Java并发编程

JUC概述

1. `Java.util.concurrent`
2. 进程与线程
    1. 操作系统进行资源分配和调度的基本单位；
    2. 线程：程序执行的最小单位；
3. 线程状态：
    1. `NEW` 新建
    2. `RUNABLE` 准备就绪
    3. `BLOCKED` 阻塞
    4. `WAITING` 一直等
    5. `TIMED_WAITING` 过时不候
    6. `TERMINATED` 终结
4. `wait/sleep`
    1. `wait` 是 `Object` 方法，任何对象实例都可以调用； `sleep` 是 `Thread` 的静态方法
    2. `sleep` 不会释放锁，也不需要占用； `wait` 会释放锁，调用前提是当前线程占有锁
    3. 都可以被 `Interrupt` 方法打断
    4. `wait` 在哪里等待，就会在哪里唤醒，继续执行下方代码
5. 并发与并行
    1. 串行：一次只取一个任务，排队一个执行
    2. 并行：多个任务同时执行，后汇总
    3. 并发：同一时刻多个线程访问同一资源，多线程对点
6. 管程（`Monitor`）——锁，同步机制，保证同一时间，只有一个线程访问被保护数据或代码

1. JVM 同步基于进入和退出，使用管程对象管理实现

7. 用户线程与守护线程
    1. 用户线程：自定义线程
    2. 守护线程：比如垃圾回收
    3. 主线程结束，用户线程运行，JVM 存活

```
public class Main{
  public static void main(String[] args){
   //自定义用户线程
  Thread aa = new Thread(() => {
 System.out.println(Thread.currentThred().getName()+"::"+Thread.c
urrentThread().isDaemon());
    while(true){

    }
    },"aa");
    aa.start();
     System.out.println(Thread.currentThred().getName()+"over");
  }
}
//输出
//main over --> 主线程结束
//aa::false --> aa是用户线程
```

    4. 无用户线程，都是守护线程，主线程结束，JVM结束

```
public class Main{
  public static void main(String[] args){
   //自定义用户线程
  Thread aa = new Thread(() => {
 System.out.println(Thread.currentThred().getName()+"::"+Thread.c
urrentThread().isDaemon());
    while(true){
```

```
        }
        },"aa");
        //将aa设置为守护线程
        aa.setDaemon(true);
        aa.start();
         System.out.println(Thread.currentThred().getName()+"over");
      }
    }
    //输出
    //main over --> 主线程结束
    //JVM结束
```

Lock接口

1. `synchronized`
2. 创建线程方式
    1. 继承 `Thread` 类
    2. 实现 `Runnable` 接口
    3. 使用 `Callable` 接口
    4. 使用线程池
3. `Lock` 接口
    1. `ReentrantLock` 可重入锁：可重入锁是一种线程同步机制，也被称为递归锁。它允许线程在持有锁的情况下多次进入同步代码块，而不会出现死锁或其他线程同步问题
    2. `Lock` 与 `Synchronized` 区别
        1. `Lock` 不是 `Java` 语言内置，`synchronized` 是 `Java` 关键字内置
        2. `synchronized` 不需要用户手动释放锁，`Lock` 必须要用户手动释放锁
        3. `synchronized` 异常时自动释放锁，Lock不行
        4. `Lock` 可以让等待锁的线程响应终端，`synchronized` 的线程会一直等待
        5. 通过 `Lock` 可以知道是否成功获取锁，`synchronized` 不行

6. Lock 可以提高多线程效率

4. 3个售票员，卖出30张票（线程3个，资源票）

synchronized 实现

```
// 创建资源类，定义属性和方法
class Ticket {
    private int count = 30;

    public synchronized void Sale() {
        if (count > 0) {
            count--;
            System.out.println(Thread.currentThread().getName() + "卖出了一张票" + "剩下：" + count);
        }
    }
}
public class SaleTicket {
    public static void main(String[] args) {
        Ticket ticket = new Ticket();
        new Thread(new Runnable() {
            @Override
            public void run() {
                //调用卖票
                for (int i = 0; i < 40; i++) {
                    ticket.Sale();
                }
            }
        }, "aa").start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                //调用卖票
                for (int i = 0; i < 40; i++) {
                    ticket.Sale();
                }
            }
```

```
        }, "bb").start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                //调用卖票
                for (int i = 0; i < 40; i++) {
                    ticket.Sale();
                }
            }
        }, "cc").start();
    }
}
```

