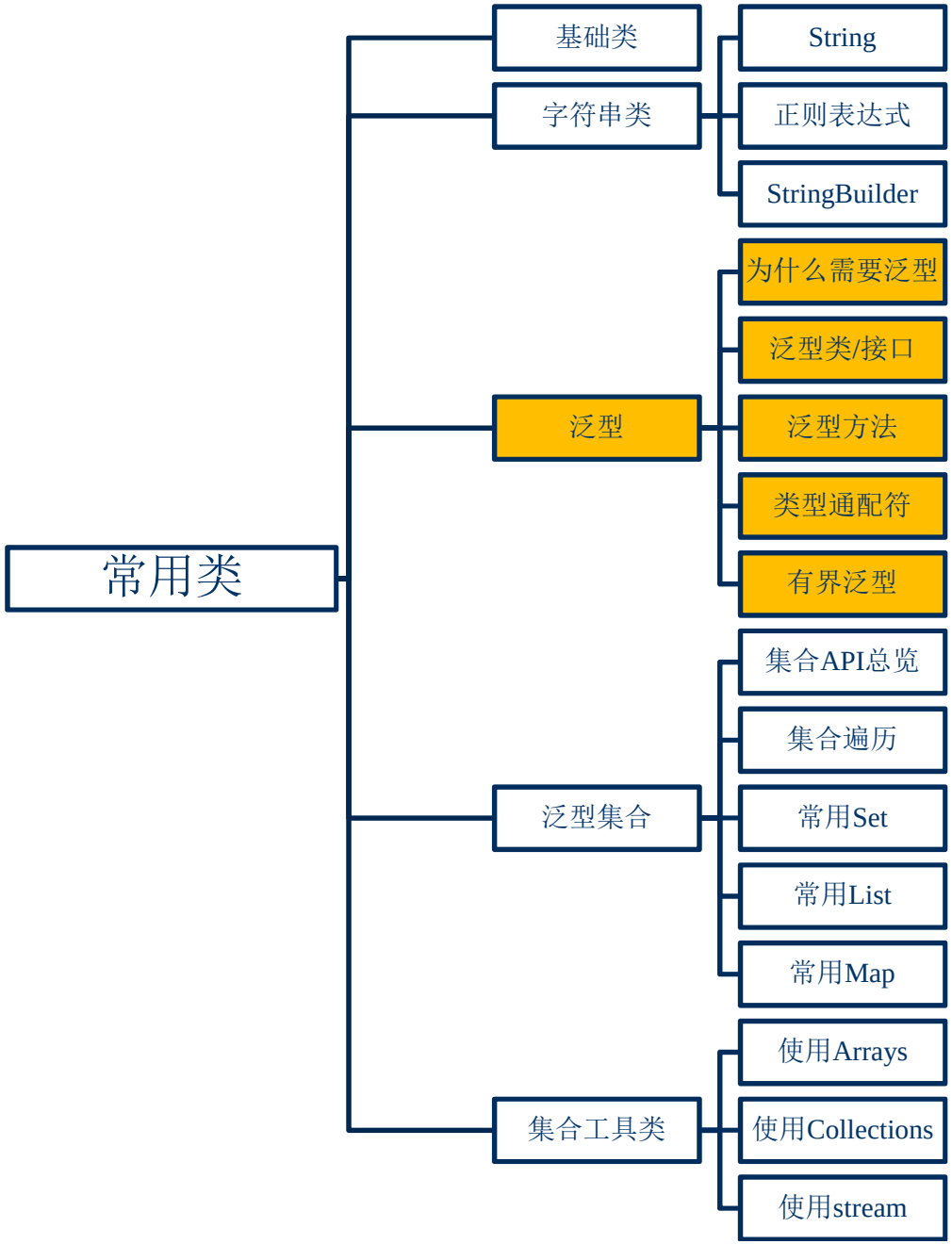
第七章 常用类(1)

第七章 常用类


常用类
基础类
String
字符串类
正则表达式
StringBuilder
泛型
为什么需要泛型
泛型类/接口
泛型方法
类型通配符
有界泛型
泛型集合
集合API总览
集合遍历
常用Set
常用List
常用Map
集合工具类
使用Arrays
使用Collections
使用stream

7.3 泛型（需要学习泛型设计的，学习本节）

- 泛型（Generics）是JDK1.5引入的特性，允许在定义类、接口、方法时由原来的具体类型变成参数形式，而在实例化对象时，再指定具体的类型即可。

- 泛型主要应用在集合框架中
  - 可提高程序的类型安全：利用泛型定义的变量，编译器可验证类型。
  - 泛型有助于避免转型错误。

7.3.1 为什么需要泛型

Jdk1.5之前，为了实现参数类型的任意化，都是通过Object类型来处理，其缺点是使用对象时需要强制转换,容易出现ClassCastException

JDK1.5之前实现泛化举例

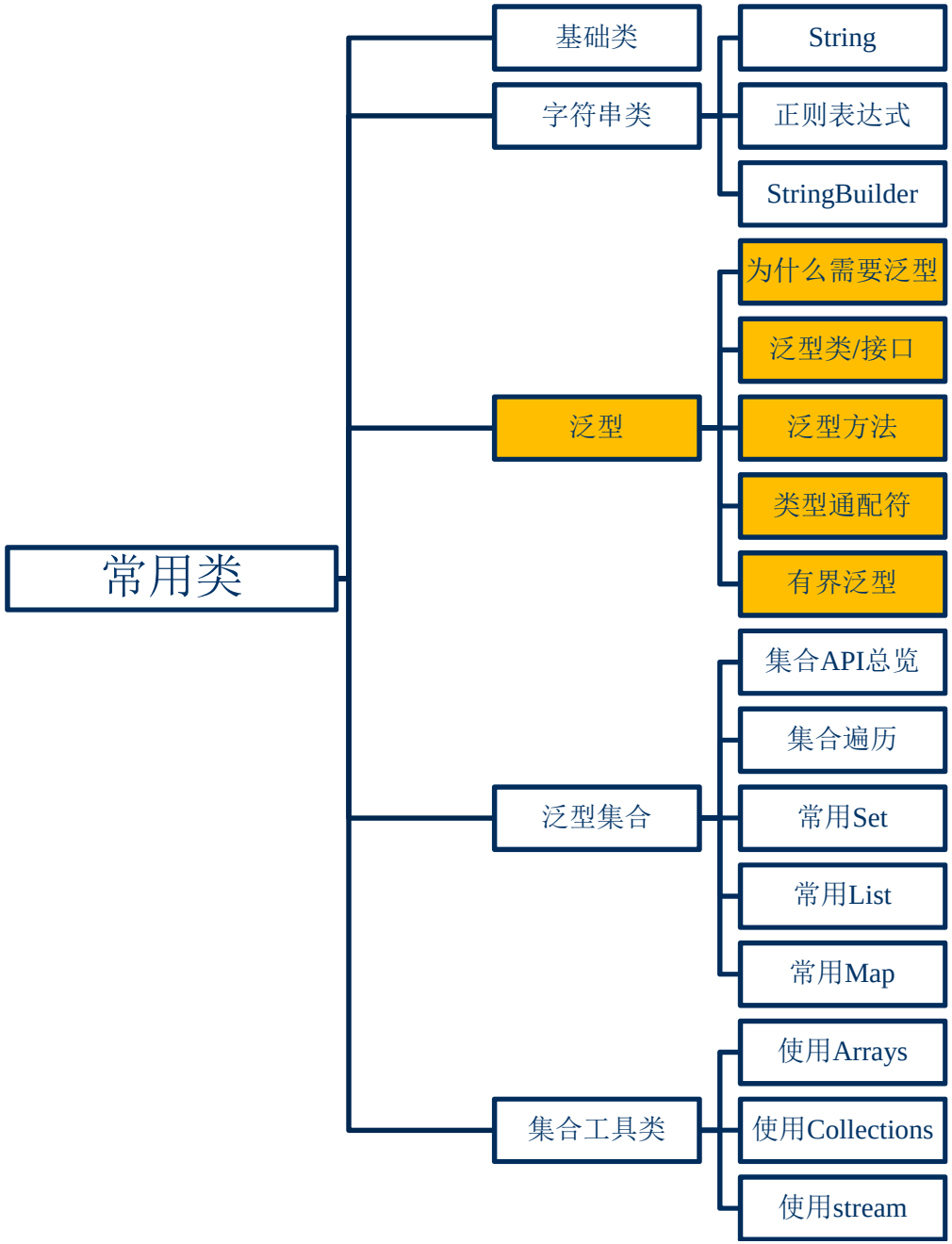
```
class Point{
 private Object x;
 private Object y;
 public void setX(Object x){ this.x=x; }
 public void setY(Object y){ this.y=y;}
 public Object getX(){ return this.x;}
 public Object getY(){  return this.y;}
}
```

```
public class GenDemo02{
 public static void main(String arg[]){
 Point p=new Point();
 p.setX("东经180度");
 p.setY(20); //利用自动装箱操作int-->Integer-->Object
 int x2=(Integer)p.getX();
 int y2=(Integer)p.getY(); }}
```

