1. 线程和进程有什么区别?

线程具有许多传统进程所具有的特征，故又称为轻型进程(Light—Weight Process)或进程元；而把传统的进程称为重型进程(Heavy—Weight Process)，它相当于只有一个线程的任务。在引入了线程的操作系统中，通常一个进程都有若干个线程，至少包含一个线程。

根本区别：进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位

资源开销：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

包含关系：如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

内存分配：同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的

影响关系：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

执行过程：每个独立的进程有程序运行的入口. 顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

2. 创建线程的三种方式的对比?

1）采用实现Runnable. Callable接口的方式创建多线程。

优势是:

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。

劣势是:

编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

2）使用继承Thread类的方式创建多线程

优势是:

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。

劣势是:

线程类已经继承了Thread类，所以不能再继承其他父类。

3）Runnable和Callable的区别

- Callable规定（重写）的方法是call()，Runnable规定（重写）的方法是run()。
- Callable的任务执行后可返回值，而Runnable的任务是不能返回值的。
- Call方法可以抛出异常，run方法不可以。
- 运行Callable任务可以拿到一个Future对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

3. 为什么要使用多线程呢?

- 从计算机底层来说： 线程可以比作是轻量级的进程，是程序执行的最小单位,**线程间的切换和调度的成本远远小于进程**。另外，多核 CPU 时代意味着多个线程可以同时运行，这减少了线程上下文切换的开销。
- 从当代互联网发展趋势来说： 现在的系统动不动就要求百万级甚至千万级的并发量，而**多线程并发编程正是开发高并发系统的基础**，利用好多线程机制可以大大提高系统整体的并发能力以及性能。

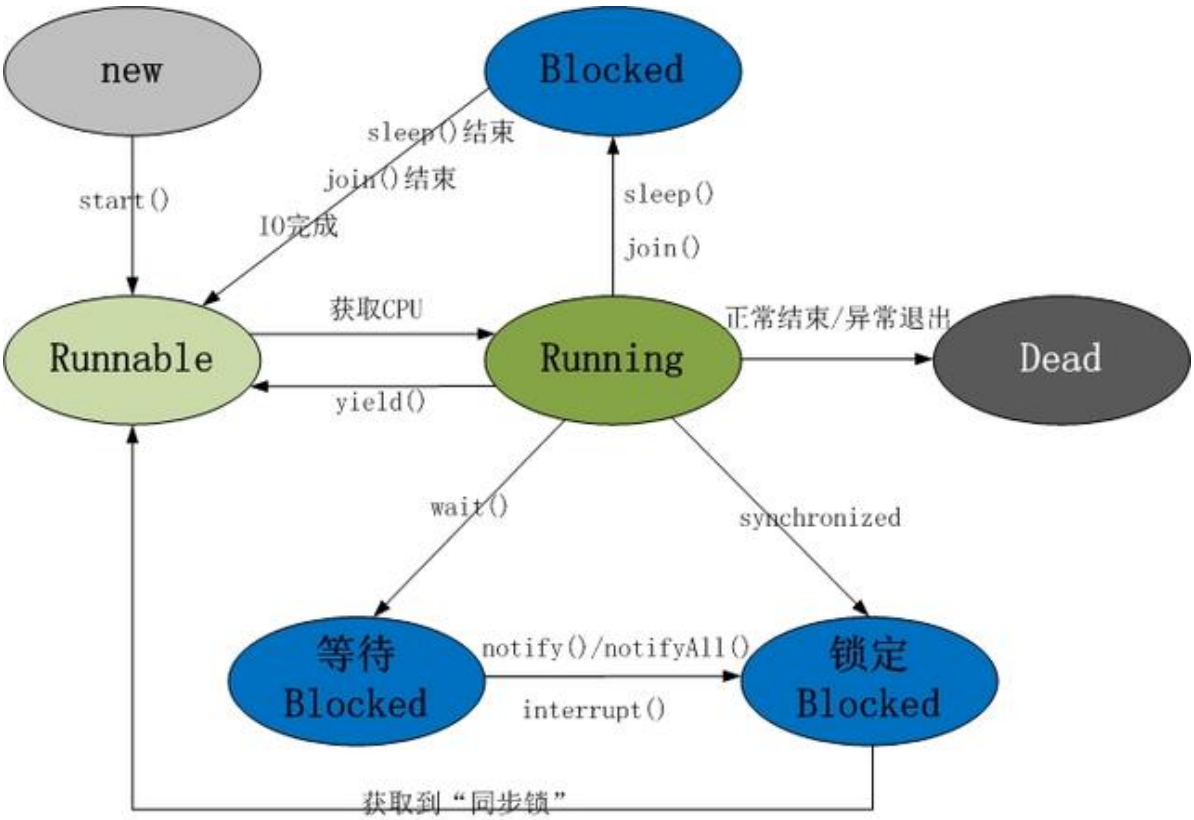
从计算机底层来说：

- 单核时代： **在单核时代多线程主要是为了提高 CPU 和 IO 设备的综合利用率**。举个例子：当只有一个线程的时候会导致 CPU 计算时，IO 设备空闲；进行 IO 操作时，CPU 空闲。我们可以简单地说这两者的利用率目前都是 50%左右。但是当有两个线程的时候就不一样了，当一个线程执行 CPU 计算时，另外一个线程可以进行 IO 操作，这样两个的利用率就可以在理想情况下达到 100%了。
- 多核时代:**多核时代多线程主要是为了提高 CPU 利用率**。举个例子：假如我们要计算一个复杂的任务，我们只用一个线程的话，CPU 只会一个 CPU 核心被利用到，而创建多个线程就可以让多个 CPU 核心被利用到，这样就提高了 CPU 的利用率。

3.对线程安全的理解

4. 线程的状态流转

线程的生命周期及五种基本状态：



Java线程具有五中基本状态

1）新建状态（New）：当线程对象对创建后，即进入了新建状态，如：Thread t = new MyThread();

2）就绪状态（Runnable）：当调用线程对象的start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了t.start()此线程立即就会执行；

3）运行状态（Running）：当CPU开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就 绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

4）阻塞状态（Blocked）：处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：

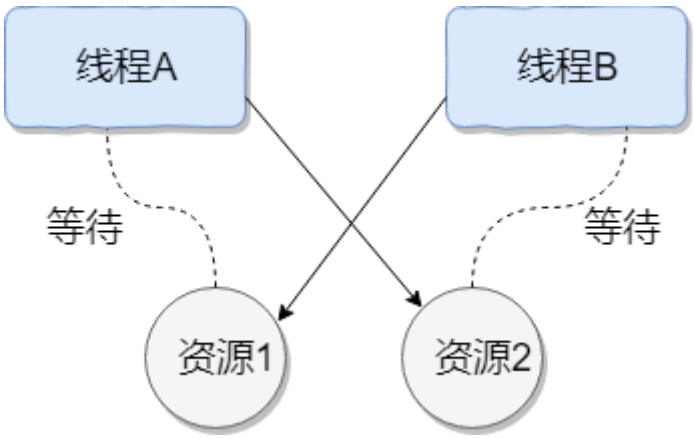
- 等待阻塞 — 运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态,该线程会释放占用的资源，JVM会把该线程放入“等待池”中，进入这个状态后，线程是不能自动唤醒的，必须依靠其他线程调用notify或者notifyAll方法才能被唤醒，wait是object类的方法；
- 同步阻塞 — 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；
- 其他阻塞 — 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时. join()等待线程终止或者超时. 或者I/O处理完毕时，线程重新转入就绪状态。

5）死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

5. 什么是线程死锁?如何避免死锁?

死锁

- 多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。



死锁必须具备以下四个条件：

- 互斥条件：该资源任意一个时刻只由一个线程占用。
- 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:线程已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
- 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。

如何避免线程死锁?

