线程池【线程治理】

1、每次新开一个任务就会创新建一个线程代码演示
2、for循环创建线程演示
3、newFixedThreadPool代码演示
4、newFixedThreadPool导致OOM的情况演示
5、newSingleThreadExecutor代码演示
6、newCachedThreadPool代码演示
7、newScheduledThreadPool代码演示
8、每个任务执行前后放钩子函数代码演示
9、关闭线程池代码演示

ThreadLocal详解

1、1000个打印日期的任务，用线程池来执行（存在线程安全问题）
2、1000个打印日期的任务，用线程池来执行（加锁来解决线程安全问题）
3、1000个打印日期的任务，用线程池来执行(利用ThreadLocal，给每个线程分配自己的dateFormat对象，保证了线程安全，高效利用内存)
4、演示ThreadLocal用法2：避免传递参数的麻烦
5、ThreadLocal空指针异常问题演示

锁

1、获取锁时被中断代码演示
2、可重入锁和非可重入锁，以ReentrantLock为例代码演示
3、公平锁和非公平锁，以ReentrantLock为例代码演示
4、读写锁，以ReentrantReadWriteLock为例代码演示（电影院买票升级）
5、演示非公平和公平的ReentrantReadWriteLock的策略
6、演示ReentrantReadWriteLock可以降级，不能升级

原子类

1、演示AtomicInteger的基本用法，对比非原子类的线程安全问题，使用了原子类之后，不需要加锁，也可以保证线程安全。
2、演示原始数组的使用方法
3、演示高并发场景下，LongAdder比AtomicLong性能好
4、演示LongAccumulator的用法
5、演示AtomicIntegerFieldUpdater的用法

CAS

1、模拟CAS操作，等价代码

Final关键字和不可变性

并发容器

1、演示组合操作并不保证线程安全，以ConcurrentHashMap为例
2、演示CopyOnWriteArrayList可以在迭代的过程中修改数组内容，但是ArrayList不行，对比

控制并发流程工具类

CountDownLatch门闩

1、模拟100米跑步，5名选手都准备好了，只等裁判员一声令下，所有人同时开始跑步；当所有人都到终点后，比赛结束。

Semaphore信号量

1、演示Semaphore用法

Condition接口

1、演示Condition的基本用法
2、演示用Condition实现生产者消费者模式

AQS

1、自己用AQS实现一个简单的线程协作器

获取子线程执行结果

1、演示FutureTask的用法
2、 演示批量提交任务时，用List来批量接收结果
3、演示get方法过程中抛出异常，for循环为了演示抛出Exception的时机：并不是说一产生异常就抛出，直到我们get执行时，才会抛出。
4、 演示get的超时方法，需要注意超时后处理，调用future.cancel()。演示cancel传入true和false的区别，代表是否中断正在执行的任务。

高性能缓存
1、最简单的缓存形式：HashMap
2、 用装饰者模式，给计算器自动添加缓存功能
3、缩小synchronized加锁粒度，但性能差，存在线程安全问题
4、使用ConcurrentHashMap解决线程安全问题
5、 利用Future，避免重复计算
6、 利用Future，避免重复计算（使用putIfAbsent优化）
7、利用Future，避免重复计算（考虑计算抛出异常情况）
8、出于安全性考虑，缓存需要设置有效期，到期自动失效，否则如果缓存一直不失效，那么带来缓存不一致等问题

线程池【线程治理】

1、每次新开一个任务就会创新建一个线程代码演示

```
public class EveryTaskOneThread {

    public static void main(String[] args) {
        Thread thread = new Thread(new Task());
        thread.start();
    }

    static class Task implements Runnable {

        @Override
        public void run() {
            System.out.println("执行了任务");
        }
    }
}
```

2、for循环创建线程演示

```
public class ForLoop {

    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {
            Thread thread = new Thread(new Task());
            thread.start();
        }
    }

    static class Task implements Runnable {

        @Override
        public void run() {
            System.out.println("执行了任务");
        }
    }
}
```

3、newFixedThreadPool代码演示

```
public class FixedThreadPoolTest {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        for (int i = 0; i < 1000; i++) {
            executorService.execute(new Task());
        }
    }
}

class Task implements Runnable {

    @Override
    public void run() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName());
    }
}
```

4、newFixedThreadPool导致OOM的情况演示

```
public class FixedThreadPoolOOM {

    private static ExecutorService executorService =
Executors.newFixedThreadPool(1);
    public static void main(String[] args) {
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            executorService.execute(new SubThread());
        }
    }
}

class SubThread implements Runnable {

    @Override
    public void run() {
        try {
            Thread.sleep(1000000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

5、newSingleThreadExecutor代码演示

```
public class SingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 1000; i++) {
            executorService.execute(new Task());
        }
    }
}
```

6、newCachedThreadPool代码演示

```
public class CachedThreadPool {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 1000; i++) {
            executorService.execute(new Task());
        }
    }
}
```

7、newScheduledThreadPool代码演示

```
public class ScheduledThreadPoolTest {

    public static void main(String[] args) {
        ScheduledExecutorService threadPool =
Executors.newScheduledThreadPool(10);
//        threadPool.schedule(new Task(), 5, TimeUnit.SECONDS);
        threadPool.scheduleAtFixedRate(new Task(), 1, 3, TimeUnit.SECONDS);
    }
}
```

8、每个任务执行前后放钩子函数代码演示

```
public class PauseableThreadPool extends ThreadPoolExecutor {

    private final ReentrantLock lock = new ReentrantLock();
    private Condition unpaused = lock.newCondition();
    private boolean isPaused;


    public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
            TimeUnit unit,
            BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }

    public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
            TimeUnit unit, BlockingQueue<Runnable> workQueue,
```

