操作系统

**进程**：一个有独立功能的程序在一个数据集上一次动态执行的过程

OS**资源分配和调度的基本单位,**分配了内存 I/O CPU

由**程序**(描述功能)、**数据集合**、**PCB**组成(进程描述和控制信息--标志)

每个进程有独立内存，但因为进程间上下文切换耗费大，所以发明了线程

比如手机里打开美团

**进程通信/同步方式**：1.无名**管道**(有亲缘关系的进程间用，数据单向流动)

2.有名管道(没亲缘关系也可以用)

管道是在内核申请一块缓冲区用于读写

3.**共享内存**(映射一段物理内存到不同进程的虚拟内存) – 最快

最快因为**不用切换到内核态**，直接从**内存里读**

**不会占用地址空间**，因为内核为它增加了一段

注意：因为是**临界资源**，所以要用**信号量保证原子性**

4. 消息队列(一个队列，元素是数，进程可以访问这个队列)

5.**信号量**(计数器，作为一种锁的机制管理进程对资源的访问)Linux里 要用自旋转锁

6.**套接字**(用于**不同机器**间进程通信)

**多进程**：操作系统中同时运行的多个程序

**线程**：进程中的一个执行任务

**程序执行**和OS**独立调度和分派的基本单位** 并行 eg 发消息 发语音

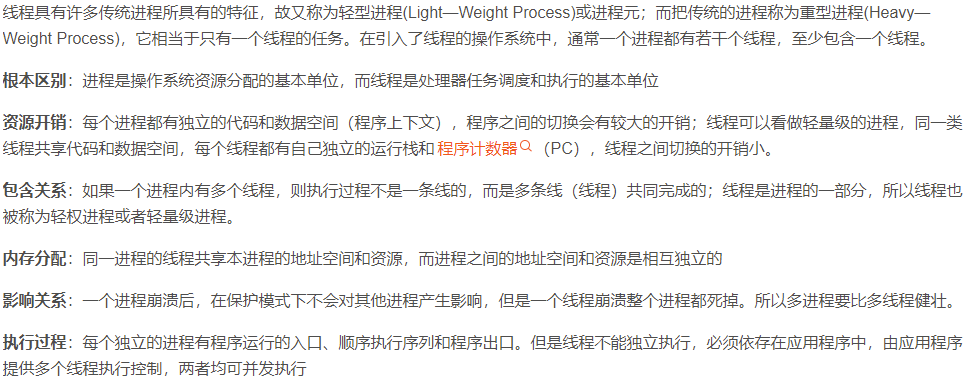
线程间共享堆和方法区，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多

**多线程**：在同一个进程中同时运行的多个任务

**两者上下文切换**：进程上下文切换涉及虚拟地址空间的切换所以比较耗时，而线程是共享所在进程的虚拟地址空间的，只要保存寄存器状态就差不多。

**为什么虚拟地址空间的切换比较耗**时：进程切换涉及到虚拟地址空间切换会导致内存中的页表也需要进行切换，虽然一个进程对应一个页表，但 CPU 中的 TLB 只有一个，页表切换后这个 TLB 就失效了。所以，TLB 在一段时间内肯定是无法被命中的，操作系统就必须去访问内存，那么虚拟地址转换为物理地址就会变慢，程序运行也就变慢

**进程和线程区别**



线程

**协程**：是一种用户态的轻量级线程，调度的话完全由用户控制，创建时不用调用OS功能，多个协程在一个主线程上，是异步机制 为了并发

并发的关键是你有处理多个任务的能力，不一定要同时

并行的关键是你有同时处理多个任务的能力

**差别**：1.**定义**

进程是操作系统资源分配的最小单位，而线程是调度和程序执行的最小单位

2. 包含关系:进程(有多个线程) **>** 线程(进程种代码执行路线) > 协程

3. **共享**:进程相互独立，线程共享内存空间，堆等，但私有栈,寄存器(程序计数器)

4.**切换**:进程切换要保存当前CPU环境并设置新的，线程只是保存和设置寄存器

联系：通常只有一个CPU并只给一个进程，总线程数>CPU数-并行，反之并发

并发是充分利用每一个核

**线程的实现方法**:

1. 线程池
2. 继承Thread类(编写简单，直接用this获取线程，但不能继承其 他类了)
3. 实现Runnable接口(重写run()方法，没返回值，不能抛出异常)
4. callable(重写call()方法，有返回值，可以抛出异常)

3和4适合多个线程资源共享

**线程通信方法**：1.**全局变量**(因为内存共有) **volatile**

2.**消息队列**(因为每个**线程有自己的消息队列**)

3.**事件**

**线程5个状态**：1.NEW新建 2.RUNNABLE(其他线程call了start方法) 3.RUNNING runnable的线程获得时间片 4.BLOCKED 放弃了CPU<- sleep、锁被占用 5.DEAD

**start()启动线程**--处于就绪状态，之后Thread类调用**run()使其运行**，run() 里面有线程内容，run结束线程也终止

线程互斥：同时只能一个线程进入临界区(critical section)对资源进行读写操作

**线程同步**：多线程通过互斥量等机制控制线程之间执行顺序

比如一个线程对临界区写，另一个读，必须先写再读。不然结果不如预期

方法：1.互斥量 只有拥有互斥量对象的线程才能访问公共资源

2.临界区 只允许一个线程对共享资源访问

3.信号量Semaphore 允许多个线程同时访问资源

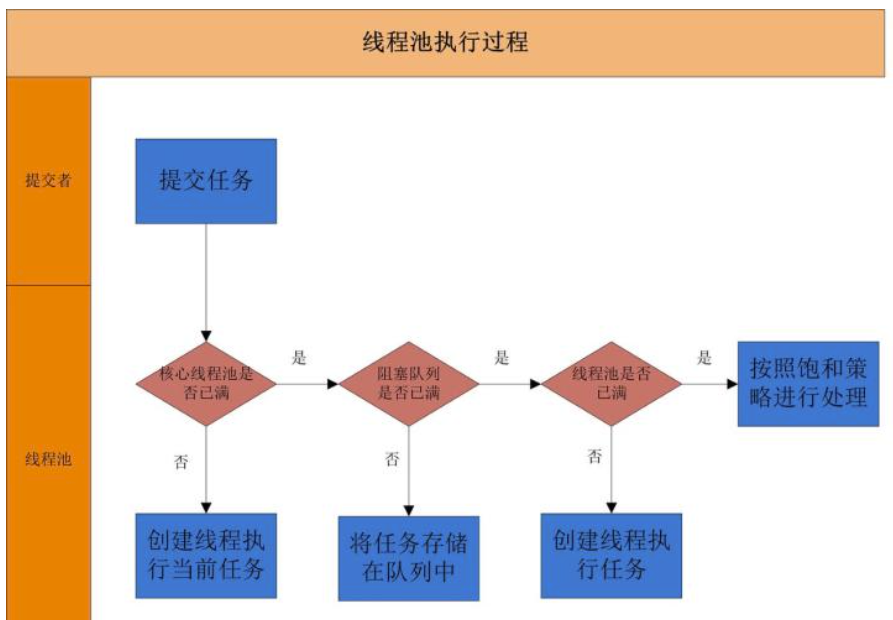
**线程安全**：多线程访问时，用了加锁机制，当一个线程访问某个数据时进行保 护，其他线程不能进行访问直到该线程读取完，不会出现数据不一致或者数据污染

**线程不安全**：不提供数据访问保护，有可能多个线程先后更改数据造成所得 到的数据是脏数据，不符合预期

**线程池**：是一种基于[池化](https://so.csdn.net/so/search?q=%E6%B1%A0%E5%8C%96&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/a1275302036/article/details/_blank)思想管理和使用线程的机制。

**实现**：创建多个可执行的线程放到一个池里，需要的时候不用自己创建直接从池里获取，使用完不销毁线程而是把它放回池里，线程池会在空闲时间来创建和销毁线程，这样服务器在处理用户请求的时候就不会有创建和销毁线程的开销，可以控制最大并发数

**好处**：提高线程的利用率，提高程序的相应速度，便于统一管理线程对象



**流程：1.提交任务：**1.工作线程数小于核心线程数直接创建新的核心工作线程。2.工作线程数大于核心线程数就尝试把任务添加到阻塞队列中去。3.如果能够加入成功，说明队列还没满，那么就做二次校验来保证添加进去的任务能成功被执行。

4.验证当前线程池中的运行状态，如果是非RUNNING状态，就要把任务从阻塞队列中移除拒绝这个任务。5.验证当前线程池中的工作线程的个数，如果是0，就要主动加一个空工作线程来执行刚刚添加到阻塞队列中的任务。6.如果加入失败，说明队列满了，就要创建新的临时工作线程来执行任务。7.如果创建成功，就直接执行这个任务。8.如果创建失败，说明工作线程数已经等于最大线程数了，只能拒绝该任务了。**2.创建工作线程：先判断3种情况**当线程池的状态是SHUTDOWN或者STOP或者线程工厂创建线程失败或者工作线程的数比核心线程数、最大线程数大就不能创建。没这些情况，线程池会通过CAS来自增工作线程的个数，如果自增成功就会创建新的工作线程，也就是Worker对象。然后加锁进行二次验证是否能够创建工作线程，如果最后创建成功，就会启动这个工作线程。**3.启动线程：**通过worker.thread.start()来启动线程，启动完了之后，就会执行Worker对象的run方法，因为Worker实现了Runnable接口，所以本质上Worker也是一个线程。**4.获取任务去执行**：Worker对象中默认绑定了一个任务，如果该任务不为空的话，那么就是直接执行。执行完了之后，就会去阻塞队列中获取任务来执行。获取任务的过程需要考虑当前工作线程的个数：

1.如果工作线程数大于核心线程数，那么就需要通过poll(keepAliveTime, timeUnit)来获取，因为这时需要对闲置线程进行超时回收。

2.如果工作线程数小于等于核心线程数，那么就通过take()来获取。因为这时所有的线程都是核心线程，不需要进行回收，前提是没有设置allowCoreThreadTimeOut（允许核心线程超时回收）为true。

**创建方式：1.通过Executors类创建**：

**Executors.newFixedThreadPool**创建一个固定大小的线程池，可控制并发的线程数，超出的线程会在队列中等待。

**应用场景**：FixedThreadPool用于负载比较重，而且负载比较稳定的场景，比如有套负载比较重的后台系统，每分钟要执行几百个复杂的SQL, 因为负载稳定，一般来说，不会出现突然涌入大量请求，导致100个线程处理不过来，然后就直接无限制的排队，然后内存溢出

**Executors.newCachedThreadPool**创建一个可缓存的线程池，若线程数超过处理所需，缓存一段时间后会回收，若线程数不够，则新建线程。

**应用场景：**用在负载很稳定的场景的话就浪费了。因为每天大部分时候可能就是负载很低的，用少量的线程就可以满足低负载，不会给系统太大压力；但每天如果有少量的高峰期，比如说中午或者是晚上，可能需要一下子搞几百个线程出来，那CachedThreadPool就可以满足这个场景，然后高峰期应付过去之后，线程如果处于空闲状态超过1分钟，就会自动被回收，就避免了给系统比较大的负载

**Executors.newSingleThreadExecutor**创建单个线程数的线程池，它可以保证先进先出的执行顺序

**Executors.newScheduledThreadPoo**l创建一个可以执行延迟任务的线程池

**2.通过ThreadPoolExecutor类创建**：最原始的创建线程池的方式，有7 个参数

**参数**：

**1.** **corePoolSize** 核心线程数，也就是可闲置的线程数量。空闲时候不会被销毁(不超过最大线程数值时，线程池中最多有核心线程数个线程工作，再来任务就不创建线程而是去阻塞队列里等待)

**2.maximumPoolSize** 指的是线程池的最大大小(线程池中最大有corePoolSize 个线程可运行)

**3.keepAliveTime** 指的是非核心线程可以保留的最长的空闲时间(当一个线程不工作时，过keepAliveTime 长时间将停止该线程)

**4.unit** 是一个枚举，表示 keepAliveTime 的单位(有NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS，7个可选值) 纳秒，微秒，毫秒，秒，分，小时，天

**5.workQueue** 阻塞(等待)队列(存需要被线程池执行的线程队列)有ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue等7种阻塞队列

**有限阻塞队列vs无限的阻塞队列**

**有限阻塞队列(ArrayBlockingQueue)**可能不能很好得满足性能，因为如果所有任务都没法执行完然后线程数又达到上限的话，新来的任务就会被用拒绝策略来处理了，这就需要调节线程数和queue的大小

**无限阻塞队列(LinkedBlockingQueue)**可以一直接受任务,但可能会耗尽系统资源

maximumPoolSize - corePoolSize = 有限阻塞队列放不下后需要创建的临时线程(有存活时间)

**6.threadFactory**创建线程的工厂，主要用来创建线程，默认为正常优先级、非守护线程。

**7.handler** 拒绝策略((线程个数达到max(满了)，对任务拒绝，此时队列也满了))

**拒绝策略：**

1.abortpolicy(线程池默认的拒绝策略，在任务不能再提交的时候，不执行新任务，直接抛出异常，提示线程池已满，及时反馈程序运行状态。如果是比较关键的业务，推荐用这个拒绝策略，这样在系统不能承载更大的并发量的时候，能够及时的通过异常发现)

2.discardPolicy(丢弃任务，但是不抛出异常。如果线程队列满了，那么后面提交的任务都会被丢弃，而且是静默丢弃)

3.discardOldestPolicy(丢弃队列最前面的任务，然后重新提交被拒绝的任务)

4.CallerRunsPolicy(由调用线程处理这个任务,如果任务被拒绝了，就让调用线程（提交任务的线程）直接执行此任务)

**保证核心线程不被回收**：1.如果工作线程数小于当前的核心线程数，就用take()方法取任务，也就是所有工作线程都是核心线程不会被超时回收。

2.如果工作线程数大于核心线程数，就用poll(keepAliveTime, timeUnit)方法取任务，一旦超时就回收，所以并没有绝对的核心线程，只要这个线程没有在存活时间内取到任务去执行就会被回收。

