Java 面试复习大纲

一、java 语言基础

1、java 编程规则

（1）避免使用 NEW 要害字来创建 String 对象。

把一个 String 常量 copy 到 String 对象中通常是多余、浪费时间的

（2）避免使用不必要的嵌套。

过多的嵌套会使你的代码复杂化,减弱可读性。

（3）避免在同一行声明不同类型的多个变量

这样可以使程序更加清楚,避免混乱

（4）在每一行里写一条语句

这条规则不包括 for 语句:比如:'for (int i = 0; i < 10; i++) x--;'可以增加代码的可读性。

（5）经常从 finalize ()中调用 super.finalize ()

这里的 finalize ()是 java 在进行垃圾收集的时候调用的,和 finally 不一样。假如你的父类没有定义 finally()的话,你也应该调用。这里有两个原因:(1)在不改变代码的情况下能够将父类的 finally 方法加到你的类中。 (2)以后你会养成习惯调用父类的 finally 方法,即使父类没有定义 finally 方法的时候。

正确的方法应该如此:

```
public class parentFinalize {
    protected void finalize () throws Throwable {
        super.finalize(); // FIXED
    }
}
```

（6）不要在 finalize ()中注销 listeners

不要再 finalize ()方法中中注销 listeners,finalize ()只有再没有对象引用的时候调用,假如 listeners 从 finalize()方法中去除了,被 finalize 的对象将不会在垃圾收集中去除。

```
public void finalize () throws Throwable {
    bButton.removeActionListener (act);
}
```

（7）不要显式的调用 finalize ()方法

虽然显式的调用这个方法可以使你确保你的调用,但是当这个方法收集了以后垃圾收集会再收集一次。

```
public class T7 {
    public void finalize() throws Throwable {
        close_resources ();
        super.finalize ();
     }
    public void close_resources() {}
```

```
}
class Test {
    void cleanup () throws Throwable {
        t71.finalize(); //  调用
        t71 = null;
    }
    private t71 = new T7 ();
}
```

对于这样的调用我们应该自己创建一个释放的方法,做最初 finalize ()所作的事情,当你每次想显式的调用 finalize ()的时候实际上调用了释放方法。然后再使用一个判定字段来确保这个方法只执行一次,以后再调用就没关系了。

```
public class T7 {
    public synchronized void release () throws Throwable{
        if (!_released) {
            close_resources ();
            // do what the old 'finalize ()' did
            _released = true;
        }
    }
    public void finalize () throws Throwable {
        release ();
        super.finalize ();
    }
    public void close_resources() {}
    private boolean _released = false;
}
class TestFixed {
    void closeTest () throws Throwable {
        t71 .release (); // FIXED
        t71 = null;
    }
    private T7 t71 = new T7 ();
}
```

（8）不要使用不推荐的 API

尽量使用 JDK1.3 推荐的 API。在类和方法或者 java 组件里有很多方法是陈旧的或者是可以选择的。有一些方法 SUN 用了"deprecated“标记。最好不要使用例如:

```
private List t_list = new List ();
t_list.addItem (str);
```

（9）为所有序列化的类创建一个'serialVersionUID'

可以避免从你各种不同的类破坏序列的兼容性。假如你不非凡制订一个 UID 的话,那么系统为自动产生一个 UID(根据类的内容)。假如 UID 在你新版本的类中改变了,即使那个被序列化的类没改变,你也不能反序列化老的版本了。

public class DUID implements java.io.Serializable { public void method () {}}

在里面加一个 UID,当这个类的序列化形式改变的时候,你也改变这个 UID 就可以了。

```
public class DUIDFixed implements java.io.Serializable {
    public void method () {}
    private static final long serialVersionUID = 1;
}
```

（10）对于 private 常量的定义

比较好的做法是对于这样的常量,加上 final 标记,这样的常量从初始化到最后结束值都不会改变。

private int size = 5;

改变后的做法是:

private final int size = 5;

（11）避免把方法本地变量和参数定义成和类变量相同的名字。

这样轻易引起混扰,建议把任何的变量字都定义成唯一的。这样看来,SCJP 里的那些题目在现实中就用不到了:)

public void method (int j) { final int i = 5; // VIOLATION } private int j = 2;

建议:

public void method (int j1) { final int i = 5; // VIOLATION } private int j = 2;

2、你是怎样理解面向对象的

面向对象是利于语言对现实事物进行抽象。面向对象具有以下四大特征：

（1）继承：继承是从已有类得到继承信息创建新类的过程

（2）封装：

通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。

（3）多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。

（4）抽象：

抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。

3、char 型变量能不能储存一个中文汉字，为什么

char 是按照字符存储的，不管英文还是中文，固定占用占用 2 个字节，用来储存 Unicode 字符。范围在 0-65535。

unicode 编码字符集中包含了汉字，所以，char 型变量中当然可以存储汉字啦。不过，如果某个特殊的汉字没有被包含在 unicode 编码字符集中，那么，这个 char 型变量中就不能存储这个特殊汉字。

4、Int 和 Integer 有什么区别，以及以下程序结果

```
public class Test03 {

    public static void main(String[] args) {
        Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;

        System.out.println(f1 == f2);
        System.out.println(f3 == f4);
    }
}
```

5、==和 equals 区别

（1）==既可以比较基本类型也可以比较引用类型，对于基本类型就是比较值，对与引用类型就是比较内存地址
（2）Equals 的话，它属于 java.lang.Object 类中的方法，如果该方法没有被重写过，默认也是==，我们可以看到 String 类的 equals 方法是被重写过的，而且 String 类在日常开发中用的比较多，久而久之，形成 equals 是比较值的错误观点
（3）具体要看这有没有重写 Object 的 hashCode 方法和 equals 方法来判断

6、谈谈你对反射的理解

反射机制:

所谓的反射机制就是 java 语言在运行时拥有一项自观的能力。通过这种能力可以彻底的了解自身的情况为下一步的动作做准备。

Java 的反射机制的实现要借助于 4 个类：class，Constructor，Field，Method;

其中 class 代表的时类对 象，Constructor－类的构造器对象，Field－类的属性对象，Method－类的方法对象。通过这四个对象我们可以粗略的看到一个类的各个组 成部分。

Java 反射的作用：

在 Java 运行时环境中，对于任意一个类，可以知道这个类有哪些属性和方法。对于任意一个对象，可以调用它的任意一个方法。这种动态获取类的信息以及动态调用对象的方法的功能来自于 Java 语言的反射（Reflection）机制。

