```
                }
                if (taskFired)
                    task.run();//执行任务
            } catch(InterruptedException e) {
            }
        }
    }
```

当任务在执行过程中抛出 InterruptedException 之外的异常时，唯一的消费线程就会因为抛出异常而终止，那么队列里的其他待执行的任务就会被清除。所以在 TimerTask 的 run 方法内最好使用 try-catch 结构捕捉可能的异常，不要把异常抛到 run 方法之外。其实要实现 Timer 功能，使用 ScheduledThreadPoolExecutor 的 schedule 是比较好的选择。如果 ScheduledThreadPoolExecutor 中的一个任务抛出异常，其他任务则不受影响。

```
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---");
```

仅供非商业用途或交流学习使用

```
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

        }

    }, 1000, TimeUnit.MICROSECONDS);

    scheduledThreadPoolExecutor.shutdown();
  }
}
```

运行结果如下。

```
TestScheduledThreadPoolExecutor [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Cont
---one Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
---two Task---
```

之所以 ScheduledThreadPoolExecutor 的其他任务不受抛出异常的任务的影响，是因为在 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。

11.5.3　小结

ScheduledThreadPoolExecutor 是并发包提供的组件，其提供的功能包含但不限于 Timer。Timer 是固定的多线程生产单线程消费，但是 ScheduledThreadPoolExecutor 是可以配置的，既可以是多线程生产单线程消费也可以是多线程生产多线程消费，所以在日常开发中使用定时器功能时应该优先使用 ScheduledThreadPoolExecutor。

仅供非商业用途或交流学习使用

11.6 对需要复用但是会被下游修改的参数要进行深复制

11.6.1 问题的产生

本节通过一个简单的消息发送 demo 来讲解。首先介绍消息发送的场景，比如每个安装有手淘 App 的移动设备有一个设备 ID，每个 App（比如手淘 App）有一个 appkey 用来标识这个应用。可以根据不同的 appkey 选择不同的发送策略，对注册到自己的设备进行消息发送，每个消息有一个消息 ID 和消息体字段。下面首先贴出实例代码，如下所示。

```
    // (1)不同appkey注册不同的服务
    static Map<Integer, StrategyService> serviceMap = new HashMap<Integer,
StrategyService>();
    static {
        serviceMap.put(111, new StrategyOneService());
        serviceMap.put(222, new StrategyTwoService());
    }

    public static void main(String[] args) {

        // (2)key为appkey,value为设备ID列表
        Map<Integer, List<String>> appKeyMap = new HashMap<Integer, List<String>>();

        // (3)创建appkey=111的设备列表
        List<String> oneList = new ArrayList<>();
        oneList.add("device_id1");
        appKeyMap.put(111, oneList);

        // 创建appkey=222的设备列表
        List<String> twoList = new ArrayList<>();
        twoList.add("device_id2");
        appKeyMap.put(222, twoList);

        // (4)创建消息
        List<Msg> msgList = new ArrayList<>();
        Msg msg = new Msg();
        msg.setDataId("abc");
        msg.setBody("hello");
        msgList.add(msg);

        // (5)根据不同的appKey使用不同的策略进行处理
        appKeyItr = appKeyMap.keySet().iterator();
```

仅供非商业用途或交流学习使用

```
        while (appKeyItr.hasNext()) {
            int appKey = appKeyItr.next();
            // 这里根据appkey获取自己的消息列表
            StrategyService strategyService = serviceMap.get(appKey);
            if(null != strategyService){
                strategyService.sendMsg(msgList, appKeyMap.get(appKey));
            }else{
                System.out.println(String.format("appkey:%s, is not registerd
                  service", appKey));
            }
        }

    }
}
```

代码（1）给不同的 appkey 注册对应的处理策略，appkey=111 和 appkey=222 时分别注册了 StrategyOneService 和 StrategyTwoService 服务，它们都实现了 StrategyService 接口，具体代码如下。

```
public interface StrategyService {

    public void sendMsg(List<Msg> msgList,List<String> deviceIdList);
}
public class StrategyOneService implements StrategyService {

    @Override
    public void sendMsg(List<Msg> msgList, List<String> deviceIdList) {
        for (Msg msg : msgList) {
            msg.setDataId("oneService_" + msg.getDataId());
            System.out.println(msg.getDataId() + " " + JSON.
toJSONString(deviceIdList));
        }
    }
}
public class StrategyTwoService implements StrategyService {

    @Override
    public void sendMsg(List<Msg> msgList, List<String> deviceIdList) {

        for (Msg msg : msgList) {
            msg.setDataId("TwoService_" + msg.getDataId());
```

仅供非商业用途或交流学习使用

```
            System.out.println(msg.getDataId() + " " + JSON.
toJSONString(deviceIdList));
        }
    }
}
```

每个消息对应一个 DataId，其用来唯一标识一个消息。在每个发送消息的实现里面都会添加一个前缀以用于分类统计。

代码（2）和代码（3）则是给对应的 appkey 新增设备列表。

代码（4）创建消息体。

代码（5）实现根据不同的 appkey 使用不同的发送策略进行消息发送。

运行上面代码，我们期望的输出结果为

```
TwoService_abc ["device_id2"]
oneService_abc ["device_id1"]
```

但是实际结果却是

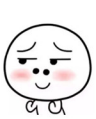
```
TwoService_abc ["device_id2"]
oneService_TwoService_abc ["device_id1"]
```

