最近看到网上流传着，各种面试经验及面试题，往往都是一大堆技术题目贴上去，而没有答案。

为此我业余时间整理了，Java基础常见的40道常见面试题，及详细答案，望各路大牛，发现不对的地方，不吝赐教，留言即可。

八种基本数据类型的大小，以及他们的封装类
引用数据类型
Switch能否用string做参数
equals与==的区别
自动装箱，常量池
Object有哪些公用方法
Java的四种引用，强弱软虚，用到的场景
Hashcode的作用
HashMap的hashcode的作用
为什么重载hashCode方法?
ArrayList、LinkedList、Vector的区别
String、StringBubber与StringBuilder的区别
Map、Set、List、Queue、Stack的特点与用法
HashMap和HashTable的区别
JDK7与JDK8中HashMap的实现
HashMap和ConcurrentHashMap的区别，HashMap的底层源码
ConcurrentHashMap能完全替代HashTable吗
为什么HashMap是线程不安全的
如何线程安全的使用HashMap
多并发情况下HashMap是否还会产生死循环
TreeMap、HashMap、LindedHashMap的区别
Collection包结构，与Collections的区别
try?catch?finally，try里有return，finally还执行么
Exception与Error包结构，OOM你遇到过哪些情况，SOF你遇到过哪些情况
Java(OOP)面向对象的三个特征与含义
Override和Overload的含义去区别
Interface与abstract类的区别
Static?class?与non?static?class的区别
foreach与正常for循环效率对比
Java?IO与NIO
java反射的作用于原理
泛型常用特点
解析XML的几种方式的原理与特点：DOM、SAX
Java1.7与1.8,1.9,10新特性
设计模式：单例、工厂、适配器、责任链、观察者等等
JNI的使用
AOP是什么
OOP是什么

AOP 与 OOP 的区别

八种基本数据类型的大小，以及他们的封装类

八种基本数据类型：int、short、float、double、long、boolean、byte、char。

封装类分别是：Integer、Short、Float、Double、Long、Boolean、Byte、Character。

引用数据类型

引用数据类型是由类的编辑器定义的，他们是用于访问对象的。这些变量被定义为不可更改的特定类型。

例如：Employee，Puppy 等等

类对象和数组变量就是这种引用数据类型。
任何引用数据类型的默认值都为空。
一个引用数据类型可以被用于任何声明类型和兼容类型的对象。

Switch 能否用 string 做参数

jdk7 之前
switch 只能支持 byte、short、char、int 这几个基本数据类型和其对应的封装类型。

switch 后面的括号里面只能放 int 类型的值，但由于 byte，short，char 类型，它们会自动转换为 int 类型（精精度小的向大的转化），所以它们也支持。

jdk1.7 后
整形，枚举类型，boolean，字符串都可以。

原理

```
switch (expression)  // 括号里是一个表达式，结果是个整数{
  case constant1:   // case 后面的标号，也是个整数
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  ...
  default:
     default group of statements
}
```

jdk1.7 后，整形，枚举类型，boolean，字符串都可以。

```
public class TestString {

    static String string = "123";
```

```
    public static void main(String[] args) {
        switch (string) {
        case "123":
            System.out.println("123");
            break;
        case "abc":
            System.out.println("abc");
            break;
        default:
            System.out.println("default");
            break;
        }
    }
}
```

为什么 jdk1.7 后又可以用 string 类型作为 switch 参数呢？

其实，jdk1.7 并没有新的指令来处理 switch string，而是通过调用 switch 中 string.hashCode，将 string 转换为 int 从而进行判断。

equals 与 == 的区别

使用==比较原生类型如：boolean、int、char 等等，使用 equals() 比较对象。

1、==是判断两个变量或实例是不是指向同一个内存空间。
equals 是判断两个变量或实例所指向的内存空间的值是不是相同。

2、==是指对内存地址进行比较。
equals() 是对字符串的内容进行比较。

3、==指引用是否相同。
equals() 指的是值是否相同。

```
public static void main(String[] args) {

        String a = new String("ab"); // a 为一个引用
        String b = new String("ab"); // b 为另一个引用，对象的内容一样
        String aa = "ab"; // 放在常量池中
        String bb = "ab"; // 从常量池中查找

        System.out.println(aa == bb); // true
        System.out.println(a == b); // false，非同一对象
        System.out.println(a.equals(b)); // true
        System.out.println(42 == 42.0);  // true
    }
```

```
public static void main(String[] args) {
    Object obj1 = new Object();
    Object obj2 = new Object();
    System.out.println(obj1.equals(obj2));//false
    System.out.println(obj1==obj2);//false
    obj1=obj2;
    System.out.println(obj1==obj2);//true
    System.out.println(obj2==obj1);//true
}
```

自动装箱，常量池

自动装箱 在 jdk1.5 之前，如果你想要定义一个 value 为 100 的 Integer 对象，则需要如下定义：

```
Integer i = new Integer(100);
```

```
int intNum1 = 100; //普通变量
Integer intNum2 = intNum1; //自动装箱
int intNum3 = intNum2; //自动拆箱
Integer intNum4 = 100; //自动装箱
```

上面的代码中，intNum2 为一个 Integer 类型的实例，intNum1 为 Java 中的基础数据类型，将 intNum1 赋值给 intNum2 便是自动装箱；而将 intNum2 赋值给 intNum3 则是自动拆箱。

八种基本数据类型：boolean byte char shrot int long bloat double，所生成的变量相当于常量。

基本类型包装类：Boolean Byte Character Short Integer Long Float Double。

自动拆箱和自动装箱定义：

自动装箱是将一个 java 定义的基本数据类型赋值给相应封装类的变量。
拆箱与装箱是相反的操作，自动拆箱则是将一个封装类的变量赋值给相应基本数据类型的变量。

Object 有哪些公用方法

Object 是所有类的父类，任何类都默认继承 Object

clone
保护方法，实现对象的浅复制，只有实现了 Cloneable 接口才可以调用该方法，否则抛出 CloneNotSupportedException 异常。

equals
在 Object 中与==是一样的，子类一般需要重写该方法。

hashCode

该方法用于哈希查找，重写了 equals 方法一般都要重写 hashCode 方法。这个方法在一些具有哈希功能的 Collection 中用到。

getClass
final 方法，获得运行时类型

wait
使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。
wait() 方法一直等待，直到获得锁或者被中断。
wait(long timeout) 设定一个超时间隔，如果在规定时间内没有获得锁就返回。

调用该方法后当前线程进入睡眠状态，直到以下事件发生

1、其他线程调用了该对象的 notify 方法。
2、其他线程调用了该对象的 notifyAll 方法。
3、其他线程调用了 interrupt 中断该线程。
4、时间间隔到了。
5、此时该线程就可以被调度了，如果是被中断的话就抛出一个 InterruptedException 异常。

notify
唤醒在该对象上等待的某个线程。

notifyAll
唤醒在该对象上等待的所有线程。

toString
转换成字符串，一般子类都有重写，否则打印句柄。

Java 的四种引用，强弱软虚，用到的场景

从 JDK1.2 版本开始，把对象的引用分为四种级别，从而使程序能更加灵活的控制对象的生命周期。这四种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

1、强引用

最普遍的一种引用方式，如 String s = "abc"，变量 s 就是字符串 "abc" 的强引用，只要强引用存在，垃圾回收器就不会回收这个对象。

2、软引用（SoftReference）

用于描述还有用但非必须的对象，如果内存足够，不回收，如果内存不足，则回收。一般用于实现内存敏感的高速缓存，软引用可以和引用队列 ReferenceQueue 联合使用，如果软引用的对象被垃圾回收，JVM 就会把这个软引用加入到与之关联的引用队列中。

3、弱引用（WeakReference）

弱引用和软引用大致相同，弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

4、虚引用（PhantomReference）

就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。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，在任何时候都可能被垃圾回收器回收。虚引用主要用来跟踪对象被垃圾回收器回收的活动。

虚引用与软引用和弱引用的一个区别在于：

虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

Hashcode 的作用

http://blog.csdn.net/seu_calv...

