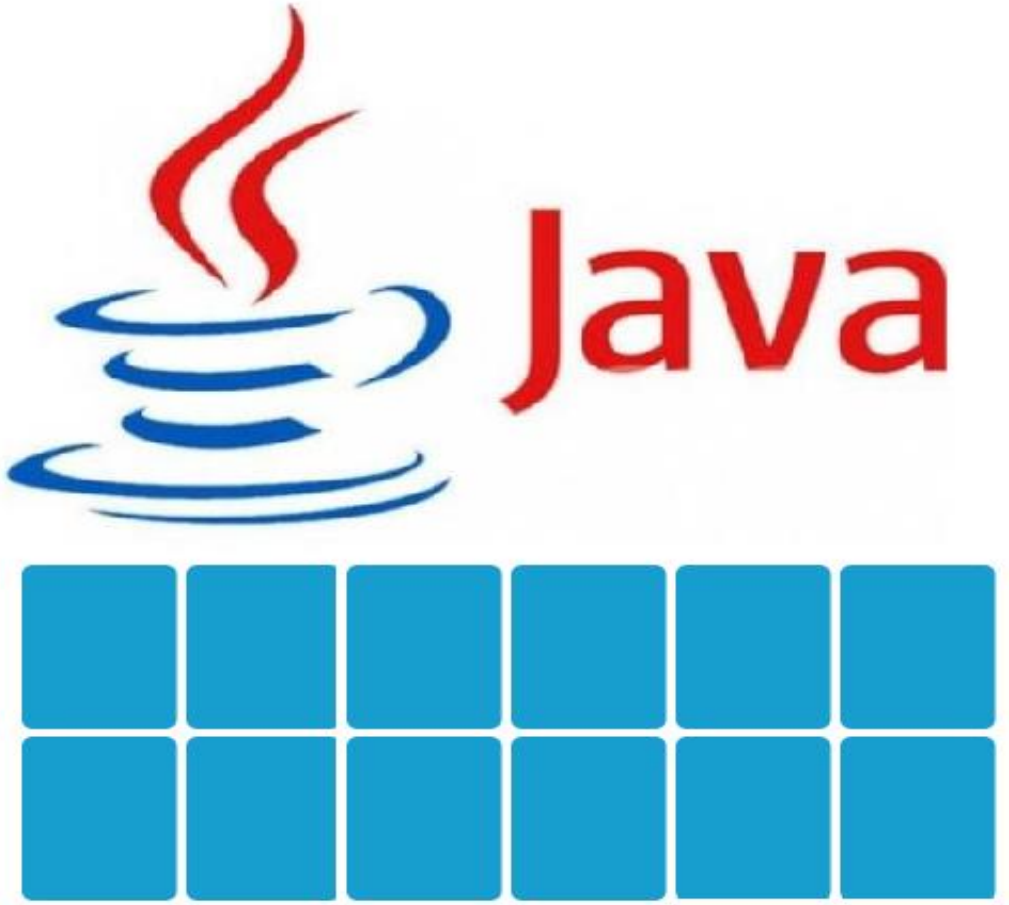


第六章

工具包 - Collections

東北林業大學
NORTHEAST FORESTRY UNIVERSITY

Chapter Objectives

- Collection Frameworks
  - 掌握常用集合的使用
  - 理解面向接口编程思想
- Functional Programming
  - 掌握基于函数式编程的集合Stream APIs的使用
  - 理解函数式编程思想
- Optional
  - 掌握基于函数式编程的Optional容器的使用

Collections

- A collection, sometimes called a container, is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

- 集合，是将许多元素组合成一个单一单元的容器对象
- 集合，可用于存储/检索/操作/传输/聚合数据

Collections Framework

- A **collections framework** is a unified architecture for **representing** and **manipulating** collections. All collections frameworks contain the following:
  - Interfaces: These are abstract data types that represent collections.
  - Implementations: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
  - Algorithms: These are the methods that perform useful computations
- 集合框架，是用于表示和操作集合的体系结构，集合框架应包含
  - 接口(Interfaces)：表示集合的抽象数据类型。使用接口，允许集合独立于其表示的细节进行操作
  - 实现(Implementations)：集合接口的具体实现，包含可重用的数据结构
  - 算法(Algorithms)：对集合执行搜索/排序等操作，是可重用的功能
- 即，Java提供了一套包含，多种集合类型，多种数据结构实现，以及操作处理算法的集合框架，供开发人员直接使用

Iterable

- Implementing this interface allows an object to be the target of the "for-each loop" statement.

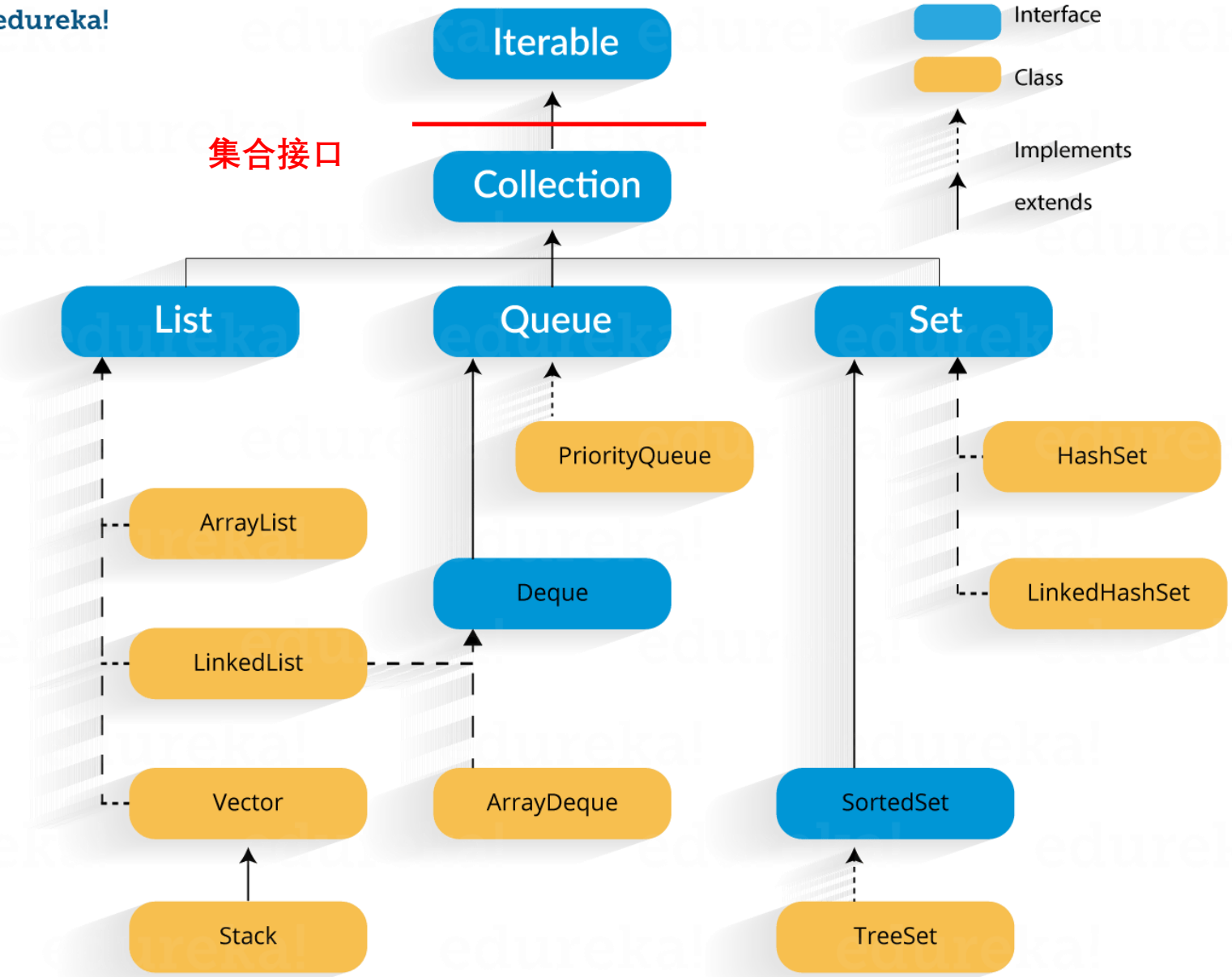
- java.lang.Iterable<T>
- Iterable接口。实现了此接口类的对象，支持ForEach循环语句
- Java8后，添加基于函数式编程的forEach()方法
- **Iterable接口不属于Java集合框架，但**

The Collection Interface

- A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.

