并发编程

楼仔 著

目录

😊 前言

大家好，我是楼仔！

为了方便大家学习，我会把所有的系列文章整理成手册，今天给大家整理的是「 Java 并发编程手册 」。

这个手册是我去年写了，当时是一边学习一遍整理，肝了我 2 个月，一共 9 篇！

1-4 章是 Java 内存模型，主要提炼程晓明的《深入理解 Java 内存模型》，5 章、6 章、9 章主要提炼 Braian Goetz 的《Java 并发编程实战》，第 7 章是整理的网络博客，第 8 章是提炼我司的多线程项目，代码可以从 Github 下载，非常有借鉴意义！

这个手册，很多知识虽然并非原创，但是整理后的内容，都是精华浓缩，无论你是面试，还是进阶，这些文章绝对不会让你失望！

📚 第 1 章：Java并发编程基础

> 主要讲解Java的并发编程的基础知识，包括原子性、可见性、有序性，以及内存模型JMM。

Java系列说明

从这篇文章开始，我就要正式开始学习Java了，之所以说是从现在开始，是因为前两个月一直在纠结是否转技术栈（细心的同学可以发现，我之前写的文章，其实和Java并没有什么关系），现在已经想清楚了，既然确定要转Java技术栈，那就踏踏实实从头开始学吧。

目前的我，可以说是Java小白，刚转团队不久，也就接触了2个月的Java，代码没写几行，既然发现自己Java很菜，那就要列个学习计划，将这块知识好好补补。目前给自己定了一年的学习计划，希望能通过一年的学习，将Java的技能从初阶直接晋级到高阶水平，可能有同学会问"我学习Java都几年的，都还是中级水平，你花一年就可以晋级到高阶？"，我只想说，我想试试，毕竟工作这么长时间，也掌握了一定的学习方法，相信跟着自己的学习节奏走，应该不会离目标太远，今天立个Flag，希望一年后不会啪啪打脸【捂脸】~~

Java系列的内容主要包括并发编程、Spring、SpringBoost、SpringCloud、Tomcat、MyBatis、Dubbo和虚拟机，然后一些经典书籍的读书笔记等，当这些都掌握到一定深度后，我想我的Java技能应该也就差不多了。

最后想说的是，Java很多系列文章，很大一部分是内容整理，之所以要通过文章的形式再写一遍，是因为看过的内容，如果自己不整理一遍，或者不让程序跑跑，很容易遗忘，所以写文章其实不是目的，主要是重新整理和回顾学习内容的过程，一方面印象深刻，另一方面，也便于自己后续查阅。

今天废话有点多，我们就从并发编程开始吧！

并发编程基本概念

原子性

> 一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

原子性是拒绝多线程操作的，不论是多核还是单核，具有原子性的量，同一时刻只能有一个线程来对它进行操作。简而言之，在整个操作过程中不会被线程调度器中断的操作，都可认为是原子性。例如 a=1是原子性操作，但是a++和a +=1就不是原子性操作。Java中的原子性操作包括：

- 基本类型的读取和赋值操作，且赋值必须是值赋给变量，变量之间的相互赋值不是原子性操作；
- 所有引用reference的赋值操作；
- java.concurrent.Atomic.* 包中所有类的一切操作。

可见性

> 指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

在多线程环境下，一个线程对共享变量的操作对其他线程是不可见的。Java提供了volatile来保证可见性，当一个变量被volatile修饰后，表示着线程本地内存无效，当一个线程修改共享变量后他会立即被更新到主内存中，其他线程读取共享变量时，会直接从主内存中读取。当然，synchronize和Lock都可以保证可见性。synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

有序性

> 即程序执行的顺序按照代码的先后顺序执行。

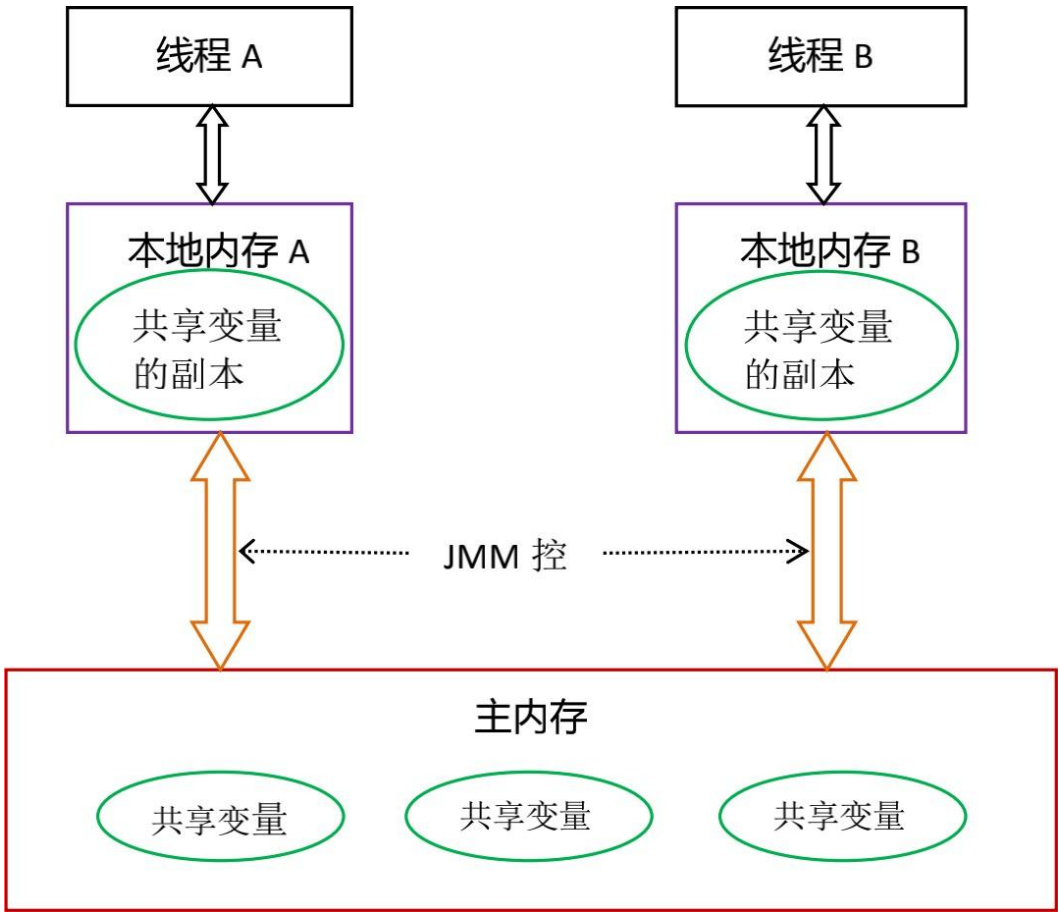
Java内存模型中的有序性可以总结为：如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句是指“线程内表现为串行语义”，后半句是指“指令重排序”现象和“工作内存主主内存同步延迟”现象。

在Java内存模型中，为了效率是允许编译器和处理器对指令进行重排序，当然重排序不会影响单线程的运行结 果，但是对多线程会有影响。Java提供volatile来保证一定的有序性。最著名的例子就是单例模式里面的DCL（双重检查锁）。另外，可以通过synchronized和Lock来保证有序性，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

> 为了让大家更好理解可见性和有序性，这个就不得不了解"内存模型"、"重排序"和"内存屏障"，因为这三个概念和他们关系非常密切。

内存模型

JMM决定一个线程对共享变量的写入何时对另一个线程可见，JMM定义了线程和主内存之间的抽象关系：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），本地内存保存了被该线程使用到的主内存的副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变 量。



对于普通的共享变量来讲，线程A将其修改为某个值发生在线程A的本地内存中，此时还未同步到主内存中去；而线程B已经缓存了该变量的旧值，所以就导致了共享变量值的不一致。解决这种共享变量在多线程模型中的不可见性问题，可以使用volatile、synchronized、final等，此时A、B的通信过程如下：

- 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去；
- 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

> JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证，需要注意

的是，JMM是个抽象的内存模型，所以所谓的本地内存，主内存都是抽象概念，并不一定就真实的对应cpu缓存和物理内存。

总结一句话，内存模型JMM控制多线程对共享变量的可见性！！！

重排序

> 重排序是指编译器和处理器为了优化程序性能而对指令序列进行排序的一种手段。

重排序需要遵守一定规则：

- 重排序操作不会对存在数据依赖关系的操作进行重排序。比如：a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。
- 重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变。 比如：a=1;b=2;c=a+b这三个操作，第一步（a=1)和第二步(b=2)由于不存在数据依赖关系， 所以可能会发生重排序，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。

重排序在单线程下一定能保证结果的正确性，但是在多线程环境下，可能发生重排序，影响结果，请看下面的示例代码：

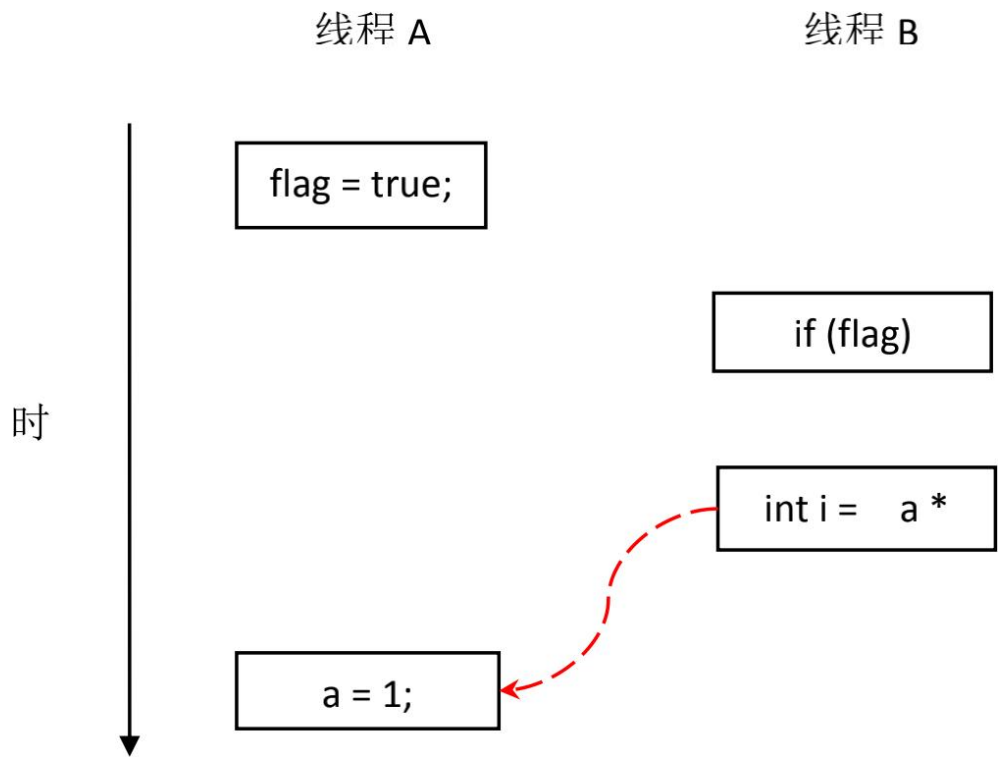
```
class ReorderExample {

   int a = 0;
   boolean flag = false; public
   void writer()
                              {
         a = 1;                          //1
         flag = true;                    //2
   }
   Public void reader()          {
         if (flag) {                          //3
               int i =   a *          a;              //4
   }
}
```

flag变量是个标记，用来标识变量a是否已被写入。这里假设有两个线程A和B，A首先执行writer()方法，随后B线程接着执行reader()方法。线程B在执行操作4时，输出是多少呢？

答案是：可能是0，也可能是1。

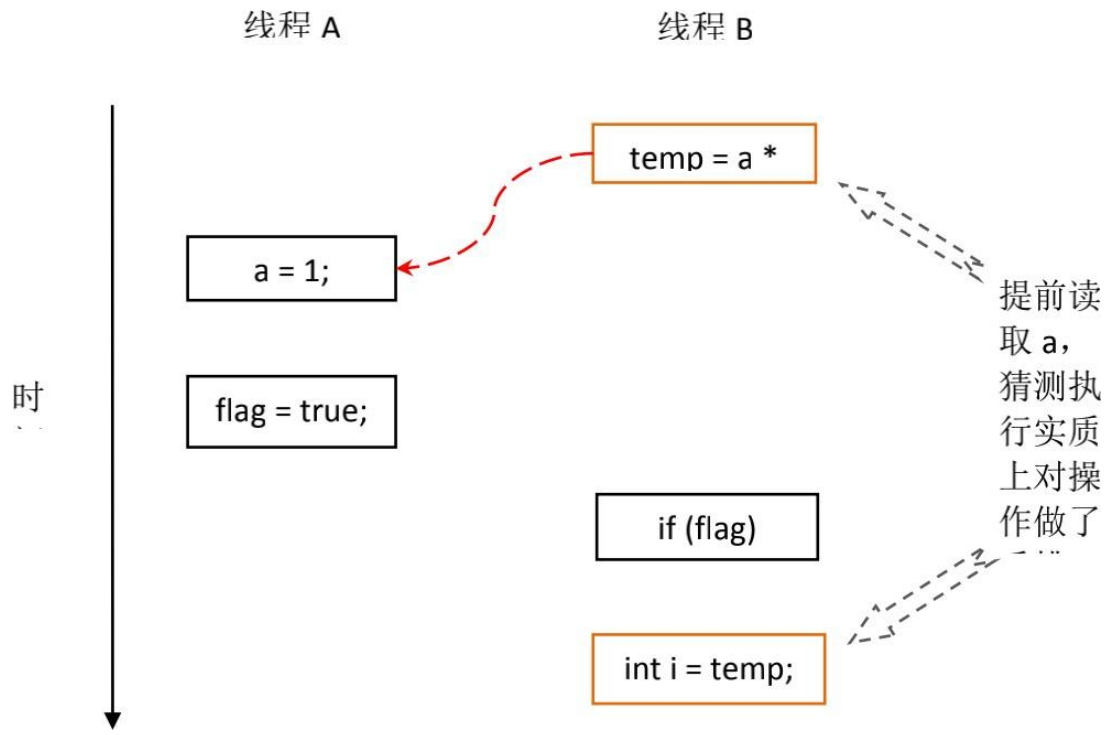
由于操作1和操作2没有数据依赖关系，编译器和处理器可以对这两个操作重排序；同样，操作3和操作4没有数据依赖关系，编译器和处理器也可以对这两个操作重排序。让我们先来看看，当操作1和操作2重排序时，可能会产生什么效果？请看下面的程序执行时序图：



如上图所示，操作1和操作2做了重排序。程序执行时，线程A首先写标记变量flag，随后线程B读这个变量。由于条件判断为真，线程B将读取变量a。此时，变量a还根本没有被线程A写入，在这里多线程程序的语义被重排序破坏了！最后输出i的结果是0。

> 温馨提示：这里其实理解起来有点绕，比如线程A先执行了writer()，然后线程B执行reader()，对于线程A，怎么会有这个重排序呢？其实这个重排序，是对线程B而言的，不是线程A哈！
>
> 有了线程B这第一视角，我们再理解一下，虽然线程A将writer()执行了，执行顺序是a=1，flag=true，但是对于线程B来说，因为重排序，线程B是根据重排序后的结果去执行的，所以才会出现上述异常情况，这么给大家解释，是不是就清晰很多呢？

下面再让我们看看，当操作3和操作4重排序时会产生什么效果（借助这个重排序，可以顺便说明控制依赖性）。下面是操作3和操作4重排序后，程序的执行时序图：



在程序中，操作3和操作4存在控制依赖关系。当代码中存在控制依赖性时，会影响指令序列执行的并行度。为此，编译器和处理器会采用猜测（Speculation）执行来克服控制相关性对并行度的影响。以处理器的猜测执行为例，执行线程B的处理器可以提前读取并计算a*a，此时结果为0，然后把计算结果临时保存到一个名为重排序缓冲（reorder buffer ROB）的硬件缓存中。当接下来操作3的条件判断为真时，就把该计算结果写入变量i中。

从图中我们可以看出，猜测执行实质上对操作3和4做了重排序。重排序在这里破坏了多线程程序的语义！因为temp的值为0，所以最后输出i的结果是0。

那如何避免重排序对多线程的影响呢，答案是“内存屏障”！

内存屏障

为了保证内存可见性，可以通过volatile、final等修饰变量，java编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。内存屏障主要有3个功能：

- 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
- 它会强制将对缓存的修改操作立即写入主存；
- 如果是写操作，它会导致其他CPU中对应的缓存行无效。

```
class ReorderExample
  { int a = 0;
  boolean volatile flag = false;
  public void writer() {
      a = 1;                    //1
      flag = true;              //2
  }
  Public void reader() {
```

假如我对上述示例的falg变量通过volatile修饰：

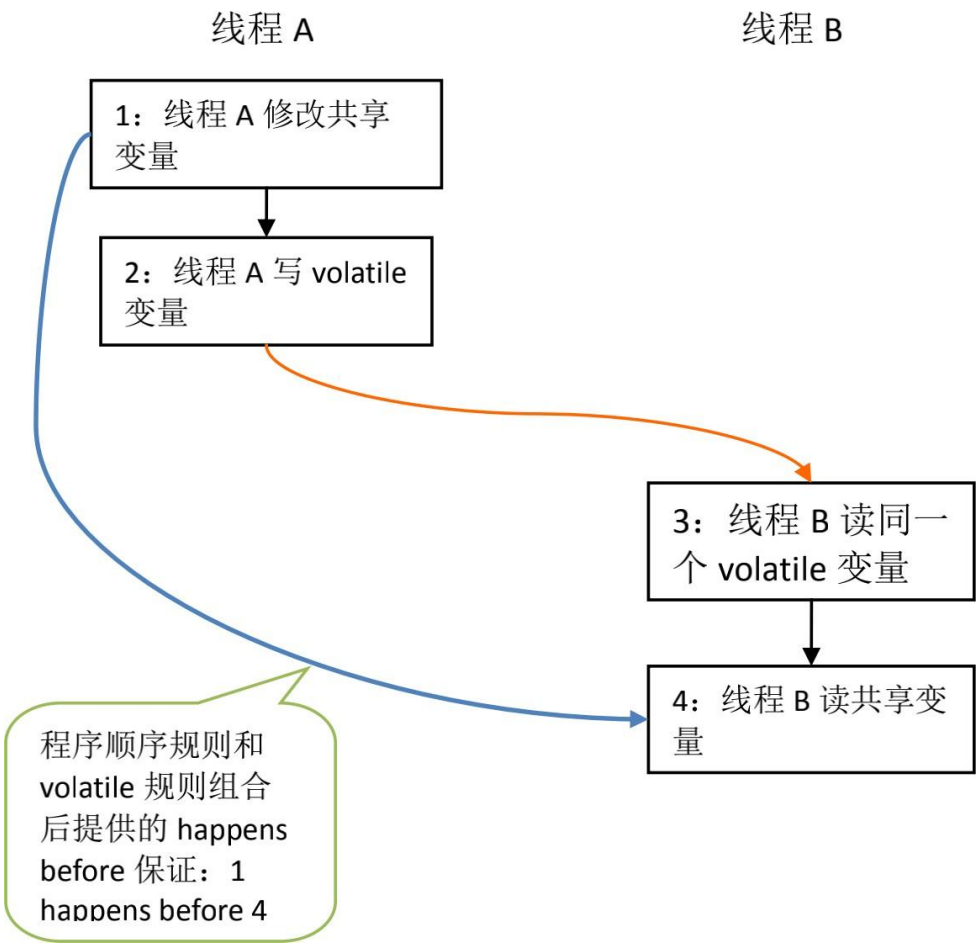
```
        if (flag) {                    //3
            int i = a * a;             //4
            System.out.println(i);
        }
    }
}
```

