字节（B） byte

位（比特） bit

1字节=8位(1 byte = 8bit)

基础

**OOP的特点：**

OOP中引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。

**1、三大特性**

**封装：**

隐藏了类的内部实现机制，可以在不影响使用的情况下改变类的内部结构，同时也保护了数据。对外界而已它的内部细节是隐藏的，暴露给外界的只是它的访问方法。

**继承：**

为了重用父类代码。两个类若存在IS-A的关系就可以使用继承。同时继承也为实现多态做了铺垫。

**多态：**

发送消息给某个对象，让该对象自行决定响应何种行为。

程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定。

对于面向对象而言，多态分为编译时多态和运行时多态。其中编辑时多态是静态的，主要是指方法的重载，它是根据参数列表的不同来区分不同的函数，通过编译之后会变成两个不同的函数。而运行时多态是动态的，它是通过动态绑定来实现的，也就是我们所说的多态性。

**2、动态绑定和静态绑定**

把一个方法与其所在的类/对象关联起来叫做方法的绑定。绑定分为静态绑定（前期绑定）和动态绑定（后期绑定）。

**静态绑定**（前期绑定）是指：在程序运行前就已经知道方法是属于那个类的，在编译的时候就可以连接到类中，定位到这个方法。

在Java中，final、private、static修饰的方法以及构造函数都是静态绑定的，不需程序运行，不需具体的实例对象就可以知道这个方法的具体内容。

**动态绑定**（后期绑定）是指：在程序运行过程中，根据具体的实例对象才能具体确定是哪个方法。

动态绑定是多态性得以实现的重要因素，它通过方法表来实现：每个类被加载到虚拟机时，在方法区保存元数据，其中，包括一个叫做 方法表（method table）的东西，表中记录了这个类定义的方法的指针，每个表项指向一个具体的方法代码。如果这个类重写了父类中的某个方法，则对应表项指向新的代码实现处。从父类继承来的方法位于子类定义的方法的前面。

动态绑定的编译、运行原理：

（1）编译：

我们知道，向上转型时，用父类引用执行子类对象，并可以用父类引用调用子类中重写了的同名方法。但是不能调用子类中新增的方法，为什么呢？

   在代码的编译阶段，编译器通过声明对象的类型（即引用本身的类型） 在方法区中该类型的方法表中查找匹配的方法（最佳匹配法：参数类型最接近的被调用），如果有则编译通过。（这里是根据声明的对象类型来查找的，所以此处是查找 Father类的方法表，而Father类方法表中是没有子类新增的方法的，所以不能调用。）

  编译阶段是确保方法的存在性，保证程序能顺利、安全运行。

 （2）运行：

编译阶段在声明对象类型的方法表中查找方法，只是为了安全地通过编译（也为了检验方法是否是存在的）。

而在实际运行这条语句时，在执行 Father ft=new Son(); 这一句时创建了一个Son实例对象，然后在 ft.say() 调用方法时，JVM会把刚才的son对象压入操作数栈，用它来进行调用。

而用实例对象进行方法调用的过程就是动态绑定：根据实例对象所属的类型去查找它的方法表，找到匹配的方法进行调用。我们知道，子类中如果重写了父类的方法，则方法表中同名表项会指向子类的方法代码；若无重写，则按照父类中的方法表顺序保存在子类方法表中。故此：动态绑定根据对象的类型的方法表查找方法是一定会匹配（因为编译时在父类方法表中以及查找并匹配成功了，说明方法是存在的。这也解释了为何向上转型时父类引用不能调用子类新增的方法：在父类方法表中必须先对这个方法的存在性进行检验，如果在运行时才检验就容易出危险——可能子类中也没有这个方法）。

**# 静态分派**

在编译期间，可以根据变量的静态类型确定方法执行版本的分派动作称作静态分派。典型的是方法的重载。

**# 动态分派**

在编译期间确定不下来，只有在方法调用时，根据对象的实际类型确定方法执行版本的分派动作称作动态分派。 重写。

**动态分派的实现**：

类在方法区中有一个方法表，方法表中存放着各个方法的实际入口地址，如果某个方法在子类没有被重写，那么子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类的实现入口。

方法的接收者和方法的参数统称为方法的**宗量**。 根据分派基于宗量多少（接收者是一个宗量，参数是一个宗量），可以将分派分为单分派和多分派。

**# 单分派**

单分派是指根据一个宗量就可以知道调用目标（即应该调用哪个方法）。

**# 多分派**

多分派需要根据多个宗量才能确定调用目标。

**3、基础规范**

Java命名规则：标识符由字母、数字、下划线、美元符号或者人民币符号组成，且首字母不能是数字。

**4、关键字的范围**



**一、String类**

**（1）String**

**String类初始化后是不可变的(immutable)**

1.String类是final类，也即意味着String类不能被继承，并且它的成员方法都默认为final方法。在Java中，被 final 修饰的类是不允许被继承的，并且该类中的成员方法都默认为final 方法。

String 用 final 修饰的原因，主要是为了”安全性“和”效率“的缘故：

（1）由于String类不可变，所以就不会被修改，这就避免了多线程竞争的线程安全问题；

（2）String 类在程序中出现的频率比较高，为了避免安全隐患，在它每次出现时都用final 来修饰，如果使用的不好，会出现很明显的性能问题，所以干脆做成final类的，放在常量池实现数据共享，节省资源，提高效率，可以在JVM里做很多优化（static final修饰的变量所在类不用初始化就可以直接使用变量）。

2.String类其实是通过char数组来保存字符串的。

3.**String对象一旦被创建就是固定不变的了，对String对象的任何改变都不影响到原对象，相关的任何change操作都会生成新的对象.**

**（2）、字符串常量池（**字符串池的优点就是避免了相同内容的字符串的创建，节省了内存，省去了创建相同字符串的时间，同时提升了性能**）**

每当我们创建字符串常量时，JVM会首先检查字符串常量池，如果该字符串已经存在常量池中，那么就直接返回常量池中的实例引用。如果字符串不存在常量池中，就会实例化该字符串并且将其放到常量池中。由于 String 字符串的不可变性我们可以十分肯定常量池中一定不存在两个相同的字符串。

Java中的常量池，实际上分为两种形态：静态常量池和运行时常量池。

2.1 所谓静态常量池，即\*.class文件中的常量池，class文件中的常量池不仅仅包含字符串(数字)字面量，还包含类、方法的信息，占用class文件绝大部分空间。

2.2 而运行时常量池，则是 jvm 虚拟机在完成类装载操作后，将 class 文件中的常量池载入到内存中，并保存在方法区中，我们常说的常量池，就是指方法区中的运行时常量池。

**(3)、字符串的创建方式**

3.1 单独使用""引号创建的字符串都是常量，编译期就已经确定存储到String Pool（字符串常量池）中；

3.2 使用 new String("")创建的对象会存储到 heap 中,是运行期新创建的；

new 创建字符串时首先查看池中是否有相同值的字符串，如果有，则拷贝一份到堆中，然后返回堆中的地址；如果池中没有，则在堆中创建一份，然后返回堆中的地址（同时在常量池中也创建一个对象）

其中String.intern（）方法的不同之处：

① JDK 1.6 中 intern（）方法会把首次遇到的字符串实例复制到永久代中，返回的也是永久代中这个实例的引用。

② JKD 1.7 中 intern（）方法不会复制实例，只是在常量池中记录首次出现的实例引用（返回的是堆中的对象引用）。

3.3 使用只包含常量的字符串连接符如"aa" + "aa"创建的也是常量,编译期就能确定,已经确定存储到String Pool中；

3.4 使用包含变量的字符串连接符如"aa" + s1创建的对象是运行期才创建的,存储在heap中；

**（4）、关于equals和==**

4.1 对于==，如果作用于基本数据类型的变量（byte,short,char,int,long,float,double,boolean ），则直接比较其存储的"值"是否相等；如果作用于引用类型的变量（String），则比较的是所指向的对象的地址（即是否指向同一个对象）。

4.2 equals方法是基类Object中的方法，因此对于所有的继承于Object的类都会有该方法。在Object类中，equals方法是用来比较两个对象的引用是否相等，即是否指向同一个对象。

4.3 对于equals方法，注意：equals方法不能作用于基本数据类型的变量。如果没有对equals方法进行重写，则比较的是引用类型的变量所指向的对象的地址；而String类对equals方法进行了重写，用来比较指向的字符串对象所存储的字符串是否相等。其他的一些类诸如Double，Date，Integer等，都对equals方法进行了重写用来比较指向的对象所存储的内容是否相等。

**（5）、String相关的+**

5.1 String c = "xx" + "yy " + a + "zz" + "mm" + b; 实质上的实现过程是： String c = new StringBuilder("xxyy ").append(a).append("zz").append("mm").append(b).toString();

5.2 非常量字符串相加时，由于相加的变量中存放的是字符串的地址引用，在编译时无法确切地知道其他具体的值，也就没有办法对其进行优化处理，这时为了达到连接的效果，其内部采用了 StringBuilder 的机制进行处理，实际上是产生了一个StringBuilder对象和一个String对象。

5.3 字符串常量间的相加并不会引起效率问题，因为会进行编译器的优化。

**（6）、String、StringBuffer、StringBuilder的区别**

6.1 可变与不可变：String是不可变字符串对象，StringBuilder和StringBuffer是可变字符串对象（其内部的字符数组长度可变）。

6.2 是否多线程安全：String中的对象是不可变的，也就可以理解为常量，显然线程安全。StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，只是StringBuffer 中的方法大都采用了synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是非线程安全的。

6.3 String、StringBuilder、StringBuffer三者的执行效率：

StringBuilder > StringBuffer > String 当然这个是相对的，不一定在所有情况下都是这样。

**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**

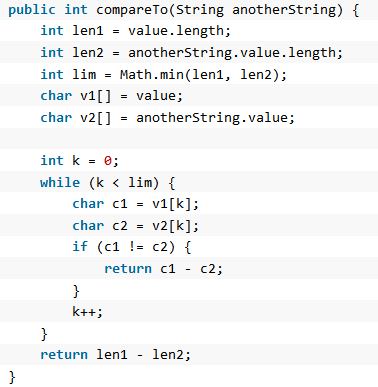
**StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况**

**（7）、String中的final用法和理解**

final只对引用的"值"(即内存地址)有效，它迫使引用只能指向初始指向的那个对象，改变它的指向会导致编译期错误。至于它所指向的对象的内容（值）变化，final是不负责的。

**（8）、compareto（）方法（字典顺序）**

按**字典顺序**比较两个字符串。该比较基于字符串中各个字符的 Unicode 值。按字典顺序将此 String 对象表示的字符序列与参数字符串所表示的字符序列进行比较。如果按字典顺序此 String 对象位于参数字符串之前，则比较结果为一个负整数。如果按字典顺序此 String 对象位于参数字符串之后，则比较结果为一个正整数。如果这两个字符串相等，则结果为 0。（左端对齐，按位比较）



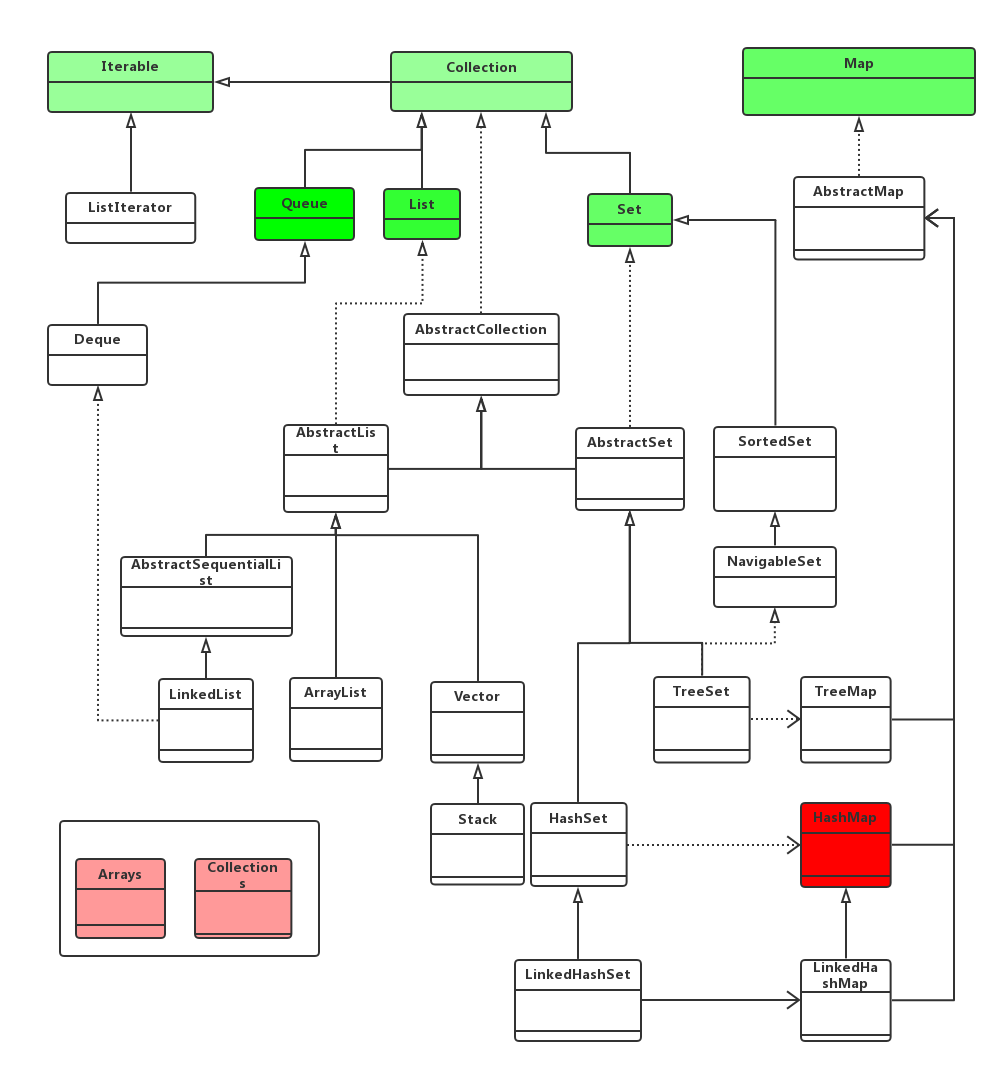
比较规则：

首先取出两个字符串的长度，比较较小的长度内，两者是否相等。

      若不相等，则直接返回该位置字符的ASCII码相减后的值。

      若各位置都相等，则将两个字符串长度的差值返回。

**二、集合类**



Java中的集合主要分为两大类：

（1）Collecton：可以理解为主要存放的是单个对象，Collection继承了Iterate接口，Iterate用于集合内迭代器抽象接口，其子类均实现接口中方法。

