**一、Java线程热点面试题**

<http://www.importnew.com/12773.html>

**1、什么是线程？**

线程是操作系统能够进行**运算调度的最小单位**，它被包含在进程之中，是*进程中的实际运作单位*。程序员可以通过它进行多处理器编程，你可以使用多线程对运算密集型任务提速。比如，如果一个线程完成一个任务要100毫秒，那么用十个线程完成改任务只需10毫秒。

**2、线程和进程有什么区别？**

线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

**3、Java中CyclicBarrier 和 CountDownLatch有什么不同？**

CyclicBarrier 和 CountDownLatch 都可以用来让一组线程等待其它线程。与 CyclicBarrier 不同的是，CountdownLatch 不能重新使用。

**4、一个线程运行时发生异常会怎样？**

如果异常没有被捕获该线程将会停止执行。Thread.**UncaughtExceptionHandler**是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候JVM会使用Thread.getUncaughtExceptionHandler()来查询线程的UncaughtExceptionHandler并将线程和异常作为参数传递给handler的uncaughtException()方法进行处理。

**5、Java中如何停止一个线程？（安全结束、**

<http://www.cnblogs.com/luckygxf/p/4737655.html>

Java提供了很丰富的API，但没有为停止线程提供API。JDK 1.0本来有一些像stop()， suspend() 和 resume()的控制方法，但是由于**潜在的死锁威胁**因此在后续的JDK版本中他们被弃用了，之后Java API的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。

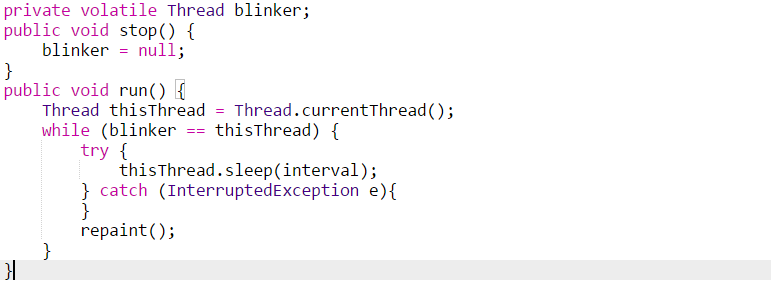
以下方法可安全停止一个线程：

* 当run() 或者 call() 方法执行完的时候线程会**自动结束**；
* **手动结束（volatile标志）：**使用共享变量**volatile 布尔变量**来退出run()方法的循环或者是取消任务来中断线程。因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号，通知中断线程的执行。

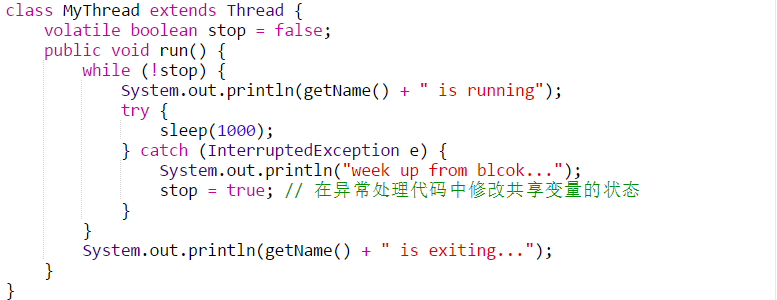


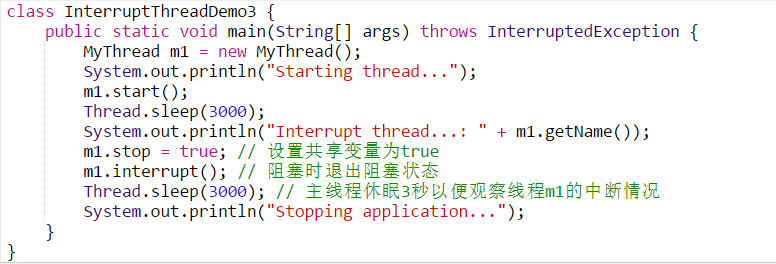
在上面代码中定义了一个退出标志exit，当exit为true时，while循环退出，exit的默认值为false。在定义exit时，使用了一个Java关键字volatile，这个关键字的目的是*使exit同步，也就是说在同一时刻只能由一个线程来修改exit的值*。

在《[Why Are Thread.stop， Thread.suspend，Thread.resume and Runtime.runFinalizersOnExit Deprecated?](http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html)》中，建议使用如下的方法来停止线程：



* **使用interrupt方法终止线程：**如果一个线程由于等待某些事件的发生而被**阻塞**，又该怎样停止该线程呢？这种情况经常会发生，比如当一个线程由于需要**等候键盘输入**而被阻塞，或者调用Thread.join()方法，或者Thread.sleep()方法，在网络中调用ServerSocket.accept()方法，或者调用了DatagramSocket.receive()方法时，都有可能导致线程阻塞，使线程处于处于不可运行状态时，即使主程序中将该线程的共享变量设置为true，但该线程此时根本无法检查循环标志，当然也就无法立即中断（阻塞状态无法中断、。这里我们给出的建议是，不要使用stop()方法，而是使用**Thread提供的interrupt()方法**，因为该方法虽然**不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常，从而使线程提前结束阻塞状态，退出堵塞代码**。





**注意：**在Thread类中有两个方法可以判断线程是否通过interrupt方法被终止。一个是静态的方法interrupted（、，一个是非静态的方法isInterrupted（、，这两个方法的区别是interrupted用来**判断当前线是否被中断**，而isInterrupted可以**用来判断其他线程是否被中断**。

**6、Java中notify 和 notifyAll有什么区别？**

多线程可以等待单监控锁，Java API 的设计人员提供了一些方法，当**等待条件改变的时候通知它们**，但是这些方法没有完全实现。notify()方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而notifyAll()唤醒所有线程并允许他们**争夺锁**确保了至少有一个线程能继续运行。

**7、为什么wait， notify 和 notifyAll这些方法不在thread类里面？**

这是个**设计相关的问题**，它考察的是面试者对现有系统和一些普遍存在但看起来不合理的事物的看法。回答这些问题的时候，你要说明为什么把这些方法放在Object类里是有意义的，还有不把它放在Thread类里的原因。一个很明显的原因是**JAVA提供的锁是对象级的而不是线程级的**，每个对象都有锁，通过线程获得。如果线程需要等待某些锁，那么调用对象中的**wait()方法**就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是**锁级别的操作**，所以把他们定义在Object类中因为**锁属于对象**。

**8、什么是ThreadLocal变量？**

ThreadLocal是Java里一种**特殊的变量**。每个线程都有一个ThreadLocal就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用ThreadLocal让SimpleDateFormat变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。线程局部变量的另一个不错的例子是ThreadLocalRandom类，它在多线程环境中减少了创建代价高昂的Random对象的个数。

ThreadLocal一般称为线程本地变量，它是一种**特殊的线程绑定机制**，将变量与线程绑定在一起，为每一个线程维护一个独立的变量副本。通过ThreadLocal可以将对象的可见范围限制在同一个线程内。

需要重点强调的的是，不要拿ThreadLocal和synchronized做类比，因为这种比较压根就是无意义的！**sysnchronized是一种互斥同步机制**，是为了保证在多线程环境下对于共享资源的正确访问。而ThreadLocal从本质上讲，无非是提供了一个**“线程级”的变量作用域**，它是一种**线程封闭（每个线程独享变量、技术**，更直白点讲，ThreadLocal可以理解为将对象的作用范围限制在一个线程上下文中，使得变量的作用域为“线程级”。

**9、Java中interrupted 和 isInterruptedd方法的区别？**

interrupted() 和 isInterrupted()的主要区别：前者会将中断状态清除而后者不会。Java多线程的中断机制是用**内部标识**来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有可能被其它线程调用中断来改变。

**10、Java中什么是竞态条件？ 举个例子说明**

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。导致竞态条件发生的代码区称作临界区。在临界区中使用适当的同步就可以避免竞态条件。临界区实现方法有两种，一种是用**synchronized**，一种是用**Lock显式锁**实现。

**竞态条件会导致程序在并发情况下出现一些bugs**。多线程对一些**资源的竞争**的时候就会产生竞态条件，如果首先要执行的程序竞争失败排到后面执行了，那么整个程序就会出现一些不确定的bugs。这种bugs很难发现而且会重复出现，因为线程间的随机竞争。

**11、为什么你应该在循环中检查等待条件?**

如果不在循环中检查等待条件，处于等待状态的线程可能会收到**错误警报和伪唤醒**，程序就会在没有满足结束条件的情况下退出。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在**notify()**方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用wait()方法效果更好的原因。（在使用**等待或通知机制**时，要配合boolean值或能够判断真假的条件，在notify之前改变boolean量的值，让wait返回后能退出while循环，一般在wait()方法的外围加一层while循环，以防止早期通知或在通知遗漏后阻塞在wait方法内、

**12、为什么wait、notify和notifyAll要在同步块中执行？**

* 如果调用某个对象的wait()、notify和notifyAll方法，当前线程必须拥有这个对象的monitor（即**锁**、，因此调用wait()、notify和notifyAll方法必须在**同步块或者同步方法**中进行（synchronized块或者synchronized方法、。
* Java API规范规定，如果**不在同步块**中执行会抛出**IllegalMonitorStateException**异常
* 为了避免wait和notify之间产生**竞态条件**。（竞态条件会导致程序在并发情况下出现一些bugs。多线程对一些资源的竞争的时候就会产生竞态条件，如果首先要执行的程序竞争失败排到后面执行了，那么整个程序就会出现一些不确定的bugs。这种bugs很难发现而且会重复出现，因为线程间的随机竞争。、

**13、Java中的同步集合与并发集合有什么区别？**

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在Java1.5之前程序员们只有同步集合来用，且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5介绍了并发集合像**ConcurrentHashMap**，不仅提供**线程安全**还用**锁分离和内部分区**等现代技术提高了可扩展性。

**14、Java中堆和栈有什么不同？**

为什么把这个问题归类在多线程和并发面试题里？因为*栈是一块和线程紧密相关的内存区域。每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的*。而堆是所有线程共享的一片**公用内存区域**。对象都在堆里创建，为了提升效率线程会从堆中弄一个**缓存**到自己的栈，如果多个线程使用该变量就可能引发问题，这时**volatile** 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

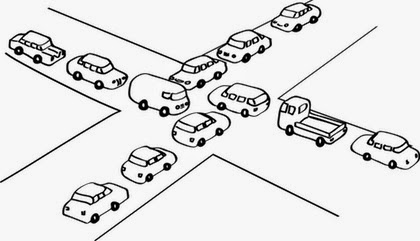
**15、什么是线程池？ 为什么要使用它？**

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且**一个进程能创建的线程数有限**。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为**线程池**，里面的线程叫**工作线程**。从JDK1.5开始，Java API提供了**Executor框架**让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池、。

**16、如何写代码来解决生产者消费者问题？**

在现实中你解决的许多线程问题都属于生产者消费者模型，就是一个线程生产任务供其它线程进行消费，你必须知道**怎么进行线程间通信**来解决这个问题。比较低级的办法是用wait和notify来解决这个问题，比较赞的办法是用**Semaphore 或者 BlockingQueue**来实现生产者消费者模型。

**17、如何避免死锁？**



Java多线程中的死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务。

例如：线程A当前持有**互斥所锁lock1**，线程B当前持有互斥锁lock2。接下来，当线程A仍然持有lock1时，它试图获取lock2，因为线程B正持有lock2，因此线程A会阻塞等待线程B对lock2的释放。如果此时线程B在持有lock2的时候，也在试图获取lock1，因为线程A正持有lock1，因此线程B会阻塞等待A对lock1的释放。二者都在等待对方所持有锁的释放，而二者却又都没释放自己所持有的锁，这时二者便会一直阻塞下去。这种情形称为死锁。

死锁的发生必须满足以下四个条件：

* **互斥条件：**一个资源每次只能被一个进程使用。
* **请求与保持条件：**一个进程因请求资源而阻塞时，对已获得的资源保持不放。
* **不剥夺条件**：进程已获得的资源，在末使用完之前，不能强行剥夺。
* **循环等待条件：**若干进程之间形成一种头尾相接的循环等待资源关系。

或者

* **互斥条件**：进程要求对所分配的资源（如打印机、进行排他性控制，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。
* **不剥夺条件**：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。
* **请求和保持条件**：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。
* **循环等待条件**：存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被 链中下一个进程所请求。即存在一个处于**等待状态的进程集合**{Pl， P2， ...， pn}，其中Pi等待的资源被P(i+1)占有（i=0， 1， ...， n-1)，Pn等待的资源被P0占有，如图2-15所示。

避免死锁的办法：

* **阻止循环等待条件**，将系统中所有的资源设置**标志位、排序**，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。
* 尽可能**减小锁的作用范围**，只在**必要的最短时间内持有锁**，考虑使用同步语句块代替整个同步方法；
* *尽量编写不在同一时刻需要持有多个锁的代码*，如果不可避免，则确保线程持有第二个锁的时间尽量短
* 创建和使用一个大锁来代替若干小锁，并把这个锁用于互斥，而不是用作单个对象的对象级别锁。

**18、Java中活锁和死锁有什么区别？**

这是上题的扩展，**活锁和死锁**类似，不同之处在于处于活锁的线程或进程的状态是不断改变的，活锁可以认为是一种特殊的饥饿。一个现实的活锁例子，是两个人在狭小的走廊碰到，两个人都试着避让对方好让彼此通过，但是因为避让的方向都一样导致最后谁都不能通过走廊。简单的说就是，活锁和死锁的主要区别是**前者进程的状态可以改变但是却不能继续执行**。

**19、死锁、饥饿以及活锁**

* **饥饿**指的线程无法访问到它需要的资源而不能继续执行时，引发饥饿最常见资源就是CPU时钟周期。虽然在Thread API中由指定线程优先级的机制，但是只能作为操作系统进行线程调度的一个参考，换句话说就是操作系统在进行**线程调度**是平台无关的，会尽可能提供公平的、活跃性良好的调度，那么即使在程序中指定了线程的优先级，也有可能在操作系统进行调度的时候映射到了同一个优先级。通常情况下，不要区修改线程的优先级，一旦修改程序的行为就会与平台相关，并且会导致饥饿问题的产生。在程序中使用的Thread.yield或者Thread.sleep表明该程序试图客服优先级调整问题，让优先级更低的线程拥有被CPU调度的机会。
* **活锁**指的是线程不断重复执行相同的操作，但每次操作的结果都是失败的。尽管这个问题**不会阻塞线程**，但是程序也无法继续执行。活锁通常发生在处理事务消息的应用程序中，如果不能成功处理这个事务那么事务将回滚整个操作。**解决活锁的办法**是在*每次重复执行的时候引入随机机制，这样由于出现的可能性不同使得程序可以继续执行其他的任务*。

**活锁的例子：**

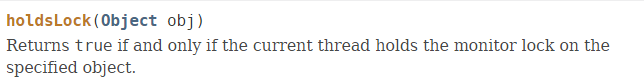
* **单一实体的活锁：**例如线程从队列中拿出一个任务来执行，如果任务执行失败，那么将任务重新加入队列，继续执行。假设任务总是执行失败，或者某种依赖的条件总是不满足，那么线程一直在繁忙却没有任何结果。
* **协同导致的活锁：**生活中的典型例子： 两个人在窄路相遇，同时向一个方向避让，然后又向另一个方向避让，如此反复。
* 通信中也有类似的例子，多个用户**共享信道**（最简单的例子是大家都用对讲机、，同一时刻只能有一方发送信息。发送信号的用户会进行**冲突检测**， 如果发生冲突，就选择避让，然后再发送。假设避让算法不合理，就导致每次发送，都冲突，避让后再发送，还是冲突。
* **计算机中的例子：**两个线程发生了某些条件的碰撞后重新执行，那么如果再次尝试后依然发生了碰撞，长此下去就有可能发生活锁。

**活锁的解决方法**：解决协同活锁的一种方案是调整重试机制。

* 比如引入一些**随机性。**例如如果检测到冲突，那么就暂停随机的一定时间进行重试。这回大大减少碰撞的可能性。典型的例子是*以太网的CSMA/CD检测机制*。
* 另外为了避免可能的死锁，适当加入一定的**重试次数**也是有效的解决办法。尽管这在业务上会引起一些复杂的逻辑处理。
* 比如**约定重试机制避免再次冲突**。例如自动驾驶的防碰撞系统（假想的例子、，可以根据序列号约定检测到相撞风险时，序列号小的飞机朝上飞， 序列号大的飞机朝下飞。

**20、怎么检测一个线程是否拥有锁？**

在java.lang.Thread中有一个方法叫**holdsLock()**，它**返回true如果当且仅当当前线程拥有某个具体对象的锁**。

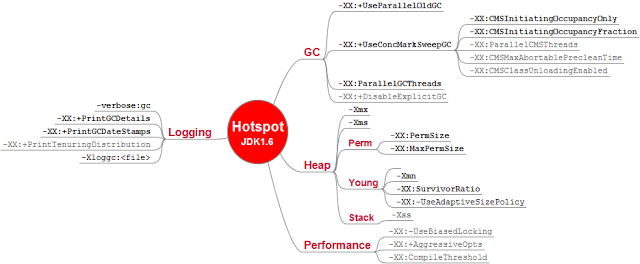


**21、JVM中哪个参数是用来控制线程的堆栈大小的**

-Xms：设置 Java 堆的初始化大小

-Xmx：设置最大的 Java 堆大小

-Xss：设置Java线程堆栈大小

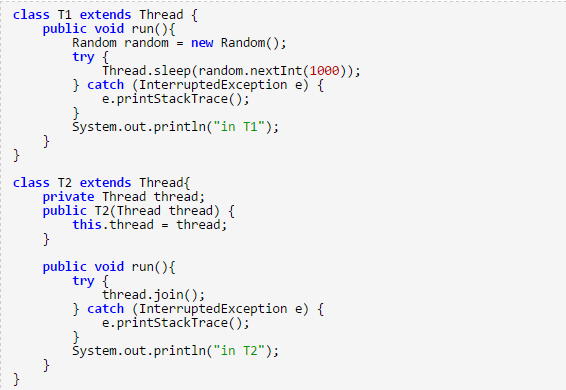


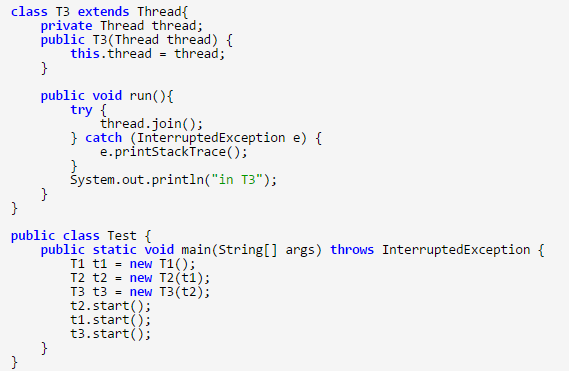
**22、Java中synchronized和 ReentrantLock有什么不同？**

Java在过去很长一段时间只能通过synchronized关键字来实现**互斥**，它有一些缺点。比如你不能扩展锁之外的方法或者块边界，**尝试获取锁时不能中途取消**等。Java 5 通过Lock接口提供了*更复杂的控制来解决这些问题*。 ReentrantLock 类实现了Lock，它拥有与 synchronized 相同的并发性和内存语义且它还具有可扩展性。

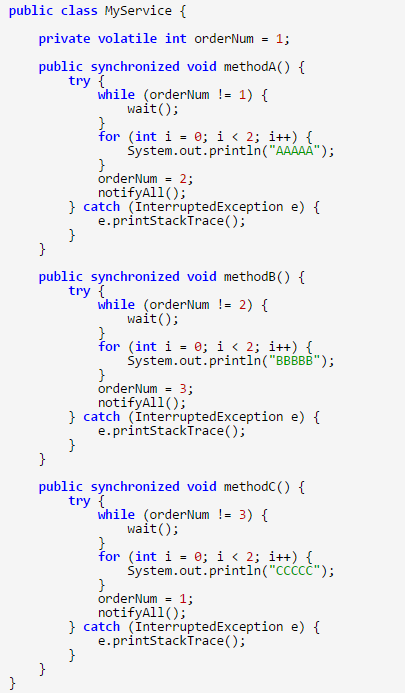
**23、有三个线程T1，T2，T3，怎么确保它们按顺序执行？**

* 在多线程中有多种方法让线程按特定顺序执行，你可以用**线程类的join()方法**在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先**启动最后一个**(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。





* 通过**共享对象锁加上可见volatile变量**来实现。**共享对象锁**，可以保证每个方法只能同时有**一个线程进入**，配合wait和notifyall方法，可以启动或者唤醒线程。



**24、Thread类中的yield方法有什么作用？**

Yield方法可以暂停当前正在执行的线程对象，让其它**有相同优先级的线程执行**。它是一个静态方法，而且*只保证当前线程****放弃****CPU占用，而不能保证使其它线程一定能占用CPU*，执行yield()的线程有可能在进入到暂停状态后**马上**又被执行。

**25、Java中ConcurrentHashMap的并发度是什么？**

ConcurrentHashMap*把实际map划分成若干部分来实现它的可扩展性和线程安全*。这种划分是使用**并发度**获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16，这样在多线程情况下就能避免争用。

**26、Java中Semaphore是什么？**

Java中的Semaphore是一种**新的同步类**，它是一个**计数信号**。从概念上讲，*信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可*。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，**Semaphore只对可用许可的号码进行计数，并采取相应的行动**。信号量常常用于多线程的代码中，比如**数据库连接池**。

**27、如果你提交任务时，线程池队列已满。会时发会生什么？**

这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果*一个任务不能被调度执行*，那么ThreadPoolExecutor’s submit()方法将会抛出一个**RejectedExecutionException**异常。如果没有任务可调度，则会抛出NullPointerException异常。

**28、Java线程池中submit() 和 execute()方法有什么区别？**

两个方法都可以向线程池提交任务，**execute()方法的返回类型是void**，它定义在Executor接口中， 而**submit()方法可以返回持有计算结果的Future对象**，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

**29、什么是阻塞式方法？**

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，**ServerSocket的accept()**方法就是一直等待客户端连接。这里的阻塞是指**调用结果返回之前，当前线程会被挂起**，直到得到结果之后才会返回。此外，还有**异步和非阻塞式方法在任务完成前就返回**。

**30、Swing是线程安全的吗？为什么？**

Swing不是线程安全的，但是你应该解释这么回答的原因即便面试官没有问你为什么。当我们说*swing不是线程安全的常常提到它的组件*，这些组件不能在多线程中进行修改，所有**对GUI组件的更新都要在AWT线程中完成**，而Swing提供了**同步和异步两种回调**方法来进行更新。

**31、Swing API中那些方法是线程安全的？**

这个问题又提到了swing和线程安全，虽然*组件不是线程安全的，但是有一些方法是可以被多线程安全调用的*，比如repaint()， revalidate()。 JTextComponent的setText()方法和JTextArea的insert() 和 append() 方法也是线程安全的

**32、Java中的ReadWriteLock是什么？**

读写锁是用来提升并发程序性能的**锁分离技术**的成果。Java中的ReadWriteLock是Java 5 中新增的一个接口，一个ReadWriteLock维护**一对关联的锁，一个用于只读操作一个用于写**。在没有写线程的情况下，一个读锁可能会同时被多个读线程持有。**写锁是独占的**，你可以使用JDK中的**ReentrantReadWriteLock**来实现这个规则，它最多支持65535个写锁和65535个读锁。

**33、多线程中的忙循环是什么?**

**忙循环就是程序员用循环让一个线程等待**，不像传统方法wait()， sleep() 或 yield() 它们都放弃了CPU控制，而**忙循环不会放弃CPU，它就是在运行一个空循环**。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

**34、volatile 变量和 atomic 变量有什么不同？**

这是个有趣的问题。首先，volatile 变量和 atomic 变量看起来很像，但功能却不一样。Volatile变量可以**确保先行关系**，即写操作会发生在后续的读操作之前， 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。而**AtomicInteger**类提供的atomic方法可以让这种操作具有**原子性**如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

**35、如果同步块内的线程抛出异常会发生什么？**

若你能想到锁是否释放这条线索来回答还有点希望答对。无论你的*同步块是正常还是异常退出的*，里面的**线程都会释放锁**，所以对比锁接口我更喜欢同步块，因为它不用我花费精力去**释放锁**，该功能可以在**finally block**里释放锁实现。

**36、单例模式的双检锁是什么？（DCL、**

这个问题在Java面试中经常被问到，但是面试官对回答此问题的满意度仅为50%。一半的人写不出**双检锁**，还有一半的人说不出它的**隐患和Java1.5**是如何对它修正的。它其实*是一个用来创建线程安全的单例的老方法*，当单例实例第一次被创建时它试图用单个锁进行性能优化，但是由于太过于复杂在JDK1.4中它是失败的，我个人也不喜欢它。无论如何，即便你也不喜欢它但是还是要了解一下，因为它经常被问到。

<http://blog.csdn.net/ns_code/article/details/17348313>



DCL对instance进行了两次null判断，第一层判断主要是**为了避免不必要的同步**，第二层的判断则是**为了在null的情况下创建实例**。DCL的分析也告诉我们一条经验原则：对引用（包括对象引用和数组引用、的**非同步访问**，即使得到该引用的最新值，却并不能保证也能得到其成员变量（对数组而言就是每个数组元素、的最新值。

**37、如何在Java中创建线程安全的Singleton？**

如果你不喜欢**双检锁**，而面试官问了创建Singleton类的替代方法，你可以*利用JVM的类加载和静态变量初始化特征来创建Singleton实例*，或者是利用**枚举类型**来创建Singleton，我很喜欢用这种方法。

**38、写出3条你遵循的多线程最佳实践**

以下三条最佳实践我觉得大多数Java程序员都应该遵循：

* **给你的线程起个有意义的名字：**这样可以方便找bug或追踪。OrderProcessor， QuoteProcessor or TradeProcessor 这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。
* **避免锁定和缩小同步的范围：**锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。
* **多用同步类少用wait 和 notify：**首先，CountDownLatch， Semaphore， CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。
* **多用并发集合少用同步集合：**这是另外一个容易遵循且受益巨大的最佳实践，*并发集合比同步集合的可扩展性更好*，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用**ConcurrentHashMap**。

**39、如何强制启动一个线程？**

这个问题就像是**如何强制进行Java垃圾回收**，目前还没有方法，虽然你可以使用System.gc()来进行垃圾回收，但是不保证能成功。在*Java里面没有办法强制启动一个线程，它是****被线程调度器****控制着，且Java没有公布相关的API*。

**40、Java多线程中调用wait() 和 sleep()方法有什么不同？**

Java程序中wait 和 sleep都会造成某种形式的**暂停**，它们可以**满足不同的需要**。wait()方法用于线程间通信，如果等待条件为真，且其它线程被唤醒时它会释放锁，而sleep()方法*仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁*。

**41、Java中的fork join框架是什么？**

fork join框架是JDK7中出现的一款高效的工具，Java开发人员可以通过它**充分利用现代服务器上的多处理器**。它是专门为了那些可以递归划分成许多子模块设计的，目的是将**所有可用的处理能力用来提升程序的性能**。fork join框架一个巨大的优势是它*使用了工作窃取算法，可以完成更多任务的工作线程可以从其它线程中窃取任务来执行*。

**42、线程池的实现原理（参数控制、**

**1）重要参数**

* **corePoolSize**：线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到**当前线程**数等于corePoolSize；如果当前线程数为corePoolSize，继续提交的任务被保存到**阻塞队列**中，等待被执行；如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。
* **maximumPoolSize**：线程池中允许的最大线程数。如果当前**阻塞队列满**了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize；
* **keepAliveTime**：线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间；默认情况下，该参数只在**线程数大于corePoolSize时才有用**；
* **unit：**keepAliveTime的单位；
* **workQueue**：用来保存等待被执行的任务的阻塞队列，且任务必须实现Runable接口，在JDK中提供了如下阻塞队列：
* **ArrayBlockingQueue**：基于数组结构的有界阻塞队列，按FIFO排序任务；
* **LinkedBlockingQuene**：基于链表结构的阻塞队列，按FIFO排序任务，**吞吐量**通常要高于ArrayBlockingQuene；
* **SynchronousQuene**：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene；
* **priorityBlockingQuene**：具有优先级的无界阻塞队列；
* **handler**：线程池的饱和策略（任务拒绝策略、，当阻塞队列满了，且**没有空闲**的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：
* **AbortPolicy**：直接抛出异常，默认策略；
* **CallerRunsPolicy**：用调用者所在的线程来执行任务；
* **DiscardOldestPolicy**：丢弃阻塞队列中**靠最前的任务**，并执行当前任务；
* **DiscardPolicy**：直接丢弃任务；

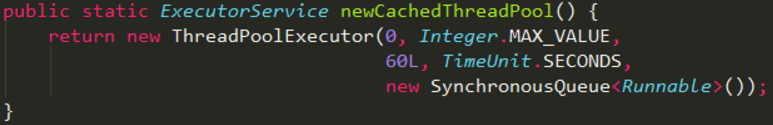
当然也可以根据应用场景实现**RejectedExecutionHandler**接口，自定义饱和策略，如**记录日志或持久化存储不能处理的任务**。

**2）任务提交给线程池之后的处理策略：**

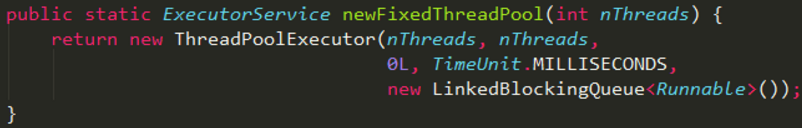
* 如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；
* 如果当前线程池中的**线程数目>=corePoolSize**，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会**等待空闲线程**将其取出去执行；若添加失败（一般来说是**任务缓存队列已满**、，则会尝试**创建新的线程去执行这个任务**；
* 如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；
* 如果线程池中的线程数量大于 corePoolSize时，如果**某线程空闲时间超过**keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么**核心池中**的线程空闲时间超过keepAliveTime，线程也会被终止。

**3）Java通过Executors提供四种线程池**

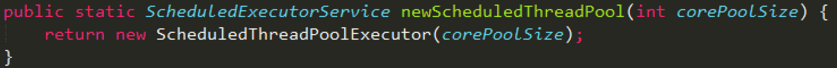
* **newCachedThreadPool**：创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程；
* 初始化一个**可以缓存线程的线程池**，默认缓存60s，线程池的线程数可达到Integer.MAX\_VALUE，即2147483647，内部使用**SynchronousQueue**作为**阻塞队列**；
* 和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的**空闲时间**超过**keepAliveTime**，会**自动释放线程资源**，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；
* 所以，使用该线程池时，一定要**注意控制并发的任务数**，否则创建大量的线程可能导致严重的性能问题。



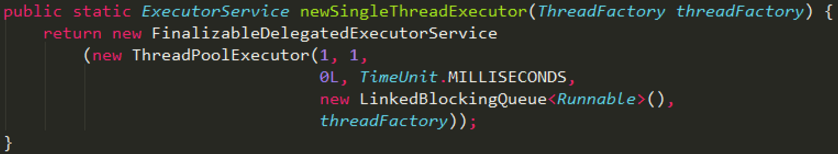
* **newFixedThreadPool**：创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待；**初始化一个指定线程数的线程池**，其中corePoolSize == maximumPoolSize，使用**LinkedBlockingQuene**作为阻塞队列，不过当线程池没有可执行任务时，也不会释放线程。



* **newScheduledThreadPool**：创建一个定长线程池，支持定时及周期性任务执行。初始化的线程池可以**在指定的时间内周期性的执行所提交的任务**，在实际的业务场景中可以使用该线程池定期的同步数据。



* **newSingleThreadExecutor**：创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO， LIFO， 优先级)执行。**初始化的线程池中只有一个线程**，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行，内部使用**LinkedBlockingQueue**作为阻塞队列。

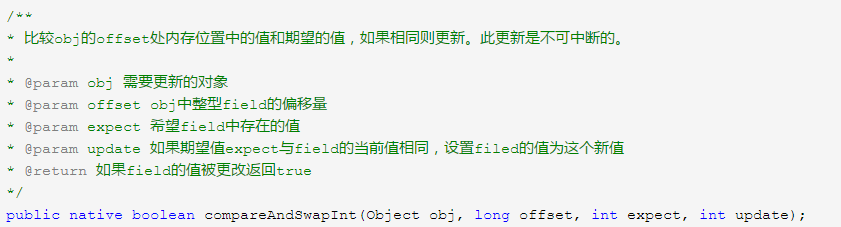


**43、CAS的实现原理**

CAS是通过unsafe类的compareAndSwap方法实现的

方法的参数的含义：

<http://www.cnblogs.com/mickole/articles/3757278.html>



一个CAS方法包含三个参数CAS(V，E，N)。V表示要更新的变量，E表示预期的值，N表示新值。只有当V的值等于E时，才会将V的值修改为N。如果V的值不等于E，说明已经被其他线程修改了，当前线程可以放弃此操作，也可以再次尝试次操作直至修改成功。基于这样的算法，CAS操作即使没有锁，也可以发现其他线程对当前线程的干扰（临界区值的修改、，并进行恰当的处理。

**44、CAS指令有什么缺点及解决方法**

**CAS存在一个很明显的问题，即ABA问题**。如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？**如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过**。针对这种情况，java并发包中提供了一个带有标记的原子引用类"**AtomicStampedReference**"，它可以**通过控制变量值的版本来保证CAS的正确性**。

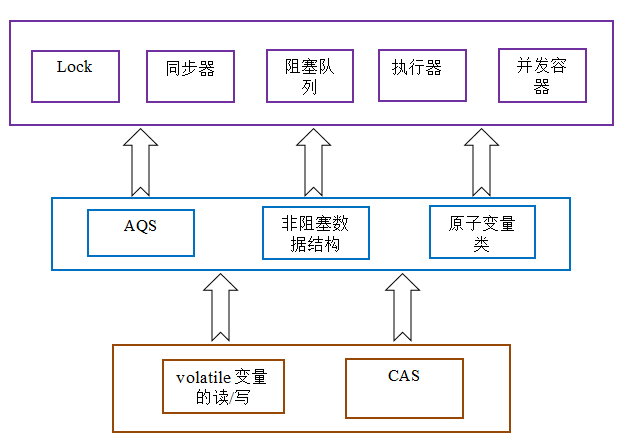
**问题参考**：<http://blog.csdn.net/ls5718/article/details/52563959>

* **ABA问题。**因为CAS需要**在操作值的时候检查下值有没有发生变化**，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是**使用版本号**。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

从Java1.5开始JDK的atomic包里提供了一个类**AtomicStampedReference**来解决ABA问题。这个类的**compareAndSet**方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。**ABA问题最容易发生在lock free 的算法**中的，CAS首当其冲，因为CAS判断的是指针的地址。如果这个地址被重用了呢，问题就很大了。（**地址被重用**是很经常发生的，一个内存分配后释放了，再分配，很有可能还是原来的地址、

关于ABA问题参考文档（实例分析、: <http://blog.hesey.NET/2011/09/resolve-aba-by-atomicstampedreference.html>

* **循环时间长开销大。**自旋**CAS如果长时间不成功，会给CPU带来非常大的执行开销**。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，**pause指令有两个作用**，第一它可以延迟流水线执行指令（de-pipeline、，使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation、而引起CPU流水线被清空（CPU pipeline flush、，从而提高CPU的执行效率。
* **只能保证一个共享变量的原子操作**。当对**一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作**，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用**锁**，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2，j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了**AtomicReference类来保证引用对象之间的原子性（引用对象操作原子性、**，你可以把多个变量放在一个**对象**里来进行**CAS操作**。



**45、什么是AQS？**

AbstractQueuedSynchronizer：抽象队列同步器

AQS（AbstractQueuedSynchronizer、是Java实现同步组建的基础框架，一般以静态内部类的形式实现在某个同步组件类中，通过代理的方式向外提供同步服务，ReentrantLock和Semaphore都是基于AQS实现的同步组件，前者是独占式同步组建，即一个线程获得后，其他线程无法获得。后者是共享式同步组件，一个线程获得后，在满足的条件下，其他线程也可以获得。

AQS的基础是CAS。AQS是JDK1.5提供的一个基于FIFO等待队列实现的一个用于实现同步器的基础框架，这个基础框架的重要性可以这么说，JCU包里面几乎所有的有关锁、多线程并发以及线程同步器等重要组件的实现都是基于AQS这个框架。**AQS的核心思想**是基于**volatile int state这样的一个属性同时配合Unsafe工具**对其原子性的操作来实现对当前锁的状态进行修改。当state的值为0的时候，标识改Lock不被任何线程所占有。

**1）工作原理**

AQS是Java实现同步组建的基础框架，其**基本思想**是用一个volatile int state变量来表示当前同步组件的状态，用getState()获取同步组件的状态，用compareAndSet(int expect， int update)来对state状态进行操作，compareAndSet可以保证对state变量更新值的原子性。AQS中很多方法是final的，即不允许用户覆盖，用户自定义的方法一般有：

* **tryAcquire:** 独占式获取同步状态，该函数一般首先查询state的值，如果state不允许继续被获取，直接返回false。如果state允许继续被获取，CAS尝试更新state的值，成功返回true，失败返回false
* **tryAcquireShared：**共享式的获取同步状态，该一般是在**CAS死循环**获取state的值，计算state被获取后的值，如果该值为负数，直接返回负数表示失败，如果该值为正值，则用CAS更新该值，当CAS更新失败时，重复上述步骤，直至返回负数或CAS更新成功返回正值。
* **tryRelease：**独占式的释放同步状态
* **tryReleaseShared：**共享式的释放同步状态，一般在**CAS死循环中反复尝试**，直至释放成功
* **isHeldExclusively：**判断当前同步器是否被当前线程占有

AQS提供的模板方法有：

* acquire:独占式的获取同步状态，获取成功则返回，获取失败则会进入**等待队列**，该方法会调用用户自定义的tryAcquire函数
* acquireInterruptibly：与acquire类似，不同在于当进入等待队列时，遇到**中断**会抛出InterruptedException异常，用户可以处理该中断异常
* tryAcquireNanos：在acquireInterruptibly的基础上增加了时间限制，一定时间内没有成功获取则返回false
* acquireShared：共享式的获取同步状态，成功则返回，失败则进入等待队列，该方法会调用用户自定义的tryAcquired函数
* acquireSharedInterruptibly：在等待队列可以相应中断，与上类似
* tryAcquireShared：在acquireSharedInterruptibly增加了超时限制
* release：独占式的释放同步状态，会调用用户自定义tryRelease函数
* releaseShared：共享式的释放同步状态，会调用用户自定义tryReleaseShared函数
* getQueuedThreads：获取等待队列线程集合

**羊群效应**

这里说一下羊群效应，当有多个线程去竞争同一个锁的时候，假设锁被某个线程占用，那么如果有成千上万个线程在等待锁，有一种做法是同时唤醒这成千上万个线程去去竞争锁，这个时候就发生了羊群效应，海量的竞争必然造成资源的剧增和浪费，因此终究只能有一个线程竞争成功，其他线程还是要老老实实的回去等待。**AQS的FIFO的等待队列给解决在锁竞争方面的羊群效应问题提供了一个思路：保持一个FIFO队列，队列每个节点只关心其前一个节点的状态，线程唤醒也只唤醒队头等待线程**。其实这个思路已经被应用到了分布式锁的实践中，见：Zookeeper分布式锁的改进实现方案。

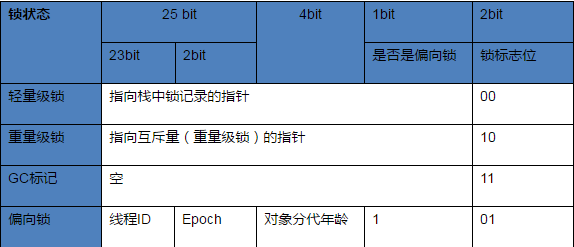
**46、Synchronized实现原理**

<http://blog.csdn.net/u012465296/article/details/53022317>

**Java对象头和moniter**是synchronized的基础

任何一个对象都一个Monitor与之关联，当且一个**Monitor**被持有后，它将处于锁定状态。**Synchronized在JVM里的实现都是基于进入和退出Monitor对象来实现方法同步和代码块同步**，虽然具体实现细节不一样，但是都可以通过成对的MonitorEnter和MonitorExit指令来实现。MonitorEnter指令插入在同步代码块的开始位置，当代码执行到该指令时，将会尝试获取该对象Monitor的所有权，即尝试获得该对象的锁，而monitorExit指令则插入在方法结束处和异常处，**JVM保证每个MonitorEnter必须有对应的MonitorExit**。

synchronized用的**锁是存在Java对象头**里的，那么什么是Java对象头呢？Hotspot虚拟机的对象头主要包括两部分数据：Mark Word（标记字段、、Klass Pointer（类型指针、。其中Klass Point是是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键。



**什么是Monitor？**

我们可以把它理解为一个**同步工具**，也可以描述为一种同步机制，它通常被描述为一个对象。与一切皆对象一样，所有的Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中 ，每一个Java对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者Monitor锁。

**Monitor 是线程私有的数据结构，每一个线程都有一个可用monitor record列表**，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个monitor关联（对象头的MarkWord中的LockWord指向monitor的起始地址、，同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。

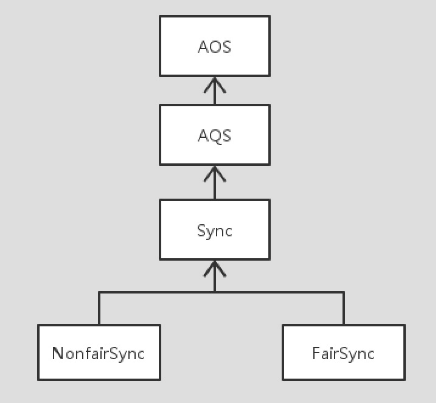
**47、ReentranLock实现原理（参数控制、**

ReentrantLock是基于AQS实现的

**什么是可重入锁？**

ReentrantLock是可重入锁，什么是可重入锁呢？可重入锁就是当前持有该锁的线程能够多次获取该锁，无需等待。可重入锁是如何实现的呢？这要从ReentrantLock的一个内部类Sync的父类说起，Sync的父类是AbstractQueuedSynchronizer（后面简称AQS、。可重入锁的好处可以在递归算法中使用锁，不可重入锁则导致无法在递归算法中使用锁。因为第二次递归时由于第一次递归已经占有锁，而导致死锁。

**ReentrantLock锁的架构：**ReentrantLoc的架构相对简单，主要包括一个Sync的内部抽象类以及Sync抽象类的两个实现类。上面已经说过了Sync继承自AQS，他们的结构示意图如下：



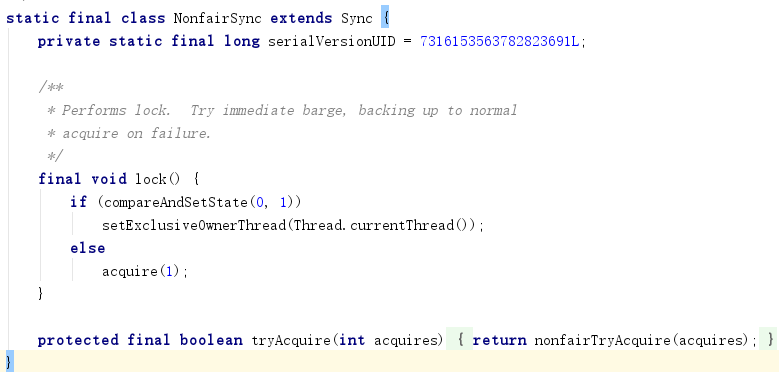
另外、Sync的两个实现类分别是**NonfairSync和FairSync**，由名字大概可以猜到，一个是用于实现公平锁、一个是用于实现非公平锁。那么Sync为什么要被设计成内部类呢？我们可以看看**AQS主要提供了哪些protect的方法用于修改state的状态**，我们发现Sync被设计成为安全的外部不可访问的内部类。ReentrantLock中所有涉及对AQS的访问都要经过Sync，其实，Sync被设计成为内部类主要是为了安全性考虑，这也是作者在AQS的comments上强调的一点。

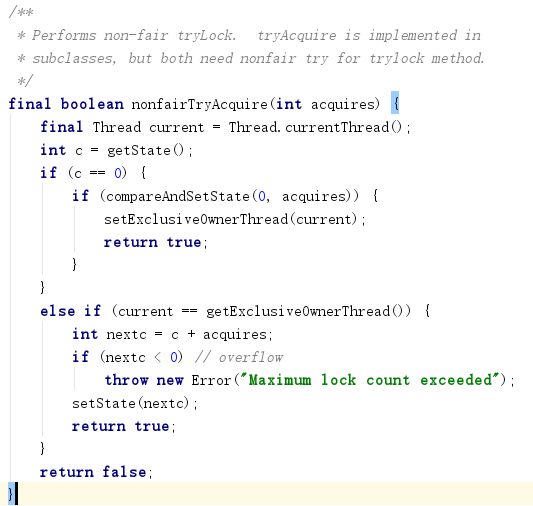
ReentrantLock的内部类Sync继承了AQS，分为公平锁FairSync和非公平锁NonfairSync。

* **公平锁：**线程获取锁的顺序和调用lock的顺序一样，FIFO；
* **非公平锁：**线程获取锁的顺序和调用lock的顺序无关，全凭运气。

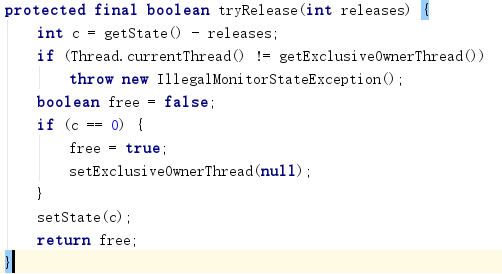
**1）源码分析**

NonfairSync继承了Sync，Sync是一个抽象类，并继承了抽象类AbstractQueuedSynchronizer。ReentrantLock是一个**独占式的锁**，所以它需要实现tryAcquire函数和tryRelease函数。





