[] args) {  Hero gareen = **new** Hero("盖伦", 1260, 130);    Hero teemo = **new** Hero("提莫", 1000, 150);    Hero bh = **new** Hero("赏金猎人", 600, 200);    Hero leesin = **new** Hero("盲僧", 800, 180);    **while**(!teemo.isDead()) {  gareen.attackHero(teemo);  }    **while**(!leesin.isDead()) {  bh.attackHero(leesin);  }  }  }  **class** Hero{  **public** String name;  **public** **float** hp;  **public** **int** damage;    **public** Hero(String name, **float** hp, **int** damage) {  **this**.name = name;  **this**.hp = hp;  **this**.damage = damage;  }    **public** **void** attackHero(Hero h) {  **try** {  //每次攻击间隔1000毫秒  // System.out.println("begin sleep");  Thread.*sleep*(1000);  // System.out.println("end sleep");  }**catch**(InterruptedException e) {  e.printStackTrace();  }  h.hp -= damage;    **if**(h.isDead()) {  System.***out***.println(h.name + "被击杀！");  }**else** {  String str = String.*format*("%s正在攻击%s，%s的血量变成了%.0f%n", **this**.name, h.name, h.name, h.hp);  System.***out***.print(str);  }    }      **public** **boolean** isDead() {  **return** 0>=hp ? **true** : **false**;  }  // 在recover前，直接加上synchronized ，其所对应的同步对象，就是this  // 和hurt方法达到的效果是一样  // 外部线程访问gareen的方法，就不需要额外使用synchronized 了  **public** **synchronized** **void** recover() {  **this**.hp = hp + 1;  }    **public** **void** hurt() {  // 表示当前Hero对象为同步对象  **synchronized**(**this**) {  **this**.hp = hp - 1;  }  }  } |

[在对象方法前，加上修饰符synchronized](https://how2j.cn/k/thread/thread-synchronized/355.html#step792) ，同步对象是当前实例。  
那么如果在类方法前，加上修饰符 synchronized，同步对象是什么呢？

线程安全的类

StringBuffer 是线程安全的  
StringBuilder 是非线程安全的  
所以当进行大量字符串拼接操作的时候，如果是单线程就用StringBuilder会更快些，如果是多线程，就需要用StringBuffer 保证数据的安全性  
**非线程安全的**为什么会比**线程安全的** 快？ 因为不需要同步嘛，省略了些时间

HashMap和Hashtable都实现了Map接口，都是键值对保存数据的方式  
区别1：  
HashMap可以存放 null  
Hashtable不能存放null  
区别2：  
HashMap不是[线程安全的类](https://how2j.cn/k/thread/thread-synchronized/355.html" \l "step793)  
Hashtable是线程安全的类

Vector是线程安全的类，而ArrayList是非线程安全的。

ArrayList是非线程安全的，换句话说，多个线程可以同时进入**一个ArrayList对象**的add方法  
借助Collections.synchronizedList，可以把ArrayList转换为线程安全的List。  
与此类似的，还有HashSet,LinkedList,HashMap等等非线程安全的类，都通过[工具类Collections](https://how2j.cn/k/collection/collection-collections/369.html)转换为线程安全的

|  |
| --- |
| **package** Thread;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;  **public** **class** ThreadSecurity {  **public** **static** **void** main(String[] args) {  List<Integer> list1 = **new** ArrayList<>();  List<Integer> list2 = Collections.*synchronizedList*(list1);    }    } |

死锁演示：

1. 线程1 首先占有对象1，接着试图占有对象2  
2. 线程2 首先占有对象2，接着试图占有对象1  
3. 线程1 等待线程2释放对象2  
4. 与此同时，线程2等待线程1释放对象1  
就会。。。一直等待下去

|  |
| --- |
| **package** Thread;  **public** **class** SisuoTest {  **public** **static** **void** main(String[] args) {  **final** Hero ahri = **new** Hero("九尾妖狐", 1200, 130);  **final** Hero annie = **new** Hero("安妮", 1400, 120);    Thread t1 = **new** Thread() {  **public** **void** run() {  **synchronized** (ahri) {  System.***out***.println("t1占有了九尾妖狐ahri");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("t1试图占有安妮annie");  System.***out***.println("t1线程等待中");  **synchronized** (annie) {  System.***out***.println("t1线程获得annie");  }  }  }  };  t1.start();    Thread t2 = **new** Thread() {  **public** **void** run() {  **synchronized**(annie) {  System.***out***.println("t2占有了安妮annie");  **try** {  Thread.*sleep*(2000);  } **catch**(InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("t2试图占有九尾妖狐ahri");  System.***out***.println("t2线程等待中");  **synchronized** (ahri) {  System.***out***.println("t2线程获得ahri");  }  }  }  };  t2.start();  }  } |

