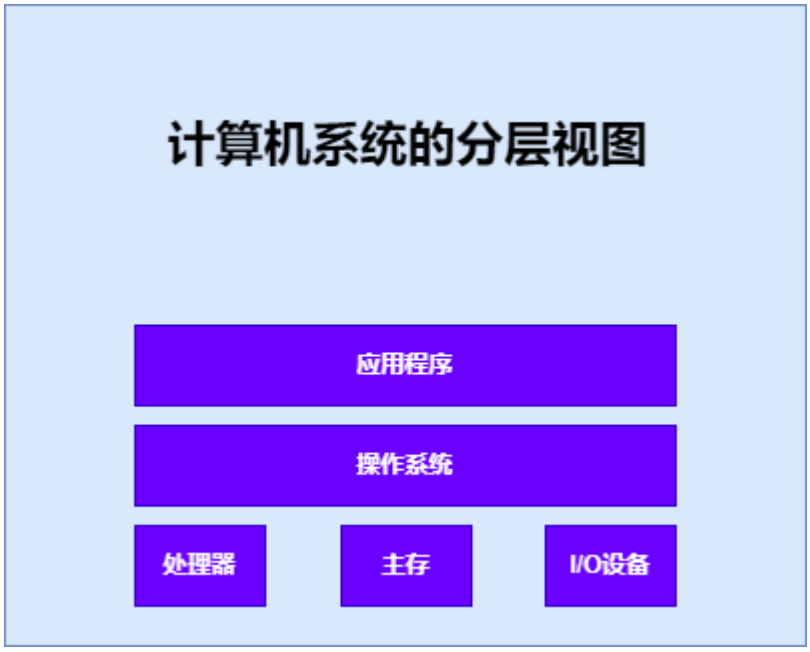
JavaEE多线程部分

操作系统（Operating System）

操作系统是一组做计算机资源管理的软件的统称。目前常见的操作系统有：Windows系列、Unix系列、Linux系列、OSX系列、Android系列、iOS系列、鸿蒙等。

操作系统的定位



操作系统由两个基本功能：

1. 防止硬件被时空的应用程序滥用；
2. 向应用程序提供简单一致的机制来控制复杂而又通常大相径庭的低级硬件设备。

认识进程

1. 进程：正在跑起来的程序~~正在运行的程序

进程是操作系统对一个正在运行的程序的一种抽象，换言之，可以把进程看做程序的一次运行过程；

同时，在操作系统内部，进程又是操作系统进行资源分配的基本单位。

2. 程序是可执行文件，只是硬盘上的一个东西（静态的）
3. 如果双击程序,此时操作系统,就会把可执行文件中的数据和指令,加载到内存中,并且让 cpu 去执行这里的指令，完成一系列相关的工作。运行起来的(动态的)，进程~~
4. 运行起来的进程会消耗CPU资源、内存资源、硬盘、网络带宽……
5. **进程是系统分配硬件资源的基本单位**
6. 计算机中"进程"管理的核心思路：**先描述再组织**
    1. 描述：会使用一个专门的结构体（PCB 进程控制块）来记录一个进程里面的各个属性
    2. 组织：会使用一系列的数据结构，把多个进程进行一个有效的组织，随时方便进行遍历，查找，汇总数据……（通常是使用双向链表这样的方式来进行组织）

进程控制块抽象(PCB Process Control Block)

PCB中大概有哪些信息?

```
// 以下代码是 Java 代码的伪码形式，重在说明，无法直接运行
class PCB {
    // 进程的唯一标识 -- pid;
    // 进程关联的程序信息，例如哪个程序，加载到内存中的区域等
   // 分配给该资源使用的各个资源
    // 进度调度信息（留待下面讲解）
}
```

1. **进程的标识（pid）**：同一个系统上,统一时刻中,每个进程的 pid 一定都是不同的
2. **内存指针**：表示了该进程对应的内存资源是咋样的

   内存资源中要存啥？最主要要存储的就是从exe可执行文件中加载过来的**指令**（二进制的，就是程序员写的代码的逻辑，进一步的再交给CPU来执行）和**数据**（执行的这些指令，会依赖到一些数据），**还**需要保存一些运行过程中的**中间结果之类**的数据
3. **文件描述符表**：每个进程就会有一个“文件描述符表”来记录,当前这个进程正在使用哪些文件

   这就和硬盘资源有关了，硬盘是硬件，应用程序一般是没法直接接触到“硬件”这一层的，实际上是操作系统抽象成“文件”这样的概念，程序操作的是文件，文件实际上是存储在硬盘上的。每个进程就会有一个“文件描述符表”来记录，当前这个进程正在使用哪些文件，操作系统打开一个文件，就会产生一个“文件描述符”（就像文件的身份标识一样，当然，只在进程内部产生），同时会使用文件描述符（类似于数组），把文件描述符给组织起来

   CPU. 进程是需要在CPU上来执行指令的

   早期的CPU都是单核心的，但算力不够就变成多核心了

   进程的调度：

   1. 并行：同一时刻，两个进程，同时运行在两个 cpu 逻辑核心上
   2. 并发：两个进程，在同一个舞台上，轮着上。由于CPU切换进程速度极快，微观上，这俩进程是串行执行的；宏观上，看起来这俩进程就是“同时”执行的
   3. 操作系统在调度这些进程时，两种都有可能
   4. 在应用程序这一层是感知不到的（在系统内核中感知到）。由于感知不到，通常用**“并发”**代指“并行”和“并发”
4. **PCB中关于进程调度相关的属性**（这些属性也就描述了进程对应的 cpu 资源的使用情况）
   1. **状态**

      就绪状态：一个进程已经随时做好了在CPU上执行的准备

      阻塞状态/睡眠状态：进程没有准备好被调度到CPU上

      实际上，进程在系统中状态还有很多种，其中最最关键的就是 就绪 和 阻塞 状态
   2. **优先级**：系统给进程进行调度的时候，也不是完全公平的，也会根据优先级的不同，来决定时间分配的权衡就可以把系统资源调配给更重要的进程上了
   3. **上下文**：这些进程是轮着上的，一次运行不完。就需要保证下次上 cpu 运行的时候，能够从上次运行到的位置，继续往后运行

      对于操作系统来说所记录的上下文，就是该进程在执行过程中，CPU的寄存器中对应的数据

      存档 读档
   4. **记账信息**：相当于是一个统计信息，会统计每个进程在 cpu 上都执行了多久了，执行了多少指令了，是对于进程的调度工作进行一个“兜底“

每个进程有需要有一定的内存资源

在**虚拟地址空间**的加持下 =>进程就具有了"**独立性**"=>每进程有自己的虚拟地址空间 =>一个进程无法直接访问或者修改其他进程虚拟地址空间的内容=>强化了系统稳定性

通过虚拟地址空间,把进程隔离开了，但是有时候，还需要让进程之间,产生点配合/联系

进程间通信：就是在进程隔离性的基础上,开个口子，能够有限制的进行相互影响

多进程已经很好的实现了并发编程的效果了，但是有明显的缺点：

1. 消耗资源更多
2. 速度更慢

认识线程

线程是什么

一个线程就是一个 "执行流". 每个线程之间都可以按照顺讯执行自己的代码. 多个线程之间 "同时" 执行 着多份代码.

轻量级进程 -> 线程（Thread）

为什么要有线程

1. 首先，“并发编程”成为“刚需”
2. 其次, 虽然多进程也能实现 并发编程, 但是线程比进程更轻量
3. 最后, 线程虽然比进程轻量, 但是人们还不满足, 于是又有了 "线程池"(ThreadPool) 和 "协程"

创建的还是进程，创建进程的时候把**资源都分配好**，后续创建的线程，让线程在进程内部（进程和线程之间的关系，**可以认为是进程包含了线程**）

后续进程中的新的**线程，直接复用前面进程这里创建好的资源**

其实一个进程至少包含一个线程，最初创建出来的这个可以认为是一个只包含一个线程的进程（此时创建的过程需要分配资源，此时第一个线程的创建开销可能是比较大的）

但是后续再在这个进程里创建线程，就可以省略分配资源的过程，资源是已经有了的

使用多进程本身已经可以完成并发编程了，但进程比较重，创建和销毁开销很大（需要申请、释放资源），引入线程可以更高效的解决上述问题

所谓的线程，也可以称为轻量级进程

一个进程可以包含一个或多个线程，这个进程中的多个线程共同复用了进程中各种资源（内存、硬盘），但这些线程各自独立在CPU上进行调度

因此，**线程既可以完成“并发编程”的效果，又可以以比较轻量级的方式运行**

线程同样是通过PCB描述的

此时**一个**PCB对应到一个**线程**，**多个**PCB对应一个**进程**了

PCB中的**内存指针、文件描述符表**，同一个进程的多个PCB中，这俩字段的内容都是**一样**的；但是**上下文、状态、优先级、记账信息（这些支持调度的属性）**，则这些PCB每个人的都**不一样**了。

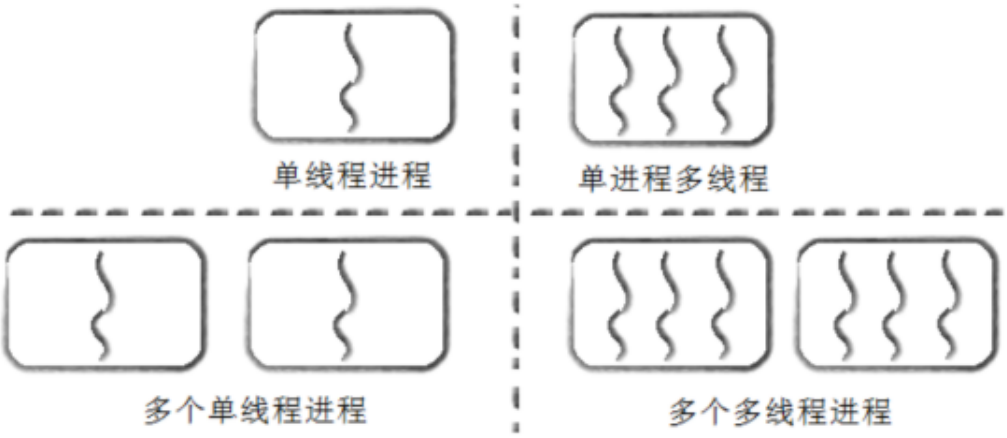
则引出这么句话：**进程是操作系统进行资源分配的基本单位；线程是操作系统进行调度执行的基本单位。**

随着线程数量的增加，整体的效率也会再进一步的提高。但也不是无限的提高，CPU的核心逻辑数是有限的。**线程太多的时候**，线程调度的开销反而会**拖慢整个程序的效率**，同时容易产生**冲突**

这样的冲突会产生bug，就会带来“线程安全问题”（多线程编程中最关键的问题）

一旦某个线程执行过程中出现异常，并且这个异常没有被很好的处理，就可能导致整个进程直接终止。（进程中的所有线程也就随之终止了）。这也体现出进程的**“隔离性”**的好处

面试题：谈谈进程和线程的区别和联系（以下三条是最核心的，能多说更好）



1. **进程包含线程**，都是为了实现并发编程的方式，线程比进程更轻量；
2. **进程是系统分配资源的基本单位，线程是系统调度执行的基本单位**，创建进程的时候把分配资源（虚拟地址空间、文件描述符表）的工作干了，后续创建线程，直接共用之前的资源即可
3. **进程有独立的地址空间**，彼此之间不会相互影响到，进程的独立性=>系统稳定性；**多个线程共用一份地址空间**，一个线程一旦抛出异常，就可能导致整个进程异常结束=>多个线程之间容易相互影响

线程是更轻量的，但也不是没有创建成本。在互联网圈子，高并发的服务器，要处理的并发量太多了，非常非常频繁的创建线程/销毁。线程开销仍然不可忽视了

此时又有两种方法解决：

1. “轻量级线程” =>协程/纤程
2. 线程池 =>把一些要释放的资源，不要着急释放，而是先放到一个池子里，以备后续使用。申请资源的时候，也是先提前把要申请的资源申请好，也放到一个"池子里“，后续申请的时候也比较方便

线程本身是操作系统提供的概念。操作系统也提供了一些ap供程序猿来使用。

Java中，就把操作系统的apl又进行了封装，提供了Thread类

```java
class MyThread extends Thread{
    @Override
    public void run() {
        while(true){
            System.out.println("hello thread");
            try {
                //这里只能trycatch，不能throws
                //此处是方法重写，对于父类的run方法，就没有throws xxx异常这样的设定
                //在重写的时候，也就不能throws异常了
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class Demo1 {
    public static void main(String[] args) {
        MyThread myThread=new MyThread();
        myThread.start();//start会调用系统API，在系统内核中把线程对应的PCB啥的给创建出来并管理好，新的线程就参与调度了
        //myThread.run();//run只是上面的入口方法（普通的方法），并没有调用API，也没有创建出真正的线程来
        while(true){
            System.out.println("hello main");
            try {
                //这里可以throws
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

//运行结果
hellomain
hellothread
hellomain
hellothread
hellothread
hellomain
hellothread
hellomain
hellothread
hellomain
hellothread
hellomain
hellothread
hellomain
hellomain
hellothread
hellomain
......
```

当点击运行程序的时候，就会先创建出一个java进程。这个进程中就包含了至少一个线程，这个线程也叫做主线程，也就是负责执行`main`方法的线程

调整了代码之后， 在`main`方法中有一个 `while`循环,在线程的 run 中也有一个 `while` 循环这两个循环都是 `while (true)`死循环

使用`start`的方式执行，此时，这俩循环都在执行

两个线程,分别执行自己的循环这两个线程都能参与到 cpu 的调度中，这俩线程(这俩`while`循环) 并发式的执行

其中的`sleep`是`Thread`类的静态方法

这俩线程在进行`sleep` 之后, 就会进入阻塞状态。当时间到，系统就会唤醒这俩线程，并且恢复对这俩线程的调度。当这俩线程都唤醒了之后，谁先调度，谁后调度，可以视为是“随机”的

每个线程都能被独立的调度执行，都是一个独立的执行流，多个线程之间是并发的关系

主线程和新线程是并发执行的关系，就看操作系统怎么调度

系统在进行多个线程调度的时候，并没有一个非常明确的的顺序，而是按照这种“随机”的方式进行调度，这样的“随机”调度的过程，称为“**抢占式执行**”。只是看起来随机，实际上概率并不均等。

创建线程

Java中**创建线程**的方式有很多：实现接口，继承类

1. 创建一个类，继承Thread类，重写run方法

```
class MyThread extends Thread {
    @Override
    public void run() {
        // 线程执行的逻辑
    }
}
```

