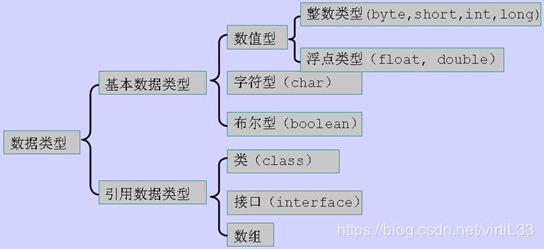
JAVA面试题整理

# 1 JAVA基础

## 1.1 JAVA类型



取值范围：如byte取 ｛-2^7，2^7-1｝ ;1B（byte，字节）= 8 bit

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 简单类型 | boolean | byte | char | short | Int | long | float | double |
| 二进制位数 | 1 | 8 | 16 | 16 | 32 | 64 | 32 | 64 |
| 封装器类 | Boolean | Byte | Character | Short | Integer | Long | Float | Double |

其中基本类型为值传递，引用类型为副本传递（String例外，其为不可变类型，所以类似于值传递）

## 1.2 final、finally、finalize

<https://www.jianshu.com/p/afaf54b9632e>

1、final可以修饰类,不能被继承  
   修饰方法,不能被重写  
   修饰变量,只能赋值一次  
    2、finally是try语句中的一个语句体,不能单独使用,用来释放资源  
    3、finalize是一个方法,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。finalize机制现在已经不推荐使用，并且在JDK 9开始被标记为deprecated。

## 1.3 Java的变量命名

1、首字母：英文字母、$和下划线。变量名：由$、字母、数字和下划线组成。

2、变量的命名遵循见名知义的原则。

3、用驼峰命名法命名多个单词组成的变量名。  ［比如： sumScore ］

4、变量名［方法名］首字母建议不用大写字母。   ［首字母大写一般是用来标识类名的］  －－ 看起来更规范而已

## 1.4 StringBuilder与StringBuffer StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串。

1、在执行速度方面的比较：StringBuilder > StringBuffer ；   
　　  2、他们都是字符串变量，是可改变的对象，每当我们用它们对字符串做操作时，实际上是在一个对象上操作的，不像String一样创建一些对象进行操作，所以速度快；   
　  　3、StringBuilder：线程非安全的；   
　　  4、StringBuffer：线程安全的；

## 1.5 Java集合

（<https://blog.csdn.net/zhangqunshuai/article/details/80660974>）

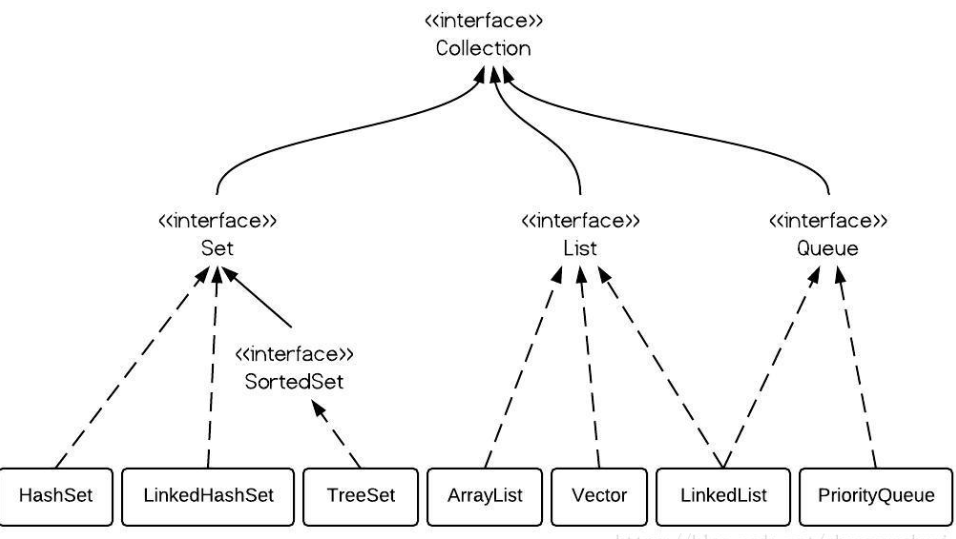
List , Set, Map都是接口，前两个继承至Collection接口，Map为独立接口

