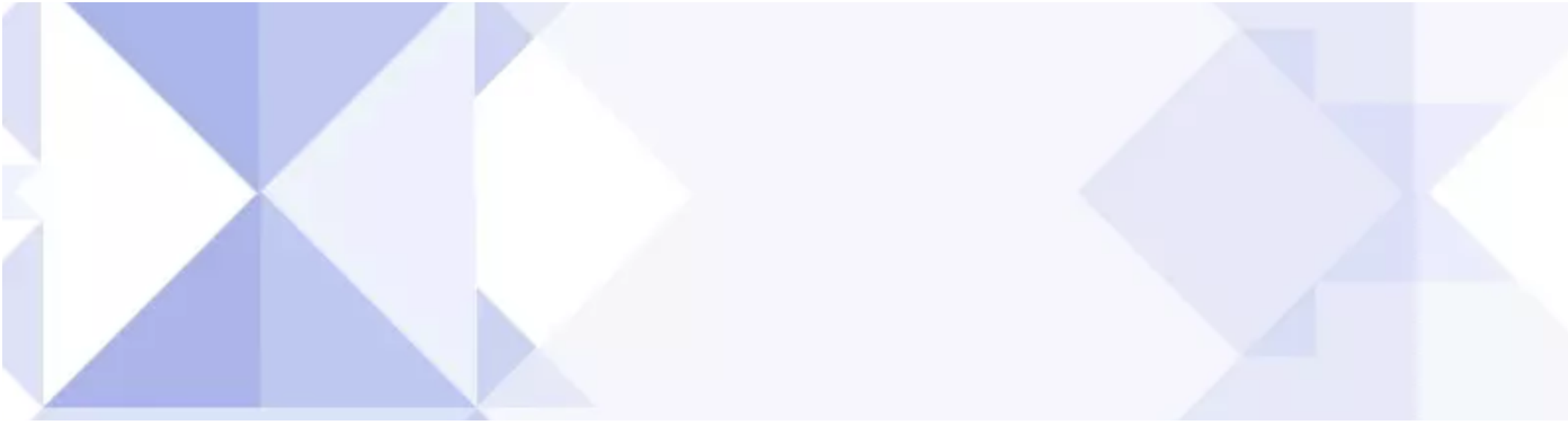
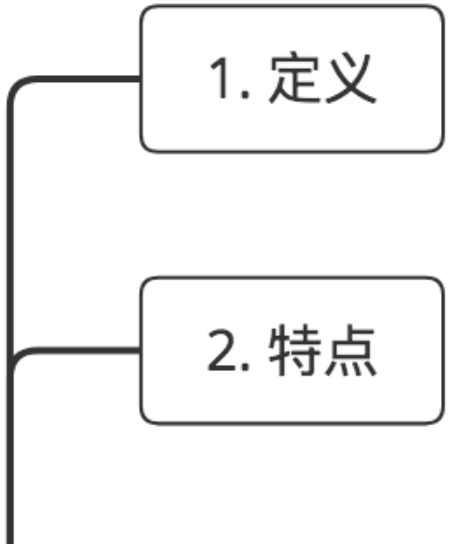
Android：这是一份全面 & 详细的Kotlin入门学习指南