这个时候，volatile禁止指令重排序也有一些规则，因为篇幅原因，改规则将会在下一章讲解，根据happens before规则，这个过程建立的happens before 关系可以分为两类：

1. 根据程序次序规则，1 happens before 2; 3 happens before 4。
2. 根据volatile规则，2 happens before 3。
3. 根据happens before 的传递性规则，1 happens before 4。

> happens before规则，其实就是重排序规则建立的代码前后依赖关系。
>
> 温馨提示：这里大家可能会有疑问，1、3的规则我理解，但是对于2，为什么"2 happens before 3"，还记得前面讲的"内存模型"么？因为你对变量flag指定了volatile，所以当线程A执行完后，变量flag=true会直接刷到内存中，然后B马上可见，所以说2一定是在3前面，不可能因为重排序，导致3在2前面执行。（然后还要提示一下，这里执行时有个前提条件，就是线程A执行完，才能执行线程B里面的逻辑，因为线程A不执行完，flag一直是false，线程B根本就进不到主流程，所以你也可以直接理解为线程A执行完后，再执行线程B，才有这么个先后关系。）

上述happens before关系的图形化表现形式如下：



在上图中，每一个箭头链接的两个节点，代表了一个happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示volatile规则；蓝色箭头表示组合这些规则后提供的happens before保证。

这里A线程写一个volatile变量后，B线程读同一个volatile变量。A线程在写volatile变量之前所有可见的共享变量，在B线程读同一个volatile变量后，将立即变得对B线程可见。

总结

今天讲解了Java并发编程的3个特性，然后基于里面的两个特性“可见性”和“有序性”引出几个重要的概念，分别为“内存模型JMM”、“重排序”和“内存屏障”，这个对后续理解volatile、synchronized、final，以及避免使用的各种坑，真的是非常非常重要！！！所以这块知识要必须！一定！！要！！！掌握。

不算之前看的内容，光写这篇文章就写了一个下午。这篇文章涉及的知识，参考了大量网上的资料，我可以说，我这篇文章写的比网上绝大部分的文章要好，我看了程晓明的《深入理解Java内存模型》，里面的内容虽然很好，但是很多知识有些啰嗦，我只提取了最重要的部分，然后也有网上的文章，写的很经典，但是对于一些概念和示例的阐述，深度还不够，我结合他们的利弊，然后整理了这篇文章，详细大家看完这篇文章后，再看其它的文章，应该就感觉好理解很多。

后记

这篇文章是我对Java并发编程的入门文章，后面会继续分别写volatile、synchronized、final，相关内容已经看完，后续直接整理输出即可。其实我也不知道这些基础知识学到哪个程度才算OK，那就边写边学，等基础知识写的差不多了，就开始写实战部分。

参考文章：
《深入理解Java内存模型》
《Java并发编程实战》

📚 第 2 章：volatile

> 主要讲解volatile的相关知识，以及容易遇到的坑。

volatile变量的特性

保证可见性，不保证原子性：

- 当写一个volatile变量时，JMM会把该线程本地内存中的变量强制刷新到主内存中去；
- 这个写会操作会导致其他线程中的volatile变量缓存无效。

禁止指令重排，我们回顾一下，重排序需要遵守一定规则：

- 重排序操作不会对存在数据依赖关系的操作进行重排序。比如：a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。
- 重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变。 比如：a=1;b=2;c=a+b这三个操作，第一步（a=1)和第二步(b=2)由于不存在数据依赖关系， 所以可能会发生重排序，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。

volatile禁止指令重排规则

使用volatile关键字修饰共享变量便可以禁止这种重排序。若用volatile修饰共享变量，在编译时，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，volatile禁止指令重排序也有一些规则：

- 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
- 在进行指令优化时，不能将对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

> 即执行到volatile变量时，其前面的所有语句都执行完，后面所有语句都未执行。且前面语句的结果对volatile变量及其后面语句可见。

volatile禁止指令重排分析

> 该部分相关内容，我直接copy上一篇文章，不是为了凑篇幅，因为有同学没有看上一篇文章，直接看这篇，为了能让每一篇文章能独立成章，可能会引用之前文章中的内容。

先看下面未使用volatile的代码：

```
class ReorderExample {

   int a = 0;
   boolean flag = false; public
   void writer()
                              {
         a = 1;                              //1
         flag = true;                        //2
   }
   Public void reader()          {
         if (flag) {                               //3
               int i =   a *         a;            //4
   }
}
```

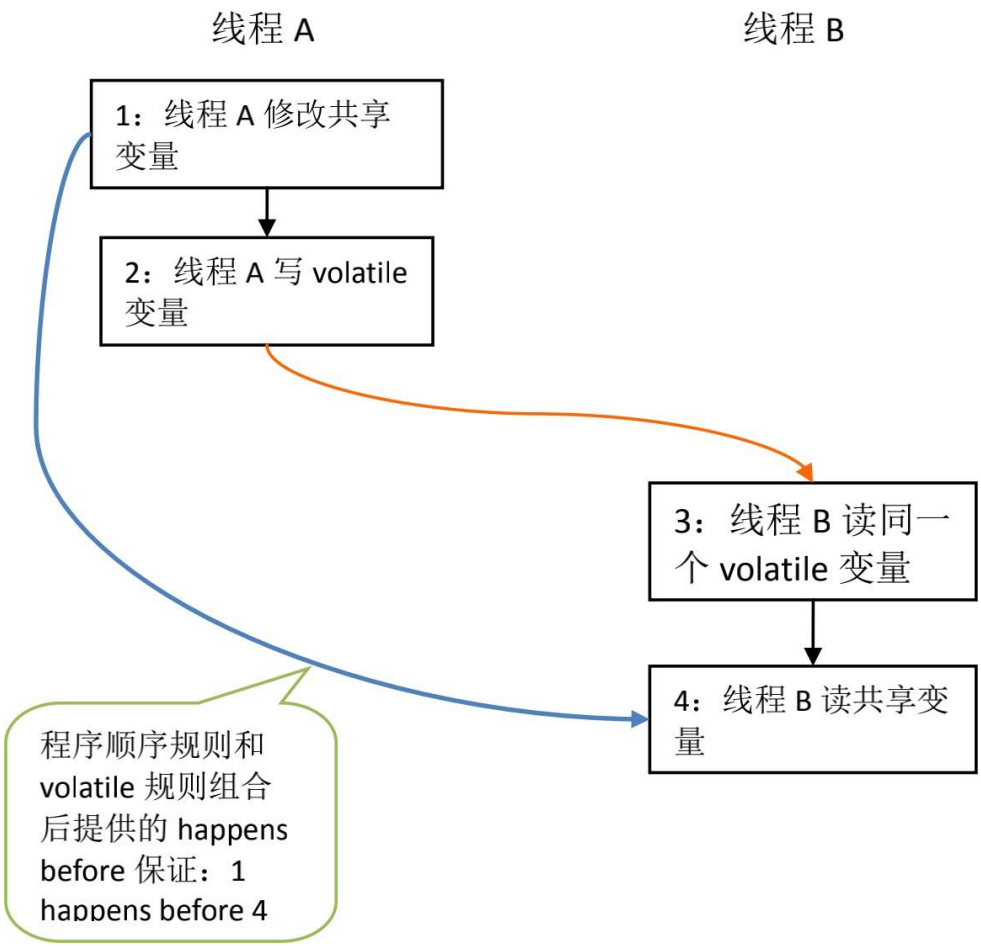
因为重排序影响，所以最终的输出可能是0，，具体分析请参考我的上一篇文章《Java并发编程系列1-基础知识》，如果引入volatile，我们再看一下代码：

```
class ReorderExample
  { int a = 0;
  boolean volatile flag = false;
  public void writer() {
      a = 1;                    //1
      flag = true;              //2
  }
  Public void reader() {
      if (flag) {                //3
          int i =  a * a;        //4
          System.out.println(i);
      }
  }
}
```

这个时候，volatile禁止指令重排序也有一些规则，这个过程建立的happens before关系可以分为两类：

1. 根据程序次序规则，1 happens before 2; 3 happens before 4。
2. 根据volatile规则，2 happens before 3。
3. 根据happens before 的传递性规则，1 happens before 4。

上述happens before关系的图形化表现形式如下：



在上图中，每一个箭头链接的两个节点，代表了一个happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示volatile规则；蓝色箭头表示组合这些规则后提供的happens before保证。

这里A线程写一个volatile变量后，B线程读同一个volatile变量。A线程在写volatile变量之前所有可见的共享变量，在B线程读同一个volatile变量后，将立即变得对B线程可见。

volatile不适用场景

volatile不适合复合操作

下面是变量自加的示例：

```
public class volatileTest
    { public volatile int inc =
    0; public void increase() {
        inc++;
    }
    public static void main(String[] args) {
        final volatileTest test = new volatileTest();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
```

```
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1) //保证前面的线程都执行完
            Thread.yield();
        System.out.println("inc output:" + test.inc);
    }
}
```

测试输出：

```
inc output:8182
```

因为inc++不是一个原子性操作，可以由读取、加、赋值3步组成，所以结果并不能达到10000。

解决方法

采用synchronized：

```
public class volatileTest1
    { public int inc = 0;
    public synchronized void increase()
        { inc++;
    }
    public static void main(String[] args) {
        final volatileTest1 test = new volatileTest1();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1) //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add synchronized, inc output:" + test.inc);
    }
}
```

采用Lock：

```
public class volatileTest2
    { public int inc = 0;
    Lock lock = new ReentrantLock();
    public void increase() {
        lock.lock();
```

```
        inc++;
        lock.unlock();
    }
    public static void main(String[] args) {
        final volatileTest2 test = new volatileTest2();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1) //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add lock, inc output:" + test.inc);
    }
}
```

采用AtomicInteger：

```
public class volatileTest3 {
    public AtomicInteger inc = new AtomicInteger();
    public void increase() {
        inc.getAndIncrement();
    }
    public static void main(String[] args) {
        final volatileTest3 test = new volatileTest3();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run()
                    { for(int
                    j=0;j<100;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1) //保证前面的线程都执行完
            Thread.yield();
        System.out.println("add AtomicInteger, inc output:" + test.inc);
    }
}
```

三者输出都是1000，如下：

```
add synchronized, inc output:1000
add lock, inc output:1000
add AtomicInteger, inc output:1000
```

单例模式的双重锁为什么要加volatile

先看一下单例代码：

```
public class penguin {
    private static volatile penguin m_penguin = null;
    // 避免通过new初始化对象
    private void penguin() {}
    public void beating() {
        System.out.println("打豆豆");
    };
    public static penguin getInstance() {       //1
        if (null == m_penguin) {                //2
            synchronized(penguin.class) {       //3
                if (null == m_penguin) {        //4
                    m_penguin = new penguin(); //5
                }
            }
        }
        return m_penguin;                       //6
    }
}
```

在并发情况下，如果没有volatile关键字，在第5行会出现问题。instance = new TestInstance();可以分解为3行伪代码：

```
a. memory = allocate() //分配内存
b. ctorInstanc(memory) //初始化对象
c. instance = memory   //设置instance指向刚分配的地
```

上面的代码在编译运行时，可能会出现重排序从a-b-c排序为a-c-b。在多线程的情况下会出现以下问题。当线程A在执行第5行代码时，B线程进来执行到第2行代码。假设此时A执行的过程中发生了指令重排序，即先执行了a和c，没有执行b。那么由于A线程执行了c导致instance指向了一段地址，所以B线程判断instance不为null，会直接跳到第6行并返回一个未初始化的对象。

何时使用volatile

大家对volatile变量的学习，关于重排序的规则，这个仅做了解即可，更重要的是掌握它的正常使用场景。那对于锁和volatile，我们什么时候才会去使用volatile呢？我们先回顾一下volatile和锁的区别：

> 加锁机制既可以保证可见性，又可以保证原子性，而volatile变量只能保证可见性。

也就是我们只在可见性上，才去使用volatile变量，比如在多线程情况下，我们需要对某个操作的完成、发生中断或者状态的标志，就可以声明为volatile，因为volatile可以保证该变量在所有线程的可见性。但是如果对于稍微复杂的操作，比如i++等复合操作，就不要使用volatile变量，如果你想通过volatile的“禁止指令重排规则”来保证volatile变量的前后变量代码的顺序性，建议你不要这样做，一方面有很多坑，另一方别人也不理解你的代码，比

如我示例中的“volatile禁止指令重排分析”，即使我讲了一遍，我自己都很容易忘，你还指望别人能理解么？如果你偏要这样去“炫技”，那你就是给别人埋坑了！就好比一行很简单的代码，有位程序员偏要用“&|”操作，结果埋个大

坑，给部门造成S级的线上事故。

下面是《Java并发编程实战》的描述，我摘抄一下，其实有第一条不理解，仅做记录：

> 当且仅当满足以下所有条件时，才应该使用volatile变量：
>
> - 对变量的写操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
> - 该变量不会与其它状态变量一起纳入不可变的条件中。
> - 在访问变量时不需要加锁。

总结

volatile可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在JVM底层volatile是采用"内存屏障"来实现的。观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令，lock前缀指令实际上相当于一个内存屏障（也称内存栅栏），内存屏障会提供3个功能：

- 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
- 它会强制将对缓存的修改操作立即写入主存；
- 如果是写操作，它会导致其他CPU中对应的缓存行无效。

最后也讲解了volatile不适用的场景，以及解决的方法，并解释了单例模式为何需要使用volatile。

📚 第 3 章：synchronized

> 主要讲解synchronized的应用方式和内存语义。

前言

看这篇文章前，建议大家先看我前面的文章《Java并发编程系列1-基础知识》，否则里面的相关知识看不懂，特别是并发编程相关的可见性、有序性，以及内存模型JMM等。

在Java中，关键字synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到synchronized另外一个重要的作用，synchronized可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到（保证可见性，完全可以替代Volatile功能）。

synchronized的三种应用方式

synchronized关键字最主要有以下3种应用方式，下面分别介绍：

- 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁；
- 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁；
- 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。

synchronized作用于实例方法

所谓的实例对象锁就是用synchronized修饰实例对象中的实例方法，注意是实例方法不包括静态方法，如下：

```
public class AccountingSync implements Runnable {
    //共享资源(临界资源)
    static int i = 0;
    // synchronized 修饰实例方法
    public synchronized void increase()
        { i ++;
    }
    @Override
    public void run() {
        for(int
            j=0;j<1000000;j++){ incr
            ease();
        }
    }
    public static void main(String args[]) throws InterruptedException
        { AccountingSync instance = new AccountingSync();
        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("static, i output:" + i);
    }
}
/**
 * 输出结果:
 * static, i output:2000000
 */
```

如果在函数increase()前不加synchronized，因为i++不具备原子性，所以最终结果会小于2000000，具体分析可以参考文章《Java并发编程系列2-volatile》。下面这点非常重要：

> 一个对象只有一把锁，当一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，所以无法访问该对象的其他synchronized实例方法，但是其他线程还是可以访问该实例对象的其他非synchronized方法。

但是一个线程 A 需要访问实例对象 obj1 的 synchronized 方法 f1(当前对象锁是obj1)，另一个线程 B 需要访问实例对象 obj2 的 synchronized 方法 f2(当前对象锁是obj2)，这样是允许的：

```
public class AccountingSyncBad implements Runnable {
    //共享资源(临界资源)
    static int i = 0;
    // synchronized 修饰实例方法
    public synchronized void increase()
        { i ++;
    }
    @Override
    public void run() {
        for(int
            j=0;j<1000000;j++){ incr
            ease();
        }
    }
    public static void main(String args[]) throws InterruptedException {
        // new 两  AccountingSync新实例
        Thread t1 = new Thread(new AccountingSyncBad());
        Thread t2 = new Thread(new AccountingSyncBad());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("static, i output:" + i);
    }
}
/**
 * 输出结果:
 * static, i output:1224617
 */
```

上述代码与前面不同的是我们同时创建了两个新实例AccountingSyncBad，然后启动两个不同的线程对共享变量i进行操作，但很遗憾操作结果是1224617而不是期望结果2000000，因为上述代码犯了严重的错误，虽然我们使用 synchronized修饰了increase方法，但却new了两个不同的实例对象，这也就意味着存在着两个不同的实例对象 锁，因此t1和t2都会进入各自的对象锁，也就是说t1和t2线程使用的是不同的锁，因此线程安全是无法保证的。

> 每个对象都有一个对象锁，不同的对象，他们的锁不会互相影响。

解决这种困境的的方式是将synchronized作用于静态的increase方法，这样的话，对象锁就当前类对象，由于无论创建多少个实例对象，但对于的类对象拥有只有一个，所有在这样的情况下对象锁就是唯一的。下面我们看看如何使用将synchronized作用于静态的increase方法。

synchronized作用于静态方法

> 当synchronized作用于静态方法时，其锁就是当前类的class锁，不属于某个对象。

> 当前类class锁被获取，不影响对象锁的获取，两者互不影响。

由于静态成员不专属于任何一个实例对象，是类成员，因此通过class对象锁可以控制静态成员的并发操作。需要注意的是如果一个线程A调用一个实例对象的非static 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，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class对象，而访问非静态synchronized方法占用的锁是当前实例对象锁，看如下代码：

```
public class AccountingSyncClass implements Runnable { static int i = 0;
    /**
     * 作用于静态方法,锁是当前class对象,也就是
     * AccountingSyncClass类对应的class对象
     */
    public static synchronized void increase() { i++;
    }
    // 非静态,访问时锁不一样不会发生互斥
    public synchronized void increase4Obj() { i++;
    }
    @Override
    public void run() {
        for(int
            j=0;j<1000000;j++){ increase()
            ;
        }
    }
    public static void main(String[] args) throws InterruptedException {
        //new新实例
        Thread t1=new Thread(new AccountingSyncClass());
        //new新实例
        Thread t2=new Thread(new AccountingSyncClass());
        //启动线程 t1.start();t2.start();
        t1.join();t2.join(); System.out.println(i);
    }
}
/**
 * 输出结果:
 * 2000000
 */
```

由于synchronized关键字修饰的是静态increase方法，与修饰实例方法不同的是，其锁对象是当前类的class对

象。注意代码中的increase4Obj方法是实例方法，其对象锁是当前实例对象，如果别的线程调用该方法，将不会产生互斥现象，毕竟锁对象不同，但我们应该意识到这种情况下可能会发现线程安全问题(操作了共享静态变量i)。

synchronized同步代码块

在某些情况下，我们编写的方法体可能比较大，同时存在一些比较耗时的操作，而需要同步的代码又只有一小部分，如果直接对整个方法进行同步操作，可能会得不偿失，此时我们可以使用同步代码块的方式对需要同步的代码进行包裹，这样就无需对整个方法进行同步操作了，同步代码块的使用示例如下：

```
public class AccountingSync2 implements Runnable {
    static AccountingSync2 instance = new AccountingSync2(); // 饿汉单例模式
    static int i=0;
    @Override
    public void run() {
        //省略其他耗时操作....
        //使用同步代码块对变量i进行同步操作,锁对象为instance
        synchronized(instance){
            for(int
                j=0;j<1000000;j++){ i++;
            }
        }
    }
    public static void main(String[] args) throws InterruptedException
        { Thread t1=new Thread(instance);
        Thread t2=new Thread(instance);
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
/**
 * 输出结果:
 * 2000000
 */
```

