

Java多线程编程

- Java多线程编程
 - 线程基础
 - 线程概念
 - 异步
 - 创建线程
 - 线程组
 - 线程状态及生命周期
 - 线程控制
 - 线程同步
 - synchronized 关键字
 - 双重检查锁定
 - 卖票的案例
 - 对 run() 添加 synchronized
 - 用同步机制来确保 quantity 变量的修改是原子操作
 - 双重检查
 - 应用 - 实现线程安全的单例模式
 - 同步锁
 - 同步锁类型
 - 死锁
 - 线程间协作
 - 等待唤醒机制
 - wait() 和 sleep() 的对比
 - 线程间通讯
 - 等待唤醒工具 - LockSupport
 - 高级同步工具
 - 显式锁 - Lock 接口
 - ReentrantLock 类
 - ReadWriteLock 接口
 - ReentrantReadWriteLock 类
 - StampedLock 类
 - 对比 Lock 与 synchronized
 - 条件变量 - Condition
 - 不可重入锁
 - 线程局部存储

- ThreadLocal
 - Supplier 函数式接口
 - 线程间共享 ThreadLocal - InheritableThreadLocal
- Executor框架 - 线程池
 - Executor框架
 - 线程池
 - 任务提交
 - 线程池的关闭
 - 示例
- 线程安全和并发工具
 - 内存可见性 - volatile 关键字
 - 内存可见性问题
 - volatile 关键字
 - 原子性
 - 原子性并发工具

线程基础

线程概念

一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的非守护线程都结束运行后才能结束。

线程是程序执行的最小单元，是操作系统能够进行运算调度的最小单位。Java 程序总是从 main 方法开始执行，main 方法本身就是一个线程。

异步

线程的异步处理是多线程编程中的一个重要概念，它允许程序中的某些操作在后台执行，而不会阻塞主线程。

在多线程环境中，线程之间通过特定的机制实现任务的并行处理，而不是顺序执行。这种机制允许一个或多个线程在等待某个任务完成时，继续执行其他任务，而不是被阻塞。

创建线程

- 继承 Thread 类

```

public class Task extends Thread{
    //run()方法执行线程需要进行的操作
    @Override
    public void run(){
        System.out.println("执行线程的任务");
    }

    public static void main(String[] args){
        //创建实例
        Task task = new Task();

        //启动线程
        task.start();
    }
}

```

缺点和局限性：

1. 缺乏灵活性：当你继承了 Thread 类后，你的类就不能再继承其他类了，这限制了类的复用性，因为Java不支持多重继承。
2. 资源限制：继承 Thread 类的线程数量是有限的。由于Java中每个线程都是一个对象，如果创建大量的线程，会消耗大量的内存资源。

- 实现 Runnable 接口

```

public class Task implements Runnable {
    //run()方法执行线程需要进行的操作
    @Override
    public void run(){
        System.out.println("执行线程的任务");
    }

    public static void main(String[] args){
        //创建实例
        Task task = new Task();

        //创建线程
        Thread thread = new Thread(task);

        //启动线程
        thread.start();
    }
}

```

- 实现 Callable 接口

```

//实现Callable接口，指定线程返回的结果类型为String
public class Result implements Callable<String> {

```

```

//call()方法执行线程的任务并返回执行结果
@Override
public String call(){
    //执行线程的任务
    System.out.println("执行线程的任务");

    //返回执行结果
    return "执行线程的结果";
}

public static void main(String[] args){
    //创建 Result 实例
    Result result = new Result();

    //创建 FutureTask 实例
    FutureTask<String> futureTask = new FutureTask<>(result);

    //创建线程
    Thread thread = new Thread(futureTask);

    //启动线程
    thread.start();

    //获取线程执行的结果
    try {
        String res = futureTask.get();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
}

```

- lambda 表达式创建

```

new Thread(() -> {
    //线程执行代码
}, name);

```

总结:

创建方式	使用场景
Thread	单继承
Runnable	无返回值任务
Callable	有返回值任务

日常开发推荐使用 `Runnable` 和 `Callable` 接口。

线程组

- 获取线程组和创建线程组

```
//返回线程所属的线程组，如果此线程死亡则返回 null
public final ThreadGroup getThreadGroup()

//创建一个线程，name指定线程组名，parent指定线程组的名称
public ThreadGroup(String name)
public ThreadGroup(ThreadGroup parent, String name)
```

- 设置线程组

`Thread` 有几种构造方法：

```
public Thread(String name)

//ThreadGroup group: 线程组，String name: 线程的名称
public Thread(ThreadGroup group, String name){
    init(group, null, name, 0);
}

//Runnable target: 线程执行的目标任务，即实现了 Runnable 接口的对象
public Thread(ThreadGroup group, Runnable target){
    init(group, target, "Thread-" + nextThreadNum(), 0);
}

public Thread(ThreadGroup group, Runnable target, String name){
    init(group, target, name, 0);
}

//long stackSize: 线程的栈大小，表示线程栈的内存大小，以字节为单位，0表示使用默认的栈大小
public Thread(ThreadGroup group, Runnable target, String name, long stackSize){
    init(group, target, name, stackSize);
}
```

线程组中常用的方法：

方法	作用
<code>getName()</code>	获取线程组名称
<code>setMaxPriority()</code>	设置线程组最大优先级
<code>getMaxPriority()</code>	获取线程组最大优先级

方法	作用
<code>activeCount()</code>	获取线程组中存活的线程的数量
<code>interrupt()</code>	中断线程组中所有的线程

线程状态及生命周期

以下是 Java 线程的主要状态及其描述：

1. 新建 (New)
线程对象已经被创建，但还没有调用 `start()` 方法。此时线程尚未开始执行。
2. 可运行 (Runnable)
线程已经调用了 `start()` 方法，并且正在 Java 虚拟机中执行。可运行状态包括了操作系统线程的就绪 (Ready) 和运行 (Running) 状态。线程调度器负责将就绪状态的线程变为运行状态。
3. 阻塞 (Blocked)
线程正在等待监视器锁（即等待 `synchronized` 同步锁）以进入同步区域或重新进入同步区域或者线程调用 `Lock.lock()` 进入阻塞状态。在等待期间，线程不会被分配 CPU 时间片。
4. 等待 (Waiting)
线程通过调用 `wait()`、`join()`、`Condition.await()` 或者 `LockSupport.park()` 方法进入等待状态。在这种状态下，线程不会被分配 CPU 时间片，直到其他线程调用 `notify()`、`notifyAll()`、`Condition.signal()`、`Condition.signalAll()` 或 `LockSupport.unpark(Thread)` 来唤醒它。
5. 计时等待 (Timed Waiting)
线程在指定的等待时间内等待某个条件发生。这通常通过调用 `sleep(long millis)`、`wait(long timeout)`、`join(long millis)`、`Condition.await()`、`LockSupport.parkNanos()`、`LockSupport.parkUntil()` 方法实现。计时等待结束后，线程会自动转换为可运行状态。
6. 终止 (Terminated)
线程的运行结束。这可能是因为线程正常执行完毕，或者因为某个未捕获的异常导致线程结束。

线程控制

- 获取当前执行任务的线程

```
public static Thread currentThread()

System.out.println(Thread.currentThread()); //输出结构: Thread[线程id, 线程名, 线程优先级]
```

- 获取和设置线程名

```
//获取线程名称
public final String getName();

//设置线程名称
public final synchronized void setName(String name);
```

- 获取和设置线程优先级

线程优先级的取值范围是 [1, 10] , 最低是 1 , 最高是 10 , 默认优先级是 5 。

```
Thread.MIN_PRIORITY;//最低优先级
Thread.MAX_PRIORITY;//最高优先级
Thread.NORM_PRIORITY;//默认优先级

//获取线程优先级
public final int getPriority()

//设置线程优先级
public final void setPriority(int priority)
```

- 线程休眠

```
/**
 * 使当前正在执行的线程进入休眠状态(暂停执行)
 * @param millsec 睡眠时间, 单位毫秒
 */
public static void sleep(long millisec)
```

- 结束线程

Java并不提供直接停止一个线程的机制, 因为强制终止一个线程可能会导致资源泄漏、数据不一致等问题。Thread 类的 stop() 方法已经被废弃, 因为它会导致线程立即停止, 可能会留下线程不安全的状态。