死锁解决方案：

线程之间有**交互通知**的需求，考虑如下情况：  
有两个线程，处理同一个英雄。  
一个加血，一个减血。  
减血的线程，发现血量=1，就停止减血，直到加血的线程为英雄加了血，才可以继续减血

不好的解决方式

故意设计减血线程频率更高，盖伦的血量迟早会到达1  
减血线程中**使用while循环判断是否是1**，如果是1就不停的循环,直到加血线程回复了血量  
这是不好的解决方式，因为会大量占用CPU,拖慢性能

使用wait和notify进行线程交互

在Hero类中：hurt()减血方法：当hp=1的时候，执行this.wait().  
this.wait()**表示 让占有this的线程等待，并临时释放占有**  
进入hurt方法的线程必然是减血线程，this.wait()会让减血线程临时释放对this的占有。 **这样加血线程，就有机会进入recover()加血方法了**。  
recover() 加血方法：增加了血量，执行this.notify();  
this.notify() 表示通知那些**等待在this的线程**，可以苏醒过来了。 等待在this的线程，恰恰就是减血线程。 一旦recover()结束， 加血线程释放了this，减血线程，就可以重新占有this，并执行后面的减血工作。

|  |
| --- |
| **package** Thread;  **public** **class** DealSisuo {  **public** **static** **void** main(String[] args) {  **final** Hero2 gareen = **new** Hero2("盖伦", 80, 10);  Thread t1 = **new** Thread() {  **public** **void** run() {  **while**(**true**) {  gareen.hurt();  **try** {  Thread.*sleep*(100);    } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  }  };  t1.start();    Thread t2 = **new** Thread() {  **public** **void** run() {  **while**(**true**) {  gareen.recover();  **try** {  Thread.*sleep*(500);    } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  }  };  t2.start();  }  }  **class** Hero2{  **public** String name;  **public** **float** hp;  **public** **int** damage;  **public** Hero2(String name, **float** hp, **int** damage) {  **this**.name = name;  **this**.hp = hp;  **this**.damage = damage;  }  **public** **synchronized** **void** recover() {  hp = hp + 1;  System.***out***.printf("%s增加1点血量，%s的血量变为%.0f%n", **this**.name, **this**.name, **this**.hp);  **this**.notify();  }    **public** **synchronized** **void** hurt() {  **if** (hp == 1) {  **try** {  **this**.wait();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }    hp = hp - 1;  System.***out***.printf("%s 减少了1点血量，%s的血量变为%.0f%n", **this**.name, **this**.name, **this**.hp);    }  } |

这里需要强调的是，wait方法和notify方法，并**不是Thread线程上的方法**，它们是Object上的方法。  
  
因为所有的Object都可以被用来作为同步对象，所以准确的讲，wait和notify是同步对象上的方法。  
  
wait()的意思是： 让占用了这个同步对象的**线程**，临时释放当前的占用，并且等待。 所以调用wait是有前提条件的，一定是在synchronized块里，否则就会出错。  
  
notify() 的意思是，通知**一个**等待在这个同步对象上的线程，**你**可以苏醒过来了，有机会重新占用当前对象了。  
  
notifyAll() 的意思是，通知**所有的**等待在这个同步对象上的线程，**你们**可以苏醒过来了，有机会重新占用当前对象了。

线程池：

每一个线程的启动和结束都是比较消耗时间和占用资源的。  
如果在系统中用到了很多的线程，大量的启动和结束动作会导致系统的性能变卡，响应变慢。  
为了解决这个问题，引入线程池，这种设计思想。  
线程池的模式很像[生产者消费者模式](https://how2j.cn/k/thread/thread-wait-notify/358.html" \l "step2591)，消费的对象是一个一个的能够运行的**任务**

