# 基础篇

## 抽象类。

1. 抽象类不能被实例化，只有抽象类的非抽象子类可以创建对象。

2. 抽象类中的抽象方法只是声明，不包含方法体。

3. 构造方法，类方法（用 static 修饰的方法）不能声明为抽象方法。

4. 抽象类的子类必须给出抽象类中的抽象方法的具体实现，除非该子类也是抽象类。

## 接口。

1．Java接口本身没有任何实现，因为Java接口不涉及表象，而只描述public行为，所以Java接口比Java抽象类更抽象化。

2．一个类可以实现多个接口。

## 为什么多接口，少继承,以及他们的区别？

1．继承的第一个危害：继承不分青红皂白的把父类的公有和受保护的方法统统继承下来。如果你的子类没有对一些方法重写，就会对你的子类产生危害。

2．Java不像C++支持多继承，所以Java可以通过实现接口来弥补这个局限。

接口是公开的，里面不能有私有的方法或变量，是用于让别人使用的，而抽象类是可以有私有方法或私有变量的，

3．另外，实现接口的一定要实现接口里定义的所有方法，而实现抽象类可以有选择地重写需要用到的方法，一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现。还有，接口可以实现多重继承，而一个类只能继承一个超类，但可以通过继承多个接口实现多重继承，接口还有标识（里面没有任何方法，如 Remote 接口）和数据共享（里面的变量全是常量）的作用.

## final、finally和finalize。

final：修饰类、方法和变量。

finally：try catch finally。即使在try里return，也是return后执行finally。

finalize：作为确保某些非内存资源(如Socket、文件等)释放的一个补充。

## GC是什么? 为什么要有GC。

清除不必要的内存；提高生产率不需要管理内存问题，帮助程序的完成性避免jvm的崩溃，但是加大了程序的负担。分为新生代、老年代和永久代。

## String s = new String(xyz);创建了几个String Object?

首先在字符串常量缓冲区，找到？不创建string对象，否则创建，这样就一个string对象，遇到new运算符号了（一定是新地址），在内存上创建string对象，并将其返回给s，又一个对象。

字符串池：存唯一的字符串  
 堆：对象（内存）、存储数据  
 栈：对象的引用、基本类型、程序指令  
 String s=new String("abc")创建了几个对象？  
 abc不存在于字符串池则创建。  
 然后new String（“abc”）在堆中建立对象。  
 最后用栈里的s指向堆里的对象。

需要注意就是jdk8，在处理字符串相加，直接做了处理，编译后直接加在了一起

## private、default、protected和public。

类、包、子类和外部。

## 八种基本类型和对应的封装类。

byte、char 、short、int 、long、float、double 和boolean。

Byte、Character、Short、Integer、Long、Float、Double和Boolean。

## 类的初始化顺序。

（静态变量、静态初始化块）>（变量、初始化块）>构造器。

## 简述 synchronized 和 java.util.concurrent.locks.Lock的异同。

### 主要相同点：Lock 能完成 synchronized 所实现的所有功能

### 主要不同点：Lock 有比 synchronized 更精确的线程语义和更好的性能。synchronized 会自动释放锁，而 Lock 一定要求程序员手工释放，并且必须在 finally 从句中释放。Lock 还有更强大的功能，例如，它的 tryLock 方法可以非阻塞方式去拿锁。而synchronized，多个线程访问一个synchronize修饰的方式的时，如果这个方法被占用，则线程会一直尝试获取锁，占用和浪费系统资源甚至会死锁造成程序中断。

## newInstance()方法和new关键字的异同。

使用getDeclaredConstructor()代替newInstance()。

newInstance():弱类型,效率低,只能调用无参构造

new():强类型,高效率,能调用任何public构造器

使用new关键字时你得保证import了这个驱动类才行，而使用class.formname时你只要保证引用了驱动的jar包就行了。

如果用new的话，当工程改变了数据库，你得修改.java文件和引入相应的驱动包。

而使用class.formname时可以将驱动的类配置在xml文件中。加载驱动对象时修改xml和引入jar包就行。

## 并发编程经常遇到的三个问题。

### 原子性问题：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。只要读取和赋值是才是原子操作。

### 可见性问题：可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。Java提供了volatile关键字来保证可见性，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

有序性问题：程序执行的顺序按照代码的先后顺序执行。

## 浅拷贝和深拷贝。

### 浅拷贝：Object类是类结构的根类，其中有一个方法为protected Object clone() throws [CloneNotSupportedException](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/CloneNotSupportedException.html)，这个方法就是进行的浅拷贝。有了这个浅拷贝模板，我们可以通过调用clone()方法来实现对象的浅拷贝。但是需要注意：1、Object类虽然有这个方法，但是这个方法是受保护的（被protected修饰），所以我们无法直接使用。2、使用clone方法的类必须实现Cloneable接口，否则会抛出异常CloneNotSupportedException。对于这两点，我们的解决方法是，在要使用clone方法的类中重写clone()方法，通过super.clone()调用Object类中的原clone方法。

