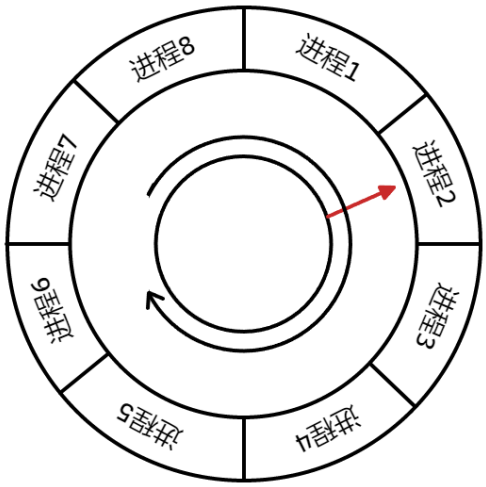
线程

1 概念

- 线程是进程中的单个顺序执行流，是一条执行路径。一个进程如果只有一条执行路径，则成为单线程程序；而如果有多条执行路径，则成为多线程程序。

1.1 CPU分时调度



- 时间片，即CPU分配给各个程序的时间，每个进程被分配一个时间段，称作它的**时间片**。即该进程允许运行的时间，使各个程序从表面上看是同时进行的。如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程，将当前进程挂起。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换，而不会造成CPU资源浪费。当又切换到之前执行的进程，把现场恢复，继续执行。
- 在宏观上：我们可以同时打开多个应用程序，每个程序并行，同时运行。
- 在微观上：由于只有一个CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。多核提高了并发能力。

1.2 单核CPU和多核CPU(了解):

- 单核CPU同时只能干一件事情，当启动多个程序时，CPU快速切换轮流处理每个程序，但如果CPU不够强劲，同时排队等待处理的任务太多，就会感觉系统会有延时、反应慢、卡顿等，甚至某一个程序在处理时出现错误，无法响应了，会造成后面排队的其他任务只能等待。
- 多核CPU是在基板上集成多个单核CPU+任务分配系统，两个核心同时处理两份任务，相比单核执行速度更快，有利于同时运行多个程序，不容易造成卡顿，更流畅！

1.3 Java程序的运行过程:

- 通过idea(java命令)运行一个Java程序，java命令会启动JVM(java虚拟机)，等于启动了一个应用程序，也就是启动了一个进程。
- 该进程会自动启动一个“主线程”，然后主线程去调用某个类的 main 方法，所以 main 方法运行在主线程中。在此之前的所有程序代码都是按照顺序依次调用的，都是单线程的。
- 如果希望程序中实现多段程序代码交替执行的效果，则需要创建多线程程序。

2 为什么要使用多线程?

- 单线程程序执行时都只有一条顺序执行流，如果程序执行某行代码时遇到了阻塞(比如:抛异常)，那么程序的运行将会停滞在这一行，其他代码将会无法执行!
- 这就像去银行办理业务，只有1个业务窗口(单线程)，所有的客户都需要在一个窗口排队办理业务，如果业务员在为某一个客户办理业务时，花费了很长时间，那么将会导致后面的客户等待很长时间，这样处理业务的效率也是非常低的。
- 但如果银行为了提高效率，同时开放了5个窗口(多线程)，客户可以分布在这5个窗口分别办理业务，即使某一个窗口在为个别客户办理业务时花费了很长时间，但不影响其他窗口办理业务的进度。
- 多线程理解起来其实非常简单:
    - 单线程的程序只有一个顺序执行流。
    - 多线程的程序则可以包括多个顺序执行流，多个顺序流之间互不干扰。

2.1 [并行]和[并发]

- 并行执行指在同一时刻，有多条指令在多个处理器上同时执行;
- 并发执行指在同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果,在微观角度而言,多个进程被分成多个片段依次随机执行。

3 多线程的特性

3.1 随机性

- 多线程的程序在执行时，在某一时间点具体执行哪个线程是不确定的，可以确定的是某一个时间点只有一个线程在执行(单核CPU)。
- 虽然我们感觉这些线程像是在同时运行，实际上是因为CPU在快速切换轮流执行这些线程，由于切换速度是纳秒级别的，所以我们感觉不到。

4 如何实现多线程

- 由于线程是依赖进程存在的，因此首先需要创建一个进程，但进程是操作系统创建的，而Java程序是不能直接调用系统功能的。但Java可以去调用C或C++写好的程序去创建进程从而实现多线程程序。
- 在Java中要想实现多线程操作有两种方式，一种是继承Thread类，另一种是实现Runnable接口。接下来针对这两种创建多线程的方式分别进行讲解。

4.1 继承Thread类

4.1.1 Thread类概述

- Thread类是在java.lang包中定义的类
- JavaSE规范中规定，一个类只要继承了Thread类，此类就是多线程的子类。
- 在Thread子类中，必须重写该类的run()方法，此方法为线程的主体。

4.1.2 通过继承Thread类创建线程

- 代码实现

```
package cn.tedu.thread;

/**
 * 多线程
 * 多线程可以并发执行多个任务
 * 线程的第一种创建方式:
 * 继承Thread类,并重写run方法,在run方法中定义需要并发执行的任务代码
 */
public class ThreadDemo01 {
    public static void main(String[] args) {
        //④创建要使用的线程实例
        MyThread01 t1 = new MyThread01();
        MyThread02 t2 = new MyThread02();
        //⑤将要执行的线程启动
        /*
         * Thread中提供的start方法,作用是将线程启动,并且纳入到线程调度器中,被统一管理,
         * 当线程被分配到CPU的时间片时,就会开始自动去执行各自线程中run方法定义的内容
         */
        t1.start();
        t2.start();
    }
}

/**
 * Thread是线程的父类
 * ①可以通过继承Thread的方式,创建线程实例
 */
class MyThread01 extends Thread {
    //②重写父类中的run方法 线程启动后,会自动执行run方法
    @Override
    public void run() {
        //③run方法中则定义线程要执行的任务
        for (int i = 0; i < 1000; i++) {
            System.out.println("我是金刚葫芦娃!");
        }
    }
}

class MyThread02 extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("我是查水表的!!");
        }
    }
}
```

- 从上面的运行结果可以看出，main方法(主线程)和run方法(子线程)中的两个for循环中的输出语句交替执行了，说明通过集成Thread类实现了多线程。(如果没有测试出主线程和子线程交替执行的效果，可以多测试几次!)

