多线程-初阶

本节目标

> - 认识多线程
> - 掌握多线程程序的编写
> - 掌握多线程的状态
> - 掌握什么是线程不安全及解决思路
> - 掌握 synchronized、volatile 关键字

0. 面试题

1、**说下乐观锁悲观锁**

2、object类的原生方法有哪些？

3、说下b树和b+树

4、**说下hashmap实现原理，为什么线程不安全？**

5、**Concurrenthashmap如何实现线程安全？**

6、**Concurrenthashmap在jdk1.8做了哪些优化？**

7、知道BIO和NIO吗？

8、仔细讲讲Websockt

9、说说Tcp断开连接的过程

10、Hashcode方法有什么作用？

12、说下equals和==的区别

13、如果判断一个链表有环，环的入口在哪

14、一共100个乒乓，每次拿一到六个，我和你轮流拿，能拿到最后一个球的人获胜，你先拿，你怎么才能赢？

15、有4个g的数据，给你256m的空间，怎么把它排序，然后写进另一个文件里？

16、知道类加载器吗？有哪些？说说类加载机制

17、你知道哪些异常？说说异常的执行过程

18、**你自己如何实现一个线程池，要用什么数据结构？**

19、**知道threatlocal吗？说一下原理**

20、用过哪些数据库，MySQL如何保证安全？有哪些锁？

21、静态方法和实例方法的区别，在内存中存放的位置

22、说下http协议，http1.0、http1.1、http2.0有什么区别？https呢？如何保证安全的？

1. 认识线程（Thread）

1.1 概念

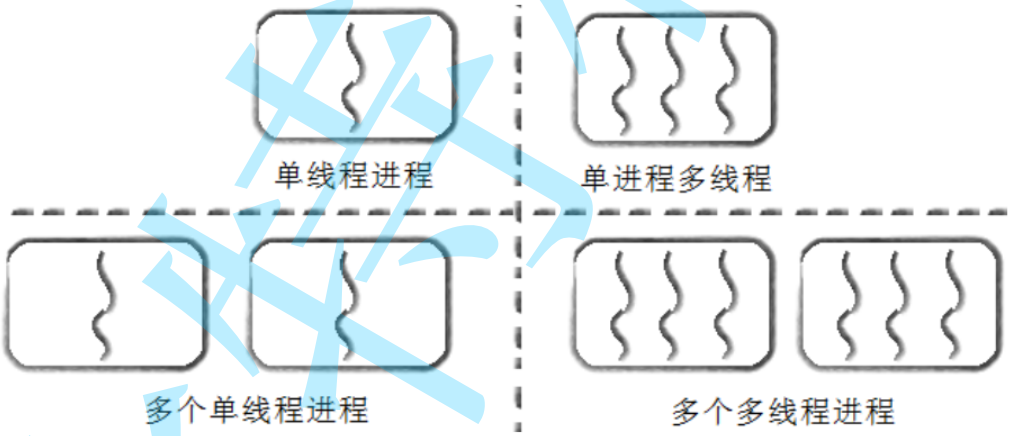
还是回到我们之前的银行的例子中。之前我们主要描述的是个人业务，即一个人完全处理自己的业务。我们进一步设想如下场景：

一家公司要去银行办理业务，既要进行财务转账，又要进行福利发放，还得进行缴社保。如果只有张三一个会计就会忙不过来，耗费的时间特别长。为了让业务更快的办理好，张三又找来两位同事李四、王五一起来帮助他，三个人分别负责一个事情，分别申请一个号码进行排队，自此就有了三个执行流共同完成任务，但本质上他们都是为了办理一家公司的业务。此时，我们就把这种情况称为多线程，将一个大任务分解成不同小任务，交给不同执行流就分别排队执行。其中李四、王五都是张三叫来的，所以张三一般被称为主线程（Main Thread）。

那这个和多进程又有什么区别的，最大的区别就是这些执行流之间是否有资源的共享。比如之前的多进程例子中，每个客户来银行办理各自的业务，但他们之间的票据肯定是不想让别人知道的，否则钱不就被其他人取走了么。而上面我们的公司业务中，张三、李四、王五虽然是不同的执行流，但因为办理的都是一家公司的业务，所以票据是共享着的。这个就是多线程和多进程的最大区别。