（2）Map：可以理解为主要存储key-value类型的对象

**1、Map**

1.1.  关于**HashMap**的一些说法：

a)  HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。HashMap的底层结构是一个数组，数组中的每一项是一条链表。

b)  HashMap的实例有俩个参数影响其性能： “初始容量” 和 装填因子。

c)  HashMap实现不同步，线程不安全。  HashTable线程安全

d)  HashMap中的key-value都是存储在Entry中的。

e)  HashMap可以存null键和null值，不保证元素的顺序恒久不变，它的底层使用的是数组和链表，通过hashCode()方法和equals方法保证键的唯一性。

f)  解决冲突主要有三种方法：开放定址法（再散列法），拉链法，再哈希法。HashMap是采用拉链法解决哈希冲突的。

g）多线程的并发问题：多线程put时可能导致元素丢失；resize时可能出现环链导致死循环；fail-fast策略：在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException。

以下是resize中调用transfer方法的部分代码：

while(null != e) {

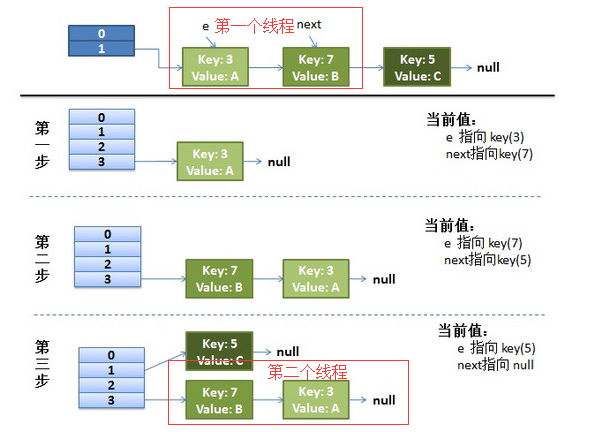
    Entry<K,V> next = e.next; //指向下一个节点

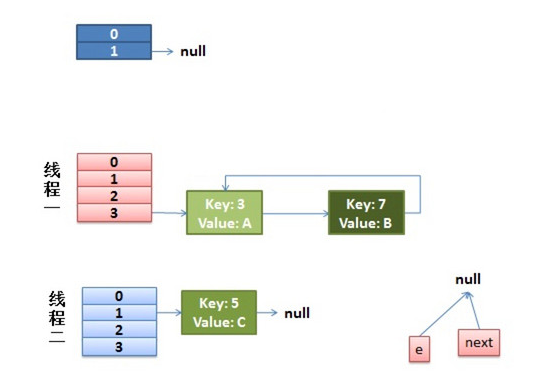
    e.next = newTable[i]; //当前元素e插入到新表的头部（插入到表尾的复杂度是 O（N））

    newTable[i] = e; //新的头指针

    e = next; //处理下一个元素

}





解决方法：使用HashTable、Collections.synchronizedMap将HashMap包装起来、ConcurrentHashMap替换HashMap。

h）为什么String这样的wrapper类适合作为键？

因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还可以保证线程安全。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。

注： 链表法是将相同hash值的对象组成一个链表放在hash值对应的槽位；

**hash 碰撞解决的方法：**

   ① 用开放定址法解决冲突的做法是：就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

插入时可能会出现多次冲突的现象，容易产生堆积问题，不适于大规模的数据存储。

删除结点不能简单地将被删结点置为空，否则将截断在它之后填人散列表的同义词结点的查找路径。这是因为各种开放地址法中，空地址单元是查找失败的条件。因此在用开放地址法处理冲突的散列表上执行删除操作，只能在被删结点上做删除标记，而不能真正删除结点。

（1）**线性探测再散列**

di=1，2，3，…，m-1。

这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

**（2）平方探测再散列**

IMG_261

这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

 ②  拉链法解决冲突的做法是： 将所有关键字为同义词（hash值相同）的结点链接在同一个单链表中 。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数 组T[0..m-1]。凡是散列地址为i的结点，均插入到以T[i]为头指针的单链表中。T中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于1，但一般均取α≤1。拉链法适合未规定元素的大小。

处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，**因此平均查找长度较短；**

**删除结点的操作易于实现**。只要简单地删去链表上相应的结点即可。

在 jdk1.8 版本后，java对HashMap做了改进，在链表长度大于8（且桶的数量超过64）的时候，将后面的数据存在红黑树中，以加快检索速度。

③ 再哈希法：同时构造多个不同的哈希函数，当一个冲突时使用下一个，直到不冲突为止。

**④ 建立公共溢出区：**将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

1.2. **Hashtable 和 HashMap的区别**：

a)   继承不同

 public class Hashtable extends Dictionary implements Map

public class HashMap extends  AbstractMap implements Map

b)  Hashtable中的方法是同步的，而HashMap中的方法在缺省情况下是非同步的。在多线程并发的环境下，可以直接使用Hashtable，但是要使用HashMap的话就要自己增加同步处理了。

c)  Hashtable 中， key 和 value 都不允许出现 null 值。 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，即可以表示 HashMap 中没有该键，也可以表示该键所对应的值为 null。因此，在 HashMap 中不能由 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。

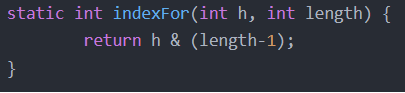
d)  哈希值的使用不同，Hashtable计算hash是直接使用key的hashcode对table数组的长度直接进行取模。HashMap计算hash对key的hashcode进行了二次hash，以获得更好的散列值，然后对table数组长度取摸。

e)  Hashtable和HashMap它们两个内部实现方式的数组的初始大小和扩容的方式。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，增加的方式是old\*2，所以一定是2的指数。

f)  两个遍历方式的内部实现上不同。Hashtable、HashMap都使用了Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式 。

**为什么 HashMap 容量一定要为2的幂呢？**

答：HashMap中的数据结构是数组+单链表的组合，我们希望元素存放的更均匀，最理想的效果就是数组中每个位置都只有一个元素，这样查询的时候效率最高，不需要遍历单链表，而且空间利用率最大，时间复杂度最优。要达到分布最均匀，最简单的办法就是使用取余运算。



**当容量一定是2^n时，h & (length - 1) == h % length**，它俩是等价不等效率的，位运算效率非常高。这里的 h 是二次哈希之后的值。

长度16或者其他2的幂，length - 1的值是所有二进制位全为1，这种情况下，index的结果等同于hashcode后几位的值。只要输入的hashcode（经过hash函数之后的值）本身分布均匀，hash算法（求下标索引）的结果就是均匀的。所以HashMap的默认长度为16，是为了降低hash碰撞的几率。

桶初始化桶数组设置太大，就会浪费内存空间，16是一个折中的大小，既不会像1，2，3那样放几个元素就扩容，也不会像几千几万那样可以只会利用一点点空间从而造成大量的浪费。

g) hashmap中没有contains方法，hashtable中有此方法。

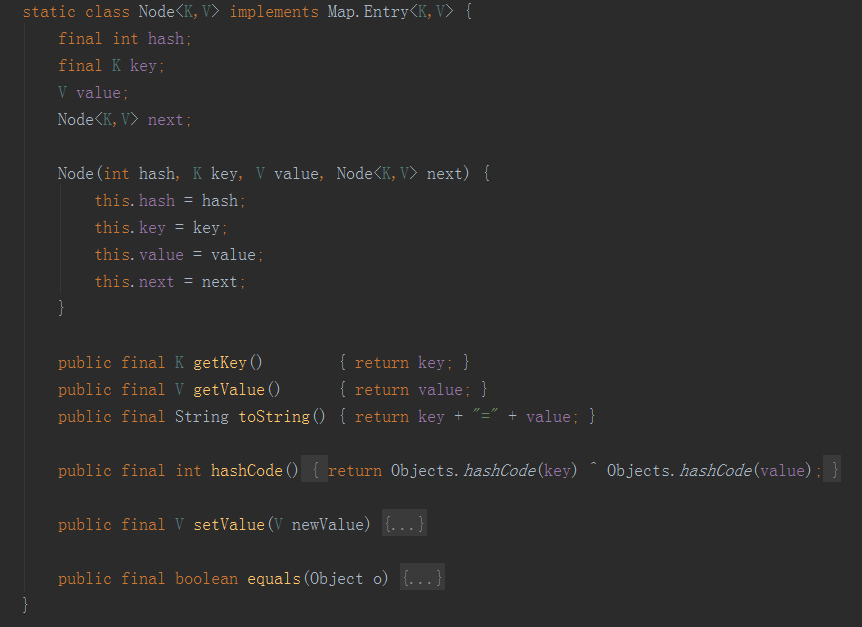
注：  HashSet子类依靠hashCode()和equal()方法来区分重复元素。

     HashSet内部使用Map保存数据，即将HashSet的数据作为Map的key值保存，这也是HashSet中元素不能重复的原因。而Map中保存key值的,会去判断当前Map中是否含有该Key对象，内部是先通过key的hashCode,确定有相同的hashCode之后，再通过equals方法判断是否相同。

1.3 **HashSet** 和 HashMap、Hashtable 的区别

HashSet 不是key value结构，仅仅是存储不重复的元素，相当于简化版的HashMap，只是包含HashMap中的key而已。HashSet内部就是使用HashMap实现，只不过HashSet里面的HashMap所有的value都是同一个Object而已，因此HashSet也是非线程安全的。

1.4 **HashMap和Hashtable的实现原理**



HashMap和Hashtable的底层实现都是数组（Node类型的数组，保存头结点）+链表（next域）结构实现的。添加、删除、获取元素时都是先计算hash，根据hash和table.length计算index也就是table数组的下标，然后进行相应操作，但是HashMap和HashTable计算hash的方法不同。

下面以HashMap为例进行说明：

**put（）**：HashMap会对null值key进行特殊处理，总是放到table[0]位置。

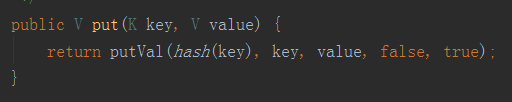
put过程是先计算hash然后通过hash与table.length取摸计算index值，然后将key放到table[index]位置，当table[index]已存在其它元素时，会在table[index]位置形成一个链表，将新添加的元素放在table[index]，原来的元素通过Entry的next进行链接，这样以链表形式解决hash冲突问题，当元素数量达到临界值(capactiy\*factor)时，则进行扩容，table数组长度变为table.length\*2。

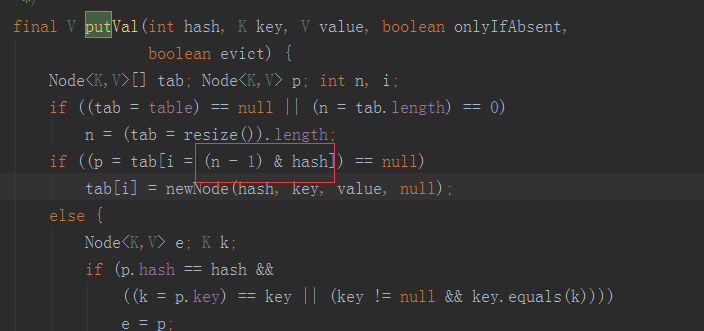
jdk 1.7的扩容是插入之前判断，而jdk 1.8是插入之后再判断是否需要扩容；