* 先得到当前线程
* 查询当前state值，如果为0则说明当前锁还未被其他线程获取，则尝试**CAS获得锁**，成功则把占有锁的线程设置为当前线程，返回true。失败返回false。
* 如果state不为0则说明该锁已经被其他线程获取，则检查获得锁的线程是否是当前线程以实现**可重入特性**，如果是，则更新state的值，并返回true。此处更新不需要CAS，因为只有当前线程可以操作state。
* 其他情况返回false



* 首先得到**释放之后的状态值c**
* 检查当前释放锁的线程，如果不是已占有锁的线程则抛出异常，因为ReentrantLock是独占式锁，释放锁的线程一定是占有锁的线程
* 如果c是等于0的，说明*获取锁的所有函数都已经返回，则锁释放成功*
* 如果c不等于0，说明只是部分递归的函数返回，部分递归函数还未返回，则释放失败，锁依然被占有

**48、同步类**

**1）CountDownLatch（闭锁、倒计数门闸锁、**

<http://www.cnblogs.com/nullzx/p/5272807.html>

CountDownLatch是一个同步工具，它主要用**线程执行之间的协作**。CountDownLatch 的作用和 Thread.join() 方法类似，让一些线程阻塞直到另一些线程完成一系列操作后才被唤醒。在直接创建线程的年代（Java 5.0 之前、，我们可以使用 Thread.join()。在**线程池出现后，因为线程池中的线程不能直接被引用**，所以就必须使用 CountDownLatch 了。

CountDownLatch主要有**两个方法**，当一个或多个线程调用await方法时，这些线程会阻塞。其它线程调用countDown方法会将计数器减1(调用countDown方法的线程不会阻塞)，当计数器的值变为0时，因await方法阻塞的线程会被唤醒，继续执行。

**实现原理：**计数器的值由构造函数传入，并用它初始化AQS的state值。当线程调用await方法时会检查state的值是否为0，如果是就直接返回（即不会阻塞、；如果不是，将表示该节点的线程入列，然后将自身阻塞。当其它线程调用countDown方法会将计数器减1，然后判断计数器的值是否为0，当它为0时，会**唤醒队列**中的第一个节点，由于CountDownLatch使用了**AQS的共享模式**，所以第一个节点被唤醒后又会唤醒第二个节点，以此类推，使得所有因await方法阻塞的线程都能被唤醒而继续执行。

从源代码和实现原理中可以看出一个CountDownLatch对象，只能使用一次，不能重复使用。

**2）CyclicBarrier（同步屏障、可重用栅栏、**

<http://www.cnblogs.com/nullzx/p/5271964.html>

CyclicBarrier 的字面意思是可循环（Cyclic、使用的屏障（Barrier、。它要做的事情是，***让一组线程到达一个屏障（也可以叫同步点、时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活***。线程进入屏障通过CyclicBarrier的await()方法。

CyclicBarrier默认的构造方法是CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。CyclicBarrier还提供一个更高级的构造函数CyclicBarrier(int parties， Runnable barrierAction)，用于在线程到达屏障时，优先执行barrierAction这个Runnable对象，方便处理更复杂的业务场景

**实现原理：**在CyclicBarrier的**内部定义了一个Lock对象**，每当一个线程调用CyclicBarrier的await方法时，将剩余拦截的线程数减1，然后判断剩余拦截数是否为0，如果不是，进入Lock对象的条件队列等待。如果是，执行barrierAction对象的Runnable方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从await方法返回，再从CyclicBarrier的await方法中返回。

**3）Semaphore（信号量、**

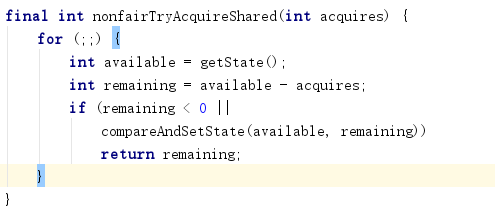
<http://www.cnblogs.com/nullzx/p/5270233.html>

在Java的并发包中，Semaphore类表示信号量。Semaphore内部主要通过AQS（AbstractQueuedSynchronizer、实现线程的管理。Semaphore有两个构造函数，参数permits表示许可数，它最后传递给了AQS的state值。线程在运行时首先获取许可，如果成功，许可数就减1，线程运行，当线程运行结束就释放许可，许可数就加1。如果许可数为0，则获取失败，**线程位于AQS的等待队列中，它会被其它释放许可的线程唤醒**。在创建Semaphore对象的时候还可以指定它的公平性。一般常用非公平的信号量，**非公平信号量**是指在获取许可时先尝试获取许可，而不必关心是否已有需要获取许可的线程位于等待队列中，如果获取失败，才会入列。而公平的信号量在获取许可时首先要查看等待队列中是否已有线程，如果有则入列。

信号量主要用于两个目的，一个是用于**多个共享资源的互斥使用**，另一个用于**并发线程数的控制**。Semaphore是JDK提供的一个**可共享的同步组建**，有n个许可，多个线程可以共同去获得许可，当线程申请的许可小于n时即可成功申请，否则申请失败。

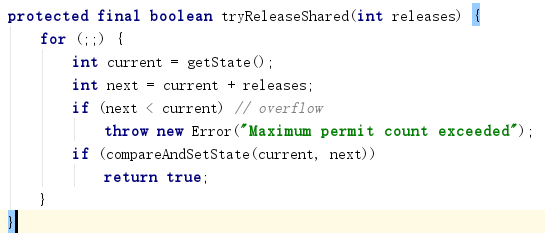
**A. 源码分析**

与ReentrantLock代码结构非常相似。Semaphore是一个**共享式的同步组建**，它应该实现tryAcquireShared和tryReleaseShared。



* 函数首先取得**当前的可用许可数**，并计算被获取acquires个许可后**剩余的许可数**。
* 如果剩余的许可数小于0直接返回剩余的许可数，即负值
* 如果大于0则尝试使用CAS循环更新state的值，更新失败则重试上述步骤，直至返回负值更新失败，或者返回非负值更新成功。

**tips：**与独占式的tryAcquire逻辑不太一样，独占式的tryAcquire在CAS操作失败后，直接返回失败。本人觉得共享式的tryAcquiredShared在CAS操作失败后，因为组件是共享的，所以再次尝试获取同步组件成功的可能性较大，所以在CAS失败后，尝试再次更新。而独占式的CAS更新失败后，组件已经被其他线程获取，再次尝试成功的可能性较小，所以没有重新尝试。纯属个人观点。



* 函数计算释放后的state值并验证是否溢出
* CAS更新state的值直至成功

通过ReentrantLock和Semaphore一个独占式的同步组件和一个非独占式的同步组件来学习怎么使用**AQS实现通组件**，具体来说分为以下步骤：

* 待实现的同步组件是独占式的还是共享式的
* 独占式的同步组件实现tryAcquire和tryRelease，非独占式的实现tryAcquireShared和tryReleaseShared
* 将我们实现的同步组建相应的方法如Lock和unLock代理到AQS对应的函数包括用户自定义的函数和AQS提供的模板函数

AQS的方便之处在于我们只需要实现tryAcquire和tryRelease或tryAcquireShared和tryReleaseShared就可以使用AQS帮我们实现好的阻塞的acquire函数，可中断的acquire函数，带超时的acquire函数等模板函数，大大简化了用户的开发量和难度。

**49、线程启动的三种方式**

**1）继承Thread类**

* 定义Thread类的子类（**子类继承**），并**重写该类的run方法**，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。
* 创建Thread**子类的实例**，即创建了线程对象。
* 调用线程对象的start()方法来启动该线程。

构造方法：



**2）实现Runnable接口**

* 定义Runnable接口的实现类，重写该接口的run()方法，run()方法的方法体同样是该线程的线程执行体;
* 创建Runnable实现类的实例，并将其作为Thread的target来创建Tread对象，该Thread对象才是真正的线程对象；
* 调用线程对象的start()方法来启动该线程

**3）通过Callable和Future创建线程（有返回值）**

* 创建**Callable接口的实现类**，并实现call()方法，该call()方法将作为线程执行体，并且有返回值；
* 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值；
* 使用FutureTask对象**作为Thread对象的target**创建并启动新线程；
* 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

**二、Java内存热点面试题**

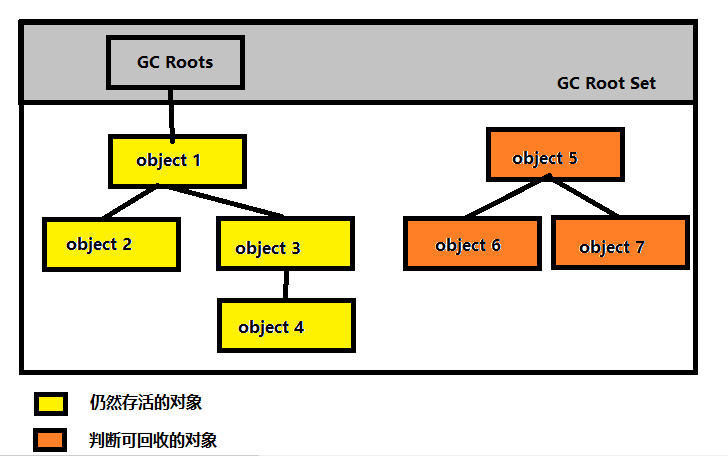
**1、Java对象存活判断方法**

判断对象是否存活一般有两种方式：

* **引用计数：**每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
* **可达性分析（Reachability Analysis、：**过一系列的称为**“GC Roots”的对象作为起始点**，从这些节点开始向下搜索，搜索所走过的路径称为**引用链（Reference Chain、**，当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达、时，则证明此对象是不可用的。如下图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。

在Java语言中，GC Roots包括：

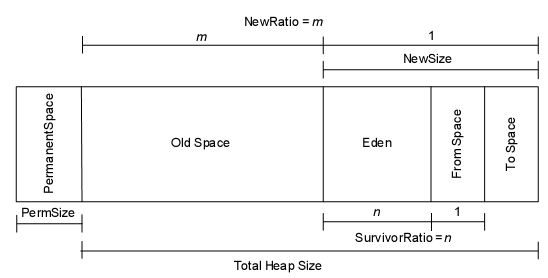
* 虚拟机栈中**引用的对象**。
* 方法区中类**静态属性实体引用的对象**。
* 方法区中**常量引用的对象**。
* 本地方法栈中**JNI引用的对象**。



**2、JVM内存调优**

<http://www.cnblogs.com/jackyrong/archive/2010/01/21/1653163.html>

<http://blog.csdn.net/cutesource/article/details/5907418>



JVM采用一种分代回收 (generational collection) 的策略，用较高的频率对年轻的对象(young generation)进行扫描和回收，这种叫做minor collection，而对老对象(old generation)的检查回收频率要低很多，称为major collection。这样就不需要每次GC都将内存中所有对象都检查一遍。

**当一个URL被访问时，内存申请过程如下：**

A. JVM会试图为相关Java对象在Eden中初始化一块内存区域

B. 当Eden空间足够时，内存申请结束。否则到下一步

C. JVM试图释放在Eden中所有不活跃的对象（这属于1或更高级的垃圾回收）, 释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区

D. Survivor区被用来作为Eden及OLD的中间交换区域，当OLD区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区

E. 当OLD区空间不够时，JVM会在OLD区进行完全的垃圾收集（0级）

F. 完全垃圾收集后，若Survivor及OLD区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现”out of memory错误”

**3、触发JVM进行Full GC的情况**

<http://blog.csdn.net/chenleixing/article/details/46706039/>

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC，对老年代GC称为Major GC,而Full GC是对整个堆来说的，在最近几个版本的JDK里默认包括了对永生带即方法区的回收（JDK8中无永生带了），出现Full GC的时候经常伴随至少一次的Minor GC,但非绝对的。Major GC的速度一般会比Minor GC慢10倍以上。下面简单介绍触发JVM进行Full GC的情况：

* **System.gc()方法的调用**：此方法的调用是建议JVM进行Full GC，虽然只是建议而非一定，但很多情况下它会触发 Full GC，从而增加Full GC的频率，也即增加了间歇性停顿的次数。强烈影响系建议能不使用此方法就别使用，让虚拟机自己去管理它的内存，可通过通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc
* **老年代内存空间不足**：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.lang.OutOfMemoryError: Java heap space，为避免以上两种状况引起的Full GC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。
* **永生区空间不足**：JVM规范中**运行时数据区域中的方法区**，在HotSpot虚拟机中又被习惯称为永生代或者永生区，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。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：java.lang.OutOfMemoryError: PermGen space，为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。
* **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。
* promotion failed是在进行Minor GC时，survivor space放不下、对象只能放入老年代，而此时老年代也放不下造成的；
* concurrent mode failure是在执行CMS GC的过程中同时有对象要放入老年代，而此时老年代空间不足造成的（有时候“空间不足”是CMS GC时当前的浮动垃圾过多导致暂时性的空间不足触发Full GC）。

**解决措施**：增大survivor space、老年代空间或调低触发并发GC的比率，但在JDK 5.0+、6.0+的版本中有可能会由于JDK的bug29导致CMS在remark完毕后很久才触发sweeping动作。对于这种状况，可通过设置-XX: CMSMaxAbortablePrecleanTime=5（单位为ms）来避免

* **统计得到的Minor GC晋升到老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。

例如程序第一次触发Minor GC后，有6MB的对象晋升到旧生代，那么当下一次Minor GC发生时，首先检查旧生代的剩余空间是否大于6MB，如果小于6MB，则执行Full GC。

当新生代采用PS GC时，方式稍有不同，PS GC是在Minor GC后也会检查，例如上面的例子中第一次Minor GC后，PS GC会检查此时旧生代的剩余空间是否大于6MB，如小于，则触发对旧生代的回收。

除了以上4种状况外，对于使用RMI来进行RPC或管理的Sun JDK应用而言，默认情况下会一小时执行一次Full GC。可通过在启动时通过- java -Dsun.rmi.dgc.client.gcInterval=3600000来设置Full GC执行的间隔时间或通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc。

* **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。

为了解决这个问题，CMS垃圾收集器提供了一个**可配置的参数**，即-XX:+UseCMSCompactAtFullCollection开关参数，用于在“享受”完Full GC服务之后额外免费赠送一个碎片整理的过程，内存整理的过程无法并发的，空间碎片问题没有了，但提顿时间不得不变长了，JVM设计者们还提供了另外一个参数 -XX:CMSFullGCsBeforeCompaction，这个参数用于设置在执行多少次不压缩的Full GC后，跟着来一次带压缩的。

**4、GC优化的目的**

<http://www.importnew.com/3146.html>

GC优化归纳了两个目的：

* **一个是将转移到老年代的对象数量降到最少**：按代的GC机制由Oracle JVM提供，不包括可以在JDK7以及更高版本中使用的G1 GC。换句话说，对象被创建在伊甸园空间，而后转化到幸存者空间，最终剩余的对象被送到老年代。某些比较大的对象会在被创建在伊甸园空间后，直接转移到老年代空间。老年代空间上的GC处理会新生代花费更多的时间。因此，减少被移到老年代对象的数据可以显著地减少Full GC的频率。减少被移到老年代空间的对象的数量，可能被误解为将对象留在新生代。但是，这是不可能的。取而代之，你可以调整新生代空间的大小；
* **另一个是减少Full GC的执行时间**：Full GC的执行时间比Minor GC要长很多。因此，如果Full GC花费了太多的时间（超过1秒），一些连接的部分可能会发生超时错误。
* 如果你试图通过消减老年代空间来减少Full GC的执行时间，可能会导致OutOfMemoryError 或者 Full GC执行的次数会增加。
* 与之相反，如果你试图通过增加老年代空间来减少Full GC执行次数，执行时间会增加。

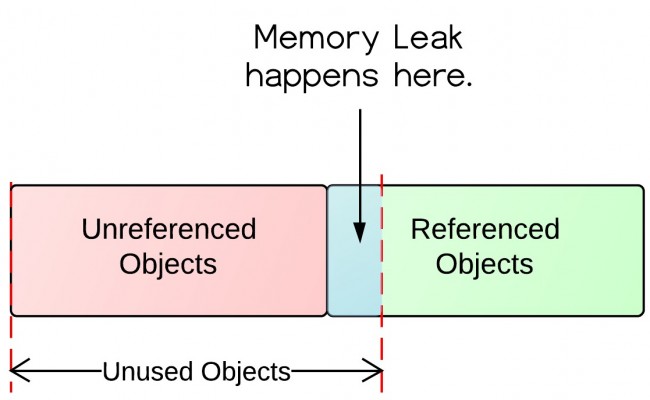
**GC优化的最基本原则**是将不同的GC参数用于2台或者多台服务器，并进行对比，并将那些被证明提高了性能或者减少了GC执行时间的参数应用于服务器。请谨记这一点。

**5、Java内存泄露原因**

<https://yq.aliyun.com/articles/61148>

<http://www.importnew.com/12961.html>

**内存泄漏的定义**：对象已经没有被应用程序使用，但是垃圾回收器没办法移除它们，因为还在被引用着。了解对象在内存中的状态，下面的这张图就解释了什么是**无用对象**以及什么是**未被引用对象**。



上面图中可以看出，里面有被引用对象和未被引用对象。未被引用对象会被垃圾回收器回收，而被引用的对象却不会。未被引用的对象当然是不再被使用的对象，因为没有对象再引用它。然而无用对象却不全是未被引用对象。其中还有被引用的。就是这种情况导致了内存泄漏。

**6、GC算法（垃圾回收算法）**

**1）标记 – 清除算法**

“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：**首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象**。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。

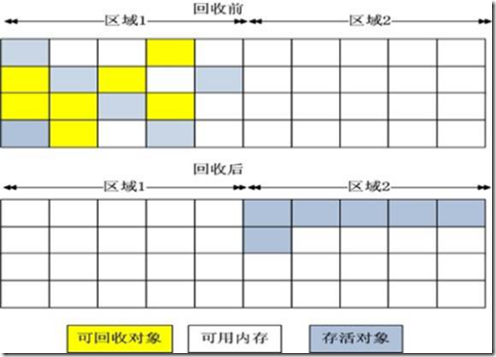
它的**主要缺点有两个**：一个是**效率问题**，标记和清除过程的效率都不高；另外一个是**空间问题**，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时**无法找到足够的连续内存**而不得不提前触发另一次垃圾收集动作。



**2）复制算法**

“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。**当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉**。

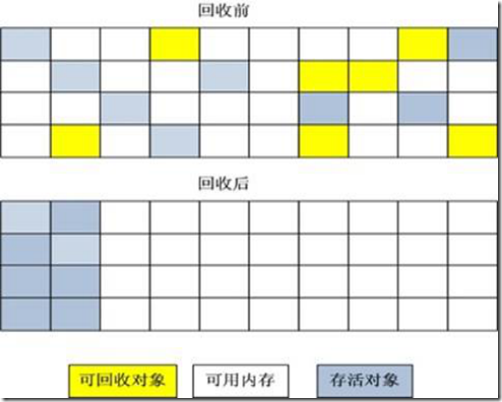
这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，**持续复制长生存期的对象则导致效率降低**。



**3）标记 – 整理算法**

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在**老年代一般不能直接选用这种算法**。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。



**4）分代收集算法**

GC分代的**基本假设**：绝大部分对象的生命周期都非常短暂，存活时间短。

“分代收集”（Generational Collection）算法，把**Java堆分为新生代和老年代**，这样就可以根据各个年代的特点采用最适当的收集算法。在**新生代**中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而**老年代**中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

**注：新生代复制算法，老年代“标记–整理”或“标记-清除”算法**

**7、JVM调优参数**

<http://www.cnblogs.com/redcreen/archive/2011/05/04/2037057.html>

Survivor区是年轻代对象向老年代对象过渡的中间内存区域

* -Xms：初始堆大小
* -Xmx：最大堆大小
* -Xmn：年轻代大小。**注意：**此处的大小是（eden+ 2 survivor space)，与jmap -heap中显示的New gen是不同的。**堆大小=年轻代大小 + 年老代大小 + 持久代大小**。增大年轻代后，将会减小年老代大小，此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。（**新版本的JDK中，把永久代分配到堆中**）
* -XX:NewSize：设置年轻代大小；-XX:MaxNewSize：年轻代最大值
* -XX:PermSize：设置持久代(perm gen)初始值；-XX:MaxPermSize：设置持久代最大值
* -Xss：每个线程的堆栈大小（线程私有）
* -XX:NewRatio：年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代)
* -XX:SurvivorRatio：年轻代内Eden区与Survivor区的大小比值。设置为8，则两个Survivor区与一个Eden区的比值为2:8，**一个Survivor区占整个年轻代的1/10**。
* -XX:MaxTenuringThreshold ：垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概率。**该参数只有在串行GC时才有效**。

对于GC的性能主要有2个方面的指标：**吞吐量throughput**（工作时间不算gc的时间占总的时间比）和**暂停pause**（gc发生时app对外显示的无法响应）。

**8、JVM调优经验**

<http://www.cnblogs.com/redcreen/archive/2011/05/04/2037057.html>

* **年轻代大小选择**
* 响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制(根据实际情况选择)。在此种情况下，年轻代收集发生的频率也是最小的。同时减少到达年老代的对象
* 吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用
* 避免设置过小：当新生代设置过小时，会导致YGC次数更加频繁，可能导致YGC对象直接进入旧生代，如果此时旧生代满了，会触发FGC。
* **老年代大小选择**
* 响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成**内存碎 片**，高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案,一般需要参考以下数据获得：并发垃圾收集信息、持久代并发收集次数、传统GC信息、花在年轻代和年老代回收上的时间比例
* 吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象
* **较小堆引起的碎片问题**

因为年老代的并发收集器使用标记-清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现**"碎片"**，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记-清除方式进行回收。如果出现"碎片"，可能需要进行如下配置：

-XX:+UseCMSCompactAtFullCollection:使用并发收集器时，开启对年老代的压缩

-XX:CMSFullGCsBeforeCompaction=0:上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

* XMX和XMS设置一样大，MaxPermSize和MinPermSize设置一样大，这样可以减轻伸缩堆大小带来的压力
* 使用CMS的好处是用尽量少的新生代，经验值是128M－256M，然后老生代利用CMS并行收集，这样能保证系统低延迟的吞吐效率。 实际上cms的收集停顿时间非常的短，2G的内存，大约20－80ms的应用程序停顿时间。
* 系统停顿的时候可能是GC的问题也可能是程序的问题，多用**jmap和jstack**查看，或者killall -3 java，然后查看java控制台日志，能看出很多问题
* 采用并发回收时，年轻代小一点，年老代要大，因为年老大用的是并发回收，即使时间长点也不会影响其他程序继续运行，网站不会停顿
* JVM参数的设置(特别是 –Xmx –Xms –Xmn -XX:SurvivorRatio -XX:MaxTenuringThreshold等参数的设置没有一个固定的公式，需要根据PV old区实际数据 YGC次数等多方面来衡量。为了避免promotion faild可能会导致xmn设置偏小，也意味着YGC的次数会增多，处理并发访问的能力下降等问题。每个参数的调整都需要经过详细的性能测试，才能找到特定应用的最佳配置

**1）promotion failed**

垃圾回收时promotion failed是个很头痛的问题，一般可能是两种原因产生：

* **第一个原因：**是救助空间（Survivor）不够，救助空间里的对象还不应该被移动到年老代，但年轻代又有很多对象需要放入救助空间；
* **第二个原因：**是年老代没有足够的空间接纳来自年轻代的对象；这两种情况都会转向Full GC，网站停顿时间较长。

**解决方方案一：**

* 第一个原因我的最终解决办法是去掉救助空间，设置-XX:SurvivorRatio=65536 -XX:MaxTenuringThreshold=0即可。
* 第二个原因我的解决办法是设置**CMSInitiatingOccupancyFraction**为某个值（假设70），这样年老代空间到70%时就开始执行CMS，年老代有足够的空间接纳来自年轻代的对象。

**解决方案一的改进方案：**

又有改进了，上面方法不太好，因为没有用到救助空间，所以年老代容易满，CMS执行会比较频繁。我改善了一下，还是用救助空间，但是把救助空间加大，这样也不会有promotion failed。具体操作上，32位Linux和64位Linux好像不一样，64位系统似乎只要配置MaxTenuringThreshold参数，CMS还是有暂停。为了解决**暂停问题和promotion failed问题**，最后我设置-XX:SurvivorRatio=1 ，并把MaxTenuringThreshold去掉，这样即没有暂停又不会有promotoin failed，而且更重要的是，年老代和永久代上升非常慢（因为好多对象到不了年老代就被回收了），所以CMS执行频率非常低，好几个小时才执行一次，这样，服务器都不用重启了。

**2）CMSInitiatingOccupancyFraction值与Xmn的关系公式**

promontion faild产生的原因是EDEN空间不足的情况下，将EDEN与From survivor中的存活对象存入To survivor区时，To survivor区的空间不足，再次晋升到old gen区，而old gen区内存也不够的情况下，产生了promontion faild，从而导致full gc。

**9、JVM调优工具（方法）**

<http://pengjiaheng.iteye.com/blog/552456>

<http://pengjiaheng.iteye.com/category/86293> java JVM调优序列文章

* **Jconsole :** jdk自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。
* **JProfiler：**商业软件，需要付费。功能强大。
* **VisualVM（jvisualvm）：**JDK自带，功能强大，与JProfiler类似。推荐。

**1）堆信息查看**

* 可查看堆空间大小分配（年轻代、年老代、持久代分配）
* 提供即时的垃圾回收功能
* 垃圾监控（长时间监控回收情况）
* 查看堆内类、对象信息查看：数量、类型等
* 对象引用情况查看

有了堆信息查看方面的功能，我们一般可以顺利**解决以下问题**：年老代年轻代大小划分是否合理、内存泄漏、垃圾回收算法设置是否合理

**2）线程监控**

* 线程信息监控：系统线程数量。
* 线程状态监控：各个线程都处在什么样的状态下
* Dump线程详细信息：查看线程内部运行情况
* 死锁检查

**3）热点分析**

* **CPU热点：**检查系统哪些方法占用的大量CPU时间
* **内存热点：**检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

**4）快照**

快照是系统运行到某一时刻的一个定格。在我们进行调优的时候，不可能用眼睛去跟踪所有系统变化，依赖快照功能，我们就可以进行系统两个不同运行时刻，对象（或类、线程等）的不同，以便快速找到问题

举例说，我要检查系统进行垃圾回收以后，是否还有该收回的对象被遗漏下来的了。那么，我可以在进行垃圾回收前后，分别进行一次堆情况的快照，然后对比两次快照的对象情况。

**5）内存泄露检查**

内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

内存泄漏一般可以理解为**系统资源（各方面的资源，堆、栈、线程等）**在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。

内存泄漏对系统危害比较大，因为可以直接导致系统的崩溃。需要区别一下，内存泄漏和系统超负荷两者是有区别的，虽然可能导致的最终结果是一样的。内存泄漏是用完的资源没有回收引起错误，而系统超负荷则是系统确实没有那么多资源可以分配了（其他的资源都在使用）。

**A. 年老代堆空间被占满**

* **异常：**java.lang.OutOfMemoryError: Java heap space



* **说明：**这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。

如上图所示，这是非常典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有谷底部分表示是一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随时间的推移，系统的堆空间被不断占满，最终会占满整个堆空间。因此可以初步认为系统内部可能有内存泄漏。（上面的图仅供示例，在实际情况下收集数据的时间需要更长，比如几个小时或者几天）

* **解决：**这种方式解决起来也比较容易，一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

**B.** **持久代被占满**

* **异常：**java.lang.OutOfMemoryError: PermGen space
* **说明：**Perm空间被占满。无法为新的class分配存储空间而引发的异常。这个异常以前是没有的，但是在**Java反射**大量使用的今天这个异常比较常见了。主要原因就是大量动态反射生成的类不断被加载，最终导致Perm区被占满。

更可怕的是，不同的classLoader即便使用了相同的类，但是都会对其进行加载，相当于同一个东西，如果有N个classLoader那么他将会被加载N次。因此，某些情况下，这个问题基本视为无解。当然，存在大量classLoader和大量反射类的情况其实也不多。

* **解决：**-XX:MaxPermSize=16m；换用JDK，比如JRocket。

**C.** **堆栈溢出**

* **异常：**java.lang.StackOverflowError
* **说明：**这个就不多说了，一般就是递归没返回，或者循环调用造成

**D. 线程堆栈满**

* **异常：**Fatal: Stack size too small
* **说明：**java中一个线程的空间大小是有限制的。JDK5.0以后这个值是1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。
* **解决：**增加线程栈大小。-Xss=2m。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

**E.** **系统内存被占满**

* **异常：**java.lang.OutOfMemoryError: unable to create new native thread
* **说明：**这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。

分配给Java虚拟机的内存愈多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。同时，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总内存生产更多的线程数。

* **解决：**

1. 重新设计系统减少线程数量。

2. 线程数量不能减少的情况下，通过-Xss减小单个线程大小。以便能生产更多的线程。

**10、Java对象的访问方式**

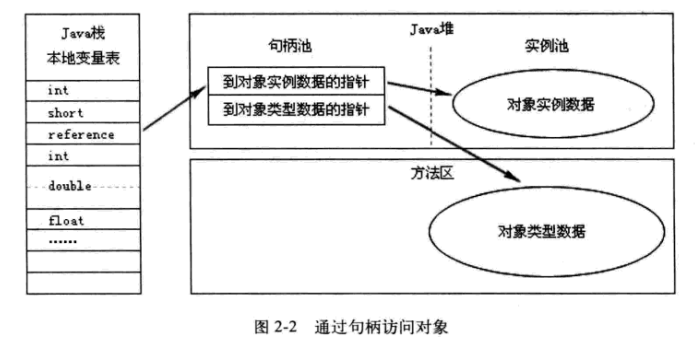
<http://www.cnblogs.com/zhguang/p/3257367.html> 该博客系列文章讲解很透彻

一般来说，一个Java的引用访问涉及到3个内存区域：JVM栈，堆，方法区。以最简单的本地变量引用：Object obj = new Object()为例：

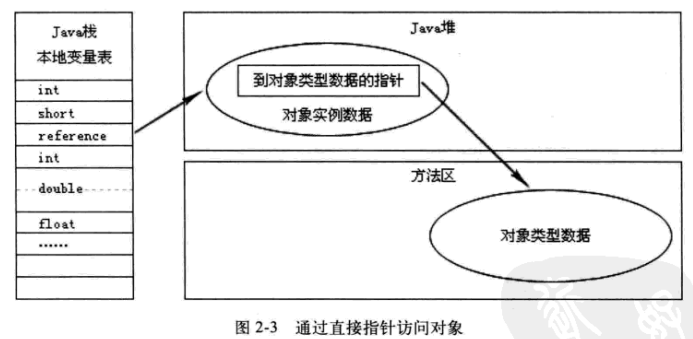
* Object obj表示一个本地引用，存储在JVM栈的本地变量表中，表示一个reference类型数据；
* new Object()作为实例对象数据存储在堆中；
* 堆中还记录了Object类的类型信息（接口、方法、field、对象类型等）的地址，这些地址所执行的数据存储在方法区中；

在Java虚拟机规范中，对于通过reference类型引用访问具体对象的方式并未做规定，目前主流的实现方式主要有两种：

* **通过句柄访问：**通过句柄访问的实现方式中，JVM堆中会专门有一块区域用来作为句柄池，存储相关句柄所执行的实例数据地址（包括在堆中地址和在方法区中的地址）。这种实现方法由于用句柄表示地址，因此十分稳定。



* **通过直接指针访问：**通过直接指针访问的方式中，reference中存储的就是对象在堆中的实际地址，在堆中存储的对象信息中包含了在方法区中的相应类型数据。这种方法最大的优势是速度快，在HotSpot虚拟机中用的就是这种方式。

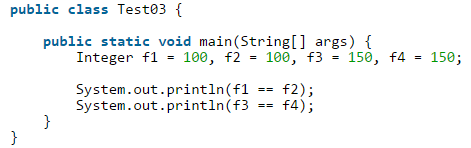


**三、Java基础知识面试题**

<http://www.importnew.com/22083.html>

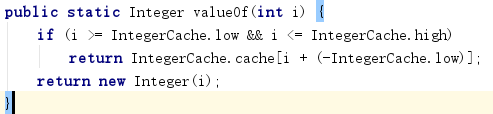
**1、int和Integer有什么区别？**

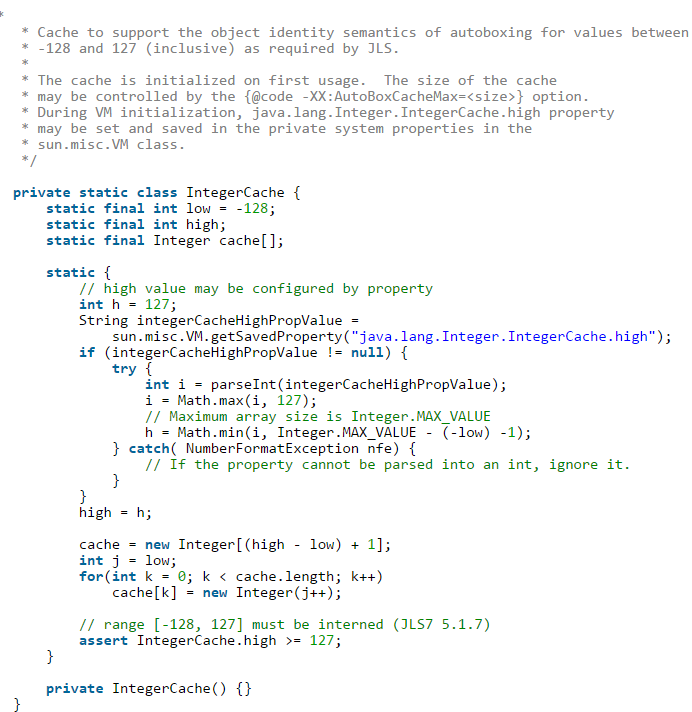
Java是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的**包装类型（wrapper class、**，int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。



**输出结果：**true、false

**分析**：如果不明就里很容易认为两个输出要么都是true要么都是false。首先需要注意的是f1、f2、f3、f4四个变量都是**Integer对象引用**，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf，如果看看valueOf的源代码就知道发生了什么。





简单的说，如果整型字面量的值在-128到127之间，那么不会new新的Integer对象，而是**直接引用常量池中的Integer对象**，所以上面的面试题中f1==f2的结果是true，而f3==f4的结果是false。

**2、jvm性能调优监控工具**

* **jps：**输出JVM中运行的进程状态信息
* **jstack：**查看某个Java进程内的线程堆栈信息。jstack可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码
* **jmap和jhat：**jmap导出堆内存，然后使用jhat来进行分析。jmap用来查看堆内存使用状况
* 打印进程的类加载器和类加载器加载的持久代对象信息：jmap -permstat pid
* 查看进程堆内存使用情况：包括使用的GC算法、堆配置参数和各代中堆内存使用：jmap -heap pid
* 查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象：jmap -histo[:live] pid
* **jstat：**JVM统计监测工具，看看各个区内存和GC的情况

**3、&和&&的区别？**

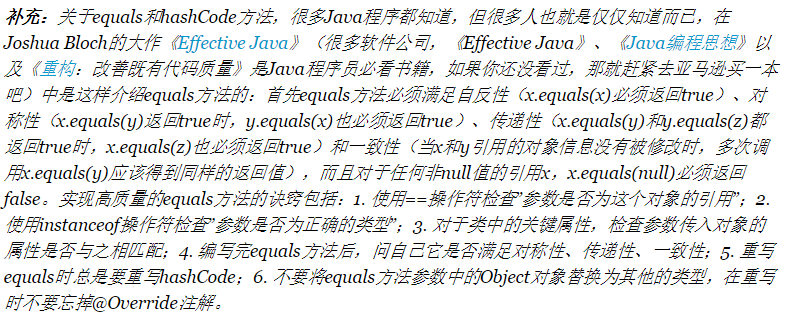
&运算符有两种用法：(1)按位与；(2)逻辑与。**&&运算符是短路与运算**。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是true整个表达式的值才是true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是null而且不是空字符串，应当写为：username != null &&!username.equals(“”)，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的equals比较，否则会产生NullPointerException异常。注意：**逻辑或运算符**（|、和**短路或运算符**（||、的差别也是如此。

**4、解释内存中的栈(stack)、堆(heap)和静态区(static area)的用法**

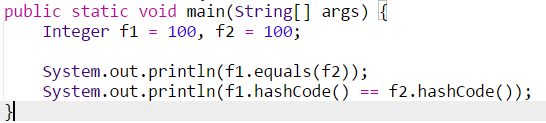
通常定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过**new关键字和构造器**创建的对象放在堆空间；程序中的**字面量（literal、**如直接书写的100、”hello”和常量都是放在静态区中。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用。

**5、两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？**

不对，如果两个对象x和y满足x.equals(y) == true，它们的哈希码（hash code、应当相同。Java对于**eqauls方法和hashCode方法**是这样规定的：(1)**如果两个对象相同（equals方法返回true、，那么它们的hashCode值一定要相同**；(2)**如果两个对象的hashCode相同，它们并不一定相同**。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在Set集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降、



例子：返回true、true



**6、是否可以继承String类？**

String 类是final类，不可以被继承。

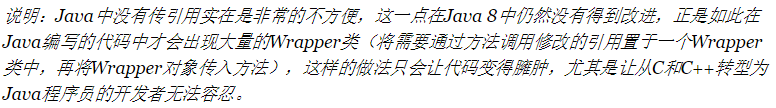
**补充：**继承String本身就是一个错误的行为，对String类型最好的重用方式是关联关系（Has-A、和依赖关系（Use-A、，而不是继承关系（Is-A、。

**事实上，所有基本类型的包装类均不可被继承**

**7、当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？**

<http://blog.csdn.net/cauchyweierstrass/article/details/49047217>

**值传递**。Java语言的方法调用**只支持参数的值传递**。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的**属性可以在被调用过程中被改变**，但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。在C#中可以编写如下所示的代码，但是在Java中却做不到。



**Java中所说的按引用传递实质上是传递该对象的地址，该地址其实是按值传递的，通过这个地址可以修改其指向内存处对象的值。改变该地址的值毫无意义，只会失去对真实对象的掌控。**

**8、String和StringBuilder、StringBuffer的区别？**

* **可变与不可变**

String类中使**用字符数组保存字符串**，如下就是，因为有**“final”修饰符**，所以可以知道string对象是不可变的：**private final char value[];**

String 为不可变对象，一旦被创建，就不能修改它的值。对于已经存在的String对象的修改都是重新创建一个新的对象，然后把新的值保存进去。

StringBuilder与StringBuffer都继承自**AbstractStringBuilder**类，在AbstractStringBuilder中也是**使用字符数组保存字符串**，如下就是，可知这两种对象都是可变的：**char[] value;**

StringBuffer：是一个可变对象，当对他进行修改的时候不会像String那样重新建立对象，它只能通过构造函数来建立，如：**StringBuffer sb = new StringBuffer();**

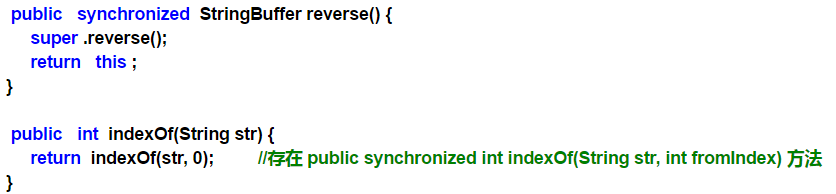
不能通过赋值符号对他进行赋值，如 **sb = "welcome to here!"; //error**

对象被建立以后，在内存中就会分配内存空间，并初始保存一个null，向StringBuffer中赋值的时候可以通过它的append方法：b.append("hello");

* **是否多线程安全**

String中的对象是不可变的，也就可以理解为**常量**， 显然**线程安全** 。

AbstractStringBuilder是StringBuilder与StringBuffer的公共父类，定义了一些字符串的基本操作，如**expandCapacity、append、insert、indexOf**等公共方法。**StringBuffer**对方法加了**同步锁或者对调用的方法加了同步锁**，所以是**线程安全**的 。看如下源码：



**StringBuilder**并没有对方法进行加同步锁，所以是**非线程安全**的 。

* **StringBuilder与StringBuffer共同点**

StringBuilder与StringBuffer有**公共父类AbstractStringBuilder**( 抽象类 )。

**抽象类与接口的其中一个区别**是：抽象类中可以定义一些子类的公共方法，子类只需要增加新的功能，不需要重复写已经存在的方法；而接口中只是对方法的申明和常量的定义。

StringBuilder、StringBuffer的方法都会调用AbstractStringBuilder中的公共方法，如super.append(...)。**只是StringBuffer会在方法上加synchronized关键字，进行同步**。

最后，如果**程序不是多线程的，那么使用StringBuilder效率高于StringBuffer**。

**效率比较**String < StringBuffer < StringBuilder，但是在String S1 =“This is only a”+“simple”+“test”时，**String效率最高**。

**9、常见的磁盘调度算法**

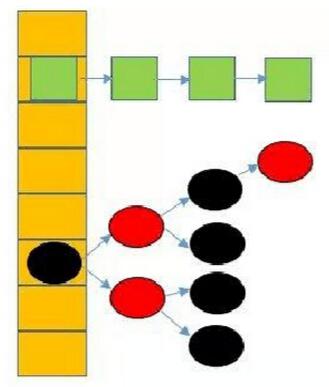
* FCFS（先到先服务算法）：这种算法的思想比较容易理解。假设当前磁道在某一位置，依次处理服务队列里的每一个磁道，这样做的优点是处理起来比较简单，但缺点是磁头移动的距离和平均移动距离会很大。
* SSTF（最短寻道时间算法）：算法的本质是利用贪心算法来实现，假设当前磁道在某一位置，接下来处理的是距离当前磁道最近的磁道号，处理完成之后再处理离这个磁道号最近的磁道号，直到所有的磁道号都服务完了程序结束。这样做的优点是性能会优于FIFO算法，但会产生距离当前磁道较远的磁道号长期得不到服务，也就是“饥饿”现象。
* SCAN（扫描算法或电梯调度算法）：当设备无访问请求时，磁头不动；当有访问请求时，磁头按一个方向移动，在移动过程中对遇到的访问请求进行服务，然后判断该方向上是否还有访问请求，如果有则继续扫描；否则改变移动方向，并为经过的访问请求服务，如此反复。
* CSCAN（循环扫描算法）：算法的思想是，访问完最里面一个要求服务的序列之后，从最外层的序号开始往里走。也就是**始终保持一个方向**。

**10、JDK7 与 JDK8 中 HashMap 的实现**

直到JDK7为止，HashMap的结构都是这么简单，基于一个**数组以及多个链表**的实现，hash值冲突的时候，就将对应节点以链表的形式存储。

这样子的HashMap性能上就抱有一定疑问，如果说成百上千个节点在hash时发生碰撞，存储一个链表中，那么如果要查找其中一个节点，那就不可避免的花费O(N)的查找时间，这将是多么大的性能损失。这个问题终于在JDK8中得到了解决。再最坏的情况下，链表查找的时间复杂度为O(n)，而**红黑树一直是O(logn)**，这样会提高HashMap的效率。

**JDK7中HashMap采用的是位桶+链表的方式**，即我们常说的散列链表的方式，而**JDK8中采用的是位桶+链表/红黑树**（有关红黑树请查看红黑树、的方式，也是非线程安全的。当某个位桶的链表的长度达到某个阀值的时候，这个链表就将转换成红黑树。

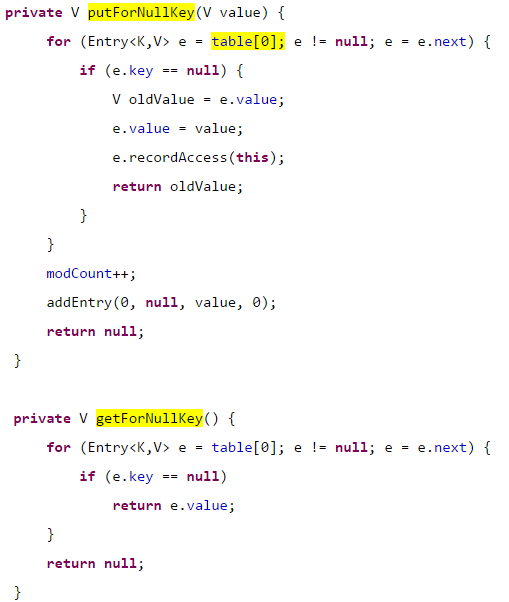


JDK8中，当同一个hash值的节点数不小于8时，将不再以单链表的形式存储了，会被调整成一颗红黑树（上图中null节点没画、。这就是**JDK7与JDK8中HashMap实现的最大区别**。

jdk7中hashmap实现原理和jdk8中hashmap的改进方法总结

<http://blog.csdn.net/xuehuagongzi000/article/details/71449179>

**null key的存取：**null key总是**存放在Entry[]数组的第一个元素**。



Jdk1.8中的HashMap实现原理

<http://blog.csdn.net/fjse51/article/details/53811465>

* **重新调整HashMap大小存在什么问题吗？**

当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免**尾部遍历(tail traversing)**。如果条件竞争发生了，那么就死循环了。因此在并发环境下，我们使用CurrentHashMap来替代HashMap

* **为什么String， Interger这样的wrapper类（包装类、适合作为键？（键的唯一性、**

因为String是**不可变的，也是final的（不变性、**，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能

* **如果两个键的hashcode相同，你如何获取值对象？**

找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。因此，设计HashMap的key类型时，如果使用**不可变的、声明作final的对象**，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择

* **HashMap的工作原理吗？get和put的原理吗？**

通过hash的方法，通过put和get存储和获取对象。

* 存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Facotr则resize为原来的2倍)。
* 获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，Hashmap通过链表将产生碰撞冲突的元素组织起来。
* 在Java 8中，如果一个bucket中**碰撞冲突**的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。
* **equals()和hashCode()的都有什么作用？**

通过对key的hashCode()进行hashing，并计算下标( n-1 & hash)，从而获得buckets的位置。如果产生**碰撞**，则利用key.equals()方法去链表或树中去查找对应的节点

