Java 并发编程之美：并发编程实践

一、前言

Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

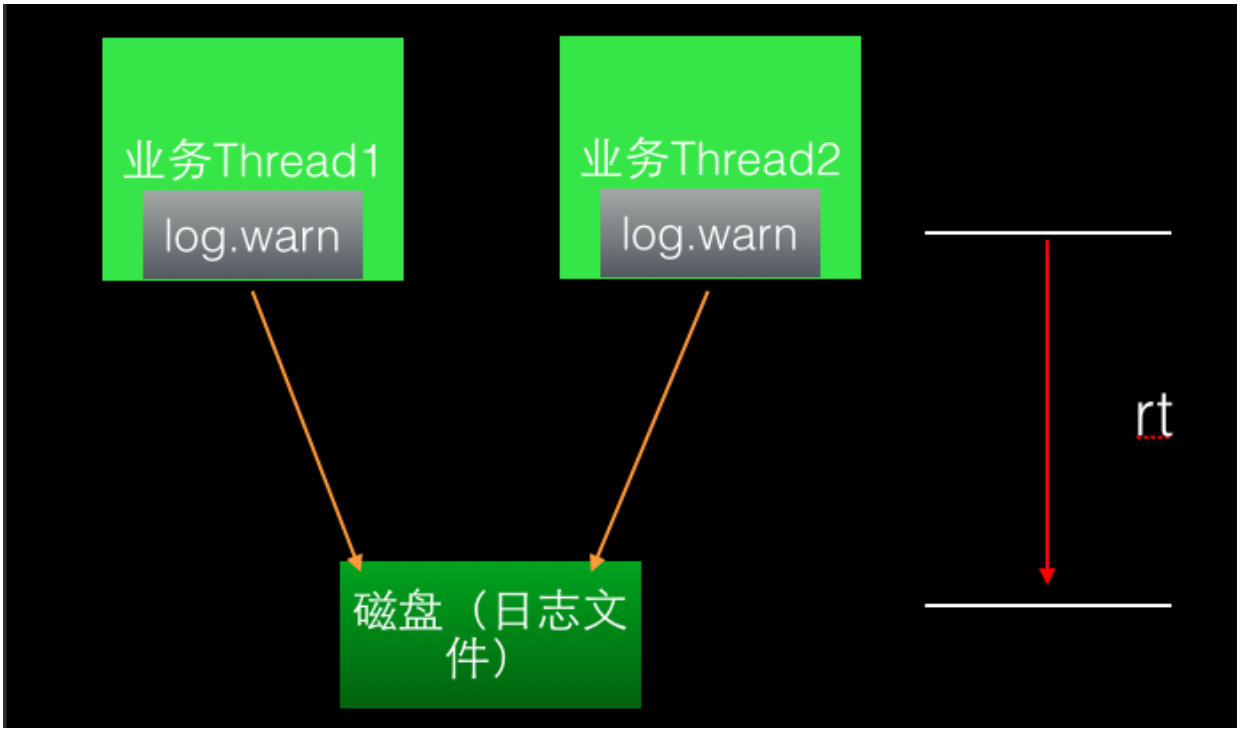
本 Chat 作为 Java 并发编程之美系列的终章，我们来讲解并发编程中的一些实践与经常会遇到的问题，内容如下：（建议先阅读 并发编程高级篇之三 - 锁 ）

- Logback 日志框架中异步日志打印中 ArrayBlockingQueue 的使用，Logback 是如何借助队列将同步转换为异步，节省调用线程 RT 响应时间的？
- 并发组件 ConcurrentHashMap 使用注意项，虽然 ConcurrentHashMap 是并发安全的组件，但是使用不当还是会造成程序错误，这里列出一些常见的出错点，并讲解如何避免。
- 使用定时器 Timer 的时候需要注意的一些问题，结合源码讲解出现问题的原因，以及如何避免。
- SimpleDateFormat 是线程不安全？为啥？应该如何正确使用？
- 线程池使用 FutureTask 时候需要注意的一点事，FutureTask 使用不当可能会造成调用线程一直阻塞，如何避免？
- 使用 ThreadLocal 不当可能会导致内存泄露，本节讲解为何会出现内存泄露，以及如何避免。

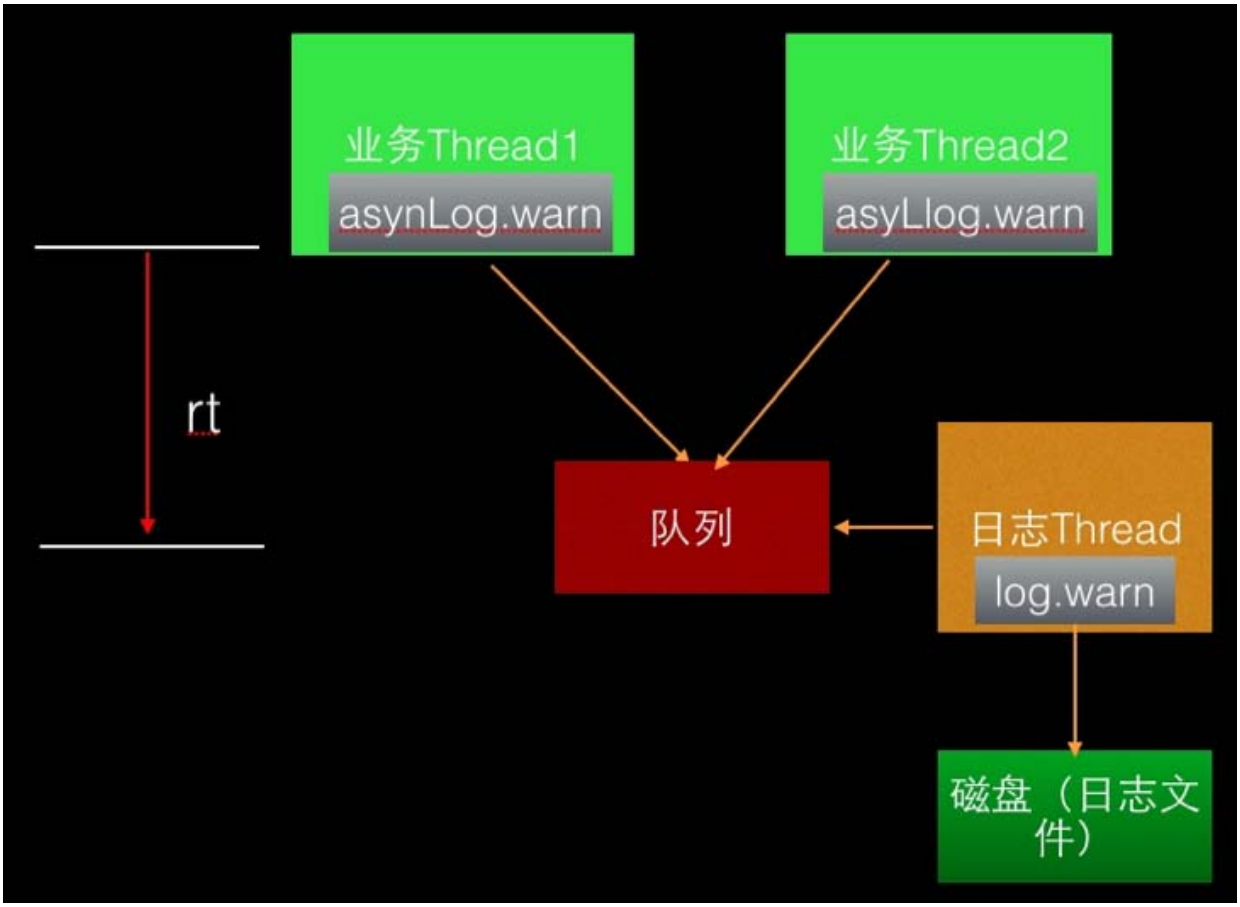
二、Logback 框架中异步日志打印中 ArrayBlockingQueue 的使用

