JAVA多线程入门

B站 IT楠老师， 公众号： IT楠说java， QQ群： 1083478826

一、进程和线程

1、进程

是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。

例如：正在运行的 QQ、IDE、浏览器就是进程

2、线程

具体执行任务的最小单位， 一个进程中至少有一个线程。

2.1 进程与线程的联系

序号	进程与线程的联系
(1)	一个进程最少拥有一个线程--**主线程**--- 运行起来就执行的线程
(2)	线程之间是共享内存资源的（这个内存资源是由进程申请的）
(3)	线程之间可以通信（进行数据传递：多数为主线程和子线程）

2.2 创建子线程的原因

如果在主线程中存在有比较耗时的操作（例如：下载视频、上传文件 数据处理） 这些操作会阻塞主线程，后面的任务必须等这些任务执行完毕之后才能执行，用户体验比较差， 为了不阻塞主线程，需要将耗时的任务放在子线程去处理。

2.3 创建线程的方法

创建线程有两种方法

2.3.1 继承Thread类实现run方法

步骤：

① 定义类继承Thread;

② 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）

③ 调用线程的start方法：(该方法有两步：启动线程，调用run方法)

```java
/**
 * @author IT楠老师
 * @date 2020/6/17
 */
public class UseThread {
    public static void main(String[] args) {
        System.out.println(1);
        System.out.println(2);
        new MyTask().start();
        System.out.println(3);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(4);
    }

    static class MyTask extends Thread{
        @Override
        public void run() {
            System.out.println("这是通过继承实现的多线程！");
        }
    }
}
```

上面一段demo中所用到的方法如下：

方法	作用
`join()`	让当前这个线程阻塞 等join()的线程执行完毕再执行
`setName()`	设置线程名称
`getName()`	获取线程名称
`currentTHread()`	获取当前运行的线程对象

2.3.2 实现Runnable接口

注意事项： 接口应该由那些打算通过某一线程执行其实例的类来实现。类必须定义一个称为run 的无参方法。

> **步骤： ① 创建任务： 创建类实现Runnable接口 ② 使用Thread 为这个任务分配线程 ③ 开启任务`Start()`**

使用方式1---创建一个类实现Runnable接口，使用Thread来操作这个任务

```
/**
 * @author IT楠老师
 * @date 2020/6/17
 */
public class UseRunnable {

    public static void main(String[] args) {

        System.out.println(1);
        System.out.println(2);
        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("这是我的第一个线程方法");
        }
    }

}
```

2.3.3 使用Lammbda表达式

```
  new Thread(()->{
            for(int i=0;i<=100;i++){
                System.out.println(Thread.currentThread().getName()+":"+i);
            }
        }).start();
```

2.4.4 有返回值的线程

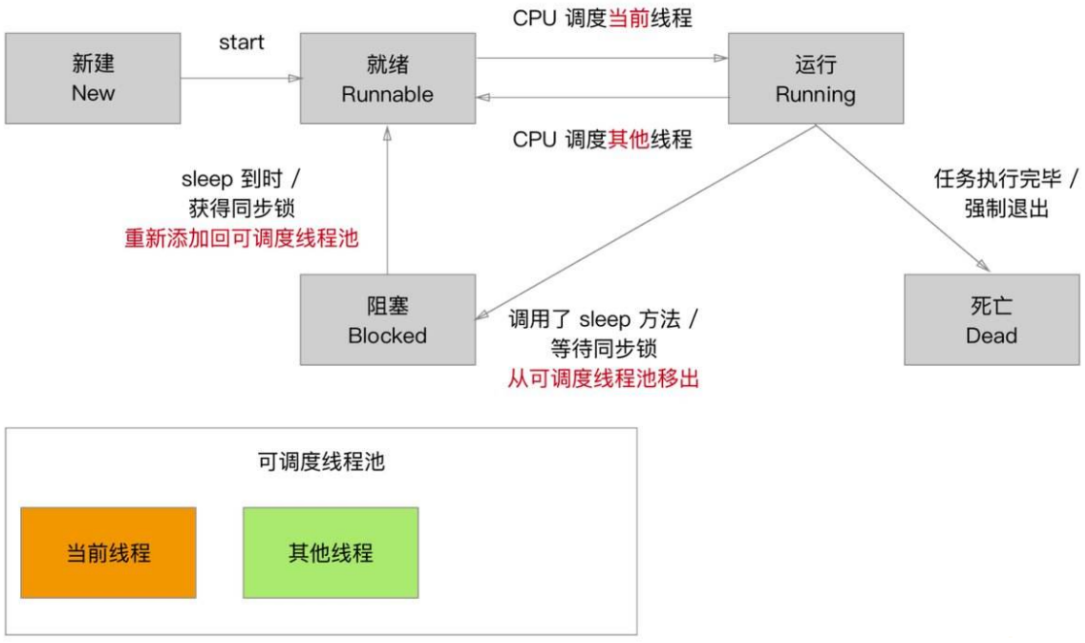
```
/**
 * @author IT楠老师
 * @date 2020/6/17
 */
public class UseCallable {

    public static void main(String[] args) throws Exception {
        System.out.println(2);
        FutureTask<Integer> futureTask = new FutureTask<Integer>(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;
        }
    }
}
```

