线程

1、程序、进程与线程

1）程序（program）：为完成特定任务，用某种语言编写的一组指令的集合。即指**一段静态的代码，静态对象**。

2）进程（process）：**程序的一次执行过程**，或是正在内存中运行的应用程序。如：运行中的QQ，运行中的网易音乐播放器。

1> 每个进程都有**一个独立的内存空间**，系统运行一个程序即是一个**进程**从**创建、运行**到**消亡**的过程。（**生命周期**）

2> **程序**是**静态**的，**进程**是**动态**的

3> **进程**作为**操作系统调度**和**分配资源**的**最小单位**（亦是**系统运行程序的基本单位**），系统在运行时为**每个进程**分配**不同的内存区域**。

4> 现代的操作系统，大都是支持多进程的，支持同时运行多个程序。比如：现在我们上课一边使用编辑器，一边使用录屏软件，同时还开着画图板，dos 窗口等软件。

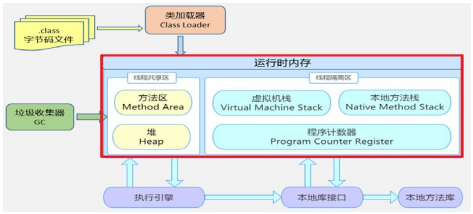
3）线程（thread）：进程可进一步细化为线程，**是程序内部的一条执行路径。一个进程中至少有一个线程**。

1> 一个进程**同一时间若并行执行多个线程**，就是支持多线程的。

2> **线程**作为 **CPU 调度和执行**的**最小单位**。

3> **一个进程中**的**多个线程共享相同的内存单元**，它们从**同一个堆中分配对象**，可以**访问相同的变量和对象**。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。

4> 下图中，红框的蓝色区域为线程独享（虚拟机栈、本地方法栈、程序计数器），黄色区域为线程共享（方法区、堆）。



【注意】**不同的进程**之间是**不共享内存**的。进程之间的数据交换和通信的成本很高。

2、线程调度

1）分时调度

所有线程**轮流使用 CPU** 的使用权，并且**平均分配**每个线程占用 CPU 的**时间**。

2）抢占式调度（Java 使用）

让**优先级高**的线程以**较大的概率优先使用 CPU**。如果线程的**优先级相同**，那么**会随机选择一个(线程随机性)，Java 使用抢占式调度。**

3、多线程程序的优点

背景：以单核 CPU 为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短（因为线程切换过程需要时间），为何仍需多线程呢？

多线程程序的优点：

1）**提高应用程序的响应**，对图形化界面更有意义，可增强用户体验；

2）**提高**计算机系统 **CPU 的利用率**；

3）**改善程序结构**，将既长又复杂的进程分为**多个线程，独立运行，利于理解和修改**。

4、单核 CPU 和多核 CPU

单核 CPU，在一个时间单元内，只能执行一个线程的任务。

问题：多核的效率是单核的倍数吗？譬如 4 核 A53 的 cpu，性能是单核 A53 的 4 倍吗？理论上是，但是实际不可能，至少有两方面的损耗：

1）**多个核心的其他共用资源限制**。譬如，4 核 CPU 对应的内存、cache、寄存器并没有同步扩充 4 倍。

2）**多核 CPU 之间的协调管理损耗**。譬如，多个核心同时运行两个相关的任务，需要考虑任务同步，这也需要消耗额外性能。

5、并行与并发

1）并行（parallel）：指两个或多个事件在**同一时刻**发生（同时发生）。指**在同一时刻，有多条指令在多个 CPU 上同时执行**。

2）并发（concurrency）：指两个或多个事件在**同一个时间段内**发生。即**在一段时间内，有多条指令在单个 CPU 上快速轮换、分时交替执行**，在**宏观上**具有多个进程**同时执行**的效果。

6、创建和启动线程

Java 语言的 JVM 允许程序运行多个线程，使用 **java.lang.Thread 类**代表线程，所有的线程对象都必须是 Thread 类或其子类的实例。

Thread 类的特性：

1）每个线程都是通过某个特定 Thread 对象的 **run()方法**来完成操作的，因此把 **run()方法体**称为**线程执行体**；

2）通过该 Thread 对象的 **start()方法**来**启动**这个线程，而**非直接调用 run()**；

3）要想**实现多线程**，必须在**主线程中创建新的线程对象**。

7、方式 1：继承 Thread 类

Java 通过继承 Thread 类来创建并启动多线程的**步骤如下**：

1）**定义 Thread 类的子类**，并**重写**该类的 **run()方法**，该 run()方法的**方法体**就代表了**线程需要完成的任务**；

2）**创建 Thread 子类实例**，即创建线程对象；

3）**调用**线程对象的 **start()方法**来**启动**该线程。

【注意】

1>如果自己**手动调用 run()方法**，那么就只是**普通方法，没有启动多线程模式**；

2>**run()方法**由 **JVM 调用，什么时候**调用，**执行的过程控制**都**由操作系统的 CPU 调度决定**；

3>想要**启动多线程，必须调用 start 方法**；

4>**一个线程对象只能调用一次 start()方法**启动，如果**重复调用**了，则将**抛出**以下的**异常**“IllegalThreadStateException”。

