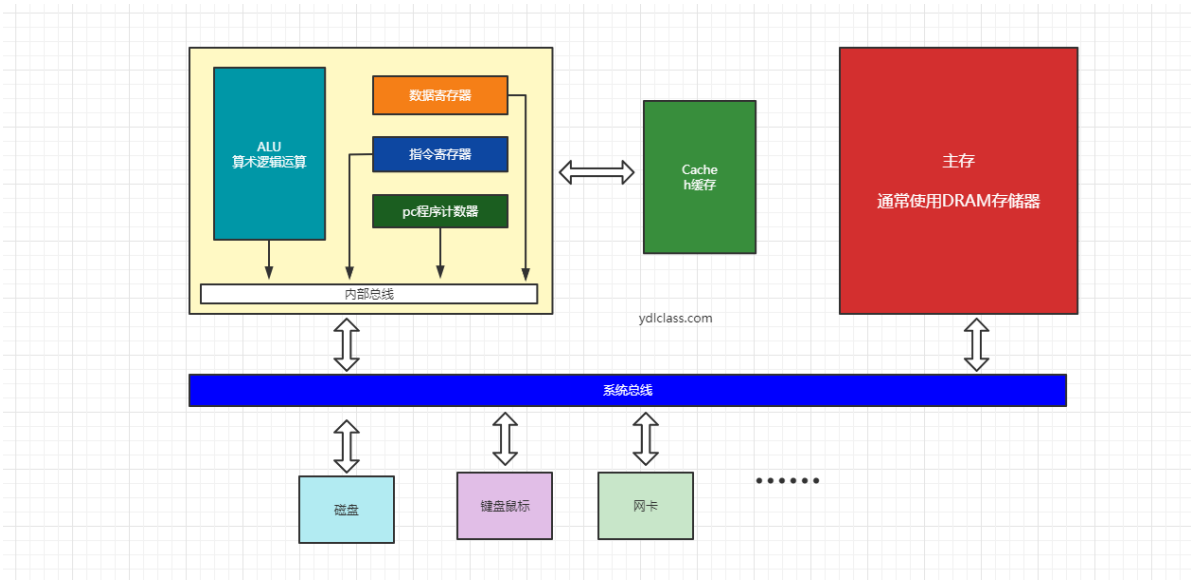
第10章 JAVA多线程入门

我们要学习多线程，就不可避免的需要学习一些计算机组成的一些知识，特别是cpu和内存相关的一些知识，我用一张简单的图来描述目前的计算机系统的基本组成：



为了完成特定任务，用某种语言编写的一个软件就是一个【程序】，程序要想运行必须加载到内存中执行。而执行程序的时候，又需要实时的将程序指令加载到cpu的指令寄存器中执行，执行过程中产生的数据要加载到数据寄存器当中。ALU负责进行算术逻辑运算的操作，比如算术运算、逻辑运算、位移运算。

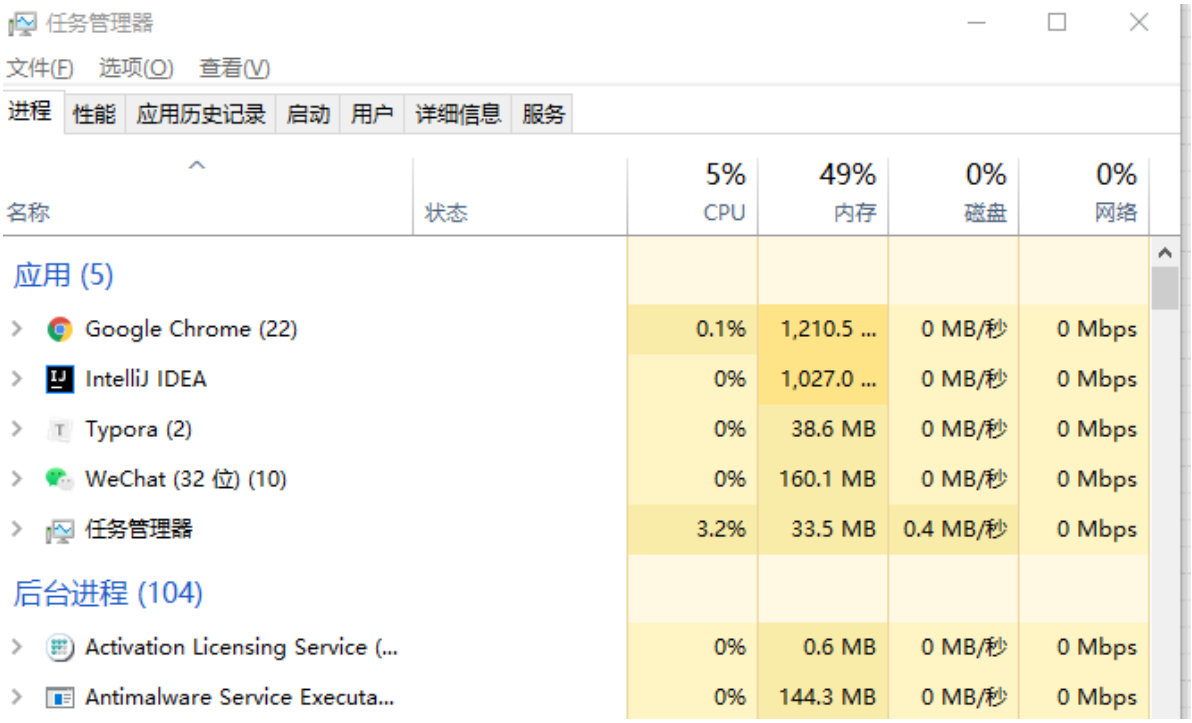
系统总线（英语：System Bus）是连接计算机系统的主要组件，这个技术的开发是用来降低成本和促进模块化。

#一、进程和线程

#1、进程

一个正在执行中的程序就是一个进程，系统会为这个进程发配独立的【内存资源】。进程是程序的一次执行过程，它有自己独立的生命周期，它会在启动程序时产生，运行程序时存在，关闭程序时消亡。

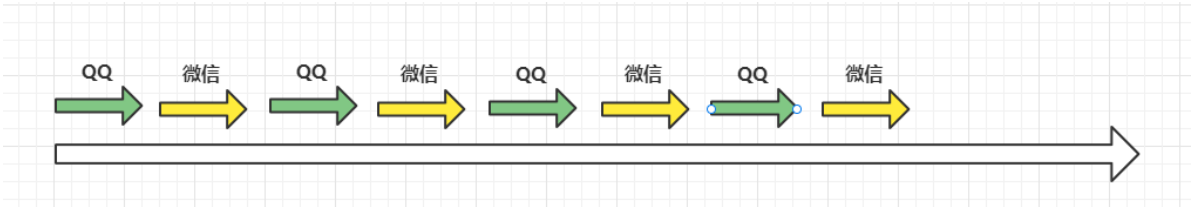
例如：正在运行的 QQ、IDE、浏览器就是进程。



其实，谈及计算机时，永远不能简单的避开计算机的发展史。

最原始的计算机就是单进程的，同一时间只能执行一个进程，我们可以把现在的【计算器】当做最原始的计算机，同一时间只能执行一段代码。比如我们要计算一个账本的总账，只能一个数字一个数字的相加，而且，在这其中你还不能做其他的事情。

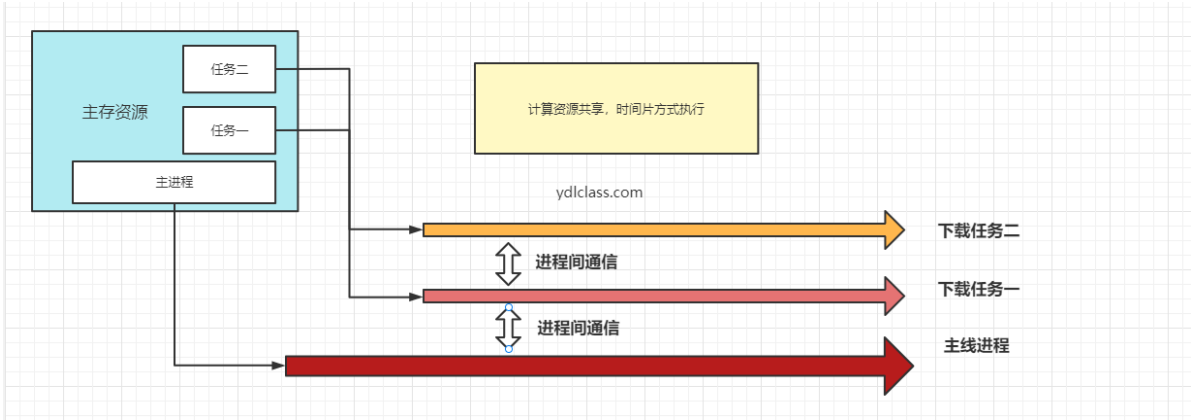
但是随着计算机的发展，计算任务的不断提升，单个进程的方式人们就很难接受了，与此同时cpu的计算能力也大幅提升，于是就产生了按照时间线交替执行不同进程的方式。 两个进程交替执行，每个执行一点点时间，在感觉上就像同时执行两个进程了。



当然，我们还会有疑问?

如果一个进程有多个任务怎么办，比如我们使用浏览器同时下载八个小电影，一种方式是一个一个下载，一个完了下一个开始，另一种方式就是同时开始，最后一个下载完成结束。

第一种就是我们的串行执行，没什么好说的，第二种就需要其他的解决方案了，给每一个下载任务分配一个进程可以吗？每一个进程会分配独立的内存资源，原则上是可以的。

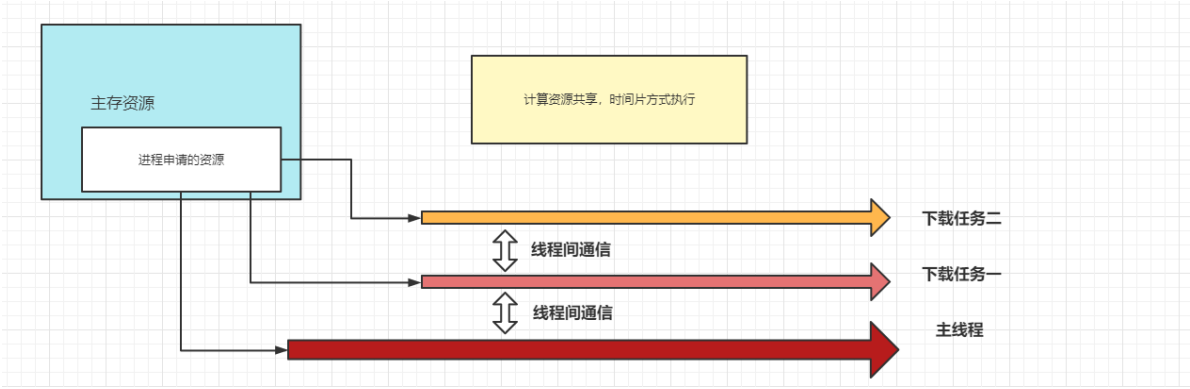


如果为每一个任务分配单独的进程去执行，进程间的通信就会不可避免，比如某一个下载任务完成了肯定要通知浏览器啊，这样就会产生一个问题，微信的进程是不是能访问QQ的进程？病毒是不是就很容易操作你运行中的进程，修改你的数据了。所以，在计算机的设计当中就引入了线程的概念。

