# 标记接口（Marker-Interface）

## Serializable

序列化接口，用于标记一个类的对象是可以进行序列化的。所谓序列化，就是讲一个对象的状态（属性）变为可存储或者可传输的过程。

序列化一个类对象是将一个对象的属性进行序列化，因此被static和transient修饰的属性不能被序列化，因为static修饰的属性属于类，不属于对象；transient修饰的属性直接被当做不能序列化；

参考SerializableDemo类

### 注意事项

* 如果一个class实现了Serializable接口，则其子类自动实现序列化，无需显示实现Serializable接口；
* 被static和transient修饰的属性不能序列化；

## Cloneable

一个标记接口，接口中无任何方法。Java中一个类要想提供 clone功能，必须实现Cloneable接口，否则如果调用clone方法会出现CloneNotSupportedException异常。clone方法来源于java.lang.Object类，默认实现是“浅拷贝”，如果要提高“深拷贝”功能，必须override上面所说的clone方法。

clone方法返回的是一个新对象，不是原有对象的引用。

## Comparable

该接口主要用来比较类的偏序关系，如果一个类实现了该接口，必须实现其中的compareTo方法，该方法给用户提供自定义偏序关系的实现。

参考Demo 项目中的CollectionsDemo类，其中在调用sort方法时，会调用具体类的compareTo接口比较两个元素的偏序关系。

## Comparator

该接口也用于比较类的偏序关系，不过类不需要实现该接口，但是需要重新定义一个比较器类，在需要判断偏序关系的地方需要引用该比较器类的实。

实现该接口的类必须实现其中的compare方法。

参考Demo 项目中的CollectionsDemo类。

## RandomAccess

标记接口，主要用于Collections类中的binarySearch静态方法。如果集合类的子类实现了RandomAccess 接口，使用Collections进行二叉查找时则通过下标二叉查找算法，否则使用迭代器的二叉查找算法（下标二叉查找算法的性能比迭代器二叉查找算法性能高）。

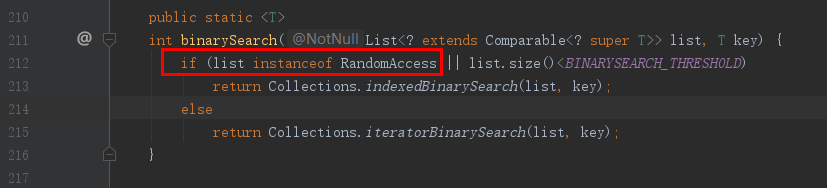


Figure 1 Collections接口二叉查找算法

第212行指出如果list实现了RandomAccess接口或者size小于 BINARYSEARCH\_THRESHOLD（值为5000），都将进入所有的indexBinarySearch，否则进入iteratorBinarySearch。