8、方式 2：实现 Runnable 接口

Java 有单继承的限制，当我们无法继承 Thread 类时，那么该如何做呢？在核心类库中提供了 Runnable 接口，我们可以**实现 Runnable 接口，重写 run()方法**，然后**再通过 Thread 类的对象代理启动和执行线程体 run()方法**。

步骤如下：

1）**定义 Runnable 接口的实现类**，并**重写**该接口的 **run()方法**，该 run()方法的方法体同样是该线程的线程执行体。

2）**创建 Runnable 实现类的实例**，并**以此实例作为 Thread 的 target 参数**来**创建 Thread 对象**，该 Thread 对象才是真正的线程对象。

3）**调用**线程对象的 **start()方法，启动线程**，从而调用 Runnable 接口实现类的 run 方法。

【注意】Runnable 对象仅仅作为 Thread 对象的 **target 参数**，Runnable 实现类里包含的 **run()方法**仅作为**线程执行体**。 而**实际的线程对象依然是 Thread 实例**，只是该 **Thread 线程负责执行**其 target 的 **run()方法**。即**所有的多线程代码都是通过运行 Thread 的 start()方法来运行的**。

9、对比两种方式

1）联系： Thread 类实际上也是实现了 Runnable 接口的类。即：

public class Thread extends Object implements Runnable

2）区别：线程代码 run()方法的位置

>继承 Thread: 线程代码存放 **Thread 子类** run 方法中；

>实现 Runnable: 线程代码存在 Runnable **接口的子类**的 run 方法。

3）实现 Runnable 接口比继承 Thread 类所具有的优势：

1> **避免了单继承的局限性**；

2> **多个线程**可以**共享同一个接口实现类的对象**，非常适合**多个相同线程**来**处理同一份资源**；

3> 增加程序的健壮性，实现**解耦**操作，**代码可以被多个线程共享，代码和线程独立**。

【补充】变形写法：使用**匿名内部类对象**来实现线程的创建和启动

```
new Thread("新的线程！"){
        @Override
        public void run() {
                for (int i = 0; i < 10; i++) {
System.out.println(getName()+"：正在执行"+i);
                }
        }
}.start();
new Thread(new Runnable(){
        @Override
        public void run() {
                for (int i = 0; i < 10; i++) {
      System.out.println(Thread.currentThread().
getName()+"：" + i);
                }
        }
}).start();
```

10、方式 3：实现 Callable 接口(JDK5.0 新增)

1）与使用 Runnable 相比， Callable 功能更强大些：

1> 相比 run()方法，可以有返回值；

2> 方法可以抛出异常；

3> 支持泛型的返回值（需要借助 FutureTask 类，获取返回结果）。

2）Future 接口（了解）

1> **可以对具体** Runnable、Callable **任务**的**执行结果**进行**取消、查询是否完成、获取结果等；**

2> FutureTask 是 Futrue 接口的**唯一的实现类**；

3> FutureTask **同时实现了 Runnable, Future 接口**，即它既可以作为 Runnable **被线程执行**，又可以作为 Future **得到 Callable 的返回值**；

3）缺点：在**获取分线程执行结果**的时候，**当前线程（或是主线程）受阻塞**，效率较低。

```
//1.创建一个实现Callable 的实现类
class NumThread implements Callable {
    //2.实现 call 方法，将此线程需要执行的操作声明在 call()中
    @Override
    public Object call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= 100; i++) {
            if (i % 2 == 0) {
                System.out.println(i);
                sum += i;
            }
        }
        return sum;
    }
}

public class CallableTest {
    public static void main(String[] args) {
        //3.创建Callable 接口实现类的对象
        NumThread numThread = new NumThread();
        //4.将此Callable 接口实现类的对象作为传递到FutureTask 构造器中
            创建FutureTask 的对象
        FutureTask futureTask = new FutureTask(numThread);
        //5.将FutureTask 的对象作为参数传递到Thread 类的构造器中，
          创建Thread 对象，并调用start()
        new Thread(futureTask).start();
        //接收返回值
        try {
          //6.获取Callable 中call 方法的返回值
          //get()返回值即为FutureTask 构造器参数Callable 实现类
            重写的call()的返回值。
            Object sum = futureTask.get();
            System.out.println("总和为：" + sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
```

11、方式 4：使用线程池(JDK5.0 新增)

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。思路：提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。

