title: 精尽 Dubbo 源码分析 —— 线程池 date: 2018-03-05 tags: categories: Dubbo permalink: Dubbo/thread-pool

---

摘要: 原创出处 http://www.iocoder.cn/Dubbo/thread-pool/ 「芋道源码」欢迎转载，保留摘要，谢谢！

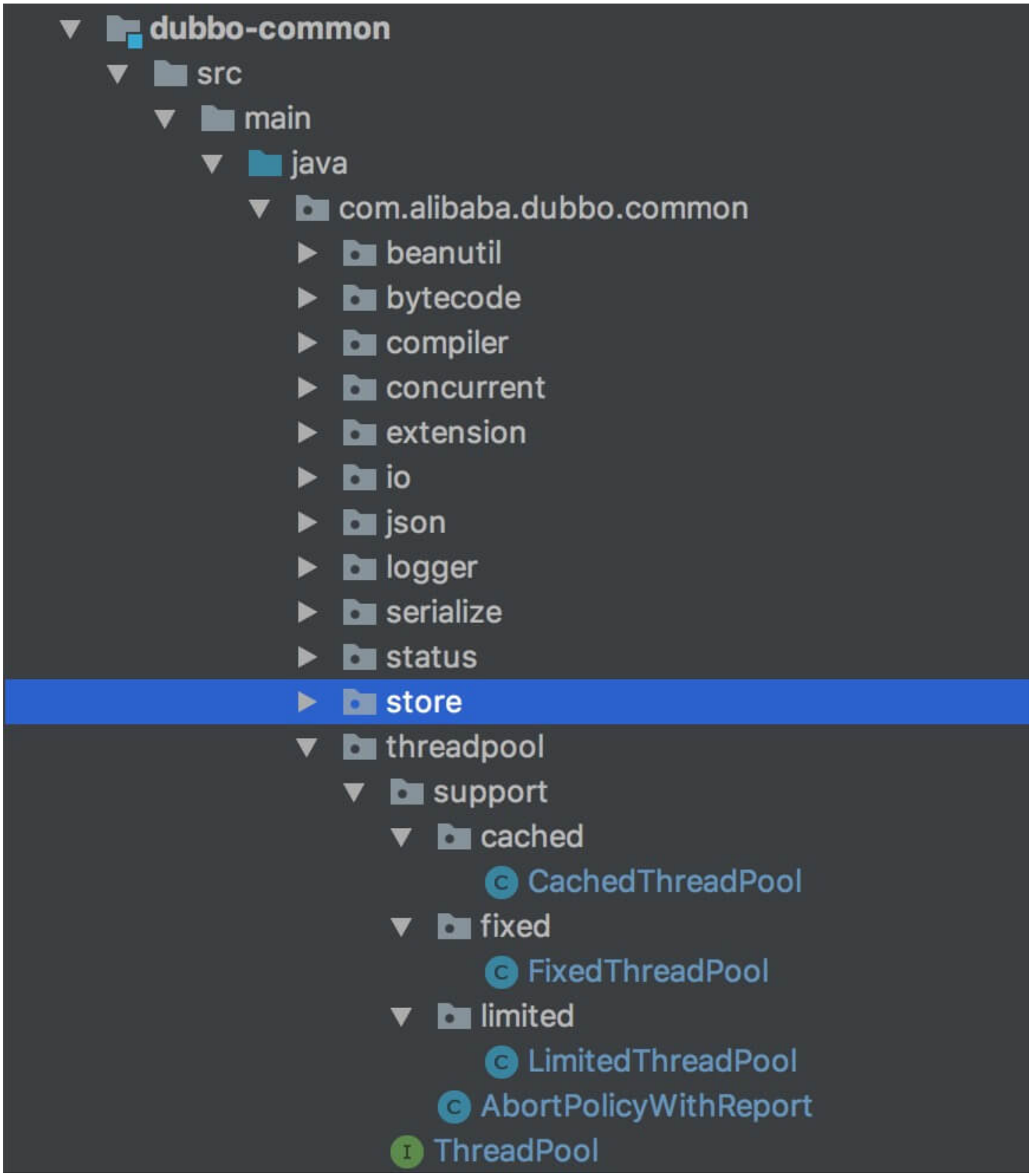
---

1. 概述

在 《Dubbo 用户指南 —— 线程模型》 一文中，我们可以看到 Dubbo 提供了**三种线程池的实现**：

> ThreadPool
>
> - `fixed` 固定大小线程池，启动时建立线程，不关闭，一直持有。(**缺省**)
> - `cached` 缓存线程池，空闲一分钟自动删除，需要时重建。
> - `limited` 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。

