变量与函数

变量

1. 对变量只允许两种关键字：var、val，通过**类型推导机制**判断实际数据类型，使用`?`声明允许为null的情况。
    1. `val` value，不可变的变量，初始赋值后不可再改变，类似于final，线程安全，不需要访问控制。
    2. `var` variable，可变的变量。

> 使用val为了解决final不合理使用的问题，永远优先使用val声明变量，无法满足需求时再改为var。
>
> 好的编程习惯：除非一个变量明确允许被修改，否则都应该加上final限制。

```
val str = "Hello World."
var test = 123
var a: Int? // 若对变量进行延迟赋值，则仍然需要显式的声明变量类型
```

2. 常量
    1. kotlin常量必须声明在对象或顶层中
    2. 新增修饰常量的`const`关键字
    3. 只有基本类型和String类型可以声明为常量

```
class Sample {
    companion object {
        const val CONST_NUMBER = 1  // 伴生对象
    }
}
const val CONST_SECOND_NUMBER = 2   // 顶层
```

3. 对数字**没有隐式拓宽转换**，必须进行明确的类型转换，如`toLong()`。
4. 区间变量

```
var range = 0..10 //表示一个[0,10]的区间
var range = 0 until 10  //表示一个[0,10)的区间
var range = 10 downTo 1 //表示一个[10,1]的降序区间
var range = 0 until 10 step 2 //表示{0，2，4，6，8},step关键字跳过元素,每次循环递增2
```

5. `Any`是所有非空对象的父类，如Java.Object；`Any?`是所有可空对象的父类。
6. `Unit`类型类似于void，可用于函数无返回值的情况。Unit是一个完备类型，可作为类型参数。

```
interface Test<T> {
    fun test(): T
}
class NoResultClass: Test<Unit> {
    override fun test(): Unit { }
}
```

7. 延迟初始化

   `lateinit`关键字，不需要在后续使用中进行?判空处理，但有可能出现`UninitialPropertyAccessException`异常。使用时确保调用前已完成初始化；只能修饰类属性，不能修饰局部变量，且只能用来修饰对象

```
private lateinit var adapter: MsgAdapter
if( !::adapter.isInitialized) { }   //::adapter.isInitialized判断是否初始化的固定写法
```

变量访问与判断

1. `==`比较数值相等性，`===`比较引用相等性

```
val intV_1: Int = 200
val intV_2: Int? = intV_1
val intV_3: Int? = intV_1
println(intV_2 == intV_3)   //true
println(intV_2 === intV_3)  //false
```

2. `字符串内嵌表达式`：在字符串值中引用局部变量需要变量名前加上字符`$`，还可用`{}`进行表达式运算；可使用索引运算符`[]`访问包含的单个字符

```
val intV = 100
println("intV value is $intV")  // intV value is 100
println("(intV + 100) value is ${intV + 100}")  // (intV + 100) value is 200
println("${$}100.99")   // $100.99，表示$字符使用${$}
val str = "leavesc"
println(str[1]) // 使用索引运算符[]
```

3. `Sequence` 又被称为惰性集合操作，`Sequence`可以在数据量较大或未知时，进行流式处理

```
/*
惰性：  - 只有result被使用到时才会执行，之前的代码并不会立即执行
       - 当出现满足条件的首元素后，不会执行之后的元素遍历，即4不会被遍历
println处理流程：  取出元素1 - map为2 - 判断2是否被3整除 ... 取出元素2 - map为4 - 判断4是否被3整除 ...
*/
val sequence = sequenceOf(1,2,3,4)
val result: Sequence<Int> = sequence
    .map { i ->
        println("Map $i")
        i * 2
    }
    .filter { i ->
        println("Filter $i")
             i % 3 == 0
    }
println(result.first())
```

`Sequence`懒加载实现的优点：

- 一旦满足遍历退出条件，就省略后续不必要的遍历过程
- 像`List`这种实现`Iterable`接口的集合类，**每调用一次函数就会生成一个新的**`Iterable`，下一个函数再基于新的`Iterable`执行，每次函数调用产生的临时`Iterable`