可以通过以下方式结束线程:

1. 中断线程

使用 interrupt() 方法可以请求线程终止。如果线程正在执行中断性的操作 (如 sleep、wait、join 等), 它会抛出 InterruptedException, 从而有机会退出执行。

2. 检查中断状态

在线程的执行过程中定期检查中断状态, 可以使用

Thread.currentThread().isInterrupted() 来检测线程是否被请求中断, 并在适当的时候退出。

3. 使用标志变量

设置一个共享的标志变量（如 `volatile boolean` 类型的变量），在其他线程中改变这个变量的值来指示目标线程应该停止执行。

4. 执行完毕后退出现

确保线程在执行完任务后能够正常退出。这通常是通过 `return` 语句实现的。

5. 关闭线程池

如果线程是由 `ExecutorService` 创建的，调用 `shutdown()` 或 `shutdownNow()` 方法可以启动关闭序列，这会等待正在执行的任务完成或尝试停止当前执行的任务。

```
//判断线程是否被中断，中断返回true，否则返回false
```

```
public boolean isInterrupted()
```

```
//判断线程是否被中断，并清除中断标记
```

```
public boolean interrupted()
```

```
//将线程标记为中断状态
```

```
public void interrupt()
```

`interrupt()` 方法不会强制线程立即停止执行，它只是提供了一种协作机制，让线程在检测到中断信号后能够优雅地结束任务。

在中断异常中结束线程示例：

```
public class ThreadInterruptionExample {
    public static void main(String[] args) {
        Thread workerThread = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                // 模拟长时间运行的任务
                System.out.println("Thread is working...");

                try {
                    // 让线程有机会响应中断
                    Thread.sleep(1000);
                } catch (InterruptedException e) { // 捕获中断异常
                    // 在此处执行中断后需要执行的操作
                    System.out.println("Thread was interrupted. Exiting loop.");
                    break;
                }
            }
        });

        workerThread.start();

        try {
            // 使main线程等待3秒，以便让worker线程有机会响应中断
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```



```

    }

    // 中断线程
    workerThread.interrupt();

    // 等待线程结束
    try {
        workerThread.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

- 暂停执行

//暂停当前正在执行的线程对象，并执行其他线程。

```

public static void yield()

```

- 等待线程死亡

//等待线程死亡

```

public final void join()

```

示例：

```

public class Task implements Runnable{
    private Thread preThread;//需要等待执行完的线程

    @Override
    public void run(){
        try {
            preThread.join();//等待线程执行完毕
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //执行线程的任务
        System.out.println("执行线程的任务");
    }
}

```

join() 方法的应用：

1. 同步

在多线程程序中，join() 方法可以用来同步线程。如果一个线程需要另一个线程的执行结果，它可以通过调用另一个线程的 join() 方法来等待那个线程完成。

2. 避免数据竞争

当多个线程需要访问共享资源时，`join()` 方法可以确保在共享资源被修改后，其他线程在继续执行之前能够获得最新的数据。

3. 资源清理

如果一个线程在结束前需要进行一些清理工作（如关闭文件、释放资源等），其他线程可以通过调用 `join()` 方法来确保这些清理工作完成。

4. 避免死锁

在复杂的多线程程序中，`join()` 方法可以用来避免死锁。通过控制线程的终止顺序，可以减少死锁发生的可能性。

5. 主线程等待子线程

在应用程序的主线程（通常是 `main` 线程）中，可以使用 `join()` 方法来等待所有子线程完成，以确保程序在所有任务完成后才退出。

6. 线程池中的优雅关闭

在使用 `ExecutorService` 等线程池时，`join()` 方法可以用来等待所有提交的任务完成后关闭线程池。

• 线程执行状态

```
//测试线程是否处于活动状态
public final boolean isAlive()
```

运行 `start()` 之前，`isAlive()` 返回 `false` 运行 `start()` 启动线程后，执行线程时返回 `true` 线程执行完毕后，`isAlive()` 返回 `false`

线程同步

synchronized 关键字

`synchronized` 的意思是同步，它可以用来给对象和方法或者代码块加锁，当它锁定一个方法或者一个代码块的时候，同一时刻最多只有一个线程执行这段代码。具体用途如下：

1. 同步方法

可以修饰实例方法或静态方法。当一个线程访问某个对象的同步实例方法时，其他线程不能访问该对象的任何其他同步实例方法。如果修饰的是静态方法，那么会锁定整个类的 `Class` 对象。

2. 同步代码块

在需要同步的代码片段上使用，可以锁定一个对象，确保同一时间只有一个线程能执行该代码块。

3. 实现锁机制

`synchronized` 可以作为一种内置的锁机制，用来控制对共享资源的访问，防止多线程环境下的数据竞争和不一致问题。

4. 内存可见性

`synchronized` 确保一个线程对共享变量的修改对其他线程是可见的。当一个线程离开同步块时，会自动刷新其工作内存到主内存，其他线程在进入同步块时会从主内存加载共享变量。

5. 有序性

`synchronized` 可以保证代码块的执行顺序，防止编译器和处理器对指令进行重排序。

6. 实现不可变对象

通过使用 `synchronized` 可以创建不可变对象，确保对象的状态在构造之后不会被修改。

7. 实现线程安全的单例

`synchronized` 可以用来实现线程安全的单例模式，确保在多线程环境下只创建一个实例。

8. 死锁处理

虽然 `synchronized` 可能导致死锁，但也可以用于死锁的检测和避免。

双重检查锁定

卖票的案例

我们从以下卖票的例子探索 **双重检查锁定** 的作用。

```
public class TicketingTask implements Runnable{
    private int quantity = 10;

    @Override
    public void run(){
        while(quantity > 0){
            System.out.println(Thread.currentThread().getName() + ":" + quantity +
                quantity--);
        }
    }

    public static void main(String args[]){
        TicketingTask ticketingTask = new TicketingTask();

        //创建线程
        Thread thread1 = new Thread(ticketingTask, "thread1");
        Thread thread2 = new Thread(ticketingTask, "thread2");
        Thread thread3 = new Thread(ticketingTask, "thread3");

        //启动线程
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

```
}  
}
```

运行以上代码的结果：

```
thread1:10号票  
thread1:9号票  
thread1:8号票  
thread1:7号票  
thread1:6号票  
thread1:5号票  
thread1:4号票  
thread1:3号票  
thread1:2号票  
thread1:1号票  
thread3:10号票  
thread2:10号票
```

显然结果出现了错误，卖出了三张 10号票。初始 `quantity = 10` 满足 `run()` 中的循环条件 `quantity > 0`，因此三个线程均能进入到循环中。而 `thread1` 首先获得了对共享资源的控制，并在其他线程有机会修改 `quantity` 之前完成了所有操作。当它执行完毕后，`quantity = 0` 使得另外两个线程输出完 10号票 后就退出了循环。

对 `run()` 添加 `synchronized`

如何进行改进呢？首先考虑对 `run()` 方法添加 `synchronized` 关键字，但这样使得只有一个线程能够运行 `run()` 方法，失去了多线程的意义了，显然不可取。

```
@Override  
public synchronized void run(){  
    while(quantity > 0){  
        System.out.println(Thread.currentThread().getName() + ":" + quantity + "号票");  
        quantity--;  
    }  
}
```

对 `while` 循环添加 `synchronized` 关键字也是一样的效果。

```
@Override  
public void run(){  
    synchronized(this) {  
        while (quantity > 0) {  
            System.out.println(Thread.currentThread().getName() + ":" + quantity + "号票");  
            quantity--;  
        }  
    }  
}
```


```
    }  
}
```

以上两种情况都是一样的运行结果：

```
thread1:10号票  
thread1:9号票  
thread1:8号票  
thread1:7号票  
thread1:6号票  
thread1:5号票  
thread1:4号票  
thread1:3号票  
thread1:2号票  
thread1:1号票
```

用同步机制来确保 `quantity` 变量的修改是原子操作

```
@Override  
public void run(){  
    while (quantity > 0) {  
        synchronized(this) {  
            System.out.println(Thread.currentThread().getName() + ":" + quantity +  
                quantity--;  
        }  
    }  
}
```



运行结果：

