集合框架

本节目标

1. Java类集简介
2. List集合接口
3. Set集合接口
4. 集合输出
5. Map集合
6. 栈与队列
7. Properties属性操作
8. Collections工具类
9. Stream数据流

1.Java类集简介

1.1 Java类集引出

类集实际上就属于动态对象数组，在实际开发之中，数组的使用出现的几率并不高，因为数组本身有一个最大的缺陷:数组长度是固定的。由于此问题的存在，从JDK1.2开始，Java为了解决这种数组长度问题，提供了动态的对象数组实现框架--Java类集框架。Java集合类框架实际上就是java针对于数据结构的一种实现。而在数据结构之中，最为基础的就是链表。

下面我们一起来回顾下链表的特点：

1. 节点关系的处理操作，核心需要一个Node类(保存数据，设置引用)。
2. 在进行链表数据的查找、删除的时候需要equals()方法支持。

实际上之前链表的实现就是参考Java集合类实现的。

1.2 Collection接口

在Java的类集里面(java.util包)提供了两个最为核心的接口:Collection、Map接口。其中Collection接口的操作形式与之前编写链表的操作形式类似，每一次进行数据操作的时候只能够对单个对象进行处理。

Collection是单个集合保存的最大父接口。

Collection接口的定义如下:

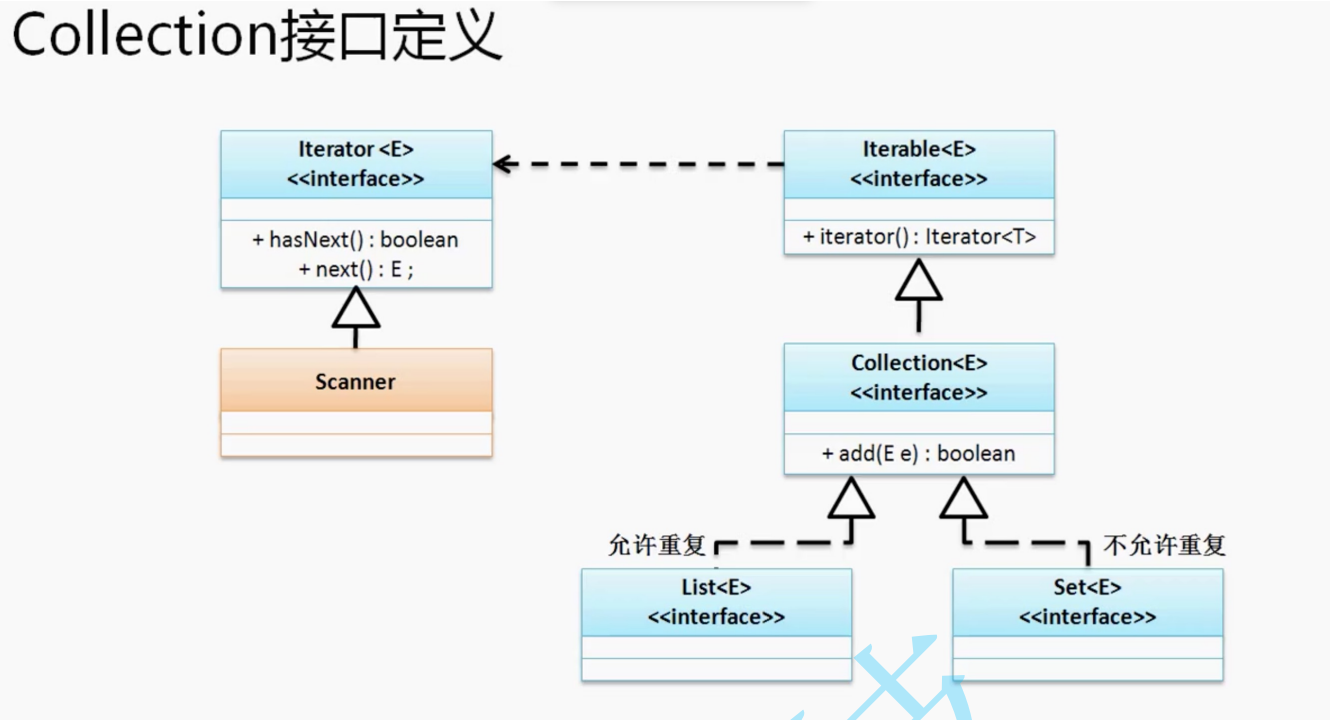
```
public interface Collection<E> extends Iterable<E>
```

从JDK1.5开始发现Collection接口上追加有泛型应用，这样的直接好处就是可以避免ClassCastException，里面的所有数据的保存类型应该是相同的。在JDK1.5之前Iterable接口中的iterator()方法是直接在Collection接口中定义的。此接口的常用方法有如下几个:

No.	方法名称	类型	描述
1.	public boolean add(E e);	普通	向集合中添加数据
2.	public boolean addAll(Collection<? extends E> c);	普通	向集合中添加一组数据
3.	public void clear();	普通	清空集合数据
4.	public boolean contains(Object o);	普通	查找数据是否存在，需要使用 equals()方法
5.	public boolean remove(Object o);	普通	删除数据，需要 equals()方法
6.	public int size();	普通	取得集合长度
7.	public Object[] toArray();	普通	将集合变为对象数组返回
8.	public Iterator<E> iterator();	普通	取得 Iterator 接口对象，用于集合输出

在开发之中如果按照使用频率来讲:add()、iterator()方法用到的最多。需要说明的一点是，我们很少会直接使用Collection接口，Collection接口只是一个存储数据的标准，并不能区分存储类型。例如：要存放的数据需要区分重复与不重复。在实际开发之中，往往会考虑使用Collection接口的子接口:List(允许数据重复)、Set(不允许数据重复)。

以上接口的继承、使用关系如下：



Collection接口中有两个重要方法:add()、iterator()。子接口都有这两个方法。

2.List接口

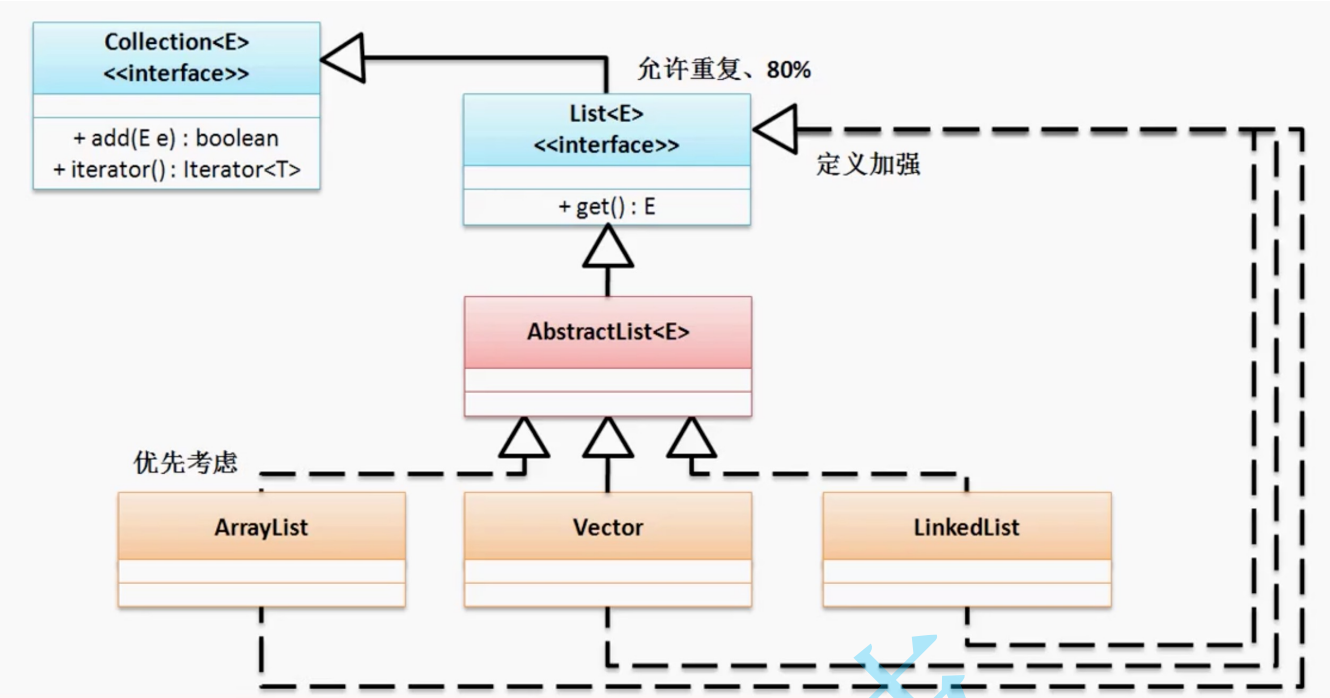
2.1 List接口概述

在实际开发之中，List接口的使用频率可以达到Collection系列的80%。在进行集合处理的时候，优先考虑List接口。

首先来观察List接口中提供的方法,在这个接口中有两个重要的扩充方法：

No.	方法名称	类型	描述
1.	public E get(int index);	普通	根据索引取得保存数据
2.	public E set(int index, E element);	普通	修改数据

List子接口与Collection接口相比最大的特点在于其有一个get()方法，可以根据索引取得内容。由于List本身还是接口，要想取得接口的实例化对象，就必须有子类，在List接口下有三个常用子类：ArrayList、Vector、LinkedList。



