没有第三个变量的情况下交换两个数：a=a+b，b=a-b，a=a-b。

生产者消费者模式：（1）用 synchronized 关键字和 wait()和 notifyAll()实现。（2）用 ReentrantLock 锁和 Condition 变量实现。（3）使用阻塞队列来实现。

（1）

```
public class Test{
    private static Integer count=0;
    private final Integer FULL=5;
    private static String lock="lock";
    public static void main(String []args) {
        Test test=new Test();
        new Thread(test.new Producer()).start();
        new Thread(test.new Consumer()).start();
        new Thread(test.new Producer()).start();
        new Thread(test.new Consumer()).start();
    }
    class Producer implements Runnable{
        @Override
        public void run(){
            for(int i=0;i<5;i++) {
                try{
                    Thread.sleep(1000);
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
                synchronized(lock){
                    while(count==FULL){
                        try{
                            lock.wait();
                        }catch(InterruptedException e){
                            e.printStackTrace();
                        }
                    }
                    count++;
                    System.out.println("生产者："+
                    Thread.currentThread().getName()+
```

```
                    "已经生产，商品数量为："+count);
                    lock.notifyAll();
                }
            }
        }
    }
    class Consumer implements Runnable{
        @Override
        public void run(){
            for(int i=0;i<5;i++){
                try{
                    Thread.sleep(1000);
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
                synchronized(lock){
                    while(count==0){
                        try{
                            lock.wait();
                        }catch(InterruptedException e){
                            e.printStackTrace();
                        }
                    }
                    count--;
                    System.out.println("消费者："+
                    Thread.currentThread().getName()+
                    "已经消费，剩余产品数量为："+count);
                    lock.notify();
                }
            }
        }
    }
}
```

（2）

```
public class Test{
```

```
private static Integer count=0;
private final Integer FULL=5;
final Lock lock=new ReentrantLock();
final Condition put=lock.newCondition();
final Condition get=lock.newCondition();
public static void main(String []args) {
    Test test=new Test();
    new Thread(test.new Producer()).start();
    new Thread(test.new Consumer()).start();
    new Thread(test.new Producer()).start();
    new Thread(test.new Consumer()).start();
}
class Producer implements Runnable{
    @Override
    public void run(){
        for(int i=0;i<5;i++) {
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
            lock.lock();
            try{
                while(count==FULL){
                    try{
                        put.await();
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                }
                count++;
                System.out.println("生产者："+
                Thread.currentThread().getName()+
                "已经生产，商品数量为："+count);
```

```
                get.signal();
            }finally{
                lock.unlock();
            }
        }
    }
}
class Consumer implements Runnable{
    @Override
    public void run(){
        for(int i=0;i<5;i++){
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
            lock.lock();
            try{
                while(count==0){
                    try{
                        lock.wait();
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                }
                count--;
                System.out.println("消费者："+
                Thread.currentThread().getName()+
                "已经消费，剩余产品数量为："+count);
                put.signal();
            }finally{
                lock.unlock();
            }
        }
```

```
            }
        }
    }
（3）public class Test{
        private static Integer count=0;
        final BlockingQueue<Integer> blockingQueue=new
        ArrayBlockingQueue<Integer>(5);
        public static void main(String []args) {
            Test test=new Test();
            new Thread(test.new Producer()).start();
            new Thread(test.new Consumer()).start();
            new Thread(test.new Producer()).start();
            new Thread(test.new Consumer()).start();
        }
        class Producer implements Runnable{
            @Override
            public void run(){
                for(int i=0;i<5;i++) {
                    try{
                        Thread.sleep(1000);
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                    try{
                        blockingQueue.put(1);
                        count++;
                        System.out.println("生产者："+
                        Thread.currentThread().getName()+
                        "已经生产，商品数量为："+count);
                    }catch(InterruptedException e){
                        e.printStackTrace();
                    }
                }
            }
```

```
    }
    class Consumer implements Runnable{
        @Override
        public void run(){
            for(int i=0;i<5;i++){
                try{
                    Thread.sleep(1000);
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
                try{
                    blockingQueue.take();
                    count--;
                    System.out.println("消费者："+
                    Thread.currentThread().getName()+
                    "已经消费，剩余产品数量为："+count);
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
            }
        }
    }
}
```