2. 创建一个类，实现Runnable接口，重写run方法

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // 线程执行的逻辑
    }
}
```

对比上面两种方法:

- 继承 Thread 类, 直接使用 this 就表示当前线程对象的引用.
- 实现 Runnable 接口, this 表示的是 MyRunnable 的引用. 需要使用 `Thread.currentThread()`

3. 继承 Thread，重写 run，基于匿名内部类

```
//3、创建了该子类的实例，用thread这个引用指向
Thread thread = new Thread() {//1、创建了一个子类，继承自Thread，但是是匿名的
    @Override
    public void run() {//2、在子类中重写run()方法
        // 在这里定义线程运行的任务
    }
};
thread.start();
```

4. 实现 Runnable，重写 run，基于匿名内部类

```
//3、创建了该子类的实例，用thread这个引用指向
Thread thread=new Thread(new Runnable(){//1、创建了一个Runnable的子类，通过构造方法传给
Thread
    @Override
    public void run() {//2、在子类中重写run()方法
        // 在这里定义线程运行的任务
    }
});
thread.start();
```

5. 使用lambda表达式，表示run()方法的内容（最推荐）

```
Thread t=new Thread(()->{
    while(true){
        System.out.println("hellothread");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
t.start();
while (true){
    System.out.println("hellomain");
    Thread.sleep(1000);
}
```

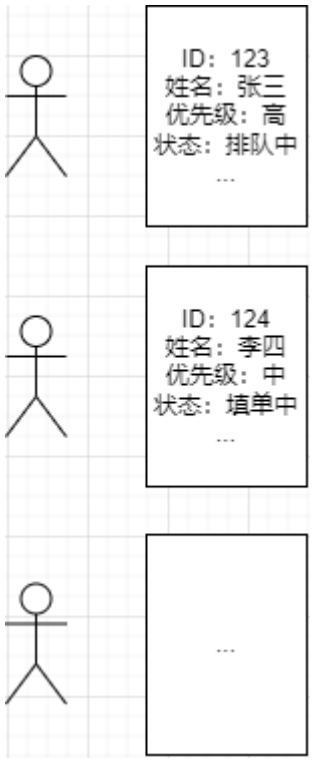
6. 基于Callable
7. 基于线程池

Thread

Thread 就是在 Java 中，线程的代言人。系统中的一个线程，就对应到 Java 中的一个 Thread 对象。围绕线程的各种操作，都是通过 Thread 来展开的

Thread 类是 JVM 用来管理线程的一个类，换句话说，每个线程都有一个唯一的 Thread 对象与之关联

用我们上面的例子来看，每个执行流，也需要有一个对象来描述，类似下图所示，而 Thread 类的对象就是用来描述一个线程执行流的，JVM 会将这些 Thread 对象组织起来，用于线程调度，线程管理。



Thread类的常用构造方法

方法	说明
Thread()	创建线程对象
Thread(Runnable target)	使用 Runnable 对象创建线程对象
Thread(String name)	创建线程对象，并命名
Thread(Runnable target, Strin name)	使用 Runnable 对象创建线程对象，并命名

```
Thread t1 = new Thread();
Thread t2 = new Thread(new MyRunnable());
Thread t3 = new Thread("这是我的名字");
Thread t4 = new Thread(new MyRunnable(), "这是我的名字");```
```

Thread类的常见属性

属性	获取方法
ID	getId()
名称	getName()
状态	getState()
优先级	getPriority()
是否后台线程	isDaemon()
是否存活	isAlive()
是否被中断	isInterrupted()

1. ID（getId）：线程的身份表示（在JVM这里给线程设定的身份标识），一个线程可以有多个身份标识
2. 名称（getName）：各种调试工具用到
3. 状态（getState）：Java中的线程状态和操作系统中有一定差异
4. 优先级（getPriority）：设置/获取线程优先级作用不是很大，线程的调度主要还是系统内核来负责的，系统调度的速度实在太快
5. 是否后台线程（isDaemon）
    - 后台线程/守护线程：不太影响进程结束
    - 前台线程：会影响进程结束，如果前台线程没执行完，进程是不会结束的
    - 一个进程中所有的前台线程都执行完，退出了，此时即使存在后台线程仍然没执行完也会随着进程一起退出
    - 影响进程退出的就是前台，不影响的就是后台
    - 创建的线程默认是前台线程，通过setDaemon显式的设置成后台
6. 是否存活（isAlive）：Thread对象，对应的线程（系统内核中）是否存活。
    - Thread对象的生命周期，并不是和系统中的线程完全一致的
7. 是否被中断（isInterrupted）：看下面

启动一个线程-start()

1. start方法，在系统中真正创建出线程
    1. 创建出PCB
    2. 把PCB加入到对应链表
2. 操作系统内核=内核+配套的程序
3. 内核：一个系统最核心的功能
    1. 对下，管理好各种硬件设备
    2. 对上，给各种程序提供稳定的运行环境
4. start方法本身执行成功是一瞬间的，只是告诉系统你要创建一个线程，调用完start后，代码就会立即继续执行start后续的逻辑

终止一个线程

1. 一个线程的run方法执行完毕，就算终止了
2. 此处的终止线程，就是想办法让run方法能够尽快执行完毕
3. 方法：
    1. 程序员手动设定标志位，通过这个来让run尽快结束

```
private static class MyRunnable implements Runnable {
    public volatile boolean isQuit = false;

    @Override
    public void run() {
```

```
        while (!isQuit) {
            System.out.println(Thread.currentThread().getName()
                    + ": 别管我，我忙着转账呢!");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(Thread.currentThread().getName()
                + ": 啊！险些误了大事");
    }
}

public static void main(String[] args) throws InterruptedException {
    MyRunnable target = new MyRunnable();
    Thread thread = new Thread(target, "李四");
    System.out.println(Thread.currentThread().getName()
            + ": 让李四开始转账。");
    thread.start();
    Thread.sleep(10 * 1000);
    System.out.println(Thread.currentThread().getName()
            + ": 老板来电话了，得赶紧通知李四对方是个骗子！");
    target.isQuit = true;
}
```

2. 直接Thread类，给我们提供好了现成的标志位，不用手动设置标志位

```
private static class MyRunnable implements Runnable {
    @Override
    public void run() {
        // 两种方法均可以
        while (!Thread.interrupted()) {
            //while (!Thread.currentThread().isInterrupted()) {
            System.out.println(Thread.currentThread().getName()
                    + ": 别管我，我忙着转账呢!");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
                System.out.println(Thread.currentThread().getName()
                        + ": 有内鬼，终止交易！");
                // 注意此处的 break
                break;
            }
        }
        System.out.println(Thread.currentThread().getName()
                + ": 啊！险些误了大事");
    }
}

public static void main(String[] args) throws InterruptedException {
    MyRunnable target = new MyRunnable();
    Thread thread = new Thread(target, "李四");
    System.out.println(Thread.currentThread().getName()
```

```
                + ": 让李四开始转账。");
        thread.start();
        Thread.sleep(10 * 1000);
        System.out.println(Thread.currentThread().getName()
                + ": 老板来电话了，得赶紧通知李四对方是个骗子！");
        thread.interrupt();
    }
```

4. 当sleep被唤醒以后，程序员可以有以下几种操作方式：
    1. 立即停止循环，立即结束线程（直接break）
    2. 继续做点别的事情，过一会儿再结束线程（catch中执行别的逻辑，执行完再break）
    3. 忽略终止的请求，继续循环（不写break）
5. thread 收到通知的方式有两种：
    1. 如果线程因为调用 wait/join/sleep 等方法而阻塞挂起，则以 InterruptedException 异常的形式通知，**清除中断标志**
        - 当出现 InterruptedException 的时候, 要不要结束线程取决于 catch 中代码的写法. 可以选择忽略这个异常, 也可以跳出循环结束线程
    2. 否则，只是内部的一个中断标志被设置，thread 可以通过
        - Thread.interrupted() 判断当前线程的中断标志被设置，**清除中断标志**
        - Thread.currentThread().isInterrupted() 判断指定线程的中断标志被设置，**不清除中断标志**

等待一个线程-join()

1. 多个线程是并发执行的。具体的执行过程都是由操作系统负责调度的。操作系统的调度线程的过程，是“随机”的。无法确定线程执行的先后顺序
2. 等待线程，就是一种规划 **线程结束** 顺序的手段
3. 注意：join()方法也会抛出Interrupted异常
4. **谁调用join方法，谁就强占cpu资源，直至执行结束**
5. A B两个线程，希望B先结束A后结束，此时就可以让A线程中调用B.join()的方法。此时，B线程还没执行完，A线程就会进入"阻塞"状态。就相当于给B留下了执行的时间。B执行完毕之后，A再从阻塞状态中恢复回来，并且继续往后执行。如果A执行到B.join()的时候，B已经执行完了，A就不必阻塞了，直接往下执行就可以了
6. 阻塞：让代码暂时不继续执行了（该线程暂时不去CPU上参与调度）
7. sleep也能让线程阻塞，阻塞是有时间限制的
8. join的阻塞，则是“死等”“不见不散”
9. **sleep、join、wait...产生阻塞之后，都是可能被interrupt方法唤醒的**，这几个方法都会在被唤醒之后自动清除标志位（和sleep类似），这些方法都会抛出Interrupted异常

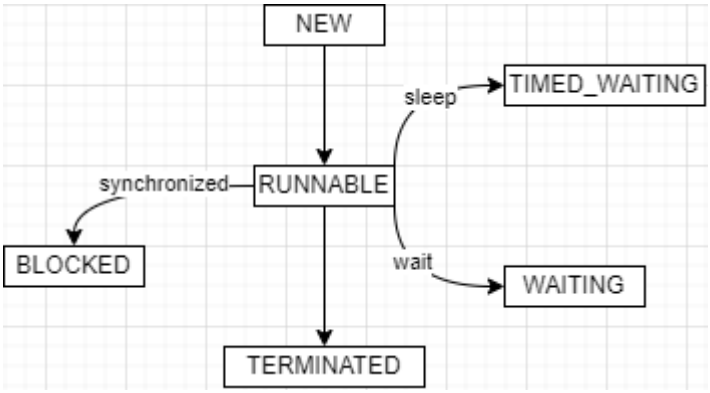
线程的状态

- 先谈谈进程里PCB里的状态字段：**就绪**和**阻塞**状态（这些是系统设定的状态，Java又把这些细分了）

- 以下则是**线程的状态：**

1. NEW**（开始前）**: 安排了工作, 还未开始行动 （Thread对象创建好了，但还没有调用start()方法）
2. RUNNABLE**（就绪状态）**: 可工作的. 又可以分成**正在工作中**和**即将开始工作**
    1. 线程正在CPU上运行
    2. 线程正在排队，随时可以去CPU运行
3. BLOCKED**（阻塞状态）**: 这几个都表示排队等着其他事情： 因为锁
4. WAITING**（阻塞状态）**: 这几个都表示排队等着其他事情： 因为调用了wait
5. TIMED_WAITING**（阻塞状态）**: 这几个都表示排队等着其他事情： 因为调用了sleep
6. TERMINATED**（结束后）**: 工作完成了

- 线程状态转换图



线程安全

如果多线程环境下代码运行的结果是符合我们预期的，即在单线程环境应该的结果，则说这个程序是线程安全的。

线程安全问题的原因

1. [根本原因]多个线程之间的调度顺序是“随机”的，操作系统使用“抢占式”执行的策略来调度线程
2. **多个**线程同时**修改同一个**遍历，容易产生线程安全问题
    - 3个条件
    1. 多个
    2. 修改
    3. 同一个
3. 进行的修改，不是“原子的”，如果修改操作能按照原子的形式完成，就不会有线程安全问题（原子，即不可再分）
4. 内存可见性，引起的线程安全问题
5. 指令重排序，引起的线程安全问题

- **以上五个原因，只有第3个原因能想办法修正**
- 通过“加锁”的方式，把一组操作给打包成一个“原子”的操作。此处的原子，就是通过锁，进行“互斥”，**我这个线程工作的时候，其他线程无法工作**

1. 原子性：

   什么是原子性

   我们把一段代码想象成一个房间，每个线程就是要进入这个房间的人。如果没有任何机制保证，A 进入房间之后，还没有出来；B 是不是也可以进入房间，打断 A 在房间里的隐私。这个就是不具备原子性的。

   那我们应该如何解决这个问题呢？是不是只要给房间加一把锁，A 进去就把门锁上，其他人是不是就进不来了。这样就保证了这段代码的原子性了。

   有时也把这个现象叫做同步互斥，表示操作是互相排斥的。

   如果一个线程正在对一个变量操作，中途其他线程插入进来了，如果这个操作被打断了，结果就可能是错误的。

2. 可见性： 一个线程对共享变量值的修改，能够及时地被其他线程看到

3. 代码顺序性：

   什么是代码重排序

   一段代码是这样的：

   1. 去前台取下 U 盘
   2. 去教室写 10 分钟作业
   3. 去前台取下快递

如果是在单线程情况下，JVM、CPU指令集会对其进行优化，比如，按 1->3->2的方式执行，也是没问题，可以少跑一次前台。这种叫做指令重排序

> 编译器对于指令重排序的前提是 "保持逻辑不发生变化". 这一点在单线程环境下比较容易判断, 但是在多线程环境下就没那么容易了, 多线程的代码执行复杂程度更高, 编译器很难在编译阶段对代码的执行效果进行预测, 因此激进的重排序很容易导致优化后的逻辑和之前不等价.

synchronized 关键字-监视器锁monitor lock

代码中的锁就是让多个线程，同一时刻，只有一个线程能使用这个变量

synchronized的特性

1. 互斥

synchronized 会起到互斥效果, 某个线程执行到某个对象的 synchronized 中时, 其他线程如果也执行到同一个对象 synchronized 就会**阻塞等待**.

- 进入 synchronized 修饰的代码块, 相当于 **加锁**
- 退出 synchronized 修饰的代码块, 相当于 **解锁**