```
thread1:10号票  
thread1:9号票  
thread1:8号票  
thread1:7号票  
thread1:6号票  
thread1:5号票  
thread1:4号票  
thread1:3号票  
thread1:2号票  
thread1:1号票  
thread3:0号票  
thread2:-1号票
```

问题解答：

1. 为什么 `thread1` 一直控制着 `quantity` 资源？

由于 `System.out.println` 操作，输出操作可能需要一些时间，这导致线程 `thread1` 持有锁的时间比必要的长。这使得其他线程（`thread2` 和 `thread3`）很难获得锁，因此它们无法执行。如果去除输出操作，那么每个线程都有机会执行。

2. 为什么卖出两张非法的票 0, -1? 当 `quantity = 1` 时，三个线程在循环内部，三个线程同时争夺同步锁，三个线程终究都会拿到锁，因此三个线程各执行一次 `quantity--`，由此出现两张非法票 0, -1。

双重检查

针对线程进入循环后卖出非法票的问题，只需要在锁内部再进行一次判断，形成同步代码块内外双重判断：

```
@Override
public void run(){
    while (quantity > 0) {
        synchronized (this) {
            if (quantity > 0) {
                System.out.println(Thread.currentThread().getName() + ":" + quantity);
                quantity--;
            }
        }
    }
}
```

运行结果：

```
thread1:10号票
thread1:9号票
thread1:8号票
thread1:7号票
thread1:6号票
thread1:5号票
thread1:4号票
thread1:3号票
thread1:2号票
thread1:1号票
```

应用 - 实现线程安全的单例模式

```
public class Singleton {
    // volatile 确保多线程环境下的可见性
    private static volatile Singleton instance;

    // 私有构造函数，防止外部通过new创建实例
    private Singleton() {
    }
}
```

```

public static Singleton getInstance() {
    // 第一次检查，如果实例不存在，则进入同步块
    if (instance == null) {
        // 同步块，确保同一时间只有一个线程可以进入
        synchronized (Singleton.class) {
            // 第二次检查，确保在同步块内没有其他线程创建了实例
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }
    return instance;
}
}

```

同步锁

同步锁是为了保证每个线程都能正常执行原子不可更改操作，同步监听对象/同步锁/同步监听器/互斥锁的一个标记锁。(同步监听对象、同步锁、同步监听器、互斥锁是一个意思)。

同步锁的作用是保证同一时刻，最多只有一个线程执行同步代码。

同步锁类型

- 对象类型

同步锁为 student 对象：

```

//创建同步锁对象
Student student = new Student();

//同步代码块
synchronized (student){

}

```

同步锁为 obj 对象：

```

Object obj = new Object();

synchronized(obj){

}

```

同步锁为 this 指代的对象：

```
synchronized (this){  
  
}
```

setName() 的同步锁为该方法的调用者：

```
public class Student{  
    private String name;  
  
    public synchronized void setName(String name){  
        this.name = name;  
    }  
}
```

- 类类型

同步锁为 Student.class :

```
synchronized (Student.class){  
  
}
```

静态方法 setName() 的同步锁为 Student.class :

```
public class Student{  
    private String name;  
  
    public static synchronized void setName(String name){  
        this.name = name;  
    }  
}
```

注意：

1. 线程休眠时不会释放锁
2. 线程争夺同一把锁时，线程会堵塞;争夺不同锁时，线程不会堵塞。

死锁

死锁是多线程编程中常见的一种同步问题，当两个或多个线程在执行过程中因争夺资源而造成的一种僵局。在这种情况下，每个线程都持有一些资源，同时又在等待其他线程释放它们所需的资源，如果这些资源被其他线程持有且也在等待资源，就会形成一个循环等待的状态，导致所有相关线程都无法继续执行。

死锁示例：


```

class DeadlockDemo {
    private static final Object resource1 = new Object();
    private static final Object resource2 = new Object();

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (resource1) {
                System.out.println("Thread 1: locked resource 1");
                try { Thread.sleep(100); } catch (InterruptedException e) {}
                synchronized (resource2) {
                    System.out.println("Thread 1: locked resource 2");
                }
            }
        }, "Thread-1").start();

        new Thread(() -> {
            synchronized (resource2) {
                System.out.println("Thread 2: locked resource 2");
                try { Thread.sleep(100); } catch (InterruptedException e) {}
                synchronized (resource1) {
                    System.out.println("Thread 2: locked resource 1");
                }
            }
        }, "Thread-2").start();
    }
}

```

示例中有两个线程 Thread-1 和 Thread-2，它们分别尝试锁定两个资源 resource1 和 resource2。Thread-1 首先锁定 resource1，然后尝试锁定 resource2；而 Thread-2 则相反，首先锁定 resource2，然后尝试锁定 resource1。如果 Thread-1 和 Thread-2 几乎同时运行，它们将几乎同时锁定一个资源并等待另一个资源，这样就形成了死锁。

线程间协作

等待唤醒机制

Java 中的线程等待唤醒机制是多线程同步的一个重要组成部分，它允许线程在某些条件下等待（挂起），并在满足特定条件时被其他线程唤醒。这个机制主要通过 Object 类中的 wait()、notify() 和 notifyAll() 方法实现。

以下是这些方法的基本用法和作用：

1. wait()：当一个线程需要等待某个条件成立时，它可以调用当前对象的 wait() 方法。调用 wait() 方法会导致当前线程释放它持有的对象锁，并进入等待状态，直到它被其他线程通过调用相同对象的 notify() 或 notifyAll() 方法唤醒。线程在等待期间不会释放对象锁，这是为了避免在等待时其他线程可以访问该对象。

2. `notify()`：当条件成立时，持有对象锁的线程可以调用该对象的 `notify()` 方法。
`notify()` 方法会随机唤醒一个正在该对象上等待的线程。被唤醒的线程会进入就绪状态，并在重新获得对象锁后继续执行。
3. `notifyAll()`：类似于 `notify()`，但 `notifyAll()` 会唤醒所有在该对象上等待的线程。所有被唤醒的线程将竞争获取对象锁，并在获得锁后继续执行。

示例：

```
public class WaitNotifyExample {
    public static void main(String[] args) {
        AtomicBoolean condition = new AtomicBoolean(false); // 支持原子操作的布尔变量
        final Object lock = new Object();
        Thread waitingThread = new Thread(() -> {
            synchronized (lock) {
                while (!condition.get()) {
                    try {
                        System.out.println("Waiting for condition...");
                        lock.wait(); // 等待
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("Condition met, continuing execution.");
            }
        });

        Thread notifyingThread = new Thread(() -> {
            synchronized (lock) {
                // 改变条件
                condition.set(true);
                System.out.println("Condition changed, notifying waiting thread.");
                lock.notify(); // 唤醒一个等待的线程
            }
        });

        waitingThread.start();
        notifyingThread.start();
    }
}
```

运行结果：

```
Waiting for condition...
Condition changed, notifying waiting thread.
Condition met, continuing execution.
```

`wait()` 和 `sleep()` 的对比

1. 方法来源

`sleep` 方法是 `Thread` 类的静态方法。 `wait` 方法是 `Object` 类的方法，与对象的锁相关联。

2. 锁状态

当调用 `sleep` 方法时，当前线程不会释放任何锁。它只是简单地暂停执行指定的时间量，期间不会影响其他线程。调用 `wait` 方法时，当前线程必须拥有对象的锁，并且会释放这个锁，进入等待状态。其他线程可以在此期间获得该锁。

3. 目的

`sleep` 主要用于让当前线程暂停执行一段时间，以便让其他线程有机会执行，但它并不关心其他线程是否需要该资源。 `wait` 用于线程间的协作，当一个线程需要等待某些条件成立时，它会释放锁并等待，直到被其他线程通过 `notify` 或 `notifyAll` 唤醒。

4. 中断处理

`sleep` 方法可以被中断，如果线程在 `sleep` 期间被中断，会抛出 `InterruptedException`。 `wait` 方法也可以被中断，如果线程在等待期间被中断，同样会抛出 `InterruptedException`，并且会重新获得对象锁。

5. 唤醒机制

`sleep` 方法没有唤醒机制，线程会在指定时间后自动恢复执行。 `wait` 方法需要其他线程调用相同对象的 `notify` 或 `notifyAll` 方法来唤醒。被唤醒的线程需要重新竞争获取对象锁才能继续执行。

6. 响应性

`sleep` 方法不会响应 `notify` 或 `notifyAll` 调用，即使在 `sleep` 期间调用了这些方法，`sleep` 也不会提前结束。 `wait` 方法会使线程对 `notify` 或 `notifyAll` 调用变得敏感，一旦调用这些方法，等待的线程可能会被唤醒。

7. 使用场景

`sleep` 通常用于简单的时间延迟，不需要考虑线程间的协作。 `wait` 通常用于需要线程间通信和同步的场景，例如生产者-消费者问题。

线程间通讯

应用场景：

1. 生产者-消费者问题

这是 `wait` 和 `notify` 最常见的应用场景之一。生产者在生产物品后使用 `notify()` 唤醒消费者，消费者在消费完物品后使用 `wait()` 等待新的生产。

2. 条件变量

在需要线程等待某些条件成立时，可以使用 `wait()` 方法。例如，一个线程可能需要等待另一个线程完成初始化操作。

3. 同步容器

如 `BlockingQueue`，内部使用 `wait` 和 `notify` 来实现线程安全的队列操作。

生产者-消费者示例：