问题产生了。这个例子运行的结果是固定的，但是如果在每个发送消息的 sendMsg 方法里面异步修改消息的 DataId，那么运行的结果就不是固定的了。

11.6.2 问题分析

分析输出结果可以知道，代码（5）先执行了 appkey=222 的发送消息服务，然后再执行 appkey=111 的服务，之所以后者打印出来的 DataId 是 oneService_TwoService 而不是 oneService，是因为在 appkey=222 的消息服务里面修改了消息体 msg 的 DataId 为 TwoService_abc，而方法 sendMsg 里面的消息是引用传递的，所以导致 appkey=111 的服务在调用 sendMsg 方法时 msg 里面的 DataId 已经变成了 TwoService_abc，然后在 sendMsg 方法内部又会在它的前面添加 oneService 前缀，最后 DataId 就变成了 oneService_TwoService_abc。

那么该问题如何解决呢？首先应该想到的是不同的 appkey 应该有自己的一份 List<Msg>，这样不同的服务只会修改自己的消息的 DataId 而不会相互影响。那么下面修



仅供非商业用途或交流学习使用

改代码（5）中的部分代码如下。

```
        serviceMap.get(appKey).sendMsg(new ArrayList<Msg>(msgList), appKeyMap.
get(appKey));
```

也就是在具体发送消息前重新 new 一个消息列表传递过去，这样应该可以了吧？其实这还是不行的，因为如果 appkey 的个数大于 1，那么在第二个 appkey 服务发送时 ArrayList 构造函数里面的 msgList 已经是第一个 appkey 的服务修改后的了。那么自然会想到应该在代码（5）前面给每个 appkey 事先准备好自己的消息列表，那么新增和修改代码（5）如下。

```
//这里给每个appkey准备自己的消息列表
Iterator<Integer> appKeyItr = appKeyMap.keySet().iterator();
Map<Integer,List<Msg>> appKeyMsgMap = new HashMap<Integer, List<Msg>>();
while(appKeyItr.hasNext()){
    appKeyMsgMap.put(appKeyItr.next(), new ArrayList<>(msgList));
}

// (5)根据不同的appKey使用不同的策略进行处理
appKeyItr = appKeyMap.keySet().iterator();
while (appKeyItr.hasNext()) {
    int appKey = appKeyItr.next();
    // 这里根据appkey获取自己的消息列表
    StrategyService strategyService = serviceMap.get(appKey);
    if(null != strategyService){
        strategyService.sendMsg(appKeyMsgMap.get(appKey), appKeyMap.

        get(appKey));
    }else{
        System.out.println(String.format("appkey:%s, is not registerd

         service", appKey));
    }
}
```

如上代码首先给每个 appkey 创建消息列表，然后放入 appKeyMsgMap。之后在代码（5）具体发送消息时根据 appkey 去获取相应的消息列表，这样应该没问题了吧？但是当你信心满满地执行并查看结果时就傻眼了，结果竟然和之前的一样。

那么问题出在哪里呢？给每个 appkey 搞一份消息列表，然后发送时使用自己的消息列表进行发送，这个策略是没问题的，那么只有一个情况，就是给每个 appkey 创建一份消息列表时出错了，所有 appkey 用的还是同一份列表。难道 new ArrayList<>(msgList) 里

仅供非商业用途或交流学习使用

面还是引用？其实确实是，因为 Msg 本身是引用类型，而 new ArrayList<>(msgList) 这种方式是浅复制，每个 appkey 消息列表都是对同一个 Msg 的引用，修改代码如下。

```
// 这里给每个appkey准备一个消息列表
Iterator<Integer> appKeyItr = appKeyMap.keySet().iterator();
Map<Integer, List<Msg>> appKeyMsgMap = new HashMap<Integer, List<Msg>>();
while (appKeyItr.hasNext()) {

    //复制每个消息到临时消息列表
    List<Msg> tempList = new ArrayList<Msg>();
    Iterator<Msg> itrMsg = msgList.iterator();
    while (itrMsg.hasNext()) {

        Msg tmpMsg = null;
        try {
           //使用BeanUtils.cloneBean对Msg对象进行属性复制
            tmpMsg = (Msg) BeanUtils.cloneBean(itrMsg.next());
        } catch (Exception e) {
            e.printStackTrace();
        }
        if (null != tmpMsg) {
            tempList.add(tmpMsg);

        }
    }
    //存放当前appkey对应的经过深复制的消息列表
    appKeyMsgMap.put(appKeyItr.next(), tempList);
}
```

如上代码使用工具类 BeanUtils.cloneBean 而不是 new ArrayList<>(msgList) 来构造每个 appkey 对应的消息列表，修改后运行结果如下。

```
TwoService_abc ["device_id2"]
oneService_abc ["device_id1"]
```

至此问题得到解决。

11.6.3 小结

本节通过一个简单的消息发送例子说明了需要复用但是会被下游修改的参数要进行深复制，否则会导致出现错误的结果；另外引用类型作为集合元素时，如果使用这个集合作为另外一个集合的构造函数参数，会导致两个集合里面的同一个位置的元素指向的是同一

个引用，这会导致对引用的修改在两个集合中都可见，所以这时候需要对引用元素进行深复制。

11.7　创建线程和线程池时要指定与业务相关的名称

在日常开发中，当在一个应用中需要创建多个线程或者线程池时最好给每个线程或者线程池根据业务类型设置具体的名称，以便在出现问题时方便进行定位。下面就通过实例来说明不设置为何难以定位问题，以及如何进行设置。

