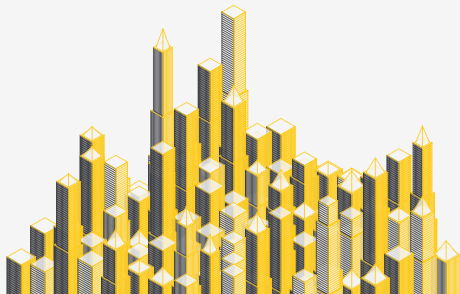




Kotlin 入坑指北

叶至灵

2020 年 12 月 1 日





1. 为什么选择 Kotlin

2. 和 NPE 说再见

3. 再短一点

4. Fun with Android

5. 参考

为什么选择 Kotlin



- **2011** JetBrains 发布, 面向 JVM
- **2016** v1.0 发布
- **2017** Google 宣布在  Android 提供 Kotlin 的最佳支持
- **2019** 已作为  Android 开发的推荐语言





让开发人员更快乐的一门现代编程语言¹。

✂ 简洁

数据类

函数式编程

单例

🔗 互操作性

Java

Android

🏰 安全

空安全

自动类型转换

🔧 工具友好

Java IDE

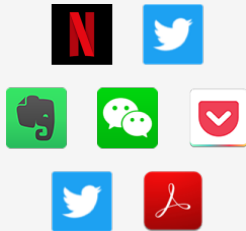
命令行

Jupyter

¹ 出自 Kotlin 语言中文站 [🔗](#)



- 越来越多应用通过 **Kotlin** 开发
- 用更**短**的代码写出更**安全**的应用
- 学习现代编程语言的新特性
类型推断、函数式编程、协程.....





val 表示变量只读，**var** 表示变量可重新赋值

```
val a: Int = 1 // 立即赋值
val b = 2      // 自动推断出 'Int' 类型
var x = 5      // 自动推断出 'Int' 类型
x += 1
```

用 **:** 表示类型，可自动推断时可省略

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
// 用表达式作为函数体
fun sum(a: Int, b: Int) = a + b
```

和 NPE 说再见



Java 最常见的运行时异常

```
String sample = null;  
System.out.println(sample.toString()); // NullPointerException!
```

用类型系统消除代码中的 NPE

```
var a: String = "abc"  
a = null // 编译错误  
var b: String? = "abd" // ? 表示变量可空  
val l = b.length // 编译错误，因为 b 可空  
val l = b?.length // 安全调用，l 的类型为可空整型 Int?  
// let 作用域函数，非空后执行  
l?.let { ..... // 只有非空才会执行 }
```



?: 用来缩写 `if not null and else`

```
// 返回值
val files = File("Test").listFiles()
println(files?.size ?: "empty")
// 执行语句
val values = .....
val email = values["email"] ?: throw IllegalStateException("Email is
    missing!")
```

i ?: 如果侧过头，可以看到猫王



```
// !! 非空断言操作符，强制转换，若空会抛出 NPE
val l = b!!.length
// 用 as? 安全转换
val aInt: Int? = a as? Int
// 智能转换可空为非空
val p = Person(first = "Zhiling", middle = null, last = "Ye")
if (p.middle != null) {
    val middleNameLength = p.middle.length // 这里变量的类型为 Int
}
// 集合过滤非空元素
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

❗ 如非必要，请勿使用!!操作符

再短一点



构造一个类来传输数据

```
// 好多行
public class Customer {
    private String mName;
    private String mEmail;
    public Customer(String name, String email) {
        this.mName = name;
        this.mEmail = email;
    }
    public String getName() { return mName; }
    public void setName(String name) { this.mName = name }
    public String getEmail() { return mEmail; }
    public void setEmail(String email) { this.mEmail = email }
}
```



`data class` 来实现 Data transfer object

// 一行

```
data class Customer(val name: String, val email: String)
```



■ 饿汉式，非 lazy loading，线程安全

```
public final class MySingleton {  
    private static final int myProperty = 3;  
    private static final MySingleton INSTANCE = new MySingleton();  
    private MySingleton() {}  
    public final int getMyProperty() {  
        return myProperty;  
    }  
    public final void myFunction() {  
        return "Hello";  
    }  
}
```



`object` 来实现 Data transfer object

```
object MySingleton {  
    val myProperty = 3  
    fun myFunction() = "Hello"  
}
```




快速创建 Array, List, Set, Map

```
val strings = arrayOf("a", "b", "c")  
// 不可变集合  
var numList = listOf(1, 2, 3)  
// 可变集合  
var numMap = mutableMapOf(1 to "one", 2 to "two", 3 to  
    "three")
```

i to 为生成 Pair 对象的中缀运算符



下面的代码会输出什么？

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
    .filter { it.startsWith("a") }
    .sortedBy { it }
    .map { it.toUpperCase() }
    .forEach { println(it) }
```

❗ 像使用  Pandas 一样，同样有 flatMap, groupBy, reduce 等操作



扩展函数可以增加已有类的功能

```
fun String.reply(): String =  
    when (this) {  
        "Hello"    -> "World"  
        "Bye"      -> "Bye"  
        else       -> "Unknown"  
    }  
// 使用  
"Hello".reply() // "World"
```



■ 用一种简洁明了的方式在对象的上下文中执行代码块

```
// 用 apply 来初始化
val adam = Person("Adam").apply {
    it.age = 20 // it 可以省略
    it.city = "London"
}
// 初始化并返回计算结果
val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}
```

❶ \$port 字符串内插，甚至可以使用表达式



不同的是这个对象在块中如何使用，以及整个表达式的结果是什么

函数	对象引用	返回值	是否是扩展函数
let	it	Lambda 表达式结果	是
run	this	Lambda 表达式结果	是
run	-	Lambda 表达式结果	不是：调用无需上下文对象
with	this	Lambda 表达式结果	不是：把上下文对象当做参数
apply	this	上下文对象	是
also	it	上下文对象	是

Fun with Android



```
// SAM 转化
loginButton.setOnClickListener { ... }
// 伴生对象
class LoginFragment : Fragment() {
    companion object {
        private const val TAG = "LoginFragment"
    }
}
// 属性委托
private val viewModel: LoginViewModel by viewModels()
```

❶ 单一抽象方法转换：用一个匿名函数来实现单一抽象方法



轻量、内存泄漏更少、内置取消支持、Jetpack 集成

```
class LoginViewModel(  
    private val loginRepository: LoginRepository  
) : ViewModel() {  
    fun login(username: String, token: String) {  
        // 创建一个新的协程  
        viewModelScope.launch(Dispatchers.IO) {  
            val jsonBody = "{ username: \"$username\", token: \"$token\" }"  
            loginRepository.makeLoginRequest(jsonBody)  
        }  
    }  
}
```

i viewModelScope 是 Android 架构组件提供的生命周期感知型协程范围




参考



- [1] JetBrains.
Kotlin 中文文档
链接: 
- [2] Ken Kousen
Kotlin Cookbook: A Problem-Focused Approach
购买: 
- [3] Google.
Android Developers
链接: 

 实际查阅的为乔禹昂的译作《Kotlin 编程实践》



	主题	penrose
	正文字体	更纱黑体 + Roboto
	等宽字体	JetBrains Mono

Creative Commons Attribution-ShareAlike 4.0 International





logoyellow

#FFC805

logoblack

#323337

penroseblue

#0496FF

penrosered

#DB4C40

penrosegreen

#709255

penroseorange

#FC814A

penrosegray

#BBBAC6

penrosepink

#FBACBE

Happy \Coding