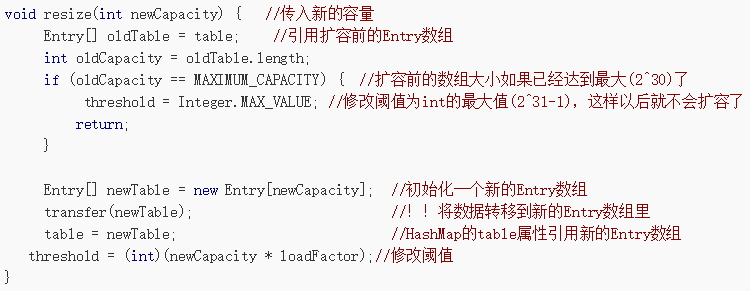
* **hash的实现吗？为什么要这样实现？**

在Java 1.8的实现中，是通过**hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)**，主要是从速度、功效、质量来考虑的，这么做可以在bucket的n比较小的时候，也能保证考虑到**高低bit都参与到hash的计算**中，同时不会有太大的开销。

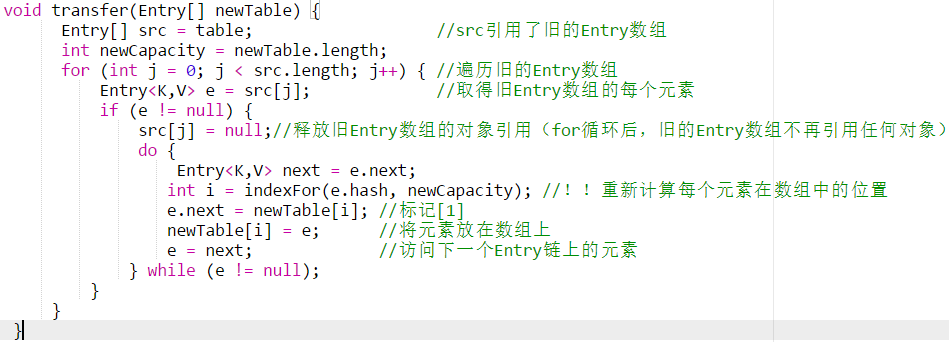
**11、JDK7 与 JDK8 中 HashMap 扩容方式的变化**

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

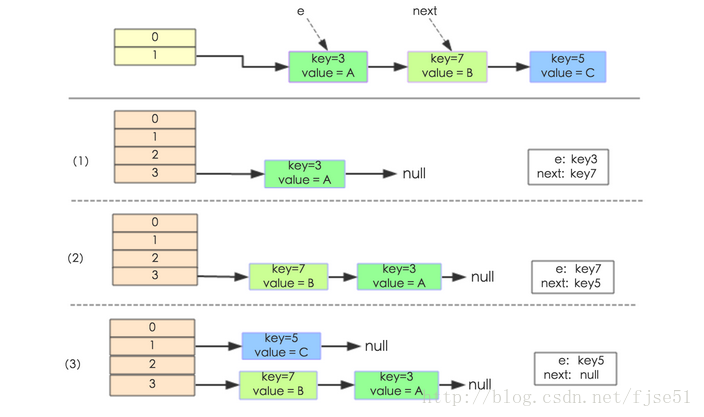
**JDK1.7的代码：**



使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里

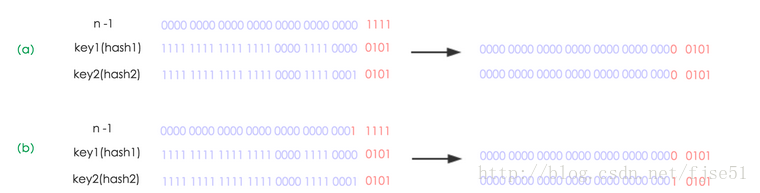


newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话、，这一点和Jdk1.8有区别，下文详解。**在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上**。

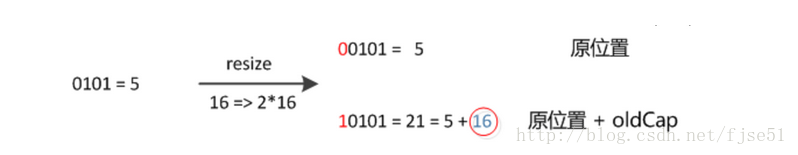


**JDK8 HashMap扩容的方式：**

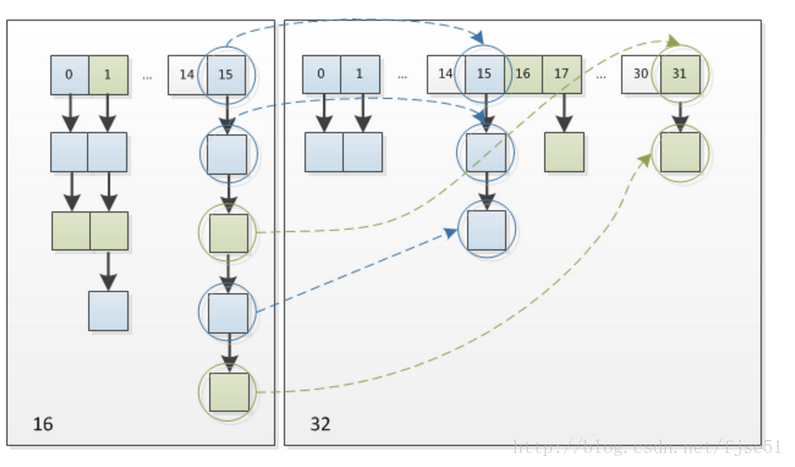
JDK1.8做了哪些优化。经过观测可以发现，使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图（a、表示扩容前的key1和key2两种key确定索引位置的示例，图（b、表示扩容后key1和key2两种key确定索引位置的示例，其中**hash1是key1对应的哈希与高位运算结果**。



元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



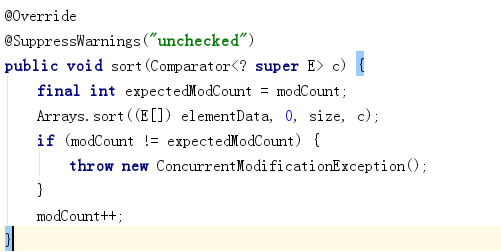
因此，在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



**12、ArrayList排序实现？**

使用Arrays.sort()方法进行排序（快速排序DualPivotQuicksort）

并发判断，通过对modCount值的比较



**13、快排算法原理**

<http://www.cnblogs.com/yuxiaofei93/p/5722714.html>

<http://www.cnblogs.com/nullzx/p/5880191.html>

<http://rerun.me/2013/06/13/quicksorting-3-way-and-dual-pivot/>

<https://github.com/arunma/DataStructuresAlgorithms>

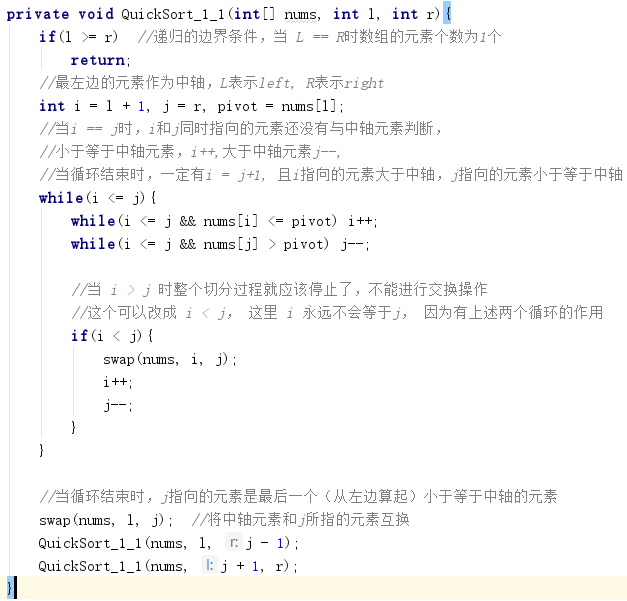
**1）单轴快速排序的基本原理**

快速排序的**基本思想**就是从一个数组中任意挑选一个元素（通常来说会选择最左边的元素、作为中轴元素，将剩下的元素以中轴元素作为比较的标准，将小于等于中轴元素的放到中轴元素的左边，将大于中轴元素的放到中轴元素的右边，然后以当前中轴元素的位置为界，将左半部分子数组和右半部分子数组看成两个新的数组，重复上述操作，直到子数组的元素个数小于等于1（因为一个元素的数组必定是有序的）。

快速排序中最重要的就是步骤就是将小于等于中轴元素的放到中轴元素的左边，将大于中轴元素的放到中轴元素的右边，我们暂时把这个步骤定义为**切分**。而剩下的步骤就是进行递归而已，递归的边界条件为数组的元素个数小于等于1。以首元素作为中轴，看看**常见的切分方式**。

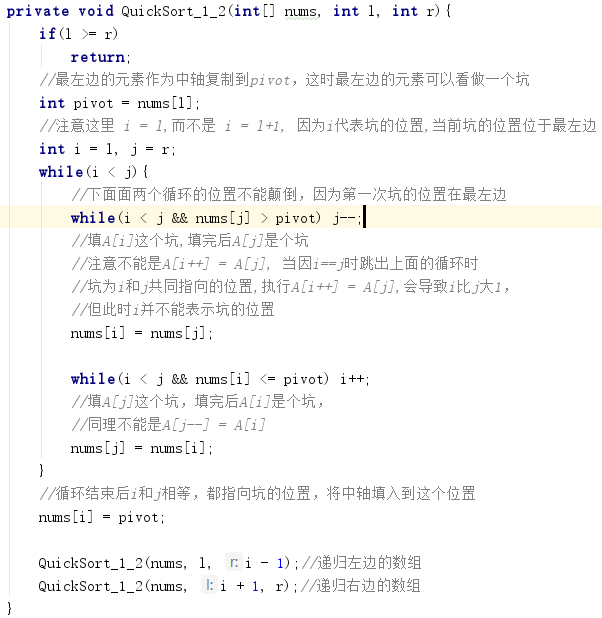
**A. 从两端扫描交换的方式**

基本思想，使用两个变量i和j，i指向首元素的元素下一个元素（最左边的首元素为中轴元素），j指向最后一个元素，我们从前往后找，直到找到一个**比中轴元素大的**，然后从后往前找，直到找到一个**比中轴元素小的**，然后**交换这两个元素**，直到这两个变量交错（i > j、（注意不是相遇 i == j，因为相遇的元素还未和中轴元素比较、。最后对左半数组和右半数组重复上述操作。



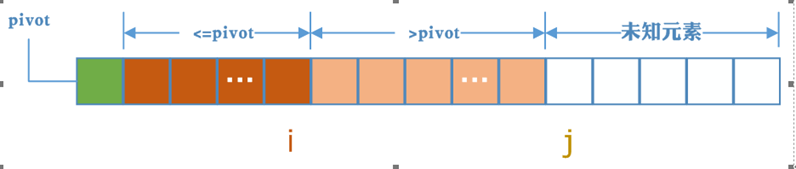
**B.两端扫描，一端挖坑，另一端填补**

基本思想，使用两个变量i和j，i指向最左边的元素，j指向最右边的元素，我们将首元素作为中轴，将首元素复制到变量pivot中，这时我们可以将首元素i所在的位置看成一个坑，我们从j的位置从右向左扫描，找一个小于等于中轴的元素A[j]，来填补A[i]这个坑，填补完成后，拿去填坑的元素所在的位置j又可以看做一个坑，这时我们在以i的位置从前往后找一个大于中轴的元素来填补A[j]这个新的坑，如此往复，直到i和j相遇（i == j，此时i和j指向同一个坑）。最后我们将中轴元素放到这个坑中。最后对左半数组和右半数组重复上述操作。



**C. 单端扫描**

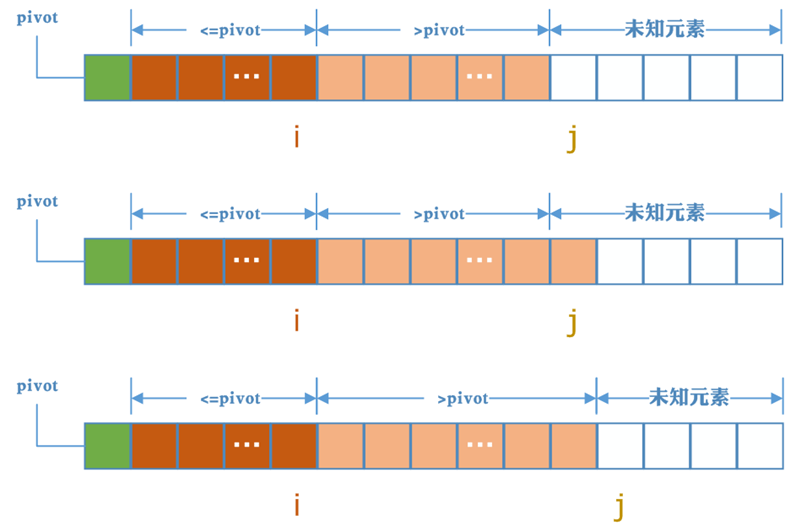
j从左向右扫描，A[1，i]表示小于等于pivot的部分，A[i+1，j-1]表示大于pivot的部分，A[j， R]表示未知元素



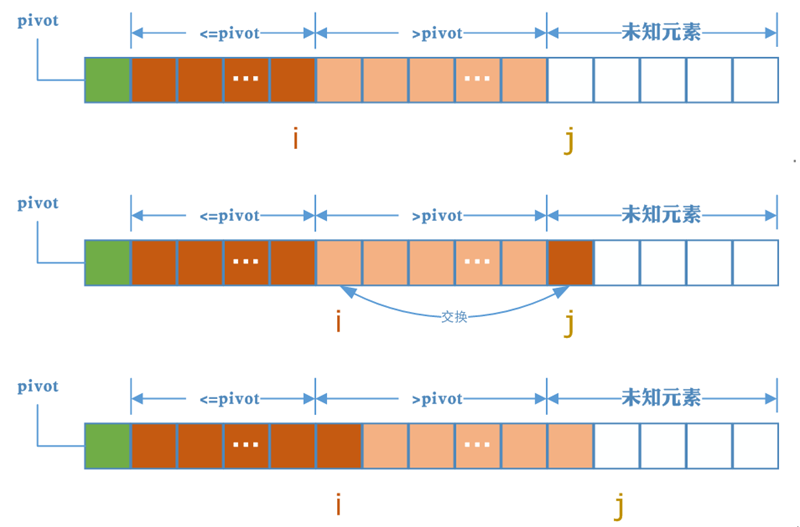
初始化时，选取最左边的元素作为中轴元素，A[1，i]表示小于等于pivot的部分，**i指向中轴元素（i < 1）**，表示小于等于pivot的元素个数为0，j以后的都是未知元素（即不知道比pivot大，还是比中轴元素小），j初始化指向第一个未知元素。



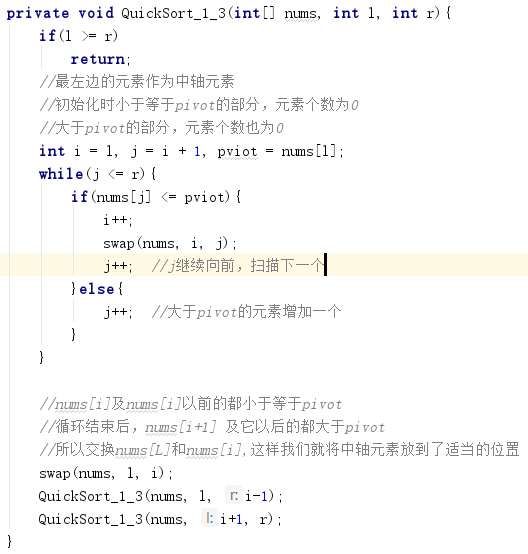
当A[j]大于pivot时，j继续向前，此时大于pivot的部分就增加一个元素



上图中假设对A[j]与pivot比较后发现A[j]大于pivot时，j的变化。当A[j]小于等于pivot时，我们注意注意**i的位置，i的下一个就是大于pivot的元素**，我们将i增加1然后交换A[i]和A[j]，交换后小于等于pivot的部分增加1，j增加1，继续扫描下一个。而i的下一个元素仍然大于pivot，又回到了先前的状态。下图中假设对A[j]与pivot比较后发现A[j] <= pivot时，i，j的变化。

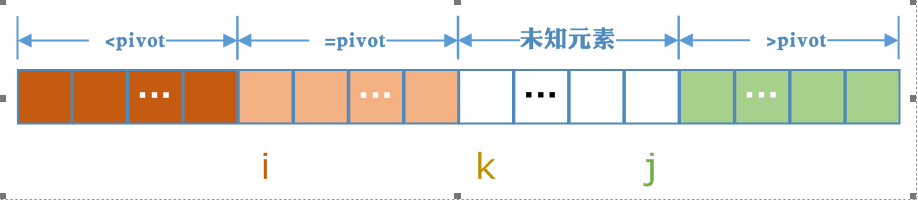


代码实现：

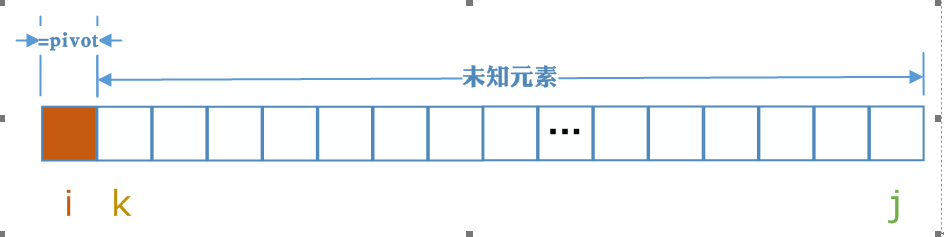


**2）三向切分的快速排序**

三向切分快速排序的**基本思想**，用i，j，k三个将数组切分成四部分，a[L， i-1]表示小于pivot的部分，a[i， k-1]表示等于pivot的部分，a[j+1]表示大于pivot的部分，而a[k， j]表示未判定的元素（即不知道比pivot大，还是比中轴元素小）。我们要注意a[i]始终位于等于pivot部分的第一个元素，a[i]的左边是小于pivot的部分。



我们选取最左边的元素作为中轴元素，初始化时，i = L，k = L+1，j=R（L表示最左边元素的索引，R表示最右边元素的索引、

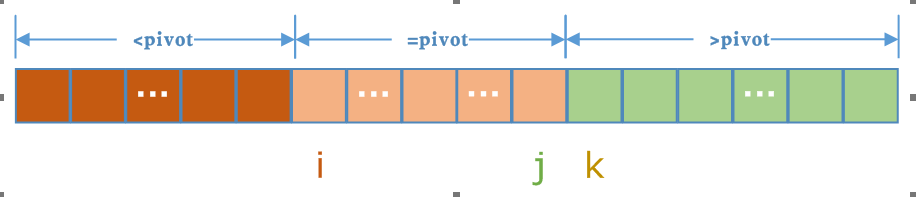


通过上一段的表述可知，初始化时<pivot部分的元素个数为0，等于pivot部分元素的个数为1，大于pivot部分的元素个数为0，这显然符合目前我们对所掌握的情况。k自左向右扫描直到k与j错过为止（k > j、。我们扫描的目的就是逐个减少未知元素，并将每个元素按照和pivot的大小关系放到不同的区间上去。

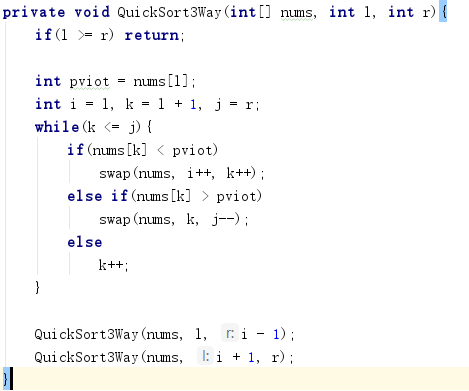
在k的扫描过程中我们可以对a[k]分为**三种情况讨论：**

* a[k] < pivot 交换a[i]和a[k]，然后i和k都自增1，k继续扫描
* a[k] = pivot k自增1，k接着继续扫描
* a[k] > pivot 这个时候显然a[k]应该放到最右端，**大于pivot的部分**。但是我们不能直接将a[k]与a[j]交换，因为目前a[j]和pivot的关系未知，所以我们这个时候应该从j的位置自右向左扫描。而a[j]与pivot的关系可以继续分为三种情况讨论
* a[j] > pivot： j自减1，j接着继续扫描
* a[j] == pivot：交换a[k]和a[j]，k自增1，j自减1，k继续扫描（注意此时j的扫描就结束了、
* a[j] < pivot：此时我们注意到**a[j] < pivot， a[k] > pivot， a[i] == pivot**，那么我们只需要将a[j]放到a[i]上，a[k]放到a[j]上，而a[i]放到a[k]上。然后i和k自增1，j自减1，k继续扫描（注意此时j的扫描就结束了）

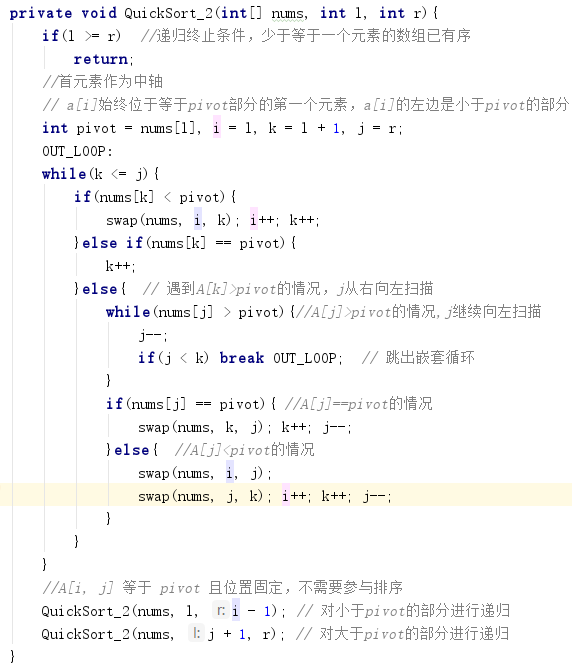
注意，当扫描结束时，**i和j的表示了=等于pivot部分的起始位置和结束位置**。我们只需要对小于pivot的部分以及大于pivot的部分重复上述操作即可。



代码实现：

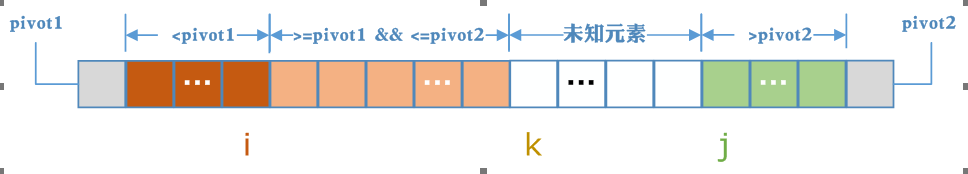


或者



**3）双轴快速排序的基本原理**

双轴快速排序算法思路和三向切分快速排序算法的**思路基本一致**，双轴快速排序算法使用两个轴，通常选取最左边的元素作为pivot1和最右边的元素作pivot2。首先要比较这两个轴的大小，如果pivot1 > pivot2，则交换最左边的元素和最右边的元素，已保证pivot1 <= pivot2。双轴快速排序同样使用i，j，k三个变量将数组分成四部分



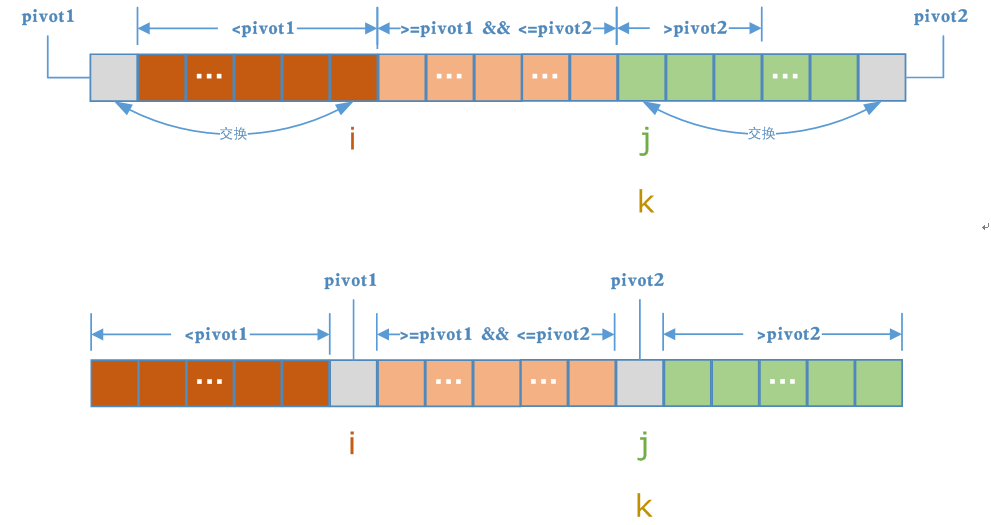
**A[L+1， i]**是小于pivot1的部分，**A[i+1， k-1]**是大于等于pivot1且小于等于pivot2的部分，**A[j， R]**是大于pivot2的部分，而A[k， j-1]是未知部分。和三向切分的快速排序算法一样，初始化i = L，k = L+1，j=R，**k自左向右扫描直到k与j相交为止（k == j）**。我们扫描的目的就是逐个减少未知元素，并将每个元素按照和pivot1和pivot2的大小关系放到不同的区间上去。

在k的扫描过程中我们可以对**a[k]分为三种情况讨论**（注意我们始终保持最左边和最右边的元素，即双轴，不发生交换、

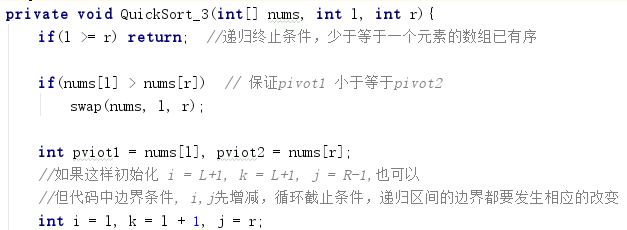
* a[k] < pivot1：i先自增，交换a[i]和a[k]，k自增1，k接着继续扫描
* a[k] >= pivot1 && a[k] <= pivot2：k自增1，k接着继续扫描
* a[k] > pivot2：这个时候显然a[k]应该放到最右端大于pivot2的部分。但此时，我们不能直接将a[k]与j的**下一个位置a[--j]交换**（可以认为A[j]与pivot1和pivot2的大小关系在上一次j自右向左的扫描过程中就已经确定了，这样做主要是j首次扫描时避免pivot2参与其中），因为目前a[--j]和pivot1以及pivot2的关系未知，所以我们这个时候应该从j的下一个位置（--j）自右向左扫描。而a[--j]与pivot1和pivot2的关系可以继续分为三种情况讨论
* a[--j] > pivot2：j接着继续扫描
* a[--j] >= pivot1且a[j] <= pivot2：交换a[k]和a[j]，k自增1，k继续扫描（注意此时j的扫描就结束了）
* a[--j] < pivot1：先将i自增1，此时我们注意到a[j] < pivot1， a[k] > pivot2， pivot1 <= a[i] <=pivot2，那么我们只需要将a[j]放到a[i]上，a[k]放到a[j]上，而a[i]放到a[k]上。k自增1，然后k继续扫描（此时j的扫描就结束了）

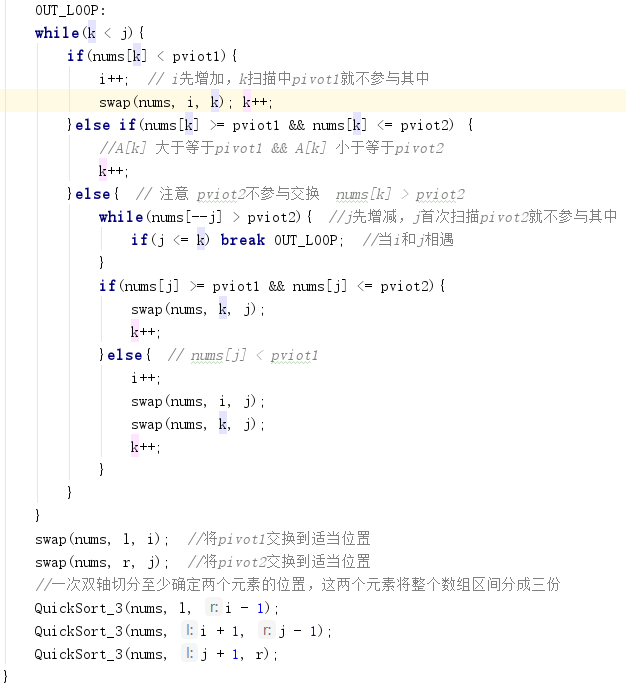
**注意：**

* pivot1和pivot2在始终不参与k，j扫描过程。
* 扫描结束时，A[i]表示了小于pivot1部分的最后一个元素，A[j]表示了大于pivot2的第一个元素，这时我们只需要交换pivot1（即A[L]、和A[i]，交换pivot2（即A[R]、与A[j]，同时我们可以**确定A[i]和A[j]所在的位置在后续的排序过程中不会发生变化**（这一步非常重要，否则可能引起无限递归导致的**栈溢出**、，最后我们只需要对A[L， i-1]，A[i+1， j-1]，A[j+1， R]这三个部分继续递归上述操作即可。



代码实现：





**14、Java中boolean类型到底占用多少个字节？**

<http://blog.csdn.net/YuanMxy/article/details/74170745>

java规范中，没有明确指出boolean的大小。

如果要分大小，那么到底占几个字节呢？

* **1个bit（1/8个字节）**：boolean类型的值只有true和false两种**逻辑值**，在编译后会使用1和0来表示，这两个数在**内存中按位算**，仅需1位（bit）即可存储，位是计算机最小的存储单位。
* **1个字节：**虽然编译后1和0只需占用1位空间，但计算机处理数据的最小单位是1个字节，1个字节等于8位，实际存储的空间是：**用1个字节的最低位存储，其他7位用0填补**，如果值是true的话则存储的二进制为：0000 0001，如果是false的话则存储的二进制为：0000 0000。
* **4个字节：**在《Java虚拟机规范》一书中的描述：“虽然定义了boolean这种数据类型，但是只对它提供了非常有限的支持。在Java虚拟机中没有任何供boolean值专用的字节码指令，Java语言表达式所操作的**boolean值，在编译之后都使用Java虚拟机中的int数据类型来代替，而boolean数组将会被编码成Java虚拟机的byte数组，每个元素boolean元素占8位**”。也就是说JVM规范指出**boolean当做int处理，也就是4字节，boolean数组当做byte数组处理**，这样我们可以得出boolean类型占了单独使用是4个字节，在数组中是确定的1个字节。

如果认同第三条，那么**存在问题：**那虚拟机为什么要用int来代替boolean呢？为什么不用byte或short，这样不是更节省内存空间吗。经过查阅资料发现，使用int的原因是，对于当下32位的处理器（CPU）来说，一次处理数据是32位（这里不是指的是32/64位系统，而是指CPU硬件层面），32 位 CPU 使用4 个字节是最为节省的，哪怕你是 1 个 bit 他也是占用 4 个字节。因为 **CPU 寻址系统只能 32 位 32 位寻址**，具有高效存取的特点。

* java规范中，没有明确指出boolean的大小。在《Java虚拟机规范》给出了4个字节，和boolean数组1个字节的定义，具体还要看虚拟机实现是否按照规范来，所以1个字节、4个字节都是有可能的。

**15、单例模式实现方法**

<http://www.importnew.com/6461.html>

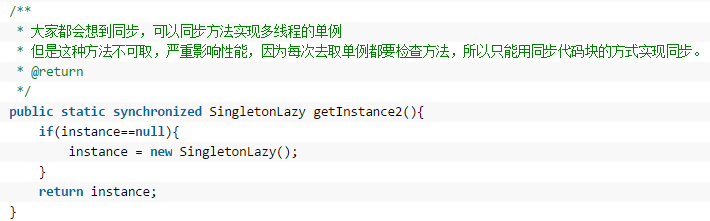
<http://blog.csdn.net/qq_29542611/article/details/52905516>

**1）饿汉模式**

单例是**静态的final变量**，当类第一次加载到内存中的时候就初始化了，所以创建的实例固然是thread-safe。

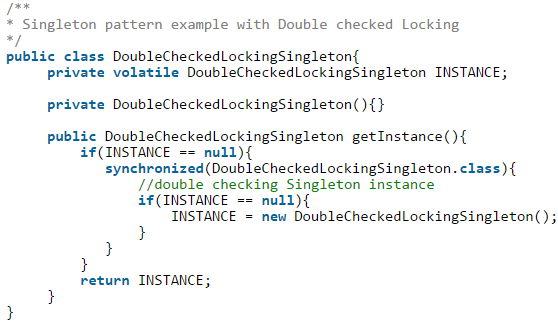


**2）懒汉模式**

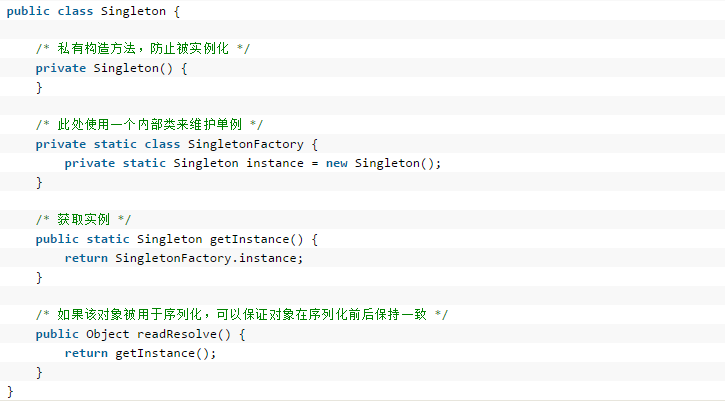


**3）双捡锁DCL（Double Check Lock）**

下面代码就是用double checked locking 方法实现的单例，这里的getInstance()方法要检查两次，确保是否实例INSTANCE是否为null或者已经实例化了，这也是为什么叫double checked locking 模式。



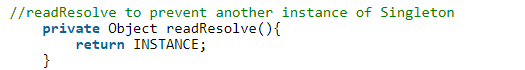
**4）静态工厂（类）**



**4）单例枚举类·**



传统单例存在的另外一个问题是一旦你实现了**序列化接口，那么它们不再保持单例了**，因为readObject()方法一直返回一个新的对象就像java的构造方法一样，你可以通过使用readResolve()方法来避免此事发生。



**创建枚举默认就是线程安全的，你不需要担心double checked locking**。

**总结：**枚举单例有序列化和线程安全的保证，而且只要几行代码就能实现是单例最好的的实现方式，不过你仍然可以使用其它的方式来实现单例。

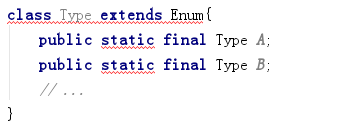
**进一步解释：**

<http://blog.csdn.net/yy254117440/article/details/52305175>

枚举类似类，一个枚举可以拥有成员变量，成员方法，构造方法。枚举最基本的用法：



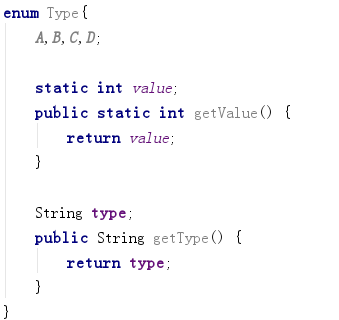
创建enum时，编译器会自动为我们生成一个继承自java.lang.Enum的类，我们上面的enum可以简单看作：



把Type看作一个类，而把A，B，C，D看作类的Type的实例。 当然，这个构建实例的过程不是我们做的，**一个enum的构造方法限制是private的**，也就是不允许我们调用。

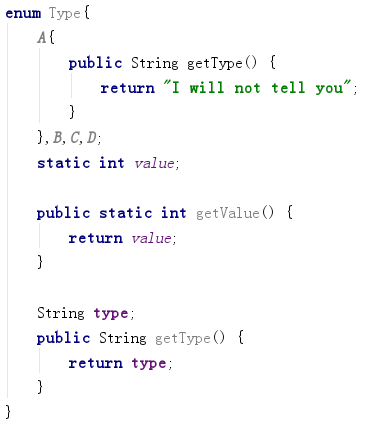
**类”方法和“实例”方法：**

在enum中，我们可以定义类和实例的变量以及方法。代码：

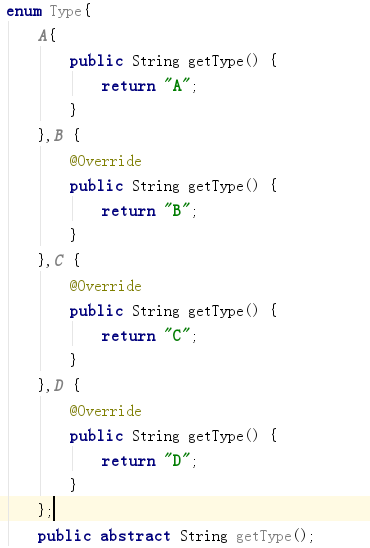


在原有的基础上，**添加了类方法和实例方法**。我们把Type看做一个类，那么enum中静态的域和方法，都可以视作类方法。和我们调用普通的静态方法一样，这里调用类方法也是通过 Type.getValue()即可调用，访问类属性也是通过Type.value即可访问。

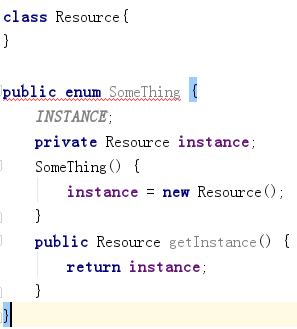
下面的是实例方法，也就是每个实例才能调用的方法。那么实例是什么呢？没错，就是A，B，C，D。所以我们调用实例方法，也就通过 **Type.A.getType()来调用就可以了**。最后，对于某个实例而言，还可以实现自己的实例方法。再看下下面的代码：



A实例后面的{…}就是属于A的实例方法，可以通过覆盖原本的方法，实现属于自己的定制。除此之外，我们还可以添加抽象方法在enum中，强制ABCD都实现各自的处理逻辑：



枚举单例的实现方法：



上面的类Resource是我们要应用单例模式的资源，具体可以表现为网络连接，数据库连接，线程池等等。获取资源的方式很简单，只要 **SomeThing.INSTANCE.getInstance()** 即可获得所要实例。下面我们来看看单例是如何被保证的：

* 首先，在**枚举中我们明确了构造方法限制为私有**，在我们访问枚举实例时会执行构造方法，同时**每个枚举实例都是static final类型的，也就表明只能被实例化一次**。在调用构造方法时，我们的**单例被实例化**。
* 也就是说，因为enum中的实例被保证只会被实例化一次，所以我们的INSTANCE也被保证实例化一次。

可以看到，枚举实现单例还是比较简单的，除此之外我们再来看一下**Enum这个类的声明：**



可以看到，**枚举也提供了序列化机制**。某些情况，比如我们要通过网络传输一个数据库连接的句柄，会提供很多帮助。

**16、JDK中的设计模式**

<http://blog.jobbole.com/62314/>

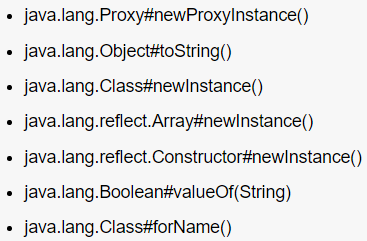
<http://www.iteye.com/news/18725>

<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>

**1）创建模式**

**（1）工厂模式（Factory）**

按照需求返回一个类型的实例（即对实现同一接口的类创建实例：传入参数、方法返回、静态方法构建）。就是一个返回具体对象的方法。

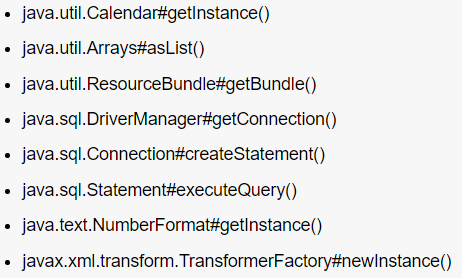


newInstance()：返回一个类对象的实例

class.newInstance()

**（2）抽象工厂模式（Abstract Factory）**

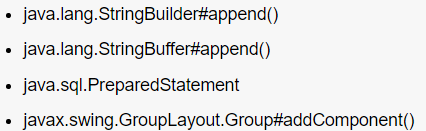
**创建一组有关联的对象实例**。这个模式在JDK中也是相当的常见，还有很多的framework例如Spring。我们很容易找到这样的实例。抽象工厂模式提供了一个协议来生成一系列的相关或者独立的对象，而不用指定具体对象的类型。它使得应用程序能够和使用的框架的具体实现进行解耦。这在JDK或者许多框架比如Spring中都随处可见。它们也很容易识别，**一个创建新对象的方法，返回的却是接口或者抽象类的，就是抽象工厂模式了**。



Arrays.asList()

**（3）构建者模式（Builder）**

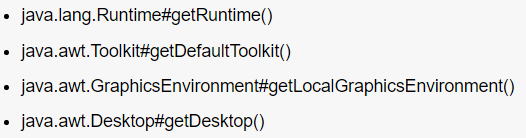
主要用来简化一个复杂的对象的创建。这个模式也可以用来实现一个 Fluent Interface。建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工厂模式的**区别就是**：工厂模式关注的是创建单个产品，而建造者模式则关注创建符合对象、多个部分。定义了一个新的类来构建另一个类的实例，以简化复杂对象的创建。建造模式通常也使用方法链接来实现。



StringBuilder、StringBuffer：append方法链接

**（4）单例模式（Singleton）**

只允许一个实例。在 Effective Java中**建议使用Emun**。



单元素的枚举类型已经成为实现Singleton的最佳方法。

Runtime#getRuntime()：返回与当前Java应用程序**关联的运行时对象**。

**（5）原型模式（Prototype）**

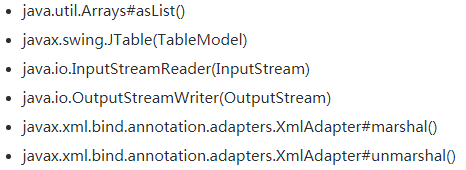
使得**类的实例能够生成自身的拷贝**。如果创建一个对象的实例非常复杂且耗时时，就可以使用这种模式，而不重新创建一个新的实例，你可以拷贝一个对象并直接修改它。



**2）结构型模式**

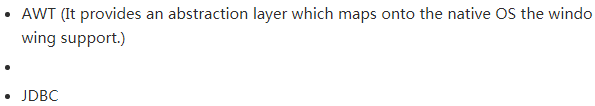
**（1）适配器模式（Adapter）**

用来把一个**接口转化成另一个接口**。集合包中java.util.**Arrays#asList()**、io包中java.io.**InputStreamReader**(**InputStream**)、java.io.**OutputStreamWriter**(**OutputStream**) 等



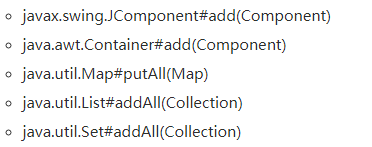
**（2）桥接模式（Bridge）**

这个模式将抽象和抽象操作的实现进行了解耦，这样使得抽象和实现可以独立地变化。



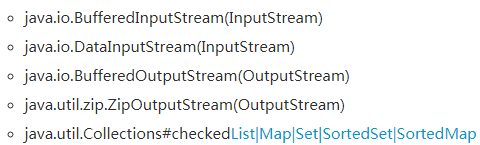
**（3）组合模式（Composite）**

使得客户端看来**单个对象和对象的组合是同等**的。换句话说，某个类型的方法同时也接受自身类型作为参数。



**（4）装饰者模式（Adapter）**

动态的给一个对象附加额外的功能，这也是子类的一种替代方式。可以看到，在创建一个类型的时候，同时也**传入同一类型的对象（实现统一接口的实现类对象）**。这在JDK里随处可见，你会发现它无处不在，所以下面这个列表只是一小部分。



checkedList|Map|Set：返回指定列表|Map|Set的**动态类型安全视图**

装饰模式动态地给一个对象添加一些额外的职责，而且是动态的，要求装饰对象和被装饰对象**实现同一个接口**，装饰对象持有被装饰对象（接口向上转型得到的）的实例，就增加功能来说，它比生成子类更灵活。也可以这样说，装饰模式把复杂类中的核心职责和装饰功能区分开了，这样既简化了复杂类，又去除了相关类中重复的装饰逻辑。装饰模式**没有通过继承原有类来扩展功能，但却达到了一样的目的**，而且比继承更加灵活，所以可以说装饰模式**是继承关系的一种替代方案**。（**具有扩展性**）

**（5）外观（门面）模式**

给一组组件，接口，抽象，或者子系统提供一个简单的接口。

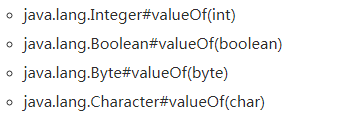


外观模式为子系统中的一组接口提供了同意的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式中，客户对各个具体的子系统是不了解的，所以对这些子系统进行了封装，对外只提供了用户所明白的单一而简单的接口，用户直接使用这个接口就可以完成操作，而不用去理睬具体的过程，而且子系统的变化不会影响到用户，这样就做到了信息隐蔽。

外观模式是为了解决类与类之间的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，**降低了类类之间的耦合度，该模式中没有涉及到接口**。

