一、多线程

    线程状态：New（新生），Runnable（可运行，包括了已经在运行的和具备运行条件的），Blocked（被阻塞），Dead（死亡）

1、如何运行多线程：

   （1）定义自己的实现了Runnable接口的类

   MyRunnable implements Runnable

   {

       public void run()

        {

           your code;

        }

   }

   MyRunnable myrunnable = new MyRunnable();

   Thread t = new Thread(myrunnable);

   t.start;

   （2）直接定义Thread类的子类(不建议这样使用)

   class MyThread extends Thread

   {

       public void run()

        {

            your code;

        }

   }

   MyThread t = new MyThread();

   t.start;

   采用继承Thread类方式：

     优点：编写简单，如果需要访问当前线程，无需使用Thread.currentThread()方法，直接使用this，即可获得当前线程。

     缺点：因为线程类已经继承了Thread类，所以不能再继承其他的父类。

   采用实现Runnable接口方式：

     优点：线程类只是实现了Runable接口，还可以继承其他的类。在这种方式下，可以多个线程共享同一个目标对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

     缺点：编程稍微复杂，如果需要访问当前线程，必须使用Thread.currentThread()方法。

     不要直接调用Thread或Runnable的Run方法，那样只是在当前线程中执行任务并不会创建新的任务，用start方法才会创建新的任务。

    尽量从机制上避免这种并行任务，因为代价太大，可以只用线程池。

2、如何控制线程间的交互：

    中断线程：

   （1）线程将在run方法返回时终止，现在已经没有强制终止线程的方法了（stop方法已被废弃）

   （2）但可以利用interrupt来终止线程，当interrupt方法在一个线程上被调用时中断状态会被置位（Boolean类型），线程应该不断的检查这个标志位。首先要用静态的Thread.currentThread来获取当前线程，这样：

        while(!Thread.currentThread().isInterrupted()&& more..... )

          {

          }

          如果一个线程被阻塞了它就无法检查中断状态了（产生InterruptedException），在处理InterruptedException时最好在catch中设置下中断状态或直接抛出，以便调用者可以检测或者捕获：

          void myclass()

          {

              try{

                   sleep(delay);

               }catch(InterruptedException e)

               {

                   Thread.currentThread().interrupt();

               }

          }

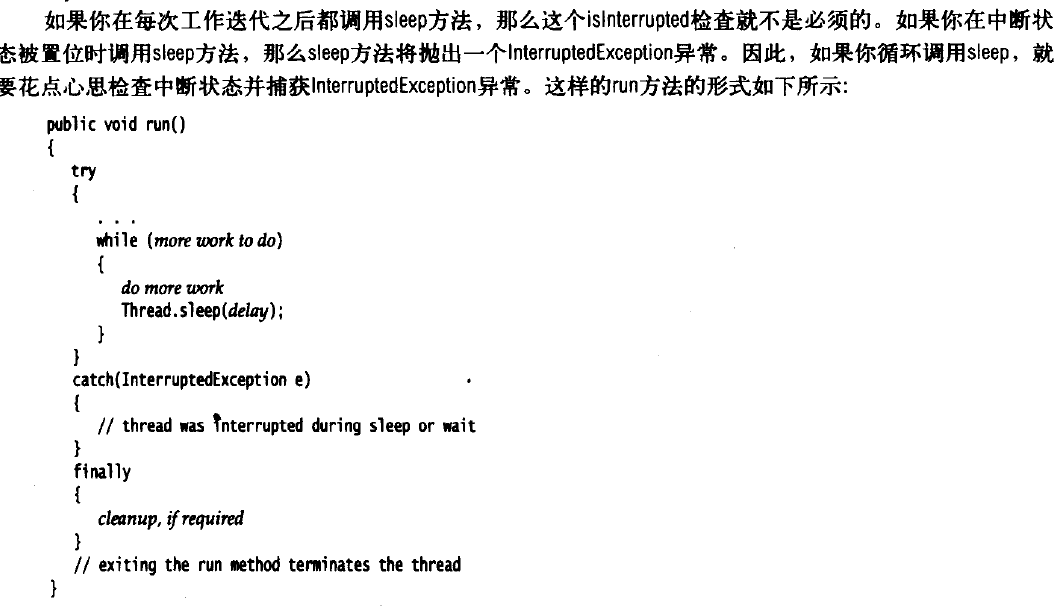
          或

          void myclass() throws InterruptedException

          {

              sleep(delay);

          }



           java.lang.Thread:

           void interrupt();给一个线程发送中断请求，将中断状态置为true

           static boolean interrupted();检测当前线程是否被中断，会清除中断状态

           boolean isInterrupted();检测一个线程是否以被终止了，不会清除中断状态

           static Thread currentThread();返回代表当前执行线程的Thread对象

           boolean isAlive();判断线程是否可运行状态

           void join();等待直到指定的线程死亡

           void join(long millis);等待指定的线程死亡或超过指定毫秒数

           stop，suspend，resume废弃了。。

**sleep, wait, yeild区别**

**sleep不释放资源不释放锁；wait释放资源释放锁；yeild不能指定时间并且只能让同优先级的线程获取执行机会；**

   （3）中断线程的目的不是让这个线程终止，而是为了引起它的注意，中断之后该怎么处理允许线程自己决定。（比如一些很重要的线程，中断后可能只记录下日志然后继续执行了）

   （4）两个类似的方法：interrupted和isInterrupted，前者是个静态方法，在检查中断时会同时清除该线程的中断状态，后者是个实例方法，只是检查中断并不改变中断状态的值。

   线程进入被阻塞状态的情况：

   （1）线程调用sleep方法进入了睡眠状态；

   （2）线程调用一个在IO上的被阻塞的操作，该输入输出未完成；

   （3）线程试图得到一个正被其他线程使用的锁；

   （4）线程在等某个触发条件；

   （5）调用了线程的suspend方法（已过时）；

   线程从被阻塞回到可运行状态：

   （1）线程被置于睡眠状态，但已超过了指定的毫秒数；

   （2）线程等待的IO操作已完成；

   （3）线程等待的锁已被其他线程释放；

   （4）线程等到了某个触发条件；

   （5）被挂起的线程上被调用了resume方法（已过时）

   注意：一个被阻塞的线程只能通过和先前阻塞它的相同过程重新进入可运行状态，而不是通过resume方法。比如，要解除IO阻塞应该使用“New I/O”库中的通道机制，当另一个线程关闭通道时被阻塞线程就变为可运行的了，而阻塞操作将抛出一个ClosedChannelException。

   线程死亡：因run方法退出而正常死亡；或因一个未被捕获的异常终止了run方法而使线程猝死。

   如何确定线程当前是否存活：Thread中的isAlive方法。如果线程是活动状态（可运行或阻塞了）则返回true，否则（处于new状态且不可运行状态或者死亡了）返回false.

                             无法确定一个活着的线程是可运行的还是被阻塞了，也无法确定一个可运行的线程是否正在运行，同样也无法区分非可运行线程和死亡线程。

   线程属性：

   （1）优先级：java有10个级别，但它高度依赖于系统，会被映射成宿主机平台的优先级，例如在Linux的java虚拟机中线程优先级完全被忽略，所有线程优先级都是一样的，所以，可以把优先级看作线程调度的一个因素，但不要将功能依赖于它。

   （2）守护线程：为其他线程服务的，如果只剩下守护线程了虚拟机就退出了，可以通过t.setDaemon()来设置成守护线程（只能在线程开始之前调用）

   （3）线程组：可以在按功能归类时使用。

                创建：String groupName = "myGroupName";

                          ThreadGroup g = new ThreadGroup(groupName);//名字是唯一标识

                           Thread t = new Thread(g, threadName);//往线程组里添加线程

                           if(g.activeCount() == 0)//判断一个线程组中是否有线程仍处于可运行状态

                           {...}

                           g.interrupt();//中断这个组中的所有线程

                    注意：线程组可以有子线程组，默认情况下一个新建的线程组是当前线程组的子组，但也可以在构造器中明确指出父线程组的名字，像activeCount()和interrupt()之类的方法引用的是本组和它的所有子组中的线程。

            java.lang.ThreadGroup:

            ThreadGroup(String name);创建一个新的ThreadGroup，它的父线程组是当前线程所在的组；（name是新线程组名字）

            ThreadGroup(ThreadGroup parent, String name);同上，并且指定了父线程组；

            int activeCount();返回线程组中活线程数量的上界；

            int enumerate(Thread[] List);得到线程组中所有活线程的引用（list是被填入线程引用的数组）

            ThreadGroup getParent();得到这个线程组的父线程组；

            void interrupt();中断这个线程组和它的子线程组中的所有线程；

   （4）未捕获异常处理器：

        线程的run方法不能抛出任何被检测的异常，但不被检查的异常可以导致线程终止，然而，并不需要任何catch子句来处理可以被传播的异常，在线程死亡之前异常将会被传递给未捕获异常处理器来处理。

          处理器必须从属于实现了Thread.UcaughtExceptionhandler接口的类（只有一个方法：void uncaughtException(Thread t,Throwable e),主要是为了在线程终止时记录日志）

          可以用setUcaughtExceptionHandler方法为任何一个线程安装处理器，也可用Thread类的静态方法setDefaultUcaughtExceptionhandler为所有线程设置默认处理器(优先使用setUcaughtExceptionHandler设置的handler，

          如果没有的话再用setDefaultUcaughtExceptionhandler的handler)

        如果没有安装默认处理器，那处理器就是Null，如果没有为一个单独的线程安装处理器，那它的默认处理器就是它的ThreadGroup对象，ThreadGroup实现了Thread.UcaughtExceptionhandler,它的uncaughtException做如下处理：

              1）如果该线程组有父线程组，则调用父线程组的uncaughtException；

               2）否则，如果Thread.getDefaultUcaughtExceptionhandler返回一个非null的处理器，则就是用这个处理器；

               3）否则，如果Throwable是一个ThreadDeath实例，则什么都不做；

               4）否则，线程的名字和Throwable堆栈踪迹将被输出到System.err

          参考收藏夹：使用UncaughtExceptionHandler重启线程

(unchecked exception即RuntimeException（运行时异常）是不需要try...catch...或throws 机制去处理的异常，就好像空指针异常一样，你可以在任何的代码中进行抛出或捕获，

而checked exception如果需要抛出的话，则必须在代码中声明，但是因为线程的run方法是你从父类Thread或者是Runnable继承过来的，你覆盖其父类方法时不能抛出父类没有声明的可捕获异常。所以你继承的run方法中不能抛出任何的exception)

java多线程中的异常处理：

在java多线程程序中，所有线程都不允许抛出未捕获的checked exception，也就是说各个线程需要自己把自己的checked exception处理掉。这一点是通过java.lang.Runnable.run()方法声明(因为此方法声明上没有throw exception部分)进行了约束。但是线程依然有可能抛出unchecked exception，当此类异常跑抛出时，线程就会终结，而对于主线程和其他线程完全不受影响，且完全感知不到某个线程抛出的异常(也是说完全无法catch到这个异常)。JVM的这种设计源自于这样一种理念：“线程是独立执行的代码片断，线程的问题应该由线程自己来解决，而不要委托到外部。”基于这样的设计理念，在Java中，线程方法的异常（无论是checked还是unchecked exception），都应该在线程代码边界之内（run方法内）进行try catch并处理掉.

