# [Java中List和ArrayList的区别](https://www.cnblogs.com/zcscnn/p/7743507.html)

第一次看这篇文章时，是在CSDN博客中看到的，作者写的初衷还是蛮好的，但是确实有错误的地方和不是很明白的地方。于是就很想去看看原文，废了半天的功夫终于找到了，原文还是一样有出错和我不理解的地方，我也把原文的地址贴在上面了。三月份学习的Java集合框架这部分，这几天拿出来整理就想再深入一点，因此也看了很多的关于Java集合框架的文章，这篇我算是有一点点的体会，我只是改了一些我认为错误的地方和不通顺的地方。

       改过的原文：

       List是一个接口，而ArrayList是List接口的一个实现类。

       ArrayList类继承并实现了List接口。

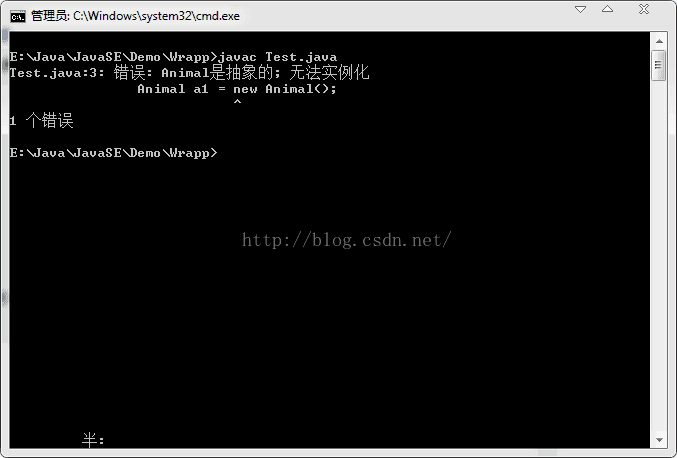
       因此，List接口不能被构造，也就是我们说的不能创建实例对象，但是我们可以像下面那样为List接口创建一个指向自己的对象引用，而ArrayList实现类的实例对象就在这充当了这个指向List接口的对象引用。

       个人见解：

       要是你已经学过了OOP，上面的部分是不难理解的，这是面向对象重要的知识点，面向对象最重要的就是多态，我们都知道接口和抽象不能被实例化，但是它们可以创建一个指向自己的对象引用，它们的实现类或子类就在充当这样的角色，我想这就是面向对象编程中多态的优势。前些日子在学习UML建模语言和Java设计模式的时候，深深地的体会到了面向对象编程的好处，Java集合框架中用到也不足为奇，Java本身就是面向对象的编程语言。

       上面的理解可能有点难度，但是我们找一个具体的实例，就会理解起来比较容易。我们定义一个动物的抽象类Animal，再定义一个继承自Animal基类的Dog类，看下面的代码就会理解抽象类和接口不能被实例化：

1. **public static void main(String[] args){**
2. Animal a1 = **new Animal();//编译出错**
3. Animal a2 = **new Dog();**
4. }
5. }
7. **abstract class Animal{**
8. //动物名字
9. String name;
11. //动物叫声
12. **public void shout(){**
13. System.out.println("叫声...");
14. }
15. }
17. **class Dog extends Animal{**
18. //狗类独有的方法
19. **public void guard(){**
20. System.out.println("狗有看门的独特本领！");
21. }
22. }



List list;//正确，list = null;

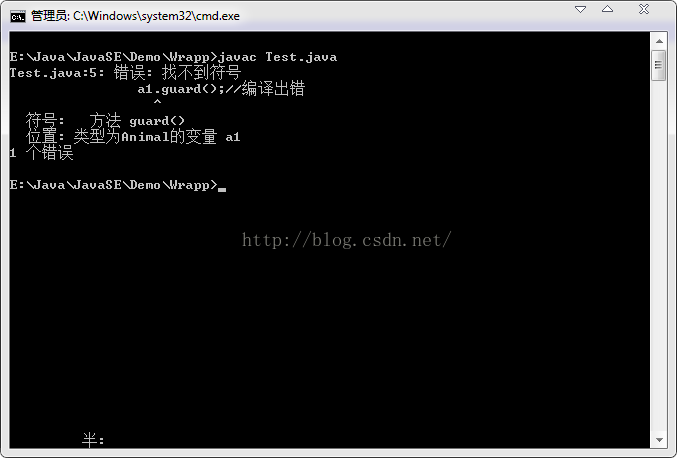
List list = new List();//是错误的用法

List list = new ArrayList();这句创建了一个ArrayList实现类的对象后把它上溯到了List接口。此时它就是一个List对象了，它有些ArrayList类具有的，但是List接口没有的属性和方法，它就不能再用了。 而ArrayList list=newArrayList();创建一对象则保留了ArrayList的所有属性和方法。

       个人见解

       我们继续上面的例子，如果我们创建的是抽象类的对象引用，那么这个对象只能调用自己的非抽象方法，下面的是shout()方法，不能调用继承它的子类的独有的方法，在下面的就是guard()方法不能被a1调用，继续测试代码：

1. **public static void main(String[] args){**
2. Animal a1 = **new Dog();**
3. a1.shout();//编译通过
4. //a1.guard();//编译出错
5. }
6. }
8. **abstract class Animal{**
9. //动物名字
10. String name;
12. //动物叫声
13. **public void shout(){**
14. System.out.println("叫声...");
15. }
16. }
18. **class Dog extends Animal{**
19. //狗类独有的方法
20. **public void guard(){**
21. System.out.println("狗有看门的独特本领！");
22. }
23. }