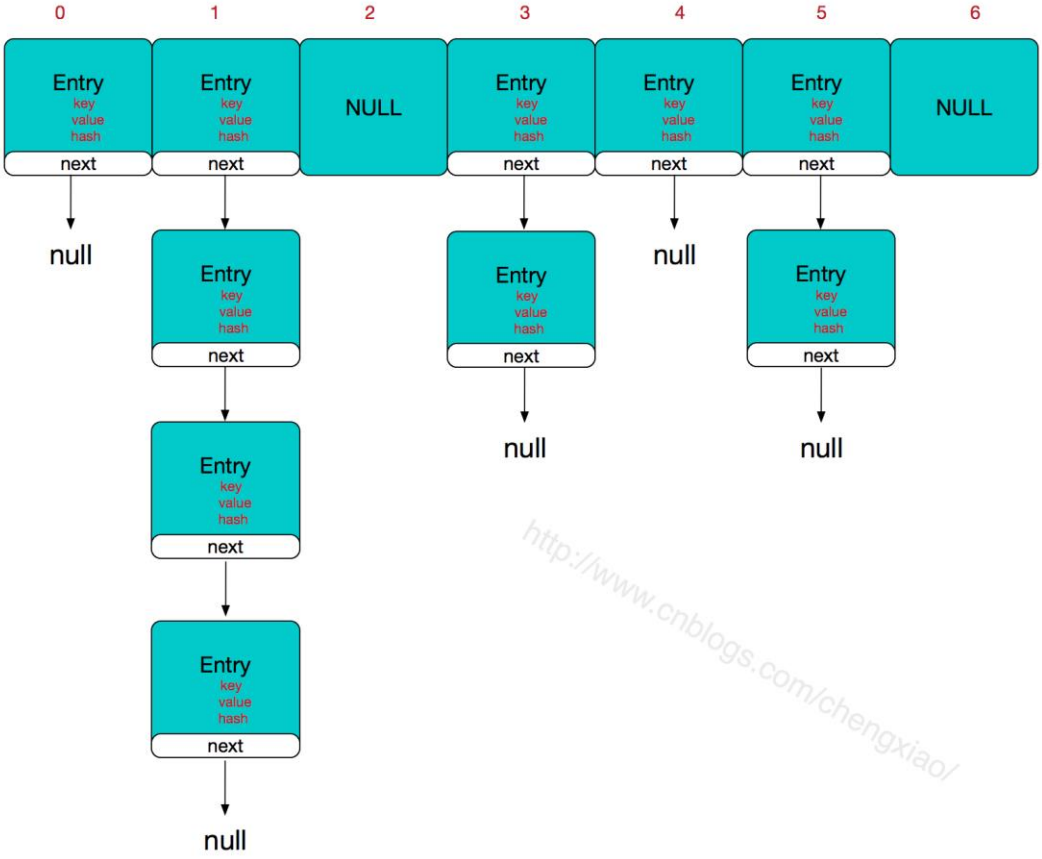
Java 反射机制提供功能

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

7、Arrarlist 和 Linkedlist 区别

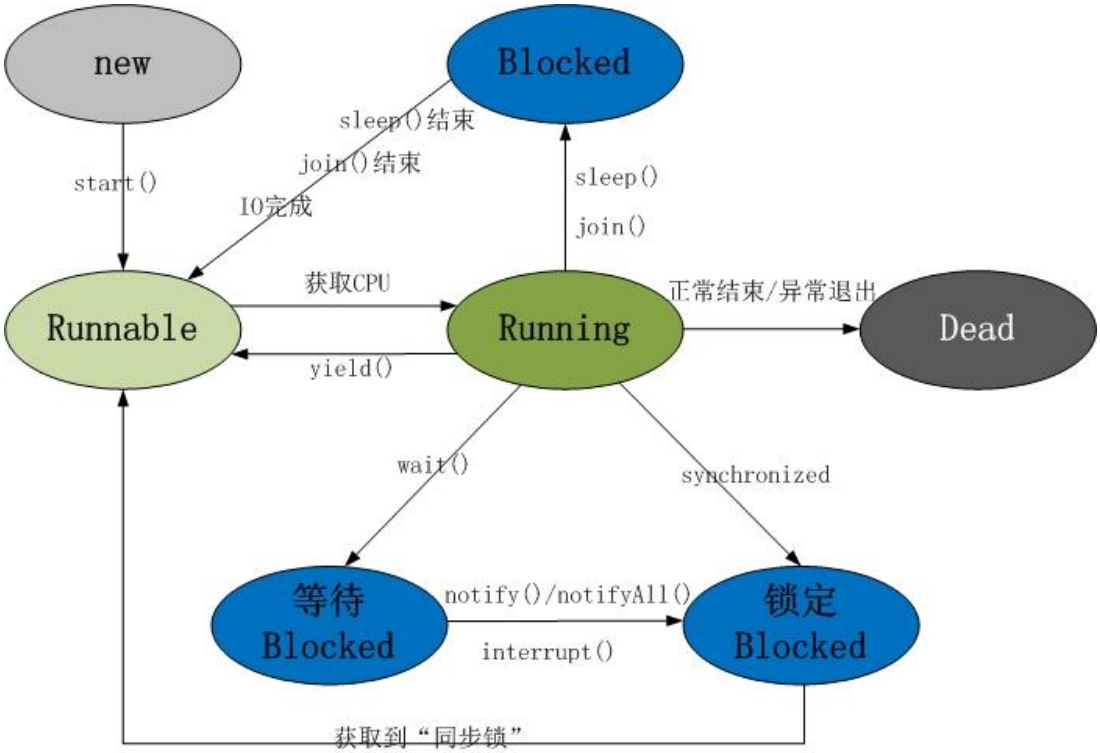
（1）ArrayList 是实现了基于动态数组的数据结构，LinkedList 基于链表的数据结构。
（2）对于随机访问 get 和 set，ArrayList 觉得优于 LinkedList，因为 LinkedList 要移动指针。
（3）对于新增和删除操作 add 和 remove，LinedList 比较占优势，因为 ArrayList 要移动数据。这一点要看实际情况的。若只对单条数据插入或删除，ArrayList 的速度反而优于 LinkedList。但若是批量随机的插入删除数据，LinkedList 的速度大大优于 ArrayList. 因为 ArrayList 每插入一条数据，要移动插入点及之后的所有数据。

8、HashMap 底层原理，HashSet 原理



HashMap 由数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前 entry 的 next 指向 null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为 O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过 key 对象的 equals 方法逐一比对查找。所以，性能考虑，HashMap 中的链表出现越少，性能才会越好。

9、线程有几种状态，产生的条件是什么



（1）新建状态(New) ：线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。

（2）就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的 start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被 CPU 调度执行。

（3）运行状态(Running)：线程获取 CPU 权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。

（4）阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃 CPU 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

1、等待阻塞 -- 通过调用线程的 wait()方法，让线程等待某工作的完成。

2、同步阻塞 -- 线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。

3、其他阻塞 -- 通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。

（5）死亡状态(Dead)：线程执行完了或者因异常退出了 run()方法，该线程结束生命周期。

10、产生死锁的基本条件

产生死锁的原因：

（1） 因为系统资源不足。

（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

死锁的解除与预防：

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。

11、Object 中有哪些方法

（1）protected Object clone()--->创建并返回此对象的一个副本。

（2）boolean equals(Object obj)--->指示某个其他对象是否与此对象“相等”。

（3）protected void finalize()--->当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

（4）Class<? extendsObject> getClass()--->返回一个对象的运行时类。

（5）int hashCode()--->返回该对象的哈希码值。

（6）void notify()--->唤醒在此对象监视器上等待的单个线程。

（7）void notifyAll()--->唤醒在此对象监视器上等待的所有线程。

（8）String toString()--->返回该对象的字符串表示。

（9）void wait()--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

void wait(long timeout)--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll()方法，或者超过指定的时间量。

void wait(long timeout, int nanos)--->导致当前的线程等待，直到其他线程调用此对象的 notify()

12、描述一下 JVM 加载 class 文件的原理机制?

Java 中的所有类，都需要由类加载器装载到 JVM 中才能运行。类加载器本身也是一个类，而它的工作就是把 class 文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

Java 类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到 jvm 中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

加载过程：

（1）装载:查找和导入 class 文件;

（2）连接:

1、检查:检查载入的 class 文件数据的正确性;

2、准备:为类的静态变量分配存储空间;

3、解析:将符号引用转换成直接引用(这一步是可选的)

（3）初始化:初始化静态变量，静态代码块。

这样的过程在程序调用类的静态成员的时候开始执行，所以静态方法 main()才会成为一般程序的入口方法。类的构造器也会引发该动作。

扩展：

Java 的类加载器有三个，对应 Java 的三种类:

（1）Bootstrap Loader // 负责加载系统类 (指的是内置类，像是 String，对应于 C#中的 System 类和 C/C++标准库中的类)

（2）ExtClassLoader // 负责加载扩展类(就是继承类和实现类)

（3）AppClassLoader // 负责加载应用类(程序员自定义的类)

三个加载器各自完成自己的工作，但它们是如何协调工作呢？哪一个类该由哪个类加载器完成呢？为了解决这个问题，Java 采用了委托模型机制。

委托模型机制的工作原理很简单：当类加载器需要加载类的时候，先请示其 Parent(即上一层加载器)在其搜索路径载入，如果找不到，才在自己的搜索路径搜索该类。这样的顺序其实就是加载器层次上自顶而下的搜索，因为加载器必须保证基础类的加载。之所以是这种机制，还有一个安全上的考虑：如果某人将一个恶意的基础类加载到 jvm，委托模型机制会搜索其父类加载器，显然是不可能找到的，自然就不会将该类加载进来。

13、请写出你最常见的 5 个 RuntimeException

（1）java.lang.NullPointerException 空指针异常；出现原因：调用了未经初始化的对象或者是不存在的对象。

（2）java.lang.ClassNotFoundException 指定的类找不到；出现原因：类的名称和路径加载错误；通常都是程序试图通过字符串来加载某个类时可能引发异常。

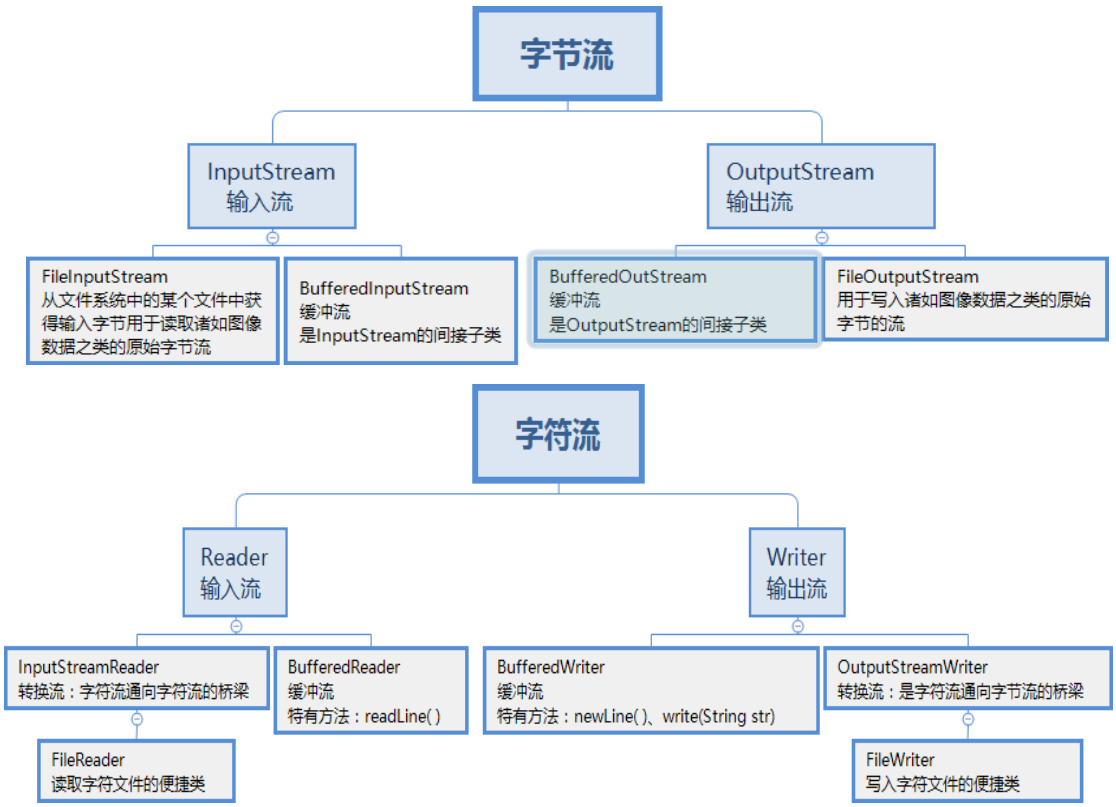
（3）java.lang.NumberFormatException 字符串转换为数字异常；出现原因：字符型数据中包含非数字型字符。

（4）java.lang.IndexOutOfBoundsException 数组角标越界异常，常见于操作数组对象时发生。

（5）java.lang.IllegalArgumentException 方法传递参数错误。

（6）java.lang.ClassCastException 数据类型转换异常。

14、Java 中有几种类型的流



15、字节流如何转为字符流

字节输入流转字符输入流通过 InputStreamReader 实现，该类的构造函数可以传入 InputStream 对象。

字节输出流转字符输出流通过 OutputStreamWriter 实现，该类的构造函数可以传入 OutputStream 对象。

16、什么是 java 序列化，如何实现 java 序列化？

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。

序列化的实现：将需要被序列的类实现 Serializable 接口，该接口没有需要实现的方法，implements Serializable 只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个 ObjectOutputStream(对象流)对象，接着，使用 ObjectOutputStream 对象的 writeObject(Object obj)方法就可以将参数为 obj 的对象写出(即保存其状态)，要恢复的话则用输入流。

二、数据结构和算法

1、选择排序

```
/**
   * author:sam
   * date:2018/1/26 14:11
   * describe:选择排序
   */
public void selectSort(int[] arr){
      for (int i = 0; i < arr.length; i++) {
            int minIndex = i;
            for (int j = i+1; j < arr.length && arr[minIndex] > arr[j]; j++) {
                  minIndex = j;
            }
            SortUtils.swap(arr,minIndex,i);
      }
}
```

2、插入排序

```
/**
   * author:sam
   * date:2018/3/7 10:49
   * describe:插入排序
   */
@Override
public void insertSort(int[] arr) {
      for (int i = 0; i < arr.length; i++) {
            for (int j = i; j > 0 && arr[j] < arr[j-1]; j--) {
                  SortUtils.swap(arr,j,j-1);
            }
      }
}

/**
   * author:sam
   * date:2018/3/7 12:21
   * describe:插入排序优化
   */
@Override
public void optimizeInsertSort(int[] arr) {

      for (int i = 1; i < arr.length; i++) {
            int e = arr[i];
```

```
            int index = i;
            for (int j = i; j > 0 && arr[j-1] > e; j--) {
                arr[j] = arr[j - 1];
                index = j - 1;
            }
            arr[index] = e;
        }
    }
    /**
     * author:sam
     * date:2018/3/8 18:43
     * describe:插入排序（数组部分内容）
     */
    public void insertionSort(int[] arr,int l,int r){

        for (int i = l + 1; i <= r; i++) {
            int tmp = arr[i];
            int index = i;
            for (int j = i - 1; j >= l && arr[j] > tmp; j--) {
                arr[j + 1] = arr[j];
                index = j;
            }
            arr[index] = tmp;
        }
    }
```

