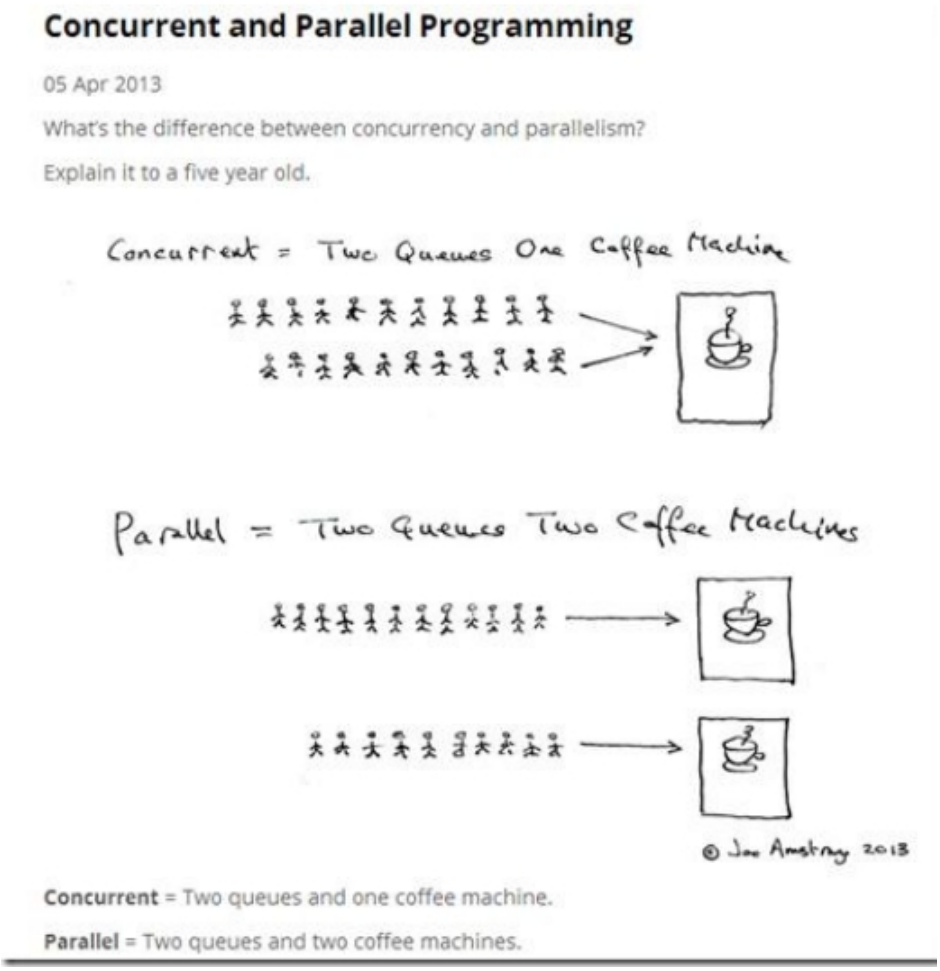
多线程

线程(Thread)的概念

并发与并行

- **并行(parallel)**： 多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的*多任务同时发生*
- **并发(concurrent)**： 只有一台电脑，通过cpu调度算法，让用户看上去多个任务同时执行，实际上从cpu操作层面还是一个个进行
  - 并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈



程序、进程与线程的基本介绍

- 程序：**静态的**，存在硬盘上的可执行指令序列
- 进程：**动态的**，运行在内存中的程序的执行实例
  - 程序的执行过程，进程是一次程序的执行，进程执行任务需要依赖线程，进程的最小执行单位是线程
- 线程：更加“节俭”的多任务操作方式
  - **线程之间共享信息**
  - 线程调度或切换要快得多，进程有自己的内存空间数据段，代码段和堆栈段

- *线程存在于进程，进程基于线程*

进程：单独运行的程序

- 有些程序支持多进程，可以同时多次运行程序，打开多个窗口，互相之间互不影响，如Word
- 有些程序不支持多进程，电脑上同时只能打开一次，如微信

进程与线程

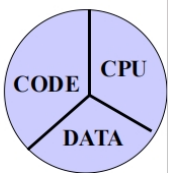
- 资源分配
  - 进程是程序执行的资源分配的基本单位
  - 线程自己一般不拥有资源（必不可少的程序计数器和寄存器外），线程可以访问所属进程的资源，如进程代码段，数据段，及其它系统资源（已打开的文件，IO设备等）
- 系统开销
  - 同一进程的多个线程共享同一地址空间，线程间同步和通信效率高
  - 进程切换，涉及整个当前进程CPU环境和内存状态的保存和新进展环境的设置
  - 线程切换，一般不涉及大量存储器管理方面的操作
- 线程又称为轻量级的进程
  - 一个进程可以创建多个线程，这多个线程就共享进程资源

