为什么要使用不可变集合

不可变对象有很多优点，包括：

- 当对象被不可信的库调用时，不可变形式是安全的；
- 不可变对象被多个线程调用时，不存在竞态条件问题
- 不可变集合不需要考虑变化，因此可以节省时间和空间。所有不可变的集合都比它们的可变形式有更好的内存利用率（分析和测试细节）
- 不可变对象因为有固定不变，可以作为常量来安全使用。

创建对象的不可变拷贝是一项很好的防御性编程技巧。Guava为所有JDK标准集合类型和Guava新集合类型都提供了简单易用的不可变版本。

JDK自带的不可变集合

JDK也提供了Collections.unmodifiableXXX方法把集合包装为不可变形式，但我们认为不够好

下面我们先看一个具体实例：

```
1. import java.util.ArrayList;
2. import java.util.Arrays;
3. import java.util.Collections;
4. import java.util.List;
5. import org.junit.Test;
6. 
7. public class ImmutableTest {
8.     @Test
9.     public void testJDKImmutable(){
10.         List<String> list=new ArrayList<String>();
11.         list.add("a");
12.         list.add("b");
13.         list.add("c");
14. 
15.         System.out.println(list);
16. 
17.         List<String> unmodifiableList=Collections.unmodifiableList(list);
18. 
19.         System.out.println(unmodifiableList);
20. 
21.         List<String> unmodifiableList1=Collections.unmodifiableList(Arrays.asList("a","b","c"));
22.         System.out.println(unmodifiableList1);
23. 
24.         String temp=unmodifiableList.get(1);
25.         System.out.println("unmodifiableList [0]: "+temp);
26. 
27.         list.add("baby");
28.         System.out.println("list add a item after list:"+list);
29.         System.out.println("list add a item after unmodifiableList:"+unmodifiableList);
30. 
31.         unmodifiableList1.add("bb");
32.         System.out.println("unmodifiableList add a item after list:"+unmodifiableList1);
33. 
34.         unmodifiableList.add("cc");
35.         System.out.println("unmodifiableList add a item after list:"+unmodifiableList);
36.     }
37. }
```

说明：Collections.unmodifiableList实现的不是真正的不可变集合，当原始集合修改后，不可变集合也发生变化。不可变集合不可以修改集合数据，当会直接抛出不可修改的错误。

总结一下JDK的Collections.unmodifiableXXX方法实现不可变集合的一些问题：

- 笨重而且累赘：不能舒适地用在所有想做防御性拷贝的场景；
- 不安全：要保证没人通过原集合的引用进行修改，返回的集合才是事实上不可变的；
- 低效：包装过的集合仍然保有可变集合的开销，比如并发修改的检查、散列表的额外空间，等等。

Guava中的不可变集合

不可变集合可以用如下多种方式创建：

- copyOf方法，如ImmutableSet.copyOf(set);
- of方法，如ImmutableSet.of(“a”, “b”, “c”)或 ImmutableMap.of(“a”, 1, “b”, 2);
- Builder工具，如

```
1. public static final ImmutableSet<Color> GOOGLE_COLORS =
2.         ImmutableSet.<Color>builder()
3.             .addAll(WEBSAFE_COLORS)
4.             .add(new Color(0, 191, 255))
```

```
5.              .build();
```

比想象中更智能的copyOf

ImmutableXXX.copyOf方法会尝试在安全的时候**避免做拷贝**

```
1. ImmutableSet<String> foobar = ImmutableSet.of("foo", "bar", "baz");
2. thingamajig(foobar);
3.
4. void thingamajig(Collection<String> collection) {
5.     ImmutableList<String> defensiveCopy = ImmutableList.copyOf(collection);
6.     ...
7. }
```

在这段代码中，ImmutableList.copyOf(foobar)会智能地直接返回foobar.asList(),它是一个ImmutableSet的常量时间复杂度的List视图

作为一种探索，ImmutableXXX.copyOf(ImmutableCollection)会试图对如下情况避免线性时间拷贝：

- 在常量时间内使用底层数据结构是可能的——例如，ImmutableSet.copyOf(ImmutableList)就不能在常量时间内完成。
- 不会造成内存泄露——例如，你有个很大的不可变集合ImmutableList<String>

hugeList， ImmutableList.copyOf(hugeList.subList(0, 10))就会显式地拷贝，以免不必要地持有hugeList的引用。

- 不改变语义——所以ImmutableSet.copyOf(myImmutableSortedSet)会显式地拷贝，因为和基于比较器的ImmutableSortedSet相比，Imr

  同语义。

在可能的情况下避免线性拷贝，可以最大限度地减少防御性编程风格所带来的性能开销。

asList视图

所有不可变集合都有一个asList()方法提供ImmutableList视图，来帮助你用列表形式方便地读取集合元素。例如，你可以使用sortedSet.asList(
k个元素。

Guava集合和不可变对应关系

可变集合接口	属于JDK还是Guava	不可变版本
Collection	JDK	ImmutableCollection
List	JDK	ImmutableList
Set	JDK	ImmutableSet
SortedSet/NavigableSet	JDK	ImmutableSortedSet
Map	JDK	ImmutableMap
SortedMap	JDK	ImmutableSortedMap
Multiset	Guava	ImmutableMultiset
SortedMultiset	Guava	ImmutableSortedMult
Multimap	Guava	ImmutableMultimap
ListMultimap	Guava	ImmutableListMultim
SetMultimap	Guava	ImmutableSetMultima
BiMap	Guava	ImmutableBiMap
ClassToInstanceMap	Guava	ImmutableClassToIns
Table	Guava	ImmutableTable