       如果我们采用Dog d1 = new Dog();那么d1可以调用抽象类和子类的所有属性和方法，这里不再测试。

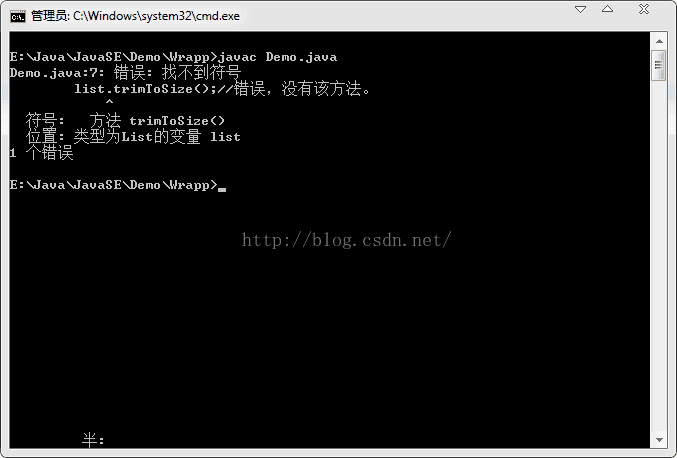
       这是一个例子：

1. **public class Demo{**
2. **public static void main(String[] args){**
3. List list = **new ArrayList();**
4. ArrayList arrayList = **new ArrayList();**
5. list.trimToSize();//错误，没有该方法。
6. arrayList.trimToSize();//ArrayList里有该方法。
7. }
8. }

       编译一下就知道结果了。

       个人见解

       我刚在前面的文章里面把Java API中List接口和ArrayList实现类的方法都列出来了，可以看到List接口中并没有trimToSize()方法，但这个方法在它的实现类ArrayList中有。因此编译的结果为：



如果是下面这个样子的：

        List a=new ArrayList();

        则a拥有List的所有属性和方法，不会拥有其实现类ArrayList的独有的属性和方法。

        如果List与ArrayList中有相同的属性(如int i),有相同的方法(如void f()),

        则a.i是调用了List中的i

        a.f()是调用了ArrayList中的f();

----------------------------------------------------------------------------------------------------------------

        问题的关键:

        为什么要用 List list = new ArrayList() ,而不用 ArrayList alist = new ArrayList()呢？

        问题就在于List接口有多个实现类，现在你用的是ArrayList，也许哪一天你需要换成其它的实现类，如 LinkedList或者Vector等等，这时你只要改变这一行就行了： List list = new LinkedList(); 其它使用了list地方的代码根本不需要改动。

       假设你开始用ArrayList alist = new ArrayList(), 这下你有的改了，特别是如果你使用了ArrayList实现类特有的方法和属性。

       个人见解

       上面的说明，我是在看了设计模式才恍然大悟的，设计模式的原则和理念果然是强大的，只有好好学习了面向对象原则和设计模式，那么理解上面的就不再是难度，这样的好处是为了代码的可维护性，可复用性，可扩展性以及灵活性，再者就是这符合了里氏代换原则和开闭原则。看来学习设计模式好处是大大的。

       地区用List arr = new ArrayList();定义;行业用ArrayList arr = new ArrayList();定义;则说明,行业里用到了ArrayList的特殊的方法.

       另外的例子就是,在类的方法中,如下声明:

       private void doMyAction(List list){}

       这样这个方法能处理所有实现了List接口的类,一定程度上实现了泛型函数.

       如果开发的时候觉得ArrayList,HashMap的性能不能满足你的需要,可以通过实现List,Map(或者Collection)来定制你的自定义类.

        个人见解

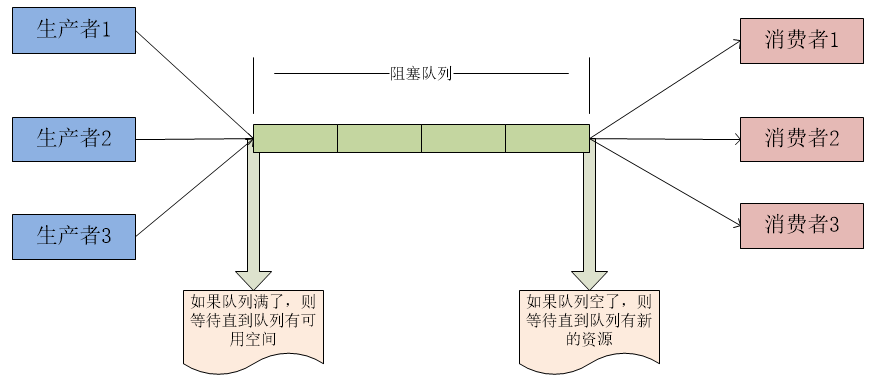
        正因为List是接口，所以它的扩展性是良好的，这是面向对象编程最大的改变，也是它的核心，在这里我是体会到了一句话，就是Java集合框架的学习最难体现你学习Java语言的程序都多深，看来我以前学习的程度只是停留在入门级别了，这几天可要好好再温习重新认识一番Java集合框架了。

        可能理解不是很深，对于原文的理解也就这些了，不对的地方自己会及时更正。

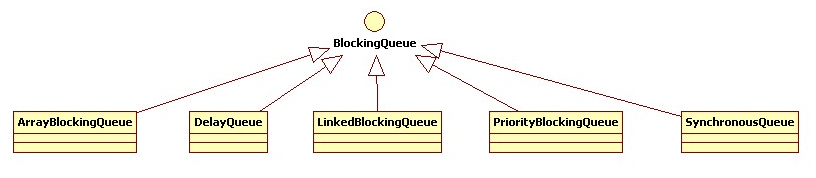
# BlockingQueue是什么

1. 前言

BlockingQueue即阻塞队列，它算是一种将ReentrantLock用得非常精彩的一种表现，依据它的基本原理，我们可以实现Web中的长连接聊天功能，当然其最常用的还是用于实现生产者与消费者模式，大致如下图所示：



