

博客园

首页

新随笔

联系

订阅

管理

Java并发- 3 Java线程

3.1. 创建和运行线程

3.1.1. 方法一，直接使用Thread

```
/**
 * @description: Thread 创建线程
 * @author: teago
 * @time: 2020/5/16 08:39
 */
@Slf4j(topic = "Example1")
public class Example1 {

    public static void main(String[] args) {
        Thread createThread = new Thread(() -> {
            log.debug("thread mode create thread");
        });
        createThread.start();
    }
}
```

3.1.2. 方法二，使用Runnable配合Thread

把【线程】和【任务】（要执行的代码）分开

- Thread 代表线程
- Runnable 可运行的任务（线程要执行的代码）

```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

/**
 * @description: Runnable 创建线程
 * @author: teago
 * @time: 2020/5/16 08:39
 */
@Slf4j
public class Example2 {

    public static void main(String[] args) {
        Runnable runnable = () -> log.debug(Thread.currentThread().getName());
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

公告

昵称: teago
园龄: 3年9个月
粉丝: 1
关注: 0
[关注成功](#)

2024年1月						
日	一	二	三	四	五	六
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

搜索

找找看

我的标签

Java(13)

JVM(6)

Spring(5)

并发(4)

LeetCode(1)

```
}
```

3.1.3. 原理之Thread与Runnable

分析Thread的源码，理清它与Runnable的关系

小结：

- 方法1是把线程和任务合并在了一起，方法2是把线程和任务分开了；
- 用Runnable更容易与线程池等高级API配合；
- 用Runnable让任务类脱离了Thread继承体系，更灵活。

3.1.4. FutureTask配合Thread

FutureTask 能够接收 Callable 类型的参数，用来处理有返回结果的情况，线程阻塞等待task执行完毕的结果。

```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

/**
 * @description: FutureTask配合Thread
 * @author: teago
 * @time: 2020/5/16 08:39
 */
@Slf4j
public class Example3 {

    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        FutureTask<String> futureTask = new FutureTask<>(() -> "futureTask");
        Thread thread = new Thread(futureTask);
        thread.setName("futureTask mode create thread ");
        thread.start();
        thread.join();
        // 主线程阻塞，同步等待 task 执行完毕的结果
        log.debug(futureTask.get());
    }
}
```

3.2. 观察多个线程同时运行

主要是理解：

- 交替执行
- 谁先谁后，不由我们控制

```
package com.bloom.concurrent.three;

import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 08:53
 */
public class Example4 {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            while (true){
                System.out.println("test jconsole use");
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "t4");
        thread.start();
    }
}
```

随笔分类

JVM(5)

Spring(5)

并发(4)

算法(1)

随笔档案

2020年5月(4)

2020年4月(11)

阅读排行榜

1. Java并发- 4 共享模型管程👤(450)

2. JVM虚拟机-03、JVM内存分配机制与垃圾回收算法(377)

3. Spring源码分析（一）-Spring容器以及Bean的实例化(266)

4. Spring 源码阅读（三）-自动注入(261)

5. Spring源码分析（二）-Spring依赖注入与方法注入(244)

```
}  
}
```

运行结果：

```
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t1] com.bloom.concurrent.three.TestMultiThread - running  
08:47:30 [t2] com.bloom.concurrent.three.TestMultiThread - running
```

3.3. 查看进程线程的方法

3.3.1. Windows

- 任务管理器可以查看进程和线程数，也可以用来杀死进程
- `tasklist` 查看进程
- `taskkill` 杀死进程

3.3.2. Linux

- `ps -ef` 查看所有进程
- `ps -fT -p <PID>` 查看某个进程（PID）的所有线程
- `kill` 杀死进程
- `top` 按大写 H 切换是否显示线程
- `top -H -p <PID>` 查看某个进程（PID）的所有线程

3.3.3. Java

- `jps` 命令查看所有 Java 进程
- `jstack <PID>` 查看某个 Java 进程（PID）的所有线程状态
- `jconsole` 来查看某个 Java 进程中线程的运行情况（图形界面）

3.3.3.1. jconsole远程监控配置

1. 编写自己的程序上传到服务器

```
package com.bloom.concurrent.three;  
  
import java.util.concurrent.TimeUnit;  
  
/**  
 * @description:  
 * @author: teago  
 * @time: 2020/5/16 08:53  
 */  
public class Example4 {  
    public static void main(String[] args) {  
        Thread thread = new Thread(() -> {  
            while (true){  
                System.out.println("test jconsole use");  
                try {  
                    TimeUnit.SECONDS.sleep(1);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }, "t4");  
        thread.start();  
    }  
}
```

2. 使用javac Example4.java 编译程序。

```

item2login.sh 22 root 10.211.55.22 root
Last login: Wed Apr  8 01:46:46 on ttys004
➤ ~ item2login.sh 22 root 10.211.55.22 root
spawn ssh -p 22 root@10.211.55.22
root@10.211.55.22's password:
Last login: Sun Apr  5 19:33:01 2020 from 10.211.55.2
[root@localhost ~]# cd /usr/local/java_thread/
[root@localhost java_thread]# ls
Example4.class  Example4.java
[root@localhost java_thread]# █

```

3. 使用如下代码运行Example4

```

java -Djava.rmi.server.hostname=127.0.0.1
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=1234
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false Example4

```

- 然后出现一下错误：

```

[root@localhost java_thread]# java -Djava.rmi.server.hostname=10.211.55.22 -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=1234 -Dcom.sun.management.jmxremote.ssl=false-Dcom.sun.management.jmxremote.authenticate=false Example4
错误: 找不到口令文件: /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.242.b08-0.el8_1.x86_64/jre/lib/management/jmxremote.password
sun.management.AgentConfigurationException
    at sun.management.jmxremote.ConnectorBootstrap.checkPasswordFile(ConnectorBootstrap.java:563)
    at sun.management.jmxremote.ConnectorBootstrap.startRemoteConnectorServer(ConnectorBootstrap.java:426)
    at sun.management.Agent.startAgent(Agent.java:262)
    at sun.management.Agent.startAgent(Agent.java:452)
[root@localhost java_thread]# █

```

4. 这时候需要进行如下操作：

- 复制 jmxremote.password 文件

```

[root@localhost java_thread]# cp /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.242.b08-0.el8_1.x86_64/jre/lib/management/jmxremote.password.template /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.242.b08-0.el8_1.x86_64/jre/lib/management/jmxremote.password
[root@localhost java_thread]# █

```

- 修改jmxremote.password信息

```

[root@localhost java_thread]# vi /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.242.b08-0.el8_1.x86_64/jre/lib/management/jmxremote.password
[root@localhost java_thread]# █

```

```
controlRole      1234
```

- 再次运行还是出现以下错误：

```

[root@localhost java_thread]# java -Djava.rmi.server.hostname=10.211.55.22 -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=1234 -Dcom.sun.management.jmxremote.ssl=false-Dcom.sun.management.jmxremote.authenticate=false Example4
错误: 必须限制口令文件读取访问权限: /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.242.b08-0.el8_1.x86_64/jre/lib/management/jmxremote.password
sun.management.AgentConfigurationException
    at sun.management.jmxremote.ConnectorBootstrap.checkPasswordFile(ConnectorBootstrap.java:577)
    at sun.management.jmxremote.ConnectorBootstrap.startRemoteConnectorServer(ConnectorBootstrap.java:426)
    at sun.management.Agent.startAgent(Agent.java:262)
    at sun.management.Agent.startAgent(Agent.java:452)
[root@localhost java_thread]# █

```

5. 需要修改jmxremote.password 和 jmxremote.access 文件的权限为600即文件所有者可读写：