11.7.1　创建线程需要有线程名

下面通过简单的代码来说明不指定线程名称为何难定位问题，代码如下。

```
public static void main(String[] args) {
   //订单模块
    Thread threadOne = new Thread(new Runnable() {
        public void run() {
            System.out.println("保存订单的线程");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            throw new NullPointerException();
        }
    });
 //发货模块
    Thread threadTwo = new Thread(new Runnable() {
        public void run() {
            System.out.println("保存收获地址的线程");
        }
    });

    threadOne.start();
    threadTwo.start();

}
```

如上代码分别创建了线程 one 和线程 two，运行上面的代码，输出如下。

```
保存订单的线程
保存收获地址的线程
Exception in thread "Thread-0" java.lang.NullPointerException
        at com.zlx.thread.ThreadName$1.run(ThreadName.java:16)
        at java.lang.Thread.run(Thread.java:745)
```

从运行结果可知，Thread-0 抛出了 NPE 异常，那么单看这个日志根本无法判断是订单模块的线程抛出的异常。首先我们分析下这个 Thread-0 是怎么来的，我们看一下创建线程时的代码。

```
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize) {
    init(g, target, name, stackSize, null);
}
```

由这段代码可知，如果调用没有指定线程名称的方法创建线程，其内部会使用 "Thread-" + nextThreadNum() 作为线程的默认名称，其中 nextThreadNum 的代码如下。

```
private static int threadInitNumber;
private static synchronized int nextThreadNum() {
    return threadInitNumber++;
}
```

由此可知，threadInitNumber 是 static 变量，nextThreadNum 是 static 方法，所以线程的编号是全应用唯一的并且是递增的。这里由于涉及多线程递增 threadInitNumber，也就是执行读取—递增—写入操作，而这是线程不安全的，所以要使用方法级别的 synchronized 进行同步。

当一个系统中有多个业务模块而每个模块又都使用自己的线程时，除非抛出与业务相关的异常，否则你根本没法判断是哪一个模块出现了问题。现在修改代码如下。

```
static final String THREAD_SAVE_ORDER = "THREAD_SAVE_ORDER";
static final String THREAD_SAVE_ADDR = "THREAD_SAVE_ADDR";

public static void main(String[] args) {
    // 订单模块
    Thread threadOne = new Thread(new Runnable() {
        public void run() {
```

```
            System.out.println("保存订单的线程");
            throw new NullPointerException();
        }
    }, THREAD_SAVE_ORDER);
    // 发货模块
    Thread threadTwo = new Thread(new Runnable() {
        public void run() {
            System.out.println("保存收货地址的线程");
        }
    }, THREAD_SAVE_ADDR);

    threadOne.start();
    threadTwo.start();

}
```

如上代码在创建线程时给线程指定了一个与具体业务模块相关的名称，运行代码，输出结果为

```
<terminated> ThreadName2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年12
保存订单的线程
保存收货地址的线程
Exception in thread "THREAD_SAVE_ORDER" java.lang.NullPointerException
        at com.zlx.thread.ThreadName2$1.run(ThreadName2.java:13)
        at java.lang.Thread.run(Thread.java:745)
```

从运行结果就可以定位到是保存订单模块抛出了 NPE 异常，一下子就可以找到问题所在。

11.7.2　创建线程池时也需要指定线程池的名称

同理，下面通过简单的代码来说明不指定线程池名称为何难定位问题，代码如下。

```
    static ThreadPoolExecutor executorOne = new ThreadPoolExecutor(5, 5, 1,
TimeUnit.MINUTES, new LinkedBlockingQueue<>());
    static ThreadPoolExecutor executorTwo = new ThreadPoolExecutor(5, 5, 1,
TimeUnit.MINUTES, new LinkedBlockingQueue<>());

    public static void main(String[] args) {

        //接受用户链接模块
        executorOne.execute(new  Runnable() {
            public void run() {
```

```
                System.out.println("接受用户链接线程");
                throw new NullPointerException();
            }
        });
        //具体处理用户请求模块
        executorTwo.execute(new  Runnable() {
            public void run() {
                System.out.println("具体处理业务请求线程");
            }
        });

        executorOne.shutdown();
        executorTwo.shutdown();
    }
```

运行代码，输出结果如下。

```
<terminated> ThreadPool [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年12月16日 下
接受用户链接线程
Exception in thread "pool-1-thread-1" java.lang.NullPointerException
        at com.zlx.thread.ThreadPool$1.run(ThreadPool.java:18)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
具体处理业务请求线程
```

同样，我们并不知道是哪个模块的线程池抛出了这个异常，那么我们看下这个 pool-1-thread-1 是如何来的。其实这里使用了线程池默认的 ThreadFactory，查看线程池创建的源码如下。

```
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), defaultHandler);
    }

   public static ThreadFactory defaultThreadFactory() {
   return new DefaultThreadFactory();
    }

 static class DefaultThreadFactory implements ThreadFactory {
        //(1)
        private static final AtomicInteger poolNumber = new AtomicInteger(1);
```

```
    private final ThreadGroup group;
    //(2)
    private final AtomicInteger threadNumber = new AtomicInteger(1);
    //(3)
    private final String namePrefix;

    DefaultThreadFactory() {
        SecurityManager s = System.getSecurityManager();
        group = (s != null) ? s.getThreadGroup() :
                              Thread.currentThread().getThreadGroup();
        namePrefix = "pool-" +
                      poolNumber.getAndIncrement() +
                     "-thread-";
    }

    public Thread newThread(Runnable r) {
       //(4)
        Thread t = new Thread(group, r,
                              namePrefix + threadNumber.getAndIncrement(),
                              0);
        if (t.isDaemon())
            t.setDaemon(false);
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
    }
}
```

