第11章 多线程

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 进程与线程
- 线程的创建与执行
- 线程的生命周期
  - 新建、就绪、运行、阻塞、死亡
- 线程的同步
  - synchronized
  - 同步机制
  - 同步代码块、同步方法
- 线程通信
  - Wait(),notify(),notifyAll()
  - 生产者与消费者模型

多进程和多线程

- 现代操作系统是多进程的，可以同时运行多个应用程序
- 很多应用程序工作在多线程环境下，发送和接收文件发生在后台（也就是另一个线程中），当有新的对话要发生时，会开启另一个线程……，这样，所有的处理看起来就像是同时进行，用户始终与应用程序的各部分进行着交互。

多进程

- 操作系统的多任务：在同一个时刻运行多个程序的能力。
  - 这些程序看起来像是同时工作，但对于一个CPU而言，操作系统在某个时间点只能执行一个程序，它将CPU的时间片轮流分配给每一个程序，给用户以并发处理的感觉，因为CPU轮转的速度通常以毫秒或微秒为单位，用户感觉不到这种切换。
- 进程（progress）：操作系统中运行的每一个任务。
  - 当一个程序进入内存运行时，即变成一个进程。

操作系统：多进程

Word	微信	……

线程和多线程

- 线程（thread）：进程的执行单元，线程在进程中是独立的、并发的执行流。
- 当进程被初始化后，主线程就被创建了
- 多线程：进程内创建的多个线程，每个线程互相独立。使得同一个进程可以同时并发处理多个任务。

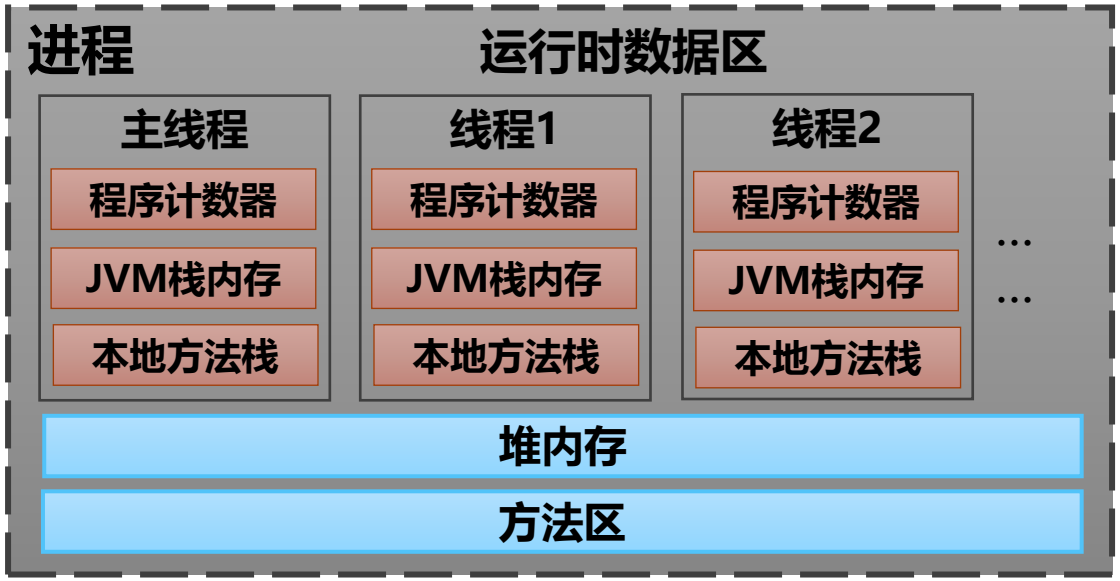
北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

操作系统：多进程

Word		微信		GUI程序
文档1	线程1	聊天	线程1	单线程
文档2	线程2	接收文件	线程2	
文档3	线程3	发送文件	线程3	

进程和线程的关系

- 进程是操作系统进行资源分配和调度的一个独立单位。线程共享进程资源（进程的堆内存区、方法区等）。
- 每个线程拥有自己独立的数据区：程序计数器、栈内存等，创建新线程时这些数据区会一并创建。
  - 程序计数器：记录每个线程执行到的指令位置；
  - 栈内存区：保存线程中每个被调用方法的相关信息。



new关键字 创建的对象

字节码文件，类的各种信息

- 线程的优势

  （1）操作系统创建进程时需要为其分配独立的内存单元及分配大量的相关资源，相比而言，线程的创建简单得多。因此多线程的多任务并发比多进程的效率要高。

  （2）进程之间不能共享内存，线程之间共享内存则非常容易。

  （3）Java语言提供了多线程的支持，不需本地操作系统的直接参与，简化了多线程编程。

- 线程：java.lang.Thread实例
- 创建线程的方式
  - 继承Thread类
  - 实现Runnable接口

（1）定义Thread的子类，并重写该类的run()方法。run()方法的方法体代表了线程需要完成的任务。

（2）创建线程对象。

（3）线程对象调用start()方法启动该线程。

【例11-1】通过Thread类创建线程。

11.2.2 实现Runnable接口创建线程

- Runnable接口只有一个run()方法。
  - 因为Runnable接口和Thread类之间没有继承关系，所以不能直接赋值。
  - 为了使run()方法中的代码在单独的线程中运行，仍需要一个Thread实例，实现对Runnable对象的包装。这样，线程相当于由两部分代码组成：Thread提供线程的支持，Runnable实现类提供线程执行体，即线程任务部分的代码。
  - Thread(Runnable thread)构造方法用于包装Runnable实现类对象，并创建线程。

