HenCoder Plus 第 34-35 课 讲义

泛型

泛型类型的创建

泛型类的创建

```
public class Wrapper<T> {
    T instance;

    public T get() {
        return instance;
    }

    public void set(T newInstance) {
        instance = newInstance;
    }
}
```

泛型接口的创建

```
public interface Shop<T> {
    T buy();
    float refund(T item);
}
```

继承

```
public class AppleShop implements Shop<Apple> {
  Apple buy() {
    return new Apple();
  }
}
```

```
public interface RepairableShop<T> extends Shop<T>
{
    void repair(<T> item);
}
```

多个类型参数

```
public interface HenCoderMap<K, V> {
  public void put(K key, V value);

  public V get(K key);
}
```

```
public interface SimShop<T, C> extends Shop<T> {
    T buy(float money);
    float refund(T item);
    C getSim(String name, String id);
}
```

`<T extends Xxx>`

```
public interface FruitShop<T extends> extends
Shop<T> {
}
```

`<? extends Xxx>`

```
ArrayList<? extends Fruit> fruitList = new
ArrayList<AppleList>();
```

用处?

```
float totalWeight(List<? extends Fruit> fruits) {
  float weight = 0;
  for (Fruit fruit : fruits) {
    weight += fruit.getWeight();
  }
  return weight;
}
```

<? super Xxx>

```
List<? super Apple> appleList = new
ArrayList<Fruit>();
```

用处?

```
public class Apple implements Fruit {
  public void addMeToList(List<? super Apple> list)
{
    list.add(this);
  }
}

apple.addMeToList(fruits);
```

泛型方法

```
public <T extends View> T findViewById(@IdRes int
id) {
  return getWindow().findViewById(id);
}

protected <T extends View> T
findViewTraversal(@IdRes int id) {
  if (id == mID) {
    return (T) this;
  }
  return null;
}
```

Type parameter 和 Type argument

- Type parameter：
  - `public class Shop<T>` 里面的那个 `<T>`；
  - 表示我要创建一个 `Shop` 类，它的内部会用到一个统一的类型，这个类型姑且称他为 `T`。
- Type argument：
  - 其它地方尖括号里的全是 Type argument，比如 `Shop<Apple> appleShop;` 的 `Apple`；
  - 表示「那个统一代号，在这里的类型我决定是这个」。
- `Type Parameter` 和 `Type Argument`；泛型的创建和泛型的实例化

泛型的意义

- 泛型的意义在于：泛型的创建者让泛型的使用者可以在使用时（实例化时）细化类型信息，从而可以触及到「使用者所细化的子类」的 API。

  或者，泛型是「有远见的创造者」创造的「方便使用者」的工具。
  - 所以泛型参数要么至少是一个方法的返回值类型：

    ```
    T buy();
    ```

  - 要么是放在一个接口的参数里，等着实现类去写出不同的实现：

    ```
    public int compareTo(T o);
    ```

- 不过，泛型由于语法自身特性，所以也有一个延伸用途：用于限制方法的参数类型或参数关系：

```
public <E extends Runnable, Serializable> void
someMethod(E param);
```

```
public <T> merge(T item, List<T> list) {
  list.add(item);
}
```

T

- 写在类名（接口名）右边的括号里，表示 Type parameter 的声明，「我要创建一个代号」；
- 写在类里的其他地方，表示「这个类型就是我那个代号的类型」；
- 只在这个类里有用，出了类就没用了（泛型方法是，只在这个方法里有用，出了这个方法就没用了）。

?

只能写在泛型声明的地方，表示「这个类型是什么都行，只要不超出 `? extends` 或者 `? super` 的限制」。

泛型类型创建的重复

- `<T>` 的重复：

  ```
  public class RefundableShop<T> extends Shop<T> {
    float refund(T item);
  }
  ```

  表示对父类（父接口）的扩展。

- 类名的重复：

  ```
  public class String implements
  Comparable<String> {
    public native int compareTo(String
  anotherString);
  }
  ```

  同样表示对父类（父接口）的扩展（囧）。

类型擦除

- 运行时，所有的 `T` 以及尖括号里的东西都会被擦除；
- `List` 和 `List<String>` 以及 `List<Integer` 都是一个类型；
- 但是所有代码中声明的变量或参数或类或接口，在运行时可以通过反射获取到泛型信息；
- 但但是，运行时创建的对象，在运行时通过反射也获取不到泛型信息（因为 class 文件里面没有）；
- 但但但是，有个绕弯的方法就是创建一个子类（哪怕用匿名类也行），用这个子类来生成对象，这样由于子类在 class 文件里就有，所以可以通过反射拿到运行时创建的对象的泛型信息。

  比如 `Gson` 的 `TypeToken` 就是这么干的。

Kotlin 的泛型

- 场景跟 Java 一样，不过用法有一点不一样；
- Java 的 `<? extends>` 在 Kotlin 里写作 `<out>`；Java 的 `<? super>` 在 Kotlin 里写作 `<in>`；
- 另外，Kotlin 还增加了 `out T` `in T` 的修饰，来在类或接口的声明处就限制使用，这样你在使用时就不必再每次都写；
- Kotlin 还有个 * 号，表示「解除限制」。

问题和建议?

课上技术相关的问题，都可以在学员群里和大家讨论，我一旦有时间也都会来解答。如果我没来就 @ 我一下吧！

具体技术之外的问题和建议，都可以找丢物线（微信：diuwuxian），丢丢会为你解答技术以外的一切。

更多内容：

- 网站：https://hencoder.com
- 微信公众号：HenCoder