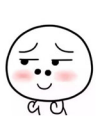
代码（1）中的 poolNumber 是 static 的原子变量，用来记录当前线程池的编号，它是应用级别的，所有线程池共用一个，比如创建第一个线程池时线程池编号为 1，创建第二个线程池时线程池的编号为 2，所以 pool-1-thread-1 里面的 pool-1 中的 1 就是这个值。

代码（2）中的 threadNumber 是线程池级别的，每个线程池使用该变量来记录该线程池中线程的编号，所以 pool-1-thread-1 里面的 thread-1 中的 1 就是这个值。

代码（3）中的 namePrefix 是线程池中线程名称的前缀，默认固定为 pool。

代码（4）具体创建线程，线程的名称是使用 namePrefix + threadNumber.getAndIncrement() 拼接的。

由此我们知道，只需对 DefaultThreadFactory 的代码中的 namePrefix 的初始化做下手脚，即当需要创建线程池时传入与业务相关的 namePrefix 名称就可以了，代码如下。



```
// 命名线程工厂
static class NamedThreadFactory implements ThreadFactory {
    private static final AtomicInteger poolNumber = new AtomicInteger(1);
    private final ThreadGroup group;
    private final AtomicInteger threadNumber = new AtomicInteger(1);
    private final String namePrefix;

    NamedThreadFactory(String name) {

        SecurityManager s = System.getSecurityManager();
        group = (s != null) ? s.getThreadGroup() : Thread.currentThread().
          getThreadGroup();
        if (null == name || name.isEmpty()) {
            name = "pool";
        }

        namePrefix = name + "-" + poolNumber.getAndIncrement() + "-thread-";
    }

    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r, namePrefix + threadNumber.
          getAndIncrement(), 0);
        if (t.isDaemon())
            t.setDaemon(false);
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
    }
}
```

创建线程池如下。

```
    static ThreadPoolExecutor executorOne = new ThreadPoolExecutor(5, 5, 1,
        TimeUnit.MINUTES,
        new LinkedBlockingQueue<>(), new NamedThreadFactory("ASYN-ACCEPT-POOL"));
static ThreadPoolExecutor executorTwo = new ThreadPoolExecutor(5, 5, 1,
         TimeUnit.MINUTES,
        new LinkedBlockingQueue<>(), new NamedThreadFactory("ASYN-PROCESS-POOL"));
```

执行结果如下。

```
<terminated> ThreadPool2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年12月16日 下午7:09:25)
接受用户链接线程Exception in thread "ASYN-ACCEPT-POOL-1-thread-1"
java.lang.NullPointerException
        at com.zlx.thread.ThreadPool2$1.run(ThreadPool2.java:50)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
具体处理业务请求线程
```

从 ASYN-ACCEPT-POOL-1-thread-1 就可以知道，这是接受用户链接线程池抛出的异常。

11.7.3　小结

本节通过简单的例子介绍了为何不为线程或者线程池起名字会给问题排查带来麻烦，然后通过源码分析介绍了线程和线程池名称及默认名称是如何来的，以及如何定义线程池名称以便追溯问题。另外，在 run 方法内使用 try-catch 块，避免将异常抛到 run 方法之外，同时打印日志也是一个最佳实践。

11.8　使用线程池的情况下当程序结束时记得调用 shutdown 关闭线程池

在日常开发中为了便于线程的有效复用，经常会用到线程池，然而使用完线程池后如果不调用 shutdown 关闭线程池，则会导致线程池资源一直不被释放。下面通过简单的例子来说明该问题。

11.8.1　问题复现

下面通过一个例子说明如果不调用线程池对象的 shutdown 方法关闭线程池，则当线程池里面的任务执行完毕并且主线程已经退出后，JVM 仍然存在。

```
public class TestShutDown {

        static void asynExecuteOne() {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(new  Runnable() {
            public void run() {
                System.out.println("--async execute one ---");
            }
        });
    }
```

```
    static void asynExecuteTwo() {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(new  Runnable() {
            public void run() {
                System.out.println("--async execute two ---");
            }
        });
    }


    public static void main(String[] args) {
       //(1)同步执行
        System.out.println("---sync execute---");
       //(2)异步执行操作one
        asynExecuteOne();
       //(3)异步执行操作two
        asynExecuteTwo();
       //(4)执行完毕
        System.out.println("---execute over---");
    }
}
```

在如上代码的主线程里面，首先同步执行了代码（1），然后执行代码（2）和代码（3），代码（2）和代码（3）使用线程池的一个线程执行异步操作，我们期望当主线程与代码（2）和代码（3）执行完线程池里面的任务后整个 JVM 就会退出，但是执行结果却如下所示。

```
TestShutDown [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (201
---sync execute---
--async execute one ---
---execute over---
--async execute two ---
```

右上的方块说明 JVM 进程还没有退出，在 Mac 上执行 ps -eaf|grep java 命令后发现 Java 进程还存在，这是什么情况呢？修改代码（2）和代码（3），在方法里面添加调用线程池的 shutdown 方法的代码。

```
        static void asynExecuteOne() {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(new  Runnable() {
            public void run() {
                System.out.println("--async execute one ---");
            }
```

```
        });

        executor.shutdown();
    }

    static void asynExecuteTwo() {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.execute(new  Runnable() {
            public void run() {
                System.out.println("--async execute two ---");
            }
        });

        executor.shutdown();
    }
```