在1.8中扩容时保持了**原来链表中的顺序**，避免出现死循环。

默认加载因子 (0.75) 在时间和空间成本上寻求一种折衷。**加载因子过高虽然减少了空间开销，但同时也增加了查询成本。**

jdk 1.8中的hashmap还引入了**红黑树**的数据结构，当桶的数量超过64且链表的长度超过8时进行树化，链表就会转成红黑树结构，提高查询效率。





**get（）**：同样当key为null时会进行特殊处理，在table[0]的链表上查找key为null的元素。get的过程是先计算hash然后通过hash与table.length取摸计算index值，然后遍历table[index]上的链表，直到找到key，然后返回。

**remove**方法和put get类似，计算hash，计算index，然后遍历查找，将找到的元素从table[index]链表移除。

**resize**方法在hashmap中并没有公开，这个方法实现了非常重要的hashmap扩容，具体过程为：先创建一个容量为 table.length\*2 的新table，修改临界值，然后把table里面元素计算hash值并使用hash与table.length\*2重新计算index放入到新的table里面。这里需要注意下是用每个元素的hash全部重新计算index，而不是简单的把原table对应index位置元素简单的移动到新table对应位置。

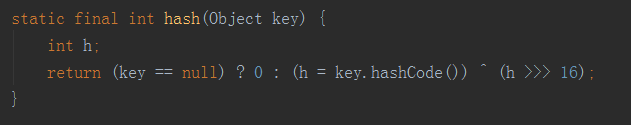
**clear**方法非常简单，就是遍历table然后把每个位置置为null，同时修改元素个数为0。需要注意的是clear方法只会清楚里面的元素，并不会重置capactiy。

**containsKey**方法是先计算hash然后使用hash和table.length取摸得到index值，遍历table[index]元素查找是否包含key相同的值。

**containsValue**方法就比较粗暴了，就是直接遍历所有元素直到找到value，由此可见HashMap的containsValue方法本质上和普通数组和list的contains方法没什么区别，你别指望它会像containsKey那么高效。

**hash函数（jdk 1.8）**：用了很多的异或，移位等运算，对key的hashcode进一步进行计算以及二进制位的调整来保证最终获取的存储位置尽量分布均匀。

自己的高16位和低16位异或是为了混合原始哈希码的高位和低位，加大低位的随机性，同时低位也加入了高位的部分特征，高位的信息也保存下来。



1.5 **LinkedHashMap（HashMap + 双向链表）**

LinkedHashMap维护着一个运行于所有条目的双向链表，此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序（insert-order）或者是访问顺序，其中默认的迭代访问顺序就是插入顺序，即可以按插入的顺序遍历元素，这点和HashMap有很大的不同。

所谓访问顺序(access-order)是指在迭代遍历列表中的元素时最近访问的元素会排在LinkedHashMap的尾部。

而HashMap则是按照key的顺序遍历元素的。

1.6 **TreeMap 和TreeSet**

**1.6.1 TreeMap的实现**

TreeMap是基于红黑树实现的，该集合根据其键的自然顺序进行排序。红黑树顾名思义就是节点是红色或者黑色的平衡二叉排序树，它通过颜色的约束来维持着二叉树的平衡。红黑树满足如下规则：

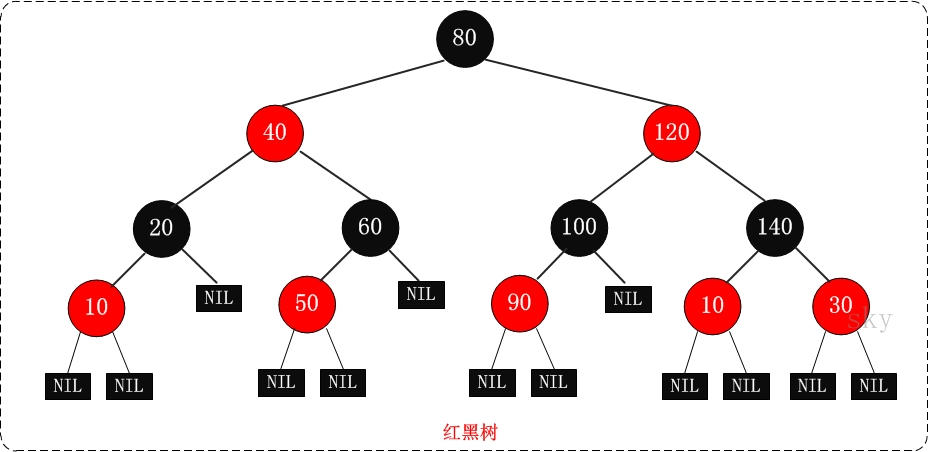
**1、每个节点都只能是红色或者黑色**

**2、根节点是黑色**

**3、每个叶节点（NIL节点，空节点）是黑色的。**

**4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。**

**5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。**



**红黑树的优势：**

红黑树相比普通的 avl （平衡树）树，在检索的时候效率其实差不多，都是通过平衡来二分查找。但对于插入、删除等操作效率提高很多，红黑树不像 avl 树一样追求绝对的平衡，他允许局部很少的不完全平衡，这样对于效率影响不大，但省去了很多没有必要的调平衡操作，avl 树调平衡有时候代价较大，所以效率不如红黑树。

红黑树能够以O(log2(N))的时间复杂度进行搜索、插入、删除操作。此外，任何不平衡都会在3次旋转之内解决，这一点是AVL所不具备的。

**红黑树的插入：**

将红黑树当作一颗二叉排序树，将节点插入，并将节点着为红色；然后通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。

**红黑树的删除：**

将红黑树当作一颗二叉排序树，将该节点从二叉排序树中删除；然后，通过"旋转和重新着色"等一系列操作来修正该树，使之重新成为一棵红黑树。

删除节点分三种情况：

（1）节点是叶节点，那么直接将该节点删除就OK了。

（2）节点只有一个孩子节点，直接删除该节点，并用该节点的唯一子节点顶替它的位置。

（3）节点有两个儿子，先找出它的后继节点，然后把“它的后继节点的内容”复制给“该节点的内容”，之后删除“它的后继节点”。

**put():**在TreeMap的put()的实现方法中主要分为两个步骤，第一：构建排序二叉树，第二：平衡二叉树（左旋、右旋、着色）。

**setColor()**:着色就是改变该节点的颜色，在红黑树中，它是依靠节点的颜色来维持平衡的。

**delete():**删除节点P的步骤是：找到一个替代子节点C来替代P，然后直接删除C，最后调整这棵红黑树。

**1.6.2 TreeMap和TreeSet的区别和联系**

TreeSet里面绝大部分方法都是直接调用TreeMap方法来实现的。

**相同点：**

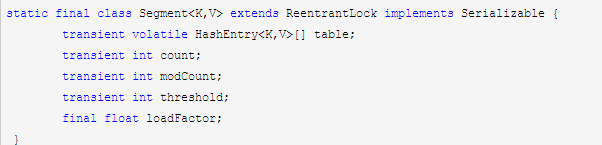
* TreeMap和TreeSet都是非同步集合，因此他们不能在多线程之间共享，不过可以使用方法Collections.synchroinzedMap()来实现同步。
* 运行速度都要比Hash集合慢，基于红黑树实现，所以他们内部对元素的操作时间复杂度为O(logN)，而HashMap/HashSet则为O(1)。
* TreeMap和TreeSet都是有序的集合，也就是说他们存储的值都是拍好序的。

**不同点：**

* 最主要的区别就是TreeSet和TreeMap分别实现Set和Map接口
* TreeSet只存储一个对象，而TreeMap存储两个对象Key和Value（仅仅key对象有序）。
* TreeSet中不能有重复对象，而TreeMap中可以存在重复对象。

**1.7 ConcurrentHashMap**

在JDK1.7版本中，ConcurrentHashMap的数据结构是由一个Segment数组（默认大小是16）和多个HashEntry组成，Segment数组的意义就是将一个大的 table 分割成多个小的table来进行加锁（锁分离技术），而每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样。



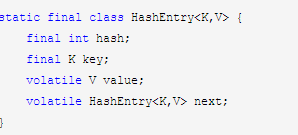
table：链表数组，每个数组元素是一个hash链表的头部。table是volatile的，使得每次都能读取到最新的table值而不需要同步。

count：Segment中元素的数量。

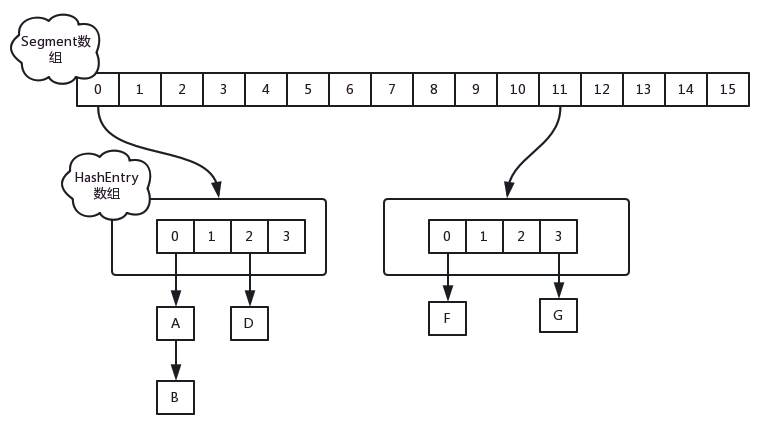
modCount：对table的结构进行修改的次数。

threshold：若 Segment 里的元素数量超过这个值，则就会对 Segment 进行扩容。

loadFactor：负载因子，threshold = capacity \* threshold。

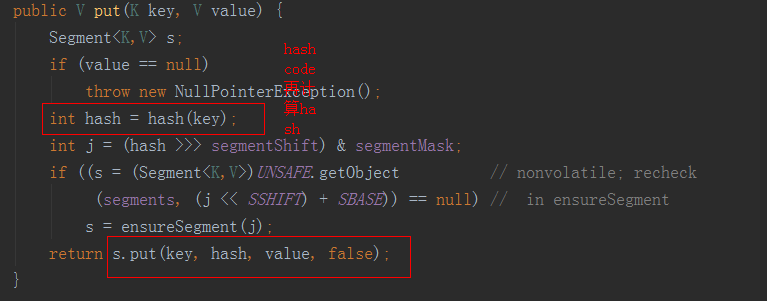


这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。



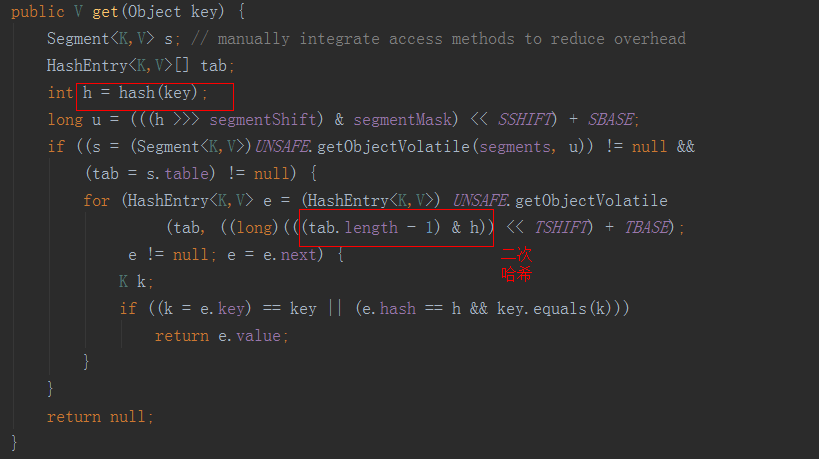
**1.7.1 put操作**

对于ConcurrentHashMap的数据插入，这里要进行两次Hash去定位数据的存储位置。当执行put操作时，会进行第一次key的hash来定位Segment的位置，如果该Segment还没有初始化，即通过CAS操作进行赋值，然后进行第二次hash操作，找到相应的HashEntry的位置，这里会利用继承过来的锁的特性，在将数据插入指定的HashEntry位置时会通过继承 ReentrantLock 的tryLock（）方法尝试去获取锁，如果获取成功就直接插入相应的位置，如果已经有线程获取该Segment的锁，那当前线程会以自旋的方式去继续的调用tryLock（）方法去获取锁，超过指定次数就挂起，等待唤醒。



**1.7.2 get操作**

ConcurrentHashMap的get操作跟HashMap类似，只是ConcurrentHashMap第一次需要经过一次hash定位到Segment的位置，然后再hash定位到指定的HashEntry，遍历该HashEntry下的链表进行对比，成功就返回，不成功就返回null。



**1.7.3 size操作**

计算ConcurrentHashMap的元素大小是一个有趣的问题，因为他是并发操作的，就是在你计算size的时候，他还在并发的插入数据，可能会导致你计算出来的size和你实际的size有相差（在你return size的时候，插入了多个数据），要解决这个问题，JDK1.7版本用两种方案：

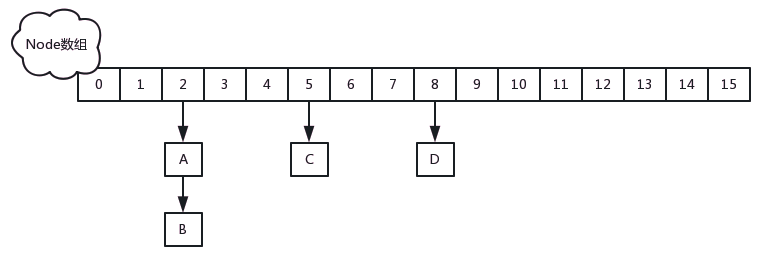
（1）第一种方案先使用不加锁的模式先尝试遍历两次ConcurrentHashMap计算size，如果两次遍历过程中所有segment中的modCount的和是一致的，则可以认为整个计算过程中的Map没有发生变化，计算的结果是准确的。