```
//进入就针对当前对象“加锁”
synchronized public void increase(){
    count++;
}
//出来就针对当前对象“解锁”
```

synchronized关键字最主要有以下3种应用方式：

- 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁

- 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
- 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码块前要获得给定对象的锁。（这里可以给实例对象的名称`Test test=new Test()`的`test`，也可以给`this`对象代表当前实例，也可以给当前类的`class`对象作为锁）
- **思考：通过加锁操作之后把并发执行=>串行执行了，此时多线程还有存在的意义吗?**
- **答：因为两个线程，可能有一部分代码是串行执行的，有一部分是并发执行的=>这仍然比纯粹的串行执行效率要高!**

> 理解 "阻塞等待".
>
> 针对每一把锁, 操作系统内部都维护了一个等待队列. 当这个锁被某个线程占有的时候, 其他线程尝试进行加锁, 就加不上了, 就会阻塞等待, 一直等到之前的线程解锁之后, 由操作系统唤醒一个新的线程, 再来获取到这个锁.
>
> **注意**:
>
> - 上一个线程解锁之后, 下一个线程并不是立即就能获取到锁. 而是要靠操作系统来 "唤醒". 这也就是操作系统线程调度的一部分工作.
> - 假设有 A B C 三个线程, 线程 A 先获取到锁, 然后 B 尝试获取锁, 然后 C 再尝试获取锁, 此时 B 和 C 都在阻塞队列中排队等待. 但是当 A 释放锁之后, 虽然 B 比 C 先来的, 但是 B 不一定就能获取到锁, 而是和 C 重新竞争, 并不遵守先来后到的规则

synchronized用的锁是存在Java对象里的。

synchronized进行加锁解锁，其实是以“对象”为维度进行展开的。

加锁目的是为了互斥使用资源。（互斥的修改变量）

使用synchronized的时候，其实是指定了某个具体的对象进行加锁，当synchronized直接修饰方法时，此时就相当于是针对this加锁（修饰方法相当于这段代码的简化写法）[不存在所谓的“同步方法”的概念]

```
class Counter{
    public int count=0;
    public void increace(){
        synchronized (this){//this就是下面调用的counter
            count++;
        }
    }
    public void increace2(){
        count++;
    }
    public synchronized static void func(){
        synchronized (Counter.class){

        }
    }
}
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter=new Counter();

        Thread t1=new Thread(()->{
```

```
            for (int i = 0; i < 50000; i++) {
                counter.increace();//这里t1的counter和下面t2的counter进行锁竞争/锁冲突
            }
        });
        Thread t2=new Thread(()->{
            for (int i = 0; i < 50000; i++) {
                counter.increace();//这里t2的counter和上面t1的counter进行锁竞争/锁冲突
            }
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println(counter.count);
    }
}
```

如果是**两个线程针对同一个对象进行加锁，就会出现锁竞争/锁冲突（一个线程能加锁成功，另一个线程阻塞等待）**

如果是两个线程针对不同对象进行加锁，就不会出现锁竞争/锁冲突，也就不存在阻塞等待的操作了

因此具体针对哪个对象加锁不重要，**重要的是两个线程，是不是针对同一个对象加锁**

- **思考：如果接下来的代码里，一个线程加锁了，一个线程没加锁，此时是否还会存在线程安全问题**
- **答：单方面加锁等于没加锁，必须得多个线程都对同一个对象加锁，才有意义**

synchronized的底层是使用操作系统的mutex lock实现的.

synchronized有且只有一条规则：

当两个线程针对同一个对象加锁的时候，就会出现锁竞争/锁冲突。一个线程能先拿到锁，另一个线程就会阻塞等待（BLOCKED）。直到第一个线程释放了锁之后，第二个线程才可能获取到锁，才能继续往下执行。

2. 刷新内存

synchronized 的工作过程:

1. 获得互斥锁
2. 从主内存拷贝变量的最新副本到工作的内存
3. 执行代码
4. 将更改后的共享变量的值刷新到主内存
5. 释放互斥锁

3. 可重入

synchronized 同步块对同一条线程来说是可重入的，不会出现自己把自己锁死的问题

> **理解 "把自己锁死"**
>
> 一个线程没有释放锁, 然后又尝试再次加锁.
>
> ```
> // 第一次加锁, 加锁成功
> lock();
> // 第二次加锁, 锁已经被占用, 阻塞等待.
> lock();
> ```
>
> 按照之前对于锁的设定, 第二次加锁的时候, 就会阻塞等待. 直到第一次的锁被释放, 才能获取到第二个锁. 但是释放第一个锁也是由该线程来完成, 结果这个线程已经躺平了, 啥都不想干了, 也就无法进行解锁操作. 这时候就会 **死锁**.
>
> 这样的锁称为 **不可重入锁**.

Java 中的 synchronized 是 **可重入锁**, 因此没有上面的问题

代码示例

在下面的代码中,

- increase 和 increase2 两个方法都加了 synchronized, 此处的 synchronized 都是针对 this 当前对象加锁的.
- 在调用 increase2 的时候, 先加了一次锁, 执行到 increase 的时候, 又加了一次锁. (上个锁还没释放, 相当于连续加两次锁)

这个代码是完全没问题的. 因为 synchronized 是可重入锁.

```
static class Counter {
    public int count = 0;
    synchronized void increase() {
        count++;
   }
    synchronized void increase2() {
        increase();
   }
}
```

在可重入锁的内部, 包含了 "线程持有者" 和 "计数器" 两个信息.

- 如果某个线程加锁的时候, 发现锁已经被人占用, 但是恰好占用的正是自己, 那么仍然可以继续获取到锁, 并让计数器自增.
- 解锁的时候计数器递减为 0 的时候, 才真正释放锁. (才能被别的线程获取到)

synchronized使用范例

synchronized 本质上要修改指定对象的 "对象头". 从使用角度来看, synchronized 也势必要搭配一个具体的对象来使用.

1. **直接修饰普通方法**:锁的 SynchronizedDemo 对象

```
public class SynchronizedDemo {
    public synchronized void methond() {
    }
}
```

2. **修饰静态方法**: 锁的 SynchronizedDemo 类的对象

```
public class SynchronizedDemo {
    public synchronized static void method() {
    }
}
```

3. **修饰代码块**: 明确指定锁哪个对象.

锁当前对象

```
public class SynchronizedDemo {
    public void method() {
        synchronized (this) {//Test test=new Test()的test也行

       }
    }
}
```

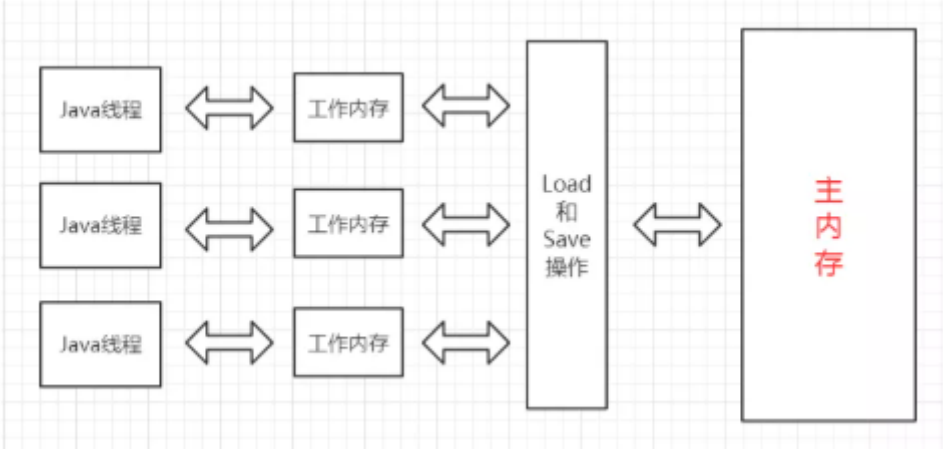
锁类对象

```
public class SynchronizedDemo {
    public void method() {
        synchronized (SynchronizedDemo.class) {
       }
   }
}
```

volatile

volatile能保证内存可见性

volatile 修饰的变量, 能够保证 "内存可见性".



代码在写入 volatile 修饰的变量的时候,

- 改变线程工作内存中volatile变量副本的值
- 将改变后的副本的值从工作内存刷新到主内存

代码在读取 volatile 修饰的变量的时候,

- 从主内存中读取volatile变量的最新值到线程的工作内存中
- 从工作内存中读取volatile变量的副本

代码示例

在这个代码中

- 创建两个线程 t1 和 t2
- t1 中包含一个循环, 这个循环以 flag == 0 为循环条件.
- t2 中从键盘读入一个整数, 并把这个整数赋值给 flag.
- 预期当用户输入非 0 的值的时候, t1 线程结束.

```
static class Counter {
    public int flag = 0;
}
public static void main(String[] args) {
    Counter counter = new Counter();
    Thread t1 = new Thread(() -> {
        while (counter.flag == 0) {
            // do nothing
       }
        System.out.println("循环结束!");
   });
    Thread t2 = new Thread(() -> {
        Scanner scanner = new Scanner(System.in);
        System.out.println("输入一个整数:");
        counter.flag = scanner.nextInt();
   });
    t1.start();
    t2.start();
}
// 执行效果
// 当用户输入非0值时, t1 线程循环不会结束. (这显然是一个 bug)
// 注意: 这里直接在控制台是看不出的
```

> t1 读的是自己工作内存中的内容.
>
> 当 t2 对 flag 变量进行修改, 此时 t1 感知不到 flag 的变化.

如果给 flag 加上 volatile

```
static class Counter {
    public volatile int flag = 0;
}
// 执行效果
// 当用户输入非0值时, t1 线程循环能够立即结束.
// 注意：这里直接在控制台同样是看不出的
```

volatile不保证原子性

volatile 和 synchronized 有着本质的区别. synchronized 能够保证原子性, volatile 保证的是内存可见性.

synchronized 也能保证内存可见性

synchronized 既能保证原子性, 也能保证内存可见性.

内存可见性问题:

1. 编译器优化
2. 内存模型
3. 多线程

- volatile保证的是内存可见性，不是原子性

内存可见性和**加锁**描述了线程安全问题的典型情况和处理方式

wait 和 notify

由于线程之间是抢占式执行的, 因此线程之间执行的先后顺序难以预知.

但是实际开发中有时候我们希望合理的协调多个线程之间的执行先后顺序.

完成这个协调工作, 主要涉及到三个方法：

- wait() / wait(long timeout): 让当前线程进入等待状态.
- notify() / notifyAll(): 唤醒在当前对象上等待的线程.

> **注意**: wait, notify, notifyAll 都是 Object 类的方法.
>
> wait（等待）和notify（通知）就是一个用来协调线程顺序的重要工具
>
> 这两个方法都是Object提供的方法，随便找个对象都可以调用

当wait引起线程阻塞之后，可以使用interrupt方法把线程唤醒，打断当前线程的阻塞状态

wait()方法

wait在执行的时候，会做三件事：

1. 解锁。object.wait，就会尝试针对object对象解锁
2. 阻塞等待
3. 当被其他线程唤醒之后，就会尝试重新加锁，加锁成功，wait执行完毕，继续往下执行其他逻辑。

wait要解锁前提是先能加上锁

- 核心解决思路：先加锁，在synchronized里头再wait，这样子的wait就会一直阻塞到其他线程进行notify了

注意事项：

1. 要想让notify能够顺利唤醒wait，就需要确保wait和notify都是使用同一个对象调用的。
2. wait和notify都需要放到synchronized之内的。虽然notify不涉及“解锁操作”，但是Java也强制要求notify要放到synchronized中。（系统的原生api中就没有这个要求）
3. 如果进行notify的时候，另一个线程并没有处于wait状态，此时，notify相当于“空打一炮”，不会有任何副作用

代码示例: 观察wait()方法使用

```
public static void main(String[] args) throws InterruptedException {
    Object object = new Object();
    synchronized (object) {
        System.out.println("等待中");
        object.wait();
        System.out.println("等待结束");
   }
}
```

这样在执行到object.wait()之后就一直等待下去，那么程序肯定不能一直这么等待下去了。这个时候就需要使用到了另外一个方法唤醒的方法notify()

线程可能有多个，比如可以有n个线程进行wait一个线程负责notify，notify操作只会唤醒一个线程。具体是唤醒了哪个线程？是随机的！

wait和sleep的区别：

- sleep有个明确的时间，到达时间自然就会被唤醒，也能提前唤醒，使用interrupt
- wait默认是个死等，一直等到其他线程notify，wait也能被interrupt提前唤醒

notify()方法

notify 方法是唤醒等待的线程.

- 方法notify()也要在同步方法或同步块中调用，该方法是用来通知那些可能等待该对象的对象锁的其它线程，对其发出通知notify，并使它们重新获取该对象的对象锁。
- 如果有多个线程等待，则有线程调度器随机挑选出一个呈 wait 状态的线程。(并没有 "先来后到")

- 在notify()方法后，当前线程不会马上释放该对象锁，要等到执行notify()方法的线程将程序执行完，也就是退出同步代码块之后才会释放对象锁。

代码示例: 使用notify()方法唤醒线程

- 创建 WaitTask 类, 对应一个线程, run 内部循环调用 wait.
- 创建 NotifyTask 类, 对应另一个线程, 在 run 内部调用一次 notify
- 注意：WaitTask 和 NotifyTask 内部持有同一个 Object locker. WaitTask 和 NotifyTask 要想配合就需要搭配同一个 Object.