3、冒泡排序

```
    /**
     * author:sam
     * date:2018/3/7 14:37
     * describe:冒泡排序
     */
    public void bubbleSort(int[] arr){

        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr.length - i -1; j++) {
                if(arr[j] > arr[j+1]){
                    SortUtils.swap(arr,j,j+1);
                }
            }
        }
    }
    /**
```

```
    * author:sam
    * date:2018/3/7 14:49
    * describe:优化冒泡排序
    */
@Override
public void optimizeBubbleSort(int[] arr){
    boolean flag;
    int n = arr.length;
    do{
        flag = false;
        for (int i = 0; i < n - 1; i++) {
            if(arr[i] > arr[i + 1]){
                SortUtils.swap(arr,i,i + 1);
                flag = true;
            }
        }
        n--;
    }while(flag);
}
```

4、shell 排序

```
/**
    * author:sam
    * date:2018/3/8 14:09
    * describe:希尔排序
    */
public void shellSort(int[] arr){
    int n = 1;
    int len = arr.length;
    //确定序列
    while(n < len/3){
        n = 3*n + 1;
    }
    while(n>0){
        for (int i = n; i < len; i += n) {
            int tmp = arr[i];
            int index = i;
            for (int j = i; i-n > 0 && arr[i-n] > arr[i]; j -= n) {
                arr[i] = arr[i-n];
                index = i - n;
            }
            arr[index] = tmp;
        }
```

```
            n /= 3;
        }
    }
```

5、归并排序

```
    /**
     * author:sam
     * date:2018/3/8 16:27
     * describe:归并排序算法
     */
    public void mergeSort(int[] arr){
        __mergeSort(arr,0,arr.length-1);
    }
    /**
     * author:sam
     * date:2018/3/8 16:15
     * describe:归并排序递归函数，arr[l,r]
     */
    private void __mergeSort(int[] arr,int l,int r){
        /*if(l >= r)
            return;*/
        if(r - l <= 15){
            insertionSort(arr,l,r);
            return;
        }

        int mid = (l + r)/2;
        __mergeSort(arr,l,mid);
        __mergeSort(arr,mid + 1,r);
        if(arr[mid] > arr[mid + 1])//归并排序优化
            __merge(arr,l,mid,r);
    }
    /**
     * author:sam
     * date:2018/3/8 16:44
     * describe:合并算法
     */
    private void __merge(int[] arr,int l,int mid,int r){
        //复制数组
        int[] aux = new int[r-l+1];
        for (int i = l; i <= r; i++) {
            aux[i - l] = arr[i];
        }
```

```
        //定义索引
        int i = l,j = mid + 1;
        for (int k = l; k <= r; k++) {
            if(i > mid){
                arr[k] = aux[j - l];
                j++;
            }else if(j > r){
                arr[k] = aux[i - l];
                i++;
            }else if(aux[i - l] > aux[j - l]){
                arr[k] = aux[j - l];
                j++;
            }else{
                arr[k] = aux[i - l];
                i++;
            }
        }
    }

    /**
     * author:sam
     * date:2018/3/8 19:17
     * describe:自底向上的归并排序
     *  没有使用
     */
    public void mergeSortBU(int[] arr){
        int len = arr.length;
        for (int i = 1; i < len/2; i += i) {
            for (int j = 0; j + i < len; j += 2*i) {
                if(j + 2*i -1 > len - 1){
                    __merge(arr,j ,j + i - 1,len -1);
                }else{
                    __merge(arr,j,i + j - 1,j + 2*i -1);
                }
            }
        }
    }
```

6、快速排序

```
    /**
     * author:sam
     * date:2018/3/8 14:10
     * describe:快速排序
```

```
     */
    public void quickSort(int[] arr){
        __quickSort(arr,0,arr.length - 1);
    }
    private void __quickSort(int[] arr,int l,int r){
        if(r - l <= 15){//优化
            insertionSort(arr,l,r);
            return;
        }
        int p = __partition(arr,l,r);
        __quickSort(arr,l,p - 1);
        __quickSort(arr,p + 1,r);
    }
    /**
     * author:sam
     * date:2018/3/9 16:22
     * describe:普通分隔算法
     */
    //arr[l...r]
    private int __partition(int[] arr, int l, int r) {
        int random = SortUtils.generateRandom(l, r);
        //在近乎有序数组的情况下，快速排序交换比较频繁，利用生成随机数来确定中间
的数据为原点数据
        SortUtils.swap(arr,l,random);//与初始位置进行数据交换
        int v = arr[l];
        int j = l;
        for (int i = l + 1; i <= r; i++) {
            if(arr[i] < v)
                SortUtils.swap(arr,++j,i);
        }
        SortUtils.swap(arr,l,j);
        return j;
    }

    /**
     * author:sam
     * date:2018/3/9 17:08
     * describe:双路快排
     */
    public void quickSort2(int[] arr){
        __quickSort2(arr,0,arr.length-1);
    }
    /**
     * author:sam
```

```
     * date:2018/3/9 17:03
     * describe:双路快排
     */
    public void __quickSort2(int[] arr,int l,int r){
        if(r - l >= 15) {
            insertionSort(arr, l, r);
            return;
        }
        int p = __partitionDouble(arr, l, r);
        __partitionDouble(arr,l,p - 1);
        __partitionDouble(arr,p + 1,r);
    }
    /**
     * author:sam
     * date:2018/3/9 16:23
     * describe:双路分割算法
     */
    private int __partitionDouble(int[] arr,int l,int r){
        int random = SortUtils.generateRandom(l, r);
        SortUtils.swap(arr,l,random);
        /*int v = arr[l];
        int j = l,k = r;
        for (int m = l + 1,n = r;m <= n && j <= k; m++ ,n--) {
            if(arr[m] <= v)
                SortUtils.swap(arr,++j,m);
            if(arr[n] >= v)
                SortUtils.swap(arr,--k,n);
        }
        SortUtils.swap(arr,l,j);*/
        int v = arr[l];
        int i = l + 1,j = r;
        while(true){
            while (arr[i] <= v) i++;
            while (arr[j] >= v) j--;
            if(i > j) break;
            SortUtils.swap(arr,i,j);
            i++;j--;
        }
        SortUtils.swap(arr,l,i);
        return i ;
    }
}
```

三、设计模式

1、你所知道的设计模式有哪些

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

2、单例设计模式

（1）单例模式定义：

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个 Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免政出多头。

（2）单例模式特点：

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

单例模式保证了全局对象的唯一性，比如系统启动读取配置文件就需要单例保证配置的一致性。

（3）线程安全的问题

一方面在获取单例的时候，要保证不能产生多个实例对象，后面会详细讲到五种实现方式；

另一方面，在使用单例对象的时候，要注意单例对象内的实例变量是会被多线程共享的，推荐使用无状态的对象，不会因为多个线程的交替调度而破坏自身状态导致线程安全问题，比如我们常用的 VO，DTO 等（局部变量是在用户栈中的，而且用户栈本身就是线程私有的内存区域，所以不存在线程安全问题）。

（4）单例模式的选择

还记得我们最早使用的 MVC 框架 Struts1 中的 action 就是单例模式的，而到了 Struts2 就使用了多例。在 Struts1 里，当有多个请求访问，每个都会分配一个新线程，在这些线程，操作的都是同一个 action 对象，每个用户的数据都是不同的，而 action 却只有一个。到了 Struts2， action 对象为每一个请求产生一个实例，并不会带来线程安全问题（实际上 servlet

容器给每个请求产生许多可丢弃的对象，但是并没有影响到性能和垃圾回收问题，有时间会做下研究）。

（5）实现单例模式的方式

1.饿汉式单例（立即加载方式）

```
// 饿汉式单例 public class Singleton1 {
    // 私有构造
    private Singleton1() {}

    private static Singleton1 single = new Singleton1();

    // 静态工厂方法
    public static Singleton1 getInstance() {
        return single;
    }
}
```

饿汉式单例在类加载初始化时就创建好一个静态的对象供外部使用，除非系统重启，这个对象不会改变，所以本身就是线程安全的。

Singleton 通过将构造方法限定为 private 避免了类在外部被实例化，在同一个虚拟机范围内，Singleton 的唯一实例只能通过 getInstance()方法访问。（事实上，通过 Java 反射机制是能够实例化构造方法为 private 的类的，那基本上会使所有的 Java 单例实现失效。此问题在此处不做讨论，姑且闭着眼就认为反射机制不存在。）

2.懒汉式单例（延迟加载方式）

```
// 懒汉式单例 public class Singleton2 {

    // 私有构造
    private Singleton2() {}

    private static Singleton2 single = null;

    public static Singleton2 getInstance() {
        if(single == null){
            single = new Singleton2();
        }
        return single;
    }
}
```

该示例虽然用延迟加载方式实现了懒汉式单例，但在多线程环境下会产生多个 single 对象，如何改造请看以下方式:

使用 synchronized 同步锁

```
public class Singleton3 {
    // 私有构造
    private Singleton3() {}
```

```
    private static Singleton3 single = null;

    public static Singleton3 getInstance() {

        // 等同于 synchronized public static Singleton3 getInstance()
        synchronized(Singleton3.class){
          // 注意：里面的判断是一定要加的，否则出现线程安全问题
            if(single == null){
                single = new Singleton3();
            }
        }
        return single;
    }
}
```

在方法上加 synchronized 同步锁或是用同步代码块对类加同步锁，此种方式虽然解决了多个实例对象问题，但是该方式运行效率却很低下，下一个线程想要获取对象，就必须等待上一个线程释放锁之后，才可以继续运行。

```
public class Singleton4 {
    // 私有构造
    private Singleton4() {}

    private static Singleton4 single = null;

    // 双重检查
    public static Singleton4 getInstance() {
        if (single == null) {
            synchronized (Singleton4.class) {
                if (single == null) {
                    single = new Singleton4();
                }
            }
        }
        return single;
    }
}
```

使用双重检查进一步做了优化，可以避免整个方法被锁，只对需要锁的代码部分加锁，可以提高执行效率。

3.静态内部类实现

```
public class Singleton6 {
    // 私有构造
    private Singleton6() {}

    // 静态内部类
```

```
    private static class InnerObject{
        private static Singleton6 single = new Singleton6();
    }

    public static Singleton6 getInstance() {
        return InnerObject.single;
    }
}
```

静态内部类虽然保证了单例在多线程并发下的线程安全性，但是在遇到序列化对象时，默认的方式运行得到的结果就是多例的。这种情况不多做说明了，使用时请注意。

4.static 静态代码块实现

```
public class Singleton6 {

    // 私有构造
    private Singleton6() {}

    private static Singleton6 single = null;

    // 静态代码块
    static{
        single = new Singleton6();
    }

    public static Singleton6 getInstance() {
        return single;
    }
}
```

