## 多线程的编写

1 多线程编写套路------上

\* 1.1 线程 操作(实例方法) 资源类

\* 1.2 高内聚 低耦合

\*

\* 2 多线程编写套路------下

\* 2.1 判断

\* 2.2 干活

\* 2.3 通知

\*/

## Lambda表达式

\* 1 拷贝中括号+写死右箭头+落地大括号

\* 2 一个接口里面有且仅有一个方法的接口,才可以使用Lambda Express

\* 3 @FunctionalInterface

\* 4 default默认方法实现

\* 5 静态方法实现

## 锁笔记

一个对象里面如果有多个synchronized方法，某一个时刻内，只要一个线程去调用其中的一个synchronized方法了，

其它的线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些synchronized方法

锁的是当前对象this，被锁定后，其它的线程都不能进入到当前对象的其它的synchronized方法

\*

\* 加个普通方法后发现和同步锁无关

换成两个对象后，不是同一把锁了，情况立刻变化。

锁定的是当前对象this,类锁,锁定的是整个class,是两个不同的对象

都换成静态同步方法后，情况又变化

所有的非静态同步方法用的都是同一把锁——实例对象本身，

也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，

可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，

所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。

所有的静态同步方法用的也是同一把锁——类对象本身，

这两把锁是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。

但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，

还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

## 获取多线程的第三种方法Callable方法

区别:

1. 有返回值获取返回值的方法是get 注意: get方法最好 是放在末尾; 因为不影响主线程的运行;
2. 方法名不同
3. 依赖泛型
4. 一个抛异常一个不抛异常
5. 面向接口编程 找一个既实现了Callable接口也实现了[Runnable](mk:@MSITStore:D:\尚硅谷教学\工作资料\开发文档\jdk1.8.chm::/java/lang/Runnable.html)接口是类 FutureTask类
6. 方法名不同一个方法名是Runnable一个是Callable
7. 多线的的流程控制

在主线程中需要执行比较耗时的操作时，但又不想阻塞主线程时，可以把这些作业交给Future对象在后台完成，

当主线程将来需要时，就可以通过Future对象获得后台作业的计算结果或者执行状态。

一般FutureTask多用于耗时的计算，主线程可以在完成自己的任务后，再去获取结果。

仅在计算完成时才能检索结果；如果计算尚未完成，则阻塞 get 方法。一旦计算完成，

就不能再重新开始或取消计算。get方法而获取结果只有在计算完成时获取，否则会一直阻塞直到任务转入完成状态，

然后会返回结果或者抛出异常。

只计算一次

get方法放到最后

二. 多线程的

\*让一些线程阻塞直到另一些线程完成一系列操作后才被唤醒。

\*

\* CountDownLatch主要有两个方法，当一个或多个线程调用await方法时，这些线程会阻塞。

\* 其它线程调用countDown方法会将计数器减1(调用countDown方法的线程不会阻塞)，

\* 当计数器的值变为0时，因await方法阻塞的线程会被唤醒，继续执行。

\*

\* 解释：5个同学陆续离开教室后值班同学才可以关门。

\* 也即 秦灭6国，一统华夏

\* main主线程必须要等前面5个线程完成全部工作后，自己才能开干

## CyclicBarrier

\* 的字面意思是可循环（Cyclic）使用的屏障（Barrier）。它要做的事情是，

\* 让一组线程到达一个屏障（也可以叫同步点）时被阻塞，

\* 直到最后一个线程到达屏障时，屏障才会开门，所有

\* 被屏障拦截的线程才会继续干活。

\* 线程进入屏障通过CyclicBarrier的await()方法。

\*