6.5技术分享

亓樱桦 | 今天修改

线程的基本概念

线程（英语：thread）是操作系统能够进行运算调度的最小单位。 它被包含在进程之中，是进程中的实际运作单位。 一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

创建线程

继承Thread

Java中java.lang.Thread这个类表示线程, 一个类可以继承Thread并重写其run方法来实现一个线程

代码块

```
public class HelloThread extends Thread {
@Override
public void run() {
System.out.println("hello");
}
}
```

定义了这个类不代表代码就会开始执行,线程需要被启动,启动需要先创建一个HelloThread对象, 然后调用Thread的start方法

代码块

```
public static void main(String[] args) {
Thread thread = new HelloThread();
thread.start();
}
```

如果不调用start,而直接调用run方法呢?屏幕的输出并不会发生变化,但并不会启动一条单独的执行流, run方法的代码依然是在main线程中执行的, run方法只是main方法调用的一个普通方法。

怎么确认 代码是在哪个线程中执行的呢?

Thread有一个静态方法currentThread,返回当前执行的线程对象:

代码块

```
1   public static native Thread currentThread();
```

修改HelloThead的run方法:

代码块

```
1   @Override
2   public void run() {
3   System.out.println("thread name: "+ Thread.currentThread().getName());
4   System.out.println("hello");
5   }
```

如果在main方法中通过start方法启动线程,程序输出为:

代码块

```
1   thread name: Thread-0
2   hello
```

如果在main方法中直接调用run方法,程序输出为:

代码块

```
1   thread name: main
2   hello
```

实现Runnable

通过继承Thread来实现线程虽然比较简单,但Java中只支持单继承,每个类最多只能有一个父类,如果类已经有父类了,就不能再继承Thread,这时,可以通过实现java.lang.Runnable接口来实现线程。

Runnable接口的定义很简单,只有一个run方法,如下所示:

代码块

```
1   public interface Runnable {
2   public abstract void run();
3   }
```

一个类可以实现该接口,并实现run方法

代码块

```
1   public class HelloRunnable implements Runnable {
2   @Override
```

```
3   public void run() {
4   System.out.println("hello");
5   }
6   }
```

仅仅实现Runnable是不够的,要启动线程,还是要创建一个Thread对象,但传递一个Runnable对象

代码块

```
1   public static void main(String[] args) {
2   Thread helloThread = new Thread(new HelloRunnable());
3   helloThread.start();
4   }
```

线程的基本属性和方法

1. id和name

   每个线程都有一个id和name 。

   id是一个递增的整数,每创建一个线程就加一 。

   name的 默认值是Thread-后跟一个编号, name可以在Thread的构造方法中进行指定,也可以通过setName方法进行 设置,给Thread设置一个友好的名字,可以方便调试。

2. 优先级：1-10 默认为5

   相关方法

   代码块

   ```
   1   public final void setPriority(int newPriority)//设置优先级
   2   public final int getPriority()//获取优先级
   ```

3. 获取状态的方法

   代码块

   ```
   1   public State getState()
   ```

   返回值类型为Thread.State,它是一个枚举类型,有如下值:

   代码块

   ```
   1   public enum State {
   2   NEW,
   3   RUNNABLE,
   ```

```
4   BLOCKED,
5   WAITING,
6   TIMED WAITING,
7   _
8   TERMINATED;
9   }
```

NEW:没有调用start的线程状态为NEW。

TERMINATED:线程运行结束后状态为TERMINATED。

RUNNABLE:调用start后线程在执行run方法且没有阻塞时状态为RUNNABLE,不过,RUNNABLE不代表CPU一定在执行该线程的代码,可能正在执行也可能在等待操作系统分配时间片,只是它没有在等待其他条件。

BLOCKED、WAITING、TIMED_WAITING:都表示线程被阻塞了,在等待一些条件

4. daemon线程

启动线程会启动一条单独的执行流,整个程序只有在所有线程都结束的时候才退 出,但daemon线程是例外,当整个程序中剩下的都是daemon线程的时候,程序就会退出。它一般是其他线程的辅助线程,在它辅助的主线程退出的时候,它就没有存在的意义了。

5. sleep方法

Thread有一个静态的sleep方法,调用该方法会让当前线程睡眠指定的时间,单位是毫秒

代码块

```
1   Thread.sleep(1000);
```

6. yield方法

这也是一个静态方法,调用该方法,是告诉操作系统的调度器:我现在不着急占用CPU,你可以先让其他线程运行。不过,这对调度器也仅仅是建议,调度器如何处理是不一定的,它可能完全忽略该调用。