foreach 在使用的过程中，会创建额外的 iterator 方法，每次调用那个 hasNext()和 next()方法会增加很多步骤，对于访问数组 for 的效率更好，foreach 是线程安全的，for 不是线程安全的。在多线程中使用 foreach，在方法中读取局部变量的操作使用 for。

虚拟内存就是将硬盘中划分出来一部分区域来当内存使用，是将部分内存持久化到硬盘，需要使用的时候从内存中读取，并将暂时不需要的内存信息存储到硬盘中。请求分页系统、请求分段系统、请求段页式系统就是针对虚拟内存，通过请求实现内存与外存的信息置换。

置换算法：（1）FIFO 先进先出算法，实现简单，比较常用，作业调度。（2）LRU 最近最少使用算法，根据使用时间到现在的长短进行判断。（3）LFU 最少使用次数算法，根据使用次数进行判断。（4）OPT 最优置换算法，保证置换出去的是不再被使用的页，或者在实际内存中最晚使用的算法。

虚拟地址，程序自身定义的地址，需要转换成物理地址才能使用。逻辑地址，段内偏移

地址，也是虚拟地址。物理地址，内存中的实际地址。线性地址，CPU 可寻址的地址，段内偏移地址+段基址。

进程通信方式：（1）共享内存，共享同一内存空间。（2）消息队列，一个进程将消息发送到一个公共的消息队列缓存区中，另一个进程去公共消息队列缓存区读取是否存在自己的消息。缺点是只能实现 P2P 通信，优点是速度快。（3）管道，借助一个中间临时文件，一个进程向该文件中写数据，另一个进程从该文件中获取数据。先进先出，而且数据具有不可再现性。（4）Socket，主要用于远程通信。

Windows 内存管理方式主要分为：（1）页式管理，基本原理是将各个进程的虚拟空间划分为若干个长度相等的页。页式管理把内存空间按照页的大小划分为片或者页面，然后把页式虚拟地址与内存地址建立对应的页表。优点是没有外碎片，内碎片不超过页的大小。缺点是程序全部装入内存，需要硬件支持。（2）段式管理，基本思想是把程序按照内容或者过程函数关系分段，每段有自己的名字。段式管理程序以段为单位分配内存，然后通过地址映射把段式虚拟地址转换为实际内存物理地址。优点是可以分别编写和编译，可以针对不同类型的段采用不同的保护，可以按段为单位进行共享，通过动态链接进行代码共享，缺点是会产生碎片。（3）段页式管理，具有前两者的优点，负责性和开销增加，需要的硬件以及占用内存增加。

泛型增加了代码的安全性，编译时检查类型安全。增加了代码的复用性，解决了方法重载问题。List<String>不能转为 List<Object>，因为泛型没有继承性，只是提供了限制功能。public void write(Integer i, Integer []ia);public void write(Double d, Double []da)的泛型版本 public<T> void wrote(T t, T [] ta)。

在 Java1.5 以前，如果一个方法中的参数要支持任意类型，则一般用 Object 类作为参数的类型，在运行时需要进行类型转换。自动类型转换容易出现当编译的时候不会报错，在运行的时候报错。泛型的好处在编译的时候检查类型安全，保证类型转换是可以正确转换的。Object 在编译时不会检查父类转子类，泛型中所有的强制转换都是自动和隐式的，提高代码的重用率。泛型信息在编译的时候会被自动擦除，在字节码中没有泛型的信息，并使用限定类型替换，无限定类型替换使用 Object。

Student s = new Student()在内存中做了什么事情。（1）加载 Student.class 文件到内存（2）在栈内存为 s 开辟空间（3）在堆内存为学生对象开辟空间（4）对学生对象的成员变量初始化（5）通过构造方法对学生对象的成员变量赋值（6）学生对象初始化完毕，把对象地址赋值给 s 变量。