只要破坏产生死锁的四个条件中的其中一个就可以了

- 破坏互斥条件
  这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）
- 破坏请求与保持条件
  一次性申请所有的资源。
- 破坏不剥夺条件
  占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
- 破坏循环等待条件
  靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。
- 锁排序法：（必须回答出来的点）
  指定获取锁的顺序，比如某个线程只有获得A锁和B锁，才能对某资源进行操作，在多线程条件下，如何避免死锁?
  通过指定锁的获取顺序，比如规定，只有获得A锁的线程才有资格获取B锁，按顺序获取锁就可以避免死锁。这通常被认为是解决死锁很好的一种方法。
- 使用显式锁中的ReentrantLock.try(long,TimeUnit)来申请锁

6. 常见的对比

Thread VS Runnable

Thread是类，Runnable是接口。Thread和Runnable的实质是继承关系 （Thread类实现了Runnable接口，Thread类提供了更多的功能），没有可比性，无论使用Thread还是Runnable,都会new Thread，然后执行run方法，用法上，如果有复杂的线程操作需求，那就选择继承Thread，如果知识简单的执行一个任务，那就事项Runnable。

Runnable VS Callable

- Callable仅在 Java 1.5 中引入,目的就是为了来处理Runnable不支持的用例。Callable 接口可以返回结果或抛出检查异常
- Runnable 接口不会返回结果或抛出检查异常，
- 如果任务不需要返回结果或抛出异常推荐使用 Runnable接口，这样代码看起来会更加简洁

- 工具类 Executors 可以实现 Runnable 对象和 Callable 对象之间的相互转换。
  （Executors.callable（Runnable task）或 Executors.callable（Runnable task，Object resule））

shutdown() VS shutdownNow()

- shutdown（） :关闭线程池，线程池的状态变为 SHUTDOWN。线程池不再接受新任务了，但是队列里的任务得执行完毕。
- shutdownNow（） :关闭线程池，线程的状态变为 STOP。线程池会终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List。
  shutdownNow的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终

isTerminated() VS isShutdown()

- isShutDown 当调用 shutdown() 方法后返回为 true。
- isTerminated 当调用 shutdown() 方法后，并且所有提交的任务完成后返回为 true

7. sleep() 方法和 wait() 方法区别和共同点?

区别

- sleep方法：是Thread类的静态方法，当前线程将睡眠n毫秒，线程进入阻塞状态。当睡眠时间到了，会解除阻塞，进入可运行状态，等待CPU的到来。睡眠不释放锁（如果有的话）。
- wait方法：是Object的方法，必须与synchronized关键字一起使用，线程进入阻塞状态，当notify或者notifyall被调用后，会解除阻塞。但是，只有重新占用互斥锁之后才会进入可运行状态。睡眠时，会释放互斥锁。
- sleep 方法没有释放锁，而 wait 方法释放了锁 ，并且会加入到等待池队列中 。
- sleep 通常被用于暂停执行Wait 通常被用于线程间交互/通信
- sleep() 方法执行完成后，线程会自动苏醒。或者可以使用 wait(long timeout)超时后线程会自动苏醒。wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法

相同

- 两者都可以暂停线程的执行。

7. sleep()、wait()、join()、yield()的区别

1. 锁池

   所有竞争同步锁的线程都会放在锁池当中，比如当前对象的锁已经被其中一个线程得到，则其他线程需要在这个锁池中进行等待，当前面的线程释放同步锁后锁池中线程区竞争同步锁，当某个线程得到互会进入到就绪队列进行等待cpu资源分配。

2. 等待池

   当我们调用wait()方法后，线程会放到等待池当中，等待池的线程是不会去竞争同步锁的（wait操作让当前线程释放锁）。只有调用了notify()方法或者notifyAll()后等待池的线程才会去竞争锁，notify()是随机从等待池中选出一个线程放到锁池（线程放到锁池可以去竞争锁了），而notifyAll()是将等待池的所有线程放到锁池当中。

3. sleep()和wait()区别见上一问

4. yield()执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格，所以有可能cpu下次进行线程调度还会让这个线程后去到执行权继续执行（**让别人优先执行**）

5. join()执行后线程进入阻塞状态，例如线程B张调用线程A.join()，那线程B会进入到阻塞队列，指导线程A结束或者中断线程

8.为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法

- new 一个 Thread，线程进入了新建状态; 调用start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，（调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。）这是真正的多线程工作。
- 直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。
  调用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

9. Thread类中的yield方法有什么作用?

Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。

10. 谈谈volatile的使用及其原理

volatile的两层语义：

1. volatile保证变量对所有线程的可见性：当volatile变量被修改，新值对所有线程会立即更新。或者理解为多线程环境下使用volatile修饰的变量的值一定是最新的。
2. jdk1.5以后volatile完全避免了指令重排优化，实现了有序性。

volatile的原理:

获取JIT（即时Java编译器，把字节码解释为机器语言发送给处理器）的汇编代码，发现volatile多加了lock addl指令，这个操作相当于一个内存屏障，使得lock指令后的指令不能重排序到内存屏障前的位置。这也是为什么JDK1.5以后可以使用双锁检测实现单例模式。

lock前缀的另一层意义是使得本线程工作内存中的volatile变量值立即写入到主内存中，并且使得其他线程共享的该volatile变量无效化，这样其他线程必须重新从主内存中读取变量值。

具体原理见这篇文章：https://www.javazhiyin.com/61019.html

11. 如何创建线程实例并运行?

`Thread` 类本质上是实现 `Runnable` 接口的一个实例，代表一个线程的实例。创建线程实例一般有两种方法：

1. 创建 Thread 的子类并重写 `run()`

复制代码

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

`run()` 方在调用 `start()` 方法后被执行，而且一旦线程启动后 `start()` 方法后就会立即返回，而不是等到 `run()` 方法执行完毕后再返回。

```
MyThread myThread = new MyThread();
myThread.start();
```

2. 实现 Runnable 接口

```
public class MyRunnable implements Runnable{
    @Override
    public void run(){
        System.out.println("MyRunnable running");
    }
}
```

在新建类时实现 `Runnable` 接口，然后在 `Thread` 类的构造函数中传入 `MyRunnable` 的实例对象，最后执行 `start()` 方法即可；

```
Thread thread = new Thread(new MyRunnable());
thread.start();
```

12. 线程阻塞的三种情况

当线程因为某种原因放弃 CPU 使用权后，即让出了 CPU 时间片，暂时就会停止运行，直到线程进入可运行状态（`Runnable`），才有机会再次获得 CPU 时间片转入 `RUNNING` 状态。一般来讲，阻塞的情况可以分为如下三种：

1. **等待阻塞（Object.wait -> 等待队列）**

`RUNNING` 状态的线程执行 `Object.wait()` 方法后，JVM 会将线程放入等待序列（waitting queue）；

2. **同步阻塞（lock -> 锁池）**

`RUNNING` 状态的线程在获取对象的同步锁时，若该 **同步锁被其他线程占用，则 JVM 将该线程放入锁池（lock pool）中**；

3. **其他阻塞（sleep/join）**

`RUNNING` 状态的线程执行 `Thread.sleep(long ms)` 或 `Thread.join()` 方法，或发出 I/O 请求时，JVM 会将该线程置为阻塞状态。当 `sleep()` 状态超时，`join()` 等待线程终止或超时. 或者 I/O 处理完毕时，线程重新转入可运行状态（`RUNNABLE`）；

13. 线程死亡的三种方式

1. **正常结束**

`run()` 或者 `call()` 方法执行完成后，线程正常结束；

2. **异常结束**

线程抛出一个未捕获的 `Exception` 或 `Error`，导致线程异常结束；

3. **调用 stop()**

直接调用线程的 `stop()` 方法来结束该线程，但是一般不推荐使用该种方式，**因为该方法通常容易导致死锁**；

14. 为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法?

JVM执行start方法，会另起一条线程执行thread的run方法，这才起到多线程的效果~

如果直接调用Thread的run()方法，其方法还是运行在主线程中，没有起到多线程效果。

15. 守护线程是什么?