11.2.2 实现Runnable接口创建线程

【例11-2】通过Runnable接口创建线程。

- 线程由新建、就绪、运行、阻塞、死亡这些状态构成了它的生命周期，呈现了其工作的过程。

11.3.1 新建和就绪状态

- 创建Thread实例：“新建”（new）。
- start()方法启动线程：“就绪”状态（runnable）。

- 处于就绪状态的线程获得了CPU时间片： “运行”状态（running）。
- 操作系统大都采用抢占式的调度策略，即系统给每个可执行的线程一个时间片来处理任务，时间片用完后线程被换下。选择下一个线程时，系统会考虑线程的优先级。

11.3.3 阻塞状态

- “阻塞”状态（blocked）是三种状态的组合体：睡眠/资源阻塞/等待。
- 共同点：线程依然是活的，但当前缺少运行它的条件，即当前是不可运行的，如果发生某个特定的事件，它将返回runnable状态。
- 当前正在执行的线程被阻塞之后，其他线程就可以获得执行的机会。被阻塞线程的阻塞解除后返回runnable状态，必须重新等待线程再次被调度。

11.3.3 阻塞状态

- 线程在运行状态时，遇到如下状况将会进入各种阻塞状态。
- 睡眠：线程调用sleep()方法睡眠一段时间。
- 资源阻塞：线程在等待一种资源，例如线程调用了阻塞式的I/O方法（等待输入流等），在该方法返回之前该线程被阻止；线程试图获得一个同步锁，但同步锁正被其他线程持有（11.5节详述）。
- 等待：线程调用wait()方法后等候其他线程的唤醒通知（11.6节详述）。

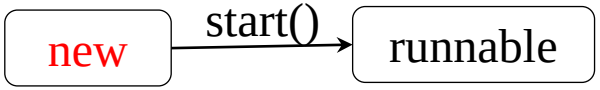
11.3.4 死亡状态

- 线程的run()方法执行完毕，线程正常结束；或者线程执行过程中抛出一个未捕获的异常或错误，线程异常结束。结束后线程处于“死亡”状态（dead）。

线程状态转换图

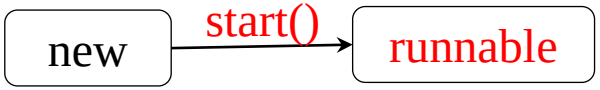

```
public class FirstThread extends Thread{
    //重写run方法
    public void run(){
        for(int i=0; i<5; i++){
            System.out.println(this.getName()+" :  "+(char)(i+'A'));
            for(int j=0; j<400000000; j++);
        }
    }
}
public class SecondThread implements Runnable{
    //重写run方法
    public void run(){
        for(int i=0; i<5; i++){
            System.out.println(Thread.currentThread().getName()
            +" :  "+(char)(i+'A'));
            for(int j=0; j<400000000; j++);
        }
    }
}
```

线程状态转换图



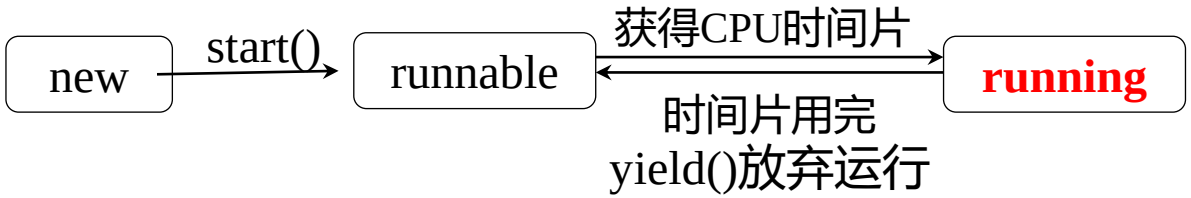

```
public static void main(String[] args) {
    Thread t1 = new FirstThread();


    Runnable target = new SecondThread();
    Thread t2 = new Thread(target);

}
```

