# Thead概述

## 构造器

•public Thread() :分配一个新的线程对象。

• public Thread(String name) :分配一个指定名字的新的线程对象。

• public Thread(Runnable target) :指定创建线程的目标对象，它实现了 Runnable 接口

中的 run 方法

• public Thread(Runnable target,String name) :分配一个带有指定目标新的线程对象并指

定名字。

## 常用方法系列

•public void run() :此线程要执行的任务在此处定义代码。

• public void start() :导致此线程开始执行; Java 虚拟机调用此线程的 run 方法。

• public String getName() :获取当前线程名称。

• public void setName(String name)：设置该线程名称。

• public static Thread currentThread() :返回对当前正在执行的线程对象的引用。在

Thread 子类中就是 this，通常用于主线程和 Runnable 实现类

• public static void sleep(long millis) :使当前正在执行的线程以指定的毫秒数暂停（暂时

停止执行）。

• public static void yield()：yield 只是让当前线程暂停一下，让系统的线程调度器重新

调度一次，希望优先级与当前线程相同或更高的其他线程能够获得执行机会，但是这

个不能保证，完全有可能的情况是，当某个线程调用了 yield 方法暂停之后，线程调

度器又将其调度出来重新执行。

•public final boolean isAlive()：测试线程是否处于活动状态。如果线程已经启动且尚未

终止，则为活动状态。

• void join() ：等待该线程终止。

void join(long millis) ：等待该线程终止的时间最长为 millis 毫秒。如果 millis 时间

到，将不再等待。

void join(long millis, int nanos) ：等待该线程终止的时间最长为 millis 毫秒 +

nanos 纳秒。

• public final void stop()：已过时，不建议使用。强行结束一个线程的执行，直接进入

死亡状态。run()即刻停止，可能会导致一些清理性的工作得不到完成，如文件，数据

库等的关闭。同时，会立即释放该线程所持有的所有的锁，导致数据得不到同步的处

理，出现数据不一致的问题。

• void suspend() / void resume() : 这两个操作就好比播放器的暂停和恢复。二者必须成

对出现，否则非常容易发生死锁。suspend()调用会导致线程暂停，但不会释放任何锁

资源，导致其它线程都无法访问被它占用的锁，直到调用 resume()。已过时，不建议

使用。

每个线程都有一定的优先级，同优先级线程组成先进先出队列（先到先服

务），使用分时调度策略。优先级高的线程采用抢占式策略，获得较多的执行

机会。每个线程默认的优先级都与创建它的父线程具有相同的优先级。

• Thread 类的三个优先级常量：

– MAX\_PRIORITY（10）：最高优先级

– MIN \_PRIORITY （1）：最低优先级

– NORM\_PRIORITY （5）：普通优先级，默认情况下 main 线程具有普通优先

级。

• public final int getPriority() ：返回线程优先级

• public final void setPriority(int newPriority) ：改变线程的优先级，范围在[1,10]之间。

## 守护线程

有一种线程，它是在后台运行的，它的任务是为其他线程提供服务的，这种线

程被称为“守护线程”。JVM 的垃圾回收线程就是典型的守护线程。

守护线程有个特点，就是如果所有非守护线程都死亡，那么守护线程自动死

亡。形象理解：兔死狗烹，鸟尽弓藏

调用 setDaemon(true)方法可将指定线程设置为守护线程。必须在线程启动之前

设置，否则会报 IllegalThreadStateException 异常。

调用 isDaemon()可以判断线程是否是守护线程。

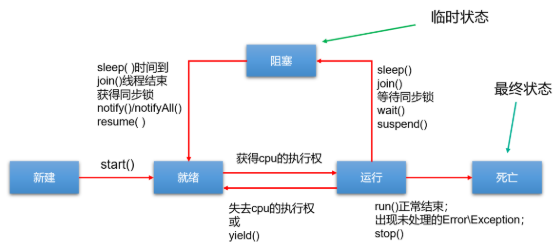
# 多线程的生命周期

## JDK1.5 之前：5 种状态

线程的生命周期有五种状态：新建（New）、就绪（Runnable）、运行

（Running）、阻塞（Blocked）、死亡（Dead）。CPU 需要在多条线程之间切

换，于是线程状态会多次在运行、阻塞、就绪之间切换。



1.新建

当一个 Thread 类或其子类的对象被声明并创建时，新生的线程对象处于新建状

态。此时它和其他 Java 对象一样，仅仅由 JVM 为其分配了内存，并初始化了

实例变量的值。此时的线程对象并没有任何线程的动态特征，程序也不会执行

它的线程体 run()。

2.就绪

但是当线程对象调用了 start()方法之后，就不一样了，线程就从新建状态转为

就绪状态。JVM 会为其创建方法调用栈和程序计数器，当然，处于这个状态中

的线程并没有开始运行，只是表示已具备了运行的条件，随时可以被调度。至

于什么时候被调度，取决于 JVM 里线程调度器的调度。

注意：

程序只能对新建状态的线程调用 start()，并且只能调用一次，如果对

非新建状态的线程，如已启动的线程或已死亡的线程调用 start()都会

报错 IllegalThreadStateException 异常。

3.运行

如果处于就绪状态的线程获得了 CPU 资源时，开始执行 run()方法的线程体代

码，则该线程处于运行状态。如果计算机只有一个 CPU 核心，在任何时刻只有

一个线程处于运行状态，如果计算机有多个核心，将会有多个线程并行

(Parallel)执行。

当然，美好的时光总是短暂的，而且 CPU 讲究雨露均沾。对于抢占式策略的系

统而言，系统会给每个可执行的线程一个小时间段来处理任务，当该时间用

完，系统会剥夺该线程所占用的资源，让其回到就绪状态等待下一次被调度。

此时其他线程将获得执行机会，而在选择下一个线程时，系统会适当考虑线程

的优先级。

4.阻塞

当在运行过程中的线程遇到如下情况时，会让出 CPU 并临时中止自己的执

行，进入阻塞状态：

• 线程调用了 sleep()方法，主动放弃所占用的 CPU 资源；

• 线程试图获取一个同步监视器，但该同步监视器正被其他线程持有；

• 线程执行过程中，同步监视器调用了 wait()，让它等待某个通知（notify）；

• 线程执行过程中，同步监视器调用了 wait(time)