会导致额外的内存消耗，而 `Sequence` 在整个流程中只有一个。

4. 原始字符串： 三引号`"""`括起来，内部没有转义且可以包含换行及任何其它字符，也支持字符串模板`${}`

```
val str = """{"gender":“男”,"name":"zhangsan"}"""
//等价于
val str = "{\"gender\":\"男\",\"name\":\"zhangsan\"}"
```

5. 每一种数据类都提供了若干相应类，如`IntArray\ByteArray\BooleanArray`，将被编译为普通java数组`int[]\byte[]\boolean[]`等

```
val intArray = IntArray(5)
val doubleArray = DoubleArray(5) { Random().nextDouble()}
val charArray = charArrayOf('H','e','l')
val array2 = arrayOfNulls<String>[10]   //包含元素均为null，只用来创建包含元素类型可空的数组
```

函数

用单行表达式与等号定义的函数叫做**表达式函数体**，可省略返回值类型；对于一般有返回值的**代码块函数体**，必须显式写出返回类型和return语句。

```
// 代码块函数体
fun methodName(param1: Int, param2: Int) : Int {
    return 0
}
// 表达式函数体
fun largeNumber(num1: Int, num2: Int): Int = max(num1,num2)
```

常用关键字

`if-else`

```
//完全等同于java的语法
if(num1 > num2) {
    value = num1
} else {
    value = num2
}
val value = if(num1 > num2) num1 else num   // 返回值简化,类似于三目运算value = num1>num2 ? num1 : num2
```

`when`

1. 控制流：分支条件可以是任意表达式 2.表达式：符合条件的分支的值就是整个表达式的值,类似if、try catch

```
when (value) {  // 类似于switch语句，但允许传入任意类型的参数，类型匹配
    in 4..9 -> println("in 4..9") //区间判断
    3 -> println("value is 3")    //相等性判断
    2, 6 -> println("value is 2 or 6")    //多值相等性判断
    is Int -> println("is Int")   //类型判断
    else -> println("else")       //如果以上条件都不满足，则执行 else
}
```

循环语句

1.where循环与java语法完全相同　2.for-i循环直接被舍弃，而将for-each变为了for-in循环

```
for(i in 0..10) {
    println(i)
}
for(i in 0 until 10 step 2) {
    println(i) //0.2.4.6.8
}
// for循环通过索引遍历
for(index in items.indices) {
    println("${items[inedx]}")
}
```

空安全

Kotlin将空指针异常检查提前到编译期，若代码存在异常风险则编译报错。

2. `?.`：安全调用运算符，把null检查和方法调用合并为一个操作
3. `?:`：Elvis运算符，用于替代`?.`直接返回默认值null的情况，若运算数1不为null，则结果为运算数1，否则为运算数2
4. `!!`：非空断言，确信该处对象不为空，跳过空指针检查，若该变量为null值则抛出NPE异常

```
if(name != null) {
    println(name.toUpperCase())
}else {
    println(null)
}
println(name?.toUpperCase())    // 等价
if(name != null) {
    println(name)
}else {
    println("default")
}
println(name ?: "default")  // 等价
```

`object`

1. 创建一个继承自某类型的匿名类对象，且该类内部可修改外部变量

```
fun showFragment() {
    var position = 0
    TransitionSet().addListener(object: TransitionListenerAdapter() {
        override fun onTransitionStart(transition: Transition?) {
            position++  //可访问、修改外部变量
            //java中 匿名内部类若需要访问、修改外部变量，需要使用final限制该类
        }
    })
}
```

2. 匿名对象只有定义为局部变量、private成员变量时，才可体现其真实类型；匿名对象为public函数返回值或public属性时，只能将其视为Any，属性、方法不可被访问

```
private fun foo() = object {
    val x:Int = 1
}
fun PublicFoo() = object {
    val x:Int = 1
}
fun test() {
    val adHoc = object {     //匿名局部对象变量
        var x:Int = 0
    }
    val x0 = adHoc.x
    val x1 = foo().x
    var x2 = publicFoo().x  //error 定义为public的匿名对象的属性、方法不可被访问
}
```