```

import java.util.LinkedList;

public class ProducerConsumerDemo {
    // 共享资源
    private final LinkedList<Integer> sharedResource = new LinkedList<>();
    // 仓库的最大容量
    private static final int MAX_SIZE = 10;

    // 生产者线程
    class Producer extends Thread {
        public void run() {
            while (true) {
                // 同步块，确保线程安全
                synchronized (sharedResource) {
                    // 如果仓库满了，生产者等待
                    while (sharedResource.size() == MAX_SIZE) {
                        System.out.println("Producer waiting. Queue is full.");
                        try {
                            sharedResource.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    // 生产商品
                    sharedResource.add(1);
                    System.out.println("Producer produced an item. Total: " + sharedResource.size());
                    // 通知消费者
                    sharedResource.notifyAll();
                }
                // 模拟生产时间
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    // 消费者线程
    class Consumer extends Thread {
        public void run() {
            while (true) {
                // 同步块，确保线程安全
                synchronized (sharedResource) {
                    // 如果仓库空了，消费者等待
                    while (sharedResource.isEmpty()) {
                        System.out.println("Consumer waiting. Queue is empty.");
                        try {
                            sharedResource.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    // 消费商品
    sharedResource.remove();
    System.out.println("Consumer consumed an item. Total: " + share);
    // 通知生产者
    sharedResource.notifyAll();
}
// 模拟消费时间
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

public static void main(String[] args) {
    ProducerConsumerDemo demo = new ProducerConsumerDemo();
    Producer producer = demo.new Producer();
    Consumer consumer = demo.new Consumer();

    producer.start();
    consumer.start();
}
}

```

等待唤醒工具 - LockSupport

LockSupport 是 Java 并发包 `java.util.concurrent` 中的一个类，它提供了基本的线程阻塞和唤醒原语。以下是 LockSupport 类提供的主要方法及其介绍：

- `park()`、`park(Object blocker)`
阻塞当前线程，直到其他线程调用 `unpark` 方法为当前线程发放许可证或者当前线程被中断。 `blocker` 对象用于记录导致线程阻塞的原因。
- `parkNanos(long nanos)`、`parkNanos(Object blocker, long nanos)`
阻塞当前线程最多 `nanos` 纳秒时间，或者直到获得许可证或线程被中断。 `blocker` 对象用于记录导致线程阻塞的原因。
- `parkUntil(long deadline)`、`parkUntil(Object blocker, long deadline)`
阻塞当前线程直到指定的截止时间（以毫秒为单位），或者直到获得许可证或线程被中断。 `blocker` 对象用于记录导致线程阻塞的原因。
- `unpark(Thread thread)`
唤醒指定的线程，如果该线程已经被 `park` 方法阻塞。如果该线程尚未被阻塞，那么下一次调用 `park` 方法时将立即返回。

LockSupport 的方法使用许可证（permit）的概念来控制线程的阻塞和唤醒。每个线程都有一个许可证状态，park 方法在没有许可证的情况下会阻塞线程，而 unpark 方法则为线程发放许可证。值得注意的是，许可证不会累积，即使多次调用 unpark，每个线程最多只能持有一个许可证。

使用示例：

```
public class LockSupportExample {
    public static void main(String[] args) {
        // 创建共享数据
        AtomicReference<String> data = new AtomicReference<>();

        // 消费者线程
        Thread consumerThread = new Thread(() -> {
            System.out.println("消费者等待数据...");
            // 消费者等待生产者生产数据
            LockSupport.park();
            System.out.println("消费者消费数据：" + data);
        });

        // 生产者线程
        Thread producerThread = new Thread(() -> {
            // 生产者生产数据
            data.set("数据[" + System.currentTimeMillis() + "]");
            System.out.println("生产者生产了数据：" + data);
            // 生产者唤醒消费者
            LockSupport.unpark(consumerThread);
        });

        // 启动线程
        consumerThread.start();
        producerThread.start();
    }
}
```

实现先进先出互斥锁：

```
public class FIFOMutex {
    //锁状态
    private final AtomicBoolean locked = new AtomicBoolean(false);

    //线程等待队列
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<>();

    /**
     * 获取锁
     */
    public void lock(){
        //记录线程是否被中断
        boolean interrupted = false;
```

```

//获取当前线程
Thread current = Thread.currentThread();
//加入线程等待队列
waiters.add(current);

//获取锁
while(waiters.peek() != current || !locked.compareAndSet(false, true)){
    //如果当前线程不是等待队列的第一个线程或者锁已经被其他线程获取，则阻塞当前线
    LockSupport.park(this);

    //当前线程等待时被中断，记录中断事件
    if(Thread.interrupted()){
        interrupted = true;
    }
}

//获取锁之后将当前线程移出队列
waiters.remove();
if(interrupted) {
    //在退出时将中断状态恢复，确保线程能够响应中断请求
    current.interrupt();
}
}

/**
 * 释放锁
 */
public void unlock(){
    locked.set(false);
    LockSupport.unpark(waiters.peek());
}

public static void main(String[] args) {
    FIFOMutex mutex = new FIFOMutex();

    // 创建并启动线程
    Thread thread1 = new Thread(() -> {
        mutex.lock();
        try {
            System.out.println("线程1获取了锁");
            // 模拟一些任务执行时间
            try {
                Thread.sleep(1000);
                System.out.println("线程1睡眠1秒");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println("线程1被中断");
            }
        } finally {
            mutex.unlock();
            System.out.println("线程1释放了锁");
        }
    });
}

```

```

Thread thread2 = new Thread(() -> {
    mutex.lock();
    try {
        System.out.println("线程2获取了锁");
        thread1.interrupt();
    } finally {
        mutex.unlock();
        System.out.println("线程2释放了锁");
    }
});

// 启动线程
thread1.start();
thread2.start();
}
}

```

注意事项：

- 线程在 `park` 后可能因为中断而返回，但不抛出 `InterruptedException` 异常，因此需要手动检查中断状态。
- 避免虚假唤醒是重要的，应该总是在循环中使用 `park` 方法，并检查相应的条件。

高级同步工具

显式锁 - Lock 接口

Java中的显式锁主要是指 `java.util.concurrent.locks` 包中的锁接口和实现类，它们提供了比传统 `synchronized` 关键字更加灵活和强大的线程同步机制。以下是显式锁的主要组成部分和特性：

`Lock` 是所有显式锁的顶层接口，它定义了锁的基本操作，如获取、释放锁，以及一些高级特性。

主要方法：

- `lock()` : 获取锁，如果锁不可用，调用线程将被阻塞，直到锁被获取。
- `unlock()` : 释放锁。
- `tryLock()` : 尝试非阻塞获取锁，如果锁不可用，则立即返回 `false`。
- `tryLock(long time, TimeUnit unit)` : 尝试在给定的时间内获取锁，如果超时仍未获取到锁，则返回 `false`。
- `lockInterruptibly()` : 可中断地获取锁，如果当前线程在等待锁的过程中被中断，则会抛出 `InterruptedException`。
- `newCondition()` : 创建与此锁相关联的新条件变量。

ReentrantLock 类

ReentrantLock 是 Lock 接口的一个实现，支持相同线程的重入，即同一个线程可以多次获取同一个锁。

- 特性

1. 重入性(Reentrant): 一个线程可以多次获取同一个锁。可重入性是通过内部计数器来实现的，每次线程获取锁时，计数器增加，每次释放锁时，计数器减少。只有当计数器达到零时，锁才真正被释放，其他线程才有机会获取这个锁。synchronized 和 ReentrantLock 均支持可重入性。
2. 公平性: 可以选择是否按照线程等待的顺序来获取锁。ReentrantLock 可以是公平锁或非公平锁。公平锁按照线程请求的顺序分配锁，而非公平锁则无序分配。synchronized 为非公平锁。

- 可重入性