线程状态转换图



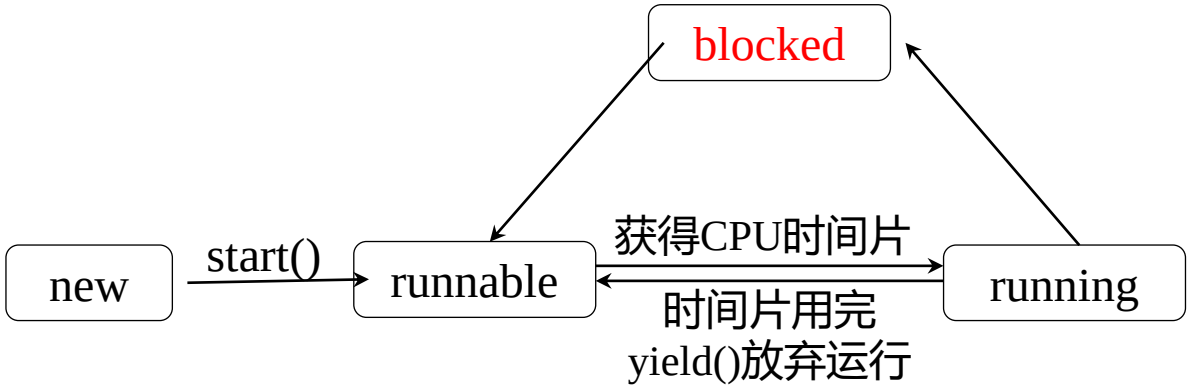

```
public static void main(String[] args) {
    Thread t1 = new FirstThread();
    t1.start();

    Runnable target = new SecondThread();
    Thread t2 = new Thread(target);
    t2.start();
}
```

线程状态转换图



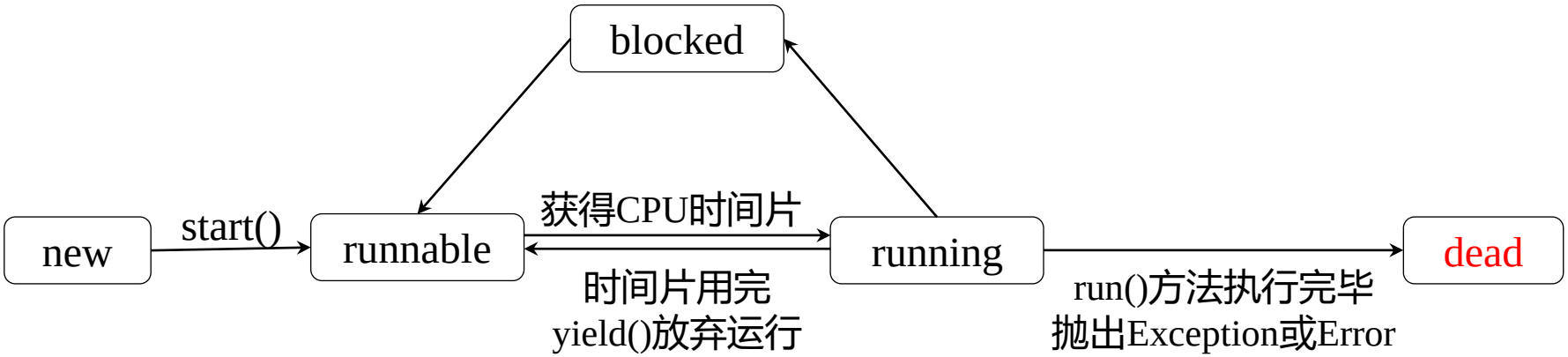
- 如果处于就绪状态的线程获得了CPU时间片，就开始执行run()方法中的线程执行体，线程进入“运行”状态。
- 除非线程的执行体特别短，在一个CPU时间片内就可以执行完毕，否则在运行过程中它将会被中断，以使其他的线程获得执行的机会。线程因失去时间片而中断时返回就绪状态。
- running状态的线程可以调用yield()方法主动放弃执行，从running转入runnable。

线程状态转换图



- “阻塞”状态是三种状态的组合体：睡眠/资源阻塞/等待。这三种状态有一个共同点：线程依然是活的，但当前缺少运行它的条件，即当前是不可运行的，如果发生某个特定的事件，它将返回就绪状态。

线程状态转换图



- 如果线程的run()方法执行完毕，线程正常结束；或者线程执行过程中抛出一个未捕获的异常或错误，线程异常结束。结束后线程处于“死亡”状态。

- 线程调度器：JVM的一部分
  - JVM通常将Java线程直接映射为本地操作系统上的本机线程。
  - 处于runnable状态的线程被放在可运行池中，它们都有资格被调度
  - 线程调度器决定在某个时刻应该运行哪个线程。
  - 决定实际运行哪个线程是线程调度器的职权，我们无法控制线程调度器，不能要求、指定某个线程去被运行。

- 线程调度器：基于优先级的抢先式调度机制。
  - 如果线程进入了runnable状态，而且它比可运行池中的任何线程以及当前运行的线程具有更高的优先级，则具有最高优先级的线程将被选择运行，较低优先级的运行中线程撤回到runnable状态。当池内线程具有相同的优先级，或者当前运行线程与池内线程具有相同优先级时，线程调度器将随意选择它“喜欢”的线程。

