泛型(了解)

本节目标

> - **以能阅读 java 集合源码为目标学习泛型**
> - 了解泛型的分类
> - 了解如何定义泛型类和泛型方法
> - 了解定义泛型类时的一些注意事项
> - 区分泛型的定义和泛型的使用

1. 泛型类的定义

1.1 为什么需要泛型

假设我们自定义了一个简单的数组类，如下：

```
public class MyArray {
    private int[] array = null;
    private int size;
    private int capacity;

    public MyArray(int capacity){
        array = new int[capacity];
        size = 0;
        this.capacity = capacity;
    }

    public void add(int index, int data){
        if(size < capacity)
            array[size++] = data;
    }

    public int get(int index){
        return array[index];
    }

    public int size(){
        return  size;
    }

    public static void main(String[] args) {
        // 创建一个MyArray对象，里面存储int类型的数据
        MyArray m1 = new MyArray(10);
        m1.add(0,1);
        m1.add(1,2);
        m1.add(2,3);
```

```
        for(int i = 0; i < m1.size(); ++i){
            System.out.print(m1.get(i) + " ");
        }
        System.out.println();

        // 但是，如果创建一个MyArray对象，里面想要存储double类型的数据
        // 遗憾的是代码不能通过编译
        MyArray m2 = new MyArray(10);
        m2.add(0,1.0);
        m2.add(1,2.0);
        m2.add(2,3.0);
    }
}
```

通过上述示例发现，**MyArray类中实际只能保存int类型的数据，对于其他类型的数据**比如：double、String或者自定义类型的对象，**根本无法存储。**

想要解决上述问题，最简单的方式就是：**对于不同的类型，分别实现各自的MyArray类即可**，但是估计你可能不愿意。

业界有大佬是按照如下方式改进的：**将上述代码中存储数据的类型全部有int改为Object，因为在Java中Object是所有类的基类。**

```
class Person{
    String name;
    Person(String name){
        this.name = name;
    }
}

public class MyArray {
    private Object[] array = null;
    private int size;
    private int capacity;

    public MyArray(int capacity){
        array = new Object[capacity];
        size = 0;
        this.capacity = capacity;
    }

    public void add(Object data){
        if(size < capacity)
            array[size++] = data;
    }

    public Object get(int index){
        return array[index];
    }

    public int size(){
        return  size;
    }
```

```
    public static void main(String[] args) {
        MyArray m1 = new MyArray(10);
        m1.add(1);
        m1.add(2);
        m1.add(3);

        MyArray m2 = new MyArray(10);
        m2.add(1.0);
        m2.add(2.0);
        m2.add(3.0);

        MyArray m3 = new MyArray(10);
        m3.add(new Person("Peter"));
        m3.add(new Person("Jim"));
        m3.add(new Person("David"));
    }
}
```

经改过之后的MyArray终于任意类型都可以存储了，最后：**代码只需实现一份，但是任意类型都可以存储**，貌似一切都比较美好。

但是大家使用之后，纷纷吐槽：因为Object是所有类的基类，那就意味着可以再一个MyArray中存储不同种类的数据类型喽：

```
public static void main(String[] args) {
        MyArray m = new MyArray(10);
        m.add(1);
        m.add(2.0);
        m.add("Peter");
        m.add(new Person("David"));

        for(int i = 0; i < m.size(); ++i){
            String s = (String)m.get(i);
            System.out.print(s + " ");
        }
    }
```

虽然代码可以通过编译，但是如果想要遍历MyArray中的数据，怎么遍历啊~~~

上述代码再运行期间报错：`Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String`

运行时出错的原因非常简单：上述代码中，由于**MyArray存储数据不都全是String类型的，那如果强转成String类型之后，肯定会发生类型转换异常。**

以上就是JDK1.5之前的解决方式，**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。**为了解决该问题，JDK1.5中引入了泛型。

