每日Tips（多线程 01.02～）

2018.01.12 ——　使用redis实现分布式锁

lock

```
public boolean lock(String key, long expireTime) {

    //
    long now = Instant.now().toEpochMilli();

    // rediskeyvalue
    long lockExpireTime = now + expireTime;

    // setIfAbsent valuevaluetruefalsevaluefalse
    boolean isLock = redisTemplate.opsForValue().setIfAbsent(key, String.
valueOf(lockExpireTime));

    if (isLock) {
        // , keylockExpireTime
        redisTemplate.expire(key, 1, TimeUnit.HOURS);
        return true;
    } else {
        // redis
        Object lockExpireTimeFromRedis = redisTemplate.opsForValue().get
(key);

        if (lockExpireTimeFromRedis != null) {
            long oldExpireTime = Long.parseLong((String)
lockExpireTimeFromRedis);

            // ,,
            if (oldExpireTime <= now) {

                //  getAndSetCAS1oldExpireTimeredisredis
                // 234 oldExpireTime redislockExpireTimeredis
                // currentExpireTime
                long currentExpireTime = Long.parseLong(redisTemplate.
opsForValue().getAndSet(key, String.valueOf(lockExpireTime)));

                //,
                if (currentExpireTime == oldExpireTime) {
                    redisTemplate.expire(key, 1, TimeUnit.HOURS);
                    return true;
                } else {
                    return false;
                }
            }
        } else {
```

```
            //
            return false;
        }
    }
    return false;
}


public boolean unlock(String key) {
    redisTemplate.delete(key);
    return true;
}
```

1. 所有的redis操作可以添加重试操作，起到一个防抖的作用
2. 该分布式锁是一个不可重入锁
3. 如果多个进程获取到的 lockExpireTime（毫秒级别）相同的话，该分布式锁会出现锁失效的问题

2018.01.11 —— ForkJoin使用演示

forkjoin

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

        long l = System.currentTimeMillis();
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        // 1+2+3+4
        CountTask task = new CountTask(1, 10000000L);
        //
        Future<Long> result = forkJoinPool.submit(task);
        try {
            System.out.println(result.get());
            if (task.isCompletedAbnormally()) {//
                task.getException().printStackTrace();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(System.currentTimeMillis() - l);
    }
}

class CountTask extends RecursiveTask<Long> {

    //
    private static final long THRESHOLD = 500000L;
    private long start;
    private long end;

    public CountTask(long start, long end) {
```

```
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        long sum = 0;
        //
        boolean canCompute = (end - start) <= THRESHOLD;
        if (canCompute) {
            for (long i = start; i <= end; i++) {
                sum += i;
            }
        } else {
            //
            long middle = (start + end) / 2;
            CountTask leftTask = new CountTask(start, middle);
            CountTask rightTask = new CountTask(middle + 1, end);
            //
            leftTask.fork();
            rightTask.fork();
            //
            long leftResult = leftTask.join();
            long rightResult = rightTask.join();
            //
            sum = leftResult + rightResult;
        }
        return sum;
    }
}
```

2018.01.10 —— 本地堆缓存

cache

```
public class CacheTest {

    private static final com.google.common.cache.Cache<Long, Long> cache =
com.google.common.cache.CacheBuilder.newBuilder()
            /**
             * Guava Cache ConcurrentHashMapconcurrencyLevelSegment
             */
            .concurrencyLevel(16)
            /**
             * 3
             */
            .expireAfterWrite(3, TimeUnit.HOURS)
```

```
            /**
             * 3TTL
             * expireAfterWrite
             */
            .expireAfterAccess(3, TimeUnit.HOURS)
            /**
             * LRU
             */
            .maximumSize(50_000L)
            /**
             * weakKeys & weakValues key & value
             */
            .weakKeys()
            .weakValues()
            /**
             * value
             */
            .softValues()
            .build();

    public static void main(String[] args) {
        /**
         *
         */
        cache.put(12L, 12L);
        /**
         *
         */
        cache.getIfPresent(12L);
        /**
         *
         */
        cache.cleanUp();
        /**
         *
         */
        try {
            System.out.println(cache.get(13L, () -> 1L));
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

