# 电话面试

# 1 JUC

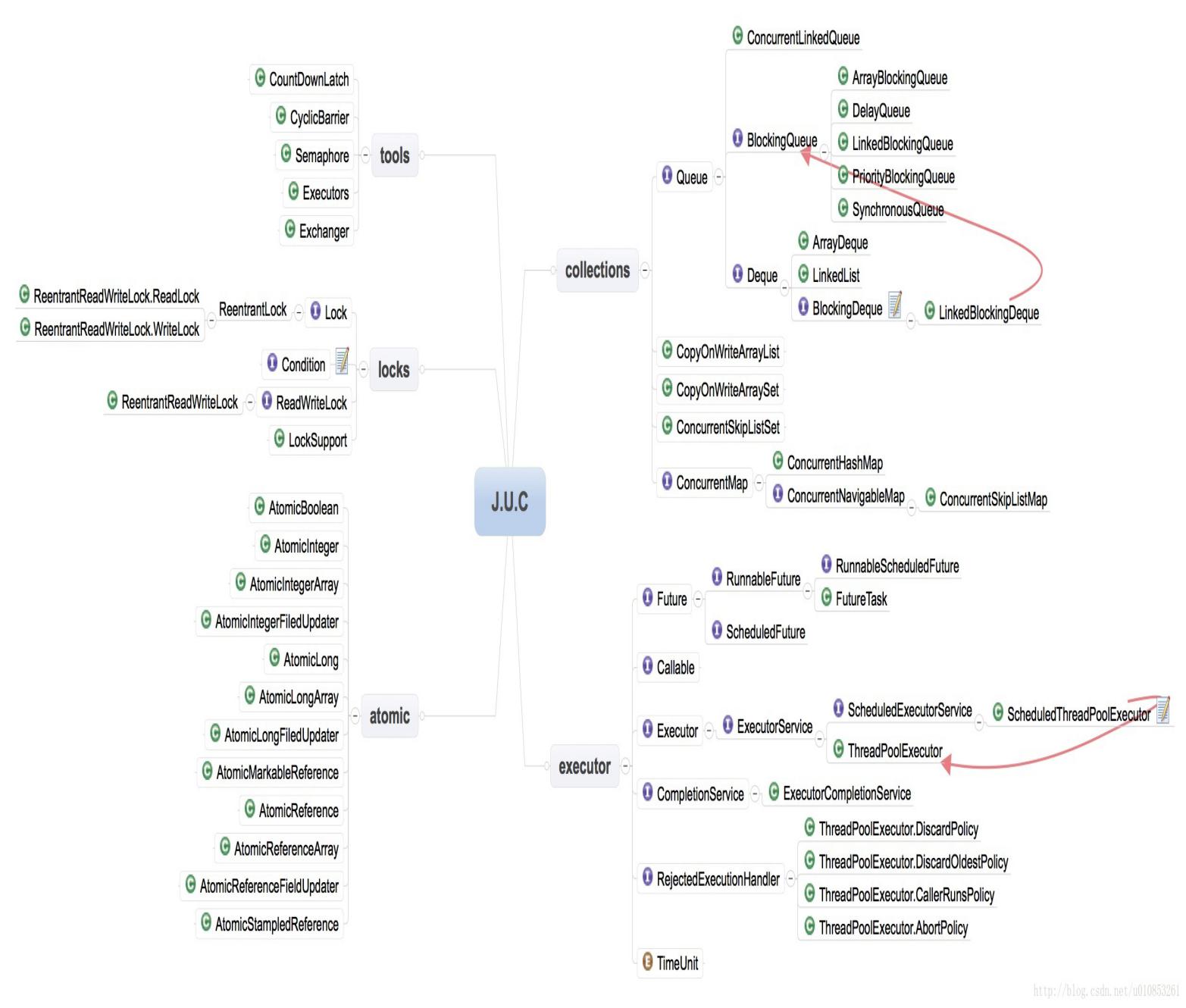
J.U.C 包下的并发类并发工具类

显示锁

原子变量类

并发集合

Executor线程执行器



# 2 多线程的东西同步机制

## 1.多线程有什么用？

1发挥多核CPU的优势，充分“压榨”CPU的性能

2防止阻塞。

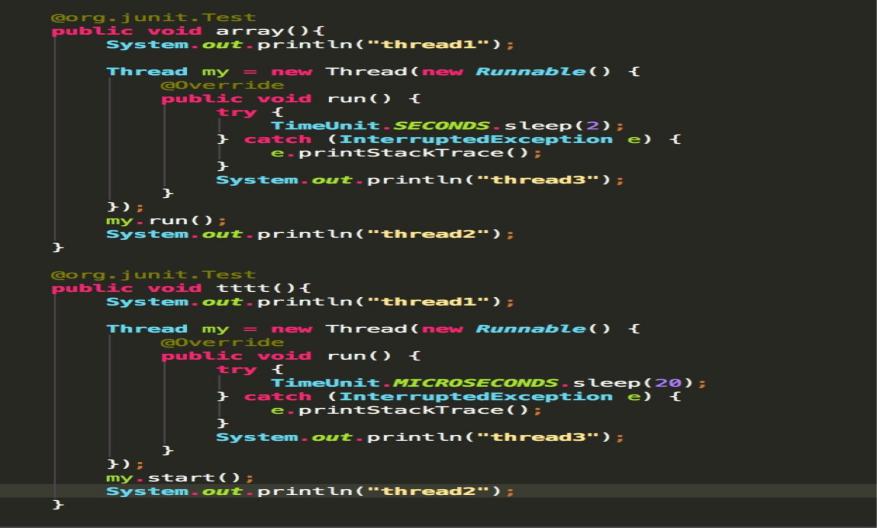
3便于建模，比如一个大的任务A，单线程完成要考虑很多，如果分成很多小任务，多线程来完成就简单很多了。

## 2.创建线程的方式

一般有三种：   
1）继承Thread类   
2）实现Runnable接口   
3）实现Callable接口   
一般来说我都是通过实现Runnable接口创建线程，也觉得这种方式好一些，毕竟Java一直提倡面向接口编程。

## 3.start()方法和run()方法的区别（unkonw）

start()是Thread里面的方法，在调用start()方法会立刻启动线程，然后会自动的调用run()方法，但是调用start()方法会立即返回，不会阻塞调用线程，从而真正实现多线程。   
1）start() 方法来启动线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码；通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。   
2）run（）方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到多线程的目的。下图，第一个是132，第二个是123



## 4.Runnable接口和Callable接口（concurrent）的区别

1）Runnable接口里面的run()方法是没有返回值的，所以实现Runnable接口一般是执行不带返回值的任务。   
2）Callable接口里面的call()方法是有返回值的，返回值是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

有没有返回值也是这两个接口的主要区别，Callable接口的功能更加强大，使用Callable+Future/FutureTask可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，非常有用。

## 5.CyclicBarrier和CountDownLatch的区别 concurrent

这两个类都是J.U.C 并发包下面非常有用的类：   
1）CountDownLatch(闭锁)相当于一扇门：在闭锁到达结束状态之前（闭锁的结束状态也就是闭锁的计数器减到零），这扇门一直是关闭的，并且没有任何线程可以通过；当闭锁到达结束状态时，这扇门会打开并允许所有的等待线程(在闭锁上调用了await()方法的线程)通过。并且当闭锁到达结束状态后不可逆转，这扇门会一直保持打开的状态。应用场景：确保某些活动直到其他活动都结束了才继续执行。

2）CyclicBarrier 栏珊，它允许一组线程互相等待，直到都到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。CyclicBarrier就象它名字的意思一样，可看成是个障碍， 所有的线程必须到齐后才能一起通过这个障碍。

## 6.volatile关键字的作用

提供了可视性机制。volatile关键字的作用主要：  
可视性：多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据。  
在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制。而声明变量是volatile的，JVM保证了每次读变量都从内存中读，跳过CPU cache这一步。

volatile的一个重要作用就是和CAS结合，保证了原子性与可视性，详细的可以参见java.util.concurrent.atomic包下的类，比如AtomicInteger。

Java语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。

## 7.什么是CAS

CAS，全称为Compare and Swap，即比较-替换。假设有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。

比如原子类：AtomicInteger 就是volatile+CAS实现的。getAndIncrease

## 8.一个线程如果出现了运行时异常会怎么样

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象锁会被立即释放

## 9.如何在两个线程之间共享数据

通过线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，这里需要注意的是要保证多线程环境下的数据安全性。比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的，

## 10.sleep方法和wait方法有什么区别

相同点：sleep方法和wait方法都可以用来放弃CPU一定的时间。   
不同点：在于如果线程持有某个对象的锁，sleep方法不会放弃这个对象的锁；wait方法会放弃这个对象的锁。

## 11.生产者消费者模型的作用是什么

1）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约。   
2）通过平衡生产者的生产能力 和 消费者的消费能力来提升整个系统的运行效率,这是生产者消费者模型最重要的作用。

## 12.ThreadLocal有什么用

简单说ThreadLocal就是一种以 空间换时间 的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了。

ThreadLocal是为解决多线程环境下变量的并发访问提出的另外一种思路（相对于加锁来说）。当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。 该作用也是ThreadLocal所表达的含义，就是线程私有的变量。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单：在ThreadLocal类中有一个虚拟的Map（这里之所以说是一个虚拟的Map是因为这个Map本身不存在，只是对于每个线程Thread对象里面有一个ThreadLocalMap对象），ThreadLocalMap用于存储每一个线程的变量副本。虚拟Map中元素的键为线程对象，而值对应线程的变量副本ThreadLocalMap中的值。

<http://blog.csdn.net/u010853261/article/details/55105173>

## 13.为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁。

## 14.wait()方法和notify()/notifyAll()方法在放弃对象锁时有什么区别

wait()方法立即释放对象锁，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象的锁。

notify是随意唤醒一个线程。

yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

个线程从wait()状态醒来是不是一定被notify()了？

**当在obj对象上调用wait操作的时候，就会释放当前持有的锁，并将线程加入到obj所属的条件队列**，而后阻塞，直到有其它线程在该obj上调用了notify操作或阻塞线程被中断或wait超时。

## 15.为什么要使用 线程池

1）避免频繁地创建和销毁线程，达到线程对象的重用。创建和销毁线程的开销还是很大的）  
2）使用线程池还可以根据项目灵活地控制并发的数目。

线程池和阻塞队列

<http://blog.csdn.net/bug_moving/article/details/56003645>

## 16.怎么检测一个线程是否持有对象监视器

有方法可以判断某个线程是否持有对象的锁：Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着”某条线程”指的是当前线程。

## 17.synchronized和ReentrantLock（重入锁）的区别

（重入性）所以在**[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "http://blog.csdn.net/mark_lq/article/details/_blank)**内部，同一线程在调用自己类中其他synchronized方法/块或调用父类的synchronized方法/块都不会阻碍该线程的执行，就是说同一线程对同一个对象锁是可重入的，而且同一个线程可以获取同一把锁多次，也就是可以多次重入。

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：   
（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁   
（2）ReentrantLock可以获取各种锁的信息   
（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的是对象头中mark word。

## \* Synchronized Lock 内部实现

数据同步需要依赖锁，那锁的同步又依赖谁？synchronized给出的答案是在软件层面依赖JVM，而Lock给出的方案是在硬件层面依赖特殊的CPU指令，大家可能会进一步追问：JVM底层又是如何实现synchronized的？

synchronized的底层实现主要依靠Lock-Free的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。

<http://www.open-open.com/lib/view/open1352431526366.html>

大湿的回答：可以先说重量级锁，也就是用系统提供的mutex来实现，竞争失败的就放到等待队列里，由mutex来搞唤醒的操作。这个等待队列不知道是不是无锁的。然后轻量级锁，就是cas修改对象头状态，标记这个对象被我这个线程锁住了，解锁的时候再改回来，要是发生竞争，自旋几次后，轻量级锁升级为重量级锁。最后是偏向锁，也是cas修改对象头状态，但是解锁了也不改回来，要是发生竞争就比较麻烦，得遍历相关的栈帧，如果没有持有锁，就重偏向给自己。要是正在持有锁，就得膨胀成轻量级锁还是重量级锁。

## ****\* synchronized****

锁的优化：偏向锁、轻量级锁、自旋锁、重量级锁

锁的膨胀模型，以及锁的优化原理，为什么要这样设计

与Concurrent包下的Lock的区别和联系

Lock能够实现synchronized的所有功能，同时，能够实现长时间请求不到锁时自动放弃、通过构造方法实现公平锁、出现异常时synchronized会由JVM自动释放，而Lock必须手动释放，因此我们需要把unLock()方法放在finally{}语句块中

****锁的优化策略****

① 读写分离

② 分段加锁

③ 减少锁持有的时间

④ 多个线程尽量以相同的顺序去获取资源

等等，这些都不是绝对原则，都要根据情况，比如不能将锁的粒度过于细化，不然可能会出现线程的加锁和释放次数过多，反而效率不如一次加一把大锁。这部分跟面试官谈了很久

## 18.隐式(Synchronized)锁跟显式(java.util.Lock)锁

显式和隐式锁都能实现对共享资源的控制,两者在内存同步上是同样的机制,但是显式锁提供了更灵活更强大的接口  
 1.synchronized 对多个锁只能按照获得锁的顺序的反序释放(先获得后释放),显式锁可以   
     按照需要释放锁,无此约束.  
  2.显式锁提供可中断的获取锁的方法,lockInterruptibly  
  3.显式提供尝试获得锁方法  
  4.提供精度更细的等待与唤醒(利用Condition)  
  特别注意显式锁的 xx.lock()方法只是获取了xx对象表达的锁,并不是获取了xx内置的隐式锁,这个要注意区分,与synchronized(xx)是两回事

## 19.什么是乐观锁和悲观锁

其实这两种锁就是对线程安全的最悲观和最乐观的假设。   
（1）乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

1. 悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

•如果对读的响应度要求非常高，比如证券交易系统，那么适合用乐观锁，因为悲观锁会阻塞读

•如果读远多于写，那么也适合用乐观锁，因为用悲观锁会导致大量读被少量的写阻塞

•如果写操作频繁并且冲突比例很高，那么适合用悲观写独占锁

## 20.ReadWriteLock是什么？

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

## 21.Java编程写一个会导致死锁的程序

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。

## 22.怎么唤醒一个阻塞的线程

如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "http://blog.csdn.net/u010853261/article/details/_blank)实现的，Java代码并没有办法直接接触到操作系统。

## 23.不可变对象对多线程有什么帮助

不可变对象保证了对象的内存可见性，而且不可变对象天生就是线程安全的。

## 24.什么是多线程的上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程，这其中涉及到运行环境的保存与切换，存在时间开销。

## 25.如果你提交任务时，线程池队列已满，这时会发生什么

1）如果使用的是无界队列，比如LinkedBlockingQueue，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务；  
2）如果使用的是有界队列。比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

## 26. Java中用到的线程调度算法是什么

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

## 27. Thread.sleep(0)的作用是什么

简单点说就是主动让出一次CPU：  
由于Java采用抢占式的线程调度[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/u010853261/article/details/_blank)，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

## 28.什么是自旋

很多synchronized里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然synchronized里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

## 29.什么是Java内存模型

Java内存模型定义了一种多线程访问Java内存的规范。Java内存模型要完整讲不是这里几句话能说清楚的，我简单总结一下Java内存模型的几部分内容：

（1）Java内存模型将内存分为了主内存和工作内存。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去

（2）定义了几个原子操作，用于操作主内存和工作内存中的变量

（3）定义了volatile变量的使用规则

（4）happens-before，即先行发生原则，定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的

## 30.什么是线程安全

就我的理解来说：一段代码在单线程和多线程并发下永远都能获得一样的结果，那么就是线程安全的。

有值得一提的地方，就是线程安全也是有几个级别的：   
（1）不可变(对象)   
像String、Integer、Long这些，都是final类型的类，还有final类型的简单变量。任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用。   
（2）绝对线程安全   
不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet   
（3）相对线程安全   
相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出ConcurrentModificationException，也就是fail-fast机制。   
（4）线程非安全   
这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类，没有加任何保证线程安全的措施。

## 31. FutureTask是什么

这个其实前面有提到过，FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

## 32.单例模式的线程安全性

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法，我总结一下：   
（1）饿汉式单例模式的写法：线程安全   
（2）懒汉式单例模式的写法：非线程安全   
（3）双检锁(DCL)单例模式的写法：线程安全

## 33.Semaphore有什么作用 concurrent

常用作并发数控制器。Semaphore就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

## 34.Hashtable的size()方法中明明只有一条语句”return count”，为什么还要做同步？

这是我之前的一个困惑，不知道大家有没有想过这个问题。某个方法中如果有多条语句，并且都在操作同一个类变量，那么在多线程环境下不加锁，势必会引发线程安全问题，这很好理解，但是size()方法明明只有一条语句，为什么还要加锁？

关于这个问题，在慢慢地工作、学习中，有了理解，主要原因有两点：  
（1）同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程A在执行Hashtable的put方法添加数据，线程B则可以正常调用size()方法读取Hashtable中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对size++，线程B就已经读取size了，那么对于线程B来说读取到的size一定是不准确的。而给size()方法加了同步之后，意味着线程B调用size()方法只有在线程A调用put方法完毕之后才可以调用，这样就保证了线程安全性

（2）CPU执行代码，执行的不是Java代码，这点很关键，一定得记住。Java代码最终是被翻译成汇编代码执行的，汇编代码才是真正可以和硬件电路交互的代码。即使你看到Java代码只有一行，甚至你看到Java代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个。一句”return count”假设被翻译成了三句汇编语句执行，完全可能执行完第一句，线程就切换了。

## 35.线程类的构造方法、静态块是被哪个线程调用的？

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被new这个线程类所在的线程所调用的，而run方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设Thread2中new了Thread1，main函数中new了Thread2，那么：   
（1）Thread2的构造方法、静态块是main线程调用的，Thread2的run()方法是Thread2自己调用的

（2）Thread1的构造方法、静态块是Thread2调用的，Thread1的run()方法是Thread1自己调用的

## 36.同步方法和同步块，哪个是更好的选择？

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：同步的范围越小越好。

借着这一条，我额外提一点，虽说同步的范围越少越好，但是在Java虚拟机中还是存在着一种叫做锁粗化的优化方法，这种方法就是把同步范围变大。这是有用的，比方说StringBuffer，它是一个线程安全的类，自然最常用的append()方法是一个同步方法，我们写代码的时候会反复append字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着Java虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此Java虚拟机会将多次append方法调用的代码进行一个锁粗化的操作，将多次的append的操作扩展到append方法的头尾，变成一个大的同步块，这样就减少了加锁–>解锁的次数，有效地提升了代码执行的效率。

## 37.什么是AQS？（\*）

简单说一下AQS，AQS全称为AbstractQueuedSychronizer，翻译过来应该是抽象队列同步器。

如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。

AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

## 38.高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：   
（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换

（2）并发不高、任务执行时间长的业务要区分开看：   
　　a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务

　　b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "http://blog.csdn.net/u010853261/article/details/_blank)的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

## 39.wait方法是否必须用在while循环体中

是，《Effective Java》第二版中文版第69条244页位置对这一点说了一页，我看着一知半解。我能理解的一点是：对于从wait中被notify的进程来说，它在被notify之后还需要重新检查是否符合执行条件，如果不符合，就必须再次被wait，如果符合才能往下执行。所以：wait方法应该使用循环模式来调用。按照上面的生产者和消费者问题来说：错误情况一：如果有两个生产者A和B，一个消费者C。当存储空间满了之后，生产者A和B都被wait，进入等待唤醒队列。当消费者C取走了一个数据后，如果调用了notifyAll（），注意，此处是调用notifyAll（），则生产者线程A和B都将被唤醒，如果此时A和B中的wait不在while循环中而是在if中，则A和B就不会再次判断是否符合执行条件，都将直接执行wait（）之后的程序，那么如果A放入了一个数据至存储空间，则此时存储空间已经满了；但是B还是会继续往存储空间里放数据，错误便产生了。错误情况二：如果有两个生产者A和B，一个消费者C。当存储空间满了之后，生产者A和B都被wait，进入等待唤醒队列。当消费者C取走了一个数据后，如果调用了notify（），则A和B中的一个将被唤醒，假设A被唤醒，则A向存储空间放入了一个数据，至此空间就满了。A执行了notify（）之后，如果唤醒了B，那么B不会再次判断是否符合执行条件，将直接执行wait（）之后的程序，这样就导致向已经满了数据存储区中再次放入数据。错误产生。

## 40.Runnable 和 Thread的区别

在程序开发中只要是多线程肯定永远以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类有如下好处：

* 避免点继承的局限，一个类可以继承多个接口。
* 适合于资源的共享

以卖票程序为例，通过Thread类完成要卖出30张票。虽然现在程序中有三个线程，但是一共卖了10张票，也就是说使用Runnable实现多线程可以达到资源共享目的。

## 41.Servlet是线程安全的吗？

Servlet不是线程安全的。

要解释为什么Servlet为什么不是线程安全的，需要了解Servlet容器（即Tomcat）使如何响应HTTP请求的。

当Tomcat接收到Client的HTTP请求时，Tomcat从线程池中取出一个线程，之后找到该请求对应的Servlet对象。如果该Servlet还未被请求过，那么将进行Servlet初始化并调用Servlet并调用service()方法。否则，直接调用service()方法。要注意的是每一个Servlet对象再Tomcat容器中只有一个实例对象，即是单例模式。如果多个HTTP请求请求的是同一个Servlet，那么着两个HTTP请求对应的线程将并发调用Servlet的service()方法。

这时候，如果在Servlet中定义了实例变量或静态变量，那么可能会发生线程安全问题（因为所有的线程都可能使用这些变量）。

## 42.condition 实现控制多线程执行顺序 [Condition-线程通信更高效的方式](http://blog.csdn.net/ghsau/article/details/7481142)

在Condition中，用await()替换wait()，用signal()替换notify()，用signalAll()替换notifyAll()，传统线程的通信方式，Condition都可以实现，这里注意，Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法。

## 43 [多线程程序设计的8个规则](http://blog.csdn.net/tang_huan_11/article/details/49490783)

规则一：找到真正不相关的计算任务

规则二：尽可能地在最高层进行并行化，一个是自底向上，另一个是自顶向下

规则三：尽早针对众核趋势做好可伸缩性的规划

规则四：尽可能利用已有的线程安全库

规则五：使用合适的多线程模型

规则六：永远不要假设具体的执行顺序

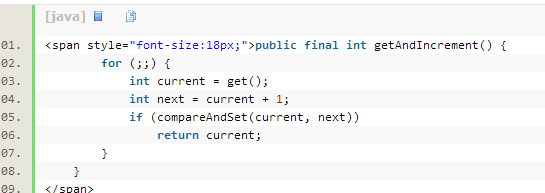
规则七：尽可能使用线程本地存储或者对特定的数据加锁

规则八：敢于更换更易并行化的[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/tang_huan_11/article/details/_blank)

## 44 atomicinteger addandget和getandadd

AtomicInteger，一个提供原子操作的Integer的类。在Java语言中，++i和i++操作并不是线程安全的。在使用的时候，不可避免的会用到synchronized关键字。而AtomicInteger则通过一种线程安全的加减操作接口。

AtomicInteger并没有使用Synchronized关键字实现原子性，几乎所有的数据更新都用到了compareAndSet(int expect, int update)这个方法。那么就不难看出AtomicInteger这个类的最核心的函数就是compareAndSet(int expect, int update)。



单看这段 代码 很难保证原子性， 因为根本没有更新value 的操作

重点在于compareAndSet() 函数

public final boolean compareAndSet(int expect,int update)

如果当前值 == 预期值，则以原子方式将 当前值 设置为给定的更新值。

参数：

expect - 预期值

update - 新值

返回：

如果成功，则返回 true。返回 False 指示实际值与预期值不相等。

    该函数 只有两个参数，可操作的确实三个值 ，即 value ,expect, update. 他使用了 由硬件保证其原子性的指令 CAS （compare and swap）。

    compareAndSet  函数保证了 比较，赋值这两步操作可以通过一个原子操作完成。

然后看整个函数， 所有代码被放到了一个循环里面， 如果compareAndSet（）执行失败，则说明 在int current = get(); 后，其他线程对value进行了更新， 于是就循环一次，重新获取当前值，直到compareAndSet（）执行成功为止。

这里需要注意的是AtomicInteger所利用的是基于冲突检测的乐观并发策略（CAS自旋锁）。 所以这种乐观在线程数目非常多的情况下，失败的概率会指数型增加。

# 3 计算机网络

## 1 TCP的三次握手和四次挥手

建立TCP需要三次握手才能建立，而断开连接则需要四次握手。  
首先Client端发送连接请求报文，Server段接受连接后回复ACK报文，并为这次连接分配资源。Client端接收到ACK报文后也向Server段发生ACK报文，并分配资源，这样TCP连接就建立了。