• 线程执行过程中，遇到了其他线程对象的加塞（join）；

• 线程被调用 suspend 方法被挂起（已过时，因为容易发生死锁）；

当前正在执行的线程被阻塞后，其他线程就有机会执行了。针对如上情况，当

发生如下情况时会解除阻塞，让该线程重新进入就绪状态，等待线程调度器再

次调度它：

• 线程的 sleep()时间到；

• 线程成功获得了同步监视器；

• 线程等到了通知(notify)；

• 线程 wait 的时间到了

• 加塞的线程结束了；

• 被挂起的线程又被调用了 resume 恢复方法（已过时，因为容易发生死锁）；

5.死亡

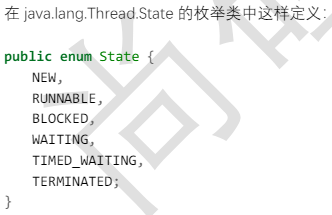
线程会以以下三种方式之一结束，结束后的线程就处于死亡状态：

• run()方法执行完成，线程正常结束

• 线程执行过程中抛出了一个未捕获的异常（Exception）或错误（Error）

• 直接调用该线程的 stop()来结束该线程（已过时）

## JDK1.5 及之后：6 种状态



• NEW（新建）：线程刚被创建，但是并未启动。还没调用 start 方法。

• RUNNABLE（可运行）：这里没有区分就绪和运行状态。因为对于 Java 对象来说，只

能标记为可运行，至于什么时候运行，不是 JVM 来控制的了，是 OS 来进行调度的，

而且时间非常短暂，因此对于 Java 对象的状态来说，无法区分。

• Teminated（被终止）：表明此线程已经结束生命周期，终止运行。

• 重点说明，根据 Thread.State 的定义，阻塞状态分为三种：BLOCKED、WAITING、

TIMED\_WAITING。

– BLOCKED（锁阻塞）：在 API 中的介绍为：一个正在阻塞、等待一个监视

器锁（锁对象）的线程处于这一状态。只有获得锁对象的线程才能有执行

机会。

• 比如，线程 A 与线程 B 代码中使用同一锁，如果线程 A 获取到

锁，线程 A 进入到 Runnable 状态，那么线程 B 就进入到 Blocked

锁阻塞状态。

– TIMED\_WAITING（计时等待）：在 API 中的介绍为：一个正在限时等待

另一个线程执行一个（唤醒）动作的线程处于这一状态。

• 当前线程执行过程中遇到 Thread 类的 sleep 或 join，Object 类

的 wait，LockSupport 类的 park 方法，并且在调用这些方法时，

设置了时间，那么当前线程会进入 TIMED\_WAITING，直到时间

到，或被中断。

– WAITING（无限等待）：在 API 中介绍为：一个正在无限期等待另一个线

程执行一个特别的（唤醒）动作的线程处于这一状态。

• 当前线程执行过程中遇到遇到 Object 类的 wait，Thread 类的

join，LockSupport 类的 park 方法，并且在调用这些方法时，没

有指定时间，那么当前线程会进入 WAITING 状态，直到被唤醒。

– 通过 Object 类的 wait 进入 WAITING 状态的要有 Object 的

notify/notifyAll 唤醒；

– 通过 Condition 的 await 进入 WAITING 状态的要有

Condition 的 signal 方法唤醒；

– 通过 LockSupport 类的 park 方法进入 WAITING 状态的要有

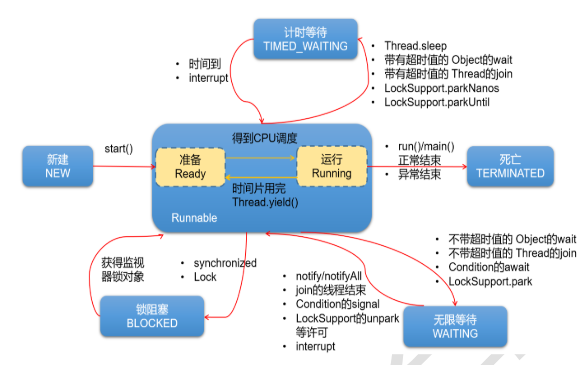
LockSupport 类的 unpark 方法唤醒

– 通过 Thread 类的 join 进入 WAITING 状态，只有调用 join

方法的线程对象结束才能让当前线程恢复；

说明：当从 WAITING 或 TIMED\_WAITING 恢复到 Runnable 状态时，如果发现

当前线程没有得到监视器锁，那么会立刻转入 BLOCKED 状态。



我们在翻阅 API 的时候会发现 Timed Waiting（计时等待） 与

Waiting（无限等待） 状态联系还是很紧密的， 比如 Waiting（无限

等待） 状态中 wait 方法是空参的，而 timed waiting（计时等待） 中

wait 方法是带参的。 这种带参的方法，其实是一种倒计时操作，相当

于我们生活中的小闹钟，我们设定好时间，到时通知，可是 如果提前

得到（唤醒）通知，那么设定好时间在通知也就显得多此一举了，那

么这种设计方案其实是一举两 得。如果没有得到（唤醒）通知，那么

线程就处于 Timed Waiting 状态，直到倒计时完毕自动醒来；如果在

倒 计时期间得到（唤醒）通知，那么线程从 Timed Waiting 状态立刻

唤醒。

# 线程安全问题

## 同步机制解决线程安全问题

当我们使用多个线程访问同一资源（可以是同一个变量、同一个文件、同一条

记录等）的时候，若多个线程只有读操作，那么不会发生线程安全问题。但是如

果多个线程中对资源有读和写的操作，就容易出现线程安全问题。

要解决上述多线程并发访问一个资源的安全性问题:也就是解决重复票与不存在

票问题，Java 中提供了同步机制 (synchronized)来解决。

为了保证每个线程都能正常执行原子操作，Java 引入了线程同步机制。注意:在

任何时候,最多允许一个线程拥有同步锁，谁拿到锁就进入代码块，其他的线程

只能在外等着(BLOCKED)。



同步机制的原理，其实就相当于给某段代码加“锁”，任何线程想要执行这段代

码，都要先获得“锁”，我们称它为同步锁。因为 Java 对象在堆中的数据分为分

为对象头、实例变量、空白的填充。而对象头中包含：

• Mark Word：记录了和当前对象有关的 GC、锁标记等信息。

• 指向类的指针：每一个对象需要记录它是由哪个类创建出来的。

• 数组长度（只有数组对象才有）

哪个线程获得了“同步锁”对象之后，”同步锁“对象就会记录这个线程的 ID，这

样其他线程就只能等待了，除非这个线程”释放“了锁对象，其他线程才能重新

获得/占用”同步锁“对象。

## 同步代码块和同步方法

同步代码块：synchronized 关键字可以用于某个区块前面，表示只对这个区块

的资源实行互斥访问。 格式:

