快速上手 Kotlin

讲给你的 Kotlin Primer

- Java/Android 开发者
- 想要了解和使用 Kotlin

- 快速学会 Kotlin 的使用
- 深入理解 Kotlin
- 解决旧项目迁移问题

目录：快速上手

- Kotlin 基础语法
- 与 Java 互操作
- 新手易碰上的问题

基础语法

```
var age: Int = 18
val name: String = "Zhang Tao"

var age = 18
val name = "Zhang Tao"
```

基础语法

String?

String

string!!

基础语法

```
var name: String = "Zhang Tao"
var name2: String? = "Zhang Tao"
name = name2 ✕
name = name2!!
name2 = name
```

基础语法

```
var name: String? = null

println(name?.length)


fun printLength(name: String){
    println(name.length)
}

printLength(null)
```

函数基础语法

```
fun echo(name: String): String? {
    println("$name")
    return name
}
```

与 Java 代码互调

- 语法变化
- Kotlin关键字处理
- 基本数据类型的处理

Java 与 Kotlin 交互的语法变化

```kotlin
// Utils.kt
fun echo(name: String) {
    println("$name")
}
```

```java
// Main.java
public static void main(String[] args) {
    UtilsKt.echo("hello");
}
```

Java 与 Kotlin 交互的语法变化

```kotlin
object Test {
    fun sayMessage(msg: String){
        println(msg)
    }
}
```

kotlin code

```kotlin
Test.sayMessage("hello")
```

java code

```java
Test.INSTANCE.sayMessage("hello");
```

Java 与 Kotlin 交互的语法变化

java code `TestMain.class`

kotlin code `TestMain::class.java`

新手易踩坑

- Kotlin 没有封装类
- Kotlin 类型空值敏感
- Kotlin 没有静态变量与静态方法

Kotlin 没有封装类

java code

```java
public interface AInterface {
    void putNumber(int num);

    void putNumber(Integer num);
}
```

kotlin code

```kotlin
a.putNumber(100)
```

Kotlin 类型空值敏感

kotlin code

```kotlin
fun sayMessage(msg: String) {
    println(msg)
}
```

java code

```java
a.sayMessage(null)
```

Kotlin 类型空值敏感

java code

```java
String format(String str) {
    return str.isEmpty() ? null : str;
}
```

kotlin code

```kotlin
fun function(str: String) {
    val fmt1 = format(str)
    val fmt2:String = format(str)
    val fmt3:String? = format(str)
}
```

Kotlin 没有静态变量与静态方法

```kotlin
object Test {

    fun sayMessage(msg: String){
        println(msg)
    }
}
```

java code

```kotlin
Test.INSTANCE.sayMessage("hello")
```

kotlin code

```kotlin
Test.sayMessage("hello")
```

Kotlin 没有静态变量与静态方法

```kotlin
object Test {
    @JvmStatic
    fun sayMessage(msg: String){
        println(msg)
    }
}
```

java code

```kotlin
Test.sayMessage("hello")
```

kotlin code

```kotlin
Test.sayMessage("hello")
```

练习1：选择题

```
fun main(args: Array<String>) {
    val age = 18
    val name = "Zhang Tao"

    println("我叫%d，我今年%d岁", name, age)
}
```

A：正常运行，输出> 我叫Zhang Tao，我今年18岁

B：正常运行，输出> 我叫%d，我今年%d岁

C：运行时出错，字符串格式化异常

D：编译时出错

练习2：以下代码有什么问题

kotlin code

```kotlin
object Utils {
    @JvmStatic
    fun sayMessage(msg: String?) {
        println("$msg")
    }
}
```

java code

```java
Utils.INSTANCE.sayHello(null)
```

函数 与 Lambda 闭包

目录：函数 与 Lambda 闭包

Kotlin 函数的语法

```
fun echo(name: String) {
    println("$name")
}

fun echo(name: String = "ZhangTao") {
    println("$name")
}


fun echo(name: String) = println("$name")
```

函数嵌套

```
fun function() {
    val str = "hello world"

    fun say(count: Int = 10) {
        println(str)
        if (count > 0) {
            say(count - 1)
        }
    }
    say()
}
```

用途:

在某些条件下触发递归的函数，或不希望被外部函数访问到的函数

扩展函数

```
fun File.readText(charset: Charset = Charsets.UTF_8):
String = readBytes().toString(charset)

val file = File()
val content = file.readText()
```

java code

```
String content = FilesKt.readText(file, Charsets.UTF_8);
```

扩展函数的静态解析

```
open class Animal
class Dog : Animal()

fun Animal.name() = "animal"
fun Dog.name() = "dog"

fun Animal.printName(anim: Animal) {
    println(anim.name())
}
fun main(args: Array<String>) {
    Dog().printName(Dog())
}
```