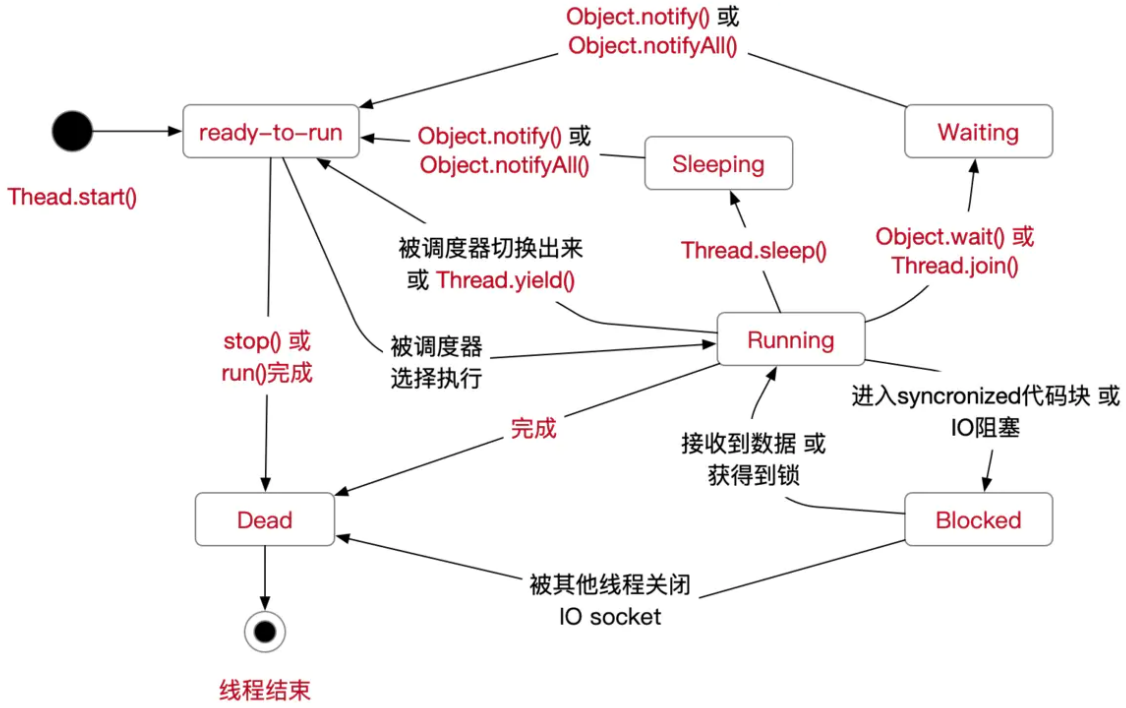
2.4 线程生命周期

一个线程的生命周期中，总共有以下6种状态

- **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，它可能正在等待来自操作系统(如处理器)的其他资源。
- **BLOCKED** - 线程被阻塞等待一个monitor锁，处于阻塞状态的线程正在等待monitor锁进入synchronized的代码块或方法，或者在调用Object.wait()方法后重新进入synchronized的代码块或方法。
- **WAITING** - 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。还有一种情况，处于等待状态的线程正在等待另一个线程执行特定的操作，比如：一个线程调用了`Object.wait()`后，等待另一个线程调用`Object.notifyAll()`或`Object.notify()`方法；或一个线程调用了`Thread.join()`方法，等待自己的线程的结束。
- **TIMED_WAITING** - 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。



Java线程的生命周期



join

```
/**
 * @author IT楠老师
 * @date 2020/6/30
 */
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("------------------------------------------------");
        ThreadUtils.sleep(100000);
```

```
    }
}
```

wait和notify

```
/**
 * @author IT楠老师
 * @date 2020/6/30
 */
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("------------------------------------------------");
        ThreadUtils.sleep(100000);
    }

    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);
        }
    }
}
```

总结

1. sleep、yield方法是静态方法；作用的是`当前执行的线程`；
2. yield方法释放了cpu的执行权，但是依然保留了cpu的执行资格。给个简单的例子：很多人排队上WC，刚好排上yield上了，现在yield说，出让它这次机会，与更急的人一起比赛谁能更快进入到WC中去。这个比赛可能是其他的人，也可能就是yield本身；
3. wait释放CPU资源，同时释放锁；
4. sleep释放CPU资源，但不释放锁；
5. 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。

2.5 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`。

用户线程是高优先级线程。**JVM将在终止任务之前等待任何用户线程完成其任务。**

另一方面，**守护线程是低优先级线程，其唯一作用是为用户线程提供服务。**

由于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，因此它们都不会退出JVM，直到所有用户线程执行完成。

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。

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。

```
/**
 * @author IT楠老师
 * @date 2020/6/30
 */
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();
    }

}
```

应用场景：

（1）来为其它线程提供服务支持的情况；（2）或者在任何情况下，程序结束时，这个线程必须正常且立刻关闭，就可以作为守护线程来使用；反之，如果一个正在执行某个操作的线程必须要正确地关闭掉否则就会出现不好的后果的话，那么这个线程就不能是守护线程，而是用户线程。通常都是些关键的事务，比方说，数据库录入或者更新，这些操作都是不能中断的。

例如1.

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

例如2.

如果 JVM 中没有一个正在运行的非守护线程，这个时候，JVM 会退出。换句话说，**守护线程拥有自动结束自己生命周期的特性，而非守护线程不具备这个特点。**JVM 中的垃圾回收线程就是典型的守护线程，如果说不具备该特性，会发生什么呢？当 JVM 要退出时，由于垃圾回收线程还在运行着，导致程序无法退出，这就很尴尬了！！！由此可见，守护线程的重要性了。

生活举例

你一边敲键盘，这是个 非守护线程， 后台还有一个 拼写检查 线程，它是个守护线程，他尽量不打扰你写稿子， 你们可以同时进行，他发现有拼写错误时在状态条显示错误，但是你可以忽略。

