

13 多线程

多线程是Java最基本的一种并发模型，本章我们将详细介绍Java多线程编程。



多线程基础

现代操作系统（Windows, macOS, Linux）都可以执行多任务。多任务就是同时运行多个任务，例如：

CPU执行代码都是一条一条顺序执行的，但是，即使是单核cpu，也可以同时运行多个任务。因为操作系统执行多任务实际上就是让CPU对多个任务轮流交替执行。

例如，假设我们有语文、数学、英语3门作业要做，每个作业需要30分钟。我们把这3门作业看成是3个任务，可以做1分钟语文作业，再做1分钟数学作业，再做1分钟英语作业：



这样轮流做下去，在有些人眼里看来，做作业的速度就非常快，看上去就像同时在做3门作业一样



类似的，操作系统轮流让多个任务交替执行，例如，让浏览器执行0.001秒，让QQ执行0.001秒，再让音乐播放器执行0.001秒，在人看来，CPU就是在同时执行多个任务。

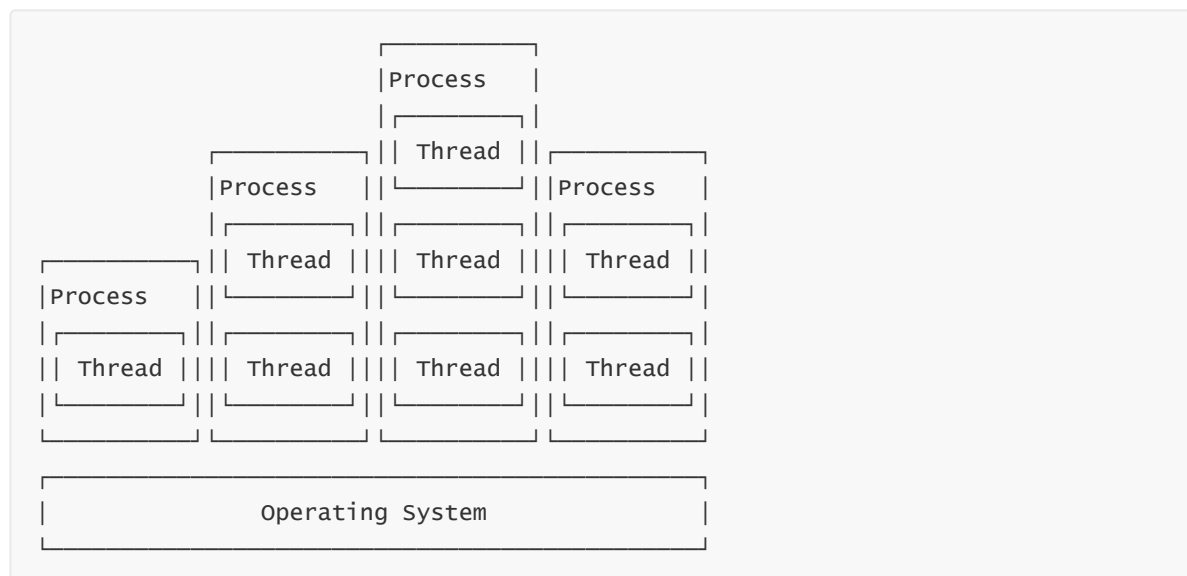
即使是多核CPU，因为通常任务的数量远远多于CPU的核数，所以任务也是交替执行的。

进程

在计算机中，我们把一个任务称为一个进程，浏览器就是一个进程，视频播放器是另一个进程，类似的，音乐播放器和Word都是进程。

某些进程内部还需要同时执行多个子任务。例如，我们在使用Word时，Word可以让我们一边打字，一边进行拼写检查，同时还可以在后台进行打印，我们把子任务称为线程。

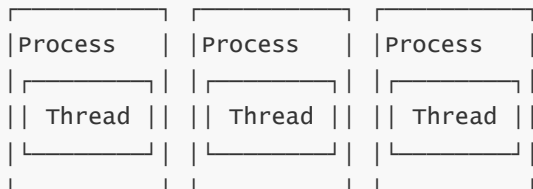
进程和线程的关系就是：一个进程可以包含一个或多个线程，但至少会有一个线程。



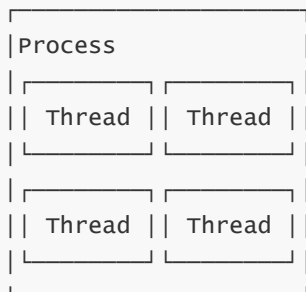
操作系统调度的最小任务单位其实不是进程，而是线程。常用的Windows、Linux等操作系统都采用抢占式多任务，如何调度线程完全由操作系统决定，程序自己不能决定什么时候执行，以及执行多长时间。

因为同一个应用程序，既可以有多个进程，也可以有多个线程，因此，实现多任务的方法，有以下几种：

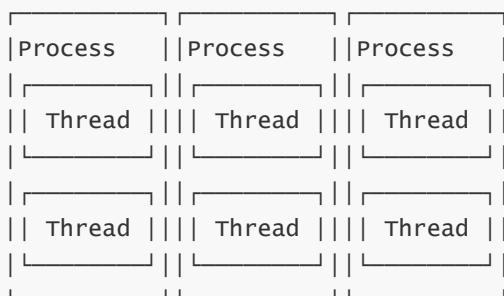
多进程模式（每个进程只有一个线程）：



多线程模式（一个进程有多个线程）：



多进程 + 多线程模式（复杂度最高）：



进程 vs 线程

进程和线程是包含关系，但是多任务既可以由多进程实现，也可以由单进程内的多线程实现，还可以混合多进程 + 多线程。

具体采用哪种方式，要考虑到进程和线程的特点。

和多线程相比，多进程的缺点在于：

- 创建进程比创建线程开销大，尤其是在Windows系统上；
- 进程间通信比线程间通信要慢，因为线程间通信就是读写同一个变量，速度很快。

而多进程的优点在于：

多进程稳定性比多线程高，因为在多进程的情况下，一个进程崩溃不会影响其他进程，而在多线程的情况下，任何一个线程崩溃会直接导致整个进程崩溃。

多线程

Java语言内置了多线程支持：一个Java程序实际上是一个JVM进程，JVM进程用一个主线程来执行 `main()` 方法，在 `main()` 方法内部，我们又可以启动多个线程。此外，JVM还有负责垃圾回收的其他工作线程等。

因此，对于大多数Java程序来说，我们说多任务，实际上是说如何使用多线程实现多任务。

和单线程相比，多线程编程的特点在于：多线程经常需要读写共享数据，并且需要同步。例如，播放电影时，就必须由一个线程播放视频，另一个线程播放音频，两个线程需要协调运行，否则画面和声音就不同步。因此，多线程编程的复杂度高，调试更困难。

Java多线程编程的特点又在于：

- 多线程模型是Java程序最基本的并发模型；
- 后续读写网络、数据库、Web开发等都依赖Java多线程模型。

因此，必须掌握Java多线程编程才能继续深入学习其他内容。

创建新线程

Java语言内置了多线程支持。当Java程序启动的时候，实际上是启动了一个JVM进程，然后，JVM启动主线程来执行 `main()` 方法。在 `main()` 方法中，我们又可以启动其他线程。

要创建一个新线程非常容易，我们需要实例化一个 `Thread` 实例，然后调用它的 `start()` 方法：

```
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread();
        t.start(); // 启动新线程
    }
}
```

但是这个线程启动后实际上什么也不做就立刻结束了。我们希望新线程能执行指定的代码，有以下几种方法：

方法一：从 `Thread` 派生一个自定义类，然后覆写 `run()` 方法：

```
public class Main {
    public static void main(String[] args) {
        Thread t = new MyThread();
        t.start(); // 启动新线程
    }
}

class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("start new thread!");
    }
}
```

执行上述代码，注意到 `start()` 方法会在内部自动调用实例的 `run()` 方法。

方法二：创建 `Thread` 实例时，传入一个 `Runnable` 实例：

```

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start(); // 启动新线程
    }
}

class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("start new thread!");
    }
}

```

或者用java8引入的lambda语法进一步简写为:

```

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("start new thread!");
        });
        t.start(); // 启动新线程
    }
}

```

有童鞋会问，使用线程执行的打印语句，和直接在 `main()` 方法执行有区别吗？

区别大了去了。我们看以下代码：

```

public class Main {
    public static void main(String[] args) {
        System.out.println("main start...");
        Thread t = new Thread() {
            public void run() {
                System.out.println("thread run...");
                System.out.println("thread end.");
            }
        };
        t.start();
        System.out.println("main end...");
    }
}

```

我们用蓝色表示主线程，也就是 `main` 线程，`main` 线程执行的代码有4行，首先打印 `main start`，然后创建 `Thread` 对象，紧接着调用 `start()` 启动新线程。当 `start()` 方法被调用时，JVM就创建了一个新线程，我们通过实例变量 `t` 来表示这个新线程对象，并开始执行。

接着，`main` 线程继续执行打印 `main end` 语句，而 `t` 线程在 `main` 线程执行的同时会并发执行，打印 `thread run` 和 `thread end` 语句。

当 `run()` 方法结束时，新线程就结束了。而 `main()` 方法结束时，主线程也结束了。

我们再来看线程的执行顺序：

1. `main` 线程肯定是先打印 `main start`，再打印 `main end`；
2. `t` 线程肯定是先打印 `thread run`，再打印 `thread end`。

但是，除了可以肯定，`main start` 会先打印外，`main end` 打印在 `thread run` 之前、`thread end` 之后或者之间，都无法确定。因为从 `t` 线程开始运行以后，两个线程就开始同时运行了，并且由操作系统调度，程序本身无法确定线程的调度顺序。

要模拟并发执行的效果，我们可以在线程中调用 `Thread.sleep()`，强迫当前线程暂停一段时间：

```
public class Main {
    public static void main(String[] args) {
        System.out.println("main start...");
        Thread t = new Thread() {
            public void run() {
                System.out.println("thread run...");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {}
                System.out.println("thread end.");
            }
        };
        t.start();
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {}
        System.out.println("main end...");
    }
}
```

`sleep()` 传入的参数是毫秒。调整暂停时间的大小，我们可以看到 `main` 线程和 `t` 线程执行的先后顺序。

要特别注意：直接调用 `Thread` 实例的 `run()` 方法是无效的：

```
public class Main {
    public static void main(String[] args) {
        Thread t = new MyThread();
        t.run();
    }
}

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

直接调用 `run()` 方法，相当于调用了一个普通的Java方法，当前线程并没有任何改变，也不会启动新线程。上述代码实际上是在 `main()` 方法内部又调用了 `run()` 方法，打印 `hello` 语句是在 `main` 线程中执行的，没有任何新线程被创建。

必须调用 `Thread` 实例的 `start()` 方法才能启动新线程，如果我们查看 `Thread` 类的源代码，会看到 `start()` 方法内部调用了一个 `private native void start0()` 方法，`native` 修饰符表示这个方法是由JVM虚拟机内部的C代码实现的，不是由Java代码实现的。

线程的优先级

可以对线程设定优先级，设定优先级的方法是：