```
static class WaitTask implements Runnable {
    private Object locker;
    public WaitTask(Object locker) {
        this.locker = locker;
    }
    @Override
    public void run() {
        synchronized (locker) {
            while (true) {
                try {
                    System.out.println("wait 开始");
                    locker.wait();
                    System.out.println("wait 结束");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
       }
    }
}
static class NotifyTask implements Runnable {
    private Object locker;
    public NotifyTask(Object locker) {
        this.locker = locker;
    }
    @Override
    public void run() {
        synchronized (locker) {
            System.out.println("notify 开始");
            locker.notify();
            System.out.println("notify 结束");
       }
    }
}
public static void main(String[] args) throws InterruptedException {
    Object locker = new Object();
    Thread t1 = new Thread(new WaitTask(locker));
    Thread t2 = new Thread(new NotifyTask(locker));
    t1.start();
    Thread.sleep(1000);
    t2.start();
}
```

notifyAll()方法

notify方法只是唤醒某一个等待线程. 使用notifyAll方法可以一次唤醒所有的等待线程.

如果就想唤醒某个指定的线程，就可以让不同的线程使用不同的对象来进行 wait，想唤醒谁，就可以使用对应的对象来notify

范例：修改 NotifyTask 中的 run 方法, 把 notify 替换成 notifyAll

```
public void run() {
    synchronized (locker) {
        System.out.println("notify 开始");
        locker.notifyAll();
        System.out.println("notify 结束");
   }
}
```

注意：虽然是同时唤醒 3 个线程, 但是这 3 个线程需要竞争锁. 所以并不是同时执行, 而仍然是有先有后的执行.

wait和sleep的区别（面试题）：

1. sleep有个明确的时间，到达时间自然就会被唤醒，也能提前唤醒，使用interrupt
2. wait默认是个死等，一直等到其他线程notify，wait也能被interrupt提前唤醒
3. wait 需要搭配 synchronized 使用，sleep 不需要.
4. wait 是 Object 的方法 sleep 是 Thread 的静态方法.

多线程案例

单例模式

单例模式是一种**设计模式**

设计模式，就是程序员的棋谱，这里介绍了很多典型场景，以及典型场景的处理方式，按照设计模式写代码，代码写的肯定不会很差。

单例模式对应的场景，有些时候希望有的对象，在整个程序中只有一个实例（对象），只能new一次

写法：

1. 饿汉模式（迫切）：程序启动，类加载之后，立即创建出实例

```
class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return instance;
   }
}
```

2. 懒汉模式（延时）：类加载的时候不创建实例，则是在第一次使用实例的时候，再创建，否则，能不创建就不创建

```
class Singleton {
    private static Singleton instance = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
       }
        return instance;
   }
}
```

- **懒汉模式-多线程版**

上面的懒汉模式的实现是线程不安全的

> 线程安全问题发生在首次创建实例时. 如果在多个线程中同时调用 getInstance 方法, 就可能导致创建出多个实例.
>
> 一旦实例已经创建好了, 后面再多线程环境调用 getInstance 就不再有线程安全问题了(不再修改 instance 了)

加上 synchronized 可以改善这里的线程安全问题

```
class Singleton {
    private static Singleton instance = null;
    private Singleton() {}
    public synchronized static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
       }
        return instance;
   }
}
```

- **懒汉模式-多线程版(改进)**

以下代码在加锁的基础上, 做出了进一步改动:

- 使用双重 if 判定, 降低锁竞争的频率.
- 给 instance 加上了 volatile

```
class Singleton {
```

```
    private static volatile Singleton instance = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {//第一个条件是为了判断是否要加锁，如果不是null就没必要加锁了
            synchronized (Singleton.class) {//是null就加锁
           if (instance == null) {//第二个条件是判断是否要创建对象，因为在多线程环境下可能有多个线程同时通过第一个条件检查
               instance = new Singleton();
               }
           }
       }
        return instance;
   }
}
```

即先判断是否要加锁，再决定是不是真的加锁

> 加锁 / 解锁是一件开销比较高的事情. 而懒汉模式的**线程不安全只是发生在首次创建实例的时候. 因此后续使用的时候, 不必再进行加锁了.**
>
> 外层的 if 就是判定下看当前是否已经把 instance 实例创建出来了.
>
> 同时为了避免 "内存可见性" 导致读取的 instance 出现偏差, 于是补充上 volatile .
>
> 当多线程首次调用 getInstance, 大家可能都发现 instance 为 null, 于是又继续往下执行来竞争锁, 其中竞争成功的线程, 再完成创建实例的操作.
>
> 当这个实例创建完了之后, 其他竞争到锁的线程就被里层 if 挡住了. 也就不会继续创建其他实例.

这里加上`volatile`还有一个作用：避免此处赋值操作的**指令重排序**

指令重排序：是编译器优化的一种手段，在原有执行逻辑不变的情况下，对代码执行顺序进行调整，是其执行效率变高

在单线程没事，但在多线程就可能出现问题了

比如我们上述代码在`instance = new Singleton();`这里我们可以分为3个步骤：

1. 给对象创建出内存空间，得到内存地址
2. 在空间上调用构造方法，对对象进行初始化
3. 把内存地址，赋值个instance引用

这里发生指令重排序，步骤可能就变为132了，同样是在单线程没事，但多线程就不行了

比如线程1在这里执行132，在3执行完后2还没执行，出现了线程切换，线程2执行的时候`instance!=null`，就直接返回`instance`了，后续可能使用`instance`的属性方法之类的，因为还没初始化，就可能出现啥情况了。于是加入`volatile`就是避免这样的事情发生

这个**懒汉模式-多线程版(改进)**的关键就3点：

1. 加锁
2. 双重`if`
3. `volatile`

阻塞队列

阻塞队列是什么

阻塞队列是一种特殊的队列. 也遵守 "先进先出" 的原则。

阻塞队列能是一种线程安全的数据结构, 并且具有以下特性:

- 当队列满的时候, 继续入队列就会阻塞, 直到有其他线程从队列中取走元素.
- 当队列空的时候, 继续出队列也会阻塞, 直到有其他线程往队列中插入元素.

阻塞队列的一个典型应用场景就是 "**生产者消费者模型**". 这是一种非常典型的开发模型.

生产者消费者模型

生产者消费者模式就是通过一个容器来解决生产者和消费者的强耦合问题。

生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给**阻塞队列**，消费者不找生产者要数据，而是直接从**阻塞队列**里取.

1. 阻塞队列就相当于一个**缓冲区**，平衡了生产者和消费者的处理能力.（**削峰**）

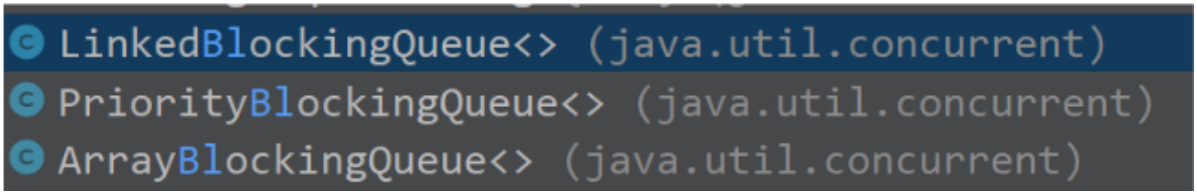
> 比如在 "秒杀" 场景下, 服务器同一时刻可能会收到大量的支付请求. 如果直接处理这些支付请求, 服务器可能扛不住(每个支付请求的处理都需要比较复杂的流程). 这个时候就可以把这些请求都放到一个**阻塞队列**中, 然后再由消费者线程慢慢的来处理每个支付请求.
>
> 这样做可以有效进行 "**削峰**", 防止服务器被突然到来的一波请求直接冲垮.

2. 阻塞队列也能使生产者和消费者之间**解耦**.

> 比如过年一家人一起包饺子. 一般都是有明确分工, 比如一个人负责擀饺子皮, 其他人负责包. 擀饺子皮的人就是 "生产者", 包饺子的人就是 "消费者".
>
> 擀饺子皮的人不关心包饺子的人是谁(能包就行, 无论是手工包, 借助工具, 还是机器包), 包饺子的人也不关心擀饺子皮的人是谁(有饺子皮就行, 无论是用擀面杖擀的, 还是拿罐头瓶擀, 还是直接从超市买的).

标准库中的阻塞队列

在 Java 标准库中内置了阻塞队列. 如果我们需要在一些程序中使用阻塞队列, 直接使用标准库中的即可.



1. 基于链表
2. 基于堆，有优先级的
3. 基于数组

Array这个版本速度更快，前提是知道有多少个元素。如果不知道有多少个元素就用Linked的

对于 `BlockingQueue` 来说， `offer()` 和 `pull()` 不带有阻塞功能， `put()` 和 `take()` 带有阻塞功能

- BlockingQueue 是一个接口. 真正实现的类是 LinkedBlockingQueue.
- put 方法用于阻塞式的入队列, take 用于阻塞式的出队列.

- BlockingQueue 也有 offer, poll, peek 等方法, 但是这些方法不带有阻塞特性.

阻塞队列实现

- 通过 "循环队列" 的方式来实现.
- 使用 synchronized 进行加锁控制.
- put 插入元素的时候, 判定如果队列满了, 就进行 wait. (注意, 要在循环中进行 wait. 被唤醒时不一定队列就不满了, 因为同时可能是唤醒了多个线程).
- take 取出元素的时候, 判定如果队列为空, 就进行 wait. (也是循环 wait)

```
class MyBlockingQueue{
    //使用一个String数组来保存数据，假设只存String
    private String[] items=new String[1000];
    volatile private int head=0;//队头
    volatile private int tail=0;//队尾
    //队列有效范围 [head，tail)，当head和tail重合时，相当于空队列

    volatile private int size=0;//使用size表示元素个数

    private Object locker=new Object();

    //入队列
    public void put(String elem) throws InterruptedException {
        //加锁
        //此处的写法相当于直接把synchronized写到方法上了
        synchronized (locker){
            while(size>=items.length){//此处的while目的不是为了循环，而是借助循环巧妙的
实现了wait被唤醒之后再次确认条件
                //队列满了
                //return;
                locker.wait();//这里的notify在出队列的方法中
            }
            items[tail++]=elem;
            if(tail>=items.length){
                tail=0;
            }//这整个判断语句的内容 <=> tail=(tail+1)%items.length，这个代码也能起到，当
tail到达末尾就能回到开头
            size++;
            //使用这个notify来唤醒队列空的阻塞情况
            locker.notify();
        }
    }

    //出队列
    public String take() throws InterruptedException {
        //加锁
        synchronized (locker){
            while(size==0){//此处的while目的不是为了循环，而是借助循环巧妙的实现了wait被唤
醒之后，再次确认条件
                //队列为空，暂时不能出队列
                //return null;
                locker.wait();//这里的notify在入队列的方法中
            }
```

```
            String elem=items[head++];
            if(head>=items.length){
                head=0;
            }
            size--;
            //使用这个notify来唤醒队列满的阻塞情况
            locker.notify();
            return elem;
        }
    }
}
//测试代码
public static void main(String[] args) throws InterruptedException {
    MyBlockingQueue queue=new MyBlockingQueue();
    //创建两个线程，表示生产者和消费者
    Thread t1=new Thread(()->{
        int count=0;
        while (true){
            try {
                queue.put(count+"");
                System.out.println("生产元素："+count);
                count++;
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    Thread t2=new Thread(()->{
        while (true){
            try {
                String count=queue.take();
                System.out.println("消费元素："+count);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    t1.start();
    t2.start();
}
```

上述代码是没问题的，但很容易写错在一个地方，就是在`put()`的`wait()`的异常处理，这里在idea可以选择抛出异常也可以选择`try-catch`处理，但如果是`try-catch`处理，`wait()`被`interrupt`唤醒后代码往下走进入`catch`，方法不会结束而是继续往下执行，就会强行添加元素然后覆盖元素，这是不应该的。而如果像上面一样抛出异常，出现异常后下面就不会继续执行了，就不会出现覆盖元素了。

定时器

定时器是什么

定时器也是软件开发中的一个重要组件. 类似于一个 "闹钟". 达到一个设定的时间之后, 就执行某个指定好的代码.

> 定时器是一种实际开发中非常常用的组件.
>
> 比如网络通信中, 如果对方 500ms 内没有返回数据, 则断开连接尝试重连.
>
> 比如一个 Map, 希望里面的某个 key 在 3s 之后过期(自动删除).
>
> 类似于这样的场景就需要用到定时器.

标准库中的定时器

- 标准库中提供了一个 Timer 类. Timer 类的核心方法为 schedule .
- schedule 包含两个参数. 第一个参数指定即将要执行的任务代码, 第二个参数指定多长时间之后执行(单位为毫秒).

```
Timer timer = new Timer();
timer.schedule(new TimerTask() {
    @Override
    public void run() {
        System.out.println("hello");
   }
}, 3000);
```

实现定时器

定时器的构成:

- 一个带优先级的阻塞队列

> 为啥要带优先级呢?
>
> 因为阻塞队列中的任务都有各自的执行时刻 (delay). 最先执行的任务一定是 delay 最小的. 使用带优先级的队列就可以高效的把这个 delay 最小的任务找出来.

- 队列中的每个元素是一个 Task 对象.
- Task 中带有一个时间属性, 队首元素就是即将
- 同时有一个 worker 线程一直扫描队首元素, 看队首元素是否需要执行

1. MyTimer 类提供的核心接口为 `schedule`, 用于注册一个任务, 并指定这个任务多长时间后执行.

```
public class MyTimer {
    public void schedule(Runnable command, long after) {
    // TODO
    }
}
```

2. MyTimerTask 类用于描述一个任务. 里面包含一个 Runnable 对象和一个 time(毫秒时间戳)

   这个对象需要放到 优先队列 中. 因此需要实现 `Comparable` 接口.

```
class MyTimerTask implements Comparable<MyTimerTask> {
    //任务啥时候执行
    private long time;
    //任务具体是啥
```

```
    private Runnable runnable;

    public MyTimerTask(Runnable runnable, long delay) {
        //delay是一个相对时间差
        //构造time要根据当前系统时间和delay进行构造
        time = System.currentTimeMillis() + delay;
        this.runnable = runnable;
    }

    public long getTime() {
        return time;
    }

    public Runnable getRunnable() {
        return runnable;
    }

    @Override
    public int compareTo(MyTimerTask o) {
        //把时间小的优先级高，把它放入队首中
        // 怎么记忆，这里是谁减去谁?? 不要记!! 记容易记错~~
        // 随便写一个顺序，然后实验一下就行了.
        return (int)(this.time-o.time);
        //return (int) (o.time - this.time);
    }
}
```

3. MyTimer 实例中, 通过 PriorityBlockingQueue 来组织若干个 MyTimerTask 对象.

   通过 schedule 来往队列中插入一个个 MyTimerTask 对象.

```
class MyTimer {
    private Object locker=new Object();

    //使用优先级队列，来保存N个任务
    private PriorityQueue<MyTimerTask> queue = new PriorityQueue<>();

    //定时器的核心方法，把要执行的任务添加到队列中
    public void schedule(Runnable runnable, long delay) {
        synchronized(locker){
            MyTimerTask task = new MyTimerTask(runnable, delay);
            queue.offer(task);
            //每次来新的任务，都唤醒一下之前的扫描线程
            locker.notify();
        }
    }
}
```

4. MyTimer 类中存在一个 t 线程, 一直不停的扫描队首元素, 看看是否能执行这个任务

> 所谓 "能执行" 指的是该任务设定的时间已经到达了

5. 引入一个 locker 对象, 借助该对象的 `wait / notify` 来解决 `while (true)` 的忙等问题.

```
class MyTimer {
```

```
    private Object locker=new Object();

    // ... 前面的代码不变

    //MyTimer中还需要构造一个“扫描线程”，一方面负责监控队首元素是否到点了，是否应该执行；
    // 一方面任务到点之后，就要调用Runnable的run方法来完成任务
    public MyTimer() {
        //扫描线程
        Thread t = new Thread(() -> {
            while (true) {
                try {
                    synchronized(locker){
                        while (queue.isEmpty()) {
                            //注意如果队列为空就不要取元素
                            //此处使用wait等待更合适，如果使用continue，就会使这个线程
while循环运行的飞快
                            //也会陷入一个高频占用CPU的状态（忙等）
                            locker.wait();
                        }
                        MyTimerTask task = queue.peek();
                        long curTime = System.currentTimeMillis();
                        if (curTime >= task.getTime()) {
                            //假设当前时间14:01，任务时间14:00，就要应该执行
                            //需要执行任务
                            queue.poll();
                            task.getRunnable().run();
                        } else {
                            //让当前扫描线程休眠一下，按照时间差来进行休眠
                            //Thread.sleep(task.getTime() - curTime);可使用sleep就
会有很多弊端，具体看7.28这天的板书
                            locker.wait(task.getTime()-curTime);
                        }
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t.start();
    }
}
```

修改 MyTimer 的 schedule 方法, 每次有新任务到来的时候唤醒一下 t 线程. (因为新插入的任务可能是需要马上执行的).

