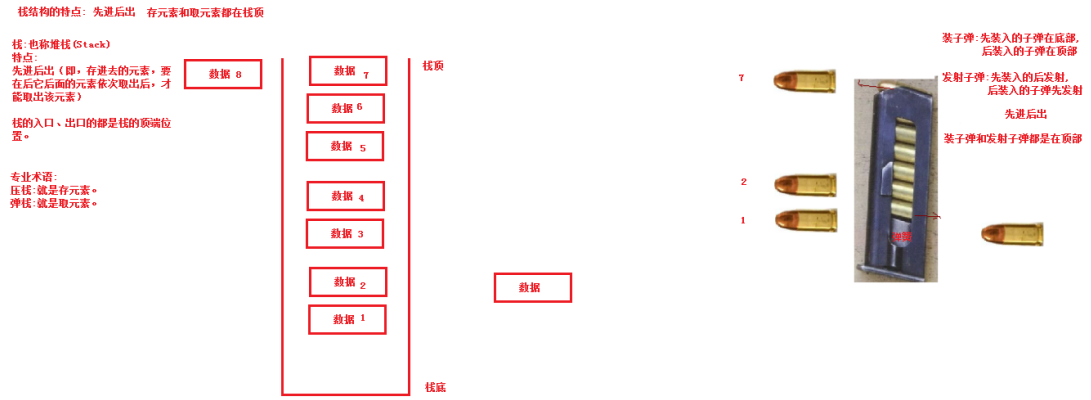
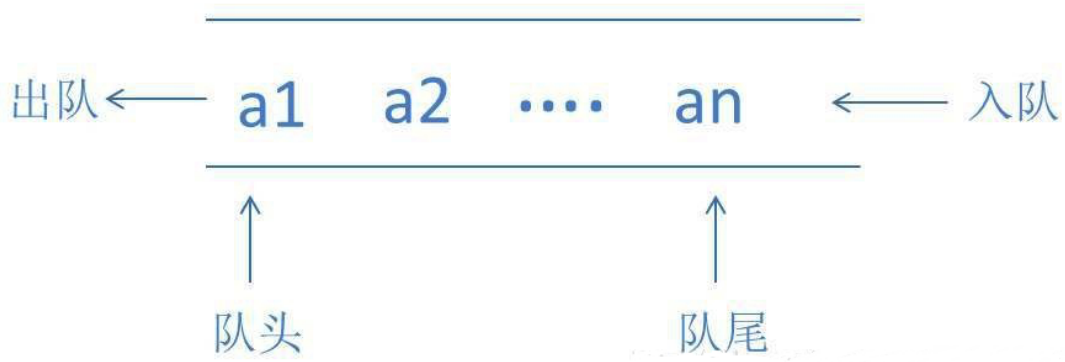
LinkedList集合和泛型

