一、Java泛型介绍

1.1 概念

> Java泛型是J2SE 1.5 版本中增加的新特性，其本质是参数化类型，也就是说所操作的数据类型被指定为一个参数（type parameter）这种参数类型可以用在类、接口和方法的创建过程声明，分别称为泛型类、泛型接口、泛型方法。
>
> Java集合（Collection）中元素的类型是多种多样的。没有泛型（Generics）的情况下，通过对类型Object的引用来实现参数的“任意化”，“任意化”带来的缺点是要作显式的强制类型转换，而这种转换是要求开发者对实际参数类型可以在预知的情况下进行的。对于强制类型转换错误的情况，编译器可能不提示错误，在运行的时候才出现异常，这是一个安全隐患。因此，为了解决这一问题，J2SE 1.5引入泛型也是自然而然的了。
>
> 《阿甘正传》生活就像一盒巧克力，你永远不知道你会得到什么!!

泛型解释

1.2 优点

1.2.1 类型安全

> 泛型的主要目标是提高Java程序的类型安全。通过知道使用泛型定义的变量的类型限制，编译器可以在非常高的层次上验证类型假设。没有泛型，这些假设就只存在于系统开发人员的头脑中。

1.2.2 消除强制类型转换

> 泛型的一个附带好处是，消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。泛型消除了强制类型转换之后，会使得代码加清晰和简洁。

1.2.3 更高的运行效率

> 在非泛型编程中，将简单类型作为Object传递时会引起Boxing（装箱）和Unboxing（拆箱）操作，这两个过程都是具有很大开销的。引入泛型后，就不必进行Boxing和Unboxing操作了，所以运行效率相对较高，特别在对集合操作非常频繁的系统中，这个特点带来的性能提升更加明显。

1.2.4 潜在的性能收益

> 泛型为较大的优化带来可能。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，Java系统开发人员会指定这些强制类型转换）插入生成的字节码中。但是更多类型信息可用于编译器这一事实，为未来版本的JVM的优化带来可能。

1.3 泛型类型

- 泛型类
- 接口泛型
- 泛型方法

1.4 泛型演示

> 通过著名数据库工具类DBUtils处理结果集的源码部分进行演示！
>
> 数据库查询方法 query,可以根据传入的解析器类型，指定确认泛型，最终达到一个query方法既可以返回map数据又可以返回数组结构，还可以返回Java实体类等数据！非常灵活！

```
/**
```

```
     * Executes the given SELECT SQL query and returns a result object.
     * The <code>Connection</code> is retrieved from the
     * <code>DataSource</code> set in the constructor.
     * @param <T> The type of object that the handler returns
     * @param sql The SQL statement to execute.
     * @param rsh The handler used to create the result object from
     * the <code>ResultSet</code>.
     * @param params Initialize the PreparedStatement's IN parameters with
     * this array.
     * @return An object generated by the handler.
     * @throws SQLException if a database access error occurs
*/
public <T> T query(String sql, ResultSetHandler<T> rsh, Object... params) throws
SQLException {
        Connection conn = this.prepareConnection();

        return this.<T>query(conn, true, sql, rsh, params);
}
```

二、泛型使用

2.1 使用泛型类

> 当我们在声明类或接口时，类或接口中定义某个成员时，该成员有些类型是不确定的，而这个类型需要在使用这个类或接口时才可以确定，那么我们可以使用泛型。

2.1.1 定义泛型类

```
public class 类名 <类型变量列表 T,X...> {

    类体可以使用自定义泛型!
}
```

注意：

- <类型变量列表>：可以是一个或多个类型变量，推荐使用单个的大写字母表示。例如：`<T>`、`<K,V>` 等。
- 当类或接口上声明了<类型变量列表>时，其中的类型变量不能用于静态成员上。
- 泛型类,在实例化具体对象的时候,需要确认泛型类型!