从代码看出，将synchronized作用于一个给定的实例对象instance，即当前实例对象就是锁对象，每次当线程进入 synchronized包裹的代码块时就会要求当前线程持有instance实例对象锁，如果当前有其他线程正持有该对象锁，那么新到的线程就必须等待，这样也就保证了每次只有一个线程执行i++;操作。当然除了instance作为对象外，我们还可以使用this对象(代表当前实例)或者当前类的class对象作为锁，如下代码：

```
//this,当前实例对象锁
synchronized(this){
    for(int
        j=0;j<1000000;j++){ i++;
    }
}
//class对象锁
synchronized(AccountingSync.class){
    for(int
        j=0;j<1000000;j++){ i++;
    }
}
```

synchronized禁止指令重排分析

> 指令重排的情况，可以参考文章《Java并发编程系列1-基础知识》

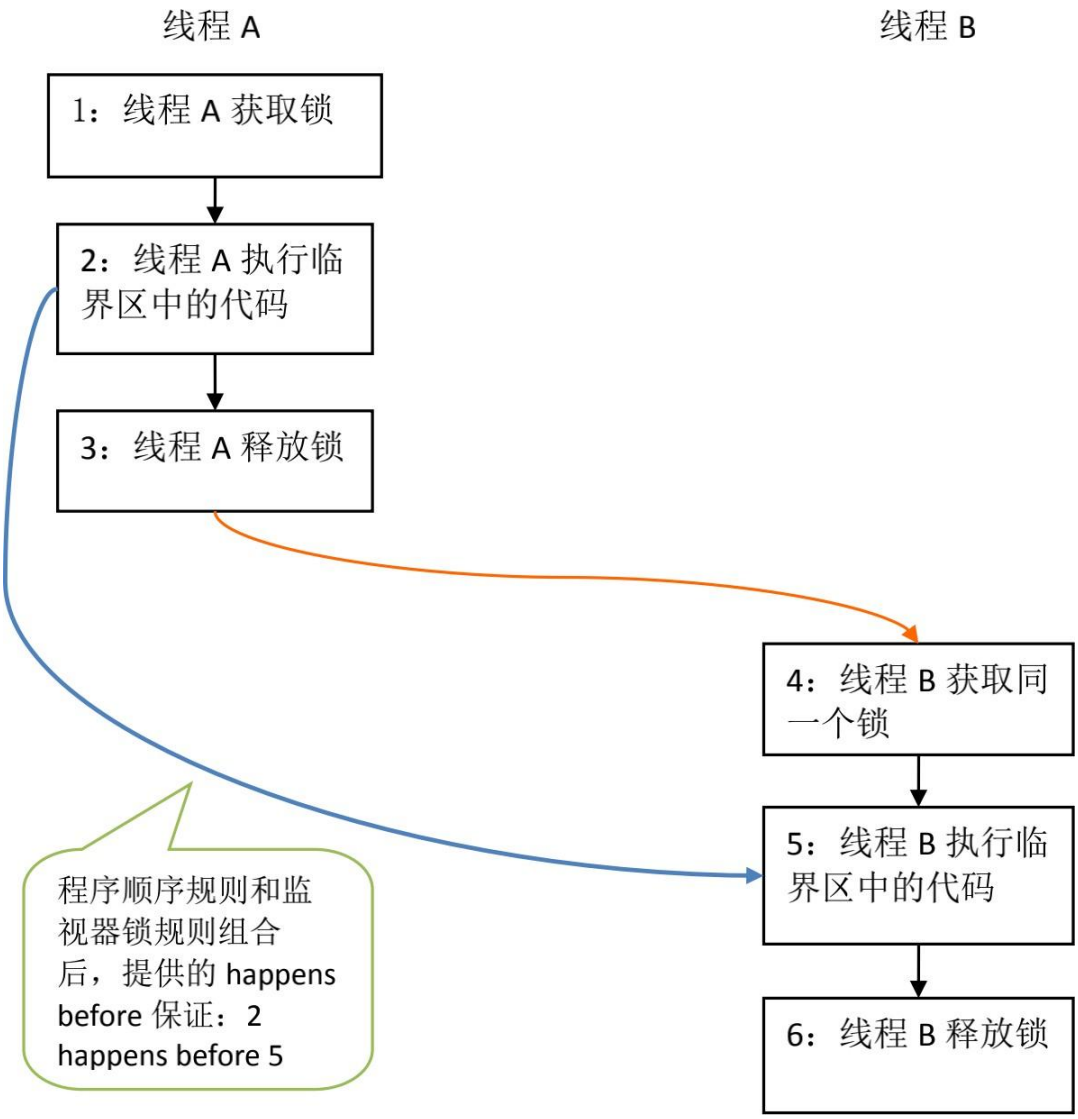
```
class MonitorExample {
    int a = 0;
    public synchronized void writer()     //1
        { a++;                            //2
    }                                     //3
    public synchronized void reader()     //4
        { int i = a;                      //5
        //……
    }                                     //6
}
```

我们先看如下代码：

假设线程A执行writer()方法，随后线程B执行reader()方法。根据happens before规则，这个过程包含的happens before关系可以分为两类：

- 根据程序次序规则，1 happens before 2, 2 happens before 3; 4 happens before 5, 5 happens before 6。
- 根据监视器锁规则，3 happens before 4。
- 根据happens before的传递性，2 happens before 5。

上述happens before 关系的图形化表现形式如下：



> 在上图中，每一个箭头链接的两个节点，代表了一个happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示监视器锁规则；蓝色箭头表示组合这些规则后提供的happens before保证。

上图表示在线程A释放了锁之后，随后线程B获取同一个锁。在上图中，2 happens before 5。因此，线程A在释放锁之前所有可见的共享变量，在线程B获取同一个锁之后，将立刻变得对B线程可见。

synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功。

synchronized就是可重入锁，因此一个线程调用synchronized方法的同时，在其方法体内部调用该对象另一个synchronized方法是允许的，如下：

```
public class AccountingSync implements Runnable{
    static AccountingSync instance=new AccountingSync();
    static int i=0;
    static int j=0;
    @Override
    public void run() {
```

```
            for(int j=0;j<1000000;j++){
                //this,当前实例对象锁
                synchronized(this){
                    i++;
                    increase();//synchronized的可重入性
                }
            }
        }
        public synchronized void
            increase(){ j++;
        }
        public static void main(String[] args) throws InterruptedException
            { Thread t1=new Thread(instance);
            Thread t2=new Thread(instance);
            t1.start();t2.start();
            t1.join();t2.join();
            System.out.println(i);
        }
    }
```

当前实例对象锁后进入synchronized代码块执行同步代码，并在代码块中调用了当前实例对象的另外一个synchronized方法，再次请求当前实例锁时，将被允许。需要特别注意另外一种情况，当子类继承父类时，子类也是可以通过可重入锁调用父类的同步方法。注意由于synchronized是基于monitor实现的，因此每次重入，monitor中的计数器仍会加1。

总结

该文章给大家讲解了synchronized的三种应用方式，指令重排情况分析，以及synchronized的可重入性，通过该文章，基本可以掌握synchronized的使用姿势，以及可能会遇到的坑。关于"线程中断与synchronized"的相关知识，因为篇幅原因就不写了，大家可以到网上查一下相关资料，进一步学习。

参考资料：

《深入理解Java内存模型》

《Java并发编程实战》

📚 第 4 章：final

> 主要讲解final的内存语义和使用方式。

前言

看这篇文章前，建议先看完《Java并发编程系列1-基础知识》，因为相关知识有很强的依赖，这篇文章也是Java内存模型JMM相关文章的最后一篇。

final禁止指令重排分析

> 该部分内容基本摘抄自《深入理解Java内存模型》，仅加入自己的总结，更详细讲解可以直接参考此书。

对final域的读和写更像是普通的变量访问，编译器和处理器要遵守两个重排序规则：

- 在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。
- 初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

```
public class FinalExample {
    int i;                              //普通变量
    final int j;                        //final变量
    static FinalExample obj;

    public void FinalExample () {       //构造函数
        i = 1;                          //写普通域
        j = 2;                          //写final域
    }
    public static void writer () {      //写线程A执行
        obj = new FinalExample ();
    }
    public static void reader () {          //读线程B执行
        FinalExample object = obj;          //读对象引用
        int a = object.i;                   //读普通域
        int b = object.j;                   //读final域
    }
}
```

下面，我们通过一些示例性的代码来分别说明这两个规则：

> 这里假设一个线程A执行writer()方法，随后另一个线程B执行reader()方法，注意两者的调用先后关系！

下面我们通过这两个线程的交互来说明这两个规则。

写final域的重排序规则

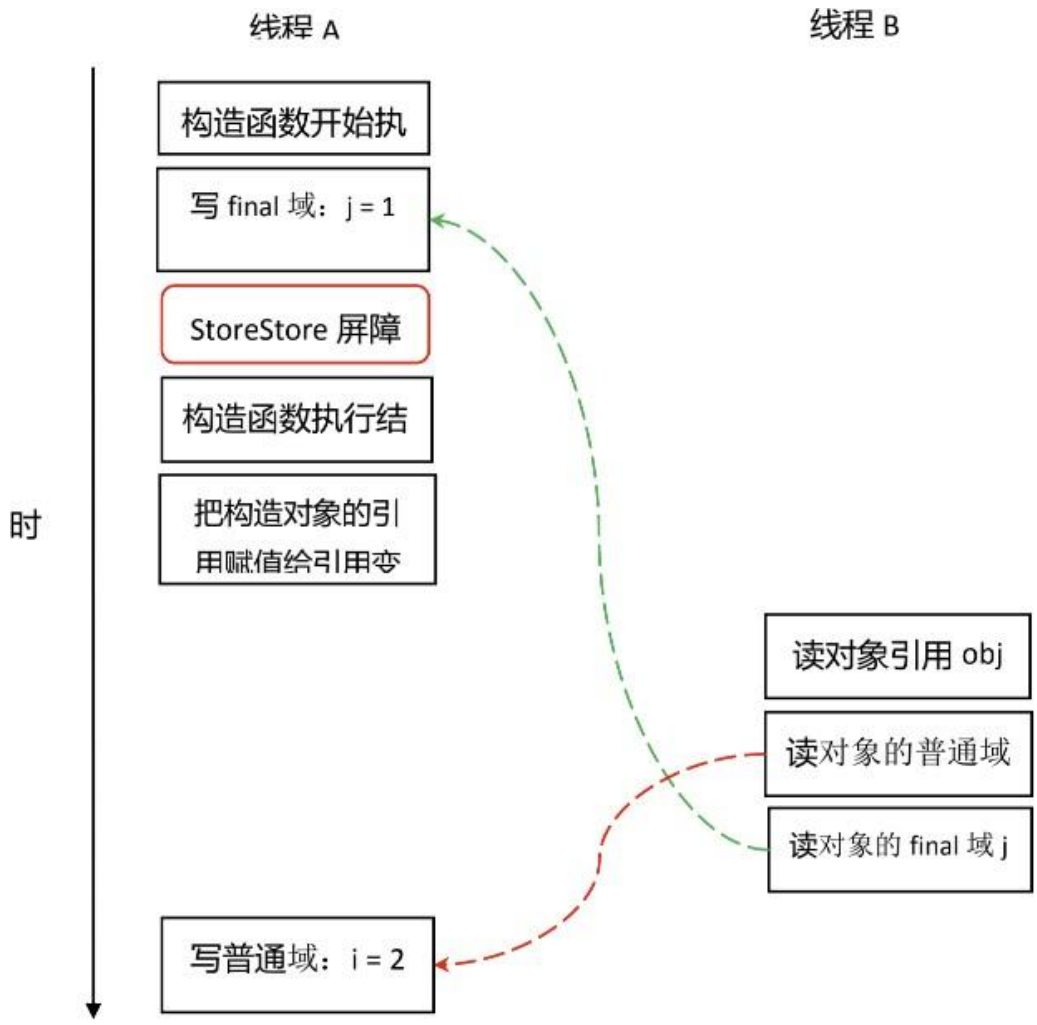
写final域的重排序规则禁止把final域的写重排序到构造函数之外。这个规则的实现包含下面2个方面：

- JMM禁止编译器把final域的写重排序到构造函数之外。
- 编译器会在final域的写之后，构造函数return之前，插入一个StoreStore屏障。这个屏障禁止处理器把final域的写重排序到构造函数之外。

现在让我们分析writer ()方法。writer ()方法只包含一行代码：finalExample = new FinalExample ()。这行代码包含两个步骤：

- 构造一个FinalExample类型的对象；
- 把这个对象的引用赋值给引用变量obj。

假设线程B读对象引用与读对象的成员域之间没有重排序（马上会说明为什么需要这个假设），下图是一种可能的执行时序：



在上图中，写普通域的操作被编译器重排序到了构造函数之外，读线程B错误的读取了普通变量i初始化之前的值。而写final域的操作，被写final域的重排序规则"限定"在了构造函数之内，读线程B正确的读取了final变量初始化之后的值。

写final域的重排序规则可以确保：在对象引用为任意线程可见之前，对象的final域已经被正确初始化过了，而普通域不具有这个保障。以上图为例，在读线程B"看到"对象引用obj时，很可能obj对象还没有构造完成（对普通域i的写操作被重排序到构造函数外，此时初始值2还没有写入普通域i）。

> 总结一下：也就是对象初始化final变量和普通变量，然后将初始化的对象引用赋值给其它变量前，final变量可以保证已经被初始化，但是普通变量不能保证，可能会导致读取的普通变量是一个空值，或者说是未初始化的值，导致异常。

读final域的重排序规则

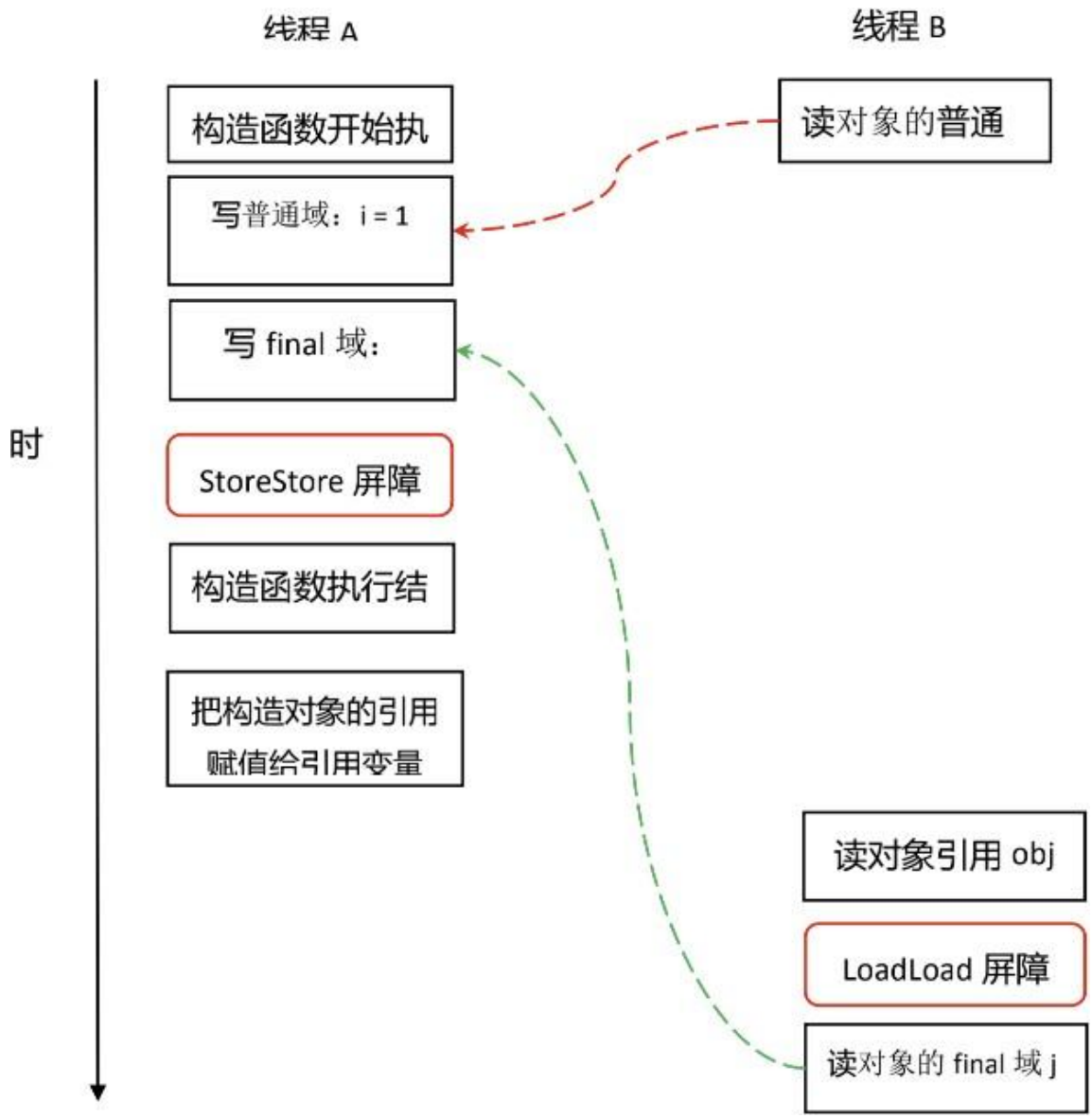
读final域的重排序规则如下：

- 在一个线程中，初次读对象引用与初次读该对象包含的final域，JMM禁止处理器重排序这两个操作（注意，这个规则仅仅针对处理器）。编译器会在读final域操作的前面插入一个LoadLoad屏障。

reader()方法包含三个操作：

1. 初次读引用变量obj;
2. 初次读引用变量obj指向对象的普通域j。
3. 初次读引用变量obj指向对象的final域i。

现在我们假设写线程A没有发生任何重排序，同时程序在不遵守间接依赖的处理器上执行，下面是一种可能的执行时序：



在上图中，读对象的普通域的操作被处理器重排序到读对象引用之前。读普通域时，该域还没有被写线程A写入，这是一个错误的读取操作。而读final域的重排序规则会把读对象final域的操作“限定”在读对象引用之后，此时该final域已经被A线程初始化过了，这是一个正确的读取操作。