synchronized(同步锁){

需要同步操作的代码

}

同步方法：synchronized 关键字直接修饰方法，表示同一时刻只有一个线程能

进入这个方法，其他线程在外面等着。

public synchronized void method(){

可能会产生线程安全问题的代码

}

## synchronized 的锁是什么

同步锁对象可以是任意类型，但是必须保证竞争“同一个共享资源”的多个线程

必须使用同一个“同步锁对象”。

对于同步代码块来说，同步锁对象是由程序员手动指定的（很多时候也是指定

为 this 或类名.class），但是对于同步方法来说，同步锁对象只能是默认的：

• 静态方法：当前类的 Class 对象（类名.class）

• 非静态方法：this

## 同步操作的思考顺序

1、如何找问题，即代码是否存在线程安全？（非常重要） （1）明确哪些代码

是多线程运行的代码 （2）明确多个线程是否有共享数据 （3）明确多线程运

行代码中是否有多条语句操作共享数据

2、如何解决呢？（非常重要） 对多条操作共享数据的语句，只能让一个线程

都执行完，在执行过程中，其他线程不可以参与执行。 即所有操作共享数据的

这些语句都要放在同步范围中

3、切记：

范围太小：不能解决安全问题

范围太大：因为一旦某个线程抢到锁，其他线程就只能等待，所以范围太大，

效率会降低，不能合理利用 CPU 资源。

## 死锁

不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要

的同步资源，就形成了线程的死锁。

【小故事】

面试官：你能解释清楚什么是死锁，我就录取你！ 面试者：你录取

我，我就告诉你什么是死锁！ …. 恭喜你，面试通过了

一旦出现死锁，整个程序既不会发生异常，也不会给出任何提示，只是所有线

程处于阻塞状态，无法继续。

诱发死锁的原因：

• 互斥条件

• 占用且等待

• 不可抢夺（或不可抢占）

• 循环等待

以上 4 个条件，同时出现就会触发死锁。

解决死锁：

死锁一旦出现，基本很难人为干预，只能尽量规避。可以考虑打破上面的诱发

条件。

针对条件 1：互斥条件基本上无法被破坏。因为线程需要通过互斥解决安全问

题。

针对条件 2：可以考虑一次性申请所有所需的资源，这样就不存在等待的问

题。

针对条件 3：占用部分资源的线程在进一步申请其他资源时，如果申请不到，

就主动释放掉已经占用的资源。

针对条件 4：可以将资源改为线性顺序。申请资源时，先申请序号较小的，这

样避免循环等待问题。

## JDK5.0 新特性：Lock(锁)

• JDK5.0 的新增功能，保证线程的安全。与采用 synchronized 相比，Lock 可提供多种

锁方案，更灵活、更强大。Lock 通过显式定义同步锁对象来实现同步。同步锁使用

Lock 对象充当。

• java.util.concurrent.locks.Lock 接口是控制多个线程对共享资源进行访问的工具。锁提

供了对共享资源的独占访问，每次只能有一个线程对 Lock 对象加锁，线程开始访问

共享资源之前应先获得 Lock 对象。

• 在实现线程安全的控制中，比较常用的是 ReentrantLock，可以显式加锁、释放

锁。

– ReentrantLock 类实现了 Lock 接口，它拥有与 synchronized 相同的并发

性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一

些特性。此外，它还提供了在激烈争用情况下更佳的性能。

• Lock 锁也称同步锁，加锁与释放锁方法，如下：

– public void lock() :加同步锁。

– public void unlock() :释放同步锁。



synchronized 与 Lock 的对比

1. Lock 是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized 是隐式锁，出了

作用域、遇到异常等自动解锁

2. Lock 只有代码块锁，synchronized 有代码块锁和方法锁

3. 使用 Lock 锁，JVM 将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性

（提供更多的子类），更体现面向对象。

4. （了解）Lock 锁可以对读不加锁，对写加锁，synchronized 不可以

5. （了解）Lock 锁可以有多种获取锁的方式，可以从 sleep 的线程中抢到锁，

synchronized 不可以

说明：开发建议中处理线程安全问题优先使用顺序为：

Lock ----> 同步代码块 ----> 同步方法

# 线程的通信

## 为什么要处理线程间通信

当我们需要多个线程来共同完成一件任务，并且我们希望他们有规律的执行，那

么多线程之间需要一些通信机制，可以协调它们的工作，以此实现多线程共同

操作一份数据。

比如：线程 A 用来生产包子的，线程 B 用来吃包子的，包子可以理解为同一资

源，线程 A 与线程 B 处理的动作，一个是生产，一个是消费，此时 B 线程必须

等到 A 线程完成后才能执行，那么线程 A 与线程 B 之间就需要线程通信，即—

— 等待唤醒机制。

## 等待唤醒机制

这是多个线程间的一种协作机制。谈到线程我们经常想到的是线程间的竞争

（race），比如去争夺锁，但这并不是故事的全部，线程间也会有协作机制。

在一个线程满足某个条件时，就进入等待状态（wait() / wait(time)）， 等

待其他线程执行完他们的指定代码过后再将其唤醒（notify()）;或可以指定

wait 的时间，等时间到了自动唤醒；在有多个线程进行等待时，如果需要，可

以使用 notifyAll()来唤醒所有的等待线程。wait/notify 就是线程间的一种协

作机制。

1. wait：线程不再活动，不再参与调度，进入 wait set 中，因此不会浪费 CPU

资源，也不会去竞争锁了，这时的线程状态是 WAITING 或 TIMED\_WAITING。它

还要等着别的线程执行一个特别的动作，也即“通知（notify）”或者等待时间

到，在这个对象上等待的线程从 wait set 中释放出来，重新进入到调度队列

（ready queue）中

2. notify：则选取所通知对象的 wait set 中的一个线程释放；

3. notifyAll：则释放所通知对象的 wait set 上的全部线程。

注意：

被通知的线程被唤醒后也不一定能立即恢复执行，因为它当初中断的

地方是在同步块内，而此刻它已经不持有锁，所以它需要再次尝试去

获取锁（很可能面临其它线程的竞争），成功后才能在当初调用 wait

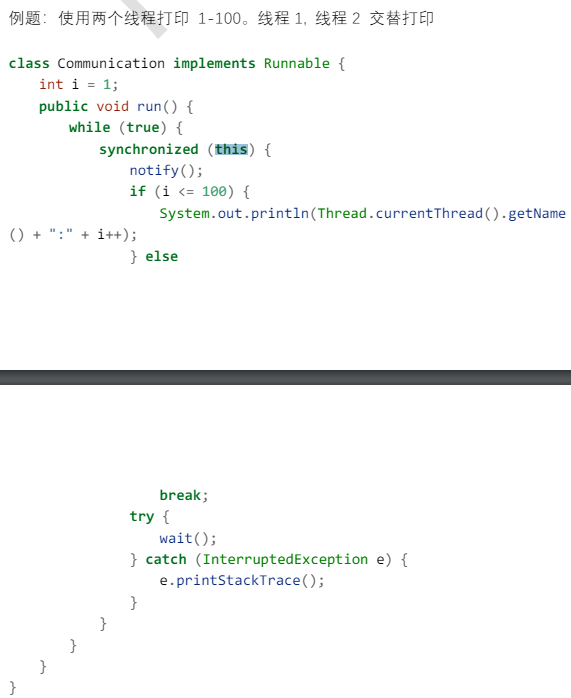
方法之后的地方恢复执行。

总结如下：

• 如果能获取锁，线程就从 WAITING 状态变成 RUNNABLE（可运行）