守护线程是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件，为所有非守护线程提供服务。在 Java 中垃圾回收线程就是特殊的守护线程。

> 人一个守护线程都是整个JVM中所有非守护线程的保姆

守护线程：为所有非守护线程提供服务的线程；任何一个守护线程都是整个JVM中所有非守护线程的保姆；

守护线程类似于整个进程的一个默默无闻的小喽喽；它的生死无关重要，它却依赖整个进程而运行；哪天其他线程结束了，没有要执行的了，程序就结束了，理都没理守护线程，就把它中断了；

注意： 由于守护线程的终止是自身无法控制的，因此千万不要把IO、File等重要操作逻辑分配给它；因为它不靠谱；

守护线程的作用是什么?
举例， GC垃圾回收线程：就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread,程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。
应用场景： （1）来为其它线程提供服务支持的情况； （2） 或者在任何情况下，程序结束时，这个线程必须正常且立刻关闭，就可以作为守护线程来使用；反之，如果一个正在执行某个操作的线程必须要正确地关闭掉否则就会出现不好的后果的话，那么这个线程就不能是守护线程，而是用户线程。通常都是些关键的事务，比方说，数据库录入或者更新，这些操作都是不能中断的。

thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。

在Daemon线程中产生的新线程也是Daemon的。

守护线程不能用于去访问固有资源，比如读写操作或者计算逻辑。因为它会在任何时候甚至在一个操作的中间发生中断。

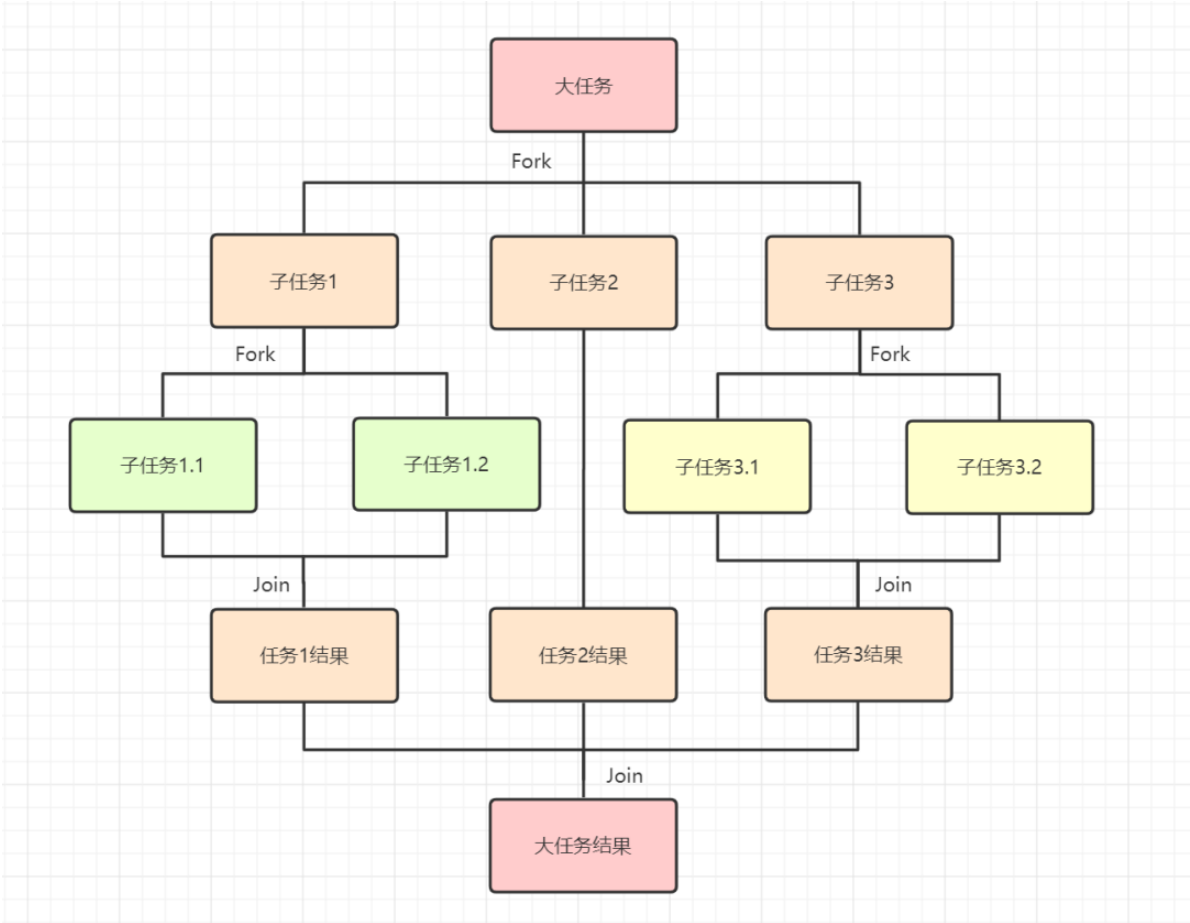
16. 了解Fork/Join框架吗?

Fork/Join框架是Java7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

Fork/Join框架需要理解两个点，**「分而治之」**和**「工作窃取算法」**。

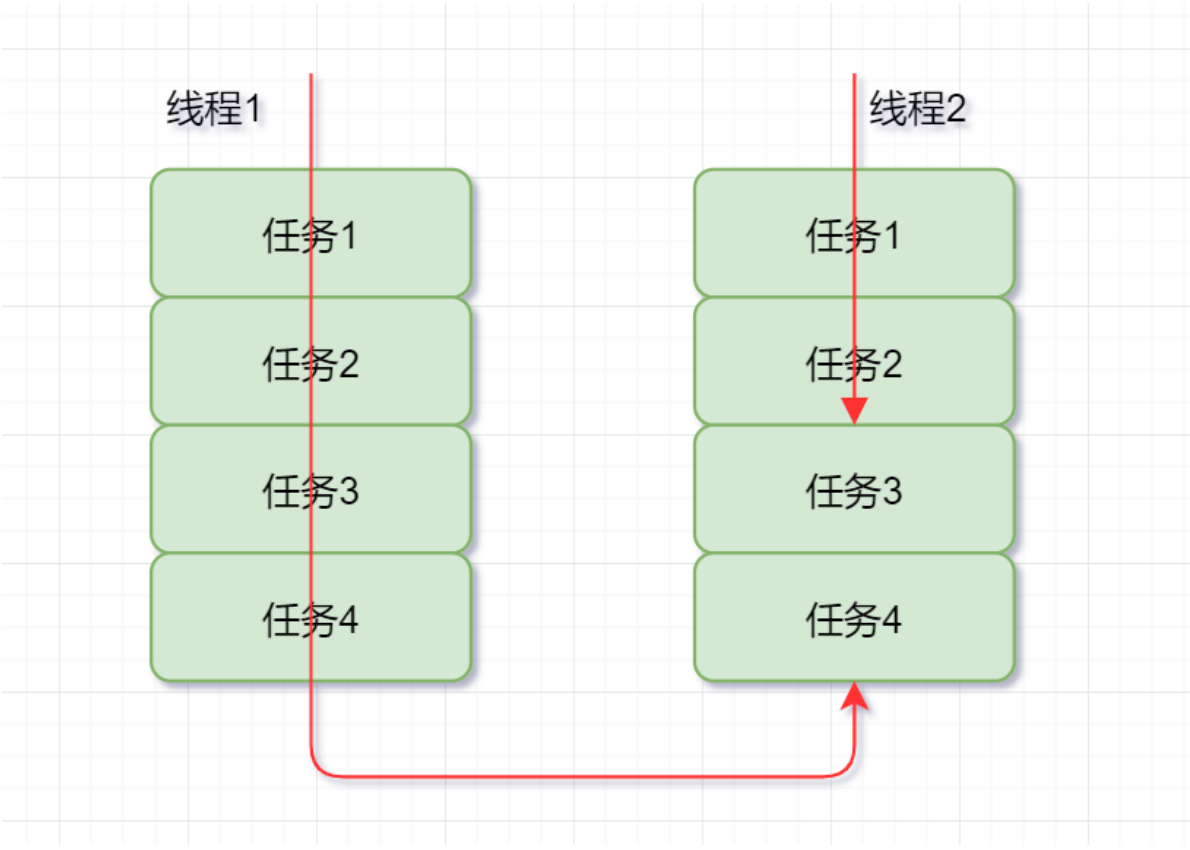
「分而治之」

以上Fork/Join框架的定义，就是分而治之思想的体现啦



「工作窃取算法」

把大任务拆分成小任务，放到不同队列执行，交由不同的线程分别执行时。有的线程优先把自己负责的任务执行完了，其他线程还在慢慢悠悠处理自己的任务，这时候为了充分提高效率，就需要工作盗窃算法啦~