4.2 实现Runnable接口

4.2.1 Runnable接口概述

- 通过继承Thread类实现了多线程，但是这种方式有一定的局限性。因为Java中只支持单继承，一个类一旦继承了某个父类就无法再继承Thread类，例如猫类Cat继承了动物类Animal，就无法通过继承Thread类实现多线程。
- 为了克服这种弊端，在Thread类中提供了两个构造方法:
  - public Thread（Runnable target）
  - public Thread（Runnable target，String name）
- 这两个构造方法都可以接收Runnable的子类实例对象，这样创建的线程将调用实现了Runnable接口类中的run()方法作为运行代码，而不需要调用Thread类的run()方法，所以就可以依靠Runnable接口的实现类启动多线程。

4.2.2 通过实现Runnable接口实现多线程

- **代码实现**

```java
package cn.tedu.thread;

/**
 * 第二种创建线程的方式:
 * 实现Runnable接口单独定义线程任务
 */
public class ThreadDemo02 {
    public static void main(String[] args) {
        //④先将要执行的线程任务实例化
        MyRunnable01 r1 = new MyRunnable01();
        MyRunnable02 r2 = new MyRunnable02();
        //⑤将线程的任务分配给线程实例
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        //⑥将线程启动,会自动执行分配给其的任务
        t1.start();
        t2.start();
    }
}

/**
 * Runnable是线程任务类接口,不是线程本身
 * //①实现Runnable,创建线程任务子类
 */
class MyRunnable01 implements Runnable {
    //②强制要求必须重写run方法,线程启动后,会自动执行run方法的内容
    @Override
    public void run() {
        //③定义线程任务
        for (int i = 0; i < 1000; i++) {
            System.out.println("我是迪迦奥特曼!!!");
        }
    }
}

class MyRunnable02 implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("我是小怪兽!!");
```

```
        }
    }
}
```

- 从上面的运行结果可以看出，main方法(主线程)和run方法(子线程)中的两个for循环中的输出语句交替执行了，说明实现Runnable接口同样也实现了多线程。

4.3 简化写法

```
package cn.tedu.thread;

/**
 * 使用匿名内部类简化两种线程的创建方式
 */
public class ThreadDemo03 {
    public static void main(String[] args) {
        //简化直接继承Thread重写run方法的创建方式
        Thread t1 = new Thread() {
            @Override
            public void run() {
                for (int i = 0; i < 1000; i++) {
                    System.out.println("我是毛利小五郎!");
                }
            }
        };
        //简化实现Runnable重写run方法的创建方式
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1000; i++) {
                    System.out.println("我是齐天大圣!!");
                }
            }
        };
        Thread t2 = new Thread(r1);
        //通过Lambda表达式简化实现Runnable重写run方法的创建方式
        Thread t3 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                System.out.println("我是武大郎!!");
            }
        });
        t1.start();
        t2.start();
        t3.start();
    }
}
```

5 Thread的常用方法和总结

5.1 CurrentThreadDemo

```
package cn.tedu.thread;

/**
 * Thread中提供了一个静态的方法currentThread方法
 * 该方法可以返回运行这个方法的线程实例
 * java中所有的代码都是依靠线程运行的,main方法也不例外,JVM启动后,会自动创建一个线程,
 * 执行main方法,所以我们会将这条线程称为主线程,而这个线程的名字也叫"main"
 */
public class CurrentThreadDemo {
    public static void main(String[] args) {
```

```
        Thread t = Thread.currentThread();
        /*
         * Thread中提供了getName方法,用于获取线程名
         */
        System.out.println("主线程:" + t.getName());
        Thread t2 = new Thread("无敌风火轮");
        System.out.println("t2线程:" + t2.getName());
        Thread t3 = new Thread(() -> {
            Thread tt = Thread.currentThread();
            System.out.println("t3线程:" + tt.getName());
        });
        t3.setName("疯狂霸王鸡");
        t3.start();
        Thread t4 = new Thread("美丽金针菇") {
            @Override
            public void run() {
                Thread tt = currentThread();
                System.out.println("t4线程:" + tt.getName());
            }
        };
        t4.start();
    }
}
```