进程是系统分配资源的最小单位，线程是系统调度的最小单位。一个进程内的线程之间是可以共享资源的。

每个进程至少有一个线程存在，即主线程。



1.2 动手接触线程

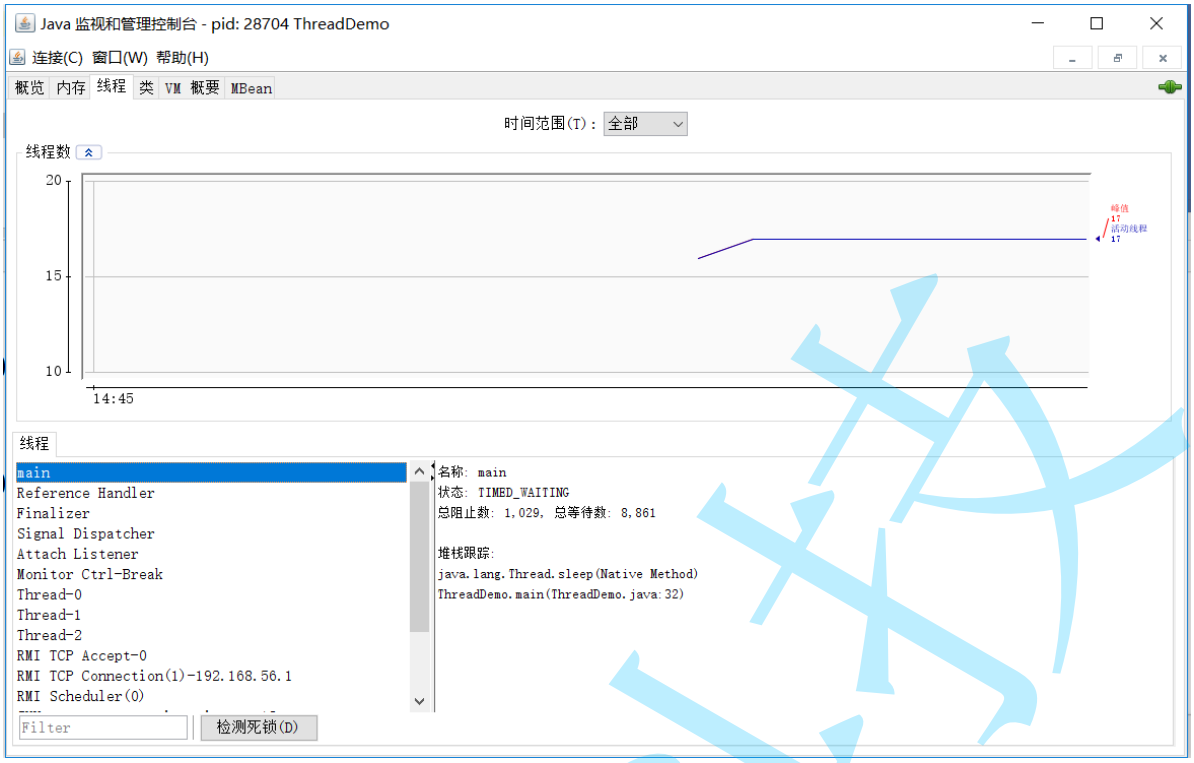
通过运行下面的程序，感受多线程程序和普通程序的区别。

```
import java.util.Random;

public class ThreadDemo {
    private static class MyThread extends Thread {
        @Override
        public void run() {
            Random random = new Random();
            while (true) {
                // 打印线程名称
                System.out.println(Thread.currentThread().getName());
                try {
```

```
                    // 随机停止运行 0-9 秒
                    Thread.sleep(random.nextInt(10));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.start();
        t2.start();
        t3.start();

        Random random = new Random();
        while (true) {
            // 打印线程名称
            System.out.println(Thread.currentThread().getName());
            try {
                Thread.sleep(random.nextInt(10));
            } catch (InterruptedException e) {
                // 随机停止运行 0-9 秒
                e.printStackTrace();
            }
        }
    }
}
```

```
Thread-0
Thread-0
Thread-2
Thread-1
Thread-2
Thread-1
Thread-0
Thread-2
main
main
Thread-2
Thread-1
Thread-0
Thread-1
main
Thread-2
Thread-2
等等
```

使用 `jconsole` 命令观察线程



1.3 多线程的优势-增加运行速度

可以观察多线程在一些场合下是可以提高程序的整体运行效率的。

```
public class ThreadAdvantage {
    // 多线程并不一定就能提高速度，可以观察，count 不同，实际的运行效果也是不同的
    private static final long count = 10_0000_0000;

    public static void main(String[] args) throws InterruptedException {
        // 使用并发方式
        concurrency();
        // 使用串行方式
        serial();
    }

    private static void concurrency() throws InterruptedException {
        long begin = System.nanoTime();

        // 利用一个线程计算 a 的值
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                int a = 0;
                for (long i = 0; i < count; i++) {
                    a--;
                }
            }
        });
        thread.start();
        // 主线程内计算 b 的值
        int b = 0;
        for (long i = 0; i < count; i++) {
            b--;
        }
        // 等待 thread 线程运行结束
```

```
        thread.join();

        // 统计耗时
        long end = System.nanoTime();
        double ms = (end - begin) * 1.0 / 1000 / 1000;
        System.out.printf("并发: %f 毫秒%n", ms);
    }

    private static void serial() {
        // 全部在主线程内计算 a、b 的值
        long begin = System.nanoTime();
        int a = 0;
        for (long i = 0; i < count; i++) {
            a--;
        }
        int b = 0;
        for (long i = 0; i < count; i++) {
            b--;
        }
        long end = System.nanoTime();
        double ms = (end - begin) * 1.0 / 1000 / 1000;
        System.out.printf("串行: %f 毫秒%n", ms);
    }
}
```

```
并发: 399.651856 毫秒
串行: 720.616911 毫秒
```

1.4 创建线程-方法1-继承 Thread 类

可以通过继承 Thread 来创建一个线程类，该方法的好处是 this 代表的就是当前线程，不需要通过 Thread.currentThread() 来获取当前线程的引用。

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("这里是线程运行的代码");
    }
}
```

```
MyThread t = new MyThread();
t.start();  // 线程开始运行
```

1.5 创建线程-方法2-实现 `Runnable` 接口

通过实现 Runnable 接口，并且调用 Thread 的构造方法时将 Runnable 对象作为 target 参数传入来创建线程对象。该方法的好处是可以规避类的单继承的限制；但需要通过 Thread.currentThread() 来获取当前线程的引用。

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "这里是线程运行的代
码");
    }
}
```

```
Thread t = new Thread(new MyRunnable());
t.start();  // 线程开始运行
```

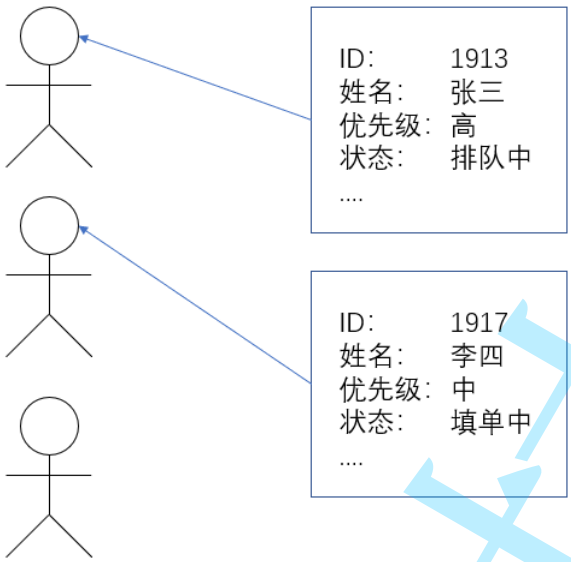
1.6 创建线程-其他变形（了解）

```
// 使用匿名类创建 Thread 子类对象
Thread t1 = new Thread() {
    @Override
    public void run() {
        System.out.println("使用匿名类创建 Thread 子类对象");
    }
};

// 使用匿名类创建 Runnable 子类对象
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("使用匿名类创建 Runnable 子类对象");
    }
});

// 使用 lambda 表达式创建 Runnable 子类对象
Thread t3 = new Thread(() -> System.out.println("使用匿名类创建 Thread 子类对象"));
Thread t4 = new Thread(() -> {
    System.out.println("使用匿名类创建 Thread 子类对象");
});
```