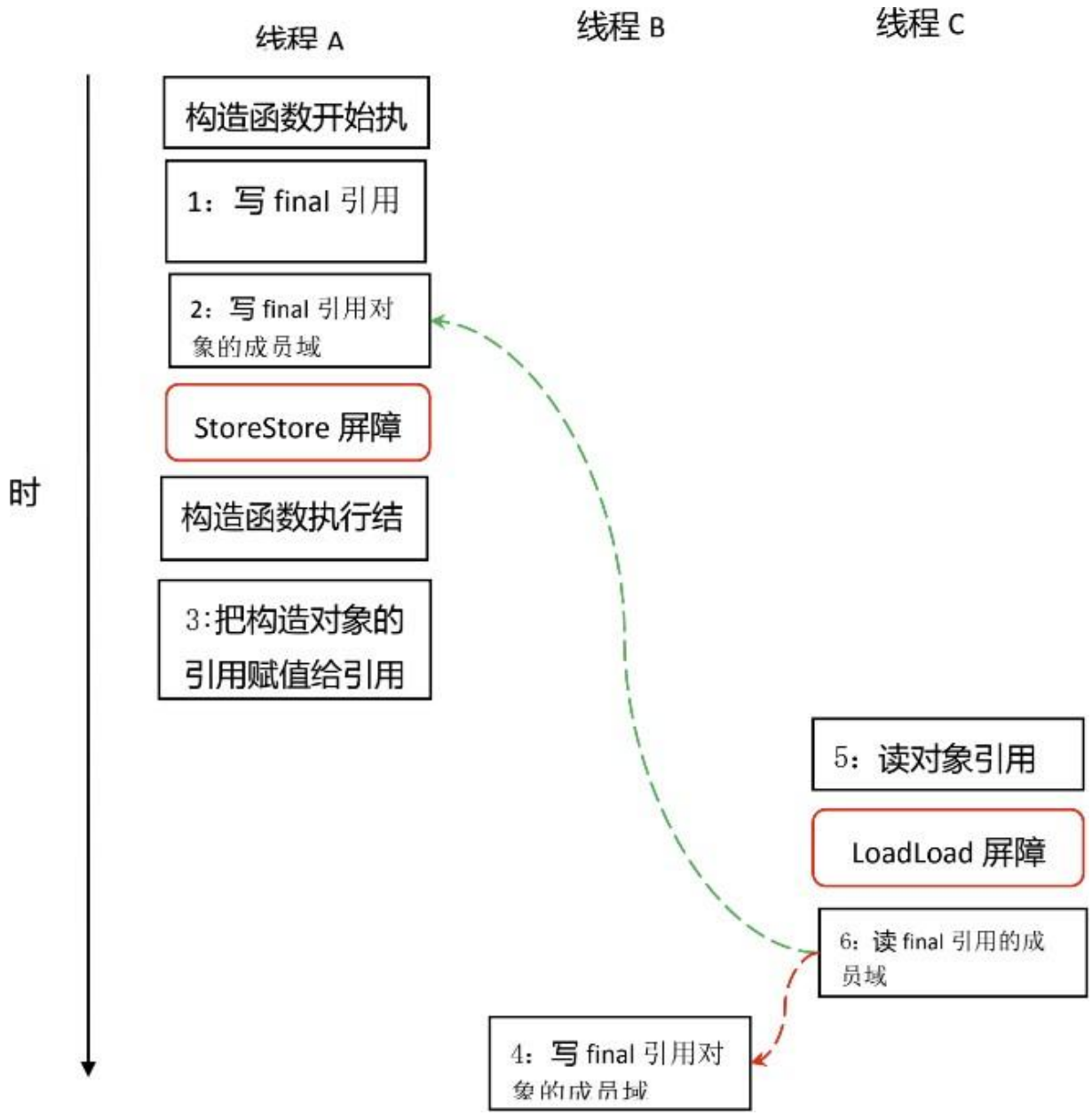
> 总结一下：在读一个对象的final变量之前，一定会先读包含这个final域的对象的引用，所以不用担心读到对象的final变量，会因为重排除导致读到的是一个未初始化的值，但是对象的普通变量就不能这样保证。

> 对读和写finlal域，整体总结一下：写final域的重排序规则会要求译编器在final域的写之后，构造函数return之前，插入一个StoreStore障屏。读final域的重排序规则要求编译器在读final域的操作前面插入一个 LoadLoad屏障。

如果final域是引用类型

上面我们看到的final域是基础数据类型，下面让我们看看如果final域是引用类型，将会有什么效果？请看下列示例代码：

```
public class FinalReferenceExample {
  final int[] intArray;                       //final是引用类型
  static FinalReferenceExample obj;
  public FinalReferenceExample () {           //构造函数
      intArray = new int[1];                  //1
      intArray[0] = 1;                        //2
  }
  public static void writerOne () {           //写线程A执行
      obj = new FinalReferenceExample ();     //3
  }
  public static void writerTwo ()             //写线程B执行
      { obj.intArray[0] = 2;                  //4
  }
  public static void reader ()                //读线程C执行
      { if (obj != null) {                    //5
          int temp1 = obj.intArray[0];        //6
      }
  }
}
```



在上图中，1是对final域的写入，2是对这个final域引用的对象的成员域的写入，3是把被构造的对象的引用赋值给某个引用变量。这里除了前面提到的1不能和3重排序外，2和3也不能重排序。

JMM可以确保读线程C至少能看到写线程A在构造函数中对final引用对象的成员域的写入。即C至少能看到数组下标0的值为1。而写线程B对数组元素的写入，读线程C可能看的到，也可能看不到。JMM不保证线程B的写入对读线程C可见，因为写线程B和读线程C之间存在数据竞争，此时的执行结果不可预知。

如果想要确保读线程C看到写线程B对数组元素的写入，写线程B和读线程C之间需要使用同步原语（lock或volatile）来确保内存可见性。

> 总结一下：如果final域为引用类型，这个其实和非引用类型禁止重排序的规则基本一样。上面的示例，writerTwo()和reader()同时对一个数据进行操作，存在竞争关系，也很好理解，我换一个非引用类型，也一样存在并发，解决方案就是加锁。

为什么final引用不能从构造函数内“逸出”

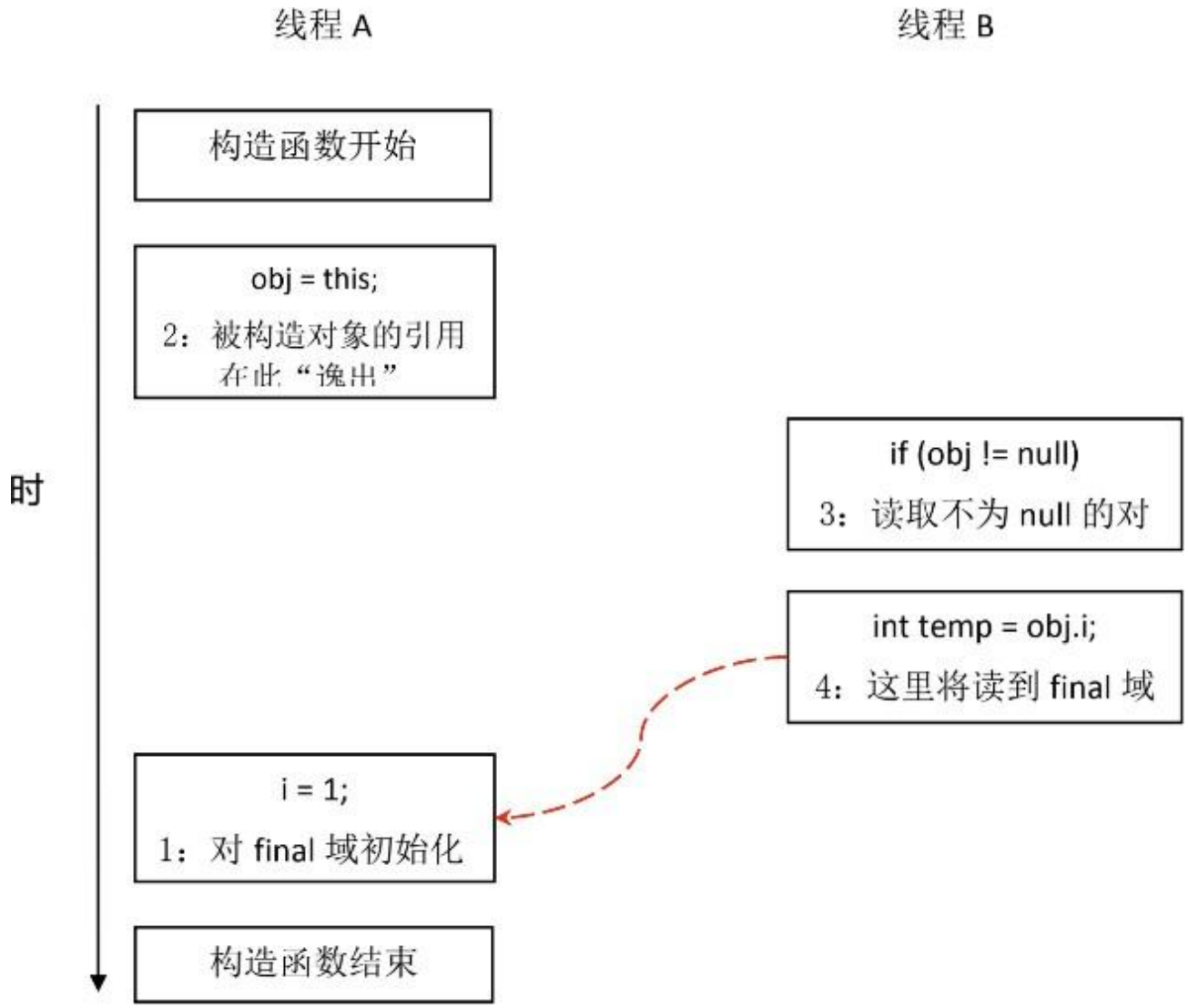
前面我们提到过，写final域的重排序规则可以确保：在引用变量为任意线程可见之前，该引用变量指向的对象的final域已经在构造函数中被正确初始化过了。其实要得到这个效果，还需要一个保证：

> 在构造函数内部，不能让这个被构造对象的引用为其他线程可见，也就是对象引用不能在构造函数中"逸出"。

为了说明问题，让我们来看下面示例代码：

```
public class FinalReferenceEscapeExample
  { final int i;
  static FinalReferenceEscapeExample obj;
  public FinalReferenceEscapeExample () {
      i = 1;                              //1写final域
      obj = this;                          //2 this引用在此"逸出"
  }
  public static void writer() {
      new FinalReferenceEscapeExample ();
  }
  public static void reader {
      if (obj != null) {                    //3
          int temp = obj.i;                 //4
      }
  }
}
```

假设一个线程A执行writer()方法，另一个线程B执行reader()方法。这里的操作2使得对象还未完成构造前就为线程B可见。即使这里的操作2是构造函数的最后一步，且即使在程序中操作2排在操作1后面，执行read()方法的线程仍然可能无法看到final域被初始化后的值，因为这里的操作1和操作2之间可能被重排序。实际的执行时序可能如下图所示：



从上图我们可以看出：在构造函数返回前，被构造对象的引用不能为其他线程可见，因为此时的final域可能还没有被初始化。在构造函数返回后，任意线程都将保证能看到final域正确初始化之后的值。

> 上面都是八股文，也很少会遇到上述使用场景，个人也仅作兴趣了解，避免踩坑，下面总结一下final的常用用法。

final关键字有几种用法

修饰普通变量

注意点：

- 用final关键字修饰的变量，只能进行一次赋值操作，并且在生存期内不可以改变它的值。final修饰的变量可以先声明，后赋值。

修饰成员变量

注意点：

- 必须初始化值。
- 被final修饰的成员变量赋值，有两种方式：1、直接赋值　2、全部在构造方法中赋初值。

- 如果修饰的成员变量是基本类型，则表示这个变量的值不能改变。
- 如果修饰的成员变量是一个引用类型，则是说这个引用的地址的值不能修改，但是这个引用所指向的对象里面的内容还是可以改变的。

```
public class Test {
    final int age = 18;
    final String name;
    final String[] hobby;
    public Test() {
        this.name = "lvmenglou";          // 正确使用
        //this.age = 20;                  // 错误使用
        this.hobby = new String[4];       // 正确使用
        this.hobby[0] = "movie";          // 正确使用
        this.hobby[1] = "sing song";      // 正确使用
    }
}
```

修饰方法

注意点：

- 被final修饰的方法不能被重写。
- 一个类的private方法会隐式的被指定为final方法。
- 如果父类中有final修饰的方法，那么子类不能去重写。

修饰类

注意点：

- 用final去修饰一个类的时候，表示这个类不能被继承；
- final类中的成员方法都会被隐式的指定为final方法；
- 被final修饰的类，final类中的成员变量可以根据自己的实际需要设计为final。

> 在自己设计一个类的时候，要想好这个类将来是否会被继承，如果可以被继承，则该类不能使用final修饰，在这里呢，一般来说工具类我们往往都会设计成为一个final类。在JDK中，被设计为final类的有String、System等。

final和static

很多时候会容易把static和final关键字混淆，static作用于成员变量用来表示只保存一份副本，而final的作用是用来保证变量不可变。看下面这个例子：

```
class MyClass {
    public final double i = Math.random();
    public static double j = Math.random();
}
public class Test2 {
    public static void main(String[] args)
        { MyClass myClass1 = new MyClass();
        MyClass myClass2 = new MyClass();
        System.out.println(myClass1.i);
        System.out.println(myClass1.j);
        System.out.println(myClass2.i);
        System.out.println(myClass2.j);

    }
}
// 输出：
// 0.6885279073145978
// 0.7678464493258529
// 0.5645174724833194
// 0.7678464493258529
```

运行这段代码就会发现，每次打印的两个j值都是一样的，而i的值却是不同的。从这里就可以知道final和static变量的区别了。

final和volatitle

这两个变量看起来没有什么关系，但是我学习JAVA内存模型时，感觉两者还是有很多异同，下面我简单总结一下。

> 禁止重排序规则：
>
> - 对于volatitle，volatitle变量前面的赋值和取值，只能排在volatitle变量的前面，volatitle变量后面赋值和取值，只能排在volatitle变量的后面，这个雷打不动！至于volatitle变量前面的赋值和取值，比如 a=1，b=2，他们的排序就不能保证了，可能是b=2，a=1，所以只能保证a和b是排在volatitle变量前 面，volatitle变量后面的排序规则同上。(volatitle的排序规则，是针对volatitle的前后变量而言的)
> - 对于final，它的禁止重排序规则和它前后的变量没有任何关系，可以排在final前，也可以排在final后。final的禁止重排序规则是针对构造函数而言的，也就是只有先给final赋值或者取值后，才能完成构造函数的初始化，但是普通变量，可能是先初始化构造函数，然后在给变量赋值。同理，当获取某个对象时，也是只能先成功获取该对象的引用，再去获取final成员变量的值，但是普通变量不能保证，可能会导致普通变量的读取，排在了获取对象引用的前面。（final的排序规则，是针对final的成员变量和对象的构造函数而言）

> 内存可见性：
>
> - 两者都有相同的内存可见性，对于final，这个值都不变，所有的线程看到的值都是一样的，当然所有线程可见。对于volatitle，它的值的改变，是强刷内存的，同时也会让其它线程的本地缓存的值也相应改变。

> 使用场景：
>
> - final主要是保证某个值不变，或者不类不被继承。volatitle主要是保证多个线程对该变量的内存可见性，常用于多个线程开始、结束的标记符等。所以说，两者的使用场景，简直是风马牛不相及。

总结

本章详细讲解了final的内存语义和使用方式，内存语义主要是涉及到并发编程相关的知识，仅供了解即可。其实我们最终还是需要注重final的使用方式，分别从变量、方法、类，对齐进行讲解，这块知识很简单，主要是做个记录，最后是final和static，也是网上看到的示例，也只做个简单的记录。

这些内容，其实是根据《深入理解Java内存模型》，然后结合网上的资料，做了相关整理和总结，其实都属于Java内存模型方面的内容，通过这4篇文章，大家对Java内存模型应该会有整体的了解，后面才真正讲解并发编程相关的知识。

参考资料：
《深入理解Java内存模型》
《Java并发编程实战》

📚 第 5 章：对象的共享

> 主要总结《Java并发编程实战》中“第3章:对象共享”的内容。

前言

在没有Java相关并发知识的前提下，第一次看这本书《Java并发编程实战》，其实有些看不太懂，因为里面的很多知识讲的比较抽象，比如可见性、volatile、final等讲的其实都不深入，所以导致自己理解的也很片面。后来就先专门看了“Java内存模型”相关的知识，再对相关知识理解起来，就要深入一些，所以才有了前面写的4篇关于“Java内存模型”相关的文章。

“第二章:基础知识”主要讲解线程安全性、原子性、加锁机制（主要讲解内置锁、synchronized重入）、用锁保护状态，这些知识在我相关系列的前面4篇中，已经讲的比较清楚，就直接跳过。

今天的这篇文章，主要是对《Java并发编程实战》中“第3章:对象共享”的内容进行总结，这章内容看了2遍，因为相关知识的匮乏，有些知识点还是理解的不全，所以也只能基于自己的理解，对所学内容总结一下，要不然很快就忘了，后续也会再重拾相关知识，再二次理解消化。

可见性

> 指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

在多线程环境下，一个线程对共享变量的操作对其他线程是不可见的。Java提供了volatile来保证可见性，当一个变量被volatile修饰后，表示着线程本地内存无效，当一个线程修改共享变量后他会立即被更新到主内存中，其他线程读取共享变量时，会直接从主内存中读取。当然，synchronize和Lock都可以保证可见性。synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

> 评价：该小节内容主要是讲解的volatile的可见性，提到volatile不具备原子性特性，也引出了synchronize，但是对于可见性，final其实可见性是最强的。这部分内容中规中矩，建议大家将volatile、synchronize和final的可见性、原子性对比起来看，最好能理解他们可见性的原理，以及重排序的机制，这样就能对“可见性”这一概念理解的更加深入。

发布和溢出

发布：发布一个对象，是对象能够在当前作用域之外的代码中使用。比如将对象的引用保存到其它代码可以访问的地方，或者在一个非私有方法中返回对象的引用，简单来说，就是外部可以访问到这个对象和里面的成员或者方法，为了保证多线程下没有问题，需要保证对象内部状态的封装性不被破坏。

> 书中写的有点像八股文，讲的也有点抽象，我理解其实就拿到一个对象时，需要保证对象内部数据已经完全初始化，然后对象内部的成员和方法，要么就完全封装，不能将修改的方式对外暴露，如果不能做到这一点，就需要保证修改和访问是线程安全的。

溢出：当一个不应该发布的对象被发布时，这种情况被称为溢出（Escaspe）。溢出的情况我总结有2种：

- 将成员变量的引用返回，外部就可以修改数据，多线程下可能会存在问题；
- 在构造函数过程中使this引用溢出。

成员变量溢出

```
class UnsafeStates {
  private String[] states = new String[] {"AK","AL" ...
  public String[] getStates {return states;}
}
```

