

1、从 0 开始深入理解并发、线程与等待通知机制

为什么我们要学习并发编程？

最直白的原因，因为面试需要，我们来看看阿里和美团对 Java 岗位的 JD：

Alibaba 招聘

首页

社会招聘

校园招聘

工作城市

个人中心

本地生活-高级JAVA开发专家-履约执行-杭州

更新于 2023-01-30 | 技术类-开发 | 杭州

☐ 申请此职位表

基础信息

所属部门:	学历:	工作年限:
本地生活	本科	5 年

职位描述

1、负责近场零售的C端、B端、D端、业财结算等复杂业务场景下相关系统的架构设计和核心模块代码编写。

2、深入理解业务需求，主动分析和发现业务痛点提出技术建议，推动落地解决，与新业务一起成长；

3、结合架构和Java技术发展趋势，进行技术预研和技术攻关，突破系统和项目中的技术难点。

职位要求

1、有物流、履约、供应链等相关业务开发和领域驱动经验者优先；

2、具有扎实的Java功底，对JVM的原理有一定的了解，具有较好的Java IO、多线程、网络等方面的编程能力；

3、3年及以上JAVA开发经验，使用过Spring、MyBatis、Struts、Tomcat等常用Java开源框架，对其运行原理有较好的理解；

4、熟悉分布式系统的设计和应用，熟悉分布式、缓存、消息等机制；能合理应用分布式常用技术解决架构问题；

5、掌握多线程编码及性能调优，有丰富的并发、高性能系统、高可用设计和开发经验；

6、精通数据库设计（Mysql优先），优秀的SQL编写及调优能力，熟悉常见NoSQL存储，如hbase、memcached、redis、mongodb等；

7、喜欢钻研及尝试新的技术，追求编写优雅的代码，具有良好的技术敏锐度，能从技术趋势和思路上能影响技术团队；

8、具有较好的沟通能力、极强的学习能力、强烈的责任心和团队合作精神

Java技术专家 社招-正式

更新时间: 2023/01/05 | 工作地点: 上海市 | 事业群: 到店事业群 | 工作经验: 5年

2.与基础架构、前端、算法团队深度协作,在视频体验、安全、成本等方面持续优化;

3.负责技术难点攻关,辅导初级工程师工作。

岗位基本要求

1.有5年及以上相关工作经验,CS基础扎实,有技术热情,愿意持续学习;

2.熟练掌握Java及面向对象程序设计,熟悉网络编程、多线程编程、后台开发相关技术;

3.熟悉数据库原理、查询优化,熟悉RPC、分布式缓存、消息队列等构建大型互联网系统常见组件和设施的使用及原理,能够选型和系统设计;

4.具备业务建模与系统架构能力,能独立完成系统架构设计并且并对落地;

5.善于交流,有良好的团队合作精神和协调沟通能力,有与产品、前端、移动端等多方密切配合的经验和意识;

从上面两大互联网公司的招聘需求可以看到,大厂的Java岗的并发编程能力属于标配。

而在非大厂的公司,并发编程能力也是面试的极大加分项,而工作时善用并发编程则可以极大提升程序员在公司的技术话语权。

为什么开发中需要并发编程?

从阿里的岗位JD其实就能看出来,并发编程和性能优化是密切相关的,使用并发编程可以做到:

(1)加快响应用户的时间

比如我们经常用的迅雷下载,都喜欢多开几个线程去下载,谁都不愿意用一个线程去下载,为什么呢?答案很简单,就是多个线程下载快啊。

我们在做程序开发的时候更应该如此,特别是我们做互联网项目,网页的响应时间若提升1s,如果流量大的话,就能增加不少转换量。做过高性能web前端调优的都知道,要将静态资源地址用两三个子域名去加载,为什么?因为每多一个子域名,浏览器在加载你的页面的时候就会多开几个线程去加载你的页面资源,提升网站的响应速度。

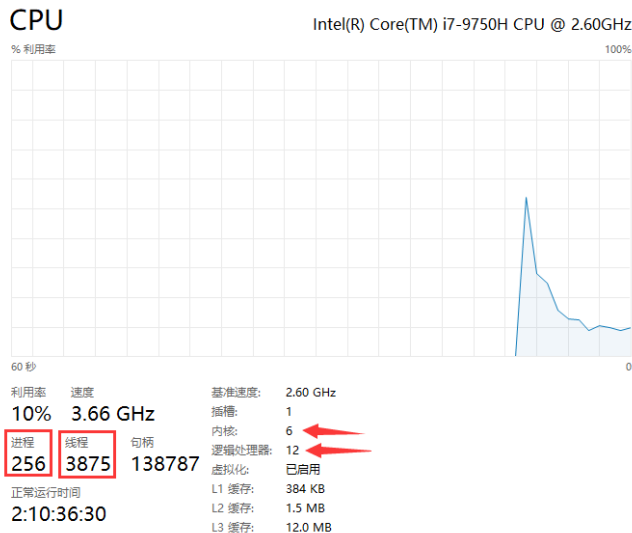
(2)使你的代码模块化,异步化,简单化

例如我们实现电商系统,下订单和给用户发送短信、邮件就可以进行拆分,将给用户发送短信、邮件这两个步骤独立为单独的模块,并交给其他线程去执行。这样既增加了异步的操作,提升了系统性能,又使程序模块化,清晰化和简单化。

多线程应用开发的好处还有很多,大家在日后的代码编写过程中可以慢慢体会它的魅力。

(3)充分利用CPU的资源

目前市面上没有CPU的内核不是多核的,比如这台机器



多核下如果还是使用单线程的技术做思路明显就 out 了,无法充分利用 CPU 的多核特点。如果设计一个多线程的程序的话,那它就可以同时多个 CPU 的多个核的多个线程上跑,可以充分地利用 CPU,减少 CPU 的空闲时间,发挥它的运算能力,提高并发量。

就像我们平时坐地铁一样,很多人坐长线地铁的时候都在认真看书,而不是为了坐地铁而坐地铁,到家了再去看书,这样你的时间就相当于有了两倍。这就是为什么有些人时间很充裕,而有些人老是说没时间的一个原因,工作也是这样,有的时候可以并发地去做几件事情,充分利用我们的时间,CPU 也是一样,也要充分利用。

当然有同学会有疑问,单核 CPU 呢?单核 CPU 一样可以利用到并发编程的好处吗?当然可以,用我们平时常用的 QQ 之类的聊天程序来举例,当我们用 QQ 聊天时,其实程序要做好几件事,比如:接受我们的键盘输入,把输入的信息通过网络发给对方,接受对方通过网络发来的信息,把对方的信息显示在屏幕上,很多的时候,这些事情是可以同时发生的。如果程序不能利用并发编程同时处理,我们和对方的通话就只能一问一答的方式进行了。

我们怎么学并发编程?

课程章节安排如下:

- 1、从 0 开始深入理解并发、线程与等待通知机制
- 2、导致 JVM 内存泄露的 ThreadLocal 详解
- 3、并发编程之 CAS&Atomic 原子操作详解
- 4、一节课学透面试必问并发安全问题
- 5、JUC 并发工具类在大厂的应用场景详解
- 6、深入理解 AQS 之 ReentrantLock 源码分析
- 7、ReentrantReadWriteLock&StampLock 详解
- 8、并发容器 (Map、List、Set) 实战及其原理
- 9、阻塞队列 BlockingQueue 实战及其原理分析

-
- 10、线程池 `ThreadPoolExecutor` 实战及其原理分析
 - 11、线程池 `ForkJoinPool` 工作原理分析
 - 12、深入理解并发可见性、有序性、原子性与 JMM 内存模型
 - 13、CPU 缓存架构详解&高性能内存队列 `Disruptor` 实战
 - 14、常用并发设计模式精讲

可以看到并发编程的课时其实是相当多的，反过来也说明并发编程在 Java 程序员的技能栈中重要地位。

对于没有或者很少接触并发编程的同学，建议主要掌握并发里的基础概念、基础用法和并发工具类、并发容器的用法，章节主要对应第 1~5 章、第 8、9、10、11。对于已经有较多并发编程经验建议全部学习，`synchronized` 底层原理则以知识小节的视频方式单独提供。

注意：以上的章节和授课顺序是根据并发编程本身的知识结构和人类学习的认知机制设计安排的，和具体授课时的课程数、课程标题可能存在一定的不匹配情况，因此出现一节课讲述多个章节和一个章节跨越多节课属于正常现象。

基础概念

在正式学习 Java 的并发编程之前，还有几个并发编程的基础概念我们需要熟悉和学习。

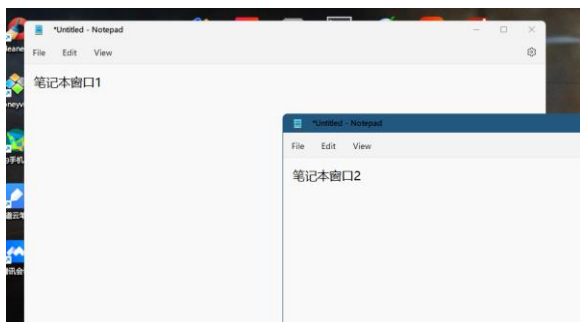
进程和线程

进程

我们常听说的是应用程序，也就是 `app`，由指令和数据组成。但是当我们不运行一个具体的 `app` 时，这些应用程序就是放在磁盘（也包括 U 盘、远程网络存储等等）上的一些二进制的代码。一旦我们运行这些应用程序，指令要运行，数据要读写，就必须将指令加载至 CPU，数据加载至内存。在指令运行过程中还需要用到磁盘、网络等设备，从这种角度来说，进程就是用来加载指令、管理内存、管理 IO 的。