1）好处：

1> 提高响应速度（减少创建新线程的时间）；
2> 降低资源消耗（重复利用线程池中线程，不需要每次都创建）；
3> 可以设置相关参数，便于线程管理
- corePoolSize：核心池的大小；
- maximumPoolSize：最大线程数；
- keepAliveTime：线程没有任务时最多保持多长时间后会终止。

2）线程池相关 API

JDK5.0 之前，我们必须手动自定义线程池。从 JDK5.0 开始，Java 内置线程池相关的 API。在 **java.util.concurrent 包**下提供了线程池相关 API：**ExecutorService** 和 **Executors**。

1> ExecutorService：真正的线程池接口。常见子类 ThreadPoolExecutor。
- void **execute**(Runnable command)：**执行任务/命令，没有**返回值，一般用来执行 **Runnable**；
- <T> Future<T> **submit**(Callable<T> task)：**执行任务，有**返回值，一般用来执行 **Callable**；
- void **shutdown**()：**关闭连接池**。

2> Executors：一个线程池的工厂类，通过此类的静态工厂方法**可以创建多种类型的线程池对象**。

Executors.newCachedThreadPool()：创建一个**可根据需要创建新线程**的线程池；

Executors.newFixedThreadPool(int nThreads); 创建一个**可重用固定线程数**的线程池；

Executors.newSingleThreadExecutor()：创建一个**只有一个线程**的线程池；

Executors.newScheduledThreadPool(int corePoolSize)：创建一个线程池，它**可安排在给定延迟后运行命令或者定期地执行**。

```
public class ThreadPoolTest {
    public static void main(String[] args) {
        //1. 提供指定线程数量的线程池
        ExecutorService service = Executors.newFixedThreadPool(10);
        ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;
        //设置线程池的属性
        //System.out.println(service.getClass());//ThreadPoolExecutor
        service1.setMaximumPoolSize(50); //设置线程池中线程数的上限
        //2.执行指定的线程的操作。需要提供实现 Runnable 接口
            或 Callable 接口实现类的对象
        service.execute(new NumberThread());//适合适用于 Runnable
        service.execute(new NumberThread1());//适合适用于 Runnable
        try {
            //适合使用于 Callable
            Future future = service.submit(new NumberThread2());
            System.out.println("总和为: " + future.get());
        } catch (Exception e) {
            e.printStackTrace();
        }
        //3.关闭连接池
        service.shutdown();
    }
}
```

12、Thread 类的常用结构

1）构造器

1> public Thread() :分配一个新的线程对象；

2> public Thread(String name) :分配一个**指定名字**的新的线程对象；

3> public Thread(Runnable target) :**指定创建线程的目标对象**，它实现了 Runnable 接口中的 run 方法；

4> public Thread(Runnable target,String name) :分配一个带有**指定目标新的线程对象**并**指定名字**。

2）常用方法系列 1

1> public void **run()**：此线程要执行的任务在此处定义代码。

2> public void **start()**：启动此线程开始执行；Java 虚拟机调用此线程的 run 方法。

3>public String **getName()**：**获取**当前线程**名称**。

4> public void **setName**(String name)：**设置**该线程**名称**。

5> public static Thread **currentThread()**：返回对当前正在执行的**线程对象的引用**。在 Thread 子类中就是 this，通常用于主线程和 Runnable 实现类

6> public static void **sleep(long millis)**：使当前正在执行的线程**以指定的毫秒数暂停**（暂时停止执行）。

7> public static void **yield()**：yield 只是**让当前线程暂停一下，让系统的线程调度器重新调度一次，希望优先级与当前线程相同或更高的其他线程能够获得执行机会**，但是这个**不能保证**，完全有可能的情况是，当某个线程调用了 yield 方法暂停之后，线程调度器又将其调度出来重新执行。

3）常用方法系列 2

1> public final boolean **isAlive()**：测试线程是**否处于活动状态**。如果线程已经启动且尚未终止，则为活动状态。

2> void **join()**：等待该线程终止。

void **join(long millis)**：等待该线程终止的时间最长为 millis 毫秒。如果 millis 时间到，将不再等待。

void **join(long millis, int nanos)**：等待该线程终止的时间最长为 millis 毫秒+nanos 纳秒。

3> public final void **stop()**：**已过时**，不建议使用。强行结束一个线程的执行，直接进入死亡状态。run()即刻停止，可能会导致一些清理性的工作得不到完成，如文件，数据库等的关闭。同时，会立即释放该线程所持有的所有的锁，导致数据得不到同步的处理，出现数据不一致的问题。

4> void **suspend()** / void **resume()**：**已过时**，不建议使用。这两个操作就好比播放器的暂停和恢复。二者必须成对出现，否则非常容易发生死锁。suspend()调用会导致线程暂停，但不会释放任何锁资源，导致其它线程都无法访问被它占用的锁，直到调用 resume()。

