## 概述

Java语言中，提供了一套数据集合框架，其中定义了一些诸如List、Set等抽象数据类型，每个抽象数据类型的各个具体实现，底层又采用了不同的实现方式，比如ArrayList和LinkedList。

除此之外，Java对于数据集合的遍历，也提供了几种不同的方式。开发人员必须要清楚的明白每一种遍历方式的特点、适用场合、以及在不同底层实现上的表现。下面就详细分析一下这一块内容。

## **数据元素是怎样在内存中存放的？**

数据元素在内存中，主要有2种存储方式：

**1、顺序存储，Random Access（或直接存储，Direct Access）：**

这种方式，相邻的数据元素存放于相邻的内存地址中，整块内存地址是连续的。可以根据元素的位置直接计算出内存地址，直接进行读取。读取一个特定位置元素的平均时间复杂度为O(1)。这种数据结构插入和删除时比较麻烦，查询比较方便。正常来说，只有基于数组实现的集合，才有这种特性。Java中以ArrayList为代表。

**2、链式存储，Sequential Access：**

这种方式，是将数据元素放在独立的空间中，在内存中每个元素的内存地址都不要求是相邻的。所以，每个元素中需要保存下一个元素的索引，即下一个元素的内存地址。所以，这种数据结构插入和删除比较方便，但是查找很麻烦，要从第一个开始遍历，因为它无法根据元素的位置直接计算出内存地址，只能按顺序读取元素。读取一个特定位置元素的平均时间复杂度为O(n)。主要以链表为代表。Java中以LinkedList为代表。

## **Java中提供的遍历方式有哪些？**

**1、传统的for循环遍历，基于计数器的：**

遍历者自己在集合外部维护一个计数器，然后依次读取每一个位置的元素，当读取到最后一个元素后，停止。主要就是需要按元素的位置来读取元素。这也是最原始的集合遍历方法。写法为：

**for** (**int** i = 0; i < list.size(); i++) {  
 list.get(i);  
}

**2、迭代器遍历，Iterator：**

迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF给出的定义为：提供一种方法访问一个容器对象中的各个元素，而又不需暴露该对象的内部细节。

从定义可见，迭代器模式是为容器而生。很明显，对容器对象的访问必然涉及到遍历算法。有一种方法是将遍历方法塞到容器对象中去；另一种方法是让使用容器的人自己去实现。这两种都能够解决问题。然而在前一种情况，容器承受了过多的功能，它不仅要负责自己“容器”内的元素维护（添加、删除等等），而且还要提供遍历自身的接口；而且由于遍历状态保存的问题，不能对同一个容器对象同时进行多个遍历。第二种方式倒是省事，却又将容器的内部细节暴露无遗。  
 而迭代器模式的出现，很好的解决了上面两种情况的弊端。Iterator本来是OO的一个设计模式，主要目的就是屏蔽不同数据集合的特点，统一遍历集合的接口。从结构上可以看出，迭代器模式在客户与容器之间加入了迭代器角色。迭代器角色的加入，就可以很好的避免容器内部细节的暴露，而且也使得设计符号“单一职责原则”。

Java作为一个OO语言，自然也在Collections中支持了Iterator模式。

迭代器模式的写法如下：

Iterator iterator = list.iterator();  
**while** (iterator.hasNext()) {  
 iterator.next();  
}

**3、foreach循环遍历：**

foreach内部也是采用了Iterator的方式实现，屏蔽了显式声明的Iterator和计数器。

优点：代码简洁，不易出错。

缺点：只能做简单的遍历，不能在遍历过程中操作（删除、替换）数据集合。

写法如下：

**for** (ElementType element : list) {  
}

## **每个遍历方法的实现原理是什么？**

**1、传统的for循环遍历，基于计数器的：**

遍历者自己在集合外部维护一个计数器，然后依次读取每一个位置的元素，当读取到最后一个元素后，停止。主要就是需要按元素的位置来读取元素，适合于遍历顺序存储的数据结构。

**2、迭代器遍历，Iterator：**

每一个具体实现的数据集合，一般都需要提供相应的Iterator。相比于传统for循环，Iterator取代了显式的遍历计数器。所以基于顺序存储集合的Iterator可以直接按位置访问数据。而基于链式存储集合的Iterator，正常的实现都是需要保存当前遍历的位置，然后根据当前位置来向前或者向后移动指针。