```
public static void main(String[] args){
    // 创建一个可重入锁
    Lock lock = new ReentrantLock();//非公平锁， new ReentrantLock(true) 为公平锁
    lock.lock();
    try {
        System.out.println("线程 " + Thread.currentThread().getName() + " 第一次获取
        // 再次获取同一把锁
        lock.lock();
        try {
            System.out.println("线程 " + Thread.currentThread().getName() + " 第二次
        } finally {
            // 释放第二次获取的锁
            lock.unlock();
        }
    } finally {
        // 释放第一次获取的锁
        lock.unlock();
    }
}
```

- 线程安全的栈

```
public class ThreadSafeStack<E> {
    private final Stack<E> stack = new Stack<>();
    private final ReentrantLock lock = new ReentrantLock();

    // 向栈中添加元素
    public void push(E item) {
        lock.lock();
        try {
            stack.push(item);
        }
    }
}
```

```

        } finally {
            lock.unlock();
        }
    }

    // 从栈中移除元素
    public E pop() {
        lock.lock();
        try {
            return stack.isEmpty() ? null : stack.pop();
        } finally {
            lock.unlock();
        }
    }
}

```

处理中断:

```

public class InterruptibleLockExample {
    private final Lock lock = new ReentrantLock();

    public void performAction() {
        try {
            // 尝试获取锁，如果线程被中断，则会抛出InterruptedException
            lock.lockInterruptibly();

            // 模拟一些需要长时间执行的操作
            try {
                Thread.sleep(3000); // 假设这里是一个耗时的操作
                System.out.println("Action performed.");
            } catch (InterruptedException e) {
                // 如果在执行耗时操作时被中断，会捕获InterruptedException
                System.out.println("Action interrupted during execution.");
                throw e;
            } finally {
                lock.unlock();
            }
        } catch (InterruptedException e) {
            //在此处处理中断
            System.out.println(Thread.currentThread().getName() + " was interrupted");
            // 重新设置中断状态，以便调用者可以处理中断
            Thread.currentThread().interrupt();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        final InterruptibleLockExample example = new InterruptibleLockExample();
        Thread thread = new Thread(() -> {
            example.performAction();
        });
    }
}

```

```

thread.start();

//中断main线程1秒
Thread.sleep(1000);

// 中断线程，尝试中断等待锁的过程
thread.interrupt();
}
}

```

ReadWriteLock 接口

ReadWriteLock 接口扩展了 Lock 接口，允许多个读操作同时进行，但写操作是排他的。

主要组件：

readLock()：返回一个用于读取操作的锁。

writeLock()：返回一个用于写入操作的锁。

	读锁	写锁
读锁	不互斥	互斥
写锁	互斥	互斥

读写锁互斥情况示例：

```

public class ReadWriteLockExample {

    private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

    public void read() {
        readWriteLock.readLock().lock(); // 获取读锁
        try {
            System.out.println(Thread.currentThread().getName() + " 正在读取数据");
            Thread.sleep(1000); // 模拟读取操作
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            readWriteLock.readLock().unlock(); // 释放读锁
        }
    }

    public void write() {
        readWriteLock.writeLock().lock(); // 获取写锁
        try {
            System.out.println(Thread.currentThread().getName() + " 正在写入数据");
            Thread.sleep(1000); // 模拟写入操作
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {

```

```

        readWriteLock.writeLock().unlock(); // 释放写锁
    }
}

public static void main(String[] args) throws InterruptedException {
    ReadWriteLockExample example = new ReadWriteLockExample();

    // 创建并启动读线程，同时读取说明读锁不互斥
    Thread reader1 = new Thread(() -> example.read(), "读取线程1");
    Thread reader2 = new Thread(() -> example.read(), "读取线程2");
    reader1.start();
    reader2.start();

    // 创建并启动写线程，依次写入说明写锁互斥
    Thread writer1 = new Thread(() -> example.write(), "写入线程1");
    Thread writer2 = new Thread(() -> example.write(), "写入线程2");
    writer1.start();
    writer2.start();

    //依次进行读取和写入说明读取和写入操作互斥
    Thread reader3 = new Thread(() -> example.write(), "读取线程3");
    Thread reader4 = new Thread(() -> example.read(), "读取线程4");
    Thread writer = new Thread(() -> example.write(), "写入线程");
    reader3.start();
    writer.start();
    reader4.start();
}
}

```

实现高并发容器示例：

```

public class HighConcurrencyContainer<K, V> {
    // 读写锁
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    // 存储数据的线程安全Map
    private final ConcurrentHashMap<K, V> map = new ConcurrentHashMap<>();
    // 用于统计元素数量的原子变量
    private final AtomicInteger count = new AtomicInteger(0);

    public void put(K key, V value) {
        lock.writeLock().lock();
        try {
            map.put(key, value);
            count.incrementAndGet();
        } finally {
            lock.writeLock().unlock();
        }
    }

    public V get(K key) {
        lock.readLock().lock();
    }
}

```

```

        try {
            return map.get(key);
        } finally {
            lock.readLock().unlock();
        }
    }

    public V remove(K key) {
        lock.writeLock().lock();
        try {
            V value = map.remove(key);
            if (value != null) {
                count.decrementAndGet();
            }
            return value;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public int size() {
        return count.get();
    }

    public void clear() {
        lock.writeLock().lock();
        try {
            map.clear();
            count.set(0);
        } finally {
            lock.writeLock().unlock();
        }
    }
}

```

ReentrantReadWriteLock 类

ReentrantReadWriteLock 是 ReadWriteLock 接口的一个实现，支持重入的读写锁。

引入 ReentrantReadWriteLock 的几个主要原因：

- 锁的公平性：ReentrantReadWriteLock 允许选择公平性。公平锁会按照线程请求锁的顺序来分配锁，而不是随机选择，这有助于防止线程饥饿问题。
- 锁的可重入性：ReentrantReadWriteLock 是可重入的，这意味着同一线程可以多次获取同一锁，而不会阻塞。这对于递归方法或者多层调用非常有用。
- 条件变量的支持：ReentrantReadWriteLock 提供了与锁关联的条件变量，可以用来实现更复杂的线程间协作，比如等待特定条件的发生。

示例 - 写锁的重入：

```

public class ReentrantReadWriteLockRecursiveExample {
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private int count = 0;

    public void increment() {
        lock.writeLock().lock();
        try {
            count++;
            System.out.println(Thread.currentThread().getName() + " incremented cou
            if (count < 5) increment(); // 递归调用
        } finally {
            lock.writeLock().unlock();
        }
    }