在 `dubbo-common` 模块的 `threadpool` 包下实现，如下图所示：



2. ThreadPool

`com.alibaba.dubbo.common.threadpool.ThreadPool` ，线程池接口。代码如下：

```
@SPI("fixed")
```

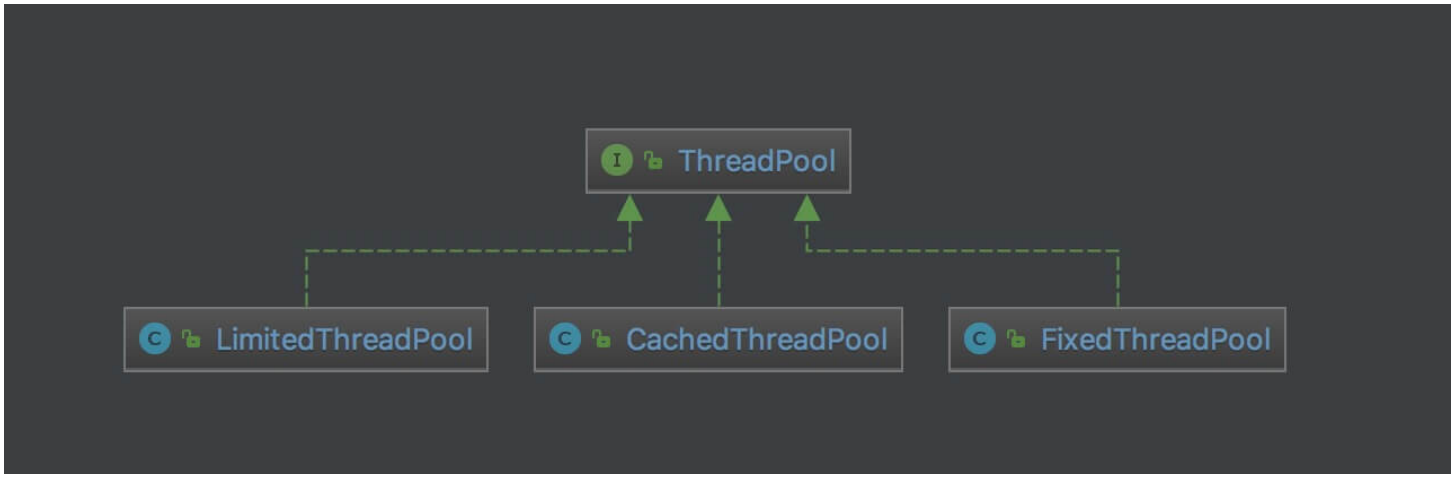
```java
public interface ThreadPool {

    /**
     * Thread pool
     *
     * @param url URL contains thread parameter
     * @return thread pool
     */
    @Adaptive({Constants.THREADPOOL_KEY})
    Executor getExecutor(URL url);

}
```

- `@SPI("fixed")` 注解，Dubbo SPI **拓展点**，默认为 `"fixed"` 。
- `@Adaptive({Constants.THREADPOOL_KEY})` 注解，基于 Dubbo SPI Adaptive 机制，加载对应的线程池实现，使用 `URL.threadpool` 属性。
  - `#getExecutor(url)` 方法，获得**对应**的线程池的执行器。

子类类图如下：



2.1 FixedThreadPool

`com.alibaba.dubbo.common.threadpool.support.fixed.FixedThreadPool` ，实现 ThreadPool 接口，固定大小线程池，启动时建立线程，不关闭，一直持有。代码如下：

```java
  1: public class FixedThreadPool implements ThreadPool {
  2:
  3:     @Override
  4:     public Executor getExecutor(URL url) {
  5:         // 线程名
  6:         String name = url.getParameter(Constants.THREAD_NAME_KEY, Constants.DE
FAULT_THREAD_NAME);
```

```
 7:         // 线程数
 8:         int threads = url.getParameter(Constants.THREADS_KEY, Constants.DEFAUL
T_THREADS);
 9:         // 队列数
10:         int queues = url.getParameter(Constants.QUEUES_KEY, Constants.DEFAULT_
QUEUES);
11:         // 创建执行器
12:         return new ThreadPoolExecutor(threads, threads, 0, TimeUnit.MILLISECON
DS,
13:                 queues == 0 ? new SynchronousQueue<Runnable>() :
14:                         (queues < 0 ? new LinkedBlockingQueue<Runnable>()
15:                                 : new LinkedBlockingQueue<Runnable>(queues)),
16:                 new NamedThreadFactory(name, true), new AbortPolicyWithReport(
name, url));
17:     }
18:
19: }
```