但如果线程确实没有自己try catch某个unchecked exception，而我们又想在线程代码边界之外（run方法之外）来捕获和处理这个异常的话，java为我们提供了一种线程内发生异常时能够在线程代码边界之外处理异常的回调机制，即Thread对象提供的setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)方法。

通过该方法给某个thread设置一个UncaughtExceptionHandler，可以确保在该线程出现异常时能通过回调UncaughtExceptionHandler接口的public void uncaughtException(Thread t, Throwable e) 方法来处理异常，这样的好处或者说目的是可以在线程代码边界之外（Thread的run()方法之外），有一个地方能处理未捕获异常。但是要特别明确的是：虽然是在回调方法中处理异常，但这个回调方法在执行时依然还在抛出异常的这个线程中！另外还要特别说明一点：如果线程是通过线程池创建，线程异常发生时UncaughtExceptionHandler接口不一定会立即回调。

比之上述方法，还有一种编程上的处理方式可以借鉴，即，有时候主线程的调用方可能只是想知道子线程执行过程中发生过哪些异常，而不一定会处理或是立即处理，那么发起子线程的方法可以把子线程抛出的异常实例收集起来作为一个Exception的List返回给调用方，由调用方来根据异常情况决定如何应对。不过要特别注意的是，此时子线程早以终结。

   同步：（查看字节码命令：Javap -c -v 类名）

       锁：Synchronized关键字  或  ReentrantLock（5.0后才引入的）

        Synchronized: wait/notify/notifyAll

        Lock/Condition： await/signal/signalAll

        1）ReentrantLock使用：

           public class Bank

            {

                private Lock bankLock = new ReentrantLock();

                public void transfer(int from, int to, int amount)

                 {

                     bankLock.lock();

                      try{

                         ...转账操作

                      }

                      finally{

                         bankLock.unLock();

                      }

                 }

            }

            条件对象：当一个线程进入了临界区却发现它必须等待某个条件满足后才能执行，这时需要使用一个条件对象来管理那些已获得了锁却不能执行有用工作的线程。

           例如，转账操作，必须将检查余额是否足够和转账操作锁成一块（否则检测完后有可能被其他线程改了余额），如果获得了锁却发现条件不满足，此时锁又被自己占着了，其他线程也没法给这个账户充值了。

            应该由这个锁对象上的对应的条件对象调用await方法使当前线程进入阻塞状态（等待该条件集），这样其他线程便有机会获得锁，来增加余额，其他线程操作完后应调一下相同条件对象的signalAll方法来解除阻塞。

            public class Bank

            {

                private Lock bankLock = new ReentrantLock();

                 private Condition sufficientFunds;

                public void transfer(int from, int to, int amount)

                 {

                     sufficientFunds = bankLock.newCondition();

                     bankLock.lock();

                      try{

                        while(余额不足)

                         {

                             sufficientFunds.await();

                         }

                         ...转账操作

                         sufficientFunds.signalAll();//为何在这里唤醒，因为如果其他线程走到了这一步说明上面的检测条件已经满足了是应该唤醒阻塞的线程也去检测一下试试了

                      }

                      finally{

                         bankLock.unLock();

                      }

                 }

            }

          注意：1）一个锁对象可以有一个或多个条件对象

                  2）等待获得锁的线程调用了await的线程之间有本质的区别，一旦一个线程调用了await方法它就进入了等待该条件集中，当锁可获得时线程也不能立刻解除阻塞它要一直保持阻塞状态直到另一个线程调用同一个条件上的signalAll为止。

                    3）就算调用了signalAll方法，也不表明因调await而阻塞的那个线程一定就能立刻返回执行，signalAll只是解除了它的阻塞状态，它必须通过竞争来获得对象的访问；

                    4）signalAll不保证条件一定满足，只是告诉阻塞的线程可以再去检测一下这个条件了；

                    5）一般来说对await的调用应在以下形式类似的循环中

                       while(!(OK to proceed))

                       condition.await();

                    6）最终必须要有其他线程调用signalAll,一个线程调用了await后它自己不能解除阻塞状态，如果没有其他线程来解除等待线程的阻塞状态它就永远不会运行了，这就成了死锁；（如果所有线程都阻塞了，最后一个活跃的也调用了await那程序就挂起了）

                    7）当线程拥有某一个条件的锁时，它只能在这个条件上调用await、signal、signalAll（await什么时候返回呢？：一是设置的超时时间到了的时候，二是其他线程调用了signal/signalAll，三是线程中断了）;

                    8）另一个方法signal,是随机解除等待集中的某个线程的阻塞状态，这比解除所有的更有用处但也更有危险，因为如果随机选中的还是无法运行的话它将再次阻塞，如果没有其他线程调signal那系统就死锁了；

                    9）锁是可重入的，线程能够重复的获取它已经拥有的锁，锁对象维护一个持有计数（为0时把锁释放）来跟踪对lock方法的嵌套调用，线程在每次调用Lock后都要调用unlock来释放锁，所以被一个锁保护的代码可以调用另一个使用相同锁的方法。

        2）Synchronized关键字(还可以同步块):

           public class Bank

            {

                public synchronized void transfer(int from, int to, int amount)

                 {

                   while(acounts(from) < amount)

                    {

                        wait;

                    }

                    ...转账操作

                    notifyAll();

                 }

            }

          注意：每个对象都有一个隐式的锁，并且每个锁都有一个隐式的条件，由锁来管理试图进入synchronized方法的线程，而由条件来管理那些调用wait的线程。

               但隐式的锁有一些缺点：

               1）你不能中断一个正在试图获得锁的线程；

               2）试图获得锁时不能设定超时；

               3）每个锁只有一个条件有时显得不够用；

               4）虚拟机的加锁原语不能很好地映射到硬件可用的最有效的加锁机制上。

Java对象锁和类锁全面解析（多线程synchronized关键字）：

对象锁和类锁（类锁是指对应的java.lang.Class类的实例的锁（在方法区））：

**事实上，synchronized修饰非静态方法、同步代码块的synchronized (this)用法和synchronized (非this对象)的用法锁的是对象，线程想要执行对应同步代码，需要获得对象锁。**

**synchronized修饰非静态方法以及同步代码块的synchronized (类.class)用法锁的是类，线程想要执行对应同步代码，需要获得类锁**

     那么我们究竟应该使用哪一种呢？Lock/condition 还是 synchronized呢：

     1）最好两者都不用，很多情况下可以使用java.util.concurrent包中的一种机制，他会为你处理所有的加锁（比如用阻塞队列来同步完成一个共同任务的线程）；

    2）sychronized比较简洁，如果能适应场景尽量用这种，这样可以减少代码数量，减少出错几率；（至少目前使用synchronized有额外的好处，监控虚拟机的工具可以提交关于隐式的锁和条件状态的报告，这对于调试死锁很有帮助）

     3）只有在非常需要lock/condition的独有特性时再用它。

{start

ReenTrantLock可重入锁（和synchronized的区别）总结

**可重入性：**

从名字上理解，ReenTrantLock的字面意思就是再进入的锁，其实synchronized关键字所使用的锁也是可重入的，两者关于这个的区别不大。两者都是同一个线程没进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

**锁的实现：**

Synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的，有什么区别，说白了就类似于操作系统来控制实现和用户自己敲代码实现的区别。前者的实现是比较难见到的，后者有直接的源码可供阅读。

**性能的区别：**

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

**功能区别：**

便利性：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

**ReenTrantLock独有的能力：**

1.      ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。

2.      ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

3.      ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。

**ReenTrantLock实现的原理：**

在网上看到相关的源码分析，本来这块应该是本文的核心，但是感觉比较复杂就不一一详解了，简单来说，ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。**想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。**

**什么情况下使用ReenTrantLock：**

答案是，如果你需要实现ReenTrantLock的三个独有功能时。

end}

Lock方法不能被中断，如果一个线程在等待获得一个锁时被中断，那么中断线程将一直阻塞直到锁可获得为止，如果发生死锁，那么lock方法就无法终止。

如果调用一个带超时的tryLock，那么如果线程在等待时被中断，将抛出一个InterruptedException（这是一个非常有用的特性，它允许程序打破死锁）。（如果希望出现这种情况时线程继续等待，那么可以使用awaitUninterruptibly来代替await）

        java.util.concurrent.locks.Condition

        boolean tryLock();尝试获得锁但是不会发生阻塞，如果成功则返回true；

        boolean tryLock(long time, TimeUnit unit);尝试获得锁，但阻塞时间不会超过给定的值，如果成功返回true；

        void lockInterruptily();获得锁，但是会不确定地发生阻塞，如果线程被中断，抛出一个InterruptedException；

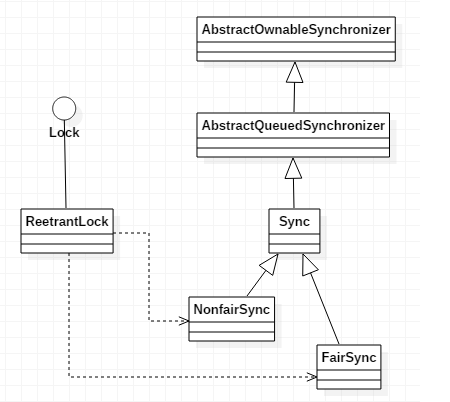
        java.util.concurrent.locks.Condition

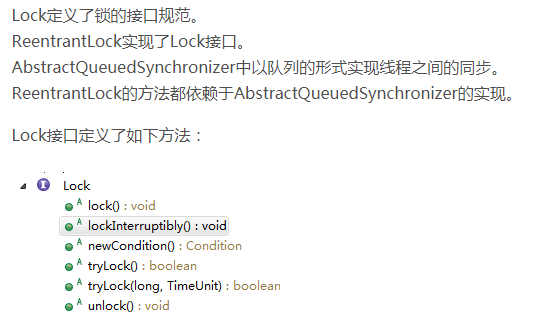
        boolean await(long time, TimeUnit unit);进入该条件的等待集，阻塞到线程被移除等待集或等待了给定的时间为止，如果等待时间到了而返回则返回false，否则返回true；

        void awaitUninterruptibly();进入该条件的等待集，阻塞到线程被移出等待集为止，若被中断不会抛出InterruptedException

reentrantlock 原理：

<http://www.cnblogs.com/xrq730/p/4979021.html>





clipboard.png

clipboard.png

读写锁：

   监视器？？？

   volatile域：

   如果只对读写实例的一两个域就使用同步，开销就大了，这时可以用volatile,volatile修饰的域每次读或写都会重新加载一遍再操作。

   在如下情况下对某个域的访问是安全的：

   1）域是volatile的；

   2）域是final的，并且在构造器调用完成后被访问；

   3）对域的访问有锁保护；

   死锁：相互等待。解决：信号量等，，，仔细设计你的程序。

   公平锁：公平策略会优待那些等待了最长时间的线程，但保证公平可能会大大影响性能，一般情况下不需要。

   Lock fairLock = new ReentrantLock(true);

   锁测试和超时：ReentrantLock中的tryLock方法（也可以设超时参数）。当tryLock被调用时如果锁可获得不管其他线程了等了多久，这个线程也会立刻获得锁（即使这个锁使用了公平锁），如果不想这样的话可以：if(myLock.tryLock(0,TimeUnit.SECONDS))....

                 lock方法不能被中断，如果线程在等待获得一个锁时被中断，那么线程将一直阻塞到锁可获得为止（如果发生死锁Lock方法将无法终止，如果调用带超时的tryLock方法线程等待锁时被中断将会抛出InterruptedException异常，这是非常有用的特性因为它允许程序打破死锁（如果希望在抛出异常后仍继续等待可以用awaitUnintteruptibly代替await，当然这样也不太合理））

   读写锁：ReentrantReadWriteLock：（读锁只排斥写，写锁排斥一切）

        使用：1>先创建ReentrantReadWriteLock对象

                  ReentrantReadWriteLock rwl = new ReentrantReadWriteLock()；

                 2>抽取读锁和写锁

                   private Lock readLock = rwl.readLock();

                    private Lock writeLock = rwl.writeLock();

                 3>使用读写锁

                   public int getSomeThing()

                    {

                        readLock.lock();

                         try{

                              写取操作...

                         }

                         finally{

                         readLock.unlock();

                         }

                    }

                    public int writeSomeThing()

                    {

                        writeLock.lock();

                         try{

                              读写操作...