2.1 异步日志打印模型概述

在高并发并且响应时间要求比较小的系统中同步打日志已经满足不了需求了，这是因为打日志本身是需要同步写磁盘的，会造成 rt 增加，如下图同步日志打印模型为：



异步模型是业务线程把要打印的日志任务写入一个队列后直接返回，然后使用一个线程专门负责从队列中获取日志任务写入磁盘，其模型具体如下图：



如图可知其实 logback 的异步日志模型是一个多生产者单消费者模型，通过使用队列把同步日志打印转换为了异步，业务线程调用异步 appender 只需要把日志任务放入日志队列，日志线程则负责使用同步的 appender 进行具体的日志打印到磁盘。

2.2 异步日志打印具体实现

要把同步日志打印改为异步需要修改 logback 的 xml 配置文件如下：

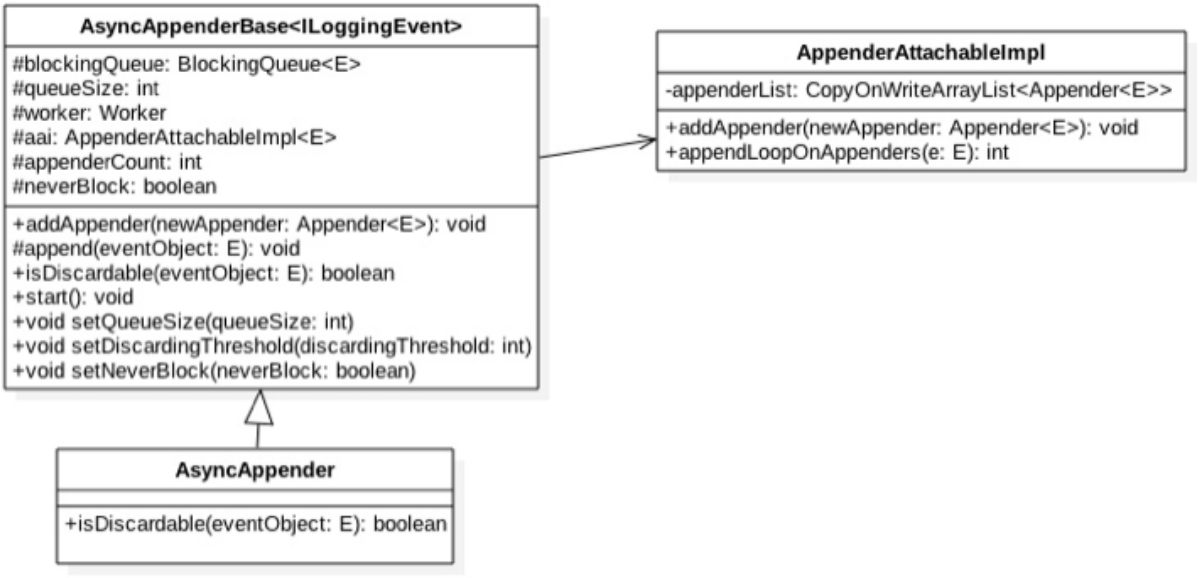
```
    <appender name="PROJECT"
class="ch.qos.logback.core.FileAppender">
        <file>project.log</file>
        <encoding>UTF-8</encoding>
        <append>true</append>

        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- daily rollover -->
            <fileNamePattern>project.log.%d{yyyy-MM-dd}
</fileNamePattern>
            <!-- keep 7 days' worth of history -->
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern><![CDATA[
%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method}
%X{requestURIWithQueryString} [ip=%X{remoteAddr},
ref=%X{referrer}, ua=%X{userAgent}, sid=%X{cookie.JSESSIONID}]%n
%-5level %logger{35} - %m%n
            ]]></pattern>
        </layout>
    </appender>

    <appender name="asyncProject"
class="ch.qos.logback.classic.AsyncAppender">
        <discardingThreshold>0</discardingThreshold>
        <queueSize>1024</queueSize>
        <neverBlock>true</neverBlock>
        <appender-ref ref="PROJECT" />
    </appender>
     <logger name="PROJECT_LOGGER" additivity="false">
        <level value="WARN" />
        <appender-ref ref="asyncProject" />
    </logger>
```

可知 AsyncAppender 是实现异步日志的关键，下节主要讲这个的内部实现。

2.3 异步日志实现原理

本文使用 logback-classic 的版本为 1.0.13。

首先从 AsyncAppender 的类图结构来从全局了解下 AsyncAppender 中组件构成：



- 如上图可知 AsyncAppender 继承自 AsyncAppenderBase，其中后者具体实现了异步日志模型的主要功能，前者只是重写了其中的一些方法。另外从上类图可知 logback 中的异步日志队列是一个阻塞队列, 后面会知道其实是一个有界阻塞队列 ArrayBlockingQueue, 其中 queueSize 是有界队列的元素个数默认为 256。
- worker 是个线程，也就是异步日志打印模型中的单消费者线程，aai 是一个 appender 的装饰器里面存放同步日志的 appender, 其中 appenderCount 记录 aai 里面附加的同步 appender 的个数；neverBlock 是当日志队列满了的时候是否阻塞打日志的线程的一个开关；discardingThreshold 是一个阈值，当日志队列里面空闲个数小于该值时候新来的某些级别的日志会被直接丢弃，下面会具体讲到。

首先我们来看下何时创建的日志队列以及何时启动的消费线程，这需要看下 AsyncAppenderBase 的 start 方法，该方法是在解析完毕配置 AsyncAppenderBase 的 xml 的节点元素后被调用：

```
public void start() {
  ...
  //（1）日志队列为有界阻塞队列
  blockingQueue = new ArrayBlockingQueue<E>(queueSize);
  //(2)如果没设置discardingThreshold则设置为队列大小的1/5
  if (discardingThreshold == UNDEFINED)
    discardingThreshold = queueSize / 5;
  //(3)设置消费线程为守护线程，并设置日志名称
  worker.setDaemon(true);
  worker.setName("AsyncAppender-Worker-" + worker.getName());
  //(4)设置启动消费线程
  super.start();
  worker.start();
}
```

- 从上代码可知 logback 使用的队列是有界队列 ArrayBlockingQueue，之所以使用有界队列是考虑到内存溢出问题，在高并发下写日志的 qps 会很高如果设置为无界队列队列本身会占用很大内存，很可能会造成 OOM。

- 这里消费日志队列的 worker 线程被设置为了守护线程，意味着当主线程运行结束并且当前没有用户线程时候该 worker 线程会随着 JVM 的退出而终止，而不管日志队列里面是否还有日志任务未被处理。另外这里设置了线程的名称是个很好的习惯，因为这在查找问题的时候很有帮助，根据线程名字就可以定位到是哪个线程。

既然是有界队列那么肯定需要考虑如果队列满了，该如何处置，是丢弃老的日志任务，还是阻塞日志打印线程直到队列有空余元素那? 要回答这个问题，我们需要看看具体进行日志打印的 AsyncAppenderBase 的 append 方法：

```
protected void append(E eventObject) {
   //（5）调用AsyncAppender重写的isDiscardable
   if (isQueueBelowDiscardingThreshold() &&
isDiscardable(eventObject)) {
       return;
   }
   ...
   //（6）放入日志任务到队列
   put(eventObject);
}

    private boolean isQueueBelowDiscardingThreshold() {
        return (blockingQueue.remainingCapacity() <
discardingThreshold);
    }
```

其中 (5) 调用了调用 AsyncAppender 重写的 isDiscardable
具体内容为：

```
    //(7)
    protected boolean isDiscardable(ILoggingEvent event) {
        Level level = event.getLevel();
        return level.toInt() <= Level.INFO_INT;
    }
```