```
class MyTimer {
    //定时器的核心方法，把要执行的任务添加到队列中
    public void schedule(Runnable runnable, long delay) {
        synchronized(locker){
            MyTimerTask task = new MyTimerTask(runnable, delay);
            queue.offer(task);
            //每次来新的任务，都唤醒一下之前的扫描线程
            locker.notify();
        }
    }
}
```

完整代码：

```
//创建一个类，描述定时器的一个任务
class MyTimerTask implements Comparable<MyTimerTask> {
    //任务啥时候执行
    private long time;
    //任务具体是啥
    private Runnable runnable;

    public MyTimerTask(Runnable runnable, long delay) {
        //delay是一个相对时间差
        //构造time要根据当前系统时间和delay进行构造
        time = System.currentTimeMillis() + delay;
        this.runnable = runnable;
    }

    public long getTime() {
        return time;
    }

    public Runnable getRunnable() {
        return runnable;
    }

    @Override
    public int compareTo(MyTimerTask o) {
        //把时间小的优先级高，把它放入队首中
        // 怎么记忆，这里是谁减去谁?? 不要记!! 记容易记错~~
        // 随便写一个顺序，然后实验一下就行了.
        return (int)(this.time-o.time);
        //return (int) (o.time - this.time);
    }
}

//定时器类本体
class MyTimer {
    private Object locker=new Object();

    //使用优先级队列，来保存N个任务
    private PriorityQueue<MyTimerTask> queue = new PriorityQueue<>();

    //定时器的核心方法，把要执行的任务添加到队列中
```

```
    public void schedule(Runnable runnable, long delay) {
        synchronized(locker){
            MyTimerTask task = new MyTimerTask(runnable, delay);
            queue.offer(task);
            //每次来新的任务，都唤醒一下之前的扫描线程
            locker.notify();
        }
    }

    //MyTimer中还需要构造一个“扫描线程”，一方面负责监控队首元素是否到点了，是否应该执行；
    // 一方面任务到点之后，就要调用Runnable的run方法来完成任务
    public MyTimer() {
        //扫描线程
        Thread t = new Thread(() -> {
            while (true) {
                try {
                    synchronized(locker){
                        while (queue.isEmpty()) {
                            //注意如果队列为空就不要取元素
                            //此处使用wait等待更合适，如果使用continue，就会使这个线程
while循环运行的飞快
                            //也会陷入一个高频占用CPU的状态（忙等）
                            locker.wait();
                        }
                        MyTimerTask task = queue.peek();
                        long curTime = System.currentTimeMillis();
                        if (curTime >= task.getTime()) {
                            //假设当前时间14:01，任务时间14:00，就要应该执行
                            //需要执行任务
                            queue.poll();
                            task.getRunnable().run();
                        } else {
                            //让当前扫描线程休眠一下，按照时间差来进行休眠
                            //Thread.sleep(task.getTime() - curTime);可使用sleep就
会有很多弊端，具体看7.28这天的板书
                            locker.wait(task.getTime()-curTime);
                        }
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t.start();
    }
}
```

线程池

线程池是什么

提前创建好一波线程，后续需要使用线程，就直接从线程池里拿即可

线程池最大的好处就是减少每次启动、销毁线程的损耗。

标准库中的线程池

- 使用 Executors.newFixedThreadPool(10) 能创建出固定包含 10 个线程的线程池.
- 返回值类型为 ExecutorService
- 通过 ExecutorService.submit 可以注册一个任务到线程池中.

```
ExecutorService pool = Executors.newFixedThreadPool(10);
pool.submit(new Runnable() {
    @Override
    public void run() {
        System.out.println("hello");
   }
});
```

Executors 创建线程池的几种方式

- newFixedThreadPool: 创建固定线程数的线程池
- newCachedThreadPool: 创建线程数目动态增长的线程池.
- newSingleThreadExecutor: 创建只包含单个线程的线程池.
- newScheduledThreadPool: 设定 延迟时间后执行命令，或者定期执行命令. 是进阶版的 Timer.

Executors 本质上是 ThreadPoolExecutor 类的封装.

ThreadPoolExecutor 提供了更多的可选参数, 可以进一步细化线程池行为的设定. (后面再介绍)

实现线程池

- 核心操作为 submit, 将任务加入线程池中
- 使用 t 描述一个工作线程. 使用 Runnable 描述一个任务.
- 使用一个 BlockingQueue 组织所有的任务
- 每个 t 线程要做的事情: 不停的从 BlockingQueue 中取任务并执行.
- 指定一下线程池中的最大线程数 maxWorkerCount; 当当前线程数超过这个最大值时, 就不再新增线程了.

```
class MyThreadPool{
    private BlockingDeque<Runnable> queue=new LinkedBlockingDeque<>();

    //通过这个方法把任务添加到线程池中
    public void submit(Runnable runnable) throws InterruptedException {
        queue.put(runnable);
    }

    // n 表示线程池里有几个线程
    //创建一个固定数量的线程池
    public MyThreadPool(int n){
        for (int i = 0; i < n; i++) {
            Thread t=new Thread(()->{
                while(true){
                    try {
```

```
                        //取出任务，并执行
                        Runnable runnable=queue.take();
                        runnable.run();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
            t.start();
        }
    }
}
```

总结-保证线程安全的思路

1. 使用没有共享资源的模型
2. 适用共享资源只读，不写的模型
    1. 不需要写共享资源的模型
    2. 使用不可变对象
3. 直面线程安全（重点）
    1. 保证原子性
    2. 保证顺序性
    3. 保证可见性

对比线程和进程

线程的优点

1. 创建一个新线程的代价要比创建一个新进程小得多
2. 与进程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多
3. 线程占用的资源要比进程少很多
4. 能充分利用多处理器的可并行数量
5. 在等待慢速I/O操作结束的同时，程序可执行其他的计算任务
6. 计算密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现
7. I/O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。

进程与线程的区别

1. 进程是系统进行资源分配和调度的一个独立单位，线程是程序执行的最小单位。
2. 进程有自己的内存地址空间，线程只独享指令流执行的必要资源，如寄存器和栈。
3. 由于同一进程的各线程间共享内存和文件资源，可以不通过内核进行直接通信。
4. 线程的创建、切换及终止效率更高。

常见的锁策略

锁策略就属于是实现锁的人要理解的。

以下指的不是某个具体的锁，而是描述锁的特性，描述的是“一类锁”

乐观锁 vs 悲观锁

乐观锁：预测该场景中，不太会出现锁冲突的情况。（后续做的工作会更少）

悲观锁：预测该场景中，非常容易出现锁冲突。（后续做的工作会更多）

锁冲突：两个线程尝试获取一把锁，一个线程获取成功，另一个线程阻塞等待

锁冲突的概率大还是小，对后续的工作是有一定影响的。

`synchronized` **初始使用乐观锁策略.当发现锁竞争比较频繁的时候，就会自动切换成悲观锁策略**

重量级锁 VS 轻量级锁

重量级锁：加锁开销比较大（花的时间多、系统资源占的多），一个悲观锁很可能是个重量级锁（不绝对）

轻量级锁：加锁开销比较小（花的时间少、系统资源占的少），一个乐观锁很可能是个轻量级锁（不绝对）

`synchronized` **开始时是一个轻量级锁，如果锁冲突严重，就可能变成重量级锁**

悲观乐观 是在加锁之前对锁冲突概率的预测，决定工作的多少

重量轻量 是在加锁之后考量实际的锁的开销

自旋锁 VS 挂起等待锁

自旋锁是轻量级锁的一种典型实现，在用户态下，通过自旋的方式（while循环），实现类似于加锁的效果

这种锁，会消耗一定CPU资源，但是可以做到最快速度拿到锁

挂起等待锁是重量级锁的一种典型实现，通过内核态，借助系统提供的锁机制，当出现锁冲突的时候，会牵扯到内核对于线程的调度，使冲突的线程出现挂起（阻塞等待）

这种锁，消耗的CPU资源是更少的，但也无法保证第一时间拿到锁

`synchronized` **中的轻量级锁策略大概率就是通过自旋锁方式实现的**

读写锁 VS 互斥锁

读写锁，把读操作加锁和写操作加锁分开了（一个事实：多线程同时去读一个变量，不涉及线程安全问题）

- 两个线程都只是读一个数据, 此时并没有线程安全问题. 直接并发的读取即可.
- 两个线程都要写一个数据, 有线程安全问题.
- 一个线程读另外一个线程写, 也有线程安全问题.

读写锁就是把读操作和写操作区分对待. Java 标准库提供了 `ReentrantReadWriteLock` 类, 实现了读写锁

- `ReentrantReadWriteLock.ReadLock` 类表示一个读锁. 这个对象提供了 lock / unlock 方法进行加锁解锁.
- `ReentrantReadWriteLock.WriteLock` 类表示一个写锁. 这个对象也提供了 lock / unlock 方法进行加锁解锁.

其中,

- 读加锁和读加锁之间, 不互斥.
- 写加锁和写加锁之间, 互斥.
- 读加锁和写加锁之间, 互斥.

> 注意, 只要是涉及到 "互斥", 就会产生线程的挂起等待. 一旦线程挂起, 再次被唤醒就不知道隔了多久了.
>
> 因此尽可能减少 "互斥" 的机会, 就是提高效率的重要途径.

公平锁 VS 非公平锁

此处定义：公平锁遵循先来后到

非公平锁：看似概率均等，实际上是不公平的，每个线程阻塞的时间不同

注意：

- **操作系统内部的线程调度就可以视为是随机的. 如果不做任何额外的限制, 锁就是非公平锁. 如果要想实现公平锁, 就需要依赖额外的数据结构, 来记录线程们的先后顺序.**
- **公平锁和非公平锁没有好坏之分, 关键还是看适用场景.**

`synchronized` **是非公平锁**

可重入锁 VS 不可重入锁

如果一个线程，针对一把锁，连续加锁两次，会出现死锁，就是不可重入锁；不会出现死锁，就是可重入锁。即**允许同一个线程多次获取同一把锁。**

Java里只要以Reentrant开头命名的锁都是可重入锁，而且**JDK提供的所有现成的Lock实现类，包括** `synchronized` **关键字锁都是可重入的。**

而 Linux 系统提供的 mutex 是不可重入锁.

可重入锁是锁记录了当前哪个线程持有了锁

死锁

死锁是什么

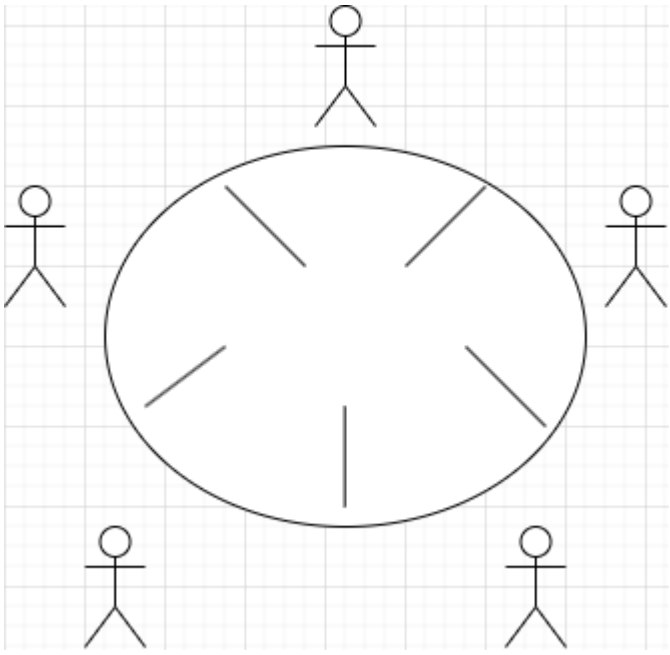
死锁是这样一种情形：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

> **举个例子理解死锁**
>
> 男神和女神一起去饺子馆吃饺子. 吃饺子需要酱油和醋.
>
> 男神抄起了酱油瓶, 女神抄起了醋瓶.
>
> 男神: 你先把醋瓶给我, 我用完了就把酱油瓶给你.
>
> 女神: 你先把酱油瓶给我, 我用完了就把醋瓶给你.
>
> 如果这俩人彼此之间互不相让, 就构成了死锁.
>
> 酱油和醋相当于是两把锁, 这两个人就是两个线程.

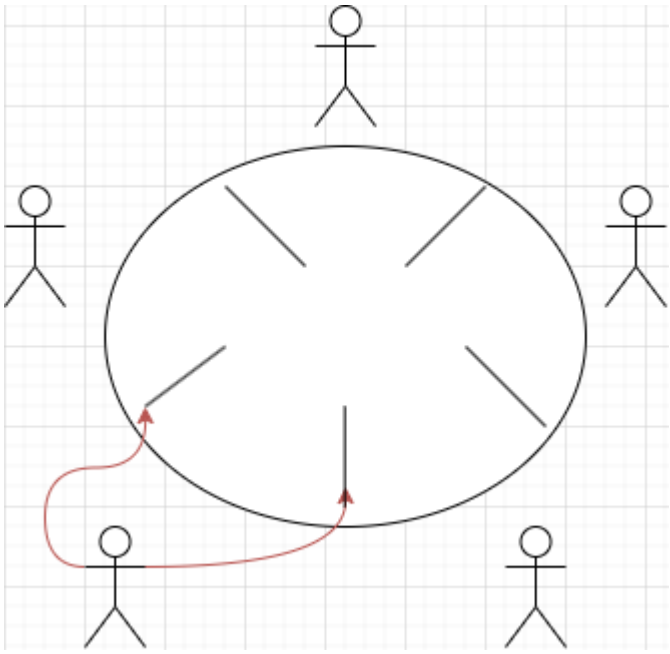
死锁的三种典型情况：

1. 一个线程一把锁，但是是不可重入锁，该线程针对这个锁连续加锁两次，就会出现死锁
2. 两个线程两把锁，这两个线程先分别获取到一把锁，然后再同时尝试获取对方的锁
3. N个线程M把锁，为了进一步阐述死锁的形成, 很多资料上也会谈论到 "哲学家就餐问题".

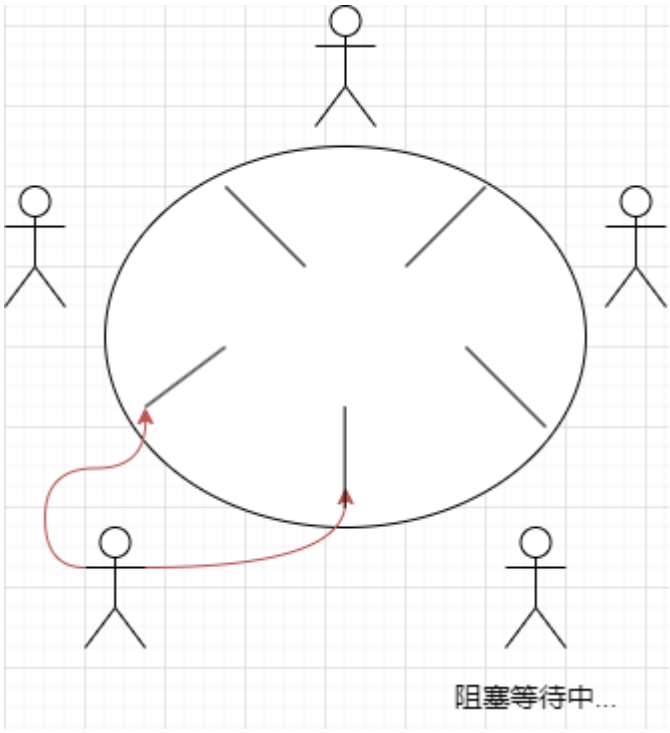
- 有个桌子，围着一圈哲学家，桌子中间放着一盘意大利面，每个哲学家两两之间, 放着一根筷子.



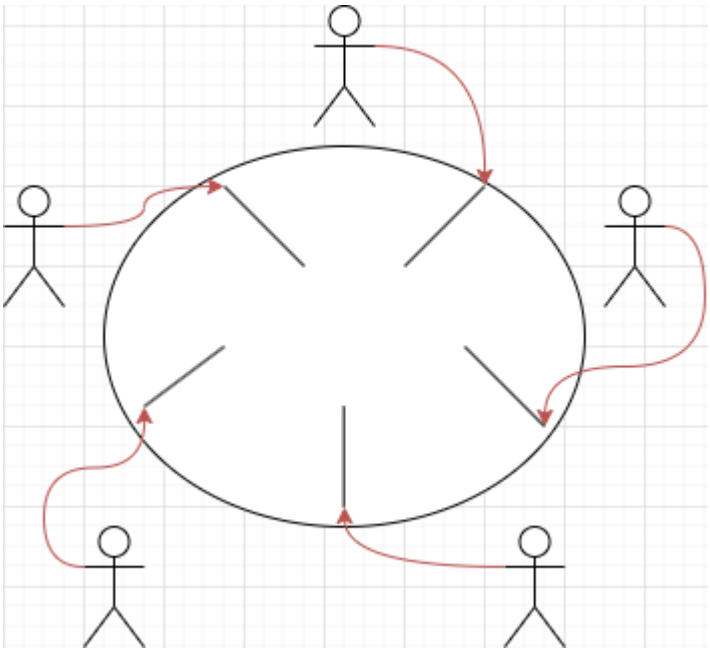
- 哲学家吃面条的时候就会拿起左右两边的筷子(先拿起左边，再拿起右边).



- 如果哲学家发现筷子拿不起来了(被别人占用了)，就会阻塞等待.



- **[关键点在这]** 假设同一时刻, 五个哲学家同时拿起左手边的筷子, 然后再尝试拿右手的筷子, 就会发现右手的筷子都被占用了. 由于哲学家们互不相让, 这个时候就形成了 **死锁**



死锁是严重的BUG！导致一个程序的线程池卡死，无法正常工作

如何避免死锁

- **死锁的四个必要条件**：（缺一不可，能破坏其中任意一个条件，就可以避免出现死锁）

1. 互斥使用，一个线程获取到一把锁以后，别的线程不能获取到这个锁。实际使用的锁，一般都是互斥的（锁的**基本特性**）
2. 不可抢占，锁只能是被持有者主动释放，而不能是被其他线程直接抢走。也是锁的**基本特性**
3. 请求和保持，一个线程去尝试获取多把锁，在获取第二把锁的过程中，会保持对第一把锁的获取状态。取决于代码结构**（很可能影响到需求）**
4. 循环等待，t1尝试获取locker2，需要t2执行完然后释放locker2，t2尝试获取locker1，需要t1执行完然后释放locker1。取决于代码结构**（解决死锁问题的最关键要点）**

当上述四个条件都成立的时候，便形成死锁。当然，死锁的情况下如果打破上述任何一个条件，便可让死锁消失。

如何具体解决死锁问题，实际方法很多（银行家算法，可以解决，但不太接地气）

> 其中最容易破坏的就是 "循环等待".

针对锁进行编号，并且规定加锁的顺序

比如，约定，每个线程如果要获取多把锁必须先获取编号小的锁后获取编号大的锁。只要所有线程加锁的顺序，都严格遵守上述顺序，就一定不会出现循环等待！

`synchronized`具体采用了哪些**锁策略**呢？

1. 既是悲观锁也是乐观锁
2. 既是重量级锁也是轻量级锁
3. 重量级锁部分是基于系统的互斥锁实现的，轻量级锁部分是基于自旋锁实现的
4. 是非公平锁（不会遵循先来后到，锁释放之后，哪个线程拿到锁，各凭本事）
5. 是可重入锁（内部会记录哪个线程拿到了锁，记录引用次数）

6. 不是读写锁

以下是`synchronized`内部实现策略（内部原理）：代码中写了个`synchronized`之后，这里可能产生一系列的“自适应的过程”，锁升级（锁膨胀）

依次从 **无锁->偏向锁->轻量级锁->重量级锁**

1. 偏向锁不是真的加锁而只是做了一个"标记",如果有别的线程来竞争锁了，才会真的加如果没有别的线程竞争，就自始至终都不会真的加锁了。加锁本身，有一定开销。能不加，就不加，非得是有人来竞争了，才会真的加锁~

偏向锁在没有其他人竞争的时候，就仅仅是一个简单的标记（非常轻量），一旦有别的线程尝试进行加锁，就会立即把偏向锁，升级　　成真正的加锁状态，让别人只能阻塞等待

2. 轻量级锁，`sychronized`通过自旋锁的方式来实现轻量级锁。我这边把锁占据了，另一个线程就会按照自旋的方式，来反复查询当前的锁的状态是不是被释放了。但是，后续，如果竞争这把锁的线程越来越多了（锁冲突更激烈了）,从轻量级锁升级成重量级锁
3. 锁消除：编译器会智能的判定，当前这个代码是否需要加锁，如果你写了加锁，但实际上没有必要加锁，就会把加锁操作自动删除掉
4. 锁粗化：关于：“锁的粒度”，如果加锁操作里包含的实际要执行的代码越多，就认为锁的粒度越大