**【注意】中断连接端可以是Client端，也可以是Server端。**

假设Client端发起中断连接请求，也就是发送FIN报文。Server端接到FIN报文后，意思是说"我Client端没有数据要发给你了"，但是如果你还有数据没有发送完成，则不必急着关闭Socket，可以继续发送数据。所以你先发送ACK，"告诉Client端，你的请求我收到了，但是我还没准备好，请继续你等我的消息"。这个时候Client端就进入FIN\_WAIT状态，继续等待Server端的FIN报文。当Server端确定数据已发送完成，则向Client端发送FIN报文，"告诉Client端，好了，我这边数据发完了，准备好关闭连接了"。Client端收到FIN报文后，"就知道可以关闭连接了，但是他还是不相信网络，怕Server端不知道要关闭，所以发送ACK后进入TIME\_WAIT状态，如果Server端没有收到ACK则可以重传。“，Server端收到ACK后，"就知道可以断开连接了"。Client端等待了2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，我Client端也可以关闭连接了。Ok，TCP连接就这样关闭了！

**【问题1】为什么连接的时候是三次握手，关闭的时候却是四次握手？**  
答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

**【问题2】为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？**

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。

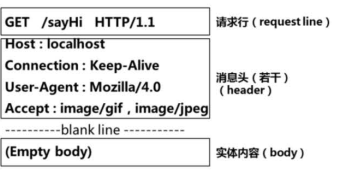
## 2 HTTP协议

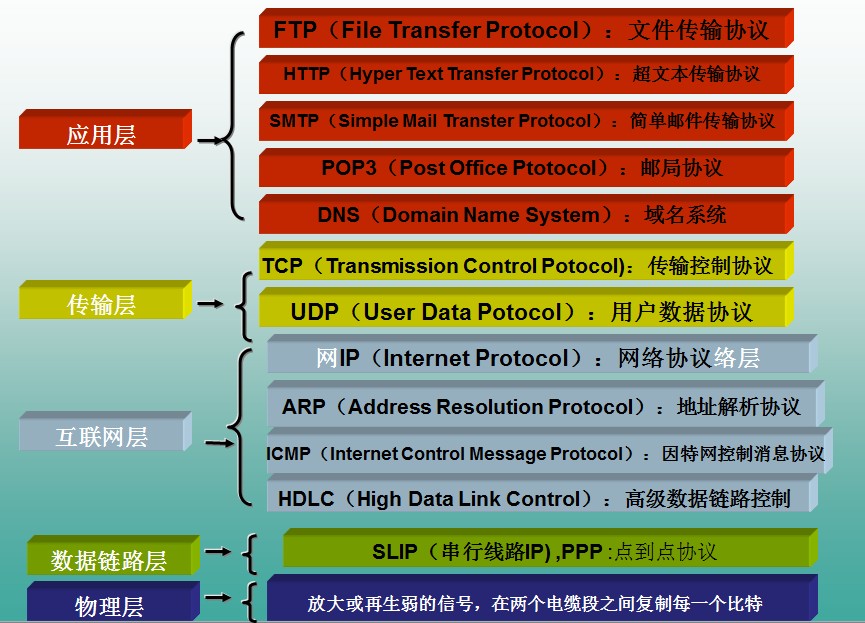
常见状态代码、状态描述、说明：  
200 OK      //客户端请求成功  
400 Bad Request  //客户端请求有语法错误，不能被服务器所理解  
401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用   
403 Forbidden  //服务器收到请求，但是拒绝提供服务  
404 Not Found  //请求资源不存在，eg：输入了错误的URL  
500 Internal Server Error //服务器发生不可预期的错误  
503 Server Unavailable  //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

 http请求由三部分组成，分别是：请求行、消息报头、请求正文

HTTP请求包括三部分：请求行(Request Line)，头部（Headers）和实体内容（Body)。其中，请求行由请求方法(method)，请求网址Request-URI和协议 (Protocol)构成，而消息头包括多个属性，实体内容（数据体）则可以被认为是附加在请求之后的文本或二进制文件，只有请求方式为post的时候，实体内容才会有数据（即请求参数）。

请求头和请求体之间用什么分割，当时回答是两个回车换行，后来查了一下原来是一个回车换行





## 3 假如服务端知道 客户端IP（假设全球IP） 问服务端能否向 客户端发消息。

答案是不能，客户端接受消息是通过从服务器拉数据的形式来获取数据而不是服务器向客户端取数据。客户端几乎也是不会监听端口接受服务器发来的消息。

## 4 计算机相互之间的通信

互联网的关键技术就是TCP/IP协议。两台计算机之间的通信是通过TCP/IP协议在因特网上进行的。实际上这个是两个协议：

TCP : Transmission Control Protocol 传输控制协议和IP： Internet Protocol  网际协议。

IP：计算机之间的通信

IP协议是计算机用来相互识别的通信的一种机制，每台计算机都有一个IP.用来在internet上标识这台计算机。  IP 负责在因特网上发送和接收数据包。通过 IP，消息（或者其他数据）被分割为小的独立的包，并通过因特网在计算机之间传送。IP 负责将每个包路由至它的目的地。

IP协议仅仅是允许计算机相互发消息，但它并不检查消息是否以发送的次序到达而且没有损坏（只检查关键的头数据）。为了提供消息检验功能，直接在IP协议上设计了传输控制协议TCP.

TCP : 应用程序之间的通信

TCP确保数据包以正确的次序到达，并且尝试确认数据包的内容没有改变。TCP在IP地址之上引端口（port），它允许计算机通过网络提供各种服务。一些端口号为不同的服务保留，而且这些端口号是众所周知。

服务或者守护进程：在提供服务的机器上，有程序监听特定端口上的通信流。例如大多数电子邮件通信流出现在端口25上，用于wwww的HTTP通信流出现在80端口上。

当应用程序希望通过 TCP 与另一个应用程序通信时，它会发送一个通信请求。这个请求必须被送到一个确切的地址。在双方“握手”之后，TCP 将在两个应用程序之间建立一个全双工 (full-duplex) 的通信，占用两个计算机之间整个的通信线路。TCP 用于从应用程序到网络的数据传输控制。TCP 负责在数据传送之前将它们分割为 IP 包，然后在它们到达的时候将它们重组。

TCP/IP 就是TCP 和 IP 两个协议在一起协同工作，有上下层次的关系。

TCP 负责应用软件（比如你的浏览器）和网络软件之间的通信。IP 负责计算机之间的通信。TCP 负责将数据分割并装入 IP 包，IP 负责将包发送至接受者，传输过程要经IP路由器负责根据通信量、网络中的错误或者其他参数来进行正确地寻址，然后在它们到达的时候重新组合它们。

## 5 HTTP工作过程

一次HTTP操作称为一个事务，其工作整个过程如下：

     1 ) 、地址解析，

     如用客户端浏览器请求这个页面：[http://localhost.com:8080/index.htm](http://localhost:8080/simple.htm" \t "http://blog.csdn.net/zhangliangzi/article/details/_blank)

     从中分解出协议名、主机名、端口、对象路径等部分，对于我们的这个地址，解析得到的结果如下：  
     协议名：http  
     主机名：localhost.com  
     端口：8080  
     对象路径：/index.htm

      在这一步，需要域名系统DNS解析域名localhost.com,得主机的IP地址。

2）封装HTTP请求数据包

     把以上部分结合本机自己的信息，封装成一个HTTP请求数据包  
     3）封装成TCP包，建立TCP连接（TCP的三次握手）

       在HTTP工作开始之前，客户机（Web浏览器）首先要通过网络与服务器建立连接，该连接是通过TCP来完成的，该协议与IP协议共同构建Internet，即著名的TCP/IP协议族，因此Internet又被称作是TCP/IP网络。HTTP是比TCP更高层次的应用层协议，根据规则，只有低层协议建立之后才能，才能进行更层协议的连接，因此，首先要建立TCP连接，一般TCP连接的端口号是80。这里是8080端口

     4）客户机发送请求命令

       建立连接后，客户机发送一个请求给服务器，请求方式的格式为：统一资源标识符（URL）、协议版本号，后边是MIME信息包括请求修饰符、客户机信息和可内容。

     5）服务器响应

     服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是MIME信息包括服务器信息、实体信息和可能的内容。

        实体消息是服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以Content-Type应答头信息所描述的格式发送用户所请求的实际数据

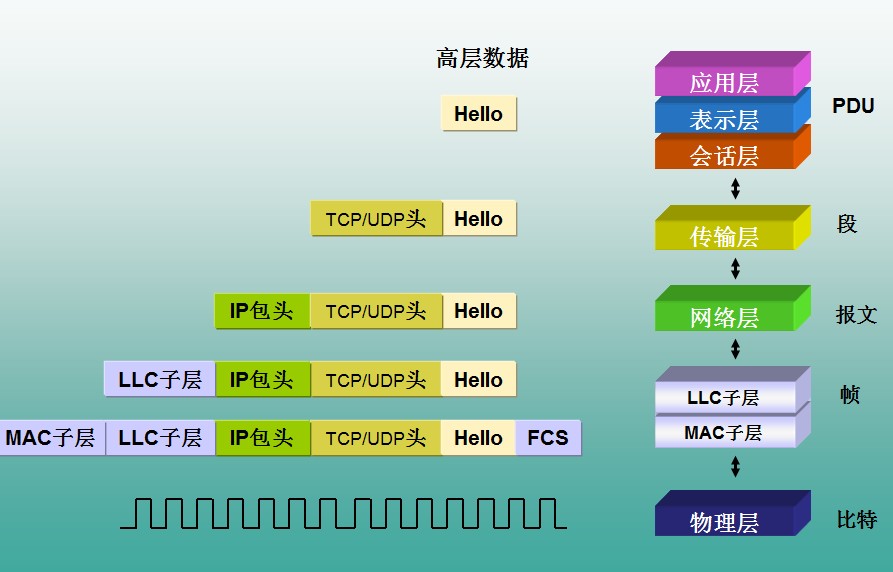
     6）服务器关闭TCP连接

     一般情况下，一旦Web服务器向浏览器发送了请求数据，它就要关闭TCP连接，然后如果浏览器或者服务器在其头信息加入了这行代码

    Connection:keep-alive

   TCP连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

## 6 HTTP协议栈中各层数据流



## 7 GET，POST区别？

基础知识：Http的请求格式如下。

<request line>     主要包含三个信息：1、请求的类型（GET或POST），2、要访问的资源（如\res\img\a.jif），3、Http版本（http/1.1）

<header>                 用来说明服务器要使用的附加信息

<blank line>              这是Http的规定，必须空一行

[<request-body>]      请求的内容数据

区别：

1、Get是从服务器端获取数据，Post则是向服务器端发送数据。

2、在客户端，Get方式通过URL提交数据，在URL地址栏可以看到请求消息，该消息被编码过；Post数据则是放在Html header内提交。

3、对于Get方式，服务器端用Request.QueryString获取变量的值；对用Post方式，服务器端用Request.Form获取提交的数据值。

4、Get方式提交的数据最多1024字节，而Post则没有限制。

5、Get方式提交的参数及参数值会在地址栏显示，不安全，而Post不会，比较安全。

## 8 Session, Cookie区别

1、Session由应用服务器维护的一个服务器端的存储空间；Cookie是客户端的存储空间，由浏览器维护。

2、用户可以通过浏览器设置决定是否保存Cookie，而不能决定是否保存Session，因为Session是由服务器端维护的。

3、Session中保存的是对象，Cookie中保存的是字符串。

4、Session和Cookie不能跨窗口使用，每打开一个浏览器系统会赋予一个SessionID，此时的SessionID不同，若要完成跨浏览器访问数据，可以使用       Application。

5、Session、Cookie都有失效时间，过期后会自动删除，减少系统开销。

## 9 如果有几千个session，怎么提高效率

把session放到 redis 或 memcache 等此类内存缓存中或着把session存储在SSD硬盘上。session对应的文件有一个特点就是小，一般在几KB左右，  
如果session以文件方式存储，如果并发数量级有几千个，此时系统硬盘的随机IO早已成了系统中的最大瓶颈了，因为会话文件是存储在多个小文件中，映射到存储空间不是一段连续的地址范围所以硬盘的随机读取能力显得非常重要，而觉机械硬盘的随机IO一般在100/iops上下，(IOPS (Input/Output Operations Per Second)，即每秒进行读写（I/O）操作的次数)SSD固态硬盘可以达几百至上千,所以在这么高并发读写的情况下如果无条件用SSD固态盘可以把session放到 redis 或 memcache 等内存缓存中，系统对内存的操作又是非常快的，  
只要你的内存足够大，再多session并发速度一样不会慢。

## 10 session是存储在什么地方，以什么形式存储的。

session变量保存在网页服务器中，你不能直接修改，当然，调用程序中的setAttribute()方法当然可以了。cookie存储的可不是具体的数据，要不岂不是太不安全了，谁都可以修改session变量了，网站也毫无安全性可言。实际，在cookie中，存储的是一个sessionId，它标示了一个服务器中的session变量，通过这种方式，服务器就知道你到底是那个session了。顺便说一句，如果客户端不支持cookie,session也是可以实现的，在服务器端通过urlEncoder，可以实现sessionId的传递。所以，记住客户端只存储session标识，实际内容在网页服务器中。

以键值对的方式存储的

## 11 哪些协议是基于TCP，哪些协议是基于UDP的啊

TCP：HTTP,FTP,SMTP,TENET,POP3,Finger,NNTP,IMAP4,  
UDP：BOOTP,DHCP,NTP,TFTP,SNMP  
DNS可以基于TCP，也可以基于UDP~~·

## 12 TCPIP协议栈

标准TCP/IP协议是用于计算机通信的一组协议，通常被称为TCP/IP协议栈，以它为基础组建的互联网是目前国际上规模最大的计算机网络。正因为互联网的广泛应用，使得TCP/IP成为了事实上的网络标准。



（1）网络接口层

　　TCP/IP协议模型的基层，负责数据帧的发送和接收。对应OSI模型中的物理层和数据链路层，是TCP/IP的最底层，不过通常在描述TCP/IP模型时还是会划分具体为物理层(PHY)和数据链路层(MAC)。

（2）网络层

　　通过互联协议将数据包封装成互联网数据包，并运行必要的路由算法。这里有4种互联协议。

　　(a)网际协议IP：负责在主机和网络之间的路径寻址和数据包路由。

　　(b)地址解析协议ARP：获得同一物理网络中的主机硬件地址。

　　(c)网际控制消息协议ICMP：发送消息，并报告有关数据包的传送错误。

　　(d)互联组管理协议IGMP：用来实现本地多路广播路由器报告。

（3）传输层 提供可靠传输 端到端

　　传输协议在主机之间提供通信会话。传输协议的选择根据数据传输方式而定。主要有以下2种传输协议：

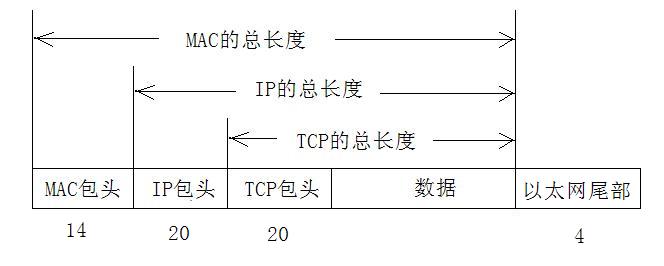
　　(a)传输控制协议TCP：为应用程序提供可靠的通信连接，适用于要求得到响应的应用程序。

　　(b)用户数据包协议UDP：提供无连接通信，且不对传输包进行可靠性确认。

（4）应用层

应用程序通过这一层访问网络，主要包括常见的FTP、HTTP、DNS和TELNET协议。

**TCP/IP协议模型对数据的封装**



## 13 输入www.baidu.com整个过程

1.客户端浏览器通过DNS解析到www.baidu.com的IP地址为220.181.0.1，通过这个ip地址找到客户端到服务器的路径，客户端浏览器发起一个http会话到220.181.0.1，然后通过TCP进行封装数据包，输入到网络层。

2.在客户端的传输层，把HTTP会话请求分成报文段，添加源和目的端口，如服务器端用80端口监听客户端的请求，客户端由系统随机选择一个端口，如5000，与客户端进行交换，服务器把相应的请求返回给客户端的5000端口。然后使用ip层的ip地址查找目的端。

3.客户端的网络层不用关心应用层和传输层的东西，主要做的是通过查找路由表确定如何到达服务器，期间可能经过多个路由器。

4,。客户端的链路层，包通过链路层发送到路由器，通过邻居协议查找给定的ip地址和MAC地址，然后发送ARP请求查找目的地址，如果得到回应后就可以使用ARP的请求应答交换的ip数据包现在就可以传输了，然后发送Ip数据包到达服务器的地址

## 14 浏览器缓存

浏览器缓存就是当你打开一个网页，浏览器会自动下载副本到你电脑上，就相当于自动帮你另存为网页到某个地方而已。一般html，后者request是get请求，而post一般不缓存。当然客户端缓存是否需要是可以在服务端代码上控制的。那就是响应头。响应头告诉缓存器不要保留缓存，缓存器就不会缓存相应内容；如果请求信息是需要认证或者安全加密的，相应内容也不会被缓存；缓存控制头Cache-Control public什么的参数

## 15 上万个用户请求，服务器如何区分

① 服务器在响应头内加上”Set-Cookie:XXXXXXXXXXXXX“(相当于一个唯一的ID符)，此信息是服务器随机生成的，放在服务器内存里，不会重复,这就是sessionid。  
②当浏览器得到这个sessionid会将它放在自己的进程内存里,.然后你继续发请求给这个网站的时候,浏览器就会把这个sessionid放在请求头里发送给该服务器了,这样服务器得到sessionid后再和自己内存里存放的sessionid对比锁定客户端,从而区分不同客户端,完成会话.  
③关闭浏览器结束进程,则这个sessionid将消失,如果用户又打开浏览器想继续这次会话的时候,就会因为发送的请求中没有这个sessionid，而使服务器无法辨别请求身份。

## 16 服务器是怎么判断用户已经登录，比如A登录了，去访问B的博客

# 4 Java基础

## 1 Object类有哪些方法 hashCode equals有哪些关系

protected Object clone()创建并返回此对象的一个副本。   
boolean equals(Object obj)指示其他某个对象是否与此对象“相等”。   
protected void finalize()当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。   
Class<?> getClass()返回此 Object 的运行时类。   
int hashCode()返回该对象的哈希码值。   
void notify()唤醒在此对象监视器上等待的单个线程。   
void notifyAll()唤醒在此对象监视器上等待的所有线程。   
String toString()返回该对象的字符串表示。   
void wait()在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。   
void wait(long timeout)在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。   
void wait(long timeout, int nanos)在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

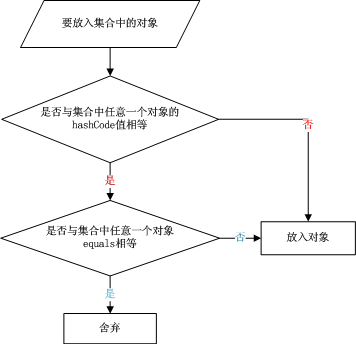
hashCode与equals关系

1、equals方法用于比较对象的内容是否相等（覆盖以后）

2、hashcode方法只有在集合中用到

3、当覆盖了equals方法时，比较对象是否相等将通过覆盖后的equals方法进行比较（判断对象的内容是否相等）。

4、将对象放入到集合中时，首先判断要放入对象的hashcode值与集合中的任意一个元素的hashcode值是否相等，如果不相等直接将该对象放入集合中。如果hashcode值相等，然后再通过equals方法判断要放入对象与集合中的任意一个对象是否相等，如果equals判断不相等，直接将该元素放入到集合中，否则不放入。



## 2 事务的隔离级别

声明式事务：（注解，XML）  
编程式事务：（代码中嵌入）  
隔离级别：  
1、Serializable：最严格的级别，事务串行执行，资源消耗最大；  
2、REPEATABLE READ：保证了一个事务不会修改已经由另一个事务读取但未提交（回滚）的数据。避免了“脏读取”和“不可重复读取”的情况，但是带来了更多的性能损失。  
3、READ COMMITTED:大多数主流数据库的默认事务等级，保证了一个事务不会读到另一个并行事务已修改但未提交的数据，避免了“脏读取”。该级别适用于大多数系统。  
4、Read Uncommitted：保证了读取过程中不会读取到非法数据。  
隔离级别在于处理多事务的并发问题。我们知道并行可以提高数据库的吞吐量和效率，但是并不是所有的并发事务都可以并发运行，这需要查看数据库教材的可串行化条件判断了。  
事务的隔离级别的话，和数据库硬件相关

## 3 java序列化的过程

1、什么是序列化和反序列化  
Serialization（序列化）是一种将对象以一连串的字节描述的过程；反序列化deserialization是一种将这些字节重建成一个对象的过程。

2、什么情况下需要序列化   
a）当你想把的内存中的对象保存到一个文件中或者[数据库](http://www.2cto.com/database/" \t "http://www.2cto.com/kf/201405/_blank)中时候；  
b）当你想用套接字在网络上传送对象的时候；  
c）当你想通过RMI传输对象的时候；

3、如何实现序列化

将需要序列化的类实现Serializable接口就可以了，Serializable接口中没有任何方法，可以理解为一个标记，即表明这个类可以序列化。

4、序列化和反序列化例子

如果我们想要序列化一个对象，首先要创建某些OutputStream(如FileOutputStream、ByteArrayOutputStream等)，然后将这些OutputStream封装在一个ObjectOutputStream中。这时候，只需要调用writeObject()方法就可以将对象序列化，并将其发送给OutputStream（记住：对象的序列化是基于字节的，不能使用Reader和Writer等基于字符的层次结构）。而反序列的过程（即将一个序列还原成为一个对象），需要将一个InputStream(如FileInputstream、ByteArrayInputStream等)封装在ObjectInputStream内，然后调用readObject()即可。会在此项目的工作空间生成一个 my.out文件。序列化后的内容稍后补齐，先看反序列化后输出如下：

name=SheepMu  
age=24

**序列化会忽略静态变量，即序列化不保存静态变量的状态**。静态成员属于类级别的，所以不能序列化。即 序列化的是对象的状态不是类的状态。这里的不能序列化的意思，是序列化信息中不包含这个静态成员域。最上面添加了static后之所以还是输出24是因为该值是JVM加载该类时分配的值。

总结:

a）当一个父类实现序列化，子类自动实现序列化，不需要显式实现Serializable接口；

b）当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化；

c） static,transient后的变量不能被序列化；

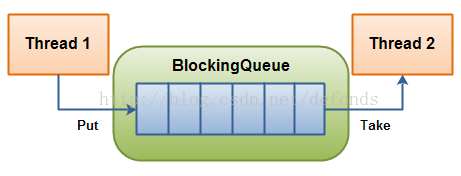
什么时候使用序列化：  
一：对象序列化可以实现分布式对象。主要应用例如：RMI要利用对象序列化运行远程主机上的服务，就像在本地机上运行对象时一样。  
二：java对象序列化不仅保留一个对象的数据，而且递归保存对象引用的每个对象的数据。可以将整个对象层次写入字节流中，可以保存在文件中或在网络连接上传递。利用对象序列化可以进行对象的"深复制"，即复制对象本身及引用的对象本身。序列化一个对象可能得到整个对象序列。

## 4 Java concurrent包BlockingQueue

Java 5 添加了一个新的包到 Java 平台，java.util.concurrent 包。这个包包含有一系列能够让 Java 的并发编程变得更加简单轻松的类。在这个包被添加以前，你需要自己去动手实现自己的相关工具类。

**阻塞队列 BlockingQueue**

**一个线程往里边放，另外一个线程从里边取的一个 BlockingQueue。**

  
一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。  
负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

**数组阻塞队列 ArrayBlockingQueue**。ArrayBlockingQueue 类实现了 BlockingQueue 接口。  
ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个数组里。

**延迟队列 DelayQueue链阻塞队列 LinkedBlockingQueue具有优先级的阻塞队列 PriorityBlockingQueue同步队列 SynchronousQueue阻塞双端队列 BlockingDeque**

**并发 Map(映射) ConcurrentMap**

ConcurrentHashMap 和 java.util.HashTable 类很相似，但 ConcurrentHashMap 能够提供比 HashTable 更好的并发性能。在你从中读取对象的时候 ConcurrentHashMap 并不会把整个 Map 锁住。此外，在你向其中写入对象的时候，ConcurrentHashMap 也不会锁住整个 Map。它的内部只是把 Map 中正在被写入的部分进行锁定。  
另外一个不同点是，在被遍历的时候，即使是 ConcurrentHashMap 被改动，它也不会抛 ConcurrentModificationException。尽管 Iterator 的设计不是为多个线程的同时使用。

