Java【并发】面试题

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

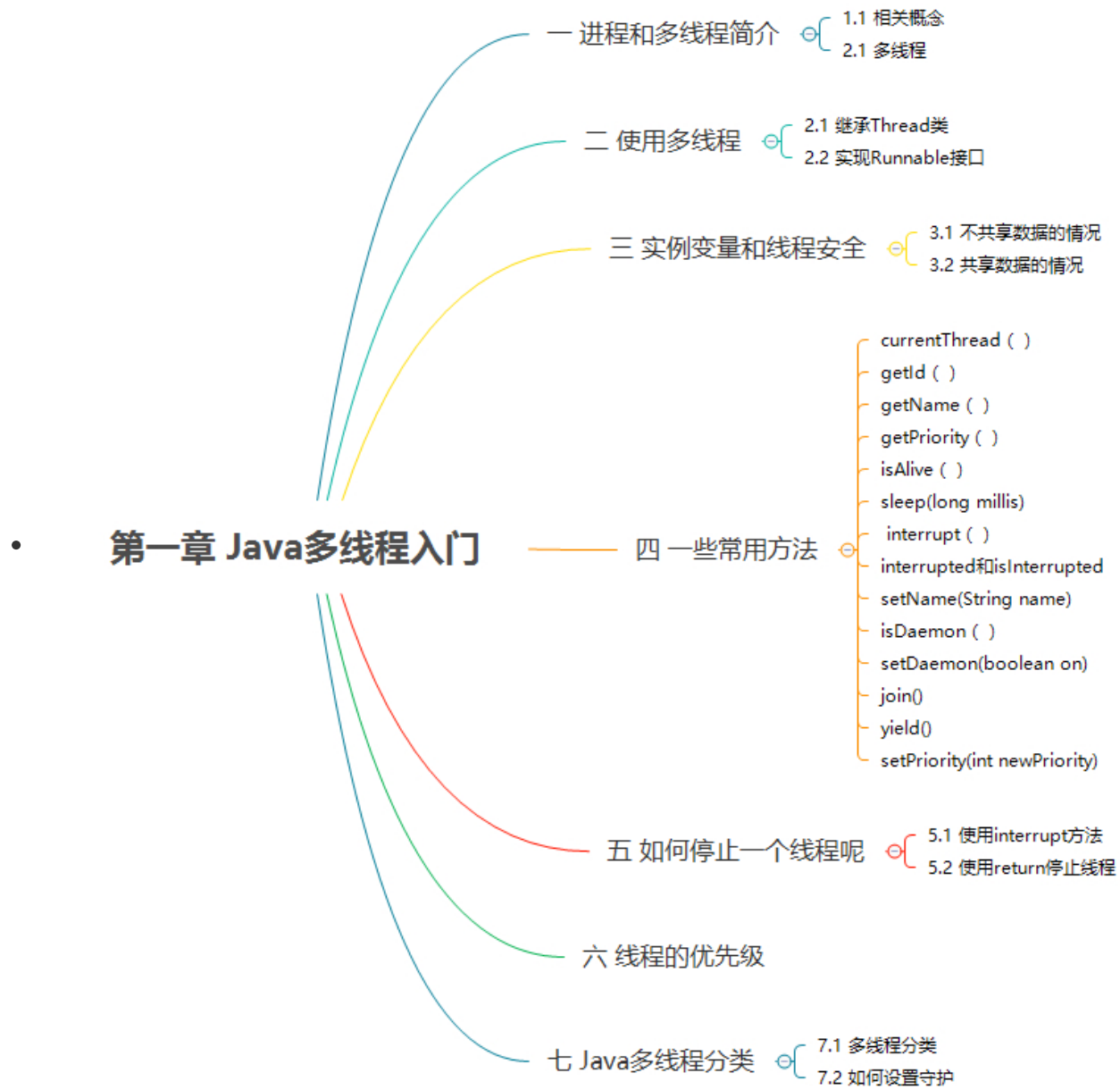
如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 Java【并发】面试题的大保健。

而题目的难度，尽量按照从容易到困难的顺序，逐步下去。

因为 Java 并发涉及到的内容会非常多，本面试题可能很难覆盖到所有的知识点，所以推荐 《Java并发编程的艺术》 。并且，本文会将面试题和该书的章节，大体保持一致。嘻嘻~

另外，本文涉及的面试题会超级超级超级多，所以已经分了小节，胖友要注意哟。

Java 线程



- 第四章 等待/通知（wait/notify）机制
  - 一 等待/通知机制介绍
    - 1.1 不使用等待/通知机制
      - 如果轮询时间的间隔太小会浪费CPU资源，轮询时间的间隔太大，就可能取不到自己想要的数据。
    - 1.2 什么是等待/通知机制？
    - 1.3 等待/通知机制的相关方法
      - notify()
      - notifyAll()
      - wait()
      - wait(long)
      - wait(long，int)
  - 二 等待/通知机制的实现
    - 2.1 我的第一个等待/通知机制程序
      - notify()执行后并不会立即释放锁
    - 2.2线程的基本状态
      - 1. 新建(new)
      - 2.可运行(runnable)
      - 3.运行(running)
      - 4. 阻塞(block)
      - 5. 死亡(dead)
    - 2.3 notify()锁不释放
      - 必须执行完notify()方法所在的synchronized代码块后才释放。
    - 2.4 当interrupt方法遇到wait方法
      - 当线程呈wait状态时，对线程对象调用interrupt方法会出现InterrupedException异常。
- 第五章 线程间通信知识点补充
  - 一 管道输入/输出流
    - 管道输入/输出流主要用于线程之间的数据传输**，而且传输的媒介为内存。
    - 面向字节： PipedOutputStream、 PipedInputStream
    - 面向字符: PipedWriter、 PipedReader
  - 二 Thread.join()的使用
    - 主线程需要等待子线程执行完成之后再结束，这个时候就要用到join()方法了。另外，一个线程需要等待另一个线程也需要用到join()方法。
    - join方法使用
    - join(long millis)方法的使用
  - 三 ThreadLocal的使用
    - hreadLocal类主要解决的就是让每个线程绑定自己的值，可以将ThreadLocal类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。
    - ThreadLocal类相关方法
      - get()
      - set(T value)
      - remove()
      - initialValue()
    - InheritableThreadLocal
- 第七章 并发编程中一些问题
  - 多线程就一定好吗？快吗？
    - CPU通过给每个线程分配CPU时间片来实现伪同时运行
    - 并发编程并不总是能提高程序运行速度的
  - 上下文切换
    - 任务从保存到再加载的过程就是一次上下文切换。
    - 上下文切换对系统来说意味着消耗大量的 CPU 时间
    - 如何减少上下文切换的次数呢？？？
  - 减少上下文切换
    - 上下文切换又分为2种：让步式上下文切换和抢占式上下文切换。
    - 减少锁的使用。因为多线程竞争锁时会引起上下文切换。
    - 使用CAS算法。这种算法也是为了减少锁的使用。CAS算法是一种无锁算法。
    - 减少线程的使用。人物很少的时候创建大量线程会导致大量线程都处于等待状态。
    - 使用协程。
    - CAS算法和协程是什么？？？
  - 避免死锁
    - 锁是一个好东西，但是使用不当就会造成死锁。一旦死锁产生程序就无法继续运行下去。所以如何避免死锁的产生，在我们使用并发编程时至关重要。
    - 避免死锁有哪几种方式？？？
  - 解决资源限制
    - 什么是资源限制？？？
    - 资源限制引发的问题
    - 如何解决资源限制的问题
    - 在资源限制的情况下如何进行并发编程

简述线程、进程、程序的基本概念?

程序

程序，是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

进程

进程，是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如CPU时间，内存空间，文件，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。

📣 线程

线程，与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

> ：如下是可选内容。
>
> 另外，Java 线程是重量级的，每个线程默认使用 1024KB 的内存，所以一个 Java 进程是无法开启大量线程的。感兴趣的胖友，可以看 《Java 中的轻量级线程？》 的讨论，没准未来 Java 也有内置的协程（Coroutine）。

📣 三者之间的关系

- 线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。
- 从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

📣 线程有什么优缺点?

1）好处

- 使用多线程可以把程序中占据时间长的任务放到后台去处理，如图片、视屏的下载。
- 发挥多核处理器的优势，并发执行让系统运行的更快、更流畅，用户体验更好。

2）坏处

- 大量的线程降低代码的可读性。
- 更多的线程需要更多的内存空间。
- 当多个线程对同一个资源出现争夺时候要注意线程安全的问题。

📣 你了解守护线程吗？它和非守护线程有什么区别?

Java 中的线程分为两种：守护线程（Daemon）和用户线程（User）。

- 任何线程都可以设置为守护线程和用户线程，通过方法 `Thread#setDaemon(boolean on)` 设置。`true` 则把该线程设置为守护线程，反之则为用户线程。
- `Thread#setDaemon(boolean on)` 方法，必须在 `Thread#start()` 方法之前调用，否则运行时会抛出异常。

唯一的区别是：

> 程序运行完毕，JVM 会等待非守护线程完成后关闭，但是 JVM 不会等待守护线程。

- 判断虚拟机(JVM)何时离开，Daemon 是为其他线程提供服务，如果全部的 User Thread 已经撤离，Daemon 没有可服务的线程，JVM 撤离。
- 也可以理解为守护线程是 JVM 自动创建的线程（但不一定），用户线程是程序创建的线程。比如，JVM 的垃圾回收线程是一个守护线程，当所有线程已经撤离，不再产生垃圾，守护线程自然就没事可干了，当垃圾回收线程是 Java 虚拟机上仅剩的线程时，Java 虚拟机会自动离开。

扩展：Thread Dump 打印出来的线程信息，含有 daemon 字样的线程即为守护进程。可能会有：服务守护进程、编译守护进程、Windows 下的监听 Ctrl + break 的守护进程、Finalizer 守护进程、引用处理守护进程、GC 守护进程。

关于守护线程的各种骚操作，可以看看 《Java 守护线程概述》 。

📣 什么是线程组，为什么在 Java 中不推荐使用?

> ：这是个小众知识，了解即可。貌似，也重来没使用过这个类。

ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。

> 简单的说，ThreadGroup 为了方便线程的管理。

为什么不推荐使用？ThreadGroup API 比较薄弱，它并没有比 Thread 提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(uncaught exception handler)。但在 Java5 中 Thread 类也添加了 `#setUncaughtExceptionHandler(UncaughtExceptionHandler eh)` 方法，所以 ThreadGroup 是已经过时的，不建议继续使用。

什么是多线程上下文切换?

多线程会共同使用一组计算机上的 CPU ，而线程数大于给程序分配的 CPU 数量时，为了让各个线程都有执行的机会，就需要轮转使用 CPU 。

不同的线程切换使用 CPU 发生的切换数据等，就是上下文切换。

- 在上下文切换过程中，CPU 会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。在程序中，上下文切换过程中的“页码”信息是保存在进程控制块（PCB）中的。PCB还经常被称作“切换帧”（switchframe）。“页码”信息会一直保存到CPU的内存中，直到他们被再次使用。
- 上下文切换是存储和恢复 CPU 状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

📣 Java 中用到的线程调度算法是什么?

假设计算机只有一个 CPU ，则在任意时刻只能执行一条机器指令，每个线程只有获得 CPU 的使用权才能执行指令。

- 所谓多线程的并发运行，其实是指从宏观上看，各个线程轮流获得 CPU 的使用权，分别执行各自的任务。
- 在运行池中，会有多个处于就绪状态的线程在等待 CPU ，Java 虚拟机的一项任务就是负责线程的调度，线程调度是指按照特定机制为多个线程分配 CPU 的使用权。

有两种调度模型：分时调度模型和抢占式调度模型。

- 分时调度模型是指让所有的线程轮流获得 CPU 的使用权,并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。
- Java 虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用 CPU ，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用 CPU 。处于运行状态的线程会一直运行，直至它不得不放弃 CPU 。

> 如非特别需要，尽量不要用，防止线程饥饿。

📣 什么是线程饥饿?

饥饿，一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java 中导致饥饿的原因：

- 高优先级线程吞噬所有的低优先级线程的 CPU 时间。
- 线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
- 线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)，因为其他线程总是被持续地获得唤醒。

你对线程优先级的理解是什么?

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。

- 我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个 int 变量(从1-10)，1 代表最低优先级，10 代表最高优先级。
- Java 的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

线程的生命周期?