    public static void main(String[] args) {
        ReentrantReadWriteLockRecursiveExample example = new ReentrantReadWriteLock
        Thread thread = new Thread(example::increment, "IncrementThread");
        thread.start();
    }
}

```

StampedLock 类

StampedLock 是一个较新的锁实现，旨在提供一种更高效的读写锁机制。它使用三种类型的锁状态：乐观读、悲观读和写锁。StampedLock 通过返回一个“stamp”（一个长整型值）锁标识来实现锁的获取和释放。这个 stamp 可以用于后续的锁操作，比如尝试获取锁或者释放锁。

乐观读 (Optimistic Read)

乐观读是一种并发控制机制，它假设在读取数据时，其他线程或进程不会修改数据。在读取数据时，乐观读不会立即锁定数据，而是在读取后检查数据是否在读取过程中被修改。如果数据在读取过程中没有被修改，那么可以安全地使用读取的数据；如果被修改了，乐观读将重新读取数据或执行其他操作。乐观读适用于写操作不频繁的场景，因为它可以减少锁的开销。

悲观读 (Pessimistic Read)

悲观读与乐观读相对，它假设在读取数据时，其他线程或进程很可能会修改数据。因此，在读取数据之前，悲观读会先锁定数据，确保在读取过程中数据不会被修改。这可以防止数据不一致的问题，但可能会增加锁的开销，特别是在高并发的场景下。

写锁 (Write Lock)

写锁是一种排他锁，它确保在写入数据时，没有其他线程或进程可以读取或写入同一数据。当一个线程或进程持有写锁时，其他想要访问相同数据的线程或进程必须等待，直到写锁被释放。写锁是必要的，以防止数据不一致和竞态条件，但它也可能导致性能瓶颈，特别是在高并发的写操作中。

主要方法：

- tryOptimisticRead() : 尝试乐观读锁。
- readLock() : 获取读锁。
- writeLock() : 获取写锁。
- tryConvertToWriteLock() : 尝试将读锁转换为写锁。

线程饥饿

一个或多个线程因为无法获得必需的资源或调度，导致长时间或无限期地等待，从而无法执行的情况。这种情况通常发生在有多个线程竞争有限资源时，某些线程可能会因为调度策略或其他线程的长时间占用资源而得不到执行的机会。

解决线程饥饿的示例：

```
public class StampedLockDemo {
    private final StampedLock stampedLock = new StampedLock();
    private int data = 0;

    public void read() {
        long stamp = stampedLock.tryOptimisticRead(); // 尝试乐观读锁
        int value = data;
        boolean valid = stampedLock.validate(stamp); // 验证锁是否仍然有效
        if (!valid) {
            stamp = stampedLock.readLock(); // 如果无效，尝试获取悲观读锁
            try {
                value = data; // 重新读取数据
                Thread.sleep(1000); // 模拟读取操作
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            } finally {
                stampedLock.unlockRead(stamp); // 释放悲观读锁
            }
        }
        System.out.println(Thread.currentThread().getName() + " 读取数据: " + value)
    }

    public void write(int value) {
        long stamp = stampedLock.writeLock(); // 获取写锁
        try {
            data = value;
            Thread.sleep(3000); // 模拟写入操作
            System.out.println(Thread.currentThread().getName() + " 写入数据: " + value)
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally {
            stampedLock.unlockWrite(stamp); // 释放写锁
        }
    }

    public static void main(String[] args) throws InterruptedException {
        StampedLockDemo demo = new StampedLockDemo();
        ExecutorService executorService = Executors.newFixedThreadPool(10);
    }
}
```

```

// 创建少量的写线程
executorService.submit(() -> {
    try {
        Thread.sleep(2000); // 两秒后写入
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    demo.write( 100);
});
executorService.submit(() -> {
    try {
        Thread.sleep(4000); // 四秒后写入
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    demo.write( 200);
});

// 创建大量的读线程
for (int i = 0; i < 10; i++) {
    executorService.submit(() -> {
        // 无限读取，会导致线程饥饿
        while (true) {
            demo.read();
            Thread.sleep(1000); // 睡眠1秒
        }
    });
}
}
}

```

使用显式锁的注意事项：

锁的释放：必须显式调用 `unlock()` 来释放锁，否则可能导致死锁。

锁的获取与释放：应在 `finally` 块中释放锁，以确保即使发生异常也能释放锁。

避免死锁：显式锁的使用需要更加小心，以避免死锁的发生。

对比 Lock 与 synchronized

- 中断性

Lock 允许中断一个正在等待获取锁的线程，通过 `lockInterruptibly()` 方法实现。
`synchronized` 没有提供中断等待锁的机制。

- 尝试非阻塞获取

Lock 提供了 `tryLock()` 方法，允许非阻塞地尝试获取锁，如果锁不可用则立即返回。
`synchronized` 没有提供非阻塞获取锁的方法。

- 超时获取

Lock 允许尝试在给定的时间内获取锁，通过 `tryLock(long, TimeUnit)` 方法实现。

`synchronized` 没有超时机制。

- 公平性

`Lock` 可以构造为公平锁，这意味着等待时间最长的线程将首先获得锁，有助于避免线程饥饿问题。 `synchronized` 没有提供公平性选项，其锁的获取顺序依赖于 JVM 的实现。

公平性指是否选择按照线程等待的顺序来获取锁，按照线程等待顺序来获取为公平锁，反之为非公平锁。

- 多个条件变量

`Lock` 可以配合多个 `Condition` 对象使用，允许更细粒度的线程间协作。 `synchronized` 只能与一个条件变量（`Object` 的 `wait()` 和 `notify()/notifyAll()` 方法）一起使用。

- 实现灵活性

`Lock` 是一个接口，可以有多种实现，例如 `ReentrantLock`、`ReadWriteLock` 等，提供不同的锁定策略和优化。 `synchronized` 是 Java 语言的关键字，其实现和行为由 JVM 控制，不够灵活。

- 使用复杂性

`Lock` 需要显式地获取和释放锁，并且在发生异常时需要在 `finally` 块中释放锁，增加了代码的复杂性。 `synchronized` 可以自动获取和释放锁，使用起来更简单。

- 性能

在某些情况下，`synchronized` 可能比 `Lock` 有更好的性能，因为它是由 JVM 直接支持的，并且有更少的上下文切换开销。然而，`Lock` 由于其高级特性，可能在复杂的同步场景中提供更好的性能。

- 可扩展性

`Lock` 由于其丰富的 API 和灵活性，更适合构建复杂的并发控制结构。 `synchronized` 由于其简单性，适合简单的同步需求，但在复杂的同步场景中可能不够用。

条件变量 - Condition

与显式锁配合使用的条件变量允许线程在某些条件不满足时挂起，并在条件满足时被唤醒。

主要方法：

`await()`：等待直到被唤醒。

`signal()`：唤醒一个等待的线程。

`signalAll()`：唤醒所有等待的线程。

示例：

```

public class ConditionExample {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private boolean conditionMet = false;

    public void waitForCondition() throws InterruptedException {
        lock.lock();
        try {
            // 等待条件满足
            while (!conditionMet) {
                System.out.println("Waiting for condition to be met.");
                condition.await();
            }
            System.out.println("Condition is met. Proceeding with the task.");
        } finally {
            lock.unlock();
        }
    }

    public void meetCondition() {
        lock.lock();
        try {
            // 改变条件状态
            conditionMet = true;
            // 唤醒等待的线程
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ConditionExample example = new ConditionExample();

        // 创建一个线程等待条件
        new Thread(() -> {
            try {
                example.waitForCondition();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();

        // 创建另一个线程来满足条件
        new Thread(() -> {
            example.meetCondition();
        }).start();
    }
}

```

不可重入锁

Java提供的锁都是可重入锁，可以通过自定义锁来实现一个不可重入锁。具体示例如下：

```
public class UnReentrantLock implements Lock {
    private Thread ownerThread;
    private int lockCount;

    /**
     * 绑定已获取锁的线程，并将锁计数器设置为1
     */
    @Override
    public void lock() {
        Thread currentThread = Thread.currentThread();
        if (currentThread == ownerThread) {
            throw new IllegalStateException("Thread " + currentThread + " already h
        }
        synchronized (this) {
            while (ownerThread != null) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt(); // 恢复中断状态
                    throw new RuntimeException("Lock acquisition interrupted.", e);
                }
            }
            ownerThread = currentThread;
            lockCount = 1;
        }
    }

    /**
     * 如果当前线程在获取锁之前被中断，则此方法会抛出 InterruptedException。
     */
    @Override
    public void lockInterruptibly() throws InterruptedException {
        Thread currentThread = Thread.currentThread();
        synchronized (this) {
            while (ownerThread != null) {
                wait();
                if (Thread.interrupted()) {
                    throw new InterruptedException();
                }
            }
            ownerThread = currentThread;
            lockCount = 1;
        }
    }