2.1.2 使用泛型类

```
//声明
package com.atguigu.generic;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2021/10/2 00:18
 * description: 创建学生类
 *                有姓名和分数属性
 *                姓名类型为字符串
 *                分数不确定，语文老师习惯给整数 80 90 100分
```

```
 *                                英语老师习惯给字符串 A B C D 等
 */
/**
 * 声明泛型
 * @param <T>
 */
public class Student<T> {
    private String name;

    /**
     * 使用泛型
     */
    private T score;

    public String getName() {
        return name;
    }

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

    public T getScore() {
        return score;
    }

    public void setScore(T score) {
        this.score = score;
    }

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


//明确泛型
public class GenericClass {

    public static void main(String[] args) {

        //泛型赋值
        Student<String> englishStudent = new Student<>();
        englishStudent.setName("marry");
        englishStudent.setScore("A");


        Student<Integer> chineseStudent  = new Student<>();
        chineseStudent.setName("二狗子");
        chineseStudent.setScore(90);

    }
}
```

2.2 泛型接口

> 接口定义泛型， 需要在实现类确定泛型!

2.2.1 泛型接口语法

```
public interface 接口名<类型变量列表> {


}
```

2.2.2 泛型接口声明

```
public interface Factory<T> {
    /**
     * 工厂生成数据方法! 生成数据方法
     * @return
     */
    T createData();

}
```

2.2.3 实现类直接明确类型

> 实现类直接确定泛型类型

```
/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/2 22:45
 * description:生成字符串数据的工程
 */
public class StringFactory implements Factory<String> {
    /**
     * 工厂生成数据方法! 生成数据方法
     *
     * @return
     */
    @Override
    public String createData() {
        return null;
    }
}
```

2.2.4 泛型传递

> 实现类不直接确定，再实例化实现类对象的时候确定

```
package com.atguigu.generic;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/2 22:49
```

```
 * description:不确定类型工厂
 */
public class OtherFactory<E> implements Factory<E> {

    /**
     * 工厂生成数据方法! 生成数据方法
     *
     * @return
     */
    @Override
    public E createData() {

        return null;
    }

    public static void main(String[] args) {

        Factory<Integer> factory = new OtherFactory<>();

        Integer data = factory.createData();
    }
}
```

2.3 泛型方法

> - 是否拥有泛型方法，与其所在的类是否泛型无关。
> - 要定义泛型方法，只需将泛型参数列表置于返回值前。
> - 泛型方法使得该方法能够独立于类而产生变化
> - 指导原则:无论何时,只要能做到泛型方法可以解决问题,就选择泛型方法! 解决问题更加清晰明白!
> - 静态方法一样可以添加泛型,但是静态方法无法使用泛型类的泛型

2.3.1 泛型方法语法

```
【修饰符】 <类型变量列表> 返回值类型 方法名(【形参列表】)【throws 异常列表】{
    //...
}
```

2.3.2 泛型方法使用

```
package com.atguigu.generic;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/2 23:35
 * description: 实例化对象
 */
public class GenericMethods {

     public <T> void gm(T t){
```

```
            System.out.println(t.getClass)
        }

    public static void main(String[] args) {
        Test1 test1 = new Test1();
        test1.gm("a");
        test1.gm(11);
        test1.gm(true);

        //type = java.lang.String
        //type = java.lang.Integer
        //type = java.lang.Boolean
    }
}
```

泛型方法和泛型类和接口不同点,泛型方法可以不用显示的指明类型,编译期会为我们自动找到具体的类型!这称为`类型参数推断[type argument inference]`!

注意:如果传入的基本数据类型,会自动转为封装类型!

2.3.3 显示指明方法泛型[了解]

上一环节,演示了方法的泛型推断,这是最常用的形式,当然了方法也可以显示的指定泛型,但是语法使用非常少!了解即可!