```
            ThreadFactory threadFactory) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
threadFactory);
    }

    public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
            TimeUnit unit, BlockingQueue<Runnable> workQueue,
            RejectedExecutionHandler handler) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
handler);
    }

    public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
            TimeUnit unit, BlockingQueue<Runnable> workQueue,
            ThreadFactory threadFactory, RejectedExecutionHandler handler) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
threadFactory,
                handler);
    }

    @Override
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        lock.lock();
        try {
            while (isPaused) {
                unpaused.await();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    private void pause() {
        lock.lock();
        try {
            isPaused = true;
        } finally {
            lock.unlock();
        }
    }

    public void resume() {
        lock.lock();
        try {
            isPaused = false;
            unpaused.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
```

```
        PauseableThreadPool pauseableThreadPool = new PauseableThreadPool(10,
20, 10l,
                TimeUnit.SECONDS, new LinkedBlockingQueue<>());
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("我被执行");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        for (int i = 0; i < 10000; i++) {
            pauseableThreadPool.execute(runnable);
        }
        Thread.sleep(1500);
        pauseableThreadPool.pause();
        System.out.println("线程池被暂停了");
        Thread.sleep(1500);
        pauseableThreadPool.resume();
        System.out.println("线程池被恢复了");
    }
}
```

9、关闭线程池代码演示

```
public class ShutDown {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        for (int i = 0; i < 100; i++) {
            executorService.execute(new ShutDownTask());
        }
        Thread.sleep(1500);
//        List<Runnable> runnableList = executorService.shutdownNow();

        executorService.shutdown();
        executorService.execute(new ShutDownTask());
//        boolean b = executorService.awaitTermination(7L, TimeUnit.SECONDS);
//        System.out.println(b);
//        System.out.println(executorService.isShutdown());
//        executorService.shutdown();
//        System.out.println(executorService.isShutdown());
//        System.out.println(executorService.isTerminated());
//        Thread.sleep(10000);
//        System.out.println(executorService.isTerminated());

//        executorService.execute(new ShutDownTask());
    }
}

class ShutDownTask implements Runnable {

    @Override
    public void run() {
```

```
        try {
            Thread.sleep(500);
            System.out.println(Thread.currentThread().getName());
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + "被中断了");
        }
    }
}
```

ThreadLocal详解

1、1000个打印日期的任务，用线程池来执行（存在线程安全问题）

```
public class ThreadLocalNormalUsage03 {

    public static ExecutorService threadPool = Executors.newFixedThreadPool(10);
    static SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            int finalI = i;
            threadPool.submit(new Runnable() {
                @Override
                public void run() {
                    String date = new ThreadLocalNormalUsage03().date(finalI);
                    System.out.println(date);
                }
            });
        }
        threadPool.shutdown();
    }

    public String date(int seconds) {
        //参数的单位是毫秒，从1970.1.1 00:00:00 GMT计时
        Date date = new Date(1000 * seconds);
        return dateFormat.format(date);
    }
}
```

2、1000个打印日期的任务，用线程池来执行（加锁来解决线程安全问题）

```
public class ThreadLocalNormalUsage04 {

    public static ExecutorService threadPool = Executors.newFixedThreadPool(10);
    static SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            int finalI = i;
            threadPool.submit(new Runnable() {
                @Override
                public void run() {
```

```
                    String date = new ThreadLocalNormalUsage04().date(finalI);
                    System.out.println(date);
                }
            });
        }
        threadPool.shutdown();
    }

    public String date(int seconds) {
        //参数的单位是毫秒，从1970.1.1 00:00:00 GMT计时
        Date date = new Date(1000 * seconds);
        String s = null;
        synchronized (ThreadLocalNormalUsage04.class) {
            s = dateFormat.format(date);
        }
        return s;
    }
}
```

3、1000个打印日期的任务，用线程池来执行(利用ThreadLocal，给每个线程分配自己的dateFormat对象，保证了线程安全，高效利用内存)

```
public class ThreadLocalNormalUsage05 {

    public static ExecutorService threadPool = Executors.newFixedThreadPool(10);

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            int finalI = i;
            threadPool.submit(new Runnable() {
                @Override
                public void run() {
                    String date = new ThreadLocalNormalUsage05().date(finalI);
                    System.out.println(date);
                }
            });
        }
        threadPool.shutdown();
    }

    public String date(int seconds) {
        //参数的单位是毫秒，从1970.1.1 00:00:00 GMT计时
        Date date = new Date(1000 * seconds);
//        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        SimpleDateFormat dateFormat =
ThreadSafeFormatter.dateFormatThreadLocal2.get();
        return dateFormat.format(date);
    }
}

class ThreadSafeFormatter {

    public static ThreadLocal<SimpleDateFormat> dateFormatThreadLocal = new
ThreadLocal<SimpleDateFormat>() {
```

```
        @Override
        protected SimpleDateFormat initialValue() {
            return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        }
    };

    public static ThreadLocal<SimpleDateFormat> dateFormatThreadLocal2 =
ThreadLocal
            .withInitial(() -> new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
}
```

4、演示ThreadLocal用法2：避免传递参数的麻烦

```
public class ThreadLocalNormalUsage06 {

    public static void main(String[] args) {
        new Service1().process("");

    }
}

class Service1 {

    public void process(String name) {
        User user = new User("超哥");
        UserContextHolder.holder.set(user);
        new Service2().process();
    }
}

class Service2 {

    public void process() {
        User user = UserContextHolder.holder.get();
        ThreadSafeFormatter.dateFormatThreadLocal.get();
        System.out.println("Service2拿到用户名：" + user.name);
        new Service3().process();
    }
}

class Service3 {

    public void process() {
        User user = UserContextHolder.holder.get();
        System.out.println("Service3拿到用户名：" + user.name);
        UserContextHolder.holder.remove();
    }
}

class UserContextHolder {

    public static ThreadLocal<User> holder = new ThreadLocal<>();