7. join方法

Thread有一个join方 法,可以让调用join的线程等待该线程结束, join方法的声明为:

代码块

```
1   public final void join() throws InterruptedException
```

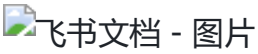
代码块

```
1   public static void main(String[] args) throws InterruptedException {
2   Thread thread = new HelloThread();
```

```
3   thread.start();
4   thread.join();
5   }
```

共享内存

每个线程表示一条单独的执行流,有自己的程序计数器,有自己的栈,但线程之间可以共享内存,它们可以访问和操作相同的对象。



在代码中,定义了一个静态变量shared和静态内部类ChildThread,在main方法中,创建并启动了两个ChildThread对象,传递了相同的list对象, ChildThread的run方法访问了共享的变量shared和list , main方法 最后输出了共享的shared和list的值,大部分情况下,会输出期望的值:

代码块

```
1   2
2   [Thread-0, Thread-1]
```

1)该例中有三条执行流, 一条执行main方法,另外两条执行ChildThread的run方法。

2)不同执行流可以访问和操作相同的变量,如本例中的shared和list变量。

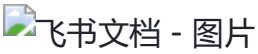
3)不同执行流可以执行相同的程序代码,如本例中incrShared方法, ChildThread的run方法,被两条ChildThread执行流执行, incrShared方法是在外部定义的,但被ChildThread的执行流执行。在分析代码执行过程时,理解代码在被哪个线程执行是很重要的。

4)当多条执行流执行相同的程序代码时,每条执行流都有单独的栈,方法中的参数和局部变量都有自己的一份。

共享内存的常见问题

竞态条件

所谓竞态条件（race condition)是指,当多个线程访问和操作同一个对象时,最终执行结果与执行时序有关,可能正确也可能不正确



这段代码容易理解,有一个共享静态变量counter ,初始值为0,在main方法中创建了1000个线程,每个线程对counter循环加1000次, main线程等待所有线程结束后输出counter的值。

期望的结果是100万,但实际执行,发现每次输出的结果都不一样, 一般都不是100万,经常是99万多。

> 原子操作是指一个或者多个不可再分割的操作。这些操作的执行顺序不能被打乱，这些步骤也不可以被切割而只执行其中的一部分（不可中断性）。

counter++这个操作不是原子操作, 它分为三个步骤:

1)取counter的当前值;

2)在当前值基础上加1；

3)将新值重新赋值给counter。

两个线程可能同时执行第一步,取到了相同的counter值,比如都取到了100,第一个线程执行完后 counter变为101,而第二个线程执行完后还是101,最终的结果就与期望不符。

内存可见性

多个线程可以共享访问和操作相同的变量,但一个线程对一个共享变量的修改,另一个线程不一定马上就能看到,甚至永远也看不到。

在这个程序中,有一个共享的boolean变量shutdown,初始为false , HelloThread在shutdown不为true的情况下一直死循环,当shutdown为true时退出并输出"exit hello" , main线程启动HelloThread后休息了一会,然后设置shutdown为true,最后输出"exit main"

期望的结果是两个线程都退出,但实际执行时,很可能会发现HelloThread永远都不会退出,也就是 说,在HelloThread执行流看来, shutdown永远为false,即使main线程已经更改为了true

这就是内存可见性问题。在计算机系统中,除了内存,数据还会被缓存在CPU的寄存器以及各级缓存中,当访问一个变量时,可能直接从寄存器或CPU缓存中获取,而不一定到内存中去取,当修改一个变量时,也可能是先写到缓存中,稍后才会同步更新到内存中。在单线程的程序中,这一般不是问题,但在多线程的程序中,尤其是在有多CPU的情况下,这就是严重的问题。一个线程对内存的修改,另一个线程看不到, 一是修改没有及时同步到内存,二是另一个线程根本就没从内存读

线程的优点及成本

优点

1)充分利用多CPU的计算能力,单线程只能利用一个CPU,使用多线程可以利用多CPU的计算能力

2)充分利用硬件资源, CPU和硬盘、网络是可以同时工作的, 一个线程在等待网络IO的同时,另一 个线程完全可以利用CPU,对于多个独立的网络请求,完全可以使用多个线程同时请求

3)在用户界面 (GUI)应用程序中,保持程序的响应性,界面和后台任务通常是不同的线程,否则,如果所有事情都是一个线程来执行,当执行一个很慢的任务时,整个界面将停止响应,也无法取消该任务。

