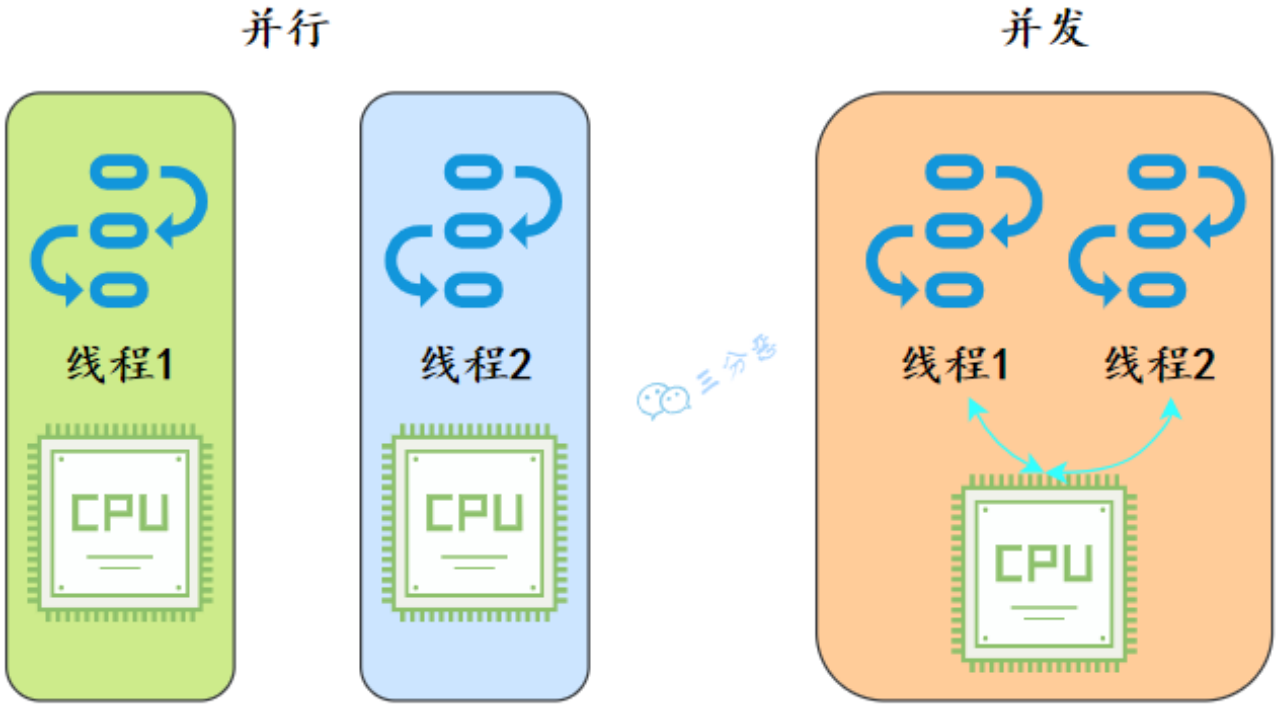
图文详解 60 道Java并发面试高频题，这次面试，一定吊打面试官，整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。

基础

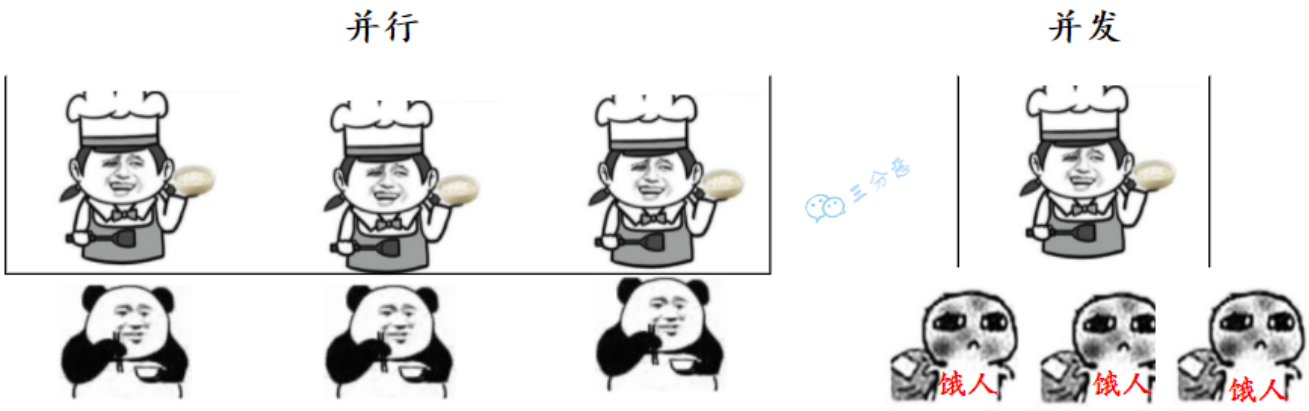
1.并行跟并发有什么区别？

从操作系统的角度来看，线程是CPU分配的最小单位。

- 并行就是同一时刻，两个线程都在执行。这就要求有两个CPU去分别执行两个线程。
- 并发就是同一时刻，只有一个执行，但是一个时间段内，两个线程都执行了。并发的实现依赖于CPU切换线程，因为切换的时间特别短，所以基本对于用户是无感知的。



就好像我们去食堂打饭，并行就是我们在多个窗口排队，几个阿姨同时打菜；并发就是我们挤在一个窗口，阿姨给这个打一勺，又手忙脚乱地给那个打一勺。



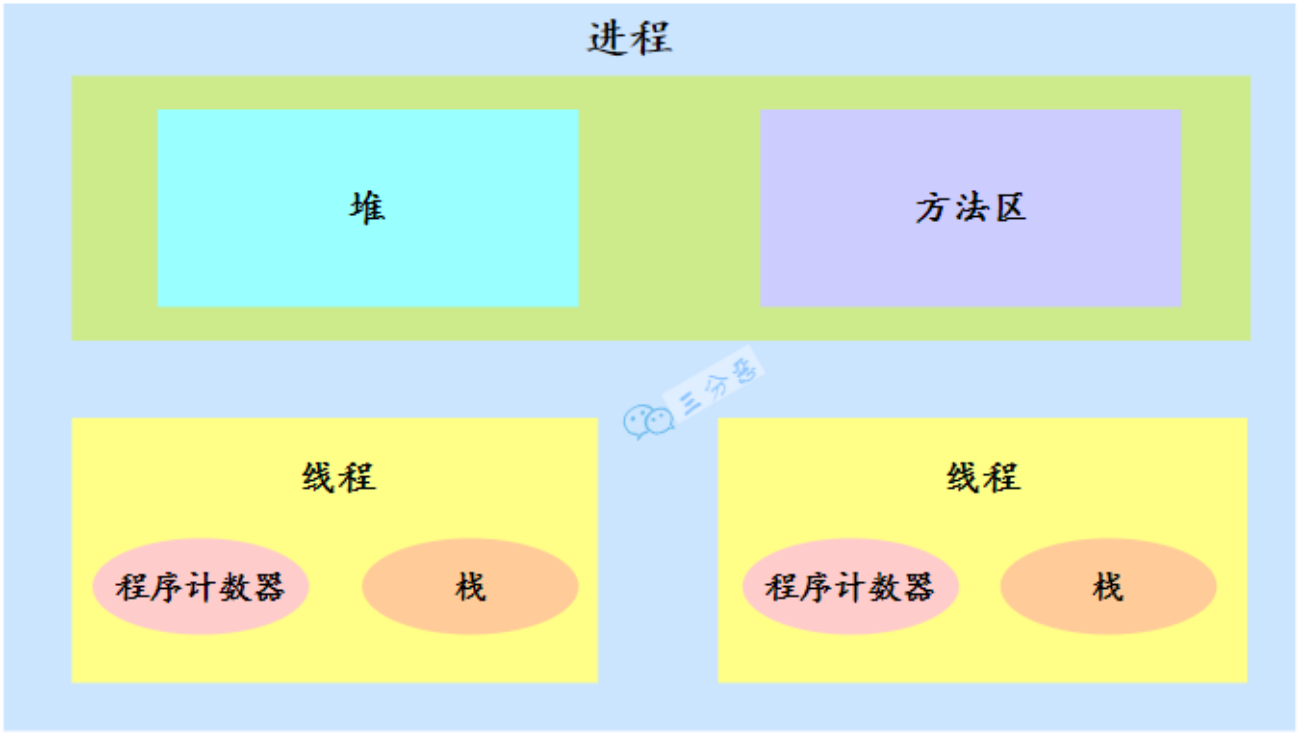
2.说说什么是进程和线程?

要说线程，必须得先说说进程。

- 进程：进程是代码在数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。
- 线程：线程是进程的一个执行路径，一个进程中至少有一个线程，进程中的多个线程共享进程的资源。

操作系统在分配资源时是把资源分配给进程的，但是 CPU 资源比较特殊，它是被分配到线程的，因为真正要占用CPU运行的是线程，所以也说线程是 CPU分配的基本单位。

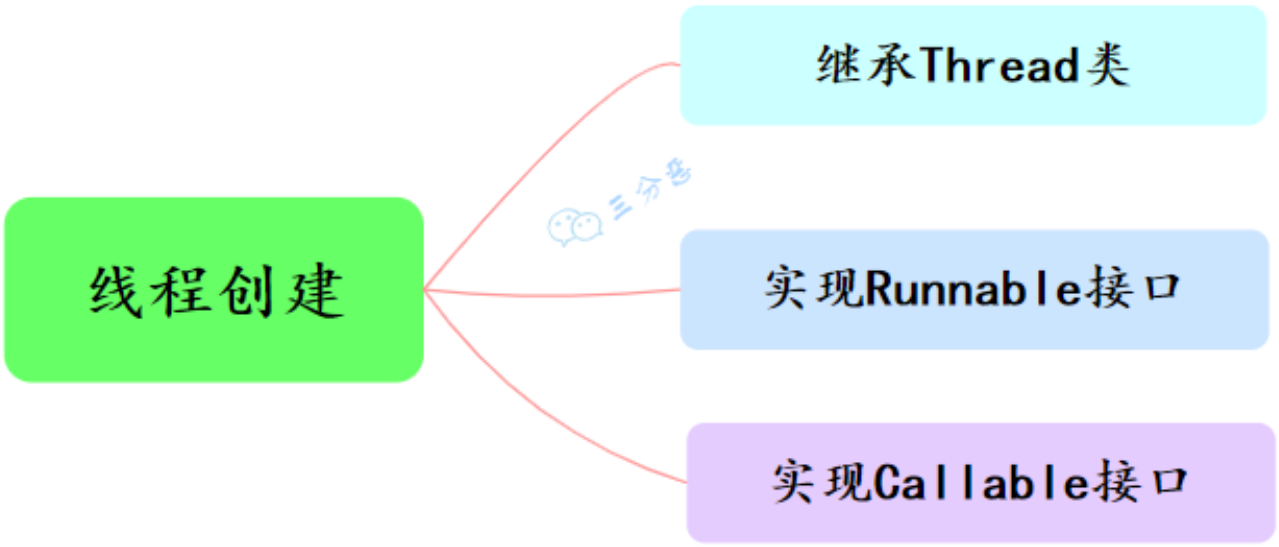
比如在Java中，当我们启动 main 函数其实就启动了一个JVM进程，而 main 函数在的线程就是这个进程中的一个线程，也称主线程。



一个进程中有多个线程，多个线程共用进程的堆和方法区资源，但是每个线程有自己的程序计数器和栈。

3.说说线程有几种创建方式?

Java中创建线程主要有三种方式，分别为继承Thread类、实现Runnable接口、实现Callable接口。



- 继承Thread类，重写run()方法，调用start()方法启动线程

```
public class ThreadTest {

    /**
     * 继承Thread类
     */
    public static class MyThread extends Thread {
        @Override
        public void run() {
            System.out.println("This is child thread");
        }
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

- 实现 Runnable 接口，重写run()方法

```
public class RunnableTask implements Runnable {
    public void run() {
        System.out.println("Runnable!");
    }

    public static void main(String[] args) {
        RunnableTask task = new RunnableTask();
        new Thread(task).start();
    }
}
```

上面两种都是没有返回值的，但是如果我们需要获取线程的执行结果，该怎么办呢？

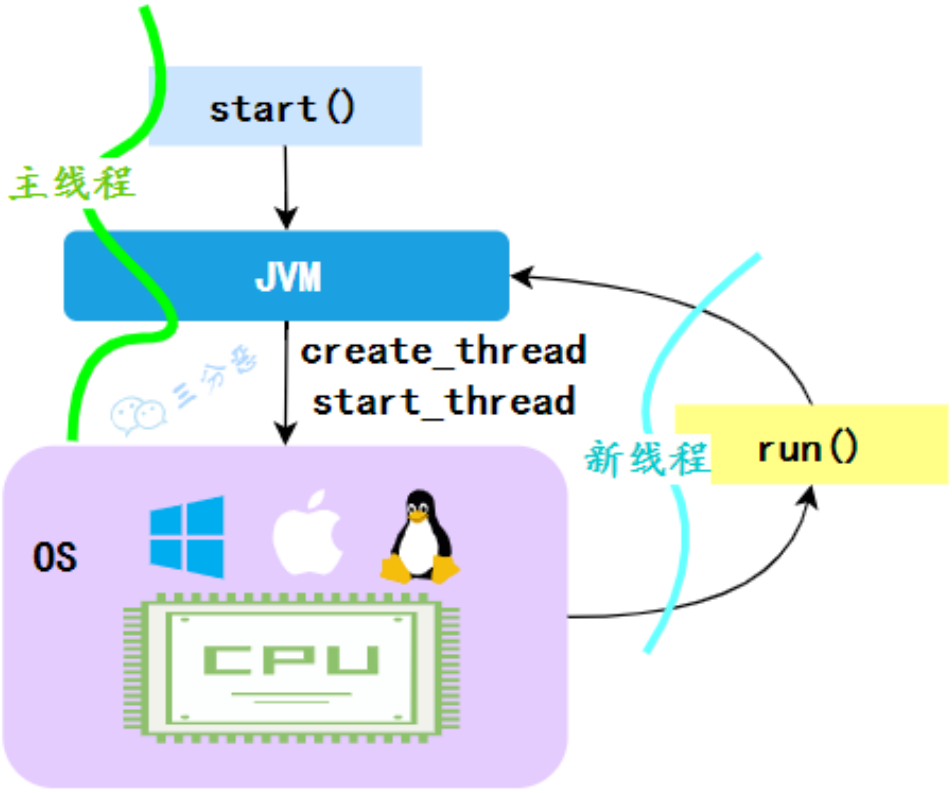
- 实现Callable接口，重写call()方法，这种方式可以通过FutureTask获取任务执行的返回值

```
public class CallerTask implements Callable<String> {
    public String call() throws Exception {
        return "Hello,i am running!";
    }
```

```
    public static void main(String[] args) {
        //创建异步任务
        FutureTask<String> task=new FutureTask<String>(new CallerTask());
        //启动线程
        new Thread(task).start();
        try {
            //等待执行完成，并获取返回结果
            String result=task.get();
            System.out.println(result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

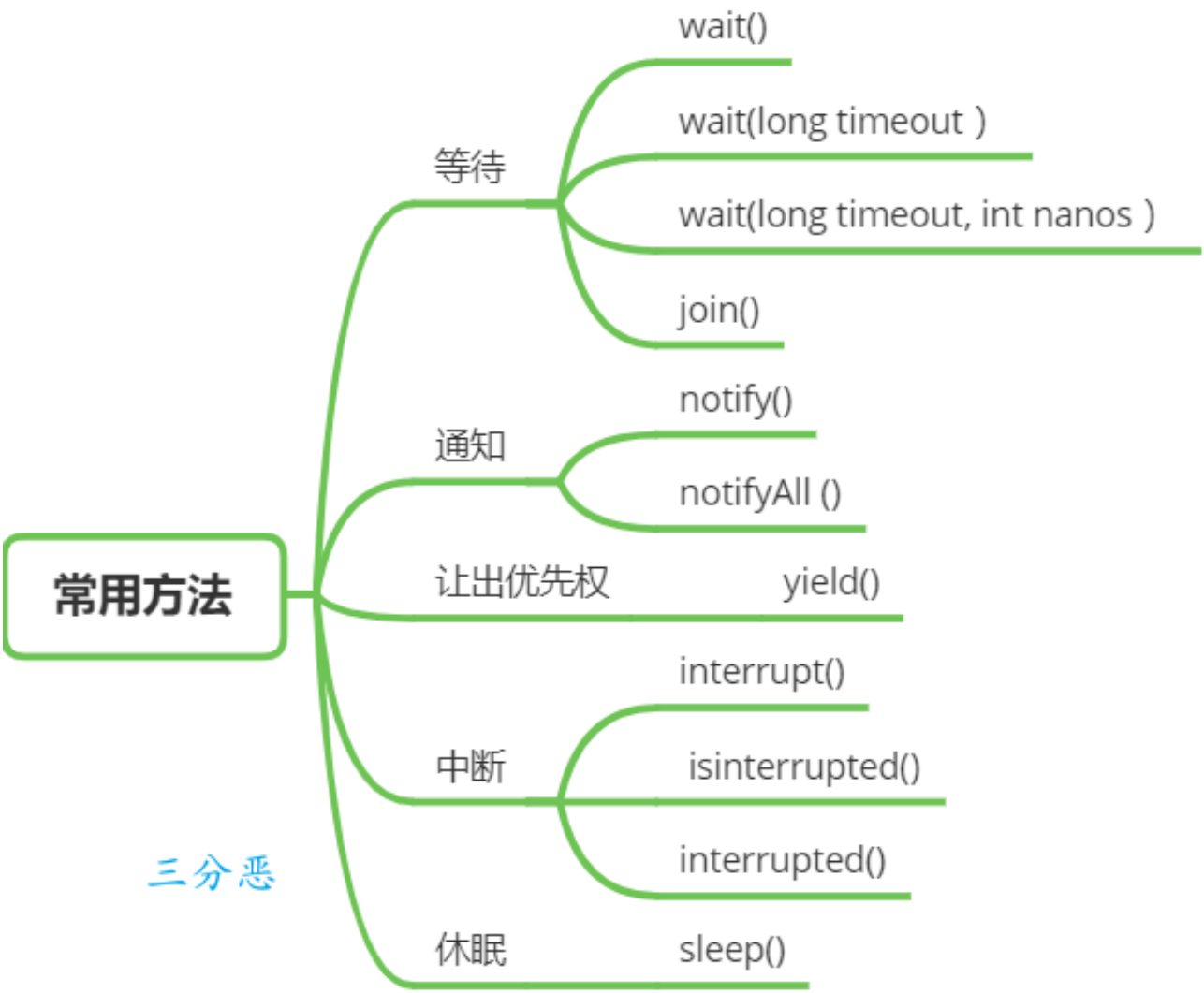
4.为什么调用start()方法时会执行run()方法，那怎么不直接调用run()方法?

JVM执行start方法，会先创建一条线程，由创建出来的新线程去执行thread的run方法，这才起到多线程的效果。



为什么我们不能直接调用**run()**方法？也很清楚， 如果直接调用Thread的run()方法，那么run方法还是运行在主线程中，相当于顺序执行，就起不到多线程的效果。

5.线程有哪些常用的调度方法?



线程等待与通知

在Object类中有一些函数可以用于线程的等待与通知。

- wait()：当一个线程A调用一个共享变量的 wait()方法时， 线程A会被阻塞挂起， 发生下面几种情况才会返回 ：
  - （1） 线程A调用了共享对象 notify()或者 notifyAll()方法；
  - （2）其他线程调用了线程A的 interrupt() 方法，线程A抛出InterruptedException异常返回。
- wait(long timeout) ： 这个方法相比 wait() 方法多了一个超时参数，它的不同之处在于，如果线程A调用共享对象的wait(long timeout)方法后，没有在指定的 timeout ms时间内被其它线程唤醒，那么这个方法还是会因为超时而返回。
- wait(long timeout, int nanos)，其内部调用的是 wait(long timout） 函数。

上面是线程等待的方法，而唤醒线程主要是下面两个方法：

- notify()：一个线程A调用共享对象的 notify() 方法后，会唤醒一个在这个共享变量上调用 wait 系列方法后被挂起的线程。 一个共享变量上可能会有多个线程在等待，具体唤醒哪个等待的线程是随机的。
- notifyAll()： 不同于在共享变量上调用 notify() 函数会唤醒被阻塞到该共享变量上的一个线程，notifyAll()方法则会唤醒所有在该共享变量上由于调用 wait 系列方法而被挂起的线程。

Thread类也提供了一个方法用于等待的方法：

- join()：如果一个线程A执行了thread.join()语句，其含义是：当前线程A等待thread线程终止之后才

  从thread.join()返回。

线程休眠

- sleep(long millis) :Thread类中的静态方法，当一个执行中的线程A调用了Thread 的sleep方法后，线程A会暂时让出指定时间的执行权，但是线程A所拥有的监视器资源，比如锁还是持有不让出的。指定的睡眠时间到了后该函数会正常返回，接着参与 CPU 的调度，获取到 CPU 资源后就可以继续运行。

让出优先权

- yield()： Thread类中的静态方法，当一个线程调用 yield 方法时，实际就是在暗示线程调度器当前线程请求让出自己的CPU ，但是线程调度器可以无条件忽略这个暗示。

线程中断

Java 中的线程中断是一种线程间的协作模式，通过设置线程的中断标志并不能直接终止该线程的执行，而是被中断的线程根据中断状态自行处理。

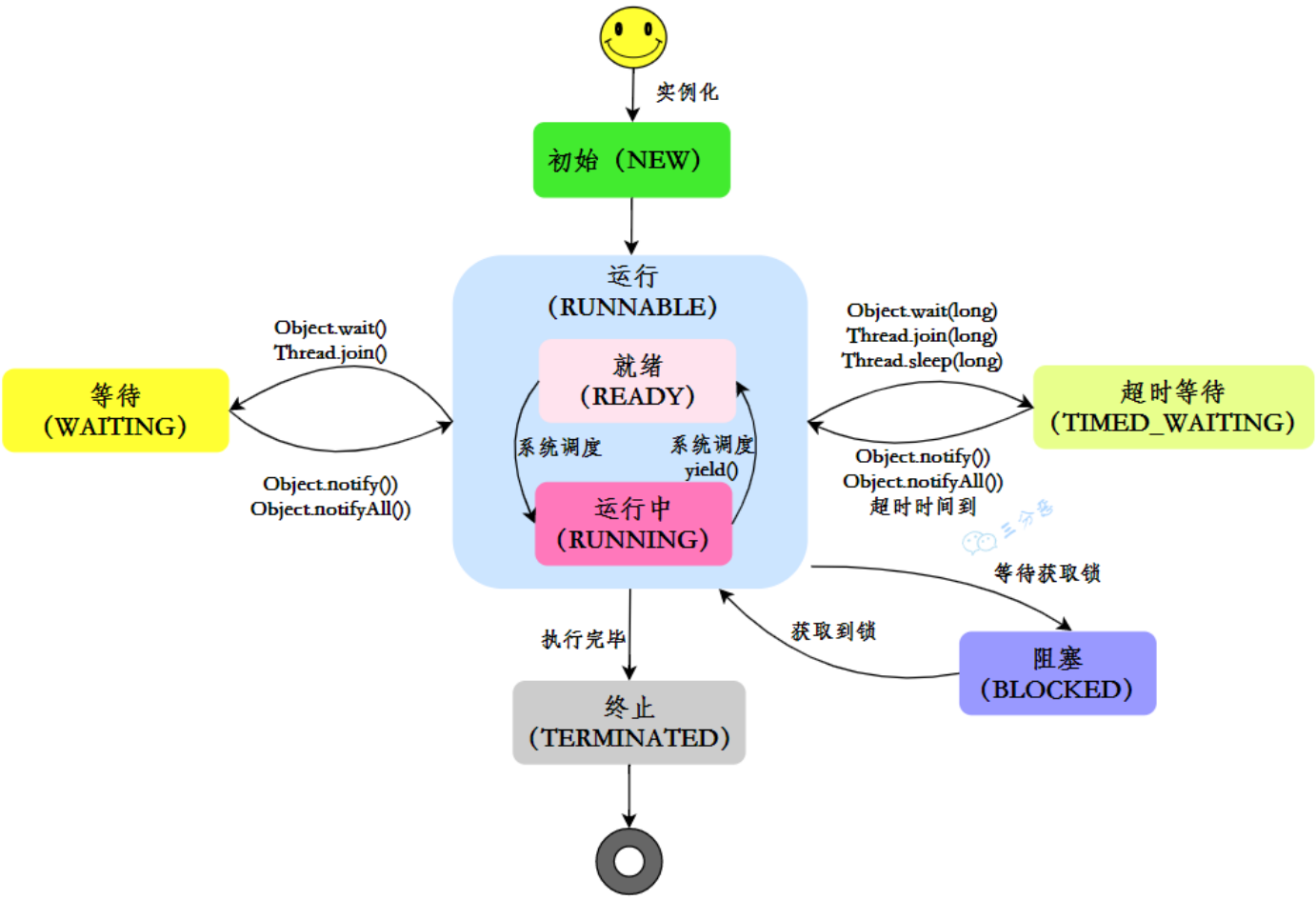
- void interrupt()： 中断线程，例如，当线程A运行时，线程B可以调用线程interrupt() 方法来设置线程的中断标志为true 并立即返回。设置标志仅仅是设置标志，线程A实际并没有被中断， 会继续往下执行。
- boolean isInterrupted() 方法： 检测当前线程是否被中断。
- boolean interrupted() 方法： 检测当前线程是否被中断，与 isInterrupted 不同的是，该方法如果发现当前线程被中断，则会清除中断标志。

6.线程有几种状态?

在Java中，线程共有六种状态：

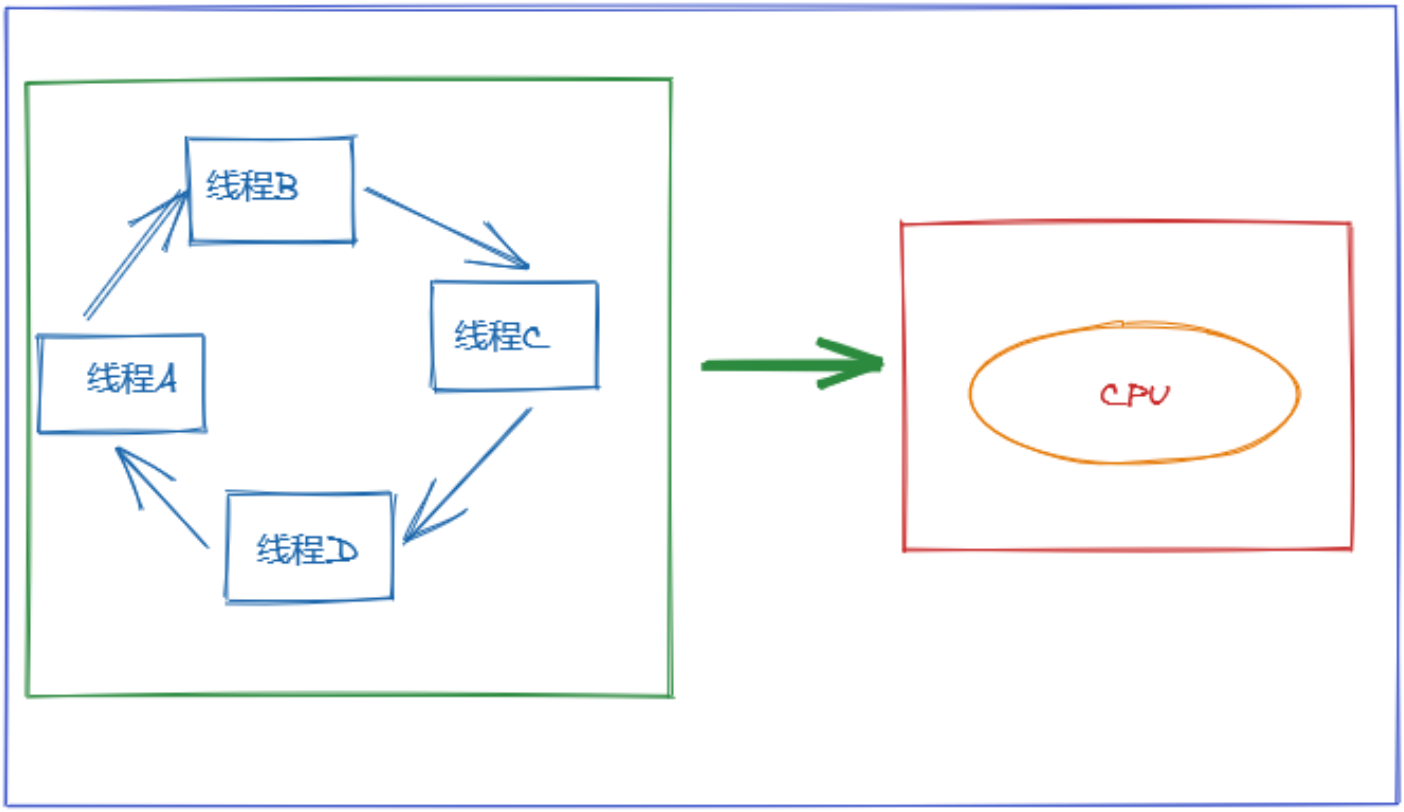
状态	说明
NEW	初始状态：线程被创建，但还没有调用start()方法
RUNNABLE	运行状态：Java线程将操作系统中的就绪和运行两种状态笼统的称作“运行”
BLOCKED	阻塞状态：表示线程阻塞于锁
WAITING	等待状态：表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态：该状态不同于 WAITIND，它是可以在指定的时间自行返回的
TERMINATED	终止状态：表示当前线程已经执行完毕

线程在自身的生命周期中，并不是固定地处于某个状态，而是随着代码的执行在不同的状态之间进行切换，Java线程状态变化如图示：

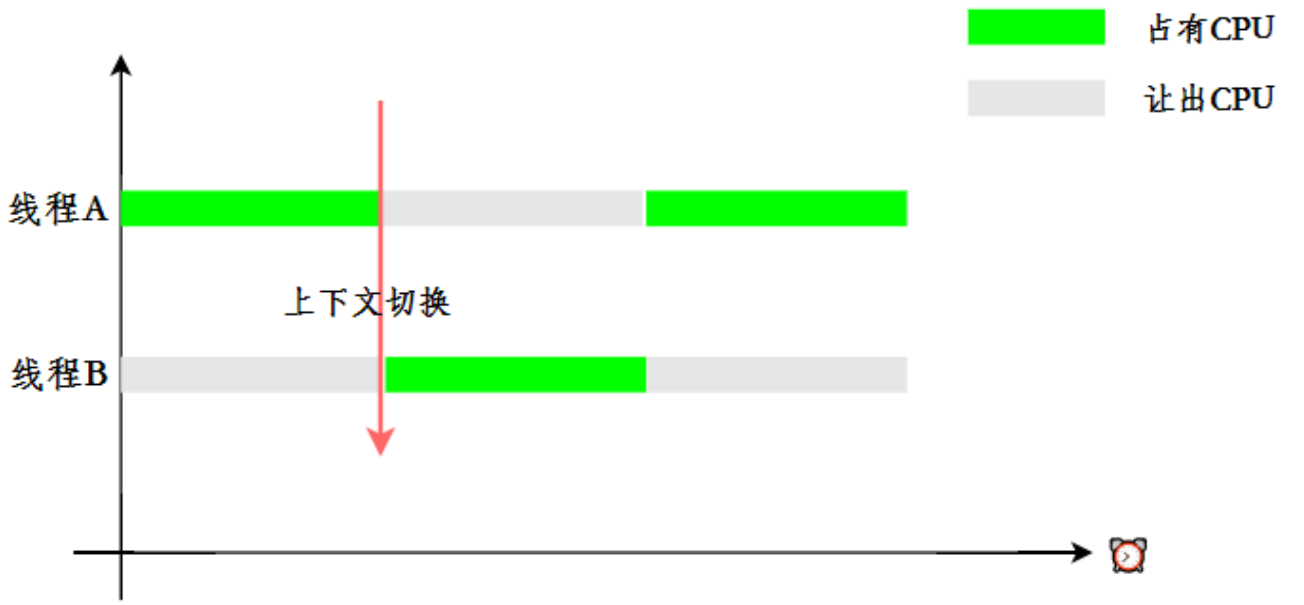


7.什么是线程上下文切换?