2. Thread 类及常见方法

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。



2.1 Thread 的常见构造方法

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, String name)	使用 Runnable 对象创建线程对象，并命名
【了解】Thread(ThreadGroup group, Runnable target)	线程可以被用来分组管理，分好的组即使线程组，这个目前我们了解即可

```
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");
```

2.2 Thread 的几个常见属性

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()

- ID 是线程的唯一标识，不同线程不会重复
- 名称是各种调试工具用到
- 状态表示线程当前所处的一个情况，下面我们会进一步说明
- 优先级高的线程理论上来说更容易被调度到
- 关于后台线程，需要记住一点：**JVM会在一个进程的所有非后台线程结束后，才会结束运行。**
- 是否存活，即简单的理解，为 run 方法是否运行结束了
- 线程的中断问题，下面我们进一步说明

```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    System.out.println(Thread.currentThread().getName() + ": 我还
活着");
                    Thread.sleep(1 * 1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println(Thread.currentThread().getName() + ": 我即将死去");
        });

        System.out.println(Thread.currentThread().getName()
                           + ": ID: " + thread.getId());
        System.out.println(Thread.currentThread().getName()
                           + ": 名称: " + thread.getName());
        System.out.println(Thread.currentThread().getName()
                           + ": 状态: " + thread.getState());
        System.out.println(Thread.currentThread().getName()
                           + ": 优先级: " + thread.getPriority());
        System.out.println(Thread.currentThread().getName()
                           + ": 后台线程: " + thread.isDaemon());
        System.out.println(Thread.currentThread().getName()
                           + ": 活着: " + thread.isAlive());
        System.out.println(Thread.currentThread().getName()
                           + ": 被中断: " + thread.isInterrupted());

        thread.start();
        while (thread.isAlive()) {}
        System.out.println(Thread.currentThread().getName()
                           + ": 状态: " + thread.getState());
    }
}
```

2.3 启动一个线程-start()

之前我们已经看到了如何通过覆写 run 方法创建一个线程对象，但线程对象被创建出来并不意味着线程就开始运行了。

- 覆写 run 方法是提供给线程要做的事情的指令清单
- 线程对象可以认为是把 李四、王五叫过来了
- 而调用 start() 方法，就是喊一声：”行动起来！“，线程才真正独立去执行了。

面试题: 一定要注意 run 方法和 start 方法是不同的，启动线程必须要调用 start 方法。

2.4 中断一个线程

李四一旦进到工作状态，他就会按照行动指南上的步骤去进行工作，不完成是不会结束的。但有时我们需要增加一些机制，例如老板突然来电话了，说转账的对方是个骗子，需要赶紧停止转账，那张三该如何通知李四停止呢？这就涉及到我们的停止线程的方式了。

目前常见的有以下两种方式：

1. 通过共享的标记来进行沟通
2. 调用 interrupt() 方法来通知

示例-1

```
public class ThreadDemo {
    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;
    }
}
```

示例-2

```
public class Thread2 {
    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！");
                    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();
    }
}
```

重点说明下第二种方法：

1. 通过 thread 对象调用 interrupt() 方法通知该线程停止运行
2. thread 收到通知的方式有两种：
    1. 如果线程调用了 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**

2. 否则，只是内部的一个中断标志被设置，thread 可以通过
    1. Thread.interrupted() 判断当前线程的中断标志被设置，**清除中断标志**
    2. Thread.currentThread().isInterrupted() 判断指定线程的中断标志被设置，**不清除中断标志**

第二种方式通知收到的更及时，即使线程正在 sleep 也可以马上收到。

示例-3

```
public class ThreadDemo {
    private static class MyRunnable implements Runnable {
        @Override
        public void run() {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("通过异常收到了中断情况");
            }
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().isInterrupted());
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyRunnable target = new MyRunnable();
        Thread thread = new Thread(target, "李四");
        thread.start();
        thread.interrupt();
    }
}
```

```
通过异常收到了中断情况 // 通过异常收到中断通知，并且标志位被清
false
false
false
false
false
false
false
false
false
false
```

示例-4

```
public class ThreadDemo {
    private static class MyRunnable implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.interrupted());
            }
        }
    }
```

```
    public static void main(String[] args) throws InterruptedException {
        MyRunnable target = new MyRunnable();
        Thread thread = new Thread(target, "李四");
        thread.start();
        thread.interrupt();
    }
}
```

```
true    // 只有一开始是 true，后边都是 false，因为标志位被清
false
false
false
false
false
false
false
false
false
```

示例-5

```
public class ThreadDemo {
    private static class MyRunnable implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().isInterrupted());
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyRunnable target = new MyRunnable();
        Thread thread = new Thread(target, "李四");
        thread.start();
        thread.interrupt();
    }
}
```

```
true    // 全部是 true，因为标志位没有被清
true
true
true
true
true
true
true
true
true
```

附录

方法	说明
public void interrupt()	中断对象关联的线程，如果线程正在阻塞，则以异常方式通知，否则设置标志位
public static boolean interrupted()	判断当前线程的中断标志位是否设置，调用后清除标志位
public boolean isInterrupted()	判断对象关联的线程的标志位是否设置，调用后不清除标志位

2.5 等待一个线程-join()

有时，我们需要等待一个线程完成它的工作后，才能进行自己的下一步工作。例如，张三只有等李四转账成功，才决定是否存钱，这时我们需要一个方法明确等待线程的结束。

```
public class Thread2 {
    public static void main(String[] args) throws InterruptedException {
        Runnable target = () -> {
            for (int i = 0; i < 10; i++) {
                try {
                    System.out.println(Thread.currentThread().getName()
                                       + ": 我还在工作！");
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println(Thread.currentThread().getName() + ": 我结束了！");
        };

        Thread thread1 = new Thread(target, "李四");
        Thread thread2 = new Thread(target, "王五");
        System.out.println("先让李四开始工作");
        thread1.start();
        thread1.join();
        System.out.println("李四工作结束了，让王五开始工作");
        thread2.start();
        thread2.join();
        System.out.println("王五工作结束了");
    }
}
```

大家可以试试如果把两个 join 注释掉，现象会是怎么样的呢?