```

chmod -R 600 jmxremote.password
chmod -R 600 jmxremote.access

```

再次运行，结果如下所示：

```

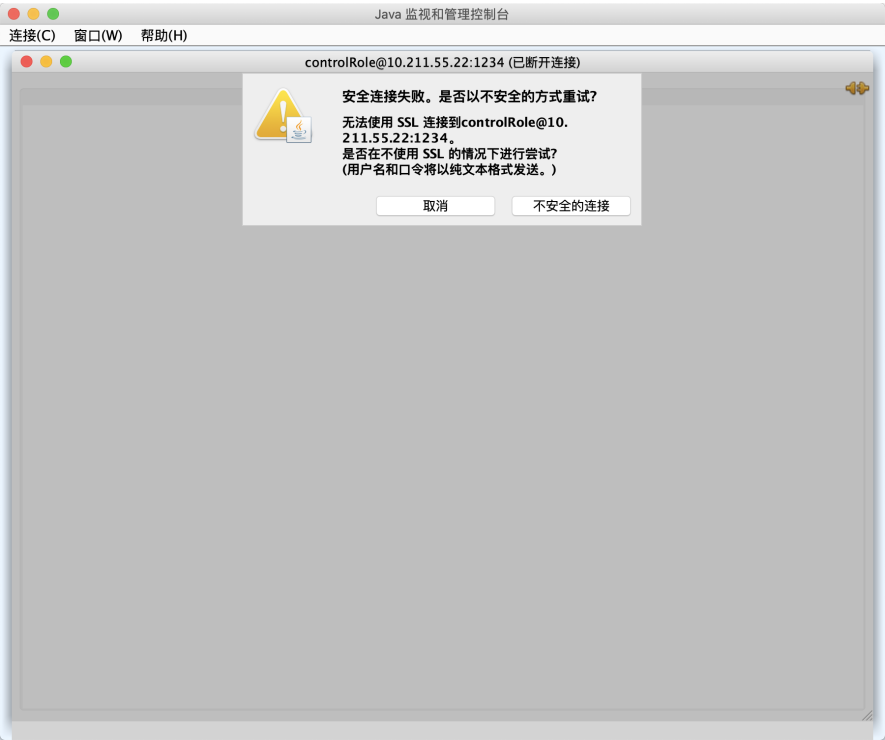
[root@localhost java_thread]# java -Djava.rmi.server.hostname=10.211.55.22 -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=1234 -Dcom.sun.management.jmxremote.ssl=false-Dcom.sun.management.jmxremote.authenticate=false Example4
test jconsole use
test jconsole use
test jconsole use
█

```

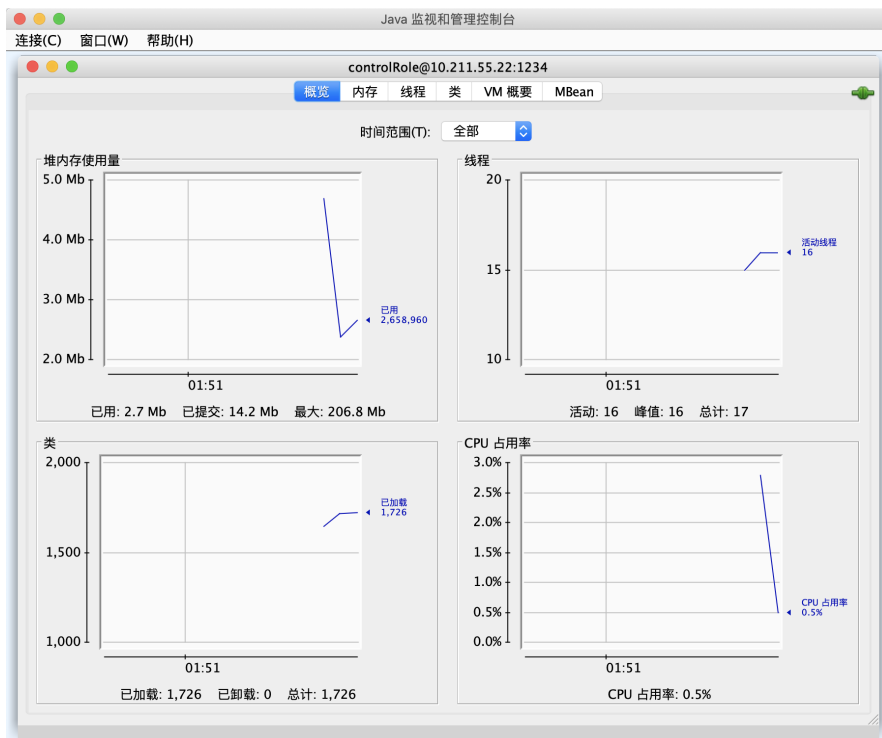
7. 在本地启动jconsole，然后进行如下操作：



- 选择连接-->不安全连接



- 运行如下：



总结:

- 需要以如下方式运行你的 java 类

```
java -Djava.rmi.server.hostname=`ip地址` \
-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=`连接端口` \
-Dcom.sun.management.jmxremote.ssl=是否安全连接 \
-Dcom.sun.management.jmxremote.authenticate=是否认证 java类
```

- 复制 jmxremote.password 文件
- 修改 jmxremote.password 和 jmxremote.access 文件的权限为 600 即文件所有者可读写
- 连接时填入 controlRole (用户名), R&D (密码)

如果要认证访问, 还需要做如下步骤:

- 修改 /etc/hosts 文件将 127.0.0.1 映射至主机名

3.4. 原理之线程运行

3.4.1. 栈与栈帧

Java Virtual Machine Stacks (Java虚拟机栈)

我们都知道JVM中有堆、栈、方法区所组成, 其中栈内存是给谁用的呢? 其实就是线程, 每个线程启动后, 虚拟机都会为其分配一块栈内存。

- 每个栈有多个栈帧 (Frame) 组成, 对应着每次方法调用时所占用的内存
- 每个线程只能有一个活动栈帧, 也就是对应着当前正在执行的那个方法。

3.4.1.1. Debug方式演示栈帧的调用关系

1. 单线程演示:

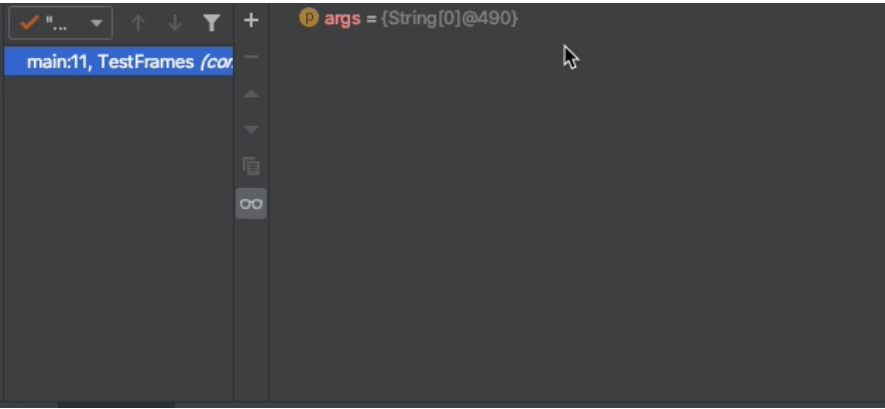
```
package com.bloom.concurrent.three;

/**
 * @description: Debug方式演示栈帧的调用关系
 * @author: teago
 * @time: 2020/5/16 09:06
 */
public class TestFrames {

    public static void main(String[] args) {
        method1(10);
    }
}
```

```
private static void method1(int x) {
    int y = x + 1 ;
    Object m = method2();
    System.out.println(m);
}

