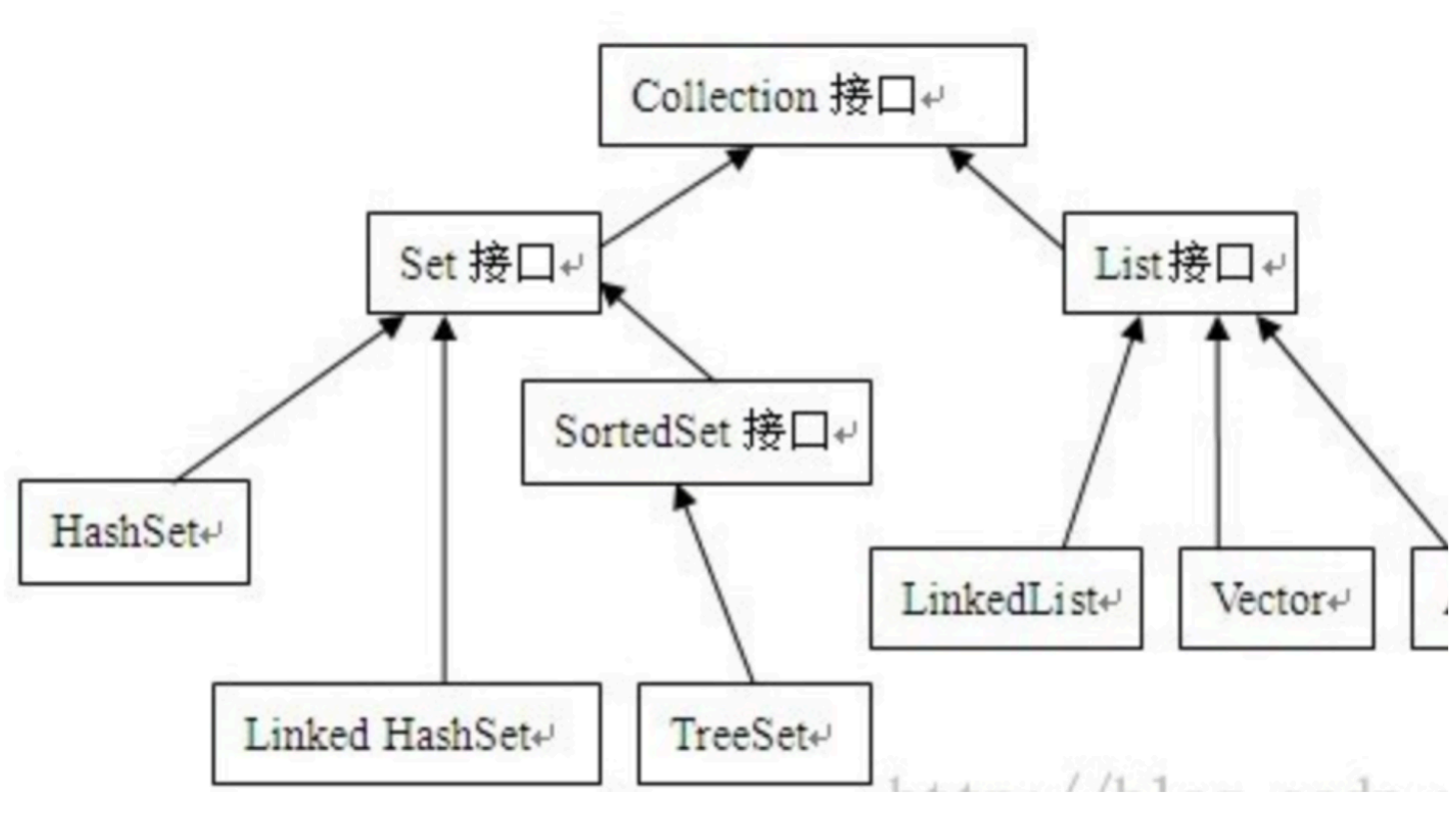
Java集合实现细节

本笔记和智神讲堂第一次总结一样。
Author: Yi
Date: Dec/22/2015
这是第一次测评时，发现的一些知识漏洞，在这里做一个简单的总结给大家，大家如果有什么补充的，不明白的或者勘误的，请帮忙指出来。望大家共同进步，谢谢！！