1、HashCode 的特性

（1）HashCode 的存在主要是用于查找的快捷性，如 Hashtable，HashMap 等，HashCode 经常用于确定对象的存储地址。

（2）如果两个对象相同，?equals 方法一定返回 true，并且这两个对象的 HashCode 一定相同。

（3）两个对象的 HashCode 相同，并不一定表示两个对象就相同，即 equals() 不一定为 true，只能够说明这两个对象在一个散列存储结构中。

（4）如果对象的 equals 方法被重写，那么对象的 HashCode 也尽量重写。

2、HashCode 作用

Java 中的集合有两类，一类是 List，再有一类是 Set。前者集合内的元素是有序的，元素可以重复；后者元素无序，但元素不可重复。

equals 方法可用于保证元素不重复，但如果每增加一个元素就检查一次，若集合中现在已经有 1000 个元素，那么第 1001 个元素加入集合时，就要调用 1000 次 equals 方法。这显然会大大降低效率。?于是，Java 采用了哈希表的原理。

哈希算法也称为散列算法，是将数据依特定算法直接指定到一个地址上。

这样一来，当集合要添加新的元素时，先调用这个元素的HashCode方法，就一下子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，它就可以直接存储在这个位置上，不用再进行任何比较了。

（2）如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了。

（3）不相同的话，也就是发生了Hash Key相同导致冲突的情况，那么就在这个Hash Key的地方产生一个链表，将所有产生相同HashCode的对象放到这个单链表上去，串在一起（很少出现）。

这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。

如何理解HashCode的作用：

从Object角度看，JVM每new一个Object，它都会将这个Object丢到一个Hash表中去，这样的话，下次做Object的比较或者取这个对象的时候（读取过程），它会根据对象的HashCode再从Hash表中取这个对象。这样做的目的是提高取对象的效率。若HashCode相同再去调用equal。

3、HashCode实践（如何用来查找）

HashCode是用于查找使用的，而equals是用于比较两个对象是否相等的。

（1）例如内存中有这样的位置

0 1 2 3 4 5 6 7
而我有个类，这个类有个字段叫ID，我要把这个类存放在以上8个位置之一，如果不用HashCode而任意存放，那么当查找时就需要到这八个位置里挨个去找，或者用二分法一类的算法。

但以上问题如果用HashCode就会使效率提高很多
定义我们的HashCode为ID%8，比如我们的ID为9，9除8的余数为1，那么我们就把该类存在1这个位置，如果ID是13，求得的余数是5，那么我们就把该类放在5这个位置。依此类推。

（2）但是如果两个类有相同的HashCode，例如9除以8和17除以8的余数都是1，也就是说，我们先通过?HashCode来判断两个类是否存放某个桶里，但这个桶里可能有很多类，那么我们就需要再通过equals在这个桶里找到我们要的类。

请看下面这个例子

```
public class HashTest {
```

```
    private int i;

    public int getI() {
        return i;
    }

    public void setI(int i) {
        this.i = i;
    }

    public int hashCode() {
        return i % 10;
    }

    public final static void main(String[] args) {
        HashTest a = new HashTest();
        HashTest b = new HashTest();
        a.setI(1);
        b.setI(1);
        Set<HashTest> set = new HashSet<HashTest>();
        set.add(a);
        set.add(b);
        System.out.println(a.hashCode() == b.hashCode());
        System.out.println(a.equals(b));
        System.out.println(set);
    }
}
```

输出结果为：

```
true
False
[HashTest@1, HashTest@1]
```

以上这个示例，我们只是重写了HashCode方法，从上面的结果可以看出，虽然两个对象的HashCode相等，但是实际上两个对象并不是相等，因为我们没有重写equals方法，那么就会调用Object默认的equals方法，显示这是两个不同的对象。

这里我们将生成的对象放到了HashSet中，而HashSet中只能够存放唯一的对象，也就是相同的（适用于equals方法）的对象只会存放一个，但是这里实际上是两个对象ab都被放到了HashSet中，这样HashSet就失去了他本身的意义了。

下面我们继续重写equals方法：

```
public class HashTest {
    private int i;
```

```
    public int getI() {
        return i;
    }

    public void setI(int i) {
        this.i = i;
    }

    public boolean equals(Object object) {
        if (object == null) {
            return false;
        }
        if (object == this) {
            return true;
        }
        if (!(object instanceof HashTest)) {
            return false;
        }
        HashTest other = (HashTest) object;
        if (other.getI() == this.getI()) {
            return true;
        }
        return false;
    }

    public int hashCode() {
        return i % 10;
    }

    public final static void main(String[] args) {
        HashTest a = new HashTest();
        HashTest b = new HashTest();
        a.setI(1);
        b.setI(1);
        Set<HashTest> set = new HashSet<HashTest>();
        set.add(a);
        set.add(b);
        System.out.println(a.hashCode() == b.hashCode());
        System.out.println(a.equals(b));
        System.out.println(set);
    }
}
```

输出结果如下所示。

从结果我们可以看出，现在两个对象就完全相等了，HashSet中也只存放了一份对象。

注意：

hashCode()只是简单示例写的，真正的生产换将不是这样的

true
true
[HashTest@1]

HashMap的hashcode的作用

hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的。

如果两个对象相同，就是适用于equals(java.lang.Object)方法，那么这两个对象的hashCode一定要相同。

如果对象的equals方法被重写，那么对象的hashCode也尽量重写，并且产生hashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点。

两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object)方法，只能够说明这两个对象在散列存储结构中，如Hashtable，他们“存放在同一个篮子里”。

什么时候需要重写？

一般的地方不需要重载hashCode，只有当类需要放在HashTable、HashMap、HashSet等等hash结构的集合时才会重载hashCode，那么为什么要重载hashCode呢？

要比较两个类的内容属性值，是否相同时候，根据hashCode重写规则，重写类的 指定字段的hashCode()，equals()方法。

例如

```
public class EmpWorkCondition{

    /**
     * 员工ID
     */
    private Integer empId;

    /**
     * 员工服务总单数
     */
```

```
    private Integer orderSum;

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        EmpWorkCondition that = (EmpWorkCondition) o;
        return Objects.equals(empId, that.empId);
    }

    @Override
    public int hashCode() {
        return Objects.hash(empId);
    }

    // 省略 getter setter
}
public static void main(String[] args) {

    List<EmpWorkCondition> list1 = new ArrayList<EmpWorkCondition>();

    EmpWorkCondition emp1 = new EmpWorkCondition();
    emp1.setEmpId(100);
    emp1.setOrderSum(90000);
    list1.add(emp1);

    List<EmpWorkCondition> list2 = new ArrayList<EmpWorkCondition>();

    EmpWorkCondition emp2 = new EmpWorkCondition();
    emp2.setEmpId(100);
    list2.add(emp2);

    System.out.println(list1.contains(emp2));

}
```

输出结果：true

上面的方法，做的事情就是，比较两个集合中的，实体类对象属性值，是否一致

OrderSum 不在比较范围内，因为没有重写它的，equals() 和 hashCode() 方法

为什么要重载 equal 方法?