最终的操作还是以接口为主，所以所有的方法只参考接口中定义的方法即可。

2.1 ArrayList子类(优先考虑)

ArrayList是一个针对于List接口的数组实现。下面首先利用ArrayList进行List的基本操作。

范例：观察List基本处理。

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
        // 此时集合里面只保存String类型
        List<String> list = new ArrayList<>() ;
        list.add("Hello") ;
        // 重复数据
        list.add("Hello") ;
        list.add("Bit") ;
        System.out.println(list) ;
    }
}
```

通过上述代码我们可以发现，List允许保存重复数据。

范例：观察其他操作

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
        // 此时集合里面只保存String类型
```

```
        List<String> list = new ArrayList<>() ;
        System.out.println(list.size()+"、" + list.isEmpty());
        list.add("Hello") ;
        // 重复数据
        list.add("Hello") ;
        list.add("Bit") ;
        System.out.println(list.size()+"、" + list.isEmpty());
        System.out.println(list) ;
        System.out.println(list.remove("Hello")) ;
        System.out.println(list.contains("ABC")) ;
        System.out.println(list.contains("Bit")) ;
        System.out.println(list);
    }
}
```

List本身有一个好的支持：存在get()方法，可以利用get()方法结合索引取得数据。

范例：List的get()操作

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
        // 此时集合里面只保存String类型
        List<String> list = new ArrayList<>() ;
        list.add("Hello") ;
        // 重复数据
        list.add("Hello") ;
        list.add("Bit") ;
        for (int i = 0; i < list.size() ; i++) {
            System.out.println(list.get(i)) ;      }
    }
}
```

get()方法是List子接口提供的。如果现在操作的是Collection接口，那么对于此时的数据取出只能够将集合变为对象数组操作。

范例：通过Collection进行输出处理。

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;

public class TestDemo {
    public static void main(String[] args) {
        // 此时集合里面只保存String类型
        Collection<String> list = new ArrayList<>() ;
        list.add("Hello") ;
        // 重复数据
        list.add("Hello") ;
        list.add("Bit") ;
        // 操作以Object为主，有可能需要向下转型，就有可能产生ClassCastException
        Object[] result = list.toArray() ;
        System.out.println(Arrays.toString(result)) ;
    }
}
```

开发中尽量不要使用Collection接口

2.2 集合与简单Java类

在以后的实际开发中，集合里面保存最多的数据类型，就是简单Java类。

范例：向集合保存简单Java类对象。

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

class Person {
    private String name ;
    private Integer age ;

    public Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(name, person.name) &&
                Objects.equals(age, person.age);
    }
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<>() ;
        personList.add(new Person("张三",10)) ;
        personList.add(new Person("李四",11)) ;
```

```
        personList.add(new Person("王五",12)) ;
        // 集合类中contains()、remove()方法需要equals()支持
        personList.remove(new Person("李四",11)) ;
        System.out.println( personList.contains(new Person("王五",12)));
        for (Person p: personList) {
            System.out.println(p) ;
        }
    }
}
```

集合操作简单java类时，对于remove()、contains()方法需要equals()方法支持。

2.3 旧的子类(Vector) 使用较少

Vector是从JDK1.0提出的，而ArrayList是从JDK1.2提出的。

范例：使用Vector

```
package www.bit.java.test;

import java.util.List;
import java.util.Vector;

public class TestDemo {
    public static void main(String[] args) {
        List<String> list = new Vector<>() ;
        list.add("hello") ;
        list.add("hello") ;
        list.add("bit") ;
        System.out.println(list) ;
        list.remove("hello") ;
        System.out.println(list) ;
    }
}
```

面试题：请解释ArrayList与Vector区别

1. 历史时间:ArrayList是从JDK1.2提供的，而Vector是从JDK1.0就提供了。
2. 处理形式：ArrayList是异步处理，性能更高；Vector是同步处理，性能较低。
3. 数据安全：ArrayList是非线程安全；Vector是线程安全。
4. 输出形式：ArrayList支持Iterator、ListIterator、foreach；Vector支持Iterator、ListIterator、foreach、Enumeration。

在以后使用的时候优先考虑ArrayList，因为其性能更高，实际开发时很多时候也是每个线程拥有自己独立的集合资源。如果需要考虑同步也可以使用concurrent包提供的工具将ArrayList变为线程安全的集合(了解)。

2.4 LinkedList子类

在List接口中还有一个LinkedList子类，这个子类如果向父接口转型的话，使用形式与之前没有任何区别。

范例：使用LinkedList

```
package www.bit.java.test;

import java.util.LinkedList;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
```

```
        List<String> list = new LinkedList<>() ;
        list.add("hello") ;
        list.add("hello") ;
        list.add("bit") ;
        System.out.println(list) ;
        list.remove("hello") ;
        System.out.println(list) ;

    }
}
```

面试题：请解释ArrayList与LinkedList区别

1. 观察ArrayList源码，可以发现ArrayList里面存放的是一个数组，如果实例化此类对象时传入了数组大小，则里面保存的数组就会开辟一个定长的数组，但是后面再进行数据保存的时候发现数组个数不够了会进行数组动态扩充。 所以在实际开发之中，使用ArrayList最好的做法就是设置初始化大小。
2. LinkedList：是一个纯粹的链表实现，与之前编写的链表程序的实现基本一样（人家性能高）。

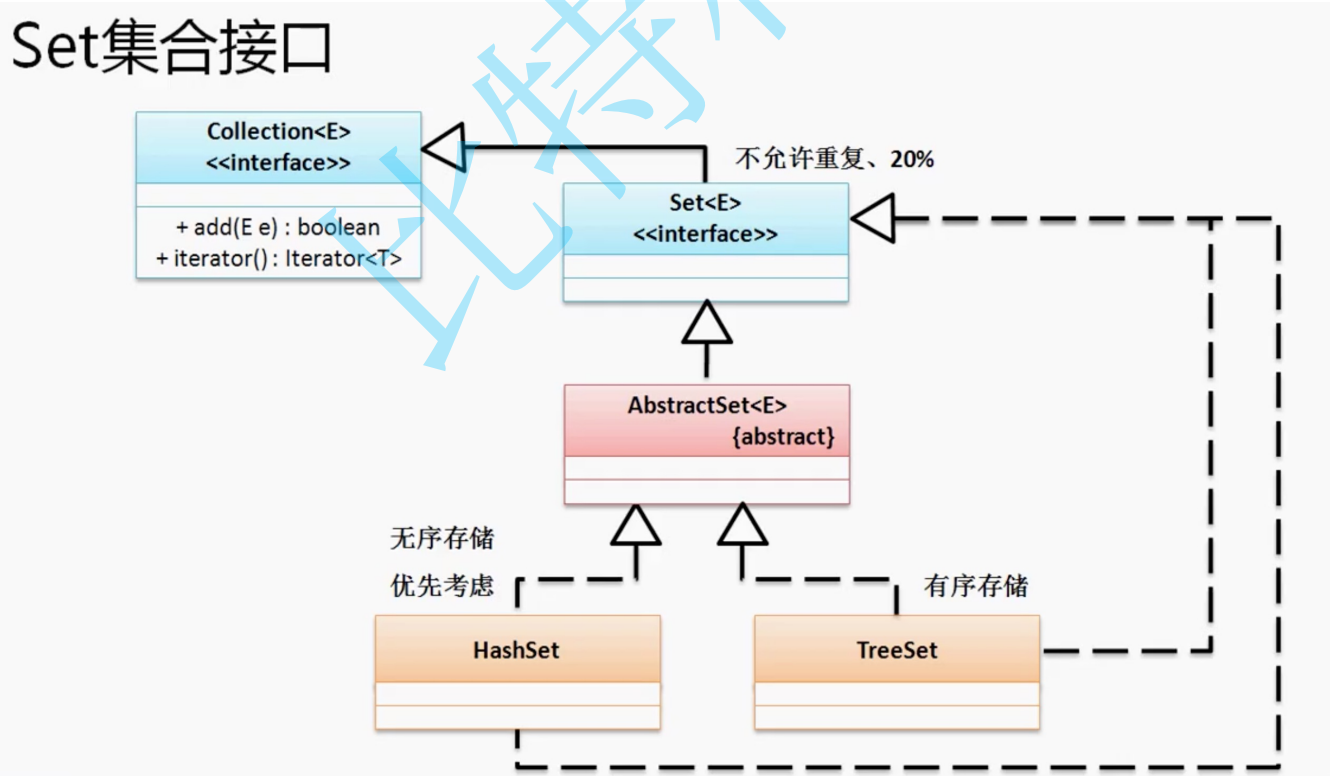
总结：ArrayList封装的是数组；LinkedList封装的是链表。ArrayList时间复杂度为1，而LinkedList的复杂度为n。

3.Set集合接口

Set接口与List接口最大的不同在于Set接口中的内容是不允许重复的。同时需要注意的是，Set接口并没有对Collection接口进行扩充，而List对Collection进行了扩充。因此，在Set接口中没有get()方法。

在Set子接口中有两个常用子类：HashSet(无序存储)、TreeSet(有序存储)

3.1 Set接口常用子类



范例：观察HashSet使用

```
package www.bit.java.test;

import java.util.HashSet;
```

```
import java.util.Set;

public class TestDemo {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>() ;
        set.add("Hello") ;
        // 重复元素
        set.add("Hello") ;
        set.add("Bit") ;
        set.add("Hello") ;
        set.add("Java") ;
        System.out.println(set) ;
    }
}
```

范例：观察TreeSet使用