- java.util.Collection<E>
- 一个集合，表示一组被称为元素的对象
- Collection接口。用于描述，最具通用性的集合。因此，也包含了最具通用性的集合操作方法
- Collection接口继承自Iterable接口。即，所有集合类型均支持foreach循环语句

Interfaces



- 核心集合接口，包含了多种不同类型的集合
- 以及多种不同数据结构的实现类

- The Collection interface contains methods that perform basic operations
  - boolean add(E e)，向集合添加元素。如调用更改了集合返回true，下同
  - boolean addAll(Collection<? extends E> c)，向集合添加一个集合
  - boolean remove(Object o)，从集合移除中指定元素
  - boolean removeAll(Collection<? extends E> c)，从集合移除包含指定集合
  - void clear()，移除集合全部元素
  - boolean contains(Object o)，判断集合是否包含指定元素
  - boolean containsAll(Collection<? extends E> c)，判断是否包含包含指定集合
  - boolean isEmpty()，判断集合是否包含元素
  - int size()，集合长度
  - T[] toArray(T[] a)，将集合转为指定类型的数组
  - Iterator<E> iterator()，获取迭代器
  - ……

- <E>，泛型。集合并不关心元素的具体类型，因此设计使用泛型
- 创建集合时，必须将泛型具体化为一个引用类型。这有助于编译器的编译时检测，减少运行时错误

- 为什么不声明为Object类型？

The List Interface

- A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection
- java.util.List<E>
- List集合。有序的，允许包含重复元素的集合。除从Collection继承的方法外，提供基于位置索引的操作方法
  - void add(int index, E element)，将指定位置元素后移，添加
  - E set(int index, E element)，替换
  - E get(int index)，获取
  - E remove(int index)，移除
  - ……
- List集合接口基本实现类，即不同的数据结构
  - java.util.ArrayList<E>类，基于对象数组数据结构的实现
  - java.util.LinkedList<E>类，基于双向链表数据结构的实现
  - 具体区别/使用场景/性能，后期讨论

- List集合的声明与创建

```
List<User> users = new ArrayList<>();
```

指定集合类型
List集合接口

元素类型

基于对象数组的实现类

声明List集合类型变量
<>括号中声明集合中元素的类型
必须为引用类型
基于对象数组存储结构ArrayList实现类
创建集合对象

++++++++++++++++++++++++++

```
List<String> strings;
List<int> numbers;
```

Type argument cannot be of primitive type

当试图声明元素类型为
基本数据类型
而非引用类型时
编译错误

++++++++++++++++++++++++++

```
List<String> strings;
List<Integer> numbers;
```

集合中的元素必须为引用类型
当需要使用基本类型时
应声明使用其对应的包装类类型

++++++++++++++++++++++++++

```
List<User> users = new Arr
users.add("BO");
```

Required type: User
Provided: String

当试图向集合中添加
不匹配类型对象时
编译错误

```
List<User> users = new Arra
users.add(new Object());
```

Required type: User

Provided: Object

元素类型的父类对象无法添加

父类具有子类的特性么?

++++++++++++++++++++++++++

```
public class Student extends User {
```

++++++++++++++++++++++++++

```
List<User> users = new ArrayList<>(
users.add(new Student( name: "BO"));
```

正常编译

向集合添加元素
符合继承/多态特性

++++++++++++++++++++++++++

```
List<User> users = new ArrayList<>();
users.add
```

- add(User e)
- add(int index, User element)
- addAll(Collection<? extends Use

- 基于此User类型以及封装的集合对象，完成List集合的学习测试

```
private static final List<User> USERS = create();
private static List<User> create() {
    User user = new User( name: "BO");
    User user2 = new User( name: "SUN");
    User user3 = new User( name: "SUN");
    List<User> users = new ArrayList<>();
    users.add(user);
    users.add(user2);
    users.add(user3);
    return users;
```

```
public class User {
    private String name;
    public User(String name) {
        this.name = name;
```

```
System.out.println(USERS.isEmpty());
System.out.println(USERS.size());
for (User u : USERS) {
    System.out.println(u.getName());
}
```

基本方法
foreach循环语句

```
false
3
BO
SUN
SUN
```

++++++++++++++++++++++++++

基本for循环基于索引获取元素对象
不建议使用

```
for (int i = 0; i < USERS.size(); i++) {
    User user = USERS.get(i);
    System.out.println(user.getName());
}
```

```
BO
SUN
SUN
```

++++++++++++++++++++++++++

```
User user = USERS.get(0);
User user2 = USERS.get(1);
System.out.println("Before: " + user.getName());
System.out.println("Before: " + user2.getName());
for (User u : USERS) {
    u.setName("ZHANG");
}
System.out.println("After: " + user.getName());
System.out.println("After: " + user2.getName());
```

是将对象的引用
保存在集合
因此，集合中的元素
与user/user2引用的是同一个对象

```
Before: BO
Before: SUN
After: ZHANG
After: ZHANG
```

- 因此，集合为逻辑上的容器，容器中仅保存元素对象的引用地址；操作集合中的元素时，实际操作的是元素引用的对象

```
User user = USERS.get(0);
USERS.add(user);
System.out.println(USERS.size());
for (User u : USERS) {
    System.out.println(u);
}
```

```
4
com.example20.collection.User@1b0375b3
com.example20.collection.User@2f7c7260
com.example20.collection.User@2d209079
com.example20.collection.User@1b0375b3
```

声明变量user
将集合第0个元素对象的引用
传递给变量
将对象再次置于集合
由于List集合支持重复的元素对象
因此,集合中对象的引用有序的,保存了2次

- List集合，允许包含重复元素

- 需求：从集合移除指定姓名的用户对象元素

```
for (User u : USERS) {
    if ("SUN".equals(u.getName())) {
        USERS.remove(u);
    }
}
```

试图在遍历集合的同时
删除集合中的元素
异常

```
ain" java.util.ConcurrentModificationException
util.ArrayList$Itr.checkForComodification(ArrayList.
```

```java
for (int i = 0; i < USERS.size(); i++) {
    if ("SUN".equals(USERS.get(i).getName())) {
        USERS.remove(i);
    }
}
for (User u : USERS) {
    System.out.println(u.getName());
}
```

```
BO
SUN
```

- 试图使用基本for循环，基于索引删除元素。连续重复属性元素并没有被移除？思考基本for循环的执行过程

- 在集合for/foreach循环中，试图改变集合的长度(增/删元素)，会产生ConcurrentModificationException异常，或其他错误*

- Java8以前，可通过迭代器实现可移除的遍历
- Java8以后，可基于集合stream流，结合Lambda表达式实现

- List to Array。toArray(T[] a)方法(Collection接口声明)

```
User[] users = USERS.toArray(new User[0]);
for (User u : users) {
    System.out.println(u.getName());
```

创建0长度数组即可
方法内部仅需类型长度自动调整

- Array to List。Arrays工具类提供asList(T... a)方法

```
User[] users = USERS.toArray(new User[0]);
List<User> userList = Arrays.asList(users);
for (User u : userList) {
    System.out.println(u.getName());
```

