Part---------JAVA 基础专题

JVM 相关

1.说一下Java 的垃圾回收机制。

它使得 Java 程序员在编写程序的时候**不再需要考虑内存管理**。**垃圾回收器**通常是作为一个单独的低级别的线程运行，**不可预知的情况下**对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员**不能实时的调用垃圾回收器**对某个对象或所有对象进行垃圾回收。程序员可以手动执行 **System.gc()**，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

垃圾回收机制可以用 3 个词来概括：where，when 和 how？

Where：运行时的内存分布情况。见下一题。

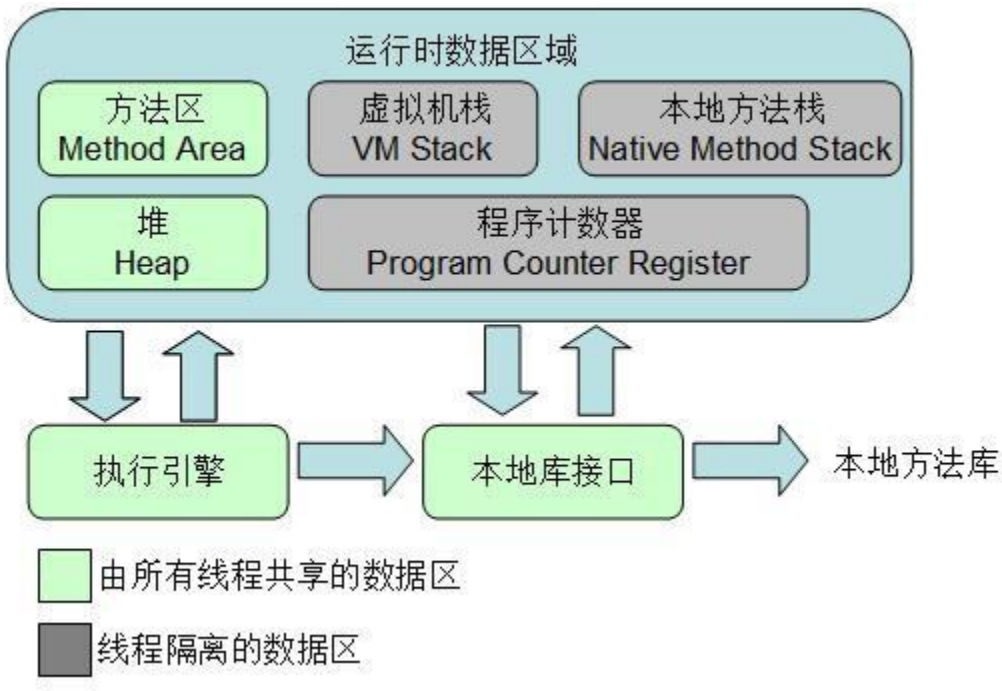
When：对象何时需要被回收的？也就是何时回收无效对象，已死对象的？

这里涉及到两种做法：引用计数法和可达性分析算法。这里还涉及到 java 中 4 种引用方式：强引用，软引用，弱引用和虚引用，其引用强度越来越来低，意味着引用越弱的对象越容易被垃圾回收的。

how：对象如何被回收的？4 种垃圾回收算法。

2.JVM 的内存布局/内存模型。（需要详细到每个区放什么）

区别于"**JAVA 的内存模型**"。



参见《深入理解 java 虚拟机》。

3.JVM 的 4 种引用和使用场景？

这 4 种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

⑴强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。 ps：强引用其实也就是我们平时 A a = new A()这个意思。

⑵软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。**软引用可用来实现内存敏感的高速缓存**（下文给出示例）。

软引用可以和一个**引用队列（ReferenceQueue）**联合使用，如果软引用所引用的对象被垃圾回收器回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。

示例：实现学生信息查询操作时有两套数据操作的方案。

一、将得到的信息存放在内存中，后续查询则直接读取内存信息（优点：读取速度快；缺点：内存空间一直被占，若资源访问量不高，则浪费内存空间）。

二、**每次查询均从数据库读取**，然后填充到 TO 返回。（优点：内存空间将被 GC 回收，不会一直被占用；缺点：在 GC 发生之前已有的 TO 依然存在，但还是执行了一次数据库查询，浪费 IO）。

可以通过软引用来解决。

⑶弱引用（WeakReference）

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。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个**引用队列（ReferenceQueue）**联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

⑷虚引用（PhantomReference）

“虚引用”顾名思义，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用，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。

```
ReferenceQueue queue = new ReferenceQueue ();
PhantomReference pr = new PhantomReference (object, queue);
```

比较容易理解的是 **Java 垃圾回收器会优先清理可达强度低的对象。**

那现在问题来了，**若一个对象的引用类型有多个**，那到底如何判断它的可达性呢？其实规则如下：（**“单弱多强”**）

1. **单条引用链**的可达性以**最弱**的一个引用类型来决定；
2. **多条引用链**的可达性以**最强**的一个引用类型来决定；

图 2 树形引用链

我们假设图 2 中引用①和③为强引用，⑤为软引用，⑦为弱引用，对于对象 5 按照这两个判断原则，路径①-⑤取最弱的引用⑤，**因此该路径对对象 5 的引用为软引用。**同样，③-⑦为弱引用。**在这两条路径之间取最强的引用**，于是**对象 5 是一个软可及对象**。

4. 说一下引用计数法与可达性分析算法。

参见《深入理解 java 虚拟机》。

扩展：GC 用的可达性分析算法中，哪些对象可作为 GC Roots 对象？

1.虚拟机栈中引用的对象。
2.本地方法栈中引用的对象。
3.方法区中静态成员或常量引用的对象。

5.如何判断对象是不是垃圾？

引用计数法与可达性分析算法。

6.堆里面的分区和各自的特点。

年轻代：年轻代又进一步可以划分为一个伊甸园(**Eden**)和两个存活区(**Survivor** space)，伊甸园是进行内存分配的地方，是一块连续的空闲内存区域，在里面进行内存分配速度非常快，因为不需要进行可用内存块的查找。新对象是总是在伊甸园中生成，只有经受住了一定的考验后才能后顺利地进入到存活区中，这种考验是什么在后面会讲到。把存活区划分为两块，其实也是为了满足垃圾回收的需要，因为在年轻代中经历了“回收大劫”未必就能够进入到年老代中。系统总是把对象放在伊甸园和一个存活区(任意的一个)，在垃圾回收时，根据其存活时间被复制到另一个存活区或者年老代中，则之前的存活区和伊甸园中剩下的都是需要被回收的对象，只对这两个区域进行清除即可，**两个存活区是交替使用，循环往复，在下一次垃圾回收时，之前被清除的存活区又用来放置存活下来的对象了。一般来说，年轻代区域较小，而且大部分对象是需要进行清除的，采用“复制算法”进行垃圾回收。**

年老代：在年轻代中经历了 N 次回收后仍然没有被清除的对象，就会被放到年老代中，**都是生命周期较长的对象。**对于年老代和永久代，**采用一种称为“标记-清除-压缩（Mark-Sweep-Compact）”的算法**。标记的过程是找出当前还存活的对象，并进行标记；清除则是遍历整个年老区，找到已标记的对象并进行清除；而压缩则是把存活的对象移动到整个内存区的一端，使得另一端是一块连续的空间，方便进行内存分配和复制。

1. **Minor GC**：当新对象生成，但在 Eden 申请空间失败时就会触发 **Minor** GC，对 Enden 区进行 GC，清除掉非存活的对象，并且把存活的对象移动到 Survivor 区中的其中一个区中。前面的提到考验就是 **Minor** GC，也就是说对象经过了 **Minor** GC 才能够进入到存活区中。这种形式的 GC 只会在年轻代中进行，因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配得太大，所以对 Eden 区的 GC 会**非常地频繁**。
2. **Full GC**：对整个对进行整理，包括了年轻代、年老代和持久代。Full GC 要对整个块进行回收，所以要比 **Minor** GC **慢得多**，因此应该尽可能减少 Full GC 的次数。

7.Minor GC 与 Full GC 分别在什么时候发生？

如果 Eden 空间占满了， 会触发 minor GC。 Minor GC 后仍然存活的对象会被复制到 S0 中去。这样 Eden 就被清空可以分配给新的对象。
又触发了一次 Minor GC ， S0 和 Eden 中存活的对象被复制到S1 中， 并且 S0 和 Eden 被清空。 在同一时刻, 只有 Eden 和一个 Survivor Space 同时被操作。
当每次对象从Eden 复制到Survivor Space 或者从Survivor Space 中的一个复制到另外一个，有一个计数器会自动增加值。 默认情况下如果复制发生超过 16

次， JVM 会停止复制并把他们移到老年代中去。

如果一个对象不能在 Eden 中被创建，它会直接被创建在老年代中。 如果老年代的空间被占满会触发老年代的 GC，也被称为 Full GC。Full GC 是一个压缩处理过程，所以它比 Minor GC 要慢很多。

***内存分配规则:**

1.对象优先分配在Eden区 如果Eden区没有足够的空间时 虚拟机执行一次Minor GC。

2.大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

3.长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过 了 1 次 Minor GC 那么对象会进入 Survivor 区，之后每经过一次 Minor GC 那么对象的年龄加 1，直到达到阀值，对象进入老年区。

4.动态判断对象的年龄。如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

5.**空间分配担保**。每次进行 Minor GC 时，JVM 会计算 Survivor 区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次 Full GC，如果小于检查 HandlePromotionFailure 设置 如果 true 则只进行 Monitor GC,如果 false 则进行 Full GC。

8.对象创建方法，对象的内存布局，对象的访问定位。

四种不同的方法创建 java 对象。

1.用 new 语句创建对象，这是最常用的创建对象的方式。

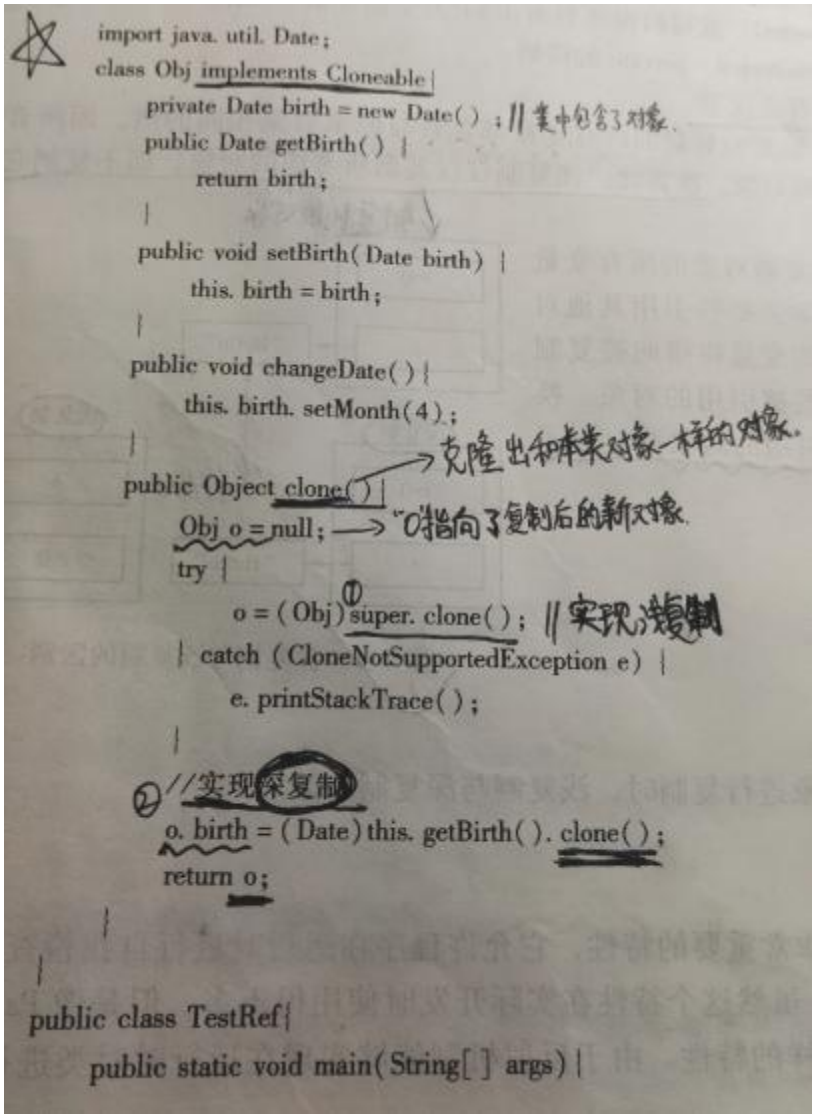
2.调用对象的 clone()方法。

```
MyObject anotherObject = new MyObject();
MyObject object = anotherObject.clone();
```

使用 clone()方法克隆一个对象的步骤：

1.被克隆的类要实现 Cloneable 接口。

2.被克隆的类要重写 clone( ) 方法。



```
import java.util.Date;
class Obj implements Cloneable{
    private Date birth = new Date();//集中包含对象.
    public Date getBirth(){
        return birth;
    }
    public void setBirth(Date birth){
        this.birth = birth;
    }
    public void changeDate(){
        this.birth.setMonth(4);
    }
    public Object clone(){
        Obj o = null;
        try{
            o = (Obj)super.clone();//实现浅复制
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        //实现深复制
        o.birth = (Date)this.getBirth().clone();
        return o;
    }
}
public class TestRef{
    public static void main(String[] args){
```

```
Obj a = new Obj();
Obj b = (Obj)a.clone();
b.changeDate();
System.out.println("a = " + a.getBirth());
System.out.println("b = " + b.getBirth());
```

程序运行结果为：

```
a = Sat Jul 13 23:58:56 CST 2013
b = Mon May 13 23:58:56 CST 2013
```

那么在编程时，如何选择使用哪种复制方式呢？首先，检查类有无非基本类型（即对象）的数据成员。①若没有，则返回 super.clone()即可；②若有，确保类中包含的所有非基本类型的成员变量都实现了深复制。

```
Object o = super.clone(); //先执行浅复制
对每一个对象 attr 执行以下语句：
o.attr = this.getAttr().clone();
最后返回 o。
```

需要注意的是，clone()方法的保护机制在 Object 中 clone()是被声明为 protected 的。以 User 类为例，通过声明为 protected，就可以保证只有 User 类里面才能“克隆” User 对象，原理可以参考前面关于 public、protected、private 的讲解。

引申：浅复制和深复制有什么区别？

浅复制（Shallow Clone）：被复制对象的所有变量都含有与原来对象相同的值，而所有其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它引用的对象。

深复制（Deep Clone）：被复制对象的所有变量都含有与原来对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制的新对象，而不再是原有的那些被引用的对象。换言之，深复制把复制的对象所引用的对象都复制了一遍。

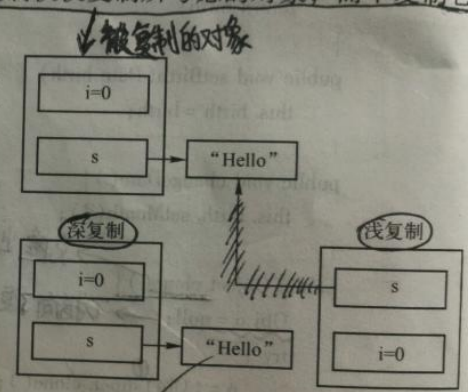


图 4-1 深复制与浅复制的区别

假如定义如下一个类。

```
class Test{
    public int i;
    public StringBuffer s;
}
```

扩展：

原型模式**主要用于对象的复制**，实现一个接口（实现 Cloneable 接口），重写一个方法（重写 Object 类中的 clone 方法），即完成了原型模式。

原型模式中的拷贝分为"浅拷贝"和"深拷贝"：

浅拷贝: 对值类型的成员变量进行**值的复制**,对引用类型的成员变量只复制

引用,不复制引用的对象.

深拷贝: 对值类型的成员变量进行**值的复制,**对引用类型的成员变量也进行引用对象的复制.

（Object 类的clone 方法只会拷贝对象中的基本数据类型的值，对于数组、容器对象、引用对象等都不会拷贝，这就是浅拷贝。如果要实现深拷贝，必须将原型模式中的数组、容器对象、引用对象等另行拷贝。）

原型模式的优点。

1.如果创建新的对象比较复杂时，可以利用原型模式**简化对象的创建过程。**

2.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。

原型模式的使用场景。

因为以上优点，所以**在需要重复地创建相似对象时可以考虑使用原型模式。**比如需要在一个循环体内创建对象，假如对象创建过程比较复杂或者循环次数很多的话，使用原型模式不但可以简化创建过程，而且可以使系统的整体性能提高很多。

3.运用反射手段，使用 Class.forName()

```
MyObject object = (MyObject) Class.forName("subin.rnd.MyObject
").newInstance();
```

其他部分见“反射的原理”部分。

4.运用反序列化手段，调用 java.io.ObjectInputStream 对象的 readObject()方法。

其余部分见“序列化和反序列化”部分。

其余 2 问参见《深入理解 java 虚拟机》。

9.说一下几种垃圾收集算法的原理和特点，应用的场景。怎么优化复 制算法？

参见《深入理解 java 虚拟机》。

优化复制算法：由于每次执行复制算法的时候，所有存活的对象都要被复制，这样效率很低。由于程序中创建的大部分对象的生命周期都很短，只有一

内存泄漏的典型例子是一个没有重写 hashCode 和 equals 方法的 Key 类在 HashMap 中保存的情况，最后会生成很多重复的对象。所有的内存泄露最后都会抛出 OutOfMemoryError 异常（Exception in thread "main" java.lang.OutOfMemoryError: Java heap space）。

造成内存泄露的原因：

在 Java 语言中，容易引起内存泄露的原因很多，主要有以下几个方面的内容：

1）静态集合类，例如 HashMap 和 Vector。如果这些容器为静态的，由于它们的生命周期与程序一致，那么容器中的对象在程序结束之前将不能被释放，从而造成内存泄露，如下例所示。

2）各种连接，例如数据库连接、网络联接以及 IO 连接等。在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用 close 方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对 Connection、Statement 或 ResultSet 不显式地关闭，将会造成大量的对象无法被回收，从而引起内存泄露。

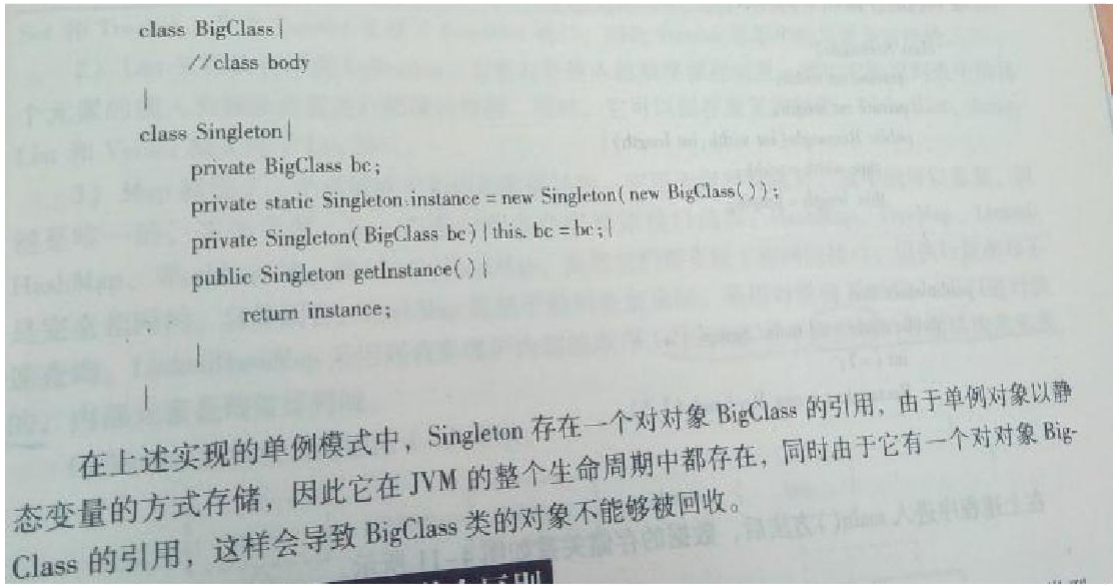
3）监听器。在 Java 语言中，往往会使用到监听器。通常一个应用中会用到多个监听器，但在释放对象的同时往往没有相应地删除监听器，这也可能导致内存泄露。