脏读：事务 T1 更新了一行记录的内容，但是并没有提交所做的修改。事务 T2 读取更新后的行，然后 T1 执行了回滚，取消了刚才所做的修改，T2 读取的行无效了。即一个事务读取了另一个事务未提交的数据。不可重复读：事务 T1 读取了一行记录，然后 T2 修改了 T1 刚才读取的那行记录，然后 T1 又读取了那行记录，发现与刚才读取的结果不同。幻读：

事务 T1 读取一条指定的 where 子句所返回的结果集，然后 T2 事务新插入一行记录，恰好可以满足 T1 所使用的查询条件。然后 T1 再次对其进行检索，又看到了 T2 新插入的数据。

数据库中的分页查询语句：select * from table limit [offset,] rows | rows offset。Limit 接受一个或者两个数字参数。参数必须是一个整数常量，如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的最大数目。初始记录行的偏移量是 0.。最基本的分页方式：Select … from … where … order by … limit。

为经常出现在关键字 order by、group by、distinct 后面的字段建立索引。在 union 等集合操作的结果集字段上建立索引。为经常用作查询选择的字段建立索引。在经常用于表连接的属性上建立索引。考虑使用索引覆盖，对于数据很少被更新的表，如果用户经常只是查询其中的几个字段，可以考虑在这几个字段上建立索引，从而将表的扫描变成索引的扫描。

索引优点：创建唯一性索引，保证数据库表中的每一行数据的唯一性。加快了数据的检索速度，创建索引最主要的原因。加速数据库表之间的连接，特别是在实现数据的参考完整性方面。在使用分组和排序子句进行数据检索时，同样可以减少查询中分组和排序的时间。通过使用索引，可以在查询中使用优化隐藏器，提高系统的性能。

索引缺点：时间上创建和维护索引都需要花费时间，空间上索引需要占用物理空间，性能上增删改速度减慢，需要维护索引的增删改。

Redis 是一个速度十分快的非关系型数据库，可以存储 key 和 5 种不同类型的 value 之间的映射，可以将存储在内存中的键值对持久化到硬盘中。与 Memcached 相比，两者都可以用来存储键值映射，性能差不多，但是 Redis 可以自动以两种不同的方式将数据写入硬盘。Redis 除了能存储普通的字符串键之外，还可以存储其他四种数据结构。Memcached 只能存储字符串键。Redis 能用作主数据库，也能作为其他存储系统的辅助数据库。

Redis 的数据结构：STRING，可以是字符串、整数或者浮点数。LIST，一个链表，链表上的每个节点包含一个字符串。SET，包含字符串的无序收集器，并且被包含的每一个字符都是不一样的。HASH，包含键值对的无序散列表。ZSET，字符串成员与浮点数分值之间的有序映射，元素的排列顺序由分值的大小决定。

事务隔离级别：（1）串行化，所有事务一个接着一个的执行，可以避免幻读，对于基于锁来实现并发控制的数据库来说，串行化要求在执行范围查询的时候，需要获取范围锁，如果不是基于锁实现并发控制的数据库，则检查到有违反串行操作的事务，需要回滚事务。（2）可重复读，所有被 select 获取的数据都不能被修改，这样可以避免一个事务前后读取不一致的情况。但是没有办法控制幻读，因为不能更改选择的数据，但是可以增加数据。（3）读已提交，被读取的数据可以被其他事务修改，可能导致不可重重复读。事务读取的时候获取读锁，读完之后立即释放，不需要等待事务结束，而写锁的则是事务提交之后才释放，释放读锁之后可以被其他事务修改数据，默认隔离级别。（4）读未提交，最低的隔离级别，允许其他事务看到没有提交的数据，会导致脏读。

