进程与线程：

进程：类似exe文件，运行程序的指令和数据，看做程序的实例，资源分配的最小单位

线程：一个进程分多个线程，一个指令流，顺序执行，线程作为最小的调度单位，负责执行指令，在windows中进程不活动，作为线程的容器。

二者对比：进程相互独立，线程存在进程内，进程拥有共享的资源，如内存，供内部的线程共享使用，进程通信比较复杂。线程通讯相对简单，共享进程内的数据，线程上下文切换（电脑成本较低，暂时不用切换到另一个存储记录相关信息）成本较低

并行和并发：

并发current：操作系统有任务调度器，会分配给不同线程，让其使用，同一时间应对对个任务叫做并发，单核cpu下，线程实际还是串行执行的。

并行parallel：现在笔记本电脑都是多核的，多个指令同时运行。

异步与同步，在单核cpu下多线程反而没有单线程执行效率快，因为存在线程的上下文切换。

IO操作不占用cpu，只是我们一般拷贝文件使用的是阻塞IO，这时相当于线程虽然不用cpu，但需要一直等待IO结束，没能充分利用线程，所以才有后面的非阻塞IO和异步IO优化。

Thread对象创建和启动线程，重写run方法。

Runnable配置Thread

FunctionInterface可以使用lambda进行编程

创建线程的第三种方式：FutureTask类间接实现了Runnable接口，也实现了Future接口，这个接口的get方法是返回任务的执行结果，方便两个线程之间将结果传递，而Runnable不行。

FutureTask配合Callable接口将线程执行结果传递给其他线程

当其他线程要获取FutureTask线程中的返回结果，会进行阻塞，直到FutureTask执行完成获取线程的执行结果。

Ps -ef命令列出正在运行的进程，有进程的Pid，还有可执行文件的位置。

Linux中top命令的使用：动态的方式查看进程的信息，还会查看进程占用cpu的百分比，可以查看更详细的线程信息：top -H -p 进程id，java中查看java中线程的快照jstack

Jsconle远程连接java程序需要程序开通远程连接端口

线程运行原理：

栈：一个线程运行虚拟机就分配栈内存，栈由多个栈帧组成，每个栈内发生方法调用就会产生栈帧内存，同一个线程内只能有一个活动栈帧。

线程上下文切换：线程cpu时间片用完，垃圾回收，有更高优先级的线程需要运行，线程自己调用了sleep、yield、wait、join、park、synchronized、lock等方法。

上下文切换时由操作系统保存当前线程状态。

线程中的常见方法：

Join方法：等待某个线程运行结束，常用于两个线程通信时使用，其中有参数的表示最多等待多少毫秒。

GetState方法：获取线程状态，有六种用枚举表示。

IsInterrupted：判断线程是否被打断，线程运行中会被其他线程打断，里面有一个打断标记，判断有没有被打断，线程自己根据这个标记判断是不是真的要打断自己，不会清除打断标记。

IsAlive方法：线程是否存活

Interrupt方法：打断线程，如果被打断的线程正在sleep、wait、join会导致打断的线程抛出interruptedException，清除打断标记；如果打断的线程正在运行则会设置打断标记，线程不会停下来，需要依赖打断标记进行判断。

CurrentThread方法：获取当前正在运行的线程对象。

Sleep方法：线程放弃cpu使用权休眠。

Yield方法：和sleep相似，但是没有具体时间。

线程中start方法不能被多次调用。

Sleep方法将线程从running状态编程TimedWaiting阻塞状态。

Yield方法会让线程从Running进入Runnable就绪状态，然后调度执行其他线程，具体的实现依赖于操作系统的任务调度器。

Sleep和Yield区别：yield在就绪状态还是有机会获取线程执行机会，而TimedWaiting阻塞状态就不行了。

线程优先级1-10，数字越大优先级越高，不靠谱。

While-true在服务器开发中经常使用，一直运行接收请求进行处理，一般会在开始使用sleep，必须加的，避免whil-true一直占用cpu空转，避免其一直占用cpu的资源。

Join方法：等待线程调用结束。

有时效的等待joing(n)