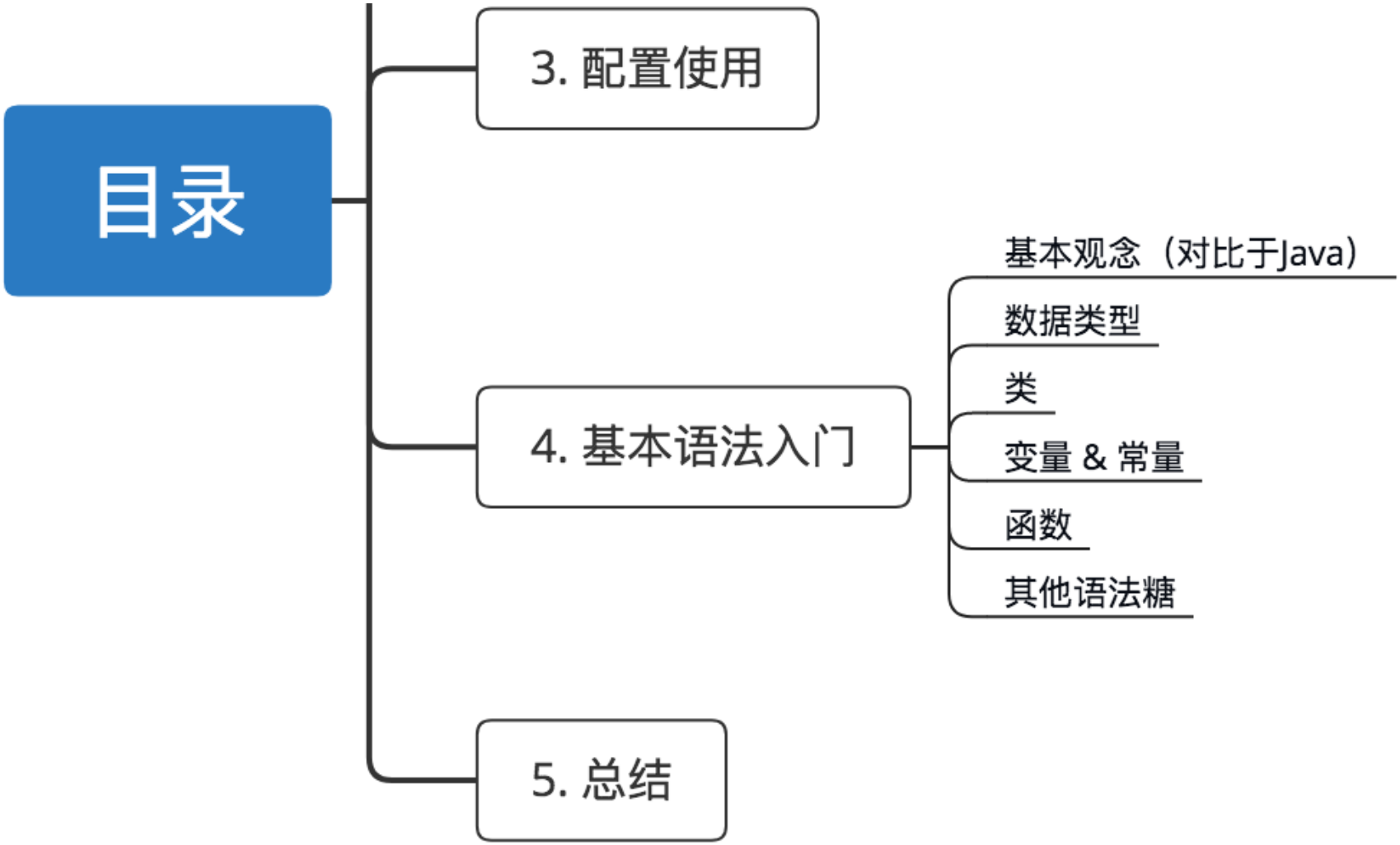
前言

- Kotlin被Google官方认为是Android开发的一级编程语言
- 今天，我将献上一份 《**全面 & 详细的Kotlin入门学习指南**》，包括定义特点、配置使用、入门语法等，希望你们会喜欢。



目录

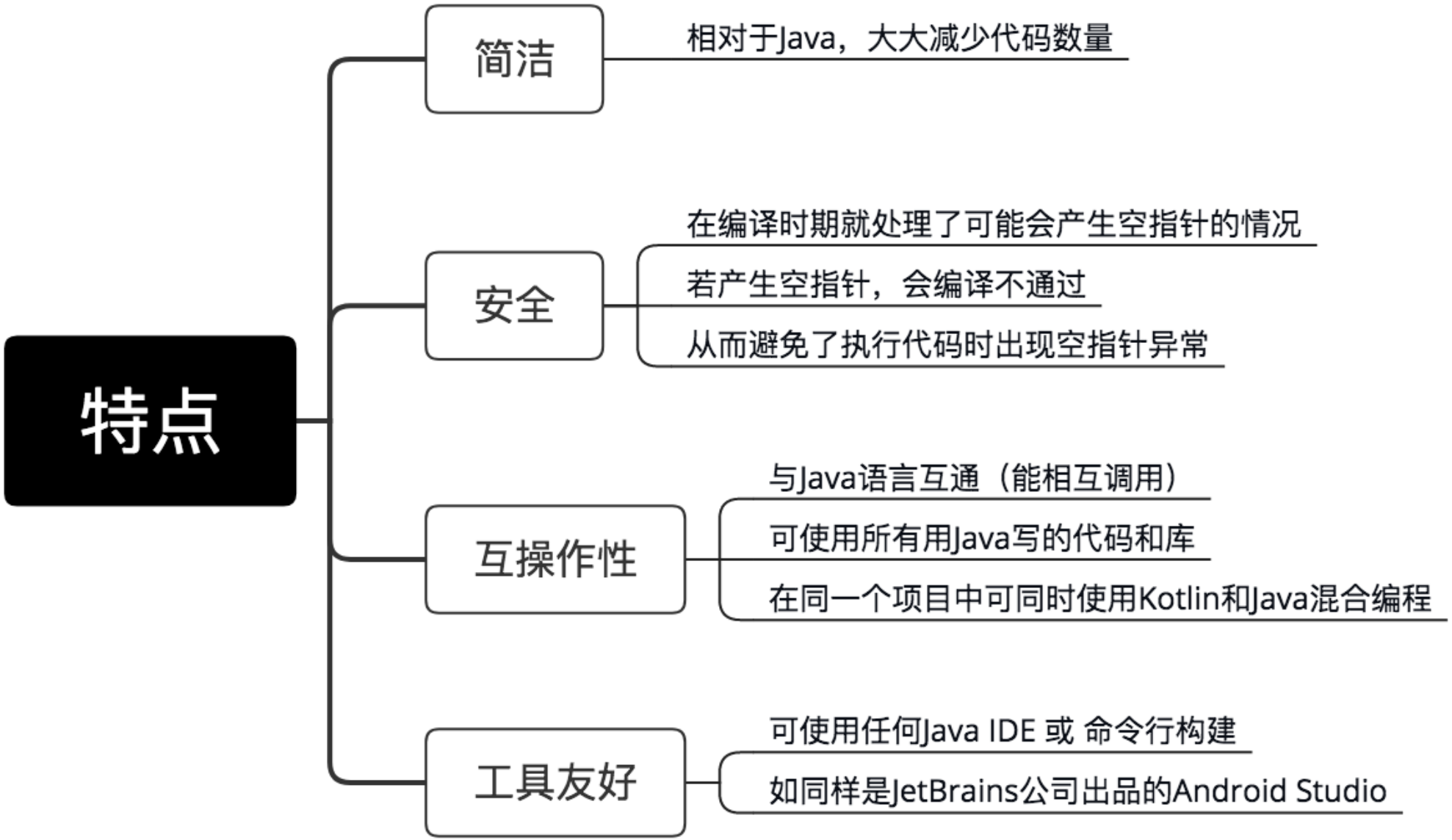




1. 定义

- **Android开发的一级编程语言**（Google官方认证）
- 由JetBrains公司在2010年推出 & 开源，与Java语言互通 & 具备多种Java尚不支持的新特性
- Android Studio3.0后的版本支持Kotlin