就像 城堡门前有个卫兵 （守护线程），里面有诸侯（非守护线程），他们是可以同时干着各自的活儿，但是 城堡里面的人都搬走了， 那么卫兵也就没有存在的意义了。

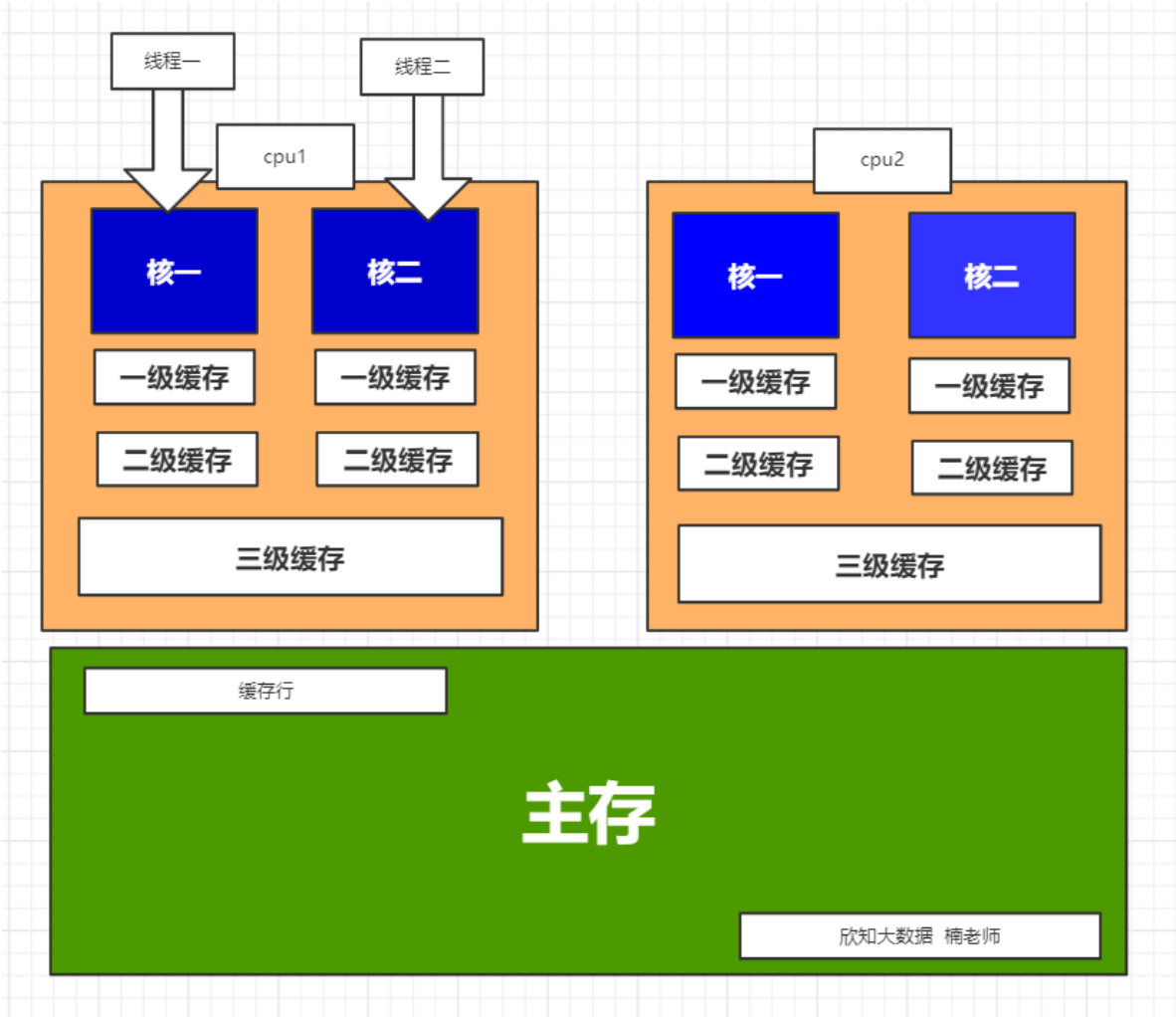
二、线程安全的讨论

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

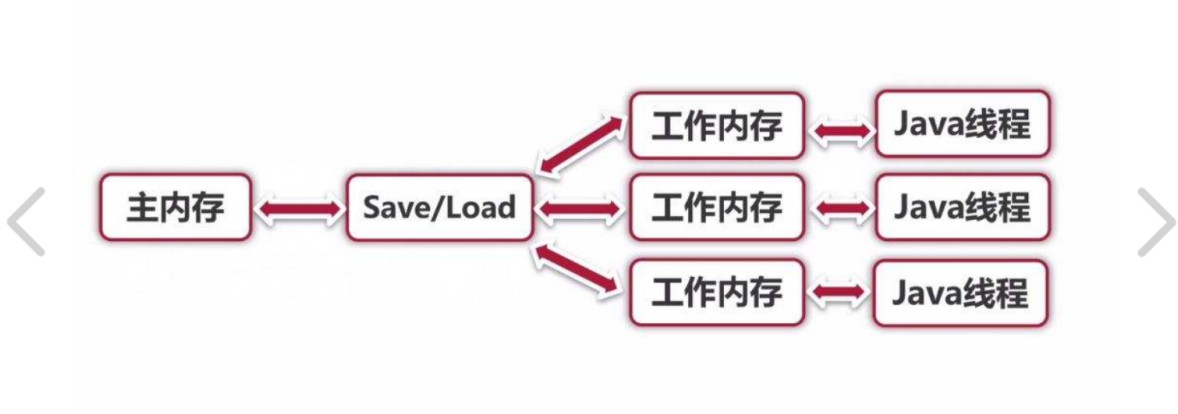
从CPU到	大约需要的 CPU 周期	大约需要的时间
主存		约60-80纳秒
QPI 总线传输 (between sockets, not drawn)		约20ns
L3 cache	约40-45 cycles,	约15ns
L2 cache	约10 cycles,	约3ns
L1 cache	约3-4 cycles,	约1ns
寄存器	1 cycle	

缓存一致性协议 因为每个CPU都有自己的缓存，容易导致一种情况就是 如果多个CPU的缓存(多CPU读取同样的数据进行缓存，进行不同运算后，写入内存中)中都有同样一份数据，那这个数据要如何处理呢？已谁的为准？ 这个时候就需要一个缓存同步协议了！

```
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：
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
专有态 (Exclusive) 此缓存与主内存一致，但是其他CPU中没有
共享态 (Shared) 此缓存与住内存一致，但也出现在其他缓存中。
无效态 (Invalid) 此缓存无效，需要从主内存中重新读取。
```

指令重排：当CPU 写缓存 时发现缓存区被其他CPU占用，为了提高CPU处理性能，可能将后面的读缓存命令优先执行。 指令重排序，遵循 as-if-serial语义。即指令重排序前后，程序执行的结果不能变化。对于数据有依赖的部分，不会进行重排序。

JMM



2、导致的问题

（1）伪共享：

缓存中的数据与主内存的数据不是实时同步的，各个CPU间缓存的数据也不是实时同步的，在同一时间点，各个CPU所看到的的同一内存地址的数据可能是不一致的

（2）CPU指令重排问题：

多核多线程，指令逻辑无法分辨因果关联，可能出现乱序执行，导致程序结果出现错误。

（3）线程安全

来看一个小例子：

```
/**
 * @author IT楠老师
 * @date 2020/6/30
 */
public class Ticket implements Runnable{
    String name;

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

    @Override
    public void run() {
        while (Constant.count > 0) {
            ThreadUtils.sleep(100);
            System.out.println(name + "出票一张，还剩"+ Constant.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();
        Thread.sleep(10000);
    }
}
```

得到的结果是：

提供两种方式解决方式

同步代码块的第一种方式---synchronized（参数就是一个监听器）

```
synchronized(监听器/对象/对象的一把锁）{
              //需要同步的代码
}
```

```
/**
 * @author IT楠老师
 * @date 2020/6/30
 */
public class Ticket implements Runnable{

    private static final Object monitor = new Object();

    String name;

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

    @Override
    public void run() {
        while (Constant.count > 0) {
            ThreadUtils.sleep(100);
            synchronized (Ticket.monitor) {
                System.out.println(name + "出票一张，还剩" + Constant.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();
        Thread.sleep(10000);
    }
}
```