状态；

• 否则，线程就从 WAITING 状态又变成 BLOCKED（等待锁） 状态



调用 wait 和 notify 需注意的细节

1. wait 方法与 notify 方法必须要由同一个锁对象调用。因为：对应的锁对象可以通

过 notify 唤醒使用同一个锁对象调用的 wait 方法后的线程。

2. wait 方法与 notify 方法是属于 Object 类的方法的。因为：锁对象可以是任意对

象，而任意对象的所属类都是继承了 Object 类的。

3. wait 方法与 notify 方法必须要在同步代码块或者是同步函数中使用。因为：必须

要通过锁对象调用这 2 个方法。否则会报 java.lang.IllegalMonitorStateException 异

常。

# 创建线程的方式

## 继承Thread类

1. 定义 Thread 类的子类，并重写该类的 run()方法，该 run()方法的方法体就代表了线程

需要完成的任务

2. 创建 Thread 子类的实例，即创建了线程对象

3. 调用线程对象的 start()方法来启动该线程

注意：

1.如果自己手动调用 run()方法，那么就只是普通方法，没有启动多线程模

式。

2.run()方法由 JVM 调用，什么时候调用，执行的过程控制都有操作系统的

CPU 调度决定。

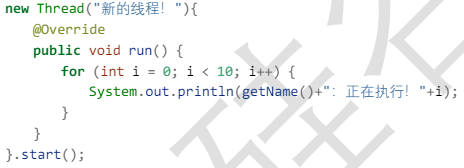
3.想要启动多线程，必须调用 start 方法。

4.一个线程对象只能调用一次 start()方法启动，如果重复调用了，则将抛出

以上的异常“IllegalThreadStateException”。



匿名内部类写法



## 实现 Runnable 接口

Java 有单继承的限制，当我们无法继承 Thread 类时，那么该如何做呢？在核心

类库中提供了 Runnable 接口，我们可以实现 Runnable 接口，重写 run()方法，

然后再通过 Thread 类的对象代理启动和执行我们的线程体 run()方法

通过实现 Runnable 接口，使得该类有了多线程类的特征。所有的分线程要执

行的代码都在 run 方法里面。

在启动的多线程的时候，需要先通过 Thread 类的构造方法 Thread(Runnable

target) 构造出对象，然后调用 Thread 对象的 start()方法来运行多线程代码。

实际上，所有的多线程代码都是通过运行 Thread 的 start()方法来运行的。因

此，不管是继承 Thread 类还是实现 Runnable 接口来实现多线程，最终还是通

过 Thread 的对象的 API 来控制线程的，熟悉 Thread 类的 API 是进行多线程编

程的基础。

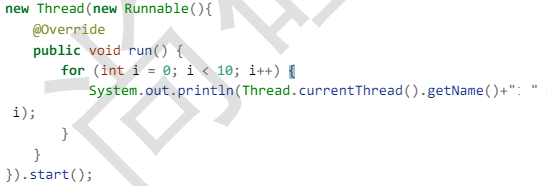
说明：Runnable 对象仅仅作为 Thread 对象的 target，Runnable 实现类里包含

的 run()方法仅作为线程执行体。 而实际的线程对象依然是 Thread 实例，只是

该 Thread 线程负责执行其 target 的 run()方法。



匿名内部类写法



## 实现 Callable 接口

• 与使用 Runnable 相比， Callable 功能更强大些

– 相比 run()方法，可以有返回值

– 方法可以抛出异常

– 支持泛型的返回值（需要借助 FutureTask 类，获取返回结果）

• Future 接口（了解）

– 可以对具体 Runnable、Callable 任务的执行结果进行取消、查询是否完

成、获取结果等。

– FutureTask 是 Futrue 接口的唯一的实现类

– FutureTask 同时实现了 Runnable, Future 接口。它既可以作为 Runnable 被

线程执行，又可以作为 Future 得到 Callable 的返回值

• 缺点：在获取分线程执行结果的时候，当前线程（或是主线程）受阻塞，效率较低。





## 使用线程池

现有问题：

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束

了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线

程需要时间。

那么有没有一种办法使得线程可以复用，即执行完一个任务，并不被销毁，而

是可以继续执行其他的任务？

思路：提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池

中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。

好处：

• 提高响应速度（减少了创建新线程的时间）

• 降低资源消耗（重复利用线程池中线程，不需要每次都创建）

• 便于线程管理

– corePoolSize：核心池的大小

– maximumPoolSize：最大线程数

– keepAliveTime：线程没有任务时最多保持多长时间后会终止

线程池相关 API

• JDK5.0 之前，我们必须手动自定义线程池。从 JDK5.0 开始，Java 内置线程池相关的

API。在 java.util.concurrent 包下提供了线程池相关 API：ExecutorService 和

Executors。

• ExecutorService：真正的线程池接口。常见子类 ThreadPoolExecutor

– void execute(Runnable command) ：执行任务/命令，没有返回值，

一般用来执行 Runnable

– <T> Future<T> submit(Callable<T> task)：执行任务，有返回

值，一般又来执行 Callable

– void shutdown() ：关闭连接池

• Executors：一个线程池的工厂类，通过此类的静态工厂方法可以创建多种类型的线

程池对象。

– Executors.newCachedThreadPool()：创建一个可根据需要创建新线

程的线程池

– Executors.newFixedThreadPool(int nThreads); 创建一个可重用

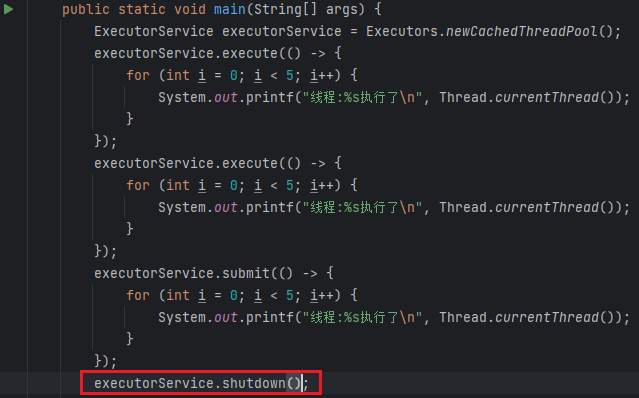
固定线程数的线程池

– Executors.newSingleThreadExecutor() ：创建一个只有一个线程的

线程池

– Executors.newScheduledThreadPool(int corePoolSize)：创建

一个线程池，它可安排在给定延迟后运行命令或者定期地执行。



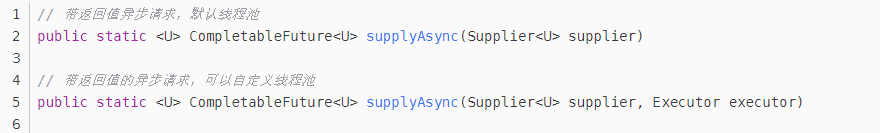
extutorServer执行完成后需要调用shutdown方法,不然程序一直未停止

