**如何在编程中尽量预防死锁呢？**

# 线程基础用法

## Thread

Thread t = new Thread(new MessageLoop());

t.join() --> t线程获得无限时间执行，且当前主线程暂停，直到t线程执行完毕后主线程可获得执行权限。（导致当前线程暂停执行，直到 t 的线程终止。）

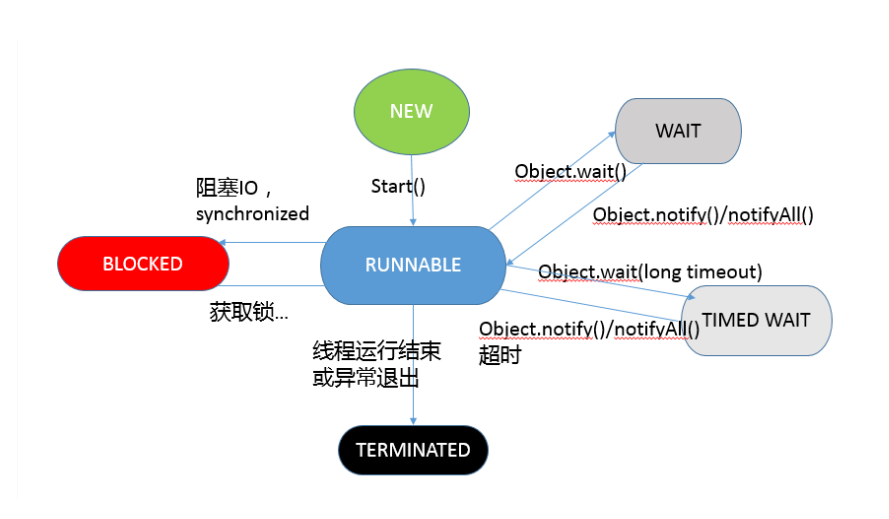
t.join(1000) --> t线程获得1秒的执行权限，1秒后当前主线程获得执行权限。（当前线程暂停1秒执行， t线程获得1秒执行时间）

t.interrupt(); 设置线程停止

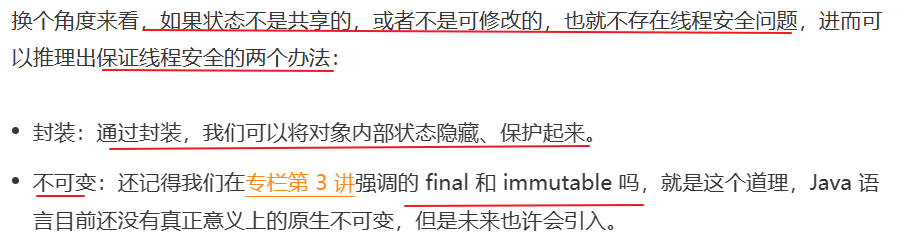
Thread.interrupted(); 获取停止标志，在run方法中。

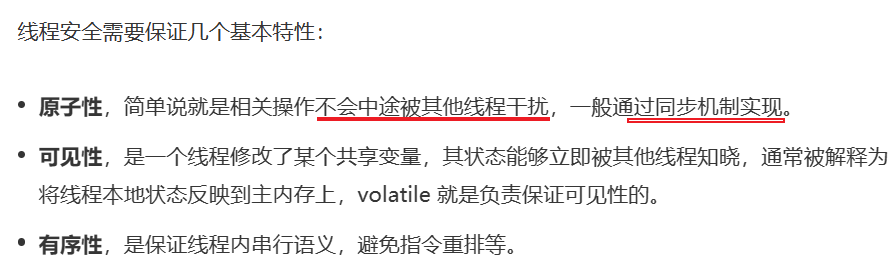
wait()；--> 在第一个个线程的synchronized方法中调用wait(),首先检查是否获取内部锁，然后释放锁(放弃此对象上的所有同步声明)且当前线程挂起。在第二个线程将获得相同的锁并调用 Object.notifyAll()后,第一个线程将与他线程竞争在 对象上同步 的权利或被唤醒，重新获得锁后，此线程继续执行后续操作。

### 线程状态



# 线程安全





## synchronized

synchronized 方法：

synchronized 修饰方法时锁定的是调用该方法的对象。也就是说：在同一个对象上的两个同步方法调用不可能交错。当一个线程为一个对象执行同步方法时，所有其他线程调用同一个对象块的同步方法(暂停执行) ，直到第一个线程完成对象。