    /**
     * 如果当前线程不持有锁，则重新检查中断状态并抛出异常。
     * 如果线程已经持有锁，则减少锁计数。
     */
}
```

```

@Override
public void unlock() {
    Thread currentThread = Thread.currentThread();
    synchronized (this) {
        if (currentThread != ownerThread) {
            throw new IllegalStateException("Thread " + currentThread + " does
        }
        if (lockCount == 1) {
            ownerThread = null;
            notifyAll(); // 唤醒所有等待的线程
        } else {
            lockCount--;
        }
    }
}

/**
 * 返回一个新的 Condition 实例。
 */
@Override
public Condition newCondition() {
    return null;
}

// 尝试获取锁，如果锁被占用则立即返回false
public boolean tryLock() {
    Thread currentThread = Thread.currentThread();
    synchronized (this) {
        if (currentThread == ownerThread) {
            return false;
        }
        if (ownerThread == null) {
            ownerThread = currentThread;
            lockCount = 1;
            return true;
        } else {
            return false;
        }
    }
}

/**
 * 尝试在指定等待时间内获取锁。
 * 如果成功获取锁，或者在超时前锁已被当前线程持有，则返回 true，
 * 否则在超时时返回 false。
 */
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException
    long nanos = unit.toNanos(timeout);
    Thread currentThread = Thread.currentThread();
    synchronized (this) {
        if (currentThread == ownerThread) {
            return false;
        }
    }
}

```

```

        while (ownerThread != null) {
            long startTime = System.nanoTime();
            wait(nanos);
            nanos -= (System.nanoTime() - startTime);
            if (nanos <= 0) {
                return false;
            }
        }
        ownerThread = currentThread;
        lockCount = 1;
        return true;
    }
}

public static void main(String[] args) {
    UnReentrantLock lock = new UnReentrantLock();

    // 创建一个任务，该任务将尝试获取锁并保持它一段时间
    Runnable task = () -> {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + " has acquire
            // 模拟一些工作
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println(Thread.currentThread().getName() + " was int
            }
        } finally {
            System.out.println(Thread.currentThread().getName() + " is releasin
            lock.unlock();
        }
    };

    // 创建并启动多个线程
    Thread thread1 = new Thread(task, "Thread-1");
    Thread thread2 = new Thread(task, "Thread-2");
    Thread thread3 = new Thread(task, "Thread-3");

    thread1.start();
    thread2.start();
    thread3.start();
}
}

```

线程局部存储

ThreadLocal

`ThreadLocal` 是 Java 中的一个实用程序类，它提供了线程局部变量。这些变量是线程隔离的，每个使用该变量的线程都有独立的变量副本，因此每个线程可以更改自己的副本而不会影响其他线程。

以下是 `ThreadLocal` 的一些关键特性：

1. 线程隔离：每个线程都有自己独立的变量副本，互不影响。
2. 内存管理：由于变量是线程隔离的，因此不需要担心多线程环境下的同步问题。
3. 延迟初始化：`ThreadLocal` 允许延迟初始化变量，即在第一次使用时才创建变量。
4. 清理：`ThreadLocal` 提供了 `remove()` 方法，用于在不再需要变量时手动清理，以避免内存泄漏。

以下是 `ThreadLocal` 类的主要方法及其用途：

- `void set(T value)` 为当前线程设置当前线程局部变量的值。如果线程死亡，建议手动调用 `remove()` 方法来清理，以避免潜在的内存泄漏。
- `T get()` 返回当前线程所对应的线程局部变量的值。如果该变量尚未被初始化，则会调用 `initialValue()` 方法来提供这个值。
- `T initialValue()` 当线程首次访问时，用于提供一个新值的方法。这是一个受保护的方法，可以被子类重写以提供线程局部变量的初始值。
- `void remove()` 移除当前线程与此线程局部变量的对应关系，以及当前线程的值。这是一个清理方法，用于在不再需要线程局部变量时释放资源。
- `ThreadLocal<T> withInitial(Supplier<T> supplier)` 创建一个具有给定初始值的 `ThreadLocal` 实例。这是一个静态方法，它接受一个 `Supplier<T>` 函数式接口作为参数，用于提供每个线程的初始值。
- `ThreadLocal<T> withInitial(T value)` 创建一个具有给定初始值的 `ThreadLocal` 实例。这是一个静态方法，它接受一个具体的值作为参数，用于提供每个线程的初始值。

Supplier 函数式接口

`Supplier` 接口的主要目的是提供一个类型为 `T` 的对象，而不需要显式地声明返回值的类型。

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

`@FunctionalInterface` 注解表示这是一个函数式接口，意味着它只有一个抽象方法，可以被 `lambda` 表达式或方法引用实现。

Supplier<T> 接口通常用于以下场景：

1. 延迟初始化：当你需要延迟对象的创建直到真正需要它的时候，可以使用 Supplier<T> 接口。
2. 提供实例：在需要创建对象时提供一个实例，例如在工厂模式中。
3. 函数式编程：作为参数传递给其他方法，使得代码更加灵活和模块化。

```
public static void main(String[] args) {  
    // 使用 Lambda 表达式创建 Supplier 实例  
    Supplier<String> stringSupplier = () -> "Hello, World!";  
  
    // 使用 get() 方法获取实例  
    String result = stringSupplier.get();  
    System.out.println(result);  
}
```

ThreadLocal 示例：

```
public static void main(String[] args) {  
    // 创建一个 ThreadLocal 实例来存储每个线程的随机数生成器  
    ThreadLocal<Random> randomHolder = ThreadLocal.withInitial(() -> new Random());  
  
    // 创建一个固定大小的线程池  
    ExecutorService executorService = Executors.newFixedThreadPool(5);  
  
    // 提交10个任务到线程池  
    for (int i = 0; i < 10; i++) {  
        executorService.submit(() -> {  
            // 获取当前线程的 Random 实例  
            Random random = randomHolder.get();  
            // 使用 Random 实例生成一个随机数  
            int randomNumber = random.nextInt(100);  
            // 打印当前线程的随机数和线程名称  
            System.out.println("Random number generated by " + Thread.currentThread()  
                .getName() + ": " + randomNumber);  
        });  
    }  
  
    // 关闭线程池  
    executorService.shutdown();  
}
```

线程间共享 ThreadLocal - InheritableThreadLocal

InheritableThreadLocal 是 Java 中 ThreadLocal 的一个子类，它允许子线程继承父线程中的线程局部变量。这在某些需要在父子线程间传递数据的场景中非常有用，比如在处理请求时，需要将请求的上下文信息传递给子线程。

示例：

```
public class InheritableThreadLocalExample {
    // 创建一个 InheritableThreadLocal 实例
    private static final InheritableThreadLocal<String> inheritableThreadLocal = new InheritableThreadLocal<>() {
        @Override
        protected String initialValue() {
            return "Parent thread value";
        }
    };

    public static void main(String[] args) {
        // 在父线程中设置值
        inheritableThreadLocal.set(Thread.currentThread().getName() + "-value");

        // 打印父线程中的值
        System.out.println("Parent thread value: " + inheritableThreadLocal.get());