# CompletableFuture

CompletableFuture是jdk8的新特性。CompletableFuture实现了CompletionStage接口和Future接口，前者是对后者的一个扩展，增加了异步会点、流式处理、多个Future组合处理的能力，使Java在处理多任务的协同工作时更加顺畅便利。

## supplyAsync

supplyAsync是创建带有返回值的异步任务。它有如下两个方法，一个是使用默认线程池（ForkJoinPool.commonPool()）的方法，一个是带有自定义线程池的重载方法



示例

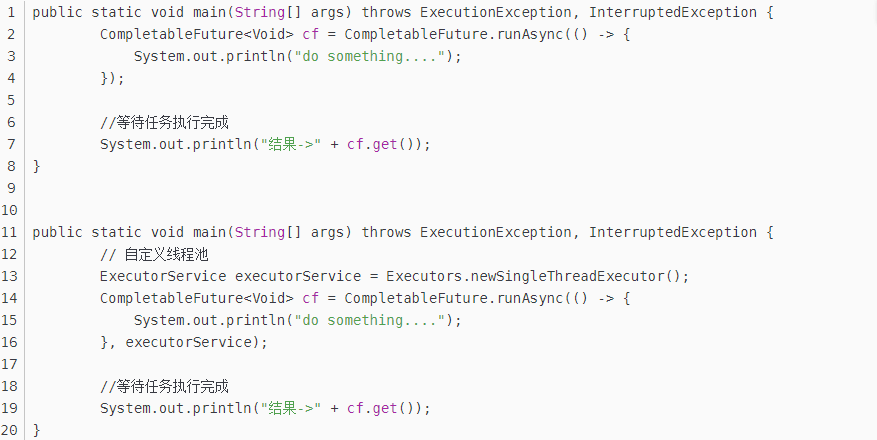


## runAsync

runAsync是创建没有返回值的异步任务。它有如下两个方法，一个是使用默认线程池（ForkJoinPool.commonPool()）的方法，一个是带有自定义线程池的重载方法



示例



## 获取任务结果的方法



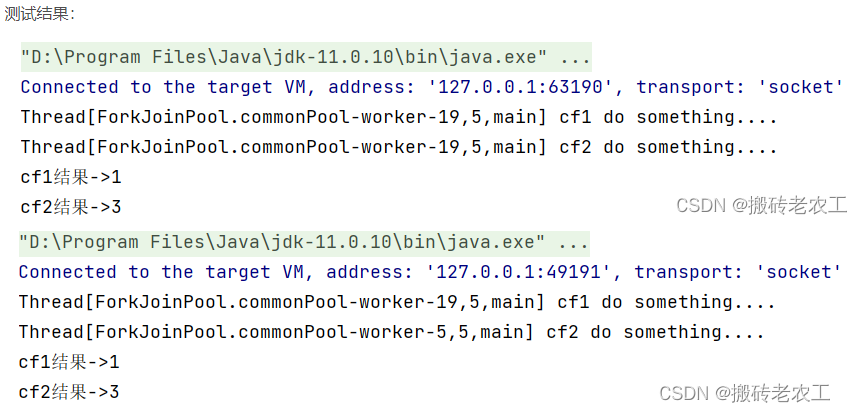
## 异步回调处理

### thenApply和thenApplyAsync

thenApply 表示某个任务执行完成后执行的动作，即回调方法，会将该任务的执行结果即方法返回值作为入参传递到回调方法中，带有返回值。

从下面的代码和测试结果我们发现thenApply和thenApplyAsync区别在于，使用thenApply方法时子任务与父任务使用的是同一个线程，而thenApplyAsync在子任务中是另起一个线程执行任务，并且thenApplyAsync可以自定义线程池，默认的使用ForkJoinPool.commonPool()线程池。