Lock实现

```
import java.util.concurrent.locks.ReentrantLock;
//创建资源类，定义属性和方法
class LTicket {
    private int count = 30;
    //创建可重入锁
    private final ReentrantLock lock = new ReentrantLock();

    //卖票
    public void LSale() {
        //上锁
        lock.lock();
        try {
            if (count > 0) {
                count--;
                System.out.println(Thread.currentThread().getName()
+ "卖出了一张票，" + "剩下：" + count);
            }
        } finally {
            //解锁
            lock.unlock();
        }
```

```
    }
}

public class LSaleTicket {
    public static void main(String[] args) {
        LTicket ticket = new LTicket();
        new Thread(() -> {
            for (int i = 0; i < 40; i++) {
                ticket.LSale();
            }
        }, "aa").start();
        new Thread(() -> {
            for (int i = 0; i < 40; i++) {
                ticket.LSale();
            }
        }, "bb").start();
        new Thread(() -> {
            for (int i = 0; i < 40; i++) {
                ticket.LSale();
            }
        }, "cc").start();
    }
}
```

线程间通信

1. 多线程编程步骤
    1. 创建资源类，在资源类创建属性和操作方法

       空调——资源类（高内聚，低耦合）
    2. 在资源类操作方法
        1. 判断

    2. 干活
    3. 通知
3. 创建多个线程，调用资源类的操作方法
4. 防止虚假唤醒
5. synchronized 实现

```
class Share {
    private int sum = 0;

    public synchronized void add() throws InterruptedException {
        // 判断，用if会造成虚假唤醒，wait在哪等待在哪唤醒，引起错误，所以要用while
        while (sum != 0) {
            this.wait();
        }
        // 干活
        sum++;
        System.out.println(Thread.currentThread().getName() + "::" + sum);
        // 通知
        this.notifyAll();
    }

    public synchronized void dec() throws InterruptedException {
        // 判断
        while (sum != 1) {
            this.wait();
        }
        // 干活
        sum--;
        System.out.println(Thread.currentThread().getName() + "::" + sum);
        // 通知
        this.notifyAll();
    }
```

```
}

public class ThreadDemo1 {
    public static void main(String[] args) {
        Share share = new Share();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    share.add();//+1
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }, "aa").start();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    share.dec();//-1
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }, "bb").start();
    }
}
```

6. Lock实现

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Share1 {
    private int count = 0;
```

```
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public void incr() throws InterruptedException {
        lock.lock();
        try {
            //判断
            while (count != 0) {
                condition.await();
            }
            //操作
            count++;
            System.out.println(Thread.currentThread().getName() +
"::" + count);
            //通知
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public void dec() throws InterruptedException {
        lock.lock();
        try {
            while (count != 1) {
                condition.await();
            }
            count--;
            System.out.println(Thread.currentThread().getName() +
"::" + count);
            //通知
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
```