**Set**下有HashSet，LinkedHashSet，TreeSet

**List**下有ArrayList，Vector，LinkedList

**Map**下有Hashtable，LinkedHashMap，HashMap，TreeMap

**Collection**接口下还有个Queue接口，有PriorityQueue类



**1、Connection接口**:

— **List 有序,可重复**

**ArrayList**

优点: 底层数据结构是数组，查询快，增删慢。

缺点: 线程不安全，效率高

**Vector**

优点: 底层数据结构是数组，查询快，增删慢。

缺点: 线程安全，效率低

**LinkedList（**LinkedList就是基于双向循环链表设计的**）**

优点: 底层数据结构是链表，查询慢，增删快。

缺点: 线程不安全，效率高

—**Set 无序,唯一(SET**线程不安全**)**

**HashSet**

底层数据结构是哈希表。(无序,唯一)

如何来保证元素唯一性?

1.依赖两个方法：hashCode()和equals()

**LinkedHashSet**

底层数据结构是链表和哈希表。(FIFO插入有序,唯一)

1.由链表保证元素有序

2.由哈希表保证元素唯一

**TreeSet**

底层数据结构是红黑树。(唯一，有序)

1. 如何保证元素排序的呢?

自然排序

比较器排序

2.如何保证元素唯一性的呢?

根据比较的返回值是否是0来决定

1. **Map接口**

Map接口实现类，分别是HashMap、TreeMap、LinkedHashMap和HashTable。

**HashMap** 遍历时数据是随机的，允许一条记录的键是NULL ，允许多条记录的值是NULL。线程不安全。

**HashTable** 是hashMap的线程安全版，支持现成的同步，效率低，键和值都不能为NULL， ConcurrentHashMap线程安全，而且锁分离

**LinkHashMap** 保存了记录插入的顺序，用Iteraor遍历时先得到的肯定是先插入的，遍历时比hashmap慢，有HashMap的全部特性。线程不安全。

**TreeMap**实现的SortMap接口，能够把它保存的记录根据键排序，默认是升序（自然），也可以指定排序的比较器，用Iterator遍历TreeMap时，得到的记录是排过序的，不允许key空，非同步 TreeMap的实现原理是红黑树实现的。线程不安全。

HashMap效率较高，Hashtable效率较低。

父类不同：Hashtable的父类是Dictionary，HashMap的父类是AbstractMap

1. **Map底层与扩容**

<https://www.cnblogs.com/dassmeta/p/5338955.html>

**JDK1.7及之前：数组+链表**（数据结构有链表（单链表）和数组两种实现对数据的存储）

**JDK1.8：数组+链表+红黑树**

HashMap底层实现还是数组，只是数组的每一项都是一条链。其中参数initialCapacity就代表了该数组的长度。，每次新建一个HashMap时，都会初始化一个table数组。table数组的元素为Entry节点。

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

int hash;

}

 其中Entry为HashMap的内部类，它包含了键key、值value、下一个节点next，以及hash值，这是非常重要的，正是由于Entry才构成了table数组的项为链表。

**第一种**：使用默认构造方法初始化HashMap。从前文可以知道HashMap在一开始初始化的时候会返回一个空的table，并且thershold（阈值）为0。因此第一次扩容的容量为默认值DEFAULT\_INITIAL\_CAPACITY也就是16。同时threshold = DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR = 12。初始容量16，加载因子0.75

比如说：

HashMap m1 =new HashMap(）;

那么m1的一开始容量为0，阈值为0，插入数据后

容量变为16，阈值是16\*0.75=12

**第二种**：指定初始容量的构造方法初始化HashMap。初始容量会等于threshold，接着threshold = 当前的容量（threshold） \* DEFAULT\_LOAD\_FACTOR。

HashMap m2 =new HashMap(7);

那么他的初始化容量是7，阈值是7\*0.75，当然是整数

它第一次扩容，容量会变成7\*2 =14 ，但是hashmap的容量必须是2的整数倍

也就是扩容后容量会变为16

**第三种**：HashMap不是第一次扩容。如果HashMap已经扩容过的话，那么每次table的容量以及threshold量为原有的2倍。（前提是扩容后不超过最大容量）

这个就不举例了，很好理解，就是如果扩大两倍之后会超过最大容量，就变为2倍

## 1.6 多线程

### 16.1 线程的生命周期

**新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5种状态**。

重复启动线程或者重新激活生命周期已结束的线程会报异常。即重复调线程的start方法报异常IllegalThreadStateException。

### 16.2 线程结束几种方式

1、使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。

thread.exit = true; // 终止线程thread

thread.join();