结合 (5)(7) 可知如果当前日志的级别小于 INFO_INT 级别并且当前队列的剩余容量小于 discardingThreshold 时候会直接丢弃这些日志任务。

下面看具体步骤 (6) 的 put 方法：

```
    private void put(E eventObject) {
        //(8)
        if (neverBlock) {
            blockingQueue.offer(eventObject);
        } else {
            try {//(9)
                blockingQueue.put(eventObject);
            } catch (InterruptedException e) {
```

```
                // Interruption of current thread when in
doAppend method should not be consumed
                // by AsyncAppender
                Thread.currentThread().interrupt();
            }
        }
    }
```

可知如果 neverBlock 设置为了 false（默认为 false）则会调用阻塞队列的 put 方法，而 put 是阻塞的，也就是说如果当前队列满了，如果在企图调用 put 方法向队列放入一个元素则调用线程会被阻塞直到队列有空余空间。

这里有必要提下其中第 (9) 步当日志队列满了的时候 put 方法会调用 await() 方法阻塞当前线程，如果其它线程中断了该线程，那么该线程会抛出 InterruptedException 异常，那么当前的日志任务就会被丢弃了。

如果 neverBlock 设置为了 true 则会调用阻塞队列的 offer 方法，而该方法是非阻塞的，如果当前队列满了，则会直接返回，也就是丢弃当前日志任务。

最后看下 addAppender 方法内做了啥：

```
    public void addAppender(Appender<E> newAppender) {
        if (appenderCount == 0) {
            appenderCount++;
            ...
            aai.addAppender(newAppender);
        } else {
            addWarn("One and only one appender may be attached to
AsyncAppender.");
            addWarn("Ignoring additional appender named [" +
newAppender.getName() + "]");
        }
    }
```

如上代码可知一个异步 appender 只能绑定一个同步 appender, 这个 appender 会被放到 AppenderAttachableImpl 的 appenderList 列表里面。

到这里我们已经分析完了日志生产线程放入日志任务到日志队列的实现，下面一起来看下消费线程是如何从队列里面消费日志任务并写入磁盘的，由于消费线程是一个线程，那就从 worker 的 run 方法看起：

```
class Worker extends Thread {

        public void run() {

            AsyncAppenderBase<E> parent = AsyncAppenderBase.this;
            AppenderAttachableImpl<E> aai = parent.aai;
```

```
                //（10）一直循环直到该线程被中断
                while (parent.isStarted()) {
                    try {//（11）从阻塞队列获取元素
                        E e = parent.blockingQueue.take();
                        aai.appendLoopOnAppenders(e);
                    } catch (InterruptedException ie) {
                        break;
                    }
                }

                //（12）到这里说明该线程被中断，则吧队列里面的剩余日志任务
                //刷新到磁盘
                for (E e : parent.blockingQueue) {
                    aai.appendLoopOnAppenders(e);
                    parent.blockingQueue.remove(e);
                }
                ...
..          }
    }
```

其中（11）从日志队列使用 take 方法获取一个日志任务，如果当前队列为空则当前线程会阻塞到 take 方法直到队列不为空才返回，获取到日志任务后会调用 AppenderAttachableImpl 的 aai.appendLoopOnAppenders 方法，该方法会循环调用通过 addAppender 注入的同步日志 appener 具体实现日志打印到磁盘的任务。

三、ConcurrentHashMap 的使用注意项

ConcurrentHashMap 虽然为并发安全的组件，但是使用不当还是会导致程序错误，本节通过使用简单的案例来复现这些问题并给出开发时候如何进行避免的策略

这里借用直播的一个场景，直播业务中，每个直播间对应一个 topic，每个用户进入直播间时候会把自己设备 id 绑定到这个 topic 上，也就是一个 topic 对应一堆用户设备，可知可以使用 map 来维护这些信息，key 为 topic，value 为设备的 list。下面通过代码模拟多用户同时进入直播间时候 map 信息的维护：

```
public class TestMap {
    //(1)创建map,key为topic,value为设备列表
    static ConcurrentHashMap<String, List<String>> map = new
ConcurrentHashMap<>();
    public static void main(String[] args) {
        //(2)进入直播间topic1 线程one
        Thread threadOne = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device1");
                list1.add("device2");
```

```
                map.put("topic1", list1);
                System.out.println(JSON.toJSONString(map));
            }
        });
        //(3)进入直播间topic1 线程two
        Thread threadTwo = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device11");
                list1.add("device22");

                map.put("topic1", list1);

                System.out.println(JSON.toJSONString(map));
            }
        });

        //(4)进入直播间topic2 线程three
        Thread threadThree = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device111");
                list1.add("device222");

                map.put("topic2", list1);

                System.out.println(JSON.toJSONString(map));
            }
        });

        //(5)启动线程
        threadOne.start();
        threadTwo.start();
        threadThree.start();
    }
}
```

- 代码（1）创建了一个并发 map 用来存放 topic 与其对应的设备列表
- 代码（2）（3）模拟用户进入直播间 topic1，代码（4）模拟用户进入直播间 topic2
- 代码（5）启动线程

运行代码输出：

```
{"topic1":["device11","device22"],"topic2":
["device111","device222"]}
{"topic1":["device11","device22"],"topic2":
["device111","device222"]}
```

```
{"topic1":["device11","device22"],"topic2":
["device111","device222"]}
```

或者

```
{"topic1":["device1","device2"],"topic2":
["device111","device222"]}
{"topic1":["device1","device2"],"topic2":
["device111","device222"]}
{"topic1":["device1","device2"],"topic2":
["device111","device222"]}
```

可知 topic1 房间中的用户会丢失一部分，这是因为 put 方法如果发现 map 里面存在这个 key, 则使用 value 覆盖该 key 对应的老的 value 值，而 putIfAbsent 方法则如果已经存在该 key 则返回该 key 对应的 value 并不进行覆盖，如果不存在则会新增该 key，并且判断和写入是原子性操作。使用 putIfAbsent 替代 put 方法后代码如下：

```
public class TestMap2 {
    //(1)创建map,key为topic,value为设备列表
    static ConcurrentHashMap<String, List<String>> map = new
ConcurrentHashMap<>();
    public static void main(String[] args) {
        //(2)进入直播间topic1 线程one
        Thread threadOne = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device1");
                list1.add("device2");
                //(2.1)
                List<String> oldList = map.putIfAbsent("topic1",
list1);
                if(null != oldList){
                    oldList.addAll(list1);
                }
                System.out.println(JSON.toJSONString(map));
            }
        });
        //(3)进入直播间topic1 线程two
        Thread threadTwo = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device11");
                list1.add("device22");

                List<String> oldList = map.putIfAbsent("topic1",
list1);
                if(null != oldList){
                    oldList.addAll(list1);
```

```
                }

                System.out.println(JSON.toJSONString(map));
            }
        });

        //(4)进入直播间topic2 线程three
        Thread threadThree = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device111");
                list1.add("device222");

                List<String> oldList = map.putIfAbsent("topic2",
list1);
                if(null != oldList){
                    oldList.addAll(list1);
                }
                System.out.println(JSON.toJSONString(map));
            }
        });

        //(5)启动线程
        threadOne.start();
        threadTwo.start();
        threadThree.start();
    }
}
```

如上代码（2.1）使用 map.putIfAbsent 方法添加新设备列表，如果 topic1 在 map 中不存在则放入 topic1 和对应设备列表到 map，要注意的是这个判断不存在和放入是原子性操作，这时候放入后会返回 null。如果 topic1 已经在 map 里面存在，则调用 putIfAbsent 会返回 topic1 对应的设备里面，代码发现返回的设备列表不为 null 则把新的设备列表添加到返回的设备列表里面，从而问题得到解决。