2018.01.09 —— RateLimiter使用

ratelimiter

```
public class RateLimit {

    /**
     * 50
     */
    private static RateLimiter rateLimiter = RateLimiter.create(50.0);
    /**
     *
     */
    private static List<String> taskList = Collections.synchronizedList
(new LinkedList<String>());

    public static void main(String[] args) throws InterruptedException {
        // 7
        ExecutorService executor = Executors.newFixedThreadPool(7);

        // 7
        for (int i = 0; i < 7; i++) {
            executor.execute(() -> {
                for (; ; ) {
                    rateLimiter.acquire();
                    taskList.add("task");
                }
            });
        }
        // 10s
        TimeUnit.SECONDS.sleep(10);
        System.out.println(String.format("task size -> %d", taskList.
size()));
        System.exit(0);
    }
}
```

1. RateLimiter限制每秒发放50个令牌，线程睡眠了十秒。所以该程序输出的task size 在500左右
2. RateLimiter使用的是一种叫令牌桶的流控算法，RateLimiter会按照一定的频率往桶里扔令牌，线程拿到令牌才能执行，比如你希望自己的应用程序QPS不要超过1000，那么RateLimiter设置1000的速率后，就会每秒往桶里扔1000个令牌
3. 在做刷接口、刷数据等需要较为精准的控流操作时可以考虑这个工具

2018.01.08 —— HashMap、HashTable、ConcurrentHashMap比较

put

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

    HashMap<String, String> hashMap = new HashMap<>();
    Hashtable<String, String> hashTable = new Hashtable();
    ConcurrentHashMap<String, String> chm = new ConcurrentHashMap<>();

    if (hashMap.get("name") == null) {
        hashTable.put("name", "yz");
    }
    if (hashTable.get("name") == null) {
        hashTable.put("name", "yz");
    }
    chm.putIfAbsent("name", "yz");
}
```

1. HashMap与Hashtable、ConcurrentHashMap主要的区别在于HashMap不是同步的、线程不安全的它不适合应用于多线程并发环境下
2. 可以使用Collections.synchronizedMap(HashMap)来包装HashMap作为同步容器，这时它的作用几乎与Hashtable一样,当每次对Map做修改操作的时候都会锁住这个Map对象，而ConcurrentHashMap会基于并发的等级来划分整个Map来达到线程安全，它只会锁操作的那一段数据而不是整个Map都上锁
3. Hashtable是jdk1的一个遗弃的类，它把所有方法都加上synchronized关键字来实现线程安全。所有的方法都同步这样造成多个线程访问效率特别低

2018.01.05 —— ThreadPoolExecutor的使用

ThreadPoolExecutor的创建

create

```
public void create() {
    ThreadPoolExecutor pool = new ThreadPoolExecutor(
            /**
             * corePoolSize
             *
             *
             * prestartAllCoreThreads
             */
            10,
            /**
             * maximumPoolSize
             *
             *
             */
            20,
            /**
             * keepAliveTime
             *
             */
            1000,
            /**
             * TimeUnit:
```

```
             *
             *
             * DAYS
             * HOURS
             * MINUTES
             * (MILLISECONDS)
             * (MICROSECONDS, )
             * (NANOSECONDS, )
             *
             */
            TimeUnit.MILLISECONDS,
            /**
             * runnableTaskQueue
             * :
             *
             * ArrayBlockingQueue
             * FIFO
             *
             * LinkedBlockingQueueFIFO
             * Executors.newFixedThreadPool()
             *
             * SynchronousQueue
             *
             * Executors.newCachedThreadPool
             *
             * PriorityBlockingQueue
             *
             */
            new ArrayBlockingQueue<>(100),
            /**
             * ThreadFactory
             * Debug
             */
            new InitThreadFactory(),
            /**
             * RejectedExecutionHandler
             *
             * JDK1.5
             *
             * CallerRunsPolicy
             * DiscardOldestPolicy
             * DiscardPolicy
             * AbortPolicy
             *
             * RejectedExecutionHandler
             *
             *
             */
            new ThreadPoolExecutor.DiscardOldestPolicy()
    );
}
```

```
/**
 *
 */
class InitThreadFactory implements ThreadFactory {

    AtomicInteger num = new AtomicInteger();

