1.你怎么理解面向对象的编程思想：

·面向过程是一种以过程为中心的编程思想，可以清晰的知道过程中的每个步骤，面向对象则是以对象为主体。

·面向对象分析OOA：

·面向对象设计OOD：

·面向对象编程OOP：是一种计算机编程的范式，利用对象和对象之间的相互作用来设计程序

·面向对象的思想是把整个世界看成是由各种对象组成的，这些对象具有属性和行为，OC中称之为属性和方法，从最简单的整数到复杂的飞机等均可看作对象，他不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

·在面向对象的思想中，具体的事物是对象，具有相同的或相似的性质的对象的抽象或者说描述就是类，因此，对象的抽象是类，类的具体化就是对象，也可以说类的实例就是对象。

·对象的抽象是类，类的抽象是元类

2.重写和重载的区别：

·重写的规则：

重写方法的参数列表必须完全与被重写的方法的参数列表相同，否则是重载

重写方法的访问修饰符一定要大于被重写的方法的访问修饰符public🡪protect🡪default🡪private

重写的方法的返回值必须和被重写的方法的放回值一致

重写的方法所抛出的异常必须和被重写的方法所抛出的异常一致，或者是其子类

被重写的方法不能为private，否则在其子类中只是新定义了一个方法，并没有重写

静态方法不能被重写为非静态的方法（编译错误）

·重载的规则：

在使用重载的时候只能通过相同的方法名，不同的参数名，不同的形参实现，不同的参数类型，不同的参数个数，不同的参数顺序

不能通过访问权限，返回类型，抛出异常进行重载

方法的异常类型和数目不会对重载造成影响

·重写和重载的区别：

重写必须继承，重载不用

重写的方法名，参数数目相同，参数类型兼容，重载方法名相同，参数列表不同

重写的方法修饰符大于父类的方法，重载和修饰符无关

重写不可以抛出父类没有抛出的一般异常，可以抛出运行时异常

3.面向对象的三大特性：封装、继承、多态（重写，重载）

·封装：就是把客观的事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象进行使用，对不可信的进行信息隐藏

·继承：可以实现现有类的部分功能，并在无需编写原类的情况下对类的功能进行扩展，通过集成的类称为子类或派生类，被继承的类称为基类或父类或超类，继承的过程，就是从一般到特殊的过程。

·多态：是允许你将父对象设置成为一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给他的子对象的特性以不同的方式运作。实现多态有两种方式：覆盖和重载。覆盖是指子类重新定义父类中的函数的做法，重载是指允许存在多个同名函数。

·多态的作用：我们知道，封装可以实现细节，使代码模块化，继承可以扩展已有的代码模块，他们的目的就是—代码重用，而多态则是为了实现另一个目的—接口重用，多态的作用就是为了在继承和派生的时候，保证使用“家谱”中任意一类的实例的某一属性时的正确调用。

4.finally和finallize的区别：

·final：他的作用有:定义变量、定义方法的参数、定义方法、定义类

Final修饰基本类型就表示是一个不可变的常量

Final修饰的是一个对象就表示这个变量被赋值的引用时不可变的，不是对象不可变

被final修饰的变量必须初始化：初始化方式有以下几种：

在定义的时候初始化

Final变量可以在初始化块中进行初始化，不可以在静态初始化块中进行初始化

Final还可以在类的构造器中初始化，但是静态final是不可变的

具有private访问权限的方法也可以增加final修饰，但无法对其进行重写。

Final类的所有方法不能被重写，但是类中的属性可以改变

·finally语句：他只能用在try{}catch(){}方法中，并且附带一个语句块，表示这段语句最终总是被执行

Finally的作用：

程序抛出异常

执行了finally的语句块，return，break，continue都不能阻止finally的执行，finally在return，continue，break之前执行

·finallize：他是一个方法，属于java.lang.Object类，他是GC(garbage collector)垃圾处理机制中的一部分，作用如下：

Finallized()方法是在GC清理它所从属的对象时被调用的，如果执行他的过程中抛出了无法捕获的异常，GC将终止对该对象的清理，并且该异常会被忽略，直到下一个GC开始清理这个对象的时候，他的finallize会再次被调用。

由于finallize()属于Object类，所以所有类都有这个方法，Object的任意子类都可以重写这个方法，在其中释放系统资源或者做其他的工作，如关闭输入输出流。

5.StringBuffer和String的区别：

StringBuffer的底层是利用什么实现的？

-128~127

·简单来说，就是一个变量和一个常量之间的关系，StringBuffer对象的内容可以进行修改，而String 对象一旦创建之后就不能修改，重新赋值实际上是两个对象。

·StringBuffer和String的内部实现方式不同，StringBuffer在进行字符串处理的时候，不生成新的对象，在内存使用上要优于String，所以在实际使用时，经常需要对一个字符串进行修改，使用StringBuffer更加合适

·String在String类中没有用来改变已有字符串中某个字符的方法，所以在jdk文档中成String类的对象是不可变的。然而，不可改变的字符串具有一个很大的优点：编译器可以把字符串设为共享的。

·StringBuffer类属于一种辅助类，可预先分配指定长度的内存块建立一个字符缓冲区，这样使用StringBuffer的append方法追加字符比String用+ 追加字符到一个已存在的字符串的后面性能更好，因为使用+在追加字符的时候字符串对象都需要找到一个新的内存空间来容纳更大的字符串，这无疑是一个非常消耗时间的操作，添加多个字符也就意味着要一次又一次的对字符串重新分配内存，使用StringBuffer就可以避免这个问题

·StringBuffer是线程安全的，在多线程中也可以很方便的使用，但是程序的执行效率相对来说就要慢一些。

·StringBuffer中的方法偏重于对字符串的变化，也是StringBuffer和String的区别

6.finally的代码一定会执行吗？

·不一定，实验证明:

当与finally对应的trycatch语句块不执行的时候，finally不执行

当与finally对应的trycatch语句块执行的时候，但执行了System.exit(0),finally不执行

当一个线程在执行trycatch语句的时候被打断或者终止，与之对应的finally语句可能不会执行

电脑死机或断电，finally语句不会执行

7.异常的分类：

Java.lang.Object

┗java.lang.Throwable

┗java.lang.Exception

Java.lang.Error

·java.lang.Throwable是所有异常的根，Error是错误

·Exception:一般分为Checked异常和Runtime异常，所有RuntimeException类及其子类的实例被称为Runtime异常，不属于该范畴的异常则被称为CheckedException

·Checked异常：只有java提供了Checked异常，java认为Checked异常都是可以被处理的异常，如果程序没有处理Checked异常，该程序在编译的时候就无法进行编译，这体现了java的设计哲学：没有完善错误处理的代码根本没有机会执行。对Checked异常的处理方法有两种：

使用try{}catch(){}

当前方法不知道如何处理，则在定义该方法时声明抛出该异常

·RuntimeException异常：Runtime是如除0或下标越界等，其产生频繁，处理麻烦，若显示申明或者捕获将会对程序的可读性和运行效率影响很大，所有系统自动检测并交给缺省的异常处理程序，常用的RuntimeException的子类有：

Java.lang.ArithmeticException

Java.lang.ArrayStoreException

Java.lang.ClassCastException

Java.lang.IndexOutOfBoundsException

Java.lang.NullPointerException

·Error：当程序发生不可控的错误的时候，通常做法是通知用户并终止程序的运行，与异常不同的是Error及其子类的对象不应该被抛出，Error是Throwable的子类，代表编译时间和系统错误，用于指示合理应用程序不应该试图捕获的严重问题。Error是由java虚拟机生成并输出的，包括动态链接失败，虚拟机错误等，程序对其不做处理。

8.collection和collections的区别：

Java.util.Collection是一个集合接口，它提供了对集合对象进行基本操作的通用接口方法：

┏Collection

│ ┣List

│ ┣LinkedList

│ ┣ArrayList

│ ┣Vector

│ ┣Stack

│ ┗Set

Java.util.Collections是一个包装类。他包含有各种有关集合操作的静态多态方法。次此类不能实例化，就想是一个工具类，服务于java的Collection框架

9.ArrayList和LinkedList的区别：

ArrayList是实现了基于动态数组的数据结构，LinkedList是基于链表的数据结构

对于随机访问的get和Set，ArrayList性能由于LinkedList，因为LinkedList要移动指针

对于新增和删除操作add和remove，LinkedList比较占优势，因为Arraylist要移动数据

对于新增和删除操作，add和remove，LinkedList比较占优势，因为Arraylist要移动数据

10.ArrayList和Vector的区别：

都是用数组实现的

Vector是多线程安全的，而ArrayList不是，Vector在方法中用到很多synchronize进行修饰，所以Vector在效率上无法与ArrayList进行相比

两个都是采用线性连续空间的存储元素，但是当空间不足的时候，两个类的增加方式是不一样的，Vector增加原来空间的一倍ArrayList增加原来空间的50%。

Vector可以设置增长因子，而ArrayList不可以

11.Properties是否是线程安全的：

Properties是继承的Hashtable，Hashtable是线程安全的，且Properties中的方法很多使用synchronize进行修饰，所以Properties是线程安全的。

\*12.HashMap的实现原理：

·Hashmap 的内部结构：和HashSet 相同，内部是由一个Entry类型数组进行存储的，Entry中有三个属性分别为：key，value，next。Next代表下一个Entry，当我们put一个元素的时候，HashMap首先找到这个键所对应下标，然后遍历，如果此下标无元素，则将此Entry放进此下标。如果此下标有元素，遍历所有的元素后，如发现有相同的Key则替换，如果没有key则将元素加入到链头。

13.HashMap的HashTable的区别：加载因子，实现接口，null

·HashTable继承自Dictiionary而HashMap继承自AbstractMap

·HashTable中的put方法：方法是同步的，key、value不能为null，方法调用了key的hashcode方法，如果key==null，就会抛出空指针异常

·HashMap中的put方法：方法是非同步的，方法允许key==null ，方法并没有对value进行任何调用，所以value可以为null

\*14.treeMap的实现：concurrentMap

·treemap的实现实际就是红黑树的数据结构，也就是说一颗子平衡的排序二叉树

·红黑树：是一种自平衡的排序二叉树，树中每个节点的值，都大于或等于在他的左子树中的所有节点的值，并且小于或等于他在中的所有节点的值，这确保红黑树在运行时可以快速的在树中查找和定位的所需节点，对于Treemap而言，由于他的底层采用红黑树来保存集合中的Entry，这意味着TreeMap添加元素，取出元素的性能都比HashMap低。

16.实现多线程的方式：三种，继承Thread方法，实现runable接口，实现callable接口

一、继承Thread类创建线程类

（1）定义Thread类的子类，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。

（2）创建Thread子类的实例，即创建了线程对象。

（3）调用线程对象的start()方法来启动该线程。

**[java]** [view plain](http://blog.csdn.net/longshengguoji/article/details/41126119) [copy](http://blog.csdn.net/longshengguoji/article/details/41126119)



1. package com.thread;
3. public class FirstThreadTest extends Thread{
4. int i = 0;
5. //重写run方法，run方法的方法体就是现场执行体
6. public void run()
7. {
8. for(;i<100;i++){
9. System.out.println(getName()+"  "+i);
11. }
12. }
13. public static void main(String[] args)
14. {
15. for(int i = 0;i< 100;i++)
16. {
17. System.out.println(Thread.currentThread().getName()+"  : "+i);
18. if(i==20)
19. {
20. new FirstThreadTest().start();
21. new FirstThreadTest().start();
22. }
23. }
24. }
26. }

上述代码中Thread.currentThread()方法返回当前正在执行的线程对象。GetName()方法返回调用该方法的线程的名字。

二、通过Runnable接口创建线程类

（1）定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。

（2）创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。

（3）调用线程对象的start()方法来启动该线程。

示例代码为：

**[java]** [view plain](http://blog.csdn.net/longshengguoji/article/details/41126119) [copy](http://blog.csdn.net/longshengguoji/article/details/41126119)



1. package com.thread;
3. public class RunnableThreadTest implements Runnable
4. {
6. private int i;
7. public void run()
8. {
9. for(i = 0;i <100;i++)
10. {
11. System.out.println(Thread.currentThread().getName()+" "+i);
12. }
13. }
14. public static void main(String[] args)
15. {
16. for(int i = 0;i < 100;i++)
17. {
18. System.out.println(Thread.currentThread().getName()+" "+i);
19. if(i==20)
20. {
21. RunnableThreadTest rtt = new RunnableThreadTest();
22. new Thread(rtt,"新线程1").start();
23. new Thread(rtt,"新线程2").start();
24. }
25. }
27. }
29. }

三、通过Callable和Future创建线程

（1）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。

（2）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。

（3）使用FutureTask对象作为Thread对象的target创建并启动新线程。

（4）调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

实例代码：

**[java]** [view plain](http://blog.csdn.net/longshengguoji/article/details/41126119) [copy](http://blog.csdn.net/longshengguoji/article/details/41126119)



1. package com.thread;
3. import java.util.concurrent.Callable;
4. import java.util.concurrent.ExecutionException;
5. import java.util.concurrent.FutureTask;
7. public class CallableThreadTest implements Callable<Integer>
8. {
10. public static void main(String[] args)
11. {
12. CallableThreadTest ctt = new CallableThreadTest();
13. FutureTask<Integer> ft = new FutureTask<>(ctt);
14. for(int i = 0;i < 100;i++)
15. {
16. System.out.println(Thread.currentThread().getName()+" 的循环变量i的值"+i);
17. if(i==20)
18. {
19. new Thread(ft,"有返回值的线程").start();
20. }
21. }
22. try
23. {
24. System.out.println("子线程的返回值："+ft.get());
25. } catch (InterruptedException e)
26. {
27. e.printStackTrace();
28. } catch (ExecutionException e)
29. {
30. e.printStackTrace();
31. }
33. }
35. @Override
36. public Integer call() throws Exception
37. {
38. int i = 0;
39. for(;i<100;i++)
40. {
41. System.out.println(Thread.currentThread().getName()+" "+i);
42. }
43. return i;
44. }
46. }

二、创建线程的三种方式的对比

采用实现Runnable、Callable接口的方式创见多线程时，优势是：

线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。

在这种方式下，多个线程可以共享同一个target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。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

使用继承Thread类的方式创建多线程时优势是：

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

劣势是：

线程类已经继承了Thread类，所以不能再继承其他父类。

\*17.runable接口和callable接口实现有什么不同

Callable规定的方法时call()，Runnable规定的方法时run();

Callable的任务执行后可以返回值，而Runnable的任务执行之后不能返回值

Call方法可以抛出异常，而run方法不可以

运行callable的任务可以拿到一个future对象，future表示一步计算的结果。他提供了检查计算是否完成的方法，以等待计算的完成，并获取计算的结果。计算完成之后只能使用get方法来获取结果，如果线程没有执行完，Future.get()方法可能会阻塞当前线程的执行；如果线程出现异常，Future.get()方法会抛出InterruptedException或ExecutionException；如果线程已经取消，会抛出CancellationException。取消由cancel方法来执行。IsDone确定任务是正常的完成还是被取消，一旦计算完成，就不能再取消计算，如果为了可取消性而使用Future但又不提供可用结果，则可以声明Future<?>形式参数，并返回null作为底层任务的结果。Future接口的定义如下：

Future模式：

Future模式在请求发生的时候，会先产生一个Future凭证给发出请求的客户，他的作用就像是Proxy物件，同时，由一个新的执行线程持续进行目标物件的生成（Thread-Per-Message），真正的目标生成之后，将之设定至Future之中，当客户端真正需要目标物件时，目标物件已经准备好，可以让客户提取使用。

结合jdk的future来看，就是当run线程之后，可以把线程的返回值赋给Future并返回一个Future对象，这是你可以立即拿到这个对象，然后进行下面的逻辑。但是如果你要get这个Future中的线程结果，就会被阻塞直到线程结束，就相当于现在的期房，你把手续和钱都交上去，就可以马上拿到合同，但只有合同没有房子，这个时候你已经是有房一族了，你可以先去买家电买装修，但是你要把家电和装修放进去，就必须等待房子完工（阻塞）