在Java中，BlockingQueue是一个接口，它的实现类有ArrayBlockingQueue、DelayQueue、 LinkedBlockingDeque、LinkedBlockingQueue、PriorityBlockingQueue、SynchronousQueue等，它们的区别主要体现在存储结构上或对元素操作上的不同，但是对于take与put操作的原理，却是类似的。下面的源码以ArrayBlockingQueue为例。



2. 分析

BlockingQueue内部有一个ReentrantLock，其生成了两个Condition，在ArrayBlockingQueue的属性声明中可以看见：

/\*\* Main lock guarding all access \*/

final ReentrantLock lock;

/\*\* Condition for waiting takes \*/

private final Condition notEmpty;

/\*\* Condition for waiting puts \*/

private final Condition notFull;

...

public ArrayBlockingQueue(int capacity, boolean fair) {

if (capacity <= 0)

throw new IllegalArgumentException();

this.items = new Object[capacity];

lock = new ReentrantLock(fair);

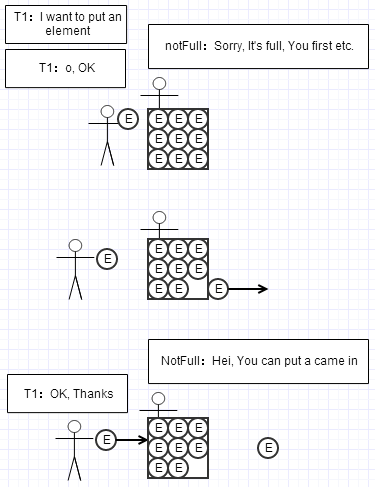
notEmpty = lock.newCondition();

notFull = lock.newCondition();

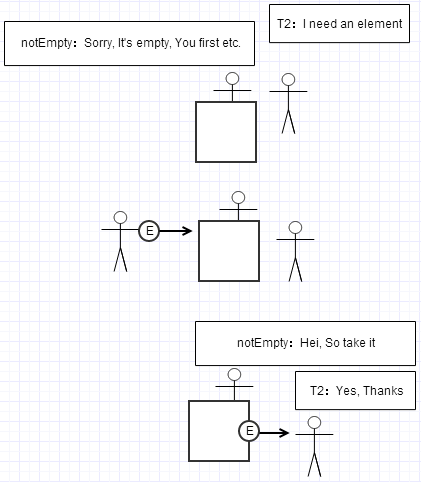
}

而如果能把notEmpty、notFull、put线程、take线程拟人的话，那么我想put与take操作可能会是下面这种流程：

put(e)



take()



其中ArrayBlockingQueue.put(E e)源码如下（其中中文注释为自定义注释，下同）：

/\*\*

\* Inserts the specified element at the tail of this queue, waiting

\* for space to become available if the queue is full.

\*

\* @throws InterruptedException {@inheritDoc}

\* @throws NullPointerException {@inheritDoc}

\*/

public void put(E e) throws InterruptedException {

checkNotNull(e);

final ReentrantLock lock = this.lock;

lock.lockInterruptibly();

try {

while (count == items.length)

notFull.await(); // 如果队列已满，则等待

insert(e);

} finally {

lock.unlock();

}

}

/\*\*

\* Inserts element at current put position, advances, and signals.

\* Call only when holding lock.

\*/

private void insert(E x) {

items[putIndex] = x;

putIndex = inc(putIndex);

++count;

notEmpty.signal(); // 有新的元素被插入，通知等待中的取走元素线程

}

ArrayBlockingQueue.take()源码如下：

public E take() throws InterruptedException {

final ReentrantLock lock = this.lock;

lock.lockInterruptibly();

try {

while (count == 0)

notEmpty.await(); // 如果队列为空，则等待

return extract();

} finally {

lock.unlock();

}

}

/\*\*

\* Extracts element at current take position, advances, and signals.

\* Call only when holding lock.

\*/

private E extract() {

final Object[] items = this.items;

E x = this.<E>cast(items[takeIndex]);

items[takeIndex] = null;

takeIndex = inc(takeIndex);

--count;

notFull.signal(); // 有新的元素被取走，通知等待中的插入元素线程

return x;

}

可以看见，put(E)与take()是同步的，在put操作中，当队列满了，会阻塞put操作，直到队列中有空闲的位置。而在take操作中，当队列为空时，会阻塞take操作，直到队列中有新的元素。

而这里使用两个Condition，则可以避免调用signal()时，会唤醒相同的put或take操作。

以上。

# ConcurrentLinkedQueue、ArrayBlockingQueue、LinkedBlockingQueue 区别及使用场景

三者区别与联系：

联系，三者 都是线程安全的。区别，就是 并发 和 阻塞，前者为并发队列，因为采用cas算法，所以能够高并发的处理；后2者采用锁机制，所以是阻塞的。注意点就是前者由于采用cas算法，虽然能高并发，但cas的特点造成操作的危险性，怎么危险性可以去查一下cas算法（但一些多消费性的队列还是用的它，原因看下边使用场景中的说明）

后2者区别：

联系，第2和第3都是阻塞队列，都是采用锁，都有阻塞容器Condition，通过Condition阻塞容量为空时的取操作和容量满时的写操作第。区别，第2就一个整锁，第3是2个锁，所以第2第3的锁机制不一样，第3比第2吞吐量 大，并发性能也比第2高。