> 调用对象/类. <指定类型>泛型方法();

```
//泛型方法
public <T> ArrayList<T> ret(){

        return new ArrayList<T>();
}

//显示指定泛型
Test1 test1 = new Test1();

ArrayList<String> ret = test1.<String>ret();
```

2.3.4 静态和可变参数方法

> 静态方法和可变参数方法一样可以使用泛型!
>
> 静态方法稍微有一点特殊,他不可以像非静态方法一样,应用类或者接口泛型![原因静态方法加载优先级高于类实例化]

```
package com.atguigu.generic;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/2 23:38
 * description:
 */
```

```java
public class StaticGenericMethod<T> {

    /**
     * 普通方法，引用类泛型
     * @param t
     * @return
     */
    public T retData(T t){

        return t;
    }

    /**
     * static 可以定义方法方法泛型！但是不能引用父类泛型！
     * @param e
     * @param <E>
     * @return
     */
    public static <E> E retStaticData(E e){

        return e;
    }


    /**
     * 这种T会报错！因为类的泛型是实例化对象创建，静态方法优先级高于实例化对象！
     * @param f
     * @param h
     * @param t
     * @param <F>
     * @param <H>
     * @return
     */
    public static  <F,H> F retStaticDataD(F f, H h,T t){

        return null;
    }


    /**
      静态+可变参数
      makeList()就演示了 Arrays.asList()方法的底层实现原理
     */
    public static <T> List<T> makeList(T...args){
        List<T> list = new ArrayList<>();

        for(T item : args){
            list.add(item);
        }
        return list;
    }

     //List<Integer> integers = Test1.makeList(1, 2, 3, 3);
     //List<String> list = Test1.<String>makeList("a", "v");

}
```

三、泛型练习

3.1 打印坐标

> 1、声明一个坐标类Coordinate，它有两个属性：x,y，都为T类型
> 2、在测试类中，创建两个不同的坐标类对象，
>
> 分别指定T类型为String和Double，并为x,y赋值，打印对象

```
public class TestExer1 {
    public static void main(String[] args) {
        Coordinate<String> c1 = new Coordinate<>("北纬38.6", "东经36.8");
        System.out.println(c1);

//      Coordinate<Double> c2 = new Coordinate<>(38.6, 38);//自动装箱与拆箱只能与对应的类型 38是int，自动装为Integer
        Coordinate<Double> c2 = new Coordinate<>(38.6, 36.8);
        System.out.println(c2);
    }
}
class Coordinate<T>{
    private T x;
    private T y;
    public Coordinate(T x, T y) {
        super();
        this.x = x;
        this.y = y;
    }
    public Coordinate() {
        super();
    }
    public T getX() {
        return x;
    }
    public void setX(T x) {
        this.x = x;
    }
    public T getY() {
        return y;
    }
    public void setY(T y) {
        this.y = y;
    }
    @Override
    public String toString() {
        return "Coordinate [x=" + x + ", y=" + y + "]";
    }

}
```

四、无界通配符

4.1 无界通配符概念

> 在确定泛型类型的时候，临时无法确认具体类型，可以使用无界通配符站位？
>
> ? 的优势，暂时不确定类型，后续方法可以再确认！
>
> ? 和 的关系： ？ 代表确认泛型，占时不确定类型而已！ T 代表声明泛型 完全两回事！
>
> ？ 对比 Object : Object一旦确定，后面无法在概念， ? 可以改变，而且还有高级语法 上限和下限！

4.2 演示案例

```
//1. 定义学员泛型类
public class Student<T>{
    private String name;
    private T score;

//2. 我们要声明一个学生管理类，这个管理类要包含一个方法，可以遍历学生数组。
//   我的工具方法只是输出学员信息，但是我不确定学生的泛型！所以 ？ 通配符！
class StudentService {
    public static void print(Student<?>[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}

//3. 确认数据调用输出信息方法
public class TestGeneric {
    public static void main(String[] args) {
        // 语文老师使用时：
        Student<String> stu1 = new Student<String>("张三", "良好");

        // 数学老师使用时：
        // Student<double> stu2 = new Student<double>("张三", 90.5);//错误，必须是引用数据类型
        Student<Double> stu2 = new Student<Double>("张三", 90.5);

        // 英语老师使用时：
        Student<Character> stu3 = new Student<Character>("张三", 'C');

        Student<?>[] arr = new Student[3];
        arr[0] = stu1;
        arr[1] = stu2;
        arr[2] = stu3;

        StudentService.print(arr);
    }
```