```
Thread.setPriority(int n) // 1~10，默认值5
```

优先级高的线程被操作系统调度的优先级较高，操作系统对高优先级线程可能调度更频繁，但我们决不能通过设置优先级来确保高优先级的线程一定会先执行。

练习

下载练习：[创建新线程](#)（推荐使用[IDE练习插件](#)快速下载）

小结

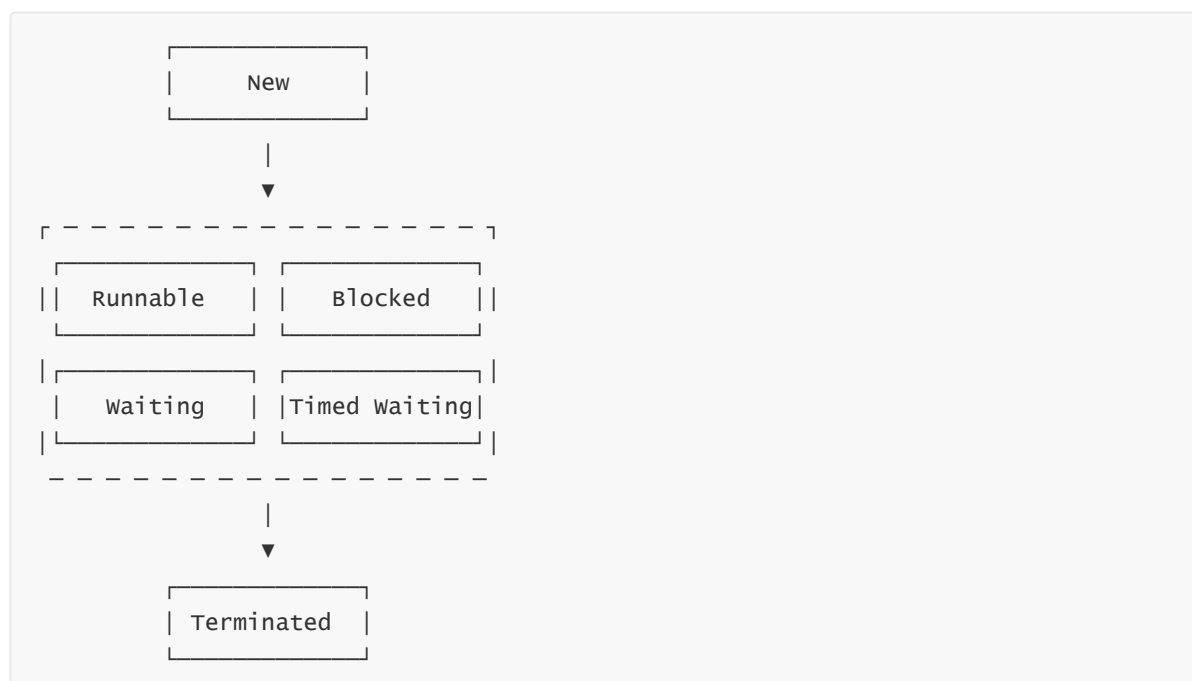
- Java用 `Thread` 对象表示一个线程，通过调用 `start()` 启动一个新线程；
- 一个线程对象只能调用一次 `start()` 方法；
- 线程的执行代码写在 `run()` 方法中；
- 线程调度由操作系统决定，程序本身无法决定调度顺序；
- `Thread.sleep()` 可以把当前线程暂停一段时间。

线程的状态

在Java程序中，一个线程对象只能调用一次 `start()` 方法启动新线程，并在新线程中执行 `run()` 方法。一旦 `run()` 方法执行完毕，线程就结束了。因此，Java线程的状态有以下几种：

- New：新创建的线程，尚未执行；
- Runnable：运行中的线程，正在执行 `run()` 方法的Java代码；
- Blocked：运行中的线程，因为某些操作被阻塞而挂起；
- Waiting：运行中的线程，因为某些操作在等待中；
- Timed Waiting：运行中的线程，因为执行 `sleep()` 方法正在计时等待；
- Terminated：线程已终止，因为 `run()` 方法执行完毕。

用一个状态转移图表示如下：



当线程启动后，它可以在 `Runnable`、`Blocked`、`Waiting` 和 `Timed waiting` 这几个状态之间切换，直到最后变成 `Terminated` 状态，线程终止。

线程终止的原因有：

- 线程正常终止：`run()` 方法执行到 `return` 语句返回；
- 线程意外终止：`run()` 方法因为未捕获的异常导致线程终止；
- 对某个线程的 `Thread` 实例调用 `stop()` 方法强制终止（强烈不推荐使用）。

一个线程还可以等待另一个线程直到其运行结束。例如，`main` 线程在启动 `t` 线程后，可以通过 `t.join()` 等待 `t` 线程结束后再继续运行：

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread() -> {
            System.out.println("hello");
        };
        System.out.println("start");
        t.start();
        t.join();
        System.out.println("end");
    }
}
```

当 `main` 线程对线程对象 `t` 调用 `join()` 方法时，主线程将等待变量 `t` 表示的线程运行结束，即 `join` 就是指等待该线程结束，然后才继续往下执行自身线程。所以，上述代码打印顺序可以肯定是 `main` 线程先打印 `start`，`t` 线程再打印 `hello`，`main` 线程最后再打印 `end`。

如果 `t` 线程已经结束，对实例 `t` 调用 `join()` 会立刻返回。此外，`join(long)` 的重载方法也可以指定一个等待时间，超过等待时间后就不再继续等待。

小结

- Java线程对象 `Thread` 的状态包括：`New`、`Runnable`、`Blocked`、`Waiting`、`Timed waiting` 和 `Terminated`；
- 通过对另一个线程对象调用 `join()` 方法可以等待其执行结束；
- 可以指定等待时间，超过等待时间线程仍然没有结束就不再等待；
- 对已经运行结束的线程调用 `join()` 方法会立刻返回。

中断线程

如果线程需要执行一个长时间任务，就可能需要能中断线程。中断线程就是其他线程给该线程发一个信号，该线程收到信号后结束执行 `run()` 方法，使得自身线程能立刻结束运行。

我们举个栗子：假设从网络下载一个100M的文件，如果网速很慢，用户等得不耐烦，就可能在下载过程中点“取消”，这时，程序就需要中断下载线程的执行。

中断一个线程非常简单，只需要在其他线程中对目标线程调用 `interrupt()` 方法，目标线程需要反复检测自身状态是否是 `interrupted` 状态，如果是，就立刻结束运行。

我们还是看示例代码：

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new MyThread();
        t.start();
    }
}
```



```

        Thread.sleep(1); // 暂停1毫秒
        t.interrupt(); // 中断t线程
        t.join(); // 等待t线程结束
        System.out.println("end");
    }
}

class MyThread extends Thread {
    public void run() {
        int n = 0;
        while (!isInterrupted()) {
            n++;
            System.out.println(n + " hello!");
        }
    }
}

```

仔细看上述代码，main 线程通过调用 `t.interrupt()` 方法中断 t 线程，但是要注意，`interrupt()` 方法仅仅向 t 线程发出了“中断请求”，至于 t 线程是否能立刻响应，要看具体代码。而 t 线程的 `while` 循环会检测 `isInterrupted()`，所以上述代码能正确响应 `interrupt()` 请求，使得自身立刻结束运行 `run()` 方法。

如果线程处于等待状态，例如，`t.join()` 会让 main 线程进入等待状态，此时，如果对 main 线程调用 `interrupt()`，`join()` 方法会立刻抛出 `InterruptedException`，因此，目标线程只要捕获到 `join()` 方法抛出的 `InterruptedException`，就说明有其他线程对其调用了 `interrupt()` 方法，通常情况下该线程应该立刻结束运行。

我们来看下面的示例代码：

```

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new MyThread();
        t.start();
        Thread.sleep(1000);
        t.interrupt(); // 中断t线程
        t.join(); // 等待t线程结束
        System.out.println("end");
    }
}

class MyThread extends Thread {
    public void run() {
        Thread hello = new HelloThread();
        hello.start(); // 启动hello线程
        try {
            hello.join(); // 等待hello线程结束
        } catch (InterruptedException e) {
            System.out.println("interrupted!");
        }
        hello.interrupt();
    }
}

class HelloThread extends Thread {
    public void run() {
        int n = 0;
        while (!isInterrupted()) {

```

```

        n++;
        System.out.println(n + " hello!");
    } try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        break;
    }
}
}
}

```

main 线程通过调用 `t.interrupt()` 从而通知 `t` 线程中断，而此时 `t` 线程正位于 `hello.join()` 的等待中，此方法会立刻结束等待并抛出 `InterruptedException`。由于我们在 `t` 线程中捕获了 `InterruptedException`，因此，就可以准备结束该线程。在 `t` 线程结束前，对 `hello` 线程也进行了 `interrupt()` 调用通知其中断。如果去掉这一行代码，可以发现 `hello` 线程仍然会继续运行，且JVM不会退出。

另一个常用的中断线程的方法是设置标志位。我们通常会用一个 `running` 标志位来标识线程是否应该继续运行，在外部线程中，通过把 `HelloThread.running` 置为 `false`，就可以让线程结束：

```

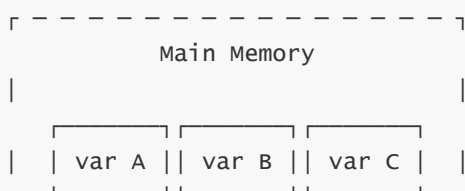
public class Main {
    public static void main(String[] args) throws InterruptedException {
        HelloThread t = new HelloThread();
        t.start();
        Thread.sleep(1);
        t.running = false; // 标志位置为false
    }
}

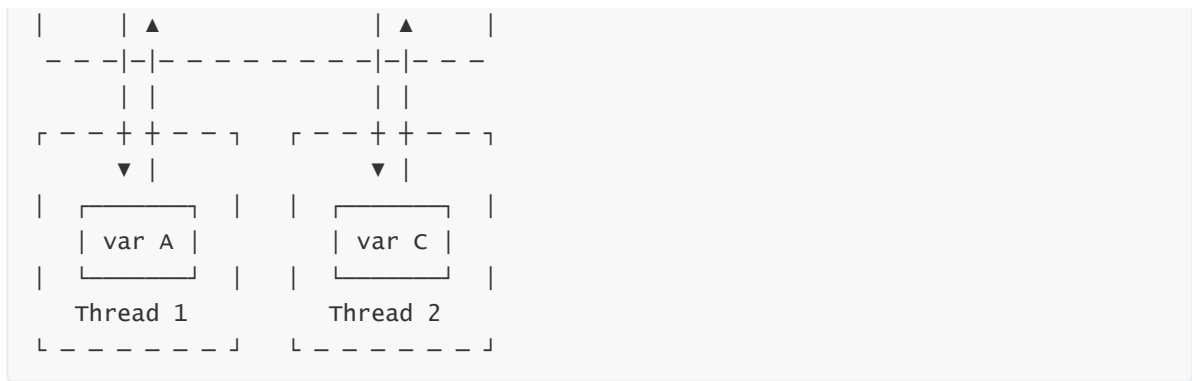
class HelloThread extends Thread {
    public volatile boolean running = true;
    public void run() {
        int n = 0;
        while (running) {
            n ++;
            System.out.println(n + " hello!");
        }
        System.out.println("end!");
    }
}

```

注意到 `HelloThread` 的标志位 `boolean running` 是一个线程间共享的变量。线程间共享变量需要使用 `volatile` 关键字标记，确保每个线程都能读取到更新后的变量值。

为什么要对线程间共享的变量用关键字 `volatile` 声明？这涉及到Java的内存模型。在Java虚拟机中，变量的值保存在主内存中，但是，当线程访问变量时，它会先获取一个副本，并保存在自己的工作内存中。如果线程修改了变量的值，虚拟机会在某个时刻把修改后的值回写到主内存，但是，这个时间是不确定的！





这会导致如果一个线程更新了某个变量，另一个线程读取的值可能还是更新前的。例如，主内存的变量 `a = true`，线程1执行 `a = false` 时，它在此刻仅仅是把变量 `a` 的副本变成了 `false`，主内存的变量 `a` 还是 `true`，在JVM把修改后的 `a` 回写到主内存之前，其他线程读取到的 `a` 的值仍然是 `true`，这就造成了多线程之间共享的变量不一致。

因此，`volatile` 关键字的目的是告诉虚拟机：

- 每次访问变量时，总是获取主内存的最新值；
- 每次修改变量后，立刻回写到主内存。

`volatile` 关键字解决的是可见性问题：当一个线程修改了某个共享变量的值，其他线程能够立刻看到修改后的值。

如果我们去掉 `volatile` 关键字，运行上述程序，发现效果和带 `volatile` 差不多，这是因为在x86的架构下，JVM回写主内存的速度非常快，但是，换成ARM的架构，就会有显著的延迟。

小结

- 对目标线程调用 `interrupt()` 方法可以请求中断一个线程，目标线程通过检测 `isInterrupted()` 标志获取自身是否已中断。如果目标线程处于等待状态，该线程会捕获到 `InterruptedException`；
- 目标线程检测到 `isInterrupted()` 为 `true` 或者捕获了 `InterruptedException` 都应该立刻结束自身线程；
- 通过标志位判断需要正确使用 `volatile` 关键字；-`volatile` 关键字解决了共享变量在线程间的可见性问题。

守护线程

Java程序入口就是由JVM启动 `main` 线程，`main` 线程又可以启动其他线程。当所有线程都运行结束时，JVM退出，进程结束。

如果有一个线程没有退出，JVM进程就不会退出。所以，必须保证所有线程都能及时结束。

但是有一种线程的目的就是无限循环，例如，一个定时触发任务的线程：