**如何合理配置线程池size大小**：根据任务类型(CPU密集型、IO密集型、混合型)配置

**CPU密集型**：任务会消耗大量的CPU资源，一般**线程池大小=CPU核数+1**。因为线程过多时线程切换会带来额外开销。

**IO密集型**：因为要等待IO操作而且IO操作不占⽤CPU，线程不是一直在执行任务，所以不要让CPU闲下来配置尽可能多的线程，比如配置**两倍CPU个数+1**。

但**最佳线程数目= (线程等待时间与线程CPU时间之比 + 1) \* CPU核数**

**混合型**：考虑拆成一个CPU密集型任务和一个IO密集型任务。如果这两个任务处理时间差不多，可以拆分，分解后执行的吞吐率要高于串行执行的吞吐率。如果处理时间差很多，处理时间取决于时间长的，还要付出合并结果的开销，就不要拆分了。

**多线程**：为了**同步完成多项任务**，**提高资源使用率**来提高系统效率(不是运行效率)，充分利用CPU，更好地利用系统资源。

如果OS支持多个处理器，每个线程分配给一个处理器。

它最有价值的地方是我们不用知道使用了几个处理器。但可能会有**共享资源的问题**，所以要用**锁**

实例：生产者消费者问题，3个地方卖20张票，用同步锁保证不会卖出同一张票

**单核CPU机器使用多线程技术为了解决的问题：**

单核机器运用多线程技术**为了实现并发**

**缺点：**单核CPU运行的多线程程序, 同一时间只能一个线程在跑, 系统给每个线程分配时间片来执行, 每个时间片大概10ms左右, 看起来像是同时跑, 但实际上是每个线程跑一点点就换到其它线程继续跑，效率不会有提高的。切换线程反倒会增加开销。

**好处：**多线程的用处在于，做某个耗时的操作时，需要等待返回结果，这时用多线程可以提高程序并发程度。如果一个不需要任何等待并且顺序执行能够完成的任务，用多线程就是浪费。因为一个程序不总是CPU密集型的，肯定穿插着很多IO调用。而IO的一个特性就是阻塞等待。做某个耗时的操作时，需要等待返回结果，这个阻塞等待的时间消耗往往是远大于线程切换消耗的时间的，比如要访问10个URL获取接口内容，假如一次http访问平均阻塞时间大概是1s，那么肯定多线程更快，所以**单核CPU情况下，多线程在CPU密集型的程序下不能提高性能甚至更浪费时间但在IO密集型的程序下可以提高性能和程序并发程度**

**并行vs并发**

**并发**：当有多个线程在操作的时候,如果系统**只有一个CPU**,只能一次运行一个线程，它就会把CPU运行时间分成几个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](https://so.csdn.net/so/search?q=%E6%8C%82%E8%B5%B7&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_47513022/article/details/_blank)状。两个或多个事件在**同一时间间隔内发生**，把任务在不同的时间点交给处理器处理，是在一台处理器上“同时”处理多个任务。在同一时间点，任务不会同时运行

**并行**：**多核情况下**，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。也就是两个或者多个事件在**同一时刻发生**。把每一个任务分配给每一个处理器独立完成，是在多台处理器上同时处理多个任务。在同一时间点，任务一定是同时运行

**信号量**

**信号量**：信号量是一个特殊的变量，它的本质是计数器，里面记录了临界资源的数目，有多少数目，信号量的值就为多少，除了信号量的值还有等待这个信号量的进程队列。进程的访问都是**原子**操作加减（pv操作，p(wait)：占用资源，v(release)：释放资源）。它的作用就是，调协进程对共享资源的访问，让一个临界区同一时间只有一个进程在访问它。



n>0：当前有可用资源，可用资源数量为n。n=0：资源都被占用，可用资源数量为0。n<0：资源都被占用，并且还有n个进程正在排队  
信号量拖着的那个队列就是用来放正在排队想要使用这一资源的进程

**具体加减**：信号量是一个非负整数（车位数），所有通过它的线程/进程（车辆）都会将该整数减一（通过它当然是为了使用资源），当该整数值为零时，所有试图通过它的线程都将处于等待状态。在信号量上有两种操作： Wait（申请资源）和 Release（释放）。当一个线程调用Wait操作时，它要么得到资源然后将信号量减一，要么一直等下去（指放入阻塞队列），直到信号量大于等于一时。Release（释放）实际上是在信号量上执行加操作，也就是释放了由信号量守护的资源。

**其他**：semWait(S)：请求分配一个资源。semSignal(S)：释放一个资源。

semWait、semSignal操作必须成对出现。

用于**互斥**时，位于同一进程内（初始值为1）；

用于**同步**时，交错出现于两个合作进程内。

（且在前事件后加semSignal，在后事件前加semWait），比如先刷牙再吃饭，那刷牙这个事件后加semSignal，在吃饭这个事件前加semWait

多个semWait操作的次序不能颠倒，否则可能导致死锁。

多个semSignal操作的次序可任意。

信号：是由用户、系统或者进程发送给目标进程的信息，以通知目标进程某个状态的改变或系统异常。

**死锁**：一组进程里每个进程都**有不释放的资源**，但又互相**申请**被其他进程**占的资源**而处于一种**永久等待状态**

**死锁4个必要条件**：

1. **互斥条件**：资源由一个进程使用，不能共享
2. **请求与保持条件**：已经得到资源申请新资源

预防：可以一次性将某个进程需要的所有资源都分配到位，这样就破坏了“请求” 条件，但凡有一种资源请求无法得到满足，那么该进程都需要等待，这样就破坏了 “保持”条件

1. **非剥夺条件**：已经分配的资源不能从相应进程中剥夺

预防：将资源属性设置为可剥夺，高优先级进程在申请低优先级进程占有资源时可 以强制让低优先级进程释放，这种方法适用于任意两个进程的优先级都不同的情况

1. **循环等待条件**：进程组成环路，每个进程都在**等相邻进程的资源**

预防:将所有类型的资源都进行编号,进程在请求资源时必须按照指定编号的顺序请求

最根本原因：没释放一个锁要去获得另一个锁

**避免死锁---银行家算法**：允许进程**动态地申请资源**，系统在每次实施资源分配之前，**先计算**资源分配的**安全性**，若这次资源分配安全，就把资源分配给进程（即资源分配后，系统能**按某种顺序来**为每个进程**分配资源**，使每个进程都可以顺利完成），否则**不分配**资源，让进程**等待**

**鸵鸟算法**：假装不知道死锁的存在，死锁发生之后对用户的影响很小，或者死锁发生的概率很小，那么就可以考虑忽略死锁

**死锁发生后的解除方法**:

1.杀掉造成死锁的进程，释放它所占有的系统资源。

2.从一个或多个进程中抢占资源，分配给死锁进程，以让其需求得到满足从而释放占有资源。

**sleep**(): **让出CPU**暂停**执行**,**不释放锁**，**阻塞**线程 不考虑优先级 Tread的静态方法

**wait():** 进入**等待池(变成blocked状态)**，**释放锁**，只有别的线程调用notify才进入锁定池准备，Object的方法，任何对象实例都能调用

**yield**(): **让出CPU**暂停**执行 不释放锁 不阻塞线程** 回到**就绪**状态 Tread的静态方法

**join()**: **等异步线程执行**完才能执行

**block()**: 线程正在等待获取锁synchronized会导致线程进入blocked状态

**block() vs wait()**：Blocked是指线程正在等待获取锁，waiting是指线程正在等别的线程调用notify，之后线程可能会进入runnable状态，也可能再次获得锁变成Blocked状态

阻塞状态是线程因为某种原因放弃CPU使用权，暂停运行。直到线程进入就绪状态，才有机会转到运行状态。

**阻塞**的情况分三种：

**等待阻塞**：运行的线程执行wait()方法，JVM会把该线程放入等待池中。

**同步阻塞**：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁定池

**其他阻塞**：运行的线程执行sleep()或join()方法，或者发出了I/O请求，JVM设置成阻塞。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态

**Shell** 是一个应用程序，它连接了用户和 Linux 内核，让用户能够更加高效、安全、低成本地使用 Linux 内核

gdb：UNIX和UNIX-like下的调试工具，用来运行程序，停在断点检查程序

**缓存cache和缓冲buffer**：cache是放被磁盘读出的 buffer是放写入磁盘的东西

cache提高**CPU和内存**之间数据交换速度，把数据放在几级cache里，减少内存访问

buffer提高**硬盘和内存**之间数据交换速度，把写操作集中进行来提高性能

**内核态 vs 用户态**

**两种CPU状态/OS运行级别：用户态**(运行用户程序) **内核态**(运行OS程序，操作硬件)

**区别**：1.**用户态**时程序运行在用户地址空间里，不能直接访问内存空间和对象，而且占有的处理器可以被抢占，被执行的代码要受到 CPU 的很多检查 比如：进程只能访问映射他地址空间的页表项里规定的在用户态下可访问页面的虚拟地址

**内核态**时候程序运行在内核地址空间中，能访问所有的内存空间和对象，而且占有的处理器不能被抢占。 CPU 可以执行任何指令。运行的代码也不受限制，可以自由地访问任何有效地址，也可以直接访问端口。

2.**指令上**：**操作系统可以使用特权指令**比如启动I/O ,内存清零,设置时钟但用户程序不能使用。**用户程序可以使用非特权指令**，比如控制转移 算数运算 取数指令 访管指令（使用户程序从用户态陷入内核态）

3.**特权级别**：有R0、R1、R2和R3(最低)，**R0=内核态，R3=用户态**

不同级别能够运行不同的指令集合

**Why/什么时候切换**：用户态的程序需要操作系统帮助完成一些它没有权力完成的工作（比如操作硬件：应用程序需要访问磁盘，读取网卡的数据、新建一个线程都需要通过系统调用接口来从用户态切换到内存态）

**切换：用户态--->内核态**：通过1.**设备中断**(设备向CPU发出中断信号)比如硬盘读写完成，东西保存在bios **2.异常**(发生异常会切换当前进程到处理此异常的内核)，**3.陷入机制**（访管指令）

**内核态--->用户态**：设置程序状态字PSW

(通常靠库函数)系统调用(不是进程切换) fork() –每个进程有两个栈 用户和内核栈。调用返回时再切换到用户态，如果队列里有更高优先级的进程就会进程切换

**切换例子(线程创建)：**

比如 Java 中需要新建一个线程，new Thread( Runnable …) 之后调用 start() 方法时, 看Hotspot Linux 的JVM 源码实现，最终是调pthread\_create 系统方法来创建的线程，这里会从用户态切换到内核态完成系统资源的分配，线程的创建

**逻辑地址** -> 线性地址(虚拟) -> 通过**MMU** 转化为物理地址

Linux里**逻辑地址=线性地址** 因为**线性地址都是从0开始**

线性地址转物理地址：从进程中得到pgdir地址->用线性地址前10位找**索引**，中间10位找page table里**page位置** ->加上后12位得到**物理页**位置

**Linux 3种内存模型**：

1. FLAT：访问**物理内存**时候，地址空间连续，不空洞，会浪费

2.Discontinuous： 地址空间不连续，有空洞

3.Sparse：连续的地址空间被**分段**，每段都支持**热插拔**

要用Wait/waitpid:取得子进程运行状态，因为父进程和子进程是个**异步**过程(父进程不知道子进程什么时候结束)

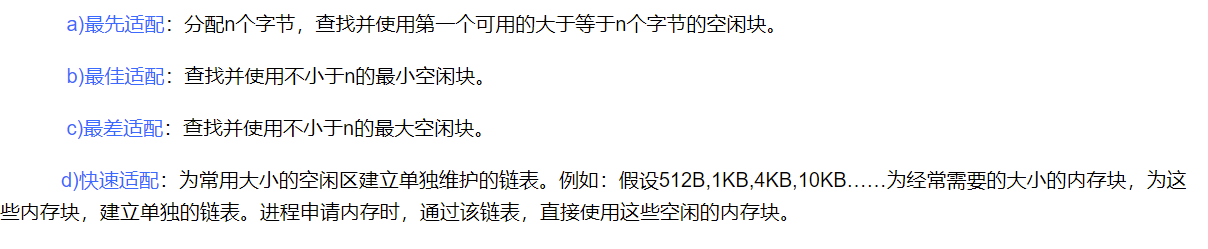
**孤儿进程**：父进程退出，它的子进程还在运行->由init收集它状态 没有危害

**僵尸进程**：子进程退出，但父进程没用wait获得子进程状态，子进程的进程描述符仍在系统中 ->多了导致没用可用进程号->不能产生新进程

静态库(扩展名.a/.lib)**compile时直接整合到程序**，优点：compile成功的文件可以**独立运行**，但函数库升级要重新compile

动态库(.so/.dll) 文件需要用库函数才读取库函数(可执行文件无法独立运行)，优点：节省内存并**减少页面交换**，实现进程资源共享

**OS内存管理**





**物理内存**：真实的内存，CPU的地址线可以直接进行寻址的内存空间大小

**虚拟内存**：[计算机系统](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%B3%BB%E7%BB%9F/7210959" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)[内存管理](https://baike.baidu.com/item/%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86/5633616" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)的一种技术。它使得[应用程序](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F/5985445" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)认为它拥有连续的可用的[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98/103614" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)（一个连续完整的[地址空间](https://baike.baidu.com/item/%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4/1423980" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)），但实际上是被分隔成多个[物理内存](https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%86%85%E5%AD%98/2502263" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)碎片，还有部分暂时存储在外部[磁盘存储器](https://baike.baidu.com/item/%E7%A3%81%E7%9B%98%E5%AD%98%E5%82%A8%E5%99%A8/2386684" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)上，在需要时进行[数据交换](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E4%BA%A4%E6%8D%A2/1586256" \t "https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98/_blank)。

**组成：**虚拟地址分为页号和页内偏移。页号作为页表的索引，页表包含物理页每页所在物理内存的基地址，这个基地址与页内偏移的组合就形成了物理内存地址

**为什么要虚拟内存：**实际的应用中，很多的应用程序都比较大，计算机实际所配置的内存空间无法满足。所以出现了**虚拟内存**

**物理内存和虚拟内存的关系例子**：对32位处理器，[虚拟内存](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/ChaseHeart/article/details/_blank)空间为4G，每个进程都认为自己拥有4G的空间；实际上，在虚拟内存对应的物理内存上，可能只对应的一点点的物理内存。进程得到的这4G虚拟内存是一个连续的地址空间（这也只是进程认为），而实际上，它通常是被**分隔成多个物理内存碎片**，还有一部分存储在外部磁盘存储器上，在需要时进行数据交换。

**虚拟内存 -> 物理内存**：由于存在两个[内存](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/ChaseHeart/article/details/_blank)地址，因此一个应用程序从编写到被执行，需要进行**两次映射**。第一次是映射到虚拟内存空间，第二次时映射到物理内存空间。在计算机系统中，这两次映射的工作是**由硬件和软件共同来完成**的。承担这个任务的**硬件部分叫做存储管理单元MMU**，软件部分就是**操作系统的内存管理模块**了。

**虚拟内存 -> 物理内存：**如果处理器启用了MMU，CPU执行单元发出的内存地址将被MMU截获，从CPU到MMU的地址称为**虚拟地址,** 之后MMU将这个地址翻译成另一个地址发到CPU芯片的外部地址引脚上，也就是**将虚拟地址映射成物理地址**