```
package www.bit.java.test;

import java.util.Set;
import java.util.TreeSet;

public class TestDemo {
    public static void main(String[] args) {
        Set<String> set = new TreeSet<>() ;
        set.add("C") ;
        set.add("C") ;
        set.add("D") ;
        set.add("B") ;
        set.add("A") ;
        System.out.println(set) ;
    }
}
```

TreeSet使用的是升序排列的模式完成的。

3.2 TreeSet排序分析

既然TreeSet子类可以进行排序，所以我们可以利用TreeSet实现数据的排列处理操作。此时要想进行排序实际上是针对于对象数组进行的排序处理，而如果要进行对象数组的排序，对象所在的类一定要实现Comparable接口并且覆写compareTo()方法，只有通过此方法才能知道大小关系。

需要提醒的是如果现在是用Comparable接口进行大小关系匹配，所有属性必须全部进行比较操作。

范例：使用TreeSet排列

```
package www.bit.java.test;

import java.util.Set;
import java.util.TreeSet;
class Person implements Comparable<Person> {
    private String name ;
    private Integer age ;

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
```

```
    }

    @Override
    public int compareTo(Person o) {
        if (this.age > o.age ) {
            return 1 ;
        }else if (this.age < o.age ){
            return -1 ;
        }else {
            return this.name.compareTo(o.name) ;
        }
    }

    public Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Set<Person> set = new TreeSet<>() ;
        set.add(new Person("张三",20)) ;
        // 重复元素
        set.add(new Person("张三",20)) ;
        set.add(new Person("李四",20)) ;
        set.add(new Person("王五",19)) ;
        System.out.println(set) ;
    }
}
```

在实际使用之中，使用TreeSet过于麻烦了。项目开发之中，简单java类是根据数据表设计得来的，如果一个类的属性很多，那么比较起来就很麻烦了。所以我们一般使用的是HashSet。

3.3 Comparable接口与Comparator接口

3.3.1 Comparable(内部排序接口)简介

Comparable 是排序接口。

若一个类实现了Comparable接口，就意味着“**该类支持排序**”。 即然实现Comparable接口的类支持排序，假设现在存在“实现Comparable接口的类的对象的List列表(或数组)”，则该List列表(或数组)可以通过 Collections.sort（或Arrays.sort）进行排序。

此外，“实现Comparable接口的类的对象”可以用作“有序映射(如TreeMap)”中的键或“有序集合(TreeSet)”中的元素，而不需要指定比较器。

Comparable 定义

Comparable 接口仅仅只包括一个函数，它的定义如下：

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

关于返回值:

可以看出compareTo方法返回一个int值，该值有三种返回值:

1. 返回负数:表示当前对象小于比较对象
2. 返回0:表示当前对象等于目标对象
3. 返回正数:表示当前对象大于目标对象

3.3.2 Comparator(外部排序接口) 简介

Comparator 是比较器接口。

我们若需要控制某个类的次序，而该类本身不支持排序(即没有实现Comparable接口)；那么，我们可以建立一个“该类的比较器”来进行排序。这个“比较器”只需要实现Comparator接口即可。

也就是说，我们可以通过“**实现Comparator类来新建一个比较器**”，然后通过该比较器对类进行排序。

Comparator 定义

Comparator 接口仅仅只包括两个个函数，它的定义如下：

```
package java.util;

public interface Comparator<T> {

    int compare(T o1, T o2);

    boolean equals(Object obj);
}
```

int compare(T o1, T o2) 是“比较o1和o2的大小”。返回“负数”，意味着“o1比o2小”；返回“零”，意味着“o1等于o2”；返回“正数”，意味着“o1大于o2”。

Comparator 和 Comparable 比较

Comparable是排序接口；若一个类实现了Comparable接口，就意味着“该类支持排序”。 而Comparator是比较器；我们若需要控制某个类的次序，可以建立一个“该类的比较器”来进行排序。

我们不难发现：Comparable相当于“内部比较器”，而Comparator相当于“外部比较器”。

范例:使用Comparator接口

```
package collection;

import java.util.Comparator;
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;
```

```java
/**
 * @author yuisama
 * @date 2018/12/21 16:46
 */

class Person {
    private String name;
    private Integer age;

    public Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}

class AscAgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getAge() - o2.getAge();
    }
}

class DescAgeComparator implements Comparator<Person> {

    @Override
    public int compare(Person o1, Person o2) {
        return o2.getAge() - o1.getAge();
    }
}


public class Test {
    public static void main(String[] args) {
        Set<Person> set = new TreeSet<>(new AscAgeComparator());
        set.add(new Person("张三",20));
        set.add(new Person("李四",18));
```

```
        System.out.println(set);
        Set<Person> set1 = new TreeSet<>(new DescAgeComparator());
        set1.add(new Person("张三",20));
        set1.add(new Person("李四",18));
        System.out.println(set1);
    }
}
```

3.4 重复元素判断(hashCode与equals方法)

在使用TreeSet子类进行数据保存的时候，重复元素的判断依靠的ComParable接口完成的。但是这并不是全部Set接口判断重复元素的方式，因为如果使用的是HashSet子类，由于其跟Comparable没有任何关系，所以它判断重复元素的方式依靠的是Object类中的两个方法：

1. hash码： public native int hashCode();
2. 对象比较：public boolean equals(Object obj);

equals()的作用是用来判断两个对象是否相等，在Object里面的定义是

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

这说明在我们实现自己的equals方法之前，equals等价于==,而==运算符是判断两个对象是不是同一个对象，即他们的**地址是否相等**。而覆写equals更多的是追求两个对象在**逻辑上的相等**，你可以说是**值相等**，也可说是**内容相等**。

覆写equals的准则

自反性：对于任何非空引用值 x，x.equals(x) 都应返回 true。

对称性：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

传递性：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true， 并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

一致性：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false， 前提是对象上 equals 比较中所用的信息没有被修改。

非空性：对于任何非空引用值 x，x.equals(null) 都应返回 false。

hashCode用于返回对象的hash值，主要用于查找的快捷性，因为hashCode也是在Object对象中就有的，所以所有Java对象都有hashCode，在Hashtable和HashMap这一类的散列结构中，都是通过hashCode来查找在散列表中的位置的。

在Java中进行对象比较的操作有两步：第一步要通过一个对象的唯一编码找到一个对象的信息，当编码匹配之后再调用equals()方法进行内容的比较。

范例：覆写hashCode()与equals()方法消除重复

```
package www.bit.java.test;

import java.util.HashSet;
import java.util.Objects;
import java.util.Set;
class Person implements Comparable<Person> {
    private String name ;
    private Integer age ;

    @Override
    public String toString() {
```

```
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(name, person.name) &&
                Objects.equals(age, person.age);
    }

    @Override
    public int hashCode() {

        return Objects.hash(name, age);
    }

    @Override
    public int compareTo(Person o) {
        if (this.age > o.age ) {
            return 1 ;
        }else if (this.age < o.age ){
            return -1 ;
        }else {
            return this.name.compareTo(o.name) ;
        }
    }

    public Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Set<Person> set = new HashSet<>() ;
        set.add(new Person("张三",20)) ;
        // 重复元素
        set.add(new Person("张三",20)) ;
        set.add(new Person("李四",20)) ;
```

```
        set.add(new Person("王五",19)) ;
        System.out.println(set) ;
    }
}
```

如果两个对象equals，那么它们的hashCode必然相等， 但是hashCode相等，equals不一定相等。

对象判断必须两个方法equals()、hashCode()返回值都相同才判断为相同。

个人建议：

1. 保存自定义对象的时候使用List接口；
2. 保存系统类信息的时候使用Set接口(避免重复)。

4.集合输出

在之前进行集合输出的时候都利用了toString()，或者利用了List接口中的get()方法。这些都不是集合的标准输出。如果从标准上来讲，集合输出一共有四种手段：**Iterator**、ListIterator、Enumeration、foreach。

4.1 迭代输出：Iterator(重要)

在JDK1.5之前，在Collection接口中就定义有iterator()方法，通过此方法可以取得Iterator接口的实例化对象；而在JDK1.5之后，将此方法提升为Iterable接口中的方法。无论你如何提升，只要Collection有这个方法，那么List、Set也一定有此方法。

对于Iterator接口最初的设计里面实际有三个抽象方法：

1. 判断是否有下一个元素: public boolean hasNext();
2. 取得当前元素: public E next();
3. 删除元素: public default void remove(); 此方法从JDK1.8开始变为default完整方法。

范例：标准的Iterator使用

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>() ;
        list.add("Hello") ;
        list.add("Hello") ;
        list.add("Bit") ;
        Iterator<String> iterator = list.iterator() ; // 实例化Iterator对象
        while (iterator.hasNext()) {
            String str = iterator.next() ;
            System.out.println(str) ;
        }
    }
}
```