附录

方法	说明
public void join()	等待线程结束
public void join(long millis)	等待线程结束，最多等 millis 毫秒
public void join(long millis, int nanos)	同理，但可以更高精度

关于 join 还有一些细节内容，我们留到下面再讲解。

2.6 获取当前线程引用

这个方法我们以及非常熟悉了

方法	说明
public static Thread currentThread();	返回当前线程对象的引用

```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName());
    }
}
```

2.7 休眠当前线程

也是我们比较熟悉一组方法，有一点要记得，因为线程的调度是不可控的，所以，这个方法只能保证实际休眠时间是大于等于参数设置的休眠时间的。

方法	说明
public static void sleep(long millis) throws InterruptedException	休眠当前线程 millis 毫秒
public static void sleep(long millis, int nanos) throws InterruptedException	可以更高精度的休眠

```
public class ThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        System.out.println(System.currentTimeMillis());
        Thread.sleep(3 * 1000);
        System.out.println(System.currentTimeMillis());
    }
}
```

关于 sleep，以后我们还会有一些知识会给大家补充。

3. 线程的状态

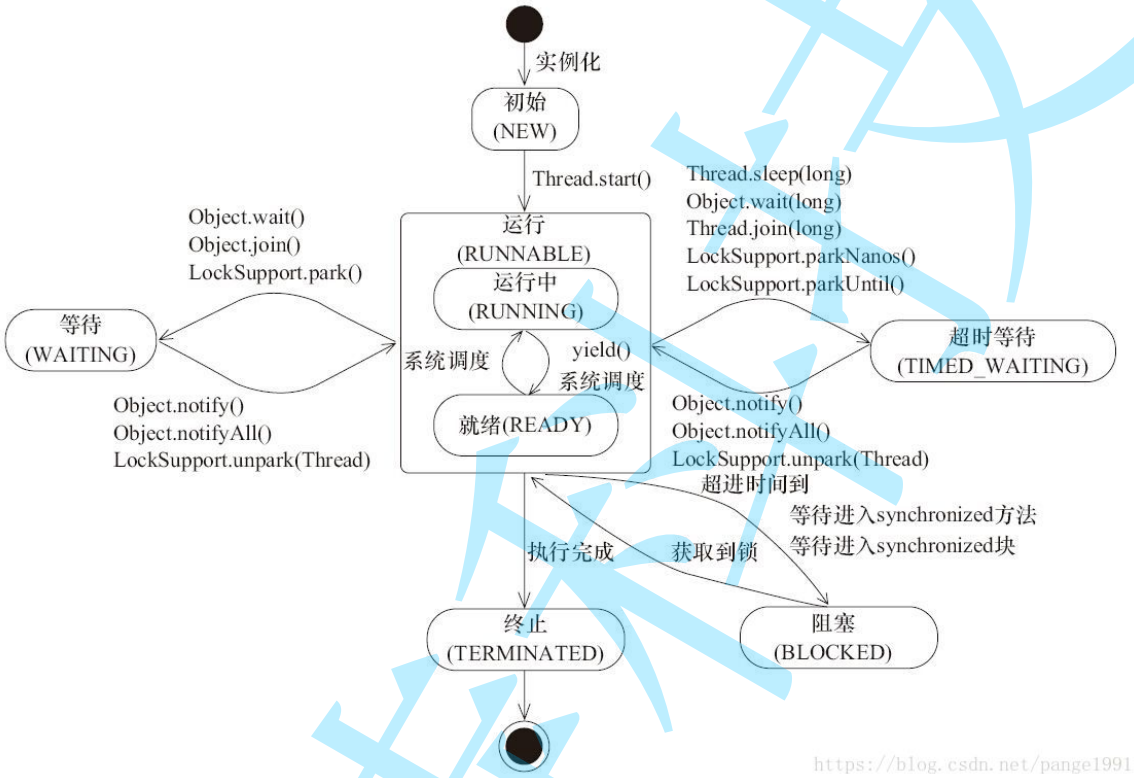
3.1 观察线程的所有状态

线程的状态是一个枚举类型 Thread.State

```
public class ThreadState {
    public static void main(String[] args) {
        for (Thread.State state : Thread.State.values()) {
            System.out.println(state);
        }
    }
}
```

```
NEW
RUNNABLE
BLOCKED
WAITING
TIMED_WAITING
TERMINATED
```

3.2 线程状态和状态转移的意义



大家不要被这个状态转移图吓到，我们重点是要理解状态的意义以及各个状态的具体意思。

还是我们之前的例子：

刚把李四、王五找来，还是给他们在安排任务，没让他们行动起来，就是 NEW 状态；

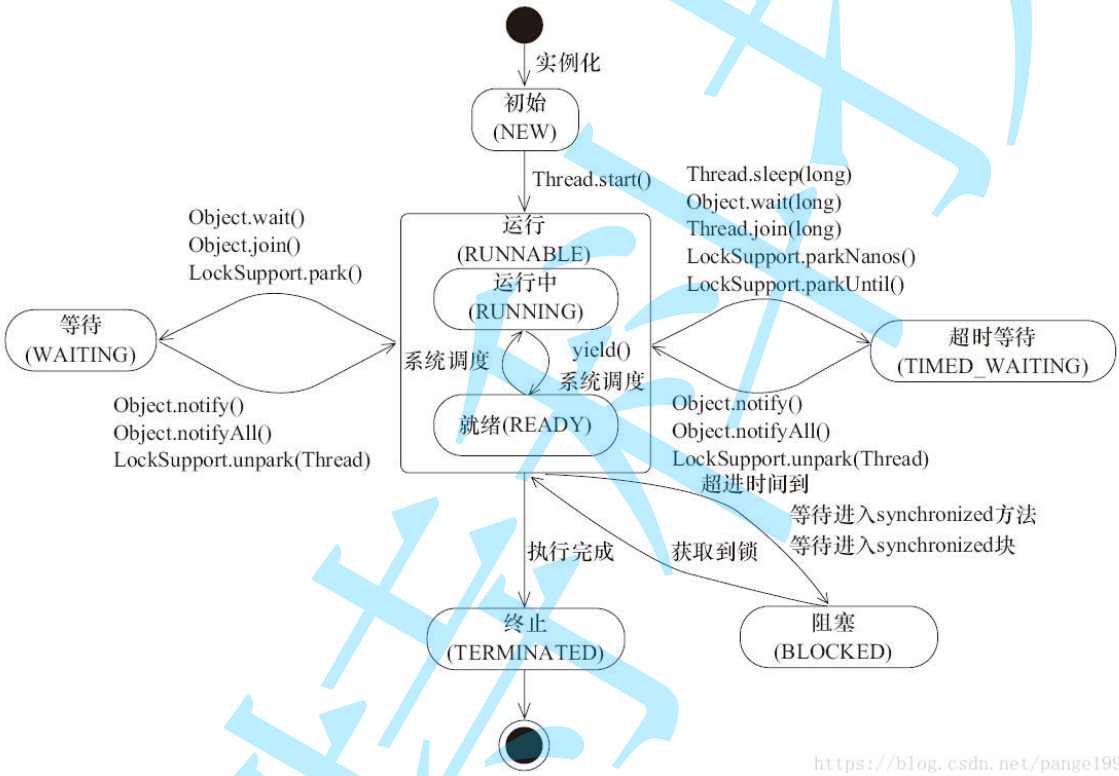
当李四、王五开始去窗口排队，等待服务，就进入到 `RUNNABLE` 状态。**该状态并不表示已经被银行工作人员开始接待，排在队伍中也是属于该状态，即可被服务的状态，是否开始服务，则看调度器的调度**；