<https://www.cnblogs.com/hapjin/p/4678773.html>

synchronized 代码块:

在方法中设定同步锁，案例如下:

private Object lock2 = new Object();

public void inc1() {

synchronized(lock1) {

c1++;

}

或

synchronized(this) { c1++;}

}

## 重入锁

Lock 对象相对于隐式锁的最大优势在于它们能够退出获取锁的尝试。如果锁没有立即可用，tryLock 方法会退出。解决死锁问题。对象化相互引用。使用方法如下：

private final Lock lock = new ReentrantLock();

lock.tryLock();

lock.unlock();

案例: <https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html>

## 线程池

最小化由于线程创建而产生的开销（线程对象使用大量的内存，在大规模应用程序中，分配和释放许多线程对象会产生大量的内存管理开销。）

这种类型的池总是有指定数量的线程在运行; 如果某个线程在仍在使用时以某种方式终止，则会自动用一个新线程替换它。任务通过一个内部队列提交给池，当活动任务比线程多时，该队列保存额外的任务。

# 如何在编程中尽量预防死锁呢？

死锁发生的基本条件如下：

1.互斥条件，类似 Java 中 Monitor 都是独占的，要么是我用，要么是你用。

2.互斥条件是长期持有的，在使用结束之前，自己不会释放，也不能被其他线程抢占。

3.循环依赖关系，两个或者多个个体之间出现了锁的链条环。

因此避免死锁的思路和方法如下：

第一种方法

如果可能的话，尽量避免使用多个锁，并且只有需要时才持有锁。

第二种方法

如果必须使用多个锁，尽量设计好锁的获取顺序，这个说起来简单，做起来可不容易。辅助手段，比如

1.将对象（方法）和锁之间的关系，用图形化的方式表示分别抽取出来

2.然后根据对象之间组合、调用的关系对比和组合，考虑可能调用时序。

3.按照可能时序合并，发现可能死锁的场景。

第三种方法

使用带超时的方法，为程序带来更多可控性。

