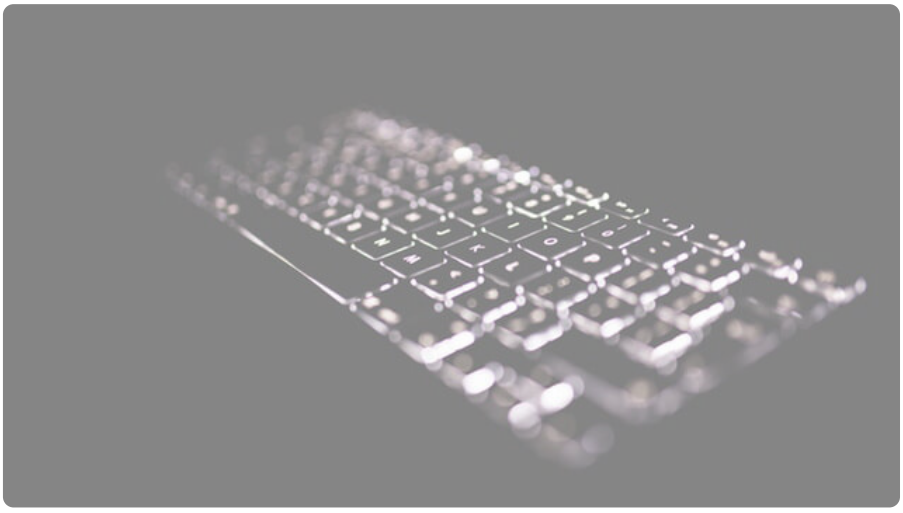
目录



# 27 Thread 源码解析

更新时间：2019-10-31 20:19:55



书籍乃世人积累智慧之长明灯。

——寇第斯



## 引导语

从本章开始我们开始学习线程的知识，线程是非常有趣的一个章节，大多数同学对于线程 API，属于不用就忘，到用时需要百度的情况，希望通过本小节的源码阅读，能够加深对线程的印象。

本小节主要三章，本章主要说线程的基本概念、使用姿势、Thread 和 Runnable 的源码；Future、ExecutorService 源码解析章节主要说异步线程执行；押宝线程源码面试题章节主要说说常遇到的源码面试题。