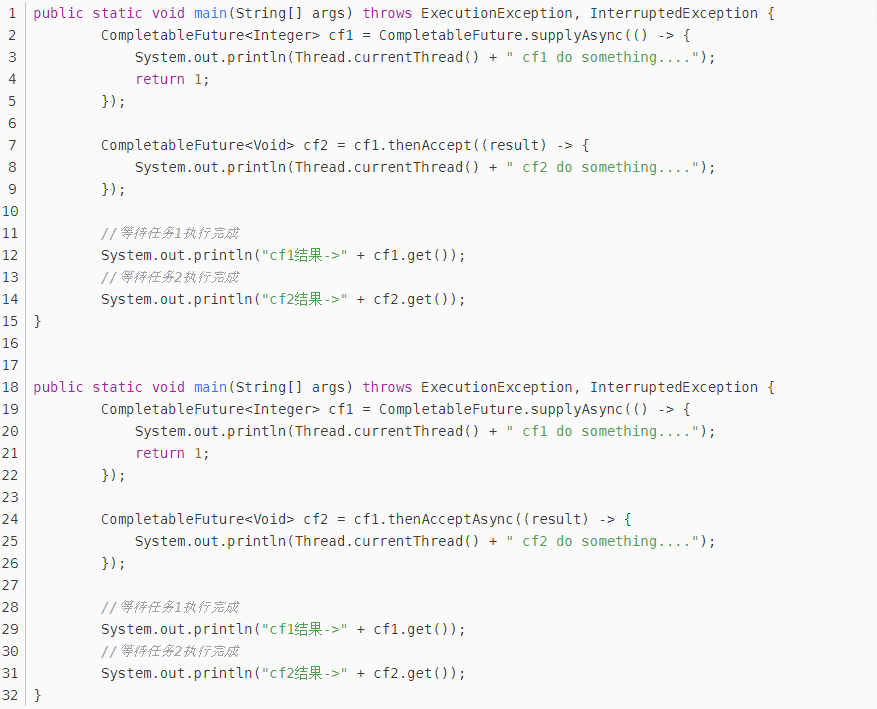


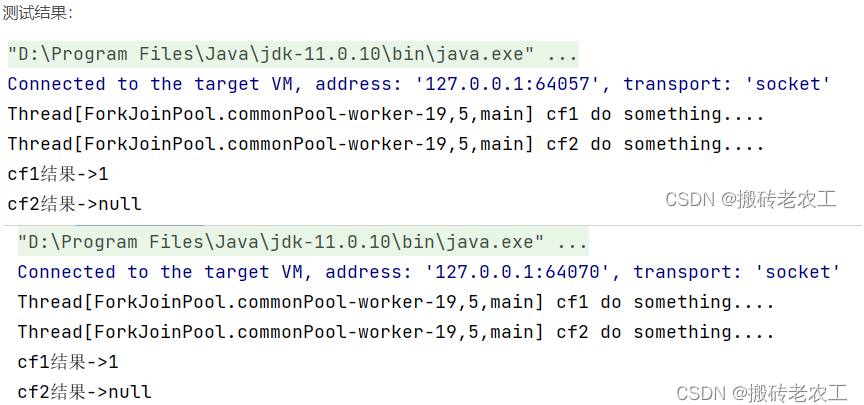


### thenAccept和thenAcceptAsync

thenAccep表示某个任务执行完成后执行的动作，即回调方法，会将该任务的执行结果即方法返回值作为入参传递到回调方法中，无返回值。

测试结果我们发现thenAccep和thenAccepAsync区别在于，使用thenAccep方法时子任务与父任务使用的是同一个线程，而thenAccepAsync在子任务中可能是另起一个线程执行任务，并且thenAccepAsync可以自定义线程池，默认的使用ForkJoinPool.commonPool()线程池。



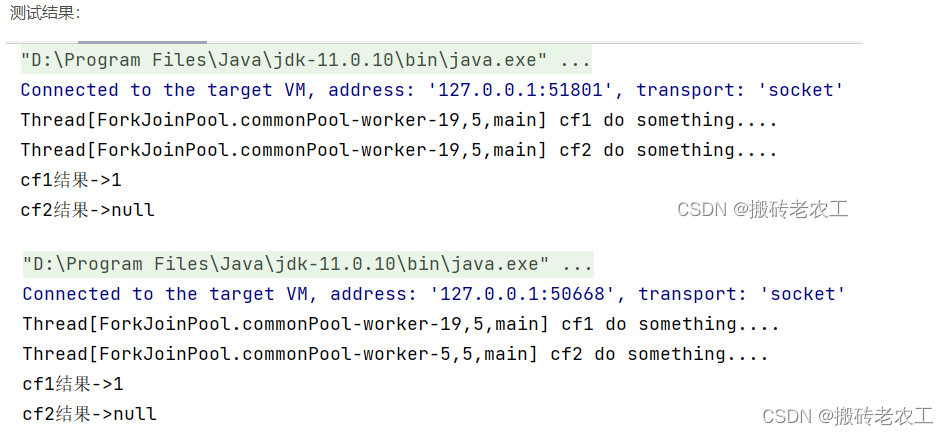


### thenRun和thenRunAsync

thenRun表示某个任务执行完成后执行的动作，即回调方法，无入参，无返回值。

从下面代码和测试结果我们发现thenRun和thenRunAsync区别在于，使用thenRun方法时子任务与父任务使用的是同一个线程，而thenRunAsync在子任务中可能是另起一个线程执行任务，并且thenRunAsync可以自定义线程池，默认的使用ForkJoinPool.commonPool()线程池。

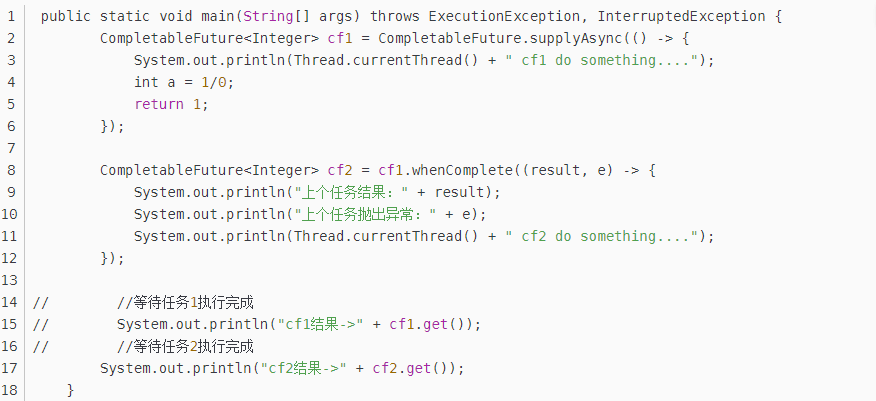


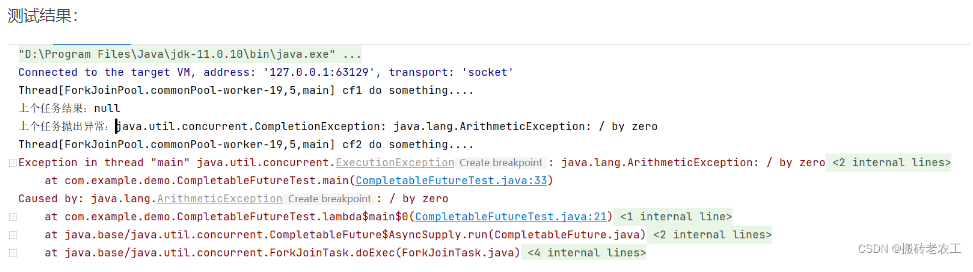


### whenComplete和whenCompleteAsync

whenComplete是当某个任务执行完成后执行的回调方法，会将执行结果或者执行期间抛出的异常传递给回调方法，如果是正常执行则异常为null，回调方法对应的CompletableFuture的result和该任务一致，如果该任务正常执行，则get方法返回执行结果，如果是执行异常，则get方法抛出异常。

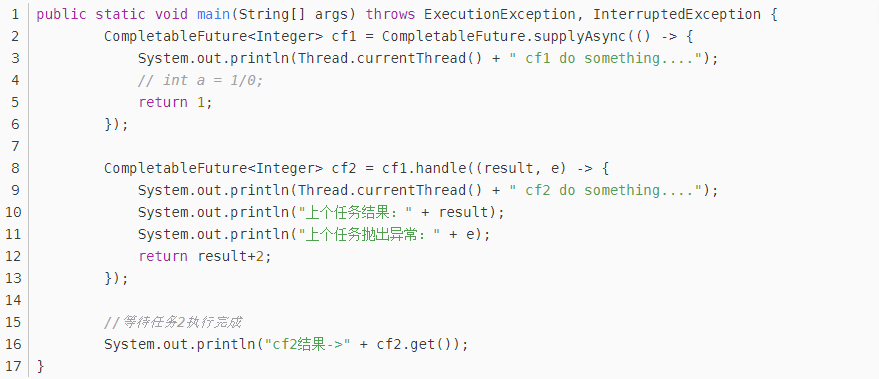
whenCompleteAsync和whenComplete区别也是whenCompleteAsync可能会另起一个线程执行任务，并且thenRunAsync可以自定义线程池，默认的使用ForkJoinPool.commonPool()线程池。