对于Iterator接口中提供的remove()方法主要解决的就是集合内容的删除操作。

范例：删除元素

```
package www.bit.java.test;
```

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>() ;
        list.add("Hello") ;
        list.add("Hello") ;
        list.add("B") ;
        list.add("Bit") ;
        list.add("Bit") ;
        Iterator<String> iterator = list.iterator() ; // 实例化Iterator对象
        while (iterator.hasNext()) {
            String str = iterator.next() ;
            if (str.equals("B")) {
//                // 使用集合提供的remove()方法，则会产生ConcurrentModificationException
//                list.remove("B") ;
                // 使用Iterator的remove方法则不会产生异常
                iterator.remove();
                continue;
            }
            System.out.println(str) ;
        }
    }
}
```

注意：**以后在进行集合输出的时候不要修改集合中元素！！！(同步问题，如何产生的？课后思考)**

4.2 双向迭代接口：ListIterator

Iterator输出有一个特点：只能够由前向后进行内容的迭代处理，而如果要想进行双向迭代，那么就必须依靠Iterator的子接口：ListIterator来实现。首先来观察一下此接口定义的方法：

1. 判断是否有上一个元素：public boolean hasPrevious();
2. 取得上一个元素:public E previous();

Iterator接口对象是由Collection接口支持的，但是ListIterator是由List接口支持的，List接口提供有如下方法：

- 取得ListIterator接口对象：public ListIterator listIterator();

范例：观察ListIterator接口使用

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class TestDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>() ;
        list.add("Hello") ;
        list.add("Hello") ;
        list.add("B") ;
        list.add("Bit") ;
        ListIterator<String> listIterator = list.listIterator() ;
        System.out.print("从前向后输出：") ;
        while (listIterator.hasNext()) {
            System.out.print(listIterator.next()+"、") ;
```

```
        }
        System.out.print("\n从后向前输出: ") ;
        while (listIterator.hasPrevious()) {
            System.out.print(listIterator.previous()+"、") ;
        }
    }
}
```

如果要想实现由后向前的输出，那么应该首先进行从前向后的输出，否则无法实现双向。

4.3 Enumeration枚举输出

在JDK1.0的时候就引入了Enumeration输出接口，而在JDK1.5的时候对其也做了更正，主要是追加了泛型的应用。首先来观察Enumeration的接口定义：

1. 判断是否有下一个元素:public boolean hasMoreElements();
2. 取得元素:public E nextElement();

但是要想取得这个接口的实例化对象，是不能依靠Collection、List、Set等接口的。**只能够依靠Vector子类，因为Enumeration最早的设计就是为Vector服务的，在Vector类中提供有一个取得Enumeration接口对象的方法:**

- 取得Enumeration接口对象:public Enumeration<E> elements()

范例：使用Enumeration输出

```
package www.bit.java.test;

import java.util.Enumeration;
import java.util.Vector;

public class TestDemo {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>() ;
        vector.add("Hello") ;
        vector.add("Hello") ;
        vector.add("B") ;
        vector.add("Bit") ;
        Enumeration<String> enumeration = vector.elements() ;
        while (enumeration.hasMoreElements()) {
            System.out.println(enumeration.nextElement()) ;
        }
    }
}
```

一些操作类库上依然只支持Enumeration，而不支持Iterator。

4.4 foreach输出

从JDK1.5开始foreach可以输出数组，实际上除了数组之外也可以输出集合。

范例：使用foreach输出

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>() ;
```

```
        list.add("Hello") ;
        list.add("Hello") ;
        list.add("B") ;
        list.add("Bit") ;
        for (String str : list) {
            System.out.println(str) ;
        }
    }
}
```

总结:

1. **看见集合输出就使用Iterator**
2. **Iterator和Enumeration中的方法掌握好**

5.Map集合

Collection集合的特点是每次进行单个对象的保存，如果现在要进行一对对象(偶对象)的保存就只能使用Map集合来完成，即Map集合中会一次性保存两个对象，且这两个对象的关系:key=value结构。这种结构最大的特点是可以通过key找到对应的value内容。

5.1 Map接口概述

首先来观察Map接口定义:

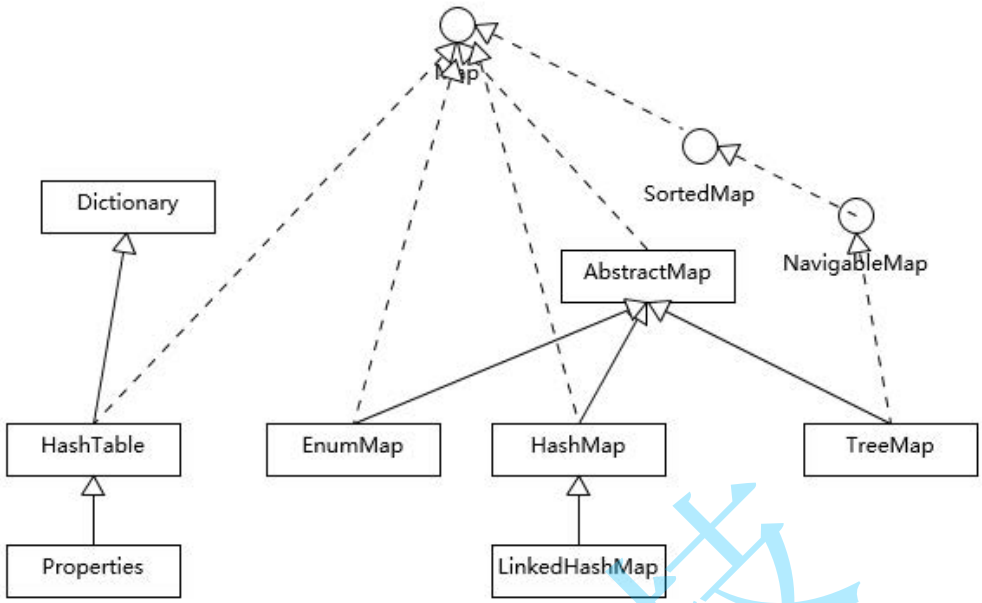
```
public interface Map<K,V>
```

在Map接口中有如下常用方法:

No.	方法名称	类型	描述
1.	public V put(K key, V value);	普通	向 Map 中追加数据
2.	public V get(Object key);	普通	根据 key 取得对应的 value，如果没有返回 null
3.	public Set<K> keySet();	普通	取得所有 key 信息、key 不能重复
4.	public Collection<V> values();	普通	取得所有 value 信息，可以重复
5.	public Set<Map.Entry<K, V>> entrySet();	普通	将 Map 集合变为 Set 集合

Map本身是一个接口，要使用Map需要通过子类进行对象实例化。Map接口的常用子类有如下四个： HashMap、Hashtable、TreeMap、ConcurrentHashMap。

首先来看Map集合的结构:



5.2 HashMap子类(常用)

HashMap是使用Map集合中最为常用的子类。

范例：Map基本操作

```
package www.bit.java.test;

import java.util.HashMap;
import java.util.Map;

public class TestDemo {
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<>() ;
        map.put(1,"hello") ;
        // key重复
        map.put(1,"Hello") ;
        map.put(3,"Java") ;
        map.put(2,"Bit") ;
        System.out.println(map) ;
        // 根据key取得value
        System.out.println(map.get(2)) ;
        // 查找不到返回null
        System.out.println(map.get(99)) ;
    }
}
```

范例：取得Map中所有key信息

```
package www.bit.java.test;

import java.util.HashMap;
```

```
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class TestDemo {
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<>() ;
        map.put(1,"hello") ;
        // key重复
        map.put(1,"Hello") ;
        map.put(3,"Java") ;
        map.put(2,"Bit") ;
        // 取得Map中所有的key信息
        Set<Integer> set = map.keySet() ;
        Iterator<Integer> iterator = set.iterator() ;
        while (iterator.hasNext()) {
            System.out.println(iterator.next()) ;
        }
    }
}
```

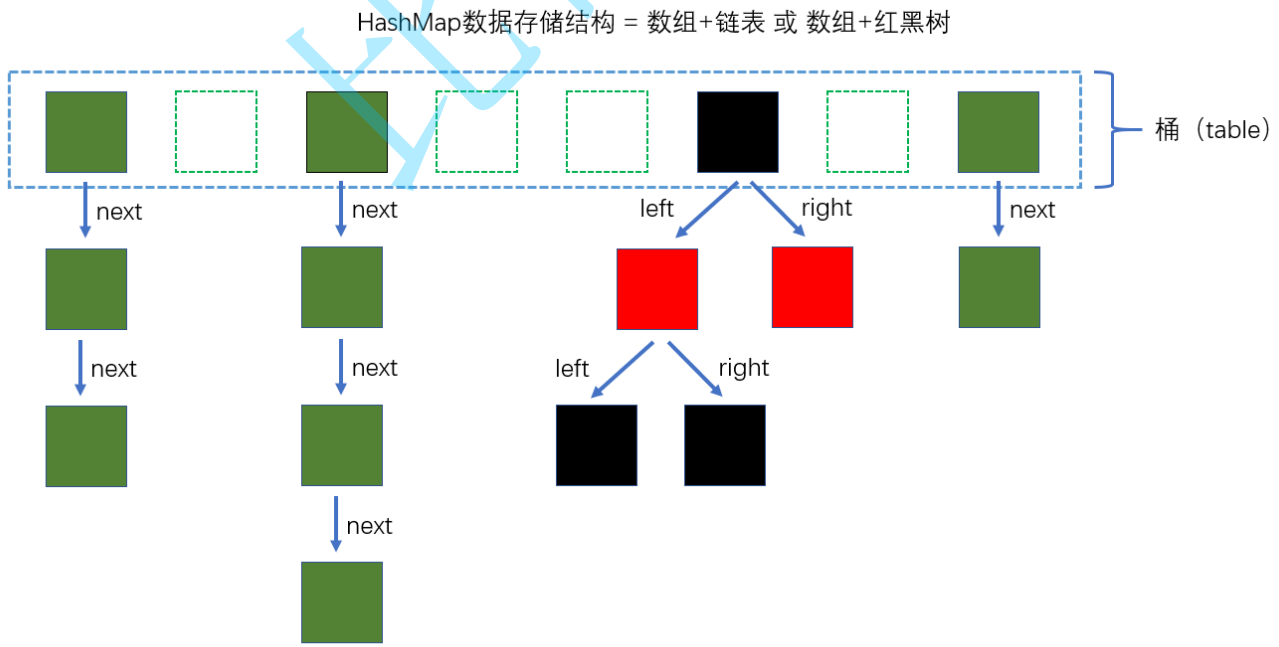
HashMap源码分析:

HashMap 设计与实现是个非常高频的面试题，所以我会在这进行相对详细的源码解读，主要围绕：

- HashMap 内部实现基本点分析。
- 容量（capcity）和负载系数（load factor）、树化

5.2.1 HashMap内部实现基本点分析

首先，我们来一起看看 HashMap 内部的结构，它可以看作是数组（Node[] table）和链表结合组成的复合结构，数组被分为一个个桶（bucket），通过哈希值决定了键值对在这个数组的寻址；哈希值相同的键值对，则以链表形式存储，你可以参考下面的示意图。这里需要注意的是，如果链表大小超过阈值（TREEIFY_THRESHOLD, 8），图中的链表就会被改造为树形结构。



从构造函数的实现来看，这个表格（数组）似乎并没有在最初就初始化好，仅仅设置了一些初始值而已。