当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程。

进程就可以视为程序的一个实例。大部分程序可以同时运行多个实例进程（例如记事本、画图、浏览器 等），也有的程序只能启动一个实例进程（例如网易云音乐、360 安全卫士等）。显然,程序是死的、静态的,进程是活的、动态的。进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程,它们就是处于运行状态下的操作系统本身,用户进程就是所有由你启动的进程。



站在操作系统的角度,进程是程序运行资源分配(以内存为主)的最小单位。

线程

一个机器中肯定会运行很多的程序,CPU 又是有限的,怎么让有限的 CPU 运行这么多程序呢?就需要一种机制在程序之间进行协调,也就所谓 CPU 调度。线程则是 CPU 调度的最小单位。

线程必须依赖于进程而存在,线程是进程中的一个实体,是 CPU 调度和分派的基本单位,它是比进程更小的、能独立运行的基本单位。线程自己基本上不拥有系统资源,,只拥有在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。一个进程可以拥有多个线程,一个线程必须有一个父进程。线程,有时也被称为轻量级进程(Lightweight Process, LWP),早期 Linux 的线程实现几乎就是复用的进程,后来才独立出自己的 API。

Java 线程的无处不在

Java 中不管任何程序都必须启动一个 main 函数的主线程;Java Web 开发里面的定时任务、定时器、JSP 和 Servlet、异步消息处理机制,远程访问接口 RM 等,任何一个监听事件,onclick 的触发事件等都离不开线程和并发的知识。

大厂面试题: 进程间的通信

同一台计算机的进程通信称为 IPC (Inter-process communication),不同计算机之间的进程通信被称为 R(mote)PC, 需要通过网络,并遵守共同的协议,比如大家熟悉的 Dubbo 就是一个 RPC 框架,而 Http 协议也经常用在 RPC 上,比如 SpringCloud 微服务。

大厂常见的面试题就是,进程间通信有几种方式?

1. 管道,分为匿名管道(pipe)及命名管道(named pipe): 匿名管道可用于具有亲缘关系的父子进程间的通信,命名管道除了具有管道所具有的功能外,它还允许无亲缘关系进程间的通信。

2. 信号(signal): 信号是在软件层次上对中断机制的一种模拟,它是比较复杂的通信方式,用于通知进程有某事件发生,一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一致的。

3. 消息队列(message queue): 消息队列是消息的链接表,它克服了上两种通信方式中信号量有限的缺点,具有写权限得进程可以按照一定得规则向消息队列中添加新信息;对消息队列有读权限得进程则可以从消息队列中读取信息。

4. 共享内存 (shared memory)：可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。

5. 信号量 (semaphore)：主要作为进程之间及同一种进程的不同线程之间得同步和互斥手段。

6. 套接字 (socket)：这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。同一机器中的进程还可以使用 Unix domain socket (比如同一机器中 MySQL 中的控制台 mysql shell 和 MySQL 服务程序的连接)，这种方式不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，比纯粹基于网络的进程间通信肯定效率更高。

CPU 核心数和线程数的关系

前面说过，目前主流 CPU 都是多核的，线程是 CPU 调度的最小单位。同一时刻，一个 CPU 核心只能运行一个线程，也就是 CPU 内核和同时运行的线程数是 1:1 的关系，也就是说 8 核 CPU 同时可以执行 8 个线程的代码。但 Intel 引入超线程技术后，产生了逻辑处理器的概念，使核心数与线程数形成 1:2 的关系。在我们前面的 Windows 任务管理器贴图就能看出来，内核数是 6 而逻辑处理器数是 12。

在 Java 中提供了 `Runtime.getRuntime().availableProcessors()`，可以让我们获取当前的 CPU 核心数，注意这个核心数指的是逻辑处理器数。

获得当前的 CPU 核心数在并发编程中很重要，并发编程下的性能优化往往和 CPU 核心数密切相关。

上下文切换 (Context switch)

既然操作系统要在多个进程 (线程) 之间进行调度，而每个线程在使用 CPU 时总是要使用 CPU 中的资源，比如 CPU 寄存器和程序计数器。这就意味着，操作系统要保证线程在调度前后的正常执行，所以，操作系统中就有上下文切换的概念，它是指 CPU (中央处理单元) 从一个进程或线程到另一个进程或线程的切换。

上下文是 CPU 寄存器和程序计数器在任何时间点的内容。

寄存器是 CPU 内部的一小部分非常快的内存 (相对于 CPU 内部的缓存和 CPU 外部较慢的 RAM 主内存)，它通过提供对常用值的快速访问来加快计算机程序的执行。

程序计数器是一种专门的寄存器，它指示 CPU 在其指令序列中的位置，并保存着正在执行的指令的地址或下一条要执行的指令的地址，这取决于具体的系统。

上下文切换可以更详细地描述为内核 (即操作系统的核心) 对 CPU 上的进程 (包括线程) 执行以下活动：

1. 暂停一个进程的处理，并将该进程的 CPU 状态 (即上下文) 存储在内存中的某个地方
2. 从内存中获取下一个进程的上下文，并在 CPU 的寄存器中恢复它

3. 返回到程序计数器指示的位置(即返回到进程被中断的代码行)以恢复进程。

从数据来说,以程序员的角度来看,是方法调用过程中的各种局部的变量与资源;以线程的角度来看,是方法的调用栈中存储的各类信息。

引发上下文切换的原因一般包括:线程、进程切换、系统调用等等。上下文切换通常是计算密集型的,因为涉及一系列数据在各种寄存器、缓存中的来回拷贝。就 CPU 时间而言,一次上下文切换大概需要 5000~20000 个时钟周期,相对一个简单指令几个乃至十几个左右的执行时钟周期,可以看出这个成本的巨大。

并行和并发

我们举个例子,如果有条高速公路 A 上面并排有 8 条车道,那么最大的并行车辆就是 8 辆此条高速公路 A 同时并排行走的车辆小于等于 8 辆的时候,车辆就可以并行运行。CPU 也是这个原理,一个 CPU 相当于一个高速公路 A,核心数或者线程数就相当于并排可以通行的车道;而多个 CPU 就相当于并排有多条高速公路,而每个高速公路并排有多个车道。

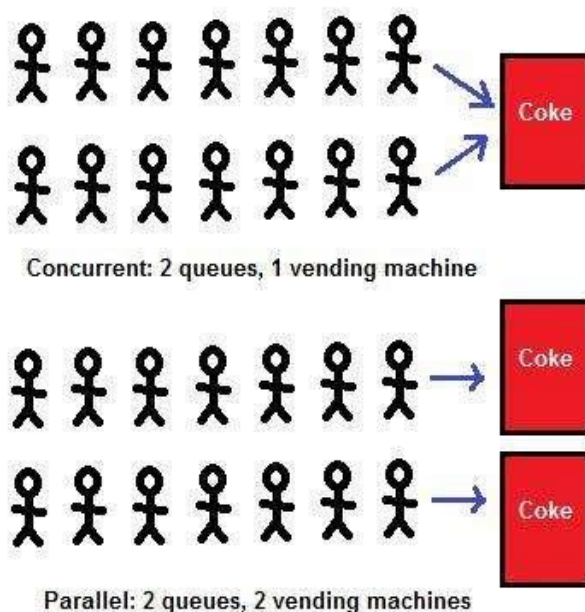
当谈论**并发**的时候一定要加个单位时间,也就是说单位时间内并发量是多少?离开了单位时间其实是没有意义的。

综合来说:

并发 **Concurrent**:指应用能够交替执行不同的任务,比如单 CPU 核心下执行多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已。

并行 **Parallel**:指应用能够同时执行不同的任务,例:吃饭的时候可以边吃饭边打电话,这两件事情可以同时执行

两者区别:一个是交替执行,一个是同时执行,如下图所示。



认识 Java 里的线程

Java 程序天生就是多线程的

一个 Java 程序从 `main()` 方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上 Java 程序天生就是多线程程序，因为执行 `main()` 方法的是一个名称为 `main` 的线程。

而一个 Java 程序的运行就算是没有用户自己开启的线程，实际也有有很多 JVM 自行启动的线程，一般来说有：

[6] Monitor Ctrl-Break // 监控 Ctrl-Break 中断信号的

[5] Attach Listener // 内存 dump，线程 dump，类信息统计，获取系统属性等

[4] Signal Dispatcher // 分发处理发送给 JVM 信号的线程

[3] Finalizer // 调用对象 `finalize` 方法的线程

[2] Reference Handler // 清除 Reference 的线程

[1] main // main 线程，用户程序入口

尽管这些线程根据不同的 JDK 版本会有差异，但是依然证明了 Java 程序天生就是多线程的。

线程的启动与中止

刚刚看到的线程都是 JVM 启动的系统线程，我们学习并发编程希望的自己能操控线程，所以我们先来看看如何启动线程。

启动

启动线程的方式有：

1、X extends Thread;，然后 X.start

```
// 创建线程对象
Thread t = new Thread() {
    public void run() {
        // 要执行的任务
    }
};
// 启动线程
t.start();
```