2. 特点


特点
简洁
相对于Java，大大减少代码数量
安全
在编译时期就处理了可能会产生空指针的情况
若产生空指针，会编译不通过
从而避免了执行代码时出现空指针异常
互操作性
与Java语言互通（能相互调用）
可使用所有用Java写的代码和库
在同一个项目中可同时使用Kotlin和Java混合编程
工具友好
可使用任何Java IDE 或 命令行构建
如同样是JetBrains公司出品的Android Studio

3. 配置使用

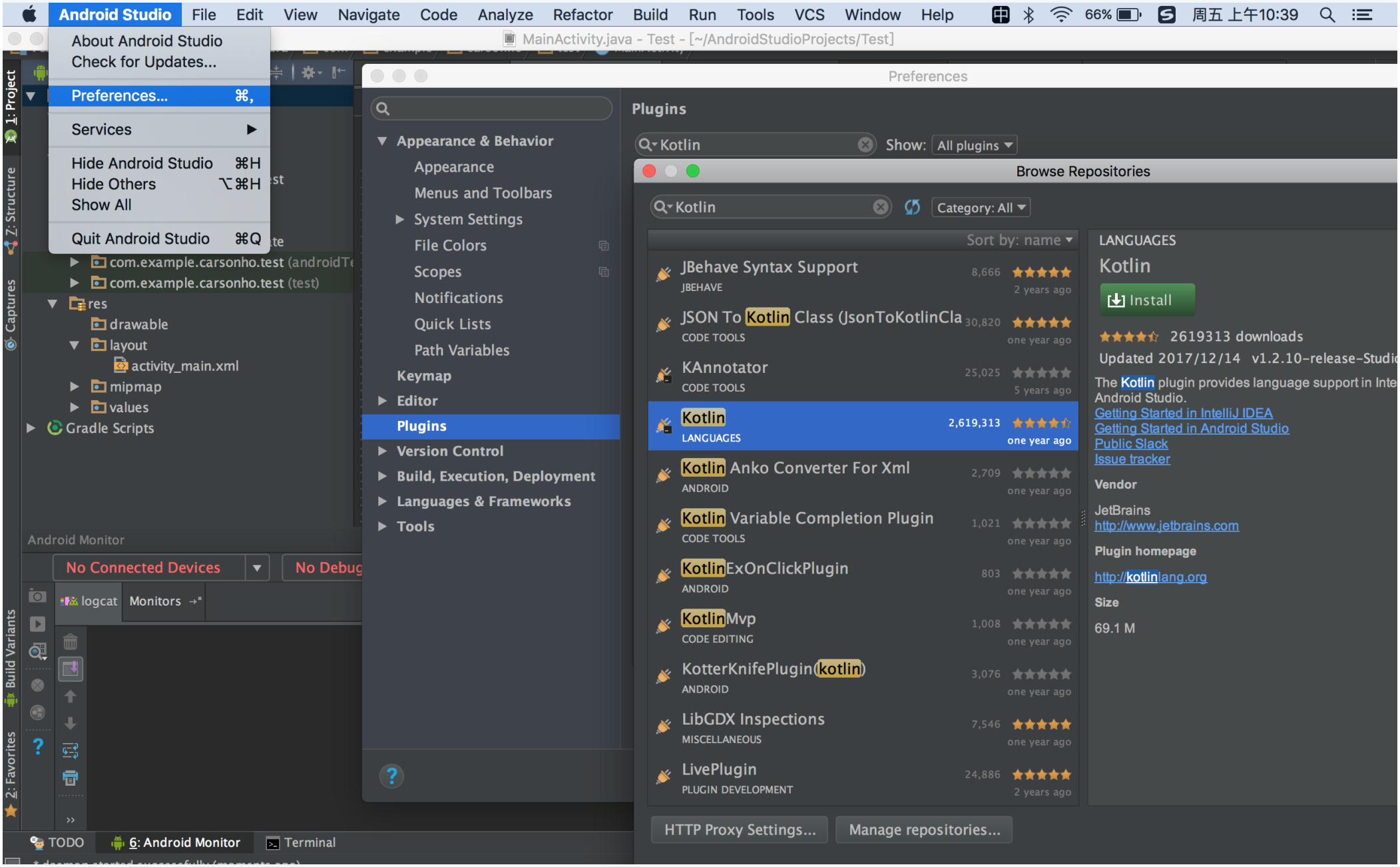
下面将讲解如何在Android Studio配置Kotlin进行使用。

3.1 Android Studio3.0前的版本

主要分为3个步骤，完成3个步骤即可完成Kotlin的配置。

步骤1：安装Kotlin插件

点击Android Studio Preference -> Plugins -> 搜索Kotlin Languages插件



步骤2：在根目录的build.gradle中加入

```
buildscript {
    ext.kotlin_version = '1.2.10'

    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

步骤3：在app/build.gradle中引入

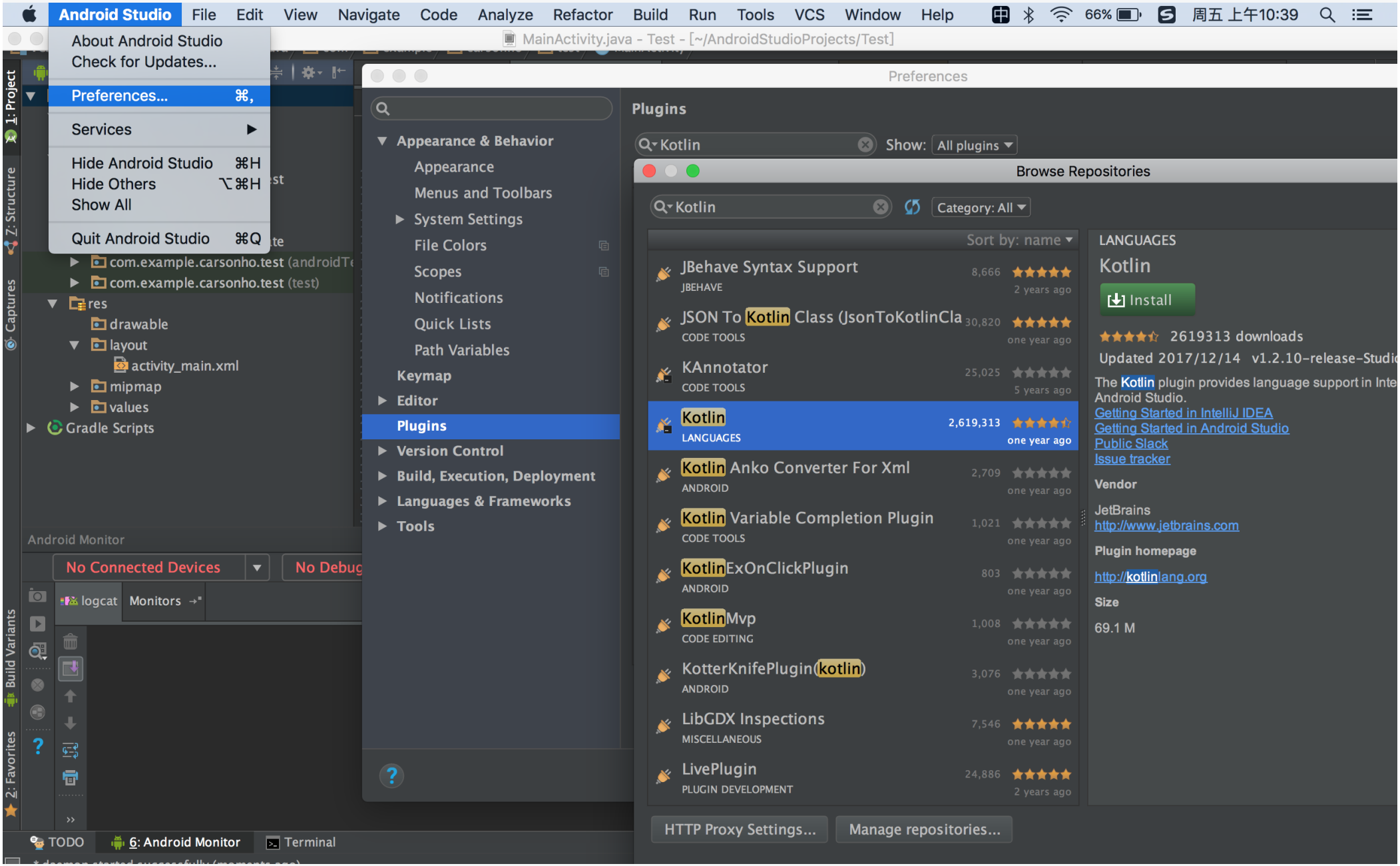
```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'

buildscript {