因为 Object 的 equal 方法默认是两个对象的引用的比较，意思就是指向同一内存，地址则相等，否则不相等；如果你现在需要利用对象里面的值来判断是否相等，则重载 equal 方法。

为什么重载 hashCode 方法?
一般的地方不需要重载 hashCode，只有当类需要放在 HashTable、HashMap、HashSet 等等 hash 结构的集合时才会重载 hashCode，那么为什么要重载 hashCode 呢?

如果你重写了 equals，比如说是基于对象的内容实现的，而保留 hashCode 的实现不变，那么很可能某两个对象明明是“相等”，而 hashCode 却不一样。

这样，当你用其中的一个作为键保存到 hashMap、hasoTable 或 hashSet 中，再以“相等的”找为一个作为键值去查找他们的时候，则根本找不到。

为什么 equals() 相等，hashCode 就一定要相等，而 hashCode 相等，却不要求 equals 相等?

1、因为是按照 hashCode 来访问小内存块，所以 hashCode 必须相等。
2、HashMap 获取一个对象是比较 Key 的 hashCode 相等和 equal 为 true。

之所以 hashCode 相等，却可以 equal 不等，就比如 ObjectA 和 ObjectB 他们都有属性 name，那么 hashCode 都以 name 计算，所以 hashCode 一样，但是两个对象属于不同类型，所以 equal 为 false。

为什么需要 hashCode?

1、通过 hashCode 可以很快的查到小内存块。
2、通过 hashCode 比较比 equal 方法快，当 get 时先比较 hashCode，如果 hashCode 不同，直接返回 false。

ArrayList、LinkedList、Vector 的区别
List 的三个子类的特点

ArrayList:

底层数据结构是数组，查询快，增删慢。
线程不安全，效率高。
Vector:

底层数据结构是数组，查询快，增删慢。
线程安全，效率低。
Vector 相对 ArrayList 查询慢(线程安全的)。
Vector 相对 LinkedList 增删慢(数组结构)。

LinkedList

底层数据结构是链表，查询慢，增删快。
线程不安全，效率高。
Vector 和 ArrayList 的区别

Vector 是线程安全的，效率低。
ArrayList 是线程不安全的，效率高。
共同点：底层数据结构都是数组实现的，查询快，增删慢。
ArrayList 和 LinkedList 的区别

ArrayList 底层是数组结果，查询和修改快。
LinkedList 底层是链表结构的，增和删比较快，查询和修改比较慢。
共同点：都是线程不安全的

List 有三个子类使用

查询多用 ArrayList。
增删多用 LinkedList。
如果都多 ArrayList。
String、StringBubber 与 StringBuilder 的区别
String：适用于少量的字符串操作的情况。
StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况。
StringBubber：适用多线程下在字符缓冲区进行大量操作的情况。
StringBuilder：是线程不安全的，而 StringBubber 是线程安全的。

这三个类之间的区别主要是在两个方面，即运行速度和线程安全这两方面。
首先说运行速度，或者说是执行速度，在这方面运行速度快慢为：StringBuilder > StringBubber > String。

String 最慢的原因

String 为字符串常量，而 StringBuilder 和 StringBubber 均为字符串变量，即 String 对象一旦创建之后该对象是不可更改的，但后两者的对象是变量，是可以更改的。

再来说线程安全

在线程安全上，StringBuilder 是线程不安全的，而 StringBubber 是线程安全的。

如果一个 StringBubber 对象在字符串缓冲区被多个线程使用时，StringBubber 中很多方法可以带有 synchronized 关键字，所以可以保证线程是安全的，但 StringBuilder 的方法则没有该关键字，所以不能保证线程安全，有可能会出现一些错误的操作。所以如果要进行的操作是多线程的，那么就要使用 StringBubber，但是在单线程的情况下，还是建议使用速度比较快的 StringBuilder。

Map、Set、List、Queue、Stack 的特点与用法

Map

Map 是键值对，键 Key 是唯一不能重复的，一个键对应一个值，值可以重复。
TreeMap 可以保证顺序。
HashMap 不保证顺序，即为无序的。
Map 中可以将 Key 和 Value 单独抽取出来，其中 KeySet() 方法可以将所有的 Keys 抽取正一个 Set。而 Values() 方法可以将 map 中所有的 values 抽取成一个集合。

Set

不包含重复元素的集合，set 中最多包含一个 null 元素。
只能用 Iterator 实现单项遍历，Set 中没有同步方法。

List

有序的可重复集合。
可以在任意位置增加删除元素。
用 Iterator 实现单向遍历，也可用 ListIterator 实现双向遍历。

Queue

Queue 遵从先进先出原则。
使用时尽量避免 add() 和 remove() 方法，而是使用 offer() 来添加元素，使用 poll() 来移除元素，它的优点是可以通过返回值来判断是否成功。
LinkedList 实现了 Queue 接口。
Queue 通常不允许插入 null 元素。

Stack

Stack 遵从后进先出原则。
Stack 继承自 Vector。
它通过五个操作对类 Vector 进行扩展，允许将向量视为堆栈，它提供了通常的 push 和 pop 操作，以及取堆栈顶点的 peek() 方法、测试堆栈是否为空的 empty 方法等。

用法

如果涉及堆栈，队列等操作，建议使用 List。
对于快速插入和删除元素的，建议使用 LinkedList。
如果需要快速随机访问元素的，建议使用 ArrayList。

更为精炼的总结

Collection 是对象集合，Collection 有两个子接口 List 和 Set

List 可以通过下标 (1,2..) 来取得值，值可以重复。
Set 只能通过游标来取值，并且值是不能重复的。

ArrayList，Vector，LinkedList 是 List 的实现类

ArrayList 是线程不安全的， Vector 是线程安全的，这两个类底层都是由数组实现的。
LinkedList 是线程不安全的，底层是由链表实现的。
Map 是键值对集合

HashTable 和 HashMap 是 Map 的实现类。
HashTable 是线程安全的，不能存储 null 值。
HashMap 不是线程安全的，可以存储 null 值。
Stack 类：继承自 Vector，实现一个后进先出的栈。提供了几个基本方法，push、pop、peak、empty、search 等。

Queue 接口：提供了几个基本方法，offer、poll、peek 等。已知实现类有 LinkedList、PriorityQueue 等。

HashMap 和 HashTable 的区别

https://segmentfault.com/a/11...

