二、多线程系列问题

线程基础

线程其他问题

为什么使用多线程？

https://blog.csdn.net/CYTDCHE/article/details/79075013

实际上，多线程并不一定能提升性能（甚至还会降低性能）；多线程也不只是为了提升性能。多线程主要有以下的应用场景：

避免阻塞

避免 CPU 空转

提升性能

下面是应用场景详情：

1、避免阻塞（异步调用）

单个线程中的程序，是顺序执行的。如果前面的操作发生了阻塞，那么就会影响到后面的操作。这时候可以采用多线程，我感觉就等于是异步调用。这样的例子有很多：

ajax 调用，就是浏览器会启一个新的线程，不阻塞当前页面的正常操作；

流程在某个环节调用 web service，如果是同步调用，则需要等待 web service 调用结果，可以启动新线程来调用，不影响主流程；

android 里，不要在 ui thread 里执行耗时操作，否则容易引发 ANR；

创建工单时，需要级联往其他表中插入数据，可以将级联插入的动作放到新线程中，先返回工单创建的结果……

2、避免 CPU 空转

以 http server 为例，如果只用单线程响应 HTTP 请求，即处理完一条请求，再处理下一条请求的话，CPU 会存在大量的闲置时间

因为处理一条请求，经常涉及到 RPC、数据库访问、磁盘 IO 等操作，这些操作的速度比 CPU 慢很多，而在等待这些响应的时候，CPU 却不能去处理新的请求，因此 http server 的性能就很差

所以很多 web 容器，都采用对每个请求创建新线程来响应的方式实现，这样在等待请求 A 的 IO 操作的等待时间里，就可以去继续处理请求 B，对并发的响应性就好了很多

当然，这种设计方式并不是绝对的，现在像 node.js、Nginx 等新一代 http server，采用了事件驱动的实现方式，用单线程来响应多个请求也是没问题的。甚至实现了更高的性能，因为多线程是一把双刃剑，在提升了响应性的同时，创建销毁线程都是需要开销的，另外 CPU 在线程之间切换，也会带来额外的开销。避免了这些额外开销，可能是 node.js 等 http server 性能优秀的原因之一吧

3、提升性能

在满足条件的前提下，多线程确实能提升性能

打一个比方，多线程就相当于，把要炒的菜放到了不同的锅里，然后用不同的炉来炒，当然速度会比较快。本来需要先炒西红柿，10 分钟；再炒白菜 10 分钟；加起来就需要 20 分钟。用了多线程以后，分别放在 2 个锅里炒，10 分钟就都炒好了

基本上，需要满足 3 个条件：

第 1，任务具有并发性，也就是可以拆分成多个子任务。并不是什么任务都能拆分的，条件还比较苛刻

子任务之间不能有先后顺序的依赖，必须是允许并行的

比如

Java 代码

```
a = b + c
d = e + f
```

这个是可以并行的；

Java 代码

```
a = b + c

d = a + e
```

这个就无法并行了，第 2 步计算需要依赖第 1 步的计算结果，即使分成 2 个线程，也不会带来任何性能提升

另外，还不能有资源竞争。比如 2 个线程都需要写一个文件，第 1 个线程将文件锁定了，第 2 个线程只能等着，这样的 2 个子任务，也不具备并发性；执行 sychronized 代码，也是同样的情况

第 2，只有在 CPU 是性能瓶颈的情况下，多线程才能实现提升性能的目的。比如一段程序，瓶颈在于 IO 操作，那么把这个程序拆分到 2 个线程中执行，也是无法提升性能的

第 3，有点像废话，就是需要有多核 CPU 才行。否则的话，虽然拆分成了多个可并行的子任务，但是没有足够的 CPU，还是只有一个 CPU 在多个线程中切换来切换去，不但达不到提升性能的效果，反而由于增加了额外的开销，而降低了性能。类似于虽然把菜放到了 2 个锅里，但是只有 1 个炉子一样

如果上述条件都满足，有一个经验公式可以计算性能提升的比例，叫阿姆达尔定律：

速度提升比例 = 1/[(1-P)+(P/N)]，其中 P 是可并行任务的比例，N 是 CPU 核心数量

假设 CPU 核心是无限的，则公式简化为 1/(1-P)

假设 P 达到了 80%（已经非常高了），那么速度提升比例也只能达到 5 倍而已

线程是不是越多越好，一般设置多少个

线程不是越多越好

服务器 CPU 核数有限，能够同时并发的线程数有限，单核 CPU 设置 10000 个工作线程没有意义

线程切换是有开销的，如果线程切换过于频繁，反而会使性能降低

N 核服务器，通过执行业务的单线程分析出本地计算时间为 x，等待时间为 y，则工作线程数（线程池线程数）设置为 N*(x+y)/x，能让 CPU 的利用率最大化。

电脑 CPU 为 4 核。适合设置多少个线程

线程的工作区

Java 的内存模型分为主内存，和工作内存。

主内存是所有的线程所共享的，工作内存是每个线程自己有一个，不是共享的。

线程工作时将要用到的变量从主内存拷贝到自己的工作内存，然后在工作内存中进行读和写。

写完之后，可能没被更新到主内存去。导致其他线程从主内存拷贝数据到自己的工作区时，拷贝的不是最新的数据。这就是内存可见性问题。

线程的优先级

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。

Java 线程的优先级是一个整数，其取值范围是 1 （Thread.MIN_PRIORITY ） - 10 （Thread.MAX_PRIORITY ）。

默认情况下，每一个线程都会分配一个优先级 NORM_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

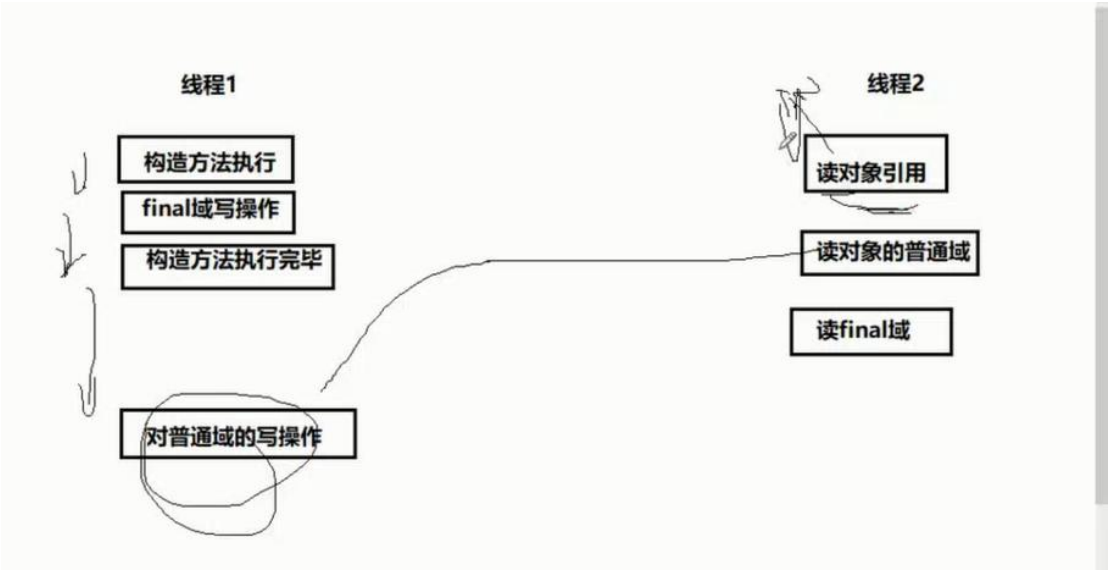
final 使用

写 final 域的重排序规则：

写 final 域的重排序的规则禁止把 final 域的写 重排序到构造方法之外。

Final 域在多线程中较普通变量的优势；

因为 final 域中的变量不会被 java 虚拟机重排序到构造方法之外，所以在另一个线程读取 final 域中的变量时一定不会出错，而普通变量在重排序就有可能被重排序到构造方法之外去，所以在另一个线程中读取普通变量时有时会读取不到正确的值。



读 final 域的重排序规则

在一个线程中，初次读对象引用和初次读对象所包含的 final 域，java 内存模型禁止处理器重排序这两个操作。

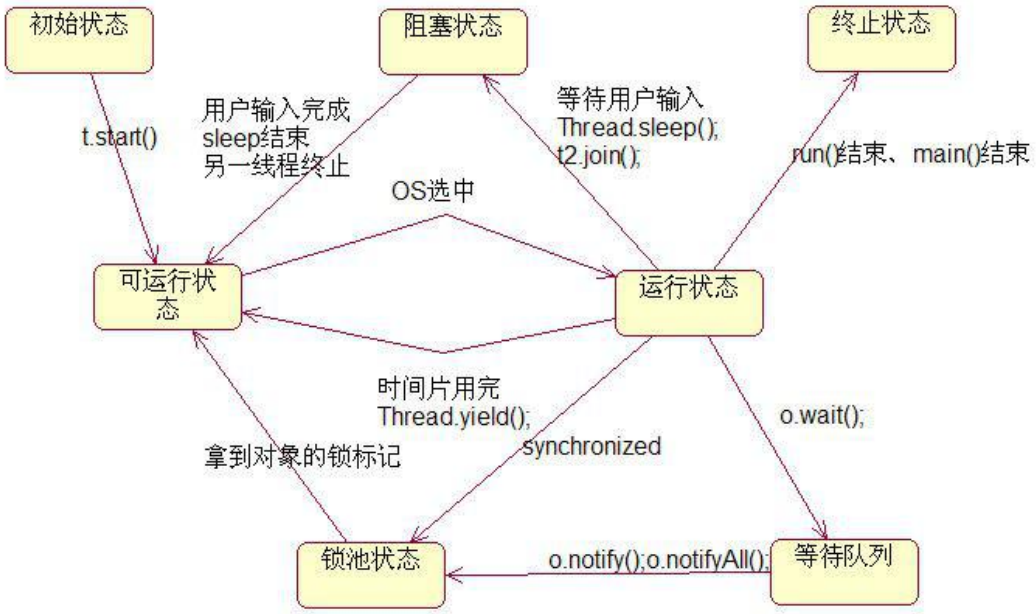
这样就保证了，在读取完对象后，final 域的初始化也能完成了。

Final 域为抽象类型

在构造方法内对一个 final 引用的对象的成员域的写入，与随后在构造方法外把这个构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

线程状态

多线程有几种状态



初始状态

阻塞状态

运行状态

可运行状态

终止状态

锁池状态

等待队列

线程状态以及 API 怎么操作会发生这种转换

1.初始状态————————————》可运行状态　　调用 start()方法

2.可运行状态————————————》运行状态　　获取 CPU 资源

3.运行状态————————————》终止状态　　run()方法或 main()方法结束

4.运行状态————————————》阻塞状态————————————》可运行状态

sleep()方法或其他线程的 join()　　sleep()结束或 join()结束后

5.运行状态————————————》可运行状态

线程调用了 yield()方法，意思是放弃当前获得的 CPU 时间片

6.可运行状态————————》锁池状态————————》可运行状态

当线程刚进入可运行状态(注意，还没运行)，发现将要调用的资源被 synchroniza(同步)，获取不到锁标记，锁池状态拿到对象锁到可运行状态

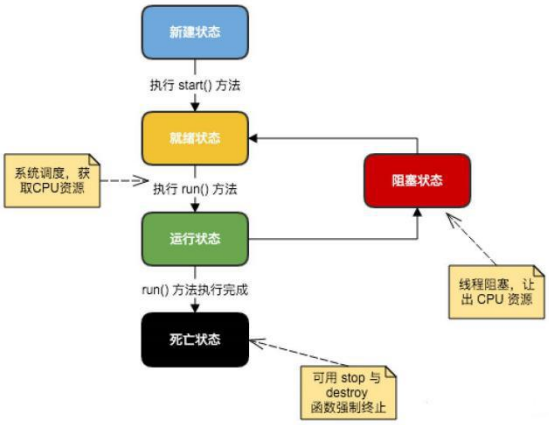
7.运行状态————————》等待队列————————》锁池状态

线程调用 wait()方法　　notifyAll()方法

8.运行状态————————》终止状态

run()结束、main 结束

线程的生命周期



此图与上一题的图一样，不过这个图将上图的阻塞状态、锁池状态、等待队列合并成一个阻塞状态

- 新建状态:

使用 new 关键字和 Thread 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 start() 这个线程。

- 就绪状态:

当线程对象调用了 start()方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待 JVM 里线程调度器的调度。

- 运行状态:

如果就绪状态的线程获取 CPU 资源，就可以执行 run()，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

- 阻塞状态:

如果一个线程执行了 sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。可以分为三种：

等待阻塞：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态。

同步阻塞：线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。

其他阻塞：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当 sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态

- 死亡状态:

一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

其他终止条件发生：

1.异常法：

在想停止的地方，抛出个 throw new InterruptedException();异常，出现异常锁自动释放。

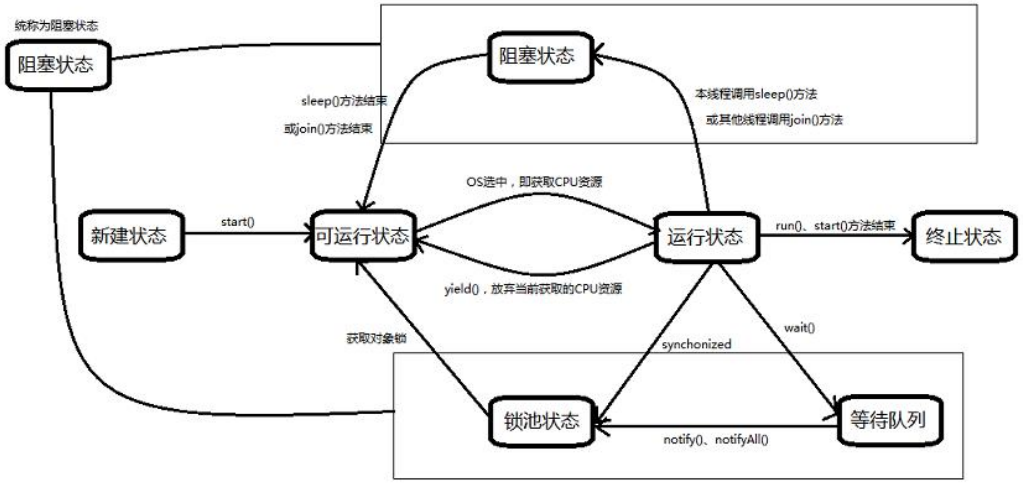
2.在沉睡中停止：

如果一个线程处于 sleep 状态时执行了 interrupt 方法后（即线程停止了）会出现异常停止

3.暴力停止：

使用 stop()方法直接就可以停止线程了，使用 stop()方法释放锁会给数据造成不一致性的结果，这样可能会出现程序处理的数据遭到破坏，最终导致程序执行的流程错误。stop()方法已经被标记为作废/过期的方法

4.使用 return 停止线程：



创建多线程

实现多线程的方式

1. 继承 Thread 类

   重写 run 方法

2. 实现 Runnable 接口

   必须完成 run 函数

使用方式：MyThreadImp 为实现的 Runable 类

```
MyThreadImp myThreadImp = new MyThreadImp();
Thread thread = new Thread(myThreadImp);
```

3. 匿名内部类方式

```
new Thread(){
            public void run() {
                System.out.println("hah");
            };
        }.start();
```

执行结果：hah

```
new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("kaka");
            }
        }).start();
```

执行结果：kaka

```
new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("runnable");
            }
        }){
            @Override
            public void run() {
                System.out.println("sub");
            }
        }.start();
```

执行结果：sub　　因为子类覆盖了父类的 run 方法

4.带返回值的线程

演示代码：

```
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

public class ThreadDemo2 implements Callable<Integer> {
    public static void main(String[] args) throws Exception {
        ThreadDemo2 demo = new ThreadDemo2();
        FutureTask<Integer> task = new FutureTask<Integer>(demo);
        Thread t = new Thread(task);
        t.start();
        System.out.println("我先干点别的");
        Integer result = task.get();
        System.out.println("线程打印的结果为：" + result);
    }
```

```
    @Override
    public Integer call() throws Exception {
        System.out.println("!!!");
        Thread.sleep(3000);
        return 1;
    }
}
```

5.定时器（quartz）

```
Timer timer = new Timer();
timer.schedule(new TimerTask() {
    @Override
    public void run() {
        System.out.println("start");
    }
}, 0, 1000);
```

6.线程池实现

```
//创建 10 个线程
ExecutorService threadPool = Executors.newFixedThreadPool(10);
for(int i = 0; i < 10; i++){
    threadPool.execute(new Runnable() {
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName());
        }
    });
}
threadPool.shutdown();
```

7.Lambda 表达式实现

```
new Thread(() -> System.out.println("Hello world !")).start();
```

8.spring 实现多线程

继承 Thread 类和实现 Runnable 接口的比较

其实在接触后我们会发现这完全是两个不同的实现多线程，继承 Thread 类是多个线程分别完成自己的任务，实现 Runnable 接口是多个线程共同完成一个任务。

实现 Runnable 接口比继承 Thread 类所具有的优势：

1）：适合多个相同的程序代码的线程去处理同一个资源

2）：可以避免 java 中的单继承的限制

3）：增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

怎么使用 Lambda 表达式创建线程

```
// 1.1 使用匿名内部类
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello world !");
    }
}).start();

// 1.2 使用  lambda expression
new Thread(() -> System.out.println("Hello world !")).start();

// 2.1 使用匿名内部类
```

```
Runnable race1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello world !");
    }
};

// 2.2 使用 lambda expression
Runnable race2 = () -> System.out.println("Hello world !");

// 直接调用 run 方法(没开新线程哦!)
race1.run();
race2.run();
```