4.2 无界通配符上限【了解】

> 用 extends 关键字声明，表示参数化的类型可能是所指定的类型，或者是此类型的子类。

我有一个父类 Animal 和几个子类，如狗、猫等，现在我需要一个动物的列表，并且计算这些动物有多少条腿：

动物集合：

```
//定义集合，使用通配符，他可以是动物或者动物的子类！上限是Animal
List<? extends Animal> listAnimals
```

计算函数：

```
static int countLegs (List<? extends Animal > animals ) {
    int retVal = 0;
    for ( Animal animal : animals )
    {
        retVal += animal.countLegs();
    }
    return retVal;
}

static int countLegs1 (List< Animal > animals ){
    int retVal = 0;
    for ( Animal animal : animals )
    {
        retVal += animal.countLegs();
    }
    return retVal;
}

public static void main(String[] args) {
    List<Dog> dogs = new ArrayList<>();
     // 不会报错
    countLegs( dogs );
    // 会报错
    countLegs1(dogs);
}
```

4.3无界通配符下限 【了解】

> < ? super E>
>
> 用 super 进行声明，表示参数化的类型可能是所指定的类型，或者是此类型的父类型，直至 Object

在类型参数中使用 super 表示这个泛型中的参数必须是 T或者 T 的父类。

```
private <T> void test(List<? super T> dst, List<T> src){
    for (T t : src) {
        dst.add(t);
    }
}
```

```
public static void main(String[] args) {
    List<Dog> dogs = new ArrayList<>();
    List<Animal> animals = new ArrayList<>();
    new Test3().test(animals,dogs);
}
// Dog 是 Animal 的子类
class Dog extends Animal {

}
```

dst 类型 “大于等于” src 的类型，这里的“大于等于”是指 dst 表示的范围比 src 要大，因此装得下 dst 的容器也就能装 src 。

五、集合泛型

> 我们日常使用的集合容器，不指定泛型都是Object,比较繁琐，所以建议使用泛型集合!

5.1 List

> compact1, compact2, compact3
>
> java.util
>
> # Interface List

Modifier and Type	Method and Description
boolean	add(E e) 将指定的元素追加到此列表的末尾。
void	add(int index, E element) 在此列表中的指定位置插入指定的元素。
boolean	addAll(Collection<? extends E> c) 按指定集合的Iterator返回的顺序将指定集合中的所有元素追加到此列表的末尾。
boolean	addAll(int index, Collection<? extends E> c) 将指定集合中的所有元素插入到此列表中，从指定的位置开始。
void	clear() 从列表中删除所有元素。
Object	clone() 返回此 ArrayList实例的浅拷贝。
boolean	contains(Object o) 如果此列表包含指定的元素，则返回 true 。
void	ensureCapacity(int minCapacity) 如果需要，增加此 ArrayList实例的容量，以确保它可以至少保存最小容量参数指
void	forEach(Consumer<? super E> action) 对 Iterable的每个元素执行给定的操作，直到所有元素都被处理或动作引发异常。
E	get(int index) 返回此列表中指定位置的元素。
int	indexOf(Object o) 返回此列表中指定元素的第一次出现的索引，如果此列表不包含元素，则返回-1。
boolean	isEmpty() 如果此列表不包含元素，则返回 true 。
Iterator<E>	iterator() 以正确的顺序返回该列表中的元素的迭代器。
int	lastIndexOf(Object o) 返回此列表中指定元素的最后一次出现的索引，如果此列表不包含元素，则返回-1。
ListIterator<E>	listIterator() 返回列表中的列表迭代器（按适当的顺序）。
ListIterator<E>	listIterator(int index)

