龙果学院
roncoo.com

2015年 版教程

Java + 您

全栈修养

技术与自我营销两手抓，做一个有追求的程序猿

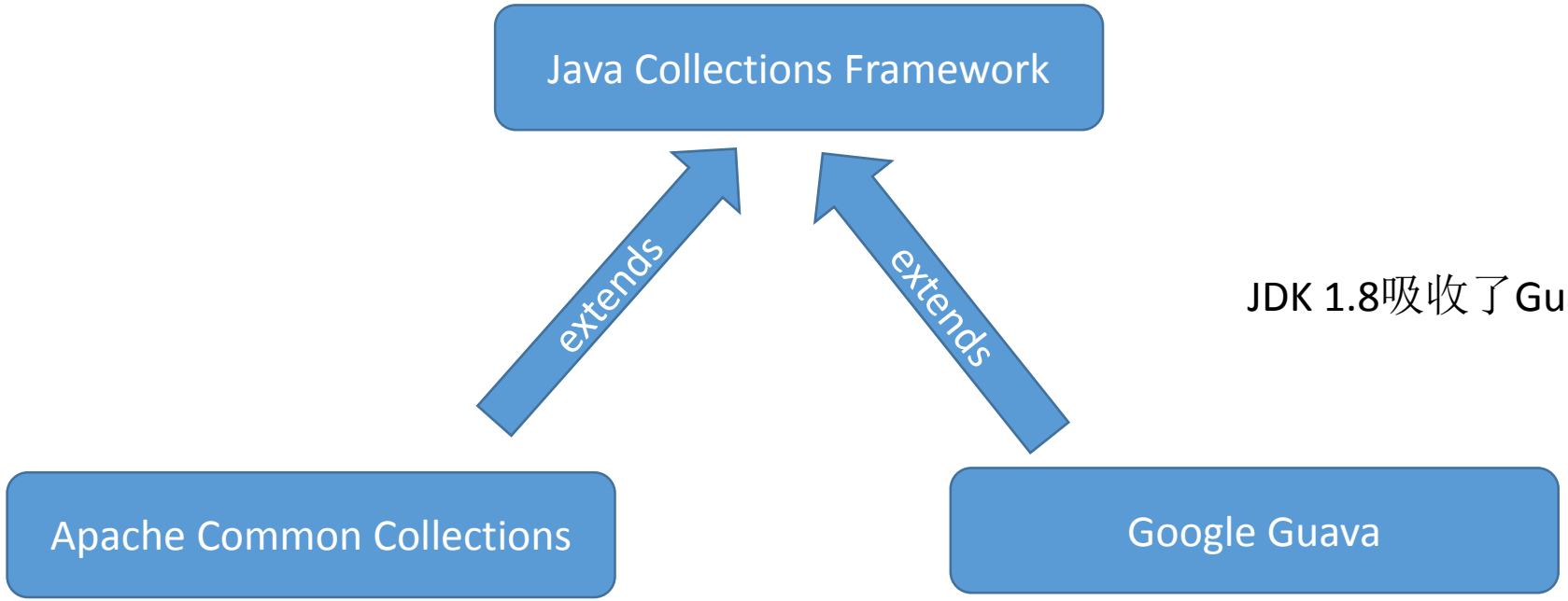
Leader.us@Mycat

Java集合框架概述

有多种集合框架

Java的集合框架是Java类库当中使用频率最高的部分

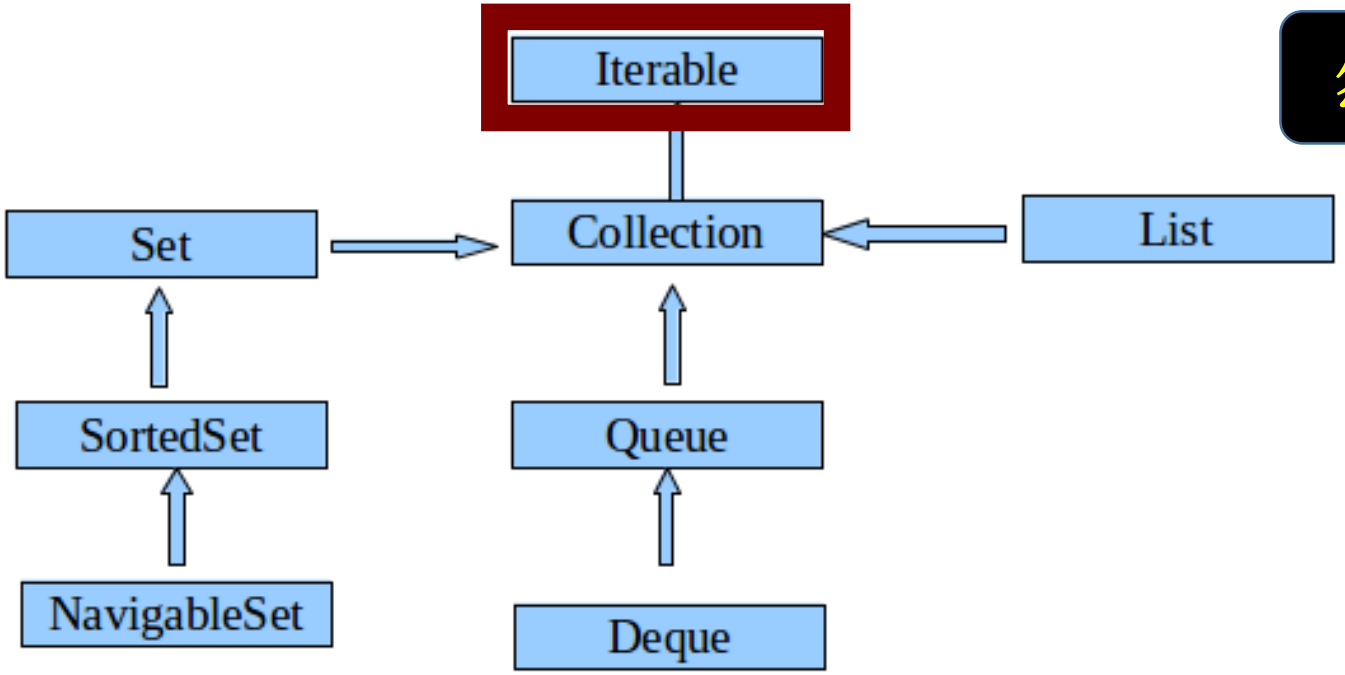
Java Collections Framework是Java 1.2里的重要里程碑，奠定了通用的数据结构在Java里设计和实现，但它的实现还是不够“丰富”，所以Apache出了一个Common collections，由于不能支持JDK5的泛型导致谷歌重新实现了自己的集合框架Guava，计划纳入JDK7。



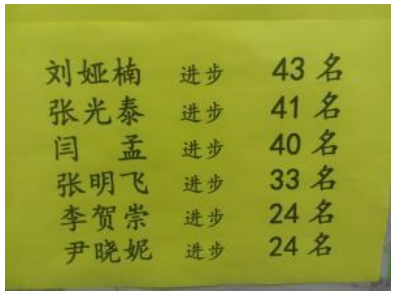
JDK 1.8吸收了Guava中的一些内容

Collection

可遍历的、可伸缩的、可查询的



然而“可排序”并不是集合的要求



如果外国人也从小就排名考试成绩的话....

不至于我都30多了还TMD每次面试都被TMD的这么复杂的集合框架面试题秒杀了....

