

猿灯塔，做程序员的引导者

www.vuandenta.com

字节一面（40分钟）

一面问基础

1. 平时做的项目属于什么业务
2. Java Object类有哪些方法，分别作用

1. 类构造器

是创建Java对象的途径之一，通过new关键字调用构造器完成对象的实例化，或通过构造器对象进行相应的初始化。在JDK的Object类源码中，系统会自动添加一个无参构造器。

```
1 public Object(){  
2 Object obj =new Object();    //构造一个Object类的对象
```

2. registerNatives方法

```
1 private static native void registerNatives();  
2 static {  
3     registerNatives();  
4 }
```

静态代码块是一个类在初始化过程中必定会执行的内容，所以在类加载时会执行该方法，通过该方法来注册本地方法。

3. getClass方法

```
1 public final native Class<?> getClass();
```

首先，该方法由final声明本地方法，不能被重写，作用是返回运行时类对象，通过这个类对象可以获取该运行时类的相关属性和方法。

class是一个类的属性，能获取该类编译时的类对象；而getClass()是一个类的方法，它是获取该类运行时的类对象。

4. hashCode方法

```
1 public native int hashCode()
```

这是一个由native声明的本地方法，作用是返回对象的哈希码（是int类型的数值）。

哈希算法也称为散列算法，是将数据依据特定算法产生的结果直接指定到一个地址上，这个结果就是由hashCode方法产生的。当集合要添加新元素时，会先调用这个元素的hashCode方法，就能定位到它应该放置的物理位置上。

1) 如果这个位置上没有元素，就直接存储在这个位置上，不用再进行任何比较；

2) 如果这个位置上已经有元素，就调用它的equals方法与新元素进行比较，相同的话就不存了；不同的话（也就是产生了hash冲突），那么就在这个key的位置产生一个链表，将所有hashCode相同的对象存放在这个单链表上。

如果两个对象的equals值比较相等，那么它们的hashCode值一定相等；如果两个对象的equals值比较不相等，那么他们的hashCode值可能相等，也可能不相等。

5. equals方法

```
1 public boolean equals(Object obj) {  
2     return (this == obj);  
3 }
```

5.1 Object中的equals方法比较的是对象的地址是否相同；equals方法可以被重写，重写后equals方法比较的是两个对象值是否相同。

5.2 在Java规范中，对equals方法的使用必须遵循以下几个规则：

自反性：对于任何非空引用值X，X.equals(X)都应返回true；

对称性：对于任何非空引用值X和Y，当且仅当Y.equals(X)返回true时，X.equals(Y)也应该返回true；

传递性：对于任何非空引用值X，Y，Z，如果X.equals(Y)返回true，并且Y.equals(Z)返回true，那么X.equals(Z)应返回true；

一致性：对于任何非空引用值X和Y，多次调用X.equals(Y)始终返回true或始终返回false。

5.3 equals和==的区别？

equals比较的是两个对象值是否相等，如果没有被重写，比较的是对象的引用地址是否相同；

==用于比较基本数据类型的值是否相等，或比较两个对象的引用地址是否相等；

```
1 String hello = new String("hello");  
2 String hello1 = new String("hello");  
3 System.out.println(hello.equals(hello1));    //重写了,比较的是值,输出结果为true  
4 System.out.println(hello == hello1);    //比较的是引用地址,输出结果为false  
5 //比较基本类型的值  
6 int age = 10;  
7 int age2 = 10;  
8 System.out.println(age == age2);    //输出为true
```

6. clone方法

猿灯塔，做程序员的引导者

www.vuandenata.com

```
1 protected native Object clone() throws CloneNotSupportedException;
```

clone方法是创建并返回一个对象复制后的结果。

如果一个类没有实现Cloneable接口（只是一个标记接口），那么对此类对象进行赋值时，会出现CloneNotSupportedException异常。

clone生成的新对象与原对象的关系，区别在于两个对象间是否存在相同的引用或对应的内存地址是否存在共用情况；若存在，则为“浅复制”，否则为“深复制”，“深复制”时需要将共同关联的引用也复制完全。

7.toString方法

```
1 public String toString() {
2     return getClass().getName() + "@" + Integer.toHexString(hashCode());
3 }
```

打印某个对象时，默认是调用toString()方法。

比如System.out.print(person)等价于System.out.print(person.toString()); //默认返回对象的地址

getClass().getName是返回对象的全类名，Integer.toHexString(hashCode())是以16进制无符号整数形式返回此哈希码的字符串表示形式。

8.notify方法

```
1 public final native void notify();
```

唤醒可能等待该对象的对象锁的其他线程。由JVM（与优先级无关）随机挑选一个处于wait状态的线程。

在调用notify()之前，线程必须获取该对象的对象锁，执行完notify()方法后，不会马上释放锁，直到退出synchronized代码块，当前线程才会释放锁；notify一次只能随机通知一个线程进行唤醒。