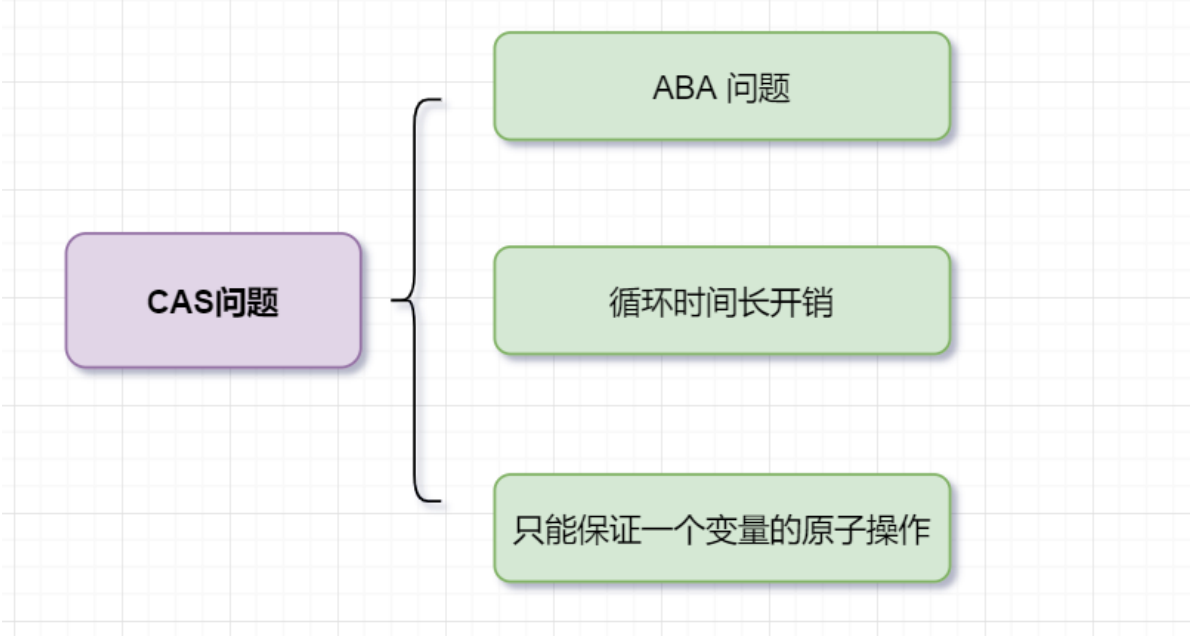
工作盗窃算法就是，**「某个线程从其他队列中窃取任务进行执行的过程」**。一般就是指做得快的线程（盗窃线程）抢慢的线程的任务来做，同时为了减少锁竞争，通常使用双端队列，即快线程和慢线程各在一端。

17. CAS了解吗?

- CAS：全称 `Compare and swap`，即**比较并交换**，它是一条 **CPU 同步原语。**是一种硬件对并发的支持，针对多处理器操作而设计的一种特殊指令，用于管理对共享数据的并发访问。
- CAS 是一种无锁的非阻塞算法的实现。
- CAS 包含了 3 个操作数:
-
  - 需要读写的内存值 V
  - 旧的预期值 A
  - 要修改的更新值 B
- 当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的 值，否则不会执行任何操作（他的功能是判断内存某个位置的值是否为预期值，如果是则更改为新的值，这个过程是原子的。）

CAS 并发原语体现在 Java 语言中的 `sum.misc.Unsafe` 类中的各个方法。调用 Unsafe 类中的 CAS 方法， JVM 会帮助我们实现出 CAS 汇编指令。这是一种完全依赖于硬件的功能，通过它实现了原子操作。再次强调，由于 CAS是一种系统原语，**原语属于操作系统用于范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断**，CAS 是一条 CPU 的原子指令，不会造成数据不一致问题。

18. CAS有什么缺陷?



1. ABA 问题

并发环境下，假设初始条件是A，去修改数据时，发现是A就会执行修改。但是看到的虽然是A，中间可能发生了A变B，B又变回A的情况。此时A已经非彼A，数据即使成功修改，也可能有问题。

可以通过AtomicStampedReference**解决ABA问题**，它，一个带有标记的原子引用类，通过控制变量值的版本来保证CAS的正确性。（加上版本号）

2. 循环时间长开销

自旋CAS，如果一直循环执行，一直不成功，会给CPU带来非常大的执行开销。

很多时候，CAS思想体现，是有个自旋次数的，就是为了避开这个耗时问题~

3. 只能保证一个变量的原子操作。

CAS 保证的是对一个变量执行操作的原子性，如果对多个变量操作时，CAS 目前无法直接保证操作的原子性的。

可以通过这两个方式解决这个问题：

- 使用互斥锁来保证原子性；
- 将多个变量封装成对象，通过AtomicReference来保证原子性。

19. synchronized 和 volatile 的区别是什么?

`volatile` 解决的是内存可见性问题，会使得所有对 `volatile` 变量的读写都直接写入主存，即 **保证了变量的可见性。**

`synchronized` 解决的是执行控制的问题，它会阻止其他线程获取当前对象的监控锁，这样一来就让当前对象中被 `synchronized` 关键字保护的代码块无法被其他线程访问，也就是无法并发执行。而且，`synchronized` 还会创建一个 **内存屏障**，内存屏障指令保证了所有 CPU 操作结果都会直接刷到主存中，从而 **保证操作的内存可见性**，同时也使得这个锁的线程的所有操作都 `happens-before` 于随后获得这个锁的线程的操作。

两者的区别主要有如下：

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。

20. synchronized 和 Lock 有什么区别?

- synchronized 可以给类. 方法. 代码块加锁；而 lock 只能给代码块加锁。
- synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。
- 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

21. synchronized 和 ReentrantLock 区别是什么?

1.两者都是可重入锁

可重入锁：重入锁，也叫做递归锁，可重入锁指的是在一个线程中可以多次获取同一把锁，比如：
一个线程在执行一个带锁的方法，该方法中又调用了另一个需要相同锁的方法，则该线程可以直接执行调用的方法，而无需重新获得锁，
两者都是同一个线程每进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

2.synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

- synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的
- ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成）

3.ReentrantLock 比 synchronized 增加了一些高级功能

相比synchronized，ReentrantLock增加了一些高级功能。主要来说主要有三点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）

- 等待可中断.通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
- ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。 ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。
- ReentrantLock类线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify()/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，**用ReentrantLock类结合Condition实例可以实现“选择性通知”**

4.使用选择

- 除非需要使用 ReentrantLock 的高级功能，否则优先使用 synchronized。
- synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 ReentrantLock 不是所有的 JDK 版本都支持。并且使用 synchronized 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放

22. synchronized的用法有哪些?

- 修饰普通方法:作用于当前对象实例，进入同步代码前要获得当前对象实例的锁
- 修饰静态方法:作用于当前类，进入同步代码前要获得当前类对象的锁,synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁
- 修饰代码块:指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁

特别注意：

①如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁

②尽量不要使用 synchronized(String s) ,因为JVM中，字符串常量池具有缓冲功能

23. Synchronized的作用有哪些?

1. 原子性：确保线程互斥的访问同步代码；
2. 可见性：保证共享变量的修改能够及时可见，其实是通过Java内存模型中的 “**对一个变量unlock操作之前，必须要同步到主内存中；如果对一个变量进行lock操作，则将会清空工作内存中此变量的值，在执行引擎使用此变量前，需要重新从主内存中load操作或assign操作初始化变量值**” 来保证的；
3. 有序性：有效解决重排序问题，即 “一个unlock操作先行发生(happen-before)于后面对同一个锁的lock操作”。

24. 说一下 synchronized 底层实现原理?

synchronized 同步代码块的实现是通过 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。

其内部包含一个计数器，当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

25. 多线程中 synchronized 锁升级的原理是什么?