出现运行时错误

ClassCastException

第七章 常用类


常用类
基础类
String
字符串类
正则表达式
StringBuilder
泛型
为什么需要泛型
泛型类/接口
泛型方法
类型通配符
有界泛型
泛型集合
集合API总览
集合遍历
常用Set
常用List
常用Map
集合工具类
使用Arrays
使用Collections
使用stream

7.3.2 泛型类/接口

1、泛型类/接口定义：

```
[修饰符]  class/interface  名称<类型参数1,类型参数2,…>{
//类体
}
```

泛型的限制

- 1、泛型类型参数，只支持引用类型，不支持简单类型
- 2、静态成员数据类型不能声明为泛型,同样，static方法也不能操作类型待定的泛型对象。

```
public class test3<T> {
    static T ob;//Error
}//在类中共享，而泛型类型是不确定的，编译器无法确定要使用的类型。
```

- 3、泛型参数只是占位符，不能直接实例化，如 new T()

```
public class test3<T>
{ T ob; test3(){
  ob = new T();//Error
} }//编译器不知道要创建那种类型的对象
```

- 4、泛型不能继承Throwable及其子类，即泛型类不能是异常类。

```
class Point<T>{
        private T var ;//合法
        static T ob;    //非法，违反规则2
        private T var2=new T();//非法，违反规则3

        public T getVar(){
                return var ;
                }
        public void setVar(T var){
                this.var = var ;
    }
        public  Point<T> getInstance() { //合法
                return new Point<T>();  //合法
            }
//非法，返回值需要实例化后确定泛型的具体类型，方法不能是static的，
违反规则2。
  public static Point<T> getInstance2(){ //非法
                return new Point<T>();
        }};
```

2、泛型使用

- 类名<具体类> 对象名=new 类名<具体类>([构造参数列表])
- 类名<具体类> 对象名=new 类名<>([构造参数列表])

【JAVA7之后，简化】

- 【示例】

```
Point<String> mynode=new Point<String>();
Point<String> mynode=new Point<>();
```

说明：

(1)一个类的子类可通过向上转型，实例化成父类类型，但是在泛型操作中，子类的泛型对象是无法赋给父类泛型引用的。因为这样会使子类泛型完全失去意义。例如：

```
Point<Object> Obj=new Point<String>();
//非法编译出错
```

（2）类型擦除：泛型使用时，如果没有指定泛型的类型，会将可变类型设置成Object，这样一来就可以接收任意的数据类型，所有的泛型信息将被擦除，恢复到jdk5之前的用法，不提倡。

```
Point Obj=new Point();  Obj.setData(10);
```

```
class Generic<T>{
        private T data;
        public Generic(){}
        public Generic(T data){this.data=data;}
        public void setData(T data){this.data=data;}
        public T getData(){
                System.out.println("类型是: "+data.getClass().getName());
                return data;
                }
        }
public class GenericDemo3 {
        public static void main(String arg[]){
                Generic<String> strObj=new Generic<String>("GenTest");

                strObj.setData(10);//非法，编译时错误，T已声明成String.
                System.out.println(strObj.getData());

    //规则1：非法，T类型的前后不统一。
                Generic<Object> strObj=new Generic<String>();

    //规则2，合法，但类型被擦除,不提倡
                Generic strObj2=new Generic();
                strObj2.setData(10);
                System.out.println(strObj2.getData());
                }
```

