OSC源创会 Opensource Innovation Meetup | IT大咖说 知识分享平台

原来Android开发也可以那么甜蜜

——刘自鹏

2010

JetBrains以创建一种 与 Java 100% 兼容，但比 Java 更安全、更简洁、更灵活，且不会过于复杂的语言为目标推出了Kotlin,名称来源为俄罗斯圣彼得堡附近的一个岛屿的名字

2012

JetBrains使用Apache 2许可证，开源了Kotlin

2016 Kotlin 1.0正式版发布

2017.3 Kotlin 1.1正式版发布，协程进入实验性阶段，并正式对Java Script提供支持。

2017.4 Kotlin Native技术预览版发布，也许未来我们可以告别JNI?

2017.5

Google IO 2017，谷歌宣布，Kotlin正式成为Android开发的官方编程语言，再也不是野路子了。

Kotlin有何特点?

Kotlin有何特点?

语法更加简洁，更少的代码，完成更多的事

Null Safe，对空指针说不

即使在Java6上，也能够使用的lambda表达式，基本告别匿名内部类。

我们可以使用扩展函数去为一个类扩展更多的特性，即使，这个类我们无法访问。

与Java近乎完美的互操作性

当然，还有更多……

Data Class

Data Class

使用Java的时候，建立一个Bean，至少是这样

```
public class Human {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

也许，还需要增加一些额外的方法，比如这样

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Human human = (Human) o;

    if (age != human.age) return false;
    return name.equals(human.name);
}

@Override
public int hashCode() {
    int result = name.hashCode();
    result = 31 * result + age;
    return result;
}

@Override
public String toString() {
    final StringBuffer sb = new StringBuffer("Human{");
    sb.append("name='").append(name).append('\'');
    sb.append(", age=").append(age);
    sb.append('}');
    return sb.toString();
}
```

我可以选择使用生成器去生成，但无论如何，添加这些视乎通用的代码，仍然是重复劳动，意义视乎不大

Data Class

而在Kotlin中，它变成了这样

```
class Human {
    var name: String? = null
    var age: Int = 0

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other?.javaClass != javaClass) return false

        other as Human

        if (name != other.name) return false
        if (age != other.age) return false

        return true
    }

    override fun hashCode(): Int {
        var result = name?.hashCode() ?: 0
        result = 31 * result + age
        return result
    }

    override fun toString(): String {
        return "Human(name=$name, age=$age)"
    }
}
```

缩短了一半，在Kotlin中，每一个属性都默认带有getter、setter，如果你的业务有需要，你可以去重写getter、setter

```
var name: String? = null
    get() = "$field :)"
    set(value) {
        if (value != null) {
            field = value
        }
    }
```

可以发现如果需要toString()等方法，任然需要去不断的生成，代码还是很多

Data Class

Kotlin为我们提供了一个Data Class来解决这类问题，使用Data关键字修饰一个类，只需要如下写法，就能得到一个完整的Bean

```
data class Human (val name:String,val age:Int){

}
```

这段代码，包括getter、setter、toString、hashCode、equals、copy甚至额外的一些函数都为我们创建好了。
因为这个类中没有别的代码，甚至可以去掉大括号。

```
data class Human (val name:String,val age:Int)
```

通过Data Class，50行代码，被简化为了一行

空安全

空安全

在Java中，我们最头疼的就是NullPointerException，尤其在代码量一大，我们很难为每一处可能为空的地方添加保护，而Kotlin，它是空安全的。

空安全，指的并不是在Kotlin中不存在空指针，而是Kotlin会强制空检查，找出可能有空指针隐患的地方。

这样的代码，在Java中会被编译通过，运行起来我们会得到一个空指针异常。

```
Movie movie = null;
movie.toString();
```

而Kotlin中,IDE会给我们一个错误，要求我们指定这个变量是可能为空的。

```
val movie: Movie = null
movie.toString()
```

而最终，它应该是这样的，我们通过?来表示这个变量是可能为空的，并且在使用的时候通过?表示如果不为空才执行。

```
val movie: Movie? = null
movie?.toString()
```

空安全

如果你确定这个变量不为空，你也可以使用!! ，它表示这个变量如果不为空则执行，如果为空则抛出一个异常，但是这种用法要尽力避免，因为如果满屏!!，显然是我的代码没有处理好。

```
val movie: Movie? = null
movie!!.toString()
```

如果你希望在你的变量为空的情况下主动抛出一个异常或做点别的什么事，你可以使用Elvis操作符，它允许使用?:do something的写法来表示如果为空要做什么。

```
val movie: Movie? = null
movie?.toString() ?: throw NullPointerException()
```