```
class TimerThread extends Thread {
    @Override
    public void run() {
        while (true) {
            System.out.println(LocalTime.now());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                break;
            }
        }
    }
}
```

如果这个线程不结束，JVM进程就无法结束。问题是，由谁负责结束这个线程？

然而这类线程经常没有负责人来负责结束它们。但是，当其他线程结束时，JVM进程又必须要结束，怎么办？

答案是使用守护线程（Daemon Thread）。

守护线程是指为其他线程服务的线程。在JVM中，所有非守护线程都执行完毕后，无论有没有守护线程，虚拟机都会自动退出。

因此，JVM退出时，不必关心守护线程是否已结束。

如何创建守护线程呢？方法和普通线程一样，只是在调用 `start()` 方法前，调用 `setDaemon(true)` 把该线程标记为守护线程：

```
Thread t = new MyThread();
t.setDaemon(true);
t.start();
```

在守护线程中，编写代码要注意：守护线程不能持有任何需要关闭的资源，例如打开文件等，因为虚拟机退出时，守护线程没有任何机会来关闭文件，这会导致数据丢失。

练习

下载练习：[使用守护线程](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- 守护线程是为其他线程服务的线程；
- 所有非守护线程都执行完毕后，虚拟机退出；
- 守护线程不能持有需要关闭的资源（如打开文件等）。

线程同步

当多个线程同时运行时，线程的调度由操作系统决定，程序本身无法决定。因此，任何一个线程都有可能任何指令处被操作系统暂停，然后在某个时间段后继续执行。

这个时候，有个单线程模型下不存在的问题就来了：如果多个线程同时读写共享变量，会出现数据不一致的问题。

我们来看一个例子：

```
public class Main {
```

```

        public static void main(String[] args) throws Exception {
            var add = new AddThread();
            var dec = new DecThread();
            add.start();
            dec.start();
            add.join();
            dec.join();
            System.out.println(Counter.count);
        }
    }

    class Counter {
        public static int count = 0;
    }

    class AddThread extends Thread {
        public void run() {
            for (int i=0; i<10000; i++) { Counter.count += 1; }
        }
    }

    class DecThread extends Thread {
        public void run() {
            for (int i=0; i<10000; i++) { Counter.count -= 1; }
        }
    }
}

```

上面的代码很简单，两个线程同时对一个 `int` 变量进行操作，一个加10000次，一个减10000次，最后结果应该是0，但是，每次运行，结果实际上都是不一样的。

这是因为对变量进行读取和写入时，结果要正确，必须保证是原子操作。原子操作是指不能被中断的一个或一系列操作。

例如，对于语句：

```
n = n + 1;
```

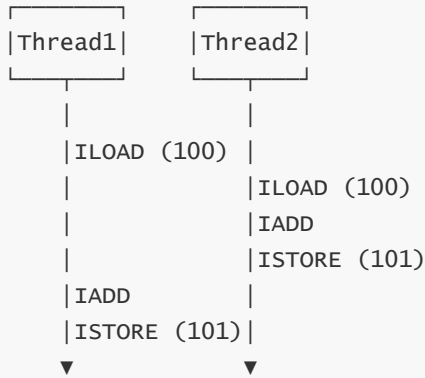
看上去是一行语句，实际上对应了3条指令：

```

ILOAD
IADD
ISTORE

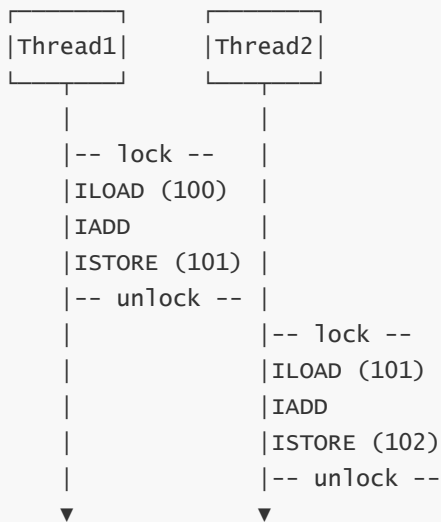
```

我们假设 `n` 的值是 100，如果两个线程同时执行 `n = n + 1`，得到的结果很可能不是 102，而是 101，原因在于：



如果线程1在执行 `ILOAD` 后被操作系统中断，此刻如果线程2被调度执行，它执行 `ILOAD` 后获取的值仍然是 `100`，最终结果被两个线程的 `ISTORE` 写入后变成了 `101`，而不是期待的 `102`。

这说明多线程模型下，要保证逻辑正确，对共享变量进行读写时，必须保证一组指令以原子方式执行：即某一个线程执行时，其他线程必须等待：



通过加锁和解锁的操作，就能保证3条指令总是在一个线程执行期间，不会有其他线程会进入此指令区间。即使在执行期线程被操作系统中断执行，其他线程也会因为无法获得锁导致无法进入此指令区间。只有执行线程将锁释放后，其他线程才有机会获得锁并执行。这种加锁和解锁之间的代码块我们称之为临界区（Critical Section），任何时候临界区最多只有一个线程能执行。

可见，保证一段代码的原子性就是通过加锁和解锁实现的。Java程序使用 `synchronized` 关键字对一个对象进行加锁：

```

synchronized(lock) {
    n = n + 1;
}

```

`synchronized` 保证了代码块在任意时刻最多只有一个线程能执行。我们把上面的代码用 `synchronized` 改写如下：

```

public class Main {
    public static void main(String[] args) throws Exception {
        var add = new AddThread();
        var dec = new DecThread();
        add.start();
        dec.start();
    }
}

```

```

        add.join();
        dec.join();
        System.out.println(Counter.count);
    }
}

class Counter {
    public static final Object lock = new Object();
    public static int count = 0;
}

class AddThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.count += 1;
            }
        }
    }
}

class DecThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.count -= 1;
            }
        }
    }
}

```

注意到代码：

```

synchronized(Counter.lock) { // 获取锁
    // ...
} // 释放锁

```

它表示用 `Counter.lock` 实例作为锁，两个线程在执行各自的 `synchronized(Counter.lock) { ... }` 代码块时，必须先获得锁，才能进入代码块进行。执行结束后，在 `synchronized` 语句块结束会自动释放锁。这样一来，对 `Counter.count` 变量进行读写就不可能同时进行。上述代码无论运行多少次，最终结果都是0。

使用 `synchronized` 解决了多线程同步访问共享变量的正确性问题。但是，它的缺点是带来了性能下降。因为 `synchronized` 代码块无法并发执行。此外，加锁和解锁需要消耗一定的时间，所以，`synchronized` 会降低程序的执行效率。

我们来概括一下如何使用 `synchronized`：

1. 找出修改共享变量的线程代码块；
2. 选择一个共享实例作为锁；
3. 使用 `synchronized(lockObject) { ... }`。

在使用 `synchronized` 的时候，不必担心抛出异常。因为无论是否有异常，都会在 `synchronized` 结束处正确释放锁：

```

public void add(int m) {
    synchronized (obj) {
        if (m < 0) {
            throw new RuntimeException();
        }
        this.value += m;
    } // 无论有无异常，都会在此释放锁
}

```

我们再来看一个错误使用 `synchronized` 的例子：

```

public class Main {
    public static void main(String[] args) throws Exception {
        var add = new AddThread();
        var dec = new DecThread();
        add.start();
        dec.start();
        add.join();
        dec.join();
        System.out.println(Counter.count);
    }
}

class Counter {
    public static final Object lock1 = new Object();
    public static final Object lock2 = new Object();
    public static int count = 0;
}

class AddThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock1) {
                Counter.count += 1;
            }
        }
    }
}

class DecThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock2) {
                Counter.count -= 1;
            }
        }
    }
}

```

结果并不是0，这是因为两个线程各自的 `synchronized` 锁住的不是同一个对象！这使得两个线程各自都可以同时获得锁：因为JVM只保证同一个锁在任意时刻只能被一个线程获取，但两个不同的锁在同一时刻可以被两个线程分别获取。

因此，使用 `synchronized` 的时候，获取到的是哪个锁非常重要。锁对象如果不对，代码逻辑就不对。

我们再看一个例子：


```

public class Main {
    public static void main(String[] args) throws Exception {
        var ts = new Thread[] { new AddStudentThread(), new DecStudentThread(),
new AddTeacherThread(), new DecTeacherThread() };
        for (var t : ts) {
            t.start();
        }
        for (var t : ts) {
            t.join();
        }
        System.out.println(Counter.studentCount);
        System.out.println(Counter.teacherCount);
    }
}

class Counter {
    public static final Object lock = new Object();
    public static int studentCount = 0;
    public static int teacherCount = 0;
}

class AddStudentThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.studentCount += 1;
            }
        }
    }
}

class DecStudentThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.studentCount -= 1;
            }
        }
    }
}

class AddTeacherThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.teacherCount += 1;
            }
        }
    }
}

class DecTeacherThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.teacherCount -= 1;
            }
        }
    }
}

```

```
    }  
    }  
}
```

上述代码的4个线程对两个共享变量分别进行读写操作，但是使用的锁都是 `Counter.lock` 这一个对象，这就造成了原本可以并发执行的 `Counter.studentCount += 1` 和 `Counter.teacherCount += 1`，现在无法并发执行了，执行效率大大降低。实际上，需要同步的线程可以分成两组：`AddStudentThread` 和 `DecStudentThread`，`AddTeacherThread` 和 `DecTeacherThread`，组之间不存在竞争，因此，应该使用两个不同的锁，即：

`AddStudentThread` 和 `DecStudentThread` 使用 `lockStudent` 锁：

```
synchronized(Counter.lockStudent) {  
    // ...  
}
```

`AddTeacherThread` 和 `DecTeacherThread` 使用 `lockTeacher` 锁：

```
synchronized(Counter.lockTeacher) {  
    // ...  
}
```

这样才能最大化地提高执行效率。

不需要synchronized的操作

JVM规范定义了几种原子操作：

- 基本类型（`long` 和 `double` 除外）赋值，例如：`int n = m;`
- 引用类型赋值，例如：`List list = anotherList。`

`long` 和 `double` 是64位数据，JVM没有明确规定64位赋值操作是不是一个原子操作，不过在x64平台的JVM是把 `long` 和 `double` 的赋值作为原子操作实现的。

单条原子操作的语句不需要同步。例如：

```
public void set(int m) {  
    synchronized(lock) {  
        this.value = m;  
    }  
}
```

就不需要同步。

对引用也是类似。例如：

```
public void set(String s) {  
    this.value = s;  
}
```

上述赋值语句并不需要同步。

但是，如果是多行赋值语句，就必须保证是同步操作，例如：

```

class Pair {
    int first;
    int last;
    public void set(int first, int last) {
        synchronized(this) {
            this.first = first;
            this.last = last;
        }
    }
}

```

有些时候，通过一些巧妙的转换，可以把非原子操作变为原子操作。例如，上述代码如果改造成：

```

class Pair {
    int[] pair;
    public void set(int first, int last) {
        int[] ps = new int[] { first, last };
        this.pair = ps;
    }
}

```

就不再需要同步，因为 `this.pair = ps` 是引用赋值的原子操作。而语句：