1.Callable<V>

2.Semaphore

3.ReentrantLock与Condition

4.BlockingQueue

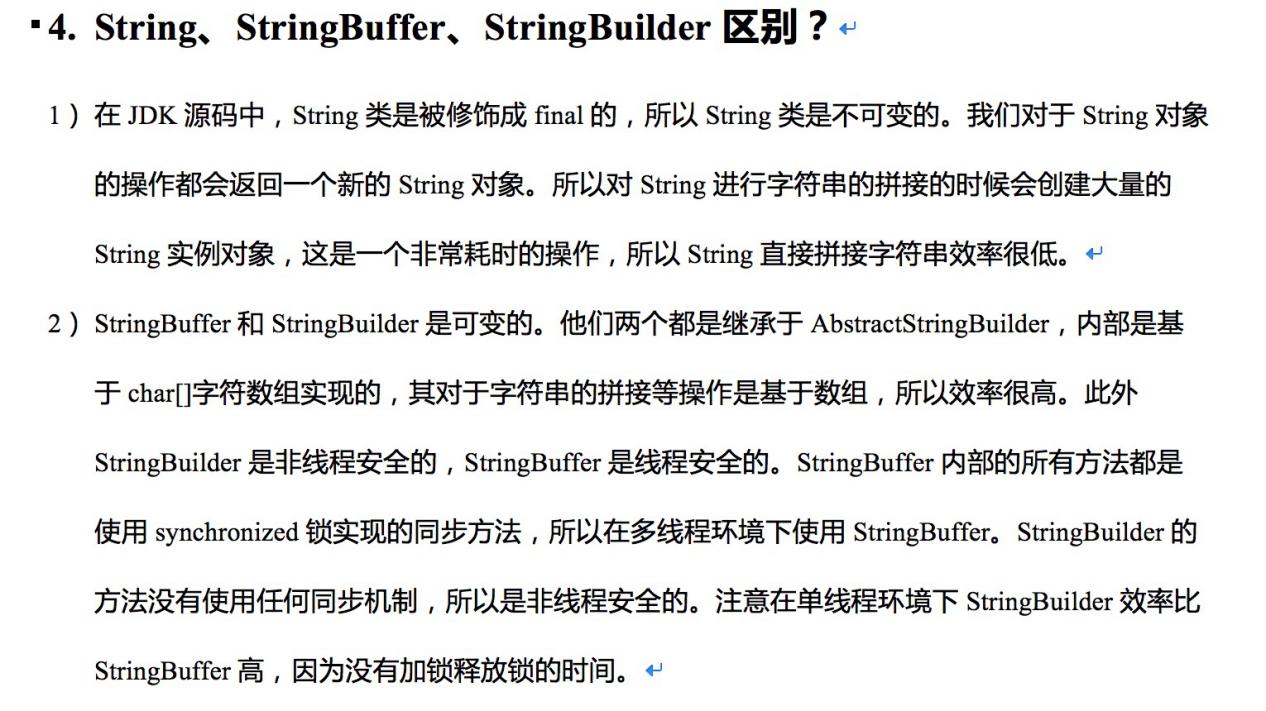
6.CountDownLatch

7.CyclicBarrier

**闭锁和栅栏见多线程的东西同步机制的5**

## 5 String stringbuffer stringbuilder区别，平时用到了哪些

1.在执行速度方面的比较：StringBuilder > StringBuffer   
2.StringBuffer与StringBuilder，他们是字符串变量，是可改变的对象，每当我们用它们对字符串做操作时，实际上是在一个对象上操作的，不像String一样创建一些对象进行操作，所以速度就快了。  
3.StringBuilder：线程非安全的  
 StringBuffer：线程安全的  
 当我们在字符串缓冲去被多个线程使用是，JVM不能保证StringBuilder的操作是安全的，虽然他的速度最快，但是可以保证StringBuffer是可以正确操作的。当然大多数情况下就是我们是在单线程下进行的操作，所以大多数情况下是建议用StringBuilder而不用StringBuffer的，就是速度的原因。  
  
对于三者使用的总结：1.如果要操作少量的数据用 = String  
2.单线程操作字符串缓冲区 下操作大量数据 = StringBuilder  
3.多线程操作字符串缓冲区 下操作大量数据 = StringBuffer



## 6 [final 关键字 & 参数传递特点](http://www.cnblogs.com/superzhao/p/4597714.html)

final

1.final 类   表示该类不可继承,

2.final 方法  表示该方法不可被覆盖

3.final 域   表示该字段一被初始化就不能再改变(必须确保在构造器执行之后final域均被设置,且不可改变)

final用于基本类型和不可变类型,对象类型不可变的只是引用没有意义

4.final 参数 表示在作用域里只能读取不能赋值该final变量

参数传递

Java传递是"值传递":

1.基础类型+布尔 是把变量copy了一份传给函数,对原变量无影响

2.对象类型 是把改引用copy了一份(新旧引用指向同一对象),

　　a.在函数内部对新引用的赋值操作不会影响原引用的指向以及指向的对象

　　b.在函数内部对新引用进行对象改变属性操作,不会影响原引用的指向,但是会影响原引用指向的对象

基础类型和对象类型传递的都是值,只是一个是值本身,一个是引用

## 7 Tomcat，Apache，JBoss的区别？

1、Apache是Http服务器，Tomcat是web服务器，JBoss是应用服务器。

2、Apache解析静态的html文件；Tomcat可解析jsp动态页面、也可充当servlet容器。

## 8 字节 kb MB

1GB=1024MB=1024\*1024KB=1024\*1024\*1024B

1MB=1024KB=1024\*1024B

1KB=1024B

1Byte=1B=8bit(八位二进制数)

字节（Ｂyte):通常将可表示常用英文字符8位二进制称为一字节。

一个英文字母（不分大小写）占一个字节的空间

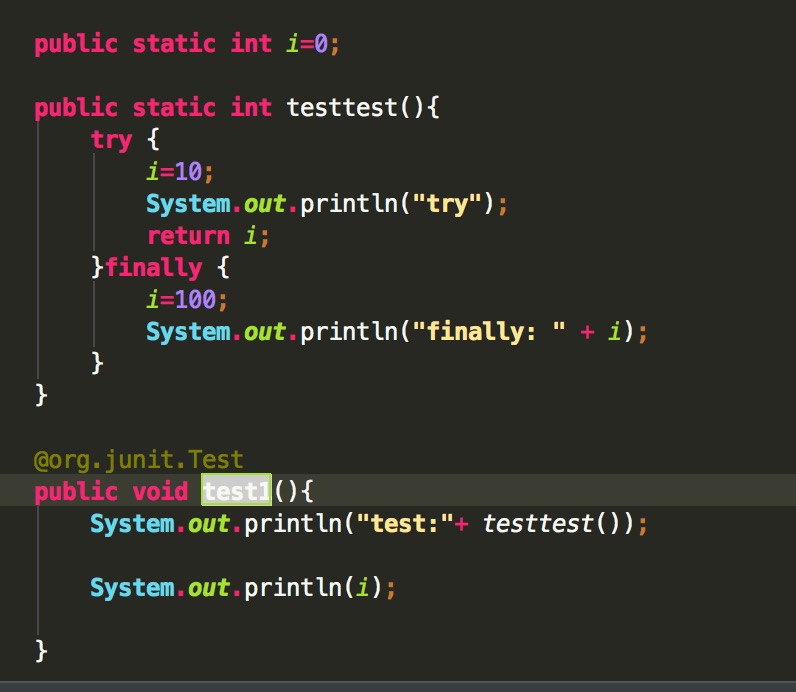
一个中文汉字占两个字节的空间．

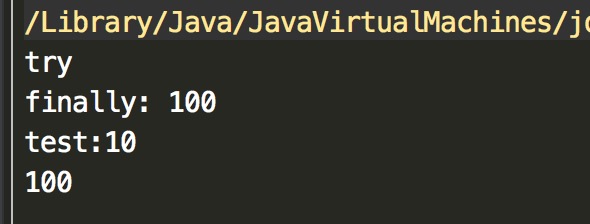
符号：英文标点占一个字节，中文标点占两个字节．

## 9 AtomicInteger底层是怎么实现的

当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。 两个问题： （1）CAS算法仍然可能会出现冲突，例如A、B两个线程，A已经进入写内存但未完成，此时A读取到的副本且读取成功，AB两个线程同时进入写内存操作，必然会造成冲突。 CAS算法本质并非完全无锁，而是把获得锁和释放锁推迟至CPU原语实现，相当于尽可能的缩小了锁的范围；直接互斥地实现系统状态的改变，它的使用基本思想是copy-on-write——在修改完对象的副本之后再用C

## 10 try里面有return finally会执行吗





## 11 equals()跟==有什么区别

==操作比较的是两个变量的值是否相等，对于引用型变量表示的是两个变量在堆中存储的地址是否相同，即栈中的内容是否相同。       equals操作表示的两个变量是否是对同一个对象的引用，即堆中的内容是否相同。

        ==比较的是2个对象的地址，而equals比较的是2个对象的内容。显然，当equals为true时，==不一定为true。

## 12 websocket





## 13 静态内部类和非静态内部类的区别？

1）java允许我们在一个类里面定义静态内部类（nested class），把nested class封闭起来的类叫外部类。在java中，我们不能用static修饰顶级类（top level class）。只有内部类可以为static。

2）静态内部类和非静态内部类之间到底有什么不同呢？

（1）静态内部类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。

（2）静态内部类不能访问外部类的非静态成员，他只能访问外部类的静态成员。

非静态内部类能够访问外部类的静态和非静态成员。

（3）一个非静态内部类不能脱离外部类实体被创建，必须通过外部类的引用创建实例。一个非静态内部类可以访问外部类的数据和方法，因为他就在外部类里面。

## 14 Java的反射作用原理与作用：

1）原理：在程序运行的时候能够获取自身的的Class信息，比如属性和方法，且能够调用它的任意的方法和属性，这就为Java提供了动态的特性。

2）Java反射的作用：

1. 在运行时判断任意一个对象所属的类；   
2. 在运行时构造任意一个类的对象；   
3. 在运行时判断任意一个类所具有的成员变量和方法；   
4. 在运行时调用任意一个对象的方法；   
5. 生成动态代理。

## 15 抽象类与接口的区别

1）抽象类与接口都不能实例化；

1. 抽象类要被子类继承，接口要被类实现；

3）接口只能声明方法，抽象类即可以声明方法也可以实现方法；

4）接口里面的变量只能是公共静态常量，抽象类里面可以定义变量；

5）抽象类必须被子类实现所有抽象方法，不然子类也是抽象类；

6）接口可以多继承接口。但类只能单继承。

## 16 Static关键字的作用

静态变量、静态方法、静态代码快、内部静态类。

## 17 Java中的异常



# 5 Java集合类

## 1 Vector,ArrayList,LinkedList的区别是什么？

1、Vector、ArrayList是类似数组的形式存储在内存中，LinkedList则以链表的形式进行存储。

2、List中的元素有序、允许有重复的元素，Set中的元素无序、不允许有重复元素。

3、Vector线程同步，ArrayList、LinkedList线程不同步。

4、LinkedList适合指定位置插入、删除操作，不适合查找；ArrayList、Vector适合查找，不适合指定位置的插入、删除操作。

5、ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间。

## 2 hashmap与hashtable的区别是什么

**Hashtable** 继承自 **Dictionary 而 HashMap**继承自**AbstractMap**

****Hashtable的put方法（线程安全的）****

注意1 方法是同步的  
注意2 方法不允许value==null  
注意3 方法调用了key的hashCode方法，如果key==null,会抛出空指针异常

**HashMap的put方法（线程非安全的）**

注意1 方法是非同步的  
注意2 方法允许key==null  
注意3 方法并没有对value进行任何调用，所以允许为null

**补充：**   
Hashtable 有一个 contains方法，容易引起误会，所以在HashMap里面已经去掉了  
当然，2个类都用containsKey和containsValue方法。

HashMap是Hashtable的轻量级实现（非线程安全的实现），他们都完成了Map接口，

**主要区别在于HashMap允许空（null）键值（key）,由于非线程安全，效率上可能高于Hashtable。**

****继承和实现方式不同****

**HashMap 继承于AbstractMap，实现了Map、Cloneable、java.io.Serializable接口。**  
**Hashtable 继承于Dictionary，实现了Map、Cloneable、java.io.Serializable接口。**

****线程安全不同****

**Hashtable的几乎所有函数都是同步的，即它是线程安全的，支持多线程。**  
**而HashMap的函数则是非同步的，它不是线程安全的。若要在多线程中使用**

****对null值的处理不同****

**HashMap的key、value都可以为null。**  
**Hashtable的key、value都不可以为null。**

****支持的遍历种类不同****

**HashMap只支持Iterator(迭代器)遍历。**  
**而Hashtable支持Iterator(迭代器)和Enumeration(枚举器)两种方式遍历。**

****通过Iterator迭代器遍历时，遍历的顺序不同****

**HashMap是“从前向后”的遍历数组；再对数组具体某一项对应的链表，从表头开始进行遍历。**  
**Hashtabl是“从后往前”的遍历数组；再对数组具体某一项对应的链表，从表头开始进行遍历。**

****容量的初始值 和 增加方式都不一样****

**HashMap默认的容量大小是16；增加容量时，每次将容量变为“原始容量x2”。**  
**Hashtable默认的容量大小是11；增加容量时，每次将容量变为“原始容量x2 + 1”**

****添加key-value时的hash值算法不同****

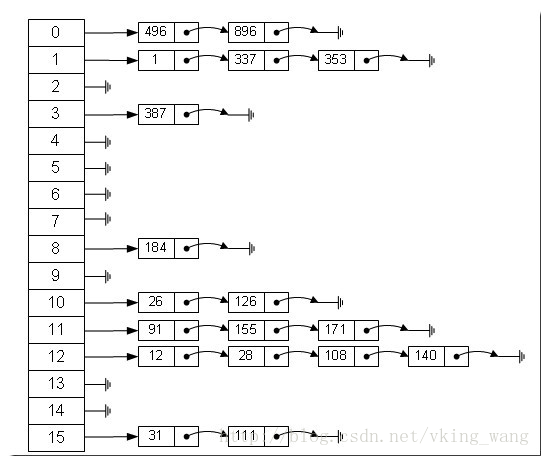
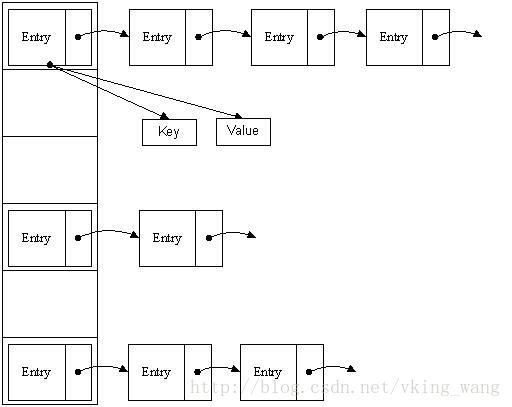
**HashMap添加元素时，是使用自定义的哈希算法。**  
**Hashtable没有自定义哈希算法，而直接采用的key的hashCode()。**

****部分API不同****

**Hashtable支持contains(Object value)方法，而且重写了toString()方法；**  
**而HashMap不支持contains(Object value)方法，没有重写toString()方法。在Entry里面写的**

## 3 HashMap的原理

HashMap的数据结构就是数组加链表（有图就是拉链法实现）



**解决hash冲突的办法**

1. 开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）
2. 再哈希法
3. **链地址法**
4. 建立一个公共溢出区

HashMap是无序的，有序的是LinkedHashMap（在内部增加了一个链表，用以存放元素的顺序）和TreeMap（实现了SortedMap接口，这就意味着可以对元素进行排序）

HashMap的默认的容量是16，装填因子是0.75

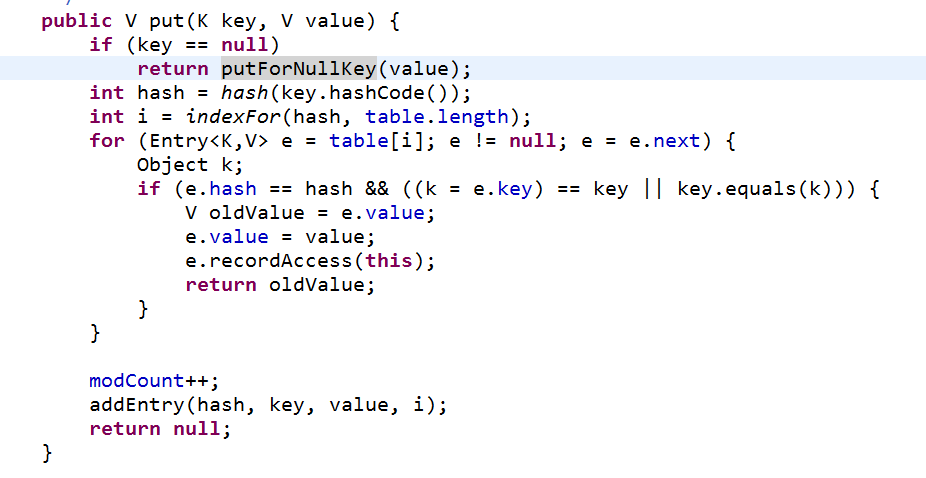
理论上来说当然是O(1)，但是实际上还有很多时间开销的，比如hash碰撞，另外hash的计算也要耗费CPU时间。所以一般我们认为它的时间复杂度是常数级的。

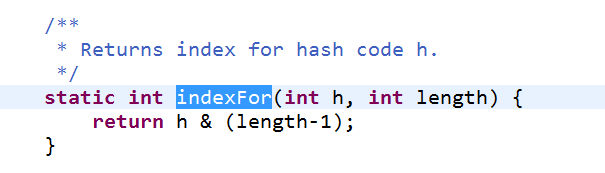
问：这个可以直接去看HashMap的源码，“java怎样哈希一个任意的类的实例呢？” 简单地说 就是通过类的equals和hashCode方法。

答：如果很好地实现了equals和hashCode方法，HashMap能保证以O(1)的复杂度查找我自定的类的实例吗？以及，如果没有重写equals和hashCode方法，会影响吗效率吗？  
（不考虑深判等）

问：你不写的话，就不能用自定义的类做key，HashMap不能保证O(1)的复杂度，它是根据hashcode计算出一个对象在桶的位置，一般情况下，同一类不同对象能保证hashcode值不同就可以了。  
答：理想的情况下在较好实现了hashcode后是复杂度是O(1)，理想情况是所有键值对在桶中均匀分布，很显然不太现实把

## 4 HashMap源码





所以他的长度要为16，然后扩容为16的2的整数倍次方，愿意就在这个indexFor这里是求余。求余%这种就有些浪费时间，如果是用位运算求余就很快。

什么时候ReHash

在介绍HashMap的内部实现机制时提到了两个参数，DEFAULT\_INITIAL\_CAPACITY和DEFAULT\_LOAD\_FACTOR，DEFAULT\_INITIAL\_CAPACITY是table数组的容量，DEFAULT\_LOAD\_FACTOR则是为了最大程度避免哈希冲突，提高HashMap效率而设置的一个影响因子，将其乘以DEFAULT\_INITIAL\_CAPACITY就得到了一个阈值threshold，当HashMap的容量达到threshold时就需要进行扩容，这个时候就要进行ReHash操作了，可以看到下面addEntry函数的实现，当size达到threshold时会调用resize函数进行扩容。

void addEntry(int hash, K key, V value, int bucketIndex) {

ntry<K,V> e = table[bucketIndex];

table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

if (size++ >= threshold)

resize(2 \* table.length);

} 在扩容的过程中需要进行ReHash操作，而这是非常耗时的，在实际中应该尽量避免。

## 5 TreeMap的实现 TreeSet、红黑树

TreeMap 的实现就是红黑树数据结构，也就说是一棵自平衡的排序二叉树，这样就可以保证当需要快速检索指定节点。

LinkedHashMap 是基于元素进入集合的顺序或者被访问的先后顺序排序，TreeMap 则是基于元素的固有顺序 (由 Comparator 或者 Comparable 确定)。

LinkedHashMap 是根据元素增加或者访问的先后顺序进行排序，而 TreeMap 则根据元素的 Key 进行排序

TreeSet 和 TreeMap 的关系TreeSet 的 ① 号、② 号构造器的都是新建一个 TreeMap 作为实际存储 Set 元素的容器，而另外 2 个构造器则分别依赖于 ① 号和 ② 号构造器，由此可见，TreeSet 底层实际使用的存储容器就是 TreeMap。

**Java 实现的红黑树**

上面的性质 3 中指定红黑树的每个叶子节点都是空节点，而且并叶子节点都是黑色。但 Java 实现的红黑树将使用 null 来代表空节点，因此遍历红黑树时将看不到黑色的叶子节点，反而看到每个叶子节点都是红色的。

* 性质 1：每个节点要么是红色，要么是黑色。
* 性质 2：根节点永远是黑色的。
* 性质 3：所有的叶节点都是空节点（即 null），并且是黑色的。
* 性质 4：每个红色节点的两个子节点都是黑色。（从每个叶子到根的路径上不会有两个连续的红色节点）
* 性质 5：从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点。

## 6 **[Java中List、Set、Map区别](http://blog.csdn.net/bug_moving/article/details/52667199)**

List,Set都是继承自Collection接口;

List特点：元素有放入顺序，元素可重复;   
Set特点：元素无放入顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的）;   
Map特点：元素按键值对存储，无放入顺序

## 7 HashMap 和 ConcurrentHashMap的区别

1）线程安全的Map，

2）基于分段锁实现，默认是16级分段，这里面的每个Segment都相当于一个HashTable

3）每个Segment只允许一个线程写，但是不限制读。

两个hash过程，第一次找到所在的桶，并将桶锁定，第二次执行写操作。

而读操作不加锁

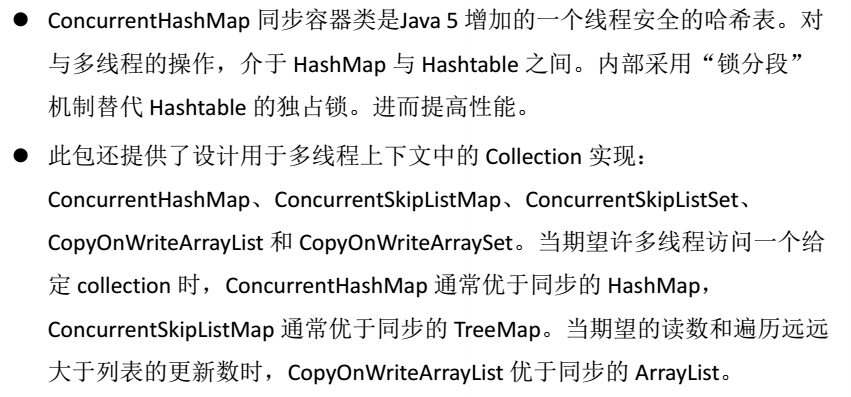
1）ConcurrentHashMap是基于分段锁实现的，具体可以理解成把一个大的Map拆分成N个小的HashTable（默认是16个），根据key.hashCode()来决定把key放到哪个HashTable。在ConcurrentHashMap中就是把Map分成了N个Segment，put和get的时候，都是先根据key.hashCode()算出放在哪个Segment中，然后对对应的Segment加锁，这样就不会影响其余Segment的并发访问。这样效率就提升了N倍，默认提升了16倍。其实也就是同时允许16个线程分别对16个Segment操作，只有写操作才需要锁住Segment，读线程基本不受限制。

2）基本上ConcurrentHashMap是HashMap 和 HashTable的结合。那么为什么ConcurrentHashMap里面的get()操作，也就是读操作不用加锁呢，除非读到的值是空的才会加锁重读？

原因是它的get方法里将要使用的共享变量都定义成volatile，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作优先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景。

1. ConcurrentHashMap不允许key或则value为空。

**Concurrent包还提供了很多collection的实现**



## 8 为什么要用HashTable

根据key查value比较快，线程安全

# 6 设计模式

## 1 设计模式分类

创建型：工厂方法模式，抽象工厂模式，生成器模式，原型模式，单例模式

行为型：责任链，命令，解释器，迭代器，中介者，备忘录，观察者，状态，策略，模板方法，访问者

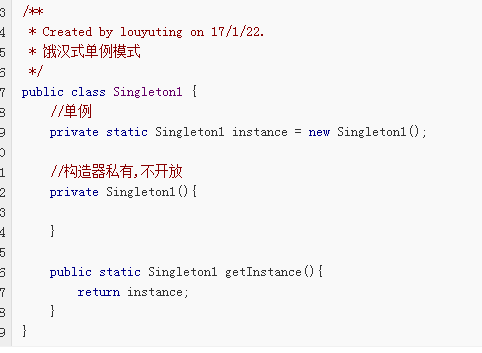
结构型：适配器，组合，代理，享元，外观，桥接，装饰

## 2 设计模式的应用

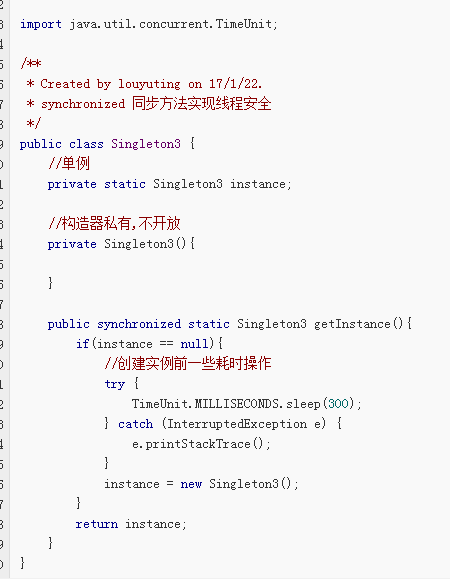
比如JDK中clone用了原型模式，IO的管道各种stream用了装饰模式，遍历的Iterator用了迭代器模式。Spring中bean使用了工程模式，BeanFactory肯定用到了模板模式，然后aop使用了动态代理模式，创建bean使用了单例模式，servlet也是单例模式