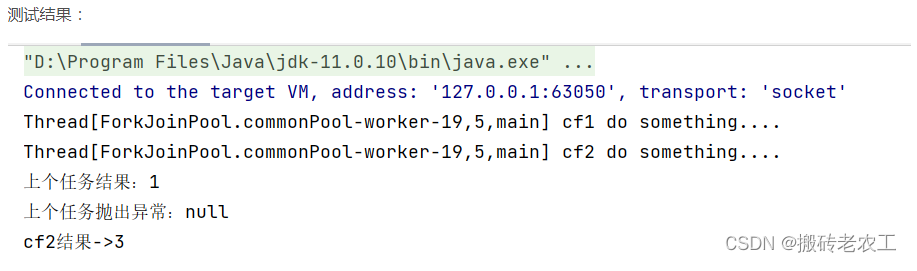




### handle和handleAsync

跟whenComplete基本一致，区别在于handle的回调方法有返回值。





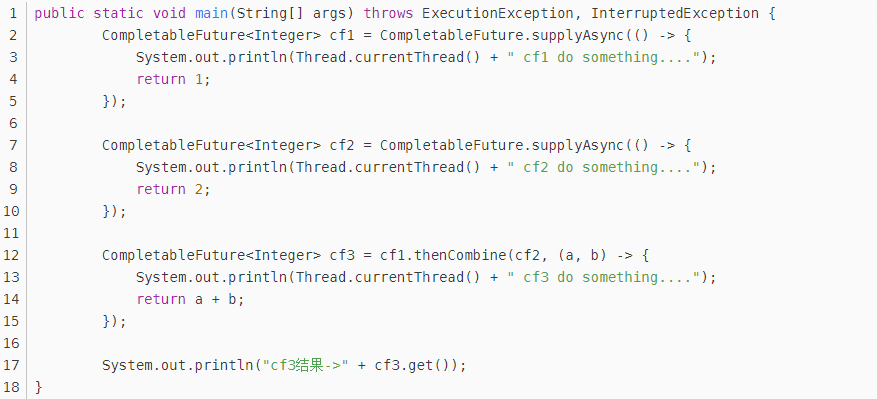
## 多任务组合处理

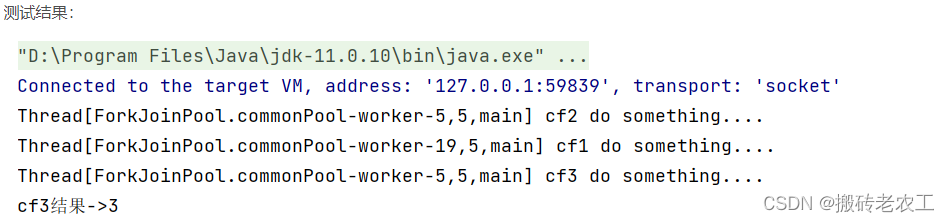
### thenCombine、thenAcceptBoth 和runAfterBoth

这三个方法都是将两个CompletableFuture组合起来处理，只有两个任务都正常完成时，才进行下阶段任务。

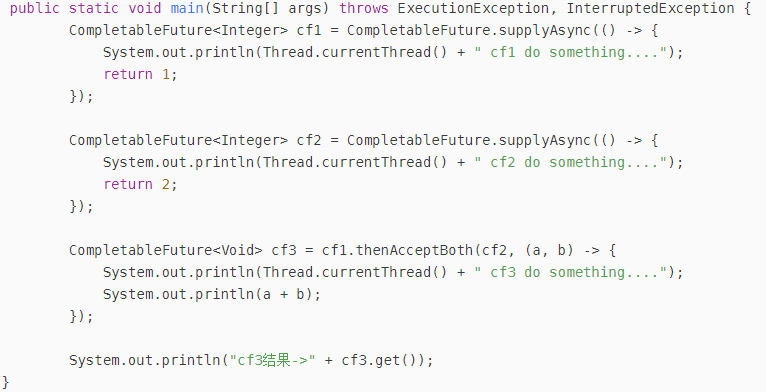
区别：thenCombine会将两个任务的执行结果作为所提供函数的参数，且该方法有返回值；thenAcceptBoth同样将两个任务的执行结果作为方法入参，但是无返回值；runAfterBoth没有入参，也没有返回值。注意两个任务中只要有一个执行异常，则将该异常信息作为指定任务的执行结果。