减少了类型转换的操作代码，而且更加安全，如果设置内容类型不对，在编译时就出错

3、泛型接口的实现

在实现泛型接口时，应该声明与接口相同的类型参数。用法如下：

```
interface 接口名<类型参数列表>{
//...
}
```

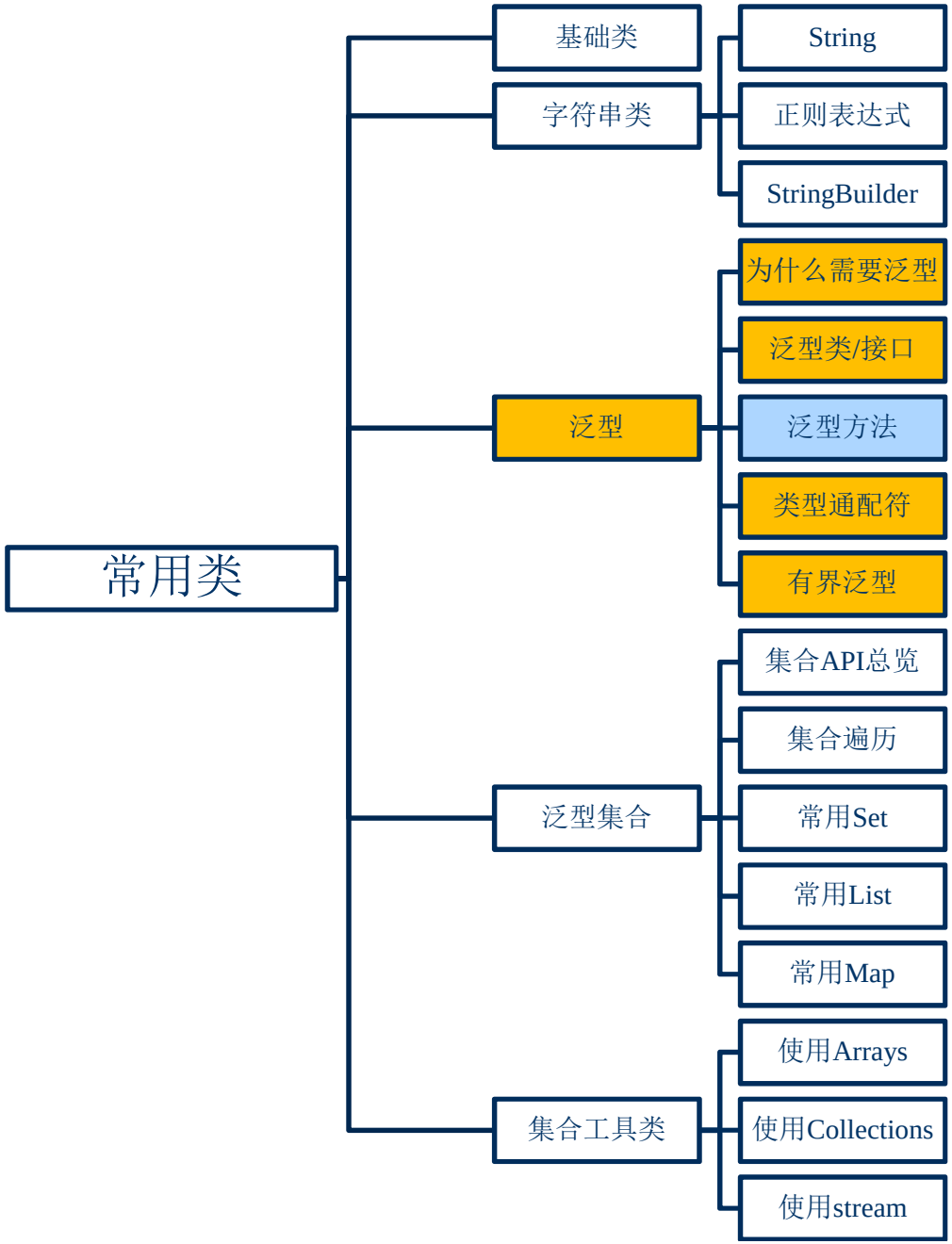
实现上述接口

```
class 类名<类型参数列表> implements 接口名<类型参数列表>{
//
}
```

例如：

```
interface ABC<K,V>{
//方法声明
}
class myABC<K,V> implements ABC<K,V>{
//方法实现
}
```

第七章 常用类


常用类
基础类
String
字符串类
正则表达式
StringBuilder
泛型
为什么需要泛型
泛型类/接口
泛型方法
类型通配符
有界泛型
泛型集合
集合API总览
集合遍历
常用Set
常用List
常用Map
集合工具类
使用Arrays
使用Collections
使用stream

7.3.3 泛型方法

除了定义一个泛型类，还可以只定义一个带类型参数的简单方法，可定义在普通类或者是泛型类中，如：

1、定义

[修饰符] <类型参数1，类型参数2,...> 返回值 方法名（参数列表）{

//方法体

}

```
class ArrayAlg{
public static <T> T getMiddle(T... a){
return a[a.length/2];
}}
```

- 注意：类型变量放在修饰符的后面(public static)
- 对比非法的 **public static** Point<T> getInstance2(){ //非法

  return new Point<T>();}};

7.3.3 泛型方法

除了定义一个泛型类，还可以只定义一个带类型参数的简单方法，可定义在普通类或者是泛型类中，如：

1、定义

[修饰符] <类型参数1，类型参数2,...> 返回值 方法名（参数列表）{

//方法体

}

```
class ArrayAlg{
public static <T> T getMiddle(T... a){
return a[a.length/2];
}}
```

- 注意：类型变量放在修饰符的后面(public static)
- 对比非法的 **public static** Point<T> getInstance2(){ //非法

  return new Point<T>();}};

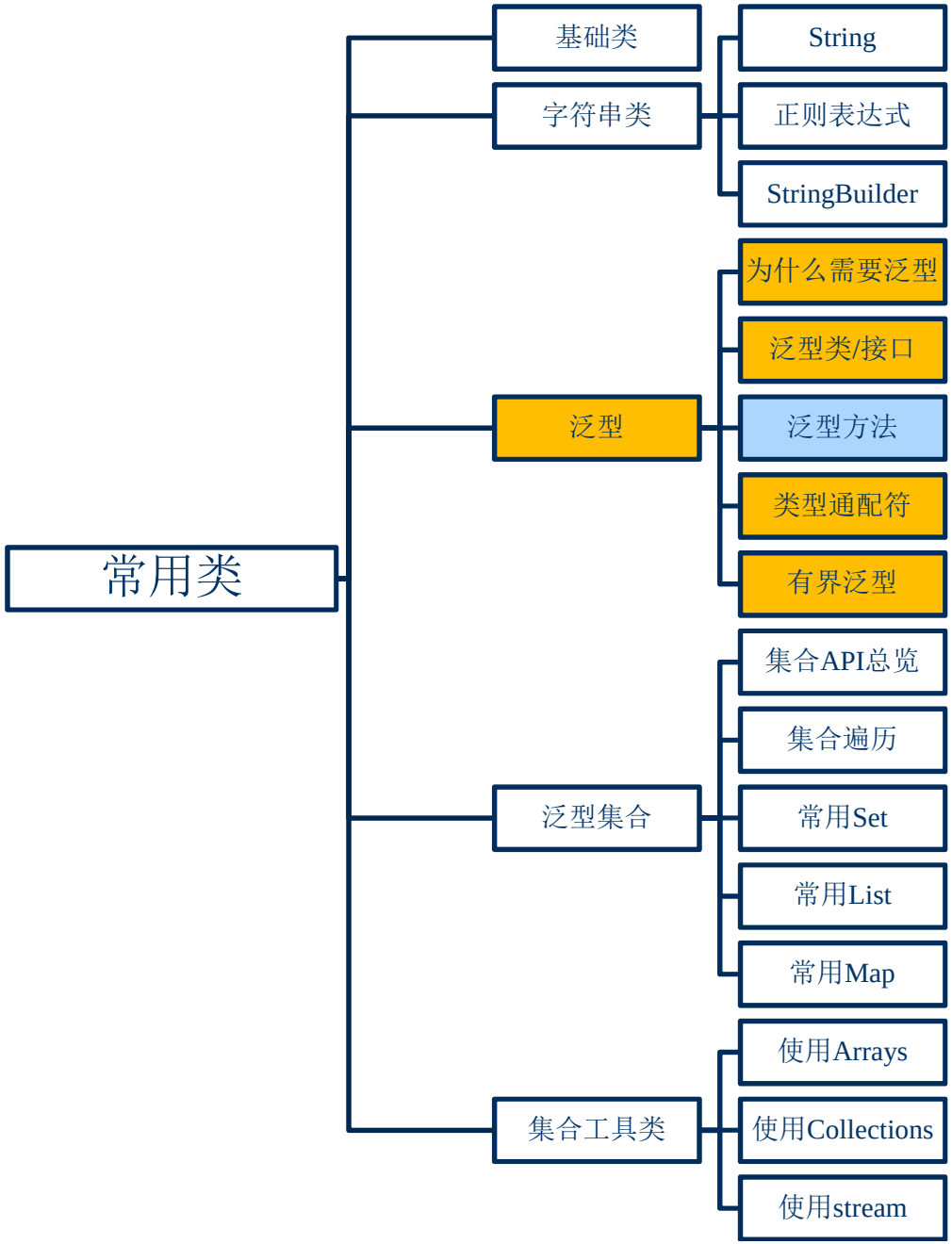
7.3.3 泛型方法

- 调用:
  - ✓ ArrayAlg.<String>getMiddle("aa","bb","c");
  - ✓ ArrayAlg.getMiddle("aa","bb","c");
  - ✓ ArrayAlg.getMiddle(5,6,7);