9.notifyAll方法

```
1 public final native void notifyAll();
```

使所有正在等待池中等待同一个共享资源的全部线程从等待状态退出，进入可运行状态，让它们同时竞争对象锁，只有获得锁的线程才能进入就绪状态。

10.wait(long timeout)方法

```
1 public final native void wait(long timeout) throws InterruptedException;
```

11.wait(long timeout,int nanos)方法

```
1 1 public final void wait(long timeout, int nanos) throws InterruptedException {
2 2     if (timeout < 0) {
3 3         throw new IllegalArgumentException("timeout value is negative");
4 4     }
5 5
6 6     if (nanos < 0 || nanos > 999999) {
7 7         throw new IllegalArgumentException(
8 8             "nanosecond timeout value out of range");
9 9     }
10 10
11 11     if (nanos > 0) {
12 12         timeout++;
13 13     }
14 14
15 15     wait(timeout);
16 16 }
```

wait(long timeout, int nanos)方法的实现只要nanos大于0，那么timeout时间就加上一毫秒，主要是更精确地控制时间，其他的跟wait(long timeout)一样。

12.wait方法

```
1 public final void wait() throws InterruptedException {
2     wait(0);
3 }
```

wait方法会引起当前线程阻塞，直到另外一个线程在对应的对象上调用notify或notifyAll方法，或达到了方法参数中指定的时间。

调用wait方法的当前线程一定要拥有对象的监视器锁。

wait方法会把当前线程放在对应的等待队列中，在这个对象上的所有同步请求都不会得到响应。线程调用将不会调用线程，线程一直处于休眠状态。要注意的是，wait方法把当前线程放置到这个对象的等待队列中，解锁也仅仅是在这个对象上；当前线程在等待过程中仍然持有其他对象的锁。

如果当前线程被其他线程在当前线程等待之前或正在等待时调用了interrupt()中断了，那么就会抛出InterruptedException异常。

猿灯塔，做程序员的引导者

www.vuandenta.com

12.1 为什么wait方法一般要写在while循环里？

答：在某个线程调用notify到等待线程被唤醒的过程中，有可能出现另一个线程得到了锁并修改了条件使得条件不再满足；只有某些等待线程的条件满足了，但通知线程调用了notifyAll有可能出现“伪唤醒”。

12.2 wait方法和sleep方法的区别？

答：wait方法属于Object类，当调用wait方法时，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify方法后本线程才会进入对象锁定池，准备获取对象锁进入运行状态。sleep方法属于Thread类，sleep方法导致程序暂停执行指定的时间，让出CPU给其他线程，但是它的监控状态依然保持，当指定的时间到了又会恢复运行状态。在调用sleep方法过程中，线程不会释放对象锁。

13. finalize方法

```
1 protected void finalize() throws Throwable { }
```

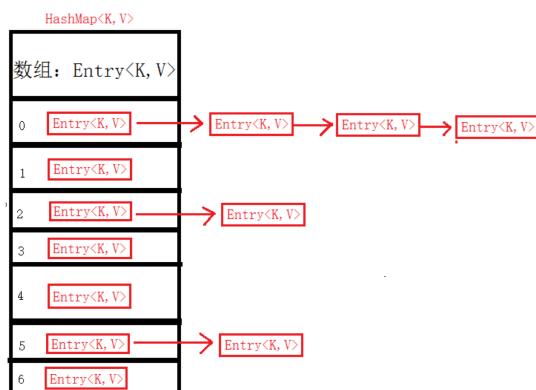
该方法用于垃圾回收，一般由JVM自动调用，不需要程序员手动调用该方法。

3. HashMap原理，线程安全？

a. HashMap简单说就是它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。

b. HashMap基于哈希表，底层结构由数组来实现，添加到集合中的元素以“key--value”形式保存到数组中，在数组中key--value被包装成一个实体来处理---也就是上面Map接口中的Entry。

c. 在HashMap中，Entry[]保存了集合中所有的键值对，当我们需要快速存储、获取、删除集合中的元素时，HashMap会根据hash算法来获得“键值对”在数组中存在的位置，以此来对应的操作方法。