多线程的基本概念

- 并发现象在现实生活中大量存在
- 许多程序运行时需要“同时”完成多个任务
- Java语言提供了多线程机制，以便支持多任务程序的运行
- 多线程—在一个程序中实现并发
  - 编程语言一般提供了串行程序设计的方法
  - 计算机的并发能力由操作系统提供
  - Java在语言级提供多线程并发的概念

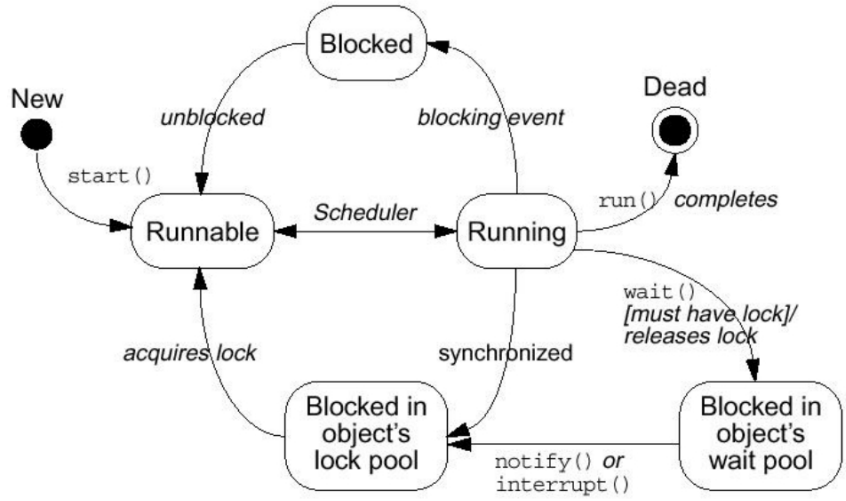
计算机怎么工作

- 简单理解计算机包括：
  - 执行运算的CPU
  - 保存CPU所要执行的程序的ROM
  - 保存程序要操作的数据的RAM
  - 按这种理解，一颗CPU一次只能执行一个任务
- 更全面的认识：现代计算机可以同时完成多个任务，我们这里仅仅从编程的角度讨论如何实现计算机同时运行多个任务
- 执行多个任务就好比有多个计算机，在这里线程(Thread)的执行上下文（Execution Context）包含一个虚拟CPU、程序代码和数据
- 线程包含三个部分：
  - 虚拟CPU
  - CPU执行的程序代码，不依赖于数据，多个线程可共享
  - 程序操作的数据，不依赖于代码，多个线程可共享



多线程的实现

线程的状态转换图



Java线程的两种实现方法

- 继承类`Thread`
- 实现接口`Runnable`

◈ 两种方式可实现多线程：
- ◈ 处理线程任务的类继承java.lang.Thread类
  - Thread类完成虚拟CPU管理
  - 子类只要定义线程需要并行完成的任务代码
  - 重写Thread类的run()方法
- ◈ 处理线程任务的类实现Runnable接口
  - 这种方式更灵活，可避免Java多继承限制
  - 在实现接口的类中，明确定义线程需要并行执行的任务代码
  - 提供run()方法的实现
- ◈ Thread类实际上也实现了接口
  - public class Thread extends Object implements Runnable

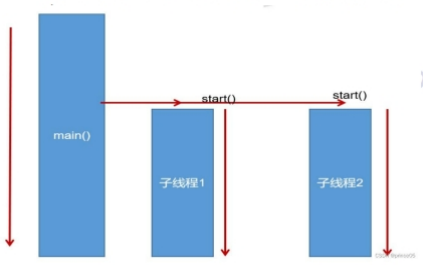
◈ 两种方法都要在run()方法中完成任务的处理代码

继承方式创建线程

◈ 面向对象编程
- ◈ 继承类Thread

◈ 实现接口解决单一继承问题

◈ Thread.currentThread() 显示当前执行线程，尝试代码中显示执行线程




```
public class MyThread extends Thread{
  int i;
  public void run(){ // 重写run方法
    i = 0;
    while(true){
      System.out.println("Hello"+i++);
      if(i == 50) break;
    }
  }
  public static void main(String args[]){
    Thread t1 = new MyThread();
    t1.start();
    Thread t2 = new MyThread();
    t2.start();
  }
}
```

举例：

JAVA

```java
public class Machine extends Thread{
        public void run() {  // 重写线程的`run`方法
                for(int i = 0; i < 50; i++)
                        System.out.println(i);
        }
        public static void main(String[] args) {
                Machine machine = new Machine();
                machine.start();
        }
}
```

Q: `Thread`是什么?