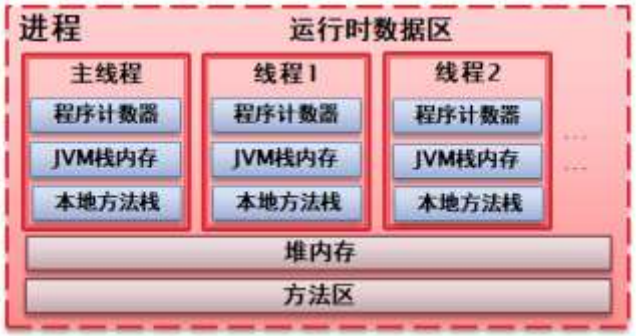
在设计多线程应用程序时不能依赖线程的优先级。线程调度优先级操作是没有保证的，只能将线程优先级作为一种提高程序效率的方法。

11.5.1 数据共享问题

- 【提出问题】夫妻共同进行取钱。
  - ①妻子线程首先执行，检查账户发现账户余额满足取款条件（妻子线程的步骤(1)完成），但在妻子取款之前线程被换下；
  - ②丈夫线程上来后检查账户余额，此时妻子还未取款，丈夫看到的是妻子取款前的账户余额，账户余额也可以满足他的取款要求（丈夫线程的步骤(1)完成），在丈夫取款之前线程被换下；
  - ③妻子线程换上来后接着运行，取走了账户中的全部余额（妻子线程的步骤(2)完成），妻子线程结束；
  - ④丈夫线程换上来后接着运行，因为之前丈夫线程已经确认账户有足够的余额可以支取，于是，丈夫在没有足够余额的情况下仍然进行了取款。

- 当多个线程共享同一个数据时，如果处理不当，很容易出现线程的安全隐患，所以多线程编程时经常需要解决线程同步问题。

```
public static void main(String[] args) {
    Runnable target = new SecondThread();

    Thread t1 = new Thread(target);
    Thread t2 = new Thread(target);

    t1.start();
    t2.start();
}
```



11.5.2 同步和锁机制

- 为了保证共享对象的线程安全，Java使用加锁机制。
  - Java中每个对象都有一个内置锁，当对象具有同步代码时，内置锁启用。
  - Java使用关键字**synchonized**修饰同步代码块或同步方法，为对象加锁。
  - 加锁后的同步代码块或同步方法，形成“原子”操作，即该操作是不可分割的。

```
synchonized (对象) {
        ……
}
```

```
synchonized 方法签名{
        ……
}
```

11.5.2 同步和锁机制

- 线程和锁的关系
  - 锁属于对象，线程持有对象的锁可以运行同步代码；同步代码运行完毕后锁被释放。
  - 因为一个对象只有一个锁，所以当一个线程对对象加锁后，其它试图对同一个对象执行同步代码的线程，都会因获取锁不成功而进入阻塞状态，从而保证了线程安全。

锁不属于线程，而是属于对象，一个线程可以拥有多个对象的锁，而只有同一个对象的锁之间才有互斥的作用。

11.5.3 同步代码块

- 注意：同步锁形成的“原子”操作会损害并发性，所以不要同步原子操作之外的其他代码。

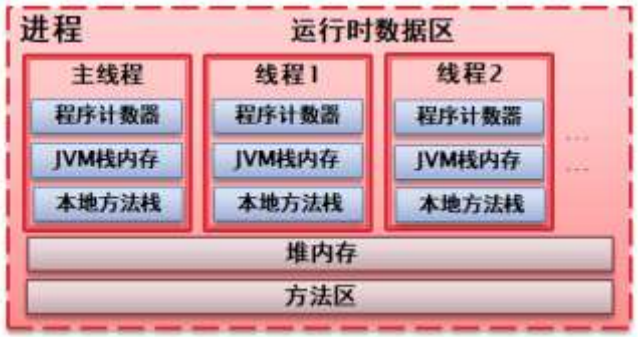
```
synchonized (对象) {
        ……
}
```

【例11-5】设计一个实现Runnable接口的线程类，输出1-5间的数字。其中，计数器变量i作为属性存在。

```
public class MyRunnable implements Runnable{
      private int i=0;   //i作为属性
      public void run(){
              while(i<5){
                      i++;
                      for (int j = 0; j < 20000000; j++);
                      System.out.print(Thread.currentThread().getName()+" ");
                      System.out.println("i="+i);
              }
      }
}
```

【例11-5】设计一个实现Runnable接口的线程类，输出1-5间的数字。其中，计数器变量i作为属性存在。

```
public class Test {
    public static void main(String[] args) {
        Runnable target = new MyRunnable();
        Thread t1 = new Thread(target, "A");
        Thread t2 = new Thread(target, "B");
        t1.start();
        t2.start();
    }
}
```



（1）状况1：i=1为什么没有被输出?