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

26. synchronized 为什么是非公平锁？非公平体现在哪些地方?

synchronized 的非公平其实在源码中应该有不少地方，因为设计者就没按公平锁来设计，核心有以下几个点：

1）当持有锁的线程释放锁时，该线程会执行以下两个重要操作：

1. 先将锁的持有者 owner 属性赋值为 null
2. 唤醒等待链表中的一个线程（假定继承者）。

在1和2之间，如果有其他线程刚好在尝试获取锁（例如自旋），则可以马上获取到锁。

2）当线程尝试获取锁失败，进入阻塞时，放入链表的顺序，和最终被唤醒的顺序是不一致的，也就是说你先进入链表，不代表你就会先被唤醒。

27. JVM对synchronized的优化有哪些?

从最近几个jdk版本中可以看出，Java的开发团队一直在对synchronized优化，其中最大的一次优化就是在jdk6的时候，新增了两个锁状态，通过锁消除、锁粗化、自旋锁等方法使用各种场景，给synchronized性能带来了很大的提升。

1. 锁膨胀

上面讲到锁有四种状态，并且会因实际情况进行膨胀升级，其膨胀方向是：**无锁——>偏向锁——>轻量级锁——>重量级锁**，并且膨胀方向不可逆。

偏向锁

一句话总结它的作用：**减少统一线程获取锁的代价。**在大多数情况下，锁不存在多线程竞争，总是由同一线程多次获得，那么此时就是偏向锁。

核心思想：

如果一个线程获得了锁，那么锁就进入偏向模式，此时`Mark Word`的结构也就变为偏向锁结构，**当该线程再次请求锁时，无需再做任何同步操作，即获取锁的过程只需要检查**`**Mark Word**`**的锁标记位为偏向锁以及当前线程ID等于**`**Mark Word**`**的ThreadID即可**，这样就省去了大量有关锁申请的操作。

轻量级锁

轻量级锁是由偏向锁升级而来，当存在第二个线程申请同一个锁对象时，偏向锁就会立即升级为轻量级锁。注意这里的第二个线程只是申请锁，不存在两个线程同时竞争锁，可以是一前一后地交替执行同步块。

重量级锁

重量级锁是由轻量级锁升级而来，当**同一时间**有多个线程竞争锁时，锁就会被升级成重量级锁，此时其申请锁带来的开销也就变大。

重量级锁一般使用场景会在追求吞吐量，同步块或者同步方法执行时间较长的场景。

2.锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，在JIT编译时，对运行上下文进行扫描，去除不可能存在竞争的锁。比如下面代码的method1和method2的执行效率是一样的，因为object锁是私有变量，不存在所得竞争关系。

```
public class Test4 {

    public void method1() {
        Object object = new Object();
        synchronized (object) {
            //  执行同步代码
            System.out.println("hello world.");
        }
    }

    //  优化后的方法，和上面method1执行效率一样
    public void method2() {
        Object object = new Object();
        System.out.println("hello world.");
    }
}
```

3. 锁粗化

锁粗化是虚拟机对另一种极端情况的优化处理，通过扩大锁的范围，避免反复加锁和释放锁。比如下面method3经过锁粗化优化之后就和method4执行效率一样了。

```java
public void method3() {
    for (int i = 0; i < 10000; ++i) {
        synchronized (Test4.class) {
            System.out.println("hello world.");
        }
    }
}

//  锁粗化，与上面一样
public void method4() {
    synchronized (Test4.class) {
        for (int i = 0; i < 10000; ++i) {
            System.out.println("hello world.");
        }
    }
}
```

4. 自旋锁与自适应自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

自旋锁：许多情况下，共享数据的锁定状态持续时间较短，切换线程不值得，通过让线程执行循环等待锁的释放，不让出CPU。如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式。但是它也存在缺点：如果锁被其他线程长时间占用，一直不释放CPU，会带来许多的性能开销。

自适应自旋锁：这种相当于是对上面自旋锁优化方式的进一步优化，它的自旋的次数不再固定，其自旋的次数由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定，这就解决了自旋锁带来的缺点。

为什么要引入偏向锁和轻量级锁？为什么重量级锁开销大?

重量级锁底层依赖于系统的同步函数来实现，在 linux 中使用 pthread_mutex_t（互斥锁）来实现。

这些底层的同步函数操作会涉及到：操作系统用户态和内核态的切换、进程的上下文切换，而这些操作都是比较耗时的，因此重量级锁操作的开销比较大。

而在很多情况下，可能获取锁时只有一个线程，或者是多个线程交替获取锁，在这种情况下，使用重量级锁就不划算了，因此引入了偏向锁和轻量级锁来降低没有并发竞争时的锁开销。

28. synchronized 锁能降级吗?

可以的。

具体的触发时机：在全局安全点（safepoint）中，执行清理任务的时候会触发尝试降级锁。

当锁降级时，主要进行了以下操作：

1）恢复锁对象的 markword 对象头；

2）重置 ObjectMonitor，然后将该 ObjectMonitor 放入全局空闲列表，等待后续使用。

29. ThreadLocal是什么?

ThreadLocal，即线程本地变量。如果你创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的一个本地拷贝，多个线程操作这个变量的时候，实际是操作自己本地内存里面的变量，从而起到**线程隔离**的作用，避免了线程安全问题。

```
//创建一个ThreadLocal变量
static ThreadLocal<String> localVariable = new ThreadLocal<>();
```

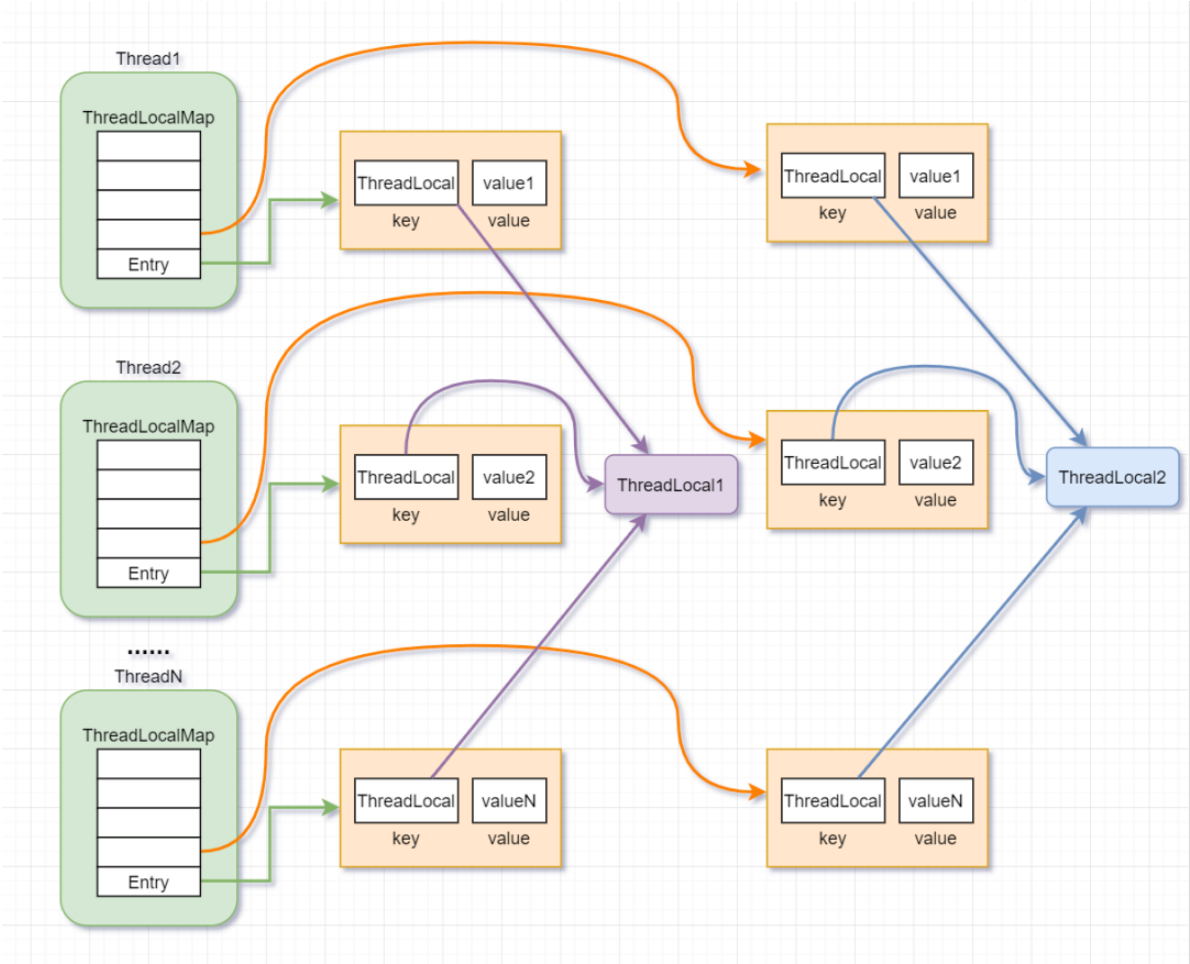
ThreadLocal的应用场景有

- 数据库连接池
- 会话管理中使用

30. ThreadLocal的实现原理

- Thread类有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，即每个线程都有一个属于自己的ThreadLocalMap。
    - ThreadLocalMap是ThreadLocal的静态内部类
- ThreadLocalMap内部维护着Entry数组（继承自WeakReference），每个Entry代表一个完整的对象，key是ThreadLocal本身，value是ThreadLocal的泛型值。
- 每个线程在往ThreadLocal里设置值的时候，都是往自己的ThreadLocalMap里存，读也是以某个ThreadLocal作为引用，在自己的map里找对应的key，从而实现了线程隔离。