在Java中，`Thread`类代表线程，它是并发编程的基础。并发意味着多个任务可以似乎同时发生。为了实现并发，Java提供了线程这种轻量级的执行单位。每个线程都有其自己的执行路径，可以独立于其他线程运行。这允许程序同时执行多个任务。

Q: `start`方法是什么?

`start`方法是`Thread`类中的一个方法，用于启动新线程。当你调用一个线程的`start`方法时，它会安排该线程的`run`方法在未来的某个时间执行。重要的是要调用`start`方法而不是直接调用`run`方法，因为直接调用`run`方法只会在当前线程的上下文中执行该方法，而不会启动新线程。

通过实现接口创建线程

JAVA

```java
public class TestThread {
        public static void main(String args[]) {
                HelloRunner r = new HelloRunner();
                Thread t = new Thread(r);
                t.start(); // 结果与上面的一样
                /*Thread t1 = new Thread(r);
                Thread t2 = new Thread(r);
                t1.start();
                t2.start();
                */
        }
}
class HelloRunner implements Runnable{
        int i;
        public void run() {
                i = 0;
                while(true) {
                        System.out.println("Hello " + i++);
                        if(i == 50) break;
                }
        }
}
```

Q：如果设置两个进程(即注释内容)，会发生什么？

```
Hello 0
Hello 2
Hello 1
Hello 3
Hello 5
Hello 6
Hello 7
Hello 4
Hello 8
Hello 10
Hello 9
Hello 11
Hello 13
Hello 14
Hello 15
Hello 12
Hello 16
Hello 18
```

发现输出字符无序

> 老师的解释：在执行`t1.start()`的时候，会执行`run()`程序，`i`先被赋值为`0`，然后开始循环，但是循环还没有结束，插入了`t2.start()`程序，又开始了`run()`，那么又会触发`i=0`的赋值，又开始循环。循环后由于数据之间共享，即共享`i`，故后面不会出现`i`值相等情况，但是会因为线程相互插入，使得输出乱序

代码等价于：

JAVA

```java
public class TestThread extends Thread{
        int i;
         public void run() {
                i = 0;
                 while(true) {
                        System.out.println("Hello" + i++);
                        if(i == 50) break;
                 }
         }
         public static void main(String args[]) {
                TestThread t1 = new TestThread();
                TestThread t2 = new TestThread();
                t1.start();
                t2.start();
         }
}
```

展示下线程的运行与转换，再用`try`语句设置`Thread`休眠：

JAVA

```java
System.out.println(Thread.currentThread() + ": Hello " + i++);
try{
        Thread.sleep(10);
}catch(InterruptedException e){}
```

令人惊讶的事情发生了！有时候程序会陷入死循环!

原因：

当你加入 `try-catch` 语句来让线程休眠时，你可能会引入一个**竞态条件**（Race Condition）。竞态条件是多线程编程中的一个常见问题，它发生在当两个或更多的线程同时访问和修改共享数据时。

在你的代码中，`i` 是一个共享变量，两个线程 `t1` 和 `t2` 都在尝试增加它的值。当你不使用 `Thread.sleep(10)` 时，线程可能会以非常快的速度交替执行，使得 `i` 的值迅速增加到 `50`，从而跳出循环。

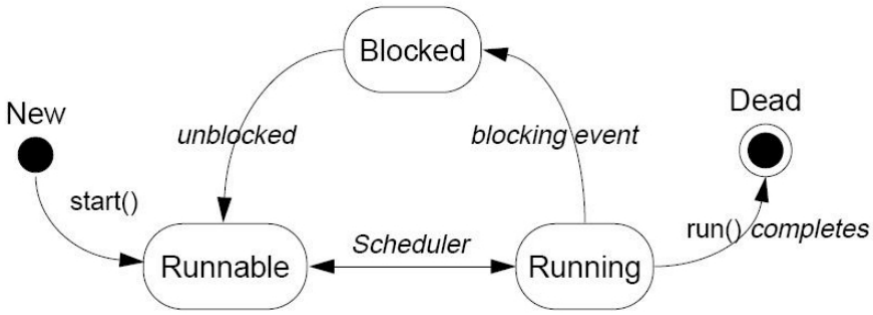
但是，当你加入 `Thread.sleep(10)` 时，事情就变得复杂了。考虑以下情况：