运行结果为：

```
{"topic1":["device1","device2","device11","device22"],"topic2":
["device111","device222"]}
{"topic1":["device1","device2","device11","device22"],"topic2":
["device111","device222"]}
{"topic1":["device1","device2","device11","device22"],"topic2":
["device111","device222"]}
```

总结：put(K key, V value) 方法如果 key 已经存在则使用 value 覆盖原来的值并返回原来的值，如果不存在则把 value 放入并返回 null。而 putIfAbsent(K key, V value) 方法如果 key 已经存在则直接返回原来对应的值并不使用 value 覆盖，如果 key 不存在则存入 value 并返回 null，另外要注意判断 key 不存在和存入是原子操作。

四、使用定时器 Timer 的时候需要注意的一些问题

4.1 问题产生

这里做了一个小的 demo 来复现问题，代码如下：

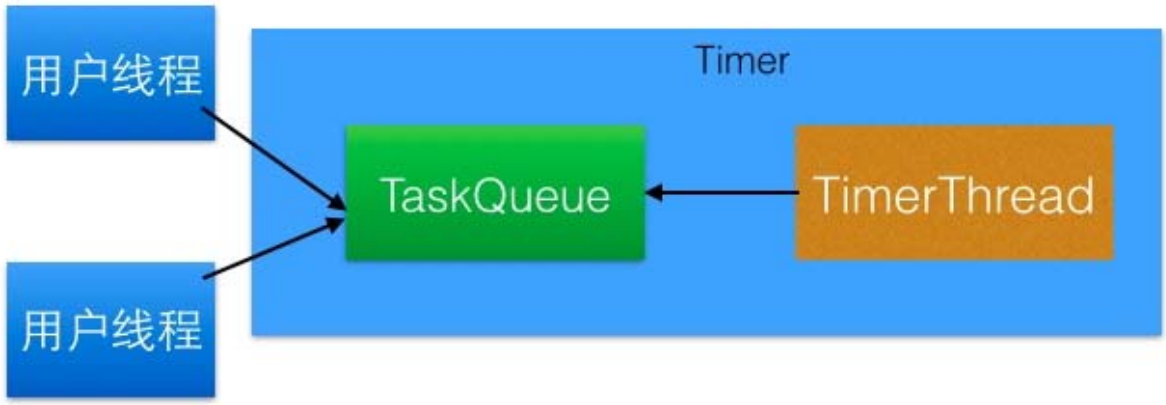
```
public class TestTimer {
    //创建定时器对象
     static Timer timer = new Timer();

     public static void main(String[] args) {
      //添加任务1,延迟500ms执行
         timer.schedule(new TimerTask() {

             @Override
             public void run() {
                 System.out.println("---one Task---");
                 try {
                     Thread.sleep(1000);
                 } catch (InterruptedException e) {
                     e.printStackTrace();
                 }
                 throw new RuntimeException("error ");
             }
         }, 500);
        //添加任务2，延迟1000ms执行
         timer.schedule(new TimerTask() {

             @Override
             public void run() {
                 for (;;) {
                     System.out.println("---two Task---");
                     try {
                         Thread.sleep(1000);
                     } catch (InterruptedException e) {
                         // TODO Auto-generated catch block
                         e.printStackTrace();
                     }
                 }
             }
         }, 1000);

     }
}
```

如上代码先添加了一个任务在 500ms 后执行，然后添加了第二个任务在 1s 后执行，我们期望的是当第一个任务输出 —one Task— 后等待 1s 后第二个任务会输出 —two Task—，但是执行完毕代码后输出结果为：

```
---one Task---
Exception in thread "Timer-0" java.lang.RuntimeException: error
    at com.zlx.Timer.TestTimer$1.run(TestTimer.java:22)
    at java.util.TimerThread.mainLoop(Timer.java:555)
    at java.util.TimerThread.run(Timer.java:505)
```

4.2 Timer 实现原理分析

下面简单介绍下 Timer 的原理，如下图是 Timer 的原理模型介绍：



- 其中 TaskQueue 是一个平衡二叉树堆实现的优先级队列，每个 Timer 对象内部有唯一一个 TaskQueue 队列。用户线程调用 timer 的 schedule 方法就是把 TimerTask 任务添加到 TaskQueue 队列，在调用 schedule 的方法时候 long delay 参数用来说明该任务延迟多少时间执行。
- TimerThread 是具体执行任务的线程，它从 TaskQueue 队列里面获取优先级最小的任务进行执行，需要注意的是只有执行完了当前的任务才会从队列里面获取下一个任务而不管队列里面是否有已经到了设置的 delay 时间，一个 Timer 只有一个 TimerThread 线程，所以可知 Timer 的内部实现是一个多生产者单消费者模型。

从实现模型可以知道要探究上面的问题只需看 TimerThread 的实现就可以了，TimerThread 的 run 方法主要逻辑代码如下：

```
public void run() {
    try {
        mainLoop();
    } finally {
        // Someone killed this Thread, behave as if Timer
cancelled
        synchronized(queue) {
            newTasksMayBeScheduled = false;
            queue.clear();  // Eliminate obsolete references
        }
    }
}
```

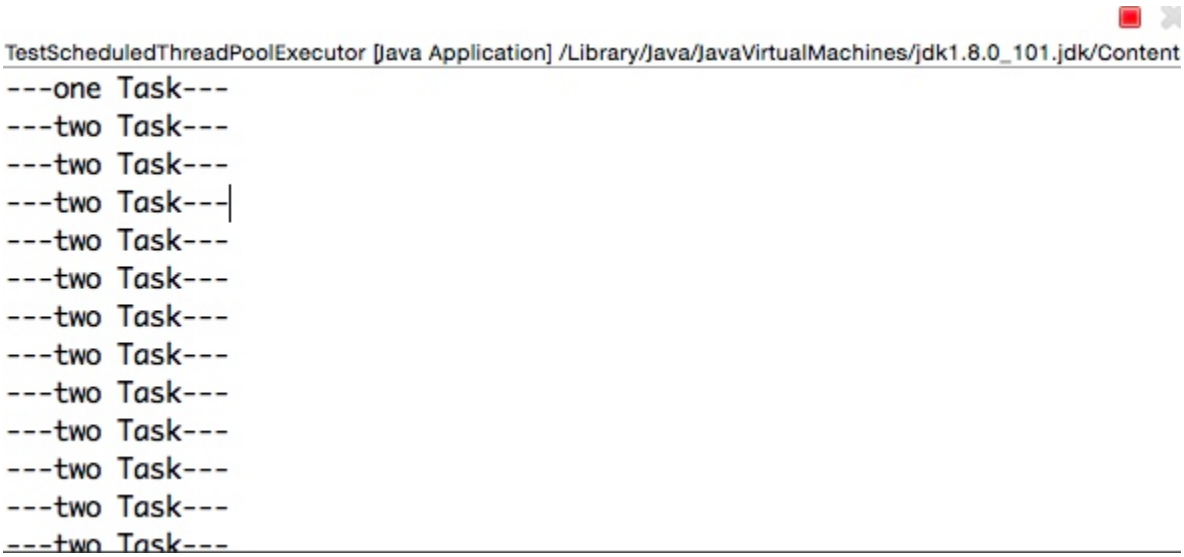
```
private void mainLoop() {
        while (true) {
            try {
                TimerTask task;
                boolean taskFired;
                //从队列里面获取任务时候要加锁
                synchronized(queue) {
                    ...
                }
                if (taskFired)
                    task.run();//执行任务
            } catch(InterruptedException e) {
            }
        }
    }
```