（2）第二种方案是如果第一种方案不符合，他就会给每个Segment加上锁，然后计算ConcurrentHashMap的size返回。



**1.7.4 concurrentHashMap 在 jdk1.7 和 jdk1.8 的比较**

JDK1.8 的实现已经摒弃了Segment的概念，而是直接用 Node数组+链表+红黑树的数据结构来实现，并发控制使用 Synchronized 和 CAS 来操作，整个看起来就像是优化过且线程安全的HashMap，虽然在JDK1.8中还能看到Segment的数据结构，但是已经简化了属性，只是为了兼容旧版本。



**总结：**

可以看出JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发，从JDK1.7版本的ReentrantLock + Segment + HashEntry，到 JDK1.8 版本中synchronized + CAS + HashEntry + 红黑树，比较如下：

1. JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）。
2. JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了。
3. JDK1.8使用红黑树来优化链表，对于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档
4. JDK1.8为什么使用内置锁synchronized来代替重入锁ReentrantLock，有以下几点：
   1. 因为粒度降低了，在相对而言的低粒度加锁方式，synchronized并不比ReentrantLock差，在粗粒度加锁中ReentrantLock可能通过Condition来控制各个低粒度的边界，更加的灵活，而在低粒度中，Condition的优势就没有了。
   2. JVM的开发团队从来都没有放弃synchronized，而且基于JVM的synchronized优化空间更大，使用内嵌的关键字比使用API更加自然。
   3. 在大量的数据操作下，对于JVM的内存压力，基于API的ReentrantLock会开销更多的内存，虽然不是瓶颈，但是也是一个选择依据。

**1.8 HashTable、Collections.synchronizedMap()、ConcurrentHashMap的比较**

HashTable给每一个方法加了锁，效率不高。

其中SynchronizedMap类是定义在Collections中的一个静态内部类，它实现了Map接口，返回的是一个SynchronizedMap 的实例，并对其中的每一个方法实现通过synchronized 关键字进行了同步控制。

Collections.synchronizedMap()和Hashtable一样，在调用map所有方法的时候，都对整个map进行同步（即不同线程可以同时调用同一个map）。**而ConcurrentHashMap的实现却更加精细，采取分段锁的方案。**ConcurrentHashMap在性能以及安全性方面，明显比Collections.synchronizedMap()更加有优势。同时，同步操作精确控制到段，所以，即使在遍历map时，其他线程试图对map进行数据修改，也不会抛出ConcurrentModificationException。

ConcurrentHashMap从类的命名就能看出，它必然是个HashMap。而Collections.synchronizedMap()可以接收任意Map实例，实现Map的同步。

**2、List**（有序结果、顺序遍历、索引、允许有重复值）

**2.1 ArrayList（动态数组实现）**

ArrayList是最常用的集合，其内部实现是一个数组，ArrayList的大小是可以动态扩充的。对于元素的随机访问效率高，其访问的时间复杂度为O(1)，对于数据的插入与删除，从尾部操作效率高，时间复杂度和随机访问一样是O(1)，若是从头部操作则效率会比较低，因为从头部插入或删除时需要移动后面所有元素，其时间复杂度为O(n-i)(n表示元素个数，i表示元素位置)。

当元素超出数组内容，会产生一个新数组（容量是原来的1.5倍），将原来数组的数据复制到新数组中，再将新的元素添加到新数组中。

2.1.1 初始化：

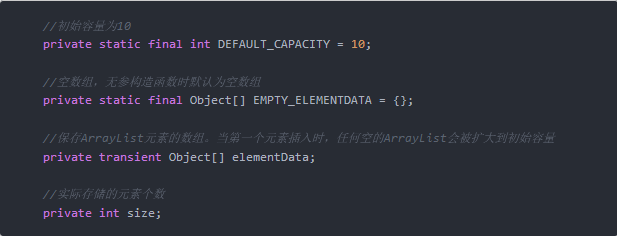
（1）ArrayList()构造一个初始容量为 10 的空列表，不够可以扩充。

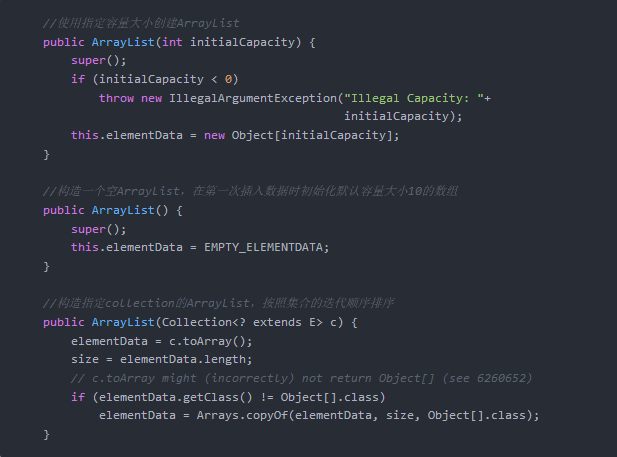
（2）ArrayList(int initialCapacity)构造一个具有指定初始容量的空列表。

2.1.2 线程不安全

非线程安全，创建线程安全的 ArrayList 可以使用 Collections.synchronizedList 进行包装或者并发包下的 CopyOnWriteArrayList 类。

2.1.3 源码实现



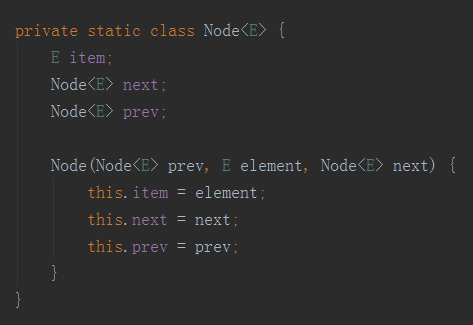




**2.2 LinkedList （链表实现）**

LinkList 对于随机访问效率是比较低的，因为它需要从头开始索引，所以其时间复杂度为O(i)。但是对于元素的增删，LinkList效率高，因为只需要修改前后指针即可，其时间复杂度为O(1)。

不可以在初始化时候指定大小，每次向其中加入元素时候，容量自动加1。



**2.3 Vector**

　与 ArrayList 类型一样，内部也是使用数组来存储对象，但是线程安全的，因为实现方法重写的时候，全部加上了同步关键字：synchronized；（一般不建议使用，性能消耗）。

（1）Vector有四个不同的构造方法。无参构造方法的容量为默认值10，仅包含容量的构造方法则将容量增长量明置为0。

    （2）注意扩充容量的方法ensureCapacityHelper。与ArrayList相同，Vector在每次增加元素（可能是1个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就先看构造方法中传入的容量增长量参数CapacityIncrement是否为0，如果不为0，就设置新的容量设置为旧容量加上容量增长量，如果为0，就设置新的容量为旧的容量的2倍，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量），而后同样用Arrays.copyof()方法（最终是还是调用了System.arraycopy）将元素拷贝到新的数组。

    （3）同样在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，Vector中也允许元素为null。

    （4）其他很多地方都与ArrayList实现大同小异，Vector现在已经基本不再使用。





**2.4 Stack**继承于vector（线程安全）

**3、Queue**

　　遵循FIFO（先入先出规则），内部出栈入栈方法，主要区别在于是否是阻塞入队或出队。

3.1 LinkedBlockingQueue

由链表实现的有界阻塞队列，但大小默认值为Integer.MAX\_VALUE，所以我们在使用LinkedBlockingQueue时建议手动传值，为其提供我们所需的大小，避免队列过大造成机器负载或者内存爆满等情况。

不接受null元素；实现被设计为主要用于生产者 - 消费者队列；实现是线程安全的，

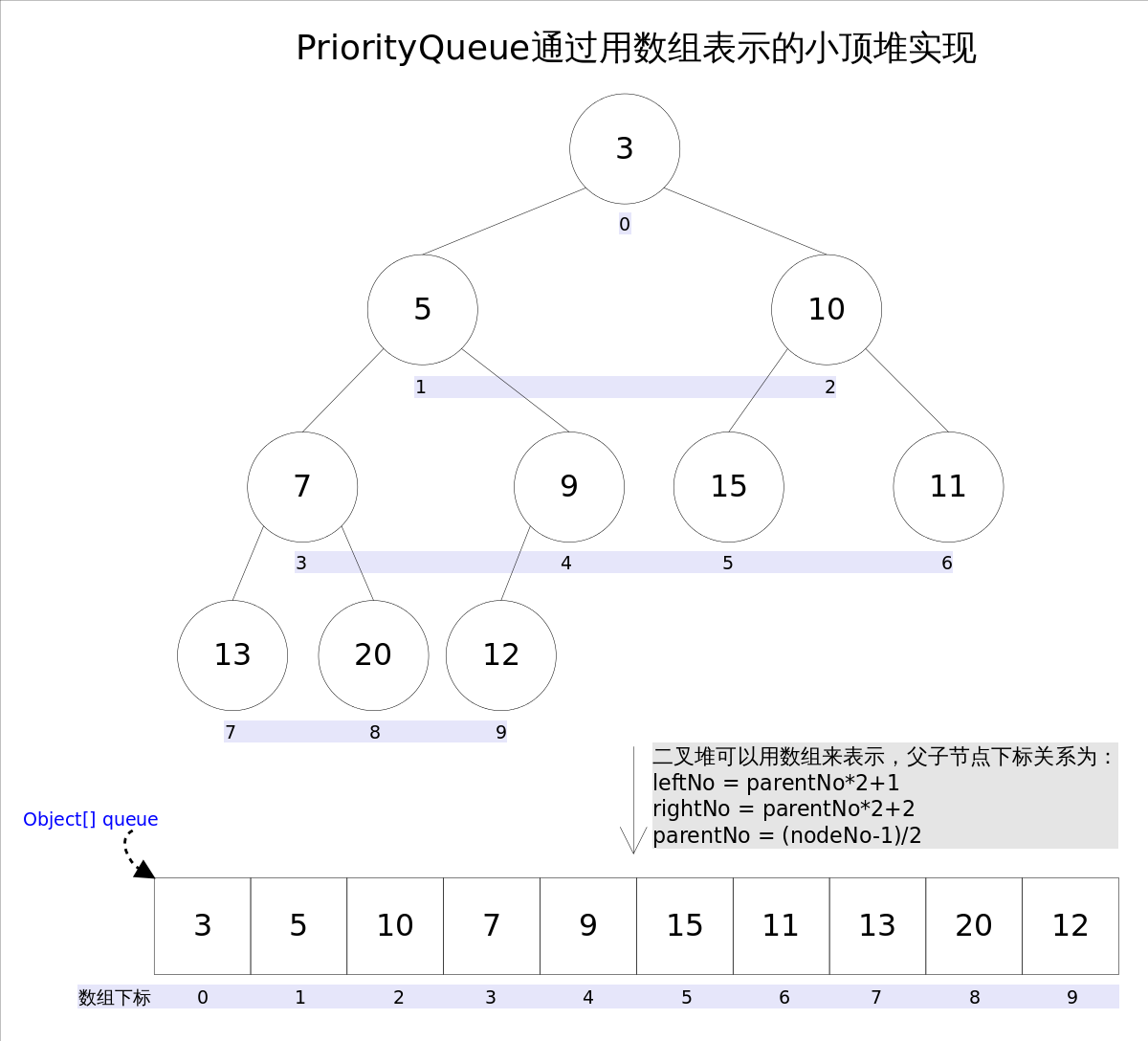
3.2 PriorityQueue（默认大小是11）

队列的特点是先进先出，但是优先队列有所不同，它不遵循先进先出的规则，而是根据队列中元素的优先权，优先权最大的先被取出。

PriorityQueue 的逻辑结构是二叉小顶堆（默认），具体说是通过完全二叉树实现的小顶堆，存储结构其实是一个数组。逻辑结构层次遍历的结果刚好是一个数组。

PriorityQueue 默认是一个小顶堆，然而可以通过传入自定义的 Comparator 函数来实现大顶堆。

当元素被添加到优先级队列中时，其容量会自动增长，此实现不同步。线程安全的是PriorityBlockingQueue类。



3.3 ConcurrentLinkedQueue

基于链接节点的无界并发deque(deque是双端队列) 。 并发插入，删除和访问操作可以跨多个线程安全执行。 A ConcurrentLinkedDeque是许多线程将共享对公共集合的访问的适当选择。像大多数其他并发集合实现一样，此类不允许使用null元素。