由于线程的概念很多，所以本章会先介绍很多线程的基本概念，说清楚后再解析源码，不然有些同学会看不懂，大家见谅。

## 1 类注释

### 1.1 Thread

1. 每个线程都有优先级，高优先级的线程可能会优先执行；

2. 父线程创建子线程后，优先级、是否是守护线程等属性父子线程是一致的；

3. JVM 启动时，通常都启动 MAIN 非守护线程，以下任意一个情况发生时，线程就会停止：

   退出方法被调用，并且安全机制允许这么做（比如调用 Thread.interrupt 方法）；

   所有非守护线程都消亡，或者从运行的方法正常返回，或者运行的方法抛出了异常；

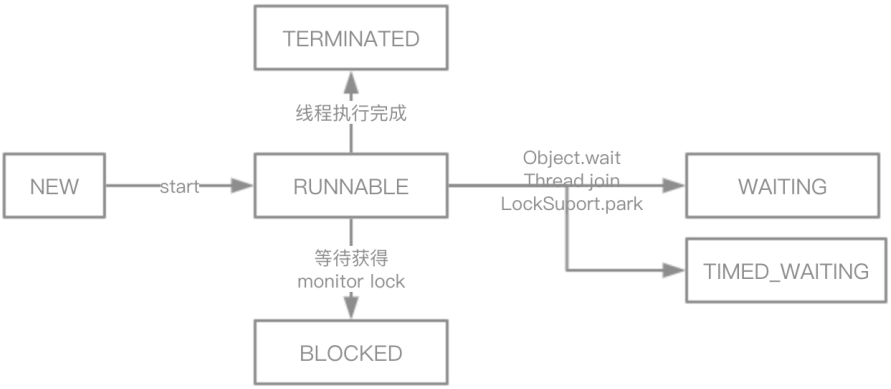
目录

## 2 线程的基本概念

我们接下来介绍一下线程的基本概念：

### 2.1 线程的状态

网上有各种介绍线程状态的文章，我们这里说线程的状态是从源码的角度，源码中一共列举了六种状态，如下图：




我们解析一下这个图：

1. NEW 表示线程创建成功，但没有运行，在 new Thread 之后，没有 start 之前，线程的状态都是 NEW；
2. 当我们运行 start 方法，子线程被创建成功之后，子线程的状态变成 RUNNABLE，RUNNABLE 表示线程正在运行中；
3. 子线程运行完成、被打断、被中止，状态都会从 RUNNABLE 变成 TERMINATED，TERMINATED 表示线程已经运行结束了；
4. 如果线程正好在等待获得 monitor lock 锁，比如在等待进入 synchronized 修饰的代码块或方法时，会从 RUNNABLE 变成 BLOCKED，BLOCKED 表示阻塞的意思；
5. WAITING 和 TIMED_WAITING 类似，都表示在遇到 Object#wait、Thread#join、LockSupport#park 这些方法时，线程就会等待另一个线程执行完特定的动作之后，才能结束等待，只不过 TIMED_WAITING 是带有等待时间的（可以看下面的 join 方法的 demo）。



再次重申，这 6 种状态并不是线程所有的状态，只是在 Java 源码中列举出的 6 种状态， Java 线程的处理方法都是围绕这 6 种状态的。

### 2.2 优先级

优先级代表线程执行的机会的大小，优先级高的可能先执行，低的可能后执行，在 Java 源码中，优先级从低到高分别是 1 到 10，线程默认 new 出来的优先级都是 5，源码如下：

```
// 最低优先级
public final static int MIN_PRIORITY = 1;

// 普通优先级，也是默认的
public final static int NORM_PRIORITY = 5;
```

目录

### 2.3 守护线程

我们默认创建的线程都是非守护线程。创建守护线程时，需要将 Thread 的 daemon 属性设置成 true，守护线程的优先级很低，当 JVM 退出时，是不关心有无守护线程的，即使还有很多守护线程，JVM 仍然会退出，我们在工作中，可能会写一些工具做一些监控的工作，这时我们都是用守护子线程去做，这样即使监控抛出异常，但因为是子线程，所以也不会影响到业务主线程，因为是守护线程，所以 JVM 也无需关注监控是否正在运行，该退出就退出，所以对业务不会产生任何影响。

### 2.4 ClassLoader

ClassLoader 我们可以简单理解成类加载器，就是把类从文件、二进制数组、URL 等位置加载成可运行 Class。

## 3 线程两种初始化方式

无返回值的线程主要有两种初始化方式：

### 3.1 继承 Thread，成为 Thread 的子类

```
// 继承 Thread，实现其 run 方法
class MyThread extends Thread{
  @Override
  public void run(){
    log.info(Thread.currentThread().getName());
  }
}
@Test
// 调用 start 方法即可，会自动调用到 run 方法的
public void extendThreadInit(){
  new MyThread().start();
}
```



上述代码打印出的线程名称是：Thread-0，而主线程的名字是：Thread [main,5,main]，由此可见，的确是开了一个子线程来执行打印的操作。

我们一起来看下 start 的底层源码：

```
// 该方法可以创建一个新的线程出来
public synchronized void start() {
    // 如果没有初始化，抛异常
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    group.add(this);
    // started 是个标识符，我们在做一些事情的时候，经常这么写
    // 动作发生之前标识符是 false，发生完成之后变成 true
    boolean started = false;
    try {
        // 这里会创建一个新的线程，执行完成之后，新的线程已经在运行了，既 target 的内容已经在这
        start0();
        // 这里执行的还是主线程
        started = true;
    } finally {
        try {
```

```
        }
      // Throwable 可以捕捉一些 Exception 捕捉不到的异常，比如说子线程抛出的异常
      } catch (Throwable ignore) {
        /* do nothing. If start0 threw a Throwable then
          it will be passed up the call stack */
      }
    }
}
// 开启新线程使用的是 native 方法
private native void start0();
```