比如LinkedList虽然没有实现RandomAccess接口，但是链表大小小于5000时仍然会执行indexBinarySearch操作。

有关binarySearch的两个方法使用Demo参考Demo项目里面的CollectionsDemo类。IndexBinarySearch

### IndexBinarySearch

该方法通过下标获取元素，思路很简单。其中重点是：

* List中的元素必须实现Comparable接口
* List必须实现了通过下标获取元素的get方法

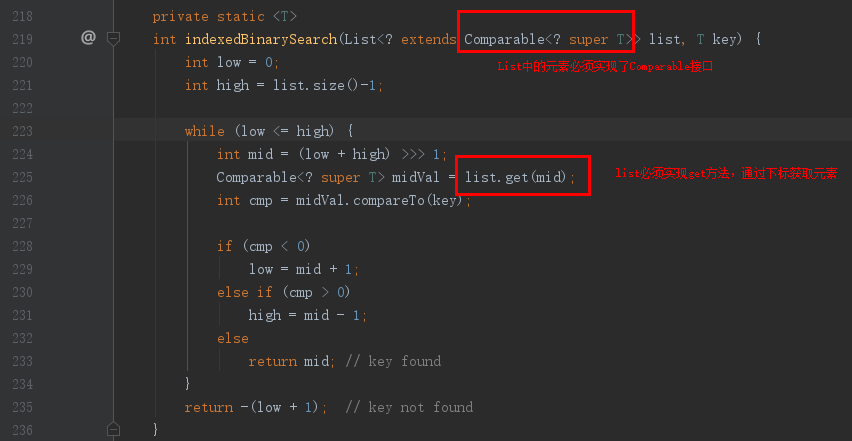


Figure 2 通过下标实现二叉查找算法

### IteratorBinarySearch

该方法的思路同indexBinarySearch，只不过在获取元素是通过iterator获取。

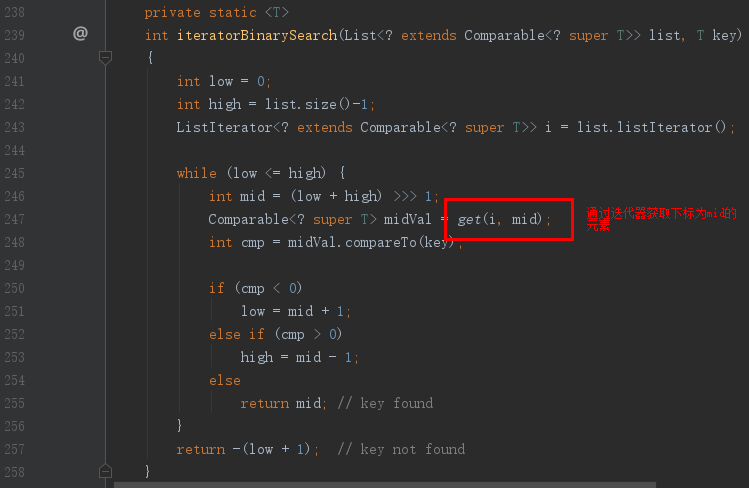


Figure 3 通过迭代器的二叉查找算法

其中的get方法如下：

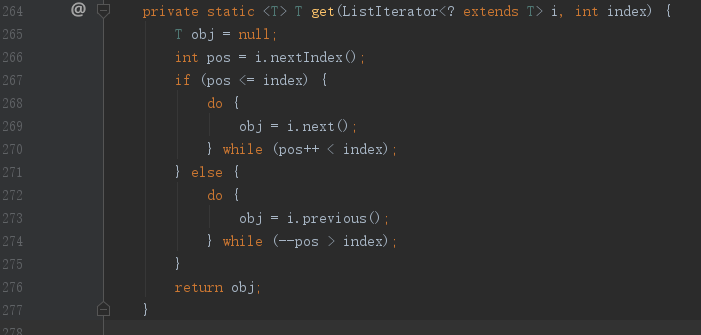


Figure 4 通过迭代器获取元素

# 功能接口

## Iterable

如果一个类实现了该接口，表明该类对象可以作为“for-each”的目标。接口方法iterator返回T类型的迭代器；只有实现了该接口的类才能调用iterator方法获取迭代器。

该接口主要使用在集合类一族，比如list和set结构。Iterable接口作为集合的顶级接口之一，实现了接口的类可以使用如下的循环方式：

For(Object o:list)

## Iterator

迭代器接口，主要包括hashNext和next接口方法，主要用于访问Collection集合的各个元素（访问的方式适合于从头到尾遍历元素）。只提供读操作，不提供写操作。

每一个Collection的实现类中都包含一个迭代器内部类，所有的内部类都实现了Iterator接口。如下表，常见Collection实现类的迭代器：

表格 1 Collection实现类中迭代器描述

|  |  |  |
| --- | --- | --- |
| 类 | 迭代器类 | 描述 |
| ArrayList | Itr | 成员内部类，实现了Iterator接口，包括一个游标cursor（不能人为设置），初始化为0，cursor只能执行递增操作，也就是游标只能往右移动，只提供读操作， |
| ListItr | 成员内部类，继承了Itr类和实现了ListIterator接口，可以指定游标的起始位置（设置cursor的初始值），既可以往左又可以往右移动；既可以读也可以写操作； |
| LinkedList | ListIterator | 成员内部类，提供双端游标功能 |