再次执行代码你会发现 JVM 已经退出了，使用 ps -eaf|grep java 命令查看，发现 Java 进程已经不存在了，这说明只有调用了线程池的 shutdown 方法后，线程池任务执行完毕，线程池资源才会被释放。

11.8.2　问题分析

下面看为何会如此？大家或许还记得在基础篇讲解的守护线程与用户线程，JVM 退出的条件是当前不存在用户线程，而线程池默认的 ThreadFactory 创建的线程是用户线程。

```
static class DefaultThreadFactory implements ThreadFactory {
    ...
    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r,
                              namePrefix + threadNumber.getAndIncrement(),
                              0);
        if (t.isDaemon())
            t.setDaemon(false);
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
    }
}
```

由如上代码可知，线程池默认的 ThreadFactory 创建的都是用户线程。而线程池里面的核心线程是一直存在的，如果没有任务则会被阻塞，所以线程池里面的用户线程一直存在。而 shutdown 方法的作用就是让这些核心线程终止，下面简单看下 shutdown 的主要代码。

```
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            ...
            //设置线程池状态为SHUTDOWN
            advanceRunState(SHUTDOWN);
            //中断所有的空闲工作线程
            interruptIdleWorkers();
            ...
        } finally {
            mainLock.unlock();
        }
           ...
        }
```

这里在 shutdown 方法里面设置了线程池的状态为 SHUTDOWN，并且设置了所有 Worker 空闲线程（阻塞到队列的 take() 方法的线程）的中断标志。那么下面来看在工作线程 Worker 里面是不是设置了中断标志，然后它就会退出。

```
  final void runWorker(Worker w) {
            ...
            try {
            while (task != null || (task = getTask()) != null) {
               ...
            }
            ...
          } finally {
            ...
        }
    }

private Runnable getTask() {
        boolean timedOut = false;

        for (;;) {
            ...
            //(1)
            if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                decrementWorkerCount();
                return null;
            }

            try {
```

```
                //(2)
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }
```

在如上代码中，在正常情况下如果队列里面没有任务，则工作线程被阻塞到代码（2）等待从工作队列里面获取一个任务。这时候如果调用线程池的 shutdown 命令（shutdown 命令会中断所有工作线程），则代码（2）会抛出 InterruptedException 异常而返回，而这个异常被捕捉到了，所以继续执行代码（1），而执行 shutdown 时设置了线程池的状态为 SHUTDOWN，所以 getTask 方法返回了 null，因而 runWorker 方法退出循环，该工作线程就退出了。

11.8.3　小结

本节通过一个简单的使用线程池异步执行任务的案例介绍了使用完线程池后如果不调用 shutdown 方法，则会导致线程池的线程资源一直不会被释放，并通过源码分析了没有被释放的原因。所以在日常开发中使用线程池后一定不要忘记调用 shutdown 方法关闭。

11.9　线程池使用 FutureTask 时需要注意的事情

线程池使用 FutureTask 时如果把拒绝策略设置为 DiscardPolicy 和 DiscardOldestPolicy，并且在被拒绝的任务的 Future 对象上调用了无参 get 方法，那么调用线程会一直被阻塞。

11.9.1　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
    }
```

输出结果为

```
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）创建了一个单线程和一个队列元素个数为 1 的线程池，并且把拒绝策略设置为 DiscardPolicy。

代码（2）向线程池提交了一个任务 one，并且这个任务会由唯一的线程来执行，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 放入阻塞队列。

代码（4）向线程池提交任务 three，由于队列已满所以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）并等待任务 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）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）会一直阻塞而不会返回，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。但是如果把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.Thr
eadPoolExecutor@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
```

11.9.2 问题分析

要分析这个问题，需要看线程池的 submit 方法都做了什么，submit 方法的代码如下。

```
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) 如果线程个数小于核心线程数则新增处理线程
    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把任务放入队列
    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
    else if (!addWorker(command, false))
        reject(command);//(5)新增失败则调用拒绝策略
}
```

在以上代码中，代码（1）装饰 Runnable 为 FutureTask 对象，然后调用线程池的 execute 方法。

代码（2）判断如果线程个数小于核心线程数则新增处理线程。

代码（3）判断如果当前线程个数已经达到核心线程数则将任务放入队列 。

代码（4）尝试新增处理线程。失败则执行代码（5），否则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，使用业务线程执行拒绝策略。

所以要找到上面例子中问题所在，只需要看代码（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) {
    }
}
```

拒绝策略的 rejectedExecution 方法什么都没做，代码（4）调用 submit 后会返回一个 Future 对象。这里有必要再次重申，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;
```

在代码（1）中使用 newTaskFor 方法将 Runnable 任务转换为 FutureTask，而在 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 Future 变为什么状态才会返回 , 那就要看下 FutureTask 的 get（）方法代码。

仅供非商业用途或交流学习使用