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "thread-" + num.incrementAndGet());
    }
}
```

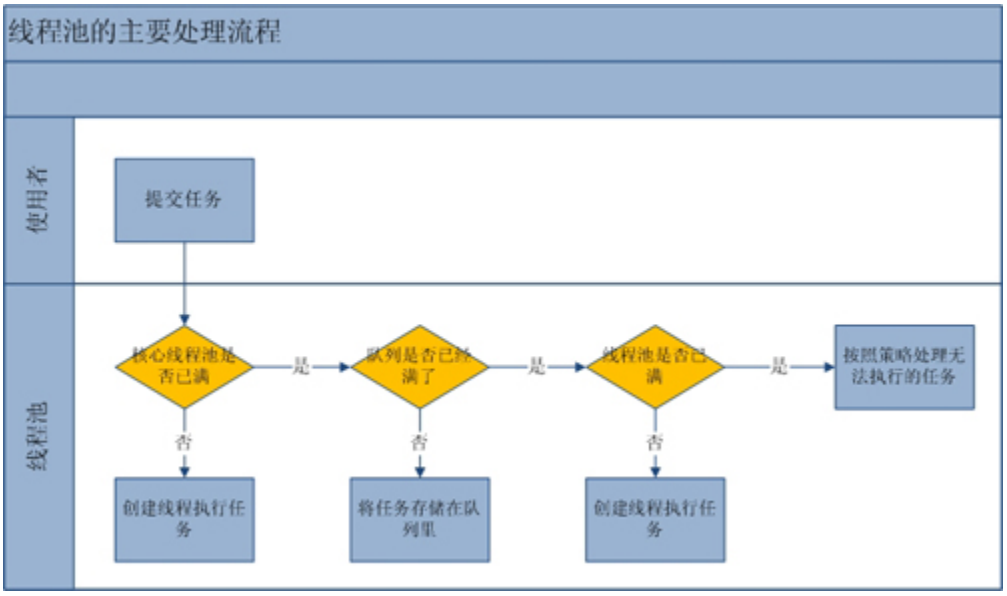
任务提交

1. ThreadPoolExecutor有两种提交方式：execute 和 submit
2. execute提交不需要返回结果的任务，submit提交需要返回结果的任务
3. submit方法会理解返回一个Future对象，Future的get方法会阻塞直到需要的结果返回才会向下执行
4. Future除了无参的get()方法还有带有参数的get(long,TimeUnit)方法，在指定时间内如果没有返回想要的结果的话会抛出TimeoutException

ThreadPoolExecutor关闭

1. shutdown和shutdownNow两个方法都可以关闭线程池
2. shutdown会将线程池的状态设置成SHUTDOWN状态，然后尝试中断所有没有正在执行任务的线程
3. shutdownNow会将线程池的状态设置成STOP，然后遍历线程池中的工作线程，逐个调用线程的interrupt方法来中断线程
4. 两个方法的相同点就在于如果任务不响应中断的话都无法关闭线程。不同点在于shutdown不会去中断正在工作的线程。例如想要关闭一个正在sleep的线程，shutdown不会去发中断信号。而shutdownNow会发中断信号，正在sleep的线程接到中断信号会抛中断异常返回

ThreadPoolExecutor大致流程



1. 线程池判断基本线程池是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程
2. 线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程
3. 线程池判断整个线程池是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务

2018.01.04 —— 自制线程池

```
```

线程池

```
public class ThreadPool<T extends Runnable> {

    private final LinkedList<T> jobs = new LinkedList<>();
    private final List<Worker> workers = new ArrayList<>();
    private int threadNum = 0;

    public ThreadPool(int num) {
        initializeWokers(num);
    }

    private void initializeWokers(int num) {
        for (int i = 0; i < num; i++) {
            Worker worker = new Worker();
            Thread thread = new Thread(worker, "ThreadPool-Worker-" +
++threadNum);
            thread.start();
            workers.add(worker);
        }
    }

    public void execute(T job) {
        if (job != null) {
            synchronized (jobs) {
                jobs.addLast(job);
                jobs.notify();
            }
        }
    }

    class Worker implements Runnable {

        private volatile boolean running = true;

        @Override
        public void run() {
            while (running) {
                T job = null;
                synchronized (jobs) {
                    while (jobs.isEmpty()) {
                        try {
                            jobs.wait();
                        } catch (InterruptedException ex) {
                            return;
                        }
                    }
                    job = jobs.removeFirst();
                }
                if (job != null) {
                    job.run();
                }
            }
```