线程中的方法原理及使用

start 方法和 run 方法区别

在 main 方法中执行的 run()方法不会创建新的线程，而在 main 方法中执行的 start()方法会启动一个新的线程，新的线程会调用 run 方法

run 方法可以多次执行，但是 start 方法不允许多次执行，多次执行 start()方法会抛出一个一个运行时异常 java.lang.IllegalThreadStateException

一个线程连着调用 start 两次会出现什么情况(这个讨论了好久好久，他说给你设计这个 start 你怎么处理这种情况，直接懵逼...提示结合那个线程状态机制想下)，即解决多次调用 start 方法的方法

```
public synchronized void start() {
        if (threadStatus != 0)
            throw new IllegalThreadStateException();
        group.add(this);
        boolean started = false;
```

```
        try {
            start0();
            started = true;
        } finally {
            try {
                if (!started) {
                    group.threadStartFailed(this);
                }
            } catch (Throwable ignore) {
            }
        }
    }
    private native void start0();
```

JDK 中的 native 方法的实现全部封装到 dll 中了，这也是 c/c++代码常用的手段。这样我们可以猜测一下 start0 方法中为我们做了什么东西，咱们可以看到 group.add(this);以及 group.threadStartFailed(this);这两段代码，没有对创建一个新的线程以及改变线程状态进行处理以及调用了 run 方法。所以猜测到 start0 方法中应该为我们实现了新建一个线程和将当前线程的状态改为就绪状态和调用 run 方法。

现在根据以上的分析继续分析一个线程调用两次 start 方法会出现什么状况，可以说在线程没有接触到 start0 方法时，执行的所有代码都是同步的。即使是同时两次调用的这个 start 方法，在 start 方法中也会有一段代码是同步执行的，第一个代码执行到 start0 方法后，第一个 start 方法新创建一个线程，由于看不到 start0 中的代码。所以我们猜测两种情况：

1、start0 方法中先创建的线程，然后再修改当前线程的状态

2、start0 方法中先修改的当前线程的状态，然后再创建个线程

根据这两种情况我们继续往下推：第一种情况，第一次调用 start 方法先创建线程，后改变状态，这样第二次调用也许会逃避掉第一行对其进行的状态判断，但是，在 group.add(this); 方法中的第一行还是一个状态判断（这里大家可以看一下 ThreadGroup 类的 add 方法），这个也许是真的逃不掉了，因为 threadStatus 这个标志位是 volatile 修饰的，即使新建出线程也会在第二次调用 start 方法时正确的确定当前线程的状态，会抛出

IllegalThreadStateException 异常。第二种情况，先改变状态，在进行创建新的线程。这样第二次调用直接在第一行那里就会被抛出异常。总之，无论 start0 方法怎么中创建线程和修改状态的顺序先后，在外侧的代码中进行过滤后都不会允许多次调用 start 方法的。

线程的五种生命周期， 新建->就绪->运行->死亡->堵塞。在每次实例化的时候，会给这个 threadStatus 为 0，用 volatile，可以保证该变量的可见性。在不同的生命周期，值会变，而在 start 方法通过判断，就可以知道执行的线程是不是一个新的线程实例。

wait sleep 区别

这两个方法来自不同的类分别是 Thread 和 Object

最主要是 sleep 方法没有释放锁，而 wait 方法释放了锁，使得其他线程可以使用同步控制块或者方法(锁代码块和方法锁)。

wait，notify 和 notifyAll 只能在同步控制方法或者同步控制块里面使用，而 sleep 可以在任何地方使用(使用范围)

sleep 必须捕获异常，而 wait，notify 和 notifyAll 不需要捕获异常

sleep 方法属于 Thread 类中方法，表示让一个线程进入睡眠状态，等待一定的时间之后，自动醒来进入到可运行状态，不会马上进入运行状态，因为线程调度机制恢复线程的运行也需要时间，一个线程对象调用了 sleep 方法之后，并不会释放他所持有的所有对象锁，所以也就不会影响其他进程对象的运行。但在 sleep 的过程中过程中有可能被其他对象调用它的 interrupt(),产生 InterruptedException 异常，如果你的程序不捕获这个异常，线程就会异常终止，进入 TERMINATED 状态，如果你的程序捕获了这个异常，那么程序就会继续执行 catch 语句块(可能还有 finally 语句块)以及以后的代码。

注意 sleep()方法是一个静态方法，也就是说他只对当前对象有效，通过 t.sleep()让 t 对象进入 sleep，这样的做法是错误的，它只会是使当前线程被 sleep 而不是 t 线程

wait 属于 Object 的成员方法，一旦一个对象调用了 wait 方法，必须要采用 notify()和 notifyAll()方法唤醒该进程;如果线程拥有某个或某些对象的同步锁，那么在调用了 wait()后，这个线程就会释放它持有的所有同步资源，而不限于这个被调用了 wait()方法的对象。wait()方法也同样会在 wait 的过程中有可能被其他对象调用 interrupt()方法而产生

wait 应用场景

生产者消费者：

资源类

```
public class P {
    private int count;
    public final int MAX_COUNT = 10;
    public synchronized void push (){
        while(count >= MAX_COUNT){
            try {
                System.out.println(Thread.currentThread().getName()  +  "
库存数量达到上限，停止生产");
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        count ++;
        System.out.println(Thread.currentThread().getName() + "  生产者生
产，当前库存为：" + count);
    }
    public synchronized void take(){
        while(count <= 0){
            try {
                System.out.println(Thread.currentThread().getName()  +  "
库存数量为0，消费者等待");
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }
        count --;
        System.out.println(Thread.currentThread().getName() + "  消费者消费，当前库存为：" + count);
        notifyAll();
    }
}
```

生产者：

```
public class PushTarget implements Runnable {
    private P p;
    public PushTarget(P p){
        this.p = p;
    }
    @Override
    public void run() {
        while(true){
            p.push();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

消费者：

```
public class TakeTarget implements Runnable {
```

```
    private P p;
    public TakeTarget(P p) {
        this.p = p;
    }
    @Override
    public void run() {
        while(true){
            p.take();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

测试类：

```
public class Main {
    public static void main(String[] args) {
        P p = new P();
        PushTarget push = new PushTarget(p);
        TakeTarget take = new TakeTarget(p);

        new Thread(push).start();
        new Thread(push).start();
        new Thread(push).start();
        new Thread(push).start();
        new Thread(push).start();
```

```
        new Thread(take).start();
        new Thread(take).start();
        new Thread(take).start();
        new Thread(take).start();
        new Thread(take).start();
    }
}
```

Thread 类的 interrupt,interrupted,isInterrupted 方法的区别

interrupt

方法用于中断线程。是用来设置中断状态的。返回 true 说明中断状态被设置了而不是被清除了。调用该方法的线程的状态将被置为"中断"状态。

interrupted

是静态方法，作用于当前线程，返回的是当前线程的中断状态。例如，如果当前线程被中断（没有抛出中断异常，否则中断状态就会被清除），你调用 interrupted 方法，第一次会返回 true。然后，当前线程的中断状态被方法内部清除了。第二次调用时就会返回 false。如果你刚开始一直调用 isInterrupted，则会一直返回 true，除非中间线程的中断状态被其他操作清除了。

isInterrupted

是作用于调用该方法的线程对象所对应的线程。interrupted 和 isInterrupted 都会调用同一个方法，只不过参数一个是 true，一个是 false。而且调用的方法的参数上也有所不同。

线程安全问题

什么是线程安全

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

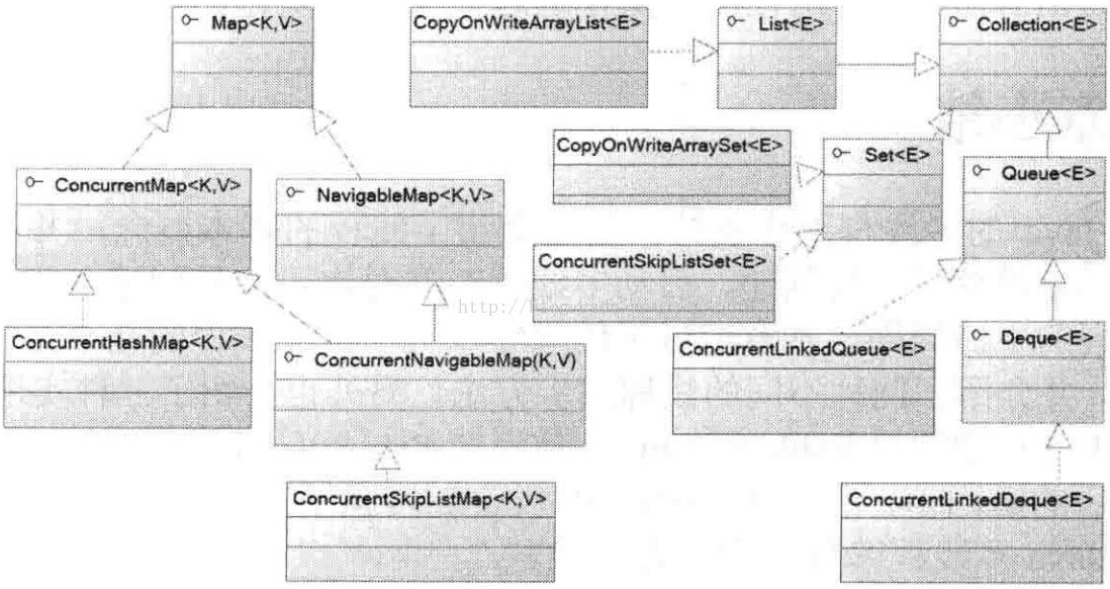
或者说:一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。

线程安全问题都是由全局变量及静态变量引起的。

若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。

存在竞争的线程不安全，不存在竞争的线程就是安全的

你知道有什么线程安全和线程不安全的类



Collection 文件包中的 atomic 文件夹中的类都是线程安全的

实现线程安全的方式

1.synchroinzied 锁（偏向锁，轻量级锁，重量级锁）

2.volatile 锁，只能保证数据可见性，不能保证原子性操作

3.JDK 提供的原子类　（至少要看懂一个源码）

4.使用 Lock 锁（共享锁，排它锁）

线程间的通信

多线程间协作与通信方式

https://blog.csdn.net/wangyy130/article/details/52039285

在并发编程中，经常会遇到多个线程之间需要相互协作的情况，即并不是多个线程同时执行，而是按照一定的顺序循环执行的情况。

那么怎样去实现这种效果呢？这里介绍三种方案。

这里都以子线程循环 10 次，然后主线程循环 10 次，然后往复循环 50 次的思路来做例子。

在下面的例子中 flag 代表一个共享变量。

一、synchronized+notify+wait+flag

```
public class communication01 {
    public static void main(String[] args){
        final besiness b=new besiness();
        new Thread(new Runnable() {
            @Override
            public void run() {
                for(int i=1;i<=50;i++){
                    b.sub(i);
                }
            }
        }).start();

        for (int i = 1; i <= 50; i++) {
            b.main(i);
        }
    }
}
    class besiness{
        private static  boolean flag=true;//flag 为 true 时允许 main 访问，为 false 时允许 suB 访问
        public synchronized void main(int i){
            while(!flag){
                try {
```

```
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        for (int j = 1; j <= 10; j++) {
            System.out.println("main thread==" + j + ",loop of " + i);
        }
        flag=false;
        this.notify();

    }
    public  synchronized  void sub(int i){
        while (flag){
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        for (int j = 1; j <= 10; j++) {
            System.out.println("sub thread==" + j + ",loop of " + i);
        }
        flag=true;
        this.notify();
    }
}
```

这种方案是通过对两个线程中分别要执行的方法加锁 synchronized，保证每次执行 main 时不被 sub 打断，执行 sub 循环时，不被 main 打断。

这里采用了对象 object 的 notify 和 wait 来实现线程之间的通信。当 main 方法执行完成

后，让执行 main 方法的线程等待，等待 sub 方法执行完成后，通知(notify)main 线程然后继续执行。这种方式有一个缺点，由于 notify 和 wait 使用的是 Object 的方法，所以不能单独的让某个特定的线程收到通知或者让他等待，而在存在多个线程同时等待时，只能通过 notifyAll 来通知所有的线程。不够灵活。

二、lock+condition+flag

```
public static void main(String[] args){
        final besiness b=new besiness();
        new Thread(new Runnable() {
            @Override
            public void run() {
                for(int i=1;i<=50;i++){
                    b.sub2(i);
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                for(int i=1;i<=50;i++){
                    b.sub3(i);
                }

            }
        }).start();
        for (int i = 1; i <= 50; i++) {
            b.main(i);
        }
    }
```

```
   static  class besiness{
        private   int flag=1;//flag 为 true 时允许 main 访问,为 false 时允许 suB
访问
        //condition1 来控制 main 和 sub2 之间的循环通信
       Condition condition1=lock.newCondition() ;
       //condition2 来控制 sub2 和 sub3 之间的循环通信
       Condition condition2=lock.newCondition() ;
       //condition1 来控制 main 和 sub3 之间的循环通信
       Condition condition3=lock.newCondition() ;
        public  void main(int i){
            lock.lock();
            try{
                while(flag!=1){
                    try {
                        condition1.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

                for (int j = 1; j <= 20; j++) {
                    System.out.println("main thread==" + j + ",loop of " +
i);

                }
                flag=2;
                condition2.signal();
            }finally {
                lock.unlock();
            }
```

```
        }
        public void sub2(int i){
            lock.lock();
            try{
                while ( flag !=2){
                    try {
                        condition2.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

                for (int j = 1; j <= 20; j++) {
                    System.out.println("sub2 thread==" + j + ",loop of " +
i);
                }
                flag=3;
                condition3.signal();
            }finally {
                lock.unlock();
            }
        }

       public void sub3(int i){
           lock.lock();

           try{
               while ( flag !=3){
                   try {
                       condition3.await();
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            for (int j = 1; j <= 20; j++) {
                System.out.println("sub3 thread==" + j + ",loop of " + i);
            }
            flag=1;
            condition1.signal();
        }finally {
            lock.unlock();
        }
    }
}
```

这种方式是利用了 Java5 中提供的 lock 和 condition，利用共享变量 flag 来实现线程之间的相互通信。同时在这个小例子中，相比上一个例子中增加了一个线程的循环。这是为了体现使用 condition 的优点。

使用 condition 可以非常灵活的去控制线程与线程之间的通信。因为在一个类中可以创建多个 condition 的实例，我们可以通过 condition 不同的实例的 signal 和 await 方法来标识不同的两个线程之间相互通信的标识，而不是统一使用 object 的 notify 和 wait 方法了。

同时利用 lock 方法可以利用锁的重入机制实现更加灵活的锁的应用。可以在需要的时候加锁或解锁。

这样我们就可以实现多个线程之间的协调通信了。

三、semaphere+flag

```
public static void main(String[] args){
    final besiness b=new besiness();

    new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                for(int i=1;i<=50;i++){
                    b.sub(i);
                }
            }
        }).start();

        for (int i = 1; i <= 50; i++) {
            b.main(i);
        }

    }
   static class besiness{
        private  int  flag=1;
       final Semaphore sp=new Semaphore(1);//声明一个信号，共享一个资源，每次只能允许一个线程执行
        public  void main(int i){
            try {
                sp.acquire();
                while(flag!=1){
                    sp.release();
                }
                for (int j = 1; j <= 10; j++) {
                    System.out.println("main thread==" + j + ",loop of " + i);
                }
                flag=2;
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }finally {
                sp.release();
            }
        }

        public   void sub(int i){
            try{
                try {
                    sp.acquire();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                while (flag !=2){

                    sp.release();
                }
                for (int j = 1; j <= 10; j++) {
                    System.out.println("sub thread==" + j + ",loop of " + i);
                }
                flag=1;
                sp.release();
            }finally {
                sp.release();
            }
        }
    }
```

这里 semaphere 代表一个信号量，它可以指示共享资源的个数，也就是同时访问资源的线程个数。这里主要通过 semaphere 的 acquire 和 release 实现锁的功能从而实现线程之间的通信。