**（6）享元模式（Flyweight）**

使用缓存来加速大量小对象的访问时间。（缓存：加速小对象的访问时间，类似共享池的作用）



**（7）代理模式（Proxy）**

代理模式是用一个简单的对象来代替一个复杂的或者创建耗时的对象。



**3）行为型模式**

**（1）责任链模式（Proxy）**

通过把请求从一个对象传递到链条中下一个对象的方式，直到请求被处理完毕，以实现对象间的解耦。在这个链上的所有的对象有相同的接口（抽象类），但却有不同的实现。



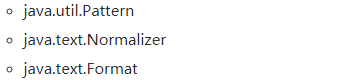
**（2）命令模式**

将操作封装到对象内，以便存储，传递和返回



**（3）解释器模式**

这个模式通常定义了一个语言的语法，然后解析相应语法的语句。



**（4）迭代器模式**

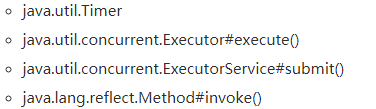
提供一个一致的方法来顺序访问集合中的对象，这个方法与底层的集合的具体实现无关。



顺序访问集中中的对象Iterator和Enumeration

**（5）中介者模式**

通过使用一个中间对象来进行消息分发以及减少类之间的直接依赖。



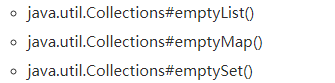
**（6）备忘录模式**

生成对象状态的一个快照，以便对象可以恢复原始状态而不用暴露自身的内容。Date对象通过**自身内部的一个long值来实现备忘录模式**。



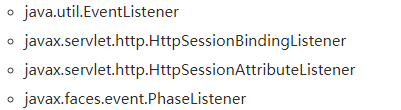
**（7）空对象模式**

这个模式通过一个无意义的对象来代替没有对象这个状态。它使得你不用额外对空对象进行处理。



**（8）观察者模式**

它使得一个对象可以灵活的将消息发送给感兴趣的对象（监听器Listener，AWT中的监听器模式EventListener）。**允许一个对象向所有的侦听的对象广播自己的消息或事件**。



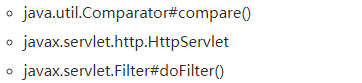
**（9）状态模式**

通过改变对象内部的状态，使得你可以在运行时动态改变一个对象的行为。



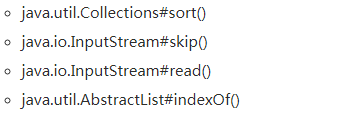
**（10）策略模式**

使用这个模式来将一组算法封装成一系列对象，通过传递这些对象可以灵活的改变程序的功能。（比较器模式Comparator）



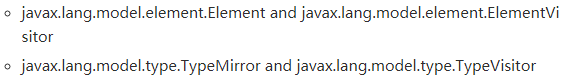
**（11）模板方法模式**

让子类可以重写方法的一部分，而不是整个重写，你可以控制子类需要重写那些操作。一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用。



**（12）访问者模式**

提供一个方便的可维护的方式来操作一组对象。它使得你在不改变操作的对象前提下，可以修改或者扩展对象的行为。



**16、comparable接口和comparator接口实现比较的区别和用法**

<http://www.cnblogs.com/Kevin-mao/p/5912775.html>

Java中通过接口**实现两个对象的比较**，比较常用就是Comparable接口和Comparator接口。首先类要实现接口，并且使用泛型规定要进行比较的对象所属的类，然后类实现了接口后，还需要实现接口定义的比较方法（compareTo方法或者compare方法），在这些方法中传入需要比较大小的另一个**对象**，通过选定的成员变量与之比较，如果大于则返回1，小于返回-1，相等返回0。

**1）comparable接口**

此接口强行对实现它的每个类的对象进行整体排序。此排序被称为该类的自然排序 ，类的 compareTo方法被称为它的自然比较方法 。实现此接口的**对象列表（和数组）**可以通过 Collections.sort（和 Arrays.sort ）进行自动排序。实现此接口的对象可以用作有序映射表中的键或有序集合中的元素，无需指定比较器。

**实现的方法：**

int compareTo(T o)：比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数

* **参数：** o - 要比较的对象。
* **返回：**负整数、零或正整数，根据此对象是小于、等于还是大于指定对象。
* **抛出：**ClassCastException - 如果指定对象的类型不允许它与此对象进行比较

**2）comparator接口**

与上面的Comparable接口不同的是：

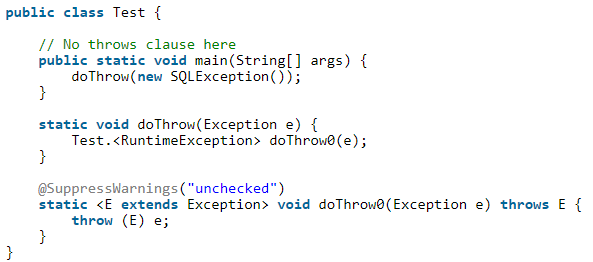
* Comparator位于包java.util下，而Comparable位于包java.lang下。
* Comparable接口将比较代码嵌入需要进行比较的类的自身代码中（即需比较的类自身实现comparable接口），而Comparator接口在一个独立的类中实现比较（即需比较的类与实现comparator接口的比较器类是相互独立的）。
* 如果前期类的设计没有考虑到**类的Compare问题**，而没有实现Comparable接口，后期可以通过Comparator接口来实现比较算法进行排序，并且为了使用不同的排序标准做准备，比如：升序、降序。
* Comparable接口**强制进行自然排序（即按字典排序）**，而Comparator接口不强制进行自然排序，可以指定排序顺序。
* **实现的方法不一致**：Comparable接口compareTo(Object o) 方法，Comparator接口compare(T o1, To2) 方法

**17、关于Java可能不知道的十件事**

<http://www.importnew.com/25468.html>

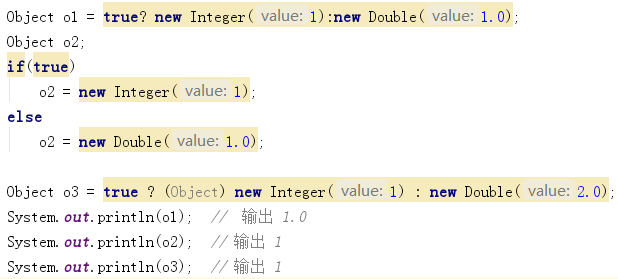
**1）受检异常（checked exception）这件事是不存在的**

JVM 完全不知道这件事，都是Java语言做的。这个不仅会编译，还会抛出 SQLException ，你甚至不需要 Lombok 的 @SneakyThrows 标签。



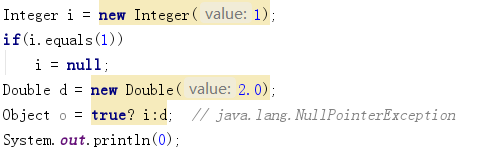
**2）条件表达式**

条件表达式实例分析：



测试输出结果不一致。

在确有必要的情况下，**条件表达式会升级数字类型**，但有些情况下，**自动装箱**会导致结果输出空指针异常，比如：



**3）复合赋值运算符**

等价形式？i 与 j 为不同的数据类型

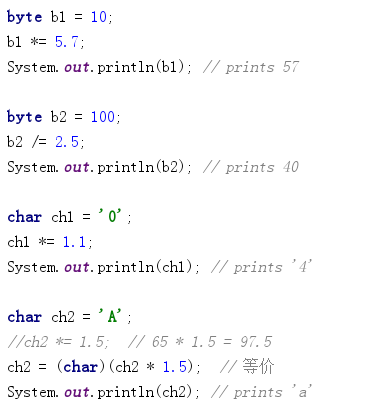
i += j;

i = i + j;

**Java 语言规范（Java Language Standard，JLS）**中这样写道：

符合赋值表达式 E1 op= E2 与 E1 = (T)((E1) op (E2)) 是等价的，这里 T 是 E1 的类型，期望 E1 只被求值一次。

类型转换例子是使用 \*= or /=：



**4）随机数**

<https://blog.jooq.org/2013/10/17/add-some-entropy-to-your-jvm/>

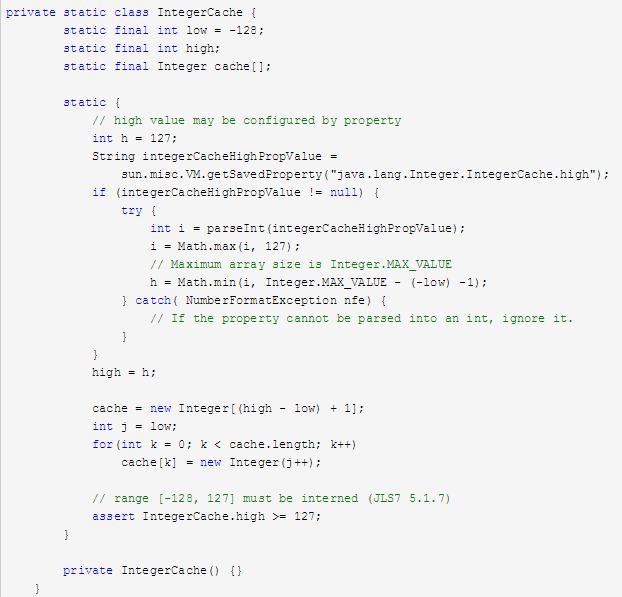
Integer 类有一个私有的嵌套类，名为 IntegerCache ，包含了值从 -128 到 127 的 Integer 对象。当代码需要**从 int 类型封箱成 Integer 对象，而且值在这个范围内时，那么 Java 运行时会使用这个缓存，而不是创建一个新的 Integer 对象**。这主要是处于性能优化的考虑，我们必须牢记在心的是很多 int 值在程序中很多时候都处于这个范围内（例如数组的下标索引）。原来Integer把-128到127（可调）的整数都提前实例化了，原来你不管创建多少个这个范围内的**Integer用ValueOf出来的都是同一个对象**。

这样做的副作用是，很多时候，使用等号操作符来比较两个 Integer 对象时，只要值在范围内都是有效的。这在单元测试中很典型。而在运行模式下，当数值大于 128 时，代码执行会失败。使用反射来访问 IntegerCache 类时会导致一些奇怪的副作用，注意这会影响到整个的 JVM。如果一个 Servlet 重新定义了小的 Integer 缓存值，那么所有运行在同一个 Tomcat 下的其他 Servlet 也遭遇同样问题。

以下**通过反射方式访问 IntegerCache**，然后使用**随机值对缓存进行填充**，即可生成一系列的随机数（反射覆盖JDK的Integer缓存，然后使用自动打包解包（auto-boxing/auto-unboxing）有关）。而这种情况在Java 1.9中已解决。



Integer缓存类IntegerCache：



**再看其它的包装器：**

* Boolean：(全部缓存)
* Byte：(全部缓存)
* Character(<= 127缓存)
* Short(-128 — 127缓存)
* Long(-128 — 127缓存)
* Float(没有缓存)
* ﻿﻿﻿﻿﻿﻿Doulbe(没有缓存)

**18、Java集合底层实现**

**1）HashSet**

<http://www.cnblogs.com/xwdreamer/archive/2012/06/03/2532999.html>

HashSet实现Set接口，由哈希表（实际上是一个HashMap实例）支持。它**不保证set 的迭代顺序；特别是它不保证该顺序恒久不变**。此类允许使用null元素。HashSet中不允许有重复元素，这是因为HashSet是基于HashMap实现的，HashSet中的元素都存放在HashMap的key上面，而value中的值都是统一的一个private static final Object PRESENT = new Object();。HashSet跟HashMap一样，都是一个**存放链表的数组**。

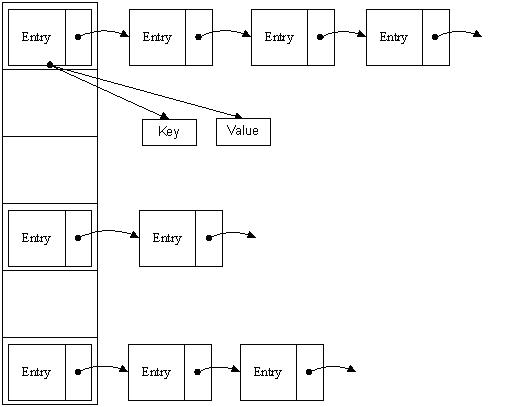
　　HashSet中add方法调用的是底层HashMap中的put()方法，而如果是在HashMap中调用put，首先会判断key是否存在，如果key存在则修改value值，如果key不存在这插入这个key-value。而在set中，因为value值没有用，也就不存在修改value值的说法，因此往HashSet中添加元素，首先判断元素（也就是key）**是否存在**，如果不存在这插入，如果存在着不插入，这样HashSet中就不存在重复值。

**2）HashTable**

<http://blog.csdn.net/zheng0518/article/details/42199477>

<http://www.cnblogs.com/sunshisonghit/p/4692304.html>

底层是基于Entry<?,?>数组实现的，线程安全（synchronized修饰），内部结构：



**3）ArrayList**

<http://blog.csdn.net/angel1hao/article/details/52162115>

ArrayList 底层的实现就是一个数组(固定大小)，当数组长度不够用的时候就会重新开辟一个新的数组，然后将原来的数据拷贝到新的数组内。使用

**newArray = Arrays.copyOf(oldArray, newLen)：**复制全部元素至新数组

**System.arraycopy(oldArray, 旧数组起始位置, newArray, 起始位置, 结束位置)**：复制某一范围的数组元素至新数组中

**扩容策略：**

ArrayList底层是使用数组存储的，当数组大小不足存放新增元素的时候，才会发生扩容。

在add操作中，ArrayList首先会调用ensureCapacityInternal方法进行扩容检测的。

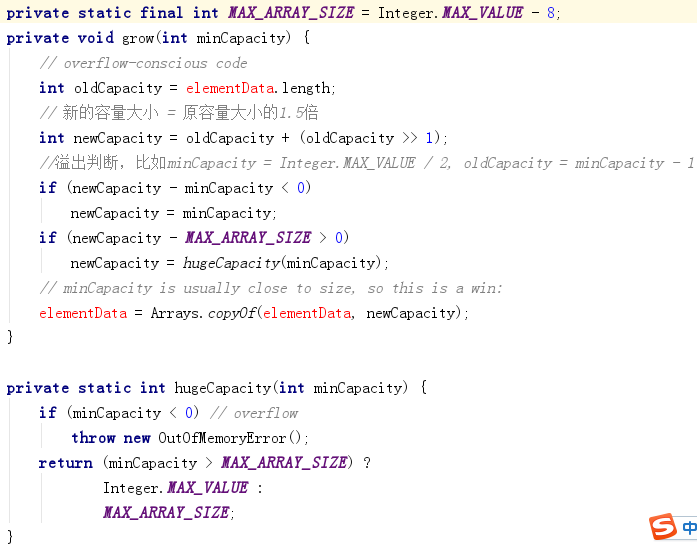
如果数组大小不足，则会自动扩容；如果扩容后的大小超出数组最大的大小，则会抛出异常。

ArrayList扩容方案，主要有两个步骤：**1.大小检测，2.扩容**

* 大小检测：
* 检测数组大小是否为0，如果是，则使用默认的扩容大小10
* 检测是否需要扩容，只有当数组最小需要容量大小大于当前数组大小时，才会进行扩容
* 扩容：grow和hugeCapacity
* 进行数组越界判断
* 拷贝原始数据到新的数组中

扩容代码（容量变为原来的1.5倍）：





**fail-fast机制的实现：**

fail-fast机制也叫作**”快速失败”机制**，是java集合中的一种错误检测机制。在对集合进行**迭代过程**中，除了迭代器可以对集合进行数据结构上进行修改，其他的对集合的数据结构进行修改，都会抛出ConcurrentModificationException错误。（即迭代过程中，除迭代器外，其余操作均不能对集合的结构进行修改）

这里，所谓的进行**数据结构上进行修改**，是指对存储的对象，进行add,set,remove操作，进而对数据发生改变。ArrayList中，有个**modCount的变量**，每次进行add，set，remove等操作，都会执行modCount++。在获取ArrayList的迭代器时，会将ArrayList中的modCount保存在迭代中，每次执行add,set,remove等操作，都会执行一次检查，调用checkForComodification方法，对modCount进行比较。如果**迭代器**中的modCount和List中的modCount不同，则抛出ConcurrentModificationException



**4）LinkedList**

<http://blog.csdn.net/u013124587/article/details/52837848>

<http://blog.csdn.net/fighterandknight/article/details/61476335>

* LinkedList 的底层结构是一个带头/尾指针的双向链表，可以快速的对头/尾节点进行操作。
* 相比数组，链表的特点就是在指定位置插入和删除元素的效率较高，但是查找的效率就不如数组那么高了
* LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。
* LinkedList 实现 **List 接口**，能对它进行队列操作。
* LinkedList 实现 **Deque 接口**，即能将LinkedList当作双端队列使用。
* LinkedList 实现了**Cloneable接口**，即覆盖了函数clone()，能克隆。
* LinkedList 实现**java.io.Serializable接口**，这意味着LinkedList支持序列化，能通过序列化去传输。
* LinkedList 是非同步的。
* LinkedList相对于ArrayList来说，是可以快速添加，删除元素，ArrayList添加删除元素的话需移动数组元素，可能还需要考虑到扩容数组长度

**5）Vector**

<http://blog.csdn.net/panweiwei1994/article/details/76972890>

扩容为原来的2倍

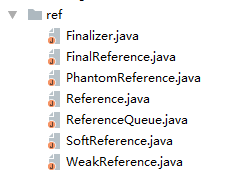
**19、Java 4种引用类型**

<http://www.cnblogs.com/mjorcen/p/3968018.html>

<http://www.importnew.com/20468.html>

<http://blog.csdn.net/u013256816/article/details/50764532>

Java中提供了4个级别的引用：强应用、软引用、弱引用和虚引用。这四个引用定义在java.lang.ref的包下：

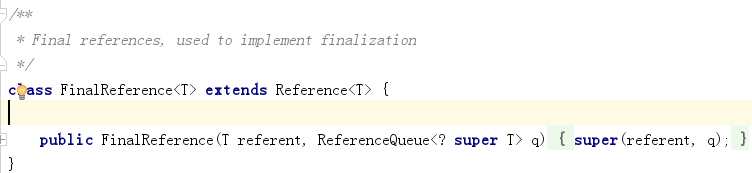


**1）强引用（Final Reference）**

指在程序代码中普遍存在的，类似Object obj = new Object()这类的引用，只要强引用还存在，垃圾收集器**永远不会回收掉被引用的对象**。

具备以下三个特点：

* 强引用可以直接访问目标对象；
* **强引用锁**指向的对象在任何时候都不会被系统回收。JVM宁愿抛出OOM异常也不回收强引用所指向的对象；
* 强应用可能导致**内存泄露**；



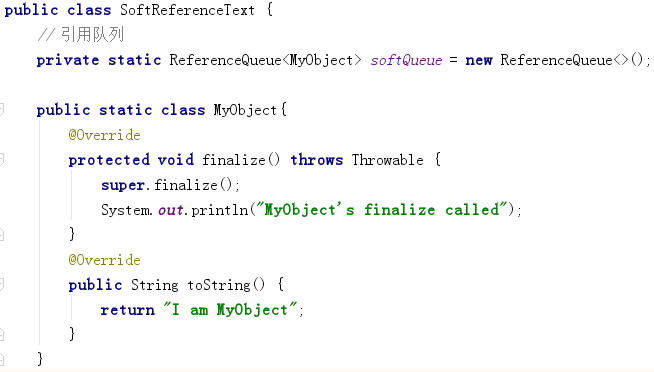
从类定义中可以看出，只有一个构造函数，根据所给的对象的应用和应用队列构造一个强引用。

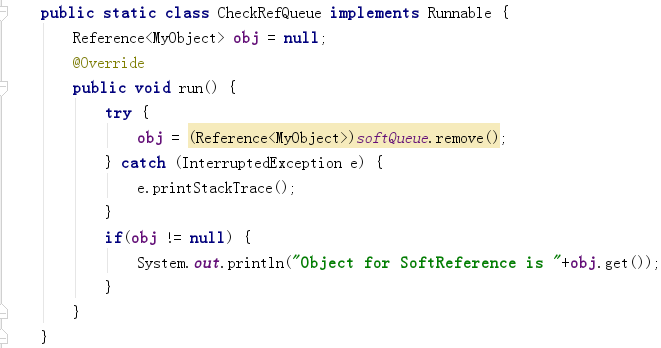
**2）软引用（Soft Reference）**

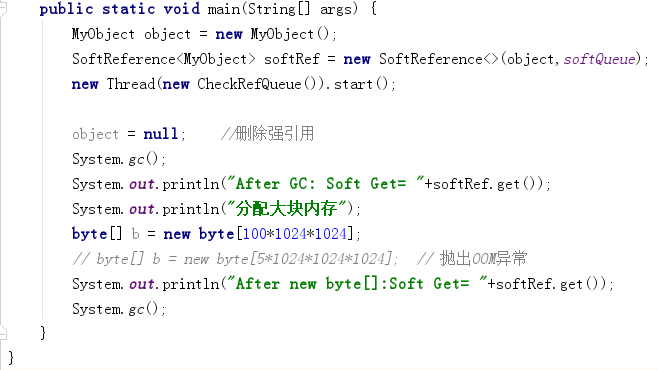
用来描述一些还有用但并非必须的对象。对于软引用关联着的对象，在系统将要发生**内存溢出**异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。

对于软引用关联着的对象，如果内存充足，则垃圾回收器不会回收该对象，如果内存不够了，就会回收这些对象的内存。在 JDK 1.2 之后，提供了 SoftReference 类来实现软引用。软引用可用来实现内存敏感的高速缓存。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的**对象**被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

案例分析：







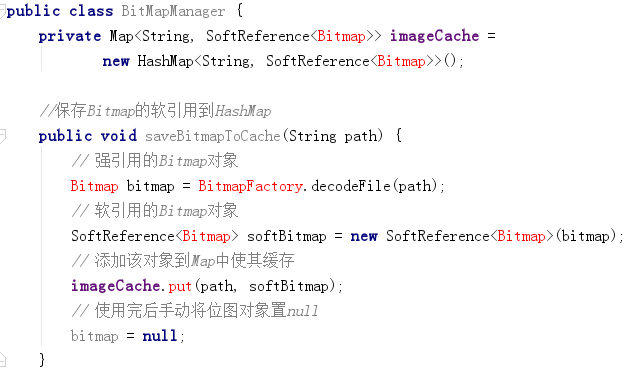
这个案例中，首先构造MyObject对象，并将其赋值给object变量，构成**强引用**。然后使用SoftReference构造这个MyObject对象的软引用softRef，并注册到softQueue引用队列。当softRef被回收时，会被加入softQueue队列。设置obj=null，删除这个强引用，因此，系统内对MyObject对象的引用只剩下软引用。此时，显示调用GC，通过软引用的get()方法，取得MyObject对象的**引用**，发现对象并未被回收，这说明GC在内存充足的情况下，不会回收软引用对象。

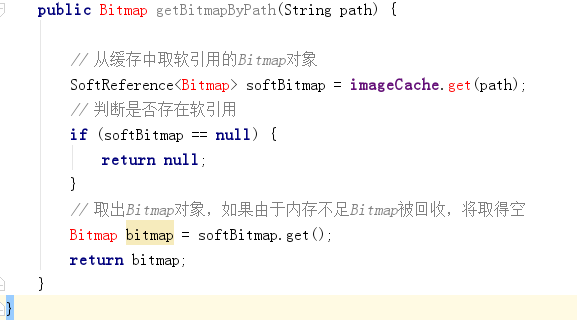
接着，请求一块大的堆空间5\*1024\*1024，这个操作会使系统堆内存使用紧张，从而产生新一轮的GC。在这次**GC后**，softRef.get()不再返回MyObject对象，而是返回null，说明在系统内存紧张的情况下，软引用被回收。软引用被回收时，会被加入注册的**引用队列**。

如果将上面案例中的数组再改大点，比如100\*1024\*1024，就会抛出OOM异常

软引用主要应用于**内存敏感的高速缓存**，在android系统中经常使用到。一般情况下，Android应用会用到大量的默认图片，这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。但是，由于图片占用内存空间比较大，**缓存**很多图片需要很多的内存，就可能比较容易发生**OutOfMemory异常**。这时，我们可以考虑使用软引用技术来避免这个问题发生。**SoftReference可以解决oom的问题**，每一个对象通过软引用进行实例化，这个对象就以cache的形式保存起来，当再次调用这个对象时，那么直接通过软引用中的get（）方法，就可以得到对象中中的资源数据，这样就没必要再次进行读取了，直接从cache中就可以读取得到，当内存将要发生OOM的时候，GC会迅速把所有的软引用清除，防止oom发生。

案例使用2：

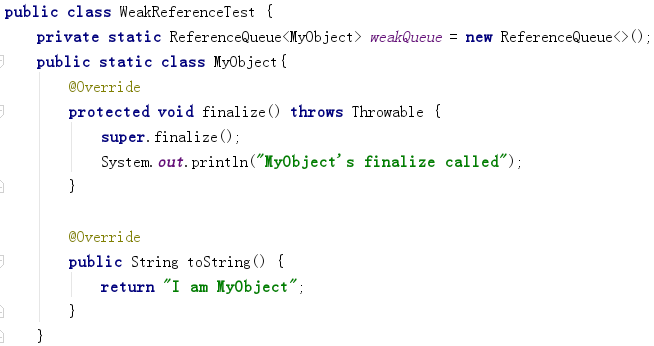


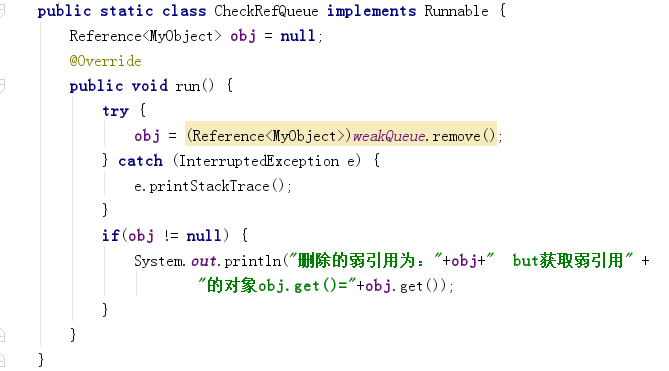


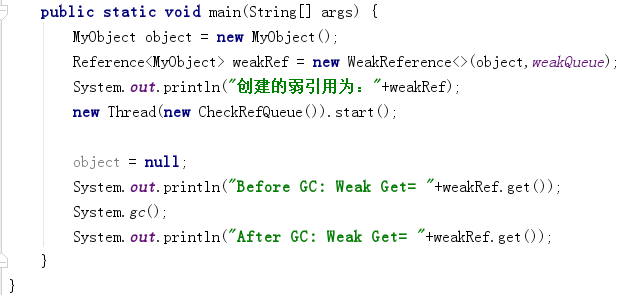
**3）弱引用（Weak Reference）**

用来描述非必须的对象，但是它的**强度比软引用更弱一些**，被弱引用关联的对象只能生存到下一次垃圾收集器工作之前。当**垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象**。一旦一个弱引用对象被垃圾回收器回收，便会加入到一个注册引用队列中。

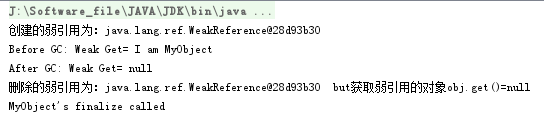
案例：







运行结果：



可知在GC之前，**弱引用对象并未被垃圾回收器发现**，因此通过 weakRef.get()可以获取对应的对象引用。但是只要进行垃圾回收，弱引用一旦被发现，便会立即被回收，并加入注册引用队列中。此时再试图通过weakRef.get()获取对象的引用就会失败。**弱引用的相关实际案例可以参考WeakHashMap**，博主会在近期整理出相关文档。

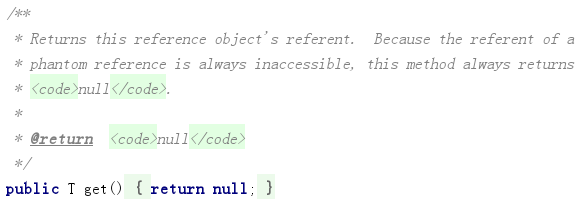
弱引用最常见的用处是在**集合类**中，尤其在哈希表中。哈希表的接口允许使用任何**Java对象作为键**来使用。当一个键值对被放入到哈希表中之后，哈希表对象本身就有了对这些键和值对象的引用。如果这种引用是强引用的话，那么只要哈希表对象本身还存活，其中所包含的键和值对象是不会被回收的。如果某个**存活时间很长的哈希表中包含的键值对很多，最终就有可能消耗掉JVM中全部的内存**。对于这种情况的解决办法就是使用弱引用来引用这些对象，这样哈希表中的键和值对象都能被垃圾回收。

*软引用、弱引用都非常适合来****保存那些可有可无的缓存数据****。如果这么做，当系****统内存不足时，这些缓存数据会被回收，不会导致内存溢出****。而当内存资源充足时，这些缓存数据又可以存在相当长的时间，从而起来加速系统的作用。*

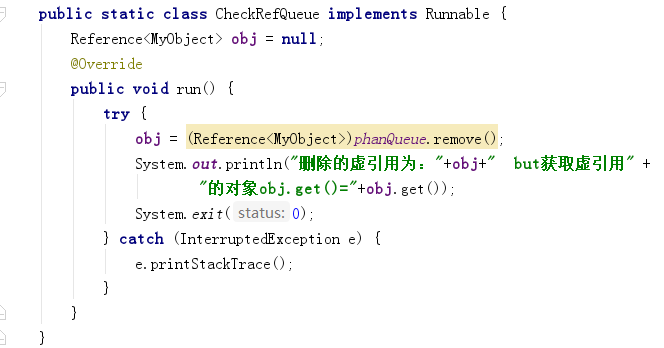
**4）虚引用（Phantom Reference）**

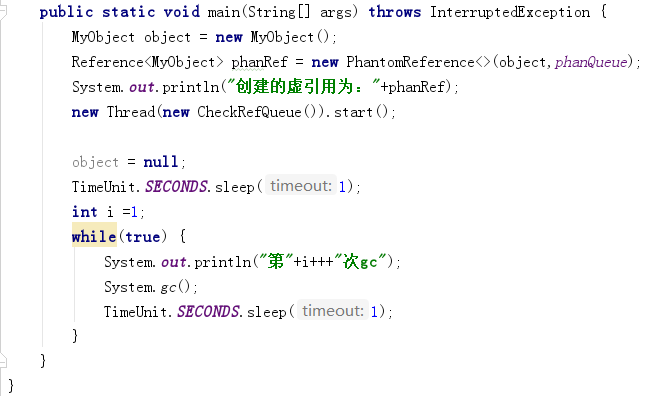
虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个持有虚引用的对象，和没有引用几乎是一样的，随时都有可能被垃圾回收器回收。当试图通过虚引用的get()方法取得强引用时，总是会**失败**。并且，**虚引用必须和引用队列一起使用，它的作用在于跟踪垃圾回收过程**。

虚引用中get方法的实现如下：

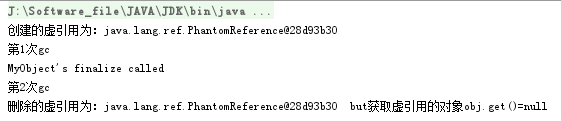


案例：





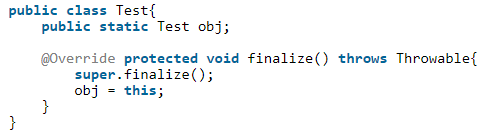
运行结果：



可知，再经过一次GC之后，系统找到了垃圾对象，并调用finalize()方法回收内存，但没有立即加入回收队列。第二次GC时，该对象真正被GC清楚，此时，加入虚引用队列。

虚引用的最大作用在于跟踪对象回收，清理被销毁对象的相关资源。JVM对某一个对象至多只执行一次被重写的finalize方法。

通常当对象不被使用时，重载该对象的类的finalize方法可以回收对象的资源。但是如果使用不慎，会使得对象复活，譬如这么编写finalize方法：



在Object类里面有个**finalize方法**，其设计的初衷是在一个对象被真正回收之前，可以用来执行一些清理的工作。因为Java并没有提供类似C++的析构函数一样的机制，就通过 finalize方法来实现。但是问题在于垃圾回收器的运行时间是不固定的，所以这些清理工作的实际运行时间也是不能预知的。幽灵引用（phantom reference）可以解决这个问题。**在创建幽灵引用PhantomReference的时候必须要指定一个引用队列**。当一个对象的finalize方法已经被调用了之后，这个对象的幽灵引用会被加入到队列中。通过检查该队列里面的内容就知道一个对象是不是已经准备要被回收了。

**20、Java 内存泄露的原因及可能出现的情况**

<http://blog.csdn.net/seelye/article/details/8269705>

<http://www.cnblogs.com/yl-saber/p/6907517.html>

**21、try catch finally的使用**

<https://www.oschina.net/question/3665510_2266610>

* finally 里 始终会被执行到。除System.exit(0)外，直接退出程序。
* 即使try中有return ，也是先执行 return 后面的语句完了之后，不立马return，而是去执行finally中的语句。
* 当try中与finally里，**同时出现**return , 则只会返回 finally 中的return 结果。
* finally中的值不能影响try中 即将返回的结果值。

注意： 若finally中没有return在try或catch中有return，那么在执行return跟着语句之后，会把语句的结果新开辟一内存空间，直接把结果的存放此内存空间中。所以，finally中的值不能影响try或catch中即将return的结果。

**22、抽象类遵循的原则及与接口的区别**

<https://www.oschina.net/question/3665510_2266610>

遵循的规则：

* abstract关键字只能修饰类和方法，不能修饰字段。
* 抽象类不能被实例化（无法使用new关键字创建对象实例），只能被继承。
* 抽象类可以包含属性，方法，构造方法，初始化块，内部类，枚举类，和普通类一样，普通方法一定要实现，变量可以初始化、不初始化但不能初始化后在抽象类中重新赋值或操作该变量（只能在子类中改变该变量）。
* 抽象类中的抽象方法（加了abstract关键字的方法）不能实现。
* 含有抽象方法的类必须定义成抽象类。

抽象类和接口的区别：

* 接口是公开的，里面不能有私有的方法或变量，是用于让别人使用的，而抽象类是可以有私有方法或私有变量的。
* abstractclass 在Java 语言中表示的是一种继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个interface，实现多重继承。接口还有标识（里面没有任何方法，如Remote接口）和数据共享（里面的变量全是常量）的作用。
* 在abstract class 中可以有自己的数据成员，也可以有非abstarct的成员方法，而在interface中，只能够有静态的不能被修改的数据成员（也就是必须是**static final**的，不过在interface中一般不定义数据成员），所有的成员方法默认都是 public abstract 类型的。
* Abstract class和interface所反映出的设计理念不同。其实abstractclass表示的是"is-a"关系，interface表示的是"has-a"关系。
* 实现接口的一定要实现接口里定义的所有方法，而实现抽象类可以有选择地重写需要用到的方法，一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现。抽象类中可以有非抽象方法。接口中则不能有实现方法。
* 接口中定义的变量默认是public static final 型，且必须给其初值，所以实现类中不能重新定义，也不能改变其值。抽象类中的变量默认是friendly 型，其值可以在子类中重新定义，也可以在子类中重新赋值

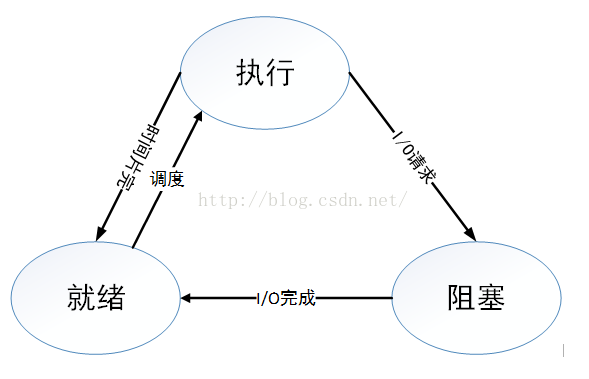
**23、进程状态**

<http://www.cnblogs.com/zxf98/p/5716296.html>

一般来说，进程有三个状态：运行、就绪、阻塞状态。

* 运行状态：进程占用CPU，并在CPU上运行；
* 就绪状态：进程已经具备运行条件，但是CPU还没有分配过来；
* 阻塞状态：进程因等待某件事发生而暂时不能运行；

进程状态转换图：



理论上上述三种状态之间转换分为六种情况；

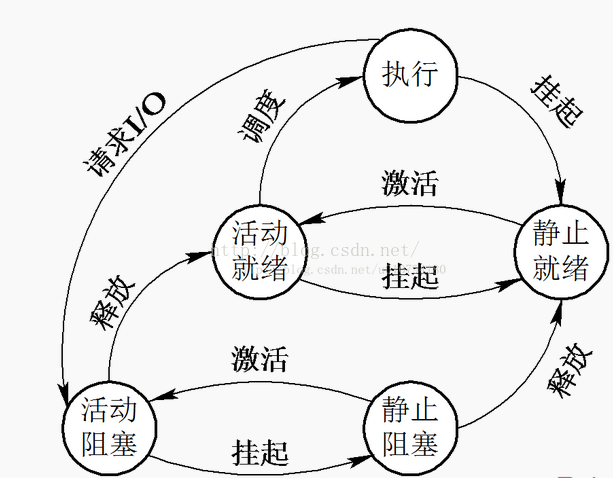
* **运行——>就绪：**1，主要是进程占用CPU的时间过长，而系统分配给该进程占用CPU的时间是有限的；2，在采用抢先式优先级调度算法的系统中,当有更高优先级的进程要运行时，该进程就被迫让出CPU，该进程便由执行状态转变为就绪状态。
* **就绪——>运行：**运行的进程的时间片用完，调度就转到就绪队列中选择合适的进程分配CPU
* **运行——>阻塞：**正在执行的进程因发生某等待事件而无法执行，则进程由执行状态变为阻塞状态，如发生了I/O请求
* **阻塞——>就绪：**进程所等待的事件已经发生，就进入就绪队列

以下两种状态是**不可能发生的：**

* **阻塞——>运行：**即使给阻塞进程分配CPU，也无法执行，操作系统在进行调度时不会从阻塞队列进行挑选，而是从就绪队列中选取
* **就绪——>阻塞：**就绪态根本就没有执行，谈不上进入阻塞态。

在一些系统中，又增加了一些**新状态**，如挂起状态，可运行状态，深度睡眠状态，浅度睡眠状态，暂停状态，僵死状态。

* **可运行状态：**运行状态和就绪状态的合并，表示进程正在运行或准备运行，Linux 中使用TASK\_RUNNING 宏表示可运行状态。
* **浅度睡眠状态：**进程正在睡眠（被阻塞），等待资源的到来是唤醒，也可以通过其他进程信号或时钟中断唤醒，进入运行队列。Linux 中使用TASK\_INTERRUPTIBLE 宏表示此状态。
* **深度睡眠状态：其**和浅度睡眠基本类似，但不可被其他进程信号或时钟中断唤醒。Linux 中使用TASK\_UNINTERRUPTIBLE 宏表示此状态。
* **暂停状态：**进程暂停执行接受某种处理。Linux 使用TASK\_STOPPED 宏表示此状态。
* **僵死状态：**进程已经结束但未释放进程控制块（PCB），Linux 使用TASK\_ZOMBIE 宏表示此状态。
* **挂起状态：**在执行状态的进程通过挂起即可进入就绪状态，如图所示，就绪状态和阻塞状态都分为活动态和静止态。由活动态向静止态转换就是通过挂起实现的。



引入挂起状态的原因有：

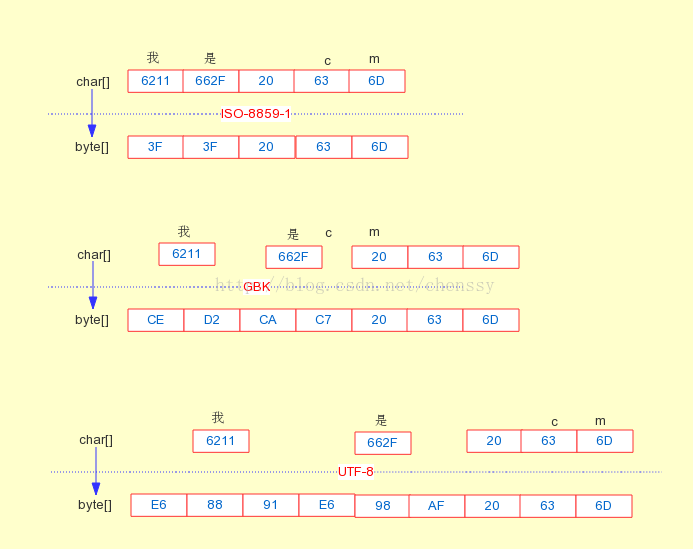
* (1) **终端用户的请求**。当终端用户在自己的程序运行期间发现有可疑问题时，希望暂时使自己的程序静止下来。亦即，使正在执行的进程暂停执行；若此时用户进程正处于就绪状态而未执行，则该进程暂不接受调度，以便用户研究其执行情况或对程序进行修改。我们把这种静止状态称为挂起状态。
* (2) **父进程请求**。有时父进程希望挂起自己的某个子进程，以便考查和修改该子进程，或者协调各子进程间的活动。
* (3) **负荷调节的需要**。当实时系统中的工作负荷较重，已可能影响到对实时任务的控制时，可由系统把一些不重要的进程挂起，以保证系统能正常运行。
* (4) **操作系统的需要**。操作系统有时希望挂起某些进程，以便检查运行中的资源使用情况或进行记账。

**24、Java中文乱码问题**

<http://blog.csdn.net/chenssy/article/details/42913511/>

在java中主要有四个场景需要进行编码解码操作：I/O操作、内存、数据库、javaWeb

乱码问题无非就是转码过程中编码格式的不统一产生的，比如编码时采用UTF-8，解码采用GBK，但**最根本的原因**是字符到字节或者字节到字符的转换出问题了，而这中情况的转换最主要的场景就是I/O操作的时候。当然I/O操作主要包括网络I/O（也就是javaWeb）和磁盘I/O。



**四、数据库相关知识**

<http://blog.jobbole.com/24006/>

**1、加载驱动程序方法**

* **Class.forName**("com.microsoft.sqlserver.jdbc.SQLServerDriver")：调用方法 Class.forName
* **DriverManager.registerDriver**(new com.mysql.jdbc.Driver())：通过registerDriver方法注册
* **System.setProperty**("jdbc.drivers"， "com.mysql.jdbc.Driver")：通过添加系统的jdbc.drivers属性

**2、什么是数据库事务？**

事务（Transaction、是***并发控制的基本单位***。所谓的事务，它是一个**操作序列**，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。例如，银行转账工作：从一个账号扣款并使另一个账号增款，这两个操作要么都执行，要么都不执行。所以，应该把它们看成一个事务。***事务是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性***。针对上面的描述可以看出，*事务的提出主要是为了解决并发情况下保持数据一致性的问题*。

事务的标准定义：指***作为单个逻辑工作单元执行的一系列操作***，而这些逻辑工作单元需要具有原子性、一致性、隔离性和持久性四个属性，统称为ACID特性。

**所谓事务**是用户定义的一个数据库操作序列，**这些操作要么全做要么全不做，是一个不可分割的工作单位**。例如，在关系数据库中，一个事务可以是一条SQL语句、一组SQL语句或整个程序。 事务和程序是两个概念。一般地讲，一个程序中包含多个事务。事务的开始与结束可以由用户显式控制。如果用户没有显式地定义事务，则由DBMS按缺省规定自动划分事 务。

**总结：**事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位，是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性。

**3、事务四大特性ACID**

如果一个数据库声称**支持事务的操作**，那么该数据库必须要具备以下四个特性：

* **原子性（Atomicity、**：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。
* **一致性（Consistency、：**一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说***一个事务执行之前和执行之后都必须处于一致性状态***。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

* **隔离性（Isolation、：**隔离性是当**多个用户并发访问数据库时**，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，**多个并发事务之间要相互隔离**。

即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

* **持久性（Durability、：**持久性是指**一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作**。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

总结：

* **原子性：**是指事务中包含的所有操作要么全部成功，要么全部失败回滚。
* **一致性**：事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。**比如：**A和B间相互转账，但转账前后两者金额的总数不变
* **隔离性：**当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。
* **持久性：**一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作

**4、数据库事务四大隔离级别**

如果不考虑事务的隔离性，会产生以下问题：

* **丢失更新：**当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，会发生丢失更新问题。每个事务都不知道其它事务的存在。最后的更新将重写由其它事务所做的更新，这将导致数据丢失。
* **脏读：**指在一个事务处理过程里读取了另一个未提交的事务中的数据，造成事务间得到的数据不一致
* **不可重复读：**指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。
* 例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了**不可重复读**。
* 不可重复读和脏读的**区别**是，*脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据*。
* **幻读（或虚读、：**幻读是**事务非独立执行时发生的一种现象**。
* 例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现**还有一行没有修改**，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了**幻读。**
* 幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同、，所不同的是不可重复读查询的都是**同一个数据项**，而幻读针对的是**一批数据整体**（比如数据的个数、

提供的**四种隔离级别（从大到小、**：

① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

③ Read committed (读已提交)：可避免脏读的发生。

④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证



在MySQL数据库中，支持上面四种隔离级别，默认的为**Repeatable read (可重复读)**；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为**Read committed级别**。

**5、什么是索引？索引的常见类型有什么？**

<http://blog.csdn.net/sinat_30186009/article/details/52169057>

<http://www.cnblogs.com/gavinsp/p/5513536.html> 数据库索引详解

索引就是加快检索表中数据的方法。数据库的索引类似于书籍的索引。在书籍中，索引允许用户不必翻阅完整个书就能迅速地找到所需要的信息。在数据库中，索引也允许数据库程序迅速地找到表中的数据，而不必扫描整个数据库。

索引一般可分为：聚集索引和非聚集索引

**1）聚集索引**

聚集索引根据数据行的键值在表或视图中排序和存储这些数据行。索引定义中包含聚集索引列。每个表只能有一个聚集索引，因为数据行本身只能按一个顺序排序。只有当表包含聚集索引时，表中的数据行才按排序顺序存储。如果表具有聚集索引，则该表称为**聚集表**。如果表没有聚集索引，则其数据行存储在一个称为堆的无序结构中。