可知当任务执行过程中抛出了除 InterruptedException 之外的异常后，唯一的消费线程就会因为抛出异常而终止，那么队列里面的其他待执行的任务就会被清除。所以 TimerTask 的 run 方法内最好使用 try-catch 结构 catch 主可能的异常，不要把异常抛出到 run 方法外。其实要实现类似 Timer 的功能使用 ScheduledThreadPoolExecutor 的 schedule 是比较好的选择。ScheduledThreadPoolExecutor 中的一个任务抛出了异常，其他任务不受影响的。

```
public class TestScheduledThreadPoolExecutor {

    static ScheduledThreadPoolExecutor
scheduledThreadPoolExecutor = new ScheduledThreadPoolExecutor(1);

    public static void main(String[] args) {

        scheduledThreadPoolExecutor.schedule(new Runnable() {

            @Override
            public void run()  {
                System.out.println("---one Task---");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                throw new RuntimeException("error ");
            }

        }, 500, TimeUnit.MICROSECONDS);

        scheduledThreadPoolExecutor.schedule(new Runnable() {
```

```
                @Override
                public void run() {
                    for (int i =0;i<2;++i) {
                        System.out.println("---two Task---");
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }

                }

            }, 1000, TimeUnit.MICROSECONDS);

            scheduledThreadPoolExecutor.shutdown();
        }
    }
```

运行结果：



之所以 ScheduledThreadPoolExecutor 的其他任务不受抛出异常的任务的影响是因为 ScheduledThreadPoolExecutor 中的 ScheduledFutureTask 任务中 catch 掉了异常，但是在线程池任务的 run 方法内使用 catch 捕获异常并打印日志是最佳实践。

五、SimpleDateFormat 是线程不安全的

SimpleDateFormat 是 Java 提供的一个格式化和解析日期的工具类，日常开发中应该经常会用到，但是由于它是线程不安全的，多线程公用一个 SimpleDateFormat 实例对日期进行解析或者格式化会导致程序出错，本节就讨论下它为何是线程不安全的，以及如何避免。

5.1 问题复现

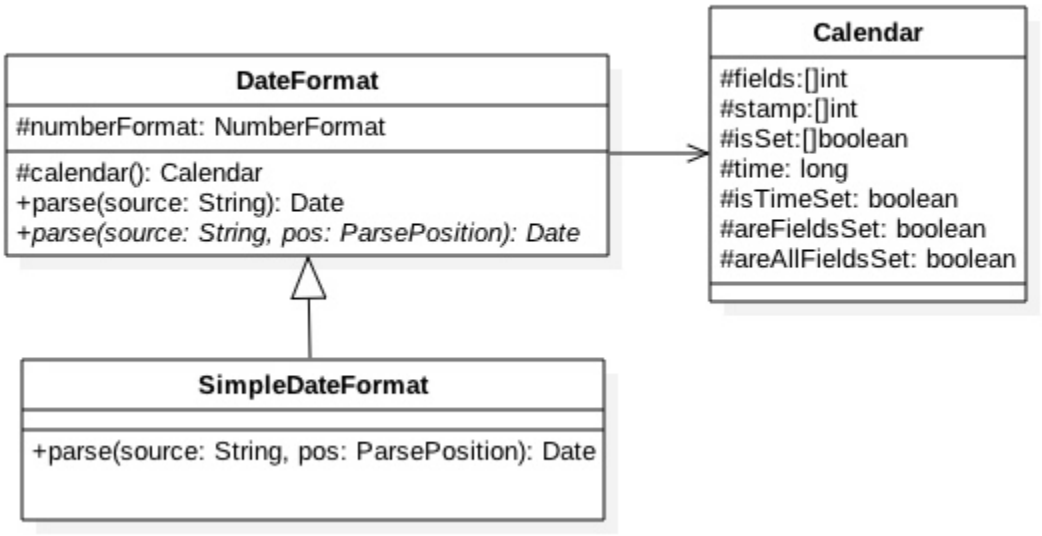
为了复现该问题，编写如下代码：

```
 public class TestSimpleDateFormat {
    //(1)创建单例实例
    static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");

    public static void main(String[] args) {
        //(2)创建多个线程，并启动
        for (int i = 0; i <10 ; ++i) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {//(3)使用单例日期实例解析文本
                        System.out.println(sdf.parse("2017-12-13
15:17:27"));
                    } catch (ParseException e) {
                        e.printStackTrace();
                    }
                }
            });
            thread.start();//(4)启动线程
        }
    }
}
```

代码（1）创建了 SimpleDateFormat 的一个实例，代码（2）创建 10 个线程，每个线程都公用同一个 sdf 对象对文本日期进行解析，多运行几次就会抛出 java.lang.NumberFormatException 异常，加大线程的个数有利于该问题复现。

5.2 问题分析

为了便于分析首先奉上 SimpleDateFormat 的类图结构：



可知每个 SimpleDateFormat 实例里面有一个 Calendar 对象，从后面会知道其实 SimpleDateFormat 之所以是线程不安全的就是因为 Calendar 是线程不安全的，后者之所以是线程不安全的是因为其中存放日期数据的变量都是线程不安全的，比如里面的 fields，time 等。

下面从代码层面看下 parse 方法做了什么事情：

```
public Date parse(String text, ParsePosition pos)
{

    //(1)解析日期字符串放入CalendarBuilder的实例calb中
    .....

    Date parsedDate;
    try {//（2）使用calb中解析好的日期数据设置calendar
        parsedDate = calb.establish(calendar).getTime();
        ...
    }

    catch (IllegalArgumentException e) {
       ...
        return null;
    }

    return parsedDate;
}
```

```
Calendar establish(Calendar cal) {
   ...
   //（3）重置日期对象cal的属性值
   cal.clear();
   //(4) 使用calb中中属性设置cal
   ...
```

```
    //(5)返回设置好的cal对象
    return cal;
}
```

- 代码（1）主要的作用是解析字符串日期并把解析好的数据放入了 CalendarBuilder 的实例 calb 中，CalendarBuilder 是一个建造者模式，用来存放后面需要的数据。
- 代码（3）重置 Calendar 对象里面的属性值，如下代码：

```
 public final void clear()
{
    for (int i = 0; i < fields.length; ) {
        stamp[i] = fields[i] = 0; // UNSET == 0
        isSet[i++] = false;
    }
    areAllFieldsSet = areFieldsSet = false;
    isTimeSet = false;
}
```

- 代码（4）使用 calb 中解析好的日期数据设置 cal 对象
- 代码（5） 返回设置好的 cal 对象

从上面步骤可知步骤（3） （4） （5）操作不是原子性操作，当多个线程调用 parse 方法时候比如线程 A 执行了步骤（3） （4）也就是设置好了 cal 对象，在执行步骤（5）前线程 B 执行了步骤（3）清空了 cal 对象，由于多个线程使用的是一个 cal 对象，所以线程 A 执行步骤（5）返回的就可能是被线程 B 清空后的对象，当然也有可能线程 B 执行了步骤（4）被线程 B 修改后的 cal 对象。从而导致程序错误。

那么怎么解决那?