利用 semaphere 不仅可以实现多个线程协调循环通信，在必要时还可以控制同一时间访

问资源的个数。更加的灵活和方便。

以上是实习多个线程之间相互协调通信的几种方案。

四、sleep/yield/join

五、CyclicBarrier 栅栏

```
package thread.blogs.cooperation;

import scala.Console;

import java.util.concurrent.CyclicBarrier;

/**
 * Created by PerkinsZhu on 2017/8/30 10:32.
 */
public class TestCyclicBarrier {
    public static void main(String[] args) {
        testCyclicBarrier();
    }
    private static void testCyclicBarrier() {
        /**
         * 注意这里等待的是三个线程。这就相当于一个线程计数器，当指定个数的线程执行 barrier.await();方法之后，才会执行后续的代码，否则每个线程都会一直进行等待。
         * 如果把 3 修改为 4，则将永远等待下去，不会开始会议。
         * 如果把 3 修改为 2，则小张到达之后就会提前开始会议，不会继续等待小王。
         */
        CyclicBarrier barrier = new CyclicBarrier(3);
```

```
        Thread 小李 = new Thread(new MyRunner(barrier, "小李", 2000));
        小李.start();
        Thread 小张 = new Thread(new MyRunner(barrier, "小张", 4000));
        小张.start();
        Thread 小王 = new Thread(new MyRunner(barrier, "小王", 5000));
        小王.start();
    }

    static class MyRunner implements Runnable {
        CyclicBarrier barrier;
        String name;
        int time;

        public MyRunner(CyclicBarrier barrier, String name, int time) {
            this.barrier = barrier;
            this.name = name;
            this.time = time;
        }

        @Override
        public void run() {
            Console.println(name + " 开始出发去公司。");
            sleep(time);
            Console.println(name + " 终于到会议室！！！");
            try {
                barrier.await();
            } catch (Exception e) {
                e.printStackTrace();
            }
            startMeeting(name);
```

```
        }
    }

    private static void startMeeting(String name) {
        Console.println(name + "说：人齐了。会议开始！！");
    }

    private static void sleep(int time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

六、CountDownLatch 闭锁

与 CycliBarrier 不同的是 CountDownLatch 是某一个线程等待其他线程执行到某一位置之后，该线程（调用 countDownLatch.await();等待的线程）才会继续后续工作。而 CycliBarrier 是各个线程执行到某位置之后，然后所有线程一齐开始后续的工作。相同的是两者都属于线程计数器。

```
package thread.blogs.cooperation;
import scala.Console;
import java.util.concurrent.CountDownLatch;
/**
 * Created by PerkinsZhu on 2017/8/30 10:32.
 */
public class TestCyclicBarrier {
    public static void main(String[] args) {
        testCyclicBarrier();
```

```
    }

    private static void testCyclicBarrier() {

        CountDownLatch countDownLatch = new CountDownLatch(3);//注意这里的参数指定了等待的线程数量
        new Thread(new MyRunner(countDownLatch, "小李", 2000)).start();
        new Thread(new MyRunner(countDownLatch, "小张", 4000)).start();
        new Thread(new MyRunner(countDownLatch, "小王", 5000)).start();

        try {
            Console.println("等待员工到来开会。。。。。。。");
            countDownLatch.await();//注意这里是 await。主线程将会一直等待在这里，当所有线程都执行 countDownLatch.countDown();之后当前线程才会继续执行
            startMeeting("Boss");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static class MyRunner implements Runnable {
        CountDownLatch countDownLatch;
        String name;
        int time;
        public MyRunner(CountDownLatch countDownLatch, String name, int time) {
            this.countDownLatch = countDownLatch;
            this.name = name;
            this.time = time;
        }
```

```
        @Override
        public void run() {
            Console.println(name + " 开始出发去公司。");
            sleep(time);
            Console.println(name + " 终于到会议室！！！");
            countDownLatch.countDown();
                Console.println(name + " 准备好了！！");
        }
    }

    private static void startMeeting(String name) {
        Console.println(name + "说：人齐了。会议开始！！");
    }

    private static void sleep(int time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

锁相关问题

线程的锁有啥

线程加锁有哪些方式？

synchronized 关键字

Java.util.concurrent 包中的 lock 接口和 ReentrantLock 实现类

说说锁，sync，lock（公平锁，非公平锁，实现） 读写锁，cas，aqs

synchronized 和 volatile

synchronized 是用及原理

https://blog.csdn.net/shandian000/article/details/54927876/

synchronized 放在方法上，内置锁就是当前类的实例

内置锁：即对象锁，java 中每一个对象都可以成为锁

互斥锁：一个线程获得锁，其他线程不能获得锁

修饰普通方法：

修饰普通方法只对当前对象加锁。

修饰静态方法：

修饰静态方式对任何一个此类的对象加锁。

修饰代码块：

对当前代码块进行加锁。

字节码底层原理：

查看 class 文件，javap -v class 文件名，查看 synchronized 在字节码中的实现

在 javap 的字节码文件中可以看到：

monitorenter

monitorexit

在这段字节码中就是 synchronized 包裹的代码。

任何对象都可以作为锁，那么锁信息又存在对象的什么地方呢？

存在对象头中：

对象头中的信息：

Mark Word

Class Metadata Address

Array Length（数组有的信息）

偏向锁：

偏向锁时，Mark Word 包括（线程 id，Epoch，对象的分代信息，是否是偏向锁，锁标志位）。每次获取锁和释放锁会浪费资源，很多情况下，竞争锁不是由多个线程，而是由一个线程在使用。偏向锁没有竞争的情况不释放锁，一旦出现竞争便可以释放锁。

轻量级锁：

当一个线程访问同步代码块是，会将 Mark Word 复制一份进行修改，如果还有线程来访问，也会将 Mark Word 复制一份进行修改，如果锁标志位标记着有线程占用同步代码块，会出现修改失败的操作，然后会继续循环修改锁的操作，直到线程释放锁。这个循环修改锁是自旋锁。轻量级锁的优点多个线程可以同时。

重量级锁：

synchronized 的膨胀机制。

volatile 关键字？

volatile 称之为轻量级锁，被 vilatile 修饰的变量，在线程之间是可见的。就是一个线程修改了这个变量，在另一个线程中能够读到这个修改之后的值。volatile 能实现变量可见性，但是不能保证原子性操作。

Synchronized 除了线程之间互斥之外，还有一个非常大的作用，就是保证变量线程间可见性。

synchronized 能完全替代 volatile，而 volatile 不能完全替代 synchronized。因为 volatile 不能保证原子性。而 volatile 实现变量可见性更轻量化。

volatile 可见性怎么实现的？

volatile 关键字解决的是内存可见性的问题，会使得所有对 volatile 变量的读写都会直接刷到主存，即保证了变量的可见性。这样就能满足一些对变量可见性有要求而对读取顺序没有要求的需求。

volatile 说下作用，举个例子

```
public class Demo {
    private volatile int a = 1;
    public int getA() {
        return a;
    }
    public void setA(int a) {
        this.a = a;
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        new Thread(new Runnable() {
            @Override
            public void run() {
                d.setA(12);
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(d.getA());
            }
        }).start();
```

```
    }
}
```

synchronized 和 volatile 区别

1）volatile 本质是在告诉 jvm 当前变量在寄存器中的值是不确定的,需要从主存中读取,synchronized 则是锁定当前变量,只有当前线程可以访问该变量,其他线程被阻塞住.

2）volatile 仅能使用在变量级别,synchronized 则可以使用在变量,方法.

3）volatile 仅能实现变量的修改可见性,而 synchronized 则可以保证变量的修改可见性和原子性.

4）volatile 不会造成线程的阻塞,而 synchronized 可能会造成线程的阻塞.

5、当一个域的值依赖于它之前的值时，volatile 就无法工作了，如 n=n+1,n++等。如果某个域的值受到其他域的值的限制，那么 volatile 也无法工作，如 Range 类的 lower 和 upper 边界，必须遵循 lower<=upper 的限制。

6、使用 volatile 而不是 synchronized 的唯一安全的情况是类中只有一个可变的域。

Lock

lock 类实现

void lock();	获取锁
void lockInterruptibly()	如果当前线程未被中断，则获取锁。
boolean tryLock()	仅在调用时锁为空闲状态才获取该锁。
boolean tryLock(long time, TimeUnitunit)	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。
void unlock()	释放锁。
Condition newCondition()	返回绑定到此 Lock 实例的新 Condition 实例。

Lock 接口的实现类 ReentrantLock，初始化时，默认情况下 ReentrantLock 实例的是非公平锁，而初始化时传值为 true 时，ReentrantLock 实例的是公平锁。在 ReentrantLock 类中有三个内部类 Syn、NonfairSync、FairSync。Syn 类继承了 AQS，NonfairSync 和 FairSync 继承了 Syn 分别实现了非公平锁类和公平锁类。

Lock 锁的使用：

```
public class Sequeuece {
    private int value;
    Lock lock = new ReentrantLock();

    public int getNext(){
        lock.lock();
        int a =  value++;
        lock.unlock();
        return a;
    }

    public static void main(String[] args) {
        Sequeuece s = new Sequeuece();

        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    System.out.println(Thread.currentThread().getName()  +
" " + s.getNext());
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    System.out.println(Thread.currentThread().getName()  +
" " + s.getNext());
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
    }
}
```

使用 Lock 锁的优点：

1.灵活性：

Lock 需要显示地获取和释放锁，繁琐，但是能让代码更灵活，可以在任意的地方去释放锁和获取锁。

synchronized 不需要显示地获取和释放锁，简单，但不灵活。

2.使用 Lock 可以方便的实现公平性。

3.非阻塞的获取锁

4.能被中断的获取锁

5.超时获取锁

lock 和 synchronized 的区别，我就直接从对象头那开始讲，到 AQS 的基于 state 和 cas。

类别	synchronized	lock
存在层次	Java 的关键字，在 jvm 层面上	是一个类
锁的释放	1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm 会让线程释放锁	在 finally 中必须释放锁，不然容易造成线程死锁
锁的获取	假设 A 线程获得锁，B 线程等待。如果 A 线程阻塞，B 线程会一直等待	分情况而定，Lock 有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待
锁状态	无法判断	可以判断
锁类型	可重入 不可中断 非公平	可重入 可判断 可公平（两者皆可）
性能	少量同步	大量同步

）synchronized 在成功完成功能或者抛出异常时，虚拟机会自动释放线程占有的锁；而 Lock 对象在发生异常时，如果没有主动调用 unLock()方法去释放锁，则锁对象会一直持有，因此使用 Lock 时需要在 finally 块中释放锁；

2）lock 接口锁可以通过多种方法来尝试获取锁包括立即返回是否成功的 tryLock(),以及一直尝试获取的 lock()方法和尝试等待指定时间长度获取的方法，相对灵活了许多比 synchronized;

3) 通过在读多，写少的高并发情况下，我们用 ReentrantReadWriteLock 分别获取读锁和写锁来提高系统的性能，因为读锁是共享锁，即可以同时有多个线程读取共享资源，而写锁则保证了对共享资源的修改只能是单线程的。

ReentrantLock 和 Synchronized 区别

除了 synchronized 的功能，多了三个高级功能。

等待可中断，公平锁，绑定多个 Condition。

1. 等待可中断：在持有锁的线程长时间不释放锁的时候，等待的线程可以选择放弃等待，tryLock(long timeout, TimeUnit unit)

2. 公平锁：按照申请锁的顺序来一次获得锁称为公平锁，synchronized 的是非公平锁，ReentrantLock 可以通过构造函数实现公平锁。new RenentrantLock(boolean fair)

3. 绑定多个 Condition：通过多次 newCondition 可以获得多个 Condition 对象，可以简单的实现比较负责的线程同步的功能，通过 await(),signal();

ReentrantLock 优缺点

reentrantlock 的优点

可以添加多个检控条件，如果使用 synchronized,则只能使用一个．使用 reentrant locks 可以有多个 wait()/notify() 队列. [译注:直接多 new 几个 ReentrantLock 就可以了,不同的场景/条件用不同的 ReentrantLock]

可以控制线程得到锁的顺序,也就是有公平锁(按照进入顺序得到资源),也可以不按照顺就像.synchronized 一样.

可以查看锁的状态，锁是否被锁上了.

可以查看当前有多少线程再等待锁.

reentrantlock 的缺点

需要使用 import 引入相关的 Class

不能忘记在 finally 模块释放锁,这个看起来比 synchronized 丑陋

synchronized 可以放在方法的定义里面，而 reentrantlock 只能放在块里面．比较起来, synchronized 可以减少嵌套

reentrantlock 的 reentrant 是什么意思

可重入

ThreadLocal

1.2 就出现了，1.5 时进行了改写。

在 java.lang 文件包中

ThreadLocal 用于保存某个线程共享变量：对于同一个 static ThreadLocal，不同线程只能从中

get，set，remove 自己的变量，而不会影响其他线程的变量。

1、ThreadLocal.get: 获取 ThreadLocal 中当前线程共享变量的值。

2、ThreadLocal.set: 设置 ThreadLocal 中当前线程共享变量的值。

3、ThreadLocal.remove: 移除 ThreadLocal 中当前线程共享变量的值。

4、ThreadLocal.initialValue: ThreadLocal 没有被当前线程赋值时或当前线程刚调用 remove 方法后调用 get 方法，返回此方法值。

ThreadLock 底层原理

ThreadLock 底层是一个 map 的数据结构，以当前线程为 key，以值为 value

使用：

```
public class Demo {
    private ThreadLocal<Integer> count = new ThreadLocal<Integer>(){
        protected Integer initialValue() {
            return new Integer(0);
        };
    };
    public int getNext(){
        Integer value = count.get();
        value++;
        count.set(value);
        return value;
    }
    public static void main(String[] args) {
        Demo d = new Demo();
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    System.out.println(Thread.currentThread().getName()  +
```

```
" " + d.getNext());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    System.out.println(Thread.currentThread().getName()  +
" " + d.getNext());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true){
                    System.out.println(Thread.currentThread().getName()  +
" " + d.getNext());
                    try {
```

```
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();
    new Thread(new Runnable() {
        @Override
        public void run() {
            while(true){
                System.out.println(Thread.currentThread().getName()  +
" " + d.getNext());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();
  }
}
```

CAS&AQS

CAS 是一种什么样的同步机制？

CAS 是英文单词 Compare and Swap 的缩写，翻译过来就是比较并替换。

CAS 机制中使用了 3 个基本操作数：内存地址 V，旧的预期值 A，要修改的新值 B。

更新一个变量的时候，只有当变量的预期值 A 和内存地址 V 当中的实际值相同时，才

会将内存地址 V 对应的值修改为 B。

从思想上来说，synchronized 属于悲观锁，悲观的认为程序中的并发情况严重，所以严防死守，CAS 属于乐观锁，乐观地认为程序中的并发情况不那么严重，所以让线程不断去重试更新。

CAS 的缺点：

1） CPU 开销过大

在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给 CPU 带来很到的压力。

2） 不能保证代码块的原子性

CAS 机制所保证的知识一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证 3 个变量共同进行原子性的更新，就不得不使用 synchronized 了。

3） ABA 问题

这是 CAS 机制最大的问题所在。（后面有介绍）

CAS 可以助记的一个例子：

1. 在内存地址 V 当中，存储着值为 10 的变量。

内存地址V

2. 此时线程 1 想把变量的值增加 1.对线程 1 来说，旧的预期值 A=10，要修改的新值 B=11.

内存地址V

线程1： A = 10 B = 11

3.在线程 1 要提交更新之前，另一个线程 2 抢先一步，把内存地址 V 中的变量值率先更新成了 11。

内存地址V

线程1： A = 10　　B = 11

线程2： 把变量值更新为11

4. 线程 1 开始提交更新，首先进行 A 和地址 V 的实际值比较，发现 A 不等于 V 的实际值，提交失败。

内存地址V

线程1： A = 10　　B = 11

A != V的值 (10!=11)

提交失败！

线程2： 把变量值更新为11

5. 线程 1 重新获取内存地址 V 的当前值，并重新计算想要修改的值。此时对线程 1 来说，A=11，B=12。这个重新尝试的过程被称为自旋。

内存地址V

线程1： A = 11　　B = 12

6. 这一次比较幸运，没有其他线程改变地址 V 的值。线程 1 进行比较，发现 A 和地址 V 的实际值是相等的。

线程1： A = 11　　B = 12
A == V的值 (11 == 11)

7. 线程 1 进行交换，把地址 V 的值替换为 B，也就是 12.

