Kotlin 中的 data class 和 sealed class


牛蛙点点申请出战 LV.3
2020年05月20日 14:17 · 阅读 4326

关注

Kotlin 中的 data class

在使用 java 的时候，我们经常会重写类的 `equals` 、 `hashCode` 和 `toString` 方法。这些方法往往都是模板化的。在 kotlin 中提供了更为简便的方法让我们使用一行代码搞定这些工作。这就是 `data class` 。

kotlin 复制代码

```kotlin
// 定义一个 Person 类
data class Person(val name: String, val age: Int) {
}
```

写好上面的代码之后， `Person` 类中的上述几个方法的重写就由 kotlin 帮我们自动完成了。运行下面的代码

ini 复制代码

```ini
fun main() {
    val p1 = Person("Jack", 24)
    val p2 = Person("Jack", 24)

    val p3 = Person("Jack", 32)
    val p4 = Person("Rose", 31)

    println(p1 == p2)

    println(p1 === p2)

    println(p1 == p3)

    println("""
        p1 hashCode = ${p1.hashCode()}
        p2 hashCode = ${p2.hashCode()}
        p3 hashCode = ${p3.hashCode()}
        p4 hashCode = ${p4.hashCode()}
    """.trimIndent())

    println("p1 = $p1")
```

```
}
```

结果如下：

ini 复制代码

```
true
false
false
p1 hashCode = 71328761
p2 hashCode = 71328761
p3 hashCode = 71328769
p4 hashCode = 79149200
p1 = Person(name=Jack, age=24)
```

可以看到 `equals` 、 `hashCode` 和 `toString` 方法可以直接调用，并且已经被覆写了。

data class 究竟做了什么?

`data class` 是如何做到上述实现的呢？查看 `Person` 类的字节码反编译得到的 java 代码，如下

kotlin 复制代码

```
public final class Person {
   @NotNull
   private final String name;
   private final int age;

   @NotNull
   public final String getName() {
      return this.name;
   }

   public final int getAge() {
      return this.age;
   }

   public Person(@NotNull String name, int age) {
      Intrinsics.checkParameterIsNotNull(name, "name");
      super();
      this.name = name;
      this.age = age;
   }

   @NotNull
```

```
public final String component1() {
   return this.name;
}

public final int component2() {
   return this.age;
}

@NotNull
public final Person copy(@NotNull String name, int age) {
   Intrinsics.checkParameterIsNotNull(name, "name");
   return new Person(name, age);
}

// $FF: synthetic method
public static Person copy$default(Person var0, String var1, int var2, int var3, Object va
   if ((var3 & 1) != 0) {
      var1 = var0.name;
   }

   if ((var3 & 2) != 0) {
      var2 = var0.age;
   }

   return var0.copy(var1, var2);
}

@NotNull
public String toString() {
   return "Person(name=" + this.name + ", age=" + this.age + ")";
}

public int hashCode() {
   String var10000 = this.name;
   return (var10000 != null ? var10000.hashCode() : 0) * 31 + this.age;
}

public boolean equals(@Nullable Object var1) {
   if (this != var1) {
      if (var1 instanceof Person) {
         Person var2 = (Person)var1;
         if (Intrinsics.areEqual(this.name, var2.name) && this.age == var2.age) {
            return true;
         }
      }

      return false;
   } else {
      return true;
```

```
        }
    }
}
```

从上面可以一目了然地看见 kotlin 是如何构造 data class 的。由于我定义的成员变量是 `val` 不可变类型的，所以没有 Getter 和 Setter。

- 对于 `equals` 方法，对于 `var1` ，如果它的地址和自己不同，那么先检查其是否是 `Person` 类型，如果是，则逐个对比每个成员变量是否相等。这里用到的 `Intrinsics.areEqual(Object o1, Object o2)` 定义如下：

```sql 复制代码
public static boolean areEqual(Object first, Object second) {
      return first == null ? second == null : first.equals(second);
   }
```