Hashtable 是基于陈旧的 Dictionary 类的，HashMap 是 Java 1.2 引进的 Map 接口的一个实现，它们都是集合中将数据无序存放的。

1、 hashMap 去掉了 HashTable? 的 contains 方法，但是加上了 containsValue() 和 containsKey() 方法

HashTable Synchronize 同步的，线程安全，HashMap 不允许空键值为空?，效率低。
HashMap 非 Synchronize 线程同步的，线程不安全，HashMap 允许空键值为空?，效率高。
Hashtable 是基于陈旧的 Dictionary 类的，HashMap 是 Java 1.2 引进的 Map 接口的一个实现，它们都是集合中将数据无序存放的。

Hashtable 的方法是同步的，HashMap 未经同步，所以在多线程场合要手动同步 HashMap 这个区别就像 Vector 和 ArrayList 一样。

查看 Hashtable 的源代码就可以发现，除构造函数外，Hashtable 的所有 public 方法声明中都有 synchronized 关键字，而 HashMap 的源代码中则连 synchronized 的影子都没有，当然，注释除外。

2、Hashtable 不允许 null 值(key 和 value 都不可以)，HashMap 允许 null 值(key 和 value 都可以)。

3、两者的遍历方式大同小异，Hashtable 仅仅比 HashMap 多一个 elements 方法。

```
Hashtable table = new Hashtable();
table.put("key", "value");
Enumeration em = table.elements();
while (em.hasMoreElements()) {
```

```
    String obj = (String) em.nextElement();
    System.out.println(obj);
}
```

4、HashTable 使用 Enumeration，HashMap 使用 Iterator

从内部机制实现上的区别如下：

哈希值的使用不同，Hashtable 直接使用对象的 hashCode

```
int hash = key.hashCode();
int index = (hash & 0x7FFFFFFF) % tab.length;
```

而 HashMap 重新计算 hash 值，而且用与代替求模：

```
int hash = hash(k);
int i = indexFor(hash, table.length);

static int hash(Object x) {
    int h = x.hashCode();

    h += ~(h << 9);
    h ^= (h >>> 14);
    h += (h << 4);
    h ^= (h >>> 10);
    return h;
}
static int indexFor(int h,  int length) {
    return h & (length-1);
```

Hashtable 中 hash 数组默认大小是 11，增加的方式是 old*2+1。HashMap 中 hash 数组的默认大小是 16，而且一定是 2 的指数。

JDK7 与 JDK8 中 HashMap 的实现

JDK7 中的 HashMap

HashMap 底层维护一个数组，数组中的每一项都是一个 Entry。

```
transient Entry<K,V>[] table;
```

我们向 HashMap 中所放置的对象实际上是存储在该数组当中。

而 Map 中的 key，value 则以 Entry 的形式存放在数组中。

```
static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        Entry<K,V> next;
        int hash;
```

总结一下 map.put 后的过程：

当向 HashMap 中 put 一对键值时，它会根据 key 的 hashCode 值计算出一个位置，该位置就是此对象准备往数组中存放的位置。

如果该位置没有对象存在，就将此对象直接放进数组当中；如果该位置已经有对象存在了，则顺着此存在的对象的链开始寻找（为了判断是否是否值相同，map 不允许<key,value>键值对重复），如果此链上有对象的话，再去使用 equals 方法进行比较，如果对此链上的每个对象的 equals 方法比较都为 false，则将该对象放到数组当中，然后将数组中该位置以前存在的那个对象链接到此对象的后面。

JDK8 中的 HashMap

JDK8 中采用的是位桶+链表/红黑树（有关红黑树请查看红黑树）的方式，也是非线程安全的。当某个位桶的链表的长度达到某个阀值的时候，这个链表就将转换成红黑树。

JDK8 中，当同一个 hash 值的节点数不小于 8 时，将不再以单链表的形式存储了，会被调整成一颗红黑树（上图中 null 节点没画）。这就是 JDK7 与 JDK8 中 HashMap 实现的最大区别。

接下来，我们来看下 JDK8 中 HashMap 的源码实现。

JDK 中 Entry 的名字变成了 Node，原因是和红黑树的实现 TreeNode 相关联。

transient Node<K,V>[] table;
当冲突节点数不小于 8-1 时，转换成红黑树。

static final int TREEIFY_THRESHOLD = 8;

HashMap 和 ConcurrentHashMap 的区别，HashMap 的底层源码

为了线程安全从 ConcurrentHashMap 代码中可以看出，它引入了一个“分段锁”的概念，具体可以理解为把一个大的 Map 拆分成 N 个小的 HashTable，根据 key.hashCode() 来决定把 key 放到哪个 HashTable 中。

Hashmap 本质是数组加链表。根据 key 取得 hash 值，然后计算出数组下标，如果多个 key 对在到同一个下标，就用链表串起来，新插入的在前面。

ConcurrentHashMap：在 hashMap 的基础上，ConcurrentHashMap 将数据分为多个 segment，默认 16 个（concurrency level），然后每次操作对一个 segment 加锁，避免多线程锁的几率，提高并发效率。

总结

JDK6,7 中的 ConcurrentHashmap 主要使用 Segment 来实现减小锁粒度，把 HashMap 分割成若干个 Segment，在 put 的时候需要锁住 Segment，get 时候不加锁，使用 volatile 来保证可见性，当要统计全局时（比如 size），首先会尝试多次计算 modcount 来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回 size。如果有，则需要依次锁住所有的 Segment 来计算。

jdk7 中 ConcurrentHashmap 中，当长度过长碰撞会很频繁，链表的增改删查操作都会消耗很长的时间，影响性能。

jdk8 中完全重写了 concurrentHashmap，代码量从原来的 1000 多行变成了 6000 多行，实现上也和原来的分段式存储有很大的区别。

JDK8 中采用的是位桶+链表/红黑树（有关红黑树请查看红黑树）的方式，也是非线程安全的。当某个位桶的链表的长度达到某个阀值的时候，这个链表就将转换成红黑树。

JDK8 中，当同一个 hash 值的节点数不小于 8 时，将不再以单链表的形式存储了，会被调整成一颗红黑树（上图中 null 节点没画）。这就是 JDK7 与 JDK8 中 HashMap 实现的最大区别。

主要设计上的变化有以下几点

1. jdk8 不采用 segment 而采用 node，锁住 node 来实现减小锁粒度。
2. 设计了 MOVED 状态 当 resize 的中过程中 线程2还在 put 数据，线程2会帮助 resize。
3. 使用 3 个 CAS 操作来确保 node 的一些操作的原子性，这种方式代替了锁。
4. sizeCtl 的不同值来代表不同含义，起到了控制的作用。

至于为什么 JDK8 中使用 synchronized 而不是 ReentrantLock，我猜是因为 JDK8 中对 synchronized 有了足够的优化吧。

ConcurrentHashMap 能完全替代 HashTable 吗

hashTable 虽然性能上不如 ConcurrentHashMap，但并不能完全被取代，两者的迭代器的一致性不同的，hash table 的迭代器是强一致性的，而 concurrenthashmap 是弱一致的。