# Java并发有哪些工具去使用



# 队列

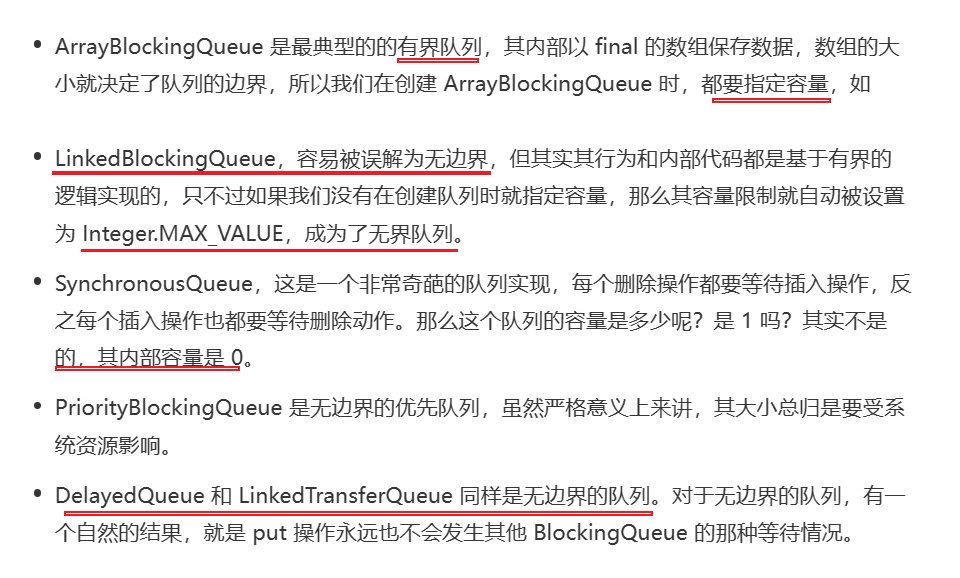
Deque接口队列

队列的 头尾都可进行插入和删除。

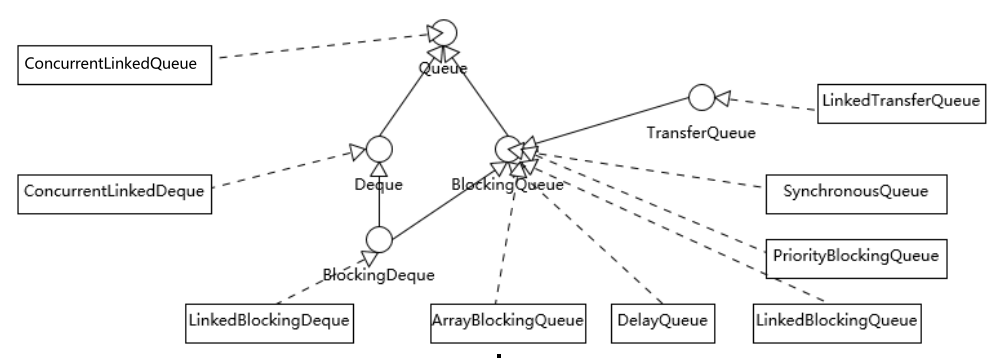
BlockingQueue接口队列

当获取时 等待元素进队，或者插入时 等待队列出现空位。

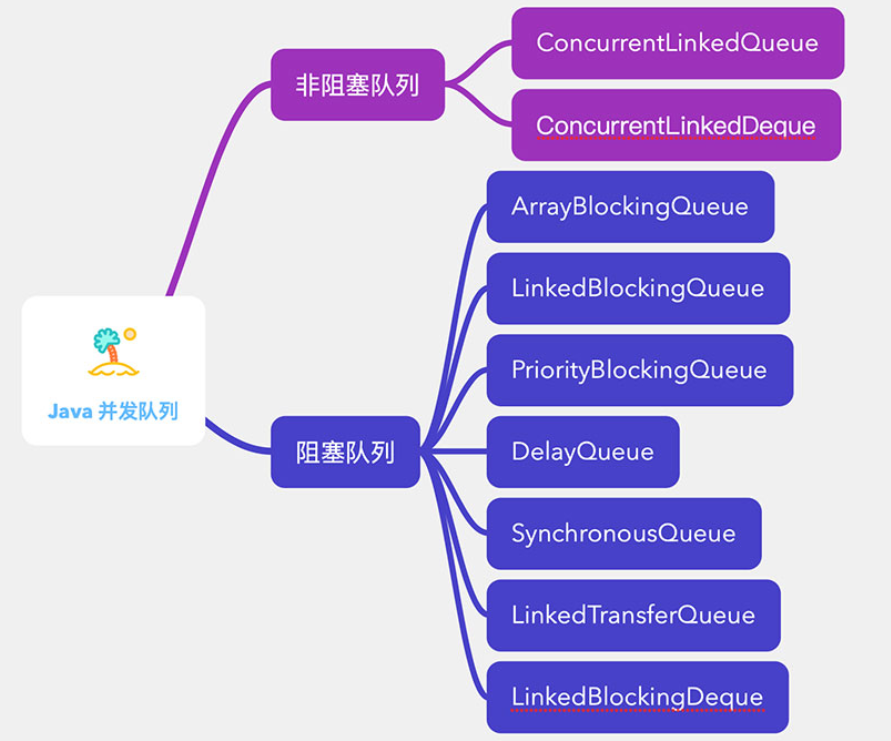
## 队列是否有界



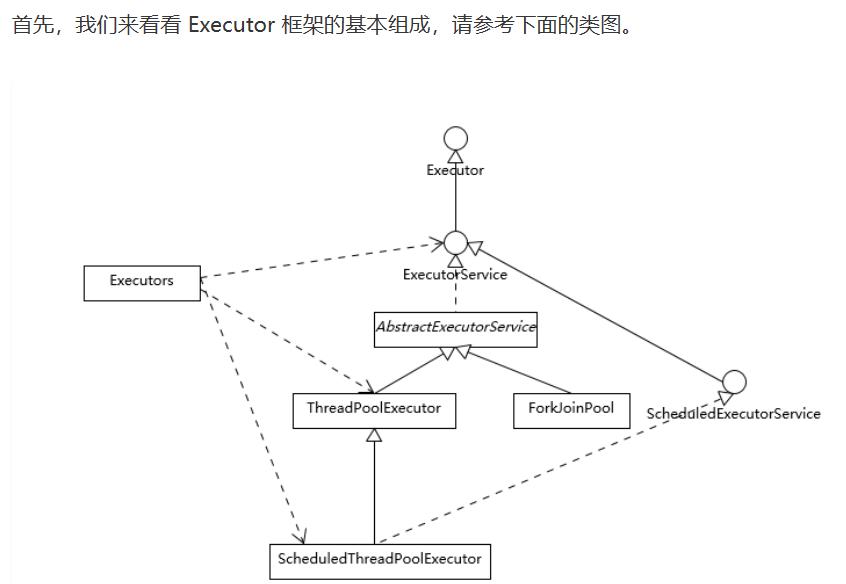
## 队列关系图



## 队列分类



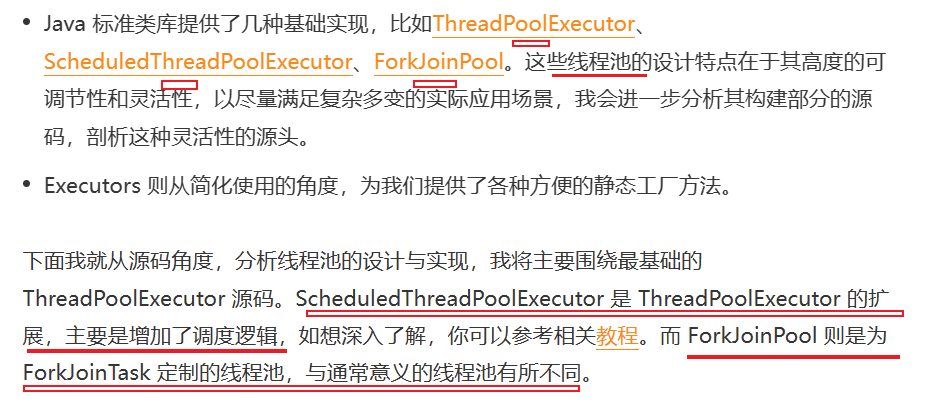
# 线程池



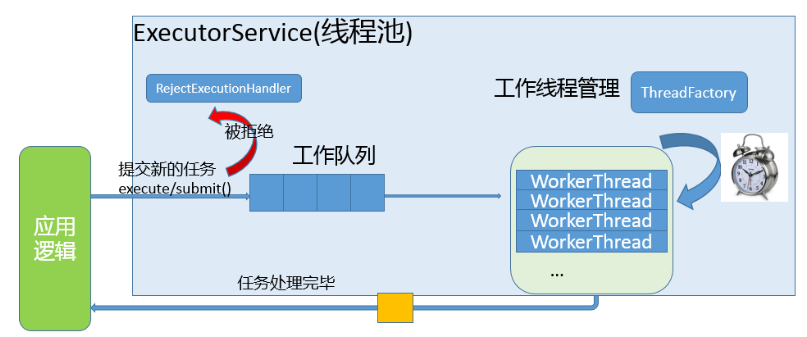
## 创建线程池的方法



## 线程池的种类

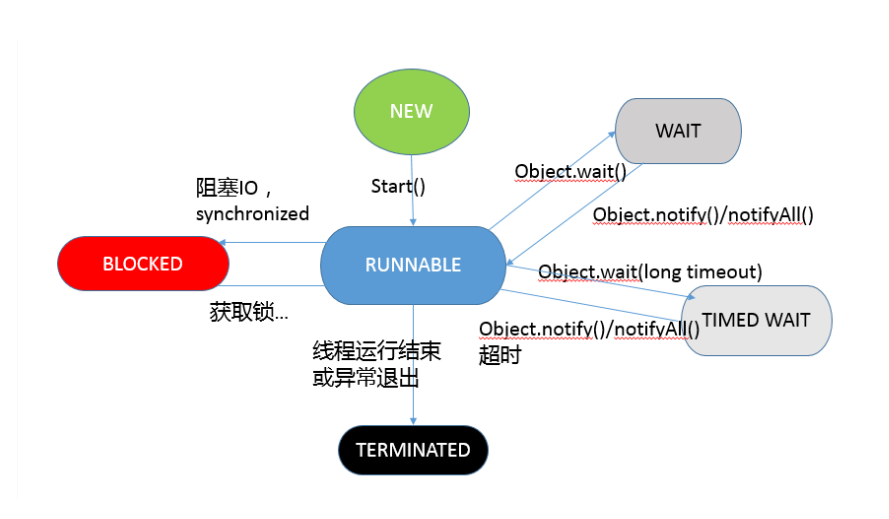
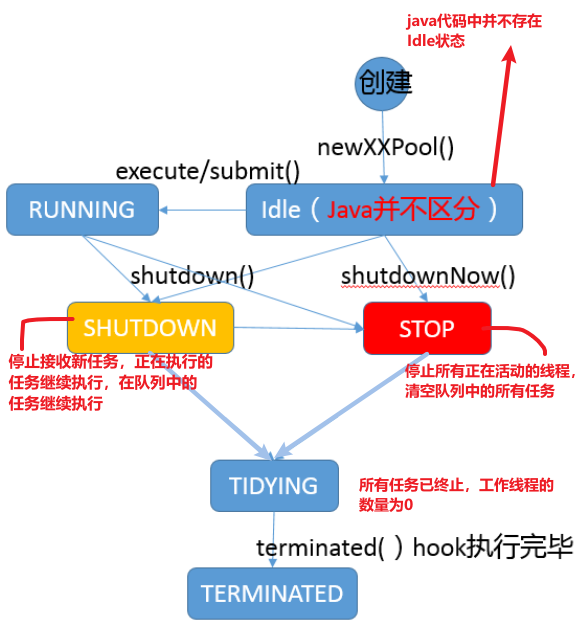


