

# 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

## 1. 认识线程 (Thread)

### 1.1 概念

#### 1) 线程是什么

一个线程就是一个 "执行流". 每个线程之间都可以按照顺序执行自己的代码. 多个线程之间 "同时" 执行着多份代码.

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）。

#### 2) 为啥要有线程

首先, "并发编程" 成为 "刚需".

- 单核 CPU 的发展遇到了瓶颈. 要想提高算力, 就需要多核 CPU. 而并发编程能更充分利用多核 CPU 资源.
- 有些任务场景需要 "等待 IO", 为了让等待 IO 的时间能够去做一些其他的工作, 也需要用到并发编程.

其次, 虽然多进程也能实现 并发编程, 但是线程比进程更轻量.

- 创建线程比创建进程更快.
- 销毁线程比销毁进程更快.
- 调度线程比调度进程更快.

最后, 线程虽然比进程轻量, 但是人们还不满足, 于是就有了 "线程池"(ThreadPool) 和 "协程"(Coroutine)

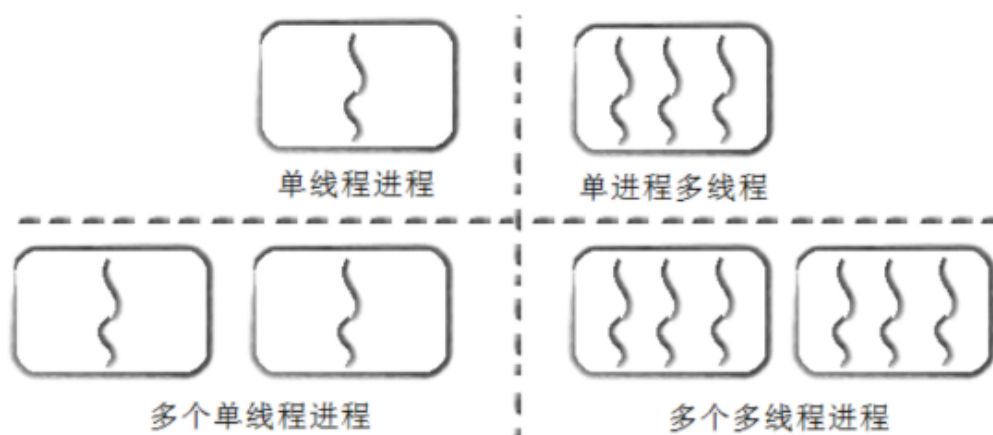
关于线程池我们后面再介绍. 关于协程的话题我们此处暂时不做过多讨论.

### 3) 进程和线程的区别

- 进程是包含线程的. **每个进程至少有一个线程存在, 即主线程。**
- 进程和进程之间不共享内存空间. 同一个进程的线程之间共享同一个内存空间.

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。



### 4) Java 的线程 和 操作系统线程 的关系

线程是操作系统中的概念. 操作系统内核实现了线程这样的机制, 并且对用户层提供了一些 API 供用户使用(例如 Linux 的 pthread 库).

Java 标准库中 Thread 类可以视为是对操作系统提供的 API 进行了进一步的抽象和封装.

## 1.2 第一个多线程程序

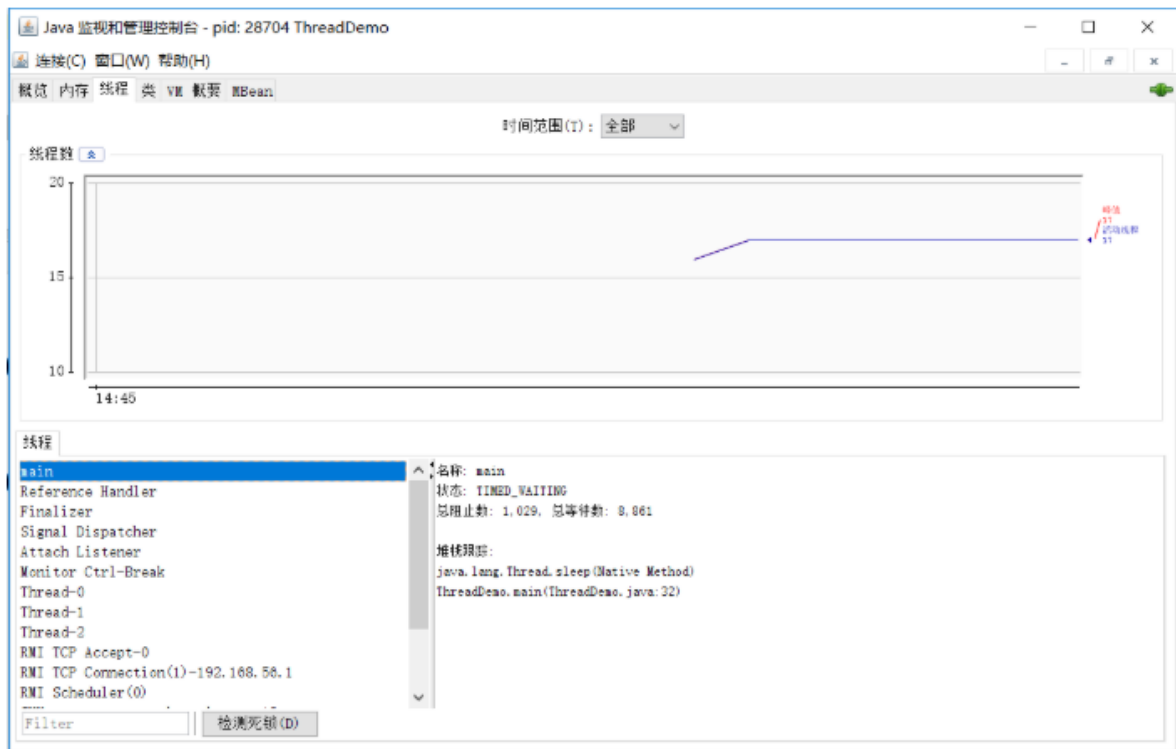
感受多线程程序和普通程序的区别:

- 每个线程都是一个独立的执行流
- 多个线程之间是 "并发" 执行的.

```
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
                }
            }
        }
    }
}
```





## 1.3 创建线程

### 方法1 继承 Thread 类

1) 继承 Thread 来创建一个线程类.

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

2) 创建 MyThread 类的实例

```
MyThread t = new MyThread();
```

3) 调用 start 方法启动线程

```
t.start(); // 线程开始运行
```

### 方法2 实现 Runnable 接口

1) 实现 Runnable 接口

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

2) 创建 Thread 类实例, 调用 Thread 的构造方法时将 Runnable 对象作为 target 参数.

```
Thread t = new Thread(new MyRunnable());
```

3) 调用 start 方法

```
t.start(); // 线程开始运行
```

**对比上面两种方法:**

- 继承 Thread 类, 直接使用 this 就表示当前线程对象的引用.
- 实现 Runnable 接口, this 表示的是 MyRunnable 的引用. 需要使用 Thread.currentThread()

## 其他变形

- 匿名内部类创建 Thread 子类对象

```
// 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");
    }
};
```

- 匿名内部类创建 Runnable 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");
    }
});
```

- lambda 表达式创建 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");
});
```

## 1.4 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。

- 使用 `System.nanoTime()` 可以记录当前系统的 纳秒 级时间戳。
- `serial` 串行的完成一系列运算. `concurrency` 使用两个线程并行的完成同样的运算.

```
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++) {
```

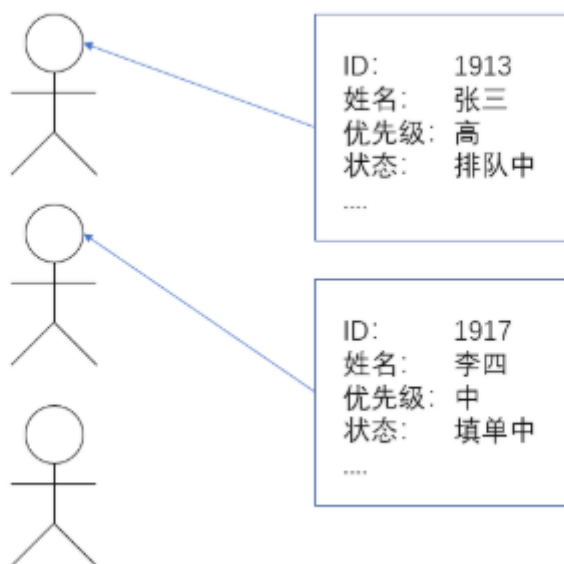
```
        b--;  
    }  
    long end = System.nanoTime();  
    double ms = (end - begin) * 1.0 / 1000 / 1000;  
    System.out.printf("串行: %f 毫秒%n", ms);  
}  
}
```

并发：399.651856 毫秒  
串行：720.616911 毫秒

## 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，分好的组即为线程组，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());