ConcurrentHashMap 的 get，clear，iterator 都是弱一致性的。Doug Lea 也将这个判断留给用户自己决定是否使用 ConcurrentHashMap。

ConcurrentHashMap 与 HashTable 都可以用于多线程的环境，但是当 Hashtable 的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。因为 ConcurrentHashMap 引入了分割（segmentation），不论它变得多么大，仅仅需要锁定 map 的某个部分，而其它的线程不需要等到迭代完成才能访问 map。简而言之，在迭代的过程中，ConcurrentHashMap 仅仅锁定 map 的某个部分，而 Hashtable 则会锁定整个 map。

那么既然 ConcurrentHashMap 那么优秀，为什么还要有 Hashtable 的存在呢？ConcurrentHashMap 能完全替代 HashTable 吗？

HashTable 虽然性能上不如 ConcurrentHashMap，但并不能完全被取代，两者的迭代器的一致性不同的，HashTable 的迭代器是强一致性的，而 ConcurrentHashMap 是弱一致的。ConcurrentHashMap 的 get，clear，iterator 都是弱一致性的。Doug Lea 也将这个判断留给用户自己决定是否使用 ConcurrentHashMap。

那么什么是强一致性和弱一致性呢？

get 方法是弱一致的，是什么含义？可能你期望往 ConcurrentHashMap 底层数据结构中加入一个元素后，立马能对 get 可见，但 ConcurrentHashMap 并不能如你所愿。换句话说，put 操作将一个元素加入到底层数据结构后，get 可能在某段时间内还看不到这个元素，若不考虑内存模型，单从代码逻辑上来看，却是应该可以看得到的。

下面将结合代码和 java 内存模型相关内容来分析下 put/get 方法。put 方法我们只需关注 Segment#put，get 方法只需关注 Segment#get，在继续之前，先要说明一下 Segment 里有两个 volatile 变量：count 和 table；HashEntry 里有一个 volatile 变量：value。

总结

ConcurrentHashMap 的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与 Hashtable 和同步的 HashMap 一样了。

为什么 HashMap 是线程不安全的

HashMap 在并发执行 put 操作时会引起死循环，导致 CPU 利用率接近 100%。因为多线程会导致 HashMap 的 Node 链表形成环形数据结构，一旦形成环形数据结构，Node 的 next 节点永远不为空，就会在获取 Node 时产生死循环。

如何线程安全的使用 HashMap

了解了 HashMap 为什么线程不安全，那现在看看如何线程安全的使用 HashMap。这个无非就是以下三种方式：

Hashtable
ConcurrentHashMap
Synchronized Map

Hashtable

例子

```
//Hashtable
Map<String, String> hashtable = new Hashtable<>();
//synchronizedMap
Map<String, String> synchronizedHashMap = Collections.synchronizedMap(new
HashMap<String, String>());
//ConcurrentHashMap
Map<String, String> concurrentHashMap = new ConcurrentHashMap<>();
```

Hashtable

先稍微吐槽一下，为啥命名不是 HashTable 啊，看着好难受不管了就装作它叫 HashTable 吧。这货已经不常用了，就简单说说吧。HashTable 源码中是使用 synchronized 来保证线

程安全的，比如下面的 get 方法和 put 方法：

```
public synchronized V get(Object key) {
    // 省略实现
}
public synchronized V put(K key, V value) {
// 省略实现
}
```

所以当一个线程访问 HashTable 的同步方法时，其他线程如果也要访问同步方法，会被阻塞住。举个例子，当一个线程使用 put 方法时，另一个线程不但不可以使用 put 方法，连 get 方法都不可以，好霸道啊！！！so~~，效率很低，现在基本不会选择它了。

ConcurrentHashMap

ConcurrentHashMap 于 Java 7 的，和 8 有区别，在 8 中 CHM 摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现，利用 CAS 算法，有时间会重新总结一下。

SynchronizedMap

synchronizedMap() 方法后会返回一个 SynchronizedMap 类的对象，而在 SynchronizedMap 类中使用了 synchronized 同步关键字来保证对 Map 的操作是线程安全的。

性能对比

这是要靠数据说话的时代，所以不能只靠嘴说 CHM 快，它就快了。写个测试用例，实际的比较一下这三种方式的效率（源码来源），下面的代码分别通过三种方式创建 Map 对象，使用 ExecutorService 来并发运行 5 个线程，每个线程添加/获取 500K 个元素。

```
Test started for: class java.util.Hashtable
2500K entried added/retrieved in 2018 ms
2500K entried added/retrieved in 1746 ms
2500K entried added/retrieved in 1806 ms
2500K entried added/retrieved in 1801 ms
2500K entried added/retrieved in 1804 ms
For class java.util.Hashtable the average time is 1835 ms

Test started for: class java.util.Collections$SynchronizedMap
2500K entried added/retrieved in 3041 ms
2500K entried added/retrieved in 1690 ms
2500K entried added/retrieved in 1740 ms
2500K entried added/retrieved in 1649 ms
2500K entried added/retrieved in 1696 ms
For class java.util.Collections$SynchronizedMap the average time is 1963 ms
```

Test started for: class java.util.concurrent.ConcurrentHashMap
2500K entried added/retrieved in 738 ms
2500K entried added/retrieved in 696 ms
2500K entried added/retrieved in 548 ms
2500K entried added/retrieved in 1447 ms
2500K entried added/retrieved in 531 ms
For class java.util.concurrent.ConcurrentHashMap the average time is 792 ms
ConcurrentHashMap 性能是明显优于 Hashtable 和 SynchronizedMap 的，CHM 花费的时间比前两个的一半还少。

多并发情况下HashMap是否还会产生死循环

今天年来提看下了 ConcurrentHashMap 的源码，ConcurrentHashMap 是 Java 5 中支持高并发、高吞吐量的线程安全 HashMap 实现。

在看很多博客在介绍 ConcurrentHashMap 之前，都说 HashMap 适用于单线程访问，这是因为 HashMap 的所有方法都没有进行锁同步，因此是线程不安全的，不仅如此，当多线程访问的时候还容易产生死循环。

虽然自己在前几天的时候看过 HashMap 的源码，感觉思路啥啥的都还清楚，对于多线程访问只知道 HashMap 是线程不安全的，但是不知道 HashMap 在多线程并发的情况下会产生死循环呢，为什么会产生，何种情况下才会产生死循环呢？？？

既然会产生死循环，为什么并发情况下，还是用 ConcurrentHashMap。
jdk 好像有，但是 Jdk8 已经修复了这个问题。

TreeMap、HashMap、LindedHashMap的区别

LinkedHashMap 可以保证 HashMap 集合有序，存入的顺序和取出的顺序一致。

TreeMap 实现 SortMap 接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 TreeMap 时，得到的记录是排过序的。

HashMap 不保证顺序，即为无序的，具有很快的访问速度。
HashMap 最多只允许一条记录的键为 Null；允许多条记录的值为 Null。
HashMap 不支持线程的同步。