```
    public V get() throws InterruptedException, ExecutionException {
        int s = state;
        //当状态值<=COMPLETING时需要等待，否则调用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则抛出异常
    if (s >= CANCELLED)
        throw new CancellationException();
    throw new ExecutionException((Throwable)x);
}
```

也就是说，当 Future 的状态 >COMPLETING 时调用 get 方法才会返回，而明显 DiscardPolicy 策略在拒绝元素时并没有设置该 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 策略也存在这样的问题，最老的任务被淘汰时没有设置被淘汰任务对应 Future 的状态。

那么默认的 AbortPolicy 策略为啥没问题呢？其实在执行 AbortPolicy 策略时，代码（5）会直接抛出 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 时，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，超时时间到了就会自动返回。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，在执行策略时设置该 Future 的状态大于 COMPLETING 即可。但是我们查看 FutureTask 提供的方法，会发现只有 cancel 方法是 public 的，并且可以设置 FutureTask 的状态大于 COMPLETING，则重写拒绝策略的具体代码如下。

```
public class MyRejectedExecutionHandler implements RejectedExecutionHandler{

    @Override
```

仅供非商业用途或交流学习使用


```
    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，由于在 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 块捕获异常，因此将代码（7）修改为如下所示。

```
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多了一个异常捕获操作。最好的情况是，重写拒绝策略时设置 FutureTask 的状态为 NORMAL，但是这需要重写 FutureTask 方法，因为 FutureTask 并没有提供接口让我们设置。

11.9.3　小结

本节通过案例介绍了在线程池中使用 FutureTask 时，当拒绝策略为 DiscardPolicy 和 DiscardOldestPolicy 时，在被拒绝的任务的 FutureTask 对象上调用 get() 方法会导致调用线程一直阻塞，所以在日常开发中尽量使用带超时参数的 get 方法以避免线程一直阻塞。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

11.10 使用 ThreadLocal 不当可能会导致内存泄漏

在基础篇已经讲解了 ThreadLocal 的原理，本节着重介绍使用 ThreadLocal 会导致内存泄漏的原因，并给出使用 ThreadLocal 导致内存泄漏的案例。

11.10.1 为何会出现内存泄漏

在基础篇我们讲了，ThreadLocal 只是一个工具类，具体存放变量的是线程的 threadLocals 变量。threadLocals 是一个 ThreadLocalMap 类型的变量，该类型如图 11-10 所示。

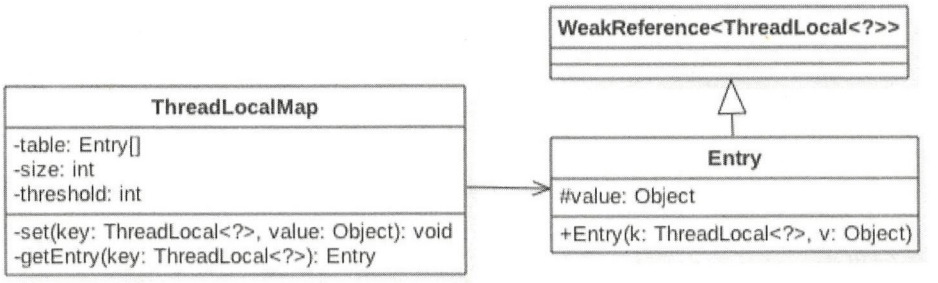


图 11-10

由图 11-10 可知，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呢？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) {
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


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

k 被传递给 WeakReference 的构造函数，也就是说 ThreadLocalMap 里面的 key 为 ThreadLocal 对象的弱引用，具体就是 referent 变量引用了 ThreadLocal 对象，value 为具体调用 ThreadLocal 的 set 方法时传递的值。

当一个线程调用 ThreadLocal 的 set 方法设置变量时，当前线程的 ThreadLocalMap 里就会存放一个记录，这个记录的 key 为 ThreadLocal 的弱引用，value 则为设置的值。如果当前线程一直存在且没有调用 ThreadLocal 的 remove 方法，并且这时候在其他地方还有对 ThreadLocal 的引用，则当前线程的 ThreadLocalMap 变量里面会存在对 ThreadLocal 变量的引用和对 value 对象的引用，它们是不会被释放的，这就会造成内存泄漏。

考虑这个 ThreadLocal 变量没有其他强依赖，而当前线程还存在的情况，由于线程的 ThreadLocalMap 里面的 key 是弱依赖，所以当前线程的 ThreadLocalMap 里面的 ThreadLocal 变量的弱引用会在 gc 的时候被回收，但是对应的 value 还是会造成内存泄漏，因为这时候 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、get 和 remove 方法里面可以找一些时机对这些 key 为 null 的 entry 进行清理，但是这些清理不是必须发生的。下面简单说下 ThreadLocalMap 的 remove 方法中的清理过程。

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

  //(1)计算当前ThreadLocal变量所在的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，遍历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);
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            return;
        }
    }
}
```