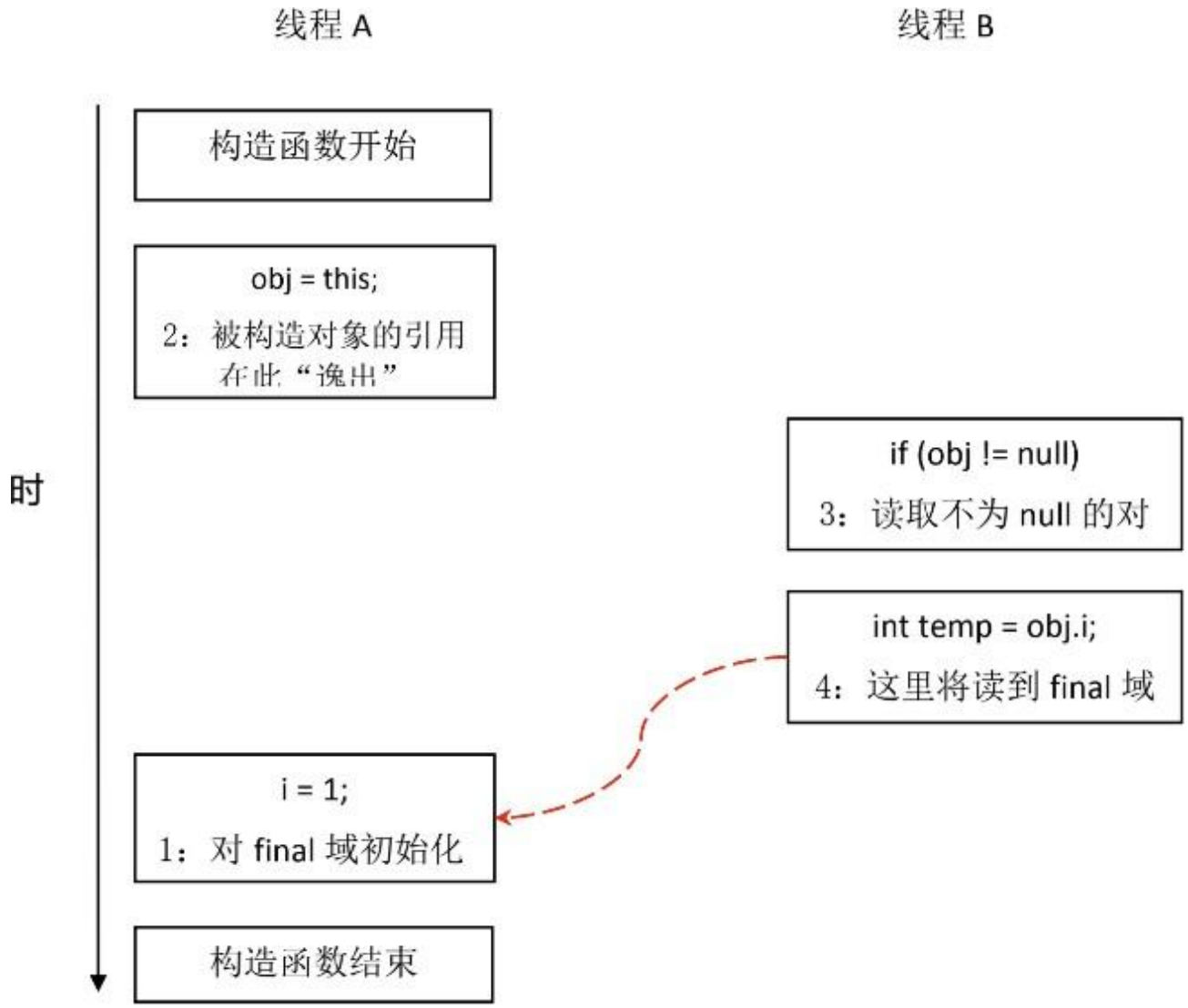
这个是书中的示例，当你发布这个对象时，任何调用者都可以修改数组中的内容，有人可能会说，我在states前面加一个final，这个其实解决不了问题，因为final可以保证states的地址不被改变，但是不能保证内部的数据不被改变。

this引用溢出

> 书中的注册示例没有看懂（还是自己太弱了。。。），我给一个前面一篇文章的示例，里面可以从原理讲解this引用是如何引出，这个应该比书中给的示例更能深入理解。

```
public class FinalReferenceEscapeExample {
  final int i;
  static FinalReferenceEscapeExample obj;
  public FinalReferenceEscapeExample () {
      i = 1;                                //1写final域
      obj = this;                           //2 this引用在此"逸出"
  }
  public static void writer() {
      new FinalReferenceEscapeExample ();
  }
  public static void reader
      { if (obj != null) {                  //3
          int temp = obj.i;                 //4
      }
  }
}
```

假设一个线程A执行writer()方法，另一个线程B执行reader()方法。这里的操作2使得对象还未完成构造前就为线程B可见。即使这里的操作2是构造函数的最后一步，且即使在程序中操作2排在操作1后面，执行read()方法的线程仍然可能无法看到final域被初始化后的值，因为这里的操作1和操作2之间可能被重排序。实际的执行时序可能如下图所示：



从上图我们可以看出：在构造函数返回前，被构造对象的引用不能为其他线程可见，因为此时的final域可能还没有被初始化。在构造函数返回后，任意线程都将保证能看到final域正确初始化之后的值。

线程封闭

文章讲解了“栈封闭”和“ThreadLocal类”这两部分内容：

- 栈封闭：“栈封闭”很好理解，类似于函数中的变量，函数结束后，变量内容自动释放，不会暴露对外，“栈封闭”其实讲的和这个原理类似，也就“栈封闭”的内容，肯定不会“溢出”。“栈封闭”可以很好的保证线程安全性。
- ThreadLocal类，这个其实和C++多线程的线程特定数据是一个道理，也就是每个线程可以有自己的ThreadLocal类，里面只保存本线程的数据，对其它线程不可见，一般该线程的全局变量，都可以保存到里面，其它线程不会干扰。

不变性

主要讲解final知识，强调了“不可变对象一定是线程安全的”，这个可以直接参考《Java并发编程系列4-final》这篇文章，里面的示例“使用volatile类型来发布不可变的对象”总结的不错，里面有一句话印象很深刻：

> 通过使用包含多个状态的容器对象来维持不变形条件，并使用一个volatile类型的引用来保证可见性，使得对象的在没有显示地使用锁的情况下，仍然是安全的。

这个怎么理解呢，书中的示例其实就是整了一个类OneValueCache和一个工厂Factory，工厂中有一个OneValueCache的成员变量，可以把Factory看成是获取OneValueCache对象的单例模式，如果需要这个单例模式对所有线程可见，就需要将该成员变量定义成volatile类型，保证所有线程可见，然后OneValueCache对象内部成员都是final，所以可以保证线程安全。上述方法总结一句话就是"通过final保证对象线程安全，通过volatile保证内存可见，实现多线性安全性"

安全发布

不安全发布

不正确的安全发布，会导致多线程运行时出现异常：

```
public class test
  { private int n;
  public void(int n) {
    this.n = n;
  }
  public void forTest()
    { if (n != n) {
      // 这里可能会进入，多线程下，因为重排序影响，this.n=n可能会排在构造函数完成外面，由于n!=n不能保证原子性，会出现问题。（如果还是不懂，建议先看一下重排序规则，里面很多类似的示例）
    }
  }
}
```

安全发布常用模式

书中总结了以下方法：

1. 在静态初始化函数中初始化一个对象的引用；
2. 将对象的引用保存在volatile类型的域或者AtomicReferance对象中；
3. 将对象的引用保存在某个正确构造对象的final类型域中；
4. 将对象的引用保存在一个由锁保护的域中。
   上面的4个方法，后面3个很好理解，对于第1个，因为静态初始器由JVM在类的初始化阶段执行，由于JVM内部存在这同步机制，因此通过这种方式初始化的任何对象都可以安全地发布：

```
public static Holder holder = new Holder(18);
```

其它概念

文章提到了"事实不可变对象"，这个概念有点绕，文中解释为"如果对象从技术上看是可变的，但是状态在发布后不会再改变"。然后给了个例子：

```
public Map<String, Data> test = Collections.synchronizedMap(new HashMap<String, Data>
())
```

虽然Data对象可变，但是test不可变，也就是通过不可变容器或者其它方式，来装载可变对象，让其处理成不可变的方式。

> 有点绕，还是个八股文，就稍微了解即可。

对于可变对象，处理时就需要加锁。

总结

学习Java并发编程，前面的基础知识学了快2周，主要包括JMM、重排序规则、原子性、可见性和安全发布，很多知识都是围绕volatile、synchronize、final三者展开，之所以学了这么久，主要还是想把基础打牢，后面的应用总结和讲解，可能就没有这么细致。

📚 第 6 章：同步工具类

> 主要讲解Java常用的同步工具类，包括闭锁/FutureTask/信号量/栅栏，最后还对"创建线程的三种方式"进行简单的扫盲。

前言

《Java并发编程实战》这本书看到第五章了，里面的同步工具类感觉比较常用，就简单总结一下。不过在讲"同步工具类"前，大家需要对创建线程的三种方法非常清楚，如果这个不清楚的话，直接看示例可能不太懂，文章最后面有"创建线程的三种方式"内容，已经给Java小白扫盲，谁让楼哥是暖男呢。

同步工具类

闭锁

用途：可用于命令一组线程在同一个时刻开始执行某个任务，或者等待一组相关的操作结束，尤其适合计算并发执行某个任务的耗时。

```
public class CountDownLatchTest {
```

```
public void timeTasks(int nThreads, final Runnable task) throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1); final
        CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++) { Thread t = new
                Thread() {
                    @Override
                    public void run()  { try {
                            // 阻塞，等待startGate.countDown()的执行
                            startGate.await(); try {
                                task.run();
                            } finally {
                                // 每次执行完毕后，计数器减1，表示有一个事件已经完成
                                endGate.countDown();
                            }
                        } catch(InterruptedException e) {
                            System.out.println("Throw Exception, e:" + e.toString());
                        }
                    }
                };
                // 启动线程
                t.start();
        }

        long start = System.nanoTime(); System.out.println("打开闭锁
        ");
        startGate.countDown(); // 打开开关，进入startGate.await()后面的逻辑
        endGate.await(); // 等待所有线程endGate.countDown()全部执行完毕
        long end = System.nanoTime();
        System.out.println("闭锁退出，共耗时" + (end-start));
}

class RunnableTask implements Runnable { @Override
        public void run() {
                System.out.println("当前线程为：" + Thread.currentThread().getName());
        }
}
```

```
public static void main(String args[]) throws InterruptedException { CountDownLatchTest
```

```
            test = new CountDownLatchTest(); test.timeTasks(5, test.new RunnableTask());
        }
    }
    // 输出：
    // 打开闭锁
```

```
// 当前线程为：Thread-0
// 当前线程为：Thread-2
// 当前线程为：Thread-1
// 当前线程为：Thread-3
// 当前线程为：Thread-4
// 闭锁退出，共耗时1985771
```

里面的注释其实已经很清晰了，简单总结一下流程：

1. 初始化startGate和endGate的计数器，分别为1和5；
2. 开启5个线程，去执行RunnableTask任务；
3. 5个线程启动后，会全部阻塞在startGate.await()；
4. 当调用startGate.countDown()，startGate计数器为0，线程阀门放开，开始一起去执行每个线程任务task.run()；
5. 每个线程执行完毕后，会调用endGate.countDown()，每调用一次，endGate的计数器减去1，线程执行过程中，主线程通过endGate.await()阻塞；
6. 当所有线程执行完毕，endGate计数器为0，主线程endGate.await()阻塞放开，执行后面收尾流程，流程结束。

> 这个和Go的sync.WaitGroup，简直一毛一样啊！看来语言的设计，很多都差不多。

FutureTask

FutureTask也可以用作闭锁，它是通过Callabale来实现，相当于一种可以生成结果的Runnable，并且可以处于以下3种状态：等待状态，正在运行和运行完成。Future.get的行为取决于任务的状态，如果任务已经完成，那么get会立即返回结果，否则get将阻塞到任务进入完成状态，然后返回结果或者抛出异常。

```
@Data
@Service
public class Cat {
    private String catName;
}

public class Preloader {
    private final FutureTask<Cat> future = new FutureTask<Cat>(new Callable<Cat>()
        { @Override
        public Cat call() throws InterruptedException
            { Cat cat = new Cat();
            cat.setCatName("罗小 ");
            for (int i = 1; i <= 5; i ++) {
                // 睡眠1秒，方便大家看执行效果
                Thread.sleep(1000L);
                System.out.println("Sleep " + i + " 秒");
            }
            return cat;
        }
    ```

    private final Thread thread = new Thread(future);
```
```

```
    public void start() { System.out.println("启动Start");
        thread.start();
    }

    public Cat get() throws ExecutionException, InterruptedException { Cat cat = null;
        try {
            System.out.println("开始获取数据！");
            // 阻塞，等待线程执行完毕
            cat = future.get(); System.out.println("获取数
            据成功！");
        } catch (ExecutionException e) {
            // 异常处理，省略...
        }
        return cat;
    }

    public static void main(String args[]) throws ExecutionException, InterruptedException {
        Preloader preloader = new Preloader(); preloader.start();
        Cat cat = preloader.get(); System.out.println(cat.toString());
    }
}
// 输出：
// 启动Start
// 开始获取数据！
// Sleep 1 秒
// Sleep 2 秒
// Sleep 3 秒
// Sleep 4 秒
// Sleep 5 秒
// 获取数据成功！
// Cat(catName=罗小黑)
```

我们可以看到，获取Cat数据时，主线程一直阻塞住，直到Cat成功构造好数据后，才正常返回，简单总结一下流程：

1. 初始化线程对象Thread和FutureTask静态对象，其中FutureTask的Callable是构造Cat数据；
2. 启动线程thread.start()，开始执行future中Callable.call()方法，开始构造Cat数据；
3. 在Cat数据构造成功前，future.get()会一直阻塞，直到future中Callable.call()成功返回，阻塞结束。

信号量

用途：用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。计数信号量可以用来实现某种资源池，或者对容器施加边界。

```
public class SemaphoreTest<T> { public final
    Set<T> set;

    private final Semaphore sema;

    public SemaphoreTest(int bound){
        this.set = Collections.synchronizedSet(new HashSet<T>()); this.sema =
        new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException{ sema.acquire();
        boolean wasAdded = false; try{
            wasAdded = set.add(o);
            return wasAdded;
        }finally{
            if(!wasAdded){ sema.release();
            }
        }
    }

    public boolean remove(T o){
        boolean wasRemoved = set.remove(o); if(wasRemoved){
            sema.release();
        }
        return wasRemoved;
    }

    public static void main(String[] args) throws InterruptedException{ int permits = 5;
        SemaphoreTest<Integer> test = new SemaphoreTest<Integer>(permits); for(int i = 0; i < 10;
        i++){
            test.add(i);
            System.out.println("set:" + test.set);
        }
    }
}
//  输出：
```

```
//  set:[0]
//  set:[0,       1]
//  set:[0,       1,  2]
//  set:[0,       1,  2, 3]
//  set:[0,       1,  2, 3, 4]
```

这个示例就很简单，描述一下流程：

1. 先初始化信号量sema的许可个数为5；
2. 通过add()添加数据，每添加一个数据，就消耗sema的一个许可；
3. 当5个许可全部消耗完毕后，如果需要再添加数据，因为sema的许可为0，阻塞请求。
   备注：如果这个时候调用了sema.release()，会释放一个许可，那么add()会继续添加一个元素，之后的请求继续阻塞，直到有新的许可释放。

栅栏

上面介绍的都是闭锁的几种实现方式，栅栏类似于闭锁，它能阻塞一组线程直到某个时间发生。栅栏和闭锁的关键区别在于，所有线程必须同时到达栅栏位置，才能继续执行。书中有一句哈总结的很好：

> 闭锁用于等待事件，而栅栏用于等待其它线程。

用途：用于阻塞一组线程直到某个事件发生。所有线程必须同时到达栅栏位置才能继续执行下一步操作，且能够被重置以达到重复利用。而闭锁式一次性对象，一旦进入终止状态，就不能被重置。

```
public class CyclicBarrierWorker implements Runnable { private int id;
    private CyclicBarrier barrier;

    public CyclicBarrierWorker(int id, final CyclicBarrier barrier) { this.id = id;
        this.barrier = barrier;
    }

    @Override
    public void run() { try {
            if (id == 5) {
                // 让第5个线程sleep 10秒
                Thread.sleep(10000);
            }
            System.out.println(id + " people wait");
            barrier.await(); // 大家等待最后一个线程到达
        } catch (InterruptedException | BrokenBarrierException e) { e.printStackTrace();
        }
    }
}

class TestCyclicBarrier {
    public static void main(String[] args) { int num = 10;
        // 新建一个栅栏
        CyclicBarrier barrier = new CyclicBarrier(num, new Runnable() { @Override
            public void run() {
```

```
                // num个线程全部执行完毕，且都调用barrier.await()，才会去执行该方法
                // 可以理解为计数器初始值为num，每调用一次barrier.await()，计数器-1，直到计数器等
于0
```

```
                System.out.println("go on together!");
            }
        });
        for (int i = 1; i <= num; i++) {
            new Thread(new CyclicBarrierWorker(i, barrier)).start();
        }
    }
}
// 输出：
// 1 people wait
// 3 people wait
// 4 people wait
// 2 people wait
// 6 people wait
// 7 people wait
// 8 people wait
// 9 people wait
// 10 people wait
// 5 people wait
// go on together!
```

我故意让第5个线程sleep了10秒，所以大家都等第五个线程，全部执行完毕后，再一起去执行栅栏中的任务，简单总结一下流程：

1. 新建一个栅栏，第一个参数num是线程个数，第二个参数是栅栏需要执行的任务；
2. 启动10个线程，每个线程传入栅栏变量，这10个线程开始执行，然后都阻塞在barrier.await()，大家都在等待最后一个线程的到达；
3. 当最后一个线程到达barrier.await()后，阻塞放开，开始执行栅栏中的方法。

创建线程的三种方式

> 如果大家对线程创建非常清楚，可以直接跳过“创建线程的三种方式”这部分内容，该部分内容主要给Java小白扫盲。

继承Thread类

继承Thread类并复写run()方法，是一种很简单的方式，代码如下：

```
public class MyThread extends Thread {
    public MyThread(String name)
        { super(name);
    }
    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + "  经运行");

    }
    public static void main(String[] args) {
```

```
        new MyThread("线程一").start();
    }
}
// 输出：
// 线程一  经运行
```

实现Runnable接口

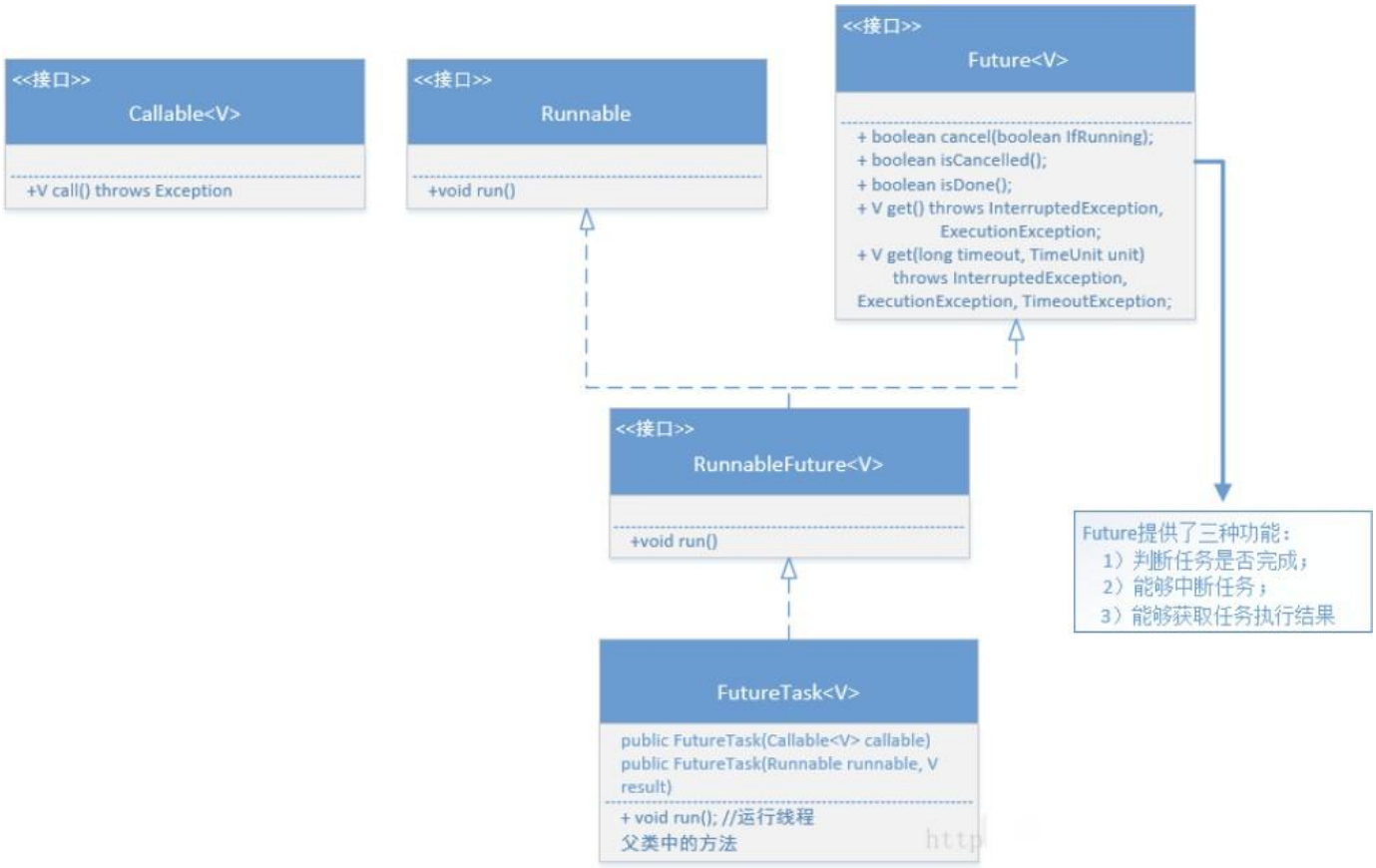
这个是我们经常使用的方式之一，代码如下：