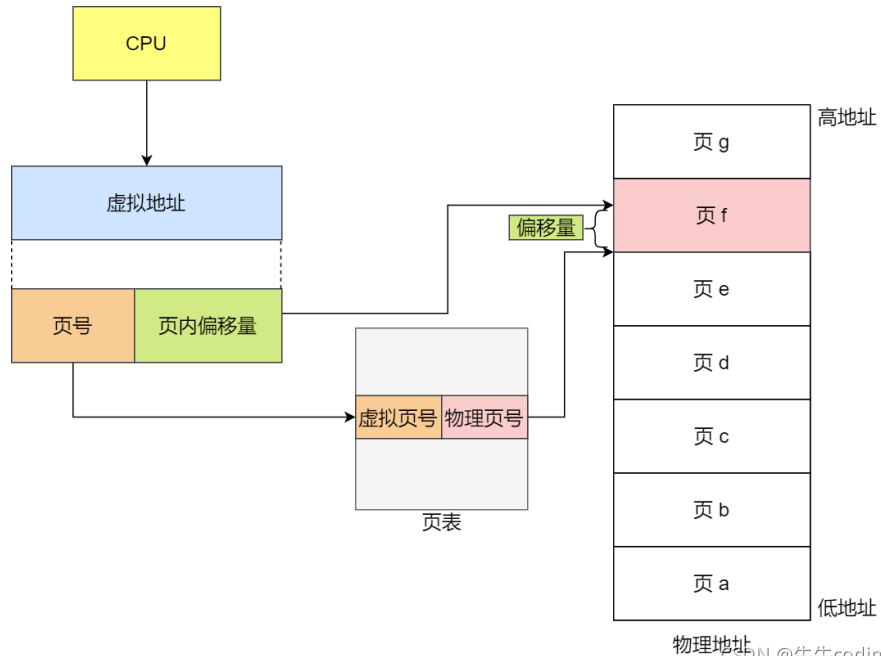
**虚拟地址和物理地址的关系**

**分页：**把整个虚拟和物理内存切成一段段固定大小的空间，连续且尺寸固定的内存空间叫页，Linux下每一页大小4KB。

**页式存储**：将程序的逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(page frame)。程序加载的时候可以把任意一页放人内存中任意一个页框，这些页框不用连续来实现了离散分配。这个方法需要CPU的硬件支持来实现逻辑地址和物理地址之间的映射。在页式存储里地址结构由两部构成，前一部分是页号，后一部分为页内地址w（位移量）

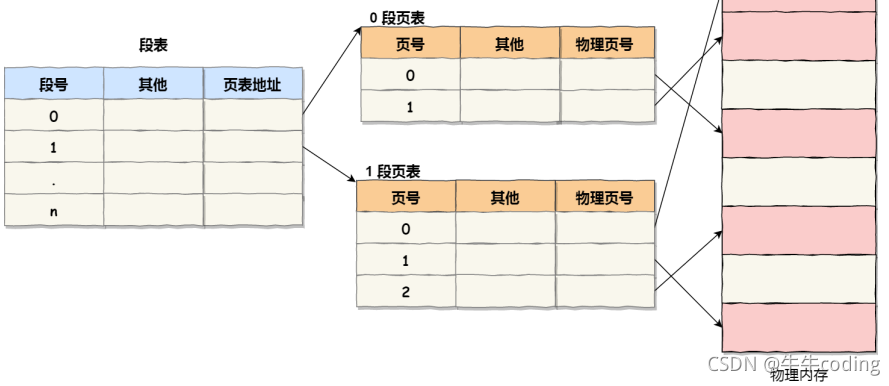
**虚拟内存和物理内存之间通过页表来映射**；虚拟地址分为：页号和页内偏移

页号作为页表的索引，页表包含物理页每页所在物理内存的基地址，这个基地址与页内偏移的组合就形成了物理内存地址



**简单分页缺点**：空间上的缺陷：32位，单进程一个页4KB，虚拟内存4GB，一个页表项4字节，结果是4GB空间映射需要4\*2^20 = 4MB存储页表。多进程100，需要400MB

**解决方法：采用多级页表或段页式存储**

****

地址结构就由**段号、段内页号和页内位移**三部分组成。

第⼀次访问段表，得到页表起始地址；第⼆次访问页表，得到物理页号；

第三次将物理页号与页内位移组合，得到物理地址。

**页表项的结构**：页地址、页偏移、各种标记位标识CPU特权级(靠中断来转换)、读写权限、是否在内存

**磁盘虚拟内存置换算法**

**1.最佳置换算法(OPT)：**置换时淘汰“未来不再使用的”或“在离当前最远位置上出现的”页面。**特点：**性能最好，但无法实现

**2.先进先出置换算法(FIFO)**：置换时淘汰最先进入内存的页面，即选择驻留在内存时间最长的页面被置换。**特点：**简单，但性能差

**3.最近最久未用置换算法(LRU)**：置换时淘汰最近一段时间最久没有使用的页面，也就是选择上次使用距当前最远的页面淘汰。**特点：**性能好，是最接近OPT性能的，但是需要专门的硬件支持，开销大。

**4.时钟算法(Clock)**：也称最近未使用算法(NRU, Not Recently Used)，它是 LRU 和 FIFO 的折衷。是一种性能和开销均平衡的算法

**算法思想**：为每个页面设置一个**访问位**，再将内存中的页面都通过指针连成一个循环队列。当某个页被访问时，其访问位置1.当需要淘汰一个页面时，只需检查页的访问位。是**0就换出**；是**1就把访问位改为0**，继续检查**下一个**页面

若**第一轮扫描**中所有的页面**都是1**，则将这些页面的访问位**都置为0后**，再进行**第二轮扫描**（第二轮扫描中一定会有访问位为0的页面，因此简单的CLOCK算法选择一个淘汰页面最多会经过两轮扫描）。

**缺页中断：**要访问的页不在主存，需要将其调⼊主存后再进⾏访问

步骤：1、保护CPU现场2、分析中断原因3、转入缺页中断处理程序进行处理(根据虚拟地址找磁盘上的位置，把他复制到干净的页面上)

1. 恢复CPU现场，继续执行

**中断**：1.外中断：外设请求或人为干预等当前执行指令之外的因素所引起的中断

2.内中断(也叫异常)：CPU执行当前指令时所发生的中断，分为1.自愿中断。比 如当前执行的是trap指令(陷阱)。2.强迫中断：当前指令在执行时发生的错 误所引起的中断，例如：地址越界，算术溢出，虚存系统的缺页等

3.软中断：是可被调用执行的程序，大部分都是人为主动引起的，不过一些外 中断发生后也会自动的向系统再发出一个软中断，比如当打印机端口产生 一个硬件中断时，会通知和硬件相关的硬中断，硬中断就会产生一个软中 断并送到操作系统内核里，这样内核就会根据这个软中断唤醒睡眠在打印 机任务队列中的处理进程。

4.硬中断：由与系统相连的外设(比如网卡 硬盘 键盘等)自动产生的. 每个设备 或设备集都有他自己的IRQ(中断请求)。

**网卡到内存的步骤**：1、网卡驱动在内存中分配一片缓冲区用来接受数据包，叫做sk\_buffer；2、将上述缓存区的大小和地址（即接受描述符），存入rx ring buffer

中；3、驱动通知网卡有一个新的描述符；4、网卡从rx ring buffer中取出描述符。从而获取sk\_buffer大小和地址；5、网卡接收到新的数据包；6、网卡将新的数据包通过DMA直接写入sk\_buffer中。

**多核CPU**：有自己的L1 Cache L2 Cache，共用L3 Cache和主内存

**解决缓存一致性问题**：需要满足写传播，也就是在一个CPU核心写入的内容，需要传播到其他CPU核心里面和保障事务的串行化，这样才能保障数据和在不同的核心上运行的程序结果是一致的。这个特性在CPU 的缓存层面和数据库层面都重要。

**所以用基于总线嗅探机制和写失效的缓存一致性协议MESI。**整个MESI的状态变更，则是根据来自自己的CPU核心的请求，以及来自其他CPU核心通过总线传输过来的操作信号和地址信息，进行状态流转的一个有限状态机。

写失效协议的好处是不需要在总线上传输数据内容，而只要传输操作信号和地址信号就好了，不会那么占总线带宽

**IO多路复用**

**IO多路复用**(同时处理多路I/O)：

多路是指⽹络连接，复⽤指的是同⼀个线程。是⼀种同步IO模型，实现⼀个线程可以监视多个⽂件描述符。调用函数让**Kernel监视**这些描述符，其中有描述符就绪可以I/O读写操作再返回，返回后我们就知道哪些文件描述符可以来I/O操作。再通知应⽤程序进⾏相应的读写操作。没有⽂件描述符就绪就会阻塞应⽤程序，交出CPU。

**文件描述符(file descriptor)**：是⼀个⽤于表述指向⽂件的引⽤的抽象化概念。是内存为了管理被打开的文件创建的索引(非负整数)---0开始

指向内核为每⼀个进程所维护的该进程打开⽂件的记录表。当程序打开⼀个现有⽂件或者创建⼀个新⽂件时，内核向进程返回⼀个⽂件描述符。

**3种Linux阻塞时(同步)I/O机制：**

**1.select**：只知道有几个I/O事件发生了，但并不知道具体是哪几个socket连接有I/O事件，还需要轮询去查找，时间复杂度为O(n)，处理的请求数越多，所消耗的时间越长。

**流程**：从用户空间拷贝fd\_set（注册的事件集合）到内核空间。遍历所有fd文件，并将当前进程挂到每个fd的等待队列中，当某个fd文件设备收到消息后，会唤醒设备等待队列上睡眠的进程，那么当前进程就会被唤醒。如果遍历完所有的fd没有I/O事件，那么当前进程进入睡眠，当有某个fd文件有I/O事件或当前进程睡眠超时后，当前进程重新唤醒再次遍历所有fd文件

**缺点：1.**单个进程所打开的FD是有限制的，通过 FD\_SETSIZE 设置，默认1024

2.每次调用 select，都需要把 fd 集合从用户态拷贝到内核态，这个开销在 fd 很多时会很大。 3.每次调用select都需要将进程加入到所有监视socket的等待队列，每次唤醒都需要从每个队列中移除。4.select函数在每次调用之前都要对参数进行重新设定，这样做比较麻烦，而且会降低性能。5.进程被唤醒后，程序并不知道哪些socket收到数据，还需要遍历一次

线程**不安全**

**2.poll**：和select没本质区别只是文件描述符能超过1024个，没有最大连接数的限制，因为它是基于链表来存储的。线程**不安全**

**3.epoll**(Linux独有)：采用的是事件驱动机制，每个fd上有注册有回调函数，当网卡接收到数据时会回调该函数，同时将该fd的引用放入rdlist就绪列表中。

当调用epoll\_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。

**流程：**调用epoll\_create()创建一个ep对象，即红黑树的根节点，返回一个文件句柄。再调用epoll\_ctl()向这个ep对象（红黑树）中添加、删除、修改感兴趣的事件。最后调用epoll\_wait()等待，当有事件发生时网卡驱动会调用fd上注册的函数并将该fd添加到rdlist中，解除阻塞

**精简介绍：只操作有变化的描述符**，和Kernel**共享**了些**内存**来放已经可读可写的文件描述符，**减少了拷贝到Kenel开销。**用**事件驱动来解决要遍历描述符才知道哪个可读可写**：进程只要等待在epoll上，epoll代替进程去各个文件描述符上等待，哪个描述符可读可写就通知epoll,epoll记录下来然后唤醒进程

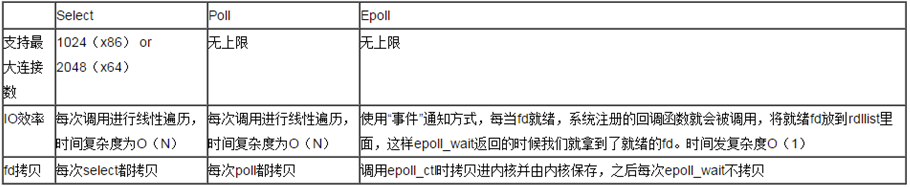
**epoll比select/poll好的地方：**

1.epoll支持的最大文件描述符上限是整个系统最大可打开的文件数目, 1G内存理论上最大创建10万个文件描述符

2.每个文件描述符上都有一个callback函数，当socket有事件发生时会回调这个函数将该fd的引用添加到就绪列表中，select和poll并不会明确指出是哪些文件描述符就绪，而epoll会。造成的区别就是，系统调用返回后，调用select和poll的程序需要遍历监听的整个文件描述符找到是谁处于就绪，而epoll则直接处理

3.select、poll采用轮询的方式来检查文件描述符是否处于就绪态，而epoll采用回调机制。造成的结果就是，随着fd的增加，select和poll的效率会线性降低，而epoll不会受到太大影响，除非活跃的socket很多

**三者比较(select, poll, epoll)**



**多路复用的缺点**：select/epoll系统调用，属于同步IO，也是阻塞IO。都需要在读写事件就绪后，自己负责进行读写，也就是说这个读写过程是阻塞的。

**其他IO模型**

**1.BIO**(Blocking IO):同步阻塞模型，读写操作阻塞在一个线程完成。需要内核IO操作彻底完成后，才返回到用户空间执行用户的操作

**服务端采⽤单线程**，当 accept ⼀个请求后，在 recv 或 send 调⽤阻塞时，将⽆法 accept 其他请求（必须等上⼀个请求处理 recv 或send 完）⽆法处理并发

**服务端采⽤多线程**，当 accept ⼀个请求后，开启线程进⾏ recv，可以完成并发处理，但随着请求数增加需要增加系统线程，⼤量的线程占⽤很⼤的内存空间，并且线程切换会带来很⼤的开销

**应用：**连接少，低负载和并发，JDK1.4之前

流程：用户线程调用read系统调用，内核(kernel)在数据准备好了之后它就会把数据从内核缓冲区拷贝到用户缓冲区，然后kernel返回结果。从开始IO读的read系统调用开始，用户线程就进入阻塞状态。一直到kernel返回结果后，用户线程才解除block的状态，重新运行起来。

**优点**：阻塞时候线程挂起，用户线程基本不会占用 CPU 资源

**缺点**：在高并发的场景下，需要大量的线程来维护大量的网络连接，内存、线程切换开销会非常巨大

**2.NIO**(Non-Blocking IO):同步非阻塞模型，读写数据到缓冲区，内核会立即返回给用户一个状态值，用户空间不用等到内核的IO操作彻底完成，可以立即返回用户空间执行用户的操作

服务器端当 accept ⼀个请求后，加⼊ fds 集合，每次轮询⼀遍 fds 集合 recv (⾮阻塞)数据，没有数据则⽴即返回错误，每次轮询所有fd （包括没有发⽣读写实际的 fd）会很浪费 CPU

**应用：**连接多且短，高负载和并发，比如聊天服务器，JDK1.4开始有

**Netty**：用于高效开发网络应用的 NIO 网络框架

**流程**：1.在内核数据没有准备好的阶段，用户线程发起IO请求时，立即返回。用户线程需要不断地发起IO系统调用。 2.内核数据到达后用户线程发起系统调用，用户线程阻塞。内核开始把数据从kernel内核缓冲区拷贝到用户缓冲区（用户内存），然后kernel返回结果。3.用户线程解除block状态，重新运行起来。经过多次的尝试，用户线程终于真正读取到数据，继续执行。