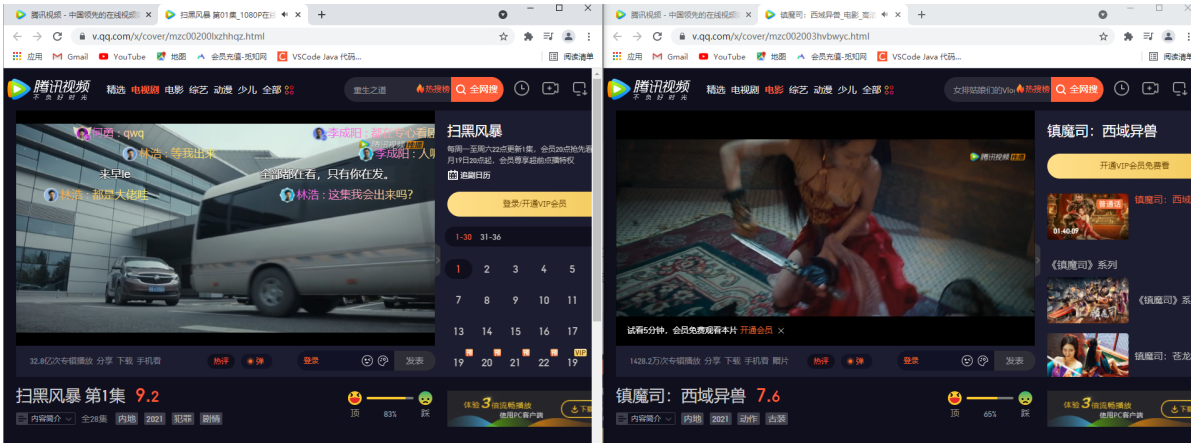
#2、线程

线程是由进程创建的，是进程的一个实体，是具体干活的人，一个进程可能有多个线程。线程不独立分配内存，而是共享进程的内存资源，线程可以共享cpu的计算资源。

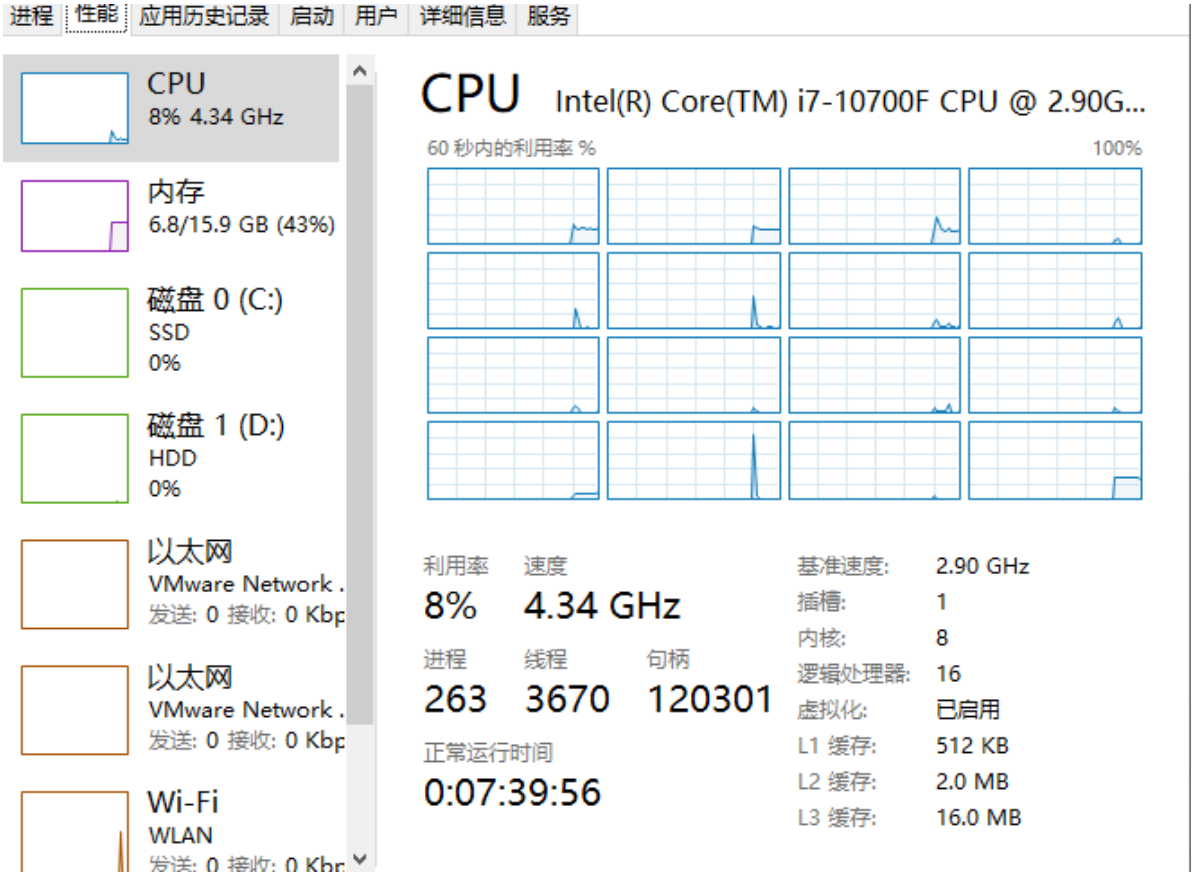
现在，进程更强调【内存资源的分配】，而线程更强调【计算资源的分配】。因为有了线程的概念，一个进程的线程就不能修改另一个线程的数据，隔离性更好，安全性更好。



我使用浏览器打开两个腾讯视频，他们可以同时播放视频，我一个浏览器可以同时下载很多个文件，谷歌浏览器本身就是一个进程，那播放两个视频或者下载多个文件就是不同线程在做的工作，否则，你一定是需要等待一个结束了，另一个才能开始。



我们可以在计算机的任务管理器中查看当前计算机的进程和线程数。



看我这个计算机哈：

大家有木有觉的奇怪，我这个电脑一颗cpu有八个核，八核又有16个逻辑处理器，也称8核16线程，这是什么意思呢。

理论上，一个核在一个时间点只能跑一个线程，但是这个cpu同一个时间能跑16个线程，他是一种什么样的结构呢？

咱们先了解一下这几个概念：

1. 物理CPU就是计算机上实际安装的CPU，就是主板上实际插入的CPU数量。
2. 物理CPU内核，每颗物理CPU可以有1个或者多个物理内核，通常每颗物理CPU的内核数都是固定的，单核CPU就是有1个物理内核，我这个电脑有八颗
3. 逻辑CPU，操作系统可以使用逻辑CPU来模拟真实CPU。在没有多核处理器的时候，一个物理CPU只能有一个物理内核，而现在有了多核技术，一个物理CPU可以有多个物理内核，可以把一个CPU当作多个CPU使用，没有开启超线程时，逻辑CPU的个数就是总的CPU物理内核数。然而开启超线程后，逻辑CPU的个数就是总的CPU物理内核数的两倍。 注：超线程（HT, Hyper-Threading）是【英特尔】研发的一种技术，于2002年发布。这个和硬件相关我们知道就行了。

#3、上下文切换

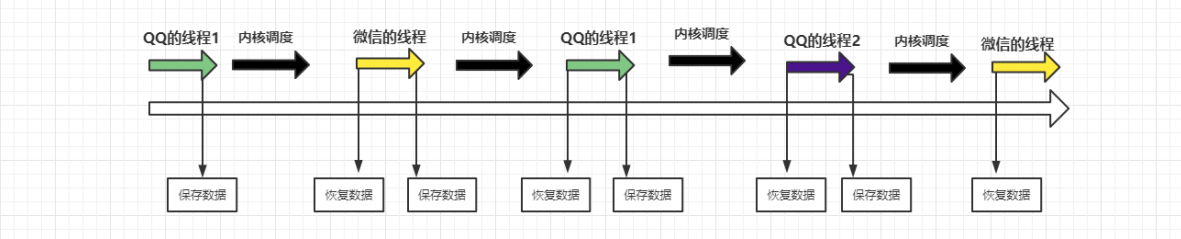
从任务管理器中我们可以看到，这台电脑上运行着263个进程，3670个线程，但是我只有16个逻辑内核，这足以证明对于每一个逻辑内核他在执行的过程当中也是按照时间片执行不同的线程的。

但是这里有几个问题：

1. 我们的进程可以直接创建、调度线程吗？QQ运行了一会说我累了，不想执行了，微信你来吧！这显然是不合理的。
2. QQ执行了一会，不执行了，那等其他线程执行完成之后，又轮上QQ了，QQ还能记得刚才运行到哪里了吗？

针对第一个问题，任何一个用户的线程是不允许调度其他的线程的，所有的线程调用都由一个大管家统一调度，这个大管家就是系统内核。

第二个问题，下一个执行时想要知道上一次的执行结果，就必须在上一次执行之后，讲运行时的数据进行保存，那么整个过程就出来了。



其中，用户线程执行的过程我们称之为【用户态】，内核调度的状态称之为【内核态】，每一个线程运行时产生的数据我们称之为【上下文】，线程的每次切换都需要进行用户态到内核态的来回切换，同时伴随着上下文的切换，是一个比较消耗资源的操作，所以一个计算机当中不是线程越多越好，线程如果太多也是有可能拖垮整个系统的。

#4、创建线程的方法

在java当中创建线程有三种基本方式：

（1） 继承Thread类重写run方法

> **步骤：**

- 定义类继承Thread;
- 重写Thread类中的run方法；（目的：将自定义代码存储在run方法，让线程运行）
- 调用线程的start方法：(该方法有两步：启动线程，调用run方法)

```
1  public class UseThread {
2      public static void main(String[] args) {
3          System.out.println(1);
4          new MyTask().start();
5          System.out.println(3);
6          try {
7              Thread.sleep(100);
8          } catch (InterruptedException e) {
9              e.printStackTrace();
10         }
11         System.out.println(4);
12     }
13
14     static class MyTask extends Thread{
15         @Override
16         public void run() {
17             System.out.println(2);
18         }
19     }
20 }
```

咱们猜一下：这个程序的输出结果是 1 2 3 4 吗?

（2） 实现Runnable接口

> **步骤：**

- 创建任务： 创建类实现Runnable接口
- 使用Thread 为这个任务分配线程

- 调用线程的start方法

```
public class UseRunnable {

    public static void main(String[] args) {
        System.out.println(1);
        //注意，这里new的是Thread
        new Thread(new Task()).start();
        System.out.println(3);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(4);
    }

    static class Task implements Runnable{
        public void run() {
            System.out.println(2);
        }
    }

}
```

（3）使用Lammbda表达式

```
public class UseRunnable {

    public static void main(String[] args) {
        System.out.println(1);
        //注意，这里new的是Thread
        new Thread(()-> System.out.println(2)).start();
        System.out.println(3);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(4);
    }

}
```

（4）有返回值的线程

```
public class UseCallable {

   public static void main(String[] args) throws ExecutionException,
InterruptedException {
        System.out.println(2);
        FutureTask<Integer> futureTask = new FutureTask<>(new Task());
        System.out.println(3);
        new Thread(futureTask).start();
        System.out.println(4);
        int result = futureTask.get();
        System.out.println(5);
```

```
        System.out.println(result);
        System.out.println(6);
    }

    static class Task implements Callable<Integer> {
        public Integer call() throws Exception {
            Thread.sleep(2000);
            return 1;
        }
    }
}
```

`futureTask.get();`这是一个阻塞的方法，意思就是，这个方法会一直等，主线程会一直等待，这个线程执行完成之后并有了返回值，才会继续执行。

#5、守护线程

Java提供两种类型的线程：`用户线程`和`守护程序线程`。守护线程旨在为用户线程提供服务，并且仅在用户线程运行时才需要。

> 守护线程的使用

守护线程对于后台支持任务非常有用，例如垃圾收集，释放未使用对象的内存以及从缓存中删除不需要的条目。大多数JVM线程都是守护线程。在比如qq等等聊天软件,主程序是非守护线程,而所有的聊天窗口是守护线程,当在聊天的过程中,直接关闭聊天应用程序时,聊天窗口也会随之关。包括word中我们在书写文字的时候，还有线程帮我们进行拼写检查，这都是守护线程。

> 创建守护线程

要将线程设置为守护线程，我们需要做的就是调用Thread.setDaemon（）。在这个例子中，我们将使用扩展Thread类的NewThread类：

```
NewThread daemonThread = new NewThread();
daemonThread.setDaemon(true);
daemonThread.start();
```

任何线程都继承创建它的线程的守护进程状态。由于主线程是用户线程，因此在main方法内创建的任何线程默认为用户线程。

```
public class Deamon {

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            int count = 10;
            Thread t2 = new Thread(() -> {
                while (true){
                    ThreadUtils.sleep(300);
                    System.out.println("我是个守护线程！");
                }
            });
            t2.setDaemon(true);
            t2.start();

            while (count >= 0){
                ThreadUtils.sleep(200);
                System.out.println("我是用户线程！");
```

```
                count--;
            }
            System.out.println("用户线程结束--------------------");
        });
        t1.setDaemon(true);
        t1.start();
    }

}
```

#6、 线程生命周期

生命周期可以通俗地理解为“从摇篮到坟墓”（Cradle-to-Grave）的整个过程。线程的生命周期包括从创建到终结的整个过程。

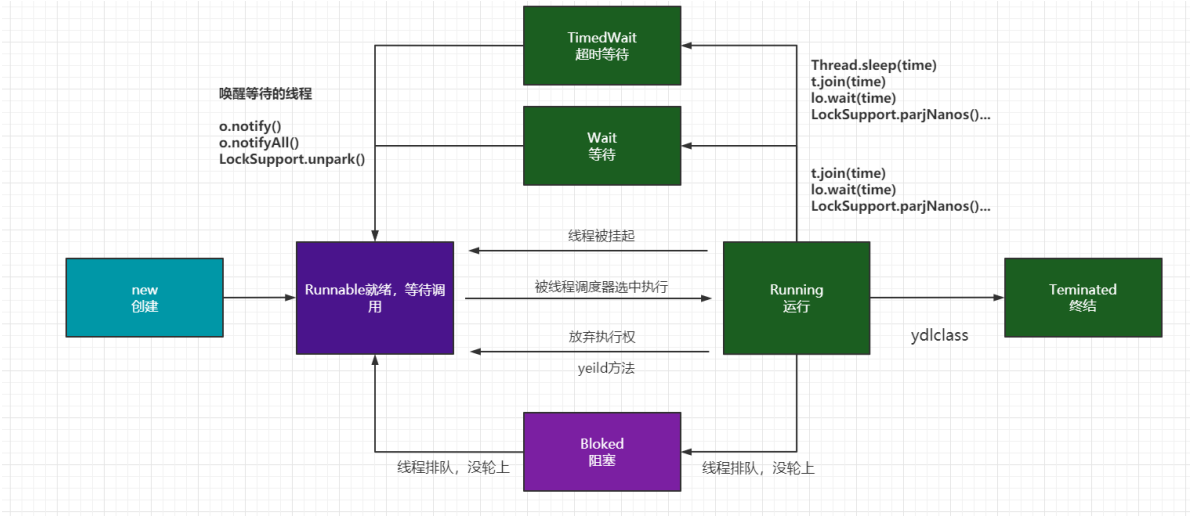
我们在Thread类中发现了一个内部枚举类，这个State就可以表示一个线程的生命周期：

```
public enum State {
        /**
         * Thread state for a thread which has not yet started.
         */
        NEW,

        /**
         * Thread state for a runnable thread.  A thread in the runnable
         * state is executing in the Java virtual machine but it may
         * be waiting for other resources from the operating system
         * such as processor.
         */
        RUNNABLE,

        BLOCKED,

        WAITING,

        TIMED_WAITING,

        TERMINATED;
    }
```