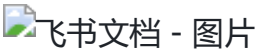
成本

线程调度和切换也是有成本的,当有大量可运行线程的时候,操作系统会忙于调度,为一个线程分配一段时间,执行完后,再让另一个线程执行, 一个线程被切换出去后,操作系统需要保存它的当前上下文状态到内存,上下文状态包括当前CPU寄存器的值、程序计数器的值等,而一个线程被切换回来 后,操作系统需要恢复它原来的上下文状态,整个过程称为上下文切换,这个切换不仅耗时,而且使CPU 中的很多缓存失效。

Synchronized

用法和基本原理

实例方法



与上节类似,我们创建了1000个线程,传递了相同的counter对象,每个线程主要就是调用Counter的 incr方法1000次, main线程等待子线程结束后输出counter的值,这次,不论运行多少次,结果都是正确的 100万。

synchronized实例方法实际保护的是同一个对象的方法调用, 确保同时只能有一个线程执行。 再具体来说, synchronized实例方法保护的是当前实例对象,即this , this对象有一个锁和一个等待队列, 锁只能被一个线程持有,其他试图获得同样锁的线程需要等待。执行synchronized实例方法的过程大致如下:

1)尝试获得锁,如果能够获得锁,继续下一步,否则加入等待队列,阻塞并等待唤醒。

2)执行实例方法体代码。

3)释放锁,如果等待队列上有等待的线程,从中取一个并唤醒,如果有多个等待的线程,唤醒哪一个是不一定的,不保证公平性。

> 当前线程不能获得锁的时候,它会加入等待队列等待,线程的状态会变为BLOCKED

synchronized保护的是对象而非代码,只要访问的是同一个对象的synchronized方法, 即使是不同的代码,也会被同步顺序访问。 比如,对于Counter中的两个实例方法getCount和incr,对同一个Counter对象,一个线程执行getCount,另一个执行incr,它们是不能同时执行的,会被synchronized同步顺序执行。

> 需要说明的是, synchronized方法不能防止非synchronized方法被同时执行。

静态方法

synchronized保护的是对象,对实例方法,保护的是当前实例对象this,对静态方法,保护的是类对象,这里是StaticCounter.class。

实际上,每个对象都有一个锁和一个等待队列,类对象也不例外。

synchronized静态方法和synchronized实例方法保护的是不同的对象,不同的两个线程,可以一个执行synchronized静态方法,另一个执行synchronized实例方法。

代码块

代码块

```
1   public class Counter {
2    private int count;
3    public void incr(){
4    synchronized(this){
5    count ++;
6    }
7    }
8    public int getCount() {
9    synchronized(this){
10   return count;
11   }
12   }
13  }
```

synchronized括号里面的就是保护的对象,对于实例方法,就是this , {}里面是同步执行的代码。

synchronized同步的对象可以是任意对象, 任意对象都有一个锁和等待队列, 或者说,任何对象都可以作为锁对象。

性质

可重入性

synchronized有一个重要的特征,它是可重入的,也就是说,对同一个执行线程,它在获得了锁之后,在调用其他需要同样锁的代码时,可以直接调用。比如,在一个synchronized实例方法内,可以直接调用其他synchronized实例方法。可重入是一个非常自然的属性,应该是很容易理解的,之所以强调,是 因为并不是所有锁都是可重入的。

可重入是通过记录锁的持有线程和持有数量来实现的,当调用被synchronized保护的代码时,检查对象是否已被锁,如果是,再检查是否被当前线程锁定,如果是,增加持有数量,如果不是被当前线程锁定,才加入等待队列,当释放锁时,减少持有数量,当数量变为0时才释放整个锁。

内存可见性

synchronized除了保证原子操作外,它还有一个重要的作用,就是保证内存可见性, 在释放锁时,所有写入都会写回内存,而获得锁后,都会从内存中读最新数据。

如果只是为了保证内存可见性,使用synchronized的成本有点高,有一个更轻量级的方式,那就是给变量加修饰符volatile

代码块

```
1  public class Switcher {
2  private volatile boolean on;
3  public boolean isOn() {
4  return on;
5  }
6  public void setOn(boolean on) {
7  this.on = on;
8  }
9  }
```

死锁

使用synchronized或者其他锁,要注意死锁。所谓死锁就是类似这种现象,比如,有a、b两个线程, a 持有锁A,在等待锁B,而b持有锁B,在等待锁A , a和b陷入了互相等待,最后谁都执行不下去