同步代码块的第二种方式---ReentrantLock

```
/**
 * @author IT楠老师
 * @date 2020/6/30
 */
public class Ticket2 implements Runnable {

    private static ReentrantLock lock = new ReentrantLock();

    String name;

    public Ticket2(String name) {
```

```
        this.name = name;
    }

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

    public static void main(String[] args) throws Exception {
        Thread one = new Thread(new Ticket2("一号窗口"));
        Thread two = new Thread(new Ticket2("一号窗口"));
        one.start();
        two.start();
        Thread.sleep(10000);
    }
}
```

三、java中的锁

上边的列子中，我们看到了synchronized的作用。

3.1 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。这块在后续介绍中会慢慢引入。

synchronized的基本语法

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。

我在网上找了张图，大致也对应上面所说的。

分类	具体分类	被锁的对象	伪代码

方法	实例方法	类的实例对象	//实例方法，锁住的是该类的实例对象
			public synchronized void method() {
			.....
			}

	静态方法	类对象	//静态方法，锁住的是类对象
			public static synchronized void method1(){
			.....
			}

代码块	实例对象	类的实例对象	//同步代码块，锁住的是该类的实例对象
			synchronized (this){
			.....
			}

	class对象	类对象	//同步代码块，锁住的是该类的类对象
			synchronized (SynchronizedDemo.class){
			.....
			}

	任意实例对象Object	实例对象Object	//同步代码块，锁住的是配置的实例对象
			//String对象作为锁
			String lock = "";
			synchronized (lock){
			.....
			}

3.2 synchronized原理分析

Java对象头和monitor是实现synchronized的基础!

3.2.1 monitor

```
package com.thread;

public class Demo1{

    private static int count = 0;
    public static void main(String[] args) {
        synchronized (Demo1.class) {
            inc();
        }
    }
    private static void inc() {
        count++;
    }
}
```

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`

```
15: astore_1
16: monitorenter
17: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
20: new           #7                  // class java/lang/StringBuilder
```

```
67: astore_2
68: aload_1
69: monitorexit
70: aload_2
```

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。

对象在内存中的布局

在 Hotspot 虚拟机中，对象在内存中的存储布局，可以分为三个区域:对象头(Header)、实例数据(Instance Data)、对齐填充(Padding)。一般而言，synchronized使用的锁对象是存储在Java对象头里。它是轻量级锁和偏向锁的关键。

对象头

- HotSpot虚拟机的对象头包括用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的长度在32位和64位的虚拟机（未开启压缩指针）中分别为32bit和64bit，官方称它为“MarkWord”
- class类型指针,指向这个对象属于哪个Class对象
- 数组长度（只有数组对象有） 如果对象是一个数组, 那在对象头中还必须有一块数据用于记录数组长度.

实例数据

实例数据部分是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类继承下来的，还是在子类中定义的，都需要记录起来。

对齐填充

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于HotSpot VM的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说，就是对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的倍数（1倍或者2倍），因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

对象大小计算

要点

1. 在32位系统下，存放Class指针的空间大小是4字节，MarkWord是4字节，对象头为8字节。
2. 在64位系统下，存放Class指针的空间大小是8字节，MarkWord是8字节，对象头为16字节。
3. 64位开启指针压缩的情况下，存放Class指针的空间大小是4字节，MarkWord是8字节，对象头为12字节。 数组长度4字节+数组对象头8字节(对象引用4字节（未开启指针压缩的64位为8字节）+数组markword为4字节（64位未开启指针压缩的为8字节）)+对齐4=16字节。
4. 静态属性不算在对象大小内。

3.2.2 Java对象头

对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。 **Mark Word**：用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等,它是实现轻量级锁和偏向锁的关键.

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

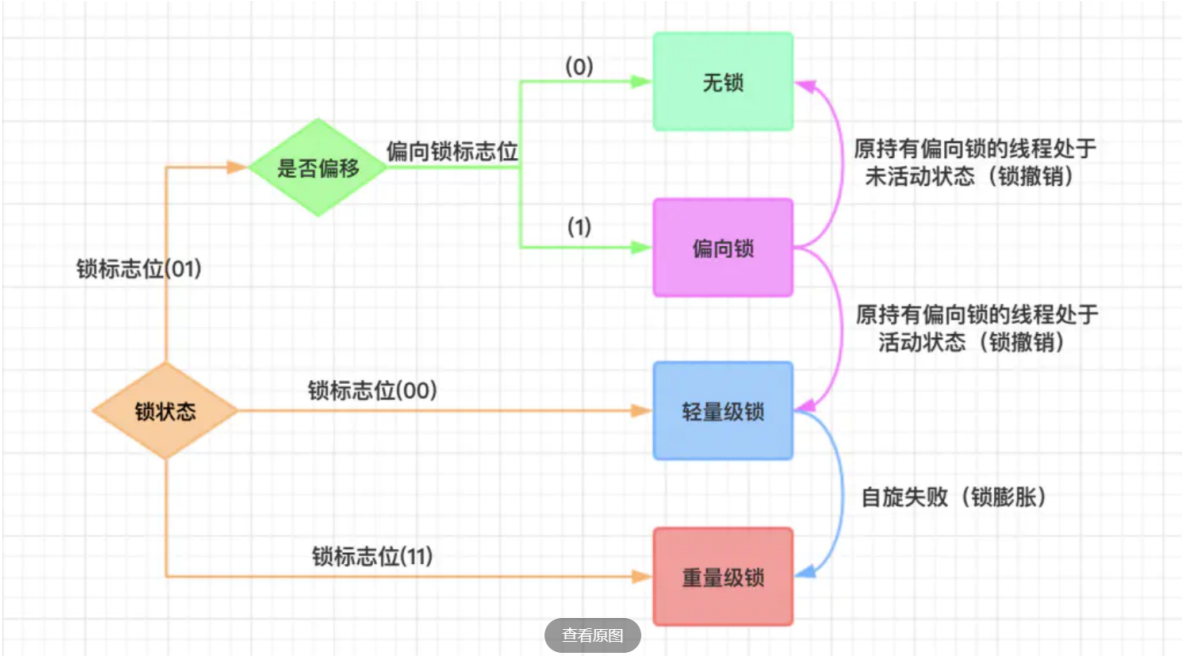
加锁时MarkWord可能储存的四种状态

3.3 synchronized 锁的升级

在 Java 语言中，使用 Synchronized 是能够实现线程同步的，即**加锁**。并且实现的是**悲观锁**，在操作同步资源的时候直接先加锁。

加锁可以使一段代码在同一时间只有一个线程可以访问，在增加安全性的同时，牺牲掉的是程序的执行性能，所以为了在一定程度上减少获得锁和释放锁带来的性能消耗，在 jdk6 之后便引入了“偏向锁”和“轻量级锁”，所以总共有4种锁状态，级别由低到高依次为：**无锁状态**、**偏向锁状态**、**轻量级锁状态**、**重量级锁状态**。这几个状态会随着竞争情况逐渐升级。

注意：锁可以升级但不能降级。



因为线程切换是非常重量级的因此JDK1.6开始对synchronized做了优化，通过上文讲的Mark World 来区分了不同场景下同步锁的不同类型，来减少线程切换的次数.