18.sleep和wait 的区别：

·sleep来自Thread类，而wait来自Object类

·sleep方法不释放锁，而wait方法释放锁，使得其他线程可以使用同步控制块或者方法

·sleep不释放系统资源，而wait进入线程等待池等待，让出系统资源，其他线程可以占用CPU

·wait没有时间限制，需要等待其他线程调用notify/notifyAll唤醒等待池中的所有线程，才会进入就绪队列等待操作系统分配系统资源。而sleep可以指定时间使他自动唤醒，如果时间不到只能调用interrupt()强行打断。Thread.sleep(0)的作用是“触发操作系统立刻重新进行一次CPU竞争”

·wait、notify、notifyAll只能在同步控制方法或同步控制块里使用，而sleep可以在任何情况下进行使用。

19.生产者消费者问题代码：

**package** com.Oracle.xcg;

**import** java.util.Random;

**public** **class** Store

{

**public** **static** **void** main(String[] args)

{

//创建一个面包店

Stores store=**new** Stores();

Maker m=**new** Maker(store);

Seller s=**new** Seller(store);

m.start();

s.start();

}

}

**class** Stores {

**int** num=0;

Random r=**new** Random();

**public** **synchronized** **void** make() **throws** Exception{

**this**.notifyAll();

**if**(num==0){

num=r.nextInt(20)+1;

System.*out*.println("生产个数："+num);

**this**.wait();

//唤醒使用同一个锁的其他线程

**this**.notifyAll();

}

}

**public** **synchronized** **void** sale() **throws** Exception{

**if**(num>0){

**int** i=r.nextInt(num)+1;

System.*out*.println("消费个数："+i);

num-=i;

}

**if**(num==0){

**this**.notifyAll();

**this**.wait();

}

}

}

**class** Maker **extends** Thread{

Stores store=**new** Stores();

**public** Maker(Stores store)

{

**super**();

**this**.store = store;

}

**public** **void** run(){

**while**(**true**){

**try**

{

store.make();

Thread.*sleep*(100);

} **catch** (Exception e)

{

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**class** Seller **extends** Thread{

Stores store=**new** Stores();

**public** Seller(Stores store)

{

**super**();

**this**.store = store;

}

**public** **void** run(){

**while**(**true**){

**try**

{

store.sale();

Thread.*sleep*(100);

} **catch** (Exception e)

{

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

20.科学家进餐问题手写代码

**package** com.Oracle.xcg;

**public** **class** ThinkAndEat

{

**public** **static** **void** main(String[] args)

{

kuai k1 = **new** kuai("筷子1");

kuai k2 = **new** kuai("筷子2");

kuai k3 = **new** kuai("筷子3");

kuai k4 = **new** kuai("筷子4");

kuai k5 = **new** kuai("筷子5");

Philosopher p1 = **new** Philosopher("1号哲学家", k1, k2);

Philosopher p2 = **new** Philosopher("2号哲学家", k2, k3);

Philosopher p3 = **new** Philosopher("3号哲学家", k3, k4);

Philosopher p4 = **new** Philosopher("4号哲学家", k4, k5);

Philosopher p5 = **new** Philosopher("5号哲学家", k5, k1);

p1.start();

p2.start();

p3.start();

p4.start();

p5.start();

}

}

**class** kuai{

String name;

**boolean** Enable=**true**;

/\*\*

\* **@param** name

\*/

**public** kuai(String name)

{

**super**();

**this**.name = name;

}

**public** **synchronized** **void** pickup(){

**this**.Enable=**false**;

}

**public** **synchronized** **void** putdown(){

**this**.Enable=**true**;

**this**.notifyAll();

}

}

**class** Philosopher **extends** Thread{

String name;

kuai left;

kuai right;

/\*\*

\* **@param** name

\* **@param** left

\* **@param** right

\*/

**public** Philosopher(String name, kuai left, kuai right)

{

**super**();

**this**.name = name;

**this**.left = left;

**this**.right = right;

}

**public** **void** run(){

**if**(left.Enable){

left.pickup();

}

**else**{

**while**(!left.Enable){

**try**{

System.*out*.println(name+"在等待"+left.name);

**if**(!right.Enable){

**this**.right.putdown();

System.*out*.println(name+"没有等到筷子，放下了"+right.name);

}

Thread.*sleep*(500);

}**catch**(Exception e){System.*out*.println(e);}

}

}

System.*out*.println(name+"拿起了"+left.name);

**try**{Thread.*sleep*(500);}**catch**(Exception e){}

**if**(right.Enable){

right.pickup();

}

**else**{

**while**(!left.Enable){

**try**{

System.*out*.println(name+"在等待"+right.name);

**if**(!left.Enable){

**this**.left.putdown();

System.*out*.println(name+"没有等到筷子，放下了"+left.name);

}

Thread.*sleep*(500);

}**catch**(Exception e){}

}

}

System.*out*.println(name+"拿起了"+right.name);

System.*out*.println(name+"左右开工，开始进餐");

**try**{Thread.*sleep*(500);}**catch**(Exception e){}

System.*out*.println(name+"吃完饭，放下了"+left.name+"和"+right.name);

left.putdown();

right.putdown();

}

}

21.TCP，ARP，UDP区别：

·TCP：Transmission Control Protocol传输控制协议（面向连接的协议）

TCP的目的是提供可靠的数据传输，并在进行通信的设备之间保持一个虚拟的连接，Tcp在数据包的接受无序、丢失、或者在交付期间被破坏时，负责数据恢复。他通过为其发送的每个数据包提供一个序号来完成此恢复。较低的网络层会将每个数据包当作一个独立的单位，因此数据包可以沿着完全不同的路径进行发送，即使他们都是同一消息的组成部分，这种路由与网络层处理分段和重新组装的方式十分相似，只是级别更高而已。

为确保在正确的接受数据，TCP要求在目标计算机成功收到信息的时候发回一个确认，即（ACK），如果在某个阶段没有接收到相应的Ack，也重新发送数据包，并在必要的时候丢弃他。

TCP的三次握手：1.客户端发送syn报文，并置发送序号为X 。2.服务端发送syn和Ack报文，并置发送序号为Y，在确认序号为X+1 。客户端发送ACK报文，并置发送序号为Z，在确认序号为Y+1 。

若Server端需要与client端断开连接，则收到client端发送的syn包后，即返回一个RST包，以拒绝client发来的建议连接的请求。若client端发送syn包后，没有收到server发来的回应，则会重新发送syn包，多次建立连接都没有相应，则此次建立连接失败。

\*TCP优点：1.面向连接的传输2.数据传输稳定可靠3.端对端的通信4.全双工方式5.以字节为单位。

\*TCP缺点：传输速度慢

·UDP：User Data Protocol用户数据包协议（面向非连接的协议）

UDP与TCP的主要区别就是UDP不一定能提供可靠的传输，事实上，该协议不能保证数据准确无误的到达目的地。UDP在很多方面非常有效，当某个程序的目标是尽快的传输更多的数据的时候。很多程序将单独使用TCP连接和单独的UDP连接，重要的状态信息用可靠的TCP传输，主数据流通过UDP协议发送消息

TCP与UDP的选择：

UDP包不具备TCP的可靠性和控制机制，UDP传输时间短，TCP性能差

\*UDP优点：1.传输速度快2.效率高

\*UDP缺点：数据包可能会丢失

·ARP：Address Resolution Protocol 地址解析协议

在局域网中，网络中实际传输的是帧，帧里面有目标主机的MAC地址，在以太网中，一个主机要与另一个主机进行直接通讯，必须要知道目标主机的MAC地址，而目标地址MAC就是通过地址解析协议获得的，就是主机在发送帧之前将目标IP地址转换成目标MAC地址的过程，ARP地址解析协议的基本功能就是通过目标设备的IP地址，查询目标设备的MAC地址，以保证通讯的正常进行

22.osi切锯模型：

·物理层：物理层所处理的单位是比特，物理层向上为数据链路层提供物理链路，实现透明的比特流传输服务，物理层向下与物理媒介相连，要确定连接物理媒体的网络接口的机械，电气，功能和过程方面的特性

·数据链路层：数据链路层负责在单个链路上的节点间传送以帧为PDU的数据，在不太可靠的物理链路上实现可靠的数据传输，数据链路层的主要功能包括：建立、维持和释放数据链路的连接，链路的访问控制、流量监控和差错控制。

·网络层：网络层传输的PDU称为分组或包，在物理网络间传输分组，负责将原主机的报文通过中间节点传送到目的端。网络层是通信子网的最高层，为主机提供虚拟电路和数据包两种方式的服务，网络层主要负责是分组转发和路由选择，根据路由表把分组逐渐由原站传送到目的站，并能适应网络中的负载及拓扑结构的变化，动态的更新路由表

·传输层：传输层传输的PDU称为报文，传输层为源节点和目的节点的用户进程之间提供端到端的可靠传输服务，端到端的传输指的是源节点和目的节点的两个传输层实体之间，不涉及路由器等中间节点。为了保证可靠的传输服务，传输层具备以下功能：面向连接、流量控制、拥塞控制、差错控制相关网络质量选择等。

·会话层：会话层在传输层的基础上增加控制会话的机制，建立、组织和协调应用进程之间的交互过程，会话层提供的会话服务种类包括双工、半双工和单工方式。会话管理的另一种方式是令牌管理，只有令牌的持有者才能执行某种操作，会话层提供会话的同步控制，当出现故障的时候，会话活动在故障点之前的同步点进行重复，而不必从头开始

·表示层：表示层定义用户或应用程序之间交换数据的格式，提供数据表示之间的转换服务，保证传输的信息到达目的端后意义不变

·应用层：应用层直接面向用户，为用户提供各种网路资源的方便访问服务

23.sychronized：

Synchronized是java中的关键字，当他来修饰一个方法或一个代码块的时候，能够保证在同一时刻做多只有一个线程执行该代码块

* 当两个并发线程访问同一个对象中的Object中的synchronized(this)同步块代码的时候，一个时间片内只有一个线程能得到执行，另一个线程必须等待当前线程执行完代码块之后才能执行。
* 然而，当一个线程访问一个Object的synchronized同步块代码的时候，其他线程仍然可以访问该Object的其他方法
* 尤其关键的是，当一个线程访问Object中的一个synchronized同步代码块的时候，其他线程对该对象中所有其他synchronized同步块的访问将被阻塞
* 当一个线程访问一个对象的synchronized的同步代码块的时候，他就获得了这个对象的对象锁，结果其他线程对该Object的所有同步块代码部分的访问暂时被阻塞
* Synchronized关键字包括两种方法，synchronized方法和synchronized块
* Synchronized方法通过方法声明中加入synchronized关键字来声明，synchronized方法控制对类成员变量的访问，每个类实例对应一把锁，每个synchronized方法都必须获得调用该方法的类实例的锁方法执行，否则所属线程阻塞，方法一旦执行，就独自占用锁，知道从该方法返回时才会释放锁，此后被阻塞的线程才会获得锁，重新进入可执行状态，这种机制确保了同一时刻对于每一个类实例，其声明的所有synchronized方法至多只有一个处于可执行状态，因此，至多只有一个线程能获得该类实例对应的锁，从而有效的避免了成员变量的访问冲突，只要有可能访问类的成员变量的方法都被声明synchronized。在java中，不仅是类实例，每一个类也对应一把锁，这样我们也可以将类的静态成员变量声明为synchronized，以控制其对类的静态成员变量的访问，
* Synchronized方法的缺陷：若将一个大的方法声明为synchronized将会大大的影响效率，典型的，若将线程类的方法run()声明为synchronized，由于线程在整个生命周期内他一直在运行，因此将导致他对本类任何synchronized方法的调用都永远不会成功。当然我们可以通过访问类成员变量的代码放到专门的方法中，将其生命为synchronized，并在主方法中调用该方法来解决问题，但是java为我们提供了跟好的解决方法，那就是synchronized块！
* Synchronized块：通过synchronized关键字来声明synchronized块语法：synchronized（Object）{} 这样一个代码块，其中的代码必须对象syncObject，的锁方法执行
* 说明：无论synchronized关键字加载方法上还是对象上，他取得的锁都是对象，而不是一段代码或者函数当作锁，而且同步方法很可能还会被其他线程的对象访问
* 每一个对象只有一个锁与之相关联
* 实现同步是需要很大的系统开销的，甚至很可能造成死锁，所以尽量少用
* Synchronized的用法：
* 1.public synchronized void method(){}同步方法，这是synchronized锁定的是调用这个同步方法的对象，也就是所，当一个对象p1在不同的线程中执行这个方法的时候，将会形成互斥，如果该对象中有多个同步方法，当一个线程访问获得执行对象中的synchronized方法后，则该对象中的所有synchronized方法不允许其他线程执行，但这个对象所属的Class所产生的另一个对象p2却能够任意的调用这个被加了synchronized关键字的方法。
* 2.public void method(SomeObject so){ synchronized(so){} } 这个锁就是so对象锁，每个对象对应唯一一把锁，所以那个线程拿到这个对象锁就能运行他所控制的那段代码，当有一个明确的对象作为锁时，就能够写这个代码，但当没有明确的对象作为锁的，只想让一段代码同步时，能够创建一个特别的instance变量（对象），让他来充当锁。例如：private byte[] lock = new byte[0]; public void method(){ synchronized(lock){ } } 说明：零长度的byte数组对象创建起来比任何对象都要经济—根据源代码，生成零长度的byte[]对象只需要3条操作码，而Object lock = new Object();则需要7行操作码。
* 3.将synchronized作用与static函数：class foo{ public synchronized static void method1(){ } public void method2(){ synchronized(Foo.class); }} 这两个同步方法都调用这个方法的对象所属的类的类锁(Class)，而不是由这个Class产生的某个具体对象
* 加入一个类中定义了一个synchronized的static函数A，也定义了一个synchronized的instance函数B，那么这个类的同一个对象Obj在多线程中分别访问A和B两个方法的时候，不会构成同步，因为他们的锁都不相同，A方法的锁时Obj所属的类的Class，而B的锁时Obj所属的这个对象

24.什么叫原子操作：volatile atomic operation

* 原子操作仅执行一次，在执行的过程中不会中断，也不会休眠，是最小的执行单元；
* 鉴于原子操作这些特性，可以利用他来解决竞态问题。往后的其他同步机制都是在原子操作的基础上进行修改的
* 原子操作有整形原子操作，64位原子操作以及位原子操作。
* 整形原子操作：要使用原子操作，需要定义一个原子变量，然后使用内核提供的接口对其进行原子操作
* 独占锁是一种悲观锁，synchronized就是一种悲观锁，会导致其他所有需要的线程挂起，等待持有锁的线程释放锁，而另一个更有效的锁就是乐观锁，所谓乐观锁就是没次不加锁而是假设没有冲突而去完成某项任务，如果因为冲突失效就重试，知道成功。
* 原子操作是不需要synchronized，所谓原子操作就是不会被多线程调度机制打断的操作，这种操作一旦开始，就会一直运行到结束，中间不会切换到另一个线程
* 通常所说的原子操作包括对long、double型的primitive进行赋值，以及返回这两者之外的primitive，之所以把他们排出在外是因为他们都比较大，而JVM的设计规范有没有要求读操作和和赋值操作必须是原子操作，不过如果你在long或double前面加了volatile，那么他肯定是原子操作。最安全的原子操作只有读取和赋值primitive，根据EvenGenerator.java所提示的，原子操作也能访问正处于无效状态的对象，long和double类型的操作不一定是原子操作
* 最安全的做法如下：如果要使用一个类的synchronized的方法，不如索性把所有的方法都设置成synchronized，因为判断哪个方法需要同步是很难的
* 删除synchronized使时要小心，通常是为了性能，但在1.3和1,4之后同步性能已经有提升

|  |  |
| --- | --- |
|  | * 很多人以为，这个是多线程并发问题，只需要在变量count之前加上volatile就可以避免这个问题，那我们在修改代码看看，看看结果是不是符合我们的期望 |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class Counter {        public volatile static int count = 0;        public static void inc() {            //这里延迟1毫秒，使得结果明显          try {              Thread.sleep(1);          } catch (InterruptedException e) {          }            count++;      }        public static void main(String[] args) {            //同时启动1000个线程，去进行i++计算，看看实际结果            for (int i = 0; i < 1000; i++) {              new Thread(new Runnable() {                  @Override                  public void run() {                      Counter.inc();                  }              }).start();          }            //这里每次运行的值都有可能不同,可能为1000          System.out.println("运行结果:Counter.count=" + Counter.count);      }  } |