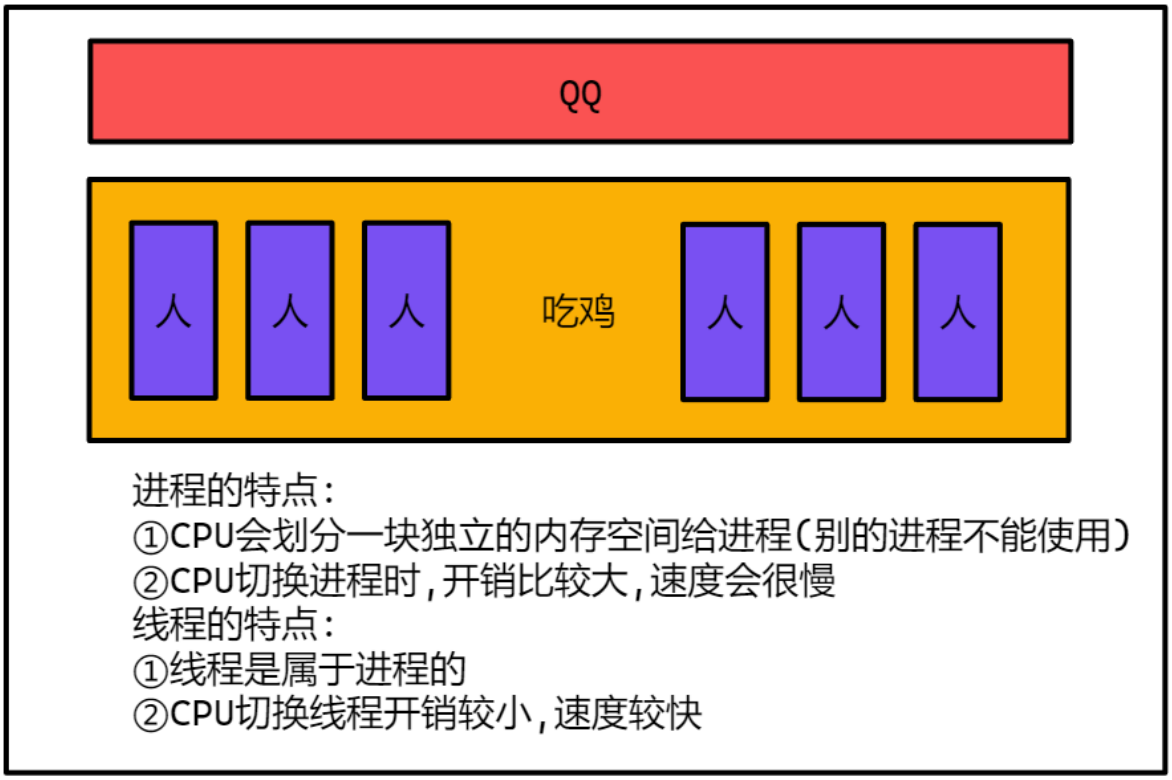
5.2 PriorityDemo

```
package cn.tedu.thread;

/**
 * 该案例学习线程的优先级
 * 线程的优先级分为10级,分别对应整数1-10,其中1是最低优先级,10是最高优先级,所有线程如果不设置
 * 优先级,则默认优先级为5
 */
public class PriorityDemo {
    public static void main(String[] args) {
        Thread min = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    System.out.println("我是min");
                }
            }
        };
        Thread norm = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    System.out.println("我是norm");
                }
            }
        };
        Thread max = new Thread() {
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    System.out.println("我是max");
                }
            }
        };
        min.setPriority(Thread.MIN_PRIORITY);//设置最低的优先级,就是1
        norm.setPriority(Thread.NORM_PRIORITY);//设置中度的优先级(不设置也是5)
        max.setPriority(Thread.MAX_PRIORITY);//设置最高的优先级,就是10
        min.start();
        norm.start();
```

```
36          max.start();
37      }
38  }
```

6 进程

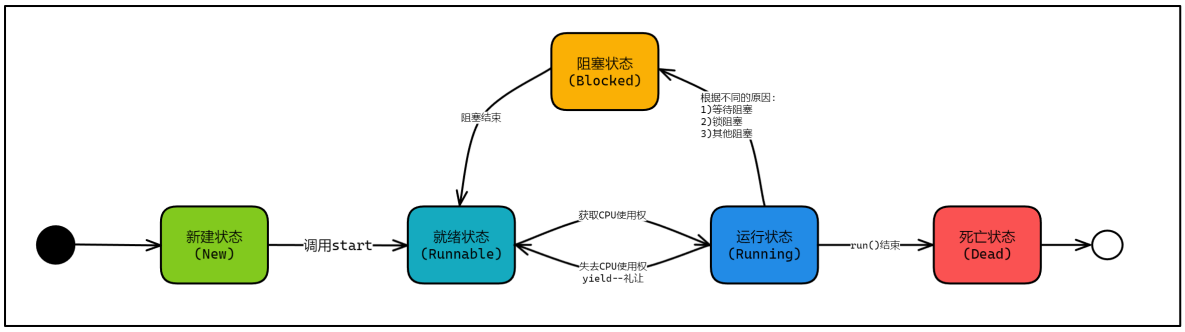
6.1 什么是进程



- 进程是操作系统中运行得到一个任务(一个应用程序运行在一个进程中).
- 进程(process)是一块包含了某些资源的内存区域.操作同利用进程把它的工作划分为一些功能单元.
- 进程中所包含的一个或者多个执行单元称为线程(thread).进程拥有一个私有的虚拟地址空间,该空间仅能被他所包含的线程访问
- 线程只能归属于一个进程并且他只能访问该进程所拥有的资源.当操作系统创建一个进程后,该进程会自动申请一个主线程或者首要线程的线程

7 线程状态

7.1 概述



①新建状态(New)：当一个线程对象被创建后，线程就处于新建状态。在新建状态中的线程对象从严格意义上看还只是一个普通的对象，还不是一个独立的线程，不会被线程调度程序调度。新建状态是线程生命周期的第一个状态。

例如: Thread t = new MyThread();

②就绪状态(Runnable)：处于新建状态中的线程被调用start()方法就会进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，但并不是说执行了start()方法线程就会立即执行;另外，在等待/阻塞状态中的线程，被解除等待和阻塞后将不直接进入运行状态，而是首先进入就绪状态。

③运行状态(Running)：处于就绪状态中的线程一旦被系统选中，使线程获取了 CPU 时间，就会进入运行状态。线程在运行状态下随时都可能被调度程序调度回就绪状态。在运行状态下还可以让线程进入到等待/阻塞状态。在通常的单核CPU中，在同一时刻只有一个线程处于运行状态。在多核的CPU中，就可能两个线程或更多的线程同时处于运行状态，这也是多核CPU运行速度快的原因。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，必须先处于就绪状态中。

④阻塞状态(Blocked)：根据阻塞产生的原因不同，阻塞状态又可以分为三种:

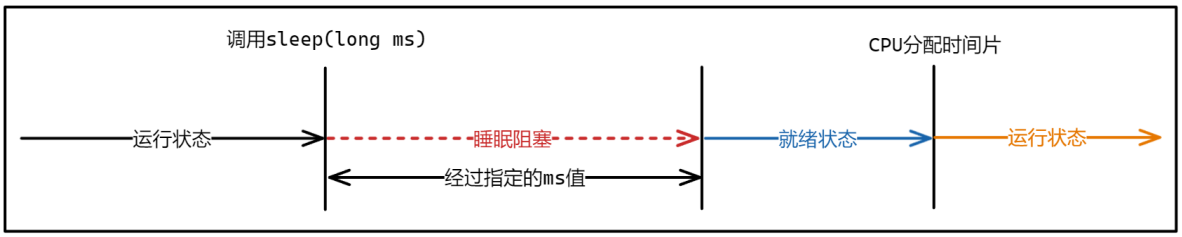
1)等待阻塞：运行状态中的线程执行wait()方法，使当前线程进入到等待阻塞状态;

2)锁阻塞：线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，线程会进入同步阻塞状态;

3)其他阻塞：通过调用线程的sleep(),suspend(), join(), 或发出了I/O请求时等，线程会进入到阻塞状态。当sleep()睡眠结束、调用resume(),?join()等待的线程终止或者超时、或I/O处理完毕时，线程重新转入就绪状态。

⑤死亡状态(Dead)：当线程中的run方法执行结束后，或者程序发生异常终止运行后，线程会进入死亡状态。处于死亡状态的线程不能再使用 start 方法启动线程。

7.2 SleepDemo01

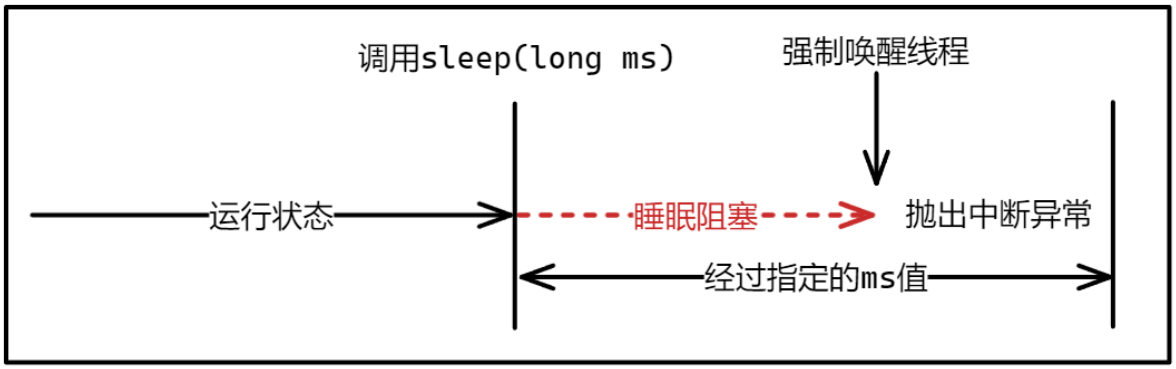



```java
package cn.tedu.thread;

/**
 * Thread中提供了一个静态的sleep方法
 * 当线程调用sleep方法后,会进入阻塞状态指定的毫秒,超过这个时间后,会自动进入到就绪状态,当CPU分配时间片后,会继续执行
 */
public class SleepDemo01 {
    public static void main(String[] args) {
        System.out.println("程序开始了!");
        try {
            //1秒=1000毫秒 让线程进入睡眠阻塞5秒时间
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("程序结束了!");
    }
}
```

7.3 SleepDemo02

```java
package cn.tedu.thread;

import java.util.Scanner;

/**
 * 利用sleep写一个倒计时的程序
 */
public class SleepDemo02 {
    public static void main(String[] args) {
        System.out.println("程序开始了!");
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.println("请输入要倒计时的时间!");
            int time = scanner.nextInt();
            //变量.forr 生成逆向for循环
            for (int i = time; i > 0; i--) {
                System.out.println(i);
                //没循环一次,睡眠阻塞1秒
                Thread.sleep(1000);
            }
            System.out.println("时间到!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("程序结束了!");
    }
}
```

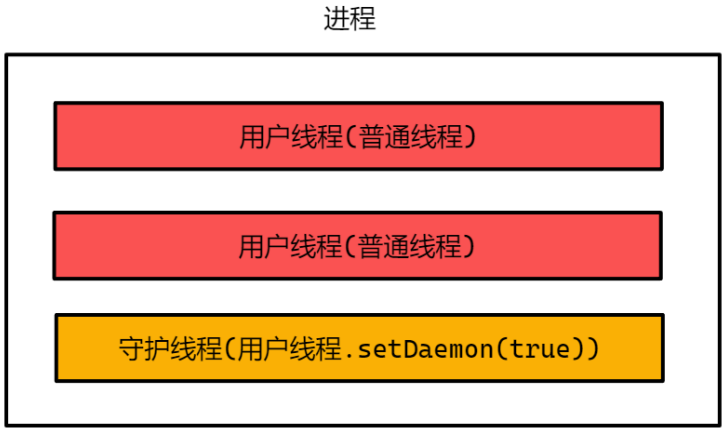
7.4 SleepDemo03




```java
package cn.tedu.thread;

/**
 * sleep方法调用时,必须要处理中断异常
 * 当一个线程调用sleep方法处于睡眠阻塞状态的过程中,如果该线程的interrupt()方法被调用时,
 * 会立即中断该睡眠阻塞,并抛出中断异常
 */
public class SleepDemo03 {
    public static void main(String[] args) {
        Thread lin = new Thread() {
            public void run() {
                System.out.println("林:刚打扫完卫生,小憩一会~");
                try {
                    Thread.sleep(10000);
                    System.out.println("林:睡的真舒服啊~~");
                } catch (InterruptedException e) {
```

```
                    System.out.println("林:干嘛呢!干嘛呢!都破了相!!!");
                }
            }
        };
        Thread huang = new Thread() {
            public void run() {
                System.out.println("黄:大锤80!小锤40!开始砸墙!");
                //5.fori
                for (int i = 0; i < 5; i++) {
                    System.out.println("黄(((;°ฅ°;))):80!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("哐啷啷!");
                System.out.println("黄:大哥!搞定!");
                //强制唤醒lin线程
                lin.interrupt();
            }
        };
        lin.start();
        huang.start();
    }
}
```