我们在开发的过程中使用 HashMap 比较多，在 Map 中在 Map 中插入、删除和定位元素，HashMap 是最好的选择。

但如果你要按自然顺序或自定义顺序遍历键，那么 TreeMap 会更好。

如果需要输出的顺序和输入的相同，那么用 LinkedHashMap 可以实现，它还可以按读取顺序来排列。

Collection 包结构，与 Collections 的区别

Collection 是集合类的上级接口，子接口主要有 Set、List 、Map。

Collecions 是针对集合类的一个帮助类，提供了操作集合的工具方法，一系列静态方法实现对各种集合的搜索、排序线性、线程安全化等操作。

例如

Map<String, Object> map4 = Collections.synchronizedMap(new HashMap<String, Object>()); 线程安全 的 HashMap
Collections.sort(List<T> list, Comparator<? super T> c); 排序 List
Collection

Collection 是单列集合

List

元素是有序的、可重复。
有序的 collection，可以对列表中每个元素的插入位置进行精确地控制。
可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。
可序放重复元素，元素存取是有序的。

List 接口中常用类

Vector：线程安全，但速度慢，已被 ArrayList 替代。底层数据结构是数组结构。
ArrayList：线程不安全，查询速度快。底层数据结构是数组结构。
LinkedList：线程不安全。增删速度快。底层数据结构是列表结构。

Set

Set 接口中常用的类

Set（集）元素无序的、不可重复。
取出元素的方法只有迭代器。不可以序放重复元素，元素存取是无序的。

HashSet：线程不安全，存取速度快。它是如何保证元素唯一性的呢？依赖的是元素的 hashCode 方法和 equals 方法。
TreeSet：线程不安全，可以对 Set 集合中的元素进行排序。它的排序是如何进行的呢？通过 compareTo 或者 compare 方法中的来保证元素的唯一性。元素是以二叉树的形式序放的。

Map

map 是一个双列集合

Hashtable：线程安全，速度快。底层是哈希表数据结构。是同步的。不允许 null 作为键，null 作为值。

Properties：用于配置文件的定义和操作，使用频率非常高，同时键和值都是字符串。是集合中可以和 IO 技术相结合的对象。

HashMap：线程不安全，速度慢。底层也是哈希表数据结构。是不同步的。允许 null 作为键，null 作为值，替代了 Hashtable。

LinkedHashMap：可以保证 HashMap 集合有序。存入的顺序和取出的顺序一致。

TreeMap：可以用来对 Map 集合中的键进行排序

try?catch?finally，try 里有 return，finally 还执行么

肯定会执行。finally{} 块的代码。
只有在 try{} 块中包含遇到 System.exit(0)。
之类的导致 Java 虚拟机直接退出的语句才会不执行。

当程序执行 try{} 遇到 return 时，程序会先执行 return 语句，但并不会立即返回——也就是把 return 语句要做的一切事情都准备好，也就是在将要返回、但并未返回的时候，程序把执行流程转去执行 finally 块，当 finally 块执行完成后就直接返回刚才 return 语句已经准备好的结果。

Exception 与 Error 包结构。OOM 你遇到过哪些情况，SO F 你遇到过哪些情况

Throwable 是 Java 语言中所有错误或异常的超类。
Throwable 包含两个子类：Error 和 Exception 。它们通常用于指示发生了异常情况。
Throwable 包含了其线程创建时线程执行堆栈的快照，它提供了 printStackTrace() 等接口用于获取堆栈跟踪数据等信息。

Java 将可抛出（Throwable）的结构分为三种类型：

被检查的异常（Checked Exception）。
运行时异常（RuntimeException）。
错误（Error）。

运行时异常 RuntimeException

定义 ：RuntimeException 及其子类都被称为运行时异常。
特点 ：Java 编译器不会检查它 也就是说，当程序中可能出现这类异常时，倘若既"没有通过 throws 声明抛出它"，也"没有用 try-catch 语句捕获它"，还是会编译通过。

例如，除数为零时产生的 ArithmeticException 异常，数组越界时产生的 IndexOutOfBoundsException 异常，fail-fail 机制产生的

ConcurrentModificationException 异常等，都属于运行时异常。

堆内存溢出 OutOfMemoryError（OOM）

除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生 OutOfMemoryError（OOM）异常的可能。

Java Heap 溢出。
一般的异常信息：java.lang.OutOfMemoryError: Java heap spacess。
java 堆用于存储对象实例，我们只要不断的创建对象，并且保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量达到最大堆容量限制后产生内存溢出异常。

堆栈溢出 StackOverflow （SOF）

StackOverflowError 的定义：
当应用程序递归太深而发生堆栈溢出时，抛出该错误。
因为栈一般默认为 1-2m，一旦出现死循环或者是大量的递归调用，在不断的压栈过程中，造成栈容量超过 1m 而导致溢出。

栈溢出的原因：

递归调用。
大量循环或死循环。
全局变量是否过多。
数组、List、map 数据过大。

Java（OOP）面向对象的三个特征与含义

封装 （高内聚低耦合 --> 解耦）

封装是指将某事物的属性和行为包装到对象中，这个对象只对外公布需要公开的属性和行为，而这个公布也是可以有选择性的公布给其它对象。在 java 中能使用 private、protected、public 三种修饰符或不用（即默认 default）对外部对象访问该对象的属性和行为进行限制。

java 的继承 （重用父类的代码）

继承是子对象可以继承父对象的属性和行为，亦即父对象拥有的属性和行为，其子对象也就拥有了这些属性和行为。

java 中的多态 （父类引用指向子类对象）

多态是指父对象中的同一个行为能在其多个子对象中有不同的表现。

有两种多态的机制：编译时多态、运行时多态。

1、方法的重载：重载是指同一类中有多个同名的方法，但这些方法有着不同的参数。，因此在编译时就可以确定到底调用哪个方法，它是一种编译时多态。
2、方法的重写：子类可以覆盖父类的方法，因此同样的方法会在父类中与子类中有着不同的表现形式。

Override 和 Overload 的含义去区别

重载 Overload 方法名相同，参数列表不同（个数、顺序、类型不同）与返回类型无关。
重写 Override 覆盖。将父类的方法覆盖。
重写方法重写：方法名相同，访问修饰符只能大于被重写的方法访问修饰符，方法签名个数，顺序个数类型相同。

Override（重写）

方法名、参数、返回值相同。
子类方法不能缩小父类方法的访问权限。
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）。
序在于父类和子类之间。
方法被定义为 final 不能被重写。
Overload（重载）

参数类型、个数、顺序至少有一个不相同。
不能重载只有返回值不同的方法名。
序在于父类和子类、同类中。
而重载的规则

1、必须具有不同的参数列表。
2、可以有不同的返回类型，只要参数列表不同就可以了。
3、可以有不同的访问修饰符。
4、可以抛出不同的异常。

重写方法的规则

1、参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。
2、返回的类型必须一直与被重写的方法的返回类型相同，否则不能称其为重写而是重载。
3、访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）。
4、重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。

例如：
父类的一个方法申明了一个检查异常 IOException，在重写这个方法是就不能抛出 Exception，只能抛出 IOException 的子类异常，可以抛出非检查异常。