1.2 泛型的概念

泛型是**java1.5**中增加的一个新特性，通过泛型可以**写与类型无关的代码，即编写的代码可以被很多不同类型的对象所重用**，经常用在一些**通用的代码实现中**，比如：java集合框架中的类几乎都是用泛型实现的。

泛型的本质是：类型参数化。类似函数传参一样，传递不同的实参，函数运行完将会产生不同的结果。

1.3 泛型的分类

泛型主要包含：**泛型类、泛型方法和泛型接口**，后序一一进行介绍。

2. 泛型类

2.1 泛型类的定义

```
class 泛型类名称<类型形参列表> {
    // 这里可以使用类型参数
}

class ClassName<T1, T2, ..., Tn> {
    // 类实现体
}
```

了解： 【规范】类型形参一般使用一个大写字母表示，常用的名称有：

- E 表示 Element
- K 表示 Key
- V 表示 Value
- N 表示 Number
- T 表示 Type
- S, U, V 等等 - 第二、第三、第四个类型

2.2 泛型类的例子

对上述MyArray类进行改造，将其写成泛型类。

注意：以下实现中，没有考虑过多的细节问题，比如插入元素时空间不够如何处理。此处主要演示泛型类的语法规则。

```
// 在实现MyArray泛型类时，E具体代表什么类型实现者不关心
// 当对泛型类进行实例化时，编译器才知道E具体代表什么类型
public class MyArray<E> {
    private E[] array = null;
    private int size;
    private int capacity;

    public MyArray(int capacity){
        // 此处为什么new Object[]，为什么需要强转后文中会解释
        array = (E[])new Object[capacity];
        size = 0;
        this.capacity = capacity;
    }

    public void add(E data){
```

```
        if(size < capacity)
            array[size++] = data;
    }

    public E get(int index){
        return array[index];
    }

    public int size(){
        return  size;
    }
}
```

2.3 泛型类的实例化

2.3.1 实例化语法

泛型类<类型实参> 变量名; 定义一个泛型类引用。 **new 泛型类<类型实参>(构造方法实参)**; 实例化一个泛型类对象。

```
public static void main(String[] args) {
    // 将泛型类使用String类型来实例化，表明m中只能存放String类型的对象
    MyArray<String> m = new MyArray<>(10);
    m.add("Peter");
    m.add("David");
    m.add("Jim");

    // 编译失败：因为在实例化时，已经明确其内部只能存储String类型的对象
    // Person对象和String对象之间不能转换
    // m.add(new Person("Lily"));

    for(int i = 0; i < m.size(); ++i){
        // 此处从m中获取到的成员，再不需要进行强制类型转换了
        String s = m.get(i);
        System.out.print(s + " ");
    }
}
```


注意：

1. **右侧<>中的类型可以省略**

   ```
   MyArray<String> m = new MyArray<>(10);
   ```

   在new MyArray<>(10)对象时，<>中未明确指明类型，编译器会根据=左侧<>中的类型来推演。

2. **左侧<>中的类型不能省略**

   ```
   MyArray<> m = new MyArray<String>(10); // 省略之后，编译失败
   ```

   编译器在推演时，是根据左侧类型来推演右侧的。

3. **虽然右侧<>可以不用写类型，但是<>不能省略**

```
MyArray<String> m = new MyArray();  // 自己永远不要这么用
```