- asList()方法为适配器模式方法，仅转换了类型，底层仍是数组。因此，执行任何修改集合长度方法(add/remove等)，将抛出异常

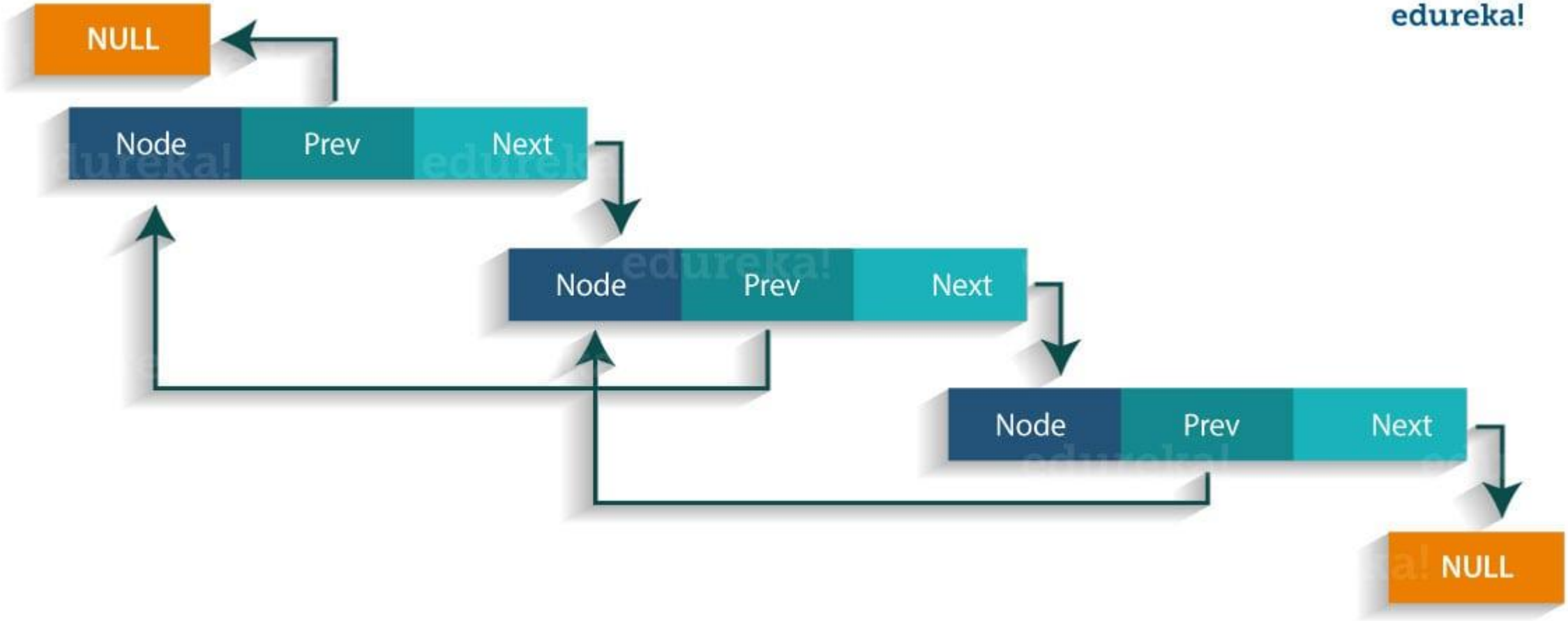
ArrayList

- ArrayList构造函数
  - ArrayList()，创建空List集合。默认创建0个元素的对象数组
  - ArrayList(int initialCapacity)，基于指定长度创建List集合。长度仅初始化集合时使用，后期添加/移除自动容量
  - ArrayList(Collection<? extends E> c)，基于指定集合创建List集合

- add()/size()，方法源码分析

LinkedList

- List集合，基于LinkedList双向链表数据结构的实现
- 会为每个元素创建2个节点，保存前/后元素的地址



Performance of ArrayList & LinkedList

- 测试环境： win10 Pro 1909; I5-6600 4核4线程 3.3GHz; 16GB DDR4; OpenJDK 11.05; 单线程操作
- 分别创建1百万user对象置于ArrayList/LinkedList集合操作

```
 ArrayList创建时间：54363600
LinkedList创建时间：175425400
 ArrayList foreach遍历：8772300
LinkedList foreach遍历：13602700
随机数：849240
 ArrayList基于随机数索引获取元素：6100
LinkedList基于随机数索引获取元素：1476300
 ArrayList基于随机数索引移除元素：79700
LinkedList基于随机数索引移除元素：3335700
 ArrayList删除随机数索引前全部元素：117092997600
LinkedList删除随机数索引前全部元素：17802400
```

反复移除第一个元素
反复创建集合

```
for (int i = 0; i < RANDOM; i++) {
    users.remove( i: 0);
```

- ArrayList，可快速基于索引访问元素对象；其底层使用Arrays.copyOf()方法实现对象数组的增删，性能损失较小

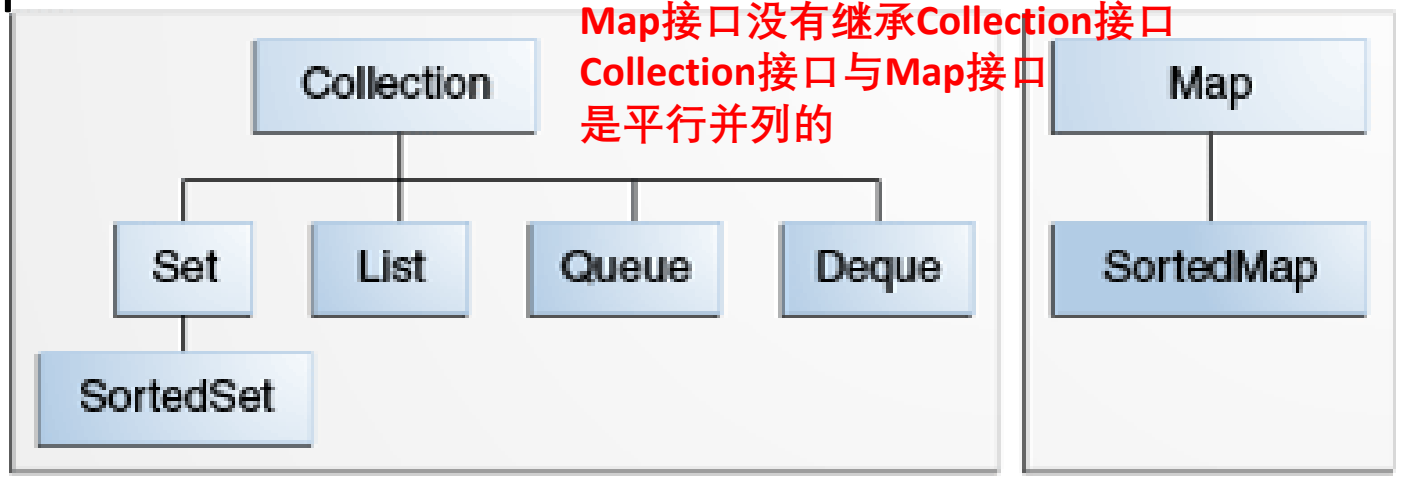
- LinkedList，当需要极其频繁的在集合头部添加元素时，效率较高。但需要为每一个元素创建两个节点对象，基于索引位置的访问需要线性时间(Positional access requires linear-time)，整体性能开销较大

- 绝大多数情况下，使用ArrayList，或不可变集合(后期讨论)即可

- 集合的多线程同步，后期讨论

The Map Interface

- A Map is an object that **maps keys to values**. A map cannot contain duplicate keys: Each key can map to at most one value
- A Map is **not a true** Collection



- java.util.Map<K,V>
- Map，用于存放键值对(key-value)
- Map不是集合
- 通过key值，保存其对应的value值
- 通过Key值获取对其对应的value值操作。例如
  - 手机通讯录中，以姓名为key，手机号码为value，通过姓名找到对应的手机号，拨打电话

- Map中key必须是唯一的，且每个key只能对应一个value
- 但不同key，可以对应同一个value
- 添加key-value时，如果key已经存在，则后一个覆盖前一个
- 通过key的hash值，判断key是否相同*
- 支持以基本数据类型为key/value；支持以任何类型对象为key/value
- 基本实现类
  - java.util.HashMap<K, V>，查询效率与内存占用最平衡，非线程安全
  - java.util.TreeMap <K, V>/HashTable<K, V>

- 常用操作方法
  - V put(K key, V value)，保存键值对
  - V get(K key)，基于key获取对应的value，如果value不存在，返回null
  - default V getOrDefault(Object key, V defaultValue)，获取对应的value，没有则使用默认值
  - V remove(Object key)
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
  - int size()
  - boolean isEmpty()
  - putAll()/clear()
  - ……..
- Map没有，基于index索引的操作
- Map没有，继承Iterable接口，不支持foreach语句遍历*

```
Map<String, String> map = new HashMap<>();
map.put("BO", "956");
map.put("SUN", "925");
System.out.println(map.size());
System.out.println(map.get("BO"));
```

```
2
956
```

定义Map变量，声明key类型与value类型
使用HashMap实现类实现
通过put()方法添加元素
获取元素个数
基于key获取其映射的value

```
Map<String, String> map = new HashM
map.put("BO", "956");
map.put("BO", "925");
System.out.println(map.size());
System.out.println(map.get("BO"));
```

```
1
925
```

Map中不能有相同的key
Key相同时，后赋值会覆盖前面的值

```
public class User {
    public static final String HAERBIN = "哈尔滨"
    public static final String BEIJING = "北京";
    private int id;
    private String name;
    private String city;
      ++++++++++++++++++++++++++++
private static final List<User> USERS = create();
private static List<User> create() {
    User u = new User( id: 1,  name: "BO", User.HAERBIN);
    User u1 = new User( id: 2,  name: "SUN", User.BEIJING);
    User u2 = new User( id: 3,  name: "ZHANG", User.BEIJING)
    User u3 = new User( id: 4,  name: "LIU", User.HAERBIN);
    List<User> users = new ArrayList<>();
    users.add(u); users.add(u1);
    users.add(u2); users.add(u3);
    return users;
```

测试用List集合
2城4人

- 需求：将居民ID为key，居民本身为value

声明Map类型变量的
Key类型；value类型

```
Map<Integer, User> uMap = new HashMap<>()
for (User u : USERS) {
    uMap.put(u.getId(), u);
}
System.out.println(uMap.size());
User u = uMap.get(1);
System.out.println(u.getCity());
```