(编译器有足够的信息能推导出类型)

```
class ArrayAlg{
public static <T> T getMiddle(T... a){
return a[a.length/2];}}
```

第七章 常用类


常用类
基础类
String
字符串类
正则表达式
StringBuilder
泛型
为什么需要泛型
泛型类/接口
泛型方法
类型通配符
有界泛型
泛型集合
集合API总览
集合遍历
常用Set
常用List
常用Map
集合工具类
使用Arrays
使用Collections
使用stream

7.3.4 类型通配符

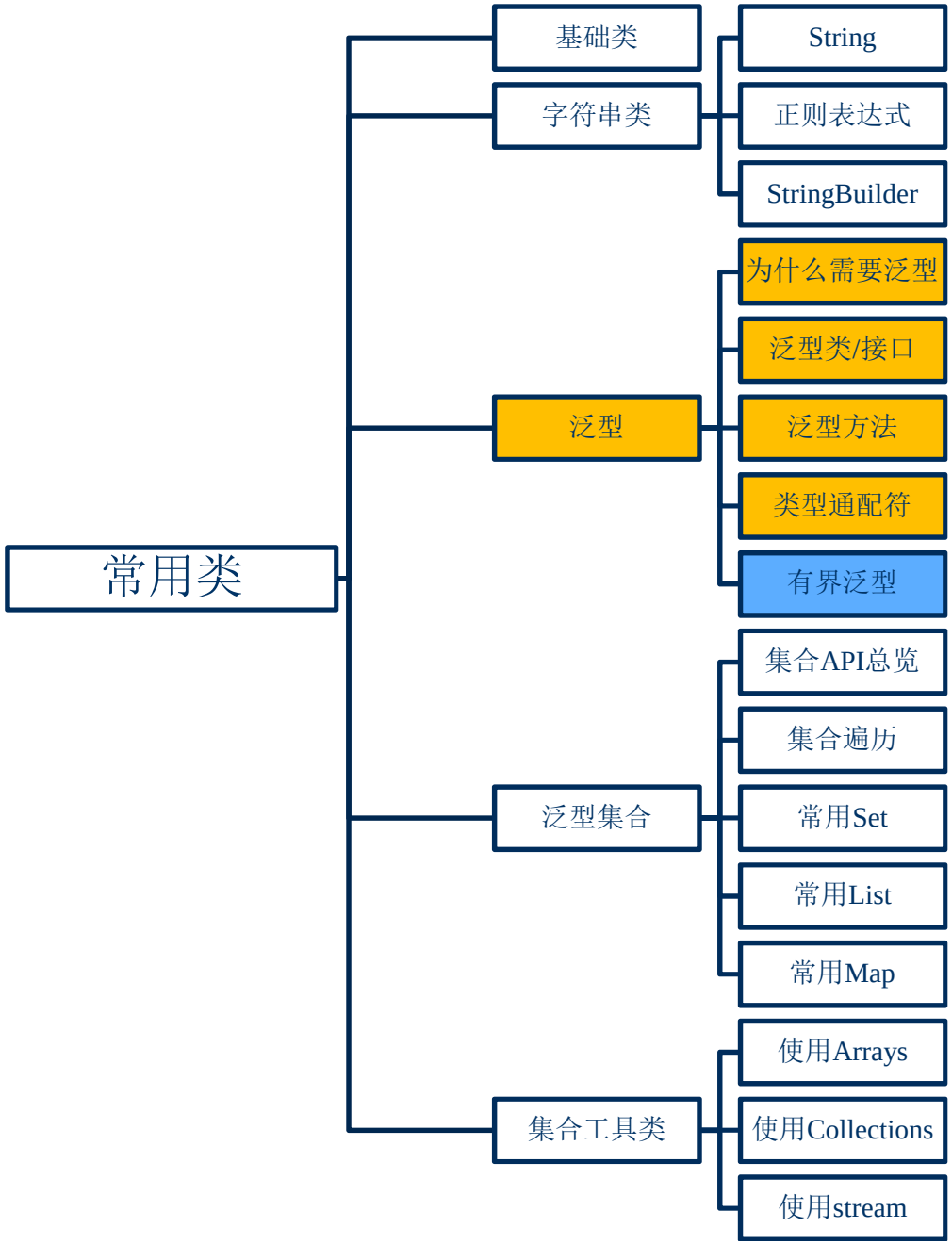
到目前为止，当我们使用泛型类创建对象时，都应该为泛型确定一个确切类型，但有些时候，比如定义方法时，如果我们使用泛型类做形参时，还不想确定类型，此时就需要通配符“？”。

```
class UseGenericDemo{
    public static void myMethod(Generic<?> g){
        System.out.println(g.getData());
              }
    public static void main(String arg[]){
        Generic<Integer> gint=new Generic<Integer>(12);
        Generic<Double> dOb=new Generic<Double>(3.14);
        myMethod(gint);
        myMethod(dOb);
}}
```

注意：

通配符是在使用泛型时出现，当真正调用myMetho方法时，需要传入具体类型的对象。

第七章 常用类


常用类
基础类
String
字符串类
正则表达式
StringBuilder
泛型
为什么需要泛型
泛型类/接口
泛型方法
类型通配符
有界泛型
泛型集合
集合API总览
集合遍历
常用Set
常用List
常用Map
集合工具类
使用Arrays
使用Collections
使用stream

7.3.5 有界泛型

泛型类型有时候需要对类型参数的取值进行一定程度的限制，Java提供了两种有界类型限制参数范围。

- 使用extends 关键字声明类型参数的上界
- 使用super关键字声明类型参数的下界

上界：extends 指定类型

泛型类型须是指定类本身或其子类。

【用法】

1、定义时限定

[修饰符] class 类名<类型参数 extends 父类或接口>{}

2、使用时限定

泛型类<? extends 父类或接口>

- 如有一个父类和多个接口限制可用& 分割，父类放第一个。

【示例】

```
1、class Generic<T extends Number>{//类体}
2、class Generic<T extends Number & Serializable>
3、void myMethod(Generic<? extends Number> g)
```

```
class BoundGenDemo{
        public static void myMethod(Generic<? extends Number> g){
                System.out.println(g.getData());
    }
        public static void main(String arg[]){
        1. Generic<String> gint=new Generic<>("abc");
        2. Generic<Double> dOb=new Generic<>(3.14);
        3. myMethod(gint); //非法，字符串不是Number类形
        4. myMethod(dOb); // 合法
        }
}
```