```
            }
        }
    }
```

虽然这个线程池简易到了无法在生产环境中使用的程度，但是通过这个简单的实现还是可以了解到在生产环境中使用的线程池是怎么工作的，其流程大致如下：

1. 调用ThreadPool初始化方法传入线程池要初始化线程的数量
2. 循环调用线程并执行start方法，这时候所有的线程都启动了，但是由于jobs是空的所以都会被wait掉
3. 当有任务提交进来的时候，jobs任务队列不再为空，并且notify一个工作线程进行工作
4. 工作完之后的线程发现jobs又空了，所以它会重新进入wait状态

上面的流程是个非常理想的状态，因为任务量很小不会出现任务堆积。如果有大量任务短时间内被塞到这个线程池的话jobs队列就会出现堆积。如果任务量再大一些可能就会出现大量任务对象存在强引用无法被垃圾回收器收集，最终导致内存溢出的尴尬情况。那么在这个时候是不是可以考虑把昨天的阻塞队列组装到这个线程池上呢，当任务量达到最大数量的时候，再提交任务的时候就会被阻塞。从而保证了系统至少是可用的。如果感觉阻塞不是特别友好，也可以去考虑把任务直接丢弃或者抛出异常等策略。

java的工程代码中是不需要去考虑这些较为复杂并且与业务无关的代码的，它提供了一系列优秀的工具来解决各种并发场景，从明天开始就为大家介绍一些常用工具的使用，敬请期待。

2018.01.03 —— ReentrantLock

使用ReentrantLock实现一个简单的有界阻塞队列

queue

```
public static class Queue<T> {

    private final LinkedList<T> queue = new LinkedList<>();

    private int size = 0;
    private int num = 0;

    private ReentrantLock lock = new ReentrantLock();
    private Condition putCondition = lock.newCondition();
    private Condition getCondition = lock.newCondition();