用到interrupt方法的一个多线程成设计模式：两阶段终止模式，在一个线程T1中优雅的终止线程T2，这里的优雅指的是给T2一个料理后事的机会。

线程stop方法会真正杀死线程，如果线程锁住了共享资源，那么当他被杀后就没有机会释放锁了，其他线程永远无法获取锁。

System.exit比较暴力，停止包括所在线程的进程了。

两阶段终止模式：通过interrupt实现。

打断park线程，LockSupport支持类中的方法，是让当前线程停下来，park线程打断标记为真的时候park不能让线程停止。

Stop、suspend、resume三个方法不建议使用，破坏同步代码块，造成对象锁不能释放，导致线程死锁。

从操作系统层面描述线程的五种状态：初始状态、可运行状态、运行状态、终止状态、阻塞状态。

API层面的六种状态（按照枚举类分的）：NEW—初始状态，刚被创建,BLOCKED—wait方法需要别人唤醒,WAITING—join没有具体的等待时间,TIMED\_WAITING—sleep方法阻塞一段时间,TERMINATED—终止状态,RUNNABLE—含操作系统的可运行、运行、阻塞（操作系统层面的，阻塞的IO操作）三种状态

共享模型之管程：

避免临界区的竟态条件发生，有两种方式的解决方案：

1. 阻塞式的解决方案：synchronized，Lock
2. 非阻塞式的解决方案：原子变量

String类为什么设计成final：避免子类覆盖父类方法，导致覆盖父类行为，导致String变成线程不安全。

Synchronize底层

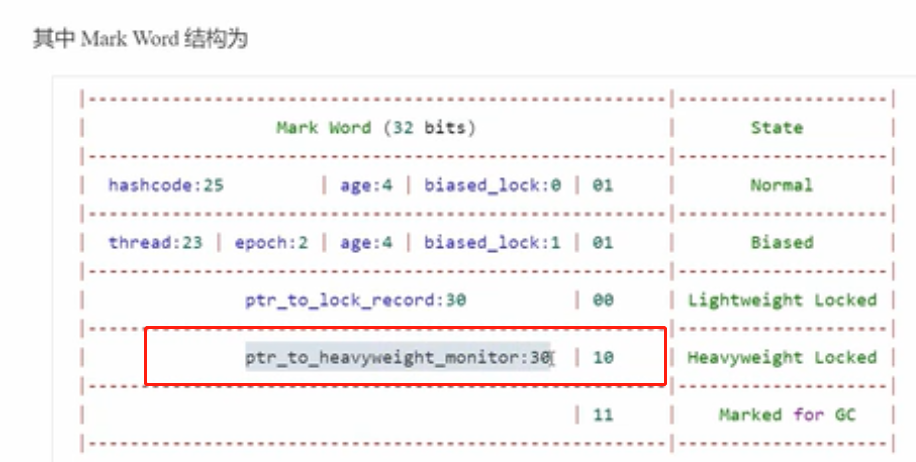
一个对象基本由两部分组成，一个是对象头，另一个是成员变量

32位的虚拟机中对象头是64bit，前32位是Mark Word 后32位Class Word，后四个字节指向这个对象的class文件，表明他的类型。

Mark word中25位hashcode，4位分代年龄，1位偏向锁，2位加锁状态，这个是普通的状态。

Monitor：synchronize的底层原理，可以理解为锁，翻译成管程或监视器，每个java对象都可以关联一个Monitor对象，如果使用synchronized给对象上锁后，该对象头的Mark Word中就被设置为只想Monitor对象的指针。

直观解释：synchronized加锁用的对象obj，将java的obj的对象与操作系统提供的Monitor对象关联，通过MarkWord指针地址指向Monitor对象。

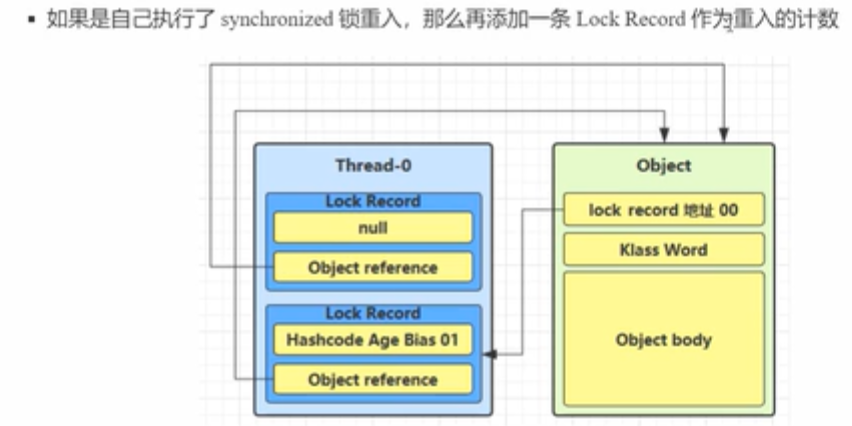


线程获取到锁后，Owner中会记录线程为所有者。当第二个线程再来的时候发现Owner已经有对象了，那么此时进入等待阻塞队列EntryList的Blocked状态。