后2者的具体信息: LinkedBlockingQueue是BlockingQueue的一种使用Link List的实现，它对头和尾（取和添加操作）采用两把不同的锁，相对于ArrayBlockingQueue提高了吞吐量。它也是一种阻塞型的容器，适合于实现“消费者生产者”模式。

ArrayBlockingQueue是对BlockingQueue的一个数组实现，它使用一把全局的锁并行对queue的读写操作，同时使用两个Condition阻塞容量为空时的取操作和容量满时的写操作。

正因为LinkedBlockingQueue使用两个独立的锁控制数据同步，所以可以使存取两种操作并行执行，从而提高并发效率。而ArrayBlockingQueue使用一把锁，造成在存取两种操作争抢一把锁，而使得性能相对低下。LinkedBlockingQueue可以不设置队列容量，默认为Integer.MAX\_VALUE.其容易造成内存溢出，一般要设置其值。

使用场景：

适用阻塞队列的好处：多线程操作共同的队列时不需要额外的同步，另外就是队列会自动平衡负载，即那边（生产与消费两边）处理快了就会被阻塞掉，从而减少两边的处理速度差距，自动平衡负载这个特性就造成它能被用于多生产者队列，因为你生成多了（队列满了）你就要阻塞等着，直到消费者消费使队列不满你才可以继续生产。 当许多线程共享访问一个公共 collection 时，ConcurrentLinkedQueue 是一个恰当的选择。   
LinkedBlockingQueue 多用于任务队列（单线程发布任务，任务满了就停止等待阻塞，当任务被完成消费少了又开始负载 发布任务）   
ConcurrentLinkedQueue 多用于消息队列（多个线程发送消息，先随便发来，不计并发的-cas特点）

多个生产者，对于LBQ性能还算可以接受；但是多个消费者就不行了mainLoop需要一个timeout的机制，否则空转，cpu会飙升的。LBQ正好提供了timeout的接口，更方便使用 如果CLQ，那么我需要收到处理sleep

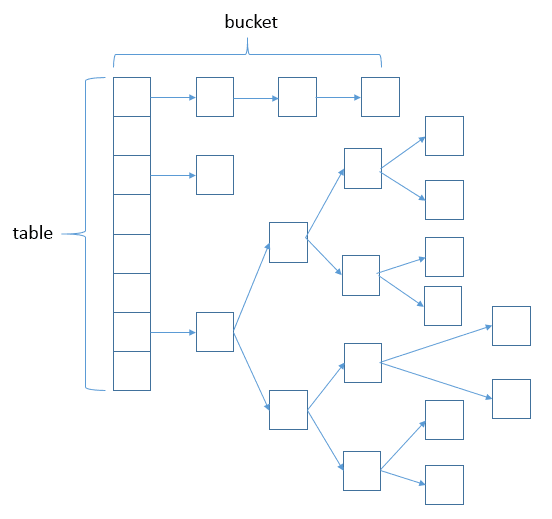
总结

单生产者，单消费者 用 LinkedBlockingqueue   
多生产者，单消费者 用 LinkedBlockingqueue   
单生产者 ，多消费者 用 ConcurrentLinkedQueue   
多生产者 ，多消费者 用 ConcurrentLinkedQueue

对上边总结：   
如消息队列，好多client发来消息，根据client发送先后放入队列中，先发送的就先放进来，然后由于队列是先进先出，是一个一个出来的，所以不涉及到线程安全问题，所以用LinkedBlockingqueue 队列。比如还拿上边消息队列那个例子，由于队列是一个一个出来的，出来一个消息协议体就由线程池分配一个线程去处理这个消息体，这个消息体对于线程池来说谈不上共享不共享的问题，即不会多个线程去抢同一个消息体去执行，所以就不需要用线程安全的队列结构了；那假如一种情况，队列里仍然是一个一个的出来，但是出来的这个元素是 线程池共享的，即大家线程都需要用到这个从队列里出来的这个元素，也就是多消费者消费同一个东西这种情况，所以就要用线程安全的队列了，即ConcurrentLinkedQueue。

# **HashMap 的数据结构？**

**哈希表结构（链表散列：数组+链表）**实现，结合数组和链表的优点。当链表长度超过 **8** 时，链表转换为**红黑树**。   
transient Node[] table;



# **HashMap 的工作原理？**

HashMap 底层是 **hash 数组**和**单向链表**实现，数组中的每个元素都是**链表**，由 **Node 内部类（实现 Map.Entry 接口）**实现，HashMap 通过 put & get 方法存储和获取。

**存储对象**时，将 K/V 键值传给 put() 方法：

1. 、调用 hash(K) 方法**计算 K 的 hash 值**，然后结合数组长度，计算得数组下标；
2. 、**调整数组大小**（当容器中的元素个数大于 capacity \* loadfactor 时，容器会进行扩容resize 为 2n）；
3. 、 i.如果 **K 的 hash 值**在 HashMap 中**不存在**，则执行**插入**，若存在，则发生**碰撞**；   
    ii.如果 K 的 hash 值在 HashMap 中**存在**，且它们两者 **equals 返回 true**，则**更新键值对**；   
    iii. 如果 K 的 hash 值在 HashMap 中**存在**，且它们两者 **equals 返回 false**，则**插入链表的尾部或者红黑树中**。

（**JDK 1.7** 之前使用**头插法**、**JDK 1.8** 使用**尾插法**）   
（注意：当碰撞导致链表大于 TREEIFY\_THRESHOLD = 8 时，就把链表转换成红黑树）

