在Java中，我们可以对List集合进行如下几种方式的遍历：

```
List<Integer> list = new ArrayList<>();
list.add(5);
list.add(23);
list.add(42);
for (int i = 0; i < list.size(); i++) {
    System.out.print(list.get(i) + ",");
}

Iterator it = list.iterator();
while (it.hasNext()) {
    System.out.print(it.next() + ",");
}

for (Integer i : list) {
    System.out.print(i + ",");
}
```

第一种就是普通的**for循环**，第二种为**迭代器遍历**，第三种是**for each循环**。后面两种方式涉及到Java中的iterator和iterable对象，接下来我们来看看这两个对象的区别以及如何在自定义类中实现for each循环。

Iterator与Iterable

iterator为Java中的迭代器对象，是能够对List这样的集合进行迭代遍历的底层依赖。而iterable接口里定义了返回iterator的方法，相当于对iterator的封装，同时实现了iterable接口的类可以支持for each循环。

iterator内部细节

jdk中Iterator接口主要方法如下：

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

iterator通过以上两个方法定义了对集合迭代访问的方法，而具体的实现方式依赖于不同的实现类，具体的集合类实现Iterator接口中的方法以实现迭代。

可以发现，在List中并没有实现Iterator接口，而是实现的Iterable接口。进一步观察Iterable接口的源码可以发现其只是返回了一个Iterator对象。

```
public interface Iterable<T> {
  Iterator<T> iterator();
}
```

所以我们可以使用如下方式来对List进行迭代了（通过调用iterator()方法）

```
Iterator it = list.iterator();
while (it.hasNext()) {
    System.out.print(it.next() + ",");
}
```

同时实现了Iterable接口的还可以使用for each循环。

for each原理

其实for each循环内部也是依赖于Iterator迭代器，只不过Java提供的语法糖，Java编译器会将其转化为Iterator迭代器方式遍历。我们对以下for each循环进行反编译：

```
 for (Integer i : list) {
       System.out.println(i);
   }
```

反编译后：

```
Integer i;
for(Iterator iterator = list.iterator(); iterator.hasNext();
System.out.println(i)){
        i = (Integer)iterator.next();
```

```
    }
```

可以看到Java的for each增强循环是通过iterator迭代器方式实现的。

深入探讨Iterable与Iterator关系

有一个问题，为什么不直接将hasNext()，next()方法放在Iterable接口中，其他类直接实现就可以了？

原因是有些集合类可能不止一种遍历方式，实现了Iterable的类可以再实现多个Iterator内部类，例如 `LinkedList` 中的 `ListItr` 和 `DescendingIterator` 两个内部类，就分别实现了双向遍历和逆序遍历。通过返回不同的 `Iterator` 实现不同的遍历方式，这样更加灵活。如果把两个接口合并，就没法返回不同的 `Iterator` 实现类了。ListItr相关源码如下：

```
    public ListIterator<E> listIterator(int index) {
        checkPositionIndex(index);
        return new ListItr(index);
    }

    private class ListItr implements ListIterator<E> {
        ...
        ListItr(int index) {
            // assert isPositionIndex(index);
            next = (index == size) ? null : node(index);
            nextIndex = index;
        }

        public boolean hasNext() {
            return nextIndex < size;
        }
        ...
```

如上所示可以通过调用 `list.listIterator()` 方法返回iterator迭代器（ `list.iterator()` 只是其默认实现）

`DescendingIterator` 源码如下：

```
    public Iterator<E> descendingIterator() {
        return new DescendingIterator();
    }
    private class DescendingIterator implements Iterator<E>     {
        private final ListItr itr = new ListItr(size());
        public boolean hasNext() {
            return itr.hasPrevious();
        }
        public E next() {
            return itr.previous();
        }
        public void remove() {
            itr.remove();
        }
    }
```

同样可以通过`list.descendingIterator()`使用该迭代器。

实现自己的迭代器

我们现在有一个自定义类ArrayMap，现在如果对其进行如下for each遍历：

```
ArrayMap<String, Integer> am = new ArrayMap<>();
am.put("hello", 5);
am.put("syrups", 10);

for (String s: am) {
   System.out.println(s);
}
```

由于我们并没有实现hashNext和next抽象方法，所以无法对其进行遍历。

自定义迭代器类

我们首先自定义一个迭代器类实现hashNext和next方法，并将其作为ArrayMap的内部类，相关代码如下:

```
    public class KeyIterator implements Iterator<K> {
         private int ptr;

         public KeyIterator() {
             ptr = 0;
         }

         @Override
         public boolean hasNext() {
             return (ptr != size);
         }

         @Override
         public K next() {
             K returnItem = keys[ptr];
             ptr += 1;
             return returnItem;
         }
     }
```

可以看到我们在next中指定的遍历规则是根据ArrayMap的key值进行遍历。有了上述迭代器类，我们就可以使用iterator方式在外部对其进行遍历了，遍历代码如下：

```
ArrayMap<String, Integer> am = new ArrayMap<>();
am.put("hello", 5);
am.put("syrups", 10);
ArrayMap.KeyIterator ami = am.new KeyIterator();
while (ami.hasNext()) {
    System.out.println(ami.next());
}
```

如上所示，通过创建KeyIterator对象进行迭代访问（注意外部类创建内部类对象的方式）。

支持for each循环

现在还不能支持for each循环访问，因为我们还没有实现iterable接口，首先在ArrayMap中实现Iterable接口：

```
public class ArrayMap<K, V> implements Iterable<K> {

    private K[] keys;
    private V[] values;
    int size;

    public ArrayMap() {
        keys = (K[]) new Object[100];
        values = (V[]) new Object[100];
        size = 0;
    }
  ....
}
```

然后重写iterator()方法，并在其中返回我们自己的迭代器对象(iterator)

```
    @Override
    public Iterator<K> iterator() {
        return new KeyIterator();
    }
```

注意我们自定义的KeyIterator类必须要实现Iterator接口，否则在iterator()方法中返回的类型不匹配。

总结与感想

（1）学会深入思考，一点点抽丝剥茧，多想想为什么这样实现，很多问题没有自己想象中的那么复杂。

（2）遇到疑惑不放弃，这是提升自己最好的机会，遇到某个疑难的点，解决的过程中会挖掘出很多相关东西。

参考资料：

（1）CS61B

(2) for each实现原理

(3) Iterable与iterator区别