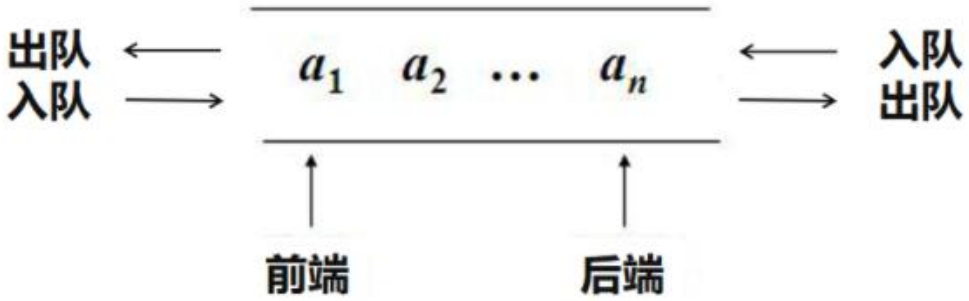
List集合之LinkedList集合

> LinkedList集合也是List集合接口中主要实现类，这个LinkedList的主要实现结构有【链表】，除了链表结构之外LinkeList还使用栈、 队列、数组，LinkedList是允许存重复数据化并且可以快速插入数据
>
> **LinkedList中实现的数据结构**
>
> - **栈**：**stack**,又称堆栈，它是运算受限的线性表，其限制是仅允许在表的一端进行插入和删除操作，不允许在其他任何位置进行添加、查找、删除等操作。
>
> 简单的说：采用该结构的集合，对元素的存取有如下的特点
>
> - **先进后出**（即，存进去的元素，要在后它后面的元素依次取出后，才能取出该元素）。例如，子弹压进弹夹，先压进去的子弹在下面，后压进去的子弹在上面，当开枪时，先弹出上面的子弹，然后才能弹出下面的子弹。
> - **栈的入口、出口的都是栈的顶端位置。**
>
> 
>
>
> 这里两个名词需要注意：
>
> - **压栈**：就是存元素。即，把元素存储到栈的顶端位置，栈中已有元素依次向栈底方向移动一个位置。
> - **弹栈**：就是取元素。即，把栈的顶端位置元素取出，栈中已有元素依次向栈顶方向移动一个位置。

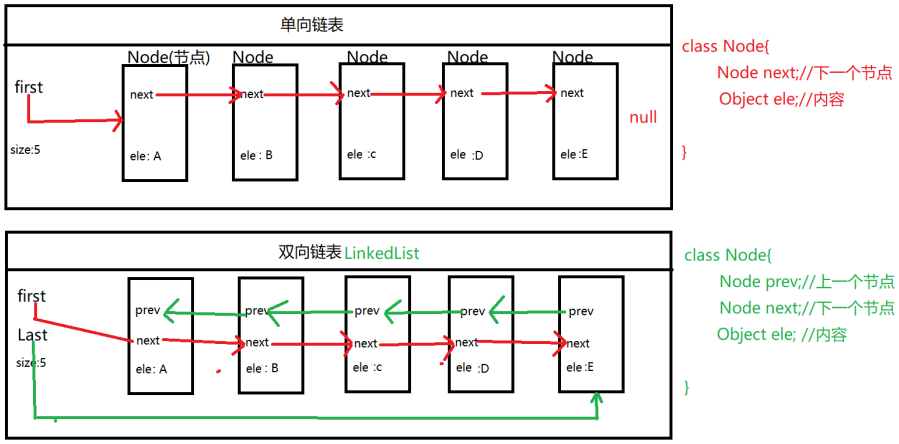
- **队列：**队列和栈有一些类似，也是一种受到限制的线性表，其限制是仅允许在表的一表进行插入，另外一段进行取出和删除，这样队列叫做单向队列【Queue】
- 队列是存在一个特点：先进先出



队列：除了单向队列之外，还提供一种非受限制线性表，这表就是双端队列，依旧遵守先进先出原则，双端队列【Deque】



链表：链表属于一个线性表，线性表中包含了存储链接位置和数据的位置，利用链接位置进行相连接，使数据之间呈现一种链接状态，链表可以理解为：老鹰抓小鸡



> 链表和队列差不多，链表也分为单向和双向链表，类似于现实生活中的【火车】通过链接方式就可以获取得到链表中数据了
>
> 单向链表和双向链表是有本质上区别的，单向链表包含两个区域【存储链接位置和数据存储位置】，通过【存储链接位置】让每一个单独链接节点进行相连，形成一个单向链表，即单向链表只能向一个方法进行遍历操作，双向链表包含是三个区域【2个存储链接位置和1个数据存储位置】，利用双向链表中两个存储链接的位置进行前后节点的链接，此时链表就形成了一个双向通道，即可以从头到尾，也可以从到头，允许直接在头尾两端进行操作

分析LinkedList的执行效率

> 主要实现方式是【双向链表】，所以使用LinkedList效果
>
> 1.保存数据： 只要执行一次即可【API中也提供 addFirst 和 addLast】
>
> 2.删除操作： 只要执行一次即可【API中也提供 removeFirst 和 removeLast】，就选中间节点(1+N)/2
>
> 3.查询操作和修改操作：平均值(N+1)/2
>
> 结论：LinkedList增删执行效率是高的，但是查询和修改执行效率是低的