线程一共有五个状态，分别如下：

- 新建(new)：当创建Thread类的一个实例（对象）时，此线程进入新建状态（未被启动）。例如：`Thread t1 = new Thread()`。
- 可运行(runnable)：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。例如：`t1.start()`。

  > 有些文章，会称可运行(runnable)为就绪，意思是一样的。

- 运行(running)：线程获得 CPU 资源正在执行任务（`#run()` 方法），此时除非此线程自动放弃 CPU 资源或者有优先级更高的线程进入，线程将一直运行到结束。
- 死亡(dead)：当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。
  - 自然终止：正常运行完 `#run()` 方法，终止。
  - 异常终止：调用 `#stop()` 方法，让一个线程终止运行。
- 堵塞(blocked)：由于某种原因导致正在运行的线程让出 CPU 并暂停自己的执行，即进入堵塞状态。直到线程进入可运行(runnable)状态，才有机会再次获得 CPU 资源，转到运行(running)状态。阻塞的情况有三种：
  - 正在睡眠：调用 `#sleep(long t)` 方法，可使线程进入睡眠方式。

    > 一个睡眠着的线程在指定的时间过去可进入可运行(runnable)状态。

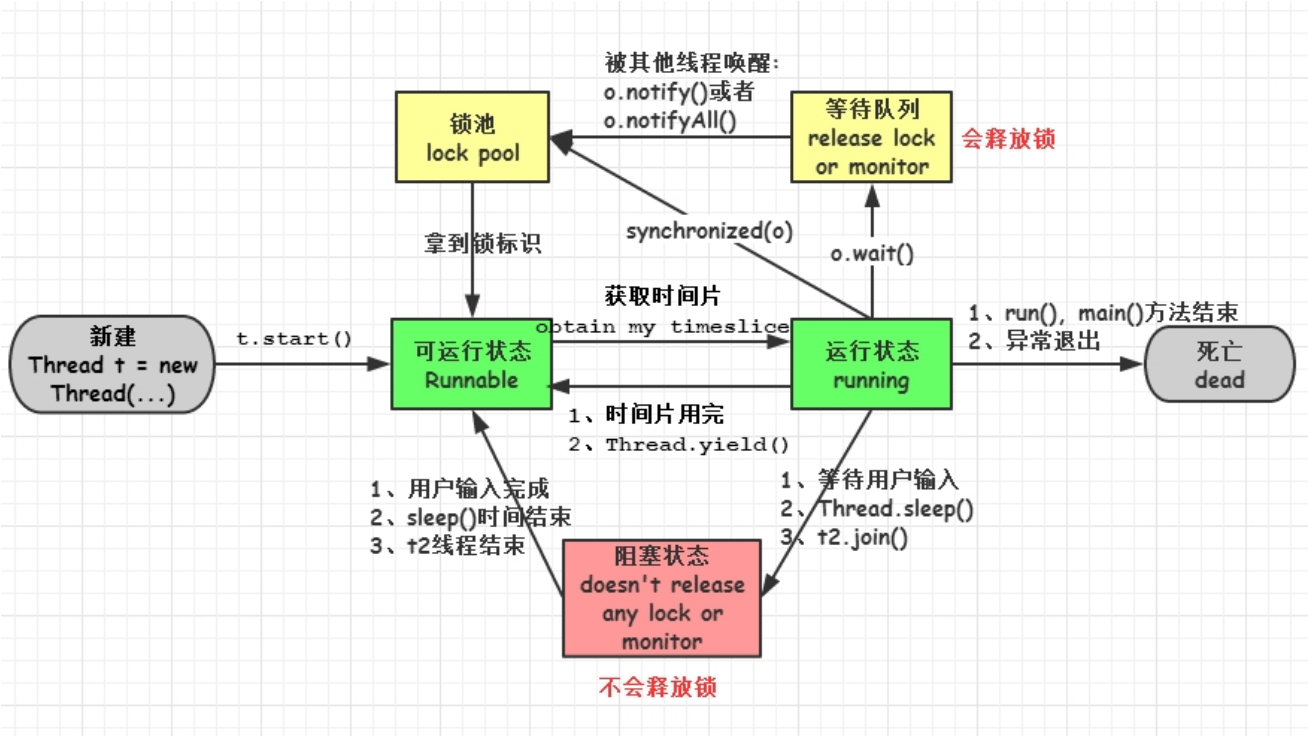
  - 正在等待：调用 `#wait()` 方法。

    > 调用 `notify()` 方法，回到就绪状态。

  - 被另一个线程所阻塞：调用 `#suspend()` 方法。

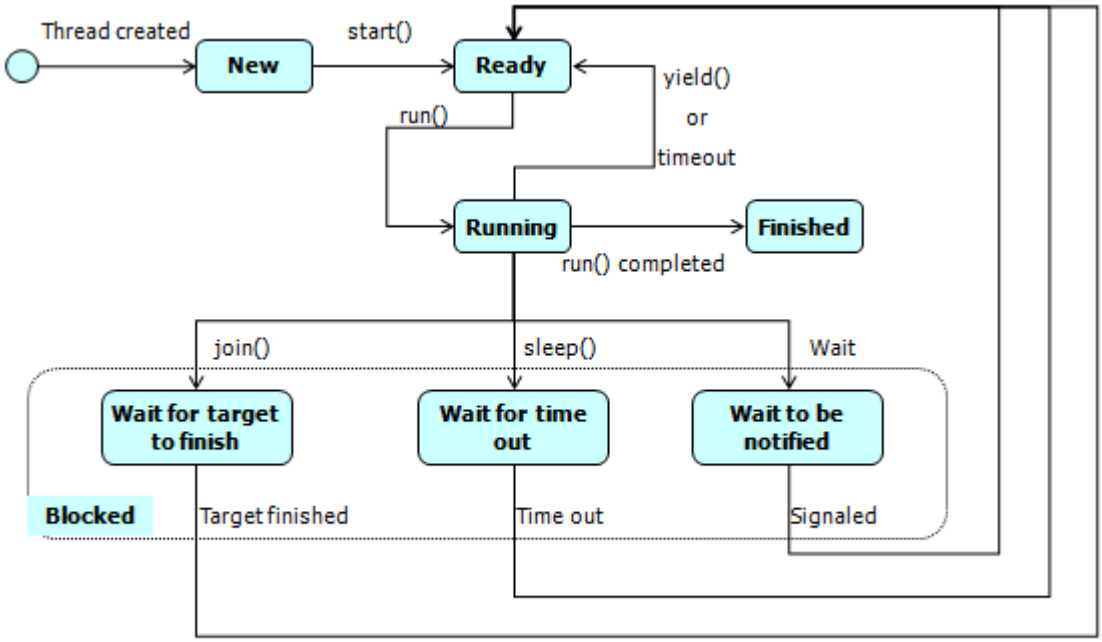
    > 调用 `#resume()` 方法，就可以恢复。

整体如下图所示：



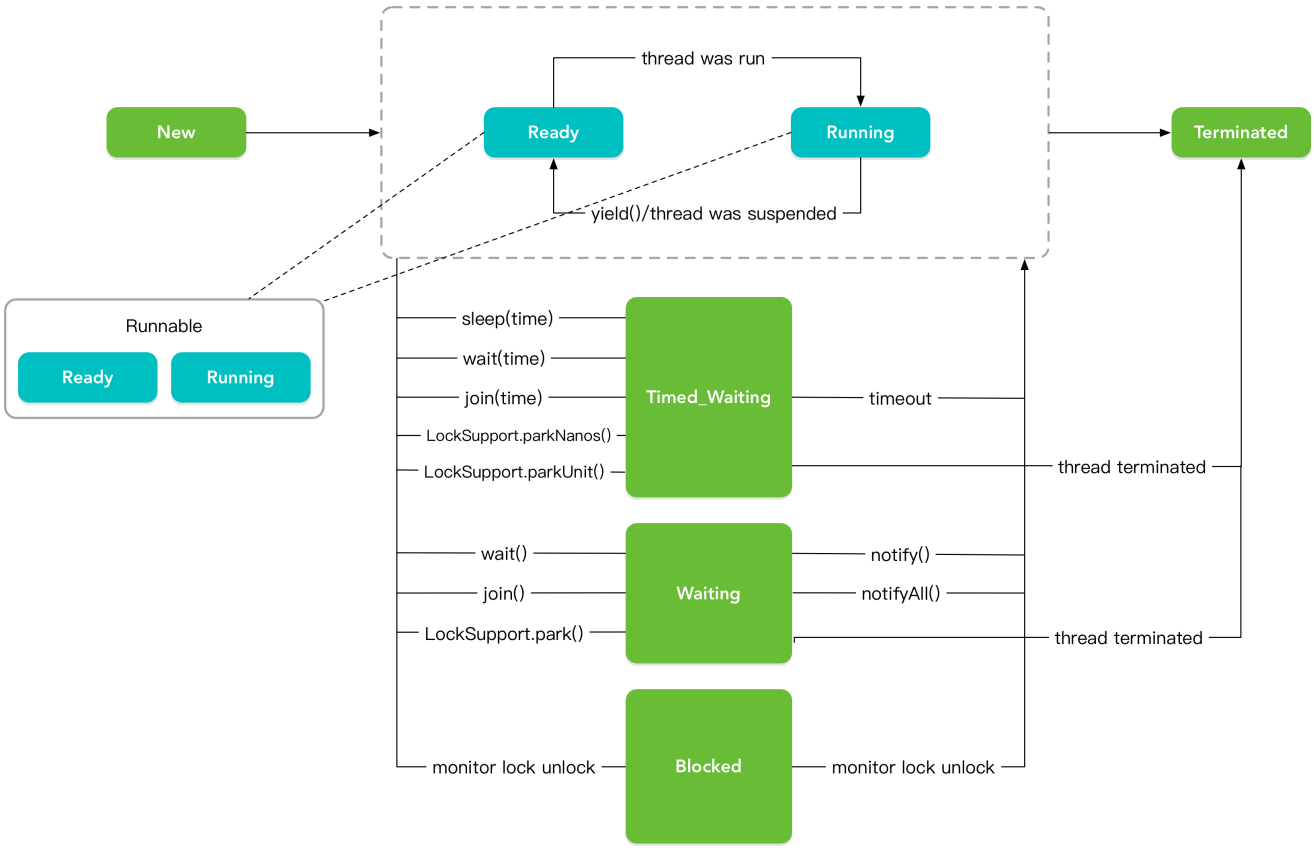
- 中间一行是线程的顺畅的执行过程的四个状态。其上下两侧，是存在对应的情况，达到阻塞状态和恢复执行的过程。
- 有一点要注意，新建(new)和死亡(dead)是单向的状态，不可重复。**
- 理解线程的状态，可以用早起坐地铁来比喻这个过程：
    - 还没起床：sleeping 。
    - 起床收拾好了，随时可以坐地铁出发：Runnable 。
    - 等地铁来：Waiting 。
    - 地铁来了，但要排队上地铁：I/O 阻塞 。
    - 上了地铁，发现暂时没座位：synchronized 阻塞。
    - 地铁上找到座位：Running 。
    - 到达目的地：Dead 。

如下是另外一个图，把阻塞的情况，放在了一起，也可以作为参考：



线程的生命周期图

无意中，又看到一张画的更牛逼的，如下图：



如何结束一个一直运行的线程?

一般来说，有两种方式：

- 方式一，使用退出标志，这个 flag 变量要多线程可见。

> 在这种方式中，之所以引入共享变量，是因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号，通知中断线程的执行。

- 方式二，使用 interrupt 方法，结合 isInterrupted 方法一起使用。

> 如果一个线程由于等待某些事件的发生而被阻塞，又该怎样停止该线程呢？这种情况经常会发生，比如当一个线程由于需要等候键盘输入而被阻塞，或者调用 `Thread#join()` 方法，或者 `Thread#sleep(...)` 方法，在网络中调用 `ServerSocket#accept()` 方法，或者调用了 `DatagramSocket#receive()` 方法时，都有可能导致线程阻塞，使线程处于处于不可运行状态时。即使主程序中将该线程的共享变量设置为 `true` ，但该线程此时根本无法检查循环标志，当然也就无法立即中断。
>
> 这里我们给出的建议是，不要使用 `Thread#stop()· 方法，而是使用 Thread 提供的`#interrupt()` 方法，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常，从而使线程提前结束阻塞状态，退出堵塞代码。

所以，方式一和方式二，并不是冲突的两种方式，而是可能根据实际场景下，进行结合。

📣 一个线程如果出现了运行时异常会怎么样?

