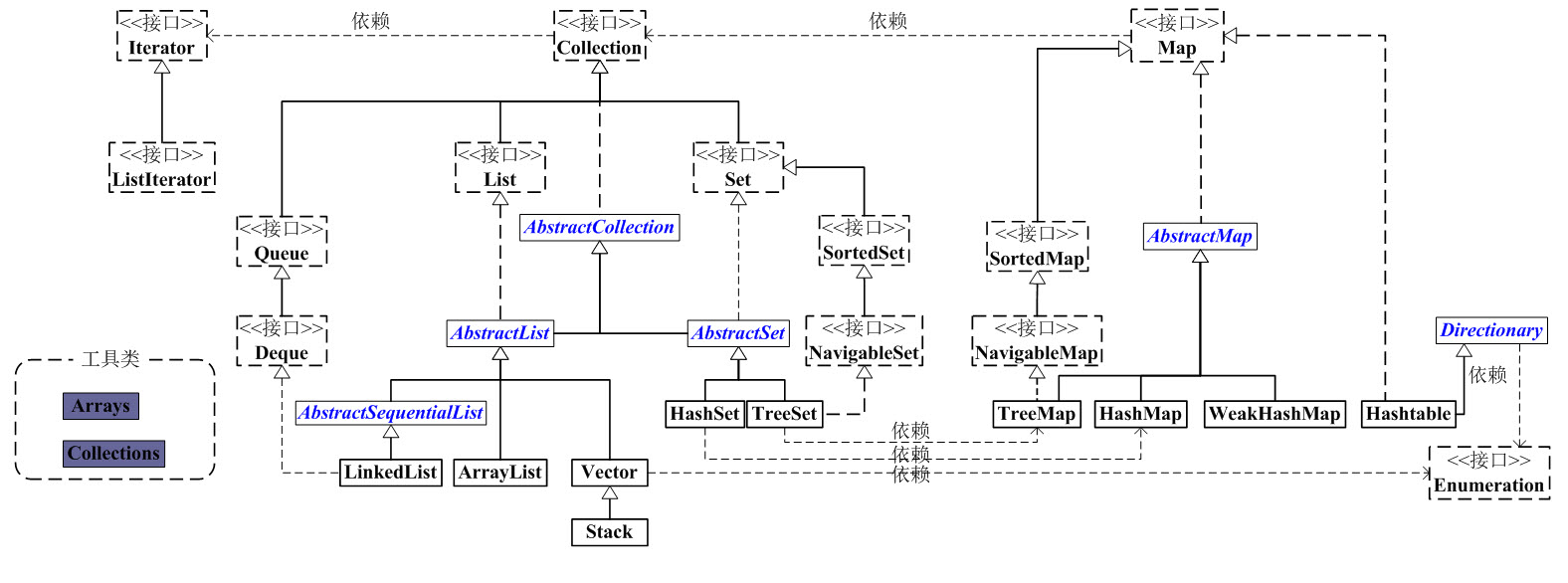
**1. Java容器**

## 1.1总体架构

Java集合是Java的一个工具包，位于java.util，主要的数据结构有集合、链表、队列、栈、数组、映射等。主要包括4个部分：List列表、Set集合、Map映射、Iterator迭代器、Enumeration枚举类、Arrays和Collections。



1、Collection是一个接口，是高度抽象出来的集合、包含了集合的基本操作和属性，包含List和Set两个大分支。

（1）List是一个有序队列，允许有重复的元素。实现类有LinkedList, ArrayList, Vector, Stack。

（2）Set是一个不允许有重复元素的集合。

Set的实现类有HastSet和TreeSet。HashSet依赖于HashMap，它实际上是通过HashMap实现的；TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的

2、Map是一个映射接口，即key-value键值对。Map中的每一个元素包含“一个key”和“key对应的value”。

AbstractMap是个抽象类，它实现了Map接口中的大部分API。而HashMap，TreeMap，WeakHashMap都是继承于AbstractMap。 Hashtable虽然继承于Dictionary，但它实现了Map接口。

3、Iterator.它是遍历集合的工具，即我们通常通过Iterator迭代器来遍历集合。我们说Collection依赖于Iterator，是因为Collection的实现类都要实现iterator()函数，返回一个Iterator对象。ListIterator是专门为遍历List而存在的。

4、Enumeration，它是JDK 1.0引入的抽象类。作用和Iterator一样，也是遍历集合；但是Enumeration的功能要比Iterator少。在上面的框图中，Enumeration只能在Hashtable, Vector, Stack中使用。

5、看Arrays和Collections。它们是操作数组、集合的两个工具类。

## 1.2 List

**1.2.1 ArrayList**

<https://www.cnblogs.com/ITtangtang/p/3948555.html>

1、ArrayList是List接口的可变数组非同步实现，并允许包括null在内的所有元素。

2、底层使用数组实现.