5.内部枚举类实现

```
public class SingletonFactory {

    // 内部枚举类
    private enum EnmuSingleton{
        Singleton;
        private Singleton8 singleton;

        //枚举类的构造方法在类加载是被实例化
        private EnmuSingleton(){
            singleton = new Singleton8();
        }
        public Singleton8 getInstance(){
            return singleton;
        }
```

```
        }
        public static Singleton8 getInstance() {
            return EnmuSingleton.Singleton.getInstance();
        }
    }
    class Singleton8{
        public Singleton8(){}
    }
```

3、工厂设计模式

（1）工厂模式（Factory Method）

常用的工厂模式是静态工厂，利用 static 方法，作为一种类似于常见的工具类 Utils 等辅助效果，一般情况下工厂类不需要实例化。

```
    interface food{}
    class A implements food{}class B implements food{}class C implements food{}
public class StaticFactory {

        private StaticFactory(){}

        public static food getA(){    return new A(); }
        public static food getB(){    return new B(); }
        public static food getC(){    return new C(); }
    }
    class Client{
        //客户端代码只需要将相应的参数传入即可得到对象
        //用户不需要了解工厂类内部的逻辑。
        public void get(String name){
            food x = null ;
            if ( name.equals("A")) {
                x = StaticFactory.getA();
            }else if ( name.equals("B")){
                x = StaticFactory.getB();
            }else {
                x = StaticFactory.getC();
            }
        }
    }
```

（2）抽象工厂模式（Abstract Factory）

一个基础接口定义了功能，每个实现接口的子类就是产品，然后定义一个工厂接口，实现了工厂接口的就是工厂，这时候，接口编程的优点就出现了，我们可以新增产品类（只需要实现产品接口），只需要同时新增一个工厂类，客户端就可以轻松调用新产品的代码。

抽象工厂的灵活性就体现在这里，无需改动原有的代码，毕竟对于客户端来说，静态工厂模式在不改动 StaticFactory 类的代码时无法新增产品，如果采用了抽象工厂模式，就

可以轻松的新增拓展类。

实例代码：

```
interface food{}
class A implements food{}class B implements food{}
interface produce{ food get();}
class FactoryForA implements produce{
    @Override
    public food get() {
        return new A();
    }
}class FactoryForB implements produce{
    @Override
    public food get() {
        return new B();
    }
}public class AbstractFactory {
    public void ClientCode(String name){
        food x= new FactoryForA().get();
        x = new FactoryForB().get();
    }
}
```

4、装饰器模式

给一类对象增加新的功能，装饰方法与具体的内部逻辑无关。例如：

```
interface Source{ void method();}public class Decorator implements Source{

    private Source source ;
    public void decotate1(){
        System.out.println("decorate");
    }
    @Override
    public void method() {
        decotate1();
        source.method();
    }
}
```

5、代理模式（Proxy）

客户端通过代理类访问，代理类实现具体的实现细节，客户只需要使用代理类即可实现操作。

这种模式可以对旧功能进行代理，用一个代理类调用原有的方法，且对产生的结果进行

控制。

```
interface Source{ void method();}
class OldClass implements Source{
    @Override
    public void method() {
    }
}
class Proxy implements Source{
    private Source source = new OldClass();

    void doSomething(){}
    @Override
    public void method() {
        new Class1().Func1();
        source.method();
        new Class2().Func2();
        doSomething();
    }
}
```

四、java 高级

1、什么是线程池，如何使用？

线程池就是事先将多个线程对象放到一个容器中，当使用的时候就不用 new 线程而是直接去池中拿线程即可，节省了开辟子线程的时间，提高的代码执行效率。

在 JDK 的 java.util.concurrent.Executors 中提供了生成多种线程池的静态方法。

ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();

ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(4);

ScheduledExecutorService newScheduledThreadPool = Executors.newScheduledThreadPool(4);

ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();

然后调用他们的 execute 方法即可。

优点：

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

2、常用的线程池有哪些？

newSingleThreadExecutor：创建一个单线程的线程池，此线程池保证所有任务的执行顺

序按照任务的提交顺序执行。

newFixedThreadPool：创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。

newCachedThreadPool：创建一个可缓存的线程池，此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

newScheduledThreadPool：创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

3、heap 和 stack 有什么区别

（1）申请方式

stack:由系统自动分配。例如，声明在函数中一个局部变量 int b; 系统自动在栈中为 b 开辟空间

heap:需要程序员自己申请，并指明大小，在 c 中 malloc 函数，对于 Java 需要手动 new Object()的形式开辟

（2）申请后系统的响应

stack：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

heap：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

（3）申请大小的限制

stack：栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（也有的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。

heap：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的， 自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见， 堆获得的空间比较灵活，也比较大。

（4）申请效率的比较：

stack：由系统自动分配，速度较快。但程序员是无法控制的。

heap：由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

（5）heap 和 stack 中的存储内容

Stack：在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址， 然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

Heap：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

4、解释内存中的栈（stack）、堆（heap）和方法区（method area）的用法

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用 JVM 中的栈空间；而通过 new 关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为 Eden、Survivor（又可分为 From Survivor 和 To Survivor）、Tenured；方法区和堆都是各个线程共享的内存区域，用于存储已经被 JVM 加载的类信息、常量、静态变量、JIT 编译器编译后的代码等数据；程序中的字面量（literal）如直接书写的 100、"hello" 和常量都是放在常量池中，常量池是方法区的一部分。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过 JVM 的启动参数来进行调整，栈空间用光了会引发 StackOverflowError，而堆和常量池空间不足则会引发 OutOfMemoryError。

String str = new String("hello");

上面的语句中变量 str 放在栈上，用 new 创建出来的字符串对象放在堆上，而"hello"这个字面量是放在方法区的。

五、JavaWEB

1、http 的长连接和短连接

HTTP 协议有 HTTP/1.0 版本和 HTTP/1.1 版本。HTTP1.1 默认保持长连接（HTTP persistent connection，也翻译为持久连接），数据传输完成了保持 TCP 连接不断开（不发 RST 包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

在 HTTP/1.0 中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。从 HTTP/1.1 起，默认使用的是长连接，用以保持连接特性。

2、http 常见的状态码有哪些？

200 OK //客户端请求成功

301 Moved Permanently（永久移除），请求的 URL 已移走。Response 中应该包含一个 Location URL，说明资源现在所处的位置

302 found 重定向

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的 URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

3、GET 和 POST 的区别？

（1）GET 请求的数据会附在 URL 之后（就是把数据放置在 HTTP 协议头中），以?分割 URL 和传输数据，参数之间以&相连，如：login.action?name=zhagnsan&password=123456。POST 把提交的数据则放置在是 HTTP 包的包体中。
（2）GET 方式提交的数据最多只能是 1024 字节，理论上 POST 没有限制，可传较大量的数据。其实这样说是错误的，不准确的：“GET 方式提交的数据最多只能是 1024 字节"，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟 URL 的长度有直接关系了。而实际上，URL 不存在参数上限的问题，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制。IE 对 URL 长度的限制是 2083 字节(2K+35)。对于其他浏览器，如 Netscape、FireFox 等，理论上没有长度限制，其限制取决于操作系统的支持。
（3）POST 的安全性要比 GET 的安全性高。注意：这里所说的安全性和上面 GET 提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的 Security 的含义，比如：通过 GET 提交数据，用户名和密码将明文出现在 URL 上，因为(1)登录页面有可能被浏览器缓存，(2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用 GET 提交数据还可能会造成 Cross-site request forgery 攻击。

Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求，在 FORM（表单）中，Method

默认为"GET"，实质上，GET 和 POST 只是发送机制不同，并不是一个取一个发！

3、Cookie 和 Session 的区别

Cookie 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie。

Session 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

Cookie 和 session 的不同点：

1、无论客户端做怎样的设置，session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie。

2、在存储的数据量方面：session 能够存储任意的 java 对象，cookie 只能存储 String 类型的对象。

4、在单点登录中，如果 cookie 被禁用了怎么办？

单点登录的原理是后端生成一个 session ID，然后设置到 cookie，后面的所有请求浏览器都会带上 cookie， 然后服务端从 cookie 里获取 session ID，再查询到用户信息。所以，

保持登录的关键不是 cookie，而是通过 cookie 保存和传输的 session ID，其本质是能获取用户信息的数据。除了 cookie，还通常使用 HTTP 请求头来传输。但是这个请求头浏览器不会像 cookie 一样自动携带，需要手工处理。

5、什么是 jsp,什么是Servlet？jsp 和 Servlet 有什么区别?

jsp 本质上就是一个 Servlet，它是 Servlet 的一种特殊形式（由 SUN 公司推出），每个 jsp 页面都是一个 servlet 实例。

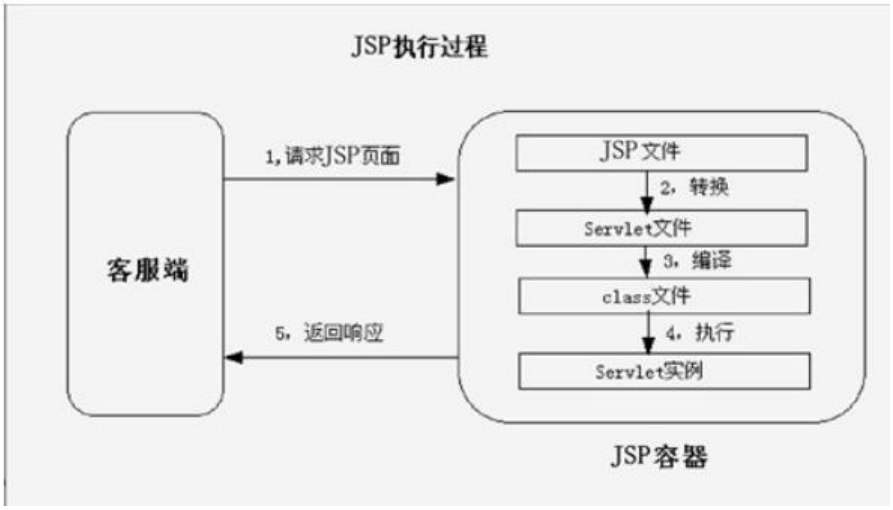
Servlet 是由 Java 提供用于开发 web 服务器应用程序的一个组件，运行在服务端，由 servlet 容器管理，用来生成动态内容。一个 servlet 实例是实现了特殊接口 Servlet 的 Java 类，所有自定义的 servlet 均必须实现 Servlet 接口。