1: Java Collection 解析

Collection层次结构：



从上图可以看出，Collection接口主要包含Set和List两个主要的接口。
Collection是最基本的集合接口，一个Collection代表一组Object，即Collection的元素。一些Collection允许相同的元素而另一些不行。一些能排序而另一些不行。Java SDK不提供直接继承自Collection的类，Java SDK提供的类都是继承自Collection的“子接口”如List和Set.
所有实现Collection接口的类都必须提供两个标准的构造函数：无参数的构造函数用于创建一个空的Collection，有一个Collection参数的构造函数用于创建一个新的Collection，这个新的Collection与传入的Collection有相同的元素。后一个构造函数允许用户复制一个Collection, 这种复制或者拷贝称为深拷贝，因为生成对象在地址中的内存和参数内存不是一致的，这些情况我们在回溯法中已经见的很多了。
无论对于那种Collection的具体类型，遍历Collection都是采用iterator()方法。

```
Iterator it = collection.iterator(); // 获得一个迭代子
    while(it.hasNext()) {
          Object obj = it.next(); // 得到下一个元素
       }
```

1.1: List接口

List的特征是其元素以线性方式存储，集合中可以存放重复对象.
List主要接口实现类包括：

- ArrayList()
- LinkedList()
- Vector()
- Stack()

ArrayList相当于一个动态数组，也就是数组的长度会随着元素的增减而增减。每个ArrayList实例都会有一个capacity容量变量，在不断添加元素的情况下，capacity会有变化，而对于capacity变化算法不存在唯一定义。size，isEmpty，get，set方法运行时间为常数。但是add方法开销为分摊的常数，添加n个元素需要O(n)的时间。其他的方法运行时间为线性。

LinkedList同样实现了List接口，在ArrayList上还添加了几个特有的操作。remove，insert方法在LinkedList的首部或尾部。这些操作使LinkedList可被用作堆栈（stack），队列（queue）或双向队列（deque）。因此对于LinkedList来说，有很好的插入和删除性能，对于随机访问不如ArrayList.

Vector非常类似ArrayList，但是Vector是同步的。由Vector创建的Iterator，虽然和ArrayList创建的 Iterator是同一接口，但是，因为Vector是同步的，当一个Iterator被创建而且正在被使用，另一个线程改变了Vector的状态（例如，添加或删除了一些元素），这时调用Iterator的方法时将抛出ConcurrentModificationException，因此必须捕获该异常。

Stack继承自Vector，实现一个后进先出的堆栈。Stack提供5个额外的方法使得Vector得以被当作堆栈使用。基本的push和pop方 法，还有peek方法得到栈顶的元素，empty方法测试堆栈是否为空，search方法检测一个元素在堆栈中的位置。Stack刚创建后是空栈。

遍历方法
由于以上四种数据结构都实现了List接口，因此其必满足利用iterator()方法来进行遍历。

```
Iterator it=list.iterator();
while(it.hashNext()){
System.out.println(it.next());
}
```

此外还可以利用for循环来进行遍历。

```
for(int i=0; i<list.size();i++){
System.out.println(list.get(i));
}
```

ArrayList VS LinkedList
ArrayList底层是由数组进行维护的，而LinkedList底层是由Entry对象进行维护的。 LinkedList底层Entry结构：

```
    Entry{
          Entry previous;
          Object element;
          Entry next;
      }
```

其中element就是我们添加的元素,最后将生成的Entry对象加入到链表中. 插入和删除操作时,采用LinkedList好,搜索时,采用ArrayList好.

ArrayList VS Vector
ArrayList不是同步的，而Vector是同步的。在这里同步的意思就是，线程安全。根据Java API Vector中的解释：

> Unlike the new collection implementations, Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.

如果对线程安全没有特别的需求，建议使用ArrayList，而非Vector。

1.2: Set接口

Set是最简单的一种集合。集合中的对象不按特定的方式排序，并且没有重复对象。 因此对于Set来说，添加顺序和取出顺序并不存在必然关系。Set接口主要实现了三个实现类：

- HashSet
- TreeSet
- LinkedHashSet

Set具有与Collection完全一样的接口，因此没有任何额外的功能,不像前面有几个不同的 List 。实际上 Set 就是 Collection ， 只是行为不同。（这是继承与多态思想的典型应用：表现不同的行为）。其次， Set 是一种不包含重复的元素的 Collection ，加入 Set 的元素必须定义 equals() 方法以确保对象的唯一性 （ 即任意的两个元素 e1 和 e2 都有 e1.equals(e2)=false ），与 List 不同的是， Set 接口不保证维护元素的次序。最后， Set 最多有一个 null 元素。很明显， Set 的构造函数有一个约束条件，传入的 Collection 参数不能包含重复的元素。