- 第 5 至 10 行：获得线程名、线程数、队列数。目前只有服务提供者使用，配置方式如下：

```
<dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoService"
>

    <dubbo:parameter key="threadname" value="shuaiqi" />
    <dubbo:parameter key="threads" value="123" />
    <dubbo:parameter key="queues" value="10" />

</dubbo:service>
```

- 第 11 至 16 行：创建执行器 ThreadPoolExecutor 对象。
  - 根据不同的队列数，使用不同的队列实现：
    - 第 13 行：`queues == 0`， SynchronousQueue 对象。
    - 第 14 行：`queues < 0`， LinkedBlockingQueue 对象。
    - 第 15 行：`queues > 0`，带队列数的 LinkedBlockingQueue 对象。
  - 推荐阅读：
    - 《Java并发包中的同步队列SynchronousQueue实现原理》
    - 《Java阻塞队列ArrayBlockingQueue和LinkedBlockingQueue实现原理分析》
    - 《聊聊并发（三）——JAVA线程池的分析和使用》
    - 《聊聊并发（七）——Java中的阻塞队列》
  - 第 16 行：创建 NamedThreadFactory 对象，用于生成**线程名**。
  - 第 16 行：创建 AbortPolicyWithReport 对象，用于**当任务添加到线程池中被拒绝时**。

2.2 CachedThreadPool

`com.alibaba.dubbo.common.threadpool.support.cached.CachedThreadPool` ，实现 ThreadPool 接口，缓存线程池，空闲一定时长，自动删除，需要时重建。代码如下：

```
  1: public class CachedThreadPool implements ThreadPool {
  2:
  3:     @Override
  4:     public Executor getExecutor(URL url) {
  5:         // 线程池名
  6:         String name = url.getParameter(Constants.THREAD_NAME_KEY, Constants.DE
FAULT_THREAD_NAME);
  7:         // 核心线程数
  8:         int cores = url.getParameter(Constants.CORE_THREADS_KEY, Constants.DEF
AULT_CORE_THREADS);
  9:         // 最大线程数
 10:         int threads = url.getParameter(Constants.THREADS_KEY, Integer.MAX_VALU
E);
 11:         // 队列数
 12:         int queues = url.getParameter(Constants.QUEUES_KEY, Constants.DEFAULT_
QUEUES);
 13:         // 线程存活时长
 14:         int alive = url.getParameter(Constants.ALIVE_KEY, Constants.DEFAULT_AL
IVE);
 15:         // 创建执行器
 16:         return new ThreadPoolExecutor(cores, threads, alive, TimeUnit.MILLISEC
ONDS,
 17:                 queues == 0 ? new SynchronousQueue<Runnable>() :
 18:                         (queues < 0 ? new LinkedBlockingQueue<Runnable>()
 19:                                 : new LinkedBlockingQueue<Runnable>(queues)),
 20:                 new NamedThreadFactory(name, true), new AbortPolicyWithReport(
name, url));
 21:     }
 22:
 23: }
```

- 第 5 至 14 行：获得线程名、核心线程数、最大线程数、队列数、线程存活时长。
  - 🙂 配置方式和 FixedThreadPool 类似，使用 `<dubbo:parameter />` 配置。
- 第 16 至 20 行：创建执行器 ThreadPoolExecutor 对象。
  - 🙂 和 FixedThreadPool 相对类似。

2.3 LimitedThreadPool

`com.alibaba.dubbo.common.threadpool.support.limited.LimitedThreadPool` ，实现 ThreadPool 接口，可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。代码如下：

```
  1: public class LimitedThreadPool implements ThreadPool {
  2:
  3:     @Override
  4:     public Executor getExecutor(URL url) {
  5:         // 线程名
  6:         String name = url.getParameter(Constants.THREAD_NAME_KEY, Constants.DE
FAULT_THREAD_NAME);
  7:         // 核心线程数
  8:         int cores = url.getParameter(Constants.CORE_THREADS_KEY, Constants.DEF
AULT_CORE_THREADS);
  9:         // 最大线程数
 10:         int threads = url.getParameter(Constants.THREADS_KEY, Constants.DEFAUL
T_THREADS);
 11:         // 队列数
 12:         int queues = url.getParameter(Constants.QUEUES_KEY, Constants.DEFAULT_
QUEUES);
 13:         // 创建执行器
 14:         return new ThreadPoolExecutor(cores, threads, Long.MAX_VALUE, TimeUnit
.MILLISECONDS,
 15:                 queues == 0 ? new SynchronousQueue<Runnable>() :
 16:                         (queues < 0 ? new LinkedBlockingQueue<Runnable>()
 17:                                 : new LinkedBlockingQueue<Runnable>(queues)),
 18:                 new NamedThreadFactory(name, true), new AbortPolicyWithReport(
name, url));
 19:     }
 20:
 21: }
```