这个枚举类阐述了一个线程的生命周期中，总共有以下6种状态

状态	描述
【NEW】	这个状态主要是线程未被Thread.start()调用前的状态。
【RUNNABLE】	线程正在JVM中被执行，等待来自操作系统(如处理器)的调度。
【BLOCKED】	阻塞，因为某些原因不能立即执行需要挂起等待。
【WAITING】	无限期等待，由于线程调用了 `Object.wait(0)` ， `Thread.join(0)` 和 `LockSupport.park` 其中的一个方法，线程处于等待状态，其中调用 `wait` ， `join` 方法时未设置超时时间。
【TIMED_WAITING】	有限期等待， 线程等待一个指定的时间，比如线程调用了 `Object.wait(long)` ， `Thread.join(long)` ， `LockSupport.parkNanos` ， `LockSupport.parkUntil` 方法之后，线程的状态就会变成TIMED_WAITING
【TERMINATED】	终止的线程状态，线程已经完成执行。

等待和阻塞两个概念有点像，但是阻塞往往因为外部原因，需要等待，而等待一般是主动调用方法，发起主动等待的动作，等待还可以传入参数确定等待的时间。



咱们不妨研究一下上边提及的几个方法：

为了避免每次调用sleep方法都需要抛出异常：

我们定义一个工具类：

```
1   public class ThreadUtils {
2
3       public static void sleep(int i) {
4           try {
5               Thread.sleep(i);
6           } catch (InterruptedException e) {
7               e.printStackTrace();
8           }
9       }
10  }
```

> 目前我们可以学习一下join方法，他是这么用的：

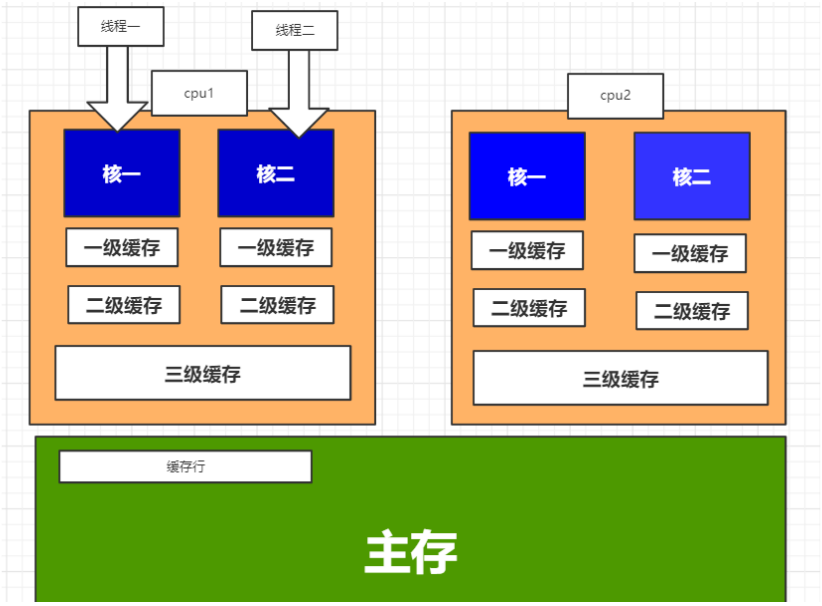
```java
public class Test {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i<10 ; i++) {
                ThreadUtils.sleep(10);
                System.out.println("这是线程1-----"+i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i<100 ; i++) {
                ThreadUtils.sleep(10);
                System.out.println("这是线程2-----"+i);
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("------------------------------------------------------");
    }
}
```

这个代码我们要分析【虚线出现的位置】，join方法的本意是阻塞主线程，直到t1线程和t2线程执行完毕后继续执行主线程

#二、线程安全的讨论

#1、CPU多核缓存架构

CPU缓存为了提高程序运行的性能，现代CPU在很多方面会对程序进行优化。CPU的处理速度是很快的，内存的速度次之，硬盘速度最慢。在CPU处理内存数据中，内存运行速度太慢，就会拖累CPU的速度。为了解决这样的问题，CPU设计了多级缓存策略。



> CPU分为三级缓存： 每个CPU都有L1,L2缓存，但是L3缓存是多核公用的。

> CPU查找数据的顺序为： CPU -> L1 -> L2 -> L3 -> 内存 -> 硬盘

从CPU到	大约需要的时间
主存	60~80纳秒
L3 cache	大约15纳秒
L2 cache	大约3纳秒
L1 cache	大约1纳秒
寄存器	大约0.3纳秒

进一步优化，CPU每次读取一个数据，并不是仅仅读取这个数据本身，而是会读取与它相邻的64个字节的数据，称之为【缓存行】，因为CPU认为，我使用了这个变量，很快就会使用与它相邻的数据，这是计算机的局部性原理。这样，就不需要每次都从主存中读取数据了。

这个其实很好理解：

```
1  public static void main(String[] args) {
2      int nums[] = new int[10];
3      for (int i = 0; i < nums.length; i++) {
4          System.out.println(nums[i]);
5      }
6  }
```

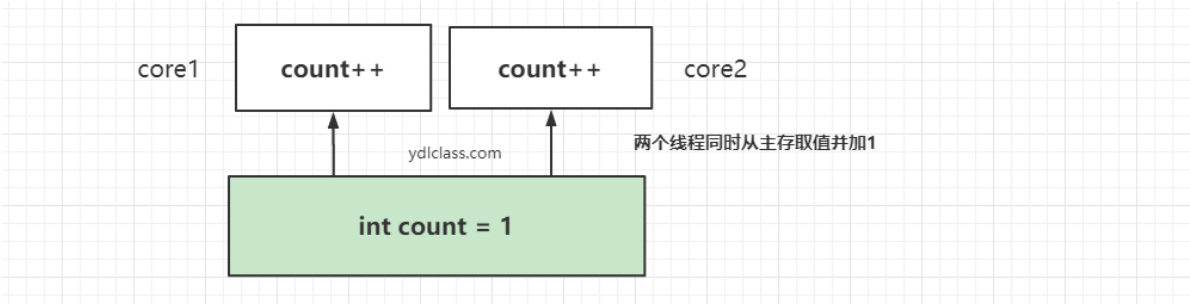
比如说这个代码，CPU读到nums[0]这后大概率就会读nums[1]，这就是局部性原理的体现，当然一个缓存行现在是64个字节，这是很多科学家调优的结果，如果设计的太小则难以命中，如果设计的大了则读取比较慢，这是目前的最优解。

这种多级缓存的结构下，会有什么问题呢？最经典的就是【可见性的问题】，可以简单的理解为，一个线程修改的值对其他线程可能不可见。

比如两个CPU读取了一个缓存行，缓存行里有两个变量，一个x一个y。第一颗CPU修改了x的数据，还没有刷回主存，此时第二颗CPU，从主存中读取了未修改的缓存行，而此时第一颗CPU修改的数据刷回主存，这时就出现，第二颗CPU读取的数据和主存不一致的情况。

为了解决数据不一致的问题，很多厂商提出了自己的解决方案，比如英特尔的MESI协议。

```
1  MESI协议规定每条缓存都有一个状态位，同时定义了一下四种状态：
2  修改态 (Modified) 此缓存被修改过，内容与住内存不同，为此缓存专有
3  专有态 (Exclusive) 此缓存与主内存一致，但是其他CPU中没有
4  共享态 (Shared) 此缓存与住内存一致，但也出现在其他缓存中。
5  无效态 (Invalid) 此缓存无效，需要从主内存中重新读取。在可见性的问题，当多个线程同时修改相同资源的
   时候，还会存在资源的争夺问题。
```



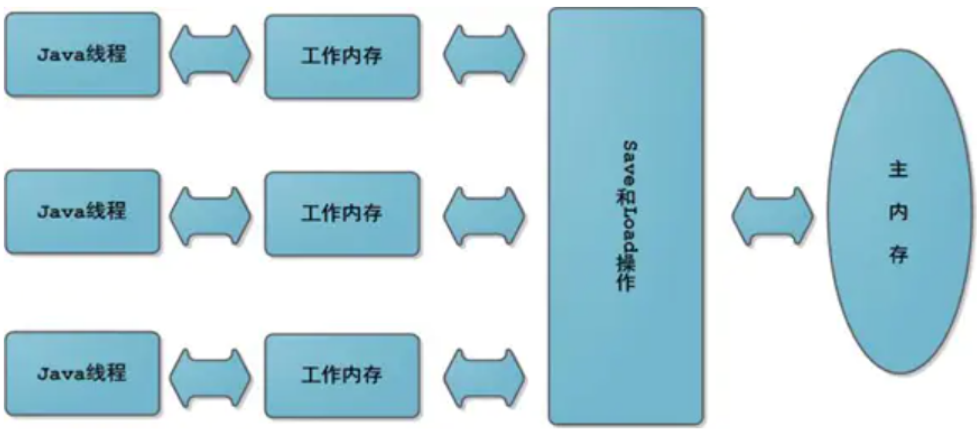
我们在执行这一段代码之后，有可能最后的结果是2。

除了增加高速缓存之外，为了使处理器内部的运算单元尽量被充分利用。处理器可能会对输入的代码进行【乱序执行】，优化处理器会在计算之后将乱序执行的结果【进行重组】，保证该结果与顺序执行的结果是一致的，但并不保证程序中各个语句的先后执行顺序与输入代码中的顺序一致。因此如果存在一个计算任务，依赖于另外一个依赖任务的中间，结果那么顺序性不能靠代码的先后顺序来保证。 Java虚拟机的即时编译器中也有【指令重排】的优化。

咱们可以举一个简单的例子来形象的说明一下，比如现在我们有这么一个需求，有四条指令，这四条指令分别是让四个人在四张纸上写下【新年快乐】四个字。但是在这个过程当中，有的人写的快，有的人写得慢，而如果我们非要按照新年快乐这四个顺序去执行这个工作的话，可能时间会浪费的多一点，那我们不妨让这四个人分别去写他们这四个字儿，我们等着这四个人最后一个写完了，然后再把这四个字组合在一起，我们就达到目的了，这样的乱序执行效率可能会更高一些。

#2、JMM-java内存模型

Java虚拟规范中曾经试图定义一种Java内存模型，来屏蔽各种硬件和操作系统的内存访问之间的差异，以实现让Java程序在各种平台上都能达到一致的内存访问效果。在此之前，主流程序语言直接使用物理内存和操作系统的内存模型，会由于不同平台的内存模型的差异，可能导致程序在一套平台上发挥完全正常，而在另一套平台上并发经常发生错误，所以在某种常见的场景下，必须针对平台来进行代码的编写。



这里的内存模型和我们的运行时数据是从不同的角度去分析java对内存的使用的。两者表达的含义和目的不同。在java内存模型当中一样会存在可见性和指令重排的问题。

#3、JMM模型当中存在的问题

- 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。

（1）指令重排

在指令重排中，有一个经典的as-if-serial语义，计算机会安装该语义对指令进行优化，其目的是不管怎么重排序（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。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，这些操作依然可能被编译器和处理器重排序。

```
{
    int a = 1;//1
    int b = 2;//2
    a + b;//3
}
```

1和3之间存在数据依赖关系，同时2和3之间也存在数据依赖关系。因此在最终执行的指令序列中，3不能被重排序到1和2的前面（3排到1和2的前面，程序的结果将会被改变）。但1和2之间没有数据依赖关系，编译器和处理器可以重排序1和2之间的执行顺序。 asif-serial语义使单线程下无需担心重排序的干扰，也无需担心内存可见性问题。

在单线程下，我们可以不用担心指令重排但是多线程下指令重排可以引发一些奇怪的问题。

我们用一个例子来证明指令重排序的存在：

```
public class OutOfOrderExecution {
    private static int x = 0, y = 0;
    private static int a = 0, b = 0;
    private static int count = 0;

    private static volatile int NUM = 0;

    public static void main(String[] args)
            throws InterruptedException {
        long start = System.currentTimeMillis();
        for (;;) {
            Thread t1 = new Thread(new Runnable() {
                @Override
                public void run() {
                    a = 1;
                    x = b;
                }
            });
            Thread t2 = new Thread(new Runnable() {
                @Override
                public void run() {
                    b = 1;
                    y = a;
                }
            });
            t1.start();
            t2.start();
            t1.join();
            t2.join();
            System.out.println("一共执行了：" + (count++) + "次");
            if(x==0 && y==0){
```

```
32                  long end = System.currentTimeMillis();
33                  System.out.println("耗时：+"+ (end-start) +"毫秒，(" + x + "," + y
    + ")");
34                  break;
35              }
36              a=0;b=0;x=0;y=0;
37          }
38      }
39  }
```

我们发现结果中绝大部分是感觉正确的，（0，1），但是也有（1，0），这两种结果都是正确的，一个是线程1先执行，一个是线程二先执行，其实只要次数足够多哪种情况都会有。

但是按道理绝对不会出现（0，0），因为出现零的情况一定是x = b; y = a; a = 1; b = 1;，如果出现了也就证明了我们的执行在执行的时候确实存在乱序。

执行顺序	结果
[a = 1] ---> [x = b] ---> [b = 1] ---> [y = a]	0,1
[b = 1] ---> [y = a]---> [a = 1] ---> [x = b]	1,0
[a = 1] ---> [b = 1] ---> [x = b] ---> [y = a]	1,1
[x = b] ---> [y = a] ---> [a = 1] ---> [b = 1]	0,0

事实上我们执行了90多万次，得到了（0，0）的结果

```
一共执行了：922394次
一共执行了：922395次
一共执行了：922396次
耗时：+112628毫秒，(0,0)
```

执行了13多万次，得到了（1，1）的结果

```
一共执行了：137738次
一共执行了：137739次
耗时：+18339毫秒，(1,1)
```

这足以证明指令被重新排序了。

解决指令重排的方法是使用内存屏障：

在Java语言中我们可以使用volatile关键字来保证一个变量在一次读写操作时的避免指令重排，【内存屏障】是在我们的读写操作之前加入一条指令，当CPU碰到这条指令后必须等到前边的指令执行完成才能继续执行下一条指令。