    ext.kotlin_version = '1.2.10'

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

3.2 Android Studio3.0前后的版本

Android Studio3.0后的版本已经完美支持Kotlin，只需安装Kotlin插件即可，即：点击Android Studio Preference -> Plugins -> 搜索Kotlin Languages插件



4. 基本语法入门

本节中，会详细讲解Kotlin的基本语法，主要包括：

- 基本观念（对比于Java）
- 数据类型
- 类
- 变量 & 常量
- 函数
- 其他语法糖（控制流、类型检查 & 转换、安全性等）

4.1 基本观念

在Kotlin中，有一些观念是和Java存在较大区别的，一些基本观念需要注意的：

- 操作对象：在Kotlin中，所有变量的成员方法和属性都是对象，若无返回值则返回Unit对象，大多数情况下Uint可以省略；Kotlin 中没有 new 关键字
- 数据类型 & 转换：在Java中通过装箱和拆箱在基本数据类型和包装类型之间相互转换；在Kotlin中，而不管是常量还是变量在声明是都必须具有类型注释或者初始化，如果在声明 & 进行初始化时，会自行推导其数据类型。
- 编译的角度：和Java一样，Kotlin同样基于JVM。区别在于：后者是静态类型语言，意味着所有变量和表达式类型在编译时已确定。
- 撰写：在Kotlin中，一句代码结束后不用添加分号 “；”；而在Java中，使用分号“;”标志一句代码结束。