private static Object method2() {
    Object n = new Object();
    return n ;
}
}
```



根据上图：首先进入的是main方法的栈，当前活动的栈帧

3.4.2. 线程上线文切换（Thread Context Switch）

因为以下一些原因导致CPU不再执行当前的线程，转而执行另一个线程的代码：

- 被动的：
 - 线程的CPU时间片用完
 - 垃圾回收
 - 有更高优先级的线程需要运行
- 主动的：
 - 线程自己调用了sleep、yield、wait、join、park、synchronized、lock等方法

当Context Switch发生时，需要由操作系统保存当前线程的状态信息，并恢复另一个线程的状态，Java中对应的概念就是**程序计数器**（Program Counter Register），它的作用是记住下一条JVM指令的执行地址，是线程私有的。

- 状态包括：**程序计数器、虚拟机栈中的每个栈帧的信息，如局部变量、操作数栈、返回地址等。**
- Context Switch频繁发生会影响性能。（怎么选择合适的线程数，后序会介绍到。）

3.5. 常见方法

方法名	static	功能说明	注意
start()		启动一个新线程，在新的线程运行 run 方法中的代码	start 方法只是让线程进入就绪，里面代码不一定立刻运行（CPU 的时间片还没分给它）。每个线程对象的start方法只能调用一次，如果调用了多次会出现 IllegalStateException
run()		新线程启动后会调用的方法	如果在构造 Thread 对象时传递了 Runnable 参数，则线程启动后会调用 Runnable 中的 run 方法，否则默认不执行任何操作。但可以创建 Thread 的子类对象，来覆盖默认为
join()		等待线程运行结束	

方法名	static	功能说明	注意
join(long n)		等待线程运行结束,最多等待n 毫秒	
getId()		获取线程长整型的 id	id 唯一
getName()		获取线程名	
setName(String)		修改线程名	
getPriority()		获取线程优先级	
setPriority(int)		修改线程优先级	java中规定线程优先级是1~10 的整数，较大的优先级能提高该线程被 CPU 调度的机率
getState()		获取线程状态	Java 中线程状态是用 6 个 enum 表示，分别为：NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
isInterrupted()		判断是否被打断，	不会清除 打断标记
isAlive()		线程是否存活（还没有运行完毕）	
interrupt()		打断线程	如果被打断线程正在 sleep，wait，join 会导致被打断的线程抛出 InterruptedException，并清除打断标记；如果打断的正在运行的线程，则会设置打断标记；park 的线程被打断，也会设置打断标记
interrupted()	static	判断当前线程是否被打断	会清除 打断标记
currentThread()	static	获取当前正在执行的线程	
sleep(long n)	static	让当前执行的线程休眠n毫秒，休眠时让出 cpu 的时间片给其它线程	
yield()	static	提示线程调度器让出当前线程对CPU的使用	主要是为了测试和调试

3.5.1. start与run

3.5.1.1. 调用run

```
package com.bloom.concurrent.three;
```



```
import lombok.extern.slf4j.Slf4j;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 10:22
 */
@Slf4j
public class Example5 {

    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            log.info(Thread.currentThread().getName() + ">>>>> t1 execute");
        });
        // log.info(thread.getState().name());
        thread.run();
        // log.info(thread.getState().name());
        // thread.start();
        // log.info(thread.getState().name());
    }
}
```

运行结果

```
10:57:48 [main] com.bloom.concurrent.three.Example5 - main>>>>> t1 execute
```

3.5.1.2. 调用start

```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 10:22
 */
@Slf4j
public class Example5 {

    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            log.info(Thread.currentThread().getName() + ">>>>> t1 execute");
        });
        // log.info(thread.getState().name());
        // thread.run();
        // log.info(thread.getState().name());
        thread.start();
        // log.info(thread.getState().name());
    }
}
```

运行结果：

```
10:58:53 [Thread-0] com.bloom.concurrent.three.Example5 - Thread-0>>>>> t1
execute
```

小结

- 直接调用 run 是在主线程中执行了 run，没有启动新的线程
- 使用 start 是启动新的线程，通过新的线程间接执行 run 中的代码

修改以上代码：

```
log.info(thread.getState().name());
thread.run();
log.info(thread.getState().name());
thread.start();
log.info(thread.getState().name());
```

运行结果：(也说明了上述的问题)

```
1:00:04 [main] com.bloom.concurrent.three.Example5 - NEW
11:00:04 [main] com.bloom.concurrent.three.Example5 - main>>>>> t1 execute
11:00:04 [main] com.bloom.concurrent.three.Example5 - NEW
11:00:04 [main] com.bloom.concurrent.three.Example5 - RUNNABLE
11:00:04 [Thread-0] com.bloom.concurrent.three.Example5 - Thread-0>>>>> t1
execute
```

3.5.2. sleep与yield

3.5.2.1. sleep

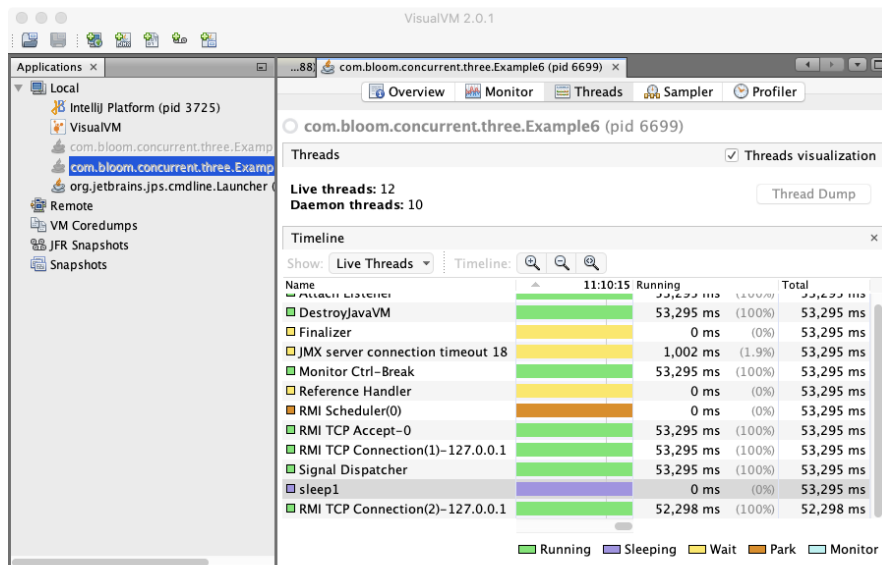
1. 调用sleep会让当前线程从Running进入Timed Waiting状态（阻塞）；

- 但是这种状态只能通过一些工具可以看到，使用如下是不能知道的

```
package com.beatshadow.concurrent.chapter3;

import java.util.concurrent.TimeUnit;

/**
 *
 * @author : <a href="mailto:gnehcgaw@gmail.com">gnehcgaw</a>
 * @since : 2020/4/27 22:12
 */
public class Example6 {
    public static void main(String[] args) {
        new Thread("td1"){
            @Override
            public void run() {
                try {
                    System.out.println(this.getState());
                    TimeUnit.MINUTES.sleep(1);
                    System.out.println(this.getState());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }
}
```



2. 其他线程可以使用interrupt方法打断正在睡眠的线程，这时sleep方法会抛出InterruptedException；

- 不打断的情况

```
package com.bloom.concurrent.three;

import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;
```

```

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 10:47
 */
@Slf4j
public class Example7 {

    @SneakyThrows
    public static void main(String[] args) {
        Thread thread = new Thread("td2"){
            @Override
            public void run() {
                try {
                    log.info("thread execute");
                    Thread.sleep(2000);
                    log.info("thread execute end");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        thread.start();
        log.info("main thread execute start");
        Thread.sleep(1000);
        log.info("main thread execute end");
        // thread_execute.interrupt();
    }
}

```

运行结果如下所示：（程序没有出错，线程在休眠之后继续执行）

```

11:12:31 [main] com.bloom.concurrent.three.Example7 - main thread execute start
11:12:31 [td2] com.bloom.concurrent.three.Example7 - thread execute
11:12:32 [main] com.bloom.concurrent.three.Example7 - main thread execute end
11:12:33 [td2] com.bloom.concurrent.three.Example7 - thread execute end

```

修改程序，添加interrupt()的调用：