**获取对象**时，将 K 传给 get() 方法：

1. 、调用 hash(K) 方法（**计算 K 的 hash 值**）从而**获取该键值所在链表的数组下标**；
2. 、顺序遍历链表，equals()方法查找**相同 Node 链表中 K 值**对应的 V 值。

hashCode 是定位的，**存储位置**；equals是定性的，**比较两者是否相等**

# HashMap 的 table 的容量如何确定？loadFactor 是什么？ 该容量如何变化？这种变化会带来什么问题？

①、**table 数组大小**是由 **capacity** 这个参数确定的，默认是**16**，也可以构造时传入，最大限制是1<<30；   
②、**loadFactor 是装载因子**，主要目的是用来**确认table 数组是否需要动态扩展**，默认值是**0.75**，比如table 数组大小为 16，装载因子为 0.75 时，threshold 就是12，当 table 的实际大小超过 12 时，table就需要动态扩容；   
③、扩容时，调用 resize() 方法，将 **table 长度变为原来的两倍**（注意是 **table 长度**，而不是 threshold）   
④、如果数据很大的情况下，扩展时将会带来性能的损失，在性能要求很高的地方，这种损失很可能很致命。

# HashMap 和 HashTable 有什么区别？

①、**HashMap** 是线程不安全的，**HashTable** 是**线程安全**的；   
②、由于线程安全，所以 HashTable 的效率比不上 HashMap；   
③、**HashMap**最多只允许**一条记录的键为null**，允许**多条记录的值为null**，而 **HashTable** 不允许；   
④、**HashMap** 默认初始化数组的大小为**16**，**HashTable** 为 **11**，前者扩容时，扩大**两倍**，后者**扩大两倍+1**；   
⑤、HashMap 需要重新计算 hash 值，而 HashTable 直接使用对象的 hashCode

# 为什么 ConcurrentHashMap 比 HashTable 效率要高？

A：**HashTable** 使用**一把锁（锁住整个链表结构）**处理并发问题，多个线程竞争一把锁，容易**阻塞**；

**ConcurrentHashMap**   
**JDK 1.7** 中使用**分段锁（ReentrantLock + Segment + HashEntry）**，相当于把一个 HashMap 分成多个段，每段分配一把锁，这样支持多线程访问。锁粒度：**基于 Segment**，包含多个 HashEntry。   
**JDK 1.8** 中使用 **CAS + synchronized + Node + 红黑树**。锁粒度：**Node（首结点）**（实现 Map.Entry）。锁粒度降低了。

# ConcurrentHashMap 简单介绍？

①、重要的常量：   
private transient volatile int **sizeCtl**;   
当为负数时，-1 表示正在初始化，-N 表示 N - 1 个线程正在进行扩容；   
当为 0 时，表示 table 还没有初始化；   
当为其他正数时，表示初始化或者下一次进行扩容的大小。

②、数据结构：   
**Node 是存储结构的基本单元**，继承 HashMap 中的 Entry，用于**存储数据**；   
**TreeNode 继承 Node**，但是数据结构换成了二叉树结构，是红黑树的存储结构，用于**红黑树中存储数据**；   
**TreeBin 是封装 TreeNode 的容器**，提供转换红黑树的**一些条件和锁的控制**。

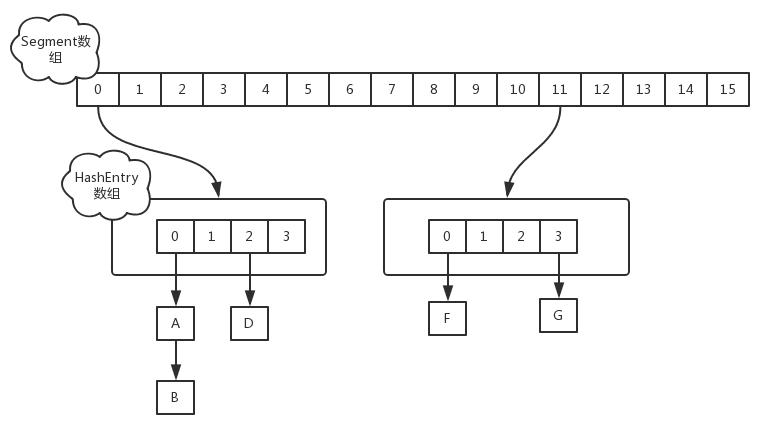
③、**存储对象**时（**put()** 方法）：   
1.如果没有初始化，就调用 initTable() 方法来进行**初始化**；   
2.如果没有 hash 冲突就直接 **CAS 无锁插入**；   
3.如果需要扩容，就先进行**扩容**；   
4.如果存在 hash 冲突，就**加锁**来保证线程安全，两种情况：一种是链表形式就直接遍历到**尾端插入**，一种是红黑树就按照红黑树结构插入；   
5.如果该链表的数量大于阀值 8，就要先**转换成红黑树**的结构，break 再一次进入循环   
6.如果添加成功就调用 **addCount() 方法统计 size**，并且**检查是否需要扩容**。

④、**扩容方法 transfer()**：默认容量为 **16**，扩容时，容量变为原来的**两倍**。   
helpTransfer()：调用**多个工作线程**一起帮助进行扩容，这样的效率就会更高。