7.5 DaemonThreadDemo

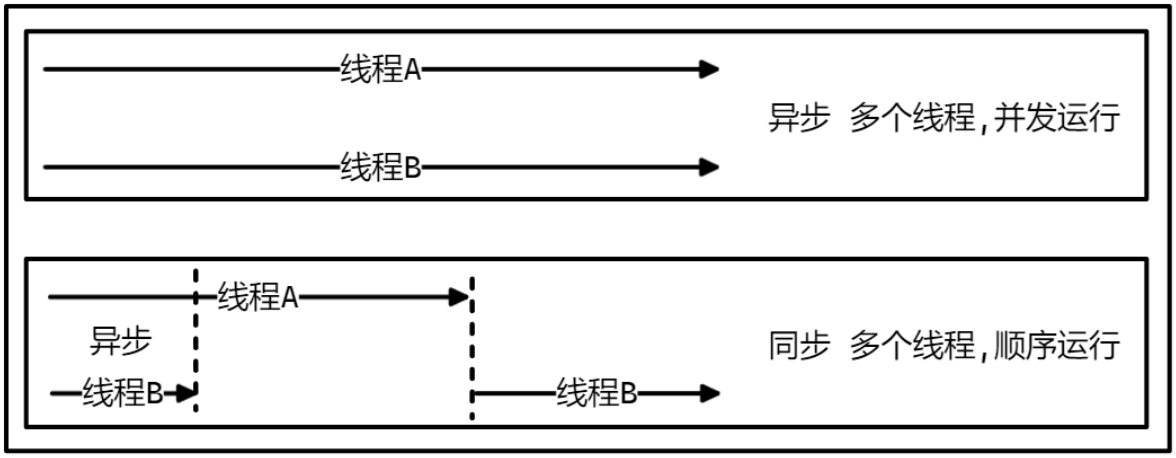


进程何时结束?
当一个进程中的所有的用户线程都结束时,进程就意味着结束了
而守护线程此时,会被强制杀死
守护线程的应用场景:
①垃圾回收
②I/O监控
③定时任务

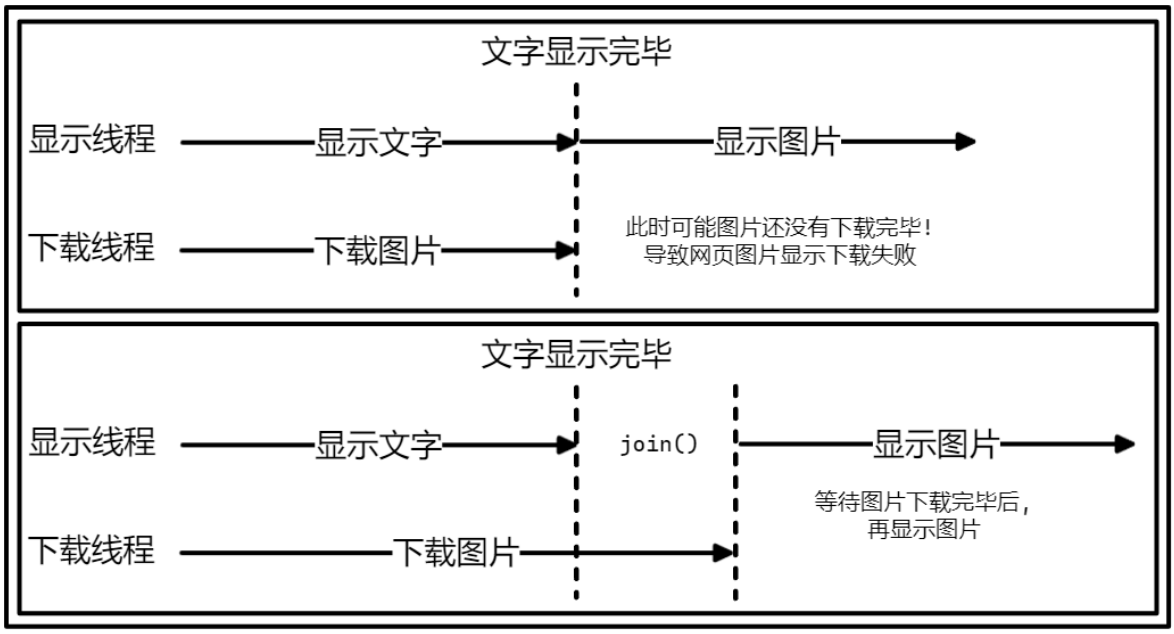
```
package cn.tedu.thread;

/**
 * 守护线程
 * java将线程分为两类,用户线程和守护线程,也称为前台线程和后台线程
 * 守护线程和用户线程区别不大,守护线程就是用户线程通过调用setDaemon(true)方法转变而来,
 * 而用户线程就是普通线程
 * 而两者最主要的区别在于当一个java进程中所有的用户线程都结束时,进程就会结束,此时会将所有的守护线程杀死
 */
public class DaemonThreadDemo {
    public static void main(String[] args) {
        Thread rose = new Thread() {
            @Override
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println("rose:Let me die!");
                    try {
```

```
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("rose:Ah, ah, ah, ah, ah, ah, ah....");
                System.out.println("噗通!咕噜噜噜噜");
            }
        };
        Thread jack = new Thread() {
            public void run() {
                while (true) {
                    System.out.println("jack:My darling,you jump!i
jump!!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };
        rose.start();
        //在jack线程启动之前,先将其设置为守护线程
        jack.setDaemon(true);
        jack.start();
        while (true);//程序会一直卡在此处,导致main永远不会结束
    }
}
```