线程1： A = 11　　B = 12
A == V的值 (11 == 11)
地址V的值更新为12

CAS 机制会出现什么问题

CAS 的 ABA 问题和解决办法：

什么是 ABA 问题：见图：

由于提款机硬件出了点问题，小灰的提款操作被同时提交了两次，开启了两个线程，两个线程都是获取当前值 100 元，要更新成 50 元。

理想情况下，应该一个线程更新成功，一个线程更新失败，小灰的存款值被扣一次。

存款余额：100元

线程 1 首先执行成功，把余额从 100 改成 50.线程 2 因为某种原因阻塞。这时，小灰的妈妈刚好给小灰汇款 50 元。

线程 2 仍然是阻塞状态，线程 3 执行成功，把余额从 50 改成了 100。

线程 2 恢复运行，由于阻塞之前获得了“当前值”100，并且经过 compare 检测，此时存款实际值也是 100，所以会成功把变量值 100 更新成 50。

原本线程 2 应当提交失败，小灰的正确余额应该保持 100 元，结果由于 ABA 问题提交成功

了。

怎么接解决：

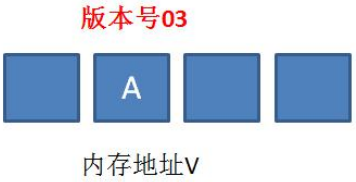
怎么解决呢？加个版本号就可以了。

真正要做到严谨的 CAS 机制，我们在 compare 阶段不仅要比较期望值 A 和地址 V 中的实际值，还要比较变量的版本号是否一致。

我们仍然以刚才的例子来说明，假设地址 V 中存储着变量值 A，当前版本号是 01。线程 1 获取了当前值 A 和版本号 01，想要更新为 B，但是被阻塞了。

线程1： 获取当前值A，版本号01，期望更新为B

这时候，内存地址 V 中变量发生了多次改变，版本号提升为 03，但是变量值仍然是 A。



线程1： 获取当前值A，版本号01，期望更新为B

随后线程 1 恢复运行，进行 compare 操作。经过比较，线程 1 所获得的值和地址的实际值都是 A，但是版本号不相等，所以这一次更新失败。

线程1： 获取当前值A，版本号01，期望更新为B
A == A
01 != 03
更新失败！

在 Java 中，AtomicStampedReference 类就实现了用版本号作比较额 CAS 机制。

1. java 语言 CAS 底层如何实现？

利用 unsafe 提供的原子性操作方法。

2.什么事 ABA 问题？怎么解决？

当一个值从 A 变成 B，又更新回 A，普通 CAS 机制会误判通过检测。

利用版本号比较可以有效解决 ABA 问题。

锁优化 CAS

JDK1.6 中高校并发是一个重要改进。里面的给出的各种锁都是为了线程间更高效的共享数据。优化的方法有下面几种。这里的锁优化主要是针对 synchronized 关键字来说，它产生的是一种重量级的锁定（重量级的锁定不是用 CAS），会有互斥，效率较低。而在 JDK1.6 后，引入的自旋锁，轻量级锁，偏向锁对互斥同步进行了优化，它们三种锁默认都是开启的。

1.锁消除，锁粗化。锁消除是判断当堆上的数据不会被其他线程访问到时，该线程上的同步加锁就无需进行。

由于加锁和解锁的开销很大，如果不断的加锁和解锁操作都是对于同一个对象，虚拟机会把整个加锁同步的范围扩张到操作序列的外部，就是只加一次锁。

2.自旋锁：互斥同步对性能最大的影响是阻塞的实现，挂起线程和恢复线程都需要转入内核中完成。现将本该要阻塞的线程不去挂起，不放弃处理器的执行时间，而是在那做一个忙循环（自旋），看看持有锁的线程是否很快释放锁。自旋的次数（循环的次数）是有限度的，默认是 10 次，如果没有获得锁就采用传统的方式去挂起线程。

3.轻量级锁：在线程没有竞争的时候，采用 CAS 操作，避免使用互斥量的开销。这里涉及到对象头的概念。

4.偏向锁：它相对于轻量级锁，减少了锁重入的开销，对于第一个获得锁的线程，后面的执行如果该锁没有被其他线程获取，则该线程将不再进行同步（CAS 操作）。

轻量级锁和偏向锁都是在没有竞争的情况下出现，一旦出现竞争就会升级为重量级锁。对于 synchronized，锁的升级情况可能是 偏向锁—>轻量锁—>自适应自旋锁—>重量锁

了解 AQS 吗？

AQS 是模板方法模式

为实现依赖于先进先出 (FIFO) 等待队列的阻塞锁和相关同步器（信号量、事件，等等）提供一个框架。

AQS 底层源码自己理解分析：

AQS 会维护一个双向列表。当一个线程申请资源时，如果当前线程别占用，就会往维护的链表上添加一个 Node 结点。

使用此类：

为了将此类用作同步器的基础，需要适当地重新定义以下方法，这是通过使用 getState()、setState(int) 和/或 compareAndSetState(int, int) 方法来检查和/或修改同步状态来实现的：

tryAcquire(int)　　试图在独占模式下获取对象状态。

tryRelease(int)　　试图设置状态来反映独占模式下的一个释放。

tryAcquireShared(int)　　试图在共享模式下获取对象状态。

tryReleaseShared(int)　　试图设置状态来反映共享模式下的一个释放。

isHeldExclusively()　　如果对于当前（正调用的）线程，同步是以独占方式进行的，则返回 true。

默认情况下，每个方法都抛出 UnsupportedOperationException。这些方法的实现在内部必须是线程安全的，通常应该很短并且不被阻塞。定义这些方法是使用此类的唯一 受支持的方式。其他所有方法都被声明为 final，因为它们无法是各不相同的。

实现 AQS 的数据结构：

底层是一个链表

```
static final class Node {
        static final Node SHARED = new Node();
        static final Node EXCLUSIVE = null;
        static final int CANCELLED =  1;    //线程已取消状态
        static final int SIGNAL    = -1;
        static final int CONDITION = -2;
        static final int PROPAGATE = -3;

        volatile int waitStatus;        //等待状态
        volatile Node prev;
        volatile Node next;
        volatile Thread thread;
```

```
        Node nextWaiter;
        final boolean isShared() {
            return nextWaiter == SHARED;
        }
        final Node predecessor() throws NullPointerException {
            Node p = prev;
            if (p == null)
                throw new NullPointerException();
            else
                return p;
        }
        Node() {    // Used to establish initial head or SHARED marker
        }
        Node(Thread thread, Node mode) {     // Used by addWaiter
            this.nextWaiter = mode;
            this.thread = thread;
        }
        Node(Thread thread, int waitStatus) { // Used by Condition
            this.waitStatus = waitStatus;
            this.thread = thread;
        }
    }
```

使用 AQS 实现一个自己的 Lock 类：

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.AbstractQueuedSynchronizer;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
public class MyLock2 implements Lock {
```

```
    private Helper helper = new Helper();
    private class Helper extends AbstractQueuedSynchronizer{
        @Override
        protected boolean tryAcquire(int arg) {
            /**
             * 如果第一个线程进来，可以拿到锁，因此我们可以返回 true
             * 如果第二个线程进来，则拿不到锁，返回 false 有种特例，如果当前进来的线程和当前线程时同一个线程，则可以拿到锁，但是有代价的，更新状态值
             * 如何判断是第一个线程进来还是其他线程进来？
             */
            int state = getState();
            Thread t = Thread.currentThread();
            if(state == 0){
                if(compareAndSetState(0,arg)){
                    setExclusiveOwnerThread(Thread.currentThread());
                    return true;
                }
            } else if(getExclusiveOwnerThread() == t) {
                setState(state + 1);
                return true;
            }
            return false;
        }
        @Override
        protected boolean tryRelease(int arg) {
            //锁的获取和释放肯定是一一对应的,那么调用此方法的线程一定是当前线程
            if(Thread.currentThread() != getExclusiveOwnerThread()){
                throw new RuntimeException();
            }
```

```
            int state = getState() - arg;
            boolean flag = false;
            if(state == 0){
                setExclusiveOwnerThread(null);
                flag = true;
            }
            setState(state);
            return flag;
        }
        Condition newCondition(){
            return new ConditionObject();
        }
    }
    @Override
    public void lock() {
        helper.acquire(1);
    }
    @Override
    public void lockInterruptibly() throws InterruptedException {
        helper.acquireInterruptibly(1);
    }
    @Override
    public boolean tryLock() {
        return helper.tryAcquire(1);
    }
    @Override
    public boolean tryLock(long time, TimeUnit unit)
            throws InterruptedException {
        return helper.tryAcquireNanos(1, unit.toNanos(time));
    }
```

```
    @Override
    public void unlock() {
        helper.release(1);
    }
    @Override
    public Condition newCondition() {
        return helper.newCondition();
    }
}
```

并发容器和工具类

介绍一下 CopyOnWriteArrayList 的应用场景以及实现原理

实现原理：在 ArrayList 基础上进行了修改，他对会出现线程安全的方法上做了改变

在修改的时候会复制出一份，然后再进行修改，再将这个新数组设置在原数组上。

使用场景：读多写少的操作，读少写多的操作用同步操作也可以

CountDownLatch 和 CyclicBarrier 的区别？

cyclicBarrier 和 CountDownLatch 的区别

1、CountDownLatch 简单的说就是一个线程等待，直到他所等待的其他线程都执行完成并且调用 countDown()方法发出通知后，当前线程才可以继续执行。

2、cyclicBarrier 是所有线程都进行等待，直到所有线程都准备好进入 await()方法之后，所有线程同时开始执行！

3、CountDownLatch 的计数器只能使用一次。而 CyclicBarrier 的计数器可以使用 reset() 方法重置。所以 CyclicBarrier 能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

4,、CyclicBarrier 还提供其他有用的方法，比如 getNumberWaiting 方法可以获得 CyclicBarrier 阻塞的线程数量。isBroken 方法用来知道阻塞的线程是否被中断。如果被中断返回 true，否

则返回 false。

CountDownLatch

CountDownLatch 是一个同步的辅助类，允许一个或多个线程，等待其他一组线程完成操作，再继续执行。

分析 CountDownLatch：

运行程序我们会发现当我们在 t1 调用 CountDownLatch 的 await()方法时，就好比我们调用了 wait()方法，当前线程会处于阻塞状态，直到等到 t2 和 t3 完全执行完毕并且调用 countDown()方法时，我们才能唤醒 t1 继续进行执行，CountDownLatch 就好比一个计时器，我们可以让当前线程调用 CountDownLatch 中的 await()方法进行等待，如果想让当前线程继续执行，我们必须让 CountDownLatch 获得初始化时候传入的构造参数个 countDown()方法，我们才能继续执行。

CountDownLatch 的使用场景：

在一些应用场合中，需要等待某个条件达到要求后才能做后面的事情；同时当线程都完成后也会触发事件，以便进行后面的操作。 这个时候就可以使用 CountDownLatch。CountDownLatch 最重要的方法是 countDown()和 await()，前者主要是倒数一次，后者是等待倒数到 0，如果没有到达 0，就只有阻塞等待了。

一个 CountDownLatch 的例子：对一个文本中所有数据并行求和

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
public class Demo2 {
    private int[] nums;
    public Demo2(int line){
        nums = new int[line];
    }
```

```
    public void calc(String line, int index, CountDownLatch latch){
        String[] nus = line.split(",");        //切分出分一个数字
        int total = 0;
        for(String num : nus){
            total += Integer.parseInt(num);
        }
        nums[index] = total;
        System.out.println(Thread.currentThread().getName() + " 执行计算任
务..." + line + " 结果为：" + total);
        latch.countDown();
    }
    public void sum(){
        int total = 0;
        for(int i = 0; i < nums.length; i++){
            total += nums[i];
        }
        System.out.println("最终的结果为：" + total);
    }
    public static void main(String[] args) {
        List<String> contents = readFile();
        int lineCount = contents.size();

        CountDownLatch latch = new CountDownLatch(lineCount);

        Demo2 d = new Demo2(lineCount);
        for(int i = 0; i < lineCount; i++){
            final int j = i;
            new Thread(new Runnable() {
                @Override
                public void run() {
```

```
                d.calc(contents.get(j), j, latch);
            }
        }).start();
    }

    try {
        latch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    d.sum();
}
private static List<String> readFile() {
    List<String> contents = new ArrayList<String>();
    String line = null;
    BufferedReader br = null;
    try {
        br = new BufferedReader(new FileReader("D:\\nums.txt"));
        while((line = br.readLine()) != null){
            contents.add(line);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if(br != null){
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
            }
        }
        return contents;
    }
}
```

CyclicBarrier

CyclicBarrier 是一个同步的辅助类，允许一组线程相互之间等待，达到一个共同点，再继续执行。

CyclicBarrier 分析结果：

上述程序我们创建了一个线程池，这个线程池中有三个线程，每个线程都传递了一个相同的 CyclicBarrier 对象和运动员的名字，我们 Runner 类中的 run 方法使每一个进来的运动员都休眠 0-5 秒的时间，然后调用 await()方法，就是说每个线程进来都需要进行等待，直到所有的 CyclicBarrier 都处于准备好了的状态，所有线程才能统一开始执行！

CyclicBarrier 使用场景

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个 Excel 保存了用户所有银行流水，每个 Sheet 保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个 sheet 里的银行流水，都执行完之后，得到每个 sheet 的日均银行流水，最后，再用 barrierAction 用这些线程的计算结果，计算出整个 Excel 的日均银行流水。

cyclicBarrier

```
import java.util.Random;
import java.util.concurrent.CyclicBarrier;

public class Demo3 {
    Random random = new Random();
```

```
    public void meeting(CyclicBarrier barrier){
        try {
            Thread.sleep(random.nextInt(4000));
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " 到达会议
室，等待开会");

        try {
            barrier.await();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        Demo3 demo = new Demo3();
        CyclicBarrier barrier = new CyclicBarrier(10, new Runnable() {
            @Override
            public void run() {
                System.out.println("开始开会。。。");
            }
        });

        for(int i = 0; i < 10; i++){
            new Thread(new Runnable() {
                @Override
                public void run() {
```

```
                    demo.meeting(barrier);
                }
            }).start();
        }
    }
}
```

Future

1. Future 的应用场景

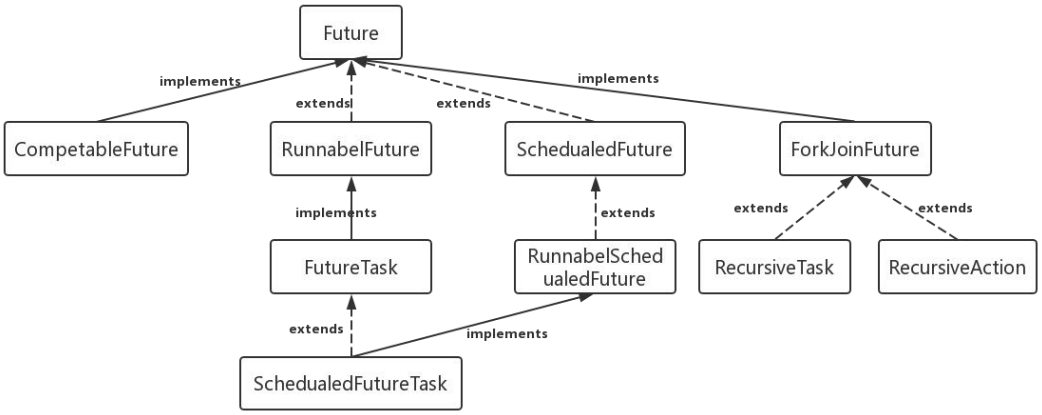
在并发编程中，我们经常用到非阻塞的模型，在之前的多线程的三种实现中，不管是继承 thread 类还是实现 runnable 接口，都无法保证获取到之前的执行结果。通过实现 Callback 接口，并用 Future 可以来接收多线程的执行结果。

Future 表示一个可能还没有完成的异步任务的结果，针对这个结果可以添加 Callback 以便在任务执行成功或失败后作出相应的操作。

举个例子：比如去吃早点时，点了包子和凉菜，包子需要等 3 分钟，凉菜只需 1 分钟，如果是串行的一个执行，在吃上早点的时候需要等待 4 分钟，但是因为你在等包子的时候，可以同时准备凉菜，所以在准备凉菜的过程中，可以同时准备包子，这样只需要等待 3 分钟。那 Future 这种模式就是后面这种执行模式。

2. Future 的类图结构

Future 接口定义了主要的 5 个接口方法，有 RunnableFuture 和 SchedualFuture 继承这个接口，以及 CompleteFuture 和 ForkJoinTask 继承这个接口。



RunnableFuture

这个接口同时继承 Future 接口和 Runnable 接口，在成功执行 run（）方法后，可以通过 Future 访问执行结果。这个接口都实现类是 FutureTask,一个可取消的异步计算，这个类提供了 Future 的基本实现，后面我们的 demo 也是用这个类实现，它实现了启动和取消一个计算，查询这个计算是否已完成，恢复计算结果。计算的结果只能在计算已经完成的情况下恢复。如果计算没有完成，get 方法会阻塞，一旦计算完成，这个计算将不能被重启和取消，除非调用 runAndReset 方法。

FutureTask 能用来包装一个 Callable 或 Runnable 对象，因为它实现了 Runnable 接口，而且它能被传递到 Executor 进行执行。为了提供单例类，这个类在创建自定义的工作类时提供了 protected 构造函数。

SchedualFuture

这个接口表示一个延时的行为可以被取消。通常一个安排好的 future 是定时任务 SchedualedExecutorService 的结果

CompleteFuture

一个 Future 类是显示的完成，而且能被用作一个完成等级，通过它的完成触发支持的依赖函数和行为。当两个或多个线程要执行完成或取消操作时，只有一个能够成功。

ForkJoinTask

基于任务的抽象类，可以通过 ForkJoinPool 来执行。一个 ForkJoinTask 是类似于线程实体，但是相对于线程实体是轻量级的。大量的任务和子任务会被 ForkJoinPool 池中的真实线程挂起来，以某些使用限制为代价。

3. Future 的主要方法

Future 接口主要包括 5 个方法

```
public interface Future<V> {

     * Attempts to cancel execution of this task.  This attempt will
    boolean cancel(boolean mayInterruptIfRunning);

     * Returns {@code true} if this task was cancelled before it completed
    boolean isCancelled();

     * Returns {@code true} if this task completed.
    boolean isDone();

     * Waits if necessary for the computation to complete, and then
    V get() throws InterruptedException, ExecutionException;

     * Waits if necessary for at most the given time for the computation
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

get（）方法可以当任务结束后返回一个结果，如果调用时，工作还没有结束，则会阻塞线程，直到任务执行完毕

get（long timeout,TimeUnit unit）做多等待 timeout 的时间就会返回结果

cancel（boolean mayInterruptIfRunning）方法可以用来停止一个任务，如果任务可以停止（通过 mayInterruptIfRunning 来进行判断），则可以返回 true,如果任务已经完成或者已经停止，或者这个任务无法停止，则会返回 false.

isDone（）方法判断当前方法是否完成

isCancel（）方法判断当前方法是否取消

4. Future 示例 demo

需求场景：等早餐过程中，包子需要 3 秒，凉菜需要 1 秒，普通的多线程需要四秒才能完成。先等凉菜，再等包子，因为等凉菜时，普通多线程启动 start()方法，执行 run()中具体方法时，没有返回结果，所以如果要等有返回结果，必须是要 1 秒结束后才知道结果。

普通多线程：