Iterator设计模式

巧妙的解决了无限的数据→有限的内存这个矛盾

四两拨千斤

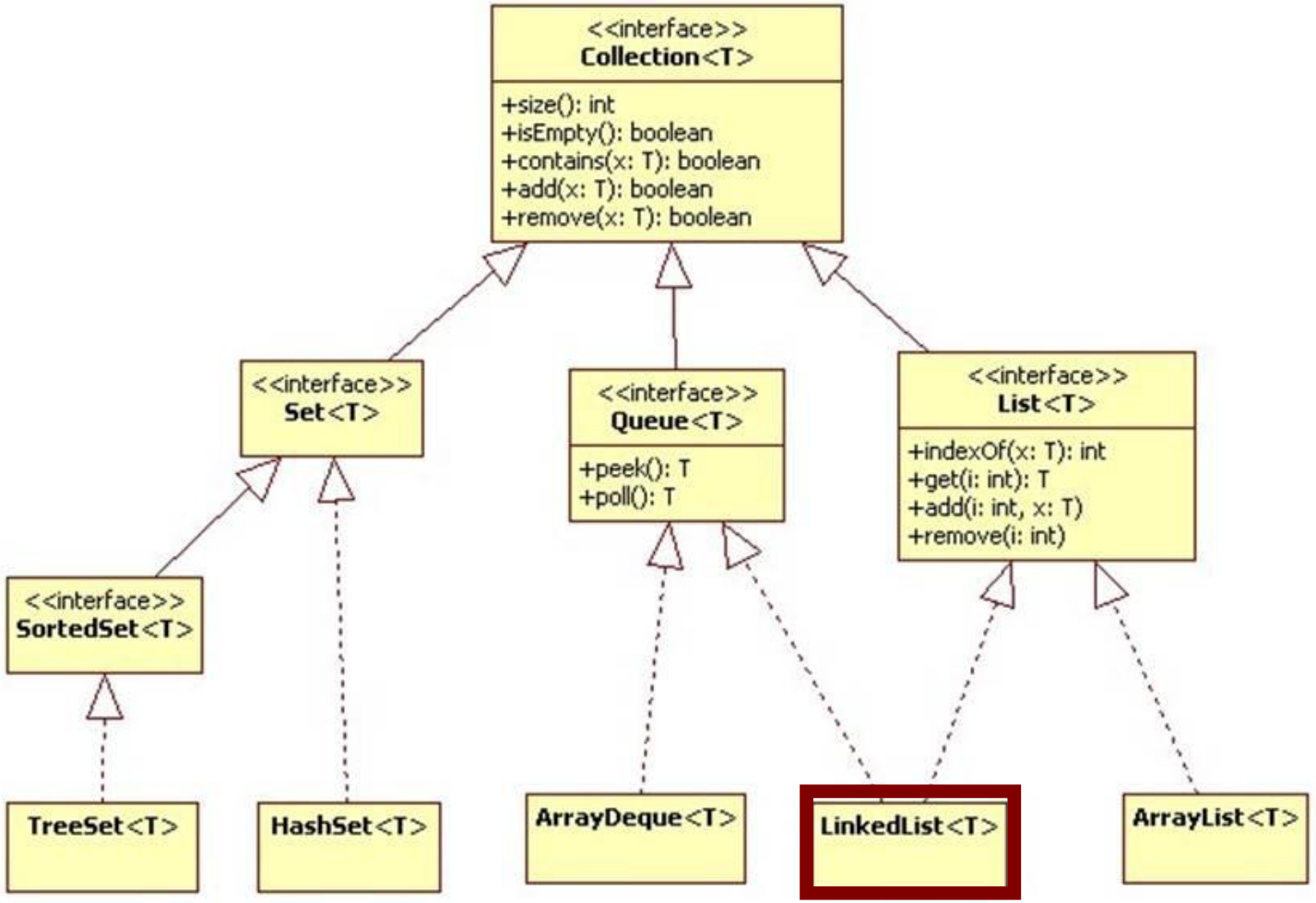
```
public interface Iterator {
        boolean hasNext();
        Object next();
        void remove();
        }
```

```
private class Itr implements Iterator {
        int cursor = 0;
        int lastRet = -1;
        int expectedModCount = modCount;
        }
```

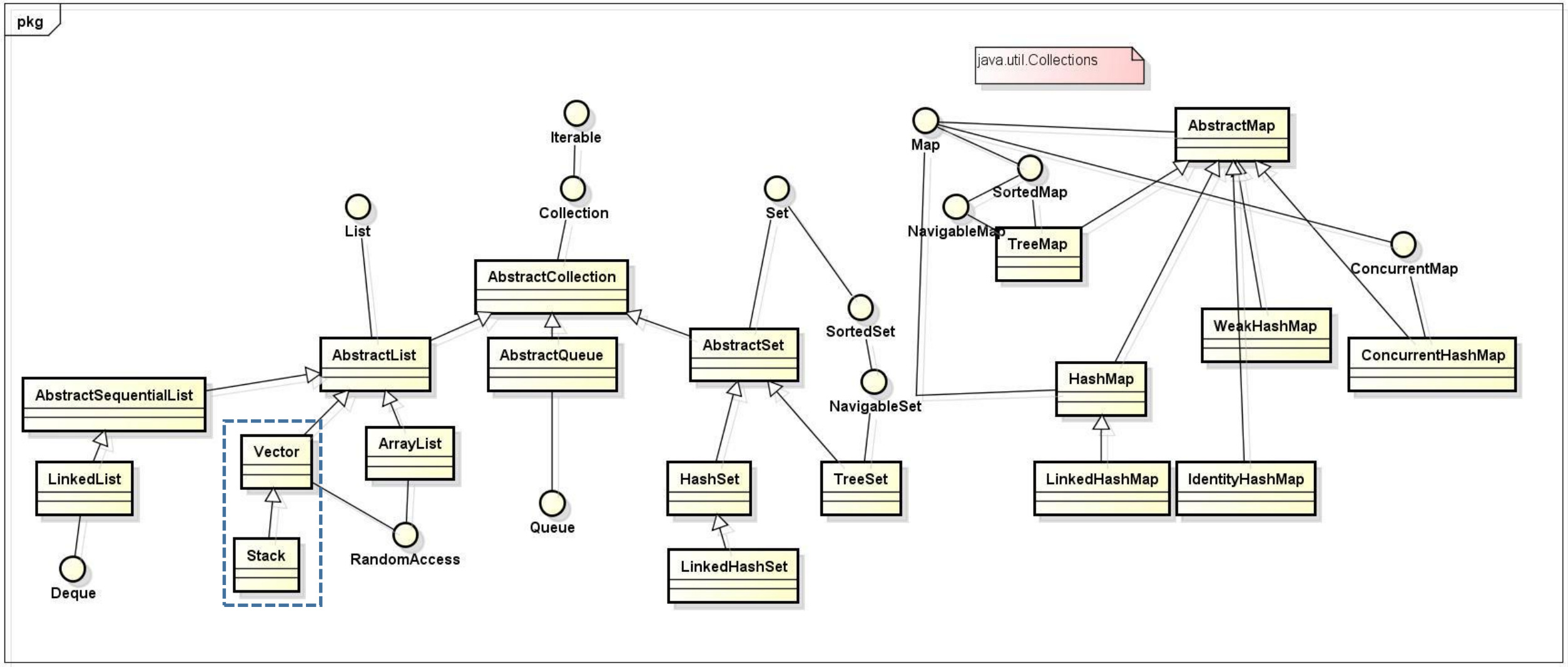
```
public boolean hasNext() {
        return cursor != size();
        }
```

```
public Object next() {
        checkForComodification();
        try {
        Object next = get(cursor);
        lastRet = cursor++;
        return next;
        } catch(IndexOutOfBoundsException e) {
        checkForComodification();
        throw new NoSuchElementException();
        }
        }
```