线程A

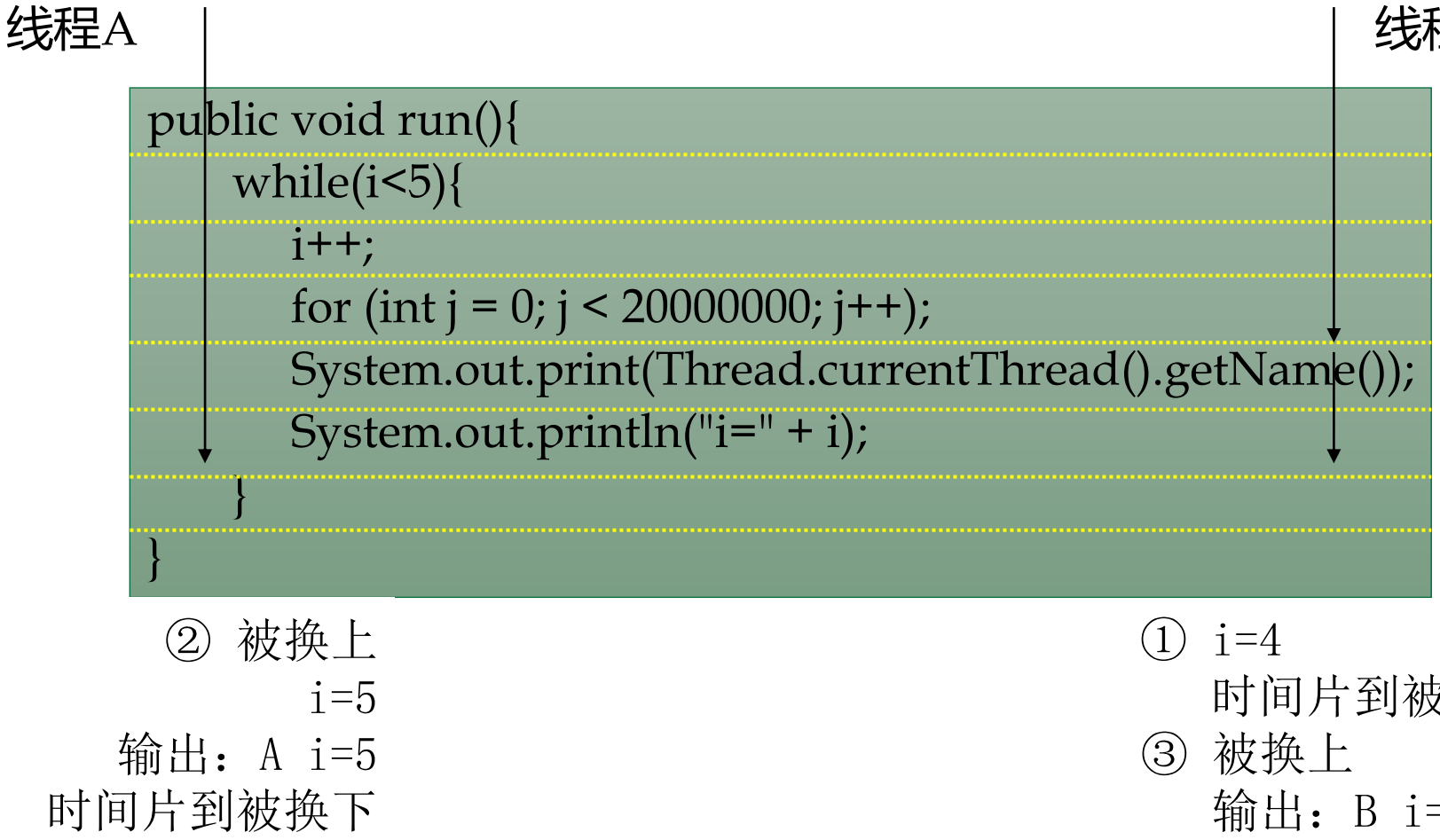
线程B

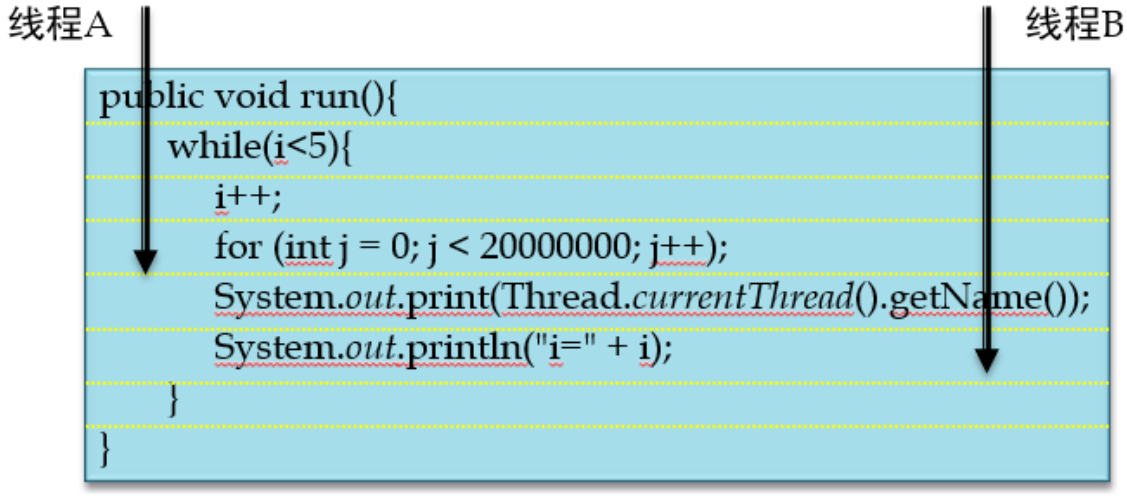
```
public void run(){
    while(i<5){
        i++;
        for (int j = 0; j < 20000000; j++);
        System.out.print(Thread.currentThread().getName());
        System.out.println("i=" + i);
    }
}
```