线程池的思路和[生产者消费者模型](https://how2j.cn/k/thread/thread-wait-notify/358.html" \l "step2591)是很接近的。  
1. 准备一个任务容器  
2. 一次性启动10个 消费者线程  
3. 刚开始任务容器是空的，所以线程都**wait**在上面。  
4. 直到一个外部线程往这个任务容器中扔了一个“任务”，就会有一个消费者线程被[唤醒notify](https://how2j.cn/k/thread/thread-wait-notify/358.html" \l "step796)  
5. 这个消费者线程取出“任务”，并且**执行这个任务**，执行完毕后，继续等待下一次任务的到来。  
6. 如果短时间内，有较多的任务加入，那么就会有多个线程被**唤醒**，去执行这些任务。  
在整个过程中，都不需要创建新的线程，而是**循环使用这些已经存在的线程**

|  |
| --- |
| **package** Thread;  **import** java.util.LinkedList;  **public** **class** ThreadPool {  //线程池大小  **int** threadPoolSize;  //任务容器  LinkedList<Runnable> tasks = **new** LinkedList<Runnable>();    //试图执行任务的线程  **public** ThreadPool() {  threadPoolSize = 10;  //启动10个执行任务的线程  **synchronized**(tasks) {  **for**(**int** i=0; i<threadPoolSize; i++) {  **new** TaskConsumeThread("执行任务的线程-" + i).start();  }  }  }    **class** TaskConsumeThread **extends** Thread{  **public** TaskConsumeThread(String name) {  **super**(name);  }    Runnable task;    **public** **void** run() {  System.***out***.println("启动线程-" + **this**.getName());  **while**(**true**) {  **synchronized**(tasks) {  **while**(tasks.isEmpty()) {  **try** {  tasks.wait();  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  task = tasks.removeLast();  //允许添加任务的线程可以继续添加任务  tasks.notifyAll();  }  System.***out***.println(**this**.getName() + " 捕获任务并执行");  task.run();  }  }  }  **public** **void** add(Runnable r) {  **synchronized** (tasks) {  // 任务加入到任务容器  tasks.add(r);  // 唤醒等待的任务执行者线程  tasks.notifyAll();  }  }  } |

|  |
| --- |
| **package** Thread;  **public** **class** ThreadPoolTest {  **public** **static** **void** main(String[] args) {  ThreadPool pool = **new** ThreadPool();    **for**(**int** i = 0; i < 5; i++) {  **final** **int** num = i;  Runnable task = **new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println("任务" + num + "执行中。。。");  }  };    pool.add(task);  **try** {  Thread.*sleep*(1000);  } **catch**(InterruptedException e) {  e.printStackTrace();  }  }  }  } |

注意：对于普通的字符串 + 操作，并不需要我们将其改写为 StringBuilder ，因为Java编译器在

编译时就自动把多个连续的 + 操作编码为 StringConcatFactory 的操作。在运行

期， StringConcatFactory 会自动把字符串连接操作优化为数组复制或者 StringBuilder 操作。

你可能还听说过 StringBuffer ，这是Java早期的一个 StringBuilder 的线程安全版本，它通过

同步来保证多个线程操作 StringBuffer 也是安全的，但是同步会带来执行速度的下降。

StringBuilder 和 StringBuffer 接口完全相同，现在完全没有必要使用 StringBuffer 。

java提供自带的线程池，而不需要自己去开发一个自定义线程池了。  
线程池类**ThreadPoolExecutor**在包**java.util.concurrent**下

ThreadPoolExecutor threadPool= new ThreadPoolExecutor(10, 15, 60, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());  
**第一个**参数10 表示这个线程池**初始化了10个**线程在里面工作  
**第二个**参数15 表示如果10个线程不够用了，就会自动增加到**最多15个线程**  
**第三个**参数60 结合第四个参数TimeUnit.SECONDS，表示经过**60秒**，多出来的线程还没有接到活儿，就会回收，最后保持池子里就10个  
**第四个**参数TimeUnit.SECONDS 如上  
**第五个**参数 new LinkedBlockingQueue() 用来放任务的集合  
**execute**方法用于添加新的任务

|  |
| --- |
| package Thread;  import java.util.concurrent.LinkedBlockingQueue;  import java.util.concurrent.ThreadPoolExecutor;  import java.util.concurrent.TimeUnit;  /\*  \* java提供自带的线程池，而不需要自己去开发一个自定义线程池了。  \* 线程池类ThreadPoolExecutor在包java.util.concurrent下  \* ThreadPoolExecutor threadPool= new ThreadPoolExecutor(10, 15, 60, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());  \* 第一个参数10 表示这个线程池初始化了10个线程在里面工作  \* 第二个参数15 表示如果10个线程不够用了，就会自动增加到最多15个线程  \* 第三个参数60 结合第四个参数TimeUnit.SECONDS，表示经过60秒，多出来的线程还没有接到活儿，就会回收，最后保持池子里就10个  \* 第四个参数TimeUnit.SECONDS 时间单位/秒  \* 第五个参数 new LinkedBlockingQueue() 用来放任务的集合  \* execute方法用于添加新的任务  \*/  public class ThreadPoolExecutorTest {  public static void main(String[] args) {  ThreadPoolExecutor threadPool = new ThreadPoolExecutor(10, 15, 60, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());  int sleepTime = 1000;  int i = 1;  while (true) {  threadPool.execute(new Runnable() {  @Override  public void run() {  System.out.printf("执行任务");  try {  Thread.sleep(1000);  } catch(InterruptedException e) {  e.printStackTrace();  }  }  });  System.out.println(i++);  try {  Thread.sleep(sleepTime);  sleepTime = sleepTime>100?sleepTime-50:100;  } catch(InterruptedException e) {  e.printStackTrace();  }    }  }  } |