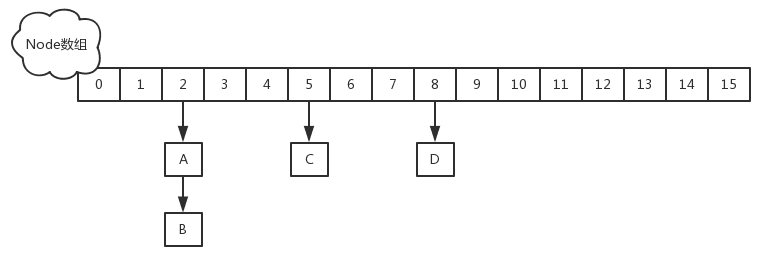
⑤、**获取对象**时（**get()**方法）：   
1.**计算 hash 值**，定位到该 table 索引位置，如果是首结点符合就返回；   
2.如果遇到扩容时，会调用标记正在扩容结点 ForwardingNode.find()方法，查找该结点，匹配就返回；   
3.以上都不符合的话，就往下遍历结点，匹配就返回，否则最后就返回 null。

# 针对 ConcurrentHashMap 锁机制具体分析（JDK 1.7 VS JDK 1.8）？

**JDK 1.7** 中，采用**分段锁**的机制，实现并发的更新操作，底层采用**数组+链表**的存储结构，包括两个核心静态内部类 **Segment 和 HashEntry**。   
①、**Segment** 继承 **ReentrantLock（重入锁）** 用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶；   
②、HashEntry 用来封装映射表的键-值对；   
③、每个桶是由若干个 HashEntry 对象链接起来的链表。



**JDK 1.8** 中，采用Node + CAS + Synchronized来保证并发安全。**取消类 Segment**，直接用 **table 数组**存储键值对；当 HashEntry 对象组成的链表长度超过 TREEIFY\_THRESHOLD 时，**链表转换为红黑树**，提升性能。底层变更为**数组 + 链表 + 红黑树**。

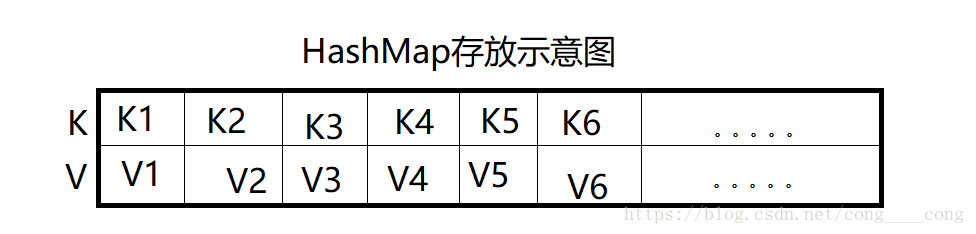


# 请说说HashSet原理，并写程序证明

HashSet在存元素时，会调用对象的hashCode方法计算出存储位置，然后和该位置上所有的元素进行equals比较，   
如果该位置没有其他元素或者比较的结果都为false就存进去，否则就不存。   
这样的原理注定了元素是按照哈希值来找存储位置，所有无序，而且可以保证无重复元素   
我们在往HashSet集合存储元素时，对象应该正确重写Object类的hashCode和equals方法   
正因为这样的原理，HashSet集合是非常高效的。   
比如，要查找集合中是否包含某个对象，首先计算对象的hashCode，折算出位置号，到该位置上去找就可以了，而不用和所有的元素都比较一遍

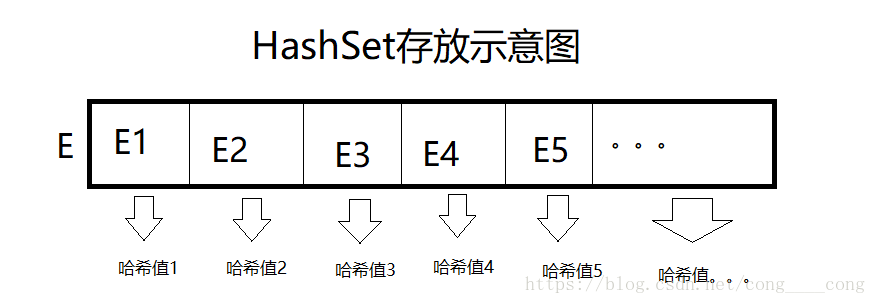
什么是HashMap?

    HashMap是实现Map<K,V>接口的一个实体类，它对键值做了一对一的映射关系，当然里面键值不能重复。Map 接口提供三种collection 视图，允许以键集、值集或键-值映射关系集的形式查看某个映射的内容。映射顺序 定义为迭代器在映射的 collection 视图上返回其元素的顺序。某些映射实现可明确保证其顺序，如 TreeMap 类；另一些映射实现则不保证顺序，如 HashMap 类。

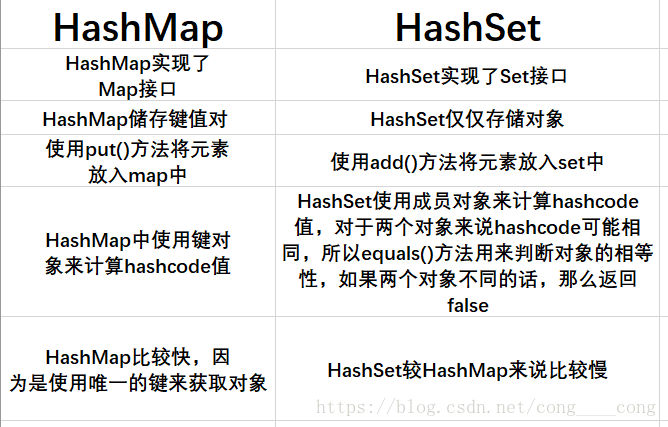


什么是HashSet?