```
public class MyTask implements Runnable
    { @Override
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + "  经运行");
    }
    public static void main(String[] args) {
        new Thread(new MyTask(),"线程二").start();
    }
}
// 输出：
// 线程二  经运行
```

> Thread类与Runnable接口的比较：
>
> - 由于Java“单继承，多实现”的特性，Runnable接口使用起来比Thread更灵活;
> - Runnable接口出现更符合面向对象，将线程单独进行对象的封装;
> - Runnable接口出现，降低了线程对象和线程任务的耦合性;
> - 如果使用线程时不需要使用Thread类的诸多方法，显然使用Runnable接口更为轻
>
> 量;所以，我们通常优先使用“实现Runnable接口”这种方式来自定义线程类。

Callable与Future创建线程

我们看到不管是Thread还是Runable接口，其run()都是无返回值的，并且无法抛出异常的，如果我们有需要返回值或者抛出异常怎么办？这个时候就需要用到Callable与Feature了。先来看类的继承关系:



Callable接口：

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

那一般是怎么使用Callable的呢？Callable一般是配合线程池工具ExecutorService来使用的，后面会给一个简单的示例。

Future接口只有几个比较简单的方法：

```
public interface Future<V> {
    // 取消任务，如果任务正在运行的，mayInterruptIfRunning为true时，表明这个任务会被打断的，并返回true；
    // 为false时，会等待这个任务执行完，返回true；若任务还没执行，取消任务后返回true，如任务执行完，返回false
    boolean cancel(boolean mayInterruptIfRunning);
    // 判断任务是否被取消了,正常执行完不算被取消
    boolean isCancelled();
    // 判断任务是否已经执行完成，任务取消或发生异常也算是完成，返回true boolean
    isDone();
    // 获取任务返回结果，如果任务没有执行完成则等待完成将结果返回，如果获取的过程中发生异常就抛出异常，
    // 比如中断就会抛出InterruptedException异常等异常
```

```
    V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,
TimeoutException;
}
```

RunnableFuture是对Runnable和Future进行组装：

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     */
    void run();
}
```

最后就是FutureTask和Callable，Callable是一个接口，里面有个V call()方法，这个V就是我们返回值类型。FutureTask类的构造函数，我们发现其中一个构造函数的参数是Callable类型，所以是通过FutureTask对Callable进行调用，并对线程进行管理。 Callable与Future的用法如下：

```
public class CallableTest {
    private void callTest() {
        //这里指定返回String类型
        Callable<String> callable = new Callable<String>() { @Override
            public String call() throws Exception
                { System.out.println("Callable 已经运行啦"); return "this is
                Callable is running";
            }
        };
        FutureTask<String> futureTask = new FutureTask<String>(callable);
        futureTask.run();
        try {
            if (futureTask.isDone()){                //任务完成
                System.out.println(futureTask.get());
            }
        } catch (InterruptedException e) { e.printStackTrace();
        } catch (ExecutionException e) { e.printStackTrace();

    public static void main(String[] args) {
        CallableTest test = new CallableTest();
        test.callTest();
    }
}
```

简单说一下调用流程：

1. 先初始化一个Callable匿名对象，然后重写call()方法；
2. 初始化一个futureTask对象，用Callable作为入参；
3. 直接执行futureTask.run()方法，调度任务；
4. 通过futureTask.isDone()判断任务是否完成，完成后，阻塞放开。

> 原理：其实FutureTask内部实现比较简单，Callable就是他的任务，而FutureTask内部维护了一个任务状态，所有的状态都是围绕这个任务来进行的，随着任务的进行，状态也在不断的更新。任务发起者调用get()方法时，如果任务没有执行完成，会将当前线程放入阻塞队列等待，当任务执行完后，会唤醒阻塞队列中的线程。

扩展一下：Callable可以交给FutureTask处理，也可以交给线程池处理。

```
public class CallableTest2 {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService executors = Executors.newCachedThreadPool();
        Future<String> future;
        for(int i = 0; i < 5 ; i++){
            // 直接把Callable对象扔给线程池，由线程池执行
            future = executors.submit(new CallableImpl(Integer.toString(i)));
            System.out.println(future.get());
        }
        executors.shutdown();

class CallableImpl implements Callable<String> {
    private String callableName;
    public CallableImpl(String callableName)
        { this.callableName = callableName;
    }
    @Override
    public String call() throws Exception
        { return "当前Callable名为: " +
        callableName;
    }
```

下面看一个Callable和线程池的调用示例：

这个就很简单，直接整个Callable，然后扔给线程池去调度。

总结

这篇文章讲解了Java常用的同步工具类，这是很多是《Java并发编程实战》书里面的内容，但是当我看书时，感觉书中讲的知识并不容易懂，对于不懂的地方，我就到网上找一些相关示例，或者对一些基础知识做一些扫盲。所以大家看书时，不懂的地方可以跳过去，然后再单独对于不懂的知识点，到网上查阅资料，因为网上有很多博客，写的真的是非常好，比很多书籍作者表述的要好很多，这个也算是我看书的一点点心得。

📚 第 7 章：线程池基本知识

> 主要讲解Java线程池的基础知识。

前言

目前书籍《Java并发编程实战》看到“第7章:取消与关闭”，里面涉及到部分线程池的内容，然后第8章就是线程池，所以打算把之前看的线程池的资料再整理一下，便于后面能更好理解书中的内容。

之前看过一篇博客，关于线程池的内容讲解的非常好，我只截取基础知识部分，把Java基础内容全部掌握后，再对里面的原理部分进行深入理解，后面会附上该篇博客的链接。

初识线程池

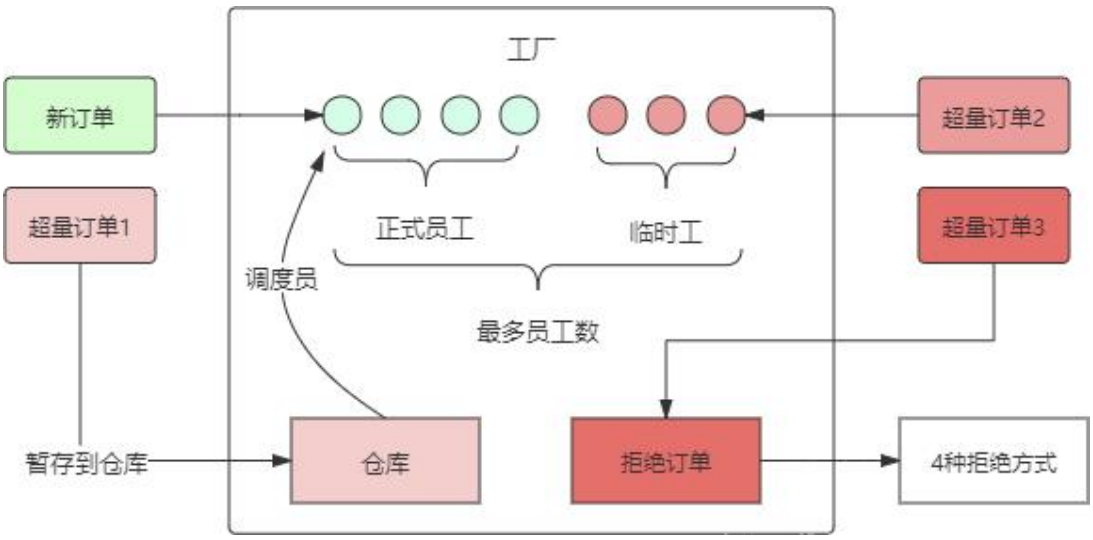
我们知道，线程的创建和销毁都需要映射到操作系统，因此其代价是比较高昂的。出于避免频繁创建、销毁线程以及方便线程管理的需要，线程池应运而生。

线程池优势

- 降低资源消耗：线程池通常会维护一些线程（数量为 corePoolSize），这些线程被重复使用来执行不同的任务，任务完成后不会销毁。在待处理任务量很大的时候，通过对线程资源的复用，避免了线程的频繁创建与销毁，从而降低了系统资源消耗。
- 提高响应速度：由于线程池维护了一批 alive 状态的线程，当任务到达时，不需要再创建线程，而是直接由这些线程去执行任务，从而减少了任务的等待时间。
- 提高线程的可管理性：使用线程池可以对线程进行统一的分配，调优和监控。

线程池设计思路

有句话叫做艺术来源于生活，编程语言也是如此，很多设计思想能映射到日常生活中，比如面向对象思想、封装、继承，等等。今天我们要说的线程池，它同样可以在现实世界找到对应的实体——工厂。先假想一个工厂的生产流程：

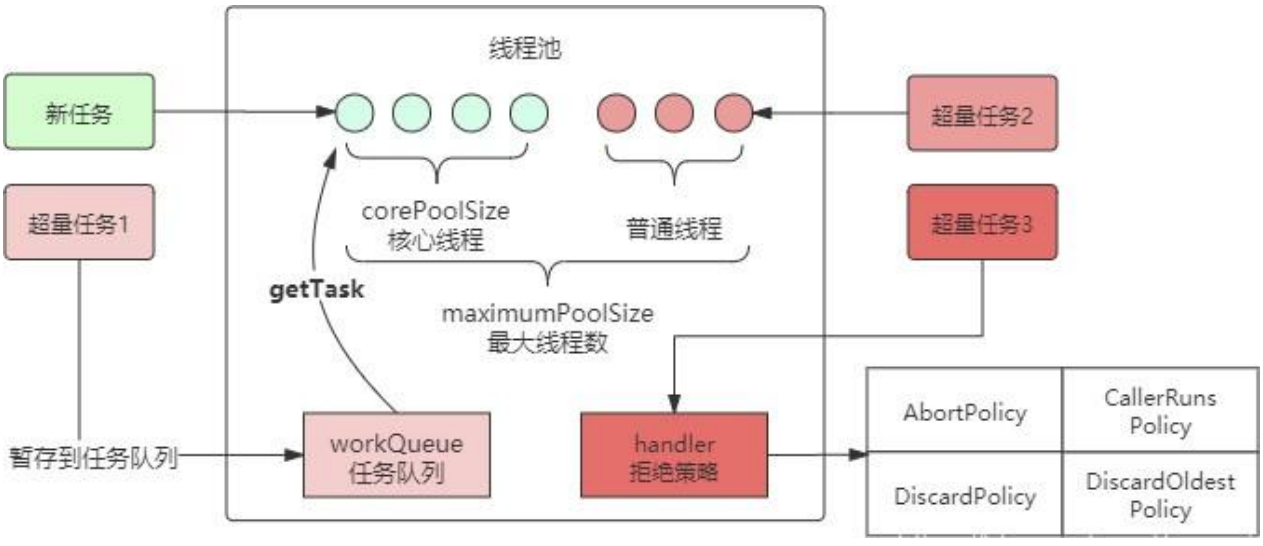


工厂中有固定的一批工人，称为正式工人，工厂接收的订单由这些工人去完成。当订单增加，正式工人已经忙不过来了，工厂会将生产原料暂时堆积在仓库中，等有空闲的工人时再处理（因为工人空闲了也不会主动处理仓库中的生产任务，所以需要调度员实时调度）。仓库堆积满了后，订单还在增加怎么办？工厂只能临时扩招一批工人来应对生产高峰，而这批工人高峰结束后是要清退的，所以称为临时工。当时临时工也招满后（受限于工位限制，临时工数量有上限），后面的订单只能忍痛拒绝了。我们做如下一番映射：

- 工厂——线程池
- 订单——任务（Runnable）
- 正式工人——核心线程
- 临时工——普通线
- 程仓库——任务队
- 列 调度员——
  getTask()

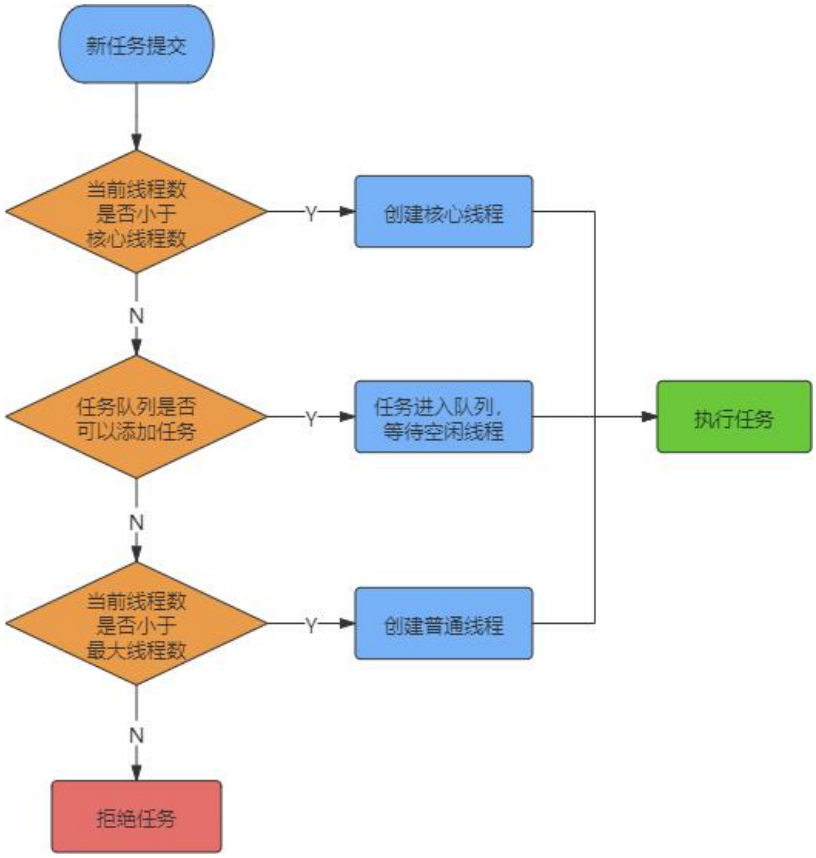
> getTask()是一个方法，将任务队列中的任务调度给空闲线程。

映射后，形成线程池流程图如下，两者是不是有异曲同工之妙？



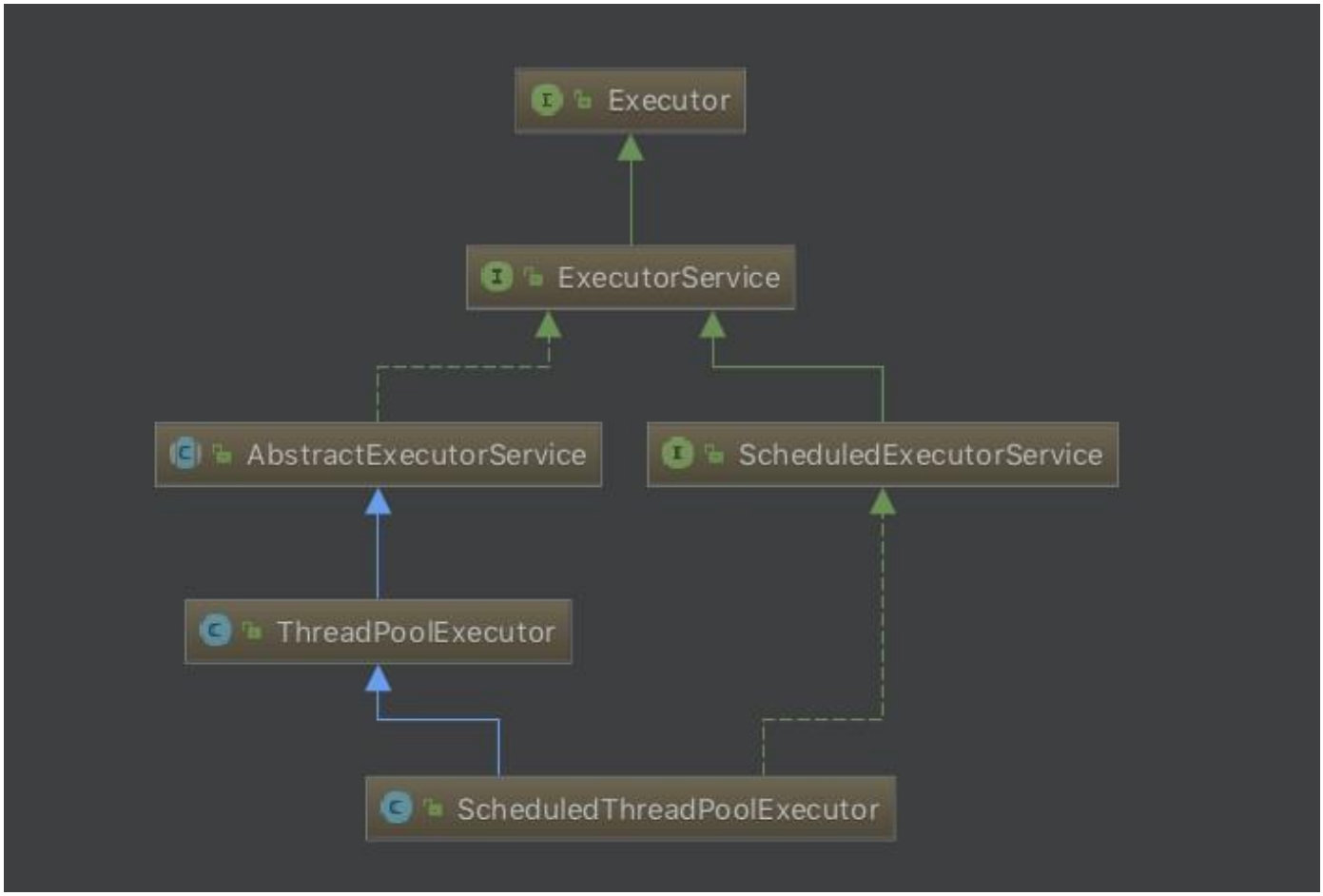
> 点评一下：感觉作者的这个类比，太TM经典了！！！直接抓住了线程调用的精髓。这就是为什么看书时不能只盯着书看，可能你看半天都不懂书中讲的啥，找一篇经典的博客，瞬间给你拨开云雾，而且还印象深刻。

这样，线程池的工作原理或者说流程就很好理解了，提炼成一个简图：



深入线程池

那么接下来，问题来了，线程池是具体如何实现这套工作机制的呢？从Java线程池Executor框架体系可以看出：线程池的真正实现类是ThreadPoolExecutor，因此我们接下来重点研究这个类。



构造方法

研究一个类，先从它的构造方法开始。ThreadPoolExecutor提供了4个有参构造方法：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue)
    { this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
    workQueue,
             Executors.defaultThreadFactory(), defaultHandler);