区别：

jsp 是 html 页面中内嵌的 Java 代码，侧重页面显示；

Servlet 是 html 代码和 Java 代码分离，侧重逻辑控制，mvc 设计思想中 jsp 位于视图层，servlet 位于控制层

Jsp 运行机制：如下图



JVM 只能识别 Java 类，并不能识别 jsp 代码！web 容器收到以.jsp 为扩展名的 url 请求时，会将访问请求交给 tomcat 中 jsp 引擎处理，每个 jsp 页面第一次被访问时，jsp 引擎将 jsp 代码解释为一个 servlet 源程序，接着编译 servlet 源程序生成.class 文件，再有 web 容器 servlet 引擎去装载执行 servlet 程序，实现页面交互。

6、servlet 生命周期

Servlet 加载—>实例化—>服务—>销毁。

（1）生命周期详解：

init（）：

在 Servlet 的生命周期中，仅执行一次 init()方法。它是在服务器装入 Servlet 时执行的，负责初始化 Servlet 对象。可以配置服务器，以在启动服务器或客户机首次访问 Servlet 时装入 Servlet。无论有多少客户机访问 Servlet，都不会重复执行 init（）。

service（）：

它是 Servlet 的核心，负责响应客户的请求。每当一个客户请求一个 HttpServlet 对象，该对象的 Service()方法就要调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在 HttpServlet 中已存在 Service()方法。默认的服务功能是调用与 HTTP 请求的方法相应的 do 功能。

destroy（）：

仅执行一次，在服务器端停止且卸载 Servlet 时执行该方法。当 Servlet 对象退出生命周期时，负责释放占用的资源。一个 Servlet 在运行 service()方法时可能会产生其他的线程，因此需要确认在调用 destroy()方法时，这些线程已经终止或完成。

（2）如何与 Tomcat 结合工作步骤：

1、Web Client 向 Servlet 容器（Tomcat）发出 Http 请求

2、Servlet 容器接收 Web Client 的请求

3、Servlet 容器创建一个 HttpRequest 对象，将 Web Client 请求的信息封装到这个对象中。

4、Servlet 容器创建一个 HttpResponse 对象

5、Servlet 容器调用 HttpServlet 对象的 service 方法，把 HttpRequest 对象与 HttpResponse 对象作为参数传给 HttpServlet 对象。

6、HttpServlet 调用 HttpRequest 对象的有关方法，获取 Http 请求信息。

7、HttpServlet 调用 HttpResponse 对象的有关方法，生成响应数据。

7、servlet 特性

单例多线程

六、数据库

1、jdbc 操作数据库流程

第一步：Class.forName()加载数据库连接驱动；

第二步：DriverManager.getConnection()获取数据连接对象;

第三步：根据 SQL 获取 sql 会话对象，有 2 种方式 Statement、PreparedStatement ;

第四步：执行 SQL 处理结果集，执行 SQL 前如果有参数值就设置参数值 setXXX();

第五步：关闭结果集、关闭会话、关闭连接。

2、关系数据库中连接池的机制是什么？

前提：为数据库连接建立一个缓冲池。

1：从连接池获取或创建可用连接

2：使用完毕之后，把连接返回给连接池

3：在系统关闭前，断开所有连接并释放连接占用的系统资源

4：能够处理无效连接，限制连接池中的连接总数不低于或者不超过某个限定值。其中

有几个概念需要大家理解：

最小连接数是连接池一直保持的数据连接。如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费掉。

最大连接数是连接池能申请的最大连接数。如果数据连接请求超过此数，后面的数据连接请求将被加入到等待队列中，这会影响之后的数据库操作。

如果最小连接数与最大连接数相差太大，那么，最先的连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，它将被放到连接池中等待重复使用或是空闲超时后被释放。

上面的解释，可以这样理解：数据库池连接数量一直保持一个不少于最小连接数的数量，当数量不够时，数据库会创建一些连接，直到一个最大连接数，之后连接数据库就会等待。

3、SQL 的 select 语句完整的执行顺序

SQL Select 语句完整的执行顺序：

1、from 子句组装来自不同数据源的数据；

2、where 子句基于指定的条件对记录行进行筛选；

3、group by 子句将数据划分为多个分组；

4、使用聚集函数进行计算；

5、使用 having 子句筛选分组；

6、计算所有的表达式； 7、select 的字段；

8、使用 order by 对结果集进行排序。

SQL 语言不同于其他编程语言的最明显特征是处理代码的顺序。在大多数据库语言中，代码按编码顺序被处理。但在 SQL 语句中，第一个被处理的子句式 FROM，而不是第一出现的 SELECT。SQL 查询处理的步骤序号：

FROM <left_table>

<join_type> JOIN <right_table>

ON <join_condition>

WHERE <where_condition>

GROUP BY <group_by_list>

WITH {CUBE | ROLLUP}

HAVING <having_condition>

SELECT

DISTINCT

ORDER BY <order_by_list>

<TOP_specification> <select_list>

以上每个步骤都会产生一个虚拟表，该虚拟表被用作下一个步骤的输入。这些虚拟表对调用者(客户端应用程序或者外部查询)不可用。只有最后一步生成的表才会会给调用者。如果没有在查询中指定某一个子句， 将跳过相应的步骤。

逻辑查询处理阶段简介：

1、 FROM：对 FROM 子句中的前两个表执行笛卡尔积(交叉联接)，生成虚拟表 VT1。

2、 ON：对 VT1 应用 ON 筛选器，只有那些使为真才被插入到 TV2。

3、 OUTER (JOIN):如果指定了 OUTER JOIN(相对于 CROSS JOIN 或 INNER JOIN)，保留表

中未找到匹配的行将作为外部行添加到 VT2，生成 TV3。如果 FROM 子句包含两个以上的表，则对上一个联接生成的结果表和下一个表重复执行步骤 1 到步骤 3，直到处理完所有的表位置。

4、 WHERE：对 TV3 应用 WHERE 筛选器，只有使为 true 的行才插入 TV4。

5、 GROUP BY：按 GROUP BY 子句中的列列表对 TV4 中的行进行分组，生成 TV5。

6、 CUTE|ROLLUP：把超组插入 VT5，生成 VT6。

7、 HAVING：对 VT6 应用 HAVING 筛选器，只有使为 true 的组插入到 VT7。

8、 SELECT：处理 SELECT 列表，产生 VT8。

9、 DISTINCT：将重复的行从 VT8 中删除，产品 VT9。

4、Mysql 性能优化

1、当只要一行数据时使用 limit 1

查询时如果已知会得到一条数据，这种情况下加上 limit 1 会增加性能。因为 mysql 数据库引擎会在找到一条结果停止搜索，而不是继续查询下一条是否符合标准直到所有记录查询完毕。

2、选择正确的数据库引擎

Mysql 中有两个引擎 MyISAM 和 InnoDB，每个引擎有利有弊。

MyISAM 适用于一些大量查询的应用，但对于有大量写功能的应用不是很好。甚至你只需要

update 一个字段整个表都会被锁起来。而别的进程就算是读操作也不行要等到当前 update 操作完

成之后才能继续进行。另外，MyISAM 对于 select count(*)这类操作是超级快的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用会比 MyISAM 还慢，但是支持“行锁”，所以在写操作比较多的时候会比较优秀。并且，它支持很多的高级应用，例如：事物。

3、用 not exists 代替 not in

Not exists 用到了连接能够发挥已经建立好的索引的作用，not in 不能使用索引。Not in 是最慢的方式要同每条记录比较，在数据量比较大的操作红不建议使用这种方式。

4、对操作符的优化，尽量不采用不利于索引的操作符

如：in not in is null is not null<> 等某个字段总要拿来搜索，为其建立索引：

Mysql 中可以利用 alter table 语句来为表中的字段添加索引，语法为：alter table 表明 add index (字段名)；

5、SQL 语句优化

（1）where 子句中可以对字段进行 null 值判断吗？

可以，比如 select id from t where num is null 这样的 sql 也是可以的。但是最好不要给数据库留 NULL，尽可能的使用 NOT NULL 填充数据库。不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段，null 不占用空间。可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：select id from t where

num= 0。

（2）select * fromadmin left join log on admin.admin_id = log.admin_id where log.admin_id>10 如何优化?

优化为：select * from (select * from admin where admin_id>10) T1 lef join log on T1.admin_id = log.admin_id。

使用 JOIN 时候，应该用小的结果驱动大的结果（left join 左边表结果尽量小如果有条件应该放到左边先处理， right join 同理反向），同时尽量把牵涉到多表联合的查询拆分多个 query（多个连表查询效率低，容易到之后锁表和阻塞）。

（3）limit 的基数比较大时使用 between

例如：select * from admin order by admin_id limit 100000,10

优化为：select * from admin where admin_id between 100000 and 100010 order by admin_id。

（4）尽量避免在列上做运算，这样导致索引失效

例如：select * from admin where year(admin_time)>2014

优化为： select * from admin where admin_time> '2014-01-01′

6、说说 mysql 事务

保证数据的一致性

基本特征：

原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

一致性：在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。

隔离性：隔离状态执行事务，使它们好像是系统在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，为了防止事务操作间的混淆， 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。

持久性：在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

隔离级别：

读未提交（READ UNCOMMITTED）：未提交读隔离级别也叫读脏，就是事务可以读取其它事务未提交的数据。

读已提交（READ COMMITTED）：在其它数据库系统比如 SQL Server 默认的隔离级别就是提交读，已提交读隔离级别就是在事务未提交之前所做的修改其它事务是不可见的。

可重复读（REPEATABLE READ）：保证同一个事务中的多次相同的查询的结果是一致的，比如一个事务一开始查询了一条记录然后过了几秒钟又执行了相同的查询，保证两次查询的结果是相同的，可重复读也是 mysql 的默认隔离级别。

可串行化（SERIALIZABLE）：可串行化就是保证读取的范围内没有新的数据插入，比如事务第一次查询得到某个范围的数据，第二次查询也同样得到了相同范围的数据，中间没有新的数据插入到该范围中。

传播行为：

事务传播行为（propagation behavior）指的就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行。

例如：methodA 事务方法调用 methodB 事务方法时，methodB 是继续在调用者 methodA 的事务中运行呢，还是为自己开启一个新事务运行，这就是由 methodB 的事务传播行为决定的。

1.PROPAGATION_REQUIRED - 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

2.PROPAGATION_SUPPORTS - 支持当前事务，如果当前没有事务，就以非事务方式执行。

3.PROPAGATION_MANDATORY - 支持当前事务，如果当前没有事务，就抛出异常。

4.PROPAGATION_REQUIRES_NEW - 新建事务，如果当前存在事务，把当前事务挂起。

5.PROPAGATION_NOT_SUPPORTED - 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