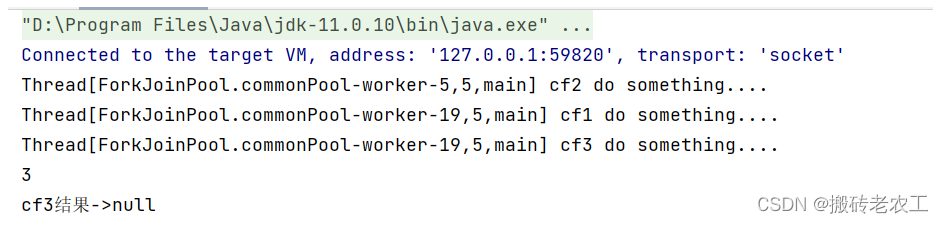
thenCombine



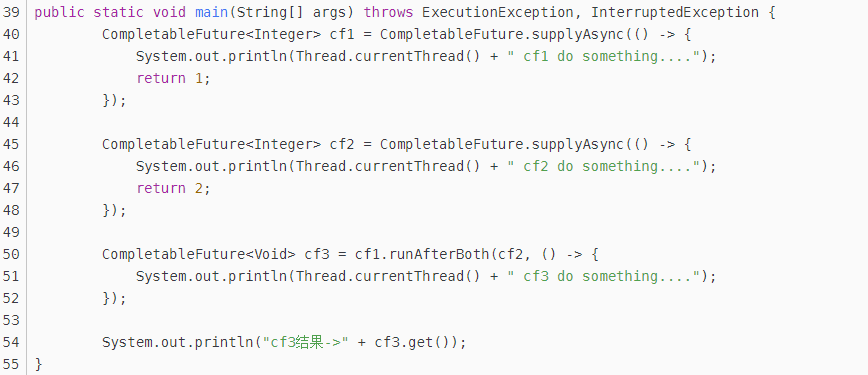


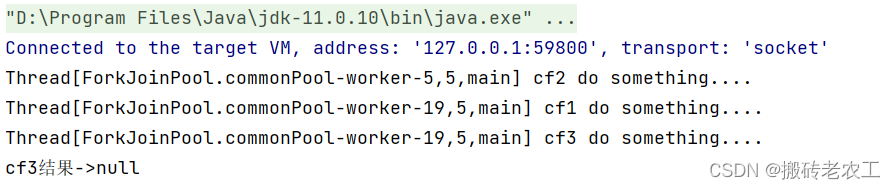
thenAcceptBoth





runAfterBoth

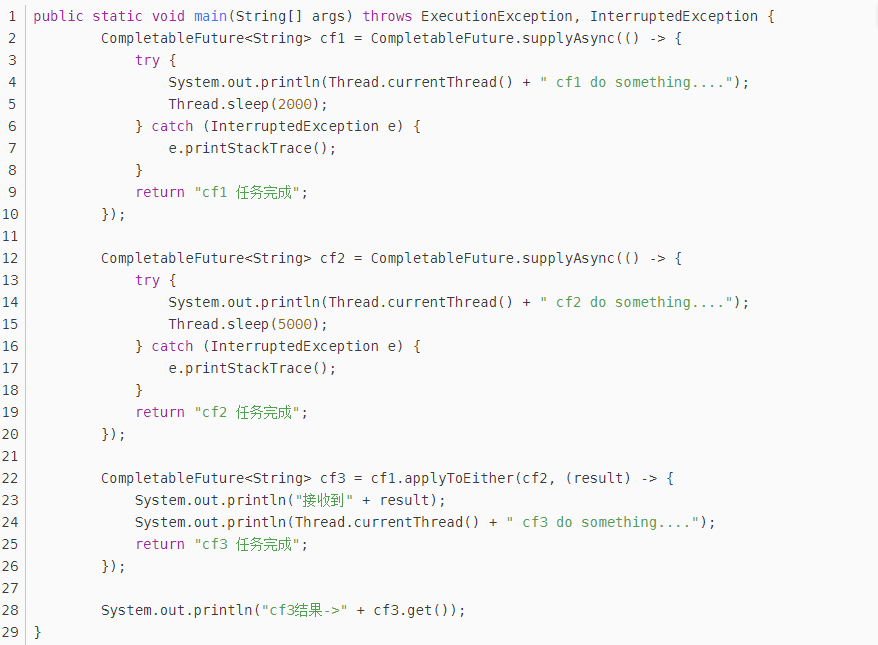




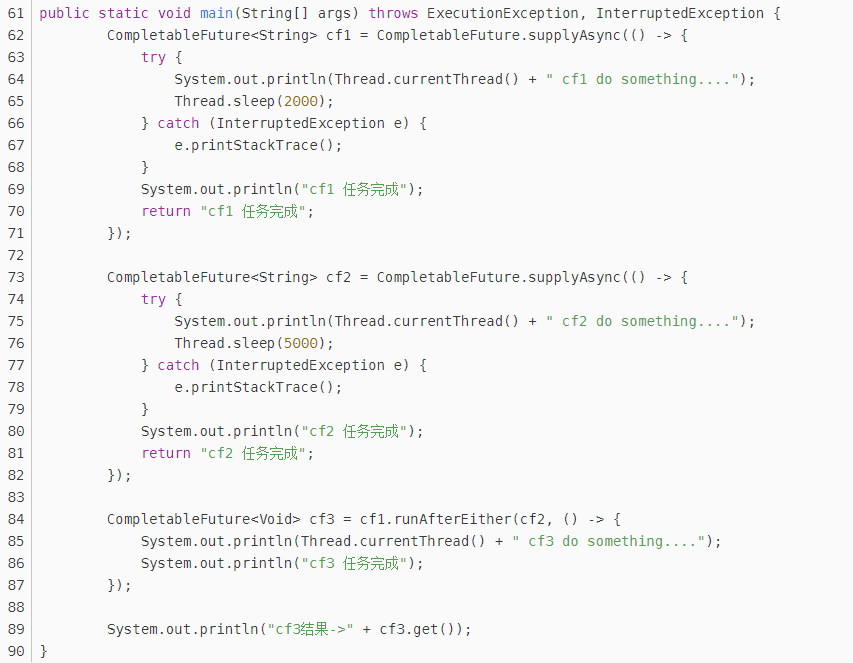
### applyToEither、acceptEither和runAfterEither

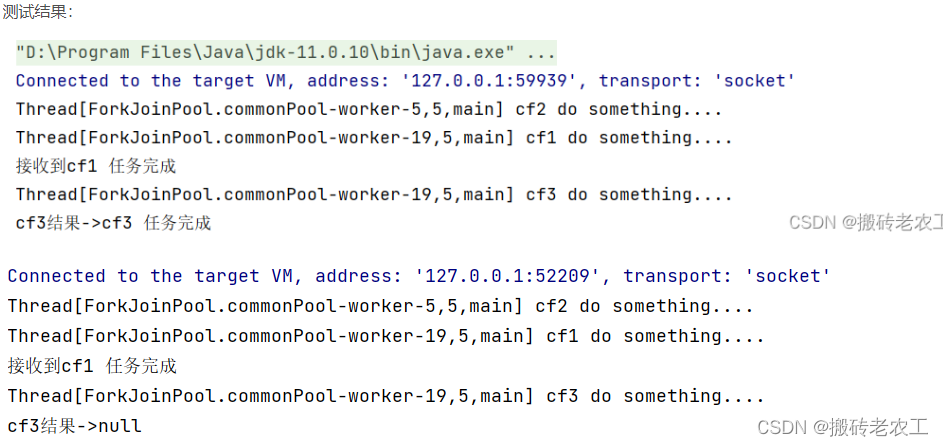
这三个方法和上面一样也是将两个CompletableFuture组合起来处理，当有一个任务正常完成时，就会进行下阶段任务。

区别：applyToEither会将已经完成任务的执行结果作为所提供函数的参数，且该方法有返回值；acceptEither同样将已经完成任务的执行结果作为方法入参，但是无返回值；runAfterEither没有入参，也没有返回值。









上面可以看出cf1任务完成需要2秒，cf2任务完成需要5秒，使用applyToEither组合两个任务时，只要有其中一个任务完成时，就会执行cf3任务，显然cf1任务先完成了并且将自己任务的结果传值给了cf3任务，cf3任务中打印了接收到cf1任务完成，接着完成自己的任务，并返回cf3任务完成；acceptEither和runAfterEither类似，acceptEither会将cf1任务的结果作为cf3任务的入参，但cf3任务完成时并无返回值；runAfterEither不会将cf1任务的结果作为cf3任务的入参，它是没有任务入参，执行完自己的任务后也并无返回值。

### allOf / anyOf

allOf：CompletableFuture是多个任务都执行完成后才会执行，只有有一个任务执行异常，则返回的CompletableFuture执行get方法时会抛出异常，如果都是正常执行，则get返回null。

anyOf ：CompletableFuture是多个任务只要有一个任务执行完成，则返回的CompletableFuture执行get方法时会抛出异常，如果都是正常执行，则get返回执行完成任务的结果。