```
thread.interrupt();
```

运行结果如下所示：

```

11:13:23 [main] com.bloom.concurrent.three.Example7 - main thread execute start
11:13:23 [td2] com.bloom.concurrent.three.Example7 - thread execute
11:13:24 [main] com.bloom.concurrent.three.Example7 - main thread execute end
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at com.bloom.concurrent.three.Example7$1.run(Example7.java:21)

```

总结：线程被打断，sleep之后的未被执行，因为程序出现异常。

3. 睡眠结束后的线程未必会立刻得到执行；
4. 建议使用TimeUnit的sleep代替Thread的sleep来获得更好的可读性（1.5之后）。

3.5.2.2. yield

让出，谦让的意思。

1. 调用yield会让当前线程从Running进入Runnable就绪状态，然后调度执行其它同优先级的线程。如果这时没有同优先级的线程，那么不能保证让当前线程暂停的效果；
2. 具体的实现依赖于操作系统的任务调度器。（想让没让出去）

3.5.2.3. sleep与yield的区别

sleep让线程从running变成waiting状态，yield让线程从running变成runnable状态；
runnable状态的线程有可能被执行，而waiting状态的线程只有能到线程休眠之后才又可能被执行。

3.5.2.4. 线程优先级（不靠谱）

- 线程优先级会提示（hint）调度器有限调度该线程，但它仅仅是一个提示，调度器可以忽略它；

- 如果CUP比较忙，那么优先级高的线程会获得更多的时间片，但CPU闲时，优先级几乎没有作用。

3.5.2.5. 案例——效率篇：防止CPU占用100%（sleep实现）

示例代码：

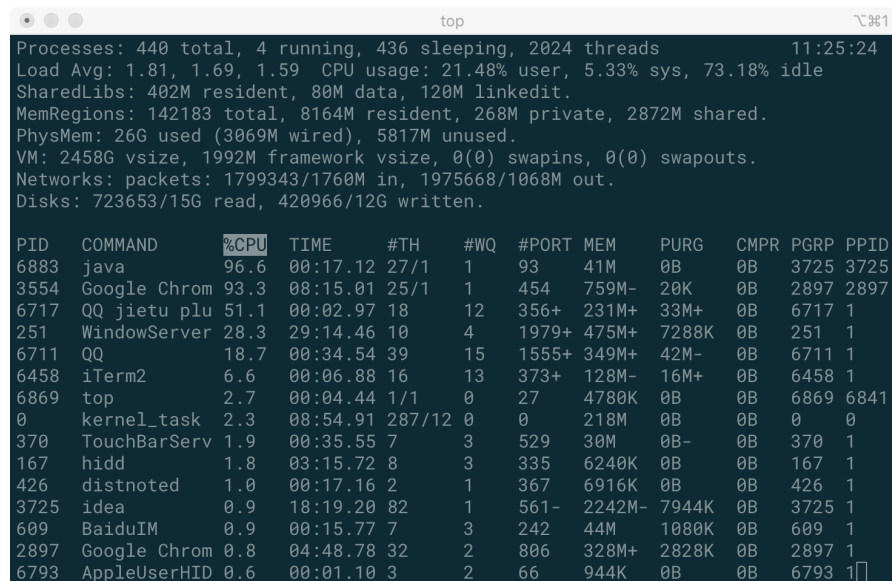
```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 11:23
 */
@Slf4j
public class Example8 {

    public static void main(String[] args) {
        new Thread(() -> {
            while (true){
                //业务代码
            }
        }).start();
    }
}
```

发现CUP占用几乎为100%



```
Processes: 440 total, 4 running, 436 sleeping, 2024 threads      11:25:24
Load Avg: 1.81, 1.69, 1.59  CPU usage: 21.48% user, 5.33% sys, 73.18% idle
SharedLibs: 402M resident, 80M data, 120M linkedit.
MemRegions: 142183 total, 8164M resident, 268M private, 2872M shared.
PhysMem: 26G used (3069M wired), 5817M unused.
VM: 2458G vsize, 1992M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 1799343/1760M in, 1975668/1068M out.
Disks: 723653/15G read, 420966/12G written.

PID   COMMAND     %CPU   TIME    #TH   #WQ   #PORT  MEM    PURG    CMPR  PGRP  PPID
6883   java         96.6   00:17.12 27/1   1     93    41M    0B     0B   3725 3725
3554   Google Chrom 93.3   08:15.01 25/1   1     454    759M-  20K    0B   2897 2897
6717   QQ jietu plu 51.1   00:02.97 18      12    356+   231M+  33M+   0B   6717 1
251    WindowServer 28.3   29:14.46 10      4    1979+  475M+  7288K  0B   251 1
6711   QQ           18.7   00:34.54 39     15    1555+  349M+  42M-   0B   6711 1
6458   iTerm2        6.6   00:06.88 16     13    373+   128M-  16M+   0B   6458 1
6869   top           2.7   00:04.44 1/1     0     27    4780K  0B     0B   6869 6841
0     kernel_task  2.3   08:54.91 287/12 0      0     218M   0B     0B    0    0
370    TouchBarServ 1.9   00:35.55 7       3     529    30M    0B-    0B    370 1
167    hidd          1.8   03:15.72 8       3     335    6240K  0B     0B    167 1
426    distnoted    1.0   00:17.16 2       1     367    6916K  0B     0B    426 1
3725   idea          0.9   18:19.20 82      1     561-   2242M- 7944K  0B   3725 1
609    BaiduIM       0.9   00:15.77 7       3     242    44M    1080K  0B    609 1
2897   Google Chrom 0.8   04:48.78 32      2     806    328M+  2828K  0B   2897 1
6793   AppleUserHID 0.6   00:01.10 3       2     66     944K   0B     0B   6793 1
```

那如何解决呢？

1.sleep实现

让线程睡眠一段时间。

```
package com.bloom.concurrent.three;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 11:26
 */
public class Example9 {

    public static void main(String[] args) {
        new Thread(() -> {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

```
}
```

在没有利用CPU计算时，不要让while(true)空转去浪费cpu，这时可以使用sleep或者yield来让出cpu的使用权给其他程序。

- 可以使用wait或条件变了达到类似的效果
- 不同的是，后两种都需要加锁，并且需要相应的唤醒操作，一般适用于进行同步的场景。
- sleep适用于无需加锁的场景。

3.5.3. join方法详解

```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

import static java.lang.Thread.sleep;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 11:28
 */
@Slf4j
public class Example10 {
    static int r = 0;

    public static void main(String[] args) {
        test1();
    }

    private static void test1() {
        log.debug("开始");
        Thread t1 = new Thread(() -> {
            log.debug("开始");
            try {
                sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.debug("结束");
            r = 10;
        });
        t1.start();
        log.debug("结果为:{}", r);
        log.debug("结束");
    }
}
```

3.5.3.1. 为什么需要join

上面的代码执行，打印 r 是什么？

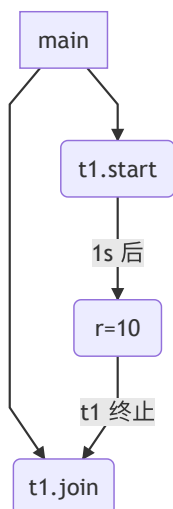
- 因为主线程和线程 t1 是并行执行的，t1 线程需要 1 秒之后才能算出 `r=10`
- 而主线程一开始就要打印 r 的结果，所以只能打印出 `r=0` 解决方法
 - 用 sleep 行不行？为什么？不行 因为sleep方法在睡眠时不释放对象锁
 - 用 join，加在 `t1.start()` 之后即可

3.5.3.2. 应用之同步（案例一）

以调用方角度来讲，如果

- 需要等待结果返回，才能继续运行就是同步

- 不需要等待结果返回，就能继续运行就是异步



3.5.3.3. 等待多个结果

问，下面代码 cost 大约多少秒？

```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

import static java.lang.Thread.sleep;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 16:36
 */
@Slf4j
public class Example11 {
    static int r1 = 0;
    static int r2 = 0;

    public static void main(String[] args) throws InterruptedException {
        test2();
    }

    private static void test2() throws InterruptedException {
        Thread t1 = new Thread(() -> {
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            r1 = 10;
        });
        Thread t2 = new Thread(() -> {
            try {
                sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            r2 = 20;
        });
        long start = System.currentTimeMillis();
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        long end = System.currentTimeMillis();
        log.debug("r1: {} r2: {} cost: {}", r1, r2, end - start);
    }
}
```