4.2 数据类型

主要包括：

- 数值（Numbers）
- 字符（Characters）
- 字符串（Strings）
- 布尔（Boolean）

- 数组（Arrays）

a. 数值类型（Numbers）

Kotlin的基本数值类型有六种：Byte、Short、Int、Long、Float、Double

类型	位宽度（Bit）
Byte	8
Short	16
Int	32
Long	64
Float	32
Double	64

> 注：区别于Java，在Kotlin中字符（char）不属于数值类型，是一个独立的数据类型。

- 补充说明：每种数据类型使用对应方法，可将其他类型转换成其他数据类型

```
toByte(): Byte
toShort(): Short
toInt(): Int
toLong(): Long
toFloat():  Float
toDouble(): Double
toChar(): Char
```

b. 字符类型（Characters）

Kotlin中的字符类型采用 Char 表示，必须使用单引号' '包含起来使用 & 不能直接和数字操作

```
val ch :Char = 1; // 错误示范
val ch :Char = '1'; // 正确示范

// 将字符类型转换成数字
val ch :Char = '8';
val a :Int = ch.toInt()
```

c. 字符串类型（Strings）

- 表示方式：String
- 特点：不可变
- 使用：通过索引访问的字符串中的字符：s [i]

```
// 使用1：一个字符串可以用一个for循环迭代输出
for (c in str) {
    println(c)
}

// 使用2：可使用三个引号 """拼接多行字符串
fun main(args: Array<String>) {
    val text = """
    字符串1
    字符串2
```

```
    """
    println(text)   // 输出存在一些前置空格
}

// 注：可通过 trimMargin()删除多余空白
fun strSample() {
    val text = """
    | str1
    |str2
    |多行字符串
    |bbbbbb
    """.trimMargin()
    println(text)    // 删除了前置空格
}
```

补充说明：字符串模版（String Templates）

- 即在字符串内通过一些小段代码求值并把结果合并到字符串中。
- 模板表达式以美元符（$）开头

```
// $：表示一个变量名 / 变量值
// 示例
val i = 10
val s = "i = $i" // 表示 "i = 10"

// ${varName.fun()}：表示变量的方法返回值
// 示例
val s = "abc"
val str = "$s.length is ${s.length}" //识别为 "abc.length is 3"
```

d. 布尔类型（Boolean）

- Kotlin的Boolean类似于Java的boolean类型，其值只有true 、false
- Boolean内置的函数逻辑运算包括：

```
|| - 短路逻辑或
&& - 短路逻辑与
! - 逻辑非
```

e. 数组类型（Arrays）

- 实现方式：使用Array类
- 使用方法：size 属性、get方法和set 方法。注：使用 [] 重载了 get 和 set 方法，可通过下标获取 / 设置数组值。
- 创建方式：方式1 = 函数arrayOf()；方式2 = 工厂函数

```
// 方式1：使用arrayOf创建1个数组：[1,2,3]
val a = arrayOf(1, 2, 3)

// 方式2：使用工厂函数创建1个数组[0,2,4]
val b = Array(3, { i -> (i * 2) })
// 工厂函数源码分析
// 参数1 = 数组长度，花括号内是一个初始化值的代码块，给出数组下标 & 初始化值
public inline constructor(size: Int, init: (Int) -> T)

// 读取数组内容
println(a[0])    // 输出结果：1
println(b[1])    // 输出结果：2

// 特别注意：除了类Array，还有ByteArray, ShortArray, IntArray用来表示各个类型的数组
// 优点：省去了装箱操作，因此效率更高
// 具体使用：同Array
val x: IntArray = intArrayOf(1, 2, 3)
```

> 注： 区别于Java，Kotlin中的数组是不型变的（invariant），即Kotlin 不允许将Array赋值给Array，以防止可能的运行时失败

4.3 类使用

a. 类的声明 & 实例化

```
// 格式
class 类名（参数名1：参数类型，参数名2：参数类型...）{}

// 示例
class User(userName: String, age: Int){}

// Kotlin支持默认参数，即在调用函数时可不指定参数，则使用默认函数
class User(userName: String = "hjc", age: Int = 26){
}
// 在实例化类时不传入参数，userName默认 = hjc，age默认 = 26
var user = User()
// 在设置默认值后，若不想用默认值可在创建实例时传入参数
var user = User("ABC" , 123)
// 命名参数：若一个默认参数在一个无默认值的参数前，那么该默认值只能通过使用命名参数调用该函数来使用
class User(userName: String = "hjc", age: Int)
var  user = User(age = 26)

// Kotlin没有new关键字，所以直接创建类的实例：
User()
```

对于构造函数，Kotlin中类可有一个主构造函数 & 多个次构造函数，下面将详细说明。

b. 主构造函数

- 属于类头的一部分 = 跟在类名后，采用 constructor 关键字
- 不能包含任何的代码。初始化的代码放到以 init 关键字作为前缀的代码块中

```
// 形式
class 类名 constructor（参数名：参数类型）{
    init {
    //...
    }
}

// 示例
class User constructor(userName: String) {
```

```
    init {
    //...
    }
}
```