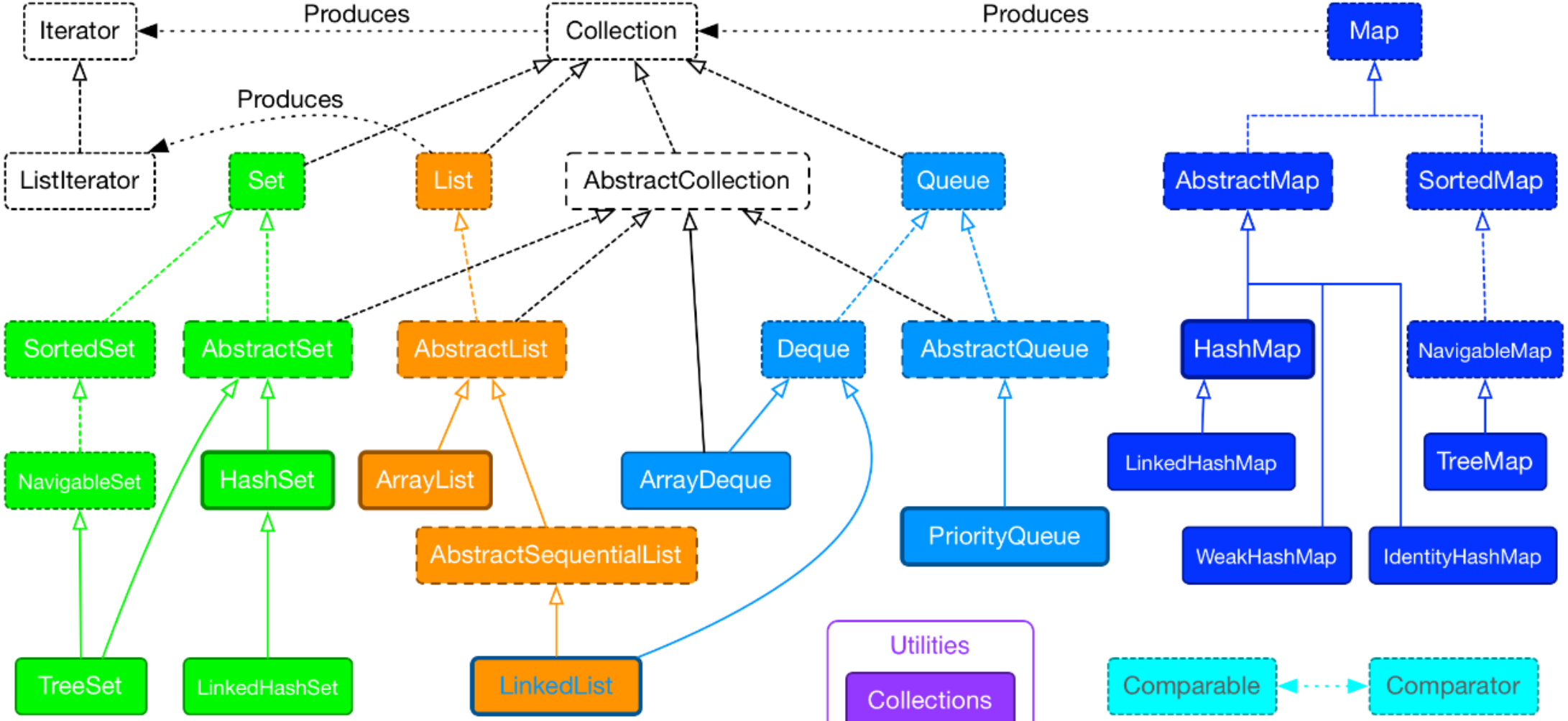
Collection



Collection

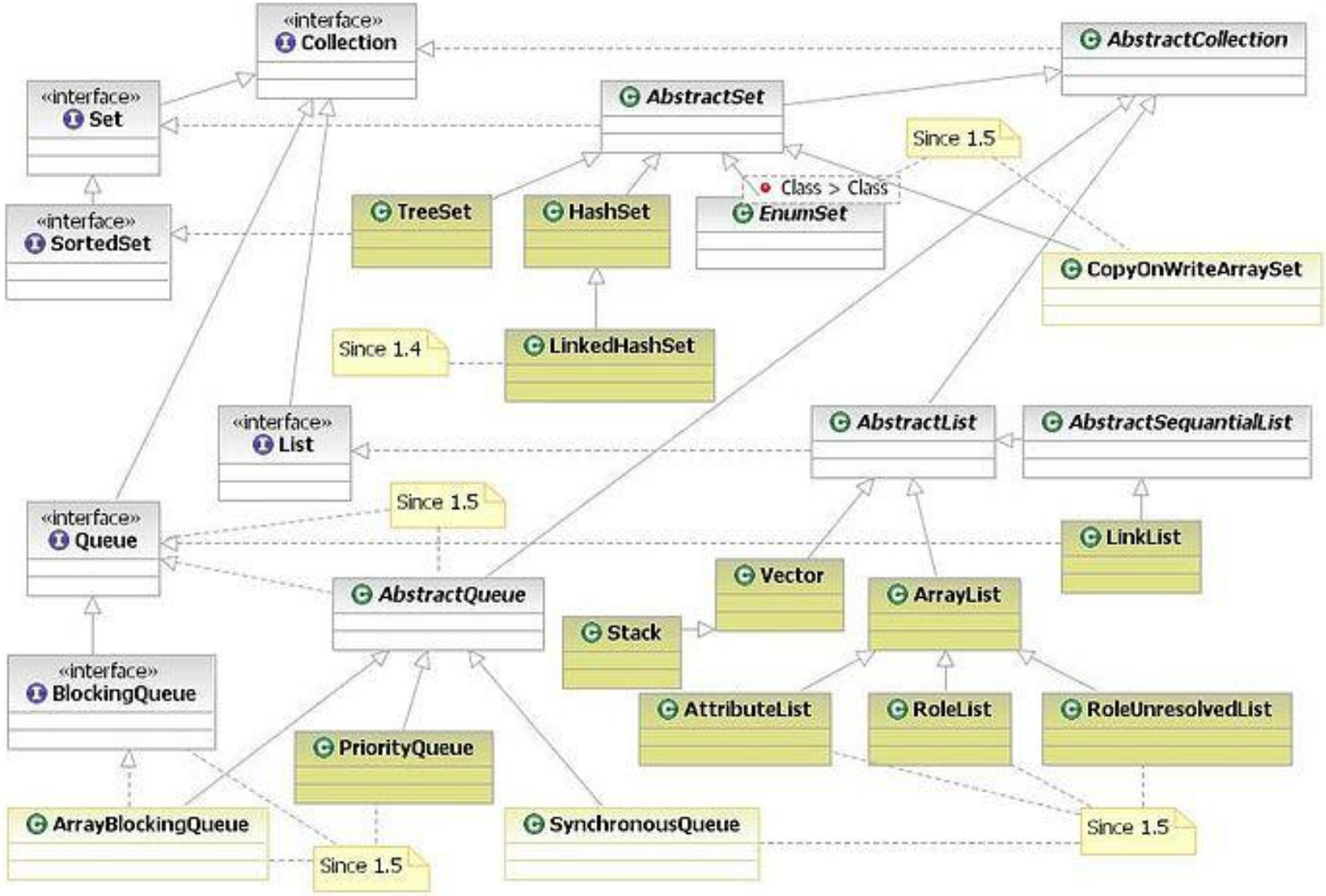


Collection

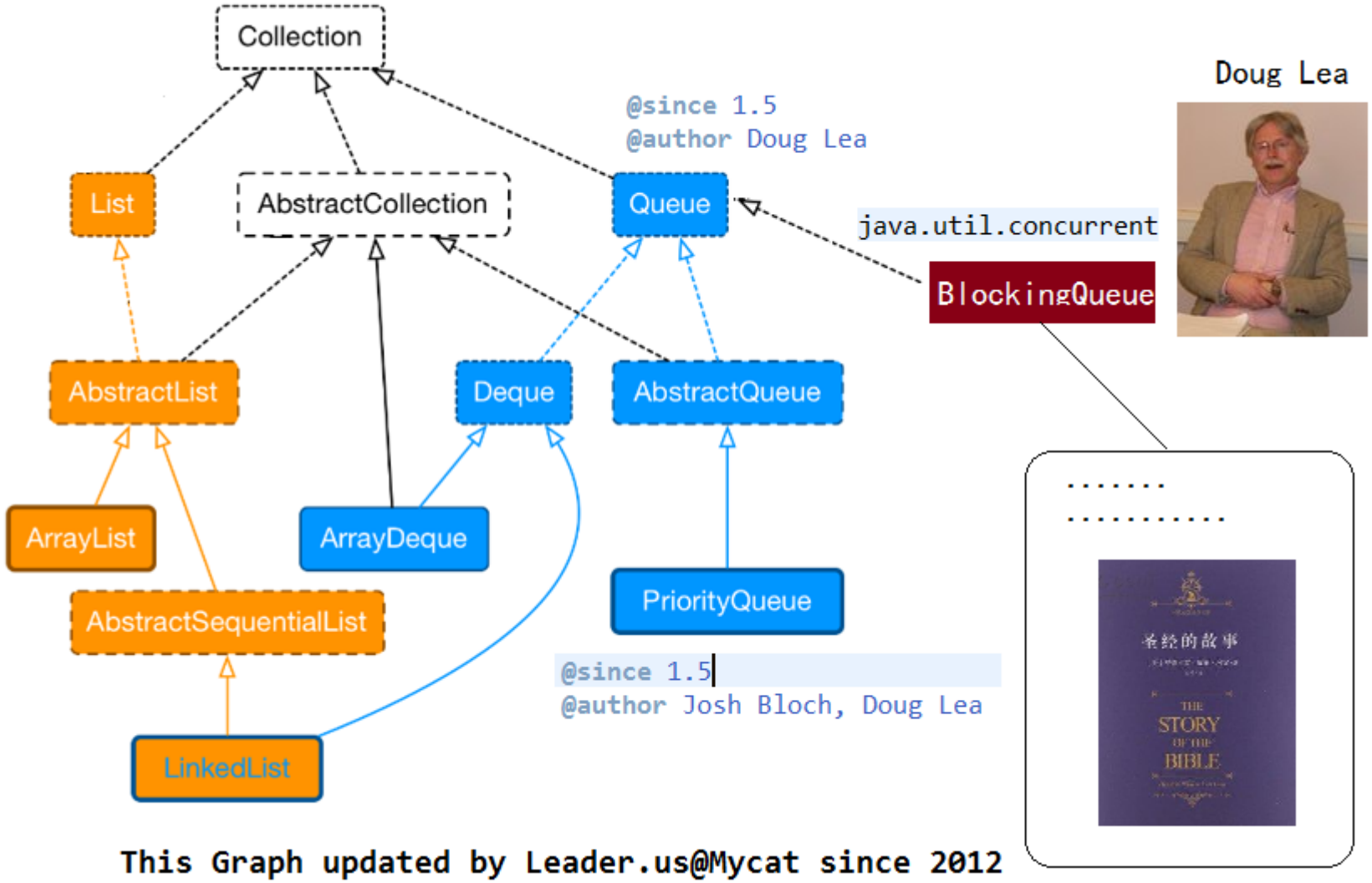


ArrayDeque 被用作栈时比 Stack 要快，当用作 Queue 时比 LinkedList 要快

Collection



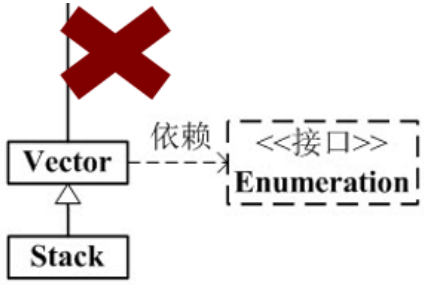
Queue&List



Doug Lea

请注意！！

Queue一点都不好玩，是美国知名IT教授的玩具，咱无证程序猿们看一眼就可以走了。



Deque不仅具有FIFO的Queue实现，也有FILO的实现，也就是不仅可以实现队列，也可以实现一个堆栈。

PriorityQueue