1. 线程 `t1` 执行并增加 `i` 的值。
2. 在 `t1` 休眠的时候，线程 `t2` 执行并增加 `i` 的值。
3. 当 `t1` 醒来并继续执行时，它会再次增加 `i` 的值。
   这意味着每次两个线程交替执行时，`i` 的值**可能会增加两次**，而不是一次。这可能导致 `i` 永远不会到达 `50`，从而使得循环永远不会结束，形成死循环。

为了避免这种情况，你需要确保对共享变量 `i` 的访问是同步的。这可以通过使用 `synchronized` 关键字或其他同步机制来实现。

线程调度

- 创建一个线程后，它并不能自动运行，必须调用start方法
- Start方法将线程的虚拟CPU转入可运行状态，也就是JVM可以安排它的执行，至于是不是立即执行那是JVM的安排

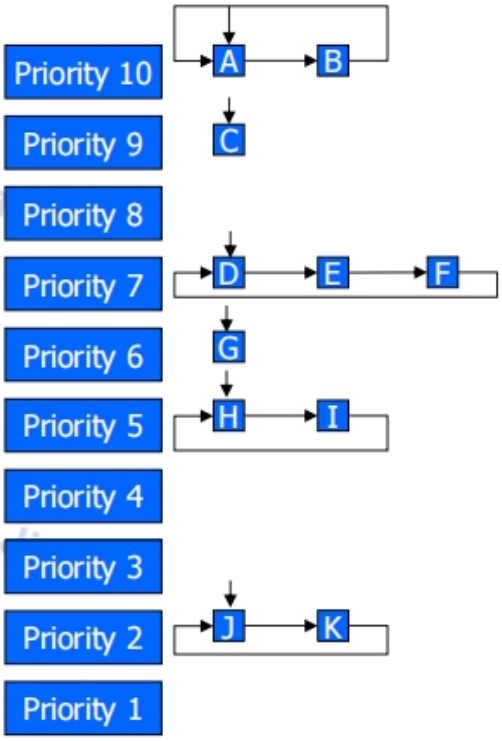


线程阻塞

- Java中线程通常是强占式的
  - 并不一定是按时间分片的，依赖具体系统而言
  - 如Solaris系统线程不强占同优先级的线程；而在Window系统中，分时间片执行，线程可能强占同或高优先级线程
- 可能多个线程在运行，但实际上只有一个在运行，这个线程一直运行到停止或其它高优先级线程开始运行（低优先级的线程被高优先级的线程强占执行）
- 一个线程有多种原因可能停止执行（线程阻塞）：
  - 线程代码执行了Thread.sleep()调用，故意停止固定周期的时间，线程可能在等待某个资源可用才能继续执行
- 所有运行中的线程根据优先级保存在一个池（pools）中，当一个阻塞线程重新运行时，又回到运行池中
  - CPU时间分配给池中非空的最高优先级的线程

◈ 线程如果遇到如下状态，暂时离开执行状态：

- ◈线程结束
- ◈线程处于I/O等待
- ◈线程调用sleep
- ◈线程调用wait
- ◈线程调用join
- ◈线程调用yield
- ◈有更高优先级的线程
- ◈时间片用完



如何终止线程?

◈ 当一个线程终止后停止执行时，不可以再次运行

- ◈ 可以使用一个标记来指示线程的run方法退出，停止线程
- ◈ 线程有一个方法stop()，可以强制线程终止，但这个方法已经过时不推荐使用（deprecated）

```
public class MyThread extends Thread{
        private boolean stop; //tag variable
        public void setStop(boolean stop){
            this.stop = stop;//set tag value
        }
        public viod run{
                while(!stop){…}
        }
}
```

线程控制

- 使用isAlive()检测线程是否已经启动。
- 使用setPriority()和getPriority()可以设置和返回线程优先级
  - 线程优先级是一个整数，Thread类包含有下面三个常数：
  - Thread.MIN_PRIORITY(1)
  - Thread.NORM_PRIORITY(5)
  - Thread.MAX_PRIORITY(10)
- 线程阻塞（Blocked）：
  - Thread.sleep()方法可以暂停线程的执行，好象指令的执行很慢一样，当暂停时间一到恢复运行，似乎什么也没有发生。
  - join()方法让当前线程等待所调用的线程终止运行
  - join()方法如果带一个long参数指定最长等待时间
  - Thread.yield()方法能够让当前线程让出执行机会给同级的其它可运行线程但不会让给低优先级的线程
  - sleep方法可以给低优先级线程执行的机会，而yield不会给低优先级线程机会