代码（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。

如下代码（6）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。

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

            //（6）去掉对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;
                    }
                }
            }
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            return i;
        }
```

代码（7）从当前元素的下标开始查看 table 数组里面是否有 key 为 null 的其他元素，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。

总结：ThreadLocalMap 的 Entry 中的 key 使用的是对 ThreadLocal 对象的弱引用，这在避免内存泄漏方面是一个进步，因为如果是强引用，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 ThreadLocal 引用是会被回收掉的。但是对应的 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 和 remove 方法，可以在一些时机下对这些 Entry 项进行清理，但是这是不及时的，也不是每次都会执行，所以在一些情况下还是会发生内存漏，因此在使用完毕后及时调用 remove 方法才是解决内存泄漏问题的王道。

11.10.2 在线程池中使用 ThreadLocal 导致的内存泄漏

下面先看一个在线程池中使用 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() {
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
                // (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都为 5 的线程池。

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 变量中。

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。

运行代码，使用 jconsole 监控堆内存变化，如图 11-11 所示。

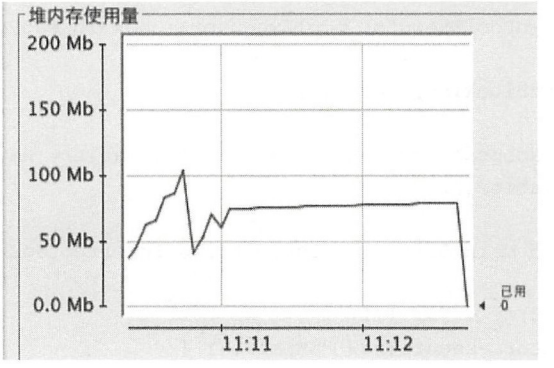


图 11-11

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

然后去掉 localVariable.remove() 注释，再运行，观察堆内存变化，如图 11-12 所示。

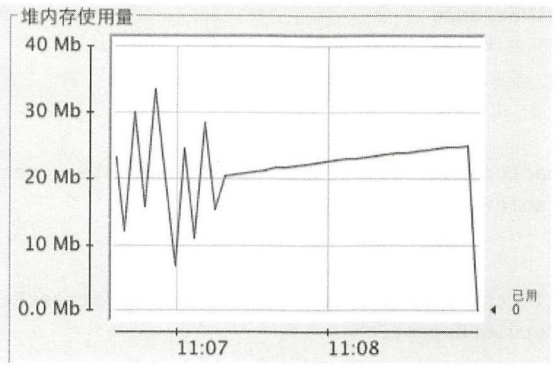


图 11-12

从运行结果一（图 11-11）可知，当主线程处于休眠时，进程占用了大概 77MB 内存，运行结果二（图 11-12）显示占用了大概 25MB 内存，由此可知运行代码一时发生了内存泄漏，下面分析泄露的原因。

第一次运行代码时，在设置线程的 localVariable 变量后没有调用 localVariable.remove() 方法，这导致线程池里面 5 个核心线程的 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了，但是线程池里面的 5 个线程会一直存在直到 JVM 进程被杀死。这里需要注意的是，由于 localVariable 被声明为了 static 变量，虽然在线程的 ThreadLocalMap 里面对 localVariable 进行了弱引用，但是 localVariable 不会被回收。第二次运行代码时，由于线程在设置 localVariable 变量后及时调用了 localVariable.remove() 方法进行了清理，所以不会存在内存泄漏问题。

总结：如果在线程池里面设置了 ThreadLocal 变量，则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，线程池的核心线程的 threadLocals 变量会一直持有 ThreadLocal 变量。

11.10.3　在 Tomcat 的 Servlet 中使用 ThreadLocal 导致内存泄漏

首先看一个 Servlet 的代码。

```
public class HelloWorldExample extends HttpServlet {
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    private static final long serialVersionUID = 1L;

    static class LocalVariable {
        private Long[] a = new Long[1024 * 1024 * 100];
    }

    //(1)
    final static ThreadLocal<LocalVariable> localVariable = new
ThreadLocal<LocalVariable>();

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
        //(2)
        localVariable.set(new LocalVariable());

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");

        out.println("<title>" + "title" + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        //(3)
        out.println(this.toString());
        //(4)
        out.println(Thread.currentThread().toString());

        out.println("</body>");
        out.println("</html>");
    }
}
```

代码（1）创建一个 localVariable 对象。

代码（2）在 Servlet 的 doGet 方法内设置 localVariable 值。

代码（3）打印当前 Servlet 的实例。

代码（4）打印当前线程。

修改 Tomcat 的 conf 下 sever.xml 配置如下。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
    maxThreads="10" minSpareThreads="5"/>

<Connector executor="tomcatThreadPool" port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

这里设置了 Tomcat 的处理线程池的最大线程数为 10，最小线程数为 5。那么这个线程池是干什么用的呢？我们回顾下 Tomcat 的容器结构，如图 11-13 所示。

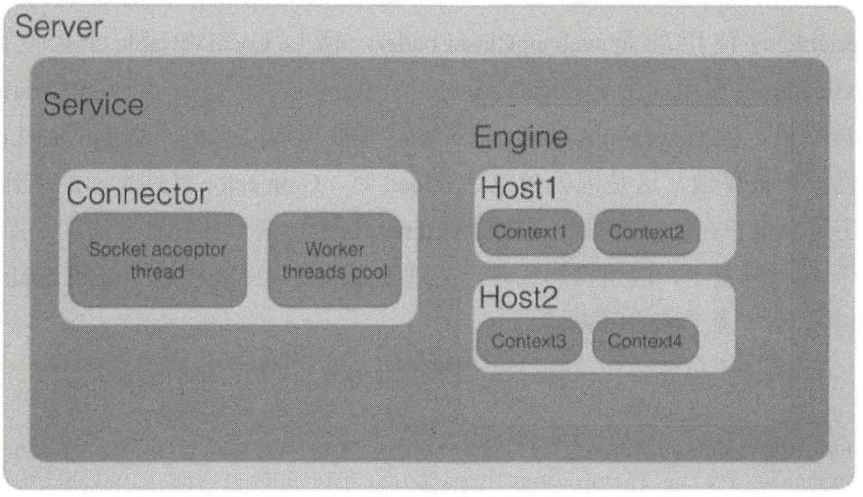


图 11-13

Tomcat 中的 Connector 组件负责接收并处理请求，其中 Socket acceptor thread 负责接收用户的访问请求，然后把接收到的请求交给 Worker threads pool 线程池进行具体处理，后者就是我们在 server.xml 里面配置的线程池。Worker threads pool 里面的线程则负责把具体请求分发到具体的应用的 Servlet 上进行处理。

那么，下面启动 Tomcat 访问该 Servlet 多次，你会发现可能输出下面的结果：