或者

```
fun main(args: Array<String>){
    val movie: Movie? = null
    movie?.toString() ?: say()
}

fun say() {
    print("hello kotlin")
}
```

空安全

也许你需要在一个代码块中多次使用一个可为空的对象，那么你可以使用let函数

```
val movie: Movie? = null
movie?.let {
    movie.toString()
    movie.subjects
    movie.count
    movie.title
}
```

有时候我们需要一些变量在使用的时候才被初始化，所以申明的时候会将其置空，但这样会导致后续即使初始化了变量，仍然被视为有风险的，那么会导致很多?或者!!的出现，这时你可以使用懒加载。

```
private var list: MutableList<String>? = null

fun test() {
    list = mutableListOf()
    list!!.add("aa")
    list!!.add("bb")
}
```

```
private val list: MutableList<String> by lazy {
    mutableListOf<String>()
}

fun test() {
    list.add("aa")
    list.add("bb")
}
```

空安全

还有一种情况，某个变量可能需要在某个事件被触发，才初始化，比如网络请求后，那么这个变量同样会被申明为一个全局的可空变量，这时又会出现?和!!满天飞的情况。这种情况可以使用Lateinit关键字去申明一个变量。

```
var movie: Movie? = null

fun getMovieSuccess(result: Movie) {
    movie = movie
    print(movie?.title)
}
```

```
lateinit var movie: Movie

fun getMovieSuccess(result: Movie) {
    movie = movie
    print(movie.title)
}
```

扩展函数

扩展函数

扩展函数是Kotlin非常重要的一个特性，Kotlin的许多语法都是通过扩展函数来实现的。

扩展函数允许我们在一个新已有的类上增加新的方法，即使这个类我们无法访问，并且可以使用this关键字与这个类的所有public方法。 嗯.. 再举一个栗子🌰

平时我们在Android中打log挺麻烦的，我希望在任何地方都可以通过一个logd或logi只传一个字符串来打印一个log，那么可以这么做。

```
fun Any.logd(msg: String) {
    Log.d(this.javaClass.simpleName, msg)
}
```

扩展函数

值得一提的是Kotlin中除了函数，属性也是可以扩展的，比如在Kotlin中，使用TextView获取Text的时候，只需要textView.text ,而设置一个Text只需要textView.text = str，他的背后其实是将setText() getText()两个方法通过扩展属性替换为了Kotlin的getter、setter.

```
public var TextView.text: CharSequence
    get() = getText()
    set(v) = setText(v)
```

lambda表达式

在使用Java开发Android的时候，我们通常一个项目中要写非常多的回调接口或类来完成数据的回调，诸如一下的写法，大家应该都非常熟悉。

```
mButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {

    }
});
```

而在Kotlin中，只需要

```
mButton.setOnClickListener {

}
```

而这就是lambda表达式的功劳，有了它我们基本可以告别匿名内部类

lambda表达式

在Kotlin中，我们可以通过 表达式名:(表达式参数)->返回类型 来定义一个lambda表达式。
在Android开发中使用RecyclerView的时候，我们需要在Adapter中去写item的点击事件，然后通过监听接口将事件回调出去，那么在Kotlin中我们可以这么做

```
class Adapter(var mShots: MutableList<Shot>, val itemClick: (Int) -> Unit)

holder.itemView.mItemCard.setOnClickListener {
    itemClick(position)
}

mListAdapter = Adapter(mShots, { position ->
    log("点击了$position")
})
```

仔细看看和我们原来写回调的方式是类似的，只是它更加简单了

lambda表达式

当我们的lambda表达式只有一个参数的时候，实际上可以不用给参数名而用it关键字来充当这个参数

```
mListAdapter = Adapter(mShots, {
    log("点击了$it")
})
```

而当一个函数中只有一个参数是lambda表达式的时候,我们甚至可以将其代码块抽出到参数列表外

```
mListAdapter = Adapter(mShots){
    log("点击了$it")
}
```

而当这个lambda表达式的代码块中只有一个方法调用，而这个方法的参数类型刚好就是这个lambda表达式的参数类型的时候，我们甚至可以将代码块去掉直接使用::

```
mListAdapter = Adapter(mShots, ::print)
```

有些同学可能会疑问，在Kotlin中是不是就不能使用匿名内部类了呢？其实是可以的，只不过，它变成了这样

```
mButton.setOnClickListener(object : View.OnClickListener {
    override fun onClick(view: View) {

    }
})
```