}

class User {
```

```
    String name;

    public User(String name) {
        this.name = name;
    }
}
```

5、ThreadLocal空指针异常问题演示

```
public class ThreadLocalNPE {

    ThreadLocal<Long> longThreadLocal = new ThreadLocal<Long>();

    public void set() {
        longThreadLocal.set(Thread.currentThread().getId());
    }

    public long get() {
        return longThreadLocal.get();
    }

    public static void main(String[] args) {
        ThreadLocalNPE threadLocalNPE = new ThreadLocalNPE();
        System.out.println(threadLocalNPE.get());
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                threadLocalNPE.set();
                System.out.println(threadLocalNPE.get());
            }
        });
        thread1.start();
    }
}
```

```
public class ThreadLocalNPE {

    ThreadLocal<Long> longThreadLocal = new ThreadLocal<Long>();

    public void set() {
        longThreadLocal.set(Thread.currentThread().getId());
    }

    public long get() {
        return longThreadLocal.get();
    }

    public static void main(String[] args) {
        ThreadLocalNPE threadLocalNPE = new ThreadLocalNPE();
        System.out.println(threadLocalNPE.get());
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                threadLocalNPE.set();
                System.out.println(threadLocalNPE.get());
```

```
            }
        });
        thread1.start();
    }
}
```

```
public class ThreadLocalNPE {

    ThreadLocal<Long> longThreadLocal = new ThreadLocal<Long>();

    public void set() {
        longThreadLocal.set(Thread.currentThread().getId());
    }

    public long get() {
        return longThreadLocal.get();
    }

    public static void main(String[] args) {
        ThreadLocalNPE threadLocalNPE = new ThreadLocalNPE();
        System.out.println(threadLocalNPE.get());
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                threadLocalNPE.set();
                System.out.println(threadLocalNPE.get());
            }
        });
        thread1.start();
    }
}
```

锁

1、获取锁时被中断代码演示

```
public class LockInterruptibly implements Runnable {

    private Lock lock = new ReentrantLock();
public static void main(String[] args) {
    LockInterruptibly lockInterruptibly = new LockInterruptibly();
    Thread thread0 = new Thread(lockInterruptibly);
    Thread thread1 = new Thread(lockInterruptibly);
    thread0.start();
    thread1.start();

//    try {
//        Thread.sleep(2000);
//    } catch (InterruptedException e) {
//        e.printStackTrace();
//    }
    thread1.interrupt();
}
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "尝试获取锁");
```

```
        try {
            lock.lockInterruptibly();
            try {
                System.out.println(Thread.currentThread().getName() + "获取到了
锁");
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + "睡眠期间被中
断了");
            } finally {
                lock.unlock();
                System.out.println(Thread.currentThread().getName() + "释放了锁");
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + "获得锁期间被中断
了");
        }
    }
}
```

2、可重入锁和非可重入锁，以ReentrantLock为例代码演示

```
public class RecursionDemo {

    private static ReentrantLock lock = new ReentrantLock();

    private static void accessResource() {
        lock.lock();
        try {
            System.out.println("已经对资源进行了处理");
            if (lock.getHoldCount()<5) {
                System.out.println(lock.getHoldCount());
                accessResource();
                System.out.println(lock.getHoldCount());
            }
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        accessResource();
    }
}
```

3、公平锁和非公平锁，以ReentrantLock为例代码演示

```
public class FairLock {

    public static void main(String[] args) {
        PrintQueue printQueue = new PrintQueue();
        Thread thread[] = new Thread[10];
        for (int i = 0; i < 10; i++) {
            thread[i] = new Thread(new Job(printQueue));
        }
        for (int i = 0; i < 10; i++) {
            thread[i].start();
```

```
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class Job implements Runnable {

    PrintQueue printQueue;

    public Job(PrintQueue printQueue) {
        this.printQueue = printQueue;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "开始打印");
        printQueue.printJob(new Object());
        System.out.println(Thread.currentThread().getName() + "打印完毕");
    }
}

class PrintQueue {

    private Lock queueLock = new ReentrantLock(true);

    public void printJob(Object document) {
        queueLock.lock();
        try {
            int duration = new Random().nextInt(10) + 1;
            System.out.println(Thread.currentThread().getName() + "正在打印，需要"
+ duration);
            Thread.sleep(duration * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            queueLock.unlock();
        }

        queueLock.lock();
        try {
            int duration = new Random().nextInt(10) + 1;
            System.out.println(Thread.currentThread().getName() + "正在打印，需要"
+ duration+"秒");
            Thread.sleep(duration * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            queueLock.unlock();
        }
    }
}
```

4、读写锁，以ReentrantReadWriteLock为例代码演示（电影院买票升级）

```
public class CinemaReadWrite {

    private static ReentrantReadWriteLock reentrantReadWriteLock = new
ReentrantReadWriteLock();
    private static ReentrantReadWriteLock.ReadLock readLock =
reentrantReadWriteLock.readLock();
    private static ReentrantReadWriteLock.WriteLock writeLock =
reentrantReadWriteLock.writeLock();

    private static void read() {
        readLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到了读锁，正在
读取");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放读锁");
            readLock.unlock();
        }
    }

    private static void write() {
        writeLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到了写锁，正在
写入");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放写锁");
            writeLock.unlock();
        }
    }

    public static void main(String[] args) {
        new Thread(()->read(),"Thread1").start();
        new Thread(()->read(),"Thread2").start();
        new Thread(()->write(),"Thread3").start();
        new Thread(()->write(),"Thread4").start();
         new Thread(()->write(),"Thread1").start();
//        new Thread(()->read(),"Thread2").start();
//        new Thread(()->read(),"Thread3").start();
//        new Thread(()->write(),"Thread4").start();
//        new Thread(()->read(),"Thread5").start();
    }
}
```

5、演示非公平和公平的ReentrantReadWriteLock的策略