**优点** ：每次发起的 IO 系统调用，在内核的等待数据过程中可以立即返回。用户线程不会阻塞，实时性较好。

**缺点**：需要不断的重复发起IO系统调用，这种不断的轮询，将会不断地询问内核，这将占用大量的 CPU 时间，系统资源利用率较低。在高并发场景下不可用

**3.AIO**(Asynchronous IO)：异步非阻塞模型，基于事件和回调实现，指的是用户空间与内核空间的调用方式反过来。用户空间线程是变成被动接受的，内核空间是主动调用者。也就是操作后直接返回，不会堵塞在那，当后台处理完成，OS通知相应线程进行后序操作

**应用：**连接多且长，比如相册服务器，JDK7开始有

**流程**：用户线程通过系统调用，告知kernel内核启动某个IO操作，用户线程返回。kernel内核在整个IO操作（包括数据准备、数据复制）完成后，通知用户程序，用户执行后续的业务操作

**缺点**：需要完成事件的注册与传递，这里边需要底层操作系统提供大量的支持，去做大量的工作

**同步 vs 异步**

同步IO是指用户空间线程是主动发起IO请求的一方，内核空间是被动接受方。异步IO则反过来，是指内核kernel是主动发起IO请求的一方，用户线程是被动接受方。

Java NIO（New IO） 不是IO模型中的NIO模型，而是IO多路复用模型（ IO multiplexing ）

**Java**

**int vs Integer**

int是基本类型，直接存数值；Integer是对象，用一个引用指向它，存的是引用的地址

1、Integer是int的包装类，int则是java的一种基本数据类型   
 2、Integer变量必须实例化后才能使用，而int变量不需要   
 3、Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向 此对象；而int则是直接存储数据值   
 4、Integer的默认值是null，int的默认值是0

**String a = "123"; String b = new String("123"); 讲讲你的理解**

答：a 指向方法区中的字符串常量池，在常量池开辟了一块空间放字符串 123，通过 a 对象指向这个常量对象。而b是new了一个字符串对象，因为使用了 new 关键字，所以会在堆空间开辟一块内存区域放字符串 123，并把内存的地址赋给 b 变量。

**创建对象的方法**：1.new

2.通过class类的newInstance class.newInstance()

3.通过Constructor的newInstance

4.使用clone方法(浅拷贝)，需要实现Cloneable接口(深拷贝)

**异常处理机制**

分为**抛出异常**和**捕捉异常** **异常分Error和Exception**(**受检异常**：用**try catch** **非受检异常**：**运行**时错误，比如**/0**)

**runtimeError:**NullPointer、(Array)IndexOutOfBounds、Arithmetic、ClassCast

**final关键字**：可以用来修饰引用、方法、类 final不能被继承和重写

·修饰引用：1.引用是基本类型，则是常量，他的值不能更改

2.引用数据类型，比如对象，数组，那么对数组、对象本身可以修改，但指向对 象或数组的地址不能修改

3.引用是类的成员变量，则必须当场赋值，否则报错

final class Person {

String name ="zs"; //3. 此处不赋值会报错

//final int age;

final int age = 10;

}

·修饰方法：方法可以被子类继承但不能被重写

·修饰类：类无法被继承(String类用final修饰所以不能被继承)

**String用final修饰**是为了：**1.实现字符串池**--实现运行时节约heap，因为不同字符串变量指向池里同一个字符串(不用final修饰的话字符串就可变了，string interning就不能实现，因为变量改变他的值会导致其他指向这个值的变量也变) **2.线程安全**(被多线程共享的时候不可变) **3.实现String可以创建不可变性Hashcode(创建string的时候hashcode就被缓存了)**

finally 和try里都有return则只会执行finally里的return

try catch里有return了 finally还是会执行，但如果finally里改变变量值无效

**static关键字**四种用法： static(静态)方法只能被继承不能被重写

1.用来修饰成员变量，把它变为类的成员来让所有对象共享这个成员；

2.用来修饰成员方法，把它变为类方法来直接用“类名.方法名”区调用，常用于工具类；

3.静态块用法，将多个类成员放在一起初始化，使得程序更加规整

4.静态导包用法，将类的方法直接导入到当前类中从而来直接用“方法名”就可调用类方 法，更加方便。

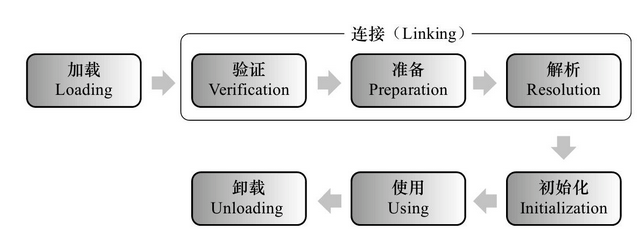
**静态内部类特点**：1 没有有外部类的引用(普通内部类也有) 2.可以直接创建实例，不用先创建外部类(普通内部类要)3.可以有静态成员变量、方法(普通内部类不行)和非静态成员变量、方法；4.只可以直接访问外部类静态成员，不可以直接访问外部类的非静态成员（普通内部类可以），需要通过传入外部类引用的方式才能访问。

**静态内部类使用场景**：外部类与内部类有很强的联系，需要通过内部类的方式维持嵌套的可读性。内部类可以单独创建。内部类不依赖于外部类，外部类需要使用内部类，而内部类不需要使用外部类（或者不合适持有外部类的强引用）。

**关键字修饰的变量什么时候赋值**：对于基本数据类型的字段来说，如果使用static final修饰，则显示赋值（直接赋值常量，而非调用方法）通常是在连接阶段的准备环节进行

对于String来说，使用字面量的方式赋值，使用static final 修饰的话，则显示赋值通常是在连接阶段的准备环节进行

**类加载5个过程：**



**1.加载**（需要类加载器的支持）：在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的入口。（注意这里不一定非得要从一个Class文件获取，这里既可以从ZIP包中读取（比如从jar包和war包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将JSP文件转换成对应的Class类））

**2.验证**：是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

**3.准备**（都是默认值）：正式为类变量分配内存并设置类变量初始值（通常情况下是数据类型的零值），这些变量用的内存都在方法区里分配。这时候进行内存分配的**只包括类变量（被static修饰的变量），不包括实例变量**，实例变量将会在对象实例化的时候随着对象一起分配在Java堆中。

**4.解析**：虚拟机将常量池内的符号引用替换为直接引用

**5.初始化**：执行类构造器clinit（）方法的过程，这个方法是由编译器自动收集类中所有的类变量的赋值动作和静态语句块中的语句合并而成

**多线程异常处理**：线程不允许抛出没捕获的checked exception—线程要把自己exception在自己的run方法里用try catch处理掉，异常被Thread.run()抛出后不能在程序捕获，只能靠jvm捕获。JVM会调用dispatchUncaughtException方法来找异常处理器，再调用他的uncaughtException()方法来处理异常，并且是直接打印到控制台

Java里是**值传递：**调用函数时，将实际参数(地址)复制一份到函数中，函数对参数修改不会影响实际参数

C里是**引用传递**：传实际参数的地址到函数，对参数修改会影响实际参数

**空指针**发生的情况：

1.访问或修改 null 对象的字段。

2.如果一个数组为null，用属性length获得其长度时，访问或修改其中某个元素

3.在需要抛出一个异常对象，而该对象为 null 时

4. 先用equal(“”) 判断再判断是不是null 会报错 NullPointer

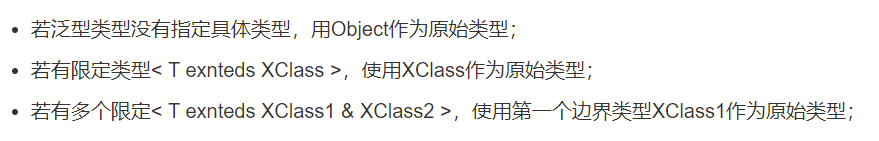
实现**Serializable**接口实现序列化(程序不运行仍保存其信息)—把对象以字节序列的形式保存

反序列化：通过字节序列得到原对象。用于系统间通信

**泛型**：通过参数化类型来实现在同一份代码上操作多种数据类型

类型擦除：List<Object>和List<String> compile后都会变成List

泛型擦除：



4.0-3.6=0.400000001因为**2进制的小数无法精确地表达10进制小数**

**抽象类VS普通类**

1. 普通类可以去实例化调用；抽象类不能被实例化
2. 普通类和抽象类都可以被继承，但是抽象类被继承后子类必须重写继承的方法，除非子类也是抽象类

**接口VS抽象类**

都用abstract声明，有抽象方法的类一定要声明成抽象类

抽象类不能被实例化(类名 a = new 类名())，只能被继承

区别：

抽象类 (is-A关系) 接口 (like-A关系)

有方法实现和构造器 Java8之后有方法实现，无构造器



**为什么接口中的变量只能是public static final的：**

**public**：接口是用来给其他类实现的，要想其实现类使用到接口中的变量就设置成public

**static**：如果变量不是static的，那么接口必须实例化才可以访问自己的变量，接口不能被实例化，所以不用static修饰的变量是无效的

**final**：如果要求将变量定义为静态变量，那么不同类实现同一接口访问到的变量都是同一个，为了防止出现一个[子类](https://so.csdn.net/so/search?q=%E5%AD%90%E7%B1%BB&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_36996785/article/details/_blank)修改了接口中的变量值影响到其他实现了这个接口的子类，所以要把变量设置为final类型，定义为常量

**集合类**

1.来自**Collection**(List,Set,Queue)

List：按位置存储数据的对象，有序 LinkedList

Queue：PriorityQueue为元素提供优先级

Set：和List区别是不允许重复元素

除了Map类都iterable

2.来自**Map**(TreeMap--基于红黑树实现，所以get contains remove..等方法都是logn)

查找效率是O(1), HashMap, LinkedHashMap(比HashMap多维护一个双向链表, 所以可以按照插入的顺序从头部或者从尾部迭代，使元素有序，但性能差)

**重写和重载 Override和Overload**

**重写**：子类实现父类方法声明一样的方法

子类方法访问权限大于父类(父protected，子public)，

子类返回类型和抛出异常类型和父类一样或为其子类型

(父 List<Integer>子ArrayList<Integer>)

static(静态)方法只能被继承不能被重写

**重载**：同一个类中两个方法名字一样，但是参数类型、个数、顺序至少有一个不同

**面向对象(OOP)是一种思想 三大特性**

1.**封装**：将类的某些信息只能类内部访问，public封装性最差，this

2.**继承**：子类拥有父类所有属性和方法, 除了private

static(静态)方法只能被继承不能被重写，final不被继承和重写

3.**多态**：对象的多种形态--3个必要条件：继承，重写，向上转型

实现多态：继承和接口

1.引用多态:父类引用指向本类和子类对象：Animal obj1 = new Dog();子类

2.方法多态:创建子类对象，调用子类重写的方法或继承的方法

**拷贝**

clone(**浅拷贝**)拷贝基本类型值 深拷贝要实现cloneable接口并重写clone方法

引用拷贝：Object obj = new Object() Person cur = obj

**String, StringBuffer, StringBuilder**

3者都是用char数组存字符串

**String(**不可变类)，修改String会生成新String对象，指针会指向新String (浪费内存)

StringBuffer和StringBuilder的对象被修改不 会产生新对象，所以字符串经常变用它们

**StringBuffer线程安全**因为所有公开**方法**都是用**synchronized修饰**的但因为要加锁所以性能差 速度慢

每次获取toString()都会直接用缓存区toStringCache()值来构造一个字符串

**StringBuilder**不是线程安全的，不能被同步访问，但速度快

需要复制一个字符串数组再构造一个字符串

一般用StringBuilder，除非要考虑线程安全

**JVM**

**Java为什么能跨平台:**同一段代码在编译后的字节码是一致的，而同一段字节码在不同的JVM实现上会映射到不同系统的API调用来实现代码不修改就可以跨平台运行

**JDK1.7 vs JDK1.8**: 1.7永久代实现HotSpot虚拟机方法区

1.8里永久代取消，用元空间来实现方法区，但元数据空间不在虚拟机中而在本地内存

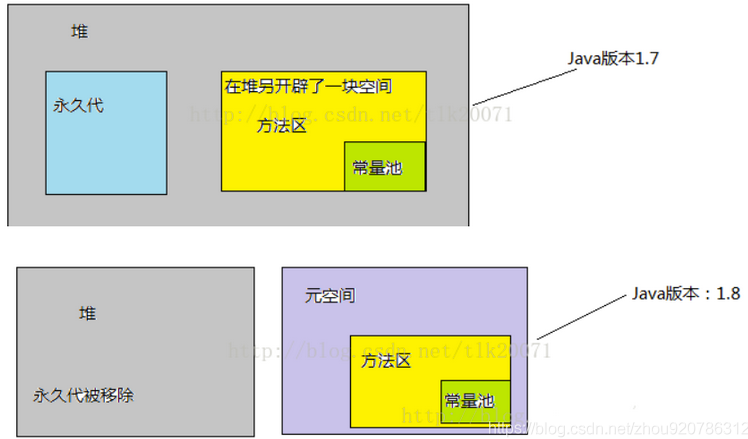
**取代是因为**元空间不用占用JVM内存，让JVM有更大的可用内存空间，永久代所需空间不好控制，不利于调优，容易OOM

**常量池**

**JDK1.7之前**运行时常量池逻辑包含字符串常量池,存放在方法区, 此时hotspot虚拟机对方法区的实现是永久代

**在JDK1.7** 字符串常量池被单独从方法区拿到堆中,运行时常量池剩下的东西还在方法区, 也就是hotspot中的永久代

**在JDK1.8** hotspot移除了永久代用元空间(Metaspace)取代, 这时候字符串常量池还在堆, 运行时常量池还在方法区, 只不过方法区的实现从永久代变成了元空间





**JVM内存组成**：**1.计数器**(当前线程执行的字节码的行号指示器，(PC)存指向下条命令的地址，切换线程后知道要从哪开始执行，线程私有) **2**.**虚拟机栈**(方法执行的模型，线程私有) **3.本地方法栈**(线程私有, 运行本地的Native方法)**4.堆**(共享, 分新生代和老年代 1:2) **5.方法区**(一块线程共享内存, 存已经被虚拟机加载的类信息、常量、静态变量)