4）常用方法系列 3

每个线程都有一定的优先级，**同优先级**线程组成先进先出队列（**先到先服务**），使用**分时调度策略**。**优先级高**的线程采用**抢占式策略**，获得较多的执行机会。**每个线程默认的**优先级都与创建**它的父线程具有相同的优先级**。

1> Thread 类的三个优先级常量：
- MAX_PRIORITY(10)：最高优先级
- MIN _PRIORITY(1)：最低优先级
- NORM_PRIORITY(5)：普通优先级，默认情况下 main 线程具有普通优先级。

2>public final int getPriority():返回线程优先级

3> public final void setPriority(int newPriority)：改变线程的优先级，范围在[1,10]之间。

5）守护线程

1）在**后台运行**的，它的任务是**为其他线程提供服务**的。**JVM 的垃圾回收线程**就是典型的守护线程。

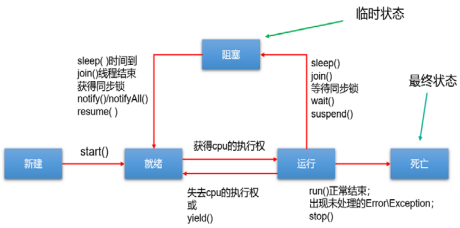
2）守护线程有个**特点**，就是如果**所有非守护线程都死亡**，那么守护线程**自动死亡**。

3）调用 **setDaemon(true)**方法可将**指定线程设置为守护线程**。必须在线程**启动之前**设置，否则会报 IllegalThreadStateException 异常。

4）调用 **isDaemon()判断**线程是否是守护线程。

13、多线程的生命周期(JDK1.5 之前)

线程的生命周期有**五种状态：新建**（New）、**就绪**（Runnable）、**运行**（Running）、**阻塞**（Blocked）、**死亡**（Dead）。CPU 需要在多条线程之间切换，于是**线程状态**会**多次**在**运行**、**阻塞**、**就绪**之间**切换**。



1）新建：当一个 Thread 类或其子类的对象**被声明并创建时**，新生的线程对象**处于新建状态**。此时它和其他 Java 对象一样，**仅仅**由 JVM 为其**分配了内存**，并**初始化了实例变量的值**。此时的线程对象**并没有任何线程的动态特征**，程序**也不会执行它的线程体 run()**。

2）就绪：但是当线程对象**调用了 start()方法之后**，线程就从**新建**状态转为**就绪状态**。JVM 会为其创建方法**调用栈和程序计数器**，当然，处于这个状态中的线程**并没有开始运行**，只是表示**已具备了运行的条件**，随时可以被调度。至于**什么时候**被调度，**取决于** JVM 里**线程调度器**的调度。

【注意】程序**只能对新建状态**的线程**调用 start()**，并且**只能调用一次**，如果对**非新建状态**的线程，如**已启动**的线程或**已死亡**的线程调用 start()都报错 IllegalThreadStateException 异常。

3）运行：如果处于就绪状态的线程**获得了 CPU 资源**时，**开始执行 run()方法**的线程体代码，则该线程**处于运行状态**。如果计算机**只有一个 CPU 核心**，在**任何时刻只有一个**线程处于**运行状态**，如果计算机有**多个核心**，将会有**多个线程并行**(Parallel)执行。

对于抢占式策略的系统而言，系统会给每个可执行的线程**一个小时间段**来处理任务，当该时间**用完**，系统会**剥夺**该线程**所占用的资源**，让其**回到就绪状态**等待下一次被调度。此时**其他线程将获得执行机会**，而在选择下一个线程时，系统会适当**考虑线程的优先级**。