```
public class NonfairBargeDemo {

    private static ReentrantReadWriteLock reentrantReadWriteLock = new
ReentrantReadWriteLock(
            true);

    private static ReentrantReadWriteLock.ReadLock readLock =
reentrantReadWriteLock.readLock();
    private static ReentrantReadWriteLock.WriteLock writeLock =
reentrantReadWriteLock.writeLock();

    private static void read() {
        System.out.println(Thread.currentThread().getName() + "开始尝试获取读锁");
        readLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到读锁，正在读
取");
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放读锁");
            readLock.unlock();
        }
    }

    private static void write() {
        System.out.println(Thread.currentThread().getName() + "开始尝试获取写锁");
        writeLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到写锁，正在写
入");
            try {
                Thread.sleep(40);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放写锁");
            writeLock.unlock();
        }
    }

    public static void main(String[] args) {
        new Thread(()->write(),"Thread1").start();
        new Thread(()->read(),"Thread2").start();
        new Thread(()->read(),"Thread3").start();
        new Thread(()->write(),"Thread4").start();
        new Thread(()->read(),"Thread5").start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                Thread thread[] = new Thread[1000];
```

```
                for (int i = 0; i < 1000; i++) {
                    thread[i] = new Thread(() -> read(), "子线程创建的Thread" + i);
                }
                for (int i = 0; i < 1000; i++) {
                    thread[i].start();
                }
            }
        }).start();
    }
}
```

6、演示ReentrantReadWriteLock可以降级，不能升级

```
public class Upgrading {

    private static ReentrantReadWriteLock reentrantReadWriteLock = new
ReentrantReadWriteLock(
            false);
    private static ReentrantReadWriteLock.ReadLock readLock =
reentrantReadWriteLock.readLock();
    private static ReentrantReadWriteLock.WriteLock writeLock =
reentrantReadWriteLock.writeLock();

    private static void readUpgrading() {
        readLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到了读锁，正在
读取");
            Thread.sleep(1000);
            System.out.println("升级会带来阻塞");
            writeLock.lock();
            System.out.println(Thread.currentThread().getName() + "获取到了写锁，升
级成功");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放读锁");
            readLock.unlock();
        }
    }

    private static void writeDowngrading() {
        writeLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到了写锁，正在
写入");
            Thread.sleep(1000);
            readLock.lock();
            System.out.println("在不释放写锁的情况下，直接获取读锁，成功降级");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            readLock.unlock();
            System.out.println(Thread.currentThread().getName() + "释放写锁");
            writeLock.unlock();
        }
    }
```

```
    public static void main(String[] args) throws InterruptedException {
//        System.out.println("先演示降级是可以的");
//        Thread thread1 = new Thread(() -> writeDowngrading(), "Thread1");
//        thread1.start();
//        thread1.join();
//        System.out.println("-------------------");
//        System.out.println("演示升级是不行的");
        Thread thread2 = new Thread(() -> readUpgrading(), "Thread2");
        thread2.start();
    }
}
```

原子类

1、演示AtomicInteger的基本用法，对比非原子类的线程安全问题，使用了原子类之后，不需要加锁，也可以保证线程安全。

```
public class AtomicIntegerDemo1 implements Runnable {

    private static final AtomicInteger atomicInteger = new AtomicInteger();

    public void incrementAtomic() {
        atomicInteger.getAndAdd(-90);
    }

    private static volatile int basicCount = 0;

    public synchronized void incrementBasic() {
        basicCount++;
    }

    public static void main(String[] args) throws InterruptedException {
        AtomicIntegerDemo1 r = new AtomicIntegerDemo1();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("原子类的结果：" + atomicInteger.get());
        System.out.println("普通变量的结果：" + basicCount);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            incrementAtomic();
            incrementBasic();
        }
    }
}
```

2、演示原始数组的使用方法

```
public class AtomicArrayDemo {

    public static void main(String[] args) {
        AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(1000);
        Incrementer incrementer = new Incrementer(atomicIntegerArray);
        Decrementer decrementer = new Decrementer(atomicIntegerArray);
        Thread[] threadsIncrementer = new Thread[100];
        Thread[] threadsDecrementer = new Thread[100];
        for (int i = 0; i < 100; i++) {
            threadsDecrementer[i] = new Thread(decrementer);
            threadsIncrementer[i] = new Thread(incrementer);
            threadsDecrementer[i].start();
            threadsIncrementer[i].start();
        }

//        Thread.sleep(10000);
        for (int i = 0; i < 100; i++) {
            try {
                threadsDecrementer[i].join();
                threadsIncrementer[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        for (int i = 0; i < atomicIntegerArray.length(); i++) {
//            if (atomicIntegerArray.get(i)!=0) {
//                System.out.println("发现了错误"+i);
//            }
            System.out.println(atomicIntegerArray.get(i));
        }
        System.out.println("运行结束");
    }
}

class Decrementer implements Runnable {

    private AtomicIntegerArray array;

    public Decrementer(AtomicIntegerArray array) {
        this.array = array;
    }

    @Override
    public void run() {
        for (int i = 0; i < array.length(); i++) {
            array.getAndDecrement(i);
        }
    }
}

class Incrementer implements Runnable {

    private AtomicIntegerArray array;
```

```
    public Incrementer(AtomicIntegerArray array) {
        this.array = array;
    }

    @Override
    public void run() {
        for (int i = 0; i < array.length(); i++) {
            array.getAndIncrement(i);
        }
    }
}
```

3、演示高并发场景下，LongAdder比AtomicLong性能好