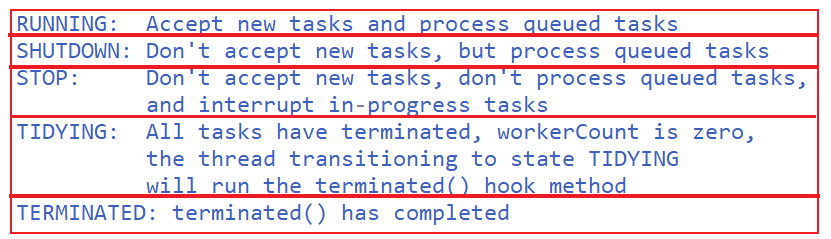
## 应用与线程池的交互和线程池的内部工作过程



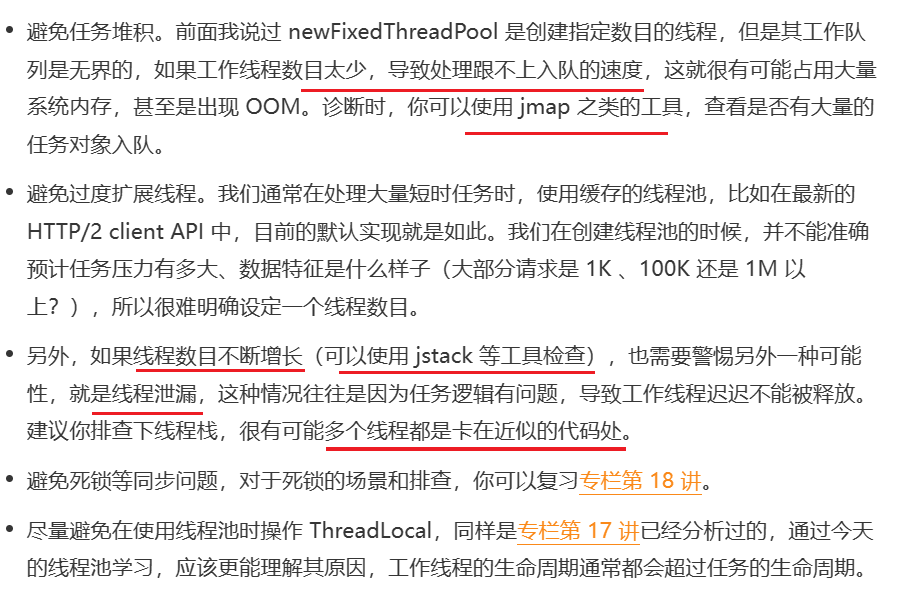
## 线程池中生命周期

[java](eclipse-javadoc:%E2%98%82=testJava/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_152%5C/jre%5C/lib%5C/rt.jar=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/javase%5C/8%5C/docs%5C/api%5C/=/%3Cjava).[util](eclipse-javadoc:%E2%98%82=testJava/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_152%5C/jre%5C/lib%5C/rt.jar=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/javase%5C/8%5C/docs%5C/api%5C/=/%3Cjava.util).[concurrent](eclipse-javadoc:%E2%98%82=testJava/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_152%5C/jre%5C/lib%5C/rt.jar=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/javase%5C/8%5C/docs%5C/api%5C/=/%3Cjava.util.concurrent).ThreadPoolExecutor.ThreadPoolExecutor