  最终调用了被比较对象的 `equals` 方法。 对于 Int 类型的成员变量 `age` ，直接使用 `==` 比较。

  这是覆写一个类的 `equals` 方法的常规写法。

- `hashCode` 和 `toString` 方法也是我们常规覆写的套路。

- 提供了 `component1` 和 `component2` 两个方法来获取成员变量。这两个方法可以用来做解构声明。如下：

```kotlin 复制代码
val (name, age) = Person("Rose", 43)
println("$name, $age")
// (name, age) 就是解构声明，name 对应 component1 ，age 对应 component2
```

- 提供了 `copy` 方法构造一个 `Person` 对象。有 // $FF: synthetic method 注释的 `copy$default` 方法是给 kotlin 编译器调用的，我们用不到。

  如果我们只想要 `age` 不同的 `Person` ，可以这样写

```ini 复制代码
val newPerson = p1.copy(age = 30)
```

  编译器编译到这句代码时，会帮我们调用 `copy$default` 来构造一个 `name` 值和 p1 一样的 `newPerson` 对象 。

总之， `data class` 就是用常规套路来生成一个已经覆写好上述方法的类。

如果 Person 类不需要自动生成 age ， 只需要把 age 从主构造函数中拿出，放到类体中就可以。如下

kotlin 复制代码

```kotlin
data class Person(val name:  String) {
    val age: Int = 0
}
```

Kotlin 中的 sealed class

`sealed class` 是一种同时拥有枚举类 `enum` 和 普通类 `class` 特性的类，叫做密封类。使用起来很简单，如下

kotlin 复制代码

```kotlin
sealed class Result
class Success(val code: Int) : Result()
class Exception(val code: Int, val message: String) : Result()
```

在同一个 kotlin 文件中声明三个类。首先声明 `sealed class` 类 `Result` , 然后定义出两个子类 `Success` , `Exception` 继承自 `Result` 。注意，密封类及其子类必须声明在同一个 kotlin 文件中。

这是一个非常常见的场景。比如对于网络请求的结果 Result ， 往往只有两种类型，成功 Success 或者是失败 Exception 。使用普通的类不能把限制关系表达出来，使用枚举类则无法灵活地自定义需要的类的内容。这时候， `sealed class` 就派上用场了。比如在处理结果 `Result` 的时候:

kotlin 复制代码

```kotlin
fun handleResult(result: Result): String{
    return when(result) {
        is Success -> {
            "success"
        }
        is Exception -> {
            "exception"
        }
    }
}
```

这样，对于 `handleResult` 的入参就做了类型的限制，防止传入类型不匹配的参数。

还有一个好处是，使用密封类的话， `when` 表达式可以覆盖所有情况，不需要再添加 `else` 语句（表达式即有返回值的 `when` ，没有返回值的称为 `when` 语句）。

sealed class 究竟做了什么?

同样地，让我们来看看 `sealed class` 在 java 层面做了什么，实现了前面的效果。 上述密封类反编译得到的 java 代码如下：

java 复制代码

```java
public final class Exception extends Result {
   private final int code;
   @NotNull
   private final String message;

   public final int getCode() {
      return this.code;
   }

   @NotNull
   public final String getMessage() {
      return this.message;
   }

   public Exception(int code, @NotNull String message) {
      Intrinsics.checkParameterIsNotNull(message, "message");
      super((DefaultConstructorMarker)null);
      this.code = code;
      this.message = message;
   }
}
// Success.java
import kotlin.Metadata;
import kotlin.jvm.internal.DefaultConstructorMarker;
public final class Success extends Result {
   private final int code;

   public final int getCode() {
      return this.code;
   }

   public Success(int code) {
      super((DefaultConstructorMarker)null);
      this.code = code;
   }
}
// Result.java
import kotlin.Metadata;
```

```
import kotlin.jvm.internal.DefaultConstructorMarker;
// 最重要的地方
public abstract class Result {
   private Result() {
   }

   // $FF: synthetic method
   public Result(DefaultConstructorMarker $constructor_marker) {
      this();
   }
}
```

可以看到， `Result` 类其实是一个抽象类， `Success` 和 `Exception` 继承了这个抽象类。 `Result` 类的构造函数是私有的，不能在外部访问到。

通过继承这个抽象类，达到限制类型的做法。

这其实和 java 中使用接口来限定参数类型的做法类似，很好理解。

分类： Android 标签： Kotlin