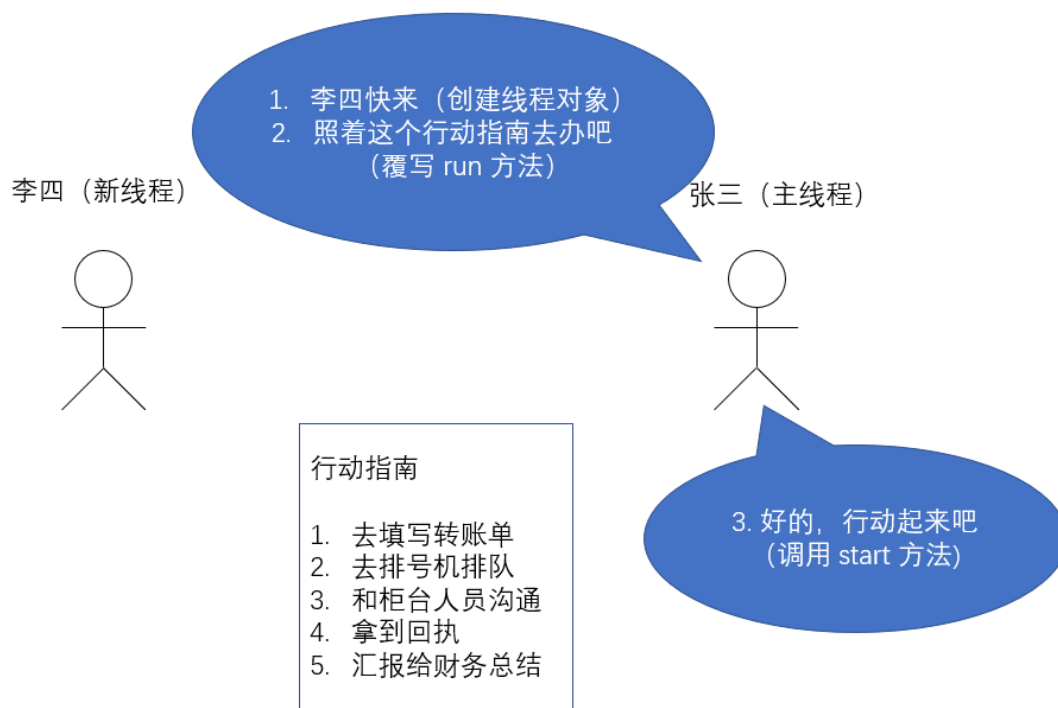
        thread.start();
        while (thread.isAlive()) {}
        System.out.println(Thread.currentThread().getName()
            + ": 状态: " + thread.getState());
    }
}

```

## 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。



调用 start 方法, 才真的在操作系统的底层创建出一个线程.

## 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:** 使用自定义的变量来作为标志位.

- 需要给标志位上加 `volatile` 关键字(这个关键字的功能后面介绍).

```
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:** 使用 `Thread.interrupted()` 或者 `Thread.currentThread().isInterrupted()` 代替自定义标志位.

Thread 内部包含了一个 `boolean` 类型的变量作为线程是否被中断的标记.

方法	说明
<code>public void interrupt()</code>	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
<code>public static boolean interrupted()</code>	判断当前线程的中断标志位是否设置，调用后清除标志位
<code>public boolean isInterrupted()</code>	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

- 使用 `Thread` 对象的 `interrupted()` 方法通知线程结束。

```

public class ThreadDemo {
    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!");
                    // 注意此处的 break
                    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();
    }
}

```

`Thread` 收到通知的方式有两种：

1. 如果线程因为调用 `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
  - 当出现 `InterruptedException` 的时候，要不要结束线程取决于 `catch` 中代码的写法。可以选择忽略这个异常，也可以跳出循环结束线程。

2. 否则，只是内部的一个中断标志被设置，thread 可以通过

- Thread.interrupted() 判断当前线程的中断标志被设置，**清除中断标志**
- Thread.currentThread().isInterrupted() 判断指定线程的中断标志被设置，**不清除中断标志**

这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 观察标志位是否清除

标志位是否清除，就类似于一个开关。

Thread.interrupted() 相当于按下开关，开关自动弹起来了。这个称为 "清除标志位"

Thread.currentThread().isInterrupted() 相当于按下开关之后，开关弹不起来，这个称为 "不清除标志位"。

- 使用 Thread.interrupted()，线程中断会清除标志位。

```
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
false
false
```

- 使用 Thread.currentThread().isInterrupted()，线程中断标记位不会清除。

```
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
true

```

## 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 ThreadDemo {
    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