2、X implements Runnable; 然后交给 Thread 运行


```

Runnable runnable = new Runnable() {
    public void run(){
        // 要执行的任务
    }
};
// 创建线程对象
Thread t = new Thread( runnable );
// 启动线程
t.start();

```

参见代码：cn.tulingxueyuan.base.abc.NewThread

Thread 和 Runnable 的区别

Thread 才是 Java 里对线程的唯一抽象，Runnable 只是对任务（业务逻辑）的抽象。Thread 可以接受任意一个 Runnable 的实例并执行。

Callable、Future 和 FutureTask

Runnable 是一个接口，在它里面只声明了一个 run() 方法，由于 run() 方法返回值为 void 类型，所以在执行完任务之后无法返回任何结果。

Callable 位于 java.util.concurrent 包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做 call()，这是一个泛型接口，call() 函数返回的类型就是传递进来的 V 类型。

Future 就是对于具体的 Runnable 或者 Callable 任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过 get 方法获取执行结果，该方法会阻塞直到任务返回结果。

```

✓ I Future
  (m) cancel(boolean): boolean
  (m) isCancelled(): boolean
  (m) isDone(): boolean
  (m) get(): V
  (m) get(long, TimeUnit): V

```

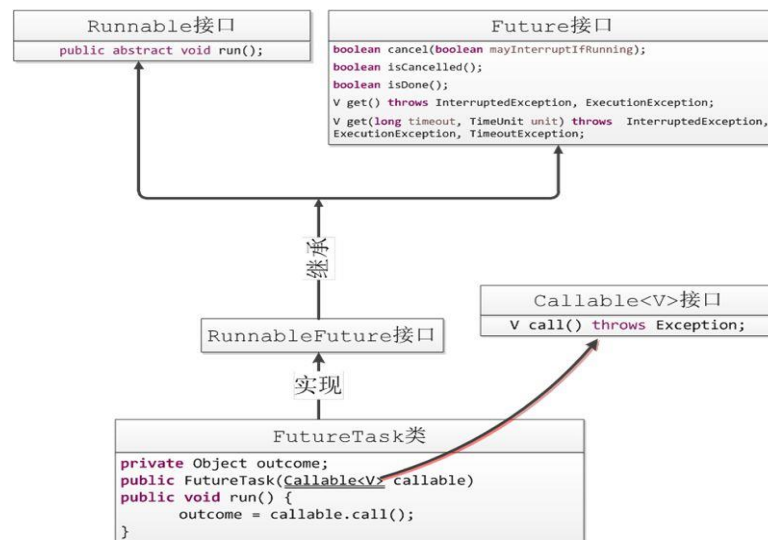
因为 Future 只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的 FutureTask。

```

public class FutureTask<V> implements RunnableFuture<V> {
    /*
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     */
    void run();
}

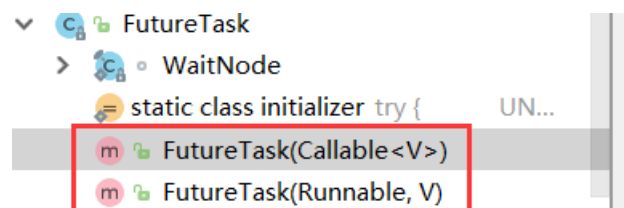
```

FutureTask 类实现了 RunnableFuture 接口，RunnableFuture 继承了 Runnable 接口和 Future 接口，而 FutureTask 实现了 RunnableFuture 接口。所以它既可以作为 Runnable 被线程执行，又可以作为 Future 得到 Callable 的返回值。



因此我们通过一个线程运行 Callable，但是 Thread 不支持构造方法中传递 Callable 的实例，所以我们需要通过 FutureTask 把一个 Callable 包装成 Runnable，然后再通过这个 FutureTask 拿到 Callable 运行后的返回值。

要 new 一个 FutureTask 的实例，有两种方法



面试题：新启线程有几种方式？

这个问题的答案其实众说纷纭，有 2 种，3 种，4 种等等答案，建议比较好的回答是：

按照 Java 源码中 Thread 上的注释：

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

官方说法是在 Java 中有两种方式创建一个线程用以执行，一种是派生自 Thread 类，另一种是实现 Runnable 接口。

当然本质上 Java 中实现线程只有一种方式，都是通过 new Thread() 创建线程对象，调用 Thread#start 启动线程。

至于基于 callable 接口的方式，因为最终是要把实现了 callable 接口的对象通过 FutureTask 包装成 Runnable，再交给 Thread 去执行，所以这个其实可以和实现 Runnable 接口看成同一类。

而线程池的方式，本质上是池化技术，是资源的复用，和新启线程没什么关系。

所以，比较赞同官方的说法，有两种方式创建一个线程用以执行。

中止

线程自然终止

要么是 `run` 执行完成了，要么是抛出了一个未处理的异常导致线程提前结束。

stop

暂停、恢复和停止操作对应在线程 `Thread` 的 API 就是 `suspend()`、`resume()` 和 `stop()`。但是这些 API 是过期的，也就是不建议使用的。不建议使用的原因主要有：以 `suspend()` 方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。同样，`stop()` 方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。正因为 `suspend()`、`resume()` 和 `stop()` 方法带来的副作用，这些方法才被标注为不建议使用的过期方法。

中断

安全的中止则是其他线程通过调用某个线程 `A` 的 `interrupt()` 方法对其进行中断操作，中断好比其他线程对该线程打了个招呼，“`A`，你要中断了”，不代表线程 `A` 会立即停止自己的工作，同样的 `A` 线程完全可以不理睬这种中断请求。线程通过检查自身的中断标志位是否被置为 `true` 来进行响应，

线程通过方法 `isInterrupted()` 来进行判断是否被中断，也可以调用静态方法 `Thread.interrupted()` 来进行判断当前线程是否被中断，不过 `Thread.interrupted()` 会同时将中断标识位改写为 `false`。

如果一个线程处于了阻塞状态（如线程调用了 `thread.sleep`、`thread.join`、`thread.wait` 等），则在线程在检查中断标示时如果发现中断标示为 `true`，则会在这些阻塞方法调用处抛出 `InterruptedException` 异常，并且在抛出异常后会立即将线程的中断标示位清除，即重新设置为 `false`。

不建议自定义一个取消标志位来中止线程的运行。因为 `run` 方法里有阻塞调用时会无法很快检测到取消标志，线程必须从阻塞调用返回后，才会检查这个取消标志。这种情况下，使用中断会更好，因为，

一、一般的阻塞方法，如 `sleep` 等本身就支持中断的检查，

二、检查中断位的状态和检查取消标志位没什么区别，用中断位的状态还可以避免声明取消标志位，减少资源的消耗。

注意：处于死锁状态的线程无法被中断

深入理解 run() 和 start()

Thread 类是 Java 里对线程概念的抽象,可以这样理解:我们通过 new Thread() 其实只是 new 出一个 Thread 的实例,还没有操作系统中真正的线程挂起钩来。只有执行了 start()方法后,才实现了真正意义上的启动线程。

从 Thread 的源码可以看到,Thread 的 start 方法中调用了 start0()方法,而 start0()是个 native 方法,这就说明 Thread#start 一定和操作系统是密切相关的。

start()方法让一个线程进入就绪队列等待分配 cpu,分到 cpu 后才调用实现的 run()方法, start()方法不能重复调用,如果重复调用会抛出异常(注意,此处可能有面试题:多次调用一个线程的 start 方法会怎么样?)。

而 run 方法是业务逻辑实现的地方,本质上和任意一个类的任意一个成员方法并没有任何区别,可以重复执行,也可以被单独调用。

深入学习 Java 的线程

线程的状态/生命周期

Java 中线程的状态分为 6 种:

1. 初始(NEW): 新创建了一个线程对象,但还没有调用 start()方法。
2. 运行(RUNNABLE): Java 线程中将就绪(ready)和运行中(running)两种状态笼统的称为“运行”。

线程对象创建后,其他线程(比如 main 线程)调用了该对象的 start()方法。该状态的线程位于可运行线程池中,等待被线程调度选中,获取 CPU 的使用权,此时处于就绪状态(ready)。就绪状态的线程在获得 CPU 时间片后变为运行中状态(running)。