使用多线程的目的是为了充分利用CPU，但是我们知道，并发其实是一个CPU来应付多个线程。



为了让用户感觉多个线程是在同时执行的， CPU 资源的分配采用了时间片轮转也就是给每个线程分配一个时间片，线程在时间片内占用 CPU 执行任务。当线程使用完时间片后，就会处于就绪状态并让出 CPU 让其他线程占用，这就是上下文切换。



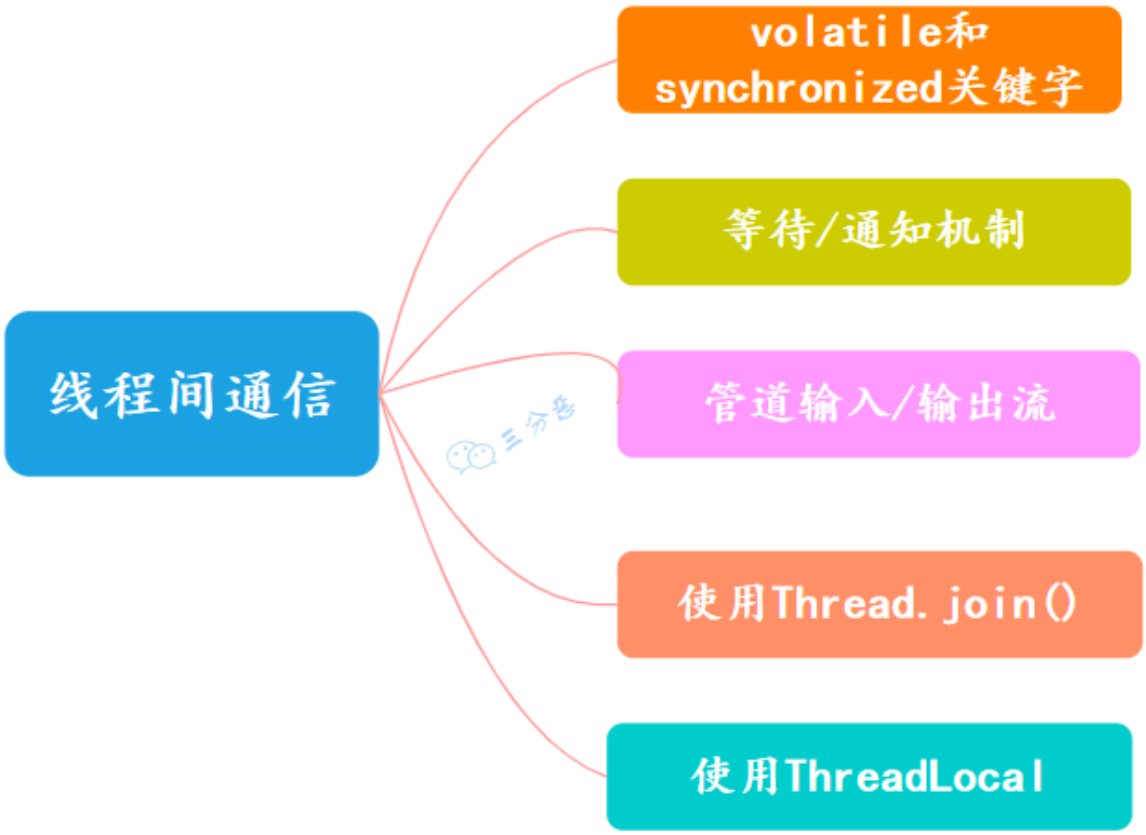
8.守护线程了解吗?

Java中的线程分为两类，分别为 daemon 线程（守护线程）和 user 线程（用户线程）。

在JVM 启动时会调用 main 函数，main函数所在的线程就是一个用户线程。其实在 JVM 内部同时还启动了很多守护线程， 比如垃圾回收线程。

那么守护线程和用户线程有什么区别呢？区别之一是当最后一个非守护线程束时， JVM会正常退出，而不管当前是否存在守护线程，也就是说守护线程是否结束并不影响 JVM退出。换而言之，只要有一个用户线程还没结束，正常情况下JVM就不会退出。

9.线程间有哪些通信方式?



- **volatile和synchronized关键字**

关键字volatile可以用来修饰字段（成员变量），就是告知程序任何对该变量的访问均需要从共享内存中获取，而对它的改变必须同步刷新回共享内存，它能保证所有线程对变量访问的可见性。

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。

- **等待/通知机制**

可以通过Java内置的等待/通知机制（wait()/notify()）实现一个线程修改一个对象的值，而另一个线程感知到了变化，然后进行相应的操作。

- **管道输入/输出流**

管道输入/输出流和普通的文件输入/输出流或者网络输入/输出流不同之处在于，它主要用于线程之间的数据传输，而传输的媒介为内存。

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。

- **使用Thread.join()**

如果一个线程A执行了thread.join()语句，其含义是：当前线程A等待thread线程终止之后才从thread.join()返回。。线程Thread除了提供join()方法之外，还提供了join(long millis)和join(long millis,int nanos)两个具备超时特性的方法。

- 使用**ThreadLocal**

ThreadLocal，即线程变量，是一个以ThreadLocal对象为键、任意对象为值的存储结构。这个结构被附带在线程上，也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值。

可以通过set(T)方法来设置一个值，在当前线程下再通过get()方法获取到原先设置的值。

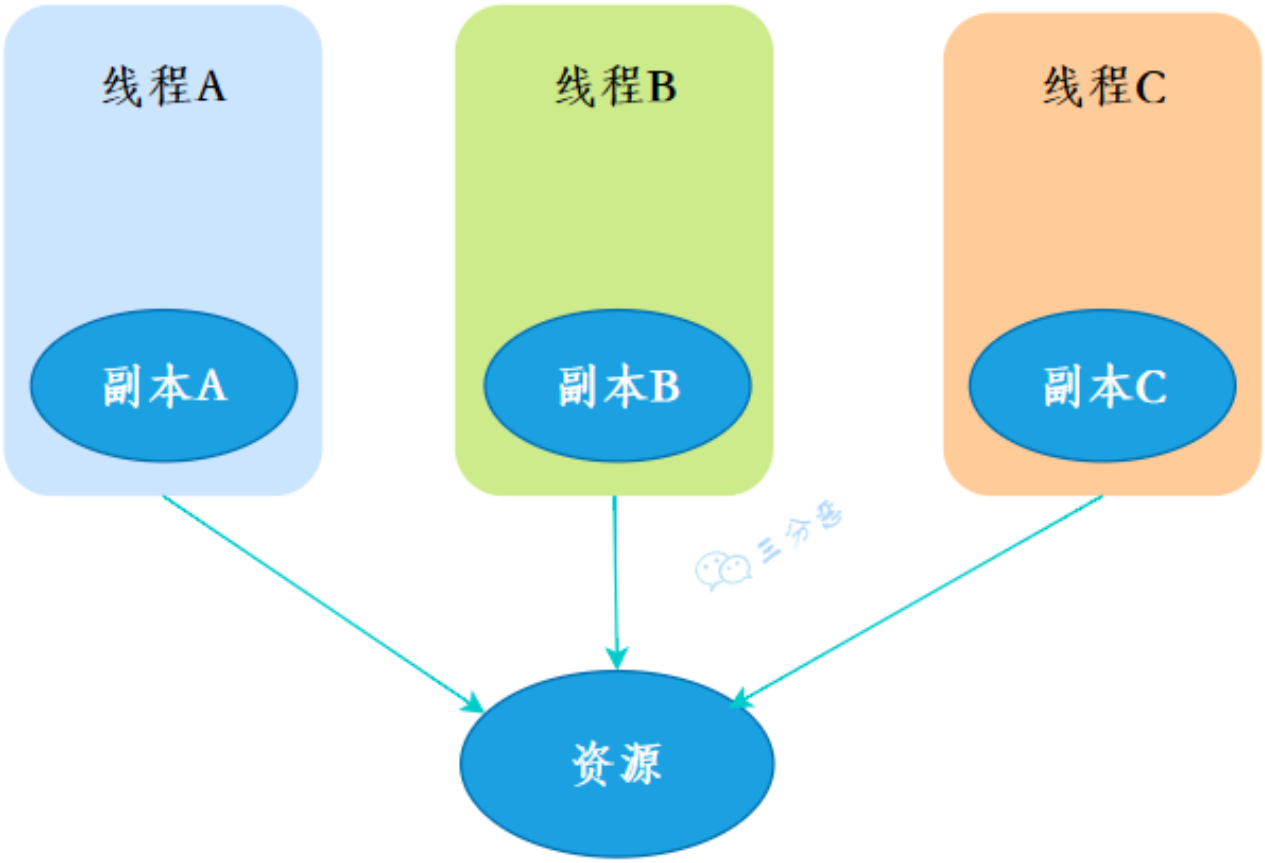
> 关于多线程，其实很大概率还会出一些笔试题，比如交替打印、银行转账、生产消费模型等等，后面老三会单独出一期来盘点一下常见的多线程笔试题。

ThreadLocal

ThreadLocal其实应用场景不是很多，但却是被炸了千百遍的面试老油条，涉及到多线程、数据结构、JVM，可问的点比较多，一定要拿下。

10.ThreadLocal是什么？

ThreadLocal，也就是线程本地变量。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

创建了一个ThreadLoca变量localVariable，任何一个线程都能并发访问localVariable。

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

- 写入

线程可以在任何地方使用localVariable，写入变量。

```
localVariable.set("鄙人三某");
```

- 读取

线程在任何地方读取的都是它写入的变量。

```
localVariable.get();
```

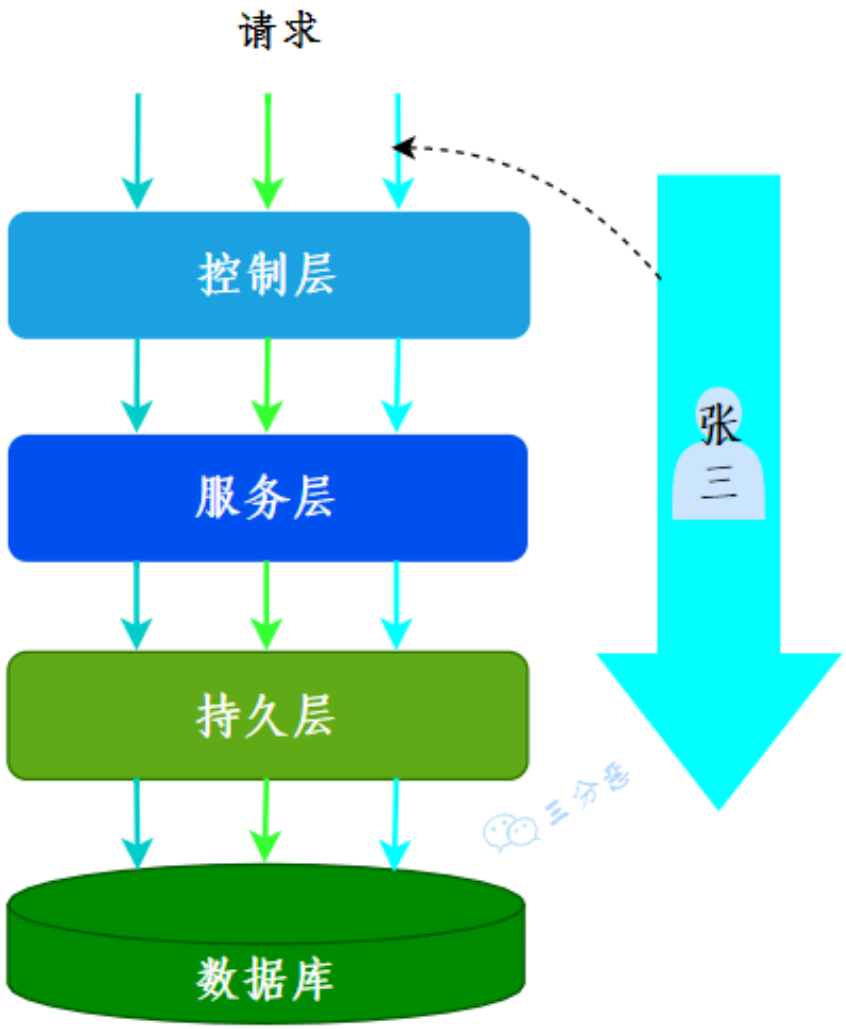
11.你在工作中用到过ThreadLocal吗？

有用到过的，用来做用户信息上下文的存储。

我们的系统应用是一个典型的MVC架构，登录后的用户每次访问接口，都会在请求头中携带一个token，在控制层可以根据这个token，解析出用户的基本信息。那么问题来了，假如在服务层和持久层都要用到用户信息，比如rpc调用、更新用户获取等等，那应该怎么办呢？

一种办法是显式定义用户相关的参数，比如账号、用户名……这样一来，我们可能需要大面积地修改代码，多少有点瓜皮，那该怎么办呢？

这时候我们就可以用到ThreadLocal，在控制层拦截请求把用户信息存入ThreadLocal，这样我们在任何一个地方，都可以取出ThreadLocal中存的用户数据。



很多其它场景的cookie、session等等数据隔离也都可以通过ThreadLocal去实现。

我们常用的数据库连接池也用到了ThreadLocal：

- 数据库连接池的连接交给ThreadLoca进行管理，保证当前线程的操作都是同一个Connnection。

12.ThreadLocal怎么实现的呢？

我们看一下ThreadLocal的set(T)方法，发现先获取到当前线程，再获取`ThreadLocalMap`，然后把元素存到这个map中。

```
    public void set(T value) {
        //获取当前线程
        Thread t = Thread.currentThread();
        //获取ThreadLocalMap
        ThreadLocalMap map = getMap(t);
        //讲当前元素存入map
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }
```

ThreadLocal实现的秘密都在这个`ThreadLocalMap`了，可以Thread类中定义了一个类型为`ThreadLocal.ThreadLocalMap`的成员变量`threadLocals`。

```
public class Thread implements Runnable {
  //ThreadLocal.ThreadLocalMap是Thread的属性
  ThreadLocal.ThreadLocalMap threadLocals = null;
}
```

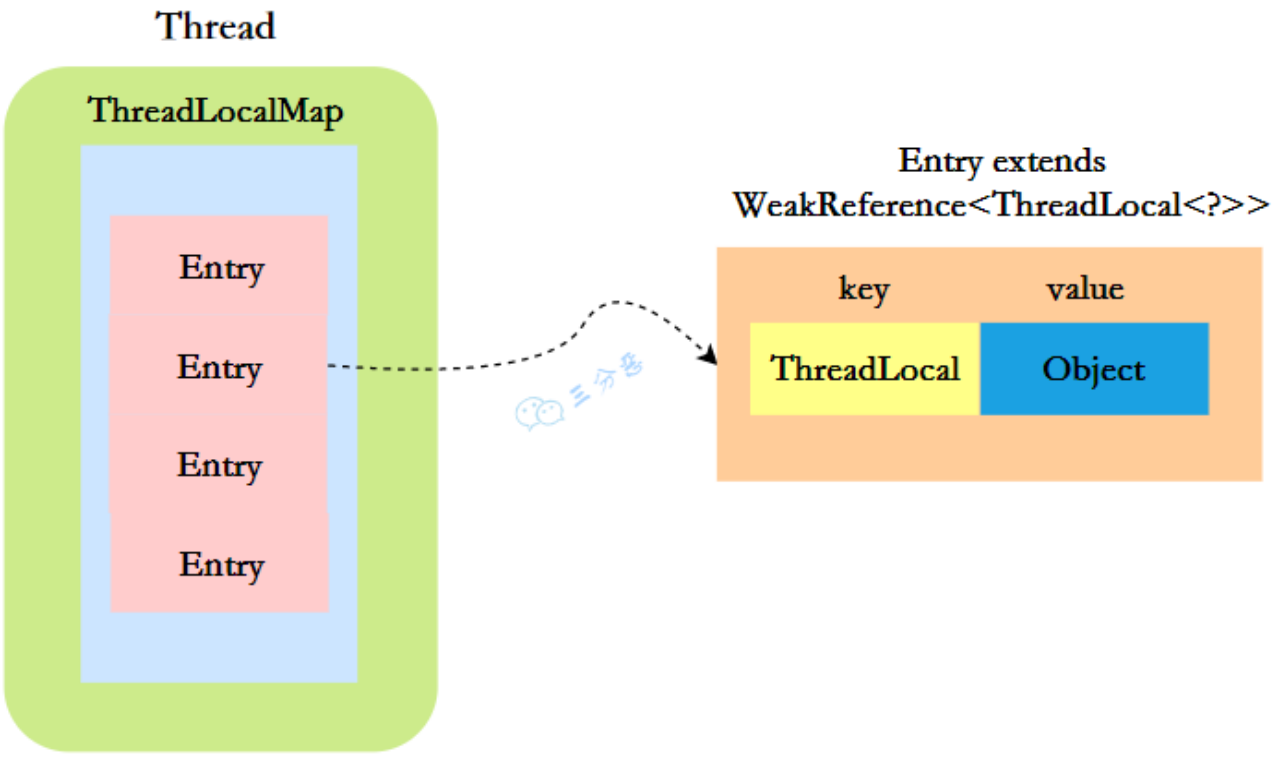
ThreadLocalMap既然被称为Map，那么毫无疑问它是<key,value>型的数据结构。我们都知道map的本质是一个个<key,value>形式的节点组成的数组，那ThreadLocalMap的节点是什么样的呢？

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

    //节点类
    Entry(ThreadLocal<?> k, Object v) {
        //key赋值
        super(k);
        //value赋值
        value = v;
    }
}
```

这里的节点，key可以简单低视作ThreadLocal，value为代码中放入的值，当然实际上key并不是ThreadLocal本身，而是它的一个**弱引用**，可以看到Entry的key继承了 WeakReference（弱引用），再来看一下key怎么赋值的：

```
public WeakReference(T referent) {
    super(referent);
}
```

key的赋值，使用的是WeakReference的赋值。



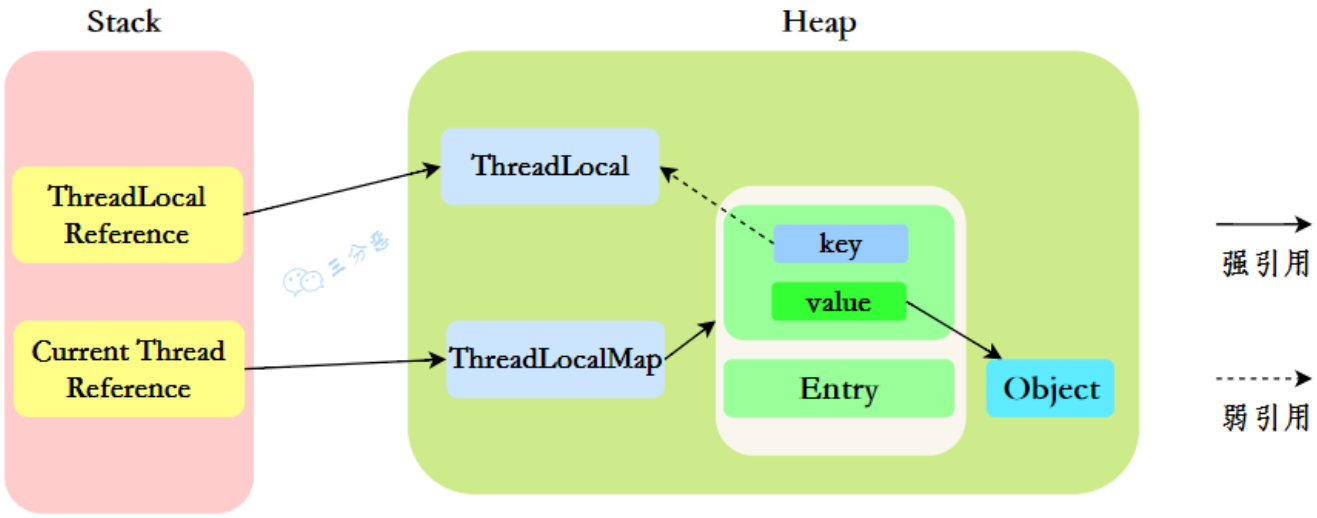
所以，怎么回答ThreadLocal原理？要答出这几个点：

- Thread类有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，每个线程都有一个属于自己的ThreadLocalMap。
- ThreadLocalMap内部维护着Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal的弱引用，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本身不存储值，它只是作为一个key来让线程往ThreadLocalMap里存取值。

13.ThreadLocal 内存泄露是怎么回事？

我们先来分析一下使用ThreadLocal时的内存，我们都知道，在JVM中，栈内存线程私有，存储了对象的引用，堆内存线程共享，存储了对象实例。

所以呢，栈中存储了ThreadLocal、Thread的引用，堆中存储了它们的具体实例。



ThreadLocalMap中使用的 key 为 ThreadLocal 的弱引用。

> “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。”

那么现在问题就来了，弱引用很容易被回收，如果ThreadLocal（ThreadLocalMap的Key）被垃圾回收器回收了，但是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。

> 那怎么解决内存泄漏问题呢？

很简单，使用完ThreadLocal后，及时调用remove()方法释放内存空间。

```
ThreadLocal<String> localVariable = new ThreadLocal();
try {
    localVariable.set("鄙人三某");
    ……
} finally {
    localVariable.remove();
}
```

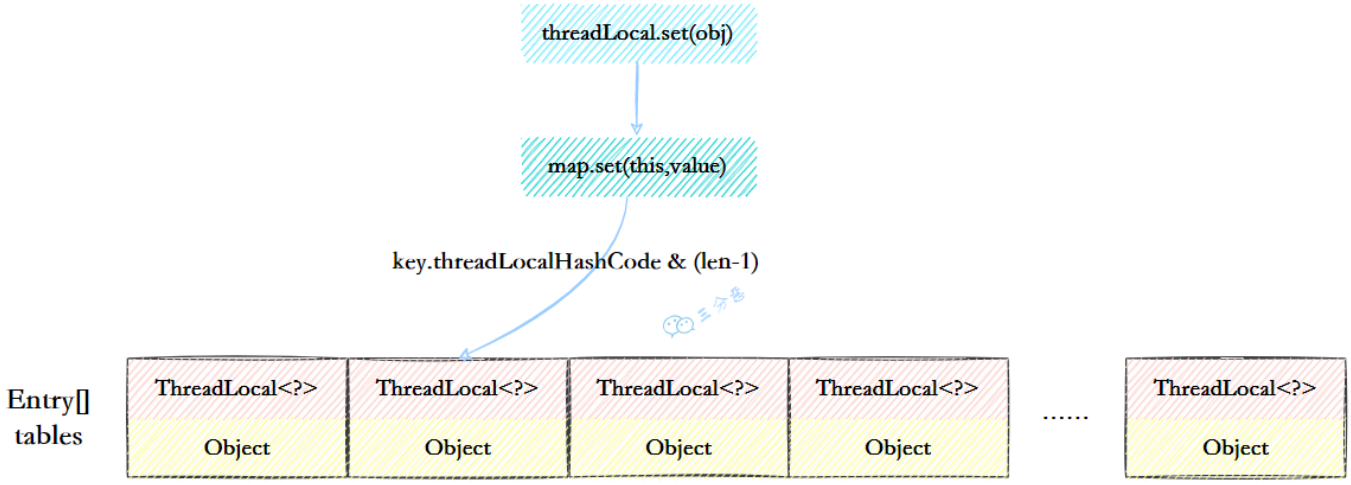
> 那为什么key还要设计成弱引用？

key设计成弱引用同样是为了防止内存泄漏。

假如key被设计成强引用，如果ThreadLocal Reference被销毁，此时它指向ThreadLoca的强引用就没有了，但是此时key还强引用指向ThreadLoca，就会导致ThreadLocal不能被回收，这时候就发生了内存泄漏的问题。

14.ThreadLocalMap的结构了解吗?

ThreadLocalMap虽然被叫做Map，其实它是没有实现Map接口的，但是结构还是和HashMap比较类似的，主要关注的是两个要素：`元素数组`和`散列方法`。



- 元素数组

  一个table数组，存储Entry类型的元素，Entry是ThreaLocal弱引用作为key，Object作为value的结构。

```
private Entry[] table;
```

- 散列方法

  散列方法就是怎么把对应的key映射到table数组的相应下标，ThreadLocalMap用的是哈希取余法，取出key的threadLocalHashCode，然后和table数组长度减一&运算（相当于取余）。

```
int i = key.threadLocalHashCode & (table.length - 1);
```

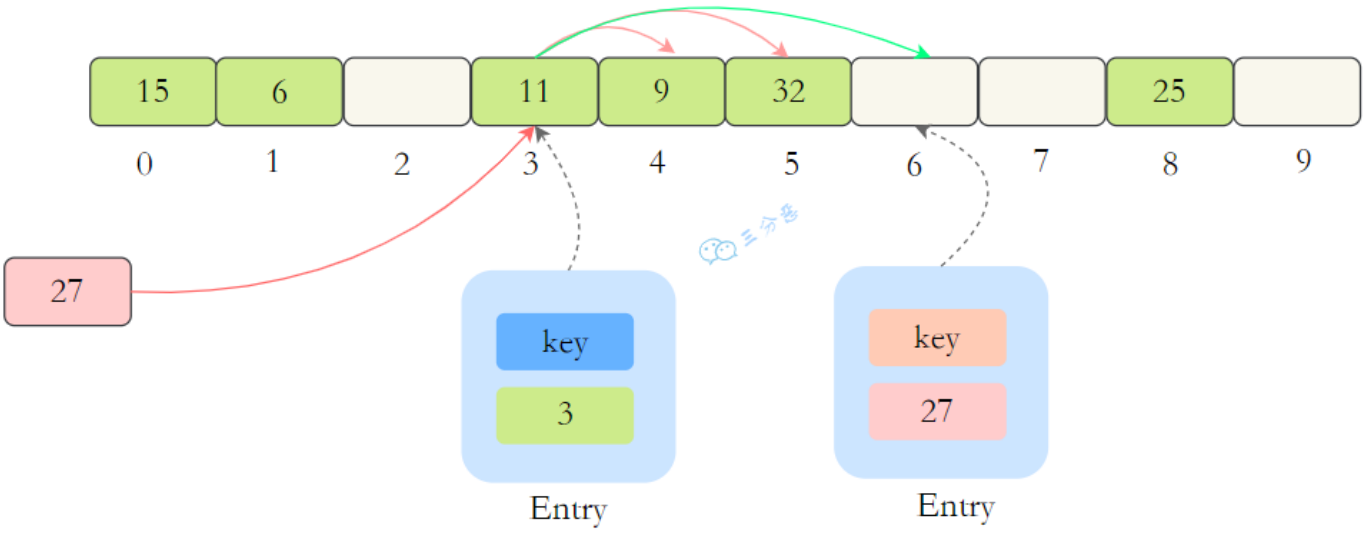
这里的threadLocalHashCode计算有点东西，每创建一个ThreadLocal对象，它就会新增`0x61c88647`，这个值很特殊，它是斐波那契数 也叫 黄金分割数。`hash`增量为 这个数字，带来的好处就是 `hash` 分布非常均匀。

```
private static final int HASH_INCREMENT = 0x61c88647;

private static int nextHashCode() {
    return nextHashCode.getAndAdd(HASH_INCREMENT);
}
```

15.ThreadLocalMap怎么解决Hash冲突的?

我们可能都知道HashMap使用了链表来解决冲突，也就是所谓的链地址法。

ThreadLocalMap没有使用链表，自然也不是用链地址法来解决冲突了，它用的是另外一种方式——开放定址法。开放定址法是什么意思呢？简单来说，就是这个坑被人占了，那就接着去找空着的坑。



如上图所示，如果我们插入一个value=27的数据，通过 hash计算后应该落入第 4 个槽位中，而槽位 4 已经有了 Entry数据，而且Entry数据的key和当前不相等。此时就会线性向后查找，一直找到 Entry 为 null的槽位才会停止查找，把元素放到空的槽中。

在get的时候，也会根据ThreadLocal对象的hash值，定位到table中的位置，然后判断该槽位Entry对象中的key是否和get的key一致，如果不一致，就判断下一个位置。

16.ThreadLocalMap扩容机制了解吗?

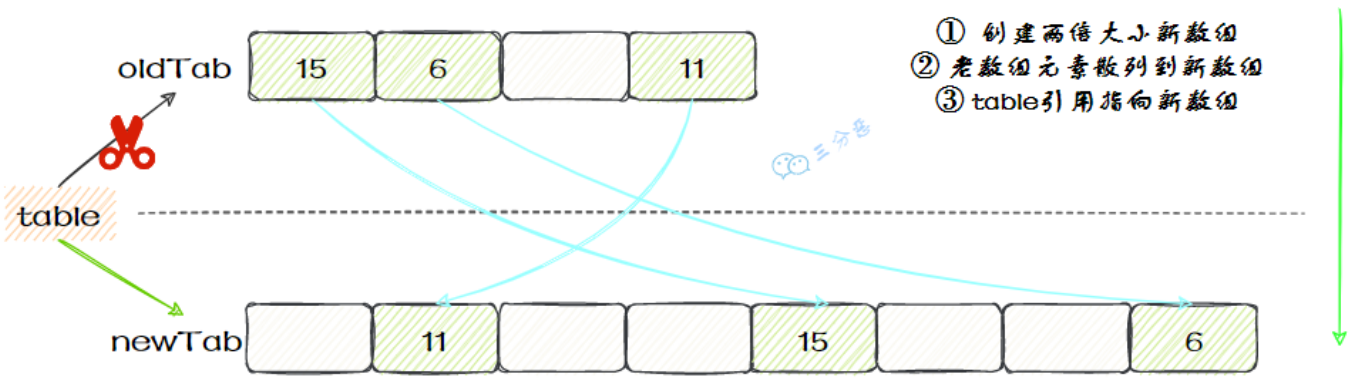
在ThreadLocalMap.set()方法的最后，如果执行完启发式清理工作后，未清理到任何数据，且当前散列数组中`Entry`的数量已经达到了列表的扩容阈值`(len*2/3)`，就开始执行`rehash()`逻辑：

```
if (!cleanSomeSlots(i, sz) && sz >= threshold)
    rehash();
```

再着看rehash()具体实现：这里会先去清理过期的Entry，然后还要根据条件判断`size >= threshold - threshold / 4` 也就是`size >= threshold* 3/4`来决定是否需要扩容。

```
private void rehash() {
    //清理过期Entry
    expungeStaleEntries();

    //扩容
    if (size >= threshold - threshold / 4)
        resize();
}

//清理过期Entry
private void expungeStaleEntries() {
    Entry[] tab = table;
    int len = tab.length;
    for (int j = 0; j < len; j++) {
        Entry e = tab[j];
        if (e != null && e.get() == null)
            expungeStaleEntry(j);
    }
}
```

接着看看具体的`resize()`方法，扩容后的`newTab`的大小为老数组的两倍，然后遍历老的table数组，散列方法重新计算位置，开放地址解决冲突，然后放到新的`newTab`，遍历完成之后，`oldTab`中所有的`entry`数据都已经放入到`newTab`中了，然后table引用指向`newTab`



具体代码：

```
        private void resize() {
            Entry[] oldTab = table;
            int oldLen = oldTab.length;
            int newLen = oldLen * 2;
            Entry[] newTab = new Entry[newLen];
            int count = 0;

            for (int j = 0; j < oldLen; ++j) {
                Entry e = oldTab[j];
                if (e != null) {
                    ThreadLocal<?> k = e.get();
                    if (k == null) {
                        e.value = null; // Help the GC
                    } else {
                        int h = k.threadLocalHashCode & (newLen - 1);
                        while (newTab[h] != null)
                            h = nextIndex(h, newLen);
                        newTab[h] = e;
                        count++;
                    }
                }
            }

            setThreshold(newLen);
            size = count;
            table = newTab;
        }
```

17.父子线程怎么共享数据？

父线程能用ThreadLocal来给子线程传值吗？毫无疑问，不能。那该怎么办？

这时候可以用到另外一个类——`InheritableThreadLocal`。

使用起来很简单，在主线程的InheritableThreadLocal实例设置值，在子线程中就可以拿到了。

```
public class InheritableThreadLocalTest {

    public static void main(String[] args) {
        final ThreadLocal threadLocal = new InheritableThreadLocal();
        // 主线程
        threadLocal.set("不擅技术");
        //子线程
        Thread t = new Thread() {
            @Override
            public void run() {
                super.run();
                System.out.println("鄙人三某，" + threadLocal.get());
            }
        };
        t.start();
    }
}
```

那原理是什么呢？

原理很简单，在Thread类里还有另外一个变量：

```
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

在Thread.init的时候，如果父线程的`inheritableThreadLocals`不为空，就把它赋给当前线程（子线程）的`inheritableThreadLocals`。

```
if (inheritThreadLocals && parent.inheritableThreadLocals != null)
    this.inheritableThreadLocals =
        ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
```