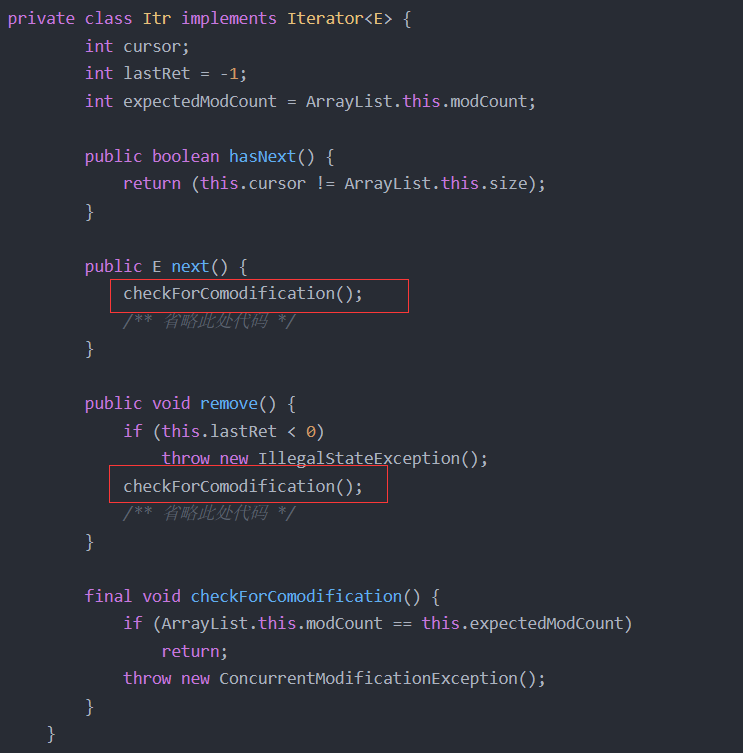
**4、Set**

Set集合的实现依赖于Map的实现，通过Map的 key值唯一性来实现。

**5、fail-fast机制**

fail-fast 是Java集合的一种错误检测机制，当多个线程对集合进行结构上的改变的时，有可能会产生fail-fast机制。

fail-fast产生的原因就在于程序在对 collection 进行迭代时，某个线程对该 collection 在结构上对其做了修改，这时迭代器就会抛出 ConcurrentModificationException 异常信息，从而产生 fail-fast。



迭代器在调用next()、remove()方法时都是调用checkForComodification()方法，该方法主要就是检测modCount （集合中定义）== expectedModCount （迭代器中定义）?是否成立，若不等则抛出ConcurrentModificationException 异常，从而产生fail-fast机制。

**三、关于“==”和“equals（）”和hashcode**

**5.4 int 和 Integer 区别**

（1）Integer是int的包装类，int则是java的一种基本数据类型。

（2）Integer变量必须实例化后才能使用，而int变量不需要 。

（3）Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向此对象；而int则是直接存储数据值。

（4）Integer的默认值是null，int的默认值是0。

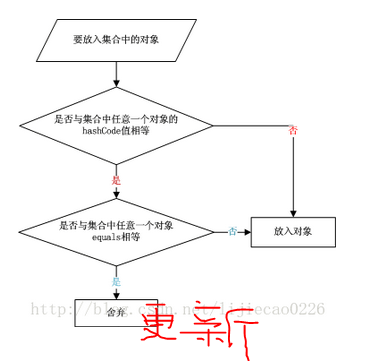
（5）对于两个非new生成的Integer对象，进行比较时，如果两个变量的值在区间-128到127之间，则比较结果为true，如果两个变量的值不在此区间，则比较结果为false。java对于-128到127之间的数，会进行缓存，Integer i = 127时，会将127进行缓存，下次再写Integer j = 127时，就会直接从缓存中取，就不会new了，所以地址是一样的。**但是超过这个范围，就是两个对象了，自然是false。**

**3.1 equals和"=="的比较**

1、基本型和基本型封装型进行“==”运算符的比较，基本型封装型将会自动拆箱变为基本型后再进行比较。 2.另外两个Integer对象进行“==”比较时，如果有一方的Integer对象是new获得的，返回false,因为比较的是两个对象的地址。 3、两个基本型的封装型进行equals()比较，首先equals()会比较类型，如果类型相同，则继续比较值，如果值也相同，返回true。 4、基本型封装类型调用equals()，但是参数是基本类型，这时候，先会进行自动装箱，基本型转换为其封装类型，若类型不同返回false,若装箱后类型相同，则比较值，如果值相同，则返回true，否则返回false。

5、hashcode和equals的区别和联系

对象放入散列集合的流程图：先进行hashCode值的比较，然后进行equals的比较。



**3.2 equals和hashcode的关系**

 1.hashCode是为了提高在散列结构存储中查找的效率，在线性表中没有作用。

       2.equals和hashCode需要同时覆盖（保证元素的唯一性）。

       3.若两个对象equals返回true，则hashCode有必要也返回相同的int数。

       4.若两个对象equals返回false，则hashCode不一定返回不同的int数,但为不相等的对象生成不同hashCode值可以提高哈希表的性能。

       5.若两个对象hashCode返回相同int数，则equals不一定返回true。

       6.若两个对象hashCode返回不同int数，则equals一定返回false。

       7.同一对象在执行期间若已经存储在集合中，则不能修改影响hashCode值的相关信息，否则会导致内存泄露问题。

**3.3 重写equals方法和hashcode方法**

**3.3.1 重写需要满足的原则：**

自反性：任何非null的引用值x，x.equals(x)必须返回true。

对称性：当y.equals(x)返回true时，x.equals(y)必须返回true。

传递性：x.equals(y)返回true，并且y.equals(z)返回true，那么x.equals(z)返回true。

一致性：equals的比较操作在对象中所用的信息没有被修改，多次调用x.equals(y)就会一致地返回相同的结果。

以下面这个类为例子：

class Coder {

private String name;

private int age;

// getters and setters

}

**3.3.2 equlas方法**的重写一般遵循三个步骤：

（1） 判断是否等于自身。

if(other == this)

return true;

（2）使用 instanceof 运算符判断 other 是否为Coder类型的对象（当前类类型的对象）。

if(!(other instanceof Coder))

return false;

（3）比较Coder类中自定义的数据域（name和age，一个都不能少）。

Coder o = (Coder)other;

return o.name.equals(name) && o.age == age;

**3.3.3 重写了equals()方法**，一定要记得**重写hashCode()方法**，hashCode()方法就是为哈希表服务的，要保证Coder对象中所有的成员属性都能在 hashCode 中得到体现。（17和31散列码）

@Override

public int hashCode() {

int result = 17;

result = result \* 31 + name.hashCode();

result = result \* 31 + age;

return result;

}

**四、Java语法糖**

语法糖是一种几乎每种语言或多或少都提供过的一些方便程序员开发代码的语法，它只是编译器实现的一些小把戏罢了，编译期间以特定的字节码或者特定的方式对这些语法做一些处理，开发者就可以直接方便地使用了。这些语法糖虽然不会提供实质性的功能改进，但是它们或能提高性能、或能提升语法的严谨性、或能减少编码出错的机会。

Java提供给了用户大量的语法糖，比如泛型、自动装箱、自动拆箱、foreach循环、变长参数、内部类、枚举类、断言（assert）等。

**4.1 泛型**

（1）Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，成功编译过后的class文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

（2）泛型仅仅是java的一颗语法糖，它不会影响java虚拟机生成的汇编代码，在编译阶段，虚拟机就会把泛型的类型擦除，还原成没有泛型的代码，顶多编译速度稍微慢一些，执行速度是完全没有什么区别的。

**4.2 可变参数**

Java可变长参数列表的实现是通过编译器把这些参数封装成一个数组来传递的。

可变长度参数必须作为方法参数列表中的的最后一个参数且方法参数列表中只能有一个可变长度参数。

**五、参数传递类型和取值范围**

**5.1 数据类型**

java中数据类型分为基本数据类型和引用数据类型：

* 基本数据类型
  + 整型：byte（1），short（2），int（4），long（8）
  + 浮点型：float（4），double（8）
  + 字符型：char（2）
  + 布尔型：boolean（只占用1个二进制位）
* 引用数据类型
  + 数组（在堆上分配空间）
  + 类
  + 接口

一般情况下，在数据做为参数传递的时候，基本数据类型是值传递，引用数据类型是引用传递（地址传递）。

String是一个类，类是引用数据类型，做为参数传递的时候，走的依然是引用传递，只不过String这个类比较特殊。

**String对象一旦创建，内容不可更改。每一次内容的更改都是重现创建出来的新对象。**

* 值传递的时候，将实参的值，copy一份给形参。
* 引用传递的时候，将实参的地址值，copy一份给形参。

也就是说，不管是值传递还是引用传递，形参拿到的仅仅是实参的副本，而不是实参本身。

**5.2 int型的取值范围**

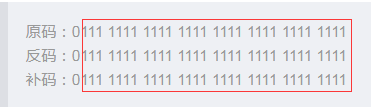
一个 int 类型占 4 字节，一个字节占 8 位二进制码，因此一个 int 总共占 32 位二进制码。去除第一位的符号位（正数为0，负数为1），剩下 31 位来表示数值。

在计算机中，数据是由二进制补码进行存储的，通过转换为原码获取它们的真实值。

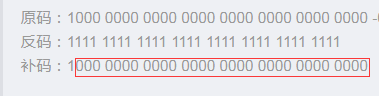
（1）当原码为正数的时候，正数的原码、反码、补码都相同。

（2）当原码为负数的时候，反码为去除符号位按位取反，补码为去除符号位按位取反再加1。

最大值：2^31 - 1



最小值：-2^31



**5.3 位、字节、字符的区别**

**位（bit）**：是计算机内部数据储存的最小单位。

**字节（byte）**：计算机信息技术用于计量存储容量的一种计量单位，表示数据量多少**。**计算机中 数据处理 的基本单位，习惯上用大写  B  来表示。

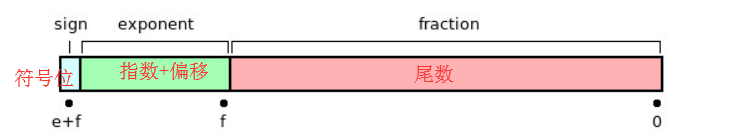
**字符**：是指计算机中使用的文字和符号，如字母、数字、汉字和符号。

**5.5 switch支持的数据类型**

byte、char、short、int、String、enum。

**5.6 小数在计算机中的表示**

计算机数字系统中的浮点数是对数学中的小数的近似（IEEE754标准），存储分为符号位、指数尾和位数部分（以32位float类型为例，分别占用1+8+23位）。先把浮点数用二进制的科学计数法表示，然后分别存储。



**六、代理**

**1. 静态代理（代理模式实现）**

静态代理通常用于对原有业务逻辑的扩充，其实也就是代理模式的一种实现，通过对真实对象的封装，来实现扩展性。

**优缺点：**

优点：扩展原功能，不侵入原代码。

缺点：同时代理多个方法时候冗余。

**2. 动态代理**

**2.1 定义**

动态代理指被代理者委托代理者完成相应的功能，是拦截器的一种实现方式，其用于拦截类或接口，内部可通过判断实现对某个方法的拦截。在程序运行时，运用反射机制动态创建而成。每一个动态代理类都必须要实现 InvocationHandler 这个接口。

**2.2 作用**

为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

**2.3 两种方式**

（1）**jdk动态代理**：

委托类必须实现接口，代理类只能对接口进行代理。使用java的反射机制，以及Proxy 类和 InvocationHandler 接口来提供动态生成代理类的能力。

每一个动态代理类都必须要实现 InvocationHandler 这个接口，并且每个代理类的实例都关联到了一个handler。当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的 invoke 方法来进行调用，invoke（）方法通过反射的方式去调用将来需要真正调用的方法。

（2）**cglib动态代理**：

代理类可以对类进行代理，使用第三方cglib库来实现，其内部使用 asm 框架生成代理类的字节码，其字节码文件更加复杂。

通过字节码技术为需要代理的目标对象创建一个子类对象，并在子类对象中拦截所有父类（即需要代理的类）方法的调用，然后在方法调用前后都可以加入自己想要执行的代码。但因为采用的是继承，所以不能对 final 修饰的类和 final 方法进行代理。

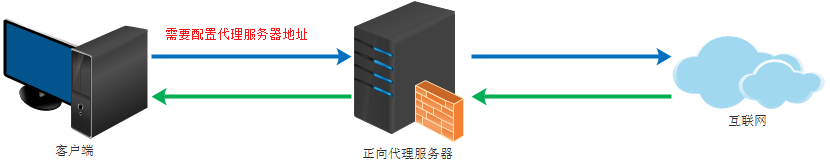
（3）两种方式的**比较**

jdk动态代理是由java内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。由此可以看出，jdk动态代理有一定的局限性，cglib这种第三方类库实现的动态代理应用更加广泛，且在效率上更有优势。