3、该集合是可变长度数组，数组扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量增长大约是其容量的1.5倍，这种操作的代价很高。

4、采用了Fail-Fast机制，面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险

**1.2.2 fail-fast 机制**

<http://www.cnblogs.com/skywang12345/p/3308762.html>

AbstractList(**ArrayList和LinkList都会继承AbstrackList**) 在使用next()和 remove()时，都会执行 checkForComodification()。若 “modCount 不等于 expectedModCount”，则抛出ConcurrentModificationException异常，产生fail-fast事件。

**1.2.3 LinkedList**

<http://www.jianshu.com/p/56c77c517e71>

<http://blog.csdn.net/fighterandknight/article/details/61476335>

**1.3 Map**

<http://www.cnblogs.com/skywang12345/p/3308931.html>

**1.3.1 HashMap**

<http://www.importnew.com/20386.html>

<http://blog.csdn.net/vking_wang/article/details/14166593>

<http://blog.csdn.net/tuke_tuke/article/details/51588156>

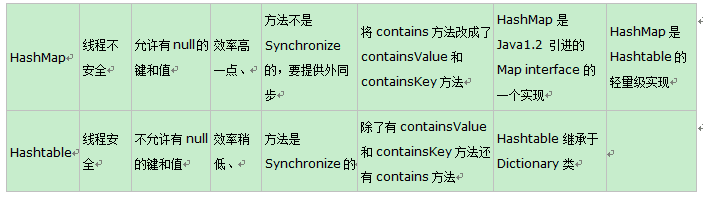
**1.3.2 HashTable**

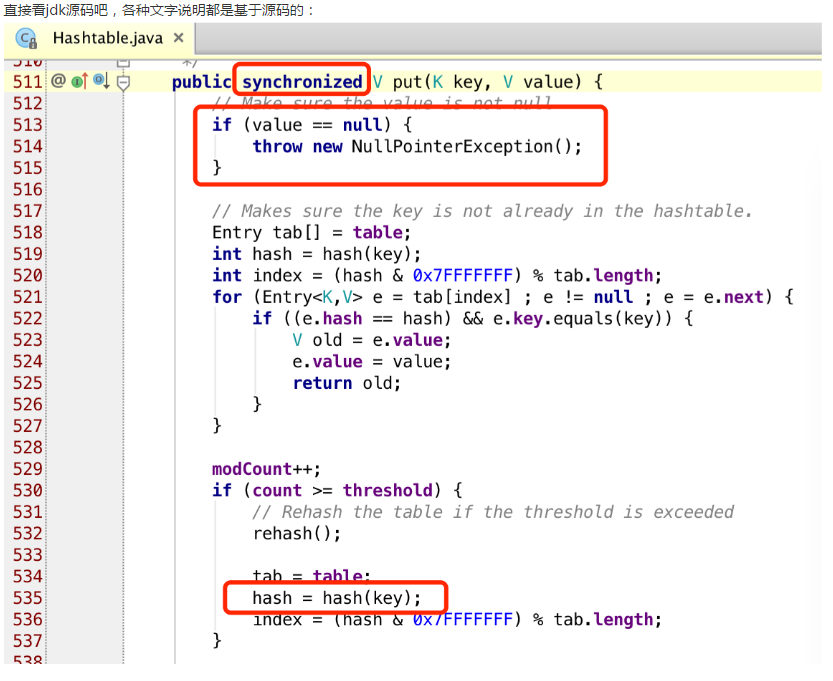
<http://www.cnblogs.com/skywang12345/p/3310887.html>

1、和HashMap一样，Hashtable 也是一个散列表，它存储的内容是键值对(key-value)映射。

2、Hashtable 继承于Dictionary，实现了Map、Cloneable、java.io.Serializable接口。

3、Hashtable 的函数都是同步的（加了synchronized关键字），这意味着它是线程安全的。它的key、value都不可以为null。此外，Hashtable中的映射也不是有序的