                         }

                         finally{

                         writeLock.unlock();

                         }

                    }

   stop被废弃是因为它暴力的终止线程，会导致对象处于不一致状态；

   suspend被废弃是因为会产生死锁。（比如A将B挂起了，但A需要的某个资源恰好被B锁住了）

3、阻塞队列（先进先出，队尾加入，头部移除）

   LinkedBlockingQueue（容量无上限，当然可以指定其上限容量）   由链表实现

   ArrayBlockingQueue（构造时必须指定容量，并且可以选择是否需要公平性，如果公平参数被设置了等待时间最长的线程会优先得到处理，但有损性能）   由数组实现

   PriorityBlockingQueue（带优先级的队列而不是先进先出，元素按优先级顺序被移除，也没有上限，如果队列为空那么取元素操作就会阻塞）     由堆来实现

   方法：add      增加一个元素，如果队列已满则抛出IllegalStateException

         remove   移除并返回队列头部元素，空则抛出NoSuchElementException

          element  返回队列头部元素，空则抛出NoSuchElementException

          offer    添加一个元素并返回true，已满则返回false

          poll     移除并返回头部元素，空则返回Null

          peek     返回头部元素，空则返回Null

          put      添加元素，满则阻塞

          take     移除并返回头部元素，空则阻塞

   注意：（1）向已满的队列添加和从空队列移除都会导致线程阻塞；

         （2）队列会自行负载平衡（如果第一个线程运行比第二个慢，第二个在等待结果时就会阻塞这样第一个就会赶上来）

         （3）Poll和peek出错时返回null，所以向队列中插入Null是不合理的

4、线程安全的集合：阻塞队列、高效队列和散列表（ConcurrentLinkedQueue和ConcurrentHashMap）、写数组的拷贝（CopyOnWriteArrayList和CopyOnWriteArraySet(它们的修改线程都有一个底层数组拷贝)）、被同步包装器包装的ArrayList和HashMap

   （1）并发散列表能有效支持多个读取器操作和固定数量的写入器操作，如果写入操作多于设定的固定值多出来的部分将会阻塞（基本原理就是允许不同的线程对数据结构的不同部分进行同时访问，将竞争最小化）

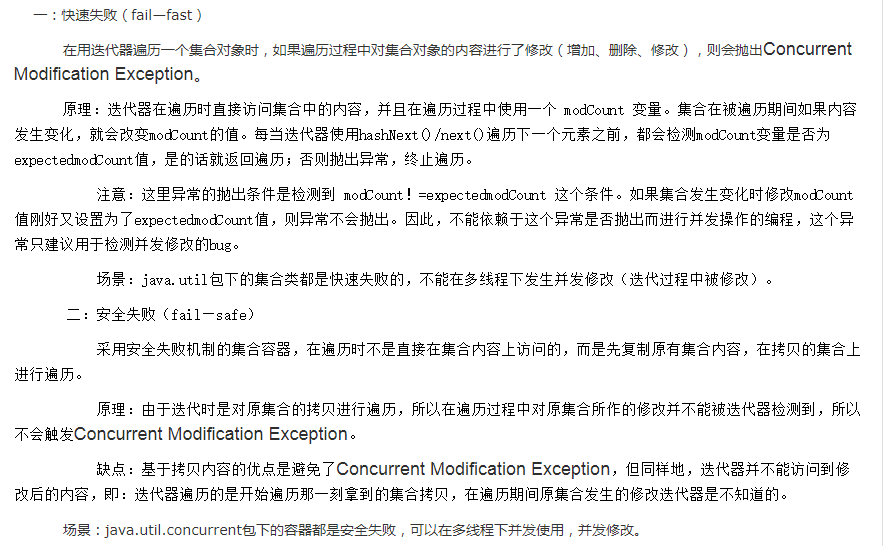
   （2）这些集合返回弱一致性迭代器（iterator），这意味着不一定能够反映出它们被构造之后所做的所有修改，但他们不会两次返回同一个值也不会抛异常（**而在Java.util包中的collection如果在构建迭代器之后发生改变其迭代器将会抛出ConcurrentModificationException**）

（Java1.5并发包（java.util.concurrent）包含线程安全集合类，允许在迭代时修改集合。迭代器被设计为fail-fast的，会抛出ConcurrentModificationException。一部分类为：CopyOnWriteArrayList、 ConcurrentHashMap、CopyOnWriteArraySet；

   在作为参数传递之前，我们可以使用**Collections.unmodifiableCollection(Collection c)**方法创建一个只读集合，这将确保改变集合的任何操作都会抛出**UnsupportedOperationException**

   Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util包下面的所有的集合类都是快速失败的，而**java.util.concurrent包下面的所有的类都是安全失败的**。快速失败的迭代器会抛出**ConcurrentModificationException**异常，而安全失败的迭代器永远不会抛出这样的异常

）



   （3）ConcurrentHashMap有一些原子性的关联插入和关联移除操作：PutIfAbsent（key,value），remove（key,value）,replace(key,oldvalue,newValue);

   （4）Vector和HashTable分别提供了线程安全的动态数组和散列表实现，但被废弃了用Arraylist和HashMap取代了它们（但它们本身却不是线程安全的），用同步包装器把它们变成安全的：

        List sychArrayList = **Collections.synchronizedList(new ArrayList());**

          Map synchHashMap = **Collections.synchronizedMap(new HashMap());**

          这时如果想迭代它就得使用同步块：

          synchronized(synchHashMap)

          {

              Iterator it = synchHashMap.keySet().iterator();

               while(it.hasNext())....

          }

java.util.concurrent.ConcurrentLinkedQueue<E> 创建一个可被多个线程安全访问的无边界的非阻塞队列；

java.util.concurrent.ConcurrentHashMap<K,V> 创建一个可被多个线程安全访问的散列表

java.lang.Object

        java.util.Collections

5、Callable和Future

   Callable可以看成是由返回值的Runnable，它只有一个call方法（Callable<Integer>表示一个最终返回Integer的异步计算）；

   Future保存异步计算的结果，它有get（获取不到抛异常，也可以设置超时时间），cancel（可设置是否允许取消），isCancel，isDone方法

   用法：(和Futuretask包装器一起用)

       Callable<Integer> myComputation = ....;

        Futuretask<Integer> task = new FutureTask<Integer>(myComputation);

        Thread t = new Thread(task);

        t.start();

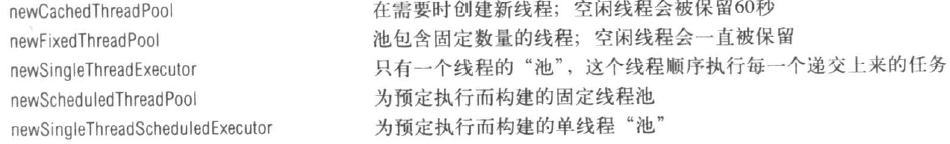
        ....做其他的事

        Integer result = task.get();

6、执行器、线程池

   使用流程：

   （1）调用Executor（执行器）类中的静态方法newCachedThreadPool或newFixedThreadPool或newSingleThreadExecutor或newScheduledThreadPool或newSingleScheduledThreadPool



   （2）调用submit来提交一个runnable或callable对象；

   （3）如果希望取消任务或者如果提交了Callable对象，那就保存好返回的future对象；

   （4）如果不想再提交任务就调用shutdown.

   预订执行：ScheduledExecutorService接口有为预订或重复执行设计的方法，Executors的newScheduledThreadPool和newSingleScheduledThreadPool将返回实现了ScheduledExecutorService接口对象。

   控制线程组：将执行器作为线程池来使用这样会增加任务执行效率，但执行器还有个更实际意义的原因：用来控制一组相关的线程（例如用shutdown来取消执行器中的所有任务）

               invokeAny方法提交所有对象到一个Callable对象的集合中，并返回某个已完成任务的结果，但你无法知道返回的究竟是哪一个任务的结果（也许是最快的那个）；

                  invokeAll方法提交所有对象到一个Callable对象的集合中，并返回所有任务列表（Futrue列表），代表所有任务的结果。

7、同步器？？

8、障栅：CyclicBarrier类实现了一个称为障栅的集合点，比如当大量集合运行在一次计算的不同部分时，当所有部分都计算好了之后结果需要被整合，当一个线程完成了它自己的那部分后可以让它运行到障栅处，一旦所有线程都到达这个障栅，障栅就撤销，线程就可以继续运行。

       java.util.concurrent.CyclicBarrier

       用法：

            （1）首先构建一个障栅，并给出参与的线程数（还可包括所有线程到达时执行的动作）

                  CyclicBarrier barrier = new CyclicBarrier(threadNum);

                    (或 Runnbale barrierAction = ...;

                        CyclicBarrier barrier = new CyclicBarrier(threadNum, barrierAction);)

             （2）每个线程都做自己个事，完成后在障栅上调用await

                  public void run（）

                    {

                        dowork();

                         barrier.await();

                         ...

                    }

          注意：1）如果任意一个在障栅出等待的线程离开了那么障栅就破坏了（线程可能是在await时设置了超时，或者被中断了），这样的话所有其他线程的await方法都将抛出一个BrokenBarrierException异常，那些已经在等待的线程立即终止await调用；

                2）障栅被称为是循环的，因为它能在所有等待线程被释放后被重用；

9、倒计时门栓（java.util.concurrent.CountDownLatch）：让一个线程集等待直至计数器变为0为止，与障栅的几个不同点：

               （1）不是所有线程都需要等待到门栓打开为止；

                  （2）门栓可以由外部事件打开；

                  （3）倒计时门栓是一次性的，当计数器到0后就不能重用了。

10、交换器（java.util.concurrent.Exchanger）：当两个线程工作在同一个数据缓冲区的两个实例上时使用（比如一个线程往缓冲区写，另一个在这里面取，都完成后，需要交换缓冲）。

       V exchange(V item);

       V exchange(V item, TimeUnit unit);

       阻塞直至另一个线程调用该方法，然后和其他线程交换item并返回从该其他线程交换而来的item；

11、同步队列（java.util.concurrent.SynchronousQueue）：一种将生产者和消费者线程配对的机制。当一个线程调用SynchronousQueue上的put方法时它会阻塞知道另一个线程调用take，反之亦然。

    注意：（1）与Exchanger不同，同步队列只在一个方向上传递，从生产者到消费者；

           （2）即使SynchronousQueue实现了BlockingQueue接口，它概念上亦然不是一个队列，它不包含任何元素。

    SynchronousQueue();

    SynchronousQueue(boolean fair);

    构建一个允许线程提交item的同步队列，若fair为true，队列将优先照顾等待时间最长的线程；

    void put(V item);

    阻塞直至另一个线程调用take来获取item为止；

    V take();

    阻塞直至另一个线程调用put为止，返回其他线程提供的item；

12、信号量（java.util.concurrent.Semaphore）：管理大量许可，为了通过信号量线程调用acquire获取许可，也可调用release释放许可（其实没有实际对象，信号量仅维护一个计数器）

      Semaphore(int permits);

      Semaphore(int permits, boolean fair);

      以给定的许可数目为最大值构建一个信号量，如果fair为true，队列优先照顾等待时间最长的线程；

      void acquire();

      等待获得一个许可；

      boolean tryAcquire();

      boolean tryAcquire(long time, TimeUnit unit);

      尝试获得一个许可，若没有任何许可可用则返回false；

      void release();

      释放一个许可；

13、线程和Swing：

    Swing不是线程安全的；

    单一线程规则：在线程中做的事不能接触用户界面，如果需要就在线程启动前来接触；

二、集合

     队列实现：链表或循环数组

1、迭代器：Iterator（等价于Enumration接口）一定要先判断hasNext再Next，因为迭代器可以看作是元素之间的，调用next时会越过下一个元素并返回刚刚越过的元素的引用。（到达尾部后如果不判断直接next的话会抛异常）

           移除元素：必须先next,再remove（不调next的话会抛出IllegalStateException）

             泛型方法：例如Iterator和Enumration都是泛型接口，你可以编写在任何种类的集合上运行的实用方法，集合类的用户可以使用泛型接口中的一组更加丰富的方法，但实际数据结构的实现者并没有要实现所有例行方法的负担。

                       API：retainAll(Collocation<?> other); 移除与other中任何元素都不相等的所有元素；

                               其他略。。

2、具体的集合：

   （1）链表（List）：

        1）在Java所有的链表都是双重链接的，也就是说每个节点中还存放着对它的前面节点的引用；

          2）链表和泛型集合间有一个重要的差别，链表是一个有序集合其中各个对象的位置起着重要的作用（例如add是加到尾部，如果要加到中间某个位置，需要使用迭代器）；

          3）集合类库提供了一个包含add的Iterator的子接口ListIterator（列表迭代器），这个子接口中的add与Collection的add不同，它不返回布尔值，因为它假设add总是要改变列表（还有两个方法：hasPrevious和previous，和hasNext，next类似，还有个se方法来设置某个位置上的新的值）；

          4）for(int i=0;i<list.size();i++) do something with list.get(i);  这种方式效率极低，因为它每次都是从头部开始搜索，不缓存任何位置信息（唯一一点优化就是如果索引比size()/2大时从尾部搜索）；