一个对象的版初始化状态

Dog dog = new Dog();

new

invokespecial init

ldc

（2）可见性

我们看一下以下代码

```
public class Test {
    private static boolean isOver = false;

    private static int number = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                while (!isOver) {
                }
                System.out.println(number);
            }
        });
        thread.start();
        Thread.sleep(1000);
        number = 50;
        // 已经改了啊，应该可以退出上边循环的值了啊！
        isOver = true;
    }
}
```

如果你直接运行上面的代码，那么你永远也看不到number的输出的，线程将会无限的循环下去。你可能会有疑问，代码当中明明已经把isOver设置为了false，为什么循环还不会停止呢？这正是因为多线程之间可见性的问题。在单线程环境中，如果向某个变量写入某个值，在没有其他写入操作的影响下，那么你总能取到你写入的那个值。然而在多线程环境中，当你的读操作和写操作在不同的线程中执行时，情况就并非你想象的理所当然，也就是说不满足多线程之间的可见性，所以为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

thread线程一直在高速缓存中读取isOver的值，不能感知主线程已经修改了isOver的值而退出循环，这就是可见性的问题，使用【volatile】关键字可以解决这个问题

```
private volatile static boolean isOver = false;
```

1

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
50
```

程序成功退出，volatile能强制对改变量的读写直接在主存中操作，从而解决了不可见的问题。

写操作是，立刻强制刷在主存，并且将其他缓存区域的值设置为不可用

> happens-before语义

JMM用【happens-before】的概念来阐述操作之间的内存可见性。在JMM中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系 。

在Java 规范提案中为让大家理解内存可见性的这个概念，提出了happens-before的概念来阐述操作之间的内存可见性。对应Java程序员来说，理解happens-before是理解JMM的关键。JMM这么做的原因是：程序员对于这两个操作是否真的被重排序并不关心，程序员关心的是程序执行时的语义不能被改变（即执行结果不能被改变）。因此，happens-before关系本质上和as-if-serial语义是一回事。as-if-serial语义保证单线程内程序的执行结果不被改变，happens-before关系保证正确同步的多线程程序的执行结果不被改变。

volatile 禁止指令重排 内存的可见性

（3）线程争抢

举一个例子证明一下，线程1和线程2分别对count累计10000次，合适的结果应该是20000才对：

```
public class Test {
    private static int COUNT = 0;

    public static void adder(){
         COUNT++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                adder();
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                adder();
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("最后的结果是："+COUNT);
    }
}
```

最后我们发现每次的结果都不一样，都是10000以上的数字，这足以说明问题了，一个线程的结果对另一个线程不可见。

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
最后的结果是：12686
```

多个线程同时争抢相同的公共资源就是线程争抢，线程争抢会造成数据安全问题，上边的例子就是最好的解释。解决线程争抢问题的最好的方案就是【加锁】

```
public class Test {
    private static volatile int COUNT = 0;

    public synchronized static void adder(){
         COUNT++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                adder();
            }
        });
```

```
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                adder();
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("最后的结果是："+COUNT);
    }
}
```

在方法上加上synchronized可以上线程排队执行内部的代码块，

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
最后的结果是：20000

Process finished with exit code 0
```

> 来看一个小例子：

```
public class Ticket implements Runnable{

    private static Integer COUNT = 100;

    String name;

    public Ticket(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        while (Ticket.COUNT > 0) {
            ThreadUtils.sleep(100);
            System.out.println(name + "出票一张，还剩" + Ticket.COUNT-- + "张！");

        }
    }

    public static void main(String[] args) throws Exception {
        Thread one = new Thread(new Ticket("一号窗口"));
        Thread two = new Thread(new Ticket("一号窗口"));
        one.start();
        two.start();
    }
}
```

得到的结果是：

```
一号窗口出票一张，还剩20张！
一号窗口出票一张，还剩20张！
一号窗口出票一张，还剩19张！
一号窗口出票一张，还剩19张！
一号窗口出票一张，还剩18张！
一号窗口出票一张，还剩18张！
一号窗口出票一张，还剩17张！
一号窗口出票一张，还剩16张！
```

我们加上synchronized，照着写就行。

```
1   public class Ticket implements Runnable{
2
3       private static final Object monitor = new Object();
4       private static Integer COUNT = 100;
5
6       String name;
7
8       public Ticket(String name) {
9           this.name = name;
10      }
11
12      @Override
13      public void run() {
14          while (Ticket.COUNT > 0) {
15              ThreadUtils.sleep(100);
16              // 在这里加入了同步代码块
17              synchronized (Ticket.monitor) {
18                  System.out.println(name + "出票一张，还剩" + Ticket.COUNT-- +
    "张！");
19              }
20          }
21      }
22
23      public static void main(String[] args) throws Exception {
24          Thread one = new Thread(new Ticket("一号窗口"));
25          Thread two = new Thread(new Ticket("一号窗口"));
26          one.start();
27          two.start();
28          Thread.sleep(10000);
29      }
30  }
```

发现重复售票的问题被完美的解决了。

#4、线程安全的实现方法

（1）数据不可变

在Java当中，一切不可变的对象（immutable）一定是线程安全的，无论是对象的方法实现还是方法的调用者，都不需要再进行任何线程安全保障的措施，比如final关键字修饰的基础数据类型，再比如说咱们的Java字符串儿。只要一个不可变的对象被正确的构建出来，那外部的可见状态永远都不会改变，永远都不会看到它在多个线程之中处于不一致的状态，带来的安全性是最直接最纯粹的。比如使用final修饰的基础数据类型（引用数据类型不可以）、比如java字符串，而一旦被创建就永远不能改变，其实谷

歌的开发工具包（guava）中也给我们提供了一些不可变的一类（immutable），咱们以后的学习过程当中可能会接触到。

（2）互斥同步

互斥同步是常见的一种并发正确性的保障手段，同步是指在多个线程并发访问共享数据时，保证共享数据在同一时刻只被一个线程使用，互斥是实现同步的一种手段，互斥是因、同步是果，互斥是方法，同步是目的。

在Java中最基本的互斥同步手段，就是 `synchronized` 字段，除了synchronize的之外，我们还可以使用ReentrantLock等工具类实现。接下来我们就尝试学习Java中的锁。

（3）非阻塞同步

互斥同步面临的主要问题是，进行线程阻塞和唤醒带来的性能开销，因此这种同步也被称为阻塞同步，从解决问题的方式上来看互斥同步是一种【悲观的并发策略】，其总是认为，只要不去做正确的同步措施，那就肯定会出现问题，无论共享的数据是否真的出现，都会进行加锁。这将会导致用户态到内核态的转化、维护锁计数器和检查是否被阻塞的线程需要被唤醒等等开销。

随着硬件指令级的发展，我们已经有了另外的选择，基于【冲突检测的乐观并发策略】。通俗的说，就是不管有没有风险，先进行操作，如果没有其他线程征用共享数据，那就直接成功，如果共享数据确实被征用产生了冲突，那就再进行补偿策略，常见的补偿策略就是不断的重试，直到出现没有竞争的共享数据为止，这种乐观并发策略的实现，不再需要把线程阻塞挂起，因此同步操作也被称为非阻塞同步，这种措施的代码也常常被称之为【无锁编程】，也就是咱们说的自旋。我们用cas来实现这种非阻塞同步，cas会在接下来的授课当中详细给大家介绍，现在先不着急。

（4）无同步方案

在我们这个工作当中，还经常遇到这样一种情况，多个线程需要共享数据，但是这些数据又可以在单独的线程当中计算，得出结果，而不被其他的线程所影响，如果能保证这一点，我们就可以把共享数据的可见范围限制在一个线程之内，这样就无需同步，也能够保证个个线程之间不出现数据征用的问题，说人话就是数据拿过来，我用我的，你用你的，从而保证线程安全，比如说咱们的ThreadLocal。

ThreadLocal提供了线程内存储变量的能力，这些变量不同之处在于每一个线程读取的变量是对应的互相独立的。通过get和set方法就可以得到当前线程对应的值。

```
1  public class Test {
2
3      private static int number = 0;
4
5      public static void main(String[] args) throws InterruptedException {
6          Thread t1 = new Thread(new Runnable() {
7              @Override
8              public void run() {
9                  for (int i = 0; i < 1000; i++) {
10                     System.out.println("t1----:" + number++);
11                 }
12             }
13         });
14         Thread t2 = new Thread(new Runnable() {
15             @Override
16             public void run() {
17                 for (int i = 0; i < 1000; i++) {
18                     System.out.println("t2----:" + number++);
19                 }
20             }
21         });
22         t1.start();
```

```
        t2.start();

    }
}
```

使用ThreadLocal改造：

```
public class Test {

    private final static ThreadLocal<Integer> number = new ThreadLocal<>();
    public static final int COUNT = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                number.set(COUNT);
                for (int i = 0; i < 1000; i++) {
                    number.set(number.get() + 1);
                    System.out.println("t1----:" + number.get());
                }
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                number.set(COUNT);
                for (int i = 0; i < 1000; i++) {
                    number.set(number.get() + 1);
                    System.out.println("t2----:" + number.get());
                }
            }
        });
        t1.start();
        t2.start();
    }
}
```

#三、锁机制

上边的例子中，我们看到了synchronized的作用。

#1、synchronized简介

在多线程并发编程中 synchronized 一直是元老级角色，很多人都会称呼它为重量级锁。但是，随着 `Java SE 1.6` 对synchronized 进行了各种优化之后，有些情况下它就并不那么重，Java SE 1.6 中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁。

synchronized 有三种方式来加锁，分别是

1. 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
2. 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
3. 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。

使用方法如下：

分类	具体分类	被锁对象	伪代码
方法	实例方法	调用该方法的实例对象	public synchronized void method(){}
方法	静态方法	类对象Class对象	public static synchronized void method(){}
代码块	this	调用该方法的实例对象	synchronized(this){}
代码块	类对象	类对象	synchronized(Demo.class){}
代码块	任意的实例对象	创建的任意对象	Object lock= new Object(); synchronized(lock){}

#2、synchronized原理分析

我们写一段简单的代码，看看synchronized编译后的字节码：

```
1  public class Test {
2      public static void main(String[] args) {
3          synchronized (Test.class) {
4              int  a = 1;
5          }
6      }
7  }
```

上面的代码demo使用了synchroized关键字，锁住的是类对象。

编译之后，切换到Demo1.class的同级目录之后，然后用javap -v Demo1.class查看字节码文件：

> 反编译后的指令中能看到 `monitorenter` 和 `monitorexit`