```
HelloWorldExample@2a10b2d2 Thread[catalina-exec-5,5,main]
HelloWorldExample@2a10b2d2 Thread[catalina-exec-1,5,main]
HelloWorldExample@2a10b2d2 Thread[catalina-exec-4,5,main]
```

输出的前半部分是 Servlet 实例，可以看出都一样，这说明多次访问的是同一个 Servlet 实例，后半部分中的 catalina-exec-5、catalina-exec-1、catalina-exec-4，则说明使用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

了 Connector 中的线程池里面的线程 5、线程 1，线程 4 来执行 Servlet。

如果在访问该 Servlet 的同时打开 jconsole 观察堆内存，会发现内存飙升，究其原因是因为工作线程在调用 Servlet 的 doGet 方法时，工作线程的 threadLocals 变量里面被添加了 LocalVariable 实例，但是后来没有清除。另外多次访问该 Servlet 可能使用的不是工作线程池里面的同一个线程，这会导致工作线程池里面多个线程都会存在内存泄漏问题。

更糟糕的还在后面，上面的代码在 Tomcat 6.0 时代，应用 reload 操作后会导致加载该应用的 webappClassLoader 释放不了，这是因为在 Servlet 的 doGet 方法里面创建 LocalVariable 时使用的是 webappClassLoader，所以 LocalVariable.class 里面持有对 webappclassloader 的引用。由于 LocalVariable 实例没有被释放，所以 LocalVariable.class 对象也没有被释放，因而 webappClassLoader 也没有被释放，那么 webappClassLoader 加载的所有类也没有被释放。这是因为当应用 reload 时，Connector 组件里面的工作线程池里面的线程还是一直存在的，并且线程里面的 threadLocals 变量并没有被清理。而在 Tomcat 7.0 中这个问题被修复了，应用在加载时会清理工作线程池中线程的 threadLocals 变量。在 Tomcat 7.0 中，加载后会有如下提示。

```
十二月 31, 2017 5:44:24 下午 org.apache.catalina.loader.WebappClassLoader
checkThreadLocalMapForLeaks
严重: The web application [/examples] created a ThreadLocal with key of type [java.
lang.ThreadLocal] (value [java.lang.ThreadLocal@63a3e00b]) and a value of type
[HelloWorldExample.LocalVariable] (value [HelloWorldExample$LocalVariable@4fd7564b])
but failed to remove it when the web application was stopped. Threads are going to
be renewed over time to try and avoid a probable memory leak.
```

11.10.4 小结

Java 提供的 ThreadLocal 给我们编程提供了方便，但是如果使用不当也会给我们带来麻烦，所以要养成良好的编码习惯，在线程中使用完 ThreadLocal 变量后，要记得及时清除掉。

11.11 总结

本章首先结合开源框架 Logback 日志系统和 Tomcat 容器讲解了并发队列的使用，然后讲解了在并发编程时容易遇到的问题以及解决方法。读者在阅读完本章后最好动手去实践，尝试在项目实践中解决类似问题。

仅供非商业用途或交流学习使用

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

业界点评

作者是一位喜欢用代码说话的同学……跟随作者进行一番代码级的探究，所产生的印象比阅读文章、死记结论，无疑要深刻得多。

——肖桦（江南白衣），唯品会资深架构师，公众号“春天的旁边”

本书作者在阿里经历过大量并发的场景，积攒了不少并发编程的经验，并毫无保留地写入本书。通过书中对JUC源码的解读，读者可以揭开JUC的神秘面纱。这是一本值得仔细品读的好书。

——你假笨/寒泉子，PerfMa CEO，公众号“你假笨”

本书用浅显易懂的文字为大家系统地介绍了 Java 并发编程的相关内容，推荐大家关注学习。

——纯洁的微笑，第三方支付公司技术总监，公众号“纯洁的微笑”

Java并发编程无处不在……如果你希望成长为一名优秀的Java程序员，有必要读一读本书。

——许令波，《深入分析Java Web技术内幕》作者

作者简介

翟陆续，花名加多，四川大学计算机学院研究生毕业。目前任淘宝技术高级开发工程师，热衷于Java并发编程，对JUC包源码有深入的研究，熟悉常用开源框架实现原理。

薛宾田，四川大学计算机学院研究生毕业，原阿里巴巴研发工程师，目前在河南牧业经济学院信息工程学院担任Java课程老师。

注册成为博文视点社区（www.broadview.com.cn）用户，即享受以下服务：

- 提勘误赚积分：可在【提交勘误】处提交对内容的修改意见，若被采纳将获赠博文视点社区积分（可用来抵扣购买电子书的相应金额）。
- 交流学习：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者共同交流。
- 页面入口：http://www.broadview.com.cn/34947

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：刘　皎
责任编辑：牛　勇
封面设计：侯士卿

欢迎投稿
邮箱：Ljiao@phei.com.cn
电话：010-88254395
新浪微博：@皎丫子

上架建议：编程语言

ISBN 978-7-121-34947-8
9 787121 349478 >
定价：89.00元