注：若主构造函数无任何注解 / 可见性修饰符，可省略 constructor 关键字

```
// 形式
class 类名（参数名：参数类型）{
    init {
    //...
    }
}

// 示例
class User (userName: String) {
    init {
    //...
    }
}
```

c. 次构造函数

- 必须加constructor关键字
- 一个类中可存在多个次构造函数，传入参数不同

```
// 形式
constructor(参数名：参数类型) :{函数体}

// 示例
class User(userName: String) {
    // 主构造函数
    init {
        println(userName)
    }
```

```
10
11      // 次构造函数1：可通过this调主构造函数
12      constructor() : this("hjc")
13
14      // 次构造函数2：可通过this调主构造函数
15      constructor(age: Int) : this("hjc") {
16          println(age)
17      }
18
19      // 次构造函数3：通过this调主构造函数
20      constructor(sex: String, age: Int) : this("hjc") {
21          println("$sex$age")
22      }
23  }
24
25  // 实例化类
26  User("hjc") // 调用主构造函数
27  User()      // 调用次构造函数1
28  User(2)     // 调用次构造函数2
29  User("male",26) // 调用次构造函数3
```

d. 类的属性

Kotlin的类可以拥有属性：关键字var（读写） / 关键字val（只读）

```
1  class User {
2      var userName: String
3      val sex: String = "男"
4  }
5
6  // 使用属性 = 名称 + 引用
7  User().sex  // 使用该属性 = Java的getter方法
8  User().userName = "hjc"  // 设置该属性 = Java的setter方法
```

e. 可见性修饰符

- private： 本类内部都可见
- protected： 本类内部 & 子类中可见
- public：能见到类声明的任何客户端都可以见（public成员）
- internal：能见到类声明的本模块内的任何客户端都可见（public成员）

区别于Java，Kotlin的可见修饰符少了default，多了internal：该成员只在相同模块内可见。（注：一个模块 = 编译在一起的一套 Kotlin 文件：
一个 IntelliJ IDEA 模块；
一个 Maven 项目；
一个 Gradle 源集；
一次 < kotlinc > Ant 任务执行所编译的一套文件。

f. 继承 & 重写

- 类似于Java，Kotlin是单继承 = 只有一个父类
- 区别：Kotlin使用冒号“ : ”继承 & 默认不允许继承（若想让类可被继承，需用open关键字来标识）

```
// 用open关键字标识该类允许被继承
open class Food

// 类Fruits继承类Food
class Fruits : Food()
```

- 对于子类重写父类的方法，在Kotlin中，方法也是默认不可重写的
- 若子类要重写父类中的方法，则需在父类的方法前面加open关键字，然后在子类重写的方法前加override关键字

```
// 父类
// 在类 & 方法前都加了关键字open，为了被继承 & 方法重写
open class Food {
    open fun banana() {}
}

// 子类
class Fruits : Food(){
```

```
    // 重写了父类的方法
    override fun banana() {
        super.banana()
    }
}
```

特殊类说明

下面将讲解一些特殊的类：

- 嵌套类（内部类）
- 接口
- 数据类
- 枚举类

```
/**
 * 1. 嵌套类（内部类）
 * 标识：关键字inner
 * 使用：通过外部类的实例调用嵌套类
 */
 class User {
    var age: Int = 0

    inner class UserName {
    }
}

var userName: User.UserName  = User().UserName()



/**
 * 2. 接口
 * 标识：关键字interface
 */
// 声明
```

```
interface A{}
interface B{}

// 方法体
// 接口中的方法可以有默认方法体，有默认方法体的方法可不重写
// 区别于Java：Java不支持接口里的方法有方法体。
interface UserImpl{
    fun getName(): String // 无默认方法体，必须重写
    fun getAge(): Int{    // 有默认方法体，可不重写
        return 22
    }
}
// 实现接口UserImpl：需重写getName() & 可不重写getAge()
class User :UserImpl{
    override fun getName(): String {
        return "hjc"
    }
}

// 实现接口：冒号:
class Food : A, B {} // Kotlin是多实现
class Fruits: Food,A, B {} // 继承 + 实现接口

 /**
  * 3. 数据类
  * 作用：保存数据
  * 标识：关键字data
 */
// 使用：创建类时会自动创建以下方法：
//      1. getter/setter方法；
//      2. equals() / hashCode() 对；
//      3. toString() ：输出"类名(参数+参数值)"；
//      4. copy() 函数：复制一个对象&改变它的一些属性，但其余部分保持不变

// 示例：
// 声明1个数据类
data class User(var userName: String, var age: Int)
```

```
// copy函数使用
var user = User("hjc",26)
var user1 = user.copy(age = 30)
// 输出user1.toString()，结果是：User(userName=hjc,age=30)

// 特别注意
// 1. 主构造方法至少要有一个参数，且参数必须标记为val或var
// 2. 数据类不能用open、abstract、sealed(封闭类)、inner标识

/**
 * 4. 枚举类
 * 标识：关键字enum
 */
 // 定义
 enum class Color {
    RED, GREEN, BLUE
}

// 为枚举类指定值
enum class Color(rgb: Int) {
    RED(0xFF0000), GREEN(0x00FF00), BLUE(0x0000FF)
}
```

4.4 变量 & 常量

```
// 变量
    // 模板： var 变量名：数据类型 = 具体赋值数值
    // 规则：
    //      1. 采用 “var” 标识
    //      2. 变量名跟在var后；数据类型在最后
    //      3. 变量名与数据类型采用冒号 ":" 隔开
    // 示例：
            var a: Int = 1
            var a: Int
            a = 2

```

```
// 常量
    // 模板： val 常量名：数据类型 = 具体赋值数值
    // 规则：
    //      1. 采用 “val” 标识
    //      2. 常量名跟在val后；数据类型在最后
    //      3. 常量名与数据类型采用冒号 ":" 隔开
    // 示例：
            val a: Int // 声明一个不初始化的变量，必须显式指定类型
            a = 2 // 常量值不能再次更改
            val b: Int = 1 // 声明并显示指定数值

// 特别注意：1. 自动类型转换 & 判断数据类型
    // 1. 自动类型转换
    // 在定义变量 / 常量时，若直接赋值，可不指定其数据类型，则能自动进行类型转换。如：
    var a = "aaa" // 此处a的数据类型是String类型
    val b = 1 // 此处的b的数据类型是Int类型

    // 2. 判断数据类型：运算符is
    n is Int // 判断n是不是整型类型
```