　　通俗点说，**聚集索引的页存储的是实际数据**。每个表只能建立唯一的聚集索引，但也可以没有。如果建立聚集索引，那么表中数据以**B树形式存储数据**。

　　对于聚集索引的理解，打个比方，即英文字典的单词编排。英文字典单词以A，B，C，D….X，Y，Z的形式顺序编排，如果我们查找 Good 单词，我们首先定位到G，然后定位o-o-d. 最终查找到Good，便是good实际存在的地方。建聚集索引需要至少相当该表120%的附加空间，以**存放该表的副本和索引中间页**。

**聚集索引确定表中数据的物理顺序**。聚集索引类似于电话簿，后者按姓氏排列数据。由于聚集索引规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引、，就像电话簿按姓氏和名字进行组织一样。

聚集索引对于那些经常要**搜索范围值的列**特别有效。使用聚集索引找到包含第一个值的行后，便可以确保包含后续索引值的行在物理相邻。例如，如果应用程序执行的一个查询经常检索**某一日期范围内的记录**，则使用聚集索引可以迅速找到包含开始日期的行，然后检索表中所有相邻的行，直到到达结束日期。这样有助于提高此类查询的性能。同样，如果对从表中检索的数据进行排序时经常要用到某一列，则可以将该表在**该列上聚集（物理排序、**，避免每次查询该列时都进行排序，从而节省成本。

当索引值唯一时，使用聚集索引查找特定的行也很有效率。例如，使用唯一雇员 ID 列 emp\_id 查找特定雇员的最快速的方法，是在 emp\_id 列上创建聚集索引或 PRIMARY KEY 约束。

**2）非聚集索引**

非聚集索引具有独立于数据行的结构。非聚集索引包含**非聚集索引键值，并且每个键值项都有指向包含该键值的数据行的指针**。从非聚集索引中的索引行指向数据行的指针称为**行定位器**。行定位器的结构取决于数据页是存储在堆中还是聚集表中。对于堆，行定位器是指向行的指针。对于聚集表，行定位器是聚集索引键。

　　通俗点说，**非聚集索引的页存储的是不是实际数据，而是实际数据的地址**。一个表可以存在多个非聚集索引。在Sql Server2005中，每个表最多可以建立249个，而在Sql server2008中，则最多可以建立999个非聚集索引。

　　对于非聚集索引的理解，即新华字典的“偏旁部首”查字法。遇到您不认识的字，不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而需要去根据**“偏旁部首”**查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。

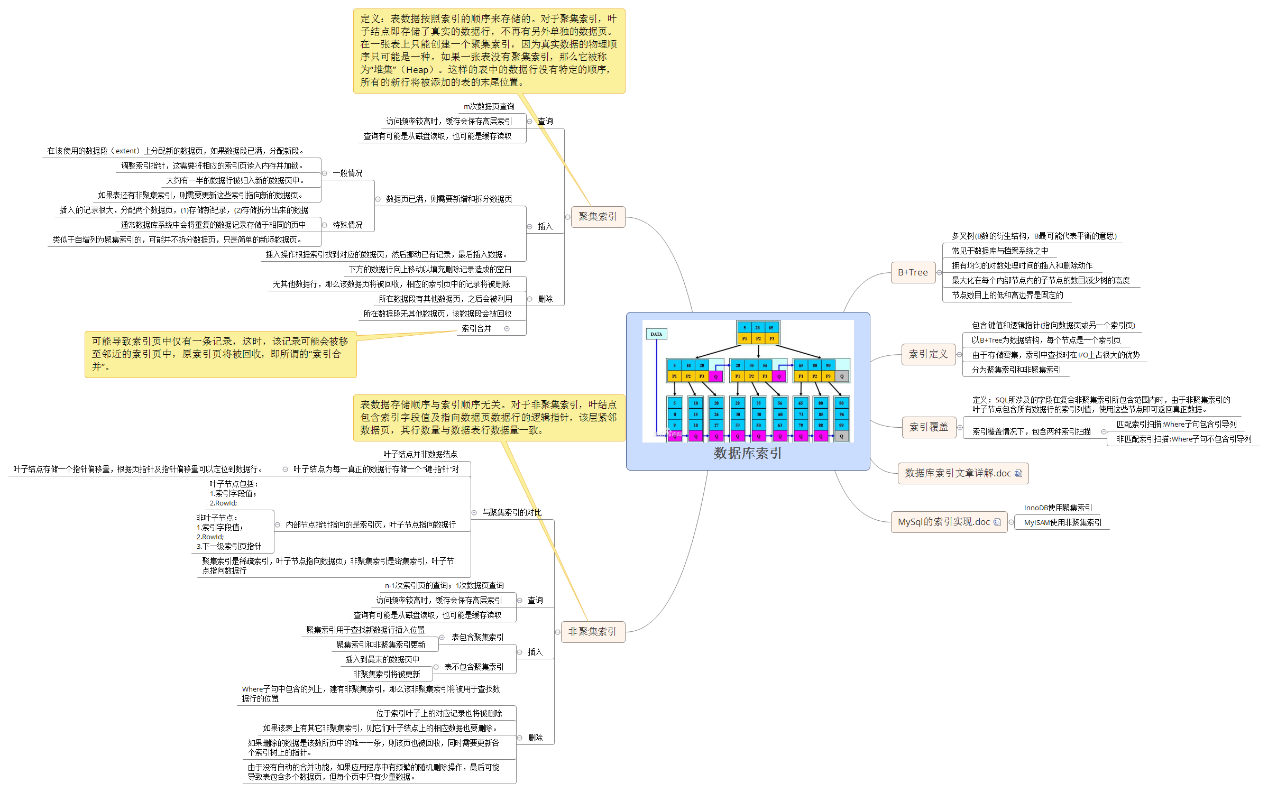
　　但您结合**“部首目录”和“检字表”**而查到的字的排序并不是真正的正文的排序方法，比如您查“张”字，我们可以看到在查部首之后的检字表中“张”的页码是672页，检字表中“张”的上面是“驰”字，但页码却是63页，“张”的下面是“弩”字，页面是390页。很显然，这些字并不是真正的分别位于“张”字的上下方，现在您看到的连续的“驰、张、弩”三字实际上就是他们在非聚集索引中的排序，是**字典正文中的字在非聚集索引中的映射**。我们可以通过这种方式来找到您所需要的字，**但它需要两个过程，先找到目录中的结果，然后再翻到您所需要的页码**。我们把这种目录纯粹是目录，正文纯粹是正文的排序方式称为“非聚集索引”。

**总结：**

* **聚集索引：**对正文内容按照一定规则排列的目录称为聚集索引
* **非聚集索引：**目录自己按照一定规则排列，正文自己按照另一种规则排列，目录主要是保存对正文的一个映射关系，这种称为非聚集索引

**两者区别：**

* 聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个
* 聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续



**3）覆盖索引**

覆盖索引是指**那些索引项中包含查寻所需要的全部信息的非聚集索引**，这种索引之所以比较快也正是因为索引页中**包含了查寻所必须的数据，不需去访问数据页**。如果非聚簇索引中包含结果数据，那么它的查询速度将快于聚集索引。

　　但是由于覆盖索引的索引项比较多，要占用比较大的空间。而且update 操作会引起索引值改变。所以如果潜在的覆盖查询并不常用或不太关键，则覆盖索引的增加反而会降低性能。

**4）主键和索引**

<http://kb.cnblogs.com/page/78142/>

**主键：**表通常具有包含唯一标识表中每一行的值的一列或一组列。这样的一列或多列称为表的主键 (PK)，**用于强制表的实体完整性**。在创建或修改表时，您可以通过定义 PRIMARY KEY 约束来创建主键。它是一种唯一索引。

|  |  |  |
| --- | --- | --- |
|  | 主键 | 聚集索引 |
| 用途 | 强制表的实体完整性 | 对数据行的排序，方便查询用 |
| 一个表多少个 | 一个表最多一个主键 | 一个表最多一个聚集索引 |
| 是否允许多个字段来定义 | 一个主键可以多个字段来定义 | 一个索引可以多个字段来定义 |
| 是否允许 null 数据行出现 | 如果要创建的数据列中数据存在null，无法建立主键。创建表时指定的 PRIMARY KEY 约束列隐式转换为 NOT NULL | **没有限制建立聚集索引的列一定必须 not null**。也就是列的数据可以是 null。参看最后一项比较 |
| 是否要求数据必须唯一 | 要求**数据必须唯一** | 数据即可以唯一，也可以不唯一。看你定义这个索引的 UNIQUE 设置。（这一点需要看后面的一个比较，虽然你数据列可能不唯一，但是系统会替你产生一个你看不到的唯一列、 |
| 创建的逻辑 | 数据库在创建主键同时，会**自动建立一个唯一索引**。如果这个表之前没有聚集索引，同时建立主键时候没有强制指定使用非聚集索引，则建立主键时候，同时建立一个唯一的聚集索引。 | **如果未使用 UNIQUE 属性创建聚集索引，数据库引擎 将向表自动添加一个四字节 uniqueifier 列**。必要时，数据库引擎 将向行自动添加一个 uniqueifier 值，使每个**键唯一**。此列和列值供内部使用，用户不能查看或访问。 |

**6、在数据库中通过什么描述聚集索引与非聚集索引的？**

<http://jingyan.baidu.com/article/5d6edee2f269fe99eadeece9.html>

索引是通过B+树的形式进行描述的，我们可以这样**区分聚集与非聚集索引的区别：**聚集索引的叶节点就是最终的数据节点，而非聚集索引的叶节仍然是索引节点，但它有一个指向最终数据的指针。

**7、索引的存储结构**

<http://kb.cnblogs.com/page/78142/>

整表扫描和索引扫描是Sql Server**数据库检索到数据的唯一的两种方式**。除此之外，没有第三种方式供Sql Server检索到数据。

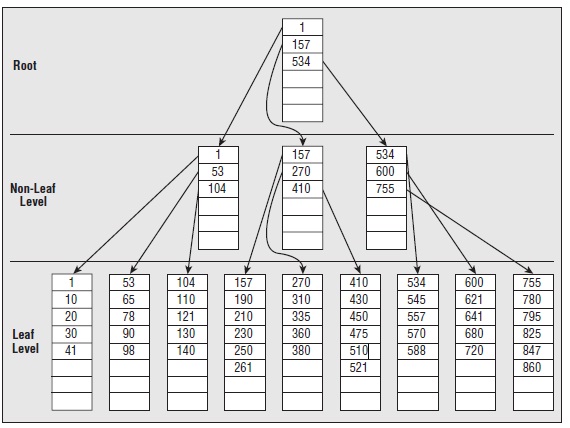
**1）整表扫描**

最直接的检索方式， Sql Server进行表扫描时，会从表头开始扫描，直到整个表结束。 当找到符合条件的记录，便把该记录存在结果集中。对于小数据量的表，这是一种很快捷的方式。如果没有为表创建索引，那么Sql server便按这种方式检索数据。

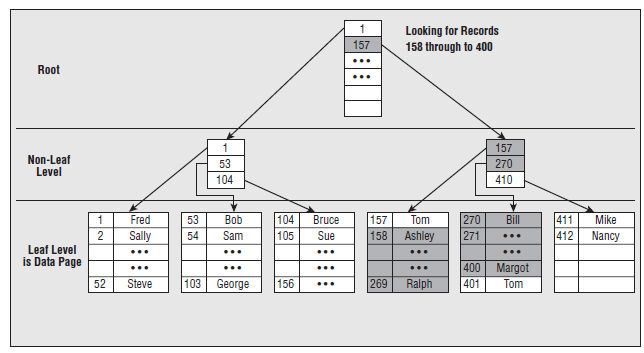
**2）索引扫描**

如果为表创建了索引，在进行检索前，Sql Server优化器会**根据查询条件，从可用的索引中选择最优化的索引**。检索时，便会遍历B树，当找到符合条件的记录，便把该记录存在结果集中。因此，检索大数据量的表，使用索引相对于整表扫描会显著地提高性能

* B-Tree



* 聚集索引：叶子节点存放的是**实际的数据**。索引的入口点存放在master->sys.indexes中



* 非聚集索引：与聚集索引很类似。不同处在：**叶子节点存放的不是实际数据，而是指向实际数据的指针**。检索速度非常接近于聚集索引，比起聚集索引，实际上只是多一步由根据指针检索到实际数据的过程



**7、使用索引应注意什么？**

* 聚集索引通常速度优于非聚集索引
* 建索引时应考虑是否有足够的空间。索引占据空间，平均约1.2倍数据库本身大小。
* 在经常用于**查询或聚合条件的字段上建立聚集索引**。这类查询条件包括 between， >， <，group by， max，min， count等。
* **不要在经常作为插入，且插入字段无序的列上建立聚集索引**。 插入数据行会涉及分页，rebuild索引会消耗大量时间。参考文末"一个不恰当使用聚集索引的例子"。
* **值高度的唯一性字段上建立索引**。不能在诸如性别的字段上建立索引。
* 只有作为索引的**第一个列**包含在查询条件中，该索引才的作用。

打个比方，我们用偏旁+部首来查汉字，那么偏旁首先必须包括在查询条件中，只有先定位偏旁，再结合部首，才能发挥偏旁+部首来检索的快速功效。

* 删除一直不用的索引。特别是对于删除和修改比较频繁的数据表，必须考虑如何精华索引。

**8、数据库的数据存储**

<http://kb.cnblogs.com/page/78142/>

* **文件：**我们一旦创建一个数据库，都会生成两个文件：
* DataBaseName.mdf: 主文件，这是数据库中的数据最终存放的地方。
* DataBaseName.ldf:日志文件，由数据操作产生的一系列日志记录。
* **分区：**在一个给定的文件中，**为表和索引分配空间的基本存储单位**。1个区占64KB，由**8个连续的页组成**。如果一个分区已满，但需存一条新的记录，那么该记录将占用整个新分区的空间。
* **页：**分区中的一个分配单位。这是实际数据行最终存放的地方，页用于存储数据行。

Sql Server有多种类型的页：Data、Index、BLOB、GAM(Global Allocation Map)、SGAM、PFS(Page Free Space)、IAM(Index Allocation Map)、BCM(Bulk Changed Map)等。

**9、mysql 为什么索引一部分放到内存，一部分放到硬盘上？**

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**，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣**最重要的指标**就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要**尽量减少查找过程中磁盘I/O的存取次数**。下面先介绍内存和磁盘存取原理，然后再结合这些原理分析B-/+Tree作为索引的效率。

**10、索引在什么情况下失效**

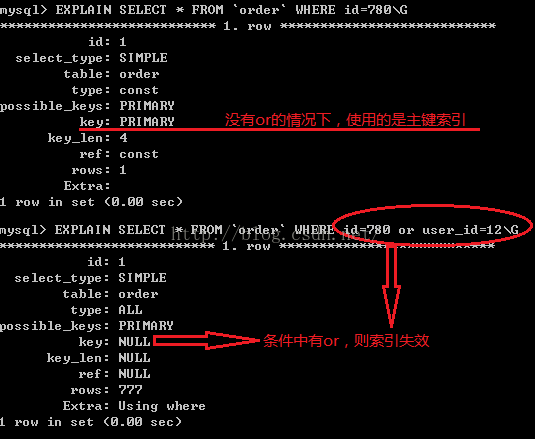
<http://www.cnblogs.com/hongfei/archive/2012/10/20/2732589.html>

<http://blog.csdn.net/guoxingege/article/details/51034387>

<http://www.cnblogs.com/binyue/p/4058931.html>

索引并不是时时都会生效的，比如以下几种情况，将导致索引失效：

* 如果条件中有or，即使其中有条件带索引也不会使用（这也是为什么尽量少用or的原因）。**注意：**要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引

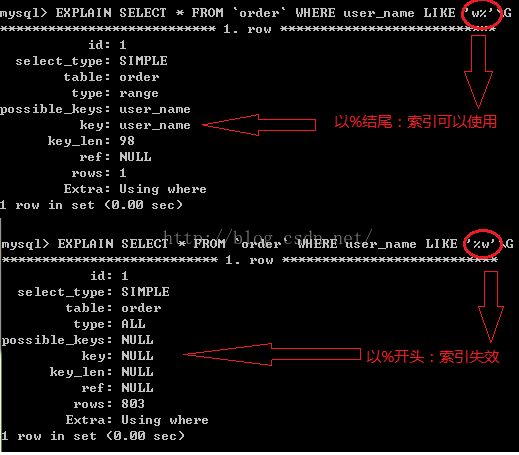


* **对于多列索引，不是使用的第一部分(第一个)，则不会使用索引**
* **索引不存储null值**

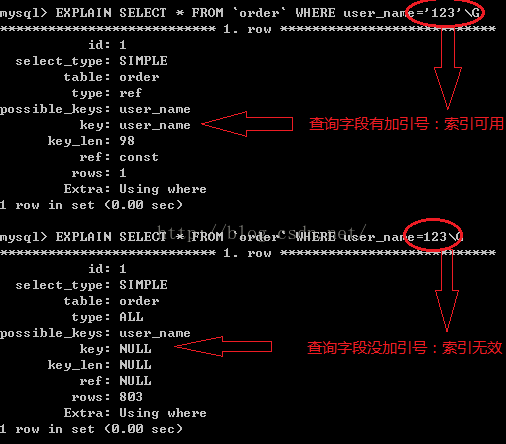
更准确的说，单列索引不存储null值，复合索引不存储全为null的值。索引不能存储Null，所以对这列采用is null条件时，因为索引上根本没Null值，不能利用到索引，只能**全表扫描**。为什么索引列不能存Null值？将索引列值进行建树，其中必然涉及到诸多的比较操作。Null值的特殊性就在于参与的运算大多取值为null。这样的话，null值实际上是不能参与进建索引的过程。也就是说，null值不会像其他取值一样出现在索引树的叶子节点上。

* like查询是以%开头；前导模糊查询不能利用索引(like '%XX'或者like '%XX%')

假如有这样一列code的值为'AAA','AAB','BAA','BAB' ,如果where code like '%AB'条件，由于前面是**模糊的**，所以不能利用索引的顺序，必须一个个去找，看是否满足条件。这样会导致**全索引扫描或者全表扫描**。如果是这样的条件where code like 'A % '，就可以查找CODE中A开头的CODE的位置，当碰到B开头的数据时，就可以停止查找了，因为后面的数据一定不满足要求。这样就可以利用索引了。



* 如果**列类型是字符串**，那一定要在条件中将数据使用引号引用起来,否则不使用索引



* 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

此外，**查看索引的使用情况**

show status like ‘Handler\_read%';

大家可以注意：

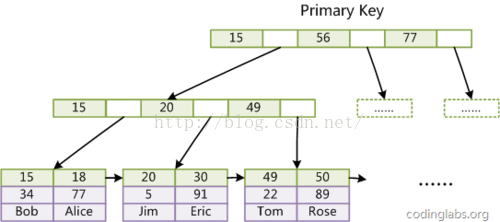
handler\_read\_key：这个值越高越好，越高表示使用索引查询到的次数

handler\_read\_rnd\_next：这个值越高，说明查询低效

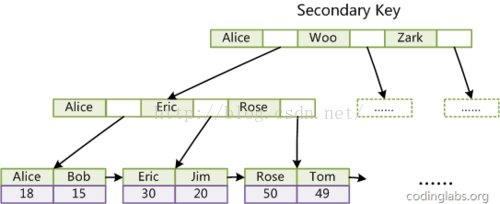
**11、InnoDB和MyISAM存储引擎**

<http://blog.csdn.net/sinat_30186009/article/details/52169057>

**InnoDB的主索引：**主索引的key是**数据表的主键**，因此InnoDB表数据文件本身就是主索引，所以必须有主键，如果没有显示定义，自动为生成一个隐含字段作为主键

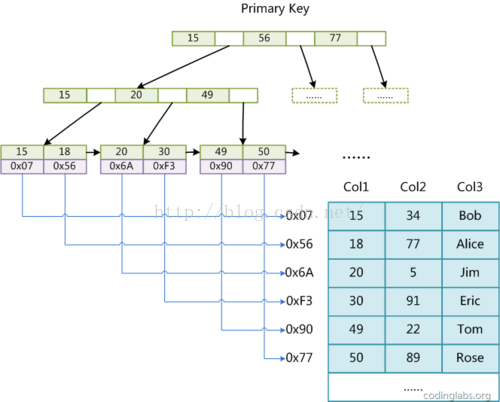


**InnoDB的辅助索引：**辅助索引也会包含主键列，比如**名字建立索引**，内部节点会包含名字，叶子节点会包含该名字对应的主键的值（下图以名字做一个辅助索引）

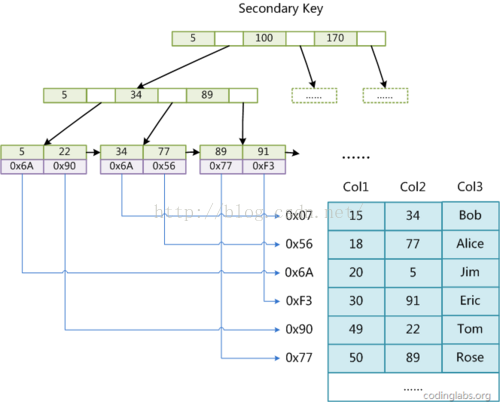


MyISAM主索引和辅助索引在结构上没有任何区别，只是***主索引要求key是唯一的，辅助索引可以重复，其叶子节点存储都是数据记录的地址***

主索引：



辅助索引：



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(聚集索引)，而MyISAM的索引和数据是分开的（非聚集索引）。
* **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。

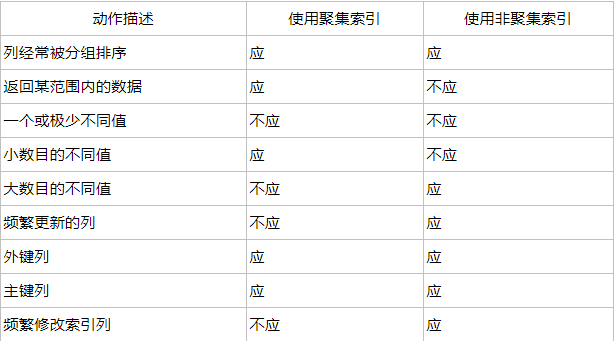
**区别：**

* InnoDB支持事务，MyISAM不支持，对于**InnoDB每一条SQL语言都默认封装成事务**，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；
* InnoDB支持**外键**，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；
* InnoDB是**聚集索引，数据文件是和索引绑在一起的**，必须要有**主键**，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而**MyISAM是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针**。主键索引和辅助索引是独立的。
* InnoDB不保存表的具体行数，执行select count(\*) from table时需要**全表扫描**。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；

**如何选择：**

* 是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；
* 如果表中绝大多数都只是**读查询**，可以考虑MyISAM，如果既有读写也挺频繁，请使用InnoDB。
* 系统奔溃后，MyISAM恢复起来更困难，能否接受；
* MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

何时使用聚集索引或非聚集索引（很重要、：



<http://kb.cnblogs.com/page/78142/> 数据库索引，你该了解的几件事

<http://jingyan.baidu.com/article/5d6edee2f269fe99eadeece9.html>

<http://www.cnblogs.com/aspnethot/articles/1504082.html> 聚集索引和非聚集索引（整理）

**12、数据库三范式**

<https://www.2cto.com/database/201203/122083.html>

<http://blog.csdn.net/w__yi/article/details/19934319>

* **数据冗余**：指数据之间的重复，也可以说是同一数据存储在不同数据文件中的现象；
* **函数依赖**：如果一个属性决定于另一个属性，那么可以说另一个属性依赖于这个属性，比如《机房收费系统》中学生表，学生号可以决定学生姓名，即知道了一个学生的学生号，就可以知道他的姓名。

函数依赖又可以分为平凡依赖和非平凡依赖。按性质分可分为部分依赖和完全依赖。

* **完全依赖**：设X,Y是关系R的两个属性集合，X’是X的**真子集**，存在X→Y，但对每一个X’都有X’!→Y，则称Y完全函数依赖于X。
* **部分依赖**：设X,Y是关系R的两个属性集合，存在X→Y，若X’是X的真子集，存在X’→Y，则称Y**部分函数**依赖于X。
* **传递函数依赖**：设X,Y,Z是关系R中互不相同的属性集合，存在X→Y(Y !→X),Y→Z，则称Z传递函数依赖于X。

数据库范式是为了**消除数据冗余、减少数据操作异常**出现的。

* **第一范式**：原子性，字段不可再分；是指数据库表的**每一列都是不可分割的基本数据项**，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个**新的实体**，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，**第一范式就是无重复的列**。
* **第二范式**：**非主属性完全依赖于主键 [消除部分子函数依赖]**。在第一范式基础上，要求数据表的中的每个实例或行必须可以被唯一的区分。为实现区分通常需要为表加上一个列，一储存各个实例的唯一标示，即没有重复的记录。这唯一标识被称为主键或关键字、主码。
* **第三范式**：**其它非主属性不依赖于其主属性，应移除 [消除传递依赖]**。第二范式基础上，消除间接依赖（某些字段不完全依赖主键的，可以将其移除）

满足第三范式的数据库表应该不存在如下依赖关系：关键字段 → 非关键字段x → 非关键字段y

**13、为什么要创建索引**

<http://blog.csdn.net/hguisu/article/details/7856239>

创建索引可以大大提高系统的性能。

* 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
* 可以**大大加快数据的检索速度**，这也是创建索引的最主要的原因。
* 可以加速表和表之间的连接，特别是在实现**数据的参考完整性**方面特别有意义。
* 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
* 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

* **创建索引和维护索引**要耗费时间，这种时间随着数据量的增加而增加。
* 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。
* 当对表中的数据进行**增加、删除和修改**的时候，索引也要**动态的维护**，这样就降低了数据的维护速度。

**14、在哪建索引**

<http://blog.csdn.net/hguisu/article/details/7856239>

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，应该在这些列上创建索引：

* 在经常需要搜索的列上，可以加快搜索的速度；
* 在作为主键的列上，强制该列的**唯一性**和组织表中数据的排列结构；
* 在经常用在连接的列上，这些列主要是一些**外键**，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为**索引已经排序**，其指定的范围是连续的；
* 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
* 在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。一般来说，**不应该创建索引**的的这些列具有下列特点：

* 对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
* 对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，**在查询的结果中，结果集的数据行占了表中数据行的很大比例**，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。
* 对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少,不利于使用索引。
* 当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改操作远远多于检索操作时，不应该创建索引。

**15、数据库锁**

<http://www.cnblogs.com/ismallboy/p/5574006.html> （概念，理论）

<http://blog.csdn.net/samjustin1/article/details/52210125> （实例理解，详细易懂）

数据库锁出现的原因是为了处理并发问题，因为数据库是一个多用户共享的资源，当出现并发的时候，就会导致出现各种各样奇怪的问题，就像程序代码一样，出现多线程并发的时候，如果不做特殊控制的话，就会出现意外的事情，比如“脏“数据、修改丢失等问题。所以数据库并发需要使用事务来控制，事务并发问题需要**数据库锁**来控制，所以数据库锁是跟并发控制和事务联系在一起的。

**1）事务特性**

事务的特性，要想成为事务，必须满足：ACID（原子性，一致性，隔离性，持久性）四特性，事务是恢复和并发控制的基本单位。

* 原子性指的是事务是数据库的逻辑工作单位，事务中操作要么都做，要么都不做；
* 一致性指的是事务的执行结果必须是使数据库从一个一致性状态变大另一个一致性状态，一致性和原子性是密切相关的；
* 隔离性指的是一个事务执行不能被其他事务干扰；
* 持久性指的是一个事务一旦提交，他对数据库中数据的改变就是永久性的

**2）悲观锁和乐观锁**

并发控制一般采用三种方法，分别是乐观锁和悲观锁以及时间戳。

* 乐观锁认为一个用户读数据的时候，别人不会去写自己所读的数据；

**解释：**乐观锁就比较简单了，就是不做控制，这只是一部分人对于并发所持有的一种态度而已。

* 悲观锁就刚好相反，觉得自己读数据库的时候，别人可能刚好在写自己刚读的数据，其实就是持一种比较保守的态度；

**解释：**悲观锁就是在读取数据的时候，为了不让别人修改自己读取的数据，就会先对自己读取的数据加锁，只有自己把数据读完了，才允许别人修改那部分数据，或者反过来说，就是自己修改某条数据的时候，不允许别人读取该数据，只有等自己的整个事务提交了，才释放自己加上的锁，才允许其他用户访问那部分数据。

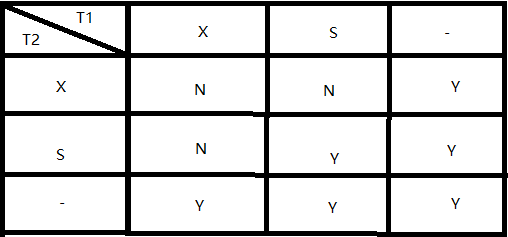
* 时间戳就是不加锁，通过时间戳来控制并发出现的问题。

**解释：**时间戳就是在数据库表中单独加一列时间戳，比如“TimeStamp”，每次读出来的时候，把该字段也读出来，当写回去的时候，把该字段加1，提交之前 ，跟数据库的该字段比较一次，如果比数据库的值大的话，就允许保存，否则不允许保存，这种处理方法虽然不使用数据库系统提供的锁机制，但是这种方法可以大大提高数据库处理的并发量，因为这种方法可以避免了长事务中的数据库加锁开销（操作员A 和操作员B操作过程中，都没有对数据库数据加锁），大大提升了大并发量下的系统整体性能表现。

需要注意的是，乐观锁机制往往基于系统中的数据存储逻辑，因此也具备一定的局限性，如在上例中，由于乐观锁机制是在我们的系统中实现，来自外部系统的用户余额更新操作不受我们系统的控制，因此可能会造成**脏数据被更新到数据库**中。在系统设计阶段，我们应该充分考虑到这些情况出现的可能性，并进行相应调整（如将乐观锁策略在数据库存储过程中实现，对外只开放基于此存储过程的数据更新途径，而不是将数据库表直接对外公开）。以上悲观锁所说的加“锁”，其实分为几种锁，分别是：排它锁和共享锁，其中排它锁又称为写锁，共享锁又称为读锁。

**3）排它锁和共享锁**

共享锁和排它锁是具体的锁，是数据库机制上的锁，存在以下关系：（x表示是排它锁(Exclusive)，s表示共享锁(Share)，Y表示yes，N表示no）



上图**表示可以共存的锁**，如，第二行表示，一个事务T1给某数据加了X锁，则事务T2就不能再给那数据加X锁了，同时也不能再加S锁了，只有到T1事务提交完成之后，才可以。默认来说，当sql脚本修改更新某条记录的时候，会给该条记录加X锁，读的话加的是S锁。

**4）死锁和活锁**

**并发控制会造成活锁和死锁**，就像操作系统那样，会因为互相等待而导致。

* 活锁指的是T1封锁了数据R，T2同时也请求封锁数据R，T3也请求封锁数据R，当T1释放了锁之后，T3会锁住R，T4也请求封锁R，则T2就会一直等待下去；

解决方法：这种处理方法就是采用“先来先服务”策略；

* 死锁就是我等你，你又等我，双方就会一直等待下去，比如：T1封锁了数据R1，正请求对R2封锁，而T2封住了R2，正请求封锁R1，这样就会导致死锁，死锁这种没有完全解决的方法，只能尽量预防。

**预防的方法：**

①一次封锁发，指的是一次性把所需要的数据全部封锁住，但是这样会扩大了封锁的范围，降低系统的并发度；

②顺序封锁发，指的事先对数据对象指定一个封锁顺序，要对数据进行封锁，只能按照规定的顺序来封锁，但是这个一般不大可能的。

**系统如何判断出现死锁呢？**

毕竟出现死锁不能一直干等下去，要及时发现死锁同时尽快解决出现的死锁，诊断和判断死锁有两种方法，一是超时法，二是等待图法。

* 超时法就是如果某个事物的等待时间超过指定时限，则判定为出现死锁；
* 等待图法指的是如果事务等待图中出现了回路，则判断出现了死锁。

**解决死锁的方法：**只能是撤销一个处理死锁代价最小的事务，释放此事务持有的所有锁，同时对撤销的事务所执行的数据修改操作必须加以恢复。

**5）行级锁和表级锁**

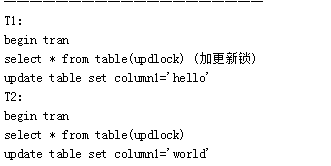
锁又包括行级锁和表级锁

* 行级锁是一种排他锁，防止其他事务修改此行；在使用以下语句时，Oracle会自动应用行级锁：INSERT、UPDATE、DELETE、SELECT … FOR UPDATE [OF columns] [WAIT n | NOWAIT]；SELECT … FOR UPDATE语句允许用户一次锁定多条记录进行更新；使用**COMMIT或ROLLBACK语句释放锁**。
* 表级锁又分为5类：
* 行共享 (ROW SHARE) – 禁止排他锁定表
* 行排他(ROW EXCLUSIVE) – 禁止使用排他锁和共享锁
* 共享锁(SHARE) - 锁定表，对记录**只读不写**，多个用户可以同时在同一个表上应用此锁
* 共享行排他(SHARE ROW EXCLUSIVE) – 比共享锁更多的限制，禁止使用共享锁及更高的锁
* 排他(EXCLUSIVE) – 限制最强的表锁，仅允许其他用户查询该表的行。禁止修改和锁定表。

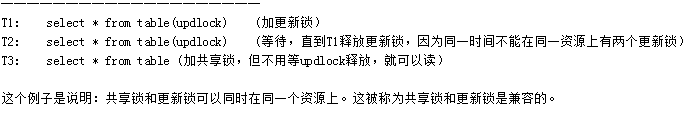
**6）更新锁uplock**

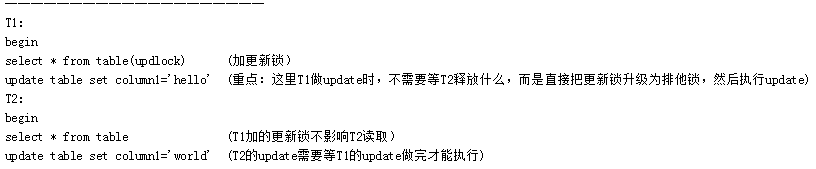
为解决死锁，引入更新锁。**更新锁的意思**是：“我现在只想读，你们别人也可以读，但我将来可能会做更新操作，我已经获取了从共享锁（用来读）到排他锁（用来更新）的资格”。一个事物只能有一个更新锁获此资格。

实例：



T1执行select，加更新锁。T2运行，准备加更新锁，但发现已经有一个更新锁在那儿了，只好等。当后来有user3、user4...需要查询table表中的数据时，并不会因为T1的select在执行就被阻塞，照样能查询，相比起例6，这提高了效率。





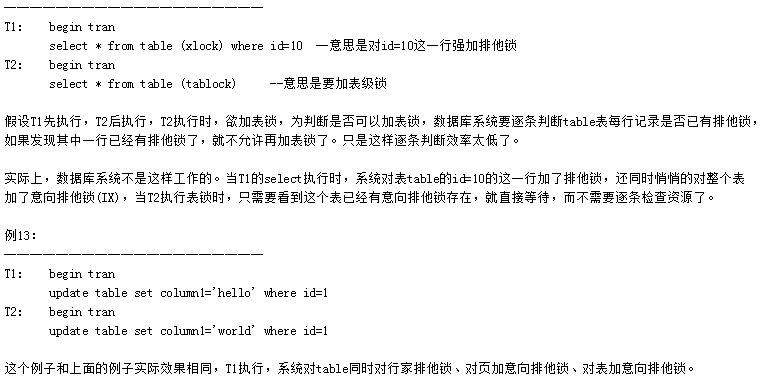
我们以这个例子来加深更新锁的理解：

* **第一种情况**：T1先达，T2紧接到达；在这种情况中，T1先对表加更新锁，T2对表加共享锁，假设T2的select先执行完，准备执行update，发现已有更新锁存在，T2等。T1执行这时才执行完select，准备执行update，更新锁升级为排他锁，然后执行update，执行完成，事务结束，释放锁，T2才轮到执行update。
* **第二种情况**：T2先达，T1紧接达；在这种情况，T2先对表加共享锁，T1达后，T1对表加更新锁，假设T2 select先结束，准备update，发现已有更新锁，则等待，后面步骤就跟第一种情况一样了。
* 这个例子是说明：**排他锁与更新锁是不兼容的，它们不能同时加在同一子资源上**

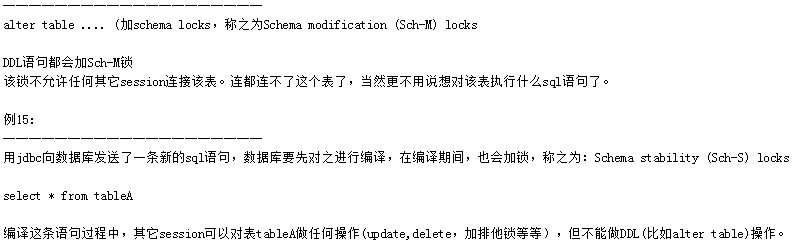
**7）意向锁（Intent Lock）**

意向锁就是说在屋（表示一个表）门口设置一个标识，说明屋子里有人（比如代表某些记录）被锁住了。另一个人想知道屋子里是否有人被锁，不用进屋子里一个一个的去查，直接看门口标识就行了。

当一个表中的某一行被加上排他锁后，该表就不能再被加表锁。数据库程序如何知道该表不能被加表锁？一种方式是逐条的判断该表的每一条记录是否已经有排他锁，另一种方式是直接在表这一层级检查表本身是否有意向锁，不需要逐条判断。显然后者效率高。



**8）计划锁（Schema Lock）**



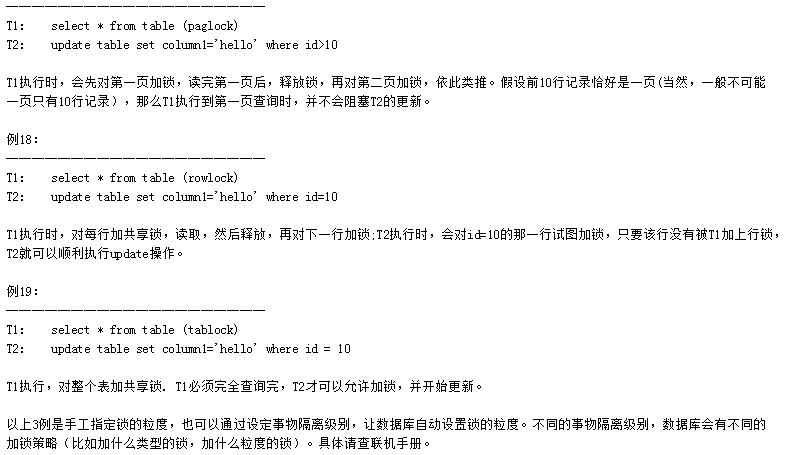
**9）何时加锁？**

如何加锁，何时加锁，加什么锁，你可以通过hint手工强行指定，但大多是数据库系统自动决定的。这就是为什么我们可以不懂锁也可以高高兴兴的写SQL。



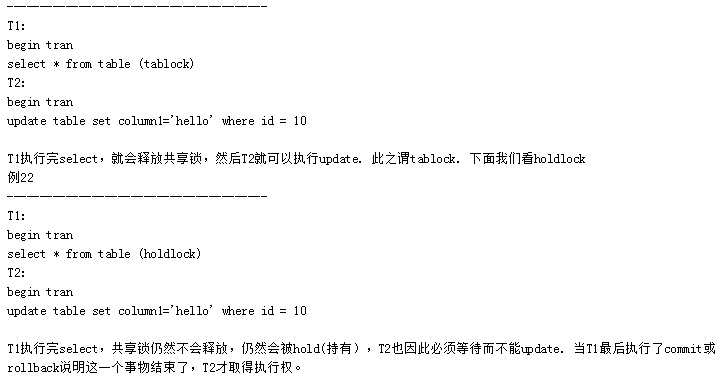
**10）锁的粒度**

锁的粒度就是指锁的生效范围，就是说是行锁，还是页锁，还是整表锁。锁的粒度同样既可以由数据库自动管理，也可以通过手工指定hint来管理。

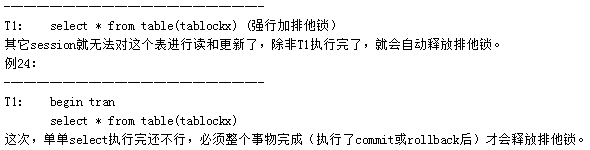


**11）数据库锁的区别**

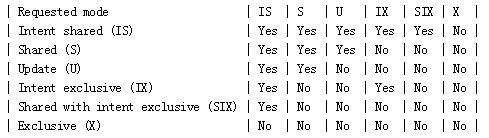
* holdlock 对表加共享锁，且事务不完成，共享锁不释放。
* tablock 对表加共享锁，只要statement不完成，共享锁不释放。与holdlock区别，见下例：



* TABLOCKX 对表加排他锁



各种锁的兼容关系：



**16、数据库SQL优化**

<http://www.cnblogs.com/yunfeifei/p/3850440.html>

<https://www.zhihu.com/question/36431635?sort=created>

常见的数据库SQL优化手段：

* 对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引
* 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num is null。最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库。

备注、描述、评论之类的可以设置为 NULL，其他的最好不要使用NULL。不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了，不管是否插入值（NULL也包含在内），都是占用 100个字符的空间的，如果是varchar这样的变长字段， null 不占用空间。可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：select id from t where num = 0

* 应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描
* 应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num=10 or Name = 'admin'。可以这样查询：select id from t where num = 10 **union all** select id from t where Name = 'admin'（union all：联合）
* in 和 not in 也要慎用，否则会导致全表扫描，如：select id from t where num in(1,2,3)；

对于连续的数值，能用 between 就不要用 in 了：select id from t where num between 1 and 3；

很多时候用 exists 代替 in 是一个好的选择：select num from a where num in(select num from b)；用下面的语句替换：select num from a where num exists(select 1 from b where num=a.num)

* 下面的查询也将导致全表扫描：select id from t where name like ‘%abc%’（前导模糊查询）；若要提高效率，可以考虑全文检索；
* 如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：select id from t where num = @num。可以改为强制查询使用索引：select id from t with(index(索引名)) where num = @num。
* 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：select id from t where num/2 = 100；应改为：select id from t where num = 100\*2。
* 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3) = ’abc’–name以abc开头的id

select id from t where datediff(day,createdate,’2005-11-30′) = 0 –‘2005-11-30 -生成的id

应改为：select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate < '2005-12-1'

* 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。
* 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。
* 不要写一些没有意义的查询，如需要生成一个空表结构：select col1,col2 into #t from t where 1=0；这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：create table #t(…)
* Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志；
* 对于多张大数据量（这里几百条就算大了）的表JOIN，要**先分页再JOIN**，否则逻辑读会很高，性能很差。
* select count(\*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。
* 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会**重建索引**，所以怎样建索引需要慎重考虑，视具体情况而定。**一个表的索引数最好不要超过6个**，若太多则应考虑一些不常使用到的列上建的索引是否有必要。
* 应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。
* 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。
* 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些；
* 任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段；
* 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）；
* 避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表；
* 在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert；
* 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。
* 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写；
* 使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。
* 与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。
* 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。
* 尽量避免大事务操作，提高系统并发能力。
* 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

**17、数据库分区分表**

<http://www.cnblogs.com/bluebluesky/articles/6413831.html>

**1）分区**

<http://blog.csdn.net/tjcyjd/article/details/11194489> 深入解析MySQL分区（Partition）功能

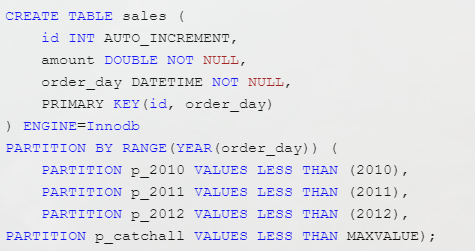
<http://blog.csdn.net/yongchao940/article/details/55266603>

就是把一张表的数据分成N个区块，在逻辑上看最终只是一张表，但底层是由N个物理区块组成的。

**什么时候考虑使用分区？**

* 一张表的查询速度已经慢到影响使用的时候。
* sql经过优化
* 数据量大
* 表中的数据是分段的
* 对数据的操作往往只涉及一部分数据，而不是所有的数据

mysql5 开始支持分区功能



分区的两种方式：

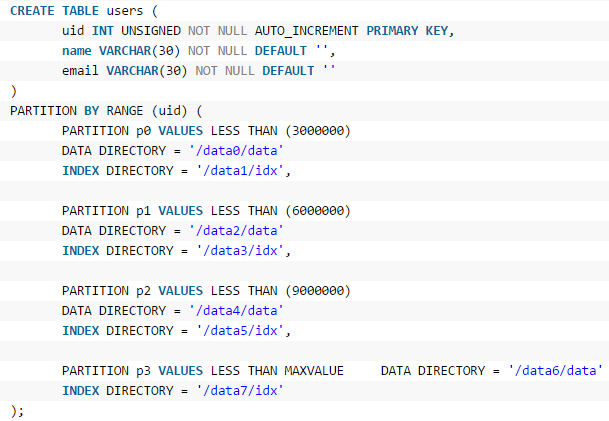
* **横向分区（水平）：**举例来说明一下，假如有100W条数据，分成十份，前10W条数据放到第一个分区，第二个10W条数据放到第二个分区，依此类推。也就是把表分成了十分，根用merge来分表。取出一条数据的时候，这条数据包含了表结构中的所有字段，也就是说横向分区，并没有改变表的结构。又比如，一个包含十年发票记录的表可以被分区为十个不同的分区，每个分区包含的是其中一年的记录。

