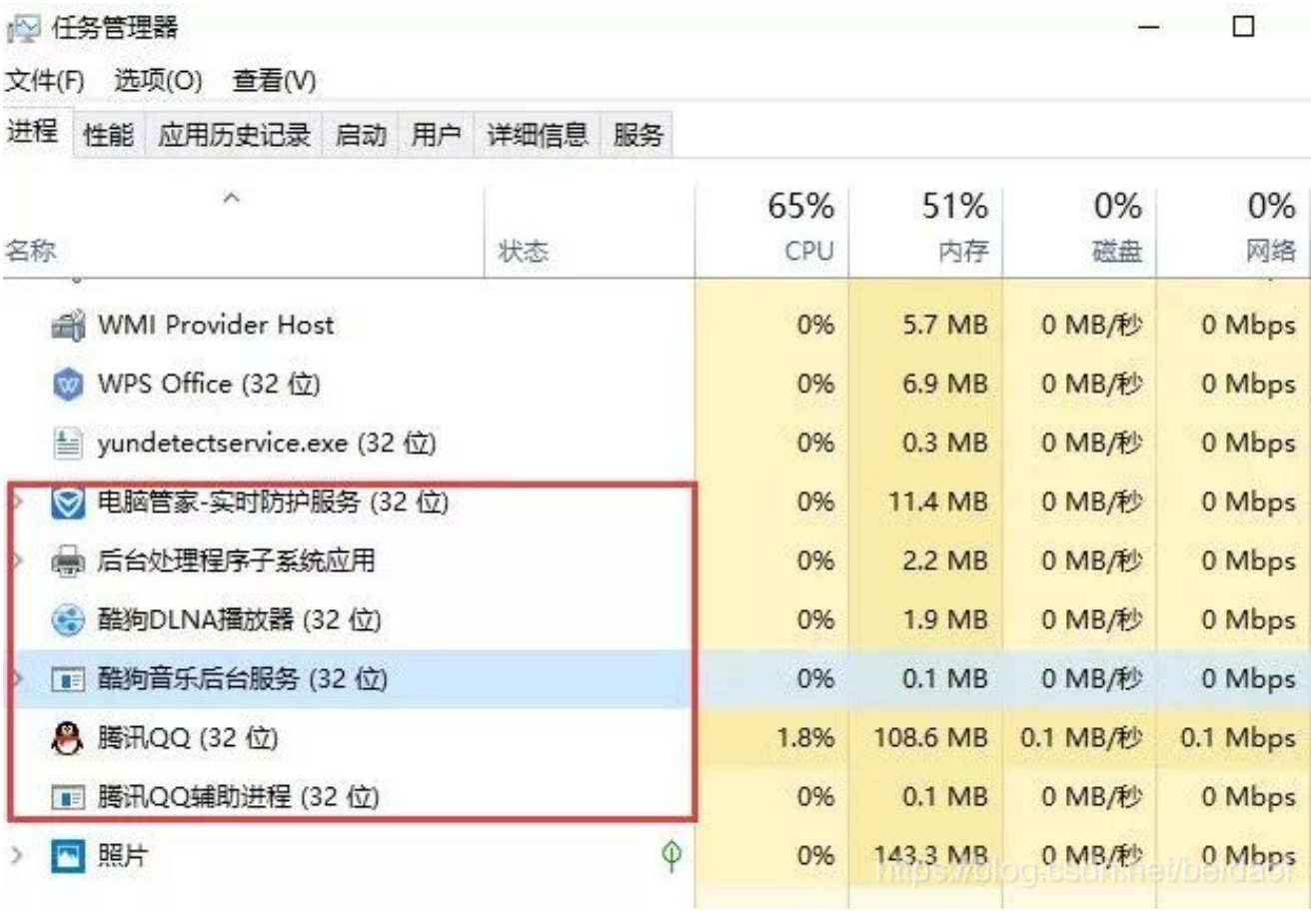
多线程

1.什么是多线程

进程：

电脑中时会有很多单独运行的程序，每个程序有一个独立的进程，而进程之间是相互独立存在的。比如下图中的QQ、酷狗播放器、电脑管家等等。



线程：

进程想要执行任务就需要依赖线程。换句话说，就是进程中的最小执行单位就是线程，并且一个进程中至少有一个线程。

多线程：

多线程就是指我们在处理多个任务时开启多个线程来处理多个任务，让多个任务同时进行

2.为什么需要多线程

> 为了更好的利用CPU的资源，如果只有一个线程，则第二个任务必须等到第一个任务结束后才能进行，如果使用多线程则在主线程执行任务的同时可以执行其他任务，而不需要等待
>
> 进程之间不能共享数据，线程可以
>
> 系统创建进程需要为该进程重新分配系统资源，创建线程代价比较小
>
> Java语言内置了多线程功能支持，简化了java多线程编程

