ArrayList、LinkedList、Vector的异同

我们可以看出ArrayList、LinkedList、Vector都实现了List的接口。

接下来分别看一下三个数据结构的说明：

- 首先是ArrayList

```
public class **ArrayList<E>** extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable
```

List 接口的**大小可变数组**的实现。实现了所有可选列表操作，并**允许包括 null 在内的所有元素**。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。（此类大致上等同于 Vector 类，除了**此类是不同步的**。）

每个 ArrayList 实例都有一个容量。该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向 ArrayList 中不断添加元素，其容量也自动增长。并未指定增长策略的细节，因为这不只是添加元素会带来分摊固定时间开销那样简单。

在添加大量元素前，应用程序可以使用 ensureCapacity 操作来增加 ArrayList 实例的容量。这可以减少递增式再分配的数量。

```
List list = Collections.synchronizedList(new ArrayList(...));
```

此类的 iterator 和 listIterator 方法返回的迭代器是快速失败的：在创建迭代器之后，除非通过迭代器自身的 remove 或 add 方法从结构上对列表进行修改，否则在任何时间以任何方式对列表进行修改，迭代器都会抛出 ConcurrentModificationException。因此，面对并发的修改，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。 注意，迭代器的快速失败行为无法得到保证，因为一般来说，不可能对是否出现不同步并发修改做出任何硬性保证。快速失败迭代器会尽最大努力抛出 ConcurrentModificationException。因此，为提高这类迭代器的正确性而编写一个依赖于此异常的程序是错误的做法：迭代器的快速失败行为应该仅用于检测 bug。

- 然后是LinkedList

```
public class **LinkedList**<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable
```

List 接口的链接列表实现。实现所有可选的列表操作，并且允许所有元素（**包括 null**）。除了实现 List 接口外，LinkedList 类还为在列表的开头及结尾 get、remove 和 insert 元素提供了统一的命名方法。这些操作允许将链接列表用作堆栈、队列或双端队列。

此类实现 Deque 接口，为 add、poll 提供先进先出队列操作，以及其他堆栈和双端队列操作。

所有操作都是按照**双重链接列表**的需要执行的。在列表中编索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端）。

注意，此实现**不是同步**的。如果多个线程同时访问一个链接列表，而其中至少一个线程从结构上修改了该列表，则它必须 保持外部同步。（结构修改指添加或删除一个或多个元素的任何操作；仅设置元素的值不是结构修改。）这一般通过**对自然封装该列表的对象进行同步操作来完成**。如果不存在这样的对象，则应该使用 Collections.synchronizedList 方法来“包装”该列表。最好在创建时完成这一操作，以防止对列表进行意外的不同步访问，如下所示：

```
List list = Collections.synchronizedList(new LinkedList(...));
```

此类的 iterator 和 listIterator 方法返回的迭代器是快速失败 的：在迭代器创建之后，如果从结构上对列表进行修改，除非通过迭代器自身的 remove 或 add 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒将来不确定的时间任意发生不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在不同步的并发修改时，不可能作出任何硬性保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的方式是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

- 最后是Vector

public class **Vector** extends AbstractList implements List, RandomAccess, Cloneable, Serializable

Vector 类可以**实现可增长的对象数组**。与数组一样，它包含可以使用整数索引进行访问的组件。但是，Vector 的大小可以根据需要增大或缩小，以适应创建 Vector 后进行添加或移除项的操作。

每个向量会试图通过维护 capacity 和 capacityIncrement 来优化存储管理。capacity 始终至少应与向量的大小相等；这个值通常比后者大些，因为随着将组件添加到向量中，其存储将按 capacityIncrement 的大小增加存储块。应用程序可以在插入大量组件前增加向量的容量；这样就减少了增加的重分配的量。

由 Vector 的 iterator 和 listIterator 方法所返回的迭代器是快速失败的：如果在迭代器创建后的任意时间从结构上修改了向量（通过迭代器自身的 remove 或 add 方法之外的任何其他方式），则迭代器将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就完全失败，而不是冒着在将来不确定的时间任意发生不确定行为的风险。Vector 的 elements 方法返回的 Enumeration 不是 快速失败的。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在不同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的方式是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测 bug。

从 Java 2 平台 v1.2 开始，此类改进为可以实现 List 接口，使它成为 Java Collections Framework 的成员。与新 collection 实现不同，**Vector 是同步的**。

- 区别

  ArrayList 本质上是一个可改变大小的**数组**.当元素加入时,其大小将会动态地增长.内部的元素可以直接通过get与set方法进行访问.元素顺序存储,**随机访问很快，删除非头尾元素慢，新增元素慢而且费资源** ,较适用于无频繁增删的情况 ,比数组效率低，如果不是需要可变数组，可考虑使用数组 ,**非线程安全**.

  LinkedList 是一个**双链表**,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList. 适用于 ：没有大规模的随机读取，大量的增加/删除操作.**随机访问很慢，增删操作很快**，不耗费多余资源 ,允许null元素,**非线程安全.**

  Vector （类似于ArrayList）但其是**同步**的，开销就比ArrayList要大。如果你的程序本身是线程安全的，那么使用ArrayList是更好的选择。 Vector和ArrayList在更多元素添加进来时会请求更大的空间。Vector每次请求其大小的双倍空间，而ArrayList每次对size增长50%.