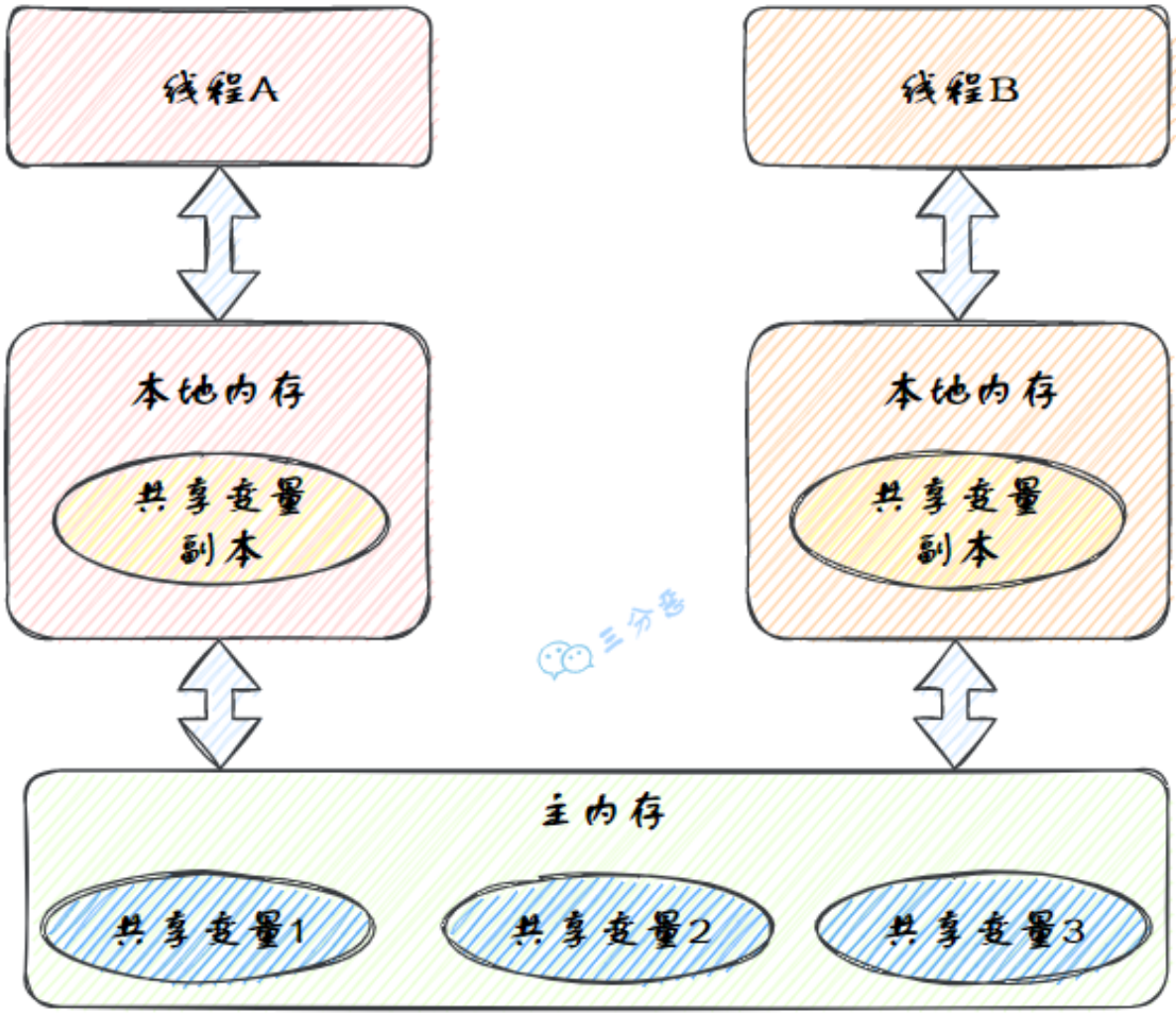
Java内存模型

18.说一下你对Java内存模型（JMM）的理解？

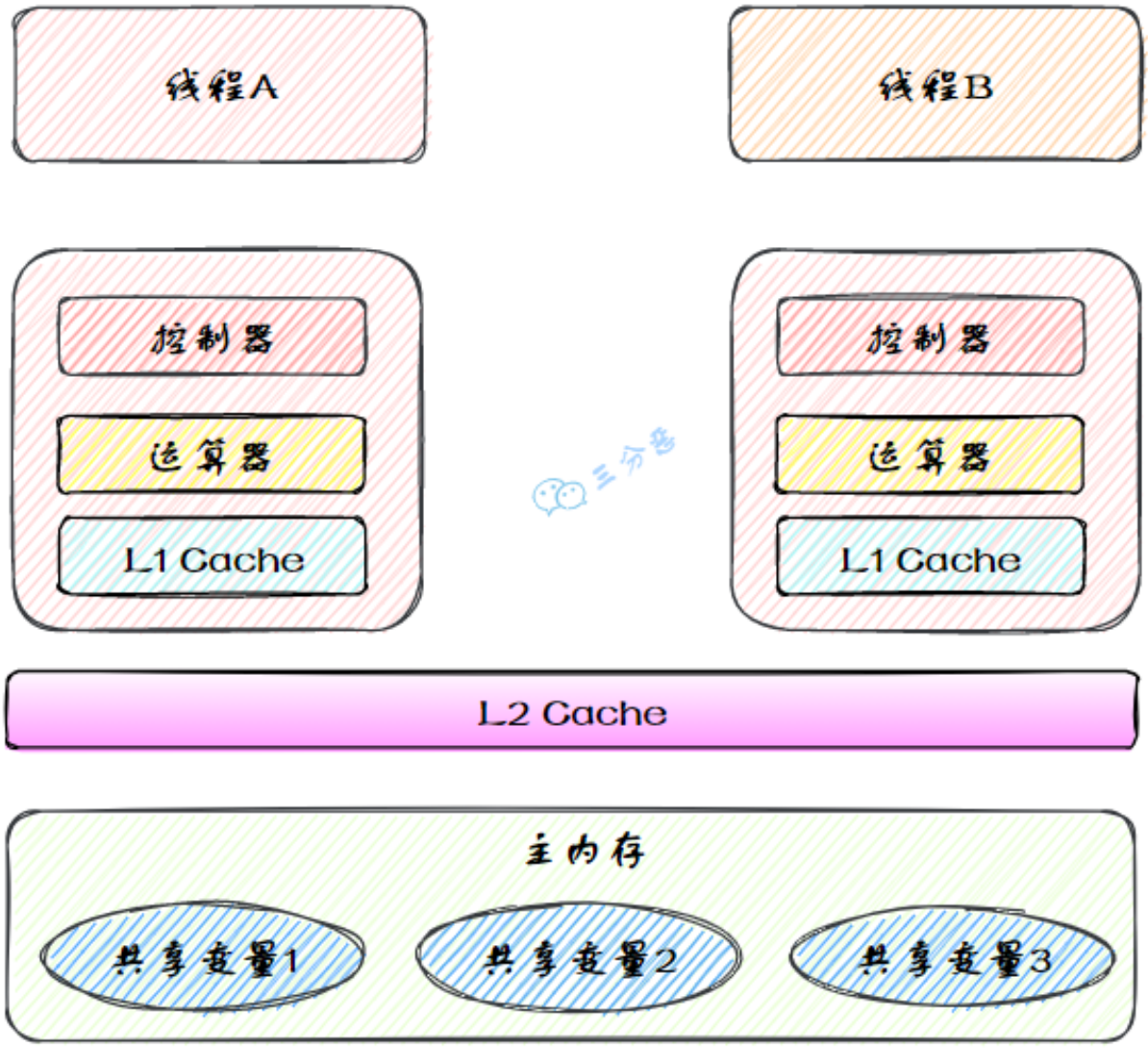
Java内存模型（Java Memory Model，JMM），是一种抽象的模型，被定义出来屏蔽各种硬件和操作系统的内存访问差异。

JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在`主内存`（Main Memory）中，每个线程都有一个私有的`本地内存`（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。

Java内存模型的抽象图：



本地内存是JMM的 一个抽象概念，并不真实存在。它其实涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。



图里面的是一个双核 CPU 系统架构，每个核有自己的控制器和运算器，其中控制器包含一组寄存器和操作控制器，运算器执行算术逻辅运算。每个核都有自己的一级缓存，在有些架构里面还有一个所有 CPU 共享的二级缓存。 那么 Java 内存模型里面的工作内存，就对应这里的 L1 缓存或者 L2 缓存或者 CPU 寄存器。

19.说说你对原子性、可见性、有序性的理解?

原子性、有序性、可见性是并发编程中非常重要的基础概念，JMM的很多技术都是围绕着这三大特性展开。

- 原子性：原子性指的是一个操作是不可分割、不可中断的，要么全部执行并且执行的过程不会被任何因素打断，要么就全不执行。
- 可见性：可见性指的是一个线程修改了某一个共享变量的值时，其它线程能够立即知道这个修改。

- 有序性：有序性指的是对于一个线程的执行代码，从前往后依次执行，单线程下可以认为程序是有序的，但是并发时有可能会发生指令重排。

分析下面几行代码的原子性？

```
int i = 2;
int j = i;
i++;
i = i + 1;
```

- 第1句是基本类型赋值，是原子性操作。
- 第2句先读i的值，再赋值到j，两步操作，不能保证原子性。
- 第3和第4句其实是等效的，先读取i的值，再+1，最后赋值到i，三步操作了，不能保证原子性。

原子性、可见性、有序性都应该怎么保证呢？

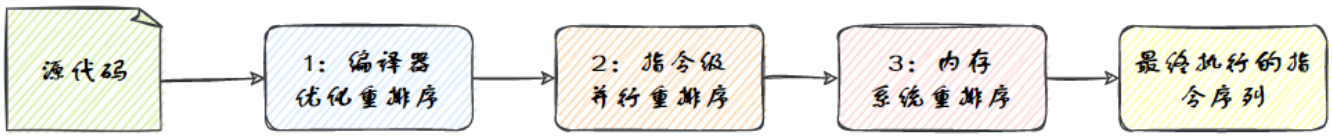
- 原子性：JMM只能保证基本的原子性，如果要保证一个代码块的原子性，需要使用 `synchronized` 。
- 可见性：Java是利用 `volatile` 关键字来保证可见性的，除此之外， `final` 和 `synchronized` 也能保证可见性。
- 有序性： `synchronized` 或者 `volatile` 都可以保证多线程之间操作的有序性。

20.那说说什么是指令重排？

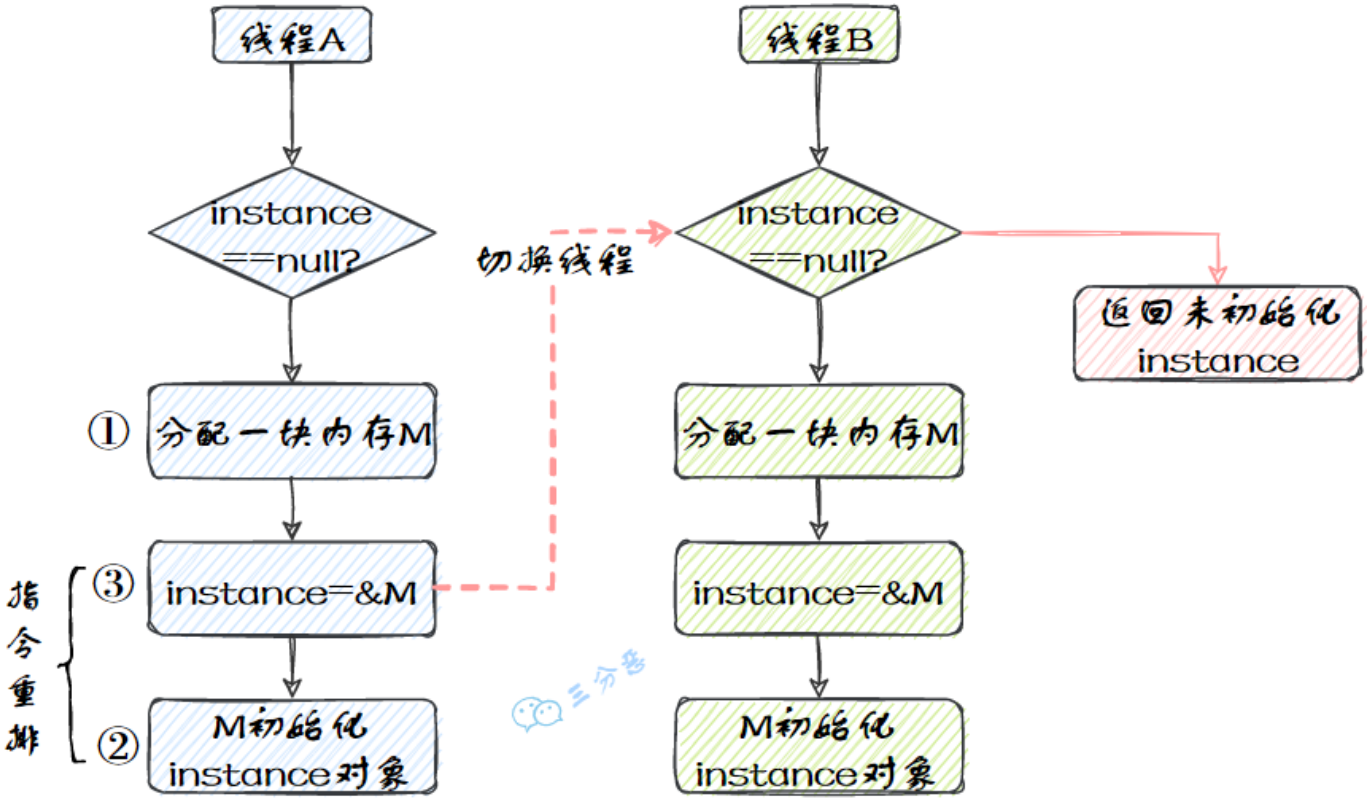
在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。重排序分3种类型。

1. 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism，ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应 机器指令的执行顺序。
3. 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从Java源代码到最终实际执行的指令序列，会分别经历下面3种重排序，如图：



我们比较熟悉的双重校验单例模式就是一个经典的指令重排的例子，`Singleton instance=new Singleton();` 对应的JVM指令分为三步：分配内存空间-->初始化对象--->对象指向分配的内存空间，但是经过了编译器的指令重排序，第二步和第三步就可能会重排序。



双重校验单例模式异常执行情形

JMM属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

21.指令重排有限制吗？happens-before了解吗？

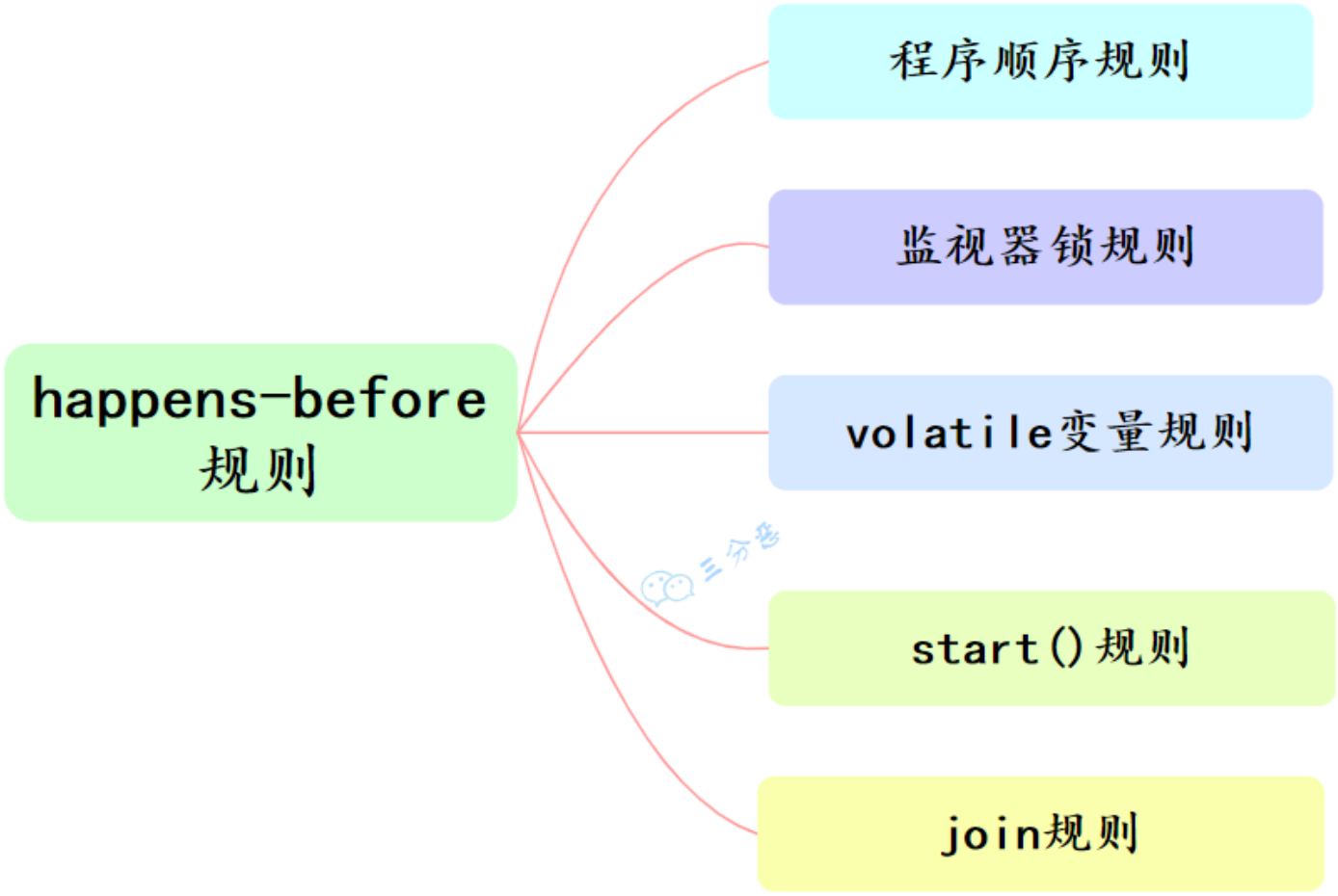
指令重排也是有一些限制的，有两个规则`happens-before`和`as-if-serial`来约束。

happens-before的定义：

- 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。

- 两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照 happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按happens-before关系来执行的结果一致，那么这种重排序并不非法

happens-before和我们息息相关的有六大规则：



- 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。
- 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。
- **volatile变量规则**：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。
- 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。
- **start()规则**：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。
- **join()规则**：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作 happens-before于线程A从ThreadB.join()操作成功返回。

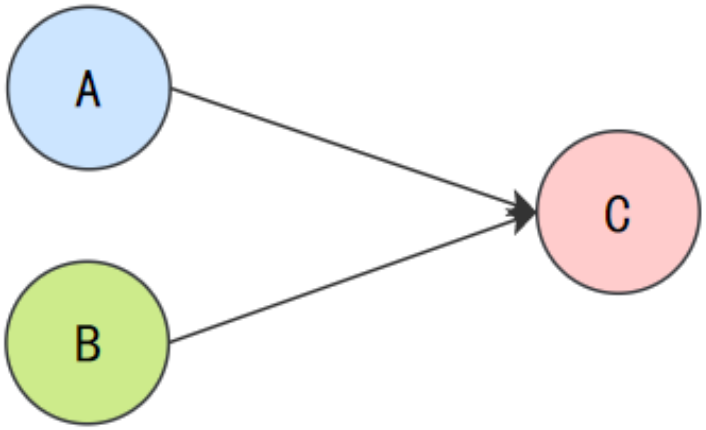
22.as-if-serial又是什么？单线程的程序一定是顺序的吗？

as-if-serial语义的意思是：不管怎么重排序（编译器和处理器为了提高并行度），单线程程序的执行结果不能被改变。编译器、runtime和处理器都必须遵守as-if-serial语义。

为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。为了具体说明，请看下面计算圆面积的代码示例。

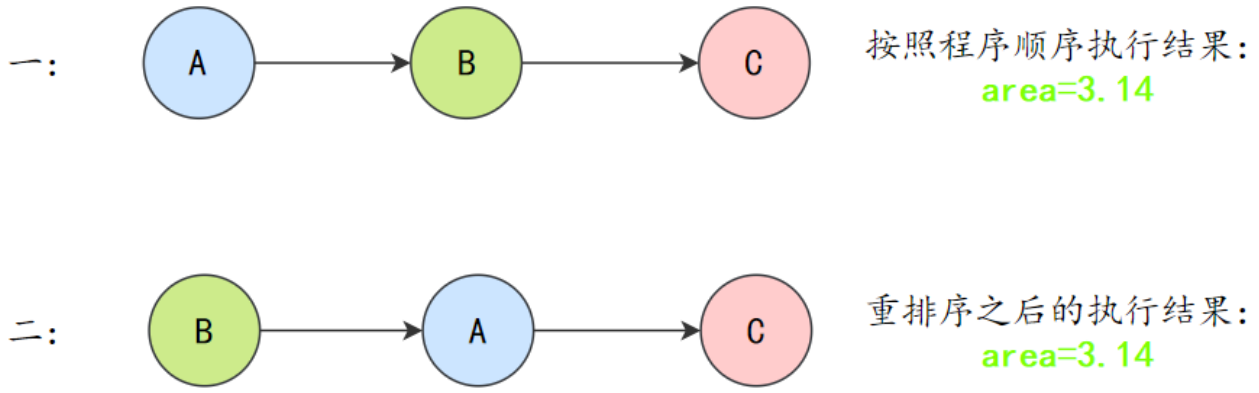
```
double pi = 3.14;    // A
double r = 1.0;    // B
double area = pi * r * r;    // C
```

上面3个操作的数据依赖关系：



A和C之间存在数据依赖关系，同时B和C之间也存在数据依赖关系。因此在最终执行的指令序列中，C不能被重排序到A和B的前面（C排到A和B的前面，程序的结果将会被改变）。但A和B之间没有数据依赖关系，编译器和处理器可以重排序A和B之间的执行顺序。

所以最终，程序可能会有两种执行顺序：



as-if-serial语义把单线程程序保护了起来，遵守as-if-serial语义的编译器、runtime和处理器共同编织了这么一个“楚门的世界”：单线程程序是按程序的“顺序”来执行的。as- if-serial语义使单线程情况下，我们不需要担心重排序的问题，可见性的问题。

23.volatile实现原理了解吗?

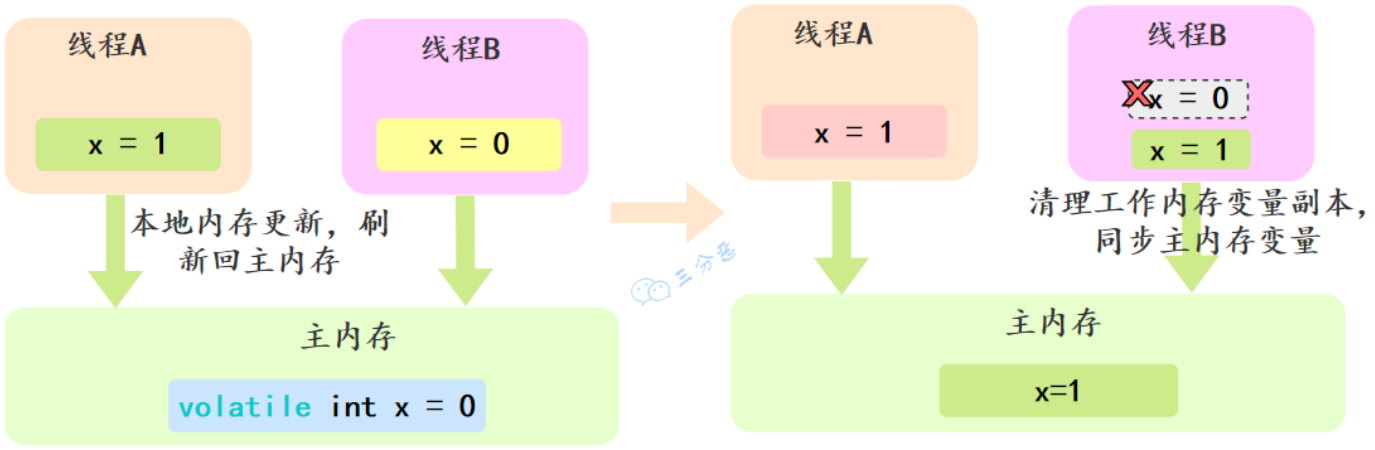
volatile有两个作用，保证可见性和有序性。

> volatile怎么保证可见性的呢?

相比synchronized的加锁方式来解决共享变量的内存可见性问题，volatile就是更轻量的选择，它没有上下文切换的额外开销成本。

volatile可以确保对某个变量的更新对其他线程马上可见，一个变量被声明为volatile 时，线程在写入变量时不会把值缓存在寄存器或者其他地方，而是会把值刷新回主内存 当其它线程读取该共享变量 ，会从主内存重新获取最新值，而不是使用当前线程的本地内存中的值。

例如，我们声明一个 volatile 变量 volatile int x = 0，线程A修改x=1，修改完之后就会把新的值刷新回主内存，线程B读取x的时候，就会清空本地内存变量，然后再从主内存获取最新值。



> volatile怎么保证有序性的呢?

重排序可以分为编译器重排序和处理器重排序，valatile保证有序性，就是通过分别限制这两种类型的重排序。

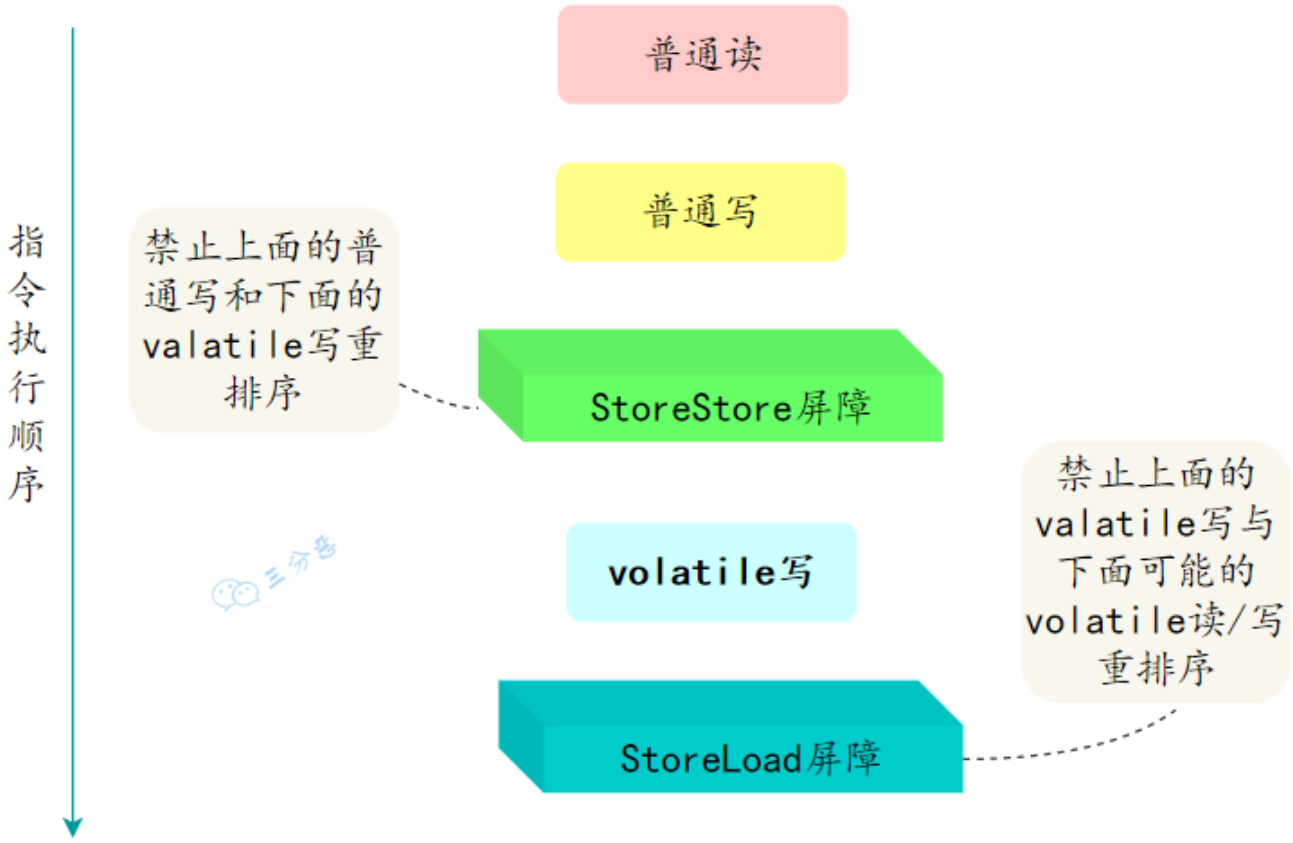
volatile重排序规则表

是否能重排序	第二个操作		
第一个操作	普通读/写	valatile读	valatile写
普通读/写	✓	✓	✗
valatile读	✗	✗	✗
valatile写	✓	✗	✗

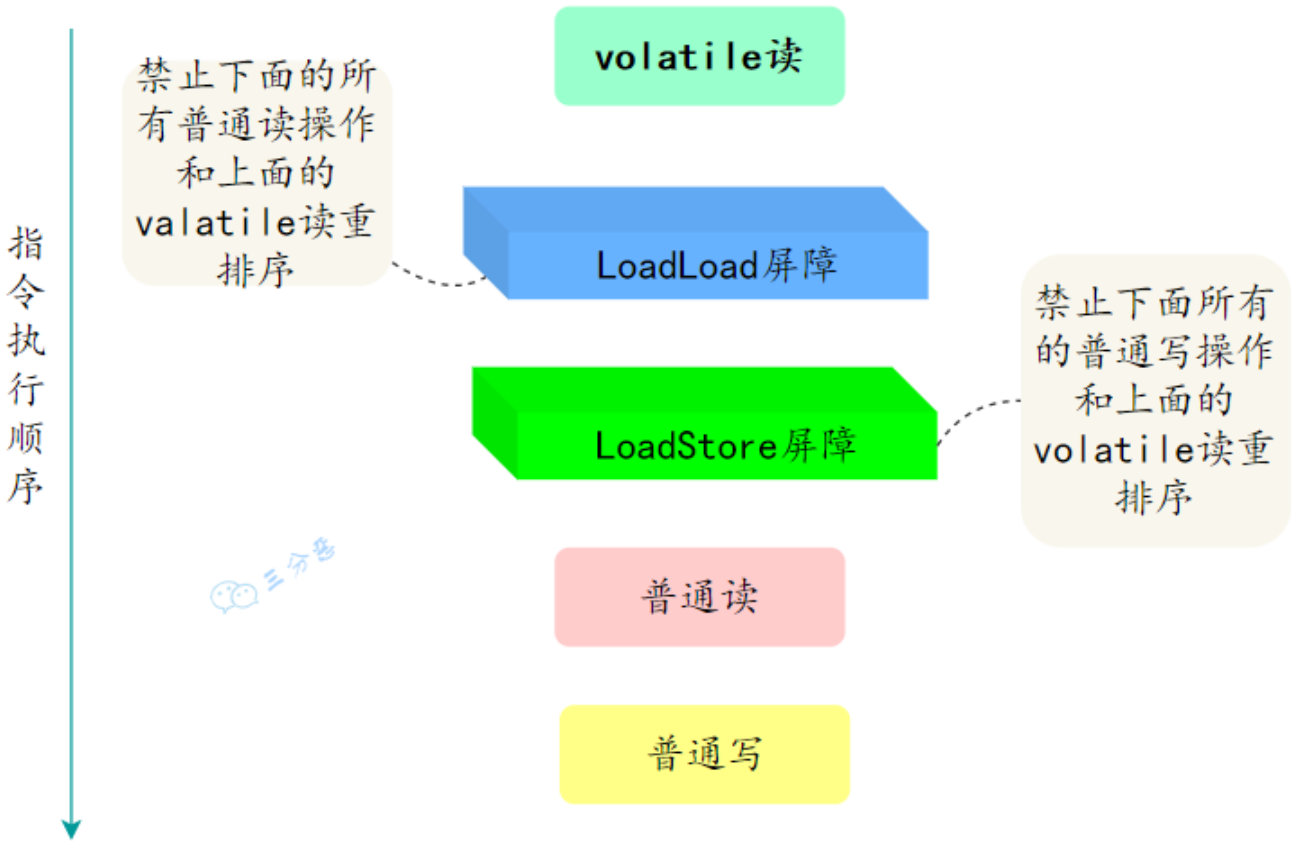
为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

1. 在每个volatile写操作的前面插入一个`StoreStore`屏障
2. 在每个volatile写操作的后面插入一个`StoreLoad`屏障
3. 在每个volatile读操作的后面插入一个`LoadLoad`屏障
4. 在每个volatile读操作的后面插入一个`LoadStore`屏障

volatile写插入内存屏障后生成的指令序列示意图



volatile写插入内存屏障后生成的指令序列示意图



锁

24.synchronized用过吗？怎么使用？

synchronized经常用的，用来保证代码的原子性。

synchronized主要有三种用法：

- **修饰实例方法：**作用于当前对象实例加锁，进入同步代码前要获得 **当前对象实例的锁**

```
synchronized void method() {
 //业务代码
}
```