```
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
```

所以，我们深刻怀疑，HashMap 也许是按照 **lazy-load** 原则，在首次使用时被初始化既然如此，我们去看看 put 方法实现，似乎只有一个 putVal 的调用：

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, onlyIfAbsent: false, evict: true);
}
```

看来主要的秘密似乎藏在 putVal 里面，到底有什么秘密呢？

```
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, next: null);
```

从 putVal 方法最初的几行，我们就可以发现几个有意思的地方：

- 如果表格是 null，resize 方法会负责初始化它，这从 tab = resize() 可以看出。
- resize 方法兼顾两个职责，创建初始存储表格，或者在容量不满足需求的时候，进行扩容（resize）。

  具体键值对在哈希表中的位置（数组 index）取决于下面的位运算：

  ```
  i = (n-1) & hash
  ```

  我们会发现，它并不是 key 本身的 hashCode，而是来自于 HashMap内部的另外一个 hash 方法。注意，为什么这里需要将高位数据移位到低位进行异或运算呢？这除是因为有些数据计算出的哈希值差异主要在高位，而 HashMap 里的哈希寻址是忽略容量以上的高位的，那么这种处理就可以有效避免类似情况下的哈希碰撞。

  下面进一步分析一下身兼多职的 resize 方法，面试官经常追问它的源码设计。

```
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
```

依据 resize 源码，不考虑极端情况（容量理论最大极限由 MAXIMUM_CAPACITY 指定，数值为 1<<30，也就是 2 的 30 次方），我们可以归纳为：

- 门限值等于（负载因子）*（容量），如果构建 HashMap 的时候没有指定它们，那么就是依据相应的默认常量值。

- 门限通常是以倍数进行调整 （newThr = oldThr << 1），我前面提到，根据 putVal 中的逻辑，当元素个数超过门限大小时，则调整 Map 大小。
- 扩容后，需要将老的数组中的元素重新放置到新的数组，这是扩容的一个主要开销来源。

5.2.2 容量、负载因子和树化

前面我们快速梳理了一下 HashMap 从创建到放入键值对的相关逻辑，现在思考一下，为什么我们需要在乎容量和负载因子呢？

这是因为容量和负载系数决定了可用的桶的数量，空桶太多会浪费空间，如果使用的太满则会严重影响操作的性能。极端情况下，假设只有一个桶，那么它就退化成了链表，完全不能提供所谓常数时间存的性能。 既然容量和负载因子这么重要，我们在实践中应该如何选择呢？

如果能够知道 HashMap 要存取的键值对数量，可以考虑预先设置合适的容量大小。具体数值我们可以根据扩容发生的条件来做简单预估，根据前面的代码分析，我们知道它需要符合计算条件：

负载因子 * 容量 > 元素数量

所以，预先设置的容量需要满足，大于“预估元素数量 / 负载因子”，同时它是 2 的幂数，结论已经非常清晰了。

而对于负载因子，我建议：

- 如果没有特别需求，不要轻易进行更改，因为 JDK 自身的默认负载因子是非常符合通用场景的需求的。
- 如果确实需要调整，建议不要设置超过 0.75 的数值，因为会显著增加冲突，降低 HashMap的性能。
- 如果使用太小的负载因子，按照上面的公式，预设容量值也进行调整，否则可能会导致更加频繁的扩容，增加无谓的开销，本身访问性能也会受影响。

树化:

树化改造，对应逻辑主要在 putVal 和 treeifyBin 中。

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
        int n, index; Node<K,V> e;
        if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
            resize();
        else if ((e = tab[index = (n - 1) & hash]) != null) {
            // 树化改造逻辑
    }
```

上面是精简过的 treeifyBin 示意，综合这两个方法，树化改造的逻辑就非常清晰了，可以理解为，当 bin 的数量大于 TREEIFY_THRESHOLD 时：

- 如果容量小于 MIN_TREEIFY_CAPACITY，只会进行简单的扩容。
- 如果容量大于 MIN_TREEIFY_CAPACITY ，则会进行树化改造。

那么，为什么 HashMap 要树化呢？

本质上这是个**安全问题。**因为在元素放置过程中，如果一个对象哈希冲突，都被放置到同一个桶里，则会形成一个链表，我们知道链表查询是线性的，会严重影响存取的性能。

而在现实世界，构造哈希冲突的数据并不是非常复杂的事情，恶意代码就可以利用这些数据大量与服务器端交互，导致服务器端 CPU 大量占用，这就构成了哈希碰撞拒绝服务攻击，国内一线互联网公司就发生过类似攻击事件。

5.3 Hashtable子类

JDK1.0提供有三大主要类：Vector、Enumeration、Hashtable。Hashtable是最早实现这种二元偶对象数据结构，后期的设计也让其与Vector一样多实现了Map接口而已。

范例：观察Hashtable

```
package www.bit.java.test;

import java.util.Hashtable;
import java.util.Map;

public class TestDemo {
    public static void main(String[] args) {
        Map<Integer,String> map = new Hashtable<>() ;
        map.put(1,"hello") ;
        // key重复
        map.put(1,"Hello") ;
        map.put(3,"Java") ;
        map.put(2,"Bit") ;
        System.out.println(map);
    }
}
```

面试题：请解释HashMap与Hashtable的区别

No.	区别	HashMap	Hashtable
1	推出版本	JDK1.2	JDK1.0
2	性能	异步处理，性能高	同步处理、性能较低
3	安全性	非线程安全	线程安全
4	null 操作	允许存放 null(有且只有一个)	key 与 value 都不为空，否则出现 NullPointerException

以后使用的时候多考虑HashMap。

5.4 ConcurrentHashMap子类

为什么需要 ConcurrentHashMap?

Hashtable 本身比较低效，因为它的实现基本就是将 put、get、size 等各种方法加上“synchronized”。简单来说，这就导致了所有并发操作都要竞争同一把锁，一个线程在进行同步操作时，其他线程只能等待，大大降低了并发操作的效率。

那么能不能利用 Collections 提供的同步包装器来解决问题呢?

看看下面的代码片段，我们发现同步包装器只是利用输入 Map 构造了另一个同步版本，所有操作虽然不再声明成为 synchronized 方法，但是还是利用了“this”作为互斥的 mutex，没有真正意义上的改进!

```
    final Object      mutex;        // Object on which to synchronize

    SynchronizedMap(Map<K,V> m) {...}

    SynchronizedMap(Map<K,V> m, Object mutex) {...}

    public int size() { synchronized (mutex) {return m.size();} }
    public boolean isEmpty() { synchronized (mutex) {return m.isEmpty();} }
    public boolean containsKey(Object key) { synchronized (mutex) {return m.containsKey(k
    public boolean containsValue(Object value) { synchronized (mutex) {return m.containsV
    public V get(Object key) { synchronized (mutex) {return m.get(key);} }