PriorityQueue其实是一个优先队列，每次出队的元素都是优先级最高的元素。那么怎么确定哪一个元素的优先级最高呢，jdk中使用堆这么一种数据结构，通过堆使得每次出队的元素总是队列里面最小的，而元素的大小比较方法可以由用户指定，这里就相当于指定优先级喽。

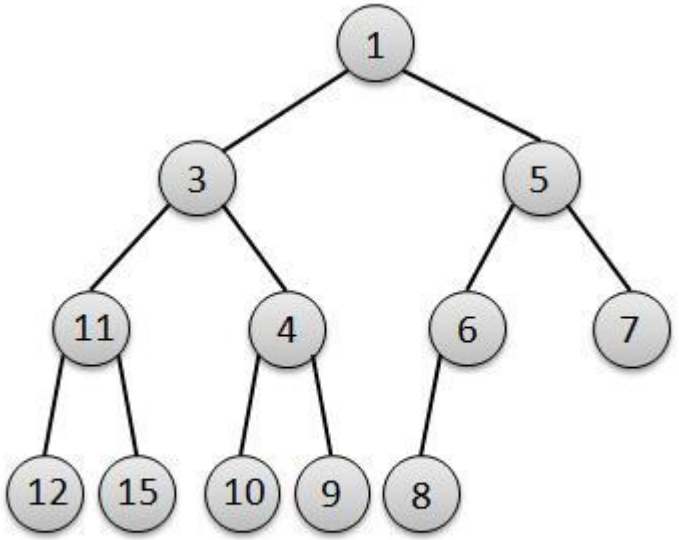
```
 */
transient Object[] queue; // non-private to simplify nested class access

/**
 * The number of elements in the priority queue.
 */
private int size = 0;

/**
 * The comparator, or null if priority queue uses elements'
 * natural ordering.
 */
private final Comparator<? super E> comparator;

/**
```