**JVM内存模型**决定一个线程对共享变量的写入什么时候对另一个线程可见

把线程间共享变量放在主内存，每个线程都有一个私有的本地内存

**JVM栈**(放引用)：

1. **存**局部变量和方法调用，FIFO数据结构，

2.**空间**比Heap小，创建和释放存储空间快，因为只要移动栈顶指针

3.栈内存属于单个线程因为线程**不共享栈**

**JVM堆 – 被分成新生代和老年代**

1. **存**储实例化对象和数组
2. **分配内存花的时间久**
3. 内存中所有对象**对线程可**见

**垃圾回收**---GC 垃圾： 无引用的对象

JVM提供的在空闲时间不定时**回收无对象引用的对象占据的内存空间的机制**

**通知**GC回收：1.System.gc 2.把不用的对象赋值null

**引用类型：存的值是另一块内存的起始地址**

强引用：只要引用存在就不回收Object a = new Object();

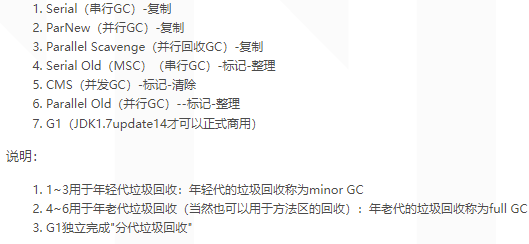
软引用：系统**内存不够**时回收，描述**可能有用**的对象

弱引用：**GC工作时**就被回收

虚引用：**无法取得对象实例**，只是希望被回收时**收到系统通知**

**解决循环引用(只用强引用引起，导致对象永远不能被释放)：**如果A强引用B，那B引用A时就用弱引用，判断是否为无用对象时仅考虑强引用计数是否为0，不用关心弱引用数

**7种垃圾回收器**：



**判断垃圾算法**：

1.**引用计数算法**(引用->计数器+1，引用释放-1，**0时清除**)可能会有循环引用问题(只用强引用导致对象不被释放)解决就是有一方用弱引用，判断是否为无用对象时仅考虑强引用计数是否为0，不用关心弱引用数

2.**根搜索法/可达性分析法**(从GC Root开始往下搜索被根对象集合所连接的目标对象是否可达。使用可达性分析算法后，内存中的存活对象都会被根对象集合直接或间接连接着，搜索所走过的路径是引用链（Reference Chain）如果目标对象没有引用链相连就是不可达的，就意味着该对象己经死亡，可以标记为垃圾对象去清除。只有能够被根对象集合直接或者间接连接的对象才是存活对象。

没有循环引用问题，防止[内存](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/chengqiuming/article/details/_blank)泄漏

**GC Root：1.** Java虚拟机栈 和 Native本地方法栈中所有引用的对象**2.**方法区中的常量和静态变量**3.**跨代引用对象**4.**和已知 GCRoots 对象同属一个CardTable 的其他对象

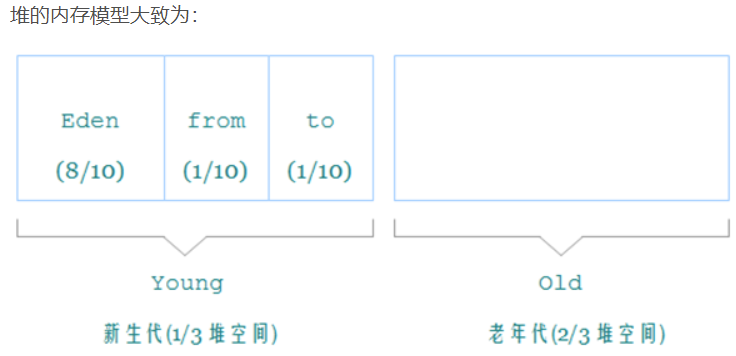
**回收算法**：1.标记清除法(**标出所有**要回收的对象统一回收)--效率低

2.标记整理法(**标出**要回收的对象，让存活的向一端**移动**，直接清边界外的)-无

3.复制法(把内存**分成两块**，先用一块，把**存活的对象放另一块**)-占内存

4.分代收集法(把内存分成**新生代**和**老年代**，分别用**复制法**和**标记整理法**)

新生代分3个区域Eden，from Survivor，To Survivor—这样分时为了更好地管理堆内存里的对象，包括内存的分配和回收

**堆大小=新生代+老年代 1：2**

**新生代**：对象存活期短，所以用复制法，只用关心哪些要被复制，一般使用两块10%的内存作为空闲和活动区间，另外80%的内存用来给新建的对象分配内存。发生GC的时候把10%的活动区间与另外80%里存活的对象转移到10%的空闲区间，再之前90%的内存全部释放 （**缺点**是浪费一半内存（**方法的局部变量引用的对象等**））

**老年代**：对象存活得比较久，但还是要被清理的。用的标记清理/整理法，回收次数相对较少，每次回收时间比较长。这部分**如果用复制法**的话没有额外空间担保(因为新生代放不下会放到老年代)，而且对象存活率高，复制的开销大（**缓存对象、单例对象等**）。

**永久代**：是**虚拟机内存中的方法区**，里面对象生成后几乎不灭，GC比较少，效率也比较低，但也必须进行垃圾回收，否则永久代内存不够用时仍然会抛出OutOfMemoryError异常（**加载过的类信息**）

**为什么分代**：不分代的话得所有root(一组活跃的引用)要扫一遍然后要知道是不是在引用链上还要扫一遍。老年期的对象长期存活，所以总的GC频率和分代以后的年轻代GC频率差不多，但不分代每次都要从GC roots完整来遍历堆，会大大增加开销

**MinorGC**清理年轻代(熬过一次清理年龄+1，到15后移到老年代)、**MajorGC**清理老年代(满的时候清)、**FullGC**清理整个堆空间，包括永久代(放.class文件)

**新生代survival区满了怎么办：**  
如果服务器内存够用，直接增大新生代空间(如 -Xmn)。如果不够用，就增加 Sur[vivo](/jump/super-jump/word?word=vivo" \t "_blank)r 空间，减少 Eden 空间， 但是注意减少 Eden 空间会增加 Minor GC 频率， 要考虑到应用对延迟和吞吐量的指标最终是否符合。要增大多少 Sur[vivo](/jump/super-jump/word?word=vivo" \t "_blank)r 空间？ 需要观察多次 Minor GC 过程， 看 Minor GC 后存活下来的对象大小， 最终确定 Sur[vivo](/jump/super-jump/word?word=vivo" \t "_blank)r 的合适大小。 整个调优过程可能需要几次调整， 才能找到比较合适的值。调整几次后， 如果内存还是不够用， 就要考虑增大服务器内存，或者把负载分担到更多 JVM 实例上

**JVM调优步骤**：1.分析GC日志及dump文件,判断是否需要优化，确定瓶颈问题点

2.确定JVM调优量化目标；3.确定JVM调优参数（根据历史JVM参数来调整）

4.调优一台服务器,对比观察调优前后的差异 5.不断的分析和调整,直到找到合适的JVM参数配置 6.找到最合适的参数,将这些参数应用到所有服务器,并进行后续跟踪。

**内存泄漏 vs 内存溢出**

**内存泄漏**：非必要的对象引用**没清除**(数组里**添加对象后不处理**)，**资源没释放**

**内存泄漏leak的情况--会导致outOfMemory**：

1.静态集合类如hashmap、Linkedlist，如果这些容器是静态的，它们的生命周期与程序一致，容器的对象在程序结束前不能被释放。

2.改变对象哈希值导致hashset里面无法删除当前对象

3.一个变量的定义的作用范围大于使用范围

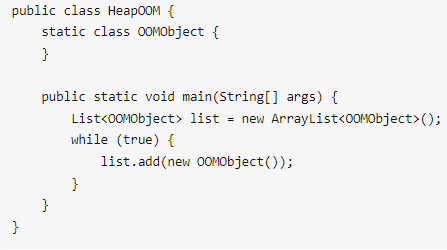
4.内部类持有外部类

5.各种连接，比如数据库连接，没用close方法就不会去回收了

**内存溢出(out of memory)**：申请资源的时候，JVM没有足够的资源(除了计数器，都有可能溢出)

**内存溢出的情况**：

1. Java堆溢出：不断创建对象，并保证GC Root到对象之间有可达路径来避免垃圾回收机制清楚这些对象。-----造成outOfMemoryError



1. 虚拟机栈和本地方法栈溢出

StackOverflowError:如果线程请求的栈深度大于虚拟机所允许的最大深度或者无限递归

OutOfMemoryError：如果虚拟机的栈内存允许动态扩展，当扩展栈容量无法申请到足够的内存时

**智能指针**：一种抽象的**数据类型**，会**跟踪**指向它们的**对象进行内存管理**，通过使资源自动分配来**防止内存泄漏**。指向对象的指针被破坏时(超出范围)，对象**也会被破坏**

**反射**：Java compile后会生成.class文件，反射就是通过**字节码文件**找到类提供**类运行时的信息**。借助**field**(可以用get和set得到和修改field对象) **method，constructor，class**这4个类。 缺点：性能消耗大

反射实现原理：在JVM层面，java的**对象引用**不仅要可以**直接**或间接地得到**对象类型**，更应该可以根据**索引**能得到对象类型

**JVM加载.class文件**：类加载器将类的.class文件中的二进制数据读入内存，将其放在运行时数据区的方法区内，然后再heap创建java.lang.Class对象用来封装类在方法区的数据结构

**双亲委派**：类加载器**加载.class文件**时，先**递归**地把这个任务委托给上级类加载器，上级类加载器没加载，自己才加载

作用：**防止重复加载**一个.class文件，保证核心.class文件不被篡改

**JVM 里 new 对象时,堆会发生抢占吗?怎么去设计JVM的堆的线程安全**

会，假设JVM虚拟机上，堆内存都是规整的。堆内存被一个指针分为两部分，左边塞满了对象，右边是没使用的区域。每一次new对象，指针就会向右移动一个对象size的距离。如果我们多线程执行new对象的方法，一个线程在给一个对象分配内存，指针还没有来的及修改，其它线程给另一个对象分配内存，而且这个线程还是引用之前的指针指向，这样就会出现抢占，也就是**指针碰撞**

**解决**：在JVM新生代开辟一块TLAB(线程本地分配缓存区)，也就是一块线程私有的内存分配区域。然后用这块区域放小对象，因为Java里有很多小对象而且小对象都是用完就被垃圾收集的，也不会被线程间去共享，而且因为线程私有所以对象分配的时候不用锁住整个堆，不存在竞争的情况，直接在自己缓冲区分配就行

TLAB**缺点**：

**但**TLAB只是让每个线程有私有的分配指针，底下存对象的内存空间还是给所有线程访问的，只是其它线程无法在这个区域分配而已 +大小固定

**ThreadLocal(线程自己的变量)**：保证线程安全的方法，创建一个变量后每个线程对其访问的时候都是访问线程自己的变量 （Java里）子线程能获得父线程ThreadLocal。

**多线程的时候不会像锁那样竞争时候导致的性能下降的影响**

private ThreadLocal<Integer> localInt = new ThreadLocal<>();

保存在ThreadLocalMap里，每个entry的key是弱引用(好处是不被其他对象引用时就能回收threadlocal这个对象，避免内存泄漏)，但value是强引用，只有thread被回收才会被回收，所以map在进行set(),get(),remove()的时候都会进行清理

## 线程同步的实现/机制

**可见性**：多个线程共同访问共享变量时，某个线程修改了此变量，其他线程能立即看到修改后的值

**可见性问题**：对于普通共享变量，线程A将变量修改后，体现在此线程的工作内存。在没同步到主内存时，线程B用这个变量，从主内存中获取到的是修改前的值，就发生了共享变量值的不一致

**JMM**(Java内存模型，定义程序中各个变量的访问规则，也就是在虚拟机中将变量存储到内存和从内存中取出变量这样的细节)：规定所有的变量都存在主内存，线程有自己的工作内存，里面保存了线程用的变量的从主内存里拷贝的副本。线程对变量的读写都必须在工作内存，而不能直接读写主内存中的变量。同时，本线程的工作内存的变量也无法被其他线程直接访问，必须通过主内存完成。

**有序性**：程序执行的顺序按照代码的先后顺序执行（由于JMM模型中允许编译器和处理器为了效率，进行指令重排序的优化。指令重排序在单线程内表现为串行语义，在多线程中会表现为无序。那么多线程并发编程中，就要考虑如何在多线程环境下可以允许部分指令重排，又要保证有序性）

**原子性：**一个操作或者多个操作要么全部执行成功要么全部失败，满足原子性的操作，中途不可被中断。

## **1.Synchronized**(独占锁/悲观锁, 可以修饰方法和代码块)

**保证原子性和有序性**：synchronized是同步锁，同步块内的代码相当于同一时刻单线程执行，所以不存在原子性和指令重排序的问题(有序性)

**底层原理**：线程1要加锁，执行monitorenter指令->看计数器 -> 0就**获取锁并计数器改为 1** -> 线程2想加锁执行了monitorenter指令发现计数器是1就阻塞->线程1执行完释放锁， 执行monitorexit将计数器置为0->线程2进入

**保证可见性：**synchronized的语义JMM有两个规定：1. 线程解锁前要把共享变量最新值刷新 到主内存2. 线程加锁前，要清空工作内存里共享变量的值，从主内存里冲洗取值

**2. Lock** 保证多个线程都是读或写时候就能一起进入 (CAS(保证了原子性—串行)和 volatile实现) 保证一个线程有资源，其他 **spin**

**ReentrantLock(重入锁)**实现**线程安全**: 底层通过**CAS操作和AQS队列**去维护 state变量的状态

**流程：**先通过CAS操作尝试修改state状态获取锁，如果获取失败就判断当前占用 锁的是不是自身，如果是的话就进行重入，如果不是就进入AQS队列等待

**3.volatile**轻量级的线程同步机制，但只能用于变量

**保证可见性：**当对volatile变量执行写操作后，JMM会把工作内存中的最新变量值强制 刷新到主内存。写操作会导致其他线程中的缓存无效。这样，其他线程使用缓存时， 发现本地工作内存中此变量无效，便从主内存中获取，这样获取到的变量便是最新 的值，实现了线程的可见性。

**保证有序性：**编译器在生成字节码时，在指令序列中添加“内存屏障”来禁止指令重排序(分为硬件层面和JMM内存屏障，一些读写屏障)。也就是JVM的实现会在volatile读写前后都加上内存屏障来保证有序性

**JMM加内存屏障的4种情况：**

1.每个volatile写操作前加StoreStore屏障防止写volatile和后面的写操作重排序

2.每个volatile写操作后加StoreLoad屏障防止写volatile与后面的读操作重排序

3.每个volatile读操作后加LoadLoad屏障防止读volatile与后面的读操作重排序

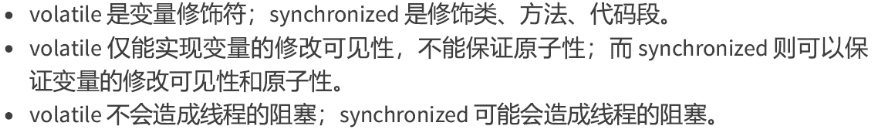
4.每个volatile读操作后加LoadStore屏障防止读volatile与后面的写操作重排序