4）变量不合理的作用域。一般而言，如果一个变量定义的作用范围大于其使用范围，很有可能会造成内存泄露，另一方面如果没有及时地把对象设置为 null，很有可能会导致内存泄露的发生，示例如下：

```
class Server{
    private String msg;
    public void recieveMsg(){
        readFromNet();//从网络接收数据保存到 msg 中
        saveDB();     //把 msg 保存到数据库中
    }
}
```


在上述伪代码中，通过 readFromNet( ) 方法接收的消息保存在变量 msg 中，然后调用 saveDB( ) 方法把 msg 的内容保存到数据库中，此时 msg 已经没用了，但是由于 msg 的生命周期与对象的生命周期相同，此时 msg 还不能被回收，因此造成了内存泄露。对于这个问题，有如下两种解决方法：第一种方法，由于 msg 的作用范围只在 recieveMsg( ) 方法内，因此可以把 msg 定义为这个方法的局部变量，当方法结束后，msg 的生命周期就会结束，此时垃圾回收器就可以回收 msg 的内容了；第二种方法，在使用完 msg 后就把 msg 设置为 null，这样垃圾回收器也会自动回收 msg 内容所占的内存空间。

5）单例模式可能会造成内存泄露。单例模式的实现方法有很多种，下例中所使用的单例模式就可能会造成内存泄露：



```
class BigClass{
    //class body
}
class Singleton{
    private BigClass bc;
    private static Singleton instance = new Singleton(new BigClass());
    private Singleton(BigClass bc){this.bc = bc;}
    public Singleton getInstance(){
        return instance;
    }
}
```

在上述实现的单例模式中，Singleton 存在一个对对象 BigClass 的引用，由于单例对象以静态变量的方式存储，因此它在 JVM 的整个生命周期中都存在，同时由于它有一个对对象 BigClass 的引用，这样会导致 BigClass 类的对象不能够被回收。

内存泄露的解决方案(重要！！)：

1、避免在循环中创建对象。

2、尽早释放无用对象的引用。（最基本的建议）

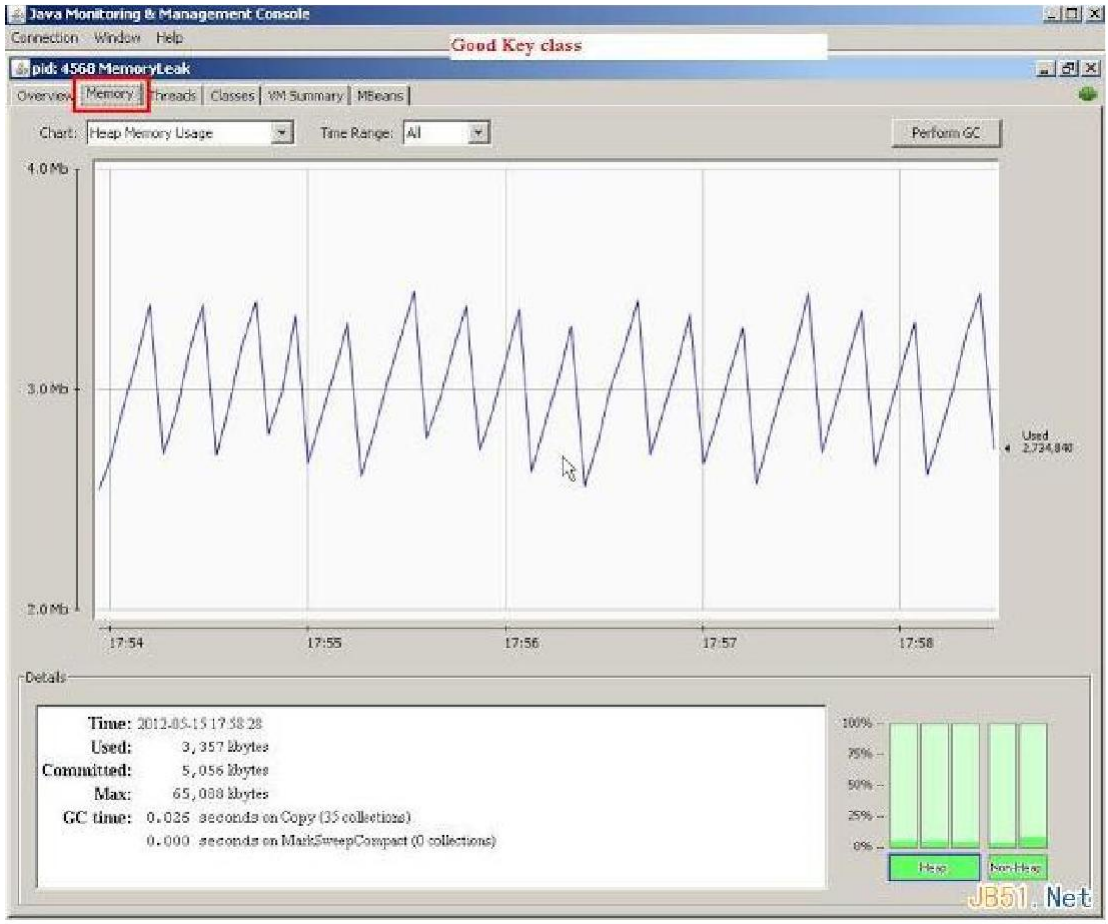
3、**尽量少用静态变量**，因为静态变量存放在永久代（方法区），永久代基本不参与垃圾回收。

4、使用字符串处理，避免使用 String，**应大量使用 StringBuffer**，每一个String对象都得独立占用内存一块区域。

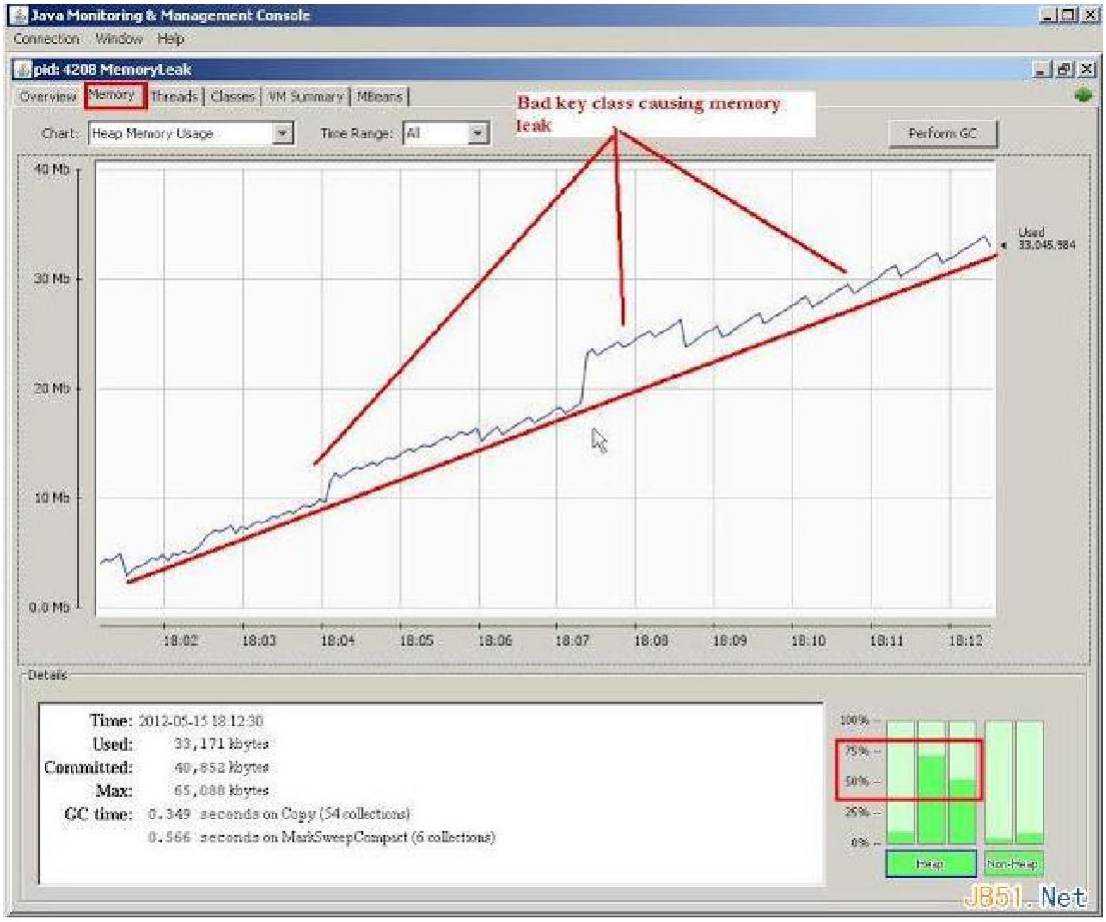
在实际场景中，你怎么查找内存泄露？

可以使用 Jconsole。

没有内存泄露的：



造成内存泄露的：如果内存的大小持续地增长，则说明系统存在内存泄漏。



内存溢出：指程序运行过程中**无法申请到足够的内存**而导致的一种错误。

内存溢出的几种情况（OOM 异常）：

OutOfMemoryError 异常：

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。

1.虚拟机栈和本地方法栈溢出

如果线程请求的栈深度大于虚拟机所允许的最大深度， 将抛出 StackOverflowError 异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间， 则抛出 OutOfMemoryError 异常。

2.堆 溢出

一般的异常信息：java.lang.OutOfMemoryError:Java heap spaces。

出现这种异常，一般手段是先通过**内存映像分析工具**(如 Eclipse Memory Analyzer)对 dump 出来的**堆转存快照**进行分析，重点是确认内存中的对象是否是必要的， 先分清是因为内存泄漏(Memory Leak) 还是内存溢出(Memory Overflow)。

如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄漏对象是通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收。

如果不存在泄漏，那就应该检查虚拟机的参数(-Xmx 与-Xms)的设置是否适当。

3.方法区溢出

异常信息：java.lang.OutOfMemoryError:**PermGen space**。

4.运行时常量池溢出

异常信息：java.lang.OutOfMemoryError:**PermGen space**。

如果要向运行时常量池中添加内容，最简单的做法就是使用 String.intern()这个 Native 方法。该方法的作用是：如果池中已经包含一个等于此 String 的字符串，则返回代表池中这个字符串的 String 对象；否则，将此 String 对象包含的字符串添加到常量池中，并且返回此 String 对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize 和 -XX:MaxPermSize 限制方法区的大小，从而间接限制其中常量池的容量。

导致内存溢出的原因：

1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；
2.集合类中有对对象的引用，使用完后未清空，使得 JVM 不能回收；
3.代码中存在死循环或循环产生过多重复的对象实体；
4.启动参数内存值设定的过小。

内存溢出的解决方法：

第一步，修改 JVM 启动参数，直接增加内存。(-Xms，-Xmx 参数一定不要忘记加。一般要将-Xms 和-Xmx 选项设置为相同，以避免在每次 GC 后调整堆的大小；建议堆的最大值设置为可用**内存的最大值的 80%**)。
第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。
第三步，对代码进行走查和分析，找出可能发生内存溢出的位置。

第四步，使用内存查看工具动态查看内存使用情况（Jconsole）。

扩展：SOF 你遇到过哪些情况？

基本上如果抛出 OutOfMemory 有两种原因：1.内存泄露。2.应用程序本身就是需要这么多的内存。

12.如何减少gc 出现的次数（java 内存管理）。（重点！！！！）

(1)对象不用时最好显式置为 Null

一般而言,为 Null 的对象都会被作为垃圾处理,所以将不用的对象显式地设为 Null,有利于 GC 收集器判定垃圾,从而提高了 GC 的效率。

(2)尽量少用 System.gc()

此函数建议JVM 进行主GC,虽然只是建议而非一定,但很多情况下它会触发主 GC,从而增加主 GC 的频率,也即增加了间歇性停顿的次数。

(3)尽量少用静态变量

静态变量属于全局变量,不会被 GC 回收,它们会一直占用内存。

（4）尽量使用 StringBuffer,而不用String 来累加字符串。

由于 String 是固定长的字符串对象,累加 String 对象时,并非在一个 String 对象中扩增,而是重新创建新的 String 对象,如 Str5=Str1+Str2+Str3+Str4,这条语句执行过程中会产生多个垃圾对象,因为对次作“+”操作时都必须创建新的 String 对象,但这些过渡对象对系统来说是没有实际意义的,只会增加更多的垃圾。避免这种情况可以改用 StringBuffer 来累加字符串,因 StringBuffer 是可变长的,它在原有基础上进行扩增,不会产生中间对象。

(5)分散对象创建或删除的时间

集中在短时间内大量创建新对象,特别是大对象,会导致突然需要大量内存,JVM 在面临这种情况时,只能进行主 GC,以回收内存或整合内存碎片,从而增加主 GC 的频率。

集中删除对象,道理也是一样的。它使得突然出现了大量的垃圾对象,空闲空间必然减少,从而大大增加了下一次创建新对象时强制主 GC 的机会。

（6） 尽量少用 finalize 函数。因为它会加大 GC 的工作量，因此尽量少用 finalize 方式回收资源。

（7） 如果需要使用经常用到的图片，可以使用软引用类型，它可以尽可能将图片保存在内存中，供程序调用，而不引起 OutOfMemory。

(8)能用基本类型如 int,long,就不用 Integer,Long 对象

基本类型变量占用的内存资源比相应包装类对象占用的少得多,如果没有必要,最好使用基本变量。

（9）增大-Xmx 的值。

13. 数组多大放在 JVM 老年代？ 永久代对象如何 GC？如果想不被 GC 怎么办？如果想在GC 中生存 1 次怎么办？

虚拟机提供了一个**-XX:PretenureSizeThreshold 参数（通常是 3MB）**，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存复制（复习一下：新生代采用复制算法收集内存）。

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(FullGC)。如果仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免 FullGC 是非常重要的原因。

让对象实现 finalize()方法，一次对象的自我拯救。

14.JVM 常见的启动参数。

-Xms：设置**堆的最小值**。

-Xmx：设置**堆的最大值**。

-Xmn: 设置**新生代的大小**。

-Xss：设置**每个线程的栈大小**。

-XX:NewSize：设置**新生代的初始值**。

-XX:MaxNewSize ：设置**新生代的最大值**。

-XX:PermSize：设置**永久代的初始值**。

-XX:MaxPermSize：设置**永久代的最大值**。

-XX:SurvivorRatio：年轻代中 **Eden 区与Survivor 区的大小比值**。

-XX:PretenureSizeThreshold：令大于这个设置值的对象直接在老年代分配。

15.说下几种常用的内存调试工具：jps、jmap、jhat、jstack、jconsole，jstat。

Java 内存泄露的问题调查定位：jmap，jstack 的使用等等。

jps:查看虚拟机进程的状况，如进程ID。

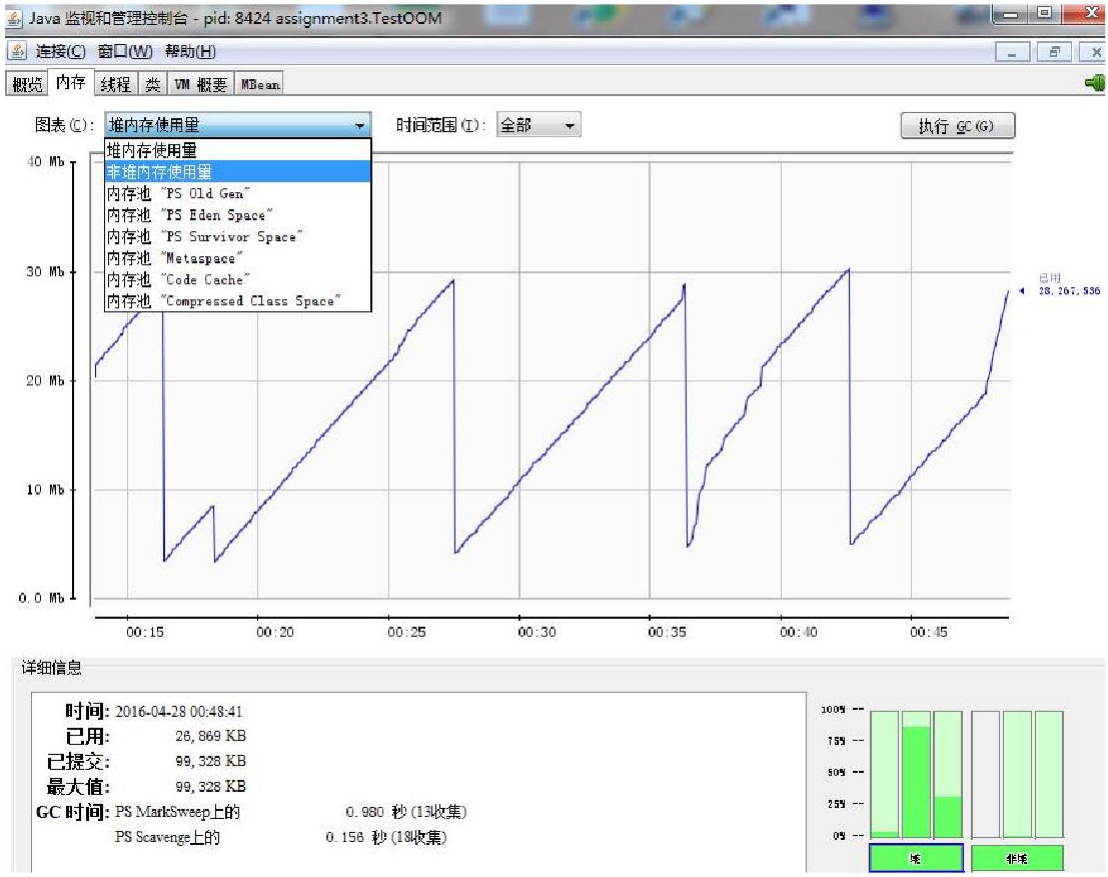
jmap:　用于生成**堆转储快照文件（某一时刻的）**。

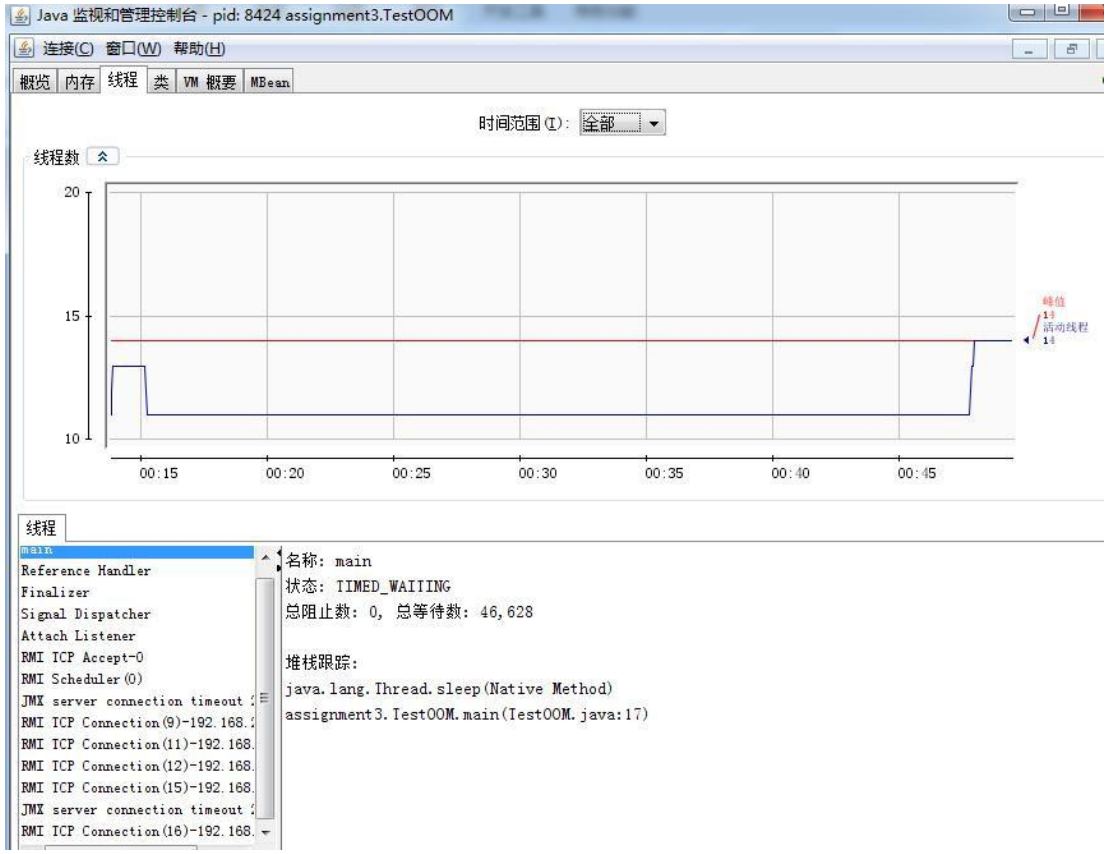
jhat：对生成的堆转储快照文件进行分析。

jstack:用来生成**线程快照（某一时刻的）**。生成线程快照的主要目的是定位线程长时停顿的原因（如死锁，死循环，等待I/O 等），通过查看各个线程的调用堆栈，就可以知道没有响应的线程在后台做了什么或者等待什么资源。