```
1   public static void main(java.lang.String[]);
2       descriptor: ([Ljava/lang/String;)V
3       flags: ACC_PUBLIC, ACC_STATIC
4       Code:
5         stack=2, locals=4, args_size=1
6            0: ldc           #2                  // class aaa/Test
7            2: dup
8            3: astore_1
9            // 监视器进入
10           4: monitorenter
11           5: iconst_1
12           6: istore_2
13           7: aload_1
14           8: monitorexit
15           9: goto          17
16          12: astore_3
17          13: aload_1
18          14: monitorexit
19          15: aload_3
20          16: athrow
21          17: return
22  }
```

线程在获取锁的时候，实际上就是获得一个监视器对象(monitor) ,monitor 可以认为是一个同步对象，所有的Java 对象是天生携带monitor。而monitor是添加Synchronized关键字之后独有的。synchronized同步块使用了monitorenter和monitorexit指令实现同步，这两个指令，本质上都是对一个对象的监视器(monitor)进行获取，这个过程是【排他】的，也就是说同一时刻只能有一个线程获取到由synchronized所保护对象的监视器。

线程执行到monitorenter指令时，会尝试获取对象所对应的monitor所有权，也就是尝试获取对象的锁，而执行monitorexit，就是释放monitor的所有权。

> 接下来我们从对象头信息中发现一些锁的信息

对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）、数组类型还有一个int类型的数组长度。

我们今天要讲的就是其中的Mark Word

1. Mark Word记录了对象和锁有关的信息，当这个对象被synchronized关键字当成同步锁时，围绕这个锁的一系列操作都和Mark Word有关。
2. Mark Word在32位JVM中的长度是32bit，在64位JVM中长度是64bit。
3. Mark Word在不同的锁状态下存储的内容不同，在64位JVM中是这么存的：

Hotspot的实现

锁状态	25位	31位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
无锁态（new）	unused	hashCode（如果有调用）	unused	分代年龄	0	0	1

锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
偏向锁	当前线程指针 JavaThread*	Epoch	unused	分代年龄	1	0	1

锁状态	62位	2bit 锁标志位	
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针	0	0
重量级锁	指向互斥量（重量级锁）的指针	1	0
GC标记信息	CMS过程用到的标记信息	1	1

其中无锁和偏向锁的锁标志位都是01，只是在前面的1bit区分了这是无锁状态还是偏向锁状态。

JDK1.6以后的版本在处理同步锁时存在锁升级的概念，JVM对于同步锁的处理是从偏向锁开始的，随着竞争越来越激烈，处理方式从偏向锁升级到轻量级锁，最终升级到重量级锁。

> 锁升级中涉及的四把锁：

- 无锁：不加锁
- 偏向锁：不锁锁，只有一个线程争夺时，偏心某一个线程，这个线程来了不加锁。
- 轻量级锁：少量线程来了之后，先尝试自旋，不挂起线程。

  注：挂起线程和恢复线程的操作都需要转入内核态中完成这些操作，给系统的并发性带来很大的压力。在许多应用上共享数据的锁定状态，只会持续很短的一段时间，为了这段时间去挂起和恢复现场并不值得，我们就可以让后边请求的线程稍等一下，不要放弃处理器的执行时间，看看持有锁的线程是否很快就会释放，锁为了让线程等待，我们只需要让线程执行一个盲循环也就是我们说的自旋，这项技术就是所谓的【自旋锁】。
- 重量级锁：排队挂起线程

> JVM一般是这样使用锁和Mark Word的：

1，当没有被当成锁时，这就是一个普通的对象，Mark Word记录对象的HashCode，锁标志位是01，是否偏向锁那一位是0。

2，当对象被当做同步锁并有一个线程A抢到了锁时，锁标志位还是01，但是否偏向锁那一位改成1，前23bit记录抢到锁的线程id，表示进入偏向锁状态。

3，当线程A再次试图来获得锁时，JVM发现同步锁对象的标志位是01，是否偏向锁是1，也就是偏向状态，Mark Word中记录的线程id就是线程A自己的id，表示线程A已经获得了这个偏向锁，可以执行同步锁的代码。

4，当线程B试图获得这个锁时，JVM发现同步锁处于偏向状态，但是Mark Word中的线程id记录的不是B，那么线程B会先用CAS操作试图获得锁。如果抢锁成功，就把Mark Word里的线程id改为线程B的id，代表线程B获得了这个偏向锁，可以执行同步锁代码。如果抢锁失败，则继续执行步骤5。

5，偏向锁状态抢锁失败，代表当前锁有一定的竞争，偏向锁将升级为轻量级锁。JVM会在【当前线程】的线程栈中开辟一块单独的空间，里面保存指向对象锁Mark Word的指针，也叫所记录（lock record），同时在对象锁Mark Word中保存指向这片空间的指针。上述两个保存操作都是CAS操作，如果保存成功，代表线程抢到了同步锁，就把Mark Word中的锁标志位改成00，可以执行同步锁代码。如果保存失败，表示抢锁失败，竞争太激烈，继续执行步骤6。

6，轻量级锁抢锁失败，JVM会使用自旋锁，自旋锁不是一个锁状态，只是代表不断的重试，尝试抢锁。从JDK1.7开始，自旋锁默认启用，自旋次数由JVM决定。如果抢锁成功则执行同步锁代码，如果失败则继续执行步骤7，自旋默认10次。

7，自旋锁重试之后如果抢锁依然失败，同步锁会升级至重量级锁，锁标志位改为10。在这个状态下，未抢到锁的线程都会被阻塞排队。当后续线程尝试获取锁时，发现被占用的锁是重量级锁，则直接将自己挂起（而不是忙等）进入阻塞状态，等待将来被唤醒。就是所有的控制权都交给了操作系统，由操作系统来负责线程间的调度和线程的状态变更。而这样会出现频繁地对线程运行状态的切换，线程的挂起和唤醒，从而消耗大量的系统资源。

#3、死锁

死锁是这样一种情形：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

Java 死锁产生的四个必要条件：

- 1、互斥使用，即当资源被一个线程使用(占有)时，别的线程不能使用
- 2、不可抢占，资源请求者不能强制从资源占有者手中夺取资源，资源只能由资源占有者主动释放。
- 3、请求和保持，即当资源请求者在请求其他资源的同时保持对原有资源的占有。
- 4、循环等待，即存在一个等待队列：P1占有P2的资源，P2占有P3的资源，P3占有P1的资源。这样就形成了一个等待环路。

  当上述四个条件都成立的时候，便形成死锁。当然，死锁的情况下如果打破上述任何一个条件，便可让死锁消失。下面用Java代码来模拟一下死锁的产生。

解决死锁问题的方法是：一种是用synchronized，一种是用Lock显式锁实现。

```
1  import java.util.Date;
2
3  public class LockTest {
4     public static String obj1 = "obj1";
5     public static String obj2 = "obj2";
```

```
    public static void main(String[] args) {
        LockA la = new LockA();
        new Thread(la).start();
        LockB lb = new LockB();
        new Thread(lb).start();
    }
}
class LockA implements Runnable{
    public void run() {
        try {
            System.out.println(new Date().toString() + " LockA 开始执行");
            while(true){
                synchronized (LockTest.obj1) {
                    System.out.println(new Date().toString() + " LockA 锁住 obj1");
                    Thread.sleep(3000); // 此处等待是给B能锁住机会
                    synchronized (LockTest.obj2) {
                        System.out.println(new Date().toString() + " LockA 锁住 obj2");
                        Thread.sleep(60 * 1000); // 为测试，占用了就不放
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
class LockB implements Runnable{
    public void run() {
        try {
            System.out.println(new Date().toString() + " LockB 开始执行");
            while(true){
                synchronized (LockTest.obj2) {
                    System.out.println(new Date().toString() + " LockB 锁住 obj2");
                    Thread.sleep(3000); // 此处等待是给A能锁住机会
                    synchronized (LockTest.obj1) {
                        System.out.println(new Date().toString() + " LockB 锁住 obj1");
                        Thread.sleep(60 * 1000); // 为测试，占用了就不放
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

我们发现程序卡在这里出不去了，因为此时LockA想要obj2，LockB想要obj1，而这两个对象都被锁住了，形成互相等待的无法推出的问题。

就像两个小孩子吵架，互相拿了对方的玩具，归还时却说你先给我，我就给你，都这么说，那谁也拿不到。

#4、线程重入

线程重入是指任意线程在获取到锁之后，再次获取该锁而不会被该锁所阻塞。

```
public class Test {
    private static final Object M1 = new Object();
    private static final Object M2 = new Object();

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (M1){
                synchronized (M2){
                    synchronized (M1){
                        synchronized (M2){
                            System.out.println("hello lock");
                        }
                    }
                }
            }
        }).start();
    }
}
```

请问这段代码创建的线程会被自己锁死吗？答案是不会的，这就叫线程的重入，synchronized是可重入锁。

#5、wait和notify

我们学习了锁的知识后就可以学习这两个方法了。

```
public class WaitTest {

    private static  int num = 10;
    private static final Object MONITOR = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 0; ; i++) {
                ThreadUtils.sleep(5);
                minus(1,i);
            }
        });


        Thread t2 = new Thread(() -> {
            for (int i = 0; ; i++) {
                ThreadUtils.sleep(10);
                plus(2,i);
            }
        });

        t1.start();
        t2.start();

        System.out.println("------------------------------------------------");
    }

```

```
    public static void minus(int code,int i){
        synchronized (MONITOR){
            if(num <= 0){
                try {
                    MONITOR.wait(200);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("这是线程"+code+"--" + --num + "---"+i);
        }
    }

    public static void plus(int code,int i){
        synchronized (MONITOR){

            if(num >= 10){
                MONITOR.notify();
            }

            System.out.println("这是线程"+code+"--" + ++num + "---"+i);
        }
    }
}
```

```
这是线程1--0---708
这是线程2--1---699
这是线程2--2---700
这是线程2--3---701
这是线程2--4---702
这是线程2--5---703
这是线程2--6---704
这是线程2--7---705
这是线程2--8---706
```

我们确实发现，当线程1为0以后，确实等了一会，当num大于10以后减法器确实被唤醒了。

> 方法总结：

1、Thread的两个静态方法：

sleep释放CPU资源，但不释放锁。

yield方法释放了CPU的执行权，但是依然保留了CPU的执行资格。这个方法不常用

```
public class Test {

    private static AtomicInteger T1_COUNT = new AtomicInteger();
    private static AtomicInteger T2_COUNT = new AtomicInteger();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i<10000 ; i++) {
                Thread.yield();
                T1_COUNT.getAndAdd(1);
            }
```

```
        });


        Thread t2 = new Thread(() -> {
            for (int i = 0; i<10000 ; i++) {
                T2_COUNT.getAndAdd(1);
            }
        });

        t1.start();
        t2.start();
//        t1.join();
        t2.join();
        System.out.println("t1执行了：" + T1_COUNT.get());
        System.out.println("t2执行了：" + T2_COUNT.get());
    }
}
```

关键是我们不要`t1.join()`，因为出让执行权，理论上t2执行的速度会快于t1，而我们就是想看在相同时间内谁的执行次数多：

```
t1执行了：255
t2执行了：10000
```

这个方法我们能明显的看到答应的t2比t1多的多的多，因为每次t1都让出执行权。

如果我注释了：

```
//                Thread.yield();
```

结果就成了：

```
t1执行了：10000
t2执行了：10000
```

2、线程实例的方法：

- join：是线程的方法，程序会阻塞在这里等着这个线程执行完毕，才接着向下执行。

3、Object的成员方法

- wait：释放CPU资源，同时释放锁。
- notify：唤醒等待中的线程。
- notifyAll：唤醒所有等待的线程。

#6、线程的退出

（1）使用退出标志，使线程正常退出，也就是当`run()`方法结束后线程终止。

```
class Thread01 extends Thread {

    // volatile关键字解决线程的可见性问题
    volatile boolean flag = true;

    @Override
    public void run() {
        while (flag) {
            try {
                // 可能发生异常的操作
```

```
                System.out.println(getName() + "线程一直在运行。。。");
            } catch (Exception e) {
                System.out.println(e.getMessage());
                this.stopThread();
            }
        }
    }

    public void stopThread() {
        System.out.println("线程停止运行。。。");
        this.flag = false;
    }
}

public class StopThreadDemo01 {

    public static void main(String[] args) {
        Thread01 thread01 = new Thread01();
        thread01.start();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread01.stopThread();
    }
}
```

(2) 使用`interrupt()`方法中断线程（只有线程在`wait`和`sleep`才会捕获`InterruptedException`异常，执行终止线程的逻辑，在运行中不会捕获）

```
class Thread02 extends Thread {
    private boolean flag = true;

    @Override
    public void run() {
        while (flag) {
            synchronized (this) {
//                try {
//                    wait();
//                } catch (InterruptedException e) {
//                    e.printStackTrace();
//                    this.stopThread();
//                }

                try {
                    sleep(10000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    this.stopThread();
                }
            }
        }
    }

    public void stopThread() {
        System.out.println("线程已经退出。。。");
```

```
            this.flag = false;
        }
    }

    public class StopThreadDemo02 {

        public static void main(String[] args) {
            Thread02 thread02 = new Thread02();
            thread02.start();
            System.out.println("线程开始");
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            thread02.interrupt();
        }
    }
```

> 调用`interrupt()`方法会抛出`InterruptedException`异常，捕获后再做停止线程的逻辑即可。

> **如果线程处于类似`while(true)`运行的状态，`interrupt()`方法无法中断线程。**

#7、LockSupport

`LockSupport`是一个线程阻塞工具类，所有的方法都是静态方法，可以让线程在任意位置阻塞，当然阻塞之后肯定得有唤醒的方法。

接下来我来看看`LockSupport`有哪些常用的方法。主要有两类方法：`park`和`unpark`。

```
public static void park(Object blocker); // 暂停当前线程
public static void parkNanos(Object blocker, long nanos); // 暂停当前线程，不过有超时时间的限制
public static void parkUntil(Object blocker, long deadline); // 暂停当前线程，直到某个时间
public static void park(); // 无期限暂停当前线程
public static void parkNanos(long nanos); // 暂停当前线程，不过有超时时间的限制
public static void parkUntil(long deadline); // 暂停当前线程，直到某个时间
public static void unpark(Thread thread); // 恢复当前线程
public static Object getBlocker(Thread t);
```

为什么叫park呢，park英文意思为停车。我们如果把Thread看成一辆车的话，park就是让车停下，unpark就是让车启动然后跑起来。

我们写一个例子来看看这个工具类怎么用的。

```
public class LockSupportTest {

    public static final Object MONITOR = new Object();

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = ()->{
            synchronized (MONITOR) {
                System.out.println("线程【" + Thread.currentThread().getName() + "】正在执行。");
```

```
                LockSupport.park();
                if (Thread.currentThread().isInterrupted()) {
                    System.out.println("被中断了");
                }
                System.out.println("继续执行");
            }
        };

        Thread t1 = new Thread(runnable,"线程一");
        Thread t2 = new Thread(runnable,"线程二");

        t1.start();
        Thread.sleep(1000L);
        t2.start();
        Thread.sleep(3000L);
        t1.interrupt();
        LockSupport.unpark(t2);
        t1.join();
        t2.join();
    }
}
```

运行的结果如下：

```
线程【线程一】正在执行。
被中断了
继续执行
线程【线程二】正在执行。
继续执行
```

这儿`park`和`unpark`其实实现了`wait`和`notify`的功能，不过还是有一些差别的。

1. `park`不需要获取某个对象的锁
2. 因为中断的时候`park`不会抛出`InterruptedException`异常，所以需要在`park`之后自行判断中断状态，然后做额外的处理

我们在park线程的时候可以传递一些信息，给调用者看，这个object什么都能传递。

比如在阻塞时：

```
LockSupport.park("我被阻塞了");
```

主线程可以在t1的阻塞期间获取它传入的信息：

```
t1.start();
Thread.sleep(1000L);
System.out.println(LockSupport.getBlocker(t1));
t2.start();
```

小结：

1. `park和unpark`可以实现类似`wait和notify`的功能，但是并不和`wait和notify`交叉，也就是说`unpark`不会对`wait`起作用，`notify`也不会对`park`起作用。
2. `park和unpark`的使用不会出现死锁的情况
3. blocker的作用是看到阻塞对象的信息

#8、Lock锁

Lock接口有几个重要方法：

```
// 获取锁
void lock()

// 仅在调用时锁为空闲状态才获取该锁，可以响应中断
boolean tryLock()

// 如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁
boolean tryLock(long time, TimeUnit unit)

// 释放锁
void unlock()
```

> 获取锁，两种写法

```
Lock lock = ...;
lock.lock();
try{
    //处理任务
}catch(Exception ex){

}finally{
    lock.unlock();   //释放锁
}
```

```
 Lock lock = ...;
 if(lock.tryLock()) {
      try{
          //处理任务
      }catch(Exception ex){

      }finally{
          lock.unlock();   //释放锁
      }
 }else {
     //如果不能获取锁，则直接做其他事情
 }
```