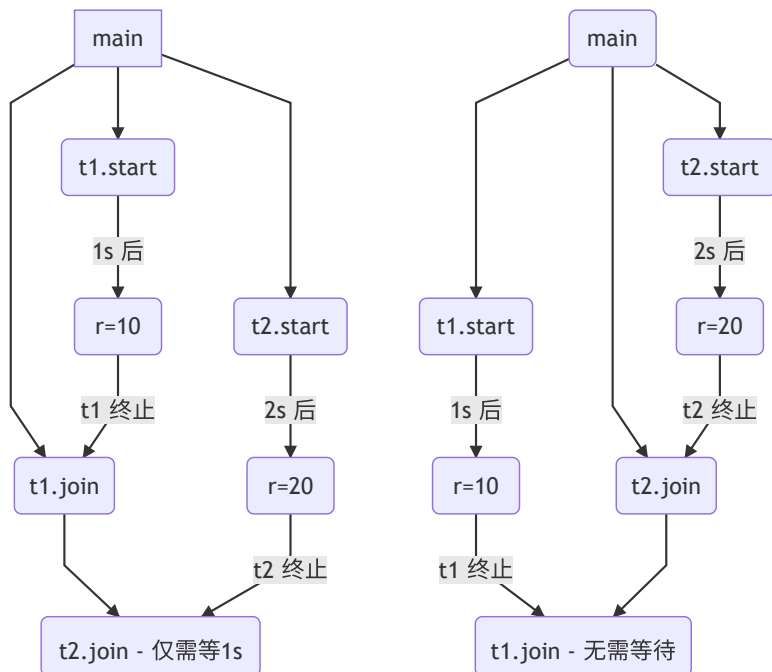
分析如下

- 第一个 join: 等待 t1 时, t2 并没有停止, 而在运行
- 第二个 join: 1s 后, 执行到此, t2 也运行了 1s, 因此也只需再等待 1s

如果颠倒两个 join 呢?

最终都是输出

```
16:38:53 [main] com.bloom.concurrent.three.Example11 - r1: 10 r2: 20 cost: 2003
```



3.5.3.4. 有时效的join

足够时间

```

static int r1 = 0;
static int r2 = 0;
public static void main(String[] args) throws InterruptedException {
    test3();
}
public static void test3() throws InterruptedException {
    Thread t1 = new Thread(() -> {
        sleep(1);
        r1 = 10;
    });

    long start = System.currentTimeMillis();
    t1.start();

    // 线程执行结束会导致 join 结束
    t1.join(1500);
    long end = System.currentTimeMillis();
    log.debug("r1: {} r2: {} cost: {}", r1, r2, end - start);
}
  
```

输出

```
20:48:01.320 [main] c.TestJoin - r1: 10 r2: 0 cost: 1010
```

没等够时间

```

static int r1 = 0;
static int r2 = 0;
public static void main(String[] args) throws InterruptedException {
    test3();
}
public static void test3() throws InterruptedException {
    Thread t1 = new Thread(() -> {
        sleep(2);
        r1 = 10;
    });
  
```

```

long start = System.currentTimeMillis();
t1.start();

// 线程执行结束会导致 join 结束
t1.join(1500);
long end = System.currentTimeMillis();
log.debug("r1: {} r2: {} cost: {}", r1, r2, end - start);
}

```

输出

```
20:52:15.623 [main] c.TestJoin - r1: 0 r2: 0 cost: 1502
```

3.6. interrupt方法详解

3.6.1. 打断sleep、wait、join的线程（阻塞）

这几个方法都会让线程进入阻塞状态

打断 sleep 的线程, 会清空打断状态（即：Thread.isInterrupt()= false），以 sleep 为例

```

package com.bloom.concurrent.three;

import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 16:44
 */
@Slf4j
public class Example12 {

    @SneakyThrows
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "t1");
        log.info("t1 start");
        t1.start();
        TimeUnit.SECONDS.sleep(1);
        t1.interrupt();
        log.info("t1 interrupt");
        log.info(t1.isInterrupted() + "");
    }
}

```

运行结果：

```

16:48:55 [main] com.bloom.concurrent.three.Example12 - t1 start
16:48:56 [main] com.bloom.concurrent.three.Example12 - t1 interrupt
16:48:56 [main] com.bloom.concurrent.three.Example12 - false
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:340)
    at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
    at com.bloom.concurrent.three.Example12.lambda$main$0(Example12.java:20)
    at java.lang.Thread.run(Thread.java:748)

Process finished with exit code 0

```

3.6.2. 打断正常运行的线程

```

package com.beatshadow.concurrent.chapter3;

```

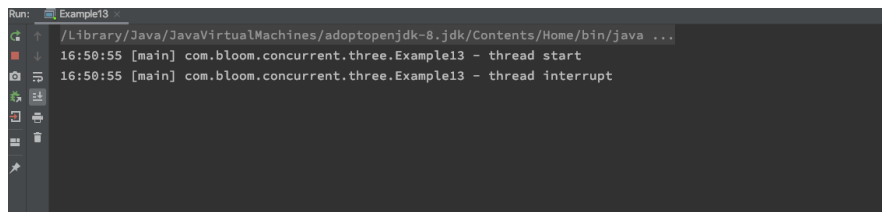


```
import lombok.extern.slf4j.Slf4j;

/**
 * @author : <a href="mailto:gnehcgnew@gmail.com">gnehcgnew</a>
 * @since : 2020/4/28 11:58
 */
@Slf4j
public class Example13 {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            while (true) {

            }
        });
        log.debug("thread start");
        thread.start();
        log.debug("thread interrupt");
        thread.interrupt();
    }
}
```

运行结果：



```
Run: Example13
/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
16:50:55 [main] com.bloom.concurrent.three.Example13 - thread start
16:50:55 [main] com.bloom.concurrent.three.Example13 - thread interrupt
```

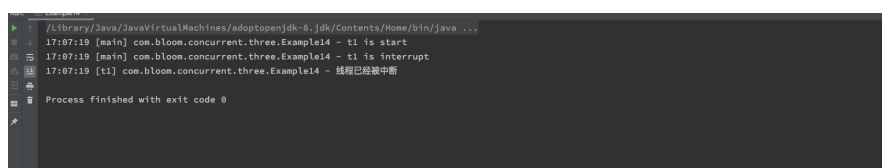
正常运行的线程如果被打断，线程并不会直接结束运行，只是说将interrupt的状态置为了true，这时候需要我们认为的代码进行干涉从而结束线程。

修改程序，使其优雅打断执行的线程：

```
package com.beatshadow.concurrent.chapter3;

import lombok.extern.slf4j.Slf4j;

/**
 * @author : <a href="mailto:gnehcgnew@gmail.com">gnehcgnew</a>
 * @since : 2020/4/28 11:58
 */
@Slf4j
public class Example13 {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            while (true) {
                //认为打断线程
                if (Thread.currentThread().isInterrupted()) {
                    log.debug("线程被打断");
                    break;
                }
            }
        });
        log.debug("thread start");
        thread.start();
        log.debug("thread interrupt");
        //这时候用户线程并未结束，只是说被打断的线程的interrupt=true，这时候需要我们根据这个状态进行人工干涉
        thread.interrupt();
    }
}
```



```
Run: Example13
/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
17:07:19 [main] com.bloom.concurrent.three.Example14 - t1 is start
17:07:19 [main] com.bloom.concurrent.three.Example14 - t1 is interrupt
17:07:19 [t1] com.bloom.concurrent.three.Example14 - 线程已经被中断
Process finished with exit code 0
```

3.6.3. 模式之两阶段终止模式

Two Phase Termination

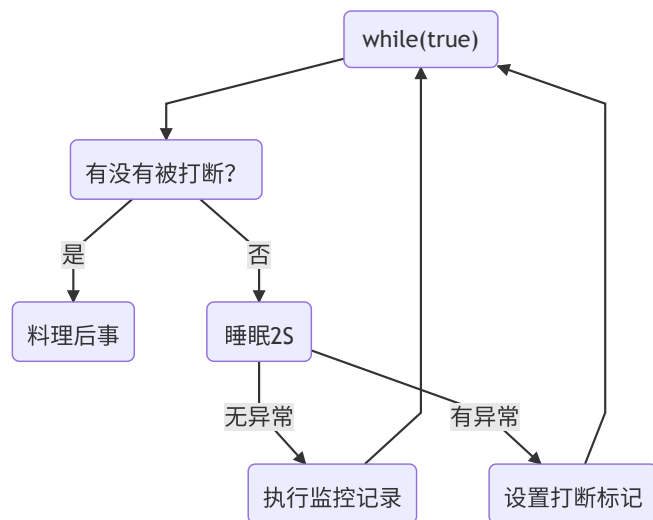
在一个线程T1只如何“优雅”终止线程T2？这里的【优雅】指的是给T2一个料理后事的机会。