Interface 与 abstract 类的区别

Interface 只能有成员常量，只能是方法的声明。
Abstract class 可以有成员变量，可以声明普通方法和抽象方法。

interface 是接口，所有的方法都是抽象方法，成员变量是默认的 public static final 类型。接口不能实例化自己。

abstract class 是抽象类，至少包含一个抽象方法的累叫抽象类，抽象类不能被自身实例化，并用 abstract 关键字来修饰。

Static class 与 non static class 的区别

static class（内部静态类）

1、用 static 修饰的是内部类，此时这个内部类变为静态内部类；对测试有用。
2、内部静态类不需要有指向外部类的引用。
3、静态类只能访问外部类的静态成员，不能访问外部类的非静态成员。

non static class（非静态内部类）

1、非静态内部类需要持有对外部类的引用。
2、非静态内部类能够访问外部类的静态和非静态成员。
3、一个非静态内部类不能脱离外部类实体被创建。
4、一个非静态内部类可以访问外部类的数据和方法。

foreach 与正常 for 循环效率对比

用 for 循环 arrayList 10 万次花费时间：5 毫秒。
用 foreach 循环 arrayList 10 万次花费时间：7 毫秒。
用 for 循环 linkList 10 万次花费时间：4481 毫秒。
用 foreach 循环 linkList 10 万次花费时间：5 毫秒。

循环 ArrayList 时，普通 for 循环比 foreach 循环花费的时间要少一点。
循环 LinkList 时，普通 for 循环比 foreach 循环花费的时间要多很多。

当我将循环次数提升到一百万次的时候，循环 ArrayList，普通 for 循环还是比 foreach 要快一点；但是普通 for 循环在循环 LinkList 时，程序直接卡死。

ArrayList：ArrayList 是采用数组的形式保存对象的，这种方式将对象放在连续的内存块中，所以插入和删除时比较麻烦，查询比较方便。

LinkList：LinkList 是将对象放在独立的空间中，而且每个空间中还保存下一个空间的索引，也就是数据结构中的链表结构，插入和删除比较方便，但是查找很麻烦，要从第一个开始遍历。

结论：

需要循环数组结构的数据时，建议使用普通for循环，因为for循环采用下标访问，对于数组结构的数据来说，采用下标访问比较好。

需要循环链表结构的数据时，一定不要使用普通for循环，这种做法很糟糕，数据量大的时候有可能会导致系统崩溃。

Java? IO 与 NIO

NIO是为了弥补IO操作的不足而诞生的，NIO的一些新特性有：非阻塞I/O，选择器，缓冲以及管道。管道（Channel），缓冲（Bubber），选择器（Selector）是其主要特征。

概念解释

Channel——管道实际上就像传统IO中的流，到任何目的地（或来自任何地方）的所有数据都必须通过一个Channel对象。一个Bubber实质上是一个容器对象。

每一种基本Java类型都有一种缓冲区类型：

ByteBubber——byte
CharBubber——char
ShortBubber——short
IntBubber——int
LongBubber——long
FloatBubber——bloat
DoubleBubber——double
Selector——选择器用于监听多个管道的事件，使用传统的阻塞IO时我们可以方便的知道什么时候可以进行读写，而使用非阻塞通道，我们需要一些方法来知道什么时候通道准备好了，选择器正是为这个需要而诞生的。

NIO和传统的IO有什么区别呢？

IO是面向流的，NIO是面向块（缓冲区）的。

IO面向流的操作一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。，导致了数据的读取和写入效率不佳。

NIO面向块的操作在一步中产生或者消费一个数据块。按块处理数据比按（流式的）字节处理数据要快得多，同时数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。通俗来说，NIO采取了“预读”的方式，当你读取某一部分数据时，他就会猜测你下一步可能会读取的数据而预先缓冲下来。

IO是阻塞的，NIO是非阻塞的

对于传统的IO，当一个线程调用read()或write()时，该线程被阻塞，直到有一些数据被

读取，或数据完全写入。该线程在此期间不能再干任何事情了。

而对于NIO，使用一个线程发送读取数据请求，没有得到响应之前，线程是空闲的，此时线程可以去执行别的任务，而不是像IO中那样只能等待响应完成。

NIO和IO适用场景

NIO是为弥补传统IO的不足而诞生的，但是凡有所短才有所长，NIO也有缺点，因为NIO是面向缓冲区的操作，每一次的数据处理都是对缓冲区进行的，那么就会有一个问题，在数据处理之前必须要判断缓冲区的数据是否完整或者已经读取完毕，如果没有，假设数据只读取了一部分，那么对不完整的数据处理没有任何意义。所以每次数据处理之前都要检测缓冲区数据。

那么NIO和IO各适用的场景是什么呢？

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，这时候用NIO处理数据可能是个很好的选择。

而如果只有少量的连接，而这些连接每次要发送大量的数据，这时候传统的IO更合适。使用哪种处理数据，需要在数据的响应等待时间和检查缓冲区数据的时间工作比较来权衡选择。

通俗解释，最后，对于NIO和传统IO

有一个网友讲的生动的例子：

以前的流总是堵塞的，一个线程只要对它进行操作，其它操作就会被堵塞，也就相当于水管没有阀门，你伸手接水的时候，不管水到了没有，你就都只能耗在接水（流）上。

nio的Channel的加入，相当于增加了水龙头（有阀门），虽然一个时刻也只能接一个水管的水，但依靠轮换策略，在水量不大的时候，各个水管里流出来的水，都可以得到妥

善接纳，这个关键之处就是增加了一个接水工，也就是Selector，他负责协调，也就是看哪根水管有水了的话，在当前水管的水接到一定程度的时候，就切换一下；临时关上当

前水龙头，试着打开另一个水龙头（看看有没有水）。

当其他人需要用水的时候，不是直接去接水，而是事前提了一个水桶给接水工，这个水桶就是Bubber。也就是，其他人虽然也可能要等，但不会在现场等，而是回家等，可以做

其它事去，水接满了，接水工会通知他们。

这其实也是非常接近当前社会分工细化的现实，也是统分利用现有资源达到并发效果的一种很经济的手段，而不是动不动就来个并行处理，虽然那样是最简单的，但也是最浪费资源的方式。

java 反射的作用于原理

什么是 Java 的反射呢？

Java 反射是可以让我们在运行时，通过一个类的 Class 对象来获取它获取类的方法、属性、父类、接口等类的内部信息的机制。

这种动态获取信息以及动态调用对象的方法的功能称为 JAVA 的反射。

反射的作用？

反射就是：在任意一个方法里：

1. 如果我知道一个类的名称/或者它的一个实例对象，我就能把这个类的所有方法和变量的信息找出来（方法名，变量名，方法，修饰符，类型，方法参数等等所有信息）

2. 如果我还明确知道这个类里某个变量的名称，我还能得到这个变量当前的值。

3. 当然，如果我明确知道这个类里的某个方法名+参数个数类型，我还能通过传递参数来运行那个类里的那个方法。

反射机制主要提供了以下功能：

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。
生成动态代理。
反射的原理？

JAVA 语言编译之后会生成一个.class 文件，反射就是通过字节码文件找到某一个类、类中的方法以及属性等。

反射的实现 API 有哪些？

反射的实现主要借助以下四个类：

Class：类的对象
Constructor：类的构造方法
Field：类中的属性对象
Method：类中的方法对象

泛型常用特点

List<String> 能否转为 List<Object>

不可以强转类型的

这个问题涉及到了，范型向上转型 和 范型向下转型问题。
List向上转换至List（等价于List）会丢失String类的身份（String类型的特有接口）。
当需要由List向下转型时，你的程序必须明确的知道将对象转换成何种具体类型，不然这将是不安全的操作。