- **修饰静态方法**：也就是给当前类加锁，会作用于类的所有对象实例 ，进入同步代码前要获得当前 class 的锁。因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管 new 了多少个对象，只有一份）。

  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，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。

```
synchronized void staic method() {
 //业务代码
}
```

- **修饰代码块** ：指定加锁对象，对给定对象/类加锁。 synchronized(this|object) 表示进入同步代码库前要获得给定对象的锁。 synchronized(类.class) 表示进入同步代码前要获得 当前 **class** 的锁

```
synchronized(this) {
 //业务代码
}
```

25.synchronized的实现原理？

synchronized是怎么加锁的呢？

我们使用synchronized的时候，发现不用自己去lock和unlock，是因为JVM帮我们把这个事情做了。

1. synchronized修饰代码块时，JVM采用`monitorenter`、`monitorexit`两个指令来实现同步，`monitorenter`指令指向同步代码块的开始位置，`monitorexit`指令则指向同步代码块的结束位置。

   反编译一段synchronized修饰代码块代码，`javap -c -s -v -l SynchronizedDemo.class`，可以看到相应的字节码指令。

```
public void method();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String synchronized demo
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
```

2. synchronized修饰同步方法时，JVM采用`ACC_SYNCHRONIZED`标记符来实现同步，这个标识指明了该方法是一个同步方法。

同样可以写段代码反编译看一下。

```
public synchronized void method();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String synchronized method
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
    LineNumberTable:
      line 10: 0
      line 11: 8
}
```

synchronized锁住的是什么呢？

monitorenter、monitorexit或者ACC_SYNCHRONIZED都是基于**Monitor**实现的。

实例对象结构里有对象头，对象头里面有一块结构叫Mark Word，Mark Word指针指向了**monitor**。

所谓的Monitor其实是一种**同步工具**，也可以说是一种**同步机制**。在Java虚拟机（HotSpot）中，Monitor是由**ObjectMonitor**实现的，可以叫做内部锁，或者Monitor锁。

ObjectMonitor的工作原理：

- ObjectMonitor有两个队列：*WaitSet*、EntryList，用来保存ObjectWaiter 对象列表。
- _owner，获取 Monitor 对象的线程进入 _owner 区时， _count + 1。如果线程调用了 wait() 方法，此时会释放 Monitor 对象， _owner 恢复为空， _count - 1。同时该等待线程进入 _WaitSet 中，等待被唤醒。

```
ObjectMonitor() {
    _header       = NULL;
    _count        = 0; // 记录线程获取锁的次数
    _waiters      = 0,
    _recursions   = 0;  //锁的重入次数
    _object       = NULL;
    _owner        = NULL;  // 指向持有ObjectMonitor对象的线程
    _WaitSet      = NULL;  // 处于wait状态的线程，会被加入到_WaitSet
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
```

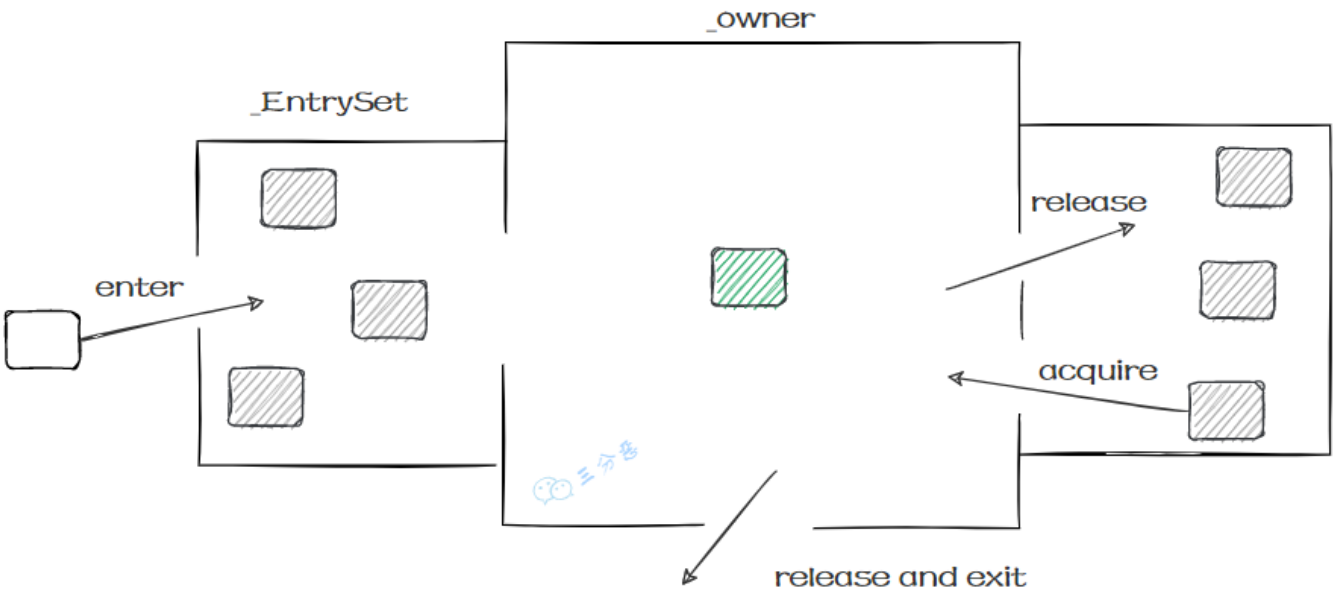
```
    _cxq        = NULL ;
    FreeNext    = NULL ;
    _EntryList  = NULL ; // 处于等待锁block状态的线程，会被加入到该列表
    _SpinFreq   = 0 ;
    _SpinClock  = 0 ;
    OwnerIsThread = 0 ;
  }
```

可以类比一个去医院就诊的例子[18]:

- 首先，患者在门诊大厅前台或自助挂号机进行挂号;
- 随后，挂号结束后患者找到对应的诊室就诊:
  - 诊室每次只能有一个患者就诊;
  - 如果此时诊室空闲，直接进入就诊;
  - 如果此时诊室内有其它患者就诊，那么当前患者进入候诊室，等待叫号;
- 就诊结束后，走出就诊室，候诊室的下一位候诊患者进入就诊室。

这个过程就和Monitor机制比较相似:

- 门诊大厅: 所有待进入的线程都必须先在入口**Entry Set**挂号才有资格;
- 就诊室: 就诊室**_Owner**里里只能有一个线程就诊，就诊完线程就自行离开
- 候诊室: 就诊室繁忙时，进入等待区（**Wait Set**），就诊室空闲的时候就从等待区（**Wait Set**）叫新的线程



所以我们就知道了，同步是锁住的什么东西：

- monitorenter，在判断拥有同步标识 ACC_SYNCHRONIZED 抢先进入此方法的线程会优先拥有 Monitor 的 owner ，此时计数器 +1。
- monitorexit，当执行完退出后，计数器 -1，归 0 后被其他进入的线程获得。

26.除了原子性，synchronized可见性，有序性，可重入性怎么实现?

synchronized怎么保证可见性?

- 线程加锁前，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值。
- 线程加锁后，其它线程无法获取主内存中的共享变量。
- 线程解锁前，必须把共享变量的最新值刷新到主内存中。

synchronized怎么保证有序性?

synchronized同步的代码块，具有排他性，一次只能被一个线程拥有，所以synchronized保证同一时刻，代码是单线程执行的。

因为as-if-serial语义的存在，单线程的程序能保证最终结果是有序的，但是不保证不会指令重排。

所以synchronized保证的有序是执行结果的有序性，而不是防止指令重排的有序性。

synchronized怎么实现可重入的呢?

synchronized 是可重入锁，也就是说，允许一个线程二次请求自己持有对象锁的临界资源，这种情况称为可重入锁。

synchronized 锁对象的时候有个计数器，他会记录下线程获取锁的次数，在执行完对应的代码块之后，计数器就会-1，直到计数器清零，就释放锁了。

之所以，是可重入的。是因为 synchronized 锁对象有个计数器，会随着线程获取锁后 +1 计数，当线程执行完毕后 -1，直到清零释放锁。

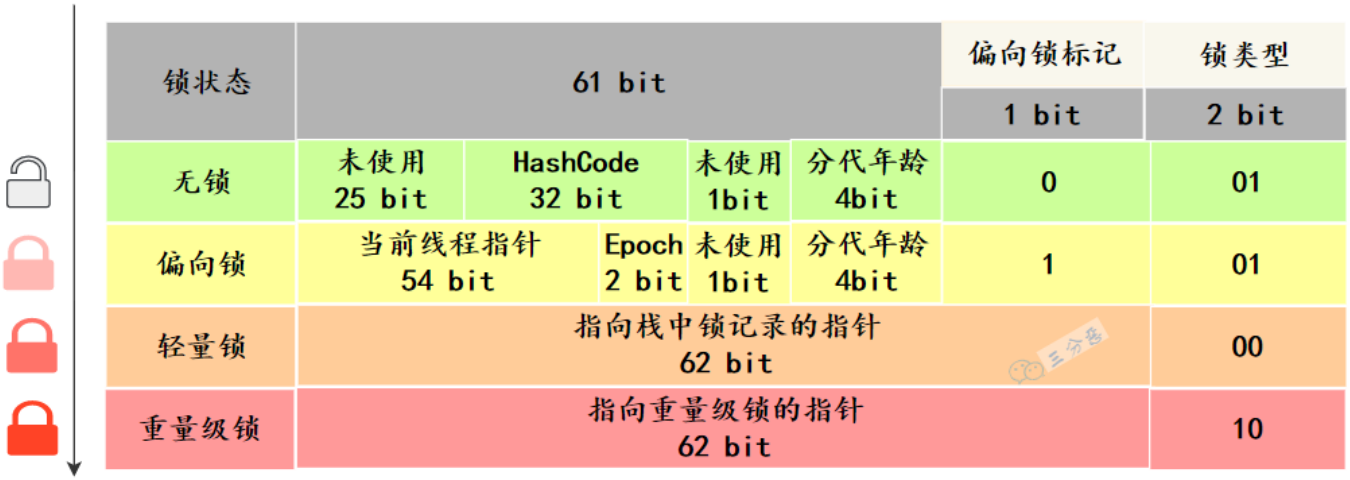
27.锁升级？synchronized优化了解吗？

了解锁升级，得先知道，不同锁的状态是什么样的。这个状态指的是什么呢？

Java对象头里，有一块结构，叫`Mark Word`标记字段，这块结构会随着锁的状态变化而变化。

64 位虚拟机 Mark Word 是 64bit，我们来看看它的状态变化：

Mark Word变化示意图



锁状态	61 bit				偏向锁标记	锁类型
					1 bit	2 bit
无锁	未使用 25 bit	HashCode 32 bit	未使用 1bit	分代年龄 4bit	0	01
偏向锁	当前线程指针 54 bit	Epoch 2 bit	未使用 1bit	分代年龄 4bit	1	01
轻量锁	指向栈中锁记录的指针 62 bit					00
重量级锁	指向重量级锁的指针 62 bit					10

Mark Word存储对象自身的运行数据，如**哈希码、GC分代年龄、锁状态标志、偏向时间戳（Epoch）**等。

synchronized做了哪些优化？

在JDK1.6之前，synchronized的实现直接调用ObjectMonitor的enter和exit，这种锁被称之为重量级锁。从JDK6开始，HotSpot虚拟机开发团队对Java中的锁进行优化，如增加了适应性自旋、锁消除、锁粗化、轻量级锁和偏向锁等优化策略，提升了synchronized的性能。

- 偏向锁：在无竞争的情况下，只是在Mark Word里存储当前线程指针，CAS操作都不做。
- 轻量级锁：在没有多线程竞争时，相对重量级锁，减少操作系统互斥量带来的性能消耗。但是，如

果存在锁竞争，除了互斥量本身开销，还额外有CAS操作的开销。

- 自旋锁：减少不必要的CPU上下文切换。在轻量级锁升级为重量级锁时，就使用了自旋加锁的方式
- 锁粗化：将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。
- 锁消除：虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。

> 锁升级的过程是什么样的？

锁升级方向：无锁-->偏向锁---> 轻量级锁---->重量级锁，这个方向基本上是不可逆的。

我们看一下升级的过程：

偏向锁：

偏向锁的获取：

1. 判断是否为可偏向状态--MarkWord中锁标志是否为‘01’，是否偏向锁是否为‘1’
2. 如果是可偏向状态，则查看线程ID是否为当前线程，如果是，则进入步骤'5'，否则进入步骤‘3’
3. 通过CAS操作竞争锁，如果竞争成功，则将MarkWord中线程ID设置为当前线程ID，然后执行‘5’；竞争失败，则执行‘4’
4. CAS获取偏向锁失败表示有竞争。当达到safepoint时获得偏向锁的线程被挂起，**偏向锁升级为轻量级锁**，然后被阻塞在安全点的线程继续往下执行同步代码块
5. 执行同步代码

偏向锁的撤销：

1. 偏向锁不会主动释放(撤销)，只有遇到其他线程竞争时才会执行撤销，由于撤销需要知道当前持有该偏向锁的线程栈状态，因此要等到safepoint时执行，此时持有该偏向锁的线程（T）有‘2’，‘3’两种情况；
2. 撤销----T线程已经退出同步代码块，或者已经不再存活，则直接撤销偏向锁，变成无锁状态----该状态达到阈值20则执行批量重偏向
3. 升级----T线程还在同步代码块中，则将T线程的偏向锁**升级为轻量级锁**，当前线程执行轻量级锁

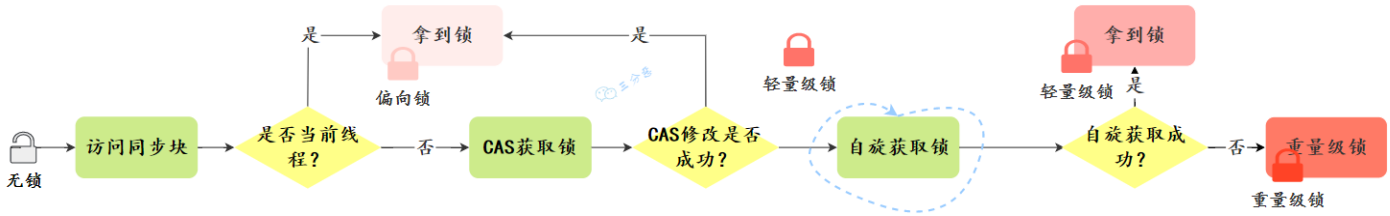
状态下的锁获取步骤----该状态达到阈值40则执行批量撤销

轻量级锁：

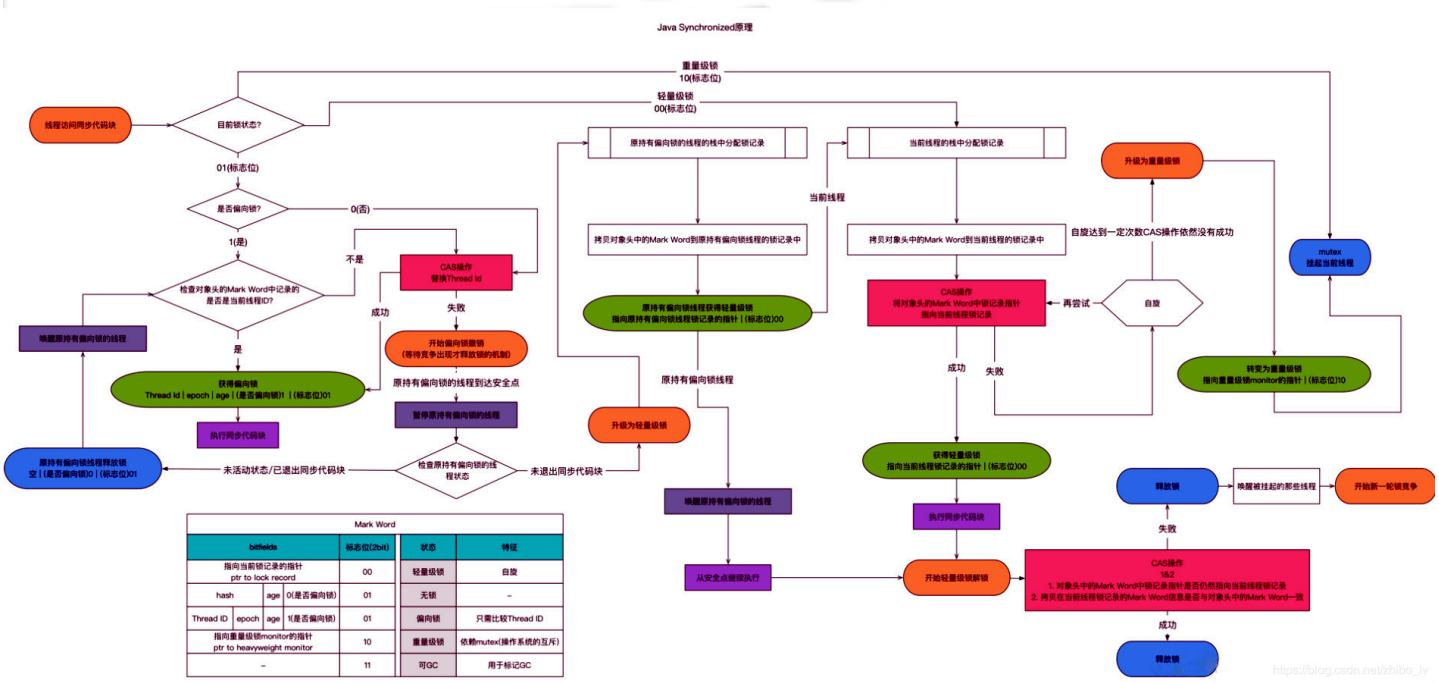
轻量级锁的获取：

1. 进行加锁操作时，jvm会判断是否已经时重量级锁，如果不是，则会在当前线程栈帧中划出一块空间，作为该锁的锁记录，并且将锁对象MarkWord复制到该锁记录中
2. 复制成功之后，jvm使用CAS操作将对象头MarkWord更新为指向锁记录的指针，并将锁记录里的owner指针指向对象头的MarkWord。如果成功，则执行‘3’，否则执行‘4’
3. 更新成功，则当前线程持有该对象锁，并且对象MarkWord锁标志设置为‘00’，即表示此对象处于轻量级锁状态
4. 更新失败，jvm先检查对象MarkWord是否指向当前线程栈帧中的锁记录，如果是则执行‘5’，否则执行‘4’
5. 表示锁重入；然后当前线程栈帧中增加一个锁记录第一部分（Displaced Mark Word）为null，并指向Mark Word的锁对象，起到一个重入计数器的作用。
6. 表示该锁对象已经被其他线程抢占，则进行**自旋等待**（默认10次），等待次数达到阈值仍未获取到锁，则**升级为重量级锁**

大体上省简的升级过程：



完整的升级过程：



28.说说synchronized和ReentrantLock的区别?

可以从锁的实现、功能特点、性能等几个维度去回答这个问题:

- **锁的实现:** synchronized是Java语言的关键字，基于JVM实现。而ReentrantLock是基于JDK的API层面实现的（一般是lock()和unlock()方法配合try/finally 语句块来完成。）
- **性能:** 在JDK1.6锁优化以前，synchronized的性能比ReenTrantLock差很多。但是JDK6开始，增加了适应性自旋、锁消除等，两者性能就差不多了。
- **功能特点:** ReentrantLock 比 synchronized 增加了一些高级功能，如等待可中断、可实现公平锁、可实现选择性通知。
    - ReentrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制
    - 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。
    - synchronized与wait()和notify()/notifyAll()方法结合实现等待/通知机制，ReentrantLock类借助Condition接口与newCondition()方法实现。
    - ReentrantLock需要手工声明来加锁和释放锁，一般跟finally配合释放锁。而synchronized不用手动释放锁。

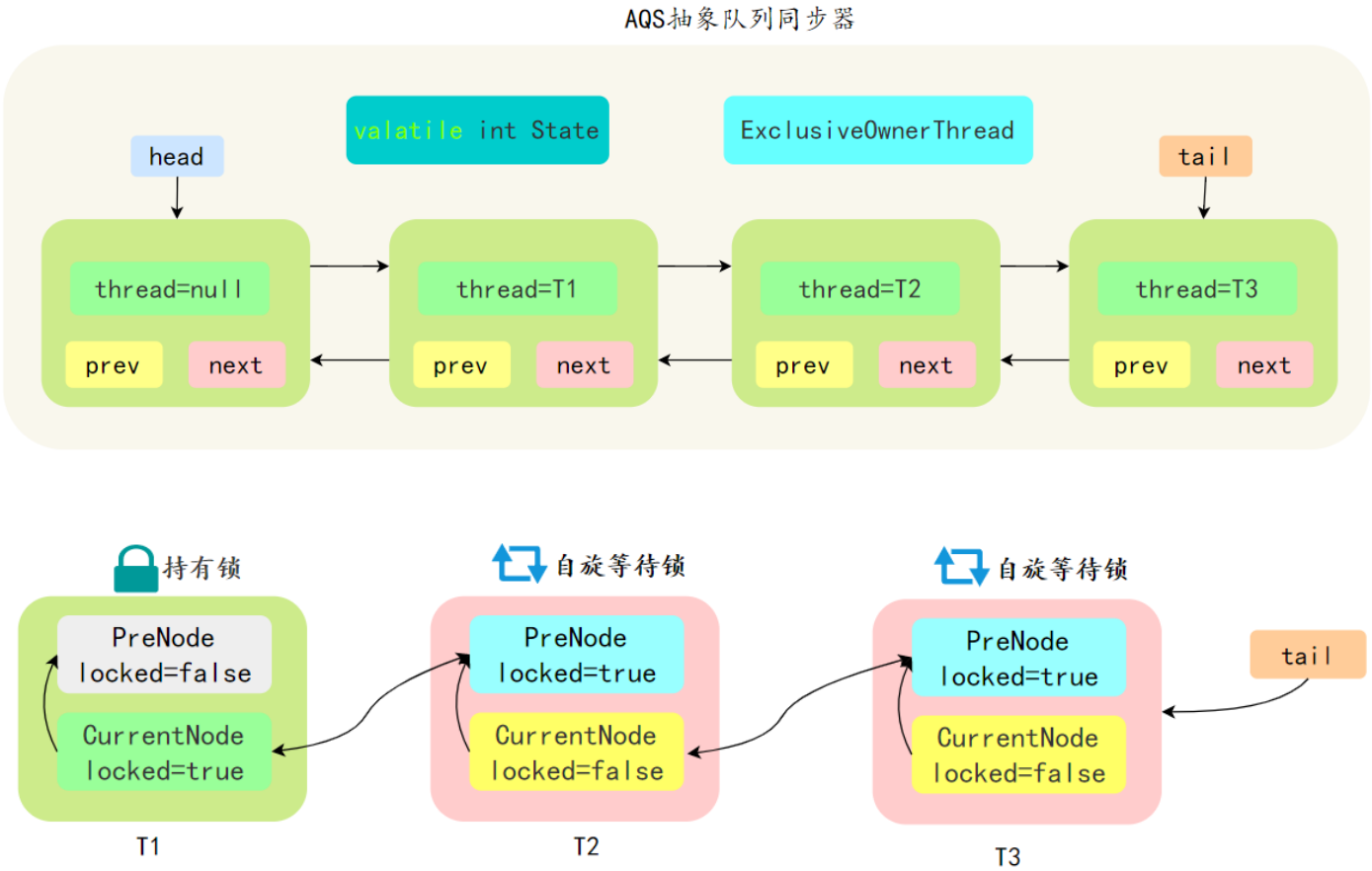
下面的表格列出出了两种锁之间的区别:

区别	sychronized	ReentrantLock
锁实现机制	对象头监视器模式	依赖AQS
灵活性	不灵活	支持响应中断、超时、尝试获取锁
释放锁形式	自动释放锁	显式调用unlocak()
支持锁类型	非公平锁	公平锁&非公平锁
条件队列	单条件队列	多个条件队列
可重入支持	支持	支持

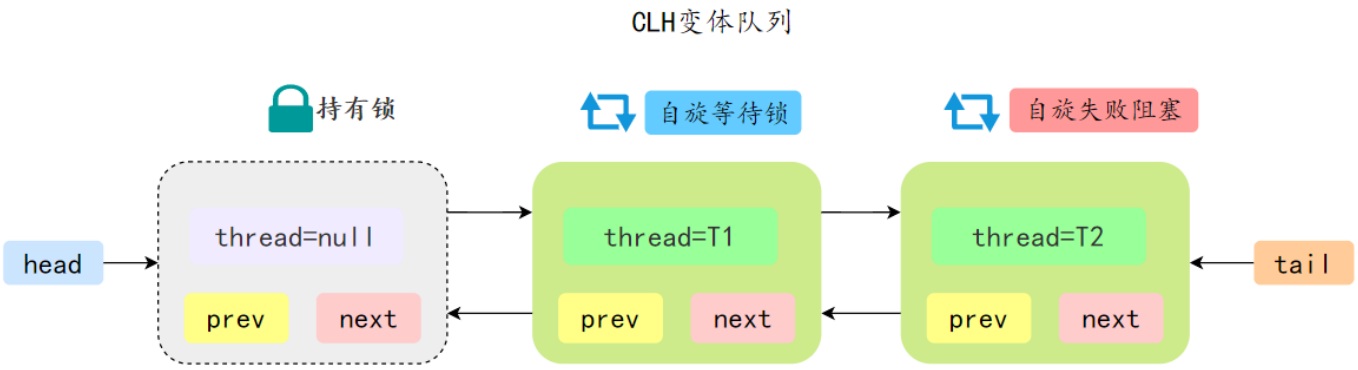
29.AQS了解多少?

AbstractQueuedSynchronizer 抽象同步队列，简称 AQS ，它是Java并发包的根基，并发包中的锁就是基于AQS实现的。

- AQS是基于一个FIFO的双向队列，其内部定义了一个节点类Node，Node 节点内部的 SHARED 用来标记该线程是获取共享资源时被阻挂起后放入AQS 队列的， EXCLUSIVE 用来标记线程是取独占资源时被挂起后放入AQS 队列
- AQS 使用一个 volatile 修饰的 int 类型的成员变量 state 来表示同步状态，修改同步状态成功即为获得锁，volatile 保证了变量在多线程之间的可见性，修改 State 值时通过 CAS 机制来保证修改的原子性
- 获取state的方式分为两种，独占方式和共享方式，一个线程使用独占方式获取了资源，其它线程就会在获取失败后被阻塞。一个线程使用共享方式获取了资源，另外一个线程还可以通过CAS的方式进行获取。
- 如果共享资源被占用，需要一定的阻塞等待唤醒机制来保证锁的分配，AQS 中会将竞争共享资源失败的线程添加到一个变体的 CLH 队列中。



AQS 中的队列是 CLH 变体的虚拟双向队列，通过将每条请求共享资源的线程封装成一个节点来实现锁的分配：



AQS 中的 CLH 变体等待队列拥有以下特性：

- AQS 中队列是个双向链表，也是 FIFO 先进先出的特性
- 通过 Head、Tail 头尾两个节点来组成队列结构，通过 volatile 修饰保证可见性
- Head 指向节点为已获得锁的节点，是一个虚拟节点，节点本身不持有具体线程
- 获取不到同步状态，会将节点进行自旋获取锁，自旋一定次数失败后会将线程阻塞，相对于 CLH 队列性能较好

ps:AQS源码里面有很多细节可问，建议有时间好好看看AQS源码。

30.ReentrantLock实现原理?

ReentrantLock 是可重入的独占锁，只能有一个线程可以获取该锁，其它获取该锁的线程会被阻塞而被放入该锁的阻塞队列里面。

看看ReentrantLock的加锁操作:

```
// 创建非公平锁
ReentrantLock lock = new ReentrantLock();
// 获取锁操作
lock.lock();
try {
    // 执行代码逻辑
} catch (Exception ex) {
    // ...
} finally {
    // 解锁操作
    lock.unlock();
}
```

`new ReentrantLock()`构造函数默认创建的是非公平锁 NonfairSync。

公平锁 FairSync

1. 公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁
2. 公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU 唤醒阻塞线程的开销比非公平锁大

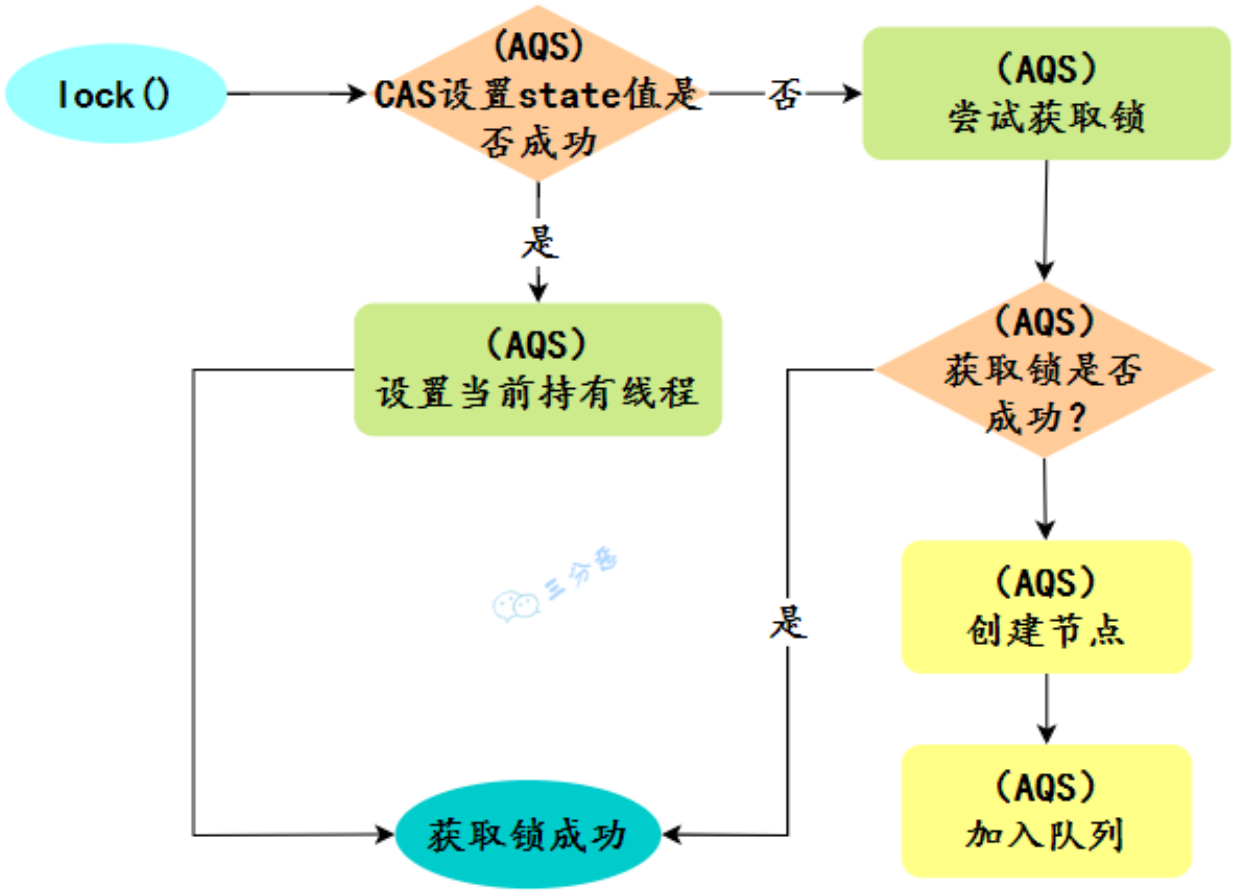
非公平锁 NonfairSync

- 非公平锁是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁
- 非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU 不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁

默认创建的对象lock()的时候:

- 如果锁当前没有被其它线程占用，并且当前线程之前没有获取过该锁，则当前线程会获取到该锁，然后设置当前锁的拥有者为当前线程，并设置 AQS 的状态值为1 ，然后直接返回。如果当前线程之前已经获取过该锁，则这次只是简单地把 AQS 的状态值加1后返回。
- 如果该锁已经被其他线程持有，非公平锁会尝试去获取锁，获取失败的话，则调用该方法线程会被

放入 AQS 队列阻塞挂起。



31.ReentrantLock怎么实现公平锁的?