```
    }

    public class ThreadDemon2 {
        public static void main(String[] args) {
            Share1 share = new Share1();
            new Thread(() -> {
                for (int i = 0; i < 10; i++) {
                    try {
                        share.incr();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }, "aa").start();
            new Thread(() -> {
                for (int i = 0; i < 10; i++) {
                    try {
                        share.dec();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }, "bb").start();
        }
    }
```

线程间定制通信

1. 例子：启动三个线程，按照如下要求：
    1. AA打印5次，BB打印10次，CC打印15次（按顺序）
    2. AA打印5次，BB打印10次，CC打印15次（按顺序）
    3. 进行10轮

    实现：
    1. 给线程添加标志位

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

// 创建资源类，定义属性和方法
class ShareResource {
    //定义标志位
    private int flag = 1;
    //创建Lock锁
    private Lock lock = new ReentrantLock();
    //创建3个condition
    private Condition c1 = lock.newCondition();
    private Condition c2 = lock.newCondition();
    private Condition c3 = lock.newCondition();

    // AA打印5次
    public void print5(int loop) throws InterruptedException {
        lock.lock();
        try {
            //判断
            while (flag != 1) {
                //等待
                c1.await();
            }
            //干活
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName()
+ "::" + i + "::" + loop);
            }
            //更改标识位
            flag = 2;
            //通知BB
            c2.signal();
        } finally {
            lock.unlock();
        }
```

```
    }

    // BB打印5次
    public void print10(int loop) throws InterruptedException {
        lock.lock();
        try {
            //判断
            while (flag != 2) {
                //等待
                c2.await();
            }
            //干活
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().getName()
+ "::" + i + "::" + loop);
            }
            //更改标识位
            flag = 3;
            //通知cc
            c3.signal();
        } finally {
            lock.unlock();
        }
    }

    // CC打印5次
    public void print15(int loop) throws InterruptedException {
        lock.lock();
        try {
            //判断
            while (flag != 3) {
                //等待
                c3.await();
            }
            //干活
            for (int i = 0; i < 15; i++) {
```

```
                System.out.println(Thread.currentThread().getName()
+ "::" + i + "::" + loop);
            }
            //更改标识位
            flag = 1;
            //通知aa
            c1.signal();
        } finally {
            lock.unlock();
        }
    }


}

public class ThreadDemon3 {
    public static void main(String[] args) {
        ShareResource share = new ShareResource();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    share.print5(i);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }, "AA").start();
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    share.print10(i);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }, "BB").start();
```

```
            new Thread(() -> {
                for (int i = 0; i < 10; i++) {
                    try {
                        share.print15(i);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }, "CC").start();
        }
    }
```

集合的线程安全

1. 集合（ArrayList）线程不安全演示（ArrayList 源码 add 方法并没有加锁）

```
    import java.util.ArrayList;
    import java.util.List;
    import java.util.UUID;

    public class ThreadDemon4 {
        public static void main(String[] args) {
            //创建ArrayList
            List<String> list = new ArrayList<>();
            // java.util.ConcurrentModificationException
            for (int i = 1; i < 100; i++) {
                new Thread(() -> {
                    //向集合添加内容
                    list.add(UUID.randomUUID().toString().substring(0,
    8));
                    //从集合取内容
                    System.out.println(list);
                }, i + "").start();
            }

        }
```

```
    }
```

解决方案：

1. `Vector`（方法加了`synchronized`）

```
    List<String> list = new Vector<>();
```

2. `Collections`其中的`synchronizedList`方法可以给你返回一个线程安全的`List`

```
    List<String> list = Collections.synchronizedList(new ArrayList<>
    ());
```

3. `CopyOnWriteArrayList`

```
    List<String> list = new CopyOnWriteArrayList<>();
```

原理：写时复制技术，读并发读，写独立写。先复制集合（此时读，读原来内容），将新内容写入复制集合，在写完后数据合并，再读。

```
    public boolean add(E e) {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                Object[] elements = getArray();
                int len = elements.length;
                Object[] newElements = Arrays.copyOf(elements, len +
    1);
                newElements[len] = e;
                setArray(newElements);
                return true;
            } finally {
                lock.unlock();
            }
        }
```

2. HashSet线程不安全
   解决方案：
   1. CopyOnWriteArraySet

      ```
      Set<String> set = new CopyOnWriteArraySet<>();
      ```

3. HashMap线程不安全
   解决方案：
   1. ConcurrentHashMap

      ```
      Map<String,String> map = new ConcurrentHashMap<>();
      ```

多线程锁

1. 锁的八种情况

   Synchronized：

   1. 对于**普通同步方法**，锁的是当前**实例对象**
   2. 对于**静态同步方法**，锁的是当前**类的Class对象**
   3. 对于**同步方法块**，锁是Synchronized**括号里配置的对象**