    public V put(K key, V value) { synchronized (mutex) {return m.put(key, value);} }
    public V remove(Object key) { synchronized (mutex) {return m.remove(key);} }
    public void putAll(Map<? extends K, ? extends V> map) { synchronized (mutex) {m.putAl
    public void clear() { synchronized (mutex) {m.clear();} }
```

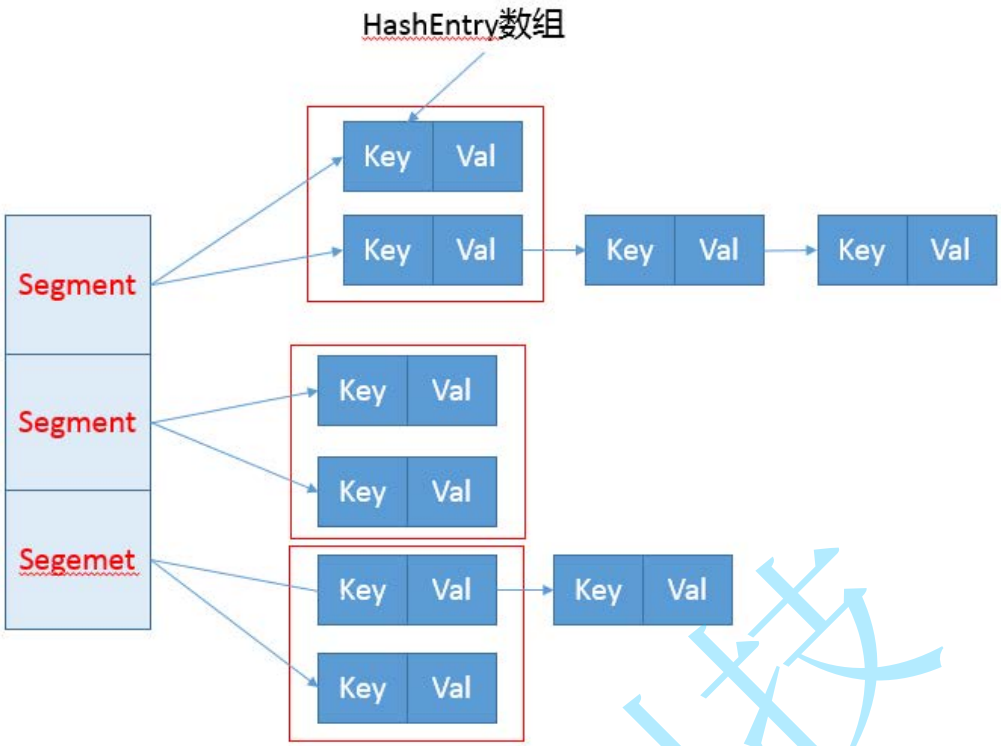
ConcurrentHashMap 分析

ConcurrentHashMap 的设计实现其实一直在演化，比如在 Java 8 中就发生了非常大的变化（Java 7 其实也有不少更新），所以，这里将比较分析结构、实现机制等方面，对比不同版本的主要区别。

早期 ConcurrentHashMap，其实现是基于：

- 分离锁，也就是将内部进行分段（Segment），里面则是HashEntry 的数组，和 HashMap类似，哈希相同的条目也是以链表形式存放。
- HashEntry 内部使用 volatile 的 value 字段来保证可见性，也利用了不可变对象的机制以改进利用 Unsafe 提供的底层能力，比如 volatile access，去直接完成部分操作，以最优化性能，毕竟 Unsafe 中的很多操作都是 JVM intrinsic 优化过的。

可以参考下面这个早期 ConcurrentHashMap 内部结构的示意图，其核心是利用分段设计，在进行并发操作的时候，只需要锁定相应段，这样就有效避免了类似 Hashtable 整体同步的问题，大大提高了性能。



在构造的时候，Segment 的数量由所谓的 concurrentcyLevel 决定，默认是 16，也可以在相应构造函数直接指定。注意，Java 需要它是 2 的幂数值，如果输入是类似 15 这种非幂值，会被自动调整到 16 之类 2 的幂数值。

在进行并发写操作时：

- ConcurrentHashMap 会获取再入锁，以保证数据一致性，Segment 本身就是基于ReentrantLock 的扩展实现，所以，在并发修改期间，相应 Segment 是被锁定的。
- 在最初阶段，进行重复性的扫描，以确定相应 key 值是否已经在数组里面，进而决定是更新还是放置操作。重复扫描、检测冲突是ConcurrentHashMap 的常见技巧。
- 在 ConcurrentHashMap中扩容同样存在。不过有一个明显区别，就是它进行的不是整体的扩容，而是单独对 Segment 进行扩容。

另外一个 Map 的 size 方法同样需要关注，它的实现涉及分离锁的一个副作用。

试想，如果不进行同步，简单的计算所有 Segment 的总值，可能会因为并发 put，导致结果不准确，但是直接锁定所有 Segment 进行计算，就会变得非常昂贵。其实，分离锁也限制了 Map的初始化等操作。

所以，ConcurrentHashMap 的实现是通过重试机制（RETRIES_BEFORE_LOCK，指定重试次数 2），来试图获得可靠值。如果没有监控到发生变化（通过对比 Segment.modCount），就直接返回，否则获取锁进行操作。

下面来对比一下，在 Java 8 和之后的版本中，ConcurrentHashMap 发生了哪些变化呢?

- 总体结构上，它的内部存储变得和HashMap 结构非常相似，同样是大的桶（bucket）数组，然后内部也是一个个所谓的链表结构（bin），同步的粒度要更细致一些。
- 其内部仍然有 Segment 定义，但仅仅是为了保证序列化时的兼容性而已，不再有任何结构上的用处。
- 因为不再使用 Segment，初始化操作大大简化，修改为 lazy-load 形式，这样可以有效避免初始开销，解决了老版本很多人抱怨的这一点。
- 数据存储利用 volatile 来保证可见性。
- 使用 CAS 等操作，在特定场景进行无锁并发操作。

- 使用 Unsafe、LongAdder 之类底层手段，进行极端情况的优化。

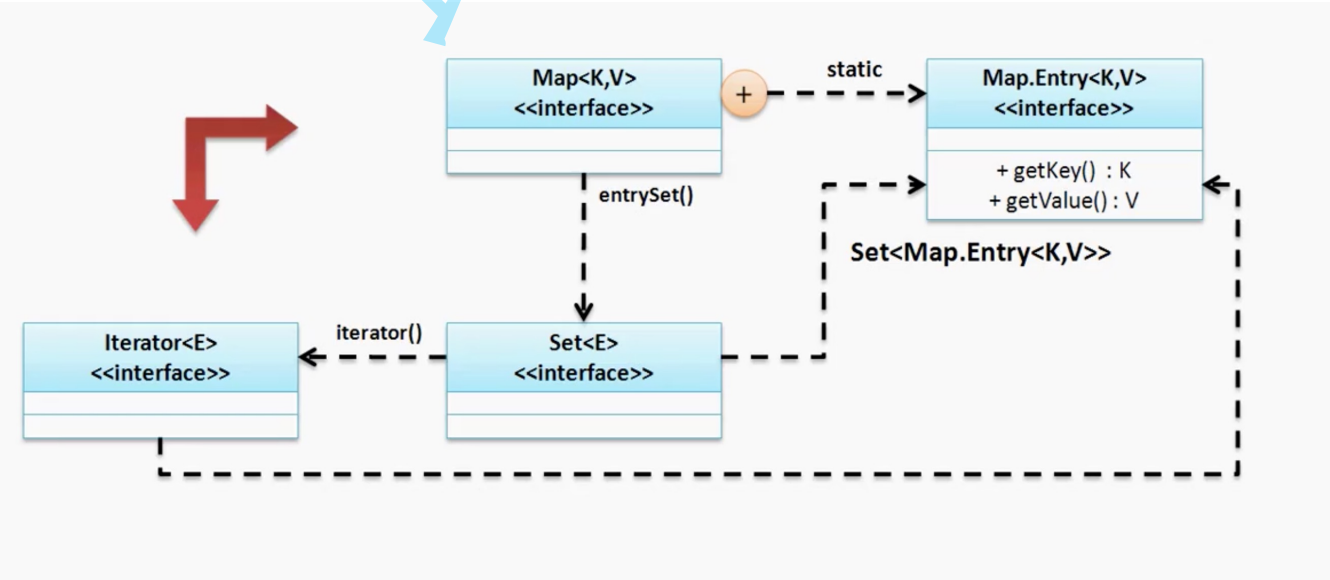
5.5 Map集合使用Iterator输出(重点)

Map接口与Collection接口不同，Collection接口有iterator()方法可以很方便的取得Iterator对象来输出，而Map接口本身并没有此方法。下面我们首先来观察Collection接口与Map接口数据保存的区别：

在Map接口里面有一个重要的方法，将Map集合转为Set集合:

```
public Set<Map.Entry<K, V>> entrySet();
```

Map要想调用Iterator接口输出，走的是一个间接使用的模式，如下图:



范例：通过Iterator输出Map集合

```
package www.bit.java.test;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class TestDemo {
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<>() ;
        map.put(1,"Hello") ;
        map.put(2,"Bit") ;
        map.put(3,"Java") ;
        // 1.将Map集合转为Set集合
        Set<Map.Entry<Integer,String>> set = map.entrySet() ;
        // 2.获取Iterator对象
        Iterator<Map.Entry<Integer,String>> iterator = set.iterator() ;
        // 3.输出
        while (iterator.hasNext()) {
            // 4.取出每一个Map.Entry对象
            Map.Entry<Integer,String> entry = iterator.next() ;
            // 5.取得key和value
            System.out.println(entry.getKey()+" = " +entry.getValue()) ;
        }
    }
}
```

以上就是Map使用Iterator输出的标准代码，需要各位熟练使用。

5.6 关于Map中key的说明

在之前使用Map集合的时候使用的都是系统类作为key(Integer,String等)。实际上用户也可采用自定义类作为key。这个时候一定要记得覆写Object类的hashCode()与equals()方法。

范例：观察自定义类作为Key，系统类作为Value的情况(未覆写)

```
package www.bit.java.test;

import java.util.HashMap;
import java.util.Map;
class Person {
    private Integer age ;
    private String name ;

    public Person(Integer age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "Person{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }

    public Integer getAge() {
        return age;
    }
```

```
    public void setAge(Integer age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Map<Person,String> map = new HashMap<>() ;
        map.put(new Person(15,"张三"),"zs") ;
        System.out.println(map.get(new Person(15,"张三")));
    }
}
```

范例：覆写hashCode()与equals()方法

```
package www.bit.java.test;

import java.util.HashMap;
import java.util.Map;
import java.util.Objects;

class Person {
    private Integer age ;
    private String name ;

    public Person(Integer age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "Person{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(age, person.age) &&
                Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {

        return Objects.hash(age, name);
    }
```

```
    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Map<Person,String> map = new HashMap<>() ;
        map.put(new Person(15,"张三"),"zs") ;
        System.out.println(map.get(new Person(15,"张三")));
    }
}
```

实际开发来讲，我们一般都是采用系统类(String，Integer等)作为Key值，这些系统类都帮助用户覆写好了hashCode()与equals()方法。

5.7 TreeMap子类

TreeMap是一个可以排序的Map子类，它是按照Key的内容排序的。

范例：观察TreeMap的使用。