上述代码，会产生编译警告

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

可以使用 `@SuppressWarnings` 注解进行警告压制

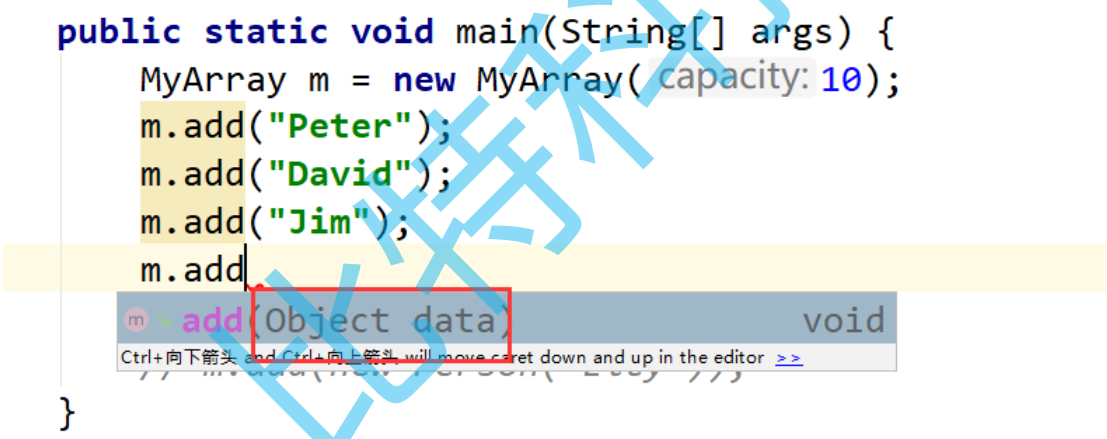
```
@SuppressWarnings("unchecked")
```

2.3.2 裸类型(Raw Type) （了解）

裸类型是一个泛型类但没有带着类型实参，例如 MyArray 就是一个裸类型

```
MyArray list = new MyArray();
```

注意： 我们不要自己去使用裸类型，**裸类型是为了兼容老版本的 API 保留的机制**



可以看到，如果类型裸类型时，泛型类中的E会被默认当成Object来处理，那就回到了JDK1.5之前的情况。

下面的类型擦除部分，我们也会讲到编译器是如何使用裸类型的。

2.4 泛型类的定义-类型边界

在定义泛型类时，**有时需要对传入的类型变量做一定的约束，可以通过类型边界来约束。**

```
class 泛型类名称<E extends U> {
    ...
}
```

在实例化时，E只能是U的子类，否则编译会报错。

```
// 将来对MyArray进行实例化时，实例化的类型必须要是Animal的子类才可以
public class MyArray<E extends Animal> {
```

```
    private E[] array = null;
    private int size;
    private int capacity;

    public MyArray(int capacity){
        array = (E[])new Object[capacity];
        size = 0;
        this.capacity = capacity;
    }

    // ...

    public static void main(String[] args) {
        // 编译成功，因为Dog是Animal的子类
        MyArray<Dog> m1 = new MyArray<>(10);
        m1.add(new Dog("旺财"));
        m1.add(new Dog("二哈"));

        // 编译成功，因为Cat是Animal的子类
        MyArray<Cat> m2 = new MyArray<>(10);
        m2.add(new Cat("肥波"));
        m2.add(new Cat("加菲"));

        // 编译失败，因为String不是Animal的子类
        MyArray<String> m3 = new MyArray<>(10);
    }
}
```

注意：**没有指定类型边界 E，可以视为 E extends Object**

2.5 泛型类的使用-通配符(Wildcards)

2.5.1 基本

`?` 用于在泛型的使用，即为通配符

示例

```
public class MyArray<E> {...}

// 可以传入任意类型的 MyArray
public static void printAll(MyArray<?> m) {
    ...
}

// 以下调用都是正确的
printAll(new MyArray<String>());
printAll(new MyArray<Integer>());
printAll(new MyArray<Double>());
printAll(new MyArray<Number>());
printAll(new MyArray<Object>());
```

2.5.2 通配符-上界

语法

```
<? extends 上界>
```

示例

```
// 传入类型实参是 Animal 子类的任意类型的 MyArray
public static void printAll(MyArray<? extends Animal> m) {
    ...
}

// 以下调用都是正确的
printAll(new MyArray<Cat>());
printAll(new MyArray<Dog>());

// 以下调用是编译错误的
printAll(new MyArray<String>());
printAll(new MyArray<Object>());
```