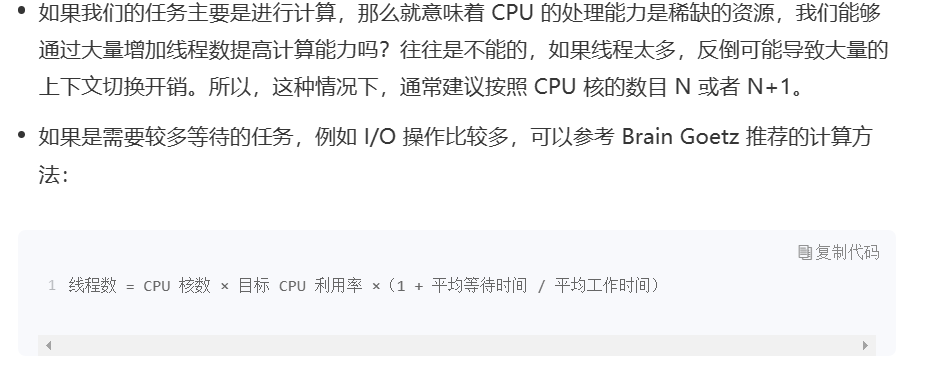


s

## 线程池可能出现的问题



线程池大小



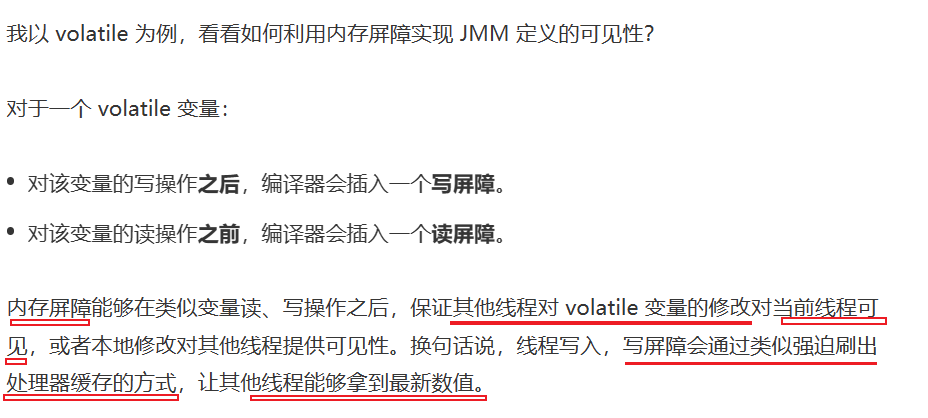
# Java内存模型中的happen-before是什么

Happen-before是 Java 内存模型中 保证多线程操作可见性的机制.

happen-before不仅仅是对执行时间的保证，也包括对内存读、写操作顺序的保证。仅 仅是时钟顺序上的先后，并不能保证线程交互的可见性。

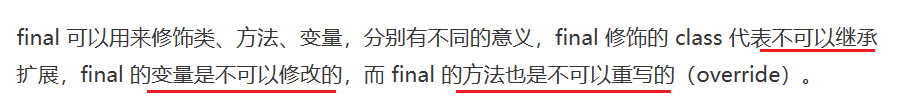
为什么需要 JMM，它试图解决什么问题？

Java内存模型的完善 解决了多线程程序的正确性 与 在不同的处理器架构上表现一致。

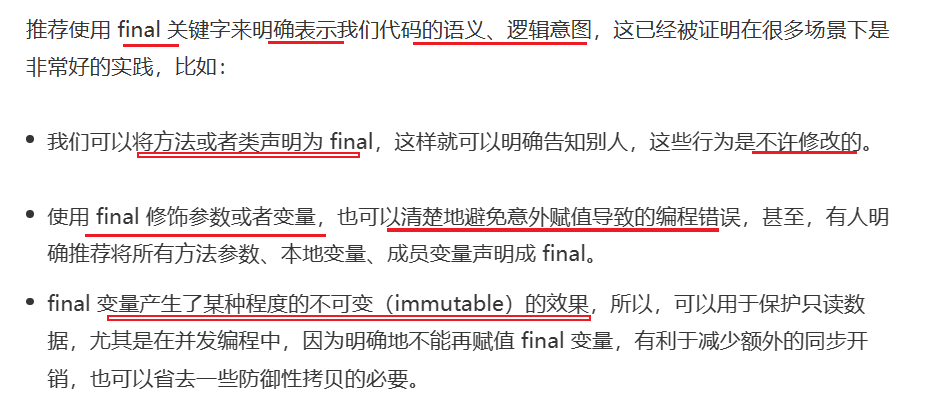


# 相关内容

## final



### final明确语义



### final不是immutable



### 如何定义不可变(immutable)类