8 线程的同步和异步



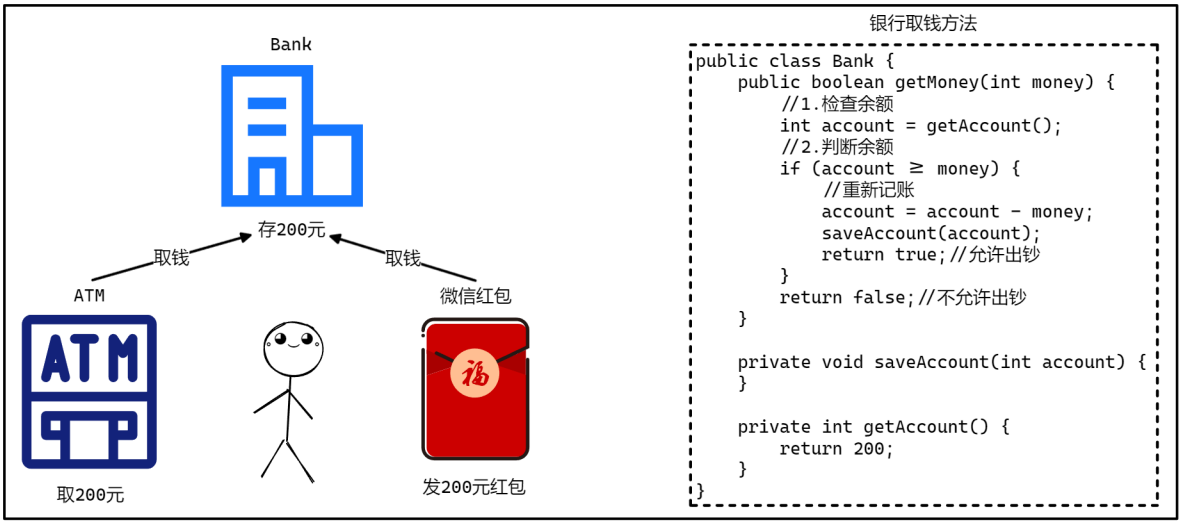
8.1 JoinDemo




```java
package cn.tedu.thread;

/**
 * 线程提供的join方法可以协调线程进入同步运行状态
 * 多线程本身就是并发运行的,所以本就是一种异步的状态
 * 而异步运行就表示: 多条线程各自执行各自的
 * 而同步运行则表示: 多条线程在运行时存在了先后的顺序
 */
public class JoinDemo {
    static boolean isFinish = false;//表示图片默认是未下载完

    public static void main(String[] args) {
        Thread download = new Thread() {
            public void run() {
                System.out.println("down: 开始下载图片...");
                for (int i = 1; i <= 100; i++) {
                    System.out.println("已下载" + i + "%");
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("down: 图片下载完毕!!!");
                isFinish = true;//表示图片此时已下载完毕
            }
        };
        Thread show = new Thread() {
            @Override
            public void run() {
                System.out.println("show: 开始显示文字...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("show: 显示文字完毕!!!");
                System.out.println("show: 开始显示图片...");
                //先等待下载线程运行结束后,再继续执行
                try {
                    /*
                     * 是show线程进入到阻塞状态,直到download执行完毕时,阻塞状态结束
```

```
                     * 理解为插队,show线程让download线程插队
                     * join方法和sleep方法的区别:
                     * ①sleep方法,可以让线程阻塞指定的时间
                     * ②join方法,可以让线程阻塞,但是时间不确定,具体得看插队的线程执
    行的时间
                     */
                    System.out.println("图片此时没下载完,等待下载ing...");
                    download.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //开始显示图片之前,判断图片下载状态
                if (isFinish == false) {
                    throw new RuntimeException("图片加载失败!!");
                }
                System.out.println("show: 图片显示完毕!!!");
            }
        };
        download.start();
        show.start();
    }
}
```

9 同步锁



9.1 同步方法

9.1.1 同步方法概述

- 除了可以将需要的代码设置成同步代码块以外，也可以使用synchronized关键字将一个方法修饰成同步方法，它能实现和同步代码块同样的功能。
- **语法格式**

```
权限修饰符 synchronized 返回值类型/void 方法名([参数1,...]){
  需要同步的代码;
}
```

- 被synchronized修饰的方法在某一时刻只允许一个线程访问，访问该方法的其他线程都会发生阻塞，直到当前线程访问完毕后，其他线程才有机会执行该方法。
- 需要注意的是，同步方法的锁是当前调用该方法的对象，也就是this指向的对象。