## 3 单例的线程安全的程序

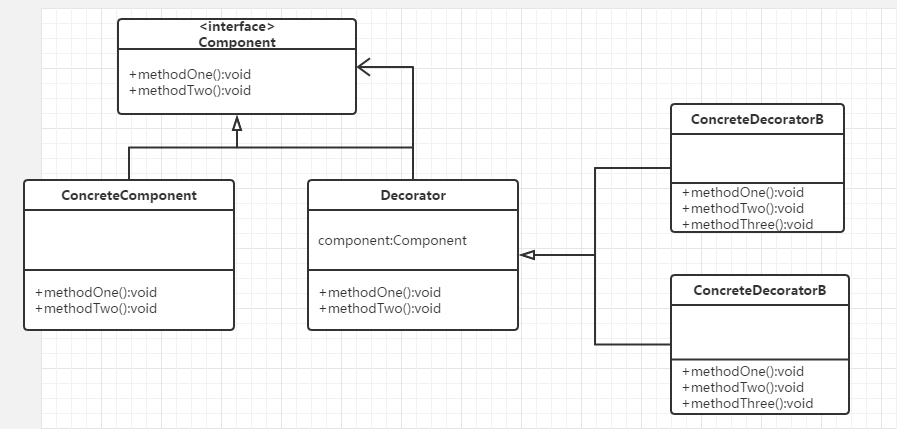
**1.饿汉式单例**饿汉式单例是指在方法调用前，实例就已经创建好了，所以肯定是线程安全的



线程安全的单例-synchronized方法



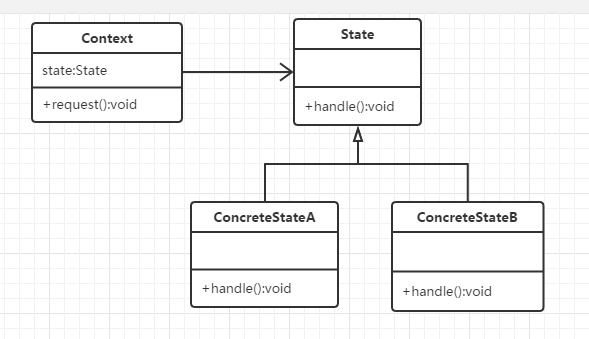
## 4 装饰模式



动态地给对象添加一些额外的职责。就功能来说装饰模式相比生成子类更为灵活。

比如我现在在玩网易游戏，然后我本来攻击距离只有3米，我吃了个苹果，攻击距离变6米，我吃了个火龙果，攻击距离变12米。

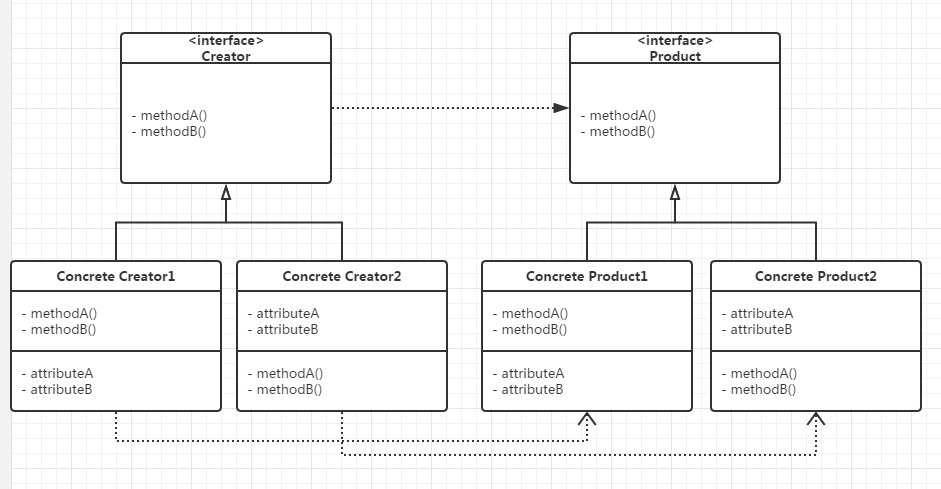
## 5 状态模式



允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。

比如我现在在玩网易的枪战游戏，我手枪有5发子弹，这是一个状态，打了一发，4发也是一个状态

## 6 工厂模式



工厂模式的优点

使用工厂模式可以让用户的代码和某个特定类的子类的代码解耦。

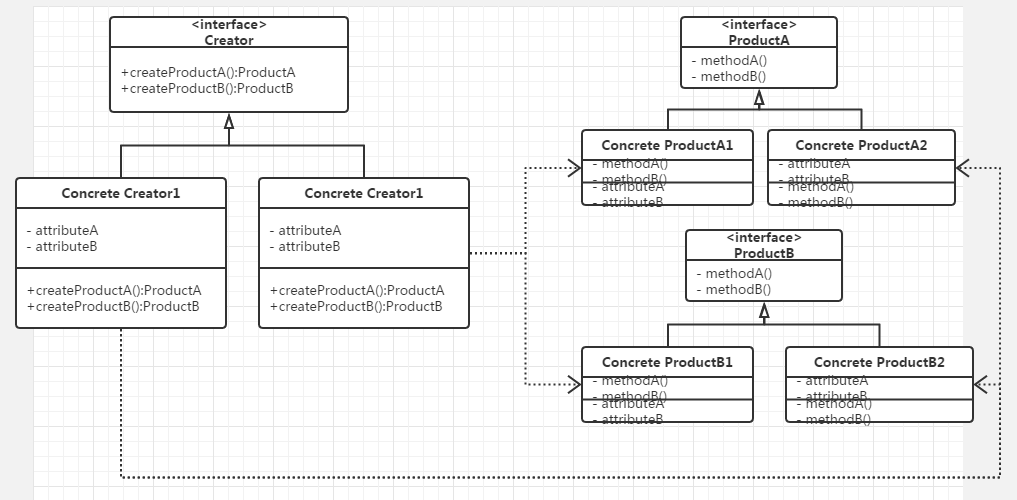
工厂方法的使用用户不必知道它所使用的对象是怎么被创建的，只需要知道该对象有哪些方法即可。

适用工厂模式的情景

用户需要一个类的子类的实例，但不希望与该类的子类形成耦合。

用户需要一个类的子类的实例，但用户不知道该类有哪些子类可用。

## 7 抽象工厂模式



抽象工厂模式的优点

抽象工厂模式可以为用户创建一系列相关的对象，使用户和创建这些对象的类脱耦。

使用抽象工厂模式可以方便的为用户配置一系列对象。用户使用不同的具体工厂就能得到一组相关的对象，同时也能避免用户混到不同系列的对象中。

在抽象工厂模式中，可以随时增加“具体工厂”为用户提供一组相关的对象。

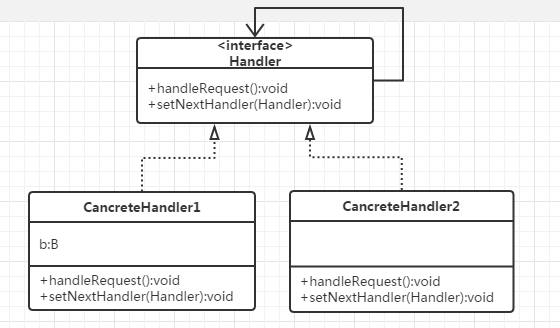
适用抽象工厂模式的情景

系统需要为用户提供多个对象，但不希望用户直接使用new运算符实例化这些对象，即希望用户和创建对象的类脱耦。

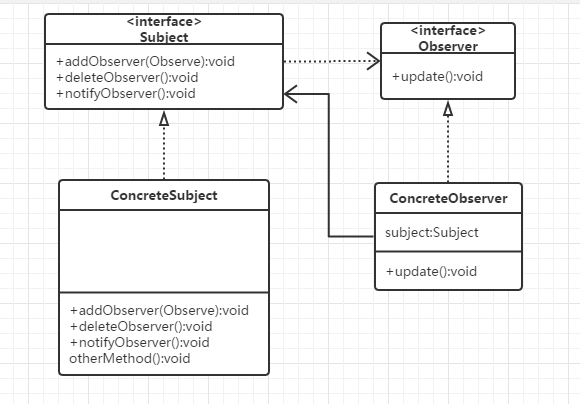
系统需要为用户提供多个相关的对象，以便用户联合使用它们，但又不希望用户来决定这些对象是如何关联的。

系统需要为用户提供一系列对象，但只需要用户知道这些对象有哪些方法可用，不需要用户知道这些对象的创建过程。

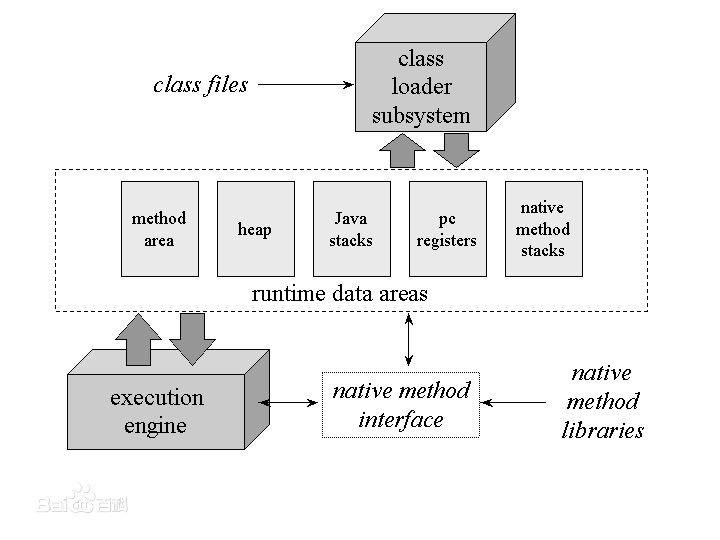
## 8 责任链模式



## 9 观察者模式



# 7 JVM



## 1. JVM运行时的数据区域(JVM的内存模型)：

（1）程序计数器--线程私有

（2）Java虚拟机栈--线程私有

每个Java方法执行时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个Java方法从调用到执行完成的过程就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

（3）本地方法栈（也叫native栈，线程私有）

1. 虚拟机栈为虚拟机执行Java方法服务；

2. 本地方法栈则为虚拟机用到的Native方法服务。

（4）Java堆（线程共享区域-虚拟机启动时创建）--也是GC管理的地方

（5）方法区（线程共享区-虚拟机启动时创建）

用于存储以被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

## 2. 对象创建与定位访问：

主要通过new关键字来实现：

首先判断有没有类加载，没有加载过就先加载类----》

为新生对象分配内存--》

内存初始化--》

对象的必要设置。

对象内存布局：对象头、实例数据、对齐填充。

访问定位：通过虚拟机栈上的reference 数据来操作堆上面具体对象。

Hotspot使用的是直接指针。reference直接指向Java堆上的对象指针。

## 3.GC—怎么判断对象是否可回收：

（1）引用计数算法：引用一次加1，失效就减1. 但是不能解决互相引用的情况。

（2）可达性分析算法：通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链（即从GC Roots到对象不可达）时，则证明此对象是不可用的。

## 4.判断对象是否可回收的可达性分析算法中，GC Roots有哪些？

1）虚拟机栈中引用的对象 ；  
2）方法区中静态属性引用的对象   
3）方法区中常量引用的对象   
4）本地方法栈中JNI（即native方法）引用的对象

## 5. Java中引用类型有哪几种？

强引用、软引用、弱引用、虚引用。引用强度依次减弱。

## 6.新生代、老年代、持久代分别指什么？

新生代：主要用来存放新生的对象。 老年代：主要用来存放应用程序中生命周期长的对象。经历了N（可配置）次垃圾回收还健在

持久代：主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大。

## 7.GC—有哪些垃圾回收算法？各有什么优缺点？各自应用场景？

（1）标记-清除算法：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。

A）缺点：标记和清除效率不高；会存在大量的不连续内存碎片

（2）复制算法----新生代才采用的算法：将可用内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。

A）缺点：内存缩小为原来的一半，内存的损失太大。

B）解决方法：将内存分为较大的Eden区域和两个较小的Survivor区域，每次回收将Eden中和Survivor中还存活的对象移动到另一块空的Survivor区域中。

（3）标记整理算法----老年代常采用算法：过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。

（4）分代收集算法：在Java中把堆分为新生代和老年代，然后根据各块的特点采用最适当的收集算法，其实也就是上面几种算法的结合使用。

## 8.JVM中使用了哪些技术来加快内存分配

（1）指针碰撞：跟踪在Eden上新创建的对象。当有新对象创建，只需要判断新创建对象的大小是否满足剩余的Eden空间。如果新对象满足要求，则其会被分配到Eden空间，同样位于Eden的最上面。所以当有新对象创建时，只需要判断此新对象的大小即可，因此具有更快的内存分配速度

（2）TBAL：在多线程环境下进行内存分配难免要进行加锁，但是我们可以对每个线程分配一个小片空间，这个空间是线程私有的，就可以实现在不加锁情况下的并发分配内存。

## 9.怎么理解Stop the world？

1）可达性分析对执行时间的敏感点的一个体现就是GC停顿上面，可达性分析工作必须在一个能确保一致性的快照中进行–这里的一致性是指在整个分析期间整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中对象引用关系还在不断变化的情况，该点不满足就无法得到保证。这一点是导致GC进行时必须停顿所有Java执行线程（sun称为 “Stop the world”）的其中一个重要原因。

2）安全点：在HotSpot虚拟机中，借助于OopMap这种数据结构的协助下，可以快速且准确的完成GCRoots的枚举。但是不可能为每条指令都生成OopMap，这样空间成本就非常高。所以HotSpot中只是在“特定的位置”记录了这些信息，这些位置称为安全点，即程序执行时并非在所有地方都停顿下来开始GC，只有到达安全点时才能暂停。

## 10.Hotspot中的垃圾收集器

Hotspot中常见的垃圾收集器如下图所示：



### （1）Serial收集器—新生代

这个收集器是一个采用复制算法的单线程的收集器，单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作，另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作，直到它收集结束为止。

### （2）ParNew收集器—新生代

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为都与Serial收集器完全一样,包括使用的也是复制算法。

### （3）Parallel收集器—新生代

Parallel收集器也是一个新生代收集器，也是用复制算法的收集器，也是并行的多线程收集器,但是它的特点是它的关注点和其他收集器不同。介绍这个收集器主要还是介绍吞吐量的概念。CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel收集器的目标则是打到一个可控制的吞吐量。所谓吞吐量的意思就是CPU用于运行用户代码时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总运行100分钟，垃圾收集1分钟，那吞吐量就是99%。另外，Parallel收集器是虚拟机运行在Server模式下的默认垃圾收集器。

### （4）Serial Old收集器—老年代

Serial收集器的老年代版本，同样是一个单线程收集器，使用“标记-整理算法”，

### （5）Parallel Old收集器—老年代

Parallel收集器的老年代版本，使用多线程和“标记-整理”算法。

### （6）CMS收集器—老年代

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的老年代收集器。

## 11.Minor GC和Full GC有什么区别？

1）新生代GC（Minor GC）:指发生在新生代的垃圾收集动作，因为大多数Java对象存活率都不高，所以Minor GC非常频繁，一般回收速度也比较快

2）老年代GC(Full GC）:指发生在老年代的垃圾收集动作，出现了Full GC，经常会伴随至少一次的Minor GC（但并不是绝对的）。Full GC的速度一般要比Minor GC慢上10倍以上

## 12.类加载的流程？

加载： 导入Class文件

—》连接

—》初始化：静态变量的初始化、静态代码块的执行。

—》使用：

—》卸载。

## 13.JVM有哪些类加载器？每种类加载器分别加载哪里的class代码？

1）启动类加载器---- 加载位置 ：$JAVA\_HOME/lib/rt.jar里所有的class或则被-Xbootclasspath参数指定的路径中。

2）扩展类加载器---- 加载位置 ：加载JAVA\_HOME/lib/ext目录下的或者被java.ext.dirs系统变量指定所指定的路径中所有类库。

3）应用程序类加载器----加载位置 ：classpath环境变量中指定的jar包及目录中class；

4）自定义加载器----自定义的类加载器，手动加载，只需要继承自ClassLoader然后实现loadClass()方法。

## 14.类加载的双亲委派模型



双亲委派模型的工作过程是：   
1）如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，向上传递。   
2）所有的加载请求最终都会传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

## 15. Java的内存模型

Java内存模型定义了一种多线程访问Java内存的规范：

（1）Java内存模型将内存分为了主内存和工作内存。类的状态是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，都会读一次主内存中的变量，并将这些变量在自己的线程的工作内存中保存一份拷贝，当运行自己线程的代码时候，操作的都是自己工作内存中的变量副本。在线程代码执行完毕之后，会将最新的值更新到主内存中去。

（2）定义了8个原子操作，用于操作主内存和工作内存中的变量的交互。

（3）定义了volatile变量，保证了变量的可视性。

（4）原子性、可见性、有序性

（5）happens-before：针对变量的可见性制定的一些通用规则，比如定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的。

## 16 GC的优化方案？

基本的原则就是尽可能地减少垃圾和减少GC过程中的开销。其中需要注意，JVM进行次GC的频率很高，但因为Minor GC占用时间极短，所以对系统产生的影响不大。更值得关注的是Full GC的触发条，具体措施包括以下几个方面:   
(1)不要显式调用System.gc()   
调用System.gc()也仅仅是一个请求(建议)。JVM接受这个消息后，并不是立即做垃圾回收，而只是对几个垃圾回收算法做了加权，使垃圾回收操作容易发生，或提早发生，或回收较多而已。但即便这样，很多情况下它会触发Full GC，也即增加了间歇性停顿的次数。   
(2)尽量减少临时对象的使用   
临时对象在跳出函数调用后，会成为垃圾，少用临时变量就相当于减少了垃圾的产生，也就减少了Full GC的概率。   
(3)对象不用时最好显式置为Null   
一般而言，为Null的对象都会被作为垃圾处理，所以将不用的对象显式地设为Null，有利于GC收集器判定垃圾，从而提高了GC的效率。   
(4)尽量使用StringBuffer,而不用String来累加字符串   
由于String是常量，累加String对象时，并非在一个String对象中扩增，而是重新创建新的String对象，如Str5=Str1+Str2+Str3+Str4,这条语句执行过程中会产生多个垃圾对象，因为对次作“+”操作时都必须创建新的String对象，但这些过渡对象对系统来说是没有实际意义的，只会增加更多的垃圾。避免这种情况可以改用StringBuffer来累加字符串，因StringBuffer是可变长的，它在原有基础上进行扩增，不会产生中间对象。   
(5)能用基本类型如Int,Long,就不用Integer,Long对象   
基本类型变量占用的内存资源比相应对象占用的少得多，如果没有必要,最好使用基本变量。   
(6)尽量少用静态对象变量   
静态变量属于全局变量，不会被GC回收，它们会一直占用内存。   
(7)分散对象创建或删除的时间   
集中在短时间内大量创建新对象，特别是大对象，会导致突然需要大量内存，JVM在面临这种情况时，只能进行Full GC，以回收内存或整合内存碎片，从而增加主GC的频率。集中删除对象，道理也是一样的。它使得突然出现了大量的垃圾对象，空闲空间必然减少，从而大大增加了下一次创建新对象时强制主GC的机会。

## 17 Java即使有了GC也会出现的内存泄漏情况？举例说明。

1、静态集合类像HashMap、Vector等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，所有的对象Object也不能被释放，因为他们也将一直被Vector等应用着。

Static Vector v = new Vector();  
for (int i = 1; i<100; i++)   
{  
Object o = new Object();   
v.add(o);   
o = null;   
}

在这个例子中，代码栈中存在Vector对象的引用v和Object对象的引用o。在For循环中，我们不断的生成新的对象，然后将其添加到Vector对象中，之后将o引用置空。问题是当o引用被置空后，如果发生GC，我们创建的Object对象是否能够被GC回收呢？答案是否定的。因为，GC在跟踪代码栈中的引用时，会发现v引用，而继续往下跟踪，就会发现v引用指向的内存空间中又存在指向Object对象的引用。也就是说尽管o引用已经被置空，但是Object对象仍然存在其他的引用，是可以被访问到的，所以GC无法将其释放掉。如果在此循环之后，Object对象对程序已经没有任何作用，那么我们就认为此Java程序发生了内存泄漏。

2.各种连接，[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/bug_moving/article/details/_blank)连接，网络连接，IO连接等没有显示调用close关闭，不被GC回收导致内存泄露。

3.监听器的使用，在释放对象的同时没有相应删除监听器的时候也可能导致内存泄露。

## 18 JVM什么时候会触发Full GC

1. System.gc()方法的调用
2. 老年代代空间不足,老年代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出如下错误：[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "http://blog.csdn.net/chenleixing/article/details/_blank).lang.OutOfMemoryError: [Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/chenleixing/article/details/_blank)heap space
3. 永生区空间不足Permanet Generation中存放的为一些class的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下也会执行Full GC。如果不足也会抛出异常。
4. CMS GC时出现promotion failed和concurrent mode failure对于采用CMS进行老年代GC的程序而言，尤其要注意GC日志中是否有promotion failed和concurrent mode failure两种状况，当这两种状况出现时可能会触发Full GC。
5. 统计得到的Minor GC晋升到旧生代的平均大小大于老年代的剩余空间这是一个较为复杂的触发情况，Hotspot为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行Minor GC时，做了一个判断，如果之前统计所得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发Full GC。

6、堆中分配很大的对象 所谓大对象，是指需要大量连续内存空间的java对象，例如很长的数组，此种对象会直接进入老年代，而老年代虽然有很大的剩余空间，但是无法找到足够大的连续空间来分配给当前对象，此种情况就会触发JVM进行Full GC。

## 19 Java相较于PHP、C#、Ruby等一样很优秀的编程语言的优势是什么？

（1）体系结构中立，跨平台性能优越。Java程序依赖于JVM运行，javac编译器编译Java程序为平台通用的字节码文件（.class），再由JVM与不同操作系统匹配，装载字节码并解释（也有可能是编译，会在第三个问题中说到）为机器指令执行。

（2）安全性优越。通过JVM与宿主环境隔离，且Java的语法也一定程度上保障了安全，如废弃指针操作、自动内存管理、异常处理机制等。

（3）多线程。防止单线程阻塞导致程序崩溃，分发任务，提高执行效率。

（4）分布式。支持分布式，提高应用系统性能。

（5）丰富的第三方开源组件。Spring、Struts、Hibernate、Mybatis、Quartz等等等等。

## 20 字节码是什么？.class字节码文件是什么？

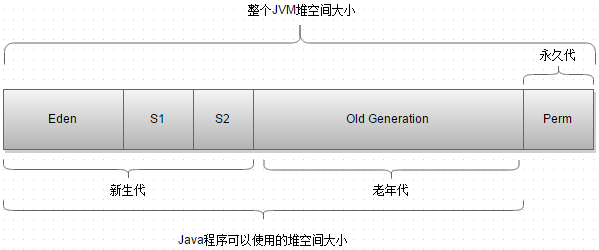
（1）字节码是包含Java内部指令集、符号集以及一些辅助信息的能够被JVM识别并解释运行的符号序列。字节码内部不包含任何分隔符区分段落，且不同长度数据都会构造成n个8位字节单位表示。

（2）.class里存放的就是Java程序编译后的字节码，包含了类版本信息、字段、方法、接口等描述信息以及常量池表，一组8位字节单位的字节流组成了一个字节码文件。

## 21 JVM一些问题，怎么确定服务器端一个死循环

服务器和客户端进行通信是通过socket，首先产生一个Socket实例，通过这个例子中，服务器调用accept这个方法来接收从客户端发送的信息，但在时间上产生Socket实例必须初始化一个端口。负责接收客户端请求！   
客户端必须向服务器发送一个消息产生一个Socket实例初始化必须指定服务器的IP地址，并指定该服务所收取的端口号，以便客户端可以找到你想要接收服务器，发现在可发遍的地方。