```
public class BumThread extends Thread{
    @Override
    public void run() {
        try {
            Thread.sleep(1000*3);
            System.out.println("包子准备完毕");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
    }
}
public class ColdDishThread extends Thread{
    @Override
    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println("凉菜准备完毕");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public static void main(String[] args) throws InterruptedException {
    long start = System.currentTimeMillis();
    // 等凉菜 -- 必须要等待返回的结果，所以要调用 join 方法
    Thread t1 = new ColdDishThread();
    t1.start();
    t1.join();
    // 等包子 -- 必须要等待返回的结果，所以要调用 join 方法
    Thread t2 = new BumThread();
    t2.start();
    t2.join();
    long end = System.currentTimeMillis();
    System.out.println("准备完毕时间："+(end-start));
}
```

采用 Future 模式：

```
public static void main(String[] args) throws InterruptedException,
ExecutionException {
    long start = System.currentTimeMillis();
```

```
// 等凉菜
Callable ca1 = new Callable(){
    @Override
    public String call() throws Exception {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "凉菜准备完毕";
    }
};
FutureTask<String> ft1 = new FutureTask<String>(ca1);
new Thread(ft1).start();
// 等包子 -- 必须要等待返回的结果，所以要调用 join 方法
Callable ca2 = new Callable(){
        @Override
        public Object call() throws Exception {
            try {
                Thread.sleep(1000*3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "包子准备完毕";
    }
};
FutureTask<String> ft2 = new FutureTask<String>(ca2);
new Thread(ft2).start();
System.out.println(ft1.get());
System.out.println(ft2.get());
```

```
    long end = System.currentTimeMillis();
    System.out.println("准备完毕时间："+(end-start));
}
```

自己写的 Future 设计模式的一个例子

需要实例化的类：

```
public class Product {
    private int id;
    private String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Product(int id, String name) {
        System.out.println("开始生产 " + name);
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.id = id;
```

```
        this.name = name;
        System.out.println(name + "生产完毕");
    }
    @Override
    public String toString() {
        return "Product [id=" + id + ", name=" + name + "]";
    }
}
```

实例化的工厂类：

```
public class Future {
    private Product product;
    private boolean down;
    public synchronized void setProduct(Product product){
        if(down){
            return;
        }
        this.product = product;
        this.down = true;
        notifyAll();
    }

    public synchronized Product getProduct(){
        while(!down){
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
        return this.product;
    }
}
```

创建另一个线程进行实例化类：

```
import java.util.Random;

public class ProductFactory {
    public Future createProduct(String name){
        Future f = new Future();  //创建一个订单
        System.out.println("下单成功，你可以去上班了");
        //生产产品
        new Thread(new Runnable() {
            @Override
            public void run() {
                Product p = new Product(new Random().nextInt(), name);
                f.setProduct(p);
            }
        }).start();
        return f;
    }
}
```

模拟使用 Future 设计模式：

```
public class Demo {
    public static void main(String[] args) {
        ProductFactory pf = new ProductFactory();
        //下单交钱
        Future f = pf.createProduct("蛋糕");
        System.out.println("我去上班了，下了班我来取蛋糕...");
```

```
            System.out.println("我拿着蛋糕回家" + f.getProduct());
        }
    }
```

解释一下信号量 (Semaphore)

Semaphore 的作用：

在 java 中，使用了 synchronized 关键字和 Lock 锁实现了资源的并发访问控制，在同一时间只允许唯一了线程进入临界区访问资源(读锁除外)，这样子控制的主要目的是为了解决多个线程并发同一资源造成的数据不一致的问题。在另外一种场景下，一个资源有多个副本可供同时使用，比如打印机房有多个打印机、厕所有多个坑可供同时使用，这种情况下，Java 提供了另外的并发访问控制--资源的多副本的并发访问控制，今天学习的信号量 Semaphore 即是其中的一种。

Semaphore 实现原理初探：

Semaphore 是用来保护一个或者多个共享资源的访问，Semaphore 内部维护了一个计数器，其值为可以访问的共享资源的个数。一个线程要访问共享资源，先获得信号量，如果信号量的计数器值大于 1，意味着有共享资源可以访问，则使其计数器值减去 1，再访问共享资源。

如果计数器值为 0,线程进入休眠。当某个线程使用完共享资源后，释放信号量，并将信号量内部的计数器加 1，之前进入休眠的线程将被唤醒并再次试图获得信号量。

就好比一个厕所管理员，站在门口，只有厕所有空位，就开门允许与空侧数量等量的人进入厕所。多个人进入厕所后，相当于 N 个人来分配使用 N 个空位。为避免多个人来同时竞争同一个侧卫，在内部仍然使用锁来控制资源的同步访问。

Semaphore 的使用：

Semaphore 使用时需要先构建一个参数来指定共享资源的数量，Semaphore 构造完成后即是获取 Semaphore、共享资源使用完毕后释放 Semaphore。

```
Semaphore semaphore = new Semaphore(10,true);
semaphore.acquire();
```

```
//do something here
semaphore.release();
```

下面的代码就是模拟控制商场厕所的并发使用：

```
public class ResourceManage {
    private final Semaphore semaphore ;
    private boolean resourceArray[];
    private final ReentrantLock lock;
    public ResourceManage() {
        this.resourceArray = new boolean[10];//存放厕所状态
        this.semaphore = new Semaphore(10,true);//控制 10 个共享资源的使用，
使用先进先出的公平模式进行共享;公平模式的信号量，先来的先获得信号量
        this.lock = new ReentrantLock(true);//公平模式的锁，先来的先选
        for(int i=0 ;i<10; i++){
            resourceArray[i] = true;//初始化为资源可用的情况
        }
    }
    public void useResource(int userId){
        semaphore.acquire();
        try{
            //semaphore.acquire();
            int id = getResourceId();//占到一个坑
            System.out.print("userId:"+userId+" 正 在 使 用 资 源 ， 资 源
id:"+id+"\n");
            Thread.sleep(100);//do something，相当于于使用资源
            resourceArray[id] = true;//退出这个坑
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            semaphore.release();//释放信号量，计数器加 1
```

```
        }
    }
    private int getResourceId(){
        int id = -1;
        lock.lock();
        try {
            //lock.lock();//虽然使用了锁控制同步，但由于只是简单的一个数组遍
历，效率还是很高的，所以基本不影响性能。
            for(int i=0; i<10; i++){
                if(resourceArray[i]){
                    resourceArray[i] = false;
                    id = i;
                    break;
                }
            }
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
        return id;
    }
}

public class ResourceUser implements Runnable{
    private ResourceManage resourceManage;
    private int userId;
    public ResourceUser(ResourceManage resourceManage, int userId) {
        this.resourceManage = resourceManage;
        this.userId = userId;
```

```
    }
    public void run(){
        System.out.print("userId:"+userId+"准备使用资源...\n");
        resourceManage.useResource(userId);
        System.out.print("userId:"+userId+"使用资源完毕...\n");
    }
    public static void main(String[] args){
        ResourceManage resourceManage = new ResourceManage();
        Thread[] threads = new Thread[100];
        for (int i = 0; i < 100; i++) {
            Thread thread = new Thread(new
ResourceUser(resourceManage,i));//创建多个资源使用者
            threads[i] = thread;
        }
        for(int i = 0; i < 100; i++){
            Thread thread = threads[i];
            try {
                thread.start();//启动线程
            }catch (Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

Fork/Join 框架

多线程的目的不仅仅是提高程序运行的性能，

但是可以充分利用 CPU 资源

Fork/Join 的出现是为了多核时代，提高 CPU 利用率，利用 Fork 将任务分开，利用 Join 将人物合并

ForkJoinTask 类就实现了这个问题

使用这个操作：

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;

public class Demo extends RecursiveTask<Integer> {
    private int begin;
    private int end;

    public Demo(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        System.out.println(Thread.currentThread().getName() + "... ");
        int sum = 0;
        //拆分任务
        if(end - begin <= 2){
            //计算
            for(int i = begin; i <= end; i++){
                sum += i;
            }
        } else {
            Demo d1 = new Demo(begin, (begin + end) / 2);
```

```
            Demo d2 = new Demo((begin + end) / 2, end);
            //执行自任务
            d1.fork();
            d2.fork();
            Integer a = d1.join();
            Integer b = d2.join();
            sum = a + b;
        }
        //拆分

        return sum;
    }

    public static void main(String[] args) throws Exception {
        ForkJoinPool pool = new ForkJoinPool();
        Future<Integer> future = pool.submit(new Demo(1, 100));
        System.out.println("....");
        System.out.println("计算的值为：" + future.get());
    }
}
```

消息队列

阻塞队列不用 java 提供的自己怎么实现，condition 和 wait 不能用；

了解其他的消息队列吗？（并不了解）

消息队列的两种形式：

点对点

发布-订阅

消息队列有哪些：

RabbitMQ（Erlang 编写）

Redis

ZeroMQ

ActiveMQ

Jafka/Kafka

实现锁定义

实现锁的方式？

synchronized 中的锁一般分为重量锁（对象锁），自旋锁，自适应自旋锁，轻量锁，偏向锁

使用 synchronized 关键字的代码块在加锁时，不会直接就加重量锁，因为重量锁对系统的开销最大。若一个线程等待获取锁对象所持续的时间非常短，这时适合使用自旋锁。所谓自旋锁，就是等待锁的线程并不进入阻塞状态，而是执行一个无意义的循环。在循环结束后查看锁是否已经被释放，若已经释放则直接进入执行状态。因为长时间无意义循环也会大量浪费系统资源，因此自旋锁适用于间隔时间短的加锁场景。

自适应自旋锁：系统根据运行时的统计信息，来调整自旋的次数。

轻量锁和偏向锁适用于没有线程竞争的情况。无法代替重量锁

作为程序员的你，下面这 14 种锁你都了解吗？

这里主要是讲锁的概念，这 14 种锁在概念范畴上可能有重叠，并不是完全的 14 种。

1.CAS

compare and set (比较更新，如果数据版本一致则更新否则不更新) 通过调用原子性汇编指令来实现原子性操作。如 java 中的 AtomicLong,AtomicInteger 等就是基于 CAS 实现。

2.偏向锁

加锁和解锁不需要额外消耗，和非同步方法仅纳秒级差距。适用于单线程或几乎无竞争的场景，如果竞争会升级为重量级锁带来额外消耗。

3.轻量级锁

竞争的线程不会阻塞，而是轮询访问。优点是响应快，缺点是消耗 CPU。

4.重量级锁

竞争的线程会阻塞，优点是不消耗 CPU，缺点是响应慢，适用于高吞吐量。

5.自旋锁

自旋锁即通过循环不断地调用 CAS 指令来实现原子性操作。自旋锁不改变线程状态，所以响应速度较快，但当并发量大时性能下降明显，因为其轮询时占用 CPU。AtomicLong 就是使用自旋锁的例子。

6.阻塞锁

阻塞锁即让线程进入阻塞状态以等待获取锁。synchronized 就是阻塞锁，Lock 既提供阻塞锁，又提供非阻塞锁。

阻塞锁的优点是等待时不占用 CPU，但进入锁和释放锁要比自旋锁慢。竞争激烈的情况下阻塞锁的性能要高于自旋锁。

7.可重入锁

可重入锁（递归锁）即同一线程外层方法获取到锁之后调用内层方法是可以再次获取到锁并且不会产生死锁。（不可重入锁也称非递归锁）。

8.读写锁

ReentrantLock 等是排他锁，在同一时刻只允许一个线程访问。读写锁在同一时刻允许多个读线程访问，一个写线程访问。读写锁通过分离读锁和写锁减少了锁的竞争。

JDK 中 ReentrantReadWriteLock 基于 AQS 实现了可重入式读写锁。

9.乐观锁

乐观锁总是乐观地认为这个数据在同一时刻没有人修改。

乐观锁的实现基于 CAS(Check And Set)，先检查当前数据版本再更新。

特点：响应速度较快，并发量大时性能较差。

应用：

atomic 系列都是乐观锁的体现

数据库版乐观锁：update t set n= newn, v = oldv+1 where v = oldv

10.悲观锁

悲观锁认为数据在同一时刻总有人在修改。

悲观锁是先获取锁，再更新。

缺点：竞争线程会阻塞，响应较慢

优点：阻塞线程不消耗 CPU，适用于高吞吐量

11.公平锁

公平锁按线程请求锁的先后顺序来分配锁，即 FIFO。公平锁要进行上下文切换，性能比非公平锁较差，除非业务需要公平性，一般不推荐使用。

12.非公平锁

非公平锁是随机抢占机制。不需要进行上下文切换，性能比公平锁较好。

13.死锁

死锁是两个以上的线程阻塞着等待其它处于死锁状态的线程所持有的锁。死锁通常发生在多个线程同时但以不同的顺序请求同一组锁的时候。

例如，如果线程 1 锁住了 A，然后尝试对 B 进行加锁，同时线程 2 已经锁住了 B，接着尝试对 A 进行加锁，这时死锁就发生了。由于线程 1 处在等待线程 2 释放锁 B 期间而不肯释放已占有的锁 A，由于线程 1 和线程 2 互相等待对方释放锁而自身又不释放锁，因此他们将永远阻塞下去，这就造成了死锁。

避免死锁：

1)加锁顺序：当多个线程访问多个锁时，如果访问的锁存在交集，则应使用相同的加锁顺序，否则可能导致死锁。

2)加锁时限：在尝试获取锁时设置超时时间，超时未能全部获取到所需锁则释放已获取的锁，随机等待然后重试。在等待的时间里别的线程就有机会获取交集中的已释放的锁，提高获取锁的成功率。

3)死锁检测：死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁和锁超时也不可行的场景。通过对访问锁的线程、访问的锁、已获取的锁作标记然后检测是否存在互相等待对方释放锁的场景，如果存在则死锁发生了。此时应该：让一方的线程释放所有锁，随机等待然后重试；更好的方案是给一部分线程设置随机的优先级，让优先级低的那些线程回退，从而避免死锁。

公平锁和非公平锁

ReentrantLock 的公平锁和非公平锁都委托了 AbstractQueuedSynchronizer#acquire 去请求获

取。

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

tryAcquire 是一个抽象方法，是公平与非公平的实现原理所在。

addWaiter 是将当前线程结点加入等待队列之中。公平锁在锁释放后会严格按照等到队列去取后续值，而非公平锁在对于新晋线程有很大优势。

acquireQueued 在多次循环中尝试获取到锁或者将当前线程阻塞。

selfInterrupt 如果线程在阻塞期间发生了中断，调用 Thread.currentThread().interrupt() 中断当前线程。

ReentrantLock 对线程的阻塞是基于 LockSupport.park(this); (见 AbstractQueuedSynchronizer#parkAndCheckInterrupt)。先决条件是当前节点有限次尝试获取锁失败。

公平锁和非公平锁在说的获取上都使用到了 volatile 关键字修饰的 state 字段，这是保证多线程环境下锁的获取与否的核心。

但是当并发情况下多个线程都读取到 state == 0 时，则必须用到 CAS 技术，一门 CPU 的原子锁技术，可通过 CPU 对共享变量加锁的形式，实现数据变更的原子操作。

volatile 和 CAS 的结合是并发抢占的关键。

公平锁 FairSync

公平锁的实现机理在于每次有线程来抢占锁的时候，都会检查一遍有没有等待队列，如果有，当前线程会执行如下步骤：

```
if (!hasQueuedPredecessors() &&
    compareAndSetState(0, acquires)) {
    setExclusiveOwnerThread(current);
    return true;
```

```
}
```

其中 hasQueuedPredecessors 是用于检查是否有等待队列的。

```
    public final boolean hasQueuedPredecessors() {
        Node t = tail; // Read fields in reverse initialization order
        Node h = head;
        Node s;
        return h != t &&
            ((s = h.next) == null || s.thread != Thread.currentThread());
    }
```

非公平锁 NonfairSync

非公平锁在实现的时候多次强调随机抢占：