> Lock的实现类 ReentrantLock

ReentrantLock，可重入锁。ReentrantLock是实现了Lock接口的类，并且ReentrantLock提供了更多的方法实现线程同步。下面通过一些实例学习如何使用 ReentrantLock。

用法上边已经讲了：

（1）ReentrantLock

可重入锁，之前使用synchronized的案例都可以使用ReentrantLock替代：

```
public class Ticket implements Runnable{

    private static final ReentrantLock lock = new ReentrantLock();
    private static Integer COUNT = 100;
```

```
5
6       String name;
7
8       public Ticket(String name) {
9           this.name = name;
10      }
11
12      @Override
13      public void run() {
14          while (Ticket.COUNT > 0) {
15              ThreadUtils.sleep(100);
16              // 就在这里
17              lock.lock();
18              try {
19                  System.out.println(name + "出票一张，还剩" + Ticket.COUNT-- +
    "张！");
20              } finally {
21                  lock.unlock();
22              }
23          }
24      }
25
26      public static void main(String[] args) throws Exception {
27          Thread one = new Thread(new Ticket("一号窗口"));
28          Thread two = new Thread(new Ticket("一号窗口"));
29          one.start();
30          two.start();
31          Thread.sleep(10000);
32      }
33  }
```

synchronized和ReentrantLock的区别：

1、区别：

- Lock是一个接口，synchronized是Java中的关键字，synchronized是内置的语言实现；
- synchronized发生异常时，会自动释放线程占用的锁，故不会发生死锁现象。Lock发生异常，若没有主动释放，极有可能造成死锁，故需要在finally中调用unLock方法释放锁；
- Lock可以让等待锁的线程响应中断，使用synchronized只会让等待的线程一直等待下去，不能响应中断
- Lock可以提高多个线程进行读操作的效率

2、ReentrantLock以下功能是synchronized不具备的：

（2）ReadWriteLock

对于一个应用而言，一般情况读操作是远远要多于写操作的，同时如果仅仅是读操作没有写操作的情况下数据又是线程安全的，读写锁给我们提供了一种锁，读的时候可以很多线程同时读，但是不能有线程写，写的时候是独占的，其他线程既不能写也不能读。在某些场景下能极大的提升效率。

```
1   public class ReadAndWriteLockTest {
2       public static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
3       public static int COUNT = 1;
4
5       public static void main(String[] args) {
6           //同时读、写
7           Runnable read = () -> {
8               ReentrantReadWriteLock.ReadLock readLock = lock.readLock();
```

```
            readLock.lock();
            try{
                ThreadUtils.sleep(2000);
                System.out.println("我在读数据：" + COUNT);
            }finally {
                readLock.unlock();
            }
        };

        //同时读、写
        Runnable write = () -> {
            ReentrantReadWriteLock.WriteLock writeLock = lock.writeLock();
            writeLock.lock();
            try{
                ThreadUtils.sleep(2000);
                System.out.println("我在写数据：" + COUNT++);
            }finally {
                writeLock.unlock();
            }
        };

        for (int i = 0; i < 100; i++) {
            Random random = new Random();
            int flag = random.nextInt(100);
            if(flag > 20){
                new Thread(read,"read").start();
            }else{
                new Thread(write,"write").start();
            }
        }
    }
}
```

#9、lock锁的原理cas和aqs

本节我们从ReentrantLock的源码，一起解释这些并发编程工具的实现原理，其实很多场景下我们使用synchronized也可以，但是毕竟他不够灵活，是由c++实现的，只能作为关键字来使用，而Java给我们提供了并发编程包，由Doug Lea编写了大量的共性能的线程同步器，而底层的实现原理就是cas和aqs。最后补充一句，能用synchronized实现我们就用synchronized，这是关键字也是jdk团队优化的主要目标。

（1）并发编程的三大特性

> 原子性

原子操作定义：原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而只执行其中的一部分（不可中断性）。将整个操作视为一个整体是原子性的核心特征。原子性不仅仅是多行代码，也可能是多条指令。

存在竞争条件，线程不安全，需要转变原子操作才能安全。方式：上锁、循环CAS；上例只是针对一个变量的原子操作改进，我们也可以实现更大逻辑的原子操作。

> 可见性

我们已经深度的了解过

> 有序性

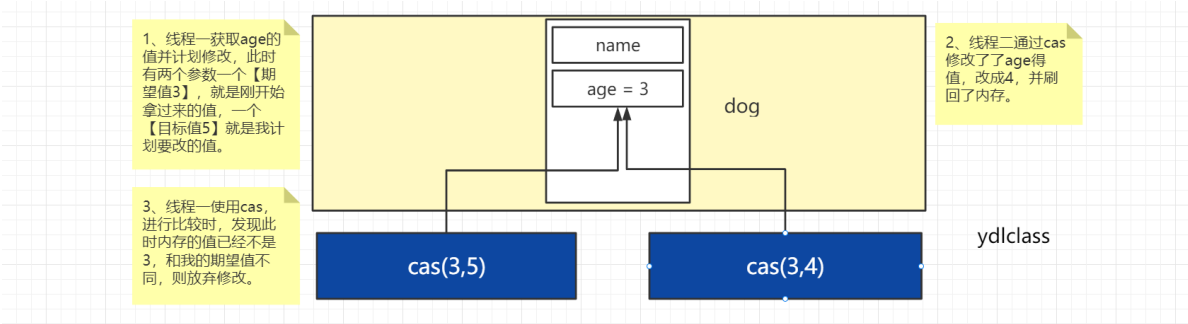
volatile：可以保证可见性和有序行

synchronized和Lock：可以保证原子性、可见性、有序性

（2）CAS

CAS，compare and swap的缩写，中文翻译成比较并交换，我发现jdk11以后改成了compare and set。

它的思路其实很简单，就是给一个元素赋值的时候，先看看内存里的那个值到底变没变，如果没变我就修改，变了我就不改了，其实这是一种无锁操作，不需要挂起线程，无锁的思路就是先尝试，如果失败了，进行补偿，也就是你可以继续尝试。这样在少量竞争的情况下能很大程度提升性能。



我们可以使用一个宏观上的例子给大家讲解一下。

```
1   public class CasTest {
2
3       public volatile static int COUNT = 0;
4
5       public synchronized static boolean compareAndSwap(int expect, int update) {
6           if (expect == COUNT) {
7               COUNT = update;
8               return true;
9           }
10          return false;
11      }
12
13      public static void main(String[] args) throws InterruptedException {
14          for (int j = 0; j < 100; j++) {
15              new Thread(() -> {
16                  ThreadUtils.sleep(1);
17                  // 模拟自旋
18                  while (!compareAndSwap(COUNT, COUNT + 1)){
19                  }
20              }).start();
21          }
22          ThreadUtils.sleep(1000);
23          System.out.println(COUNT);
24      }
25  }
```

这个案例其实有些不恰当，我们想做的是在赋值阶段可以通过尝试比较预期值的方式来判断是否能修改当前值。但事实上还是使用了synchronized，这脱离了初衷，CAS在计算机底层也是三个动作：【取值】、【比较】、【赋值】，只不过这三个动作是CPU原语级别的原子动作，不需要我们程序员担心。

Java中的CAS是通过sun.misc.Unsafe类提供的。Unsafe中的CAS

```
/**
*Object var1       你要修改哪个对象的成员变量
*long offset       这个值在这个对象中的偏移量
*Object expected   期望值
*Object x          实际值
*/
public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);
```

jdk11中改为了：

```
@HotSpotIntrinsicCandidate
public final native boolean compareAndSetObject(Object o, long offset,
                                                Object expected,
                                                Object x);
```

CAS保证的是对一个对象写操作的无锁原子性，加syncronized的也具有原子性。

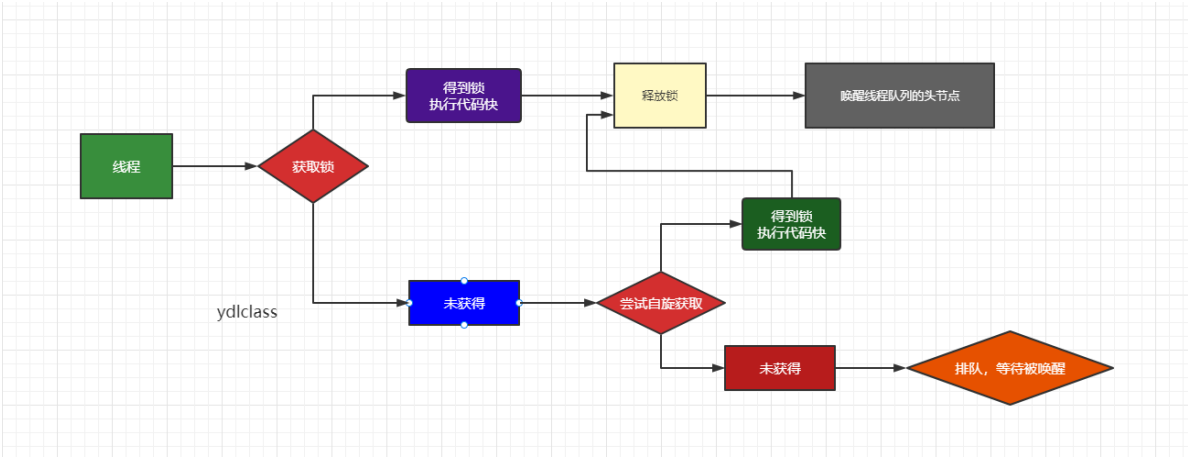
但是CAS还是有几个缺点：

1. ABA问题。当第一个线程执行CAS操作，尚未修改为新值之前，内存中的值已经被其他线程连续修改了两次，使得变量值经历 A -> B -> A的过程。绝大部分场景我们对ABA不敏感。解决方案：添加版本号作为标识，每次修改变量值时，对应增加版本号； 做CAS操作前需要校验版本号。JDK1.5之后，新增AtomicStampedReference类来处理这种情况。
2. 循环时间长开销大。如果有很多个线程并发，CAS自旋可能会长时间不成功，会增大CPU的执行开销。
3. 只能对一个变量进行原子操作。JDK1.5之后，新增AtomicReference类来处理这种情况，可以将多个变量放到一个对象中。

（3）AQS

抽象队列同步器，用来解决线程同步执行的问题。

AQS解决问题的思路如下：



我们可以在AQS中看到这样的代码：

```
static final class Node {
        /** Marker to indicate a node is waiting in shared mode */
        static final Node SHARED = new Node();
        /** Marker to indicate a node is waiting in exclusive mode */
```

```
5           static final Node EXCLUSIVE = null;
6
7           /** waitStatus value to indicate thread has cancelled. */
8           static final int CANCELLED =  1;
9           /** waitStatus value to indicate successor's thread needs unparking. */
10          static final int SIGNAL    = -1;
11          /** waitStatus value to indicate thread is waiting on condition. */
12          static final int CONDITION = -2;
13          /**
14           * waitStatus value to indicate the next acquireShared should
15           * unconditionally propagate.
16           */
17          static final int PROPAGATE = -3;
18
19          //CANCELLED（1）：取消状态，当线程不再希望获取锁时，设置为取消状态
20          //SIGNAL（-1）：当前节点的后继者处于等待状态，当前节点的线程如果释放或取消了同步状态，
    通知后继节点
21          //CONDITION（-2）：等待队列的等待状态，当调用signal()时，进入同步队列
22          //PROPAGATE（-3）：共享模式，同步状态的获取的可传播状态
23          //0：初始状态
24          volatile int waitStatus;
25
26          volatile Node prev;
27
28          volatile Node next;
29
30          /**
31           * The thread that enqueued this node.  Initialized on
32           * construction and nulled out after use.
33           */
34          volatile Thread thread;
35
36          Node nextWaiter;
37
38          /**
39           * Returns true if node is waiting in shared mode.
40           */
41          final boolean isShared() {
42              return nextWaiter == SHARED;
43          }
44
45          final Node predecessor() {
46              Node p = prev;
47              if (p == null)
48                  throw new NullPointerException();
49              else
50                  return p;
51          }
52
53          /** Establishes initial head or SHARED marker. */
54          Node() {}
55
56          /** Constructor used by addWaiter. */
57          Node(Node nextWaiter) {
58              this.nextWaiter = nextWaiter;
59              THREAD.set(this, Thread.currentThread());
60          }
61      }
```

```
62
63      private transient volatile Node head;
64
65      private transient volatile Node tail;
```

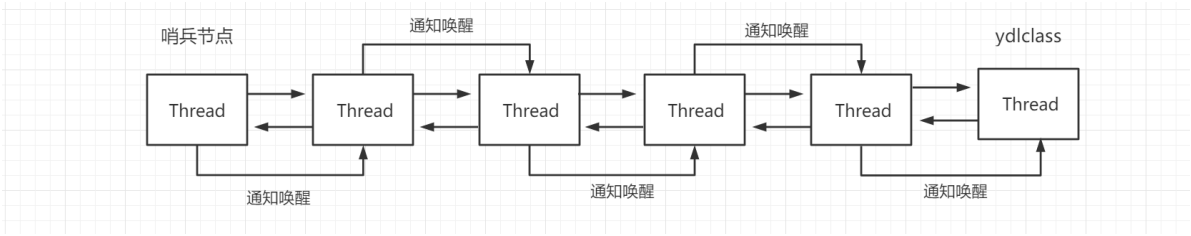
从这段代码中我们看到AQS中维护了一个队列，这个队列是个双向队列，里边保存了一个线程，还有一个状态。

简单的聊聊这个队列，他叫【CLH队列】，这种队列有什么特性：

1、它是一个双向链表

2、CLH同步队列中，一个节点表示一个线程，它保存着线程的引用（thread）、状态（waitStatus）、前驱节点（prev）、后继节点（next）等信息。

结构如下图：



我们以`ReentrantLock`来分析其中的过程：

有一个sunc

```
1    abstract static class Sync extends AbstractQueuedSynchronizer
```