```
public class LongAdderDemo {

    public static void main(String[] args) throws InterruptedException {
        LongAdder counter = new LongAdder();
        ExecutorService service = Executors.newFixedThreadPool(20);
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            service.submit(new Task(counter));
        }
        service.shutdown();
        while (!service.isTerminated()) {

        }
        long end = System.currentTimeMillis();
        System.out.println(counter.sum());
        System.out.println("LongAdder耗时: " + (end - start));
    }

    private static class Task implements Runnable {

        private LongAdder counter;

        public Task(LongAdder counter) {
            this.counter = counter;
        }

        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                counter.increment();
            }
        }
    }
}
```

```
public class AtomicLongDemo {

    public static void main(String[] args) throws InterruptedException {
        AtomicLong counter = new AtomicLong(0);
        ExecutorService service = Executors.newFixedThreadPool(20);
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
```

```
            service.submit(new Task(counter));
        }
        service.shutdown();
        while (!service.isTerminated()) {

        }
        long end = System.currentTimeMillis();
        System.out.println(counter.get());
        System.out.println("AtomicLong耗时：" + (end - start));
    }

    private static class Task implements Runnable {

        private AtomicLong counter;

        public Task(AtomicLong counter) {
            this.counter = counter;
        }

        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                counter.incrementAndGet();
            }
        }
    }
}
```

4、演示LongAccumulator的用法

```
public class LongAccumulatorDemo {

    public static void main(String[] args) {
        LongAccumulator accumulator = new LongAccumulator((x, y) -> 2 + x * y,
1);
        ExecutorService executor = Executors.newFixedThreadPool(8);
        IntStream.range(1, 10).forEach(i -> executor.submit(() ->
accumulator.accumulate(i)));

        executor.shutdown();
        while (!executor.isTerminated()) {

        }
        System.out.println(accumulator.getThenReset());
    }
}
```

5、演示AtomicIntegerFieldUpdater的用法

```
public class AtomicIntegerFieldUpdaterDemo implements Runnable{

    static Candidate tom;
    static Candidate peter;

    public static AtomicIntegerFieldUpdater<Candidate> scoreUpdater =
AtomicIntegerFieldUpdater
```

```
            .newUpdater(Candidate.class, "score");

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            peter.score++;
            scoreUpdater.getAndIncrement(tom);
        }
    }

    public static class Candidate {

        volatile int score;
    }

    public static void main(String[] args) throws InterruptedException {
        tom=new Candidate();
        peter=new Candidate();
        AtomicIntegerFieldUpdaterDemo r = new AtomicIntegerFieldUpdaterDemo();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("普通变量: "+peter.score);
        System.out.println("升级后的结果"+ tom.score);
    }
}
```

CAS

1、模拟CAS操作，等价代码

```
ublic class TwoThreadsCompetition implements Runnable {

    private volatile int value;

    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue) {
            value = newValue;
        }
        return oldValue;
    }

    public static void main(String[] args) throws InterruptedException {
        TwoThreadsCompetition r = new TwoThreadsCompetition();
        r.value = 0;
        Thread t1 = new Thread(r,"Thread 1");
        Thread t2 = new Thread(r,"Thread 2");
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(r.value);
```

```
    }

    @Override
    public void run() {
        compareAndSwap(0, 1);
    }
}
```

Final关键字和不可变性

###

并发容器

1、演示组合操作并不保证线程安全，以ConcurrentHashMap为例

```
public class OptionsNotSafe implements Runnable {

    private static ConcurrentHashMap<String, Integer> scores = new
ConcurrentHashMap<String, Integer>();

    public static void main(String[] args) throws InterruptedException {
        scores.put("小明", 0);
        Thread t1 = new Thread(new OptionsNotSafe());
        Thread t2 = new Thread(new OptionsNotSafe());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(scores);
    }


    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            while (true) {
                Integer score = scores.get("小明");
                Integer newScore = score + 1;
                boolean b = scores.replace("小明", score, newScore);
                if (b) {
                    break;
                }
            }
        }

    }
}
```

2、演示CopyOnWriteArrayList可以在迭代的过程中修改数组内容，但是ArrayList不行，对比

```
public class CopyOnWriteArrayListDemo1 {

    public static void main(String[] args) {
//        ArrayList<String> list = new ArrayList<>();
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();

        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");

        Iterator<String> iterator = list.iterator();

        while (iterator.hasNext()) {
            System.out.println("list is" + list);
            String next = iterator.next();
            System.out.println(next);

            if (next.equals("2")) {
                list.remove("5");
            }
            if (next.equals("3")) {
                list.add("3 found");
            }
        }
    }
}
```

控制并发流程工具类

CountDownLatch门闩

1、模拟100米跑步，5名选手都准备好了，只等裁判员一声令下，所有人同时开始跑步；当所有人都到终点后，比赛结束。

```
public class CountDownLatchDemo1And2 {

    public static void main(String[] args) throws InterruptedException {
        CountDownLatch begin = new CountDownLatch(1);

        CountDownLatch end = new CountDownLatch(5);
        ExecutorService service = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 5; i++) {
            final int no = i + 1;
            Runnable runnable = new Runnable() {
                @Override
                public void run() {
                    System.out.println("No." + no + "准备完毕，等待发令枪");
                    try {
                        begin.await();
                        System.out.println("No." + no + "开始跑步了");
```

```
                    Thread.sleep((long) (Math.random() * 10000));
                    System.out.println("No." + no + "跑到终点了");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    end.countDown();
                }
            }
        };
        service.submit(runnable);
    }
    //裁判员检查发令枪...
    Thread.sleep(5000);
    System.out.println("发令枪响，比赛开始！");
    begin.countDown();

    end.await();
    System.out.println("所有人到达终点，比赛结束");
  }
}
```

Semaphore信号量

1、演示Semaphore用法

```
public class SemaphoreDemo {

    static Semaphore semaphore = new Semaphore(5, true);