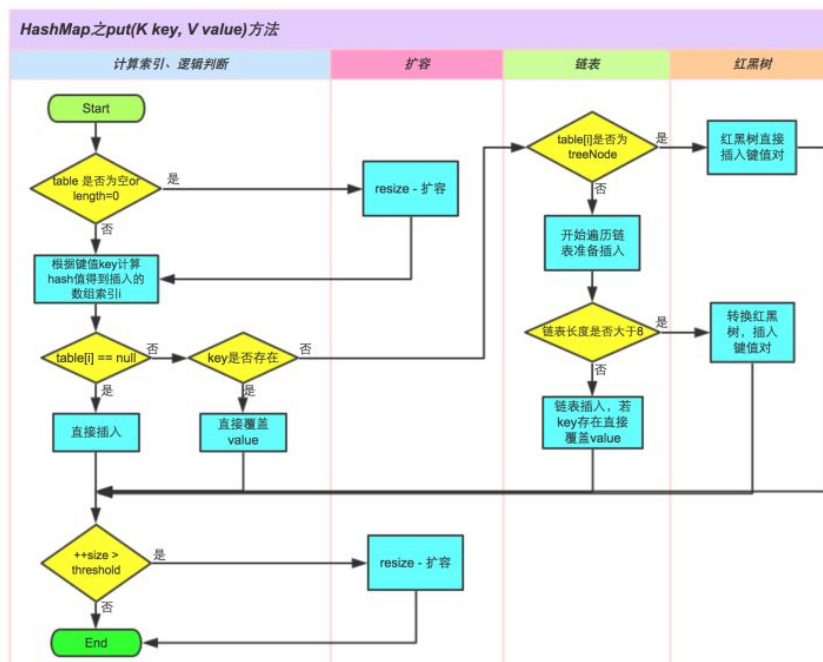
d. HashMap底层是采用数组来维护的Entry静态内部类的数组

```
1 /**
2  * The table, resized as necessary. Length MUST Always be a power of two.
3  */
4 transient Entry[] table;
5
6 static class Entry<K, V> implements Map.Entry<K, V> {
7     final K key;
8     V value;
9     Entry<K, V> next;
10    final int hash;
11    .....
12 }
```

c. HashMap添加元素：将准备增加到map中的对象与该位置上的对象进行比较(equals方法),如果相同,那么就将该位置上的那个对象(Entry类型)的value值替换掉,否则沿着该Entry的链继续重复上述过程,如果到链的最后任然没有找到与此对象相同的对象,那么这个时候就会被增加到数组中,将数组中该位置上的那个Entry对象链到该对象的后面(先hashcode计算位置，如果找到相同位置便替换值，找不到则重复hashcode计算，直到最后在添加到hashmap最后面；)

猿灯塔，做程序员的引导者

www.vuandenata.com



d. HashMap是基于哈希表的Map接口的非同步实现，允许null键值，但不保证映射的顺序；底层使用数组实现，数组中的每项是一个链表；存储时根据key的hash算法来决定其存储位置；数组扩容需要重新计算扩容后每个元素在数组中的位置很耗性能；

e. ConcurrentHashMap是HashMap的线程安全实现，允许多个修改操作同时进行(使用了锁分离技术)，它使用了多个锁来控制对hash表的不同段进行的修改，每个段其实就是一个小的hashtable，它们有自己的锁。使用了多个子hash表(段Segment)，允许多个读操作并发进行，读操作并不需要锁，因为它的HashEntry几乎是不可变的：

HashMap线程不安全：因为多线程环境下，使用HashMap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

如下代码：

```
1 final HashMap<String, String> map = new HashMap<String, String>(2);
2 Thread t = new Thread(new Runnable() {
3     @Override
4     public void run() {
5         for (int i = 0; i < 10000; i++) {
6             new Thread(new Runnable() {
7                 @Override
8                 public void run() {
9                     map.put(UUID.randomUUID().toString(), "");
10                }
11            }, "ftf" + i).start();
12        }
13    }
14 }, "ftf");
15 t.start();
16 t.join();
```

4. Java如何进行线程同步

- 使用synchronized关键字
- wait与notify
- volatile关键字

猿灯塔，做程序员的引导者

www.vuandenta.com

d. Lock

e. ThreadLocal类

详读这篇：https://blog.csdn.net/qq_41478279/article/details/88344460

5. CAS原理

CAS: Compare and Swap, 即比较再交换。

jdk5增加了并发包java.util.concurrent.*,其下面的类使用CAS算法实现了区别于synchronouse同步锁的一种乐观锁。

详读：<https://www.jianshu.com/p/ab2c8fce878b>

6. JVM垃圾回收之GC算法

1. 引用计数法(没有被java采用):

a. 原理: 对于一个对象A, 只要有任何一个对象引用了A, 则A的引用计数器就加1, 当引用失效时, 引用计数器就减1, 只要对象A的引用计数器的值为0, 则对象A就会被回收。

b. 问题:

- 引用和去引用伴随加法和减法, 影响性能;
- 很难处理循环引用。

2. 标记清除法:

a. 原理: 现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段: 标记阶段和清除阶段。一种可行的实现是, 在标记节点, 首先通过根节点, 标记所有从根节点开始的可达对象。因此, 未被标记的对象就是未被引用的垃圾对象。然后在清除阶段, 清除所有未被标记的对象。

b. 问题:

- 标记和清除两个过程效率不高, 产生内存碎片导致需要分配较大对象时无法找到足够的连续内存而需要触发一次GC操作。

3. 标记压缩法:

a. 原理: 适合用于存活对象较多的场合, 如老年代。它在标记-清除算法的基础上做了一些优化。标记阶段一样, 但之后, 将所有存活对象压缩到内存的一端。之后, 清除边界外所有的空间。

b. 优点:

- 解决了标记-清除算法导致的内存碎片问题和在存活率较高时复制算法效率低的问题。

4. 复制算法:

a. 原理: 将原有的内存空间分为两块, 每次只使用其中一块, 在垃圾回收时, 将正在使用的内存中的存活对象复制到未使用的内存块中, 之后清除正在使用的内存块中的所有对象, 交换两个内存的角色, 完成垃圾回收。

b. 问题:

- 不适用于存活对象比较多的场合, 如老年代。

5. 分代回收法:

a. 原理: 根据对象存活周期的不同将内存划分为几块, 一般是新生代和老年代, 新生代基本采用复制算法, 老年代采用标记整理算法。

7. Mysql索引原理以及查询优化:

索引的目的在于提高查询效率, 与我们查阅图书所用的目录是一个道理: 先定位到章, 然后定位到该章下的一个小节, 然后找到页数。相似的例子还有: 查字典, 查火车车次, 飞机航班等

本质都是: 通过不断地缩小想要获取数据的范围来筛选出最终想要的结果, 同时把随机的事件变成顺序的事件, 也就是说, 有了这种索引机制, 我们可以总是用同一种查找方式来锁定数据。

推荐阅读: <https://www.cnblogs.com/Eva-J/articles/10126413.html>

8. TCP, 拥塞控制

(1) 当主机开始发送数据时, 如果立即将较大的发送窗口的全部数据字节都注入网络, 由于不清楚网络的状况, 可能会引发网络拥塞

(2) 比较好的方式就是从小到大逐渐增大发送端的拥塞控制窗口数值

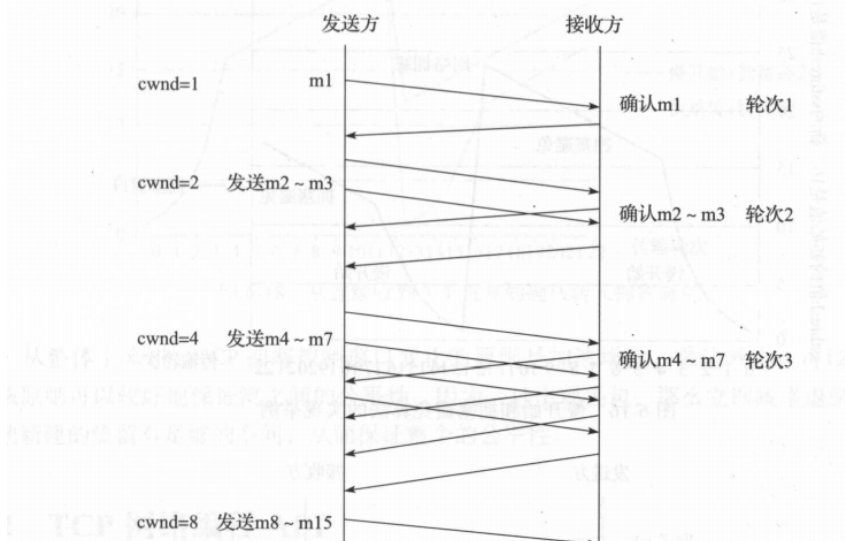
(3) 一开始发送报文段的时候就将拥塞窗口cwnd的打星设置为最大报文段的MSS。如果收到了对新报文段的确认, 那就就在大家一个MSS的数值

当rwnd朱勾搭的时候, 为防止拥塞窗口cwind引起网络拥塞, 就开始使用慢开始门限sssthresh进行控制, sssthresh的使用方法如下

猿灯塔，做程序员的引导者

www.vuandentata.com

当 $cwnd < ssthresh$ 时，使用慢开始算法。
当 $cwnd > ssthresh$ 时，改用拥塞避免算法。
当 $cwnd = ssthresh$ 时，慢开始与拥塞避免算法任意。



详读: <https://www.cnblogs.com/tijie/p/10574050.html>

9. 算法: 给定一棵二叉树, 找到这棵树中最最后一行中最左边的值。

<https://cloud.tencent.com/developer/article/1453637>

字节二面 (60分钟)

(二面感觉主要考察的就是代码能力, 基本一直在码)

1. 知道什么设计模式, 分别介绍

1、工厂方法模式 (利用创建同一接口的不同实例):

1、普通工厂模式: 建立一个工厂类, 对实现了同一接口的一些类进行实例的创建;

```
1 public class SendFactory {
2
3     public Sender produce(String type) {
4         if ("mail".equals(type)) {
5             return new MailSender();
6         } else if ("sms".equals(type)) {
7             return new SmsSender();
8         } else {
9             System.out.println("请输入正确的类型!");
10            return null;
11        }
12    }
13 }
```

2、多个工厂方法模式: 提供多个工厂方法, 分别创建对象;

```
1 public class SendFactory {
2
3     public Sender produceMail(){
4         return new MailSender();
5     }
6
7     public Sender produceSms(){
8         return new SmsSender();
9     }
10 }
```