4）阻塞：当在运行过程中的线程遇到如下情况时，会**让出 CPU** 并**临时中止自己**的执行，**进入阻塞状态：**

1> 线程**调用了 sleep()**方法，**主动放弃**所占用的 CPU 资源；

2> 线程试图获取一个同步监视器，但该同步监视器正被其他线程持有；（**等待同步锁**）

3> 线程执行过程中，同步监视器**调用了 wait()**，让它**等待**某个通知（**notify**）；

4> 线程执行过程中，同步监视器**调用了 wait(time)**；

5> 线程执行过程中，遇到了其他线程对象的**加塞（join）**；

6> 线程被**调用 suspend** 方法**被挂起**（已过时，因为容易发生死锁）；

解除阻塞：针对如上情况，当发生如下情况时会解除阻塞，让该线程**重新进入就绪**状态，**等待**线程调度器再次**调度**它：

1> 线程的 **sleep()时间到**；

2> 线程成功**获得了同步监视器**；

3> 线程**等到了通知(notify)**；

4> 线程 **wait 的时间到了**

5> **加塞(join)的线程结束了**；

6> **被挂起的**线程又**被调用了 resume 恢复方法**（已过时，因为容易发生死锁）；

5）死亡：线程会以以下三种方式之一结束，结束后的线程就处于死亡状态：

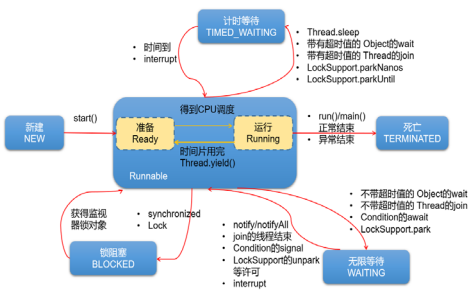
1> **run()方法执行完成**，线程正常结束；

2> 线程执行过程中**抛出**了一个**未捕获**的**异常**（Exception）**或错误**（Error）；

3> 直接**调用该线程的 stop()**来结束该线程（已过时）。

14、多线程的生命周期(JDK1.5 之后)

6 种状态：**新建**（NEW）、**可运行**（RUNNABLE）、**被终止**（Teminated）、**锁阻塞**（BLOCKED）、**计时等待**（TIMED_WAITING）、**无限等待**（WAITING）。



1）NEW（新建）：线程**刚被创建**，但是**并未启动**。还没调用 start 方法。

2）RUNNABLE（可运行）：这里**没有区分就**

绪和运行状态。(因为对于 Java 对象来说，只能标记为可运行，至于什么时候运行，不是 JVM 来控制的了，是 OS 来进行调度的，而且时间非常短暂，因此对于 Java 对象的状态来说，无法区分。)

3）Teminated（被终止）：表明此线程已经结束生命周期，终止运行。

重点说明，根据 Thread.State 的定义，阻塞状态分为三种：BLOCKED、WAITING、TIMED_WAITING。

4）BLOCKED（锁阻塞）：一个正在阻塞、等待一个监视器锁（锁对象）的线程处于这一状态。只有获得锁对象的线程才能有执行机会。比如，线程 A 与线程 B 代码中使用同一锁，如果线程 A 获取到锁，线程 A 进入到 Runnable 状态，那么线程 B 就进入到 Blocked 锁阻塞状态。

5）TIMED_WAITING（计时等待）：一个正在限时等待另一个线程执行一个（唤醒）动作的线程处于这一状态。

当前线程执行过程中遇到 Thread 类的 sleep 或 join，Object 类的 wait，LockSupport 类的 park 方法，并且在调用这些方法时设置了时间，则当前线程会进入 TIMED_WAITING，直到时间到，或被中断。

6）WAITING（无限等待）：一个正在无限期等待另一个线程执行一个特别的（唤醒）动作的线程处于这一状态。

当前线程执行过程中遇到 Object 类的 wait，Thread 类的 join，LockSupport 类的 park 方法，并且在调用这些方法时没有指定时间，那么当前线程会进入 WAITING 状态，直到被唤醒。

1> 通过 Object 类的 wait 进入 WAITING 状态的要有 Object 的 notify/notifyAll 唤醒；

2> 通过 Condition 的 await 进入 WAITING 状态的要有 Condition 的 signal 方法唤醒；

3> 通过 LockSupport 类的 park 方法进入 WAITING 状态的要有 LockSupport 类的 unpark 方法唤醒；

4> 通过 Thread 类的 join 进入 WAITING 状态，只有调用 join 方法的线程对象结束才能让当前线程恢复。

【说明】

1> 计时等待(Timed Waiting)与无限等待(Waiting)状态联系紧密，Waiting（无限等待）状态中是空参的，而 timed waiting（计时等待）中 wait 方法是带参的。如果没有得到（唤醒）通知，那么线程就处于 Timed Waiting 状态，直到倒计时完毕自动醒来；如果在倒计时期间得到（唤醒）通知，那么线程从 Timed Waiting 状态立刻唤醒。

2>当从 WAITING 或 TIMED_WAITING 恢复到 Runnable 状态时，如果发现当前线程没有得到监视器锁，那么会立刻转入 BLOCKED 状态。

15、线程安全问题原因：一个线程处理共享资源过程中，在尚未结束的情况下，其他线程也参与进来对同一资源进行处理。

解决思路：必须保证一个线程在处理该资源过程中，其他线程必须等待，直到该线程对该资源处理结束以后，其他线程才能去抢夺 CPU 资源，完成对应的操作，保证了数据的同步性，解决了线程不安全的现象。

16、同步机制解决线程安全问题的原理：

相当于给某段代码加“锁”，任何线程想要执行这段代码，都要先获得“锁”，我们称它为同步锁。哪个线程获得了“同步锁”对象之后，“同步锁”对象就会记录这个线程的 ID，这样其他线程就只能等待了，除非这个线程“释放”了锁对象，其他线程才能重新获得/占用“同步锁”对象。