运行后aThread和bThread陷入了相互等待。怎么解决呢?首先, 应该尽量避免在持有一个锁的同时去申请另一个锁,如果确实需要多个锁,所有代码都应该按照相同的顺序去申请锁。 比如,对于上面的例子,可以约定都先申请lockA,再申请lockB。

线程的基本协作机制

协作场景

1)生产者/消费者协作模式:这是一种常见的协作模式,生产者线程和消费者线程通过共享队列进行 协作,生产者将数据或任务放到队列上,而消费者从队列上取数据或任务,如果队列长度有限,在队列满 的时候,生产者需要等待,而在队列为空的时候,消费者需要等待。

2)同时开始:类似运动员比赛,在听到比赛开始枪响后同时开始,在一些程序,尤其是模拟仿真程序中,要求多个线程能同时开始。

3)等待结束:主从协作模式也是一种常见的协作模式,主线程将任务分解为若干子任务,为每个子任务创建一个线程,主线程在继续执行其他任务之前需要等待每个子任务执行完毕。

4)异步结果:在主从协作模式中,主线程手工创建子线程的写法往往比较麻烦, 一种常见的模式是将子线程的管理封装为异步调用,异步调用马上返回,但返回的不是最终的结果,而是一个一般称为Future的对象,通过它可以在随后获得最终的结果。

5)集合点:类似于学校或公司组团旅游,在旅游过程中有若干集合点,比如出发集合点,每个人从 不同地方来到集合点,所有人到齐后进行下一项活动,在一些程序,比如并行迭代计算中,每个线程负责 一部分计算,然后在集合点等待其他线程完成,所有线程到齐后,交换数据和计算结果,再进行下一次迭代。

wait/notify

我们知道, Java的根父类是Object , Java在Object类而非Thread类中定义了一些线程协作的基本方法,使得每个对象都可以调用这些方法,这些方法有两类, 一类是wait,另一类是notify。

主要有两个wait方法:

代码块

```
1   public final void wait() throws InterruptedException
2   public final native void wait(long timeout) throws InterruptedException;
```

一个带时间参数,单位是毫秒,表示最多等待这么长时间,参数为0表示无限期等待; 一个不带时间 参数,表示无限期等待,实际就是调用wait（0)。在等待期间都可以被中断,如果被中断,会抛出InterruptedException 。

每个对象都有一把锁和等待队列, 一个线程在进入synchronized代码块时,会尝试获取锁,如果获取不到则会把当前线程加入等待队列中,其实, 除了用于锁的等待队列,每个对象还有另一个等待队列,表示条件队列,该队列用于线程间的协作。 调用wait就会把当前线程放到条件队列上并阻塞,表示当前线程执行不下去了,它需要等待一个条件,这个条 件它自己改变不了,需要其他线程改变。当其他线程改变了条件后,应该调用Object的notify方法:

代码块

```
1   public final native void notify();
2   public final native void notifyAll();
```

notify做的事情就是从条件队列中选一个线程,将其从队列中移除并唤醒, notifyAll和notify的区别是,它会移除条件队列中所有的线程并全部唤醒。

示例代码中有两个线程, 一个是主线程, 一个是WaitThread,协作的条件变量是fire , WaitThread等待该变量变为true ,在不为true的时候调用wait,主线程设置该变量并调用notify。

两个线程都要访问协作的变量fire,容易出现竞态条件,所以相关代码都需要被synchronized保护。实际上, wait/notify方法只能在synchronized代码块内被调用, 如果调用wait/notify方法时,当前线程没有持有对象锁,会抛出异常java.lang.IllegalMonitor-StateException。

wait必须被synchronized保护,那一个线程在wait时,另一个线程怎么可能调用同样被synchronized保护的notify方法呢?它不需要等待锁吗?我们需要进一步理解wait的内部过程, 虽然是在synchronized方法内,但调用wait时,线程会释放对象锁。

wait的具体过程是:

1)把当前线程放入条件等待队列,释放对象锁,阻塞等待,线程状态变为WAITING或 TIMED_WAITING。

2)等待时间到或被其他线程调用notify/notifyAll从条件队列中移除,这时,要重新竞争对象锁: ·如果能够获得锁,线程状态变为RUNNABLE,并从wait调用中返回。

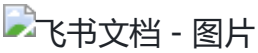
·否则,该线程加入对象锁等待队列,线程状态变为BLOCKED,只有在获得锁后才会从wait调用中返回。