底层原理：修饰后在总线开启MESI缓存协议和CPU总线的监听 -> 这个变量值修改就会传到总线 -> 总线告诉其他在使用这个变量的线程 -> 其他线程获取新的值

**4.wait()/notify()**

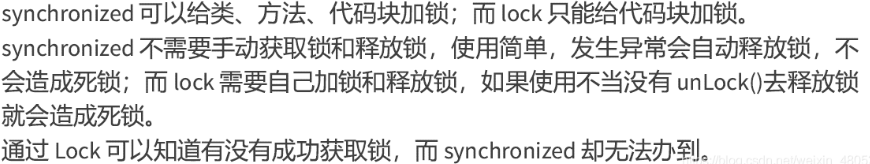
**比较：**

volatile vs synchronized

****

volatile比Synchronized性能好

Lock vs synchronized



Lock有Synchronized所有功能, 但需要手动加锁/解锁(因为可以手动控制，所以再复杂的并发场景用得上), Synchronized会加锁解锁，操作简单，用于一般并发场景

**锁升级/锁机制切换**：根据竞争激烈程度进行

1.在几乎无竞争的条件下， 会使用偏向锁 2.在轻度竞争的条件下， 会由偏向锁升级为轻量级锁 3.在重度竞争的情况下， 会升级到重量级锁。

**ReentrantLock(用AQS构建)和synchronized的区别**

1. 从底层实现上来说，synchronized 是JVM层面的锁，是Java关键字，可以修饰方法和代码，它是通过monitor对象来完成，对象只有在同步块或同步方法中才能调用wait/notify

ReentrantLock 是从jdk1.5（java.util.concurrent.locks.Lock）提供的API层面的锁

synchronized 的实现涉及到锁的升级 (具体为无锁、偏向锁、自旋锁、向OS申请重量级锁)

ReentrantLock实现是通过利用CAS（CompareAndSwap）自旋机制保证线程操作的原子性和volatile保证数据可见性来实现锁

2.能不能手动释放：

synchronized不用用户释放锁，它的代码执行完系统会自动让线程释放锁

ReentrantLock要手动释放锁，没释放可能会导致死锁，在final语句块里解锁

同一个线程在外层方法获取锁的时候，进入内层方法会自动获取锁。好处：避免死锁

3.能中断：

synchronized是不可中断类型的锁(除非加锁的代码中出现异常或正常执行完成) ReentrantLock可以中断，(可以通过trylock(long timeout,TimeUnit unit)设置超时方法或者将lockInterruptibly()放到代码块中，调用interrupt方法进行中断）

4.公平锁？

synchronized是非公平锁

ReentrantLock可以是公平锁也可以是非公平锁(通过构造方法new ReentrantLock时传入boolean值进行选择，为空默认false非公平锁，true为公平锁)

5.能不能绑定condition：

synchronized不能绑定

ReentrantLock可以绑定(通过绑定Condition然后结合await()/singal()方法实现线程的精确唤醒，而不是像synchronized通过Object类的wait()/notify()/notifyAll()方法要么随机唤醒一个线程要么唤醒全部线程)

6.锁的对象

synchronzied锁的是对象，(锁保存在对象头里面，根据对象头数据来标识是否有线程获得锁/争抢锁)

ReentrantLock锁的是线程，(根据进入的线程和state标识锁的获得和争抢)

**AQS**（Abstract Queued Synchronizer）

AQS是一个用来构建锁和同步器的框架，可以构建比如ReentrantLock，信号量，ReentrantReadWriteLock

**核心思想**：1.如果被请求的共享资源**空闲**，则将当前请求资源的线程设置为有效的工作线程，同时把共享资源设置为锁定状态。

2.如果被请求的共享资源被**占用**，那就要一套线程阻塞等待和被唤醒时锁分配的机制，这个机制AQS用**CLH队列锁**实现：把暂时获取不到锁的线程加到队列里。然后 AQS用voliate int成员变量来表示同步状态，它是通过内置的FIFO队列对获取的线程排队

AQS用CAS对这个同步状态进行原子操作来实现对它的值的修改。

AQS两种**资源获取方式**：**独占式**(只有一个线程有锁，又根据是否按**队列的顺序**分为公平锁和非公平锁，如ReentrantLock)和**共享式**(多个线程同时获取锁访问资源，如Semaphore/CountDownLatch，Semaphore、CountDownLatCh、 CyclicBarrier) ReentrantReadWriteLock 可以看成是组合式，允许多个线程同时对某一资源进行读

AQS**底层**：用了模板方法模式，自定义同步器了，然后在实现的时候只要实现共享资源 state 的获取与释放方式就可以了(至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在上层已经帮我们实现好了)

**CLH同步队列是怎么实现非公平和公平的**？--公平锁和非公平锁

CLH(FIFO的双向链表)用于等待资源释放的队列，AQS用它来管理同步状态

公平锁和非公平锁在于hasQueuedPredecessors()方法，如果头节点不是尾节点，第一个节点不为空，而且当前节点是头节点就返回true

线程在doAcquire方法里获取锁的时候会先加到同步队列再根据情况陷入阻塞

阻塞后的节点一段时间后醒来，这个时候来了更多的新线程来抢锁，这些新线程还没有被加到同步队列里去，也就是还在tryAcquire方法里获取锁

在**公平锁**情况下，这些新线程会发现同步队列里有节点在等待，然后这些新线程就不能获取到锁，就去排队了

在**非公平锁**下，这些新线程会跟排队苏醒的线程抢锁，失败的线程就去同步队列里排队。所以这个**公不公平针对的是**苏醒线程和还没加到同步队列的线程，

然后根据AQS节点唤醒机制和同步队列的FIFO性质，那些已经在同步队列里阻塞的线程，它们内部其实是公平的，因为它们是会按顺序被唤醒的

## Java锁的类型

1.乐观锁：拿数据时认为别人不会修改，所以不会上锁，但是在更新时候会

适用于读操作多时，实现方式有compare and swap(CAS)算法

轻量级锁的算法是CAS是线程安全的，因为他有原子性(全执行成功or 全失败)

悲观锁：每次拿数据都认为别人会改，所以每次都上锁

适应于写操作多的场景，因为加锁保证写时数据正确

synchronized和lock都是悲观锁

2.独享锁(一次只能一个线程持有) 共享锁(可以被多个线程持有)

实现：互斥锁和读写锁

3.公平锁：线程**按申请锁的顺序获取锁**，非公平锁相反(都是双向链表)

非公平锁优点：吞吐量比公平锁大

4.可重入锁：同一个线程在外层方法获取锁的时候，进入内层方法会自动获取锁。好处：避免死锁

**Object类的方法：equals() hashCode() wait()**

== 获取对象的值

**equals**重写场景：比较对象内容 String，Integer -> 使equals和= =不同

**hashcode**重写场景：不希望造成多个对象hash值相同，值不被覆盖

相等的对象必须有相等的哈希码，两个对象hashCode相同他们不一定相同

**Hashmap和Hashtable区别**

底层都是哈希算法，接口一样

**HashMap**：区别：1.线程不安全(多线程下put会形成环导致死循环)因为1.7里扩容会造成数据丢失 1.8会有数据覆盖 2.可以存null:null 3.继承AbstractMap类

**put原理**：先对key用hashcode方法，然后用hashcode的结果找到bucket位置来存键值对，也就是作为了Map.Entry，如果hashcode一样就发生哈希冲突了--有一些方法解决。用equals比较，因为hashcode一样不一定就相同，equals比较还一样就替换

**get原理**：对key用hashcode后得到桶位置，之后用equals找链表里的节点

**作为key：**String，Integer这样的wrapper类，因为他们是不可变—线程安全，也是final的而且重写了equals和hashcode方法

**1.7：当哈希冲突严重时，桶上的链表会很长，查询很慢O(n)**

**1.8：链表长度小于8，用链表 因为查询快**

**链表长度>8+数组长度>64转化为红黑树--泊松分布统计得出**

查询为O(logn) 不用AVL树因为插入太慢，经常会调整树的结构

**扩容**：初始大小(固定2的倍数)=16 扩容因子=0.75 扩容一倍 也就是容量16在有12个元素的时候就扩容。0.75是提高空间利用率和减少查询成本的折中，主要是泊松分布，0.75的话碰撞最少。扩容因子越大，碰撞概率也越大

**为什么是2的幂次方：**因为在hashMap的length等于2的n次方的时候，才会有hash%length==hash&(length-1)；哈希算法的目的是为了加快哈希计算和减少哈希冲突，所以&操作更合适，所以在length等于2的幂次方的时候，可以使用&操作加快操作而且减少冲突，所以hashMap长度是2的幂次方

**链表扩容机制：**单线程调用rehash遍历链表，把元素每次用transfer方法放到新的链表头，链表元素次序会反过来因为每次都是从头插入来避免O(n)去尾部遍历

多线程下transfer会导致**死锁**，比如A.next=B, B.next=A—用ConcurrentHashMap

**为什么HashMap不用平衡树**(AVL)--解决二叉树退化成链表的情况

因为**平衡树(AVL树)**追求绝对平衡(search时候更好)，条件比较苛刻(左右子树高度差不能超过1)，实现起来也比较麻烦，每次插入和删除节点都要判断，然后之后需要旋转的次数也不能预知。总之就是实现的代码量会特别大，而且会增加额外开销，所以用红黑树

**红黑树**用颜色标识，是大致平衡的，在和平衡二叉树的时间复杂度相差不大的情况下，能保证每次插入最多只要三次旋转就能平衡，实现起来也更简单

他们插入节点rebalance都要O(1)，但删除的话AVL树就要O(logn) 因为要维护从被删除节点到根节点root这条路径上所有节点的平衡

**红黑树的性质---导致大致平衡**：

1.节点是红色或黑色2.根节点是黑色3.每个叶子节点都是黑色的空节点（NIL节点)4.每个红色节点的两个子节点都是黑色(从每个叶子到根的所有路径上不能有两个连续的红色节点)5.从任一节点到它每个叶子的所有路径都有相同数目的黑色节点

从根到叶子的最长的可能路径不会大于最短的可能路径的两倍长。因为操作比如插入、删除和查找某个值的最坏情况时间都要求和树的高度成比例，这样就允许**红黑树在最坏情况下都是高效的**，(这个是不同于普通的二叉查找树)

**HashTable**：区别：线程安全(每个方法已有synchronize方法，所以多并发时候用)， key或value不能为null，继承Dictionary类(废弃了)，用hashcode 取余计算hash值

**HashSet** 实现了Set接口，是HashMap的一个实例，只存key不存value，因此重写了equals和hashCode来判断Key存不存在(先用hashcode和其他对象比较结果，一样再用equals，返回true说明有一样的存在)，不存在才插入，所以key**不重复**

**ConcurrentHashMap**

**1.7**(两个静态内部类)(底层是分段数组和链表)用HashEntry(封装映射表的键值对)和Segment(充当锁)守护若干个桶(每次在每个链表头插入节点，所以节点顺序和插入顺序相反)每个线程只能访问不同的段，提高了并发率

**线程安全,** 因为它用了分段锁，把一个 Segment切割成多个HashEntry数组，然后 Segment 可以看成一个 HashMap，然后Segment 继承了 ReentrantLock，在操作的时候会锁住 Segment，想要修改HashEntry必须获得对应的Segment的锁，这样每个segment线程安全了就全局线程安全了。但是容易冲突让链表太长，这样查询就慢了

**1.8**(底层和HashMap一样链表长度大于8 | 数组>64变成红黑树)**放弃了segment**，在节点上用**CAS+Synchonized**(锁住红黑树的root)来保证**并发的安全**，只要hash不冲突就不会有并发问题。然后是用volatile关键字来记录元素个数，这样变量值变了的话其他线程也能看见。就不会脏读了，提升了可见性

**插入put过程**：判断需要插入的key/vlaue是否为null(不允许有null)，计算hash。

遍历存放数据的Node数组，并判断数组是否为空，为空就初始化。根据算出数组下标并判断该点是否为空，若为空，用CAS尝试添加操作。还要判断数组是不是正在扩容，是的话就协助扩容。之后对这个node节点加锁，同时插入元素。再判断链表长度，看看是否要转化红黑树。最后维护数组长度，并判断要不要扩容。

**get()过程**：get过程不需要加锁，get方法里将要使用的共享变量都定义成volatile

检查桶的长度是否为空，或者根据key得到具体哪个桶中没有数据，那么直接返回null，如果桶不为空且要查询的桶中有元素，那么桶中第一个元素的[hash](https://so.csdn.net/so/search?q=hash&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_29066329/article/details/_blank)值如果大于0说明为链表节点，就遍历链表查询，小于0说明是树节点或者正在扩容，那么调用Node子类的find函数进行查询

**size()方法**：**1.7**用不加锁的模式最多三次计算size，比较计算结果，一致的话就认为当前没有元素加入，计算结果是正确的。如果这种方法不行就给每个segment上锁然后计算size。**1.8**里通过对 baseCount 和 counterCell 进行 CAS 计算，最终通过 baseCount和遍历 CounterCell 数组得出 size

**初始化保证线程安全**：多个线程同时进行put操作，在初始化Node[]数组时用CAS操作来决定哪个线程有资格进行初始化，其他线程只能等待。用volatile保证每次获取的都是最新值，CAS保证sizeCtl标记位的原子性

**优化了什么**/为什么用：

HashMap用synchronized修饰对象整体会被锁住

HashTable大小增加到一定的时候，性能会急剧下降，因为迭代时候要被锁很久

CAS(compare and swap)：乐观锁的一种实现机制，实现线程安全

思想：包含三个操作数——内存位置（V）、预期原值（A）和新值（B）：当一个线程需要修改一个共享变量的值，完成这个操作需要先取出共享变量的值，赋给A，基于A进行计算，得到新值B，再用预期原值A和内存中的共享变量值比较，如果相同就认为其他线程没有进行修改，然后把新值写入内存

**CAS缺点**：1.CPU开销大：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，又因为自旋的时候会一直占用CPU，如果CAS一直更新不成功就会一直占用，造成CPU的浪费

2.不能保证代码块的原子性：CAS机制所保证的只是一个变量的原子性操作，比如要保证3个变量共同进行原子性更新就要用synchronized

3.ABA问题



**ABA解决:**

1.使用版本号，也就是在变量前面追加版本号，每次变量更新时把版本号加1顺序递增，那么A-B-A就会变成1A-2B-3A。

2.从jdk1.5开始，jdk中的Atomic包里提供了一个类AtomicStampedReference来支持原子条件的更新。这个类的compareAndSet方法的作用首先检查当前引用和标志是不是等于预期的引用和标志，如果都相等，就以原子方式把这个引用和标志的值设为给定的更新值。