注意： 需要区分 **泛型使用**中的通配符上界 和 **泛型定义**中的类型上界

2.5.3 通配符-下界

语法

```
<? super 下界>
```

示例

```
// 可以传入类型实参是 Cat 父类的任意类型的 MyArray
public static void printAll(MyArray<? super Cat> list) {
    ...
}

// 以下调用都是正确的
printAll(new MyArrayList<Cat>());
printAll(new MyArrayList<Animal>());
printAll(new MyArrayList<Object>());
// 以下调用是编译错误的
printAll(new MyArrayList<String>());
printAll(new MyArrayList<Dog>());
```

2.6 泛型中的父子类型（重要）

```
public class MyArray<E> { ... }

// MyArray<Object> 不是 MyArray<Animal> 的父类型
// MyArray<Animal> 也不是 MyArray<Cat> 的父类型

// 需要使用通配符来确定父子类型
// MyArray<?> 是 MyArray<? extends Animal> 的父类型
// MyArray<? extends Animal> 是 MyArrayList<Dog> 的父类型
```

3. 泛型方法

3.1 语法格式

```
方法限定符 <类型形参列表> 返回值类型 方法名称(形参列表) { ... }
```

3.2 示例

```
public class Util {
    public static <E> void swap(E[] array, int i, int j) {
        E t = array[i];
        array[i] = array[j];
        array[j] = t;
    }
}


// 没有显式指定类型，编译期间需要进行类型推到
Integer[] a = { ... };
swap(a, 0, 9);

String[] b = { ... };
swap(b, 0, 9);


// 显式指定类型，编译期间，不用进行类型推导
Integer[] a = { ... };
Util.<Integer>swap(a, 0, 9);

String[] b = { ... };
Util.<String>swap(b, 0, 9);
```