调用notify会把在条件队列中等待的线程唤醒并从队列中移除,但它不会释放对象锁,也就是说,只有在包含notify的synchronized代码块执行完后,等待的线程才会从wait调用中返回。

wait/notify方法被不同的线程调用,但共享相同的锁和条件等待队列(相同对象的synchronized代码块内),它们围绕一个共享的条件变量进行协作, 这个条件变量是程序自己维护的,当条件不成立时,线程调用wait进入条件等待队列,另一个线程修改了条件变量后调用notify,调用wait的线程唤醒后需要重新检查条件变量。从多线程的角度看,它们围绕共享变量进行协作,从调用wait的线程角度看,它阻塞等待一个条件的成立。我们在设计多线程协作时,需要想清楚协作的共享变量和条件是什么,这是协作的核心。

生产者/消费者模式

在生产者/消费者模式中,协作的共享变量是队列,生产者往队列上放数据,如果满了就wait,而消费者从队列上取数据,如果队列为空也wait。



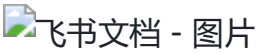
同时开始

同时开始,类似于运动员比赛,在听到比赛开始枪响后同时开始,下面,我们模拟这个过程。这

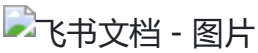
里,有一个主线程和N个子线程,每个子线程模拟一个运动员,主线程模拟裁判,它们协作的共享变量是一个开始信号。

协作对象FireFlag

表示比赛运动员的类



主程序代码



等待结束

我们使用join方法让主线程等待子线程结束, join实际上就是调用了wait

代码块

```
1   while (isAlive()) {
2   wait(0);
3   }
```

只要线程是活着的, isAlive ()返回true , join就一直等待。当线程运行结束的时候, Java系统调用notifyAll来通知。

异步结果

在主从模式中,手工创建线程往往比较麻烦, 一种常见的模式是异步调用,异步调用返回一个一般称为Future的对象,通过它可以获得最终的结果。异步结果是指在执行某个操作后，不会立即返回结果，而是通过某种机制（如回调函数、事件监听器或 Future 对象）在操作完成后再通知调用者结果的一种编程模式。这种模式在处理耗时操作（如网络请求、文件 I/O、计算密集型任务等）时非常有用，因为它允许程序在等待操作完成的同时继续执行其他任务，从而提高程序的效率和响应速度。

集合点

各个线程先是分头行动,各自到达一个集合点,在集合点需要集齐所有线程,交换数据,然后再进行下一步动作。怎么表示这种协作呢?协作的共享变量依然是一个数,这个数表示未到集合点的线程个数,初始值为子线程个数,每个线程到达集合点后将该值减一,如果不为0,表示还有别的线程未到,进行等待,如果变为0,表示自己是最后一个到的,调用notifyAll唤醒所有线程。

线程的中断

取消/关闭的机制

停止一个线程的主要机制是中断,中断并不是强迫终止一个线程,它是一种协作机制,是给线程传递一个取消信号,但是由线程来决定如何以及何时退出。

关于中断的方法

代码块

```
1   public boolean isInterrupted()
2   public void interrupt()//中断线程
3   public static boolean interrupted()
```

这三个方法名字类似,比较容易混淆,我们解释一下。isInterrupted ()和interrupt ()是实例方法, 调用它们需要通过线程对象; interrupted ()是静态方法,实际会调用Thread.currentThread ()操作当前线程。

每个线程都有一个标志位,表示该线程是否被中断了。

1. isInterrupted:返回对应线程的中断标志位是否为true。
2. interrupted:返回当前线程的中断标志位是否为true ,但它还有一个重要的副作用,就是清空中断标志位, 也就是说,连续两次调用interrupted (),第一次返回的结果为true ,第二次一般就是false (除非同时又发生了一次中断)。
3. interrupt:表示中断对应的线程。

线程对中断的反应

·RUNNABLE :线程在运行或具备运行条件只是在等待操作系统调度。

·WAITING/TIMED_WAITING :线程在等待某个条件或超时。条件队列

·BLOCKED :线程在等待锁,试图进入同步块。 锁队列

·NEW/TERMINATED :线程还未启动或已结束。

RUNNABLE

如果线程在运行中,且没有执行操作, interrupt ()只是会设置线程的中断标志位,没有任何其他作用。线程应该在运行过程中合适的位置检查中断标志位,比如,如果主体代码是一个循环,可以在循环开始处进行检查,如下所示

