Kotlin 进阶 讲义

次级构造

```
class CodeView : TextView {
   constructor(context: Context): super(context)
}
```

主构造器

```
class CodeView constructor(context: Context) : TextView(context)
```

如果没有被「可见性修饰符」「注解」标注，那么`constructor`可以省略

成员变量初始化可以直接访问到主构造参数

```
class CodeView constructor(context: Context) : TextView(context){
  val color = context.getColor(R.color.white)
}
```

init 代码块

主构造不能包含任何的代码，初始化代码可以放到`init`代码块中

```
class CodeView constructor(context: Context) : TextView(context) {

  init {
    //setTextSize()
  }
}
```

在初始化的时候初始化块会按照它们在「文件中出现的顺序」执行。

```
class CodeView constructor(context: Context) : TextView(context) {

  init {
    //setTextSize()
  }
  val paint = Paint()  // 会在 init{} 之后运行
}
```

构造属性

在主构造参数前面加上`var/val`使构造参数同时成为成员变量

```
class User constructor(var username: String?, var password: String?, var code: String?)
```

data class

数据类同时会生成

- `toString()`
- `hashCode()`
- `equals()`
- `copy()`(浅拷贝)
- `componentN()`...

相等性

- `==`结构相等 （调用`equals()`）
- `===`引用相等

解构

可以把一个对象「解构」成很多变量。

```
val (code, message, body) = response
```

实际的代码

```
val code = response.component1()
val message = response.component2()
val body = response.component3()
```

Elvis 操作符

可以通过 `?:` 的操作来简化 `if null` 的操作

```
// lesson.date 为空时使用默认值
val date = lesson.date?: "日期待定"

// lesson.state 为空时提前返回函数
val state = lesson.state?: return

// lesson.content 为空时抛出异常
val content = lesson.content ?: throw IllegalArgumentException("content
expected")
```

when 操作符

when 表达式可以接受返回值，多个分支相同的处理方式可以放在一起，用逗号分隔

```
val colorRes = when (lesson.state) {
  Lesson.State.PLAYBACK, null -> R.color.playback
  Lesson.State.LIVE -> R.color.live
  Lesson.State.WAIT -> R.color.wait
}
```

when 表达式可以用来取代 `if-else-if` 链。如果不提供参数，所有的分支条件都是布尔表达式

```
val colorRes = when {
  (lesson.state == Lesson.State.PLAYBACK) -> R.color.playback
  (lesson.state == null) -> R.color.playback
  (lesson.state == Lesson.State.LIVE) -> R.color.live
  (lesson.state == Lesson.State.WAIT) -> R.color.wait
  else -> R.color.playback
}
```

operator

通过 `operator` 修饰「特定函数名」的函数，例如`plus` 、`get`，可以达到重载运算符的效果

表达式	翻译为
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)

lambda

如果函数的最后一个参数是 lambda，那么 lambda 表达式 可以放在圆括号之外:

```
lessons.forEach(){ lesson : Lesson ->
  // todo
}
```

如果你的函数传入参数只有一个 lambda 的话，那么小括号可以省略的

```
lessons.forEach { lesson : Lesson ->
  // todo
}
```

如果 lambda 表达式只有一个参数，那么可以省略，通过隐式的`it` 来访问

```
lessons.forEach { // it
  // todo
}
```

循环

通过标准函数 `repeat()`

```
repeat(100) {
  // todo
}
```

通过区间

```
for (i in 0..99) {

}
// until 不包括右边界
for (i in 0 until 100) {

}
```

infix

必须是成员函数或扩展函数

必须只能接受一个参数，并且不能有默认值

```
// until 源码
public infix fun Int.until(to: Int): IntRange {
    if (to <= Int.MIN_VALUE) return IntRange.EMPTY
    return this .. (to - 1).toInt()
}
```

嵌套函数

Kotlin 中可以在函数中继续声明函数。

```
fun func(){
  fun innerFunc(){

  }
}
```

- 内部函数可以访问外部函数的参数
- 每次调用时，会产生一个函数对象

注解使用处目标

当某个元素可能会包含多种内容（例如构造属性，成员属性），使用注解时可以通过「注解使用处目标」，让注解对目标发生作用，例如 `file` 、`get`、`set` 等。

函数简化

可以通过 `=` 简化直接 `return` 的函数

```
fun get(key :String) = SP.getString(key,null)
```

函数参数默认值

可以通过函数参数默认值来代替 Java 的函数重载

```
fun toast(text: CharSequence, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, text, duration).show()
}
```

@JvmOverloads

使用 @JvmOverloads 对 Java 暴露重载函数

扩展

扩展函数可以为任何类添加上一个函数，从而代替工具类

扩展函数和成员函数相同时，成员函数优先被调用

扩展函数是静态解析的，在编译时就确定了调用函数（没有多态）

函数类型

函数类型由「传入参数类型」和「返回值类型」组成，用「->」连接，传入参数需要用「()」，如果返回值为 Unit 不能省略

函数类型实际是一个接口，我们传递函数的时候可以通过「::函数名」,或者「匿名函数」或者使用「lambda」

内联函数

使用 `inline` 关键字声明的函数是「内联函数」，在编译时会将「内联函数」中的函数体，直接插入到调用出。

所以在写内联函数的时候需要注意，尽量将内联函数中的代码行数减少

部分禁用内联

`noinline` 可以禁止部分参数参与编译

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { ......
}
```

内联函数配合函数类型，可以减少函数类型生成的对象

具体化的类型参数

因为内联函数的存在，我们可以通过配合 `inline + reified` 达到「真泛型」的效果

```
val RETROFIT = Retrofit.Builder()
        .baseUrl("https://api.hencoder.com/")
        .build()

inline fun <reified T> create(): T {
    return RETROFIT.create(T::class.java)
}

val api = create<API>()
```

抽象属性

在 Kotlin 中，我们可以声明抽象属性，子类对抽象属性重写的时候需要重写对应的`setter/getter`

委托

属性委托

有些常见的属性操作，我们可以通过委托的方式，让它只实现一次，例如

- `lazy` 延迟属性：值只在第一次访问的时候计算

- `observable` 可观察属性：属性发生改变时的通知
- `map` 集合：将属性存在一个 `map` 中

对于一个只读属性(即 `val` 声明的)，委托对象必须提供一个名为 `getValue` 的函数

对于一个可变属性(即 var 声明的)，委托对象必须额外提供一个名为 `setValue` 的函数

类委托

可以通过类委托的模式来减少继承

类委托的时，编译器回优先使用自身重写的函数，而不是委托对象的函数

```
interface Base {
  fun print()
}
class BaseImpl(val x: Int) : Base {
  override fun print() {
    print(x)
  }
}
// Derived 的 print 实现会通过构造参数中的 b 对象来完成。
class Derived(b: Base) : Base by b
```

Kotlin 标准函数

使用时可以通过简单的规则作出一些判断

- 返回自身

  从 `apply` 和 `also` 中选

  - 作用域中使用 this 作为参数选择 apply
  - 作用域中使用 it 作为参数选择 also
- 不需要返回自身

  从`run` 和`let` 中选择

  - 作用域中使用 this 作为参数，选择 run
  - 作用域中使用 it 作为参数，选择 let

`apply` 适合对一个对象做附加操作的时候

`let` 适合配合空判断的时候

`with` 适合对同一个对象进行多次操作的时候