4.5 函数

a. 定义 & 调用

```
// 模板：
  fun 函数名（参数名：参数类型）：返回值类型{
    函数体
    return 返回值
}

// 说明：
//      1. 采用 “fun” 标识
//      2. 括号里的是传入函数的参数值和类型

// 示例：一个函数名为“abc”的函数，传入参数的类型是Int，返回值的类型是String
 fun abc(int: Int): String {
```

```
    return "carson_ho"
}

// 特别注意：存在简写方式，具体示例如下：
// 正常写法
fun add(a: Int, b: Int): Int {
    return a + b
}
// 简写：若函数体只有一条语句 & 有返回值，那么可省略函数体的大括号，变成单表达式函数
fun add(a: Int, b: Int) = a + b;

// 调用函数：假设一个类中有一个foo的函数方法
User().foo()
```

b. 默认参数

```
// 给int参数指定默认值为1
fun foo(str: String, int: Int = 1) {
    println("$str  $i")
}

// 调用该函数时可不传已经设置了默认值的参数，只传无设默认值的参数
foo("abc")
// 结果： abc  1

// 注：若有默认值的参数在无默认值的参数前，要略过有默认值的参数去给无默认值的参数指定值，需用命名参数来指定值
// 有默认值的参数（int）在无默认值的参数（str）前
fun foo(int: Int = 1, str: String) {
    println("$str  $i")
}

// 调用
foo(str = "hello")  // 使用参数的命名来指定值
// 结果： hello  1

foo("hello")  // 出现编译错误
```

c. 特别注意

一个函数，除了有传入参数 & 有返回值的情况，还会存在：

- 有传入参数 & 无返回值
- 无传入参数 & 无返回值

```
// 有传入参数 & 无返回值
    // 模板：
      fun 函数名（参数名：参数类型）{
        函数体
    }
    // 或返回Unit（类似Java的void，无意义）
      fun 函数名（参数名：参数类型）：Unit{
        函数体
    }

// 无传入参数 & 无返回值
    // 模板：
      fun 函数名（）{
        函数体
    }
    // 或返回Unit（类似Java的void，无意义）
      fun 函数名（）：Unit{
        函数体
    }
```

4.6 其他语法糖

关于Kotlin的一些实用语法糖，主要包括：

- 控制流（if、when、for、 while）
- 范围使用（in、downTo、step、until）
- 类型检查 & 转换（is、智能转换、as）
- 相等性（equals（）、=、 ==）

- 空安全

a. 控制流语句

控制流语句主要包括：if、when、for 和 while。

if语句

- Kotlin中的if语句与Java用法类似
- 区别在于：Kotlin的if语句本身是一个表达式，存在返回值

```
var c = if (a > b) 3 else 4

// 若a > b，则返回3给c
// 若a < b，则返回4给c
// 类似Java中的三元表达式
c = a > b ? 3 : 4; // 若a>b，c=3，否则c=4

// 若if后面是代码块
var c = if (a > b) {
    代码块1
} else {
    代码块2
}
// 若a > b，则执行代码块1，否则执行代码块2
```

when语句

类似Java中的switch语句

```
// Java中的Switch语句
int a = 0;
switch (a) {
    case 0:
        break;
    case 1:
```

```
        break;
    default:
        break;
}

// Kotlin中的when语句
var a = 0
when (a) {
    0 -> {代码块1}
    1 -> {代码块2}
    2,3 -> {代码块3}
    else -> {代码块4}
}
// 说明：
// 当a=0时执行代码块1
// 当a=1时执行代码块2
// 当a=2，3时，执行代码块3
// 当a=其他值时，执行代码块4

// 注意：when语句在满足条件的分支执行后，会终止when语句执行
```

for语句

类似Java中的for语句

```
// 示例1：表达一个数字是否在目的范围内
// Java中的for语句
for (int i = 0; i < 4; i++) {
    System.out.println(i);
}
// Kotlin中的for语句
if (i in 1..4){
    println(i)
}
// 注：关键字用于表示数字是否在目标范围内，上面的示例表示判断i是否在代表1-4范围内

```

```
// 示例2：通过索引遍历一个数组
// Java中的for语句
for (int i = 0; i < 4; i++) {
    System.out.println(i);
}
// Kotlin中的for语句
for (i in array.indices) {
    println(array[i])
}
```

while语句

类似Java中的while语句，分为while 和 do…while语句：

```
var i = 5

while(i in 1..4){
   代码块1
}

do{ 代码块2
  }while(i in 1..4){
}
```

b. 范围使用

主要用于表示范围，主要包括：in、downTo、step、until

```
/**
 *  1. in
 *  作用：在...范围内
 **/
// 表示：若i在1-5范围内，则执行下面代码
// 注：闭区间，[1,5]
if (i in 1..5) {
    println("i 在 1-5 内")
```

```
}

// 表示：若i不在1-5范围内，则执行下面代码
// !in表示不在...范围内
if (i !in 1..5) {
    println("i 不在 1-5 内")
}

/**
 *  2. until
 *  作用：表示开区间
 **/
// 输出1234
for (i in 1 until 5) {
 println(i)
}

/**
 *  3. downTo
 *  作用：倒序判断
 **/
 for (i in 5 downTo 1) {
    println(i)
 }

/**
 *  4. step
 *  作用：调整步长
 **/
// 设置步长为2，顺序输出1、3、5
for (i in 1..5 step 2) println(i)

// 设置步长为2，倒序输出5、3、1
for (i in 1 downTo 5 step 2) println(i)
```