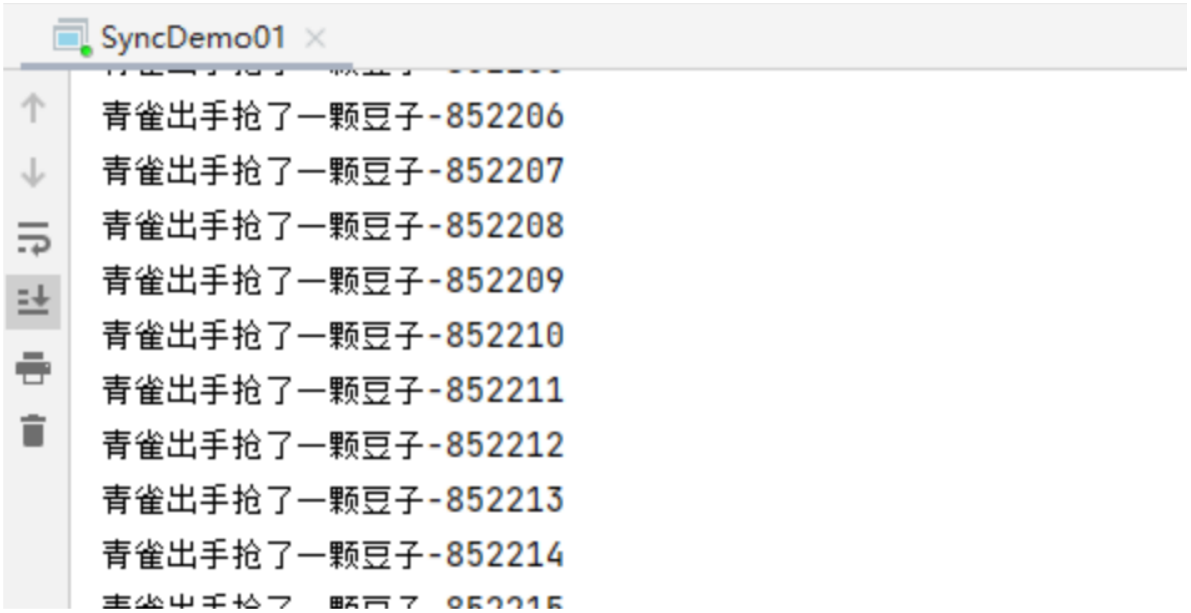
9.1.2 同步方法的使用

- 需要注意的是:
  - 将有可能发生线程安全问题的方法使用synchronized修饰，同一时间只允许一个线程进入同步方法中
  - synchronized方法的锁对象是当前调用该方法的对象，也就是this指向的对象。
- 如果当前方法是非静态方法，this表示的是调用当前方法的对象
- 如果当前方法的静态方法，this表示的是当前类。
- **示例1: SyncDemo1**

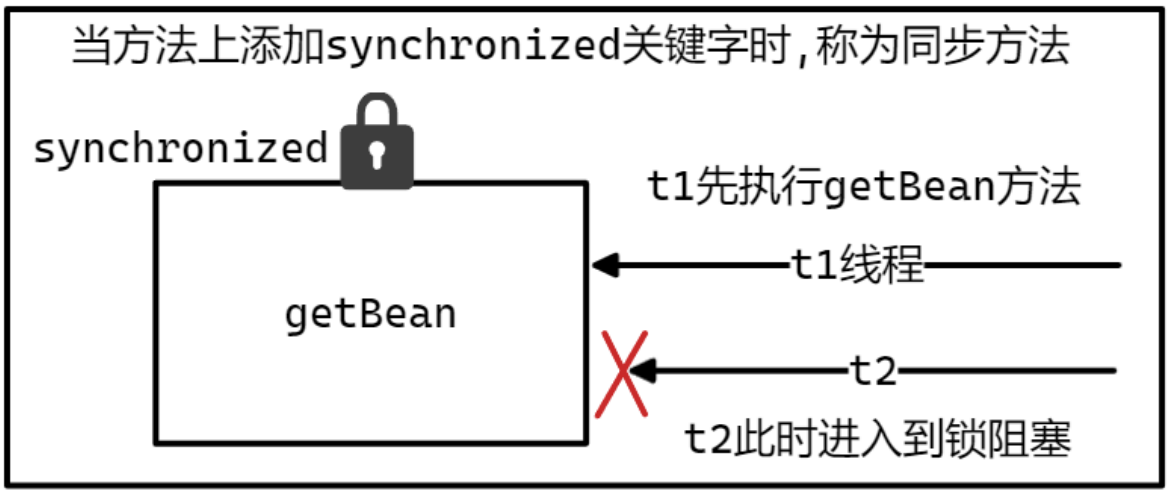
```
①t1线程执行getBean
②t1线程校验豆子数量,此时还剩1颗
③校验通过,但是恰巧停止到'return beans--'执行之前
④t2线程执行getBean
⑤t2线程校验豆子数量,此时还剩1颗
⑥校验通过,t2线程将最后一颗取走
⑦t1线程继续执行,此时直接将豆子数量减1
⑧之后t1和t2线程取豆子都可以直接取走
```

```
class Table {
    private int beans = 20;//桌子上有20颗豆子

    public int getBean() {
        if (beans == 0) {
            throw new RuntimeException("桌子上已经没有豆子了!!!");
        }
 —t1→ //
 —t2→ return beans--;
    }
}
```

- 出现并发安全问题



- 解决方法,使用同步方法



```java
package cn.tedu.thread;

/**
 * 多线程并发安全问题
 * 当多个线程并发操作同一临界资源时,由于线程的切换存在不可确定性,这就会导致线程的切换顺序出现混乱,而产生各种的逻辑错误
 * 而临界资源就是指操作资源的完整过程应该同一时刻只能由单线程执行
 */
public class SyncDemo01 {
    public static void main(String[] args) {
        Table table = new Table();
        Thread t1 = new Thread("白露") {
            @Override
            public void run() {
                while (true) {
                    int bean = table.getBean();
                    System.out.println(getName() + "抢一颗豆子,此时豆子数量为:" + (bean - 1));
                }
            }
        };
        Thread t2 = new Thread("青雀") {
            @Override
            public void run() {
                while (true) {
                    int bean = table.getBean();
                    System.out.println(getName() + "抢一颗豆子,此时豆子数量为:" + (bean - 1));
                }
            }
        };
        t1.start();
        t2.start();
    }
}

class Table {
    private int beans = 20;//桌子上有20颗豆子

    /*
     * 当一个方法使用关键字synchronized时,该方法称为"同步方法"
     * 同步: 指多个线程之间存在先后顺序执行
     * 同步方法: 指多个线程调用该方法需要有先后顺序
     * 多线程的并发安全问题通过让线程排队执行,可以有效解决该问题
     */
    public synchronized int getBean() {
        if (beans == 0) {
            throw new RuntimeException("桌子上已经没有豆子了!!!");
        }
        //礼让线程,主动让出CPU分配给他的时间片
        Thread.yield();
        return beans--;
    }
}
```