两个

```
1   static final class FairSync extends Sync
```

```
1   static final class NonfairSync extends Sync
```

#a、构造

我们发现不传值是非公平锁，传入`true`是公平锁，有啥区别咱们慢慢看：

```
1   public ReentrantLock() {
2           sync = new NonfairSync();
3       }
4
5   public ReentrantLock(boolean fair) {
6       sync = fair ? new FairSync() : new NonfairSync();
7   }
```

#b、加锁

（获取锁）acquire就是获取的意思：

```
// NonfairSync 非公平的加锁动作一上来就抢一下，这是非公平锁的第一次抢锁
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}

// FairSync 公平锁直接调用acquire(1)
final void lock() {
    acquire(1);
}
```

sync.acquire(1) 方法：

```
public final void acquire(int arg) {
    if (   !tryAcquire(arg)
        &&
        acquireQueued(   addWaiter(Node.EXCLUSIVE), arg)   )
        selfInterrupt();
}
```

将if语句拆开了，会有以下三个步骤：

1. !tryAcquire(arg)
2. addWaiter(Node.EXCLUSIVE), arg)
3. acquireQueued(addWaiter(Node.EXCLUSIVE), arg)

首先，!tryAcquire(arg) 尝试获取锁，公平锁和非公平锁的差别就在这里：

> 非公平锁的获取锁方式

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



final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        // 直接设置状态，并将当前的锁持有者改成自己，第二次自旋获取，非公平锁有两次抢锁的机会
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

> 公平锁的获取锁方式

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        // 先看看有没有排队的节点，再尝试获取锁
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
}
```

公平锁 会看看有没有队列，有队列就排队，而非公平锁根本不管有无队列都直接抢锁。

#c、入队

如果没有获得锁，就排队，addWaiter(Node.EXCLUSIVE) 添加一个节点到队列

```
private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
        Node pred = tail;
        if (pred != null) {
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        enq(node);
        return node;
    }
```

```
private Node enq(final Node node) {
        for (;;) {
            Node t = tail;
            // 插入了一个空节点，就是一个哨兵，因为每一个真实的线程节点都会坚挺前一个节点的状态
            if (t == null) { // Must initialize
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else {
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
```

```
                }
            }
        }
    }
```

#d、阻塞

入队完成之后再判断一次当前是否有可能获得锁，也就是前一个节点是head的话，前一个线程有可能已经释放了，再获取一次，如果获取成功，设置当前节点为头节点，整个获取过程完成。

```
final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                final Node p = node.predecessor();
                // 不死心，进了队伍了，发现我是第二个，还要尝试获取一下
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
                }
                // 真正的挂起线程
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

获取失败的话先将之前的节点等待状态设置为SIGNAL，如果之前的节点取消了就向前一直找。

```
// 就是要将我的前一个节点的等待状态改为SIGNAL
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
            /*
             * This node has already set status asking a release
             * to signal it, so it can safely park.
             * 前驱节点已经设置了SIGNAL，闹钟已经设好，现在我可以安心睡觉（阻塞）了。
             * 如果前驱变成了head，并且head的代表线程exclusiveOwnerThread释放了锁，
             * 就会来根据这个SIGNAL来唤醒自己
             */
            return true;
        if (ws > 0) {
            /*
             * 发现传入的前驱的状态大于0，即CANCELLED。说明前驱节点已经因为超时或响应了中断，
             * 而取消了自己。所以需要跨越掉这些CANCELLED节点，直到找到一个<=0的节点
             */
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
```

```
            /*
             * 进入这个分支，ws只能是0或PROPAGATE。
             * CAS设置ws为SIGNAL
             */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

上一个条件完成之后，我就可以安心的阻塞了，然后一直等待直到被唤醒

```
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

上面就是获取锁并等待的过程，总结起来就是：

当`lock()`执行的时候：

- 先快速获取锁，当前没有线程执行的时候直接获取锁
- 尝试获取锁，当没有线程执行或是当前线程占用锁，可以直接获取锁
- 将当前线程包装为node放入同步队列，设置为尾节点
- 前一个节点如果为头节点，再次尝试获取一次锁
- 将前一个有效节点设置为SIGNAL
- 然后阻塞直到被唤醒

e、释放锁

当ReentrantLock进行释放锁操作时，调用的是AQS的`release(1)`操作

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

在`tryRelease(arg)`中会将锁释放一次，如果当前state是1，且当前线程是正在占用的线程，释放锁成功，返回true，否则因为是可重入锁，释放一次可能还在占用，应一直释放直到state为0为止

```
 private void unparkSuccessor(Node node) {
     int ws = node.waitStatus;
     if (ws < 0)
         compareAndSetWaitStatus(node, ws, 0);
     Node s = node.next;
     // 如果没有下一个节点，或者下个节点的状态被取消了，就从尾节点开始找，找到最前面一个可用的节点
     if (s == null || s.waitStatus > 0) {
         s = null;
         for (Node t = tail; t != null && t != node; t = t.prev)
             if (t.waitStatus <= 0)
                 s = t;
     }
     // 唤醒下一个节点
```

```
        if (s != null)
            LockSupport.unpark(s.thread);
    }
```

然后优先找下一个节点，如果取消了就从尾节点开始找，找到最前面一个可用的节点

#四、JUC并发编程包

#1、原子类

（1）认识 Atomic 原子类

Atomic 翻译成中文是原子的意思。在化学中，原子是构成一般物质的最小单位，是不可分割的。而在这里，Atomic 表示当前操作是不可中断的，即使是在多线程环境下执行，Atomic 类，是具有原子操作特征的类。

Java 的原子类都存放在并发包 `java.util.concurrent.atomic` 下。

（2） JUC 包中的原子类

> 基本类型

使用原子的方式更新基本类型

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

> 数组类型

使用原子的方式更新数组里的某个元素

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

> 引用类型

- AtomicReference：引用类型原子类
- AtomicStampedReference：原子更新引用类型里的字段原子类
- AtomicMarkableReference ：原子更新带有标记位的引用类型

> 对象的属性修改类型**

- AtomicIntegerFieldUpdater：原子更新整形字段的更新器
- AtomicLongFieldUpdater：原子更新长整形字段的更新器
- 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，可用于解决原子的更新数据和数据的版本号，以及解决使用 CAS 进行原子更新时可能出现的 ABA 问题

(3) 讲讲 AtomicInteger 的使用

打开 AtomicInteger 源码，我们发现该类常用方法有以下

```
public final int get();  // 获取当前的值
public final int getAndSet(int newValue);  // 获取当前的值，并设置新的值
public final int getAndIncrement();  // 获取当前的值，并自增
public final int getAndDecrement();  // 获取当前的值，并自减
public final int getAndAdd(int delta);  // 获取当前的值，并加上预期的值
boolean compareAndSet(int expect, int update);  // 如果输入的数值等于预期值，则以原子方式将该值设置为输入值（update）
public final void lazySet(int newValue);  // 最终设置为 newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

AtomicInteger 类使用示例，我们开启1000个线程做加法，发现结果没问题，然而我们并没有直接使用锁

```
public class Test {

    private static AtomicInteger ADDER = new AtomicInteger();

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

        for (int i = 0; i < 1000; i++) {
            Thread thread = new Thread(() -> {
                ADDER.getAndIncrement();
            });
            thread.start();
            thread.join();
        }
        System.out.println(ADDER.get());
    }
}
```

(4) AtomicInteger 类原理

以 AtomicInteger 类为例，以下是部分源代码：

该类维护一个volatile修饰的int，保证了可见性和有序性：

```
private volatile int value;
```

所有的方法都是使用 cas 保证了原子性，所以这几个类都是线程安全的：

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

```
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;
}
```

我们发现原子类中的任何操作都没有上锁，是无锁操作。

#2、线程池

为什么要使用线程池?

(1) 降低资源消耗。 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

(2) 提高响应速度。 当任务到达时，任务可以不需要等到线程创建就能立即执行。

(3) 提高线程的可管理性。 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配、调优和监控。

（1）jdk自带的四种线程池

Java通过Executors提供四种线程池，分别为：

1. `newCachedThreadPool` 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
2. `newFixedThreadPool` 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
3. `newScheduledThreadPool` 创建一个定长线程池，支持定时及周期性任务执行。
4. `newSingleThreadExecutor` 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序执行。

> 简单使用

```
1   public class UseExecutors {
2       public static void main(String[] args) {
3           Runnable taskOne = () ->
    System.out.println(Thread.currentThread().getName()+":taskOne");
4           // ExecutorService pools = Executors.newCachedThreadPool();
5           // ExecutorService pools = Executors.newSingleThreadExecutor();
6           // ExecutorService pools = Executors.newScheduledThreadPool(10);
7           ExecutorService pools = Executors.newFixedThreadPool(10);
8           for (int i = 0; i < 40; i++) {
9               pools.submit(taskOne);
10          }
11      }
12  }
```

> **无论是哪一个都是调用ThreadPoolExecutor 构造方法：**

```
1   public ThreadPoolExecutor
2       (int corePoolSize,
3        int maximumPoolSize,
4        long keepAliveTime,
5        TimeUnit unit,
6        BlockingQueue<Runnable> workQueue,
7        ThreadFactory threadFactory,
8        RejectedExecutionHandler handler)
```

（2）参数的意义-重要

corePoolSize	指定了线程池里的线程数量，核心线程池大小
maximumPoolSize	指定了线程池里的最大线程数量
keepAliveTime	当线程池线程数量大于corePoolSize时候，多出来的空闲线程，多长时间会被销毁
unit	时间单位，TimeUnit
workQueue	任务队列，用于存放提交但是尚未被执行的任务
threadFactory	线程工厂，用于创建线程，线程工厂就是给我们new线程的
handler	所谓拒绝策略，是指将任务添加到线程池中时，线程池拒绝该任务所采取的相应策略

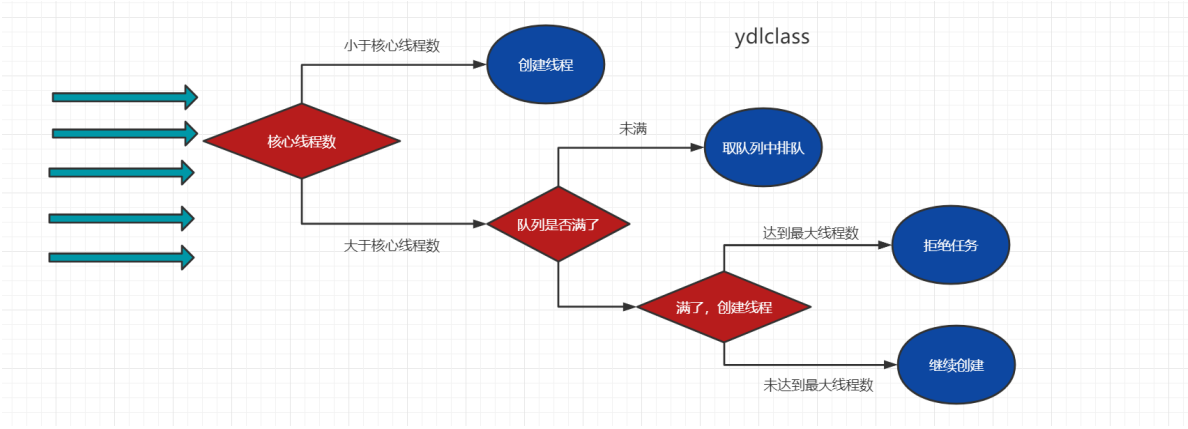
> 常见的工作队列我们有如下选择，这些都是阻塞队列，阻塞队列的意思是，当队列中没有值的时候，取值操作会阻塞，一直等队列中产生值。

- ArrayBlockingQueue：基于数组结构的有界阻塞队列，FIFO。
- LinkedBlockingQueue：基于链表结构的有界阻塞队列，FIFO。

> 线程池提供了四种拒绝策略：

- AbortPolicy：直接抛出异常，默认策略；
- CallerRunsPolicy：用调用者所在的线程来执行任务；
- DiscardOldestPolicy：丢弃阻塞队列中最靠前的任务，并执行当前任务；
- DiscardPolicy：直接丢弃任务；

> 线程池按以下行为执行任务



我们来看一下这四种线程池都是使用`ThreadPoolExecutor`进行构造的:

> newCachedThreadPool

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

通过指定参数,返回ThreadPoolExecutor来实现. 参数为:

```
核心线程池大小=0
最大线程池大小为Integer.MAX_VALUE
线程过期时间为60s
使用SynchronousQueue作为工作队列.
```

所以线程池为0-max个线程,并且会60s过期,实现了可以缓存的线程池。

> newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
核心线程池大小=传入参数
最大线程池大小为传入参数
线程过期时间为0ms
LinkedBlockingQueue作为工作队列.
```

通过最小与最大线程数量来控制实现定长线程池.

> newScheduledThreadPool

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue());
}
核心线程池大小=传入参数
最大线程池大小为Integer.MAX_VALUE
线程过期时间为0ms
DelayedWorkQueue作为工作队列.
```

主要是通过DelayedWorkQueue来实现的定时线程。

> newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
核心线程池大小=1
最大线程池大小为1
线程过期时间为0ms
LinkedBlockingQueue作为工作队列.
```

综上，Java提供的4种线程池,只是预想了一些使用场景,使用参数定义的而已,我们在使用的过程中,完全可以根据业务需要,自己去定义一些其他类型的线程池来使用(如果需要的话).