```
if (c == 0) {
    if (compareAndSetState(0, acquires)) {
        setExclusiveOwnerThread(current);
        return true;
    }
}
```

与公平锁的区别在于新晋获取锁的进程会有多次机会去抢占锁。如果被加入了等待队列后则跟公平锁没有区别。

乐观锁和悲观锁

乐观锁

总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或 CAS 操作实现。

version 方式：一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时，version 值会加一。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。

核心 SQL 代码：

update table set x=x+1, version=version+1 where id=#{id} and version=#{version};

CAS 操作方式：即 compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

悲观锁

总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在 Java 中，synchronized 的思想也是悲观锁。

悲观锁和乐观锁的原理和应用场景

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

总结：两种所各有优缺点，读取频繁使用乐观锁，写入频繁使用悲观锁。

解释一下自旋

没有获得锁的调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，这就是自旋锁，他不用将线城阻塞起来（NON-BLOCKING)；

应用场景：

真正的自旋锁是针对多核 CPU，而往往应用是单进程的，所以我们见到的自旋锁是使用了自旋锁的概念的一种实现，却是针对多线程的。

怎么写一个会发生死锁的程序出来

真正理解什么是死锁，这个问题其实不难，几个步骤：

（1）两个线程里面分别持有两个 Object 对象：lock1 和 lock2。这两个 lock 作为同步代码块

的锁；

（2）线程 1 的 run()方法中同步代码块先获取 lock1 的对象锁，Thread.sleep(xxx)，时间不需要太多，50 毫秒差不多了，然后接着获取 lock2 的对象锁。这么做主要是为了防止线程 1 启动一下子就连续获得了 lock1 和 lock2 两个对象的对象锁

（3）线程 2 的 run)(方法中同步代码块先获取 lock2 的对象锁，接着获取 lock1 的对象锁，当然这时 lock1 的对象锁已经被线程 1 锁持有，线程 2 肯定是要等待线程 1 释放 lock1 的对象锁的

这样，线程 1”睡觉”睡完，线程 2 已经获取了 lock2 的对象锁了，线程 1 此时尝试获取 lock2 的对象锁，便被阻塞，此时一个死锁就形成了。代码就不写了，占的篇幅有点多。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

什么时候发生死锁？如何解决？（死锁产生的四大条件，通过破坏四个必要条件之一，如调整加锁顺序、设定加锁时限超时放弃、死锁检测、死锁避免的银行家算法可解决死锁问题）

处理死锁的思路。

预防死锁

破坏死锁的四个必要条件中的一个或多个来预防死锁。

避免死锁

和预防死锁的区别就是，在资源动态分配过程中，用某种方式防止系统进入不安全的状态。

检测死锁

运行时出现死锁，能及时发现死锁，把程序解脱出来

解除死锁

发生死锁后，解脱进程，通常撤销进程，回收资源，再分配给正处于阻塞状态的进程。

预防死锁方法

破坏请求和保持条件

协议 1

所有进程开始前，必须一次性地申请所需的所有资源，这样运行期间就不会再提出资源要求，破坏了请求条件，即使有一种资源不能满足需求，也不会给它分配正在空闲的资源，这样它就没有资源，就破坏了保持条件，从而预防死锁的发生。

协议 2

允许一个进程只获得初期的资源就开始运行，然后再把运行完的资源释放出来。然后再请求新的资源。

破坏不可抢占条件

当一个已经保持了某种不可抢占资源的进程，提出新资源请求不能被满足时，它必须释放已经保持的所有资源，以后需要时再重新申请。

破坏循环等待条件

对系统中的所有资源类型进行线性排序，然后规定每个进程必须按序列号递增的顺序请求资源。假如进程请求到了一些序列号较高的资源，然后有请求一个序列较低的资源时，必须先释放相同和更高序号的资源后才能申请低序号的资源。多个同类资源必须一起请求。

可重入锁为什么不会导致死锁？（因为上一个问题我回答了不可重入锁会导致死锁，面试官接着就问了可重入锁的原理，我就说了一下第一次加锁就获取该对象的 Monitor，当 Monitor 计数器不为 0 时，只有获得锁的线程才能再次获得锁，并且每次加锁 Monitor 计数器就会加一解锁就会减一，当计数为零就释放对象的锁了）

一个例子：

synchronized 标记的同步是要绑定一个对象的，不写的话实际上实际上就是 synchronized（this），即绑定当前对象，这个 this 对象就是锁（synchronized 中可以认为就是监视器），当 LoggingWidget 执行 dosomething 的时候获得了这把锁（this），那么他去调用父类（Widget）的 dosomthing 的时候，父类的 dosomething 方法也要得到这个锁（this），但是子类的这个方法还没有运行完毕，所以还持有这个锁，父类方法在等，子类不释放锁还拼命的让父类方法执行，却不知道父类方法在眼巴巴的等着这个锁，这样就死锁了・・・・・・

线程池

线程池的概念，里面的参数，拒绝策略等等？

https://blog.csdn.net/Ch97CKd/article/details/80745137

https://www.jianshu.com/p/5df6e38e4362

什么是线程池：

线程池是指在初始化一个多线程应用程序过程中创建一个线程集合，然后在需要执行新的任务时重用这些线程而不是新建一个线程。线程池中线程的数量通常完全取决于可用内存数量和应用程序的需求。然而，增加可用线程数量是可能的。线程池中的每个线程都有被分配一个任务，一旦任务已经完成了，线程回到池子中并等待下一次分配任务。

为什么使用线程池：

基于以下几个原因在多线程应用程序中使用线程是必须的：

1. 线程池改进了一个应用程序的响应时间。由于线程池中的线程已经准备好且等待被分配任务，应用程序可以直接拿来使用而不用新建一个线程。
2. 线程池节省了 CLR 为每个短生存周期任务创建一个完整的线程的开销并可以在任务完成后回收资源。
3. 线程池根据当前在系统中运行的进程来优化线程时间片。
4. 线程池允许我们开启多个任务而不用为每个线程设置属性。
5. 线程池允许我们为正在执行的任务的程序参数传递一个包含状态信息的对象引用。
6. 线程池可以用来解决处理一个特定请求最大线程数量限制问题。

线程池的优势：

1.降低资源消耗。通过重复利用以创建的线程降低线程和销毁造成的消耗。

2.提高响应速度。当任务到达时，如果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。线程时稀缺资源，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分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

线程池的饱和策略：

CallerRunsPolicy：只用调用者所在线程来运行任务

DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务

DiscardPolicy：不处理，丢弃掉

线程池源码解析：

参数认识：

1.corePoolSize：线程池的基本大小，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本路线能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不在创建。如果调用了线程池的 prestartAllCoreThreads 方法，线程池会提前创建并启动所有基本线程。

2.runnableTaskQueue：任务队列，用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。

ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按 FIFO（先进先出）排序元素，吞吐量通常要高于 ArrayBlockingQueue。静态工厂方法 Executors.newFixedThreadPool()使用了这个队列。

SychronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常高于 LinkedBlockingQueue。静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。

PriorityBlockingQueue：一个具有优先级的无限阻塞队列

3.maximumPoolSize：线程池最大大小，线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会在创建新的线程执行任务，值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

4.ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字，Debug 和定位问题时非常有帮助。

5.RejectedExecutionHandle（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交新的任务。这个策略默认情况下是 AbortPolicy，表示无

法处理新任务时抛出异常。

CallerRunsPolicy：只用调用者所在线程来运行任务

DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务

DiscardPolicy：不处理，丢弃掉

当然也可以根据应用场景需要来实现 RejectedExecutionHandler 接口自定义策略。如记录日志或持久化不能处理的任务。

6.keepAliveTime：线程活动保持时间，线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。

7.TimeUnit：线程活动保持时间的单位，可选单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒（MILLISECONDS），微妙（MICROSECONDS，千分之一秒）和毫微秒（MANOSECONDS，千分之一微妙）

线程池的其他变量

```
//线程池的控制状态：用来表示线程池的运行状态（整型的高 3 位）和运行的 worker 数量（低 29 位）
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
//29 位的偏移量
private static final int COUNT_BITS = Integer.SIZE - 3;
//最大容量（2^29 - 1）
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
```

线程池的运行状态：

```
//接受新任务并且处理已经进入阻塞队列的任务
private static final int RUNNING    = -1 << COUNT_BITS;
//不接受任务，但是处理已经进入阻塞队列的任务
private static final int SHUTDOWN   =  0 << COUNT_BITS;
//不接受新任务，不处理已经进入阻塞队列的任务并且中断正在运行的任务
private static final int STOP       =  1 << COUNT_BITS;
//所有任务都已经终止，workerCount 为 0，线程转化为 TIDYING 状态并且调用 terminated 钩子函数
```

```
private static final int TIDYING    =  2 << COUNT_BITS;
//terminated 钩子函数已经运行完成
private static final int TERMINATED =  3 << COUNT_BITS;
```

对于参数的详细解析另一篇文章：

```
private final BlockingQueue<Runnable> workQueue;
// 任务阻塞队列
private final ReentrantLock mainLock = new ReentrantLock();   // 互斥锁
private final HashSet<Worker> workers = new HashSet<Worker>();
// 线程集合.一个 Worker 对应一个线程
private final Condition termination = mainLock.newCondition();// 终止条件
private int largestPoolSize;          // 线程池中线程数量曾经达到过的最大值
private long completedTaskCount;      // 已完成任务数量
private volatile ThreadFactory threadFactory;
// ThreadFactory 对象，用于创建线程。
private volatile RejectedExecutionHandler handler;// 拒绝策略的处理句柄
private volatile long keepAliveTime;   // 线程池维护线程所允许的空闲时间
private volatile boolean allowCoreThreadTimeOut;
//是否允许核心线程也会 timeout,默认是 false
private volatile int corePoolSize;
// 线程池维护线程的最小数量，哪怕是空闲的
private volatile int maximumPoolSize;  // 线程池维护的最大线程数量
```

其中有几个重要的规则需要说明一下：

1、corePoolSize 与 maximumPoolSize：由于 ThreadPoolExecutor 将根据 corePoolSize 和 maximumPoolSize 设置的边界自动调整池大小，当执行 execute(java.lang.Runnable) 方法提交新任务时：

(1). 如果运行的线程少于 corePoolSize，则创建新线程来处理请求，即使其他辅助线程是空闲的；

(2). 如果设置的 corePoolSize 和 maximumPoolSize 相同，则创建的线程池是大小固定的，如果运行的线程与 corePoolSize 相同，当有新请求过来时，若 workQueue 任务阻塞队列未满，则将请求放入 workQueue 中，等待有空闲的线程从 workQueue 中取出任务并处理。

(3). 如果运行的线程多于 corePoolSize 而少于 maximumPoolSize，则仅当 workQueue 任务阻塞队列满时才创建新线程去处理请求；

(4). 如果运行的线程多于 corePoolSize 并且等于 maximumPoolSize，若 workQueue 任务阻塞队列已满，则通过 handler 所指定的策略来处理新请求；

(5). 如果将 maximumPoolSize 设置为基本的无界值（如 Integer.MAX_VALUE），则允许池适应任意数量的并发任务。

也就是说，处理任务的优先级为：

1. 核心线程 corePoolSize > 阻塞队列 workQueue > 最大线程 maximumPoolSize，如果三者都满了，使用 handler 处理被拒绝的任务。

2. 当池中的线程数大于 corePoolSize 的时候，多余的线程会等待 keepAliveTime 长的时间，如果无请求处理，就自行销毁。

corePoolSize：在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了 prestartAllCoreThreads()或者 prestartCoreThread()方法，从这 2 个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建 corePoolSize 个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为 0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到 corePoolSize 后，就会把到达的任务放到阻塞队列当中；

maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

2、 workQueue 线程池所使用的任务阻塞队列，该阻塞队列的长度决定了能够缓冲任务的最大数量，阻塞队列有以下三种选择：

```
ArrayBlockingQueue; //有界队列
LinkedBlockingQueue; //无界队列
SynchronousQueue; //特殊的一个队列，只有存在等待取出的线程时才能加入队列，可以说容量为 0，是无界队列
```

ArrayBlockingQueue 和 PriorityBlockingQueue 使用较少，一般使用 LinkedBlockingQueue 和

Synchronous。

线程池的排队策略与 BlockingQueue 有关。

缓冲队列有三种通用策略：

(1). 直接提交。工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性;

(2). 无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize（因此，maximumPoolSize 的值也就无效了）。当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性;

(3). 有界队列。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

3、 keepAliveTime 表示线程没有任务执行时最多保持多久时间会终止。

默认情况下，只有当线程池中的线程数大于 corePoolSize 时，keepAliveTime 才会起作用。直到线程池中的线程数不大于 corePoolSize。即当线程池中的线程数大于 corePoolSize 时。如果一个线程空闲的时间达到 keepAliveTime，则会终止，直到线程池中的线程数不超过 corePoolSize。

但是如果调用了 allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于 corePoolSize 时，

keepAliveTime 参数也会起作用，直到线程池中的线程数为 0；

4、 unit 参数 keepAliveTime 的时间单位，有 7 种取值，在 TimeUnit 类中有 7 种静态属性：

```
TimeUnit.DAYS;               //天
TimeUnit.HOURS;              //小时
TimeUnit.MINUTES;            //分钟
TimeUnit.SECONDS;            //秒
TimeUnit.MILLISECONDS;       //毫秒
TimeUnit.MICROSECONDS;       //微妙
TimeUnit.NANOSECONDS;        //纳秒
```


5.添加任务处理的流程

当一个任务通过 execute(Runnable)方法欲添加到线程池时：

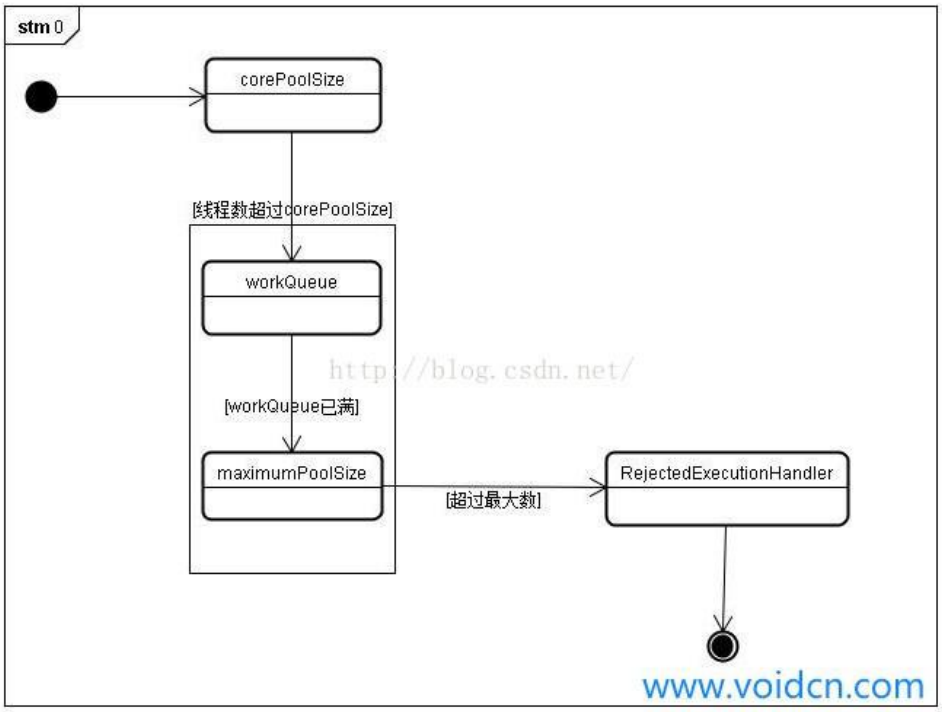
如果当前线程池中的数量小于 corePoolSize，并线程池处于 Running 状态，创建并添加的任务。

如果当前线程池中的数量等于 corePoolSize，并线程池处于 Running 状态，缓冲队列 workQueue 未满，那么任务被放入缓冲队列、等待任务调度执行。

如果当前线程池中的数量大于 corePoolSize，缓冲队列 workQueue 已满，并且线程池中的数量小于 maximumPoolSize，新提交任务会创建新线程执行任务。

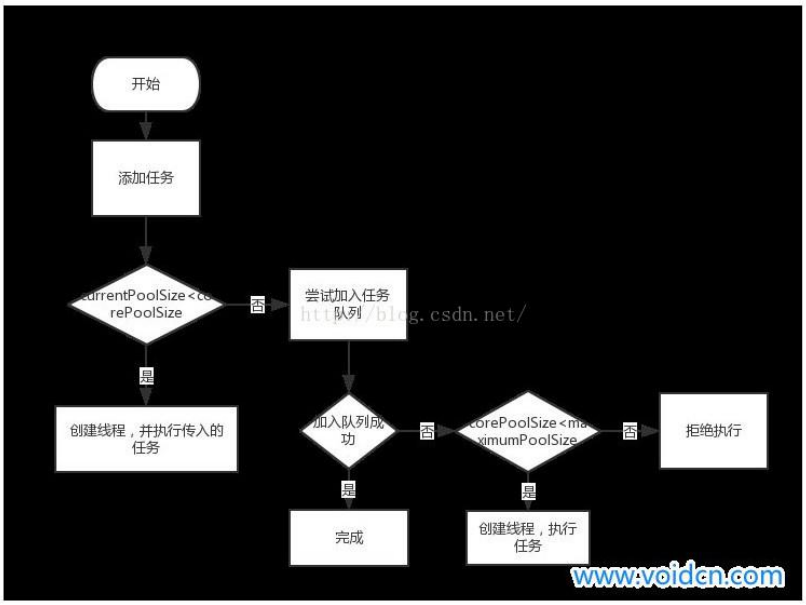
如果当前线程池中的数量大于 corePoolSize，缓冲队列 workQueue 已满，并且线程池中的数量等于 maximumPoolSize，新提交任务由 Handler 处理。

当线程池中的线程大于 corePoolSize 时，多余线程空闲时间超过 keepAliveTime 时，会关闭这部分线程。

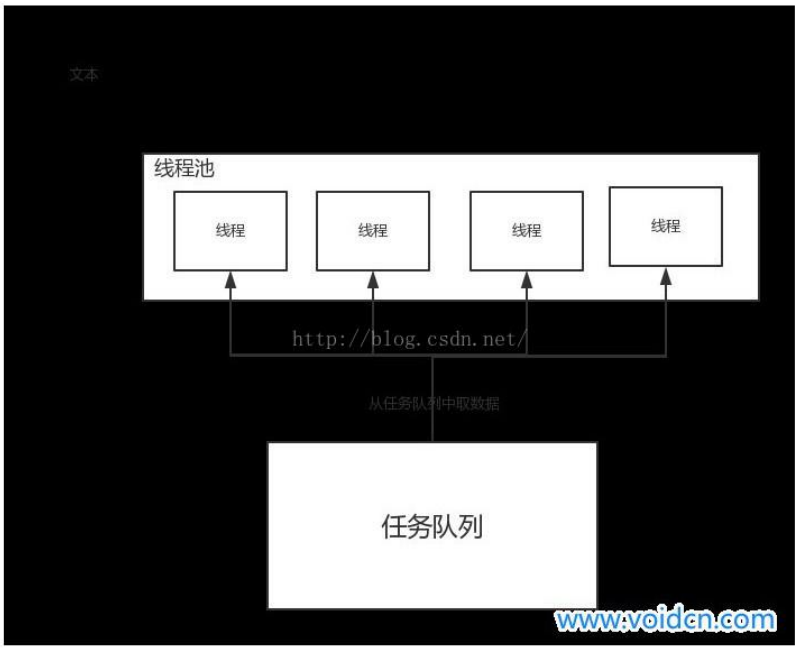


6、 我们先看两张图

任务执行流程图



任务队列图



几个关键点

corePoolSize: 核心线程池大小，线程池刚启动时来一个任务就会创建一个线程去执行这个任务，直到达到 corePoolSize 个线程，这时再来的任务就会加到任务队列中去，直到任务队列满了，这时会尝试创建新的线程，但是总线程数量必须少于 maximumPoolSize，如果线程也不能创建了，这时就会拒绝执行任务了。

maximumPoolSize: 最大线程数，线程池总线程数量不得多于 maximumPoolSize

keepAliveTime 线程存货时间，主要是在最大线程数到核心线程数之间的线程存活时间，比如 corePoolSize = 10，maximumPoolSize =100。当线程池达到 100 时，这时每个线程不执行且过了 keepAliveTime 的时间后就会消失，但是当线程池数量到达 corePoolSize 线程不在消失。也就是说 keepAliveTime 不起作用了。

Queue 任务队列 有三种类型（无限队列、直接队列、有限队列）

直接队列来一个任务会直接尝试创建新线程执行。

无限队列会当线程池数量达到 corePoolSize 时新来的任务都会加入队列中，这时 maximumPoolSize 就不起作用了。

有限队列在线程达到 corePoolSize 且任务队列已满时来一个任务会尝试继续创建线程，知道达到 maximumPoolSize

Executors 是创建线程池的工厂类。提供了以下几个主要方法