因为 Java 对象在堆中的数据分为对象头、实例变量、空白的填充。而对象头中包含：

1）Mark Word：记录了和当前对象有关的 GC、锁标记等信息。

2）指向类的指针：每一个对象需要记录它是由哪个类创建出来的。

3）数组长度（只有数组对象才有）

17、同步代码块

synchronized 关键字可以用于某个区块前面，表示只对这个区块的资源实行互斥访问。

```
synchronized(同步锁){
        需要同步操作的代码
}
```

18、同步方法

synchronized 关键字直接修饰方法，表示同一时刻只有一个线程能进入这个方法，其他线程在外面等着。

```
public synchronized void method(){
        可能会产生线程安全问题的代码
}
```

【说明】synchronized 好处：解决线程安全问题；弊端：在操作共享数据时，多线程其实是串行的，性能低。

19、同步锁对象可以是任意类型，但是必须保证竞争“同一个共享资源”的多个线程必须使用同一个“同步锁对象”。

对于同步代码块来说，同步锁对象是由程序员手动指定的（很多时候也是指定为 this 或类名.class），但是对于同步方法来说，同步锁对象只能是默认的：

静态方法→类名.class（当前类的 Class 对象）；

非静态方法→this；

【注意】锁范围太小：不能解决安全问题；

锁范围太大：因为一旦某个线程抢到锁，其他线程就只能等待，所以范围太大，效率会降低，不能合理利用 CPU 资源。

20、单例设计模式的线程安全问题

1）饿汉式没有线程安全问题

饿汉式在类初始化时就直接创建单例对象，而类初始化过程是没有线程安全问题的。

```
public class HungrySingle {
    //对象是否声明为 final 都可以
    private static HungrySingle INSTANCE = new HungrySingle();
    private HungrySingle(){}

    public static HungrySingle getInstance(){
        return INSTANCE;
    }
}
```

2）懒汉式线程安全问题

懒汉式延迟创建对象，第一次调用 getInstance 方法再创建对象。

形式一：synchronized 关键字

```
public class LazyOne {
    private static LazyOne instance;
    private LazyOne(){}
    //方式 1:
    public static synchronized LazyOne getInstance1(){
        if(instance == null){
            instance = new LazyOne();
        }
        return instance;
    }
}
```

```
public class LazyOne {
    private static LazyOne instance;
    private LazyOne(){}
    //方式 2:
    public static LazyOne getInstance2(){
        synchronized(LazyOne.class) {
            if (instance == null) {
                instance = new LazyOne();
            }
            return instance;
        }
    }
}
```

```
public class LazyOne {
    private static volatile LazyOne instance;
    private LazyOne(){}
    //方式 3:
    public static LazyOne getInstance3(){
        if(instance == null){
            synchronized (LazyOne.class) {
                try {
                    Thread.sleep(10);//加这个代码，暴露问题
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if(instance == null){
                    instance = new LazyOne();
                }
            }
        }
        return instance;
    }
```

【注意】上述方式 3 中，有指令重排问题；mem = allocate(); 为单例对象分配内存空间；instance = mem; instance 引用现在非空，但还未初始化；ctorSingleton(instance); 为单例对象通过 instance 调用构造器；

从 JDK2 开始，分配空间、初始化、调用构造器会在线程的工作存储区一次性完成，然后复制到主存储区。但是需要 volatile 关键字，避免指令重排。

形式二：使用内部类

```
public class LazySingle {
    private LazySingle(){}
    public static LazySingle getInstance(){
        return Inner.INSTANCE;
    }
    private static class Inner{
        static final LazySingle INSTANCE = new LazySingle();
    }

}
```

【说明】内部类只有在外部类被调用才加载，产生 INSTANCE 实例；又不用加锁。

此模式具有之前两个模式的优点，同时屏蔽了它们的缺点，是最好的单例模式。

此时的内部类，也可以使用 enum 进行定义。

21、产生死锁的四个必要条件：

1）互斥条件：在一段时间内某资源只由一个进程占用；

2）请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放；

3）不能被剥夺条件：进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放；

4）循环等待条件：发生死锁时，必然存在一个进程资源的环形链。

22、解决死锁

死锁一旦出现，基本很难人为干预，只能尽量规避。可以考虑打破上面的诱发条件。

针对条件 1：互斥条件基本上无法被破坏。因为线程需要通过互斥解决安全问题。

针对条件 2：可以考虑一次性申请所有所需的资源，这样就不存在等待的问题。

针对条件 3：占用部分资源的线程在进一步申请其他资源时，如果申请不到，就主动释放掉已经占用的资源。

针对条件 4：可以将资源改为线性顺序。申请资源时，先申请序号较小的，这样避免循环等待问题。

23、Lock(锁)（JDK5.0 新特性）

1）与采用 synchronized 相比，Lock 可提供多种锁方案，更灵活、更强大。Lock 通过显式定义同步锁对象来实现同步。同步锁使用 Lock 对象充当。