运行结果:Counter.count=992

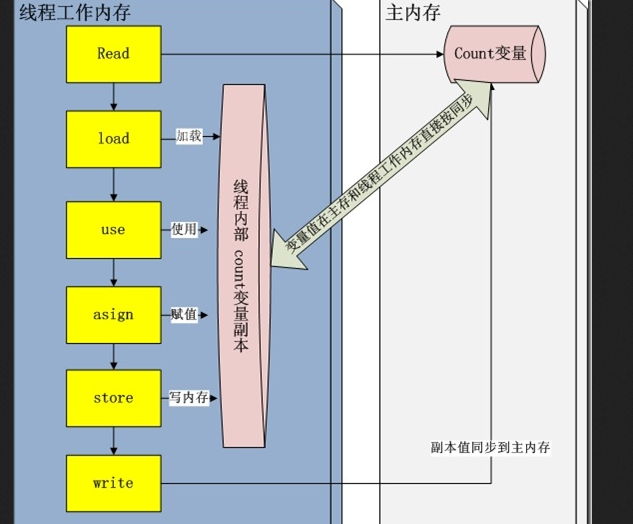
运行结果还是没有我们期望的1000，下面我们分析一下原因

在 java 垃圾回收整理一文中，描述了jvm运行时刻内存的分配。其中有一个内存区域是jvm虚拟机栈，每一个线程运行时都有一个线程栈，

线程栈保存了线程运行时候变量值信息。当线程访问某一个对象时候值的时候，首先通过对象的引用找到对应在堆内存的变量的值，然后把堆内存

变量的具体值load到线程本地内存中，建立一个变量副本，之后线程就不再和对象在堆内存变量值有任何关系，而是直接修改副本变量的值，

在修改完之后的某一个时刻（线程退出之前），自动把线程变量副本的值回写到对象在堆中变量。这样在堆中的对象的值就产生变化了。下面一幅图

描述这写交互

read and load 从主存复制变量到当前工作内存  
use and assign  执行代码，改变共享变量值   
store and write 用工作内存数据刷新主存相关内容

其中use and assign 可以多次出现

但是这一些操作并不是原子性，也就是 在read load之后，如果主内存count变量发生修改之后，线程工作内存中的值由于已经加载，不会产生对应的变化，所以计算出来的结果会和预期不一样

对于volatile修饰的变量，jvm虚拟机只是保证从主内存加载到线程工作内存的值是最新的

例如假如线程1，线程2 在进行read,load 操作中，发现主内存中count的值都是5，那么都会加载这个最新的值

在线程1堆count进行修改之后，会write到主内存中，主内存中的count变量就会变为6

线程2由于已经进行read,load操作，在进行运算之后，也会更新主内存count的变量值为6

导致两个线程及时用volatile关键字修改之后，还是会存在并发的情况。

\***Java 理论与实践: 正确使用 Volatile 变量**

Java™ 语言包含两种内在的同步机制：同步块（或方法）和 volatile 变量。这两种机制的提出都是为了实现代码线程的安全性。其中 Volatile 变量的同步性较差（但有时它更简单并且开销更低），而且其使用也更容易出错。在这期的 Java 理论与实践 中，Brian Goetz 将介绍几种正确使用 volatile 变量的模式，并针对其适用性限制提出一些建议。

Java 语言中的 volatile 变量可以被看作是一种 “程度较轻的 synchronized”；与 synchronized 块相比，volatile 变量所需的编码较少，并且运行时开销也较少，但是它所能实现的功能也仅是 synchronized 的一部分。本文介绍了几种有效使用 volatile 变量的模式，并强调了几种不适合使用 volatile 变量的情形。

锁提供了两种主要特性：互斥（mutual exclusion） 和可见性（visibility）。互斥即一次 只允许一个线程持有某个特定的锁，因此可使用该特性实现对共享数据的协调访问协议，这样，一次就只有一个线程能够使用该共享数据。可见性要更加复杂一些， 它必须确保释放锁之前对共享数据做出的更改对于随后获得该锁的另一个线程是可见的 —— 如果没有同步机制提供的这种可见性保证，线程看到的共享变量可能是修改前的值或不一致的值，这将引发许多严重问题。

## Volatile 变量

Volatile 变量具有 synchronized 的可见性特性，但是不具备原子特性。这就是说线程能够自动发现 volatile 变量的最新值。Volatile 变量可用于提供线程安全，但是只能应用于非常有限的一组用例：多个变量之间或者某个变量的当前值与修改后值之间没有约束。因此，单独使用 volatile 还不足以实现计数器、互斥锁或任何具有与多个变量相关的不变式（Invariants）的类（例如 “start <=end”）。

出于简易性或可伸缩性的考虑，您可能倾向于使用 volatile 变量而不是锁。当使用 volatile 变量而非锁时，某些习惯用法（idiom）更加易于编码和阅读。此外，volatile 变量不会像锁那样造成线程阻塞，因此也很少造成可伸缩性问题。在某些情况下，如果读操作远远大于写操作，volatile 变量还可以提供优于锁的性能优势。

### 正确使用 volatile 变量的条件

您只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

* 对变量的写操作不依赖于当前值。
* 该变量没有包含在具有其他变量的不变式中。

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

第一个条件的限制使 volatile 变量不能用作线程安全计数器。虽然增量操作（x++）看上去类似一个单独操作，实际上它是一个由读取－修改－写入操作序列组成的组合操作，必须以原子方式执行，而 volatile 不能提供必须的原子特性。实现正确的操作需要使 x 的值在操作期间保持不变，而 volatile 变量无法实现这点。（然而，如果将值调整为只从单个线程写入，那么可以忽略第一个条件。）

大多数编程情形都会与这两个条件的其中之一冲突，使得 volatile 变量不能像 synchronized 那样普遍适用于实现线程安全。清单 1 显示了一个非线程安全的数值范围类。它包含了一个不变式 —— 下界总是小于或等于上界。

##### 清单 1. 非线程安全的数值范围类

@NotThreadSafe

public class NumberRange {

private int lower, upper;

public int getLower() { return lower; }

public int getUpper() { return upper; }

public void setLower(int value) {

if (value > upper)

throw new IllegalArgumentException(...);

lower = value;

}

public void setUpper(int value) {

if (value < lower)

throw new IllegalArgumentException(...);

upper = value;

}

}

这种方式限制了范围的状态变量，因此将 lower 和 upper 字段定义为 volatile 类型不能够充分实现类的线程安全；从而仍然需要使用同步。否则，如果凑巧两个线程在同一时间使用不一致的值执行 setLower 和 setUpper 的话，则会使范围处于不一致的状态。例如，如果初始状态是 (0, 5)，同一时间内，线程 A 调用 setLower(4) 并且线程 B 调用 setUpper(3)，显然这两个操作交叉存入的值是不符合条件的，那么两个线程都会通过用于保护不变式的检查，使得最后的范围值是 (4, 3) —— 一个无效值。至于针对范围的其他操作，我们需要使 setLower() 和 setUpper() 操作原子化 —— 而将字段定义为 volatile 类型是无法实现这一目的的。

### 性能考虑

使用 volatile 变量的主要原因是其简易性：在某些情形下，使用 volatile 变量要比使用相应的锁简单得多。使用 volatile 变量次要原因是其性能：某些情况下，volatile 变量同步机制的性能要优于锁。

很难做出准确、全面的评价，例如 “X 总是比 Y 快”，尤其是对 JVM 内在的操作而言。（例如，某些情况下 VM 也许能够完全删除锁机制，这使得我们难以抽象地比较 volatile 和 synchronized 的开销。）就是说，在目前大多数的处理器架构上，volatile 读操作开销非常低 —— 几乎和非 volatile 读操作一样。而 volatile 写操作的开销要比非 volatile 写操作多很多，因为要保证可见性需要实现内存界定（Memory Fence），即便如此，volatile 的总开销仍然要比锁获取低。

volatile 操作不会像锁一样造成阻塞，因此，在能够安全使用 volatile 的情况下，volatile 可以提供一些优于锁的可伸缩特性。如果读操作的次数要远远超过写操作，与锁相比，volatile 变量通常能够减少同步的性能开销。

## 正确使用 volatile 的模式

很多并发性专家事实上往往引导用户远离 volatile 变量，因为使用它们要比使用锁更加容易出错。然而，如果谨慎地遵循一些良好定义的模式，就能够在很多场合内安全地使用 volatile 变量。要始终牢记使用 volatile 的限制 —— 只有在状态真正独立于程序内其他内容时才能使用 volatile —— 这条规则能够避免将这些模式扩展到不安全的用例。

### 模式 #1：状态标志

也许实现 volatile 变量的规范使用仅仅是使用一个布尔状态标志，用于指示发生了一个重要的一次性事件，例如完成初始化或请求停机。

很多应用程序包含了一种控制结构，形式为 “在还没有准备好停止程序时再执行一些工作”，如清单 2 所示：

##### 清单 2. 将 volatile 变量作为状态标志使用

volatile boolean shutdownRequested;

...

public void shutdown() { shutdownRequested = true; }

public void doWork() {

while (!shutdownRequested) {

// do stuff

}

}

很可能会从循环外部调用 shutdown() 方法 —— 即在另一个线程中 —— 因此，需要执行某种同步来确保正确实现 shutdownRequested 变量的可见性。（可能会从 JMX 侦听程序、GUI 事件线程中的操作侦听程序、通过 RMI 、通过一个 Web 服务等调用）。然而，使用 synchronized 块编写循环要比使用清单 2 所示的 volatile 状态标志编写麻烦很多。由于 volatile 简化了编码，并且状态标志并不依赖于程序内任何其他状态，因此此处非常适合使用 volatile。

这种类型的状态标记的一个公共特性是：通常只有一种状态转换；shutdownRequested 标志从 false 转换为 true，然后程序停止。这种模式可以扩展到来回转换的状态标志，但是只有在转换周期不被察觉的情况下才能扩展（从 false 到 true，再转换到 false）。此外，还需要某些原子状态转换机制，例如原子变量。

### 模式 #2：一次性安全发布（one-time safe publication）

缺乏同步会导致无法实现可见性，这使得确定何时写入对象引用而不是原语值变得更加困难。在缺乏同步的情况下，可能会遇到某个对象引用的更新值（由另一个线 程写入）和该对象状态的旧值同时存在。（这就是造成著名的双重检查锁定（double-checked-locking）问题的根源，其中对象引用在没有 同步的情况下进行读操作，产生的问题是您可能会看到一个更新的引用，但是仍然会通过该引用看到不完全构造的对象）。

实现安全发布对象的一种技术就是将对象引用定义为 volatile 类型。清单 3 展示了一个示例，其中后台线程在启动阶段从数据库加载一些数据。其他代码在能够利用这些数据时，在使用之前将检查这些数据是否曾经发布过。

##### 清单 3. 将 volatile 变量用于一次性安全发布

public class BackgroundFloobleLoader {

public volatile Flooble theFlooble;

public void initInBackground() {

// do lots of stuff

theFlooble = new Flooble(); // this is the only write to theFlooble

}

}

public class SomeOtherClass {

public void doWork() {

while (true) {

// do some stuff...

// use the Flooble, but only if it is ready

if (floobleLoader.theFlooble != null)

doSomething(floobleLoader.theFlooble);

}

}

}

如果 theFlooble 引用不是 volatile 类型，doWork() 中的代码在解除对 theFlooble 的引用时，将会得到一个不完全构造的 Flooble。

该模式的一个必要条件是：被发布的对象必须是线程安全的，或者是有效的不可变对象（有效不可变意味着对象的状态在发布之后永远不会被修改）。volatile 类型的引用可以确保对象的发布形式的可见性，但是如果对象的状态在发布后将发生更改，那么就需要额外的同步。

### 模式 #3：独立观察（independent observation）

安全使用 volatile 的另一种简单模式是：定期 “发布” 观察结果供程序内部使用。例如，假设有一种环境传感器能够感觉环境温度。一个后台线程可能会每隔几秒读取一次该传感器，并更新包含当前文档的 volatile 变量。然后，其他线程可以读取这个变量，从而随时能够看到最新的温度值。

使用该模式的另一种应用程序就是收集程序的统计信息。清单 4 展示了身份验证机制如何记忆最近一次登录的用户的名字。将反复使用 lastUser 引用来发布值，以供程序的其他部分使用。

##### 清单 4. 将 volatile 变量用于多个独立观察结果的发布

public class UserManager {

public volatile String lastUser;

public boolean authenticate(String user, String password) {

boolean valid = passwordIsValid(user, password);

if (valid) {

User u = new User();

activeUsers.add(u);

lastUser = user;

}

return valid;

}

}

该模式是前面模式的扩展；将某个值发布以在程序内的其他地方使用，但是与一次性事件的发布不同，这是一系列独立事件。这个模式要求被发布的值是有效不可变的 —— 即值的状态在发布后不会更改。使用该值的代码需要清楚该值可能随时发生变化。

### 模式 #4：“volatile bean” 模式

volatile bean 模式适用于将 JavaBeans 作为“荣誉结构”使用的框架。在 volatile bean 模式中，JavaBean 被用作一组具有 getter 和/或 setter 方法 的独立属性的容器。volatile bean 模式的基本原理是：很多框架为易变数据的持有者（例如 HttpSession）提供了容器，但是放入这些容器中的对象必须是线程安全的。

在 volatile bean 模式中，JavaBean 的所有数据成员都是 volatile 类型的，并且 getter 和 setter 方法必须非常普通 —— 除了获取或设置相应的属性外，不能包含任何逻辑。此外，对于对象引用的数据成员，引用的对象必须是有效不可变的。（这将禁止具有数组值的属性，因为当数组 引用被声明为 volatile 时，只有引用而不是数组本身具有 volatile 语义）。对于任何 volatile 变量，不变式或约束都不能包含 JavaBean 属性。清单 5 中的示例展示了遵守 volatile bean 模式的 JavaBean：

##### 清单 5. 遵守 volatile bean 模式的 Person 对象

@ThreadSafe

public class Person {

private volatile String firstName;

private volatile String lastName;

private volatile int age;

public String getFirstName() { return firstName; }

public String getLastName() { return lastName; }

public int getAge() { return age; }

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public void setAge(int age) {

this.age = age;

}

}

## volatile 的高级模式

前面几节介绍的模式涵盖了大部分的基本用例，在这些模式中使用 volatile 非常有用并且简单。这一节将介绍一种更加高级的模式，在该模式中，volatile 将提供性能或可伸缩性优势。

volatile 应用的的高级模式非常脆弱。因此，必须对假设的条件仔细证明，并且这些模式被严格地封装了起来，因为即使非常小的更改也会损坏您的代码！同样，使用更高级 的 volatile 用例的原因是它能够提升性能，确保在开始应用高级模式之前，真正确定需要实现这种性能获益。需要对这些模式进行权衡，放弃可读性或可维护性来换取可能的性 能收益 —— 如果您不需要提升性能（或者不能够通过一个严格的测试程序证明您需要它），那么这很可能是一次糟糕的交易，因为您很可能会得不偿失，换来的东西要比放弃的 东西价值更低。

### 模式 #5：开销较低的读－写锁策略

目前为止，您应该了解了 volatile 的功能还不足以实现计数器。因为 ++x 实际上是三种操作（读、添加、存储）的简单组合，如果多个线程凑巧试图同时对 volatile 计数器执行增量操作，那么它的更新值有可能会丢失。

然而，如果读操作远远超过写操作，您可以结合使用内部锁和 volatile 变量来减少公共代码路径的开销。清单 6 中显示的线程安全的计数器使用 synchronized 确保增量操作是原子的，并使用 volatile 保证当前结果的可见性。如果更新不频繁的话，该方法可实现更好的性能，因为读路径的开销仅仅涉及 volatile 读操作，这通常要优于一个无竞争的锁获取的开销。

##### 清单 6. 结合使用 volatile 和 synchronized 实现 “开销较低的读－写锁”

@ThreadSafe

public class CheesyCounter {

// Employs the cheap read-write lock trick

// All mutative operations MUST be done with the 'this' lock held

@GuardedBy("this") private **volatile** int value;

public int getValue() { return value; }

public **synchronized** int increment() {

return value++;