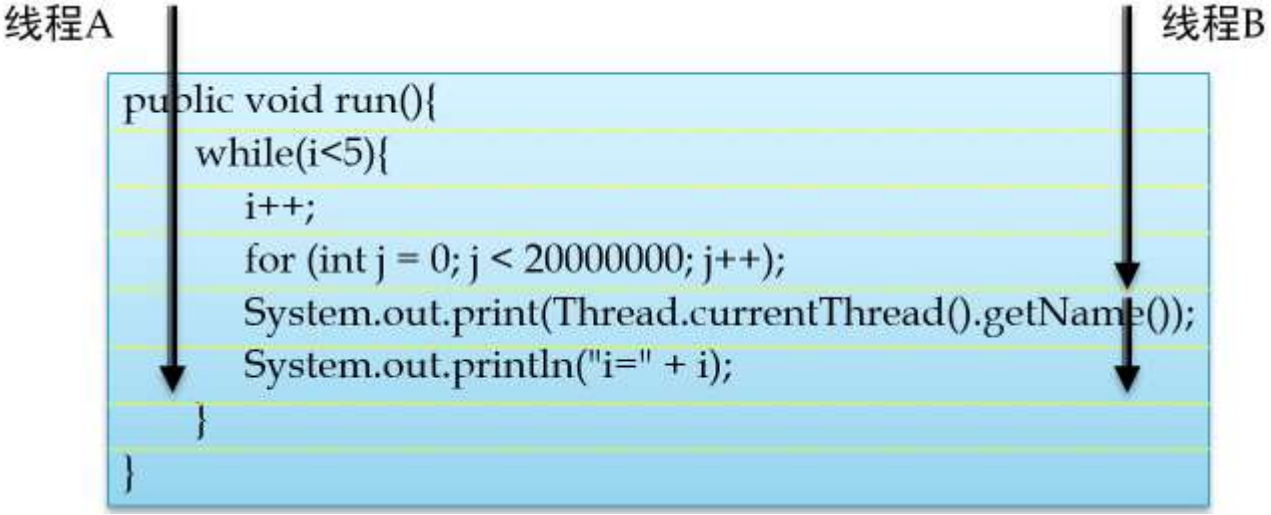
①i=1
时间片到被换下

② 被换上
i=2
输出：B i=2

(2) 状况2：i=5为什么输出了两次?




线程A
线程B
public void run(){
while(i<5){
i++;
for (int j = 0; j < 20000000; j++);
System.out.print(Thread.currentThread().getName());
System.out.println("i=" + i);
}
}


线程A
线程B
public void run(){
while(i<5){
i++;
for (int j = 0; j < 20000000; j++);
System.out.print(Thread.currentThread().getName());
System.out.println("i=" + i);
}
}

同步代码块

```
synchonized (对象) {
        ……
}
```

```
public void run(){
      while(i<5){
          synchronized(this){  //加锁
              i++;
              for(int i=1; i<=10000000;i++);
              System.out.print(Thread.currentThread().getName());
              System.out.println(" i="+i);
          }
      }
}
```

```
public class MyRunnable implements Runnable{
    private int i=0;  //i作为属性
    public void run(){
        while(i<5){
            synchronized(this){
                i++;
                for (int j=0; j<20000000; j++);
                System.out.print(Thread.currentThread().getName()+" ");
                System.out.println("i="+i);
            }
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Runnable target = new MyRunnable();
        Thread t1 = new Thread(target, "A");
        Thread t2 = new Thread(target, "B");
        t1.start();
        t2.start();
    }
}
```

（3）状况3：为什么输出i=6?