**水平分区的几种模式：**

* **Range（范围）** – 这种模式允许DBA将数据划分不同范围。例如DBA可以将一个表通过年份划分成三个分区，80年代（1980's）的数据，90年代（1990's）的数据以及任何在2000年（包括2000年）后的数据；
* **Hash（哈希）** – 这中模式允许DBA通过对表的一个或多个列的Hash Key进行计算，最后通过这个Hash码不同数值对应的数据区域进行分区，。例如DBA可以建立一个对表主键进行分区的表；
* **Key（键值）** – 上面Hash模式的一种延伸，这里的**Hash Key**是MySQL系统产生的；
* **List（预定义列表）** – 这种模式允许系统通过DBA定义的列表的值所对应的行数据进行分割。例如：DBA建立了一个横跨三个分区的表，分别根据2004年2005年和2006年值所对应的数据；
* **Composite（复合模式）** - 以上模式的组合使用而已。举例：在初始化已经进行了Range范围分区的表上，我们可以对其中一个分区再进行hash哈希分区。
* **纵向分区（垂直）：**举例来说明，在设计用户表的时候，开始的时候没有考虑好，而把个人的所有信息都放到了一张表里面去，这样这个表里面就会有比较大的字段，如个人简介，而这些简介呢，也许不会有好多人去看，所以等到有人要看的时候，在去查找，分表的时候，可以把这样的大字段，分开来。

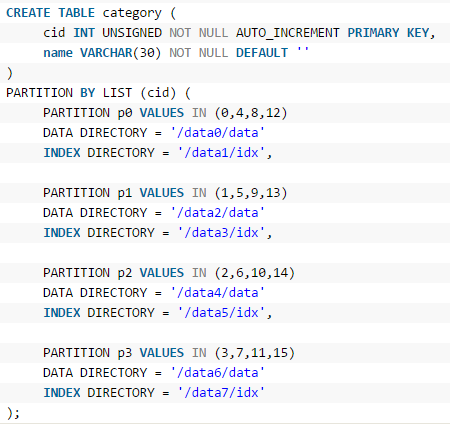
**A. RANGE 类型**

将用户表分成4个分区，以每300万条记录为界限，每个分区都有自己独立的数据、索引文件的存放目录（Data Directory和Index Directory），与此同时，这些目录所在的物理磁盘分区可能也都是完全独立的，可以提高磁盘IO吞吐量。



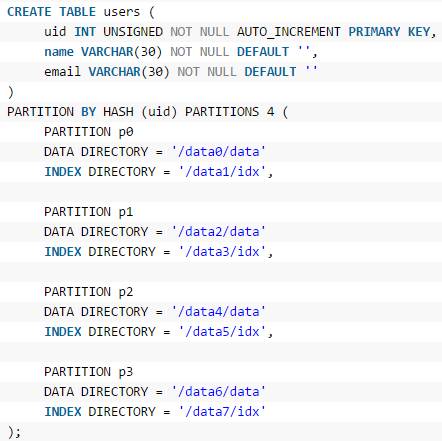
**B. List 类型**

分成4个区，数据文件和索引文件单独存放

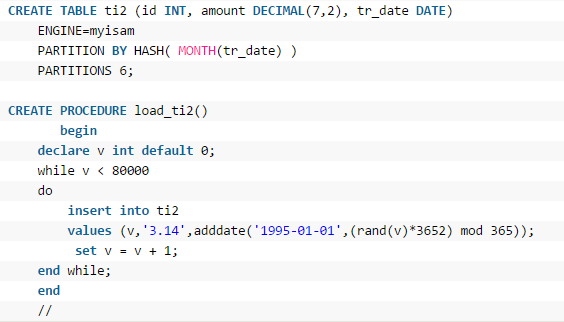


**C. Hash 类型**

分成4个区，数据文件和索引文件单独存放。

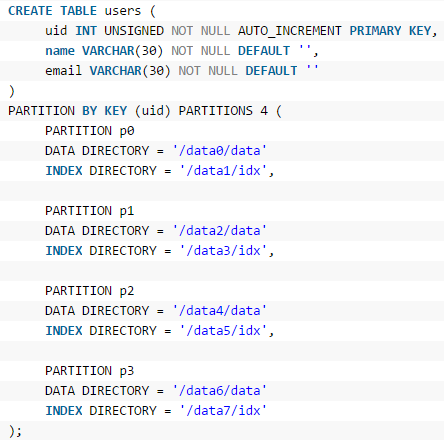


实例：



**D. Key 类型**

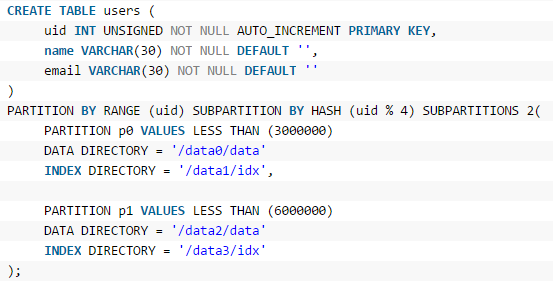
分成4个区，数据文件和索引文件单独存放。



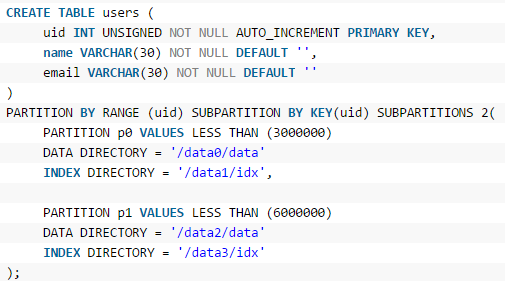
**E. 子分区**

子分区（SubPartitions）是针对 RANGE/LIST 类型的分区表中每个分区的再次分割。再次分割可以是 HASH/KEY 等类型。例如：

* 对 RANGE 分区再次进行子分区划分，子分区采用 HASH 类型：



* 对 RANGE 分区再次进行子分区划分，子分区采用 KEY 类型：



**注意：**

* 如果一个分区中创建了子分区，其他分区也要有子分区
* 如果创建了了分区，每个分区中的子分区数必有相同
* 同一分区内的子分区，名字不相同，不同分区内的子分区名子可以相同（5.1.50不适用）

**2）分表**

<http://blog.51yip.com/mysql/949.html>

就是把一张表按一定的规则分解成N个具有独立存储空间的实体表。系统读写时需要根据定义好的规则得到对应的字表明，然后操作它。

**分区和分表的区别与联系：**

* 分区和分表的目的都是减少数据库的负担，提高表的增删改查效率。
* 分区只是一张表中的数据的存储位置发生改变，分表是将一张表分成多张表。
* 当访问量大，且表数据比较大时，两种方式可以互相配合使用。
* 当访问量不大，但表数据比较多时，可以只进行分区。

**A. 为什么要分表？**

当一张的数据达到几百万时，你查询一次所花的时间会变多，如果有联合查询的话，我想有可能会死在那儿了。分表的目的就在于此，减小数据库的负担，缩短查询时间。根据个人经验，mysql执行一个sql的过程如下：接收到sql；把sql放到排队队列中；执行sql；返回执行结果。

在这个执行过程中最花时间在什么地方呢？第一，是排队等待的时间，第二，sql的执行时间。其实这二个是一回事，等待的同时，肯定有sql在执行。所以我们要缩短sql的执行时间。

mysql中有一种机制是**表锁定和行锁定（表级锁和行级锁）**，为什么要出现这种机制，是为了保证数据的完整性，举个例子来说吧，如果有二个sql都要修改同一张表的同一条数据，这个时候怎么办呢，是不是二个sql都可以同时修改这条数据呢？很显然mysql对这种情况的处理是，**一种是表锁定（myisam存储引擎），一个是行锁定（innodb存储引擎）**。表锁定表示你们都不能对这张表进行操作，必须等我对表操作完才行。行锁定也一样，别的sql必须等我对这条数据操作完了，才能对这条数据进行操作。如果数据太多，一次执行的时间太长，等待的时间就越长，这也是我们为什么要分表的原因。

分表方法：

* **做mysql集群，例如：利用mysql cluster ，mysql proxy，mysql replication，drdb等等**：有人会问mysql集群跟分表有什么关系吗？虽然它不是实际意义上的分表，但是它启到了分表的作用，做集群的意义是什么呢？为一个数据库减轻负担，说白了就是减少sql排队队列中的sql的数量，举个例子：有10个sql请求，如果放在一个数据库服务器的排队队列中，他要等很长时间，如果把这10个sql请求，分配到5个数据库服务器的排队队列中，一个数据库服务器的队列中只有2个，这样等待时间是不是大大的缩短了呢？这已经很明显了。所以我把它列到了分表的范围以内，我做过一些mysql的集群：

**优点：**扩展性好，没有多个分表后的复杂操作（php代码）

**缺点：**单个表的数据量还是没有变，一次操作所花的时间还是那么多，硬件开销大。

* **预先估计会出现大数据量并且访问频繁的表，将其分为若干个表**：这种预估大差不差的，论坛里面发表帖子的表，时间长了这张表肯定很大，几十万，几百万都有可能。 聊天室里面信息表，几十个人在一起一聊一个晚上，时间长了，这张表的数据肯定很大。像这样的情况很多。所以这种能预估出来的大数据量表，我们就事先分出个N个表，这个N是多少，根据实际情况而定。以**聊天信息表为例**：

事先建100个这样的表，message\_00、message\_01、message\_02、..........、message\_98、message\_99。然后根据用户的ID来判断这个用户的聊天信息放到哪张表里面，你可以用hash的方式来获得，可以用求余的方式来获得，方法很多，各人想各人的吧。下面用hash的方法来获得表名：

**优点：**避免一张表出现几百万条数据，缩短了一条sql的执行时间

**缺点：**当一种规则确定时，打破这条规则会很麻烦，例如开始使用hash算法是crc32，如果以后不想用这个算法了，改用md5后，会使同一个用户的消息被存储到不同的表中，这样数据乱套了。扩展性很差。

* **利用merge存储引擎来实现分表：**那些没有事先考虑，而已经出现了数据查询慢的情况。这个时候如果要把已有的大数据量表分开比较痛苦，最痛苦的事就是改代码，因为程序里面的sql语句已经写好了，现在一张表要分成几十张表，甚至上百张表，这样sql语句是不是要重写呢？举个例子，我很喜欢举子

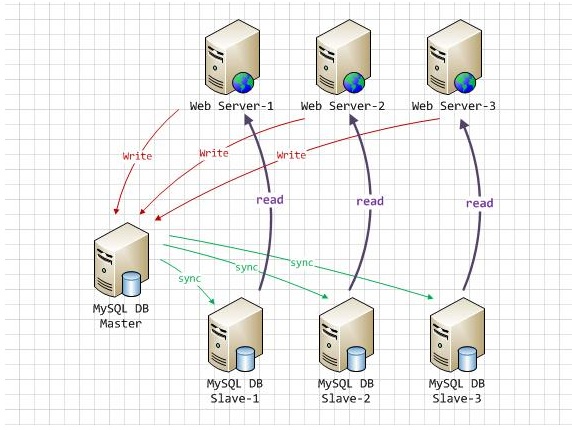
**18、数据库主从复制**

<http://blog.csdn.net/u010098331/article/details/50828820>

<http://www.cnblogs.com/luckcs/articles/2543607.html>

MySQL 内建的复制功能是构建大型高性能应用程序的基础。将 MySQL的数据分布到到多个系统上去，这种分步的机制，是通过将 MySQL 的某一台主机的数据复制到其它主机( Slave )上，并**重新执行一遍**来实现的。复制过程中一个服务器充当服务器，而一个或多个其它服务器充当从服务器。主服务器将更新记录写入二进制日志，并维护文件的一个索引以跟踪日志循环，这些日志可以记录发送到从服务器的更新。当一个从服务器连接主服务器时，它通知主服务器从服务器在日志中读取的最后一次成功更新的位置，从服务器接收从那时起发生的任何更新，然后封锁并等待主服务器通知新的更新。**注意**当你进行复制时，所有对复制中的表的更新必须在主服务器上进行。否则，你必须要小心，以避免用户对主服务器上的表进行的更新与对服务器上的表所进行的更新之间的冲突

**1）读写分离和主从复制**



如上图所示，当web server1/2/3要写入数据时，则向mysql db Master(主服务器）发出写入请求（即写入到master），如果要进入读操作时，则只向从服务器 mysql DB Slave1或2或3发出读取请求。如此将原本读写在同一台服务器的工作量分摊到了一台负责写入，N台负责读取（大部分的网站都是读取请求远大于写入请求），从而从一定程度时实现了负载均衡（如果有N台从服务器，则由主从代理系统自动分配某个具体请求从哪一个从服务器读取）。

**读写分离是靠主从复制来实现**的。即当一个数据写入到主服务器后，主服务器会将写入信息写入到binlog（二进制日志）里，同时同步（或异步或半同步）到从服务器里。从服务器根据主服务器传来的binlog，生成relay-log(中继日志），然后mysql服务器再用relay-log的信息将数据写入到数据库。

如此做的优点除了实现负责均衡之外，还保留了两份实时热备的数据binlog和relay-log。当服务器发生灾难时，我们可以用他们将数据恢复到任何一个时间点。

**2）MySQL支持的复制类型**

* **基于语句的复制**：在主服务器上执行的 SQL 语句，在从服务器上执行同样的语句。否则，你必须要小心，以避免用户对主服务器上的表进行的更新，与对服务器上的表所进行的更新之间的冲突，配置：binlog\_format = 'STATEMENT'
* **基于行的复制**：把改变的内容复制过去，而不是把命令在从服务器上执行一遍，从 MySQL 5.0开始支持，配置：binlog\_format = 'ROW'；
* **混合类型的复制**：默认采用基于语句的复制，一旦发现基于语句的无法精确的复制时，就会采用基于行的复制,配置：binlog\_format = 'MIXED'。

mysql复制解决的问题：

* 数据分布
* 负载平衡
* 备份
* 高可用性和容错行

**3）复制步骤**

* 1、Master 将改变记录到二进制日志中。
* 2、Slave 将 Master 的二进制日志拷贝到它的中继日志( Relay\_log )
* 3、Slave 重做中继日志中的事件，将改变反映它自己的数据

**说明：**

* 1、Master 记录二进制的日志。在每个事务更新数据之前，Master 在**二进制日志**记录这些改变。 MySQL 将事务日志的写入二进制日志，即使事务中的语句相互交叉执行的。在事件写入二进制日志完成后，Master 通知存储引擎提交事务。
* 2、Slave 将 Master 的 Binary log 拷贝到它自己的**中继日志**。首先 Slave 开始一个工作线程--I/O线程。I/O 线程在 Master 上打开一个连接，然后开始从二进制日志中读取事件，如果已经连上 Master，它会等待master产生新的事件，随后I/O线程将事件写入中继日志。
* 3、SQL Slave Thread ( SQL从线程)处理该过程的最后一步。SQL从中继日志读取事件，并重做其内的事件并更新 Slave 的数据。使其它与 Master 中的数据保持一致。只要该线程与 I/O 线程保持一致，中继日志通常会位于 OS 的缓存中，所以中继日志的开销很小。
* 此处，在 Master 中也有一个工作线程，和其他 MySQL 的连接一样，Slave 在 Master 中打开一个连接也会使得 Master 开始一个线程。**复制过程有一个很重要的限制**---复制在 Slave 上是串行化的，也就是说 Master 上的并行更新操作不能在 Slave 上并行操作。

**4）复制类型**

复制的体系结构基本原则：

* 每个 Slave 只能有一个 Master；
* 每个 Slave 只能有一个唯一的服务器ID；
* 每个 Master 可以有很多 Slave;
* 如果你设置了 log\_slave\_updates，Slave 可以是其他 Slave 的 Master，从而扩散 Master 的更新
* MySQL 不支持多主服务器复制---即一个 Slave 可以有多个 Master，但是，通过一些简单的组合，我们却可以建立灵活而强大的复制体系结构

**A. 一主多从复制架构**

* **场景：**在主库读取请求压力非常大的场景下，可以通过配置一主多从复制架构实现读写分离，把大量对实时性要求不是特别高的**读请求通过负载均衡**到多个从库上，降低主库的读取压力。在主库出现异常宕机的情况下，可以把一个从库切换为主库继续提供服务。
* **建议：**
* 当 Slave 增加到一定数量时，Slave 对 Master 的**负载以及网络带宽**都会成为一个严重的问题。
* 不同的 Slave 扮演不同的作用(例如使用**不同的索引**，或者**不同的存储引擎**)
* 用一个 Slave 作为**备用 Master**，只进行**复制**
* 用一个远程的 Slave，用于**灾难恢复**。

**B. 多级复制架构**

* **场景：**一主多从的架构能够解决大部分读请求压力特别大的场景需求，但主库的I/O压力和网络压力会随着从库的增加而增长，而使用多级复制架构就可以解决一主多从场景下，主库额外的I/O和网络压力。 但要注意的是，多级复制场景下主库的数据是经历两次才到达读取的从库，期间的**延时**比一主多从复制场景下只经历一次复制的要大。
* **建议：**
* 可能存在延时较长的风险
* 这种方案可以与第三方软件结合使用，例如Slave+LVS+Keepalived 实现高可用。

**C. 双主复制/Dual Master架构**

* **场景：**双主/Dual Master架构适用于写压力比较大的场景，或者DBA做维护需要主从切换的场景，通过双主/Dual master架构避免了重复搭建从库的麻烦。
* **建议：**
* 最大问题就是更新冲突。
* 可以采用MySQL **Cluster**，以及将Cluster和Replication结合起来，可以建立强大的高性能的数据库平台。

**19、数据库表、临时表与视图的区别**

**1）表与视图**

<http://blog.csdn.net/u010846873/article/details/52244300>

<http://www.cnblogs.com/binyao/p/4899970.html>

表与视图很相似，都是可以包含相同类型的数据的二维结构，都有行，列，单元格，在select语句的from字句中，都可以将他们用作数据源。当需要区分表和视图的时候，通常表被称为基表或者数据表。

**区别：**

* 表直接将数据存储在磁盘上，视图是将sql语句存储到磁盘上；
* 视图是建立在表的基础上，表存储数据库中的数据，而视图显示已经在表中的数据的外观；
* 表是静态的，而视图是动态的，意思是表中数据发生了改变，其建立在表基础的视图跟着改变；
* 通过视图不能改变表中数据。
* 删除视图，表不受影响，而删除表，视图不再起作用
* 视图本身没有数据，只保存了sql语句
* 表是物理存在的，可以理解成计算机中的文件！视图是虚拟的内存表，可以理解成Windows的快捷方式！
* 视图是数据库数据的特定子集。可以禁止所有用户访问数据库表，而要求用户只能通过视图操作数据，这种方法可以保护用户和应用程序不受某些数据库修改的影响。
* 视图是抽象的，他在使用时，从表里提取出数据，形成虚的表。不过对他的操作有很多的限制 。
* 而且视图是永远不会自己消失的除非你删除它。视图有时会对提高效率有帮助。临时表几乎是不会对性能有帮助，是资源消耗者。视图一般随该数据库存放在一起，临时表永远都是在tempdb里的。
* 视图适合于多表连接浏览时使用!不适合增、删、改.，存储过程适合于使用较频繁的SQL语句，这样可以提高执行效率!

**2）临时表与视图**

<http://blog.csdn.net/zhangzeguang88/article/details/26134345>

* **视图：**从一个或多个表（或视图）导出的表。
* **视图与表的不同之处：**视图是一个虚表，即视图所对应的数据不进行实际存储，数据库只存储视图的定义，对视图的数据进行操作时，系统根据视图的定义去操作与视图相关联的基本表（**视图的本质特征**）。
* 视图主要用于系统的安全、查询和效率，在安全方面，举个例子：例如，你只想让用户看到某一表的某几个字段，有些字段想不让用户看见，这是用视图解决会很好，当然在select时也可以实现。第在查询方面，对于比较复杂的查询，可以大大减少频繁编写sql语句的烦恼。同时，在效率上，数据分布在多台服务器上，视图一定会带来效率上的好处。
* **临时表：**临时表与永久表相似，但临时表存储在 tempdb 中，当不再使用时会自动删除。
* 临时表有两种类型：**本地和全局**。它们在名称、可见性以及可用性上有区别。本地临时表的名称以单个数字符号 (#) 打头；它们仅对当前的用户连接是可见的；当用户从 SQL Server 实例断开连接时被删除。全局临时表的名称以两个数字符号 (##) 打头，创建后对任何用户都是可见的，当所有引用该表的用户从 SQL Server 断开连接时被删除。
* 例如，如果创建了 employees 表，则任何在数据库中有使用该表的安全权限的用户都可以使用该表，除非已将其删除。如果数据库会话创建了本地临时表 #employees，则仅会话可以使用该表，会话断开连接后就将该表删除。如果创建了 ##employees 全局临时表，则数据库中的任何用户均可使用该表。如果该表在您创建后没有其他用户使用，则当您断开连接时该表删除。如果您创建该表后另一个用户在使用该表，则 SQL Server 将在您断开连接并且所有其他会话不再使用该表时将其删除。临时表主要用来提高效率。
* **区别：**
* **空间分配：**物理空间的分配不一样，视图不分配空间，临时表会分配空间
* **虚实：**视图是一条预编译的SQL语句，并不保存实际数据，而临时表是保存在tempdb中的实际的表。即视图是一个快照，是一个虚表，而临时表是客观存在的表类型对象。它们的结构一个是表、一个快照。可以把视图想象成联合表的快捷方式。

**20、SQL注入问题与防止**

<http://www.cnblogs.com/klsw/p/5259969.html>

<http://blog.csdn.net/lifushan123/article/details/45226299>

* 永远不要信任用户的输入，要对用户的输入进行校验，可以通过正则表达式，或限制长度，对**单引号和双"-"进行转换**等（**’--’表示注释**，构造OR条件语句为真）。
* 永远不要使用动态拼装SQL，可以使用**参数化的SQL**或者直接使用存储过程进行数据查询存取，参数化是防SQL注入框架级方案的重要部分。（参数化存储过程进行数据查询存取）。即在程序中，组织SQL语句时，应该尽量将用户输入的字符串以参数的形式来进行包装，而**不是直接嵌入SQL语言。**

使用Java系列，可以使用PreparedStatement代替Statement；SQL Server数据库中可以使用存储过程，结合Parameters集合；Parameters集合提供了长度验证和类型检查的功能，Parameters集合内的内容将被视为字符值而不是可执行代码。

* 永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。（**严格区分数据库访问权限**）
* 不要把机密信息明文存放，请**加密**或者hash掉密码和敏感的信息。
* 应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装，把异常信息存放在独立的表中。即对**数据库查询**中的**出错信息进行屏蔽**，尽量减少攻击者根据数据库的查询出错信息来猜测数据库特征的可能；
* 多层架构下的防治策略：在多层环境下，**用户输入数据的校验与数据库的查询**被分离成多个层次。此时，应该采用以下方式来进行验证：
* 用户输入的所有数据，都需要进行**验证（输入数据校验）**，通过验证才能进入下一层；此过程与数据库分离的；
* 没有通过验证的数据，应该被数据库拒绝，并向上一层报告错误信息

**SQL注入攻击的主要危害包括：**

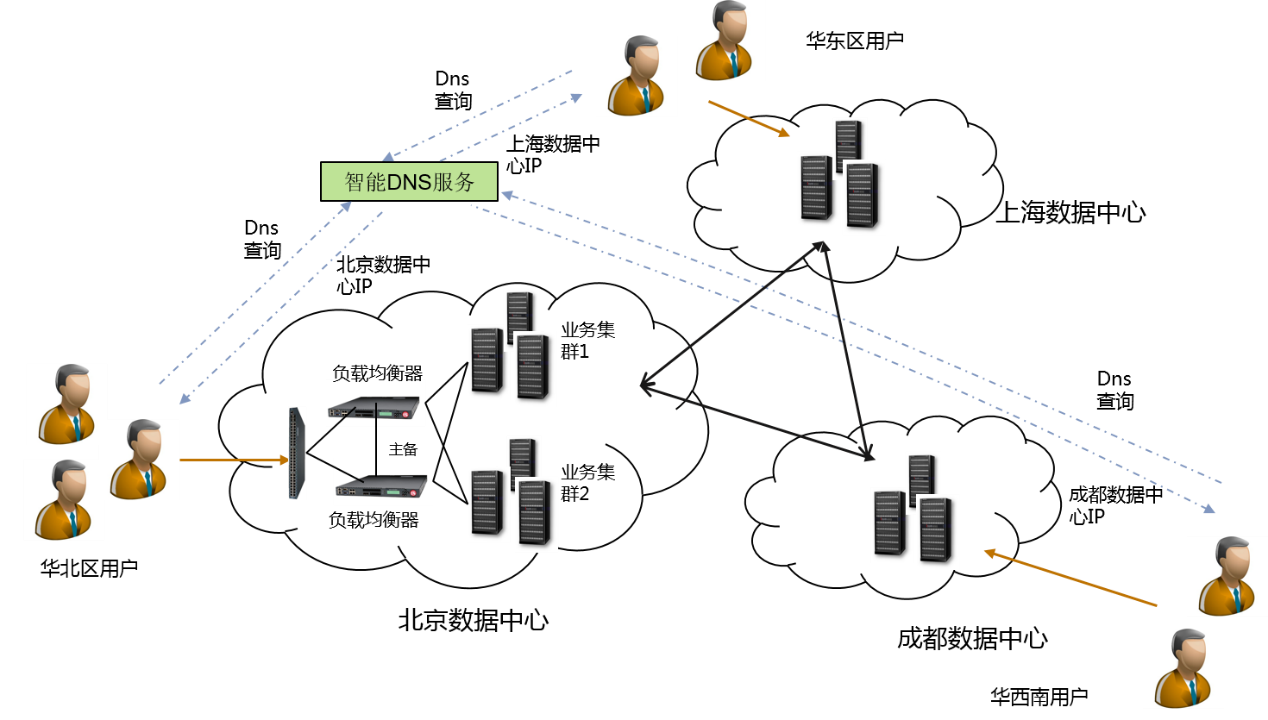
* 非法读取、篡改、添加、删除数据库中的数据；
* **盗取**用户的各类敏感信息，获取利益；
* 通过修改数据库来修改网页上的内容；
* 私自添加或删除账号；
* 注入木马；等等。

**21、互联网金融的高并发方案**

<http://blog.csdn.net/u012517198/article/details/51480911>

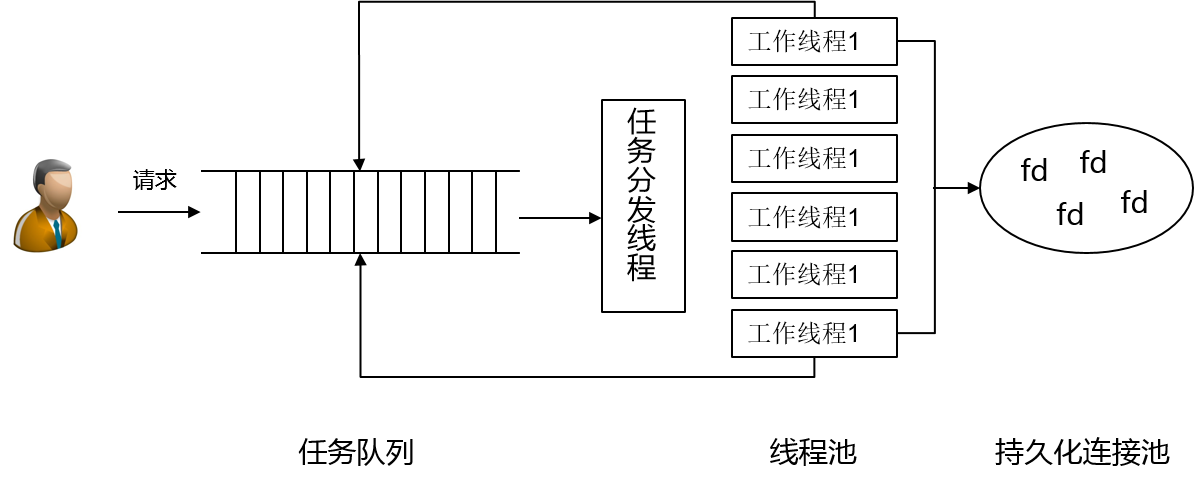
**1）负载均衡**

根据业务模型和业务服务协议，一般可选择的负载均衡方案包括：链路层负载均衡、IP层负载均衡、Http反向代理、DNS域名解析负载均衡、Http重定向负载均衡。大型网站或业务服务往往采用多种手段进行流量的负载均衡，比如先**基于DNS**实现多数据中心的负载均衡，再根据IP实现数据中心内多业务负载均衡，最后在基于反向代理实现统一业务的不同服务器之间的负载均衡。（**区域请求负载均衡**）



**2）异步IO**

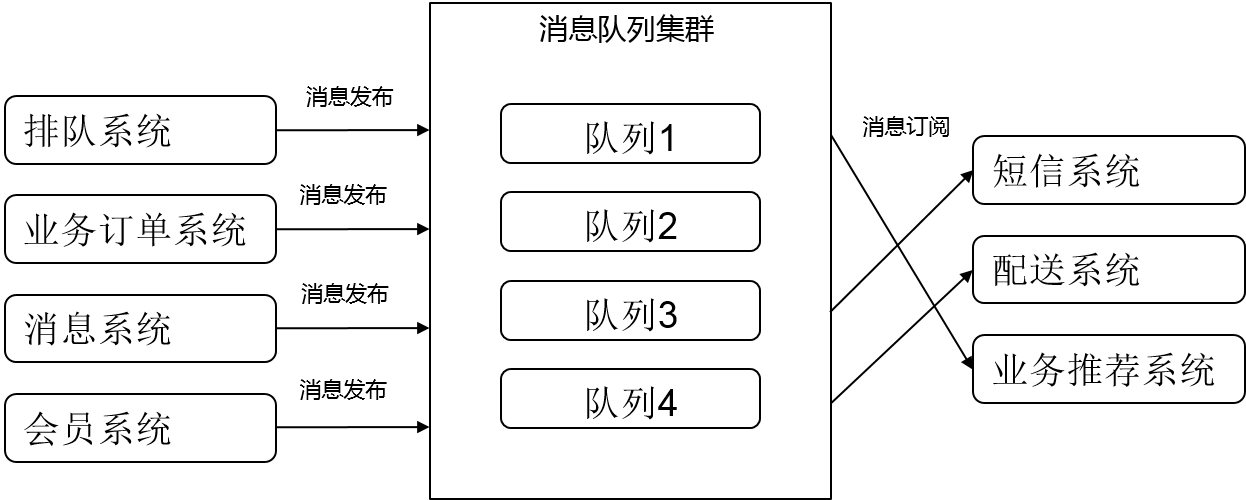
**异步IO**是提高系统并发性的**重要技术**，和异步IO共同出现的还有任务（消息）队列、线程池和持久化连接等技术。**异步IO技术是事件驱动的编程模型实际应用的典范**：用户请求先被放入任务队列，然后唤醒任务分发器，任务分发器从任务队列取下任务分发到空闲的线程上，线程触发异步IO操作并注册回调方法，当IO返回后回调方法重新从任务队列中把任务取下并把结果返回。整个过程如下：



**3）消息队列**

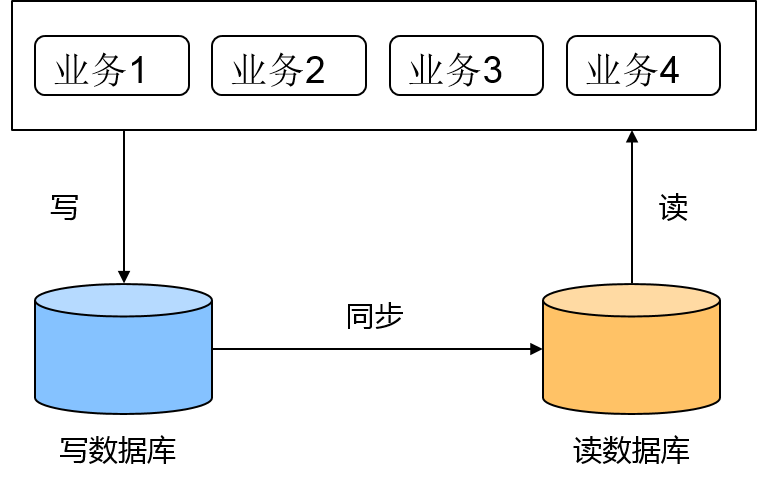
消息队列对于提高系统并发性能具有四个方面的作用：

* 通过消息队列实现**异步处理**，如上述异步IO中的任务队列就是可以基于消息队列实现
* 任务**并行执行**，通过消息队列可以把传统串行执行的任务尽量改造成可并行的程序；
* 应用**解耦**，提高系统的扩展性；
* 流量削峰，通过消息队列引入排队机制，可以把**尖峰负载尽量平整化**。下图为一个Web网站的消息系统。

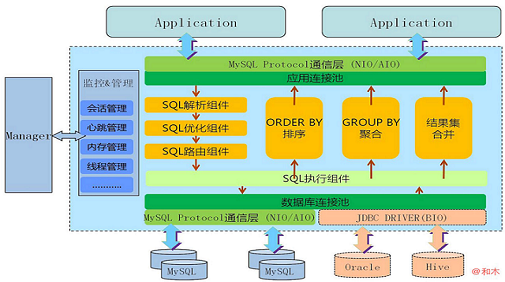


**4）数据库读写分离/分库分表**

随着访问量的增多，数据库系统的压力会越来越大。在一个信息系统中，数据库系统的性能往往是对系统整体性能影响最为关键的指标。从数据库架构设计的角度，常用的优化手段为**读写分离与分库分表**。前者是采用读写请求分别路由到不同的库中来降低数据库系统压力的一种技术，采用该技术可以最大程度上提高系统的并发读，特别是对**读多写少**的访问模式十分有效。两个库之间通过数据同步，可以确保数据的一致性。读写分离模式如下图示：



随着业务的运行，数据库中的数据量随之不断增多。当达到一定的记录条目时，一次查询往往需要消耗很长时间才能返回结果。这是**分库分表**设计就提到了日程。分库设计一般根据业务把不同的内容存到不同的数据库中，也成为垂直拆分。这种拆分模式比较灵活，也易于操作，不足之处在于需要考虑跨多数据库的符合业务查询join问题。分表设计也叫水平拆分，就是把同一个表中的数据拆分到两个甚至多个数据库中。产生数据水平拆分的原因是某个业务的数据量或者更新量到达了单个数据库的瓶颈，这时就可以把这个表拆分到两个或更多个数据库中。Mycat是最为常用的分库分库中间件，下图为Mycat的架构，有兴趣的同学可以前往Mycat官方网站学习了解。



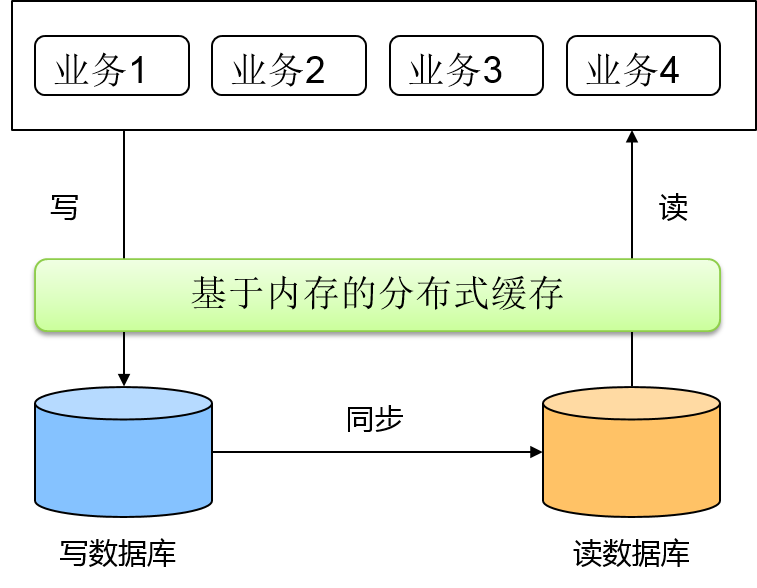
**5）服务拆分**

服务拆分是把过去全部运行在一个应用容器内部的业务逻辑子系统拆分出来，单独运行在独立的容器内部。这样做有两个好处：

* 可以降低系统耦合度，使得业务具备快速迭代能力；
* 方便的定位影响性能的子系统，针对性的进行性能优化。例如，短信子系统从整个系统中拆分出来后，系统可以方便的测试短信收发的并发效率及延迟，这样可以针对性的进行设计改进与架构优化。

**6）内存缓存**

随着访问量的增加，逐渐出现了许多用户访问同一部分内容的情况，对于这些比较**热门的内容**，没必要每次都从数据库读取。我们可以使用缓存技术，例如可以使用memcacahe作为应用层的缓存，也可以使用redis作为数据库层的缓存。另外，缓存系统也可以用来保存一些需要分享的数据，比如用户登录的会话信息（Session）。通过缓存系统共享会话是实现单点登录及会话管理的重要技术。加入缓存后的系统架构如下。



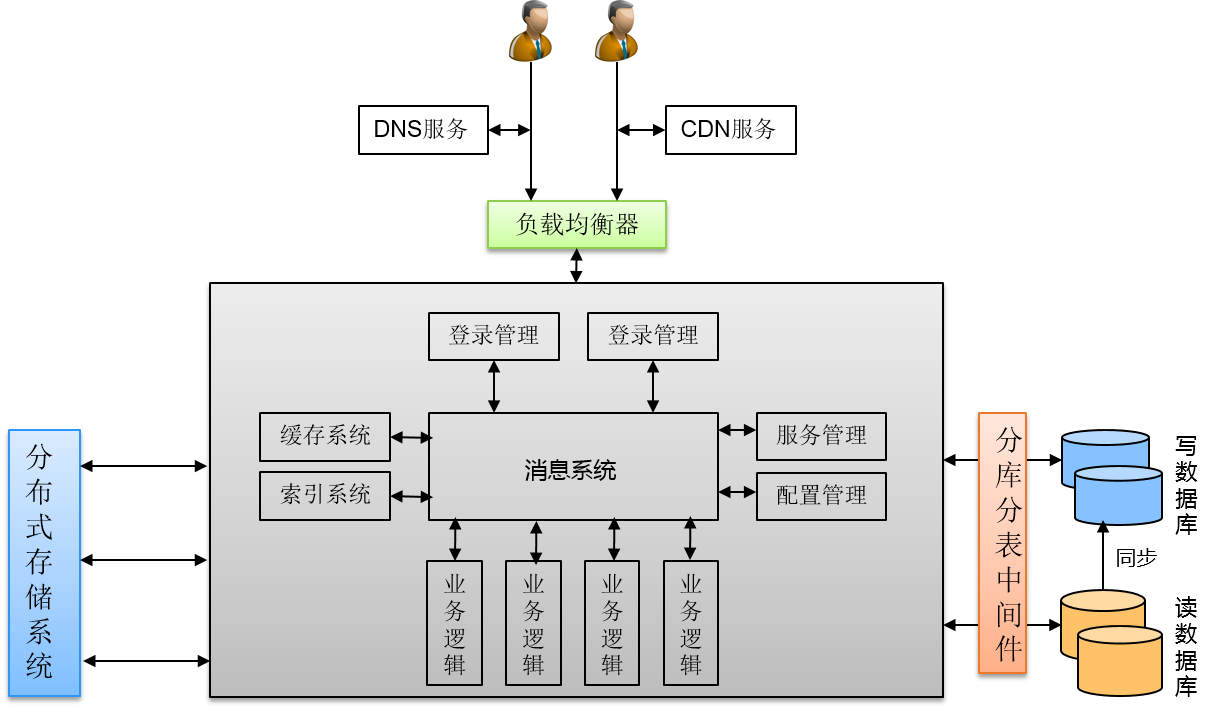
**7）索引系统**

对于模糊查找，利用读数据库进行查询往往力不从心，即使做了读写分离，这个问题依然是影响性能的一种重要场景。以交易网站型为例，基于关键词查找商品或服务是一种最为常用的功能，尤其是根据商品的标题来查找对应的商品。对于这种需求，在数据库操作中我们都是通过like功能来实现的，但是这种方式的开销很大，且针对大数量查询时非常耗时。此时我们可以使用**搜索引擎的索引**来完成。

**8）分布式存储系统/CDN**

针对非结构化数据的访问优化，一般的策略是构建分布式存储系统。支撑分布式存储系统是具备良好扩展性和并发性能的存储系统，设计良好的分布式存储系统能够实现访问文件的快速定位、加速读写、实现高并发性。例如ceph就是一个优秀的开源分布式存储系统。 CDN是更大尺度的优化手段，通常用户大型或超大型网络服务运营。利用CDN，可以把不常变化的资源放置在网络的边缘，加速终端用户获取资源的速度。

利用各种优化手段后整个互联网应用架构如下图所示：



**22、SQL group by语句介绍**

<http://www.cnblogs.com/allensun-193/p/5897925.html>

<http://blog.csdn.net/xxpyeippx/article/details/8059910>

“Group By”从字面意义上理解就是根据“By”指定的规则对数据进行分组，所谓的分组就是将一个“数据集”划分成若干个“小区域”，然后针对若干个“小区域”进行数据处理。而且group by语句中select指定的字段必须是**“分组依据字段”**，其他字段若想出现在select中则必须包含在**聚合函数**中，常见的聚合函数如下表：

* sum(列名)：求和
* max(列名)：最大值
* min(列名)：最小值
* avg(列名)：平均值
* count(列名)：统计记录数，注意和**count(\*)（求各组记录数目）**的区别

Having与Where的区别：

* where 子句的作用是在对查询结果进行分组前，将不符合where条件的行去掉，即在分组之前过滤数据，where条件中不能包含聚组函数，使用where条件过滤出特定的行，且必须在GROUP BY 子句之前指定 WHERE 子句。（**分组前过滤行数据**）
* having 子句的作用是筛选满足条件的组，即在分组之后过滤数据，条件中经常**包含聚组函数**，使用having 条件过滤出特定的组，也可以使用多个分组标准进行分组。（**分组后过滤组数据**）

写的顺序：select ... from... where.... group by... having... order by..

执行顺序：from... where...group by... having.... select ... order by...