          5）list.listIterator(n)将返回一个迭代器，它指向索引为n的元素前面的位置；

   （2）数组列表（ArrayList）：

        1）如果需要一个动态数组时可以用Arraylist(也可以用Vector)

          2）为何用Arraylist替代了Vector了呢？因为Vector的所有方法都是同步的，而ArrayList不是，大多数操作我们不需要同步方法时没必要为同步浪费太多的性能；

   （3）散列集：通过散列码（通过String类的hashCode）快速查找需要的对象（实现自己的类时要实现相应的hashCode，并且要和equals兼容，也就是如果a.equals(b)，那a和b的hashCode要相同）

        1）散列表：是个链表的数组，每个链表是一个散列表元。（有研究认为应将散列表元数量设置成一个素数，以防止键的聚焦，类库中设置的是2的幂次，一般为16）

          2）加载因子决定何时再散列（默认值为0.75）

   （4）树集（TreeSet）：和散列表类似，不过有所改进，它是一个有序集合；（使用的红黑树数据结构）。将一个元素添加进去的速度要比添加到散列表中慢，因为要计算比较;

                         TreeSet();

                              TreeSet(Comparator<? extends E> c);构造时指定比较器

   （5）优先级队列：使用堆的数据结构。能够在以任意顺序插入元素后再按排序顺序读取这些元素的数据结构；

   （6）映射表：HashMap、TreeMap

   （7）专用的集和映射表类：

        1）弱散列映射表：WeakHashMap（内部机制是使用弱引用来持有键）。因为垃圾回收器跟踪的是活的对象，只要映射表是活得，那它里面的所有散列表元就都不会被回收，当某个对象的所有引用都消亡时这个元素理应被回收。

                          内部运行机制：？？

          2）链接散列集和链接散列映射表：LinkedHashSet，LinkedHashMap

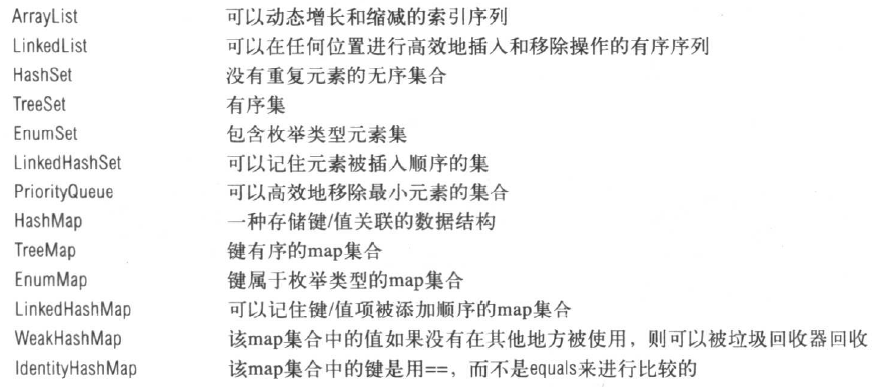
          3）枚举集和枚举映射表：EnumSet和EnumMap

          4）标识散列映射表：IdentityHashMap

**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，那么它们的hashCode值一定要相同；

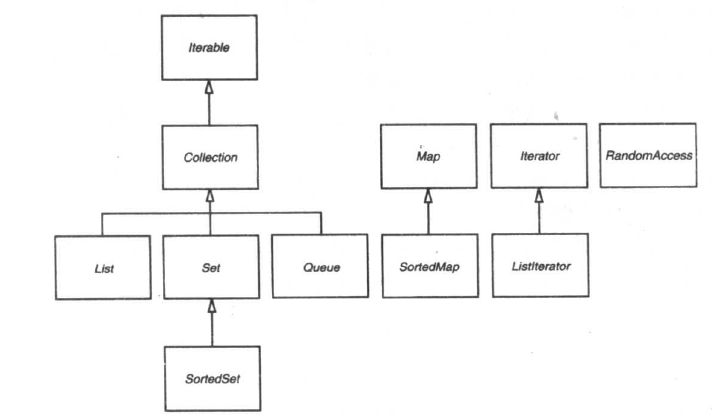
2、如果两个对象的hashCode相同，它们并不一定相同

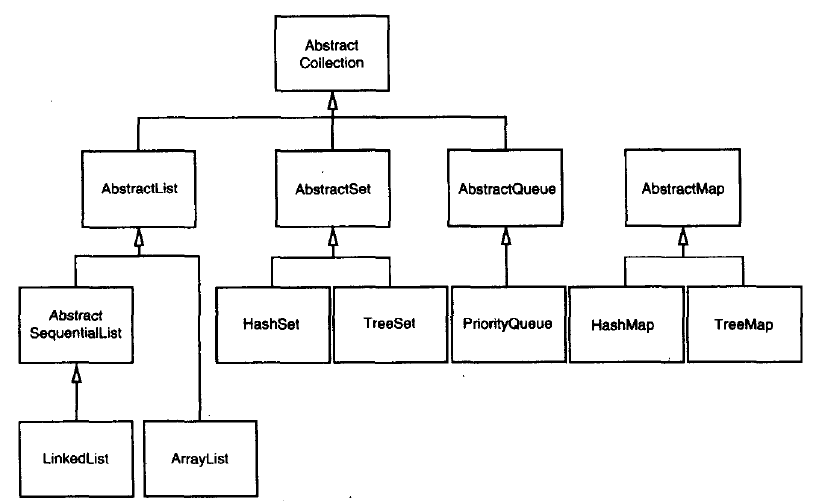


3、集合框架：？？？

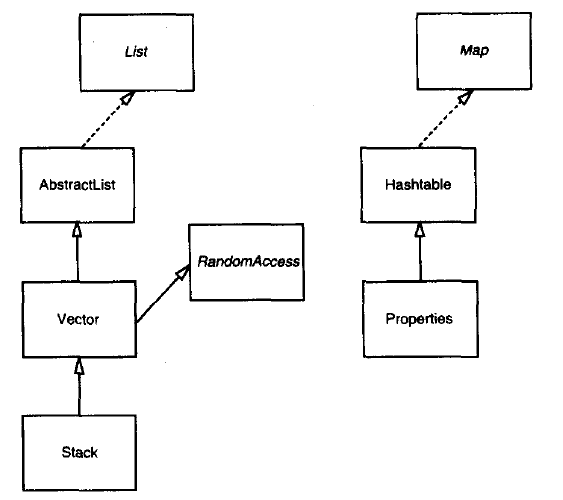
Collection 和 Map:

**集合关系：**





废弃的：



视图和包装器：

（1）轻量级集合包装器：

        Card[] cardDeck = new Card[52];

        List<Card> CardList = Arrays.asList(cardDeck);//返回的不是一个ArrayList，而是一个视图对象，所有改变数组大小的操作如add、remove等都会抛出UnsupportedOperationException；

        List<String> names=Arrays.asList("Ammy", "Bob", "Carl");

        Collections.nCopies(n, anObject);//产生一个不可修改的对象（包含N个元素），每个元素都像是一个object的错觉，但是这个object只被存储了一次；

（2）不可修改视图



（3）同步视图：同步包装器包装一下

（4）被检查视图：

         ArrayList<String> strings = new ArrayList<String>();

         List<String> safeStrings = Collections.checkedList(strings, String.class);

         ArrayList rawList = safeStrings;

         rawList.add(new Date());//会抛ClassCastException，否则不会抛异常，而是在后面的get并将其转化为string时才会抛异常；

（5）批操作：retainAll(b); removeAll(b);之类的

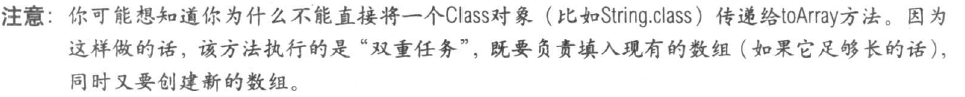
（6）集合与数组的转换：

        数组转集合：Arrays.asList(b);

        集合转数组：staff.toArray();//但是该方法返回的是一个对象数组Object[]，也不能强制类型转换,这样会报错：String[] values=(String[])staff.toArray();

                           但可以这样：String[] values=staff.toArray(new String[0]);

                                               staff.toArray(new String[staff.size()]);//这样没有创建任何新的数组



（7）算法：

         Collections.sort(items);//items必须实现了Comparable接口

       或Collections.sort(items, itemComparater);

（8）遗留：BitSet，Stack等。。。。

4、**UnsupportedOperationException是什么？**

UnsupportedOperationException是用于表明操作不支持的异常。在JDK类中已被大量运用，在集合框架java.util.Collections.UnmodifiableCollection将会在所有add和remove操作中抛出这个异常

public class CollectionsTest {

     @Test

     public void test(){

          Collection<String> c = new ArrayList<String>();

          Collection<String> s = Collections.unmodifiableCollection(c);

          c.add("str");

          System.out.println(s);

     }

}

5、**我们能否使用任何类作为Map的key？**

我们可以使用任何类作为Map的key，然而在使用它们之前，需要考虑以下几点：

（1）如果类重写了equals()方法，它也应该重写hashCode()方法。

（2）类的所有实例需要遵循与equals()和hashCode()相关的规则。请参考之前提到的这些规则。

（3）如果一个类没有使用equals()，你不应该在hashCode()中使用它。

（4）用户自定义key类的最佳实践是使之为不可变的，这样，hashCode()值可以被缓存起来，拥有更好的性能。不可变的类也可以确保hashCode()和equals()在未来不会改变，这样就会解决与可变相关的问题了。

比如，我有一个类MyKey，在HashMap中使用它。

//传递给MyKey的name参数被用于equals()和hashCode()中

MyKey key = new MyKey('Pankaj'); //assume hashCode=1234

myHashMap.put(key, 'Value');

// 以下的代码会改变key的hashCode()和equals()值

key.setName('Amit'); //assume new hashCode=7890

//下面会返回null，因为HashMap会尝试查找存储同样索引的key，而key已被改变了，匹配失败，返回null

myHashMap.get(new MyKey('Pankaj'));

那就是为何String和Integer被作为HashMap的key大量使用。

6、**HashMap和HashTable有何不同？**

（1）HashMap允许key和value为null，而HashTable不允许。

（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。

（3）在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你很容易从HashMap转向LinkedHashMap，但是HashTable不是这样的，它的顺序是不可预知的。

（4）HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。

（5）HashTable被认为是个遗留的类，如果你寻求在迭代的时候修改Map，你应该使用CocurrentHashMap。

7、**如何决定选用HashMap还是TreeMap？**

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

8、**ArrayList和Vector有何异同点？**

ArrayList和Vector在很多时候都很类似。

（1）两者都是基于索引的，内部由一个数组支持。

（2）两者维护插入的顺序，我们可以根据插入顺序来获取元素。

（3）ArrayList和Vector的迭代器实现都是fail-fast的。

（4）ArrayList和Vector两者允许null值，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点。

（1）Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。

（2）ArrayList比Vector快，它因为有同步，不会过载。

（3）ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

9、**Array和ArrayList有何区别？什么时候更适合用Array？**

Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

Array是指定大小的，而ArrayList大小是固定的。

Array没有提供ArrayList那么多功能，比如addAll、removeAll和iterator等。尽管ArrayList明显是更好的选择，但也有些时候Array比较好用。

（1）如果列表的大小已经指定，大部分情况下是存储和遍历它们。

（2）对于遍历基本数据类型，尽管Collections使用自动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得很慢。

（3）如果你要使用多维数组，使用[][]比List<List<>>更容易。

10、**哪些集合类提供对元素的随机访问？**

ArrayList、HashMap、TreeMap和HashTable类提供对元素的随机访问

11、**EnumSet是什么？**

java.util.EnumSet是使用枚举类型的集合实现。当集合创建时，枚举集合中的所有元素必须来自单个指定的枚举类型，可以是显示的或隐示的。EnumSet是不同步的，不允许值为null的元素。它也提供了一些有用的方法，比如copyOf(Collection c)、of(E first,E…rest)和complementOf(EnumSet s)。

12、**哪些集合类是线程安全的？**

Vector、HashTable、Properties和Stack是同步类，所以它们是线程安全的，可以在多线程环境下使用。Java1.5并发API包括一些集合类，允许迭代时修改，因为它们都工作在集合的克隆上，所以它们在多线程环境中是安全的。