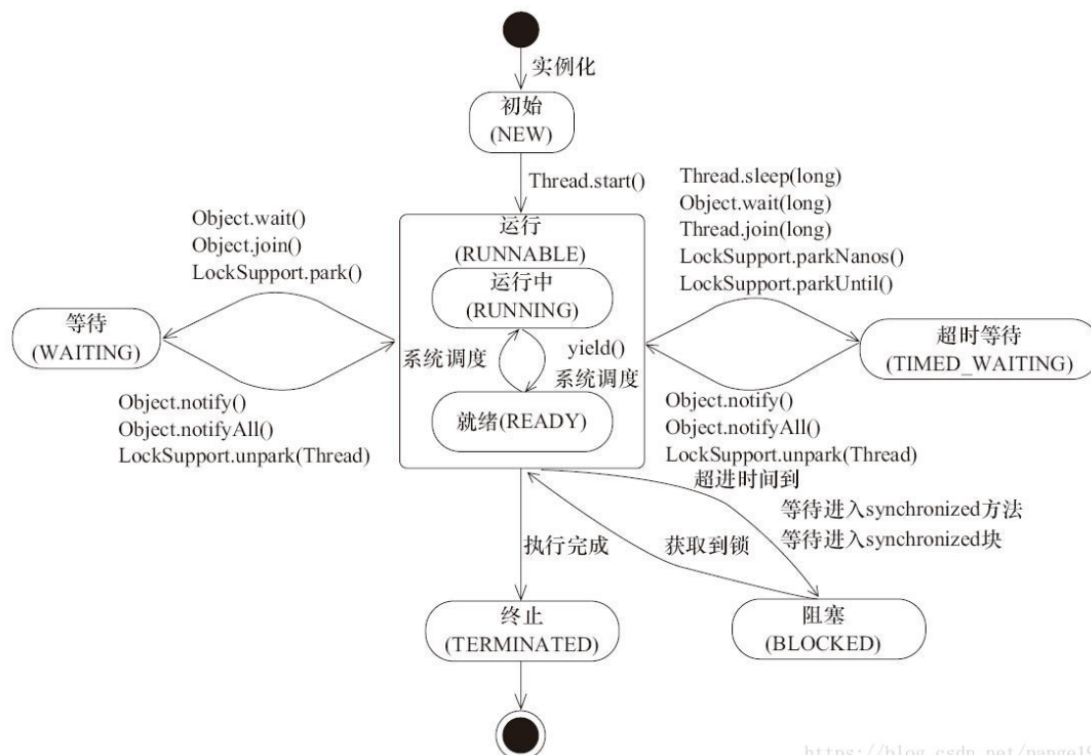
### 3.1 观察线程的所有状态

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);  
        }  
    }  
}
```

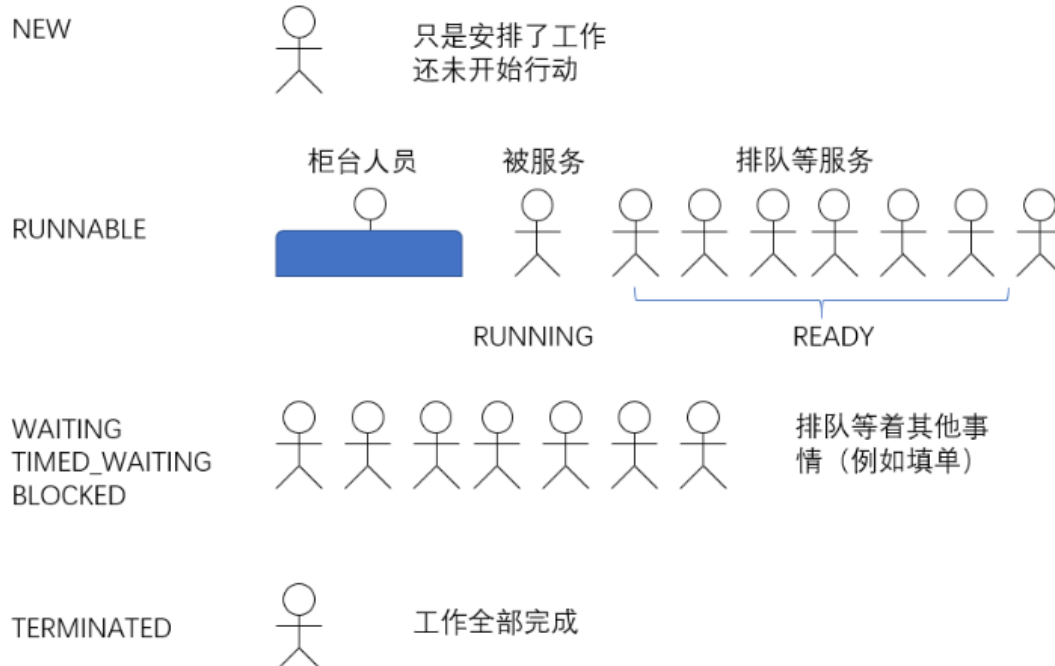
- NEW: 安排了工作, 还未开始行动
- RUNNABLE: 可工作的. 又可以分成正在工作中和即将开始工作.
- BLOCKED: 这几个都表示排队等着其他事情
- WAITING: 这几个都表示排队等着其他事情
- TIMED\_WAITING: 这几个都表示排队等着其他事情
- TERMINATED: 工作完成了.

### 3.2 线程状态和状态转移的意义



<https://blog.csdn.net/pange1991>

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；**

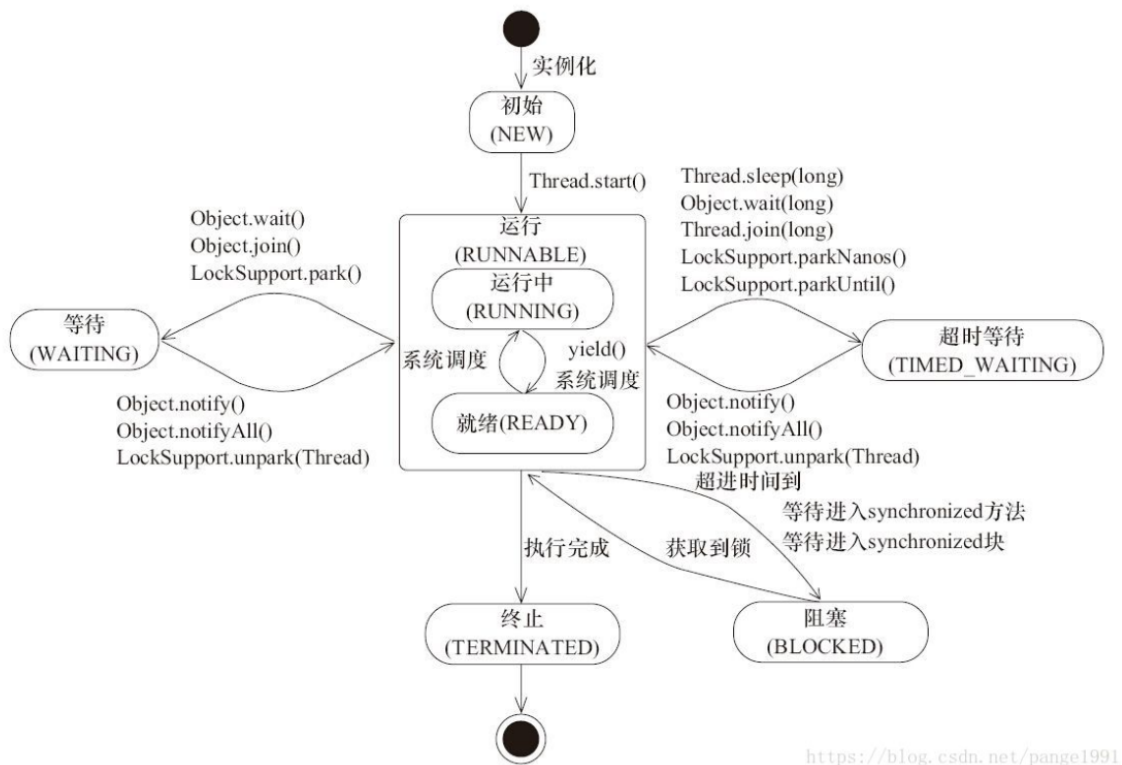
当李四、王五因为一些事情需要去忙，例如需要填写信息、回家取证件、发呆一会等等时，进入 `BLOCKED`、`WAITING`、`TIMED_WAITING` 状态，至于这些状态的细分，我们以后再详解；

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

- 使用 `isAlive` 方法判定线程的存活状态.

```
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 static void main(String[] args) {
    final Object object = new Object();

    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (object) {
                while (true) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }) {
    }
}
```

```

    }
    }
}
}, "t1");
t1.start();

Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        synchronized (object) {
            System.out.println("hehe");
        }
    }
}, "t2");
t2.start();
}

```

使用 jconsole 可以看到 t1 的状态是 TIMED\_WAITING , t2 的状态是 BLOCKED

修改上面的代码, 把 t1 中的 sleep 换成 wait

```

public static void main(String[] args) {
    final Object object = new Object();

    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (object) {
                try {
                    // [修改这里就可以了!!!!]
                    // Thread.sleep(1000);
                    object.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }, "t1");
    ...
}

```

使用 jconsole 可以看到 t1 的状态是 WAITING

### 结论:

- BLOCKED 表示等待获取锁, WAITING 和 TIMED\_WAITING 表示等待其他线程发来通知.
- TIMED\_WAITING 线程在等待唤醒, 但设置了时限; WAITING 线程在无限等待唤醒

### 观察-3: yield() 大公无私, 让出 CPU

```

Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {

```

```

        while (true) {
            System.out.println("张三");
            // 先注释掉，再放开
            // Thread.yield();
        }
    }
}, "t1");
t1.start();

Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        while (true) {
            System.out.println("李四");
        }
    }
}, "t2");
t2.start();

```

可以看到:

1. 不使用 yield 的时候, 张三李四大概五五开
2. 使用 yield 时, 张三的数量远远少于李四

**结论:**

yield 不改变线程的状态, 但是会重新去排队.

## 4. 多线程带来的风险-线程安全 (重点)

### 4.1 观察线程不安全

```

static class Counter {
    public int count = 0;

    void increase() {
        count++;
    }
}

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

    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 50000; i++) {
            counter.increase();
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 50000; i++) {
            counter.increase();
        }
    });
    t1.start();
    t2.start();
}

```

```
t1.join();
t2.join();