下界：super

限制此类型必须是指定类型本身或其父类

【用法】

1、下界通常在：使用时限定

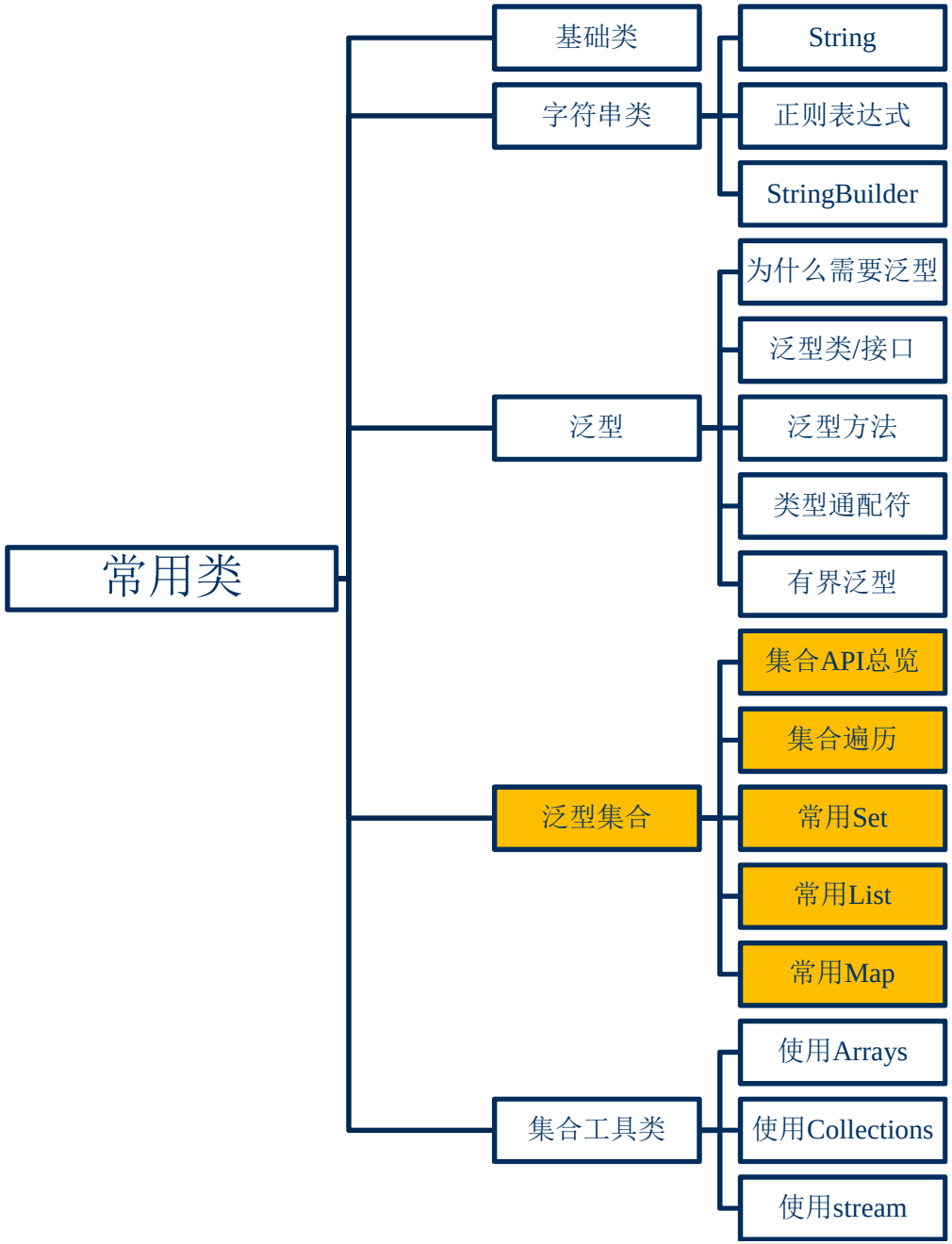
泛型类<? super 类型>

【示例】

```
void myMethod(Generic<? super Person> g)
    //泛型必须是Person类型，或其父类Object.
```

【注意】super比较少用。

第七章 常用类


常用类
基础类
String
字符串类
正则表达式
StringBuilder
泛型
为什么需要泛型
泛型类/接口
泛型方法
类型通配符
有界泛型
泛型集合
集合API总览
集合遍历
常用Set
常用List
常用Map
集合工具类
使用Arrays
使用Collections
使用stream

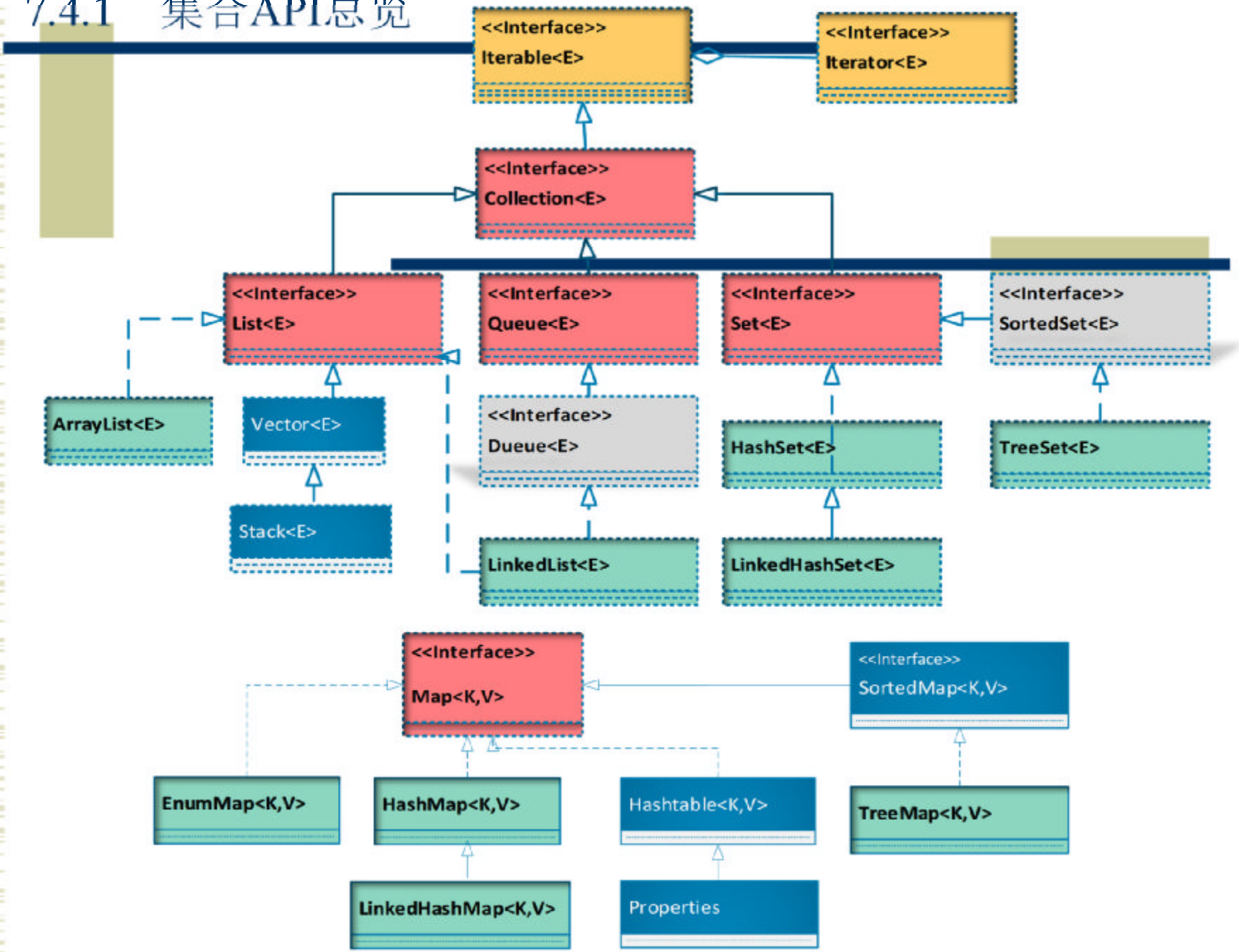
7.4 泛式数据结构

- **Collection接口**
  - **List：** 列表可以包含重复的元素。：如：`ArrayList LinkedList`
  - **Set：** Set表示的是不重复元素的集合：如：`HashSet TreeSet`
  - **Queue：** 队列从操作上可理解为一种操作受限的列表，但队列接口继承自Collection而不是List。
- **Map：**