jstat:虚拟机**统计信息**监视工具。如显示垃圾收集的情况，内存使用的情况。

Jconsole:主要是内存监控和线程监控。内存监控：可以显示内存的使用情况。线程监控：遇到线程停顿时，可以使用这个功能。


Java 监视和管理控制台 - pid: 8424 assignment3.TestOOM
连接(C) 窗口(W) 帮助(H)
概览 内存 线程 类 VM 概要 MBean
图表(C): 堆内存使用量
时间范围(T): 全部
执行 GC(G)
堆内存使用量
非堆内存使用量
内存池 "PS Old Gen"
内存池 "PS Eden Space"
内存池 "PS Survivor Space"
内存池 "Metaspace"
内存池 "Code Cache"
内存池 "Compressed Class Space"
40 Mb
30 Mb
20 Mb
10 Mb
0.0 Mb
00:15
00:20
00:25
00:30
00:35
00:40
00:45
已用
28,267,536
详细信息
时间: 2016-04-28 00:48:41
已用: 26,869 KB
已提交: 99,328 KB
最大值: 99,328 KB
GC 时间: PS MarkSweep上的 0.980 秒 (13收集)
PS Scavenge上的 0.156 秒 (18收集)
100%
75%
50%
25%
0%
堆
非堆



16. 说下虚拟机的类加载机制。

参见《深入理解 java 虚拟机》。

17. 说下双亲委派模型。双亲委派模型中有哪些方法。用户如何自定义类加载器 。怎么打破双亲委托机制？

参见《深入理解 java 虚拟机》。

双亲委派模型中用到的方法：

findLoadedClass(),

loadClass()

findBootstrapClassOrNull()

findClass()

defineClass()：把二进制数据转换成字节码。

resolveClass()

自定义类加载器的方法：继承 ClassLoader 类,重写 findClass()方法 。

18.描述Java 类加载器的工作原理及其组织结构。

Java 类加载器的作用就是在运行时加载类。

Java 类加载器基于三个机制：**委托性、可见性和单一性。**

1.委托机制是指双亲委派模型。当一个类加载和初始化的时候，类仅在有需要加载的时候被加载。假设你有一个应用需要的类叫作 Abc.class，首先加载这个类的请求由 Application 类加载器委托给它的父类加载器Extension 类加载器，然后再委托给 Bootstrap 类加载器。Bootstrap 类加载器 会先看看 **rt.jar** 中有没有这个类，因为并没有这个类，所以这个请求又回到 Extension 类加载器，它会查看 **jre/lib/ext 目录下**有没有这个类，如果这个类被 Extension 类加载器找到了， 那么它将被加载，而 Application 类加载器不会加载这个类；而如果这个类没有被 Extension 类加载器找到，那么再由 Application 类加载器从 **classpath 中**寻找， 如果没找到，就会抛出异常。

双亲委托机制的优点就是能够提高软件系统的**安全性**。因为在此机制下，用户自定义的类加载器不可能加载本应该由父加载器加载的可靠类，从而防止不可靠的恶意代码代替由父类加载器加载的可靠代码。如，java.lang.Object 类总是由根类加载器加载的，其他任何用户自定义的类加载器都不可能加载含有恶意代码的 java.lang.Object 类。

2.可见性原理是子类的加载器可以看见所有的父类加载器加载的类，而父类加载器看不到子类加载器加载的类。

3.单一性原理是指仅加载一个类一次，这是由委托机制确保子类加载器不会再次加载父类加载器加载过的类。

Java 的类加载器有三个，对应 Java 的三种类:

Bootstrap Loader // 负责加载**系统类** (指的是**内置类**，像 String)

|

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

|

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

Java 提供了**显式加载类**的 API：**Class.forName**(classname)。

21.Java 编译的过程。

参见《深入理解 java 虚拟机》。P304

22. class 文件是什么类型文件（字节码文件的格式）？

参见《深入理解 java 虚拟机》。

23. 即时编译器的优化方法。P345

参见《深入理解 java 虚拟机》。

24. 静态分派与动态分派。

参见《深入理解 java 虚拟机》。P246

25. 编译阶段对程序做了哪些优化？

编译期其实是个“不确定”的操作过程，它既可能指的是前端编译器的编译过程，也可能指的是后端运行期编译器的编译过程。详细参见《深入理解 java 虚拟机》。

26. new 的对象如何不分配在堆而分配在栈上？

方法逃逸。

JAVA 中的集合类

1. ArrayList、LinkedList、Vector 的区别和实现原理。

ArrayList 和 Vector 只能按顺序存储元素（从下标为 0 的位置开始），删除元素的时候，需要移位并置空，**默认初始容量都是 10**。

ArrayList 和 Vector 基于数组实现的，LinkedList 基于**双向循环链表**实现的（含有头结点）。

一.线程安全性

ArrayList 不具有线程安全性，用在单线程环境中。LinkedList 也是线程不安全的，**如果在并发环境下使用它们，** 可以用 Colletions 类中的静态方法 **synchronizedList()** 对 ArrayList 和 LinkedList 进行调用即可。

Vector 是线程安全的，即它的大部分方法都包含有关键字 synchronized。Vector 的效率没有 ArrayList 和 LinkedList 高。

二.扩容机制

从内部实现机制来讲，ArrayList 和 Vector 都是使用 Objec 的数组形式来存储的。当你向这两种类型中增加元素的时候，若容量不够，需要进行扩容。ArrayList 扩容后的容量是之前的 1.5 倍，然后，**把之前的数据拷贝到新建的数组**。Vector 默认情况下扩容后的容量是之前的 2 倍。

Vector 可以**设置容量增量**，而 ArrayList 不可以。在 Vector 中有 capacityIncrement：向量的大小大于其容量时， 容量自动增加的量。如果在创建 Vector 时， 指定了 capacityIncrement 的大小；则每次当 Vector 中动态数组容量需要增加时，如果**容量的增量大于零**，增加的大小都是 capacityIncrement。如果**容量的增量小于等于零**，则每次需要增大容量时，向量的容量将增大为之前的 2 倍。

可变长度数组的原理：当元素个数超过数组的长度时，会产生一个新数组，将原数组的数据复制到新数组，再将新的元素添加到新数组中。

三.增删查改的效率

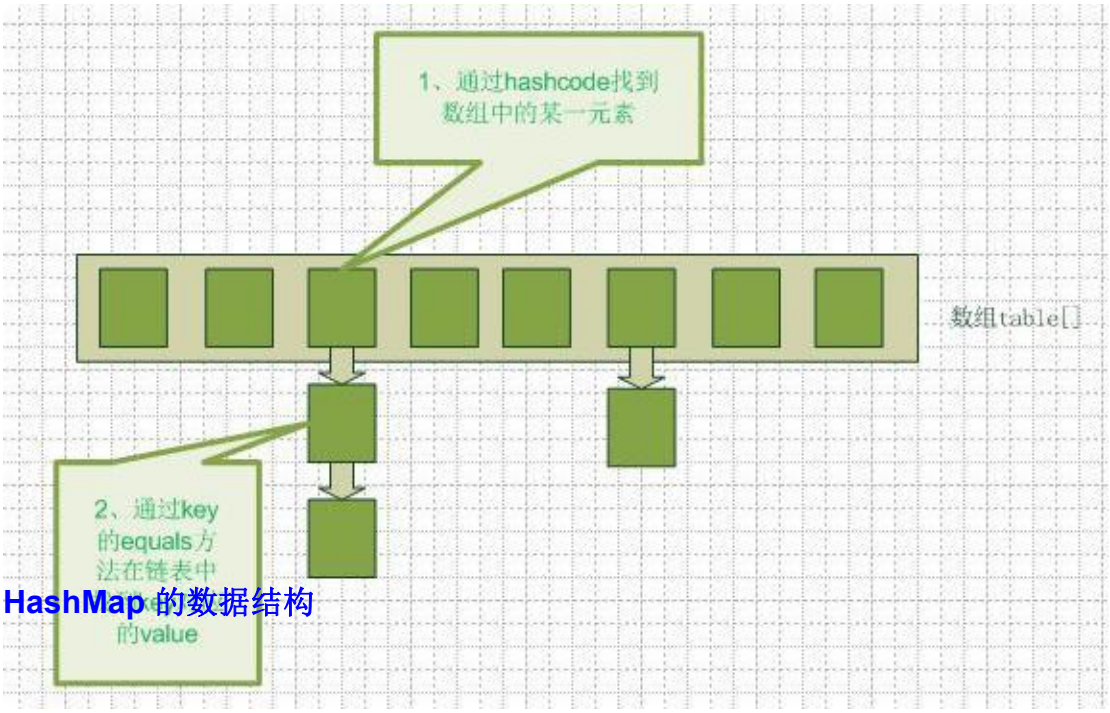
ArrayList 和 Vector 中，**从指定的位置（用 index）检索一个对象，或在集合的末尾插入、删除一个对象的时间是一样的，可表示为 O(1)**。但是，如果在集合的其他位置增加或移除元素那么花费的时间是 O（n）。LinkedList 中，在插入、删除集合中任何位置的元素所花费的时间都是一样的—O(1)，但它在索引一个元素的时候比较慢 O（n）。

所以，如果只是查找特定位置的元素或只在集合的末端增加、移除元素，那么使用 Vector 或 ArrayList 都可以 如果是对其它**指定位置的插入 删除操作** 最好选择LinkedList

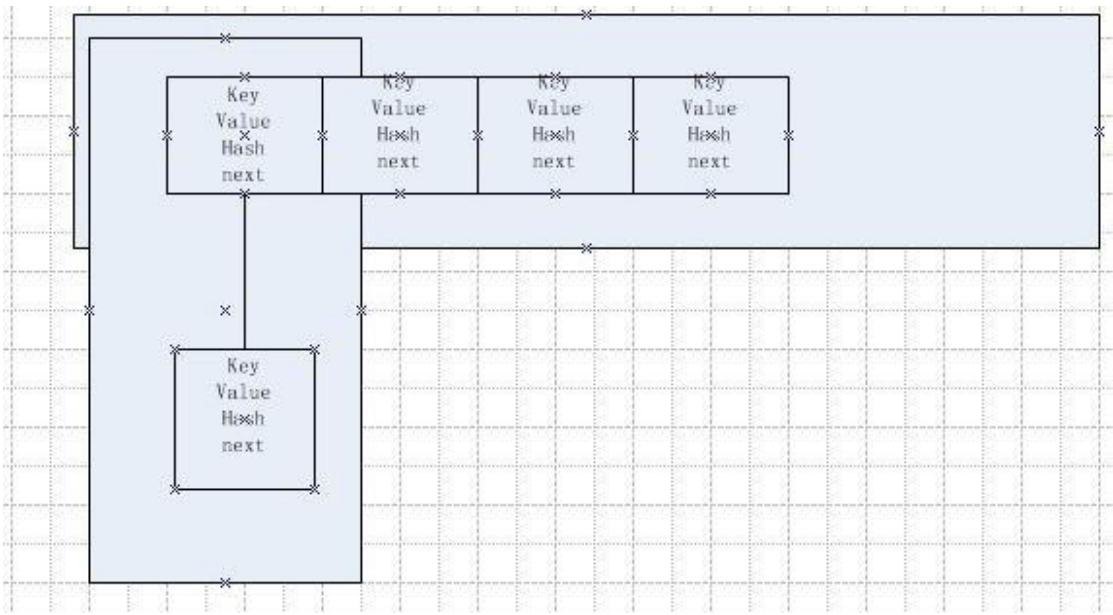
2.HashMap、HashTable、LindedHashMap、ConcurrentHashMap、WeakHashMap 的区别和实现原理。

一． HashMap VS HashTable

1.首先说下 HashMap 的原理。



HashMap 的数据结构



```
/**
 * The table, resized as necessary. Length MUST Always be a power of two.
 */
transient Entry[] table;

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

HashMap 存储函数的实现 put(K key, V value)：

根据下面 put 方法的源代码可以看出，当程序试图将一个 key-value 对放入 HashMap 中时，程序首先计算该 key 的 hashCode() 值，然后对该哈希码值进行再哈希，然后把哈希值和（数组长度-1）进行按位与操作，得到存储的数组下标，如果该位置处没有链表节点，那么就直接把包含<key,value>的节点放入该位置。如果该位置有结点，就对链表进行遍历，看是否有 hash，key 和要放入的节点相同的节点，如果有的话，就替换该节点的 value 值，如果没有相同的话，就创建节点放入值，**并把该节点插入到链表表头(头插法)**。

```
public V put(K key, V value) {
    // HashMap允许存放null键和null值。
    // 当key为null时，调用putForNullKey方法，将value放置在数组第一个位置。
    if (key == null)
        return putForNullKey(value);
    // 根据key的keyCode重新计算hash值。
    int hash = hash(key.hashCode());
    // 搜索指定hash值在对应table中的索引。
    int i = indexFor(hash, table.length);
    // 如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素。
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    // 如果i索引处的Entry为null，表明此处还没有Entry。
    modCount++;
    // 将key、value添加到i索引处。
    addEntry(hash, key, value, i);
    return null;
}
```

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    // 获取指定 bucketIndex 索引处的 Entry
    Entry<K,V> e = table[bucketIndex];
    // 将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    // 如果 Map 中的 key-value 对的数量超过了极限
    if (size++ >= threshold)
    // 把 table 对象的长度扩充到原来的2倍。
        resize(2 * table.length);
}
```

```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY)
        { threshold = Integer.MAX_VALUE;
        return;
    }
    Entry[] newTable = new Entry[newCapacity];//新创建一个数组
    transfer(newTable);
    table = newTable;
    threshold = (int) (newCapacity * loadFactor);
}
```

```
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++)
    { Entry<K, V> e = src[j];
    if (e != null)
        { src[j] =
        null; do {
            //过程(1)
            Entry<K, V> next = e.next;
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        } while (e != null);
        }
    }
```

}

```
static int indexFor(int h, int length)
        { return h & (length-1);
}
```

```
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

再哈希，其目的是为了减少哈希冲突，使元素能够均匀的分布在数组上，从而提高数组的存取效率。

扩展：为何数组的长度是 2 的 n 次方呢？

1.这个方法非常巧妙， 它通过 h & (table.length -1) 来得到该对象的保存位， 而 HashMap 底层**数组的长度总是 2 的 n 次方**，2^n-1 得到的二进制数的每个位上的值都为 1，那么**与全部为 1 的一个数进行与操作，速度会大大提升**。

2.当 length 总是 2 的 n 次方时， h& (length-1)运算等价于对 length 取模，也就是 h%length，但是**&比%具有更高的效率。**

3.当数组长度为 2 的 n 次幂的时候，**不同的 key 算得的 index 相同的几率较小**，那么数据在数组上分布就比较均匀，也就是说**碰撞的几率小**，相对的，查询的时候就不用遍历某个位置上的链表，这样**查询效率也就较高了**。

HashMap 读取函数的实现 get(Object key)：

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
        e != null;
        e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
    return null;
}
```

hashMap 的 get 方法，是首先通过 key 的两次 hash 后的值与数组的长度-1 进行与操作，定位到数组的某个位置，然后对该列的链表进行遍历,一般情况下，hashMap 的这种查

找速度是非常快的，hash 值相同的元素过多，就会造成链表中数据很多，而链表中的数据查找是通过遍历所有链表中的元素进行的，这可能会影响到查找速度，找到即返回。特别注意：**当返回为 null 时，你不能判断是没有找到指定元素，还是在 hashmap中存着一个 value 为 null 的元素，因为 hashmap 允许 value 为 null.**

HashMap 的扩容机制：

当 HashMap 中的结点个数超过数组大小*loadFactor（加载因子）时，就会进行数组扩容，loadFactor 的默认值为 0.75。也就是说，默认情况下，数组大小为 16，那么当 HashMap 中结点个数超过 16*0.75=12 的时候，就把数组的大小扩展为 2*16=32，即扩大一倍，**然后重新计算每个元素在数组中的位置，并放进去，而这是一个非常消耗性能的操作**。

多线程下 HashMap 出现的问题：

1. 多线程 put 操作后，get 操作导致死循环,导致 cpu100%的现象。主要是多线程同时 put 时，如果同时触发了 rehash 操作，会导致扩容后的 HashMap 中的链表中出现循环节点，进而使得后面 get 的时候，会死循环。

2. 多线程 put 操作，导致元素丢失，也是发生在多个线程对 hashmap 扩容时。

2.hashTable 的原理。

它的原理和 hashMap 基本一致。

3.HashMap 和 HashTable 的区别。

1. Hashtable 是线程安全的，方法是 Synchronized 的，适合在多线程环境中使用，效率稍低；HashMap 不是线程安全的，方法不是 Synchronized 的，效率稍高，适合在单线程环境下使用，所以在多线程场合下使用的话，需要手动同步 HashMap，Collections.synchronizedMap()。

PS:HashTable 的效率比较低的原因？

在线程竞争激烈的情况下 HashTable 的效率非常低下。因为当一个线程访问 HashTable 的同步方法时，访问其他同步方法的线程就可能会进入阻塞或者轮训状态。如线程 1 使用 put 进行添加元素，线程 2 不但不能使用 put 方法添加元素，并且也不能使用 get 方法来获取元素，所以竞争越激烈效率越低。

2.**HashMap 的 key 和 value 都可以为 null 值，**HashTable 的 key 和 value 都不允许有 Null 值。

3.HashMap 中数组的**默认大小是 16**，而且**一定是 2 的倍数，**扩容后的数组长度是之前数组长度的 **2 倍**。HashTable 中数组**默认大小是 11**，扩容后的数组长度是之前数组长度的 **2 倍+1**。

4.**哈希值的使用不同。**

而 HashMap 重新计算 hash 值，而且**用&代替求模：**

```
int hash = hash(key.hashcode());
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);    //hashmap 的表长永远是 2^n。
}
```