```
public static ExecutorService newCachedThreadPool() {
```

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

可以看出创建了一个核心线程数为 0，最大线程数为一个 Integer 能表示的最大值，存活时间为 60 秒。队列为 SynchronousQueue 直接队列。这种情况下每来一个任务就创建一个线程执行。直到创建的线程数大于 Integer.MAX_VALUE 当然一般情况下不会出现。每一个线程的存货时间是 60 秒超过 60 秒后自动销毁。

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

可以看出创建了一个核心线程数为传入的 nThreads，最大线程数也为 nThreads，存活时间为 0 秒。队列为 LinkedBlockingQueue 即不保存队列任务。这种情况下每来一个任务首先看线程池数量有没有到 nThreads，如果没有达到则创建一个新的线程并执行，否则加入队列中，等待执行。此处 LinkedBlockingQueue 没有指定大小即最多可以接收 Integer.MAX_VALUE 个缓冲任务。相当于创建了 nThreads 个线程来执行队列中的任务。

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

和上面的 newFixedThreadPool 类似，只不过这里只有一个线程，即使用一个线程执行任务队列中的任务。

--线程池,如何根据 CPU 的核数来设计线程大小，如果是计算机密集型的呢，如果是 IO 密集型的呢？

1.任务类型举例:

1.1: CPU 密集型:

例如,一般我们系统的静态资源,比如 js,css 等,会存在一个版本号,如 main.js?v0,每当用户访问这个资源的时候,会发送一个比对请求到服务端,比对本地静态文件版本和服务端的文件版本是否一致,不一致则更新.这种任务一般不占用大量 IO,所以后台服务器可以快速处理,压力落在 CPU 上.

1.2: I/O 密集型:

比方说近期我们做的万科 CRM 系统,常有大数据量的查询和批量插入操作,此时的压力主要在 I/O 上.

2.线程数与任务类型的关系:

2.1:与 CPU 密集型的关系:

一般情况下,CPU 核心数 == 最大同时执行线程数.在这种情况下(设 CPU 核心数为 n),大量客户端会发送请求到服务器,但是服务器最多只能同时执行 n 个线程.

设线程池工作队列长度为 m,且 m>>n,则此时会导致 CPU 频繁切换线程来执行(如果 CPU 使用的是 FCFS,则不会频繁切换,如使用的是其他 CPU 调度算法,如时间片轮转法,最短时间优先,则可能会导致频繁的线程切换).

所以这种情况下,无需设置过大的线程池工作队列,(工作队列长度 = CPU 核心数 || CPU 核心数+1) 即可.

2.2:与 I/O 密集型的关系:

1 个线程对应 1 个方法栈,线程的生命周期与方法栈相同.

比如某个线程的方法栈对应的入站顺序为:controller()->service()->DAO(),由于 DAO 长时间的 I/O 操作,导致该线程一直处于工作队列,但它又不占用 CPU,则此时有 1 个 CPU 是处于空闲状态的.

所以,这种情况下,应该加大线程池工作队列的长度(如果 CPU 调度算法使用的是 FCFS,则无法切换),尽量不让 CPU 空闲下来,提高 CPU 利用率.

Executor、Executors 和 ExecutorService 区别

Executor：是 Java 线程池的超级接口；提供一个 execute(Runnable command)方法;我们一般

用它的继承接口 ExecutorService。

Executors：是 java.util.concurrent 包下的一个类，提供了若干个静态方法，用于生成不同类型的线程池。Executors 一共可以创建下面这四类线程池：

newFixedThreadPool 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。

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。

newScheduledThreadPool 创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

newSingleThreadExecutor 创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

ExecutorService：它是线程池定义的一个接口，继承 Executor。有两个实现类，分别为 ThreadPoolExecutor,ScheduledThreadPoolExecutor。

线程池的继承树：

ExecutorService 常用的几个方法：

execute(Runnable)从父类继承过来的方法

submit(Runnable)

submit(Callable)

invokeAny(...)

invokeAll(...)

shutdown()

execute 方法：方法接收一个 Runnable 实例，并且异步的执行，请看下面的实例：

```
public class Demo1 {
    public static void main(String[] args) {
        ExecutorService                executorService                =
Executors.newSingleThreadExecutor(); //创建一个单线程
        executorService.execute(new Runnable() { //接收一个 Runnable 实例
```

```
            public void run() {
                System.out.println("Asynchronous task");
            }
        });
        executorService.shutdown();
    }
}
```

这个方法有个问题，就是没有办法获知 task 的执行结果。如果我们想获得 task 的执行结果，我们可以传入一个 Callable 的实例（下面会介绍）。

submit(Runnable)方法：返回一个 Future 对象，通过返回的 Future 对象，我们可以检查提交的任务是否执行完毕。

```
public class Demo2 {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newSingleThreadExecutor(); //创建一个单线程
        Future future = executorService.submit(new Runnable() { //接收一个 Runnable 实例
            public void run() {
                System.out.println("Asynchronous task");
            }
        });
        System.out.println(future.get());  // 任务执行结束返回 null.
executorService.shutdown();
    }
}
```

submit(Callable)：与 submit(Callable)类似，也会返回一个 Future 对象，但是除此之外，submit(Callable)接收的是一个 Callable 的实现，Callable 接口中的 call()方法有一个返回值，可以返回任务的执行结果，而 Runnable 接口中的 run()方法是 void 的，没有返回值。

```
public class Demo1 {
```

```
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService                executorService                =
Executors.newSingleThreadExecutor(); //创建一个单线程
        Future<Object>      future      =      executorService.submit(new
Callable<Object>() { //接收一个 Callable 实例
            public Object call() {
                System.out.println("Asynchronous task");
                return "Callable Result";
            }
        });
        System.out.println("future.get()="+future.get());
        executorService.shutdown();
    }
}
```

invokeAny(...)：方法接收的是一个 Callable 的集合，执行这个方法不会返回 Future，但是会返回所有 Callable 任务中其中一个任务的执行结果。这个方法也无法保证返回的是哪个任务的执行结果，反正是其中的某一个。

```
public class Demo2 {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService                executorService                =
Executors.newSingleThreadExecutor();
        Set<Callable<String>> callables = new HashSet<Callable<String>>();

        callables.add(new Callable<String>(){
            @Override
            public String call() throws Exception {
                // TODO Auto-generated method stub
```

```
                return "Result1";
            }

        });

        callables.add(new Callable<String>(){
            @Override
            public String call() throws Exception {
                // TODO Auto-generated method stub
                return "Result2";
            }

        });

        callables.add(new Callable<String>(){
            @Override
            public String call() throws Exception {
                // TODO Auto-generated method stub
                return "Result3";
            }

        });

        String result = executorService.invokeAny(callables);
        System.out.println(result);
        executorService.shutdown();
    }
}
```

invokeAll(...)：与 invokeAny(...)类似也是接收一个 Callable 集合，但是前者执行之后会返回一个 Future 的 List，其中对应着每个 Callable 任务执行后的 Future 对象。

```
public class Demo3 {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService                executorService                =
Executors.newSingleThreadExecutor();
        Set<Callable<String>> callables = new HashSet<Callable<String>>();

        callables.add(new Callable<String>(){
            @Override
            public String call() throws Exception {
                // TODO Auto-generated method stub
                return "Result1";
            }

        });

        callables.add(new Callable<String>(){
            @Override
            public String call() throws Exception {
                // TODO Auto-generated method stub
                return "Result2";
            }

        });

        callables.add(new Callable<String>(){
            @Override
            public String call() throws Exception {
                // TODO Auto-generated method stub
                return "Result3";
```

```
            }

        });

        List<Future<String>>                    futures                    =
executorService.invokeAll(callables);    //返回一个 Future 的 List 集合
        for(Future<String> future:futures){
            System.out.println("future.get()="+future.get());
        }
        executorService.shutdown();
    }
}
```

shutdown()：我们使用完成 ExecutorService 之后应该关闭它，否则它里面的线程会一直处于运行状态。

举个例子，如果的应用程序是通过 main()方法启动的，在这个 main()退出之后，如果应用程序中的 ExecutorService 没有关闭，这个应用将一直运行。之所以会出现这种情况，是因为 ExecutorService 中运行的线程会阻止 JVM 关闭。

如果要关闭 ExecutorService 中执行的线程，我们可以调用 ExecutorService.shutdown()方法。在调用 shutdown()方法之后，ExecutorService 不会立即关闭，但是它不再接收新的任务，直到当前所有线程执行完成才会关闭，所有在 shutdown()执行之前提交的任务都会被执行。如果我们想立即关闭 ExecutorService，我们可以调用 ExecutorService.shutdownNow()方法。这个动作将跳过所有正在执行的任务和被提交还没有执行的任务。但是它并不对正在执行的任务做任何保证，有可能它们都会停止，也有可能执行完成。

callable 原理

如果是一个多线程协作程序，比如菲波拉切数列，1，1，2，3，5，8...使用多线程来计算。 但后者需要前者的结果，就需要用 callable 接口了。

callable 用法和 runnable 一样，只不过调用的是 call 方法，该方法有一个泛型返回值类型，你可以任意指定

Callable 异步执行，应该不会陌生，那么在 java 中是怎么用的呢？又是如何实现的？下面我们循序渐进，慢慢分析。

先看一个例子，实现 Callable 接口，进行异步计算：

```
package com.demo;import java.util.concurrent.*;
public class Demo {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executor = Executors.newCachedThreadPool();
        Future<String> future = executor.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                System.out.println("call");
                TimeUnit.SECONDS.sleep(1);
                return "str";
            }
        });
        System.out.println(future.get());
    }
}
```

这段代码是很简单的一种方式利用 Callable 进行异步操作，结果自己可以执行下。

如何实现异步

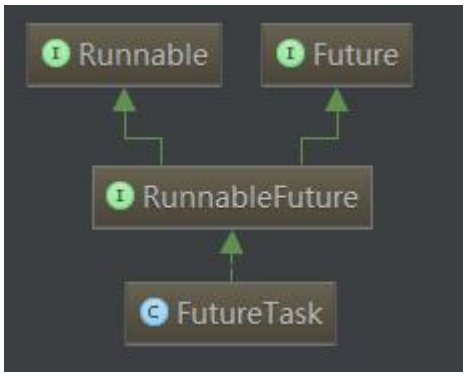
在不阻塞当前线程的情况下计算，那么必然需要另外的线程去执行具体的业务逻辑，上面代码中可以看到，是把 Callable 放入了线程池中，等待执行，并且立刻返回 futrue。可以猜想下，需要从 Future 中得到 Callable 的结果，那么 Future 的引用必然会被两个线程共享，一个线程执行完成后改变 Future 的状态位并唤醒挂起在 get 上的线程，到底是不是这样呢？

源码分析

首先我们从任务提交开始，在 AbstractExecutorService 中的源码如下：

```
    public <T> Future<T> submit(Callable<T> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task);
        execute(ftask);
        return ftask;
    }
       protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T
value) {
        return new FutureTask<T>(runnable, value);
    }
        public FutureTask(Callable<V> callable) {
        if (callable == null)
            throw new NullPointerException();
        this.callable = callable;
        this.state = NEW;       // ensure visibility of callable
    }
```

可以看到 Callable 任务被包装成了 RunnableFuture 对象，通过了线程池的 execute 方法提交任务并且立刻返回对象本身，而线程池是接受 Runnable，必然 RunnableFuture 继承了 Runnable，我们看下其继承结构。



从继承中可以清楚的看到，FutureTask 是 Runnable 和 Future 的综合。

到这里我们应该有些头绪了，关键点应该在 FutureTask 对象上，线程池不过是提供一个线程运行 FutureTask 中的 run 方法罢了。

FutureTask

从上面的分析，FutureTask 被生产者和消费者共享，生产者运行 run 方法计算结果，消费者通过 get 方法获取结果，那么必然就需要通知，如何通知呢，肯定是状态位变化，并唤醒线程。

FutureTask 状态

```
//FutureTask 类中用来保存状态的变量，下面常量就是具体的状态表示
private volatile int state;
private static final int NEW          = 0;
private static final int COMPLETING   = 1;
private static final int NORMAL       = 2;
private static final int EXCEPTIONAL  = 3;
private static final int CANCELLED    = 4;
private static final int INTERRUPTING = 5;
private static final int INTERRUPTED  = 6;
```

run 方法

```
//我修剪后的代码，可以看出其逻辑，执行 Callable 的 call 方法获取结果
 public void run() {
           Callable<V> c = callable;
           if (c != null && state == NEW) {
               V result = c.call();
               set(result);
           }
       }
       //把结果保存到属性字段中，finishCompletion 是最后的操作，唤醒等待结果的线程
       protected void set(V v) {
       if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING))
{
           outcome = v;
```

```
            UNSAFE.putOrderedInt(this, stateOffset, NORMAL);//正常结束设置
状态为 NORMAL
            finishCompletion();
        }
    }
    //waiters 是 FutureTask 类的等待线程包装类，以链表的形式连接多个，WaitNode
对象是在调用 get 方法时生成，并挂起 get 的调用者线程
     private void finishCompletion() {
        for (WaitNode q; (q = waiters) != null;) {
            if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null))
{
                for (;;) {
                    Thread t = q.thread;
                    if (t != null) {
                        q.thread = null;
                        LockSupport.unpark(t); //唤醒 get 上等待的线程
                        if (next == null)
                        break;
                    }
                   break;
            }
        }
    }
```

等待的线程

为了清除的看到如何挂起 get 的线程，我们可以分析下 get 的源码

```
    public V get() throws InterruptedException, ExecutionException {
        int s = state;
        if (s <= COMPLETING)
            s = awaitDone(false, 0L); //会一直挂起知道处理业务的线程完成，唤
```

```
醒等待线程
        return report(s);
    }
    public V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException
{
        if (unit == null)
            throw new NullPointerException();
        int s = state;
        if (s <= COMPLETING &&
            (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING)//
通过判断返回的状态是否为已完成做不同的处理
            throw new TimeoutException();
        return report(s);
    }private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        WaitNode q = null;
        boolean queued = false;
        for (;;) {
            int s = state;
            if (s > COMPLETING) {
                if (q != null)
                    q.thread = null;
                return s;
            }
            else if (s == COMPLETING) // cannot time out yet
                Thread.yield();
            else if (q == null)
                q = new WaitNode();
```