    HashSet是实现Set<E>接口的一个实体类，数据是以哈希表的形式存放的，里面的不能包含重复数据。Set接口是一种一个不包含重复元素的 collection。



HashMap和HashSet的区别？



# Set里的元素是不能重复的，那么用什么方法来区分重复与否呢? 是用==还是equals()? 它们有何区别?

1、什么是Set?(what)

     Set是Collection容器的一个子接口，它不允许出现重复元素，当然也只允许有一个null对象。

2、如何来区分重复与否呢？(how)

      “ 用 *iterator()* 方法来区分重复与否 ”，这是在网上流传的答案，个人认为这是个错误的答案。JPI中写的很明白：“set 不包含满足e1.equals(e2) 的元素对 e1 和 e2 ”，由此可见回答使用equals()区分更合适。

3、为什么用equals()而不用==来区分？(why)

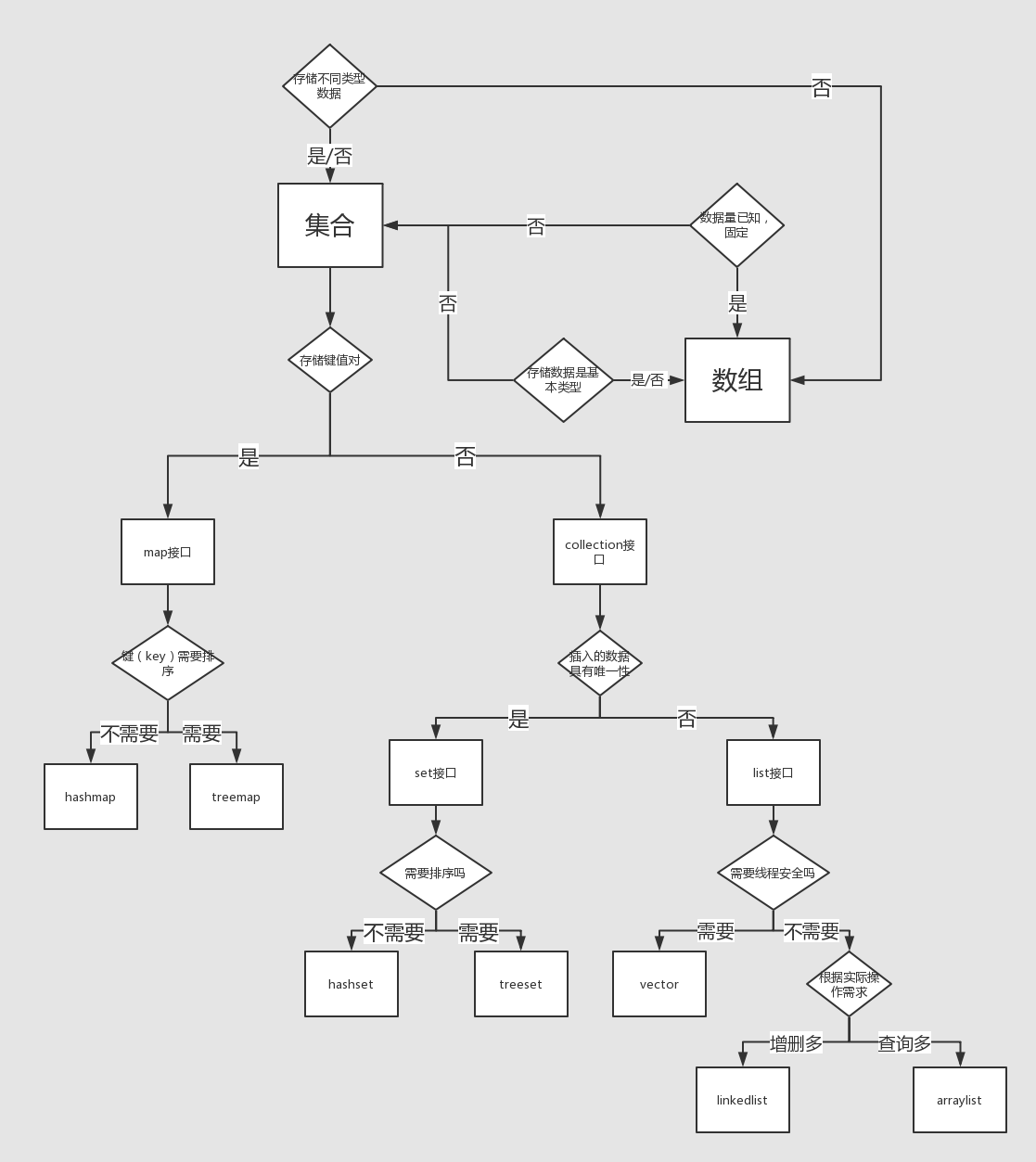
     应该从它俩的区别谈起，==是用来判断两者是否是同一对象（同一事物），而equals是用来判断是否引用同一个对象。再看一下Set里面存的是对象，还是对象的引用。根据java的存储机制可知，set里面存放的是对象的引用，所以当两个元素只要满足了equals()时就已经指向同一个对象，也就出现了重复元素。所以应该用equals()来判断。

# 为什么Map接口不继承Collection接口？

尽管Map接口和它的实现也是集合框架的一部分，但Map不是集合，集合也不是Map。因此，Map继承Collection毫无意义，反之亦然。  
如果Map继承Collection接口，那么元素去哪儿？Map包含key-value对，它提供抽取key或value列表集合的方法，但是它不适合“一组对象”规范。