1.2.1: HashSet and LinkedHashSet

主要来看看HashSet, LinkedHastSet只是在HashSet的基础上，在底层维护了一个双向链表，保证了插入顺序和取出顺序的一致性，即结合了List和Set的性能。如果对插入顺序和取出顺序没有特别的要求，一般建议使用HashSet即可。此外，HashSet底层是由HashMap实现的，因此虽然其接口规范不同，但实质上实现方式是一致的。
HashSet既然是实现了Set接口，那么必定保证其添加的元素的唯一性。对于Java自带的数据类型来说，一般可以直接使用。但如果想添加自定义的数据类型，那么则必须重写equals()和hashCode()两个方法。在这里就涉及到Set是如何维护其添加元素的唯一性的。
既然是HashSet，那么其必定利用了Hash表的特性，即对于每一个存入的对象，通过调用hashCode()方法，会得到一个哈希值，但是这个哈希值并不是唯一的。每一个对象只能得到一个哈希值，但是一个哈希值可以对应到多个对象。因此单纯的hashCode()方法并不能真正意义上维护元素的唯一性，那么就需要一个更加保险但是效率没有hashCode()方法高的方法，也就是equals()方法。其实在真正意义上，只是这个equals()方法维护了插入元素的唯一性，而引入hashCode()方法的原因是因为它更加高效，hashCode()方法你可以单纯返回一个固定值，但这个方法必须存在。所以，当一个元素插入到Set中时候，会进行两步操作：

1. 首先通过hashCode()方法计算其哈希值，如果哈希值所指向的对象并不存在，那么我们认为这个对象不存在，可以安全插入；
2. 若通过hashCode()方法计算的哈希值所指向的对象存在，那么会调用equals()方法。如果返回值是false，证明该对象在集合中不存在，那么再一次通过哈希函数计算哈希值，重新插入。如果哈希值所指向的对象和即将插入对象一致，那么我们认为该对象已经存在于集合中，插入失败。

hashCode()是用来产生哈希值的，而哈希值是用来在散列存储结构中确定对象的存储地址的，像util包中的带hash的集合类都是用这种存储结构：HashMap,HashSet, 他们在将对象存储时（严格说是对象引用），需要确定他们的地址，而HashCode()就是这个用途的，一般都需要重新定义它的，因为默认情况下，由 Object类定义的 hashCode 方法会针对不同的对象返回不同的整数，这一般是通过将该对象的内部地址转换成一个整数来实现的，现在举个例子来说，就拿HashSet来说，在将对象存入其中时，通过被存入对象的hashCode() 来确定对象在HashSet中的存储地址，通过equals()来确定存入的对象是否重复，hashCode()， equals()都需要自己重新定义，因为hashCode()默认前面已经说啦，而equals() 默认是比较的对象引用，你现在想一下，如果你不定义equals()的话，那么同一个类产生的两个内容完全相同的对象都可以存入Set，因为他们是通过默认的equals()来确定的，也就是内存地址，这样就使得HashSet失去了他的意义。

根据上述文字，我们可以得到：

- 1) 默认的equals()方法和hashCode()方法比较的是对象的地址，因此这无法保证插入对象的唯一性。如下例：

```
String str1 = new String("string");
String str2 = new String("string");
```

显然str1和str2在内存中的地址不一样，但我们更加关心的是他们的值，而这个二者是相同的。因此，一般情况下，我们重写equals()方法以保证插入元素拥有不同的值，而hashCode()方法用来确定其内存地址是否一致。

- 2) 所以在自定义对象时，如需维护其插入唯一性，必须重写hashCode()和equals()方法。这个可以作为结论记住。

1.2.2: TreeSet

TreeSet能够对集合中的对象排序,当向TreeSet集合中加入一个对象时,会把它插入到有序的对象序列中。因此插入顺序和取出顺序会有不同，这点是满足Set特性的。
和HashSet一样，TreeSet底层是由TreeMap实现的，虽然接口规范不一致，但实质上实现方式一致。
TreeSet的两种排序方式：

1. 让**元素本身**具有可比较性
   元素本身要实现Comparable接口并实现里面的compareTo方法以保证元素本身具有比较性
2. 让**容器自身**具有可比较性
   当元素本身不具有比较性或者具备的比较性不是所需要的，就在TreeSet建立实例的时候，传入Comparator接口的实现子类的实例。这个Comparator子类必须实现compare方法。

因此为了存入的元素具有可比较性，元素类必须实现comparable的compareTo()方法。
那么何时使用comparable接口，何时使用comparator接口呢。 总结一下几点，如有补充，请指出：

1. 存入元素不具有可比较性，也就是存入元素的类没有实现comparable接口；
2. 存入元素具有可比较性，但是比较的规则并不是我们所需要的；
3. 元素类不能被修改

comparator的使用方法：

```
1. public TreeSet()
无参构造方法，按照自然顺序排序。对于integer按大小，对于string按unicode编码。
2. public TreeSet(Comparator<?super E>comparator)
有参构造方法，按照我们自己定义的规则进行排序。

comparator接口：
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
为了使对象元素按照我们想要的规则排序，必须实现compare()方法。
```

【比较优先级】
当元素和容器本身都具有比较性的时候,容器比较器优先.