## 22 Java堆分块（现有内存放不下）



上图中，刻画了Java程序运行时的堆空间,可以简述成如下2条

1.JVM中堆空间可以分成三个大区，新生代、老年代、永久代

2.新生代可以划分为三个区，Eden区，两个幸存区

## 23 OOM异常解析

堆内存的OOM异常   
a）如何产生？  
堆内存用于存储实例对象，当我们不断创建对象，并且对象都有引用指向(GC Roots到对象之间有可达路径)，那么垃圾回收机制就不会清理这些对象，当对象多到挤满堆内存的上限后，就产生OOM异常。

b）模拟堆内存OOM异常  
PS:在eclipse的Arguments中可以设置VM arguments，这就是JVM的一些参数。   
－Xms：设置堆的最小值   
－Xmx：设置堆的最大值

public class A(){

public static void main(String[] args){

while(true){

new Person();

}

}

}//运行结果中出现：java.lang.OutOFMemory:Java heap space//说明是在堆内存中发生了OOM异常。

c）如何解决？  
使用内存映像分析工具：Eclipse Memory Analyzer对dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，即要搞明白是内存泄漏还是内存溢出。   
PS：内存泄漏导致的OOM：new出来的很多对象已经不需要了，但仍然有引用指向，所以垃圾回收机制无法回收。   
PS：内存溢出：new出来的对象都是需要的，但堆内存太小装不下了。   
如果是内存泄漏，通过工具查看泄漏对象到GC Roots的引用链。找到泄漏对象是通过怎样的路径与GC Roots发生关联，然后导致垃圾回收机制无法自动回收的。   
如果不存在内存泄漏，也就是所有的对象都必须存在，这时候就调大堆内存。

JVM栈和本地方法栈的OOM异常   
a）StackOverFlowError   
当线程请求的栈深度大于虚拟机所允许的最大栈深度，就会抛出这个异常。   
b）OutOfMemeoryError   
当虚拟机要扩展栈时无法申请到足够空间的内存，就会抛出这个异常。   
PS：这两种异常其实是对同一个问题的两种描述。在单一线程下，不论是栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是StackOverFlowError。通过[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/u010425776/article/details/_blank)发现，如果给每个线程的JVM栈分配的内存越大，大的栈帧在这个JVM栈中也能装得下，理应StackOverFlowError会减少，但事实却恰恰相反：当每个线程的JVM栈越大，那么所能创建的线程数就越少，稍微建立几个线程可能就会把有限的内存资源耗尽。

运行时常量池的OOM异常   
我们通过String类的intern()方法向方法区中的常量池添加内容。   
intern方法的作用是：当常量池中已经有这个String类型所对应的字符串的话，就返回这个字符串的引用；如果常量池中没有这个字符串的话就将这个字符串添加到常量池中，再返回这个字符串的引用。

方法区的OOM异常   
a）如何产生？  
方法区中存放的是Class的相关信息，如：类名、访问修饰符、常量池、字段描述、方法描述等。  
如果产生大量的类就有可能将方法区填满，从而产生方法区的OOM异常。

b）注意点   
方法区的OOM异常是非常常见的，特别是在一些动态生成大量Class的应用中(JSP)，需要特别注意类的回收。

本机直接内存的OOM异常

# 8 框架

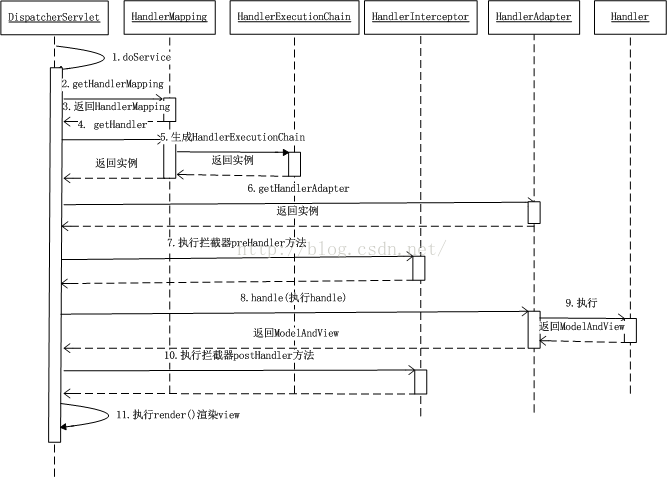
## 1 SpringMVC框架

### 1 SpringMVC的转发机制（转发和重定向）时序图

SpingMVC的架构其实也很简单，它通过一个共通的入门DispatcherServlet来接收所有的request，接下来根据request要求的页面通过handler转送给Controller处理，处理结果返回View给用户，因此对SpringMVC来说，其核心为DispatcherServlet。  
DispatcherServlet作为一扇门，我们要使用这个门，首先得安装上这个门，安装在哪呢？没错，就安装在web.xml这个大屋子，象其他servlet一样，描述servlet的名字和类，再给他一个mapping描述对应映射的网页。

重定向和转发有一个重要的不同：当使用转发时，JSP容器将使用一个内部的方法来调用目标页面，新的页面继续处理同一个请求，而浏览器将不会知道这个过程。 与之相反，重定向方式的含义是第一个页面通知浏览器发送一个新的页面请求。因为，当你使用重定向时，浏览器中所显示的URL会变成新页面的URL, 而当使用转发时，该URL会保持不变。重定向的速度比转发慢，因为浏览器还得发出一个新的请求。同时，由于重定向方式产生了一个新的请求，所以经过一次重定向后，request内的对象将无法使用。

不要仅仅为了把变量传到下一个页面而使用session作用域，那会无故增大变量的作用域，转发也许可以帮助你解决这个问题。  
重定向：以前的request中存放的变量全部失效，并进入一个新的request作用域。  
转发：以前的request中存放的变量不会失效，就像把两个页面拼到了一起。



### 2 SpringMVC解释视图

**SpringMVC 如何解析视图**

请求处理方法返回值类型是

**String**   
**ModelAndView**   
**View**

SpringMVC都会奖其转化为**ModelAndView**对象，再给ViewResolver，最后变成视图对象，如JSP、JSTL、PDF等。

### 3 SpringMVC的优势

1. 上手块，有点web经验的，基本上搭建好框架就能写代码了
2. 轻量，没有struct那么复杂，笨重，使用注解大量简化配置，而且良好的支持rest，而且structs自从那次漏洞之后，损失了不少名誉
3. Spring，SpringMVC完美结合，是一家人嘛，甚至结合spring data（SpringBoot）甚至可以一站式开发

## 2 Spring框架（依赖注入的源码）

### 1 Spring中IOC用到了java中的那些特性

ioc的思想最核心的地方在于，资源不由使用资源的双方管理，而由不使用资源的第三方管理，这可以带来很多好处。第一，资源集中管理，实现资源的可配置和易管理。第二，降低了使用资源双方的依赖程度，也就是我们说的耦合度。

所谓的依赖注入，则是，甲方开放接口，在它需要的时候，能够讲乙方传递进来(注入)  
所谓的控制反转，甲乙双方不相互依赖，交易活动的进行不依赖于甲乙任何一方，整个活动的进行由第三方负责管理。责任被反转了。

IoC最原初的目的就是充分利用OO的多态性，使得通过配置文件而不是在代码里硬编码（hardcode）的方式来实例化对象和装配对象图，这样就有了为不同的客户场景服务的灵活性（不同的客户通过配置文件使用不同的子类）。IoC本质上和插件化代码的思路很接近

让我们看看Spring 到底是怎么依赖注入的吧，其实依赖注入的思想也很简单，它是通过反射机制实现的，在实例化一个类时，它通过反射调用类中set方法将事先保存在HashMap中的类属性注入到类中。

### 2 Spring AOP是用什么方法具体实现的

AOP技术利用一种称为"横切"的技术，能够剖解开封装的对象内部，并将那些影响了多个类并且与具体业务无关的公共行为 封装成一个独立的模块（称  
为切面）。更重要的是，它又能以巧夺天功的妙手将这些剖开的切面复原，不留痕迹的融入核心业务逻辑中。这样，对于日后横切功能的编辑和重用都能够带来极大的方便。AOP技术实现，通过动态代理技术或者是在程序编译期间进行静态的"织入"方式。几个基本术语：1、join point（连接点）：是程序执行中的一个精确执行点，例如类中的一个方法。它是一个抽象的概念，在实现AOP时，并不需要去定义一个join point。2、point cut（切入点）：本质上是一个捕获连接点的结构。在AOP中，可以定义一个point cut，来捕获相关方法的调用。3、advice（通知）：是point cut的执行代码，是执行“方面”的具体逻辑。4、aspect（方面）：point cut和advice结合起来就是aspect，它类似于OOP中定义的一个类，但它代表的更多是对象间横向的关系。5、introduce（引入）：为对象引入附加的方法或属性，从而达到修改对象结构的目的。有的OP工具又将其称为mixin。

AOP的源码中用到了两种动态代理来实现拦截切入功能：jdk动态代理和cglib动态代理

AOP:面向切面编程。（Aspect-Oriented Programming）  
AOP可以说是对OOP的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。  
将程序中的交叉业务逻辑（比如安全，日志，事务等），封装成一个切面，然后注入到目标对象（具体业务逻辑）中去。  
实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码

### 3 spring支持五种事务隔离设置

DEFAULT 使用[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/jjavaboy/article/details/_blank)设置的隔离级别 ( 默认 ) ，由 DBA 默认的设置来决定隔离级别   
READ\_UNCOMMITTED 会出现脏读、不可重复读、幻读 ( 隔离级别最低，并发性能高 )   
READ\_COMMITTED  会出现不可重复读、幻读问题（锁定正在读取的行）   
REPEATABLE\_READ 会出幻读（锁定所读取的所有行）   
SERIALIZABLE 保证所有的情况不会发生（锁表）

### 4 spring 中一共定义了六种事务传播属性

PROPAGATION\_REQUIRED -- 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。  
PROPAGATION\_SUPPORTS -- 支持当前事务，如果当前没有事务，就以非事务方式执行。  
PROPAGATION\_MANDATORY -- 支持当前事务，如果当前没有事务，就抛出异常。  
PROPAGATION\_REQUIRES\_NEW -- 新建事务，如果当前存在事务，把当前事务挂起。  
PROPAGATION\_NOT\_SUPPORTED -- 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。  
PROPAGATION\_NEVER -- 以非事务方式执行，如果当前存在事务，则抛出异常。  
PROPAGATION\_NESTED -- 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作。  
前六个策略类似于EJB CMT，第七个（PROPAGATION\_NESTED）是Spring所提供的一个特殊变量。  
它要求事务管理器或者使用JDBC 3.0 Savepoint API提供嵌套事务行为（如Spring的DataSourceTransactionManager）

### 5 @autowired @resource

1、@Autowired与@Resource都可以用来装配bean. 都可以写在字段上,或写在setter方法上。  
2、@Autowired默认按类型装配（这个注解是属业spring的），默认情况下必须要求依赖对象必须存在，如果要允许null 值，可以设置它的required属性为false，如：@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier注解进行使用，如下：

Java代码

@Autowired()@Qualifier("baseDao")

private BaseDao baseDao;

3、@Resource（这个注解属于J2EE的），默认安照名称进行装配，名称可以通过name属性进行指定，如果没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。 当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

Java代码

@Resource(name="baseDao")

private BaseDao baseDao;  
我喜欢用 @Resource注解在字段上，且这个注解是属于J2EE的，减少了与spring的耦合。最重要的这样代码看起就比较优雅。

### 6 [Spring三种bean注入方式](http://glzaction.iteye.com/blog/1299441)

Spring中依赖注入有三种注入方式：

一、构造器注入；xml里面有constructor-arg这个标签

二、设值注入（setter方式注入）；

三、Feild方式注入（注解方式注入）。

### 7 Spring bean单例/多例

bean id="borrowDao": 标识该bean的名称，通过factory.getBean(“id”)来获得实例，  
Singleton : 默认为true，即单实例模式，每次getBean(“id”)时获取的都是同一个实例，如果设置为false，即原型模式，则每次获取的是新创建的实例。

## 3 MyBatis框架

MyBatis 是支持普通SQL查询，存储过程和高级映射的优秀持久层框架。几乎消除了所有JDBC代码和参数的手工设置。以及结果集的检索。使用简单的XML或注解用于配置和原始映射。

1> 加载配置 ①通过配置文件 ②Java中的注解

2> SQL解析

3> SQL执行

4> 结果集映射：可以将结果集转换成HashMap，JavaBean或基本数据类型

xml: <insert></insert> <delete></delete> id parameterType

@select

DB -> Entity -> Mapper.xml -> mapper.java -> service.java -> controller.java -> jsp

### 1 mybatis一级缓存与二级缓存

mybatis一级缓存是一个**SqlSession级别**，sqlsession只能访问自己的一级缓存的数据，二级缓存是跨sqlSession，是**mapper级别**的缓存，对于mapper级别的缓存不同的sqlsession是可以共享的。

### 2 MyBatis和Hibernate的比较

1、开发上手难度 只要会写sql，很快就会用mybatis

2、SQL优化方面Mybatis的SQL会更灵活、可控性更好、更优化。

3、移植性，MyBatis项目中所有的SQL语句都是依赖所用的数据库的，所以不同数据库类型的支持不好。Hibernate与具体[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/a63297066/article/details/_blank)的关联只需在XML文件中配置即可

4、JDBC Hibernate进行了封装，而Mybatis是原生的JDBC的，运行速度上的优势。

5、功能、特性丰富程度Hibernate提供了诸多功能和特性。要全掌握很难。

Mybatis 自身功能很有限，但Mybatis支持plugin，可以使用开源的plugin来扩展功能。

6、动态SQL Mybatis mapper xml 支持动态SQL Hibernate不支持

**实际项目关于Hibernate和Mybatis的选型：**

1、数据量：有以下情况最好选用Mybatis

如果有超过千万级别的表

如果有单次业务大批量数据提交的需求（百万条及以上的），这个尤其不建议用Hibernate

如果有单次业务大批量读取需求（百万条及以上的）(注，hibernate多表查询比较费劲，用不好很容易造成性能问题)

2、表关联复杂度

如果主要业务表的关联表超过20个（大概值），不建议使用hibernate

3、人员

如果开发成员多数不是多年使用hibernate的情况，建议使用mybatis

4、数据库对于项目的重要程度

如果项目要求对于数据库可控性好，可深度调优，用mybatis

### 3 MyBaits中批量插入

<select id="selectProduct" resultMap="Map">

SELECT \*

FROM PRODUCT

WHERE PRODUCTNO IN

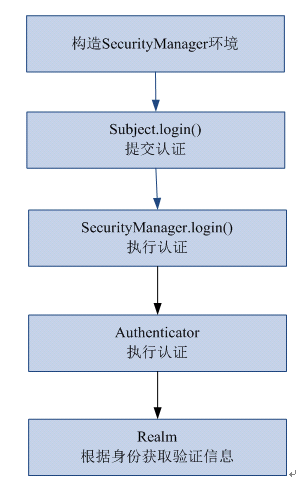
<foreach item="productNo" index="index" collection="参数的类型List或array">

#{productNo}

</foreach>

</select>

## 4 Shiro框架

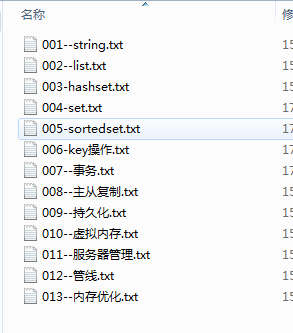


自定义realm，然后配置文件配置realm，在controller里面读取配置文件，就可以自动调用realm的授权与认证的方法。

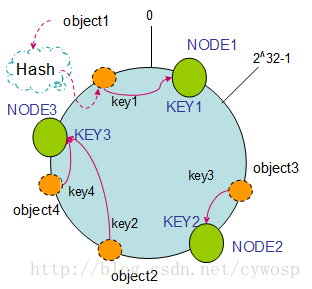
用于认证doGetAuthenticationInfo 用于授权的doGetAuthorizationInfo

管理员和用户在两张表里面，shiro默认是所有的参与认证，

## 5 redis框架



### 1 [Jedis分片连接池(分布式)](http://blog.csdn.net/lang_man_xing/article/details/38405269)



从运行结果中可以看到，不同的key被分配到不同的Redis-Server上去了。

总结：客户端jedis的一致性哈稀进行分片原理：初始化ShardedJedisPool的时候，会将上面程序中的jdsInfoList数据进行一个算法技术，主要计算依据为list中的index位置来计算

存在扩容问题和单点故障问题（主库宕机之后经过虚拟ip找到从库，从库改了之后要和主库换一下slaveofnoone，才能保证修复期的一致性）

## 6 Hadoop框架

Hadoop 分布式系统基础框架

Hadoop 实现了一个分布式文件系统 HDFS 它的核心 1> HDFS 提供存储 2> MapReduce 提供计算

# 9 分布式、高并发分布式问题

## 1 1个亿的数据要我选出最大的10个数问时间空间最省的情况用什么算法,然后时间复杂度是多少

由于(1)输入的大量数据；(2)只要前K个，对整个输入数据的保存和排序是相当的不可取的。可以利用数据结构的最小堆来处理该问题。

最小堆如图所示，对于每个非叶子节点的数值，一定不大于孩子节点的数值。这样可用含有K个节点的最小堆来保存K个目前的最大值(当然根节点是其中的最小数值)。每次有数据输入的时候可以先与根节点比较。若不大于根节点，则舍弃；否则用新数值替换根节点数值。并进行最小堆的调整。

因为要前K个嘛，如果是最大堆，只能知道最上面的是最大的，而无法判断最小的节点具体在哪儿，就无法一下子知道下一个数是不是前K大 ，如果是最小堆，上面的元素就是目前前K的，如果比它大就说明这个不是前K大，换掉重新调整最小堆就行，如果比目前前K还小，直接剔除就行。

我只回答了会考虑到两个问题，一是数据存储问题（比如一个服务器挂了，怎么保证数据不受损失），另外一个是大量用户访问时候，服务器承受问题。具体怎么解决，用什么访问，我没有给出相关的解决方案，读到此文的希望查一下

## 2 实现淘宝秒杀

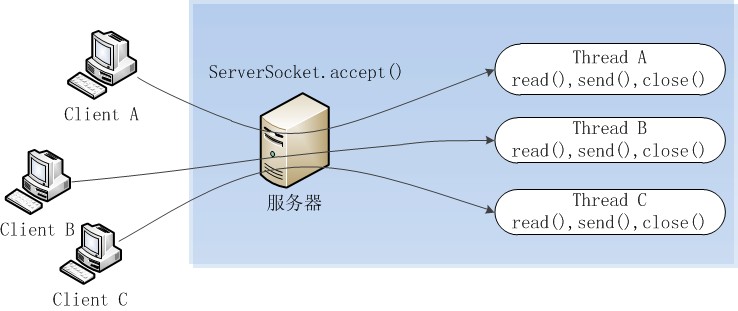
所有的秒杀请求都会入到同一个请求队列（这个请求队列实现方式有多种），当收到的请求数等于被秒杀的对象数，之后所有的请求都会被拒绝（直接返回），之前的请求都会得到相应的回复。

比如请求队列可以用redis实现。

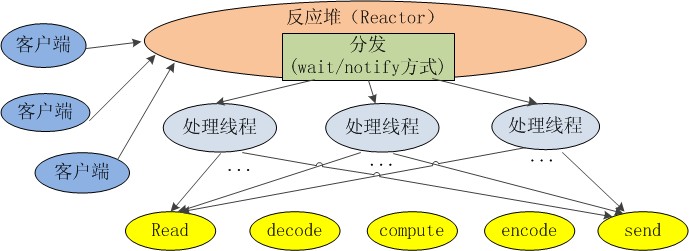
## 3 Java NIO

### **1 java NIO 和阻塞I/O的区别**

1. 阻塞I/O通信模型   
假如现在你对阻塞I/O已有了一定了解，我们知道阻塞I/O在调用InputStream.read()方法时是阻塞的，它会一直等到数据到来时（或超时）才会返回；同样，在调用ServerSocket.accept()方法时，也会一直阻塞到有客户端连接才会返回，每个客户端连接过来后，服务端都会启动一个线程去处理该客户端的请求。阻塞I/O的通信模型示意图如下：



如果你细细分析，一定会发现阻塞I/O存在一些缺点。根据阻塞I/O通信模型，我总结了它的两点缺点：  
1. 当客户端多时，会创建大量的处理线程。且每个线程都要占用栈空间和一些CPU时间  
2. 阻塞可能带来频繁的上下文切换，且大部分上下文切换可能是无意义的。  
在这种情况下非阻塞式I/O就有了它的应用前景。  
2. java NIO原理及通信模型   
Java NIO是在jdk1.4开始使用的，它既可以说成“新I/O”，也可以说成非阻塞式I/O。下面是java NIO的工作原理：  
1. 由一个专门的线程来处理所有的 IO 事件，并负责分发。   
2. 事件驱动机制：事件到的时候触发，而不是同步的去监视事件。   
3. 线程通讯：线程之间通过 wait,notify 等方式通讯。保证每次上下文切换都是有意义的。减少无谓的线程切换。阅读过一些资料之后，下面贴出我理解的java NIO的工作原理图：



### 2 IO与NIO的区别2

IO是面向流的，而NIO是面向块的。面向块的方式中，一次性可以获取或者写入一整块数据，而不需要一个字节一个字节的从流中读取。面向块的方式处理数据的速度会比流方式更快。

### 3 NIO基础

通道 Channel与缓冲器Buffer

Buffer是一个保存数据的地方，包括刚刚写入的数据，以及被读取的数据，主要用来追踪系统的读写进程。

Channel与流模式比较类似，但是，永远无法将数据直接写入到Channel或者从Channel中读取数据。需要通过Buffer与Channel交互。

### 4 NIO的读与写

读取

第一步，获取通道

FileInputStream inputStream = new FileInputStream("read.txt");FileChannel inputChannel = inputStream.getChannel();

第二步，创建缓冲器

ByteBuffer buffer = ByteBuffer.allocate(1024);

第三步，将channel中的数据读取到buffer中。相当于写入buffer

int readBytes = inputChannel.read(buffer);

写入

将数据写入到buffer中，然后再将buffer中的数据写入到channel中

FileOutputStream outputStream = new FileOutputStream("output.txt");

FileChannel outputChannel = inputStream.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(1024);

buffer.put(new String("message").getBytes());

buffer.filp();

outputChannel.write(buffer);

## 4 大型网站性能逐步优化过程：

（1）应用和数据的分离:也就是专门的应用服务器、数据库服务器、文件服务器

（2）使用缓存，专门的缓存服务器(集群) 改善数据访问性能。注意这里的数据访问二八定律。

（3）应用服务器集群，改善并发处理请求能力。

（4）数据库的读写分离改善数据库访问压力(主从热备功能)

（5）CDN加速和方向代理提高访问速度。（都是基于缓存

1. CDN部署在网络提供商的机房，使用户请求网站服务时可以从距离自己最近的网络提供商机房获取数据）

2. 反向代理则部署在网站的机房中心，当用户请求到达机房中心后，首先访问的服务器是反向代理服务器，如果反向代理服务器中缓存着用户请求的资源，就将其直接返回给用户，就不用经过应用服务器。

不管是CDN还是反向代理的目的都是为了尽早的把数据返回给用户，一方面加快用户访问速度，也能减轻后端服务器的负载压力。

（6）分布式数据库服务器

（7）业务拆分

（8）分布式服务，使用消息队列。

记住：首先是使用业务手段解决业务问题，再用技术手段解决。

## 5 高性能

（1）前端优化：

1. 使用浏览器缓存、静态资源使用独立的域名(减少cookie传输)、JS压缩

2. CDN加速：使用CDN缓存的一般都是一些静态资源，比如：图片，CSS，Script脚本，静态网页等等。这些文件访问频率极高，将其缓存在CDN可极大的改善网页的打开速度。也就是实现所谓的动静（资源）分离

3. 反向代理：配置缓存功能加快web请求，当用户第一次请求静态内容时，静态内容就被缓存在反向代理服务器上。这样当其他用户访问时就可以直接从反向代理服务器直接返回，加快web请求的响应速度，减轻后端服务器压力。

（2）应用服务器优化：缓存、集群、异步。

1. 缓存：分布式缓存存在一致性Hash问题，

2. 集群：负载均衡：负载均衡算法

3. 异步：消息队列（存在延迟）， 优点：提高可用性、加快访问速度、消除峰值

## 6 高可用：服务器硬件故障时服务依旧可用

session集群：session定位？

高可用数据：

## 7 CAP理论（一致性、可用性、分区容错性）

A是必须要保证的，所以对于CP基本是放弃的； 对于分布式系统一般是保证AP放弃强一致性C（对于分布式系统来说P是基本要求）

## 8 BASE理论：

Basically Available（基本可用）、

Soft state（软状态：允许系统中的数据存在中间状态）

Eventually consistent（数据最终一致性）

## 9 伸缩性：应用服务器集群

（1）http重定向负载均衡：利用Http的重定向功能，客户端重定向到实际应用服务器

（2）DNS域名解析负载均衡：DNS负载均衡服务器解析请求 并返回实际的IP地址，然后再次访问实际应用服务器；

常规用法是：DNS域名解析作为第一级负载均衡，获得的一组服务器是同样提供负载均衡的内网服务器，然后内网的负载均衡服务器再在内网进行第二级负载均衡，分发到实际Web应用服务器。