堆又是什么一种数据结构呢、它有什么样的特点呢？

(1)堆中某个节点的值总是不大于或不小于其父节点的值；

(2)堆总是一棵完全树。

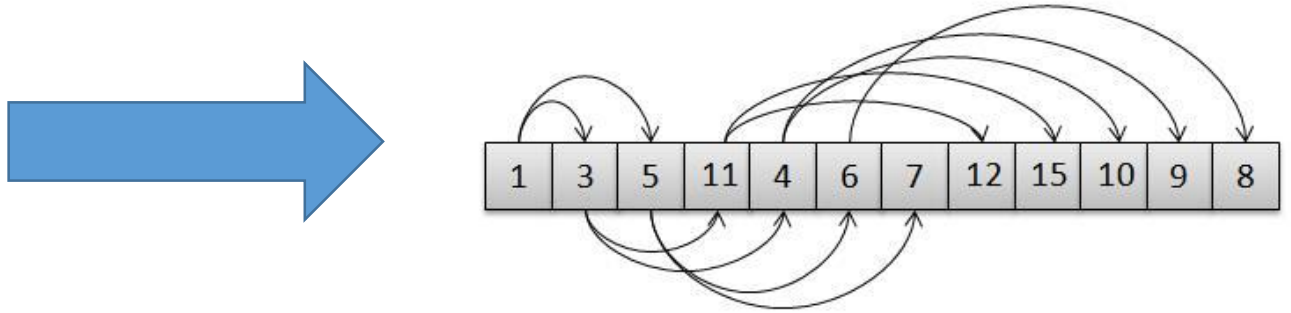
常见的堆有二叉堆、斐波那契堆等。而PriorityQueue使用的便是二叉堆



二叉堆

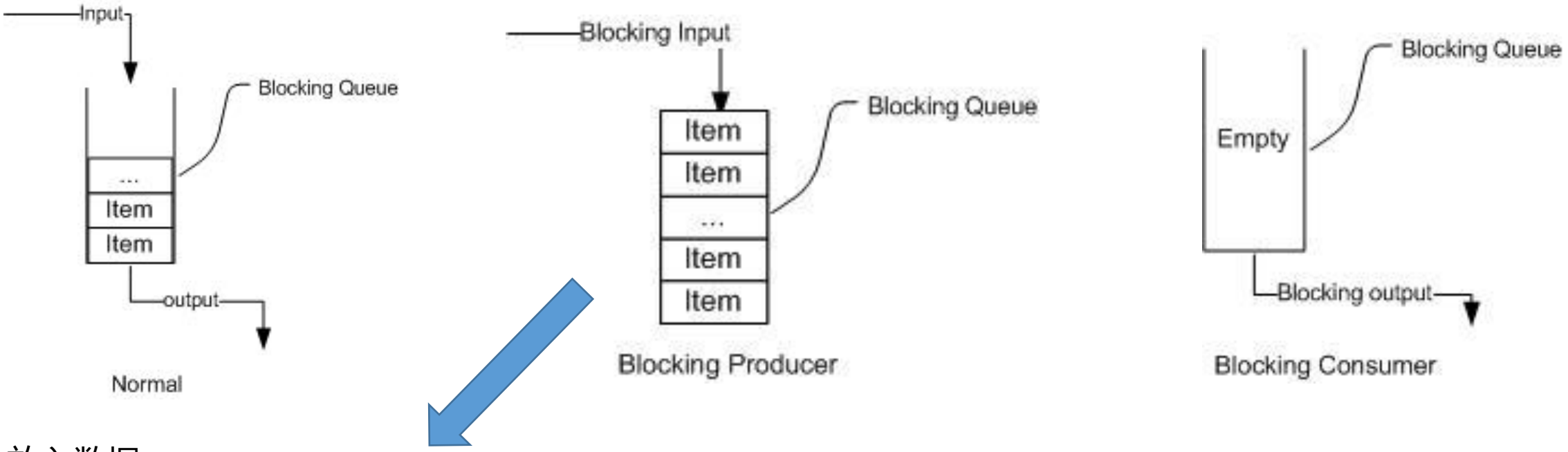
二叉堆又可以用数组来表示而不是用链表：

对于数组中任意位置的n上元素，其左孩子在[2n+1]位置上，右孩子[2(n+1)]位置，它的父亲则在[(n-1)/2]上，而根的位置则是[0]。



用数组表示二叉堆

BlockingQueue



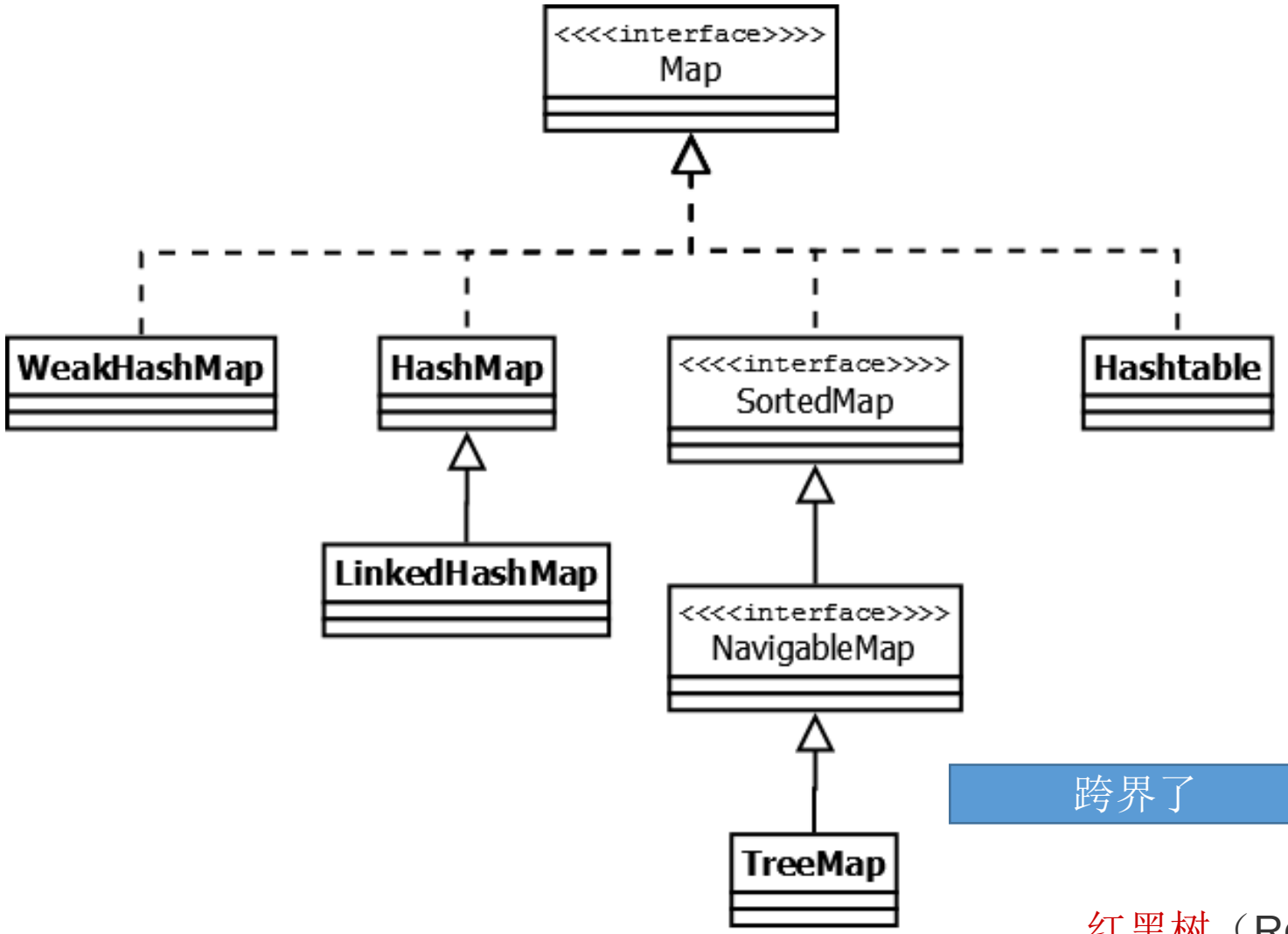
放入数据：

offer(anObject):表示如果可能的话,将anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则返回false.（本方法不阻塞当前执行方法的线程）

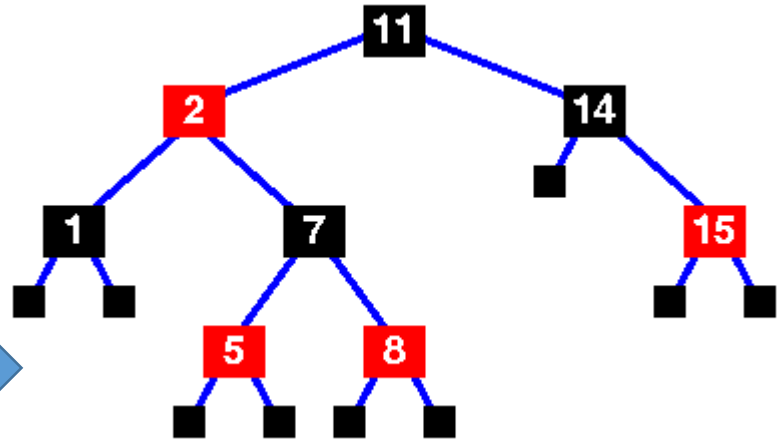
offer(E o, long timeout, TimeUnit unit),可以设定等待的时间，如果在指定的时间内，还不能往队列中加入BlockingQueue，则返回失败。

put(anObject):把anObject加到BlockingQueue里,如果BlockQueue没有空间,则调用此方法的线程被阻断直到BlockingQueue里面有空间再继续.