总结
HashSet和HashMap具有较好的性能,是Set和Map首选实现类,只有在需要排序的场合,才考虑使用TreeSet和TreeMap. LinkedList 和 ArrayList各有优缺点,如果经常对元素执行插入和删除操作,那么可以用LinkedList,如果经常随机访问元素,那么可以用ArrayList.
如果自定义的类既重写了hashCode和equals，又实现了compareTo，那么这个类的元素既可以存入HashSet容器，也可以存入TreeSet容器。

1.3: Map

请注意，Map并没有实现collection接口，它和collection是两个并行存在的集合。在collection中，我们存放的只是单一的“值”，而在Map中，我们可以存放key-value pair对，也就是相当于一个小型数据库。一个Map中只能存放唯一的key，一个key只能对应着一个唯一的value。
Map接口实现类包括(不限于)：

- HashTable
- HashMap
- LinkedHashMap
- TreeMap

如何遍历一个Map

```
1. 调用keySet()方法，这回返回一个key的迭代器，利用这个迭代器和Map的key－value pair性质，就可以映射得到响应的值；
            Map map = new HashMap();
           Set keySet= map.keySet();
           Irerator iterator = keySet.iterator();
           while(iterator.hasNext()) {
                   Object key = iterator.next();
                   Object value = map.get(key);
           }
2. 调用entrySet()方法，这会返回一个Map.Entry 接口的对象集合。集合中每个对象都是底层Map中一个特定的键-值对。也就是说队于返回对象的任意一个对象，其中都有key和value。
            Map map = new HashMap();
           Irerator iterator = map.entrySet().iterator();
           while(iterator.hasNext()) {
                   Map.Entry entry = iterator.next();
                   Object key = entry.getKey();
                   Object value = entry.getValue();
           }
3. 由于Map底层是用entry结构进行维护的，因此entrySet()遍历方法比keySet()方法更为高效。
```

1.3.1: HashTable

Hashtable 继承 Map 接口，实现一个 key-value 映射的哈希表。任何非空（ non-null ）的对象都可作为key或者value。添加数据使用 put(key, value) ，取出数据使用 get(key) ，这两个基本操作的时间开销为常数。
值得注意的是，由于在HashMap中，key是用来计算哈希码的，因此如果使用自定义类来作为key，必须实现hashCode()和equals()方法，理由在前面介绍hashSet已有阐述。**请注意，必须两个方法同时重写。**
此外，HashTable是**同步**的。

1.3.2: HashMap VS LinkedHashMap

LinkedHashMap就是在HashMap的基础上，在底层还维护了一个双向链表，以保证插入顺序和取出顺序的一致性。如果对于顺序没有特别要求，建议使用HashMap.
HashMap 和 Hashtable 类似，也是基于散列表的实现。不同之处在于 HashMap 是**非同步**的，并且允许 null，即 null value 和 null key 。将 HashMap 视为 Collection 时（ values() 方法可返回 Collection ），插入和查询“键值对”的开销是固定的，但其迭代子操作时间开销和 HashMap 的容量成比例。

1.3.3: TreeMap

TreeMap 的实现使用了红黑树数据结构，也就是一棵自平衡的排序二叉树，这样就可以保证快速检索指定节点。对于 TreeMap 而言，它采用一种被称为“红黑树”的排序二叉树来保存 Map 中每个 Entry —— 每个 Entry 都被当成“红黑树”的一个节点对待. (红黑树有时间再写总结)
正像HashMap和HashTable那样，在TreeMap中，我们排序的对象是key而非value，因此我们在实现TreeMap的插入entry对象时，必须保证key对象类是有可比较性的。因此如果key对象类是自定义的，或者其默认排序规则不是我们所需要的，那么我们必须给key对象实现comparable接口或者让TreeMap容器具有可比较性，也就是构造一个有参TreeMap对象，参数为comparator。这点和之前点TreeSet是一致的。

comparator的使用方法：

```
1. public TreeMap<Key, Value>()
无参构造方法，按照Key自然顺序排序。对于integer按大小，对于string按unicode编码
。
2. public TreeMap<Key, Value>(Comparator<?super E>comparator)
有参构造方法，按照我们自己定义的规则进行排序。

comparator接口：
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
为了使对象元素按照我们想要的规则排序，必须实现compare()方法。
```