当李四、王五因为一些事情需要去忙，例如需要填写信息、回家取证件、发呆一会等等时，进入 `BLOCKED`、`WATING`、`TIMED_WAITING` 状态，至于这些状态的细分，我们以后再详解；

如果李四、王五已经忙完，为 `TERMINATED` 状态。

所以，之前我们学过的 isAlive() 方法，可以认为是处于不是 NEW 和 TERMINATED 的状态都是活着的。

3.3 观察线程的状态和转移



观察 1: 关注 `NEW`、`RUNNABLE`、`TERMINATED` 状态的转换

```
public class ThreadStateTransfer {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> {
            for (int i = 0; i < 1000_0000; i++) {
            }
        }, "李四");

        System.out.println(t.getName() + ": " + t.getState());;
        t.start();
        while (t.isAlive()) {
            System.out.println(t.getName() + ": " + t.getState());;
        }
        System.out.println(t.getName() + ": " + t.getState());;
    }
}
```

观察 2: 关注 `WAITING`、`BLOCKED`、`TIMED_WAITING` 状态的转换

```
public class ThreadStateTransfer {
    public static void main(String[] args) throws InterruptedException {
        Object object = new Object();

        Thread t = new Thread(() -> {
            synchronized (object) {
                try {
                    object.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                for (int i = 0; i < 1000_0000; i++) {}
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        System.out.println(t.getState());;
        t.start();
        System.out.println(t.getState());
        Thread.sleep(10);
        synchronized (object) {
            for (int i = 0; i < 10; i++) {
                System.out.println(t.getState());
            }
            object.notify();
        }
        while (t.isAlive()) {
            System.out.println(t.getState());
        }
    }
}
```

观察-3: yield() 大公无私，让出 CPU

```
public class ThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(() -> {
            while (true) {
                System.out.println(Thread.currentThread().getName()
                                   + ": 我跑着呢");
                // Thread.yield();
            }
        }, "李四");
        Thread thread2 = new Thread(() -> {
            while (true) {
                System.out.println(Thread.currentThread().getName()
                                   + ": 我跑着呢");
            }
        }, "王五");
        thread1.start();
        thread2.start();
    }
}
```

同学们可以观察下，是否注释掉 Thread.yield() 这句话，现象是否相同。

1. yield() 只是让出 CPU，并不会改变自己的状态。也就上面途中，我从柜台前站起，又重新去排队去了
2. 因为李四总是无私的让出座位，王五并不让座位，所以会导致王五的打印更多。

4. 多线程带来的的风险-线程安全（重点）

4.1 观察线程不安全

```
public class ThreadDemo {
    private static class Counter {
        private long n = 0;

        public void increment() {
            n++;
        }

        public void decrement() {
            n--;
        }

        public long value() {
            return n;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        final int COUNT = 1000_0000;
        Counter counter = new Counter();

        Thread thread = new Thread(() -> {
            for (int i = 0; i < COUNT; i++) {
                counter.increment();
            }
        }, "李四");

        thread.start();
        for (int i = 0; i < COUNT; i++) {
            counter.decrement();
        }
        thread.join();

        // 期望最终结果应该是 0
        System.out.println(counter.value());
    }
}
```

大家观察下是否适用多线程的现象是否一致？同时尝试思考下为什么会有这样的现象发生呢？

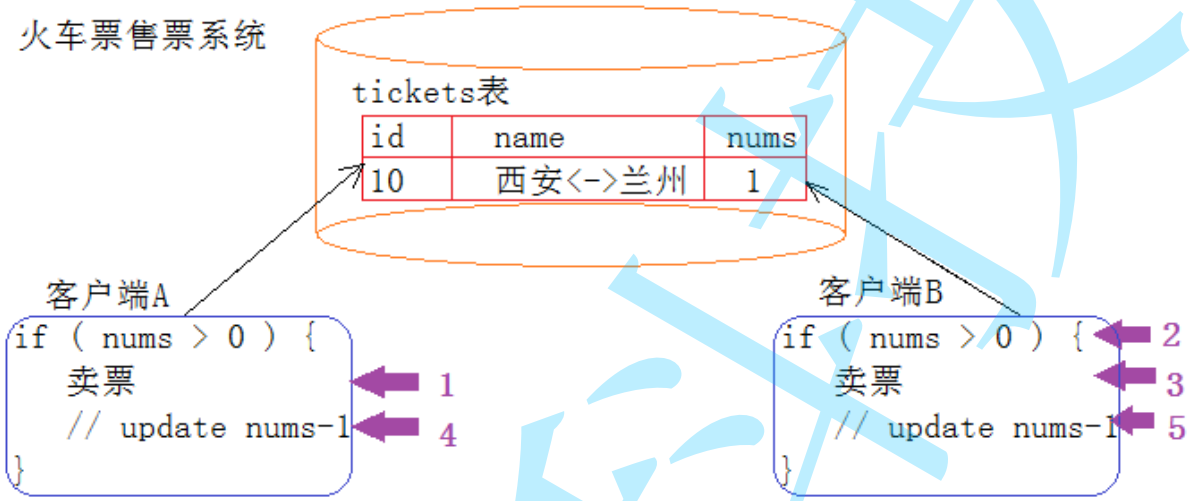
4.2 线程安全的概念

想给出一个线程安全的确切定义是复杂的，但我们可以这样认为：

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。

4.3 线程不安全的原因

4.3.1 原子性



当客户端A检查还有一张票时，将票卖掉，还没有执行更新数据库时，客户端B检查了票数，发现大于0，于是又卖了一次票。然后A将票数更新回数据库。这是就出现了同一张票被卖了两次。