3.开启多线程的三种方式

继承Thread类的方式进行实现

```
package com.thread.threadcase01;

public class ThreadDemo {
    public static void main(String[] args) {

        /**
         * 多线程的第一种启动方式：继承Thread类
         *          *   1. 自己定义一个类继承Thread类
         *          *   2. 重写run()方法，在run()方法中实现线程要执行的任务
         *          *   3. 创建Thread类的对象，并调用start()方法启动线程
         */



        MyThread myThread = new MyThread();
        myThread.setName("myThread"); //设置线程名称
        MyThread myThread2 = new MyThread();
        myThread2.setName("myThread2"); //设置线程名称


        myThread.start(); //启动线程
        myThread2.start(); //启动线程
    }
}
```

继承thread

```
package com.thread.threadcase01;

```

```
public class MyThread extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
    }
}

```

优点：编程简单，可直接使用Thread类的方法。 缺点：可扩展性差，不能再继承其他的类

实现Runnable接口的方式进行实现

```
package com.thread.threadcase02;

public class ThreadDemo {
    public static void main(String[] args) {
        /**
         * 多线程的第二种启动方式:
         * 1. 实现Runnable接口
         * 2. 重写run()方法
         * 3. 创建自己的类的对象
         * 4. 创建一个Thread类的对象，并启动线程
         */

        MyRun myRun = new MyRun();

        Thread thread = new Thread(myRun);
        Thread thread2 = new Thread(myRun);

        thread.setName("线程1");
        thread2.setName("线程2");

        thread.start();
        thread2.start();
    }
}

```

```
public class MyRun implements Runnable {
    @Override
```

```
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
    }
}

```

优点：扩展性强，缺点：编程相对复杂

利用Callable接口和Future接口方式实现

```
package com.thread.threadcase03;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class ThreadDemo {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        /**
         * 多线程的第三种实现方式
         *    特点：可以获得多线程运行的结果
         *
         *    1. 创建一个类MyCallable，实现Callable接口
         *    2. 重写call()方法
         *
         *    3. 创建MyCallable类的实例对象
         *    4. 创建FutureTask的对象(管理多线程的结果)
         *    5. 创建线程对象，并启动
         */

        MyCallable myCallable = new MyCallable();

        FutureTask<Integer> futureTask = new FutureTask<>(myCallable);

        Thread thread = new Thread(futureTask);

        thread.start();

        Integer i = futureTask.get();

        System.out.println(i);
    }
```

```
}
```

```
package com.thread.threadcase03;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {


        int sum = 0;

        for (int i = 1; i <= 100; i++) {
            sum += i;
        }

        return sum;
    }
}