**注意：**order by与group by 一起使用时，是先执行group by语句，后执行order by。而且MySQL的group by与Oracle有所不同，查询得字段可以不用写聚合函数，查询结果取得是每一组的第一行记录（注意使用时的区别）。

**五、Linux面试知识**

**1、进程间通信方式**

<http://www.cnblogs.com/melons/p/5791796.html>

<http://www.cnblogs.com/CheeseZH/p/5264465.html>

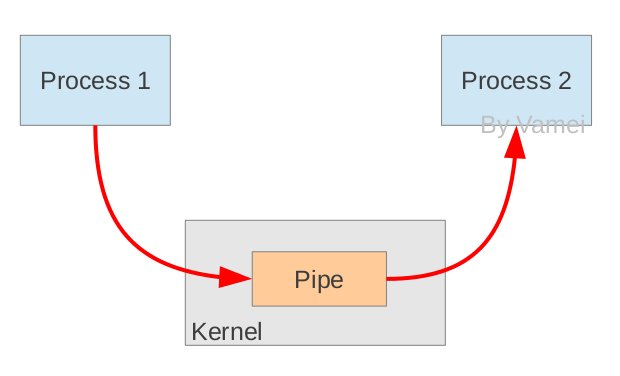
<http://blog.csdn.net/c15522627353/article/details/52972941>

进程间通信指在不同进程之间传播或交换信息。

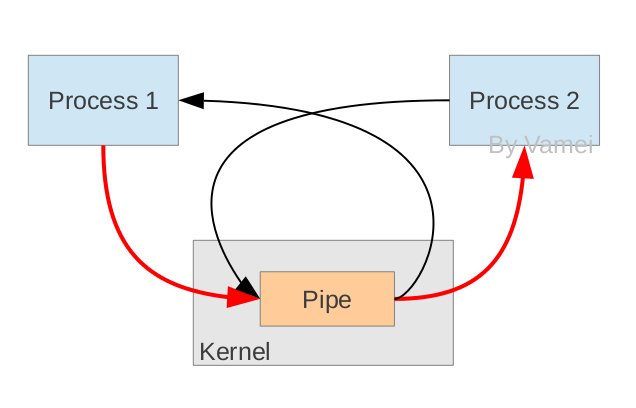
* **管道（Pipe）、及命名管道（named pipe）：**管道可用于具有亲缘关系进程间的通信，命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；
* 普通管道PIPE：通常有两种限制，一是单工，只能**单向传输**；二是只能在父子或者兄弟进程间使用.
* 流管道s\_pipe: 去除了第一种限制，为半双工，只能在父子或兄弟进程间使用，可以双向传输。
* 命名管道:name\_pipe：去除了第二种限制，可以在许多并不相关的进程之间进行通讯
* **信号（Signal）：**信号是比较复杂的通信方式，用于**通知接受进程有某种事件发生**，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，sigaction函数重新实现了signal函数、
* **报文（Message）、队列（消息队列）：**消息队列是**消息的链接表**，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
* **共享内存：**共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量配合使用，来实现进程间的**同步和通信**。
* **信号量（semaphore）：**信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
* **套接字（Socket）：**更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix 系统上：Linux和System V的变种都支持套接字。

**1）管道**

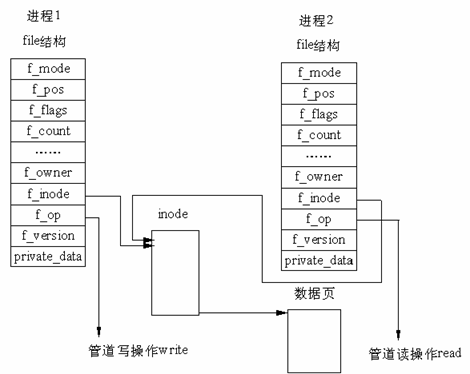
管道是由内核管理的一个缓冲区，相当于我们放入内存中的一个纸条。管道的一端连接一个进程的输出。这个进程会向管道中放入信息。管道的另一端连接一个进程的输入，这个进程取出被放入管道的信息。一个缓冲区不需要很大，它被设计成为**环形的数据结构，以便管道可以被循环利用**。当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。当管道被放满信息的时候，尝试放入信息的进程会等待，直到另一端的进程取出信息。当两个进程都终结的时候，管道也自动消失。



从原理上，管道利用**fork机制建立**，从而让两个进程可以连接到同一个PIPE上。最开始的时候，上面的两个箭头都连接在同一个进程Process 1上(连接在Process 1上的两个箭头)。当fork复制进程的时候，会将这两个连接也复制到新的进程(Process 2)。随后，每个进程关闭自己不需要的一个连接 (两个黑色的箭头被关闭; Process 1关闭从PIPE来的输入连接，Process 2关闭输出到PIPE的连接)，这样，剩下的红色连接就构成了如上图的PIPE



**实现细节：**在 Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的file结构和VFS的索引节点inode。通过将两个 file 结构指向同一个**临时的 VFS 索引节点**，而这个 VFS 索引节点又指向一个物理页面而实现的。如下图



有两个 file 数据结构，但它们**定义文件操作例程地址是不同的**，其中一个是向管道中写入数据的进程地址，而另一个是从管道中读出数据的进程地址。这样，用户程序的系统调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作。

**2）命名管道**

由于基于fork机制，所以管道只能用于父进程和子进程之间，或者拥有相同祖先的两个子进程之间 (有亲缘关系的进程之间)。为了解决这一问题，Linux提供了FIFO方式连接进程。FIFO又叫做命名管道(named PIPE)。

FIFO (First in, First out)为一种特殊的文件类型，它在文件系统中有对应的路径。当一个进程以读(r)的方式打开该文件，而另一个进程以写(w)的方式打开该文件，那么内核就会在这两个进程之间建立管道，所以FIFO实际上也由**内核管理**，不与硬盘打交道。之所以叫FIFO，是因为管道本质上是一个先进先出的队列数据结构，最早放入的数据被最先读出来，从而保证信息交流的顺序。FIFO只是借用了文件系统(file system,命名管道是一种特殊类型的文件，因为Linux中所有事物都是文件，它在文件系统中以文件名的形式存在。)来为管道命名。写模式的进程向FIFO文件中写入，而读模式的进程从FIFO文件中读出。当删除FIFO文件时，管道连接也随之消失。FIFO的好处在于我们可以**通过文件的路径来识别管道**，从而让没有亲缘关系的进程之间建立连接。

**FIFO读写规则：**

* 从FIFO中读取数据：如果一个进程为了从FIFO中读取数据而阻塞打开了FIFO，那么称该进程内的读操作为设置了阻塞标志的读操作
* 从FIFO中写入数据：如果一个进程为了向FIFO中写入数据而阻塞打开FIFO，那么称该进程内的写操作为设置了阻塞标志的写操作。

**3）信号**

信号是比较复杂的通信方式，用于**通知接受进程有某种事件发生**，除了用于进程间通信外，进程还可以发送信号给进程本身；Linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。

**信号本质：**

信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。

信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。信号机制经过POSIX实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。

**信号来源：**

信号事件的发生有两个来源：硬件来源(比如我们按下了键盘或者其它硬件故障)；软件来源，最常用发送信号的系统函数是kill, raise, alarm和setitimer以及sigqueue函数，软件来源还包括一些非法运算等操作

**4）信号量**

<http://www.cnblogs.com/biyeymyhjob/archive/2012/11/04/2753535.html>

信号量(semaphore)实际是一个整数，它的值由多个进程进行测试(test)和设置(set)。就每个进程所关心的测试和设置操作而言，这两个操作是不可中断的，或称**“原子”操作**，即一旦开始直到两个操作全部完成。测试和设置操作的结果是：信号量的当前值和设置值相加，其和或者是正或者为负。根据测试和设置操作的结果，一个进程可能必须睡眠，直到有另一个进程改变信号量的值。

**信号量可用来实现所谓的“临界区”的互斥使用**，临界区指同一时刻只能有一个进程执行其中代码的代码段。为了进一步理解信号量的使用，下面我们举例说明。

假设你有很多相互协作的进程，它们正在读或写一个数据文件中的记录。你可能希望严格协调对这个文件的存取，于是你使用初始值为1的信号量，在这个信号量上实施两个操作，首先测试并且给信号量的值减1，然后测试并给信号量的值加1。当第一个进程存取文件时，它把信号量的值减1，并获得成功，信号量的值现在变为0，这个进程可以继续执行并存取数据文件。但是，如果另外一个进程也希望存取这个文件，那么它也把信号量的值减1，结果是不能存取这个文件，因为信号量的值变为-1。这个进程将被挂起，直到第一个进程完成对数据文件的存取。当第一个进程完成对数据文件的存取，它将增加信号量的值，使它重新变为1，现在，等待的进程被唤醒，它对信号量的减1操作将获得成功。

信号量与已经介绍过的IPC机构（管道、FIFO以及消息列队）不同。它是一个计数器，用于多进程对共享数据对象的存取。为了获得共享资源，进程需要执行下列操作：

* 测试控制该资源的信号量。
* 若此信号量的值为正，则进程可以使用该资源。进程将信号量值减1，表示它使用了一个资源单位。
* 若此信号量的值为0，则进程进入睡眠状态，直至信号量值大于0。若进程被唤醒后，它返回至(第(1)步)

**5）消息队列**

消息队列也称为报文队列，消息队列是随内核持续的，只有在内核重起或显示删除一个消息队列时，该消息队列才会真正删除。系统中记录消息队列的数据结构struct ipc\_ids msg\_ids位于内核中，系统中所有消息队列都可以在结构msg\_ids中找到访问入口。

消息队列其实就是一个**消息的链表，每个消息队列有一个队列头**，称为struct msg\_queue，这个队列头描述了消息队列的key值，用户ID，组ID等信息，但它存于内核中而结构体struct msqid\_ds能够返回或设置消息队列的信息，这个结构体位于用户空间中，与msg\_queue结构相似消息队列允许一个或多个进程向它写入或读取消息，消息队列是消息的链表。

消息是按消息类型访问，进程必须指定消息类型来读取消息，同样，当向消息队列中写入消息时也必须给出消息的类型，如果读队列使用的消息类型为0，则读取队列中的第一条消息。内核空间的结构体msg\_queue描述了对应key值消息队列的情况，而对应于用户空间的msqid\_ds这个结构体，因此，可以操作msgid\_ds这个结构体来操作消息队列。

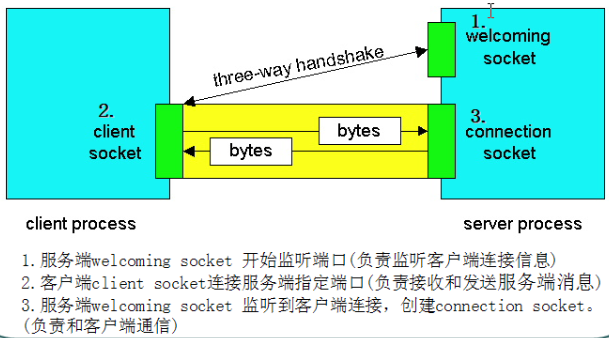
**6）共享内存**

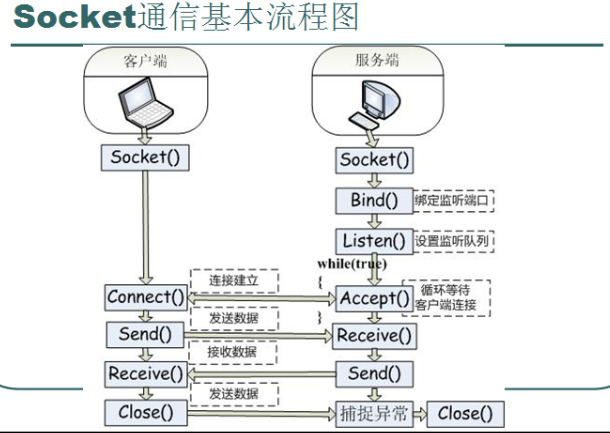
共享内存是运行在同一台机器上的**进程间通信最快的方式**，因为数据不需要在不同的进程间复制。通常由一个进程创建一块共享内存区，其余进程对这块内存区进行读写。共享内存往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

**7）套接字**

Socket类型：

* 流式Socket(STREAM)：是一种面向连接的Socekt，针对面向连接的TCP服务应用，安全，但是效率低；
* 数据报式Socket(DATAGAM)：是一种无连接的Socket，对应于无连接的UDP服务应用。不安(丢失，顺序混乱，在接受端要分析重排及要求重发)，但效率高。





**2、linux查看内存占用情况**

<http://www.cnblogs.com/xd502djj/archive/2011/03/01/1968041.html>

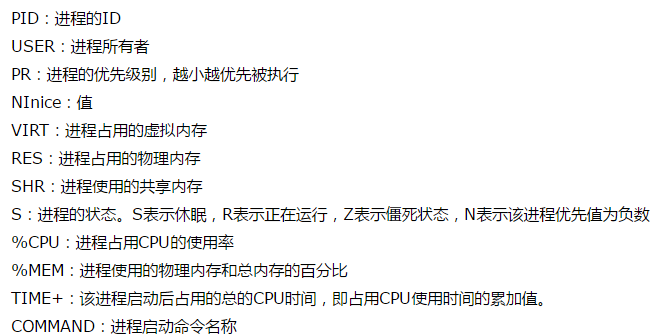
<http://blog.csdn.net/doupeihua/article/details/52062752>

**1）top**

在系统维护的过程中，随时可能有需要查看 CPU 使用率，并根据相应信息分析系统状况的需要。在 CentOS 中，可以通过 top 命令来查看 CPU 使用状况。运行 top 命令后，CPU 使用状态会以全屏的方式显示，并且会处在对话的模式 -- 用基于 top 的命令，可以控制显示方式等等。退出 top 的命令为 q （在 top 运行中敲 q 键一次）。

top命令是Linux下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于Windows的任务管理器。可以直接使用top命令后，查看%MEM的内容。可以选择按进程查看或者按用户查看，如想查看oracle用户的进程内存使用情况的话可以使用如下的命令：$ top -u oracle

**内容解释：**



top 的全屏对话模式可分为3部分：**系统信息栏、命令输入栏、进程列表栏**。

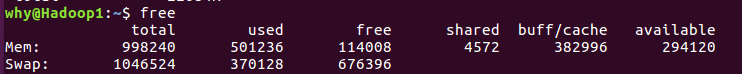
**2）pmap**

可以根据进程查看进程相关信息占用的内存情况，(进程号可以通过ps查看)如下所示

$ pmap -d 5647

**3）free**

free命令是一个快速查看内存使用情况的方法，它是对 /proc/meminfo 收集到的信息的一个概述。在Linux下查看内存我们一般用free命令：

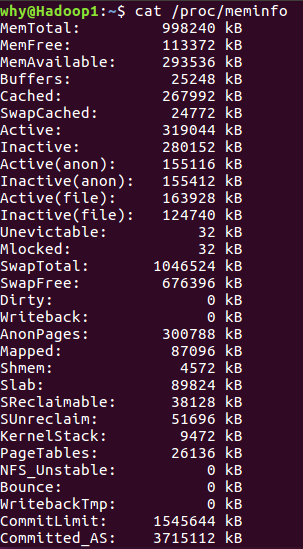


数值解释：

* total：总计物理内存的大小。
* used：已使用多大。
* free：可用有多少。
* Shared：多个进程共享的内存总额。
* Buffers/cached：磁盘缓存的大小

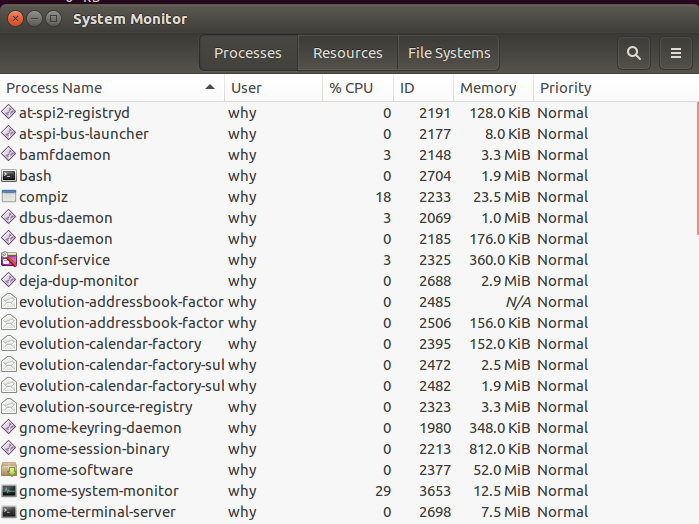
**4）/proc/meminfo**

查看RAM使用情况最简单的方法是通过/proc/meminfo。这个**动态更新的虚拟文件**实际上是许多其他内存相关工具(如：free / ps / top)等的组合显示。/proc/meminfo列出了所有你想了解的内存的使用情况。进程的内存使用信息也可以通过/proc/<pid>/statm 和 /proc/<pid>/status 来查看。



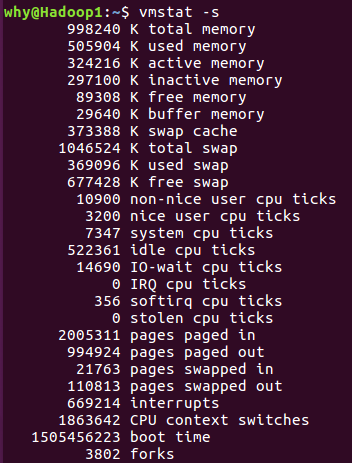
**5）gnome-system-monitor**

GNOME System Monitor 是一个显示最近一段时间内的CPU、内存、交换区及网络的使用情况的视图工具。它还提供了一种查看CPU及内存使用情况的方法。



**6）vmstat**

vmstat命令**显示实时的和平均的统计**，覆盖CPU、内存、I/O等内容。例如内存情况，不仅显示物理内存，也统计虚拟内存。



**7）memstat**

memstat是一个有效识别executable(s), process(es) and shared libraries使用虚拟内存情况的命令。给定一个进程ID，memstat可以列出这个进程相关的可执行文件、数据和共享库。

**8）atop**

atop命令是一个**终端环境的监控命令**。它显示的是各种系统资源（CPU, memory, network, I/O, kernel）的综合，并且在高负载的情况下进行了彩色标注。

**3、linux查询java进程**

ps 命令用于查看当前正在运行的进程

grep 是搜索

例如： ps -ef | grep java

表示查看所有进程里 CMD 是 java 的进程信息

ps -aux | grep java

-aux 显示所有状态

面试问题

<https://www.nowcoder.com/discuss/12476>

**六、计算机网络面试知识**

<https://www.nowcoder.com/discuss/1937>

<http://www.cnblogs.com/xymqx/p/4442465.html>

<https://zhuanlan.zhihu.com/p/24001696>

**1、在浏览器中输入www.baidu.com后执行的全部过程**

现在假设如果我们在客户端（客户端、浏览器中输入http://www.baidu.com，而baidu.com为要访问的服务器（服务器、，下面详细分析客户端为了访问服务器而执行的一系列关于协议的操作：（应用到的协议有：DNS、HTTP、TCP、ARP、

* 1、客户端浏览器通过**DNS解析到www.baidu.com的IP地址220.181.27.48**，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220.161.27.48，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，如服务器使用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，与服务器进行交换，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系应用层或者传输层的东西，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，这些都是由路由器来完成的工作，我不作过多的描述，无非就是通过查找路由表决定通过那个路径到达服务器。
* 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IP地址的**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。

**2、HTTP协议包括哪些请求？**

* GET：请求读取由URL所标志的信息；
* POST：给服务器添加信息（如注释、；
* PUT：在给定的URL下存储一个文档；
* DELETE：删除给定的URL所标志的资源；
* HEAD：类似于GET请求，不过返回的响应中没有具体内容，用于**获取报头**
* TRACE：发送一个**请求副本**，以**跟踪其处理进程**
* OPTIONS：返回所有可用的方法，检查服务器支持哪些方法
* CONNECT：用于ssl隧道的基于代理的请求

**3、HTTP中POST与GET的区别**

从原理性看：

* 根据HTTP规范，GET用于信息获取，而且应该是**安全和幂等**的
* 根据HTTP规范，POST请求表示可能修改服务器上资源的请求

从表面上看：

* GET请求的数据会附在URL后面，POST的**数据放在HTTP包体**
* POST安全性比GET安全性高

I.所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，**GET 请求一般不应产生副作用**。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

II.幂等的意味着**对同一URL的多个请求应该返回同样的结果**。

**4、TCP/IP中，每一层对应的协议**

* **网络层：**IP协议、ICMP协议（网络控制报文协议）、ARP协议（地址解析协议）、RARP协议（逆地址解析协议）
* **传输层：**UDP协议、TCP协议。
* **应用层：**FTP（文件传送协议）、Telenet（远程登录协议）、DNS（域名解析协议）、SMTP（邮件传送协议）、POP3协议（邮局协议）、HTTP协议（超文本传输协议）。

**5、TCP对应的协议和UDP对应的协议**

* TCP对应的协议：
* FTP：定义了文件传输协议，使用21端口。常说某某计算机开了FTP服务便是启动了文件传输服务。下载文件，上传主页，都要用到FTP服务。
* Telnet：它是一种用于**远程登陆的端口**，用户可以以自己的身份远程连接到计算机上，通过这种端口可以提供一种基于DOS模式下的通信服务。如以前的BBS是-纯字符界面的，支持BBS的服务器将23端口打开，对外提供服务。
* SMTP：定义了**简单邮件传送协议**，现在很多邮件服务器都用的是这个协议，用于发送邮件。如常见的免费邮件服务中用的就是这个邮件服务端口，所以在电子邮件设置-中常看到有这么SMTP端口设置这个栏，服务器开放的是25号端口。
* POP3：它是和SMTP对应，**POP3用于接收邮件**。通常情况下，POP3协议所用的是110端口。也是说，只要你有相应的使用POP3协议的程序（例如Fo-xmail或Outlook），就可以不以Web方式登陆进邮箱界面，直接用邮件程序就可以收到邮件（如是163邮箱就没有必要先进入网易网站，再进入自己的邮-箱来收信）。
* HTTP协议：是从Web服务器传输超文本到本地浏览器的**传送协议**。
* UDP对应的协议：
* SNMP：简单网络管理协议，使用161号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。
* TFTP(Trival File Transfer Protocal)，**简单文件传输协议**，该协议在熟知端口69上使用UDP服务。
* 两者均可使用
* DNS：用于域名解析服务，将域名地址转换为IP地址。DNS用的是53号端口

**6、SNMP使用UDP而不使用TCP协议的原因？**

SNMP定义为依赖UDP数据报服务的应用层协议。之所以UDP协议而不选择TCP协议，是因为UDP效率高，这样不会太多地加大网络负载。但是由于UDP不可靠，因此SNMP报文容易丢失。

**7、ARP是地址解析协议，简单语言解释一下工作原理**

* 首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示**IP地址和MAC地址之间的对应关系**。
* 当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP数据包，该数据包包括的内容有：源主机 IP地址，源主机MAC地址，目的主机的IP 地址。
* 当本网络的所有主机收到该**ARP数据包**时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先**从数据包中取出源主机的IP和MAC地址写入到ARP列表**中，如果已经存在，则覆盖，然后将自己的MAC地址写入ARP响应包中，告诉源主机自己是它想要找的MAC地址。
* 源主机收到ARP响应包后。将目的主机的IP和MAC地址写入ARP列表，并利用此信息发送数据。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。

**广播发送ARP请求，单播发送ARP响应。**

**8、各种协议名词概念**

* ICMP协议： 因特网控制报文协议。它是TCP/IP协议族的一个子协议，用于在IP主机、路由器之间传递控制消息。
* TFTP协议： 是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。
* HTTP协议： 超文本传输协议，是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。
* DHCP协议：动态主机配置协议，是一种让系统得以**连接到网络**上，并获取所需要的配置参数手段。
* NAT协议：**网络地址转换**属接入广域网(WAN)技术，是一种将私有（保留）地址转化为合法IP地址的转换技术
* DHCP协议：一个局域网的网络协议，使用UDP协议工作，用途：**给内部网络或网络服务供应商自动分配IP地址**，给用户或者内部网络管理员作为对所有计算机作中央管理的手段。

**9、TCP和UDP的区别？**

* TCP提供面向连接的、可靠的数据流传输，而UDP提供的是非面向连接的、不可靠的数据流传输。
* TCP传输单位称为TCP报文段，UDP传输单位称为用户数据报。
* TCP注重**数据安全性**，UDP数据**传输快**，因为不需要连接等待，少了许多操作，但是其安全性却一般

**10、TCP的三次握手过程？为什么会采用三次握手，若采用二次握手可以吗？**

建立连接的过程是利用客户服务器模式，假设主机A为客户端，主机B为服务器端。

* TCP的三次握手过程：主机A向B发送连接请求；主机B对收到的主机A的**报文段**进行确认；主机A再次对主机B的确认进行确认。
* 采用三次握手是为了防止失效的连接请求报文段突然又传送到主机B，因而产生错误。**失效的连接请求报文段是指：**主机A发出的连接请求没有收到主机B的确认，于是经过一段时间后，主机A又重新向主机B发送连接请求，且建立成功，顺序完成数据传输。考虑这样一种**特殊情况**，主机A第一次发送的连接请求并没有丢失，而是因为网络节点导致延迟达到主机B，主机B以为是主机A又发起的新连接，于是主机B同意连接，并向主机A发回确认，但是此时主机A根本不会理会，主机B就一直在等待主机A发送数据，导致主机B的资源浪费。
* 采用两次握手不行，原因就是上面说的实效的连接请求的特殊情况。

**11、TCP三次握手和四次挥手**

**1）三次握手**

TCP是面向连接的，无论哪一方向另一方发送数据之前，都必须先在双方之间建立一条连接。在TCP/IP协议中，TCP 协议提供可靠的连接服务，连接是通过三次握手进行初始化的。三次握手的**目的**是同步连接双方的序列号和确认号并交换 TCP窗口大小信息。

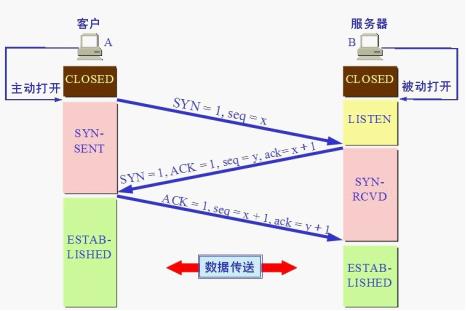
**步骤：**

* A 的 TCP 向 B 发出连接请求报文段，其首部中的同步位 SYN = 1，并选择序号 seq = x，表明传送数据时的第一个数据字节的序号是 x；
* B 的 TCP 收到**连接请求报文段**后，如同意，则发回确认(B 在确认报文段中应使 SYN = 1，使 ACK = 1，其**确认号**ack = x﹢1，自己选择的序号 seq = y)；
* A 收到此报文段后向 B 给出确认，其 ACK = 1，确认号 ack = y﹢1(A 的 TCP 通知上层应用进程，连接已经建立，B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立)。

或者：

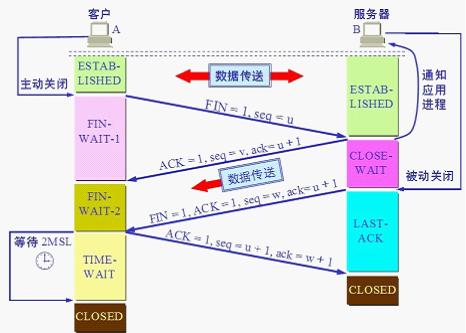
* 第一次握手：建立连接。客户端**发送连接请求报文段**，将SYN位置为1，Sequence Number为x;然后，客户端进入**SYN\_SENT**状态，等待服务器的确认；
* 第二次握手：服务器收到SYN报文段。服务器收到客户端的**SYN报文段**，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1);同时，自己自己还要发送**SYN请求信息**，将SYN位置为1，Sequence Number为y;服务器端将上述所有信息放到一个报文段(即**SYN+ACK报文段**)中，一并发送给客户端，此时服务器进入**SYN\_RECV**状态；
* 第三次握手：客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入**ESTABLISHED**状态，完成TCP三次握手。

**图释：**



**2）四次挥手**

**图释：**



**步骤：**

* 数据传输结束后，通信的双方都可释放连接.现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接(A 把连接释放报文段首部的 FIN = 1，其序号seq = u，等待 B 的确认)；
* B 发出确认，确认号 ack = u＋1，而这个报文段自己的序号 seq = v(**TCP 服务器进程通知高层应用进程**.从 A 到 B 这个方向的连接就释放了，**TCP 连接处于半关闭状态**.B 若发送数据，A 仍要接收)
* 若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接
* A 收到连接释放报文段后，必须发出确认，在确认报文段中 ACK = 1，确认号 ack=w﹢1，自己的序号 seq = u + 1

**或者：**

* 第一次分手：主机1(可以使客户端，也可以是服务器端)，设置Sequence Number和Acknowledgment Number，向主机2发送一个**FIN报文段**;此时，主机1进入FIN\_WAIT\_1状态;这表示主机1**没有数据**要发送给主机2了;
* 第二次分手：主机2收到了主机1发送的FIN报文段，向主机1回一个**ACK报文段**，Acknowledgment Number为Sequence Number加1;主机1进入**FIN\_WAIT\_2**状态;主机2告诉主机1，我也没有数据要发送了，可以进行关闭连接了;
* 第三次分手：主机2向主机1发送**FIN报文段**，请求关闭连接，同时主机2进入**CLOSE\_WAIT**状态;
* 第四次分手：主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入**TIME\_WAIT状态**;主机2收到主机1的ACK报文段以后，就关闭连接;此时，主机1**等待2MSL**后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

**注意：**

TCP 连接必须经过时间 **2MSL** 后才真正释放掉(2MSL 的时间的用意 --- 为了保证 **A 发送的最后一个 ACK 报文段能够到达；B.防止 “已失效的连接请求报文段”出现在本连接中**。A 在发送完最后一个 ACK 报文段后，再经过时间 2MSL，就可以使本连接持续的时间内所产生的所有报文段，都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段)。

**3）为什么要三次握手?**

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

**例子解释：**

"已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的**滞留**了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server**发出确认**，新的**连接就建立**了。由于现在client并没有发出建立连接的请求，因此不会**理睬**server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。"这就很明白了，防止了服务器端的一直等待而浪费资源。

**4）为什么要四次分手?**

那四次分手又是为何呢?**TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议**。TCP是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2， 它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文 段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN 报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。如果要正确的理解四次分手的原理，就需要了解四次分手过程中的**状态变化**。

* **FIN\_WAIT\_1:** 这个状态要好好解释一下，其实FIN\_WAIT\_1和FIN\_WAIT\_2状态的**真正含义**都是表示等待对方的FIN报文。而这两种状态的区别是：FIN\_WAIT\_1状态实际上是**当SOCKET在ESTABLISHED状态时， 它想主动关闭连接，向对方发送了FIN报文，此时该SOCKET即进入到FIN\_WAIT\_1状态**。而当对方回应ACK报文后，则进入到FIN\_WAIT\_2状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应ACK 报文，所以FIN\_WAIT\_1状态一般是比较难见到的，而FIN\_WAIT\_2状态还有时常常可以用netstat看到。(主动方)
* **FIN\_WAIT\_2：**上面已经详细解释了这种状态，实际上FIN\_WAIT\_2状态下的SOCKET，表示**半连接**，也即有一方要求close连接，但另外还告诉对方，我暂时还有点**数据**需要传送给你(ACK信息)，稍后再关闭连接。 (主动方)
* **CLOSE\_WAIT**：这种状态的含义其实是表示在**等待关闭**。怎么理解呢?当对方close一个SOCKET后发送FIN 报文给自己，你系统毫无疑问地会回应一个ACK报文给对方，此时则进入到**CLOSE\_WAIT状态**。接下来呢，实际上你真正需要考虑的事情是察看你**是否还有数据发送给对方**，如果没有的话，那么你也就可以 close这个 SOCKET，发送FIN报文给对方，也即关闭连接。所以你在CLOSE\_WAIT状态下，需要完成的事情是等待你去关闭连接。(被动方)
* **LAST\_ACK**: 这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。(被动方)
* **TIME\_WAIT**: 表示收到了对方的FIN报文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。 如果FINWAIT1状态下，收到了对方**同时带FIN标志和ACK标志的报文**时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。(主动方)
* **CLOSED**: 表示连接中断。

**12、交换机、路由器和网关概念及用途**

* **交换机**
* 计算机网络系统中，交换机是针对共享工作模式的弱点而推出的。交换机拥有一条高带宽的背部总线和内部交换矩阵。交换机的所有的端口都挂接在这条背部总线上，当控制电路收到数据包以后，**处理端口会查找内存中的地址对照表以确定目的MAC（网卡的硬件地址）的NIC（网卡）挂接在哪个端口上**，通过内部交换矩阵迅速将数据包传送到目的端口。目的MAC若不存在，交换机才广播到所有的端口，接收端口回应后交换机会“学习”新的地址，并把它添加入内部地址表中。
* 交换机工作于OSI参考模型的第二层，即**数据链路层**。交换机内部的CPU会在每个端口成功连接时，通过ARP协议学习它的MAC地址，保存成一张 ARP表。在今后的通讯中，发往该MAC地址的数据包将仅送往其对应的端口，而不是所有的端口。因此，交换机可用于划分数据链路层广播，即**冲突域**；但它不能划分网络层广播，即广播域。
* 交换机被广泛应用于二层网络交换，俗称“二层交换机”。
* 交换机的种类有：二层交换机、三层交换机、四层交换机、七层交换机分别工作在OSI七层模型中的第二层、第三层、第四层盒第七层，并因此而得名。
* **路由器**
* 路由器（Router）是一种计算机网络设备，提供了**路由与转送**两种重要机制，可以决定数据包从来源端到目的端所经过的**路由路径（host到host之间的传输路径）**，这个过程称为路由；将路由器输入端的数据包移送至适当的路由器输出端(在路由器内部进行)，这称为**转送**。路由工作在OSI模型的第三层——即网络层，例如网际协议。
* 路由器的一个作用是**连通不同的网络**，另一个作用是**选择信息传送的线路**。 路由器与交换器的差别，路由器是属于OSI第三层的产品，交换器是OSI第二层的产品(这里特指二层交换机)。
* **网关**
* 网关顾名思义就是连接两个网络的设备，区别于路由器（由于历史的原因，许多有关TCP/IP 的文献曾经把网络层使用的路由器（Router）称为网关，在今天很多局域网采用都是路由来接入网络，因此现在通常指的**网关就是路由器的IP**），经常在家庭中或者小型企业网络中使用，用于连接局域网和Internet。 网关也经常指把一种协议转成另一种协议的设备，比如语音网关。
* 在传统TCP/IP术语中，网络设备只分成两种，一种为**网关（gateway）**，另一种为**主机（host）**。网关能在网络间转递数据包，但主机不能转送数据包。在主机（又称终端系统，end system）中，数据包需经过TCP/IP四层协议处理，但是在网关（又称中介系统，intermediate system）只需要到达**网际层**（Internet layer），决定路径之后就可以转送。在当时，网关 （gateway）与路由器（router）还没有区别。
* 在现代网络术语中，网关（gateway）与路由器（router）的定义不同。网关（gateway）能在**不同协议间**移动数据，而路由器（router）是在**不同网络间**移动数据，相当于传统所说的IP网关（IP gateway）。
* **网关是连接两个网络的设备**，对于语音网关来说，他可以连接PSTN网络和以太网，这就相当于VOIP，把不同电话中的模拟信号通过网关而转换成数字信号，而且加入协议再去传输。在到了接收端的时候再通过网关还原成模拟的电话信号，最后才能在电话机上听到。
* 对于以太网中的网关只能转发三层以上数据包，这一点和路由是一样的。而不同的是**网关中并没有路由表，只能按照预先设定的不同网段来进行转发**。网关最重要的一点就是**端口映射**，在外网看来子网内用户只是外网的IP地址对应着不同的端口，这样看来就会保护子网内的用户（保护子网）

**13、IPv6与IPv4地址间的关系**

<http://blog.csdn.net/chenjhit/article/details/70224531>

**1）IPv6地址表示**

**32位IPv4地址被分割为4个 8位段**，其中每个8位段的数字大小在0~255之间，并且每个8位段之间使用英文符号句点“.” 来分开，因此有时也使用术语“点分十进制表示法”来专指IPv4地址的这种表示法。而**128位的IPv6地址则被分割成8个16位段**来表示，其中每个16位段书写为大小在0x0000~0xFFFF之间的十六进制的数宇表示，并且每个16位段之问使用英文符号冒号“∶”来分开，如3ffe∶1944∶0100∶000a∶0000∶00bc∶ 2500∶0d0b。

简化ipv6地址书写规则：

* 第一条规则：任何一个16位段中起始的0不必写出来；任何一个16位段如果少于4个十六进制的数字，就认为忽略书写的数字是起始的0；**注意的是**，只有起始的0才可以被忽略掉；末尾的0是不能忽略的，因为这样做会使16位分段变得不确定，你无法确切地判断所省略的0是在所写的数字之前还是在其之后。

如：ff02∶0000∶0000∶0000∶0000∶0000∶0000∶0005

简化为：ff02∶0∶ 0∶ 0∶ 0∶ 0∶ 0∶ 5

* 第二个规则：任何由全0组成的1个或多个16位段的单个连续的字符串都可以用一个双冒号“：：”来表示。利用这条规则，上面例子中的地址可以表示成如下格式 **ff02∷5**。**注意的是**，这条规则强调的是仅仅对于单个连续不间断的全 0字符串分段部分能够用一个双冒号“：：”来表示，在一个IPv6地址中**使用多于—个以上的双冒号会引起含混不清**。

如：2001∶0d02∶0000∶0000∶0014∶0000∶0000∶0095

简化：2001∶d02∷14∶ 0∶ 0∶ 95 或 2001∶d02∶ 0∶ 0∶ 14∷95，这两种简化方式均是正确的，都只使用了 一次双冒号。而 2001∶d02∷ 14∷95 是错误的，因为它中间的两个全0字符串的长度是含混不清的，从而无法确定它们的长度。

不像IPv4协议的前缀 (即地址的网络部分)可以通过点分十进制或十六进制地址掩码标识，或可以通过位计数 (bitcount)来标识，IPv6协议的前缀始终通过位计数的方式来标识。更确切地说，通过在IPv6地址后面加—个斜线 “ /” ，随 后再跟一个十进制的数字来标识一个IPv6地址的起始位有多少位是前缀位。举一个例子，下面这个地址的前缀就是起始的64位 **3ffe∶1944∶100∶ a∷bc∶ 2500∶d0b/64**。当读者需要书写一个**IP地址的前缀**时，也使用和IP叫地址一样的书写方式将所有的主机位设置为0。如：3ffe∶1944∶100∶ a∷/64。

—个由全0组成的IPv6地 址能够被简单地写成—个**双冒号**。两种实例：

* **缺省地址**：可表示为全 0的形式，并且它的前缀长度也是0，即 **∷/0**
* **未指定地址**：未指定地址使用在某些邻居发现协议过程中，一个未指定地址就像—个填充器，用来标识一个还未确定的实际IPv6地址。它与缺省地址的书写方式是有区别的，它们的**前缀长度不同 ∷/128**

除上述形式外，在实际应用中还可能遇到一种情况，如下所示：

* **短格式地址段**表示为：”2001:d02::14:0:0:95-25f4:55ed”（单个ipv6基本表示-不包含::的单个ipv6后半部分表示）
* **长格式地址段**表示为：”2001:d02::14:0:0:95-::25f4:55ed”（单个ipv6基本表示-单个ipv6基本表示）

所表示的是一组ipv6地址范围，此范围以第一个ipv6地址为起点至第二个ipv6地址或者补全的ipv6地址为终点。

**2）ipv6与ipv4之间的转换**

由于ipv4到ipv6之间有一个漫长的过渡过程，因此需要将**ipv6兼容ipv4或者映射为ipv4**。

* **兼容：**如果ipv4表示为“X.X.X.X”，那么对应的ipv6即为“::X.X.X.X”（高位补零）
* **映射：**如果ipv6表示为“::FFFF:X.X.X.X”（**33-128位为::FFFF**），这类情况下的ipv6将会被映射为ipv4，使用java.net.InetAddress包中的InetAddress.getByName(str)方法将自动实现映射。

**14、Http状态码**

<https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin>

* **消息1xx：**这一类型的状态码，代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束。由于 HTTP/1.0 协议中没有定义任何 1xx 状态码，所以除非在某些试验条件下，服务器禁止向此类客户端发送 1xx 响应。
* **成功2xx：**代表请求已成功被服务器接收、理解、并接受
* **重定向3xx：**代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的 Location 域中指明。
* **请求错误4xx：**代表了客户端看起来可能发生了错误，妨碍了服务器的处理。除非响应的是一个 HEAD 请求，否则服务器就应该返回一个解释当前错误状况的实体，以及这是临时的还是永久性的状况。这些状态码适用于任何请求方法。浏览器应当向用户显示任何包含在此类错误响应中的实体内容。
* **服务器错误5xx和6xx：**代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。除非这是一个HEAD 请求，否则服务器应当包含一个解释当前错误状态以及这个状况是临时的还是永久的解释信息实体。浏览器应当向用户展示任何在当前响应中被包含的实体。

基本状态码：

* 200：请求被正常处理
* 204：请求被受理但没有资源可以返回
* 206：客户端只是请求资源的一部分，服务器只对请求的部分资源执行GET方法，相应报文中通过Content-Range指定范围的资源。
* 301：永久性重定向
* 302：临时重定向
* 303：与302状态码有相似功能，只是它希望客户端在请求一个URI的时候，能通过GET方法重定向到另一个URI上
* 304：发送附带条件的请求时，条件不满足时返回，与重定向无关
* 307：临时重定向，与302类似，只是强制要求使用POST方法
* 400：请求报文语法有误，服务器无法识别（请求错误）
* 401：请求需要认证（身份验证）
* 403：请求的对应资源禁止被访问（禁止访问）
* 404：服务器无法找到对应资源（请求资源不存在）
* 500：服务器内部错误
* 503：服务器正忙

**15、详解http和https的作用与区别**

<http://www.cnblogs.com/qiangxia/p/5261813.html>

HTTPS（Secure Hypertext Transfer Protocol）安全超文本传输协议。它是一个安全通信通道，它基于HTTP开发，用于在客户计算机和服务器之间交换信息。它使用安全套接字层(SSL)进行信息交换，简单来说它是HTTP的 安全版。 它是由Netscape开发并内置于其浏览器中，用于对数据进行压缩和解压操作，并返回网络上传送回的结果。HTTPS实际上应用了Netscape的安全全套接字层（SSL）作为HTTP应用层的子层。（HTTPS使用端口443，而不是象HTTP那样使用端口80来和TCP/IP进行通信）SSL使用40 位关键字作为RC4流加密算法，这对于商业信息的加密是合适的。HTTPS和SSL支持使用X.509数字认证，如果需要的话用户可以确认发送者是谁。

**HTTPS和HTTP的区别：**

* https协议需要到ca申请证书，一般免费证书很少，需要交费。
* http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议。
* http和https使用的是完全不同的连接方式，用的**端口也不一样**，前者是80，后者是443。
* http的连接很简单，是无状态的。
* HTTPS协议是由SSL+HTTP协议构建的可进行**加密传输、身份认证的网络协议**，要比http协议安全。

**16、TCP通信存在的问题**

<http://www.cnblogs.com/orange1438/p/4693470.html>

**1）原因**

如果通信中发现缺少数据或者丢包，那么，最大的可能在于**程序发送的过程或者接收的过程出现问题**。例如服务器给客户端发大量数据，Send的频率很高，那么就有可能在Send时发生错误（原因可能是又多种，可能是程序处理逻辑问题，多线程同步问题，缓冲区溢出问题等等），如果没有对Send失败做处理重发数据，那么客户端收到的数据就会比理论应该收到的少，就会造成丢数据，**丢包**的现象。

这种现象，其实本质上来说不是丢包，也不是丢数据，只是因为程序处理有错误，导致有些数据没有成功地被socket发送出去。**常用的解决方法如下**：拆包、加包头、发送，组合包，如果客户端、服务端掉线，常采用心跳测试。

tcp是一个“流”的协议，一个完整的包可能会被TCP拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送，这就是所谓的**TCP粘包和拆包问题**。

**2）问题说明**

假设客户端分别发送数据包D1和D2给服务端，由于服务端一次性读取到的字节数是不确定的，所以可能存在以下**4种情况：**

* 服务端分2次读取到了两个独立的包，分别是D1、D2，没有粘包和拆包；
* 服务端一次性接收了两个包，D1和D2粘在一起了，被成为**TCP粘包**；
* 服务端分2次读取到了两个数据包，第一次读取到了完整的D1和D2包的部分内容,第二次读取到了D2包的剩余内容，这被称为**拆包**；
* 服务端分2次读取到了两个数据包，第一次读取到了部分D1，第二次读取D1剩余的部分和完整的D2包；
* 如果此时服务端TCP接收滑动窗非常小，而数据包D1和D2都很大，很有可能发送第五种可能，即服务端多次才能把D1和D2接收完全，期间多次发生拆包情况。（TCP接收滑动窗：是接收端的大小，随着流量大小而变化）

**3）解决策略**

由于底层的TCP无法理解上层的业务逻辑，所以在底层是无法确保数据包不被拆分和重组的，这个问题只能通过**上层的应用协议栈设计来解决**，根据业界的主流协议的解决方案，归纳如下：

* **消息定长（数据长度固定）**，例如每个报文的大小为固定长度200字节，如果不够，空位补**空格**；发送定长包。如果每个消息的大小都是一样的，那么在接收对等方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息；
* 在包尾增加回车换行符进行分割，例如FTP协议；**包尾加上\r\n标记**。FTP协议正是这么做的。但问题在于如果数据正文中也含有\r\n，则会误判为消息的边界；
* 将消息分为**消息头和消息体**，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思路是消息头的第一个字段用int来表示消息的总长度；（我之前linux C开发，就用的这种）。**包头加上包体长度**。包头是定长的4个字节，说明了包体的长度。接收对等方先接收包体长度，依据包体长度来接收包体。
* 更复杂的应用层协议；

**粘包问题：**

<http://www.cnblogs.com/QG-whz/p/5537447.html>

<http://www.cnblogs.com/qiaoconglovelife/p/5733247.html>

<http://www.cnblogs.com/kex1n/p/6502002.html>

TCP是一个基于**字节流的传输服务**，"流"意味着TCP所传输的数据是没有边界的。这不同于UDP提供基于消息的传输服务，其传输的数据是有边界的。TCP的发送方无法保证对等方每次接收到的是一个完整的数据包。粘包问题的**最本质原因**在与接收对等方无法分辨消息与消息之间的边界在哪。

* 定长包解决方案的缺点在于会导致增加网络的负担，无论每次发送的有效数据是多大，都得按照定长的数据长度进行发送。

**17、TCP的长连接和短连接**

<http://www.cnblogs.com/onlysun/p/4520553.html>

<http://blog.csdn.net/zxy987872674/article/details/52653101> （详细）

**1）长连接**

所谓长连接，指在一个TCP连接上可以连续发送多个数据包，在TCP连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接，一般需要自己做在线维持（不发生RST包和四次挥手）。

连接→数据传输→保持连接(心跳)→数据传输→保持连接(心跳)→……→关闭连接（一个TCP连接通道多个读写通信）；

这就要求长连接在没有数据通信时，定时发送数据包**(心跳)**，以维持连接状态；TCP保活功能，保活功能主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将应远等待客户端的数据，**保活功能就是试图在服务器端检测到这种半开放的连接**。

如果一个给定的连接在两小时内没有任何的动作，则服务器就向客户发一个**探测报文段**，客户主机必须处于以下4个状态之一：

* 客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。
* 客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在75秒后超时。服务器总共发送10个这样的探测 ，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。
* 客户主机崩溃并已经重新启动。服务器将收到一个**对其保活探测的响应**，这个响应是一个复位，使得服务器终止这个连接。
* 客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探查的响应。

**2）短连接**

短连接是指通信双方有数据交互时，就建立一个TCP连接，数据发送完成后，则断开此TCP连接（管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段）；

连接→数据传输→关闭连接；

**3）应用场景**

长连接多用于操作频繁（读写），点对点的通讯，而且连接数不能太多情况。每个TCP连接都需要三步握手，这需要**时间**，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，第二次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

而像WEB网站的http服务一般都用短链接（http1.0只支持短连接，1.1keep alive 带时间，操作次数限制的长连接），因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但**每个用户无需频繁操作情况下需用短连好**；

在长连接中一般是没有条件能够判断读写什么时候结束，所以必须要加长度报文头。读函数先是读取报文头的长度，再根据这个长度去读相应长度的报文。

**4）什么是“长连接”和“短连接”？（进一步解释）**

* **解释1：**
* 所谓长连接指建立SOCKET连接后不管是否使用都保持连接，但安全性较差；
* 所谓短连接指建立SOCKET连接后发送后接收完数据后马上断开连接，一般银行都使用短连接
* **解释2：**
* 长连接就是指在基于tcp的通讯中，一直保持连接，不管当前是否发送或者接收数据。
* 而短连接就是只有在有数据传输的时候才进行连接，客户－服务器通信/传输数据完毕就关闭连接。
* **解释3：**

长连接和短连接这个概念好像只有移动的CMPP协议中提到了，其他的地方没有看到过。

各网元之间共有两种连接方式：长连接和短连接。所谓长连接，指在一个TCP连接上可以连续发送多个数据包，在TCP连接保持期间，如果没有数据包发送，需要**双方发检测包以维持此连接**。短连接是指通信双方有数据交互时，就建立一个TCP连接，数据发送完成后，则断开此TCP连接，即每次TCP连接只完成一对 CMPP消息的发送。现阶段，要求ISMG之间必须采用长连接的通信方式，建议SP与ISMG之间采用长连接的通信方式。

* **解释4：**
* 短连接：比如http的，只是连接、请求、关闭，过程时间较短,服务器若是一段时间内没有收到请求即可关闭连接。
* 长连接：有些服务需要长时间连接到服务器，比如CMPP，一般需要自己做在线维持。