猿灯塔，做程序员的引导者

www.vuandentata.com

```
10 }
```

3、静态工厂方法模式：将上面的多个工厂方法置为静态的，不需要创建工厂实例，直接调用即可；

4、适用场景：凡是出现了大量不同种类的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。

2、抽象工厂模式(多个工厂)：创建多个工厂类，提高工厂的扩展性，不用像上面一样如果增加产品则要去修改唯一的工厂类；

3、单例模式(保证对象只有一个实例)：保证在一个JVM中，该对象只有一个实例存在；

1、适用场景：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

2、代码：

```
1 public class Singleton {
2
3     /* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 */
4     private static Singleton instance = null;
5
6     /* 私有构造方法，防止被实例化 */
7     private Singleton() {
8     }
9
10    /* 静态工程方法，创建实例 */
11    public static Singleton getInstance() {
12        if (instance == null) {
13            instance = new Singleton();
14        }
15        return instance;
16    }
17
18    /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */
19    public Object readResolve() {
20        return instance;
21    }
22 }
```

3、分类：

1、饿汉式：类初始化时创建单例，线程安全，适用于单例占内存小的场景，否则推荐使用懒汉式延迟加载；

```
1 public class Singleton{
2     private static Singleton instance = new Singleton();
3     private Singleton(){}
4     public static Singleton newInstance(){
5         return instance;
6     }
7 }
```

2、懒汉式：需要创建单例实例的时候再创建，需要考虑线程安全(性能不太好)：

```
1 public class Singleton{
2     private static Singleton instance = null;
3     private Singleton(){}
4     public static synchronized Singleton newInstance(){
5         if(null == instance){
6             instance = new Singleton();
7         }
8         return instance;
9     }
10 }
```

猿灯塔，做程序员的引导者

www.vuandentata.com

```
9     }
10 }
```

3、**双重检验锁**：效率高；（解决问题：假如两个线程A、B，A执行了if (instance == null)语句，它会认为单例对象没有创建，此时线程切到B也执行了同样的语句，B也认为单例对象没有创建，然后两个线程依次执行同步代码块，并分别创建了一个单例对象。）

```
1 public class Singleton {
2     private static volatile Singleton instance = null; //volatile的一个语义是禁止指令重排序优化
3     private Singleton() {}
4     public static Singleton getInstance() {
5         if (instance == null) {
6             synchronized (Singleton.class) {
7                 if (instance == null) { //2
8                     instance = new Singleton();
9                 }
10            }
11        }
12        return instance;
13    }
14 }
```

4、**静态内部类方式**：可以同时保证延迟加载和线程安全。

```
1 public class Singleton{
2     private static class SingletonHolder{
3         public static Singleton instance = new Singleton();
4     }
5     private Singleton() {}
6     public static Singleton newInstance(){
7         return SingletonHolder.instance;
8     }
9 }
```

5、**枚举**：使用枚举除了线程安全和防止反射调用构造器之外，还提供了自动序列化机制，防止反序列化的时候创建新的对象。

```
1 public enum Singleton{
2     instance;
3     public void whateverMethod() {}
4 }
```

4、原型模式（对一个原型对象进行复制、克隆产生类似新对象）：将一个对象作为原型，对其进行复制、克隆，产生一个和元对象类似的新对象；

1、核心：它的核心是原型类Prototype，需要实现Cloneable接口，和重写Object类中的clone方法；

2、作用：使用原型模式创建对象比直接new一个对象在性能上要好的多，因为Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。

5、适配器模式（接口兼容）：将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。

1、类的适配器模式：

猿灯塔，做程序员的引导者

www.vuandentata.com

核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里，看代码：

```
[java] 01. public class Source {  
02.  
03.     public void method1() {  
04.         System.out.println("this is original method!");  
05.     }  
06. }
```

```
[java] 01. public interface Targetable {  
02.  
03.     /* 与原类中的方法相同 */  
04.     public void method1();  
05.  
06.     /* 新类的方法 */  
07.     public void method2();  
08. }
```

```
[java] 01. public class Adapter extends Source implements Targetable {  
02.  
03.     @Override  
04.     public void method2() {  
05.         System.out.println("this is the targetable method!");  
06.     }  
07. }
```

2、对象的适配器模式：

只需要修改Adapter类的源码即可：

```
[java] 01. public class Wrapper implements Targetable {  
02.  
03.     private Source source;  
04.  
05.     public Wrapper(Source source){  
06.         super();  
07.         this.source = source;  
08.     }  
09.     @Override  
10.     public void method2() {  
11.         System.out.println("this is the targetable method!");  
12.     }  
13.  
14.     @Override  
15.     public void method1() {  
16.         source.method1();  
17.     }  
18. }
```

3、接口的适配器模式：

猿灯塔，做程序员的引导者

www.vuandentata.com

这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

```
[java] 01. public interface Sourceable {  
02.  
03.     public void method1();  
04.     public void method2();  
05. }
```