system.out.println(counter.count);
}
```

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：

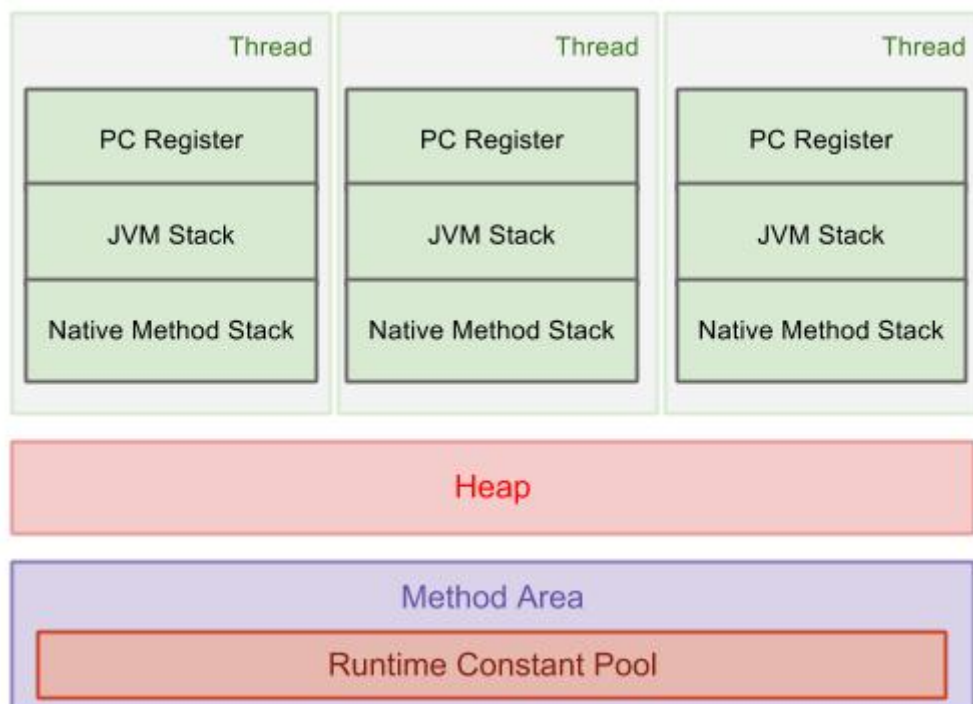
如果多线程环境下代码运行的结果是符合我们预期的，即在单线程环境应该的结果，则说这个程序是线程安全的。

## 4.3 线程不安全的原因

### 修改共享数据

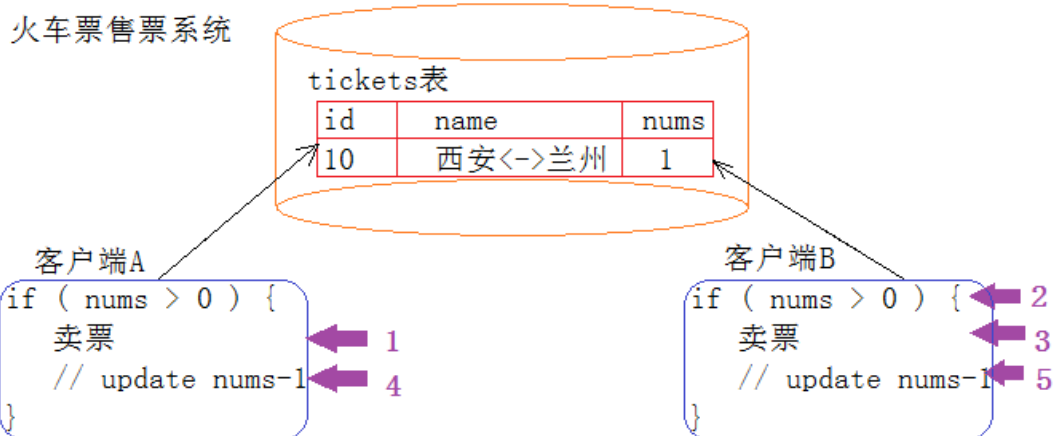
上面的线程不安全的代码中，涉及到多个线程针对 `counter.count` 变量进行修改。

此时这个 `counter.count` 是一个多个线程都能访问到的 "共享数据"



`counter.count` 这个变量就是在堆上，因此可以被多个线程共享访问。

### 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。于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。

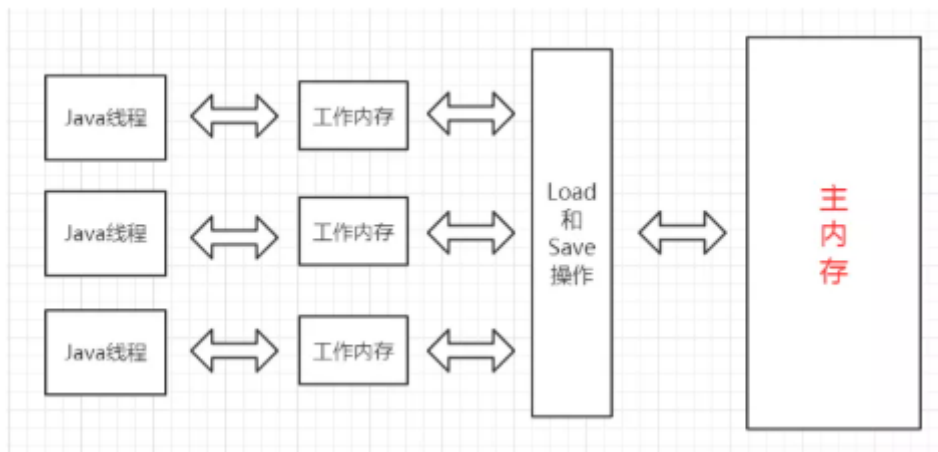
这点也和线程的抢占式调度密切相关。如果线程不是“抢占”的，就算没有原子性，也问题不大。

## 可见性

可见性指，一个线程对共享变量值的修改，能够及时地被其他线程看到。

**Java 内存模型 (JMM):** Java虚拟机规范中定义了Java内存模型。

目的是屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的并发效果。

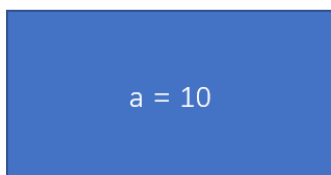


- 线程之间的共享变量存在 主内存 (Main Memory).
- 每一个线程都有自己的 "工作内存" (Working Memory) .
- 当线程要读取一个共享变量的时候, 会先把变量从主内存拷贝到工作内存, 再从工作内存读取数据.
- 当线程要修改一个共享变量的时候, 也会先修改工作内存中的副本, 再同步回主内存.

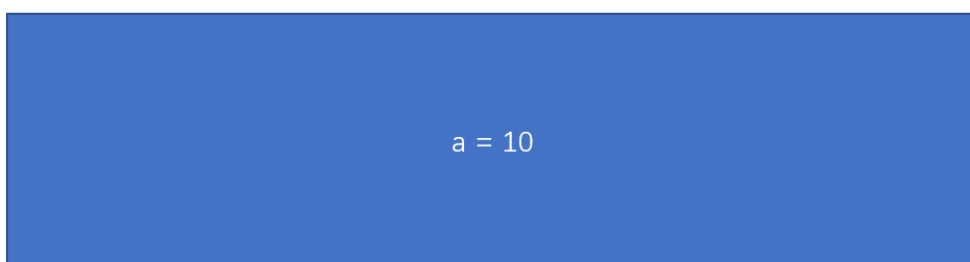
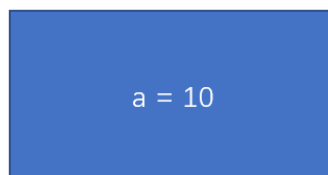
由于每个线程有自己的工作内存, 这些工作内存中的内容相当于同一个共享变量的 "副本". 此时修改线程 1 的工作内存中的值, 线程 2 的工作内存不一定会及时变化.

1) 初始情况下, 两个线程的工作内存内容一致.

线程1工作内存



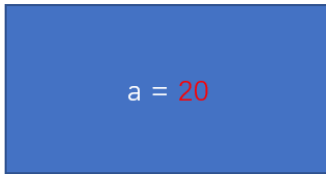
线程2工作内存



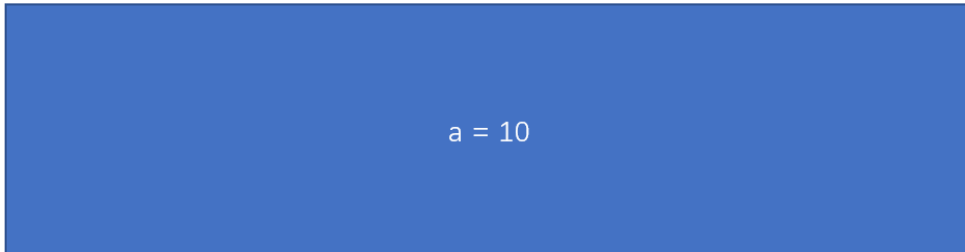
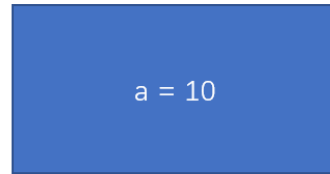
主内存

2) 一旦线程1 修改了 a 的值, 此时主内存不一定能及时同步. 对应的线程2 的工作内存的 a 的值也不一定能及时同步.

线程1工作内存



线程2工作内存



主内存

这个时候代码中就容易出现问题.

**此时引入了两个问题:**

- 为啥要整这么多内存?
- 为啥要这么麻烦的拷来拷去?

1) 为啥整这么多内存?

实际并没有这么多 "内存". 这只是 Java 规范中的一个术语, 是属于 "抽象" 的叫法.

所谓的 "主内存" 才是真正硬件角度的 "内存". 而所谓的 "工作内存", 则是指 CPU 的寄存器和高速缓存.

2) 为啥要这么麻烦的拷来拷去?

因为 CPU 访问自身寄存器的速度以及高速缓存的速度, 远远超过访问内存的速度(快了 3 - 4 个数量级, 也就是几千倍, 上万倍).

比如某个代码中要连续 10 次读取某个变量的值, 如果 10 次都从内存读, 速度是很慢的. 但是如果只是第一次从内存读, 读到的结果缓存到 CPU 的某个寄存器中, 那么后 9 次读数据就不必直接访问内存了. 效率就大大提高了.

那么接下来问题又来了, 既然访问寄存器速度这么快, 还要内存干啥??

答案就是一个字: 贵



## 来自贫穷的凝视

值得一提的是, 快和慢都是相对的. CPU 访问寄存器速度远远快于内存, 但是内存的访问速度又远远快于硬盘.

对应的, CPU 的价格最贵, 内存次之, 硬盘最便宜.

### 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

编译器对于指令重排序的前提是 "保持逻辑不发生变化". 这一点在单线程环境下比较容易判断, 但是在多线程环境下就没那么容易了, 多线程的代码执行复杂程度更高, 编译器很难在编译阶段对代码的执行效果进行预测, 因此激进的重排序很容易导致优化后的逻辑和之前不等价.

重排序是一个比较复杂的话题, 涉及到 CPU 以及编译器的一些底层工作原理, 此处不做过多讨论

## 4.4 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。

```
static class Counter {  
    public int count = 0;  
  
    synchronized void increase() {  
        count++;  
    }  
}  
  
public static void main(String[] args) throws InterruptedException {
```



```

final Counter counter = new Counter();

Thread t1 = new Thread() -> {
    for (int i = 0; i < 50000; i++) {
        counter.increase();
    }
};
Thread t2 = new Thread() -> {
    for (int i = 0; i < 50000; i++) {
        counter.increase();
    }
};
t1.start();
t2.start();
t1.join();
t2.join();

System.out.println(counter.count);
}

```

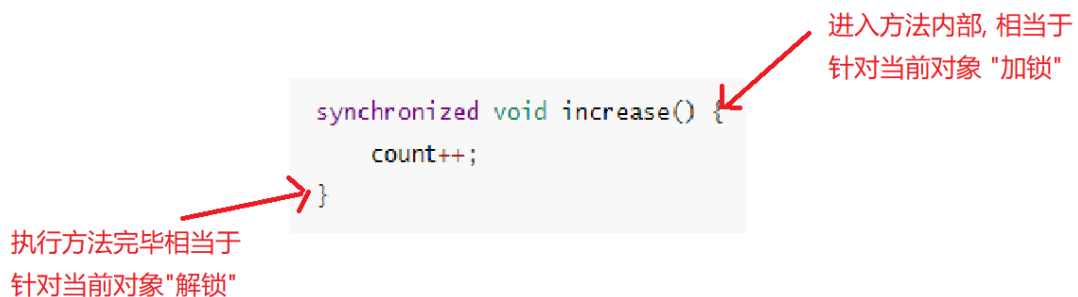
## 5. synchronized 关键字-监视器锁 monitor lock

### 5.1 synchronized 的特性

#### 1) 互斥

synchronized 会起到互斥效果, 某个线程执行到某个对象的 synchronized 中时, 其他线程如果也执行到同一个对象 synchronized 就会**阻塞等待**.

- 进入 synchronized 修饰的代码块, 相当于 **加锁**
- 退出 synchronized 修饰的代码块, 相当于 **解锁**