9.2 同步代码块

9.2.1 同步代码块概述

- 同步是指多个操作在同一个时间段内只能有一个线程进行，其他线程要等待此线程完成之后才可以继续执行。
- Java为线程的同步操作提供了synchronized关键字，使用该关键字修饰的代码块被称为同步代码块。

语法格式

```
synchronized( 同步对象 ){
  需要同步的代码;
}
```

- 注意: 在使用同步代码块时必须指定一个需要同步的对象，也称为锁对象，这里的锁对象可以是任意对象。但多个线程必须使用同一个锁对象。

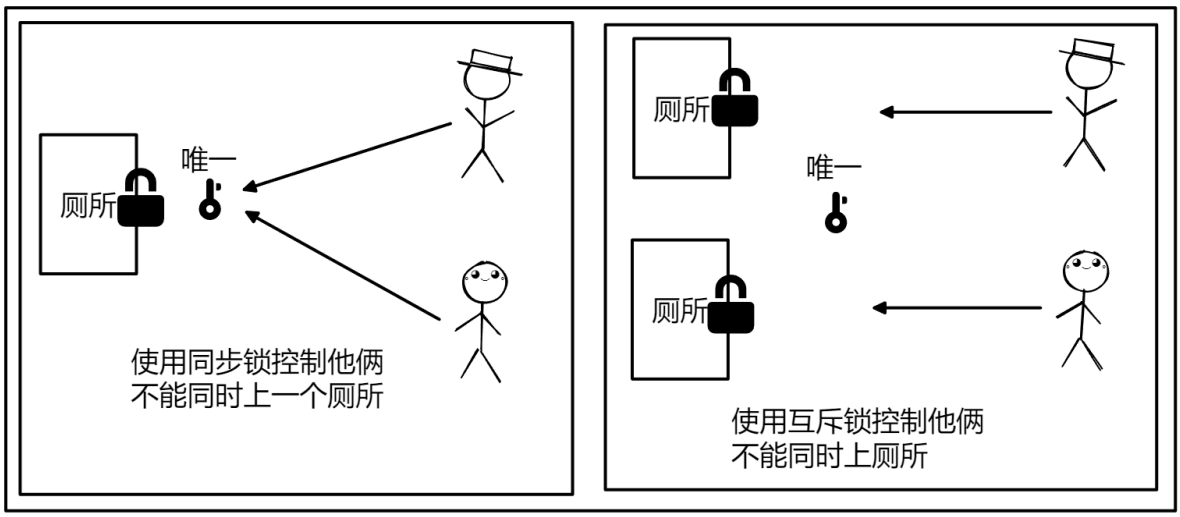
9.2.2 同步代码块的使用

- 需要注意的是:
  - 将有可能发生线程安全问题的代码包含在同步代码块中，同一时间只允许一个线程进入同步代码块
  - synchronized代码块中的锁对象可以是任意对象，但必须只能是一个锁。
  - 若使用this作为锁对象，需保证多个线程执行时，this指向的是同一个对象
- **代码案例**

```
package cn.tedu.thread;

/**
 * 同步块的应用
 * 有效的缩小同步范围,并可以在保证并发安全的情况下,尽可能的提高并发效率
 */
public class SyncDemo02 {
    public static void main(String[] args) {
        Shop shop = new Shop();
        Thread t1 = new Thread("缪铖航") {
            @Override
            public void run() {
                shop.buy();
            }
        };
        Thread t2 = new Thread("薛宏举") {
            @Override
            public void run() {
                shop.buy();
            }
        };
        t1.start();
        t2.start();
    }
}

class Shop {
    public void buy() {
        try {
            Thread t = Thread.currentThread();
            System.out.println(t.getName() + ": 正在挑衣服...");
            Thread.sleep(5000);
            /*
```

```
                 * 同步块在指定同步监视器对象时,可以是任何引用类型实例,
                 * 只要保证多个执行该代码片段的线程看到的这个对象是"同一个"即可
                 * 此处使用this this代表当前实例化对象的引用,也就是调用buy方法的实例对象
                 * t1线程中调用buy方法时,this指向的是shop实例,而t2线程也是shop实例,所以此处可以使用this
                 */
                synchronized (this) {
                    System.out.println(t.getName() + ": 正在试衣服...");
                    Thread.sleep(5000);
                }
                System.out.println(t.getName() + ": 结账离开!!!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

9.3 互斥锁




```
package cn.tedu.thread;

/**
 * 互斥锁
 * 当使用多个synchronized关键字锁定多个代码片段,并且指定的锁对象都是相同的,那么这些代码片段之间就是互斥的
 */
public class SyncDemo03 {
    public static void main(String[] args) {
        Person person = new Person();
        Thread t1 = new Thread() {
            @Override
            public void run() {
                person.eat();
            }
        };
        Thread t2 = new Thread() {
            @Override
            public void run() {
                person.breath();
            }
        };
        t1.start();
        t2.start();
    }
}
```

```
26
27  class Person {
28      public synchronized void eat() {
29          try {
30              Thread t = Thread.currentThread();
31              System.out.println(t.getName() + ": 正在吃饭...");
32              Thread.sleep(5000);
33              System.out.println(t.getName() + ": 吃饭完毕!!!");
34          } catch (InterruptedException e) {
35              e.printStackTrace();
36          }
37      }
38
39      public synchronized void breath() {
40          try {
41              Thread t = Thread.currentThread();
42              System.out.println(t.getName() + ": 正在呼吸...");
43              Thread.sleep(5000);
44              System.out.println(t.getName() + ": 呼吸完毕!!!");
45          } catch (InterruptedException e) {
46              e.printStackTrace();
47          }
48      }
49  }
```

10 线程池

```
1
```