}

}

之所以将这种技术称之为 “开销较低的读－写锁” 是因为您使用了不同的同步机制进行读写操作。因为本例中的写操作违反了使用 volatile 的第一个条件，因此不能使用 volatile 安全地实现计数器 —— 您必须使用锁。然而，您可以在读操作中使用 volatile 确保当前值的可见性， 因此可以使用锁进行所有变化的操作，使用 volatile 进行只读操作。其中，锁一次只允许一个线程访问值，volatile 允许多个线程执行读操作，因此当使用 volatile 保证读代码路径时，要比使用锁执行全部代码路径获得更高的共享度 —— 就像读－写操作一样。然而，要随时牢记这种模式的弱点：如果超越了该模式的最基本应用，结合这两个竞争的同步机制将变得非常困难。

## 结束语

与锁相比，Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 volatile 的使用条件 —— 即变量真正独立于其他变量和自己以前的值 —— 在某些情况下可以使用 volatile 代替 synchronized 来简化代码。然而，使用 volatile 的代码往往比使用锁的代码更加容易出错。本文介绍的模式涵盖了可以使用 volatile 代替 synchronized 的最常见的一些用例。遵循这些模式（注意使用时不要超过各自的限制）可以帮助您安全地实现大多数用例，使用 volatile 变量获得更佳性能。

25.什么是竞争条件，如何发现和解决

·如果两个线程竞争使用相同的资源和数据，就称之为竞争条件，竞争条件不仅仅会发生在两个线程同时更改相同数据的场景中，还可能发生在一个线程正在修改某一数据而其他线程正在读取这个数据的时候。竞争条件主要由两大原因造成的：时下流行的just-in-time（JIT）编译器的优化以及java内存模型。

* 发现和解决：按照性能从坏到好：阻塞式加锁，非阻塞式加锁，无锁。**阻塞式加锁**：即采用synchronized和ReentrantLock，他们都是重量级操作，使用ReentrantLock常常是为了其他的特性，不如公平锁，条件，可中断等特性。**非阻塞是加锁**：非阻塞式加锁就是所说的乐观锁，他不会尝试先锁后操作，而是先进行操作，如果操作没有冲突就代表操作成功，如果有冲突，再采取其他的补偿措施，如重试，非阻塞式加锁是靠硬件指令集的发展才带动起来的，常见的非阻塞式加锁的指令有：测试并设置，获取并增加，交换，比较并交换。**无锁策略**：分三类，1.不可变类，说明该类是类似String类的对象，所谓不可变类，即将该类声明为final，表明此状态不能被修改。2.线程封闭，如果一个方法只依赖与传进来的草书而不依赖与对象状态，则这个方法是可重入的，简单的将就是这个方法不要调用对象中的实例变量，通过这种方式可以减少很多不必要的同步，例如：Servlet中级你了那个不要定义和调用实例变量，因为每一个请求都会被一个进程所调用，然后去调用同一个Servlet的同一个方法，如果该方法调用了实例变量，势必造成错误。3.ThreadLocal：ThreadLocal相当于每一个线程都有一个变量的保险箱，最好的理解就是把他想象成一个以线程为key，你要保管的实例为value和Hashmap。在ThreadLocal源码中，你会发现ThreadLocal将this传入作为key来查找变量。
* 为什么会引起竞争条件：因为操作缺少原子性

26.start和run的区别：

·start方法用来启动线程，真正的实现了多线程的运行，这是无需等待run方法体代码执行完毕而继续执行，下面的代码，通过Thread类的start方法来启动一个线程，这是的线程处于就绪状态，一旦获得时间片，就开始运行，这里所说的run方法称为线程体，他包含了要执行线程的内容，Run 方法运行结束，此线程运行终止。

·run 方法只是一个类的普通方法，如果直接调用Run方法程序中依旧只有主线程这一个线程，其程序的执行路径依旧是只有一条，还是顺序执行，然等待Run方法结束了之后才能执行以后的代码，这样就没有达到线程的目的。

27.乐观锁和悲观锁：

·悲观锁：它是指对数据被外界保持保守状态，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往是依靠数据库提供的锁机制（也只有数据库层的锁机制才能真正的保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）

·数据库的锁机制：未提交锁，提交锁，重复读，序列化

·锁机制：共享锁：其他事务可以读，但不能修改。排它锁：其他事务不能读取

·锁粒度：一般分为：行锁，表锁，库锁

·未提交读：一个更新数据库的事务A在未提交之前，另一个事务B正在读取事务A更新的记录，会产生脏读现象，这是因为A事务在开启DBTransaction后，做一些DML操作时，记录会保存在内存中，这是B事务读取了A事务提交在内存中的数据，产生了脏读。

·提交读：数据的修改只有在提交之后才会被读取

·重复读：当数据库隔离级别设置成repeatable read后，事务A中的select的过程中事务B可以修改A读取的部分数据，当A第二次执行同样的sql时候，返回和上次相同的数据，消除不可重复读

·序列化：当数据库隔离机制设置成serializeable后，事务A中的select会以共享锁锁定相关的数据，（在select返回数据结果集），这些数据不可以被修改，若事务B对这些数据进行修改操作，会处于等待状态，消除幻读。

·悲观锁：相对于悲观锁而言，乐观锁采用了更加宽松的加锁机制，悲观锁大多数是依靠数据库的锁实现，以保证操作操作最大程度的独占性，而随之而来的就是数据库性能的大量开销，特别是对于长事务而言，这样的开销通常是无法承受。乐观锁大多数是基于数据版本记录机制实现，何为数据版本，即为数据增加版本标识，在基于数据表的版本解决方案中，一般是为一个数据表增加一个version字段来实现，读取数据时，将此版本号一同读出，之后更新时，为此版本号加一，此时，将提交数据的数据版本与数据库表中的版本信息进行对比，如果提交的数据版本号大于数据库当前版本号，则予以更新，否则认为是过期数据。

28死锁，活锁，饥饿 之间的区别

·死锁，是指两个或两个以上的线程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，他们都将无法推进下去，此时系统处于死锁状态或系统产生了死锁，这些永远在互相等待的线程称为死锁线程。

·死锁发生的条件：1互斥条件，线程对资源的访问时排他性的，如果一个线程占用了某个资源，那么其他线程继续等待，知道资源被释放。2请求和保持条件，线程T1至少已经保持了一个资源R1的占用，但又提出对另一个资源的R2请求，而此时，资源R2被其他线程R2锁占用，于是线程R1也必须等待，但对自己保持的资源又不释放。3不剥夺条件，线程已获得资源，在未使用完之前，不能被其他线程剥夺，只能在使用完之后自己释放。4环路等待条件，在死锁发生时，必然存在一个进程-资源环形链，即，{p0,p1,p2…pn}进程P0等待p1占用的资源，p1等待p2资源，pn等待p0占用的资源。

·活锁，是指线程1可以使用资源，但是他很礼貌，让其他线程先使用资源，线程2也可以使用资源，但他很绅士，让其他线程先使用资源，这样互相让最后谁也没有使用资源。

·饥饿，是指线程T1占用了资源R，线程T2又请求资源R，于是T2等待，T3也请求资源R，当T1释放了R上的封锁后，系统首先批准了T3的请求，T2仍然等待，然后T4又请求了R，当T3释放了R，系统又批给了T4，T2有可能永远等待下去。

·区别：死锁发生在当一进程请求其他的进程占有的资源而被阻塞使。另一方面，活锁不会被阻塞，而是不同检测一个不可能为真的条件。出去进程本身持有的资源外，活锁状态的进程会持续消耗宝贵的CPU时间。最后，进程会处于饥饿转态是因为持续的有其他优先级更高的进程请求相同的资源。不想死锁或活锁，饥饿能够被解开。

29.死锁发生的四个必要条件：

互斥条件、请求保持、不可剥夺、环路等待

30.线程的调度算法：

·选择调度算法准则：第一是公平性，第二就是CPU的有效利用。

·从大的分类来说，线程调度算法可以分为非抢占式和抢占式算法。

·典型的线程调度算法：1先到先服务算法2时间片轮转法3优先级调度

31.线程组和线程池的区别：

·线程组存在的意义首要的原因是安全。Java默认创建的线程都是属于系统线程组，而同一个线程组的线程是可以相互修改对方的数据的。但如果不在同一线程组中，那么就不能跨线程组修改数据，可以从一定程度上保证数据安全

·线程池存在的意义首要作用是效率。线程的创建和结束都需要消耗一定的系统时间，不同的创建和删除线程会浪费大量的时间，所以在创建出一条线程并使其在执行完任务后不结束，而使其进入休眠状态，等待使用的时候再唤醒，那么可以节省一定的空间。

·线程组合线程池共有的特点：都是管理一定数量的线程，都可以对线程进行控制，包括休眠、唤醒、结束、创建、中断，但并不一定全部包含这些操作。

32.线程和进程的区别：

·进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。他可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的代码，还包括当前的活动，通过程序计数器的值和处理寄存器的内容来表示。

·进程是一个执行中的程序，程序是一个没有生命的实体，只有处理器赋予程序声明时，他才能称为一个活动的实体，称为线程

·通常在一个进程中包含若干个线程，他们可以利用进程所拥有的资源。在引入线程的操作系统中，通常把进程当作分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位，由于线程比进程小，基本上不用有系统资源，故对线程掉地锁付出的开销会就很小，能高效的提高系统内多个程序间的并发执行的程度。

·线程和进程的区别在于，子进程和父进程有不同的代码和数据空间，而多个线程则共享数据空间，每个线程有自己的执行堆栈和程序计数器为其执行上下文，多线程主要是为了节约CPU时间，发挥利用，根据具体情况而定，线程的运行中需要使用计算机的内存资源和CPU

·归纳：

地址空间和其他资源：进程间相互独立，同一进程的各线程间共享，某进程内的线程在其他线程不可见

通信：进程间通信IPC，线程间可以直接读写进程数据段来进行通信，需要进程同步和互斥手段的辅助，以保证数据的一致性

调度和切换：线程上下文的切换比进程上下文的切换要快得多

在多线程的OS中，进程不是一个可执行的实体

·进程是具有一定独立运行能力的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位，线程是进程的一个实体，是CPU调度和分派的基本单位。线程自己基本上不拥有资源，值拥有在运行中必不可少的资源，如程序计数器，一组寄存器和栈，但他可与同属一个进程的其他线程共享进程所拥有的全部资源。

33.锁的隔离级别：

·事务的四个基本特性：1原子性：原子性用于表示事物是否完全完成，一个事务的任何更新要在系统上完全的完成，由于某种原因出错，事务不能完成他的全部任务，系统将返回到事务开始的状态。2一致性：事务在系统完整性中实施一致性，这通过保证系统的任何事务最后都处于有效状态来实现。如果事务成功的完成，那么系统中所有的变化将正确的应用，系统处于有效状态，如果在事务中出现错误，那么系统中所有的变化将自动的回滚，系统返回到原始状态，因为事务开始处于一致性状态，所以现在系统仍处于一致性状态。3隔离性：在隔离状态执行事务，使他们好像是系统给定时间内执行的唯一操作，如果有两个事务运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统，这种属性有时称为串行化，为了防止事务操作的混淆，必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据，重要的是，在隔离状态执行事务，系统的状态有可能不一致，在结束事务之前，应确保系统处于一致状态。但是每个单独的事务中，系统的状态可能会发生变化，如果事务不在隔离状态下运行，他就可能从系统中访问数据，而系统可能处于不一致的状态，通过提供事务隔离，可以组织这类事件的发生。4持久性：持久性意味着一旦事务执行成功，在系统中产生的变化将是永久的，应该存在一些检查点防止在系统失败是丢失信息，甚至硬件本身失败，系统的状态仍能通过在日志中记录事务完成的任务进行重建。持久性的概念允许开发者认为不管系统以后发生了什么变化，完成的事务是系统永久的部分。

·数据库并发操作的异常情形：

·丢失更新：两个事务同时更新一行数据，但是第二个事务却中途失败退出，导致对数据的两个修改都失效

·脏读：一个事务开始读取了某行数据，但是另一个事务已经更新了此数据但没有能够及时的提交，这是很危险的，因为可能所有的操作都被回滚。

·非重复读：一个事务对同一行数据重复读取两次，但都得到不同的结果，同一查询在同一事物中多次进行，由于其他提交事务所做的修改或删除，每次都返回不同的结果集，此时发生重复读。

·二类丢失更新：无法重复读取的特例，有两个并发事务同时读取一行数据，然后其中一个对他进行修改提交，而另一个也进行了修改提交，这就会造成第一次写操作失败。

·幻想读：事务在操作过程中进行两次查询，第二次查询的结果包含了第一次查询的未出现的数据，这是因为两次查询之间有另外一个事务插入数据造成的。

·数据库的隔离级别：

·未提交读：就是读未提交的意思，意思就是即使一个更新语句没有提交，但是别的事务可以读到这个改变。允许任务读取数据库中未提交的数据更改，也称为脏读

·提交读：就是读提交的意思，可防止脏读，意思就是语句提交以后即执行了COMMIT以后，别的事务就能读到这个改变，只能读到已经提交的数据。

·可重复读：这是说在同一事务里面先后执行同一个查询语句的时候，得到的结果都是一样的，在同一个事物内查询都是事务开始时刻一致的，InnoDB默认级别。在SQL标准中，该隔离级别消除了不可重复读，但还存在幻想读。

·串行读：就是序列化的意思，意思就是说这个事务执行的时候，不允许别的事物并发执行，完全串行化的读，每次读都需要获得表即共享锁，读写都会相互阻塞。

34. DR脏读 LU丢失/更新 NRR不可重复读/读 PR幻读 SLU二类丢失更新

未提交读 Y Y Y Y Y

已提交读 N N Y Y Y

可重复读 N N N Y N

可串行读 N N N N N

35.Hibernate中三种对象的状态：

·在Hibernate中，对象分为临时态(Transient)、持久态(Persistent)、游离态(Detached)

·处于持久态的对象也称为PO(PersistenceObject)，临时对象和游离对象也称为VO(ValueObject)

·临时状态：由new命令开辟出来的内存空间的java对象，临时对象在内存孤立存在，他是携带信息的载体，不和数据库的数据有任何关联关系。a.如果没有变量对该对象进行引用，他将被gc回收。b.在Hibernate中可通过session的save和saveOrUpdate方法将瞬时对象与数据库相关联，并将数据对应的插入数据库中，此时该临时对象转变成持久对象

·持久状态：处于该状态的对象在数据库中具有对应的记录，并拥有一个持久化标识，通过session的get和load等方法获得的对象都是持久化对象。a.如果使用Hibernate的delete方法，对应的持久化对象就会变为临时对象，因为数据库中对应的数据已经删除，该对象不再与数据库中的值相关联。b.当一个session执行close、clear、evict之后，持久化对象变成游离对象，此时该对象虽然具有数据库识别值，但他已经不在Hibernate的持久层的管理下。C.持久层对象具有如下特点：和session实例关联、在数据库中有与之对应的关联的记录，并拥有持久化标志

·游离状态：当某持久化对象关联的session被关闭后，该持久化对象转变为游离对象，当游离对象被重新关联到session上的时候，又再次转变成持久对象。a.通过update、saveOrUpdate等方法将游离态对象转换成持久化对象。b.如果使用delete方法，游离态对象就会变成临时对象，因为数据库中的对应数据已经被删除，该对象不再与数据库进行关联。C.游离态特点：本质上与瞬时态相同，在没有任何变量引用他的时候，JVM会适当的将其回收。游离态比瞬时态多了一个数据库记录标识值。

·session的不同操作对对象状态的影响：save：临时变持久、lock：将对象同session相关联但不强制更新、saveOrUpdate：方法对于临时态，执行save方法，对于游离态，执行update方法、load和get：这两个方法加载对象都是位于session缓存中，属于持久对象、delete：如果传入的是持久类，则执行delete，如果传入的是游离对象，先和session关联，再delete、evict：从session的缓存中删除一个持久对象

36.get和load的区别：

·get和load方法是根据id取得一个记录。

·从返回结果上对比：load方式检索不到会抛出org.hibernate.ObjectNotFoundException异常，而get方法检索不到会返回null。

·从检索执行机制上对比：get方法和find方法都是直接从数据库中检索，而load方法的执行则比较复杂，首先查找session的persistent Context中是否有缓存，如果有直接返回，如果没有判断是否是lazy，如果不是直接访问数据库检索，查到记录返回，查不到抛出异常，如果是lazy需要建立代理对象，对象的initialized属性为false，target属性为null，在访问获得的代理对象的属性的时候，检索数据库，如果找到记录则把记录的对象复制到代理对象target上，并将initalized=true，如果找不到就抛出异常