ThreadLocal内存结构图：



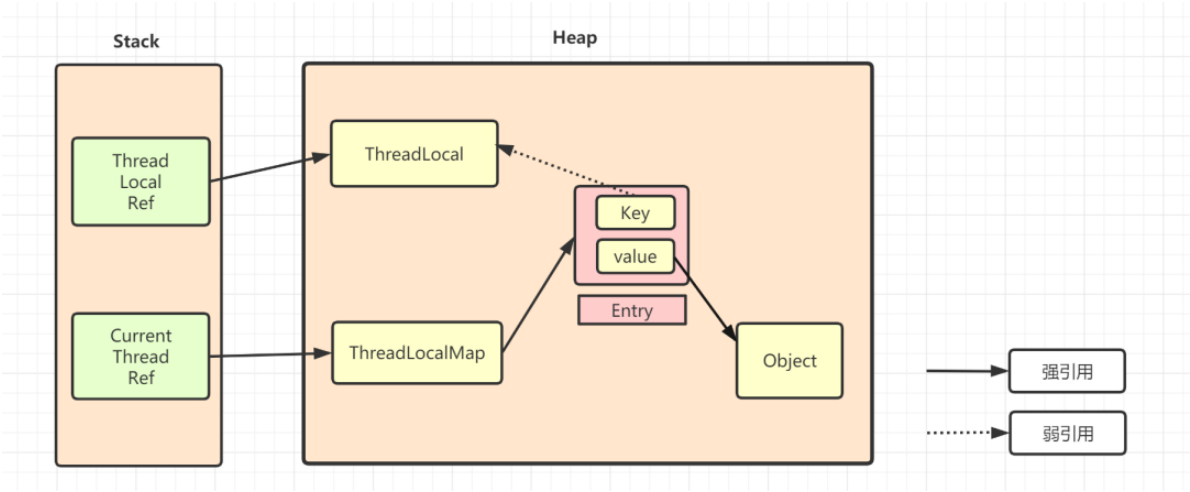
由结构图是可以看出：

- Thread对象中持有一个ThreadLocal.ThreadLocalMap的成员变量。

- ThreadLocalMap内部维护了Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal本身，value是ThreadLocal的泛型值。

31. 知道ThreadLocal 内存泄露问题吗?

先看看一下的TreadLocal的引用示意图哈，



ThreadLocalMap中使用的 key 为 ThreadLocal 的弱引用，如下：

```
static class ThreadLocalMap {
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
}
```

> 弱引用：只要垃圾回收机制一运行，不管JVM的内存空间是否充足，都会回收该对象占用的内存。

弱引用比较容易被回收。因此，如果ThreadLocal（ThreadLocalMap的Key）被垃圾回收器回收了，但是因为ThreadLocalMap生命周期和Thread是一样的，它这时候如果不被回收，就会出现这种情况：ThreadLocalMap的key没了，value还在，这就会**「造成了内存泄漏问题」**。

如何**「解决内存泄漏问题」**？

1. 使用完ThreadLocal后，及时调用remove()方法释放内存空间。
2. JDK建议ThreadLocal定义为private static，这样ThreadLocal的弱引用问题则不存在了。

32. 了解ReentrantLock吗?

ReetrantLock是一个可重入的独占锁，主要有两个特性，一个是支持公平锁和非公平锁，一个是可重入。
ReetrantLock实现依赖于AQS(AbstractQueuedSynchronizer)。

ReetrantLock主要依靠AQS维护一个阻塞队列，多个线程对加锁时，失败则会进入阻塞队列。等待唤醒，重新尝试加锁。

33. ReadWriteLock是什么?

首先ReentrantLock某些时候有局限，如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能

线程池专题

1. 为什么要用线程池?

线程池提供了一种限制和管理资源（包括执行一个任务）。 每个线程池还维护一些基本统计信息，例如已完成任务的数量。

使用线程池的好处：

- **降低资源消耗。** 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。** 当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- **提高线程的可管理性。** 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

2. 执行execute()方法和submit()方法的区别是什么呢?

- `execute()` **方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；**
- **submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功**，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用 `get（long timeout，TimeUnit unit）` 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

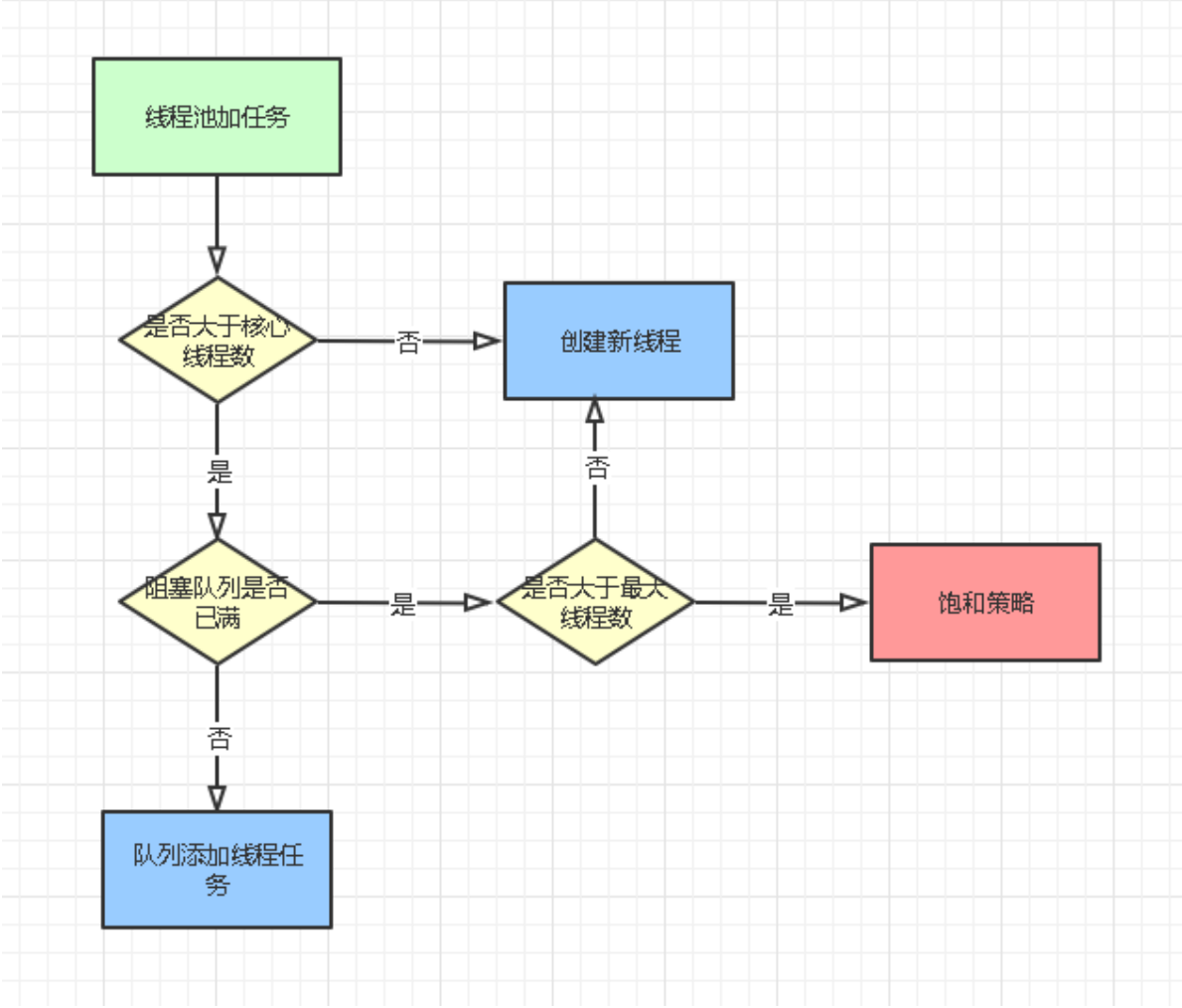
3. 你说下线程池核心参数? （七个）

- corePoolSize： 代表核心线程数，也就是正常情况下创建工作的线程数，这些线程创建后并不会消除，而是一种常驻线程。线程池一直运行，核心线程就不会停止。（**核心线程回收是随着线程池一起回收的**）
- maximumPoolSize： 代表最大线程数，它与核心线程数相对应，表示最大荀彧被创建的线程数，比如当前任务较多，将核心线程数用完了，还无法满足需求时，此时就会创建新的线程，但是线程池内线程总数不会超过最大线程数。非核心线程数量=maximumPoolSize-corePoolSize
- keepAliveTime： 非核心线程的心跳时间。如果非核心线程在keepAliveTime内没有运行任务，非核心线程会消亡（核心线程不会消亡）。
- unit：KeepAliveTime的时间单位
- workQueue： 用来存放待执行的任务，假设我们核心线程都已经使用，还有任务进来则全部放入队列，直到整个队列被放满但任务还在持续进入则会开始创建新的线程
- handler： 饱和策略（拒绝策略）。ThreadPoolExecutor类中一共有4种饱和策略。通过实现**RejectedExecutionHandler**接口。