线程A

线程B

```
public void run(){
    while(i<5){
        synchronized(this){ //加锁
            i++;
            for(int i=1; i<=10000000;i++);
            System.out.print(Thread.currentThread().getName());
            System.out.println(" i="+i);
        }
    }
}
```

①i=4, 通过i<5
时间片到被换下

③ 被换上
获得锁，进入同步块
i=6
输出 A i=6
……

②被换上
加锁，进入同步块
i=5
输出： B i=5
释放锁
时间片到被换下

最终的代码

```
public class MyRunnable implements Runnable{
    private int i=0;  //i作为属性
    public void run(){
        while(i<5){
            synchronized(this){
                if(i==5) break;
                i++;
                for (int j = 0; j < 20000000; j++);
                System.out.print(Thread.currentThread().getName()+" ");
                System.out.println("i="+i);
            }
        }
    }
}
```

11.5.4 同步方法

- 如果一个方法内的所有代码组成“原子”操作，那么可以将该方法定义为同步方法，使用synchronized关键字修饰。

【例11-6】完成多线程环境下的银行取钱操作。

11.6 线程间的通信

- 在多线程环境中，线程之间经常需要协调通信从而共同完成一件任务。Java传统的线程通信是通过Object类中的wait()和notify()方法完成。
- Java SE5.0中增加了阻塞队列BlockingQueue等方式控制线程通信。

11.6.1 wait()和notify()方法

- Object类中提供了wait()，notify()和notifyAll()3个方法来操作线程。
- 它们只能在同步代码块或者同步方法内使用，而且只能通过进行同步控制的对象（同步监视器）来调用。

```
synchonized (对象) {
        ……
}
```

```
synchonized 方法签名{
        ……
}
```

对于同步方法，该方法所在类的实例this就是同步监视器，所以可以在同步方法中直接调用这3个方法，如this.wait()直接简写为wait()。

11.6.1 wait()和notify()方法

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

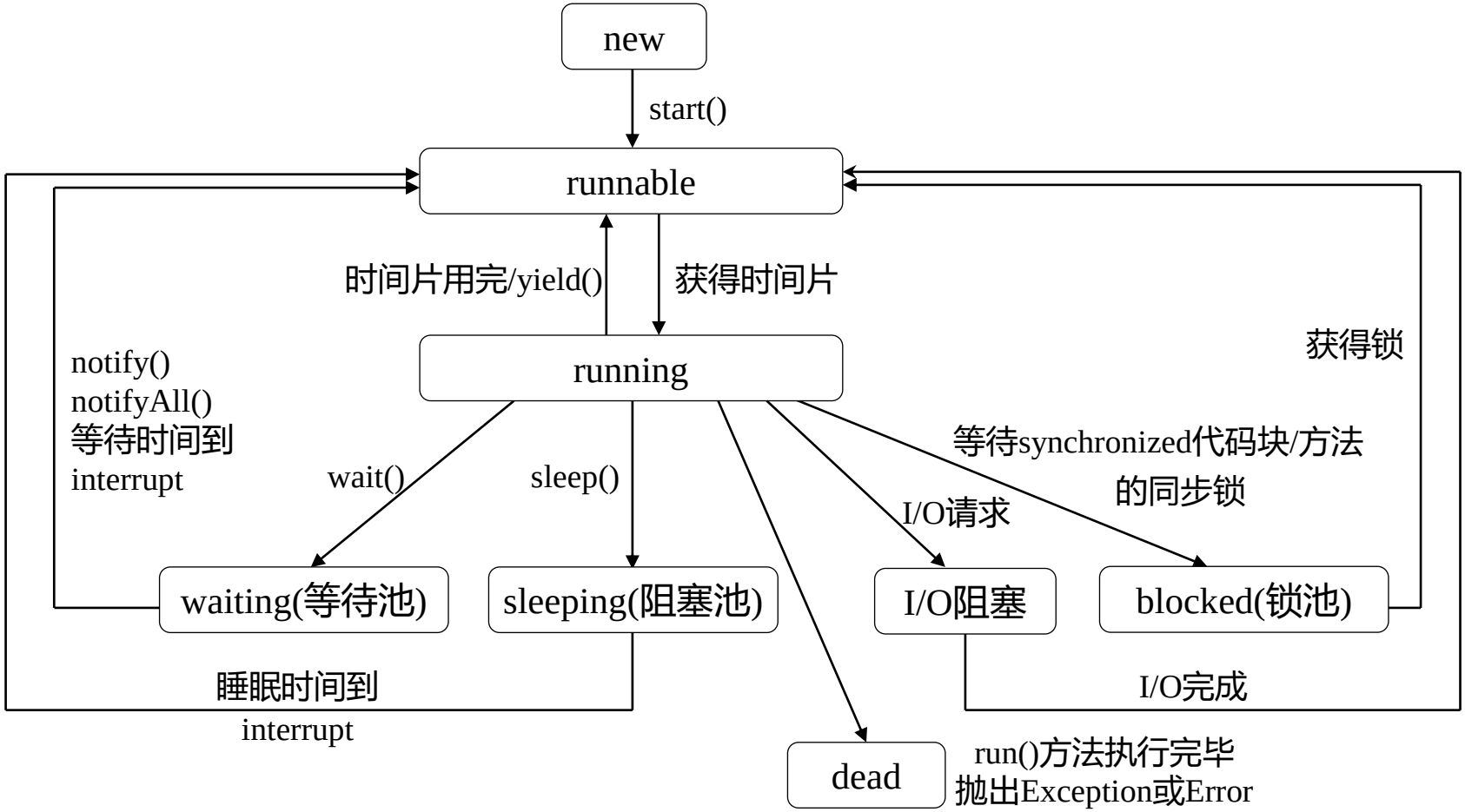
1. wait()方法
   - 在线程已获得对象锁的情形下，如果该线程需要再满足一些条件才能继续执行线程任务，此时该线程可调用wait()方法进入等待池（阻塞状态的一种）。
   - 线程调用wait()方法会解除对象的锁，让出CPU资源，并使该线程处于等待状态，使其它线程可以获取该对象的锁，执行该对象的同步代码块或方法。
     - void wait()：线程会一直等待，直到其他线程调用同步监视器的notify()或notifyAll()方法后苏醒；
     - void wait(long timeout)，wait(long timeout,int nanos)：方法指定了等待时间，所以如果线程在等待时间内没有被同步监视器的notify()方法唤醒，则在等待指定时间后自动苏醒。