    public Queue(int size) {
        this.size = size;
    }

    public void put(T t) {
        try {
            lock.lock();
            while (num == size) {
                putCondition.await();
            }
            queue.addLast(t);
            num++;
            getCondition.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public T get() {
        try {
            lock.lock();
            while (num == 0) {
                getCondition.await();
            }
            T t = queue.removeFirst();
            num--;
            putCondition.signalAll();
            return t;
        } catch (InterruptedException e) {
            e.printStackTrace();
            return null;
        } finally {
            lock.unlock();
        }
    }
}
```

ReentrantLock与synchronized的区别：

1. synchronized既可以加在方法上，也可以加载特定代码块上，而ReentrantLock需要显示地指定起始位置和终止位置
2. synchronized是托管给JVM执行的，ReentrantLock的锁定是通过代码实现的，它有比synchronized更精确的线程语义
3. synchronized获取锁和释放锁的方式都是在块结构中，当获取多个锁时，必须以相反的顺序释放，并且是自动解锁。而ReentrantLock则需要开发人员手动释放，并且必须在finally中释放，否则会引起死锁

2018.01.02 —— 基本概念

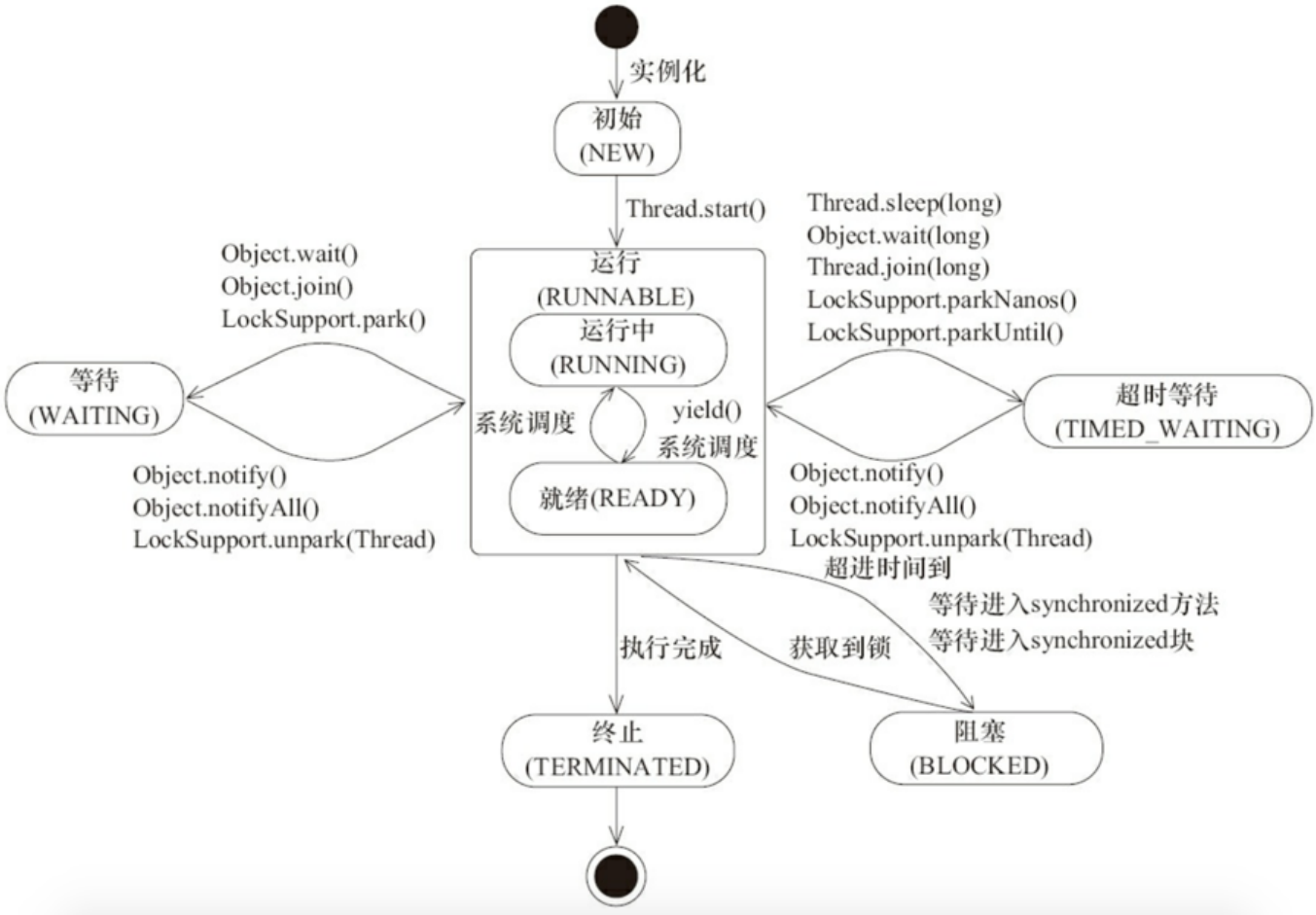
1. 什么是线程

现代操作系统在运行一个程序时，会为其创建一个进程。例如，启动一个Java程序，操作系统就会创建一个Java进程。现代操作系统调度的最小单元是线程，也叫轻量级进程（Light Weight Process），在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。处理器在这些线程上高速切换，让使用者感觉到这些线程在同时执行。

2. 为什么要使用多线程

1. 线程是现在操作系统调度的最小单元，它相比于进程来说最大的优势在于轻量。可以最大程度的利用多处理器所带来的的优势
2. 同一进程内可以创建多个线程，并且这些线程共享内存空间。所以同一进程内线程与线程之间可以低成本高效率的通信
3. 一些不需要同步的模型，可以利用多线程技术抽离出异步任务，在不改变原来任务正确性的前提下减少整个方法的RT

3. 线程的几种状态及转换条件



图片中出现的LockSupport相关方法请暂时忽略

运行中（RUNNABLE）与就绪（READY）两种状态可以并称为运行状态（RUNNABLE）。所以在Java线程模型中一般会有六种状态。初始（NEW）、运行（RUNNABLE）、结束（TERMINATED）、等待（WAITING）、超时等待（TIMED_WAITING）、阻塞（BLOCKED）。

记起来非常简单，一个程序执行下来一定有开始、运行、结束。运行期间如果遇到锁会变为阻塞状态，如果遇到需要等待的操作，会变为等待状态。为了防止线程一直处在等待状态没有被唤醒，某些场景下也会需要设置超时时间变为超时等待状态，超时等待与等待相比好处就在于超时时间到后如果还没有被通知结束等待，就会自己结束等待状态。

这里比较难区分的是阻塞与等待两种状态，可以稍微讨论一下

1. 区分方法：如果当前等待状态可以依赖其他线程的通知被唤醒，那么就是等待状态，如果不能，那么这个线程就是处于阻塞状态了
2. 本质区别：等待状态会响应中断（Thread的 interrupt() 方法），而阻塞状态则不会响应

启动新线程

start

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

    Thread t = new Thread(() -> {
        System.out.println("~~");
    });
    t.start();
}
```