总结：

1. **什么是Java集合API**
   Java集合框架API是用来表示和操作集合的统一框架，它包含接口、实现类、以及帮助程序员完成一些编程的算法。简言之，API在上层完成以下几件事：
    - 编程更加省力，提高城程序速度和代码质量
    - 非关联的API提高互操作性
    - 节省学习使用新API成本
    - 节省设计新API的时间
    - 鼓励、促进软件重用
2. 具体来说，有6个集合接口，最基本的是Collection接口，由三个接口Set、List、SortedSet继承，另外两个接口是Map、SortedMap，这两个接口不继承Collection，表示映射而不是真正的集合。
3. **什么是Iterator**
   一些集合类提供了内容遍历的功能，通过java.util.Iterator接口。这些接口允许遍历对象的集合。依次操作每个元素对象。当使用 Iterators时，在获得Iterator的时候包含一个集合快照。通常在遍历一个Iterator的时候不建议修改集合本身。
4. **Iterator与ListIterator有什么区别？**
   Iterator：只能正向遍历集合，适用于获取移除元素。ListIerator：继承Iterator，可以双向列表的遍历，同样支持元素的修改。
5. **什么是HaspMap和Map？**
   Map是接口，Java 集合框架中一部分，用于存储键值对，HashMap是用哈希算法实现Map的类。
6. **HashMap与HashTable有什么区别？对比Hashtable VS HashMap**
   两者都是用key-value方式获取数据。Hashtable是原始集合类之一（也称作遗留类）。HashMap作为新集合框架的一部分在Java2的1.2版本中加入。它们之间有一下区别：
    - HashMap和Hashtable大致是等同的，除了非同步和空值（HashMap允许null值作为key和value，而Hashtable不可以）。
    - HashMap没法保证映射的顺序一直不变，但是作为HashMap的子类LinkedHashMap，如果想要预知的顺序迭代（默认按照插入顺序），你可以很轻易的置换为HashMap，如果使用Hashtable就没那么容易了。
    - HashMap不是同步的，而Hashtable是同步的。
    - 迭代HashMap采用快速失败机制，而Hashtable不是，所以这是设计的考虑点。
7. **在Hashtable上下文中同步是什么意思？**
   同步意味着在一个时间点只能有一个线程可以修改哈希表，任何线程在执行hashtable的更新操作前需要获取对象锁，其他线程等待锁的释放。
8. **什么叫做快速失败特性**
   从高级别层次来说快速失败是一个系统或软件对于其故障做出的响应。一个快速失败系统设计用来即时报告可能会导致失败的任何故障情况，它通常用来停止正常的操作而不是尝试继续做可能有缺陷的工作。当有问题发生时，快速失败系统即时可见地发错错误告警。在Java中，快速失败与iterators有关。如果一个iterator在集合对象上创建了，其它线程欲“结构化”的修改该集合对象，并发修改异常 （ConcurrentModificationException）抛出。因此有快速失败特性的实现类，都线程不同步。
9. **怎样使Hashmap同步？**
   HashMap可以通过Map m = Collections.synchronizedMap（hashMap）来达到同步的效果。
10. **什么时候使用Hashtable，什么时候使用HashMap**
   基本的不同点是Hashtable同步HashMap不是的，所以无论什么时候有多个线程访问相同实例的可能时，就应该使用Hashtable，反之使用HashMap。非线程安全的数据结构能带来更好的性能。
   如果在将来有一种可能—你需要按顺序获得键值对的方案时，HashMap是一个很好的选择，因为有HashMap的一个子类 LinkedHashMap。所以如果你想可预测的按顺序迭代（默认按插入的顺序），你可以很方便用LinkedHashMap替换HashMap。反观要是使用的Hashtable就没那么简单了。同时如果有多个线程访问HashMap，Collections.synchronizedMap（）可以代替，总的来说HashMap更灵活。

Reference

这是我在网上查阅相关资料时的一些博客，大家感兴趣的可以看看。

Java集合类详解
Java集合类TreeSet和TreeMap

博客里面还有一些链接，在这里不一一列出来，大家有时间可以看看。