    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(50);
        for (int i = 0; i < 100; i++) {
            service.submit(new Task());
        }
        service.shutdown();
    }

    static class Task implements Runnable {

        @Override
        public void run() {
            try {
                semaphore.acquire(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "拿到了许可证");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "释放了许可证");
            semaphore.release(2);
        }
    }
}
```

Condition接口

1、演示Condition的基本用法

```
public class ConditionDemo1 {
    private ReentrantLock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    void method1() throws InterruptedException {
        lock.lock();
        try{
            System.out.println("条件不满足，开始await");
            condition.await();
            System.out.println("条件满足了，开始执行后续的任务");
        }finally {
            lock.unlock();
        }
    }

    void method2() {
        lock.lock();
        try{
            System.out.println("准备工作完成，唤醒其他的线程");
            condition.signal();
        }finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ConditionDemo1 conditionDemo1 = new ConditionDemo1();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                    conditionDemo1.method2();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        conditionDemo1.method1();
    }
}
```

2、演示用Condition实现生产者消费者模式

```
public class ConditionDemo2 {

    private int queueSize = 10;
    private PriorityQueue<Integer> queue = new PriorityQueue<Integer>
(queueSize);
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
```

```
    private Condition notEmpty = lock.newCondition();

    public static void main(String[] args) {
        ConditionDemo2 conditionDemo2 = new ConditionDemo2();
        Producer producer = conditionDemo2.new Producer();
        Consumer consumer = conditionDemo2.new Consumer();
        producer.start();
        consumer.start();
    }

    class Consumer extends Thread {

        @Override
        public void run() {
            consume();
        }

        private void consume() {
            while (true) {
                lock.lock();
                try {
                    while (queue.size() == 0) {
                        System.out.println("队列空，等待数据");
                        try {
                            notEmpty.await();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    queue.poll();
                    notFull.signalAll();
                    System.out.println("从队列里取走了一个数据，队列剩余" +
queue.size() + "个元素");
                } finally {
                    lock.unlock();
                }
            }
        }
    }

    class Producer extends Thread {

        @Override
        public void run() {
            produce();
        }

        private void produce() {
            while (true) {
                lock.lock();
                try {
                    while (queue.size() == queueSize) {
                        System.out.println("队列满，等待有空余");
                        try {
                            notFull.await();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
```

```
                    }
                    queue.offer(1);
                    notEmpty.signalAll();
                    System.out.println("向队列插入了一个元素，队列剩余空间" +
(queueSize - queue.size()));
                } finally {
                    lock.unlock();
                }
            }
        }
    }
}
```

AQS

1、自己用AQS实现一个简单的线程协作器

```
public class OneShotLatch {

    private final Sync sync = new Sync();

    public void signal() {
        sync.releaseShared(0);
    }
    public void await() {
        sync.acquireShared(0);
    }

    private class Sync extends AbstractQueuedSynchronizer {

        @Override
        protected int tryAcquireShared(int arg) {
            return (getState() == 1) ? 1 : -1;
        }

        @Override
        protected boolean tryReleaseShared(int arg) {
           setState(1);

           return true;
        }
    }


    public static void main(String[] args) throws InterruptedException {
        OneShotLatch oneShotLatch = new OneShotLatch();
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    System.out.println(Thread.currentThread().getName()+"尝试获取
latch，获取失败那就等待");
                    oneShotLatch.await();
                    System.out.println("开闸放
行"+Thread.currentThread().getName()+"继续运行");
                }
            }).start();
```

```
        }
        Thread.sleep(5000);
        oneShotLatch.signal();

        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName()+"尝试获取
latch，获取失败那就等待");
                oneShotLatch.await();
                System.out.println("开闸放行"+Thread.currentThread().getName()+"继
续运行");
            }
        }).start();
    }
}
```

获取子线程执行结果

1、演示FutureTask的用法

```
public class FutureTaskDemo {

    public static void main(String[] args) {
        Task task = new Task();
        FutureTask<Integer> integerFutureTask = new FutureTask<>(task);
//        new Thread(integerFutureTask).start();
        ExecutorService service = Executors.newCachedThreadPool();
        service.submit(integerFutureTask);

        try {
            System.out.println("task运行结果："+integerFutureTask.get());

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class Task implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        System.out.println("子线程正在计算");
        Thread.sleep(3000);
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            sum += i;
        }
        return sum;
    }
}
```

2、 演示批量提交任务时，用List来批量接收结果

```
public class MultiFutures {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(20);
        ArrayList<Future> futures = new ArrayList<>();
        for (int i = 0; i < 20; i++) {
            Future<Integer> future = service.submit(new CallableTask());
            futures.add(future);
        }
        Thread.sleep(5000);
        for (int i = 0; i < 20; i++) {
            Future<Integer> future = futures.get(i);
            try {
                Integer integer = future.get();
                System.out.println(integer);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
    }

    static class CallableTask implements Callable<Integer> {

        @Override
        public Integer call() throws Exception {
            Thread.sleep(3000);
            return new Random().nextInt();
        }
    }
}
```

3、演示get方法过程中抛出异常，for循环为了演示抛出Exception的时机：并不是说一产生异常就抛出，直到我们get执行时，才会抛出。

```
public class GetException {

    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(20);
        Future<Integer> future = service.submit(new CallableTask());


        try {
            for (int i = 0; i < 5; i++) {
                System.out.println(i);
                Thread.sleep(500);
            }
            System.out.println(future.isDone());
            future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
            System.out.println("InterruptedException异常");
        } catch (ExecutionException e) {
            e.printStackTrace();
            System.out.println("ExecutionException异常");
        }
    }


    static class CallableTask implements Callable<Integer> {

        @Override
        public Integer call() throws Exception {
            throw new IllegalArgumentException("Callable抛出异常");
        }
    }
}
```

4、 演示get的超时方法，需要注意超时后处理，调用future.cancel()。演示cancel传入true和false的区别，代表是否中断正在执行的任务。