HashTable 直接使用对象的 hashCode 值：

```
int hash = key.hashCode();    //注意区分 2 者的 hash 值！！
int index = (hash & 0x7FFFFFFF) % tab.length;
```

5.判断是否含有某个键

在 HashMap 中，null 可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为 null。**当 get()方法返回 null 值时，既可以表示 HashMap 中没有该键，也可以表示该键所对应的值为 null。**因此，在 HashMap 中**不能用 get()方法来判断 HashMap 中是否存在某个键**，而应该用 **containsKey()**方法来判断。Hashtable 的键值都不能为 null，所以可以用 get()方法来判断是否含有某个键。

二．ConcurrentHashMap 的原理。

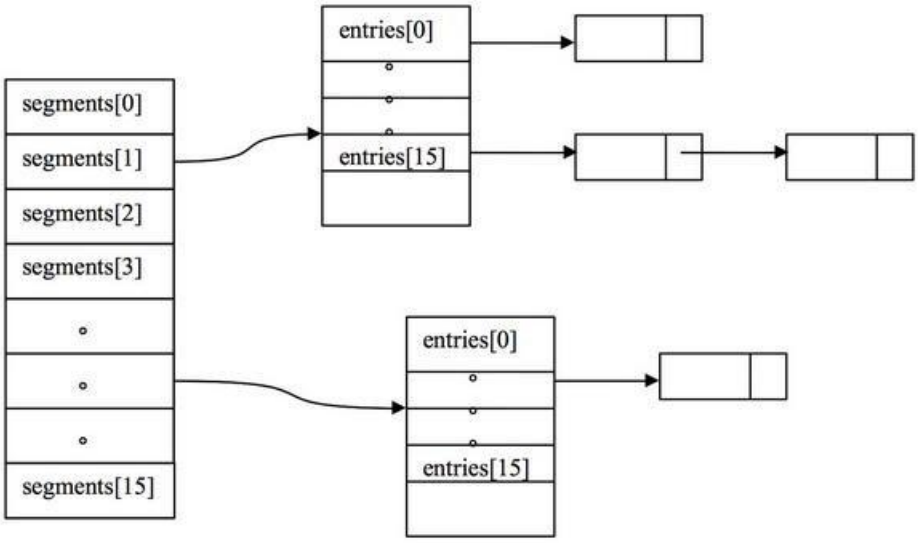
在 ConcurrentHashMap 中，不允许用 null 作为键和值。

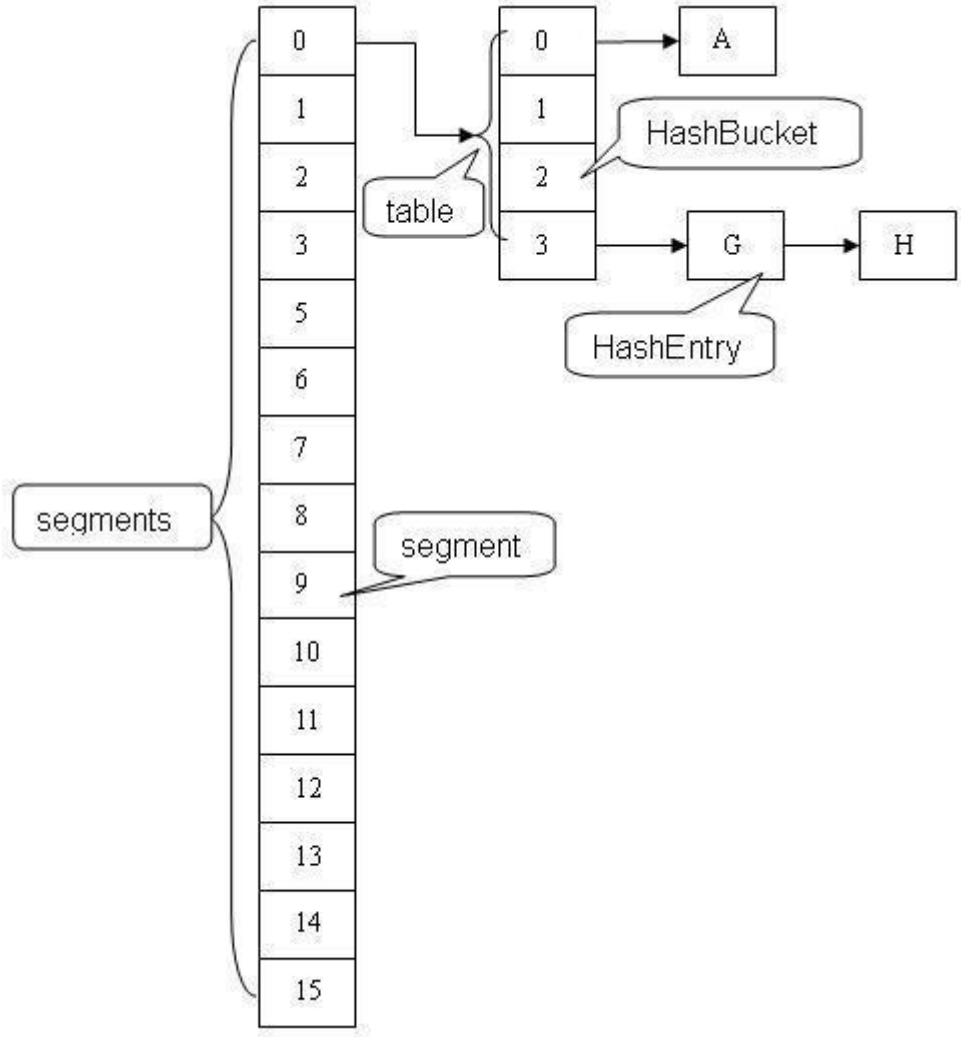
ConcurrentHashMap 使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。读操作大部分时候都不需要用到锁。只有在 size 等操作时才需要锁住整个 hash 表。

它把区间按照并发级别(concurrentLevel)，分成了若干个 segment。默认情况下内部按并发级别为 16 来创建。对于每个 segment 的容量，默认情况也是 16。当然并发级别(concurrentLevel)和每个段(segment)的初始容量都是可以通过构造函数设定的。ConcurrentHashMap 使用 segment 来分段和管理锁，

segment 继 承 自 ReentrantLock， 因 此 **ConcurrentHashMap 使 用 ReentrantLock 来保证线程安全**。

创建好默认的 ConcurrentHashMap 之后，它的结构大致如下图：






```
static final class HashEntry<K,V> {
    final K key;
    final int hash;
    volatile V value;
    final HashEntry<K,V> next;
}
```

可以看到除了 value 不是 final 的，其它值都是 final 的，这意味着不能从 hash 链的中间或尾部添加或删除节点，因为这需要修改 next 引用值，所有的节点的修改只能从头部开始。**为了确保读操作能够看到最新的值，将 value 设置成 volatile，这避免了加锁。**

1. get(Object key)的实现

```
public V get(Object key) {
        int hash = hash(key.hashCode());
        return segmentFor(hash).get(key, hash);
    }
```

清单 9.get 操作

```
V get(Object key, int hash) {
          if(count != 0) {       // 首先读 count 变量
              HashEntry<K,V> e = getFirst(hash);
              while(e != null) {
                  if(e.hash == hash && key.equals(e.key)) {
                      V v = e.value;
                      if(v != null)
                          return v;
                      // 如果读到 value 域为 null，说明发生了重排序，加锁后重新读取
                      return readValueUnderLock(e);
                  }
                  e = e.next;
              }
          }
          return null;
      }
```

```
1.  V readValueUnderLock(HashEntry<K,V> e) {
2.      lock();
3.      try {
4.          return e.value;
5.      } finally {
6.          unlock();
7.      }
8.  }
```

get 操作的高效之处在于整个 get 过程不需要加锁，**除非读到的值是空的才会加锁重读**。

之所以不会读到过期的值，是根据 java 内存模型的 happen before 原则，**对 volatile 字段的写入操作先于读操作，即使两个线程同时修改和获取 volatile 变量，get 操作也能拿到最新的值**，这是用 **volatile 替换锁**的经典应用场景。

2. put(K key,V value)的实现--用 lock 加锁

首先，根据 key 计算出对应的 hash 值：

清单 4.Put 方法的实现

```
public V put(K key, V value) {
        if (value == null)          //ConcurrentHashMap 中不允许用 null 作为映射值
            throw new NullPointerException();
        int hash = hash(key.hashCode());        // 计算键对应的散列码
        // 根据散列码找到对应的 Segment
        return segmentFor(hash).put(key, hash, value, false);
}
```

然后，根据 hash 值找到对应的 Segment 对象：

清单 5.根据 hash 值找到对应的 Segment

```
/**
     * 使用 key 的散列码来得到 segments 数组中对应的 Segment
     */
final Segment<K,V> segmentFor(int hash) {
   // 将散列值右移 segmentShift 个位，并在高位填充 0
   // 然后把得到的值与 segmentMask 相“与”
// 从而得到 hash 值对应的 segments 数组的下标值
// 最后根据下标值返回散列码对应的 Segment 对象
        return segments[(hash >>> segmentShift) & segmentMask];
}
```

最后，在这个 Segment 中执行具体的 put 操作：

清单 6.在 Segment 中执行具体的 put 操作

```
V put(K key, int hash, V value, boolean onlyIfAbsent) {
          lock();  // 加锁，这里是锁定某个 Segment 对象而非整个 ConcurrentHashMap
          try {
              int c = count;

              if (c++ > threshold)     // 如果超过再散列的阈值
                  rehash();              // 执行再散列，table 数组的长度将扩充一倍

              HashEntry<K,V>[] tab = table;
              // 把散列码值与 table 数组的长度减 1 的值相“与”
              // 得到该散列码对应的 table 数组的下标值
              int index = hash & (tab.length - 1);
              // 找到散列码对应的具体的那个桶
              HashEntry<K,V> first = tab[index];

              HashEntry<K,V> e = first;
              while (e != null && (e.hash != hash || !key.equals(e.key)))
                  e = e.next;

              V oldValue;
              if (e != null) {            // 如果键 / 值对以经存在
                  oldValue = e.value;
                  if (!onlyIfAbsent)
                      e.value = value;    // 设置 value 值
              }
              else {                        // 键 / 值对不存在
                  oldValue = null;
                  ++modCount;          // 要添加新节点到链表中，所以 modCont 要加 1
                  // 创建新节点，并添加到链表的头部
                  tab[index] = new HashEntry<K,V>(key, hash, first, value);
                  count = c;               // 写 count 变量
              }
              return oldValue;
          } finally {
              unlock();                    // 解锁
          }
      }
```

插入操作需要经历两个步骤，**第一步判断**是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置然后放在 HashEntry 数组里。

是否需要扩容。在插入元素前会先判断 Segment 里的 HashEntry 数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，**Segment的扩容判断比 HashMap 更恰当，因为 HashMap 是在插入元素后判断元素是否已经到达容量的**，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时 HashMap 就进行了一次无效的扩容。

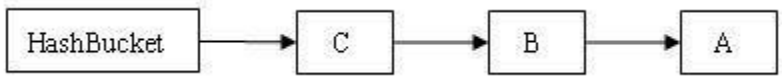
如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再 hash 后插入到新的数组里。

注意：这里的加锁操作是针对某个具体的 Segment，锁定的是该 Segment 而不是整个 ConcurrentHashMap。因为插入键 / 值对操作只是在这个 Segment 包含的某个桶中完成，不需要锁定整个 ConcurrentHashMap。此时，其他写线程对另外 15 个 Segment 的加锁并不会因为当前线程对这个 Segment 的加锁而阻塞。同时，所有读线程几乎不会因本线程的加锁而阻塞（**除非读线程刚好读到这个 Segment 中某个 HashEntry 的 value 域的值为 null，此时需要加锁后重新读取该值**）。

空值的唯一源头就是 HashEntry 中的默认值，因为 HashEntry 中的 value 不是 final 的，非同步读取有可能读取到空值。仔细看下 put 操作的语句：**tab[index] = new HashEntry<K,V>(key, hash, first, value)，在这条语句中，HashEntry 构造函数中对value 的赋值以及对tab[index]的赋值可能被重新排序，这就可能导致结点的值为空**。这种情况应当很罕见，一旦发生这种情况，ConcurrentHashMap 采取的方式是在持有锁的情况下再读一遍，这能够保证读到最新的值，并且一定不会为空值。（newEntry 对象是通过 new HashEntry(K k , V v, HashEntry next) 来创建的。如果另一个线程刚好 new 这个对象时，当前线程来 get 它。因为没有同步，就可能会出现当前线程得到的 newEntry 对象是一个没有完全构造好的对象引用。这里同样有可能一个线程 new 这个对象的时候还没有执行完构造函数就被另一个线程得到这个对象引用。）

由于 HashEntry 的 next 域为 final 型，所以新节点只能在链表的表头处插入。下图是在一个空桶中依次插入 A，B，C 三个 HashEntry 对象后的结构图：

图 1. 插入三个节点后桶的结构示意图：



注意：由于只能在表头插入，所以链表中节点的顺序和插入的顺序相反。

3. remove(Object key)的实现

下面来分析 remove 操作，先让我们来看看 remove 操作的源代码实现。

```
1.  public V remove(Object key) {
2.   hash = hash(key.hashCode());
3.      return segmentFor(hash).remove(key, hash, null);
4.  }
```

```
V remove(Object key, int hash, Object value) {
          lock();        // 加锁
          try{
              int c = count - 1;
              HashEntry<K,V>[] tab = table;
              // 根据散列码找到 table 的下标值
              int index = hash & (tab.length - 1);
              // 找到散列码对应的那个桶
              HashEntry<K,V> first = tab[index];
              HashEntry<K,V> e = first;
              while(e != null&& (e.hash != hash || !key.equals(e.key)))
                  e = e.next;

              V oldValue = null;
              if(e != null) {
                  V v = e.value;
                  if(value == null|| value.equals(v)) { // 找到要删除的节点
                      oldValue = v;
                      ++modCount;
                      // 所有处于待删除节点之后的节点原样保留在链表中
                      // 所有处于待删除节点之前的节点被克隆到新链表中
                      HashEntry<K,V> newFirst = e.next;// 待删节点的后继结点
                      for(HashEntry<K,V> p = first; p != e; p = p.next)
                          newFirst = new HashEntry<K,V>(p.key, p.hash,
                                                        newFirst, p.value);
                      // 把桶链接到新的头结点
                      // 新的头结点是原链表中，删除节点之前的那个节点
                      tab[index] = newFirst;
                      count = c;      // 写 count 变量
                  }
              }
              return oldValue;
          } finally{
              unlock();              // 解锁
          }
      }
```

首先根据散列码找到具体的链表，然后遍历这个链表找到要删除的节点；最后把待删除节点之后的所有节点原样保留在新链表中，把待删除节点之前的每个节点克隆到新链表中。下面通过图例来说明 remove 操作。假设写线程执行 remove 操作，要删除链表的 C 节点，另一个读线程同时正在遍历这个链表。

图 4. 执行删除之前的原链表：

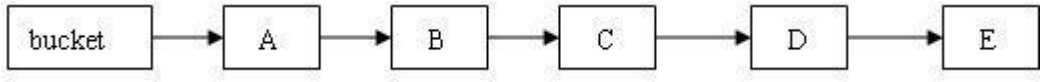
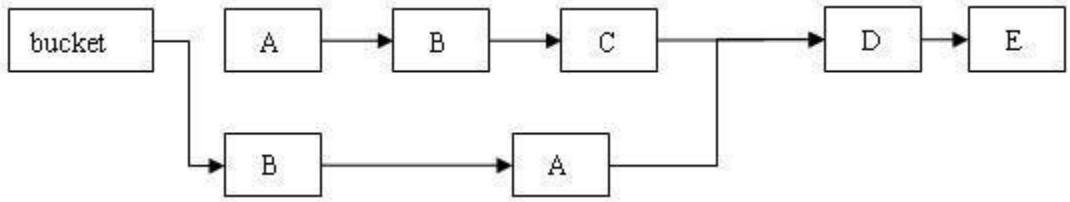


图 5. 执行删除之后的新链表



从上图可以看出，节点 C 之后的所有节点原样保留到新链表中；删除节点 C 之前的每个节点被克隆到新链表中，注意：它们在新链表中的链接顺序被反转了。

在执行 remove 操作时，原始链表并没有被修改，也就是说：读线程不会受同时执行 remove 操作的并发写线程的干扰。

综合上面的分析我们可以看出，写线程对某个链表的结构性修改不会影响其他的并发读线程对这个链表的遍历访问。

用 HashEntry 对象的不变性来降低读操作对加锁的需求

HashEntry 类的 value 域被声明为 Volatile 型，Java 的内存模型可以保证：某个写线程对 value 域的写入马上可以被后续的某个读线程“看”到。在 ConcurrentHashMap 中， 不允许用 Null 作为键和值， 当读线程读到某个 HashEntry 的 value 域的值为 null 时，便知道产生了冲突——**发生了重排序现象**，需要加锁后重新读入这个 value 值。**这些特性互相配合，使得读线程即使在不加锁状态下，也能正确访问 ConcurrentHashMap。**

下面分别来分析线程写入的两种情形：对散列表做非结构性修改的操作和对散列表做结构性修改的操作。

非结构性修改操作只是更改某个 HashEntry 的 value 域的值。**由于对 Volatile 变量的写入操作将与随后对这个变量的读操作进行同步。当一个写线程修改了某个 HashEntry 的 value 域后，另一个读线程读这个值域，Java 内存模型能够保证读线程读取的一定是更新后的值。**所以，写线程对链表的非结构性修改能够被后续不加锁的读线程“看到”。

结构性修改，实质上是对某个桶指向的链表做结构性修改。如果能够确保：**在读线程遍历一个链表期间，写线程对这个链表所做的结构性修改不影响读线程继续正常遍历这个链表。那么读 / 写线程之间就可以安全并发访问这个 ConcurrentHashMap。**

从上面的代码清单“在 Segment 中执行具体的 put 操作”中，我们可以看出：put 操作如果需要插入一个新节点到链表中时 , 会在链表头部插入这个新节点。此时，链表中的原有节点的链接并没有被修改。也就是说：插入新健 / 值对到链表中的操作不会影响读线程正常遍历这个链表。

ConcurrentHashMap 的**高并发性**主要来自于三个方面：

1.用**分离锁**实现多个线程间的更深层次的共享访问。

2.用 **HashEntry 对象的不变性**来降低执行读操作的线程在遍历链表期间对加锁的需求。

3.通过对**同一个 Volatile 变量的写 / 读访问**，协调不同线程间读 / 写操作的内存可见性。

通过 HashEntry 对象的不变性及对同一个 Volatile 变量的读 / 写来协调内存可见性，使得读操作大多数时候不需要加锁就能成功获取到需要的值。

ConcurrentHashMap **存在的问题**？

弱一致性的。

三．WeakHashMap VS HashMap

WeakHashMap 中的 key 采用的是“弱引用”的方式，只要 WeakHashMap 中的 key 不再被外部引用，所对应的键值对就可以被垃圾回收器回收。

HashMap 中的 key 采用的是“强引用”的方式，当 key 不再被外部引用时，只有当这个 key 从 HashMap 中删除后，才可以被垃圾回收器回收。

HashMap 和 TreeMap 区别：

1.实现方式的区别：

HashMap：基于**哈希表**实现。TreeMap：基于**红黑树**实现。

2.TreeMap 能够把它保存的记录根据**键排序.**

3.HashMap：适用于在 Map 中**插入、删除**和**查找**元素。

Treemap：适用于**按自然顺序**或自定义顺序**遍历键**(key)。

HashMap 通常比 TreeMap 快一点。

Hashset 的实现原理。

对于 HashSet 而言，它是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成。HashSet 中的元