如果这个异常没有被捕获的话，这个线程就停止执行了。

另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放。

创建线程的方式及实现?

Java 中创建线程主要有三种方式：

> 具体的每种方式的代码实现，可以看看 《Java创建线程的四种方式》 。
>
> 关于文章中的方式四，实际是基于线程池的方式，使用下面的三种方式，也是生产实践中，最为推荐和常用的方式。

- 方式一，继承 Thread 类创建线程类。
- 方式二，通过 Runnable 接口创建线程类。
- 方式三，通过 Callable 和 Future 创建线程。

创建线程的三种方式的对比：

- 使用方式一
  - 优点：编写简单，如果需要访问当前线程，则无需使用 `Thread#currentThread()` 方法，直接使用 `this` 即可获得当前线程。
  - 缺点：线程类已经继承了 Thread 类，所以不能再继承其他父类。
- 使用方式二、或方式三
  - 优点：
    - 线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。
    - 在这种方式下，多个线程可以共享同一个 `target` 对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将 CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

```
Runnable runner = new Runnable(){ ... };
// 通过new Thread(target, name) 方法创建新线程
new Thread(runna,"新线程1").start();
new Thread(runna,"新线程2").start();
```

        - 当然，实际比较少这么用。
    - **【最重要】可以使用线程池。**
  - 缺点：编程稍微复杂，如果要访问当前线程，则必须使用`Thread#currentThread()`方法。

📣 start 和 run 方法有什么区别?

- 当你调用 start 方法时，你将创建新的线程，并且执行在 run 方法里的代码。
- 但是如果你直接调用 run 方法，它不会创建新的线程也不会执行调用线程的代码，只会把 run 方法当作普通方法去执行。

一个线程运行时发生异常会怎样?

如果异常没有被捕获该线程将会停止执行。`Thread.UncaughtExceptionHandler`是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候 JVM 会使用`Thread#getUncaughtExceptionHandler()`方法来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给 handler 的`#uncaughtException(exception)`方法进行处理。

具体的使用，可以看看 《JAVA 多线程之 UncaughtExceptionHandler —— 处理非正常的线程中止》 。

如何使用 wait + notify 实现通知机制?

> wait + notify 对于大多数胖友，一开始理解可能会比较困难，多看多理解吧。

在 Java 发展史上，曾经使用 suspend、resume 方法对于线程进行阻塞唤醒，但随之出现很多问题，比较典型的还是死锁问题。

解决方案可以使用以对象为目标的阻塞，即利用 Object 类的 wait 和 notify方法实现线程阻塞。

- 首先，wait、notify 方法是针对对象的，调用任意对象的 wait 方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify 方法则将随机解除该对象阻塞的线程，但它需要重新获取改对象的锁，直到获取成功才能往下执行。
- 其次，wait、notify 方法必须在`synchronized`块或方法中被调用，并且要保证同步块或方法的锁对象与调用 wait、notify 方法的对象是同一个，如此一来在调用 wait 之前当前线程就已经成功获取某对象的锁，执行 wait 阻塞后当前线程就将之前获取的对象锁释放。

具体的实现，看看 《Wait / Notify通知机制解析》 文章。

通过 wait + notify 的组合，可以通知机制，不过我们也可以使用其它工具，胖友可以思考下。例如如下的每一个方式：

- Condition
- CountDownLatch
- Queue
- Future
- ...

> ：这个问题可以衍生下，Java 如何实现多线程之间的通讯和协作？具体的可以看看 《Java多线程——线程间协作方式总结及使用示例》 文章，当然不仅限于该文章所提供的方式。😺 胖友可以认真思索下。

📣 Thread类的 sleep 方法和对象的 wait 方法都可以让线程暂停执行，它们有什么区别?

> 关于这个问题，可以结合 「线程的生命周期？」 问题的图，一起瞅瞅。

- sleep 方法，是线程类 Thread 的静态方法。调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态）

- wait 方法，是 Object 类的方法。调用对象的 `#wait()` 方法，会导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的 `#notify()` 方法（或 `#notifyAll()` 方法）时，才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

📣 请说出与线程同步以及线程调度相关的方法?

- wait 方法，使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁。
- sleep 方法，使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常。
- notify 方法，唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关。
- notityAll 方法，唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态。

📣 notify 和 notifyAll 有什么区别?

当一个线程进入 wait 之后，就必须等其他线程 notify/notifyAll 。

- 使用 notifyAll,可以唤醒所有处于 wait 状态的线程，使其重新进入锁的争夺队列中，而 notify 只能唤醒一个。
- 如果没把握，建议 notifyAll ，防止 notify 因为信号丢失而造成程序错误。

关于 notify 的信息丢失，可以看看 《wait 和 notify 的坑》 文章。

📣 为什么 wait, notify 和 notifyAll 这三方法不在 Thread 类里面?

一个很明显的原因是 Java 提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。

由于 wait，notify 和 notifyAll 方法都是锁级别的操作，所以把它们定义在 Object 类中，因为锁属于对象。

📣 为什么 wait 和 notify 方法要在同步块中调用?

- 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。
- 还有一个原因是为了避免 wait 和 notify 之间产生竞态条件。

📣 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。

所以，我们不能写 `if (condition)` 而应该是 `while (condition)` ，特别是 CAS 竞争的时候。示例代码如下：

```
// The standard idiom for using the wait method
synchronized (obj) {
    while (condition does not hold) {
        obj.wait(); // (Releases lock, and reacquires on wakeup)
    }
    ... // Perform action appropriate to condition
}
```

- 另外，也可以看看 《wait 必须放在 while 循环里面的原因探析》

sleep、join、yield 方法有什么区别?

1）sleep 方法

在指定的毫秒数内，让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。让其他线程有机会继续执行，但它并不释放对象锁。也就是如果有 `synchronized` 同步块，其他线程仍然不能访问共享数据。注意该方法要捕获异常。

比如有两个线程同时执行(没有 `synchronized`)，一个线程优先级为 `MAX_PRIORITY` ，另一个为 `MIN_PRIORITY` 。

- 如果没有 sleep 方法，只有高优先级的线程执行完成后，低优先级的线程才能执行。但当高优先级的线程 `#sleep(5000)` 后，低优先级就有机会执行了。
- 总之，sleep 方法，可以使低优先级的线程得到执行的机会，当然也可以让同优先级、高优先级的线程有执行的机会。

2）yield 方法

yield 方法和 sleep 方法类似，也不会释放“锁标志”，区别在于：

- 它没有参数，即 yield 方法只是使当前线程重新回到可执行状态，所以执行yield 的线程有可能在进入到可执行状态后马上又被执行。
- 另外 yield 方法只能使同优先级或者高优先级的线程得到执行机会，这也和 sleep 方法不同。

3）join 方法

Thread 的非静态方法 join ，让一个线程 B “加入”到另外一个线程 A 的尾部。在线程 A 执行完毕之前，线程 B 不能工作。示例代码如下：

```
Thread t = new MyThread();
t.start();
t.join();
```

- 保证当前线程停止执行，直到该线程所加入的线程 `t` 完成为止。然而，如果它加入的线程 `t` 没有存活，则当前线程不需要停止。

📣 线程的 sleep 方法和 yield 方法有什么区别?

- sleep 方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会。yield 方法只会给相同优先级或更高优先级的线程以运行的机会。
- 线程执行 sleep 方法后转入阻塞（blocked）状态，而执行 yield 方法后转入就绪（ready）状态。
- sleep 方法声明抛出 InterruptedException 异常，而 yield 方法没有声明任何异常。
- sleep 方法比 yield 方法（跟操作系统 CPU 调度相关）具有更好的可移植性。

> ：实际场景下，我们很少使用 yield 方法噢。

📣 为什么 Thread 类的 sleep 和 yield 方法是静态的?

Thread 类的 sleep 和 yield 方法，将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

📣 sleep(0) 有什么用途?

`Thread#sleep(0)` 方法，并非是真的要线程挂起 0 毫秒，意义在于这次调用 `Thread#sleep(0)` 方法，把当前线程确实的被冻结了一下，让其他线程有机会优先执行。 `Thread#sleep(0)` 方法，是你的线程暂时放弃 CPU，也就是释放一些未用的时间片给其他线程或进程使用，就相当于一个**让位动作**。

感兴趣的胖友，可以看看 《Sleep(0) 的妙用》 的示例。

📣 你如何确保 main 方法所在的线程是 Java 程序最后结束的线程?

> 考点，就是 join 方法。

我们可以使用 Thread 类的 `#join()` 方法，来确保所有程序创建的线程在 main 方法退出前结束。

interrupted 和 isInterrupted 方法的区别?

1）interrupt 方法

`Thread#interrupt()` 方法，用于中断线程。调用该方法的线程的状态为将被置为"中断"状态。

注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（也就是线程中断后会抛出 InterruptedException 的方法）就是在监视线程的中断状态，一旦线程的中断状态被置为"中断状态"，就会抛出中断异常。

2）interrupted

`Thread#interrupted()` **静态**方法，查询当前线程的中断状态，并且清除原状态。如果一个线程被中断了，第一次调用 `#interrupted()` 方法则返回 `true` ，第二次和后面的就返回 `false` 了。

```
// Thread.java

public static boolean interrupted() {
    return currentThread().isInterrupted(true); // 清理
}

private native boolean isInterrupted(boolean ClearInterrupted);
```

3）interrupted

`Thread#isInterrupted()` 方法，查询指定线程的中断状态，不会清除原状态。代码如下：

```
// Thread.java

public boolean isInterrupted() {
    return isInterrupted(false); // 不清楚
}

private native boolean isInterrupted(boolean ClearInterrupted);
```

什么叫线程安全?

线程安全，是编程中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。

📣 Servlet 是线程安全吗?

Servlet 不是线程安全的，Servlet 是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。

📣 Struts2 是线程安全吗?

Struts2 的 Action 是多实例多线程的，是线程安全的，每个请求过来都会 `new` 一个新的 Action 分配给这个请求，请求完成后销毁。

📣 SpringMVC 是线程安全吗?

不是的，和 Servlet 类似的处理流程。

📣 单例模式的线程安全性?

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法，我总结一下：

- 饿汉式单例模式的写法：线程安全
- 懒汉式单例模式的写法：非线程安全
- 双检锁单例模式的写法：线程安全

多线程同步和互斥有几种实现方法，都是什么?

1）**线程同步**

线程同步，是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。

线程间的同步方法，大体可分为两类：用户模式和内核模式。顾名思义：

- 内核模式，就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态。内核模式下的方法有：
  - 事件
  - 信号量
  - 互斥量
- 用户模式，就是不需要切换到内核态，只在用户态完成操作。用户模式下的方法有：
  - 原子操作（例如一个单一的全局变量）
  - 临界区

2）**线程互斥**

线程互斥，是指对于共享的进程系统资源，在各单个线程访问时的排它性。

- 当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。
- 线程互斥可以看成是一种特殊的线程同步。

📣 如何在两个线程间共享数据?

在两个线程间**共享变量**，即可实现共享。

一般来说，共享变量要求变量本身是线程安全的，然后在线程内使用的时候，如果有对共享变量的复合操作，那么也得保证复合操作的线程安全性。

📣 怎么检测一个线程是否拥有锁?

调用 `Thread#holdsLock(Object obj)` **静态**方法，它返回 `true` 如果当且仅当当前线程拥有某个具体对象的锁。代码如下：

```
// Thread.java

public static native boolean holdsLock(Object obj);
```

📣 10 个线程和 2 个线程的同步代码，哪个更容易写?

从写代码的角度来说，两者的复杂度是相同的，因为同步代码与线程数量是相互独立的。

但是同步策略的选择依赖于线程的数量，因为越多的线程意味着更大的竞争，所以你需要利用同步技术，如锁分离，这要求更复杂的代码和专业知识。

什么是 ThreadLocal 变量?

ThreadLocal ，是 Java 里一种特殊的变量。每个线程都有一个 ThreadLocal 就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。

它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用 ThreadLocal 让 SimpleDateFormat 变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。

- 首先，通过复用减少了代价高昂的对象的创建个数。
- 其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

😺 所以，ThreadLocal 很适合实现线程级的单例。

详细的，可以看看 《Java并发编程：深入剖析ThreadLocal》 文章。

关于源码，可以看看 《【死磕 Java 并发】—— 深入分析 ThreadLocal》 。

📣 什么是 InheritableThreadLocal ?