```

synchronized void increase() {
    count++;
}

```

进入方法内部, 相当于  
针对当前对象 "加锁"

执行方法完毕相当于  
针对当前对象 "解锁"

synchronized用的锁是存在Java对象头里的。

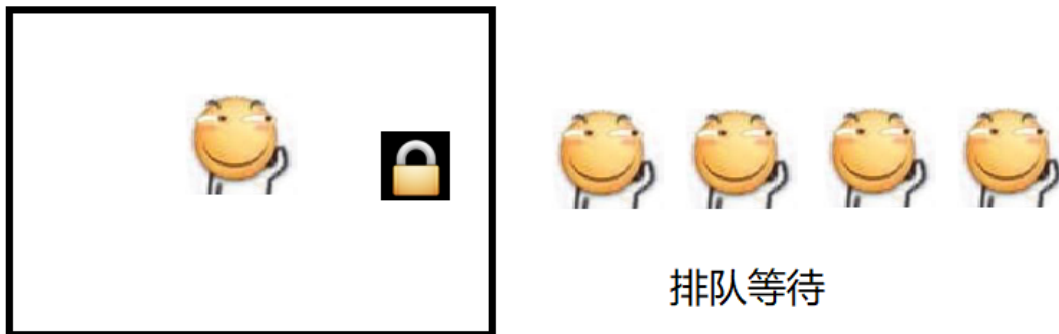


可以粗略理解成, 每个对象在内存中存储的时候, 都存有一块内存表示当前的 "锁定" 状态(类似于厕所的 "有人/无人").

如果当前是 "无人" 状态, 那么就可以使用, 使用时需要设为 "有人" 状态.

如果当前是 "有人" 状态, 那么其他人无法使用, 只能排队

## 对象



一个线程先上了锁, 其他线程只能等待这个线程释放

理解 "阻塞等待".

针对每一把锁, 操作系统内部都维护了一个等待队列. 当这个锁被某个线程占有的时候, 其他线程尝试进行加锁, 就加不上了, 就会阻塞等待, 一直等到之前的线程解锁之后, 由操作系统唤醒一个新的线程, 再来获取到这个锁.

**注意:**

- 上一个线程解锁之后, 下一个线程并不是立即就能获取到锁. 而是要靠操作系统来 "唤醒". 这也就是操作系统线程调度的一部分工作.
- 假设有 A B C 三个线程, 线程 A 先获取到锁, 然后 B 尝试获取锁, 然后 C 再尝试获取锁, 此时 B 和 C 都在阻塞队列中排队等待. 但是当 A 释放锁之后, 虽然 B 比 C 先来的, 但是 B 不一定就能获取到锁, 而是和 C 重新竞争, 并不遵守先来后到的规则.

synchronized的底层是使用操作系统的mutex lock实现的.

## 2) 刷新内存

synchronized 的工作过程:

1. 获得互斥锁
2. 从主内存拷贝变量的最新副本到工作的内存
3. 执行代码
4. 将更改后的共享变量的值刷新到主内存
5. 释放互斥锁

所以 `synchronized` 也能保证内存可见性. 具体代码参见后面 `volatile` 部分.

### 3) 可重入

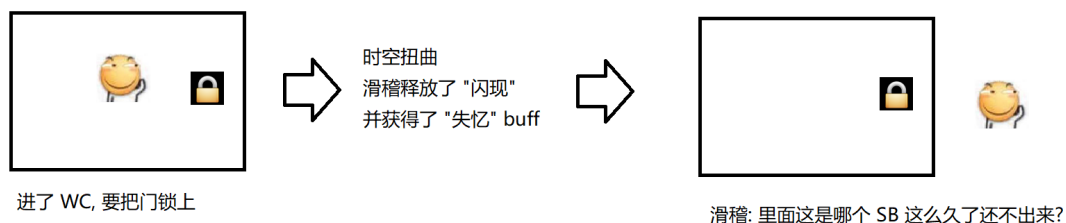
`synchronized` 同步块对同一条线程来说是可重入的, 不会出现自己把自己锁死的问题;

#### 理解 "把自己锁死"

一个线程没有释放锁, 然后又尝试再次加锁.

```
// 第一次加锁, 加锁成功
lock();
// 第二次加锁, 锁已经被占用, 阻塞等待.
lock();
```

按照之前对于锁的设定, 第二次加锁的时候, 就会阻塞等待. 直到第一次的锁被释放, 才能获取到第二个锁. 但是释放第一个锁也是由该线程来完成, 结果这个线程已经躺平了, 啥都不想干了, 也就无法进行解锁操作. 这时候就会 **死锁**.



这样的锁称为 **不可重入锁**.

Java 中的 `synchronized` 是 **可重入锁**, 因此没有上面的问题.

#### 代码示例

在下面的代码中,

- `increase` 和 `increase2` 两个方法都加了 `synchronized`, 此处的 `synchronized` 都是针对 `this` 当前对象加锁的.
- 在调用 `increase2` 的时候, 先加了一次锁, 执行到 `increase` 的时候, 又加了一次锁. (上个锁还没释放, 相当于连续加两次锁)

这个代码是完全没问题的. 因为 `synchronized` 是可重入锁.

```
static class Counter {
    public int count = 0;

    synchronized void increase() {
        count++;
    }

    synchronized void increase2() {
        increase();
    }
}
```

在可重入锁的内部, 包含了 "线程持有者" 和 "计数器" 两个信息.

- 如果某个线程加锁的时候, 发现锁已经被别人占用, 但是恰好占用的正是自己, 那么仍然可以继续获取到锁, 并让计数器自增.
- 解锁的时候计数器递减为 0 的时候, 才真正释放锁. (才能被别的线程获取到)

## 5.2 synchronized 使用示例

synchronized 本质上要修改指定对象的 "对象头". 从使用角度来看, synchronized 也势必要搭配一个具体的对象来使用.

### 1) 直接修饰普通方法: 锁的 SynchronizedDemo 对象

```
public class SynchronizedDemo {
    public synchronized void method() {
    }
}
```

### 2) 修饰静态方法: 锁的 SynchronizedDemo 类的对象

```
public class SynchronizedDemo {
    public synchronized static void method() {
    }
}
```

### 3) 修饰代码块: 明确指定锁哪个对象.

锁当前对象

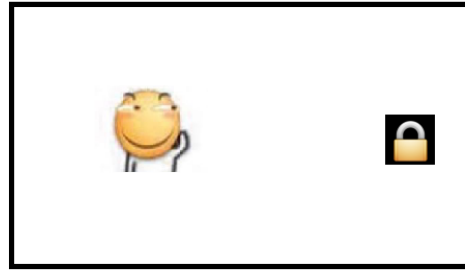
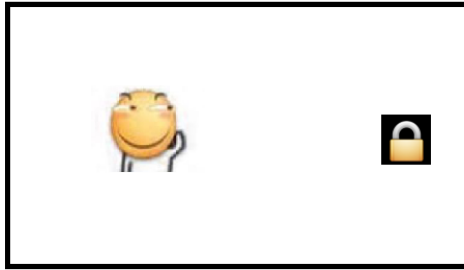
```
public class SynchronizedDemo {
    public void method() {
        synchronized (this) {
        }
    }
}
```

锁类对象

```
public class SynchronizedDemo {  
    public void method() {  
        synchronized (SynchronizedDemo.class) {  
  
        }  
    }  
}
```

我们重点要理解，**synchronized 锁的是什么**。两个线程竞争同一把锁，才会产生阻塞等待。

两个线程分别尝试获取两把不同的锁，不会产生竞争。



## 5.3 Java 标准库中的线程安全类

Java 标准库中很多都是线程不安全的。这些类可能会涉及到多线程修改共享数据，又没有任何加锁措施。

- ArrayList
- LinkedList
- HashMap
- TreeMap
- HashSet
- TreeSet
- StringBuilder

但是还有一些是线程安全的。使用了一些锁机制来控制。

- Vector (不推荐使用)
- Hashtable (不推荐使用)
- ConcurrentHashMap
- StringBuffer

```
@Override  
public synchronized StringBuffer append(Object obj) {  
    toStringCache = null;  
    super.append(String.valueOf(obj));  
    return this;  
}
```

StringBuffer 的核心方法都带有 `synchronized`。

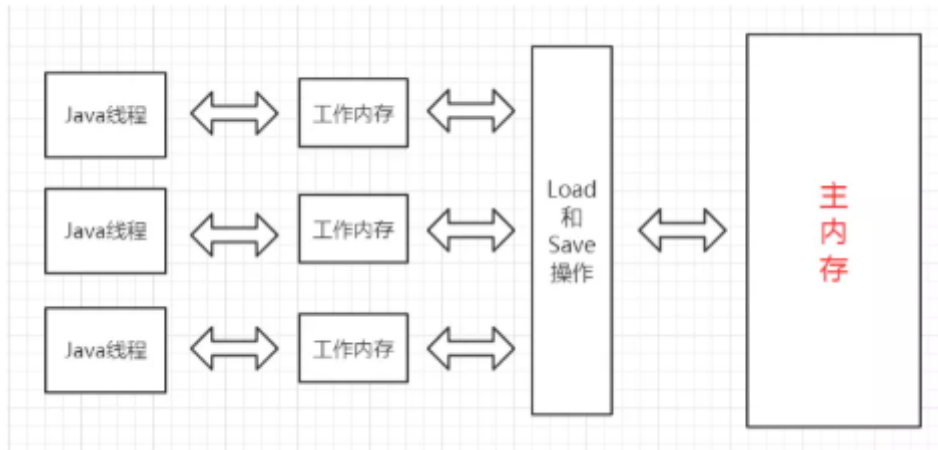
还有的虽然没有加锁, 但是不涉及 "修改", 仍然是线程安全的

- String

## 6. volatile 关键字

### volatile 能保证内存可见性

volatile 修饰的变量, 能够保证 "内存可见性".



代码在写入 volatile 修饰的变量的时候,

- 改变线程工作内存中volatile变量副本的值
- 将改变后的副本的值从工作内存刷新到主内存

代码在读取 volatile 修饰的变量的时候,

- 从主内存中读取volatile变量的最新值到线程的工作内存中
- 从工作内存中读取volatile变量的副本

前面我们讨论内存可见性时说了, 直接访问工作内存(实际是 CPU 的寄存器或者 CPU 的缓存), 速度非常快, 但是可能出现数据不一致的情况.

加上 volatile, 强制读写内存. 速度是慢了, 但是数据变的更准确了.



## 代码示例

在这个代码中

- 创建两个线程 t1 和 t2
- t1 中包含一个循环, 这个循环以 flag == 0 为循环条件.
- t2 中从键盘读入一个整数, 并把这个整数赋值给 flag.
- 预期当用户输入非 0 的值的时候, t1 线程结束.