素都存放在 HashMap 的 key 上面，而 value 中的值都是统一的一个 **private static final Object PRESENT = new Object();**

```
    // 底层使用 HashMap 来保存 HashSet 中所有元素。
    private transient HashMap<E,Object> map;

   // 定义一个虚拟的 Object 对象作为 HashMap 的 value，将此对象定义为 static final。
    private static final Object PRESENT = new Object();
 /*
    * 默认的无参构造器，构造一个空的 HashSet。
    * 实际底层会初始化一个空的 HashMap，并使用默认初始容量为 16 和加载因子 0.75 */
      public HashSet() {
          map = new HashMap<E,Object>();
       }

     public boolean add(E e) {
          return map.put(e, PRESENT)==null;
       }
```

3.讲一下集合中的 fail-fast 机制。

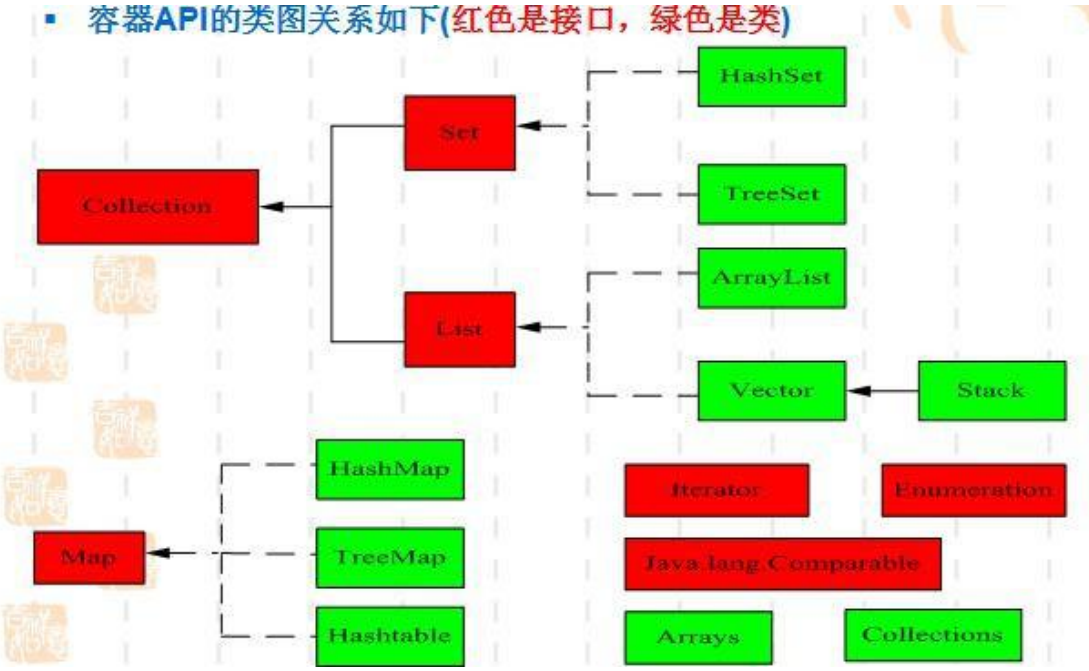
例如：假设存在两个线程（线程 1、线程 2），线程 1 通过 Iterator 在遍历集合 A 中的元素，在某个时候线程 2 修改了集合 A 的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。

产生的原因：

当调用容器的 iterator（）方法返回 Iterater 对象时，把容器中包含对象的个数赋值给了一个变量 expectedModCount,在**调用 next（）方法**时，会比较 expectedModCount 与容器中实际对象的个数是否相等，若二者不相等，则会抛出 ConcurrentModificationException 异常。

如果在遍历集合的同时，需要删除元素的话，可以用 iterator 里面的 remove()方法删除元素。

4.介绍一下 Java 中的集合框架？



5.Collection 包结构与 Collections 的区别。

集合框架：Collection：List 列表， Set 集

Map：Hashtable，HashMap，TreeMap

List 元素是有序的、可重复。

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

ArrayList：线程不安全，查询速度快。底层数据结构是数组

LinkedList：线程不安全。增删速度快。底层数据结构是链表

Set(集) 元素无序的、不可重复。

取出元素的方法只有迭代器。**不可以存放重复元素**，元素存取是无序的。**因此存入 Set 中的每个对象都必须重写 equals()和 hashCode()方法来确保对象的唯一性。**

Set 接口中常用的类

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euqals 方法。

TreeSet：线程不安全，可以对 Set 集合中的**元素进行排序**。TreeSet 底层数据结构是**二叉树**。

它的排序是如何进行的呢？通过compareTo 或者compare 方法来保证元素的有序性。元素是以二叉树的形式存放的。

Map 是一个双列集合

|--**Hashtable**:线程安全，速度慢。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。（当用自定义类型对象作为 HashTable 的 key 时，需要重新改写该对象的 equals()和 hashCode()方法来确保 key 的唯一性）。
|--**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，**线程安全类**。
|--**HashMap**:线程不安全，速度快。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。（当用自定义类型对象作为 HashMap 的 key 时，需要重新改写该对象的 equals()和 hashCode()方法来确保 key 的唯一性）。
|--**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**.

Collection 和 Collections 的区别：

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

Collections 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、线程安全化**等操作。

设计模式相关

1.编写线程安全的单例模式。

实现方式一：

利用**静态内部类**实现单例模式（极力推荐）

```
public class Singleton{ //“延迟加载”，“线程安全”
  private Singleton(){  }
  private static class T{ //静态内部类在使用的时候才加载，且只加载一次
   private static Singleton t=new Singleton();
  }
```

```
    public static Singleton getInstance(){
        return T.t;
    }
}
```

实现方式二：

双重校验锁 DCL(double checked locking)--使用 volatile 的场景之一

```
public class Singleton {
    //注意需要用 volatile 关键字
    private static volatile Singleton instance = null;
    private Singleton() {
    }
    //双重检查加锁，只有在第一次实例化时，才启用同步机制，提高了性能。
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();//注意这条语句，非原子操作
                }
            }
        }

        return instance;
    }
}
```

现在想象一下有线程 A 和 B 在调用 `getInstance()`，线程 A 先进入，在执行到 instance=new `Singleton()`:的时候被踢出了 cpu。然后线程 B 进入，B 看到的是 instance 已经不是 null 了（**内存已经分配**），于是它开始放心地使用 instance，但这个是错误的，因为 A 还没有来得及完成 instance 的初始化,而线程 B 就返回了未被初始化的 instance 实例（**为了禁止指令重排序，就用 volatile关键字**）。

看似简单的一段赋值语句：instance= new Singleton()，但是很不幸它并不是一个原子操作，其实际上可以抽象为下面几条 JVM 指令：

memory =allocate(); //1：分配对象的内存空间

ctorInstance(memory); //2：初始化对象

instance =memory; //3：设置 instance 指向刚分配的内存地址

上面操作 2 依赖于操作 1，但是操作 3 并不依赖于操作 2，所以 JVM 是可以针对它们进行指令的优化重排序的，经过重排序后如下：

memory =allocate(); //1：分配对象的内存空间

instance =memory; //3：instance 指向刚分配的内存地址，此时对象还未初始化

ctorInstance(memory); //2：初始化对象

可以看到指令重排之后，instance 指向分配好的内存放在了前面，而这段内存的初始化被排在了后面。

在线程A 执行这段赋值语句，在初始化分配对象之前就已经将其赋值给 instance 引用，恰好另一个线程进入方法判断 instance 引用不为 null，然后就将其返回使用，导致出错。

实现方式三：

```
public class SafeLazyInitialization {
   private static Resource resource;

   public synchronized static Resource getInstance() {
      if (resource == null)
         resource = new Resource();
      return resource;
   }
}
```

单例模式应用场景：

数据库连接池，多线程的线程池。 Windows 的任务管理器就是很典型的单例模式。

2.享元模式

String 采用了**享元设计模式（flyweight）**。

3.原型模式

原型模式**主要用于对象的复制**，实现一个接口（实现 Cloneable 接口），重写一个方法（重写 Object 类中的 clone 方法），即完成了原型模式。

原型模式中的拷贝分为"浅拷贝"和"深拷贝":

浅拷贝: 对值类型的成员变量进行**值的复制**,对引用类型的成员变量只复制引用,不复制引用的对象.

深拷贝: 对值类型的成员变量进行**值的复制,**对引用类型的成员变量也进行引用对象的复制.

（Object 类的clone 方法只会拷贝对象中的基本数据类型的值，对于数组、容器对象、引用对象等都不会拷贝，这就是浅拷贝。如果要实现深拷贝，必须将原型模式中的数组、容器对象、引用对象等另行拷贝。）

原型模式的优点。

1.如果创建新的对象比较复杂时，可以利用原型模式**简化对象的创建过程。**

2.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。

原型模式的使用场景。

因为以上优点，所以**在需要重复地创建相似对象时可以考虑使用原型模式。**比如需要在一个循环体内创建对象，假如对象创建过程比较复杂或者循环次数很多的话，使用原型模式不但可以简化创建过程，而且可以使系统的整体性能提高很多。

迭代器及迭代器模式

工厂、适配器、责任链、观察者

手写一个工厂模式。

外观模式。

命令模式

组合模式

常见设计模式，如 singlen，factory，abstract factory，strategy，chain，adaptor，decorator，composite，template，absever 等。
工厂方法模式的优点（低耦合、高内聚，开放封闭原则）

1.观察者设计模式

观察者模式定义了对象间的一对多依赖关系，让一个或多个**观察者对象**观察一个**主题对象**。当主题对象的状态发生变化时，系统能通知所有的依赖于此对象的观察者对象，从而使得观察者对象能够自动更新。

在观察者模式中，被观察的对象常常也被称为目标或**主题（Subject）**，依赖的对象被称为**观察者（Observer）**。

2.装饰者设计模式

现在需要一个汉堡，主体是鸡腿堡，可以选择添加生菜、酱、辣椒等等许多其他的配料，这种情况下就可以使用装饰者模式。

3.工厂设计模式

4.1 简单工厂模式

简单工厂模式的实质是由一个工厂类**根据传入的参数，动态决定**应该创建出哪一个产品类的实例。

4.2 抽象工厂模式

在这个模式中的几类角色,有抽象工厂，实体工厂，抽象产品，实体产品。

http://www.cnblogs.com/suizhouqiwei/archive/2012/06/26/2563332.html

4.3 工厂方法模式

JAVA 语言相关

1.标识符的命名规则。

标识符只能由数字、字母（a-z、A-Z）、下划线（_）和$组成，并且第一个字符不能为数字。

2.instanceof 关键字的作用。

用法：对象 A **instanceof** 类 B。

instanceof 通过返回一个布尔值来指出，这个对象是否是**这个特定类**或者是**它的子类**的一个实例。注意：如果对象 A 为 null，则返回 false。

3.strictfp 关键字的作用。

strictfp 可以用来修饰一个类、接口或者方法，在所声明的范围内，所有浮点数的计算都是精确的。当一个类被 strictfp 修饰时，所有方法默认也被 strictfp 修饰。

4.什么是不可变类？

不可变类：当创建了一个类的实例后，就不允许修改它的值了。特别注意：**String 和包装类（Integer,Float.....）**都是不可变类。String 采用了**享元设计模式（flyweight）**。

扩展问题 1：`new String("abc");`创建了几个对象？

1 个或者 2 个对象。如果常量池中原来有”abc”，那么只创建一个对象；如果常量池中原来没有字符串”abc”，那么就会创建 2 个对象。

扩展问题 2：

```
String s = "abc";
String ss = "ab" + "c";
System.out.println(s == ss);
```

输出为：true

解析： "ab" + "c"**在编译时**就被转换为”abc”。

扩展问题 3：

```
String s = "abc";
char[] ch={'a','b','c'};
System.out.println(s.equals(ch));
```

输出为：false

解析：S 和 ch 分别为字符串类型和数组类型，所以输出为 false。

```
public static void changeStringBuffer(StringBuffer ss1,
StringBuffer ss2) {
    ss1.append(" world");
    ss2 = ss1;
}
```

```
public static void main(String[] args)
    { Integer a = 1;//包装类是不可变类
    Integer b = a;
    b++;
    System.out.println(a);
    System.out.println(b);
    StringBuffer s1 = new StringBuffer("Hello");
    StringBuffer s2 = new StringBuffer("Hello");
    changeStringBuffer(s1, s2);
    System.out.println(s1);
    System.out.println(s2);
}
```

输出结果：

```
1
2
Hello world
Hello
```

5.Java 中的基本数据类型占据几个字节？

在 java 中，

占 1 个字节：byte，boolean

占 2 个字节：char,short

占 4 个字节：int,float

占 8 个字节：long,double

他们对应的封装类型是：**Integer** ,Double ,Long ,Float,

Short,Byte,**Character**,Boolean。

基本数据类型和对应的包装类的区别：

初始值的不同。当基本数据类型变量和包装类的对象作为**类的实例变量**时，默认初始值是不同的。包装类的对象默认初始值是 null，基本数据类型变量的默认初始值根据变量类型不同而不同，如 Int 的默认初始值是 0。如下。另外，**包装类是不可变类。**

```
public class Test {
    Integer i;
    int j;

    public static void main(String[] args)
        { System.out.println(new Test().i);
        System.out.println(new Test().j);
    }
}
```

结果输出为：

```
null
0
```

其他需要注意的问题：

1.默认声明的小数是double 类型的，比如 1.2 默认就是 double 类型的，因此，在对 float 类型的变量进行赋值的时候需要进行类型转换。如 `float i=1.2f;`

2.null **不是一个合法的 object 实例**,所以**并没有为其分配内存。**null 仅仅用于表明该引用目前没有指向任何对象。

3.Integer 的缓存策略。

在类加载时就将-128 到 127 的 Integer 对象创建了，并保存在 cache 数组中(**Integer cache[]**)，一旦程序调用 **Integer.valueOf(i)** 方法，如果 i 的值是在**-128 到 127** 之间就直接在 cache 缓存数组中**去取 Integer 对象，不**

在的话，就创建新的包装类对象。

Short(-128 — 127 缓 存)

Long(-128 — 127 缓存)

Float(没有缓存)

Doulbe(没有缓存)

```
1.     public static Integer valueOf(int i) {
2.         if(i >= -128 && i <= IntegerCache.high)
3.             return IntegerCache.cache[i + 128];
4.         else
5.             return new Integer(i);
6.     }
```

例子：

装箱是将一个原始数据类型赋值给相应封装类的变量，而拆箱则是将一个封装类的变量赋值给相应原始数据类型的变量。

```
//装箱。装箱就是 jdk 自己帮你完成了调用 Integer.valueOf(100)。
Integer integer1 = 3;
Integer integer2 = 3;
System.out.println(integer1 == integer2);

Integer integer3 = 300;
Integer integer4 = 300;
System.out.println(integer3 == integer4);

Integer a3 = new Integer(100);
Integer a4 = new Integer(100);
System.out.println(a3 == a4);

int a = 1000, b = 1000;
System.out.println(a == b);

Integer c = 1000, d = 1000;
System.out.println(c == d);

Integer e = 100, f = 100;
```

```
        System.out.println(e == f);
```

输出：

```
true
false
false
true
false
true
```

```
        int i = 128;
        Integer i2 = 128;
        Integer i3 = new Integer(128);
        // 当 Integer 和 int 类型数值进行比较的时候，Integer 会自动拆箱为 int 再比较，所以为 true
        System.out.println(i == i2);
        System.out.println(i == i3);
        System.out.println("***************");
        Integer i5 = 127;// java 在编译的时候,被翻译成-> Integer i5 = Integer.valueOf(127);
        Integer i6 = 127;
        System.out.println(i5 == i6);// true
        Integer ii5 = new Integer(127);
        System.out.println(i5 == ii5); // false
```

4.包装类作为参数传递时，仍是按值传递。

```
public static void fun(Integer
     i){ i=i+2;
}

public static void main(String[] args)
     { Integer p=new Integer(5);
      fun(p);
     System.out.println(p);
}
```

输出的结果仍然是 5。

6.运算符的优先级。

（++,--）> (*,/,%) > (+,-) > (<<,>>) > (&) > (|) >&& > ||

例子：

```
public static void main(String[] args) {
    byte a = 5;
    int b = 5;
    int c = a >> 2 + b >> 2;
    System.out.println(c);
}
```

输出是 0.

```
public static void main(String[] args)
    { int a = 10, b = 4, c = 5, d = 9;
    System.out.println(++a * b + c * --d);
}
```

输出是 84.

7.强制类型转换时的规则有哪些。

1.当对小于 int 的数据类型（byte,char,short）进行运算时，首先会把这些类型的**变量值**强制转为 int 类型，对 int 类型的值进行运算，最后得到的值也是 int 类型的。因此，如果把 2 个 short 类型的值相加，最后得到的结果是 int 类型，如果需要得到 short 类型的结果，就必须显示地运算结果转为 short 类型。

例子: `short s1=1; s1=s1+1;`
编译出错。正确的写法是 `short s1 = 1; s1 = (short) (s1 + 1);`

例子: `short s1 = 1;s1 += 1;`
编译通过。

2.基本数据类型和 boolean 类型是不能相互转换的。

3.char 类型的数据转为高级类型时，会转换为对应的 ASCII 码。

例题 1：

```
int i=1;
if(i)
  System.out.println(i);
```

编译出错。基本数据类型和 boolean 类型是不能相互转换的。

例题 2：

```
short i = 128;
System.out.println((byte) i);
```

i 对应的二进制为 00000000 10000000,由于 byte 只占一个字节，**在强制转换的时候，从前面开始截掉**，因此截掉后的值为二进制的 10000000，也就是十进制的-128。