5.2 Set

> compact1, compact2, compact3
>
> java.util
>
> # Interface Set
>
> - 参数类型
>
> E - 由此集合维护的元素的类型
>
> - All Superinterfaces:
>
> Collection， Iterable
>
> - All Known Subinterfaces:
>
> NavigableSet， SortedSet

Modifier and Type	Method and Description
boolean	add(E e) 如果指定的元素不存在，则将其指定的元素添加（可选操作）。
boolean	addAll(Collection<? extends E> c) 将指定集合中的所有元素添加到此集合（如果尚未存在）（可选操作）。
void	clear() 从此集合中删除所有元素（可选操作）。
boolean	contains(Object o) 如果此集合包含指定的元素，则返回 true 。
boolean	containsAll(Collection<?> c) 返回 true如果此集合包含所有指定集合的元素。
boolean	equals(Object o) 将指定的对象与此集合进行比较以实现相等。
int	hashCode() 返回此集合的哈希码值。
boolean	isEmpty() 如果此集合不包含元素，则返回 true 。
Iterator<E>	iterator() 返回此集合中元素的迭代器。
boolean	remove(Object o) 如果存在，则从该集合中删除指定的元素（可选操作）。
boolean	removeAll(Collection<?> c) 从此集合中删除指定集合中包含的所有元素（可选操作）。
boolean	retainAll(Collection<?> c) 仅保留该集合中包含在指定集合中的元素（可选操作）。
int	size() 返回此集合中的元素数（其基数）。
default Spliterator<E>	spliterator() 在此集合中的元素上创建一个 Spliterator 。
Object[]	toArray() 返回一个包含此集合中所有元素的数组。
<T> T[]	toArray(T[] a)

5.3 Map

> compact1, compact2, compact3
>
> java.util
>
> # Interface Map<K,V>
>
> - 参数类型
>
> K - 由此地图维护的键的类型
>
> V - 映射值的类型
>
> - All Known Subinterfaces:
>
> Bindings , ConcurrentMap <K, V>, ConcurrentNavigableMap <K, V>, LogicalMessageContext , MessageContext , NavigableMap <K, V>, SOAPMessageContext , SortedMap <K, V>

Modifier and Type	Method and Description
void	clear() 从该地图中删除所有的映射（可选操作）。
default V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) 尝试计算指定键的映射及其当前映射的值（如果没有当前映射， null ）。
default V	computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) 如果指定的键尚未与值相关联（或映射到 null ），则尝试使用给定的映射函数计算其值，并将其输入到此映射中，除非 null
default V	computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) 如果指定的密钥的值存在且非空，则尝试计算给定密钥及其当前映射值的新映射。
boolean	containsKey(Object key) 如果此映射包含指定键的映射，则返回 true 。
boolean	containsValue(Object value) 如果此地图将一个或多个键映射到指定的值，则返回 true 。
Set<Map.Entry<K,V>>	entrySet() 返回此地图中包含的映射的Set视图。
boolean	equals(Object o) 将指定的对象与此映射进行比较以获得相等性。
default void	forEach(BiConsumer<? super K,? super V> action) 对此映射中的每个条目执行给定的操作，直到所有条目都被处理或操作引发异常。
V	get(Object key) 返回到指定键所映射的值，或 null如果此映射包含该键的映射。
default V	getOrDefault(Object key, V defaultValue) 返回到指定键所映射的值，或 defaultValue如果此映射包含该键的映射。
int	hashCode() 返回此地图的哈希码值。
boolean	isEmpty() 如果此地图不包含键值映射，则返回 true 。
Set<K>	keySet() 返回此地图中包含的键的Set视图。
default V	merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) 如果指定的键尚未与值相关联或与null相关联，则将其与给定的非空值相关联。
V	put(K key, V value) 将指定的值与该映射中的指定键相关联（可选操作）。