2. 公平锁和非公平锁
   - 非公平锁：线程饿死，效率高
   - 公平锁：平均，效率低

   1. ReentrantLock默认非公平锁，可以传参true变为公平锁

3. 可重入锁：synchronized（隐式--自动上下锁） Lock（显式--手动上下锁）
   可重入锁允许一个线程多次获取同一个锁，而不会导致死锁。当一个线程多次获取同一个锁时，只有当它完全释放该锁时，其他线程才能获取该锁。这种机制可以确保线程不会因为自己持有的锁而被阻塞，从而避免了死锁。

   synchronized例子：

   ```
   public class SynchronizedExample {
       public synchronized void outer() {
           System.out.println("outer method");
           inner();
   ```

```
    }

    public synchronized void inner() {
        System.out.println("inner method");
    }

    public static void main(String[] args) {
        SynchronizedExample example = new SynchronizedExample();
        example.outer();
    }
}
```

在上面的例子中，我们定义了一个名为 SynchronizedExample 的类，其中包含了两个同步方法 outer 和 inner 。在 outer 方法中，我们使用 synchronized 关键字修饰方法，表示该方法是同步方法。在 inner 方法中，我们也使用 synchronized 关键字修饰方法。由于 synchronized 关键字是一种可重入锁，因此在调用 inner 方法时，线程可以再次获得同一个锁（为什么是同一个锁，因为此时 synchronized 关键字修饰普通方法，锁的是同一个实例对象 example ），而不会导致死锁。这个例子可以帮助理解 synchronized 关键字的概念和实现方式。

ReentrantLock例子：

```
import java.util.concurrent.locks.ReentrantLock;

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

    public void outer() {
        lock.lock();
        try {
            System.out.println("outer method");
            inner();
          } finally {
            lock.unlock();
        }
    }
```

```
        public void inner() {
            lock.lock();
            try {
                System.out.println("inner method");
            } finally {
                lock.unlock();
            }
        }

        public static void main(String[] args) {
            ReentrantLockExample example = new ReentrantLockExample();
            example.outer();
        }
    }
```

在上面的例子中，我们定义了一个名为 `ReentrantLockExample` 的类，其中包含了两个同步方法 `outer` 和 `inner` 。在 `outer` 方法中，我们首先获得了锁，然后执行了一些操作，并且调用了 `inner` 方法。在 `inner` 方法中，我们又获得了同一个锁（同一个 `lock` ），并执行了一些操作。在最后，我们释放了锁。由于我们使用的是可重入锁，因此在调用 `inner` 方法时，线程可以再次获得同一个锁，而不会导致死锁。这个例子可以帮助理解可重入锁的概念和实现方式。

4. 死锁
    1. 原因
    2. 验证是否死锁
        1. `jps` ，类似于linux ps -ef
        2. `jstack` ， `jvm` 自带堆栈跟踪工具

`Callable`接口

特点：

- `call()` 有返回值， `Runnable` 的 `run()` 没有
- `call()` 可以引发异常， `run()` 不行

FutureTask 实现

```
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

// Runnable接口
class MyThread1 implements Runnable {
    @Override
    public void run() {

    }
}

// Callable接口
class MyThread2 implements Callable {
    @Override
    public Integer call() throws Exception {
        return 200;
    }
}

public class Main {
    public static void main(String[] args) {
        // FutureTask
        FutureTask<Integer> futureTask = new FutureTask<>(new
MyThread2());
        // lambda表达式
        FutureTask<Integer> futureTask2 = new FutureTask<>(() -> {
            return 200;
        });
    }
}
```

JUC 强大辅助类

1. 减少计数 CountDownLatch
    1. 有参构造，初始化计数
    2. await() 当计数 !=0 等待
    3. countDown() 计数-1

```
import java.util.concurrent.CountDownLatch;

// 演示CountDownLatchDemo
public class CountDownLatchDemo {
    public static void main(String[] args) throws
InterruptedException {
        // 创建CountDownLatch，初始化计数六个同学
        CountDownLatch countDownLatch = new CountDownLatch(6);
        // 六个同学离开
        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName()
+ "号同学离开了");
                // 每走一个同学--
                countDownLatch.countDown();
            }, i + "").start();
        }
        //等待，计数不为0会等待
        countDownLatch.await();
        System.out.println(Thread.currentThread().getName() + "班长锁
门走了");
    }
}
```