```
static class Counter {
    public int flag = 0;
}

public static void main(String[] args) {
    Counter counter = new Counter();
    Thread t1 = new Thread(() -> {
        while (counter.flag == 0) {
            // do nothing
        }
        System.out.println("循环结束!");
    });

    Thread t2 = new Thread(() -> {
        Scanner scanner = new Scanner(System.in);
        System.out.println("输入一个整数:");
        counter.flag = scanner.nextInt();
    });

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

// 执行效果
// 当用户输入非0值时, t1 线程循环不会结束。(这显然是一个 bug)
```

t1 读的是自己工作内存中的内容.

当 t2 对 flag 变量进行修改, 此时 t1 感知不到 flag 的变化.

如果给 flag 加上 volatile

```
static class Counter {
    public volatile int flag = 0;
}

// 执行效果
// 当用户输入非0值时, t1 线程循环能够立即结束.
```

## volatile 不保证原子性

volatile 和 synchronized 有着本质的区别. synchronized 能够保证原子性, volatile 保证的是内存可见性.

## 代码示例

这个是最初的演示线程安全的代码.

- 给 increase 方法去掉 synchronized
- 给 count 加上 volatile 关键字.

```
static class Counter {
    volatile public int count = 0;

    void increase() {
        count++;
    }
}

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

    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 50000; i++) {
            counter.increase();
        }
    });
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 50000; i++) {
            counter.increase();
        }
    });
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    System.out.println(counter.count);
}
```

此时可以看到, 最终 count 的值仍然无法保证是 100000.

## synchronized 也能保证内存可见性

synchronized 既能保证原子性, 也能保证内存可见性.

对上面的代码进行调整:

- 去掉 flag 的 volatile
- 给 t1 的循环内部加上 synchronized, 并借助 counter 对象加锁.

```
static class Counter {
    public int flag = 0;
}

public static void main(String[] args) {
    Counter counter = new Counter();
    Thread t1 = new Thread(() -> {
        while (true) {
            synchronized (counter) {
```



```

        if (counter.flag != 0) {
            break;
        }
    }
    // do nothing
}
System.out.println("循环结束!");
});

Thread t2 = new Thread(() -> {
    Scanner scanner = new Scanner(System.in);
    System.out.println("输入一个整数:");
    counter.flag = scanner.nextInt();
});

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

```

## 7. wait 和 notify

由于线程之间是抢占式执行的, 因此线程之间执行的先后顺序难以预知.

但是实际开发中有时候我们希望合理的协调多个线程之间的执行先后顺序.



球场上的每个运动员都是独立的 "执行流", 可以认为是一个 "线程".

而完成一个具体的进攻得分动作, 则需要多个运动员相互配合, 按照一定的顺序执行一定的动作, 线程1 先 "传球", 线程2 才能 "扣篮".

完成这个协调工作, 主要涉及到三个方法

- wait() / wait(long timeout): 让当前线程进入等待状态.
- notify() / notifyAll(): 唤醒在当前对象上等待的线程.

**注意:** wait, notify, notifyAll 都是 Object 类的方法.

## 7.1 wait()方法

wait 做的事情:

- 使当前执行代码的线程进行等待. (把线程放到等待队列中)
- 释放当前的锁
- 满足一定条件时被唤醒, 重新尝试获取这个锁.

wait 要搭配 synchronized 来使用. 脱离 synchronized 使用 wait 会直接抛出异常.

wait 结束等待的条件:

- 其他线程调用该对象的 notify 方法.
- wait 等待时间超时 (wait 方法提供一个带有 timeout 参数的版本, 来指定等待时间).
- 其他线程调用该等待线程的 interrupted 方法, 导致 wait 抛出 `InterruptedException` 异常.

**代码示例:** 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 方法是唤醒等待的线程.

- 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, 则有线程调度器随机挑选出一个呈 wait 状态的线程. (并没有 "先来后到")
- 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。

**代码示例:** 使用notify()方法唤醒线程

- 创建 WaitTask 类, 对应一个线程, run 内部循环调用 wait.
- 创建 NotifyTask 类, 对应另一个线程, 在 run 内部调用一次 notify
- 注意, WaitTask 和 NotifyTask 内部持有同一个 Object locker. WaitTask 和 NotifyTask 要想配合就需要搭配同一个 Object.

```
static class WaitTask implements Runnable {
    private Object locker;

    public WaitTask(Object locker) {
        this.locker = locker;
    }
}
```

```

    }

    @Override
    public void run() {
        synchronized (locker) {
            while (true) {
                try {
                    System.out.println("wait 开始");
                    locker.wait();
                    System.out.println("wait 结束");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

static class NotifyTask implements Runnable {
    private Object locker;

    public NotifyTask(Object locker) {
        this.locker = locker;
    }

    @Override
    public void run() {
        synchronized (locker) {
            System.out.println("notify 开始");
            locker.notify();
            System.out.println("notify 结束");
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    Object locker = new Object();
    Thread t1 = new Thread(new WaitTask(locker));
    Thread t2 = new Thread(new NotifyTask(locker));
    t1.start();
    Thread.sleep(1000);
    t2.start();
}

```

## 7.3 notifyAll()方法

notify方法只是唤醒某一个等待线程. 使用notifyAll方法可以一次唤醒所有的等待线程.

范例: 使用notifyAll()方法唤醒所有等待线程, 在上面的代码基础上做出修改.

- 创建 3 个 WaitTask 实例. 1 个 NotifyTask 实例.

```

static class WaitTask implements Runnable {
    // 代码不变

```

```

}

static class NotifyTask implements Runnable {
    // 代码不变
}

public static void main(String[] args) throws InterruptedException {
    Object locker = new Object();
    Thread t1 = new Thread(new WaitTask(locker));
    Thread t3 = new Thread(new WaitTask(locker));
    Thread t4 = new Thread(new WaitTask(locker));
    Thread t2 = new Thread(new NotifyTask(locker));
    t1.start();
    t3.start();
    t4.start();
    Thread.sleep(1000);
    t2.start();
}

```

此时可以看到, 调用 `notify` 只能唤醒一个线程.

- 修改 `NotifyTask` 中的 `run` 方法, 把 `notify` 替换成 `notifyAll`

```

public void run() {
    synchronized (locker) {
        System.out.println("notify 开始");
        locker.notifyAll();
        System.out.println("notify 结束");
    }
}

```

此时可以看到, 调用 `notifyAll` 能同时唤醒 3 个 wait 中的线程

**注意:** 虽然是同时唤醒 3 个线程, 但是这 3 个线程需要竞争锁. 所以并不是同时执行, 而仍然是有先有后的执行.

理解 `notify` 和 `notifyAll`

`notify` 只唤醒等待队列中的一个线程. 其他线程还是乖乖等着



面试房间

兄弟们, 俺先进去  
给你们探探路哈



应聘者在排队

加油嗷老哥~



`notifyAll` 一下全都唤醒, 需要这些线程重新竞争锁



面试房间



放开, 让我先进去



GNMD, 让劳资先来



你们这群辣鸡都闪开



明明是我先来的 T-T

## 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.

当然为了面试的目的, 我们还是总结下:

1. wait 需要搭配 synchronized 使用. sleep 不需要.
2. wait 是 Object 的方法 sleep 是 Thread 的静态方法.

## 9. 多线程案例

### 9.1 单例模式

单例模式是校招中最常考的设计模式之一.

#### 啥是设计模式?

设计模式好比象棋中的 "棋谱". 红方当头炮, 黑方马来跳. 针对红方的一些走法, 黑方应招的时候有一些固定的套路. 按照套路来走局势就不会吃亏.

软件开发中也有很多常见的 "问题场景". 针对这些问题场景, 大佬们总结出了一些固定的套路. 按照这个套路来实现代码, 也不会吃亏.

单例模式能保证某个类在程序中只存在唯一一份实例, 而不会创建出多个实例.

这一点在很多场景上都需要. 比如 JDBC 中的 DataSource 实例就只需要一个.

单例模式具体的实现方式, 分成 "饿汉" 和 "懒汉" 两种.

## 饿汉模式

类加载的同时, 创建实例.

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

## 懒汉模式-单线程版

类加载的时候不创建实例. 第一次使用的时候才创建实例.

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

## 懒汉模式-多线程版

上面的懒汉模式的实现是线程不安全的.

线程安全问题发生在首次创建实例时. 如果在多个线程中同时调用 `getInstance` 方法, 就可能导致创建出多个实例.

一旦实例已经创建好了, 后面再多线程环境调用 `getInstance` 就不再有线程安全问题了(不再修改 `instance` 了)

加上 `synchronized` 可以改善这里的线程安全问题.

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

## 懒汉模式-多线程版(改进)

以下代码在加锁的基础上, 做出了进一步改动:

- 使用双重 if 判定, 降低锁竞争的频率.
- 给 instance 加上了 volatile.