```
public class TestDemo {
    public static void main(String[] args) {
        Map<Integer,String> map = new TreeMap<>() ;
        map.put(2,"C") ;
        map.put(0,"A") ;
        map.put(1,"B") ;
        System.out.println(map);
    }
}
```

这个时候的排序处理依然按照的是Comparable接口完成的。

结论：**有Comparable出现的地方，判断数据就依靠compareTo()方法完成，不再需要equals()与hashCode()**

Map集合小结：

1. Collection保存数据的目的一般用于输出(Iterator)，Map保存数据的目的是为了根据key查找，找不到返回null。
2. Map使用Iterator输出(Map.Entry的作用)
3. HashMap数据结构一定要理解(链表与红黑树)、HashMap与Hashtable区别

6.栈与队列

6.1 Stack栈

栈是一种先进后出的数据结构。浏览器的后退、编辑器的撤销、安卓Activity的返回等都属于栈的功能。

在Java集合中提供有Stack类，这个类时Vector的子类。需要注意的是，使用这个类的时候使用的不是Vector类中的方法，并且在使用时不要进行向上转型。因为要操作的方法不是由List定义的，而是由Stack定义的。

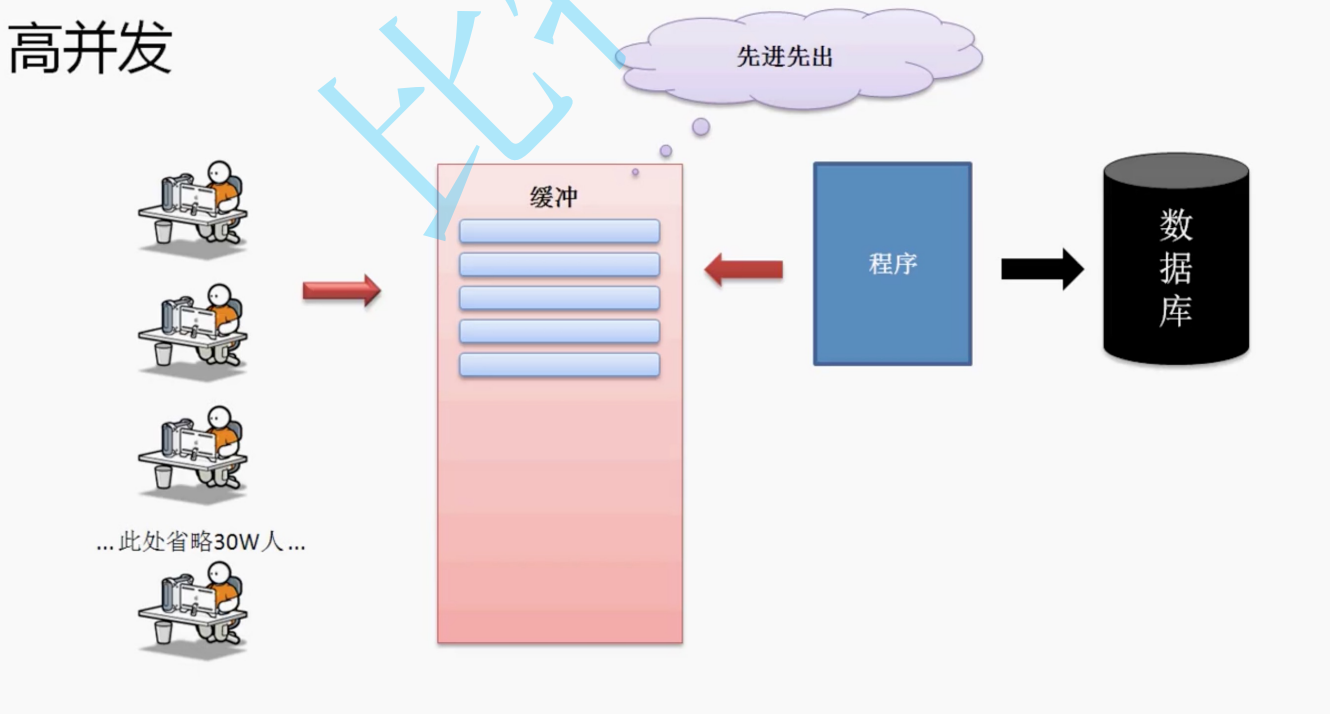
1. 入栈 : public E push(E item)
2. 出栈 : public synchronized E pop()
3. 观察栈顶元素 : public synchronized E peek() 范例：观察出入栈操作

```
package www.bit.java.test;

import java.util.Stack;

public class TestDemo {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>() ;
        stack.push("A") ;
        stack.push("B") ;
        stack.push("C") ;
        System.out.println(stack.peek()) ;
        System.out.println(stack.pop()) ;
        System.out.println(stack.pop()) ;
        System.out.println(stack.pop()) ;
        // EmptyStackException
        System.out.println(stack.pop()) ;
    }
}
```

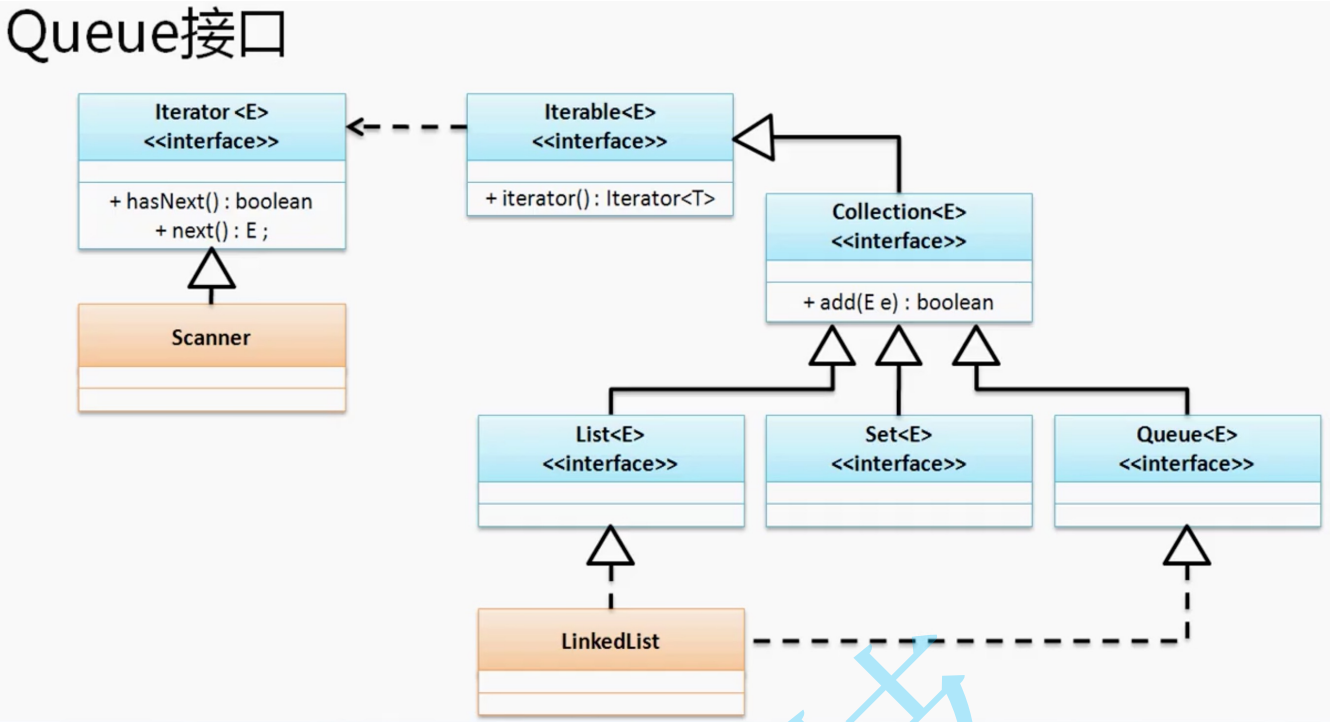
如果栈已经空了，那么再次出栈就会抛出空栈异常。

6.2 Queue队列

Stack是先进后出，与之对应的Queue是先进先出。



在java.util包中使用Queue来实现队列处理操作。Queue接口有一个子类LinkedList。来看Queue接口继承关系:



使用Queue接口主要是进行先进先出的实现，在这个接口里面有如下的方法:

- 按照队列取出内容: public E poll();

范例：观察Queue的poll操作

```
package www.bit.java.test;


import java.util.LinkedList;
import java.util.Queue;

public class TestDemo {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>() ;
        queue.add("A") ;
        queue.add("B") ;
        queue.add("C") ;
        System.out.println(queue.peek()) ;
        System.out.println(queue.poll()) ;
        System.out.println(queue.poll()) ;
        System.out.println(queue.poll()) ;
        // 取完数据继续poll返回null
        System.out.println(queue.poll()) ;
    }
}
```

既然队列在整个的操作之中可以起到一个缓冲的作用，那么可以利用队列修改之前多线程部分讲解的生产者和消费者程序模型。(课后作业)

7.Properties属性文件操作

在java中有一种属性文件(资源文件)的定义：*.properties文件，在这种文件里面其内容的保存形式为"key = value"，通过ResourceBundle类读取的时候只能读取内容，要想编辑其内容则需要通过Properties类来完成，这个类是专门做属性处理的。

Properties是Hashtable的子类，这个类的定义如下:

```
public class Properties extends Hashtable<Object,Object>
```

所有的属性信息实际上都是以字符串的形式出现的，在进行属性操作的时候往往会使用Properties类提供的方法完成

1. 设置属性 : public synchronized Object setProperty(String key, String value)
2. 取得属性 : public String getProperty(String key),如果没有指定的key则返回null
3. 取得属性 : public String getProperty(String key, String defaultValue)，如果没有指定的key则返回默认值

范例：观察属性操作

```
package www.bit.java.test;


import java.util.Properties;

public class TestDemo {
    public static void main(String[] args) {
        Properties properties = new Properties() ;
        properties.setProperty("xa","Xi'An") ;
        properties.setProperty("sh","ShangHai") ;
        System.out.println(properties.get("xa")) ;
        System.out.println(properties.get("bj")) ;
    }
}
```

在Properties类中提供有IO支持的方法:

1. 保存属性: public void store(OutputStream out, String comments) throws IOException
2. 读取属性: public synchronized void load(InputStream inStream) throws IOException

范例：将属性输出到文件

```
package www.bit.java.test;


import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

public class TestDemo {
    public static void main(String[] args) throws IOException {
        Properties properties = new Properties() ;
        properties.setProperty("xa","Xi'An") ;
        properties.setProperty("sh","ShangHai") ;
        File file = new File("/Users/yuisama/Desktop/test.properties") ;
        properties.store(new FileOutputStream(file),"testProperties") ;
    }
}
```

范例：通过属性文件读取内容

```
package www.bit.java.test;


import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;
```

```
public class TestDemo {
    public static void main(String[] args) throws IOException {
        Properties properties = new Properties() ;
        File file = new File("/Users/yuisama/Desktop/test.properties") ;
        properties.load(new FileInputStream(file)) ;
        System.out.println(properties.getProperty("xa")) ;
    }
}
```

Properties只能操作String，它可以进行远程属性内容的加载。

8.Collections工具类

Collections是一个集合操作的工具类，包含有集合反转、排序等操作。

范例：利用Collections进行集合操作

```
package www.bit.java.test;


import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) throws IOException {
        List<String> list = new ArrayList<>() ;
        // 相当于调用了三次add()方法
        Collections.addAll(list,"A","B","C") ;
        System.out.println(list) ;
        // 集合反转
        Collections.reverse(list) ;
        System.out.println(list) ;
    }
}
```

9.Stream数据流

JDK1.8发行的时候实际上是世界上大数据兴起的时候，在整个大数据的开发里面有一个最经典的模型:MapReduce。实际上这属于数据的两个操作阶段：

1. Map: 处理数据
2. Reduce: 分析数据

而在Java类集中，由于其本身的作用就可以进行大量数据的存储，所以就顺其自然的产生了MapReduce操作，而这些操作可以通过Stream数据流来完成。

9.1 Collection改进

从JDK1.8开始，Collection口里面除了定义一些抽象方法外，也提供了一些普通方法，下面来观察如下几个方法：

1. forEach()输出支持: default void forEach(Consumer<? super T> action)
2. 取得Stream数据流对象: default Stream stream()

范例：使用forEach()输出

```
package www.bit.java.test;


import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestDemo {
    public static void main(String[] args)  {
        List<String> list = new ArrayList<>() ;
        Collections.addAll(list,"Java","C++","Python","JavaScript") ;
        // 方法引用
        list.forEach(System.out::println) ;
    }
}
```

Collection接口里提供有一个重要的stream()方法，这个方法是整个JDK1.8中数据操作的关键。

范例：观察Stream

```
package www.bit.java.test;


import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Stream;

public class TestDemo {
    public static void main(String[] args)  {
        List<String> list = new ArrayList<>() ;
        Collections.addAll(list,"Java","C++","Python","JavaScript") ;
        // 实例化Stream对象
        Stream<String> stream = list.stream() ;
        System.out.println(stream.count()) ;
    }
}
```

将集合数据交给Stream之后，就相当于这些数据一个一个进行处理。

9.2 Stream操作

在第一节使用的count()方法是针对数据量做了一个统计操作，除此之外，也可以进行数据的过滤。例如：满足某些条件的内容才允许做数量统计。

范例：数据过滤

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Stream;

public class TestDemo {
    public static void main(String[] args)  {
        List<String> list = new ArrayList<>() ;
        Collections.addAll(list,"Java","C++","Python","JavaScript") ;
        // 实例化Stream对象
        Stream<String> stream = list.stream() ;
```

```
        // 统计出这些数据中带有Java的个数
        System.out.println(stream.filter((e)->e.contains("Java")).count());
    }
}
```

现在加入希望在数据过滤后得到具体数据，就可以使用收集器来完成。

- 收集器:public <R, A> R collect(Collector<? super T, A, R> collector)

范例：收集器

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestDemo {
    public static void main(String[] args)  {
        List<String> list = new ArrayList<>() ;
        Collections.addAll(list,"Java","C++","Python","JavaScript") ;
        // 实例化Stream对象
        Stream<String> stream = list.stream() ;
        // 收集过滤后的数据
        System.out.println(stream.filter((e)->e.contains("Java"))
                .collect(Collectors.toList()));
    }
}
```

收集完的数据依然属于List集合，所以可以直接使用List进行接收。

范例：使用List集合接收过滤后的数据

```
// 收集过滤后的数据
List<String> resultList = stream.filter((e)->e.contains("Java"))
    .collect(Collectors.toList()) ;
System.out.println(resultList);
```

在Stream接口中重点有两个操作方法:

1. 设置取出最大内容: public Stream limit(long maxSize);
2. 跳过的数据量: public Stream skip(long n);

范例：使用skip与limit方法

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class TestDemo {
    public static void main(String[] args)  {
        List<String> list = new ArrayList<>() ;
        Collections.addAll(list,"1、Java","2、C++","3、Python","4、JavaScript","5、Nginx","6、
Tomcat") ;
        // 实例化Stream对象
```

```
        Stream<String> stream = list.stream() ;
        List<String> resultList = stream.skip(0).limit(3)
                .map((s)->s.toUpperCase())
                .collect(Collectors.toList()) ;
        System.out.println(resultList);
    }
}
```

以上代码就使用了skip()、limit()方法对数据做了分页处理，还结合了map()方法做了简单的数据处理。

9.3 MapReduce模型

MapReduce是整个Stream的核心所在。MapReduce的操作也是由两个阶段所组成：

1. map():指的是针对于数据进行先期的操作处理。例如：简单的数学运算等
2. reduce():进行数据的统计分析。

范例：编写一个简单的数据统计操作

```
class Order {
    private String title ;
    private double price ;
    private int amount ;

    public Order(String title, double price, int amount) {
        this.title = title;
        this.price = price;
        this.amount = amount;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getAmount() {
        return amount;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }
}
```

随后在List集合中保存这些订单的信息。

范例：实现订单信息保存随后进行总量的统计

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.List;

public class TestDemo {
    public static void main(String[] args) {
        List<Order> orderList = new ArrayList<>();
        orderList.add(new Order("Iphone", 8999.99, 10));
        orderList.add(new Order("外星人笔记本", 12999.99, 5));
        orderList.add(new Order("MacBookPro", 18999.99, 5));
        orderList.add(new Order("Java从入门到放弃.txt", 9.99, 20000));
        orderList.add(new Order("中性笔", 1.99, 200000));
        double allPrice = orderList.stream().map((obj) -> obj.getPrice() * obj.getAmount())
                .reduce((sum, x) -> sum + x).get();
        System.out.println("所花费的总数额为: "+allPrice);
    }
}
```

为了进一步观察更加丰富的处理操作，可以再做一些数据的统计分析。对于当前的操作如果要进行一些数量的统计，其最终的结果应为double型数据。在Stream接口中就提供有一个map结果变为Double型的操作:

- 统计分析: public DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);

此时的方法返回的是一个DoubleStream接口对象，这里面就可以完成统计操作，这个统计使用的方法如下:

- 统计方法:DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);

范例：统计分析

```
package www.bit.java.test;

import java.util.ArrayList;
import java.util.DoubleSummaryStatistics;
import java.util.List;

class Order {
    private String title;
    private double price;
    private int amount;

    public Order(String title, double price, int amount) {
        this.title = title;
        this.price = price;
        this.amount = amount;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
```

```
    }

    public int getAmount() {
        return amount;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }
}

public class TestDemo {
    public static void main(String[] args) {
        List<Order> orderList = new ArrayList<>();
        orderList.add(new Order("Iphone", 8999.99, 10));
        orderList.add(new Order("外星人笔记本", 12999.99, 5));
        orderList.add(new Order("MacBookPro", 18999.99, 5));
        orderList.add(new Order("Java从入门到放弃.txt", 9.99, 20000));
        orderList.add(new Order("中性笔", 1.99, 200000));
        DoubleSummaryStatistics dss = orderList.stream().mapToDouble((obj) -> obj.getPrice() *
obj.getAmount())
                .summaryStatistics();
        System.out.println("总量:  " + dss.getCount());
        System.out.println("平均值:  " + dss.getAverage());
        System.out.println("最大值:  " + dss.getMax());
        System.out.println("最小值:  " + dss.getMin());
        System.out.println("总和:  " + dss.getSum());
    }
}
```

以上代码就是Java1.8中提供的大数据的相关操作(入门)