**3、foreach循环遍历：**

根据反编译的字节码可以发现，foreach内部也是采用了Iterator的方式实现，只不过Java编译器帮我们生成了这些代码。

## **各遍历方式对于不同的存储方式，性能如何？**

**1、传统的for循环遍历，基于计数器的：**

因为是基于元素的位置，按位置读取。所以我们可以知道，对于顺序存储，因为读取特定位置元素的平均时间复杂度是O(1)，所以遍历整个集合的平均时间复杂度为O(n)。而对于链式存储，因为读取特定位置元素的平均时间复杂度是O(n)，所以遍历整个集合的平均时间复杂度为O(n2)。

ArrayList按位置读取的代码：直接按元素位置读取。

**transient** Object[] elementData;  
**public** E get(**int** index) {  
 rangeCheck(index);  
 **return** elementData(index);  
}  
E elementData(**int** index) {  
 **return** (E) elementData[index];  
}

**LinkedList按位置读取的代码：每次都需要从第0个元素开始向后读取，虽然它内部做了小小的优化。**

**transient int** size = 0;  
**transient** Node<E> first;  
**transient** Node<E> last;  
**public** E get(**int** index) {  
 checkElementIndex(index);  
 **return** node(index).item;  
}  
Node<E> node(**int** index) {  
 **if** (index < (size >> 1)) { *//查询位置在链表前半部分，从链表头开始查找* Node<E> x = first;  
 **for** (**int** i = 0; i < index; i++)  
 x = x.next;  
 **return** x;  
 } **else** { *//查询位置在链表后半部分，从链表尾开始查找* Node<E> x = last;  
 **for** (**int** i = size - 1; i > index; i--)  
 x = x.prev;  
 **return** x;  
 }  
}

**2、迭代器遍历，Iterator：**

那么对于RandomAccess类型的集合来说，没有太多意义，反而因为一些额外的操作，还会增加额外的运行时间。但是对于Sequential Access的集合来说，就有很重大的意义了，因为Iterator内部维护了当前遍历的位置，所以每次遍历，读取下一个位置并不需要从集合的第一个元素开始查找，只要把指针向后移一位就行了，这样一来，遍历整个集合的时间复杂度就降低为O(n)；以LinkedList的迭代器为例，其内部实现，就是维护当前遍历的位置，然后操作指针移动就可以了：

代码：

**public** E next() {  
 checkForComodification();  
 **if** (!hasNext())  
 **throw new** NoSuchElementException();  
 lastReturned = next;  
 next = next.next;  
 nextIndex++;  
 **return** lastReturned.item;  
}  
**public** E previous() {  
 checkForComodification();  
 **if** (!hasPrevious())  
 **throw new** NoSuchElementException();  
 lastReturned = next = (next == **null**) ? last : next.prev;  
 nextIndex--;  
 **return** lastReturned.item;  
}

**3、foreach循环遍历：**

分析Java字节码可知，foreach内部实现原理是通过Iterator实现的，只不过这个Iterator是Java编译器帮我们生成的，所以我们不需要再手动去编写。但是因为每次都要做类型转换检查，所以花费的时间比Iterator略长，而时间复杂度和Iterator是一样。

## **各遍历方式的适用于什么场合？**

**1、传统的for循环遍历，基于计数器的：**

适用于遍历顺序存储集合，读取性能比较高；不适用于遍历链式存储的集合，时间复杂度太大。

**2、迭代器遍历，Iterator：**

对于顺序存储的数据结构，如果不是太在意时间，推荐选择此方式，毕竟代码更加简洁，也防止了Off-By-One的问题。

链式存储：推荐此种遍历方式，平均时间复杂度降为O(n)

**3、foreach循环遍历：**

foreach只是让代码更加简洁了，但是它有一些缺点，就是遍历过程中不能操作数据集合（删除等），所以有些场合不使用。而且它本身就是基于Iterator实现的，但是由于类型转换的问题，所以会比直接使用Iterator慢一点，但是还好，时间复杂度都是一样的。

## **Java的最佳实践**

Java数据集合框架中，提供了一个RandomAccess接口，该接口没有方法，只是一个标记。通常被List接口的实现使用，用来标记该List的实现是否支持Random Access。一个数据集合实现了该接口，就意味着它支持Random Access，按位置读取元素的平均时间复杂度为O(1)，比如ArrayList。

而没有实现该接口的，就表示不支持Random Access，比如LinkedList。所以看来JDK开发者也是注意到这个问题的，那么推荐的做法就是，如果想要遍历一个List，那么先判断是否支持Random Access，如下：

**if** (list **instanceof** RandomAccess) {  
 *//使用传统的for循环遍历。*} **else** {  
 *//使用Iterator或者foreach。*}