4. 泛型接口

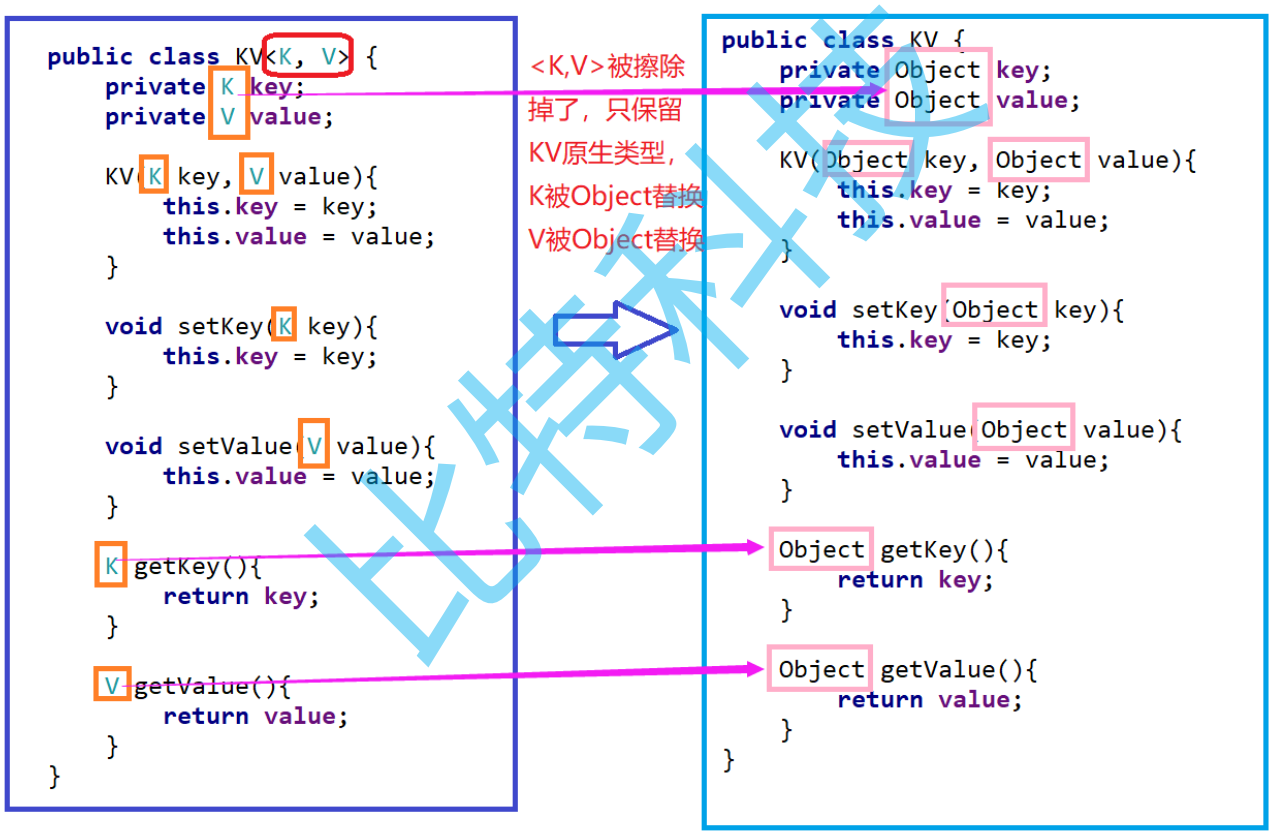
给大家看个示例，这个主要后面来讲。

```java
// 与定义泛型类非常相似---该接口主要是用来对对象进行比较的
// 比如sort方法，可以排任意类型的数据，可以排升序也可以排降序
// sort在排序过程中，元素的比较规则就可以通过来实现该 泛型接口 来处理
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

5. 类型擦除

5.1 什么是类型擦除

Java泛型这个特性是从**JDK 1.5才开始加入**的，因此**为了兼容之前的版本，Java泛型的实现采取了“伪泛型”的策略**，即**Java在语法上支持泛型，但是在编译阶段会进行所谓的“类型擦除”（Type Erasure）**，将所有的泛型表示（尖括号中的内容）都替换为具体的类型（其对应的原生态类型），就像完全没有泛型一样。 即**泛型类和普通类在 java 虚拟机内是没有什么特别的地方。**



5.2 类型擦除的规则

泛型的类型擦除原则是：

- 消除类型参数声明，即**删除<>及其包围的部分。**
- 根据类型参数的上下界推断并替换所有的类型参数为原生态类型：如果**类型参数是无限制通配符或没有上下界限定则替换为Object**，如果**存在上下界限定则根据子类替换原则取类型参数的最左边限定类型**（即父类）。
- 为了**保证类型安全，必要时插入强制类型转换代码。**
- 自动产生**“桥接方法”以保证擦除类型后的代码仍然具有泛型的“多态性”。**

```
// 1. 无限制类型擦除---<E>和<?>类型参数都被替换为Object
class MyArray<E> {
    // E 会被擦除为 Object
}

// 2. 有限制类型擦除---<T extends Animal>和<? extends Animal>的类型参数被替换为Animal
//                    <? super Animal>被替换为Object
class MyArray<E extends Animal> {
    // E 被擦除为 Animal
}


// 3. 擦除方法中的类型参数
public class Util {
    public static <E> void swap(E[] array, int i, int j) {
        // ...
        // <E>删除掉   E被擦除为Object
    }
}
```

总结： 即类型擦除主要看其类型边界而定

注意： 编译器在类型擦除阶段在做什么?

1. 将类型变量用擦除后的类型替换，即 Object 或者 Comparable
2. 加入必要的类型转换语句
3. 加入必要的 `bridge method` 保证多态的正确性

6. 泛型的优缺点

泛型的优点：

1. 提高代码的复用性
2. 提高开发效率
3. 可以实现一些通用类型的容器或算法

泛型的缺点：

1. 泛型类型参数不支持基本数据类型
2. 无法实例化泛型类型的对象
3. 无法使用泛型类型声明静态的属性
4. 无法使用 instanceof 判断带类型参数的泛型类型
5. 无法创建泛型类数组
6. 无法 create、catch、throw 一个泛型类异常（异常不支持泛型）
7. 泛型类型不是形参一部分，无法重载