### 3.2 实现 Runnable 接口，作为 Thread 的入参

```
Thread thread = new Thread(new Runnable() {
  @Override
  public void run() {
    log.info("{} begin run",Thread.currentThread().getName());
  }
});
// 开一个子线程去执行
thread.start();
// 不会新起线程，是在当前主线程上继续运行
thread.run();
```

这种就是实现 Runnable 的接口，并作为 Thread 构造器的入参，我们调用时使用了两种方式，可以根据情况选择使用 start 或 run 方法，使用 start 会开启子线程来执行 run 里面的内容，使用 run 方法执行的还是主线程。



我们来看下 run 方法的源码：

```
// 简单的运行，不会新起线程，target 是 Runnable
public void run() {
    if (target != null) {
        target.run();
    }
}
```

源码中的 target 就是在 new Thread 时，赋值的 Runnable。

## 4 线程初始化

线程初始化的源码有点长，我们只看比较重要的代码 (不重要的被我删掉了)，如下：

```
// 无参构造器，线程名字自动生成
public Thread() {
    init(null, null, "Thread-" + nextThreadNum(), 0);
}
// g 代表线程组，线程组可以对组内的线程进行批量的操作，比如批量的打断 interrupt
// target 是我们要运行的对象
// name 我们可以自己传，如果不传默认是 "Thread-" + nextThreadNum()，nextThreadNum 方法
// stackSize 可以设置堆栈的大小
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc) {
    if (name == null) {
        throw new NullPointerException("name cannot be null");
```

```
    // 当前线程作为父线程
    Thread parent = currentThread();
    this.group = g;
    // 子线程会继承父线程的守护属性
    this.daemon = parent.isDaemon();
    // 子线程继承父线程的优先级属性
    this.priority = parent.getPriority();
    // classLoader
    if (security == null || isCCLOverridden(parent.getClass()))
        this.contextClassLoader = parent.getContextClassLoader();
    else
        this.contextClassLoader = parent.contextClassLoader;
    this.inheritedAccessControlContext =
            acc != null ? acc : AccessController.getContext();
    this.target = target;
    setPriority(priority);
    // 当父线程的 inheritableThreadLocals 的属性值不为空时
    // 会把 inheritableThreadLocals 里面的值全部传递给子线程
    if (parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
            ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    this.stackSize = stackSize;
    /* Set thread ID */
    // 线程 id 自增
    tid = nextThreadID();
}
```



从初始化源码中可以看到，很多属性，子线程都是直接继承父线程的，包括优先性、守护线程、inheritableThreadLocals 里面的值等等。

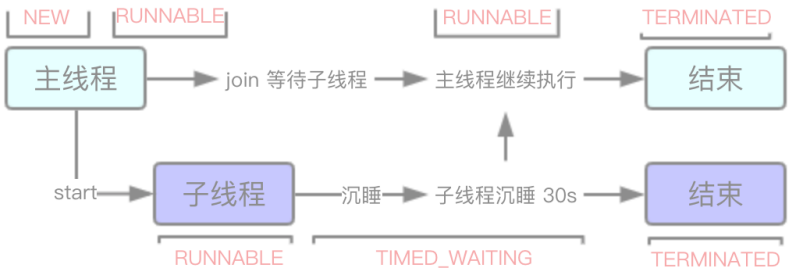
## 5 线程其他操作

### 5.1 join

join 的意思就是当前线程等待另一个线程执行完成之后，才能继续操作，我们写了一个 demo，如下：

```
@Test
public void join() throws Exception {
  Thread main = Thread.currentThread();
  log.info("{} is run。",main.getName());
  Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
      log.info("{} begin run",Thread.currentThread().getName());
      try {
        Thread.sleep(30000L);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      log.info("{} end run",Thread.currentThread().getName());
    }
  });
  // 开一个子线程去执行
  thread.start();
  // 当前主线程等待子线程执行完成之后再执行
  thread.join();
```