数据库四大特性：（1）原子性，保证事务中的所有操作全部执行或者全部不执行。（2）一致性，保证数据库始终保持数据的一致性，事务操作之前和之后是一致的。（3）隔离性，多个事务并发执行，结果应该与多个事务串行执行的效果一致。（4）持久性，事务操作完成后，对数据库的影响是持久的，即使数据库因故障而破坏，数据库也能恢复日志。

String、StringBuffer、StringBuilder。都是 final 类，不允许被继承。String 长度不可变，StringBuilder 和 StringBuffer 长度时可变的。StringBuffer 是线程安全的，StringBuilder 不是线程安全的，StringBuilder 比 StringBuffer 有更好的性能。如果一个 String 类型的字符串，在编译时可以确定是一个字符串常量，则编译完成后，字符串会自动拼接成一个常量。此时，String 的速度比 StringBuffer 和 StringBuilder 性能好。String 类是被 final 修饰的，不能被继承，在用+号连接字符串时会创建新的字符串。

String s = new String(“Hello world”)可能会创建一个对象，也可能会创建两个对象。如果静态区中有“Hello world”字符串常量对象，则仅在堆中创建一个对象。如果静态区中没有“Hello world”对象，则在堆上和静态区中都需要创建一个对象。

String 重写了 Object 类的 hashCode 和 toString 方法。当 equals 方法被重写时，经常需要重写 hashCode 方法来维护 hashCode 的常规协定，声明了相对等的两个对象必须有相同的 hashCode。如果 object1.equals(object2)为 true，object1.hashCode() == object2.hashCode()为 true。如果 object1.hashCode() == object2.hashCode()为 false ，object1.equals(object2)为 false。但是如果 object1.hashCode() == object2.hashCode()为 true ，object1.equals(object2)不一定为 true。

如果在存储散列结合时，原对象.equals(新对象)，没有重写 hashCode，即两个对象拥有不同的 hashCode，则在集合中将会存储两个值相同的对象。因此在重写 equals 方法时，必须重写 hashCode 方法。

Java 序列化，将那些实现了 Serializable 接口的对象转换为一个字节序列，并能在之后将这个对象完全恢复为原来的对象，序列化可以弥补不同操作系统之间的差异。Java 序列化的作用：（1）RMI 远程方法调用。（2）对 JavaBeans 进行序列化。实现序列化的方法：（1）实现 Serializable 接口，该接口只是一个可序列化的标志，并没有包含实际的属性和方法。为了保证安全性，可以使用 transient 关键字修饰不必序列化的属性，在反序列化时，private 修饰的属性能被查看。（2）实现 ExternalSerializable 方法，自己对要序列化的内容进行控制，哪些属性能够被序列化，哪些属性不能被序列化。

Java 反序列化，实现了 Serializable 接口的对象在序列化时不需要调用对象所在类的构造方法，完全基于字节。实现了 ExternalSerializable 接口的方法，在序列化时会调用构造方法。被 static 修饰的属性不会被序列化，对象的类名、属性会被序列化，方法不会被序列化。要保证序列化对象所在类的属性也能够被序列化。当通过网络、文件进行序列化时，必须按照写入的顺序读取对象。反序列化时必须有序列化对象的 class 文件。最好显示声明 serializableID，不同 JVM 之间，默认生成 serializableID 可能不同，会造成反序列化失败。

Java 实现多线程方式：（1）继承 Thread 类，重写 run 函数。（2）实现 Runnable 接口。（3）实现 Callable 接口。实现 Runnable 接口可以避免 Java 单继承特性的局限，增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的，适合多个相同程序代码的线程区处理同一资源。继承 Thread 类和实现 Runnable 接口启动线程都是使用 start 方法，然后 JVM 将线程放到就绪队列中，有处理器可用则执行 run 方法。（3）实现 Callable 接口要实现 call 方法，并且线程执行完毕后会有返回值，其他两种没有返回值。

线程池就是事先创建若干个可执行的线程放入一个容器中，需要的时候从容器中获取线程不需要自行创建，使用完毕不用销毁，直接放入容器中，从而减少线程对象创建和销毁的开销。