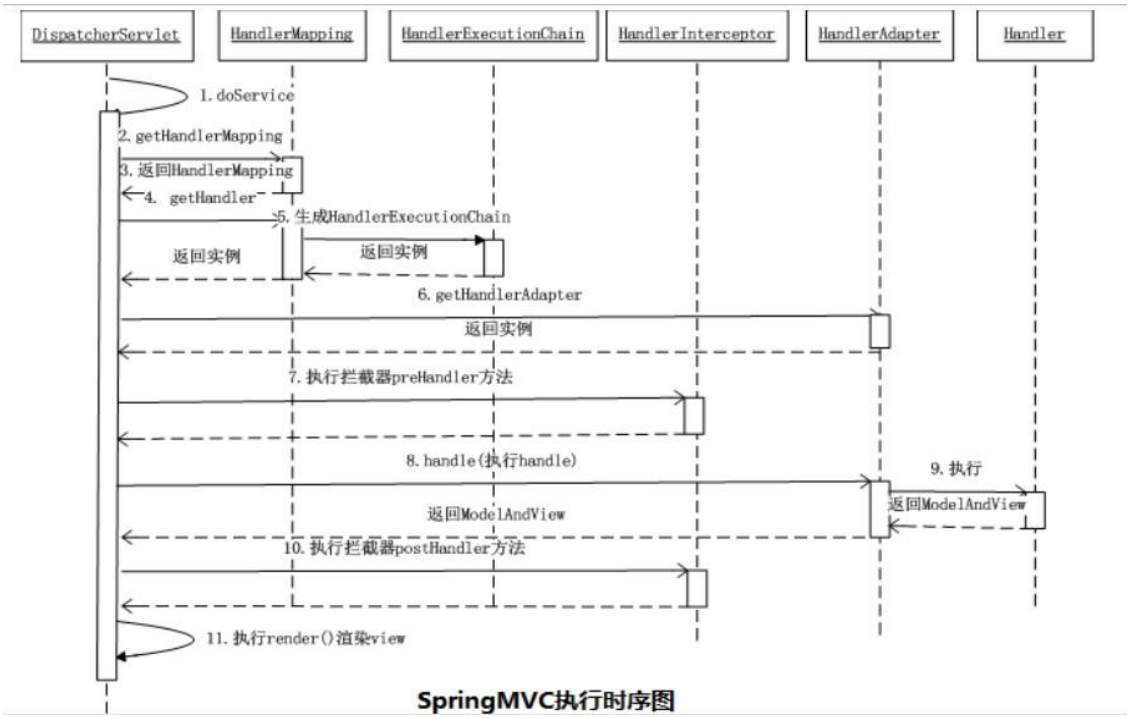
6.PROPAGATION_NEVER - 以非事务方式执行，如果当前存在事务，则抛出异常。

7.PROPAGATION_NESTED - 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与 PROPAGATION_REQUIRED 类似的操作。

七、SpringMVC、Mybatis、Spring

1、SpringMVC 的工作原理

（1）用户向服务器发送请求，请求被 springMVC 前端控制器 DispatchServlet 捕获；
（2）DispatcherServle 对请求 URL 进行解析，得到请求资源标识符（URL），然后根据该 URL 调用 HandlerMapping 将请求映射到处理器 HandlerExcutionChain；
（3）DispatchServlet 根据获得 Handler 选择一个合适的 HandlerAdapter 适配器处理；
（4）Handler 对数据处理完成以后将返回一个 ModelAndView（）对象给 DisPatchServlet;
（5）Handler 返回的 ModelAndView() 只是一个逻辑视图并不是一个正式的视图，DispatcherSevlet 通过 ViewResolver 试图解析器将逻辑视图转化为真正的视图 View;
（6）DispatcherServle 通过 model 解析出 ModelAndView()中的参数进行解析最终展现出完整的 view 并返回给客户端;



SpringMVC执行时序图

2、SpringMVC 常用注解都有哪些？

@requestMapping 用于请求 url 映射。
@RequestBody 注解实现接收 http 请求的 json 数据，将 json 数据转换为 java 对象。
@ResponseBody 注解实现将 controller 方法返回对象转化为 json 响应给客户。

3、如何开启注解处理器和适配器？

我们在项目中一般会在 springmvc.xml 中通过开启 <mvc:annotation-driven>来实现注解处理器和适配器的开启。

4、谈谈你对 Spring 的理解

Spring 是一个开源框架，为简化企业级应用开发而生。Spring 可以是使简单的 JavaBean 实现以前只有 EJB 才能实现的功能。Spring 是一个 IOC 和 AOP 容器框架。

Spring 容器的主要核心是：

控制反转（IOC），传统的 java 开发模式中，当需要一个对象时，我们会自己使用 new 或者 getInstance 等直接或者间接调用构造方法创建一个对象。而在 spring 开发模式中，spring 容器使用了工厂模式为我们创建了所需要的对象，不需要我们自己创建了，直接调用 spring 提供的对象就可以了，这是控制反转的思想。

依赖注入（DI），spring 使用 javaBean 对象的 set 方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程，就是依赖注入的思想。

面向切面编程（AOP），在面向对象编程（oop）思想中，我们将事物纵向抽成一个个

的对象。而在面向切面编程中，我们将一个个的对象某些类似的方面横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用 CGLIB 方式实现动态代理。

5、Spring 中的设计模式

（1）单例模式——spring 中两种代理方式，若目标对象实现了若干接口，spring 使用 jdk 的 java.lang.reflect.Proxy 类代理。若目标兑现没有实现任何接口，spring 使用 CGLIB 库生成目标类的子类。单例模式——在 spring 的配置文件中设置 bean 默认为单例模式。
（2）模板方式模式——用来解决代码重复的问题。
比如：RestTemplate、JmsTemplate、JpaTemplate
（3）前端控制器模式——spring 提供了前端控制器 DispatherServlet 来对请求进行分发。
试图帮助（view helper）——spring 提供了一系列的 JSP 标签，高效宏来帮助将分散的代码整合在试图中。
（4）依赖注入——贯穿于 BeanFactory/ApplacationContext 接口的核心理念。
（5）工厂模式——在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用同一个接口来指向新创建的对象。Spring 中使用 beanFactory 来创建对象的实例。

6、简单介绍一下 Spring bean 的生命周期

（1）bean 定义：在配置文件里面用<bean></bean>来进行定义。
（2）bean 初始化：有两种方式初始化:
在配置文件中通过指定 init-method 属性来完成实现
org.springframwork.beans.factory.InitializingBean 接口
（3）bean 调用：有三种方式可以得到 bean 实例，并进行调用
（4）bean 销毁：销毁有两种方式
使用配置文件指定的 destroy-method 属性实现
org.springframwork.bean.factory.DisposeableBean

7、请描述一下 Spring 的事务

1、声明式事务管理的定义：用在 Spring 配置文件中声明式的处理事务来代替代码式的处理事务。这样的好处是，事务管理不侵入开发的组件，具体来说，业务逻辑对象就不会意识到正在事务管理之中，事实上也应该如此，因为事务管理是属于系统层面的服务，而不是业务逻辑的一部分，如果想要改变事务管理策划的话，也只需要在定义文件中重新配置即可，这样维护起来极其方便。

基于 TransactionInterceptor 的声明式事务管理：两个次要的属性：transactionManager，用来指定一个事务治理器， 并将具体事务相关的操作请托给它； 其他一个是 Properties 类型的 transactionAttributes 属性，该属性的每一个键值对中，键指定的是方法名，方法名可以行使通配符， 而值就是表现呼应方法的所运用的事务属性。

2、基于 @Transactional 的声明式事务管理：Spring 2.x 还引入了基于 Annotation

的体式格式，具体次要触及@Transactional 标注。@Transactional 可以浸染于接口、接口方法、类和类方法上。算作用于类上时，该类的一切 public 方法将都具有该类型的事务属性。

3、编程式事物管理的定义：在代码中显式挪用 beginTransaction()、commit()、rollback()等事务治理相关的方法， 这就是编程式事务管理。Spring 对事物的编程式管理有基于底层 API 的编程式管理和基于 TransactionTemplate 的编程式事务管理两种方式。

8、spring 的 aop 理解

9、Mybatis 中#和$的区别？

#相当于对数据 加上 双引号，$相当于直接显示数据

#将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user_id#，如果传入的值是 111,那么解析成 sql 时的值为 order by "111", 如果传入的值是 id，则解析成的 sql 为 order by "id".

$将传入的数据直接显示生成在 sql 中。如：order by $user_id$，如果传入的值是 111,那么解析成 sql 时的值为order by user_id, 如果传入的值是 id，则解析成的 sql 为 order by id.

#方式能够很大程度防止 sql 注入。$方式无法防止 Sql 注入。

$方式一般用于传入数据库对象，例如传入表名。一般能用#的就别用$.

10、Mybatis 的编程步骤是什么样的

1、创建 SqlSessionFactory
2、通过 SqlSessionFactory 创建 SqlSession
3、通过 sqlsession 执行数据库操作
4、调用 session.commit()提交事务
5、调用 session.close()关闭会话

11、Mybatis 中一级缓存与二级缓存？

一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空。

二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。作用域为 namespance 是指对该 namespance 对应的配置文件中所有的 select 操作结果都缓存，这样不同线程之间就可以共用二级缓存。启动二级缓存：在 mapper 配置文件中：<cache />。

二级缓存可以设置返回的缓存对象策略：<cache readOnly="true">。当 readOnly="true" 时，表示二级缓存返回给所有调用者同一个缓存对象实例，调用者可以 update 获取的缓存实例，但是这样可能会造成其他调用者出现数据不一致的情况（因为所有调用者调用的是同

一个实例）。当 readOnly="false"时，返回给调用者的是二级缓存总缓存对象的拷贝，即不同调用者获取的是缓存对象不同的实例，这样调用者对各自的缓存对象的修改不会影响到其他的调用者，即是安全的，所以默认是 readOnly="false";

对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

12、MyBatis 在 insert 插入操作时返回主键 ID

（1）数据库为 MySql 时：

```
<insert id="insert" parameterType="com.test.User" keyProperty="userId"
useGeneratedKeys="true" >
```

“keyProperty”表示返回的 id 要保存到对象的那个属性中，“useGeneratedKeys”表示主键 id 为自增长模式。MySQL 中做以上配置就 OK

（2）oracle

```
<insert id="insert" parameterType="com.test.User">
    <selectKey resultType="INTEGER" order="BEFORE" keyProperty="userId">
        SELECT SEQ_USER.NEXTVAL as userId from DUAL
    </selectKey>
    insert into user (user_id, user_name, modified, state) values
(#{userId,jdbcType=INTEGER}, #{userName,jdbcType=VARCHAR},
#{modified,jdbcType=TIMESTAMP}, #{state,jdbcType=INTEGER})
</insert>
```

八、分布式技术

1、Redis 的特点？

Redis 本质上是一个 Key-Value 类型的内存数据库，很像 memcached，整个 数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬盘上进行保存。因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value DB。

Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构，此外单个 value 的最大限制是 1GB，不像 memcached 只能保存 1MB 的数据，另外 Redis 也可以对存入的 Key-Value 设置 expire 时间。Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

2、为什么 redis 需要把所有数据放到内存中？

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重

影响 redis 的性能。在内存越来越便宜的今天， redis 将会越来越受欢迎。如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