·根本区别：如果你是用load，那么Hibernate就认为该id对应的对象在数据库中是一定存在的，所以他可以放心的查，他可以放心的使用代理来延迟加载该对象，在用到对象中的其他属性数据时才查找数据库，但是万一数据库中没有这条记录，那么只能抛出异常。（这是由于延迟加载作怪）。由于session的缓存对于Hibernate是各廉价的资源，所有在load先查一下缓存看看对象是否存在，不存在则创建代理。

·对于get方法，Hibernate会确认一下该id对应的数据是否存在，首先在session缓存中找，然后在二级缓存中找，还没有就查找数据库，数据库没有就返回null。

·对于“get方法永远只返回实体类”，这是不对的，get方法如果在session中查找到对象，如果刚好对象在前面是被代理过的，如被load方法使用过，或者被其他延迟对象加载过，那么返回的还是原先的代理对象，而不是实体类对象。如果该代理对象没有被加载实体数据，在从二级缓存中查，但是返回的还是代理对象，只不过已经加载了实体数据

·get方法首先查询session，然后查询二级缓存。而load方法创建时首先查找session缓存，没有就创建代理，实际使用数据时才会查询二级缓存和数据库

37.struts2 和springMVC的区别：

·struts2是类级别的拦截器，一个类对应一个request上下文，springMVC是方法级别拦截器，一个方法对应一个request上下文，而方法同时又跟一个url对应，所以说从架构本身上springMVC就容易实现restful url，而struts2实现起来就要费力，因为struts2的Action的一个方法可以对应一个url，而且类属性却被所有方法共享，这就无法用注解或其他方法标识所属方法。

·springMVC的方法之间的参数是基本独立的，独享request和response数据，请求数据通过参数获取，处理结果通过modelMap交还给框架，方法之间不共享变量，而struts2虽然方法之间也是独立的，但是所有Action变量是共享的。

·由于struts2要针对每个request进行封装，把request，session等Servlet生命周期的变量封装成一个Map，供给每个Action使用，并保证线程安全，所以在原则上是比较消耗内存的。

·拦截器实现机制上，struts2有以自己的interceptor机制，springMvc是独立的AOP方式，这样导致了struts2的配置文件量比springMVC要大

·SpringMVC的入口是Servlet，为struts2的是filter，这就导致了两者的机制不同

·springMVC集成了Ajax，使用非常方便，只需要一个注解ResponseBody就可以实现，然后直接返回相应文本即可，而struts2拦截器集成了ajax，在Action中处理的时候必须要安装插件或自己写代码集成进去

·springMVC验证支持JSR303，处理起来更加的灵活，而struts2验证比较繁琐

·springMVC是spring无缝的，项目的管理和安全高于struts2

·设计思想上，struts2更符合OOP的编程思想，springMVC就比较谨慎，在Servlet上扩展

·springMVC开发性能高

·springMVC可以认为100%零配置

38.Hibernate和mybatis的区别：

·两者最大的区别：针对简单逻辑，Hibernate和mybatis都有相应的代码生成工具，可以生成简单的DAO层方法。针对高级查询，mybatis需要手动编写sql语句，以及ResultMap，而Hibernate有良好的映射机制，开发者无需关心sql语句的生成与结果映射，更专注于业务流程。

·开发难度对比：Hibernate的开发难度要大于mybatis，主要是由于Hibernate比较复杂，庞大，学习周期比较长。而mybatis则相对来说要简单一些，并且mybatis主要依赖sql语句的书写，让开发者感觉更熟悉

·sql书写比较：mybatis的sql语句是手动编辑的，所以可以按照需求指定查询的字段，不过没有自己的日志系统，所以要借助log4j来进行记录日志。Hibernate也可以自己写sql来指定要查询的字段，但这样就破坏了Hibernate开发的简洁性，不过Hibernate有自己的日志统计

·数据库性能比较：mybatis由于所有的sql都是依赖数据库书写的，所以扩展性，迁移性比较差。Hibernate与数据库具体的关联都在xml上，所以hql对具体是用什么数据库并不是很关心

·缓存机制比较：相同点：Hibernate和mybatis的二级缓存除了采用系统默认的缓存机制外，都可以通过实现你自己的缓存或为其他第三方缓存方案，创建适配器来完全覆盖缓存行为。不同点：Hibernate的二级缓存配置在sessionFactory生成的配置文件中进行详细配置，然后再在具体的表-对象映射文件中配置缓存

·mybatis的二级缓存机制都是在每个具体的表-对象映射中详细配置，这样针对不同的表可以自定义不同的缓存机制，并且mybatis可以在命名空间中共享相同的缓存配置和实例，通过cache-ref来实现

·两者相比，因为Hibernate对查询对象有良好的管理机制，用户不需要关心sql，所以在使用二级缓存时如果出现脏数据，系统会报出错误提示。

·而mybatis在这一方面，使用二级缓存时需要特别小心，如果不能完全确定数据更新操作的波动范围，避免cache的盲目使用否则脏数据的出现会更系统的运行带来很大的隐患。

·Hibernate与MyBatis都可以是通过SessionFactoryBuider由XML配置文件生成SessionFactory，然后由SessionFactory 生成Session，最后由Session来开启执行事务和SQL语句。而MyBatis的优势是MyBatis可以进行更为细致的SQL优化，可以减少查询字段，并且容易掌握。Hibernate的优势是DAO层开发比MyBatis简单，Mybatis需要维护SQL和结果映射。数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。有更好的二级缓存机制，可以使用第三方缓存。MyBatis本身提供的缓存机制不佳。

39.IOC和AOP的原理和区别：scope是什么意思

·IOC（Inversion of control）是指容器控制程序对象之间的关系，而不是传统的实现，有程序代码直接操控。控制权限由应用代码中转到了外部容器，控制权的转移是所谓反转，碎语spring而言，就是由spring来控制对象的生命周期和对象之间的关系，IOC也叫做依赖注入，即组件之间的依赖关系有容器在运行期间决定

·在spring的工作方式之中，所有的类都会在spring容器中登记，告诉spring这是什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你的东西交给其他需要的程序，所有类的创建和销毁都由spring控制，也就是说控制对象的生存周期不再是引用他的对象，而是spring，对于具体的对象而言，以前是他控制其他对象，现在是所有对象都被spring控制，所以这叫做控制反转

·在系统运行中，动态的向某个对象提供他所需要的其他对象

·依赖注入的思想是通过反射机制实现的，在实例化一个类的时候，他通过反射调用类中set方法将实现保存在HashMap中的类属性注入到类中。总而言之，在传统的对象创建方法中通常有调用者来创建被调用的实例，而在spring中创建被调用者的工作有spring来完成，然后注入调用者，就是所谓的依赖注入或控制反转

·IOC的优点：降低了组件之间的耦合，降低了业务对象之间替换的复杂性，使之能更灵活的管理对象。

·AOP（aspect oriented programming）:AOP面向方法编程基于IOC，是对OOP的有益补充。

·AOP利用一种称为横切的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为：aspect，即方面。所谓方面，简单来说，就是那些与业务无关却为业务模块所共同调用的逻辑或责任封装起来，比如日志记录，便于减少系统的重复代码，降低了模块间的耦合度，并有利于未来的可操作性和可维护性。

·AOP代表的是一个横向的关系，将对象比作一个空心额圆柱体，其中封装的是对象的属性和行为，则面向方法编程的方法，就是将这个圆柱体以切面的形式剖开，选择性的提供业务逻辑，而剖开的切面，也就是所谓的方面了，然后他又将这些剖面复原，不留痕迹，但完成了效果。

·实现AOP的技术，主要分为两大类，一个是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行，而是采用静态植入的方法，引入特定的语法创建方面，从而使编译器可以在编译期间植入有关方面的代码。

·spring实现AOP：jdk动态代理和cglib代理

·AOP使用场景：权限查看、缓存、内容传递、延迟加载、调试、日志记录，跟踪，优化，校准、性能优化、持久化、资源池、同步、事务管理

·java优化：

\*\*

**1.没有必要时请不用使用静态变量**

    使用Java的开发者都知道，当某个对象被定义为stataic变量所引用，这个对象所占有的内存将不会被回收。有时，开发者会将经常调用的对象或者变量 定义为static，以便提高程序的运行性能。因此，不是常用到的对象或者变量，不要定义为static类型的变量，尤其是静态类对象的定义，一定要仔细 考虑是否有必要。例如

public class X{

static Y a = new Y();

}

    类X创建了，没有被回收的话，静态变量a一直占用内存。

**2.充分利用单例机制**

    实用单例可以减少对资源的加载，缩短运行的时间，提高系统效率。但是，单例并不是所有地方都适用于。简单来说，单例可以适用于以下两个方面：

1.   控制资源的使用，通过线程同步来控制资源的并发访问；

2.   控制实例的产生，以达到节约资源的目的；

**3. 减少对象的创建**

尽量避免在经常调用的方法中循环使用new对象，由于系统不仅要花费时间来创建对 象，而且还要花时间对这些对象进行垃圾回收和处理。设计模式中的享元模式就是为了减少对象的多次创建而来的。在我们可以控制的范围内，最大限度的重用对 象；在有些时候，最好能用基本的数据类型或数组来替代对象。

**4.使用final修饰符**

带有final修饰符的类是不可派生的。在Java核心API中，有许多应用 final的例子，例如java.lang.String。为String类指定final防止了使用者覆盖length()方法。另外，如果一个类是 final的，则该类所有方法都是final的。Java编译器会寻找机会内联（inline）所有的final方法（这和具体的编译器实现有关）。此举 能够使性能平均提高50%。

**5.尽量使用局部变量**

调用方法时传递的参数以及在调用中创建的临时变量都保存在分配给改方法的栈（Stack）中，速度较快。其他变量，如静态变量、实例变量等，都在堆（Heap）中创建，速度较慢。

**6.处理好包装类型和基本类型两者的使用场所**

    虽然包装类型和基本类型在使用过程中是可以相互转换，但它们两者所产生的内存区域是完全不同的，基本类型数据产生和处理都在栈中处理，而包装类型是对象，是在堆中产生实例。在集合类对象，有对象方面需要的处理适用包装类型，其他的情况，建议提倡使用基本类型。

7.学会用StringBuilder和StringBuffer  
    这个两个类的区别就不用说了吧，单线程使用StringBuilder，多线程情况下使用StringBuffer，这样性能会有很大提升。

**7.尽量不要使用finalize方法**

实际上，将资源清理放在finalize方法中完成是非常不好的选择。由于GC的工作量很大，尤其是回收Young代内存时，大都会引起应用程序暂停，所以再选择使用finalize方法进行资源清理，会导致GC负担更大，程序运行效率更差。

**8.尽量使用基本数据类型代替对象**

String str = "hello";

上面这种方式会创建一个“hello”字符串，而且JVM的字符缓存池还会缓存这个字符串。

String str = new String("hello");

此时程序除创建字符串外，str所引用的String对象底层还包含一个char[]数组，这个char[]数组依次存放了h,e,l,l,o

   这个问题我也写个一个博客，请查看我写的有关性能优化的博客。

**9.学会使用HashMap、ArrayList**

HashTable、Vector等使用在多线程的场合，内部使用了同步机制，这个会降低程序的性能。  
10. 深入理解HashMap原理

当你要创建一个比较大的hashMap时，充分利用另一个构造函数

public HashMap(int initialCapacity, float loadFactor)避免HashMap多次进行了hash重构,扩容是一件很耗费性能的事，在默认initialCapacity只有16，而 loadFactor是 0.75，需要多大的容量，你最好能准确的估计你所需要的最佳大小，同样的Hashtable，Vectors也是一样的道理。

**11.减少对变量的重复计算**

如

for(int i=0;i<list.size();i++)

应该改写为：

for(int i=0,len=list.size();i<len;i++)

或者

for(int i = list.size(); I > -1; i--)

并且在循环中应该避免使用复杂的表达式，在循环中，循环条件会被反复计算，如果不使用复杂表达式，而使循环条件值不变的话，程序将会运行的更快。

**12.避免不必要的对象创建**

如

A a = new A();

if(i==1){list.add(a);}

应该改为

if(i==1){

A a = new A();

list.add(a);}

**13.尽量在finally块中释放资源**

程序中使用到的资源应当被释放，以避免资源泄漏。这最好在finally块中去做。不管程序执行的结果如何，finally块总是会执行的，以确保资源的正确关闭。

**14.使用移位来代替乘法或者除法（'a/b'，仅适合2^n情况）的操作**

"/"是一个代价很高的操作，使用移位的操作将会更快和更有效

如

int num = a / 4;

int num = a / 8;

应该改为

int num = a >> 2;

int num = a >> 3;

但注意的是使用移位应添加注释，因为移位操作不直观，比较难理解

同样的，对于'\*'操作，使用移位的操作将会更快和更有效

如

int num = a \* 4;

int num = a \* 8;

应该改为

int num = a << 2;

int num = a << 3;

**15. 确定StringBuffer的容量**

StringBuffer 的构造器会创建一个默认大小（通常是16）的字符数组。在使用中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再 丢弃旧的数组。在大多数情况下，你可以在创建 StringBuffer的时候指定大小，这样就避免了在容量不够的时候自动增长，以提高性能。

如：StringBufferbuffer = new StringBuffer(1000);

**16.尽量早释放无用对象的引用**

大部分时，方法局部引用变量所引用的对象 会随着方法结束而变成垃圾，因此，大部分时候程序无需将局部，引用变量显式设为null。

例如：

Public **void** doJob(){

Object obj =**new** Object();

……

Obj=**null**;

}

上面这个就没必要了，随着方法doJob()的执行完成，程序中obj引用变量的作用域会被gc回收。但是如果是改成下面：

public **void** doJob(){

Object obj =**new** Object();

……

Obj=**null**;

//执行耗时，耗内存操作；或调用耗时，耗内存的方法

……

}

这时候就有必要将obj赋值为null，可以尽早的释放对Object对象的引用。

**17.尽量避免使用split**

除非是必须的，否则应该避免使用split，split由于支持正则表达式，所以效 率比较低，如果是频繁的几十，几百万的调用将会耗费大量资源，如果确实需要频繁的调用split，可以考虑使用apache的 StringUtils.split(string,char)，频繁split的可以缓存结果。

**18.ArrayList & LinkedList**

一个是线性表，一个是链表，一句话，随机查询尽量使用 ArrayList，ArrayList优于LinkedList，LinkedList还要移动指针，添加删除的操作LinkedList优于 ArrayList，ArrayList还要移动数据，不过这是理论性分析，事实未必如此，重要的是理解好数据结构。

**19.尽量使用System.arraycopy ()代替通过来循环复制数组**

System.arraycopy()要比通过循环来复制数组快的多

**20.尽量缓存经常使用的对象**

尽可能将经常使用的对象进行缓存，可以使用数组，或HashMap的容器来进行缓存，但这种方式可能导致系统占用过多的缓存，性能下降，推荐可以使用一些第三方的开源工具，如EhCache，Oscache进行缓存，他们基本都实现了FIFO/FLU等缓存算法。

**21.尽量避免非常大的内存分配**

有时候问题不是由当时的堆状态造成的，而是因为分配失败造成的。分配的内存块都必须是连续的，而随着堆heap越来越满，找到较大的连续块越来越困难。

**22.慎用异常**

当创建一个异常时，需要收集一个栈跟踪(stack track)，这个栈跟踪用于描述异常是在何处创建的。构建这些栈跟踪时需要为运行时栈做一份快照，正是这一部分开销很大。当需要创建一个 Exception 时，JVM 不得不说：先别动，我想就您现在的样子存一份快照，所以暂时停止入栈和出栈操作。栈跟踪不只包含运行时栈中的一两个元素，而是包含这个栈中的每一个元素。

如果您创建一个 Exception ，就得付出代价。好在捕获异常开销不大，因此可以使用 try-catch 将核心内容包起来。从技术上讲，您甚至可以随意地抛出异常，而不用花费很大的代价。招致性能损失的并不是 throw 操作——尽管在没有预先创建异常的情况下就抛出异常是有点不寻常。真正要花代价的是创建异常。幸运的是，好的编程习惯已教会我们，不应该不管三七二十一就 抛出异常。异常是为异常的情况而设计的，使用时也应该牢记这一原则。