        // 创建并启动子线程
        new Thread(() -> {
            // 子线程中获取值
            System.out.println("Value in child thread: " + inheritableThreadLocal.get());
        }).start();
    }
}
```

Executor框架 - 线程池

Executor框架

Executor 框架是一个用于管理线程的执行的框架，它允许你将任务的提交与任务的执行分离开来。

核心类和接口：

- **Executor 接口** Executor是Executor 框架的核心接口，它只有一个方法 `execute(Runnable command)`，用于提交一个 `Runnable` 任务以供执行。
- **Executors 类** Executors 是一个工厂类，提供了一些静态方法来创建不同类型的 `Executor` 实现，如固定大小的线程池、可缓存的线程池、单线程执行器和计划线程池等。
- **ExecutorService 接口** ExecutorService 是 `Executor` 的子接口，它提供了管理任务生命周期的方法，如 `submit()` 用于提交一个可返回结果的任务，`shutdown()` 用于关闭执行器，`invokeAll()` 用于批量执行任务。
- **ScheduledExecutorService 接口**：ScheduledExecutorService 是 `ExecutorService` 的子接口，它增加了定时任务和周期性任务的执行能力。

线程池

线程池是 `Executor` 框架的一个具体实现，它管理一个线程集合来执行任务。线程池的主要优点包括：

1. 资源管理：线程池可以限制并发执行的线程数量，从而避免因为创建过多的线程而导致的资源耗尽。
2. 性能提升：通过重用已存在的线程来执行新的任务，减少了线程创建和销毁的开销。
3. 控制并发级别：可以根据不同的场景设置合适的线程池大小，控制并发执行的线程数量。
4. 提高线程的可管理性：线程池提供了线程的监控、调试和维护的机制。

线程池类型：

1. 固定大小的线程池

`Executors.newFixedThreadPool(int nThreads)` 创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。

2. 可缓存的线程池

`Executors.newCachedThreadPool()` 创建一个可根据需要创建新线程的线程池，对于短生命周期的异步任务非常合适。

3. 单线程执行器

`Executors.newSingleThreadExecutor()` 创建一个只有一个线程的线程池，它可以确保所有提交的任务按顺序执行。

4. 计划线程池

`Executors.newScheduledThreadPool(int corePoolSize)` 创建一个线程池，它可以安排在给定的延迟后运行任务，或者定期执行任务。

任务提交

- `execute(Runnable command)` 提交一个 `Runnable` 任务以供执行，不返回任何结果。
- `submit(Callable<T> task)` 提交一个 `Callable` 任务以供执行，并返回一个 `Future` 对象，该对象可以用来检查计算是否完成，等待计算结果，或者取消任务。
- `invokeAll(Collection<? extends Callable<T>> tasks)` 批量执行一组 `Callable` 任务，并返回一个 `List`，其中包含每个任务的 `Future`。

线程池的关闭

- `shutdown()` 启动一次顺序的关闭，执行以前提交的任务，不接受新任务。
- `shutdownNow()` 尝试停止所有正在执行的任务，并返回等待执行的任务列表。
- `awaitTermination(long timeout, TimeUnit unit)` 等待线程池终止，直到指定的等待时间。

示例

`ExecutorService` 示例：

```

public class ExecutorServiceExample {
    public static void main(String[] args) {
        // 创建一个固定大小的线程池
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // 提交任务给线程池
        for (int i = 0; i < 10; i++) {
            int taskNumber = i;
            executor.submit(() -> {
                System.out.println("Executing task " + taskNumber + " on thread " +
                    return taskNumber;
            }); // 参数为supplier函数，在线程池中执行该函数，并返回future对象
        }

        // 关闭线程池
        executor.shutdown();

        try {
            // 等待线程池中所有任务完成
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                // 终止当前正在执行的任务
                executor.shutdownNow();
                // 等待任务响应中断
                if (!executor.awaitTermination(60, TimeUnit.SECONDS))
                    System.err.println("Pool did not terminate");
            }
        } catch (InterruptedException ie) {
            // 重新设置中断标志
            Thread.currentThread().interrupt();
            // 终止当前正在执行的任务
            executor.shutdownNow();
        }

        System.out.println("Finished all threads");
    }
}

```

线程安全和并发工具

内存可见性 - volatile 关键字

内存可见性问题

内存可见性问题是指在多线程环境中，当一个线程修改了共享变量的值，其他线程可能无法立即看到这个修改。这是因为每个线程可能在自己的私有内存（如处理器缓存）中缓存了共享变量的副本，而不是直接从主内存中读取。这就导致了不同线程之间的数据不一致性。

内存可见性的原因：

1. 缓存一致性：现代处理器使用缓存来减少访问主内存的延迟。每个处理器可能有自己的缓存，这导致了缓存一致性问题。
2. 指令重排：为了优化性能，编译器和处理器可能会对指令进行重排序，这可能会影响内存可见性。
3. 线程间的数据隔离：每个线程可能有自己的栈，这意味着线程局部变量不会在线程之间共享。

volatile 关键字

volatile 意思是不稳定的、易变的。volatile 是 Java 中的一个关键字，它可以解决内存可见性问题。当一个变量被声明为 volatile 时，它有以下特性：

- 保证可见性：对 volatile 变量的写操作对所有线程都是可见的。当一个线程修改了 volatile 变量的值，新值会立即被写入主内存，其他线程读取该变量时会从主内存中读取最新值。
- 禁止指令重排：volatile 关键字会阻止编译器和处理器对读写操作进行重排序，确保在执行当前线程中的后续操作之前，先完成 volatile 变量的写入。

volatile 适用于以下场景：

- 状态标志：用于表示某个状态的变化，如线程的运行状态。
- 单次操作：对于简单的读写操作，volatile 可以保证可见性。

注意：volatile 只能保证可见性，不能保证复合操作（如递增、递减）的原子性。频繁使用 volatile 可能会降低程序性能，因为它需要与主内存进行同步。

示例：

```
public class VolatileExample {
    //public static volatile boolean stopped = false;
    public static boolean stopped = false;

    public static void main(String[] args) throws InterruptedException {
        //工作内存中的缓存的stopped一直为false
        new Thread(() -> {
            //如果stopped变量没有用volatile修饰，则线程无法退出循环
            while(!stopped){
            }
            System.out.println("Thread stopped");
        }).start();

        Thread.sleep(1000);

        new Thread(() -> stopped = true).start();
    }
}
```

```
}  
}
```

如果用 `volatile` 修饰 `stopped` 变量时，线程将从主内存中读取 `stopped` 的值，以上示例中的线程将不会陷入死循环。

原子性

原子性 (Atomicity) 是指一个操作或者一组操作要么全部执行并且执行过程不会被任何因素打断，要么就全部都不执行。在多线程环境中，原子性非常重要，因为它确保了当多个线程同时访问和修改同一个变量时，每个线程都能看到变量的完整和一致的状态。

在Java中，原子性可以通过以下几种方式实现：

- `synchronized` 关键字：通过使用 `synchronized` 关键字，可以确保同一时刻只有一个线程能够执行同步代码块或者同步方法，从而保证操作的原子性。
- `Lock` 接口：Java并发API中的 `Lock` 接口提供了比 `synchronized` 更灵活的锁定机制，通过显式地获取和释放锁，可以更精细地控制同步。
- `volatile` 关键字：虽然 `volatile` 关键字并不能保证复合操作的原子性，但它确保了变量的可见性，即一个线程对 `volatile` 变量的修改对其他线程是立即可见的。

原子性并发工具

Java并发框架提供了多种工具来帮助开发者实现原子性，包括：

原子变量类：如 `AtomicInteger`、`AtomicLong` 等，它们提供了一组原子操作，如增加、减少、比较和交换等。

原子数组：如 `AtomicIntegerArray`、`AtomicLongArray` 等，它们提供了对数组元素的原子操作。

原子更新器：如 `AtomicIntegerUpdater`、`AtomicLongUpdater` 等，它们允许对对象的属性进行原子更新。

原子类提供了一种机制，通过底层的硬件支持来实现原子操作。这些类利用了CAS (Compare-And-Swap) 算法，这是一种常见的无锁算法，它通过比较内存中的值和预期值是否相等来实现原子操作。

CAS算法

比较：线程检查内存中的值 (V) 是否与预期值 (A) 相等。

交换：如果相等，线程将内存中的值更新为新值 (B)。

返回：CAS操作的结果，通常是布尔值，表示操作是否成功。

优点

无锁：CAS提供了一种无锁的线程同步机制，避免了锁的开销和潜在的死锁问题。

性能：在没有线程竞争的情况下，CAS操作的性能通常优于传统的锁机制。

减少上下文切换：由于避免了线程阻塞和唤醒，CAS可以减少上下文切换的开销。

缺点

ABA问题：如果一个变量的值原来是A，变成了B，然后又变回A，CAS操作将无法检测到这种变化，因为它只比较当前值是否与预期值相等。

循环时间长开销大：如果多个线程同时竞争同一个变量，CAS操作可能会进入长时间的自旋（spin），导致CPU资源的浪费。

只能保证单个变量的原子性：CAS算法不能保证多个变量的复合操作的原子性。

```
public class AtomicExample {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        System.out.println("Incremented value: " + count.getAndIncrement());
    }

    public void decrement() {
        System.out.println("Decrement value: " + count.getAndDecrement());
    }

    public int getCount() {
        return count.get();
    }

    public static void main(String[] args) {
        AtomicExample example = new AtomicExample();

        example.increment();
        example.decrement();

        System.out.println("Current count: " + example.getCount());
    }
}
```