循环集合
取出元素对象
取出对象ID属性
将元素自己置位value

从Map中取出
指定key对应的value

```
4
哈尔滨
```

++++++++++++++++++++++++++++++

```
User u = uMap.get(100);
System.out.println(u.getCity());
```

如果指定的key不存在
则返回空引用null

```
ad "main" java.lang.NullPointerException
nple22.listsetmap.mapTest.getMapBaseOp3(
nple22.listsetmap.mapTest.main(mapTest.jav
```

- 需求：以城市名为key，以下对应的居民集合为value，分组置于Map

```
List<User> hList = new ArrayList
hList.add(USERS.get(0));
hList.add(USERS.get(3));
List<User> bList = new ArrayList
bList.add(USERS.get(1));
bList.add(USERS.get(2));
```

++++++++++++++++++++++

以城市名称为key
以居民集合为value

```
Map<String, List<User>> uMap = new HashMap<>()
uMap.put(User.HAERBIN, hList);
uMap.put(User.BEIJING, bList);
System.out.println(uMap.size());
for (User u : uMap.get(User.BEIJING)) {
    System.out.println(u.getName());
}
```

从Map分组中取出指定居民

```
2
SUN
ZHANG
```

- 需求：遍历User List集合，以城市名称为key，对应的居民集合为value自动分组。即，遍历的同时基于不确定的城市名称，创建对应集合，再分组

```
String n = new String( original: "BO");
String n2 = new String( original: "BO");
System.out.println(n.equals(n2));
System.out.println(n == n2);

Map<String, String> map = new HashMap<
map.put(n,  v: "956");
map.put(n2,  v: "925");
System.out.println(map.size());
```

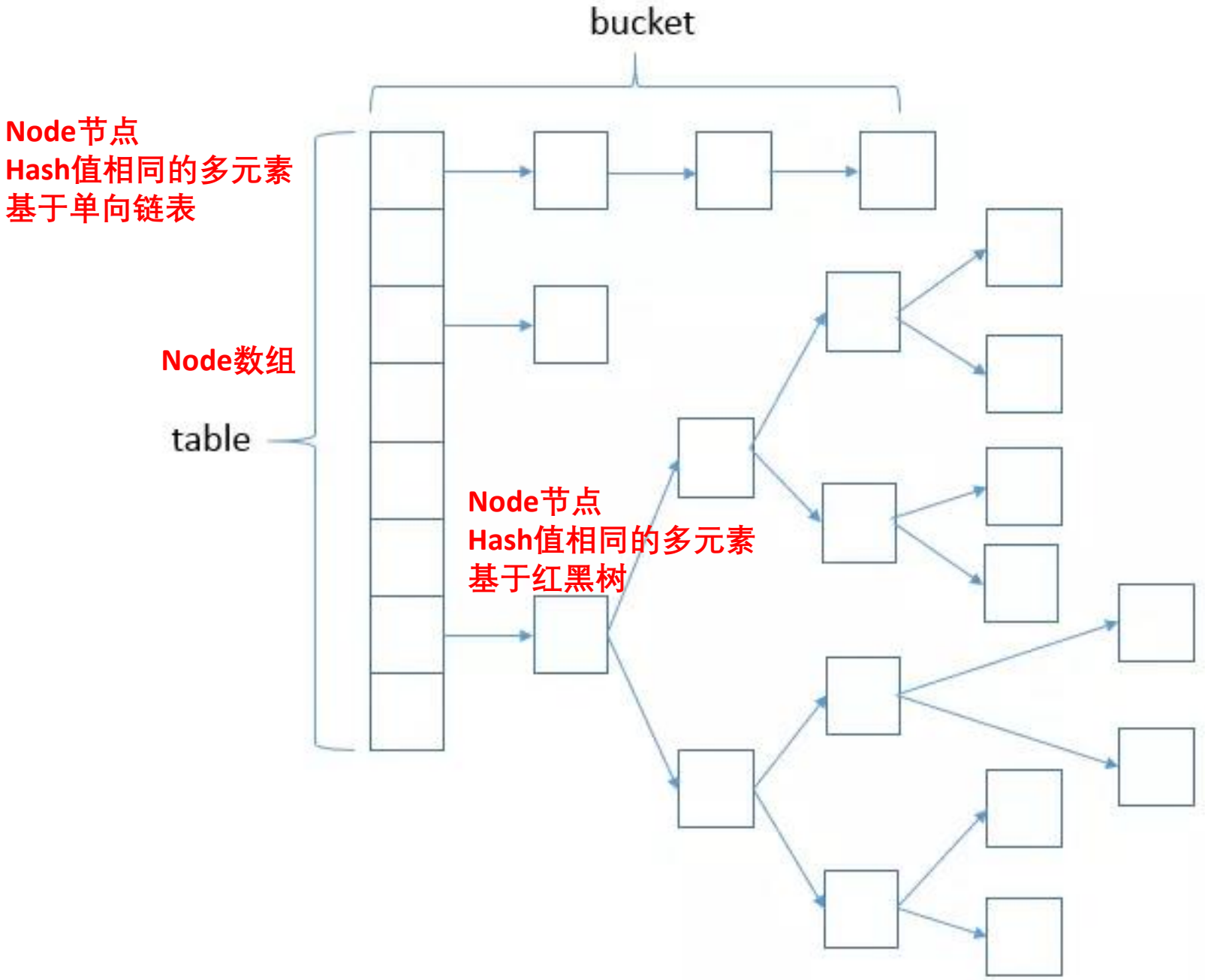
2个String对象
但，equals判断相对

```
true
false
1
```

- String重写了hashCode()/equals()方法，因此直接基于字符串值，而非对象的hash值比较

HashMap

- 基于hashCode()方法获计算key的hash值
- 创建Node数组，基于加载因子扩容，平衡内存占用与执行效率
- 创建Node对象，基于key的hash值+算法，计算Node对象在数组中的索引。Node对象中，封装Key/value对象，相同位置以及存在Node对象
  - 减少数量小于等于6个，基于单向链表保存
  - 增加数量大于等于8个，基于红黑树保存
  - 数量改变时，转换数据结构
- 获取时，基于key的hash值+算法，直接在Node数组获取对应的Node对象，基于具体数据结构(单向链表/红黑树)进一步获取value对象


bucket
Node节点
Hash值相同的多元素
基于单向链表
Node数组
table
Node节点
Hash值相同的多元素
基于红黑树

The Set Interface

- A **Set** is a Collection that cannot contain **duplicate** elements. The Set interface contains **only** methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- java.util.Set<E>
- Set集合，不包含重复元素(数学中集合的抽象)
- Set接口，只包含继承自Collection方法，并添加禁止重复元素的限制
- 基本实现类
  - java.util.HashSet<E>，元素无序(底层基于HashMap确定元素是否重复)
  - java.util.LinkedHashSet<E>，元素有序
  - java.util.TreeSet <E>，元素有序
- 无论使用有序/无序实现，均无基于索引的操作方法

```
Set<User> users = new HashSet<>();
User user = new User( name: "BO");
User user1 = new User( name: "SUN");
users.add(user);
users.add(user1);
System.out.println(users.size());
users.add(user);
for (User u : users) {
    System.out.println(u.getName());
```

```
2
SUN
BO
```

试图添加重复元素
没有改变集合

遍历时元素无序输出

++++++++++++++++++++++++++

```
Set<User> users = new HashSet<>()
users.get
Use  m getClass()                    Clas
Use  Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in t
```

无序
则无基于索引的获取元素方法

```
Set<User> users = new HashSet<>();
List<User> users2 = new Arra
User user = new User
User user2 = new Use
```

```
ArrayList<>() (java.util)
ArrayList<>(int initialCapacity) (java.…
ArrayList<>(Collection<? extends User> c)
```

```
Set<User> uSet = new HashSet<>();
List<User> uList = new ArrayList<>(uSet);
```

List/Set集合均提供
基于Collection接口类型的构造函数
可以将2种集合相互转换

- Set集合适合描述，逻辑上不能重复的对象集合
  - 扑克牌
  - 不重复的关系，人，事物等

```java
private static final List<User> USERS = create();
private static List<User> create() {
    User user = new User( name: "BO");
    User user2 = new User( name: "SUN");
    User user3 = new User( name: "SUN");
    List<User> users = new ArrayList<>();
    users.add(user);
    users.add(user2);
    users.add(user3);
    return users;
```

++++++++++++++++++++++++++++++++

```java
for (User u : USERS) {
    if ("BO".equals(u.getName())) {
        USERS.remove(u);
    }
```

试图删移除第一个元素

```
java.util.ConcurrentModificationException
.ArrayList$Itr.checkForComodification(Array
```

++++++++++++++++++++++++++++++++

Why?

```java
for (int i = 0; i < USERS.size(); i++) {
    if ("SUN".equals(USERS.get(i).getName())) {
        USERS.remove(i);
    }
}
```

Idea提示操作存疑

Suspicious 'List.remove()' in the loop

?