2、 使用stop方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果）。

thread.stop();

3、使用interrupt方法中断线程。

（1）线程处于阻塞状态，如使用了sleep方法。

（2）使用while(!isInterrupted()){...}来判断线程是否被中断。

### 16.3 线程创建方式

1、继承Thread类实现多线程。见16-3-1。

2、实现Runnable接口，重写run方法（）。需要thread类辅助。见16-3-2。

3、实现Callable接口，重写call()方法。见16-3-3。

4、通过线程池启动多线程。

----**16-3-1**如下 --

public class MyThread extends Thread {

public MyThread() {

}

public void run() {

for(int i=0;i<10;i++) {

System.out.println(Thread.currentThread()+":"+i);

}

}

public static void main(String[] args) {

MyThread mThread1=new MyThread();

MyThread mThread2=new MyThread();

MyThread myThread3=new MyThread();

mThread1.start();

mThread2.start();

myThread3.start();

}

----**16-3-2**如下 --

public class MyThread implements Runnable{

public static int count=20;

public void run() {

while(count>0) {

try {

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+"-当前剩余票数:"+count--);

}

}

public static void main(String[] args) {

MyThread Thread1=new MyThread();

Thread mThread1=new Thread(Thread1,"线程1");

Thread mThread2=new Thread(Thread1,"线程2");

Thread mThread3=new Thread(Thread1,"线程3");

mThread1.start();

mThread2.start();

myThread3.start();

}

}

----**16-3-3**如下 --

public class MyThread implements Callable<String> {

private int count = 20;

@Override

public String call() throws Exception {

for (int i = count; i > 0; i--) { System.out.println(Thread.currentThread().getName()+"当前票数：" + i);

}

return "sale out";

}

public static void main(String[] args) throws InterruptedException, ExecutionException {

Callable<String> callable =new MyThread();

FutureTask <String>futureTask=new FutureTask<>(callable);

Thread mThread=new Thread(futureTask);

Thread mThread2=new Thread(futureTask);

Thread mThread3=new Thread(futureTask);

// mThread.setName("hhh");

mThread.start();

mThread2.start();

mThread3.start();

System.out.println(futureTask.get());

}

### \*16.4 线程冲突等产生于解决

### \*16.5 线程同步

### \*16.6 Thread常用方法

**Thread类构造方法：**

1.Thread（）；

2.Thread（String name）；

3.Thread（Runable r）；

4.Thread（Runable r, String name）；  
**常用方法：**

start();//启动线程

getId();//获得线程ID

getName();//获得线程名字

getPriority();//获得优先权

isAlive();//判断线程是否活动

isDaemon();//判断是否守护线程

getState();//获得线程状态

sleep(long mill);//休眠线程

join();//等待线程结束

yield();//放弃cpu使用权利

interrupt();//中断线程

currentThread();//获得正在执行的线程对象

### \*16.6 线程阻塞（什么时候达到5中状态）

### **\*16.7 sleep和wait的区别**

1、sleep是Thread的静态方法，wait是Object的方法，任何对象实例都能调用。  
2、sleep不会释放锁，它也不需要占用锁。wait会释放锁，针对此对象调用notify()或者notifyAll()方法后本线程才进入对象锁定池准备，但调用它的前提是当前线程占有锁(即代码要在synchronized中)。  
3、它们都可以被interrupted方法中断。

## 1.7 线程池

<https://www.cnblogs.com/williamjie/p/9485723.html>

风险：死锁、资源不足、线程泄漏、并发错误，请求过载；

## 1.8 JDBC

public class JdbcUtil {  
 private static final String *driver* = *getValue*("jdbc.driver");  
 private static final String *url* = *getValue*("jdbc.url");  
 private static final String *user* = *getValue*("jdbc.username");  
 private static final String *password* = *getValue*("jdbc.password");  
 static{  
 // 加载驱动  
 try {  
 Class.*forName*(*driver*);  
 } catch (ClassNotFoundException e) {  
 System.*out*.println("加载驱动失败！"+e.getMessage());  
 }  
 }  
  