InheritableThreadLocal 类，是 ThreadLocal 类的子类。ThreadLocal 中每个线程拥有它自己的值，与 ThreadLocal 不同的是，**InheritableThreadLocal 允许一个线程以及该线程创建的所有子线程都可以访问它保存的值。**

- 具体的实现原理，可以看看 《Java 多线程：InheritableThreadLocal 实现原理》 文章。
- 具体的使用示例，可以看看 《Spring Cloud中Hystrix 线程隔离导致ThreadLocal数据丢失》 。

📣 在多线程环境下，SimpleDateFormat 是线程安全的吗?

不是，非常不幸，DateFormat 的所有实现，包括 SimpleDateFormat 都不是线程安全的，因此你不应该在多线程序中使用，除非是在对外线程安全的环境中使用，**如将 SimpleDateFormat 限制在 ThreadLocal 中。**

如果你不这么做，在解析或者格式化日期的时候，可能会获取到一个不正确的结果。因此，从日期、时间处理的所有实践来说，我强力推荐 joda-time 库。

如何在 Java 中获取线程堆栈?

- `kill -3 [java pid]`

  不会在当前终端输出，它会输出到代码执行的或指定的地方去。比如， `kill -3 tomcat pid` ，输出堆栈到 log 目录下。

- `Jstack [java pid]`

  这个比较简单，在当前终端显示，也可以重定向到指定文件中。

- `JVisualVM: Thread Dump`

  不做说明，打开 JVisualVM 后，都是界面操作，过程还是很简单的。

什么是Java Timer 类?

`java.util.Timer`，是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer 类可以用安排一次性任务或者周期任务。

`java.util.TimerTask`，是一个实现了 Runnable 接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用 Timer 去安排它的执行。

目前有开源的 Qurtz 可以用来创建定时任务。

你有哪些多线程开发良好的实践?

- 1、给线程命名。

  > 这样可以方便找 bug 或追踪。OrderProcessor、QuoteProcessor、TradeProcessor 这种名字比 Thread-1、Thread-2、Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。

- 2、最小化同步范围。

  > 锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。

- 3、优先使用 `volatile`，而不是 `synchronized`。
- 4、尽可能使用更高层次的并发工具而非 wait 和 notify 方法来实现线程通信。

  > 首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用 wait 和 notify 很难实现对复杂控制流的控制。
  >
  > 其次，这些类是由最好的企业编写和维护在后续的 JDK 中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。

- 5、优先使用并发容器，而非同步容器。

  > 这是另外一个容易遵循且受益巨大的最佳实践，并发容器比同步容器的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到 Map，我们应该首先想到用 ConcurrentHashMap 类。

- 6、考虑使用线程池。

并发编程和并行编程有什么区别?

并发（Concurrency）和并行（Parallellism）是:

- 解释一：并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。
- 解释二：并行是在不同实体上的多个事件；并发是在同一实体上的多个事件。
- 解释三：在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如 Hadoop 分布式集群。

所以并发编程的目标是，充分的利用处理器的每一个核，以达到最高的处理性能。

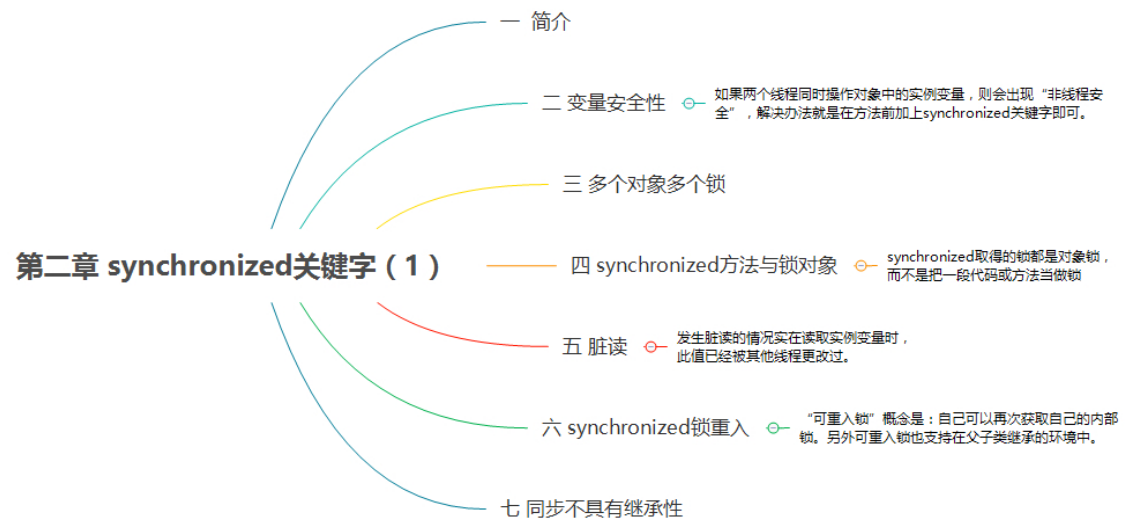
同步和异步有何异同，在什么情况下分别使用他们?

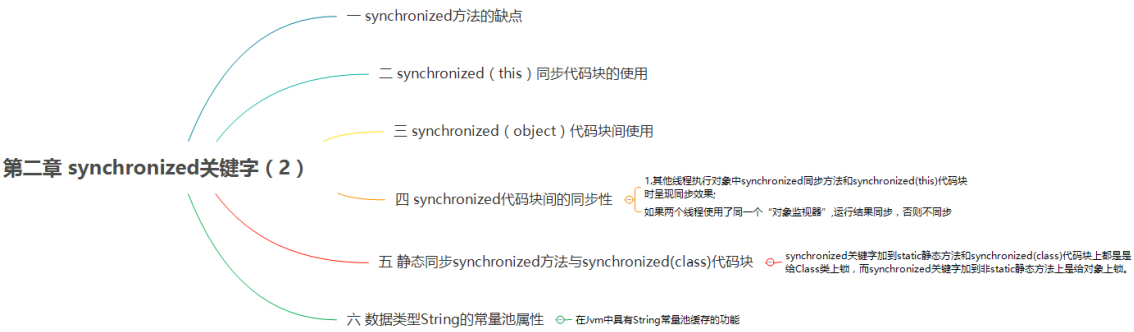
如果数据将在线程间共享。例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就是共享数据，必须进行**同步**存取。

当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用**异步**编程，在很多情况下采用异步途径往往更有效率。

> 当然，如果我们对效率没有特别大的要求，也不一定需要使用异步编程，因为它会带来编码的复杂性。总之，合适才是正确的。

Java 锁

- `synchronized`
  - 

  - 

- `volatile`

`synchronized` 的原理是什么?

`synchronized` 是 Java 内置的关键字，它提供了一种独占的加锁方式。

- `synchronized` 的获取和释放锁由JVM实现，用户不需要显示的释放锁，非常方便。
- 然而，`synchronized` 也有一定的局限性。
  - 当线程尝试获取锁的时候，如果获取不到锁会一直阻塞。
  - 如果获取锁的线程进入休眠或者阻塞，除非当前线程异常，否则其他线程尝试获取锁必须一直等待。

关于原理，直接阅读 《【死磕 Java 并发】—— 深入分析 synchronized 的实现原理》 文章，有几个重点要注意看。

- 实现原理
- Java 对象头、Monitor
- 锁优化
  - 自旋锁
    - 适应自旋锁
  - 锁消除
  - 锁粗化
  - 锁的升级
    - 重量级锁
    - 轻量级锁
    - 偏向锁

📣 当一个线程进入某个对象的一个 synchronized 的实例方法后，其它线程是否可进入此对象的其它方法?

- 如果其他方法没有 `synchronized` 的话，其他线程是可以进入的。
- 所以要开放一个线程安全的对象时，得保证每个方法都是线程安全的。

📣 同步方法和同步块，哪个是更好的选择?

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

📣 在监视器(Monitor)内部，是如何做线程同步的?

监视器和锁在 Java 虚拟机中是一块使用的。监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块。每一个监视器都和一个对象引用相关联。**线程在获取锁之前不允许执行同步代码。**

📣 Java 如何实现“自旋”（spin）

参考 《Java 锁的种类以及辨析（一）：自旋锁》

代码如下：

```
public class SpinLock {

    private AtomicReference<Thread> sign =new AtomicReference<>();

    public void lock() { // <1>
        Thread current = Thread.currentThread();
        while(!sign .compareAndSet(null, current)) {
            // <1.1>
        }
    }

    public void unlock () { // <2>
        Thread current = Thread.currentThread();
        sign .compareAndSet(current, null);
    }

}
```

- `<1>` 处，`#lock()` 方法，如果获得不到锁，就会“死循环”，直到或得到锁为止。考虑到“死循环”会持续占用 CPU ，可能导致其它线程无法获得到 CPU 执行，可以在 `<1.1>` 处增加 `Thread.yiead()` 代码段，出让下 CPU 。
- `<2>` 处，`#unlock()` 方法，释放锁。

volatile 实现原理

`volatile` 涉及的内容，其实蛮多的，所以胖友直接看：

- 《【死磕 Java 并发】—– 深入分析 volatile 的实现原理》
- 聊聊并发（一）——深入分析Volatile的实现原理

📣 volatile 有什么用?

`volatile` 保证内存可见性和禁止指令重排。

> 同时，`volatile` 可以提供部分原子性。

简单来说，`volatile` 用于多线程环境下的单次操作(单次读或者单次写)。

📣 volatile 变量和 atomic 变量有什么不同?

- `volatile` 变量，可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。例如用 `volatile` 修饰 `count` 变量，那么 `count++` 操作就不是原子性的。
- AtomicInteger 类提供的 atomic 方法，可以让这种操作具有原子性。例如 `#getAndIncrement()` 方法，会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

📣 可以创建 volatile 数组吗?

Java 中可以创建 `volatile` 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到 `volatile` 的保护，但是如果多个线程同时改变数组的元素，`volatile` 标示符就不能起到之前的保护作用了。

同理，对于 Java POJO 类，使用 `volatile` 修饰，只能保证这个引用的可见性，不能保证其内部的属性。

📣 volatile 能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 `long` 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 `volatile` 。为什么？因为 Java 中读取 `long` 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 `long` 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 `volatile` 型的 `long` 或 `double` 变量的读写是原子。

> 如下的内容，可以作为上面的内容的补充。
>
> 一种实践是用 `volatile` 修饰 `long` 和 `double` 变量，使其能按原子类型来读写。`double` 和 `long` 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 `volatile` 型的 `long` 或 `double` 变量的读写是原子的。

📣 volatile 类型变量提供什么保证?

`volatile` 主要有两方面的作用：

1. 避免指令重排
2. 可见性保证

例如，JVM 或者 JIT 为了获得更好的性能会对语句重排序，但是 `volatile` 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。

- `volatile` 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。
- 某些情况下，`volatile` 还能提供原子性，如读 64 位数据类型，像 `long` 和 `double` 都不是原子的(低 32 位和高 32 位)，但 `volatile` 类型的 `double` 和 `long` 就是原子的。**不过需要在 64 位的 JVM 虚拟机上。**详细的分析，可以看看 《Java中 long 和 double 的原子性》 。

📣 volatile 和 synchronized 的区别?

1. `volatile` 本质是在告诉 JVM 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取。`synchronized` 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. `volatile` 仅能使用在变量级别。`synchronized` 则可以使用在变量、方法、和类级别的。
3. `volatile` 仅能实现变量的修改可见性，不能保证原子性。而 `synchronized` 则可以保证变量的修改可见性和原子性。
4. `volatile` 不会造成线程的阻塞。`synchronized` 可能会造成线程的阻塞。
5. `volatile` 标记的变量不会被编译器优化。`synchronized` 标记的变量可以被编译器优化。

> 另外，会有面试官会问 `volatile` 能否取代 `synchronized` 呢？答案肯定是不能，虽然说 `volatile` 被称之为轻量级锁，但是和 `synchronized` 是有本质上的区别，原因就是上面的几点落。

📣 什么场景下可以使用 volatile 替换 synchronized ?

1. 只需要保证共享资源的可见性的时候可以使用 `volatile` 替代，`synchronized` 保证可操作的原子性一致性和可见性。
2. `volatile` 适用于新值不依赖于旧值的情形。
3. 1 写 N 读。

4. 不与其他变量构成不变性条件时候使用 `volatile` 。

什么是死锁、活锁?