3、Redis 的持久化

RDB 持久化：该机制可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。

AOF 持久化：记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小

无持久化：让数据只在服务器运行时存在。

同时应用 AOF 和 RDB：当 Redis 重启时，它会优先使用 AOF 文件来还原数据集，因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。

扩展：

RDB 的优缺点：

优点：RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。RDB 在

恢复大数据集时的速度比 AOF 的恢复速度要快。

缺点：如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为 RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端；如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。

AOF 的优缺点。

优点：

1、使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。

2、Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写

后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。

缺点：

对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB

一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。

AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

4、ActiveMQ 如果消息发送失败怎么办？

Activemq 有两种通信方式，点到点形式和发布订阅模式。

如果是点到点模式的话，如果消息发送不成功此 消息默认会保存到 activemq 服务端知道有消费者将其消费，所以此时消息是不会丢失的。

如果是发布订阅模式的通信方式，默认情况下只通知一次，如果接收不到此消息就没有了。这种场景只适 用于对消息送达率要求不高的情况。如果要求消息必须送达不可以丢失的话，需要配置持久订阅。每个订阅端定义一个 id， 在订阅是向 activemq 注册。发布消息和接收消息时需要配置发送模式为持久化。此时 如果客户端接收不到消息，消息会持久化到服务端，直到客户端正常接收后为止。

5、如何使用 ActiveMQ 解决分布式事务？

在互联网应用中，基本都会有用户注册的功能。在注册的同时，我们会做出如下操作：收集用户录入信息，保存到数据库向用户的手机或邮箱发送验证码等等…

如果是传统的集中式架构，实现这个功能非常简单：开启一个本地事务，往本地数据库中插入一条用户数据，发送验证码，提交事物。

但是在分布式架构中，用户和发送验证码是两个独立的服务，它们都有各自的数据库，那么就不能通过本地事物保证操作的原子性。这时我们就需要用到 ActiveMQ（消息队列）来为我们实现这个需求。

在用户进行注册操作的时候，我们为该操作创建一条消息，当用户信息保存成功时，把这条消息发送到消息队列。验证码系统会监听消息，一旦接受到消息，就会给该用户发送验证码。

扩展：

1.如何防止消息重复发送？

解决方法很简单：增加消息状态表。通俗来说就是一个账本，用来记录消息的处理状态，每次处理消息之前，都去状态表中查询一次，如果已经有相同的消息存在，那么不处理，可

以防止重复发送。

6、Dubbo 的连接方式有哪些？

Dubbo 的客户端和服务端有三种连接方式，分别是：广播，直连和使用 zookeeper 注册中心。

7、Dubbo 的容错机制有哪些。

1、Failover Cluster 模 式

失败自动切换，当出现失败，重试其它服务器。(默认) 2）Failfast Cluster

快速失败，只发起一次调用，失败立即报错。 通常用于非幂等性的写操作，比如新增记录。

2、Failsafe Cluster

失败安全，出现异常时，直接忽略。 通常用于写入审计日志等操作。

3、Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。 通常用于消息通知操作。

4、Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks=”2”来设置最大并行数。

5、Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0 开始支持) 通常用于通知所有提供者更新缓存或日志等本地资源信息。

总结： 在实际应用中查询语句容错策略建议使用默认 Failover Cluster ，而增删改建议使用 Failfast Cluster 或者 使用 Failover Cluster（retries=”0”） 策略 防止出现数据 重复添加等等其它问题！建议在设计接口时候把查询接口方法单独做一个接口提供查询。

8、使 用 dubbo 遇到过哪些问题？

（1）增加提供服务版本号和消费服务版本号

这个具体来说不算是一个问题,而是一种问题的解决方案,在我们的实际工作中会面临各种环境资源短缺的问题,也是很实际的问题,刚开始我们还可以提供一个服务进行相关的开发和测试,但是当有多个环境多个版本,多个任务的时候就不满足我们的需求,这时候我们可以通过给提供方增加版本的方式来区分.这样能够剩下很多的物理资源,同时为今后更换接口定义发布在线时，可不停机发布，使用版本号.

引用只会找相应版本的服务,例如：

<dubbo:serviceinterface=“com.xxx.XxxService”ref=“xxxService” version=“1.0” />

<dubbo:referenceid=“xxxService”interface=“com.xxx.XxxService” version=“1.0”/>

（2）dubbo reference 注解问题

@Reference 只能在 springbean 实例对应的当前类中使用，暂时无法在父类使用；如果确实要在父类声明一个引用，可通过配置文件配置 dubbo:reference，然后在需要引用的地方跟引用 springbean 一样就可以了.

（3）出现 RpcException: No provider available for remote service 异常，表示没有可用的服务提供者

1、检查连接的注册中心是否正确

2、到注册中心查看相应的服务提供者是否存在

3、检查服务提供者是否正常运行

（4）服务提供者没挂，但在注册中心里看不到

首先，确认服务提供者是否连接了正确的注册中心，不只是检查配置中的注册中心地址，而且要检查实际的网络连接。

其次，看服务提供者是否非常繁忙，比如压力测试，以至于没有 CPU 片段向注册中心发送心跳，这种情况，减小压力，将自动恢复。

9、什么是 ElasticSearch？

Elasticsearch 是一个基于 Lucene 的搜索引擎。它提供了具有 HTTP Web 界面和无架构 JSON 文档的分布式，多租户能力的全文搜索引擎。Elasticsearch 是用 Java 开发的，根据 Apache 许可条款作为开源发布。

10、Elasticsearch 中的倒排索引是什么？

倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索。倒排索引是一种像数据结构一样的散列图，可将用户从单词导向文档或网页。它是搜索引擎的核心。其主要目标是快速搜索从数百万文件中查找数据。

11. ZooKeeper 是什么？

ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 保证了如下分布式一致性特性：

顺序一致性

原子性

单一视图

可靠性

实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 zxid（Zookeeper Transaction Id）。而读请求只会相对

于更新有序，也就是读请求的返回结果中会带有这个 zookeeper 最新的 zxid。

12、Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。

工作机制：

客户端注册 watcher

服务端处理 watcher

客户端回调 watcher

Watcher 特性总结：

一次性

无论是服务端还是客户端，一旦一个 Watcher 被触发，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

客户端串行执行

客户端 Watcher 回调的过程是一个串行同步的过程。

轻量

Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。

watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。

注册 watcher getData、exists、getChildren

触发 watcher create、delete、setData

当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

九、Git

1、reset 与 rebase, pull 与 fetch 的区别

git reset 不修改 commit 相关的东西，只会去修改.git 目录下的东西。

git rebase 会试图修改你已经 commit 的东西，比如覆盖 commit 的历史等，但是不能使用

rebase 来修改已经 push 过的内容，容易出现兼容性问题。rebase 还可以来解决内容的冲突，解决两个人修改了同一份内容，然后失败的问题。
git pull pull=fetch+merge,
使用 git fetch 是取回远端更新，不会对本地执行 merge 操作，不会去动你的本地的内容。pull 会更新你本地代码到服务器上对应分支的最新版本

2、git merge 和 git rebase 的区别

git merge 把本地代码和已经取得的远程仓库代码合并。
git rebase 是复位基底的意思，gitmerge 会生成一个新的节点，之前的提交会分开显示，而 rebase 操作不会生成新的操作，将两个分支融合成一个线性的提交。

3、git 如何解决代码冲突

git stash
git pull
git stash pop
这个操作就是把自己修改的代码隐藏，然后把远程仓库的代码拉下来，然后把自己隐藏的修改的代码释放出来，让 gie 自动合并。
如果要代码库的文件完全覆盖本地版本。
git reset －hard
git pull

十一、电商项目问题

1、项目参与人

项目经理：1 人
开发组 A：
组长：1
前端：1
后台：3
测试：1
产品：1
开发组 B：
组长：1
前端：1
后台：3
测试：1
产品：1

运维：1

2、项目周期

6 个月，两个小组并行开发，一个小组 3 个服务

3、数据量

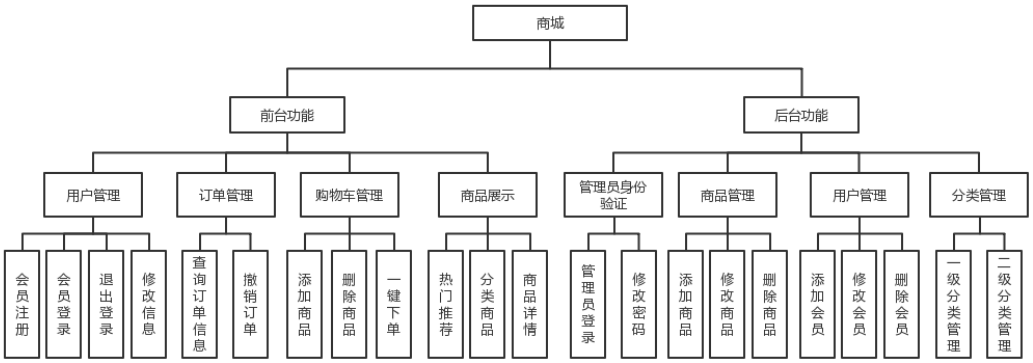
表：60
业务表：45
开发者使用表：10
用户量：3000
数据量：100000

4、系统吞度量要素

（1）QPS（TPS）：每秒钟 request/事务 数量
（2）并发数： 系统同时处理的 request/事务数
（3）响应时间： 一般取平均响应时间
QPS（TPS）= 并发数/平均响应时间或者并发数 = QPS*平均响应时间
一个典型的上班签到系统，早上 8 点上班，7 点半到 8 点的 30 分钟的时间里用户会登录签到系统进行签到。公司员工为 1000 人，平均每个员上登录签到系统的时长为 5 分钟。可以用下面的方法计算。
QPS = 1000/(30*60) 事务/秒
平均响应时间为 = 5*60 秒
并发数= QPS*平均响应时间 = 1000/(30*60) *(5*60)=166.7
说明：
一个系统吞吐量通常由 QPS（TPS）、并发数两个因素决定，每套系统这两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其它消耗导致系统性能下降。
决定系统响应时间要素
我们做项目要排计划，可以多人同时并发做多项任务，也可以一个人或者多个人串行工作，始终会有一条关键路径，这条路径就是项目的工期。
系统一次调用的响应时间跟项目计划一样，也有一条关键路径，这个关键路径是就是系统影响时间；
关键路径是有 CPU 运算、IO、外部系统响应等等组成。

5、服务器数量

6、系统分多少个模块，你负责哪些



7、你常用哪些表，都有哪些字段

8、说说你最近做的这个项目的背景，简单的介绍一下你这个项目？

项目背景：

电商项目的背景一般是由市场推动的，比如行业竞争或者经营方式的改变(营销理念)。竞争的形态也发生了巨大的变化，从以产品、价格为主的竞争转向以服务为主的竞争，服务成为主导竞争格局的重要因素。渠道作为企业完成客户沟通、产品/服务交换过程以及实现价值、产生效益的重要载体，发挥了采集、传达客户和竞争对手等市场信息，为买卖双方提供便利，协调供需矛盾，为客户提供合适的产品与服务，向客户传递产品/服务信息，实现营销/服务目标等重要的功能。