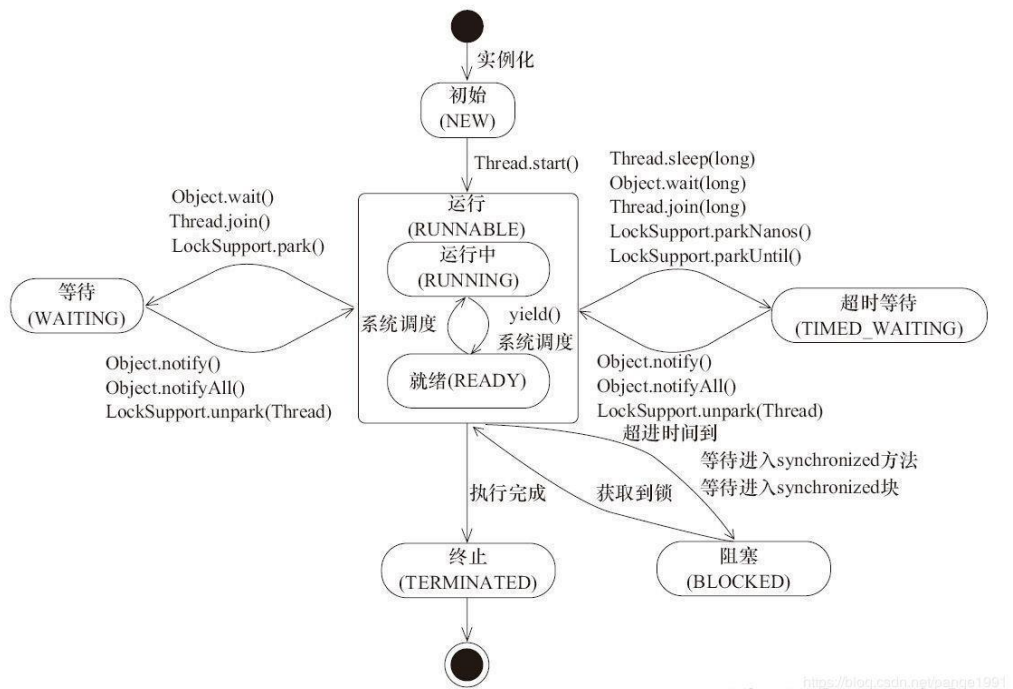
3. 阻塞(BLOCKED): 表示线程阻塞于锁。
4. 等待(WAITING): 进入该状态的线程需要等待其他线程做出一些特定动作(通知或中断)。
5. 超时等待(TIMED_WAITING): 该状态不同于 WAITING,它可以在指定的时间后自行返回。
6. 终止(TERMINATED): 表示该线程已经执行完毕。

```
main" #1 prio=5 os_prio=0 cpu=1031.25ms elapsed=17861.74s tid=0x000000bb0c976000 nid=0x1c94 in Object.wait() [0x000000bb0c15f000]
java.lang.Thread.State: TIMED_WAITING (on object monitor)
  at java.lang.Object.wait(java.base@11.0.3/Native Method)
  - waiting on <no object reference available>
  at com.intellij.execution.rmi.RemoteServer.start(RemoteServer.java:94)
  - waiting to re-lock in wait() <0x00000000d04679d8> (a java.lang.Object)
  at org.jetbrains.idea.maven.server.RemoteMavenServer36.main(RemoteMavenServer36.java:23)

Reference Handler" #2 daemon prio=10 os_prio=2 cpu=0.00ms elapsed=17861.71s tid=0x000000bb24506000 nid=0x34a0 waiting on condition [0x000000bb2517e000]
java.lang.Thread.State: RUNNABLE
  at java.lang.ref.Reference.waitForReferencePendingList(java.base@11.0.3/Native Method)
  at java.lang.ref.Reference.processPendingReferences(java.base@11.0.3/Reference.java:241)
  at java.lang.ref.Reference$ReferenceHandler.run(java.base@11.0.3/Reference.java:213)

Finalizer" #3 daemon prio=8 os_prio=1 cpu=0.00ms elapsed=17861.71s tid=0x000000bb24507000 nid=0x377c in Object.wait() [0x000000bb2527e000]
java.lang.Thread.State: WAITING (on object monitor)
  at java.lang.Object.wait(java.base@11.0.3/Native Method)
  - waiting on <no object reference available>
  at java.lang.ref.ReferenceQueue.remove(java.base@11.0.3/ReferenceQueue.java:155)
  - waiting to re-lock in wait() <0x00000000d0448de0> (a java.lang.ref.ReferenceQueue$Lock)
  at java.lang.ref.ReferenceQueue.remove(java.base@11.0.3/ReferenceQueue.java:176)
  at java.lang.ref.Finalizer$FinalizerThread.run(java.base@11.0.3/Finalizer.java:170)
```

状态之间的变迁如下图所示



掌握这些状态可以让我们在进行 Java 程序调优时可以提供很大的帮助。

其他的线程相关方法

yield()方法：使当前线程让出 CPU 占有权，但让出的时间是不可设定的。也不会释放锁资源。同时执行 **yield()** 的线程有可能在进入到就绪状态后会被操作系统再次选中马上又被执行。