JAVA

```java
public class Machine extends Thread{
        public void run(){
                for(int a=0;a<50;a++)
                System.out.println(getName()+":"+a);
        }
        public static void main(String args[])throws Exception{
                Machine machine=new Machine();
                machine.setName("m1");
                machine.start();
                System.out.println("main:join machine");
                machine.join(); //主线程等待machine线程运行结束
                System.out.println("main:end");
        }
}
```

后台线程(感觉不重要)

- 后台线程是指为其他线程提供服务的线程，也称为守护线程
- 后台线程与前台线程相伴相随，只有当所有前台线程结束生命周期，还在运行的后台线程才会被Java虚拟机终止生命周期
- 只要有一个前台线程还没有运行结束，运行中的后台线程就不会被Java虚拟机终止生命周期
- 主线程默认情况下是前台线程，由前台线程创建的线程默认情况下也是前台线程
- 调用Thread类的setDaemon(true)方法，就能把一个线程设置为后台线程
- Thread类的isDaemon()方法判断一个线程是否是后台线程

定时器

在JDK的`java.util`包中

- 提供了一个实用类`Timer`，它能够*定时执行特定的任务*
- 提供一个实用类`TimerTask`类，表示*定时器执行的一项任务*

- `Timer`类的`schedule(TimerTasktask, long delay, long period)`方法可用来*设置定时器需要定时执行的任务*
  - `task`参数表示任务
  - `delay`参数表示延迟执行的时间，以毫秒为单位
  - `period`参数表示每次执行任务的间隔时间，以毫秒为单位

JAVA

```java
import java.util.Timer;
import java.util.TimerTask;
public class MachineThread extends Thread{
   private int a;

   public void start(){
      super.start();
      Timer timer=new Timer(true) ; //后台线程
      TimerTask task=new TimerTask(){
         public void run(){ a=0; }
      };
      timer.schedule(task,2,5);
   }
   public void run(){
      for(int i=0;i<1000;i++){
         System.out.println(getName()+":"+a++);
         Thread.yield();
      }
   }
   public static void main(String args[]) throws Exception{
        MachineThread machine=new MachineThread();
        machine.start();
   }
}
```

```
Thread-0:0
Thread-0:1
Thread-0:2
Thread-0:3
Thread-0:4
Thread-0:5
Thread-0:6
Thread-0:7
Thread-0:8
Thread-0:9
Thread-0:10
Thread-0:11
Thread-0:12
Thread-0:13
Thread-0:14
```

线程引发的问题

并发问题

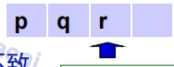
- 线程间总会或多或少共享数据
- 无法保证线程间的执行顺序，哪个线程先于其他线程抢到CPU
- Java使用同步机制(`synchronized`)解决并发问题
- 如果整个方法需要保护，可以将`synchronized`放到方法头
  - 同一时间内只能一个线程访问共享数据

- 其它线程必须等访问数据的线程操作完后才能继续访问共享数据

MyStack

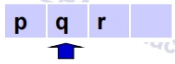
◈ 线程1和2使用同一MyStack对象，共享idx和data：

◈ 线程1刚刚将一个字符存入，还没有增加下标。这时，其它线程强占执行，此时对象不连贯（idx=2）



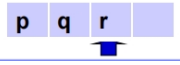
◈ 如果线程1立即恢复执行，将idx增加还不致错误，如果此时线程2进入执行，移出一个字符

◈ 此时的pop方法将先将idx下标减少，然后返回字符q。这样就好象字符r不存在（idx=1）



◈ 等到线程1恢复执行时，idx增加到2

◈ 如果有个线程再次执行pop方法，返回的还将是q




```
public class MyStack{
  int idx = 0;
  char[] data =new char[6];

  public void push(char c){
    data[idx] = c;
    idx++;
  }

  public char pop(){
    idx --;
    return(data[idx]);
  }
}
```

Synchronize关键字

◈ Java中特别提供了一个访问共享数据的机制

◈ 在Java中，每个对象都有一个标记，可以看作一把锁

◈ 使用关键字synchronized可以与这个标记进行交互，并可以排它执行要访问共享数据的代码。

◈ 当程序执行到synchronized语句，检查参数对象，并在继续执行之前获得对象的锁标记

```
public class MyStack{
  …
  public void push(char c){
    synchronized(this){
      data[idx] = c;
      idx++;
    }
  }
  …
}
```