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。

一条 java 语句不一定是原子的，也不一定只是一条指令

比如刚才我们看到的 n++，其实是由三步操作组成的：

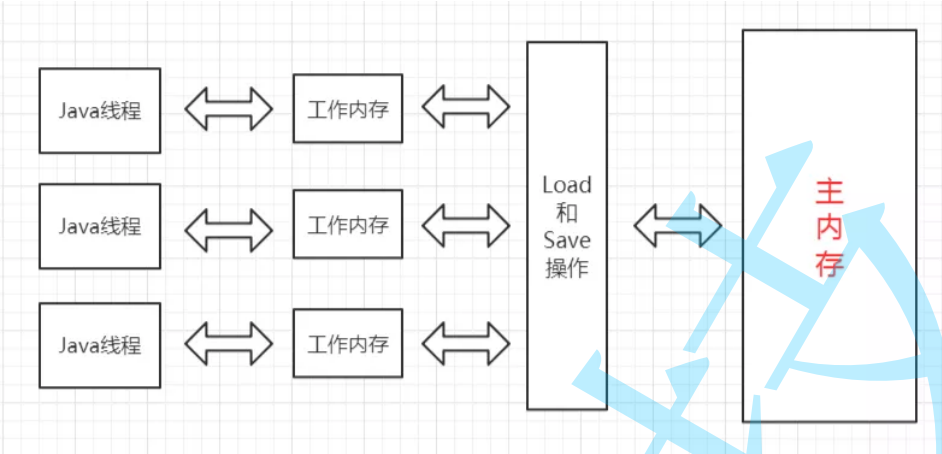
1. 从内存把数据读到 CPU
2. 进行数据更新
3. 把数据写回到 CPU

不保证原子性会给多线程带来什么问题

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。

4.3.2 可见性

主内存-工作内存



为了提高效率，JVM在执行过程中，会尽可能的将数据在工作内存中执行，但这样会造成一个问题，共享变量在多线程之间不能及时看到改变，这个就是可见性问题。

4.3.3 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

```
public class ThreadDemo {
    private static class Counter {
        private int n1 = 0;
        private int n2 = 0;
        private int n3 = 0;
        private int n4 = 0;
        private int n5 = 0;
        private int n6 = 0;
        private int n7 = 0;
        private int n8 = 0;
        private int n9 = 0;
        private int n10 = 0;

        public void write() {
            n1 = 1;
            n2 = 2;
            n3 = 3;
            n4 = 4;
            n5 = 5;
            n6 = 6;
            n7 = 7;
            n8 = 8;
            n9 = 9;
            n10 = 10;
        }
```

```
        public void read() {
            System.out.println("n1 = " + n1);
            System.out.println("n2 = " + n2);
            System.out.println("n3 = " + n3);
            System.out.println("n4 = " + n4);
            System.out.println("n5 = " + n5);
            System.out.println("n6 = " + n6);
            System.out.println("n7 = " + n7);
            System.out.println("n8 = " + n8);
            System.out.println("n9 = " + n9);
            System.out.println("n10 = " + n10);
        }
    }
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread thread1 = new Thread(() -> {
            counter.read();
        }, "读");
        Thread thread2 = new Thread(() -> {
            counter.write();
        }, "写");
        thread1.start();
        thread2.start();
    }
}
```

代码重排序会给多线程带来什么问题

刚才那个例子中，单线程情况是没问题的，优化是正确的，但在多线程场景下就有问题了，什么问题呢。可能快递是在你写作业的10分钟内被另一个线程放过来的，或者被人变过了，如果指令重排序了，代码就会是错误的。

1.
    1.

4.5 解决之前的线程不安全问题

这里用到的机制，我们马上会给大家解释。

```
public class ThreadDemo {
    private static class Counter {
        private long n = 0;

        public synchronized void increment() {
            n++;
        }

        public synchronized void decrement() {
            n--;
        }

        public synchronized long value() {
            return n;
        }
    }
```

```
    public static void main(String[] args) throws InterruptedException {
        final int COUNT = 1000_0000;
        Counter counter = new Counter();

        Thread thread = new Thread(() -> {
            for (int i = 0; i < COUNT; i++) {
                counter.increment();
            }
        }, "李四");

        thread.start();
        for (int i = 0; i < COUNT; i++) {
            counter.decrement();
        }
        thread.join();

        // 期望最终结果应该是 0
        System.out.println(counter.value());
    }
}
```

5. synchronized 关键字-监视器锁`monitor lock`

synchronized的底层是使用操作系统的mutex lock实现的。

- 当线程释放锁时，JMM会把该线程对应的工作内存中的共享变量刷新到主内存中
- 当线程获取锁时，JMM会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须从主内存中读取共享变量

synchronized用的锁是存在Java对象头里的。

synchronized同步快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；

同步块在已进入的线程执行完之前，会阻塞后面其他线程的进入。

锁的 SynchronizedDemo 对象

```
public class SynchronizedDemo {
    public synchronized void methond() {
    }
    public static void main(String[] args) {
        SynchronizedDemo demo = new SynchronizedDemo();
        demo.method();  // 进入方法会锁 demo 指向对象中的锁；出方法会释放 demo 指向的对象
中的锁
    }
}
```

锁的 SynchronizedDemo 类的对象

```
public class SynchronizedDemo {
    public synchronized static void methond() {
    }
    public static void main(String[] args) {
        method();   // 进入方法会锁 SynchronizedDemo.class 指向对象中的锁；出方法会释放
SynchronizedDemo.class 指向的对象中的锁
    }
}
```

明确锁的对象

```
public class SynchronizedDemo {
    public void methond() {
        // 进入代码块会锁 this 指向对象中的锁；出代码块会释放 this 指向的对象中的锁
        synchronized (this) {

        }
    }
    public static void main(String[] args) {
        SynchronizedDemo demo = new SynchronizedDemo();
        demo.method();
    }
}
```

```
public class SynchronizedDemo {
    public void methond() {
        // 进入代码块会锁 SynchronizedDemo.class 指向对象中的锁；出代码块会释放
SynchronizedDemo.class 指向的对象中的锁
        synchronized (SynchronizedDemo.class) {

        }
    }
    public static void main(String[] args) {
        SynchronizedDemo demo = new SynchronizedDemo();
        demo.method();
    }
}
```