有两种情况会产生拒绝，第一种是当我们调用shutdown等方法关闭线程池后，这时候及时线程池内部还有没执行完的任务正在执行，但是由于线程池已经关闭，我们再继续向线程池提交任务就会遭到拒绝；另一种情况就是当达到最大线程数，线程池已经没有能力继续处理新提交的任务时，也会遭到拒绝。

    - AbortPolicy： 线程任务丢弃报错。默认饱和策略。
    - DiscardPolicy： 线程任务直接丢弃不报错。
    - DiscardOldestPolicy： 将workQueue**队首任务丢弃**，将最新线程任务重新加入队列执行。
    - CallerRunsPolicy： 线程池之外的线程直接调用run方法执行。
- ThreadFactory： 实际上是一个线程工厂，用来生产线程执行任务，我们可以选择使用默认的创建工厂，产生的线程都在同一组内，拥有相同的优先级，且都不是守护线程。当然我们也可以选择自定义线程工厂，一般我们会根据业务来制定不同的线程工厂。

4. 线程池执行任务的流程?



1. 线程池执行execute/submit方法向线程池添加任务，当任务小于核心线程数corePoolSize，线程池中可以创建新的线程。
2. 当任务大于核心线程数corePoolSize，就向阻塞队列添加任务。
3. 如果阻塞队列已满，需要通过比较参数maximumPoolSize，在线程池创建新的线程，当线程数量大于maximumPoolSize，说明当前设置线程池中线程已经处理不了了，就会执行饱和策略。

> 为什么先添加到阻塞队列而不是先创建最大线程?
>
> 在创建新线程的时候，是要获取全局锁的，这个时候其他线程就得阻塞，影响整体效率。
>
> 在核心线程已满时，如果任务继续增加那么放在队列中，等队列满了而任务还在增加那么就要创建临时线程了，这样代价低。

5. 常用的JAVA线程池有哪几种类型?

1、newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

这种类型的线程池特点是：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统OOM。

2、newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

3、newSingleThreadExecutor

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

4、newScheduleThreadPool

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。

6. 线程池常用的阻塞队列有哪些?

一般的队列只能保证作为一个优先长度的缓冲区，如果超出缓冲长度，就无法保留当前的任务了，阻塞队列通过阻塞可以保留住当前想要继续入队的任务

阻塞队列可以保证任务队列中没有任务时阻塞获取任务的线程，使得线程进入wait状态，释放cpu资源。

阻塞队列自带阻塞和唤醒的功能，不需要额外处理，无任务执行时，线程池利用阻塞队列的take方法挂起，从而维持核心线程的存活，不至于一直占用cpu资源。

FixedThreadPool	LinkedBlockingQueue
SingleThreadExecutor	LinkedBlockingQueue
CachedThreadPool	SynchronousQueue
ScheduledThreadPool	DelayedWorkQueue
SingleThreadScheduledExecutor	DelayedWorkQueue

表格左侧是线程池，右侧为它们对应的阻塞队列，可以看到 5 种线程池对应了 3 种阻塞队列

1. LinkedBlockingQueue
   对于 FixedThreadPool 和 SingleThreadExector 而言，它们使用的阻塞队列是容量为 Integer.MAX_VALUE 的 LinkedBlockingQueue，可以认为是无界队列。由于 FixedThreadPool 线程池的线程数是固定的，所以没有办法增加特别多的线程来处理任务，这时就需要 LinkedBlockingQueue 这样一个没有容量限制的阻塞队列来存放任务。

   这里需要注意，由于线程池的任务队列永远不会放满，所以线程池只会创建核心线程数量的线程，所以此时的最大线程数对线程池来说没有意义，因为并不会触发生成多于核心线程数的线程。

2. SynchronousQueue
   第二种阻塞队列是 SynchronousQueue，对应的线程池是 CachedThreadPool。线程池 CachedThreadPool 的最大线程数是 Integer 的最大值，可以理解为线程数是可以无限扩展的。CachedThreadPool 和上一种线程池 FixedThreadPool 的情况恰恰相反，FixedThreadPool 的情况是阻塞队列的容量是无限的，而这里 CachedThreadPool 是线程数可以无限扩展，所以 CachedThreadPool 线程池并不需要一个任务队列来存储任务，因为一旦有任务被提交就直接转发给线程或者创建新线程来执行，而不需要另外保存它们。
   我们自己创建使用 SynchronousQueue 的线程池时，如果不希望任务被拒绝，那么就需要注意设置最大线程数要尽可能大一些，以免发生任务数大于最大线程数时，没办法把任务放到队列中也没有足够线程来执行任务的情况。

3. DelayedWorkQueue
   第三种阻塞队列是DelayedWorkQueue，它对应的线程池分别是 ScheduledThreadPool 和 SingleThreadScheduledExecutor，这两种线程池的最大特点就是可以延迟执行任务，比如说一定时间后执行任务或是每隔一定的时间执行一次任务。

DelayedWorkQueue 的特点是内部元素并不是按照放入的时间排序，而是会按照延迟的时间长短对任务进行排序，内部采用的是“堆”的数据结构。之所以线程池 ScheduledThreadPool 和 SingleThreadScheduledExecutor 选择 DelayedWorkQueue，是因为它们本身正是基于时间执行任务的，而延迟队列正好可以把任务按时间进行排序，方便任务的执行。

7. 源码中线程池是怎么复用线程的？（线程池中线程复用原理）

线程池将线程和任务进行解耦，线程是线程，任务时任务，摆脱了之前Thread创建线程时一个线程必须对应一个任务的限制。

在线程池中，同一个线程可以从阻塞队列中不断的获取新任务执行，其核心原理在于线程池对Thread进行了封装，并不是每一次调用线程都会调用Thread.start()来创建新线程，而是让每个线程去执行循环任务，在这个循环任务中不停检查是否有任务需要被执行，如果有则直接执行，也就是调用任务中的run方法，将run方法当成一个普通方法执行，通过这种方式只使固定的线程就将所有任务run方法串联起来了。

> 源码中ThreadPoolExecutor中有个内置对象Worker，线程池内的线程都被包装成一个Worker对象，worker线程数量和参数有关，每个worker会while死循环从阻塞队列中取数据，**通过置换worker中Runnable对象，运行其run方法起到线程置换的效果**，这样做的好处是避免多线程频繁线程切换，提高程序运行性能。

8. 如何合理配置线程池参数?

自定义线程池就需要我们自己配置最大线程数 maximumPoolSize ，为了高效的并发运行，这时需要看我们的业务是IO密集型还是CPU密集型。

CPU密集型
CPU密集的意思是该任务需要最大的运算，而没有阻塞，CPU一直全速运行。CPU密集任务只有在真正的多核CPU上才能得到加速(通过多线程)。而在单核CPU上，无论你开几个模拟的多线程该任务都不可能得到加速，因为CPU总的运算能力就那么多。

IO密集型
IO密集型，即该任务需要大量的IO，即大量的阻塞。在单线程上运行IO密集型的任务会导致大量的CPU运算能力浪费在等待。所以在IO密集型任务中使用多线程可以大大的加速程序运行，即使在单核CPU上这种加速主要就是利用了被浪费掉的阻塞时间。

IO 密集型时，大部分线程都阻塞，故需要多配制线程数。公式为：

```
CPU核数*2
CPU核数/(1-阻塞系数) 阻塞系数在0.8~0.9之间
查看CPU核数：
System.out.println(Runtime.getRuntime().availableProcessors());
```

当以上都不适用时，选用动态化线程池，看美团技术团队的实践：https://tech.meituan.com/2020/04/02/java-pooling-pratice-in-meituan.html

9. Executor和Executors的区别?

Executors 工具类的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。

AQS

1. 说一说什么是AQS?