- 和 CachedThreadPool 实现是基本一致的，差异点在 `alive == Integer.MAX_VALUE` ，**空闲时间无限大**，即不会自动删除。

3. AbortPolicyWithReport

`com.alibaba.dubbo.common.threadpool.support.AbortPolicyWithReport` ，实现 `java.util.concurrent.ThreadPoolExecutor.AbortPolicy` ，拒绝策略实现类。**打印 JStack ，分析线程状态**。

3.1 属性

```
/**
 * 线程名
 */
private final String threadName;
/**
 * URL 对象
 */
private final URL url;
/**
 * 最后打印时间
 */
private static volatile long lastPrintTime = 0;
/**
 * 信号量，大小为 1 。
 */
private static Semaphore guard = new Semaphore(1);

public AbortPolicyWithReport(String threadName, URL url) {
    this.threadName = threadName;
    this.url = url;
}
```

3.2 rejectedExecution

`#rejectedExecution(Runnable, ThreadPoolExecutor)` **实现**方法，代码如下：

```
  1: @Override
  2: public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
  3:     // 打印告警日志
  4:     String msg = String.format("Thread pool is EXHAUSTED!" +
  5:                     " Thread Name: %s, Pool Size: %d (active: %d, core: %d, ma
x: %d, largest: %d), Task: %d (completed: %d)," +
  6:                     " Executor status:(isShutdown:%s, isTerminated:%s, isTermi
nating:%s), in %s://%s:%d!",
  7:             threadName, e.getPoolSize(), e.getActiveCount(), e.getCorePoolSize
(), e.getMaximumPoolSize(), e.getLargestPoolSize(),
  8:             e.getTaskCount(), e.getCompletedTaskCount(), e.isShutdown(), e.isT
erminated(), e.isTerminating(),
  9:             url.getProtocol(), url.getIp(), url.getPort());
 10:     logger.warn(msg);
 11:     // 打印 JStack ，分析线程状态。
```

```
12:     dumpJStack();
13:     // 抛出 RejectedExecutionException 异常
14:     throw new RejectedExecutionException(msg);
15: }
```

- 第 3 至 10 行：打印**告警日志**。
- 第 12 行：调用 `#dumpJStack()` 方法，打印 **JStack** ，分析线程状态。
- 第 14 行：抛出 RejectedExecutionException 异常。

3.3 dumpJStack

`#dumpJStack()` 方法，打印 JStack。代码如下：

```
 1: private void dumpJStack() {
 2:     long now = System.currentTimeMillis();
 3:
 4:     // 每 10 分钟，打印一次。
 5:     // dump every 10 minutes
 6:     if (now - lastPrintTime < 10 * 60 * 1000) {
 7:         return;
 8:     }
 9:
10:     // 获得信号量
11:     if (!guard.tryAcquire()) {
12:         return;
13:     }
14:
15:     // 创建线程池，后台执行打印 JStack
16:     Executors.newSingleThreadExecutor().execute(new Runnable() {
17:         @Override
18:         public void run() {
19:
20:             // 获得系统
21:             String OS = System.getProperty("os.name").toLowerCase();
22:             // 获得路径
23:             String dumpPath = url.getParameter(Constants.DUMP_DIRECTORY, Syste
m.getProperty("user.home"));
24:             SimpleDateFormat sdf;
25:             // window system don't support ":" in file name
26:             if(OS.contains("win")){
27:                 sdf = new SimpleDateFormat("yyyy-MM-dd_HH-mm-ss");
28:             }else {
29:                 sdf = new SimpleDateFormat("yyyy-MM-dd_HH:mm:ss");
30:             }
31:             String dateStr = sdf.format(new Date());
```

```
 32:             // 获得输出流
 33:             FileOutputStream jstackStream = null;
 34:             try {
 35:                 jstackStream = new FileOutputStream(new File(dumpPath, "Dubbo_
JStack.log" + "." + dateStr));
 36:                 // 打印 JStack
 37:                 JVMUtil.jstack(jstackStream);
 38:             } catch (Throwable t) {
 39:                 logger.error("dump jstack error", t);
 40:             } finally {
 41:                 // 释放信号量
 42:                 guard.release();
 43:                 // 释放输出流
 44:                 if (jstackStream != null) {
 45:                     try {
 46:                         jstackStream.flush();
 47:                         jstackStream.close();
 48:                     } catch (IOException e) {
 49:                     }
 50:                 }
 51:             }
 52:             // 记录最后打印时间
 53:             lastPrintTime = System.currentTimeMillis();
 54:         }
 55:     });
 56:
 57: }
```