死锁，是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

产生死锁的必要条件：

- 互斥条件：所谓互斥就是进程在某一时间内独占资源。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：进程已获得资源，在未使用完之前，不能强行剥夺。
- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

死锁的解决方法：

- 撤消陷于死锁的全部进程。
- 逐个撤消陷于死锁的进程，直到死锁不存在。
- 从陷于死锁的进程中逐个强迫放弃所占用的资源，直至死锁消失。
- 从另外一些进程那里强行剥夺足够数量的资源分配给死锁进程，以解除死锁状态。

📣 什么是活锁?

活锁，任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

📣 死锁与活锁的区别?

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”，而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

实际上，聪慧的胖友是不是已经发现，死锁就是悲观锁可能产生的结果，而活锁是乐观锁可能产生的结果。

什么是悲观锁、乐观锁?

1）悲观锁

悲观锁，总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。

- 传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。
- 再比如 Java 里面的同步原语 `synchronized` 关键字的实现也是悲观锁。

2）乐观锁

乐观锁，顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量。

- 像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。

  > 例如，version 字段（比较跟上一次的版本号，如果一样则更新，如果失败则要重复读-比较-写的操作）

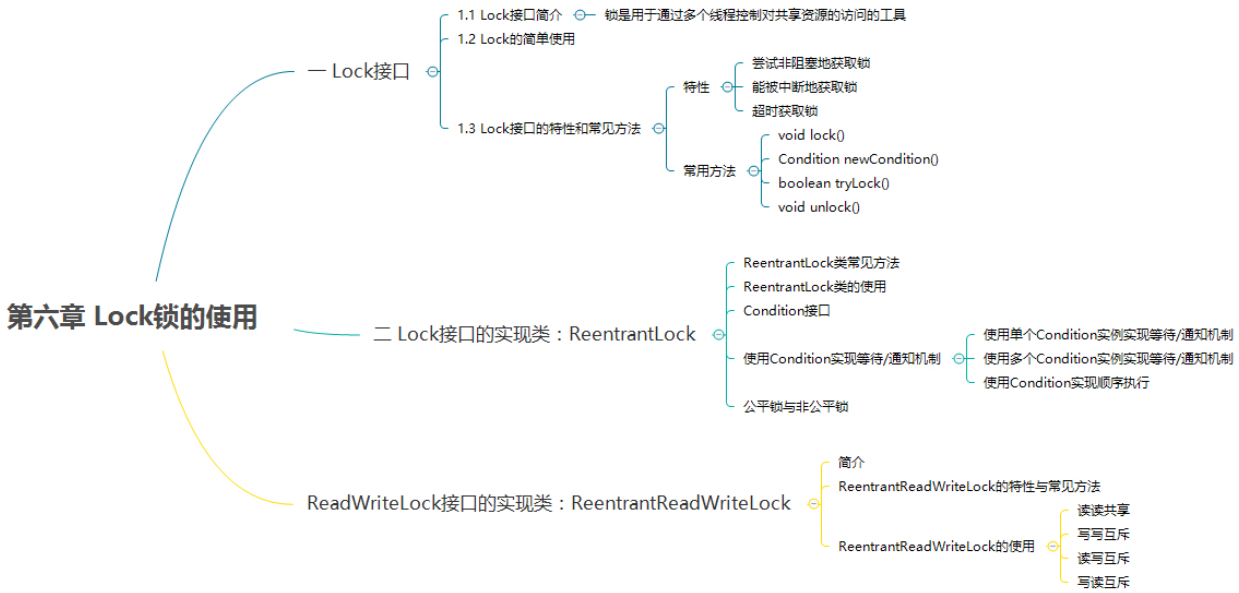
- 在 Java 中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

乐观锁的实现方式：

- 使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。
- Java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

Java Lock 接口

> ：虽然 Lock 也翻译成锁，但是和上面的「Java 锁」分开，它更多强调的是 `synchronized` 和 `volatile` 关键字带来的重量级和轻量级锁。而 Lock 是 Java 锁接口，提供了更多灵活的功能。



Java AQS

`java.util.concurrent.locks.AbstractQueuedSynchronizer` 抽象类，简称 AQS ，是一个用于构建锁和同步容器的同步器。事实上 `concurrent` 包内许多类都是基于 AQS 构建。例如 ReentrantLock，Semaphore，CountDownLatch，ReentrantReadWriteLock，等。AQS 解决了在实现同步容器时设计的大量细节问题。

AQS 使用一个 FIFO 的队列表示排队等待锁的线程，队列头节点称作“哨兵节点”或者“哑节点”，它不与任何线程关联。其他的节点与等待线程关联，每个节点维护一个等待状态 `waitStatus` 。

可能这么说，胖友会一脸懵逼，最好的方式，还是直接去撸源码，可见如下的四篇文章。

> 可能胖友在阅读时，会有一定的挫败感，没关系，大家都是如此，包括，还有我认识的各种大佬。

- 《【死磕 Java 并发】—— J.U.C 之 AQS：AQS 简介》
- 《【死磕 Java 并发】—— J.U.C 之 AQS：CLH 同步队列》
- 《【死磕 Java 并发】—— J.U.C 之 AQS：同步状态的获取与释放》
- 《【死磕 Java 并发】—— J.U.C 之 AQS：阻塞和唤醒线程》

什么是 Java Lock 接口?

`java.util.concurrent.locks.Lock` 接口，比 `synchronized` 提供更具拓展行的锁操作。它允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。它的优势有：

- 可以使锁更公平。
- 可以使线程在等待锁的时候响应中断。
- 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间。
- 可以在不同的范围，以不同的顺序获取和释放锁。

什么是可重入锁（ReentrantLock）？

举例来说明锁的可重入性。代码如下：

```
public class UnReentrant{

    Lock lock = new Lock();

    public void outer() {
        lock.lock();
        inner();
        lock.unlock();
    }

    public void inner() {
        lock.lock();
        //do something
        lock.unlock();
    }

}
```

- `#outer()` 方法中调用了 `#inner()` 方法，`#outer()` 方法先锁住了 `lock` ，这样 `#inner()` 就不能再获取 `lock` 。
- 其实调用 `#outer()` 方法的线程已经获取了 `lock` 锁，但是不能在 `#inner()` 方法中重复利用已经获取的锁资源，这种锁即称之为不可重入。
- 可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

`synchronized`、ReentrantLock 都是可重入的锁，可重入锁相对来说简化了并发编程的开发。

关于 ReentrantLock 类，详细的源码解析，可以看看 《【死磕 Java 并发】—– J.U.C 之重入锁：ReentrantLock》 。

> 简单来说，ReenTrantLock 的实现是一种自旋锁，通过循环调用 CAS 操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。

synchronized 和 ReentrantLock 异同?

- 相同点
  - 都实现了多线程同步和内存可见性语义。
  - 都是可重入锁。
- 不同点

- 同步实现机制不同
  - `synchronized` 通过 Java 对象头锁标记和 Monitor 对象实现同步。
  - ReentrantLock 通过CAS、AQS（AbstractQueuedSynchronizer）和 LockSupport（用于阻塞和解除阻塞）实现同步。 *
- 可见性实现机制不同
  - `synchronized` 依赖 JVM 内存模型保证包含共享变量的多线程内存可见性。
  - ReentrantLock 通过 ASQ 的 `volatile state` 保证包含共享变量的多线程内存可见性。
- 使用方式不同
  - `synchronized` 可以修饰实例方法（锁住实例对象）、静态方法（锁住类对象）、代码块（显示指定锁对象）。
  - ReentrantLock 显示调用 tryLock 和 lock 方法，需要在 `finally` 块中释放锁。
- 功能丰富程度不同
  - `synchronized` 不可设置等待时间、不可被中断（interrupted）。
  - ReentrantLock 提供有限时间等候锁（设置过期时间）、可中断锁（lockInterruptibly）、condition（提供 await、condition（提供 await、signal 等方法）等丰富功能
- 锁类型不同
  - `synchronized` 只支持非公平锁。
  - ReentrantLock 提供公平锁和非公平锁实现。当然，在大部分情况下，非公平锁是高效的选择。

> 在 `synchronized` 优化以前，它的性能是比 ReenTrantLock 差很多的，但是自从 `synchronized` 引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用 `synchronized` 。
>
> 并且，实际代码实战中，可能的优化场景是，通过读写分离，进一步性能的提升，所以使用 ReentrantReadWriteLock 。😁

ReadWriteLock 是什么?

ReadWriteLock ，读写锁是，用来提升并发程序性能的锁分离技术的 Lock 实现类。可以用于 “多读少写” 的场景，读写锁支持多个读操作并发执行，写操作只能由一个线程来操作。

ReadWriteLock 对向数据结构相对不频繁地写入，但是有多个任务要经常读取这个数据结构的这类情况进行了优化。ReadWriteLock 使得你可以同时有多个读取者，只要它们都不试图写入即可。如果写锁已经被其他任务持有，那么任何读取者都不能访问，直至这个写锁被释放为止。

ReadWriteLock 对程序性能的提高主要受制于如下几个因素：

1. 数据被读取的频率与被修改的频率相比较的结果。
2. 读取和写入的时间
3. 有多少线程竞争
4. 是否在多处理机器上运行

ReadWriteLock 的源码解析，可以看看 《【死磕 Java 并发】—– J.U.C 之读写锁：ReentrantReadWriteLock》 。

Condition 是什么?

在没有 Lock 之前，我们使用 `synchronized` 来控制同步，配合 Object 的 `#wait()`、`#notify()` 等一系列方法可以实现**等待 / 通知模式**。在 Java SE 5 后，Java 提供了 Lock 接口，相对于 `synchronized` 而言，Lock 提供了条件 Condition ，对线程的等待、唤醒操作更加详细和灵活。下图是 Condition 与 Object 的监视器方法的对比（摘自《Java并发编程的艺术》）：

对比项	Object监视器方法	Condition
前置条件	获取对象的锁	调用Lock.lock()获取锁 调用Lock.newCondtion()获取Condition对象
调用方式	直接调用	直接调用
等待队列个数	一个	多个
当前线程释放锁并进入等待状态	支持	支持
当前线程释放锁并进入等待状态，在等待状态中不响应中断	不支持	支持
当前线程释放锁并进入超时等待状态	支持	支持
当前线程释放锁并进入从等待状态到将来的某个时间	不支持	支持
唤醒等待队列中的某个线程	支持	支持
唤醒等待队列中的全部线程	支持	支持

- Condition 的使用，可以看看 《怎么理解 Condition》
- Condition 的源码，可以看看 《【死磕 Java 并发】—– J.U.C 之 Condition》 。

用三个线程按顺序循环打印 abc 三个字母，比如 abcabcabc ?

- 使用 Lock + Condition 来实现。具体代码，参看 《用三个线程按顺序循环打印 abc 三个字母，比如 abcabcabc》 。
- 使用 `synchronized` + await/notifyAll 来实现，参看 《Java用三个线程按顺序循环打印 abc 三个字母,比如 abcabcabc》 。

LockSupport 是什么?