```

int[] ps = new int[] { first, last };

```

这里的 `ps` 是方法内部定义的局部变量，每个线程都会有各自的局部变量，互不影响，并且互不可见，并不需要同步。

小结

- 多线程同时读写共享变量时，会造成逻辑错误，因此需要通过 `synchronized` 同步；
- 同步的本质就是给指定对象加锁，加锁后才能继续执行后续代码；
- 注意加锁对象必须是同一个实例；
- 对JVM定义的单个原子操作不需要同步。

同步方法

我们知道Java程序依靠 `synchronized` 对线程进行同步，使用 `synchronized` 的时候，锁住的是哪个对象非常重要。

让线程自己选择锁对象往往会使得代码逻辑混乱，也不利于封装。更好的方法是把 `synchronized` 逻辑封装起来。例如，我们编写一个计数器如下：

```

public class Counter {
    private int count = 0;

    public void add(int n) {
        synchronized(this) {
            count += n;
        }
    }

    public void dec(int n) {
        synchronized(this) {

```

```

        count += n;
    }
}

public int get() {
    return count;
}
}

```

这样一来，线程调用 `add()`、`dec()` 方法时，它不必关心同步逻辑，因为 `synchronized` 代码块在 `add()`、`dec()` 方法内部。并且，我们注意到，`synchronized` 锁住的对象是 `this`，即当前实例，这又使得创建多个 `Counter` 实例的时候，它们之间互不影响，可以并发执行：

```

var c1 = Counter();
var c2 = Counter();

// 对c1进行操作的线程：
new Thread(() -> {
    c1.add();
}).start();
new Thread(() -> {
    c1.dec();
}).start();

// 对c2进行操作的线程：
new Thread(() -> {
    c2.add();
}).start();
new Thread(() -> {
    c2.dec();
}).start();

```

现在，对于 `Counter` 类，多线程可以正确调用。

如果一个类被设计为允许多线程正确访问，我们就说这个类就是“线程安全”的（thread-safe），上面的 `Counter` 类就是线程安全的。Java标准库的 `java.lang.StringBuffer` 也是线程安全的。

还有一些不变类，例如 `String`，`Integer`，`LocalDate`，它们的所有成员变量都是 `final`，多线程同时访问时只能读不能写，这些不变类也是线程安全的。

最后，类似 `Math` 这些只提供静态方法，没有成员变量的类，也是线程安全的。

除了上述几种少数情况，大部分类，例如 `ArrayList`，都是非线程安全的类，我们不能在多线程中修改它们。但是，如果所有线程都只读取，不写入，那么 `ArrayList` 是可以安全地在线程间共享的。

没有特殊说明时，一个类默认是非线程安全的。

我们再观察 `Counter` 的代码：

```

public class Counter {
    public void add(int n) {
        synchronized(this) {
            count += n;
        }
    }
    // ...
}

```

当我们锁住的是 `this` 实例时，实际上可以用 `synchronized` 修饰这个方法。下面两种写法是等价的：

```
public void add(int n) {
    synchronized(this) { // 锁住this
        count += n;
    } // 解锁
}
public synchronized void add(int n) { // 锁住this
    count += n;
} // 解锁
```

因此，用 `synchronized` 修饰的方法就是同步方法，它表示整个方法都必须用 `this` 实例加锁。

我们再思考一下，如果对一个静态方法添加 `synchronized` 修饰符，它锁住的是哪个对象？

```
public synchronized static void test(int n) {
    // ...
}
```

对于 `static` 方法，是没有 `this` 实例的，因为 `static` 方法是针对类而不是实例。但是我们注意到任何一个类都有一个由JVM自动创建的 `Class` 实例，因此，对 `static` 方法添加 `synchronized`，锁住的是该类的 `class` 实例。上述 `synchronized static` 方法实际上相当于：

```
public class Counter {
    public static void test(int n) {
        synchronized(Counter.class) {
            // ...
        }
    }
}
```

我们再考察 `Counter` 的 `get()` 方法：

```
public class Counter {
    private int count;

    public int get() {
        return count;
    }
    // ...
}
```

它没有同步，因为读一个 `int` 变量不需要同步。

然而，如果我们把代码稍微改一下，返回一个包含两个 `int` 的对象：

```

public class Counter {
    private int first;
    private int last;

    public Pair get() {
        Pair p = new Pair();
        p.first = first;
        p.last = last;
        return p;
    }
    ...
}

```

就必须要同步了。

小结

- 用 `synchronized` 修饰方法可以把整个方法变为同步代码块，`synchronized` 方法加锁对象是 `this`；
- 通过合理的设计和封装可以让一个类变为“线程安全”；
- 一个类没有特殊说明，默认不是thread-safe；-多线程能否安全访问某个非线程安全的实例，需要具体问题具体分析。

死锁

Java的线程锁是可重入的锁。

什么是可重入的锁？我们还是来看例子：

```

public class Counter {
    private int count = 0;

    public synchronized void add(int n) {
        if (n < 0) {
            dec(-n);
        } else {
            count += n;
        }
    }

    public synchronized void dec(int n) {
        count -= n;
    }
}

```

观察 `synchronized` 修饰的 `add()` 方法，一旦线程执行到 `add()` 方法内部，说明它已经获取了当前实例的 `this` 锁。如果传入的 `n < 0`，将在 `add()` 方法内部调用 `dec()` 方法。由于 `dec()` 方法也需要获取 `this` 锁，现在问题来了：

对同一个线程，能否在获取到锁以后继续获取同一个锁？

答案是肯定的。JVM允许同一个线程重复获取同一个锁，这种能被同一个线程反复获取的锁，就叫做可重入锁。

由于Java的线程锁是可重入锁，所以，获取锁的时候，不但要判断是否是第一次获取，还要记录这是第几次获取。每获取一次锁，记录+1，每退出 `synchronized` 块，记录-1，减到0的时候，才会真正释放锁。

死锁

一个线程可以获取一个锁后，再继续获取另一个锁。例如：

```
public void add(int m) {
    synchronized(lockA) { // 获得lockA的锁
        this.value += m;
        synchronized(lockB) { // 获得lockB的锁
            this.another += m;
        } // 释放lockB的锁
    } // 释放lockA的锁
}

public void dec(int m) {
    synchronized(lockB) { // 获得lockB的锁
        this.another -= m;
        synchronized(lockA) { // 获得lockA的锁
            this.value -= m;
        } // 释放lockA的锁
    } // 释放lockB的锁
}
```

在获取多个锁的时候，不同线程获取多个不同对象的锁可能导致死锁。对于上述代码，线程1和线程2如果分别执行 `add()` 和 `dec()` 方法时：

- 线程1：进入 `add()`，获得 `lockA`；
- 线程2：进入 `dec()`，获得 `lockB`。

随后：

- 线程1：准备获得 `lockB`，失败，等待中；
- 线程2：准备获得 `lockA`，失败，等待中。

此时，两个线程各自持有不同的锁，然后各自试图获取对方手里的锁，造成了双方无限等待下去，这就是死锁。

死锁发生后，没有任何机制能解除死锁，只能强制结束JVM进程。

因此，在编写多线程应用时，要特别注意防止死锁。因为死锁一旦形成，就只能强制结束进程。

那么我们应该如何避免死锁呢？答案是：线程获取锁的顺序要一致。即严格按照先获取 `lockA`，再获取 `lockB` 的顺序，改写 `dec()` 方法如下：

```
public void dec(int m) {
    synchronized(lockA) { // 获得lockA的锁
        this.value -= m;
        synchronized(lockB) { // 获得lockB的锁
            this.another -= m;
        } // 释放lockB的锁
    } // 释放lockA的锁
}
```

练习

请观察死锁的代码输出，然后修复。

下载练习：[死锁](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- Java的 `synchronized` 锁是可重入锁；
- 死锁产生的条件是多线程各自持有不同的锁，并互相试图获取对方已持有的锁，导致无限等待； - 避免死锁的方法是多线程获取锁的顺序要一致。

使用wait和notify

在Java程序中，`synchronized` 解决了多线程竞争的问题。例如，对于一个任务管理器，多个线程同时往队列中添加任务，可以用 `synchronized` 加锁：

```
class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
    }
}
```

但是 `synchronized` 并没有解决多线程协调的问题。

仍然以上面的 `TaskQueue` 为例，我们再编写一个 `getTask()` 方法取出队列的第一个任务：

```
class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
    }

    public synchronized String getTask() {
        while (queue.isEmpty()) {
        }
        return queue.remove();
    }
}
```

上述代码看上去没有问题：`getTask()` 内部先判断队列是否为空，如果为空，就循环等待，直到另一个线程往队列中放入了一个任务，`while()` 循环退出，就可以返回队列的元素了。

但实际上 `while()` 循环永远不会退出。因为线程在执行 `while()` 循环时，已经在 `getTask()` 入口获取了 `this` 锁，其他线程根本无法调用 `addTask()`，因为 `addTask()` 执行条件也是获取 `this` 锁。

因此，执行上述代码，线程会在 `getTask()` 中因为死循环而100%占用CPU资源。

如果深入思考一下，我们想要的执行效果是：

- 线程1可以调用 `addTask()` 不断往队列中添加任务；
- 线程2可以调用 `getTask()` 从队列中获取任务。如果队列为空，则 `getTask()` 应该等待，直到队列中至少有一个任务时再返回。

因此，多线程协调运行的原则就是：当条件不满足时，线程进入等待状态；当条件满足时，线程被唤醒，继续执行任务。

对于上述 `TaskQueue`，我们先改造 `getTask()` 方法，在条件不满足时，线程进入等待状态：

```
public synchronized String getTask() {
    while (queue.isEmpty()) {
        this.wait();
    }
    return queue.remove();
}
```

当一个线程执行到 `getTask()` 方法内部的 `while` 循环时，它必定已经获取到了 `this` 锁，此时，线程执行 `while` 条件判断，如果条件成立（队列为空），线程将执行 `this.wait()`，进入等待状态。

这里的关键是：`wait()` 方法必须在当前获取的锁对象上调用，这里获取的是 `this` 锁，因此调用 `this.wait()`。

调用 `wait()` 方法后，线程进入等待状态，`wait()` 方法不会返回，直到将来某个时刻，线程从等待状态被其他线程唤醒后，`wait()` 方法才会返回，然后，继续执行下一条语句。

有些仔细的童鞋会指出：即使线程在 `getTask()` 内部等待，其他线程如果拿不到 `this` 锁，照样无法执行 `addTask()`，肿么办？

这个问题的关键就在于 `wait()` 方法的执行机制非常复杂。首先，它不是一个普通的Java方法，而是定义在 `Object` 类的一个 `native` 方法，也就是由JVM的C代码实现的。其次，必须在 `synchronized` 块中才能调用 `wait()` 方法，因为 `wait()` 方法调用时，会释放线程获得的锁，`wait()` 方法返回后，线程又会重新试图获得锁。

因此，只能在锁对象上调用 `wait()` 方法。因为在 `getTask()` 中，我们获得了 `this` 锁，因此，只能在 `this` 对象上调用 `wait()` 方法：

```
public synchronized String getTask() {
    while (queue.isEmpty()) {
        // 释放this锁：
        this.wait();
        // 重新获取this锁
    }
    return queue.remove();
}
```

当一个线程在 `this.wait()` 等待时，它就会释放 `this` 锁，从而使得其他线程能够在 `addTask()` 方法获得 `this` 锁。

现在我们面临第二个问题：如何让等待的线程被重新唤醒，然后从 `wait()` 方法返回？答案是在相同的锁对象上调用 `notify()` 方法。我们修改 `addTask()` 如下：

```
public synchronized void addTask(String s) {
    this.queue.add(s);
    this.notify(); // 唤醒在this锁等待的线程
}
```

注意到在往队列中添加了任务后，线程立刻对 `this` 锁对象调用 `notify()` 方法，这个方法会唤醒一个正在 `this` 锁等待的线程（就是在 `getTask()` 中位于 `this.wait()` 的线程），从而使得等待线程从 `this.wait()` 方法返回。

我们来看一个完整的例子：