LinkedList的API说明

```
import java.util.Iterator;
import java.util.LinkedList;

public class LinkedListAPIDemo {
    public static void main(String[] args) {
        //因为LinkedList实现了List集合接口，所以具备所有List集合操作方法
        //LinkedList中List方法操作完全参考ArrayList即可

        //LinkedList集合独有方法

        //1.创建LinkedList对象
        //1.1 创建一个空的LinkedList集合对象
        LinkedList linkedList = new LinkedList();
```

```
        //1.2使用参数中Collection集合对象中存储数据初始化
LinkedList集合对象
        LinkedList linkedList1 = new
LinkedList(linkedList);

        //独有API
        //1.向集合开头的位置添加元素
        linkedList.addFirst(1);
        System.out.println(linkedList);
        //2.向集合结尾的位置添加元素
        linkedList.addLast(2);
        System.out.println(linkedList);
        //3.获取集合第一个元素的数据【但是不删除集合数据】
        Object element = linkedList.element();
        System.out.println(element);
        //4.获取集合中第一个数据和最后一个数据【但是不删除集合数据】
        Object first = linkedList.getFirst();
        Object last = linkedList.getLast();
        //PS:  LinkedList允许使用下标形式进行数据获取
        Object o = linkedList.get(0);
        //5.添加元素到集合的末尾
        linkedList.offer(3);
        //添加到第一个位置offerFirst和最后一个位置offerLast
        //6.获取集合集合第一个元素但是【不删除集合数据】，有元素就返
回，没有就是null
        Object peek = linkedList.peek();
        Object o1 = linkedList.peekFirst();
        //获取最后一个元素的值，有元素就返回，没有就是null
        Object o2 = linkedList.peekLast();

        //7.poll系列也是也可以获取集合中第一个和最后一个元素的【删
除中数据】
//        Object poll = linkedList.poll();
//        Object o3 = linkedList.pollFirst();
//        Object o4 = linkedList.pollLast();

        //8. 弹出集合中第一个数据
        Object pop = linkedList.pop();
        //向集合中添加数据
        linkedList.push(4);
```

```
        //9.删除集合中第一个或最后一个元素值
        linkedList.removeFirst();
        linkedList.removeLast();
        //10.List集合特点就是允许存储重复数据，下面两个方法的作用就是遍历集合
        //First的方法从前之后      Last的方法从后向前
        //遇到第一个相同数据删除
        linkedList.removeFirstOccurrence(1);
        linkedList.removeLastOccurrence(1);

        //LinkedList有一个独有的迭代器，允许逆向的遍历迭代器中数据值
        //但是它的操作方式和iterator是一样
        Iterator iterator = linkedList.descendingIterator();
        //它的方式只有三个  hasNext  next  和 remove
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }
        //不同点在于：它的光标最先开始是放置在迭代器最后一个元素的位置，你hasNext是向前判断


    }
}
```

List集合

> List集合是Collection集合子集合即List集合是继承与Collection集合，因为List集合是接口所以无法直接操作，Java就提供了两个可以便捷操作List集合实现类**ArrayList和LinkedList**
>
> List集合的特点：允许存储重复数据并给存储数据是有顺序
>
> 在官方API文档中可以查看到：
>
> `public interface List extends Collection`
>
> 有序的 collection（也称为*序列*）。此接口的用户可以对列表中每个元素的插入位置进行精确地控制。用户可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。

- **所有已知实现类：** `ArrayList`和`LinkedList`
- **所有超级接口：** `Collection<E>`和`Iterable<E>`