抽象类Wrapper2:

```
[java] 01. public abstract class Wrapper2 implements Sourceable{  
02.  
03.     public void method1() {}  
04.     public void method2() {}  
05. }
```

```
[java] 01. public class SourceSub1 extends Wrapper2 {  
02.     public void method1() {  
03.         System.out.println("the sourceable interface's first Sub1!");  
04.     }  
05. }
```

```
[java] 01. public class SourceSub2 extends Wrapper2 {  
02.     public void method2() {  
03.         System.out.println("the sourceable interface's second Sub2!");  
04.     }  
05. }
```

4、使用场景：

1、类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

2、对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

3、接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

6、装饰模式 (给对象动态增加新功能，需持有对象实例)：装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例：

1、示例：

猿灯塔，做程序员的引导者

www.vuandenta.com

Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

```
[java]
01. public interface Sourceable {
02.     public void method();
03. }
```

```
[java]
01. public class Source implements Sourceable {
02.
03.     @Override
04.     public void method() {
05.         System.out.println("the original method!");
06.     }
07. }
```

```
[java]
01. public class Decorator implements Sourceable {
02.
03.     private Sourceable source;
04.
05.     public Decorator(Sourceable source){
06.         super();
07.         this.source = source;
08.     }
09.     @Override
10.     public void method() {
11.         System.out.println("before decorator!");
12.         source.method();
13.         System.out.println("after decorator!");
14.     }
15. }
```

2、使用场景：



- 1、需要扩展一个类的功能。
- 2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）



7、代理模式（持有被代理类的实例，进行操作前后控制）：采用一个代理类调用原有的方法，且对产生的结果进行控制。

猿灯塔，做程序员的引导者

www.vuandenata.com

```
01. public interface Sourceable {  
02.     public void method();  
03. }
```

```
[java]    
01. public class Source implements Sourceable {  
02.  
03.     @Override  
04.     public void method() {  
05.         System.out.println("the original method!");  
06.     }  
07. }
```

```
[java]    
01. public class Proxy implements Sourceable {  
02.  
03.     private Source source;  
04.     public Proxy() {  
05.         super();  
06.         this.source = new Source();  
07.     }  
08.     @Override  
09.     public void method() {  
10.         before();  
11.         source.method();  
12.         atfer();  
13.     }  
14.     private void atfer() {  
15.         System.out.println("after proxy!");  
16.     }  
17.     private void before() {  
18.         System.out.println("before proxy!");  
19.     }  
20. }
```

8、外观模式 (集合所有操作到一个类)：外观模式是为了解决类与类之间的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度。

猿灯塔，做程序员的引导者

www.vuandentata.com

```
public class Computer {
    private CPU cpu;
    private Memory memory;
    private Disk disk;

    public Computer() {
        cpu = new CPU();
        memory = new Memory();
        disk = new Disk();
    }

    public void startup() {
        System.out.println("start the computer!");
        cpu.startup();
        memory.startup();
        disk.startup();
        System.out.println("start computer finished!");
    }

    public void shutdown() {
        System.out.println("begin to close the computer!");
        cpu.shutdown();
        memory.shutdown();
        disk.shutdown();
        System.out.println("computer closed!");
    }
}
```

9、**桥接模式 (数据库驱动桥接)**：桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。

10、**组合模式 (部分整体模式)**：组合模式有时又叫部分-整体模式在处理类似树形结构的问题时比较方便。

11、**享元模式 (共享池、数据库连接池)**：享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，如数据库连接池；

12、**策略模式 (多种算法封装)**：策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口：

```
1 ICalculator cal = new Plus(); //ICalculator是统一接口，Plus是实现类(多个)
2 int result = cal.calculate(exp); //jvm根据实现类不同而调用不同实现类的方法
```

13、**模板方法模式 (抽象方法作为骨架，具体逻辑让子类实现)**：定义一个操作中算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变算法的结构即可重定义该算法中的某些特定步骤。完成公共动作和特殊动作的分离。

```
1 //题目：排序并打印：
2 abstract class AbstractSort {
3     /**
4      * 将数组array由小到大排序
5      * @param array
6      */
7     protected abstract void sort(int[] array);
8
9     public void showSortResult(int[] array){
10         System.out.print("排序结果："); //打印
11     }
12 }
13 //排序
14 class ConcreteSort extends AbstractSort {
15
16     @Override
17     protected void sort(int[] array){
18         for(int i=0; i<array.length-1; i++){
```

猿灯塔，做程序员的引导者

www.vuandentata.com

```
19         selectSort(array, i);
20     }
21 }
22
23 private void selectSort(int[] array, int index) {
24     //排序的实现逻辑
25 }
26 }
27 //测试
28 public class Client {
29     public static int[] a = { 10, 32, 1, 9, 5, 7, 12, 0, 4, 3 }; // 预设数据数组
30     public static void main(String[] args){
31         AbstractSort s = new ConcreteSort();
32         s.showSortResult(a);
33     }
34 }
```

14、观察者模式(发布-订阅模式)：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。类似于邮件订阅和RSS订阅，当你订阅了该文章，如果后续有更新，会及时通知你。

15、迭代器模式(遍历集合)：迭代器模式就是顺序访问聚集中的对象。

16、责任链模式(多任务形成一条链，请求在链上传递)：有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的调整。