◈ push()中使用关键字synchronized

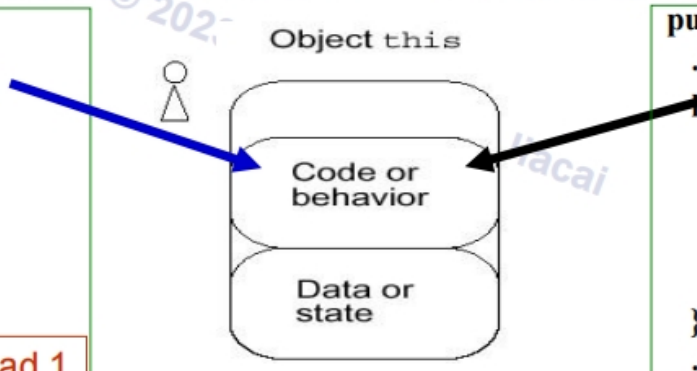
- 我们仅仅保证多个线程中，只有一个线程可以执行对象的push方法中的代码
- 但其它线程还可以使用pop方法，潜在危险没有排除
- 所以必须同时在pop方法中也使用关键字synchronized

Lock Flag

- 线程2执行到对象的pop方法时，然而线程1获得了对象的Lock Flag
  - 当线程2执行synchrozied(this)语句时，尽可能获得对象的Lock Flag，
  - 如果没有得到钥匙，它无法继续，只能到相关联对象的Lock Flag的池中（Lock Pool）等待
  - 当Lock Flag回到对象时，Lock Pool池中得到Lock Flag的线程继续
- 持有Lock Flag的线程必须释放Lock Flag
  - 同步代码块结束时，线程自动释放
  - 线程在同步代码内部碰到异常或break、retutn跳出同步代码块之前自动释放

```
public class MyStack{
  …
  public void push(char c){
    synchronized(this){
      data[idx] = c;
      idx++;
    }
  }
  …
}
```

Thread 1




```
public class MyStack{
  …
  public void pop(){
    synchronized(this){
      idx --;
      return(data[idx]);
    }
  }
  …
}
```

Thread 2

敏感数据的保护

- 只有当对象的敏感数据的访问全部放在同步代码块中时，同步才有意义
  - 所以对象的对象的敏感数据必须声明为private
- 当一个方法的全部代码都放在synchronized代码块中时，可以将关键字synchronized放到方法头中，比如下面两种方法。
  - 当在方法声明中使用synchronized时，要等整个方法执行完成后才释放Lock Flag，会不必要地过长持有Lock Flag

```
public synchronized char pop(){
  …
}
```

```
public void push(char c){
  synchronized(this){
    …
  }
}
```

并发中的内存模型

- CPU执行指令过程中，势必涉及到数据的读取和写入
  - 由于CPU执行速度很快，而从内存读取/写入数据跟CPU执行指令的速度比起来要慢的多
  - 因此在CPU里面就有了高速缓存（每个线程有虚拟CPU，也就有了自己的高速缓存）
  - 会将运算需要的数据从主存复制一份到CPU的高速缓存当中，CPU进行计算时就可以直接使用高速缓存来读取/写入数据
  - 当运算结束之后，再将高速缓存中的数据刷新到主存当中
- i = i + 1
  - 先从主存当中读取i的值，复制到高速缓存
  - 然后CPU对i进行加1操作，并将加1结果写入高速缓存
  - 最后将高速缓存值刷新到主存
  - 单线程中运行没有任何问题的，但是在多线程中运行就会有问题了

- i = i + 1多线程中执行
  - 有2个线程执行这段代码，并共享数据i。
  - 假如初始时i的值为0
  - 可能存在下面一种情况：
    - 初始时，两个线程分别读取i的值存入各自所在的CPU的高速缓存当中
    - 然后线程1进行加1操作，然后把i的最新值1写入到内存，i==1
    - 此时线程2的高速缓存当中i的值还是0，进行加1操作之后，i的值为1，然后线程2把i的值写入内存，i==1
    - 实际中线程1和线程2各加了一次，执行完后应该是i==2

并发编程三个概念

- 原子性：一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。
- 可见性：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。
- 有序性：即程序执行的顺序按照代码的先后顺序执行

volatile保证可见性

```
//线程1
boolean stop = false;
while(!stop){
    doSomething();
}

//线程2
stop = true;
```

```
//线程1
boolean volatile stop = false;
while(!stop){
    doSomething();
}

//线程2
stop = true;
```


- 普通变量
  - 当线程2更改了stop变量的值，还在缓存中，但是还没来得及写入主存当中，线程2转去做其他事情了
  - 线程1由于不知道线程2对stop变量的更改，因此还会继续循环
- 用volatile修饰之后：
  - 使用volatile关键字会强制将修改的值立即写入主存
  - 使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存无效，线程1看见stop值的更新

volatile能保证原子性吗?