\*\*

1、生成对象时，合理分配空间和大小  
new ArrayList(100);

2、优化for循环  
Vector vect = new Vector(1000);  
for( inti=0; i＜vect.size(); i++){  
...  
}  
for循环部分改写成：  
int size = vect.size();  
for( int i=0; i＞size; i++){  
...  
}   
如果size=1000，就可以减少1000次size()的系统调用开销，避免了循环体重复调用。

3、new一个实例对象，new在什么位置(尽量在使用时再创建该对象)。

4、异常处理技巧

5、尽量使用局部变量和静态变量

6、尽量不适用多线程同步

7、尽可能的使用Java自身提供的API

8、尽量减少I/O操作(控制台、日志)

9、尽量使用缓存流(尽可能使用带有Buffer的类代替没有Buffer的类，BufferedReader、BufferedWriter、BufferedInputStream)

10、sql优化、存储过程、视图、连接池(C3P0 、DBCP)

11、数据库数据分级存储  
将经常访问的数据和访问频度低的数据，分别存放到不同的分区，甚至存放到不同的数据库服务器，以便合进分配硬盘I/O及系统I/O。

12、缓存策略  
如果有些数据要经常要从数据库中读取，同时，这些数据又不经常变化，这些数据就可以在系统中缓存起来，使用时直接读取缓存，而不用频繁的访问数据库读取数据。  
缓存工作可以在系统初始化时一次性读取数据，特别是一些只读的数据，当数据更新时更新数据库内容，同时更新缓存的数据值。  
Java常用的缓存技术产品有：Redis、MemoryCache、OSCache等。

13、html静态化

14、不用保存太多的信息在HttpSession中

15、在使用大数据对象时，因此建议在对象使用完毕后，手动设置成null(避免内存溢出)

\*\*

**1.1 不用new关键词创建类的实例**

用new关键词创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了Cloneable接口，我们可以调用它的clone()方法。clone()方法不会调用任何类构造函数。

在使用设计模式（Design Pattern）的场合，如果用Factory模式创建对象，则改用clone()方法创建新的对象实例非常简单。例如，下面是Factory模式的一个典型实现：

|  |
| --- |
| public static Credit getNewCredit() {return new Credit();} |

改进后的代码使用clone()方法，如下所示：

|  |
| --- |
| private static Credit BaseCredit = new Credit(); public static Credit getNewCredit() { return (Credit) BaseCredit.clone(); } |

上面的思路对于数组处理同样很有用。

**1.2 使用非阻塞I/O**

版本较低的JDK不支持非阻塞I/O API。为避免I/O阻塞，一些应用采用了创建大量线程的办法（在较好的情况下，会使用一个缓冲池）。这种技术可以在许多必须支持并发I/O流的应用中见 到，如Web服务器、报价和拍卖应用等。然而，创建Java线程需要相当可观的开销。

JDK 1.4引入了非阻塞的I/O库（java.nio）。如果应用要求使用版本较早的JDK，在[这里](http://www.cs.berkeley.edu/%7Emdw/proj/java-nbio/download.html)有一个支持非阻塞I/O的软件包。

请参见Sun中国网站的《[调整Java的I/O性能](http://www.sun.com.cn/developers/scda/java-io.html)》。

**1.3 慎用异常**

异常对性能不利。抛出异常首先要创建一个新的对象。Throwable接口的构造函数调用名为fillInStackTrace()的本地 （Native）方法，fillInStackTrace()方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，VM就必须调整调用堆栈，因为在处理过 程中创建了一个新的对象。

异常只能用于错误处理，不应该用来控制程序流程。

**1.4 不要重复初始化变量**

默认情况下，调用类的构造函数时， Java会把变量初始化成确定的值：所有的对象被设置成null，整数变量（byte、short、int、long）设置成0，float和 double变量设置成0.0，逻辑值设置成false。当一个类从另一个类派生时，这一点尤其应该注意，因为用new关键词创建一个对象时，构造函数链 中的所有构造函数都会被自动调用。

**1.5 尽量指定类的final修饰符**

带有final修饰符的类是不可派生的。在Java核心API中，有许多应用final的例子，例如java.lang.String。为String类指定final防止了人们覆盖length()方法。

另外，如果指定一个类为final，则该类所有的方法都是final。Java编译器会寻找机会内联（inline）所有的final方法（这和具体的编译器实现有关）。此举能够使性能平均提高50%。

**1.6 尽量使用局部变量**

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快。其他变量，如静态变量、实例变量等，都在堆（Heap）中创建，速度较慢。另外，依赖于具体的编译器/JVM，局部变量还可能得到进一步优化。请参见《[尽可能使用堆栈变量](http://www-900.ibm.com/developerWorks/java/praxis/pr35.shtml)》。

**1.7 乘法和除法**

考虑下面的代码：

|  |
| --- |
| for (val = 0; val < 100000; val +=5) { alterX = val \* 8; myResult = val \* 2; } |

用移位操作替代乘法操作可以极大地提高性能。下面是修改后的代码：

|  |
| --- |
| for (val = 0; val < 100000; val += 5) { alterX = val << 3; myResult = val << 1; } |

修改后的代码不再做乘以8的操作，而是改用等价的左移3位操作，每左移1位相当于乘以2。相应地，右移1位操作相当于除以2。值得一提的是，虽然移位操作速度快，但可能使代码比较难于理解，所以最好加上一些注释。

前面介绍的改善性能技巧适合于大多数Java应用，接下来要讨论的问题适合于使用JSP、EJB或JDBC的应用。

一、避免在循环条件中使用复杂表达式   
  
在不做编译优化的情况下，在循环中，循环条件会被反复计算，如果不使用复杂表达式，而使循环条件值不变的话，程序将会运行的更快。   
  
例子：

import java.util.vector;

class cel {

void method (vector vector) {

for (int i = 0; i < vector.size (); i++) // violation

; // ...

}

}

更正：

class cel\_fixed {

void method (vector vector) {

int size = vector.size ()

for (int i = 0; i < size; i++)

; // ...

}

}

二、为'vectors' 和 'hashtables'定义初始大小   
  
jvm为vector扩充大小的时候需要重新创建一个更大的数组，将原原先数组中的内容复制过来，最后，原先的数组再被回收。可见vector容量的扩大是一个颇费时间的事。   
通常，默认的10个元素大小是不够的。你最好能准确的估计你所需要的最佳大小。   
  
例子：

import java.util.vector;

public class dic {

public void addobjects (object[] o) {

// if length > 10, vector needs to expand

for (int i = 0; i< o.length;i++) {

v.add(o); // capacity before it can add more elements.

}

}

public vector v = new vector(); // no initialcapacity.

}

更正：   
自己设定初始大小。

public vector v = new vector(20);

public hashtable hash = new hashtable(10);

参考资料：   
dov bulka, "java performance and scalability volume 1: server-side programming   
techniques" addison wesley, isbn: 0-201-70429-3 pp.55 – 57   
  
三、在finally块中关闭stream   
  
程序中使用到的资源应当被释放，以避免资源泄漏。这最好在finally块中去做。不管程序执行的结果如何，finally块总是会执行的，以确保资源的正确关闭。   
           
例子：

import java.io.\*;

public class cs {

public static void main (string args[]) {

cs cs = new cs ();

cs.method ();

}

public void method () {

try {

fileinputstream fis = new fileinputstream ("cs.java");

int count = 0;

while (fis.read () != -1)

count++;

system.out.println (count);

fis.close ();

} catch (filenotfoundexception e1) {

} catch (ioexception e2) {

}

}

}

           
更正：   
在最后一个catch后添加一个finally块   
  
参考资料：   
peter haggar: "practical java - programming language guide".   
addison wesley, 2000, pp.77-79   
四、使用'system.arraycopy ()'代替通过来循环复制数组   
  
'system.arraycopy ()' 要比通过循环来复制数组快的多。   
           
例子：

public class irb

{

void method () {

int[] array1 = new int [100];

for (int i = 0; i < array1.length; i++) {

array1 [i] = i;

}

int[] array2 = new int [100];

for (int i = 0; i < array2.length; i++) {

array2 [i] = array1 [i]; // violation

}

}

}

           
更正：

public class irb

{

void method () {

int[] array1 = new int [100];

for (int i = 0; i < array1.length; i++) {

array1 [i] = i;

}

int[] array2 = new int [100];

system.arraycopy(array1, 0, array2, 0, 100);

}

}

           
参考资料：   
http://www.cs.cmu.edu/~jch/java/speed.html   
  
五、让访问实例内变量的getter/setter方法变成”final”   
  
简单的getter/setter方法应该被置成final，这会告诉编译器，这个方法不会被重载，所以，可以变成”inlined”   
  
例子：

class maf {

public void setsize (int size) {

\_size = size;

}

private int \_size;

}

更正：

class daf\_fixed {

final public void setsize (int size) {

\_size = size;

}

private int \_size;

}

参考资料：   
warren n. and bishop p. (1999), "java in practice", p. 4-5   
addison-wesley, isbn 0-201-36065-9   
  
六、避免不需要的instanceof操作   
  
如果左边的对象的静态类型等于右边的，instanceof表达式返回永远为true。   
           
例子：        

public class uiso {

public uiso () {}

}

class dog extends uiso {

void method (dog dog, uiso u) {

dog d = dog;

if (d instanceof uiso) // always true.

system.out.println("dog is a uiso");

uiso uiso = u;

if (uiso instanceof object) // always true.

system.out.println("uiso is an object");

}

}

           
更正：           
删掉不需要的instanceof操作。   
        

class dog extends uiso {

void method () {

dog d;

system.out.println ("dog is an uiso");

system.out.println ("uiso is an uiso");

}

}

七、避免不需要的造型操作   
  
所有的类都是直接或者间接继承自object。同样，所有的子类也都隐含的“等于”其父类。那么，由子类造型至父类的操作就是不必要的了。   
例子：

class unc {

string \_id = "unc";

}

class dog extends unc {

void method () {

dog dog = new dog ();

unc animal = (unc)dog; // not necessary.

object o = (object)dog; // not necessary.

}

}

           
更正：        

class dog extends unc {

void method () {

dog dog = new dog();

unc animal = dog;

object o = dog;

}

}

参考资料：   
nigel warren, philip bishop: "java in practice - design styles and idioms   
for effective java".  addison-wesley, 1999. pp.22-23   
八、如果只是查找单个字符的话，用charat()代替startswith()   
  
用一个字符作为参数调用startswith()也会工作的很好，但从性能角度上来看，调用用string api无疑是错误的!   
           
例子：

public class pcts {

private void method(string s) {

if (s.startswith("a")) { // violation

// ...

}

}

}

           
更正           
将'startswith()' 替换成'charat()'.

public class pcts {

private void method(string s) {

if ('a' == s.charat(0)) {

// ...

}

}

}

           
参考资料：   
dov bulka, "java performance and scalability volume 1: server-side programming   
techniques"  addison wesley, isbn: 0-201-70429-3   
九、使用移位操作来代替'a / b'操作   
  
"/"是一个很“昂贵”的操作，使用移位操作将会更快更有效。   
  
例子：

public class sdiv {

public static final int num = 16;

public void calculate(int a) {

int div = a / 4; // should be replaced with "a >> 2".

int div2 = a / 8; // should be replaced with "a >> 3".

int temp = a / 3;

}

}

更正：

public class sdiv {

public static final int num = 16;

public void calculate(int a) {

int div = a >> 2;

int div2 = a >> 3;

int temp = a / 3; // 不能转换成位移操作

}

}

十、使用移位操作代替'a \* b'   
  
同上。   
[i]但我个人认为，除非是在一个非常大的循环内，性能非常重要，而且你很清楚你自己在做什么，方可使用这种方法。否则提高性能所带来的程序晚读性的降低将是不合算的。   
  
例子：

public class smul {

public void calculate(int a) {

int mul = a \* 4; // should be replaced with "a << 2".

int mul2 = 8 \* a; // should be replaced with "a << 3".

int temp = a \* 3;

}

}

更正：

package opt;

public class smul {

public void calculate(int a) {

int mul = a << 2;

int mul2 = a << 3;

int temp = a \* 3; // 不能转换

}

}

十一、在字符串相加的时候，使用 ' ' 代替 " "，如果该字符串只有一个字符的话   
  
  
例子：

public class str {

public void method(string s) {

string string = s + "d" // violation.

string = "abc" + "d" // violation.

}

}

更正：   
将一个字符的字符串替换成' '

public class str {

public void method(string s) {

string string = s + 'd'

string = "abc" + 'd'

}

}

十二、不要在循环中调用synchronized(同步)方法   
  
方法的同步需要消耗相当大的资料，在一个循环中调用它绝对不是一个好主意。   
  
例子：

import java.util.vector;

public class syn {

public synchronized void method (object o) {

}

private void test () {

for (int i = 0; i < vector.size(); i++) {

method (vector.elementat(i)); // violation

}

}

private vector vector = new vector (5, 5);

}

更正：   
不要在循环体中调用同步方法，如果必须同步的话，推荐以下方式：

import java.util.vector;

public class syn {

public void method (object o) {

}

private void test () {

synchronized{//在一个同步块中执行非同步方法

for (int i = 0; i < vector.size(); i++) {

method (vector.elementat(i));

}

}

}

private vector vector = new vector (5, 5);

}

十三、将try/catch块移出循环   
  
把try/catch块放入循环体内，会极大的影响性能，如果编译jit被关闭或者你所使用的是一个不带jit的jvm，性能会将下降21%之多!   
           
例子：        

import java.io.fileinputstream;

public class try {

void method (fileinputstream fis) {

for (int i = 0; i < size; i++) {

try { // violation

\_sum += fis.read();

} catch (exception e) {}

}

}

private int \_sum;

}

           
更正：           
将try/catch块移出循环        

void method (fileinputstream fis) {

try {

for (int i = 0; i < size; i++) {

\_sum += fis.read();

}

} catch (exception e) {}

}

           
参考资料：   
peter haggar: "practical java - programming language guide".   
addison wesley, 2000, pp.81 – 83   
  
十四、对于boolean值，避免不必要的等式判断   
  
将一个boolean值与一个true比较是一个恒等操作(直接返回该boolean变量的值). 移走对于boolean的不必要操作至少会带来2个好处：   
1)代码执行的更快 (生成的字节码少了5个字节)；   
2)代码也会更加干净 。   
  
例子：

public class ueq

{

boolean method (string string) {

return string.endswith ("a") == true; // violation

}

}

更正：

class ueq\_fixed

{

boolean method (string string) {

return string.endswith ("a");

}

}

十五、对于常量字符串，用'string' 代替 'stringbuffer'   
  
常量字符串并不需要动态改变长度。   
例子：

public class usc {

string method () {

stringbuffer s = new stringbuffer ("hello");

string t = s + "world!";

return t;

}

}

更正：   
把stringbuffer换成string，如果确定这个string不会再变的话，这将会减少运行开销提高性能。   
  
十六、用'stringtokenizer' 代替 'indexof()' 和'substring()'   
  
字符串的分析在很多应用中都是常见的。使用indexof()和substring()来分析字符串容易导致 stringindexoutofboundsexception。而使用stringtokenizer类来分析字符串则会容易一些，效率也会高一些。   
  
例子：

public class ust {

void parsestring(string string) {

int index = 0;

while ((index = string.indexof(".", index)) != -1) {

system.out.println (string.substring(index, string.length()));

}

}

}

参考资料：   
graig larman, rhett guthrie: "java 2 performance and idiom guide"   
prentice hall ptr, isbn: 0-13-014260-3 pp. 282 – 283   
  
十七、使用条件操作符替代"if (cond) return; else return;" 结构   
  
条件操作符更加的简捷   
例子：

public class if {

public int method(boolean isdone) {

if (isdone) {

return 0;

} else {

return 10;

}

}

}

更正：

public class if {

public int method(boolean isdone) {

return (isdone ? 0 : 10);

}

}

十八、使用条件操作符代替"if (cond) a = b; else a = c;" 结构   
  
例子：

public class ifas {

void method(boolean istrue) {

if (istrue) {

\_value = 0;

} else {

\_value = 1;

}

}

private int \_value = 0;

}

更正：

public class ifas {

void method(boolean istrue) {

\_value = (istrue ? 0 : 1); // compact expression.