```

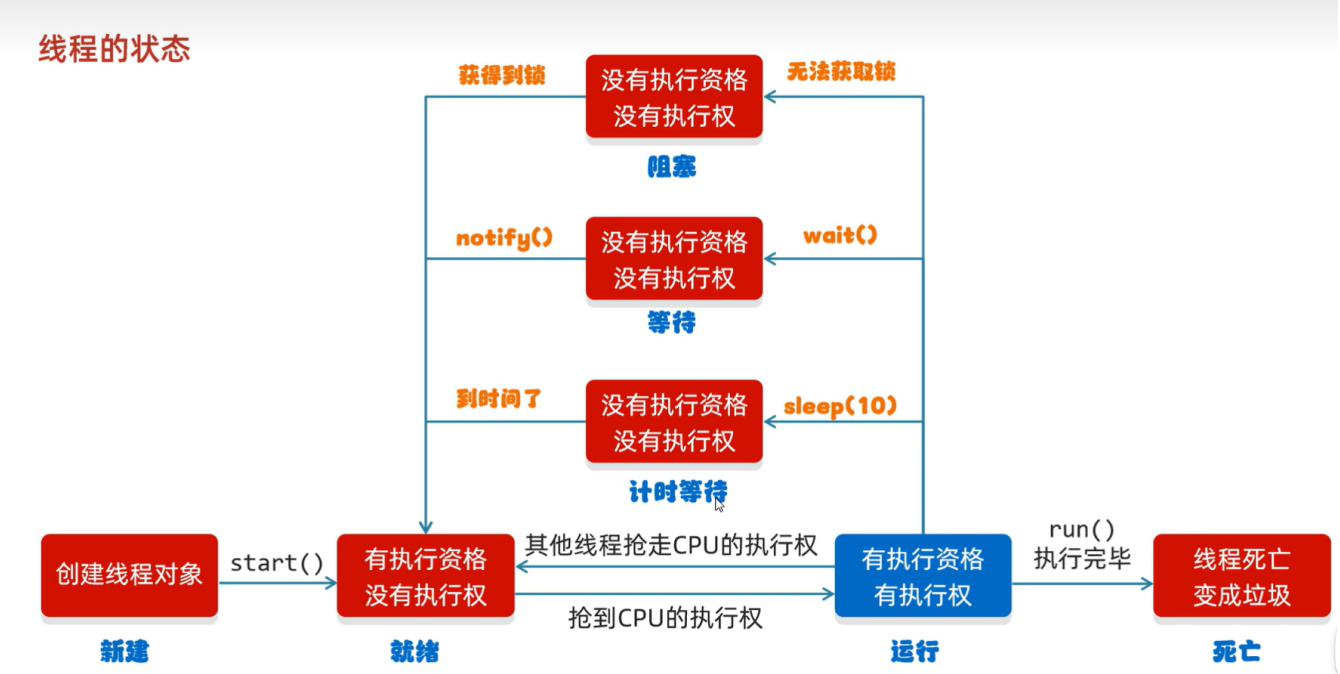
优点：可扩展性强，可返回运行结果 缺点：编程相对复杂

4.多线程中的常见成员方法

常见的成员方法

方法名称	说明
String getName()	返回此线程的名称
void setName(String name)	设置线程的名字（构造方法也可以设置名字）
static Thread currentThread()	获取当前线程的对象
static void sleep(long time)	让线程休眠指定的时间，单位为毫秒
setPriority(int newPriority)	设置线程的优先级
final int getPriority()	获取线程的优先级
final void setDaemon(boolean on)	设置为守护线程
public static void yield()	出让线程/礼让线程
public static void join()	插入线程/插队线程

5.线程的生命周期



6.线程的安全问题

6.1 什么是线程安全问题

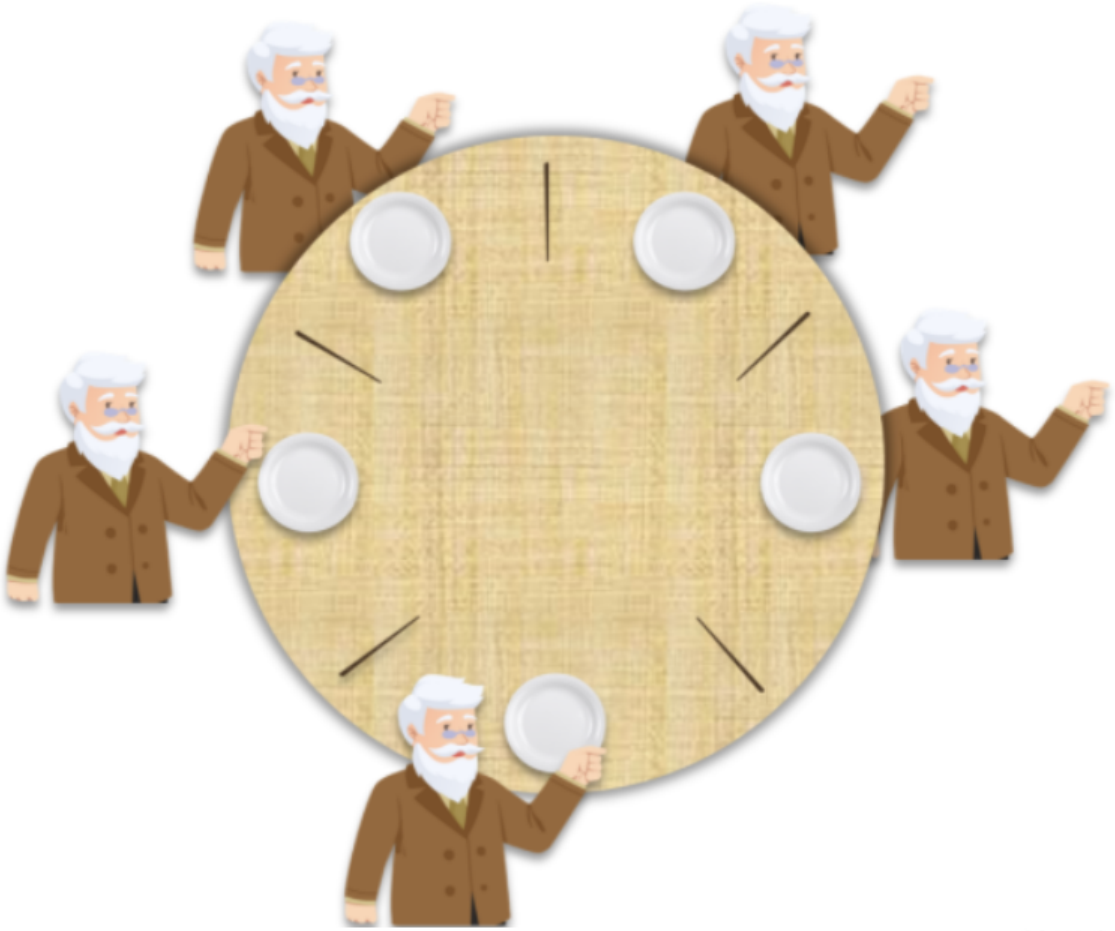
线程的调度是抢占式执行的，或者说是随机的，这就造成线程调度执行时线程的执行顺序是不确定的，有一些代码执行顺序不同不影响程序运行的结果，但也有一些代码执行顺序发生改变了重写的运行结果会受影响，这就造成程序会出现bug，对于多线程并发时会使程序出现bug的代码称作线程不安全的代码，这就是线程安全问题。