3.6.3.1. 错误思路

1. 使用线程对象的stop()方法停止线程
 - stop方法会真正杀死线程，如果这时线程锁住了共享资源，那么当它被杀死后就再也没有机会释放锁，其他线程永远无法获取锁。
2. 使用System.exit(int)方法停止线程
 - 目的仅是停止一个线程，但是这种做法会让整个程序都停止。

3.6.3.2. 两阶段终止模式

Java多线程编程实战指南——两阶段终止模式



演示代码:

```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 17:09
 */
public class Example15 {

    public static void main(String[] args) {
        TwoPhaseTermination twoPhaseTermination = new TwoPhaseTermination();
        twoPhaseTermination.start("demo");
        try {
            Thread.sleep(3500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        twoPhaseTermination.stop();
    }

}

@Slf4j
class TwoPhaseTermination {
    private Thread thread;

    public void start(String threadName) {
        log.info(threadName + "线程启动");
        thread = new Thread(() -> {
            while (true) {
                if (thread.isInterrupted()) {
                    log.info("料理后事");
                    break;
                } else {
                    try {
                        TimeUnit.SECONDS.sleep(2); // 打断1
                        log.info("执行监控记录");
                    } catch (InterruptedException e) {
```

```

        e.printStackTrace();
        // 重新设置打断标记
        thread.interrupt();
    }
}

}, threadName);
thread.start();

}

public void stop() {
    if (null != thread) {
        thread.interrupt();
    }
}

}

```

运行结果：

```

17:16:17 [main] com.bloom.concurrent.three.TwoPhaseTermination - demo线程启动
17:16:19 [demo] com.bloom.concurrent.three.TwoPhaseTermination - 执行监控记录
java.lang.InterruptedExecution: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:340)
    at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
    at
com.bloom.concurrent.three.TwoPhaseTermination.lambda$start$0(Example15.java:40)
    at java.lang.Thread.run(Thread.java:748)
17:16:21 [demo] com.bloom.concurrent.three.TwoPhaseTermination - 料理后事

```

3.6.4. 打断park线程

park: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/LockSupport.html#park->

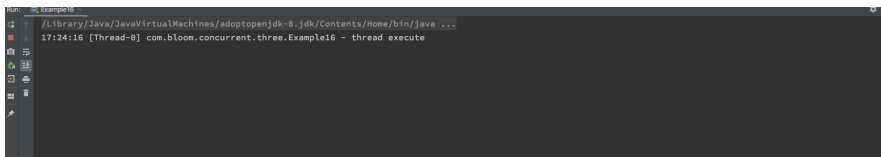
=

```

Thread thread = new Thread() -> {
    log.debug("thread execute");
    LockSupport.park();
    log.debug("unpark");
    log.debug("continue to execute");
});
thread.start();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
// thread.interrupt();
}

```

运行结果：



使用了park()方法，以下的程序不会执行。

使用Thread.interrupt()可以打断park的中断

```

package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.locks.LockSupport;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 17:23
 */
@Slf4j

```

```

public class Example16 {

    public static void main(String[] args) {
        Thread thread = new Thread() -> {
            log.debug("thread execute");
            LockSupport.park();
            log.debug("unPark");
            log.debug("continue to execute");
        });
        thread.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread.interrupt();
    }
}

```

运行结果：

```

/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
17:25:22 [Thread-0] com.bloom.concurrent.three.Example16 - thread execute
17:25:23 [Thread-0] com.bloom.concurrent.three.Example16 - unPark
17:25:23 [Thread-0] com.bloom.concurrent.three.Example16 - continue to execute
Process finished with exit code 0

```

但是被中断的线程如果interrupt = true，后续要执行park，则不会被停止中断

演示代码：

```

package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.locks.LockSupport;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 17:23
 */
@Slf4j
public class Example16 {

    public static void main(String[] args) {
        Thread thread = new Thread() -> {
            log.debug("thread execute");
            LockSupport.park();
            log.debug("unPark");
            log.debug("continue to execute");
            LockSupport.park();
            log.debug("seconds continue to execute ");
        });
        thread.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread.interrupt();
    }
}

```

运行结果：（发现第二次调用park(), 并没有被终止后续代码的执行）

```

/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
17:28:04 [Thread-0] com.bloom.concurrent.three.Example16 - thread execute
17:28:05 [Thread-0] com.bloom.concurrent.three.Example16 - unPark
17:28:05 [Thread-0] com.bloom.concurrent.three.Example16 - continue to execute
17:28:05 [Thread-0] com.bloom.concurrent.three.Example16 - seconds continue to execute
Process finished with exit code 0

```

这时候需要使用Thread.interrupted()清除打断标记，具体如下所示：

```

package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.locks.LockSupport;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 17:23
 */
@Slf4j
public class Example16 {

    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            log.debug("thread execute");
            LockSupport.park();
            //清除打断标记
            Thread.interrupted();
            log.debug("unPark");
            log.debug("continue to execute");
            LockSupport.park();
            log.debug("seconds continue to execute ");
        });
        thread.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread.interrupt();
    }
}

```

运行结果：

```

17:29:46 [Thread-0] com.bloom.concurrent.three.Example16 - thread execute
17:29:47 [Thread-0] com.bloom.concurrent.three.Example16 - unPark
17:29:47 [Thread-0] com.bloom.concurrent.three.Example16 - continue to execute

```

3.7. 不推荐使用的方法

还有一些不推荐使用的方法，这些方法已过时，容易破坏同步代码块，造成线程死锁

方法名	static	功能说明
stop()		停止线程运行
suspend()		挂起（暂停）线程运行
resume()		恢复线程运行

可以使用interrupt处理

3.8. 主线程与守护线程

默认情况下，Java 进程需要等待所有线程都运行结束，才会结束。

```

package com.beatshadow.concurrent.chapter3;

import lombok.extern.slf4j.Slf4j;

/**
 * @author : <a href="mailto:gnehcgaw@gmail.com">gnehcgaw</a>
 * @since : 2020/4/28 14:45
 */
@Slf4j

```

```

public class Example16 {
    public static void main(String[] args) {
        Thread thread = new Thread() -> {
            while (true){

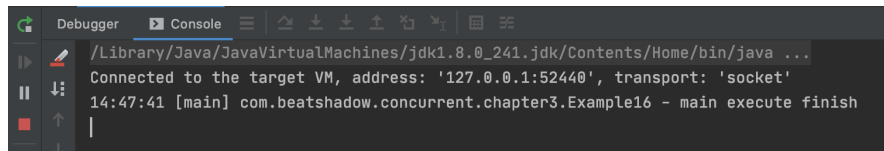
            }

        };

        thread.start();
        try {
            Thread.sleep(1000);
            log.debug("main execute finish");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

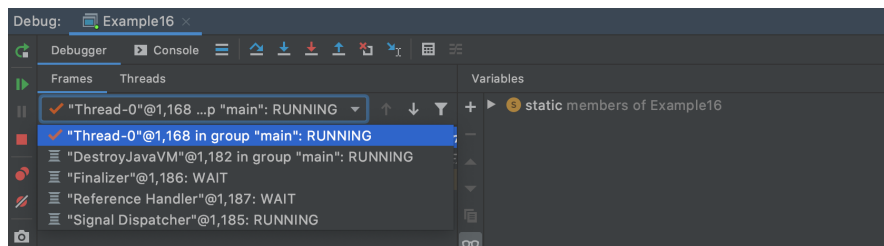
```