2）java.util.concurrent.locks.Lock 接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对 Lock 对象加锁，线程开始访问共享资源之前应先获得 Lock 对象。

3）在实现线程安全的控制中，比较常用的是

ReentrantLock，可以显式加锁、释放锁。
ReentrantLock 类实现了 Lock 接口，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。
4）Lock 锁也称同步锁，加锁与释放锁方法，
- public void **lock()**：加同步锁。
- public void **unlock()**：释放同步锁。

【注意】如果同步代码有异常，要将 unlock()写入 finally 语句块。

```
class Window implements Runnable{
    int ticket = 100;
    //1. 创建Lock 的实例，必须确保多个线程共享同一个 Lock 实例
    private final ReentrantLock lock = new ReentrantLock();
    public void run(){
        while(true){
            try{
                //2. 调动 lock()，实现需共享的代码的锁定
                lock.lock();
                if(ticket > 0){
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(ticket--);
                }else{
                    break;
                }
            }finally{
                //3. 调用unLock()，释放共享代码的锁定
                lock.unlock();
            }
        }
    }


public class ThreadLock {
    public static void main(String[] args) {
        Window t = new Window();
        Thread t1 = new Thread(t);
        Thread t2 = new Thread(t);

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

24、synchronized 与 Lock 的对比
1）**Lock** 是**显式锁**（手动开启和关闭锁，别忘记关闭锁），**synchronized** 是**隐式锁**，出了作用域、遇到异常等**自动解锁**；
2）**Lock** 只有**代码块锁**，synchronized 有**代码块锁**和**方法锁**；
3）**使用 Lock 锁**，JVM 将花费**较少的时间**来**调度线程，性能更好**，并且具有更好的扩展性（提供更多的子类），更体现面向对象；
4）（了解）**Lock 锁**可以**对读不加锁，对写加锁**，synchronized 不可以；
5）（了解）Lock 锁可以有**多种获取锁**的方式，可以**从 sleep 的线程中抢到锁**，synchronized 不可以；
【说明】开发建议中处理线程安全问题优先使用顺序为：Lock →同步代码块→同步方法。
25、线程间通信
当我们需要**多个线程**来**共同完成一件任务**，并且我们希望他们**有规律的执行**，那么多线程之间需要一些**通信机制（等待唤醒机制）**，可以**协调它们的工作**，以此实现多线程**共同操作一份数据**。
26、等待唤醒机制
在一个线程满足某个条件时，就进入等待状态（wait() / wait(time)），等待其他线程执行完他们的指定代码过后再将其唤醒（notify()）；或可以指定 wait 的时间，等时间到了自动唤醒；在有多个线程进行等待时，如果需要，可以使用 notifyAll()来唤醒所有的等待线程。
wait/notify 就是线程间的一种协作机制。
1）wait：线程不再活动，不再参与调度，进入 wait set 中，因此不会浪费 CPU 资源，也不会去竞争锁了，这时的线程状态是 WAITING 或 TIMED_WAITING。它还要等着别的线程执行一个特别的动作，也即“通知（notify）”或者等待时间到，在这个对象上等待的线程从 wait set 中释放出来，重新进入到调度队列（ready queue）中；
2）notify：则选取所通知对象的 wait set 中的一个线程释放；
3）notifyAll：则释放所通知对象的 wait set 上的全部线程。
【注意】被通知的线程被唤醒后也**不一定能立即恢复**执行，因为它当初中断的地方是在同步块内，而此刻它已经不持有锁，所以它**需要再次尝试去获取锁**（很可能面临其它线程的竞争），**成功后**才能在**当初调用 wait 方法之后的地方**恢复执行。
即：如果能**获取锁**，线程就从 WAITING 状态**变成 RUNNABLE（可运行）**状态；
否则，线程就从 WAITING 状态又变成 **BLOCKED（等待锁）**状态。

例题：使用两个线程打印 1-100。线程 1，线程 2 交替打印

```
class Communication implements Runnable {
    int i = 1;
    public void run() {
        while (true) {
            synchronized (this) {
                notify();
                if (i <= 100) {
                    System.out.println(Thread.currentThread().getName() +
                                       ":" + i++);
                } else
                    break;
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