因为也继承了Iterable所以List集合中是支持的迭代器【Iterator】

List接口中提供常用方法已经在ArrayList集合中完全进行演示，所以使用List集合时只要参考ArrayList提供方法演示就可以操作集合

ArrayList和LinkedList实现类是实现List接口，所以List相当于是他们的父类，所以List集合接口支持多态

```
List集合接口允许这样这样创建对象
List  list =  new ArrayList();
或者
List  list = new LinkedList();
"这样直接使用较少，多用于在方法中如果需要设置List集合参数，优先会将List作为参数类型，而不是实现类"
"因为这样做可以接收ArrayList或LinkedList"
"上面这种方式是不使用泛型，所以默认数据类型时Object，操作时就一定注意转换问题【对象的向下转型】"
"Java中所有集合都是有泛型语法，所以可以使用泛型方式进行创建对象"
List<数据类型> list = new  ArrayList<>();
"泛型语法形式："
集合的数据类型<数据类型> 集合对象名字 = new 集合数据类型<>();

"集合都是使用new关键字创建，所以它是引用类型"
"现在学习ArrayList和LinkedList都是线程不安全的集合，在多线程操作前提下是不能使用这两个集合进行数据存储操作，这两个集合只适合在单线程下使用"
```

List集合接口还有一个实现类Vector

Vector在JDK1.0的时候就已经存在，这个类其实是ArrayList集合类“前身”，在JDK1.2时候Java开始强调了集合框架概念，所以提供Collection集合框架，在Collection下面提供List和Set，Vector的实现也是数组，所以被归类到了List接口的实现类中，现在开发已经不在使用Vector这个类，主要使用ArrayList这个类。

Vector和ArrayList之间相似点和区别在于什么?

相似点：都是List集合接口的实现类，都使用数组作为集合数据结构进行实现

> 不同点：Vector这个集合是线程安全的，但是效率低，提供较早并且使用率低
>
> ArrayList这个集合是线程不安全，但是效率高，提供较晚并且使用率高
>
> PS：就算Vector是线程安全的，但实际开发中也不会使用Vector，还是使用ArrayList
>
> 从Java5开始提供ArrayList和LinkeList线程安全处理模式:
>
> "集合中是存在一个工具类，这个类叫做Collections，这个工具类提供了如果将线程不安全集合转换为线程安装集合的方法"
> Collections.synchronizedList(List集合对象); --->》可以将一个线程不安全集合转换为线程安全
> "除了这种处理方式之外,现在主要处理方式可以使用Java在JDK1.5中提供的一个新包java.util.concurrent 并发工具包，这个包中提供大量的线程安全处理时可以使用的集合"
>
> Stack(栈集合)和Queue、Deque(队列集合)
>
> Stack是List集合的实现类，而Queue、Deque集合接口：有专门实现类来实现主要体现在于LinkedList

泛型

> **什么是泛型?**
>
> 泛型是一个特殊类型，泛型是统称指代任何引用数据类型，泛型本身代表【通用类型含义】，在定义泛型时如果没有对泛型进行数据类型赋值之前，泛型本身是不具备任何含义，只有赋值为具体数据类型时，泛型才会真正意义。
>
> **为什么要使用这个泛型?**
>
> 泛型解决了集合中存储数据类型的问题

```
import java.util.ArrayList;

public class CenericityList {
    public static void main(String[] args){
        /*
```

```
        在学习集合之前，是没有使用泛型，所以集合中默认类型就是Object
        我们面临的问题就是取出数据时，需要进行向下转型操作才可以
         */
        ArrayList list  = new ArrayList();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        //对集合存储的数据进行计算求和
        int sum =0 ;
        for(Object obj : list){
            sum += ((Integer) obj);
        }
        //因为不使用泛型默认类型时Object类型所以和这个集合中就可以存储任何数据类型只要是Object子类
        list.add('9');
        list.add("7");
        list.add(new Student());
        list.add(true);
        //如果你在不清楚集合中存储在什么样数据类型数据时，如果进行转换操作？代码就无形中增加开发成本

        /*
        所以在这种情况下，集合建议使用泛型，来约束集合中存储数据
        一旦集合使用泛型就要可以约束集合只能 存储泛型中提供的数据类型对应数据，从而减少转换操作
        此时list1这个集合中只能存储Integer类型数据，使用泛型作为约束
         */
        ArrayList<Integer> list1 = new ArrayList<>();
        list1.add(1);
        //list1.add("1");
        int sum1 = 0;
        for(Integer i : list1){
            sum1+=i;
        }
    }
}
```

> PS： 如何定义泛型集合?