一个线程池有（1）线程管理器，用来创建并管理线程池，包括创建线程、销毁线程、添加新任务。（2）工作线程，线程池中有线程，在没有任务时处于等待状态，可以循环执行任务。（3）任务接口，每个任务必须实现的接口，工作线程调度任务的执行，主要规定的任务的入口，任务执行完毕后的收尾及任务执行状态。（4）任务队列，用于存放没有处理的任务，提供一种缓冲机制。

synchronized 是对类的当前实例进行加锁，防止其他线程同时访问该类的该实例的所有 synchronized 块，同一个类的两个不同实例没有约束。static synchronized 是限制线程同时访问 JVM 中该类所有实例同时访问对应的代码块。

二分查找算法，在数组中查找指定元素。

```
public class BinarySearch{
    public int getPosition(int []arr,int n,int val){
        int res=-1;
        int position=0;
        int left=0;
        int right=n;
        position=(left+right)/2;
        while(position>=left&&position<=right){
            if(arr[position]>val){
                right=position;
                position=(left+right)/2;
            } else if(arr[position]<val){
                left=position;
                position=(left+right)/2;
            }else{
                res=position;
```

```
                    for(int i=0;i<=position;i++){
                        if(arr[position-i]==val){
                            res=position-i;
                        }else{
                            return res;
                        }
                    }
                    return res;
                }
            }
            return res;
        }
    }
```

常用 hash 算法：（1）加法 hash，把输入元素一个个加起来构成最后的结果。（2）位运算 hash，利用各种位运算混合输入元素。（3）乘法 hash，利用了乘法的不相关性，32 位 FNV 算法。（4）除法 hash。（5）查表 hash，CRC 系列算法。（6）混合 hash。

Minor GC 发生，当 JVM 无法为新的对象分配内存空间的时候发生，所以分配对象的频率越高，发生的几率越大。Full GC 发生，当老年代无法分配内存的时候发生，当发生 Minor GC 时会触发 Full GC，老年代对年轻代的保护作用。

类加载的五个过程：（1）加载，当遇到 new 关键字或者 static 关键字的时候就会发生或者当使用反射方法进行动态加载时。（2）验证，目的是确保 class 文件的字节流中包含的信息符合当前虚拟机的要求，并不会危害到虚拟机自身的安全。（3）准备，为对象分配内存空间，然后初始化类中的属性变量，该阶段的初始化只是按照系统初始化，初始化为 0 或者 null。（4）解析，虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就是 class 文件常量池中的各种引用，没有在内存中分配空间，仅仅是一个标识，直接引用直接指向了内存中的地址。（5）初始化，执行对象的构造函数，给类的静态字段按照程序进行初始化。初始化由两个函数完成，<clinit>初始化所有的类变量，该函数不会初始哈父类变量，<init>对类中的实例变量进行初始化，要初始化父类的变量。

静态分派和动态分派都是多态的内容，多态的实现依赖于编译阶段和运行阶段，在编译阶段主要表现在静态分派。静态分派是通过静态类型和方法参数个数来选择哪一个版本，体现在方法的重载，在编译的时候就能确定调用哪一个函数，所以叫静态分派。在运行时阶段体现在动态分派，当一个父类引用指向子类对象，通过该父类引用去调用一个该方法。由于要去常量池中搜索每一类的方法名和描述符，效率低。在方法区为每一类维护一张虚方法表或者接口方法表，虚表中存放了该方法的实际入口地址，因此要查找方法名直接去搜索该方

法名对应的直接地址然后执行。

电脑上访问网页的全过程：（1）浏览器输入网址后，浏览器会尝试先从 host 文件中获取该域名对应的 IP 地址。（2）如果取不到 IP 地址，会使用 DNS 协议获取 IP。（3）得到 IP 之后，使用 TCP 协议进行三次握手，中间还需要 IP 协议、ARP 协议、OSPF 协议。（4）使用 HTTP 协议请求网页内容。OSPF 是开放式最短路径优先，是一种路由选择协议，是链路状态协议，使用 Dijkstra 算法。RIP 是路由信息协议，基于距离矢量算法。