Map

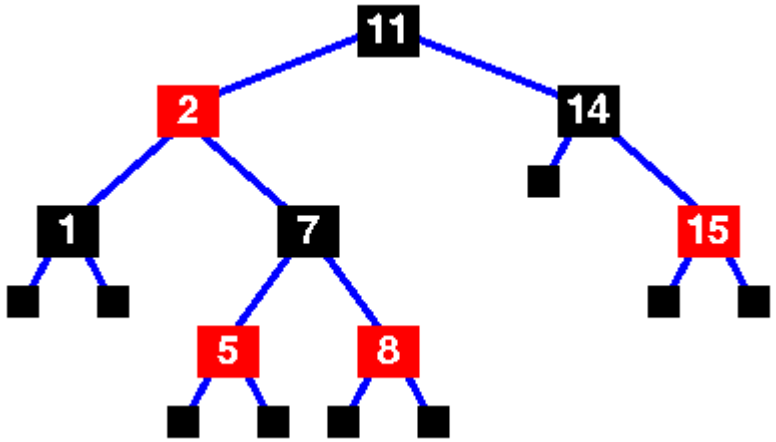


跨界了

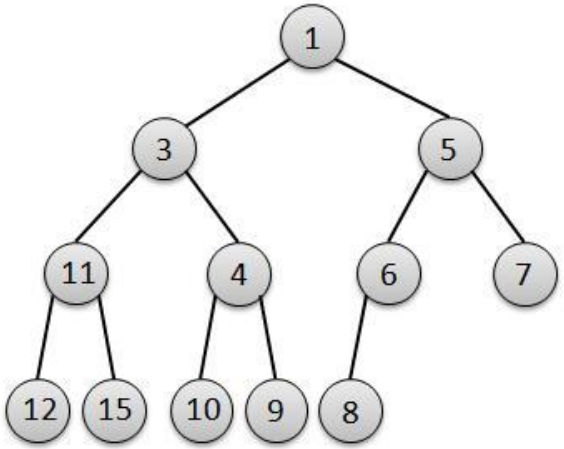


红黑树（Red Black Tree ）是一种自平衡二叉查找树

红黑树VS最小堆



PK



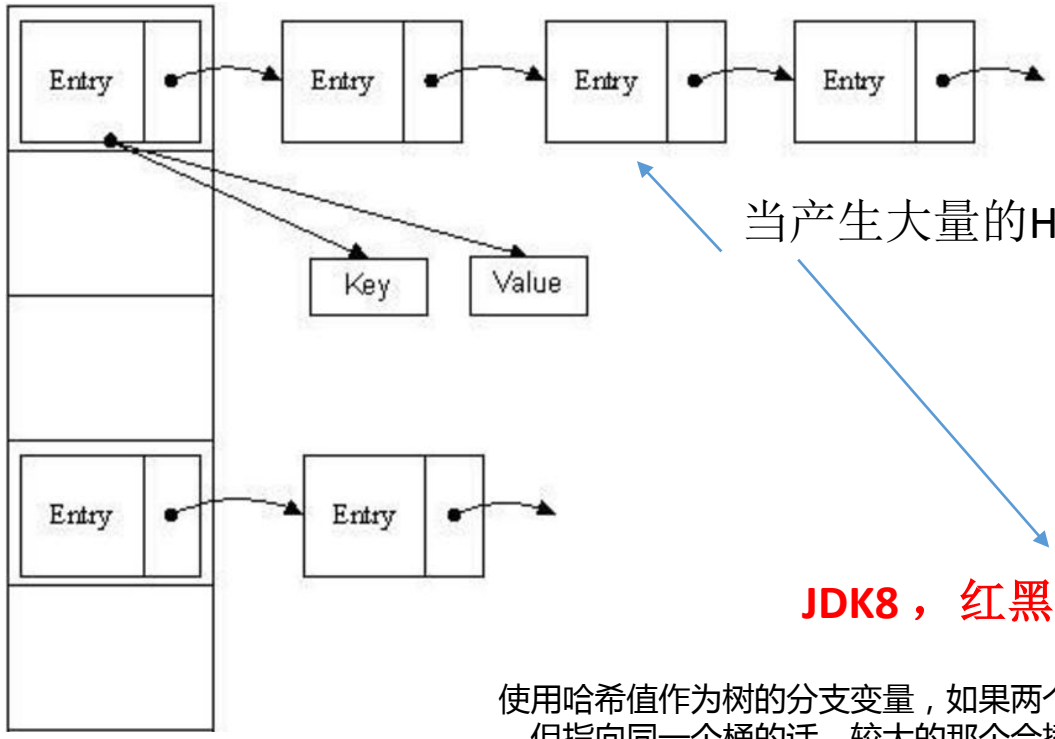
二叉堆

- 由于最小堆一般是采用堆的方式实现，元素访问速度远高于采用链表方式的红黑树，插入性能快红黑树好几倍，但最小堆的删除性能比红黑树差很多
- 最小堆查询顶节点是O(1),而删除顶节点在任何情况下都是个最坏的情况，需要比较2logN次
- 最小堆最大的优势在于取最小值性能是O(1),如果插入的数据基本有序，那么最小堆能获得比较好的性能

如果插入+查找最小+删除最小总体来确定红黑树和最小堆，红黑树占优，性能波动相对较小，红黑树不过多的依赖比较，相对最小堆由比较带来的的性能影响较小（如果比较是调用自定义函数、比较的是字符串等，那么性能就牺牲很大）

Hashmap+Tree?

桶



当产生大量的Hash冲突的时候，Map退化为“链表结构”，性能下降

JDK8，红黑树替代链表

使用哈希值作为树的分支变量，如果两个哈希值不等，但指向同一个桶的话，较大的那个会插入到右子树里

关键点：实现compareTo()接口

```
//链表节点
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
    //省略
}
//红黑树节点
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
        TreeNode<K,V> parent;  // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;    // needed to unlink next upon deletion
        boolean red;
        TreeNode(int hash, K key, V val, Node<K,V> next) {
            super(hash, key, val, next);
        }
        //省略
}
```

当Map中的键值对个数超过Map的容量*加载因子时候，会产生翻倍容量的扩容

Hashmap的性能取决于键值的分布状态

WeakHashMap

o是一个强引用，强引用指向的对象无论在何时，都不会被GC 清理掉

Object o = new Object(); ——→ 强引用就好像结婚证，一证在手，不怕老婆走丢！

java.lang.ref.Reference ——→ （无证经营的）女盆友、好基友、好同事、好同学

辛苦加了一个月的班，木然回首，**只见阑珊灯火不见TA....**

- java.lang.ref.SoftReference 软引用
- java.lang.ref.WeakReference 弱引用
- ava.lang.ref.PhantomReference 虚引用

强->软->弱->虚

求助啊,程序员真是一条不归路吗?_程序员吧_百度贴吧

19条回复 - 发帖时间: 2013年7月12日

想达到高级程序员的位置,至少还要多奋斗几年,很多已经程序程序员的人都劝我们准备踏进不归路的人赶紧改行,我都迷茫了,怎么办啊。该继续学不

存折还有钱=对象还在　口袋还有钱=对象还在　已经回天无力，你得到一个坏消息！

从JDK 1.2版本开始，把对象的引用分为4种级别，从而使程序能更加灵活地控制对象的生命周期。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。

WeakHashMap

SoftReference<String> ss = new SoftReference<String>("abc");
WeakReference<String> ss = new WeakReference<String>("abc");

ss指向abc这个对象，abc 会在一定时机被 GC自动清理

GC的过程中

- 发现当一个对象只有WeakReference引用的时候，**GC会清理掉该对象。（有一个猫捉老鼠的过程）**
- 只有发现可用内存不够了，马上要**OOM了，就决定清理只有SoftReference引用的对象**

SoftReference：尽量珍惜TA，不到万不得已不放手！
WeakReference：丢了就丢了，再造一个！

SoftReference可用于内存敏感的缓存系统

Project: *dlna* File: *YouPlayerInterfaceFile.java* *View source code*

```
public Bitmap getFileIcon(Context context) {
        Bitmap temp = null;
        if ((null!= imageCache) && imageCache.containsKey(path)) {
                LOG.v("getFileIcon", "getFileIcon", "get imagecache");
            SoftReference<Bitmap> softReference = imageCache.get(path);
            temp = softReference.get();
            LOG.v("getFileIcon", "getFileIcon", "temp : "+ temp);
        }
        if(temp == null) {
                temp = getDefaultIcon(context);
        }
        return temp;
}
```

WeakHashMap

SoftReference<String> ss = new SoftReference<String>("abc");
WeakReference<String> ss = new WeakReference<String>("abc");

ss指向abc这个对象，abc 会在一定时机被 GC自动清理

ReferenceQueue

当Reference对应的Object被清理，则Reference会放入

GC的过程中

- 发现当一个对象只有WeakReference引用的时候，**GC会清理掉该对象。（有一个猫捉老鼠的过程）**
- **只有发现可用内存不够了，马上要OOM了，就决定清理只有SoftReference引用的对象**

SoftReference：尽量珍惜TA，不到万不得已不放手！
WeakReference：丢了就丢了，再造一个！

SoftReference可用于内存敏感的缓存系统

Project: *dlna* File: *YouPlayerInterfaceFile.java* *View source code*

```
public Bitmap getFileIcon(Context context) {
        Bitmap temp = null;
        if ((null!= imageCache) && imageCache.containsKey(path)) {
                LOG.v("getFileIcon", "getFileIcon", "get imagecache");
            SoftReference<Bitmap> softReference = imageCache.get(path);
            temp = softReference.get();
            LOG.v("getFileIcon", "getFileIcon", "temp : "+ temp);
        }
        if(temp == null) {
                temp = getDefaultIcon(context);
        }
        return temp;
}
```

WeakHashMap

Hash table based implementation of the Map interface, with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use.

Implementation note: The value objects in a WeakHashMap are held by ordinary strong references. Thus care should be taken to ensure that **value objects do not strongly refer to their own keys, either directly or indirectly, since that will prevent the keys from being discarded** More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently from other Map implementations.

expungeStaleEntries()这个方法，这个是整个WeakHashMap的精髓

```
// 清空table中无用键值对。原理如下：
// (01) 当WeakHashMap中某个“弱引用的key”由于没有再被引用而被GC收回时，
// 被回收的“该弱引用key”也被会被添加到"ReferenceQueue(queue)"中。
// (02) 当我们执行expungeStaleEntries时，
// 就遍历"ReferenceQueue(queue)"中的所有key
// 然后就在“WeakReference的table”中删除与“ReferenceQueue(queue)中key”对应的键值对
```

WeakHashMap

WeakHashMap *isn't* useful as a cache, at least the way most people think of it. As you say, it uses weak *keys*, not weak *values*, so it's not designed for what most people want to use it for (and, in fact, I've *seen* people use it for, incorrectly).

WeakHashMap is mostly useful to keep metadata about objects whose lifecycle you don't control. For example, if you have a bunch of objects passing through your class, and you want to keep track of extra data about them without needing to be notified when they go out of scope, and without your reference to them keeping them alive.

A simple example (and one I've used before) might be something like:

```
WeakHashMap<Thread, SomeMetaData>
```

where you might keep track of what various threads in your system are doing; when the thread dies, the entry will be removed silently from your map, and you won't keep the Thread from being garbage collected if you're the last reference to it. You can then iterate over the entries in that map to find out what metadata you have about active threads in your system.