```
            else if (!queued)
                queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                     q.next = waiters, q);
            else if (timed) {  //如果是超时的 get 那么会挂起一段时间
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {//等待时间过后则会移除等待线程返回当前
futureTask 状态
                    removeWaiter(q);
                    return state;
                }
                LockSupport.parkNanos(this, nanos);
            }
            else
                LockSupport.park(this);
        }
    }
```

如果想搞明白可以自行研究下，这种经过优化的并发代码确实可读性差，基本原理就是生产者与消费者模型。

进程

你知道进程吗？有进程为何还有线程？

线程可以增加并发的程度啊。其实多进程也是可以并发，但是为什么要是线程呢？因为线程是属于进程的，是个轻量级的对象。所以再切换线程时只需要做少量的工作，而切换进程消耗很大。这是从操作系统角度讲。

从用户程序角度讲，有些程序在逻辑上需要线程，比如扫雷，它需要一个线程等待用户的输入，另一个线程的来更新时间。还有一个例子就是聊天程序，一个线程是响应用户输入，一个线程是响应对方输入。如果没有多线程，那么只能你说一句我说一句，你不说我这里就不能动，我还不能连续说。所以用户程序有这种需要，操作系统就要提供响应的机制

进程和线程的区别

一个程序最少需要一个进程，而一个进程最少需要一个线程。关系是线程－>进程－>程序的大致组成结构。所以线程是程序执行流的最小单位，而进程是系统进行资源分配和调度的一个独立单位。以下我们所有讨论的都是建立在线程基础之上。

线程的调度方式，线程和进程间通信

进程间交互方式了解不？

进程与线程的区别：

通俗的解释

一个系统运行着很多进程，可以比喻为一条马路上有很多马车

不同的进程可以理解为不同的马车

而同一辆马车可以有很多匹马来拉--这些马就是线程

假设道路的宽度恰好可以通过一辆马车

道路可以认为是临界资源

那么马车成为分配资源的最小单位(进程)

而同一个马车被很多匹马驱动(线程)--即最小的运行单位

每辆马车马匹数=1

所以马匹数=1 的时候进程和线程没有严格界限，只存在一个概念上的区分度

马匹数 1 的时候才可以严格区分进程和线程

专业的解释：

简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

线程的划分尺度小于进程，使得多线程程序的并发性高。另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行

资源分配和调度的一个独立单位.

线程是进程的一个实体,是 CPU 调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

一个线程可以创建和撤销另一个线程；同一个进程中的多个线程之间可以并发执行

进 程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个 进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序 健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。如果有兴趣深入的 话，我建议你们看看《现代操作系统》或者《操作系统的设计与实现》。对就个问题说得比较清楚。

+++

进程概念

进程是表示资源分配的基本单位，又是调度运行的基本单位。例如，用户运行自己的程序，系统就创建一个进程，并为它分配资源，包括各种表格、内存空间、磁盘空间、I/O 设备等。然后，把该进程放人进程的就绪队列。进程调度程序选中它，为它分配 CPU 以及其它有关资源，该进程才真正运行。所以，进程是系统中的并发执行的单位。

在 Mac、Windows NT 等采用微内核结构的操作系统中，进程的功能发生了变化：它只是资源分配的单位，而不再是调度运行的单位。在微内核系统中，真正调度运行的基本单位是线程。因此，实现并发功能的单位是线程。

线程概念

线 程是进程中执行运算的最小单位，亦即执行处理机调度的基本单位。如果把进程理解为在逻辑上操作系统所完成的任务，那么线程表示完成该任务的许多可能的子任 务之一。例如，假设用户启动了一个窗口中的数据库应用程序，操作系统就将对数据库的调用表示为一个进程。假设用户要从数据库中产生一份工资单报表，并传到 一个文件中，这是一个子任务；在产生工资单报表的过程中，用户又可以输人数据库查询请求，这又是一个子任务。这样，操作系统则把每一个请求－－工资单报表 和新输人的数据查询表示为数据库进程中的

独立的线程。线程可以在处理器上独立调度执行，这样，在多处理器环境下就允许几个线程各自在单独处理器上进行。操 作系统提供线程就是为了方便而有效地实现这种并发性

引入线程的好处

(1)易于调度。

(2)提高并发性。通过线程可方便有效地实现并发性。进程可创建多个线程来执行同一程序的不同部分。

(3)开销少。创建线程比创建进程要快，所需开销很少。

(4)利于充分发挥多处理器的功能。通过创建多线程进程(即一个进程可具有两个或更多个线程)，每个线程在一个处理器上运行，从而实现应用程序的并发性，使每个处理器都得到充分运行。

++

进程和线程的关系：

(1)一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。

(2)资源分配给进程，同一进程的所有线程共享该进程的所有资源。

(3)处理机分给线程，即真正在处理机上运行的是线程。

(4)线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。

线程是指进程内的一个执行单元,也是进程内的可调度实体.

与进程的区别：

(1)调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位

(2)并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行

(3)拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源.

(4)系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。+++

进程间的通信方式：

1.管道(pipe)及有名管道(named pipe)：

管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

2.信号(signal)：

信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一致的。

3.消息队列(message queue)：

消息队列是消息的链接表，它克服了上两种通信方式中信号量有限的缺点，具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息。

4.共享内存(shared memory)：

可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。

5.信号量(semaphore)：

主要作为进程之间及同一种进程的不同线程之间得同步和互斥手段。

6.套接字(socket)；

这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

++

线程之间的同步通信：

1.信号量二进制信号量互斥信号量整数型信号量记录型信号量

2.消息消息队列消息邮箱

3.事件 event

互斥型信号量：必须是同一个任务申请，同一个任务释放，其他任务释放无效。同一个任务可以递归申请。(互斥信号量是二进制信号量的一个子集)

二进制信号量：一个任务申请成功后，可以由另一个任务释放。(与互斥信号量的区别)

整数型信号量：取值不局限于 0 和 1,可以一个任务申请，另一个任务释放。(包含二进制信号量，二进制信号量是整数型信号量的子集)

二进制信号量实现任务互斥：

打印机资源只有一个，a bc 三个任务共享，当 a 取得使用权后，为了防止其他任务错误地释放了信号量(二进制信号量允许其他任务释放)，必须将打印机房的门关起来(进入临界段)，

用完后，释放信号量，再把门打开(出临界段)，其他任务再进去打印。(而互斥型信号量由于必须由取得信号量的那个任务释放，故不会出现其他任务错误地释放了信号量的情况出现，故不需要有临界段。互斥型信号量是二进制信号量的子集。)

二进制信号量实现任务同步：

a 任务一直等待信号量，b 任务定时释放信号量，完成同步功能

记录型信号量(record semaphore)：

每个信号量 s 除一个整数值 value(计数)外，还有一个等待队列 List，其中是阻塞在该信号量的各个线程的标识。当信号量被释放一个，值被加一后，系统自动从等待队列中唤醒一个等待中的线程，让其获得信号量，同时信号量再减一。

+++

同步和互斥的区别：

当 有多个线程的时候，经常需要去同步这些线程以访问同一个数据或资源。例如，假设有一个程序，其中一个线程用于把文件读到内存，而另一个线程用于统计文件中 的字符数。当然，在把整个文件调入内存之前，统计它的计数是没有意义的。但是，由于每个操作都有自己的线程，操作系统会把两个线程当作是互不相干的任务分 别执行，这样就可能在没有把整个文件装入内存时统计字数。为解决此问题，你必须使两个线程同步工作。

所 谓互斥，是指散布在不同进程之间的若干程序片断，当某个进程运行其中一个程序片段时，其它进程就不能运行它们之中的任一程序片段，只能等到该进程运行完这 个程序片段后才可以运行。如果用对资源的访问来定义的话，互斥某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资 源的访问顺序，即访问是无序的

所谓同步，是指散步在不同进程之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。如果用对资源的访问来定义的话，同步是指在互斥的基础上(大多数情况)，通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源

concurrent 包

介绍一下 Java 并发包(并发容器，同步设备，原子对象，锁，fork-join，执行器，详细介绍了 concurrent Hashmap，Countdownlatch 的底层实现及应用场景)

还用过并发包哪些类

concurrent 包里的一些类了解吗，原理是什么

atomicinteger 如何实现，什么是 CAS

为什么使用 AtomicLong 而不使用 Long?AtomicLong 的底层是怎么实现的？

AtomicInteger 底层原理

同步

同步和异步的区别，同步的实现原理(加锁，Java 种锁的底层实现，AQS,CAS)

1、同步、异步有什么区别

在进行网络编程时，我们通常会看到同步、异步、阻塞、非阻塞四种调用方式以及他们的组合。

其中同步方式、异步方式主要是由客户端（client）控制的，具体如下：

同步（Sync）

所谓同步，就是发出一个功能调用时，在没有得到结果之前，该调用就不返回或继续执行后续操作。

根据这个定义，Java 中所有方法都是同步调用，应为必须要等到结果后才会继续执行。我们在说同步、异步的时候，一般而言是特指那些需要其他端协作或者需要一定时间完成的任务。

简单来说，同步就是必须一件一件事做，等前一件做完了才能做下一件事。

例如：B/S 模式中的表单提交，具体过程是：客户端提交请求->等待服务器处理->处理完毕

返回，在这个过程中客户端（浏览器）不能做其他事。

异步（Async）

异步与同步相对，当一个异步过程调用发出后，调用者在没有得到结果之前，就可以继续执行后续操作。当这个调用完成后，一般通过状态、通知和回调来通知调用者。对于异步调用，调用的返回并不受调用者控制。

对于通知调用者的三种方式，具体如下：

状态

即监听被调用者的状态（轮询），调用者需要每隔一定时间检查一次，效率会很低。

通知

当被调用者执行完成后，发出通知告知调用者，无需消耗太多性能。

回调

与通知类似，当被调用者执行完成后，会调用调用者提供的回调函数。

例如：B/S 模式中的 ajax 请求，具体过程是：客户端发出 ajax 请求->服务端处理->处理完毕执行客户端回调，在客户端（浏览器）发出请求后，仍然可以做其他的事。

总结来说，同步和异步的区别：请求发出后，是否需要等待结果，才能继续执行其他操作。

线程同步有哪几种方式

【1】同步代码方法

sychronized 关键字修饰的方法

【2】同步代码块

sychronized 关键字修饰的代码块

【3】使用特殊变量域 volatile 实现线程同步

volatile 关键字为域变量的访问提供了一种免锁机制

【4】使用重入锁实现线程同步。reentrantlock 类是可冲入、互斥、实现了 lock 接口的锁

他与 sychronized 方法具有相同的基本行为和语义

同步接口和异步接口区别（这个当时没听过啊）

ava 中交互方式分为同步和异步两种：

同步交互：指发送一个请求,需要等待返回,然后才能够发送下一个请求，有个等待过程；

异步交互：指发送一个请求,不需要等待返回,随时可以再发送下一个请求，即不需要等待。

区别：一个需要等待，一个不需要等待，在部分情况下，我们的项目开发中都会优先选择不需要等待的异步交互方式。

同步接口（英文：Synchronous Serial Interface，SSI）是一种常用的工业用通信接口。ARM、飞思卡尔、德州仪器、美国国家半导体等公司都支持这种接口。在这种接口协议下，每一响应数据帧的长度可在 4-16 位之间变化，数据帧总长度可达 25 位。

扩展知识：同步通信必须以相同的时钟频率进行，大多数串口都是采用的同步方式进行通信的，与其相对应的为：异步接口。

并发

系统如何提高并发性

说一下并行和并发的区别

一：

并发是指一个处理器同时处理多个任务。

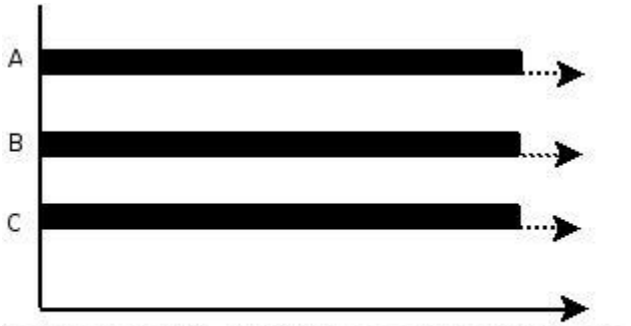
并行是指多个处理器或者是多核的处理器同时处理多个不同的任务。

并发是逻辑上的同时发生（simultaneous），而并行是物理上的同时发生。

来个比喻：并发是一个人同时吃三个馒头，而并行是三个人同时吃三个馒头。

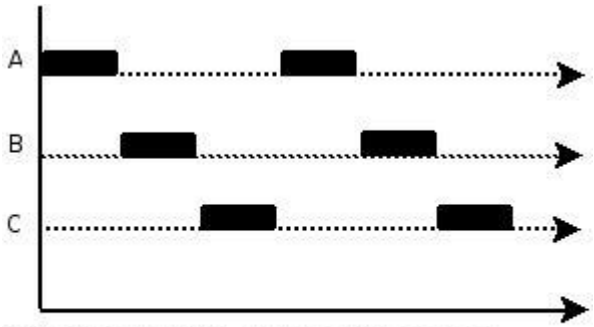
二：

并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。就好像两个人各拿一把铁锹在挖坑，一小时后，每人一个大坑。所以无论从微观还是从宏观来看，二者都是一起执行的。



Parallelism : 1. Multiprocessores, Multicore
2. Physically simultaneous processing

并发(concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。这就好像两个人用同一把铁锹，轮流挖坑，一小时后，两个人各挖一个小一点的坑，要想挖两个大一点得坑，一定会用两个小时。



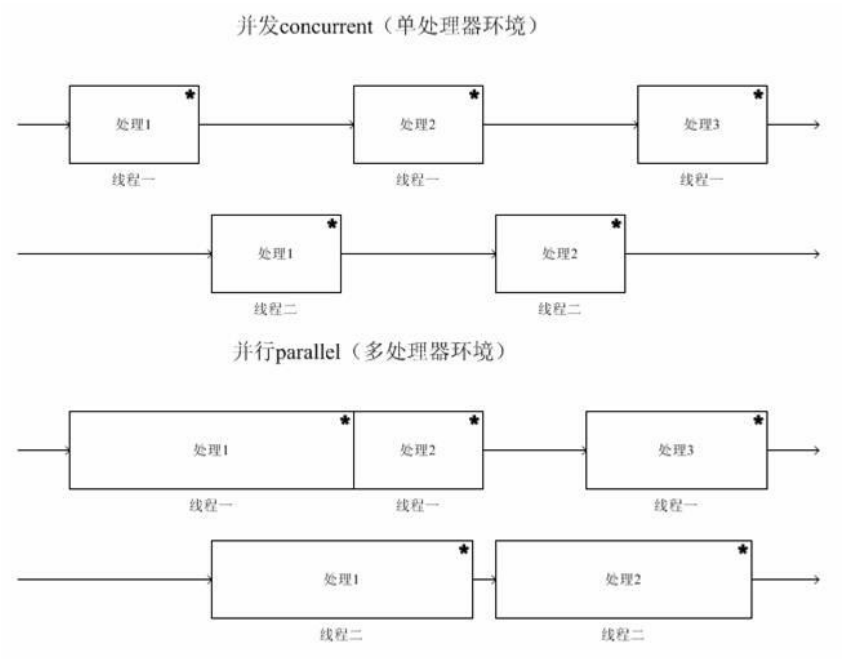
Concurrency : 1. Single Processor
2. logically simultaneous processing

并行在多处理器系统中存在，而并发可以在单处理器和多处理器系统中都存在，并发能够在单处理器系统中存在是因为并发是并行的假象，并行要求程序能够同时执行多个操作，而并发只是要求程序假装同时执行多个操作（每个小时间片执行一个操作，多个操作快速切换执行）。

三：

当有多个线程在操作时,如果系统只有一个 CPU,则它根本不可能真正同时进行一个以上的线程,它只能把 CPU 运行时间划分成若干个时间段,再将时间段分配给各个线程执行,在一个时间段的线程代码运行时,其它线程处于挂起状态.这种方式我们称之为并发(Concurrent)。

当系统有一个以上CPU时,则线程的操作有可能非并发.当一个CPU执行一个线程时,另一个CPU可以执行另一个线程,两个线程互不抢占CPU资源,可以同时进行,这种方式我们称之为并行(Parallel)。



高并发、高并发、高并发！重要的说三遍。如何解决？答得不太好，一脸懵逼！

简单的问了问，然后问我高并发怎么优化，这方面不会啊

说说并发，里面包括什么

应用场景题

1.一个接口，要去调用另外 5 个接口，每一个接口都会返回数据给这个调用接口，调用接口要对数据进行合并并返回给上层。

这样一种场景可能用到并发包下的哪些类？你会怎么去实现这样的业务场景？

2.三个线程顺序执行
信号量，wait notify，加锁，问我还有没办法。join？

3.给个淘宝场景，怎么设计一消息队列

4.大量数据，高并发访问如何优化

5.线程是不是开的越多越好，开多少合适，如何减少上下文切换开销，如何写个 **shell** 脚本获取上下文切换的开销？

6.火车票抢票，只有一台服务器，瞬时访问量很大，如何系统的解决？

7.i++,线程 **A**：**i++,**线程 **B**：**i--**，在非线程安全的情况下，**i** 有几种取值，采用什么方法使得 **i** 线程安全

8.怎么解决秒杀，瞎说的，不太会

9.消费者生产者模式怎么设计的，如果中间一个篮子只能放一个苹果，生产者和消费者各只有一个，怎么设计，如果很多线程又怎么设计

10.A 和 **B** 相互转账可能会发生死锁，设计程序避免这种情况

11.两个线程打印 **1.2.3.4** 打印到 **100** 怎么实现，这里刚开始说的是加锁用生产者消费者来做，后来说了 **semaphore**，感觉后面的才是面试官想要的答案。

12.消息队列的生产者消费者中消费者没有收到消息怎么办，消息有顺序比如 **1.2.3** 但是收到的却是 **1.3.2** 怎么办？消息发过来的过程中损坏或者出错怎么办

13.高并发场景的限流，你怎么来确定限流限多少，模拟场景和实际场景有区别怎么解决，动态改变限流阀值遇到的问题

14.自己设计一个数据库连接池怎么设计；