结束线程

end

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

    Thread t1 = new Thread(() -> {
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("t1~");
        }
    });
    t1.start();

    Thread t2 = new Thread(() -> {
        while (true) {
            System.out.println("t2~");
            try {
                TimeUnit.SECONDS.sleep(5 * 1000);
            } catch (InterruptedException e) {
                System.out.println("~");
                return;
            }
        }
    });
    t2.start();

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

1. 请自动忽略jdk中已不推荐使用的如stop等方法
2. `t.interrupt()tt`

3. 响应中断信号有两种方式一种是判断中断标识另一种是抛出中断异常

4. t1如果不收到中断信号 Thread.currentThread().isInterrupted() 将一直返回false，当收到中断信号后会返回true。while循环结束，由于后面再没有代码，所以整个线程执行完成并结束
5. Thread.currentThread().isInterrupted() 这个判断条件可以被 Thread.interrupted() 替换。两者的不同在于前者不会改变线程的中断标识，后者会将中断标识置位
6. 如t2所示，线程在运行阶段如果遇到sleep、wait等使线程处于等待状态的方法时，如果中断标志为true则抛出中断异常。捕获该异常做响应的中断处理
7. 被synchronize阻塞的方法不会响应中断，但是jdk1.5中的Lock除外，因为Lock的等待是LockSupport实现的，而它本质上也是使线程进入了等待而不是阻塞

join方法

join

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

    Thread t = new Thread(() -> {
        try {
            System.out.println("begin");
            TimeUnit.SECONDS.sleep(10);
            System.out.println("AAB");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    t.start();
    try {
        t.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("CCD");
}
```

1. 该程序的执行现象是首先输出 begin，等10秒钟之后，输出AAB，紧接着输出CCD
2. join的作用是等待t线程执行完之后 t.join() 这个方法调用才会返回，之后主线程才会继续执行
3. t.join() t.interrupt() jointjoin

wait方法

wait

```
private static final Object LOCK = new Object();

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

    Thread t = new Thread(() -> {
        synchronized (LOCK) {
            try {
                LOCK.wait();
                System.out.println("10~");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    t.start();

    TimeUnit.SECONDS.sleep(10);
    synchronized (LOCK) {
        LOCK.notify();
    }
}
```

1. 这段代码执行的现象是10秒之后打印了 “10秒之后这里执行了~” 这句话
2. wait的作用是释放当前持有的锁状态，并进入等待状态，直到被notify或者收到中断才会继续执行。两者相同之处在于notify和被中断都会去重新尝试获得锁状态，不同之处在于notify持有锁状态后会正常执行，而如果被中断持有锁状态后会抛出异常
3. 需要注意synchronized的锁对象和调用wait、notify的对象是一个对象，并且wait、notify都是在持有锁之后调用的，如果锁对象不是同一个或者没有在锁环境下调用都会抛出异常
4. 我的理解jdk之所以会有2中这样的设计是因为本身wait、notify的存在就是为了实现同一个锁下多个线程之间的通信，如果脱离锁，它们就没有意义了
5. wait(long) 和sleep(long)的区别就在于wait放弃了持有的锁状态而sleep没有放弃，并且sleep是允许在非持有锁的状态下执行的。甚至可以说wait和sleep不是一个概念上的东西

4. 线程的一些其他概念

1. 在Oracle为Linux提供的java虚拟机中，线程的优先级被忽略，因此所有线程具有相同的优先级
2. 设置为守护线程的线程会在所有非守护线程执行完之后也停止运行，甚至守护线程中的finally代码块在那个时候也不保证一定会被执行