}

private int \_value = 0;

}

十九、不要在循环体中实例化变量   
  
在循环体中实例化临时变量将会增加内存消耗   
  
例子：        

import java.util.vector;

public class loop {

void method (vector v) {

for (int i=0;i < v.size();i++) {

object o = new object();

o = v.elementat(i);

}

}

}

           
更正：           
在循环体外定义变量，并反复使用        

import java.util.vector;

public class loop {

void method (vector v) {

object o;

for (int i=0;i<v.size();i++) {

o = v.elementat(i);

}

}

}

二十、确定 stringbuffer的容量   
  
stringbuffer 的构造器会创建一个默认大小(通常是16)的字符数组。在使用中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再 丢弃旧的数组。在大多数情况下，你可以在创建stringbuffer的时候指定大小，这样就避免了在容量不够的时候自动增长，以提高性能。   
  
例子：        

public class rsbc {

void method () {

stringbuffer buffer = new stringbuffer(); // violation

buffer.append ("hello");

}

}

           
更正：           
为stringbuffer提供寝大小。        

public class rsbc {

void method () {

stringbuffer buffer = new stringbuffer(max);

buffer.append ("hello");

}

private final int max = 100;

}

           
参考资料：   
dov bulka, "java performance and scalability volume 1: server-side programming   
techniques" addison wesley, isbn: 0-201-70429-3 p.30 – 31   
  
二十一、尽可能的使用栈变量   
  
如果一个变量需要经常访问，那么你就需要考虑这个变量的作用域了。static? local?还是实例变量？访问静态变量和实例变量将会比访问局部变量多耗费2-3个时钟周期。   
           
例子：

public class usv {

void getsum (int[] values) {

for (int i=0; i < value.length; i++) {

\_sum += value[i]; // violation.

}

}

void getsum2 (int[] values) {

for (int i=0; i < value.length; i++) {

\_staticsum += value[i];

}

}

private int \_sum;

private static int \_staticsum;

}

           
更正：           
如果可能，请使用局部变量作为你经常访问的变量。   
你可以按下面的方法来修改getsum()方法：        

void getsum (int[] values) {

int sum = \_sum; // temporary local variable.

for (int i=0; i < value.length; i++) {

sum += value[i];

}

\_sum = sum;

}

           
参考资料：           
peter haggar: "practical java - programming language guide".   
addison wesley, 2000, pp.122 – 125   
  
二十二、不要总是使用取反操作符(!)   
  
取反操作符(!)降低程序的可读性，所以不要总是使用。   
  
例子：

public class dun {

boolean method (boolean a, boolean b) {

if (!a)

return !a;

else

return !b;

}

}

更正：   
如果可能不要使用取反操作符(!)   
  
二十三、与一个接口 进行instanceof操作   
  
基于接口的设计通常是件好事，因为它允许有不同的实现，而又保持灵活。只要可能，对一个对象进行instanceof操作，以判断它是否某一接口要比是否某一个类要快。   
  
例子：

public class insof {

private void method (object o) {

if (o instanceof interfacebase) { } // better

if (o instanceof classbase) { } // worse.

}

}

class classbase {}

interface interfacebase {}

参考资料：   
graig larman, rhett guthrie: "java 2 performance and idiom guide"   
prentice hall ptr, 2000.  pp.207

* 17:26
* 浏览 (26)
* [评论](http://xixian.javaeye.com/blog/664971#comments) (0)
* 分类: [java](http://xixian.javaeye.com/category/102812)

2010-05-12

[缩略显示](http://xixian.javaeye.com/category/102812?show_full=false)

[**Java性能优化技巧**](http://xixian.javaeye.com/blog/664911)

**文章分类:**[**Java编程**](http://www.javaeye.com/blogs/category/java)

转载：http://blog.csdn.net/kome2000/archive/2010/04/28/5537591.aspx   
  
  
[size=small]在JAVA程序中，性能问题的大部分原因并不在于JAVA语言，而是程序本身。养成良好的编码习惯非常重要，能够显著地提升程序性能。   
  
1. 尽量使用final修饰符。   
带 有final修饰符的类是不可派生的。在JAVA核心API中，有许多应用final的例子，例如 java.lang.String。为String类指定final防止了使用者覆盖length()方法。另外，如果一个类是final的，则该类所有 方法都是final的。java编译器会寻找机会内联（inline）所有的final方法（这和具体的编译器实现有关）。此举能够使性能平均提高 50%。   
  
2.尽量重用对象。   
特别是String对象的使用中，出现字符串连接情况时应使用StringBuffer代替，由于系统不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理。因此生成过多的对象将会给程序的性能带来很大的影响。   
  
3. 尽量使用局部变量。   
调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快。其他变量，如静态变量，实例变量等，都在堆（Heap）中创建，速度较慢。   
  
4.不要重复初始化变量。   
默 认情况下，调用类的构造函数时，java会把变量初始化成确定的值，所有的对象被设置成null，整数变量设置成0，float和double变量设置成 0.0，逻辑值设置成false。当一个类从另一个类派生时，这一点尤其应该注意，因为用new关键字创建一个对象时，构造函数链中的所有构造函数都会被 自动调用。   
这里有个注意，给成员变量设置初始值但需要调用其他方法的时候，最好放在一个方法比如initXXX()中，因为直接调用某方法赋值可能会因为类尚未初始化而抛空指针异常，public int state = this.getState();   
  
5.在java+Oracle的应用系统开发中，java中内嵌的SQL语言应尽量使用大写形式，以减少Oracle解析器的解析负担。   
  
6.java编程过程中，进行数据库连接，I/O流操作，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销。   
  
7.过分的创建对象会消耗系统的大量内存，严重时，会导致内存泄漏，因此，保证过期的对象的及时回收具有重要意义。   
JVM的GC并非十分智能，因此建议在对象使用完毕后，手动设置成null。   
  
8.在使用同步机制时，应尽量使用方法同步代替代码块同步。   
  
9.尽量减少对变量的重复计算。   
比如

for(int i=0;i<list.size();i++)

应修改为

for(int i=0,len=list.size();i<len;i++)

10. 采用在需要的时候才开始创建的策略。   
例如：

String str="abc";

if(i==1){ list.add(str);}

应修改为：

if(i==1){String str="abc"; list.add(str);}

11.慎用异常，异常对性能不利。   
抛 出异常首先要创建一个新的对象。Throwable接口的构造函数调用名为fillInStackTrace()的本地方 法，fillInStackTrace()方法检查栈，收集调用跟踪信息。只要有异常被抛出，VM就必须调整调用栈，因为在处理过程中创建了一个新的对 象。   
异常只能用于错误处理，不应该用来控制程序流程。   
  
12.不要在循环中使用Try/Catch语句，应把Try/Catch放在循环最外层。   
Error是获取系统错误的类，或者说是虚拟机错误的类。不是所有的错误Exception都能获取到的，虚拟机报错 Exception就获取不到，必须用Error获取。   
  
13.通过StringBuffer的构造函数来设定他的初始化容量，可以明显提升性能。   
StringBuffer 的默认容量为16，当StringBuffer的容量达到最大容量时，她会将自身容量增加到当前的2倍+2，也就是2\*n+2。无论何时，只要 StringBuffer到达她的最大容量，她就不得不创建一个新的对象数组，然后复制旧的对象数组，这会浪费很多时间。所以给StringBuffer 设置一个合理的初始化容量值，是很有必要的！   
  
14.合理使用java.util.Vector。   
Vector 与StringBuffer类似，每次扩展容量时，所有现有元素都要赋值到新的存储空间中。Vector的默认存储能力为10个元素，扩容加倍。   
vector.add(index,obj) 这个方法可以将元素obj插入到index位置，但index以及之后的元素依次都要向下移动一个位置（将其索引加 1）。 除非必要，否则对性能不利。   
同 样规则适用于remove(int index)方法，移除此向量中指定位置的元素。将所有后续元素左移（将其索引减 1）。返回此向量中移除的元素。所以删除vector最后一个元素要比删除第1个元素开销低很多。删除所有元素最好用 removeAllElements()方法。   
如果要删除vector里的一个元素可以使用 vector.remove(obj)；而不必自己检索元素位置，再删除，如int index = indexOf（obj）;vector.remove(index)；   
  
15.当复制大量数据时，使用 System.arraycopy();   
  
16.代码重构，增加代码的可读性。   
  
17.不用new关键字创建对象的实例。   
用 new关键词创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了Cloneable接口，我们可以调用她的clone() 方法。clone()方法不会调用任何类构造函数。   
下面是Factory模式的一个典型实现。

public static Credit getNewCredit()

{

return new Credit();

}

改进后的代码使用clone() 方法，

private static Credit BaseCredit = new Credit();

public static Credit getNewCredit()

{

return (Credit)BaseCredit.clone();

}

18. 乘除法如果可以使用位移，应尽量使用位移，但最好加上注释，因为位移操作不直观，难于理解。   
  
19.不要将数组声明为：public static final。   
  
20.HaspMap的遍历。

Map<String, String[]> paraMap = new HashMap<String, String[]>();

for( Entry<String, String[]> entry : paraMap.entrySet() )

{

String appFieldDefId = entry.getKey();

String[] values = entry.getValue();

}

利用散列值取出相应的Entry做比较得到结果，取得entry的值之后直接取key和 value。   
  
21.array(数组)和ArrayList的使用。   
array 数组效率最高，但容量固定，无法动态改变，ArrayList容量可以动态增长，但牺牲了效率。   
  
22.单线程应尽量使用 HashMap, ArrayList,除非必要，否则不推荐使用HashTable,Vector，她们使用了同步机制，而降低了性能。   
  
23.StringBuffer,StringBuilder 的区别在于：java.lang.StringBuffer 线程安全的可变字符序列。一个类似于String的字符串缓冲区，但不能修改。StringBuilder与该类相比，通常应该优先使用 StringBuilder类，因为她支持所有相同的操作，但由于她不执行同步，所以速度更快。为了获得更好的性能，在构造StringBuffer或 StringBuilder时应尽量指定她的容量。当然如果不超过16个字符时就不用了。   
相同情况下，使用StringBuilder比使用 StringBuffer仅能获得10%~15%的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用StringBuffer。   
  
24. 尽量使用基本数据类型代替对象。   
  
25.用简单的数值计算代替复杂的函数计算，比如查表方式解决三角函数问题。   
  
26.使用具体类比使用接口效率高，但结构弹性降低了，但现代IDE都可以解决这个问题。   
  
27.考虑使用静态方法，   
如果你没有必要去访问对象的外部，那么就使你的方法成为静态方法。她会被更快地调用，因为她不需要一个虚拟函数导向表。这同事也是一个很好的实践，因为她告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。   
  
28.应尽可能避免使用内在的GET,SET方法。   
android编程中，虚方法的调用会产生很多代价，比实例属性查询的代价还要多。我们应该在外包调用的时候才使用get，set方法，但在内部调用的时候，应该直接调用。   
  
29. 避免枚举，浮点数的使用。   
  
30.二维数组比一维数组占用更多的内存空间，大概是10倍计算。   
  
31.SQLite数据库读取整张表的全部数据很快，但有条件的查询就要耗时30-50MS,大家做这方面的时候要注意，尽量少用，尤其是嵌套查找！ [/size][align=left][/align]

* 16:39
* 浏览 (27)
* [评论](http://xixian.javaeye.com/blog/664911#comments) (0)
* 分类: [java](http://xixian.javaeye.com/category/102812)

2010-05-12

[缩略显示](http://xixian.javaeye.com/category/102812?show_full=false)

[**《java解惑》转**](http://xixian.javaeye.com/blog/664875)

**文章分类:**[**Java编程**](http://www.javaeye.com/blogs/category/java)

转载于:<http://jiangzhengjun.javaeye.com/blog/652623>   
**数值表达式**   
1. 奇偶判断   
  
不要使用 i % 2 == 1 来判断是否是奇数，因为i为负奇数时不成立，请使用 i % 2 != 0 来判断是否是奇数，或使用   
  
高效式 (i & 1) != 0来判断。   
  
  
2. 小数精确计算

System.out.println(2.00 -1.10);//0.8999999999999999

上面的计算出的结果不是 0.9，而是一连串的小数。问题在于1.1这个数字不能被精确表示为一个double，因此它被表   
  
示为最接近它的double值，该程序从2中减去的就是这个值，但这个计算的结果并不是最接近0.9的double值。   
  
  
一般地说，问题在于并不是所有的小数都可以用二进制浮点数精确表示。   
  
  
二进制浮点对于货币计算是非常不适合的，因为它不可能将1.0表示成10的其他任何负次幂。   
  
  
解决问题的第一种方式是使用货币的最小单位（分）来表示：

System.out.println(200-110);//90

第 二种方式是使用BigDecimal，但一定要用BigDecimal(String)构造器，而千万不要用 BigDecimal(double)来构造（也不能将float或double型转换成String再来使用BigDecimal(String)来构 造，因为在将float或double转换成String时精度已丢失）。   
例如new BigDecimal(0.1)，   
它将返回一个BigDecimal，   
也即0.1000000000000000055511151231257827021181583404541015625，   
正确使用BigDecimal，程序就可以打印出我们所期   
  
望的结果0.9：

System.out.println(new BigDecimal("2.0").subtract(new BigDecimal("1.10")));// 0.9

另外，如果要比较两个浮点数的大小，要使用BigDecimal的compareTo方法。   
  
3. int整数相乘溢出   
  
我们计算一天中的微秒数：

long microsPerDay = 24 \* 60 \* 60 \* 1000 \* 1000;// 正确结果应为：86400000000

System.out.println(microsPerDay);// 实际上为：500654080

  问题在于计算过程中溢出了。这个计算式完全是以int运算来执行的，并且只有在运算完成之后，其结果才被提升为long，而此时已经太迟：计算已经溢出。   
  解决方法使计算表达式的第一个因子明确为long型，这样可以强制表达式中所有的后续计算都用long运算来完成，这样结果就不会溢出：

long microsPerDay = 24L \* 60 \* 60 \* 1000 \* 1000;

4. 负的十六进制与八进制字面常量   
  
“数 字字面常量”的类型都是int型，而不管他们是几进制，所以“2147483648”、“0x180000000（十六进制，共33位，所以超过了整数的 取值范围）”字面常量是错误的，编译时会报超过int的取值范围了，所以要确定以long来表示 “2147483648L”“0x180000000L”。   
  
  
十进制字面常量只有一个特性，即所有的十进制字面常量都是正数，如果想写一个负的十进制，则需要在正的十进制   
  
字面常量前加上“-”即可。   
  
  
十六进制或八进制字面常量可就不一定是正数或负数，是正还是负，则要根据当前情况看：如果十六进制和八进制字   
  
面常量的最高位被设置成了1，那么它们就是负数：

System.out.println(0x80);//128

//0x81看作是int型，最高位(第32位)为0，所以是正数

System.out.println(0x81);//129

System.out.println(0x8001);//32769

System.out.println(0x70000001);//1879048193

//字面量0x80000001为int型，最高位(第32位)为1，所以是负数

System.out.println(0x80000001);//-2147483647

//字面量0x80000001L强制转为long型，最高位（第64位）为0，所以是正数

System.out.println(0x80000001L);//2147483649

//最小int型

System.out.println(0x80000000);//-2147483648

//只要超过32位，就需要在字面常量后加L强转long，否则编译时出错

System.out.println(0x8000000000000000L);//-9223372036854775808

从上面可以看出，十六进制的字面常量表示的是int型，如果超过32位，则需要在后面加“L”，否则编译过不过。如果为32，则为负int正数，超过32位，则为long型，但需明确指定为long。

System.out.println(Long.toHexString(0x100000000L + 0xcafebabe));// cafebabe

结 果为什么不是0x1cafebabe？该程序执行的加法是一个混合类型的计算：左操作数是long型，而右操作数是int类型。为了执行该计算，Java 将int类型的数值用拓宽原生类型转换提升为long类型，然后对两个long类型数值相加。因为int是有符号的整数类型，所以这个转换执行的是符号扩 展。   
  这个加法的右操作数0xcafebabe为32位，将被提升为long类型的数值0xffffffffcafebabeL，之后这个数值加上了左操   
  
作0x100000000L。当视为int类型时，经过符号扩展之后的右操作数的高32位是-1，而左操作数的第32位是1，两个数   
  
值相加得到了0：   
  0x 0xffffffffcafebabeL   
+0x 0000000100000000L   
-----------------------------   
0x 00000000cafebabeL   
  