扩展：**数据类型自动转换**

自动转换按从低到高的顺序转换。不同类型数据间的优先关系如下：

低 ----------> 高

byte,short,char-> **int -> long -> float -> double**

8.数组初始化时需要注意的问题。

1.数组被创建后会根据数组存放的数据类型**默认初始化为对应的初始值**，例如，int 类型会初始化为 0，对象类型会初始化为 null。

2.2 维数组中，每行元素个数可以不同。

9.如何在 main()方法执行前输出“Hello world”？

用静态代码块。静态代码块在类加载的初始化阶段就会被调用。

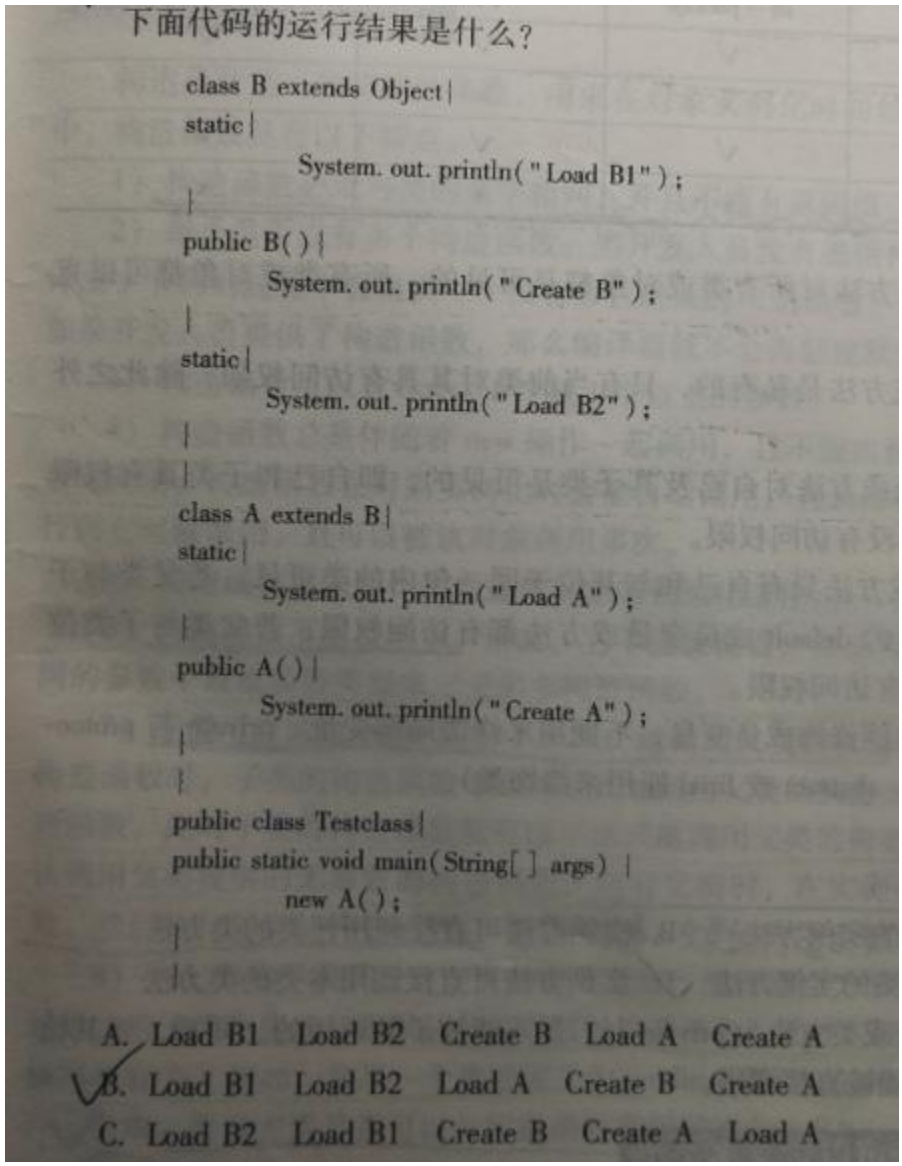
10.Java 程序初始化的顺序（对象实例化的过程）。

1.父类的静态变量、父类的静态代码块 （谁在前，谁先初始化）

2.子类的静态变量、子类的静态代码块（谁在前，谁先初始化）

3.父类的非静态变量、父类的非静态代码块（谁在前，谁先初始化）、父类的构造函数

4.子类的非静态变量、子类的非静态代码块（谁在前，谁先初始化）、子类的构造函数


下面代码的运行结果是什么？

```
class B extends Object{
static{
        System.out.println("Load B1");
}
public B(){
        System.out.println("Create B");
}
static{
        System.out.println("Load B2");
}
}
class A extends B{
static{
        System.out.println("Load A");
}
public A(){
        System.out.println("Create A");
}
}
public class Testclass{
public static void main(String[] args){
        new A();
}
}
```

A. Load B1 Load B2 Create B Load A Create A

B. Load B1 Load B2 Load A Create B Create A

C. Load B2 Load B1 Create B Create A Load A

11.构造函数的特点。

1.构造函数必须和类名一样（但和类名一样的不一定是构造方法，普通方法也可以和类名同名），并且不能有返回值，返回值也不能为 void。

2.构造函数总是伴随着new 操作一起调用，并且不能由程序的编写者调用，只能由系统调用。

3.构造函数不能被继承。

4.子类可以通过 super()来显示调用父类的构造函数。

```
public class Test {
   public Test()
      { System.out.println("constructor...")
      ;
   }
   public void Test()
```

```
{ System.out.println("call test14….");
```

```
    }
    public static void main(String[] args) {
        new Test().Test();
    }
}
```

输出结果为：

```
constructor...
call test14....
```

解析：**和类同名的方法，除了构造方法以外，也可以是普通方法。构造方法不带返回值，而普通方法是必须有返回值的，这就是区别他们的方法。**

12.序列化与反序列化。

1.定义

把 **Java 对象**转换为**字节序列**的过程称为对象的序列化。把**字节序列**恢复为 **Java 对象**的过程称为对象的反序列化。

2.实现方式

所有实现序列化的类都必须实现 Serializable 接口，它是一种标记接口，里面没有任何方法。当序列化的时候，需要用到 **ObjectOutputStream** 里面的 **writeObject()**;当反序列化的时候，需要用到 **ObjectInputStream** 里面的 **readObject()**方法。

3.特点

3.1 序列化时，**只对对象的状态进行保存**，而不管对象的方法。

3.2 当一个**父类实现序列化**时，子类自动实现序列化，不需要显示实现 Serializable 接口。

3.3 当一个对象的实例变量引用了其他对象时，序列化该对象时，也把引用对象进行序列 化。

3.4 对象中**被 static 或者 transient 修饰的变量**，在序列化时其变量值是不被保存的。

4.好处

一是，实现了**数据的持久化**，通过序列化可以把数据永久地保存到硬盘上（通常存放在 文件里）；二是，利用序列化实现**远程通信**，即在网络上传送对象的字节序列。

下面说下版本号。

我们在写程序的时候，最好在被序列化的类中显示声明 serialVersionUID.这么做的好处：

1.提高程序的运行效率。如果在类中没有显示声明 serialVersionUID，那么在序列化的时候会通过计算得到该值。如果显示声明该值的话，会省去计算的过程。

2.如果你的类没有提供 serialVersionUID，那么编译器会自动生成，而这个 serialVersionUID 就是对象的 hashcode 值。那么如果加入新的成员变量，重新生成的 serialVersionUID 将和之前的不同，那么在进行反序列化的时候就会产生 java.io.InvalidClassException 的异常。

扩展：ArrayList 和 LinkedList 能否序列化？

都可以序列化。ArrayList 里面的数组elementData 是声明为transient 的，表示ArrayList 在序列化的时候，默认不会序列化这些数组元素。因为 ArrayList 实际上是动态数组，每次

在放满以后会扩容，如果数组扩容后，实际上只放了一个元素，那就会序列化很多 null 元素，浪费空间，所以 ArrayList **把元素数组设置为 transient，仅仅序列化已经保存的数据**。

对象实现 java.io.Serializable 接口以后，序列化的动作不仅取决于对象本身，还取决于执行序列化的对象。

以 ObjectOutputStream 为例，如果 ArrayList 或自定义对象实现了 writeObject()，readObject()，那么在序列化和反序列化的时候，就按照自己定义的方法来执行动作，所以 **ArrayList 就自定义了 writeObject 和 readObject 方法**，然后在 writeObject 方法内完成数组元素的自定义序列化动作，在 readObject 方法内完成数组元素的自定义反序列化动作。

13. Switch 能否用 string 做参数？

在 Java 7 之前，switch 只能支持 **byte、short、char、int** 或者其对应的**包装类**以及 **Enum 类型**。在 **Java 7 中，String** 支持被加上了。

在使用 switch 时，需要注意另外一个问题，如果和匹配的 case 情况中省略了 break，那么匹配的 case 值后的所有情况都会执行，而不管 case 是否匹配，一直遇到 break 结束。

```
int x = 2;
switch (x)
{ case 2:
    System.out.println(x);
case 3:
    System.out.println(x);
case 4:
    System.out.println(x);
    break;
default:
    System.out.println("dddddd");
}
```

输出结果为：

```
2
2
2
```

14. equals 与==的区别。

“==”可用于**比较基本数据类型**（比较的是他们的值是否相等），也可以用于比较**对象在内存中的存放地址是否相等**。

JAVA 当中所有的类都是继承于 Object 这个基类的，在 Object 中的基类中定义了一个 equals 的方法，**这个方法的初始行为是比较对象的内存地址，但在一些类库当中这个方法被覆盖掉了，如 String，包装类**在这些类当中 equals 有其自身的实现 而不再是比较类在堆内存中的存放地址了（**对于 String 的 equals（）方法比较二个对象的内容是否相等**）。

因此，对于复合数据类型之间进行 equals 比较，**在没有覆写 equals 方法的情况下，他们之间的比较还是基于他们在内存中的存放位置的地址值的**，因为 Object 的 equals 方法也是用双等号（==）进行比较的，所以比较后的结果跟双等号（==）的结果相同。

```
public class Example4
{
public static void main(String[] args)
{
   Example4 e=new Example4();
   Example4  e4=new
   Example4();
   System.out.println("用 equals(Object) 比较结果");
   System.out.println(e.equals(e4));   //结果为 false
   System.out.println(" 用 == 比 较 结 果 ");
   System.out.println(e==e4);          //结果为 false
 }
}
```

扩展： 当调用 **intern 方法**时， 如果池已经包含一个等于此 String 对象的字符串（该对象由 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并且返回此 String 对象的引用。

15.Hashcode 的作用。

hashCode()方法是从Object 类继承过来的 Object 类中的hashCode()方法返回的是**对象在内存中的地址转换成的 int 值，如果对象没有重写 hashCode()方法，任何对象的 hashCode()方法的返回值都是不相等的。**

重写方法：Java 中的 hashCode 方法就是根据一定的规则将与对象相关的信息（**比如对象的存储地址，对象的字段等**）映射成一个数值，这个数值称作为散列值。

主要作用是用于查找的，为了配合**基于散列的集合**一起正常运行，这样的散列集合包括 **HashSet**、**HashMap** 以及 **HashTable**，hashCode 是用来在散列存储结构中确定对象的存储地址的。

考虑一种情况，当向集合中插入对象时，如何判别在集合中是否已经存在该对象了？（注意：集合中不允许重复的元素存在）。

当集合要添加新的对象时，先调用这个对象的 hashCode 方法，得到对应的 hashcode 值；如果在该位置有值，就调用它的 equals 方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址。

扩展：为什么在重写 equals 方法的同时，必须重写 hashCode 方法？

回答：比如在使用 set 集合时，往其中放入内容相同的对象，如果没有重写 hashCode()方法，那么 set 中将会放入内容相同的对象（因为 2 个对象的地址不同），这和 set 集合的性质不同。因此需要在重写 equals 方法的同时，必须重写 hashCode 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 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);
        }
    }
```

此时得到的结果就会如下：

```
true
true
[com.ubs.sae.test.HashTest@1]
```

Java 对于 equals()方法和 hashCode()方法是这样规定的：

1.如果 2 个对象的 equals()方法返回 true，则 hashCode()返回的值也相同。

2.如果 2 个对象的 equals()方法返回 false，则 hashCode()返回的值可能相同，也可能不相同。

3.如果 2 个对象的 hashCode()方法返回值相同，则 equals()返回的值可能为 true，也可能为 false。

4.如果 2 个对象的 hashCode()方法返回值不同，则 equals()返回的值为 false。

16.hashCode() 与 equals() 生成算法、方法怎么重写。

1. 尽量保证使用对象的**同一个属性**来生成 hashCode()和 equals()两个方法。

2. 在重写 equals 方法的同时，必须重写 hashCode 方法（二者必须同时重写）。

```
//age 是 int 类型的，id 是 Integer 类型的
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Employee other = (Employee) obj;
    if (age != other.age)
        return false;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    return true;
}

public int hashCode()
    { final int prime =
    31; int result = 1;
    result = prime * result + age;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}
```

17. Object 有哪些公用方法？

1. clone() ：创建并返回此对象的一个副本。

2. equals(Object obj)：指示其他某个对象是否与此对象的地址“相等”。
   hashCode()：返回该对象的哈希码值。

3. wait() ：
   notify() ：唤醒在此对象监视器上等待的单个线程。
   notifyAll()：唤醒在此对象监视器上等待的所有线程。

4. toString()：返回该对象的字符串表示。

5. finalize()：当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
6. getClass() ：返回此 Object 的运行时类。

18. String、StringBuffer 与 StringBuilder 的区别。

1.可变与不可变

string 对象是不可变的；StringBuilder 与 StringBuffer 对象是可变的。

因此**在每次对 String 类型进行改变的时候**，都会生成一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会**对系统性能产生影响**，特别当内存中**无引用对象**多了以后， JVM 的 GC 就会开始工作，性能就会降低。

修改 String 对象的原理：首先创建一个 StringBuffer 对象，然后调用 append()方法，最后调用 toString()方法。

```
String s=”Hello”;
S+=”World”;
```

等价于：

```
StringBuffer  sb=new StringBuffer(s);
sb.append(“World”);
sb.toString();
```

使用 StringBuffer 类时，每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。所以多数情况下推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。

2.是否多线程安全

String 和 StringBuffer 是线程安全的。 StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

3.初始化方式的不同

StringBuffer 和 StringBuilder 只能用构造函数的形式来初始化。String 除了用构造函数进行初始化外，还可以直接赋值。

19. try catch finally，try 里有 return，finally 还执行么？如果会的话，什么时候执行，在 return 之前还是 return 之后？

1.finally 块里的代码在 return 之前执行。

2.如果 finally 块中有 return 语句的话，它将覆盖掉函数中其他 return 语句

1 如果 finally 块中，改变了 try 块中返回变量的值，该变量为基本数据类型的话，则 finally 块中改变变量值的语句将不起作用；如果该变量为引用变量的话，则起作用。

2 Finally 块中的代码不一定执行。

比如，try 块执行之前，出现了异常，则程序终止。

比如，在 try 块中执行了 System.*exit*(0)。

20. 说下异常的原理。

异常是指**程序运行时**所发生的错误。

Throwable 是所有异常的父类，它有 2 个子类：Error 和 Exception。

1 Error 表示程序在运行期间发生了**非常严重的错误**，并且该错误是**不可恢复**的。Error 不需要捕捉。如 OutOfMemoryError。

2 Exception 是**可恢复的异常**。它包含 2 种类型：检查异常和运行时异常。

2.1 检查异常（Checked Exception）

比如 IOException、SQLException 和 FileNotFoundException 都是检查异常。它发生在**编译阶段**，编译器会强制程序去捕获此类异常，需要在编码时用 try-catch 捕捉。

2.2 运行时异常（RuntimeException）

它发生在**运行阶段，编译器不会检查运行时异常**。比如空指针异常，算数运算异常，数组越界异常等。如果代码会产生 RuntimeException 异常，则需要通过修改代码进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

在处理异常的时候，需要注意：捕获异常的时候，**先捕获子类异常，再捕获父类异常。**

21.Java 面向对象的三个特征与含义。

封装：属性的封装和方法的封装。把属性定义为私有的，get(),set()方法。

好处是信息隐藏和模块化，提高安全性。封装的主要作用在于对外隐藏内部实现细节，增强程序的安全性。

继承：子类可以继承父类的成员变量和成员方法。继承可以提高代码的**复用性。**

继承的特性：

1.单一继承。

2.子类只能继承父类的**非私有成员变量和方法**。

3.成员变量的隐藏和方法的覆盖。

多态：当同一个操作作用在不同对象时，会产生不同的结果。实现原理见下面。

22. java 多态的实现原理（实现机制）。

有 2 种方式来实现多态，一种是**编译时多态**，另外一种是**运行时多态**；编译时多态是通过**方法的重载**来实现的，运行时多态是通过**方法的重写**来实现的。

方法的重载，指的是同一个类中有多个同名的方法，但这些方法有着不同的参数。**在编译时就可以确定到底调用哪个方法。**

方法的重写，子类重写父类中的方法。父类的引用变量不仅可以指向父类的实例对象，还可以指向子类的实例对象。**当父类的引用指向子类的对象时，只有在运行时才能确定调用哪个方法。**

特别注意：只有类中的方法才有多态的概念，类中成员变量没有多态的概念。

其余部分见“重载和覆盖的区别”。

23. Override（覆盖、重写）和 Overload（重载）的区别。

重载和覆盖是 java **多态性的不同表现方式**。

重载是在一个类中多态性的一种表现，是指在一个类中定义了多个同名的方法，但是他们有不同的参数个数或有不同的参数类型。

在使用重载时要注意以下几点：

1. 重载只能通过不同的方法参数来区分。例如**不同的参数类型，不同的参数个数，不同的参数顺序。**

2. **不能**通过**访问权限、返回类型、抛出的异常**进行重载。

覆盖是指子类函数覆盖父类中的函数。

在覆盖时要注意以下几点（重点！！）：

1.覆盖的方法的函数名和参数必须要和被覆盖的方法的**函数名和参数**完全匹配，才能达到覆盖的效果；

2.覆盖的方法的返回值必须和被覆盖的方法的**返回值类型一致**；

3.覆盖的方法所抛出的异常必须和被覆盖方法的**所抛出的异常一致，或者是其子类；**

4.**被覆盖的方法不能为 private**，否则在其子类中只是新定义了一个方法，并没有对其进行覆盖。

5. 子类函数的**访问修饰权限要大于等于**父类的**（public>protected>default>private）**。**（重要！！！）**

特别注意：Java 中，子类无法覆盖父类的 static 方法或 private 方法。

```
class Parent {
    public static void p()
        { System.out.println("parent....");
    }
}

public class Test19 extends Parent {
    public static void p()
        { System.out.println("child...")
        ;
    }

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

```
{ Parent parent = new Test19();
```

```
        parent.p();
    }
}
```

输出：

```
parent....
```

原因：Java 中，子类无法覆盖父类的 static 方法。

24.接口与抽象类的区别。

1.语法层面上的区别

1) 抽象类可以提供成员方法的实现细节（**注：可以只包含非抽象方法**），而接口中只能存在 public abstract 方法，方法默认是 public abstract 的，但是，java8 中接口可以有 default 方法；

2) 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 public static final 类型的；

3) 抽象类可以**有静态代码块和静态方法和构造方法;**接口中不能含有静态代码块以及静态方法以及构造方法。但是，java8 中接口可以有静态方法；

4) 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

2.设计层面上的区别

1) 抽象层次不同。抽象类是对类的整体抽象，包括属性和行为的抽象。而接口只是对行为的抽象。

2) 跨域不同。抽象类所体现的是一种继承关系，父类和派生类之间必须存在"is-a"关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的， 仅仅是实现了接口定义的契约而已，其设计理念是“has-a”的关系（有没有、具备不具备的关系），实现它的子类可以不存在任何关系，共同之处。例如猫、狗可以抽象成一个动物类抽象类，具备叫的方法。鸟、飞机可以实现飞Fly **接口**，具备飞的行为，这里我们总不能将鸟、飞机共用一个父类吧！

3) 设计层次不同。对于抽象类而言，它是自下而上来设计的，**我们要先知道子类才能抽象出父类**，而接口则不同，它根本就不需要知道子类的存在， 只需要定义一个规则即可，至于什么子类、什么时候怎么实现它一概不知。比如我们只有一个猫类在这里，如果你这时就抽象成一个动物类，是不是设

计有点儿过度？我们起码要有两个动物类，猫、狗在这里，我们再抽象他们的共同点形成动物抽象类吧！所以说抽象类往往都是通过重构而来的！但是接口就不同，比如说飞，我们根本就不知道会有什么东西来实现这个飞接口，怎么实现也不得而知，**我们要做的就是事前定义好飞的行为接口**。所以说抽象类是自底向上**抽象而来的**，接口是自顶向下**设计出来的**。

25. 静态内部类和非静态内部类的区别。

```
/* 下面程序演示如何在 java 中创建静态内部类和非静态内部类 */
class OuterClass {
    private static String msg = "GeeksForGeeks";

    // 静态内部类
    public static class NestedStaticClass {

        // 静态内部类只能访问外部类的静态成员和静态方法
        public void printMessage() {
            // 试着将 msg 改成非静态的，这将导致编译错误
            System.out.println("Message from nested static class:
" + msg);
        }
    }

    // 非静态内部类
    public class InnerClass {
        // 不管是静态方法还是非静态方法都可以在非静态内部类中访问
        public void display() {
            System.out.println("Message from non-static nested
class: " + msg);
        }
    }
}

 public class Main {
    // 怎么创建静态内部类和非静态内部类的实例
    public static void main(String args[]) {

        // 创建静态内部类的实例（注意前面还是要加外部类的名字的！！！）
        OuterClass.NestedStaticClass printer = new
OuterClass.NestedStaticClass();
```

```
        // 调用静态内部类的非静态方法
        printer.printMessage();

        // 为了创建非静态内部类，我们需要外部类的实例
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();

        // 调用非静态内部类的非静态方法
        inner.display();

        // 我们也可以结合以上步骤，一步创建的内部类实例
        OuterClass.InnerClass innerObject = new OuterClass().new
InnerClass();

        // 同样我们现在可以调用内部类方法
        innerObject.display();
    }
}
```

