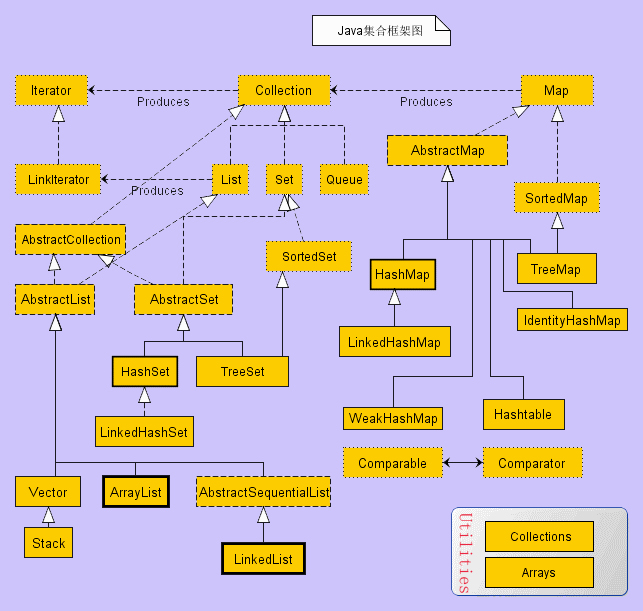
# 集合类框架



ArrayList和LinkedList类实现了List接口。List接口提供了访问列表中第n个元素的方法，也就是说一个集合类应该实现RandomAccess接口，这是一个没有方法的标记接口。

public class **ArrayList**<E> extends **AbstractList**<E> implements **List**<E>, **RandomAccess**, **Cloneable**, java.io.**Serializable**

public class **LinkedList**<E> extends **AbstractSequentialList**<E> implements **List**<E>, **Deque**<E>, **Cloneable**, java.io.**Serializable**

在代码中应尽可能使用接口，比如构造一个ArrayList后将引用存储为一个List类型的变量。

List<Integer> arrayList = new ArrayList<>();

当实现一个处理结合的方法时，**使用限制最少的接口作为参数类型**，通常Collection、List、Map就可以了。

# 迭代器

每个集合都提供了按某种顺序迭代元素的方式，Collection的父接口Iterable<T>定义了一个方法。

Iterator<E> **iterator**()

Iterator接口提供了一个remove方法，可以移除之前访问的元素

Iterator<Integer> iterator = arrayList.**iterator**();

while (iterator.**hasNext**()){

int i = iterator.**next**();

if(i == 2){

iterator.**remove**();

iterator.**remove**(); //java.lang.IllegalStateException

}

}

remove方法移除了迭代器**返回的最后一个元素**，并不是迭代器指向的元素，所以不可以连续调用两次remove方法。

如果使用多个迭代器访问一个数据结构，并且其中一个使数据结构发生改变，那么其他的迭代器可能会失效，继续使用一个无效的迭代器会异常。

Iterator<Integer> iterator = arrayList.**iterator**();

Iterator<Integer> iterator2 = arrayList.**iterator**();

while (iterator.**hasNext**()){

int i = iterator.**next**();

iterator2.**next**();//java.util.ConcurrentModificationException

if(i == 2){

iterator.**remove**();

}

}

# map

不管是set还是map，如果不需要按顺序访问键的话，HashSet和HashMap是更好的选择，如果需要按顺序访问键的话，使用TreeSet和TreeMap。

Map<String, Integer> map = new HashMap<>();

map.**put**("alice", 1);

map.**put**("bob", 2);

获取一个键所对应的值的时候：

int value = map.**get**("tom"); // java.lang.NullPointerException

如果map中不存在一个键，get方法返回值为null。上面的代码在自动拆箱的时候回导致空指针异常。

int value = map.**getOrDefault**("tom", -1);

上面的方法是更好的选择，如果键不存在，value将被赋值为-1。

当试图更新map里的value值时，首先要检查键是否存在，如果存在加1，否则赋值。

map.**merge**("tom", 3, Integer::sum);

map.**merge**("tom", 3, (x, y)->Integer.**sum**(x, y));

通过上面的方法可以简化操作。

可以获得map中键、值、记录的试图：

Set<String> keys = map.**keySet**();

Collection<Integer> values = map.**values**();

Set<Map.Entry<String, Integer>> entries = map.**entrySet**();

这三个方法返回的都不是map数据的拷贝，他们与map是相关联的，从任一个视图中移除记录，那么该记录也会从关联的map中移除。

# stack和queue

Java集合类框架中没有Stack接口，仅有一个早期遗留的Stack类。如果需要使用Stack、Queue和双端队列，并且不关心线程安全问题，可以使用ArrayDeque。

使用栈时，使用push和pop方法。

ArrayDeque<Integer> stack = new ArrayDeque<>();

stack.**push**(1);

stack.**pop**();

stack.**poll**();

stack.**pop**();*//java.util.NoSuchElementException*

pop和poll都是出栈操作，不同点是如果stack为空，pop会抛异常，而poll放回null。

使用队列时，使用add和remove方法。

ArrayDeque<Integer> queue = new ArrayDeque<>();

queue.**add**(1);

queue.**remove**();

使用优先级队列可以容纳实现了Comparable接口的类的元素，或者可以在该类的构造方法中提供一个Comparator。优先级队列在添加和删除元素时，最小值的元素总是在根部。

public class **Main** {

public static void **main**(String[] args){

PriorityQueue<Job> jobs = new PriorityQueue<>();

jobs.**add**(new **Job**(4, "system call"));

jobs.**add**(new **Job**(2, "exception"));

jobs.**add**(new **Job**(10, "user call"));

while(jobs.**size**() > 0){

out.**println**(jobs.**remove**());

}

}

}

class **Job** implements **Comparable**<Job>{

private int priority;

private String description;

public **Job**(int priority, String description){

this.priority = priority;

this.description = description;

}

@Override

public int **compareTo**(Job o) {

return this.priority - o.priority;

}

@Override

public String **toString**() {

return "Job{" +

"priority=" + priority +

", description='" + description + '\'' +

'}';

}

}