27、调用 wait 和 notify 需注意的细节
1）wait 方法与 notify 方法必须要由同一个锁对象调用。因为：对应的锁对象可以通过 notify 唤醒使用同一个锁对象调用的 wait 方法后的线程。
2）wait 方法与 notify 方法是属于 Object 类的方法的。因为：锁对象可以是任意对象，而任意对象的所属类都是继承了 Object 类的。
3）wait 方法与 notify 方法必须要在同步代码块或者是同步函数中使用。因为：必须要通过锁对象调用这两个方法。否则会报异常 java.lang.IllegalMonitorStateException。
28、区分 sleep()和 wait()
相同点：一旦执行，都会使得当前线程结束执行状态，进入阻塞状态。
不同点：
① **定义方法所属的类**：sleep():**Thread** 中定义；wait()：**Object** 中定义；
② **使用范围的不同**：sleep()可以在**任何需要使用的位置**被调用；wait():必须使用**在同步代码块**或**同步方法**中；
③ **都在同步结构中使用的时候，是否释放同步监视器的操作不同**：sleep():**不会释放**同步监视器；wait():**会释放**同步监视器；
④ **结束等待的方式不同**：sleep()：**指定时间一到**就结束阻塞；wait()：可以**指定时间**也可以**无限等待直到 notify 或 notifyAll**。
29、生产者与消费者问题
1）该问题描述了**两个（多个）共享固定大小缓冲区的线程**。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程；与此同时，消费者也在缓冲区消耗这些数据。该问题的**关键**就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。
2）生产者与消费者问题中其实隐含两个问题：
1> 线程安全问题：因为生产者与消费者共享数据缓冲区，产生安全问题。不过这个问题可以**使用同步解决**。
2> 线程的协调工作问题：要解决该问题，就必须让生产者线程在缓冲区**满时等待(wait)**，暂停进入阻塞状态，**等到下次消费者消耗了**缓冲区中的数据的时候，**通知(notify)**正在等待的线程恢复到就绪状态，重新开始**往缓冲区添加数据**。同样，也可以让**消费者线程**在缓冲区**空时进入等待(wait)**，暂停进入阻塞状态，**等到生产者往缓冲区添加**数据之后，再**通知(notify)**正在等待的线程恢复到就绪状态。通过这样的通信机制来解决此类问题。

```
public class ConsumerProducerTest {
        public static void main(String[] args) {
                Clerk clerk = new Clerk();
                Producer p1 = new Producer(clerk);
            Consumer c1 = new Consumer(clerk);
            Consumer c2 = new Consumer(clerk);
                p1.setName("生产者 1");
                c1.setName("消费者 1");
                c2.setName("消费者 2");
                p1.start();
                c1.start();
                c2.start();}}
class Producer extends Thread{//生产者
        private Clerk clerk;
    public Producer(Clerk clerk){this.clerk = clerk;}
        @Override
        public void run() {
                System.out.println("生产产品");
                while(true){
                        try {Thread.sleep(40);
                }catch (InterruptedException e){
                                e.printStackTrace();}
                        //要求 clerk 去增加产品
                        clerk.addProduct();
                }}}
class Consumer extends Thread{//消费者
        private Clerk clerk;
        public Consumer(Clerk clerk){
                this.clerk = clerk;}
        @Override
        public void run() {
                System.out.println("消费产品");
                while(true){
                        try {Thread.sleep(90);
                    }catch (InterruptedException e) {
                                e.printStackTrace();}
                        //要求 clerk 去减少产品
                        clerk.minusProduct();
                }}}
class Clerk {//资源类
        private int productNum = 0;//产品数量
    private static final int MAX_PRODUCT= 20;
    private static final int MIN_PRODUCT = 1;
        public synchronized void addProduct() {
            if(productNum < MAX_PRODUCT){
                        productNum++;
        System.out.println(" 生 产 了 第 " +
productNum + "个产品");
                        this.notifyAll();//唤醒消费者
                }else{
                        try {this.wait();
                    }catch (InterruptedException e) {
                                e.printStackTrace();
                        }}}
    public synchronized void minusProduct(){
            if(productNum >= MIN_PRODUCT){
        System.out.println(" 消 费 了 第 " +
productNum + "个产品");
                        productNum--;
                        this.notifyAll();//唤醒生产者
                }else{
                        try {this.wait();
                    }catch (InterruptedException e) {
                                e.printStackTrace();}}}}
```

30、是否释放锁的操作
1）释放锁的操作：1>当前线程的同步方法、同步代码块执行结束；2>当前线程在同步代码块、同步方法中遇到 break、return 终止了该代码块、该方法的继续执行；3>当前线程在同步代码块、同步方法中出现了未处理的 Error 或 Exception，导致当前线程异常结束；4>当前线程在同步代码块、同步方法中执行了锁对象的 wait()方法，当前线程被挂起，并释放锁。
2）不会释放锁的操作：1>线程执行同步代码块或同步方法时，程序调用 Thread.sleep()、Thread.yield()方法暂停当前线程的执行；2>线程执行同步代码块时，其他线程调用了该线程的 suspend()方法将该该线程挂起，该线程不会释放锁（同步监视器）。