See WeakHashMap in not a cache! for more information.

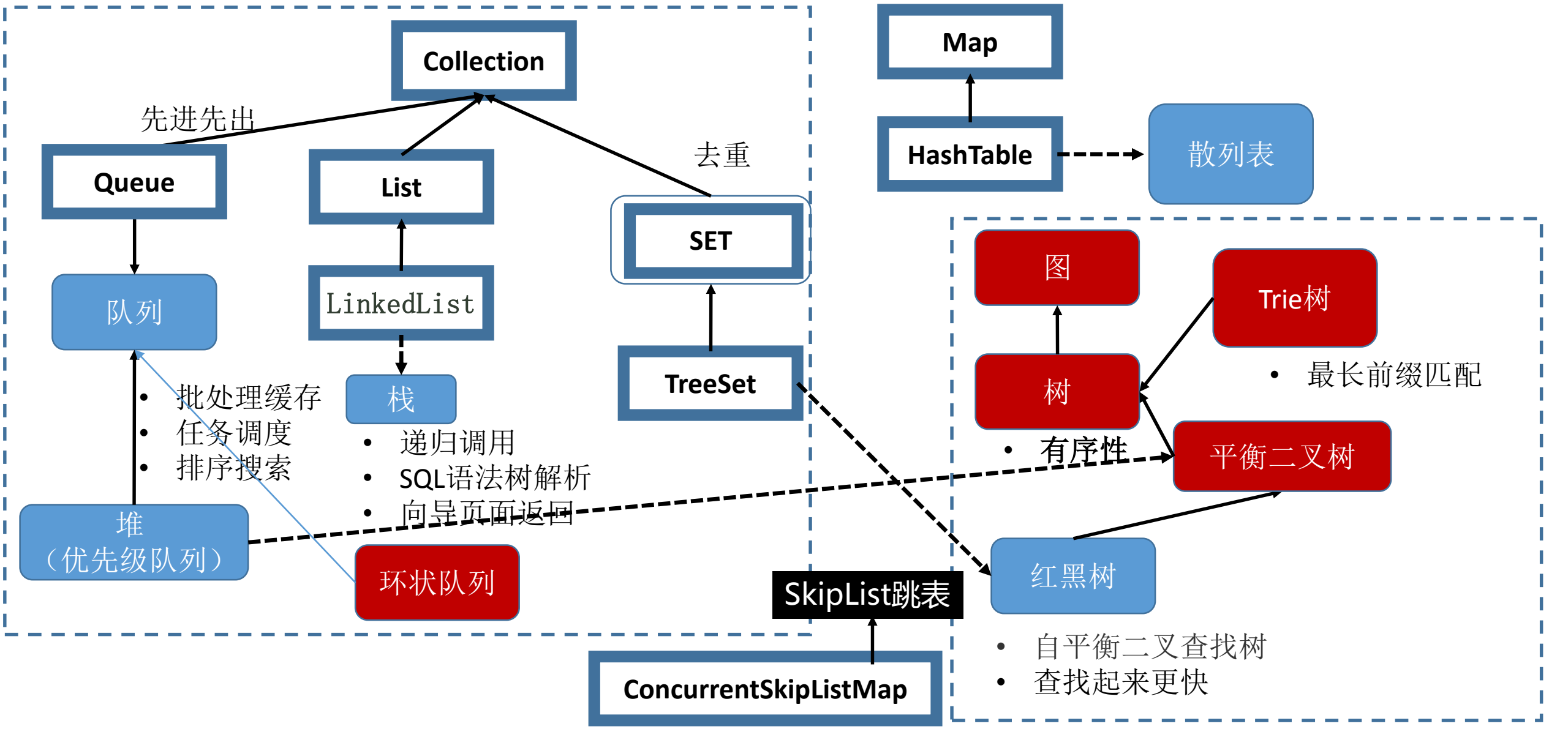
For the type of cache you're after, either use a dedicated cache system (e.g. EHCache) or look at google-collections' MapMaker class; something like

```
new MapMaker().weakValues().makeMap();
```

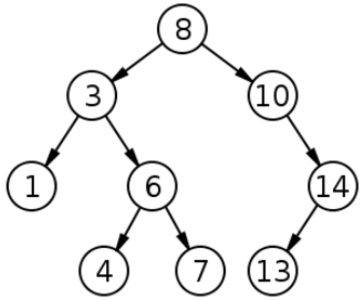
will do what you're after, or if you want to get fancy you can add timed expiration:

```
new MapMaker().weakValues().expiration(5, TimeUnit.MINUTES).makeMap();
```

Java集合框架VS传统数据结构



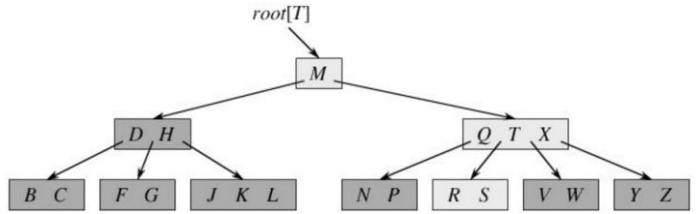
复杂的“树”家族



二叉树

二叉排序树

Trie树



B树每个内结点有多个分支，即多叉平衡查找树

左子树和右子树的高度之差的绝对值不超过1

平衡二叉（排序）树

B树

数据库索引目标

B+树

K维空间的扩展

R树

B*树

空间利用率提升

KD-TREE

AVL树

最先发明的自平衡二叉查找树

红黑树

红黑树并不追求“完全平衡”,降低了对旋转的要求，从而提高了性能

Treap

Treap纪录一个额外的数据，就是优先级。Treap在以关键码构成二叉排序树的同时，还满足堆的性质

是一种高维索引树形数据结构，常用于在大规模的高维数据空间进行最近邻查找(Nearest Neighbor)和近似最近邻查找(Approximate Nearest Neighbor)
已知1000个Point（包含X与Y坐标）结构的数组，给定一个Point，要求查找数组中与之最接近（比如：欧氏距离最短）的点。

对于所有叶子中存储的记录，保存了最小的可以在空间中完全覆盖这些记录所代表的点的矩形

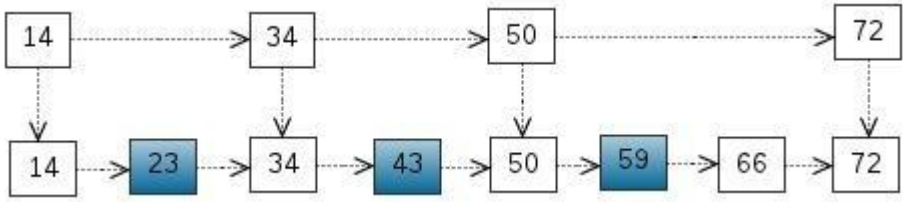
SkipList跳表

红黑树实现起来很复杂，而跳表则接近于红黑树的查找效率，而且实现起来简单方便，它是一种随机化的数据结构，目前开源软件 Redis 和 LevelDB 都有用到它。更重要的，多线程并发情况下，可以实现无锁机制，所以它的性能表现优异。