静态内部类和非静态内部类主要的不同：

（1）静态内部类**不依赖于外部类实例而被实例化**，而非静态内部类**需要在外部类实例化后才可以被实例化。**

（2）静态内部类**不需要持有外部类的引用**。但非静态内部类**需要持有对外部类的引用**。

（3）静态内部类不能访问外部类的非静态成员和非静态方法。它**只能访问外部类的静态成员和静态方法**。非静态内部类**能够访问外部类的静态和非静态成员和方法。**

扩展：内部类都有哪些？

有四种：静态内部类，非静态内部类，局部内部类，匿名内部类。

1.静态内部类和 **2.非静态内部类**的讲解见上面的部分。

3.局部内部类：在外部类的方法中定义的类。其作用的范围是所在的方法内。**它不能被 public,private,protected 来修饰。它只能访问方法中定义为 final 类型的局部变量。**

```
class outerClass{
  public void f(){
```

```
        class innerClass{//局部内部类

        }
    }
}
```

4.匿名内部类

```
interface Person {
    public abstract void eat();
}

public class Test4 {
    public static void main(String[] args)
        { Person p = new Person() {
            public void eat()
                { System.out.println("eat
                something");
            }
        };
        p.eat();
    }
}
```

是一种没有类名的内部类。

需要注意的是：

4.1 匿名内部类一定是在 new 的后面，这个匿名内部类必须继承一个父类或者实现一个接口。

4.2 匿名内部类不能有构造函数。

4.3 只能创建匿名内部类的一个实例。

4.4 在 Java 8 之前，如果匿名内部类需要访问外部类的局部变量，则必须使用 final 来修饰外部类的局部变量。**在现在的 Java 8 已经去取消了这个限制。**

26. static 的使用方式。

static 有 4 种使用方式：修饰类（静态内部类），修饰成员变量（静态变量），修饰成员方法（静态成员方法），静态代码块。

1.修饰类（静态内部类）

参考上面的介绍。

2.修饰成员变量（静态变量）

静态变量属于类，只要静态变量所在的类被加载，这个静态变量就会被分配空间，在内存中只有一份，所有对象共享这个静态变量。使用有二种方式，一个是类名.静态变量，还有一种是对象.静态变量。**特别注意：不能在方法体中定义静态变量（无论该方法是静态的或是非静态的）。VS** 实例变量属于类，只有对象创建后，实例变量才会分配空间。

3.修饰成员方法（静态成员方法）

静态成员方法属于类，不需要创建对象就可以使用。而非静态方法属于对象，只有在对象创建出来以后才可以被使用。静态方法里面只能访问所属类的静态成员变量和静态成员方法。

4.静态代码块

静态代码块经常被用来初始化静态变量，在类加载的初始化阶段会执行为静态变量赋值的语句和静态代码块的内容，静态代码块只会被执行一次。

27.反射的作用与原理。如何提高反射效率？

1.**定义**：反射机制是**在运行时**，对于**任意一个类**，都能够知道这个类的所有属性和方法；对于**任意一个对象**，都能够**调用**它的任意一个方法。**在 java 中，只要给定类的名字，那么就可以通过反射机制来获得类的所有信息。**

2.反射机制主要提供了以下**功能**：在运行时判定任意一个对象所属的类；在运行时创建对象；在运行时判定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。

3.哪里用到反射机制？

jdbc 中有一行代码：

Class.forName('com.mysql.jdbc.Driver.class');//加载 MySql 的驱动类。这就是反射，现在很多框架都用到反射机制，hibernate，struts 都是用反射机制实现的。

4.反射的实现方式

在 Java 中实现反射最重要的一步，也是第一步就是获取 Class 对象，得到 Class 对象后可以通过该对象调用相应的方法来获取该类中的属性、方法以及调用该类中的方法。

有 4 种方法可以得到 Class 对象：

1.Class.forName(“类的路径”);

2.类名.class。

3. 对象名.getClass()。

4. 如果是基本类型的包装类，则可以通过调用包装类的 Type 属性来获得该包装类的 Class 对象。

例如：Class<?> clazz = Integer.TYPE;

4. 实现 Java 反射的类

1) Class：它表示正在运行的 Java 应用程序中的类和接口。

2) Field：提供有关类或接口的属性信息，以及对它的动态访问权限。

3) Constructor：提供关于类的单个构造方法的信息以及对它的访问权限

4) Method：提供关于类或接口中某个方法信息。

注意：Class 类是Java反射中最重要的一个功能类，所有获取对象的信息(包括：方法/属性/构造方法/访问权限)都需要它来实现。

5. 反射机制的优缺点？

优点：

（1） 能够**运行时动态获取类的实例，大大提高程序的灵活性**。

（2） 与 Java **动态编译**相结合，可以实现无比强大的功能。

缺点：

（1）使用反射的**性能较低**。 java 反射是要解析字节码，将内存中的对象进行解析。

解决方案：

1. 由于 JDK 的安全检查耗时较多，所以通过 **setAccessible(true)的**方式关闭安全检查来（**取消对访问控制修饰符的检查**）提升反射速度。

2. 需要多次动态创建一个类的实例的时候，有缓存的写法会比没有**缓存**要快很多:

3. **ReflectASM 工具类** ，通过**字节码生成的方式**加快反射速度。

（2）使用反射相对来说**不安全，破坏了类的封装性**，可以通过反射获取这个类的私有方法和属性。

28.. Java 和 C++/C 的区别,JAVA 的优点。

JAVA 和 C/C++的区别：

1 运行过程的不同。JAVA 源程序经过编译器编译成字节码文件，然后由 JVM 解释执行。而 C++/C 经过编译、链接后生成可执行的二进制代码。因此 C/C++的执行速度比 JAVA 快。

2 跨平台性。JAVA 可以跨平台，而 C++/C 不可以跨平台。

3 JAVA 没有指针，C++/C 有指针。

4 JAVA 不支持多重继承，但是可以同时实现多个接口来达到类似的目的。C++支持多重继承。

5 JAVA 不需要对内存进行管理，有垃圾回收机制。C/C++需要对内存进行显示的管理。

6 JAVA 不支持运算符重载。C/C++支持运算符重载。

7 JAVA 中每个数据类型在不同的平台上所占字节数固定的，而 C/C++则不然。

JAVA 的优点：

1 跨平台。JAVA 语言可以“一次编译，到处运行”。跨平台的含义：无论是在 windows 平台还是在 Linux 平台对 Java 程序进行编译，编译后的程序都可以在其他平台上运行。编译器会把 JAVA 代码编译成字节码文件，然后在 JVM 上解释执行，由于字节码与平台无关，因此，JAVA 可以很好地跨平台执行。

2 垃圾回收机制。

3 去掉了 C++中难以理解的东西，比如指针和运算符重载。

4 具有较好的安全性。比如 JAVA 有数组边界检测，但是 C/C++里面没有。

29.同一个.java 文件中是否可以有多个 main()方法？

每个类中都可以定义 main()方法，但只有用 public 修饰的类且与文件名相同的类中的 main()方法才可以作为整个程序的入口方法。

例子：

创建了一个名为 Test17.java 的文件。

```
class T {

    public static void main(String[] args)
       { System.out.println("T main");
    }
}

public class Test17 {
```

```
    public static void main(String[] args)
        { System.out.println("Test main");
    }
}
```

输出：

```
Test main
```

30.JAVA 中的类和成员的访问控制。

类的访问控制。可以用 public 和不含 public 的来修饰。

成员的访问控制。

如果一个类是用 public 来修饰的，它的成员用第一列的访问修饰符时，在不同范围的类和对象是否有权访问它们。

表 3-1 访问修饰符的访问等级

访问修饰符	同一个类	同 包	不同包的子类	不同包的非子类
private	√			
protected	√	√	√	
public	√	√	√	√
默认	√	√		

31.System.out.println()方法使用时需要注意的问题。

```
public class Test18 {
    public static void main(String[] args)
        { System.out.println(1 + 2 + "");
        System.out.println("" + 1 + 2);
    }
}
```

输出：

```
3
12
```

32.继承和组合区别。

组合和继承是**代码复用**的 2 种方式。

1.组合是在新类里面创建原有类的对象，重复利用已有类的功能。

2.组合关系**在运行期决定**，而继承关系**在编译期**就已经决定了。

3.**使用继承关系**时，可以实现类型的回溯，即用父类变量引用子类对象，这样便可以**实现多态**，而组合没有这个特性。

4.从逻辑上看，组合最主要地体现的是一种整体和部分的思想，例如在电脑类是由内存类，CPU 类，硬盘类等等组成的，而继承则体现的是一种可以回溯的父子关系，子类也是父类的一个对象。

33.final finally finalize 的区别。

final 的用法：

final 可以用来修饰类，变量和方法。

1 当一个类被 final 修饰的时候，表示该类不能被继承。类中方法默认被 final 修饰。

2 当 final **修饰基本数据类型的变量**时，表示该值在被初始化后不能更改；

当 final **修饰引用类型的变量**时，表示该引用在初始化之后不能再指向其他的对象。

注意：**final 修饰的变量必须被初始化**。可以在定义的时候初始化，也可以在构造函数中进行初始化。

3 当 final 修饰方法时，表示这个方法不能被子类**重写**。

使用 final 方法的原因有 2 个：

第一、**把方法锁定**，防止任何继承类修改它的实现。

第二、**高效**。当要使用一个被声明为 final 的方法时，直接将方法主体插入到调用处，**而不进行方法调用**，可以提高程序的执行效率（ps.**如果过多的话，这样会造成代码膨胀**）。

扩展：String 类为什么要设计成 final 类型的？

java 中不仅 String 是 final 的。 Long, Double, Integer 之类的全都是 final 的。

1. Requirement of String Pool

If a string is not immutable, changing the string with one reference will lead to the wrong value for the other references.

2. Caching Hashcode

因为字符串是不可变的，所以在它创建的时候 hashcode 就被**缓存**了，不需要重新计算。

3.线程安全

Because immutable objects can not be changed, they can be shared among multiple threads freely. This eliminates the requirements of doing synchronization（不用同步了）。.

4.为了防止扩展类无意间**破坏原来方法的实现**。

finally 的用法：

finally 是异常处理的一部分，只能用在 try-catch 语句中，表示这段代码一般情况下，一定会执行。经常用在需要释放资源的情况下。

finalize 的用法：

它是 Object 类的一个方法，在垃圾回收器执行时会调用被回收对象的 finalize()方法。

34.JDK1.7 和 1.8 的区别。

接口的默认方法。

Java 8 允许我们给接口添加一个非抽象的方法实现，只需要使用 default 关键字即可，这个特征又叫做扩展方法，Default 方法带来的好处是，往接口新增一个 Default 方法，在接口添加新功能特性，而不破坏现有的实现架构。示例如下：

```
interface Formula {
    double calculate(int a);
    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Formula 接口在拥有 calculate 方法之外同时还定义了 sqrt 方法，实现了 Formula 接口的子类只需要实现一个 calculate 方法，默认方法 sqrt 将在子类上可以直接使用。

```
Formula formula = new Formula() {
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};
formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

35.List<String>能否转为 List<Object>?能否 List<Object> list= new ArrayList<String>()？List< String > list= new ArrayList< Object >()？原因？

都不可以，**会出现编译错误**。List<Object>可以存储任何类型的对象，包括String,Integer 等，而 ArrayList<String>只能用来存储字符串。

36.泛型的好处？

在编译的时候检查类型安全，确保只能把正确类型的对象放入集合中；消除强制类型转换。

Part---------JAVA 基础专题（补1）

1.String 中的”+”操作是怎么回事?

情况 1：

1. String a = "ab";
2. String bb = "b";
3. String b = "a" + bb;

System.out.println((a == b));

“**String+变量**”因为编译时无法进行优化，所以这一条语句的操作是**在运行时进行的**，且会产生新的对象，而不是直接从 jvm 的 string 池中获取。

java 中 String 的+运算符编译后其实是转换成了这样的代码：

3. String b = **new StringBuilder().append("a").append(bb).toString();**

其中，StringBuilder 的 toString 方法，使用 new String(...);来构造一个 String 对象。

```
public String toString() {  //StringBuilder 的 toString 方法
    // Create a copy, don't share the array
    return new String(value, 0, count);
}
```

情况 2：

一个特殊的例子：

String str = “This is only a” + “ simple” + “ test”; // “+”连接的都是字符串常量

StringBuffer builder = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);

你会很惊讶的发现，生成 str 对象的速度简直太快了，而这个时候 StringBuffer 居然速度上根本一点都不占优势。其实这是 JVM 的一个把戏，实际上：

String str = “This is only a” + “ simple” + “test”;

其实就是：

String str = “This is only a simple test”;

所以不需要太多的时间。但大家这里要注意的是，如果你的字符串是来自另外的 String 对象的话，速度就没那么快了，譬如：

String str2 = “This is only a”;

String str3 = “ simple”;

String str4 = “ test”;

String str1 = str2 +str3 + str4; //“+”连接的是字符串变量

这时候 JVM 会规规矩矩的按照原来的方式去做。

2.StringBuilder 和 StringBuffer 底层是怎么实现的。

每个 StringBuffer 对象都有一定的缓冲区容量（**可变长的字符数组**，类似于 ArrayList 的底层实现），默认初始容量为 16 个字符。当字符串大小没有超过容量时，不会分配新的容量；当字符串大小超过容量时，会自动扩容（扩容后的容量大约是之前的 2 倍）。StringBuilder 和 StringBuffer，字符串都是存放在 char[]中的。

3.String 类中常用的方法。

1.charAt(int index) ：返回指定索引处的 char 值。

2. intern() ：返回字符串对象的规范化表示形式。

3.isEmpty() ：当且仅当 length() 为 0 时返回 true。

4.length() ：返回此字符串的长度。

5.substring(int beginIndex) ：返回一个新的字符串，它是此字符串的一个子字符串。

substring(int beginIndex, int endIndex) ：返回一个新字符串，它是此字符串的一个子字符串。

6.toLowerCase() ：将此 String 中的所有字符都转换为小写。

toUpperCase()：将此 String 中的所有字符都转换为大写。

7.trim()：去除字符串首尾空格。

8.valueOf(Object obj) ： 返回 Object 参数的字符串表示形式。

9.equals(Object anObject) ：将此字符串与指定的对象比较。

4.创建虚引用的时候，构造方法传入一个 ReferenceQueue，作用是什么。

虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

5.栈溢出的原因和解决方法。

原因：

① 大量的递归调用，在不断的压栈过程中，造成栈容量超过而导致溢出。

② 由于分配了过大的局部变量。

解决方法：

（1） 用栈把递归转换成非递归。

（2） 使用静态对象替代非静态局部对象。

在递归函数设计中，可以使用静态对象替代非静态局部对象（即栈对象），这不仅可以减少每次递归调用和返回时产生和释放非静态对象的开销，而且静态对象还可以保存递归调用的中间状态，并且可为各个调用层所访问。

（3）增加堆栈的大小

6.HashMap 的加载因子的作用。

加载因子是表示 Hash 表中元素的填满程度。若加载因子越大，填满的元素越多，好处是，空间利用率高了，但冲突的机会加大了，增加查询数据的时间开销。反之，加载因子越小，填满的元素越少，好处是，冲突的机会减小了， 会提高数据查询的性能，但空间浪费多了。特别地，JDK1.8 中对 HashMap 进行了增强，如果一个桶上的节点数量过多，链表+数组的结构就会转换为红黑树。

7.HashMap 中的 key 可以是任意对象吗？（Set 中元素的内容可以改变吗？）

可变对象和不可变对象都可以。可变对象是指创建后自身状态能改变的对象。换句话说，可变对象是该对象在创建后它的哈希值（由类的 hashCode（）方法可以得出哈希值）可能被改变。

如果可变对象在 HashMap 中被用作 key，当对象中的属性改变时，则对象 HashCode 也可能进行相应的改变，这将导致下次无法查找到已存在 Map 中的数据；或者当想删除被改变对象的时候，由于找不到该对象，无法删除，会造成内存泄漏。 所以当 hashCode( )重写的时候，最好不能依赖该类的易变属性，这样就可以保证成员变量改变的时候，该对象的哈希值不变。

8.如果你定义一个类，包括学号，姓名，分数，如何把这个对象作为 key？

需要重写 equals( )方法和 hashCode（）方法，必须保证对象的属性改变时，其 hashCode()返回值不能改变。

9.java 是如何实现跨平台的。

java 跨平台的实质是虚拟机的跨平台。JVM 也是一个**软件**，不同的平台有不同的版本。我们编写的 Java 源码，编译后会生成一种 .class 文件，称为字节码文件。Java 虚拟机就是负责将字节码文件翻译成特定平台下的机器码然后运行。也就是说，只要在不同平台上安装对应的 JVM，就可以运行字节码文件，运行我们编写的 Java 程序。

10.什么是泛型，为什么要使用以及类型擦除。

泛型的本质就是“参数化类型”，也就是说所操作的数据类型被指定为一个参数。创建集合时就指定集合元素的数据类型，该集合只能保存其指定类型的元素，避免使用强制类型转换。

Java 编译器生成的字节码是不包含泛型信息的，泛型类型信息将在**编译处理时**被擦除，这个过程即**类型擦除**。类型擦除可以简单的理解为将泛型 java 代码转换为普通 java 代码，只不过编译器更直接点，将泛型 java 代码直接转换成普通 java 字节码。

类型擦除的主要过程如下：

一.将所有的泛型参数用其最左边界（最顶级的父类型）类型替换。

二.移除所有的类型参数。

11.Java 中的 NIO，BIO 分别是什么。NIO 主要用来解决什么问题。

NIO 的主要作用就是用来解决速度差异的。

Java 中 NIO 和 IO 之间的区别。

1.面向流与面向缓冲

Java NIO 和 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。

Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。在 Java NIO 中把数据读取到一个缓冲区中，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。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

2.阻塞与非阻塞 IO

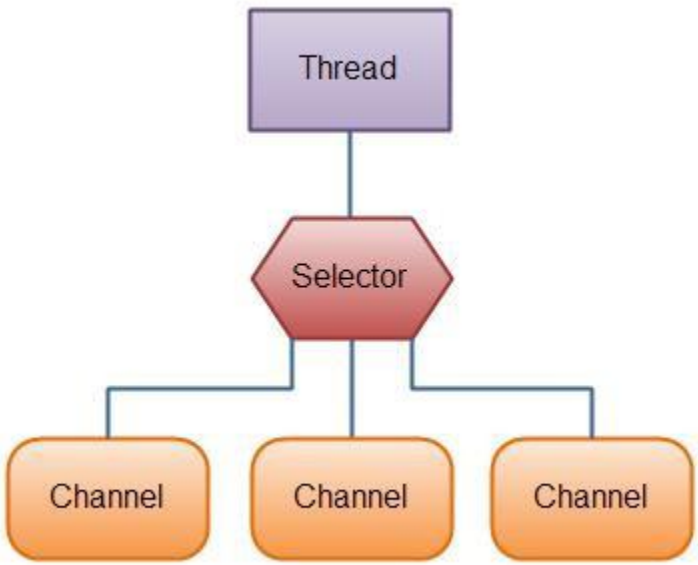
Java IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write() 时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

3.选择器（Selector）

Java NIO 的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里

已经有可以处理的输入，或者选择已准备写入的通道。为了将 Channel 和 Selector 配合使用，必须**将 channel 注册到 selector 上**，通过 SelectableChannel.register()方法来实现。这种选择机制，使得一个单独的线程很容易来管理多个通道。**只要 Channel 向 Selector 注册了某种特定的事件，Selector 就会监听这些事件是否会发生，一旦发生某个事件，便会通知对应的 Channel。使用选择器，借助单一线程，就可对数量庞大的活动 I/O 通道实施监控和维护。**