**1.3.3 红黑树**

<https://www.cnblogs.com/skywang12345/p/3245399.html>

**1.3.4 TreeMap**

<http://blog.csdn.net/a616413086/article/details/52586006>

**1.3.5 LinkedHashMap**

<https://www.cnblogs.com/whgk/p/6169622.html>

<http://www.cnblogs.com/leesf456/p/5248868.html>

**1.3.6 LinkedHashMap和HashMap的比较**

1、底层实现：linkedHashMap继承了HashMap，即底层也是用哈希表实现的，但是，linkedHashMap还维护了一个双向链表结构，用来维护entry的顺序（详情见<https://www.cnblogs.com/whgk/p/6169622.html>）。

2、HashMap不保证元素的顺序，即添加的HashMap的顺序和储存在HashMap内的顺序是不一致的

3、LinkedHashMap保证元素的结果和添加元素的顺序一致（或者与插入）-----深入一点讲，有两种迭代元素的方式，一种是按照插入元素时的顺序迭代，比如，插入A,B,C，那么迭代也是A,B,C，另一种是按照访问顺序，比如，在迭代前，访问了B，那么迭代的顺序就是A,C,B，比如在迭代前，访问了B，接着又访问了A，那么迭代顺序为C,B,A，比如，在迭代前访问了B，接着又访问了B，然后在访问了A，迭代顺序还是C,B,A。要说明的意思就是不是近期访问的次数最多，就放最后面迭代，而是看迭代前被访问的时间长短决定。

**1.4 Set**

**1.4.1 HashSet**

<http://www.cnblogs.com/skywang12345/p/3311252.html>

1、HashSet的底层，实质上是通过HashMap实现的

2、HashSet添加元素，实际上是调用HashMap的put方法，而且，是把要添加的元素当做HashMap的key，所以，HashSet没有重复的元素，源码如下：其中PRESENT为：private static final Object PRESENT = new Object(); 这也说明，调用HashMap的put方法时，存放在value的值只是相当于占位符，没有确定的值，是一个new Object()。

**public boolean add(E e) {**

**return map.put(e, PRESENT)==null;**

**}**

3、HashSet的元素都存储在HashMap的key上面。

**1.4.2 TreeSet**

底层实现实际上TreeMap

<http://www.cnblogs.com/skywang12345/p/3311268.html>

<https://www.cnblogs.com/ningvsban/archive/2013/05/06/3062535.html>

**1.4.3 LinkedHashSet**

<http://zhangshixi.iteye.com/blog/673319>

1、LinkedHashSet继承了HashSet。

2、LinkedHashSet集合同样是根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起 来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。  
3、LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。

4、LinkedHashSet所有的构造函数，都会调用父类HashSet的下面这个构造方法: 这个方法，是调用LinkedHashMap的构造方法。所以LinkedHashSet底层实际上是LinkedHashMap实现的。

HashSet(**int** initialCapacity, **float** loadFactor, **boolean** dummy) {  
 **map** = **new** LinkedHashMap<>(initialCapacity, loadFactor);  
}

**1.5 PriorityQueue**

<http://blog.csdn.net/hiphopmattshi/article/details/7334487>

<https://www.cnblogs.com/CarpenterLee/p/5488070.html>

**1.6 多线程中的集合类**

保证线程安全的两种方法：

1、前面的集合类都是**非同步的**。比如如果多个线程同时访问一个哈希 set，而其中至少一个线程修改了该 set，那么它必须保持外部同步。这通常是通过对自然封装该set的对象执行同步操作来完成的。如果不存在这样的对象，则应该使用 Collections.synchronizedSet 方法来“包装” set。最好在创建时完成这一操作，以防止对该 set 进行意外的不同步访问，方法如下：