```
import java.util.*;
public class Main {
    public static void main(String[] args) throws InterruptedException {
        var q = new TaskQueue();
        var ts = new ArrayList<Thread>();
        for (int i=0; i<5; i++) {
            var t = new Thread() {
                public void run() {
                    // 执行task:
                    while (true) {
                        try {
                            String s = q.getTask();
                            System.out.println("execute task: " + s);
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                }
            };
            t.start();
            ts.add(t);
        }
        var add = new Thread(() -> {
            for (int i=0; i<10; i++) {
                // 放入task:
                String s = "t-" + Math.random();
                System.out.println("add task: " + s);
                q.addTask(s);
                try { Thread.sleep(100); } catch (InterruptedException e) {}
            }
        });
        add.start();
        add.join();
        Thread.sleep(100);
        for (var t : ts) {
            t.interrupt();
        }
    }
}

class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
        this.notifyAll();
    }

    public synchronized String getTask() throws InterruptedException {
        while (queue.isEmpty()) {
            this.wait();
        }
        return queue.remove();
    }
}
```

这个例子中，我们重点关注 `addTask()` 方法，内部调用了 `this.notifyAll()` 而不是 `this.notify()`，使用 `notifyAll()` 将唤醒所有当前正在 `this` 锁等待的线程，而 `notify()` 只会唤醒其中一个（具体哪个依赖操作系统，有一定的随机性）。这是因为可能有多个线程正在 `getTask()` 方法内部的 `wait()` 中等待，使用 `notifyAll()` 将一次性全部唤醒。通常来说，`notifyAll()` 更安全。有些时候，如果我们的代码逻辑考虑不周，用 `notify()` 会导致只唤醒了一个线程，而其他线程可能永远等待下去醒不过来了。

但是，注意到 `wait()` 方法返回时需要重新获得 `this` 锁。假设当前有3个线程被唤醒，唤醒后，首先要等待执行 `addTask()` 的线程结束此方法后，才能释放 `this` 锁，随后，这3个线程中只能有一个获取到 `this` 锁，剩下两个将继续等待。

再注意到我们在 `while()` 循环中调用 `wait()`，而不是 `if` 语句：

```
public synchronized String getTask() throws InterruptedException {  
    if (queue.isEmpty()) {  
        this.wait();  
    }  
    return queue.remove();  
}
```

这种写法实际上是错误的，因为线程被唤醒时，需要再次获取 `this` 锁。多个线程被唤醒后，只有一个线程能获取 `this` 锁，此刻，该线程执行 `queue.remove()` 可以获取到队列的元素，然而，剩下的线程如果获取 `this` 锁后执行 `queue.remove()`，此刻队列可能已经没有任何元素了，所以，要始终在 `while` 循环中 `wait()`，并且每次被唤醒后拿到 `this` 锁就必须再次判断：

```
while (queue.isEmpty()) {  
    this.wait();  
}
```

所以，正确编写多线程代码是非常困难的，需要仔细考虑的条件非常多，任何一个地方考虑不周，都会导致多线程运行时不正常。



小结

`wait` 和 `notify` 用于多线程协调运行：

- 在 `synchronized` 内部可以调用 `wait()` 使线程进入等待状态；
- 必须在已获得的锁对象上调用 `wait()` 方法；
- 在 `synchronized` 内部可以调用 `notify()` 或 `notifyAll()` 唤醒其他等待线程；
- 必须在已获得的锁对象上调用 `notify()` 或 `notifyAll()` 方法；
- 已唤醒的线程还需要重新获得锁后才能继续执行。

使用ReentrantLock

从Java 5开始，引入了一个高级的处理并发的 `java.util.concurrent` 包，它提供了大量更高级的并发功能，能大大简化多线程程序的编写。

我们知道Java语言直接提供了 `synchronized` 关键字用于加锁，但这种锁一是很重，二是获取时必须一直等待，没有额外的尝试机制。

`java.util.concurrent.locks` 包提供的 `ReentrantLock` 用于替代 `synchronized` 加锁，我们来看一下传统的 `synchronized` 代码：

```
public class Counter {
    private int count;

    public void add(int n) {
        synchronized(this) {
            count += n;
        }
    }
}
```

如果用 `ReentrantLock` 替代，可以把代码改造为：

```
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int count;

    public void add(int n) {
        lock.lock();
        try {
            count += n;
        } finally {
            lock.unlock();
        }
    }
}
```

因为 `synchronized` 是Java语言层面提供的语法，所以我们不需要考虑异常，而 `ReentrantLock` 是Java代码实现的锁，我们就必须先获取锁，然后在 `finally` 中正确释放锁。

顾名思义，`ReentrantLock` 是可重入锁，它和 `synchronized` 一样，一个线程可以多次获取同一个锁。

和 `synchronized` 不同的是，`ReentrantLock` 可以尝试获取锁：

```
if (lock.tryLock(1, TimeUnit.SECONDS)) {
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

上述代码在尝试获取锁的时候，最多等待1秒。如果1秒后仍未获取到锁，`tryLock()` 返回 `false`，程序就可以做一些额外处理，而不是无限等待下去。

所以，使用 `ReentrantLock` 比直接使用 `synchronized` 更安全，线程在 `tryLock()` 失败的时候不会导致死锁。

小结

- `ReentrantLock` 可以替代 `synchronized` 进行同步；
- `ReentrantLock` 获取锁更安全；
- 必须先获取到锁，再进入 `try {...}` 代码块，最后使用 `finally` 保证释放锁；
- 可以使用 `tryLock()` 尝试获取锁。

使用Condition

使用 `ReentrantLock` 比直接使用 `synchronized` 更安全，可以替代 `synchronized` 进行线程同步。

但是，`synchronized` 可以配合 `wait` 和 `notify` 实现线程在条件不满足时等待，条件满足时唤醒，用 `ReentrantLock` 我们怎么编写 `wait` 和 `notify` 的功能呢？

答案是使用 `Condition` 对象来实现 `wait` 和 `notify` 的功能。

我们仍然以 `TaskQueue` 为例，把前面用 `synchronized` 实现的功能通过 `ReentrantLock` 和 `Condition` 来实现：

```
class TaskQueue {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private Queue<String> queue = new LinkedList<>();

    public void addTask(String s) {
        lock.lock();
        try {
            queue.add(s);
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public String getTask() {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                condition.await();
            }
            return queue.remove();
        } finally {

```

```
        lock.unlock();
    }
}
```

可见，使用 `Condition` 时，引用的 `Condition` 对象必须从 `Lock` 实例的 `newCondition()` 返回，这样才能获得一个绑定了 `Lock` 实例的 `Condition` 实例。

`Condition` 提供的 `await()`、`signal()`、`signalAll()` 原理和 `synchronized` 锁对象的 `wait()`、`notify()`、`notifyAll()` 是一致的，并且其行为也是一样的：

- `await()` 会释放当前锁，进入等待状态；
- `signal()` 会唤醒某个等待线程；
- `signalAll()` 会唤醒所有等待线程；
- 唤醒线程从 `await()` 返回后需要重新获得锁。

此外，和 `tryLock()` 类似，`await()` 可以在等待指定时间后，如果还没有被其他线程通过 `signal()` 或 `signalAll()` 唤醒，可以自己醒来：

```
if (condition.await(1, TimeUnit.SECOND)) {
    // 被其他线程唤醒
} else {
    // 指定时间内没有被其他线程唤醒
}
```

可见，使用 `Condition` 配合 `Lock`，我们可以实现更灵活的线程同步。

小结

- `Condition` 可以替代 `wait` 和 `notify`；
- `Condition` 对象必须从 `Lock` 对象获取。

使用 `ReadWriteLock`

前面讲到的 `ReentrantLock` 保证了只有一个线程可以执行临界区代码：

```
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int[] counts = new int[10];

    public void inc(int index) {
        lock.lock();
        try {
            counts[index] += 1;
        } finally {
            lock.unlock();
        }
    }

    public int[] get() {
        lock.lock();
        try {
            return Arrays.copyOf(counts, counts.length);
        } finally {
            lock.unlock();
        }
    }
}
```

```
    }  
    }  
}
```

但是有些时候，这种保护有点过头。因为我们发现，任何时刻，只允许一个线程修改，也就是调用 `inc()` 方法是必须获取锁，但是，`get()` 方法只读取数据，不修改数据，它实际上允许多个线程同时调用。

实际上我们想要的是：允许多个线程同时读，但只要有一个线程在写，其他线程就必须等待：

	读	写
读	允许	不允许
写	不允许	不允许

使用 `ReadWriteLock` 可以解决这个问题，它保证：

- 只允许一个线程写入（其他线程既不能写入也不能读取）；
- 没有写入时，多个线程允许同时读（提高性能）。

用 `ReadWriteLock` 实现这个功能十分容易。我们需要创建一个 `ReadWriteLock` 实例，然后分别获取读锁和写锁：

```
public class Counter {  
    private final ReadWriteLock rwlock = new ReentrantReadWriteLock();  
    private final Lock rlock = rwlock.readLock();  
    private final Lock wlock = rwlock.writeLock();  
    private int[] counts = new int[10];  
  
    public void inc(int index) {  
        wlock.lock(); // 加写锁  
        try {  
            counts[index] += 1;  
        } finally {  
            wlock.unlock(); // 释放写锁  
        }  
    }  
  
    public int[] get() {  
        rlock.lock(); // 加读锁  
        try {  
            return Arrays.copyOf(counts, counts.length);  
        } finally {  
            rlock.unlock(); // 释放读锁  
        }  
    }  
}
```

把读写操作分别用读锁和写锁来加锁，在读取时，多个线程可以同时获得读锁，这样就大大提高了并发读的执行效率。

使用 `ReadWriteLock` 时，适用条件是同一个数据，有大量线程读取，但仅有少数线程修改。

例如，一个论坛的帖子，回复可以看做写入操作，它是不频繁的，但是，浏览可以看做读取操作，是非常频繁的，这种情况就可以使用 `ReadWriteLock`。

小结

使用 `ReadWriteLock` 可以提高读取效率：

- `ReadWriteLock` 只允许一个线程写入；
- `ReadWriteLock` 允许多个线程在没有写入时同时读取；
- `ReadWriteLock` 适合读多写少的场景。

使用StampedLock

前面介绍的 `ReadWriteLock` 可以解决多线程同时读，但只有一个线程能写的问题。

如果我们深入分析 `ReadWriteLock`，会发现它有个潜在的问题：如果有线程正在读，写线程需要等待读线程释放锁后才能获取写锁，即读的过程中不允许写，这是一种悲观的读锁。

要进一步提升并发执行效率，Java 8引入了新的读写锁：`StampedLock`。

`StampedLock` 和 `ReadWriteLock` 相比，改进之处在于：读的过程中也允许获取写锁后写入！这样一来，我们读的数据就可能不一致，所以，需要一点额外的代码来判断读的过程中是否有写入，这种读锁是一种乐观锁。

乐观锁的意思就是乐观地估计读的过程中大概率不会有写入，因此被称为乐观锁。反过来，悲观锁则是读的过程中拒绝有写入，也就是写入必须等待。显然乐观锁的并发效率更高，但一旦有小概率的写入导致读取的数据不一致，需要能检测出来，再读一遍就行。

我们来看例子：

```
public class Point {
    private final StampedLock stampedLock = new StampedLock();

    private double x;
    private double y;

    public void move(double deltaX, double deltaY) {
        long stamp = stampedLock.writeLock(); // 获取写锁
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            stampedLock.unlockWrite(stamp); // 释放写锁
        }
    }

    public double distanceFromOrigin() {
        long stamp = stampedLock.tryOptimisticRead(); // 获得一个乐观读锁
        // 注意下面两行代码不是原子操作
        // 假设x,y = (100,200)
        double currentX = x;
        // 此处已读取到x=100，但x,y可能被写线程修改为(300,400)
        double currentY = y;
        // 此处已读取到y，如果没有写入，读取是正确的(100,200)
        // 如果有写入，读取是错误的(100,400)
        if (!stampedLock.validate(stamp)) { // 检查乐观读锁后是否有其他写锁发生
            stamp = stampedLock.readLock(); // 获取一个悲观读锁
            try {
                currentX = x;
                currentY = y;
            }
        }
    }
}
```



```

        } finally {
            stampedLock.unlockRead(stamp); // 释放悲观读锁
        }
    }
    return Math.sqrt(currentX * currentX + currentY * currentY);
}
}