使用Lock对象实现同步效果

Lock是一个接口，为了使用一个Lock对象，需要用到

Lock lock = new ReentrantLock();  
与 **synchronized (someObject)**类似的，**lock()**方法，表示当前线程占用lock对象，一旦占用，其他线程就不能占用了。  
与 **synchronized**不同的是，一旦synchronized 块结束，就会自动释放对**someObject**的占用。 lock却必须调用**unlock**方法进行手动释放，为了保证释放的执行，往往会把unlock() 放在finally中进行。

|  |
| --- |
| **package** Thread;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  /\*  \* Lock是一个接口，为了使用一个Lock对象，需要用到  \* Lock lock = new ReentrantLock();  \* 与 synchronized (someObject) 类似的，lock()方法，  \* 表示当前线程占用lock对象，一旦占用，其他线程就不能占用了。  \* 与 synchronized 不同的是，一旦synchronized 块结束，  \* 就会自动释放对someObject的占用。 lock却必须调用unlock方法进行手动释放，  \* 为了保证释放的执行，往往会把unlock() 放在finally中进行。  \*/  **public** **class** LockTest {  **public** **static** String now() {  **return** **new** SimpleDateFormat("HH:mm:ss").format(**new** Date());  }    **public** **static** **void** log(String msg) {  System.***out***.printf("%s %s %s %n", *now*(), Thread.*currentThread*().getName(), msg);  }    **public** **static** **void** main(String[] args) {  Lock loc = **new** ReentrantLock();    Thread t1 = **new** Thread() {  **public** **void** run() {  **try** {  *log*("线程启动");  *log*("试图占有loc对象");    loc.lock();    *log*("占有对象：loc");  *log*("进行5秒的业务操作");  Thread.*sleep*(5000);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  *log*("释放对象：loc");  loc.unlock();  }  *log*("线程结束");  }  };    t1.setName("t1");  t1.start();    **try** {  //先让t1执行两秒  Thread.*sleep*(2000);  } **catch**(InterruptedException e) {  e.printStackTrace();  }    Thread t2 = **new** Thread() {  **public** **void** run() {  **try** {  *log*("线程启动");  *log*("试图占有loc对象");    loc.lock();    *log*("占有对象：loc");  *log*("进行5秒的业务操作");  Thread.*sleep*(5000);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  *log*("释放对象：loc");  loc.unlock();  }  *log*("线程结束");  }  };    t2.setName("t2");  t2.start();  }  } |

Trylock方法：

synchronized 是**不占用到手不罢休**的，会一直试图占用下去。  
与 synchronized 的**钻牛角尖**不一样，Lock接口还提供了一个trylock方法。  
trylock会在指定时间范围内**试图占用**，占成功了，就嘿嘿嘿。 如果时间到了，还占用不成功，扭头就走~  
注意： 因为使用trylock有可能成功，有可能失败，所以后面unlock释放锁的时候，需要判断是否占用成功了，如果没占用成功也unlock,就会抛出异常。

|  |
| --- |
| **package** Thread;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **import** java.util.concurrent.TimeUnit;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** TryLockTest {  **public** **static** String now() {  **return** **new** SimpleDateFormat("HH:mm:ss").format(**new** Date());  }    **public** **static** **void** log(String msg) {  System.***out***.printf("%s %s %s %n", *now*(), Thread.*currentThread*().getName(), msg);  }    **public** **static** **void** main(String[] args) {  Lock loc = **new** ReentrantLock();    Thread t1 = **new** Thread() {  **public** **void** run() {  **boolean** locked = **false**;  **try** {  *log*("线程1启动");  *log*("试图占有对象：loc");    locked = loc.tryLock(1, TimeUnit.***SECONDS***);  **if**(locked) {  *log*("占有对象：loc");  *log*("进行5秒的业务操作");  Thread.*sleep*(5000);  }**else** {  *log*("经过1秒的努力，还没有占有对象，放弃占有");  }  } **catch**(InterruptedException e) {  e.printStackTrace();  } **finally** {  **if**(locked) {  *log*("释放对象：loc");  loc.unlock();  }  }  *log*("线程结束");  }  };  t1.setName("t1");  t1.start();    **try** {  Thread.*sleep*(5000);  } **catch**(InterruptedException e) {  e.printStackTrace();  }    Thread t2 = **new** Thread() {  **public** **void** run() {  **boolean** locked = **false**;  **try** {  *log*("线程2启动");  *log*("试图占有对象：loc");    locked = loc.tryLock(1, TimeUnit.***SECONDS***);  **if**(locked) {  *log*("占有对象：loc");  *log*("进行5秒的业务操作");  Thread.*sleep*(5000);  }**else** {  *log*("经过1秒的努力，还没有占有对象，放弃占有");  }  } **catch**(InterruptedException e) {  e.printStackTrace();  } **finally** {  **if**(locked) {  *log*("释放对象：loc" );  loc.unlock();  }  }  *log*("线程2结束");  }  };    t2.setName("t2");  t2.start();  }  } |