示例

```
class MyThread implements Runnable {
    private int ticket = 1000 ; // 一共十张票
```

```
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            // 在同一时刻，只允许一个线程进入代码块处理
            synchronized(this) { // 表示为程序逻辑上锁
            if(this.ticket>0) { // 还有票
                    try {
                        Thread.sleep(20);
                    } catch (InterruptedException e) {
                    }
                    System.out.println(Thread.currentThread().getName()+",还有"
+this.ticket -- +" 张票");
                }
            }
        }
    }
}
public class TestDemo {
    public static void main(String[] args) {
        MyThread mt = new MyThread() ;
        Thread t1 = new Thread(mt,"黄牛A");
        Thread t2 = new Thread(mt,"黄牛B");
        Thread t3 = new Thread(mt,"黄牛C");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

我们重点要理解，synchronized 锁的是什么

6. volatile 关键字

修饰的共享变量，可以保证可见性，部分保证顺序性

```
class ThraedDemo {
    private volatile int n;
}
```

7. 通信-对象的等待集`wait set`

1.wait()的作用是让当前线程进入等待状态，同时，wait()也会让当前线程释放它所持有的锁。“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法”，当前线程被唤醒(进入“就绪状态”)

2.notify()和notifyAll()的作用，则是唤醒当前对象上的等待线程；notify()是唤醒单个线程，而notifyAll()是唤醒所有的线程。

3.wait(long timeout)让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的notify()方法或notifyAll() 方法，或者超过指定的时间量”，当前线程被唤醒(进入“就绪状态”)。

7.1 wait()方法**

其实wait()方法就是使线程停止运行。

1. 方法wait()的作用是使当前执行代码的线程进行等待，wait()方法是Object类的方法，该方法是用来将当前线程置入“预执行队列”中，并且在wait()所在的代码处停止执行，直到接到通知或被中断为止。
2. wait()方法只能在同步方法中或同步块中调用。如果调用wait()时，没有持有适当的锁，会抛出异常。
3. wait()方法执行后，当前线程释放锁，线程与其它线程竞争重新获取锁。

范例：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("等待已过...");
        }
        System.out.println("main方法结束...");
    }
```

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()。

7.2 notify()方法

notify方法就是使停止的线程继续运行。

1. 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，则有线程规划器随机挑选出一个呈wait状态的线程。
2. 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。

范例：使用notify()方法唤醒线程

```
package bit.java.thread;

class MyThread implements Runnable {
    private boolean flag;
    private Object obj;

    public MyThread(boolean flag, Object obj) {
        super();
        this.flag = flag;
        this.obj = obj;
    }

    public void waitMethod() {
        synchronized (obj) {
            try {
                while (true) {
```

```
                    System.out.println("wait()方法开始.. " +
Thread.currentThread().getName());
                    obj.wait();
                    System.out.println("wait()方法结束.. " +
Thread.currentThread().getName());
                    return;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    public void notifyMethod() {
        synchronized (obj) {
            try {
                System.out.println("notify()方法开始.. " +
Thread.currentThread().getName());
                obj.notify();
                System.out.println("notify()方法结束.. " +
Thread.currentThread().getName());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void run() {
        if (flag) {
            this.waitMethod();
        } else {
            this.notifyMethod();
        }
    }

}

public class TestThread {
    public static void main(String[] args) throws InterruptedException {
        Object object = new Object();
        MyThread waitThread = new MyThread(true, object);
        MyThread notifyThread = new MyThread(false, object);
        Thread thread1 = new Thread(waitThread, "wait线程");
        Thread thread2 = new Thread(notifyThread, "notify线程");
        thread1.start();
        Thread.sleep(1000);
        thread2.start();
        System.out.println("main方法结束!!");
    }
}
```

从结果上来看第一个线程执行的是一个waitMethod方法，该方法里面有个死循环并且使用了wait方法进入等待状态将释放锁，如果这个线程不被唤醒的话将会一直等待下去，这个时候第二个线程执行的是notifyMethod方法，该方法里面执行了一个唤醒线程的操作，并且一直将notify的同步代码块执行完毕之后才会释放锁然后继续执行wait结束打印语句。

注意：wait，notify必须使用在synchronized同步方法或者代码块内。

7.3 notifyAll()方法

以上讲解了notify方法只是唤醒某一个等待线程，那么如果有多个线程都在等待中怎么办呢，这个时候就可以使用notifyAll方法可以一次唤醒所有的等待线程，看示例。

范例：使用notifyAll()方法唤醒所有等待线程

```
package bit.java.thread;

class MyThread implements Runnable {
    private boolean flag;
    private Object obj;

    public MyThread(boolean flag, Object obj) {
        super();
        this.flag = flag;
        this.obj = obj;
    }

    public void waitMethod() {
        synchronized (obj) {
            try {
                while (true) {
                    System.out.println("wait()方法开始.. " +
Thread.currentThread().getName());
                    obj.wait();
                    System.out.println("wait()方法结束.. " +
Thread.currentThread().getName());
                    return;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    public void notifyMethod() {
        synchronized (obj) {
            try {
                System.out.println("notifyAll()方法开始.. " +
Thread.currentThread().getName());
                obj.notifyAll();
                System.out.println("notifyAll()方法结束.. " +
Thread.currentThread().getName());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void run() {
        if (flag) {
            this.waitMethod();
        } else {
            this.notifyMethod();
```

```
        }
    }
}

public class TestThread {
    public static void main(String[] args) throws InterruptedException {
        Object object = new Object();
        MyThread waitThread1 = new MyThread(true, object);
        MyThread waitThread2 = new MyThread(true, object);
        MyThread waitThread3 = new MyThread(true, object);
        MyThread notifyThread = new MyThread(false, object);
        Thread thread1 = new Thread(waitThread1, "wait线程A");
        Thread thread2 = new Thread(waitThread2, "wait线程B");
        Thread thread3 = new Thread(waitThread3, "wait线程C");
        Thread thread4 = new Thread(notifyThread, "notify线程");
        thread1.start();
        thread2.start();
        thread3.start();
        Thread.sleep(1000);
        thread4.start();
        System.out.println("main方法结束!!");
    }
}
```