```

和 `ReadWriteLock` 相比，写入的加锁是完全一样的，不同的是读取。注意到首先我们通过 `tryOptimisticRead()` 获取一个乐观读锁，并返回版本号。接着进行读取，读取完成后，我们通过 `validate()` 去验证版本号，如果在读取过程中没有写入，版本号不变，验证成功，我们就可以放心地继续后续操作。如果在读取过程中有写入，版本号会发生变化，验证将失败。在失败的时候，我们再通过获取悲观读锁再次读取。由于写入的概率不高，程序在绝大部分情况下可以通过乐观读锁获取数据，极少数情况下使用悲观读锁获取数据。

可见，`StampedLock` 把读锁细分为乐观读和悲观读，能进一步提升并发效率。但这也是有代价的：一是代码更加复杂，二是 `StampedLock` 是不可重入锁，不能在一个线程中反复获取同一个锁。

`StampedLock` 还提供了更复杂的将悲观读锁升级为写锁的功能，它主要使用在 if-then-update 的场景：即先读，如果读的数据满足条件，就返回，如果读的数据不满足条件，再尝试写。

小结

- `StampedLock` 提供了乐观读锁，可取代 `ReadWriteLock` 以进一步提升并发性能；
- `StampedLock` 是不可重入锁。

使用 Concurrent 集合

我们在前面已经通过 `ReentrantLock` 和 `Condition` 实现了一个 `BlockingQueue`：

```

public class TaskQueue {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private Queue<String> queue = new LinkedList<>();

    public void addTask(String s) {
        lock.lock();
        try {
            queue.add(s);
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public String getTask() {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                condition.await();
            }
            return queue.remove();
        } finally {
            lock.unlock();
        }
    }
}

```

```
}
```

BlockingQueue 的意思就是说，当一个线程调用这个 TaskQueue 的 `getTask()` 方法时，该方法内部可能会让线程变成等待状态，直到队列条件满足不为空，线程被唤醒后，`getTask()` 方法才会返回。

因为 BlockingQueue 非常有用，所以我们不必自己编写，可以直接使用 Java 标准库的 `java.util.concurrent` 包提供的线程安全的集合：`ArrayBlockingQueue`。

除了 BlockingQueue 外，针对 List、Map、Set、Deque 等，`java.util.concurrent` 包也提供了对应的并发集合类。我们归纳一下：

interface	non-thread-safe	thread-safe
List	ArrayList	CopyOnWriteArrayList
Map	HashMap	ConcurrentHashMap
Set	HashSet / TreeSet	CopyOnWriteArraySet
Queue	ArrayDeque / LinkedList	ArrayBlockingQueue / LinkedBlockingQueue
Deque	ArrayDeque / LinkedList	LinkedBlockingDeque

使用这些并发集合与使用非线程安全的集合类完全相同。我们以 `ConcurrentHashMap` 为例：

```
Map<String, String> map = ConcurrentHashMap<>();  
// 在不同的线程读写：  
map.put("A", "1");  
map.put("B", "2");  
map.get("A", "1");
```

因为所有的同步和加锁的逻辑都在集合内部实现，对外部调用者来说，只需要正常按接口引用，其他代码和原来的非线程安全代码完全一样。即当我们需要多线程访问时，把：

```
Map<String, String> map = HashMap<>();
```

改为：

```
Map<String, String> map = ConcurrentHashMap<>();
```

就可以了。

`java.util.Collections` 工具类还提供了一个旧的线程安全集合转换器，可以这么用：

```
Map unsafeMap = new HashMap();  
Map threadSafeMap = Collections.synchronizedMap(unsafeMap);
```

但是它实际上是用一个包装类包装了非线程安全的 Map，然后对所有读写方法都用 `synchronized` 加锁，这样获得的线程安全集合的性能比 `java.util.concurrent` 集合要低很多，所以不推荐使用。

小结

- 使用 `java.util.concurrent` 包提供的线程安全的并发集合可以大大简化多线程编程；
- 多线程同时读写并发集合是安全的；
- 尽量使用Java标准库提供的并发集合，避免自己编写同步代码。

使用Atomic

Java的 `java.util.concurrent` 包除了提供底层锁、并发集合外，还提供了一组原子操作的封装类，它们位于 `java.util.concurrent.atomic` 包。

我们以 `AtomicInteger` 为例，它提供的主要操作有：

- 增加值并返回新值：`int addAndGet(int delta)`
- 加1后返回新值：`int incrementAndGet()`
- 获取当前值：`int get()`
- 用CAS方式设置：`int compareAndSet(int expect, int update)`

Atomic类是通过无锁（lock-free）的方式实现的线程安全（thread-safe）访问。它的主要原理是利用了CAS：Compare and Set。

如果我们自己通过CAS编写 `incrementAndGet()`，它大概长这样：

```
public int incrementAndGet(AtomicInteger var) {
    int prev, next;
    do {
        prev = var.get();
        next = prev + 1;
    } while ( ! var.compareAndSet(prev, next));
    return prev;
}
```

CAS是指，在这个操作中，如果 `AtomicInteger` 的当前值是 `prev`，那么就更新为 `next`，返回 `true`。如果 `AtomicInteger` 的当前值不是 `prev`，就什么也不干，返回 `false`。通过CAS操作并配合 `do ... while` 循环，即使其他线程修改了 `AtomicInteger` 的值，最终的结果也是正确的。

我们利用 `AtomicLong` 可以编写一个多线程安全的全局唯一ID生成器：

```
class IdGenerator {
    AtomicLong var = new AtomicLong(0);

    public long getNextId() {
        return var.incrementAndGet();
    }
}
```

通常情况下，我们并不需要直接用 `do ... while` 循环调用 `compareAndSet` 实现复杂的并发操作，而是用 `incrementAndGet()` 这样的封装好的方法，因此，使用起来非常简单。

在高度竞争的情况下，还可以使用Java 8提供的 `LongAdder` 和 `LongAccumulator`。

小结

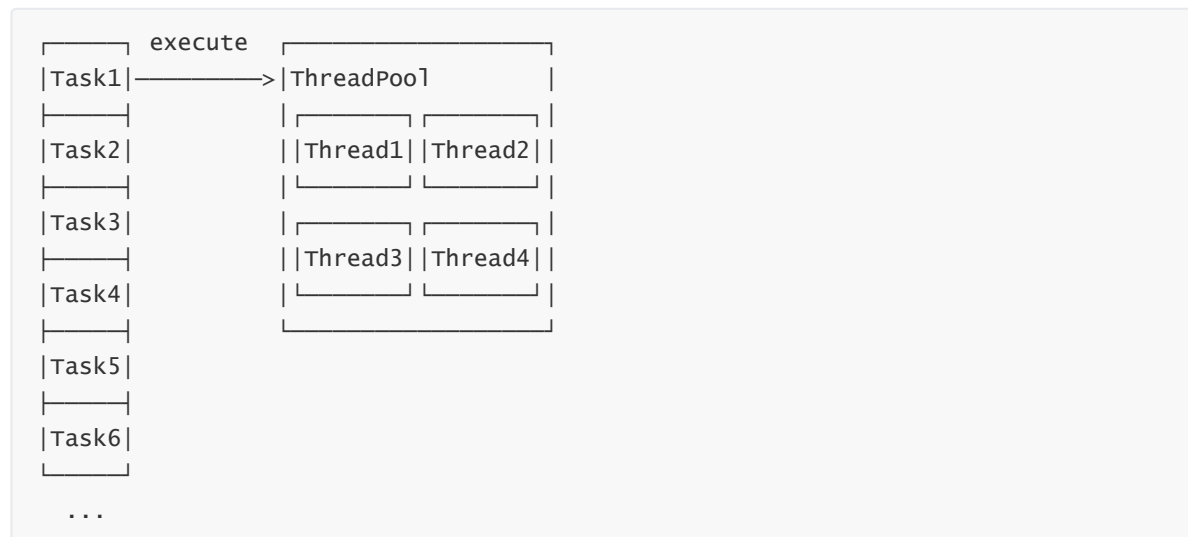
使用 `java.util.concurrent.atomic` 提供的原子操作可以简化多线程编程：

- 原子操作实现了无锁的线程安全；
- 适用于计数器，累加器等。

使用线程池

Java语言虽然内置了多线程支持，启动一个新线程非常方便，但是，创建线程需要操作系统资源（线程资源，栈空间等），频繁创建和销毁大量线程需要消耗大量时间。

如果可以复用一组线程：



那么我们就可以把很多小任务让一组线程来执行，而不是一个任务对应一个新线程。这种能接收大量小任务并进行分发处理的就是线程池。

简单地说，线程池内部维护了若干个线程，没有任务的时候，这些线程都处于等待状态。如果有新任务，就分配一个空闲线程执行。如果所有线程都处于忙碌状态，新任务要么放入队列等待，要么增加一个新线程进行处理。

Java标准库提供了 `ExecutorService` 接口表示线程池，它的典型用法如下：

```
// 创建固定大小的线程池：
ExecutorService executor = Executors.newFixedThreadPool(3);
// 提交任务：
executor.submit(task1);
executor.submit(task2);
executor.submit(task3);
executor.submit(task4);
executor.submit(task5);
```

因为 `ExecutorService` 只是接口，Java标准库提供的几个常用实现类有：

- `FixedThreadPool`：线程数固定的线程池；
- `CachedThreadPool`：线程数根据任务动态调整的线程池；
- `SingleThreadExecutor`：仅单线程执行的线程池。

创建这些线程池的方法都被封装到 `Executors` 这个类中。我们以 `FixedThreadPool` 为例，看看线程池的执行逻辑：

```

import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        // 创建一个固定大小的线程池：
        ExecutorService es = Executors.newFixedThreadPool(4);
        for (int i = 0; i < 6; i++) {
            es.submit(new Task("" + i));
        }
        // 关闭线程池：
        es.shutdown();
    }
}

class Task implements Runnable {
    private final String name;

    public Task(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println("start task " + name);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        System.out.println("end task " + name);
    }
}

```

我们观察执行结果，一次性放入6个任务，由于线程池只有固定的4个线程，因此，前4个任务会同时执行，等到有线程空闲后，才会执行后面的两个任务。

线程池在程序结束的时候要关闭。使用 `shutdown()` 方法关闭线程池的时候，它会等待正在执行的任务先完成，然后再关闭。`shutdownNow()` 会立刻停止正在执行的任务，`awaitTermination()` 则会等待指定的时间让线程池关闭。

如果我们把线程池改为 `CachedThreadPool`，由于这个线程池的实现会根据任务数量动态调整线程池的大小，所以6个任务可一次性全部同时执行。

如果我们想把线程池的大小限制在4~10个之间动态调整怎么办？我们查看

`Executors.newCachedThreadPool()` 方法的源码：

```

public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                   60L, TimeUnit.SECONDS,
                                   new SynchronousQueue<Runnable>());
}

```

因此，想创建指定动态范围的线程池，可以这么写：

```
int min = 4;
int max = 10;
ExecutorService es = new ThreadPoolExecutor(min, max,
        60L, TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
```

ScheduledThreadPool

还有一种任务，需要定期反复执行，例如，每秒刷新证券价格。这种任务本身固定，需要反复执行的，可以使用 `ScheduledThreadPool`。放入 `ScheduledThreadPool` 的任务可以定期反复执行。

创建一个 `ScheduledThreadPool` 仍然是通过 `Executors` 类：

```
ScheduledExecutorService ses = Executors.newScheduledThreadPool(4);
```