运行结果：



原因：（主线程执行完成，但是进程并未停止，原因是thread线程还在执行）

验证：在main线程执行完之后，在if处打断点，有如下情况：



说明：此时thread线程还在执行。

那如何结束这个线程呢？

设置为守护线程，有一种特殊的线程叫做守护线程，只要其它非守护线程运行结束了，即使守护线程的代码没有执行完，也会强制结束。

使用thread.setDaemon(true);

```

package com.beatshadow.concurrent.chapter3;

import lombok.extern.slf4j.Slf4j;

/**
 * @author : <a href="mailto:gnehcgaw@gmail.com">gnehcgaw</a>
 * @since : 2020/4/28 14:45
 */
@Slf4j
public class Example16 {
    public static void main(String[] args) {
        Thread thread = new Thread() -> {
            while (true){
                if (Thread.currentThread().isInterrupted()){
                    break;
                }
            }
        };

        //设置当前线程为守护线程，默认为false
        thread.setDaemon(true);
        thread.start();
        try {
            Thread.sleep(1000);
            log.debug("main execute finish");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

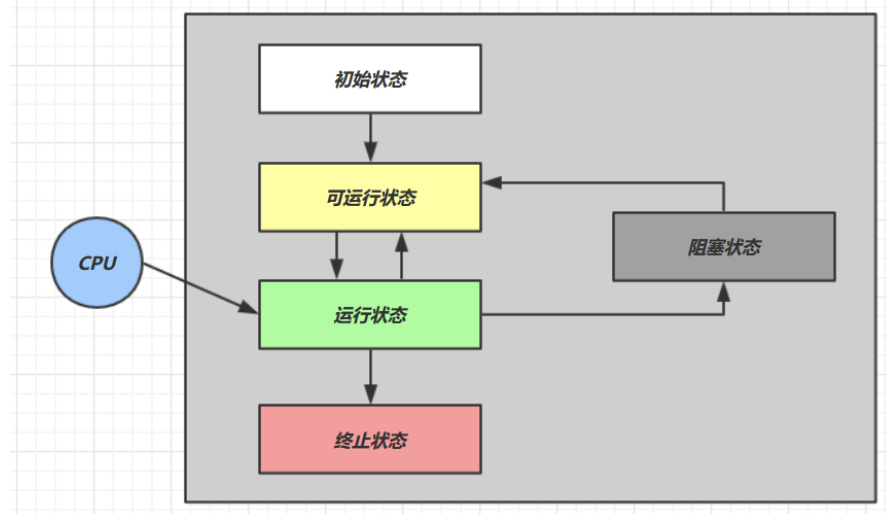
```

****注意****

- 垃圾回收器线程就是一种守护线程(当线程结束之后,就不需要进行垃圾回收了)
- Tomcat 中的 Acceptor 和 Poller 线程都是守护线程,所以 Tomcat 接收到 shutdown 命令后,不会等待它们处理完当前请求

3.9. 线程——五种状态（操作系统层面）

这是从 **操作系统** 层面来描述的



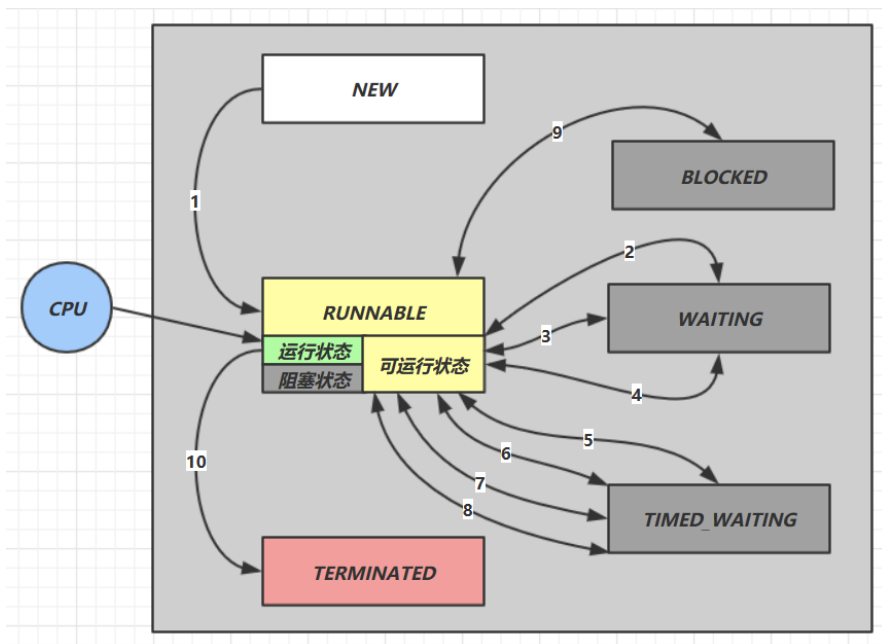
- 【初始状态】仅是在语言层面创建了线程对象,还未与操作系统线程关联
- 【可运行状态】（就绪状态）指该线程已经被创建（与操作系统线程关联），可以由 CPU 调度执行
- 【运行状态】指获取了 CPU 时间片运行中的状态
 - 当 CPU 时间片用完,会从【运行状态】转换至【可运行状态】,会导致线程的上下文切换
- 【阻塞状态】
 - 如果调用了阻塞 API,如 BIO 读写文件,这时该线程实际不会用到 CPU,会导致线程上下文切换,进入【阻塞状态】
 - 等 BIO 操作完毕,会由操作系统唤醒阻塞的线程,转换至【可运行状态】
 - 与【可运行状态】的区别是,对【阻塞状态】的线程来说只要它们一直不唤醒,调度器就一直不会考虑调度它们
- 【终止状态】表示线程已经执行完毕,生命周期已经结束,不会再转换为其它状态

3.10. 线程——六种状态

3.10.1. 概述

这是从 **Java API** 层面来描述的

根据 Thread.State 枚举,分为六种状态



- **NEW** 线程刚被创建，但是还没有调用 **start()** 方法
- **RUNNABLE** 当调用了 **start()** 方法之后，注意，**Java API** 层面的 **RUNNABLE** 状态涵盖了 **操作系统** 层面的【可运行状态】、【运行状态】和【阻塞状态】（由于 BIO 导致的线程阻塞，在 Java 里无法区分，仍然认为是可运行）
- **BLOCKED**，**WAITING**，**TIMED_WAITING** 都是 **Java API** 层面对【阻塞状态】的细分，后面会在状态转换一节详述
- **TERMINATED** 当线程代码运行结束

3.10.2. 演示代码

```

package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

/**
 * @author : <a href="mailto:gnehcgcnaw@gmail.com">gnehcgcnaw</a>
 * @since : 2020/4/28 15:21
 */
@Slf4j
public class TestState {
    public static void main(String[] args) {

        Thread thread1 = new Thread(() -> {

        }, "t1");

        Thread thread2 = new Thread(() -> {
            while (true) {

            }
        }, "t2");
        thread2.start();

        Thread thread3 = new Thread(() -> {
            try {
                thread2.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "t3");
        thread3.start();

        Thread thread4 = new Thread(() -> {
            synchronized (TestState.class) {
                try {
                    Thread.sleep(100000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "t4");
    }
}

```



```
thread4.start();

Thread thread5 = new Thread(() -> {
    synchronized (TestState.class) {
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}, "t5");
thread5.start();

Thread thread6 = new Thread(() -> {

}, "t6");
thread6.start();

log.debug(String.valueOf(thread1.getState()));
log.debug(String.valueOf(thread2.getState()));
log.debug(String.valueOf(thread3.getState()));
log.debug(String.valueOf(thread4.getState()));
log.debug(String.valueOf(thread5.getState()));
log.debug(String.valueOf(thread6.getState()));

}
}
```

运行结果：

```
18:38:47 [main] com.bloom.concurrent.three.TestState - NEW
18:38:47 [main] com.bloom.concurrent.three.TestState - RUNNABLE
18:38:47 [main] com.bloom.concurrent.three.TestState - WAITING
18:38:47 [main] com.bloom.concurrent.three.TestState - TIMED_WAITING
18:38:47 [main] com.bloom.concurrent.three.TestState - BLOCKED
18:38:47 [main] com.bloom.concurrent.three.TestState - TERMINATED
```

3.11. 应用之统筹

3.11.1. 分析

阅读华罗庚《统筹方法》，给出烧水泡茶的多线程解决方案，提示

- 参考图二，用两个线程（两个人协作）模拟烧水泡茶过程
 - 文中办法乙、丙都相当于任务串行
 - 而图一相当于启动了 4 个线程，有点浪费
- 用 `sleep(n)` 模拟洗茶壶、洗水壶等耗费的时间