我们看到大部分Java中类似setOnClickListener这样的方法的匿名内部类参数都被转成了lambda表达式，这是因为Kotlin有一个SAM转换的机制，它能够将Java中的匿名内部类转换为lambda表达式，但是它的能力是有限的，它要求这个匿名内部类必须是接口，且只能有一个回调方法。

lambda表达式

Kotlin同样支持stream操作，比如它允许我们通过forEach函数直接遍历集合

```
val list = listOf<String>("aa", "bb", "cc")
list.forEach { print(it) }
```

而Kotlin的流式API就是lambda表达式与扩展函数的典型实例，以forEach为例，它的背后是这样的

```
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit {
    for (element in this) action(element)
}
```

Anko

Anko

既然前面已经讲了扩展函数和lambda表达式，那么就不得不顺带提一下Kotlin的招牌扩展库Anko了
Anko是Kotlin团队开发的一个扩展库，该库的大多数特性基于扩展函数，可以大大简化不少操作。
该库分为了Commons、Layout、SQLite、Coroutines等几个部分。

```
doAsync {
    //请求网络

    uiThread {
        //更新ui
    }
}
```

```
val toolbar = find<Toolbar>(R.id.toolbar)
```

```
toast("a toast!!")
```

```
alert("是否要退出登录", "提示") {
    yesButton { toast("正在退出") }
    noButton { }
}.show()
```

Anko

Anko Layout的作用是帮助我们使用DSL的方式去代替XML布局UI

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    verticalLayout {
        padding = dip(16)
        gravity = Gravity.CENTER
        val text = textView("Hello world")
        button("按钮") {
            onClick {
                text.text = "Hello Anko"
            }
        }
    }
}
```

更多特性

更多特性

When表达式

```
val name = 1
when (name) {
    1 -> print("tracy")
    24 -> print("kobe")
    23 -> print("lbj")
    else -> print("NBA")
}

val x = TextView(this)
val a = when (x) {
    is TextView -> x.text = "is TextView"
    is ViewGroup -> print("is ViewGroup")
    else -> "text"
}

val num = 10
val b = when (num) {
    in 0..10 -> print("0..10")
    in 10..100 -> print("10..100")
    else -> "text"
}
```

if表达式

```
val No = 1
if (No == 1) {
    print("T-Mac")
} else if (No == 24) {
    print("Kobe")
} else if (No == 23) {
    print("LBJ")
}

val name =
        if (No == 1)
            "T-Mac"
        else "unknown"
```

更多特性

字符串模板

有了字符串模板，我们可以完全的告别+

```
val num = 1
textView.text = "数字$num"
```

```
val list = listOf<String>("aa", "bb")
textView.text = "数字${list[0]}"
```

for循环

```
val list = listOf<String>("aa", "bb")
for (str in list) {
    print(str)
}
```

```
for (i in 0 until list.size) {
    print(list[i])
}
```

```
val map = mapOf("1" to "tracy",
        "24" to "kobe",
        "23" to "lbj")

for ((key, value) in map) {
    print("$key to $value")
}
```

更多特性

默认参数

在过去，使用Java时，我们希望一个方法接收不同的参数完成不同的事，通常会通过重载来实现，例如

```
public static final String TOKEN = "token";

public void getNews(String url, String token) {
    request(url, token);
}

public void getNews(String url) {
    request(url, TOKEN);
}
```

尤其是在自己封装库的时候，会出现非常多的重载方法，而Kotlin可以通过默认参数来有效的减少重载的出现

```
fun getNews(url: String, token: String = TOKEN) {
    request(url, token)
}
```

单利

在过去，使用Java时，我们实现一个单利，说难不难，但代码总是要写不少的，而且还要为各种性能问题去做处理。

```java
public class Single {
    public static Single mInstance = new Single();

    private Single() {

    }

    public static Single getInstance() {
        return mInstance;
    }
}
```

而在Kotlin中，一个单利只需要通过Object关键字去申明就能得到

```kotlin
object Single {

}
```

更多特性

infix关键字

infix关键字很有意思，被它申明的函数，被调用时传参数可以免去括号

```
class test {
    infix fun say(msg: String) {
        print(msg)
    }
}
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_second)
    val test = test()
    test say "Hello Kotlin"
}
```

extensions

extensions是kotlin的一个插件支持，需要在gradle中额外配置，它可以帮助我们从此告别findViewById，

如果你有一个控件叫mButton，那么你只要在IDE中输入mButton，并导入正确指向它所在的布局文件的那个包，你就可以自如的使用这个控件，无论你是在Activity中、fragment中还是ViewHolder中，你都可以得到extensions的支持


JET
BRAINS


UBER


沪江
学习，成为更好的自己


豆

Q&A