`new ReentrantLock()` 构造函数默认创建的是非公平锁 NonfairSync

```
public ReentrantLock() {
    sync = new NonfairSync();
}
```

同时也可以在创建锁构造函数中传入具体参数创建公平锁 FairSync

```
ReentrantLock lock = new ReentrantLock(true);
--- ReentrantLock
// true 代表公平锁，false 代表非公平锁
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

FairSync、NonfairSync 代表公平锁和非公平锁，两者都是 ReentrantLock 静态内部类，只不过实现不同锁语义。

非公平锁和公平锁的两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

```java
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}

final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // State 等于0表示此时无锁
    if (c == 0) {
        // 再次使用CAS尝试获取锁，表现为非公平锁特性
        if (compareAndSetState(0, acquires)) {
            // 设置线程为独占锁线程
            setExclusiveOwnerThread(current);
            return true;
        }
    // 如果当前线程等于已获取锁线程，表现为可重入锁特性
    } else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        // 设置 State
        setState(nextc);
        return true;
    }
    // 如果state不等于0并且独占线程不是当前线程，返回 false
    return false;
}
```

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

32.CAS呢？CAS了解多少？

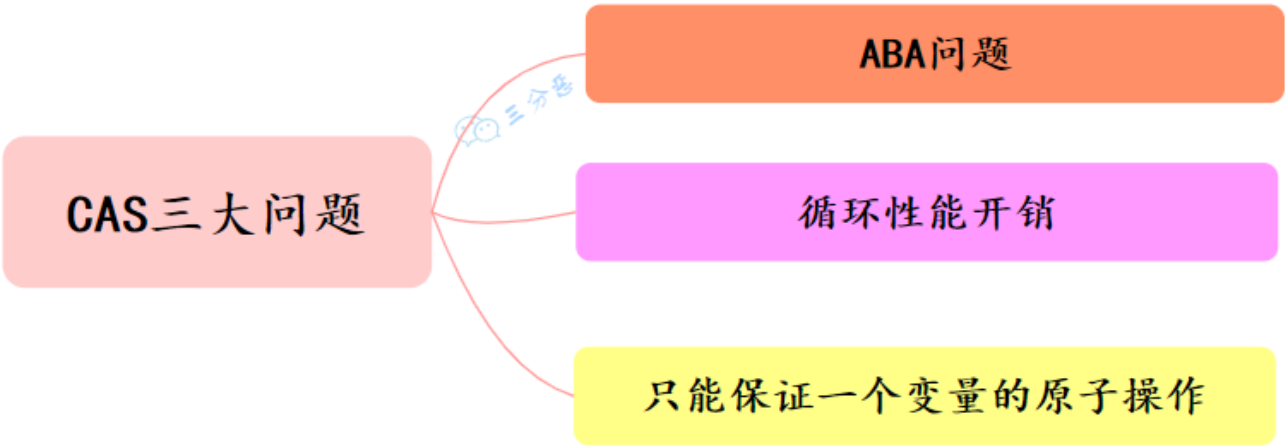
CAS叫做CompareAndSwap，比较并交换，主要是通过处理器的指令来保证操作的原子性的。

CAS 指令包含 3 个参数：共享变量的内存地址 A、预期的值 B 和共享变量的新值 C。

只有当内存中地址 A 处的值等于 B 时，才能将内存中地址 A 处的值更新为新值 C。作为一条 CPU 指令，CAS 指令本身是能够保证原子性的 。

33.CAS 有什么问题？如何解决？

CAS的经典三大问题：



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。

> 怎么解决ABA问题？

- 加版本号

每次修改变量，都在这个变量的版本号上加1，这样，刚刚A->B->A，虽然A的值没变，但是它的版本号已经变了，再判断版本号就会发现此时的A已经被改过了。参考乐观锁的版本号，这种做法可以给数据带上了一种实效性的检验。

Java提供了AtomicStampReference类，它的compareAndSet方法首先检查当前的对象引用值是否等于预期引用，并且当前印戳（Stamp）标志是否等于预期标志，如果全部相等，则以原子方式将引用值和印戳标志的值更新为给定的更新值。

循环性能开销

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。

> 怎么解决循环性能开销问题？

在Java中，很多使用自旋CAS的地方，会有一个自旋次数的限制，超过一定次数，就停止自旋。

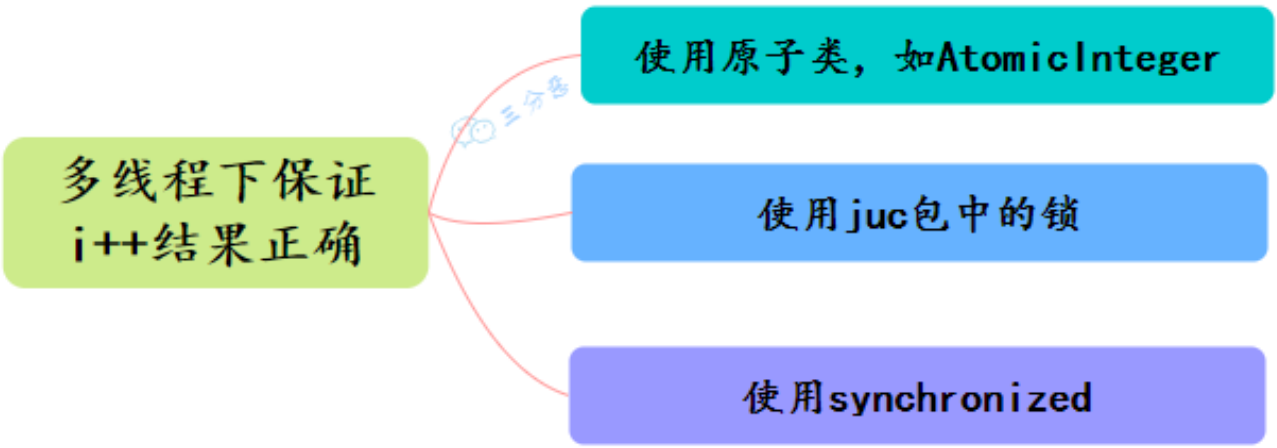
只能保证一个变量的原子操作

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。

> 怎么解决只能保证一个变量的原子操作问题？

- 可以考虑改用锁来保证操作的原子性
- 可以考虑合并多个变量，将多个变量封装成一个对象，通过AtomicReference来保证原子性。

34.Java有哪些保证原子性的方法？如何保证多线程下i++ 结果正确？



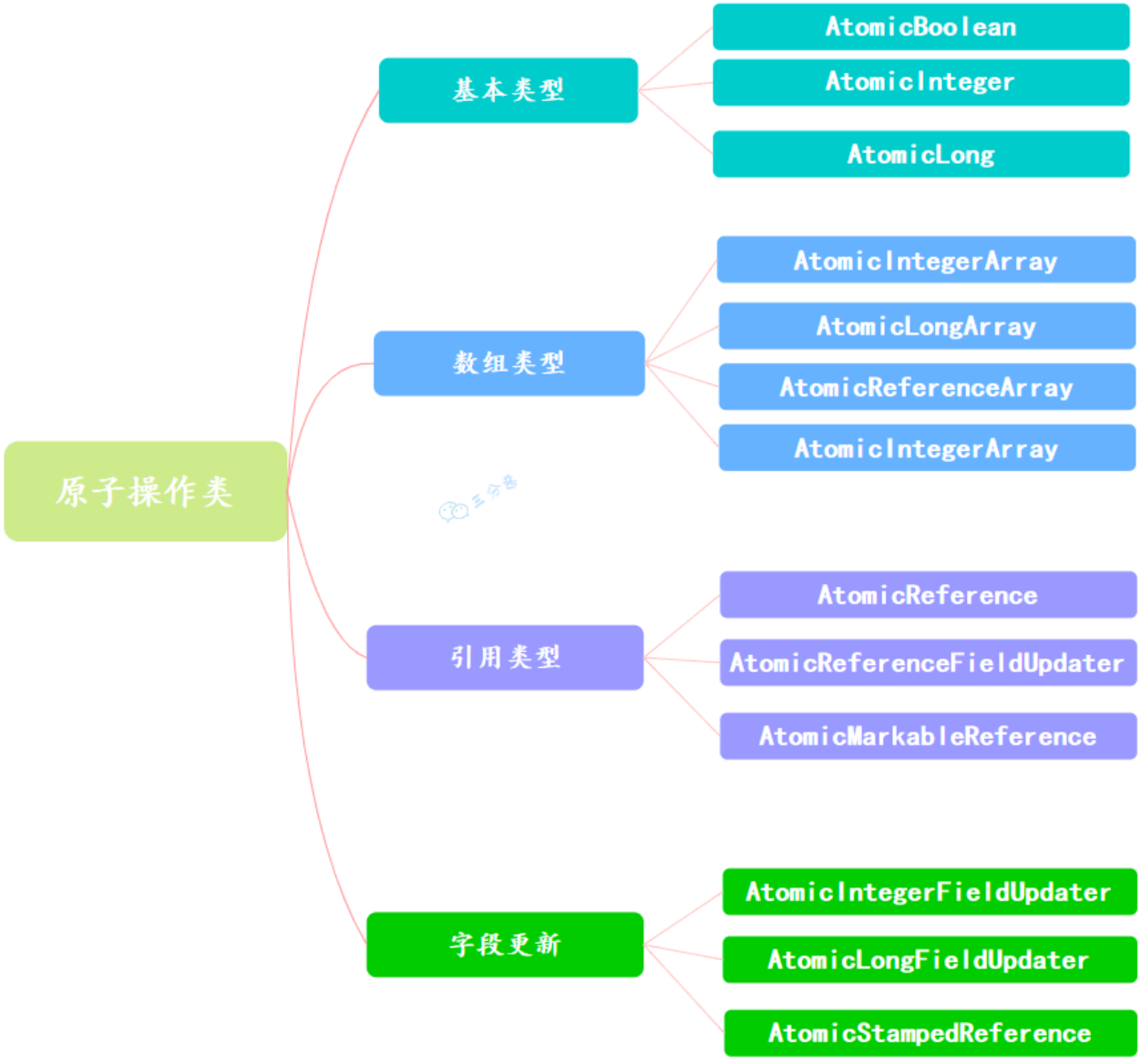
- 使用循环原子类，例如AtomicInteger，实现i++原子操作
- 使用juc包下的锁，如ReentrantLock ，对i++操作加锁lock.lock()来实现原子性
- 使用synchronized，对i++操作加锁

35.原子操作类了解多少？

当程序更新一个变量时，如果多线程同时更新这个变量，可能得到期望之外的值，比如变量i=1，A线程更新i+1，B线程也更新i+1，经过两个线程操作之后可能i不等于3，而是等于2。因为A和B线程在更新变量i的时候拿到的i都是1，这就是线程不安全的更新操作，一般我们会使用synchronized来解决这个问题，synchronized会保证多线程不会同时更新变量i。

其实除此之外，还有更轻量级的选择，Java从JDK 1.5开始提供了java.util.concurrent.atomic包，这个包中的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式。

因为变量的类型有很多种，所以在Atomic包里一共提供了13个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性（字段）。



Atomic包里的类基本都是使用Unsafe实现的包装类。

使用原子的方式更新基本类型，Atomic包提供了以下3个类：

- AtomicBoolean：原子更新布尔类型。

- AtomicInteger：原子更新整型。
- AtomicLong：原子更新长整型。

通过原子的方式更新数组里的某个元素，Atomic包提供了以下4个类：

- AtomicIntegerArray：原子更新整型数组里的元素。
- AtomicLongArray：原子更新长整型数组里的元素。
- AtomicReferenceArray：原子更新引用类型数组里的元素。
- AtomicIntegerArray类主要是提供原子的方式更新数组里的整型

原子更新基本类型的AtomicInteger，只能更新一个变量，如果要原子更新多个变量，就需要使用这个原子更新引用类型提供的类。Atomic包提供了以下3个类：

- AtomicReference：原子更新引用类型。
- AtomicReferenceFieldUpdater：原子更新引用类型里的字段。
- AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和引用类型。构造方法是AtomicMarkableReference（V initialRef，boolean initialMark）。

如果需原子地更新某个类里的某个字段时，就需要使用原子更新字段类，Atomic包提供了以下3个类进行原子字段更新：

- AtomicIntegerFieldUpdater：原子更新整型的字段的更新器。
- AtomicLongFieldUpdater：原子更新长整型字段的更新器。
- AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于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。

36.AtomicInteger 的原理?

一句话概括：**使用CAS实现。**

以AtomicInteger的添加方法为例：

```
    public final int getAndIncrement() {
        return unsafe.getAndAddInt(this, valueOffset, 1);
    }
```

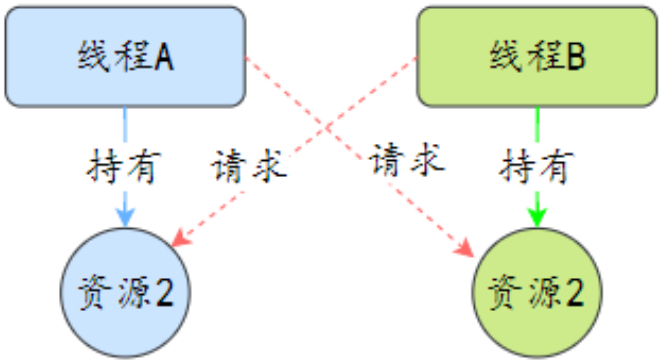
通过`Unsafe`类的实例来进行添加操作，来看看具体的CAS操作：

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

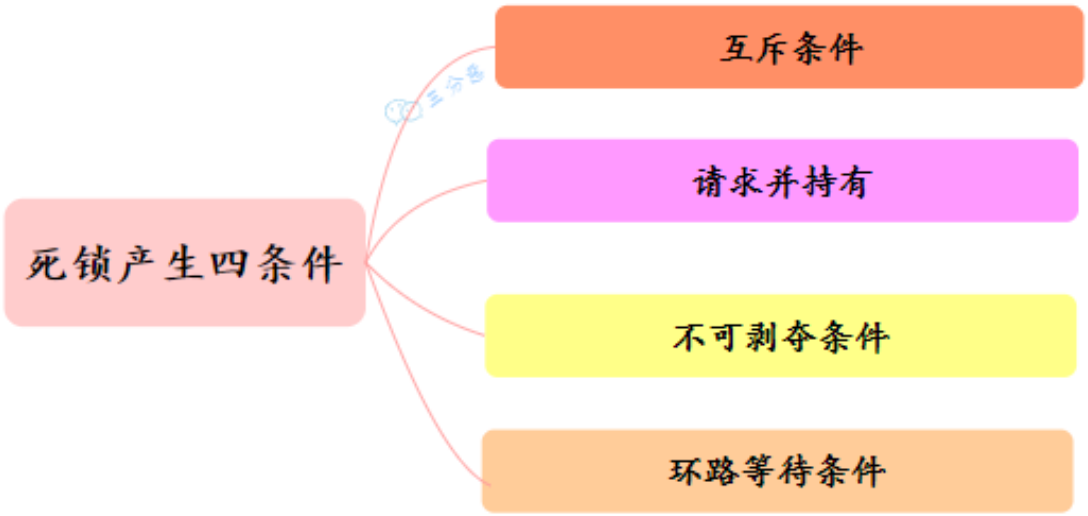
compareAndSwapInt 是一个native方法，基于CAS来操作int类型变量。其它的原子操作类基本都是大同小异。

37.线程死锁了解吗？该如何避免？

死锁是指两个或两个以上的线程在执行过程中，因争夺资源而造成的互相等待的现象，在无外力作用的情况下，这些线程会一直相互等待而无法继续运行下去。



那么为什么会产生死锁呢？ 死锁的产生必须具备以下四个条件：



- 互斥条件：指线程对已经获取到的资源进行它性使用，即该资源同时只由一个线程占用。如果此时还有其它线程请求获取获取该资源，则请求者只能等待，直至占有资源的线程释放该资源。
- 请求并持有条件：指一个 线程已经持有了至少一个资源，但又提出了新的资源请求，而新资源已被其它线程占有，所以当前线程会被阻塞，但阻塞 的同时并不释放自己已经获取的资源。
- 不可剥夺条件：指线程获取到的资源在自己使用完之前不能被其它线程抢占，只有在自己使用完毕后才由自己释放该资源。
- 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，即线程集合 {T0，T1，T2,…… ，Tn} 中 T0 正在等待一 T1 占用的资源，T11正在等待 T2用的资源，…… Tn 在等待已被 T0占用的资源。

该如何避免死锁呢？答案是**至少破坏死锁发生的一个条件**。

- 其中，互斥这个条件我们没有办法破坏，因为用锁为的就是互斥。不过其他三个条件都是有办法破坏掉的，到底如何做呢？
- 对于“请求并持有”这个条件，可以一次性请求所有的资源。
- 对于“不可剥夺”这个条件，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，这样不可抢占这个条件就破坏掉了。
- 对于“环路等待”这个条件，可以靠按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源序号小的，再申请资源序号大的，这样线性化后就不存在环路了。

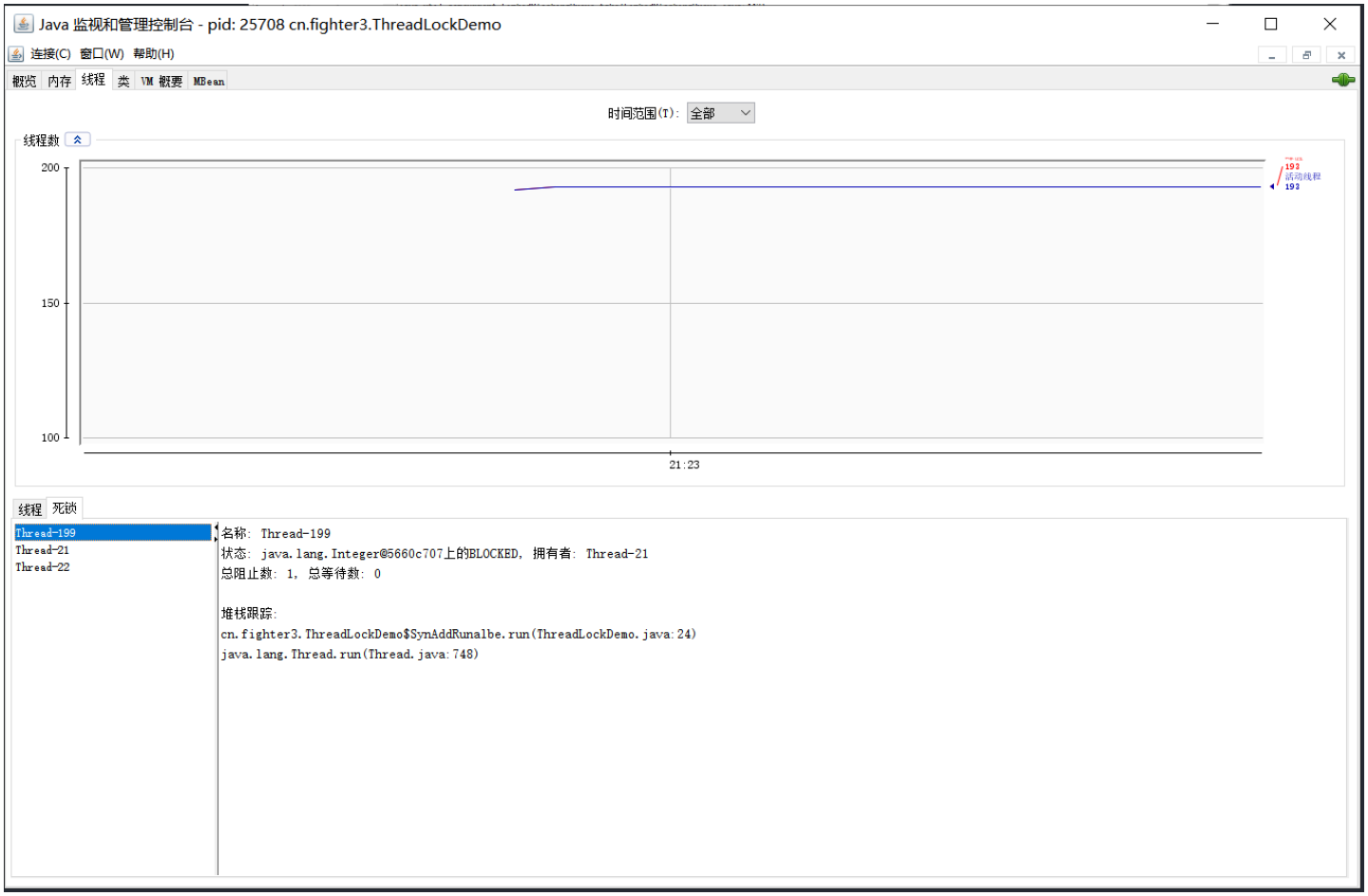
38.那死锁问题怎么排查呢？

可以使用jdk自带的命令行工具排查：

```
1. 使用jps查找运行的Java进程：jps -l
2. 使用jstack查看线程堆栈信息：jstack -l 进程id
```

基本就可以看到死锁的信息。

还可以利用图形化工具，比如JConsole。出现线程死锁以后，点击JConsole线程面板的`检测到死锁`按钮，将会看到线程的死锁信息。


Java 监视和管理控制台 - pid: 25708 cn.fighter3.ThreadLockDemo
连接(C) 窗口(W) 帮助(H)
概览 内存 线程 类 VM 概要 MBean
时间范围(T): 全部
线程数
200
150
100
21:23
线程 死锁
Thread-199
Thread-21
Thread-22
名称: Thread-199
状态: java.lang.Integer@5660c707上的BLOCKED, 拥有者: Thread-21
总阻止数: 1, 总等待数: 0
堆栈跟踪:
cn.fighter3.ThreadLockDemo$SynAddRunalbe.run(ThreadLockDemo.java:24)
java.lang.Thread.run(Thread.java:748)


关注沉默王二
学Java不迷路


16:58
公众号
沉默王二
发消息
技术文通俗易懂，吹水文风趣幽默。
689篇原创内容 7780个朋友关注
视频号：沉默王二
精彩内容不容错过，加个星标吧
每天一篇硬核Java后端文章，带你
循序渐进学Java！
消息
视频号
服务
设为星标
推荐给朋友

并发工具类

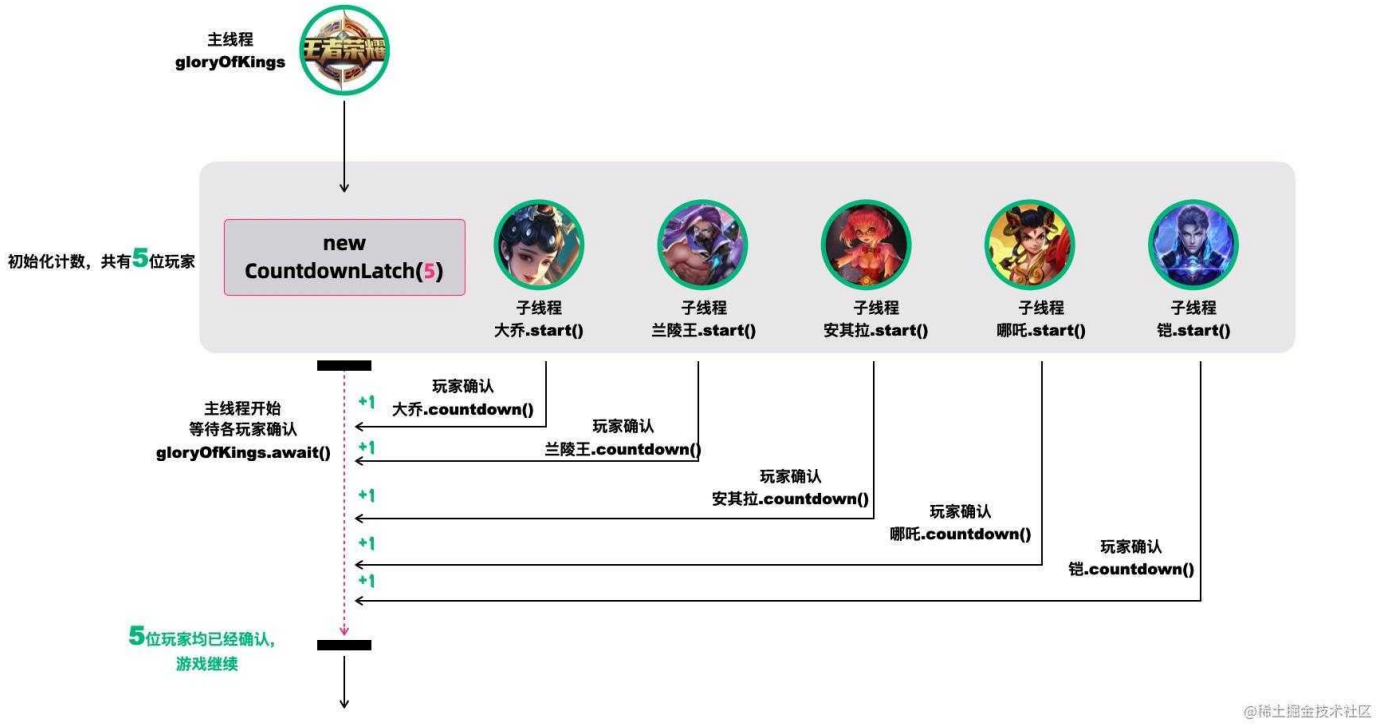
39.CountDownLatch（倒计数器）了解吗？

CountDownLatch，倒计数器，有两个常见的应用场景[18]：

场景1：协调子线程结束动作：等待所有子线程运行结束

CountDownLatch允许一个或多个线程等待其他线程完成操作。

例如，我们很多人喜欢玩的王者荣耀，开黑的时候，得等所有人都上线之后，才能开打。



CountDownLatch模仿这个场景(参考[18])：

创建大乔、兰陵王、安其拉、哪吒和铠等五个玩家，主线程必须在他们都完成确认后，才可以继续运行。

在这段代码中，`new CountDownLatch(5)`用户创建初始的latch数量，各玩家通过`countDownLatch.countDown()`完成状态确认，主线程通过`countDownLatch.await()`等待。

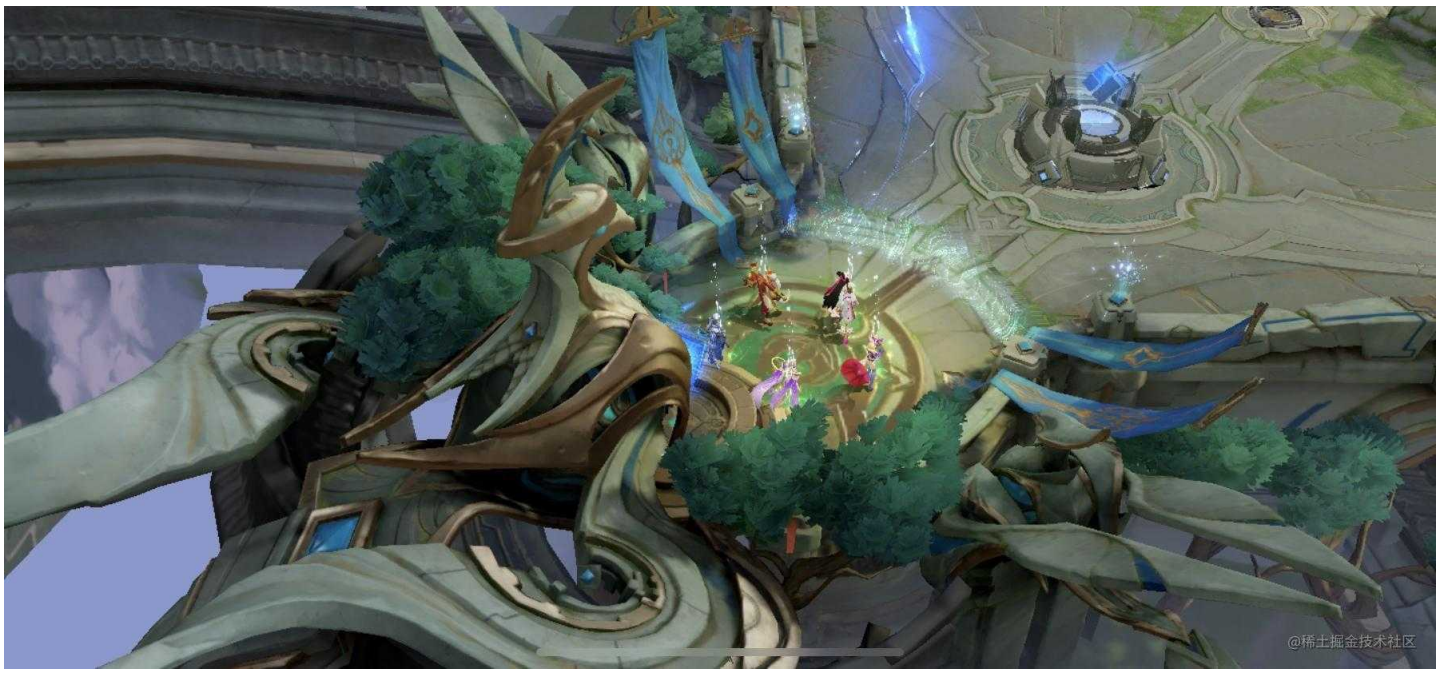
```
public static void main(String[] args) throws InterruptedException {
    CountDownLatch countDownLatch = new CountDownLatch(5);

    Thread 大乔 = new Thread(countDownLatch::countDown);
    Thread 兰陵王 = new Thread(countDownLatch::countDown);
    Thread 安其拉 = new Thread(countDownLatch::countDown);
    Thread 哪吒 = new Thread(countDownLatch::countDown);
    Thread 铠 = new Thread(() -> {
        try {
```

```
            // 稍等，上个卫生间，马上到...
            Thread.sleep(1500);
            countDownLatch.countDown();
        } catch (InterruptedException ignored) {}
    });

    大乔.start();
    兰陵王.start();
    安其拉.start();
    哪吒.start();
    铠.start();
    countDownLatch.await();
    System.out.println("所有玩家已经就位！");
}
```

场景2. 协调子线程开始动作：统一各线程动作开始的时机

王者游戏中也有类似的场景，游戏开始时，各玩家的初始状态必须一致。不能有的玩家都出完装了，有的才降生。

所以大家得一块出生，在



在这个场景中，仍然用五个线程代表大乔、兰陵王、安其拉、哪吒和铠等五个玩家。需要注意的是，各玩家虽然都调用了`start()`线程，但是它们在运行时都在等待`countDownLatch`的信号，在信号未收到前，它们不会往下执行。

```
public static void main(String[] args) throws InterruptedException {
    CountDownLatch countDownLatch = new CountDownLatch(1);
```

```
        Thread 大乔 = new Thread(() -> waitToFight(countDownLatch));
        Thread 兰陵王 = new Thread(() -> waitToFight(countDownLatch));
        Thread 安其拉 = new Thread(() -> waitToFight(countDownLatch));
        Thread 哪吒 = new Thread(() -> waitToFight(countDownLatch));
        Thread 铠 = new Thread(() -> waitToFight(countDownLatch));

        大乔.start();
        兰陵王.start();
        安其拉.start();
        哪吒.start();
        铠.start();
        Thread.sleep(1000);
        countDownLatch.countDown();
        System.out.println("敌方还有5秒达到战场，全军出击！");
    }

    private static void waitToFight(CountDownLatch countDownLatch) {
        try {
            countDownLatch.await(); // 在此等待信号再继续
            System.out.println("收到，发起进攻！");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

CountDownLatch的核心方法也不多：

- `await()`：等待latch降为0；
- `boolean await(long timeout, TimeUnit unit)`：等待latch降为0，但是可以设置超时时间。比如有玩家超时未确认，那就重新匹配，总不能为了某个玩家等到天荒地老。
- `countDown()`：latch数量减1；
- `getCount()`：获取当前的latch数量。