- 第一种方式：每次使用时候 new 一个 SimpleDateFormat 的实例，这样可以保证每个实例使用自己的 Calendar 实例, 但是每次使用都需要 new 一个对象，并且使用后由于没有其它引用，就会需要被回收，开销会很大。
- 第二种方式：究其原因是因为多线程下步骤（3） （4） （5）三个步骤不是一个原子性操作，那么容易想到的是对其进行同步，让（3） （4） （5）成为原子操作，可以使用 synchronized 进行同步，具体如下：

```
public class TestSimpleDateFormat {
    // (1)创建单例实例
    static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");

    public static void main(String[] args) {
        // (2)创建多个线程，并启动
        for (int i = 0; i < 10; ++i) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
```

```
                    try {// (3)使用单例日期实例解析文本
                        synchronized (sdf) {
                            System.out.println(sdf.parse("2017-
12-13 15:17:27"));
                        }
                    } catch (ParseException e) {
                        e.printStackTrace();
                    }
                }
            });
            thread.start();// (4)启动线程
        }
    }
}
```

使用同步意味着多个线程要竞争锁，在高并发场景下会导致系统响应性能下降。

- 第三种方式：使用 ThreadLocal，这样每个线程只需要使用一个 SimpleDateFormat 实例相比第一种方式大大节省了对象的创建销毁开销，并且不需要对多个线程直接进行同步，使用 ThreadLocal 方式代码如下：

```
public class TestSimpleDateFormat2 {
    // (1)创建threadlocal实例
    static ThreadLocal<DateFormat> safeSdf = new
ThreadLocal<DateFormat>(){
        @Override
        protected SimpleDateFormat initialValue(){
            return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        }
    };

    public static void main(String[] args) {
        // (2)创建多个线程，并启动
        for (int i = 0; i < 10; ++i) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {// (3)使用单例日期实例解析文本

System.out.println(safeSdf.get().parse("2017-12-13 15:17:27"));
                    } catch (ParseException e) {
                        e.printStackTrace();
                    }finally {
                        //(4)使用完毕记得清除，避免内存泄露
                        safeSdf.remove();
                    }
                }
            });
            thread.start();// (4)启动线程
        }
```

```
        }
    }
```

代码（1）创建了一个线程安全的 SimpleDateFormat 实例，步骤（3）在使用的时候首先使用 get() 方法获取当前线程下 SimpleDateFormat 的实例，在第一次调用 ThreadLocal 的 get（）方法适合会触发其 initialValue 方法用来创建当前线程所需要的 SimpleDateFormat 对象。另外需要注意的是代码（4）使用完毕线程变量后要记得进行清理，以避免内存泄露。

六、线程池使用 FutureTask 时候需要注意的一点事

线程池使用 FutureTask 的时候如果拒绝策略设置为了 `DiscardPolicy` 和 `DiscardOldestPolicy` 并且在被拒绝的任务的 Future 对象上调用无参 get 方法那么调用线程会一直被阻塞。

6.1 问题复现

下面先通过一个简单的例子来复现问题：

```
public class FutureTest {

    //(1)线程池单个线程，线程池队列元素个数为1
        private final static ThreadPoolExecutor executorService =
new ThreadPoolExecutor(1, 1, 1L, TimeUnit.MINUTES,
            new ArrayBlockingQueue<Runnable>(1),new
ThreadPoolExecutor.DiscardPolicy());

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

        //(2)添加任务one
        Future futureOne = executorService.submit(new Runnable()
{
            @Override
            public void run() {

                System.out.println("start runable one");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        //(3)添加任务two
```

```
        Future futureTwo = executorService.submit(new Runnable()
{
            @Override
            public void run() {
                System.out.println("start runable two");
            }
        });

        //(4)添加任务three
        Future futureThree=null;
        try {
            futureThree = executorService.submit(new Runnable() {
                @Override
                public void run() {
                    System.out.println("start runable three");
                }
            });
        } catch (Exception e) {
            System.out.println(e.getLocalizedMessage());
        }

        System.out.println("task one " + futureOne.get());//(5)等
待任务one执行完毕
        System.out.println("task two " + futureTwo.get());//(6)等
待任务two执行完毕
        System.out.println("task three " + (futureThree==null?
null:futureThree.get()));// (7)等待任务three执行完毕

        executorService.shutdown();//(8)关闭线程池，阻塞直到所有任务执
行完毕
    }
```

运行代码结果为：

```
FutureTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年12月12日 下午11:00:32)
start runable one
task one null
start runable two
task two null
```

- 代码 (1) 创建了一个单线程并且队列元素个数为 1 的线程池，并且拒绝策略设置为了 DiscardPolicy
- 代码（2）向线程池提交了一个任务 one，那么这个任务会使用唯一的一个线程进行执行，任务在打印 start runable one 后会阻塞该线程 5s.
- 代码（3）向线程池提交了一个任务 two，这时候会把任务 two 放入到阻塞队列
- 代码（4）向线程池提交任务 three，由于队列已经满了则会触发拒绝策略丢弃任务 three, 从执行结果看在任务 one 阻塞的 5s 内，主线程执行到了代码 (5) 等待任务 one 执行完毕，当任务 one 执行完毕后代码（5）返回，主线程打印出 task one

null。任务 one 执行完成后线程池的唯一线程会去队列里面取出任务 two 并执行所以输出 start runable two 然后代码（6）会返回，这时候主线程输出 task two null，然后执行代码（7）等待任务 three 执行完毕，从执行结果看代码（7）会一直阻塞不会返回，至此问题产生，如果把拒绝策略修改为 DiscardOldestPolicy 也会存在有一个任务的 get 方法一直阻塞只是现在是任务 two 被阻塞。但是如果拒绝策略设置为默认的 AbortPolicy 则会正常返回，并且会输出如下结果：

```
start runable one
Task java.util.concurrent.FutureTask@135fbaa4 rejected from
java.util.concurrent.ThreadPoolExecutor@45ee12a7[Running, pool
size = 1, active threads = 1, queued tasks = 1, completed tasks =
0]
task one null
start runable two
task two null
task three null
```

6.2 问题分析

要分析这个问题需要看下线程池的 submit 方法里面做了什么，submit 方法代码如下：

```
    public Future<?> submit(Runnable task) {
        ...
        //（1）装饰Runnable为Future对象
        RunnableFuture<Void> ftask = newTaskFor(task, null);
        execute(ftask);
        //(6)返回future对象
        return ftask;
    }

        protected <T> RunnableFuture<T> newTaskFor(Runnable
runnable, T value) {
        return new FutureTask<T>(runnable, value);
    }
```

```
 public void execute(Runnable command) {
         ...
        //(2) 如果线程个数消息核心线程数则新增处理线程处理
        int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        //（3）如果当前线程个数已经达到核心线程数则任务放入队列
        if (isRunning(c) && workQueue.offer(command)) {
```

```
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        //（4）尝试新增处理线程进行处理
        else if (!addWorker(command, false))
            reject(command);//(5)新增失败则调用拒绝策略
    }
```

根据代码可以总结如下：

- 代码（1）装饰 Runnable 为 FutureTask 对象，然后调用线程池的 execute 方法
- 代码 (2) 如果线程个数消息核心线程数则新增处理线程处理
- 代码（3）如果当前线程个数已经达到核心线程数则任务放入队列
- 代码（4）尝试新增处理线程进行处理，失败则进行代码（5），否者直接使用新线程处理
- 代码（5）执行具体拒绝策略，从这里也可以看出拒绝策略执行是使用的业务线程。

所以要分析上面例子中问题所在只需要看步骤（5）对被拒绝任务的影响，这里先看下拒绝策略 DiscardPolicy 的代码：

```
    public static class DiscardPolicy implements
RejectedExecutionHandler {
        public DiscardPolicy() { }
        public void rejectedExecution(Runnable r,
ThreadPoolExecutor e) {
        }
    }
```

可知拒绝策略 rejectedExecution 方法里面什么都没做，所以代码（4）调用 submit 后会返回一个 future 对象，这里有必要在重新说 future 是有状态的，future 的状态枚举值如下：

```
    private static final int NEW          = 0;
    private static final int COMPLETING   = 1;
    private static final int NORMAL       = 2;
    private static final int EXCEPTIONAL  = 3;
    private static final int CANCELLED    = 4;
    private static final int INTERRUPTING = 5;
    private static final int INTERRUPTED  = 6;
```

在步骤（1）的时候使用 newTaskFor 方法转换 Runnable 任务为 FutureTask，而 FutureTask 的构造函数里面设置的状态就是 New。

```
    public FutureTask(Runnable runnable, V result) {
        this.callable = Executors.callable(runnable, result);
        this.state = NEW;       // ensure visibility of callable
    }
```