附：华罗庚《统筹方法》

统筹方法，是一种安排工作进程的数学方法。它的实用范围极广泛，在企业管理和基本建设中，以及关系复杂的科研项目的组织与管理中，都可以应用。

怎样应用呢？主要是把工序安排好。

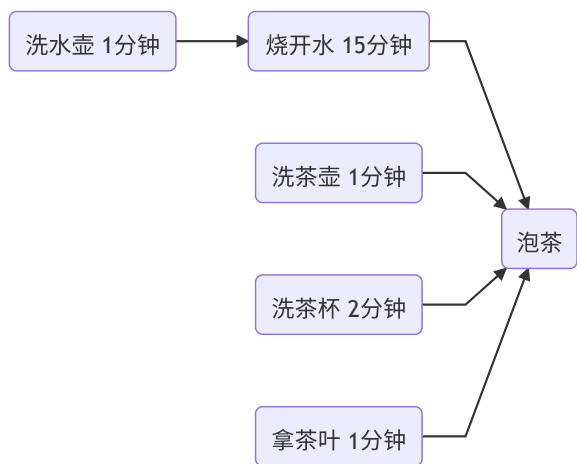
比如，想泡茶喝茶。当时的情况是：开水没有；水壶要洗，茶壶、茶杯要洗；火已生了，茶叶也有了。怎么办？

- 办法甲：洗好水壶，灌上凉水，放在火上；在等待水开的时间里，洗茶壶、洗茶杯、拿茶叶；等水开了，泡茶喝。
- 办法乙：先做好一些准备工作，洗水壶，洗茶壶茶杯，拿茶叶；一切就绪，灌水烧水；坐待水开了，泡茶喝。
- 办法丙：洗净水壶，灌上凉水，放在火上，坐待水开；水开了之后，急急忙忙找茶叶，洗茶壶茶杯，泡茶喝。

哪一种办法省时间？我们能一眼看出，第一种办法好，后两种办法都窝了工。

这是小事，但这是引子，可以引出生产管理等方面有用的方法来。

水壶不洗，不能烧开水，因而洗水壶是烧开水的前提。没开水、没茶叶、不洗茶壶茶杯，就不能泡茶，因而这些又是泡茶的前提。它们的相互关系，可以用下边的箭头图来表示：

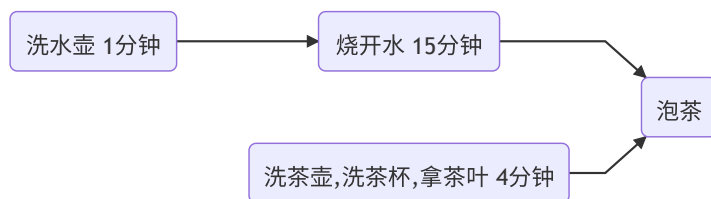


> 从这个图上可以一眼看出，

办法甲总共要16分钟（而办法乙、丙需要20分钟）。如果要缩短工时、提高工作效率，应当主要抓烧开水这个环节，而不是抓拿茶叶等环节。同时，洗茶壶茶杯、拿茶叶总共不过4分钟，大可利用“等水开”的时间来做。

是的，这好像是废话，卑之无甚高论。有如走路要用两条腿走，吃饭要一口一口吃，这些道理谁都懂得。但稍有变化，临事而迷的情况，常常是存在的。在近代工业的错综复杂的工艺过程中，往往就不是像泡茶喝这么简单了。任务多了，几百几千，甚至有好几万个任务。关系多了，错综复杂，千头万绪，往往出现“万事俱备，只欠东风”的情况。由于一两个零件没完成，耽误了一台复杂机器的出厂时间。或往往因为抓的不是关键，连夜三班，急急忙忙，完成这一环节之后，还得等待旁的环节才能装配。

洗茶壶，洗茶杯，拿茶叶，或先或后，关系不大，而且同是一个人的活儿，因而可以合并成为：



看来这是“小题大做”，但在工作环节太多的时候，这样做就非常必要了。

这里讲的主要是时间方面的事，但在具体生产实践中，还有其他方面的许多事。这种方法虽然不一定能直接解决所有问题，但是，我们利用这种方法来考虑问题，也是不无裨益的。

3.11.2. 实现

3.11.2.1. join方式实现

```
package com.bloom.concurrent.three;

import lombok.extern.slf4j.Slf4j;

/**
 * @description:
 * @author: teago
 * @time: 2020/5/16 18:44
 */
@Slf4j
public class Example18 {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            try {
                log.debug("洗水壶");
                Thread.sleep(5000);
                log.debug("烧开水");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "老王");

        Thread thread2 = new Thread(() -> {
            try {
```

```
        log.debug("洗茶壶");
        Thread.sleep(1000);
        log.debug("拿茶叶");
        thread1.join();
        log.debug("泡茶");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}, "小王");

thread1.start();
thread2.start();
}
}
```

运行结果：

```
18:45:26 [老王] com.bloom.concurrent.three.Example18 - 洗水壶
18:45:26 [小王] com.bloom.concurrent.three.Example18 - 洗茶壶
18:45:27 [小王] com.bloom.concurrent.three.Example18 - 拿茶叶
18:45:31 [老王] com.bloom.concurrent.three.Example18 - 烧开水
18:45:31 [小王] com.bloom.concurrent.three.Example18 - 泡茶
```

缺陷：（后续改进）

- 上面模拟的是小王等老王水烧开了，小王再去泡茶，如果反过来要实现老王等小王的茶叶拿来了，老王泡茶呢？代码最好能适应这两种情况。
- 上面的两个线程其实是各执行各自的，如果要模拟老王吧水壶交给小王泡茶呢，或模拟小王吧茶叶交给老王泡茶呢？

3.12. 本章小节

本章的重点在于掌握

- 线程创建
- 线程重要 api，如 start，run，sleep，join，interrupt 等
- 线程状态
- 应用方面
 - 异步调用：主线程执行期间，其它线程异步执行耗时操作
 - 提高效率：并行计算，缩短运算时间
 - 同步等待：join
 - 统筹规划：合理使用线程，得到最优效果
- 原理方面
 - 线程运行流程：栈、栈帧、上下文切换、程序计数器
 - Thread 两种创建方式 的源码
- 模式方面
 - 终止模式之两阶段终止

分类： 并发

标签： 并发， Java

好文要顶

关注成功

收藏该文



teago

粉丝 - 1 关注 - 0

关注成功

1

推荐

0

反对


支持成功 撤回

升级成为会员

« 上一篇： Java并发-2 进程与线程





» 下一篇： Java并发-4 共享模型编程👤

posted @ 2020-05-16 18:48 teago 阅读(188) 评论(0) 编辑 收藏 举报


 发表评论 [升级成为园子VIP会员](#)

编辑

预览

B

支持 Markdown

 自动补全

[提交评论](#) [退出](#) [订阅评论](#) [我的博客](#)

[Ctrl+Enter快捷键提交]

- 【推荐】通义灵码，灵动指间，快码加编，你的智能编码助手
- 【推荐】编程路上的催化剂：大道至简，给所有人看的编程书
- 【推荐】阿里云云市场联合博客园推出开发者商店，欢迎关注
- 【推荐】阿里云暖冬特惠，2核2G轻量应用服务器首购61元/年



- 编辑推荐：
- 一个例子形象地理解同步与异步
 - C# 线程本地存储 为什么线程间值不一样
 - [动画进阶] 神奇的 3D 卡片反光闪烁动效
 - 记一次缓存失效引发的惨案！
 - 记一次 .NET某MES自动化桌面程序 卡死分析

- 阅读排行：
- 通义灵码，降临博客园
 - 他凌晨1:30给我开源的游戏加了UI | 模拟龙生，挂机冒险
 - C# 线程本地存储 为什么线程间值不一样
 - .NET集成IdGenerator生成分布式全局唯一ID
 - 什么是 doris，为什么几乎国内大厂都会使用它