6.2 解决方法（上锁）

1. synchronized(锁) {} 同步代码块
2. 将synchronized加在方法上 同步方法 锁对象（this, 当前类的字节码文件对象）
3. lock锁 lock() unlock()

7.死锁

循环等待，即存在一个等待队列：P1占有P2的资源，P2占有P3的资源，P3占有P1的资源。这样就形成了一个等待环路。



8.生产者与消费者（等待唤醒机制）

一、作用

如果有两条线程相互争夺运行，他的结果可能如下：

A A A B B B A A A B B A B A B A B B A B

等待唤醒机制的作用就是打破这种随机性，让这两个线程轮流执行

A B A B A B A B A B A B A B A B A B A B

生产者负责生产数据，消费者负责消费数据

二、生产者和消费者（常见方法）

方法名称	说明

void wait()	当前线程等待，知道被其他线程唤醒
void notify()	随机唤醒单个线程
void notifyAll()	唤醒所有线程

三、代码示例

ThreadDemo

```
package com.thread.threadcase05;

public class ThreadDemo {
    public static void main(String[] args) {

        Cook cook = new Cook();
        Foodie foodie = new Foodie();

        Thread cookThread = new Thread(cook);
        Thread foodieThread = new Thread(foodie);

        cookThread.setName("Cook");
        foodieThread.setName("Foodie");

        cookThread.start();
        foodieThread.start();
    }
}

```

Cook

```
package com.thread.threadcase05;

/**
 * 生产者
 */
public class Cook implements Runnable {
    @Override
    public void run() {
        while (true) {
            synchronized (Desk.lock) {
                if (Desk.count == 0) {
```

```
                    break;
                } else  {
                    if (Desk.foodFlag == 0) {
                        System.out.println(Thread.currentThread().getName()
+ "正在烹饪...");
                        Desk.foodFlag = 1;
                        Desk.lock.notifyAll();
                    } else {
                        try {
                            Desk.lock.wait();
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                }
            }
        }

    }
}

```

Desk

```
package com.thread.threadcase05;

public class Desk {

    public static int foodFlag = 0;

    public static int count = 10;

    public static final Object lock = new Object();
}

```

Foodie

```
package com.thread.threadcase05;

import sun.security.krb5.internal.crypto.Des;

/**
 * 消费者
 */
public class Foodie implements Runnable {
    @Override
    public void run() {
```

```
        while (true) {
            synchronized (Desk.lock) {
                if (Desk.count == 0) {
                    break;
                } else {
                    if (Desk.foodFlag == 0) {
                        try {
                            Desk.lock.wait();
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    } else {
                        System.out.println(Thread.currentThread().getName()
+ " is eating food." );
                        Desk.lock.notifyAll();
                        Desk.count--;
                        Desk.foodFlag = 0;
                    }
                }

            }
        }
    }
}

```

9.参考资料

【黑马Java进阶教程，全面剖析Java多线程编程，含抢红包、抽奖实战案例】
https://www.bilibili.com/video/BV1LG4y1T7n2?p=21&vd_source=255fa06449e475e08fca96c700e92b97