17、命令模式(实现请求和执行的解耦)：命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

18、备忘录模式(保存和恢复对象状态)：主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象。

19、状态模式(对象状态改变时改变其行为)：当对象的状态改变时，同时改变其行为。状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。

```
/**
 * 状态模式的切换类    2012-12-1
 * @author erqing
 */
public class Context {

    private State state;

    public Context(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    public void setState(State state) {
        this.state = state;
    }

    public void method() {
        if (state.getValue().equals("state1")) {
            state.method1();
        } else if (state.getValue().equals("state2")) {
            state.method2();
        }
    }
}
```

猿灯塔，做程序员的引导者

www.vuandentata.com

20、访问者模式 (数据接口稳定，但算法易变)：访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到一个被访问者动态添加新的操作而无需做其它的修改的效果。

21、中介者模式：中介者模式也是用来降低类类之间的耦合的。如果使用中介者模式，只需关心和Mediator类的关系，具体类类之间的关系及调度交给Mediator就行，这有点像spring容器的作用。

22、解释器模式 (对于一些固定文法构建一个解释句子的解释器，如正则表达式)：解释器模式用来做各种各样的解释器，如正则表达式等的解释器。

23、建造者模式 (创建复合对象)：工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性

2. 手写单例 -> 线程安全的 -> 还可以怎么写

1、饿汉式：类初始化时创建单例，线程安全，适用于单例占内存小的场景，否则推荐使用懒汉式延迟加载：

```
1 public class Singleton{
2     private static Singleton instance = new Singleton();
3     private Singleton(){}
4     public static Singleton newInstance(){
5         return instance;
6     }
7 }
```

2、懒汉式：需要创建单例实例的时候再创建，需要考虑线程安全 (性能不太好)：

```
1 public class Singleton{
2     private static Singleton instance = null;
3     private Singleton(){}
4     public static synchronized Singleton newInstance(){
5         if(null == instance){
6             instance = new Singleton();
7         }
8         return instance;
9     }
10 }
```

3、双重检验锁：效率高； (解决问题：假如两个线程A、B，A执行了if (instance == null)语句，它会认为单例对象没有创建，此时线程切到B也执行了同样的语句，B也认为单例对象没有创建，然后两个线程依次执行同步代码块，并分别创建了一个单例对象。)

```
1 public class Singleton {
2     private static volatile Singleton instance = null; //volatile的一个语义是禁止指令重排序优化
3     private Singleton(){}
4     public static Singleton getInstance() {
5         if (instance == null) {
6             synchronized (Singleton.class) {
7                 if (instance == null) { //2
8                     instance = new Singleton();
9                 }
10            }
11        }
12        return instance;
13    }
14 }
```

4、静态内部类方式：可以同时保证延迟加载和线程安全。

```
1 public class Singleton{
2     private static class SingletonHolder{
3         public static Singleton instance = new Singleton();
4     }
5     private Singleton(){}
6     public static Singleton newInstance(){
```

猿灯塔，做程序员的引导者

www.vuandenata.com

```
7     return SingletonHolder.instance;
8 }
9 }
```

5、枚举：使用枚举除了线程安全和防止反射调用构造器之外，还提供了自动序列化机制，防止反序列化的时候创建新的对象。

```
1 public enum Singleton{
2     instance;
3     public void whateverMethod(){}
4 }
```

3. 算法：求无序数组中第k大的数（快速搜索）

从数组S中随机找出一个元素X，把数组分为两部分Sa和Sb。Sa中的元素大于等于X，Sb中元素小于X。这时有两种情况：

1. Sa中元素的个数小于k，则Sb中的第k-|Sa|个元素即为第k大数；
2. Sa中元素的个数大于等于k，则返回Sa中的第k大数。

时间复杂度近似为O(n)。

```
1 public class disorderSearchBin {
2
3     public static int quickSortOneTime(int[] array, int low, int high){ //一趟快速排序
4         int key = array[low];
5         while(low < high){
6             while(key < array[high] && low < high) high--;
7             array[low] = array[high];
8             while(key > array[low] && low < high) low++;
9             array[high] = array[low];
10        }
11        array[high] = key;
12        return high;
13    }
14
15    public static int Select_k(int[] array, int low, int high, int k) {
16        int index;
17        if(low == high) return array[low];
18        int partition = quickSortOneTime(array, low, high);
19        index = high - partition + 1; //找到的是第几个大值
20        if(index == k) {
21            return array[partition];
22        }else if(index < k) { //此时向左查找
23            return Select_k(array, low, partition-1, k-index); //查找的是相对位置的值，k在Z
24        }else {
25            return Select_k(array, partition+1, high, k);
26        }
27    }
28
29    public static void main(String[] args) {
30        // TODO Auto-generated method stub
31        int[] array = new int[] {92, 5, 88, 13, 80};
32        int index = Select_k(array, 0, array.length-1, 2);
33        System.out.print(index);
34    }
35
36 }
```