（3）反向代理负载均衡：反向代理服务器除了缓存资源外还可以提供负载均衡。

（4）IP负载均衡：在网络层修改请求的目标地址进行负载均衡。

（5）数据链路层负载均衡：在通信协议的数据链路层修改Mac地址进行负载均衡。

## 10 路由算法：

（1）余数Hash算法：扩容时存在大量的命中失效

（2）一致性Hash算法：使用Hash环均匀分布； 为了解决一致性hash 算法带来的负载不均衡问题，通过加入加一层虚拟层来解决。

## 11 负载均衡算法：

(1) 轮询法（Round Robin）

(2)随机法（Random）

(3)源地址Hash法（Hash）

(4)加权轮询法（Weight Round Robin）

(5)加权随机法（Weight Random）

(6)最小连接数法（Least Connections）

## 12 常用的hash算法有哪些？

1）余数Hash算法

2）一致性Hash算法

## 13 解决hash冲突的方法有哪些：

1）开放定址法

2）再哈希法

3）链地址法

4）建立一个公共的溢出区。

## 14 什么是一致性hash

一致性hash是一种分布式hash算法。分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的余数hash算法会导致大部分的原有数据失效，这是非常致命的。一致性hash算法，利用一个hash环的数据结构来实现，将集群中的不同机器通过计算其hash值均匀的分布在hash环上。当一个任意的key计算出其hash值之后，只需要在环上寻找顺时针下最近的结点。如果对于集群中任意一个机器的删除或则增加，只会使很小的一部分的数据失效，大部分数据还是有效的。当机器的增加或则删除导致的负载不均衡，我们可以通过增加一个虚拟层来解决。



## 15 怎么理解分布式锁？（淘宝秒杀为例）

分布式锁主要是解决数据的最终一致性。先来看看分布式锁应该是怎样的？

1. 分布式环境下，一个方法同一时间只能被一个机器上一个线程执行；
2. 可重入锁(避免死锁)
3. 阻塞锁；
4. 高可用的获取锁和释放锁的性能

简单点说：分布式锁就是控制分布式系统中同步访问共享资源的方式。（比如淘宝秒杀）

**# 1. 基于数据库实现：**

1. 创建一张锁表，通过操作数据表中的数据实现，表的结构是这样的：



表中的method\_name 做了唯一性约束，当有多个请求同时提交到数据库的时候，数据库会保证只有一个操作可以成功，操作成功的线程获得了锁。

1. 当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

b）注意这里的记录必须以该资源作为唯一标识(主键)

该方法有一些缺点：

1. 依赖数据库的高可用高性能，数据库挂掉就不可用了。
2. 没有设置失效时间，一旦解锁失败会导致锁永远不被释放。
3. 锁只能是非阻塞的，因为insert语句操作失败就会报错直接返回。
4. 不可重入。

缺点解决办法：

1. 数据库可以配置主从同步；
2. 数据库设置定时任务，定时清理失效记录；
3. 非阻塞？可以在方法前加while循环直到成功；

数据库的另外一种实现方案：排它锁

select ………. For update

在查询语句后面增加 **for update** ，数据库会在查询过程中给数据库表增加排他锁。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。我们可以认为获得排它锁的线程即可获得分布式锁。

**# 2.基于缓存：比如redis**

先来看看redis的一些基本命令：

顺便提一句redis本身就是单线程的。

# SETNX key value

如果key不存在，就设置key对应字符串value。在这种情况下，该命令和SET一样。当key已经存在时，就不做任何操作。SETNX是”SET if Not exists”。

# expire KEY seconds   
设置key的过期时间。如果key已过期，将会被自动删除。   
# del KEY

删除key

1. 怎么实现加锁：key是于商品ID相关的字符串来唯一标识，value不重要，只要这个唯一的key-value存在，就表示这个商品已经上锁。
2. 怎么释放锁：redis里面删除相应的键值对
3. 阻塞实现，如果已经上锁就阻塞，不过可以在客户端使用自旋锁轮询。

设置过期时间解决异常问题。

# 10 数据库（优化\*）

## 1 sql复习

内链接

两张表如果有外键关系可以使用内链接，因为通过内链接，每一条只能返回单条记录

只要在关联表中查询出一条记录就可以使用内链接

外链接

只有部分记录可以从关联表查询到，主查询表想要显示所有记录，只能和关联表通过外链接

左外链接 右外链接

left join left左边是主查询表，left右边是关联查询表

right join right右边是主查询表，right左边是关联查询表

子查询

通过关联表的主键关联查询绝对没有问题，也可以不通过关联表的主键查询（但是子查询关联查询的结果集只能有一条记录）

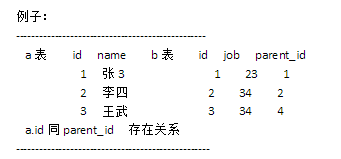
在朱表上作子查询，主表记录会全部显示

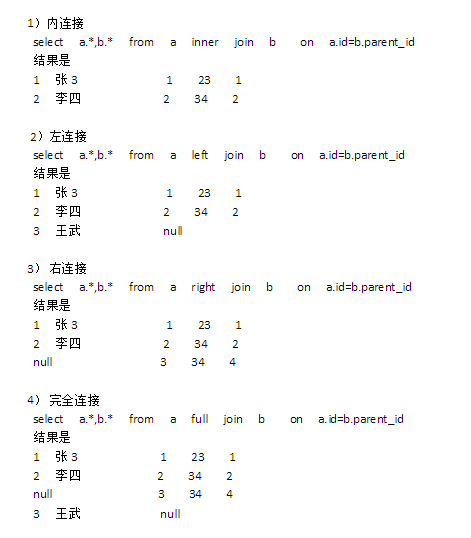
嵌套表查询

将嵌套的结果集当成一个类，嵌套子查询中列不能重复的

嵌套表的层级有限制

举例：





## 2 事物的特性

事务四大特性(简称ACID)

1、原子性(Atomicity)：事务中的全部操作在[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/hao123huo/article/details/_blank)中是不可分割的，要么全部完成，要么均不执行。

2、一致性(Consistency)：几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。

3、隔离性(Isolation)：事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。

4、持久性(Durability)：对于任意已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

## 3 mysql的索引 elasticsearch的倒排索引

在列上加索引时事有条件的：  
 1、经常被查询的列  
 2、order by子句中使用的列  
 3、是外键或者主键的列  
 4、列是唯一的列  
 5、两个或多个列经常同时出现在where子句中或者连接条件中

elasticsearch的倒排索引

一个倒排索引包含一系列不同的单词，这些单词出现在任何一个文档，

对于每个单词，对应着所有它出现的文档。

比如说，我们有2个文档，每个文档有一个conteng字段。

内容如下：

“ The quick brown fox jumped over the lazy dog”

“ Quick brown foxes leap over lazy dogs in summer”

为了创建倒排索引，

我们首先对每个字段进行分词，我们称之为terms或者tokens,创建了一些列有序列表，

然后列举了每个单词所出现的文档，结果如下：

Term      Doc\_1  Doc\_2

-------------------------

Quick   |       |  X

The     |   X   |

brown   |   X   |  X

dog     |   X   |

dogs    |       |  X

fox     |   X   |

foxes   |       |  X

in      |       |  X

jumped  |   X   |

lazy    |   X   |  X

leap    |       |  X

over    |   X   |  X

quick   |   X   |

summer  |       |  X

the     |   X   |

------------------------

现在，如果我们想搜索"quick brown",我们只需要找到每个单词出现的文档。

Term      Doc\_1  Doc\_2

-------------------------

brown   |   X   |  X

quick   |   X   |

------------------------

Total   |   2   |  1

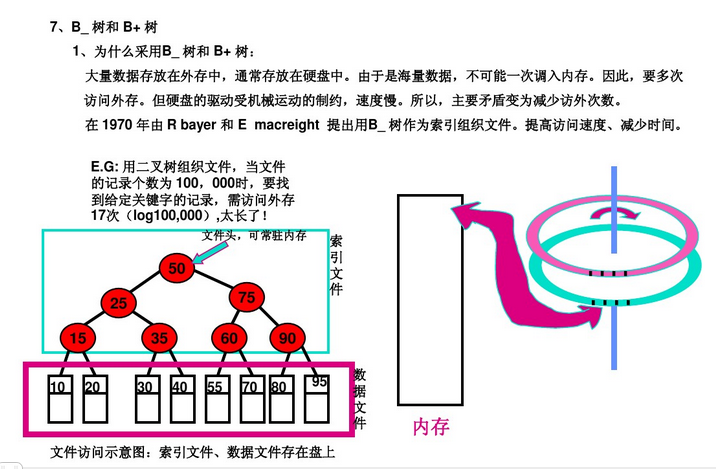
两个文档都匹配，但是第一个文档有更高的匹配度，

如果我们采用一个简单的相似算法，我们可以说，第一个文档比第2个文档有更高的匹配度。

也更相关。

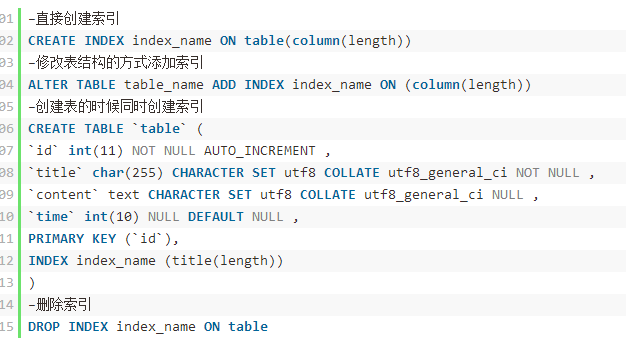
索引分为聚簇索引和非聚簇索引两种，聚簇索引是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

**MySQL索引的类型（B树，hash索引（等值的链接条件（大于等于，小于等于，或者等于）））**



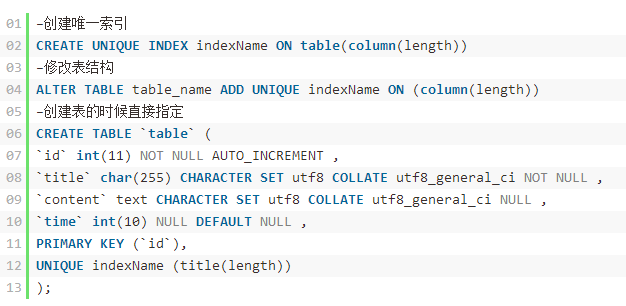
**1. 普通索引**

这是最基本的索引，它没有任何限制，比如上文中为title字段创建的索引就是一个普通索引，MyIASM中默认的BTREE类型的索引，也是我们大多数情况下用到的索引。



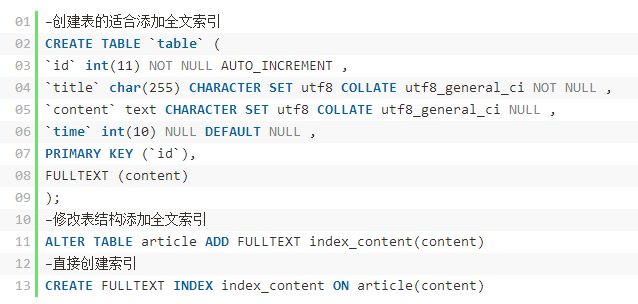
**2. 唯一索引**

与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。



**3. 全文索引（FULLTEXT）**

MySQL从3.23.23版开始支持全文索引和全文检索，FULLTEXT索引仅可用于 MyISAM 表；他们可以从CHAR、VARCHAR或TEXT列中作为CREATE TABLE语句的一部分被创建，或是随后使用ALTER TABLE 或CREATE INDEX被添加。////对于较大的数据集，将你的资料输入一个没有FULLTEXT索引的表中，然后创建索引，其速度比把资料输入现有FULLTEXT索引的速度更为快。不过切记对于大容量的数据表，生成全文索引是一个非常消耗时间非常消耗硬盘空间的做法。



**4. 单列索引、多列索引**

多个单列索引与单个多列索引的查询效果不同，因为执行查询时，MySQL只能使用一个索引，会从多个索引中选择一个限制最为严格的索引。

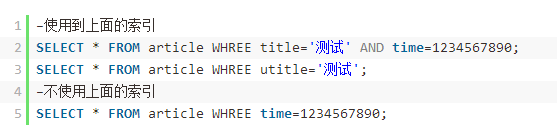
**5. 组合索引（最左前缀）**

平时用的SQL查询语句一般都有比较多的限制条件，所以为了进一步榨取MySQL的效率，就要考虑建立组合索引。例如上表中针对title和time建立一个组合索引：ALTER TABLE article ADD INDEX index\_titme\_time (title(50),time(10))。建立这样的组合索引，其实是相当于分别建立了下面两组组合索引：

–title,time

–title

为什么没有time这样的组合索引呢？这是因为MySQL组合索引“最左前缀”的结果。简单的理解就是只从最左面的开始组合。并不是只要包含这两列的查询都会用到该组合索引，如下面的几个SQL所示：



**MySQL索引的优化**

上面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快。索引只是提高效率的一个因素，如果你的MySQL有[大数据](http://lib.csdn.net/base/hadoop" \o "Hadoop知识库" \t "http://blog.csdn.net/xluren/article/details/_blank)量的表，就需要花时间研究建立最优秀的索引，或优化查询语句。下面是一些总结以及收藏的MySQL索引的注意事项和优化方法。

**1. 何时使用聚集索引或非聚集索引？**

|  |  |  |
| --- | --- | --- |
| **动作描述** | **使用聚集索引** | **使用非聚集索引** |
| 列经常被分组排序 | 使用 | 使用 |
| 返回某范围内的数据 | 使用 | 不使用 |
| 一个或极少不同值 | 不使用 | 不使用 |
| 小数目的不同值 | 使用 | 不使用 |
| 大数目的不同值 | 不使用 | 使用 |
| 频繁更新的列 | 不使用 | 使用 |
| 外键列 | 使用 | 使用 |
| 主键列 | 使用 | 使用 |
| 频繁修改索引列 | 不使用 | 使用 |

事实上，我们可以通过前面聚集索引和非聚集索引的定义的例子来理解上表。如：返回某范围内的数据一项。比如您的某个表有一个时间列，恰好您把聚合索引建立在了该列，这时您查询2004年1月1日至2004年10月1日之间的全部数据时，这个速度就将是很快的，因为您的这本字典正文是按日期进行排序的，聚类索引只需要找到要检索的所有数据中的开头和结尾数据即可；而不像非聚集索引，必须先查到目录中查到每一项数据对应的页码，然后再根据页码查到具体内容。其实这个具体用法我还不是很理解，只能等待后期的项目开发中慢慢学学了。

**2. 索引不会包含有NULL值的列**

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

**3. 使用短索引**

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的列，如果在前10个或20个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

**4. 索引列排序**

MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

**5. like语句操作**

一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

**6. 不要在列上进行运算**

例如：select \* from users where YEAR(adddate)<2007，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成：select \* from users where adddate<’2007-01-01′。关于这一点可以围观：[一个单引号引发的MYSQL性能损失。](http://www.zendstudio.net/archives/single-quotes-or-no-single-quotes-in-sql-query" \t "http://blog.csdn.net/xluren/article/details/_blank)

很明显，不使用单引号没有用上主索引，并进行了全表扫描，使用单引号就能使用上索引了。

最后总结一下，MySQL只对一下操作符才使用索引：<,<=,=,>,>=,between,in,以及某些时候的like(不以通配符%或\_开头的情形)。而理论上每张表里面最多可创建16个索引，不过除非是数据量真的很多，否则过多的使用索引也不是那么好玩的，比如我刚才针对text类型的字段创建索引的时候，系统差点就卡死了。

## 4 数据库事务的隔离级别

**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/fg2006/article/details/_blank)**事务的隔离级别有4个，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable（可串行化的），这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

√: 可能出现    ×: 不会出现

|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复读 | 幻读 |
| Read uncommitted（大多数默认） | √ | √ | √ |
| Read committed | × | √ | √ |
| Repeatable read（MySQL默认） | × | × | √ |
| Serializable | × | × | × |

## 5 数据库回滚

rollback 回滚的意思。 就是数据库里做修改后 （ update ,insert , delete）未commit 之前 使用rollback 可以恢复数据到修改之前。

## 6 explain命令 如何查看mysql使用索引来执行select语句

对mysql explain讲的比较清楚的

在 explain的帮助下，您就知道什么时候该给表添加索引，以使用索引来查找记录从而让select 运行更快。  
如果由于不恰当使用索引而引起一些问题的话，可以运行 analyze table来更新该表的统计信息，例如键的基数，它能帮您在优化方面做出更好的选择。

explain 返回了一行记录，它包括了 select语句中用到的各个表的信息。这些表在结果中按照mysql即将执行的查询中读取的顺序列出来。mysql用一次扫描多次连接（single- sweep,multi-join）的方法来解决连接。这意味着mysql从第一个表中读取一条记录，然后在第二个表中查找到对应的记录，然后在第三个表中查找，依次类推。当所有的表都扫描完了，它输出选择的字段并且回溯所有的表，直到找不到为止，因为有的表中可能有多条匹配的记录下一条记录将从该表读取，再从下一个表开始继续处理。  
在mysql version 4.1中，explain输出的结果格式改变了，使得它更适合例如 union语句、子查询以及派生表的结构。更令人注意的是，它新增了2个字段： id和 select\_type。当你使用早于mysql4.1的版本就看不到这些字段了。

7 数据库设计时，字段可以选择默认值或者null，为何选择默认值

null值在有些存储引擎中按最大字节数存储，占空间，null不能被索引，影响性能；

## 8 大规模的delete或insert操作是否会对引起表锁定，如何解决

分批次delete或insert

## 9 怎么在SQL层面做一些优化

sql层面优化：尽量不要用\*,in和not in等  
数据库层面优化：走索引，建分区表。

SQL优化基本就是一些常识性的东西，比如不能用SELELCT \*，少用DISTINCT、GROUP BY之类的命令。

SQL优化步骤

1.   使用pt-query-digest分析慢日志，拿到调用频率高并且执行时间长的语句。

2.   获取语句相关表的信息，以及explain等相关信息，一般收集如下信息

* l  执行计划：explain query;
* l  详细执行计划：explain extended query;show warnings;
* l  相关表信息：show create table test1;
* l  语句执行开销：show profile for query num;
* l  数据量：select count(0) from table;
* l  相关表的索引信息：show index for table;
* l  my.cnf查询相关参数如各类buffer等

3.   进行初步分析，诊断SQL语句，查看在SQL层面能否带来提升，一般应对手段如下

* 条件上拉、下推
* 子查询合并、展开
* 添加索引、优化索引
* distinct、group by、orderby尽量利用索引减少文件排序
* 子查询尽量转换为join，最好消除
* 尽量减少聚合函数，将复杂查询转换为多表链接
* 尽量优化等价谓词，尽可能利用索引
* join的一些优化，连接消除、去除中间表等

4.   SQL无优化空间考虑从业务逻辑层面入手。常用手段如下

* l  增加冗余字段
* l  拆分表（水平/竖直）
* l  使用分布式
* l  将大事务拆分为多个小事务
* l  精简数据库的设计，减少使用存储过程、触发器等

5.   对硬件进行优化，升级服务器硬件。

3、4步骤可与调整系统参数相结合，具体使用方法视具体业务而定。

## 10 **[MySQL性能优化——易实现的MySQL优化方案汇总](http://blog.csdn.net/zhangliangzi/article/details/52329355)**

1. **索引优化**

1、合理使用索引，在经常查询而不经常增删改操作的字段加索引，一个表上的索引不应该超过6个。

2、Order by与group by后应直接使用字段，而且字段应该是索引字段。

3、索引字段长度应较短而长度固定。

4、索引字段重复不能过多。

**5、Hash索引与BTree索引区别（MyISAM与InnoDB不支持Hash索引）**

(1)、BTree索引使用多路搜索树的**[数据结构](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/zhangliangzi/article/details/_blank)**，可以减少定位的中间过程；综合效率较高，默认使用的索引。

(2)、Hash索引使用Hash**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/zhangliangzi/article/details/_blank)**构建索引；精确的等值查询一次定位，效率极高，但特别不适合范围查询；使用Hash的复合索引是把复合索引键共同计算hash值，故不能单独使用。

**6、会导致引擎放弃使用索引，改为进行全表的几种情况，都要在开发中尽量避免出现！**

(1)、where子句中使用like关键字时，前置百分号会导致索引失效（起始字符不确定都会失效）。如：select id from test where name like "%吉坤"。

(2)、where子句中使用is null或is not null时，因为null值会被自动从索引中排除，索引一般不会建立在有空值的列上。

(3)、where子句中使用or关键字时，or左右字段如果存在一个没有索引，有索引字段也会失效；而且即使都有索引，因为二者的索引存储顺序并不一致，效率还不如顺序全表扫描，这时引擎有可能放弃使用索引，所以要慎用or。

(4)、where子句中使用in或not in关键字时，会导致全表扫描，能使用exists或between and替代就不使用in。

(5)、where子句中使用!=操作符时，将放弃使用索引，因为范围不确定，使用索引效率不高，会被引擎自动改为全表扫描；

(6)、where子句中应尽量避免对索引字段操作（表达式操作或函数操作），比如select id from test where num/2 = 100应改为num = 200。

(7)、在使用复合索引时，查询时必须使用到索引的第一个字段，否则索引失效；并且应尽量让字段顺序与索引顺序一致。

(8)、查询时必须使用正确的数据类型。数据库包含了自动了类型转换，比如纯数字赋值给字符串字段时可以被自动转换，但如果查询时不加引号查询，会导致引擎忽略索引。

1. **表结构优化**
2. 设计符合第三范式的表结构。

2、尽量使用数字型字段，提高数据比对效率。

3、对定长、MD5哈希码、长度较短的字段使用char类型，提高效率；对边长而且可能较长字段使用varchar类型，节约内存。

4、适当的进行水平分割与垂直分割，比如当表列数过多时，就将一部分列移出到另一张表中。

**关于水平分割与垂直分割表详解：**

**水平分割表：**一种是当多个过程频繁访问数据表的不同行时，水平分割表，并消除新表中的冗余数据列；若个别过程要访问整个数据，则要用连接\*作，这也无妨分割表；典型案例是电信话单按月分割存放。另一种是当主要过程要重复访问部分行时，最好将被重复访问的这些行单独形成子集表（冗余储存），这在不考虑磁盘空间开销时显得十分重要；但在分割表以后，增加了维护难度，要用触发器立即更新、或存储过程或应用代码批量更新，这也会增加额外的磁盘I/O开销。

水平分割会给应用增加复杂度，它通常在查询时需要多个表名，查询所有数据需要union操作。在许多数据库应用中，这种复杂性会超过它带来的优点，因为只要索引关键字不大，则在索引用于查询时，表中增加两到三倍数据量，查询时也就增加读一个索引层的磁盘次数。

**垂直分割表（不破坏第三范式）**：一种是当多个过程频繁访问表的不同列时，可将表垂直分成几个表，减少磁盘I/O（每行的数据列少，每页存的数据行就多，相应占用的页就少），更新时不必考虑锁，没有冗余数据。缺点是要在插入或删除数据时要考虑数据的完整性，用存储过程维护。另一种是当主要过程反复访问部分列时，最好将这部分被频繁访问的列数据单独存为一个子集表（冗余储存），这在不考虑磁盘空间开销时显得十分重要；但这增加了重叠列的维护难度，要用触发器立即更新、或存储过程或应用代码批量更新，这也会增加额外的磁盘I/O开销。垂直分割表可以达到最大化利用Cache的目的。

垂直分割可以使得数据行变小(因为列少了，一行数据就变小)，一个数据页就能存放更多的数据，在查询时就会减少I/O 次数。其缺点是需要管理冗余列，查询所有数据需要join操作 。

1. **临时表优化——临时表常常用于排序或分组，所以Order By与Group By后的字段尽量使用索引**

临时表可以根据实际需求使用，但要尽力避免磁盘临时表的生成。

**1、常见的会产生内存临时表的情况**

1、UNION查询。  
2、子查询（所以我们一般用join代替子查询）。  
3、join查询中，如果order by 与 group by如果使用的不都是第一张表上的字段，就会产生临时表。  
4、order by中使用distinct函数。