 */\*\*  
  \* 连接数据库对象  
  \* @return Connection 连接对象  
  \*/* public static Connection getConnection(){  
 Connection conn = null;  
 try {  
 conn = DriverManager.*getConnection*(*url*, *user*, *password*);  
 } catch (SQLException e) {  
 System.*out*.println("数据库连接失败！"+e.getMessage());  
 }  
 return conn;  
 }  
  
 */\*\*获取文件中的数据库连接配置信息  
  \* @param key 键  
  \* @return 值  
  \*/* private static String getValue(String key){  
 //方式一：  
 //ResourceBundle bundle = ResourceBundle.getBundle("jdbc");  
 //return bundle.getString(key);  
 InputStream in = JdbcUtil.class.getResourceAsStream("/jdbc.properties");  
 Properties properties = new Properties();  
 try {  
 properties.load(in);  
 } catch (IOException e) {  
 // *TODO Auto-generated catch block* e.printStackTrace();  
 }  
 return properties.getProperty(key);  
 }  
  
 */\*\*  
  \* 关闭连接  
  \* @param conn 连接对象  
  \* @param stmt 创建sql语句对象  
  \* @param rs 执行sql语句对象  
  \*/* public static void close(Connection conn , Statement stmt, ResultSet rs){  
 try {  
 if(rs!=null){  
 rs.close();  
 }  
 if(stmt!=null){  
 stmt.close();  
 }  
 if(conn!=null){  
 conn.close();  
 }  
 } catch (SQLException e) {  
 System.*out*.println("关闭资源失败！"+e.getMessage());  
 }  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(*getConnection*());  
 }  
  
}

**Dao**

      public class Dao {  
   */\*\*  
         \*更新指定的员工 信息 --预编译  
         \* @param emp 员工信息  
         \*/*        public void update(Employee emp) throws DataAccessExcption{  
            Connection conn = JdbcUtil.*getConnection*();  
            StringBuffer sql = new StringBuffer("update myemp set ename=? hiredate=?  where empno=?");  
            PreparedStatement ps = null;  
            try {  
                 ps = conn.prepareStatement(sql.toString());  
                 int i = 1;  
                 //setObject(i++, 值);  
                 ps.setObject(i++, emp.getEname());  
                 //加sql  
                 ps.setObject(i++, new java.sql.Date(emp.getHiredate().getTime()));  
                 ps.setObject(i++, emp.getEmpno());  
                 System.*out*.println(sql.toString());  
                 //不要sql  
                 int row = ps.executeUpdate();  
                 System.*out*.println("更新成功！");       
            } catch (SQLException e) {  
                throw new DataAccessExcption();  
            }finally{  
                try {  
                    ps.close();  
                } catch (SQLException e) {  
                    // *TODO Auto-generated catch block*                    e.printStackTrace();  
                }  
                JdbcUtil.*close*(conn,null, null);  
            }          
            }  
  
          
        */\*\*  
         \* 员工名做模糊查询--非预编译  
         \* @param likeName  
         \* @return     Employee 员工信息  
         \*/*        public List<Employee> findLike(String likeName){  
            Connection conn = JdbcUtil.*getConnection*();  
            Statement stmt = null;  
            ResultSet rs ;  
            Employee emp = null;  
            String sql = "select \* from myemp where ename like "+"'%"+likeName+"%'";  
            System.*out*.println(sql);  
            List<Employee> list = new ArrayList<Employee>();  
            try {  
                 stmt = conn.createStatement();  
                 rs = stmt.executeQuery(sql);  
                 while(rs.next()){  
                        Integer empno = rs.getInt("empno");  
                        String ename = rs.getString("ename");  
                        String job = rs.getString("job");  
                        Integer mgr = rs.getInt("mgr");  
                        Date hiredate = rs.getDate("hiredate");  
                        Double sal = rs.getDouble("sal");  
                        Double comm = rs.getDouble("comm");  
                        Integer deptno = rs.getInt("deptno");  
  
                        emp = new Employee(empno, ename, job, mgr, hiredate,  
                                sal, comm, deptno);  
                        list.add(emp);  
                 }  
                   
            } catch (SQLException e) {  
                System.*out*.println("没有此员工！");  
                return null;      
            }  
            return list;      
        }  
          
        public static void main(String[] args) {  
            Scanner input=new Scanner(System.*in*);//创建一个键盘扫描类对象  
            System.*out*.print("请您输入内容:");  
            int contents=input.nextInt(); //输入整型  
            String contents1=input.next(); //输入字符串型  
 }  
}