**3.正反向代理**

**3.1 正向代理**

（1）正向代理类似一个跳板机，代理访问外部资源。



（2）用途

　　（1）访问原来无法访问的资源，如google

      （2） 可以做缓存，加速访问资源

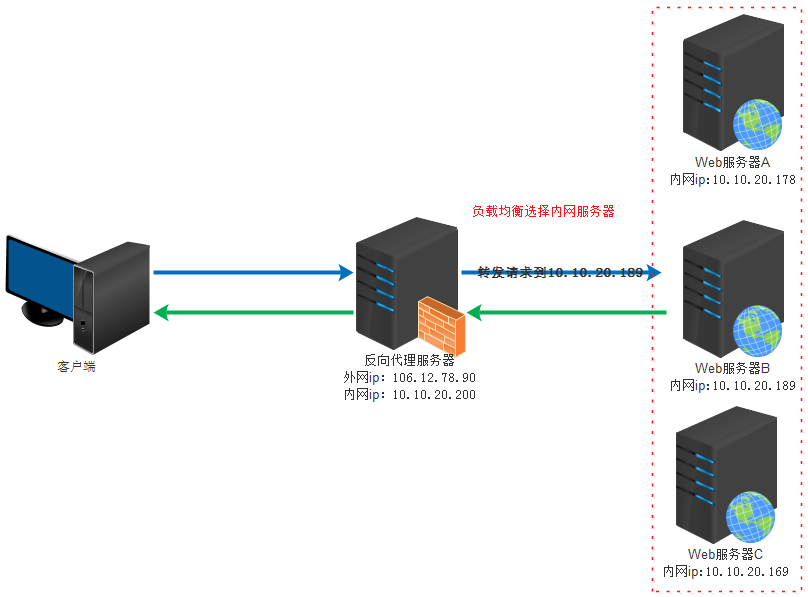
　　（3）对客户端访问授权，上网进行认证

　　（4）代理可以记录用户访问记录（上网行为管理），对外隐藏用户信息

**3.2 反向代理**

（1）客户端是无法感知代理的存在的，反向代理对外都是透明的，访问者者并不知道自己访问的是一个代理。因为客户端不需要任何配置就可以访问。

反向代理（Reverse Proxy）实际运行方式是指代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个服务器。



（2）作用

（1）保证内网的安全，可以使用反向代理提供WAF功能，阻止web攻击

大型网站，通常将反向代理作为公网访问地址，Web服务器是内网。

（2）负载均衡，通过反向代理服务器来优化网站的负载。

**七、反射**

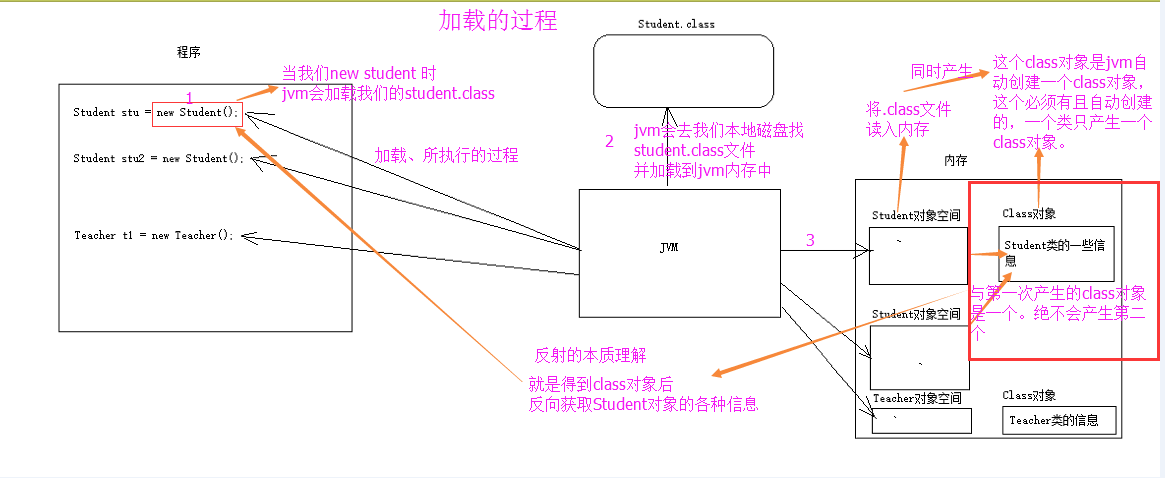
**1.定义**

反射是Java语言的一个特性，它允许程序**在运行时（注意不是编译的时候）来进行自我检查并且对内部的成员进行操作**。

动态获取信息以及动态调用对象的方法称为java语言的反射机制。JAVA反射机制是在运行状态中，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和属性。反射就是把java类中的各种成分映射成一个个的Java对象。

**2. 原理（得到class对象之后反向获取原始对象的各种信息，从类的加载开始说）**

Java中创建一个类，JVM就会创建一个该类对应的Class类的Class对象，该Class对象保存了该类相关的类型信息，保存到方法区中（HotSpot虚拟机）。



**3. 反射的作用**

（1）在运行时判断任意一个对象所属的类；

（2）在运行时获取类的对象；

（3）在运行时访问java对象的属性，方法，构造方法等。

**4. 获取 class 对象的三种方式**

（1）对象.getClass()

(2) 任何数据类型（包括基本数据类型）都有一个“静态”的class属性,需要导入类的包，依赖太强，不导包就抛编译错误。

(3) 通过Class类的静态方法：forName（String  className）(此字符串必须是真实路径，就是带包名的类路径，包名.类名  ，最常用的方法)。

**5. Class.forName 和 ClassLoader 区别**

java中class.forName()和classLoader都可用来对类进行加载。

（1）class.forName()除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块，static参数也会被初始化。Class.forName(name, initialize, loader)带参函数也可控制是否加载static块。

（2）而classLoader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容,只有在newInstance才会去执行static块。

**6. 反射的优缺点**

（1）优点

可以**实现动态创建对象和编译，体现出很大的灵活性。**通过反射机制我们可以获得类的各种内容，进行反编译。对于JAVA这种先编译再运行的语言来说，反射机制可以使代码更加灵活，更加容易实现面向对象。

（2）缺点

**对性能有影响**。使用反射基本上是一种解释操作，我们可以告诉JVM，我们希望做什么并且让它满足我们的要求。这类操作总是慢于直接执行相同的操作。

**7.Class类中的一些方法**

（1）Class.forName（）得到class对象。

（2）Constructor[ ] getConstructors() 得到当前类的 public 的构造方法

（3）Constructor[ ] getDeclaredConstructors（）当前类的所有的构造方法

（4）getFields() 得到当前类的公有字段

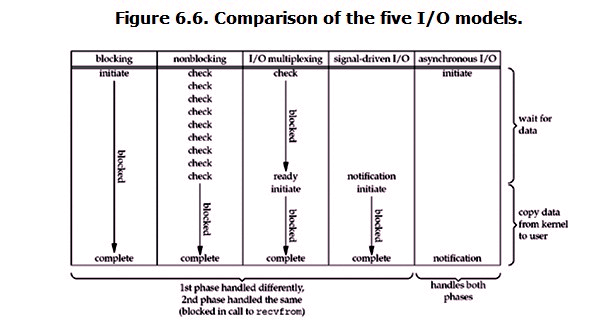
（5）getDeclaredFields() 得到当前类的所有字段（包括私有、受保护、默认的）

（6）getMethods() 得到公有的方法（包括继承的类或接口的方法）

（7）getDeclaredMethods() 得到当前类中所有的方法（不包括继承的方法）

**八、Java NIO**

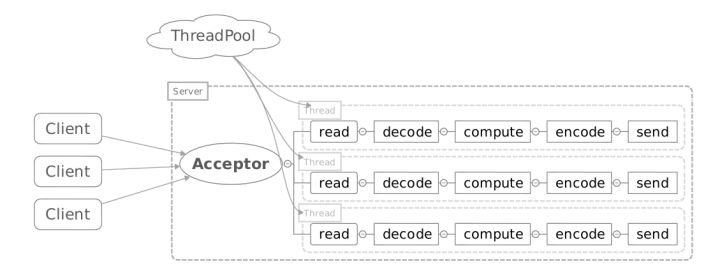
**1. IO的分类**



**阻塞IO**或者**同步IO**是指：IO的请求发出去之后，请求者一直在等待回复，当IO的数据到来之后，请求者就开始接受数据。阻塞的意思就是：IO请求发出去之后请求线程就停止在那里，一直等待数据到来。

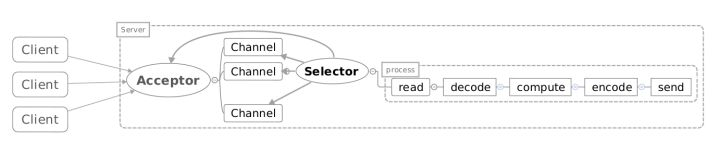
在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来和服务端进行通信，默认情况下服务端需要建立一堆线程等待请求，而客户端发送请求后，先询问服务端是否有线程响应，如果没有则会一直等待或者遭到拒绝请求。

一个连接就需要对应一个线程。



**非阻塞IO**：IO请求发出去之后，会立刻收到回复，这个回复可能是IO可以立即进行，或者是无法进行IO的错误。所以非阻塞IO请求者必须一直调用那个API，一直到API返回可以进行IO的信号。

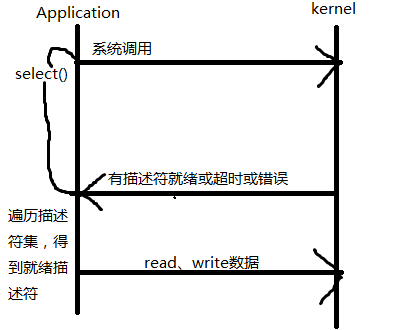
NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会通知应用程序进行处理，应用程序再将流读取到缓冲区或写入操作系统。 这个时候已经不是一个连接就要对应一个处理线程了，而是有效的请求对应一个线程，当连接没有数据时是没有工作线程来处理的。



**多路复用IO**：他的两个步骤处理是分开的，也就是说，

（1）select

select()机制中提供了一种fd\_set的数据结构，实际上是一long类型的[数组](http://baike.baidu.com/view/209670.htm)， 每一个数组元素都能与一打开的[文件句柄](http://baike.baidu.com/view/1706628.htm)（不管是Socket句柄，还是其他文件或[命名管道](http://baike.baidu.com/view/1197718.htm)或设备句柄）建立联系，建立联系的工作由程序员完成， 当调用select()时，由[内核](http://baike.baidu.com/view/1366.htm)根据IO状态修改fd\_set的内容，由此来通知执行了select()的进程哪一Socket或文件可读或可写。

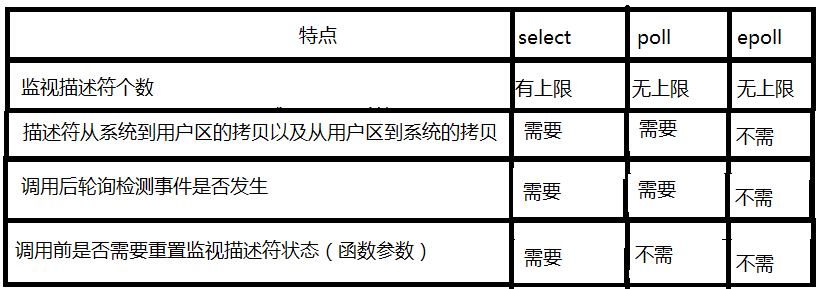


(2) poll

poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd\_set结构，其他的都差不多。

(3) epoll（红黑树）

epoll是linux内核为处理大批量文件描述符而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。epoll除了提供select/poll那种IO事件的水平触发（Level Triggered）外，还提供了边缘触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率。



**信号驱动IO**：这种IO模型主要用在嵌入式开发。

**异步IO**：IO请求发出之后，后台会建立一个进程，处理IO，当IO都处理完之后，把处理完的数据交给IO的请求者。（他的数据请求和数据处理都是异步的，数据请求一次返回一次，适用于长连接的业务场景）

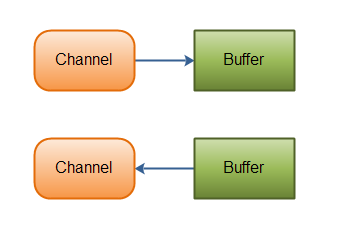
**2. NIO**

NIO即New IO，这个库是在JDK1.4中才引入的。它在标准java代码中提供了高速的面向块的IO操作。

Java NIO引入了选择器的概念，选择器可以监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道，这也是非阻塞IO的核心。而在标准IO的Socket编程中，单个线程则只能在一个端口监听。

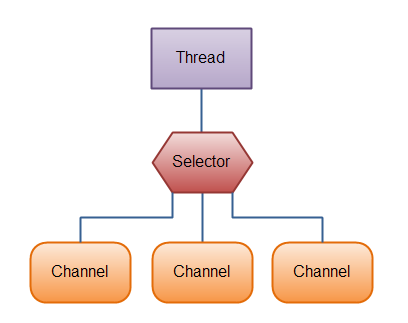
NIO的三大组成：Channel(通道)，Buffer(缓冲区), Selector。

（1）.**Channel**和IO中的Stream(流)是差不多一个等级的。只不过Stream是单向的，譬如：InputStream, OutputStream.而Channel是双向的，既可以用来进行读操作，又可以用来进行写操作。通道是一个对象，通过它可以读取和写入数据，将数据从通道读入缓冲区，再从缓冲区获取这个字节。