理解：synchronized每次进入需要获取monitor锁，是重量级的锁，性能不高，java6后进行改进使用了偏向锁、轻量级锁等进行了优化。

轻量级锁的使用场景：一个对象虽有多个线程访问，但多线程访问的时间错开的（没有竞争），那么可以使用轻量级锁来优化，轻量级锁对于开发者是透明的。

此处的详细见课件



锁膨胀：如果在尝试加轻量级锁的过程中，CAS操作无法成功，这时一种情况就是有其他线程为此对象加上了轻量级锁（有竞争），这时需要进行锁膨胀，将轻量级锁变为重量级锁。

自旋优化：重量级锁竞争的时候，还可以使用自旋来进行优化，如果当前线程自旋成功（即这时候持锁线程已经推出了同步块，释放了锁），这时当前线程就可以避免阻塞，避免上下文的切换，自旋需要使用cpu，适合多核cpu比较有意义，自旋失败就进入阻塞entryList中。

偏向锁优化：轻量级锁在没有竞争时（就自己这个线程），每次重入仍然需要执行CAS操作，java6中进一步引入偏向锁进行优化，只有第一次使用CAS将线程ID设置到对象的Mark Word头，之后发现这个线程ID是自己的就表示没有竞争，不用重新CAS，以后只要不发生竞争，这个对象就归线程所有。

如果程序场景中线程比较多，需要进行禁用偏向锁，禁用了偏向锁就使用的是轻量级锁。

所以优先级划分：有偏向锁采用偏向锁，其他对象使用了锁，就会撤销偏向锁，采用轻量级锁，轻量级锁有竞争发生就会锁膨胀为重量级锁。

调用了对象的hashCode会禁用掉对象的偏向锁：因为线程id的存储和hashCode的存储发生冲突，有限hashcode，为什么轻量级锁和重量级锁不会出现这种情况，因为一个hashcode在线程栈帧的锁记录中，一个字monitor中。

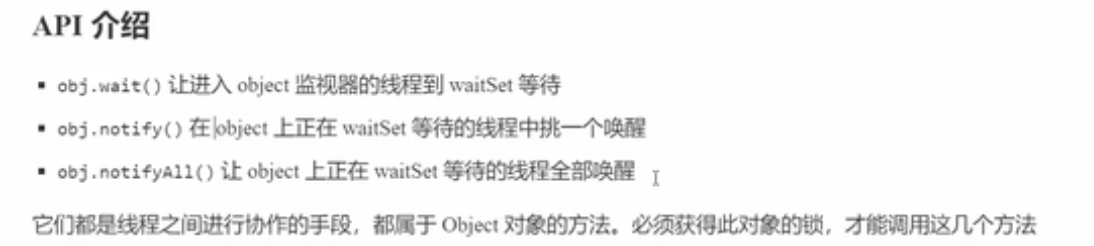
偏向锁和轻量级锁的前提是多个线程错开。

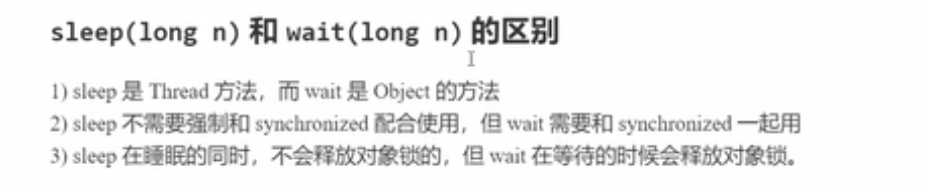
调用wait和notify机制也会撤销偏向锁和轻量级锁。

批量重偏向：如果对象虽然被多个线程访问，但没有竞争，这时偏向了线程T1的对象仍有机会重新偏向T2，重偏向会重置对象的Thread ID，当撤销偏向锁阈值超过20次后，jvm会这样觉得，我是不是偏向错了，于是会在给这些对象加锁时重新偏向至加锁线程。