```
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;  
    }  
}
```

### 理解双重 if 判定 / volatile:

加锁 / 解锁是一件开销比较高的事情. 而懒汉模式的线程不安全只是发生在首次创建实例的时候. 因此后续使用的时候, 不必再进行加锁了.

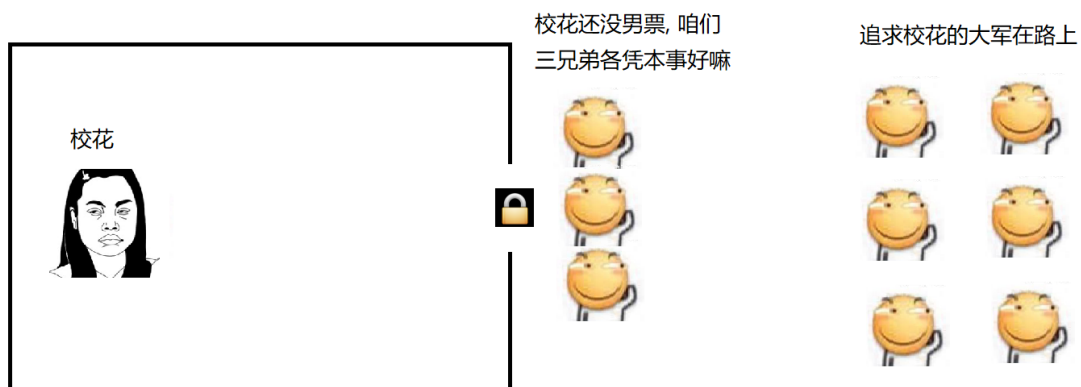
外层的 if 就是判定下看当前是否已经把 instance 实例创建出来了.

同时为了避免 "内存可见性" 导致读取的 instance 出现偏差, 于是补充上 volatile .

当多线程首次调用 getInstance, 大家可能都发现 instance 为 null, 于是又继续往下执行来竞争锁, 其中竞争成功的线程, 再完成创建实例的操作.

当这个实例创建完了之后, 其他竞争到锁的线程就被里层 if 挡住了. 也就不会继续创建其他实例.

1) 有三个线程, 开始执行 getInstance, 通过外层的 if (instance == null) 知道了实例还没有创建的消息. 于是开始竞争同一把锁.



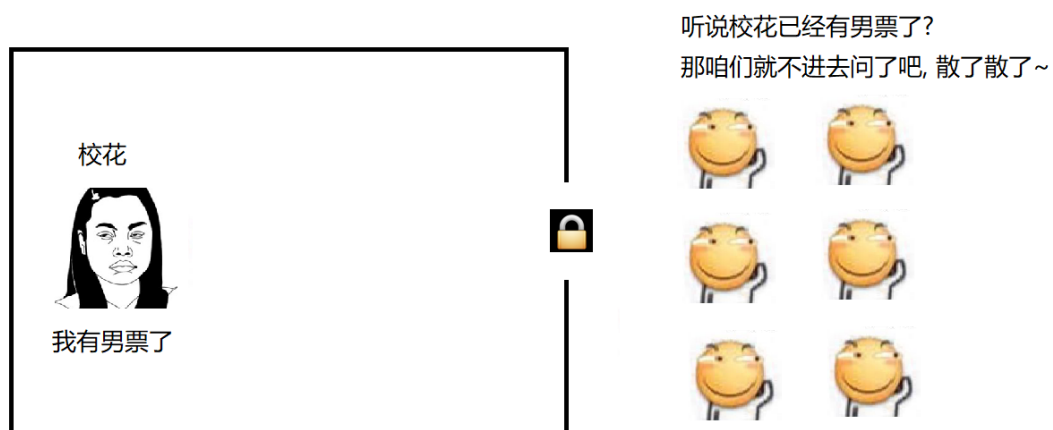
2) 其中线程1 率先获取到锁, 此时线程1 通过里层的 if (instance == null) 进一步确认实例是否已经创建. 如果没创建, 就把这个实例创建出来.



3) 当线程1 释放锁之后, 线程2 和 线程3 也拿到锁, 也通过里层的 `if (instance == null)` 来确认实例是否已经创建, 发现实例已经创建出来了, 就不再创建了.



4) 后续的线程, 不必加锁, 就直接通过外层 `if (instance == null)` 就知道实例已经创建了, 从而不再尝试获取锁了. 降低了开销.



## 9.2 阻塞式队列

### 阻塞队列是什么

阻塞队列是一种特殊的队列. 也遵守 "先进先出" 的原则.

阻塞队列能是一种线程安全的数据结构, 并且具有以下特性:

- 当队列满的时候, 继续入队列就会阻塞, 直到有其他线程从队列中取走元素.
- 当队列空的时候, 继续出队列也会阻塞, 直到有其他线程往队列中插入元素.

阻塞队列的一个典型应用场景就是 "生产者消费者模型". 这是一种非常典型的开发模型.



## 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。

1) 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。

比如在 "秒杀" 场景下, 服务器同一时刻可能会收到大量的支付请求. 如果直接处理这些支付请求, 服务器可能扛不住(每个支付请求的处理都需要比较复杂的流程). 这个时候就可以把这些请求都放到一个阻塞队列中, 然后再由消费者线程慢慢的来处理每个支付请求.

这样做可以有效进行 "削峰", 防止服务器被突然到来的一波请求直接冲垮.

2) 阻塞队列也能使生产者和消费者之间 **解耦**.

比如过年一家人一起包饺子. 一般都是有明确分工, 比如一个人负责擀饺子皮, 其他人负责包. 擀饺子皮的人就是 "生产者", 包饺子的人就是 "消费者".

擀饺子皮的人不关心包饺子的人是谁(能包就行, 无论是手工包, 借助工具, 还是机器包), 包饺子的人也不关心擀饺子皮的人是谁(有饺子皮就行, 无论是用擀面杖擀的, 还是拿罐头瓶擀, 还是直接从超市买的).

## 标准库中的阻塞队列

在 Java 标准库中内置了阻塞队列. 如果我们需要在一些程序中使用阻塞队列, 直接使用标准库中的即可.

- BlockingQueue 是一个接口. 真正实现的类是 LinkedBlockingQueue.
- put 方法用于阻塞式的入队列, take 用于阻塞式的出队列.
- BlockingQueue 也有 offer, poll, peek 等方法, 但是这些方法不带有阻塞特性.

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();
// 入队列
queue.put("abc");
// 出队列. 如果没有 put 直接 take, 就会阻塞.
String elem = queue.take();
```

## 生产者消费者模型

```
public static void main(String[] args) throws InterruptedException {
    BlockingQueue<Integer> blockingQueue = new LinkedBlockingQueue<Integer>();

    Thread customer = new Thread(() -> {
        while (true) {
            try {
                int value = blockingQueue.take();
                System.out.println("消费元素: " + value);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }, "消费者");
```

```

customer.start();

Thread producer = new Thread() -> {
    Random random = new Random();
    while (true) {
        try {
            int num = random.nextInt(1000);
            System.out.println("生产元素: " + num);
            blockingQueue.put(num);
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}, "生产者");
producer.start();

customer.join();
producer.join();
}

```

## 阻塞队列实现

- 通过 "循环队列" 的方式来实现.
- 使用 synchronized 进行加锁控制.
- put 插入元素的时候, 判定如果队列满了, 就进行 wait. (注意, 要在循环中进行 wait. 被唤醒时不一定队列就不满了, 因为同时可能是唤醒了多个线程).
- take 取出元素的时候, 判定如果队列为空, 就进行 wait. (也是循环 wait)

```

public class BlockingQueue {
    private int[] items = new int[1000];
    private volatile int size = 0;
    private int head = 0;
    private int tail = 0;

    public void put(int value) throws InterruptedException {
        synchronized (this) {
            // 此处最好使用 while.
            // 否则 notifyAll 的时候, 该线程从 wait 中被唤醒,
            // 但是紧接着并未抢占到锁. 当锁被抢占的时候, 可能又已经队列满了
            // 就只能继续等待
            while (size == items.length) {
                wait();
            }
            items[tail] = value;
            tail = (tail + 1) % items.length;
            size++;
            notifyAll();
        }
    }

    public int take() throws InterruptedException {
        int ret = 0;
        synchronized (this) {
            while (size == 0) {

```

```

        wait();
    }
    ret = items[head];
    head = (head + 1) % items.length;
    size--;
    notifyAll();
}
return ret;
}

public synchronized int size() {
    return size;
}

// 测试代码
public static void main(String[] args) throws InterruptedException {
    BlockingQueue blockingQueue = new BlockingQueue();

    Thread customer = new Thread(() -> {
        while (true) {
            try {
                int value = blockingQueue.take();
                System.out.println(value);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }, "消费者");
    customer.start();

    Thread producer = new Thread(() -> {
        Random random = new Random();
        while (true) {
            try {
                blockingQueue.put(random.nextInt(10000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }, "生产者");
    producer.start();

    customer.join();
    producer.join();
}
}

```

## 9.3 定时器

### 定时器是什么

定时器也是软件开发中的一个重要组件. 类似于一个 "闹钟". 达到一个设定的时间之后, 就执行某个指定好的代码.