（3）自定义线程池

这里是针对JDK1.8版本，使用JDK自带的线程池会出现OOM问题，中小型公司一般很难遇到，在阿里巴巴开发文档上面有明确的标识：

上边我们已经分析了线程池的几个参数，这几个参数核心线程数、最大线程数、活跃时间和单位根据服务器本身的性能和程序的特性设定，这个是个经验值，如果我们去设置可能效果不太好，但是起码这几个只是数字我们自定义的时候可以很简单的填入。但是线程工厂、决绝策略、阻塞队列又该怎么搞呢？

1. 拒绝策略其实很简单，ExecutorService构造时可以不传递拒绝策略，默认使用异常抛出的方式。
2. 阻塞队列我们搞一个定长的队列就好了，ArrayBlockingQueue<>(DEFAULT_SIZE)
3. 线程工厂的获取我们可以使用以下的方法：

> 第一种办法，看看原生的怎么搞一个线程工厂：

```
public ThreadPoolExecutor( @Range(from = 0, to = java.lang.Integer.MAX_VALUE) int corePoolS
                           @Range(from = 1, to = java.lang.Integer.MAX_VALUE) int maximumPo
                           @Range(from = 0, to = java.lang.Long.MAX_VALUE) long keepAlive
                           @NotNull TimeUnit unit,
                           @NotNull BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
```

进入看他的源码：

```
1   static class DefaultThreadFactory implements ThreadFactory {
2           private static final AtomicInteger poolNumber = new AtomicInteger(1);
3           private final ThreadGroup group;
4           private final AtomicInteger threadNumber = new AtomicInteger(1);
5           private final String namePrefix;
6
7           DefaultThreadFactory() {
8               SecurityManager s = System.getSecurityManager();
9               group = (s != null) ? s.getThreadGroup() :
10                                    Thread.currentThread().getThreadGroup();
11              namePrefix = "pool-" +
12                           poolNumber.getAndIncrement() +
13                          "-thread-";
14          }
15
16          public Thread newThread(Runnable r) {
17              Thread t = new Thread(group, r,
18                                    namePrefix + threadNumber.getAndIncrement(),
19                                    0);
20              if (t.isDaemon())
21                  t.setDaemon(false);
22              if (t.getPriority() != Thread.NORM_PRIORITY)
23                  t.setPriority(Thread.NORM_PRIORITY);
24              return t;
```

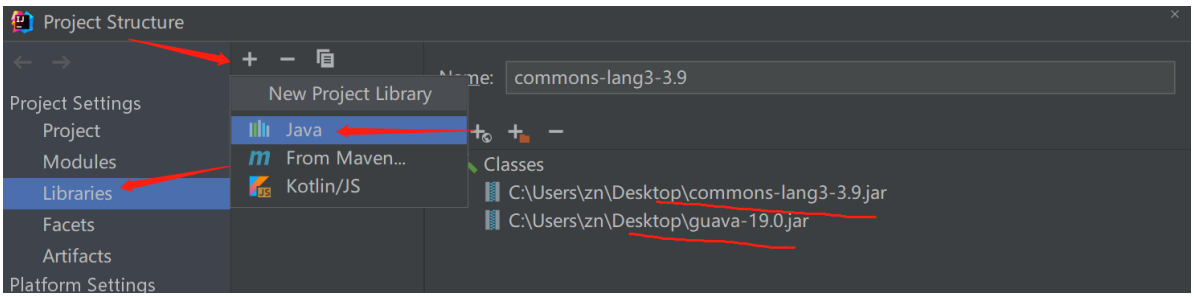
```
        }
    }
```

我们可以按照他的方式自己写一个，看不懂无所谓，起码从源码中我们看见了，线程工厂就是创建线程的，这里用到了一种设计模式，叫工厂设计模式。

```
public class MyThreadFactory {
    private static final AtomicInteger poolNumber = new AtomicInteger(1);
    private final ThreadGroup group;
    private final AtomicInteger threadNumber = new AtomicInteger(1);
    private final String namePrefix;

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

    MyThreadFactory(){
        this("default");
    }

    public Thread newThread(Runnable r) {
        // 就是在创建线程
        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;
    }
}
```

> 第二种：Google guava 工具类 提供的 `ThreadFactoryBuilder` 。

需要引入jar包，这就是别人写的类：点击File ---》 project structure




```
ThreadFactory guavaThreadFactory = new
ThreadFactoryBuilder().setNameFormat("retryClient-pool-").build();
```

> 第三种：Apache commons-lang3 提供的 `BasicThreadFactory` 。

```
ThreadFactory basicThreadFactory = new BasicThreadFactory.Builder()
        .namingPattern("basicThreadFactory-").build();
```

看怎么去定义一下线程池：

```
public class AsyncProcessor {

    /**
     * 默认最大并发数<br>
     */
    private static final int DEFAULT_MAX_CONCURRENT = Runtime.getRuntime().availableProcessors() * 2;

    /**
     * 线程池名称格式
     */
    private static final String THREAD_POOL_NAME = "ydlclasslog-%d";

    /**
     * 线程工厂名称
     */
    private static final ThreadFactory FACTORY = new BasicThreadFactory.Builder().namingPattern(THREAD_POOL_NAME)
            .daemon(true).build();

    /**
     * 默认队列大小
     */
    private static final int DEFAULT_SIZE = 500;

    /**
     * 默认线程存活时间
     */
    private static final long DEFAULT_KEEP_ALIVE = 60L;

    /**
     * NewEntryServiceImpl.java:689
     * Executor
     */
    private static ExecutorService executor;

    /**
     * 执行队列
     */
    private static BlockingQueue<Runnable> executeQueue = new ArrayBlockingQueue<>(DEFAULT_SIZE);

    static {
        executor = new ThreadPoolExecutor(
                DEFAULT_MAX_CONCURRENT,
                DEFAULT_MAX_CONCURRENT * 4,
                DEFAULT_KEEP_ALIVE,
                TimeUnit.SECONDS,
                executeQueue,
                FACTORY);

    }

```

```
51      /**
52       * 此类型无法实例化
53       */
54      private AsyncProcessor() {
55      }
56
57      public static boolean executeTask(Runnable task) {
58          try {
59              executor.execute(task);
60          } catch (RejectedExecutionException e) {
61              System.out.println("Task executing was rejected.");
62              return false;
63          }
64          return true;
65      }
66
67      /**
68       * 提交任务，并可以在稍后获取其执行情况<br>
69       * 当提交失败时，会抛出 {@link }
70       * @param task
71       * @return
72       */
73      public static <T> Future<T> submitTask(Callable<T> task) {
74
75          try {
76              return executor.submit(task);
77          } catch (RejectedExecutionException e) {
78              throw new UnsupportedOperationException("Unable to submit the task,
    rejected.", e);
79          }
80      }
81  }
```

这个要根据实际情况来决定，比如最大容忍的响应时间，任务数，以及任务的复杂度来决定。这是一个不断积累的过程，公式反而不是很有用，因为服务器的环境是复杂的，我们其实可以通过压测来进行评估。

#3、线程同步

这些类为JUC包，它们都起到线程同步作用

#1、CountDownLatch （倒计时器）

这个类常常用于等待，等多个线程执行完毕，再让某个线程执行。

CountDownLatch的典型用法就是：某一线程在开始运行前等待n个线程执行完毕。

使用方法如下：

1. 将 CountDownLatch 的计数器初始化为n ： new CountDownLatch(n)，
2. 每当一个任务线程执行完毕，就将计数器减1 countdownlatch.countDown()，当计数器的值变为0时，

   在CountDownLatch上 await() 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

```
1   public class CountDownLatchTest {
2
```

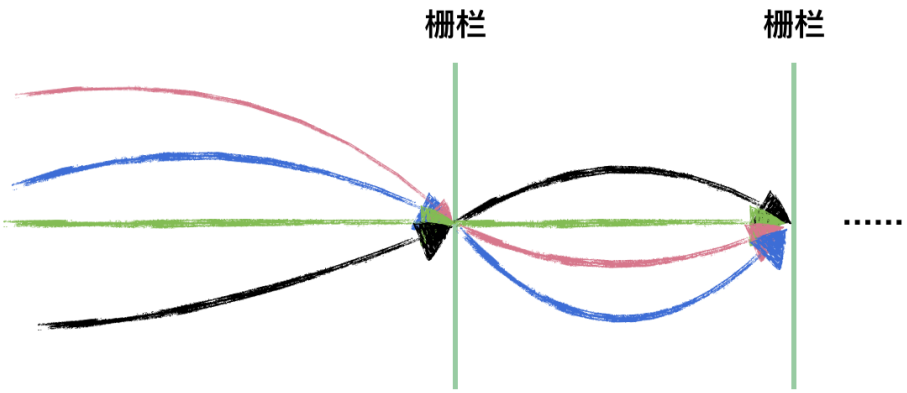
```
3       public static void main(String[] args) throws InterruptedException {
4
5           ExecutorService pool = Executors.newCachedThreadPool();
6           CountDownLatch countDownLatch = new CountDownLatch(3);
7
8           Runnable task1 = () -> {
9               ThreadUtils.sleep(new Random().nextInt(5000));
10              System.out.println("计算山西分公司的账目");
11              countDownLatch.countDown();
12          };
13          Runnable task2 = () -> {
14              ThreadUtils.sleep(new Random().nextInt(5000));
15              System.out.println("计算北京分公司的账目");
16              countDownLatch.countDown();
17          };
18          Runnable task3 = () -> {
19              ThreadUtils.sleep(new Random().nextInt(5000));
20              System.out.println("计算上海分公司的账目");
21              countDownLatch.countDown();
22          };
23          pool.submit(task1);
24          pool.submit(task2);
25          pool.submit(task3);
26          countDownLatch.await();
27          System.out.println("计算总账！");
28
29      }
30  }
```

CountDownLatch是一次性的，计数器的值只能在构造方法中初始化一次，之后没有任何机制再次对其设置值，当CountDownLatch使用完毕后，它不能再次被使用。

#2、CyclicBarrier(循环栅栏)

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

看 如下示意图，CyclicBarrier 和 CountDownLatch 是不是很像，只是 CyclicBarrier 可以有不止一个栅栏，因为它的栅栏（Barrier）可以重复使用（Cyclic）。




```
public class CyclicBarrierTest {

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

        ExecutorService pool = Executors.newCachedThreadPool();
        // 计算总账的主线程
        Runnable main = () -> System.out.println("计算总账！");
        CyclicBarrier cyclicBarrier = new CyclicBarrier(3,main);

        Runnable task1 = () -> {
            ThreadUtils.sleep(new Random().nextInt(5000));
            System.out.println("计算山西分公司的账目");
            try {
                cyclicBarrier.await();
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        };
        Runnable task2 = () -> {
            ThreadUtils.sleep(new Random().nextInt(5000));
            System.out.println("计算北京分公司的账目");
            try {
                cyclicBarrier.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        };
        Runnable task3 = () -> {
            ThreadUtils.sleep(new Random().nextInt(5000));
            System.out.println("计算上海分公司的账目");
            try {
                cyclicBarrier.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        };
        pool.submit(task1);
        pool.submit(task2);
        pool.submit(task3);

        // 重复利用
        ThreadUtils.sleep(5000);
        cyclicBarrier.reset();
        System.out.println("-------------reset-----------");
        pool.submit(task1);
        pool.submit(task2);
        pool.submit(task3);
    }
}
```

> CyclicBarrier与CountDownLatch的区别

至此我们难免会将CyclicBarrier与CountDownLatch进行一番比较。这两个类都可以实现一组线程在到达某个条件之前进行等待，它们内部都有一个计数器，当计数器的值不断的减为0的时候所有阻塞的线程将会被唤醒。

有区别的是CyclicBarrier的计数器由自己控制，而CountDownLatch的计数器则由使用者来控制，在CyclicBarrier中线程调用await方法不仅会将自己阻塞还会将计数器减1，而在CountDownLatch中线程调用await方法只是将自己阻塞而不会减少计数器的值。

另外，CountDownLatch只能拦截一轮，而CyclicBarrier可以实现循环拦截。一般来说用CyclicBarrier可以实现CountDownLatch的功能，而反之则不能。总之，这两个类的异同点大致如此，至于何时使用CyclicBarrier，何时使用CountDownLatch，还需要读者自己去拿捏。

#3、Semaphore(信号量)

`java.util.concurrent`t包中有`Semaphore`的实现，可以设置参数，控制同时访问的个数。

下面的Demo中申明了一个只有5个许可的Semaphore，而有20个线程要访问这个资源，通过acquire()和release()获取和释放访问许可。

```
public class SemaphoreTest {

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

        final Semaphore semaphore = new Semaphore(5);
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int index = 0; index < 100; index++) {
            Runnable run = () -> {
                try {
                    // 获取许可
                    semaphore.acquire();
                    System.out.println("开进一辆车...");
                    Thread.sleep((long) (Math.random() * 5000));
                    // 访问完后，释放
                    semaphore.release();
                    System.out.println("离开一辆车...");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            };
            exec.execute(run);
        }
        exec.shutdown();
    }
}
```

最后的结果是开始五辆车全部进入，因为停车场是空的，后边就是出一辆进一辆了。

#五、单例

> 懒汉模式

线程不安全，延迟初始化，严格意义上不是单例模式

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

> 饿汉模式

线程安全，比较常用，但容易产生垃圾，因为一开始就初始化

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
        return instance;
    }
}
```

> 双重锁模式，双重检查（重点）

线程安全，延迟初始化。这种方式采用双锁机制，安全且在多线程情况下能保持高性能。

```
 public class Singleton {
     // volatile如果不加可能会出现半初始化的对象
     private volatile static Singleton singleton;
     private Singleton (){}

     public static Singleton getSingleton() {
         if (singleton == null) {
             synchronized (Singleton.class) {
                 if (singleton == null) {
                     singleton = new Singleton();
                 }
             }
         }
         return singleton;
         }
}
```