所以使用 DiscardPolicy 策略提交后返回了一个状态为 NEW 的 future 对象。

那么我们下面就需要看下当调用 future 的无参 get 方法时候当 future 变为什么状态时候才会返回那, 那就需看下 FutureTask 的 get（）方法代码：

```
    public V get() throws InterruptedException,
ExecutionException {
        int s = state;
        //当状态值<=COMPLETING时候需要等待，否者调用report返回
        if (s <= COMPLETING)
            s = awaitDone(false, 0L);
        return report(s);
    }

   private V report(int s) throws ExecutionException {
   Object x = outcome;
   //状态值为NORMAL正常返回
   if (s == NORMAL)
       return (V)x;
   //状态值大于等于CANCELLED则抛异常
   if (s >= CANCELLED)
       throw new CancellationException();
   throw new ExecutionException((Throwable)x);
}
```

也就是说当 future 的状态 > COMPLETING 时候调用 get 方法才会返回，而明显 DiscardPolicy 策略在拒绝元素的时候并没有设置该 future 的状态，后面也没有其他机会可以设置该 future 的状态，所以 future 的状态一直是 NEW，所以一直不会返回，同理 DiscardOldestPolicy 策略也是这样的问题，最老的任务被淘汰时候没有设置被淘汰任务对于 future 的状态。

那么默认的 AbortPolicy 策略为啥没问题那？其实 AbortPolicy 策略时候步骤（5）直接会抛出 RejectedExecutionException 异常，也就是 submit 方法并没有返回 future 对象，这时候 futureThree 是 null。

所以当使用 Future 的时候，尽量使用带超时时间的 get 方法，这样即使使用了 DiscardPolicy 拒绝策略也不至于一直等待，等待超时时间到了会自动返回的，如果非要使用不带参数的 get 方法则可以重写 DiscardPolicy 的拒绝策略在执行策略时候设置该 Future 的状态大于 COMPLETING 即可，但是查看 FutureTask 提供的方法发现只有 cancel 方法是 public 的并且可以设置 FutureTask 的状态大于 COMPLETING，重写拒绝策略具体代码可以如下：

```
public class MyRejectedExecutionHandler implements
RejectedExecutionHandler{

    @Override
    public void rejectedExecution(Runnable runable,
ThreadPoolExecutor e) {
         if (!e.isShutdown()) {
             if(null != runable && runable instanceof FutureTask)
{
                 ((FutureTask) runable).cancel(true);
             }
         }
    }

}
```

使用这个策略时候由于从 report 方法知道在 cancel 的任务上调用 get() 方法会抛出异常所以代码（7）需要使用 try-catch 捕获异常代码（7）修改为如下：

```
         try{
             System.out.println("task three " +
(futureThree==null?null:futureThree.get()));// (6)等待任务three
         }catch(Exception e){
             System.out.println(e.getLocalizedMessage());
         }
```

执行结果为：

```
<terminated> FutureTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年12月13日 上午9:47:30)
start runable one
task one null
start runable two
task two null
null
```

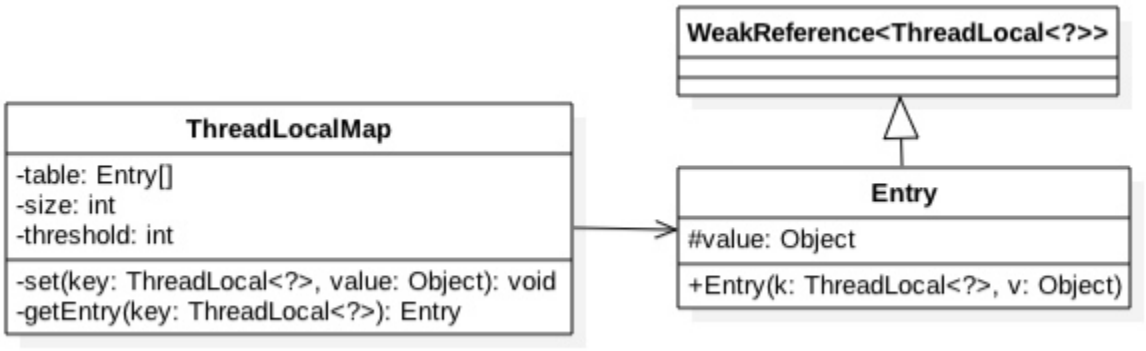
当然这相比正常情况下多了一个异常捕获，其实最好的情况是重写拒绝策略时候设置 FutureTask 的状态为 NORMAL，但是这需要重写 FutureTask 方法了，因为 FutureTask 并没有提供接口进行设置。

七、使用 ThreadLocal 不当可能会导致内存泄露

基础篇已经讲解了 ThreadLocal 的原理，本节着重来讲解下使用 ThreadLocal 会导致内存泄露的原因，并讲解使用 ThreadLocal 导致内存泄露的案例。

7.1 为何会出现内存泄露

基础篇我们讲到了 ThreadLocal 只是一个工具类，具体存放变量的是在线程的 threadLocals 变量里面，threadLocals 是一个 ThreadLocalMap 类型的，



如上图 ThreadLocalMap 内部是一个 Entry 数组, Entry 继承自 WeakReference，Entry 内部的 value 用来存放通过 ThreadLocal 的 set 方法传递的值，那么 ThreadLocal 对象本身存放到哪里了吗? 下面看看 Entry 的构造函数：

```
 Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
 }

public WeakReference(T referent) {
   super(referent);
}

Reference(T referent) {
   this(referent, null);
}

Reference(T referent, ReferenceQueue<? super T> queue) {

   this.referent = referent;
   this.queue = (queue == null) ? ReferenceQueue.NULL : queue;
}
```

可知 k 被传递到了 WeakReference 的构造函数里面，也就是说 ThreadLocalMap 里面的 key 为 ThreadLocal 对象的弱引用，具体是 referent 变量引用了 ThreadLocal 对象，value 为具体调用 ThreadLocal 的 set 方法传递的值。

当一个线程调用 ThreadLocal 的 set 方法设置变量时候，当前线程的 ThreadLocalMap 里面就会存放一个记录，这个记录的 key 为 ThreadLocal 的引用，value 则为设置的值。

但是考虑如果这个 ThreadLocal 变量没有了其他强依赖，而当前线程还存在的情况下，由于线程的 ThreadLocalMap 里面的 key 是弱依赖，则当前线程的 ThreadLocalMap 里面的 ThreadLocal 变量的弱引用会被在 gc 的时候回收，但是对应 value 还是会造成内存泄露，这时候 ThreadLocalMap 里面就会存在 key 为 null 但是 value 不为 null 的 entry 项。

其实在 ThreadLocal 的 set 和 get 和 remove 方法里面有一些时机是会对这些 key 为 null 的 entry 进行清理的，但是这些清理不是必须发生的，下面简单说下 ThreadLocalMap 的 remove 方法的清理过程：

```
private void remove(ThreadLocal<?> key) {

  //(1)计算当前ThreadLocal变量所在table数组位置，尝试使用快速定位方法
  Entry[] tab = table;
  int len = tab.length;
  int i = key.threadLocalHashCode & (len-1);
  //(2)这里使用循环是防止快速定位失效后，变量table数组
  for (Entry e = tab[i];
       e != null;
       e = tab[i = nextIndex(i, len)]) {
      //(3)找到
      if (e.get() == key) {
          //(4)找到则调用WeakReference的clear方法清除对ThreadLocal的
弱引用
          e.clear();
          //(5)清理key为null的元素
          expungeStaleEntry(i);
          return;
      }
  }
}
```

```
 private int expungeStaleEntry(int staleSlot) {
            Entry[] tab = table;
            int len = tab.length;

            //（6）去掉去value的引用
            tab[staleSlot].value = null;
            tab[staleSlot] = null;
            size--;

            Entry e;
            int i;
            for (i = nextIndex(staleSlot, len);
                 (e = tab[i]) != null;
                 i = nextIndex(i, len)) {
                ThreadLocal<?> k = e.get();

                //(7)如果key为null,则去掉对value的引用。
                if (k == null) {
                    e.value = null;
                    tab[i] = null;
                    size--;
                } else {
```

```
                    int h = k.threadLocalHashCode & (len - 1);
                    if (h != i) {
                        tab[i] = null;
                        while (tab[h] != null)
                            h = nextIndex(h, len);
                        tab[h] = e;
                    }
                }
            }
            return i;
        }
```