13、**并发集合类是什么？**

Java1.5并发包（java.util.concurrent）包含线程安全集合类，允许在迭代时修改集合。迭代器被设计为fail-fast的，会抛出ConcurrentModificationException。一部分类为：CopyOnWriteArrayList、 ConcurrentHashMap、CopyOnWriteArraySet

14、**BlockingQueue是什么？**

Java.util.concurrent.BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间。

BlockingQueue接口是Java集合框架的一部分，主要用于实现生产者-消费者模式。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在BlockingQueue的实现类中被处理了。

Java提供了集中BlockingQueue的实现，比如ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue等

15、**队列和栈是什么，列出它们的区别？**

栈和队列两者都被用来预存储数据。java.util.Queue是一个接口，它的实现类在Java并发包中。队列允许先进先出（FIFO）检索元素，但并非总是这样。Deque接口允许从两端检索元素。

栈与队列很相似，但它允许对元素进行后进先出（LIFO）进行检索。

Stack是一个扩展自Vector的类，而Queue是一个接口

16、**Collections类是什么？**

Java.util.Collections是一个工具类仅包含静态方法，它们操作或返回集合。它包含操作集合的多态算法，返回一个由指定集合支持的新集合和其它一些内容。这个类包含集合框架算法的方法，比如折半搜索、排序、混编和逆序等

17、**Comparable和Comparator接口是什么？**

如果我们想使用Array或Collection的排序方法时，需要在自定义类里实现Java提供Comparable接口。

Comparable接口有compareTo(T OBJ)方法，它被排序方法所使用。我们应该重写这个方法，如果“this”对象比传递的对象参数更小、相等或更大时，它返回一个负整数、0或正整数。

但是，在大多数实际情况下，我们想根据不同参数进行排序。

比如，作为一个CEO，我想对雇员基于薪资进行排序，一个HR想基于年龄对他们进行排序。这就是我们需要使用Comparator接口的情景，因为Comparable.compareTo(Object o)方法实现只能基于一个字段进行排序，我们不能根据对象排序的需要选择字段。

Comparator接口的compare(Object o1, Object o2)方法的实现需要传递两个对象参数，若第一个参数比第二个小，返回负整数；若第一个等于第二个，返回0；若第一个比第二个大，返回正整数。

**Comparable和Comparator接口有何区别？**

Comparable和Comparator接口被用来对对象集合或者数组进行排序。Comparable接口被用来提供对象的自然排序，我们可以使用它来提供基于单个逻辑的排序。

Comparator接口被用来提供不同的排序算法，我们可以选择需要使用的Comparator来对给定的对象集合进行排序。

18、**我们如何对一组对象进行排序？**

如果我们需要对一个对象数组进行排序，我们可以使用Arrays.sort()方法。如果我们需要排序一个对象列表，我们可以使用Collection.sort()方法。

两个类都有用于自然排序（使用Comparable）或基于标准的排序（使用Comparator）的重载方法sort()。Collections内部使用数组排序方法，所有它们两者都有相同的性能，只是Collections需要花时间将列表转换为数组

19、**当一个集合被作为参数传递给一个函数时，如何才可以确保函数不能修改它？**

在作为参数传递之前，我们可以使用Collections.unmodifiableCollection(Collection c)方法创建一个只读集合，这将确保改变集合的任何操作都会抛出UnsupportedOperationException。

20、**我们如何从给定集合那里创建一个synchronized的集合？**

我们可以使用Collections.synchronizedCollection(Collection c)根据指定集合来获取一个synchronized（线程安全的）集合

21、**集合框架里实现的通用算法有哪些？**

Java集合框架提供常用的算法实现，比如排序和搜索。Collections类包含这些方法实现。大部分算法是操作List的，但一部分对所有类型的集合都是可用的。部分算法有排序、搜索、混编、最大最小值。

22、**与Java集合框架相关的有哪些最好的实践？**

（1）根据需要选择正确的集合类型。比如，如果指定了大小，我们会选用Array而非ArrayList。如果我们想根据插入顺序遍历一个Map，我们需要使用TreeMap。如果我们不想重复，我们应该使用Set。

（2）一些集合类允许指定初始容量，所以如果我们能够估计到存储元素的数量，我们可以使用它，就避免了重新哈希或大小调整。

（3）基于接口编程，而非基于实现编程，它允许我们后来轻易地改变实现。

（4）总是使用类型安全的泛型，避免在运行时出现ClassCastException。

（5）使用JDK提供的不可变类作为Map的key，可以避免自己实现hashCode()和equals()。

（6）尽可能使用Collections工具类，或者获取只读、同步或空的集合，而非编写自己的实现。它将会提供代码重用性，它有着更好的稳定性和可维护性。

三、网络

1、Socket使用：

              ServerSocket s = new ServerSocket(8189);

                 Socket incoming = s.accept();

                 try{

                     InputStream inStream = incoming.getInputStream();

                      OutPutStream outStream = incoming.getOutPutStream();

                      Scanner in = new Scanner(inStream);

                      PrintWriter out = new PrintWriter(outStream,true);

                      ...

                      while(in.hasNext())

                      {

                         in.nextLine();

                      }

                 }

                 finally{

                     incoming.close();

                 }

     正常情况下服务端是每来一个请求就起一个线程去处理的：

               ServerSocket s = new ServerSocket(8189);

                 while(true)

                 {

                     Socket incoming = s.accept();

                      Runnable r = new ThreadedHandler(incoming);

                      Thread t = new Thread(r);

                      t.start();

                 }

                 class ThreadedHandler implements Runnable

                 {

                    构造器...

                     try{

                     InputStream inStream = incoming.getInputStream();

                      OutPutStream outStream = incoming.getOutPutStream();

                      Scanner in = new Scanner(inStream);

                      PrintWriter out = new PrintWriter(outStream,true);

                      ...

                      while(in.hasNext())

                      {

                         in.nextLine();

                      }

                 }

                 finally{

                     incoming.close();

                 }

                 }

2、发送邮件：也是用的套接字，具体略

3、建立URL链接：

    （1）URL/URI：统一资源定位符/统一资源标识符（URL是URI的特例），URI不包含任何访问资源的方法，它的作用就是解析；URL可以打开一个到达资源的流，比如http、https、ftp、本地文件等。

    （2）URlConnection

    （3）提交表单数据：

         GET方式：只需将参数加在URL尾部即可，但对请求中包含的字符数做了限制；

         Post方式：不需要在URL加任何参数，将name/value写入到流中即可

4、高级套接字编程

   （1）可设置套接字超时

   （2）可中断套接字(线程不会阻塞而是以抛异常的形式结束)：使用SocketChannel来实现 SocketChannel channel = SocketChannel.open(new InetSocketAddress(host,port));

   （3）半关闭：当客户端发一个请求给服务器时，服务端必须能确定该请求何时结束。使用半关闭可以通过关闭一个套接字的输出流来表示发给服务端的请求数据已结束（socket.shutdownOutput()），但必须保留输入流来读取服务器返回的信息；（只适用于一站式的服务）

四、数据库编程

1、JDBC看下代码实现。。

2、事务？？？？

3、LDAP？？

五、**分布式对象**

1、使用代理的**远程方法调用**，三种方式：Java RMI，CORBA，SOAP

   SOAP：完全独立于语言，只需提供一个接口描述，以说明你的对象能够处理的方法的签名和数据类型，接口描述是用一特殊语言编写的对于SOAP而言是WSDL；（主要用于WEB系统，如果所有通信对象都是JAVA实现的则使用RMI），SOAP使用XML具有可阅读和有助于查错的特点，但处理XML也是一个重要的性能瓶颈。

2、RMI：

   （1）存根与参数编组：当客户端调用一个远程方法时实际上是调用的代理对象上的一个普通方法，这个代理对象就是存根；（存在于客户端）

        参数编组的目的是将参数转换成适合在虚拟机间进行传递的的格式：将远程方法所需的参数打包成一组字节（数字以大尾数字节顺序发送，对象以序列化形式

          存根方法构造的信息块：

             被调用的远程对象标识符；被调用的方法描述；被编组后的参数；

          服务器端接收到之后：

             反编组参数；定位要调用的对象；调用所需的方法；捕获返回值或者产生的异常，并对它编组；将返回值编组并打包送回给客户端存根；

   （2）远程对象会像本地对象一样垃圾回收，采用引用计数制（循环引用需要程序员自己解决打破循环）

   （3）动态类加载：当一个远程对象作为远程方法的参数或返回值，传给另一个参数时，该程序必须拥有此远程对象的类文件，加入返回的是一个子类，客户端只有父类却没有子类时就需要类加载器能够从服务器加载所需的类（类加载器，安全管理器）

   （4）配置远程方法调用：首先远程对象的所有接口必须实现Remote接口；服务器类通常继承自UnicastRemoteObject或RemoteServer（UnicastRemoteObject继承自抽象类RemoteServer，并且是实体类，可以直接使用无需另写代码）

        有时服务器类已经继承了其他的类，这时要亲自初始化服务器对象，并将它们传给静态的exportObject方法，可以这样：UnicastRemoteObject.export(this,0);0表示任意合适的端口都可用来监听客户连接；

   （5）存根类生成：从jdk5.0开始可自动生成，之前的版本：rmic -v1.2 ProductImpl

   （6）定位服务器对象：要访问服务器上的一个远程对象时客户端首先需要一个本地的存根对象，可是客户端如何对该存根发出请求呢，最普通的方法就是调用服务端的远程方法以返回值的方式取得存根对象，这就成了先有鸡还是先有蛋的问题。RMI类库提供了自举注册服务来定位第一个服务器对象。

        //Server：

          ProductImpl p1 = new ProductImpl("Blackwell Toaster");

          Context namingContext = new InitialContext();

          namingContext.bind("rmi:toaster",p1);

          //client

          Product p = (Product) namingContext.loopup("rmi://yourserver.com/toaster");

          启动服务器：

          UNIX：

          rmiregistry &

          java ProductServer &

          Windows:

          start rmiregistry

          start java ProductServer

          客户端列举远程对象：NamingEnumration<NameClassPair> e = namingContext.list("rmi:");

          一般客户端程序应该安装一个安全管理器：System.setSecurityManager(new RMISecurityManagerRMI);

          客户端程序部署之后它需要获得许可才能加载存根类，要允许客户端连接RMI注册表以及服务器对象必须提供一个策略文件（将Java.security.policy属性设为策略文件名字）

3、远程方法中的参数传递：

    注意：（1）存根只能由实现了远程接口的类生成，而且只有接口中指明的方法才会出现在存根类中，如果子类没有实现远程接口但父类实现了，传递给远程方法的是子类的对象，那么只能访问父类的方法。

           （2）存根对象的equals和hashCode方法只查看服务器对象的位置，只要它们指向相同的服务器对象equals就认为这两个存根相同，指向不同服务器对象存根不可能相同，即使两个对象有一样的内容；同样，散列码只能通过对象的标识符来计算。

            （3）存根没有Clone方法（不要指望通过存根的clone来复制远程对象）。（定义clone时就要求不抛出CloneNotSupportedException以外的异常）

            （4）服务器对象激活：激活机制允许延迟构造服务器对象，仅当至少有一个客户端调用服务器对象上的远程方法时才去真正构造该服务器对象：

                                1）继承Activatable类

                                     2）在服务器类的构造器中放入Activatable.export(this,id,0);

4、CORBA？？？

5、远程方法调用与SOAP：需要深入研究下？？

   WEB服务有连个部分：1）一个通过将简单对象访问协议（SOAP）作为传输层协议就可以被访问到的服务器；

                      2）一个采用Web服务描述语言（WSDL）格式的服务描述文件；

六、安全

    Java提供了三种确保安全的机制：

     （1）语言设计特性，例如对数组的边界进行检查，只进行合法的类型转换等；

     （2）访问控制机制，控制代码能够执行的功能，比如文件访问网络访问等；

     （3）代码签名，利用该特性代码的作者就能够用标准的加密算法来表明Java代码的身份。

     无论是加载类的默认机制还是自定义的类加载器，都需要与负责控制运行的安全管理器类协同工作。

1、类加载器：引导类加载器：负责加载系统类，是虚拟机的一部分

             扩展类加载器：负责从jre/lib/ext加载标准的扩展

               系统类加载器：加载应用类

    注意：（1）类加载器有一种父子关系，除了引导类加载器外，每个类加载器都有一个父类加载器，并且只有在父类加载器失败时才加载该给定的类；

           （2）在一个虚拟机中可以存在两个包名和类名相同的类，类是由它的全名和类加载器来确定的（这项技术在加载来自多处的代码时很有用）；

            （3）编写自己的类加载器：只需继承ClassLoader类，然后覆盖findClass(String className);（ClassLoader超类的loadClass方法用于将类的加载委托其父类加载器去进行，只有当该类尚未加载并且父类加载器也无法加载该类时，才调用findClass方法）；