（2）**缓冲区**：在NIO库中，所有数据都是用缓冲区处理的。

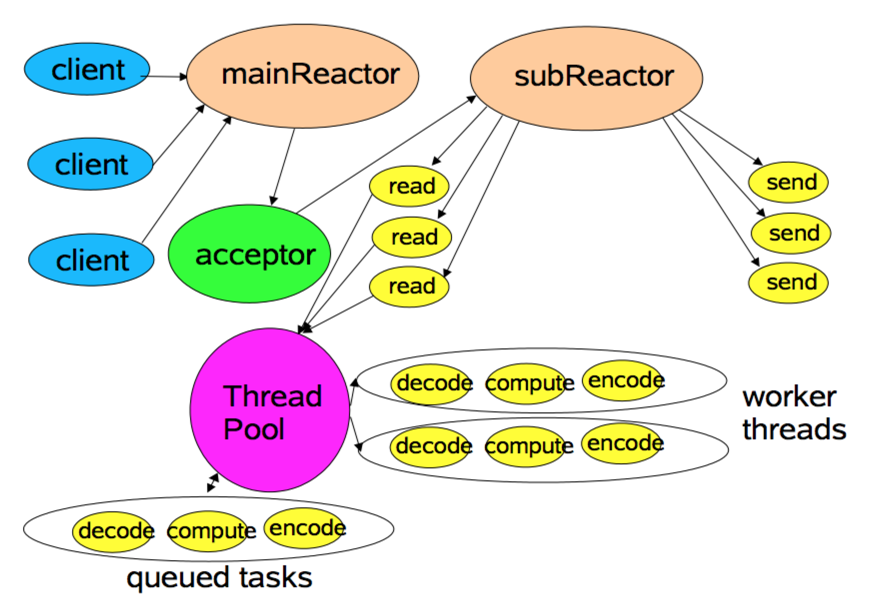
（3）**Selector**允许单线程处理多个 Channel。要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。能检测一个或多个通道 (channel) 上的事件，并将事件分发出去。



**3.Netty（NIO的最佳实践）**

**（1）定义：**Netty 是一个利用 Java 的高级网络的能力，隐藏其背后的复杂性而提供一个易于使用的 API 的客户端/服务器框架。

**（2）Reactor主从模型**



Reactor分成两部分，mainReactor负责监听server socket，accept新连接；并将建立的socket分派给subReactor。subReactor负责多路分离已连接的socket，读写网络数据，对业务处理功能交给worker线程池完成。Netty的线程模型是Reactor模型的变种，就是上述模型去掉线程池的变种，这也是Netty NIO的默认模式。

**（3）Netty的三大组件：**

Selector

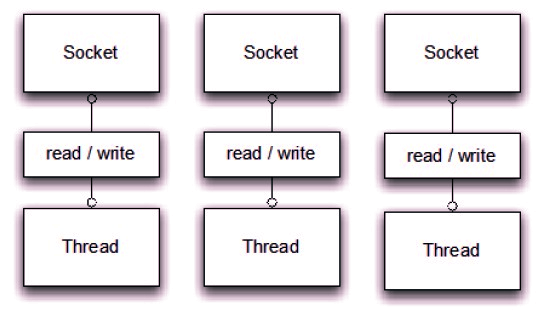
EventLoopGroup/EventLoop

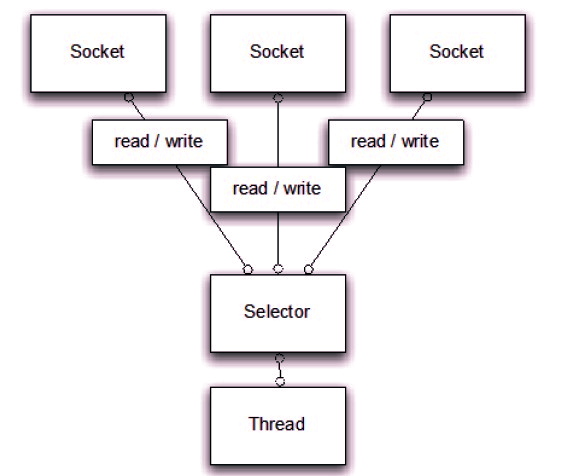
ChannelPipeline

Netty其实本质上就是Reactor模式的实现，Selector作为多路复用器，EventLoop作为转发器，Pipeline作为事件处理器。但是和一般的Reactor不同的是，Netty使用串行化实现，并在Pipeline中使用了责任链模式。

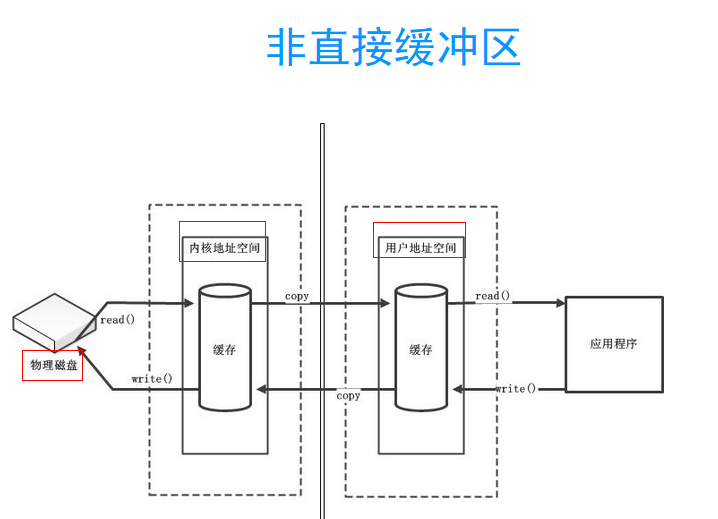
**（4）优点**

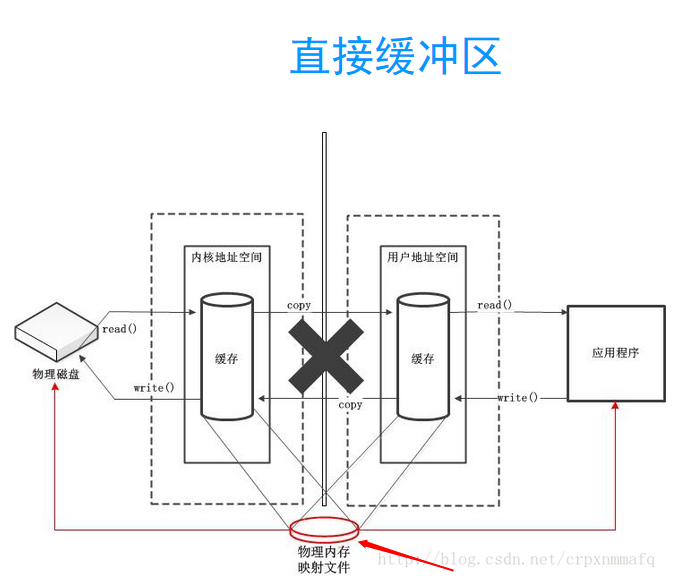
**高并发：**下面分别是阻塞IO和nio的通信方式





**传输快**：Netty的传输快其实也是依赖了NIO的一个特性——零拷贝。当他需要接收数据的时候，他会在**堆内存之外开辟一块内存**，不需要进行字节缓冲区的二次拷贝，数据就直接从IO读到了那块内存中去，在 netty 里面通过 ByteBuf 可以直接对这些数据进行直接操作，从而加快了传输速度。





**封装好**：

**九、克隆**

**1. Clone接口**

Cloneable接口是一个标记接口,也就是没有任何内容：

public interface Cloneable {

}

clone方法是在Object中定义的,而且是protected型的,只有实现了这个接口，才可以在该类的实例上调用clone方法,否则会抛出CloneNotSupportException。Object提供了一个对象拷贝的**默认方法clone**方法，但是该方法是有缺陷的，它提供了一种浅拷贝方式，也就是它并不会把对象所有属性全部拷贝一份，而是有选择性的拷贝，拷贝规则如下：

**（1）、基本类型**

如果变量是基本类型，则拷贝其值，比如：int、float、long等。

**（2）、String字符串**

这个比较特殊，拷贝的是地址，是个引用，但是在修改的时候，它会从字符串池（String Pool）中重新生成新的字符串，原有的字符串对象保持不变，此处可以认为String是个基本类型。

**（3）、对象**

如果变量是一个实例对象，则拷贝地址引用，也就是说此时新拷贝出的对象与原有对象共享该实例变量，不受访问权限的限制。这在Java中很疯狂，因为它突破了访问权限的定义，一个private修饰的变量，竟然可以被两个实例对象访问。

**2.浅拷贝（**内存泄露、相互影响**）**

在java中，对象创建后需要有一个引用变量来指向该对象实际的地址空间，也就是说引用变量与对象实体是两个不同的数据体。在Object类的clone()方法中，对对象字段进行复制时，如果字段是基本数据类型（如int、double等），则会复制字段的值到一个新的变量中，而字段是引用类型，则仅会将引用值复制给新对象中的相应字段中，也就是说，两个字段指向了同一个对象实例。

**3.深拷贝（**重新分配内存空间，值一致**）**

对于引用型变量，深拷贝会开辟一块新的内存空间，将被复制引用所指向的对象实例的各个属性复制到新的内存空间中，然后将新的引用指向块内存（也就是一个新的实例）。要想实现深拷贝的功能，我们在重写clone()方法的时候，就不能再简单地调用Object的本地clone()方法，而是要对clone（）方法进行重写。

**十、Java常见面试题目**

**1.final,finally,finalize的区别**

final 修饰符（关键字）如果一个类被申明为final,意味着它不能再派生出新的子类，因此一个类不能既被声明为abstract的，又被声明为final的。将变量和方法声明为final变量的，可以保证它们在使用中不被改变。对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象，但是它指向的对象的内容是可变的。被申明为final的方法也同样只能使用不能重写。

finally 在异常处理时提供finally块来执行任何清除操作。如果抛出一个异常，那么相匹配的 catch 子句就会执行，然后控制就会进入finally块（如果有的话）。

finalize 方法名。它是在object类中定义的，因此所有的类都继承了它。这个方法是由垃圾收集器在确定这个对象没有被引用的情况下对这个对象的调用。

**2.Java程序中如何检测死锁（可以使用jconsole工具）**

（1）输入 jps 命令，可以看到 java 的进程 id 列表。

（2）再输入 jstack + 进程ID，可以看到所有的线程栈。

（3）栈列表最下方，可以看到Found one Java-level deadlock既是发生死锁的线程。

**3.序列化和反序列化的原理**

**3.1 概念**

Java序列化是指把Java对象转换为字节序列的过程，而Java反序列化是指把字节序列恢复为Java对象的过程。

**3.2 为什么需要序列化与反序列化**

Java序列化的好处。其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。

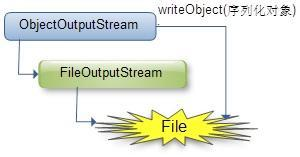
总的来说可以归结为以下几点：

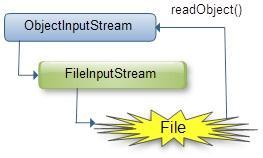
（1）永久性保存对象，保存对象的字节序列到本地文件或者数据库中；

（2）通过序列化以字节流的形式使对象在网络中进行传递和接收；

（3）通过序列化在进程间传递对象；

**3.3 原理**





只有实现了Serializable或Externalizable接口的类的对象才能被序列化，否则抛出异常。

（1）java.io.ObjectOutputStream：表示对象输出流；

它的writeObject(Object obj)方法可以对参数指定的obj对象进行序列化，把得到的字节序列写到一个目标输出流中；

（2）java.io.ObjectInputStream：表示对象输入流；

它的readObject()方法源输入流中读取字节序列，再把它们反序列化成为一个对象，并将其返回；

**4.Tomcat集群如何同步**

集群的具体同步机制，tomcat共提供了两种。一种是集群增量会话管理器，另一种是集群备份会话管理器。

**（1）集群增量会话管理器**

这是tomcat默认的集群会话管理器，它主要用于集群中各个节点之间会话状态的同步维护。这是一种全节点复制模式，全节点复制指的是集群中一个节点发生改变后会同步到其余全部节点。（那么非全节点复制，顾名思义，指的是集群中一个节点发生改变后，只同步到其余一个或部分节点。）

除了这一特点，集群增量会话管理器还具有只同步会话增量的特点，增量是以一个完整请求为周期，也就是说会在一个请求被响应之前同步到其余节点上。

**（2）集群备份会话管理器**

全节点复制模式存在的一个很大的问题就是用于备份的网络流量会随着节点数的增加而急速增加，这也就是无法构建较大规模集群的原因。为了解决这个问题，tomcat提出了集群备份会话管理器，每个会话只有一个备份，且备份与原件不会保存在同一个节点上。这样就可构建大规模的集群。

**5. Java后端技术**

服务框架：Dubbo，zookeeper，Rest服务

缓存：redis，memcache

消息中间件：ActiveMQ，kafka

负责均衡：Nginx

分布式文件：FastDFS

安全框架：Apache shiro

任务调度：quartz

持久层框架：mybatis

日志：log4j

项目基础搭建：spring，springmvc，

开发工具：eclipse、idea、svn、maven等

**服务器**：tomcat，jetty，Resin、**JBoss**、WebSphere、**WebLogic**等。

Web服务器是运行及发布Web应用的容器，只有将开发的Web项目放置到该容器中，才能使网络中的所有用户通过浏览器进行访问。同时还包括了Java应用服务器。

**（1）Tomcat 服务器**

　　目前最为流行的Tomcat服务器是Apache开源项目中的一个子项目，是一个小型、轻量级的支持JSP和Servlet 技术的Web服务器，也是初学者学习开发JSP应用的首选。