批量撤销：当撤销的阈值超过40次后，jvm会这样觉得，自己确实偏向错了，根本就不该偏向，于是整个类的所有对象都会变成不可偏向的，新建的对象也是不可偏向的。

锁消除：JIT即时编译期优化字节码，去掉synchronized，可以通过锁消除开关进行优化，默认开关打开。



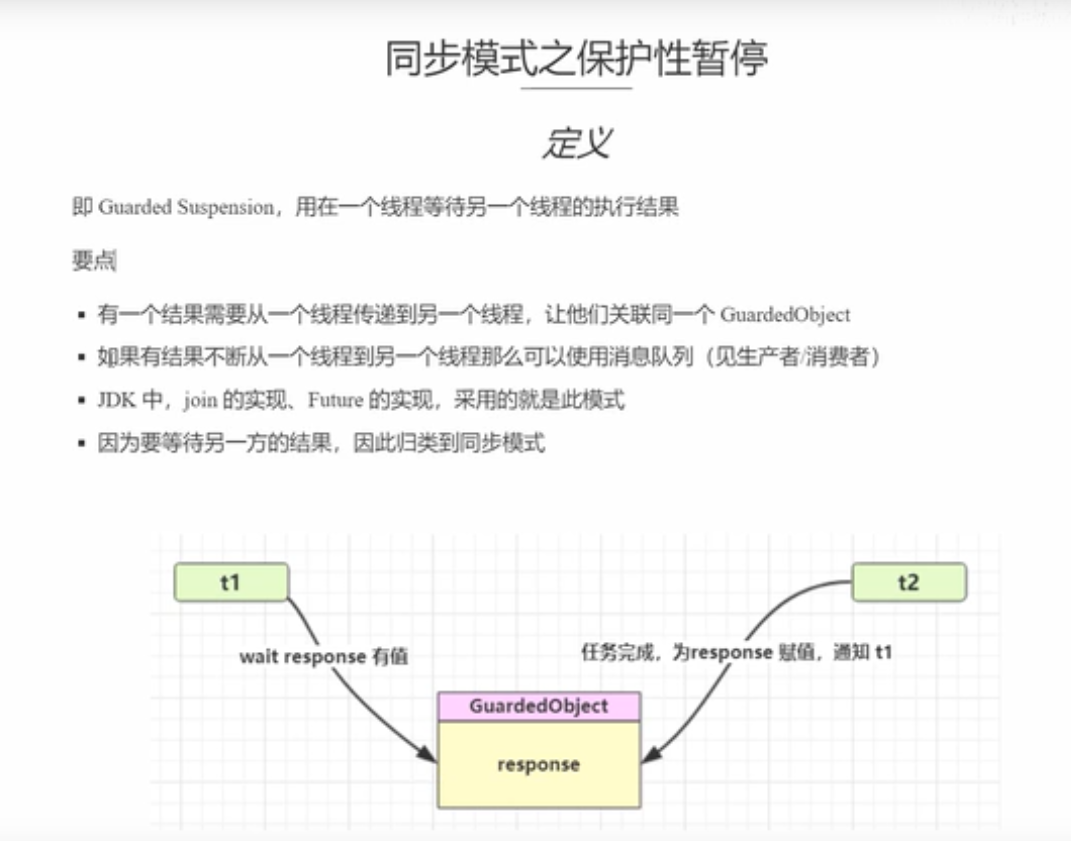


一般锁对象都加上final，保证锁对象不会变成。

Sleep只是放弃了cpu的使用权，没有放弃锁，sleep和wait的线程状态一样都是timed\_waite状态。

同步模式之保护性暂停：

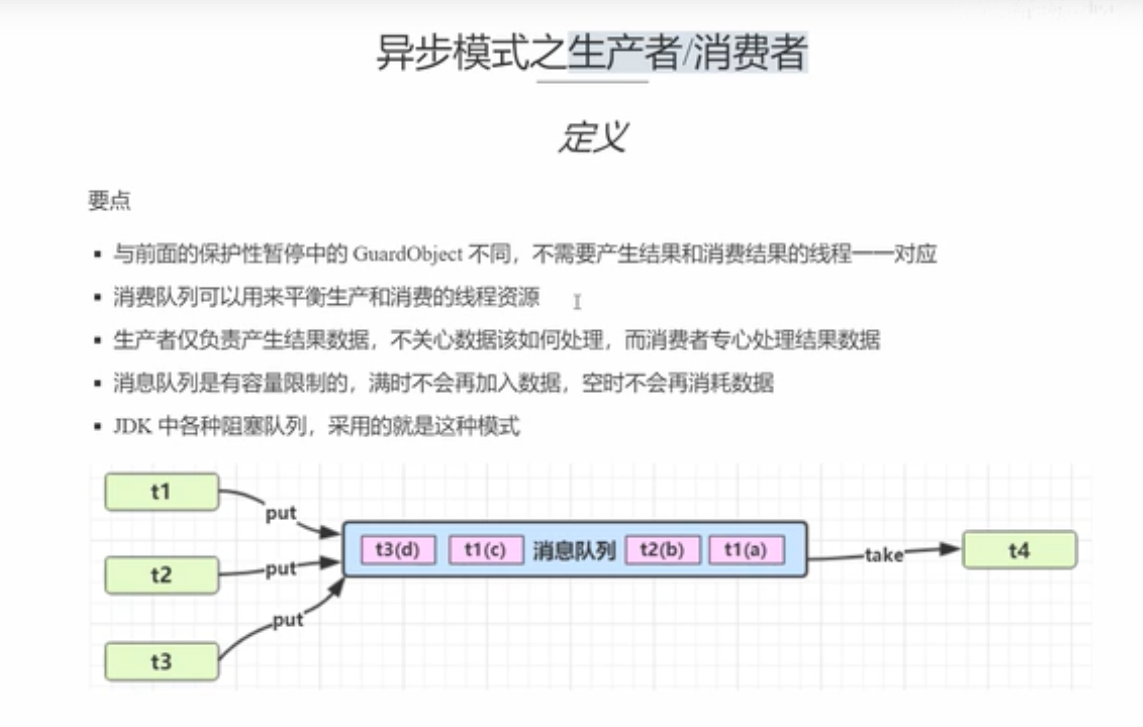
用在一个线程等待另外一个线程的执行结果。