                                     实现该方法：1）为来自本地文件系统或者其他来源的类加载其字节码；

                                                       2）调用ClassLoader超类的defineClass方法向虚拟机提供字节码；

2、字节码校验：当类加载器将新加载的Java平台的字节码传递给虚拟机时，这些字节码首先要接受校验器的校验，校验有明显破坏性的操作，除系统类外所有类都要被校验（不过可以用-noverify钝化校验）

            校验：变量要在使用前初始化；

                     方法调用与对象引用类型间要匹配；

                      访问私有数据和方法的规则没有被违反；

                      对本地变量的访问都在运行时堆栈内；

                      运行时堆栈没有溢出；

     注意：由Java编译器生成的类文件总是可以通过校验的，但是可以手工的创建或修改一个类文件（用十六进制编辑器）

3、安全管理器（控制访问权限）：一旦一个类被某个类加载器加载到虚拟机中，并由校验器校验通过之后Java平台的第三种安全机制就会启动：安全管理器。

    安全管理器负责的操作：

     1）当前线程是否能够创建一个新的类加载器；

     2）当前线程是否能够终止虚拟机的运行；

     3）某个类是否能访问另一个类的成员；

     4）当前线程是否能访问本地文件；

     5）当前线程是否能打开到达外部主机的Socket链接；

     6）某个类是否能启动打印作业；

     7）某个类是否能够访问AWT事件队列；

     8）当前线程是否可信任以打开一个顶层窗口；

     注意：1）在运行Java应用程序时默认设置是不安装安全管理器的，可以调用System类的静态方法setSecurityManager方法来安装一个特定的安全管理器，一旦安装了，如果再安装第二个就当第一个同意被替换时第二个才能成功安装

           2）安全策略文件

4、数字签名

Effective java篇：

1、静态工场方法代替构造器

   优点：

  （1）静态工场方法不同于构造器第一大优势是有名称；

  （2）第二大优势是可以不用每次都创建新的对象（利于编写实例受控的类）；

  （3）第三大优势是可以返回原类型的任何子类型；

  （4）第四：创建参数化类型实例时，它使代码变得更加简洁；

       普通用法：Map<String,List<String>> m = new HashMap<String, List<String>>();

                  需要两次提供参数类型；

        静态工场方式：public static <K,V> HashMap<K,V> newInstance(){

                          return new HashMap<K,V>();

                      }

   缺点：

  （1）类如果不含有公有的或者受保护的构造器就不能被子类化；

  （2）与其他静态方法实际上没有区别；

2、遇到多个构造器参数时要考虑用构建器

   静态工场和构造器有个共同的缺点：它们都不能很好的扩展到大量的可选参数。

  （1）当有许多参数时，重叠构造器代码会很难写；

  （2）用javabeans模式时用setter方法来设置每个参数值，弥补了重叠构造器的缺点，但因为构造过程被

       分到了几个调用中，所以会导致javabean可能处于不一致状态，需要额外的工作来保证线程安全；

  （3）参数多时最好使用builder模式：让客户端利用所有必要的参数构造器（或静态工场）得到一个builder对象

       然后客户端在builder对象上调用类似于setter的方法来设置每个相关的可选参数，最好客户端调用无参的

        build方法来生成不可变对象；例如：NutritionFacts cocaCola = new NutritionFacts.Builder(240,8).

        calories(100).sodium(35).carbohydrate(27).build();

3、用私有构造器或者枚举类型强化Singleton属性

  （1）

Java虚拟机篇：

1、动态绑定

2、

面试题：

1、transient和volatile是java关键字吗？

如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。例如：

class T

{

      transient int a;  //不需要维持

      int b;  //需要维持

}

这里，如果T类的一个对象写入一个持久的存储区域，a的内容不被保存，但b的将被保存。

volatile修饰符告诉编译器被volatile修饰的变量可以被程序的其他部分改变。在多线程程序中，有时两个或更多的线程共享一个相同的实例变量。考虑效率问题，每个线程可以自己保存该共享变量的私有拷贝。实际的变量副本在不同的时候更新，如当进入synchronized方法时。 用strictfp修饰类或方法，可以确保浮点运算（以及所有切断）正如早期的Java版本那样准确。切断只影响某些操作的指数。当一个类被strictfp修饰，所有的方法自动被strictfp修饰。

strictfp的意思是FP-strict，也就是说精确浮点的意思。在Java虚拟机进行浮点运算时，如果没有指定strictfp关键字时，Java的编译器以及运行环境在对浮点运算的表达式是采取一种近似于我行我素的行为来完成这些操作，以致于得到的结果往往无法令你满意。而一旦使用了strictfp来声明一个类、接口或者方法时，那么所声明的范围内Java的编译器以及运行环境会完全依照浮点规范IEEE-754来执行。因此如果你想让你的浮点运算更加精确，而且不会因为不同的硬件平台所执行的结果不一致的话，那就请用关键字strictfp。

你可以将一个类、接口以及方法声明为strictfp，但是不允许对接口中的方法以及构造函数声明strictfp关键字，例如下面的代码：

strictfp interface A {}

public strictfp class FpDemo1 {

      strictfp void f() {}

}

错误的使用方法

interface A {

    strictfp void f();

}

public class FpDemo2 {

    strictfp FpDemo2() {}

}

一旦使用了关键字strictfp来声明某个类、接口或者方法时，那么在这个关键字所声明的范围内所有浮点运算都是精确的，符合IEEE-754规范

的。例如一个类被声明为strictfp，那么该类中所有的方法都是strictfp的。

2、抽象类和接口有什么区别？

（1.abstract class 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。

（2.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），所有的成员方法都是abstract的。

（3.abstract class和interface所反映出的设计理念不同。其实abstract class表示的是"is-a"关系，interface表示的是"like-a"关系。

（4.实现抽象类和接口的类必须实现其中的所有方法。抽象类中可以有非抽象方法。接口中则不能有实现方法。

（5.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。

（6.抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。

（7.接口中的方法默认都是 public,abstract 类型的。

3、能说一下java的反射(reflection)机制吗？

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

Java反射机制主要提供了以下功能： 在运行时判断任意一个对象所属的类；在运行时构造任意一个类的对象；在运行时判断任意一个类所具有的成员变量和方法；在运行时调用任意一个对象的方法；生成动态代理。

**（1. 得到某个对象的属性**

**public Object getProperty(Object owner, String fieldName) throws Exception {**

**Class ownerClass = owner.getClass();  //得到该对象的Class**

**Field field = ownerClass.getField(fieldName); //通过Class得到类声明的属性**

**Object property = field.get(owner); //通过对象得到该属性的实例，如果这个属性是非公有的，这里会报IllegalAccessException**

**return property;**

**}**

**（2. 得到某个类的静态属性**

**public Object getStaticProperty(String className, String fieldName)**

**throws Exception {**

**Class ownerClass = Class.forName(className); //首先得到这个类的Class**

**Field field = ownerClass.getField(fieldName); //通过Class得到类声明的属性**

**Object property = field.get(ownerClass); //这里和上面有些不同，因为该属性是静态的，所以直接从类的Class里取**

**return property;**

**}**

**（3. 执行某对象的方法**

**public Object invokeMethod(Object owner, String methodName, Object[] args) throws Exception {**

**Class ownerClass = owner.getClass(); //首先还是必须得到这个对象的Class**

**Class[] argsClass = new Class[args.length];**

**for (int i = 0, j = args.length; i < j; i++) {**

**argsClass[i] = args[i].getClass();**

**}                                     //配置参数的Class数组，作为寻找Method的条件**

**Method method = ownerClass.getMethod(methodName,argsClass); //通过methodName和参数的argsClass（方法中的参数类型集合）数组得到要执行的Method**

**return method.invoke(owner, args); //执行该Method.invoke方法的参数是执行这个方法的对象owner，和参数数组args，可以这么理解：owner对象中带有参数args的method方法。返回值是Object，也既是该方法的返回值**

**}**

**（4. 执行某个类的静态方法**

**public Object invokeStaticMethod(String className, String methodName,**

**Object[] args) throws Exception {**

**Class ownerClass = Class.forName(className);**

**Class[] argsClass = new Class[args.length];**

**for (int i = 0, j = args.length; i < j; i++) {**

**argsClass[i] = args[i].getClass();**

**}**

**Method method = ownerClass.getMethod(methodName,argsClass);**

**return method.invoke(null, args);  //基本的原理和实例3相同，不同点是最后一行，invoke的一个参数是null，因为这是静态方法，不需要借助实例运行**

**}**

**（5. 新建实例**

**public Object newInstance(String className, Object[] args) throws Exception {**

**Class newoneClass = Class.forName(className);**

**Class[] argsClass = new Class[args.length];**

**for (int i = 0, j = args.length; i < j; i++) {**

**argsClass[i] = args[i].getClass();**

**}**

**Constructor cons = newoneClass.getConstructor(argsClass);**

**return cons.newInstance(args);**

**}**

**（6. 判断是否为某个类的实例**

**public boolean isInstance(Object obj, Class cls) {**

**return cls.isInstance(obj);**

**}**

（7. 得到数组中的某个元素

public Object getByArray(Object array, int index) {

     return Array.get(array,index);

}

4、在java中怎样实现多线程？

5、线程的四种状态:New（新生），Runnable（可运行，包括了已经在运行的和具备运行条件的），Blocked（被阻塞），Dead（死亡）

6、线程的优先级：可以调用 Thread 类的方法 getPriority() 和 setPriority()来存取线程的优先级，线程的优先级界于1(MIN\_PRIORITY)和10(MAX\_PRIORITY)之间，缺省是5(NORM\_PRIORITY)。

。。。。。。。。

7、你用过哪种设计模式？

模板模式;

单例模式:构造方法私有

工厂模式：为创建对象定义一个接口，让子类决定初始化哪个类，工厂方法让一个类的初始化延迟至子类；

     public interface Product{}

     public class ConcreteProdcut implements Product{}

     public interface Factroy{

          Product createProduct();

     }

     public class ConcreteFactory implements Factroy{

          public Product createProduct(){

               return new ConcreteProdcut();

          }

     }

     public class Client{

          private Factroy factroy;

          public Client (Factroy factroy){

               this.factroy=factroy;

          }

          public void doSomeThing(){

               Product product = factroy.createProduct();

               ....

          }

          public static void main(String args[])

          {

               Client client = new Client(new ConcreteProdcut());

               client.doSomeThing();

          }

     }

原型模式（prototype）：使用原型实例指定将要创建的对象类型，通过copy这个实例创建新的对象；

    浅copy

     深度copy

     1）实现java.lang.Cloneable接口来实现clone；需递归调用属性对象的clone方法；

     2）序列化，实现java.io.Serializable接口：

         public class DeepCopyBean implements Serializable{

               private String objectField;

               private int primitiveField;

               public DeepCopyBean deepCopy(){

                    try{

                         ByteArrayOutputStream buf = new ByteArrayOutputStream();

                         ObjectOutputStream o = new ObjectOutputStream(buf);

                         o.writeObject(this);

                         ObjectInputStream in = new ObjectInputStream(new ByteArrayInputStream(buf.toByteArray()));

                         return (DeepCopyBean) in.readObject();

                    }catch(IOException e)

                    {...