现在可以看出notifyAll确实是唤醒了所有线程了。

注意：唤醒线程不能过早，如果在还没有线程在等待中时，过早的唤醒线程，这个时候就会出现先唤醒，在等待的效果了。这样就没有必要在去运行wait方法了。

7.3 注意点

wait 执行时需要对要 wait

7.4 wait 和 sleep 的对比（面试题）

其实理论上 wait 和 sleep 完全是没有可比性的，因为一个是用于线程之间的通信的，一个是让线程阻塞一段时间，唯一的相同点就是都可以让线程放弃执行一段时间。用生活中的例子说的话就是婚礼时会吃糖，和家里自己吃糖之间有差别。说白了放弃线程执行只是 wait 的一小段现象。

当然为了面试的目的，我们还是总结下：

1. wait 之前需要请求锁，而wait执行时会先释放锁，等被唤醒时再重新请求锁。这个锁是 wait 对像上的 monitor lock
2. sleep 是无视锁的存在的，即之前请求的锁不会释放，没有锁也不会请求。
3. wait 是 Object 的方法
4. sleep 是 Thread 的静态方法

9. 多线程案例

9.1 单例模式

9.1.1 饿汉模式

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

9.1.2 懒汉模式-单线程版

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

9.1.3 懒汉模式-多线程版-性能低

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

9.1.4 懒汉模式-多线程版-二次判断-性能高

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

9.1 阻塞式队列

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，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。这个阻塞队列就是用来给生产者和消费者解耦的。

```
import java.util.Random;

public class MessageQueue {
    private int[] items;
    private int putIndex;
    private int takeIndex;
    private volatile int size;

    public MessageQueue(int capacity) {
        items = new int[capacity];
        putIndex = 0;
        takeIndex = 0;
        size = 0;
    }

    public void put(int message) throws InterruptedException {
        do {
            synchronized (this) {
                if (size < items.length) {
                    items[putIndex] = message;
                    putIndex = (putIndex + 1) % items.length;
                    size++;
                    notifyAll();
                    return;
                }
            }

            while (size == items.length) {
                synchronized (this) {
                    if (size == items.length) {
                        wait();
                    }
                }
            }
        } while (true);
    }

    public int take() throws InterruptedException {
        do {
            synchronized (this) {
                if (size > 0) {
                    int message = items[takeIndex];
                    takeIndex = (takeIndex + 1) % items.length;
                    size--;
                    notifyAll();
                    return message;
                }
            }
```

```
            while (size == 0) {
                synchronized (this) {
                    if (size == 0) {
                        wait();
                    }
                }
            }
        } while (true);
    }

    public synchronized int size() {
        return size;
    }

    public static void main(String[] args) {
        MessageQueue queue = new MessageQueue();

        Thread producer = new Thread(() -> {
            Random random = new Random();
            for (int i = 0; i < 100_0000_0000; i++) {
                queue.put(random.nextInt(100000));
            }
        }, "生产者");

        Thread customer = new Thread(() -> {
            while (true) {
                int message = queue.take();
                System.out.println(message);
            }
        }, "消费者");
    }
}
```

9.2 定时器

```
import java.util.concurrent.PriorityBlockingQueue;

public class Timer {
    private static class TimerTask implements Comparable<TimerTask> {
        private Runnable command;
        private long time;

        private TimerTask(Runnable command, long after) {
            this.command = command;
            this.time = System.currentTimeMillis() + after;
        }

        private void run() {
            command.run();
        }

        @Override
        public int compareTo(TimerTask o) {
            return (int)(time - o.time);
        }
    }
```

```
    private PriorityBlockingQueue<TimerTask> queue = new PriorityBlockingQueue<>
();
    private Object mailbox = new Object();

    private class Worker extends Thread {
        @Override
        public void run() {
            while (true) {
                try {
                    TimerTask task = queue.take();
                    long ms = System.currentTimeMillis();
                    if (task.time > ms) {
                        queue.put(task);
                        synchronized (mailbox) {
                            mailbox.wait(task.time - ms);
                        }
                    } else {
                        task.run();
                    }
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    }

    public Timer() {
        Worker worker = new Worker();
        worker.start();
    }

    public void schedule(Runnable command, long after) {
        TimerTask task = new TimerTask(command, after);
        queue.offer(task);
        synchronized (mailbox) {
            mailbox.notify();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Timer timer = new Timer();
        Runnable command = new Runnable() {
            @Override
            public void run() {
                System.out.println("我来了");
                timer.schedule(this, 3 * 1000);
            }
        };
        timer.schedule(command, 3 * 1000);
    }
}
```

9.3 线程池

为什么需要线程池呢?

想象这么一个场景:

在学校附近新开了一家快递店，老板很精明，想到一个与众不同的办法来经营。店里没有雇人，而是每次有业务来了，就现场找一名同学过来把快递送了，然后解雇同学。这个类比我们平时来一个任务，起一个线程进行处理的模式。

很快老板发现问题来了，每次招聘 + 解雇同学的成本还是非常高的。老板还是很善于变通的，知道了为什么大家都要雇人了，所以指定了一个指标，公司业务人员会扩张到 3 个人，但还是随着业务逐步雇人。于是再有业务来了，老板就看，如果现在公司还没 3 个人，就雇一个人去送快递，否则只是把业务放到一个本本上，等着 3 个快递人员空闲的时候去处理。这个就是我们要带出的线程池的模式。

线程池最大的好处就是减少每次启动、销毁线程的损耗。

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class ThreadPool {
    private static class Worker extends Thread {
    }

    private BlockingQueue<Runnable> jobQueue;
    private int nCurrentThreads;
    private int nThreads;
    private Worker[] workers;

    public ThreadPool(int nThreads, int nCachedJobs) {
        this.jobQueue = new ArrayBlockingQueue<>(nCachedJobs);
        this.nCurrentThreads = 0;
        this.nThreads = nThreads;
        this.workers = new Worker[nThreads];
    }

    public void execute(Runnable command) throws InterruptedException {
        if (nCurrentThreads < nThreads) {
            Worker worker = new Worker();
            workers[nCurrentThreads++] = worker;
            worker.start();
        } else {
            jobQueue.put(command);
        }
    }
}
```

10. 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

11. 对比线程和进程

11.1 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。

11.2 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。

内容重点总结

- 多线程
- 线程安全

课后作业

- 总结原子性、可见性、顺序性
- 编写线程的实践