2. notify()和notifyAll()方法

- notify()方法唤醒一个处于等待状态的线程，使之进入runnable状态。
- 某个线程执行完同步代码，或该线程使另一个线程所等待的条件得到满足，这时它利用同步监视器调用notify()方法，以唤醒一个因该同步监视器而处于等待状态的线程再次进入runnable状态。从等待状态进入runnable状态的线程，将再次尝试获得同步监视器的锁。
- notifyAll()方法：使因该同步监视器而处于等待状态的全部线程进入runnable状态。

编程时应该用notifyAll()取代notify()，因为用notify()只是从等待池释放出一个线程，至于是哪一个由线程调度器决定，是不可保证的。

线程状态转换图


new
start()
runnable
时间片用完/yield()
获得时间片
running
notify()
notifyAll()
等待时间到
interrupt
wait()
sleep()
等待synchronized代码块/方法
的同步锁
I/O请求
获得锁
waiting(等待池)
sleeping(阻塞池)
I/O阻塞
blocked(锁池)
睡眠时间到
interrupt
I/O完成
dead
run()方法执行完毕
抛出Exception或Error

11.6.1 wait()和notify()方法

【例11-8】线程间的通信。写两个线程，线程A“做”10个披萨，线程B“做”20份意大利面，要求线程A每做一个披萨，就通知线程B去做两份意大利面，线程B完成两份意大利面后通知线程A继续做披萨……。

关键问题：找到两个线程中互相向对方通信的对象。

- 两个线程的wait()和notifyAll()方法需要使用同一个同步监视器。
- 同步监视器应被两个线程共享。
- 每个线程都引入这个对象，利用构造方法对其初始化。

11.6.2 消费者和生产者模型

- 【问题描述】生产者和消费者在同一时间段内共用同一个缓冲区（仓库），生产者向缓冲区存放数据，而消费者从缓冲区取用数据。
- 问题的关键：保证生产者不会在缓冲区满时再加入数据，消费者也不会在缓冲区空时再消耗数据。

本章思维导图


多线程
I. 概念
进程
操作系统分配资源的单位
线程
进程中独立、并发的执行流
多线程
线程间有独立数据区
共享所属进程期内存、方法区
主线程+其他线程
II. 创建和执行线程
继承Thread类
重写Thread类的run()方法
new 线程类名()
start()
实现Runnable接口
实现Runnable接口的run()方法
new Thread(Runnable对象)
start()
III. 线程的生命周期
new
runnable
running
blocked
睡眠
sleep()
I/O方法等待
资源阻塞
锁被占用
等待
wait()
dead
状态转换图11-10
IV. 线程调度
调度策略
抢先式调度
线程优先级
Thread.MIN_PRIORITY(1)
Thread.NORM_PRIORITY(5)
Thread.MAX_PRIORITY(10)
影响优先级的因素
setPriority(int)
修改线程优先级
static yield()
放弃执行
join()
至于其他线程之后
static sleep()
抛异常
interrupt()
InterruptedException
V. 案例
【例11-1】通过Thread类创建线程
线程的创建方式1.执行
【例11-2】通过Runnable接口创建线程
线程的创建方式2.执行
【例11-3】join()对线程调度的影响
【例11-4】打断线程的睡眠
interrupt()
InterruptedException
【例11-5】多线程同步的程序设计
多线程共享代码的问题
synchronized对象代码块
【例11-6】完成多线程环境下的银行取钱问题
多线程共享数据的问题
synchronized方法
【例11-7】多线程环境下的集合类
Collections.synchronizedXxx()
synchronized方法
【例11-8】线程间通信
wait()
notifyAll()
notify()
共享
VI. 线程间通信
传统方式
synchronized代码块/方法内使用
wait()
notifyAll()
notify()
阻塞队列java.util.concurrent.BlockingQueue接口
Java SE 5.0
ArrayBlockingQueue
LinkedBlockingQueue
VII. 线程同步
why
多线程共享数据/代码
how
同步锁
建立"原子性"
synchronized
对象代码块
方法