**2、常见的会产生磁盘临时表的情况**

1、数据表中包含BLOB/TEXT列。  
2、Group by、distinct、union查询中包含超过512字节的列。

1. **其他优化**

1、不使用Select \*，只查询需要的字段。

2、在只查询一条字段时，limit 1。

3、避免大事务操作，提高并发能力。

4、在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

5、尽量少使用游标。

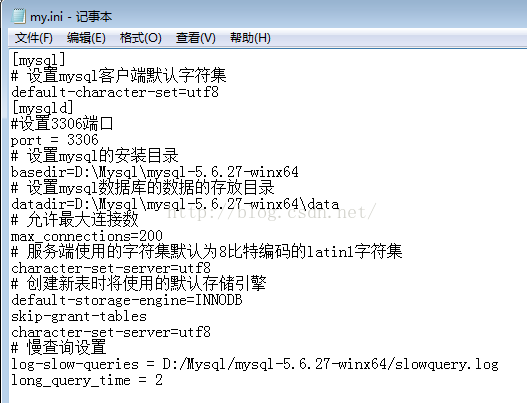
6、多去关注慢查询，总有我们提前考虑不到的问题，出现了就去解决它！

**附慢查询开启方式：**

在**[MySQL](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/zhangliangzi/article/details/_blank)**安装目录下，找到my.ini配置文件，在mysqld下加上如下配置：

log-slow-queries = D:/Mysql/mysql-5.6.27-winx64/slowquery.log

long\_query\_time = 2



## 11 三范式

第一：列满足原子性，即求每一列都不允许再次拆分（列不可分）  
如：表中若有“地址"列。则地址还可以拆分为：国家、省份、城市等这些列，就说明地址这列还可拆分，则不满足第三范式  
第二：满足第一的基础上，除主键以外每一列都依赖于主键（不能有部分依赖）  
如：一张表是描述学员的；有学员编号，学校名称；其中学校名称和学员编号没有依赖关系。应把学校名称放在学校表中  
第三：满足第一第二的基础上，除主键以外的列都直接依赖主键。（不能有传递依赖）

如：城镇表依赖市表，市表依赖省表，则可以推断出城镇依赖省表；现在城镇表和省表之间就是间接相关

## 12 select for update

**MySQL SELECT ... FOR UPDATE 的Row Lock 与Table Lock**  
上面介绍过SELECT ... FOR UPDATE 的用法，不过锁定(Lock)的数据是判别就得要注意一下了。由于InnoDB 预设是Row-Level Lock，所以只有「明确」的指定主键，MySQL 才会执行Row lock (只锁住被选取的数据) ，否则MySQL 将会执行Table Lock (将整个数据表单给锁住)。

举个例子:  
假设有个表单products ，里面有id 跟name 二个栏位，id 是主键。  
例1: (明确指定主键，并且有此数据，row lock)

SELECT \* FROM products WHERE id='3' FOR UPDATE;

例2: (明确指定主键，若查无此数据，无lock)

SELECT \* FROM products WHERE id='-1' FOR UPDATE;

例2: (无主键，table lock)

SELECT \* FROM products WHERE name='Mouse' FOR UPDATE;

例3: (主键不明确，table lock)

SELECT \* FROM products WHERE id<>'3' FOR UPDATE;

例4: (主键不明确，table lock)

SELECT \* FROM products WHERE id LIKE '3' FOR UPDATE;

**乐观所和悲观锁策略**  
悲观锁：在读取数据时锁住那几行，其他对这几行的更新需要等到悲观锁结束时才能继续 。  
乐观所：读取数据时不锁，更新时检查是否数据已经被更新过，如果是则取消当前更新，一般在悲观锁的等待时间过长而不能接受时我们才会选择乐观锁。

## 13 MySQL创建用户与授权及撤销用户权限方法

**一, 创建用户**

命令:CREATE USER 'username'@'host' IDENTIFIED BY 'password';

**二,授权**

命令:GRANT privileges ON databasename.tablename TO 'username'@'host'

**三.设置与更改用户密码**

命令:SET PASSWORD FOR 'username'@'host' = PASSWORD('newpassword');如果是当前登陆用户用SET PASSWORD = PASSWORD("newpassword");

**四.撤销用户权限**

命令: REVOKE privilege ON databasename.tablename FROM 'username'@'host';

**五.删除用户**

命令: DROP USER 'username'@'host';

## 14 [prepareStatement与Statement的区别](http://www.cnblogs.com/plume/archive/2010/03/06/1711715.html" \t "http://blog.csdn.net/zsm653983/article/details/_blank)

1.区别：（用法）

stmt=conn.CreateStatement();  
resultSet rs=stmt.executeQuery(sql);  
上面是statement的用法  
============================  
下面是PrepareStatement的用法  
ptmt=conn.PreparedStatement(sql);  
resultSet rs=ptmt.executeQuery();  
================================  
Statement 是PreparedStatement的父类  
还有就是sql放置的位置不同 。

在开发中一般用PrepareStatement

stmt.executeUpdate("insert into tb\_name (col1,col2,col2,col4) values ('"+var1+"','"+var2+"',"+var3+",'"+var4+"')");  
  
perstmt = con.prepareStatement("insert into tb\_name (col1,col2,col2,col4) values (?,?,?,?)");  
perstmt.setString(1,var1);  
perstmt.setString(2,var2);  
perstmt.setString(3,var3);  
perstmt.setString(4,var4);  
perstmt.executeUpdate();

二.PreparedStatement尽最大可能提高性能.

当然并不是所以预编译语句都一定会被缓存,数据库本身会用一种策略,比如使用频度等因素来决定什么时候不再缓存已有的预编译结果.以保存有更多的空间存储新的预编译语句.

三.最重要的一点是极大地提高了安全性.

## 15 数据库的索引是如何实现的，主键索引和联合索引数据结构有什么区别

主键是表中的一个或多个字段，它的值用于惟一地标识表中的某一条记录.且不能为空; 索引是对数据库表中一列或多列的值进行排序的一种结构，只有当经常查询索引列中的数据时，才需要在表上创建索引，使用索引可快速访问数据库表中的特定信息。 索引占用磁盘空间，并且降低添加、删除和更新行的速度。当然索引也有好处就是查询速度快，它利还是大于弊的所以请慎重使用索引。 比如：一个学生表（t\_stu ）有1000条数据，给它id列建个主键和索引， 你想查询id=1000；的这条信息，如果没有索引，它就一条一条的比对查找，系统运行1000次才找到，要是创建了索引，你查询id=1000的这条信息，系统只运行一次就找到了。

索引是b＋树 主键索引可能是聚集索引 多列索引的话还是一棵树  
只是如果第一列相同的话，就根据第二列的值来排序

where后面的条件 顺序影响执行效率吗

如果where后面的条件次序跟多列索引的次序不同，就不能充分利用多列索因

或者比方说有两个单列索引，一个表有a,b,c这三列，b列和c列有单列索引，然后其中b=1的记录有100条，c=1的记录有1000000条，但是b=1&&c=1的记录只有一条

现在我们select \* from table where b=1 and c = 1，用哪个索引，就比较讲究

## 16 数据库连接池

1 dbcp  
dbcp可能是使用最多的开源连接池，原因大概是因为配置方便，而且很多开源和tomcat应用例子都是使用的这个连接池吧。这个连接池可以设置最大和最小连接，连接等待时间等，基本功能都有。使用评价：在具体项目应用中，发现此连接池的持续运行的稳定性还是可以，不过速度稍慢，在大并发量的压力下稳定性有所下降，此外不提供连接池监控  
2 c3p0  
c3p0是另外一个开源的连接池，在业界也是比较有名的，这个连接池可以设置最大和最小连接，连接等待时间等，基本功能都有。使用评价：在具体项目应用中，发现此连接池的持续运行的稳定性相当不错，在大并发量的压力下稳定性也有一定保证，此外不提供连接池监控。  
3 proxool  
proxool这个连接池可能用到的人比较少，但也有一定知名度，这个连接池可以设置最大和最小连接，连接等待时间等，基本功能都有。使用评价：在具体项目应用中，发现此连接池的持续运行的稳定性有一定问题，有一个需要长时间跑批的任务场景任务，同样的代码

## 17 MySQL中的锁概念

[MySQL](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/hsd2012/article/details/_blank)中不同的存储引擎支持不同的锁机制。比如MyISAM和MEMORY存储引擎采用的表级锁，BDB采用的是页面锁，也支持表级锁，InnoDB存储引擎既支持行级锁，也支持表级锁，默认情况下采用行级锁。

Mysql3中锁特性如下：

表级锁：开销小，加锁块；不会出现死锁，锁定粒度大，发生锁冲突的概率最高，并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发性也最高。

页面锁：开销和加锁界于表锁和行锁之间，会出现死锁；锁定粒度界与表锁和行锁之间，并发一般。

## 18 B+树在数据库中的应用

主要是在所有的叶子结点中增加了指向下一个叶子节点的指针，因此InnoDB建议为大部分表使用默认自增的主键作为主索引。

为什么使用B+树?言简意赅，就是因为：

1. B+树空间利用率更高，可减少I/O次数

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗。而因为B+树的内部节点只是作为索引使用，而不像B-树那样每个节点都需要存储硬盘指针。

也就是说：B+树中每个非叶节点没有指向某个关键字具体信息的指针，所以每一个节点可以存放更多的关键字数量，即一次性读入内存所需要查找的关键字也就越多，减少了I/O操作。

2.增删文件（节点）时，效率更高

因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

1. B+树的查询效率更加稳定

因为B+树的每次查询过程中，都需要遍历从根节点到叶子节点的某条路径。所有关键字的查询路径长度相同，导致每一次查询的效率相当。

## 19 实践中如何优化MySQL

我当时是按以下四条依次回答的，他们四条从效果上第一条影响最大，后面越来越小。

① SQL语句及索引的优化

② 数据库表结构的优化

③ 系统配置的优化

④ 硬件的优化

## 20 什么情况下设置了索引但无法使用

① 以“%”开头的LIKE语句，模糊匹配

② OR语句前后没有同时使用索引

③ 数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）

## 21 SQL语句的优化

order by要怎么处理

alter尽量将多次合并为一次

insert和delete也需要合并

## 22 InnoDB索引和MyISAM索引的区别（\*）

一是主索引的区别，InnoDB的数据文件本身就是索引文件。而MyISAM的索引和数据是分开的。

二是辅助索引的区别：InnoDB的辅助索引data域存储相应记录主键的值而不是地址。而MyISAM的辅助索引和主索引没有多大区别。

# 11 操作系统 Linux

## Linux

### 1 进程间通信的方式有哪几种？

1）管道2）有名管道3）信号量4）消息队列5）共享内存；6）套接字socket

### 2 Linux下如何进行进程调度的？

主要有三种调度策略：

1.分时调度策略

2.实时调度策略—FIFO

3.实时调度策略—时间片轮转。

### **3 Linux 交换空间 物理内存和虚拟内存** 我们知道，直接从物理内存读写数据要比从硬盘读写数据要快的多，因此，我们希望所有数据的读取和写入都在内存完成，而内存是有限的，这样就引出了物理内存与虚拟内存的概念。 物理内存就是系统硬件提供的内存大小，是真正的内存，相对于物理内存，在linux下还有一个虚拟内存的概念，虚拟内存就是为了满足物理内存的不足而提出的策略，它是利用磁盘空间虚拟出的一块逻辑内存，用作虚拟内存的磁盘空间被称为交换空间（Swap Space）。 作为物理内存的扩展，linux会在物理内存不足时，使用交换分区的虚拟内存，更详细的说，就是内核会将暂时不用的内存块信息写到交换空间，这样以来，物理内存得到了释放，这块内存就可以用于其它目的，当需要用到原始的内容时，这些信息会被重新从交换空间读入物理内存。 Linux的内存管理采取的是分页存取机制，为了保证物理内存能得到充分的利用，内核会在适当的时候将物理内存中不经常使用的数据块自动交换到虚拟内存中，而将经常使用的信息保留到物理内存。

### 4 内存的监控

top命令能够实时显示系统中各个进程的资源占用状况，类似于Windows下的资源管理器。

作为一名Linux系统管理员，监控内存的使用状态是非常重要的，通过监控有助于了解内存的使用状态，比如内存占用是否正常，内存是否紧缺等等，监控内存最常使用的命令有free、[top](http://www.linuxeye.com/command/top.html" \t "http://www.linuxeye.com/Linux/_blank)等，下面是某个系统free的输出：

每个选项的含义：  
第一行：  
total：物理内存的总大小  
used：已经使用的物理内存大小  
free：空闲的物理内存大小  
shared：多个进程共享的内存大小  
buffers/cached：磁盘缓存的大小  
  
第二行Mem：代表物理内存使用情况  
第三行(-/+ buffers/cached)：代表磁盘缓存使用状态  
第四行：Swap表示交换空间内存使用状态  
  
free命令输出的内存状态，可以通过两个角度来查看：一个是从内核的角度来看，一个是从应用层的角度来看的

### 5 Linux常用命令

1. 文件目录相关：ls、cd、pwd、mkdir、rmdir、mv、cp、touch、cat、
2. 文件查找类型的：which、find
3. 压缩和解压：tar、unzip、
4. 文件权限：chmod
5. 进程：ps、kill
6. 网络：ssh、ping、ipconfig、netstat、scp

7）Linux 获取上一条命令的返回值。 一般情况下，命令执行正确，返回0, 命令异常，返回其他值。

命令： echo $?

8）Windows 获取上一条命令的返回值。 一般情况下，命令执行正确，返回0, 命令异常，返回其他值。

命令：echo %ERRORLEVEL%

### 6 Linux下查看网络端口状态

netstat命令：用于显示与IP、TCP、UDP和ICMP协议相关的统计数据，一般用于检验本机各端口的网络连接情况。netstat是在内核中访问网络及相关信息的程序，它能提供TCP连接，TCP和UDP监听，进程内存管理的相关报告。

### 7 Linux shell 定期删除七天之前的日志文件

**1.删除文件命令：**

find 对应目录 -mtime +天数 -name "文件名" -exec rm -rf {} \;

实例命令：

find /opt/soft/log/ -mtime +30 -name "\*.log" -exec rm -rf {} \;

说明：

将/opt/soft/log/目录下所有30天前带".log"的文件删除。具体参数说明如下：

find：linux的查找命令，用户查找指定条件的文件；

/opt/soft/log/：想要进行清理的任意目录；

-mtime：标准语句写法；

+30：查找30天前的文件，这里用数字代表天数；

"\*.log"：希望查找的数据类型，"\*.jpg"表示查找扩展名为jpg的所有文件，"\*"表示查找所有文件，这个可以灵活运用，举一反三；

-exec：固定写法；

rm -rf：强制删除文件，包括目录；

{} \; ：固定写法，一对大括号+空格+\+;

**2.计划任务：**

若嫌每次手动执行语句太麻烦，可以将这小语句写到一个可执行shell脚本文件中，再设置cron调度执行，那就可以让系统自动去清理相关文件。

<http://www.cnblogs.com/peida/archive/2013/03/25/2980121.html>

### 8 [Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "http://blog.csdn.net/zhonglinzhang/article/details/_blank)内核主要有几种内核锁

主要有自旋锁和信号量

自旋锁是防止多处理器并发而引入的一种锁，在内核中大量应用于中断处理

### 9 Linux用户模式和内核模式

用户模式是受限模式，对内存和硬件的访问必须通过系统调用实现

内核模式是一种高特权模式，能直接访问内存和硬件

## 操作系统

### 1 操作系统寻址方式

立即数寻址

寄存器寻址

直接寻址

间接寻址

相对寻址

基址变址寻址

相对基址变址寻址

### 2 外部碎片和内部碎片

外部碎片：由于大量信息先后写入，置换，删除而形成的空间碎片；

6间仓库，装了1-5，然后清空了4，剩下4和6是空的，现在需要连续的两间仓库来存储信息，是存不了的，造成了碎片。

内部碎片：由于存储信息的容量与最小存储空间单位不完全相符而造成的空间碎片。

6间仓库，管理仓库的最小空间单位是间。现在存了2.5间，3号仓库只存了一半，那么3号仓库剩下的半间就不能用了，因为管理仓库的最小空间单位是间，造成了碎片。

### 3 操作系统什么情况下会死锁

一般都是存在资源竞争：比如获取一个资源必须获取两把锁A和B。线程甲已经获取了锁A，线程乙已经获取到了锁B，但是线程甲要获取锁B，线程乙要获取锁A，这时候互相等待，就会导致死锁。

### 4 进程和线程

进程是资源分配的基本单位，所有与该进程相关的资源，都被记录在进程控制块PCB中，以表示该进程拥有这些资料并正在使用它们。

线程是轻量级的进程，是独立运行和独立调度的基本单位，是一个基本的CPU执行单元。它与资源分配无关，线程本身不拥有系统资源，但是它可与同属一个进程的其他线程共享进程所拥有的全部资源。通常在一个进程中可以包含若干个线程。

进程和线程的区别可以归纳为4点：

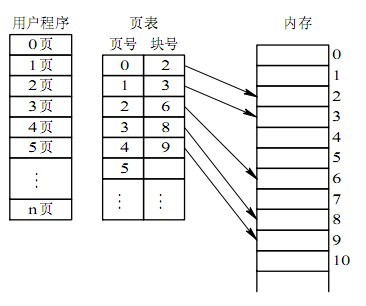
1. 地址空间和其他资源：进程间相互独立，同一进程间的各线程共享；
2. 通信：进程间通信IPC，线程间可以直接读写进程数据段进行通信；
3. 调度和切换：线程上下文切换比进程上下文切换快得多；
4. 在多线程操作系统中，进程不是一个可执行的实体。（？）

### 5 常用进程调度算法

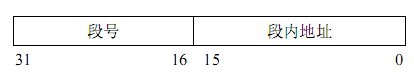
1. 先来先服务；
2. 短进程优先；（从就绪队列中选出一个估计时间最短的进程，分配CPU，使它立即执行并执行到完成，或者发生过某时间而被阻塞时再重新调度）
3. 高优先权优先调度；（抢占式，非抢占式）
4. 高响应比优先；（优先权=（等待时间+要求服务时间）/要求服务时间）
5. 时间片轮转法。

### 6 操作系统内存管理

页式管理地址



段式管理地址



<http://blog.csdn.net/hguisu/article/details/5713164>

### 7 [Windows进程间各种通信方式浅谈](http://blog.csdn.net/left_la/article/details/11564539)

1 文件映射

应用程序有三种方法来使多个进程共享一个文件映射对象。   
(1)继承：第一个进程建立文件映射对象，它的子进程继承该对象的句柄。   
(2)命名文件映射：第一个进程在建立文件映射对象时可以给该对象指定一个名字(可与文件名不同)。第二个进程可通过这个名字打开此文件映射对象。   
另外，第一个进程也可以通过一些其它IPC机制(有名管道、邮件槽等)把名字传给第二个进程。  
(3)句柄复制：第一个进程建立文件映射对象，然后通过其它IPC机制(有名管道、邮件槽等)把对象句柄传递给第二个进程。

2 共享内存

Win32 API中共享内存(Shared Memory)实际就是文件映射的一种特殊情况。进程在创建文件映射对象时用0xFFFFFFFF来代替 文件句柄(HANDLE)，由于共享内存是用 文件映射实现的，所以它也有较好的安全性，也只能运行于同一计算机上的进程之间。

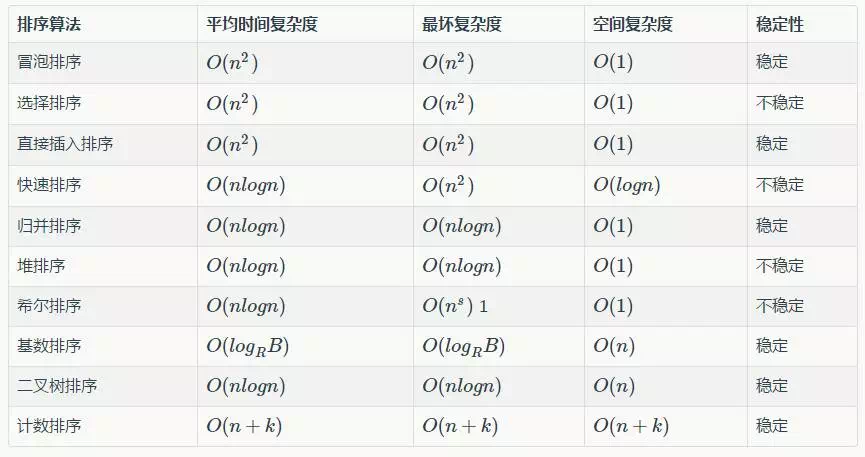
3 匿名管道

管道(Pipe)是一种具有两个端点的通信通道：有一端句柄的进程可以和有另一端句柄的进程通信。管道可以是单向－一端是只读的，另一端点是只写的；也可以是双向的一管道的两端点既可读也可写。匿名管道(Anonymous Pipe)是 在父进程和子进程之间，或同一父进程的两个子进程之间传输数据的无名字的单向管道。通常由父进程创建管道

4 剪切板 5 动态链接库 DLL 6 远程过程调用

# 12 数据结构

## 1 各种排序（一张表）



## 2 冒泡排序

1）定义：以从小到大排序为例，每一轮排序就找出未排序序列中最大值放在最后。

1. 实现1：以最常规的思路，两层遍历。第一层表示需要n轮排序过程，第二层表示第i轮时候将最大值排序在a[n-i]位置。但是
2. 优化1：如果序列已经有序了，那么就可以避免不必要的再比较。所以利用一个标志位flag表示某一轮循环是否发生了交换，如果没有发生交换就直接结束循环，表示已经排序完成。

4）优化2：记录排序过程中发生交换的尾边界，每次循环只需要遍历到尾边界就OK。

上面的优化过程的核心都是：减少比较的次数。

具体的代码可以参考：<http://blog.csdn.net/u010853261/article/details/54891710>

## 3 直接插入排序

1. 定义：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中的适当位置，直到全部记录插入完成为止。
2. 实现1：
3. 初始时，a[0]自成1个有序区，无序区为a[1..n-1]。令i=1
4. 将a[i]并入当前的有序区a[0…i-1]中形成a[0…i]的有序区间。

C）i++并重复第二步直到i==n-1，排序完成。

1. 优化1：从后往前比较，边比较边移动。即每次a[i]先和前面一个数据a[i-1]比较，如果a[i] > a[i-1]说明a[0…i]也是有序的，无须调整。否则就令j=i-1,temp=a[i]。然后一边将数据a[j]向后移动一边向前搜索，当有数据a[j]小于temp时就找到了插入位置。

4）优化2：将前面的数据移动变为数据交换。如果a[j]前一个数据a[j-1] > a[j]，就交换a[j]和a[j-1]，直到a[j-1] <= a[j]。这样也可以实现将一个新数据新并入到有序区间。

上面的时间复杂度均为O(n^2)，不过后面的优化能够改善一点点性能。

实现代码：<http://blog.csdn.net/u010853261/article/details/54891737>

## 4 快速排序

快排就是先一个定位函数，定位函数是找到一个数（可以是随机，一般用第一个数），然后把比这个数小的都放在左边，把比这个数大的都放在右边，然后快排递归调用这个定位函数即可。

快速排序是非常重要的，这在笔试经常会用到：

1）定义：基于分治的思想。首先在数组中选择一个基准点并把基准点放于序列的开头（该基准点的选取是优化快排的关键点），然后分别从数组的两端扫描数组，设两个指示标志（lo指向起始位置，hi指向末尾)，首先从后半部分开始，如果发现有元素比该基准点的值小，就交换lo和hi位置的值，然后从前半部分开始扫秒，发现有元素大于基准点的值，就交换lo和hi位置的值，如此往复循环，直到lo>=hi,然后把基准点的值放到hi这个位置，一次排序就完成了。之后再采用递归的方式分别对前半部分和后半部分排序，当前半部分和后半部分均有序时该数组自然也就有序了。

1. 一轮快速排序的过程：



下面值得一说的是一轮快速排序我们能得到什么？

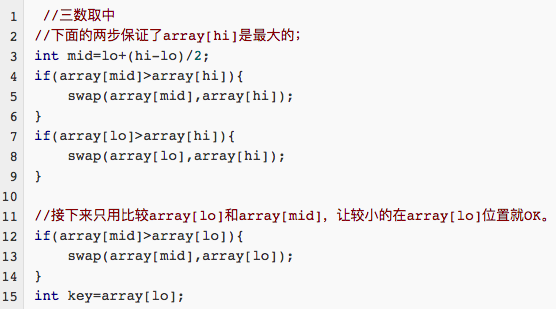
1. 举个例子，对于4、3、7、6、1这个序列，以4为基准，一轮快速排序之后的序列是1、3、4、6、7。 对于一轮快速排序之后的序列有这样的规律，对于基准4前面的序列b数据都比4小，对于4后面的序列c都比4大。
2. 上面的一轮快排的规律我们就能有所应用，比如对于一些求序列中最小的k个数。这时候序列b中数据都比基准小，所以当序列b数据量小于k-1时，说明序列b数据都是满足条件的，我们继续在序列c中再次执行快排，找结果。
3. 实现快排排序：