语法：

集合数据类型<存储数据的数据类型> 集合对象名 = new 集合数据类型<>();

此时就可以使用【存储数据的数据类型】约束集合中存储的数据了

PS：集合泛型你就可以理解为 就是创建数组是

数组中存储元素数据类型[] ---》相当于 集合数据类型<存储数据的数据类型>

泛型解决通用性问题

编程原则：DRY【不要重复你自己（不要写重复性代码）】

需求： 求点中x和y的值

```
class Point{
private Integer x;
private Integer y;
}
class Point2{
private Double x;
private Double y;
}
class Point3{
private Long x;
private Long y;
}
class Point4{
private String x;
private String y;
}
```

为了Point类型可以获得不同参数类型数据，需要提供大量相同逻辑类，此时就触发DRY原则，现有解决方案就是将数据类型替换成Object类型，但是我们需要面临数据类型转换和传递非计算类型问题，有没有什么方式可以动态决定类中x和y属性类型操作，根据外界传递类型来进行x和y的限制操作 ---》 可以使用"泛型"

```
class Point<T>{
private T x;
```

```
private T y;
}
```

泛型的定义与使用

泛型的定义

> PS：泛型中会出现一个【占位符】的概念，这个占位符本身是没有任何意义，就是一个占位，对泛型进行赋值时，占位符才会有具体的意义
>
> 泛型的概念是Java5开始引入到Java中，它可以通过对泛型赋值进行对数据赋值约束，通过泛型动态决定数据类型时什么
>
> 语法：

```
<占位符>
PS：这种语法在Java中叫做"菱形语法"，这样语法与占位符组合就是成为"泛型"
这个语法可以使用在 "类、方法和接口"上
占位符"一般是一个大写字母[A~Z]",不建议使用其他形式进行占位符定义
习惯书写占位符是 "T" --> "Type(类型)"  --->  <T> 泛型T
占位符可以在一个语法存在多个需要使用","分隔，使用占位符的多少就相当于你定义多个泛型
泛型语法:只能存在在编译时期，一旦程序运行泛型就会自动消失"称之为泛型擦除"
        定义的泛型在编译字节码文件中即[.class文件]看不到
```

泛型的使用之集合

```
public class CenericityList {
    public static void main(String[] args){
        // 集合的数据类型<泛型赋值数据类型> 集合对象名字 = new 集合的数据类型<>();
        // 声明的就是带有泛型集合，泛型可以作为集合类型一部分，可以出现在方法参数位置和返回值类型的位置
        ArrayList<Integer> list1 = new ArrayList<>();
        list1.add(1);
        //一旦集合使用泛型之后，集合只允许使用泛型定义数据类型，非泛型定义数据类型时无法存储到集合中
        //list1.add("1");
```

```
        int sum1 = 0;
        //集合使用泛型之后，确定了集合中存储数据的数据类型，在集合中的数据就无需向下转型操作
        for(Integer i : list1){
            sum1+=i;
        }
    }
    public static void showList(ArrayList<Integer> list){}

    public static ArrayList<Integer> showList(){}
}
```