## 1.9 常见分页语句

## 1.10 重写与重载

**override（重写）**

　　 1、方法名、参数、返回值相同。

　　 2、子类方法不能缩小父类方法的访问权限。

　　 3、子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。

　　 4、存在于父类和子类之间。

　　 5、方法被定义为final不能被重写。

**overload（重载）**

　　1、参数类型、个数、顺序至少有一个不相同。

　　2、不能重载只有返回值不同的方法名。

3、存在于父类和子类、同类中。

## 1.11 int与String互转

public class StringConvertInt {  
 public static void main(String[] args) {  
 Scanner input = new Scanner(System.*in*);  
 int a = input.nextInt();  
 String str = String.*valueOf*(a);  
 System.*out*.println("str = [" + str + "]");  
 str = Integer.*toString*(a);  
 System.*out*.println("str = [" + str + "]");  
  
 String Strq = "111";  
 int b = Integer.*valueOf*(111);  
 System.*out*.println("b = [" + b + "]");  
 }  
}

## 1.11 Integer的==与equals()

public static void main(String[] args) {  
 Integer a = 127;  
 Integer b = 127;  
 System.*out*.println("a==b:"+(a==b));//true  
 System.*out*.println("a.equals(b):"+(a.equals(b)));//true  
  
 a = 128;  
 b = 128;  
 System.*out*.println("a==b:"+(a==b));//Integer内部缓存到127，大于127会重新new对象，==比较结果 false  
 System.*out*.println("a.equals(b):"+(a.equals(b)));//true  
  
 a = new Integer(127);  
 b = 127;  
 System.*out*.println("a==b:"+(a==b));//创建了新对象，==比较结果 false  
 System.*out*.println("a.equals(b):"+(a.equals(b)));//true  
  
}

## 1.12 数组的length属性，字符串的length()方法，集合size()方法

## 1.13 GB2312转UTF-8

# String str = new String(str.getBytes("GB2312"),"UTF-8");

## 1.14 Date、Calendar与LocalDateTime

Date：java.util.Date包，包含日期，时间，毫秒数。

　　Calendar：java.util.Calendar包，abstract修饰，Date的很多方法已经过时迁移到了Calendar类上。

LocalDate/LocalDateTime：java.time.LocalDate/java.time.LocalDateTime包，这个类是不可变的和线程安全的。使用equals方法比较 。

## 1.15 Collection与Collections

1、区别

Collection，是单列集合的接口，有子接口List和Set

Collections，是针对集合操作的工具类，其中包含对集合进行排序和二分查找的方法

2、Collections常用方法

public static <T> void sort (List<T> list)：排序，默认情况下是自然排序

public static <T> int binarySearch (List<?> list, T key)：二分查找

public static <T> T max (Collection<?> coll)：最大值（最小值类似用法）

public static void reverse (List<?> list) ：顺序反转

public static void shuffle (List<?> list)：随机置换

## 1.16 Object类常用方法

Object类有12个成员方法，按照用途可以分为以下几种   
1，构造函数   
2，hashCode和equale函数用来判断对象是否相同,   
3，wait(),wait(long),wait(long ,int),notify(),notifyAll()   
4，toString()和getClass,   
5，clone()   
6，finalize()用于在垃圾回收

## 1.17 单一职责

单一职责原则的基本思想是各司其职，即每个模块，类或者方法只承担单个职责，避免多个职责交叉，从而导致修改其中一个职责的时候，影响另外一个职责。

**单一职责的优点：**

降低类的复杂度，一个类只负责一项职责。

提高类的可读性，可维护性。

降低变更引起的风险。

通常情况下，我们应该遵守单一职责原则，只有逻辑足够简单，才可以在代码级别违反单一职责原则：只有当类中方法数量足够少，可以在方法级别保持单一职责原则。

## 1.18 设计模式

**1、单例模式**

<https://www.runoob.com/design-pattern/singleton-pattern.html>

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

### **（1）懒汉式，线程不安全（线程安全需要加**synchronized **）**

是否 Lazy 初始化：是

是否多线程安全：否

实现难度：易

描述：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。  
这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