- 第 2 至 8 行：每 10 分钟，**只**打印一次。
- 第 10 至 13 行：获得信号量。保证，同一时间，**有且仅有**一个线程执行打印。
- 第 15 至 54 行：创建线程池，**后台**执行打印 JStack 。
  - 第 20 至 31 行：获得路径。
  - 第 32 至 35 行：获得文件输出流。
  - 第 37 行：调用 `JVMUtil#jstack(OutputStream)` 方法，打印 JStack 。
  - 第 42 行：释放信号量。
  - 第 44 至 50 行：释放输出流。
  - 第 53 行：记录最后打印时间。

4. JVMUtil

`com.alibaba.dubbo.common.utils.JVMUtil` ，JVM 工具类。目前，仅有 JStack 功能，胖友可以点击链接，自己查看下代码。

如下是一个 JStack 日志的示例：

```
123312:tmp yunai$ cat Dubbo_JStack.log.2018-03-27_18\:57\:32
"pool-2-thread-1" Id=11 RUNNABLE
    at sun.management.ThreadImpl.dumpThreads0(Native Method)
    at sun.management.ThreadImpl.dumpAllThreads(ThreadImpl.java:454)
    at com.alibaba.dubbo.common.utils.JVMUtil.jstack(JVMUtil.java:34)
    at com.alibaba.dubbo.common.threadpool.support.AbortPolicyWithReport$1.run(Abor
tPolicyWithReport.java:122)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:11
42)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:6
17)
    at java.lang.Thread.run(Thread.java:745)

    Number of locked synchronizers = 1
    - java.util.concurrent.ThreadPoolExecutor$Worker@5cbc508c

"Monitor Ctrl-Break" Id=5 RUNNABLE (in native)
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
    at java.net.SocketInputStream.read(SocketInputStream.java:171)
    at java.net.SocketInputStream.read(SocketInputStream.java:141)
    at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
    at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
    at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
    -  locked java.io.InputStreamReader@5c7efb52
    at java.io.InputStreamReader.read(InputStreamReader.java:184)
    at java.io.BufferedReader.fill(BufferedReader.java:161)
    at java.io.BufferedReader.readLine(BufferedReader.java:324)
    -  locked java.io.InputStreamReader@5c7efb52
    at java.io.BufferedReader.readLine(BufferedReader.java:389)
    at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:64)

"Signal Dispatcher" Id=4 RUNNABLE

"Finalizer" Id=3 WAITING on java.lang.ref.ReferenceQueue$Lock@197c6eb9
    at java.lang.Object.wait(Native Method)
    -  waiting on java.lang.ref.ReferenceQueue$Lock@197c6eb9
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)
    at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)

"Reference Handler" Id=2 WAITING on java.lang.ref.Reference$Lock@7b19fa34
    at java.lang.Object.wait(Native Method)
    -  waiting on java.lang.ref.Reference$Lock@7b19fa34
    at java.lang.Object.wait(Object.java:502)
```

```
    at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"main" Id=1 TIMED_WAITING
    at java.lang.Thread.sleep(Native Method)
    at com.alibaba.dubbo.common.threadpool.AbortPolicyWithReportTest.jStackDumpTest
(AbortPolicyWithReportTest.java:44)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62
)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl
.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.
java:50)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.j
ava:12)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.ja
va:47)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.jav
a:17)
    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.jav
a:78)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.jav
a:57)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
    at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
    at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
    at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
    at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestR
unner.java:68)
    at com.intellij.rt.execution.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(
IdeaTestRunner.java:47)
    at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitSta
rter.java:242)
    at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:70)
```

另外，胖友可以看看 《如何使用jstack分析线程状态》 文章。

666. 彩蛋

水更小文，为后面做铺垫