第一步：偏向锁

偏向锁的作用是当有线程访问同步代码或方法时，线程只需要判断对象头的Mark Word中判断一下是否有偏向锁指向线程ID.

偏向锁记录过程

- 线程抢到了对象的同步锁(锁标志为01参考上图即无其他线程占用)
- 对象Mark World 将是否偏向标志位设置为1
- 记录抢到锁的线程ID
- 进入偏向状态

锁状态	31bit(25bit unused)		4bit	1bit	2bit
	54bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向轻量级锁的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空				11

偏向锁的优势

通过加偏向锁的方式可以看到，对象中记录了获取到对象锁的线程ID，这就意味如果`短时间同一个线程再次访问`这个加锁的同步代码或方法时，该线程只需要对对象头Mark Word中去判断一下是否有偏向锁指向它的ID，不需要在进入Monitor去竞争对象了。

顾名思义：就是这把锁偏像某个线程。

什么时候升级成轻量级锁?

偏向锁 -> 轻量级锁 -> 重量级锁

一旦出现其他线程竞争资源时，偏向锁就会被撤销。

偏向锁的插销需要等待全局安全点，暂停持有该锁的线程，同时检查该线程是否还在执行该方法，如果是，则升级锁。反之则其他线程抢占。

第二部：轻量级锁

当有另外一个线程竞争获取这个锁时，由于该锁已经是偏向锁，当发现对象头 Mark Word 中的线程 ID 不是自己的线程 ID，就会尝试获取锁，**如果获取成功**，直接替换 Mark Word 中的线程 ID 为自己的 ID，该锁会保持偏向锁状态；**如果获取锁失败**，代表当前锁有一定的竞争，偏向锁将升级为轻量级锁。

第一个前来获取：记录了偏向的线程

第二个过来尝试获取，如果成功了，说明第一个线程已经不再使用，锁则偏向第二个线程

如果失败，说明存在竞争,升级轻量级锁

第三部：自旋锁

JVM 提供了一种自旋锁，可以通过自旋方式不断尝试获取锁，从而避免线程被挂起阻塞。这是基于大多数情况下，线程持有锁的时间都不会太长，毕竟线程被挂起阻塞可能会得不偿失。

从 JDK1.7 开始，自旋锁默认启用，自旋次数由 JVM 设置决定，这里我不建议设置的重试次数过多，因为 CAS 重试操作意味着长时间地占用 CPU。自旋锁重试之后如果抢锁依然失败，同步锁就会升级至重量级锁，锁标志位改为 10。在这个状态下，未抢到锁的线程都会进入 Monitor，之后会被阻塞在 _WaitSet 队列中。默认的自旋次数是十次

第四部：重量级锁

自旋失败，很大概率 再一次自选也是失败，因此直接升级成重量级锁，进行线程阻塞，减少cpu消耗。

当锁升级为重量级锁后，未抢到锁的线程都会被阻塞，进入阻塞队列。

> **总结**

synchronized锁升级实际上是把本来的悲观锁变成了 在一定条件下 使用无锁(同样线程获取相同资源的偏向锁)，以及使用乐观(自旋锁 cas)和一定条件下悲观(重量级锁)的形式。

偏向锁:适用于单线程适用锁的情况

轻量级锁：适用于竞争较不激烈的情况(这和乐观锁的使用范围类似)

重量级锁：适用于竞争激烈的情况

3.4 Lock

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，即 可重入锁。ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法。下面通过一些实例学习如何使用 ReentrantLock。

用法上边已经讲了

3.5 ReadWriteLock

```
public class ReadAndWriteLockTest {
```

```
    public static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

    public static void main(String[] args) {
        //同时读、写
        ExecutorService service = Executors.newCachedThreadPool();
        service.execute(new Runnable() {
            @Override
            public void run() {
                readFile(Thread.currentThread());
            }
        });
        service.execute(new Runnable() {
            @Override
            public void run() {
                writeFile(Thread.currentThread());
            }
        });
    }

    // 读操作
    public static void readFile(Thread thread) {
        lock.readLock().lock();
        boolean readLock = lock.isWriteLocked();
        if (!readLock) {
            System.out.println("当前为读锁！");
        }
        try {
            for (int i = 0; i < 5; i++) {
                try {
                    Thread.sleep(20);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(thread.getName() + ":正在进行读操作……");
            }
            System.out.println(thread.getName() + ":读操作完毕！");
        } finally {
            System.out.println("释放读锁！");
            lock.readLock().unlock();
        }
    }

    // 写操作
    public static void writeFile(Thread thread) {
        lock.writeLock().lock();
        boolean writeLock = lock.isWriteLocked();
        if (writeLock) {
            System.out.println("当前为写锁！");
        }
        try {
            for (int i = 0; i < 5; i++) {
                try {
                    Thread.sleep(20);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(thread.getName() + ":正在进行写操作……");
            }
```

```
                System.out.println(thread.getName() + ":写操作完毕！");
            } finally {
                System.out.println("释放写锁！");
                lock.writeLock().unlock();
            }
        }
    }
```