2. 循环栅栏 CyclicBarrier

1. 有参构造，第一个参数数量（实际上是等待的线程数量），第二个参数方法，在满足数量后会执行方法
2. `await` 没有满足数量，会一直等待，让线程等待（在逻辑上是表现为 `curNum++`，直到 `curNum == num`，才会执行方法）

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

// 演示CyclicBarrier
public class CyclicBarrierDemo {
    public static final int num = 7;

    public static void main(String[] args) {
        // 创建CyclicBarrier对象
        // 这里传入的num实际上是等待的线程数量
        CyclicBarrier cyclicBarrier = new CyclicBarrier(num, () -> {
            System.out.println("集齐七龙珠召唤神龙");
        });
        for (int i = 0; i < 7; i++) {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName()
+ " 星龙珠收集");
                try {
                    // 等待
                    cyclicBarrier.await();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                } catch (BrokenBarrierException e) {
                    throw new RuntimeException(e);
                }
            }, i + "").start();
        }
    }
}
```

3. 信号灯 Semaphore
    1. 有参构造， permits 许可量
    2. acquire()，抢占许可量，执行一次 permits--
    3. release 释放许可量， permits++

```
import java.util.Random;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

// 演示Semaphore
// 6辆汽车，3个停车位
public class SemaphoreDemo {
    public static void main(String[] args) {
        // 创建Semaphore，初始化许可量
        Semaphore semaphore = new Semaphore(3);
        for (int i = 1; i <= 6; i++) {
            new Thread(() -> {
                try {
                    // 抢占车位
                    semaphore.acquire();

 System.out.println(Thread.currentThread().getName() + "抢到了车位");
                    // 设置随机停车时间
                    TimeUnit.SECONDS.sleep(new Random().nextInt(5));

 System.out.println(Thread.currentThread().getName() + "----驶离了车
位");
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                } finally {
                    // 释放车位
                    semaphore.release();
                }
            }, i + "").start();
        }
```

```
        }
    }
```

ReentrantReadWriteLock 读写锁

- 悲观锁： 不支持并发，效率低
- 乐观锁：支持并发，根据版本号同步
- 表锁：操作数据库时，对整个表上锁
- 行锁：操作数据库时，对某一行上锁（会发生死锁）
- 读锁：共享锁，不会发生死锁，但是一个线程如果同时持有读锁和写锁，等待的进程持有了读锁，这可能会导致写进程无法获取写锁而发生死锁。
- 写锁：独占锁，会发生死锁

`volatile` 关键字：

1. 确保变量的内存可见性。在Java中，一个线程对变量的修改可能无法立即被其他线程看到，这是由于现代计算机的多级缓存结构和JVM的优化导致的。使用volatile关键字可以确保变量的值在多个线程之间具有内存可见性，即一个线程对变量的修改会立即被其他线程看到。这是因为使用volatile关键字会告诉JVM，该变量是共享变量，需要采取一些措施来确保其正确性，例如将变量的值写回主内存等。
2. 禁止指令重排序优化。在Java中，JVM可能会对指令进行重排序优化，以提高程序的性能。但是，如果一个变量没有被声明为volatile，JVM可能会将变量的读取操作和写入操作重排序，这可能会导致其他线程看到的变量值不正确。使用volatile关键字可以禁止JVM对变量的指令重排序优化，从而确保变量的值得到正确的更新和读取。

读锁、写锁演示 `Demo`

```
import java.util.HashMap;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class MyCache {
    // 资源
```

```
    private volatile HashMap<String, Object> map = new HashMap<>();
    // 创建读写锁
    private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