扩展函数的静态解析

```
public static final String name(Animal receiver){
    return "animal";
}
public static final String name(Dog receiver){
    return "dog";
}

public static final void printName(Animal r, Animal a){
    String str = name(a);
    System.out.println(str);
}

public static final void main(String[] args){
    printName((Animal)new Dog(), (Animal)new Dog());
}
```

Lambda 闭包

```
public static void main(String[] args) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            // …
        }
    });

    thread.start();
}
```

Java8 Lambda 语法

```
public static void main(String[] args) {
    Thread thread = new Thread(() -> {

    });
    thread.start();
}
```

Kotlin Lambda 语法

```
fun main(args: Array<String>) {
    val thread = Thread({ -> Unit   })

    thread.start()
}
```

Kotlin Lambda 语法

```
fun main(args: Array<String>) {
    val thread = Thread({ })

    thread.start()
}
```

如果Lambda没有参数，可以省略箭头符号 —>

Kotlin Lambda 语法

```
fun main(args: Array<String>) {
    val thread = Thread(){ }

    thread.start()
}
```

如果Lambda是函数的最后一个参数，可以将大括号放在小括号外面

Kotlin Lambda 语法

```
fun main(args: Array<String>) {
    val thread = Thread{ }

    thread.start()
}
```

如果函数只有一个参数且这个参数是Lambda，则可以省略小括号

Lambda 闭包声明

```
val echo = { name: String ->
    println(name)
}

fun main(args: Array<String>) {
    echo.invoke("Zhang Tao”)
    echo("Zhang Tao")
}
```

Lambda

```kotlin
val echo = { name: String ->
    println(name)
}
```

```kotlin
Function1<String, Unit> echo =
                (Function1) echo.INSTANCE;
```

注：Kotlin 的 Lambda 参数是有上限的，最多22个

高阶：函数(Lambda)的参数是函数(Lambda)

```
fun onlyif(isDebug: Boolean, block: () -> Unit) {
    if (isDebug) block()
}

fun main(args: Array<String>): Unit {
    onlyif(true) {
        println("打印日志")
    }
}
```

高阶：函数(Lambda)的参数是函数(Lambda)

```
fun onlyif(isDebug: Boolean, block: () -> Unit) {
    if (isDebug) block()
}

fun main(args: Array<String>) {
    onlyif(true) {
        println("打印日志")
    }
}
```

重点：函数是“一等公民”

```
val runnable = Runnable {
    println("Runnable::run")
}

val function: () -> Unit

function = runnable::run
```

用内联优化代码

- Kotlin 的 Lambda 是一个匿名对象
- 可以使用 inline 修饰方法，这样当方法在编译时就会拆解方法的调用为语句调用，进而减少创建不必要的对象

高阶：函数(Lambda)的参数是函数(Lambda)

```
public static final void onlyif(boolean isDebug,
                                Function0<Unit> block) {
    if (isDebug) {
        block.invoke();
    }
}

public static final void main(String[] args) {
    boolean isDebug = true;
    if (isDebug){
        String str = "打印日志";
        System.out.println(str);
    }
}
```

练习1：选择题

```
val lambdaA = { a: Int, b: Int, c: Int, d: Int, e: Int, f: Int, g: Int, h: Int,
                i: Int, j: Int, k: Int, l: Int, m: Int, n: Int, o: Int, p: Int,
                q: Int, r: Int, s: Int, t: Int, u: Int, v: Int, w: Int ->
    println("Zhang Tao")
}

fun main(args: Array<String>) {
    lambdaA(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
}
```

A： 正常运行，输出> Zhang Tao

B： 编译正常，运行时出错

C： 编译时出错

练习2：如何让练习1的代码正确运行

```
val lambdaA = { a: Int, b: Int, c: Int, d: Int, e: Int, f: Int, g: Int, h: Int,
                i: Int, j: Int, k: Int, l: Int, m: Int, n: Int, o: Int, p: Int,
                q: Int, r: Int, s: Int, t: Int, u: Int, v: Int, w: Int ->
    println("Zhang Tao")
}

fun main(args: Array<String>) {
    lambdaA(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
}
```