比如，**ConcurrentHashMap#initTable** 方法中就使用了这个方法，

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(o: this, SIZECTL, sc, i1: -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    /unchecked/
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
```

这是因为 **ConcurrentHashMap** 中可能被多个线程同时初始化 **table**，但是其实这个时候只允许一个线程进行初始化操作，其他的线程就需要被阻塞或等待，但是初始化操作其实很快，这里 **Doug Lea** 大师为了避免阻塞或者等待这些操作引发的上下文切换等等开销，就让其他不执行初始化操作的线程干脆执行 **yield()** 方法，以让出 CPU 执行权，让执行初始化操作的线程可以更快的执行完成。

wait()/notify()/notifyAll(): 后面会单独讲述

线程的优先级

在 Java 线程中，通过一个整型成员变量 `priority` 来控制优先级，优先级的范围从 1~10，在线程构建的时候可以通过 `setPriority(int)` 方法来修改优先级，默认优先级是 5，优先级高的线程分配时间片的数量要多于优先级低的线程。

设置线程优先级时，针对频繁阻塞（休眠或者 I/O 操作）的线程需要设置较高优先级，而偏重计算（需要较多 CPU 时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。在不同的 JVM 以及操作系统上，线程规划会存在差异，有些操作系统甚至会忽略对线程优先级的设定。

线程的调度

线程调度是指系统为线程分配 CPU 使用权的过程，主要调度方式有两种：

协同式线程调度(Cooperative Threads-Scheduling)

抢占式线程调度(Preemptive Threads-Scheduling)

使用协同式线程调度的多线程系统，线程执行的时间由线程本身来控制，线程把自己的工作执行完之后，要主动通知系统切换到另外一个线程上。使用协同式线程调度的最大好处是实现简单，由于线程要把自己的事情做完后才会通知系统进行线程切换，所以没有线程同步的问题，但是坏处也很明显，如果一个线程出了问题，则程序就会一直阻塞。

使用抢占式线程调度的多线程系统，每个线程执行的时间以及是否切换都由系统决定。在这种情况下，线程的执行时间不可控，所以不会有「一个线程导致整个进程阻塞」的问题出现。

Java 线程调度就是抢占式调度，为什么？后面会分析。

在 Java 中，`Thread.yield()` 可以让出 CPU 执行时间，但是对于获取执行时间，线程本身是没有办法的。对于获取 CPU 执行时间，线程唯一可以使用的手段是设置线程优先级，Java 设置了 10 个级别的程序优先级，当两个线程同时处于 Ready 状态时，优先级越高的线程越容易被系统选择执行。

线程和协程

为什么 Java 线程调度是抢占式调度？这需要我们了解 Java 中线程的实现模式。

我们已经知道线程其实是操作系统层面的实体，Java 中的线程怎么和操作系统层面对应起来呢？

任何语言实现线程主要有三种方式：使用内核线程实现（1:1 实现），使用用户线程实现（1:N 实现），使用用户线程加轻量级进程混合实现（N:M 实现）。

内核线程实现

使用内核线程实现的方式也被称为 1: 1 实现。内核线程（Kernel-Level Thread，KLT）就是直接由操作系统内核（Kernel，下称内核）支持的线程，

这种线程由内核来完成线程切换， 内核通过操纵调度器（Scheduler） 对线程进行调度， 并负责将线程的任务映射到各个处理器上。

由于内核线程的支持，每个线程都成为一个独立的调度单元，即使其中某一个在系统调用中被阻塞了，也不会影响整个进程继续工作，相关的调度工作也不需要额外考虑，已经由操作系统处理了。

局限性：首先，由于是基于内核线程实现的，所以各种线程操作，如创建、析构及同步，都需要进行系统调用。而系统调用的代价相对较高，需要在用户态（User Mode）和内核态（Kernel Mode）中来回切换。其次，每个语言层面的线程都需要有一个内核线程的支持，因此要消耗一定的内核资源（如内核线程的栈空间），因此一个系统支持的线程数量是有限的。

用户线程实现

严格意义上的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知到用户线程的存在及如何实现的。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。如果程序实现得当，这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也能够支持规模更大的线程数量，部分高性能数据库中的多线程就是由用户线程实现的。

用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援，所有的线程操作都需要由用户程序自己去处理。线程的创建、销毁、切换和调度都是用户必须考虑的问题，而且由于操作系统只把处理器资源分配到进程，那诸如“阻塞如何处理”“多处理器系统中如何将线程映射到其他处理器上”这类问题解决起来将会异常困难，甚至有些是不可能实现的。因为使用用户线程实现的程序通常都比较复杂，所以一般的应用程序都不倾向使用用户线程。Java 语言曾经使用过用户线程，最终又放弃了。但是近年来许多新的、以高并发为卖点的编程语言又普遍支持了用户线程，譬如 Golang。

混合实现

线程除了依赖内核线程实现和完全由用户程序自己实现之外，还有一种将内核线程与用户线程一起使用的实现方式，被称为 **N: M** 实现。在这种混合实现下，既存在用户线程，也存在内核线程。

用户线程还是完全建立在用户空间中，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发。

同样又可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过内核线程来完成。在这种混合模式中，用户线程与轻量级进程的数量比是不定的，是 **N: M** 的关系。

Java 线程的实现

J Java 线程在早期的 Classic 虚拟机上（JDK 1.2 以前），是用户线程实现的，但从 JDK 1.3 起，主流商用 Java 虚拟机的线程模型普遍都被替换为基于操作系统原生线程模型来实现，即采用 **1: 1** 的线程模型。

以 HotSpot 为例，它的每一个 Java 线程都是直接映射到一个操作系统原生线程来实现的，而且中间没有额外的间接结构，所以 HotSpot 自己是不会去干涉线程调度的，全权交给底下的操作系统去处理。

所以，这就是我们说 Java 线程调度是抢占式调度的原因。而且 Java 中的线程优先级是通过映射到操作系统的原生线程上实现的，所以线程的调度最终取决于操作系统，操作系统中线程的优先级有时并不能和 Java 中的一一对应，所以 Java 优先级并不是特别靠谱。

协程

出现的原因

随着互联网行业的发展，目前内核线程实现在很多场景已经有点不适宜了。比如，互联网服务架构在处理一次对外部业务请求的响应，往往需要分布在不同机器上的大量服务共同协作来实现，也就是我们常说的微服务，这种服务细分的架构在减少单个服务复杂度、增加复用性的同时，也不可避免地增加了服务的数量，缩短了留给每个服务的响应时间。这要求每一个服务都必须在极短的时间内完成计算，这样组合多个服务的总耗时才不会太长；也要求每一个服务提供者都要能同时处理数量更庞大的请求，这样才不会出现请求由于某个服务被阻塞而出现等待。

Java 目前的并发编程机制就与上述架构趋势产生了一些矛盾，1:1 的内核线程模型是如今 Java 虚拟机线程实现的主流选择，但是这种映射到操作系统上的线程天然的缺陷是切换、调度成本高昂，系统能容纳的线程数量也很有限。以前处理一个请求可以允许花费很长时间在单体应用中，具有这种线程切换的成本也是无伤大雅的，但现在在每个请求本身的执行时间变得很短、数量变得很多的前提下，用户本身的业务线程切换的开销甚至可能会接近用于计算本身的开销，这就会造成严重的浪费。

另外我们常见的 Java Web 服务器，比如 Tomcat 的线程池的容量通常在几十个到两百之间，当把数以百万计的请求往线程池里面灌时，系统即使能处理得过来，但其中的切换损耗也是相当可观的。

这样的话，对 Java 语言来说，用户线程的重新引入成为了解决上述问题一个非常可行的方案。

其次，Go 语言等支持用户线程等新型语言给 Java 带来了巨大的压力，也使得 Java 引入用户线程成为了一个绕不开的话题。

协程简介

为什么用户线程又被称为协程呢？我们知道，内核线程的切换开销是来自于保护和恢复现场的成本，那如果改为采用用户线程，这部分开销就能够省略掉吗？答案还是“不能”。但是，一旦把保护、恢复现场及调度的工作从操作系统交到程序员手上，则可以通过很多手段来缩减这些开销。

由于最初多数的用户线程是被设计成协同式调度（Cooperative Scheduling）的，所以它有了一个别名——“协程”（Coroutine）完整地做调用栈的保护、恢复工作，所以今天也被称为“有栈协程”（Stackfull Coroutine）。

协程的主要优势是轻量，无论是有栈协程还是无栈协程，都要比传统内核线程要轻量得多。如果进行量化的话，那么如果不显式设置，则在 64 位 Linux 上 HotSpot 的线程栈容量默认是 1MB，此外内核数据结构(Kernel Data Structures)还会额外消耗 16KB 内存。与之相对的，一个协程的栈通常在几百个字节到几 KB 之间，所以 Java 虚拟机里线程池容量达到两百就已经不算小了，而很多支持协程的应用中，同时并存的协程数量可数以十万计。

协程当然也有它的局限，需要在应用层面实现的内容（调用栈、调度器这些）特别多，同时因为协程基本上是协同式调度，则协同式调度的缺点自然在协程上也存在。

总的来说，协程机制适用于被阻塞的，且需要大量并发的场景（网络 io），不适合大量计算的场景，因为协程提供规模(更高的吞吐量)，而不是速度(更低的延迟)。

纤程-Java 中的协程

在 JVM 的实现上，以 HotSpot 为例，协程的实现会有些额外的限制，Java 调用栈跟本地调用栈是做在一起的。如果在协程中调用了本地方法，还能否正常切换协程而不影响整个线程？另外，如果协程中遇传统的线程同步措施会怎样？譬如 Kotlin 提供的协程实现，一旦遭遇 synchronize 关键字，那挂起来的仍将是整个线程。

所以 Java 开发组就 Java 中协程的实现也做了很多努力，OpenJDK 在 2018 年创建了 Loom 项目，这是 Java 的官方解决方案，并用了“纤程（Fiber）”这个名字。

Loom 项目背后的意图是重新提供对用户线程的支持，但这些新功能不是为了取代当前基于操作系统的线程实现，而是会有两个并发编程模型在 Java 虚拟机中并存，可以在程序中同时使用。新模型有意地保持了与目前线程模型相似的 API 设计，它们甚至可以拥有一个共同的基类，这样现有的代码就不需要为了使用纤程而进行过多改动，甚至不需要知道背后采用了哪个并发编程模型。

根据 Loom 团队在 2018 年公布的他们对 Jetty 基于纤程改造后的测试结果，同样在 5000QPS 的压力下，以容量为 400 的线程池的传统模式和每个请求配以一个纤程的新并发处理模式进行对比，前者的请求响应延迟在 10000 至 20000 毫秒之间，而后者的延迟普遍在 200 毫秒以下，

目前 Java 中比较出名的协程库是 Quasar[^{'kweɪzɑ:(r)}]（Loom 项目的 Leader 就是 Quasar 的作者 Ron Pressler），Quasar 的实现原理是字节码注入，在字节码层面对当前被调用函数中的所有局部变量进行保存和恢复。这种不依赖 Java 虚拟机的现场保护虽然能够工作，但影响性能。

Quasar 实战

本实战的代码是单独的项目 quasar。

Quasar 的使用其实并不复杂，首先引入 Maven 依赖

```
<dependency>
  <groupId>co.paralleluniverse</groupId>
  <artifactId>quasar-core</artifactId>
  <version>0.7.9</version>
</dependency>
```

在具体的业务场景上，我们模拟调用某个远程的服务，假设远程服务处理耗时需要 1S，使用休眠 1S 来代替。为了比较，用多线程和协程分别调用这个服务 10000 次，来看看两者所需的耗时。

Quasar 的：

```
CountDownLatch count = new CountDownLatch(10000);
StopWatch stopWatch = new StopWatch();
stopWatch.start();
IntStream.range(0,10000).forEach(i-> new Fiber() {
    @Override
    protected String run() throws SuspendExecution, InterruptedException {
        //Quasar中Thread和Fiber都被称为Strand,Fiber不能调用Thread.sleep休眠
        Strand.sleep( millis: 1000 );
        count.countDown();
        return "aa";
    }
}).start();
count.await();
stopWatch.stop();
System.out.println("结束了: " + stopWatch.prettyPrint());
```

线程的：

```
CountDownLatch count = new CountDownLatch(10000);
StopWatch stopWatch = new StopWatch();
stopWatch.start();
ExecutorService executorService = Executors.newCachedThreadPool();
IntStream.range(0,10000).forEach(i-> executorService.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep( timeout: 1);
    } catch (InterruptedException ex) { }
    count.countDown();
}));
count.await();
stopWatch.stop();
System.out.println("结束了: " + stopWatch.prettyPrint());
```

从代码层面来看，两者的代码高度相似，忽略两者的公共部分，代码不同的地方也就 2、3 行。

其中的 Fiber 就是 Quasar 为我们提供的协程相关的类，可以类比为 Java 中的 Thread 类。

其他的 CountDownLatch（闭锁，线程的某种协调工具类）、Executors.newCachedThreadPool（线程池）是并发编程后面的课程将要学习的知识。StopWatch 是 Spring 的一个工具类，一个简单的秒表工具，可以计时指定代码段的运行时间以及汇总这个运行时间。

上面的代码现在看不懂不要紧，随着后面并发编程知识的继续学习，这些代码是很容易理解的，现在只要知道这些代码的业务意义即可：调用远程服务 10000 次，每次耗时 1S，然后统计总耗时。

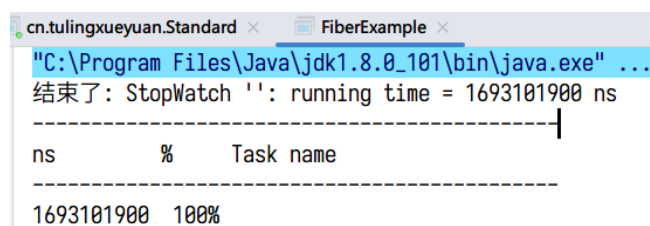
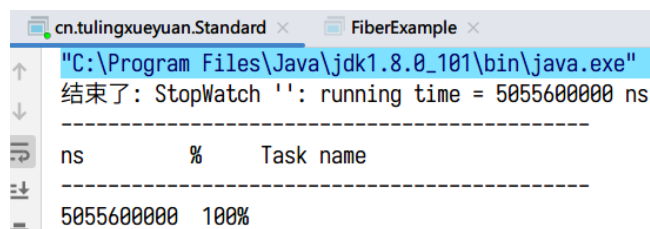
在执行 Quasar 的代码前，还需要配置 VM 参数（Quasar 的实现原理是字节码注入，所以，在运行应用前，需要配置好 quasar-core 的 java agent 地址）

-javaagent:D:\Maven\repository\co\paralleluniverse\quasar-core\0.7.9\quasar-core-0.7.9.jar

自己运行前记得修改为本机的 Maven 仓库路径



看看执行的结果：



可以看到性能的提升还是非常明显的。而且上面多线程编程时，并没有指定线程池的大小，在实际开发中是绝不允许的。一般我们会设置一个固定大小的线程池，因为线程资源是宝贵，线程多了费内存还会带来线程切换的开销。上面的场景在设置 200 个固定大小线程池时（`Executors.newFixedThreadPool(200)`），在本机的测试结果达到了 50 多秒，几乎是数量级的增加。

由这个结果也可以看到协程在需要处理大量 IO 的情况下非常具有优势，基于固定的几个线程调度，可以轻松实现百万级的协程处理，而且内存消耗非常平稳。

更多 Quasar 的使用方法和技巧，请大家自行挖掘和学习。

JDK19 的虚拟线程

2022 年 9 月 22 日，JDK19（非 LTS 版本）正式发布，引入了协程，并称为轻量级虚拟线程。但是这个特性目前还是预览版，还不能引入生成环境。因为环境所限，本课程不提供实际的范例，只讲述基本用法和原理。

要使用的话，需要通过

使用 `javac --release 19 --enable-preview XXX.java` 编译程序，并使用 `java --enable-preview XXX` 运行该程序

在具体使用上和原来的 Thread API 差别不大：`java.lang.Thread.Builder`，可以创建和启动虚拟线程，例如：