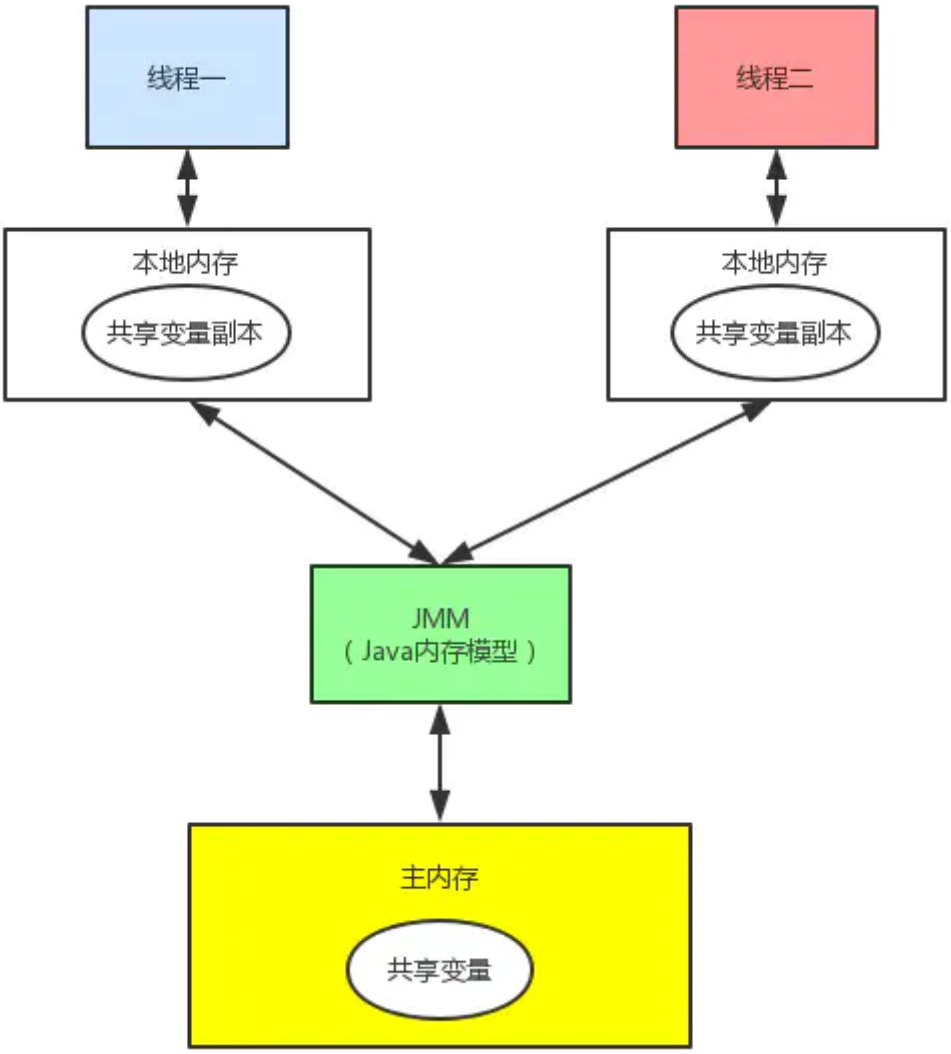
四、补充

4.1 volatile

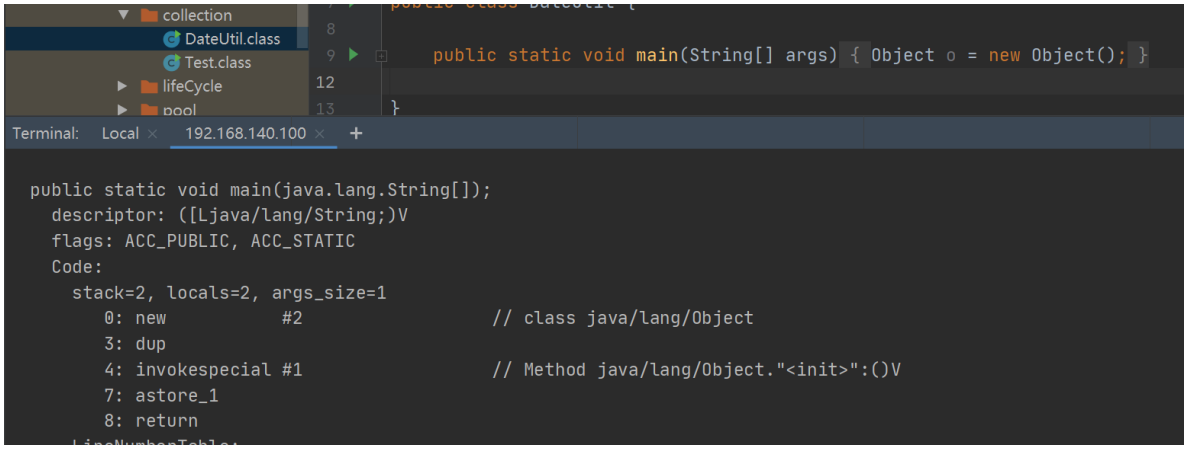
volatile：Java虚拟机提供的轻量级的同步机制

- 保证可见性（一个线程修改了主内存的值，其他线程立即得知改变），线程访问此变量会从主内存中去读写，而不是线程自己的缓存中
- 不保证原子性
- 禁止指令重排

JMM（Java内存模型），工作内存中存储着主内存的**变量副本拷贝**。工作内存是每个线程的私有区域，JMM规定所有变量都存储在主内存，主内存是共享内存区域，所有线程均可访问，线程对变量的操作（读取赋值等）只能在工作内存操作，首先将变量从主内存拷贝到自己的工作内存空间，然后对变量操作，操作完成后**将变量写回主内存**，不能直接操作主内存的变量。



javap -v DateUtil.class



五、线程池

1、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

```
/**
 * @author IT楠老师
 * @date 2020/6/17
 */
public class UseExecutors {
    public static void main(String[] args) {
        Runnable taskOne = () -> System.out.println("taskOne");
        Runnable taskTwo = () -> System.out.println("taskTwo");
        ExecutorService pools = Executors.newFixedThreadPool(10);
        for (int i = 0; i < 40; i++) {
            pools.submit(taskOne);
        }
        pools.execute(taskTwo);
    }
}
```

无论是哪一个都是调用ThreadPoolExecutor 构造方法：

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

2、参数的意义-重要

1. corePoolSize 指定了线程池里的线程数量，核心线程池大小
2. maximumPoolSize 指定了线程池里的最大线程数量
3. 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。
4. unit 时间单位。TimeUnit
5. workQueue 任务队列，用于存放提交但是尚未被执行的任务。

我们可以选择如下几种：

- 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。
- SynchronousQueue：不存储元素的阻塞队列，每个插入操作都必须等待一个移出操作，反之亦然。
- PriorityBlockingQueue：具有优先级别的阻塞队列。

6. threadFactory 线程工厂，用于创建线程，一般可以用默认的
7. handler 拒绝策略，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。

   什么时候拒绝？当向线程池中提交任务时，如果此时线程池中的线程已经饱和了，而且阻塞队列也已经满了，则线程池会选择一种拒绝策略来处理该任务，该任务会交给 RejectedExecutionHandler 处理。

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：丢弃阻塞队列中靠最前的任务，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

- 当线程数小于核心线程数时，创建线程。
- 当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。
- 当线程数大于等于核心线程数，且任务队列已满

- 若线程数小于最大线程数，创建线程 - 若线程数等于最大线程数，抛出异常，拒绝任务

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).