BO

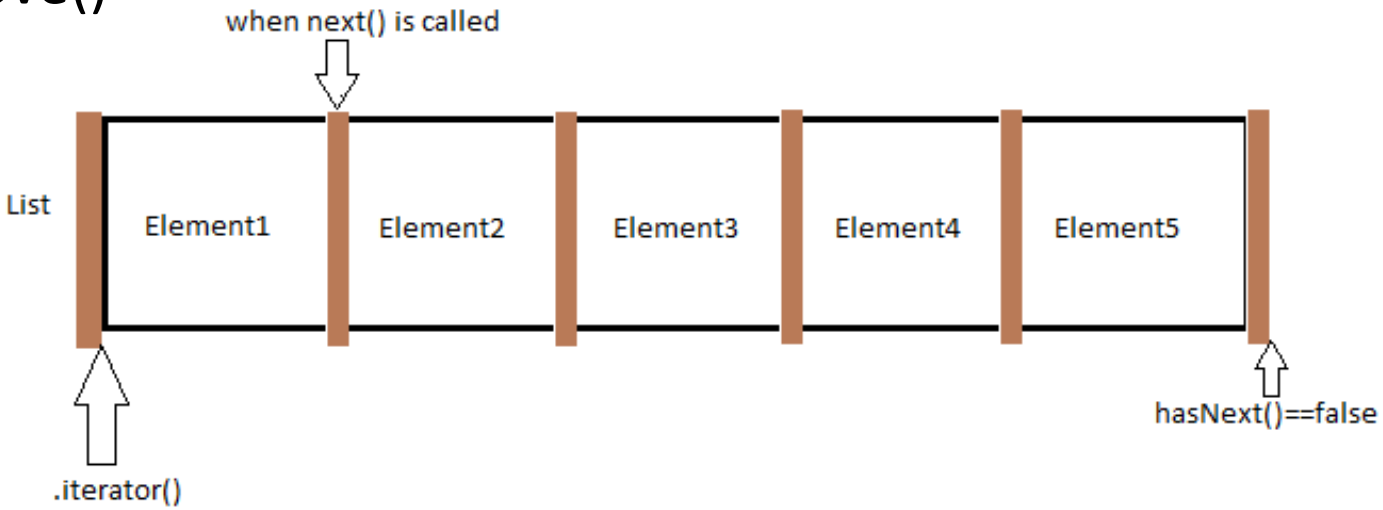
SUN

Iterators

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

- java.util.Iterator<E>
- Iterator接口。迭代器，允许遍历集合，并根据需求选择性地从集合中移除元素
- 不同的集合类型，的不同数据结构的实现类，有不同的迭代器实现，但仅需面向Iterator接口完成遍历与移除
- Iterator<E> iterator()方法，Collection接口方法，获取集合对象的迭代器

- Iterator通过移动游标遍历集合。初始时，游标位于第一个元素前
  - boolean hasNext()，Iterator中是否有下一个元素
  - E next()，向后移动游标，同时返回游标指向的元素
  - void remove()




```
Iterator<User> iUsers = USERS.iterator();
while (iUsers.hasNext()) {
    User u = iUsers.next();
    System.out.println(u.getName());
}
```

- void remove()，从基础集合移除当前游标指向的元素

```
Iterator<User> iUsers = USERS.iterator();
while (iUsers.hasNext()) {
    User u = iUsers.next();
    if ("SUN".equals(u.getName())) {
        iUsers.remove();
    }
```

获取iterator迭代器对象
循环判断
获取元素对象，判断
删除迭代器当前游标指向的元素

Benefits of the Java Collections Framework

- Java集合框架，提供了多种类型集合，以及高性能/高质量的数据结构实现
- 集合之间可以方便的实现相互转换
- 各种类型的集合接口与具体实现分离，将对集合的操作(方法)，与集合底层的具体实现方式，解耦
- 从而使开发者将时间精力致力于改善程序本身的质量和性能
  - 减少编程工作量
  - 提高程序速度和质量
  - 减少学习和使用新API的工作量
  - ……
- 例如，避免了直接操作对象数组带来的同步/异步/性能/效率/异常/接口设计等一系列繁琐问题
- Java集合框架，是典型的面向接口编程的体现

- 例如，向List集合中添加元素，无论底层基于arrays/linked，均是add()方法，切换存储实现而无需修改逻辑代码
- 即，在使用Java集合框架时，完全屏蔽了底层集合实现的具体细节与方法，使程序员更专注与基于集合实现应用的业务逻辑，而非关注于集合本身

- Java集合框架，是典型的面向接口编程的体现

第六章

工具包 – Collection Stream APIs

東北林業大學
NORTHEAST FORESTRY UNIVERSITY

Functional Programming

- Functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- 函数式编程，是一种构建程序结构的编程范式。是一种与面向对象程序设计，完全不同的应用程序设计思想
- 在函数式编程中，函数的输出应且仅应依赖于函数的本身。即，函数的执行，不应依赖于函数外部数据的状态(闭包)
- 函数式编程与面向对象编程是不同场景下，分析设计应用的思考方式，无优劣之分

Chapter Objectives

```
int x;
public int multiply() {
    return x * x;
}
```

方法的执行结果
不仅仅依赖于方法本身
且，依赖于外部的属性
面向对象的设计

++++++++++++++++++++++++++

```
public int multiply(int x) {
    return x * x;
```

方法的执行结果
仅依赖于方法参数及方法本身
函数式的设计

Lambda Expressions

- Java lambda expressions are Java's first step into functional programming. A Java lambda expression is thus a function which can be created without belonging to any class. A Java lambda expression can be passed around as if it was an object and executed on demand.

- 通过函数式接口，设计一个函数(约束函数的参数/返回类型的抽象方法)(后期讨论)
- 通过Lambda表达式，定义一个已经设计好的函数(函数式接口的实现)
- 函数，可以像对象一样传递，在需要时执行。并且具有
  - 匿名。无，修饰符/返回类型/名称(Think More, Write Less)
  - 函数。不属于任何类，有参数列表，函数体以及返回值
  - 传递。可作为参数传递给方法，或作为变量的值

- Lambda表达式语法

(arg1, arg2) -> expression
(arg1, arg2) -> {body}

- 箭头，函数参数列表与表达式/函数主体的分隔符
- Lambda表达式可包含0或多个参数
- 参数列表，当参数为空时，需声明空括号；当只有一个参数时，可省略括号；参数类型可省略，编译器自动完成类型推导

```
(int num1, int num2) -> n
```

等效

```
(num1, num2) -> n
```

```
num1 ->
```

1个参数，省略括号

Chapter Objectives

- Lambda 表达式的函数体，可包含0或多条语句
- 函数体，只有一条语句的表达式，可省略{}号；包含一条以上语句，必须包含在括号中(代码块)；返回类型必须匹配；没有可不声明返回值

```
() -> { return 3.1415 };
```

无参数有返回值的函数
基于语句返回结果，如果使用语句必须使用{}

```
() -> System.out.println("Hello World");
```

无参数无返回值的函数
仅有一条语句，可省略{}

```
(s) -> {
    System.out.println(s);
};
```

有参数无返回值的函数
基于语句，也可省略{}

```
(x, y) -> {
    System.out.println("Result: ");
    return x + y;
};
```

有2参数有返回值的函数
多条语句，必须使用{}
必须声明返回结果

Chapter Objectives

测试类型

```
public class Apple {
    public enum Color{
        RED, GREEN
    }
    private int id;
    private Color color;
    private int weight;
    public Apple(int id, Color color, int weight) {
        this.id = id;
        this.color = color;
        this.weight = weight;
    }
```

++++++++++++++++++++++++++

测试集合

```
private static final List<Apple> APPLES = create();
private static List<Apple> create() {
    Apple a = new Apple( id: 1, Apple.Color.RED, weight: 200);
    Apple a2 = new Apple( id: 2, Apple.Color.GREEN, weight: 250);
    Apple a3 = new Apple( id: 3, Apple.Color.RED, weight: 260);
    Apple a4 = new Apple( id: 4, Apple.Color.GREEN, weight: 230);
    List<Apple> apples = new ArrayList<>();
    apples.add(a); apples.add(a2);
    apples.add(a3); apples.add(a4);
    return apples;
```

返回指定颜色的所有苹果？

Chapter Objectives

SQL语句
没有循环实现
没有筛选实现
即，无需实现查询
仅需表达与描述

```
"SELECT a FROM apple a WHERE a.color=?"
```