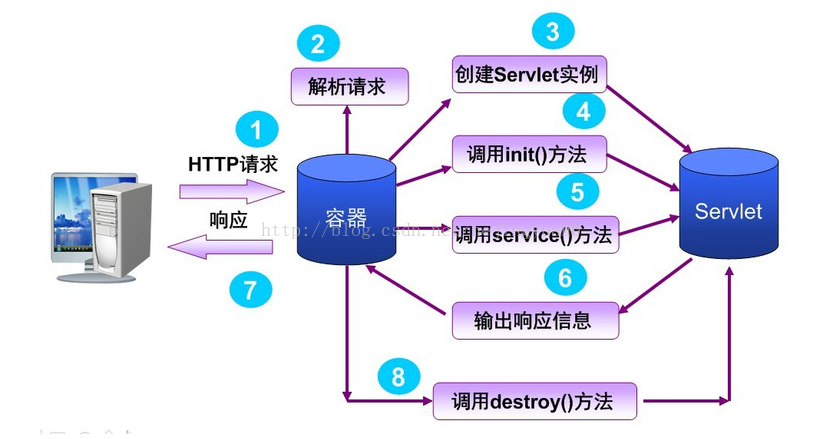
**七、Java Web**

**1、Servlet 生命周期、工作原理**

<http://www.cnblogs.com/xuekyo/archive/2013/02/24/2924072.html>

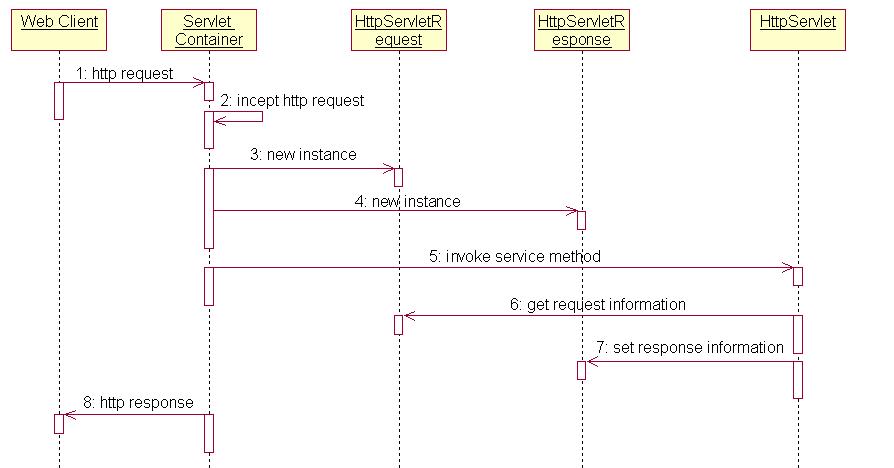
<http://blog.csdn.net/insistgogo/article/details/20788749>

* Servlet 生命周期：Servlet 加载与实例化---> 初始化 --->处理请求--->销毁。



* init（）：在Servlet的生命周期中，仅执行一次init()方法。它是在服务器装入Servlet时执行的，负责初始化Servlet对象。可以配置服务器，以在启动服务器或客户机首次访问Servlet时装入Servlet。无论有多少客户机访问Servlet，都不会重复执行init（）。
* service（）：它是Servlet的核心，负责响应客户的请求。每当一个客户请求一个HttpServlet对象，该对象的Service()方法就要调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在HttpServlet中已存在Service()方法。默认的服务功能是调用与HTTP请求的方法相应的do功能。
* destroy（）： 仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

**Tomcat 与 Servlet 是如何工作的：**



步骤：

1. Web Client 向Servlet容器（Tomcat）发出Http请求
2. Servlet容器接收Web Client的请求
3. Servlet容器创建一个HttpRequest对象，将Web Client请求的信息封装到这个对象中。
4. Servlet容器创建一个HttpResponse对象
5. Servlet容器调用**HttpServlet对象的service方法**，把HttpRequest对象与HttpResponse对象作为参数传给 HttpServlet 对象。
6. HttpServlet调用HttpRequest对象的有关方法，获取Http请求信息。
7. HttpServlet调用HttpResponse对象的有关方法，生成响应数据。
8. Servlet容器把HttpServlet的响应结果传给Web Client。

**Servlet工作原理：**

* 1、首先简单解释一下Servlet接收和响应客户请求的过程，首先客户发送一个请求，Servlet是调用**service()方法**对请求进行响应的，通过源代码可见，service()方法中对请求的方式进行了匹配，选择调用doGet、doPost等这些方法，然后再进入对应的方法中调用逻辑层的方法，实现对客户的响应。在Servlet接口和GenericServlet中是没有doGet（）、doPost（）等等这些方法的，HttpServlet中定义了这些方法，但是都是返回error信息，所以，我们每次定义一个Servlet的时候，都必须实现doGet或doPost等这些方法。
* 2、每一个自定义的Servlet都必须实现Servlet的接口，Servlet接口中定义了五个方法，其中比较重要的三个方法涉及到Servlet的**生命周期**，分别是上文提到的init()、service()、destroy()方法。GenericServlet是一个通用的，不特定于任何协议的Servlet,它实现了Servlet接口。而HttpServlet继承于GenericServlet，因此HttpServlet也实现了Servlet接口。所以我们定义Servlet的时候只需要继承HttpServlet即可。
* 3、Servlet接口和GenericServlet是不特定于任何协议的，而HttpServlet是特定于HTTP协议的类，所以HttpServlet中实现了service()方法，并将请求ServletRequest、ServletResponse 强转为HttpRequest 和 HttpResponse。

**创建Servlet对象的时机：**

* Servlet容器启动时：读取web.xml配置文件中的信息，构造指定的Servlet对象，创建ServletConfig对象，同时将ServletConfig对象作为参数来调用Servlet对象的init方法。
* 在Servlet容器启动后：客户首次向Servlet发出请求，Servlet容器会判断内存中是否存在指定的Servlet对象，如果没有则创建它，然后根据客户的请求创建HttpRequest、HttpResponse对象，从而调用Servlet 对象的service方法。
* Servlet Servlet容器在启动时自动创建Servlet，这是由在web.xml文件中为Servlet设置的<load-on-startup>属性决定的。从中我们也能看到同一个类型的Servlet对象在Servlet容器中以单例的形式存在。

**2、共享数据方法**

同一用户的不同页面共享数据方法总结

* **Cookie：**

服务器再客户端保存用户的信息，比如登录名、密码等，就是cookie

cookie一般保存在客户端的C:\Document and Setting

**功能：**

a. 保存用户名、密码，在一定时间不用重新登录

b. 记录用户访问网站的喜好（如有无背景音乐、网页背景颜色是什么）

c. 网站的个性化，比如定制网站的服务、内容

* **sendRedirect跳转**

通过该方法将信息传给下一页面，比如sendRedirect("welcome?uname=shunping")的形式

优点：传送信息速度比较快

缺点：只能传送字符串，并不能传送一个对象

**注意：**

a. welcome代表要跳转的那个servlet的url

b. servlet url 名和值间有"?"号

c. 如要传递两个以上的值，它们之间要用"&"号分开，比如：

sendRedierct("welcome?uname=shunping&upass=ok")

d. 如果传递的是中文，将得到乱码

* 隐藏表单：常见且简单的

形如：

<form action=login>

<input type=hidden name=a value=b>

</form>

* session：同一用户不同页面共享session中的数据(慎用)

当用户打开浏览器，访问某个网站时，服务器就会在服务器内存为该浏览器分配一个空间，该空间被这个浏览器独占这个空间就是session空间，该空间中的数据默认存在时间为30min，也可修改。

**功能：**

a. 网上商城的购物车

b. 保存登录用户的信息

c. 将某些数据放入到session中，供同一用户的各个页面使用

d. 防止用户非法登录到某个页面

.....

**3、cookie与session比较**

* 存在位置：cookie保存在客户端，session保存在服务器端
* 安全性：cookie的安全性比session弱
* 网络传输量：cookie通过网络在客户端与服务器传输，而session保存在服务器端，不需要传输
* 生命周期：cookie的周期是累积的，而session是间隔的；另外关机会造成session周期的结束，而cookie不变

**4、jsp工作原理**

<http://blog.csdn.net/insistgogo/article/details/20788749>



当客户端浏览器向服务器请求一个 JSP 页面时，服务器收到该请求后，首先检查所请求的这个JSP 文件内容 ( 代码 ) 是否已经被更新，或者是否是 JSP 文件创建后的第一次被访问：

* --如果是，那么，这个 JSP 文件就会在服务器端的 JSP 引擎作用下转化为一个 Servlet 类的 Java 源代码文件。紧接着，这个 Servlet 类会在 Java 编译器的作用下被编译成一个字节码文件，并装载到 jvm 解释执行。剩下的就等同于 Servlet 的处理过程了。
* --如果被请求的 JSP 文件内容 ( 代码 ) 没有被修改，那么它的处理过程也等同于一个 Servlet 的处理过程。即直接由服务器检索出与之对应的 Servlet 实例来处理。

需要注意的是，JSP 文件不是在服务器启动的时候转换成 Servlet 类的。而是在被客户端访问的时候才可能发生转换的 ( 如 JSP 文件内容没有被更新等，就不再发生 Servlet 转换 )。

**5、Servlet 与 JSP**

<http://www.blogjava.net/fancydeepin/archive/2013/09/30/fan_servlet.html>

JSP 本质是一个 Servlet，它的运行也需要容器的支持。在 JSP 和 Servlet 文件中都可以编写 Java 和 HTML 代码，不同的是，Servlet 虽然也可以动态的生成页面内容，但更加偏向于逻辑的控制。

JSP 最终被转换成 Servlet 在 jvm 中解释执行，在 JSP 中虽然也可以编写 Java 代码，但它更加偏向于页面视图的展现。在 MVC 架构模式中，就 JSP 和 Servlet 而言，C 通常由 Servlet 充当，V 通常由 JSP 来充当。

**6、JSP四大域范围**

* **page域：**只能在当前jsp页面中使用。作用于当前页面，当前页面可见。（当前页面）
* **request域：**只能在同一个请求中使用。作用于请求，转发间请求共享。（转发）
* **session域：**只能在同一个会话（session对象）中使用。作用于整个Web应用，单个用户之间共享（私有的）
* **context域：**只能在同一个web应用中使用。作用于整个Web应用，多个用户之间共享（全局的）

**7、JSP 9大内置对象**

* **out（JspWriter）：**作用是用来向客户端输出。out对象是通过调用pageContext对象的getOut方法返回的，其作用和用法与ServletResponse.getWriter方法返回的PrintWriter对象非常相似。JSP页面中的out隐式对象的类型为JspWriter，JspWriter相当于一种带缓存功能的PrintWriter，设置JSP页面的page指令的buffer属性可以调整它的缓存大小，甚至可以关闭它的缓存。
* **request（HttpServletRequest）：**接收客户端的http请求，类似于servlet中的 HttpServletRequest。客户端的请求信息被封装在request对象中，通过它才能了解到客户端的需求，然后做出响应。它是HttpServletRequest类的实例。request对象具有请求域，即完成客户端的请求之前，该对象一直有效。
* **response（HttpServletResponse）：**封装jsp产生的回应，类似于servlet中的 HttpServletResponse。response对象包含了响应客户请求的有关信息，但在JSP中很少直接用到它。它是HttpServletResponse类的实例。response对象具有页面作用域，及访问一个页面时，该页面内的response对象只能对这次访问有效，其他的response对象对当前页面无效。

常用方法：

* String getCharacterEncoding()：返回响应用的是何种字符编码
* void setContentType()：设置响应的MIME类型，一般为"text/html, charset=UTF-8"
* PrintWriter getWriter()：返回可以向客户端输出字符的一个对象("注意比较：PrintWriter与内置out对象的区别，PrintWriter对象的其输出总是提前于内置out对象，或者在out中手动flush")
* sendRedirect(loaction)：重新定向客户端的请求
* **session（HttpSession）：**用于保存用户信息，跟踪用户行为，被浏览器独占的某一内存空间。
* session表示客户端与服务器的一次会话，Web中的session指的是用户在浏览某个网站时，从进入网站到浏览器关闭所经过的这段时间，也就是用户浏览这个网站所花费的时间。
* 在服务器的内存中，为不同的用户保存着不同的session。
* session对象是一个JSP内置的对象。
* session对象在第一个JSP页面被装载是自动创建，完成会话期管理。
* 从一个客户打开浏览器并连接到服务器开始，到客户关闭浏览器离开这个服务器结束，被称为一个会话。
* 当一个客户访问一个服务器时，肯能会在服务器的几个页面之间切换，服务器应当通过某种办法知道这是一个客户，就需要session对象。
* session对象是HttpSession类的实例。
* **application（servletContext）：**多个用户**共享该对象**，可以做计数器，相当于servlet的ServletContext。
* application对象实现了用户间数据的共享，可存放全局变量。
* application开始于服务器的启动，终止于服务器的关闭。
* 在用户的前后连接或不同用户之间的连接中，可以对application对象的同一属性进行操作。
* 在任何地方对application对象属性的操作，都将影响到其他用户对此的访问。
* 服务器的启动和关闭决定了application对象的生命。
* application对象是ServletContext类的实例。
* **pageContext：**代表jsp页面的上下文。pageContext对象是JSP技术中最重要的一个对象，它代表JSP页面的运行环境，这个对象不仅封装了对其它8大隐式对象的引用，它自身还是一个域对象(三个域对象：ServletContext、Session、Request)，这个域对象的生命周期最短，作用域最小，他的作用域就是当前的jsp页面，当然它可以用来保存数据。并且，这个对象还封装了web开发中经常涉及到的一些常用操作，例如引入和跳转其它资源、检索其它域对象中的属性等。
* **exception：**代表运行时的一个异常。Exception对象是一个异常对象，当一个页面在运行过程中发生了异常，就产生这个对象。如果一个jsp页面要应用此对象，就必须报isErrorPage设为true，否则无法编译。它实际上是java.lang.Throwable的对象。页面要想在有异常时得到处理，就要指定异常处理页面<% page errorPage="exception.jsp"%>
* **page：**代表jsp这个实例本身（使用较少），类似于servlet中的this。page对象就是指向当前jsp页面本身，有点像类中的this指针，它是java.lang.Object类的实例。常用方法就是Object类的成员方法。
* **config（servletConfig）：**代表jsp对应的servlet的配置，可以得到web.xml中的参数，类似servlet的ServletConfig。Config对象是在一个Servlet初始化时，JSP引擎向它传递信息用的，此信息包括Servlet初始化时所要用到的参数（通过属性名和属性值构成）以及服务器的有关信息（通过传递一个ServletContext对象）

**8、CGI与Servlet对比**

公共网关接口（Common Gateway Interface，CGI）脚本是生成动态内容的主要技术。虽然使用得非常广泛，但CGI脚本技术有很多的缺陷，这包括**平台相关性和缺乏可扩展性**。为了避免这些局限性，Java Servlet技术因应而生，它能够以一种可移植的方法来提供动态的、面向用户的内容。处理用户请求。

* **对比一：**当用户浏览器发出一个Http/CGI的请求，或者说调用一个CGI程序的时候，服务器端就要新启用一个进程(而且是每次都要调用)，调用CGI程序越多(特别是访问量高的时候)，就要消耗系统越多的处理时间，只剩下越来越少的系统资源，对于用户来说，只能是漫长的等待服务器端的返回页面了，这对于电子商务激烈发展的今天来说，不能不说是一种技术上的遗憾。

而Servlet充分发挥了服务器端的资源并高效的利用。每次调用Servlet时并不是新启用一个进程，而是在一个Web服务器的进程中**共享和分离线程**，而线程最大的好处在于可以共享一个数据源，使系统资源被有效利用。故servlet不是线程安全的，单实例多线程的。

* **对比二：**传统的CGI程序，不具备平台无关性特征，系统环境发生变化，CGI程序就要瘫痪，而Servlet具备Java的平台无关性，在系统开发过程中保持了系统的可扩展性、高效性。
* **对比三：**传统技术中，一般大都为二层的系统架构，即Web服务器+数据库服务器，导致网站访问量大的时候，无法克服CGI程序与数据库建立连接时速度慢的瓶颈，从而死机、数据库死锁现象频繁发生。而我们的Servlet有连接池的概念，它可以利用多线程的优点，在系统缓存中事先建立好若干与数据库的连接，到时候若想和数据库打交道可以随时跟系统"要"一个连接即可，反应速度可想而知。

**9、Servlet如何同时处理多个请求？**

Servlet采用多线程来处理多个请求的同时访问。Servlet容器通过线程池来管理维护服务请求。所谓线程池，相当于**数据库连接池**，实际上是等待执行代码的一组线程，叫做工作者线程。Servlet容器通过一个调度线程来管理工作者线程。

* 当容器收到一个**Servlet的访问请求**，调度者线程就从线程池中选出一个工作者线程，将用户请求传递给该线程，然后由**该线程处理Servlet的service()方法**；
* 当这个线程在执行的时候，容器收到一个新的请求，调度者线程再次从线程池中选出一个新的工作者线程；
* 当容器同时收到对同一个Servlet的多个请求时，那么Servlet的service方法将在多线程中并发执行。

**注：**

* Servlet容器默认采用单实例多线程的方式来处理请求。这样减少了产生Servlet实例的开销，提升了对请求的响应时间；
* 对于Tomcat容器来讲，可以在其server.xml中通过<Connector>中设置线程池中的线程数目。

**10、如何开发线程安全的Servlet？**

Servlet容器采用多线程来处理请求，提高性能的同时也造成了**线程安全问题**。要开发线程安全的Servlet应该从一下几个方面进行：

* **变量的线程安全：**多线程并不共享局部变量，所以我们要尽可能的在Servlet中使用局部变量；
* **代码块的线程安全：**使用同步块Synchronized，防止可能调用的代码块；但是要注意的是，要**尽可能得缩小同步代码的方范围**，不要在service方法和响应方法上直接使用同步，这会严重影响性能。
* **属性的线程安全**：ServletContext，HttpSession，ServletRequest对象中属性；
* **使用同步集合：**使用**Vector**代替ArrayList，使用HashTable代替HashMap；
* 不要在Servlet中创建自己的线程来完成某个功能； Servlet本身就是多线程的，如果再创建新的线程，将会导致线程执行复杂化，出现线程安全问题；
* 在多个Servlet中，**对外部对象**，比如：文件；进行修改操作一定要加锁，做到**互斥访问**；

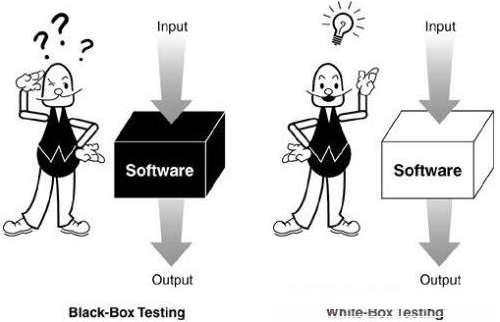
**八、测试相关知识**

**1、白盒测试与黑盒测试**

<http://www.cnblogs.com/Bonnieh/p/5845728.html>

<http://www.51testing.com/html/51/n-3714551.html>

黑盒测试就当整个程序是个黑盒子，我们看不到它里面做了些什么事情，只能通过输入输出看是否能得到我们所需的来测试。而白盒测试可以当盒子是透明的，里面的一切我们都看的清楚，从而我们可以通过去测内部结构来测试。



**1）黑盒测试**

黑盒测试又称为功能测试、数据驱动测试或基于规格说明书的测试，是一种从用户观点出发的测试。测试人员一般把被测程序当作一个黑盒子。

黑盒测试主要测到的**错误类型**有：不正确或遗漏的功能；接口、界面错误；性能错误；数据结构或外部数据访问错误；初始化或终止条件错误等等。

常用的黑盒**测试方法**有：等价类划分法；边界值分析法；因果图法；场景法；正交实验设计法；判定表驱动分析法；错误推测法；功能图分析法。

* **等价类划分法**：等价类分为有效等价类和无效等价类，其中，有效等价类是指对于程序的规格说明来说是合理的，有意义的输入数据构成的集合；而无效等价类是指对于程序的规格说明来说是不合理的，没有意义的输入数据构成的集合。

确定等价类的原则：

* 在输入条件规定了取值范围或值的个数的情况下，则可以确定一个有效等价类和两个无效等价类，比如若规定的输入范围是1<x<10的所有数，则有效的是1~10内的任意数，而无效的是 -∞~0.9999 和10.00001~+∞;
* 在输入条件规定了输入值的集合或者规定了“必须如何”的条件的情况下，则可以确立一个有效等价类和一个无效等价类；
* 在输入条件是一个布尔量的情况下，可以确定一个有效的等价类和一个无效的等价类；
* 在规定了输入数据的一组值（假定n个），并且程序要对每一个输入值分别处理的情况下，可以确立n个有效等价类和一个无效等价类；
* 在规定了输入数据必须遵守的规则的情况下，可以确立一个有效等价类（符合规则）和若干个无效等价类（**从不同角度违反规则**）；
* 在确知已划分的等价类中各元素在程序处理中的方式不同的情况下，则应再将改等价类进一步的划分为更小的等价类。
* **边界值分析法**：根据经验法则，大量的错误是发生在输入或输出范围的**边界上**的，而不是发生在输入输出范围的内部。因此针对各种边界情况涉及测试用例，可以查出更多的错误。而使用边界值的分析方法涉及测试用例，首先应确定边界情况。通常输入和输出等价类的边界，应当选取正好等于，刚刚大于或者刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值作为测试数据。

边界值选择的原则：

* 如果输入条件规定了值的范围，则应取刚达到这个范围的边界的值，以及刚刚超越这个范围的边界的值作为测试输入数据。 例如，如果程序的规格说明中规定：“重量在10公斤至50公斤范围内的邮件，其邮费计算公式为……”。作为测试用例，我们应取10及50，还应取10.01,49.99,9.99及50.01等；
* 如果输入条件**规定了值的个数**，则用最大个数，最小个数，比最小个数小一，和比最大个数大一的数作为测试数据。例如，一个输入文件应包括1~255个记录，则测试用例可取1和255，还应取0及256等。
* 将前面两个原则应用于输出条件，即涉及测试用例使输出值达到边界值及其左右的值。
* 如果程序的规格说明给出的输入域或输出域是**有序集合**，则应选取集合的第一个元素和最后一个元素作为测试用例。
* 如果程序中使用了一个内部数据结构，则应当选择这个内部数据结构的边界上的值作为测试用例数据；
* 分析规格说明，找出其他可能的边界条件
* **因果图法**：因果图是一种利用图解法分析输入的各种组合情况，从而设计测试用例的方法，它适合于检查程序输入条件的各种组合情况。

利用因果图生成测试用例的基本步骤有：

1. 分析软件规格说明描述中哪些是原因（输入，包括前提条件，操作过程），哪些是结果（输出），并给每个原因和结果赋予一个标识符；

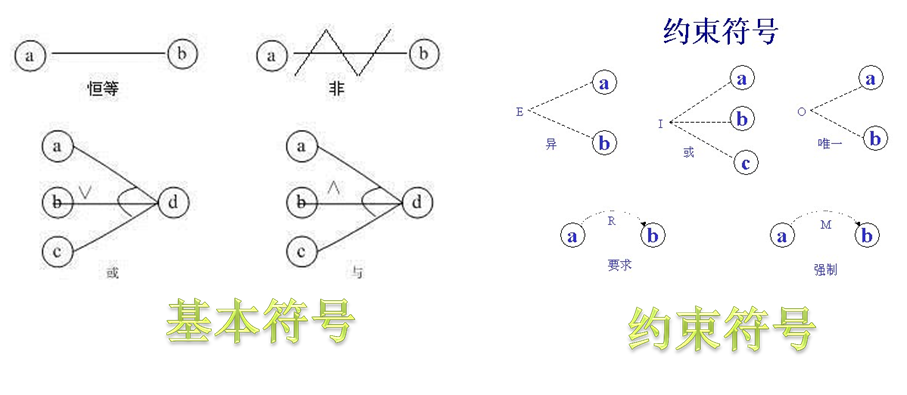
2. 分析软件规格说明描述的语义，找出原因和结果之间，原因和原因之间的关系，根据这些关系，画出**因果图**；

3. 在因果图上用一些记号表明约束或限制条件；

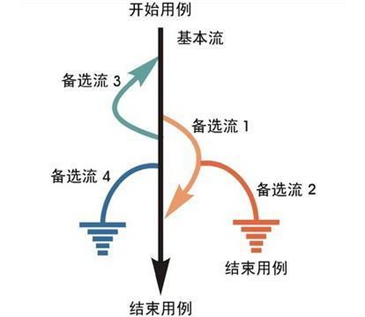
4. 把因果图转换为判定表；

5. 把判定表的每一列拿出来作为依据，设计测试用例。

因果图的基本符号和约束符号：



* **场景法**：场景法一般包含基本流和备用流，从一个流程开始，通过描述经过的路径来确定的过程，经过遍历所有的基本流和备用流来完成整个场景。其中，基本流就是正常的，正确场景；备选流一般指中断操作的。



* **正交实验设计法**：使用正交试验设计法首先要知道正交表，正交表是研究多因素多水平的一种设计方法，它是格局正交性从全面试验中挑选出部分有代表性的点进行试验，这些有代表性的点具备了“均匀分散，齐整可比”的特点，正交试验设计是一种基于正交表的、高效率、快速、经济的试验设计方法。正交表由三个成分构成，Runs：正交表的行数，即实验的次数；Factors：正交表的列数，即因素数；Levels：水平数，任何单个因素能够取得的值的最大个数。 正交表的表现形式是： L行数 (水平数因素数) Lruns（levels^factors ）。

用正交表设计测试用例的步骤：

1. 有哪些因素（变量）；

2. 每个因素有那几个水平（变量的取值）：用等价类划分出来的；

3. 选择一个**合适的正交表**；

4. 把变量的值映射到表中；

5. 把每一行的各因素水平的组合作为一个测试用例；

6. 加上你认为可以且没有在表中出现的组合。

如何选择正交表呢？

取行数最少的一个，情况分三种：

1. 因素数（变量）、水平数（变量值）相符；

2. 因素数不相同: 取因素数最接近但略大的实际值的表；

3. 水平数不相同: 有五个因素（变量）A、B、C、D和E。两个因素有两个水平（变量的取值）、两个因素有三个水平，一个因素有六个水平。行数取最少的一个（ 行数取最少的一个（L49(78)、 L18（3661）

* **判定表驱动分析法**：判定表是分析和表达多逻辑条件下执行不同操作的情况的工具。判定表由以下部分构成：

1. **条件桩（Condition Stub）**：列出了问题的所有条件。通常认为列出的条件的次序无关紧要；

2. **动作桩（Action Stub）**：列出了问题规定可能采取的操作。这些操作的排列顺序没有约束；

3. **条件项（Condition Entry）**：列出针对它左列条件的取值在所有可能情况下的真假值；

4. **动作项（Action Entry）**：列出在条件项的各种取值情况下应该采取的动作

* **错误推测法**：错误推测法是基于经验和直觉推测程序中所有可能存在的各种错误, 从而有针对性的设计测试用例的方法。错误推测方法的基本思想: 列举出程序中所有可能有的错误和容易发生错误的特殊情况, 根据它们选择测试用例。

最后，对于黑盒测试的**综合策略**是：首先，考虑边界值分析法，边界值分析法是在任何情况下都必须使用的方法，因为经验表明这种方法设计的测试用例发现程序错误的能力最强。必要时用等价类划分方法补充一些测试用例。其次，可以用错误推测法再追加一些测试用例。这时，可以对照程序逻辑，检查已设计出的测试用例的逻辑覆盖率，如果没有达到要求的覆盖标准，应当再补充足够的测试用例；如果程序的功能说明中含有输入条件的组合情况，则一开始就可以选用因果图法。

**2）白盒测试**

为什么要考虑白盒测试呢？举个栗子说明，设x=0,y=5, 程序设计z=y/x, 如果只考虑输入输出的黑盒测试，我们看不出程序内部的逻辑错误(分母不能为零)，即使我们使用黑盒测试的测试用例覆盖的再全也发现不了这个问题，这个时候就要考虑使用白盒测试了。

什么是白盒测试呢？白盒测试也称为**结构测试或逻辑驱动测试，是针对被测单元内部是如何进行工作的测试**。它根据程序的控制结构设计测试用例，主要用于软件或程序验证。白盒测试法检查程序内部逻辑结构，对所有的逻辑路径进行测试，是一种穷举路径的测试方法，但即使每条路径都测试过了，但仍然有可能存在错误。因为：穷举路径测试无法检查出程序本身是否违反了设计规范，即程序是否是一个错误的程序；穷举路径测试不可能检查出程序因为遗漏路径而出错；穷举路径测试发现不了一些与数据相关的错误。

白盒测试需要**遵循的原则**有：

* 保证一个模块中的所有独立路径至少被测试一次；
* 所有逻辑值均需要测试真（true）和假（false）；两种情况；
* 检查程序的内部数据结构，保证其结构的有效性；
* 在上下边界及可操作范围内运行所有循环。

白盒测试方法有：静态测试&动态测试

* **静态测试**：静态测试是不用运行程序的测试，包括代码检查、静态结构分析、代码质量度量、文档测试等等，它可以由人工进行，充分发挥人的逻辑思维优势，也可以借助软件工具（Fxcop）自动进行。
* **动态测试**：动态测试则需要执行代码，也是我们用的最多的一种测试，通过运行程序找到问题，包括功能确认与接口测试、覆盖率分析、性能分析、内存分析等

单元测试、代码检查、同行评审、技术评审

要检验测试的效果，就要去看**测试的覆盖率**，当覆盖率越大，说明测试成效越好，软件的缺陷越少。但覆盖率不是目标（质量才是目标），而只是一种手段。测试覆盖率包括功能点覆盖率和结构覆盖率，其中，功能点覆盖率大致用于表示软件已经实现的功能与软件需要实现的功能之间的比例关系；而结构覆盖率包括语句覆盖率、分支覆盖率、循环覆盖率和路径覆盖率等等

而逻辑覆盖法中根据覆盖目标的不同和覆盖源程序语句的详尽程度，逻辑覆盖又可分为：语句覆盖；判定覆盖；条件覆盖；判定/条件覆盖；组合覆盖；路径覆盖，且这些覆盖程度越往后越全。

**3）灰盒测试**

灰盒测试更像是白盒测试和黑盒测试的混合测试，现阶段对灰盒测试没有更明确的定义，但更多的时候，我们的测试做的就是灰盒测试，即既会做黑盒测试又会做白盒测试。

**九、其他**

**1、推荐算法知识**

<http://www.cnblogs.com/wentingtu/archive/2011/12/16/2289926.html>

**1）推荐引擎工作原理**

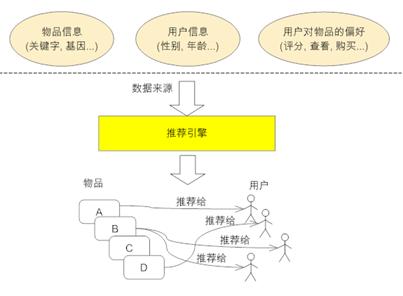


图 1 给出了推荐引擎的工作原理图，这里先将推荐引擎看作黑盒，它接受的输入是推荐的数据源，一般情况下，推荐引擎所需要的**数据源**包括：

* 要推荐物品或内容的元数据，例如关键字，基因描述等；
* 系统用户的基本信息，例如性别，年龄等
* 用户对物品或者信息的偏好，根据应用本身的不同，可能包括用户对物品的评分，用户查看物品的记录，用户的购买记录等。其实这些**用户的偏好信息可以分为两类：**
* 显式的用户反馈：这类是用户在网站上自然浏览或者使用网站以外，显式的提供反馈信息，例如用户对物品的评分，或者对物品的评论。
* 隐式的用户反馈：这类是用户在使用网站是产生的数据，隐式的反应了用户对物品的喜好，例如用户购买了某物品，用户查看了某物品的信息等等。
* 特点分析：显式的用户反馈能准确的反应用户对物品的真实喜好，但需要用户付出额外的代价，而隐式的用户行为，通过一些分析和处理，也能反映用户的喜好，只是数据不是很精确，有些行为的分析存在较大的噪音。但只要选择正确的行为特征，隐式的用户反馈也能得到很好的效果，只是行为特征的选择可能在不同的应用中有很大的不同，例如在电子商务的网站上，购买行为其实就是一个能很好表现用户喜好的隐式反馈。

推荐引擎根据不同的推荐机制可能用到数据源中的一部分，然后根据这些数据，分析出一定的规则或者直接对用户对其他物品的喜好进行预测计算。这样推荐引擎可以在用户进入的时候给他推荐他可能感兴趣的物品。

**2）推荐引擎分类**

推荐引擎的分类可以根据很多指标进行分类，如下：

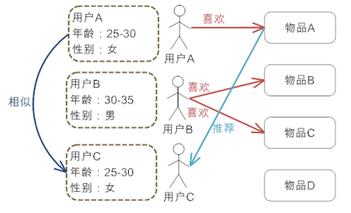
* **推荐引擎是不是为不同的用户推荐不同的数据**：根据这个指标，推荐引擎可以分为基于大众行为的推荐引擎和个性化推荐引擎。
* 大众行为的推荐引擎，对每个用户都给出同样的推荐，这些推荐可以是静态的由系统管理员人工设定的，或者基于系统所有用户的反馈统计计算出的当下比较流行的物品。
* 个性化推荐引擎，对不同的用户，根据他们的口味和喜好给出更加精确的推荐，这时，系统需要了解需推荐内容和用户的特质，或者基于社会化网络，通过找到与当前用户相同喜好的用户，实现推荐
* 这是一个最基本的推荐引擎分类，其实大部分人们讨论的推荐引擎都是将个性化的推荐引擎，因为从根本上说，只有个性化的推荐引擎才是更加智能的信息发现过程。
* **根据推荐引擎的数据源**：其实这里讲的是如何发现数据的相关性，因为**大部分推荐引擎的工作原理还是基于物品或者用户的相似集进行推荐**。那么参考图 1 给出的推荐系统原理图，根据不同的数据源发现数据相关性的方法可以分为以下几种：
* 根据系统用户的基本信息发现用户的相关程度，这种被称为基于人口统计学的推荐（Demographic-based Recommendation）
* 根据推荐物品或内容的元数据，发现物品或者内容的相关性，这种被称为基于内容的推荐（Content-based Recommendation）
* 根据用户对物品或者信息的偏好，发现物品或者内容本身的相关性，或者是发现用户的相关性，这种被称为基于协同过滤的推荐（Collaborative Filtering-based Recommendation）
* **根据推荐模型的建立方式**：可以想象在海量物品和用户的系统中，推荐引擎的计算量是相当大的，要实现实时的推荐务必需要建立一个推荐模型，关于推荐模型的建立方式可以分为以下几种：
* 基于物品和用户本身的，这种推荐引擎将每个用户和每个物品都当作独立的实体，预测每个用户对于每个物品的喜好程度，这些信息往往是用一个二维矩阵描述的。由于用户感兴趣的物品远远小于总物品的数目，这样的模型导致大量的数据空置，即我们得到的二维矩阵往往是一个很大的稀疏矩阵。同时为了减小计算量，我们可以对物品和用户进行聚类， 然后记录和计算一类用户对一类物品的喜好程度，但这样的模型又会在推荐的准确性上有损失。
* 基于关联规则的推荐（Rule-based Recommendation）：关联规则的挖掘已经是数据挖掘中的一个经典的问题，主要是挖掘一些数据的依赖关系，典型的场景就是“购物篮问题”，通过关联规则的挖掘，我们可以找到哪些物品经常被同时购买，或者用户购买了一些物品后通常会购买哪些其他的物品，当我们挖掘出这些关联规则之后，我们可以基于这些规则给用户进行推荐。
* 基于模型的推荐（Model-based Recommendation）：这是一个典型的机器学习的问题，可以将已有的用户喜好信息作为训练样本，训练出一个预测用户喜好的模型，这样以后用户在进入系统，可以基于此模型计算推荐。这种方法的问题在于如何将用户实时或者近期的喜好信息反馈给训练好的模型，从而提高推荐的准确度

其实在**现在的推荐系统中，很少有只使用了一个推荐策略的推荐引擎，一般都是在不同的场景下使用不同的推荐策略从而达到最好的推荐效果**，例如 Amazon 的推荐，它将基于用户本身历史购买数据的推荐，和基于用户当前浏览的物品的推荐，以及基于大众喜好的当下比较流行的物品都在不同的区域推荐给用户，让用户可以从全方位的推荐中找到自己真正感兴趣的物品。

**3）各推荐机制的工作原理**

**A. 基于人口统计学的推荐**

基于人口统计学的推荐机制（Demographic-based Recommendation）是一种最易于实现的推荐方法，它只是简单的根据系统用户的基本信息发现用户的相关程度，然后将相似用户喜爱的其他物品推荐给当前用户，图 2 给出了这种推荐的工作原理。



从图中可以很清楚的看到，首先，系统对每个用户都有一个用户 Profile 的建模，其中包括用户的基本信息，例如用户的年龄，性别等等；然后，系统会根据用户的 Profile 计算用户的相似度，可以看到用户 A 的 Profile 和用户 C 一样，那么系统会认为用户 A 和 C 是相似用户，在推荐引擎中，可以称他们是“邻居”；最后，基于“邻居”用户群的喜好推荐给当前用户一些物品，图中将用户 A 喜欢的物品 A 推荐给用户 C。

这种基于人口统计学的推荐机制的**好处在于**：

* 因为不使用当前用户对物品的喜好历史数据，所以对于新用户来讲没有“冷启动（Cold Start）”的问题。
* 这个方法不依赖于物品本身的数据，所以这个方法在不同物品的领域都可以使用，它是**领域独立**的（domain-independent）。

那么这个方法的**缺点和问题是什么呢**？这种基于用户的基本信息对用户进行分类的方法过于粗糙，尤其是对品味要求较高的领域，比如图书，电影和音乐等领域，无法得到很好的推荐效果。可能在一些电子商务的网站中，这个方法可以给出一些简单的推荐。另外一个局限是，这个方法可能涉及到一些与信息发现问题本身无关却比较敏感的信息，比如用户的年龄等，这些用户信息不是很好获取。

**B. 基于内容推荐机制的推荐**

基于内容的推荐是在推荐引擎出现之初应用最为广泛的推荐机制，它的**核心思想**是根据推荐物品或内容的元数据，发现物品或者内容的相关性，然后基于用户以往的喜好记录，推荐给用户相似的物品。图 3 给出了基于内容推荐的基本原理。

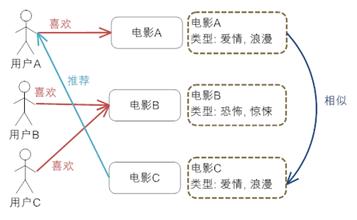


图 3 中给出了基于内容推荐的一个典型的例子，**电影推荐系统**，首先我们需要对电影的元数据有一个建模，这里只简单的描述了一下电影的类型；然后通过电影的元数据发现电影间的相似度，因为类型都是“爱情，浪漫”电影 A 和 C 被认为是相似的电影（当然，只根据类型是不够的，要得到更好的推荐，我们还可以考虑电影的导演，演员等等）；最后实现推荐，对于用户 A，他喜欢看电影 A，那么系统就可以给他推荐类似的电影 C。

这种基于内容的推荐机制的好处在于它能很好的建模用户的口味，能提供更加精确的推荐。但它也**存在以下几个问题：**

* 需要对物品进行分析和建模，推荐的质量依赖于对物品模型的完整和全面程度。在现在的应用中我们可以观察到关键词和标签（Tag）被认为是描述物品元数据的一种简单有效的方法。
* 物品相似度的分析仅仅依赖于物品本身的特征，这里没有考虑人对物品的态度。
* 因为需要基于用户以往的喜好历史做出推荐，所以对于新用户有“冷启动”的问题。

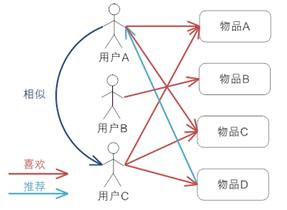
虽然这个方法有很多不足和问题，但他还是成功的应用在一些电影，音乐，图书的社交站点，有些站点还请专业的人员对物品进行基因编码，比如潘多拉，在一份报告中说道，在潘多拉的推荐引擎中，每首歌有超过 100 个元数据特征，包括歌曲的风格，年份，演唱者等等。

**C. 基于协同过滤的推荐**

它的原理很简单，就是根据用户对物品或者信息的偏好，发现物品或者内容本身的相关性，或者是发现用户的相关性，然后再基于这些关联性进行推荐。基于协同过滤的推荐可以分为三个子类：基于用户的推荐（User-based Recommendation），基于项目的推荐（Item-based Recommendation）和基于模型的推荐（Model-based Recommendation）。下面我们一个一个详细的介绍着三种协同过滤的推荐机制。

**a. 基于用户的协同过滤推荐**

基于用户的协同过滤推荐的**基本原理**是，根据所有用户对物品或者信息的偏好，发现与当前用户口味和偏好相似的“邻居”用户群，在一般的应用中是采用计算“K- 邻居”的算法；然后，基于这 K 个邻居的历史偏好信息，为当前用户进行推荐。下图 4 给出了原理图。

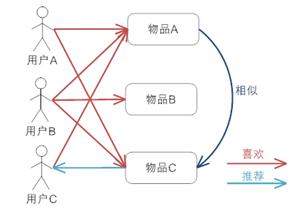


上图示意出基于用户的协同过滤推荐机制的基本原理，假设用户 A 喜欢物品 A，物品 C，用户 B 喜欢物品 B，用户 C 喜欢物品 A ，物品 C 和物品 D；从这些用户的历史喜好信息中，我们可以发现用户 A 和用户 C 的口味和偏好是比较类似的，同时用户 C 还喜欢物品 D，那么我们可以推断用户 A 可能也喜欢物品 D，因此可以将物品 D 推荐给用户 A。

基于用户的协同过滤推荐机制和基于人口统计学的推荐机制都是计算用户的相似度，并基于“邻居”用户群计算推荐，但它们所不同的是**如何计算用户的相似度**，基于人口统计学的机制只考虑用户本身的特征，而基于用户的协同过滤机制可是在用户的历史偏好的数据上计算用户的相似度，它的基本假设是，喜欢类似物品的用户可能有相同或者相似的口味和偏好。

**b. 基于项目的协同过滤推荐**

基于项目的协同过滤推荐的基本原理也是类似的，只是说它使用所有用户对物品或者信息的偏好，发现物品和物品之间的相似度，然后根据用户的历史偏好信息，将类似的物品推荐给用户，图 5 很好的诠释了它的基本原理。



假设用户 A 喜欢物品 A 和物品 C，用户 B 喜欢物品 A，物品 B 和物品 C，用户 C 喜欢物品 A，从这些用户的历史喜好可以分析出物品 A 和物品 C 时比较类似的，喜欢物品 A 的人都喜欢物品 C，基于这个数据可以推断用户 C 很有可能也喜欢物品 C，所以系统会将物品 C 推荐给用户 C。

与上面讲的类似，基于项目的协同过滤推荐和基于内容的推荐其实都是基于物品相似度预测推荐，只是相似度计算的方法不一样，前者是从用户历史的偏好推断，而后者是基于物品本身的属性特征信息。

同时协同过滤，在**基于用户和基于项目两个策略中应该如何选择呢？**其实基于项目的协同过滤推荐机制是 Amazon 在基于用户的机制上改良的一种策略，因为在大部分的 Web 站点中，物品的个数是远远小于用户的数量的，而且物品的个数和相似度相对比较稳定，同时基于项目的机制比基于用户的实时性更好一些。但也不是所有的场景都是这样的情况，可以设想一下在一些新闻推荐系统中，也许物品，也就是新闻的个数可能大于用户的个数，而且新闻的更新程度也有很快，所以它的形似度依然不稳定。所以，其实可以看出，推荐策略的选择其实和具体的应用场景有很大的关系。

**c. 基于模型的协同过滤推荐**

基于模型的协同过滤推荐就是基于样本的用户喜好信息，训练一个推荐模型，然后根据实时的用户喜好的信息进行预测，计算推荐。

基于协同过滤的推荐机制是现今应用最为广泛的推荐机制，它有以下几个显著的优点：

* 它不需要对物品或者用户进行严格的建模，而且不要求物品的描述是机器可理解的，所以这种方法也是**领域无关的**。
* 这种方法计算出来的推荐是开放的，可以共用他人的经验，很好的支持用户发现潜在的兴趣偏好

而它也存在以下几个问题：

* 方法的核心是基于历史数据，所以对新物品和新用户都有“冷启动”的问题。
* 推荐的效果依赖于用户历史偏好数据的多少和准确性。
* 在大部分的实现中，用户历史偏好是用稀疏矩阵进行存储的，而稀疏矩阵上的计算有些明显的问题，包括可能少部分人的错误偏好会对推荐的准确度有很大的影响等等。
* 对于一些特殊品味的用户不能给予很好的推荐。
* 由于以历史数据为基础，抓取和建模用户的偏好后，很难修改或者根据用户的使用演变，从而导致这个方法不够灵活

**d. 混合推荐机制**

在现行的 Web 站点上的推荐往往都不是单纯只采用了某一种推荐的机制和策略，他们往往是将多个方法混合在一起，从而达到更好的推荐效果。关于如何组合各个推荐机制，这里讲几种比较流行的组合方法。

* **加权的混合（Weighted Hybridization）**: 用线性公式（linear formula）将几种不同的推荐按照一定权重组合起来，具体权重的值需要在测试数据集上反复实验，从而达到最好的推荐效果。
* **切换的混合（Switching Hybridization）**：前面也讲到，其实对于不同的情况（数据量，系统运行状况，用户和物品的数目等），推荐策略可能有很大的不同，那么切换的混合方式，就是允许在不同的情况下，选择最为合适的推荐机制计算推荐。
* **分区的混合（Mixed Hybridization）**：采用多种推荐机制，并将不同的推荐结果分不同的区显示给用户。其实，Amazon，当当网等很多电子商务网站都是采用这样的方式，用户可以得到很全面的推荐，也更容易找到他们想要的东西。
* **分层的混合（Meta-Level Hybridization）**: 采用多种推荐机制，并将一个推荐机制的结果作为另一个的输入，从而综合各个推荐机制的优缺点，得到更加准确的推荐。