40.CyclicBarrier（同步屏障）了解吗？

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运行。

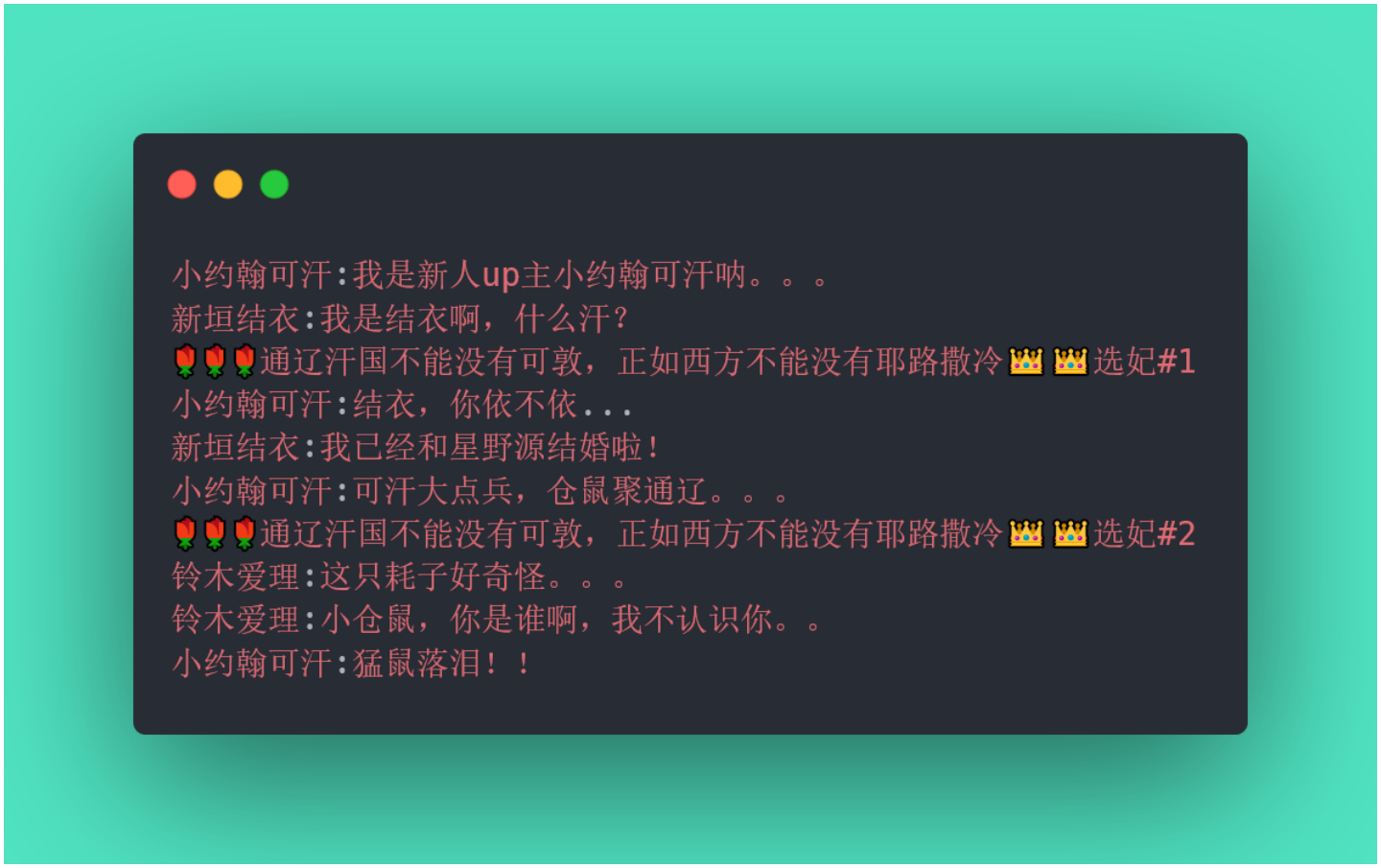
它和CountDownLatch类似，都可以协调多线程的结束动作，在它们结束后都可以执行特定动作，但是为什么要有CyclicBarrier，自然是它有和CountDownLatch不同的地方。

不知道你听没听过一个新人UP主小约翰可汗，小约翰生平有两大恨——“想结衣结衣不依,迷爱理爱理不理。”我们来还原一下事情的经过：小约翰在亲政后认识了新垣结衣，于是决定第一次选妃，向结衣表白，等待回应。然而新垣结衣回应嫁给了星野源，小约翰伤心欲绝，发誓生平不娶，突然发现了铃木爱理，于是小约翰决定第二次选妃，求爱理搭理，等待回应。

我们拿代码模拟这一场景，发现CountDownLatch无能为力了，因为CountDownLatch的使用是一次性的，无法重复利用，而这里等待了两次。此时，我们用CyclicBarrier就可以实现，因为它可以重复利用。

```java
public class CyclicBarrierTest {
    //第几次选妃
    private static String draftTime = "选妃#1";

    public static void main(String[] args) {
        //创建栅栏
        CyclicBarrier cyclicBarrier = new CyclicBarrier(2, () -> System.out.println("🌹🌹🌹通辽汗国不能没有可敦，正如西方不能没有耶路撒冷👑👑"+draftTime ));

        Thread 小约翰可汗 = new Thread(() -> {
                    say("我是新人up主小约翰可汗呐。。。");
                    try {
                        //第一次调用await
                        cyclicBarrier.await();
                        say("结衣，你依不依...");
                        Thread.sleep(2600); //等待结衣回应...
                        draftTime = "选妃#2"; // 第二次选妃
                        Thread.sleep(1500); //求爱理搭理中……
                        note("可汗大点兵，仓鼠聚通辽。。。");
                        cyclicBarrier.await(); // 可汗聚兵，仓鼠十万，选妃*2 ！！！注意这里是第二次调用await
                        Thread.sleep(100);
                        say("猛鼠落泪！！");
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }, "小约翰可汗");

        Thread 新垣结衣 = new Thread(() -> {
                    try {
                        Thread.sleep(500); //铠打野中
                        say("我是结衣啊，什么汗？");
                        cyclicBarrier.await(); //等待中。。
                        Thread.sleep(1000); //忽略了。。。
                        say("我已经和星野源结婚啦！"); //小约翰可汗突然看到了铃木爱理。。
                    } catch (Exception ignored) {}
                }, "新垣结衣");

        Thread 铃木爱理 = new Thread(() -> {
                    try {
                        Thread.sleep(2500);
                        cyclicBarrier.await();
                        note("这只耗子好奇怪。。。");
                        say("小仓鼠，你是谁啊，我不认识你。。");
                    } catch (Exception ignored) {}
                }, "铃木爱理");

        小约翰可汗.start();
        新垣结衣.start();
        铃木爱理.start();

    }

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

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

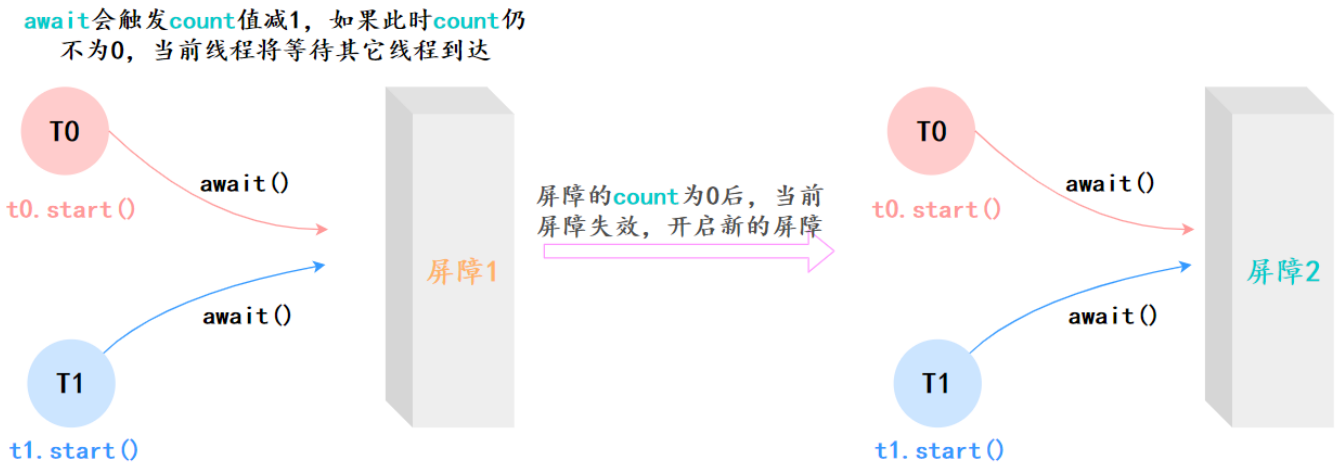
运行结果：



CyclicBarrier最最核心的方法，仍然是await()：

- 如果当前线程不是第一个到达屏障的话，它将会进入等待，直到其他线程都到达，除非发生被中断、屏障被拆除、屏障被重设等情况；

上面的例子抽象一下，本质上它的流程就是这样就是这样：



41.CyclicBarrier和CountDownLatch有什么区别?

两者最核心的区别[18]:

- CountDownLatch是一次性的，而CyclicBarrier则可以多次设置屏障，实现重复利用;
- CountDownLatch中的各个子线程不可以等待其他线程，只能完成自己的任务；而CyclicBarrier中的各个线程可以等待其他线程

它们区别用一个表格整理:

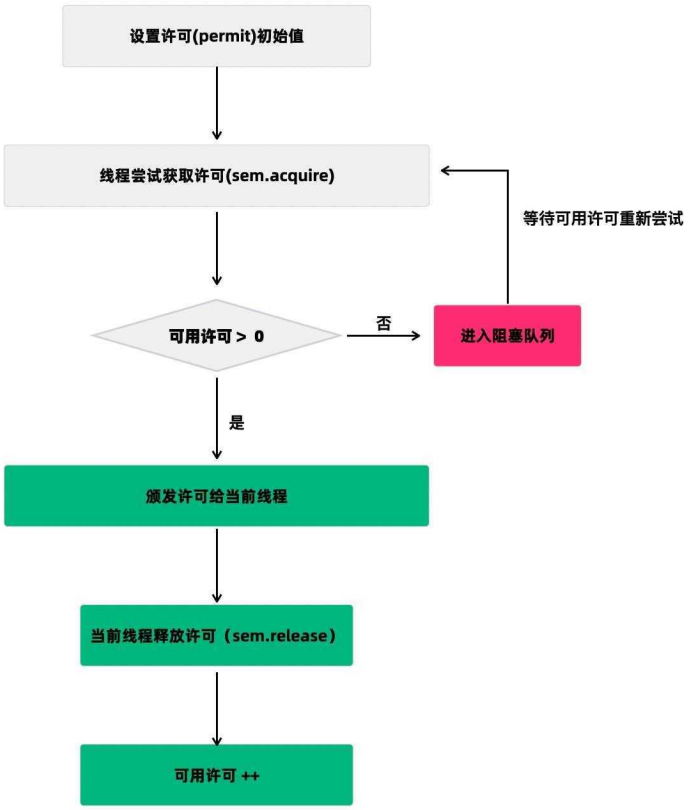
CyclicBarrier	CountDownLatch
CyclicBarrier是可重用的，其中的线程会等待所有的线程完成任务。届时，屏障将被拆除，并可以选择性地做一些特定的动作。	CountDownLatch是一次性的，不同的线程在同一个计数器上工作，直到计数器为0.
CyclicBarrier面向的是线程数	CountDownLatch面向的是任务数
在使用CyclicBarrier时，你必须在构造中指定参与协作的线程数，这些线程必须调用await()方法	使用CountDownLatch时，则必须要指定任务数，至于这些任务由哪些线程完成无关紧要
CyclicBarrier可以在所有的线程释放后重新使用	CountDownLatch在计数器为0时不能再使用
在CyclicBarrier中，如果某个线程遇到了中断、超时等问题时，则处于await的线程都会出现问题	在CountDownLatch中，如果某个线程出现问题，其他线程不受影响

42.Semaphore（信号量）了解吗?

Semaphore（信号量）是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源。

听起来似乎很抽象，现在汽车多了，开车出门在外的一个老大难问题就是停车 。停车场的车位是有限的，只能允许若干车辆停泊，如果停车场还有空位，那么显示牌显示的就是绿灯和剩余的车位，车辆就可以驶入；如果停车场没位了，那么显示牌显示的就是绿灯和数字0，车辆就得等待。如果满了的停车场有车离开，那么显示牌就又变绿，显示空车位数量，等待的车辆就能进停车场。

我们把这个例子类比一下，车辆就是线程，进入停车场就是线程在执行，离开停车场就是线程执行完毕，看见红灯就表示线程被阻塞，不能执行，Semaphore的本质就是协调多个线程对共享资源的获取。



我们再来看一个Semaphore的用途：它可以用于做流量控制，特别是公用资源有限的应用场景，比如数据库连接。

假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发地读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有10个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，就可以使用Semaphore来做流量控制，如下：

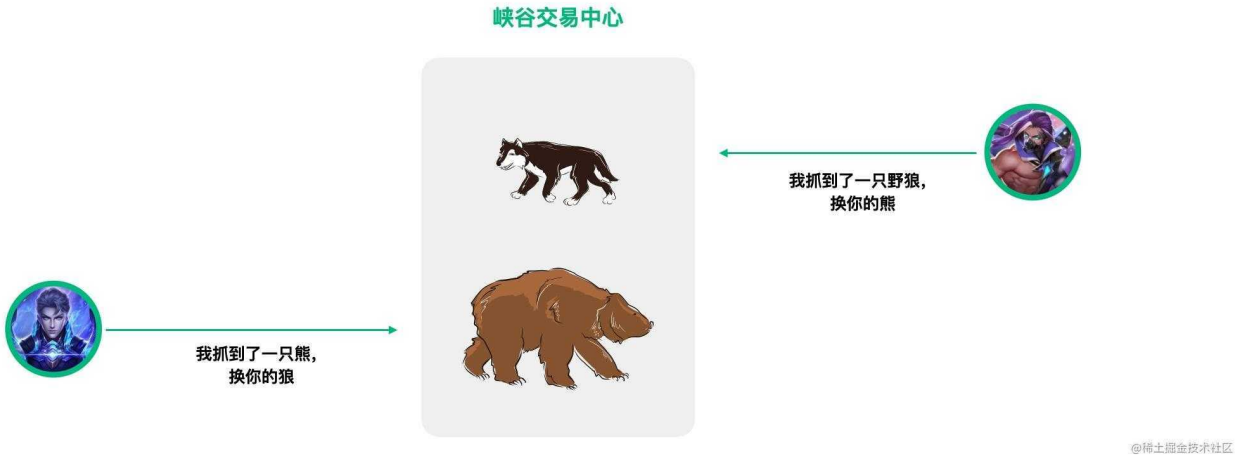
```
public class SemaphoreTest {
    private static final int THREAD_COUNT = 30;
    private static ExecutorService threadPool =
Executors.newFixedThreadPool(THREAD_COUNT);
    private static Semaphore s = new Semaphore(10);

    public static void main(String[] args) {
        for (int i = 0; i < THREAD_COUNT; i++) {
            threadPool.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        s.acquire();
                        System.out.println("save data");
```

```
                    s.release();
                } catch (InterruptedException e) {
                }
            }
        });
    }
    threadPool.shutdown();
  }
}
```

在代码中，虽然有30个线程在执行，但是只允许10个并发执行。Semaphore的构造方法`Semaphore(int permits)`接受一个整型的数字，表示可用的许可证数量。`Semaphore(10)`表示允许10个线程获取许可证，也就是最大并发数是10。Semaphore的用法也很简单，首先线程使用 Semaphore的acquire()方法获取一个许可证，使用完之后调用release()方法归还许可证。还可以用tryAcquire()方法尝试获取许可证。

43.Exchanger 了解吗?

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。



这两个线程通过 exchange方法交换数据，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

Exchanger可以用于遗传算法，遗传算法里需要选出两个人作为交配对象，这时候会交换两人的数据，并使用交叉规则得出2个交配结果。Exchanger也可以用于校对工作，比如我们需要将纸制银行流水通过人工的方式录入成电子银行流水，为了避免错误，采用AB岗两人进行录入，录入到Excel之后，系统需要加载这两个Excel，并对两个Excel数据进行校对，看看是否录入一致。

```
public class ExchangerTest {
    private static final Exchanger<String> exgr = new Exchanger<String>();
    private static ExecutorService threadPool = Executors.newFixedThreadPool(2);

    public static void main(String[] args) {
        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    String A = "银行流水A"; // A录入银行流水数据
                    exgr.exchange(A);
                } catch (InterruptedException e) {
                }
            }
        });
        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    String B = "银行流水B"; // B录入银行流水数据
                    String A = exgr.exchange("B");
                    System.out.println("A和B数据是否一致：" + A.equals(B) + "，A录入的是："
                            + A + "，B录入是：" + B);
                } catch (InterruptedException e) {
                }
            }
        });
        threadPool.shutdown();
    }
}
```

假如两个线程有一个没有执行exchange()方法，则会一直等待，如果担心有特殊情况发生，避免一直等待，可以使用`exchange(V x, long timeOut, TimeUnit unit)`设置最大等待时长。

线程池

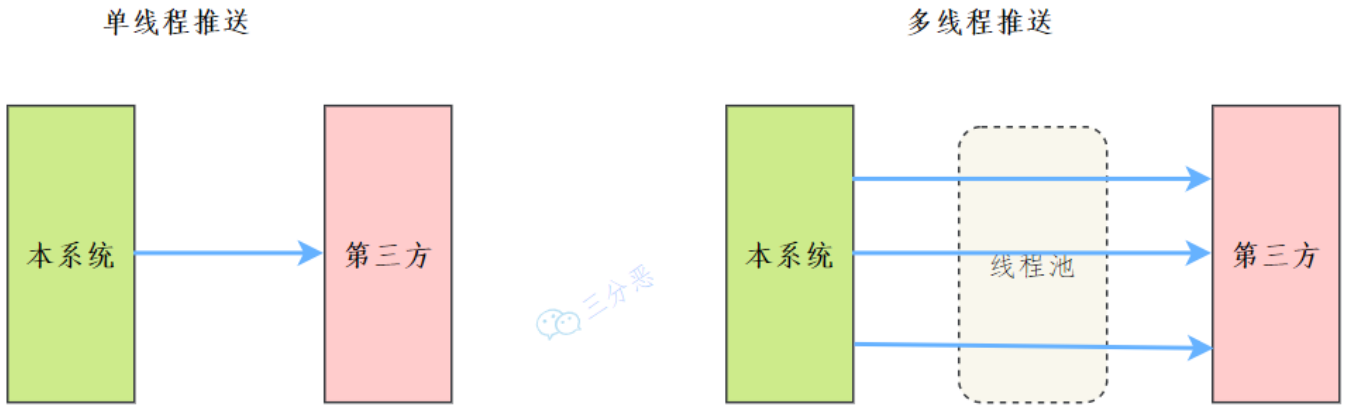
44.什么是线程池？

线程池： 简单理解，它就是一个管理线程的池子。

- 它帮我们管理线程，避免增加创建线程和销毁线程的资源损耗。因为线程其实也是一个对象，创建一个对象，需要经过类加载过程，销毁一个对象，需要走GC垃圾回收流程，都是需要资源开销的。
- 提高响应速度。 如果任务到达了，相对于从线程池拿线程，重新去创建一条线程执行，速度肯定慢很多。
- 重复利用。 线程用完，再放回池子，可以达到重复利用的效果，节省资源。

45.能说说工作中线程池的应用吗?

之前我们有一个和第三方对接的需求，需要向第三方推送数据，引入了多线程来提升数据推送的效率，其中用到了线程池来管理线程。



主要代码如下：

```
@Service
public class PushProcessServiceImpl implements PushProcessService {
    @Autowired
    private PushUtil pushUtil;
    @Autowired
    private PushProcessMapper pushProcessMapper;

    private final static Logger logger = LoggerFactory.getLogger(PushProcessServiceImpl.class);

    //每个线程每次查询的条数
    private static final Integer LIMIT = 5000;
    //核心线程数:设置为操作系统CPU数乘以2
    private static final Integer CORE_POOL_SIZE = Runtime.getRuntime().availableProcessors() * 2;
    //最大线程数:设置为和核心线程数相同
    private static final Integer MAXIMUM_POOl_SIZE = CORE_POOL_SIZE;
    //创建线程池
    ThreadPoolExecutor pool = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOl_SIZE * 2, 0,
TimeUnit.SECONDS, new LinkedBlockingQueue<>(100));

    @Override
    public void pushData() throws ExecutionException, InterruptedException {
        //计数器，需要保证线程安全
        int count = 0;
        //未推送数据总数
        Integer total = pushProcessMapper.countPushRecordsByState(0);
        logger.info("未推送数据条数：{}", total);
        //计算需要多少轮
        int num = total / (LIMIT * CORE_POOL_SIZE) + 1;
        logger.info("要经过的轮数:{}", num);
        //统计总共推送成功的数据条数
        int totalSuccessCount = 0;
        for (int i = 0; i < num; i++) {
            //接收线程返回结果
            List<Future<Integer>> futureList = new ArrayList<>(32);
            //起CORE_POOL_SIZE个线程并行查询更新库，加锁
            for (int j = 0; j < CORE_POOL_SIZE; j++) {
                synchronized (PushProcessServiceImpl.class) {
```

```
                    int start = count * LIMIT;
                    count++;
                    //提交线程，用数据起始位置标识线程
                    Future<Integer> future = pool.submit(new PushDataTask(start, LIMIT, start));
                    //先不取值，防止阻塞,放进集合
                    futureList.add(future);
                }
            }
            //统计本轮推送成功数据
            for (Future f : futureList) {
                totalSuccessCount = totalSuccessCount + (int) f.get();
            }
        }
        //关闭线程池
        //pool.shutdown();
        //更新推送标志
        pushProcessMapper.updateAllState(1);
        logger.info("推送数据完成，需推送数据:{},推送成功：{}", total, totalSuccessCount);
    }

    /**
     * 推送数据线程类
     */
    class PushDataTask implements Callable<Integer> {
        int start;
        int limit;

        PushDataTask(int start, int limit, int threadNo) {
            this.start = start;
            this.limit = limit;
        }

        @Override
        public Integer call() throws Exception {
            //设置线程名字
            Thread.currentThread().setName("Thread" + start);
            int count = 0;
            //推送的数据
            List<PushProcess> pushProcessList = pushProcessMapper.findPushRecordsByStateLimit(0, start,
limit);
            if (CollectionUtils.isEmpty(pushProcessList)) {
                return count;
            }
            logger.info("推送区间：[{},{}]数据", start, start + limit);
            for (PushProcess process : pushProcessList) {
                boolean isSuccess = pushUtil.sendRecord(process);
                if (isSuccess) {   //推送成功
                    //更新推送标识
                    pushProcessMapper.updateFlagById(process.getId(), 1);
                    count++;
                } else {  //推送失败
                    pushProcessMapper.updateFlagById(process.getId(), 2);
                }
            }
            logger.info("推送区间[{},{}]数据,推送成功{}条！", start, start + limit, count);
            return count;
        }
    }
}
```

完整可运行代码地址：https://gitee.com/fighter3/thread-demo.git

线程池的参数如下：

- corePoolSize：线程核心参数选择了CPU数×2

- maximumPoolSize：最大线程数选择了和核心线程数相同
- keepAliveTime：非核心闲置线程存活时间直接置为0
- unit：非核心线程保持存活的时间选择了 TimeUnit.SECONDS 秒
- workQueue：线程池等待队列，使用 LinkedBlockingQueue阻塞队列

同时还用了synchronized 来加锁，保证数据不会被重复推送：