不能

JAVA

```java
public class TestVolatile {
    public volatile int inc = 0;

    public void increase() {
        inc++;
    }

    public static void main(String[] args) {
        final TestVolatile test = new TestVolatile();
        for(int i=0;i<10;i++){
            new Thread(){  // 创建匿名内部类
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }

        while(Thread.activeCount()>1)
            Thread.yield();
        System.out.println(test.inc);
    }
}
```

发现代码始终不能按照我们想的到达 10000，但是如果在`increase()`方法中增加`synchronized`修饰符则成功实现

总结：

- 自增操作不具备原子性的，假如某个时刻变量`inc = 10`

1. 假如**线程1**进行*自增*操作：**线程1**先读取了变量inc的原始值，然后线程1被阻塞了；
2. 然后**线程2**进行*自增*操作，**线程2**也去读取变量inc的原始值，线程1没有对变量进行修改操作，所以线程2会直接去主存读取inc的值10，然后进行加1操作，并把11写入工作内存，最后写入主存
3. 然后**线程1**接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以**线程1**对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存
4. 那么两个线程分别进行了一次自增操作后，inc只增加了1

死锁(deadlock)问题

◈ 计算机程序中可能多个线程竞争访问多个资源，这样就会出现死锁(deadlock)情况。比如线程1等待线程2手中的锁，而线程2又又在等待线程1手中的一把锁。因为两个线程谁也无法继续，这样就造成了死锁。

- ◈ 帐户1：$2,000
- ◈ 帐户2：$3,000
- ◈ 线程1：把$3,000从帐户1转移到帐户2
- ◈ 线程2：把$4,000从帐户2转移到帐户1

```
public synchronized void transfer(account1, account2, ammount){
    …
}
```

Thread1：transfer(account1, account2, 3000);
Thread2：transfer(account2, account1, 4000);

◈ 这里线程1拿着锁，但它又必须线程2拿到锁，调了$4000过来后才能继续。但线程2拿不到锁，无法执行transfer同步方法

- ◈ 帐户1：$2,000
- ◈ 帐户2：$3,000
- ◈ 线程1：transfer(account1, account2, 3000)
- ◈ 线程2：transfer(account2, account1, 4000)

```
public synchronized void transfer(account1, account2, ammount){
    …
}
```

◈ Java并不能自动排除这种情况，只能依赖于程序员自己保证死锁不可能发生

◈ 一个首要规则是：如果要同步访问多个对象，全局规划获取这些锁的顺序，程序一定要严格遵循这个顺序，并依据相反的顺序释放这些锁

示例：

JAVA

```java
package lesson12;

public class SynchBankTest{
        public static final int NACCOUNTS = 10;
        public static final int INITIAL_BALANCE = 10000;

        public static void main(String[] args){
                Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
                int i;
                for (i = 0; i < NACCOUNTS; i++){
                        TransferThread t = new TransferThread(b, i, INITIAL_BALANCE);
                        t.setPriority(Thread.NORM_PRIORITY + i % 2);
                        t.start();
                }
        }
}

class Bank{
           public static final int NTEST = 10000;
           private final int[] accounts;
           private long ntransacts = 0;
           static final int INITIAL_BALANCE = 10000;

           public Bank(int n, int initialBalance){
              accounts = new int[n];
              for (int i = 0; i < accounts.length; i++) accounts[i] = initialBalance;
              ntransacts = 0;
           }

           public synchronized void transfer(int from, int to, int amount)throws
InterruptedException {
              while (accounts[from] < amount)   wait();   // 避免死锁
              accounts[from] -= amount;
              accounts[to] += amount;
              ntransacts++;
              notifyAll();
              if (ntransacts % NTEST == 0) test();  // 测试转账次数到10000次后，总额有没
有改变
           }

           public synchronized void test() {   // 测试总额会不会改变
              int sum = 0;
              for (int i = 0; i < accounts.length; i++)  sum += accounts[i];
              System.out.println("Transactions:" + ntransacts  + " Sum: " + sum + "
and Accounts[0]: " + accounts[0]);
           }

           public int size() { return accounts.length; }

        }
```

```
class TransferThread extends Thread{
        private Bank bank;
        private int fromAccount;
        private int maxAmount;

        public TransferThread(Bank b, int from, int max) {
           bank = b;
           fromAccount = from;
           maxAmount = max;
        }

        public void run(){
           try{
              while (!interrupted()) {
                 int toAccount = (int)(bank.size() * Math.random());
                 int amount = (int)(maxAmount * Math.random());
                 bank.transfer(fromAccount, toAccount, amount);
                 sleep(1);
              }
           }catch(InterruptedException e) {}
        }
}
```