### 深拷贝：与通过重写clone方法实现浅拷贝的基本思路一样，只需要为对象图的每一层的每一个对象都实现Cloneable接口并重写clone方法，最后在最顶层的类的重写的clone方法中调用所有的clone方法即可实现深拷贝。简单的说就是：每一层的每个对象都进行浅拷贝=深拷贝。

## 四舍五入。

BigDecimal.ROUND\_UP;远于0。

ROUND\_DOWN;趋向0。

ROUND\_CEILING;趋向正无穷。

ROUND\_FLOOR;趋向负无穷。

ROUND\_HALF\_UP;四入五入。

ROUND\_HALF\_DOWN;四舍五入。

ROUND\_HALF\_EVEN;银行家算法，舍去位后不为零进，否则看前一位，奇进偶退。

## 内部类

成员内部类

可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）。

当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，需要以下面的形式进行访问：“外部类.this.成员变量”。

在外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象。

成员内部类是依附外部类而存在的，也就是说，如果要创建成员内部类的对象，前提是必须存在一个外部类的对象。

局部内部类

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。

匿名内部类

一般使用匿名内部类的方法来编写事件监听代码。同样的，匿名内部类也是不能有访问修饰符和 static 修饰符的。匿名内部类是唯一一种没有构造器的类。正因为其没有构造器，所以匿名内部类的使用范围非常有限，大部分匿名内部类用于接口回调。匿名内部类在编译的时候由系统自动起名为 Outter$1.class。一般来说，匿名内部类用于继承其他类或是实现接口，并不需要增加额外的方法，只是对继承方法的实现或是重写。

静态内部类：

成员内部类加了static关键字。

## 对象的比较

Comparable接口-java.lang：

实现int compareTo（T o）方法，可以通过 Collections.sort（和 Arrays.sort）进行自动排序。

Comparator接口-java.util：

实现int compare（T o1，T o2）单独一个类实现。(o1, o2) -> o2.age-o1.age。

## 不同系统的编码格式如何传输String

String s=new String(str.getBytes(“iso8859-9”),”utf-8”);

## Volatile

1:虚拟机在代码编译优化的时候可能会改变代码的顺序,在不影响结果的前提下,会优先运行耗时短的,但是虽然没有对变量的值发生影响,但是可能会出现线程的安全,就是说在多线程代码的顺序虽然对本线程无影响,但是对其他线程会造成影响.volatile关键字就是为了避免优化改变顺序的,会保证之前的代码一定比自己先执行,后面的代码一定在自己之后.当然volatile并不能保证线程的安全,毋庸置疑.

2:jvm为了性能优化每个线程都有自己的工作内存,并且是拷贝于主内存的,加了volatile会保证变量每次都从主内存取值.

3:volatile并不是原子操作!!!!!!!!!

## 注解

用@Retention(RetentionPolicy.CLASS)修饰的注解，表示注解的信息被保留在class文件(字节码文件)中当程序编译时，但不会被虚拟机读取在运行的时候；  
用@Retention(RetentionPolicy.SOURCE )修饰的注解,表示注解的信息会被编译器抛弃，不会留在class文件中，注解的信息只会留在源文件中；  
用@Retention(RetentionPolicy.RUNTIME )修饰的注解，表示注解的信息被保留在class文件(字节码文件)中当程序编译时，会被虚拟机保留在运行时，

@interface注解

@Target({ElementType.TYPE})：用于描述注解的使用范围

1.CONSTRUCTOR:用于描述构造器  
　　2.FIELD:用于描述域  
　　3.LOCAL\_VARIABLE:用于描述局部变量  
　　4.METHOD:用于描述方法  
　　5.PACKAGE:用于描述包  
　　6.PARAMETER:用于描述参数  
　　7.TYPE:用于描述类、接口(包括注解类型) 或enum声明  
 @Retention(RetentionPolicy.RUNTIME):该Annotation被保留的时间长短

SOURCE:仅在源文件中有效,会被编译器抛弃，不会留在class文件中；  
　　2.CLASS:在class文件中有效,被保留在class文件(字节码文件)中当程序编译时，但不会被虚拟机读取在运行的时候；  
　　3.RUNTIME:在运行时有效,表示注解的信息被保留在class文件(字节码文件)中当程序编译时，会被虚拟机保留在运行时;  
@Documented表示该注解是否可以生成到 API文档中。

@Inherited：使用@Inherited定义的注解具备继承性。

## 数据库引擎介绍

|  |  |  |
| --- | --- | --- |
| 名字 | 特点 | 使用场景 |
| InnoDB | 事务，行级锁，外键约束 | 并发，自增长， |
| MyISAM | 快速读取操作，不能恢复数据 | Select和insert为主， |
| Memory | 极快的访问速度，数据不稳定 | 内容变化不频繁，中间表 |
| merge | 就是一组MyISAM |  |