## 实例

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

### （2）**饿汉式**

是否 Lazy 初始化：否

是否多线程安全：是

实现难度：易

描述：这种方式比较常用，但容易产生垃圾对象。  
优点：没有加锁，执行效率会提高。  
缺点：类加载时就初始化，浪费内存。  
它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

实例

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() {

return instance;

}

}

### **（3）双检锁/双重校验锁（DCL，即 double-checked locking）**

JDK 版本：JDK1.5 起

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。  
getInstance() 的性能对应用程序很关键。

实例

public class Singleton {

private volatile static Singleton singleton; private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null) { synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton(); }

}

} return singleton;

} }

# 1. 封装、继承、多态

**封装**：隐藏对象的属性和实现细节，对外提供公共的访问方式，以防止数据的随意访问和修改。

**继承**：通过扩展一个已有的类，并继承该类的属性和行为，来创建一个新的类。

**多态**：  
三个条件：  
1.满足继承关系  
2.父类引用变量指向子类对象  
3.子类重写父类的方法

子类转换成父类时的规则:

将一个父类的引用指向一个子类的对象,称为向上转型(upcastiog),自动进行类型转换.

此时通过父类引用调用的方法是子类覆盖或继承父类的方法,不是父类的方法.

此时通过父类引用变量无法调用子类特有的方法.

如果父类要调用子类的特有方法就得将一个指向子类对象的父类引用赋给一个子类的引用,称为向下转型,此时必须进行强制类型转换.（用子类对象来接收一个指向子类对象的父类引用）

## 1.19 怎么创建一个不可变类

要创建不可变类，要实现下面几个步骤：

1. 将类声明为final，所以它不能被继承
2. 将所有的成员声明为私有的，这样就不允许直接访问这些成员
3. 对变量不要提供setter方法
4. 将所有可变的成员声明为final，这样只能对它们赋值一次
5. 通过构造器初始化所有成员，进行深拷贝(deep copy)
6. 在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝
7. 为了理解第5和第6条，我将使用FinalClassExample来阐明。

* 代码实例

public final class FinalClassExample {

private final int id;

private final String name;

private final HashMap testMap;

public int getId() {

return id;

}

public String getName() {

return name;

}

/\*\*

\* 可变对象的访问方法

\*/

public HashMap getTestMap() {

//return testMap;

return (HashMap) testMap.clone();

}

/\*\*

\* 实现深拷贝(deep copy)的构造器

\* @param i

\* @param n

\* @param hm

\*/

public FinalClassExample(int i, String n, HashMap hm){

System.out.println("Performing Deep Copy for Object initialization");

this.id=i;

this.name=n;

HashMap tempMap=new HashMap();

String key;

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

while(it.hasNext()){

key=it.next();

tempMap.put(key, hm.get(key));

}

this.testMap=tempMap;

}

/\*\*

\* 实现浅拷贝(shallow copy)的构造器

\* @param i

\* @param n

\* @param hm

\*/

/\*\*

public FinalClassExample(int i, String n, HashMap hm){

System.out.println("Performing Shallow Copy for Object initialization");

this.id=i;

this.name=n;

this.testMap=hm;

}

\*/

/\*\*

\* 测试浅拷贝的结果

\* 为了创建不可变类，要使用深拷贝

\* @param args

\*/

public static void main(String[] args) {

HashMap h1 = new HashMap();

h1.put("1", "first");

h1.put("2", "second");

String s = "original";

int i=10;

FinalClassExample ce = new FinalClassExample(i,s,h1);

//Lets see whether its copy by field or reference

System.out.println(s==ce.getName());

System.out.println(h1 == ce.getTestMap());

//print the ce values

System.out.println("ce id:"+ce.getId());

System.out.println("ce name:"+ce.getName());

System.out.println("ce testMap:"+ce.getTestMap());

//change the local variable values

i=20;

s="modified";

h1.put("3", "third");

//print the values again

System.out.println("ce id after local variable change:"+ce.getId());

System.out.println("ce name after local variable change:"+ce.getName());

System.out.println("ce testMap after local variable change:"+ce.getTestMap());

HashMap hmTest = ce.getTestMap();

hmTest.put("4", "new");

System.out.println("ce testMap after changing variable from accessor methods:"+ce.getTestMap());

}

}

## 1.20构造函数为何不能用abstract, static, final修饰

不同于方法，构造器不能是abstract, static, final的.  
1.构造器不是通过继承得到的，所以没有必要把它声明为final的。  
2.同理，一个抽象的构造器将永远不会被实现，所以它也不能声明为abstract的。  
3.构造器总是关联一个对象而被调用，所以把它声明为static是没有意义的。

## 1.21 catch里面可以做什么处理

## 1.22 存在抽象静态方法吗？

抽象方法是需要继承来实现,而静态方法根本不能被继承,两者互相矛盾.  
所以,根本就不存在抽象静态方法.