**Set s = Collections.synchronizedSet(new HashSet(...));**

2、或者使用线程安全的集合类。

**1.6.1 List**

<http://blog.csdn.net/yangzl2008/article/details/39456817>

<http://blog.csdn.net/wind5shy/article/details/5396887>

#### 1.6.1.1 Collections.synchronizedList

<http://blog.csdn.net/lzm1340458776/article/details/42455577>

synchronizedList对部分操作加上了synchronized关键字以保证线程安全。但其iterator()操作还不是线程安全的

#### 1.6.1.2 CopyOnWriteArrayList

<https://www.cnblogs.com/dolphin0520/p/3938914.html>

<http://blog.csdn.net/imzoer/article/details/9751591>

**1.6.2 Map**

#### 1.6.2.1 ConcurrentHashMap

<http://www.importnew.com/26049.html>

<https://www.cnblogs.com/zhuawang/p/4779649.html>

#### 1.6.2.2 ConcurrentSkipListMap

**SkipList数据结构**

<https://www.cnblogs.com/xuqiang/archive/2011/05/22/2053516.html>

**ConcurrentSkipListMap**

<http://blog.csdn.net/vickyway/article/details/49507615>

<http://blog.csdn.net/guangcigeyun/article/details/8278349>

**1.6.3 Set**

#### 1.6.3.1 CopyOnWriteArraySet

<http://blog.csdn.net/tanga842428/article/details/52637324>

#### 1.6.3.2 ConcurrentSkipListSet

<https://www.cnblogs.com/leesf456/p/5549820.html>

**1.6.4 Queue**

<http://blog.csdn.net/tingting256/article/details/52488651>

阻塞队列与普通队列的区别在于，当队列是空的时，从队列中获取元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。同样，试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来，如从队列中移除一个或者多个元素，或者完全清空队列.

1.ArrayDeque, （数组双端队列）

2.PriorityQueue, （优先级队列）

3.ConcurrentLinkedQueue, （基于链表的并发队列）

4.DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）

5.ArrayBlockingQueue, （基于数组的并发阻塞队列）

6.LinkedBlockingQueue, （基于链表的FIFO阻塞队列）

7.LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）

8.PriorityBlockingQueue, （带优先级的无界阻塞队列）

9.SynchronousQueue （并发同步阻塞队列）

#### 1.6.4.1 PriorityBlockingQueue

<http://blog.csdn.net/vickyway/article/details/49948701>

offer(E): 入队操作，此处虽然PriorityBlockingQueue是阻塞队列，但是其并没有阻塞的入队方法，因为该队列是无界的，所以入队是不会阻塞的。

#### 1.6.4.2 DelayQueue

<https://www.cnblogs.com/sunzhenchao/p/3515085.html>

**1.6.4.3 ArrayBlockingQueue**

<https://www.cnblogs.com/skywang12345/p/3498652.html>

#### 1.6.6.4 LinkBlockingQueue

<http://www.cnblogs.com/skywang12345/p/java_threads_category.html>

<http://chenjumin.iteye.com/blog/2182322>

# 2．Java反射

1、Java反射让我们可以在运行期间得到类的各种信息

2、spring mvc中封装了BeanUtils这个类，用这个工具类可以对JavaBean的属性进行设置值，详情参考：

<https://www.cnblogs.com/vmax-tam/p/4159985.html>

Java反射参考:

<https://www.cnblogs.com/qifengshi/p/6267511.html>

<http://blog.csdn.net/yongjian1092/article/details/7364451>

1. **public** **class** ReflectionUtils {
2. /\*\*
3. \* 利用反射原理，调用某一个类set类型方法
4. \* @param o
5. \* @param type
6. \* @param methodName
7. \* @param value
8. \*/
9. **public** **static** **void** setMethod(Object o, Class<?> type, String methodName, Object value) {
10. **try** {
11. Class clazz = o.getClass();
12. Method method = clazz.getMethod(methodName, type);
13. method.invoke(o, value);
14. } **catch** (Exception e) {
15. e.printStackTrace();
16. }
17. }
18. /\*\*
19. \* 用反射原理，调用某一个类get类型方法
20. \* @param o
21. \* @param methodName
22. \* @return
23. \*/
24. **public** **static** Object getMethod(Object o, String methodName) {
25. **try** {
26. Class clazz = o.getClass();
27. Method method = clazz.getMethod(methodName);
28. Object value = method.invoke(o);
29. **return** value;
30. } **catch** (Exception e) {
31. e.printStackTrace();
32. **return** **null**;
33. }
34. }
35. }

# 3．Java泛型

<https://www.cnblogs.com/lwbqqyumidi/p/3837629.html>

<http://blog.csdn.net/seu_calvin/article/details/52230032>

<http://blog.csdn.net/s10461/article/details/53941091>

**1、泛型的插除：泛型只在编译阶段有效**

1. **public** **class** GenericTest {
2. **public** **static** **void** main(String[] args) {
3. Box<String> name = **new** Box<String>("corn");
4. Box<Integer> age = **new** Box<Integer>(712);
5. System.out.println("name class:" + name.getClass());      // com.qqyumidi.Box
6. System.out.println("age class:" + age.getClass());        // com.qqyumidi.Box
7. System.out.println(name.getClass() == age.getClass());    // true
8. }
9. }

