JUC 工具类

CountDownLatch：减法计数器

可以用来倒计时，当两个线程同时执行时，如果要确保一个线程优先执行，可以使用计数器，当计数器清零的时候，再让另一个线程执行。

```
package com.southwind.demo;

import java.util.concurrent.CountDownLatch;

public class Test {
    public static void main(String[] args) {
        //创建一个 CountDownLatch
        CountDownLatch countDownLatch = new CountDownLatch(100);

        new Thread(()->{
            for (int i = 0; i < 100; i++) {
                System.out.println("+++++++++++++++Thread");
                countDownLatch.countDown();
            }
        }).start();

        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        for (int i = 0; i < 100; i++) {
            System.out.println("main-----------------");
        }
    }
}
```

coutDown()：计数器减一

await()：计数器停止，唤醒其他线程

new CountDownLatch(100)、coutDown()、await() 必须配合起来使用，创建对象的时候赋的值是多少，coutDown() 就必须执行多少次，否则计数器是没有清零的，计数器就不会停止，其他线程也无法唤醒，所以必须保证计数器清零，coutDown() 的调用次数必须大于构造函数的参数值。

CyclicBarrier：加法计数器

```
public class CyclicBarrierTest {
    public static void main(String[] args) {
```

```
        CyclicBarrier cyclicBarrier = new CyclicBarrier(100,()->{
            System.out.println("放行");
        });

        for (int i = 0; i < 100; i++) {
            final int temp = i;
            new Thread(()->{
                System.out.println("-->"+temp);
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }).start();
        }
    }
}
```

await()：在其他线程中试图唤醒计数器线程，当其他线程的执行次数达到计数器的临界值时，则唤醒计数器线程，并且计数器是可以重复使用的，当计数器的线程执行完成一次之后，计数器自动清零，等待下一次执行。

new CyclicBarrier(30），for 执行 90 次，则计数器的任务会执行 3 次。

Semaphore：计数信号量

实际开发中主要使用它来完成限流操作，限制可以访问某些资源的线程数量。

Semaphore 只有 3 个操作：

- 初始化
- 获取许可
- 释放

```
public class SemaphoreTest {
    public static void main(String[] args) {
        //初始化
        Semaphore semaphore = new Semaphore(5);
        for (int i = 0; i < 15; i++) {
            new Thread(()->{
                //获得许可
                try {
                    semaphore.acquire();
                    System.out.println(Thread.currentThread().getName()+"进店购
物");
                    TimeUnit.SECONDS.sleep(5);
                    System.out.println(Thread.currentThread().getName()+"出
店");
                } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }finally {
                    //释放
                    semaphore.release();
                }

            },String.valueOf(i)).start();
        }
    }
}
```

每个线程在执行的时候，首先需要去获取信号量，只有获取到资源才可以执行，执行完毕之后需要释放资源，留给下一个线程。

读写锁

接口 ReadWriteLock，实现类是 ReentrantReadWriteLock，可以多线程同时读，但是同一时间内只能有一个线程进行写入操作。

读写锁也是为了实现线程同步，只不过粒度更细，可以分别给读和写的操作设置不同的锁。

```
public class ReadWriteLockTest {
    public static void main(String[] args) {
        Cache cache = new Cache();
        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread(()->{

                cache.write(temp,String.valueOf(temp));

            }).start();
        }

        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread(()->{
                cache.read(temp);
            }).start();
        }
    }
}

class Cache{
    private Map<Integer,String> map = new HashMap<>();
    private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

    /**
     * 写操作
     */
```

```
    public void write(Integer key,String value){
        readWriteLock.writeLock().lock();
        System.out.println(key+"开始写入");
        map.put(key,value);
        System.out.println(key+"写入完毕");
        readWriteLock.writeLock().unlock();
    }

    /**
     * 读操作
     */
    public void read(Integer key){
        readWriteLock.readLock().lock();
        System.out.println(key+"开始读取");
        map.get(key);
        System.out.println(key+"读取完毕");
        readWriteLock.readLock().unlock();
    }
}
```

写入锁也叫独占锁，只能被一个线程占用，读取锁也叫共享锁，多个线程可以同时占用。

线程池

预先创建好一定数量的线程对象，存入缓冲池中，需要用的时候直接从缓冲池中取出，用完之后不要销毁，还回到缓冲池中，为了提高资源的利用率。

优势：

- 提高线程的利用率
- 提高响应速度
- 便于统一管理线程对象
- 可以控制最大的并发数

1、线程池初始化的时候创建一定数量的线程对象。

2、如果缓冲池中没有空闲的线程对象，则新来的任务进入等待队列。

3、如果缓冲池中没有空闲的线程对象，等待队列也已经填满，可以申请再创建一定数量的新线程对象，直到到达线程池的最大值，这时候如果还有新的任务进来，只能选择拒绝。

无论哪种线程池，都是工具类 Executors 封装的，底层代码都一样，都是通过创建 ThreadPoolExecutor 对象来完成线程池的构建。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
```

```
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = (System.getSecurityManager() == null)
        ? null
        : AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

- corePoolSize：核心池大小，初始化的线程数量
- maximumPoolSize：线程池最大线程数，它决定了线程池容量的上限

corePoolSize 就是线程池的大小，maximumPoolSize 是一种补救措施，任务量突然增大的时候的一种补救措施。

- keepAliveTime：线程对象的存活时间
- unit：线程对象存活时间单位
- workQueue：等待队列
- threadFactory：线程工厂，用来创建线程对象
- handler：拒绝策略