O(n)。
* **适用场景**：适用于读多写少的场景。

```
List<String> arrayList = new ArrayList<>();
arrayList.add("Alice");
arrayList.add("Bob");
```

2.2 `LinkedList`

* **特点**：基于双向链表的实现，插入和删除操作性能较好。
* **存储顺序**：按插入顺序存储。
* **性能**：查找操作的时间复杂度为O(n)，插入和删除操作的时间复杂度为O(1)。
* **适用场景**：适用于写多读少的场景。

```
List<String> linkedList = new LinkedList<>();
linkedList.add("Alice");
linkedList.add("Bob");
```

2.3 `Vector`

* **特点**：古老的实现类，线程安全。
* **存储顺序**：按插入顺序存储。
* **性能**：由于是线程安全的，因此性能较`ArrayList`低。
* **适用场景**：适用于多线程环境，但更推荐使用`Collections.synchronizedList`。

```
List<String> vector = new Vector<>();
vector.add("Alice");
vector.add("Bob");
```

```

##### 2.4 `CopyOnWriteArrayList`

* **特点**：线程安全，基于复制机制的实现。
* **存储顺序**：按插入顺序存储。
* **性能**：读操作性能较好，写操作需要复制整个数组，性能较差。
* **适用场景**：适用于读多写少的多线程环境。

```
List<String> cowList = new CopyOnWriteArrayList<>();
cowLis nchronizedList.add(”Alice”);
synchronizedList.add(”Bob”);
```

需要注意的是，使用`Collections.synchronizedList`包装的`List`在迭代时需要手动同步：

```
synchronized (synchronizedList) {
    Iterator<String> iterator = synchronizedList.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

6. `List`的常见应用场景

* **存储有序数据**：`List`适用于存储有序的数据集合，如待办事项列表、学生名单等。
* **需要随机访问**：`ArrayList`在需要频繁随机访问元素的场景中表现优异，如实现缓存机制。
* **频繁插入和删除**：`LinkedList`在需要频繁插入和删除元素的场景中表现优异，如实现队列或双端队列。
* **多线程环境**：在多线程环境中，`CopyOnWriteArrayList`适用于读多写少的场景，如实现事件监听器列表。
```

7. 总结

Java中的`List`接口及其实现类为我们提供了强大的有序集合存储和操作功能。选择合适的`List`实现类可以有效提升程序的性能和可维护性。在实际开发中，根据具体需求选择`ArrayList`、`LinkedList`、`Vector`或`CopyOnWriteArrayList`，并灵活运用`List`接口提供的方法，能够让我们的代码更加高效和简洁。

希望本文能帮助你更好地理解和使用Java中的`List`。如果有任何问题或建议，欢迎留言讨论！

原文链接: https://juejin.cn/post/7377643248188653579