## 1. 抽象方法与抽象类

1.含有抽象方法的类一定是抽象类，抽象类不一定含有抽象方法

2.由于抽象方法会被子类重写，所以abstract关键字不能和final,static,private关键字配合使用。

## 1. 抽象类与接口

| **参数** | **抽象类** | **接口** |
| --- | --- | --- |
| 默认的方法实现 | 它可以有默认的方法实现 | 接口完全是抽象的。它根本不存在方法的实现 |
| 实现 | 子类使用 extends 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。 | 子类使用关键字 implements 来实现接口。它需要提供接口中所有声明的方法的实现 |
| 构造器 | 抽象类可以有构造器 | 接口不能有构造器 |
| 与正常 Java 类的区别 | 除了你不能实例化抽象类之外，它和普通Java类没有任何区别 | 接口是完全不同的类型 |
| 访问修饰符 | 抽象方法可以有 public、protected 和 default 这些修饰符 | 接口方法默认修饰符是 public。你不可以使用其它修饰符。 |
| main 方法 | 抽象方法可以有 main 方法并且我们可以运行它 | 接口没有 main 方法，因此我们不能运行它。 |
| 多继承 | 抽象方法可以继承一个类和实现多个接口 | 接口只可以继承一个或多个其它接口 |
| 速度 | 它比接口速度要快 | 接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。 |
| 添加新方法 | 如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。 | 如果你往接口中添加方法，那么你必须改变实现该接口的类。 |

# 1. IOC/DI 控制反转与AOP面向切面

IOC   inversion of control  控制反转

DI   Dependency Injection  依赖注入

**IoC控制反转**（IoC，Inversion of Control）--反射机制，降低耦合

    是一个概念，是一种思想。控制反转就是对对象控制权的转移，从程序代码本身反转到了外部容器。把对象的创建、初始化、销毁等工作交给spring容器来做。由spring容器控制对象的生命周期。即是**将new 的过程交给spring容器去处理**

　　(例如：早上去早餐店买了一个包子，包子是早餐店做的，你通过钱买下了包子，包子的决定权就从早餐店转移到了你的手上)

**DI依赖注入**：

依赖注入DI是指程序运行过程中，若需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部容器，由外部容器创建后传递给程序。依赖注入是目前最优秀的解耦方式。依赖注入让Spring的Bean之间以配置文件的方式组织在一起，而不是以硬编码的方式耦合在一起的。

**AOP面向切面 --**动态代理设计模式

AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。AOP是OOP的延续，是（Aspect Oriented Programming）的缩写，意思是面向切面（方面）编程。

主要的功能是：让关注点代码与业务代码分离，如日志记录，性能统计，安全控制，事务处理，异常处理等等。

**关注点：**重复代码就叫做关注点；

**切面：**关注点形成的类，就叫切面(类)！面向切面编程，就是指对很多功能都有的重复的代码抽取，再在运行的时候网业务方法上动态植入“切面类代码”。

**切入点：**执行目标对象方法，动态植入切面代码。可以通过切入点表达式，指定拦截哪些类的哪些方法；给指定的类在运行的时候植入切面类代码。

1.23 new String();

底层是一个char[];