```
public class Timeout {

    private static final Ad DEFAULT_AD = new Ad("无网络时候的默认广告");
    private static final ExecutorService exec =
Executors.newFixedThreadPool(10);

    static class Ad {

        String name;

        public Ad(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return "Ad{" +
                    "name='" + name + '\'' +
                    '}';
        }
    }


    static class FetchAdTask implements Callable<Ad> {

        @Override
        public Ad call() throws Exception {
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                System.out.println("sleep期间被中断了");
                return new Ad("被中断时候的默认广告");
            }
            return new Ad("旅游订票哪家强？找某程");
        }
```

```
    }


    public void printAd() {
        Future<Ad> f = exec.submit(new FetchAdTask());
        Ad ad;
        try {
            ad = f.get(2000, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            ad = new Ad("被中断时候的默认广告");
        } catch (ExecutionException e) {
            ad = new Ad("异常时候的默认广告");
        } catch (TimeoutException e) {
            ad = new Ad("超时时候的默认广告");
            System.out.println("超时，未获取到广告");
            boolean cancel = f.cancel(true);
            System.out.println("cancel的结果: " + cancel);
        }
        exec.shutdown();
        System.out.println(ad);
    }

    public static void main(String[] args) {
        Timeout timeout = new Timeout();
        timeout.printAd();
    }
}
```

高性能缓存

1、最简单的缓存形式：HashMap

```
public class ImoocCache1 {

    private final HashMap<String,Integer> cache = new HashMap<>();

    public synchronized Integer computer(String userId) throws
InterruptedException {
        Integer result = cache.get(userId);
        //先检查HashMap里面有没有保存过之前的计算结果
        if (result == null) {
            //如果缓存中找不到，那么需要现在计算一下结果，并且保存到HashMap中
            result = doCompute(userId);
            cache.put(userId, result);
        }
        return result;
    }

    private Integer doCompute(String userId) throws InterruptedException {
        TimeUnit.SECONDS.sleep(5);
        return new Integer(userId);
    }

    public static void main(String[] args) throws InterruptedException {
        ImoocCache1 imoocCache1 = new ImoocCache1();
        System.out.println("开始计算了");
        Integer result = imoocCache1.computer("13");
```

```
        System.out.println("第一次计算结果："+result);
        result = imoocCache1.computer("13");
        System.out.println("第二次计算结果："+result);

    }
}
```

2、 用装饰者模式，给计算器自动添加缓存功能

```
public class ImoocCache2<A,V> implements Computable<A,V> {

    private final Map<A, V> cache = new HashMap();

    private  final Computable<A,V> c;

    public ImoocCache2(Computable<A, V> c) {
        this.c = c;
    }

    @Override
    public synchronized V compute(A arg) throws Exception {
        System.out.println("进入缓存机制");
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }

    public static void main(String[] args) throws Exception {
        ImoocCache2<String, Integer> expensiveComputer = new ImoocCache2<>(
                new ExpensiveFunction());
        Integer result = expensiveComputer.compute("666");
        System.out.println("第一次计算结果："+result);
        result = expensiveComputer.compute("13");
        System.out.println("第二次计算结果："+result);
    }
}
```

```
/**
 * 描述：      有一个计算函数computer，用来代表耗时计算，每个计算器都要实现这个接口，这样就可以无侵入实现缓存功能
 */
public interface Computable <A,V>{

    V compute(A arg) throws Exception;
}
```

```
/**
 * 描述：     耗时计算的实现类，实现了Computable接口，但是本身不具备缓存能力，不需要考虑缓存的事情
 */
public class ExpensiveFunction implements Computable<String, Integer>{

    @Override
    public Integer compute(String arg) throws Exception {
        Thread.sleep(5000);
        return Integer.valueOf(arg);
    }
}
```

3、缩小synchronized加锁粒度，但性能差，存在线程安全问题

```
public class ImoocCache4<A, V> implements Computable<A, V> {

    private final Map<A, V> cache = new HashMap();

    private final Computable<A, V> c;

    public ImoocCache4(Computable<A, V> c) {
        this.c = c;
    }

    @Override
    public V compute(A arg) throws Exception {
        System.out.println("进入缓存机制");
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            synchronized (this) {
                cache.put(arg, result);
            }
        }
        return result;
    }

    public static void main(String[] args) throws Exception {
        ImoocCache4<String, Integer> expensiveComputer = new ImoocCache4<>(
                new ExpensiveFunction());
        Integer result = expensiveComputer.compute("666");
        System.out.println("第一次计算结果：" + result);
        result = expensiveComputer.compute("666");
        System.out.println("第二次计算结果：" + result);
    }
}
```

4、使用ConcurrentHashMap解决线程安全问题

```
public class ImoocCache5<A, V> implements Computable<A, V> {

    private final Map<A, V> cache = new ConcurrentHashMap<>();

    private final Computable<A, V> c;
```

```
    public ImoocCache5(Computable<A, V> c) {
        this.c = c;
    }

    @Override
    public V compute(A arg) throws Exception {
        System.out.println("进入缓存机制");
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }

    public static void main(String[] args) throws Exception {
        ImoocCache5<String, Integer> expensiveComputer = new ImoocCache5<>(
                new ExpensiveFunction());
        Integer result = expensiveComputer.compute("666");
        System.out.println("第一次计算结果：" + result);
        result = expensiveComputer.compute("666");
        System.out.println("第二次计算结果：" + result);
    }
}
```

5、 利用Future，避免重复计算