线程交互

使用synchronized方式进行线程交互，用到的是同步对象的wait,notify和notifyAll方法  
Lock也提供了类似的解决办法，首先通过lock对象得到一个Condition对象，然后分别调用这个Condition对象的：**await**, **signal**,**signalAll**方法  
**注意**： 不是Condition对象的wait,nofity,notifyAll方法,是await,signal,signalAll

|  |
| --- |
| **package** Thread;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **import** java.util.concurrent.locks.Condition;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** ThreadInteractive {  **public** **static** String now() {  **return** **new** SimpleDateFormat("HH:mm:ss").format(**new** Date());  }    **public** **static** **void** log(String msg) {  System.***out***.printf("%s %s %s %n", *now*(), Thread.*currentThread*().getName(), msg);  }    **public** **static** **void** main(String[] args) {  Lock loc = **new** ReentrantLock();  Condition condition = loc.newCondition();    Thread t1 = **new** Thread() {  **public** **void** run() {  **try** {  *log*("线程启动");  *log*("试图占有对象：loc");    loc.lock();    *log*("占有对象；loc");  *log*("进行5秒的业务操作");  Thread.*sleep*(5000);  *log*("临时释放对象loc，并等待");  condition.await();  *log*("重新占有对象loc，并行进行5秒的业务操作");  Thread.*sleep*(5000);    } **catch**(InterruptedException e) {  e.printStackTrace();  } **finally** {  *log*("释放对象：loc");  loc.unlock();  }  *log*("线程结束");  }  };  t1.setName("t1");  t1.start();    **try** {  Thread.*sleep*(3000);  } **catch**(InterruptedException e) {  e.printStackTrace();  }    Thread t2 = **new** Thread() {  **public** **void** run() {  **try** {  *log*("线程启动");  *log*("试图占有对象：loc");    loc.lock();    *log*("占有对象；loc");  *log*("进行5秒的业务操作");  Thread.*sleep*(5000);  *log*("唤醒等待中的线程");  condition.signal();  }**catch**(InterruptedException e) {  e.printStackTrace();  }**finally** {  *log*("释放对象；loc");  loc.unlock();  }  *log*("线程结束");  }  };  t2.setName("t2");  t2.start();  }  } |

1. Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现，Lock是代码层面的实现。  
  
2. Lock可以选择性的获取锁，如果一段时间获取不到，可以放弃。synchronized不行，会一根筋一直获取下去。 借助Lock的这个特性，就能够规避死锁，synchronized必须通过谨慎和良好的设计，才能减少死锁的发生。  
  
3. synchronized在发生异常和同步块结束的时候，会自动释放锁。而Lock必须手动释放， 所以如果忘记了释放锁，一样会造成死锁。

原子性操作概念

所谓的**原子性操作**即不可中断的操作，比如赋值操作

int i = 5;

**原子性操作本身是线程安全的**  
但是 i++ 这个行为，事实上是有3个原子性操作组成的。  
步骤 1. 取 i 的值  
步骤 2. i + 1  
步骤 3. 把新的值赋予i  
这三个步骤，每一步都是一个原子操作，但是合在一起，就不是原子操作。就**不是线程安全**的。  
换句话说，一个线程在步骤1 取i 的值结束后，还没有来得及进行步骤2，另一个线程也可以取 i的值了。  
这也是[分析同步问题产生的原因](https://how2j.cn/k/thread/thread-synchronized/355.html#step787) 中的原理。  
i++ ，i--， i = i+1 这些都是非原子性操作。  
只有int i = 1,这个赋值操作是原子性的。

正则表达式单个字符的匹配规则如下：