# 2 关系型数据库（MYSQL为主）

## 2.1sql查询

### 2.1.1查询s\_id重复数量大于等于2（复用2.1.2）

SELECT s.s\_id  
FROM score s  
GROUP BY s.s\_id  
HAVING COUNT(1)>=2

### 2.1.2成绩1大于成绩2的所有学生

CREATE TABLE `score` (  
 `c\_id` int(11) NOT NULL COMMENT '课程id',  
 `s\_id` int(11) NOT NULL COMMENT '学号',  
 `grade` decimal(5,2) NOT NULL COMMENT '分数',  
 `creator` int(11) NOT NULL COMMENT '创建时间',  
 `create\_date` datetime NOT NULL COMMENT '创建时间',  
 `editor\_date` int(11) NOT NULL COMMENT '创建时间',  
 `update\_date` datetime NOT NULL COMMENT '更新时间',  
 PRIMARY KEY (`c\_id`,`s\_id`) USING BTREE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='成绩表';  
  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (1, 1001, 80.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (1, 1002, 80.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (1, 1003, 60.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (2, 1001, 90.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (2, 1002, 75.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (2, 1003, 75.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (3, 1001, 90.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (3, 1002, 76.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');  
 INSERT INTO `score`(`c\_id`, `s\_id`, `grade`, `creator`, `create\_date`, `editor\_date`, `update\_date`) VALUES (3, 1003, 60.00, 1, '2020-04-26 20:57:53', 1, '2020-04-26 20:57:57');

--- 答案

SELECT s1.s\_id  
FROM score s1,score s2  
WHERE s1.s\_id=s2.s\_id and s1.c\_id='1'  
AND s2.c\_id='2' AND s1.grade>s2.grade

### 删除除去最小id（或者保留最新时间数据）的所有name数据。

### 每门课程大于80分的学生

### 平均分大于75的所有学生

## 2.2 sql优化

执行计划与索引

**2.3sql基础**

### 2.3.1事务特性

### 2.3.2事务隔离性

### 2.3.3 char、varchar、varchar2区别

Mysql 5中  
非空CHAR的最大总长度是255【字节】；非空VARCHAR的最大总长度是65533【字节】。  
可空CHAR的最大总长度是254【字节】；可空VARCHAR的最大总长度是65532【字节】。  
原因：非空标记需要占据一个字节，VARCHAR超过255需要用2个字节标记字段长度，不超过255用1个字节标记字段长度.

char是定长的，varchar是变长的。varchar2（在ORACLE varchar对于汉字占两个字节，对于英文是一个字节，占的内存小，varchar2都是占两个字节。varchar对空串不处理，varchar2将空串当做null来处理。）应该是varchar的升级，只有ORACLE才有，这里不作讨论。

char 定长存储，速度快，但是存在一定的空间浪费，适用于字段不是很大，对速度要求高的场合。速度快是因为其在物理上是按定长存储的，这样，就可以根据偏移址一次取出固定长度的字符。

varchar 变长存储，效率不如char。varchar在存储时，在物理上要先存储该字段的实际长度，然后才是内容。这样读取的时候，就要读取两次，一次读它的长度，然后才是内容。所以它的访问速度会比char慢一些。但它可以节省空间。

### 2.3.4常用类型与长度

### 2.3.5 mysql引擎的区别

### 2.3.5 mysql底层用什么存

<https://www.csdn.net/gather_2e/MtTaIgwsNjU1Ny1ibG9n.html>

mysql 的数据其实是存储在磁盘中的。当往mysql存数据时，mysql是将表存在磁盘文件里的。而需要读取的时候，mysql 再从磁盘中将数据读取到内存中。这就涉及到磁盘和内存的交互IO

为了降低IO的花费，mysql 采用了B+数做数据库的数据库的索引以降低对磁盘的访问次数。

### 2.3.5 mysql连接数与并发

# 3 非关系型数据库

# 4 反向代理

# 5 分布式

# 6 搜索引擎ES与Solar

# 7 消息队列

KAFKA

rabbitMQ不支持数据库的持久化，只支持内存以及文件持久化

# 8 数据结构算法

## 8.1 排序

## 8.2 树

## 8.3 图

8.4 怎么快速找到一亿个文件（每个文件里面1w个单词）中哪些含有特定字段的所有文件列表。

# 9 SPRING

## 9.1 Mybatis

### 9.1.1mybatis里#和$的区别，及其优缺点。

答：#{} 占位符，预编译，对应变量会自动加单引号,能防止sql注入。

${} 拼接符，直接编译，对应变量不会加单引号，有sql注入风险；

9.1.1 like

## 9.2 MVC

### 9.2.1简述你对对MVC结构的理解和实践

9.3 spring常用注解

9.4 springboot

注解与启动

# 10 JVM

# 11 计算机网络

## 11.1网络层级

## 11.2 http与https

# 12 hadoop

**13 linux**