```
synchronized (PushProcessServiceImpl.class) {}
```

ps:这个例子只是简单地进行了数据推送，实际上还可以结合其他的业务，像什么数据清洗啊、数据统计啊，都可以套用。

46.能简单说一下线程池的工作流程吗?

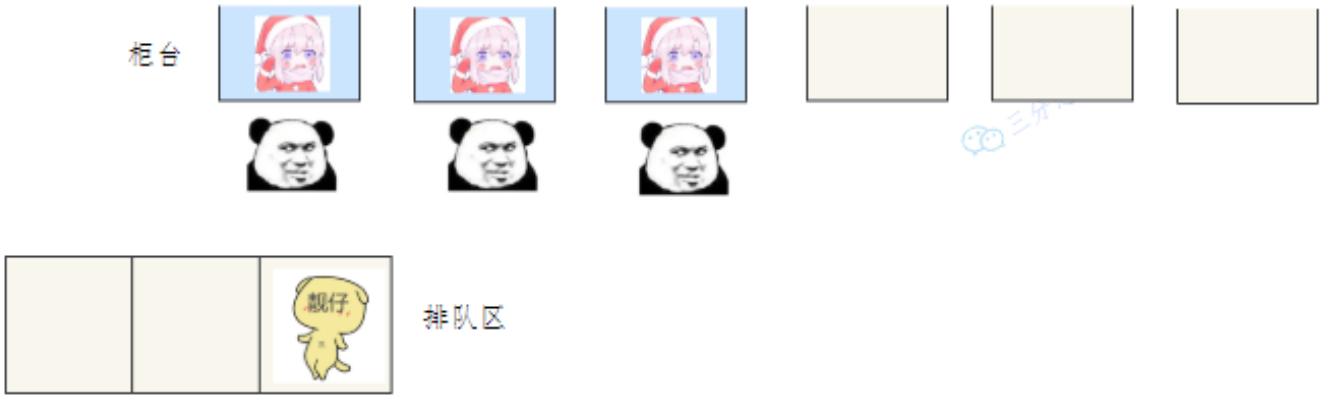
用一个通俗的比喻：

有一个营业厅，总共有六个窗口，现在开放了三个窗口，现在有三个窗口坐着三个营业员小姐姐在营业。

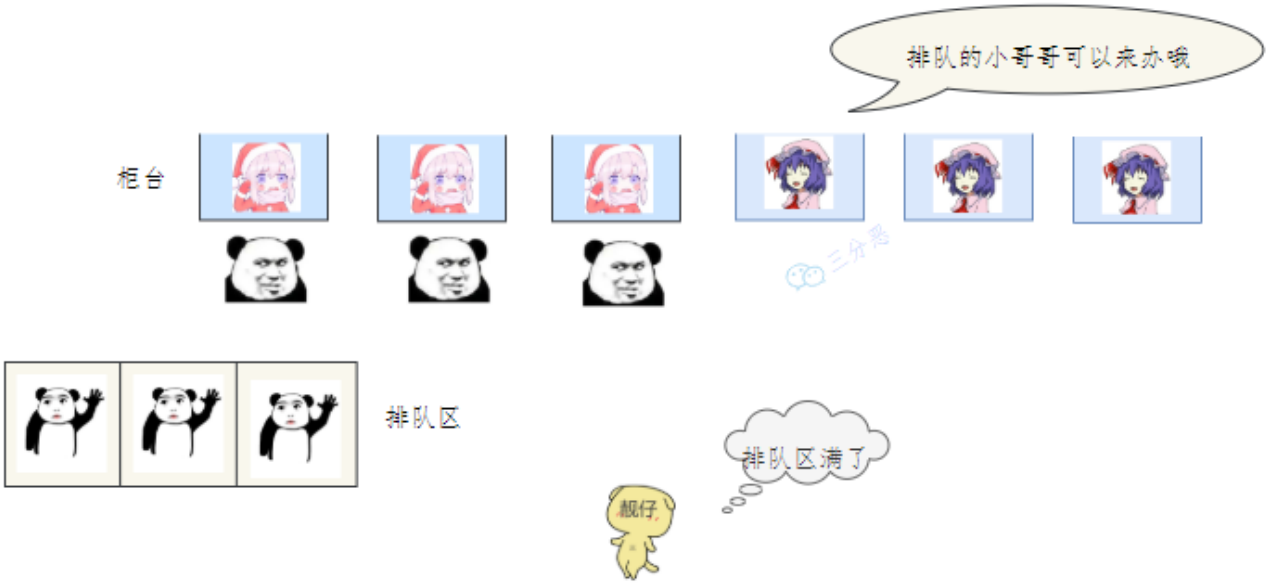
老三去办业务，可能会遇到什么情况呢?

1. 老三发现有空间的在营业的窗口，直接去找小姐姐办理业务。

2. 老三发现没有空闲的窗口，就在排队区排队等。



3．老三发现没有空闲的窗口，等待区也满了，蚌埠住了，经理一看，就让休息的小姐姐赶紧回来上班，等待区号靠前的赶紧去新窗口办，老三去排队区排队。小姐姐比较辛苦，假如一段时间发现他们可以不用接着营业，经理就让她们接着休息。



4．老三一看，六个窗口都满了，等待区也没位置了。老三急了，要闹，经理赶紧出来了，经理该怎么办呢？

1. 我们银行系统已经瘫痪
2. 谁叫你来办的你找谁去
3. 看你比较急，去队里加个塞
4. 今天没办法，不行你看改一天

上面的这个流程几乎就跟 JDK 线程池的大致流程类似，

1. 营业中的 3个窗口对应核心线程池数：corePoolSize
2. 总的营业窗口数6对应：maximumPoolSize
3. 打开的临时窗口在多少时间内无人办理则关闭对应：unit
4. 排队区就是等待队列：workQueue
5. 无法办理的时候银行给出的解决方法对应：RejectedExecutionHandler
6. threadFactory 该参数在 JDK 中是 线程工厂，用来创建线程对象，一般不会动。

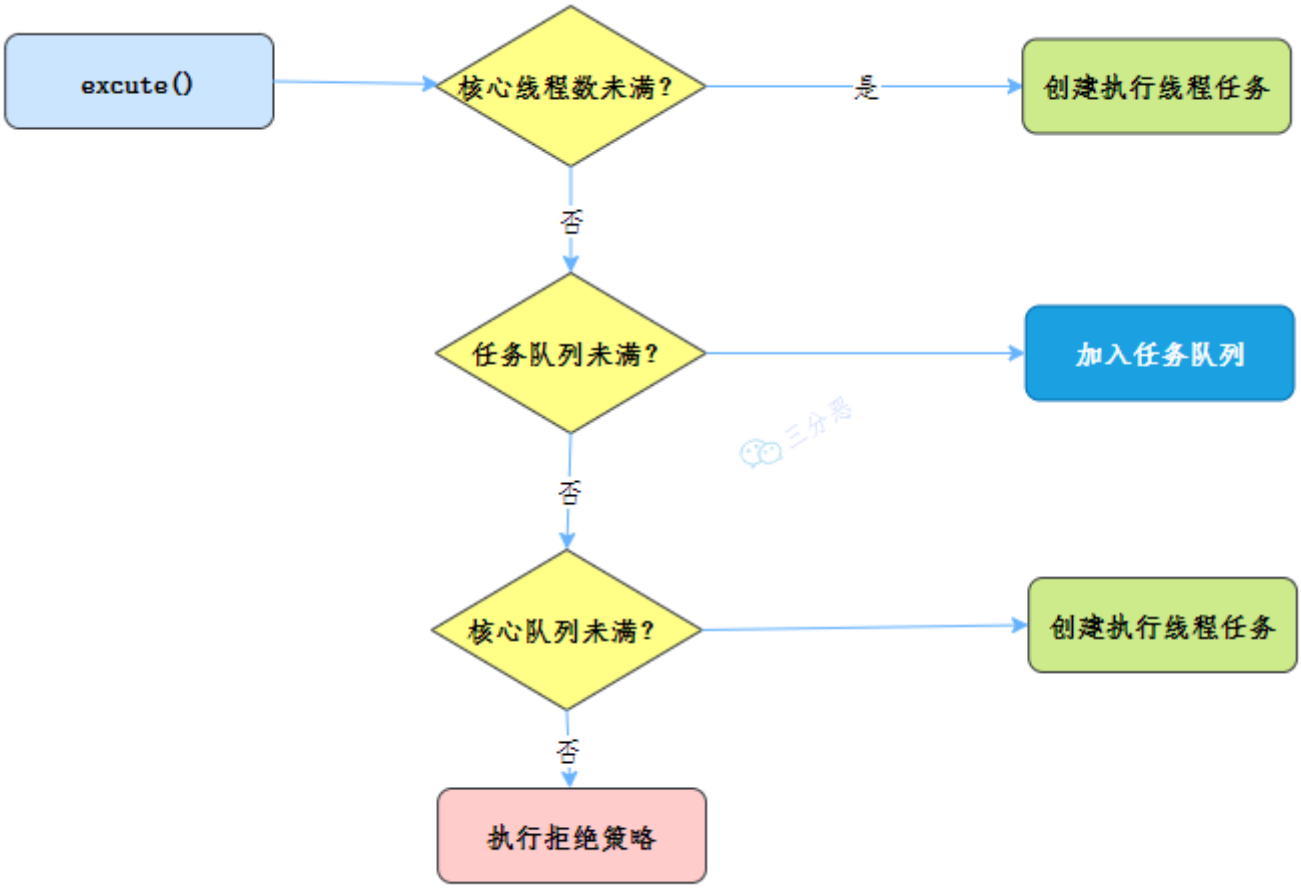
所以我们线程池的工作流程也比较好理解了：

1. 线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面有任务，线程池也不会马上执行它们。
2. 当调用 execute() 方法添加一个任务时，线程池会做如下判断：

- 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；
- 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；
- 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心

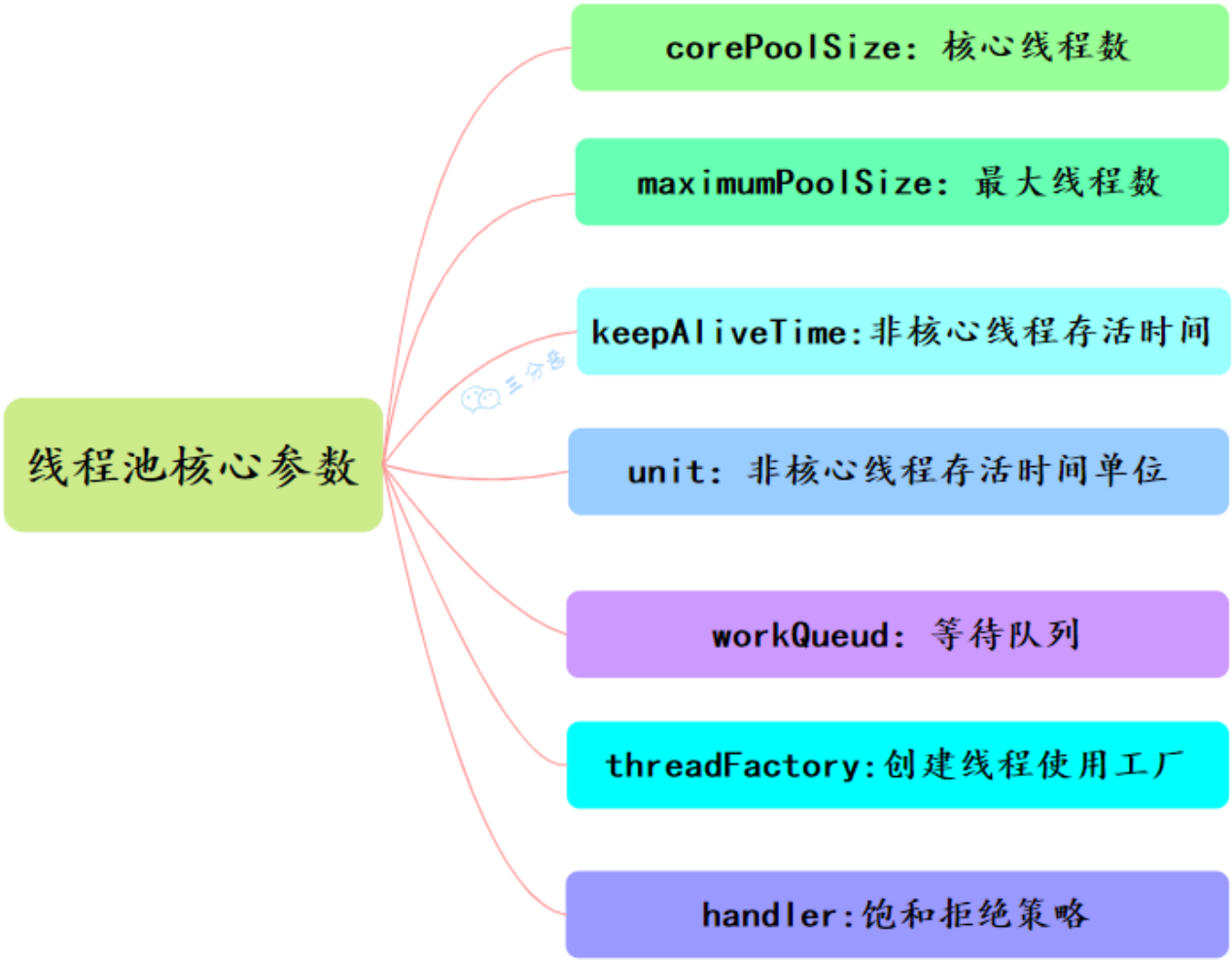
线程立刻运行这个任务；

- 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会根据拒绝策略来对应处理。



3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

47.线程池主要参数有哪些?



线程池有七大参数，需要重点关注`corePoolSize`、`maximumPoolSize`、`workQueue`、`handler`这四个。

1. corePoolSize

此值是用来初始化线程池中核心线程数，当线程池中线程池数< `corePoolSize`时，系统默认是添加一个任务才创建一个线程池。当线程数 = corePoolSize时，新任务会追加到workQueue中。

2. maximumPoolSize

`maximumPoolSize`表示允许的最大线程数 =（非核心线程数+核心线程数），当`BlockingQueue`也满了，但线程池中总线程数 < `maximumPoolSize`时候就会再次创建新的线程。

3. keepAliveTime

非核心线程 =(maximumPoolSize - corePoolSize),非核心线程闲置下来不干活最多存活时间。

4. unit

线程池中非核心线程保持存活的时间的单位

- TimeUnit.DAYS; 天
- TimeUnit.HOURS; 小时
- TimeUnit.MINUTES; 分钟
- TimeUnit.SECONDS; 秒
- TimeUnit.MILLISECONDS; 毫秒
- TimeUnit.MICROSECONDS; 微秒
- TimeUnit.NANOSECONDS; 纳秒

5. workQueue

线程池等待队列，维护着等待执行的`Runnable`对象。当运行当线程数= corePoolSize时，新的任务会被添加到`workQueue`中，如果`workQueue`也满了则尝试用非核心线程执行任务，等待队列应该尽量用有界的。

6. threadFactory

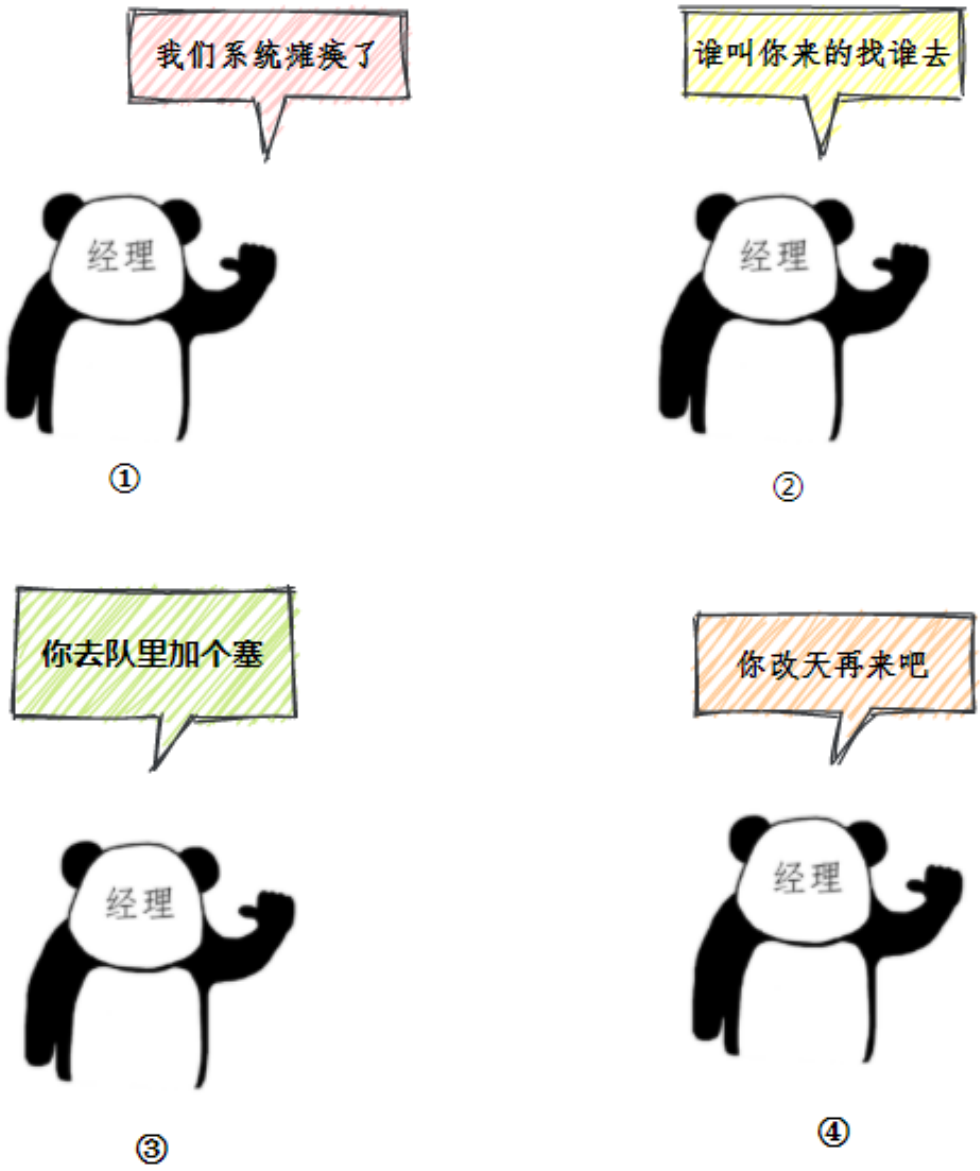
创建一个新线程时使用的工厂，可以用来设定线程名、是否为daemon线程等等。

7. handler

`corePoolSize`、`workQueue`、`maximumPoolSize`都不可用的时候执行的饱和策略。

48.线程池的拒绝策略有哪些?

类比前面的例子，无法办理业务时的处理方式，帮助记忆:

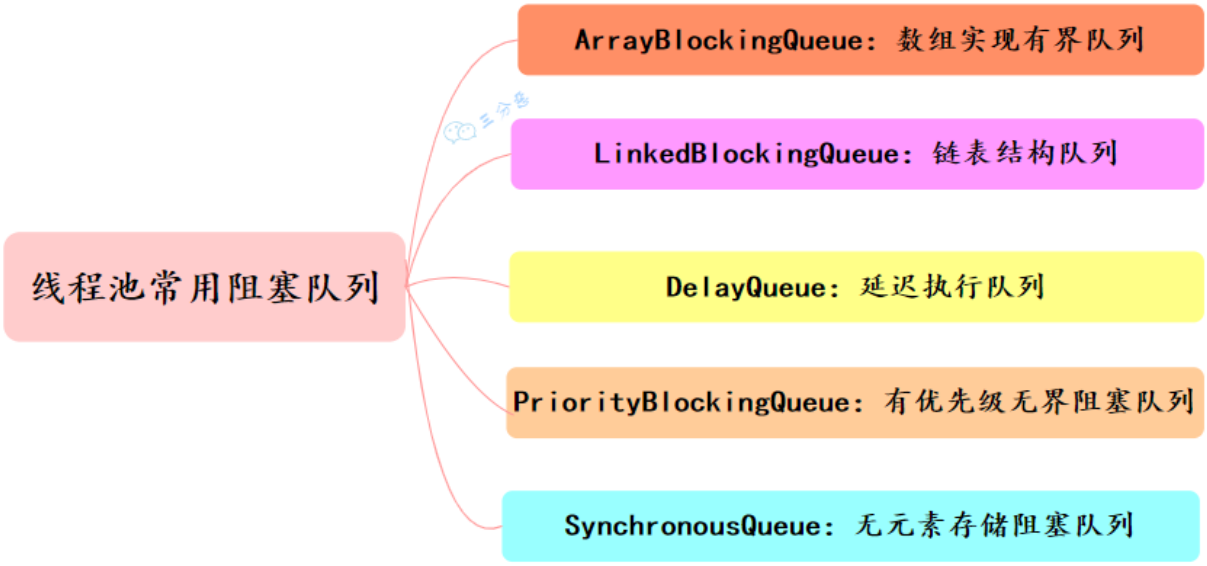


- AbortPolicy ： 直接抛出异常，默认使用此策略
- CallerRunsPolicy：用调用者所在的线程来执行任务
- DiscardOldestPolicy：丢弃阻塞队列里最老的任务，也就是队列里靠前的任务
- DiscardPolicy ： 当前任务直接丢弃

想实现自己的拒绝策略，实现RejectedExecutionHandler接口即可。

49.线程池有哪几种工作队列?

常用的阻塞队列主要有以下几种：



- ArrayBlockingQueue：ArrayBlockingQueue（有界队列）是一个用数组实现的有界阻塞队列，按FIFO排序量。
- LinkedBlockingQueue：LinkedBlockingQueue（可设置容量队列）是基于链表结构的阻塞队列，按FIFO排序任务，容量可以选择进行设置，不设置的话，将是一个无边界的阻塞队列，最大长度为Integer.MAX_VALUE，吞吐量通常要高于ArrayBlockingQuene；newFixedThreadPool线程池使用了这个队列
- DelayQueue：DelayQueue（延迟队列）是一个任务定时周期的延迟执行的队列。根据指定的执行时间从小到大排序，否则根据插入到队列的先后排序。newScheduledThreadPool线程池使用了这个队列。
- PriorityBlockingQueue：PriorityBlockingQueue（优先级队列）是具有优先级的无界阻塞队列
- SynchronousQueue：SynchronousQueue（同步队列）是一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene，newCachedThreadPool线程池使用了这个队列。

50.线程池提交execute和submit有什么区别？

1. execute 用于提交不需要返回值的任务

```
threadsPool.execute(new Runnable() {
    @Override public void run() {
        // TODO Auto-generated method stub }
    });
```

2. 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

```
Future<Object> future = executor.submit(harReturnValuetask);
try { Object s = future.get(); } catch (InterruptedException e) {
    // 处理中断异常
} catch (ExecutionException e) {
    // 处理无法执行任务异常
} finally {
    // 关闭线程池 executor.shutdown();
}
```

51.线程池怎么关闭知道吗?

可以通过调用线程池的`shutdown`或`shutdownNow`方法来关闭线程池。它们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止。

shutdown() 将线程池状态置为shutdown,并不会立即停止:

1. 停止接收外部submit的任务
2. 内部正在跑的任务和队列里等待的任务，会执行完
3. 等到第二步完成后，才真正停止

shutdownNow() 将线程池状态置为stop。一般会立即停止，事实上不一定:

1. 和shutdown()一样，先停止接收外部提交的任务
2. 忽略队列里等待的任务
3. 尝试将正在跑的任务interrupt中断
4. 返回未执行的任务列表

shutdown 和shutdownnow简单来说区别如下:

- shutdownNow()能立即停止线程池，正在跑的和正在等待的任务都停下了。这样做立即生效，但是风险也比较大。
- shutdown()只是关闭了提交通道，用submit()是无效的；而内部的任务该怎么跑还是怎么跑，跑完再彻底停止线程池。

52.线程池的线程数应该怎么配置?

线程在Java中属于稀缺资源，线程池不是越大越好也不是越小越好。任务分为计算密集型、IO密集型、混合型。

1. 计算密集型：大部分都在用CPU跟内存，加密，逻辑操作业务处理等。
2. IO密集型：数据库链接，网络通讯传输等。

方案	问题
$N_{cpu} = number\ of\ CPUs$ $U_{cpu} = target\ CPU\ utilization, 0 \le U_{cpu} \le 1$ $\frac{W}{C} = ratio\ of\ wait\ time\ to\ compute\ time$ *The optimal pool size for keeping the processors at the desired utilization is :* $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

一般的经验，不同类型线程池的参数配置：

1. 计算密集型一般推荐线程池不要过大，一般是CPU数 + 1，+1是因为可能存在页缺失(就是可能存在有些数据在硬盘中需要多来一个线程将数据读入内存)。如果线程池数太大，可能会频繁的 进行线程上下文切换跟任务调度。获得当前CPU核心数代码如下：

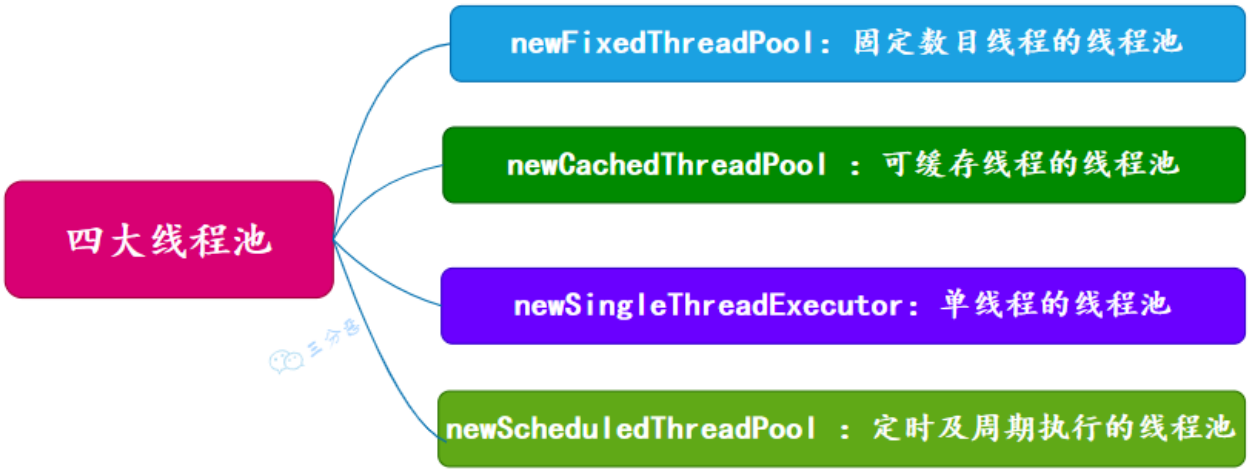
```
Runtime.getRuntime().availableProcessors();
```

2. IO密集型：线程数适当大一点，机器的Cpu核心数*2。
3. 混合型：可以考虑根绝情况将它拆分成CPU密集型和IO密集型任务，如果执行时间相差不大，拆分可以提升吞吐量，反之没有必要。

当然，实际应用中没有固定的公式，需要结合测试和监控来进行调整。

53.有哪几种常见的线程池？

面试常问，主要有四种，都是通过工具类Excutors创建出来的，需要注意，阿里巴巴《Java开发手册》里禁止使用这种方式来创建线程池。



- newFixedThreadPool （固定数目线程的线程池）
- newCachedThreadPool（可缓存线程的线程池）
- newSingleThreadExecutor（单线程的线程池）
- newScheduledThreadPool（定时及周期执行的线程池）

54.能说一下四种常见线程池的原理吗？

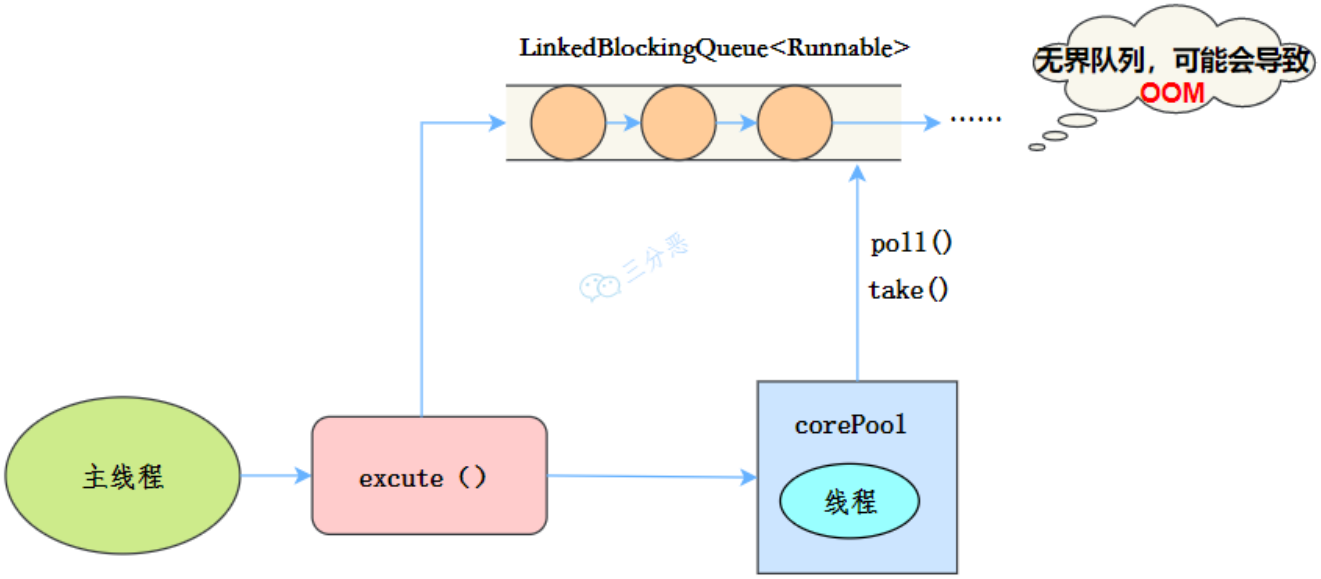
前三种线程池的构造直接调用ThreadPoolExecutor的构造方法。

newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>(),
                                threadFactory));
}
```

线程池特点

- 核心线程数为1
- 最大线程数也为1
- 阻塞队列是无界队列LinkedBlockingQueue，可能会导致OOM
- keepAliveTime为0



工作流程：

- 提交任务
- 线程池是否有一条线程在，如果没有，新建线程执行任务
- 如果有，将任务加到阻塞队列
- 当前的唯一线程，从队列取任务，执行完一个，再继续取，一个线程执行任务。

适用场景

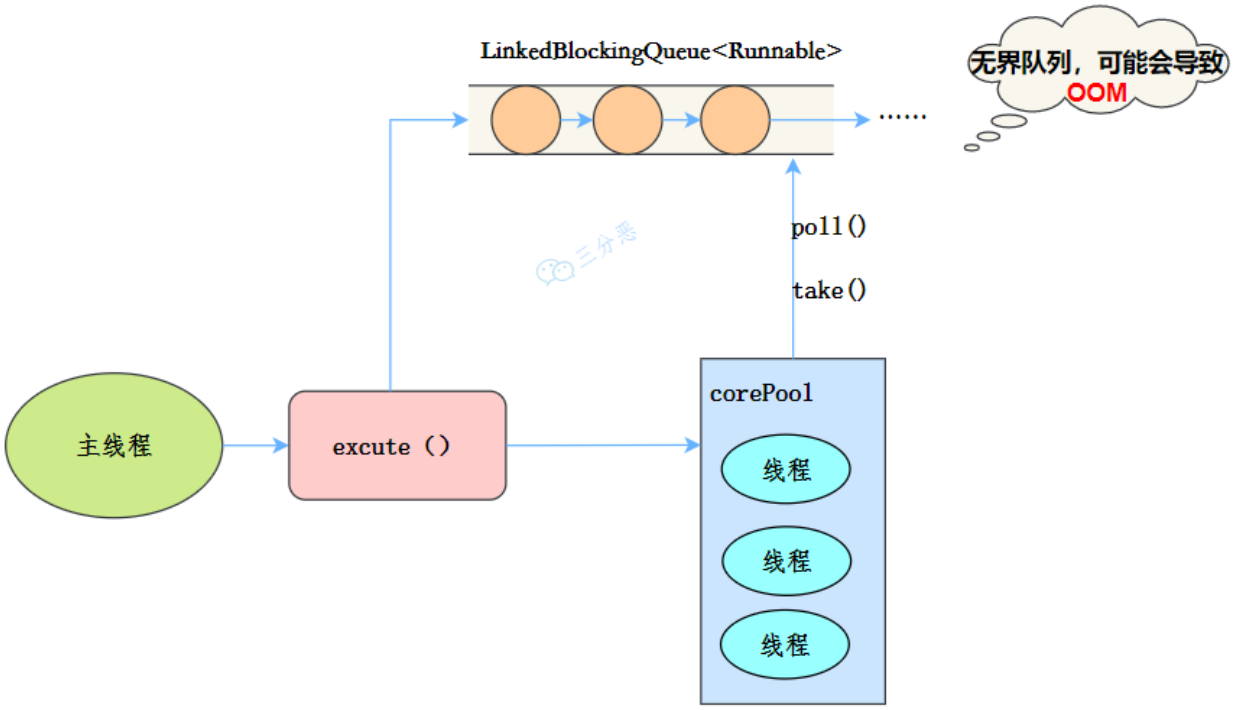
适用于串行执行任务的场景，一个任务一个任务地执行。

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>(),
                                  threadFactory);
}
```

线程池特点：

- 核心线程数和最大线程数大小一样
- 没有所谓的非空闲时间，即keepAliveTime为0
- 阻塞队列为无界队列LinkedBlockingQueue，可能会导致OOM



工作流程：

- 提交任务
- 如果线程数少于核心线程，创建核心线程执行任务
- 如果线程数等于核心线程，把任务添加到LinkedBlockingQueue阻塞队列
- 如果线程执行完任务，去阻塞队列取任务，继续执行。

使用场景

FixedThreadPool 适用于处理CPU密集型的任务，确保CPU在长期被工作线程使用的情况下，尽可能的少的分配线程，即适用执行长期的任务。

newCachedThreadPool

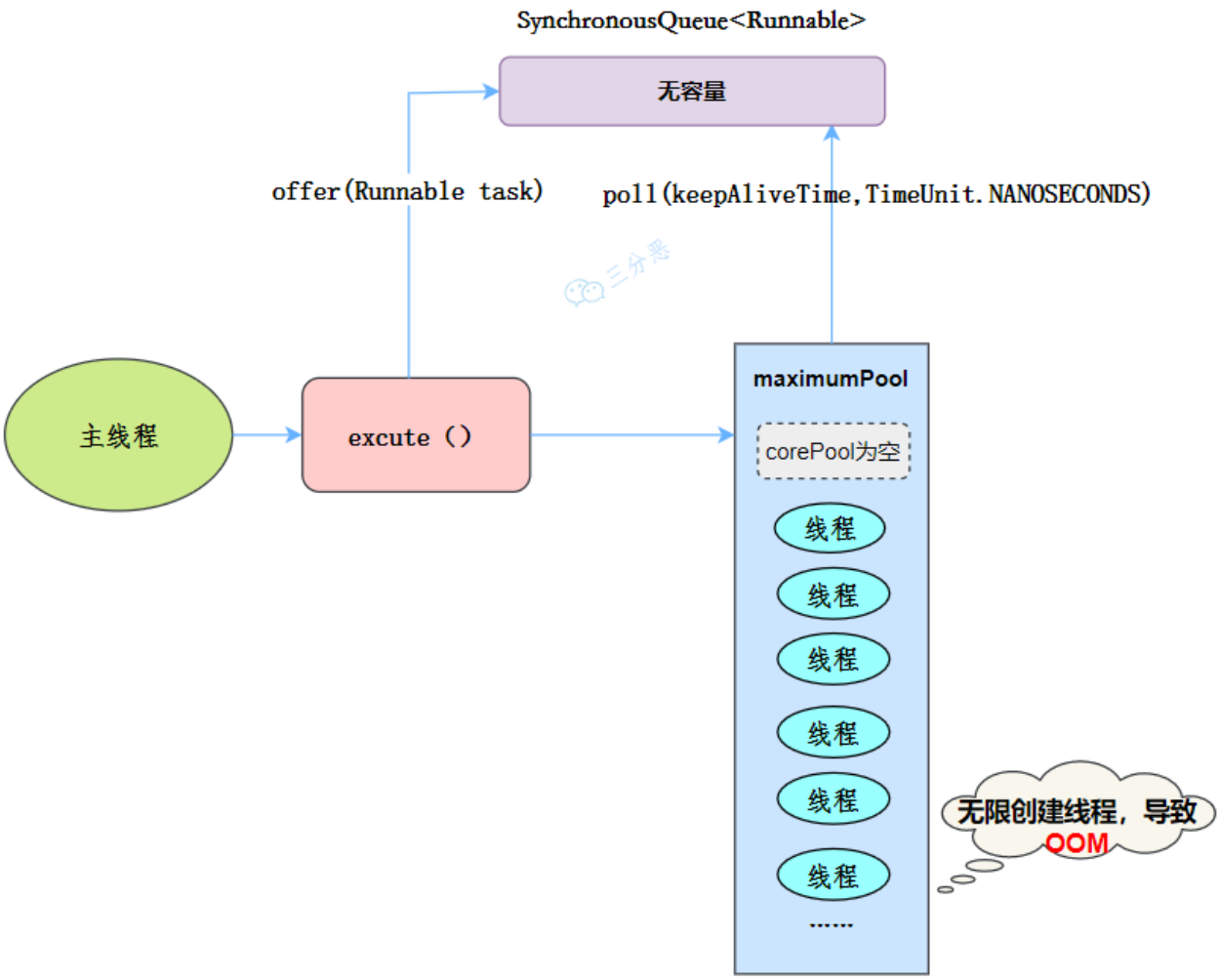
```
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>(),
                                  threadFactory);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

线程池特点：

- 核心线程数为0
- 最大线程数为Integer.MAX_VALUE，即无限大，可能会因为无限创建线程，导致OOM
- 阻塞队列是SynchronousQueue

- 非核心线程空闲存活时间为60秒

当提交任务的速度大于处理任务的速度时，每次提交一个任务，就必然会创建一个线程。极端情况下会创建过多的线程，耗尽 CPU 和内存资源。由于空闲 60 秒的线程会被终止，长时间保持空闲的 CachedThreadPool 不会占用任何资源。



工作流程：

- 提交任务
- 因为没有核心线程，所以任务直接加到SynchronousQueue队列。
- 判断是否有空闲线程，如果有，就去取出任务执行。
- 如果没有空闲线程，就新建一个线程执行。
- 执行完任务的线程，还可以存活60秒，如果在这期间，接到任务，可以继续活下去；否则，被销毁。