    // 放
    public void put(String key, Object value) {
        // 写锁
        readWriteLock.writeLock().lock();
        System.out.println(Thread.currentThread().getName() + "正在写操
作");
        try {
            TimeUnit.MICROSECONDS.sleep(300);
            map.put(key, value);
            System.out.println(Thread.currentThread().getName() + "写完
成");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally {
            // 写锁释放
            readWriteLock.writeLock().unlock();
        }
    }

    // 读
    public Object get(String key) {
        // 读锁
        readWriteLock.readLock().lock();
        Object res = null;
        try {
            System.out.println(Thread.currentThread().getName() + "正在读
取操作");
            TimeUnit.MICROSECONDS.sleep(300);
            res = map.get(key);
            System.out.println(Thread.currentThread().getName() + "读完
成");
        } catch (InterruptedException e) {
```

```
                throw new RuntimeException(e);
            } finally {
                // 读锁释放
                readWriteLock.readLock().unlock();
            }
            return res;
        }
    }

    public class ReadWriteDemo {
        public static void main(String[] args) {
            MyCache myCache = new MyCache();
            for (int i = 1; i <= 5; i++) {
                final int num = i;
                new Thread(() -> {
                    myCache.put(num + "", num);
                }, i + "").start();
            }
            for (int i = 1; i <= 5; i++) {
                final int num = i;
                new Thread(() -> {
                    myCache.get(num + "");
                }, i + "").start();
            }
        }
    }
```

- 读写锁：一个资源可以同时被多个读线程，或者被一个写线程访问，但是不能同时存在读写线程，**读写互斥，读读共享**。
- 锁降级：将写入锁降级为读锁

阻塞队列 BlockingQueue

1. 接口， put() ， take() 方法
2. 常见阻塞队列

1. `ArrayBlockingQueue()` 维护了一个定长数组，由数组结构维护的一个阻塞队列
2. `LinkedBlockingQueue` 由链表结构组成的有界（大小默认为 `Integer.MAX_VALUE` ）
3. `DelayQueue` 使用优先级队列实现的延迟无界阻塞队列
4. `PriorityBlockingQueue` 支持优先级排序无界阻塞队列
5. `SynchronousQueue` 不存储元素的阻塞队列，也即单个元素的队列
6. `LinkedTransferQueue` 由链表组成的无界阻塞队列
7. `LinkedBlockingDeque` 由链表结构组成的双向阻塞队列

3. 方法

方法类型	抛出异常	特殊值	阻塞	超时
插入	`add(e)`	`offer(e) return true/false`	`put(e)`	`offer(e,time,unit)`
移除	`remove(e)`	`poll() return e/null`	`take()`	`poll(time,unit)`
检查	`element(e)`	`peek()`	不可用	不可用

ThreadPool 线程池

1. 线程池使用方式
    1. 一池N线程 `Executors.newFixedThreadPool(int n)`
    2. 一池一线程 `Executors.newSingleThreadExecutor()`
    3. 线程池根据需求创建线程，可扩容，遇强则强 `Executors.newCachedThreadPool()`

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// 演示线程池三种常用分类
public class ThreadPoolDemo {
    public static void main(String[] args) {
        // 一池N（5）线程
```

```
//        ExecutorService threadPool1 =
Executors.newFixedThreadPool(5);

        // 一池一线程
//        ExecutorService threadPool2 =
Executors.newSingleThreadExecutor();

        // 一池可扩容
        ExecutorService threadPool3 =
Executors.newCachedThreadPool();
        // 10个顾客
        try {
            for (int i = 1; i < 11; i++) {
                // 执行
                threadPool3.execute(() -> {

 System.out.println(Thread.currentThread().getName() + "正在办理业
务");
                });
            }
        } finally {
            threadPool3.shutdown();
        }

    }
}
```

2. 线程池七大参数

```
new ThreadPoolExecutor(
  int corePoolSize,//常驻线程数量（核心线程数量）
  int maximumPoolSize,//最大支持线程数量
  long keepAliveTime,//线程存活时间值
  TimeUnit unit,//线程存活时间单位
  BlockingQueue<Runnable> workQueue,//阻塞队列实现
  ThreadFactory threadFactory,//线程工厂，创建线程
  RejectedExecutionHandler handler//拒绝策略
)
```