AtomicMarkableReference更新(对象引用, 布尔值)二元组，通过这种二元组把节点保存在链表中同时把他们标记为已删除节点

**ArrayList、LinkedList**

**1.Array**能**包含基本类型和对象类型** + **大小固定**

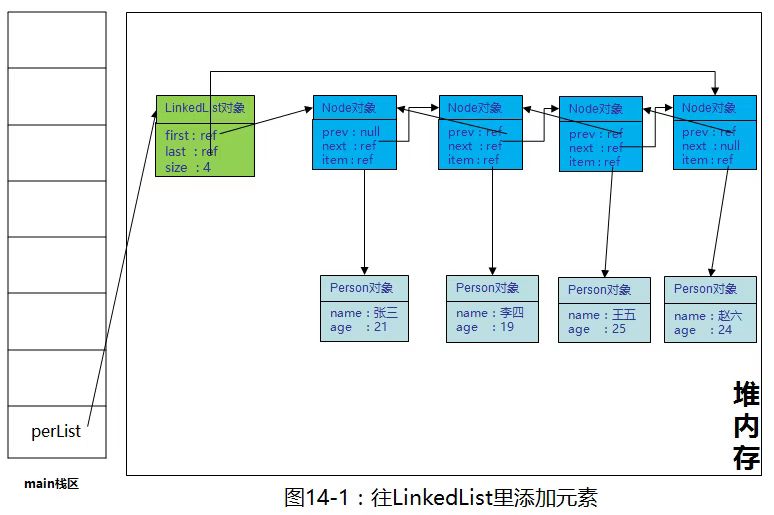
**2.ArrayList**只能包含**对象类型+大小动态变化** 访问效率高, 插入和删除可能要移动整个list。**初始大小=10 每次1.5倍扩容**

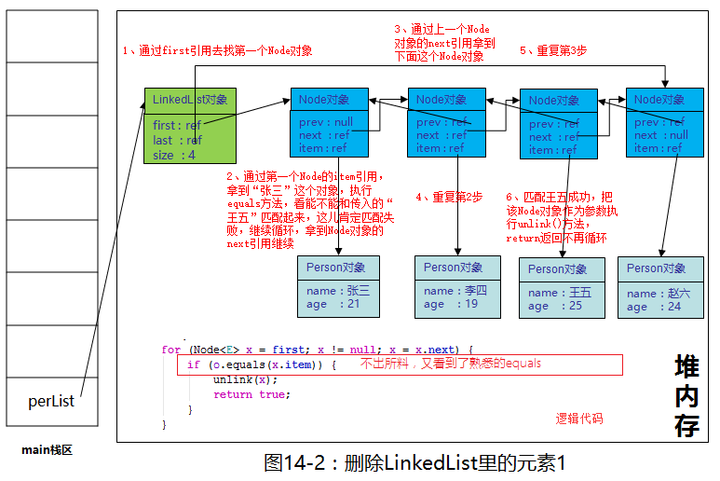
**Add底层：**add(元素)会加在最后，不会触发底层的数组的复制，add(元素,index)会触发数组复制，最坏时间复杂度是O(n)。new ArrayList的时候是一个空的object数组，大小=0，第一次加元素，大小变成10，每次add都会去检查数组空间够不够，不够就用grow方法扩容1.5倍，扩容完用System.arraycopy拷贝数组

**Remove底层：**删除一个元素后把后面的元素往前移一个index并且把最后一个index设为null，底层移动是用System.arraycopy(elementData, index + 1, elementData, index,

numMoved);把后面的元素到前一位

**3.LinkedList**增删节点效率高，只是修改了引用地址所以是O(1)，但链表定位到要增删的地方是O(n)，但写入是O(1)，Arraylist是O(n)。所以Arraylist擅长读取，链表擅长写入 双向链表 没有扩容机制

**Add底层：**

**Remove底层：**

ArrayList(无原子性)和LinkedList**变安全**：进行读操作时获取读锁，进行增删改操作时获取写锁(用读写锁类ReentrantReadWriteLock)

**4.Vector**也是用数组方式存储，但加了synchronized修饰，所以线程安全，

但是性能比ArrayList差

**设计模式**

**创建型：**提供对象创建的功能，并把对象创建的过程进行封装隐藏，让使用者只关注具体的使用而不是对象的创建过程。它包含有**[单例模式](https://so.csdn.net/so/search?q=%E5%8D%95%E4%BE%8B%E6%A8%A1%E5%BC%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/jianghao233/article/details/_blank)、工厂模式**、原型模式、抽象工厂模式、建造者模式

**结构型：**关注的是对象的结构，它用组合的方式把类结合起来用来实现新的功能。它包含**代理模式**、组合模式、装饰模式和外观模式

**行为型：**关注对象的行为，它是把对象之间的关系进行梳理来划分和归类。它包含**观察者模式、策略模式、访问者模式**、**模板方法模式**、命令模式和责任链模式。

**一、单例模式(创建型):用来保证一个类有且只有一个实例，用来解决一个全局使用的类频繁的创建和销毁问题，从而节约系统资源。(性能高)**

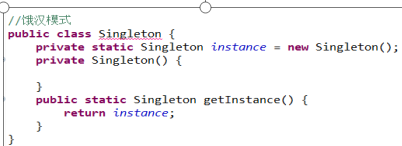
**应用于IOC，Spring 中 Bean 实例就是单例模式**

**实现方法**：实例存在直接返回，不存在才创建实例

**优点**：节约内存，提高对象访问速度，避免了创建和销毁对象的消耗

**缺点**：一般不会实现接口，扩展性不好，而且单例类在一个方法中既创建了类又提供类对象的复合操作，这样违背了单一职责原则

**饿汉模式**：没有用Synchronize关键字，在方法调用前，实例就已经创建好了，所以效率较高，**线程并发安全**, 但是没有达到lazy loading的效果

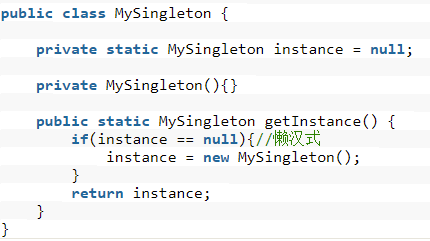


**懒汉模式**：在方法调用获取实例时才创建实例。用静态内部实现，使用时才会创建实例对象。(严格来说加了synchronized才算单例)

**加了synchronized才能线程安全**不然多线程并行被调用的时候会创建多个实例， 但。相应的我们需要保持同步的代码块仅仅是：

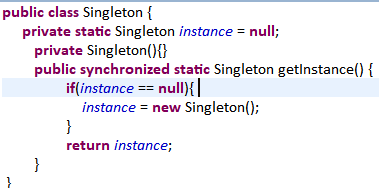
instance = new Singleton();所以只要在该代码处添加同步锁就可以了，也就是双检锁的里的效果

**多线程不安全版本**：

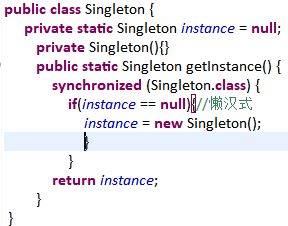


**多线程安全的两种实现**：

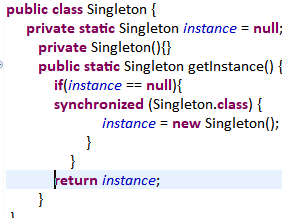
**1.方法中声明synchronized保证多线程下线程安全，但不能保证高性能因为**synchronized加在了方法上:public static synchronized Singleton getInstance(){.....}，多线程每次在调用getInstance()时都会产生一个同步



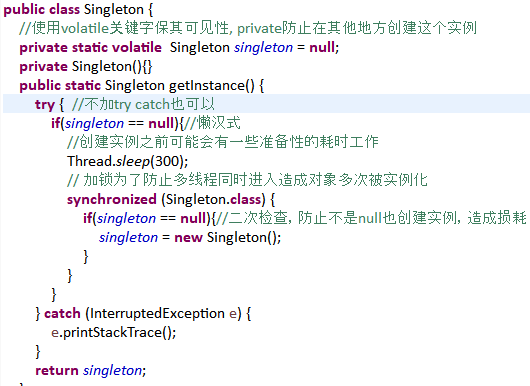
1. **同步代码块实现，保证多线程下线程安全，但是效率同样很低**因为这样的实现把全部的代码都被锁上了



**3.对单独代码进行同步，保证高性能，但多线程不安全**



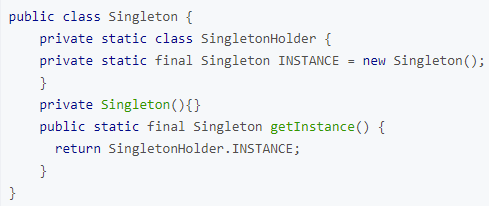
**双检锁/双重检验锁：用volatile和synchronized来保证线程安全且在大多情况保持高性能**，达到lazy loading效果。



**静态内部类**

⽤了⼀个私有的静态内部类，来存储外部类的单例，这种静态内部类，⼀般称为 Holder。⽽利⽤静态内部类的特性，外部类的 getInstance() ⽅法，可以获取指向 Holder 持有的对象。Holder 模式借⽤了饿汉模式的优势，就是在加载类（内部类）的同时对 instance 对象进⾏初始化。由于⾃始⾄终类只会加载⼀次，所以在多线程的情况下，也能够**保持单例的性质**。

这种写法由于 SingletonHolder 是私有的，除了 getInstance() 之外没有办法访问它，又因为外部类加载时、并不需要立即加载内部类，因此该写法是**懒汉式的、懒加载**的。同时访问实例对象的时候，不需要进行同步措施，还能保证**线程安全，没有性能缺陷**

  
**二、工厂模式(创建型)**:定义用于创建对象的接口，让子类决定实例化哪个类

**应用**：给那些因为需求变化而变化比较剧烈的子类提供稳定的接口

**优点**：使代码结构清晰，能更有效的封装, 对调用者屏蔽产品实现来降低耦合度

**缺点**：对于简单对象，使用工厂模式会增加其系统的复杂度

**三、原型模式(创建型)**指通过“克隆”来产生一个新的对象。所以它的核心方法是 clone()，我们通过该方法就可以复制出一个新的对象。只需要实现 Cloneable 接口，并重写 clone() 方法就可以实现克隆了

**优点**：性能高，因为它是通过直接拷贝内存中的二进制流实现的复制

**缺点**：在对象层级嵌套比较深的时候，复制的代码实现难度比较大

**四、代理模式(结构型)**：为其他对象创建代理来控制对这个个对象的访问，在一些已有的方法在使用的时候需要对已有的方法进行拓展，可以用这个模式

**应用**：AOP

**使用场景：**当直接访问某个对象存在困难或者不想直接访问时可以通过代理对象来间接访问，为了保证客户端使用的透明性，委托对象与代理对象需要实现相同的接口

**静态代理：**使用时要定义接口或者父类，被代理对象和代理对象一起实现相同的接口或者是继承相同父类。

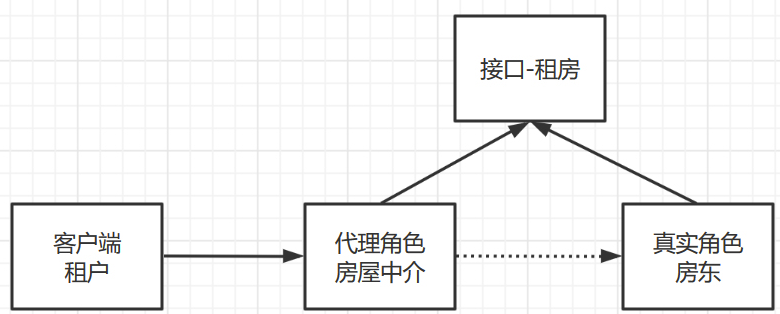
例子：HttpServletRequestWrapper(代理类) HttpServletRequest(接口)

**抽象主题类**：用接口或者抽象类来表示，声明真实主题和代理的共同接口方法。

**真实主题类(被代理类)**：定义了代理所表示的真实对象，由它执行具体的业务逻 辑方法，客户类通过代理类间接地调用真实主题中定义的方法

**代理类**：持有一个对真实主题类的引用，在它所实现的接口方法中调用真实主题类中相应的接口方法执行来起到代理的作用

**客户类**：使用代理类的类型



**关系**：客户端和代理类：客户端直接访问代理(房屋中介)，客户端不能越过代理 来访问真实类

**优点：**可以使真实主题类的操作更纯粹，不用关注一些公共的业务。公共业务都交给代理角色，实现了业务的分工。公共业务发生扩展的时候，方便集中管理

**缺点**：一个真实主题类就会产生一个代理角色，代码量会翻倍，开发效率低

**动态代理**

有2种实现：JDK自带的Proxy代理和第三方工具Cglib

**好处**：1.一个动态代理类代理的是一个接口，接口对应的是某一类业务，比如：房屋租赁业务。

2.一个动态代理类可以代理多个类，这些类实现同一个接口即可。

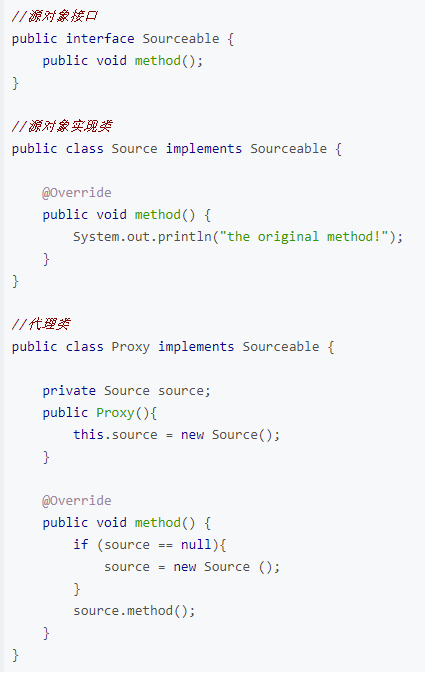
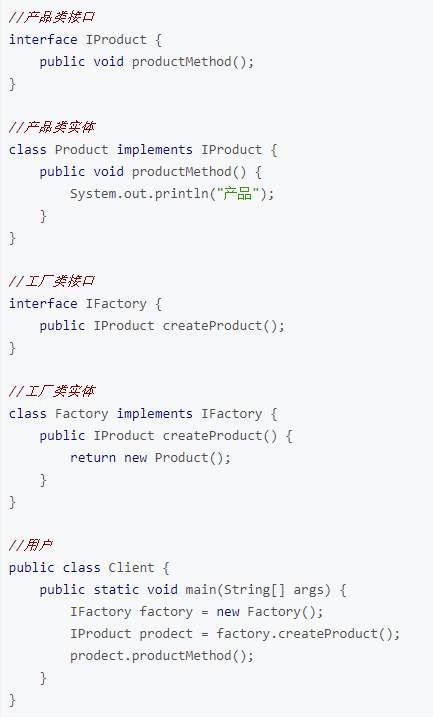
**相比于静态代理**一个代理绑定一个真实主题类。动态代理一个代理绑定多个真实主题，代码层面就是都实现了一个的接口