考虑如下有序链表

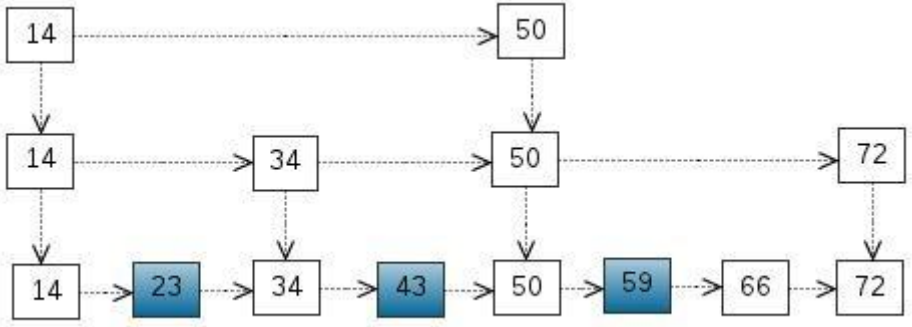
跳表的核心思想，其实也是一种通过“空间来换取时间”的一个算法

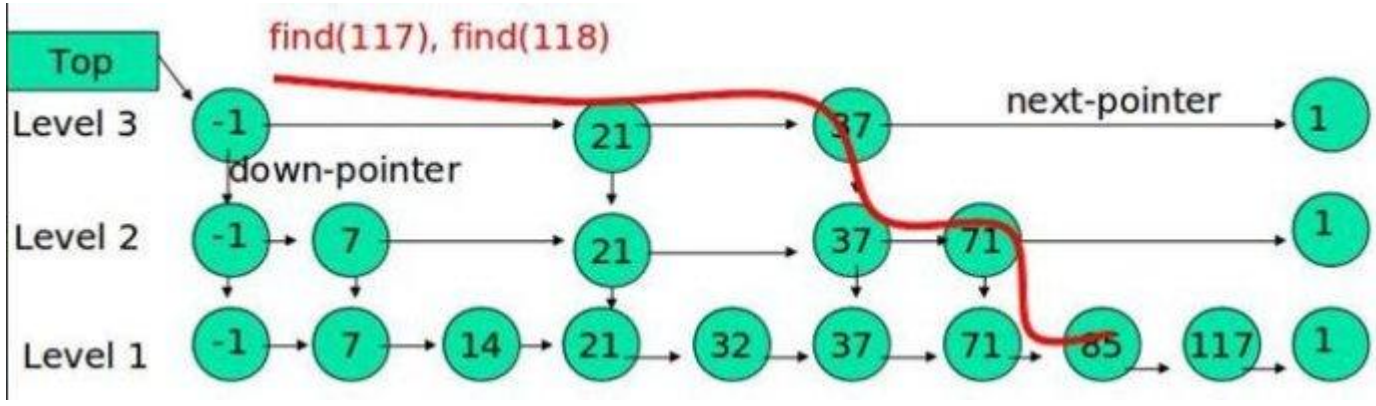
从该有序表中搜索元素 < 23，43，59 > ，需要比较的次数分别为 < 2，4，6 >，总共比较的次数为 2 + 4 + 6 = 12 次。
链表是有序的，但不能使用二分查找。 如何优化？类似二叉搜索树，我们把一些节点提取出来，作为索引。得到如下结构：



这里我们把 < 14，34，50，72 > 提取出来作为一级索引，这样搜索的时候就可以减少比较次数了。

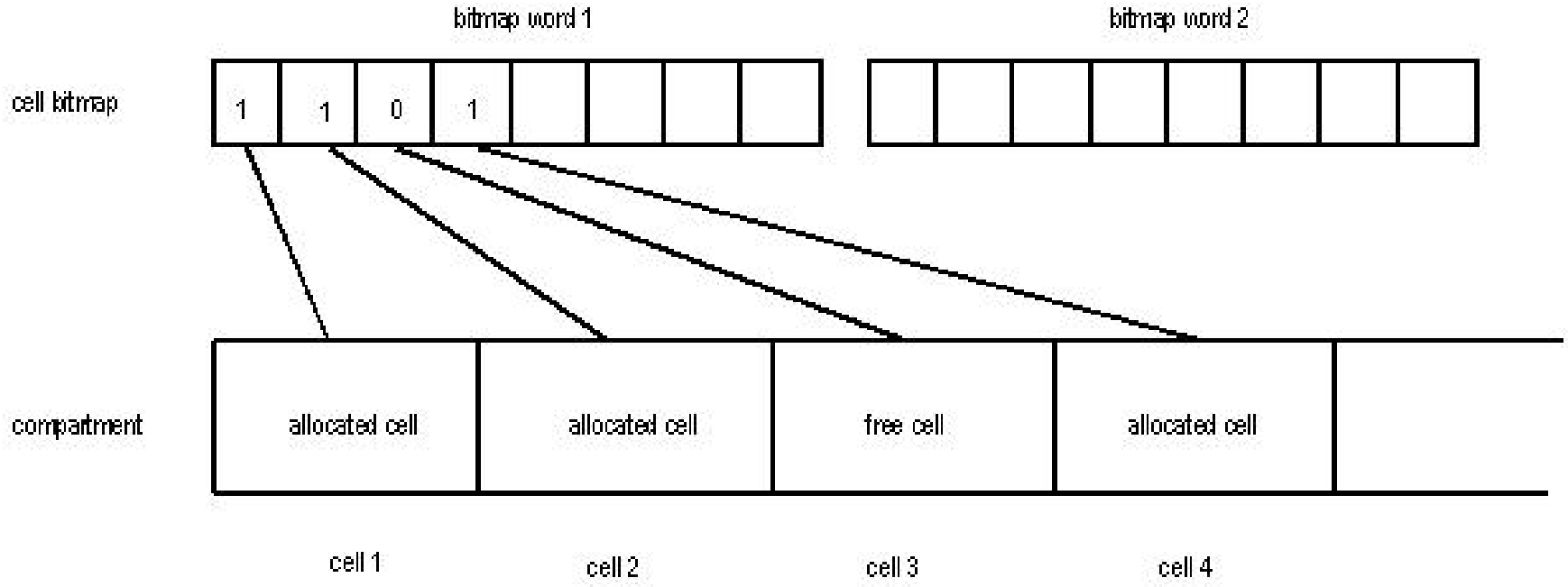
我们还可以再从一级索引提取一些元素出来，作为二级索引，变成如下结构：

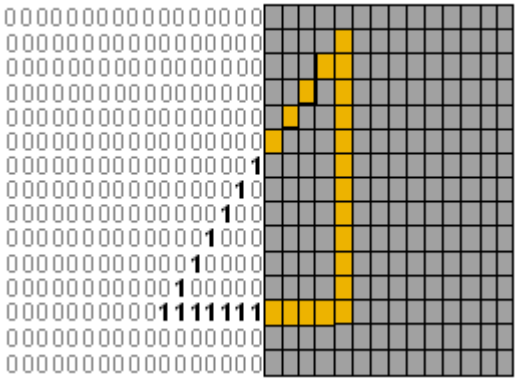




特殊的数据结构之位图

位图数据结构，就是用**一块内存区域的每个比特**表示一个对象的数据结构，叫做bitmap 或者 bitplane，优点是**速度快**，**内存空间占用小**，能表示大范围的数据





内存分配时候bitmap来记录分配情况，很有价值

https://github.com/MyCATApache/Mycat-Lab/tree/master/Mycat-Memory

Java 集合框架最佳实践

永远使用接口去定义你的类型。包括变量声明，参数类型和方法返回类型
底层的集合实际上是空的情况下，返回长度是0的集合或者是数组，不要返回null
hashCode 和 equals对集合中的元素很重要
熟练掌握**Arrays**与**Collections**里的工具类，这里有包括排序，查找，集合类型转换，集合操作（合集，交集等）的实现逻辑
LinkedList插入更新效率高，ArrayList查询效率高，如元素的大小是固定的，而且能事先知道，我们就应该用Array而不是ArrayList，有些集合类允许指定初始容量，初始容量很重要
为了类型安全，可读性和健壮性的原因总是要使用泛型
使用集合的构造器构造新类，从而带来执行效率提升，如TreeMap treeMap = new TreeMap(hashMap);
在遍历一个List的时候，可以使用ListIterator来增加或删除一个元素以及双向遍历
当你需要频繁地查找某个元素是否在一个List或数组中时，最好的方法是先对它进行排序
当你需要元素的有序遍历时，考虑使用“Linked”家族，LinkedHashSet和LinkedHashMap能保证元素按照插入的顺序遍历
使用WeakHashMap来维护短暂使用的数据，entry会在它的key不再被任何线程的执行栈中引用到时，在下一次垃圾回收的周期被GC回收掉
在多线程环境下，优先选择concurrent的家族成员

- **在并发环境下，当List中的元素被频繁遍历而少量修改时，可考虑使用CopyOnWriteArrayList**
- 在并发环境下，如果不需要强一致性的遍历器，可考虑使用ConcurrentHashMap
- 想要在并发环境下使用TreeMap，请考虑ConcurrentSkipListSet和ConcurrentSkipListMap

Java 集合框架最佳实践

删除出现在oth列表中的所有元素

```
public class RemoveAllProfile{
    public static List removeAll(List src,List oth){
        LinkedList result = new LinkedList(src);//大集合用linkedlist LinkedList插入更新效率高
        HashSet othHash = new HashSet(oth);//小集合用hashset
        Iterator iter = result.iterator();//采用Iterator迭代器进行数据的操作
        while(iter.hasNext()){
            if(othHash.contains(iter.next())){
                iter.remove();
            }
        }
        return result;
    }
}
```

龙果学院
roncoo.com

谢谢观看

Leader全栈Java报名群QQ 332702697