阿里不让用

其实看了上面的Excutors的五个方法后，在阿里规约里面时不可以使用除第一个外的四个包内线程的，因为他们都会引起OOM。 从源码我们可以看出都是有ThreadPoolExcutor这个类传入不同的参数而实现的所以说只要我们搞懂其中传入的7个参数的含义，就可以大概搞懂线程池的冰山一角了

```
package com.xinzhi.pool;

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * @author IT楠老师
 * @date 2020/6/17
 */
public class ManualBuild {

    public static void main(String[] args) throws InterruptedException {
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
                4,
                8,
                5,
                TimeUnit.SECONDS,
                new LinkedBlockingDeque<>(300),
                new MyThreadFactory(new ThreadGroup("xinzhi"),"goods-thread"),
                new ThreadPoolExecutor.AbortPolicy()
        );

        while (true){
            Thread.sleep(1000);
            threadPoolExecutor.execute(()->{
                System.out.println(123);
            });
        }
```

```
    }

    static class MyThreadFactory implements ThreadFactory{

        private AtomicInteger number = new AtomicInteger(0);
        private ThreadGroup group;
        private String namePrefix;

        public MyThreadFactory(ThreadGroup group, String namePrefix) {
            this.group = group;
            this.namePrefix = namePrefix;
        }

        @Override
        public Thread newThread(Runnable r) {
            Thread t = new Thread(group,r,namePrefix+"-
"+number.getAndIncrement());
            if (t.isDaemon()){
                t.setDaemon(false);
            }
            if (t.getPriority() != Thread.NORM_PRIORITY){
                t.setPriority(Thread.NORM_PRIORITY);
            }
            return t;
        }
    }

}
```

3、如何设置参数

（1）需要根据几个值来决定

- tasks：每秒的任务数，假设为500~1000
- taskcost：每个任务花费时间，假设为0.1s
- responsetime：系统允许容忍的最大响应时间，假设为1s

（2）做几个计算

- corePoolSize = 每秒需要多少个线程处理?
- threadcount = tasks/(1/taskcost) =tasks*taskcout* = *(500~1000)*0.1 = 50~100 个线程。
- corePoolSize设置应该大于50，根据8020原则，如果80%的每秒任务数小于800，那么corePoolSize设置为80即可
- queueCapacity = (coreSizePool/taskcost)*responsetime 计算可得 queueCapacity = 80/0.1*1 = 80。意思是队列里的线程可以等待1s，超过了的需要新开线程来执行 切记不能设置为Integer.MAX_VALUE，这样队列会很大，线程数只会保持在corePoolSize大小，当任务陡增时，不能新开线程来执 行，响应时间会随之陡增。
- maxPoolSize = (max(tasks)- queueCapacity)/(1/taskcost) 计算可得 maxPoolSize = (1000-80)/10 = 92 （最大任务数-队列容量）/每个线程每秒处理能力 = 最大线程数 rejectedExecutionHandler：根据具体情况来决定，任务不重要可丢弃，任务重要则要利用一些缓冲机制来处理 keepAliveTime和allowCoreThreadTimeout采用默认通常能满足

以上都是理想值，实际情况下要根据机器性能来决定。如果在未达到最大线程数的情况机器cpu load已经满了，则需要通过升级硬件（呵呵）和优化代码，降低taskcost来处理。

六、线程同步类

这些类为JUC包，他们都起到线程同步作用

6.1 CountDownLatch （倒计时器）

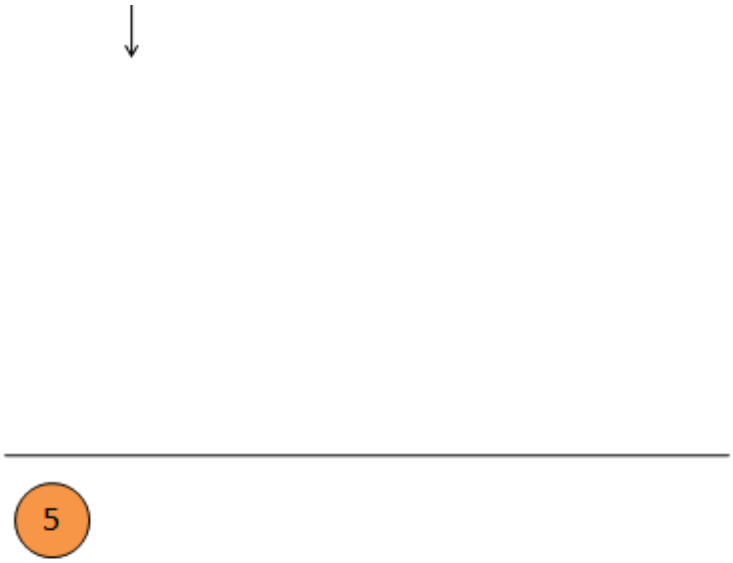
这个类常常用于等待，等多个线程执行完毕，再让某个线程执行。 CountDownLatch的典型用法就是：①某一线程在开始运行前等待n个线程执行完毕。将 CountDownLatch 的计数器初始化为n ： new CountDownLatch(n)，每当一个任务线程执行完毕，就将计数器减1 countdownlatch.countDown()，当计数器的值变为0时，在CountDownLatch上 await() 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

```
public class CountDownLatchExample1 {
    //请求的数量
    private static final int threadCount = 550;

    public static void main(String[] args) throws InterruptedException{
        ExecutorService executorService = Executors.newFixedThreadPool(300);
        final CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i=0;i<threadCount;i++){
            final  int threadNum = i;
            executorService.execute(() ->{
                try {
                    test(threadNum);
                }catch (InterruptedException e){
                    e.printStackTrace();
                }finally {
                    //表示一个请求已完成
                    countDownLatch.countDown();
                }

            });
        }
        //await之后的都要等待，等countDownLatch减到0，await之后的才能执行
        countDownLatch.await();
        executorService.shutdown();

    }
    public static void test(int threadCount)  throws InterruptedException{
        //模拟请求的耗时操作
        Thread.sleep(1000);
        System.out.println("threadCount:"+threadCount);
        Thread.sleep(1000);
    }

}
```