LockSupport 是 JDK 中比较底层的类，用来创建锁和其他同步工具类的基本线程阻塞。

- Java 锁和同步器框架的核心 AQS(AbstractQueuedSynchronizer)，就是通过调用 `LockSupport#park()` 和 `LockSupport#unpark()` 方法，来实现线程的阻塞和唤醒的。
- LockSupport 很类似于二元信号量(只有 1 个许可证可供使用)，如果这个许可还没有被占用，当前线程获取许可并继续执行；如果许可已经被占用，当前线程阻塞，等待获取许可。

对于 LockSupport 了解即可，面试一般问的不多。感兴趣的胖友，可以看看如下文章：

- 《多线程同步工具 —— LockSupport》
- 《Java 并发编程 —— LockSupport》 带部分源码解析。

Java 内存模型

关于 Java 内存模型，涉及的内容会很多，所以建议胖友看如下的 《深入Java内存模型.pdf》 这本小书。

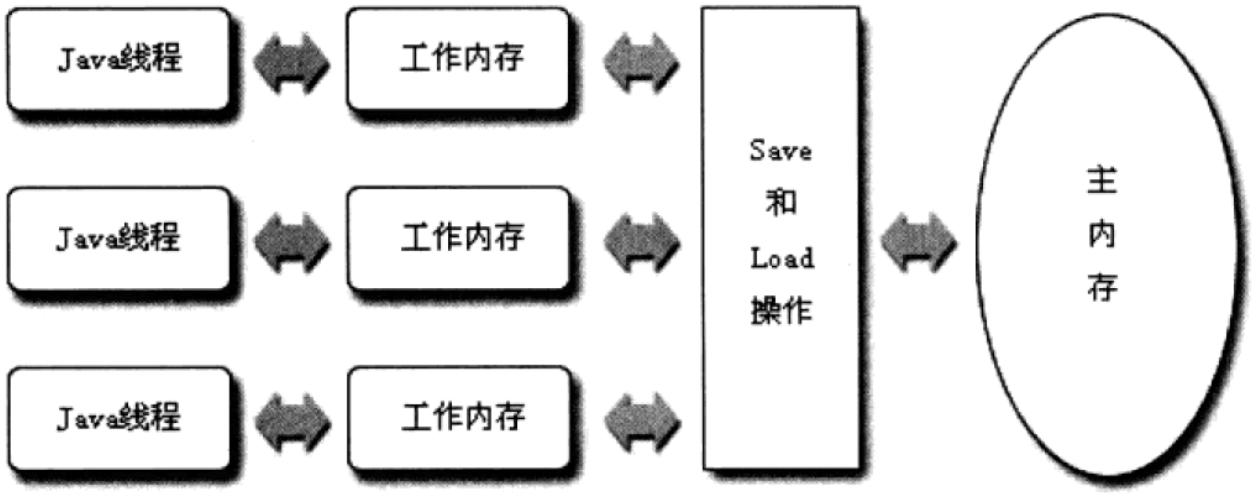
然后，看完之后你肯定会忘记，就可以靠 《《深入理解 Java 内存模型》读书笔记》 来补刀。

再另外，《深入拆解 Java 虚拟机》 的 「第五部分 高效并发」 也推荐阅读。

什么是 Java 内存模型?

Java 虚拟机规范中试图定义一种 Java 内存模型（Java Memory Model，JMM）来屏蔽掉各层硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。

Java 内存模型规定了所有的变量都存储在主内存（Main Memory）中。每条线程还有自己的工作内存（Working Memory），线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在主内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间的变量值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的关系如下图：



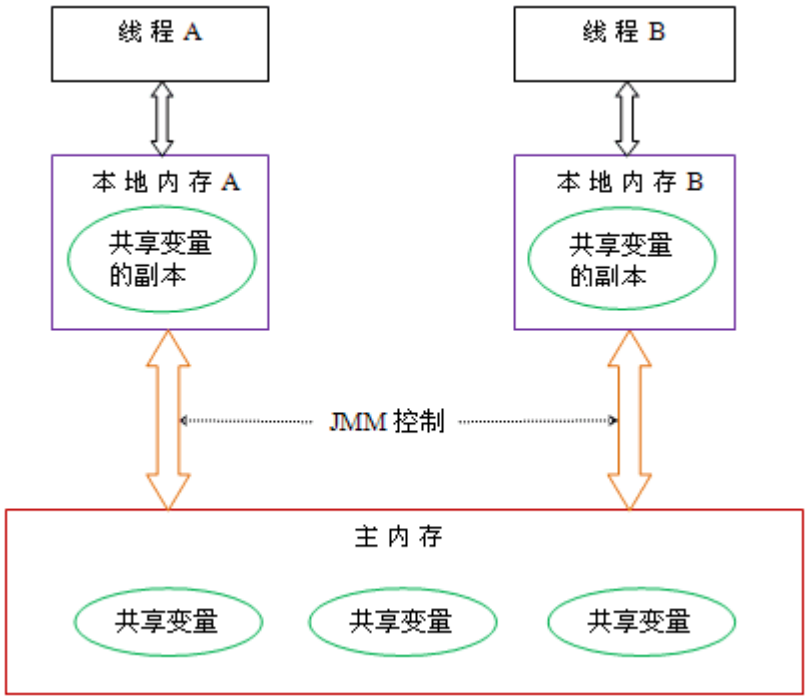
> ：当然，有个面试官会把 Java 内存模型，和 JVM 内存结构搞混淆。所以，在回答之前，可以先和面试官确认下说的是哪个。
>
> 关于 JVM 内存结构的面试题，我们在 《精尽 Java【虚拟机】面试题》 中在详细分享。

两个线程之间是如何通信的呢?

线程之间的通信方式，目前有共享内存和消息传递两种。

1）共享内存

在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。典型的共享内存通信方式，就是通过共享对象进行通信。

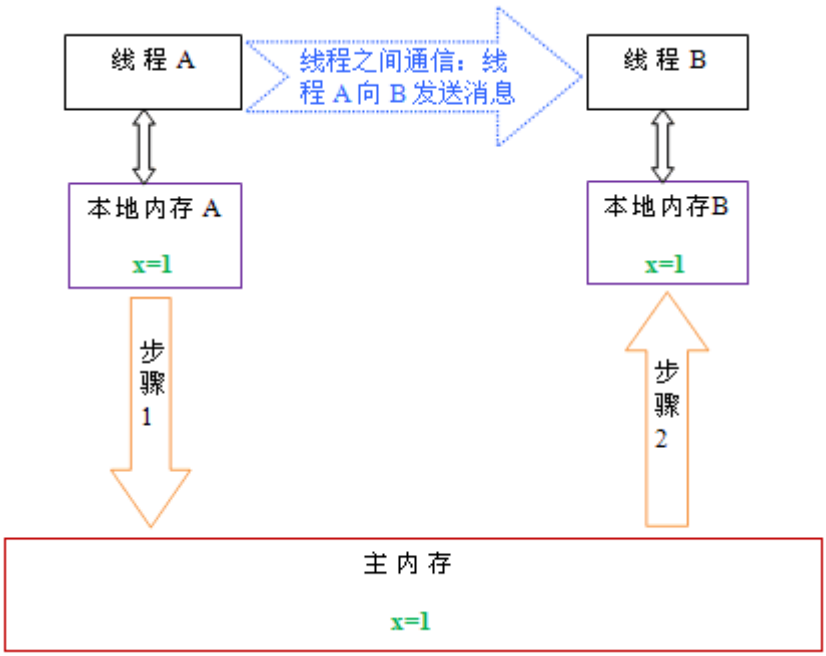


例如上图线程 A 与 线程 B 之间如果要通信的话，那么就必须经历下面两个步骤：

1. 首先，线程 A 把本地内存 A 更新过得共享变量刷新到主内存中去。
2. 然后，线程 B 到主内存中去读取线程 A 之前更新过的共享变量。

2）消息传递

在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。在 Java 中典型的消息传递方式，就是 `#wait()` 和 `#notify()` ，或者 BlockingQueue 。



为什么代码会重排序?

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

- 在单线程环境下不能改变程序运行的结果。
- 存在数据依赖关系的不允许重排序

需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

什么是内存模型的 happens-before 呢?

详细看 《【死磕 Java 并发】—— Java 内存模型之 happens-before》 文章。

什么是内存屏障?

内存屏障，又称内存栅栏，是一组处理器指令，用于实现对内存操作的顺序限制。

内存屏障为何重要?

对主存的一次访问一般花费硬件的数百次时钟周期。处理器通过缓存（caching）能够从数量级上降低内存延迟的成本这些缓存为了性能重新排列待定内存操作的顺序。也就是说，程序的读写操作不一定会按照它要求处理器的顺序执行。当数据是不可变的，同时/或者数据限制在线程范围内，这些优化是无害的。如果把这些优化与对称多处理（symmetric multi-processing）和共享可变状态（shared mutable state）结合，那么就是一场噩梦。

当基于共享可变状态的内存操作被重新排序时，程序可能行为不定。一个线程写入的数据可能被其他线程可见，原因是数据写入的顺序不一致。适当的放置内存屏障，通过强制处理器顺序执行待定的内存操作来避免这个问题。

Java 并发容器

什么是并发容器的实现?

何为同步容器？可以简单地理解为通过 `synchronized` 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。

- 比如 Vector，Hashtable，以及 `Collections#synchronizedSet()`，`Collections#synchronizedList()` 等方法返回的容器。
- 可以通过查看 Vector，Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 `synchronized` 。

并发容器，使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性。

- 例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制，可以称为分段锁。在这种锁机制下，允许任意数量的读线程并发地访问 map ，并且执行读操作的线程和写操作的线程也可以并发的访问 map ，同时允许一定数量的写操作线程并发地修改 map ，所以它可以在并发环境下实现更高的吞吐量。
- 再例如，CopyOnWriteArrayList 。

SynchronizedMap 和 ConcurrentHashMap 有什么区别?

- SynchronizedMap
  - 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map 。
- ConcurrentHashMap

    - 使用分段锁来保证在多线程下的性能。ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get,put,remove 等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。【注意，这块是 JDK7 的实现。在 JDK8 中，具体的实现已经改变】
    - 另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，当 iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException 异常，取而代之的是在改变时 `new` 新的数据从而不影响原有的数据，iterator 完成后再将头指针替换为新的数据 ，这样 iterator 线程可以使用原来老的数据，而写线程也可以并发的完成改变。

关于 ConcurrentHashMap 的源码解析，推荐胖友看看如下两篇文章：

- 《【死磕 Java 并发】—— J.U.C 之 Java并发容器：ConcurrentHashMap》
- 《【死磕 Java 并发】—— J.U.C 之 ConcurrentHashMap 红黑树转换分析》

Java 中 ConcurrentHashMap 的并发度是什么?

在 JDK8 前，ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是 ConcurrentHashMap 类构造函数的一个可选参数，默认值为 16 ，这样在多线程情况下就能避免争用。

在 JDK8 后，它摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现，利用 CAS 算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

ConcurrentHashMap 为何读不用加锁?

在 JDK7 以及以前

- HashEntry 中的 `key` 、 `hash` 、 `next` 均为 `final` 型，只能表头插入、删除结点。
    - HashEntry 类的 `value` 域被声明为 `volatile` 型。
    - 不允许用 `null` 作为键和值，当读线程读到某个 HashEntry 的 `value` 域的值为 `null` 时，便知道产生了冲突——发生了重排序现象（put 方法设置新 `value` 对象的字节码指令重排序），需要加锁后重新读入这个 `value` 值。
- `volatile` 变量 `count` 协调读写线程之间的内存可见性，写操作后修改 `count` ，读操作先读 `count` ，根据 happen-before 传递性原则写操作的修改读操作能够看到。

在 JDK8 开始

- Node 的 `val` 和 `next` 均为 `volatile` 型。
- `#tabAt(..,)` 和 `#casTabAt(...)` 对应的 Unsafe 操作实现了 `volatile` 语义。

CopyOnWriteArrayList 可以用于什么应用场景?

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException 异常。在 CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

- 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 ygc 或者 fgc 。
- 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的,虽然 CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求。

CopyOnWriteArrayList 透露的思想：

- 读写分离，读和写分开
- 最终一致性
- 使用另外开辟空间的思路，来解决并发冲突

CopyOnWriteArrayList 适用于读操作远远多于写操作的场景。例如，缓存。

关于 CopyOnWriteArrayList 的源码，可以看看 《CopyOnWriteArrayList 实现原理及源码分析》 文章。

Java 阻塞队列

什么是阻塞队列？有什么适用场景？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：

- 在队列为空时，获取元素的线程会等待队列变为非空。
- 当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景：

- 生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程
- 阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

> ：如下的内容，和上面是相对重复的，或者是换一个说法，重新描述。

BlockingQueue 接口，是 Queue 的子接口，它的主要用途并不是作为容器，而是作为线程同步的的工具，因此他具有一个很明显的特性：

- 当生产者线程试图向 BlockingQueue 放入元素时，如果队列已满，则线程被阻塞。
- 当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞。
- 正是因为它所具有这个特性，所以在程序中多个线程交替向BlockingQueue中 放入元素，取出元素，它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景，就是 Socket 客户端数据的读取和解析：

- 读取数据的线程不断将数据放入队列。
- 然后，解析线程不断从队列取数据解析。

Java 提供了哪些阻塞队列的实现？

JDK7 提供了 7 个阻塞队列。分别是：

> Java5 之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好 wait、notify、notifyAll、`sychronized` 这些关键字。
>
> 而在 Java5 之后，可以使用阻塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。

- 【最常用】ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列。

  > 此队列按照先进先出（FIFO）的原则对元素进行排序，但是默认情况下不保证线程公平的访问队列，即如果队列满了，那么被阻塞在外面的线程对队列访问的顺序是不能保证线程公平（即先阻塞，先插入）的。

- LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列。

> 此队列按照先出先进的原则对元素进行排序

- PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
- DelayQueue：支持延时获取元素的无界阻塞队列，即可以指定多久才能从队列中获取当前元素。
- SynchronousQueue：一个不存储元素的阻塞队列。

> 每一个 put 必须等待一个 take 操作，否则不能继续添加元素。并且他支持公平访问队列。

- LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

> 相对于其他阻塞队列，多了 tryTransfer 和 transfer 方法。
>
> - transfer 方法：如果当前有消费者正在等待接收元素（take 或者待时间限制的 poll 方法），transfer 可以把生产者传入的元素立刻传给消费者。如果没有消费者等待接收元素，则将元素放在队列的 tail 节点，并等到该元素被消费者消费了才返回。
> - tryTransfer 方法：用来试探生产者传入的元素能否直接传给消费者。如果没有消费者在等待，则返回 false 。和上述方法的区别是该方法无论消费者是否接收，方法立即返回。而 transfer 方法是必须等到消费者消费了才返回。

- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

> 优势在于多线程入队时，减少一半的竞争。

具体的源码解析，可以看看如下文章：

- 《【死磕 Java 并发】—— J.U.C 之阻塞队列：ArrayBlockingQueue》
- 《【死磕 Java 并发】—— J.U.C 之阻塞队列：PriorityBlockingQueue》
- 《【死磕 Java 并发】—— J.U.C 之阻塞队列：DelayQueue》
- 《【死磕 Java 并发】—— J.U.C 之阻塞队列：SynchronousQueue》
- 《【死磕 Java 并发】—— J.U.C 之阻塞队列：LinkedTransferQueue》
- 《【死磕 Java 并发】—— J.U.C 之阻塞队列：LinkedBlockingDeque》
- 《【死磕 Java 并发】—— J.U.C 之阻塞队列：BlockingQueue 总结》

📣 阻塞队列提供哪些重要方法?