注意虚假唤醒

Join的原理：使用保护性暂停模式等待线程执行完成，join没有参数表示一直等待，RPC框架经常使用保护性暂停模式解耦结果产生者和结果消费者—产生结果线程和使用结果的线程一一对应。

生产者/消费者：

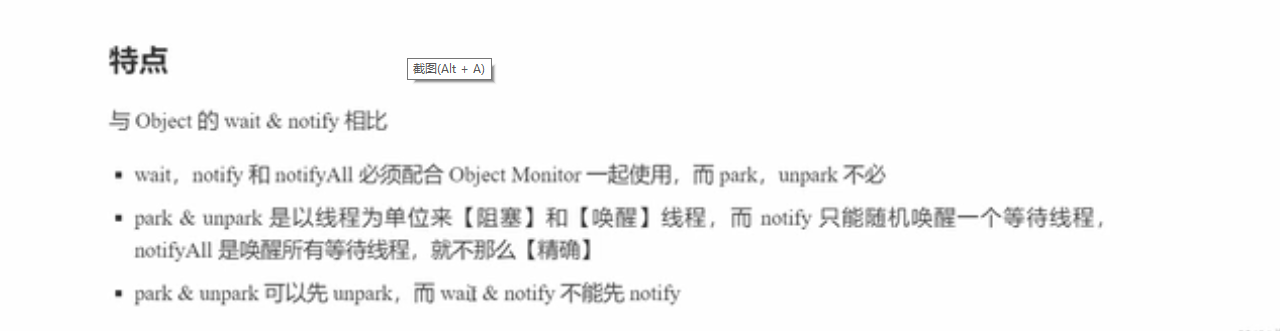


当消息队列满了后，生产者就会被阻塞不再生产消息。

Lambda表达式要求引用的外部局部变量必须是final的。

Park与Unpark方法同wait和notify方法相似，都是工具类LockSupport中的方法，暂停某个线程或回复某个线程，可以根据线程名称恢复线程，使用Park线程的状态就是waite状态的。

Unpark即可以在park之前调用，也可以在之后调用。



P114