类与对象

目录：类与对象

- 构造函数
- 访问修饰符
- 伴生对象
- 单例类
- 动态代理
- Kotlin 特有的类

Kotlin 类

```
class MainActivity : AppCompatActivity()

class MainActivity : AppCompatActivity(),OnClickListener
```

Kotlin 的类默认是 public final 的

```
open class MainActivity : AppCompatActivity()

open class MainActivity :
                    AppCompatActivity(),OnClickListener
```

Kotlin 构造方法

```
class TestView : View {
    constructor(context: Context) : super(context) {
        println("constructor")
    }

    constructor(context: Context, attrs: AttributeSet?) :
                                                this(context, attrs, 0)

    constructor(context: Context, attrs: AttributeSet?,
                defStyleAttr: Int) : super(context, attrs, defStyleAttr)
}
```

访问修饰符

```
private
protected
public
internal
```

```
public class StringUtils {
    public static boolean isEmpty(String str) {
        return "".equals(str);
    }
}
```

伴生对象

```
class StringUtils {
    companion object {
        fun isEmpty(str: String): Boolean {
            return "" == str
        }
    }
}
```

单例

```
class Single private constructor() {
    companion object {
        fun get(): Single {
            return Holder.instance
        }
    }

    private object Holder {
        val instance = Single()
    }
}
```

类的动态代理

```
interface Animal {
    fun bark()
}

class Dog : Animal {
    override fun bark() {
        println("Wang")
    }
}

class Zoo(animal: Animal) : Animal by animal

fun main(args: Array<String>) {
    Zoo(Dog()).bark()
}
```

数据类

```
data class User(var id: Int, var name: String)

public final     getter()/setter()

toString()
hashCode()
equals()
copy()
```

数据类

```
data class User(var id: Int, var name: String)

class VIP(id: Int, name: String) : User(id, name)
```

枚举类

```
enum class Command {
    A, B, C, D
}

fun exec(command: Command) = when (command) {
    Command.A -> {
    }
    Command.B -> {
    }
    Command.C -> {
    }
    Command.D -> {
    }
}
```

密闭类

```
sealed class SuperCommand {
    object A : SuperCommand()
    object B : SuperCommand()
    object C : SuperCommand()
    object D : SuperCommand()
}

fun exec(superCommand: SuperCommand) = when (superCommand) {
    SuperCommand.A -> {
    }
    SuperCommand.B -> {
    }
    SuperCommand.C -> {
    }
    SuperCommand.D -> {
    }
}
```

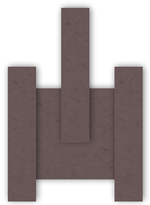
超级枚举：密闭类

```
sealed class SuperCommand {
    object A : SuperCommand()
    object B : SuperCommand()
    object C : SuperCommand()
    object D : SuperCommand()
    class E : SuperCommand()
}

fun exec(superCommand: SuperCommand) = when (superCommand) {
    SuperCommand.A -> {
    }
    //…
    is SuperCommand.E -> {
    }
}
```

超级枚举：密闭类

```
sealed class SuperCommand {
    object A : SuperCommand()
    object B : SuperCommand()
    object C : SuperCommand()
    object D : SuperCommand()
    class E(var id: Int) : SuperCommand()
}

fun exec(superCommand: SuperCommand) = when (superCommand) {
    SuperCommand.A -> {
    }
    //…
    is SuperCommand.E -> {
    }
}
```

超级枚举：密闭类



```
sealed class SuperCommand {
    object UP : SuperCommand()
    object DOWN : SuperCommand()
    object LEFT : SuperCommand()
    object RIGHT : SuperCommand()
    class PACE(var pace: Int) : SuperCommand()
}

fun exec(tank: Tank, superCommand: SuperCommand)
                                = when (superCommand) {
    SuperCommand.UP -> {
    }
    //…
    is SuperCommand.PACE -> {
    }
}
```

超级枚举：密闭类

```
sealed class SuperCommand {
    object UP : SuperCommand()
    object DOWN : SuperCommand()
    object LEFT : SuperCommand()
    object RIGHT : SuperCommand()
    class PACE(var pace: Int) : SuperCommand()
}

fun exec(view: View, superCommand: SuperCommand)
                                = when (superCommand) {
    SuperCommand.UP -> {
    }
    //…
    is SuperCommand.PACE -> {
    }
}
```

代码练习

音乐播放器，有系统内置的两种颜色的皮肤，
每个用户都可以选择自己的播放器皮肤颜色，
当不同的用户登录以后，显示不同的播放器皮肤
同时需要注意皮肤颜色的可扩展性