  元素包含一个键（key）值（value）对，如

  `HashMap TreeMap`

7.4.1 集合API总览


<<Interface>>
Iterable<E>
<<Interface>>
Iterator<E>
<<Interface>>
Collection<E>
<<Interface>>
List<E>
<<Interface>>
Queue<E>
<<Interface>>
Set<E>
<<Interface>>
SortedSet<E>
ArrayList<E>
Vector<E>
<<Interface>>
Dueue<E>
HashSet<E>
TreeSet<E>
Stack<E>
LinkedList<E>
LinkedHashSet<E>
<<Interface>>
Map<K,V>
<<Interface>>
SortedMap<K,V>
EnumMap<K,V>
HashMap<K,V>
Hashtable<K,V>
TreeMap<K,V>
LinkedHashMap<K,V>
Properties

7.4.2 集合遍历

集合类在使用过程中，经常需要遍历每一个元素

1、 增强型for循环

2、 forEach方法

3、 迭代器Iterator

4、 使用普通的for循环

```java
import java.util.ArrayList;
import java.util.Iterator;
public class listDemo {
 public static void main(String arg[]) {
  ArrayList<String> a1=new ArrayList<String>();
  a1.add("NO.1");
  a1.add("NO.2");
  a1.add ("NO.3");
  a1.remove("NO.2");          //a1.remove(1);

  for(String e0:a1){          //方法1：增强式for循环
  System.out.println(e0);
  }
 a1.forEach(o->{System.out.println(o);}); //方法2：foreach方法

  Iterator<String> e=a1.iterator();  //方法3：迭代器Iterator:
   while(e.hasNext())
     System.out.println(e.next());
}

  for(int i=0;i<a1.size();i++)      //方法4：普通的for循环
       System.out.println(a1.get(i));
}
```

```
public class UseMap {
  public static void main(String arg[]) {
    Map<Integer,String> hmap=new HashMap<>();
    Map<Integer, String> tmap=new TreeMap<>();
    int temp;
    for(int i=0;i<4;i++) {
      temp=(int)(Math.random()*30);
      hmap.put(temp, "HashMap"+temp);
      tmap.put(temp, "TreeMap"+temp);
     }
    System.out.println(hmap);
    System.out.println(tmap);

    System.out.println("所有的值");
    for(String str:hmap.values())
        System.out.print(str+", ");

    System.out.printf("\n哈希映射的键-值对\n");
    for(Map.Entry<Integer, String> en: hmap.entrySet())
      System.out.printf("键: %d 值: %s \n ",en.getKey(),en.getValue());

    System.out.printf("\n树映射的 键-值对\n");
    for(Integer key: tmap.keySet())
       System.out.printf("键: %d, 值: %s \n",key, tmap.get(key));
       }}
```

【运行一次的结果】

```
{24=HashMap24, 25=HashMap25, 10=HashMap10, 14=HashMap14}
{10=TreeMap10, 14=TreeMap14, 24=TreeMap24, 25=TreeMap25}
```

所有的值

```
HashMap24, HashMap25, HashMap10, HashMap14,
```

哈希映射的键-值对

```
24---HashMap24; 25---HashMap25; 10---HashMap10; 14---HashMap14;
```

树映射的 键-值对

```
10---TreeMap10; 14---TreeMap14; 24---TreeMap24; 25---TreeMap25;
```

7.4.6 遗留容器类

尽量不要用

1、Vector

Vector实现了List接口，大致功能也和ArrayList相似，当不要求线程安全时，应选择ArrayList，如果要求线程安全，往往会选择CopyOnWriteArrayList代替Vector。