**JDK的Proxy动态代理(基于接口)**：是委托机制，只能对实现接口的类生成代理，主要通过使用反射机制实现AOP的动态代理，生成一个实现代理接口的匿名类，在动态生成的实现类里面在调用具体方法前委托hanlder去调用原始实现类方法来处理。代理对象不需要实现接口，但是目标对象一定要实现接口，否则不能使用动态代理。

**实现：**Java 提供一个便捷的动态代理**接口 InvocationHandler** ,实现这个接口需要重写他的调用方法 invoke()，这个方法是proxy代理实例的调用处理程序。每一个proxy代理实例都有一个关联的调用处理程序；代理实例调用方法时，方法调用被编码分派到调用处理程序的invoke方法。**Proxy类**就是用来创建一个代理对象的类，它提供了很多方法，最常用的是newProxyInstance方法，

**Cglib动态代理(基于类)**：针对类实现代理，使用的继承机制，也就是被代理类和代理类是继承关系，底层利用ASM(Java字节码处理框架)把代理对象类的class文件加载进来，通过修改它的字节码生成子类再用方法拦截技术拦截所有父类方法的调用并覆盖其中的方法来处理。也就是目标对象通过继承这个类来重构这个类的方法。所以代理类是可以赋值给被代理类的，如果被代理类有接口，那么代理类也可以赋值给接口

**区别/比较：**JDK代理用的是反射机制，CGLIB代理用字节码处理框架ASM，通过修改字节码生成子类。所以**JDK动态代理的方式创建代理对象效率高，执行效率较低，cglib创建效率低，执行效率高**；



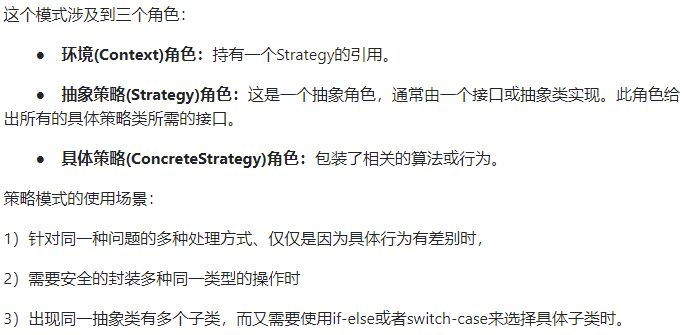
**例子：**论坛权限控制代理 在一个论坛中已注册用户和游客的权限不同，已注册的用户拥有发帖、修改自己的注册信息、修改自己的帖子等功能；而游客只能看到别人发的帖子，没有其他权限。使用代理模式来设计这个权限管理模块。也就是用代理模式中的保护代理(用于控制对一个对象的访问)给不同的用户提供不同级别的使用权限

**五、观察者模式(行为型)**：用于在易用和低耦合下实现一个对象改变给其他对象通知的功能。用于消息队列

**优点**：观察者和被观察抽象耦合的

**缺点**：当观察者较多时，完成通知耗时较长，观察者和被观察是循环依赖时，会导致循环调用，可能会使导致系统崩溃

**六、策略模式(行为型)：**针对一组算法，把每一个算法封装到具有共同接口的独立的类里，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化



**七、访问者模式(行为型)：**一种把数据操作和数据结构分离的设计模式

**大概**是：相同的数据对象根据不同的visitor产生不同的行为操作。也就是在结构不变的情况下实现动态的改变内部元素的逻辑动作。核心是多态即面向抽象编程

**具体**：软件系统里有一个由许多对象构成的、比较稳定的对象结构，这些对象的类都拥有一个 accept 方法用来接受访问者对象的访问。访问者是一个接口，它有一个 visit 方法，这个方法对访问到的对象结构中不同类型的元素做出不同的处理。在对象结构的一次访问过程中，我们遍历整个对象结构，对每一个元素都实施 accept 方法，在每一个元素的 accept 方法中会调用访问者的 visit 方法，从而使访问者可以处理对象结构的每一个元素，我们可以针对对象结构设计不同的访问者类来完成不同的操作，达到区别对待的效果。

**八、命令模式(行为型)**把发送者、接收者和调用命令封装成独立的对象给客户端调用，并且提供命令的撤销和恢复功能

**应用**：Spring 框架中的 JdbcTemplate

**优点**：降低了系统的耦合度，新增的命令可以很容易地添加到系统中

**缺点**：如果命令很多就会造成命令类的代码很长，增加了维护的复杂性

**九、模板模式(行为型)**：定义一个操作中算法的骨架，把一些步骤延迟到子类，这使得子类不用改变一个算法的结构就能重新定义这个算法的一些步骤

**应用**：一个操作步骤稳定，而具体细节改变延迟的子类，比如restTemplate

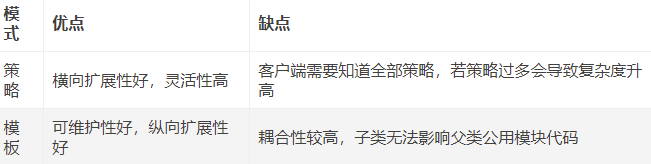
**模板模式 vs 策略模式**

**模板模式**：由抽象类来定义一个算法，在算法实现的不同步骤上抽象方法由子类继承并提供具体实现，常见的就是不同步骤提供doXXX抽象方法留给子类实现。模板模式一般有两部分组成，即抽象模板和具体模板。

**策略模式：**以接口形式提供抽象接口。由具体实现类提供不同算法。策略模式一般由3部分组成：1.一个Context持有所有策略实现类引用，提供给客户端运行

2.一个策略接口提供 3.具体的策略实现类

**区别**：模板模式一般只针对一套算法，注重对同一个算法的不同细节进行抽象提供不同的实现。而策略模式注重多套算法多套实现，在算法中间不应该有交集，因此算法和算法之间一般不会有冗余代码！因为不同算法只间的实现一般不同很相近。所以策略模式的关注点更广，模板模式的关注点更深。而且两种模式可以一起使用，即具体某个策略下可以通过模板减少不同步骤的冗余代码



**生产者消费者**：

1. 当队列为空时，消费者线程阻塞，否则唤醒消费者线程

2. 当队列为满时，生产者线程阻塞，否则唤醒生成者线程

因为消费者只从队列里面拿数据，用take方法

而生产者只放东西到队列，用put方法，这两个方法是独立的

**Java SE vs Java EE vs Java ME**

1.java SE个人计算机应用 eclipse里用

2.java EE是java企业版，用于服务端应用，提供web服务，组建模型和管理和通信API，可以用来实现企业级的面向服务体系结构

3.java ME—微型版，用于移动产品和车载产品，给设备提供灵活的界面和健壮的安全模式。jvm把.java文件编译成字节码文件.class，之后不同的jvm都可以解析.class文件来在不同OS上执行

**哈希碰撞**：两个不同的输入产生一个相同的hash值。

**应用**：分布式系统里的负载均衡和数据分布

**哈希2种使用**：

**1.哈希取模**：有5个服务器做负载均衡，然后我们就对请求的ip和用户id用哈希函数，然后把计算出的hash值对5取模，根据余数来分配到对应的服务器上，也就是用节点数量作为除数

缺点：有状态的服务下，新增一个服务器会导致大多数请求要重新映射到别的节点，在web负载均衡的情况下，session会存在每个节点里，如果增删节点会让几乎所有数据要迁移

**2.一致性哈希**：用2^32作为除数. 也就是0-2^32首尾连成一个环。先对服务器节点的IP哈希，然后除以2^32，有请求来了，同样哈希然后除以2^32, 然后在环上顺时针找到第一个节点，这个节点就负责这个请求

**优点**：不会因为横向的伸缩导致数据大规模变动

**缺点**：在节点数量少的时候，会出现分布不均匀的情况(解决方案是在环上加虚拟节点)

**哈希表特点**：关键字在表中位置和它之间存在一种确定的关系

**哈希冲突**：把key哈希后的结果作为地址去放键值对的时候(hashmap)，但是地址上已经有别的键值对了

**处理/解决哈希冲突/溢出**：

**1**.**开发地址法**：哈希冲突时候，用探测技术在哈希表里得到一个探测序列。按照不同探测序列的形成方法可以分成3种：节点规模小时，再散列法节省空间

**1.线性探查法**：从已存的地址一个个往后找。

缺点：1.处理溢出需要另编程序2.产生堆聚现象(存入哈希表的记录连成一片)，冲突可能性变大。

**2.线性补偿法:**探测的步长数不是1，而是别的数，这个数要和表的长度互质这样才能探测表中所有单元

**3.随机探测**：探测步长是随机数 这样能避免堆聚

**2.拉链法**：把哈希表变成指针数组，每个单元都是一个头指针，如果键值对有一样的散列地址就插到这个单元的链表里

**优点**：1.不容易有堆聚现象，平均查找长度短 2.装载因子大，节省空间 3.删节点简单因为不用像再散列法不能直接把节点置位空 因为会截断查找路径

**负载因子**：存储数据的长度/创建hashmap的长度

**缺点**：指针要额外空间，节点规模小时，再散列法节省空间

**3.再哈希法**

用哈希函数去散列一个输入的时候，输出是同一个位置就再次哈希，直至不发生冲突位置。 缺点：每次冲突都要重新哈希，计算时间增加

**哈希函数构造方法**：

1.数字分析法：知道关键字集合的话，从中选出分布均匀的几位构成哈希地址

2.平均取中法：求关键字的平方值, 然后取平方值的中间几位作为哈希地址

3.分段叠加法：按哈希表地址位数把关键字分成位数相等的几部分然后相加，舍弃最高进位后的结果就是这个关键字的哈希地址

4.除留余数法：哈希函数key 对小于等于哈希表长度的最大素数取模

**查找两个大(内存放不下)文件(每行都是一个字符串)里的相同的字符串：hash+分治**

1.用hash算法对两个文件分别进行hash成a，b个小文件

2.因为hash的缘故，相同的字符串肯定在同个文件对里面。所以比较这些小文件对<a1,b1>.........

3.统计小文件对，可以继续采用hash，对a1的每一字符串建立hash表，遍历b1的字符串看是否在之前构建的hash表里面（和上面一样）

   一般来说，如果内存可以存放，可以构造hash表，进行查找。如果内存无法加载，那么可以通过hash把大文件分成多个小文件，从而进行比较。

 hash算法在海量数据中的运用：

单机处理大数据的问题也和mapreduce一样，分而治之，把海量数据切分成若干个小份进行处理。

    1）分而治之

         采用hash进行取模进行等价映射，将巨大的文件进行等价分割（符合一定规律的数据会被划分到同一个文件），划分成若干个小文件再进行处理。

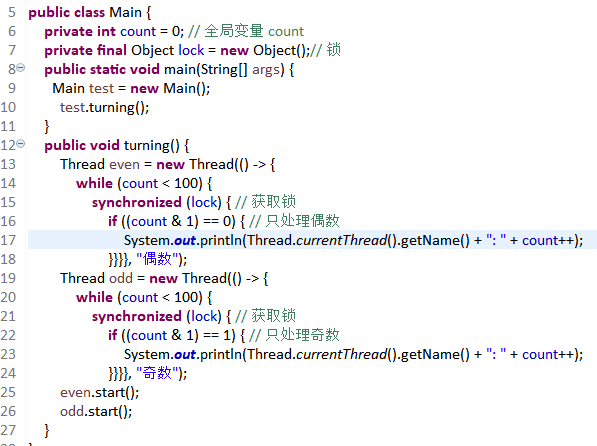
    2）利用hashmap进行内存统计

　　　　利用hashMap对小文件里面的数据进行统计

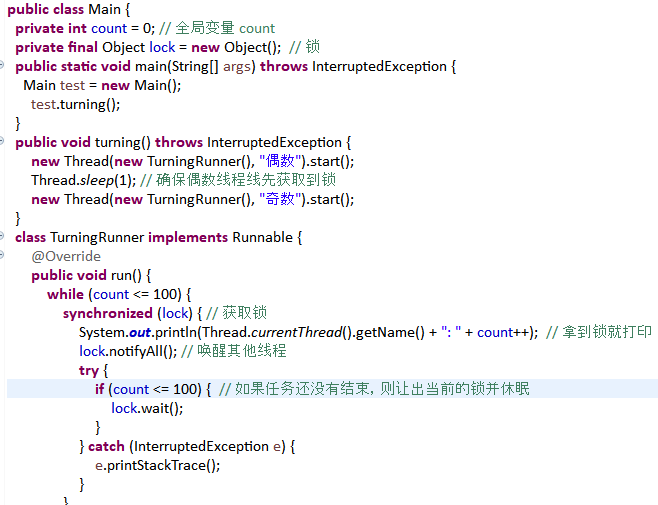
    3）排序

**线程交替打印**

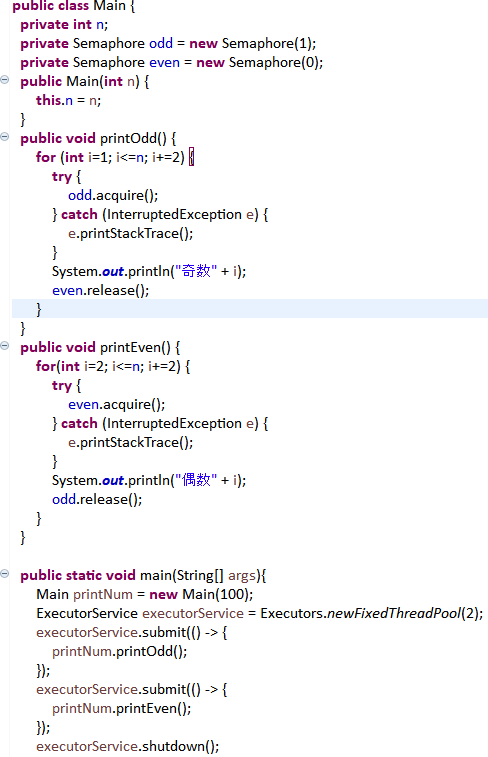
**方法1：**有一个全局变量 count，当 count 等于偶数的时候，我们就让线程1打印，当 count 等于奇数的时候，我们就让线程2打印。实现：用锁来保证同一时刻只有一个线程在执行。但是如果同一个线程一直抢到锁，而另一个线程一直没有拿到，就会导致线程做很多无谓的空转，效率非常低下



**方法2：**原理：线程1打印之后唤醒其他线程，然后让出锁，自己进入休眠状态。因为进入了休眠状态就不会与其他线程抢锁，此时只有线程2在获取锁，所以线程2必然会拿到锁。线程2用同样的逻辑执行，唤醒线程1并且让出自己持有的锁，自己进入休眠状态。这样来来回回，持续执行直到任务完成。就达到了两个线程交替获取锁的效果了。



**方法3：信号量**



**方法4：**用condition实现**3个线程交替打印**