SQL代码语句描述
查询苹果表中
指定颜色的
所有苹果

++++++++++++++++++++++++++

Java代码
无法描述问题本身

```
private static List<Apple> getRedApples(String color
    List<Apple> reds = new ArrayList<>();
    for (Apple a : apples) {
        if (color.equals(a.getColor())) {
            reds.add(a);
        }
    }
    System.out.println(reds.size());
    return reds;
```

Java代码语句描述
创建装指定颜色苹果的集合
循环遍历源苹果集合
判断本次遍历苹果颜色
将其加入集合

Processing Data with Streams

- Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- java.util.stream.Stream<T>接口
- 鉴于Java对集合操作的复杂性，Java8中引入Stream API，用于操作处理集合中的元素
- 集合是存储元素对象的容器
- 而Stream(集合流)，并不是存储元素的数据结构，而是操作集合元素的管道
- 商品(元素)从仓库(集合)取出，经商品包装流水线(集合流)操作，达到出厂销售的结果
- Stream操作的是集合中的元素，而非集合本身。因此，将创建新集合聚合Stream操作的结果，而不影响源集合结构

- Stream仅会对流过的元素操作一次(流走了)(与Iterator的游标相似)
- 因此，必须生成一个新Stream才能继续操作
- Stream上的操作会被延迟处理，即针对一个stream的多次操作会被优化后执行，从而提高执行效率
- Stream提供了一系列操作集合元素的函数，供开发者直接使用

- Collection接口中，通过stream()方法获取当前集合的Stream对象

- Intermediate Operations，中间操作
- Terminal Operations，终止操作

Terminal Operations

- Terminal Operations。终止操作，终止stream操作处理，消费stream操作产生的结果
  - collect()：聚合在stream中间操作的结果
  - forEach()：迭代stream的每个元素
  - ……
- Collectors(java.util.stream.Collectors)类，用于操作聚合结果的工具类
  - groupingBy()/mapping()
  - toList()/toSet()/toMap()
  - ……
- forEach()方法，Java8 Iterable接口提供的新遍历方法。是方法，不是for-each-loop循环

```
for (Apple a : APPLES) {
    System.out.println(a.getWeight());
}
```

原
基于foreach语句

++++++++++++++++++++++++++++++++++++

集合
每一次迭代元素

```
APPLES.forEach(a -> {
    System.out.println(a.getWeight());
});
```

新
基于forEach()方法
以及Lambda描述的函数

++++++++++++++++++++++++++++++++++++

```
APPLES.forEach(a -> System.out.println(a.getWeight()));
```

单条语句
可进一步简化省略{}

Intermediate Operations

- Intermediate Operations。中间操作，对集合中元素的执行的具体操作
  - Stream filter()：基于参数选择stream中的元素，过滤
  - Stream map()：基于stream操作映射为新的类型，映射
  - Stream sorted()：排序stream中的元素，排序
  - Long count()：获取stream中元素个数，计数
  - ……

- 中间操作执行后，将结果置入一个新Stream，从而允许基于新Stream实现后续操作，形成基于Stream的操作链

- Stream<T> filter()。过滤stream中元素，表达式结果**必须为boolean值**，为真置于新stream，为假过滤掉

需求：获取指定颜色的苹果

```
vate static void getStreamMap(Apple.Color c) {
 // 基于源集合，创建流
 Stream<Apple> appleStream = APPLES.stream();
 // 基于颜色，过滤流中的元素对象，将符合的元素置于新的流
 Stream<Apple> colorStream = appleStream.filter(a -> c == a.getColor());
 // 将流中元素，聚合为List集合
 List<Apple> apples = colorStream.collect(Collectors.toList());
```

++++++++++++++++++++++++++++

基于stream的方法链

```
List<Apple> apples2 = APPLES
        .stream()
        .filter(a -> c == a.getColor())
        .collect(Collectors.toList());
```

需求：获取指定颜色，且大于等于重量的苹果

Chapter Objectives

```
List<Apple> apples = APPLES.stream()
        .filter(a -> a.getColor() == c && a.getWeight() >= weight)
        .collect(Collectors.toList());
```

效果相同
但语义描述更好
且执行时将自动优化

操作流
过滤颜色
过滤重量
聚合

```
List<Apple> apples = APPLES.stream()
        .filter(a -> a.getColor() == c)
        .filter(a -> a.getWeight() >= weight)
        .collect(Collectors.toList());
```

- Stream<T> map()。映射Stream中元素，基于条件将元素映射为新类型元素

需求：将苹果重量收集为新集合，并打印输出

此时流中的类型为映射类型

```
APPLES.stream()                    Stream<Apple>
      .map(a -> a.getWeight())     Stream<Integer>
      .collect(Collectors.toList()) List<Integer>
      .forEach(i -> System.out.println(i));
```

获取重量
并映射到流中

聚合的结果为
封装整型的集合

++++++++++++++++++++++++++++++++++

进一步简化

```
APPLES.stream()                    Stream<Apple>
      .map(Apple::getWeight)       Stream<Ir
      .collect(Collectors.toList()) Li
      .forEach(System.out::println);
```

Idea的自动提示
非代码

当lambda表达式
只对参数结果操作时
通过
类型::类型中的方法
简化代码
自动将唯一的参数注入方法

Functional Interfaces

- Stream<T> Sorted()。对stream中元素排序
- Comparator类。比较器。控制顺序
  - comparing()，基于指定值排序
  - reversed()，倒序

以苹果重量顺序排序

```
APPLES.stream()
        .sorted(Comparator.comparing(Apple::getWeight))
        .collect(Collectors.toList())
        .forEach(a -> {
            System.out.println(a.getId() + "/"
                    + a.getColor() + "/" + a.getWeight());
        });
```

```
1/RED/200
4/GREEN/230
2/GREEN/250
3/RED/260
```

以苹果ID逆序排序

```
APPLES.stream()
        .sorted(Comparator.comparing(Apple::getId).reversed())
        .collect(Collectors.toList())
        .forEach(a -> {
            System.out.println(a.getId() + "/"
                    + a.getColor() + "/" + a.getWeight());
        });
```

```
4/GREEN/230
3/RED/260
2/GREEN/250
1/RED/200
```

Chapter Objectives

- Stream的其他方法
  - takeWhile()，从同遍历置于流，到结果为true停止
  - dropWhile()，从头遍历，到结果为true开始置于流
  - flatMap()，将多层映射合并
  - findFirst()，从流中取第一个符合条件元素，封装到Optional
  - findAny()，从流中取任意一个符合条件元素
  - anyMatch()，任意一个元素符合条件，返回true
  - allMatch()，全部元素符合条件，返回true
  - ……

Chapter Objectives

- T collect()。聚合，收集stream一系列中间操作产生的结果
- Collectors(java.util.stream.Collectors)类，用于操作聚合结果的工具类
  - groupingBy()/mapping()
  - toList()/toSet()/toMap()
  - ……

需求：将所有苹果的颜色映射为新集合

```
APPLES.stream() Stream<Apple>
        .map(Apple::getWeight) Stream<Ir
        .collect(Collectors.toList()) Li
        .forEach(System.out::println);
```

```
RED
GREEN
RED
GREEN
```

聚合为Set集合
过滤相同颜色

```
APPLES.stream() Stream<Apple>
        .map(Apple::getColor) Stream<Ap
        .collect(Collectors.toSet()) Se
        .forEach(System.out::println);
```

```
RED
GREEN
```

- groupingBy()，基于给定数据，以Map分组集合

需求：将不同颜色苹果分组

```
Map<Apple.Color, List<Apple>> map = APPLES
        .stream()
        .collect(Collectors.groupingBy(a -> a.getColor()));
```

- toMap(K, V)，基于给定键值，以Map分组集合

需求：以ID为键，以元素对象为值

```
Map<Integer, Apple> map2 = APPLES
        .stream()
        .collect(Collectors.toMap(a -> a.getId(), a -> a));
```

- Java8支持Map forEach循环，可直接获取每次遍历元素的键与值

按ID分组
并获取重量

```
Map<Integer, Apple> map2 = APPLES
        .stream()
        .collect(Collectors.toMap(a -> a.getId(), a -> a));

map2.forEach((k, v) -> {
    System.out.println(k + "/" + v.getWeight());
});
```

```
1/200
2/250
3/260
4/230
```

apter Objectives

```
int count = 0;
APPLES.forEach(a -> {
    count++;
    S
});
```

Variable used in lambda expression should be final or effectively final

外部数据
必须为final
或同等效果

函数式编程
不应，对外部数据产生影响
函数是独立于类/对象的
函数在执行时
外部数据可能已不存在或改变
(局部变量可能已销毁)