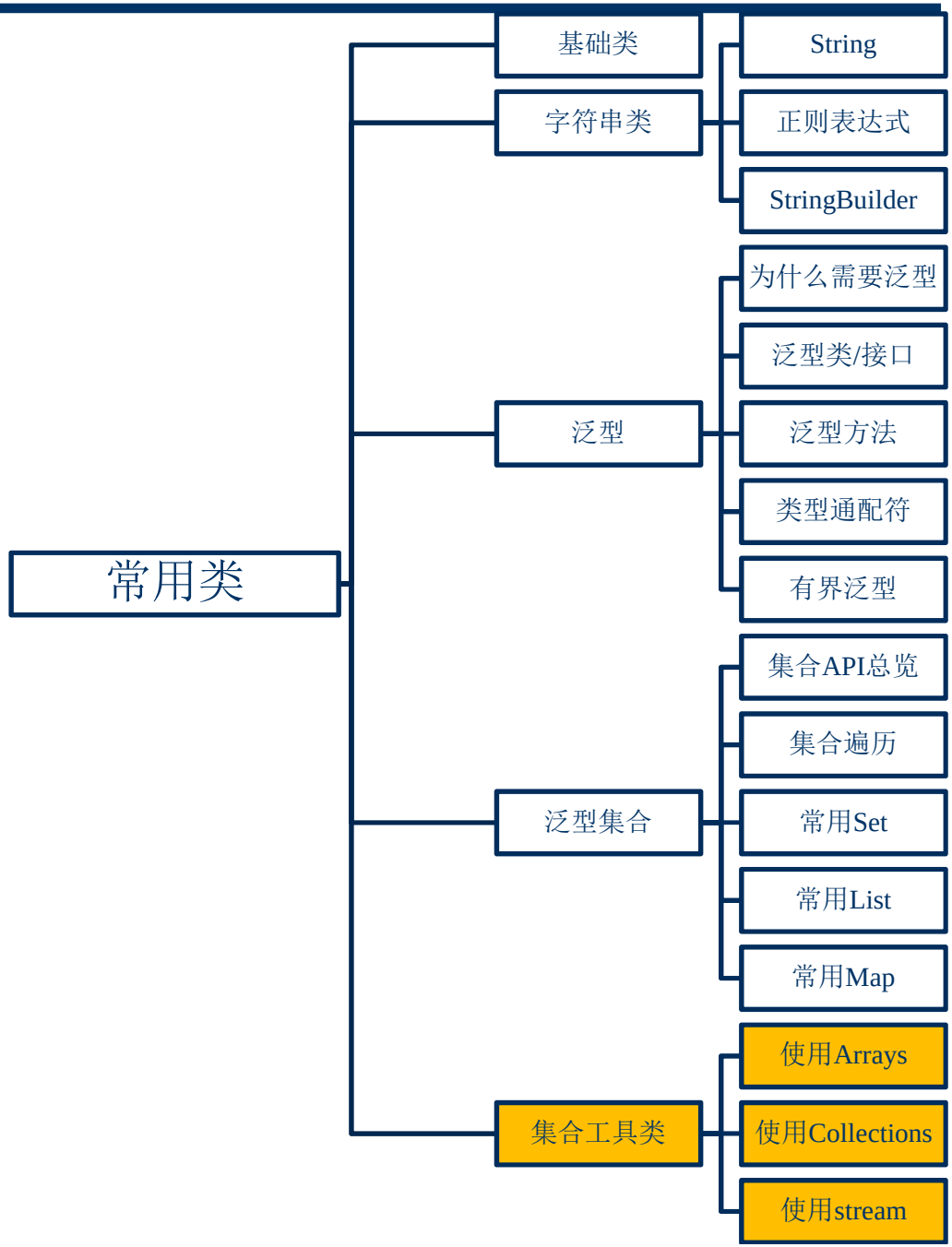
2、Stack

Stack是Vector的子类，表示具有后进先出特性（LIFO :Last In First Out）的栈。虽然线程安全，但是并发性不高。当不要求线程安全时，建议选择LinkedList类，

3、Hashtable

存储键值对的容器。当面对不要求线程安全的应用场景时我们会用HashMap或TreeMap代替，如果要求线程安全，可以使用ConcurrentHashMap类。

第七章 常用类


常用类
基础类
String
字符串类
正则表达式
StringBuilder
泛型
为什么需要泛型
泛型类/接口
泛型方法
类型通配符
有界泛型
泛型集合
集合API总览
集合遍历
常用Set
常用List
常用Map
集合工具类
使用Arrays
使用Collections
使用stream

7.5 集合工具类

在实际应用中，我们经常需要对容器或数组进行整体操作，如：

查找、替换、排序、反转。Java容器框架提供了工具类如：

Arrays

Collections

流Stream，

这些工具类提供了一系列静态方法，用以实现上述操作。

了解

表 7-15 Arrays 的常用 API

分类	方法	功能
获取列表	List<T> asList(T... a)	利用可变长序列获得一个列表
排序	void sort(xxx[] a, [int from, int to])	对数据进行升序排序，数据类型可以是除 boolean 类型的基础类型和对象类型（需实现 Comparable 接口），排序范围可选。
查找	int binarySearch(xxx[] a, xxx key)	利用二分查找法在数组 a 中找元素 key，查找之前数组 a 需先进行排序。
比较	int compare(xxx[] a, xxx[] b, [Comparator<? super T> cmp])	比较两数组的大小，a>b 结果为正数，a=b 为 0，a<b 结果为负数，比较器 cmp 可选。
	boolean equals(xxx[] a, xxx[] a2)	比较两个数组是否相等
复制	xxx[] copyOf(xxx[] or, int Length)	拷贝数组 or，length 指定副本长度，若 length 小于 or 的长度，则返回 length 个元素，若大于则用 or 类型的默认值填充
填充	void fill(xxx[] a, xxx val)	用 val 填充数组 a 的每个元素
获取流	xxxStream stream(xxx[] array)	以数组 array 为数据源返回其序列流

```java
import java.util.Arrays;
public class UseArrays {
   public static void main(String arg[]) {
        String[] s1=new String[5];
         for(int i=0;i<s1.length;i++) //随机生成数组元素“arrayX”
        s1[i]="array"+(int)(Math.random()*10);
         for(String e0:s1)   //打印生成的元素
        System.out.print(e0+"; ");
         Arrays.sort(s1); //用默认的字符串比较方法进行排序
         System.out.println("\n排序后的s1结果是");
         for(String e1:s1)  //打印排序后的数组元素
        System.out.print(e1+", ");
         System.out.println("\n查询 array0");
         int position=Arrays.binarySearch(s1, "array0");
         System.out.println(position);
         String[] c1=Arrays.copyOf(s1, s1.length+1);
         System.out.print("复制后的数组c1为");
         for(String e1:c1)
            System.out.print(e1+", ");
         System.out.printf("\ns1与c1比较结果为%d",Arrays.compare(s1, c1));
```

```
array5; array9; array0; array2; array2;
排序后的s1结果是
array0, array2, array2, array5, array9,
查询 array0
0
复制后的数组c1为 array0, array2, array2, array5, array9, null,
s1与c1比较结果为-1
```

7.5.2 使用Collections（了解）

表 7-16 Collections 的常用 API

分类	方法	功能
排序	void sort(List list)	根据元素的自然顺序对列表 List 按升序进行排序。
	void sort(List<T> list, Comparator<? super T> c)	对列表中元素按照比较器规则排序
查找	int binarySearch(List<? extends Comparable<? super T>> list, T key)	用二分查找算法在列表 list 中搜索指定元素 key。调用此方法前要保证列表有序
	int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)	二分查找列表 list 中指定元素 key。前提：保证列表已经利用比较器 c 进行升序排列
打乱	void shuffle(List<?> list)	将列表 list 的元素随机打乱
逆序	void reverse(List<?> list)	反转列表 list 中的所有元素
填充	void fill(List<? super T> list, T obj)	让列表 list 每个元素为 obj
相交	boolean disjoint(Collection<?> c1, Collection<?> c2)	判断 c1 和 c2 是否不相交
复制	void copy(List<? super T> dest, List<? extends T> src)	将列表 src 复制到列表 dest
最值	T min(Collection<? extends T> coll)	返回集合 coll 最小值
	T max(Collection<? extends T> coll)	返回集合 coll 最大值

```java
class Student implements Comparable<Student>{
        private int level;  //级别
        private String name;//姓名
        private int age;//年龄
//带参构造方法
        public Student(String name,int age,int level) {
                this.name=name;
                this.age=age;
                this.level=level;
        }
        public String getName() {return name;}

        public int getAge() {return age;}

        public int getLevel() {return level;}
        @Override
        //实现Comparable<Student>中的接口方法,,比较姓名
        public int compareTo(Student s2) {
                return name.compareTo(s2.name);
        }
        //自定义比较规则：比较年龄
        public int compareByAge(Student s2) {
                return this.getAge()-s2.getAge();
        }
        public String toString() {
                return "["+level+","+ name+": "+age+"]" ;
        }
}
```

```
public class UseCol
    public static v
        ArrayList<S
        int pos;
        slist.add(n
        slist.add(new Student("lisi",40,1));
        slist.add(new Student("wangwu",30,2));
        slist.add(new Student("zhaoliu",60,2));
        //用Comparable接口方法comparaTo定义的默认规则排序。
        Collections.sort(slist);
        slist.forEach(a->System.out.print(a.toString()+", "));
        //用二分法查询，比较规则与Student的内置比较规则一致
         Student s1= new Student("lisi",40,1);
        pos=Collections.binarySearch(slist, s1);
        System.out.println("\n the position of lisi is:"+pos);
        //等价于 Collections.sort(slist,(s1,s2)->s1.compareByAge(s2));
       Collections.sort(slist, Student::compareByAge);
        slist.forEach(a->System.out.print(a.toString()+", "));
        //用二分法查询，比较规则与排序时自定义的比较器Comparator比较规则一致
        pos=Collections.binarySearch(slist, s1, Student::compareByAge);
        System.out.println("\n the position of lisi  is:"+pos);
    }}
```

```
[1,lisi: 40], [2,wangwu: 30], [1,zhangsan: 50], [2,zhaoliu: 60],
 the position of lisi is:0
[2,wangwu: 30], [1,lisi: 40], [1,zhangsan: 50], [2,zhaoliu: 60],
 the position of lisi  is:1
```