## ListIterator

该接口和Iterator的区别是，该接口既提高左移，又提供右移，并且提供修改迭代器指向元素的值；

## Spliterator

JDK1.8之后新增的接口。意为可分割的迭代器。重点是要掌握如何分割迭代器的。

### ArrayListSpliterator

几个重点变量：

* list：所需要迭代的list结构的引用；
* index：每个子迭代器的起点下标，执行集合结构的第index个元素
* fence：每个子迭代器的尾部节点下标
* expectedModCount：记录修改次数的变量

1、ArrayList中获取ArrayListSpliterator的方法



Figure ArrayListSpliterator获取方法

通过方法可知，index初始值为0，fence初始值为-1，expectedModCount初始值为0。

2、获取当前自迭代器的fence

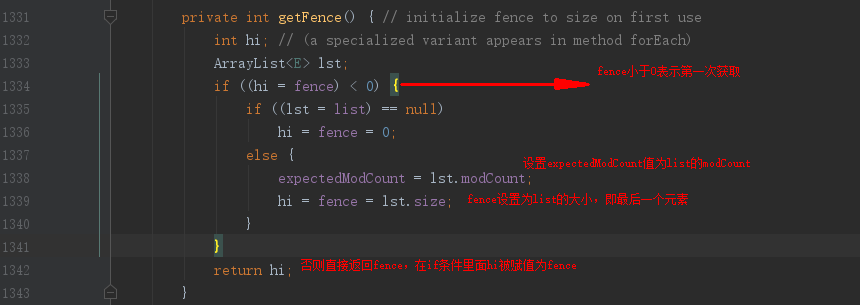


Figure 获取fence方法

3、分割方法

分割方法的思想就是将当前的迭代器一分为二，并返回前半部分的子迭代器（此处有个巧妙的方法就是迭代器一分为二后，返回前半部分子迭代器，但是当前迭代器的index被修改为后半部分的起始地址）。

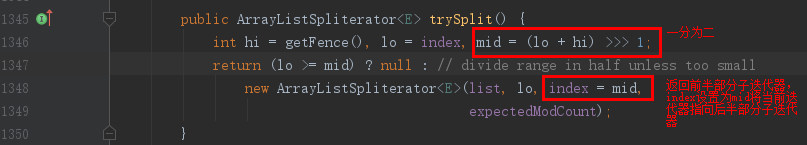


Figure Spliterator一分为二

参考

http://www.cnblogs.com/nevermorewang/p/9368431.html

## Fast-fail机制

Fast-fail机制是集合结构通过迭代器遍历访问时的一种快速错误检测机制。当一个集合结构在通过迭代器访问该集合时，如果集合结构发生了改变（增加或者删除元素，修改元素不会引起集合结构改变），调用迭代器的next等方法时会理解抛出ConcurrentModificationException异常，也就是集合的Fast-Fail机制。

Fast-Fail机制的实现很简单，通过一个modCount计数就可以判断。其实现方式大致为：集合结构中存储一个modCount值，如果该结构增加/删除元素（修改元素时不变）时就对该值+1。如果客户端通过迭代器访问集合结构时会将modCount保存在迭代器的expectedModCount变量中，每次在执行next等操作时，首先会通过expectedModCount和集合结构中的modCount做比较，如果不相等，则抛出异常。

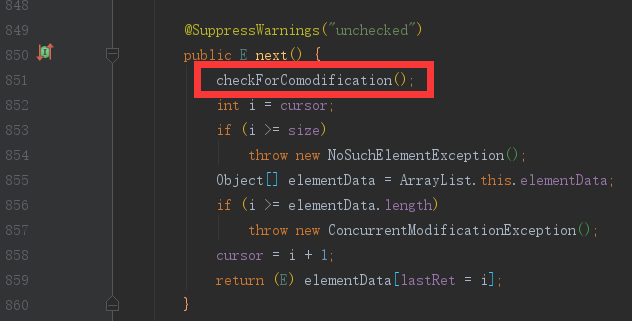


Figure 8 ArrayList迭代器next方法Fast-Fail检测（1）

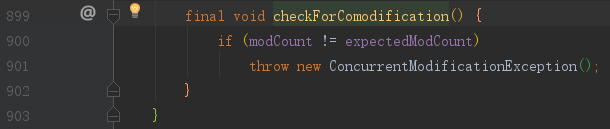


Figure 9 ArrayList迭代器next方法Fast-Fail检测（2）

# 工具类

## Collections