```
public class ImoocCache7<A, V> implements Computable<A, V> {

    private final Map<A, Future<V>> cache = new ConcurrentHashMap<>();

    private final Computable<A, V> c;

    public ImoocCache7(Computable<A, V> c) {
        this.c = c;
    }

    @Override
    public V compute(A arg) throws Exception {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> callable = new Callable<V>() {
                @Override
                public V call() throws Exception {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<>(callable);
            f = ft;
            cache.put(arg, ft);
            System.out.println("从FutureTask调用了计算函数");
            ft.run();
        }
        return f.get();
    }
```

```
    public static void main(String[] args) throws Exception {
        ImoocCache7<String, Integer> expensiveComputer = new ImoocCache7<>(
                new ExpensiveFunction());
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666");
                    System.out.println("第一次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666");
                    System.out.println("第三次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("667");
                    System.out.println("第二次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

6、 利用Future，避免重复计算（使用putIfAbsent优化）

```
public class ImoocCache8<A, V> implements Computable<A, V> {

    private final Map<A, Future<V>> cache = new ConcurrentHashMap<>();

    private final Computable<A, V> c;

    public ImoocCache8(Computable<A, V> c) {
        this.c = c;
    }

    @Override
    public V compute(A arg) throws Exception {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> callable = new Callable<V>() {
                @Override
```

```
                public V call() throws Exception {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<>(callable);
            f = cache.putIfAbsent(arg, ft);
            if (f == null) {
                f = ft;
                System.out.println("从FutureTask调用了计算函数");
                ft.run();
            }
        }
        return f.get();
    }

    public static void main(String[] args) throws Exception {
        ImoocCache8<String, Integer> expensiveComputer = new ImoocCache8<>(
                new ExpensiveFunction());
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666");
                    System.out.println("第一次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666");
                    System.out.println("第三次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("667");
                    System.out.println("第二次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

7、利用Future，避免重复计算（考虑计算抛出异常情况）

```
public class ImoocCache9<A, V> implements Computable<A, V> {

    private final Map<A, Future<V>> cache = new ConcurrentHashMap<>();

    private final Computable<A, V> c;

    public ImoocCache9(Computable<A, V> c) {
        this.c = c;
    }

    @Override
    public V compute(A arg) throws InterruptedException, ExecutionException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                Callable<V> callable = new Callable<V>() {
                    @Override
                    public V call() throws Exception {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<>(callable);
                f = cache.putIfAbsent(arg, ft);
                if (f == null) {
                    f = ft;
                    System.out.println("从FutureTask调用了计算函数");
                    ft.run();
                }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                System.out.println("被取消了");
                cache.remove(arg);
                throw e;
            } catch (InterruptedException e) {
                cache.remove(arg);
                throw e;
            } catch (ExecutionException e) {
                System.out.println("计算错误，需要重试");
                cache.remove(arg);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        ImoocCache9<String, Integer> expensiveComputer = new ImoocCache9<>(
                new MayFail());
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666");
                    System.out.println("第一次的计算结果：" + result);
                } catch (Exception e) {
```

```
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666");
                    System.out.println("第三次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("667");
                    System.out.println("第二次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

```
/**
 * 描述：       耗时计算的实现类，有概率计算失败
 */
public class MayFail implements Computable<String, Integer>{

    @Override
    public Integer compute(String arg) throws Exception {
        double random = Math.random();
        if (random > 0.5) {
            throw new IOException("读取文件出错");
        }
        Thread.sleep(3000);
        return Integer.valueOf(arg);
    }
}
```

8、出于安全性考虑，缓存需要设置有效期，到期自动失效，否则如果缓存一直不失效，那么带来缓存不一致等问题

```
public class ImoocCache10<A, V> implements Computable<A, V> {

    private final Map<A, Future<V>> cache = new ConcurrentHashMap<>();

    private final Computable<A, V> c;

    public ImoocCache10(Computable<A, V> c) {
```

```
        this.c = c;
    }

    @Override
    public V compute(A arg) throws InterruptedException, ExecutionException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                Callable<V> callable = new Callable<V>() {
                    @Override
                    public V call() throws Exception {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<>(callable);
                f = cache.putIfAbsent(arg, ft);
                if (f == null) {
                    f = ft;
                    System.out.println("从FutureTask调用了计算函数");
                    ft.run();
                }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                System.out.println("被取消了");
                cache.remove(arg);
                throw e;
            } catch (InterruptedException e) {
                cache.remove(arg);
                throw e;
            } catch (ExecutionException e) {
                System.out.println("计算错误，需要重试");
                cache.remove(arg);
            }
        }
    }

    public V computeRandomExpire(A arg) throws ExecutionException,
InterruptedException {
        long randomExpire = (long) (Math.random() * 10000);
        return compute(arg, randomExpire);
    }

    public final static ScheduledExecutorService executor =
Executors.newScheduledThreadPool(5);

    public V compute(A arg, long expire) throws ExecutionException,
InterruptedException {
        if (expire>0) {
            executor.schedule(new Runnable() {
                @Override
                public void run() {
                    expire(arg);
                }
            }, expire, TimeUnit.MILLISECONDS);
        }
        return compute(arg);
```

```
    }

    public synchronized void expire(A key) {
        Future<V> future = cache.get(key);
        if (future != null) {
            if (!future.isDone()) {
                System.out.println("Future任务被取消");
                future.cancel(true);
            }
            System.out.println("过期时间到，缓存被清除");
            cache.remove(key);
        }
    }
    public static void main(String[] args) throws Exception {
        ImoocCache10<String, Integer> expensiveComputer = new ImoocCache10<>(
                new MayFail());
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666",5000L);
                    System.out.println("第一次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("666");
                    System.out.println("第三次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Integer result = expensiveComputer.compute("667");
                    System.out.println("第二次的计算结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();


        Thread.sleep(6000L);
        Integer result = expensiveComputer.compute("666");
        System.out.println("主线程的计算结果：" + result);
    }
}
```