                    }catch(ClassNotFoundException e)

                    {...}

               }

          }

在Java中，对象的序列化可以通过实现两种接口来实现，若实现的是Serializable接口，则所有的序列化将会自动进行，若实现的是Externalizable接口，则没有任何东西可以自动序列化，需要在writeExternal方法中进行手工指定所要序列化的变量，这与是否被transient修饰无关

IFormatter

XmlSerializer

SoapFormatter

装饰器模式（Decorator）：比如Java的IO

代理模式（Proxy）：有时需要对实际访问的对象进行控制，把这层访问控制封装成一个新的代理对象来代替实际对象；

    1）代理一些开销很大的对象，这样能快速返回，进行其他操作，只有在真正需要时才去实例化；

     2）安全控制，同步控制，缓存处理结果，缓存初始化开销很大的对象统计对象的使用及异常等方面的处理；

     3）分布式对象的访问控制，使得客户端像使用本地对象一样使用分布式对象  （EJB（RMI））

适配器模式（Adapter）：

    public class TwoWayPegAdapter implements IRoundPeg,IPeg{

         private IPeg squarePeg;

          private IRoundPeg roundPeg;

          public TwoWayPegAdapter(IPeg squarePeg){

              this.squarePeg=squarePeg;

          }

          public TwoWayPegAdapter(IRoundPeg roundPeg){

              this.roundPeg = roundPeg;

          }

          public void insertIntoRoundHole(){

              squarePeg.insertIntoHole();

          }

          public void insertIntoHole(){

              roundPeg.insertIntoRoundHole();

          }

     }

外观模式（Facade）：只和你的朋友说话，不要和陌生人说话；

    助理准备晚餐：prepareDinner（）

     老板讲话：address（）

     助理结束晚餐：endDinner（）

组合模式

策略模式

状态模式

观察者模式

8、MVC（Modle，Viewer，Controller）

9、请说一下java中为什么要引入内部类？还有匿名内部类？

10、请说一下final，finally和finalize的区别

    final 用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承；

     finally是异常处理语句结构的一部分，表示总是执行；

     finalize是Object类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等；

11、请说一下HTTP请求的基本过程

12、java中存在内存泄漏问题吗？请举例说明？

    会。。1）语法问题，虽然对象没用了但是一直被某个变量引用着

           2）对于非引用类对象，垃圾回收机制就不能起作用，比如打开过多的数据库连接

    内存泄露：无法释放已申请的内存（占着茅坑不拉屎）

     内存溢出：申请不到足够的内存（内存不够用了）

13、请说一下java中的内存回收机制所采用的算法

14、CORBA是什么?用途是什么?

15、请说一下System.gc()函数的作用。什么时候可以调用垃圾回收器？

    垃圾回收函数,手动调用的。

    当一个对象停止被活动声明所引用，它就变成了垃圾（garbage）可以被回收重新使用

16、Java异常的基类为java.lang.Throwable，java.lang.Error和java.lang.Exception继承 Throwable

    1) java.lang.Error: Throwable的子类，用于标记严重错误。合理的应用程序不应该去try/catch这种错误。绝大多数的错误都是非正常的，就根本不该出现的。

java.lang.Exception: Throwable的子类，用于指示一种合理的程序想去catch的条件。即它仅仅是一种程序运行条件，而非严重错误，并且鼓励用户程序去catch它。

    2)  Error和RuntimeException 及其子类都是未检查的异常（unchecked exceptions），而所有其他的Exception类都是检查了的异常（checked exceptions）.

checked exceptions: 通常是从一个可以恢复的程序中抛出来的，并且最好能够从这种异常中使用程序恢复。比如FileNotFoundException, ParseException等。检查了的异常发生在编译阶段，必须要使用try…catch（或者throws）否则编译不通过。

unchecked exceptions: 通常是如果一切正常的话本不该发生的异常，但是的确发生了。发生在运行期，具有不确定性，主要是由于程序的逻辑问题所引起的。比如ArrayIndexOutOfBoundException, ClassCastException等。从语言本身的角度讲，程序不该去catch这类异常，虽然能够从诸如RuntimeException这样的异常中catch并恢复，但是并不鼓励终端程序员这么做，因为完全没要必要。因为这类错误本身就是bug，应该被修复，出现此类错误时程序就应该立即停止执行。 因此，面对Errors和unchecked exceptions应该让程序自动终止执行，程序员不该做诸如try/catch这样的事情，而是应该查明原因，修改代码逻辑。

RuntimeException：RuntimeException体系包括错误的类型转换、数组越界访问和试图访问空指针等等。

处理RuntimeException的原则是：如果出现RuntimeException，那么一定是程序员的错误。例如，可以通过检查数组下标和数组边界来避免数组越界访问异常。其他（IOException等等）checked异常一般是外部错误，例如试图从文件尾后读取数据等，这并不是程序本身的错误，而是在应用环境中出现的外部错误。

17、普通的类方法是可以和类名同名的，和构造方法唯一的区分就是，构造方法没有返回值

18、守护线程最典型的应用就是 GC

19、存在使i + 1 < i的数吗？

   如果i为int型，那么当i为int能表示的最大整数时，i+1就溢出变成负数了，此时不就<i了吗

20、0.6332的数据类型是：默认为double型，如果为float型需要加上f显示说明，即0.6332f

21、Java的IO操作中有面向字节(Byte)和面向字符(Character)两种方式。

面向字节的操作为以8位为单位对二进制的数据进行操作，对数据不进行转换，这些类都是InputStream和OutputStream的子类。

面向字符的操作为以字符为单位对数据进行操作，在读的时候将二进制数据转为字符，在写的时候将字符转为二进制数据，这些类都是Reader和Writer的子类。

总结：以InputStream（输入）/OutputStream（输出）为后缀的是字节流；

      以Reader（输入）/Writer（输出）为后缀的是字符流。

22、Java接口的修饰符可以为（）

A private     B protected     C final       D abstract

答案：CD（接口的方法默认是public abstract）

接口中不可以定义变量即只能定义常量(加上final修饰就会变成常量)。所以接口的属性默认是public static final 常量，且必须赋初值。

注意：final和abstract不能同时出现。

23、ArrayList list = new ArrayList(20);中的list扩充几次（）

A 0     B 1     C 2      D 3

答案：A

解析：这里有点迷惑人，大家都知道默认ArrayList的长度是10个，所以如果你要往list里添加20个元素肯定要扩充一次（扩充为原来的1.5倍），但是这里显示指明了需要多少空间，所以就一次性为你分配这么多空间，也就是不需要扩充了

24、 常用的对称加密算法有：DES、3DES、RC2、RC4、AES

     常用的非对称加密算法有：RSA、DSA、ECC

     使用单向散列函数的加密算法：MD5、SHA

25、public class NULL {

    public static void haha(){

        System.out.println("haha");

    }

    public static void main(String[] args) {

        ((NULL)null).haha();

    }

}

输出为haha，因为null值可以强制转换为任何java类类型,(String)null也是合法的。但null强制转换后是无效对象，其返回值还是为null，而static方法的调用是和类名绑定的，不借助对象进行访问所以能正确输出。反过来，没有static修饰就只能用对象进行访问，使用null调用对象肯定会报空指针错了

26、class HelloA {

    public HelloA() {

        System.out.println("HelloA");

    }

    { System.out.println("I'm A class"); }

    static { System.out.println("static A"); }

}

public class HelloB extends HelloA {

    public HelloB() {

        System.out.println("HelloB");

    }

    { System.out.println("I'm B class"); }

    static { System.out.println("static B"); }

    public static void main(String[] args) {

        System.out.println("-------main start-------");

        new HelloB();

        new HelloB();

        System.out.println("-------main end-------");

    }

}

输出：

static A

static B

-------main start-------

I'm A class

HelloA

I'm B class

HelloB

I'm A class

HelloA

I'm B class

HelloB

-------main end-------

**对象的初始化顺序：（1）类加载之后，按从上到下（从父类到子类）执行被static修饰的语句；（2）当static语句执行完之后,再执行main方法；（3）如果有语句new了自身的对象，将从上到下执行构造代码块、构造器（两者可以说绑定在一起）**

静态语句块、构造语句块（就是只有大括号的那块）以及构造函数的执行顺序：

27、因为在java里没有引用传递，只有值传递这个值指的是实参的地址的拷贝，得到这个拷贝地址后，你可以通过它修改这个地址的内容（引用不变），因为此时这个内容的地址和原地址是同一地址，但是你不能改变这个地址本身使其重新引用其它的对象，也就是值传递。

28、抽象类遵循的原则：

（1）abstract关键字只能修饰类和方法，不能修饰字段。

（2）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。

（3）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，变量可以初始化或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）。

（4）抽象类中的抽象方法（加了abstract关键字的方法）不能实现。

（5）含有抽象方法的类必须定义成抽象类

29、抽象类和接口的区别，做个总结吧：

（1）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。

（2）abstract class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。

（3）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。

（4）abstract class和interface所反映出的设计理念不同。其实abstract class表示的是"is-a"关系，interface表示的是"has-a"关系。

（5）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。

（6）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

30、java 的transient关键字为我们提供了便利，你只需要实现Serilizable接口，将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中；

    transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口；

     被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化；

    在Java中，**对象的序列化可以通过实现两种接口来实现，若实现的是Serializable接口，则所有的序列化将会自动进行，若实现的是Externalizable接口，则没有任何东西可以自动序列化，需要在writeExternal方法中进行手工指定所要序列化的变量，这与是否被transient修饰无关；**

31、子类没有显示调用父类构造函数，不管子类构造函数是否带参数都默认调用父类无参的构造函数，若父类没有则编译出错

32、Java中是否可以覆盖(override)一个private或者是static的方法？

Java中static方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而static方法是编译时静态绑定的。static方法跟类的任何实例都不相关，所以概念上不适用

33、Java不支持像C++中那样的复制构造函数，这个不同点是因为如果你不自己写构造函数的情况下，Java不会创建默认的复制构造函数

34、在监视器(Monitor)内部，是如何做线程同步的？程序应该做哪种级别的同步？

    监视器和锁在Java虚拟机中是一块使用的。监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块。每一个监视器都和一个对象引用相关联。线程在获取锁之前不允许执行同步代码

35、如何确保N个线程可以访问N个资源同时又不导致死锁？

    使用多线程的时候，一种非常简单的避免死锁的方式就是：指定获取锁的顺序，并强制线程按照指定的顺序获取锁。因此，如果所有的线程都是以同样的顺序加锁和释放锁，就不会出现死锁了

36、Java集合类框架的基本接口有哪些？

37、快速失败(fail-fast)和安全失败(fail-safe)的区别是什么？

Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util包下面的所有的集合类都是快速失败的，而java.util.concurrent包下面的所有的类都是安全失败的。快速失败的迭代器会抛出ConcurrentModificationException异常，而安全失败的迭代器永远不会抛出这样的异常

对于非并发集合来说，在其进行迭代时，例如iterator迭代时，iterator是另起一个线程，若有其他线程（如Collection）进行结构修改（修改了增减了集合中的内容），这个迭代会马上感知到，并且立即抛出 ConcurrentModificationException 异常，而不是迭代完成后才告诉你出错了，引起快速失败。若用iterator进行修改则不会出现这个问题，如iterator.move();

38、HashMap的一些重要的特性是它的容量(capacity)，负载因子(load factor)和扩容极限(threshold resizing)。

39、hashCode()和equals()方法的重要性体现在什么地方？

Java中的HashMap使用hashCode()和equals()方法来确定键值对的索引，当根据键获取值的时候也会用到这两个方法。如果没有正确的实现这两个方法，两个不同的键可能会有相同的hash值，因此，可能会被集合认为是相等的。而且，这两个方法也用来发现重复元素。所以这两个方法的实现对HashMap的精确性和正确性是至关重要的

40、HashMap和Hashtable有什么区别？

?HashMap和Hashtable都实现了Map接口，因此很多特性非常相似。但是，他们有以下不同点：

?HashMap允许键和值是null，而Hashtable不允许键或者值是null。

?Hashtable是同步的，而HashMap不是。因此，HashMap更适合于单线程环境，而Hashtable适合于多线程环境。

?HashMap提供了可供应用迭代的键的集合，因此，HashMap是快速失败的。另一方面，Hashtable提供了对键的列举(Enumeration)。 ?一般认为Hashtable是一个遗留的类