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             threadFactory, defaultHandler);
}

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
```

```
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), handler);
}

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

> 感觉构造方法有点多，其实前面的基础参数都是一样的，就后面两个可选参数"线程工厂"和"拒绝策略"不一样，组合一下就是2*2=4种情况。

解释一下构造方法中涉及到的参数：

- corePoolSize（必需）：核心线程数。即池中一直保持存活的线程数，即使这些线程处于空闲。但是将allowCoreThreadTimeOut参数设置为true后，核心线程处于空闲一段时间以上，也会被回收。
- maximumPoolSize（必需）：池中允许的最大线程数。当核心线程全部繁忙且任务队列打满之后，线程池会临时追加线程，直到总线程数达到maximumPoolSize这个上限。
- keepAliveTime（必需）：线程空闲超时时间。当非核心线程处于空闲状态的时间超过这个时间后，该线程将被回收。将allowCoreThreadTimeOut参数设置为true后，核心线程也会被回收。
- unit（必需）：keepAliveTime参数的时间单位。有：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）
- workQueue（必需）：任务队列，采用阻塞队列实现。当核心线程全部繁忙时，后续由execute方法提交的Runnable将存放在任务队列中，等待被线程处理。
- threadFactory（可选）：线程工厂。指定线程池创建线程的方式。
- handler（可选）：拒绝策略。当线程池中线程数达到maximumPoolSize且workQueue打满时，后续提交的任务将被拒绝，

handler可以指定用什么方式拒绝任务。

> 了解完“线程的设计思路”，再看构造方法的这些参数，感觉就非常容易懂了。

任务队列

使用ThreadPoolExecutor需要指定一个实现了BlockingQueue接口的任务等待队列。在ThreadPoolExecutor线程池的API文档中，一共推荐了三种等待队列：

1. SynchronousQueue：同步队列。这是一个内部没有任何容量的阻塞队列，任何一次插入操作的元素都要等待相对的删除/读取操作，否则进行插入操作的线程就要一直等待，反之亦然。
2. LinkedBlockingQueue：无界队列（严格来说并非无界，上限是Integer.MAX_VALUE），基于链表结构。使用无界队列后，当核心线程都繁忙时，后续任务可以无限加入队列，因此线程池中线程数不会超过核心线程数。这种队列可以提高线程池吞吐量，但代价是牺牲内存空间，甚至会导致内存溢出。另外，使用它时可以指定容量，这样它也就是一种有界队列了。
3. ArrayBlockingQueue：有界队列，基于数组实现。在线程池初始化时，指定队列的容量，后续无法再调整。这种有界队列有利于防止资源耗尽，但可能更难调整和控制。

另外，Java还提供了另外4种队列：

1. PriorityBlockingQueue：支持优先级排序的无界阻塞队列。存放在PriorityBlockingQueue中的元素必须实现 Comparable接口，这样才能通过实现compareTo()方法进行排序。优先级最高的元素将始终排在队列的头 部；PriorityBlockingQueue不会保证优先级一样的元素的排序，也不保证当前队列中除了优先级最高的元素以外的元素，随时处于正确排序的位置。
2. DelayQueue：延迟队列。基于二叉堆实现，同时具备：无界队列、阻塞队列、优先队列的特征。DelayQueue延迟队列中存放的对象，必须是实现Delayed接口的类对象。通过执行时延从队列中提取任务，时间没到任务取不出来。更多内容请见DelayQueue。
3. LinkedBlockingDeque：双端队列。基于链表实现，既可以从尾部插入/取出元素，还可以从头部插入元素/取出元素。
4. LinkedTransferQueue：由链表结构组成的无界阻塞队列。这个队列比较特别的时，采用一种预占模式，意思就是消费者线程取元素时，如果队列不为空，则直接取走数据，若队列为空，那就生成一个节点（节点元素为null）入队，然后消费者线程被等待在这个节点上，后面生产者线程入队时发现有一个元素为null的节点，生产者线程就不入队了，直接就将元素填充到该节点，并唤醒该节点等待的线程，被唤醒的消费者线程取走元素。

> 感觉这提供的队列有些多啊，总共7个！说实话，我现在还不知道实际场景用哪个比较好，后面我们可以看看Java封装好的线程池，里面用的队列都是哪种。

拒绝策略

线程池有一个重要的机制：拒绝策略。当线程池workQueue已满且无法再创建新线程池时，就要拒绝后续任务了。拒绝策略需要实现RejectedExecutionHandler接口，不过Executors框架已经为我们实现了4种拒绝策略：

1. AbortPolicy（默认）：丢弃任务并抛出RejectedExecutionException异常。
2. CallerRunsPolicy：直接运行这个任务的run方法，但并非是由线程池的线程处理，而是交由任务的调用线程处理。
3. DiscardPolicy：直接丢弃任务，不抛出任何异常。

4. DiscardOldestPolicy：将当前处于等待队列列头的等待任务强行取出，然后再试图将当前被拒绝的任务提交到线程池执行。

线程工厂指定创建线程的方式，这个参数不是必选项，Executors类已经为我们非常贴心地提供了一个默认的线程工厂：

```
/**
 * The default thread factory
 */
static class DefaultThreadFactory implements ThreadFactory {
    private static final AtomicInteger poolNumber = new AtomicInteger(1);
    private final ThreadGroup group;
    private final AtomicInteger threadNumber = new AtomicInteger(1);
    private final String namePrefix;

    DefaultThreadFactory() {
        SecurityManager s = System.getSecurityManager();
        group = (s != null) ? s.getThreadGroup() :
                              Thread.currentThread().getThreadGroup();
        namePrefix = "pool-" +
                      poolNumber.getAndIncrement() +
                     "-thread-";
    }

    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r,
                              namePrefix + threadNumber.getAndIncrement(),
                              0);
        if (t.isDaemon())
            t.setDaemon(false);
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
    }
}
```

线程池状态

线程池有5种状态：

```
volatile int runState;
// runState is stored in the high-order bits
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;
```

runState表示当前线程池的状态，它是一个 volatile 变量用来保证线程之间的可见性。

下面的几个static final变量表示runState可能的几个取值，有以下几个状态：

- RUNNING：当创建线程池后，初始时，线程池处于RUNNING状态；
- SHUTDOWN：如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的

任务，它会等待所有任务执行完毕；

- STOP：如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；
- TERMINATED：当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

初始化&容量调整&关闭

线程初始化

默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。

在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

- prestartCoreThread()：boolean prestartCoreThread()，初始化一个核心线程
- prestartAllCoreThreads()：int prestartAllCoreThreads()，初始化所有核心线程，并返回初始化的线程数

```
public boolean prestartCoreThread() {
    return addIfUnderCorePoolSize(null); //注意传进去的参数是null
}

public int prestartAllCoreThreads() {
    int n = 0;
    while (addIfUnderCorePoolSize(null))//注意传进去的参数是null
    ++n;
    return n;
}
```

线程池关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭：

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

线程池容量调整

ThreadPoolExecutor提供了动态调整线程池容量大小的方法：

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

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

使用线程池

ThreadPoolExecutor

通过构造方法使用ThreadPoolExecutor是线程池最直接的使用方式，下面看一个实例：

```
public class ThreadPoolExecutorTest {
    public static void main(String args[]) {
        // 创建线程池(核心线程数是3，最大线程数是5，超时时间是5秒)
        ThreadPoolExecutor threadPool = new ThreadPoolExecutor(3,5,5, TimeUnit.SECONDS, new
                ArrayBlockingQueue<Runnable>(5));

        // 向线程池提交任务
        for (int i = 0; i < threadPool.getCorePoolSize(); i++) { threadPool.execute(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < 2; j++) { System.out.println(Thread.currentThread().getName() + ":" + j);
                        try {
                            Thread.sleep(2000);
                        } catch (InterruptedException e) { e.printStackTrace();
                        }
                    }
                }
            });
        }
        // 关闭线程池
        threadPool.shutdown(); // 设置线程池的状态为SHUTDOWN，然后中断所有没有正在执行任务的线程
        // threadPool.shutdownNow(); // 设置线程池的状态为STOP，然后尝试停止所有的正在执行或暂停任务的
线程，并返回等待执行任务的列表，该方法要慎用，容易造成不可控的后果
    }
}
// 输出：
// pool-1-thread-1:0
// pool-1-thread-3:0
// pool-1-thread-2:0
// pool-1-thread-2:1
// pool-1-thread-1:1
// pool-1-thread-3:1
```

Executors封装线程池

另外，Executors封装好了4种常见的功能线程池（还是那么地贴心）：

FixedThreadPool

固定容量线程池。其特点是最大线程数就是核心线程数，意味着线程池只能创建核心线程，keepAliveTime为0，即线程执行完任务立即回收。任务队列未指定容量，代表使用默认值Integer.MAX_VALUE。适用于需要控制并发线程的场景。

```
// 使用默认线程工
public static ExecutorService newFixedThreadPool(int nThreads)
    { return new ThreadPoolExecutor(nThreads, nThreads,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>());
}
// 需要自定义线程工
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>(),
                                    threadFactory);
}
```

使用示例：

```
public class FixedThreadPoolTest {
    public static void main(String args[]) {
        // 创建线程池对象，设置核心线程和最大线程数为5
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5); fixedThreadPool.execute(new
        Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + " is running."); try {
                    Thread.sleep(10000L);
                } catch (InterruptedException e) { e.printStackTrace();
                    System.out.println("Throw Exception.");
                }
                System.out.println(Thread.currentThread().getName() + " after sleep, is still running.");
            }
        });
        //fixedThreadPool.shutdown();
        fixedThreadPool.shutdownNow(); // 不建议这样使用，很危险，这里仅用于测试
    }
}
// 输出：
// pool-1-thread-1 is running.
// Throw Exception.
// pool-1-thread-1 after sleep, is still running.
// java.lang.InterruptedException: sleep interrupted
//   at java.lang.Thread.sleep(Native Method)
```

```
//  at com.java.parallel.pool.FixedThreadPoolTest$1.run(FixedThreadPoolTest.java:15)
//  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
//  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
```

```
//  at java.lang.Thread.run(Thread.java:748)
```

> 作者的这个示例，我加了一点料，我就想看看直接中断会是什么效果，结果发现直接中断后，线程直接抛出异常，我捕获异常后，输出了一些结果。正常情况下，捕获异常是需要做一些处理，我这里仅作测试。

SingleThreadExecutor

单线程线程池。特点是线程池中只有一个线程（核心线程），线程执行完任务立即回收，使用有界阻塞队列（容量未指定，使用默认值Integer.MAX_VALUE）

```
public static ExecutorService newSingleThreadExecutor()
    { return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
// 为节省篇幅，省略了自定义线程工厂方式的源码
```

使用示例：

```
public class SingleThreadExecutorTest
    { public static void main(String args[])
    {
        // 创建线程池对象，设置核心线程和最大线程数为1
        ExecutorService singleThreadPool = Executors.newSingleThreadExecutor();
        singleThreadPool.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + " is running.");
            }
        });
        singleThreadPool.shutdown();
    }
}
// 输出：
```

ScheduledThreadPool

定时线程池。指定核心线程数量，普通线程数量无限，线程执行完任务立即回收，任务队列为延时阻塞队列。这是一个比较特别的线程池，适用于执行定时或周期性的任务。

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
    { return new ScheduledThreadPoolExecutor(corePoolSize);
}

// 继承了 ThreadPoolExecutor
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor
        implements ScheduledExecutorService {
```

```
    // 构造函数，省略了自定义线程工厂的构造函数
  public ScheduledThreadPoolExecutor(int corePoolSize)
      { super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
            new DelayedWorkQueue());
  }

  // 延时执行任务
  public ScheduledFuture<?> schedule(Runnable command,
                                     long delay,
                                     TimeUnit unit) {
        ...
    }
  // 定时执行任务
  public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                long initialDelay,
                                                long period,
                                                TimeUnit unit) {  }
}
```

使用示例：

```
public class ScheduledThreadPoolTest
    { public static void main(String args[])
    {
        // 创建定时线程池
        ScheduledExecutorService scheduledThreadPool =
Executors.newScheduledThreadPool(5);
        // 向线程池提交任务
        scheduledThreadPool.schedule(new
            Runnable(){ public void run() {
                System.out.println(Thread.currentThread().getName() + "--->运行");
            }
        }, 5, TimeUnit.SECONDS); // 延迟5s后执行任务
        scheduledThreadPool.shutdown();
    }
}
// 输出：
```

CachedThreadPool

缓存线程池。没有核心线程，普通线程数量为Integer.MAX_VALUE（可以理解为无限），线程闲置60s后回收，任务队列使用SynchronousQueue这种无容量的同步队列。适用于任务量大但耗时低的场景。

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

使用示例：

```
public class CachedThreadPoolTest {
    public static void main(String args[]) {
        ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
        cachedThreadPool.execute(new Runnable(){
            public void run()
                { System.out.println(Thread.currentThread().getName() + "--->运行
                ");
            }
        });
        cachedThreadPool.shutdown();
    }
}
// 输出：
// pool-1-thread-1--->运行
```

最后总结一下Executors封装线程池，每种方式用的是哪种队列：

- 固定容量线程池FixedThreadPool：LinkedBlockingQueue无界队
- 列 单线程线程池SingleThreadExecutor：LinkedBlockingQueue无
- 界队列定时线程池ScheduledThreadPool：DelayQueue延迟队列
- 缓存线程池CachedThreadPool：SynchronousQueue同步

队列稍微解读一下：

- FixedThreadPool：构造函数的keepAliveTime=0，然后核心线程个数和最大线程个数都限制死了，所以虽然用的是LinkedBlockingQueue无界队列，但是其实不会用到无界的特性，其实是当有界来用。
- SingleThreadExecutor：核心线程和最大线程都是1，keepAliveTime=0，这个也是拿LinkedBlockingQueue无界队列当有界队列使用。
- ScheduledThreadPool：因为需要延时，这个使用DelayQueue延迟队列就没有任何毛病。
- CachedThreadPool：没有核心线程，普通线程设置成最大值，用的SynchronousQueue没有任何容量，这个不太理解，后面再研究一下。

总结

这篇文章其实是我刚开始接触线程池时找到了，感觉非常经典，刚好现在需要系统学习Java并发编程，就把这篇文章重新整理一下，然后有些地方加入自己的解读，希望对大家有所帮助。

源文链接：https://blog.csdn.net/mu_wind/article/details/113806680

📚 第 8 章：多线程实战

> 主要基于小米最近的多线程项目，抽离出里面的多线程实例。

前言

Java多线程的学习，也有大半个月了，从开始学习Java多线程时，就给自己定了一个小目标，希望能写一个多线程的Demo，今天主要是兑现这个小目标。

这个多线程的示例，其实是结合最近小米的一个多线程异步任务的项目，我把里面涉及到多线程的代码抽离出来，然后进行一定的改造，之所以不自己重写一个，一方面是自己能力还不够，另一方面是想学习现在项目中多线程的实现姿势，至少这个示例是实际项目中应用的。先学习别人怎么造轮子，后面就知道自己怎么去造轮子了。

业务需求

做这个多线程异步任务，主要是因为我们有很多永动的异步任务，什么是永动呢？就是任务跑起来后，需要一直跑下去，比如消息Push任务，因为一直有消息过来，所以需要一直去消费DB中的未推送消息，就需要整一个Push的永动异步任务。

我们的需求其实不难，简单总结一下：

1. 能同时执行多个永动的异步任务；
2. 每个异步任务，支持开多个线程去消费这个任务的数据；
3. 支持永动异步任务的优雅关闭，即关闭后，需要把所有的数据消费完毕后，再关闭。

完成上面的需求，需要注意几个点：

1. 每个永动任务，可以开一个线程去执行；
2. 每个子任务，因为需要支持并发，需要用线程池控制；
3. 永动任务的关闭，需要通知子任务的并发线程，并支持永动任务和并发子任务的优雅关闭。

项目示例

线程池

对于子任务，需要支持并发，如果每个并发都开一个线程，用完就关闭，对资源消耗太大，所以引入线程池：

```
public class TaskProcessUtil {
    // 每个任务，都有自己单独的线程池
    private static Map<String, ExecutorService> executors = new ConcurrentHashMap<>();

    // 初始化一个线程池
    private static ExecutorService init(String poolName, int poolSize) { return new
        ThreadPoolExecutor(poolSize, poolSize,
                0L, TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<Runnable>(),
                new ThreadFactoryBuilder().setNameFormat("Pool-" + poolName).setDaemon(false).build(),
                new ThreadPoolExecutor.CallerRunsPolicy());
    }

    // 获取线程池
    public static ExecutorService getOrInitExecutors(String poolName,int poolSize) { ExecutorService
        executorService = executors.get(poolName);
        if (null == executorService) { synchronized
            (TaskProcessUtil.class) {
                executorService = executors.get(poolName); if (null ==
                executorService) {
                    executorService = init(poolName, poolSize); executors.put(poolName,
                    executorService);
                }
            }
        }
        return executorService;
    }
```

```
    // 回收线程资源
    public static void releaseExecutors(String poolName) { ExecutorService
        executorService = executors.remove(poolName); if (executorService != null) {
            executorService.shutdown();
        }
    }
```

```
}
```

这是一个线程池的工具类，这里初始化线程池和回收线程资源很简单，我们主要讨论获取线程池。获取线程池可能会存在并发情况，所以需要加一个synchronized锁，然后锁住后，需要对executorService进行二次判空校验，这个和Java单例的实现很像，具体可参考《【设计模式系列5】单例模式》这篇文章。

单个任务

为了更好讲解单个任务的实现方式，我们的任务主要就是把Cat的数据打印出来，Cat定义如下：

```
@Data
@Service
public class Cat {
    private String catName;
    public Cat setCatName(String name)
        { this.catName = name;
        return this;
    }
}
```

单个任务主要包括以下功能：

- 获取永动任务数据：这里一般都是扫描DB，我直接就简单用queryData()代替。
- 多线程执行任务：需要把数据拆分成4份，然后分别由多线程并发执行，这里可以通过线程池支持；
- 永动任务优雅停机：当外面通知任务需要停机，需要执行完剩余任务数据，并回收线程资源，退出任务；
- 永动执行：如果未收到停机命令，任务需要一直执行下去。

直接看代码：

```
public class ChildTask {

    private final int POOL_SIZE = 3; // 线程池大小
    private final int SPLIT_SIZE = 4; // 数据拆分大小 private String
    taskName;

    // 接收ivm关闭信号，实现优雅停机

    public ChildTask(String taskName) {
        this.taskName = taskName;
    }

    // 程序执行入口
    public void doExecute() { int i =
        0; while(true) {
            System.out.println(taskName + ":Cycle-" + i + "-Begin");
            // 获取数据
            List<Cat> datas = queryData();
            // 处理数据
            taskExecute(datas);
            System.out.println(taskName + ":Cycle-" + i + "-End"); if (terminal) {
                // 只有应用关闭，才会走到这里，用于实现优雅的下线
                break;
            }
```

```
        // 回收线程池资源
        TaskProcessUtil.releaseExecutors(taskName);
    }

    // 优雅停机
    public void terminal() {
        // 关机
        terminal = true;
        System.out.println(taskName + " shut down");
    }

    // 处理数据
    private void doProcessData(List<Cat> datas, CountDownLatch latch) { try {
                for (Cat cat : datas) {
                    System.out.println(taskName + ":" + cat.toString() + ",ThreadName:" +
Thread.currentThread().getName());
                    Thread.sleep(1000L);
                }
        } catch (Exception e) { System.out.println(e.getStackTrace());
        } finally {
            if (latch != null) { latch.countDown();
            }
        }
    }