通信问题

产生不同的线程以执行不同的任务，有时候执行同一个任务的不同线程间有某种相关性，这就需要线程间互相通信

- 解决这个问题的方案是：
  - 你和司机需要在有需要的时候要有通信方式
  - 当你下楼时，走到一个睡觉的司机前，叫醒他；然后你可以休息，当到达目的地后，司机通知你。然后司机开始休息等待下一个客人
- Java语言中Object类提供了两个方法用于线程通信：
  - wait()，如果一个线程调用了任务对象x的wait方法，本线程就暂停执行，阻塞在对象x的wait池中，并释放对象的锁
  - notify()，一个对象调用了同一个对象x的notify方法，对象x的wait池中一个线程转入对象的锁池中等待锁钥匙
  - 司机想休息时，“司机”线程调用cab.wait()方法，cab是对象
  - 你要坐车，在“你”的线程中调用cab.notify()方法叫醒司机线程。随后调用cab.wait()方法将“你”线程睡眠
  - 一个线程要调用对象的wait或notify方法必须先得到对象的锁。也就是说，这两个方法必须在这个方法所要调用对象的同步代码块中
  - 我们这里需要语句synchronized(cab)来决定是否允许调用cab.wait()或cab.notify()方法

wait 和 notify

- 当一个线程调用了对象的wait方法后，这个线程自动进入对象的等待池（Wait Pool）中，自动放弃对象的Lock Flag
  - 如果一个线程持有多个对象的Lock Flag，那么调用cab.wait()后，只自动释放对象cab的Lock Flag
  - 调用wait方法两种方式：wait()和wait(long timeOut)
- 调用对象的notify方法时，对象的Wait Pool中的任意一个线程转移到Lock Pool，在这里线程等待锁然后执行
  - notifyAll方法通知对象的Wait Pool中所有线程都去Lock Pool中，只有Lock Pool中，线程能取到锁，然后从调用wait方法的地方继续执行
  - 如果Wait Pool没有线程，那么调用对象的notify方法没有任何效果， notify调用不会被保存起来

- 同步：控制线程的执行顺序可控
- 通信：在两个独立线程中用wait()和notify()，notifyAll()叫醒所有的线程（当多个线程处于等待状态）
- 线程同步必须保证多个线程的公用对象保持一致，同时要避免死锁
  - 比如出租车的例子中，焦点对象是出租车，出租车对象的wait和notify方法被调用
- 如果每个人想坐公交车，那么就应当调用bus.notify。注意
  - 同一个Wait Pool中的所有线程都必须通过这个等待池的控制对象来唤醒
  - 同一个Wait Pool中的线程永远不可能通过不同对象来唤醒。
- 使用wait和notify方法来解决一个经典的生产者-消费者（producer-consumer）问题。

JAVA

```java
public class WaitTest {
        public static void main(String [] args) {
                ThreadB b = new ThreadB(); b.start();
                System.out.println("Total is: " + b.getTotal());
        }
}
class ThreadB extends Thread {
        int total;
        public void run() {
                synchronized(this) {
                        for(int i=0;i<1000;i++) {
                                total += 1;
                                try{Thread.sleep(5);}catch(Exception e){}
                                notify();
                        }
                }
        }
        synchronized public int getTotal() {
                try{ wait(); }catch(InterruptedException e) {}
                return total;
        }
}
```

注意：

◈ 在哪些情况线程会被阻塞？

a)当线程处于运行状态，如果执行了某个对象的wait()方法。

b)当线程处于运行状态，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其它线程占用。

c)线程执行了sleep()方法，或者调用了其他线程的join()方法，或者发出了I/O请求。

d)线程执行了yield()方法。

答案：a b c

课堂小问题(5)

◈ 下面哪些语句放到“以上插入行”，将使程序打印“running”？

a) System.out.println("running");
b) t.start();
c) rt.go();
d) rt.start(1);

```
public class RunTest implements Runnable {
  public static void main(String[] args) {
    RunTest rt = new RunTest();
    Thread t =new Thread(rt);
    //此处插入一行
  }
  public void run() {
    System.out.println("running");
  }
  void go() {
    start(1);
  }
  void start(int i) {}
}
```


答案：a b