泛型的使用之泛型类

```
/**
当前类在没有使用任何其他修饰符之前【final 或 abstract】,没有使用泛型语法之前 都是一个普通类
 public class CenericityClass {}
 利用泛型语法  <占位符> 可以将当前类变成泛型类
 语法:
 public class  类名<占位符>{     这个类就是泛型类
    此时这个泛型是定义在类上，所以在类中成员变量和成员方法都可以使用这个泛型
    作为数据类型使用
 }
 */
public class CenericityClass<T> { // 使用泛型是 T 这个T现在是没有意义 只是一个占位符号
    //此时这个T是没意义，为了保证语法不错误，占位使用
    //泛型T只有被赋值之后【数据类型（必须是引用数据类型）】 T才会有意义
    private  T x;
    public  T y;
    public CenericityClass(){}
    public CenericityClass(T x, T y) {
        this.x = x;
        this.y = y;
    }
    public void show(T t){
        System.out.println(t);
    }
```

```
}
//在类上定义泛型，如何确定泛型数据类型，只要在创建类的对象时对泛型进行赋值，泛型就有具体的数据类型
class Test{
    public static void main(String[] args) {
        // 类名<泛型赋值数据类型>  对象名字 = new 类名<>();
        CenericityClass<Integer>  cc = new CenericityClass<>();
        cc.y = 1;
        cc.show(1);
        //这样创建对象,创建泛型的同时不对泛型进行赋值操作
        //此时没有对泛型进行赋值，所以泛型类型默认使用Object
        CenericityClass cc1 = new CenericityClass();
        cc1.y = 1;
        cc1.y = "1";
        cc1.show('1');
    }
}

"PS：泛型在动态决定数据类型时什么的时候，不存在继承关系"
"不要这样写，这个语法是错误，对泛型进行赋值什么数据类型时，就决定这个数类型，所不允许赋值为其他类"
    CenericityClass<Object>  cc = new CenericityClass<Integer>();
```

泛型使用之泛型方法

> 泛型方法的定义主要是为了摆脱使用泛型类上或泛型接口上的泛型约束问题，就相当于方法向使用自己定义泛型作为类型操作，就可以定义为泛型方法
>
> 泛型类和泛型接口上定义泛型，不能在静态方法上使用，所以只能定义泛型方法对方法进行泛型使用修饰

```
public class GenericityMethod<T> { // 使用泛型是 T 这个T现在是没有意义 只是一个占位符号
    //提供一个成员方法 泛型类上定义泛型就可以在方法中使用
    public void show(T t){
        System.out.println(t);
    }
    //泛型类上定义泛型是不能在静态方法上使用
```

```
    //public static void  showInfos(T t){ }

    //给方法添加泛型
    /*
     泛型静态方法:
     访问权限修饰符   static<占位符>   返回值类型  方法名(参数列表){
            此时这个占位符可以使用在返回值类型上  和  参数列表定义中
     }
     */
    public static<E> void showInfos(E e){
        //E a;// 方法上泛型主要是为了 返回值类型和参数类型而提供的，在内部就不在使用
        System.out.println(e);
    }
    /*
     泛型成员方法:
     访问权限修饰符<占位符>   返回值类型  方法名(参数列表){
            此时这个占位符可以使用在返回值类型上  和  参数列表定义中
     }
     此时不仅可以使用泛型方法上定义泛型，也可以使用类上或接口上定义泛型

     */
    public<F> void showInfoss(F f,T t){
        //F a;// 方法上泛型主要是为了 返回值类型和参数类型而提供的，在内部就不在使用
        System.out.println(f);
    }
    //定义泛型方法泛型，不能单独使用在方法返回值类型位置，需要配合使用参数列表定义
    public static<O>  O  showInfosss(O o){ //静态方法可以这样操作，成员的不可以
        //return o;
        return  o;
    }
}
class Test1{
    public static void main(String[] args) {
        //如何给方法定义泛型进行赋值操作---》赋值数据类型
        //在调用方法对泛型定义参数列表赋值时，可以决定方法的泛型是什么，定义泛型方法时
        //一定要将泛型定义在参数列表中，以确定数据类型是什么
```

```
        GenericityMethod.showInfos("1");
        GenericityMethod genericityMethod = new
GenericityMethod();
        Integer integer = GenericityMethod.showInfosss(1);
    }
}
```