PING 的整个过程：

第一种情况，同一网段内。主机 A 想去 PING 主机 B：（1）主机 A 封装二层报文，先在自己的 MAC 地址表中找主机 B 的地址，如果找不到，就发送一个 ARP 广播。（2）交换机收到这个报文后，会检查自己有没有保存主机 B 的 MAC 地址，如果有就直接返回主机 A。（3）如果没有，向所有的端口发送 ARP 广播，其他主机收到后发现不是找自己就丢弃该报文。主机 B 收到报文后就立即响应。同时得到主机 A 的 MAC 地址，并按同样的报文格式返回给主机 A。（4）主机 A 知道了主机 B 的 MAC 地址，就把 MAC 地址封装到 ICMP 协议的二层报文中向主机 B 发送。（5）主机 B 收到这个报文后，发现是主机 A 的 ICMP 回显请求，就按照同样的格式，返回一个值给主机 A。

第二种情况，不同网段内。如果主机 A 想要 PING 主机 C，主机 A 发现主机 C 的 IP 和自己不是同一个网段，就去找网关转发，如果主机 A 不知道网关的 MAC，就会发送一个 ARP 广播，学习到网关的 MAC 地址，然后发 ICMP 报文给网关路由器。

MAC 地址，介质访问控制，物理地址、硬件地址、在 OSI 的第二层。因此一个主机会有一个 MAC 地址，每个网络位置会有一个专属的 IP 地址。ARP，地址解析协议。根据 IP 地址获取物理地址的一个 TCP/IP 协议。主机发送消息时将包含目的主机 IP 地址的 ARP 请求广播到网络上的所有主机，并接收返回消息，来确定目标的物理地址。ICMP，因特网控制报文协议，用于传输出错报文控制信息，对于用户数据的传递起着十分重要的作用。

IP 地址分类由第一个八位组的值来确定。（1）任何一个 0 到 127 间的网络地址是一个 A 类地址。（2）任何一个 128 到 191 间的网络地址是一个 B 类地址。（3）任何一个 192 到 223 间的网络地址是一个 C 类地址。（4）任何一个 224 到 239 件的网络地址是一个 D 类地址，即组播地址。（5）E 类是保留地址。

路由器用于不同网络间数据的跨网络传输，交换机用于同一网络内部数据传输。路由器工作在网络层，交换机工作在数据链路层。

TCP/IP 架构：（1）应用层，文件传输，电子邮件，文件服务。HTTP、FTP、SMTP、DNS、Telnet。（2）传输层，提供端到端的接口，TCP、UDP。（3）网络层，为数据包选择路由，IP、ICMP、IGMP、RIP、OSPF、BGP。（4）数据链路层，传输有地址的帧和错误检测功能，ARP、RARP、PPP、MTU。（5）物理层，二进制数据格式在物理媒体上传输数据，ISO2110、IEEE802。

OSI 架构：（1）应用层，文件传输，电子邮件，文件服务。HTTP、FTP、SMTP、DNS、

Telnet。（2）表示层，数据格式化，代码转换，数据加密，没有协议。（3）会话层，解除或者建立与别的层的联系。（4）传输层，提供端到端的接口，TCP、UDP。（5）网络层，为数据包选择路由，IP、ICMP、IGMP、RIP、OSPF、BGP。（6）数据链路层，传输有地址的帧和错误检测功能，ARP、RARP、PPP、MTU。（7）物理层，二进制数据格式在物理媒体上传输数据，ISO2110、IEEE802。

HTTP 协议的请求类型：（1）GET，向指定资源发出请求。（2）POST，向指定资源提交数据进行请求处理。（3）PUT，向指定资源位置上传内容。（4）DELETE，请求服务器删除资源。（5）MOVE，请求服务器将指定的页面移动到另一个网络地址。