**（2）Jetty**

Jetty 目前的是一个比较被看好的 Servlet 引擎，它的架构比较简单，也是一个可扩展性和非常灵活的应用服务器。

虽然 Jetty 正常成长为一个优秀的 Servlet 引擎，但是目前的 Tomcat 的地位仍然难以撼动。相比较来看，它们都有各自的优点与缺点。

①从架构上来说，显然 Jetty 比 Tomcat 更加简单。

②Tomcat 在处理少数非常繁忙的连接（连接的生命周期短）上更有优势。而 Jetty 刚好相反，Jetty 可以同时处理大量连接而且可以长时间保持这些连接。

**（3）Resin 服务器**

　　Resin是一个非常流行的支持Servlet和JSP的服务器，速度非常快。Resin本身包含了一个支持HTML的Web服务器，这使它不仅可以显示动态内容，而且显示静态内容的能力也毫不逊色，因此许多网站都是使用Resin服务器构建。

**（4）JBoss服务器**

　　Jboss作为Java EE应用服务器，它不但是Servlet容器，而且是EJB（Enterprise [java bean](http://zhidao.baidu.com/search?word=java bean&fr=qb_search_exp&ie=utf8)）容器，速度慢一些，不适合开发阶段，可以用于真实运行环境（免费）。从而受到企业级开发人员的欢迎，从而弥补了Tomcat只是一个Servlet容器的缺憾。

**（5）WebSphere 服务器**

　　WebSphere是IBM公司的产品，可进一步细分为 WebSphere Performance Pack、Cache Manager 和WebSphere Application Server等系列，其中WebSphere Application Server 是基于Java 的应用环境，可以运行于 Sun Solaris、Windows NT 等多种操作系统平台，用于建立、部署和管理Internet和Intranet Web应用程序。

**（6）WebLogic 服务器**

　　WebLogic 可进一步细分为 WebLogic Server、WebLogic Enterprise 和 WebLogic Portal 等系列，其中 WebLogic Server 的功能特别强大。WebLogic 支持企业级的、多层次的和完全分布式的Web应用，并且服务器的配置简单、界面友好。对于那些正在寻求能够提供Java平台所拥有的一切应用服务器的用户来说，WebLogic是一个十分理想的选择。不适合开发阶段，太慢了，适合于运行环境（收费）。

**apache 和 nginx的比较：**

|  |  |
| --- | --- |
| **Apache** | **Nginx** |
| 发展到现在，模块超多,成熟稳定 | 设计高度模块化，编写模块相对简单，轻量级，占用更少的内存和资源 |
| apache 则是阻塞型的,容易出现进程数飙升，从而拒绝服务的现象 | 处理请求是异步非阻塞的，负载能力比 apache 高很多 |
| 处理动态请求有优势 | 处理静态网页上表现的更好（简单、占资源少） |
| 只是web容器 | 作为负载均衡服务器，支持 7 层负载均衡 |

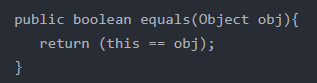
**6. Object类**

Java.lang.Object 类是所有Java类的最高层次父类，该类中没有定义任何属性，只有几个方法。

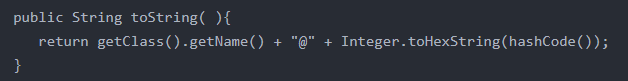
**（1）hashCode( )方法**

   public int hashCode ( ) 返回当前对象的哈希值(哈希码可理解为系统为每一个Java对象自动创建的整型编号，而且此编号唯一)，默认实现是将该对象的内部地址转换成一个整数返回。

**（2）equals( )方法**



**（3）toString( )方法**



**（4）finalize( )方法**

 Java运行环境中的垃圾收集器在销毁一个对象之前，会自动判断该对象是否有必要执行 finalize() 方法（当对象没有覆盖finalize（）方法或者方法已经被虚拟机执行过则不需要执行），且最多执行一次。如果判定一个对象有必要执行 finalize（）方法，那么则会把这个对象放置在F-Queue队列中，稍后 GC 会对 F-Queue 中的对象进行第二次小规模的标记，如果对象要完成自我救赎只需要与引用链上的对象建立关联即可。

**（5）clone( )方法**

拷贝一个现有的对象，得到一个新对象(和现有对象拥有完全相同的属性值信息)，之后二者使用互不干预。

**（6）wait( )方法和notify( )方法**

主要用于多线程之间的同步。

**7. Maven**

**7.1 Maven 依赖原则**

（1）路径最短优先原则

（2）pom文件中申明顺序优先

（3）覆写优先（子pom内声明的优先于父pom中的依赖）

**7.2 解决 jar 冲突**

遇到冲突的时候第一步要找到maven加载的是什么版本的jar包，通过们mvn dependency:tree查看依赖树，通过maven的依赖原则来调整依赖在pom文件的申明顺序。

**8. 接口和抽象类的区别**

8.1 **语法层面上的区别**

1）抽象类可以含有非抽象方法，而接口中只能存在public abstract 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，接口中不能含有静态代码块以及静态方法；

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。

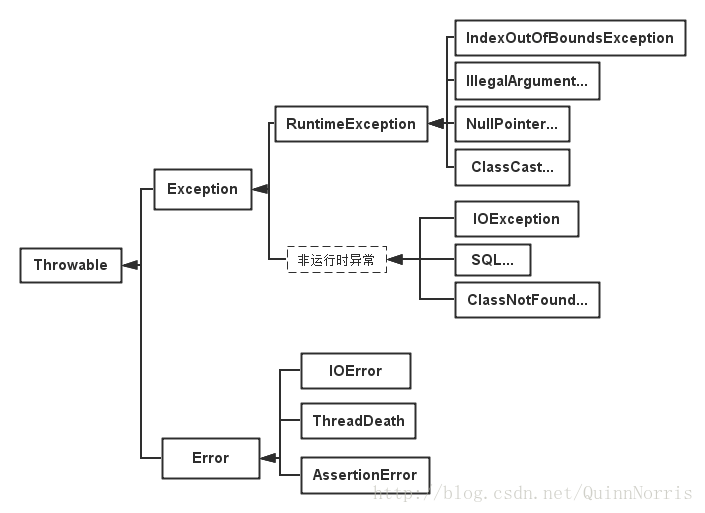
8.2 **设计层面上的区别**

1）抽象类体现的是一种继承关系，父类和子类在概念上是一致的，是“is-a”的关系；接口并不要求实现者和接口在概念上是一致的，仅仅是实现了接口的契约而已，是“like-a”的关系。

2）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。

3）抽象类的设计目的，是**代码复用**。比如男人、女人可以抽象出人的公有属性和特征；接口是对类的**行为进行约束**，约束了行为的有无，但是不对行为的实现进行限制，比如吃饭这个动作，但是可以站着吃，坐着吃等。

**9. Java 异常机制**



在java中，所有的异常都有一个共同的祖先 Throwable类。Throwable类有两个子类，一个是Error类，一个是Exception。在java中能通过代码处理的我们叫做异常，而我们不能处理的才叫做错误。错误指的是：JVM运行错误，栈空间用尽，类定义错误等等非常严重的问题。Exception主要负责处理类似：空指针、除数为零、数组越界这类问题。

**（1）RunTimeException表示那些逻辑性错误，可以避免。**

NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)。在我们编写代码的时候，我们从来不会希望编写出有空指针或者下标越界的代码，因为这是错误的，在java中，这些逻辑上的，因为我们大意而造成的错误都叫做运行时错误。

六个常见的运行时异常：

**NullPointerException(空指针异常)**

**IndexOutOfBoundsException(数组越界异常)，ArrayIndexOutOfBoundsException**

**ClassCastException(类转换异常)**

**NumberFormatException 数字格式异常**

**IllegalArgumentException 传递非法参数异常**

**ArrayStoreException(数据存储异常，操作数组时类型不一致)**

**（2）非运行时异常表示有可能发生的异常，我们需要声明。**

为了防止代码在运行时出现问题，java强制规定：非运行时异常必须被处理。

运行异常是程序逻辑错误，无法预料，改正就可以，无需抛出或处理。非运行时异常是显然可能存在的错误，我们被强制必须抛出或处理来保证程序的安全性。

**10. comparable接口和comparator的区别**

①、Comparable位于包java.lang下，而Comparator位于包java.util下。

②、Comparable接口将比较代码（重写compareTo(T t)方法）嵌入需要进行比较的类的自身代码中，而Comparator接口在一个独立的类中实现比较（重写compare(T t1, T t2)方法）。

③、Comparable接口强行对实现它的每个类的对象进行整体排序（自然排序)，而Comparator接口不强制进行自然排序，可以指定排序顺序。

④、如果前期类的设计没有考虑到类的Compare问题而没有实现Comparable接口，后期可以通过Comparator接口来实现比较算法进行排序。

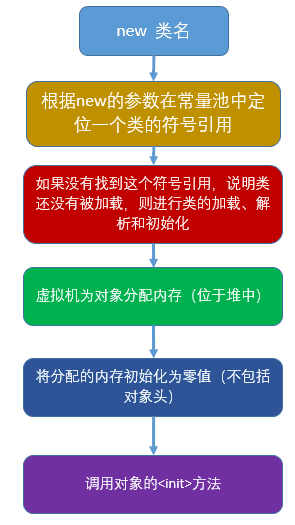
其中Comparator接口相当于策略模式的抽象接口，具体的比较器实现类是具体的策略实现，集合操作类相当于Context ，在Context中使用具体策略进行大小比较。

**11. Arrays.sort() 的实现**

基本数据类型数组的操作，使用经过优化的快速排序算法，当数组的规模较小时（jdk1.8中小于47）使用直接插入排序。

引用数据类型数组的排序，在jdk1.7之前使用经过优化的归并排序算法，当数组规模较小时，使用直接插入排序。现在使用的是TimSort算法，就是找到已经排好序数据的子序列，然后对剩余部分排序，然后合并起来。

**12. Java对象的创建步骤**



**13. 无限循环 for(;;) 与 while（true） 的区别**

在Java中都是优化成 goto 指令没区别；对于早期的C语言，两种写法性能会不一样，for语句编译器会优化成一条汇编指令，而while判断则编译器会生成好几条汇编指令。

**14. 应用程序变慢如何定位？**

第一步：登录后台服务器，查看**系统资源**是否达到上限，例如：CPU、内存、磁盘、I/O、网络带宽等，如果是这些问题，先将这些问题逐一解决：

      如果是CPU的问题，则需要查看一下CPU占比比较高的进程，然后使用jstack命令生成进程的堆栈信息，看是否发生频繁Full GC，如果是的话，还需要看一下内存快照，分析一下内存情况（可以使用java自带的或第三方工具）；如果是磁盘空间满了，及时清理磁盘；

第二步：检查**应用服务器**（Jboss/Tomcat）的线程池配置是否合理，看一下请求的排队现象是否严重，如果严重则需要重新设置合理的线程池。同样，检查一下数据库的连接池设置是否合理，增大连接池设置，同时检查一下是否有慢sql，如果有慢sql，则进行优化（优化方案是查看执行计划，设置合理的索引等）。

第三步：查看访问慢的**服务的调用链**，查看一下调用链中的每一步响应时间是否合理，如果不合理，则联系相关系统的负责人进行排查和解决。

第四步：检查web服务器的**请求日志**，看一下是否存在Doss攻击，如果有Doss攻击，则将攻击者的IP添加到防火墙的黑名单里。

**十一、java8的新特性**

**1、接口中的默认方法和静态方法**

默认方法就像一个普通Java方法，只是方法用default关键字修饰，其目的是为了解决接口的修改与已有的实现不兼容的问题。

静态方法就像一个普通Java静态方法，但方法的权限修饰只能是public或者不写。默认方法和静态方法使Java的功能更加丰富。

**2、函数式接口和Lambda表达式**

函数式接口是为Java 8中的lambda而设计的，lambda表达式的方法体其实就是函数接口的实现。“lambda表达式”是一段可以传递的代码，因为他可以被执行一次或多次。

**3、Stream API**

一个Stream表面上与一个集合很类似，允许你改变和获取数据，但实际上却有很大区别：

（1）Stream自己不会存储元素。元素可能被存储在底层的集合中，或者根据需要产生出来。

（2）Stream操作符不会改变源对象。相反，他们返回一个持有新结果的Stream。

（3）Stream操作符可能是延迟执行的。意思是它们会等到需要结果的时候才执行。

Stream相对于循环操作有更好的可读性。并且可以并行计算，Stream遵循“做什么，而不是怎么去做”的原则。只需要描述需要做什么，而不用考虑程序是怎样实现的。

**4、新的日期和时间 API**

**5、杂项改进**

Java8在String类中只添加了一个新方法，就是join，该方法实现了字符串的拼接，可以把它看作split方法的逆操作。

Java8为使用流读取文件行及访问目录项提供了一些简便的方法（Files.lines和Files.list）。