我们可以提交一次性任务，它会在指定延迟后只执行一次：

```
// 1秒后执行一次性任务：
ses.schedule(new Task("one-time"), 1, TimeUnit.SECONDS);
```

如果任务以固定的每3秒执行，我们可以这样写：

```
// 2秒后开始执行定时任务，每3秒执行：
ses.scheduleAtFixedRate(new Task("fixed-rate"), 2, 3, TimeUnit.SECONDS);
```

如果任务以固定的3秒为间隔执行，我们可以这样写：

```
// 3秒后开始执行定时任务，以3秒为间隔执行：
ses.scheduleWithFixedDelay(new Task("fixed-delay"), 2, 3, TimeUnit.SECONDS);
```

注意 `FixedRate` 和 `FixedDelay` 的区别。`FixedRate` 是指任务总是以固定时间间隔触发，不管任务执行多长时间：



而 `FixedDelay` 是指，上一次任务执行完毕后，等待固定的时间间隔，再执行下一次任务：



因此，使用 `ScheduledThreadPool` 时，我们要根据需求选择执行一次、`FixedRate` 执行还是 `FixedDelay` 执行。

细心的童鞋还可以思考下面的问题：

- 在 `FixedRate` 模式下，假设每秒触发，如果某次任务执行时间超过1秒，后续任务会不会并发执行？
- 如果任务抛出了异常，后续任务是否继续执行？

Java标准库还提供了一个 `java.util.Timer` 类，这个类也可以定期执行任务，但是，一个 `Timer` 会对应一个 `Thread`，所以，一个 `Timer` 只能定期执行一个任务，多个定时任务必须启动多个 `Timer`，而一个 `ScheduledThreadPool` 就可以调度多个定时任务，所以，我们完全可以用 `ScheduledThreadPool` 取代旧的 `Timer`。

练习

下载练习：[使用线程池](#)（推荐使用[IDE练习插件](#)快速下载）

小结

JDK提供了 `ExecutorService` 实现了线程池功能：

- 线程池内部维护一组线程，可以高效执行大量小任务；
- `Executors` 提供了静态方法创建不同类型的 `ExecutorService`；
- 必须调用 `shutdown()` 关闭 `ExecutorService`；
- `ScheduledThreadPool` 可以定期调度多个任务。

使用Future

在执行多个任务的时候，使用Java标准库提供的线程池是非常方便的。我们提交的任务只需要实现 `Runnable` 接口，就可以让线程池去执行：

```
class Task implements Runnable {
    public String result;

    public void run() {
        this.result = longTimeCalculation();
    }
}
```

`Runnable` 接口有个问题，它的方法没有返回值。如果任务需要一个返回结果，那么只能保存到变量，还要提供额外的方法读取，非常不便。所以，Java标准库还提供了一个 `Callable` 接口，和 `Runnable` 接口比，它多了一个返回值：

```
class Task implements Callable<String> {
    public String call() throws Exception {
        return longTimeCalculation();
    }
}
```

并且 `Callable` 接口是一个泛型接口，可以返回指定类型的结果。

现在的问题是，如何获得异步执行的结果？

如果仔细看 `ExecutorService.submit()` 方法，可以看到，它返回了一个 `Future` 类型，一个 `Future` 类型的实例代表一个未来能获取结果的对象：

```
ExecutorService executor = Executors.newFixedThreadPool(4);
// 定义任务:
Callable<String> task = new Task();
// 提交任务并获得Future:
Future<String> future = executor.submit(task);
// 从Future获取异步执行返回的结果:
String result = future.get(); // 可能阻塞
```

当我们提交一个 `Callable` 任务后，我们会同时获得一个 `Future` 对象，然后，我们在主线程某个时刻调用 `Future` 对象的 `get()` 方法，就可以获得异步执行的结果。在调用 `get()` 时，如果异步任务已经完成，我们就直接获得结果。如果异步任务还没有完成，那么 `get()` 会阻塞，直到任务完成后才返回结果。

一个 `Future` 接口表示一个未来可能会返回的结果，它定义的方法有：

- `get()`：获取结果（可能会等待）
- `get(long timeout, TimeUnit unit)`：获取结果，但只等待指定的时间；
- `cancel(boolean mayInterruptIfRunning)`：取消当前任务；
- `isDone()`：判断任务是否已完成。

练习

下载练习：[使用Future](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- 对线程池提交一个 `Callable` 任务，可以获得一个 `Future` 对象；
- 可以用 `Future` 在将来某个时刻获取结果。

使用CompletableFuture

使用 `Future` 获得异步执行结果时，要么调用阻塞方法 `get()`，要么轮询看 `isDone()` 是否为 `true`，这两种方法都不是很好，因为主线程也会被迫等待。

从Java 8开始引入了 `CompletableFuture`，它针对 `Future` 做了改进，可以传入回调对象，当异步任务完成或者发生异常时，自动调用回调对象的回调方法。

我们以获取股票价格为例，看看如何使用 `CompletableFuture`：

```
// CompletableFuture
import java.util.concurrent.CompletableFuture;
public class Main {
    public static void main(String[] args) throws Exception {
        // 创建异步执行任务:
        CompletableFuture<Double> cf =
        CompletableFuture.supplyAsync(Main::fetchPrice);
        // 如果执行成功:
        cf.thenAccept((result) -> {
            System.out.println("price: " + result);
        });
        // 如果执行异常:
        cf.exceptionally((e) -> {
            e.printStackTrace();
            return null;
        });
        // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭:
```



```

        Thread.sleep(2000);
    }

    static Double fetchPrice() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        if (Math.random() < 0.3) {
            throw new RuntimeException("fetch price failed!");
        }
        return 5 + Math.random() * 20;
    }
}

```

创建一个 `CompletableFuture` 是通过 `CompletableFuture.supplyAsync()` 实现的，它需要一个实现了 `Supplier` 接口的对象：

```

public interface Supplier<T> {
    T get();
}

```

这里我们用lambda语法简化了一下，直接传入 `Main::fetchPrice`，因为 `Main.fetchPrice()` 静态方法的签名符合 `Supplier` 接口的定义（除了方法名外）。

紧接着，`CompletableFuture` 已经被提交给默认的线程池执行了，我们需要定义的是 `CompletableFuture` 完成时和异常时需要回调的实例。完成时，`CompletableFuture` 会调用 `Consumer` 对象：

```

public interface Consumer<T> {
    void accept(T t);
}

```

异常时，`CompletableFuture` 会调用 `Function` 对象：

```

public interface Function<T, R> {
    R apply(T t);
}

```

这里我们都用lambda语法简化了代码。

可见 `CompletableFuture` 的优点是：

- 异步任务结束时，会自动回调某个对象的方法；
- 异步任务出错时，会自动回调某个对象的方法；
- 主线程设置好回调后，不再关心异步任务的执行。

如果只是实现了异步回调机制，我们还看不出 `CompletableFuture` 相比 `Future` 的优势。`CompletableFuture` 更强大的功能是，多个 `CompletableFuture` 可以串行执行，例如，定义两个 `CompletableFuture`，第一个 `CompletableFuture` 根据证券名称查询证券代码，第二个 `CompletableFuture` 根据证券代码查询证券价格，这两个 `CompletableFuture` 实现串行操作如下：

```

// CompletableFuture
import java.util.concurrent.CompletableFuture;

```

```

public class Main {
    public static void main(String[] args) throws Exception {
        // 第一个任务:
        CompletableFuture<String> cfQuery = CompletableFuture.supplyAsync(() ->
{
            return queryCode("中国石油");
        });
        // cfQuery成功后继续执行下一个任务:
        CompletableFuture<Double> cfFetch = cfQuery.thenApplyAsync((code) -> {
            return fetchPrice(code);
        });
        // cfFetch成功后打印结果:
        cfFetch.thenAccept((result) -> {
            System.out.println("price: " + result);
        });
        // 主线程不要立刻结束, 否则CompletableFuture默认使用的线程池会立刻关闭:
        Thread.sleep(2000);
    }

    static String queryCode(String name) {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {}
        return "601857";
    }

    static Double fetchPrice(String code) {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {}
        return 5 + Math.random() * 20;
    }
}

```

除了串行执行外, 多个 `CompletableFuture` 还可以并行执行。例如, 我们考虑这样的场景:

同时从新浪和网易查询证券代码, 只要任意一个返回结果, 就进行下一步查询价格, 查询价格也同时从新浪和网易查询, 只要任意一个返回结果, 就完成操作:

```

// CompletableFuture
import java.util.concurrent.CompletableFuture;
public class Main {
    public static void main(String[] args) throws Exception {
        // 两个CompletableFuture执行异步查询:
        CompletableFuture<String> cfQueryFromSina =
        CompletableFuture.supplyAsync(() -> {
            return queryCode("中国石油", "https://finance.sina.com.cn/code/");
        });
        CompletableFuture<String> cfQueryFrom163 =
        CompletableFuture.supplyAsync(() -> {
            return queryCode("中国石油", "https://money.163.com/code/");
        });

        // 用anyOf合并为一个新的CompletableFuture:
    }
}

```

```

CompletableFuture<Object> cfQuery =
CompletableFuture.anyOf(cfQueryFromSina, cfQueryFrom163);

// 两个CompletableFuture执行异步查询:
CompletableFuture<Double> cfFetchFromSina =
cfQuery.thenApplyAsync((code) -> {
    return fetchPrice((String) code,
"https://finance.sina.com.cn/price/");
});
CompletableFuture<Double> cfFetchFrom163 = cfQuery.thenApplyAsync((code)
-> {
    return fetchPrice((String) code, "https://money.163.com/price/");
});

// 用anyOf合并为一个新的CompletableFuture:
CompletableFuture<Object> cfFetch =
CompletableFuture.anyOf(cfFetchFromSina, cfFetchFrom163);

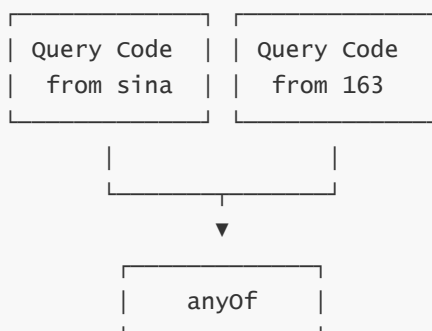
// 最终结果:
cfFetch.thenAccept((result) -> {
    System.out.println("price: " + result);
});
// 主线程不要立刻结束, 否则CompletableFuture默认使用的线程池会立刻关闭:
Thread.sleep(2000);
}

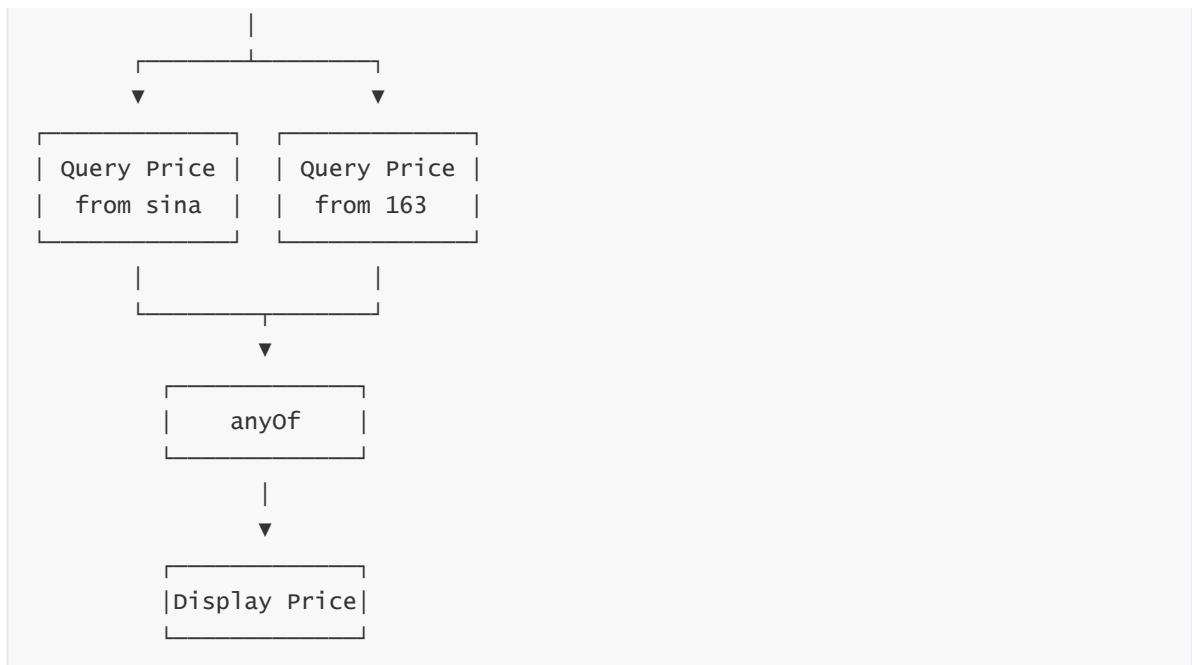
static String queryCode(String name, String url) {
    System.out.println("query code from " + url + "...");
    try {
        Thread.sleep((long) (Math.random() * 1000));
    } catch (InterruptedException e) {
    }
    return "601857";
}

static Double fetchPrice(String code, String url) {
    System.out.println("query price from " + url + "...");
    try {
        Thread.sleep((long) (Math.random() * 1000));
    } catch (InterruptedException e) {
    }
    return 5 + Math.random() * 20;
}
}