    // 处理单个任务数据
    private void taskExecute(List<Cat> sourceDatas) { if
        (CollectionUtils.isEmpty(sourceDatas)) {
            return;
        }
        // 将数据拆成4份
        List<List<Cat>> splitDatas = Lists.partition(sourceDatas, SPLIT_SIZE); final CountDownLatch
        latch = new CountDownLatch(splitDatas.size());

        // 并发处理拆分的数据，共用一个线程池
        for (final List<Cat> datas : splitDatas) { ExecutorService
            executorService =
TaskProcessUtil.getOrInitExecutors(taskName, POOL_SIZE); executorService.submit(new Runnable() {
```

```
            @Override
            public void run() { doProcessData(datas,
                    latch);
            }
        });
    }
```

```
            try {
                latch.await();
            } catch (Exception e)
                { System.out.println(e.getStackTrace());
            }
        }

        //获取永动任务数据
        private List<Cat> queryData() { List<Cat>
            datas = new ArrayList<>(); for (int
            i = 0; i < 5; i ++) {
                datas.add(new Cat().setCatName("罗小  " + i));
            }
            return datas;
    }
```

简单解释一下：

- queryData：用于获取数据，实际应用中其实是需要把queryData定为抽象方法，然后由各个任务实现自己的方法。
- doProcessData：数据处理逻辑，实际应用中其实是需要把doProcessData定为抽象方法，然后由各个任务实现自己的方法。
- taskExecute：将数据拆分成4份，获取该任务的线程池，并交给线程池并发执行，然后通过latch.await()阻塞。当这4份数据都执行成功后，阻塞结束，该方法才返回。
- terminal：仅用于接受停机命令，这里该变量定义为volatile，所以多线程内存可见，详见《【Java并发编程系列2】volatile》；
- doExecute：程序执行入口，封装了每个任务执行的流程，当terminal=true时，先执行完任务数据，然后回收线程池，最后退出。

任务入口

直接上代码：

```
public class LoopTask {
    private List<ChildTask> childTasks;
    public void initLoopTask() {
        childTasks = new ArrayList();
        childTasks.add(new ChildTask("childTask1"));
        childTasks.add(new ChildTask("childTask2"));
        for (final ChildTask childTask : childTasks) {
            new Thread(new Runnable()
                { @Override
                public void run()
                    { childTask.doExecute();
                }
            }).start();
        }
    }
```

```
    public void shutdownLoopTask() {
        if (!CollectionUtils.isEmpty(childTasks))
            { for (ChildTask childTask : childTasks)
            {
                childTask.terminal();
            }
        }
    }
    public static void main(String args[]) throws
        Exception{ LoopTask loopTask = new LoopTask();
        loopTask.initLoopTask();
        Thread.sleep(5000L);
        loopTask.shutdownLoopTask();
}
```

每个任务都开一个单独的Thread，这里我初始化了2个永动任务，分别为childTask1和childTask2，然后分别执行，后面Sleep了5秒后，再关闭任务，我们可以看看是否可以按照我们的预期优雅退出。

结果分析

执行结果如下：

```
childTask1:Cycle-0-Begin
childTask2:Cycle-0-Begin
childTask1:Cat(catName=罗小黑0),ThreadName:Pool-childTask1
childTask1:Cat(catName=  罗  小  黑  4),ThreadName:Pool-childTask1
childTask2:Cat(catName=  罗  小  黑  4),ThreadName:Pool-childTask2
childTask2:Cat(catName=  罗  小  黑  0),ThreadName:Pool-childTask2
childTask1:Cat(catName=  罗  小  黑  1),ThreadName:Pool-childTask1
childTask2:Cat(catName=  罗  小  黑  1),ThreadName:Pool-childTask2
childTask2:Cat(catName=  罗  小  黑  2),ThreadName:Pool-childTask2
childTask1:Cat(catName=  罗  小  黑  2),ThreadName:Pool-childTask1
childTask2:Cat(catName=  罗  小  黑  3),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑3),ThreadName:Pool-childTask1
childTask2:Cycle-0-End
childTask2:Cycle-1-Begin
childTask1:Cycle-0-End
childTask1:Cycle-1-Begin
childTask2:Cat(catName=罗小黑0),ThreadName:Pool-childTask2
childTask2:Cat(catName=  罗  小  黑  4),ThreadName:Pool-childTask2
childTask1:Cat(catName=  罗  小  黑  4),ThreadName:Pool-childTask1
childTask1:Cat(catName=  罗  小  黑  0),ThreadName:Pool-childTask1
childTask1 shut down
childTask2 shut down
childTask2:Cat(catName=  罗  小  黑  1),ThreadName:Pool-childTask2
```

childTask1:Cat(catName= 罗 小 黑 1),ThreadName:Pool-childTask1
childTask1:Cat(catName= 罗 小 黑 2),ThreadName:Pool-childTask1
childTask2:Cat(catName= 罗 小 黑 2),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑3),ThreadName:Pool-childTask1

```
childTask2:Cat(catName=罗小黑 3),ThreadName:Pool-childTask2
childTask1:Cycle-1-End
childTask2:Cycle-1-End
```

输出数据中，“Pool-childTask”是线程池名称，“childTask”是任务名称，“Cat(catName=罗小黑)”是执行的结果，“childTask shut down”是关闭标记，“childTask:Cycle-X-Begin”和“childTask:Cycle-X-End”是每一轮循环的开始和结束标记。

我们分析一下执行结果：childTask1和childTask2分别执行，在第一轮循环中都正常输出了5条罗小黑数据，第二轮执行过程中，我启动了关闭指令，这次第二轮执行没有直接停止，而是先执行完任务中的数据，再执行退出，所以完全符合我们的优雅退出结论。

结语

这其实是一个比较经典的线程池使用示例，是我们公司的一位同事写的，感觉整个流程没有毛病，实现的也非常优雅，非常值得我学习的。

然后学习Java多线程的过程中，我感觉我目前的掌握速度还算是比较快的，从Java内存模型、到Java多线程的基本知识和常用工具，到最后的多线程实战，一共8篇文章，真的是可以让你从Java小白到能写出比较健壮的多线程程序。

其实之前学习语言或者技术，更多是偏向看一些八股文，其实八股文要看，更重要的是自己实践，需要多写，所以之前的文章很多是纯理论，现在更多是理论和实战相结合，那怕是看到网上的一些示例，我都会Copy下来，让程序跑一遍才安心。

Java多线程部分，后面打算再写1-2篇文章，这个系列就先暂停，因为我的目标是把Java生态的相关技术都学完，所以先尽快吃一遍，等全部学习完后，再重点学习更深入的知识。

📚 第 9 章：锁

> 主要讲解Java中常见的锁。

前言

并发编程系列应该快接近尾声，锁可能是这个系列的最后一篇，重要的基本知识应该都涵盖了。然后对于书籍《Java并发编程实战》，最后面的几章，我也只看了锁的部分，这篇文章主要是对该书中锁的内容进行一个简单的总结。

死锁

> 死锁是指一组互相竞争资源的线程因互相等待，导致“永久”阻塞的现象。

锁顺序死锁

我们先看一个死锁的示例，我们先定义个BankAccount对象，来存储基本信息，代码如下：

```
public class BankAccount
    { private int id;
    private double balance;
    private String password;
    public int getId() {
        return id;
    }
    public void setId(int id)
        { this.id = id;
    }
    public double getBalance()
        { return balance;
    }
    public void setBalance(double balance)
        { this.balance = balance;
    }
}
```

接下来，我们使用细粒度锁来尝试完成转账操作：

```
public class BankTransferDemo {
    public void transfer(BankAccount sourceAccount, BankAccount targetAccount, double
amount) {
        synchronized(sourceAccount)
            { synchronized(targetAccount) {
                if (sourceAccount.getBalance() > amount)
                    { System.out.println("Start transfer.");
                    sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                    targetAccount.setBalance(targetAccount.getBalance() + amount);
                }
            }
        }
    }
}
```

如果进行下述调用，就会产生死锁：

```
transfer(myAccount, yourAccount, 10);
transfer(yourAccount, myAccount, 10);
```

如果执行顺序不当，那么A可能获取myAccount的锁并等待yourAccount的锁，然而B此时持有yourAccount的锁，并正在等待myAccount的锁。

通过顺序来避免死锁

由于我们无法控制参数的顺序，如果要解决这个问题，必须定义锁的顺序，并在整个应用程序中按照这个顺序来获取锁。我们可以通过Object.hashCode返回的值，来定义锁的顺序：

```
public class BankTransferDemo {

    private static final Object tieLock = new Object();

    public void transfer(BankAccount sourceAccount, BankAccount targetAccount, double amount) {

        int sourceHash = System.identityHashCode(sourceAccount); int
        targetHash = System.identityHashCode(targetAccount);

        if (sourceHash < targetHash) { synchronized(sourceAccount)
            {
                synchronized(targetAccount) {
                    if (sourceAccount.getBalance() > amount)
                        { sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                        targetAccount.setBalance(targetAccount.getBalance() + amount);
                    }
                }
            }
        } else if (sourceHash > targetHash)
            { synchronized(targetAccount) {
                synchronized(sourceAccount) {
                    if (sourceAccount.getBalance() > amount)
                        { sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                        targetAccount.setBalance(targetAccount.getBalance() + amount);
                    }
                }
            }
```

```
        } else {
            synchronized (tieLock) { synchronized(targetAccount){
                synchronized(sourceAccount) {
                    if (sourceAccount.getBalance() > amount)
                        { sourceAccount.setBalance(sourceAccount.getBalance() -

amount);                        targetAccount.setBalance(targetAccount.getBalance() +

amount);
```

```
                                }
                            }
                        }
                    }
                }
            }
        }
```

无论你入参怎么变化，通过hash值的大小，我们永远是先锁住hash值小的数据，再锁hash值大的数据，这样就保证的锁的顺序。

但是在极少数情况下，两个对象的Hash值相同，如果顺序错了，仍可能导致死锁，所以在获取两个锁之前，使用"加时赛（Tie-Breaking）"锁，保证每次只有一个线程以未知的顺序获取到该锁。但是如果程序经常出现Hash冲突的情况，这里会成为并发的瓶颈，因为final变量是内存可见，会让所有的线程都阻塞到该锁上，不过这种概率会很低。

在协作对象之间发生死锁

这里我就只简单说明一下，就是有两个对象A和B，A.action_A1()会调用B中的方法action_B1()，同时B.action_B2()会调用A中的方法action_A2()，由于这四个方法action_A1()、action_A2()、action_B1()、action_B2()都通过 synchronized加锁，我们知道都通过synchronized在方法上加的是对象锁，所以可能存在A调用B的方法时，B也正在调用A的方法，导致互相等待出现死锁的情况。

> 具体的示例，大家可以参考《Java并发编程实战》书籍第174页的内容。

ReentrantLock

使用方法

在协调对象的访问时可以使用的机制只有synchronized和volatile，Java 5.0增加了一种新的机制：ReentrantLock。ReentrantLock并不是一种替代内置锁的方法，而是当内置锁机制不适用时，作为一种可选的高级功能。

下面看一个简单的示例：

```
Lock lock = new ReentrantLock();
//...
lock.lock();
try {
    // ...
} finally {
    lock.unlock();
}
```

除了上述不可替换synchronized的原因，就是需要手动通过lock.unlock()释放该锁，如果忘记释放，那将是个非常严重的问题。

通过tryLock避免顺序死锁

还是沿用上面的死锁示例，我们通过tryLock()进行简单改造：

```
public boolean transfer(BankAccount sourceAccount, BankAccount targetAccount, double
amount, long timeout, TimeUnit unit) {
    long stopTime = System.nanoTime() + unit.toNanos(timeout);
    while (true) {
        if (sourceAccount.lock.tryLock())
            { try {
                if (targetAccount.lock.tryLock())
                    { try {
                        if (sourceAccount.getBalance() > amount)
                            { sourceAccount.setBalance(sourceAccount.getBalance() -
amount);
                            targetAccount.setBalance(targetAccount.getBalance() +
amount);
                        }
                    } finally {
                        targetAccount.lock.unlock();
                    }
                }
            } finally {
                sourceAccount.lock.unlock();
            }
        }
        if (System.nanoTime() < stopTime)
            { return false;
        }
        // Sleep一会…
    }
}
```

我们先尝试获取sourceAccount的锁，如果获取成功，再尝试获取targetAccount的锁，如果获取失败，我们就释放sourceAccount的锁，避免长期占用sourceAccount锁而导致的死锁问题。

带有时间限制的加锁

我们也可以对tryLock()指定超时时间，如果等待的时间超时，不会一直等待，直接执行后续的逻辑：

```
long stopTime = System.nanoTime() + unit.toNanos(timeout);
while (true) {
    long nanosToLock = unit.toNanos(timeout);
    if (sourceAccount.lock.tryLock(nanosToLock, TimeUnit.NANOSECONDS))
        { try {
            // 省略…
        } finally {
            sourceAccount.lock.unlock();
        }
    }
    if (System.nanoTime() < stopTime) {
```

```
            return false;
        }
        // Sleep一会…
    }
```

synchronized vs ReentrantLock

ReentrantLock在加锁和内存上提供的语义与内置锁相同，此外它还提供了一些其他的功能，包括定时的锁等待、可中断的锁等待、公平性，以及实现非块结构的加锁。ReentrantLock的性能上似乎优于内置锁，其中在Java 6.0中略有胜出，而在Java 5.0中则远远胜出，那是否我们都用ReentrantLock，直接废弃掉synchronized么？

与显示锁相比，内置锁仍然具有很大的优势。内置锁为许多开发人员所熟悉，并且简洁紧凑。ReentrantLock的危险性比同步机制要高，如果忘记在finally块中调用unlock，那么虽然代码表面上看起来能正常运行，但实际上已经埋下了一颗定时炸弹，并很有可能伤及其它代码。仅当内置锁不能满足需求时，才可以考虑使用ReentrantLock。

> 使用原则：ReentrantLock可以作为一种高级工具，当需要一些高级功能，比如可定时的、可轮训与可中断的锁获取操作，公平队列，以及非块结构的锁。否则，还是优先使用synchronized。

然后有一点需要重点强调一下，synchronized和ReentrantLock都是可重入锁，可重入的概念，请参考文章《【Java并发编程系列3】synchronized》。

读写锁

读写锁的使用和Go中的读写锁用法一致，先看读写锁接口定义：

```
public interface ReadWriteLock {
    /**
     * 返回读锁
     */
    Lock readLock();
    /**
     * 返回写锁
     */
    Lock writeLock();
```

ReadWriteLock管理一组锁，一个是只读的锁，一个是写锁。
Java并发库中ReetrantReadWriteLock实现了ReadWriteLock接口并添加了可重入的特性。

```
public class ReadWriteMap<K,V>
    { private final Map<K,V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();
    public ReadWriteMap(Map<K,V> map) {
        this.map = map;
    }
```

下面看一下使用姿势：

```
    public V put(K key, V value) {
        w.lock();
        try {
            return map.put(key,value);
        } finally {
            w.unlock();
        }
    }
    public V get(Object key)
        { r.lock();
        try {
            return map.get(key);
        } finally {
            r.unlock();
        }
    }
}
```

这样可以多个线程去读取数据，但是只有一个线程可以去写数据，然后读和写不能同时进行。

其它

自旋锁

这个仅作为扩展知识，觉得有些意思，就写进来，那么什么是自旋锁呢？

> 自旋锁的定义：当一个线程尝试去获取某一把锁的时候，如果这个锁此时已经被别人获取(占用)，那么此线程就无法获取到这把锁，该线程将会等待，间隔一段时间后会再次尝试获取。这种采用循环加锁 -> 等待的机制被称为自旋锁(spinlock)。

自旋锁的原理

自旋锁的原理比较简单，如果持有锁的线程能在短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞状态，它们只需要等一等(自旋)，等到持有锁的线程释放锁之后即可获取，这样就避免了用户进程和内核切换的消耗。

因为自旋锁避免了操作系统进程调度和线程切换，所以自旋锁通常适用在时间比较短的情况下。由于这个原因，操作系统的内核经常使用自旋锁。但是，如果长时间上锁的话，自旋锁会非常耗费性能，它阻止了其他线程的运行和调度。线程持有锁的时间越长，则持有该锁的线程将被 OS(Operating System) 调度程序中断的风险越大。如果发生中断情况，那么其他线程将保持旋转状态(反复尝试获取锁)，而持有该锁的线程并不打算释放锁，这样导致的是结果是无限期推迟，直到持有锁的线程可以完成并释放它为止。

解决上面这种情况一个很好的方式是给自旋锁设定一个自旋时间，等时间一到立即释放自旋锁。

自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用 cpu 做无用功，占着 XX 不 XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要 cpu 的线程又不能获取到 cpu，造成 cpu 的浪费。所以这种情况下我们要关闭自旋锁。

自旋锁的实现

```
public class SpinLockTest {
    private AtomicBoolean available = new AtomicBoolean(false);
    public void lock(){
        // 循环检测尝试获取锁
        while (!tryLock()){
            // doSomething...
        }
    }
    public boolean tryLock(){
        // 尝试获取锁，成功返回true，失败返回false
        return available.compareAndSet(false,true);
    }
    public void
        unLock(){ if(!available.compareAndSet(true,false)
        ){
            throw new RuntimeException("释放锁失败");
        }
    }
}
```

这种简单的自旋锁有一个问题：无法保证多线程竞争的公平性。对于上面的 SpinlockTest，当多个线程想要获取锁时，谁最先将available设为false谁就能最先获得锁，这可能会造成某些线程一直都未获取到锁造成线程饥饿。就像我们下课后蜂拥的跑向食堂，下班后蜂拥地挤向地铁，通常我们会采取排队的方式解决这样的问题，类似地，我们把这种锁叫排队自旋锁(QueuedSpinlock)。计算机科学家们使用了各种方式来实现排队自旋锁，如 TicketLock， MCSLock，CLHLock。

锁的特性

Java 中的锁有很多，可以按照不同的功能、种类进行分类，下面是我对 Java 中一些常用锁的分类，包括一些基本的概述：

- 从线程是否需要对资源加锁可以分为“悲观锁”和“乐观锁”
- 从资源已被锁定，线程是否阻塞可以分为“自旋锁”
- 从多个线程并发访问资源，也就是Synchronized可以分为无锁、偏向锁、轻量级锁和重量级
- 锁从锁的公平性进行区分，可以分为“公平锁”和“非公平锁”
- 从根据锁是否重复获取可以分为“可重入锁”和“不可重入锁”从
-

那个多个线程能否获取同一把锁分为“共享锁”和“排他锁”

具体可以参考文章《不懂什么是锁？看看这篇你就明白了》：https://mp.weixin.qq.com/s?__biz=MzkwMDE1MzkwNQ==&mid=2247496038&idx=1&sn=10b96d79a1ff5a24c49523cdd2be43a4&chksm=c04ae638f73d6f2e1ead614f2452ebaeab26cf77b095d6f634654699a1084365e7f5cf6ca4f9&token=1816689916&lang=zh_CN#rd

总结

这篇文章主要讲解了死锁，死锁的解决方式，ReentrantLock，ReentrantLock和内置锁synchronized的比较，最后也讲解了自旋锁，前面内容是核心部分，自旋锁仅仅作为扩展知识。

锁的内容目前总结完了，所以Java并发编程系列我就先学到这个地方，后续如果学习到了其它Java并发知识，会持续维护这个系列。之前给自己定了Flag，今年需要把Java的基础知识都学完，所以我下个系列将会是Spring，希望和我一样的Java小白，能一起共同进步。