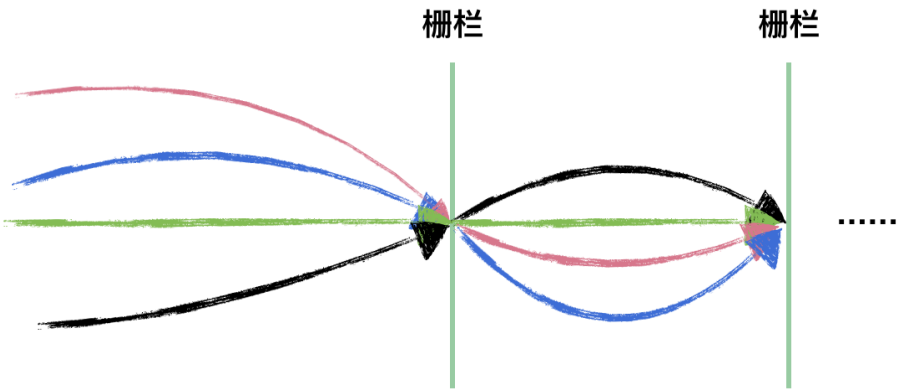
CountDownLatch是一次性的，计数器的值只能在构造方法中初始化一次，之后没有任何机制再次对其设置值，当CountDownLatch使用完毕后，它不能再次被使用。

6.2 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）。




```
 * @date 2020/6/30
 */
public class CyclicBarrierTest {
```

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

        ExecutorService pool = Executors.newCachedThreadPool();

        Runnable main = () -> System.out.println("执行主进程！");
        CyclicBarrier cyclicBarrier = new CyclicBarrier(2,main);
        Runnable sub1 = ()->{
            for (int i = 0; i < 20; i++) {
                ThreadUtils.sleep(100);
                System.out.println("r1");
            }
            try {
                cyclicBarrier.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        };

        Runnable sub2 = ()->{
            for (int i = 0; i < 10; i++) {
                ThreadUtils.sleep(100);
                System.out.println("r2");
            }
            try {
                cyclicBarrier.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        };

        pool.submit(sub1);
        pool.submit(sub2);

    }
}
```

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。

除此之外，CyclicBarrier还提供了：resert()、getNumberWaiting()、isBroken()等比较有用的方法。

6.3 Semaphore(信号量)

java.util.concurrent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。

```
final Semaphore semp = new Semaphore(5);
ExecutorService exec = Executors.newCachedThreadPool();
for (int index = 0; index < 20; index++) {
    final int NO = index;
    Runnable run = new Runnable() {
        public void run() {
            try {
                // 获取许可
                semp.acquire();
                System.out.println("Accessing: " + NO);
                Thread.sleep((long) (Math.random() * 10000));
                // 访问完后，释放
                semp.release();
                System.out.println("-----------------" +
semp.availablePermits());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };
    exec.execute(run);
}
exec.shutdown();
```

七、原子类

7.1 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。

JUC 原子类概览

7.2 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

7.3 讲讲 AtomicInteger 的使用

AtomicInteger 类常用方法

```
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

```
class AtomicIntegerTest {
```

```
    private AtomicInteger count = new AtomicInteger();

    public void increment() {
        // 使用 AtomicInteger 之后，不用对 `increment()` 方法加锁也可以保证线程安全
        count.incrementAndGet();
    }

    public int getCount() {
        return count.get();
    }

}
```

7.4 cas（重点理解）

全称（Compare And Swap）,比较并交换

Unsafe类是CAS的核心类，提供**硬件级别的原子操作。**

```
// 对象、对象的地址、预期值、修改值
public final native boolean compareAndSwapInt(Object var1, long var2, int var4,
int var5);
```

CAS(Compare-And-Swap) 算法是硬件对于并发的支持,针对多处理器操作而设计的处理器中的一种特殊指令,用于管理对共享数据的并发访问;

CAS 是一种无锁的非阻塞算法的实现;

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

> **CAS的缺点**

cas无法解决aba的问题。如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A - B - A 就会变成1A-2B - 3A。

从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

7.5 AtomicInteger 类原理

以 AtomicInteger 类为例，以下是部分源代码：

```
    // setup to use Unsafe. compareAndSwapInt for updates（更新操作时提供“比较并替换”的作用）
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

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

    private volatile int value;
```

AtomicInteger 类利用 `CAS (Compare and Swap) + volatile + native` 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS 的原理，是拿期望值和原本的值作比较，如果相同，则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是个本地方法，这个方法是用来拿“原值”的内存地址，返回值是 valueOffset；另外，value 是一个 volatile 变量，因此 JVM 总是可以保证任意时刻的任何线程总能拿到该变量的最新值。

八、单例

所谓单例，就是整个程序有且仅有一个实例。该类负责创建自己的对象，同时确保只有一个对象被创建。在Java，一般常用在工具类的实现或创建对象需要消耗资源。 **特点**

- 类构造器私有
- 持有自己类型的属性
- 对外提供获取实例的静态方法

懒汉模式

线程不安全，延迟初始化，严格意义上不是不是单例模式

```
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 {
    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;
    }
}
```

双重检查模式，进行了两次的判断，第一次是为了避免不要的实例，第二次是为了进行同步，避免多线程问题。由于`singleton=new Singleton()`对象的创建在JVM中可能会进行重排序，在多线程访问下存在风险，使用`volatile`修饰`signleton`实例变量有效，解决该问题。

静态内部类单例模式（常用）

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

只有第一次调用getInstance方法时，虚拟机才加载 Inner 并初始化instance ，只有一个线程可以获得对象的初始化锁，其他线程无法进行初始化，保证对象的唯一性。目前此方式是所有单例模式中最推荐的模式，但具体还是根据项目选择。

枚举单例模式

```
public enum Singleton {
    INSTANCE;
}
```

默认枚举实例的创建是线程安全的，并且在任何情况下都是单例。实际上

- 枚举类隐藏了私有的构造器。
- 枚举类的域 是相应类型的一个实例对象 那么枚举类型日常用例是这样子的：

```
public enum Singleton  {
    INSTANCE

    //doSomething 该实例支持的行为

    //可以省略此方法，通过Singleton.INSTANCE进行操作
    public static Singleton get Instance() {
        return Singleton.INSTANCE;
    }
}
```

枚举单例模式在《Effective Java》中推荐的单例模式之一。但枚举实例在日常开发是很少使用的，就是很简单以导致可读性较差。 在以上所有的单例模式中，推荐静态内部类单例模式。主要是非常直观，即保证线程安全又保证唯一性。