执行的结果，就是主线程在执行 thread.join (); 代码后会停住，会等待子线程沉睡 30 秒后再执行，这里的 join 的作用就是让主线程等待子线程执行完成，我们画一个图示意一下：




从图中可以看出，主线程一直等待子线程沉睡 30s 后才继续执行，在等待期间，主线程的状态也是 TIMED_WAITING。

### 5.2 yield

yield 是个 native 方法，底层代码如下：

```
public static native void yield();
```

意思是当前线程做出让步，放弃当前 cpu，让 cpu 重新选择线程，避免线程过度使用 cpu，我们在写 while 死循环的时候，预计短时间内 while 死循环可以结束的话，可以在循环里面使用 yield 方法，防止 cpu 一直被 while 死循环霸占。



有点需要说明的是，让步不是绝不执行，重新竞争时，cpu 也有可能重新选中自己。

### 5.3 sleep

sleep 也是 native 方法，可以接受毫秒的一个入参，也可以接受毫秒和纳秒的两个入参，意思是当前线程会沉睡多久，沉睡时不会释放锁资源，所以沉睡时，其它线程是无法得到锁的。

接受毫秒和纳秒两个入参时，如果给定纳秒大于等于 0.5 毫秒，算一个毫秒，否则不算。

### 5.4 interrupt

interrupt 中文是打断的意思，意思是可以打断中止正在运行的线程，比如：

1. Object#wait ()、Thread#join ()、Thread#sleep (long) 这些方法运行后，线程的状态是 WAITING 或 TIMED_WAITING，这时候打断这些线程，就会抛出 InterruptedException 异常，使线程的状态直接到 TERMINATED；
2. 如果 I/O 操作被阻塞了，我们主动打断当前线程，连接会被关闭，并抛出 ClosedByInterruptException 异常；

我们举一个例子来说明如何打断 WAITING 的线程，代码如下：

```
@Test
public void testInterrupt() throws InterruptedException {
  Thread thread = new Thread(new Runnable() {
    @Override
```

目录

```
            log.info("子线程开始沉睡 30 s");
            Thread.sleep(30000L);
        } catch (InterruptedException e) {
            log.info("子线程被打断");
            e.printStackTrace();
        }
        log.info("{} end run",Thread.currentThread().getName());
    }
});
// 开一个子线程去执行
thread.start();
Thread.sleep(1000L);
log.info("主线程等待 1s 后，发现子线程还没有运行成功，打断子线程");
thread.interrupt();
}
```

例子主要说的是，主线程会等待子线程执行 1s，如果 1s 内子线程还没有执行完，就会打断子线程，子线程被打断后，会抛出 InterruptedException 异常，执行结束，运行的结果如下图：

```
07:05:03.678 [Thread-0] INFO demo.five.ThreadDemo - Thread-0 begin run
07:05:03.684 [Thread-0] INFO demo.five.ThreadDemo - 子线程开始沉睡 30 s
07:05:04.681 [main] INFO demo.five.ThreadDemo - 主线程等待 1s 后，发现子线程还没有运行成功，打断子线程
07:05:04.681 [Thread-0] INFO demo.five.ThreadDemo - 子线程被打断
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at demo.five.ThreadDemo$4.run(ThreadDemo.java:99)
    at java.lang.Thread.run(Thread.java:745)
07:05:04.682 [Thread-0] INFO demo.five.ThreadDemo - Thread-0 end run
```

子线程被打断，抛出异常



## 6 总结

本章主要介绍了线程的基本概念、状态、无返回值线程的初始化方式和线程的常用操作，这些知识也是工作中常用的，也是大家都必须了解的，为后面的学习打下基础。

26 惊叹面试官：由浅入深手写队列

28 Future、ExecutorService 源码解析

**精选留言 0**

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

千学不如一看，千看不如一练