41、数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用Array而不是ArrayList？

?Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。

?Array大小是固定的，ArrayList的大小是动态变化的。

?ArrayList提供了更多的方法和特性，比如：addAll()，removeAll()，iterator()等等。

?对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

42、Comparable和Comparator接口是干什么的？列出它们的区别

Java提供了只包含一个compareTo()方法的Comparable接口。这个方法可以个给两个对象排序。具体来说，它返回负数，0，正数来表明输入对象小于，等于，大于已经存在的对象。

Java提供了包含compare()和equals()两个方法的Comparator接口。compare()方法用来给两个输入参数排序，返回负数，0，正数表明第一个参数是小于，等于，大于第二个参数。equals()方法需要一个对象作为参数，它用来决定输入参数是否和comparator相等。只有当输入参数也是一个comparator并且输入参数和当前comparator的排序结果是相同的时候，这个方法才返回true

43、什么是Java优先级队列(Priority Queue)？

PriorityQueue是一个基于优先级堆的无界队列，它的元素是按照自然顺序(natural order)排序的。在创建的时候，我们可以给它提供一个负责给元素排序的比较器。PriorityQueue不允许null值，因为他们没有自然顺序，或者说他们没有任何的相关联的比较器。最后，PriorityQueue不是线程安全的，入队和出队的时间复杂度是O(log(n))。

44、你了解大O符号(big-O notation)么？你能给出不同数据结构的例子么？

大O符号描述了当数据结构里面的元素增加的时候，算法的规模或者是性能在最坏的场景下有多么好。

大O符号也可用来描述其他的行为，比如：内存消耗。因为集合类实际上是数据结构，我们一般使用大O符号基于时间，内存和性能来选择最好的实现。大O符号可以对大量数据的性能给出一个很好的说明

45、HashSet和TreeSet有什么区别？

HashSet是由一个hash表来实现的，因此，它的元素是无序的。add()，remove()，contains()方法的时间复杂度是O(1)。

另一方面，TreeSet是由一个树形的结构来实现的，它里面的元素是有序的。因此，add()，remove()，contains()方法的时间复杂度是O(logn)。

46、System.gc()和Runtime.gc()

47、Anonymous Inner Class (匿名内部类) 是否可以extends(继承)其它类，是否可以implements(实现)interface(接口)?

匿名的内部类是没有名字的内部类。不能extends(继承) 其它类，但一个内部类可以作为一个接口，由另一个内部类实现。

48、只要一个类是抽象的或是一个接口，那么其子类中的方法都可以使用匿名内部类来实现最常用的情况就是在多线程的实现上，因为要实现多线程必须继承Thread类或是继承Runnable接口

49、Nested Class （一般是C++的说法），Inner Class (一般是JAVA的说法)。Java内部类与C++嵌套类最大的不同就在于是否有指向外部的引用上

50、Collection 和 Collections的区别

Collections是个java.util下的类，它包含有各种有关集合操作的静态方法

Collection是个java.util下的接口，它是各种集合结构的父接口

51、String s = new String("xyz");创建了几个String Object?

两个对象，一个是"xyx",一个是指向"xyx"的引用对象s

52、short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 += 1;有什么错?

short s1 = 1; s1 = s1 + 1;有错，s1是short型，s1+1是int型,不能显式转化为short型。可修改为s1 =(short)(s1 + 1) 。short s1 = 1; s1 += 1正确

**53、sleep() 和 wait() 有什么区别?**

**sleep指线程被调用时，占着CPU不工作，形象地说明为“占着CPU睡觉”，此时，系统的CPU部分资源被占用，其他线程无法进入，会增加时间限制。**

**wait指线程处于进入等待状态，形象地说明为“等待使用CPU”，此时线程不占用任何资源，不增加时间限制。**

54、Set里的元素是不能重复的，那么用什么方法来区分重复与否呢?

Set里的元素是不能重复的，那么用iterator()方法来区分重复与否

55、给我一个你最常见到的runtime exception

ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DOMException, EmptyStackException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException,

ImagingOpException, IndexOutOfBoundsException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NullPointerException, ProfileDataException, ProviderException, RasterFORMatException, SecurityException, SystemException, UndeclaredThrowableException, UnmodifiableSetException, UnsupportedOperationException

56、error和exception有什么区别?

error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。

exception 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况

57、List, Set, Map是否继承自Collection接口?

List，Set是

Map不是，Map继承自AbstractMap

58、abstract的method是否可同时是static,是否可同时是native，是否可同时是synchronized

都不能

59、接口是否可继承接口? 抽象类是否可实现(implements)接口? 抽象类是否可继承实体类(concrete class)?

接口可以继承接口。抽象类可以实现(implements)接口，抽象类是否可继承实体类，**但前提是实体类必须有明确的构造函数**。

60、构造器Constructor是否可被override?

构造器Constructor不能被继承，因此不能重写Overriding，但可以被重载Overloading

61、是否可以继承String类?

String类是final类故不可以继承

62、当一个线程进入一个对象的一个synchronized方法后，其它线程是否可进入此对象的其它方法?

1)：进入此对象的非同步方法

答案：可以

2)：进入此对象的同步方法

答案：不可以

**63、try {}里有一个return语句，那么紧跟在这个try后的finally {}里的code会不会被执行，什么时候被执行，在return前还是后?**

**会执行，在return前执行**

64、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，有相同的hash code

65、swtich是否能作用在byte上，是否能作用在long上，是否能作用在String上?

switch（expr1）中，expr1是一个整数表达式。因此传递给 switch 和 case 语句的参数应该是 int、 short、 char 或者 byte。long,string 都不能作用于swtich

66、strictfp 关键字可应用于类、接口或方法。使用 strictfp 关键字声明一个方法时，该方法中所有的float和double表达式都严格遵守FP-strict的限制,符合IEEE-754规范。当对一个类或接口使用 strictfp 关键字时，该类中的所有代码，包括嵌套类型中的初始设定值和代码，都将严格地进行计算

第11章 晚期优化

JVM中有解释器和编译器

现代JVM逐渐采用分层编译，

第0层：程序解释执行，解释器不开启性能监控功能（Profiling），可触发第1层编译。

第1层：也成为C1（Client）编译，将字节码编译为本地代码，进行简单可靠的优化，如有必要加入性能监控的逻辑。

第2层：也称C2（Server）编译，也是将字节码编译为本地代码，但会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

热点代码有两类：

1.被多次调用的方法

2.被多次执行的循环体

热点代码探测方式：

1.基于采样的热点探测（Sample Based Hot Spot Detection）

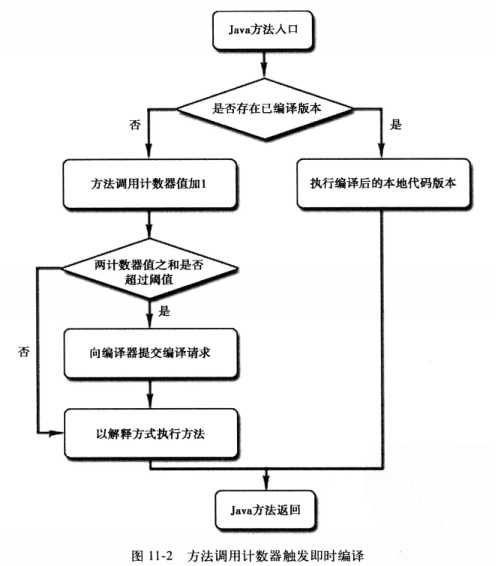
     周期性的检查各个线程的栈顶，如果发现某个方法经常出现在栈顶，那这个方法就是热点方法，这样简单高效，缺点是精确度不高。

2.基于计数器的热点检测（Counter Based Hot Spot Detection）

     为每个方法（甚至代码块）维护一个计数器，麻烦但精确。

HotSpot用的是第二种，为每个方法准备了两个计数器，方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）。

在确定虚拟机参数的前提下，这两个计数器都有一个阈值，当阈值超出时，即触发JIT编译。



默认设置下，计数器并不统计绝对次数，而是相对的执行频率，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给JIT编译，那么这个方法的调用计数器会被减少一半，这称为方法调用计数器的热度衰减（Counter Decay），而这段时间就称为此方法统计的半衰周期（Counter Half Life Time）

回边计数器，用于统计一个方法中循环体代码（回边）执行的次数，在字节码中遇到控制流向后跳转的指令就称为“回边（Back Edge）”， 为了触发OSR（Online Stack Replacement）编译。

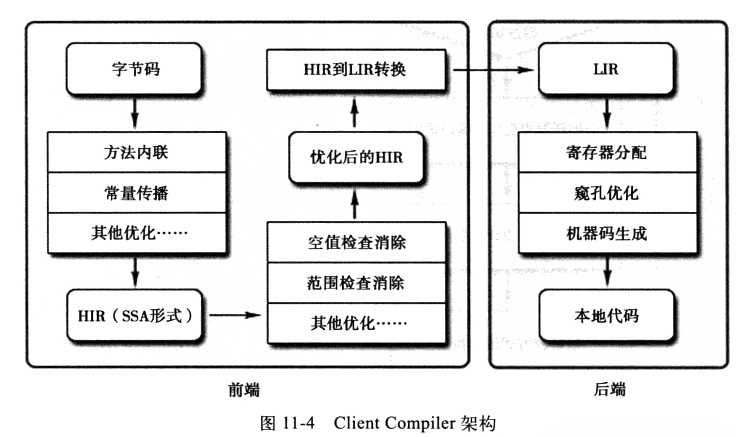
回边计数器没有计数热度衰减，所以它统计的是绝对次数。

Client Compiler 是简单快速的三段式编译器，主要的关注点在局部性的优化，放弃了许多耗时较长的优化。

1.一个平台独立的前端将字节码构造成一种高级中间代码表示（High-Level Intermediate Representation, HIR），HIR使用静态单分配（Static Single Assignment, SSA）的形式表示代码值，这样会使得HIR构造之中和之后的优化动作更容易实现。在HIR之前，编译器会完成如方法内联，常量传播等优化。

2.一个平台相关的后端从HIR中产生低级中间代码表示（Low-Level Intermediate Representation），在此之前会在HIR上完成如空值检查消除，范围检查消除等，为了让HIR达到更高效的代码表示形式。

3.在平台相关的后端使用线性扫描算法（linear scanner Register Allocation）在LIR上分配寄存器，并在LIR上做窥孔（Peephole）优化，然后产生机器代码。



Server Compiler 是高级优化了，能优化的都尽量优化了。

公共子表达式消除

1.公共子表达式消除

即一个表达式E如果已经被计算过了，那么如果从先前的计算到现在E的值都没发生过变化，则E的这次出行就成为了公共表达式。

又根据作用范围不同分为局部和全局的。

2.数组边界检查消除

3.方法内嵌（内联）

因为JAVA的特性，单纯的内联是不能work的，为此引入了类型继承关系分析（Class Hierarchy Analysis， CHA）的技术，这是基于整个应用程序的技术，用于确定在目前已加载的类中，某个接口是否有多于一种的实现，某个类是否存在子类且子类是否为抽象类等信息。

所以编译器在进行内联时，如果是非虚方法，那么直接内联就可以，如果是虚方法，那么通过CHA查询此方法在当前程序下是否有多个版本可供选择，如果只有一个版本的话，可以进行内联，但此时属于激进优化，要预留一个“逃生门”（Guard条件不成立时的Slow Path）的，成为守护内联。所以如果后面加载到了导致继承关系发生变化的类，那么就需要舍弃已编译的代码，退回到解释状态执行，再重新进行编译。

还有“内联缓存”，也类似，用过的先缓存。

4.逃逸分析

基本行为就是分析对象动态作用域，当一个对象在方法里面被定义后，它可能被外部方法所引用，这成为方法逃逸，还有可能被外部线程访问到，称为线程逃逸。

如果能确定一个对象不会逃逸到线程或方法之外，那么就可以进行一些优化：

1.栈上分配

将对象分配在栈上，避免GC，提高效率

2.同步消除

因为变量只在当前线程使用，所以可以消除同步。

3.标量替换

不直接创建整个对象，而创建对象中被使用到的标量（最小单位，int, long等）

Spring：

1、org.springframework.beans和org.springframework.context

   一个BeanFactory可以用接口org.springframework.beans.factroy.BeanFactory表示，这个接口有多个实现，最常用的是org.springframework.beans.factroy.xml.XmlBeanFactory(ApplicationContext是BeanFactory的子类)