## InnoDB存储引擎的逻辑结构？

## MySQL中常见的索引与其区别？

## MySQL中的空值为什么不走索引？

## MySQL中隔离级别的底层实现？

## MySQL主从库如何存入的？binary log有几种类型？

## Threadlocal什么情况可能发生内存泄漏？

## 谈谈Jvm的GC Root？

## Java的四种引用类型

强引用FinalReference：gc永远不会回收，即使OOM，可以设置null。

软引用SoftReference：内存不够会被清理。可以用来实现缓存技术

弱引用WeakReference：只要gc就会被清理。

ThreadLocal的ThreadLocalMap下Entry继承WeakReference

虚引用PhantomReference：垃圾回收器就能够随意释放对象。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，它随时可能会被回收

String str = new String("疯狂Java讲义");   
ReferenceQueue rq = new ReferenceQueue();

## 单例模式

特点：构造方法私有化，实例化变量的私有化，获取变量方法共有

优点：减少频繁的创建和销毁，减少内存的占用。

为什么使用枚举单例模式？反射和反序列化不会破坏单一原则。

反射攻击可以在无参构造方法添加检测返回。

# 并发

## 公平锁和非公平锁，乐观锁和悲观锁

|  |  |  |
| --- | --- | --- |
|  | 解释 | 实现 |
| 公平锁 | 能保证每个线程都能拿到锁，FIFO | reentrantLock |
| 非公平锁 | 不能保证每个线程都能拿到锁，随机 | Synchronized, reentrantLock(false) |
| 悲观锁 | 假设一定会发生并发冲突，加锁 | lock等 |
| 乐观锁 | 假设不会发生并发冲突，不加锁 | CAS |

## 重入锁和非重入锁

判断标准，当前线程已经获取到该锁，当再次尝试获取锁的时候，如果支持则为重入锁。

重入锁：

public class Lock {  
 boolean isLocked = false;  
 Thread lockedBy = null;  
 int lockedCount = 0;  
 public synchronized void lock() throws InterruptedException {  
 Thread thread = Thread.*currentThread*();  
 while (isLocked && lockedBy != thread) {wait();}  
 isLocked = true;  
 lockedCount++;  
 lockedBy = thread;  
 }  
 public synchronized void unlock() {  
 if (Thread.*currentThread*() == this.lockedBy) {  
 lockedCount--;  
 if (lockedCount == 0) {  
 isLocked = false;  
 notify();  
 }  
 }  
 }  
}

非重入锁：

public class Lock{  
 private boolean isLocked = false;  
 public synchronized void lock() throws InterruptedException{  
 while(isLocked){  
 wait();  
 }  
 isLocked = true;  
 }  
 public synchronized void unlock(){  
 isLocked = false;  
 notify();  
 }  
}

## CAS

compareAndSwapLong(Object var1, long var2, long var4, long var6)

第一个参数：期望修改的对象

第二个参数：属性相对于当前对象的偏移量

sun.misc.Unsafe UNSAFE;long mapOffset;

UNSAFE = sun.misc.Unsafe.getUnsafe();//必须使jvm源码调用才行

Class<?> k = Class.class;//当前类

mapOffset = UNSAFE.objectFieldOffset (k.getDeclaredField("name"));//当前类下的变量

第三个参数：期待值

第四个参数：要修改的值

ABA问题

A线程修改单项链表（Q->W->null）的head为W，此时BC线程使链表先后变成了（Q->E->R->null），head没变，但是此时W为游离状态。导致修改单项链表为（W->null）

解决ABA问题

AtomicStampedReference类

Pair<T> {final T reference;final int stamp;

private Pair(T reference, int stamp) {

this.reference = referencethis.stamp = stamp;}

static <T> Pair<T> of(T reference, int stamp) {

return new Pair<T>(reference, stamp);}}

**.compareAndSet(V expectedReference,V newReference,int expectedStamp,int newStamp)**

Pair<V> current = pair;

return expectedReference == current.reference &&expectedStamp == current.stamp &&

((newReference == current.reference &&newStamp == current.stamp) ||

casPair(current, Pair.of(newReference, newStamp)));

**casPair：UNSAFE.compareAndSwapObject(this, pairOffset, cmp, val)**

## 同步的核心三要素

原子性：i++三步操作有，读取i，+1，再赋值。

可见性：volatile保证内存只有一份

有序性：volatile 保证执行顺序

## LongAdder

写大于读的次数时候使用。

LongAdder在AtomicLong的基础上将单点的更新压力分散到各个节点，在低并发的时候通过对base的直接更新可以很好的保障和AtomicLong的性能基本保持一致，而在高并发的时候通过分散提高了性能。

缺点是LongAdder在统计的时候如果有并发更新，可能导致统计的数据有误差。

## SemPhore，CountDownLatch，CyclicBarrier

new SemPhore(n),acquire,release：允许n个线程进入代码段

new CountDownLatch(n),countDown,await：唤醒

new CyclicBarrier(n,Runnable),await：等待n个线程一起进入代码段