1. AQS 是一个锁框架，它定义了锁的实现机制，并开放出扩展的地方，让子类去实现，比如我们在 lock 的时候，AQS 开放出 state 字段，让子类可以根据 state 字段来决定是否能够获得锁，对于获取不到锁的线程 AQS 会自动进行管理，无需子类锁关心，这就是 lock 时锁的内部机制，封装的很好，又暴露出子类锁需要扩展的地方；

2. AQS 底层是由同步队列 + 条件队列联手组成，同步队列管理着获取不到锁的线程的排队和释放，条件队列是在一定场景下，对同步队列的补充，比如获得锁的线程从空队列中拿数据，肯定是拿不到数据的，这时候条件队列就会管理该线程，使该线程阻塞；
3. AQS 围绕两个队列，提供了四大场景，分别是：获得锁、释放锁、条件队列的阻塞，条件队列的唤醒，分别对应着 AQS 架构图中的四种颜色的线的走向。

2. AQS使用了哪些设计模式?

AQS同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

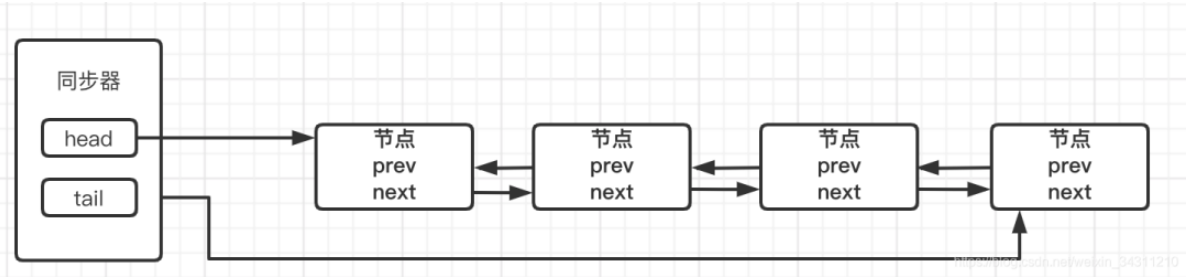
1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）
2. 将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法：

```
isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。
tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。
tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。
tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。
```

3. 了解AQS中同步队列的数据结构吗?



- 当前线程获取同步状态失败，同步器将当前线程机等待状态等信息构造成一个Node节点加入队列，放在队尾，同步器重新设置尾节点
- 加入队列后，会阻塞当前线程
- 同步状态被释放并且同步器重新设置首节点，同步器唤醒等待队列中第一个节点，让其再次获取同步状态

4. 了解AQS 对资源的共享方式吗?

AQS定义两种资源共享方式

- Exclusive

  （独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：

  - 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
  - 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
- **Share**（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。Semaphore、CountDownLatCh、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式，因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。

5. AQS 组件了解吗?

- **Semaphore(信号量)-允许多个线程同时访问：** synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。
- **CountDownLatch （倒计时器）：** CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。
- **CyclicBarrier(循环栅栏)：** CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的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。

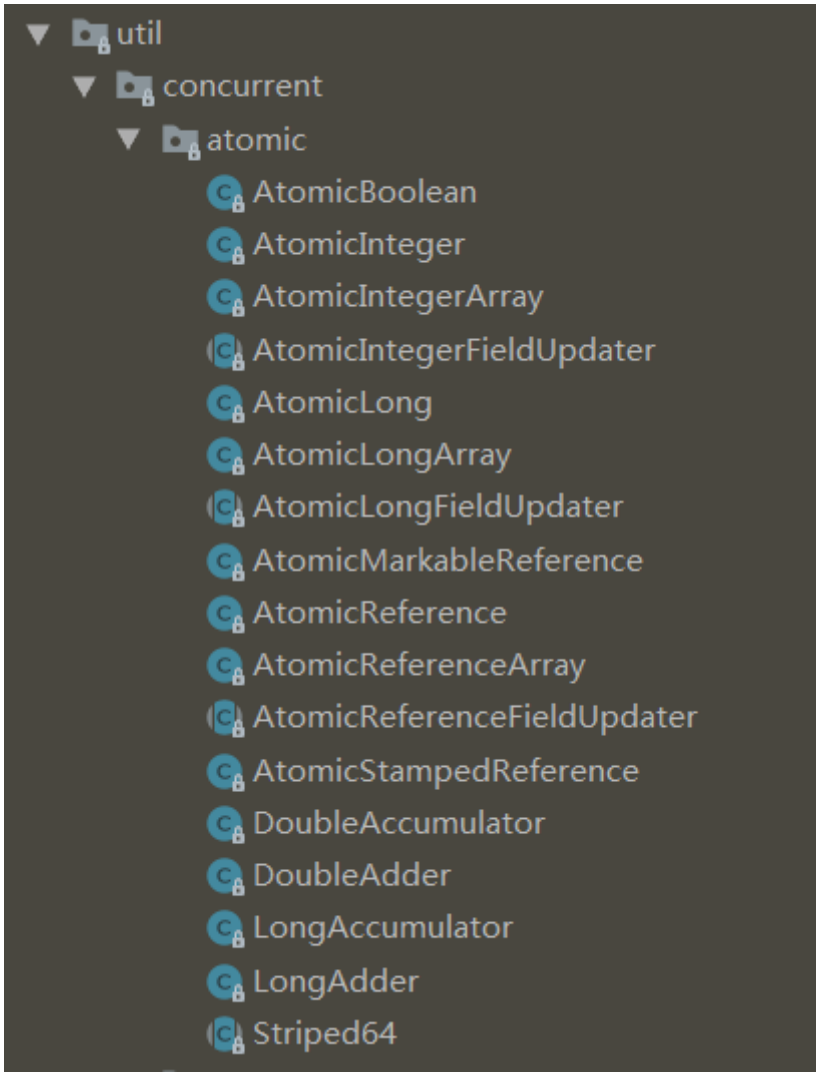
Atomic 原子类

1. 介绍一下 Atomic 原子类

Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。

所以，所谓原子类说简单点就是具有原子 / 原子操作特征的类。

并发包 java.util.concurrent 的原子类都存放在 java.util.concurrent.atomic 下：



2. JUC 包中的原子类是哪4类?

基本类型
使用原子的方式更新基本类型：

- AtomicInteger： 整型原子类
- AtomicLong： 长整型原子类
- AtomicBoolean： 布尔型原子类

数组类型
使用原子的方式更新数组里的某个元素：

- AtomicIntegerArray： 整型数组原子类
- AtomicLongArray： 长整型数组原子类
- AtomicReferenceArray： 引用类型数组原子类

引用类型
使用原子的方式更新引用类型：

- AtomicReference： 引用类型原子类
- AtomicStampedReference： 原子更新带有版本号的引用类型。该类将整型数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicMarkableReference： 原子更新带有标记位的引用类型。**对象属性修改类型**
- AtomicIntegerFieldUpdater： 原子更新整型字段的更新器
- AtomicLongFieldUpdater： 原子更新长整型字段的更新器
- AtomicMarkableReference： 原子更新带有标记位的引用类型

3. 简单介绍一下 AtomicInteger 类的原理

AtomicInteger 类主要利用 CAS和 volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

`AtomicInteger` 类的部分源码：

```
    // 更新操作时提供“比较并替换”的作用
    private static final Unsafe unsafe = Unsafe.getUnsafe();

    private static final long valueOffset;

    static {
        try{
            valueOffset =
unsafe.objectFieldOffset(AutomicInteger.class.getDeclaredField("value"));
        }catch(Exception ex){
            throw new Error(ex);
        }
    }

    private volatile int value;
```

参考

https://www.cnblogs.com/java1024/p/13390538.html

https://segmentfault.com/a/1190000039258680

https://github.com/pengMaster/BestNote

http://static.kancloud.cn/alex_wsc/java_source_interview/1875015

https://blog.csdn.net/zycxnanwang/article/details/105321401

https://blog.csdn.net/weixin_45124488/article/details/115200512