方法处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e, time, unit)
移除方法	remove()	poll()	take()	poll(time, unit)
检查方法	element()	peek()	不可用	不可用

📣 ArrayBlockingQueue 与 LinkedBlockingQueue 的区别?

Queue	阻塞与否	是否有界	线程安全保障	适用场景	注意事项
ArrayBlockingQueue	阻塞	有界	一把全局锁	生产消费模型，平衡两边处理速度	用于存储队列元素的存储空间是预先分配的，使用过程中内存开销较小（无须动态申请存储空间）
LinkedBlockingQueue	阻塞	可配置	存取采用2把锁	生产消费模型，平衡两边处理速度	无界的时候注意内存溢出问题，用于存储队列元素的存储空间是在其使用过程中动态分配的，因此它可能会增加 JVM 垃圾回收的负担。

感兴趣的胖友，可以看看如下两篇文章：

- 《ArrayBlockingQueue 与 LinkedBlockingQueue》
- 《从一个故障说说 Java 的三个 BlockingQueue》

什么是双端队列?

在上面，我们看到的 LinkedBlockingQueue、ArrayBlockingQueue、PriorityBlockingQueue、SynchronousQueue 等，都是阻塞队列。

而 ArrayDeque、LinkedBlockingDeque 就是双端队列，类名以 Deque 结尾。

- 正如阻塞队列适用于生产者消费者模式，双端队列同样适用与另一种模式，即

  工作密取

  。在生产者-消费者设计中，所有消费者共享一个工作队列，而在工作密取中，每个消费者都有各自的双端队列。
  - 如果一个消费者完成了自己双端队列中的全部工作，那么他就可以从其他消费者的双端队列末尾秘密的获取工作。具有更好的可伸缩性，这是因为工作者线程不会在单个共享的任务队列上发生竞争。
  - 在大多数时候，他们都只是访问自己的双端队列，从而极大的减少了竞争。当工作者线程需要访问另一个队列时，它会从队列的尾部而不是头部获取工作，因此进一步降低了队列上的竞争。
- 适用于：网页爬虫等任务中

> 实际场景下，双端队列，我们使用比较少。根本没用过。

延迟队列的实现方式，DelayQueue 和时间轮算法的异同?

JDK 的 Timer 和 DelayQueue 插入和删除操作的平均时间复杂度为 `O(nlog(n))` ，而基于时间轮可以将插入和删除操作的时间复杂度都降为 `O(1)` 。

- 关于 DelayQueue 的实现方式，在 《【死磕 Java 并发】—– J.U.C 之阻塞队列：DelayQueue》》 。
- 关于实践论的实现方法，在 《Kafka解惑之时间轮（TimingWheel）》 。

简述 ConcurrentLinkedQueue 和 LinkedBlockingQueue 的用处和不同之处?

参考 《LinkedBlockingQueue 和 ConcurrentLinkedQueue的用法及区别》 。

在 Java 多线程应用中，队列的使用率很高，多数生产消费模型的首选数据结构就是队列(先进先出)。

Java 提供的线程安全的 Queue 可以分为

- 阻塞队列，典型例子是 LinkedBlockingQueue 。

  > 适用阻塞队列的好处：多线程操作共同的队列时不需要额外的同步，另外就是队列会自动平衡负载，即那边（生产与消费两边）处理快了就会被阻塞掉，从而减少两边的处理速度差距。

- 非阻塞队列，典型例子是 ConcurrentLinkedQueue 。

  > 当许多线程共享访问一个公共集合时，`ConcurrentLinkedQueue` 是一个恰当的选择。

具体的选择，如下：

- LinkedBlockingQueue 多用于任务队列。
  - 单生产者，单消费者
  - 多生产者，单消费者
- ConcurrentLinkedQueue 多用于消息队列。
  - 单生产者，多消费者
  - 多生产者，多消费者

Java 原子操作类

什么是原子操作?

原子操作（Atomic Operation），意为”不可被中断的一个或一系列操作”。

- 处理器使用基于对缓存加锁或总线加锁的方式，来实现多处理器之间的原子操作。
- 在 Java 中，可以通过锁和循环 CAS 的方式来实现原子操作。CAS操作 —— Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作。

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

- `int++` 并不是一个原子操作，所以当一个线程读取它的值并加 1 时，另外一个线程有可能会读到之前的值，这就会引发错误。
- 为了解决这个问题，必须保证增加操作是原子的，在 JDK5 之前我们可以使用同步技术来做到这一点。到 JDK5 后，`java.util.concurrent.atomic` 包提供了 `int` 和 `long` 类型的原子包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

`java.util.concurrent` 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。

- 原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference 。
- 原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray 。
- 原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater 。
- 解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个 `boolean` 来反映中间有没有变过），AtomicStampedReference（通过引入一个 `int` 来累加来反映中间有没有变过）。

关于 CAS 的内容，建议胖友在看看 《【死磕 Java 并发】—- 深入分析 CAS》 。

CAS 操作有什么缺点?

1）**ABA 问题**

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。

从 Java5 开始 JDK 的 `atomic` 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

2）**循环时间长开销大**

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 `synchronized` 。

3）**只能保证一个共享变量的原子操作**

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

Java 并发工具类

Semaphore 是什么?

Semaphore，是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。

- 如有必要，在许可可用前会阻塞每一个 `#acquire()` 方法，然后再获取该许可。
- 每个 `#release()` 方法，添加一个许可，从而可能释放一个正在阻塞的获取者。
- 但是，不使用实际的许可对象，Semaphore 只对可用许可的数量进行计数，并采取相应的行动。

信号量常常用于多线程的代码中，比如数据库连接池。

- 使用方式，可以看看 《JAVA多线程 – 信号量(Semaphore)》 。
- 源码解析，可以看看 《【死磕 Java 并发】—– J.U.C 之并发工具类：Semaphore》 。

说说 CountDownLatch 原理

CountDownLatch，字面意思是减小计数（CountDown）的门闩（Latch）。它要做的事情是，等待指定数量的计数被减少，意味着门闩被打开，然后进行执行。

CountDownLatch 默认的构造方法是 `CountDownLatch(int count)` ，其参数表示需要减少的计数，主线程调用 `#await()` 方法告诉 CountDownLatch 阻塞等待指定数量的计数被减少，然后其它线程调用 CountDownLatch 的 `#countDown()` 方法，减小计数(不会阻塞)。等待计数被减少到零，主线程结束阻塞等待，继续往下执行。

- CountDownLatch 的使用示例，请看《Java 多线程 CountDownLatch 用法》。
- CountDownLatch 的源码解析，请看《【死磕 Java 并发】—– J.U.C 之并发工具类：CountDownLatch》

说说 CyclicBarrier 原理

CyclicBarrier，字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。

CyclicBarrier 默认的构造方法是 `CyclicBarrier(int parties)`，其参数表示屏障拦截的线程数量，每个线程调用 `#await()` 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞，直到 `parties` 个线程到达，结束阻塞。

- CyclicBarrier 的使用示例，请看《CyclicBarrier 的用法》
- CyclicBarrier 的源码解析，请看《【死磕 Java 并发】—- J.U.C 之并发工具类：CyclicBarrier》。

说说 Exchanger 原理

实际场景下，问了一圈朋友，Exchanger 基本没在业务中使用过。

- Exchanger 的使用示例，请看《【Java并发】线程同步工具Exchanger的使用》。
- Exchanger 的源码解析，请看《【死磕 Java 并发】—– J.U.C 之并发工具类：Exchanger》

CyclicBarrier 和 CountdownLatch 有什么区别?

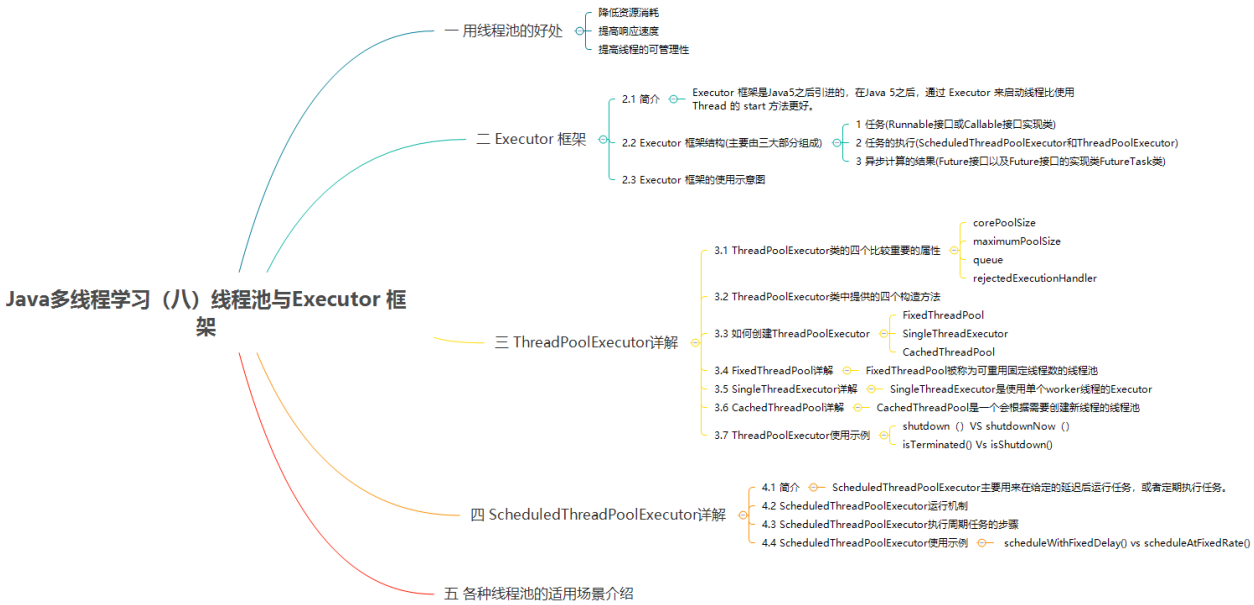
CyclicBarrier 可以重复使用，而 CountdownLatch 不能重复使用。

- CountDownLatch 其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作。
    - 你可以向 CountDownLatch 对象设置一个初始的数字作为计数值，任何调用这个对象上的 `#await()` 方法都会阻塞，直到这个计数器的计数值被其他的线程减为 0 为止。所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier 。
    - CountDownLatch 的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个 CountDownLatch 对象的 `#await()` 方法，其他的任务执行完自己的任务后调用同一个 CountDownLatch 对象上的 `#countDown()` 方法，这个调用 `#await()` 方法的任务将一直阻塞等待，直到这个 CountDownLatch 对象的计数值减到 0 为止。
- CyclicBarrier 一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环的 barrier 。

整理表格如下：

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为 0 时释放所有等待的线程	计数达到指定值时释放所有等待线程
计数为 0 时，无法重置	计数达到指定值时，计数置为 0 重新开始
调用 `#countDown()` 方法计数减一，调用 `#await()` 方法只进行阻塞，对计数没任何影响	调用 `#await()` 方法计数加 1 ，若加 1 后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

Java 线程池



什么是 Executor 框架?