```
Thread thread = Thread.ofVirtual().name("duke").unstarted(runnable);
```

```
// Thread.ofPlatform() 则创建传统意义的实际
```

或者

```
Thread.startVirtualThread(Runnable)
```

并通过 `Executors.newVirtualThreadPerTaskExecutor()` 提供了虚拟线程池功能。

在具体实现上，虚拟线程当然是基于用户线程模式实现的，JDK 的调度程序不直接将虚拟线程分配给处理器，而是将虚拟线程分配给实际线程，是一个 M:N 调度，具体的调度程序由已有的 `ForkJoinPool` 提供支持。

但是虚拟线程不是协同调度的，JDK 的虚拟线程调度程序通过将虚拟线程挂载到平台线程上来分配要在平台线程上执行的虚拟线程。在运行一些代码之后，虚拟线程可以从其载体卸载。此时平台线程是空闲的，因此调度程序可以在其上挂载不同的虚拟线程，从而使其再次成为载体。

通常，当虚拟线程阻塞 I/O 或 JDK 中的其他阻塞操作(如 `BlockingQueue.take()`)时，它将卸载。当阻塞操作准备完成时(例如，在套接字上已经接收到字节)，它将虚拟线程提交回调度程序，调度程序将在运营商上挂载虚拟线程以恢复执行。虚拟线程的挂载和卸载频繁且透明，并且不会阻塞任何 OS 线程。

守护线程

Daemon（守护）线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作。这意味着，当一个 Java 虚拟机中不存在非 Daemon 线程的时候，Java 虚拟机将会退出。可以通过调用 `Thread.setDaemon(true)` 将线程设置为 Daemon 线程。我们一般用不上，比如垃圾回收线程就是 Daemon 线程。

Daemon 线程被用作完成支持性工作，但是在 Java 虚拟机退出时 Daemon 线程中的 `finally` 块并不一定会执行。在构建 Daemon 线程时，不能依靠 `finally` 块中的内容来确保执行关闭或清理资源的逻辑。

线程间的通信和协调、协作

很多的时候，孤零零的一个线程工作并没有什么太多用处，更多的时候，我们是很多线程一起工作，而且是这些线程间进行通信，或者配合着完成某项工作，这就离不开线程间的通信和协调、协作。

管道输入输出流

我们已经知道，进程间有好几种通信机制，其中包括了管道，其实 Java 的线程里也有类似的管道机制，用于线程之间的数据传输，而传输的媒介为内存。

设想这么一个应用场景：通过 Java 应用生成文件，然后需要将文件上传到云端，比如：

1、页面点击导出后，后台触发导出任务，然后将 mysql 中的数据根据导出条件查询出来，生成 Excel 文件，然后将文件上传到 oss，最后发步一个下载文件的链接。

2、和银行以及金融机构对接时，从本地某个数据源查询数据后，上报 xml 格式的数据，给到指定的 ftp、或是 oss 的某个目录下也是类似的。

我们一般的做法是，先将文件写入到本地磁盘，然后从文件磁盘读出来上传到云盘，但是通过 Java 中的管道输入输出流一步到位，则可以避免写入磁盘这一步。

Java 中的管道输入/输出流主要包括了如下 4 种具体实现：
PipedOutputStream、PipedInputStream、PipedReader 和 PipedWriter，前两种面向字节，而后两种面向字符。

示例代码可参见 `cn.tulingxueyuan.base.Piped`

join 方法

面试题

现在有 T1、T2、T3 三个线程，你怎样保证 T2 在 T1 执行完后执行，T3 在 T2 执行完后执行？

答：用 `Thread#join` 方法即可，在 T3 中调用 `T2.join`，在 T2 中调用 `T1.join`。

join()

把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行。比如在线程 B 中调用了线程 A 的 `Join()` 方法，直到线程 A 执行完毕后，才会继续执行线程 B 剩下的代码。

synchronized 内置锁

线程开始运行，拥有自己的栈空间，就如同一个脚本一样，按照既定的代码一步一步地执行，直到终止。但是，每个运行中的线程，如果仅仅是孤立地运行，那么没有一点儿价值，或者说价值很少，如果多个线程能够相互配合完成工作，包括数据之间的共享，协同处理事情。这将会带来巨大的价值。

Java 支持多个线程同时访问一个对象或者对象的成员变量，但是多个线程同时访问同一个变量，会导致不可预料的结果。关键字 `synchronized` 可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性，使多个线程访问同一个变量的结果正确，它又称为内置锁机制。

对象锁和类锁：

对象锁是用于对象实例方法，或者一个对象实例上的，类锁是用于类的静态方法或者一个类的 `class` 对象上的。

```
private static synchronized void synClass(){
    System.out.println(Thread.currentThread().ge
        +"synClass going...");
    SleepTools.second( seconds: 1);
    Count count = new Count(Thread.currentThread().getThreadID());
}
```

比如上面的 synClass 方法就使用了类锁。

我们知道，类的对象实例可以有很多个，所以当对同一个变量操作时，用来做锁的对象必须是同一个，否则加锁毫无作用。比如下面的示例代码：

```
public void incCountBlock(){
    synchronized (this){
        count++;
    }
}

//线程
4 usages
private static class Count extends Thread{
    2 usages
    private SynInstance2 simpl0per;
    2 usages
    public Count(SynInstance2 simpl0per) { this.simpl0per = simpl0per; }

    @Override
    public void run() {
        for(int i=0;i<10000;i++){
            simpl0per.incCountBlock();
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    SynInstance2 simpl0per = new SynInstance2();
    SynInstance2 simpl0per2 = new SynInstance2();
    //启动两个线程
    Count count1 = new Count(simpl0per);
    Count count2 = new Count(simpl0per2);
    count1.start();
    count2.start();
    Thread.sleep(1000);
}
```

但是有一点必须注意的是，其实类锁只是一个概念上的东西，并不是真实存在的，类锁其实锁的是每个类的对应的 class 对象，但是每个类只有一个 class 对象，所以每个类只有一个类锁。

同样的，当对同一个变量操作时，类锁和对象（非 class 对象）锁混用也同样毫无用处。

错误的加锁和原因分析

参见代码 TestIntegerSyn，执行结果

```
"C:\Program Files\Java\jdk1.8.0_101"
Thread-0--@1914489481
Thread-0-----[i=2]-@2094738106
Thread-3--@2094738106
Thread-3-----[i=3]-@1124110223
Thread-2--@2094738106
Thread-4--@1124110223
Thread-4-----[i=5]-@361843692
Thread-2-----[i=4]-@1939275867
Thread-1--@361843692
Thread-1-----[i=6]-@1145673791
```

可以看到 i 的取值会出现乱序或者重复取值的现象

原因：虽然我们对 i 进行了加锁，但是

```
private Integer i;

public Worker(Integer i) {
    this.i=i;
}

@Override
public void run() {
    synchronized (i) {
        Thread thread=Thread.currentThread();
        System.out.println(thread.getName()+"--@"+System
i++;
        System.out.println(thread.getName()+"-----"+i+
try {
    Thread.sleep( millis: 3000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

        System.out.println(thread.getName()+"-----"+i+
    }
```

但是当我们反编译这个类的 class 文件后，可以看到 i++ 实际是，

```
= Thread.currentThread();
ntln(thread.getName() + "--@" + System.identityHashCode(this.i));
nteger1 = this.i; Integer localInteger2 = this.i = Integer.valueOf(this.i.intValue() + 1);
ntln(thread.getName() + "-----" + this.i + "--@" + System.identityHashCode(this.i));
```

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

本质上是返回了一个新的 Integer 对象。也就是每个线程实际加锁的是不同的 Integer 对象，所以说到底，还是当对同一个变量操作时，用来做锁的对象必须是同一个，否则加锁毫无作用。

volatile，最轻量的通信/同步机制

volatile 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