```

上述逻辑实现的异步查询规则实际上是:





除了 `anyOf()` 可以实现“任意个 `CompletableFuture` 只要一个成功”，`allOf()` 可以实现“所有 `CompletableFuture` 都必须成功”，这些组合操作可以实现非常复杂的异步流程控制。

最后我们注意 `CompletableFuture` 的命名规则：

- `xxx()`：表示该方法将继续在已有的线程中执行；
- `xxxAsync()`：表示将异步在线程池中执行。

练习

下载练习：[使用CompletableFuture](#)（推荐使用[IDE练习插件](#)快速下载）

小结

`CompletableFuture` 可以指定异步处理流程：

- `thenAccept()` 处理正常结果；
- `exceptional()` 处理异常结果；
- `thenApplyAsync()` 用于串行化另一个 `CompletableFuture`；
- `anyOf()` 和 `allOf()` 用于并行化多个 `CompletableFuture`。

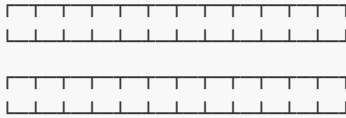
使用ForkJoin

Java 7开始引入了一种新的Fork/Join线程池，它可以执行一种特殊的任务：把一个大任务拆成多个小任务并行执行。

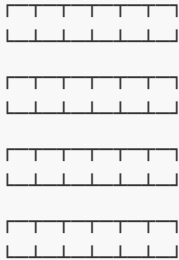
我们举个例子：如果要计算一个超大数组的和，最简单的做法是用一个循环在一个线程内完成：



还有一种方法，可以把数组拆成两部分，分别计算，最后加起来就是最终结果，这样可以用两个线程并行执行：



如果拆成两部分还是很大，我们还可以继续拆，用4个线程并行执行：



这就是Fork/Join任务的原理：判断一个任务是否足够小，如果是，直接计算，否则，就拆分成几个小任务分别计算。这个过程可以反复“裂变”成一系列小任务。

我们来看如何使用Fork/Join对大数据进行并行求和：

```
import java.util.Random;
import java.util.concurrent.*;
public class Main {
    public static void main(String[] args) throws Exception {
        // 创建2000个随机数组成的数组：
        long[] array = new long[2000];
        long expectedSum = 0;
        for (int i = 0; i < array.length; i++) {
            array[i] = random();
            expectedSum += array[i];
        }
        System.out.println("Expected sum: " + expectedSum);
        // fork/join:
        ForkJoinTask<Long> task = new SumTask(array, 0, array.length);
        long startTime = System.currentTimeMillis();
        Long result = ForkJoinPool.commonPool().invoke(task);
        long endTime = System.currentTimeMillis();
        System.out.println("Fork/join sum: " + result + " in " + (endTime -
startTime) + " ms.");
    }

    static Random random = new Random(0);

    static long random() {
        return random.nextInt(10000);
    }
}

class SumTask extends RecursiveTask<Long> {
    static final int THRESHOLD = 500;
    long[] array;
    int start;
    int end;

    SumTask(long[] array, int start, int end) {
```

```

        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        if (end - start <= THRESHOLD) {
            // 如果任务足够小,直接计算:
            long sum = 0;
            for (int i = start; i < end; i++) {
                sum += this.array[i];
                // 故意放慢计算速度:
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                }
            }
            return sum;
        }
        // 任务太大,一分为二:
        int middle = (end + start) / 2;
        System.out.println(String.format("split %d~%d ==> %d~%d, %d~%d", start,
end, start, middle, middle, end));
        SumTask subtask1 = new SumTask(this.array, start, middle);
        SumTask subtask2 = new SumTask(this.array, middle, end);
        invokeAll(subtask1, subtask2);
        Long subresult1 = subtask1.join();
        Long subresult2 = subtask2.join();
        Long result = subresult1 + subresult2;
        System.out.println("result = " + subresult1 + " + " + subresult2 + " ==>
" + result);
        return result;
    }
}

```

观察上述代码的执行过程, 一个大的计算任务0~2000首先分裂为两个小任务0~1000和1000~2000, 这两个小任务仍然太大, 继续分裂为更小的0~500, 500~1000, 1000~1500, 1500~2000, 最后, 计算结果被依次合并, 得到最终结果。

因此, 核心代码 SumTask 继承自 RecursiveTask, 在 compute() 方法中, 关键是如何“分裂”出子任务并且提交子任务:

```

class SumTask extends RecursiveTask<Long> {
    protected Long compute() {
        // “分裂”子任务:
        SumTask subtask1 = new SumTask(...);
        SumTask subtask2 = new SumTask(...);
        // invokeAll会并行运行两个子任务:
        invokeAll(subtask1, subtask2);
        // 获得子任务的结果:
        Long result1 = fork1.join();
        Long result2 = fork2.join();
        // 汇总结果:
        return result1 + result2;
    }
}

```

Fork/Join线程池在Java标准库中就有应用。Java标准库提供的 `java.util.Arrays.parallelSort(array)` 可以进行并行排序，它的原理就是内部通过Fork/Join对大数组分拆进行并行排序，在多核CPU上就可以大大提高排序的速度。

练习

下载练习：[使用Fork/Join](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- Fork/Join是一种基于“分治”的算法：通过分解任务，并行执行，最后合并结果得到最终结果。
- `ForkJoinPool` 线程池可以把一个大任务分拆成小任务并行执行，任务类必须继承自 `RecursiveTask` 或 `RecursiveAction`。
- 使用Fork/Join模式可以进行并行计算以提高效率。

使用ThreadLocal

多线程是Java实现多任务的基础，`Thread` 对象代表一个线程，我们可以在代码中调用 `Thread.currentThread()` 获取当前线程。例如，打印日志时，可以同时打印出当前线程的名字：

```
public class Main {
    public static void main(String[] args) throws Exception {
        log("start main...");
        new Thread(() -> {
            log("run task...");
        }).start();
        new Thread(() -> {
            log("print...");
        }).start();
        log("end main.");
    }

    static void log(String s) {
        System.out.println(Thread.currentThread().getName() + ": " + s);
    }
}
```

对于多任务，Java标准库提供的线程池可以方便地执行这些任务，同时复用线程。Web应用程序就是典型的多任务应用，每个用户请求页面时，我们都会创建一个任务，类似：

```
public void process(User user) {
    checkPermission();
    doWork();
    saveStatus();
    sendResponse();
}
```

然后，通过线程池去执行这些任务。

观察 `process()` 方法，它内部需要调用若干其他方法，同时，我们遇到一个问题：如何在一个线程内传递状态？

`process()` 方法需要传递的状态就是 `User` 实例。有的童鞋会想，简单地传入 `User` 就可以了：

```
public void process(User user) {
    checkPermission(user);
    doWork(user);
    saveStatus(user);
    sendResponse(user);
}
```

但是往往一个方法又会调用其他很多方法，这样会导致 `User` 传递到所有地方：

```
void doWork(User user) {
    queryStatus(user);
    checkStatus();
    setNewStatus(user);
    log();
}
```

这种在一个线程中，横跨若干方法调用，需要传递的对象，我们通常称之为上下文（Context），它是一种状态，可以是用户身份、任务信息等。

给每个方法增加一个 context 参数非常麻烦，而且有些时候，如果调用链有无法修改源码的第三方库，`User` 对象就传不进去了。

Java 标准库提供了一个特殊的 `ThreadLocal`，它可以在一个线程中传递同一个对象。

`ThreadLocal` 实例通常总是以静态字段初始化如下：

```
static ThreadLocal<String> threadLocalUser = new ThreadLocal<>();
```

它的典型使用方式如下：

```
void processUser(user) {
    try {
        threadLocalUser.set(user);
        step1();
        step2();
    } finally {
        threadLocalUser.remove();
    }
}
```

通过设置一个 `User` 实例关联到 `ThreadLocal` 中，在移除之前，所有方法都可以随时获取到该 `User` 实例：

```
void step1() {
    User u = threadLocalUser.get();
    log();
    printUser();
}

void log() {
    User u = threadLocalUser.get();
    println(u.name);
}

void step2() {
```



```
    User u = threadLocalUser.get();
    checkUser(u.id);
}
```

注意到普通的方法调用一定是同一个线程执行的，所以，`step1()`、`step2()` 以及 `log()` 方法内，`threadLocalUser.get()` 获取的 `user` 对象是同一个实例。

实际上，可以把 `ThreadLocal` 看成一个全局 `Map`：每个线程获取 `ThreadLocal` 变量时，总是使用 `Thread` 自身作为 `key`：

```
Object threadLocalValue = threadLocalMap.get(Thread.currentThread());
```

因此，`ThreadLocal` 相当于给每个线程都开辟了一个独立的存储空间，各个线程的 `ThreadLocal` 关联的实例互不干扰。

最后，特别注意 `ThreadLocal` 一定要在 `finally` 中清除：

```
try {
    threadLocalUser.set(user);
    // ...
} finally {
    threadLocalUser.remove();
}
```

这是因为当前线程执行完相关代码后，很可能会被重新放入线程池中，如果 `ThreadLocal` 没有被清除，该线程执行其他代码时，会把上一次的状态带进去。

为了保证能释放 `ThreadLocal` 关联的实例，我们可以通过 `AutoCloseable` 接口配合 `try (resource) {...}` 结构，让编译器自动为我们关闭。例如，一个保存了当前用户名的 `ThreadLocal` 可以封装为一个 `UserContext` 对象：

```
public class UserContext implements AutoCloseable {

    static final ThreadLocal<String> ctx = new ThreadLocal<>();

    public UserContext(String user) {
        ctx.set(user);
    }

    public static String currentUser() {
        return ctx.get();
    }

    @Override
    public void close() {
        ctx.remove();
    }
}
```

使用的时候，我们借助 `try (resource) {...}` 结构，可以这么写：

```
try (var ctx = new UserContext("Bob")) {  
    // 可任意调用UserContext.currentUser():  
    String currentUser = UserContext.currentUser();  
} // 在此自动调用UserContext.close()方法释放ThreadLocal关联对象
```

这样就在 `UserContext` 中完全封装了 `ThreadLocal`，外部代码在 `try (resource) {...}` 内部可以随时调用 `UserContext.currentUser()` 获取当前线程绑定的用户名。

练习

下载练习：[ThreadLocal练习](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- `ThreadLocal` 表示线程的“局部变量”，它确保每个线程的 `ThreadLocal` 变量都是各自独立的；
- `ThreadLocal` 适合在一个线程的处理流程中保持上下文（避免了同一参数在所有方法中传递）；
- 使用 `ThreadLocal` 要用 `try ... finally` 结构，并在 `finally` 中清除。