XXX 商城之前主要以实体店为主，进行批发与零售。业务也相对比较传统，为了提升业务绩效，增强客户满意度和粘性，另一方面，也为基于互联网的商务模式创新奠定基础。针对上述行业环境变化和业务战略目标，xxx 商城网上终端预约销售基础上，即将启动网上商城建设项目，用于建立网上终端、营销案在线销售及相关辅助功能，包含商品管理、订单管理、类目管理、客户管理、合作商管理、客服管理、购物平台、内容管理等，很大程度上分担了人工的压力，对提高客户服务效率和客户满意度能够起到较好的作用。基于此，XXX 公司提出建设网上商城建设项目工程。

项目介绍：

商城项目打造的是“社区+电商”的模式，用户不只是在社区中有自己的圈子，还可以将电商加入到社区中，整个电商网站实现的功能非常之多，采用分布式的架构设计，包括后台管理、前台系统、订单系统、单点登录系统、搜索系统、会员系统等。

①该项目是自己公司的产品，我们公司负责整个网站的运营，属于 B2C 平台；

②系统的用途，主要是提供 B2C 的平台，其中自营商品也有商家入住，类似天猫。

③系统架构，采用分布式的系统架构，其中前台系统和单点登录系统采用了集群的方式部署，在后台管理系统中采用了 Maven 的多模块化的管理，其中采用了水平切分的方式，将

pojo、dao、service、web 分层开发，这样做的好处就是可以重用性更高。

系统方面：

（1）系统内部接口调用采用 dubbo 完成个系统调用，并且使用 SpringBoot 发布，接口提供端采用 RESTful 方式的接口定义；

（2）系统之间的通知机制采用 MQ 的方式，使用 RabbitMQ 的实现，使用了 RabbitMQ 的消息订阅模式的消息机制；

（3）系统的接口还对 JS 的跨域做了支持，采用了 jsonp 的解决方法，在后台接口中扩展了 spirng 提供的 jackson 数据转化器实现；

部署方面：

采用了 Nginx+tomcat 的模式，其中 nginx 的作用一方面是做反向代理、负载均衡、另一方面是做图片等静态资源的服务器。

8、你承担这个项目的哪些核心模块？

（1）后台管理系统:主要实现商品管理、商品规格参数管理、订单管理、会员管理等、CMS(内容管理系统)等，并且提供了跨域支持；

（2）前台系统:主要是面向用户访问，使用 dubbo 和后台系统接口做交互，并且该系统在部署上采用集群的方式；

（3）单点登录系统:主要是提供集中用户登录凭证的集中解决方案，提供和用户信息相关的接口:比如说用户注册、查询等接口。

（4）订单系统:主要是提供和订单相关的业务接口，在订单系统了做了严格的数据校验以及高并发写的支持（这里可以说使用队列实现），并且使用了 Quartz 定时任务实现对订单的定时扫描，比如说关闭超时未付款的订单；

（5）搜索系统:主要是提供商品的搜索，采用开源企业级系统 ES 实现，采用了 MQ 机制保证了商品数据可以及时同步到 ES 中；

（6）会员系统:主要是维护用户的信息，已购买订单、优惠券、系统消息、修改密码、绑定手机等功能；

（7）缓存:主要是用 Redis 实现，并且对 Redis 做了集群来保证 Redis 服务的高可用。

（8）支付系统:，主要是负责订单的支付、对账等功能，主要是对接了支付宝的接口；

9、在做这个项目的时候你碰到了哪些问题？你是怎么解决的？

（1）开发 SpringBoot 接口出现客户端和服务端不同步，导致接口无法测试，产生的原因沟通不畅。

（2）订单提交时由于本地 bug 或者意外故障导致用户钱支付了但是订单不成功，采用对账方式来解决。

（3）上线的时候一定要把支付的假接口换成真接口。

（4）项目中用到了曾经没有用过的技术，解决方式：用自己的私人时间主动学习

（5）在开发过程中与测试人员产生一些问题，本地环境 ok 但是测试环境有问题，环境的问题产生的，浏览器环境差异，服务器之间的差异

（6）系统运行环境问题，有些问题是在开发环境下 OK，但是到了测试环境就问题，比如说系统文件路径问题、导出报表中的中文问题（报表采用 POI），需要在系统 jdk 中添加相应的中文字体才能解决；

10、你做完这个项目后有什么收获？

首先，在数据库方面，我现在是真正地体会到数据库的设计真的是一个程序或软件设计的重要和根基。因为数据库怎么设计，直接影响到一个程序或软件的功能的实现方法、性能和维护。由于我做的模块是要对数据库的数据进行计算和操作的，所以我对数据库的设计对程序的影响是深有体会，就是因为我们的数据库设计得不好，搞得我在对数据库中的数据进行获取和计算利润、总金时，非常困难，而且运行效率低，时间和空间的复杂也高，而且维护起来很困难，过了不久，即使自己有注释，但是也要认真地看自己的代码才能明白自己当初的想法和做法。加上师兄的解说，让我对数据库的重要的认识更深一层，数据库的设计真的是重中之重。

其次，就是分工的问题。虽然这次的项目我们没有在四人选出一个组长，但是，由于我跟其他人都比较熟，也有他们的号码，然后我就像一个小组长一样，也是我对他们进行了分工。俗话也说，分工合作，分好了工，才能合作。但是这次项目，我们的分工却非常糟糕，我们在分工之前分好了模块，每个模块实现什么功能，每个人负责哪些模块。本以为我们的分工是明确的，后来才发现，我们的分工是那么的一踏糊涂，一些功能上紧密相连的模块分给了两个人来完成，使两个人都感到迷惘，不知道自己要做什么，因为两个人做的东西差不多。我做的，他也在做，那我是否要继续做下去？总是有这样的疑问。从而导致了重复工作，浪费时间和精力，并打击了队员的激情，因为自己辛辛苦苦写的代码，最后可能没有派上用场。我也知道，没有一点经验的我犯这样的错是在所难免，我也不过多地怪责自己，吸取这次的教训就好。分工也是一门学问。

再者，就是命名规范的问题。可能我们以前都是自己一个人在写代码，写的代码都是给自己看的，所以我们都没有注意到这个问题。就像师兄说的那样，我们的代码看上去很上难看很不舒服，也不知道我们的变量是什么类型的，也不知道是要来做什么的。但是我觉得我们这一组人的代码都写得比较好看，每个人的代码都有注释和分隔，就是没有一个统一的规范，每个人都人自己的一个命名规则和习惯，也不能见名知义。还有就是没有定义好一些公共的部分，使每个人都有一个自己的“公共部分”，从而在拼起来时，第一件事，就是改名字。而这些都应该是在项目一开始，还没开始写代码时应该做的。

然后，我自己在计算时，竟然太大意算错了利润，这不能只一句我不小心就敷衍过去，也是我的责任，而且这也是我们的项目的核心部分，以后在做完一个模块后，一定要测试多次，不能过于随便地用一个数据测试一下，能成功就算了，要用可能出现的所有情况去测试程序，让所有的代码都有运行过一次，确认无误。

最后，也是我比较喜欢的东西，就是大家一起为了一个问题去讨论和去交流。因为我觉得，无论是谁，他能想的东西都是有限的，别人总会想到一些自己想不到的地方。跟他人讨论和交流能知道别人的想法、了解别人是怎样想一个问题的，对于同样的问题自己又是怎样想的，是别人的想法好，还是自己的想法好，好在什么地方。因为我发现问题的能力比较欠缺，所以我也总是喜欢别人问我问题，也喜欢跟别人去讨论一个问题，因为他们帮我发现了我自己没有发现的问题。在这次项目中，我跟植荣的讨论就最多了，很多时候都是不可开交的那种，不过我觉得他总是能够想到很多我想不到的东西，他想的东西也比我深入很多，虽然很多时候我们好像闹得很僵，但是我们还是很要好的!嘻嘻！而且在以后的学习和做项目

的过程中，我们遇到的问题可能会多很多，复杂很多，我们一个人也不能解决，或者是没有想法，但是懂得与他人讨论与交流就不怕这个问题，总有人的想法会给我们带来一片新天地。相信我能做得更好。

还有就是做项目时要抓准客户的要求，不要自以为是，自己觉得这样好，那样好就把客户的需求改变，项目就是项目，就要根据客户的要求来完成。

10、你们这个项目中订单 ID 是怎么生成的？我们公司最近打算做一个电商项目，如果让你设计这块，你会考虑哪些问题？

生成订单 ID 的目的是为了使订单不重复，本系统订单 ID 生成规则：
用户 ID+当前系统的时间戳

String orderId = order.getUserId() + "" + System.currentTimeMillis();

设计的时候我会考虑：
（1）订单 ID 不能重复
（2）订单 ID 尽可能的短（占用存储空间少，实际使用方便，客服相关）
（3）订单 ID 要求是全数字（客服）

11、多台 tomcat 之间的 session 是怎么同步的？

不用 session，我们使用单点登陆，使用 redis，存在 redis，生成，A 同步到 B，B 同步到 C。

13、如何解决并发问题的？

集群，负载均衡，nginx(主备，一般主在工作，备闲置；资源浪费)，lvs(在 2 个 Nginx 前做一个拦截，接收后进行分工)。有问题，如果 nginx 挂掉，整个系统就挂了。可以主备解决，可以前面搭一个 lvs。这块不是你做的，但是你知道怎么解决(非常复杂，但是必须了解。针对具体的情况去具体对待，CPU，内存，不要一刀切。)

14、你们生产环境的服务器有多少台？

面试前要数好，一般是十几到二十台。(用在哪里？这是重点)
Nginx 至少 2 台
Tomcat 至少 3 台以上
数据库至少 2 台
Redis 至少一台

15、备份是怎么做的？有没有做读写分离？

主从(一主多从，主要是备份主)，每天备份，备份的文件不要放到数据库服务器上，可以 FTP。要检查有效否。读写分离自己查一下，分库分表做过。

16、你们使用什么做支付的？如果使用支付宝做支付，请求超时了怎么处理？

（1）重试，一般三次，每次重试都要停顿一会，比如，以第一次停顿 1 秒，第二次停顿 2 秒，第三次停顿 3 秒；
（2）给订单标识付款异常状态，并且发出警告(邮件、短信)给相关人员。
（3）写个定时任务，定时处理异常状态的订单。

17、如果付款后支付宝没有返回，或者返回超时了，但是钱又已经扣了，你怎么办？

（1）我们请求了支付宝，但是没有接受到响应，我们就认为该订单没有支付成功，并且将订单标识为异常状态；
（2）使用定时任务处理；
（3）做一个对账的任务，实时处理异常状态的订单。

18、购物车和订单库存是怎么控制的