如果要强转类型，Json 序列化转型

List<String> str = new ArrayList<String>();
List<Object> obj= JSONObject.parseArray(JSONObject.toJSONString(str));
或者遍历，或者克隆，但是取出来就是（Object）了，需要强转，String 因为类型丢了。

解析XML的几种方式的原理与特点：DOM、SAX

Android中主种常用解析XML的方式（DOM、SAX、PULL）简介及区别。

http://blog.csdn.net/cangchen...

xml解析的两种基本方式：DOM和SAX的区别是？

DOM：document object model。
SAX：simple api for xml。
dom一次性把xml文件全部加载到内存中简历一个结构一模一样的树，效率低。
SAX解析器的优点是解析速度快，占用内存少，效率高。

DOM在内存中以树形结构存放，因此检索和更新效率会更高。但是对于特别大的文档，解析和加载整个文档将会很耗资源。

DOM，它是生成一个树，有了树以后你搜索、查找都可以做。
SAX，它是基于流的，就是解析器从头到尾解析一遍xml文件，解析完了以后你不过想再查找重新解析。
sax解析器核心是事件处理机制。例如解析器发现一个标记的开始标记时，将所发现的数据会封装为一个标记开始事件，并把这个报告给事件处理器。

平时工作中，xml解析你是使用什么？

JDOM
DOM4J

Java1.7与1.8,1.9,10新特性

1.5

自动装箱与拆箱
枚举（常用来设计单例模式）
静态导入

可变参数
内省
1.6

Web 服务元数据
脚本语言支持
JTable 的排序和过滤
更简单，更强大的 JAX-WS
轻量级 Http Server
嵌入式数据库 Derby
1.7

switch 中可以使用字串了
运用 List tempList = new ArrayList<>(); 即泛型实例化类型自动推断
语法上支持集合，而不一定是数组
新增一些取环境信息的工具方法
Boolean 类型反转，空指针安全，参与位运算
两个 char 间的 equals
安全的加减乘除
map 集合支持并发请求，且可以写成 Map map = {name:"xxx",age:18};
1.8

允许在接口中有默认方法实现
Lambda 表达式
函数式接口
方法和构造函数引用
Lambda 的范围
内置函数式接口
Streams
Parallel Streams
Map
时间日期 API
Annotations
1.9

Jigsaw 项目；模块化源码
简化进程 API
轻量级 JSON API
钱和货币的 API
改善锁争用机制
代码分段缓存
智能 Java 编译，第二阶段
HTTP 2.0 客户端
Kulla 计划：Java 的 REPL 实现

10

本地变量类型推断
统一 JDK 仓库
垃圾回收器接口
G1 的并行 Full GC
应用程序类数据共享
ThreadLocal 握手机制
设计模式：单例、工厂、适配器、责任链、观察者等等
什么是设计模式

设计模式是一种解决方案，用于解决在软件设计中普遍存在的问题，是前辈们对之前软件设计中反复出现的问题的一个总结。

我们学设计模式，是为了学习如何合理的组织我们的代码，如何解耦，如何真正的达到对修改封闭对扩展开放的效果，而不是去背诵那些类的继承模式，然后自己记不住，回过头来就骂设计模式把你的代码搞复杂了，要反设计模式。

设计模式的六大原则

开闭原则：实现热插拔，提高扩展性。
里氏代换原则：实现抽象的规范，实现子父类互相替换；
依赖倒转原则：针对接口编程，实现开闭原则的基础；
接口隔离原则：降低耦合度，接口单独设计，互相隔离；
迪米特法则，又称不知道原则：功能模块尽量独立；
合成复用原则：尽量使用聚合，组合，而不是继承；
1、开闭原则（Open Close Principle）

开闭原则的意思是：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

2、里氏代换原则（Liskov Substitution Principle）

里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

3、依赖倒转原则（Dependence Inversion Principle）

这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

4、接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

5、迪米特法则，又称最少知道原则（Demeter Principle）

最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

6、合成复用原则（Composite Reuse Principle）

合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承。

JNI 的使用

https://www.cnblogs.com/larryzeal/p/5687392.html

JNI 是 Java Native Interface 的缩写，它提供了若干的 API 实现了 Java 和其他语言的通信（主要是 C&C++）。从 Java1.1 开始，JNI 标准成为 java 平台的一部分，它允许 Java 代码和其他语言写的代码进行交互。JNI 一开始是为了本地已编译语言，尤其是 C 和 C++而设计的，但是它并不妨碍你使用其他编程语言，只要调用约定受支持就可以了。使用 java 与本地已编译的代码交互，通常会丧失平台可移植性。

JNI 步骤

java 类中编写带有 native 声明的方法。
使用 javac 命令编译所编写的 java 类。
使用 javah 命令生成头文件。
使用 C/C++实现本地方法。
生成动态连接库。
执行（java）。
JNI 实例

```
public class HelloWorld {
    public native void displayHelloWorld();//所有 native 关键词修饰的都是对本地的声明
    static {
        System.loadLibrary("hello");//载入本地库
    }
    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
```

}

AOP是什么

AOP(Aspect Oriented Programming)面向切面编程，是目前软件开发中的一个热点，是Spring框架内容，利用AOP可以对业务逻辑的各个部分隔离，从而使的业务逻辑各部分的耦合性降低，提高程序的可重用性，踢开开发效率，主要功能：日志记录，性能统计，安全控制，事务处理，异常处理等。

AOP实现原理是java动态代理，但是jdk的动态代理必须实现接口，所以spring的aop是用cglib这个库实现的，cglis使用里asm这个直接操纵字节码的框架，所以可以做到不使用接口的情况下实现动态代理。

OOP是什么

OOP面向对象编程，针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。

AOP与OOP的区别

OOP面向对象编程，针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。而AOP则是针对业务处理过程中的切面进行提取，它所面对的是处理过程的某个步骤或阶段，以获得逻辑过程的中各部分之间低耦合的隔离效果。这两种设计思想在目标上有着本质的差异。

举例：

对于“雇员”这样一个业务实体进行封装，自然是OOP的任务，我们可以建立一个“Employee”类，并将“雇员”相关的属性和行为封装其中。而用AOP设计思想对“雇员”进行封装则无从谈起。

同样，对于“权限检查”这一动作片段进行划分，则是AOP的目标领域。

OOP面向名词领域，AOP面向动词领域。

总之AOP可以通过预编译方式和运行期动态代理实现在不修改源码的情况下，给程序动态统一添加功能的一项技术。