c. 类型检查 & 转换

```
/**
 *  1. is
 *  作用：判断一个对象与指定的类型是否一致
 **/
// 判断变量a的数据类型是否是String
var a: Any = "a"
if (a is String) {
    println("a是String类型")
}
if (a !is Int) {
    println("a不是Int类型")
}

/**
 *  2. 智能转换
 *  说明： kotlin不必使用显式类型转换操作，因为编译器会跟踪不可变值的is检查以及显式转换，并在需要时自动插入（安全的）转换
 **/
 var a: Any = "a"
if (a is String) {
    println("a是String类型")
    println(a.length) // a 自动转换为String类型
    //输出结果为：1
}

// 反向检查： a自动转换为String类型
if (a !is String) {
    print(a.length)
}

// 在 && 和 || 的右侧也可以智能转换：
// `&&` 右侧的 a 自动转换为String
if (a is String && a.length > 0)
// `||` 右侧的 a 自动转换为String
if (a is String || a.length > 0)

// 在when表达式和while循环里也能智能转换：
when(a){
```

```
    is String -> a.length
    is Int -> a + 1
}

// 需要注意：当编译器不能保证变量在检查和使用之间不可改变时，智能转换不能用。智能转换能否适用根据以下规则：
// 1. val 局部变量——总是可以，局部委托属性除外；
// 2. val 属性——如果属性是 private 或 internal，或者该检查在声明属性的同一模块中执行。智能转换不适用于 open 的属性或者具有自定义 getter 的属性；
// 3. var 局部变量——如果变量在检查和使用之间没有修改、没有在会修改它的 lambda 中捕获、并且不是局部委托属性；
// 4. var 属性——决不可能（因为该变量可以随时被其他代码修改）

/**
 *  3. 强制类型转换：as
 **/
var any: Any = "abc"
var str: String = any as String

// 强制类型转换是不安全的，若类型不兼容则会抛出一个异常
var int: Int = 123
var str: String = int as String
// 抛出ClassCastException

/**
 *  4. 可空转换操作符：as?
 *  作用：null不能转换为String，因该类型不是可空的，此时使用可空转换操作符as?
 **/
var str = null
var str2 = str as String
// 抛出TypeCastException

// 使用安全转换操作符as?可以在转换失败时返回null，避免了抛出异常。
var str = null
var str2 = str as? String
println(str2) //输出结果为：null
```

d. 相等性判断

在Kotlin中，存在结构相等 & 引用相等 两种相等判断。

```
/**
 *  1. 结构相等：equals()或 ==
 *  作用：判断两个结构是否相等
 **/
var a = "1"
var b = "1"
if (a.equals(b)) {
    println("a 和 b 结构相等")
    // 输出结果为：a 和 b 结构相等
}

var a = 1
var b = 1
if (a == b) {
    println("a 和 b 结构相等")
    // 输出结果为：a 和 b 结构相等
}

/**
 *  2. 引用相等：===
 *  作用：判断两个引用是否指向同一对象
 */
// 设置一个类如下
data class User(var name: String, var age: Int)

// 设置值
var a = User("Czh", 22)
var b = User("Czh", 22)
var c = b
var d = a

// 对比两个对象的结构
if (c == d) {
    println("a 和 b 结构相等")
} else {
```

```
    println("a 和 b 结构不相等")
}

// 对比两个对象的的引用
if (c === d) {
    println("a 和 b 引用相等")
} else {
    println("a 和 b 引用不相等")
}

// 输出结果：
a 和 b 结构相等
a 和 b 引用不相等
```

e. 空安全

- 在Java中，NullPointerException异常十分常见
- 而Kotlin的优点则是可以尽可能避免执行代码时出现的空指针异常

```
/**
 *  1. 可空类型与非空类型
 *  在Kotlin中，有两种情况最可能导致出现NullPointerException
 **/

// 情况1：显式调用 throw NullPointerException()
// 情况2：使用!! 操作符
// 说明：!!操作符将任何值转换为非空类型，若该值为空则抛出异常
var a = null
a!!
// 抛出KotlinNullPointerException

// 情况3：数据类型不能为null
// 在 Kotlin 中，类型系统区分一个引用可以容纳 null （可空引用） 和 不能容纳（非空引用）
// 如：String类型变量不能容纳null
// 若要允许为空，可声明一个变量为可空字符串：在字符串类型后面加一个问号?
对于String，则是写作：String?
```

```
var b: String? = "b"
b = null



/**
 *  2. 安全调用操作符
 *  作用：表示如果若不为null才继续调用
 **/
 b?.length
 // 表示：若b不为null，才调用b.length

 // 注：安全调用符还可以链式调用
 a?.b?.c?.d
 // 假设a不为null，才继续往下调用，以此类推
 // 若该链式调用中任何一个属性为null，整个表达式都会返回null。
 // 若只对非空值执行某个操作，可与let一起使用
a?.b?.let { println(it) }
```

至此，关于 `Kotlin` 的入门语法讲解完毕。

5. 总结

- 本文全面介绍了Kotlin入门学习知识，包括定义特点、配置使用、入门语法等
- 接下来推出的文章，我将继续讲解Kotlin的相关知识，包括使用、语法特点等，感兴趣的读者可以继续关注我的博客哦：Carson_Ho的Android博客