定时器是一种实际开发中非常常用的组件.

比如网络通信中, 如果对方 500ms 内没有返回数据, 则断开连接尝试重连.

比如一个 Map, 希望里面的某个 key 在 3s 之后过期(自动删除).

类似于这样的场景就需要用到定时器.

## 标准库中的定时器

- 标准库中提供了一个 Timer 类. Timer 类的核心方法为 `schedule`.
- `schedule` 包含两个参数. 第一个参数指定即将要执行的任务代码, 第二个参数指定多长时间之后执行 (单位为毫秒).

```
Timer timer = new Timer();
timer.schedule(new TimerTask() {
    @Override
    public void run() {
        System.out.println("hello");
    }
}, 3000);
```

## 实现定时器

定时器的构成:

- 一个带优先级的阻塞队列

为啥要带优先级呢?

因为阻塞队列中的任务都有各自的执行时刻 (delay). 最先执行的任务一定是 delay 最小的. 使用带优先级的队列就可以高效的把这个 delay 最小的任务找出来.

- 队列中的每个元素是一个 Task 对象.
- Task 中带有时间属性, 队首元素就是即将
- 同时有一个 worker 线程一直扫描队首元素, 看队首元素是否需要执行

1) Timer 类提供的核心接口为 `schedule`, 用于注册一个任务, 并指定这个任务多长时间后执行.

```
public class Timer {
    public void schedule(Runnable command, long after) {
        // TODO
    }
}
```

2) Task 类用于描述一个任务(作为 Timer 的内部类). 里面包含一个 Runnable 对象和一个 time(毫秒时间戳)

这个对象需要放到 优先队列 中. 因此需要实现 Comparable 接口.

```
static class Task implements Comparable<Task> {
    private Runnable command;
    private long time;

    public Task(Runnable command, long time) {
        this.command = command;
        // time 中存的是绝对时间, 超过这个时间的任务就应该被执行
        this.time = System.currentTimeMillis() + time;
    }

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

    @Override
    public int compareTo(Task o) {
        // 谁的时间小谁排前面
        return (int)(time - o.time);
    }
}
```

3) Timer 实例中, 通过 PriorityBlockingQueue 来组织若干个 Task 对象.

通过 schedule 来往队列中插入一个个 Task 对象.

```
class Timer {
    // 核心结构
    private PriorityBlockingQueue<Task> queue = new PriorityBlockingQueue();

    public void schedule(Runnable command, long after) {
        Task task = new Task(command, after);
        queue.offer(task);
    }
}
```

4) Timer 类中存在一个 worker 线程, 一直不停的扫描队首元素, 看看是否能执行这个任务.

所谓 "能执行" 指的是该任务设定的时间已经到达了.

```

class Timer {
    // ... 前面的代码不变

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

    class Worker extends Thread{
        @Override
        public void run() {
            while (true) {
                try {
                    Task task = queue.take();
                    long curTime = System.currentTimeMillis();
                    if (task.time > curTime) {
                        // 时间还没到，就把任务再塞回去
                        queue.put(task);
                    } else {
                        // 时间到了，可以执行任务
                        task.run();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    break;
                }
            }
        }
    }
}

```

但是当前这个代码中存在一个严重的问题, 就是 `while (true)` 转的太快了, 造成了无意义的 CPU 浪费.

比如第一个任务设定的是 1 min 之后执行某个逻辑. 但是这里的 `while (true)` 会导致每秒钟访问队首元素几万次. 而当前距离任务执行的时间还有很久呢.

5) 引入一个 `mailBox` 对象, 借助该对象的 `wait / notify` 来解决 `while (true)` 的忙等问题.

```

class Timer {
    // 存在的意义是避免 worker 线程出现忙等的情况
    private Object mailBox = new Object();
}

```

修改 Worker 的 `run` 方法, 引入 `wait`, 等待一定的时间.

```

public void run() {
    while (true) {
        try {
            Task task = queue.take();
            long curTime = System.currentTimeMillis();
            if (task.time > curTime) {
                // 时间还没到，就把任务再塞回去
                queue.put(task);
            }
        }
    }
}

```

```

        // [引入 wait] 等待时间按照队首元素的时间来设定。
        synchronized (mailBox) {
            // 指定等待时间 wait
            mailBox.wait(task.time - curTime);
        }

        } else {
            // 时间到了，可以执行任务
            task.run();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
        break;
    }
}
}
}

```

修改 Timer 的 schedule 方法, 每次有新任务到来的时候唤醒一下 worker 线程. (因为新插入的任务可能是需要马上执行的).

```

public void schedule(Runnable command, long after) {
    Task task = new Task(command, after);
    queue.offer(task);

    // [引入 notify] 每次有新的任务来了，都唤醒一下 worker 线程，检测下当前是否有
    synchronized (mailBox) {
        mailBox.notify();
    }
}
}

```

## 完整代码

```

/**
 * 定时器的构成：
 * 一个带优先级的阻塞队列
 * 队列中的每个元素是一个 Task 对象。
 * Task 中带有时间属性，队首元素就是即将
 * 同时有一个 worker 线程一直扫描队首元素，看队首元素是否需要执行
 */
public class Timer {
    static class Task implements Comparable<Task> {
        private Runnable command;
        private long time;

        public Task(Runnable command, long time) {
            this.command = command;
            // time 中存的是绝对时间，超过这个时间的任务就应该被执行
            this.time = System.currentTimeMillis() + time;
        }

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

```

```

@Override
public int compareTo(Task o) {
    // 谁的时间小谁排前面
    return (int)(time - o.time);
}
}

// 核心结构
private PriorityBlockingQueue<Task> queue = new PriorityBlockingQueue();
// 存在的意义是避免 worker 线程出现忙等的情况
private Object mailBox = new Object();

class Worker extends Thread{
    @Override
    public void run() {
        while (true) {
            try {
                Task task = queue.take();
                long curTime = System.currentTimeMillis();
                if (task.time > curTime) {
                    // 时间还没到，就把任务再塞回去
                    queue.put(task);
                    synchronized (mailBox) {
                        // 指定等待时间 wait
                        mailBox.wait(task.time - curTime);
                    }
                } else {
                    // 时间到了，可以执行任务
                    task.run();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
                break;
            }
        }
    }
}

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

// schedule 原意为 "安排"
public void schedule(Runnable command, long after) {
    Task task = new Task(command, after);
    queue.offer(task);
    synchronized (mailBox) {
        mailBox.notify();
    }
}

public static void main(String[] args) {
    Timer timer = new Timer();
    Runnable command = new Runnable() {
        @Override
        public void run() {

```



```

        System.out.println("我来了");
        timer.schedule(this, 3000);
    }
};

timer.schedule(command, 3000);
}
}

```

## 9.4 线程池

### 线程池是什么

虽然创建线程 / 销毁线程 的开销

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。**

### 标准库中的线程池

- 使用 `Executors.newFixedThreadPool(10)` 能创建出固定包含 10 个线程的线程池。
- 返回值类型为 `ExecutorService`
- 通过 `ExecutorService.submit` 可以注册一个任务到线程池中。

```

ExecutorService pool = Executors.newFixedThreadPool(10);
pool.submit(new Runnable() {
    @Override
    public void run() {
        System.out.println("hello");
    }
});

```

Executors 创建线程池的几种方式

- `newFixedThreadPool`: 创建固定线程数的线程池
- `newCachedThreadPool`: 创建线程数目动态增长的线程池。
- `newSingleThreadExecutor`: 创建只包含单个线程的线程池。
- `newScheduledThreadPool`: 设定 延迟时间后执行命令，或者定期执行命令。是进阶版的 Timer。

Executors 本质上是 `ThreadPoolExecutor` 类的封装。

ThreadPoolExecutor 提供了更多的可选参数, 可以进一步细化线程池行为的设定. (后面再介绍)

## 实现线程池

- 核心操作为 submit, 将任务加入线程池中
- 使用 Worker 类描述一个工作线程. 使用 Runnable 描述一个任务.
- 使用一个 BlockingQueue 组织所有的任务
- 每个 worker 线程要做的事情: 不停的从 BlockingQueue 中取任务并执行.
- 指定一下线程池中的最大线程数 maxWorkerCount; 当当前线程数超过这个最大值时, 就不再新增线程了.

```
class Worker extends Thread {
    private LinkedBlockingQueue<Runnable> queue = null;

    public Worker(LinkedBlockingQueue<Runnable> queue) {
        super("worker");
        this.queue = queue;
    }

    @Override
    public void run() {
        // try 必须放在 while 外头, 或者 while 里头应该影响不大
        try {
            while (!Thread.interrupted()) {
                Runnable runnable = queue.take();
                runnable.run();
            }
        } catch (InterruptedException e) {}
    }
}
```

```
public class MyThreadPool {
    private int maxWorkerCount = 10;
    private LinkedBlockingQueue<Runnable> queue = new LinkedBlockingQueue();

    public void submit(Runnable command) {
        if (workerList.size() < maxWorkerCount) {
            // 当前 worker 数不足, 就继续创建 worker
            Worker worker = new Worker(queue);
            worker.start();
        }
        // 将任务添加到任务队列中
        queue.put(command);
    }

    public static void main(String[] args) throws InterruptedException {
        MyThreadPool myThreadPool = new MyThreadPool();
        myThreadPool.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println("吃饭");
            }
        });
    }
}
```

```
        Thread.sleep(1000);  
    }  
}
```

## 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。