```
for(...){
    synchronized(this){
        count++;
    }
}//锁的粒度小


synchronized(this){
    for(...){
        count++;
    }
}//锁的粒度大
```

有的时候希望锁的粒度小比较好，并发程度更高
有的时候，也希望锁的粒度大比较好(因为加锁解锁本身也有开销)

CAS

什么是CAS

CAS：全称Compare and swap，字面意思:”比较并交换“，能够比较和交换某个**寄存器中的值**和**内存中的值**是否相等，如果相等，则把另一个寄存器中的值和内存进行交换。一个 CAS 涉及到以下操作：

> 我们假设内存中的原数据V，旧的预期值A，需要修改的新值B。
>
> 1. 比较 A 与 V 是否相等。（比较）
> 2. 如果比较相等，将 B 写入 V。（交换）
> 3. 返回操作是否成功。

CAS伪代码

> 下面写的代码不是原子的, 真实的 CAS 是**一个原子的硬件指令完成**的. 这个伪代码只是辅助理解CAS的工作流程.

```
boolean CAS(address, expectValue, swapValue) {
 if (&address == expectedValue) {
   &address = swapValue;
        return true;
   }
    return false;
}
```

由于CAS是通过一条指令完成的，这也就给我们实现线程安全除了加锁以外又多了一条方式——无锁编程。但CAS的使用范围是有局限性的，加锁适用性更广

CAS是怎么实现的

主要是通过CPU硬件支持，软件层面才做得到

CAS有哪些应用

实现原子类

比如，多线程针对一个`count`变量进行++，在Java标准库中已经提供了一组原子类

标准库中提供了`java.util.concurrent.atomic`包, 里面的类都是基于这种方式来实现的。其中的`AtomicInteger`和`AtomicLong`最常用，提供了自增/自减/自增任意值/自减任意值的操作。这些操作就是基于CAS按照无锁编程的方式来实现

```
private static AtomicInteger count = new AtomicInteger(0);
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 50000; i++) {
            count.getAndIncrement();//count++
            /*
            count.incrementAndGet();//++count
            count.getAndDecrement();//count--
            count.decrementAndGet();//--count
            */
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 50000; i++) {
            count.getAndIncrement();
        }
    });
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println(count.get());//获取count的值
}
//100000
```

上述的原子类，没有进行任何的加锁，就是基于CAS来实现的

伪代码实现：

```
class AtomicInteger {
    private int value;
    public int getAndIncrement() {
        int oldValue = value;
        while ( CAS(value, oldValue, oldValue+1) != true) {
            oldValue = value;
       }
        return oldValue;
   }
}
```

> ①这里最开始value=0。此时赋值oldValue=value，oldValue为0。
>
> ②在循环里的CAS含义是value和oldValue比较是否相等，相等则将value=oldValue+1，返回的是false，不进入循环，最终返回oldValue即0，value是1，这就是相当于i++的效果。
>
> ③假设此时有别的线程在①的步骤后穿插进来，则会出现完整的自增流程，此时value=1，oldValue=0，因为还没有进行第二步
>
> ④此时重新回到②中，此时value!=oldValue，则函数返回false，将进入循环，将oldValue赋值为新的value=1，则oldValue为1，重新CAS函数比较，此时value和oldValue相等，则value=oldValue+1=2，返回true，退出循环，最终返回oldValue即1，value是2，这就是相当于i++的效果

当两个线程并发的执行i++操作的时候，如果不加任何限制，意味着这两个i++可能是串行的，则计算正确，也可能是穿插进行的，就会出现问题

实现线程安全：

1. **加锁**保证线程安全：通过锁，强制避免出现穿插
2. **原子类/CAS**保证线程安全：借助CAS来识别当前是否出现“穿插”的情况，如果没穿插，此时直接修改，就是安全的。如果出现穿插了，就重新获取内存中最新的值，再次尝试修改

实现自旋锁

```
public class SpinLock {
    private Thread owner = null;
    public void lock(){
        // 通过 CAS 看当前锁是否被某个线程持有.
        // 如果这个锁已经被别的线程持有, 那么就自旋等待.
        // 如果这个锁没有被别的线程持有, 那么就把 owner 设为当前尝试加锁的线程（Thread.currentThread()）.
        while(!CAS(this.owner, null, Thread.currentThread())){
       }
   }
    public void unlock (){
        this.owner = null;
   }
}
```

> 此处`owner`表示当前是哪个线程持有这把锁。`null`解锁状态

> `Thread.currentThread()`获取当前线程引用。哪个线程调用`lock()`，这里得到的结果就是哪个线程的引用
>
> 当该锁已经处于加锁状态，这里就会返回`false`，CAS就不会进行实际的交换操作，接下来循环条件成立，继续进入下一轮循环。但循环体为空，就又进行这个函数判断，因此一瞬间就执行很多轮次，这样就达到了自旋锁效果。一旦有另一个线程把锁释放了，这里就可以立即得到那个锁

CAS的ABA问题

CAS的关键要点是比较寄存器1和内存的值，通过这里的是否相等，来判定内存的值，是否发生了改变。

如果内存值变了，存在其他线程进行了修改；

如果内存值没变，没有别的线程修改，接下来进行的修改就是安全的。

问题是这里的值没变，就一定没有别的线程修改吗？

A-B-A：另一个线程从A->B，又从B->A，此时本线程区分不了，这个值始终没变，还是出现了变化又回来了的情况。

CAS判定的是“值相同”，实际上我们期望的是"值没有变化过"

解决方案：（约定"值"只能增长不能减小，这个值不是要修改的值，是下面引入的“版本号”）

给要修改的值, 引入版本号. 在 CAS 比较数据当前值和旧值的同时, 也要比较版本号是否符合预期.

- CAS 操作在读取旧值的同时, 也要读取版本号.
- 真正修改的时候,
  - 如果当前版本号和读到的版本号相同, 则修改数据, 并把版本号 + 1.
  - 如果当前版本号高于读到的版本号. 就操作失败(认为数据已经被修改过了).

CAS 也是属于多线程开发中的一种典型的思路

但是咱们在这里并没有介绍 Java 中提供的 CAS 的api 怎么用，也没介绍系统中提供的 CAS api 咋用

实际开发中，一般不会直接使用 CAS，都是用库里已经基于 CAS 装好的组件(像原子类这种)

后面如果大家被问到了原子类咋实现的，自旋锁咋实现的，就可以用CAS解答了

JUC(java.util.concurrent)的常见类

concurrent：并发（多线程）

Callable 接口

`Callable` 是一个 `interface`。也是一种创建多线程的方式，相当于把线程封装了一个 "返回值"。方便程序员借助多线程的方式计算结果。

`Runnable`能表示一个任务（run方法），返回void

`Callable`也能表示一个任务（call方法），返回一个具体的值，类型可以通过泛型参数来指定（Object）

如果进行多线程操作，如果你关心多线程执行的过程，使用`Runnable`，比如线程池，定时器，就是用的`Runnable`只关心过程

如果进行多线程操作，如果你关心多线程的计算结果，使用`Callable`，比如通过多线程的方式计算一个公式，计算1+2+...+1000

代码示例： 创建线程计算 1 + 2 + 3 + ... + 1000

不使用 Callable 版本

- 创建一个类 Result , 包含一个 sum 表示最终结果, lock 表示线程同步使用的锁对象.
- main 方法中先创建 Result 实例, 然后创建一个线程 t. 在线程内部计算 1 + 2 + 3 + ... + 1000.
- 主线程同时使用 wait 等待线程 t 计算结束. (注意, 如果执行到 wait 之前, 线程 t 已经计算完了, 就不必等待了).
- 当线程 t 计算完毕后, 通过 notify 唤醒主线程, 主线程再打印结果.