```
public class VolatileCase {
    private static boolean ready;
    private static int number;
```

不加 volatile 时，子线程无法感知主线程修改了 ready 的值，从而不会退出循环，而加了 volatile 后，子线程可以感知主线程修改了 ready 的值，迅速退出循环。但是 volatile 不能保证数据在多个线程下同时写时的线程安全，volatile 最适用的场景：一个线程写，多个线程读。

等待/通知机制

线程之间相互配合，完成某项工作，比如：一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作，整个过程开始于一个线程，而最终执行又是另一个线程。前者是生产者，后者就是消费者，这种模式隔离了“做什么”（what）和“怎么做”（How），简单的办法是让消费者线程不断地循环检查变量是否符合预期在 while 循环中设置不满足的条件，如果条件满足则退出 while 循环，从而完成消费者的工作。却存在如下问题：

1) 难以确保及时性。

2) 难以降低开销。如果降低睡眠的时间，比如休眠 1 毫秒，这样消费者能更加迅速地发现条件变化，但是却可能消耗更多的处理器资源，造成了无端的浪费。

等待/通知机制则可以很好的避免，这种机制是指一个线程 A 调用了对象 O 的 wait()方法进入等待状态，而另一个线程 B 调用了对象 O 的 notify()或者 notifyAll()方法，线程 A 收到通知后从对象 O 的 wait()方法返回，进而执行后续操作。上述两个线程通过对象 O 来完成交互，而对象上的 wait()和 notify/notifyAll()的关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

notify():

通知一个在对象上等待的线程,使其从 wait 方法返回,而返回的前提是该线程获取到了对象的锁，没有获得锁的线程重新进入 WAITING 状态。

notifyAll():

通知所有等待在该对象上的线程

wait()

调用该方法的线程进入 WAITING 状态,只有等待另外线程的通知或被中断才会返回.需要注意,调用 wait()方法后,会释放对象的锁

wait(long)

超时等待一段时间,这里的参数时间是毫秒,也就是等待长达 n 毫秒,如果没有通知就超时返回

wait (long,int)

对于超时时间更细粒度的控制,可以达到纳秒

等待和通知的标准范式

等待方遵循如下原则。

1) 获取对象的锁。

2) 如果条件不满足，那么调用对象的 wait()方法，被通知后仍要检查条件。

3) 条件满足则执行对应的逻辑。

```
synchronized(对象) {  
    while(条件不满足) {  
        对象.wait();  
    }  
    对应的处理逻辑  
}
```

通知方遵循如下原则。

- 1) 获得对象的锁。
- 2) 改变条件。
- 3) 通知所有等待在对象上的线程。

```
synchronized(对象) {  
    改变条件  
    对象.notifyAll();  
}
```

在调用 **wait()**、**notify()** 系列方法之前，线程必须要获得该对象的对象级别锁，即只能在同步方法或同步块中调用 **wait()** 方法、**notify()** 系列方法，进入 **wait()** 方法后，当前线程释放锁，在从 **wait()** 返回前，线程与其他线程竞争重新获得锁，执行 **notify()** 系列方法的线程退出调用了 **notifyAll** 的 **synchronized** 代码块的时候后，他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出 **synchronized** 代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

notify 和 notifyAll 应该用谁

尽可能用 **notifyAll()**，谨慎使用 **notify()**，因为 **notify()** 只会唤醒一个线程，我们无法确保被唤醒的这个线程一定就是我们需要唤醒的线程

等待超时模式实现一个连接池

调用场景：调用一个方法时等待一段时间（一般来说是给定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，超时返回默认结果。

假设等待时间段是 **T**，那么可以推断出在当前时间 **now+T** 之后就会超时
等待持续时间：**REMAINING=T**。

• 超时时间：**FUTURE=now+T**。

// 对当前对象加锁

```
public synchronized Object get(long mills) throws InterruptedException {  
    long future = System.currentTimeMillis() + mills;  
    long remaining = mills;  
    // 当超时大于 0 并且 result 返回值不满足要求  
    while ((result == null) && remaining > 0) {
```

```
        wait(remaining);
        remaining = future - System.currentTimeMillis();
    }
    return result;
}
```

具体实现参见：包 `cn.tulingxueyuan.base.pool` 下的代码

客户端获取连接的过程被设定为等待超时的模式，也就是在 1000 毫秒内如果无法获取到可用连接，将会返回给客户端一个 `null`。设定连接池的大小为 10 个，然后通过调节客户端的线程数来模拟无法获取连接的场景。

它通过构造函数初始化连接的最大上限，通过一个双向队列来维护连接，调用方需要先调用 `fetchConnection(long)` 方法来指定在多少毫秒内超时获取连接，当连接使用完成后，需要调用 `releaseConnection(Connection)` 方法将连接放回线程池

面试题

方法和锁

调用 `yield()`、`sleep()`、`wait()`、`notify()` 等方法对锁有何影响？

`yield()`、`sleep()` 被调用后，都不会释放当前线程所持有的锁。

调用 `wait()` 方法后，会释放当前线程持有的锁，而且当前被唤醒后，会重新去竞争锁，锁竞争到后才会执行 `wait` 方法后面的代码。

调用 `notify()` 系列方法后，对锁无影响，线程只有在 `syn` 同步代码执行完后才会自然而然的释放锁，所以 `notify()` 系列方法一般都是 `syn` 同步代码的最后一行。

wait 和 notify

为什么 `wait` 和 `notify` 方法要在同步块中调用？

原因

主要是因为 Java API 强制要求这样做，如果你不这么做，你的代码会抛出 `IllegalMonitorStateException` 异常。其实真实原因是：

这个问题并不是说只在 Java 语言中会出现，而是会在所有的多线程环境下出现。

假如我们有两个线程，一个消费者线程，一个生产者线程。生产者线程的任务可以简化成将 `count` 加一，而后唤醒消费者；消费者则是将 `count` 减一，而后在减到 0 的时候陷入睡眠：

生产者伪代码：

```
count+1;
notify();
```


消费者伪代码：

```
while(count<=0)
```

```
wait()
```

```
count--
```

这里面有问题。什么问题呢？

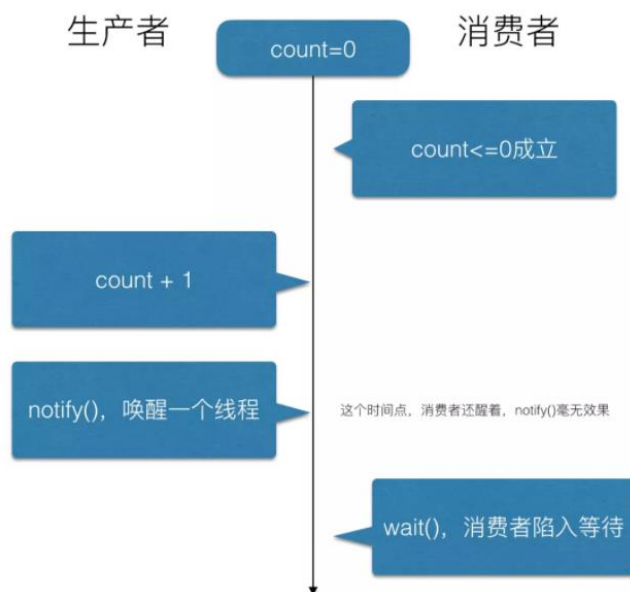
生产者是两个步骤：

1. `count+1;`
2. `notify();`

消费者也是两个步骤：

1. 检查 `count` 值；
2. 睡眠或者减一；

万一这些步骤混杂在一起呢？比如说，初始的时候 `count` 等于 0，这个时候消费者检查 `count` 的值，发现 `count` 小于等于 0 的条件成立；就在这个时候，发生了上下文切换，生产者进来了，噼里啪啦一顿操作，把两个步骤都执行完了，也就是发出了通知，准备唤醒一个线程。这个时候消费者刚决定睡觉，还没睡呢，所以这个通知就会被丢掉。紧接着，消费者就睡过去了.....



这就是所谓的 `lost wake up` 问题。

那么怎么解决这个问题呢？

现在我们应该就能够看到，问题的根源在于，消费者在检查 `count` 到调用 `wait()` 之间，`count` 就可能被改掉了。

这就是一种很常见的竞态条件。

很自然的想法是，让消费者和生产者竞争一把锁，竞争到了的，才能够修改 `count` 的值。

为什么你应该在循环中检查等待条件?

处于等待状态的线程可能会收到错误警报和伪唤醒,如果不在循环中检查等待条件,程序就会在没有满足结束条件的情况下退出。因此,当一个等待线程醒来时,不能认为它原来的等待状态仍然是有效的,在 `notify()` 方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用 `wait()` 方法效果更好的原因。

CompletableFuture

Java 的 1.5 版本引入了 `Future`,可以把它简单的理解为运算结果的占位符,它提供了两个方法来获取运算结果。

`get()`: 调用该方法线程将会无限期等待运算结果。

`get(long timeout, TimeUnit unit)`: 调用该方法线程将仅在指定时间 `timeout` 内等待结果,如果等待超时就会抛出 `TimeoutException` 异常。

`Future` 可以使用 `Runnable` 或 `Callable` 实例来完成提交的任务,它存在如下几个问题:

阻塞 调用 `get()` 方法会一直阻塞,直到等待直到计算完成,它没有提供任何方法可以在完成时通知,同时也不具有附加回调函数的功能。

链式调用和结果聚合处理 在很多时候我们想链接多个 `Future` 来完成耗时较长的计算,此时需要合并结果并将结果发送到另一个任务中,该接口很难完成这种处理。









异常处理 `Future` 没有提供任何异常处理的方式。

JDK1.8 才新加入的一个实现类 `CompletableFuture`,很好的解决了这些问题,`CompletableFuture` 实现了 `Future<T>`, `CompletionStage<T>`两个接口。实现了 `Future` 接口,意味着可以像以前一样通过阻塞或者轮询的方式获得结果。

创建

除了直接 `new` 出一个 `CompletableFuture` 的实例,还可以通过工厂方法创建 `CompletableFuture` 的实例

工厂方法:

```
  runAsync(Runnable): CompletableFuture<Void>  
  runAsync(Runnable, Executor): CompletableFuture<Void>  
  
  supplyAsync(Supplier<U>): CompletableFuture<U>  
  supplyAsync(Supplier<U>, Executor): CompletableFuture<U>
```

`Async` 表示异步,而 `supplyAsync` 与 `runAsync` 不同在于, `supplyAsync` 异步返回一个结果, `runAsync` 是 `void`。第二个函数第二个参数表示是用我们自己创建的线程池,否则采用默认的 `ForkJoinPool.commonPool()` 作为它的线程池。

获得结果的方法

```
public T get()
```

```
public T get(long timeout, TimeUnit unit)
```

```
public T getNow(T valuelIfAbsent)
```

```
public T join()
```

`getNow` 有点特殊，如果结果已经计算完则返回结果或者抛出异常，否则返回给定的 `valuelIfAbsent` 值。

`join` 返回计算的结果或者抛出一个 `unchecked` 异常(`CompletionException`)，它和 `get` 对抛出的异常的处理有些细微的区别。

辅助方法

```
public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
```

```
public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
```

`allOf` 方法是当所有的 `CompletableFuture` 都执行完后执行计算。

`anyOf` 方法是当任意一个 `CompletableFuture` 执行完后就会执行计算，计算的结果相同。

`CompletionStage` 是一个接口，从命名上看得知是一个完成的阶段，它代表了一个特定的计算的阶段，可以同步或者异步的被完成。你可以把它看成一个计算流水线上一个单元，并最终会产生一个最终结果，这意味着几个 `CompletionStage` 可以串联起来，一个完成的阶段可以触发下一阶段的执行，接着触发下一次，再接着触发下一次，.....。

总结 `CompletableFuture` 几个关键点：

1、计算可以由 `Future`，`Consumer` 或者 `Runnable` 接口中的 `apply`，`accept` 或者 `run` 等方法表示。

2、计算的执行主要有以下

a. 默认执行

b. 使用默认的 `CompletionStage` 的异步执行提供者异步执行。这些方法名使用 `someActionAsync` 这种格式表示。

c. 使用 `Executor` 提供者异步执行。这些方法同样也是 `someActionAsync` 这种格式，但是会增加一个 `Executor` 参数。

`CompletableFuture` 里大约有五十种方法，但是可以进行归类，

变换类 `thenApply`：

```
m thenApply(Function<? super T, ? extends U>): Completable
m thenApplyAsync(Function<? super T, ? extends U>): Cor
m thenApplyAsync(Function<? super T, ? extends U>, Exec
```

关键入参是函数式接口 `Function`。它的入参是上一个阶段计算后的结果，返回值是经过转化后结果。

消费类 `thenAccept`：

```
m thenAccept(Consumer<? super T>): CompletableFuture
m thenAcceptAsync(Consumer<? super T>): Completable
m thenAcceptAsync(Consumer<? super T>, Executor): Cor
```

关键入参是函数式接口 `Consumer`。它的入参是上一个阶段计算后的结果，没有返回值。

执行操作类 `thenRun`:

```
m  thenRun(Runnable): CompletableFuture<Void> ↑Comple
m  thenRunAsync(Runnable): CompletableFuture<Void> ↑C
m  thenRunAsync(Runnable, Executor): CompletableFuture
```

对上一步的计算结果不关心，执行下一个操作，入参是一个 `Runnable` 的实例，表示上一步完成后执行的操作。

结合转化类:

```
m  thenCombine(CompletionStage<? extends U>, BiFunction<? super T, ? super U, ? extends V>): CompletableFuture<V>
m  thenCombineAsync(CompletionStage<? extends U>, BiFunction<? super T, ? super U, ? extends V>): CompletableFuture
m  thenCombineAsync(CompletionStage<? extends U>, BiFunction<? super T, ? super U, ? extends V>, Executor): Complet
```

需要上一步的处理返回值，并且 `other` 代表的 `CompletionStage` 有返回值之后，利用这两个返回值，进行转换后返回指定类型的值。

两个 `CompletionStage` 是并行执行的，它们之间并没有先后依赖顺序，`other` 并不会等待先前的 `CompletableFuture` 执行完毕后再执行。

结合转化类

```
m  thenCompose(Function<? super T, ? extends CompletionStage<U>>): CompletableFuture<U>
m  thenComposeAsync(Function<? super T, ? extends CompletionStage<U>>): CompletableFuture
m  thenComposeAsync(Function<? super T, ? extends CompletionStage<U>>, Executor): Comple
```

对于 `Compose` 可以连接两个 `CompletableFuture`，其内部处理逻辑是当第一个 `CompletableFuture` 处理没有完成时会合并成一个 `CompletableFuture`，如果处理完成，第二个 `future` 会紧接上一个 `CompletableFuture` 进行处理。

第一个 `CompletableFuture` 的处理结果是第二个 `future` 需要的输入参数。

结合消费类:

```
m  thenAcceptBoth(CompletionStage<? extends U>, BiConsumer<? super T, ? super U>): CompletableFuture<Void>
m  thenAcceptBothAsync(CompletionStage<? extends U>, BiConsumer<? super T, ? super U>): CompletableFuture<V>
m  thenAcceptBothAsync(CompletionStage<? extends U>, BiConsumer<? super T, ? super U>, Executor): Completable
```

需要上一步的处理返回值，并且 `other` 代表的 `CompletionStage` 有返回值之后，利用这两个返回值，进行消费

运行后执行类:

```
m  runAfterBoth(CompletionStage<?>, Runnable): CompletableFuture<Void> ↑CompletionStag
m  runAfterBothAsync(CompletionStage<?>, Runnable): CompletableFuture<Void> ↑Completi
m  runAfterBothAsync(CompletionStage<?>, Runnable, Executor): CompletableFuture<Void> ↑
```

不关心这两个 `CompletionStage` 的结果，只关心这两个 `CompletionStage` 都执行完毕，之后再进行操作（`Runnable`）。

取最快转换类:

```

m  applyToEither(CompletionStage<? extends T>, Function<? super T, U>): CompletableFuture<U> ↑CompletionSta
m  applyToEitherAsync(CompletionStage<? extends T>, Function<? super T, U>): CompletableFuture<U> ↑Completi
m  applyToEitherAsync(CompletionStage<? extends T>, Function<? super T, U>, Executor): CompletableFuture<U> ↑

```

两个 CompletionStage，谁计算的快，我就用那个 CompletionStage 的结果进行下一步的转化操作。现实开发场景中，总会碰到有两种渠道完成同一个事情，所以就可以调用这个方法，找一个最快的结果进行处理。

取最快消费类：

```

m  acceptEither(CompletionStage<? extends T>, Consumer<? super T>): CompletableFuture<Void> ↑Comple
m  acceptEitherAsync(CompletionStage<? extends T>, Consumer<? super T>): CompletableFuture<Void> ↑C
m  acceptEitherAsync(CompletionStage<? extends T>, Consumer<? super T>, Executor): CompletableFuture<

```

两个 CompletionStage，谁计算的快，我就用那个 CompletionStage 的结果进行下一步的消费操作。

取最快运行后执行类：

```

m  runAfterEither(CompletionStage<?>, Runnable): CompletableFuture<Void> ↑CompletionSta
m  runAfterEitherAsync(CompletionStage<?>, Runnable): CompletableFuture<Void> ↑Comple
m  runAfterEitherAsync(CompletionStage<?>, Runnable, Executor): CompletableFuture<Void>

```

两个 CompletionStage，任何一个完成了都会执行下一步的操作(Runnable)。

异常补偿类：

```

public CompletableFuture<T> exceptionally(
    Function<Throwable, ? extends T> fn) {
    return uniExceptionallyStage(fn);
}

```

当运行时出现了异常，可以通过 exceptionally 进行补偿。

运行后记录结果类：

```

m  whenComplete(BiConsumer<? super T, ? super Throwable>): CompletableFuture<T> ↑CompletionSta
m  whenCompleteAsync(BiConsumer<? super T, ? super Throwable>): CompletableFuture<T> ↑Completi
m  whenCompleteAsync(BiConsumer<? super T, ? super Throwable>, Executor): CompletableFuture<T> ↑

```

action 执行完毕后它的结果返回原始的 CompletableFuture 的计算结果或者返回异常。所以不会对结果产生任何的作用。

运行后处理结果类：

```

m  handle(BiFunction<? super T, Throwable, ? extends U>): CompletableFuture<U> ↑CompletionSta
m  handleAsync(BiFunction<? super T, Throwable, ? extends U>): CompletableFuture<U> ↑Completi
m  handleAsync(BiFunction<? super T, Throwable, ? extends U>, Executor): CompletableFuture<U> ↑C

```

运行完成时，对结果的处理。这里的完成时有两种情况，一种是正常执行，返回值。另外一种是因为异常抛出造成程序的中断。

本文档链接：

https://note.youdao.com/ynotesshare/index.html?id=7d4dcca431354d32d7a541fd685126b7&type=note&_time=1677239897610