7.5.3 使用stream

1、Java8引入了流式操作stream的概念

- 流式操作是指能够串行或并行地对数据流进行函数式操作。流是通过数组及集合建立的数据流，
- 进入流的元素可以进行如过滤、排序、转换、等操作，可以通过lambda表达式指定具体的计算逻辑，这极大的简化了代码，是Java编程思维的一大改进。

2、stream不但提供了强大的数据操作能力，更重要的是stream既支持串行也支持并行，使得stream获得性能提升。

1、获取流

使用流之前首先要获取流，我们可以从数组和集合上获取，也可以利用Stream或它的子接口直接创建流。子接口流有：

（1）IntStream:整数的流

（2）LongStream：长整数的流

（3）DoubleStream: 实数的流

表 7-17 获取流的常用方法

功能	获得流的方式	示例
在数组上获得流	**Arrays**.stream(数组)	**Arrays**.stream(**a1**)
在数组上获得并发流	**Arrays.**stream(数组).parallel()	**Arrays**.stream(**a1**).parallel()
在集合（Collection 的子类型）上获得流	**集合对象**.stream()	**col1**.stream()
在集合上获得并发流	**集合对象**.parallelStream()	**col1**.parallelStream()
用 Stream.of 获得流，或用 IntStream.range(int s int e) 获得步长为 1 的整数流	**Stream**.of(一组元素) **IntStream**.range(开始，结束)	**Stream**.of(1,3,5,7) **IntStream**.range(3,10)
java.util.Random 生成随机数序列流	Random 对象 r 方法 r.Xxxs (long Size,int Origin, int bound) (Xxx 可为 int long double)	Random r=new Random(); r.ints(5,100,200)

2、流式操作

产生数据流后，可以用流的方法对数据进行多次处理和操作。流式API有两类，一类称为中间操作，一类称为终端操作。

表 7-18 Stream 接口中的常用方法

返回类型	方法	功能
中间操作		
Stream	filter(Predicate<? super T> pre)	返回一个符合匹配条件的流
Stream	map(Function<? super T,? extends R> m	对每个元素执行函数 m 后的结果流
Stream	sorted([Comparator<? super T> c])	返回排序后的结果流，比较器 c 可选
Stream	distinct()	返回去掉所有重复元素的流
Stream	limit(long maxSize)	返回长度不超过 maxSize 的子流
XxStream	mapToXx(ToXxFunction<? Super T> m)	依据 m 进行类型转换，Xx 可以是 Int Long Double
终端操作		
Optional	max/min(Comparator<? super T> c)	根据比较器 c 求最值
void	forEach(Consumer<? super T> action)	为流中的每个元素执行 action 操作
Object[]	toArray()	以数组形式返回流的数据
long	count()	返回流中的元素数。
T	reduce(T id, BinaryOperator<T> ac)	通过二目操作 ac 将流中元素积累到一起
R	collect(Collector<? super T,A,R> collector)	收集操作：将元素按一定规则收集到一集合中

例如：

```
List<String>
ls=Arrays.asList("a","bcdef","ghijk","lmn","opq","rstuv");

ls.stream()
.filter(s->s.length()>=3)
.map(s->s.toUpperCase())
.forEach(e->System.out.print(e+", "));
```

```
public class UseStream {
	public static void main(String arg[]) {
	//1、对数组流进行操作
	System.out.println("1、对数组进行流式操作");
	String str[]=
	{"a","bcdef","ghijk","lmn","opq","rstuv","a"};

	String result=Arrays.stream(str)
			.parallel()
			.distinct()                    //去掉重复元素
			.map(e->e.toUpperCase()) //映射成大写字母
			.reduce("", (a,b)->a+b);
                                //通过二目运算，将字符串连接起来
	System.out.println(result);
```

```
//2、对Random生成的整数流进行操作
System.out.println("2、利用Random的ints()方法生成流并进行操作");
Random r=new Random();
OptionalDouble max=r.ints(10,100,200)//生成10个100~200间的数组成的流
                .filter(i->i<150) //过滤，>=150的数
                .mapToDouble(i->i*i)//映射得到每个数的平方
                .sorted()  //排序
                .max();
System.out.println(max.isPresent()? "最大值"+max.getAsDouble():"无值");

//3、对集合进行操
System.out.print
List<Student> ls



ls.stream()       //获取流
  .filter(e->e.getAge()>30) //过滤
  .sorted(Student::compareByAge) //排序
  .forEach(e->System.out.print(e+","));
}
}
```

【运行一次结果】

```
1、对数组进行流式操作
ABCDEFGHIJKLMNOPQRSTUV
2、利用Random的ints()方法生成流并进行操作
最大值：21904.0
3、对Student集合进行流式操作
  [1,lisi: 40],[1,zhangsan: 50],[2,zhaoliu: 60],
```

3、数据收集(了解)

- Stream类中collect(Collector<? super T,A,R> collector)方法，将元素按一定的规则收集到一个集合中，其功能可理解为高级的“数据过滤+数据映射”，是对数据的深加工，但这些复杂操作不是由Stream实现的，而是由collect的形参Collector实现的。

- Collectors收集器类是Collector接口的实现类，它提供了丰富的API，下面只给出Collectors类中两个方法的用法：

（1）Collector<T,?,List<T>>　　toList()

　　返回一个收集器，其将输入的元素累积成一个新的List。

（2）Collector<T,?,Map<K,List<T>>>　　groupingBy(Function<? super T,? extends K> classifier)

　　返回一个实现分组操作的收集器，根据分类函数对元素进行分组，并将结果以Map形式返回。

```
public class TestCollector {
public static void main(String arg[]) {
                List<Student> ls=Arrays.asList(new Student("zhangsan",50,1),
                                        new Student("lisi",40,1),
                                         new Student("wangwu",30,2),
                                         new Student("zhaoliu",60,2));

    System.out.println("\n一、收集操作，收集成一个列表");
       List<Student> aslist=ls.stream()
                .sorted(Student::compareByAge) //排序
                .collect(Collectors.toList()); //收集，转成一个列表

        aslist.forEach(e->System.out.print(e+","));//打印列表

        System.out.println("\n二、收集操作， 分组进行收集");
        Map<Integer,List<Student>> sMap=ls.stream()   //获取流
                .collect(Collectors.groupingBy(Student::getLevel));//分组收集

     //打印分组情况
        Set<Integer> keySet=sMap.keySet();
        for(Integer i:keySet) {
        System.out.println(i+"级别的学生列表");
        List<Student> levelList=sMap.get(i);
     //将特定级别的学生列表元素打印出来
        levelList.forEach(e->System.out.print(e+", "));
        System.out.println();
        }}
}
```