适用场景

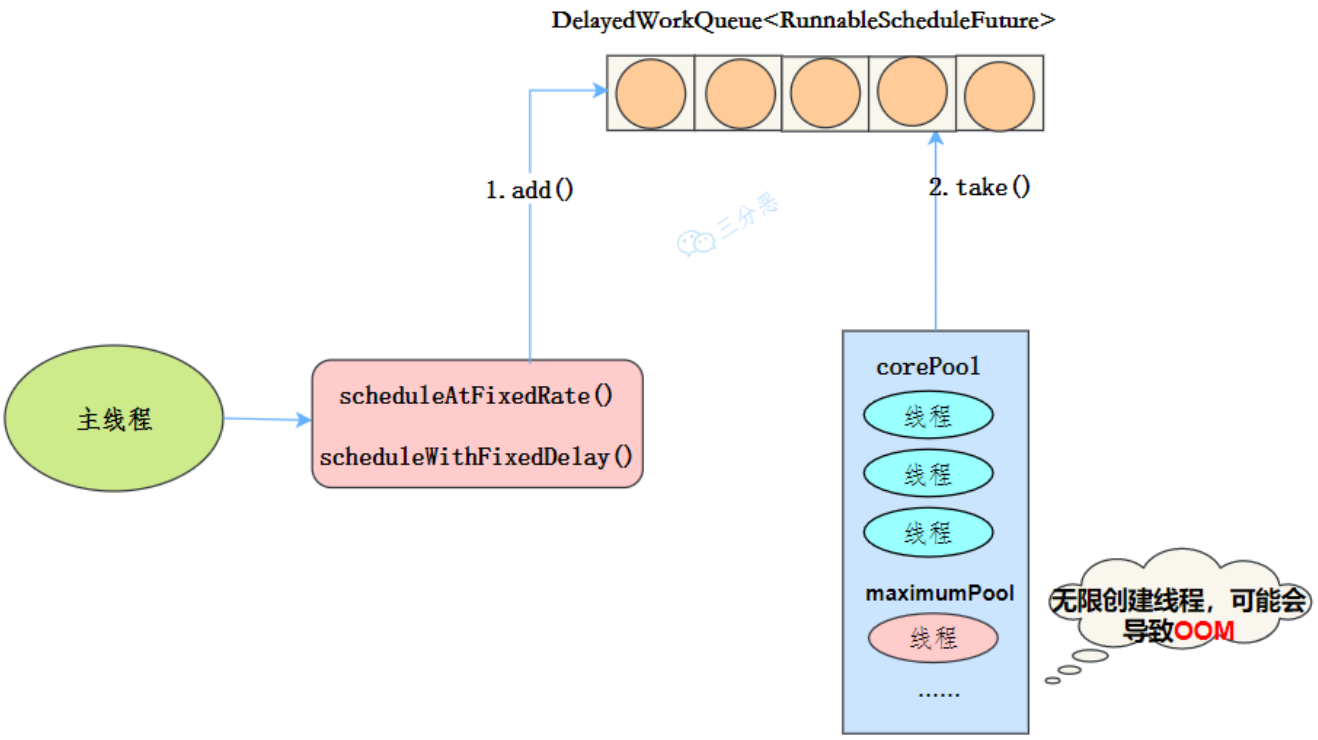
用于并发执行大量短期的小任务。

newScheduledThreadPool

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
   super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
         new DelayedWorkQueue());
}
```

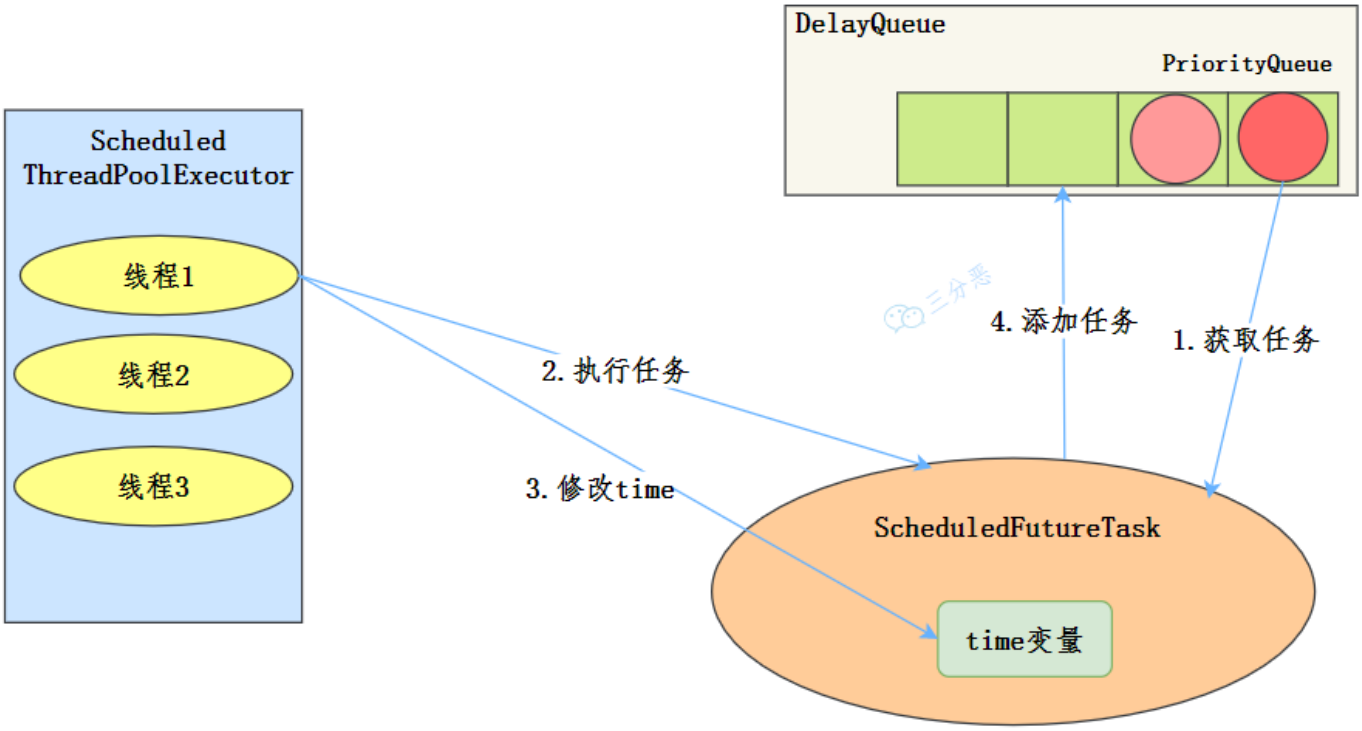
线程池特点

- 最大线程数为Integer.MAX_VALUE，也有OOM的风险
- 阻塞队列是DelayedWorkQueue
- keepAliveTime为0
- scheduleAtFixedRate()：按某种速率周期执行
- scheduleWithFixedDelay()：在某个延迟后执行



工作机制

- 线程从DelayQueue中获取已到期的ScheduledFutureTask（DelayQueue.take()）。到期任务是指ScheduledFutureTask的time大于等于当前时间。
- 线程执行这个ScheduledFutureTask。
- 线程修改ScheduledFutureTask的time变量为下次将要被执行的时间。
- 线程把这个修改time之后的ScheduledFutureTask放回DelayQueue中（DelayQueue.add()）。



使用场景

周期性执行任务的场景，需要限制线程数量的场景

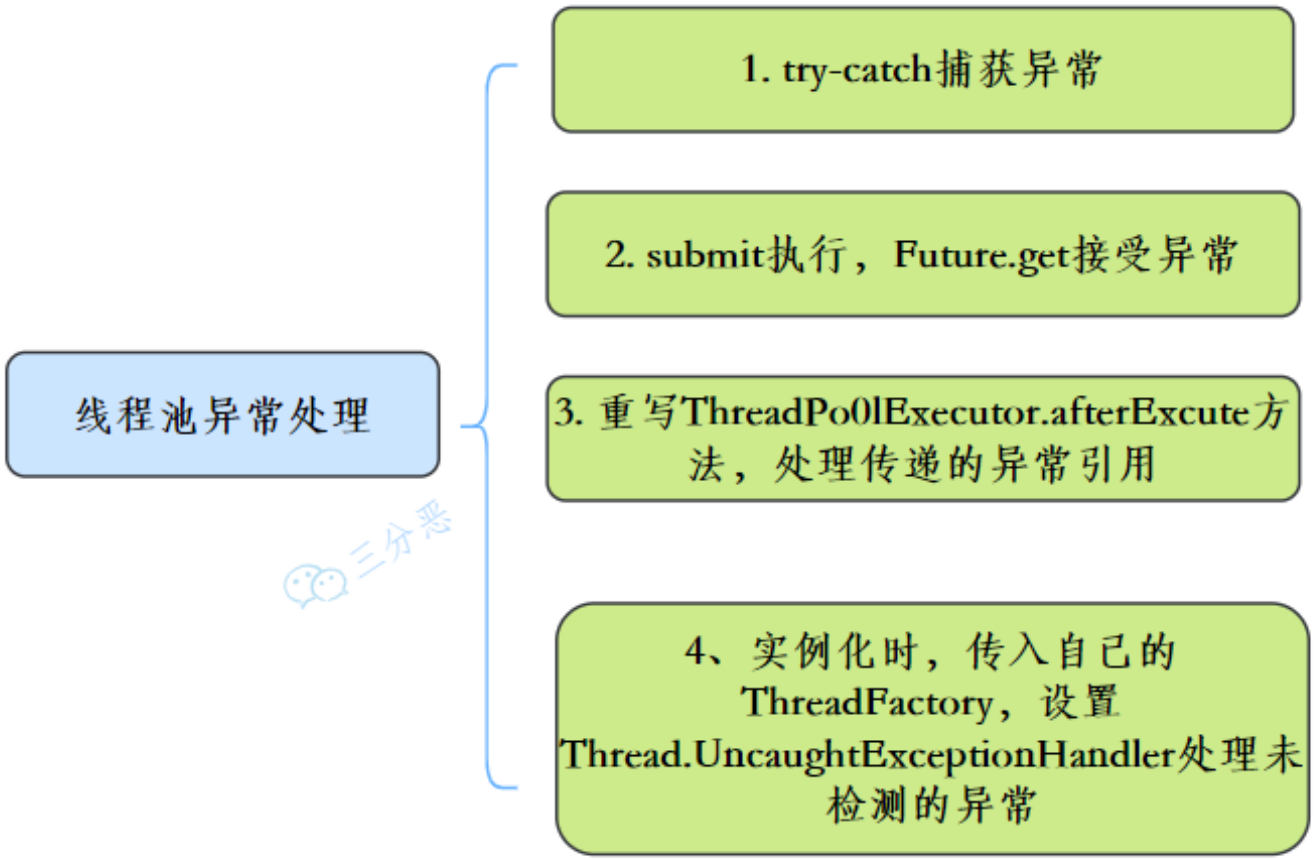
> 使用无界队列的线程池会导致什么问题吗?

例如newFixedThreadPool使用了无界的阻塞队列LinkedBlockingQueue，如果线程获取一个任务后，任务的执行时间比较长，会导致队列的任务越积越多，导致机器内存使用不停飙升，最终导致OOM。

55.线程池异常怎么处理知道吗?

在使用线程池处理任务的时候，任务代码可能抛出RuntimeException，抛出异常后，线程池可能捕获它，也可能创建一个新的线程来代替异常的线程，我们可能无法感知任务出现了异常，因此我们需要考虑线程池异常情况。

常见的异常处理方式:

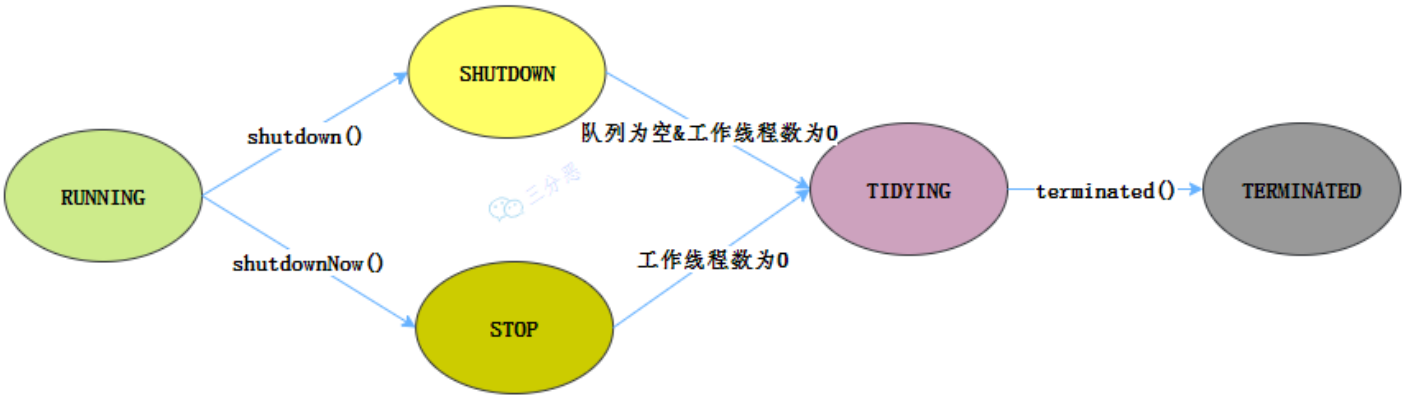


56.能说一下线程池有几种状态吗?

线程池有这几个状态：RUNNING,SHUTDOWN,STOP,TIDYING,TERMINATED。

```
//线程池状态
private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;
```

线程池各个状态切换图：



RUNNING

- 该状态的线程池会接收新任务，并处理阻塞队列中的任务;
- 调用线程池的shutdown()方法，可以切换到SHUTDOWN状态;
- 调用线程池的shutdownNow()方法，可以切换到STOP状态;

SHUTDOWN

- 该状态的线程池不会接收新任务，但会处理阻塞队列中的任务;
- 队列为空，并且线程池中执行的任务也为空,进入TIDYING状态;

STOP

- 该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务;
- 线程池中执行的任务为空,进入TIDYING状态;

TIDYING

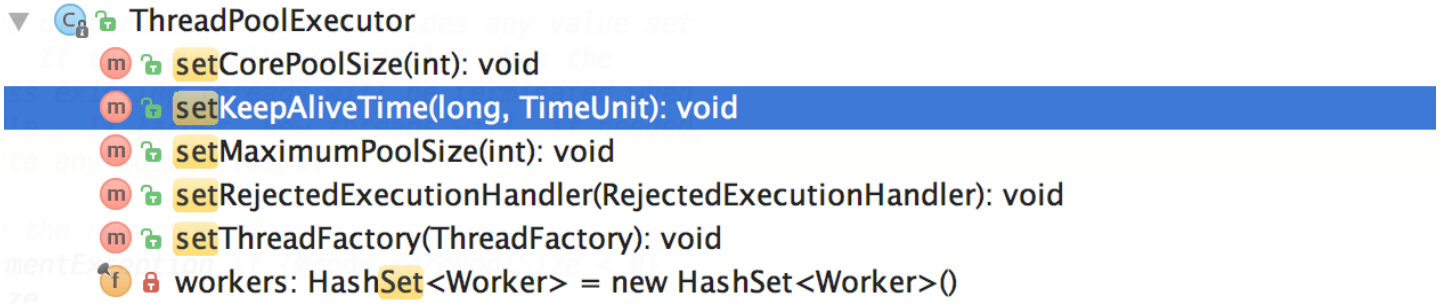
- 该状态表明所有的任务已经运行终止，记录的任务数量为0。
- terminated()执行完毕，进入TERMINATED状态

TERMINATED

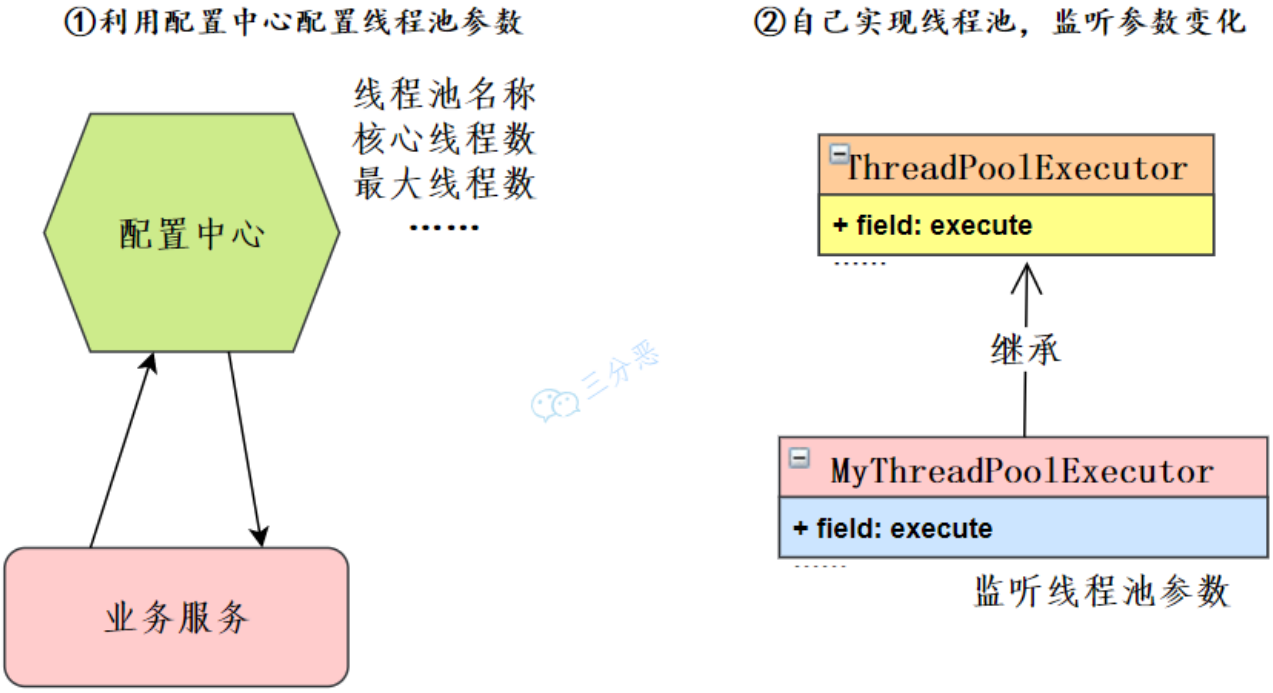
- 该状态表示线程池彻底终止

57.线程池如何实现参数的动态修改?

线程池提供了几个 setter方法来设置线程池的参数。



这里主要有两个思路：



- 在我们微服务的架构下，可以利用配置中心如Nacos、Apollo等等，也可以自己开发配置中心。业务服务读取线程池配置，获取相应的线程池实例来修改线程池的参数。
- 如果限制了配置中心的使用，也可以自己去扩展**ThreadPoolExecutor**，重写方法，监听线程池参数变化，来动态修改线程池参数。

线程池调优了解吗？

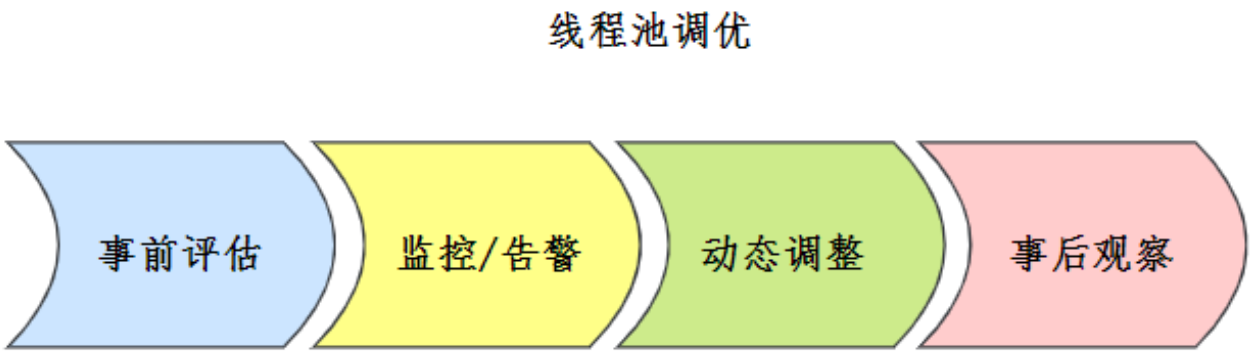
线程池配置没有固定的公式，通常事前会对线程池进行一定评估，常见的评估方案如下：

方案	问题
$N_{cpu} = number\ of\ CPUs$ $U_{cpu} = target\ CPU\ utilization, 0 \le U_{cpu} \le 1$ $\frac{W}{C} = ratio\ of\ wait\ time\ to\ compute\ time$ *The optimal pool size for keeping the processors at the desired utilization is :* $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

上线之前也要进行充分的测试，上线之后要建立完善的线程池监控机制。

事中结合监控告警机制，分析线程池的问题，或者可优化点，结合线程池动态参数配置机制来调整配置。

事后要注意仔细观察，随时调整。

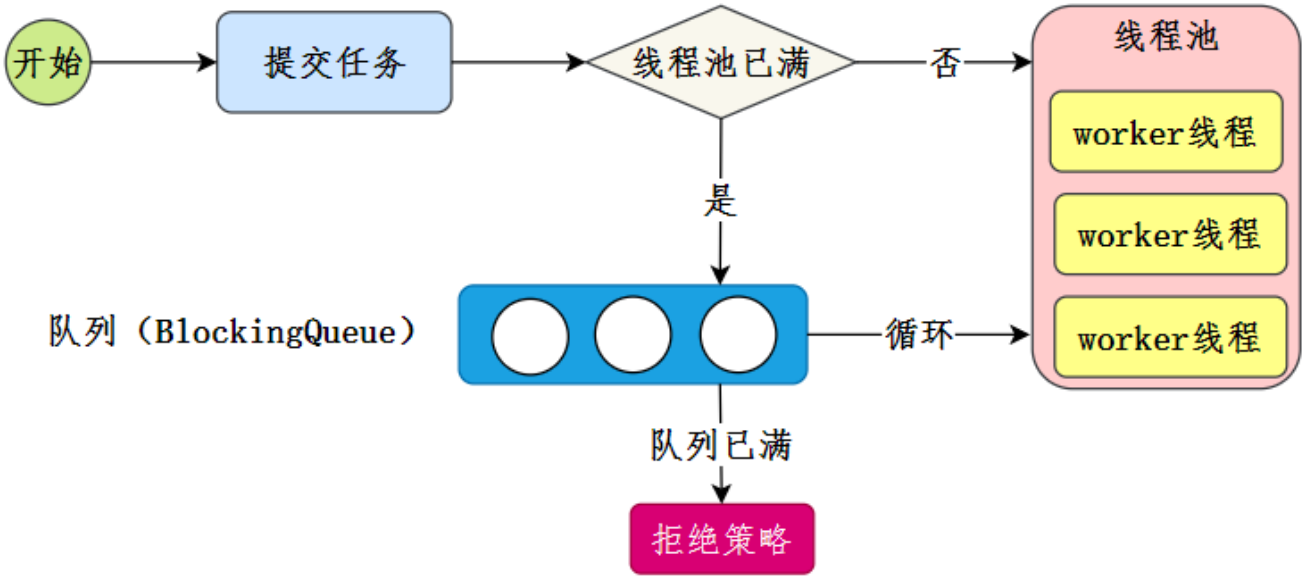


具体的调优案例可以查看参考[7]美团技术博客。

58.你能设计实现一个线程池吗?

⭐这道题在阿里的面试中出现频率比较高

线程池实现原理可以查看 要是以前有人这么讲线程池，我早就该明白了！ ，当然，我们自己实现，只需要抓住线程池的核心流程-参考[6]：



我们自己的实现就是完成这个核心流程：

- 线程池中有N个工作线程
- 把任务提交给线程池运行
- 如果线程池已满，把任务放入队列
- 最后当有空闲时，获取队列中任务来执行

实现代码[6]：

```
public class MyThreadPoolExecutor implements Executor {

    //记录线程池中线程数量
    private final AtomicInteger ctl = new AtomicInteger(0);

    //核心线程数
    private volatile int corePoolSize;
    //最大线程数
    private volatile int maximumPoolSize;

    //阻塞队列
    private final BlockingQueue<Runnable> workQueue;

    public MyThreadPoolExecutor(int corePoolSize, int maximumPoolSize, BlockingQueue<Runnable>
workQueue) {
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
    }

    /**
     * 执行
```

```
     *
     * @param command
     */
    @Override
    public void execute(Runnable command) {
        //工作线程数
        int c = ctl.get();
        //小于核心线程数
        if (c < corePoolSize) {
            //添加任务失败
            if (!addWorker(command)) {
                //执行拒绝策略
                reject();
            }
            return;
        }
        //任务队列添加任务
        if (!workQueue.offer(command)) {
            //任务队列满，尝试启动线程添加任务
            if (!addWorker(command)) {
                reject();
            }
        }
    }

    /**
     * 饱和拒绝
     */
    private void reject() {
        //直接抛出异常
        throw new RuntimeException("Can not execute!ctl.count: "
                + ctl.get() + "workQueue size: " + workQueue.size());
    }

    /**
     * 添加任务
     *
     * @param firstTask
     * @return
     */
    private boolean addWorker(Runnable firstTask) {
        if (ctl.get() >= maximumPoolSize) return false;
        Worker worker = new Worker(firstTask);
        //启动线程
        worker.thread.start();
        ctl.incrementAndGet();
        return true;
    }

    /**
     * 线程池工作线程包装类
     */
    private final class Worker implements Runnable {
        final Thread thread;
        Runnable firstTask;

        public Worker(Runnable firstTask) {
            this.thread = new Thread(this);
            this.firstTask = firstTask;
        }

        @Override
        public void run() {
            Runnable task = firstTask;
            try {
                //执行任务
                while (task != null || (task = getTask()) != null) {
                    task.run();
                    //线程池已满，跳出循环
                    if (ctl.get() > maximumPoolSize) {
                        break;
                    }
```

```
                    task = null;
                }
            } finally {
                //工作线程数增加
                ctl.decrementAndGet();
            }
        }

        /**
         * 从队列中获取任务
         *
         * @return
         */
        private Runnable getTask() {
            for (; ; ) {
                try {
                    System.out.println("workQueue size:" + workQueue.size());
                    return workQueue.take();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    //测试
    public static void main(String[] args) {
        MyThreadPoolExecutor myThreadPoolExecutor = new MyThreadPoolExecutor(2, 2,
                new ArrayBlockingQueue<Runnable>(10));
        for (int i = 0; i < 10; i++) {
            int taskNum = i;
            myThreadPoolExecutor.execute(() -> {
                try {
                    Thread.sleep(1500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("任务编号: " + taskNum);
            });
        }
    }
}
```

这样，一个实现了线程池主要流程的类就完成了。

59.单机线程池执行断电了应该怎么处理?

我们可以对正在处理和阻塞队列的任务做事务管理或者对阻塞队列中的任务持久化处理，并且当断电或者系统崩溃，操作无法继续下去的时候，可以通过回溯日志的方式来撤销`正在处理`的已经执行成功的操作。然后重新执行整个阻塞队列。

也就是说，对阻塞队列持久化；正在处理任务事务控制；断电之后正在处理任务的回滚，通过日志恢复该次操作；服务器重启后阻塞队列中的数据再加载。

并发容器和框架

关于一些并发容器，可以去看看 面渣逆袭：Java集合连环三十问 ，里面有`CopyOnWriteList`和`ConcurrentHashMap`这两种线程安全容器类的问答。。

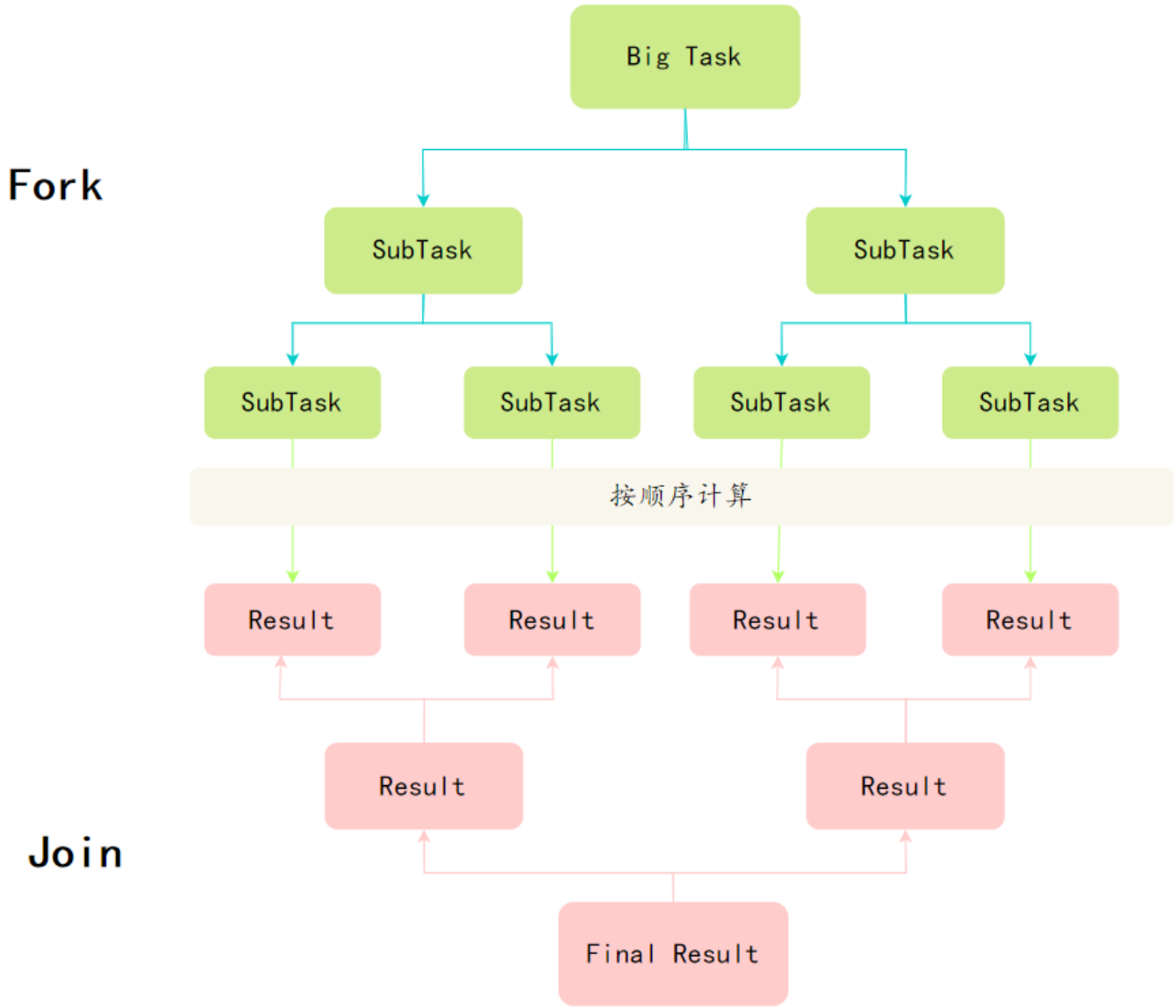
60.Fork/Join框架了解吗?

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。

要想掌握Fork/Join框架，首先需要理解两个点，**分而治之**和**工作窃取算法**。

分而治之

Fork/Join框架的定义，其实就体现了分治思想：将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。

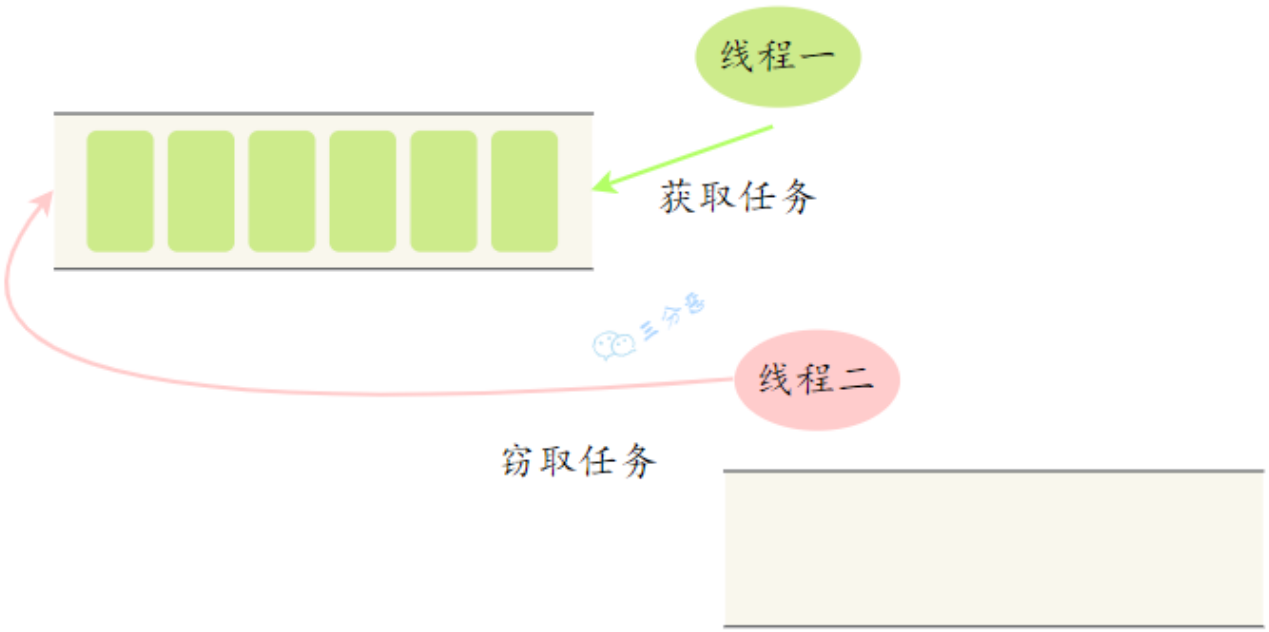


工作窃取算法

大任务拆成了若干个小任务，把这些小任务放到不同的队列里，各自创建单独线程来执行队列里的任务。

那么问题来了，有的线程干活块，有的线程干活慢。干完活的线程不能让它空下来，得让它去帮没干完活的线程干活。它去其它线程的队列里窃取一个任务来执行，这就是所谓的工作窃取。

工作窃取发生的时候，它们会访问同一个队列，为了减少窃取任务线程和被窃取任务线程之间的竞争，通常任务会使用双端队列，被窃取任务线程永远从双端队列的头部拿，而窃取任务的线程永远从双端队列的尾部拿任务执行。



看一个Fork/Join框架应用的例子，计算1~n之间的和： 1+2+3+...+n

- 设置一个分割阈值，任务大于阈值就拆分任务
- 任务有结果，所以需要继承RecursiveTask

```
public class CountTask extends RecursiveTask<Integer> {
    private static final int THRESHOLD = 16; // 阈值
    private int start;
    private int end;

    public CountTask(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
```

```
        int sum = 0;
        // 如果任务足够小就计算任务
        boolean canCompute = (end - start) <= THRESHOLD;
        if (canCompute) {
            for (int i = start; i <= end; i++) {
                sum += i;
            }
        } else {
            // 如果任务大于阈值，就分裂成两个子任务计算
            int middle = (start + end) / 2;
            CountTask leftTask = new CountTask(start, middle);
            CountTask rightTask = new CountTask(middle + 1, end);
            // 执行子任务
            leftTask.fork();
            rightTask.fork(); // 等待子任务执行完，并得到其结果
            int leftResult = leftTask.join();
            int rightResult = rightTask.join(); // 合并子任务
            sum = leftResult + rightResult;
        }
        return sum;
    }

    public static void main(String[] args) {
        ForkJoinPool forkJoinPool = new ForkJoinPool(); // 生成一个计算任务，负责计算
1+2+3+4
        CountTask task = new CountTask(1, 100); // 执行一个任务
        Future<Integer> result = forkJoinPool.submit(task);
        try {
            System.out.println(result.get());
        } catch (InterruptedException e) {
        } catch (ExecutionException e) {
        }
    }

}
```

ForkJoinTask与一般Task的主要区别在于它需要实现compute方法，在这个方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果比较大，就必须分割成两个子任务，每个子任务在调用fork方法时，又会进compute方法，看看当前子任务是否需要继续分割成子任务，如果不需要继续分割，则执行当前子任务并返回结果。使用join方法会等待子任务执行完并得到其结果。

没有什么使我停留——除了目的，纵然岸旁有玫瑰、有绿荫、有宁静的港湾，我是不系之舟。

系列内容：

- 面渣逆袭 Java SE 篇👍
- 面渣逆袭 Java 集合框架篇👍
- 面渣逆袭 Java 并发编程篇👍
- 面渣逆袭 JVM 篇👍
- 面渣逆袭 Spring 篇👍
- 面渣逆袭 Redis 篇👍
- 面渣逆袭 MyBatis 篇👍
- 面渣逆袭 MySQL 篇👍
- 面渣逆袭操作系统篇👍
- 面渣逆袭计算机网络篇👍

图文详解 60 道Java并发面试高频题，这次面试，一定吊打面试官，整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。