**为什么要有泛型**

我们直接将**需要的类型传入，就像一个参数一样，获取的时候就是我们需要的类型，这就是类型参数化**。

### 泛型的使用

我们可以将泛型理解为一个 大于一般类型，小于Object类型 的类型，比如，T一定是Object，但是Object不一定是T，所以T小于Object，String或Integer可以是T，但是T不一定是String或Integer，所以T大于String，所以可以简单的理解为：

**一般类型 < 泛型 < Object类型**

1 泛型类和泛型接口

泛型类的使用很简单，比如上面我们创建的WrapperObject类就是泛型类，它有个特点，就是**类名后面跟上一个用尖括号括起来的类型，当然可以有多个**，比如:

public class Fuck<A, B, C> {

private A a;

private B b;

private C c;

}

泛型接口跟泛型类是一样的，因为接口也是类的一种，比如常见的List，Map接口:

public interface List<E> extends Collection<E> {

}

可以看到，泛型接口/类 也是可以继承的，跟一般类没啥区别。

2 泛型方法

##### 泛型作为参数

很简单，我们需要在返回类型之前添加**尖括号括起来的类型**，然后在参数列表就可以像一般类型一样的使用，比如:

public <T> void test(T t) {

}

public static <T> void test(T t) {

}

##### 泛型作为返回值

跟参数一样，返回类型之前加上**尖括号括起来的类型**，然后返回类型改为泛型即可:

public <T> T test(T t) {

return t;

}

public static <T> T test(T t) {

return t;

}

**泛型的界**

1 泛型的上界

我们需要一个**是Number的子类的泛型**，这就等价于限制了泛型的上界，也就是指定了它爹，代码如下:

// 通过<T extends Number>来指定上界

public <T extends Number> T add(T a, T b) {

return a + b;

}

public void test() {

int a = 10;

int b = 20;

// 传入的是int，返回的也是int，并且因为传入的是Number的子类，所以能使用加法

int c = add(a, b);

}

泛型的上界可以为类，接口，甚至另一个泛型，也可以有多个泛型，比如:

public <E, T extends E> T add(T a, T b) {

}

这里需要注意一点，**两个"泛型类"之间不具有继承关系**，比如:Integer是Number的子类，但是List<Integer>不是List<Number>的子类。因为List<Integer>整体是一个类型。

2 泛型的下界

泛型没有下界，没有下界，没有下界！如果要使用下界，可以使用通配符?，比如:

public void test(List<? super Integer> list) {

}

3 泛型多边界

我们知道泛型可以指定上界，我们又知道泛型就等价于一般类型，一般类型可以继承一个父类，可以实现多个接口，那么泛型指定一个上界，可以看成是继承一个父类，那么可不可以再指定一个接口上界呢，可以！

public <T extends Number & Serializable> void test() {

}

我们可以指定一个类和多个接口，等价于java类的单继承和多实现。而且这里面有个规则，类一定要紧跟在extends后面，接口使用&连接在后面。接口可以有多个，类只能有一个。比如:

// 错误，因为Number是类，需要紧跟在extends后面。

public <T extends Serializable & Number> void test() {

}

// 正确，可以有多个接口

public <T extends Number & Serializable & Comparable> void test() {

}

Tips: kotlin的泛型写法比较费劲，需要用where连接，比如

fun getName(origin: T): T where T : Number, T : Parcelable {

return origin

}

class User where T : Number, T : Parcelable {

}

等价的java版本的代码就是:

private <T extends Number & Serializable> T getName() {

return null;

}

class User<T extends Number & Serializable> {

}

**泛型的实现原理**

1 泛型擦除

java代码在编译的时候，会将所有的泛型给删掉，变成不含泛型的代码，这就叫**泛型擦除**，那么既然擦除了，运行时又是怎么知道泛型的实际类型呢。因为jvm在擦除的时候，除了将原有泛型全部用Object替换外，还会添加对应的类型强转代码，比如:

public class TypeTest<T> {

private T t;

public TypeTest(T t) {

this.t = t;

}

public T getContent() {

return t;

}

}

public void test2() {

TypeTest<String> hello = new TypeTest<>("hello");

String content = hello.getContent();

}

在泛型擦除后，也就是jvm编译后，就会转变为如下代码:

// 这里将T擦除，对应的T替换为Object

public class TypeTest {

private Object t;

public TypeTest(Object t) {

this.t = t;

}

public Object getContent() {

return t;

}

}

public void test2() {

TypeTest hello = new TypeTest("hello");

// 这里直接进行强转

String content = (String) hello.getContent();

}

所以，在编译后的代码中，是没有泛型的影子的，也就是说，泛型只在**编辑代码期生效，就是为了coder方便书写**，所以java的泛型，准确的说是jvm的泛型，也叫**伪泛型**。

#### 2 Signature属性

上面我们知道，jvm在编译期间会擦除所有泛型，但是我们通过反射却可以获取到泛型信息，而反射工作在运行时，运行时肯定在编译期之后，那么为什么还能获取到泛型相关信息，原因就是: **Signature**属性。

Signature属性是.class文件 属性表集合 里面的 一个属性，它记录了泛型的相关信息，编译时，会将原有的Coder属性里面的泛型信息擦除，然后记录到Signature属性里面，反射api会直接从这里面去获取有关泛型的信息。