如果要得到正确的结果0x1cafebabe，则需在第二个操作数组后加上“L”明确看作是正的long型即可，此时相加时拓   
  
展符号位就为0：

System.out.println(Long.toHexString(0x100000000L + 0xcafebabeL));// 1cafebabe

5. 窄数字类型提升至宽类型时使用符号位扩展还是零扩展

System.out.println((int)(char)(byte)-1);// 65535

结果为什么是65535而不是-1？   
  
  
窄的整型转换成较宽的整型时符号扩展规则：如果最初的数值类型是有符号的，那么就执行符号扩展（即如果符号位   
  
为1，则扩展为1，如果为零，则扩展为0）；如果它是char，那么不管它将要被提升成什么类型，都执行零扩展。   
  
  
了解上面的规则后，我们再来看看迷题：因为byte是有符号的类型，所以在将byte数值-1（二进制为：11111111）提   
  
升到char时，会发生符号位扩展，又符号位为1，所以就补8个1，最后为16个1；然后从char到int的提升时，由于是   
  
char型提升到其他类型，所以采用零扩展而不是符号扩展，结果int数值就成了65535。   
  
  
如果将一个char数值c转型为一个宽度更宽的类型时，只是以零来扩展，但如果清晰表达以零扩展的意图，则可以考虑   
  
使用一个位掩码：

int i = c & 0xffff;//实质上等同于：int i = c ;

如果将一个char数值c转型为一个宽度更宽的整型，并且希望有符号扩展，那么就先将char转型为一个short，它与   
  
char上个具有同样的宽度，但是它是有符号的：

int i = (short)c;

如果将一个byte数值b转型为一个char，并且不希望有符号扩展，那么必须使用一个位掩码来限制它：

char c = (char)(b & 0xff);// char c = (char) b;为有符号扩展

[size=medium]   
6. ((byte)0x90 == 0x90)?   
[/size]   
答案是不等的，尽管外表看起来是成立的，但是它却等于false。为了比较byte数值(byte)0x90和int数值0x90，Java   
  
通过拓宽原生类型将byte提升为int，然后比较这两个int数值。因为byte是一个有符号类型，所以这个转换执行的是   
  
符号扩展，将负的byte数值提升为了在数字上相等的int值（10010000◊111111111111111111111111 10010000）。在本例中，该转换将(byte)0x90提升为int数值-112，它不等于int数值的0x90，即+144。   
  
  
解决办法：使用一个屏蔽码来消除符号扩展的影响，从而将byte转型为int。

((byte)0x90 & 0xff)== 0x90

7. 三元表达式（?:）

char x = 'X';

int i = 0;

System.out.println(true ? x : 0);// X

System.out.println(false ? i : x);// 88

条件表达式结果类型的规则：   
（1） 如果第二个和第三个操作数具有相同的类型，那么它就是条件表达式的类型。   
（2） 如果一个操作的类型是T，T表示byte、short或char，而另一个操作数是一个int类型的“字面常量”，并且   
  
它的值可以用类型T表示，那条件表达式的类型就是T。   
（3） 否则，将对操作数类型进行提升，而条件表达式的类型就是第二个和第三个操作被提升之后的类型。   
  
  
现来使用以上规则解上面的迷题，第一个表达式符合第二条规则：一个操作数的类型是char，另一个的类型是字面常   
  
量为0的int型，但0可以表示成char，所以最终返回类型以char类型为准；第二个表达式符合第三条规则：因为i为int   
  
型变量，而x又为char型变量，所以会先将x提升至int型，所以最后的结果类型为int型，但如果将i定义成final时，   
  
则返回结果类型为char，则此时符合第二条规则，因为final类型的变量在编译时就使用“字面常量0”来替换三元表   
  
达式了：

final int i = 0;

System.out.println(false ? i : x);// X

在JDK1.4版本或之前，条件操作符 ?: 中，当第二个和延续三个操作数是引用类型时，条件操作符要求它们其中一个   
  
必须是另一个的子类型，那怕它们有同一个父类也不行：

public class T {

public static void main(String[] args) {

System.out.println(f());

}

public static T f() {

// !!1.4不能编译，但1.5可以

// !!return true?new T1():new T2();

return true ? (T) new T1() : new T2();// T1

}

}

class T1 extends T {

public String toString() {

return "T1";

}

}

class T2 extends T {

public String toString() {

return "T2";

}

}

在5.0或以上版本中，条件操作符在延续二个和第三个操作数是引用类型时总是合法的。其结果类型是这两种类型的最   
  
小公共超类。公共超类总是存在的，因为Object是每一个对象类型的超类型，上面的最小公共超类是T，所以能编译。

**在JAVA程序中，性能问题的大部分原因并不在于JAVA语言，而是程序本身。养成良好的编码习惯非常重要，能够显著地提升程序性能。   
  
1. 尽量使用final修饰符。   
带 有final修饰符的类是不可派生的。在JAVA核心API中，有许多应用final的例子，例如 java.lang.String。为String类指定final防止了使用者覆盖length()方法。另外，如果一个类是final的，则该类所有 方法都是final的。java编译器会寻找机会内联（inline）所有的final方法（这和具体的编译器实现有关）。此举能够使性能平均提高 50%。   
  
2.尽量重用对象。   
特别是String对象的使用中，出现字符串连接情况时应使用StringBuffer代替，由于系统不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理。因此生成过多的对象将会给程序的性能带来很大的影响。   
  
3. 尽量使用局部变量。   
调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快。其他变量，如静态变量，实例变量等，都在堆（Heap）中创建，速度较慢。   
  
4.不要重复初始化变量。   
默 认情况下，调用类的构造函数时，java会把变量初始化成确定的值，所有的对象被设置成null，整数变量设置成0，float和double变量设置成 0.0，逻辑值设置成false。当一个类从另一个类派生时，这一点尤其应该注意，因为用new关键字创建一个对象时，构造函数链中的所有构造函数都会被 自动调用。   
这里有个注意，给成员变量设置初始值但需要调用其他方法的时候，最好放在一个方法比如initXXX()中，因为直接调用某方法赋值可能会因为类尚未初始化而抛空指针异常，public int state = this.getState();   
  
5.在java+Oracle的应用系统开发中，java中内嵌的SQL语言应尽量使用大写形式，以减少Oracle解析器的解析负担。   
  
6.java编程过程中，进行数据库连接，I/O流操作，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销。   
  
7.过分的创建对象会消耗系统的大量内存，严重时，会导致内存泄漏，因此，保证过期的对象的及时回收具有重要意义。   
JVM的GC并非十分智能，因此建议在对象使用完毕后，手动设置成null。   
  
8.在使用同步机制时，应尽量使用方法同步代替代码块同步。   
  
9.尽量减少对变量的重复计算。   
比如**

1. **for(int i=0;i<list.size();i++)**

**应修改为**

1. **for(int i=0,len=list.size();i<len;i++)**

**10. 采用在需要的时候才开始创建的策略。   
例如：**

1. **String str="abc";**
2. **if(i==1){ list.add(str);}**

**应修改为：**

1. **if(i==1){String str="abc"; list.add(str);}**

**11.慎用异常，异常对性能不利。   
抛 出异常首先要创建一个新的对象。Throwable接口的构造函数调用名为fillInStackTrace()的本地方 法，fillInStackTrace()方法检查栈，收集调用跟踪信息。只要有异常被抛出，VM就必须调整调用栈，因为在处理过程中创建了一个新的对 象。   
异常只能用于错误处理，不应该用来控制程序流程。   
  
12.不要在循环中使用Try/Catch语句，应把Try/Catch放在循环最外层。   
Error是获取系统错误的类，或者说是虚拟机错误的类。不是所有的错误Exception都能获取到的，虚拟机报错 Exception就获取不到，必须用Error获取。   
  
13.通过StringBuffer的构造函数来设定他的初始化容量，可以明显提升性能。   
StringBuffer 的默认容量为16，当StringBuffer的容量达到最大容量时，她会将自身容量增加到当前的2倍+2，也就是2\*n+2。无论何时，只要 StringBuffer到达她的最大容量，她就不得不创建一个新的对象数组，然后复制旧的对象数组，这会浪费很多时间。所以给StringBuffer 设置一个合理的初始化容量值，是很有必要的！   
  
14.合理使用java.util.Vector。   
Vector 与StringBuffer类似，每次扩展容量时，所有现有元素都要赋值到新的存储空间中。Vector的默认存储能力为10个元素，扩容加倍。   
vector.add(index,obj) 这个方法可以将元素obj插入到index位置，但index以及之后的元素依次都要向下移动一个位置（将其索引加 1）。 除非必要，否则对性能不利。   
同 样规则适用于remove(int index)方法，移除此向量中指定位置的元素。将所有后续元素左移（将其索引减 1）。返回此向量中移除的元素。所以删除vector最后一个元素要比删除第1个元素开销低很多。删除所有元素最好用 removeAllElements()方法。   
如果要删除vector里的一个元素可以使用 vector.remove(obj)；而不必自己检索元素位置，再删除，如int index = indexOf（obj）;vector.remove(index)；   
  
15.当复制大量数据时，使用 System.arraycopy();   
  
16.代码重构，增加代码的可读性。   
  
17.不用new关键字创建对象的实例。   
用 new关键词创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了Cloneable接口，我们可以调用她的clone() 方法。clone()方法不会调用任何类构造函数。   
下面是Factory模式的一个典型实现。**

1. **public static Credit getNewCredit()**
2. **{**
3. **return new Credit();**
4. **}**

**改进后的代码使用clone() 方法，**

1. **private static Credit BaseCredit = new Credit();**
2. **public static Credit getNewCredit()**
3. **{**
4. **return (Credit)BaseCredit.clone();**
5. **}**

**18. 乘除法如果可以使用位移，应尽量使用位移，但最好加上注释，因为位移操作不直观，难于理解。   
  
19.不要将数组声明为：public static final。   
  
20.HaspMap的遍历。**

1. **Map<String, String[]> paraMap = new HashMap<String, String[]>();**
2. **for( Entry<String, String[]> entry : paraMap.entrySet() )**
3. **{**
4. **String appFieldDefId = entry.getKey();**
5. **String[] values = entry.getValue();**
6. **}**

**利用散列值取出相应的Entry做比较得到结果，取得entry的值之后直接取key和 value。   
  
21.array(数组)和ArrayList的使用。   
array 数组效率最高，但容量固定，无法动态改变，ArrayList容量可以动态增长，但牺牲了效率。   
  
22.单线程应尽量使用 HashMap, ArrayList,除非必要，否则不推荐使用HashTable,Vector，她们使用了同步机制，而降低了性能。   
  
23.StringBuffer,StringBuilder 的区别在于：java.lang.StringBuffer 线程安全的可变字符序列。一个类似于String的字符串缓冲区，但不能修改。StringBuilder与该类相比，通常应该优先使用 StringBuilder类，因为她支持所有相同的操作，但由于她不执行同步，所以速度更快。为了获得更好的性能，在构造StringBuffer或 StringBuilder时应尽量指定她的容量。当然如果不超过16个字符时就不用了。   
相同情况下，使用StringBuilder比使用 StringBuffer仅能获得10%~15%的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用StringBuffer。   
  
24. 尽量使用基本数据类型代替对象。   
  
25.用简单的数值计算代替复杂的函数计算，比如查表方式解决三角函数问题。   
  
26.使用具体类比使用接口效率高，但结构弹性降低了，但现代IDE都可以解决这个问题。   
  
27.考虑使用静态方法，   
如果你没有必要去访问对象的外部，那么就使你的方法成为静态方法。她会被更快地调用，因为她不需要一个虚拟函数导向表。这同事也是一个很好的实践，因为她告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。   
  
28.应尽可能避免使用内在的GET,SET方法。   
android编程中，虚方法的调用会产生很多代价，比实例属性查询的代价还要多。我们应该在外包调用的时候才使用get，set方法，但在内部调用的时候，应该直接调用。   
  
29. 避免枚举，浮点数的使用。   
  
30.二维数组比一维数组占用更多的内存空间，大概是10倍计算。   
  
31.SQLite数据库读取整张表的全部数据很快，但有条件的查询就要耗时30-50MS,大家做这方面的时候要注意，尽量少用，尤其是嵌套查找！**

56.comparable和comparator的区别：

1. Comparator 和 Comparable 相同的地方  
  
他们都是java的一个接口, 并且是用来对自定义的class比较大小的,  
  
什么是自定义class: 如 public class Person{ String name; int age }.  
  
当我们有这么一个personList,里面包含了person1, person2, persion3....., 我们用Collections.sort( personList ),   
是得不到预期的结果的. 这时肯定有人要问, 那为什么可以排序一个字符串list呢:  
  
如 StringList{"hello1" , "hello3" , "hello2"}, Collections.sort( stringList ) 能够得到正确的排序, 那是因为   
String 这个对象已经帮我们实现了 Comparable接口 , 所以我们的 Person 如果想排序, 也要实现一个比较器。  
   
2. Comparator 和 Comparable 的区别  
  
Comparable  
  
Comparable 定义在 Person类的内部:  
  
public class Persion implements Comparable {..比较Person的大小..},  
  
 因为已经实现了比较器,那么我们的Person现在是一个可以比较大小的对象了,它的比较功能和String完全一样,可以随时随地的拿来  
比较大小,因为Person现在自身就是有大小之分的。Collections.sort(personList)可以得到正确的结果。  
  
Comparator  
  
Comparator 是定义在Person的外部的, 此时我们的Person类的结构不需要有任何变化,如  
  
public class Person{ String name; int age },  
  
然后我们另外定义一个比较器:  
  
public PersonComparator implements Comparator() {..比较Person的大小..},  
  
在PersonComparator里面实现了怎么比较两个Person的大小. 所以,用这种方法,当我们要对一个 personList进行排序的时候,   
我们除了了要传递personList过去, 还需要把PersonComparator传递过去,因为怎么比较Person的大小是在PersonComparator  
里面实现的, 如:  
  
Collections.sort( personList , new PersonComparator() ).  
  
3. Comparator 和 Comparable 的实例  
  
Comparable:  
  
实现Comparable接口要覆盖compareTo方法, 在compareTo方法里面实现比较：  
public class Person implements Comparable {  
 String name;  
 int age  
 public int compareTo(Person another) {  
 int i = 0;  
 i = name.compareTo(another.name); // 使用字符串的比较  
 if(i == 0) { // 如果名字一样,比较年龄, 返回比较年龄结果  
 return age - another.age;  
 } else {  
 return i; // 名字不一样, 返回比较名字的结果.  
 }  
 }  
}  
 这时我们可以直接用 Collections.sort( personList ) 对其排序了.  
  
Comparator:  
  
实现Comparator需要覆盖 compare 方法：  
public class Person{  
 String name;  
 int age  
}  
  
class PersonComparator implements Comparator {   
 public int compare(Person one, Person another) {  
 int i = 0;  
 i = one.name.compareTo(another.name); // 使用字符串的比较  
 if(i == 0) { // 如果名字一样,比较年龄,返回比较年龄结果  
 return one.age - another.age;  
 } else {  
 return i; // 名字不一样, 返回比较名字的结果.  
 }  
 }  
}  
 Collections.sort( personList , new PersonComparator()) 可以对其排序  
  
 4:总结  
  
两种方法各有优劣, 用Comparable 简单, 只要实现Comparable 接口的对象直接就成为一个可以比较的对象,  
但是需要修改源代码, 用Comparator 的好处是不需要修改源代码, 而是另外实现一个比较器, 当某个自定义  
的对象需要作比较的时候,把比较器和对象一起传递过去就可以比大小了, 并且在Comparator 里面用户可以自  
己实现复杂的可以通用的逻辑,使其可以匹配一些比较简单的对象,那样就可以节省很多重复劳动了。

62.线程池的冲断策略：四种：中止，调用省运行，容器策略，遗弃策略

·AbortPolicy（中止）：它是默认的策略。  
·CallerRunsPolicy (调用者运行)：它既不会丢弃任务，也不会抛出任何异常，它会把任务推回到调用者那里去,以此缓解任务流  
·DiscardPolicy（遗弃）策略：它默认会放弃这个任务

·DiscardOldestPolicy（遗弃最旧的）：它选择的丢弃的任务，是它本来要执行的（可怜的娃，就这样被新加入的给排挤了）