如需使用
外部数据必须为final才能保持引用

removeIf

- Boolean removeIf()
- The default implementation traverses all elements of the collection using its iterator(). Each matching element is removed using **Iterator.remove()**.
- Collection接口中定义。移除符合函数表达式的元素。底层依然基于Iterator迭代器实现

```
users.removeIf(u -> "SUN".equals(u.getName()));
users.forEach(u -> System.out.println(u.getName
```

```
BO
ZHANG
```

极简洁优雅的实现了元素的移除
使代码关注于对集合的业务操作本身
而非游标/位置/移动等非业务逻辑操作

```
Iterator<User> iUsers = users.iterator();
while (iUsers.hasNext()) {
```

The loop could be replaced with Collection.removeIf more... (Ctrl+F1)

Idea自动提示优化

```
     * @since 1.8
     */
@Contract(mutates="this") default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
```

removeIf()源码
底层依然基于
Iterator迭代器实现

Functional Interfaces

- A functional interface is an interface that contains **one and only one** abstract method.
- 函数式接口，能且只能包含1个抽象方法的接口
- 函数式接口中，仅声明定义函数的参数/参数类型/返回类型，使用时具体实现
- @FunctionalInterface(java.lang.FunctionalInterface)注解，声明该接口为函数式接口，只要是只包含一个抽象方法的函数式接口，可以省略；当接口中定义多于1个抽象方法时，无法编译

- Java定义了多个函数接口，供集合Stream等使用

```java
@FunctionalInterface
public interface MyFunction {
    int getValue(int x);
}
```

声明一个函数
函数的参数，以及返回类型

```
++++++++++++++++++++++++
```

```java
MyFunction f = x -> {
    return x * x;
};
```

基于声明的函数
实现一个函数
实现的结果为将传入函数的参数平方返回

```
++++++++++++++++++++++++
```

```java
public class MyList {
    private List<Integer> list;
    public void forEach(MyFunction f) {
        for (Integer i : list) {
            int result = f.getValue(i);
            System.out.println(result);
        }
    }
```

将函数作为forEach()方法的参数传入
循环封装的集合
将每次遍历的元素i
作为函数的参数传入函数
执行函数并获取执行结果

```java
List<Integer> nums = new ArrayList<>();
nums.add(1); nums.add(5); nums.add(7);

MyFunction f = x -> {
    return x * x;
};
MyList list = new MyList(nums);
list.forEach(f);
```

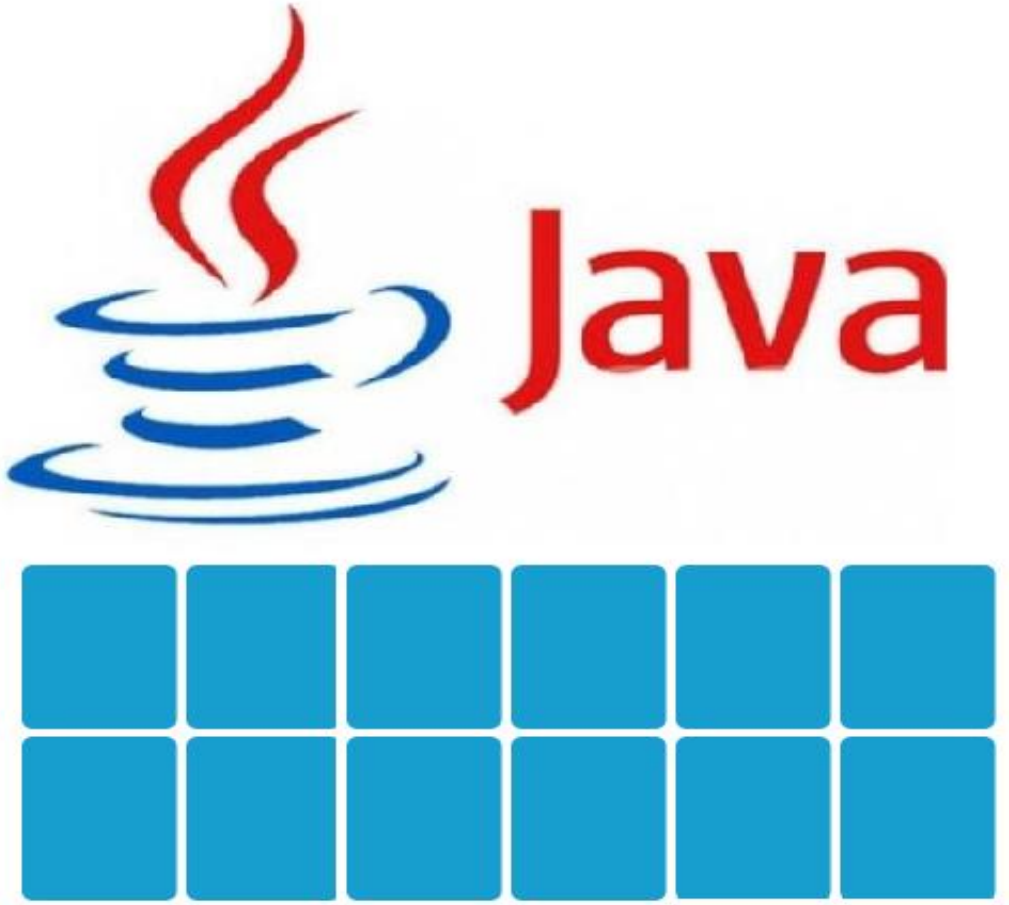
封装一个集合
执行forEach方法时
传入预执行的函数

```
1
25
49
```

- **Stream仅操作源集合中的元素**，基于操作产生新集合，不改变源集合结构。即，源集合/新集合，均持有该元素对象的引用
  - filter()，过滤
  - map()，映射
  - sorted()，排序
  - collect()，聚合
- 操作均返回stream，因此可以链接形成一条单向的管道
- 支持多线程并发处理，而无需显式创建线程(后期讨论)

Summary

	content
Java集合框架	Java集合框架的优点；核心接口；实现集合的基本数据结构；设计集合接口与集合实现分离的目的与优点；
Collection接口	不同集合/不同数据结构间集合的转换；Collection接口中的常用方法；
List接口	特点；基本方法；声明方法；元素为基本类型的声明使用方法；继承关系对象的添加；引用传递；了解iterator迭代器；
Set接口	特点；基本方法；
Map接口	键值对；键与值的支持类型；相同key的判断；常用方法；string类型的键；
Java集合Stream	Stream的特点；forEach()/map()/filter()/sorted()操作方法；collect()操作结果的处理；Collectors类的常用方法toList() toSet() toMap()；Map的foreach循环；
函数式编程	特点，与面向对象的区别；方法与函数；Lambda表达式的声明；函数式接口；@FunctionalInterface函数式接口注解；



第六章

工具包 – Optional

東北林業大學
NORTHEAST FORESTRY UNIVERSITY

```
public class USB {
    private String version;
++++++++++++++++++++++++++
public class Soundcard {
    private USB usb;
++++++++++++++++++++++++++
public class Computer {
    private Soundcard soundcard;
```

编写方法
实现传入computer对象
返回其上的声卡上的usb上的版本
如果不存在，返回UNKNOWN

```
lic static String getVersion0(Computer com) {
 String version = com.getSoundcard().getUsb().getVersion();
 return version == null ? "UNKNOWN" : version;
```

```
private static String getVersion1(Computer com) {
    String version = "UNKNOWN";
    if (com != null) {
        Soundcard soundcard = com.getSoundcard();
        if (soundcard != null) {
            USB usb = soundcard.getUsb();
            if (usb != null) {
                version = usb.getVersion();
            }
        }
    }
    return version;
```

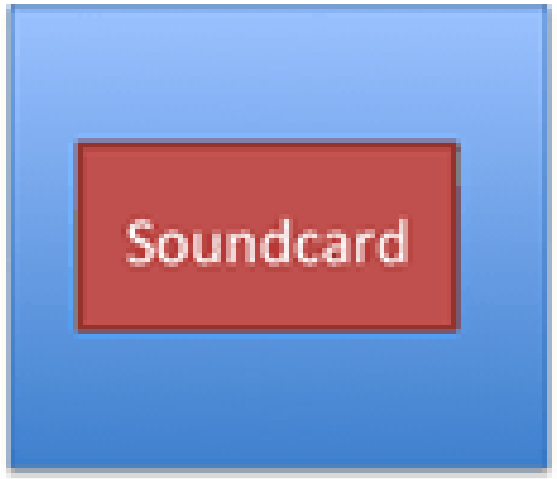
在无法确定使用对象情况时
必须显式判断对象是否为null
大量冗余的啰嗦的难看的
与业务逻辑无关的样板代码