泛型使用之接口泛型

> 接口泛型和泛型类差不多，在接口上定义泛型，这个泛型可以在接口内部使用

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

/*
  public interface 接口名<占位符>{}泛型接口
 */
public interface GenericityInterface<T>{
    //在接口上定义泛型可以在接口内部使用
    void run(T t);
    //支持抽象方法自定义泛型
    public abstract<E> void  show (E e);
}
//1. 在使用类实现接口时可以对泛型接口的泛型进行赋值，决定泛型类型是什么
class Demo implements  GenericityInterface<Integer>{
    //在接口中定义方法使用泛型位置都会变成数据类型
    @Override
    public void run(Integer integer) {
    }

    @Override
    public <E> void show(E e) {
    }
}
//2. 使用泛型类实现泛型接口，使用泛型类中泛型作为接口中新泛型
class Demo2<P> implements  GenericityInterface<P>{
    //在创建泛型类对象时可以决定泛型类型
    @Override
    public void run(P p) {
```

```
    }
    @Override
    public <E> void show(E e) {

    }

    public static void main(String[] args) {
        GenericityInterface<Double> gi = new Demo2<>();
        gi.run(1.0);
        Demo2<String> demo2 = new Demo2<>();
        demo2.run("1");
        //这种操作其实就是List集合中使用List集合创建对象的方式
        List<String> list = new ArrayList<>();
        ArrayList<Long> list2 = new ArrayList<>();

    }
}
//3.直接使用匿名内部类的形似进行泛型接口上泛型的赋值
class Demo3{
    public static void main(String[] args) {
        new GenericityInterface<Integer>(){

            @Override
            public void run(Integer integer) {
            }

            @Override
            public <E> void show(E e) {
            }
        };
    }
}
```

泛型的限定

> 泛型限定其实就是定义泛型可以赋值哪些数据类型，只有满足限定要求的数据类型才可以进行定义赋值操作
>
> **泛型通配符【?】**

> ?代表未知，可以作为通配符使用，但是不能作为参数类型单独使用，通配符多用于在泛型限定上

```
public class GenericityDemo {
    public static void main(String[] args) {
        // ?  是一个通配符
        List<?> list = new ArrayList<>();
        list.add(1); // 这里是无法确定数据类型，无法单独使用
    }
}
```

> **<? extends 类> 这是使用途径 此时？ 就代表着可以接收extends关键字后的相同类型或子类**
>
> **<? super 类> 这是使用途径 此时？ 就代表着可以接收super关键字后的相同类型或父类**

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

public class GenericityDemo {
    public static void main(String[] args) {
        //提供4个List集合对象
        List<Integer> list1 = new ArrayList<>();
        List<String> list2 = new ArrayList<>();
        List<Number> list3 = new ArrayList<>();
        List<Object> list4 = new ArrayList<>();

        //调用泛型的上限操作，即参数类型使用的是List<? extends Number>进行限制
        /*
        泛型限定必须是 Number类型或Number子类
        而list1是Integer类型即Number子类 list3是Number类型即Number类型所以可以进行传递
        而list2和list4分别是String和Object，既不是Number类型也不是Number子类，所以无法传递到方法中
         */
        doWork1(list1);
        //doWork1(list2);
        doWork1(list3);
        //doWork1(list4);
```

```
        //调用泛型下限操作，即参数类型使用的是List<? super Number>进行限制
        /*
        泛型限定必须是Number类型或Number父类
        而list1和list2分别是Integer和String，既不是Number类型也不是Number父类，所以无法传递到方法中
        list3和list4 分别是Number和Object  满足了必须是Number类型或Number父类，所以传递到方法中
         */
        //doWork2(list1);
        //doWork2(list2);
        doWork2(list3);
        doWork2(list4);

    }
    //泛型的上限，此时的泛型中?(通配符) 必须是Number的类型或Number子类
    public static void doWork1(List<? extends Number> list){

    }
    //泛型的下限，此时的泛型中?(通配符) 必须是Number的类型或Number父类
    public static void doWork2(List<? super Number> list){

    }

}
```