1. 优化：对于快排的优化，一般都是基准点选取的优化。

基准点一般有三种选择：固定、随机、三数取样。下面给出三数取中的实现：

三数取中的思想也就是：比如对于a、b、c，先比较a和c 以及 b和c 保证c是最大的，然后再比较a和b，选出中间值。



5）时间复杂度和空间复杂度

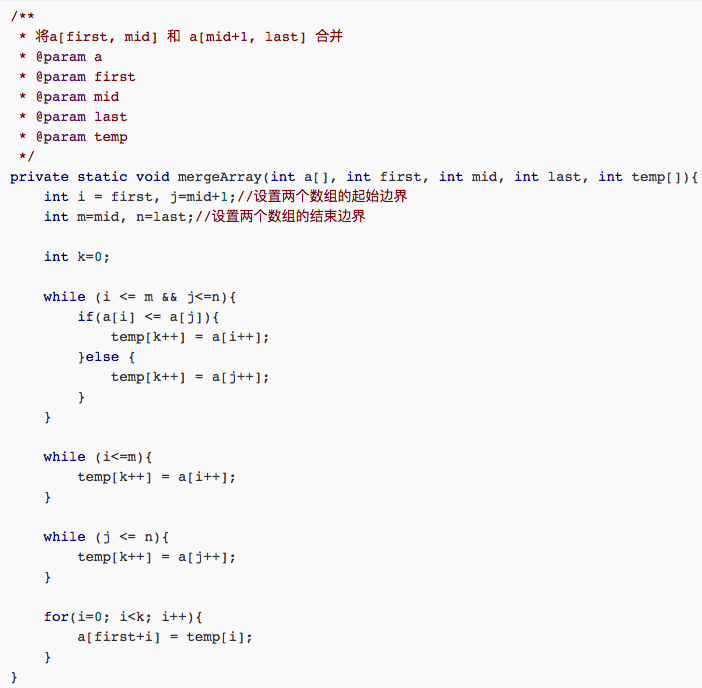
对于快排来说平均时间复杂度是O(nlogn)，但是当序列本身已经有序的情况下时间复杂度最糟糕是O(n^2)。 并且快速排序是不稳定的。

## 5 归并排序

1. 定义：归并算法是采用分治法（Divide and Conquer）的一个非常典型的应用，归并排序将两个已排序的表合并成一个表。

在这里实现归并排序其实就是两个过程：将一个序列先递归拆分序列，再归并。

1. 首先思考归并两个有序序列的实现，其实这个实现比较简单，时间复杂度也是O(n):这里归并数组a[first，mid] 和 a[mid+1，last]这两个数组。这里需要借助一个辅助数组。



1. 再看递归的拆分：



1. 前面的二路归并是基于数组实现的，但是如果我们要基于单链表实现呢？我们还是从两个角度考虑：拆分和合并。
2. 合并：对于两个有序单链表的合并并返回合并之后的单链表头结点，这个的是比较简单的，和数组的实现类似。
3. 拆分：对于拆分，可能就没有数组那么方便了，因为我们不能直接通过首尾索引值取平均数，但是链表也是可以实现的，我们可以使用双指针法，对于两个指针p1和p2，分别往后移动，p1移动一次，p2移动两次，当p2遍历到尾结点时，p1就指向中间结点实现了拆分。

具体的实现就不给出代码了，参考博客：<http://blog.csdn.net/u010853261/article/details/54884650>

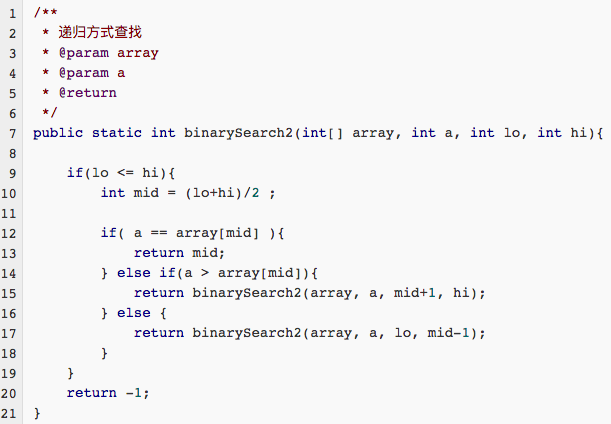
## 6 二分查找

1）定义：二分查找的前提是待查找的序列有序。每次取中间位置的值与待查关键字比较，如果中间位置的值比待查关键字大，则在前半部分循环这个查找的过程，如果中间位置的值比待查关键字小，则在后半部分循环这个查找的过程。直到查找到了为止，否则序列中没有待查的关键字。

1. 非递归实现：



3）递归实现：



## 7 什么是二叉平衡树，如何插入节点，删除节点，说出关键步骤

1）二叉平衡树：或则是一颗空树 或则它的任意一个节点的左右两个子树的高度差的绝对值不超过1，且左右子树都是二叉平衡树。

2）插入：和二叉查找树类似，只是可能会破坏树的平衡性，然后进行相应的旋转操作。

3）删除：（1）叶节点直接删除。 （2）非叶节点，要处理失衡情况。

## 8 桶排序

桶排序的基本思想是将一个数据表分割成许多buckets，然后每个bucket各自排序，或用不同的排序算法，或者递归的使用bucket sort算法。也是典型的divide-and-conquer分而治之的策略。它是一个分布式的排序，介于MSD基数排序和LSD基数排序之间。

# 13 简历

## 1 项目

### 1 紫领网

<http://www.open-open.com/jsoup/dom-navigation.htm>

Jsoup.parse检查 <script><form> <a href><img src>

git stash: 备份当前的工作区的内容，从最近的一次提交中读取相关内容，让工作区保证和上次提交的内容一致。同时，将当前的工作区内容保存到Git栈中。

git stash pop: 从Git栈中读取最近一次保存的内容，恢复工作区的相关内容。由于可能存在多个Stash的内容，所以用栈来管理，pop会从最近的一个stash中读取内容并恢复。

git stash list: 显示Git栈内的所有备份，可以利用这个列表来决定从那个地方恢复。

git stash clear: 清空Git栈。此时使用gitg等图形化工具会发现，原来stash的哪些节点都消失了。

## 1 Java编程思想

## 2 JavaCore

## 3 EffectJava

## 4 JVM

## 5 爬虫

Java是基于HttpClient库，但是如果运维分析日志发现都是 IP(1.1.1.1)这个用户，并且 useragent 还是 JavaClient1.6，基于这两点判断非人类后直接在Nginx 服务器上封杀。这个时候就得变换了下策略：

1. useragent模仿百度("Baiduspider...")

$ip = '220.181.108.91'; // 百度蜘蛛

//伪造百度蜘蛛IP

//伪造百度蜘蛛头部

1. IP每爬半个小时就换一个IP代理。

ADSL拨号上网使用动态IP地址，每一次拨号得到的IP都不一样，所以我们可以通过程序来自动进行重新拨号以获得新的IP地址，以达到突破反爬虫封锁的目的。假设有10个线程在跑，大家都正常的跑，跑着跑着达到限制了，WEB服务器提示你“非常抱歉，来自您ip的请求异常频繁”，于是大家争先恐后（几乎是同时）请求拨号，这个时候同步的作用就显示出来了，只会有一个线程能拨号，在他结束之前其他线程都在等，等他拨号成功之后，其他线程会被唤醒并返回。

算法描述：

1. 假设我现在有N个线程在抓取网页，发现被封锁之后就依次排队请求锁，假设线程1抢先得到锁，并设置flag=true开始拨号，注意，注意：线程1设置isDialing = true后其他线程才可能获得锁。  
   3、其他线程（2-N）依次获得锁，发现isDialing = true，于是wait。注意：获得锁并判断一个布尔值，跟后面的拨号操作比起来，时间可以忽略。  
   4、线程1拨号完毕isDialing = false。注意：这个时候可以断定，其他所有线程必定是处于wait状态等待唤醒。  
   5、线程1唤醒其他线程，其他线程和线程1返回开始抓取网页。  
   6、抓了一会儿之后，又会被封锁，于是回到步骤1。

运维也发现了对应的变化，于是在 Nginx 上设置了一个频率限制，每分钟超过120次请求的再屏蔽IP。 同时考虑到百度家的爬虫有可能会被误伤，想想市场部门每月几十万的投放，于是写了个脚本，通过 hostname 检查下这个 ip 是不是真的百度家的，对这些 ip 设置一个白名单。

写爬虫的发现了新的限制后，想着我也不急着要这些数据，留给服务器慢慢爬吧，于是修改了代码，随机1-3秒爬一次，爬10次休息10秒，每天只在8-12，18-20点爬，隔几天还休息一下。  
运维看着新的日志头都大了，再设定规则不小心会误伤真实用户，于是准备换了一个思路，当3个小时的总请求超过50次的时候弹出一个验证码弹框，没有准确正确输入的话就把 IP 记录进黑名单。

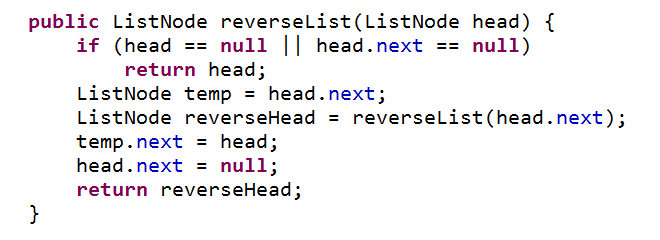
# 14 算法题

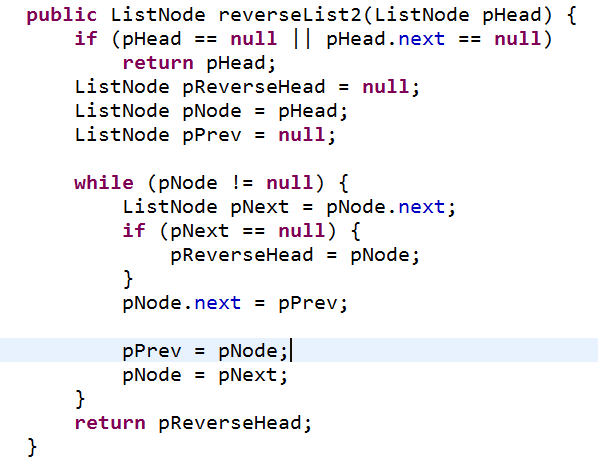
## 剑指offer

### 8 旋转数组的最小数字

用二分查找，最开始比较第一个数字和最后一个数字，如果最后一个数字比第一个数字大，那就是有序的，没有旋转过。如果是旋转过，如果中间这个数比开始那个数大，那么最小的肯定在他的右边，如果小，肯定在左边，但是有特殊情况，如果是特殊情况，我们必须按照遍历那种去算。

### 16 反转链表





### 36 数组中的逆序对

先把数组分割成子数组，先统计出子数组内部的逆序对的数目，然后统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。其实这个过程就是归并排序。

## 1 两个无序数组，求出共同的

构建哈希表插入操作的过程中，如果元素已经插入过，即其哈希地址有值，则该元素必为两数组的交集，打印输出即可。（前提数组中的元素不重复）

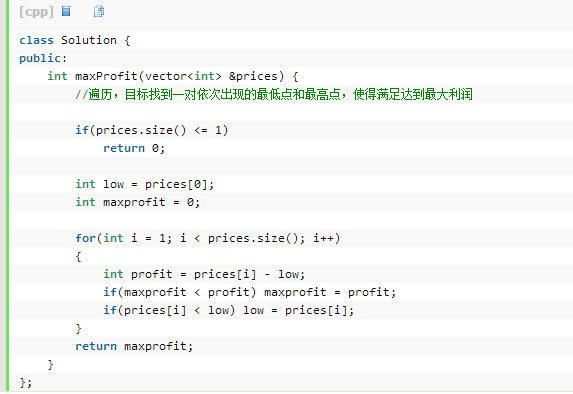
## 2 判断平衡二叉树

求深度递归，但是时间复杂度高了，要求一次遍历（后续遍历）

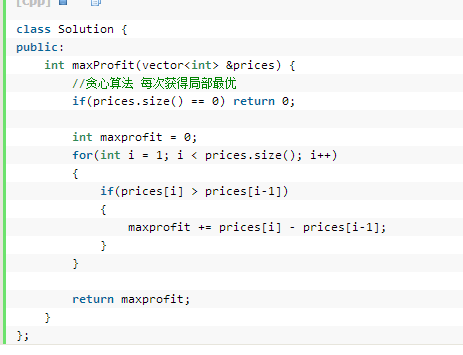
## 3 股价 贪心算法

### 1 买卖一次

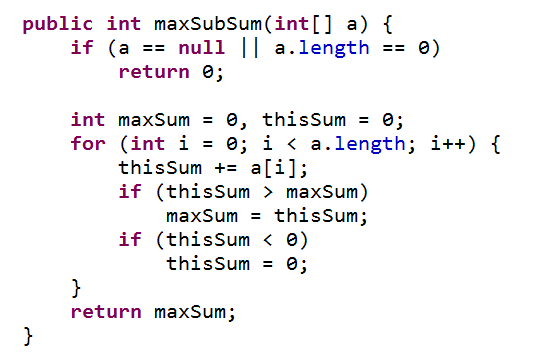
从前往后，用当前价格减去此前的最低价格，就是在当前点卖出股票能获得的最高利润。



### 2 买卖多次



## 4 最大子序列



## 5 一个排好序的数组，找出两数之和为m的所有组合

前后同时往中间加就行

## 6 自然数序列，找出任意连续之和等于n的所有子序列

连续的数与n比较，如果比n小就把后面的数往上加，如果比n大就把前面的数减下去。

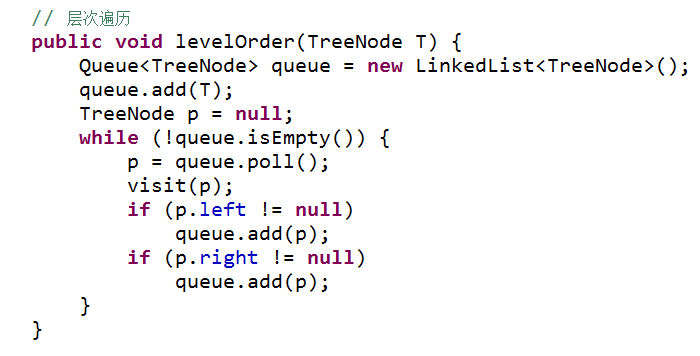
## 7 给定2,3,5面值的硬币若干，给出一个数字，计算拼凑这个数字最少用到硬币的个数

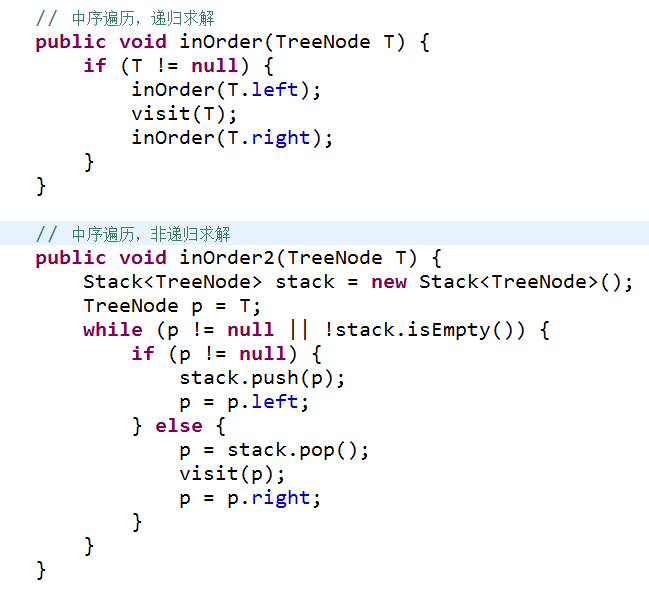
动态规划

先与最大的求余嘛，得到的商记录下来，然后把这些记录加起来就是硬币的个数

## 8 二叉树BFS（广度优先） DFS（深度）

BFS相当于层次遍历，用栈 DFS相当于中序遍历，用栈





## 9 旋转数组找到指定数

二分查找，根据中间个数比目标的数大小，如果比目标数大，那么目标数在左边，如果比目标数小，此时目标数又比最右边的数小，就在中间数的右边，如果比最右边的数大，那肯定在左边。

## 10 对单链表排序（快排、归并）

根据普通快排的思路,选择1个点为中心点,保证中心点左边比中心点小,中心点右边比中心点大即可.

单链表的实现为：

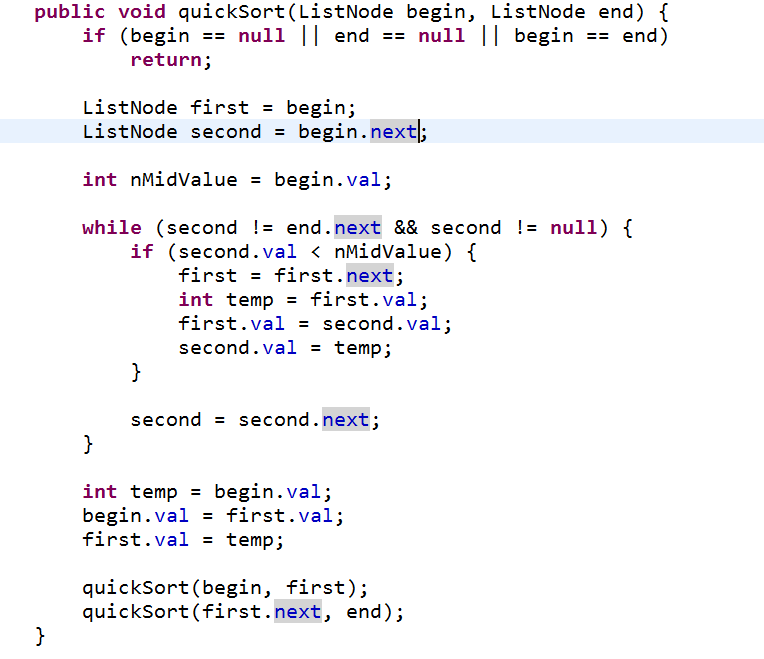
1.使第一个节点为中心点.

2.创建2个指针(p,q),p指向头结点,q指向p的下一个节点.

3.q开始遍历,如果发现q的值比中心点的值小,则此时p=p->next,并且执行当前p的值和q的值交换,q遍历到链表尾即可.

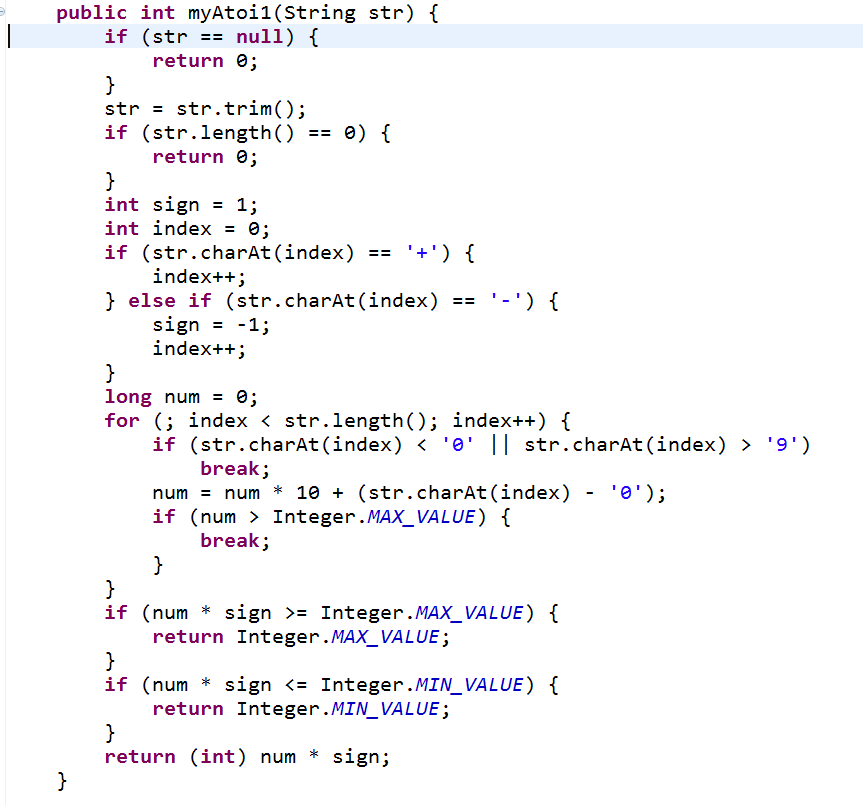
4.把头结点的值和p的值执行交换.此时p节点为中心点,并且完成1轮快排

5.使用递归的方法即可完成排序



## 11 实现内存复制

## 12 字符串转换整数



## 13 排序电话号码（桶排序，bit位）

1 基数排序 2 用桶排序，分到不同的桶可以用treeset维护

## 14 实现MIN栈加辅助栈，但是要优化辅助栈

# 15 开放性问题 海量数据

<http://www.educity.cn/wenda/389464.html>

## 1 给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，找出a，b共同的url

方案1：可以估计每个文件安的大小为50G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

s 遍历文件a，对每个url求取点击查看大图，然后根据所取得的值将url分别存储到1000个小文件（记为点击查看大图）中。这样每个小文件的大约为300M。

s 遍历文件b，采取和a相同的方式将url分别存储到1000各小文件（记为点击查看大图）。这样处理后，所有可能相同的url都在对应的小文件（点击查看大图）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。

s 求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash\_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。

方案2：如果允许有一定的错误率，可以使用Bloom filter，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是共同的url（注意会有一定的错误率）

## 有10个文件，每个文件1G，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。要求你按照query的频度排序。

方案1：

s 顺序读取10个文件，按照hash(query)%10的结果将query写入到另外10个文件（记为点击查看大图）中。这样新生成的文件每个的大小大约也1G（假设hash函数是随机的）。

s 找一台内存在2G左右的机器，依次对点击查看大图用hash\_map(query, query\_count)来统计每个query出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的query和对应的query\_cout输出到文件中。这样得到了10个排好序的文件（记为点击查看大图）。

s 对点击查看大图这10个文件进行归并排序（内排序与外排序相结合）。

方案2：

一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash\_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案3：

与方案1类似，但在做完hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如MapReduce），最后再进行合并。

## 3. 有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。

方案1：顺序读文件中，对于每个词x，取点击查看大图，然后按照该值存到5000个小文件（记为点击查看大图）中。这样每个文件大概是200k左右。如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，知道分解得到的小文件的大小都不超过1M。对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用trie树/hash\_map等），并取出出现频率最大的100个词（可以用含100个结点的最小堆），并把100词及相应的频率存入文件，这样又得到了5000个文件。下一步就是把这5000个文件进行归并（类似与归并排序）的过程了。

## 4. 海量日志数据，提取出某日访问百度次数最多的那个IP。

方案1：首先是这一天，并且是访问百度的日志中的IP取出来，逐个写入到一个大文件中。注意到IP是32位的，最多有点击查看大图个IP。同样可以采用映射的方法，比如模1000，把整个大文件映射为1000个小文件，再找出每个小文中出现频率最大的IP（可以采用hash\_map进行频率统计，然后再找出频率最大的几个）及相应的频率。然后再在这1000个最大的IP中，找出那个频率最大的IP，即为所求。

## 5. 在2.5亿个整数中找出不重复的整数，内存不足以容纳这2.5亿个整数。

方案1：采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存点击查看大图内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看bitmap，把对应位是01的整数输出即可。

方案2：也可采用上题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

## 6. 海量数据分布在100台电脑中，想个办法高校统计出这批数据的TOP10。

方案1：

s 在每台电脑上求出TOP10，可以采用包含10个元素的堆完成（TOP10小，用最大堆，TOP10大，用最小堆）。比如求TOP10大，我们首先取前10个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是TOP10大。

s 求出每台电脑上的TOP10后，然后把这100台电脑上的TOP10组合起来，共1000个数据，再利用上面类似的方法求出TOP10就可以了。

## 7. 怎么在海量数据中找出重复次数最多的一个？

方案1：先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

## 8. 上千万或上亿数据（有重复），统计其中出现次数最多的钱N个数据。

方案1：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用hash\_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前N个出现次数最多的数据了，可以用第6题提到的堆机制完成。

## 9.1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

方案1：这题用trie树（字典树）比较合适，hash\_map也应该能行。

## 10. 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。

方案1：这题是考虑时间效率。用trie树（字典树）统计每个词出现的次数，时间复杂度是O(n\*le)（le表示单词的平准长度）。然后是找出出现最频繁的前10个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是O(n\*lg10)。所以总的时间复杂度，是O(n\*le)与O(n\*lg10)中较大的哪一个。

## 11. 一个文本文件，找出前10个经常出现的词，但这次文件比较长，说是上亿行或十亿行，总之无法一次读入内存，问最优解。

方案1：首先根据用hash并求模，将文件分解为多个小文件，对于单个文件利用上题的方法求出每个文件件中10个最常出现的词。然后再进行归并处理，找出最终的10个最常出现的词。