Executor 框架，是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程，会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用 Executor 框架，可以非常方便的创建一个线程池。

📣 为什么使用 Executor 框架?

1. 每次执行任务创建线程 `new Thread()` 比较消耗性能，创建一个线程是比较耗时、耗资源的。
2. 调用 `new Thread()` 创建的线程缺乏管理，被称为野线程，而且可以无限制的创建，线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪，还有线程之间的频繁交替也会消耗很多系统资源。
3. 接使用 `new Thread()` 启动的线程不利于扩展，比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

📣 在 Java 中 Executor 和 Executors 的区别?

- Executors 是 Executor 的工具类，不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。
- Executor 接口对象，能执行我们的线程任务。

    - ExecutorService 接口，继承了 Executor 接口，并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。
        - 使用 ThreadPoolExecutor ，可以创建自定义线程池。
    - Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成，并可以使用 `#get()` 方法，获取计算的结果。

讲讲线程池的实现原理

- 《Java并发编程：线程池的使用》
- 《【死磕 Java 并发】—— J.U.C 之线程池：线程池的基础架构》
- 《【死磕 Java 并发】—— J.U.C 之线程池：ThreadPoolExecutor》
- 《【死磕 Java 并发】—— J.U.C 之线程池：ScheduledThreadPoolExecutor》

创建线程池的几种方式?

Java 类库提供一个灵活的线程池以及一些有用的默认配置，我们可以通过Executors 的静态方法来创建线程池。

> Executors 创建的线程池，分成普通任务线程池，和定时任务线程池。

- 普通任务线程池
    - 1、`#newFixedThreadPool(int nThreads)` 方法，创建一个固定长度的线程池。
        - 每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化。
        - 当线程发生未预期的错误而结束时，线程池会补充一个新的线程。
    - 2、`#newCachedThreadPool()` 方法，创建一个可缓存的线程池。
        - 如果线程池的规模超过了处理需求，将自动回收空闲线程。
        - 当需求增加时，则可以自动添加新线程。线程池的规模不存在任何限制。
    - 3、`#newSingleThreadExecutor()` 方法，创建一个单线程的线程池。
        - 它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它。
        - 它的特点是，能确保依照任务在队列中的顺序来串行执行。
- 定时任务线程池
    - 4、`#newScheduledThreadPool(int corePoolSize)` 方法，创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似 Timer 。
    - 5、`#newSingleThreadExecutor()` 方法，创建了一个固定长度为 1 的线程池，而且以延迟或定时的方式来执行任务，类似 Timer 。

📣 如何使用 ThreadPoolExecutor 创建线程池?

Executors 提供了创建线程池的常用模板，实际场景下，我们可能需要自动以更灵活的线程池，此时就需要使用 ThreadPoolExecutor 类。

```
// ThreadPoolExecutor.java

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
```

```
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

- `corePoolSize` 参数，核心线程数大小，当线程数 < corePoolSize ，会创建线程执行任务。
- `maximumPoolSiz`e参数，最大线程数， 当线程数 >= corePoolSize 的时候，会把任务放入`workQueue`队列中。
  - `keepAliveTime` 参数，保持存活时间，当线程数大于 `corePoolSize` 的空闲线程能保持的最大时间。
  - `unit` 参数，时间单位。
- `workQueue`参数，保存任务的阻塞队列。
  - `handler` 参数，超过阻塞队列的大小时，使用的拒绝策略。
- `threadFactory` 参数，创建线程的工厂。

📣 ThreadPoolExecutor 有哪些拒绝策略?

ThreadPoolExecutor 默认有四个拒绝策略：

- `ThreadPoolExecutor.AbortPolicy()` ，直接抛出异常 RejectedExecutionException 。
- `ThreadPoolExecutor.CallerRunsPolicy()` ，直接调用 run 方法并且阻塞执行。
- `ThreadPoolExecutor.DiscardPolicy()` ，直接丢弃后来的任务。
- `ThreadPoolExecutor.DiscardOldestPolicy()` ，丢弃在队列中队首的任务。

如果我们有需要，可以自己实现 RejectedExecutionHandler 接口，实现自定义的拒绝逻辑。当然，绝大多数是不需要的。

📣 **

线程池的关闭方式有几种?

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是：

- `#shutdown()` 方法，不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务。
- `#shutdownNow()` 方法，立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务。

实际场景下，一般会结合这两个方法，一起实现线程池的优雅关闭。示例代码如下：

```
void shutdownAndAwaitTermination(ExecutorService pool) {
  pool.shutdown(); // Disable new tasks from being submitted
  try {
    // Wait a while for existing tasks to terminate
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
      pool.shutdownNow(); // Cancel currently executing tasks
      // Wait a while for tasks to respond to being cancelled
      if (!pool.awaitTermination(60, TimeUnit.SECONDS))
          System.err.println("Pool did not terminate");
      }
    }
  } catch (InterruptedException ie) {
    // (Re-)Cancel if current thread also interrupted
    pool.shutdownNow();
    // Preserve interrupt status
    Thread.currentThread().interrupt();
  }
}
```

Java 线程池大小为何会大多被设置成 CPU 核心数 +1？

详细的可以看看 《如何合理地估算线程池大小？》 。如下是简单的总结和整理：

一般说来，大家认为线程池的大小经验值应该这样设置：（其中 N 为CPU的个数）

- 如果是 CPU 密集型应用，则线程池大小设置为 N+1

  > 因为 CPU 密集型任务使得 CPU 使用率很高，若开过多的线程数，只能增加上下文切换的次数，因此会带来额外的开销。

- 如果是 IO 密集型应用，则线程池大小设置为 2N+1

  > IO密 集型任务 CPU 使用率并不高，因此可以让 CPU 在等待 IO 的时候去处理别的任务，充分利用 CPU 时间。

- 如果是混合型应用，那么分别创建线程池

  > 可以将任务分成 IO 密集型和 CPU 密集型任务，然后分别用不同的线程池去处理。 只要分完之后两个任务的执行时间相差不大，那么就会比串行执行来的高效。
  >
  > 因为如果划分之后两个任务执行时间相差甚远，那么先执行完的任务就要等后执行完的任务，最终的时间仍然取决于后执行完的任务，而且还要加上任务拆分与合并的开销，得不偿失。

如果一台服务器上只部署这一个应用并且只有这一个线程池，那么这种估算或许合理，具体还需自行测试验证。

但是，IO 优化中，这样的估算公式可能更适合：最佳线程数目 = （（线程等待时间 + 线程 CPU 时间）/ 线程 CPU 时间 ）* CPU 数目 **因为很显然，线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。**

下面举个例子：比如平均每个线程 CPU 运行时间为 0.5s ，而线程等待时间（非 CPU 运行时间，比如 IO）为 1.5s ，CPU 核心数为 8 。 那么根据上面这个公式估算得到：`((0.5 + 1.5) / 0.5) * 8 = 32`。这个公式进一步转化为：最佳线程数目 = （线程等待时间与线程 CPU 时间之比 + 1）* CPU数目。

线程池容量的动态调整?

ThreadPoolExecutor 提供了动态调整线程池容量大小的方法：

- setCorePoolSize：设置核心池大小。
- setMaximumPoolSize：设置线程池最大能创建的线程数目大小。

当上述参数从小变大时，ThreadPoolExecutor 进行线程赋值，还可能立即创建新的线程来执行任务。

什么是 Callable、Future、FutureTask？

1）**Callable**

Callable 接口，类似于 Runnable ，从名字就可以看出来了，但是Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值，这个返回值可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。

> 简单来说，可以认为是带有回调的 Runnable 。

2）**Future**

Future 接口，表示异步任务，是还没有完成的任务给出的未来结果。所以说 Callable 用于产生结果，Future 用于获取结果。

3）**FutureTask**

在 Java 并发程序中，FutureTask 表示一个可以取消的异步运算。

- 它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。
- 一个 FutureTask 对象，可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是继承了 Runnable 接口，所以它可以提交给 Executor 来执行。

线程池执行任务的过程?

刚创建时，里面没有线程调用 execute() 方法，添加任务时：

1. 如果正在运行的线程数量小于核心参数 `corePoolSize`，继续创建线程运行这个任务
    - 否则，如果正在运行的线程数量大于或等于 `corePoolSize` ，将任务加入到阻塞队列中。
    - 否则，如果队列已满，同时正在运行的线程数量小于核心参数 `maximumPoolSize` ，继续创建线程运行这个任务。
    - 否则，如果队列已满，同时正在运行的线程数量大于或等于 `maximumPoolSize` ，根据设置的拒绝策略处理。
2. 完成一个任务，继续取下一个任务处理。
    - 没有任务继续处理，线程被中断或者线程池被关闭时，线程退出执行，如果线程池被关闭，线程结束。
    - 否则，判断线程池正在运行的线程数量是否大于核心线程数，如果是，线程结束，否则线程阻塞。因此线程池任务全部执行完成后，继续留存的线程池大小为 `corePoolSize` 。

📣 线程池中 submit 和 execute 方法有什么区别?

两个方法都可以向线程池提交任务。

- `#execute(...)` 方法，返回类型是 `void` ，它定义在 Executor 接口中。
- `#submit(...)` 方法，可以返回持有计算结果的 Future 对象，它定义在 ExecutorService 接口中，它扩展了 Executor 接口，其它线程池类像 ThreadPoolExecutor 和 ScheduledThreadPoolExecutor 都有这些方法。

📣 如果你提交任务时，线程池队列已满，这时会发生什么？

> ：重点在于线程池的队列是有界还是无界的。

- 如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务。
- 如果你使用的是有界队列比方说 ArrayBlockingQueue 的话，任务首先会被添加到 ArrayBlockingQueue 中，ArrayBlockingQueue满了，则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy 。

Fork/Join 框架是什么?

> ：这是，可能了解的人不多，我也是。大体知道就好。

Oracle 的官方给出的定义是：Fork/Join 框架是一个实现了 ExecutorService接口 的多线程处理器。它可以把一个大的任务划分为若干个小的任务并发执行，充分利用可用的资源，进而提高应用的执行效率。

我们再通过 Fork 和 Join 这两个单词来理解下 Fork/Join 框架。

- Fork 就是把一个大任务切分为若干子任务并行的执行，Join 就是合并这些子任务的执行结果，最后得到这个大任务的结果。
- 比如计算 `1+2+...+10000` ，可以分割成 10 个子任务，每个子任务分别对 1000 个数进行求和，最终汇总这 10 个子任务的结果。

感兴趣的胖友，可以看看如下文章：

- 《JDK 7 中的 Fork/Join 模式》
- 《聊聊并发（八）—— Fork/Join 框架介绍》

如何让一段程序并发的执行，并最终汇总结果?

- 1、CountDownLatch：允许一个或者多个线程等待前面的一个或多个线程完成，构造一个 CountDownLatch 时指定需要 countDown 的点的数量，每完成一点就 countDown 一下。当所有点都完成，CountDownLatch 的 `#await()` 就解除阻塞。
- 2、CyclicBarrier：可循环使用的 Barrier ，它的作用是让一组线程到达一个 Barrier 后阻塞，直到所有线程都到达 Barrier 后才能继续执行。

  > CountDownLatch 的计数值只能使用一次，CyclicBarrier 可以通过使用 reset 重置，还可以指定到达栅栏后优先执行的任务。

- 3、Fork/Join 框架，fork 把大任务分解成多个小任务，然后汇总多个小任务的结果得到最终结果。使用一个双端队列，当线程空闲时从双端队列的另一端领取任务。

彩蛋

真多，真多，好他喵的多！耐心看。

哈哈哈，实际面试吧，也不会问这么多。嘻嘻~

参考与推荐如下文章：

- 《无锁队列的实现》
- 《Java并发编程 73 道面试题及答案》

- 《史上最全 Java 面试题（带全部答案）》
- 《最近 5 年 133 个 Java 面试问题列表》
- 《Java 基础知识》
- 《Java并发编程指南》
- 《Java 线程池大小为何会大多被设置成 CPU 核心数 +1？》
- 《Java线程面试题 Top 50》