4. 算法：求旋转数组找最小值（二分法）

猿灯塔，做程序员的引导者

www.vuandenta.com

https://blog.csdn.net/qq_28081081/article/details/80751075

5. 算法：判断二叉树是否镜像（递归）

https://blog.csdn.net/qq_40550018/article/details/83721209

字节三面（40分钟）

三面感觉问的问题都比较开放

1. 你如何理解前后端分离

<https://blog.csdn.net/u012145252/article/details/81512228>

2. 有哪些后端开发经验，做了什么

这里介绍自己做的几个比较大的项目，最好是自己在项目中有主导地位的重点介绍一下。

3. 介绍HashMap，与TreeMap区别

HashMap通过hashCode对其内容进行快速查找，而 TreeMap中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用TreeMap（HashMap中元素的排列顺序是不固定的）。

HashMap 非线程安全 TreeMap 非线程安全。

4. 用HashMap实现一个有过期功能的缓存，怎么实现

使用ConcurrentHashMap

```
1 一、创建缓存类
2  package cache;
3
4  import java.util.Map;
5  import java.util.concurrent.ConcurrentHashMap;
6
7  public final class Cache {
8
9      /**
10       * 预缓存信息
11       */
12     private static final Map<String, Object> CACHE_MAP = new ConcurrentHashMap<String, Object>();
13
14     /**
15      * 每个缓存生效时间12小时
16      */
17     public static final long CACHE_HOLD_TIME_12H = 12 * 60 * 60 * 1000L;
18
19     /**
20      * 每个缓存生效时间24小时
21      */
22     public static final long CACHE_HOLD_TIME_24H = 24 * 60 * 60 * 1000L;
23
24     /**
25      * 存放一个缓存对象，默认保存时间12小时
26      * @param cacheName
27      * @param obj
28      */
29     public static void put(String cacheName, Object obj) {
30         put(cacheName, obj, CACHE_HOLD_TIME_12H);
31     }
32
33     /**
34      * 存放一个缓存对象，保存时间为holdTime
35      * @param cacheName
36      * @param obj
```

猿灯塔，做程序员的引导者

www.vuandenata.com

```
37     * @param holdTime
38     */
39     public static void put(String cacheName, Object obj, long holdTime) {
40         CACHE_MAP.put(cacheName, obj);
41         CACHE_MAP.put(cacheName + "_HoldTime", System.currentTimeMillis() + holdTime); //缓存失效时
42     }
43
44     /**
45     * 取出一个缓存对象
46     * @param cacheName
47     * @return
48     */
49     public static Object get(String cacheName) {
50         if (checkCacheName(cacheName)) {
51             return CACHE_MAP.get(cacheName);
52         }
53         return null;
54     }
55
56     /**
57     * 删除所有缓存
58     */
59     public static void removeAll() {
60         CACHE_MAP.clear();
61     }
62
63     /**
64     * 删除某个缓存
65     * @param cacheName
66     */
67     public static void remove(String cacheName) {
68         CACHE_MAP.remove(cacheName);
69         CACHE_MAP.remove(cacheName + "_HoldTime");
70     }
71
72     /**
73     * 检查缓存对象是否存在,
74     * 若不存在, 则返回false
75     * 若存在, 检查其是否已过有效期, 如果已经过了则删除该缓存并返回false
76     * @param cacheName
77     * @return
78     */
79     public static boolean checkCacheName(String cacheName) {
80         Long cacheHoldTime = (Long) CACHE_MAP.get(cacheName + "_HoldTime");
81         if (cacheHoldTime == null || cacheHoldTime == 0L) {
82             return false;
83         }
84         if (cacheHoldTime < System.currentTimeMillis()) {
85             remove(cacheName);
86             return false;
87         }
88         return true;
```

猿灯塔，做程序员的引导者

www.vuandenata.com

```
89     }
90 }
91
92
93
94 二、使用缓存方法
95 package cache;
96
97 import java.util.List;
98
99 import javax.annotation.Resource;
100
101 public class TestCache {
102
103     @Resource
104     ConfigDAO configDAO;
105
106     @SuppressWarnings("unchecked")
107     public List<String> getConfigCache() {
108         //缓存对象名称
109         String cacheName = "CONFIG_CACHE";
110
111         //从缓存中获取对象
112         List<String> list = (List<String>) Cache.get(cacheName);
113
114         //如果缓存中没有，则从数据库获取数据,并将获取的结果存入缓存中
115         if (list == null) {
116             list = configDAO.getAllConfig();
117             Cache.put(cacheName, list, Cache.CACHE_HOLD_TIME_24H); //加入缓存，有效期为24小时
118         }
119         return list;
120     }
121 }
122
```

5. 平时怎么学习新知识

- 看看书和博客文章：工作相关、兴趣爱好、技术热点；
- 观看新技术相关教学视频，系统性学习新技术；
- 动手实践并总结博客：类似课后题和课程设计，实践总结；
- 刷刷LeetCode算法题：提升算法能力和写代码的手感；

6. 最近看了什么书

自由发挥。