1.1由此，我们发现，在使用泛型类时，虽然传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是原来的最基本的类型（本实例中为Box），当然，在逻辑上我们可以理解成多个不同的泛型类型。

1.2究其原因，在于Java中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出，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。

**1.3对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。**

**2、泛型的作用**

1) **消除强制类型转换**：如下面demo没有用泛型，在//1处需要进行类型强制转换

1. **public** **class** GenericTest {
3. **public** **static** **void** main(String[] args) {
4. List list = **new** ArrayList();
5. list.add("qqyumidi");
6. list.add("corn");
7. list.add(100);
9. **for** (**int** i = 0; i < list.size(); i++) {
10. String name = (String) list.get(i); // 1
11. System.out.println("name:" + name);
12. }
13. }
14. }

**2）类型安全：**上面代码在//1处会报java.lang.ClassCastException错误，加了泛型的话就不会出现此类错误

Java语言引入泛型的好处是**安全简单**。泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，提高代码的重用率

# 4．Java注解

# 5．IO

## 5.1 transient关键字

**被标记为transient的属性在对象被序列化的时候不会被保存。**

1. **public** **class** UserInfo **implements** Serializable {
2. **private** **static** **final** **long** serialVersionUID = 996890129747019948L;
3. **private** String name;
4. **private** **transient** String psw;
5. **public** UserInfo(String name, String psw) {
6. **this**.name = name;
7. **this**.psw = psw;
8. }
9. **public** String toString() {
10. **return** "name=" + name + ", psw=" + psw;
11. }
12. }
14. **public** **class** TestTransient {
15. **public** **static** **void** main(String[] args) {
16. UserInfo userInfo = **new** UserInfo("张三", "123456");
17. System.out.println(userInfo);
18. **try** {
19. // 序列化，被设置为transient的属性没有被序列化
20. ObjectOutputStream o = **new** ObjectOutputStream(**new** FileOutputStream(
21. "UserInfo.out"));
22. o.writeObject(userInfo);
23. o.close();
24. } **catch** (Exception e) {
25. // TODO: handle exception
26. e.printStackTrace();
27. }
28. **try** {
29. // 重新读取内容
30. ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream(
31. "UserInfo.out"));
32. UserInfo readUserInfo = (UserInfo) in.readObject();
33. //读取后psw的内容为null
34. System.out.println(readUserInfo.toString());
35. } **catch** (Exception e) {
36. // TODO: handle exception
37. e.printStackTrace();
38. }
39. }
40. }

**运行结果为：**

name=张三, psw=123456

name=张三, psw=null

Process finished with exit code 0

# 6. Java虚拟机内存以及Jmeter压测(大模块)

# 7. Java多线程理解与使用(大模块)

# 8．其他

**8.1 Java接口继承接口**

<https://www.cnblogs.com/littlepanpc/p/3616300.html>

**8.2 Java抽象类实现接口**

<http://fishswing.iteye.com/blog/1527166>

**8.3 Java 中比较大小的两种方法**

<http://www.jianshu.com/p/5562c55580e4>

**方法1：实现Comparator接口，并重写compare方法**

1. **public** **class** TestComparator {
2. **public** **static** **void** main(String[] args) {
3. Integer[] a = {3, 5, 1, 4, 0, 8};
4. List<Integer> list = Arrays.asList(a);
5. Collections.sort(list, **new** Com());  //对集合排序
6. Arrays.sort(a, **new** Com());          //对数组排序
7. **for** (Integer integer : a) {
8. System.out.print(integer + " ");
9. }
10. }
11. }
12. **class** Com **implements** Comparator<Integer> {
13. @Override
14. **public** **int** compare(Integer arg0, Integer arg1) {
15. **return** arg0 - arg1;    //升序
16. // return arg1 - arg0;  //降序
17. }
18. }

**方法2： 直接重写Collections.sort**

1. **public** **class** TestComparator1 {
2. **public** **static** **void** main(String[] args) {
3. Integer[] a = {3, 5, 1, 4, 0, 8};
4. List<Integer> list = Arrays.asList(a);
5. Collections.sort(list, ((arg0, arg1) -> {
6. **int** a1 = (**int**) arg0;
7. **int** a2 = (**int**) arg1;
8. **return** a1 - a2;
9. }));
10. **for** (Integer integer : list) {
11. System.out.print(integer + " ");
12. }
13. }
14. }

**8.4 Java回调方法**

<https://www.cnblogs.com/heshuchao/p/5376298.html>