```
static class Result {
    public int sum = 0;
    public Object lock = new Object();
}
public static void main(String[] args) throws InterruptedException {
    Result result = new Result();
    Thread t = new Thread() {
        @Override
        public void run() {
            int sum = 0;
            for (int i = 1; i <= 1000; i++) {
                sum += i;
           }
            synchronized (result.lock) {
                result.sum = sum;
                result.lock.notify();
           }
       }
   };
    t.start();
    synchronized (result.lock) {
        while (result.sum == 0) {
            result.lock.wait();
       }
        System.out.println(result.sum);
   }
}
```

> 可以看到, 上述代码需要一个辅助类 Result, 还需要使用一系列的加锁和 wait notify 操作, 代码复杂, 容易出错.

使用 Callable 版本

- 创建一个匿名内部类, 实现 Callable 接口. Callable 带有泛型参数. 泛型参数表示返回值的类型.
- 重写 Callable 的 call 方法, 完成累加的过程. 直接通过返回值返回计算结果.
- 把 callable 实例使用 FutureTask 包装一下.

- 创建线程, 线程的构造方法传入 FutureTask . 此时新线程就会执行 FutureTask 内部的 Callable 的 call 方法, 完成计算. 计算结果就放到了 FutureTask 对象中.
- 在主线程中调用 `futureTask.get()` 能够阻塞等待新线程计算完毕. 并获取到 FutureTask 中的结果.

```
Callable<Integer> callable = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= 1000; i++) {
            sum += i;
       }
        return sum;
   }
};
FutureTask<Integer> futureTask = new FutureTask<>(callable);
Thread t = new Thread(futureTask);//使用Callable不能作为Thread的构造方法参数，是借助FutureTask
t.start();
int result = futureTask.get();//通过FutureTask获取Callable的call方法的结果，get类似join一样，如果call方法没算完会阻塞等待
System.out.println(result);
```

> 可以看到, 使用 Callable 和 FutureTask 之后, 代码简化了很多, 也不必手动写线程同步代码了.

理解Callable

> Callable 和 Runnable 相对, 都是描述一个 "任务". Callable 描述的是带有返回值的任务, Runnable 描述的是不带返回值的任务.
>
> Callable 通常需要搭配 FutureTask 来使用. FutureTask 用来保存 Callable 的返回结果. 因为 Callable 往往是在另一个线程中执行的, 啥时候执行完并不确定.
>
> FutureTask 就可以负责这个等待结果出来的工作.

理解FutureTask

> 想象去吃麻辣烫. 当餐点好后, 后厨就开始做了. 同时前台会给你一张 "小票" . 这个小票就是 FutureTask. 后面我们可以随时凭这张小票去查看自己的这份麻辣烫做出来了没

目前为止学到的创建线程的方式：

1. 直接继承Thread
2. 实现Runnable
3. 使用lambda
4. 使用线程池
5. 使用Callable

其中1、2、5搭配匿名内部类使用

ReentrantLock

可重入互斥锁，和 synchronized 定位类似，都是用来实现互斥效果，保证线程安全

ReentrantLock的用法：

- lock()：加锁，如果获取不到锁就死等
- trylock(超时时间)：加锁，如果获取不到锁，等待一定的时间之后就放弃加锁
- unlock()：解锁

ReentrantLock具有一些特点，是synchronized 不具备的功能：

1. 提供了一个tryLock方法进行加锁：

   对于lock操作，如果加锁不成功，就会阻塞等待（死等）

   对于tryLock，如果加锁失败，直接返回false/也可以设定等待时间

2. ReentrantLock有两种模式，可以在工作在公平锁状态下，也可以工作在非公平锁状态下。构造方法中通过参数设定的公平/非公平模式
3. ReentrantLock也有等待通知机制，搭配Condition这样的类来完成，这里的等待通知要比wait、notify功能更强
4. 但是ReentrantLock也可能容易遗漏`unlock()`，通常使用`finally`来执行

ReentrantLock 和 synchronized 的区别：

- synchronized 是一个关键字，是 JVM 内部实现的(大概率是基于 C++ 实现)。ReentrantLock 是标准库的一个类，在 JVM 外实现的(基于 Java 实现)。
- synchronized 使用时不需要手动释放锁。ReentrantLock 使用时需要手动释放。使用起来更灵活，但是也容易遗漏 unlock.
- synchronized 在申请锁失败时，会死等. ReentrantLock 可以通过 trylock 的方式等待一段时间就放弃.
- synchronized 是非公平锁。ReentrantLock 默认是非公平锁，可以通过构造方法传入一个 true 开启公平锁模式.。

如何选择使用哪个锁?

- **锁竞争不激烈的时候，使用 synchronized，效率更高，自动释放更方便**（实际开发也是用synchronized多）
- 锁竞争激烈的时候，使用 ReentrantLock，搭配 trylock 更灵活控制加锁的行为，而不是死等
- 如果需要使用公平锁，使用 ReentrantLock

原子类

原子类内部用的是 CAS 实现，所以性能要比加锁实现 i++ 高很多。原子类有以下几个

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicLong
- AtomicReference
- AtomicStampedReference

以 AtomicInteger 举例，常见方法有

```
addAndGet(int delta);   i += delta;
decrementAndGet(); --i;
getAndDecrement(); i--;
incrementAndGet(); ++i;
getAndIncrement(); i++;
```

应用场景：

1. 计数需求

   播放量、点赞量……

2. 统计效果

   统计出现错误的请求数目、统计收到的请求数目（衡量服务器的压力）、统计每个请求的响应时间=>平均响应时间（衡量服务器的运行效率）……

   通过上述统计内容，实现监控服务器，获取/统计/展示/报警

线程池

虽然创建销毁线程比创建销毁进程更轻量，但是在频繁创建销毁线程的时候还是会比较低效.

线程池就是为了解决这个问题。如果某个线程不再使用了，并不是真正把线程释放，而是放到一个 "池子" 中，下次如果需要用到线程就直接从池子中取，不必通过系统来创建了。

ExecutorService 和 Executors

代码示例：

- ExecutorService 表示一个线程池实例.
- Executors 是一个工厂类，能够创建出几种不同风格的线程池.
- ExecutorService 的 submit 方法能够向线程池中提交若干个任务.

```
ExecutorService pool = Executors.newFixedThreadPool(10);
pool.submit(new Runnable() {
    @Override
    public void run() {
        System.out.println("hello");
   }
});
```

Executors 创建线程池的几种方式

- newFixedThreadPool：创建固定线程数的线程池
- newCachedThreadPool：创建线程数目动态增长的线程池.
- newSingleThreadExecutor：创建只包含单个线程的线程池.
- newScheduledThreadPool：设定 延迟时间后执行命令，或者定期执行命令。是进阶版的 Timer.

Executors 本质上是 ThreadPoolExecutor 类的封装.

ThreadPoolExecutor

ThreadPoolExecutor 提供了更多的可选参数，可以进一步细化线程池行为的设定.

ThreadPoolExecutor 的构造方法

信号量 Semaphore

信号量, 用来表示 "可用资源的个数". 本质上就是一个计数器，描述的是当前这个线程，是否“有临界资源可以用”

临界资源：多个线程/进程等并发执行的实体可以公共使用到的资源（多个线程修改同一个资源，这个变量就可以认为是临界资源）

理解信号量

> 可以把信号量想象成是停车场的展示牌: 当前有车位 100 个. 表示有 100 个可用资源.
>
> 当有车开进去的时候, 就相当于申请一个可用资源, 可用车位就 -1 (这个称为信号量的 P 操作)
>
> 当有车开出来的时候, 就相当于释放一个可用资源, 可用车位就 +1 (这个称为信号量的 V 操作)
>
> 如果计数器的值已经为 0 了, 还尝试申请资源, 就会阻塞等待, 直到有其他线程释放资源.
>
> 这个阻塞等待的过程有点像锁是吧。
>
> 锁，本质上就是一个特殊的信号量（里面的数值，非0即1，二元信号量）
>
> 信号量要比锁更广义，不仅可以描述一个资源，还可以描述N个资源。
>
> 但还是锁用的更多一些

Semaphore 的 PV 操作中的加减计数器操作都是原子的, 可以在多线程环境下直接使用.

CountDownLatch

针对特定场景一个组件

同时等待 N 个任务执行结束

> 好像跑步比赛，10个选手依次就位，哨声响才同时出发；所有选手都通过终点，才能公布成绩。

- 构造 CountDownLatch 实例，初始化 10 表示有 10 个任务需要完成.
- 每个任务执行完毕, 都调用 latch.countDown()，在 CountDownLatch 内部的计数器同时自减.
- 主线程中使用 latch.await()，阻塞等待所有任务执行完毕，相当于计数器为 0 了。（这里await的a是all的意思）

集合类

原来的集合类，大部分都不是线程安全的

> Vector，Stack，HashTable，是线程安全的(不建议用)，其他的集合类不是线程安全的
>
> Vector、HashTable是上古时期搞出来的集合类
>
> 加了锁不一定就线程安全了，不加锁也不一定就线程不安全了

多线程环境使用 ArrayList

1. 自己使用同步机制 (synchronized 或者 ReentrantLock)
2. `Collections.synchronizedList(new ArrayList)`

> synchronizedList 是标准库提供的一个基于 synchronized 进行线程同步的 List.
>
> synchronizedList 的关键操作上都带有 synchronized
>
> 相当于让ArrayList像Vector一样使用

3. 使用 CopyOnWriteArrayList

> CopyOnWrite容器即写时复制的容器。
>
> - 当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，
> - 添加完元素之后，再将原容器的引用指向新的容器。
>
> 这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会
>
> 添加任何元素。
>
> 所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。
>
> **优点**:
>
> 在读多写少的场景下, 性能很高, 不需要加锁竞争.
>
> **缺点**:
>
> 1. 占用内存较多.
> 2. 新写的数据不能被第一时间读取到.

多线程环境使用队列

1. ArrayBlockingQueue

> 基于数组实现的阻塞队列

2. LinkedBlockingQueue

> 基于链表实现的阻塞队列

3. PriorityBlockingQueue

> 基于堆实现的带优先级的阻塞队列

4. TransferQueue

> 最多只包含一个元素的阻塞队列

多线程环境使用哈希表

HashMap 本身不是线程安全的

在多线程环境下使用哈希表可以使用:

- Hashtable
- ConcurrentHashMap

1. Hashtable

只是简单的把关键方法加上了 synchronized 关键字

```
public synchronized V put(K key, V value) {
```

```
public synchronized V get(Object key) {
```

这相当于直接针对 Hashtable 对象本身加锁

- 如果多线程访问同一个 Hashtable 就会直接造成锁冲突
- size 属性也是通过 synchronized 来控制同步，也是比较慢的
- 一旦触发扩容, 就由该线程完成整个扩容过程。这个过程会涉及到大量的元素拷贝，效率会非常低

2. ConcurrentHashMap

相比于 Hashtable 做出了一系列的改进和优化. 以 Java1.8 为例

- 读操作没有加锁(但是使用了 volatile 保证从内存读取结果), 只对写操作进行加锁. 加锁的方式仍然是用 synchronized, 但是不是锁整个对象, 而是 "锁桶" (用每个链表的头结点作为锁对象), 大大降低了锁冲突的概率.
- 充分利用 CAS 特性. 比如 size 属性通过 CAS 来更新. 避免出现重量级锁的情况.
- 优化了扩容方式: 化整为零
    - 发现需要扩容的线程, 只需要创建一个新的数组, 同时只搬几个元素过去.
    - 扩容期间, 新老数组同时存在.

- 后续每个来操作 ConcurrentHashMap 的线程, 都会参与搬家的过程. 每个操作负责搬运一小
- 部分元素.
- 搬完最后一个元素再把老数组删掉.
- 这个期间, 插入只往新数组加.
- 这个期间, 查找需要同时查新数组和老数组

文件操作——IO

文件=>在硬盘上存储数据的方式

硬盘用来存储数据，和内存相比，硬盘的存储空间更大，访问速度更慢，成本更低，适合持久化存储

操作系统通过“文件系统”这样的模块来管理硬盘

树型结构组织和目录

不同的文件系统，管理文件的方式都是类似的。通过“目录”-文件，构成了“N叉树”树形结构。（**目录/文件夹**）

文件路径

文件路径：Windows上是用/或者\分割不同的目录，尽量用/

1. 以 盘符 开头的路径，叫做“绝对路径”。从“此电脑”这里出发找文件

   d:/tmp/cat.jpg　　d:\tmp\cat.jpg
2. 以 . 或者 .. 开头的路径，叫做“相对路径”。需要一个“基准文件”或者“工作目录”，从这个基准文件出发找文件

   如果以D:为基准目录　　./tmp/cat.jpg .表示当前所在目录，./其实可以不写

   如果以D:/tmp为基准　　./cat.jpg .表示当前所在目录

   如果以D:/tmp/111为基准　　../cat.jpg ..表示当前目录的上一层目录

   如果以D:/tmp/111/aaa为基准　　../../cat.jpg

同样是一个cat.jpg的文件，站在不同的基准目录上，查找的路径是不相同的

文件系统上存储的文件具体来说可以分为两大类。

1. 文本文件：存储的是字符
2. 二进制文件：二进制的数据

用记事本打开文件，看得懂就是文本文件，看不懂就是二进制文件

像word文档这种里面是包含很多信息的，这种称为“富文本”

后续针对文件的操作，文本和二进制，操作方式是完全不同的

Java中操作文件

Java 中通过 `java.io.File` 类来对一个文件（包括目录）进行抽象的描述。注意，有 File 对象，并不代表真实存在该文件。

文件系统操作

创建文件、删除文件、创建目录

通过File对象来描述一个具体的文件，File对象可以对应到一个真实存在的文件，也可以对应到一个不存在的文件

File概述

属性

修饰符及类型	属性	说明
static String	pathSeparator	依赖于系统的路径分隔符，String 类型的表示
static char	pathSeparator	依赖于系统的路径分隔符，char 类型的表示

构造方法

签名	说明
File(File parent, String child)	根据父目录 + 孩子文件路径，创建一个新的 File 实例
File(String pathname)	根据文件路径创建一个新的 File 实例，路径可以是绝对路径或者相对路径
File(String parent, String child)	根据父目录 + 孩子文件路径，创建一个新的 File 实例，父目录用路径表示

方法

修饰符及返回值类型	方法签名	说明
String	getParent()	返回 File 对象的父目录文件路径
String	getName()	返回 FIle 对象的纯文件名称
String	getPath()	返回 File 对象的文件路径
String	getAbsolutePath()	返回 File 对象的绝对路径
String	getCanonicalPath()	返回 File 对象的修饰过的绝对路径
boolean	exists()	判断 File 对象描述的文件是否真实存在
boolean	isDirectory()	判断 File 对象代表的文件是否是一个目录

修饰符及返回值类型	方法签名	说明
boolean	isFile()	判断 File 对象代表的文件是否是一个普通文件
boolean	createNewFile()	根据 File 对象，自动创建一个空文件。成功创建后返回 true
boolean	delete()	根据 File 对象，删除该文件。成功删除后返回 true
void	deleteOnExit()	根据 File 对象，标注文件将被删除，删除动作会到 JVM 运行结束时才会进行
String[]	list()	返回 File 对象代表的目录下的所有文件名
File[]	listFiles()	返回 File 对象代表的目录下的所有文件，以 File 对象表示
boolean	mkdir()	创建 File 对象代表的目录
boolean	mkdirs()	创建 File 对象代表的目录，如果必要，会创建中间目录
boolean	renameTo(File dest)	进行文件改名，也可以视为我们平时的剪切、粘贴操作
boolean	canRead()	判断用户是否对文件有可读权限
boolean	canWrite()	判断用户是否对文件有可写权限

观察 get 系列的特点和差异