代码块

```
1   public class InterruptRunnableDemo extends Thread {
2   @Override
3   public void run() {
4   while(!Thread.currentThread().isInterrupted()) {
5   //…单次循环代码 }
6   System.out.println("done "); }
7   //其他代码 }
```

WAITING/TIMED_WAITING

线程调用join/wait/sleep方法会进入WAITING或TIMED_WAITING状态,在这些状态时,对线程对象调用interrupt（)会使得该线程抛出InterruptedException。需要注意的是，抛出异常后,中断标志位会被清空,而不是被设置。

1. 线程的等待状态

当线程调用以下方法时，会进入特定的等待状态：

- `join()`：等待另一个线程终止。
- `wait()`：在对象的监视器上等待。
- `sleep()`：使线程暂停执行指定的时间。

这些方法会使线程进入以下两种状态之一：

- **WAITING**：无时间限制的等待状态（如 `wait()`、`join()`）。
- **TIMED_WAITING**：有时间限制的等待状态（如 `sleep()`、带超时时间的 `wait()`）。

2. 中断等待中的线程

如果在线程处于上述等待状态时调用该线程的 `interrupt()` 方法，线程会抛出 `InterruptedException`。这个异常表示线程的等待被中断。

3. 中断标志位的清空

当线程抛出 `InterruptedException` 时，它的中断标志位会被自动清空（即设置为 `false`）。这意味着：

- 如果你在捕获 `InterruptedException` 后再次检查中断状态，`isInterrupted()` 方法会返回 `false`。
- 这是 `InterruptedException` 的一个关键特性：它不仅通知线程被中断，还会清除中断标志位。

代码块

```
1  public class TestInterrupt {
2      public static void main(String[] args) {
3          Thread thread = new Thread(() -> {
4              try {
```

```
                Thread.sleep(1000); // 进入 TIMED_WAITING 状态
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught.");
                // 中断标志位已被清空
                System.out.println("isInterrupted: " + Thread.currentThread().
            }
        });

        thread.start();
        thread.interrupt(); // 中断线程
    }
}
```

在这个例子中：

1. 线程调用 `Thread.sleep(1000)` 进入 `TIMED_WAITING` 状态。
2. 主线程调用 `thread.interrupt()` 中断该线程。
3. 睡眠中的线程会抛出 `InterruptedException`。
4. 在捕获异常后，打印中断状态，会发现 `isInterrupted()` 返回 `false`，因为抛出异常时中断标志位已被清空。

处理方式

捕获到InterruptedException,通常表示希望结束该线程,线程大致有两种处理方式:

1)向上传递该异常,这使得该方法也变成了一个可中断的方法,需要调用者进行处理;

2)有些情况,不能向上传递异常,比如Thread的run方法,它的声明是固定的,不能抛出任何受检异常,这时,应该捕获异常,进行合适的清理操作,清理后, 一般应该调用Thread的interrupt方法设置中断标志位,使得其他代码有办法知道它发生了中断。

BLOCKED

如果线程在等待锁,对线程对象调用interrupt ()只是会设置线程的中断标志位,线程依然会处于BLOCKED状态,也就是说, interrupt ()并不能使一个在等待锁的线程真正“ 中断” 。

NEW/TERMINATE

如果线程尚未启动 (NEW),或者已经结束 (TERMINATED),则调用interrupt ()对它没有任何效果,中断标志位也不会被设置。

如何正确地取消/关闭线程

对于以线程提供服务的程序模块而言,它应该封装取消/关闭操作,提供单独的取消/关闭方法给调用者,外部调用者应该调用这些方法而不是直接调用interrupt 。

代码块

```
boolean cancel(boolean mayInterruptIfRunning);
void shutdown();
List<Runnable> shutdownNow();
```

https://www.cnblogs.com/dolphin0520/p/3933404.html 同步容器

https://github.com/Werun-backend/resource/blob/main/%E6%8A%80%E6%9C%AF%E5%88%86%E4%BA%AB/2024%E5%B9%B4%E6%98%A5%E5%AD%A3/%E9%99%88%E5%AD%90%E6%B6%B5/%E5%A4%9A%E7%BA%BF%E7%A8%8B.pdf

https://github.com/Werun-backend/resource/blob/main/%E6%8A%80%E6%9C%AF%E5%88%86%E4%BA%AB/2024%E5%B9%B4%E7%A7%8B%E5%AD%A3/%E7%89%9F%E5%BF%97%E9%B9%8F/Java%E5%B9%B6%E5%8F%91%E7%BC%96%E7%A8%8B%E5%85%A5%E9%97%A8.md