# 集合类框架的最佳实践有哪些?

根据应用的需要选择合适的集合对性能是非常重要的。如果一个集合的元素数量是固定的，而且我们能够提前知道固定的数量，那么就可以使用数组，而不是ArrayList。 每个集合都可以设置初始容量，如果我们提前能够估算出它的初始容量，那么就可以避免重新计算它的hash值与扩容。 为了保证程序的类型安全、健壮性与可读性，一般我们都需要使用泛型。泛型可以避免运行时的ClassCastException。同时如果使用不可变类作为map的键，那么可以避免为我们自己实现的类实现equals()与hashcode()方法。 编程的时候接口优于实现。 如果底层的集合为空，返回的集合要为长度为0的集合或者数组，不能为空。 总而言之一句话，如果频繁用于查找，则用数组类型的集合，如果频繁用于删除，则使用链表类型的集合，下面是一张总的集合适用场合的图片



# 数组、ArrayList和List三者的区别

**List**是针对特定类型、任意长度的。

**Array** 是针对任意类型、固定长度的。

**ArrayList**是针对任意类型、任意长度的。

Array 和 ArrayList 是通过存储 object 实现任意类型的，所以使用时要转换。

在C#中数组，ArrayList，List都能够存储一组对象，那么这三者到底有什么样的区别呢。

数组

    数组在C#中最早出现的。在内存中是连续存储的，所以它的索引速度非常快，而且赋值与修改元素也很简单。

1. //数组
2. **string**[] s=**new** **string**[2];
4. //赋值
5. s[0]="a";
6. s[1]="b";
7. //修改
8. s[1]="a1";

    但是数组存在一些不足的地方。在数组的两个数据间插入数据是很麻烦的，而且在声明数组的时候必须指定数组的长度，数组的长度过长，会造成内存浪费，过短会造成数据溢出的错误。如果在声明数组时我们不清楚数组的长度，就会变得很麻烦。

    针对数组的这些缺点，C#中最先提供了ArrayList对象来克服这些缺点。 

ArrayList

    ArrayList是命名空间System.Collections下的一部分，在使用该类时必须进行引用，同时继承了IList接口，提供了数据存储和检索。ArrayList对象的大小是按照其中存储的数据来动态扩充与收缩的。所以，在声明ArrayList对象时并不需要指定它的长度。

1. //ArrayList
2. ArrayList list1 = **new** ArrayList();
4. //新增数据
5. list1.Add("cde");
6. list1.Add(5678);
8. //修改数据
9. list[2] = 34;
11. //移除数据
12. list.RemoveAt(0);
14. //插入数据
15. list.Insert(0, "qwe");

从上面例子看，ArrayList好像是解决了数组中所有的缺点，为什么又会有List？

    我们从上面的例子看，在List中，我们不仅插入了字符串cde，而且插入了数字5678。这样在ArrayList中插入不同类型的数据是允许的。因为ArrayList会把所有插入其中的数据当作为object类型来处理，在我们使用ArrayList处理数据时，很可能会报类型不匹配的错误，也就是ArrayList不是类型安全的。在存储或检索值类型时通常发生装箱和取消装箱操作，带来很大的性能耗损。

    装箱与拆箱的概念：  
    简单的说:  
装箱：就是将值类型的数据打包到引用类型的实例中  
    比如将string类型的值abc赋给object对象obj

1. String  i=”abc”;
2. **object** obj=(**object**)i;

拆箱：就是从引用数据中提取值类型  
    比如将object对象obj的值赋给string类型的变量i

1. **object** obj=”abc”;
2. **string** i=(**string**)obj;

装箱与拆箱的过程是很损耗性能的。 

泛型List

    因为ArrayList存在不安全类型与装箱拆箱的缺点，所以出现了泛型的概念。List类是ArrayList类的泛型等效类，它的大部分用法都与ArrayList相似，因为List类也继承了IList接口。最关键的区别在于，在声明List集合时，我们同时需要为其声明List集合内数据的对象类型。

比如：

1. List<**string**> list = **new** List<**string**>();
3. //新增数据
4. list.Add(“abc”);
6. //修改数据
7. list[0] = “def”;
9. //移除数据
10. list.RemoveAt(0);

上例中，如果我们往List集合中插入int数组123，IDE就会报错，且不能通过编译。这样就避免了前面讲的类型安全问题与装箱拆箱的性能问题了。

总结：

    数组的容量是固定的，您只能一次获取或设置一个元素的值，而ArrayList或List<T>的容量可根据需要自动扩充、修改、删除或插入数据。

    数组可以具有多个维度，而 ArrayList或 List< T> 始终只具有一个维度。但是，您可以轻松创建数组列表或列表的列表。特定类型（Object 除外）的数组 的性能优于 ArrayList的性能。 这是因为 ArrayList的元素属于 Object 类型；所以在存储或检索值类型时通常发生装箱和取消装箱操作。不过，在不需要重新分配时（即最初的容量十分接近列表的最大容量），List< T> 的性能与同类型的数组十分相近。

    在决定使用 List<T> 还是使用ArrayList 类（两者具有类似的功能）时，记住List<T> 类在大多数情况下执行得更好并且是类型安全的。如果对List< T> 类的类型T 使用引用类型，则两个类的行为是完全相同的。但是，如果对类型T使用值类型，则需要考虑实现和装箱问题。

# ArrayList

如果空间不够,会通过Arrays.copyOf创建一个新的内存空间,新空间的大小最小为原始空间的**3/2倍+1**,并将原始的内容拷贝进去;

# HashMap

大家熟知的HashMap浪费空间更加严重,它的代码里面有一个0.75因子,当写入HashMap的数据个数(不是说所使用的数组下标个数,而是所有元素个数,也就是说,包含了同一个下标的链表中的所以元素个数)达到数组长度的0.75后,数组会自动扩展1倍;并且还需要做一个rehash操作,其实这个时候也许很多桶上的节点都是空的.