- 步骤（4）调用了 Entry 的 clear 方法，实际调用的是父类 WeakReference 的 clear 方法，作用是去掉对 ThreadLocal 的弱引用。
- 步骤（6）是去掉对 value 的引用，到这里当前线程里面的当前 ThreadLocal 对象的信息被清理完毕了。
- 代码（7）从当前元素的下标开始看 table 数组里面的其他元素是否有 key 为 null 的，有则清理。循环退出的条件是遇到 table 里面有 null 的元素。所以这里知道 null 元素后面的 Entry 里面 key 为 null 的元素不会被清理。

注：ThreadLocalMap 内部 Entry 中 key 使用的是对 ThreadLocal 对象的弱引用，这为避免内存泄露是一个进步，因为如果是强引用，那么即使其他地方没有对 ThreadLocal 对象的引用，ThreadLocalMap 中的 ThreadLocal 对象还是不会被回收，而如果是弱引用则这时候 ThreadLocal 引用是会被回收掉的。

但是对于的 value 还是不能被回收，这时候 ThreadLocalMap 里面就会存在 key 为 null 但是 value 不为 null 的 entry 项，虽然 ThreadLocalMap 提供了 set,get,remove 方法在一些时机下会对这些 Entry 项进行清理，但是这是不及时的，也不是每次都会执行的，所以一些情况下还是会发生内存泄露，所以在使用完毕后即使调用 remove 方法才是解决内存泄露的王道。

7.2 线程池中使用 ThreadLocal 导致的内存泄露

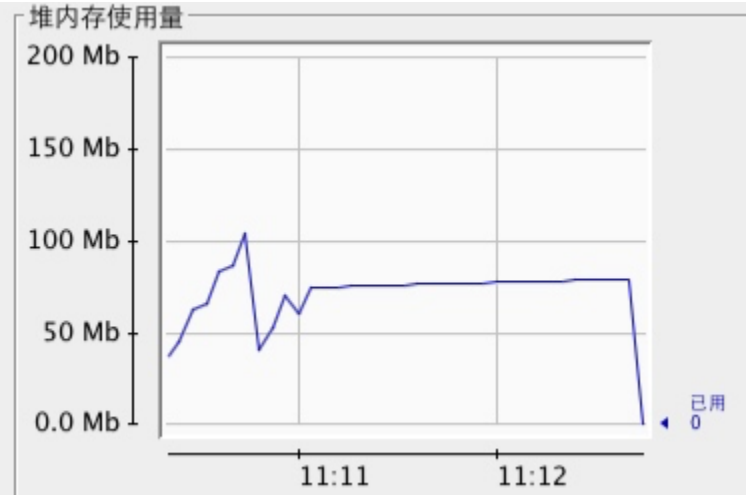
下面先看线程池中使用 ThreadLocal 的例子：

```
public class ThreadPoolTest {

    static class LocalVariable {
        private Long[] a = new Long[1024*1024];
    }

    // (1)
    final static ThreadPoolExecutor poolExecutor = new
ThreadPoolExecutor(5, 5, 1, TimeUnit.MINUTES,
            new LinkedBlockingQueue<>());
    // (2)
    final static ThreadLocal<LocalVariable> localVariable = new
```

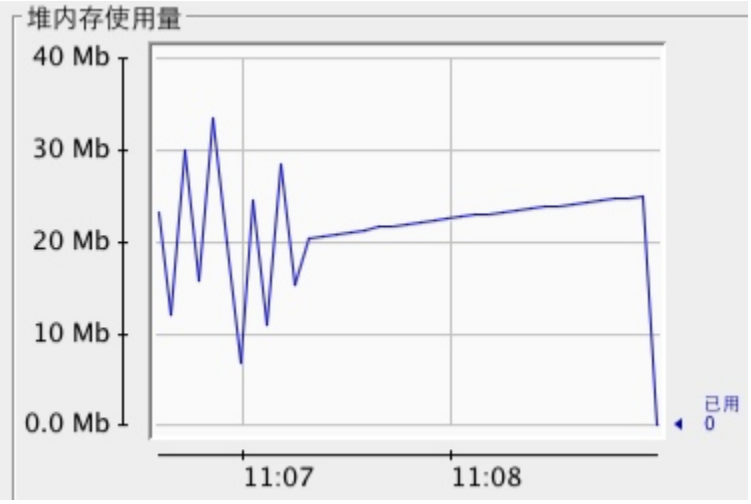
```
    ThreadLocal<LocalVariable>();

        public static void main(String[] args) throws
    InterruptedException {
            // (3)
            for (int i = 0; i < 50; ++i) {
                poolExecutor.execute(new Runnable() {
                    public void run() {
                        // (4)
                        localVariable.set(new LocalVariable());
                        // (5)
                        System.out.println("use local varaible");
                        //localVariable.remove();

                    }
                });

                Thread.sleep(1000);
            }
            // (6)
            System.out.println("pool execute over");
        }
```

- 代码（1）创建了一个核心线程数和最大线程数为 5 的线程池，这个保证了线程池里面随时都有 5 个线程在运行。
- 代码（2）创建了一个 ThreadLocal 的变量，泛型参数为 LocalVariable，LocalVariable 内部是一个 Long 数组。
- 代码（3）向线程池里面放入 50 个任务
- 代码（4）设置当前线程的 localVariable 变量，也就是把 new 的 LocalVariable 变量放入当前线程的 threadLocals 变量。
- 由于没有调用线程池的 shutdown 或者 shutdownNow 方法所以线程池里面的用户线程不会退出，进而 JVM 进程也不会退出。

运行当前代码，使用 jconsole 监控堆内存变化如下图：



然后解开 localVariable.remove() 注释，然后在运行，观察堆内存变化如下:



从运行结果一可知，当主线程处于休眠时候进程占用了大概 77M 内存，运行结果二则占用了大概 25M 内存，可知运行代码一时候内存发生了泄露，下面分析下泄露的原因。

运行结果一的代码，在设置线程的 localVariable 变量后没有调用 `localVariable.remove()` 方法，导致线程池里面的 5 个线程的 threadLocals 变量里面的 `new LocalVariable()` 实例没有被释放，虽然线程池里面的任务执行完毕了，但是线程池里面的 5 个线程会一直存在直到 JVM 退出。这里需要注意的是由于 localVariable 被声明了 static，虽然线程的 ThreadLocalMap 里面是对 localVariable 的弱引用，localVariable 也不会被回收。运行结果二的代码由于线程在设置 localVariable 变量后即使调用了 `localVariable.remove()` 方法进行了清理，所以不会存在内存泄露。

总结：线程池里面设置了 ThreadLocal 变量一定要记得及时清理，因为线程池里面的核心线程是一直存在的，如果不清理，那么线程池的核心线程的 threadLocals 变量一直会持有 ThreadLocal 变量。

八、总结

本文针对并发编程中经常遇到的一些问题进行讲解，希望读者能慢慢体会，以免在生产环境遇到坑，实际在并发编程中还会遇到其他的问题，限于篇幅本文没有提起，更多实战问题敬请期待《并发编程之美》一书的出版，这本书最后一章专门讲并发编程实战。