任何一处没有检测
都可能引起运行时空指针异常

Optional

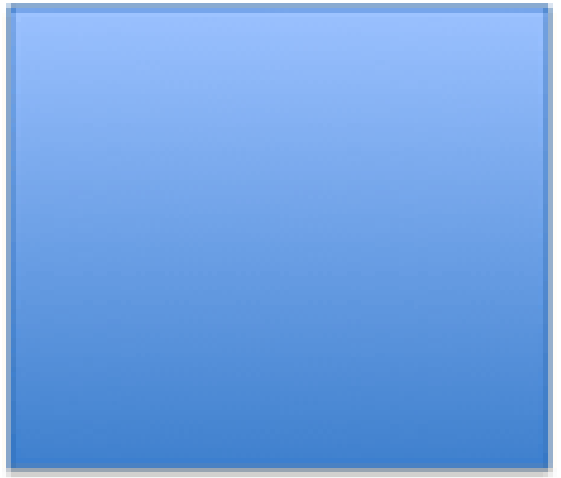
- Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values.
- Make your code more readable and protect it against null pointer exceptions.
- java.util.Optional<T>
- 为解决空引用异常引入的，用于封装单值元素的容器(single-value container)
- 即，基于Optional提供的一系列方法，操作封装在Optional容器中的，可能引起空引用的元素对象

Optional<Soundcard>



Contains an object of type Soundcard

Optional<Soundcard>



An empty Optional

Optional

- 创建容器
  - ofNullable() / of()
- 执行操作，基于容器是否为空，执行操作，**无返回值**
  - ifPresent() / ifPresentOrElse()
- 中间操作，将操作结果，置于新Optional容器以执行后续操作，结果为空，也会返回相同类型的空容器
  - filter() / map() / or()(需手动创建容器注入)
- 获取操作，获取容器中对象
  - orElse() / orElseGet() / get()
- 判断方法，判断当前容器是否为空
  - isEmpty() / isPresent()

- Creating Optional Objects
- Optional<T> Optional.of(T value)，基于必不为空对象，创建optional容器，注入为空元素将抛出NullPointerException异常
- Optional<T> Optional.ofNullable(T value)，基于可能为空的对象，创建optional容器

```
Optional<Computer> op = Optional.ofNullable(null);

Optional<Computer> op2 = Optional.ofNullable(new Computer());
```

- Conditional Action With，void ifPresent(action)
- 当optional容器不为空时，执行指定函数；为空忽略执行

```
/ate static void getIfPresent(USB usb) {
 Optional<USB> usbOP = Optional.ofNullable(usb);
 usbOP.ifPresent(u -> {
     System.out.println(u.getVersion());
 });
```

创建optional容器
将对象置于容器

当容器不为空
基于容器中元素
执行函数

++++++++++++++++++++++++++++

```
 Optional.ofNullable(usb)
         .ifPresent(u -> {
             System.out.println(u.getVersion());
         });
```

基于方法链
简化代码

++++++++++++++++++++++++++++

```
Optional.ofNullable(new Computer())
        .ifPresent(c -> {
            System.out.println(c.getSoundcard().getUsb().getVersion());
        });
```

++++++++++++++++++++++++++++

```
"main" java.lang.NullPointerException
3.optional.OptionalTest.lambda$getIfPre
```

仅检测指定元素对象
但嵌套对象为空时
依然异常

- Void ifPresentOrElse(action，emptyAction)，(Java9)
- 2个函数的参数，当容器不为空时执行第一个函数；为空执行第二个函数

```
vate static void ifPresentOrElse(USB usb) {
 Optional.ofNullable(usb)
        .ifPresentOrElse(u -> {
            System.out.println(u.getVersion());
        }, () -> {
            System.out.println("usb为空");
        });
```

不为空时注入容器内元素

第一个函数
不为空时执行

为空，函数无参数

第二个函数
为空时执行

- Optional<T> filter().If a value is present, and the value matches the given predicate, returns an Optional describing the value, otherwise returns an empty Optional.
- 过滤容器中元素。容器为空，返回空容器；容器不为空，执行过滤；符合条件，将对象置于新容器，不符合条件，返回空容器
- 即，无论是否匹配，均返回一个容器，用于后续操作

需求：如果版本为3.0打印，否则不执行操作

```
vate static void filter(USB usb) {
    Optional.ofNullable(usb)
            .filter(u -> "3.0".equals(u.getVersion()))
            .ifPresent(u -> {
                System.out.println(u.getVersion());
            });
```

置入容器

过滤，并将结果置于新容器

基于新容器操作

- Transforming Value with， Optional<T> map(mapper)
- If a value is present, returns an Optional describing (as if by ofNullable(T)) the result of applying the given mapping function to the value, otherwise returns an empty Optional.
- 基于容器中对象，映射。容器为空，返回相同类型的空容器；容器不为空，执行映射函数，将映射结果封装在新容器中
- 即，无论是否匹配，均返回一个容器，用于后续操作

需求：基于map映射usb版本，打印

```
vate static void map(USB usb) {
 Optional.ofNullable(usb)
         .map(USB::getVersion)
         .ifPresent(System.out::println);
```

将容器中元素
映射为一个新类型
置于新容器
返回类型为
Optional<String>

- Optional<T> or()，容器为空，执行函数，且必须返回一个，相同类型的容器，可以为空容器

需求：打印USB版本
如果版本为UNKNOWN，创建USB 1.1，打印
不为UNKNOWN，直接打印

```
vate static void or(USB usb) {
    Optional.ofNullable(usb)
            .filter(u -> !"UNKNOWN".equals(u.getVersion()))
            .or(() -> {
                return Optional.of(new USB( version: "1.1"));
            })
            .ifPresent(u -> {
                System.out.println(u.getVersion());
            });
```

置入容器

过滤

如果UNKNOWN
必须返回容器

此时
容器中

- Default Value With， T orElse()/orElseGet()
- 返回容器中对象，容器为空，则创建默认对象替代
- T orElseGet(supplier)，容器为空时，执行函数，且必须返回容器中相同类型对象；容器不为空不执行
- T orElse(T other)，无论是否为空，均创建默认对象
- 返回的不再是Optional容器，而是容器中最终的元素对象
- Exceptions with orElseThrow()，当为空时，抛出指定异常

需求：获取usb版本
为空创建一个usb 1.1

```
String v2 = Optional.ofNullable(usb)
        .orElse(new USB( version: "1.1"))
        .getVersion();
System.out.println(v2);
```

容器无论是否为空
均创建默认对象

容器不为空，返回容器中对象
容器为空，返回默认对象

Optional

```
USB usb = null;
String v1 = Optional.ofNullable(usb)
        .orElseGet(() -> {
            USB u = new USB( version: "1.1")
            return u;
        }).getVersion();
System.out.println(v1);
```

容器为空
执行函数
函数必须返回相同类型对象
不为空
则不会执行函数

- Returning Value with，T get()
- 返回容器中对象。但如果容器为空，抛出**NoSuchElementException异常**
- 因此，此方法应结合其他操作，确保容器不为空时使用

```
USB usb = null;
Optional.ofNullable(usb)
        .get();
```

```
orkspace-2019\java-examples\out\production\j
util.NoSuchElementException: No value presen
nal.get(Optional.java:148)
```

```
USB usb = null;
Optional.ofNullable(usb)
        .or(() -> {
            System.out.println("怎么会没有呢");
            return Optional.of(new USB( version: "1.1"));
        })
        .get()
        .getVersion();
```

编写方法
实现传入computer对象
返回其上的声卡上的usb上的版本
如果不存在，返回UNKNOWN

```
static void getVersion2(Computer com) {
ng version = Optional.ofNullable(com) Optional<Computer>
    .map(Computer::getSoundcard) Optional<Soundcard>
    .map(Soundcard::getUsb) Optional<USB>
    .map(USB::getVersion) Optional<String>
    .orElse( other: "UNKNOWN");
```

将computer对象置于容器
获取映射soundcard对象置于新容器
获取映射usb对象置于新容器
获取映射string对象置于新容器

假设usb为空
仍然返回空类型为USB的容器
供后续使用

orElse()
当前容器不为空则返回容器中元素
为空，返回默认值

总是返回空容器，资源消耗?

- CC：Optional容器对象无法被序列化。因此，Optional不应作为属性。非特殊情况，不建议作为方法的返回类型/参数。应仅作为库/工具方法，用于执行与空引用有关的操作

```
@PostMapping("/cookielogin")
public Map postCookieLogin(@CookieValue(Constant.COOKIS_NUMBER) Optional<String> cookie) {
    User u = cookie.map(c -> {
```

需要使用用户请求cookie
但如果用户实际请求时没有提供
将在此方法调用前抛出异常
因此
将可能为空的参数
封装到Optional容器