```
//File的使用
public class Demo1 {
    public static void main(String[] args) throws IOException {
        File file=new File("./test.txt");
        System.out.println(file.getParent());
        System.out.println(file.getName());
        System.out.println(file.getPath());
        System.out.println(file.getAbsolutePath());
        System.out.println(file.getCanonicalFile());
    }
}

//运行结果
.
test.txt
.\test.txt
C:\Users\幽琴健\java_Code\giteedemo\system_code\.\test.txt
C:\Users\幽琴健\java_Code\giteedemo\system_code\test.txt
```

普通文件的创建

```
public class Demo2 {
    public static void main(String[] args) throws IOException {
```

```
        File file=new File("./test.txt");

        //创建文件
        file.createNewFile();

        System.out.println(file.exists());
        System.out.println(file.isFile());
        System.out.println(file.isDirectory());
    }
}

//运行结果
true
true
false
```

普通文件的删除

```
//文件删除
public class Demo3 {
    public static void main(String[] args) throws InterruptedException,
IOException {
        File file =new File("./test.txt");
        System.out.println(file.createNewFile());
        //System.out.println(file.delete());

        //是等到程序退出了再删除，不是立即删除；
        file.deleteOnExit();
        System.out.println(file.exists());
        Thread.sleep(5000);
    }
}

//运行结果
true
true
```

观察目录的创建1

```
public class Demo4 {
    public static void main(String[] args) {
        File file=new File("./testDir");// 要求该目录不存在，才能看到相同的现象

        System.out.println(file.mkdir());
        System.out.println(file.isDirectory());
    }
}

//运行结果
true
true
```

观察目录的创建2

```
public class Demo4 {
    public static void main(String[] args) {
        File file=new File("./testDir/111/222/333");// 要求该目录不存在，才能看到相同的现象

        System.out.println(file.mkdirs());
        System.out.println(file.isDirectory());
    }
}

//运行结果
true
true
```

观察文件重命名

```
//文件重命名
public class Demo5 {
    public static void main(String[] args) {
        File file=new File("./test.txt");
        File file2=new File("./test1.txt");
        System.out.println(file.renameTo(file2));
    }
}

//运行结果
true
```

以上文件系统的操作，都是基于File类来完成的。另外还需要文件内容的操作

文件内容的读写——文件流 stream

文件这里的内容本质是来自于硬盘，硬盘又是操作系统管理的。使用某个编程语言操作文件，本质上都是需要调用系统的api

文件内容的操作核心步骤有四个：

1. 打开文件 fopen
2. 关闭文件 fclose
3. 读文件 fread
4. 写文件 fwrite

字节流：InputStream、OutputStream，是操作字节为单位（二进制文件）

字符流：Reader、Writer，是操作字符为单位（文本文件）

Java IO 流是一个比较庞大的体系，涉及到非常多的类。这些不同类，都有各自不同的特性但是总的来说，使用方法都是类似的。

1. 构造方法，打开文件
2. close方法，关闭文件。（可以通过finally或者**try()**的方式去关闭，后者更优雅）

3. 如果衍生自InputStream或者Reader就可以使用read方法来读数据
4. 如果衍生自OutputStream或者Writer就可以使用write方法来写数据

注意：**close()方法这个释放必要的资源这个操作非常重要。**让一个进程打开一个文件是要从系统这里申请一定的资源的（占用进程的PCB里的文件描述符表中的一个表项，这个表是个顺序表，长度有限且不会扩容），如果不释放就会出现“文件资源泄露”，如果一直打开，文件描述符表就会被占满，后续就无法继续打开新的文件了

文本文件也可以用字节流打开，只不过此时你读到的每个字节就不是完整的字符了

InputStream

方法

修饰符及返回值类型	方法签名	说明
int	read()	读取一个字节的数据，返回 -1 代表已经完全读完了
int	read(byte[] b)	最多读取 b.length 字节的数据到 b 中，返回实际读到的数量；-1 代表以及读完了
int	read(byte[] b,int off, int len)	最多读取 len - off 字节的数据到 b 中，放在从 off 开始，返回实际读到的数量；-1 代表以及读完了
void	close()	关闭字节流

说明

InputStream 只是一个抽象类，要使用还需要具体的实现类。关于 InputStream 的实现类有很多，基本可以认为不同的输入设备都可以对应一个 InputStream 类，我们现在只关心从文件中读取，所以使用 **FileInputStream**

FileInputStream概述

构造方法

签名	说明
FileInputStream(File file)	利用 File 构造文件输入流
FileInputStream(String name)	利用文件路径构造文件输入流

```
//Reader使用
public class Demo6 {
    public static void main(String[] args) throws IOException {
//        FileReader 构造方法, 可以填写一个文件路径(绝对路径/相对路径都行), 也可以填写一个构造好的 File 对象
//        Reader reader = new FileReader("d:/test.txt");
//        try {
//            // 中间的代码无论出现啥情况, close 都能保证执行到.
//        } finally {
```

```
//            // 抛出异常，或者 return，close 就都执行不到了~~
//            reader.close();
//        }

        // 上述使用 finally 的方式能解决问题，但是不优雅.
        // 使用 try with resources 是更好的解决方案.
        try(Reader reader=new FileReader("d:/test.txt")){
            while(true){
                char[] buf=new char[1024];
                int n=reader.read(buf);
                if(n==-1){
                    //读到文件末尾了
                    break;
                }
                for(int i=0;i<n;i++){
                    System.out.print(buf[i]+" ");
                }
            }
        }

    }
}
```

```
public class Demo7 {
    public static void main(String[] args) throws IOException {
        try (InputStream inputStream = new FileInputStream("d:/test.txt")) {
            while (true) {
                byte[] buf = new byte[1024];
                int n = inputStream.read(buf);
                if (n == -1) {
                    break;
                }
                for (int i = 0; i < n; i++) {
                    System.out.printf("%x ", buf[i]);
                }
                String s = new String(buf, 0, n, "utf8");
                System.out.println(s);
            }
        }
    }
}
```

利用 Scanner 进行字符读取

上述例子中，我们看到了对字符类型直接使用 InputStream 进行读取是非常麻烦且困难的，所以，我们使用一种我们之前比较熟悉的类来完成该工作，就是 Scanner 类。

构造方法	说明
Scanner(InputStream is, String charset)	使用 charset 字符集进行 is 的扫描读取

```java
public class Demo8 {
    public static void main(String[] args) throws IOException {
        try(InputStream inputStream=new FileInputStream("d:/test.txt")){
            Scanner scanner=new Scanner(inputStream);
            //此时就是从test.txt这个文件中读取数据了
            String s=scanner.next();
            System.out.println(s);
        }
    }
}
```

OutputStream 概述

方法

修饰符及返回值类型	方法签名	说明
void	write(int b)	写入要给字节的数据
void	write(byte[] b)	将 b 这个字符数组中的数据全部写入 os 中
int	write(byte[] b, int off,int len)	将 b 这个字符数组中从 off 开始的数据写入 os 中，一共写 len 个
void	close()	关闭字节流
void	flush()	重要：我们知道 I/O 的速度是很慢的，所以，大多的 OutputStream 为了减少设备操作的次数，在写数据的时候都会将数据先暂时写入内存的一个指定区域里，直到该区域满了或者其他指定条件时才真正将数据写入设备中，这个区域一般称为缓冲区。但造成一个结果，就是我们写的数据，很可能会遗留一部分在缓冲区中。需要在最后或者合适的位置，调用 flush（刷新）操作，将数据刷到设备中。

说明

OutputStream 同样只是一个抽象类，要使用还需要具体的实现类。我们现在还是只关心写入文件中，所以使用 **FileOutputStream**

```java
public class Demo9_1 {
    public static void main(String[] args) throws IOException {
        try(OutputStream outputStream=new FileOutputStream("d:/test.txt")){
            outputStream.write('H');
            outputStream.write('e');
            outputStream.write('l');
            outputStream.write('l');
            outputStream.write('o');
```

```
            //不要忘记了 flsuh()
            outputStream.flush();
        }
    }
}
```

练习

扫描指定目录，并找到名称中包含指定字符的所有普通文件（不包含目录），并且后续询问用户是否要删除该文件

```
public class Demo10 {
    private static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        //1、让用户输入一个目录，后续的查找都是针对这个目录来进行的
        System.out.println("请输入要搜索的根目录：");
        File rootPath = new File(scanner.next());
        //2、再让用户输入要搜索/要删除的关键词
        System.out.println("请输入要删除的关键词：");
        String word = scanner.next();
        //3、判定一下当前要输入的目录是否有效
        if (!rootPath.isDirectory()) {
            System.out.println("您输入的路径不是合法目录");
            return;
        }
        //4、遍历目录，从根目录出发，按照深度优先（递归）的方式进行遍历
        scanDir(rootPath,word);
    }

    public static void scanDir(File curDir, String word) {
        //1、先列出当前目录中都包含哪些内容
        File[] files = curDir.listFiles();
        if (files == null || files.length == 0) {
            //非法目录或者空目录
            return;
        }
        //2、遍历列出的文件，分两种情况讨论
        for (File f : files) {
            //加个日志，方便查看程序执行的过程
            System.out.println(f.getAbsolutePath());

            if (f.isFile()) {
                //3、如果当前文件是普通文件，看看文件名是否包含了word，来决定是否要删除
                dealFile(f, word);
            } else {
                //4、如果当前文件是目录文件，就递归执行scanDir
                scanDir(f, word);
            }
        }
    }
```

```
    private static void dealFile(File f, String word) {
        //1、先判定当前文件名是否包含word
        if (!f.getName().contains(word)) {
            //此时这个文件不包含word关键词，直接跳过
            return;
        }
        //2、包含word就需要询问用户是否需要删除该文件？
        System.out.println("该文件是：" + f.getAbsolutePath() + "，是否确认删除
（Y/N）");
        String choice = scanner.next();
        if (choice.equals("Y") || choice.equals("y")) {
            f.delete();
        }
        //如果是其他值就忽略
    }
}

//运行结果
请输入要搜索的根目录：
d:/tmp
请输入要删除的关键词：
test
d:\tmp\111
d:\tmp\111\aaa
d:\tmp\111\aaa\test.txt
该文件是：d:\tmp\111\aaa\test.txt，是否确认删除（Y/N）
n
d:\tmp\222
d:\tmp\222\bbb
d:\tmp\222\bbb\新建 文本文档.txt
d:\tmp\333
d:\tmp\333\ccc
```

进行普通文件的复制

```
public class Demo11 {
    public static void main(String[] args) throws IOException {
        Scanner scanner = new Scanner(System.in);
        //1、输入路径并且合法性判定
        System.out.println("请输入要复制的源文件路径：");
        String src = scanner.next();
        File srcFile = new File(src);
        if (!srcFile.isFile()) {
            System.out.println("您输入的源文件路径非法");
            return;
        }
        System.out.println("请输入要复制到的目标路径：");
        String dest = scanner.next();
        File destFile = new File(dest);
        //不要求目标文件本身存在. 但是得保证目标文件所在的目录, 得是存在的.
        //假设目标文件写作 d:/tmp/cat2.jpg, 就需要保证 d:/tmp 目录是存在的.
        if (!destFile.getParentFile().isDirectory()) {
            System.out.println("您输入的目标路径非法");
            return;
        }
```

```
        //2、进行复制操作的过程，按照字节流打开
        try (InputStream inputStream = new FileInputStream(srcFile);
             OutputStream outputStream = new FileOutputStream(destFile)) {
            while (true) {
                byte[] buffer = new byte[20480];
                int n = inputStream.read(buffer);
                System.out.println("n=" + n);
                if (n == -1) {
                    System.out.println("读取到EOF，循环结束");
                    break;
                }
                outputStream.write(buffer, 0, n);
            }
        }
    }
}

//运行结果
请输入要复制的源文件路径：
d:/tmp/111/calculator.png
请输入要复制到的目标路径：
d:/tmp/222/calculator2.png
n=20480
n=220
n=-1
读取到EOF，循环结束
```

扫描指定目录，并找到名称或者内容中包含指定字符的所有普通文件（不包含目录）

```
public class Demo12 {
    public static void main(String[] args) throws IOException {
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入要扫描的根目录：");
        String rootPath = scanner.next();
        File rootDir = new File(rootPath);
        if (!rootDir.isDirectory()) {
            System.out.println("您输入的根目录不是非法目录");
            return;
        }

        System.out.println("请输入要找出的文件名中的字符");
        String token = scanner.next();

        List<File> result = new ArrayList<>();
        //因为文件系统是树形结构，所以我们使用深度优先遍历（递归）完成遍历
        scanDirWithContent(rootDir, token, result);
        System.out.println("共找到了符合条件的文件 " + result.size() + " 个，它们分别
是：");

        for (File file : result) {
            System.out.println(file.getCanonicalFile());
        }
    }
```

```
    private static void scanDirWithContent(File rootDir, String token, List<File>
result) throws IOException {
        File[] files = rootDir.listFiles();
        if (files == null || files.length == 0) {
            return;
        }

        for (File file : files) {
            if (file.isDirectory()) {
                scanDirWithContent(file, token, result);
            } else {
                if (isContentContains(file, token)) {
                    result.add(file.getAbsoluteFile());
                }
            }
        }
    }

    //我们全部按照utf-8的字符文件来处理
    private static boolean isContentContains(File file, String token) throws
IOException {
        StringBuilder sb = new StringBuilder();
        try (InputStream inputStream = new FileInputStream(file)) {
            try (Scanner scanner = new Scanner(inputStream, "UTF-8")) {
                while (scanner.hasNextLine()) {
                    sb.append(scanner.nextLine());
                    sb.append("\r\n");
                }
            }
        }
        return sb.indexOf(token) != -1;
    }
}

//运行结果
请输入要扫描的根目录:
d:/tmp
请输入要找出的文件名中的字符
hello
共找到了符合条件的文件 3 个，它们分别是:
D:\tmp\111\aaa\test.txt
D:\tmp\222\bbb\新建 文本文档.txt
D:\tmp\333\ccc\新建 文本文档.txt
```