例如，在一个聊天服务器中。这是在一个单线程中使用一个 Selector 处理 3 个 Channel 的图示：



要使用 Selector，得向 Selector 注册 Channel，然后调用它的 select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。反应器模式。

NIO 的原理。

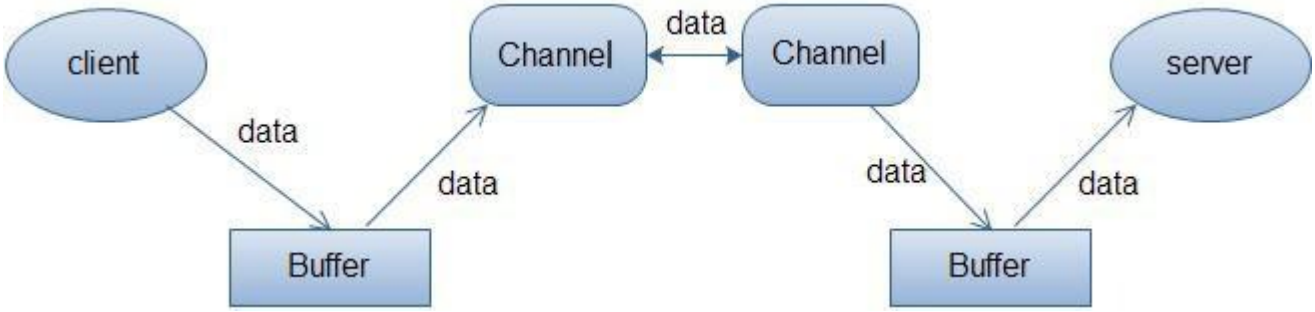
在 NIO 中有几个核心对象：缓冲区（Buffer）、通道（Channel）、选择器（Selector）。

一．缓冲区（Buffer）

缓冲区实际上是一个容器对象，其实就是一个数组，在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，它也是写入到缓冲区中的；任何时候访问 NIO 中的数据，都是将它放到

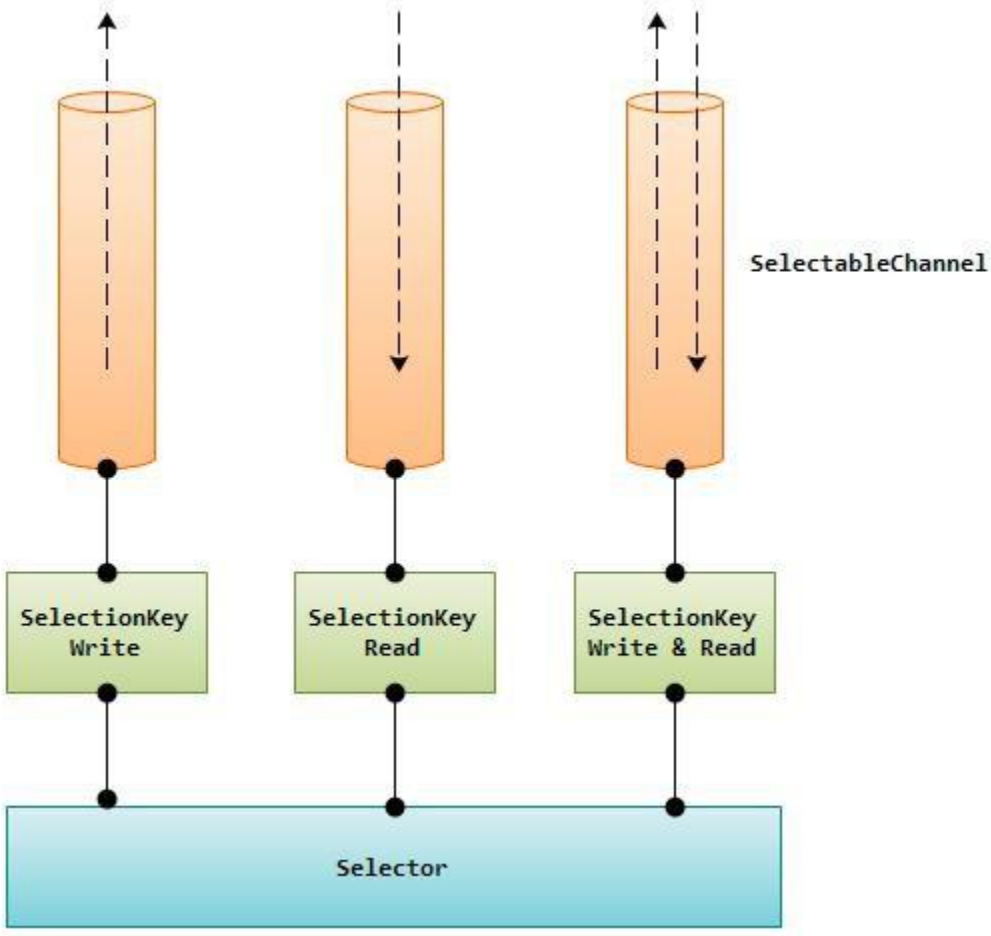
缓冲区中。在 NIO 中，所有的缓冲区类型都继承于抽象类 Buffer，最常用的就是 ByteBuffer。

而在面向流 I/O 系统中，所有数据都是直接写入或者直接将数据读取到 Stream 对象中。



二．通道（Channel）

通道是一个对象，通过它可以读取和写入数据，所有数据都通过 Buffer 对象来处理。我们永远不会将字节直接写入通道中，相反是将数据写入包含一个或者多个字节的缓冲区。同样不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。**通道与流的不同之处**在于**通道是双向的**而流只是在一个方向上移动(一个流必须是 InputStream 或者 OutputStream 的子类，比如 InputStream 只能进行读取操作，OutputStream 只能进行写操作)，而通道是双向的，可以用于读、写或者同时用于读写。



从图中可以看出，当有读或写等任何注册的事件发生时，可以从 Selector 中获得相应的SelectionKey，同时从 SelectionKey 中可以找到发生的事件和该事件所发生的具体的 SelectableChannel，以获得客户端发送过来的数据。

三．选择器（Selector）

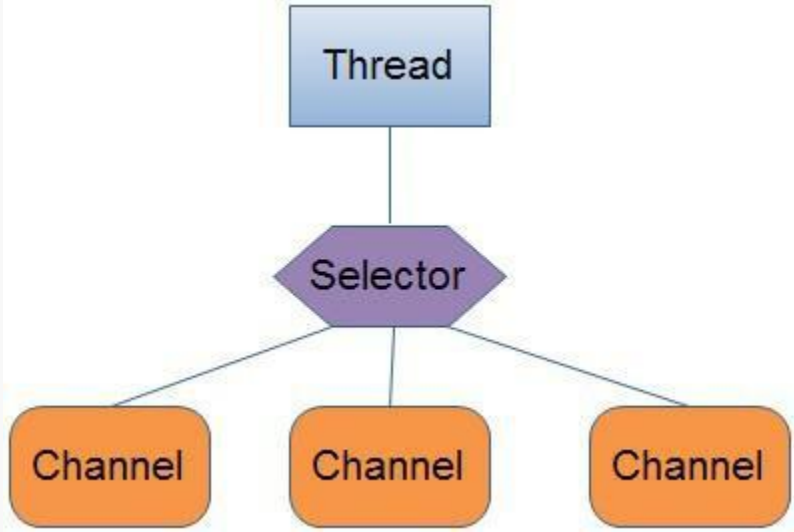
NIO 有一个主要的类 Selector,这个类似一个观察者，只要我们把需要探知的 socketchannel 告诉 Selector,我们接着做别的事情，**当有事件发生时，他会通知我们，传回一组 SelectionKey,我们读取这些 Key,就会获得我们刚刚注册过的 socketchannel,然后，我们从这个 Channel 中读取数据**，放心，包准能够读到，接着我们可以处理这些数据。

Selector 内部原理实际是在做一个**对所注册的 channel 的轮询访问**，不断地轮询，一旦轮询到一个 channel 有所注册的事情发生，比如数据来了，他就会站起来报告，**交出一把钥匙**，让我们**通过这把钥匙**来读取这个 channel 的内容。

Selector 的作用就是用来轮询每个注册的 Channel，一旦发现 Channel 有注册的事件发生，便获取事件然后进行处理。用单线程处理一个 Selector，然后

通过 Selector.select()方法来获取到达事件，在获取了到达事件之后，就可以逐个地对这些事件进行响应处理。

Selector 类是 NIO 的核心类，Selector 能够检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的响应处理。这样一来，只是用一个单线程就可以管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销。

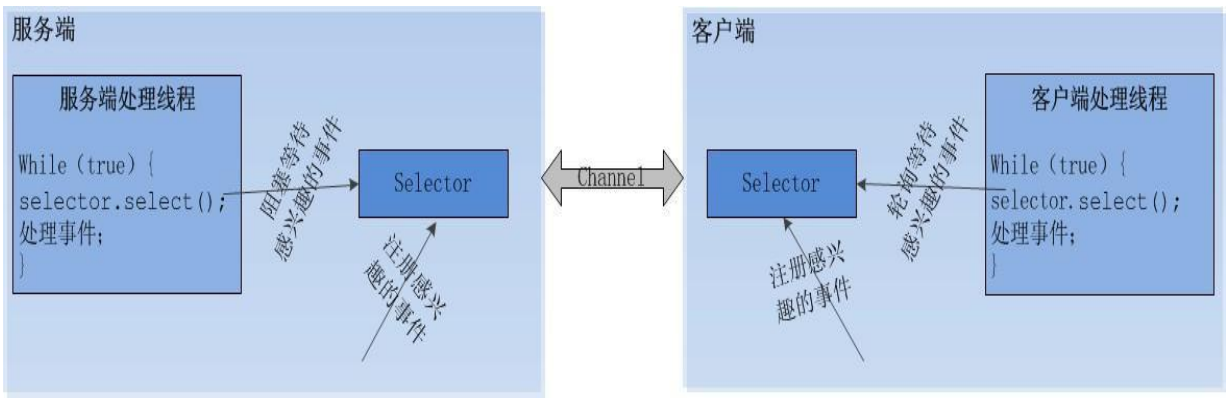


Java NIO 的服务端**只需启动一个专门的线程**来处理所有的 IO 事件，这种通信模型是怎么实现的呢？java NIO 采用了**双向通道（channel）**进行数据传输，而不是单向流（stream），**在通道上可以注册我们感兴趣的事件**。一共有以下**四种事件：**

事件 名	对应 值

服务端接收客户端连接事件	SelectionKey.OP_ACCEPT(16)
客户端连接服务端事件	SelectionKey.OP_CONNECT(8)
读事件	SelectionKey.OP_READ(1)
写事件	SelectionKey.OP_WRITE(4)

服务端和客户端各自维护一个管理通道的对象，我们称之为 selector，该对象能检测一个或多个通道（channel）上的事件。我们以服务端为例，如果服务端的 selector 上注册了读事件，某时刻客户端给服务端送了一些数据，阻塞 I/O 这时会调用 read()方法阻塞地读取数据，而 NIO 的服务端会在 selector 中添加一个读事件。服务端的处理线程会轮询地访问 selector，如果访问 selector 时发现有感兴趣的事件到达，则处理这些事件，如果没有感兴趣的事件到达，则处理线程会一直阻塞直到感兴趣的事件到达为止。下面是我理解的 java NIO 的通信模型示意图：



12.面向对象的 6 个基本原则（设计模式的 6 个基本原则）。

s(Single-Resposibility Principle): 单一职责原则。
o(Open-Closed principle): 开放封闭原则。
l(Liskov-Substituion Principle): 里氏替换原则。
i(Interface-Segregation Principle): 接口隔离原则。
d(Dependecy-Inversion Principle): 依赖倒置原则。
合成/聚合复用。

单一职责：是指一个类的功能要单一，一个类只负责一个职责。一个类只做它该做的事情(高内聚)。在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则。

开放封闭：软件实体应当对扩展开放，对修改关闭。对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。对修改封闭，意味着类一旦设计完成，就可以独立其工作，而不要对类尽任何修改。在开发阶段，我们都知道，如果对一个功能进行扩展，如果只是一味地对方法进行修改，可能会造成一些问题，诸如可能会引入新的 bug，或者增加代码的复杂度，对代码结构造成破坏、冗余，还需要重新进行全面的测试。那么该怎么解决这些问题？很简单，这就需要系统能够支持扩展，只有扩展性良好的系统，才能在不进行修改已有实现代码的基础上，引进新的功能。

里氏替换：任何使用基类的地方，都能够使用子类替换，而且在替换子类后，系统能够正常工作。子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。 一个软件实体如果使用的是一个基类，那么当把这个基类替换成继承该基类的子类，程序的行为不会发生任何变化。软件实体察觉不出基类对象和子类对象的区别。

接口隔离： 即应该将接口粒度最小化，将功能划分到每一个不能再分的子角色，为每一个子角色创建接口，通过这样，才不会让接口的实现类实现一些不必要的功能。 建立单一的接口，不要建立臃肿的庞大的接口，也就是说接口的方法尽量少。接口要小而专，绝不能大而全。臃肿的接口是对接口的污染， 既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。

依赖倒置：即我们的类要依赖于抽象，而不是依赖于具体，也就是我们经常听到的“要面向接口编程”。（该原则说得具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代）。 依赖倒置原则的本质就是通过抽象（抽象类或接口）使各个类或模块的实现彼此独立，不相互影响，实现模块间的松耦合。减少类间的耦合性。

合成/聚合复用：优先使用聚合或合成关系复用代码。

13.JDK 源码中用到的设计模式 。

1.**单例模式**： java.lang.Runtime 。

Runtime 类封装了 Java 运行时的环境。每一个 java 程序实际上都是启动了一个 JVM 进程，那么每个 JVM 进程都是对应这一个 Runtime 实例，此实例是由 JVM 为其实例化的。每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接。由于 Java 是单进程的，所以，在一个 JVM 中，Runtime 的实例应该只有一个。所以应该使用单例来实现。一般不能实例

化一个 Runtime 对象，应用程序也不能创建自己的 Runtime 类实例，但可以通过 getRuntime 方法获取当前 Runtime 运行时对象的引用。
2.**享元模式**：String 常量池和 Integer 等包装类的缓存策略：Integer.valueOf(int i)等。
3.**原型模式**：Object.clone；Cloneable。
4.**装饰器模式**：IO 流中。
5.**迭代器模式**：Iterator 。

14.执行 Student s = new Student();在内存中做了哪些事情?
1.类加载的过程。
2.对象初始化的顺序（对象实例化的过程）。 见 P28
3.在栈内存为 s 开辟空间，把对象地址赋值给 s 变量 。

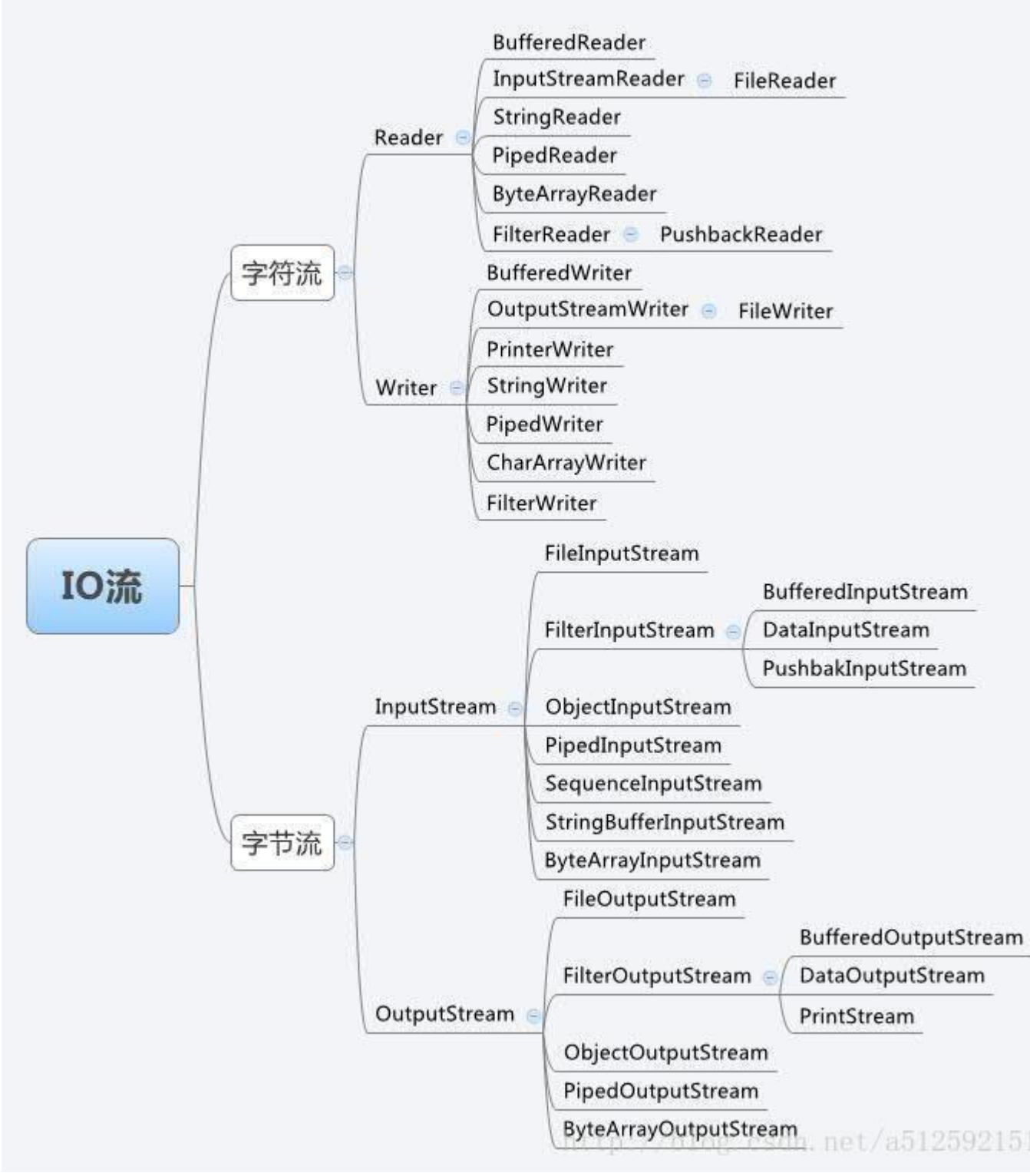
15.你知道的开源软件有哪些？
1.JDK
2.Eclipse
3.Tomcat
4.MySQL

16.String 型变量如何转成 int 型变量，反过来呢？
String num =“1000〞;
int id=Integer.valueOf(num);
————————————————————
Int a=1000;
String string=String.valueOf(a);

17.怎么判断数组是 null 还是为空?
输出 array.length 的值，如果是 0,说明数组为空。如果是 null 的话，会抛出空指针异常。

18.怎样让一个线程放弃锁。
Object.wait();
condition.await();

19.IO 里面常见的类。



20. xml 解析方式。

DOM 和 SAX 的不同：

1. DOM 是基于内存的，不管文件有多大，都会把 xml 文档都读到内存中，从而消耗很大的内存空间。而 SAX 解析 xml 文件采用的是事件驱动的方式，当

某个事件被触发时，获取相应的 XML 的部分数据读到内存进行解析，只占用了少量的内存空间。

2. DOM 可以读取 XML，也可以向 XML 文件中插入数据，而 SAX 却只能对 XML 进行读取，而不能在文件中插入数据。

3.DOM 可以指定要访问的元素进行随机访问，而 SAX 则不行。SAX 是从文档开始执行遍历的，并且只能遍历一次。也就是说不能随机访问 XML 文件，只能从头到尾的将 XML 文件遍历一次，但可以随时停止解析。