3. 线程池底层工作流程
    1. 主线程 -->执行 `execute()` 方法，线程创建
    2. 先使用 `corePool` 常驻线程
    3. 如果此时 `corePool` 线程满了，则会进入阻塞队列等待
    4. 当阻塞队列满足 `size==maximumPool-corePool` ，则创建新的线程
    5. 当最大线程数满了，就会执行拒绝策略
        - `AbortPolicy` 默认策略，直接抛出异常
        - `CallerRunsPolicy` 调用者运行，不会抛出任务或异常，将任务回退给调用者
        - `DiscardOldestPolicy` 抛弃队列中等待最久的任务，将当前任务加入队列中
        - `DiscardPolicy` 不作任何处理
4. 自定义线程池（实际开发使用， `Executors` 返回的线程池对象，允许队列长度和创建线程数量均为 `Integer.MAX_VALUE` ，可能会堆积大量请求/创建大量线程，造成栈溢出）

```
import java.util.concurrent.*;

public class ThreadPoolDemo1 {
    public static void main(String[] args) {
        ExecutorService threadPool = new ThreadPoolExecutor(
                2,
                5,
                2,
                TimeUnit.SECONDS,
```

```
                    new ArrayBlockingQueue<>(3),
                    Executors.defaultThreadFactory(),
                    new ThreadPoolExecutor.AbortPolicy());
            try {
                for (int i = 1; i < 11; i++) {
                    // 执行
                    threadPool.execute(() -> {

     System.out.println(Thread.currentThread().getName() + "正在办理业
    务");
                    });
                }
            } finally {
                threadPool.shutdown();
            }
        }
```

Fork/Join

`Fork`：将复杂任务拆分

`Join`：把拆分任务结果进行合并

```
    import java.util.concurrent.ExecutionException;
    import java.util.concurrent.ForkJoinPool;
    import java.util.concurrent.ForkJoinTask;
    import java.util.concurrent.RecursiveTask;

    class MyTask extends RecursiveTask<Integer> {
        // 拆分差值不超过10
        private static final Integer VALUE = 10;
        private int begin;
        private int end;
        private int res;
```

```
    public MyTask(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }

    // 拆分与合并
    @Override
    protected Integer compute() {
        // 判断差值
        if (end - begin <= 10) {
            for (int i = begin; i <= end; i++) {
                res += i;
            }
        } else {// 进一步拆分
            int mid = begin + (end - begin) / 2;
            // 左区间
            MyTask myTask01 = new MyTask(begin, mid);
            // 右区间
            MyTask myTask02 = new MyTask(mid + 1, end);
            // 拆分
            myTask01.fork();
            myTask02.fork();
            // 合并结果
            res = myTask01.join() + myTask02.join();
        }
        return res;
    }
}

public class ForkJoinDemo {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // 创建任务
        MyTask myTask = new MyTask(0, 100);
        // 创建分支合并池
```

```
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        ForkJoinTask<Integer> forkJoinTask =
forkJoinPool.submit(myTask);
        // 获取最终合并结果
        Integer res = forkJoinTask.get();
        System.out.println(res);
        // 关闭池对象
        forkJoinPool.shutdown();
    }
}
```

CompletableFuture异步回调

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureDemo {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // 异步调用无返回值
        CompletableFuture<Void> completableFuture1 =
CompletableFuture.runAsync(() -> {
            System.out.println(Thread.currentThread().getName() + "---
completableFuture1");
        });
        completableFuture1.get();
        // 异步调用有返回值
        CompletableFuture<Integer> completableFuture2 =
CompletableFuture.supplyAsync(() -> {
            System.out.println(Thread.currentThread().getName() + "---
completableFuture2");
            return 1024;
        });
        completableFuture2.whenComplete((t, u) -> {
```

```
            // 方法返回值
            System.out.println("---t:" + t);
            // 异常信息
            System.out.println("---u:" + u);
        });
        completableFuture2.get();
    }
}
```