3. 创建单例类对应于Java的饿汉式单例模式，始终线程安全

```
object Singleton { }
```

4. companion伴生对象，类比static的静态方法、成员，也可用顶层方法实现。
    - 一个类中只可定义一个`companion object`，可以直接使用类名引用伴生对象，伴生对象在类加载时初始化。对应于Java生成一个静态内部类
    - `@JvmStatic`注解将使方法编译为真正的静态方法，只能加在`单例类`或`companion object`的方法上。
    - 顶层方法指没有定义在任何类中的方法，编写在.kt文件中的方法都是顶层方法。

    > - 如果实现工具类功能，直接创建文件，写top_level顶层函数
    > - 如果需要继承其他类或实现接口，使用`object`或`companion object`

`data`

data 数据类，简化Java生成POJO的大量模板代码，自动生成hashCode、equals、componentx、copy、toString等方法。

```
data class cellphone(val brand: String, val price: Double)
```

对于data数据类：

1. 主构造函数需要至少一个参数，可以有默认值；所有参数都需要标记为val、var，表示为该数据类的成员变量
2. final限制，不能为抽象、开放、密封、内部的

3. 数据类可以有超类

对于data数据类由kotlin编译器自动从主构造函数中声明的所有属性导出的方法：

1. equals|hashcode　对象比较
2. toString　输出对象字符串`User(name=john,age=42)`
3. componentN　按声明顺序对应于所有属性，可用于解构声明
4. copy　复制一个对象改变一些属性，其余部分不变

```
val jack = User("Jack",1)
val olderJack = jack.copy(age = 2)  // copy一个对象，指定修改的部分
// componentN函数的解构声明
val (name,age) = jack
println("$name,$age years of age")  // Jack,1 years of age
val (name,_) = jack // 只取某参数，可以使用_占位符替代。
```

类型的判断和强转

`as`：强制类型转换，类比Java.instanceOf；`as?`：安全的强制类型转换，避免了类型错误抛出异常

```
class NewActivity: MainActivity() {
    fun action() {}
}
// java
void main() {
    Activity activity = new NewActivity();  // 子类对象指向父类引用，java中称为多态
    if(activity instanceof NewActivity) {
        ((NewActivity)activity).action();
    }
}
// kotlin
fun main() {
    var activity: Activity = NewActivity()
    activity.action()   // 无法直接调用
    // 使用is关键字进行类型判断
   if(activity is NewActivity) {
       activity.action()    // 正常执行
   }
   // 使用as关键字直接进行强转调用
   (activity as NewActivity).action()   // 正常运行，但有强转错误类型的异常隐患
   // 使用as?关键字进行安全强转
   (activity as? NewActivity).action()  // 强转成功就执行，否则不执行任何操作
}
```

面向对象

实例化对象

```
val p = Person()    //不需要new关键字
```

`open`

需要为基类加上open关键字，该类才可被子类继承。默认任何非抽象类都不可被继承。

> 如果一个类不是专门为继承设计，则应加上final声明，禁止其被继承

1. `override`重写方法有遗传性，可被其子类重写，函数可见性继承自父类，需要加上`final`关闭遗传
2. `open`没有遗传性，子类仍然需要加上`open`来表示可继承

```
open class Person {
    var name = ""
    var age = 0
    fun eat() {
        println(name + "is eating. He is " + age + " years old.")
    }
}
```

继承、接口

继承基类、接口都使用`:`关键字，使用`,`隔开，基类需要加括号，接口不需要。

```
class Student : Person(),Comparable {   // 子类主构造函数调用父类中的构造函数，直接通过括号指定
    ...
}
// 接口函数可默认实现
interface Study {
    fun readBooks()
    fun doHomework() {
        println("...")
    }
}
```

主构造函数

最常用的构造函数，没有函数体，直接定义在类名后

如果在主构造函数的参数声明时加上`var`或者`val`，就等价于在类中创建了该名称的属性，并且初始值就是主构造器中该参数的值。

```
class Student(val sno: String, val grade: Int) : Person() { }
//init方法可以编写主构造函数中的逻辑
init{
    println("sno is " + sno)
    println("grade is " + grade)
}
```

次构造函数

所有的次构造函数都必须调用主构造函数,通过constructor关键字定义

```
class Student(val sno: String, name: String): Person(name) {
    constructor(name:String) : this("", name) { }
    constructor() : this("", "") { }
}
```

可见性修饰符

1. public　所有类可见，Kotlin默认为public
2. protected 对当前类、子类可见
3. internal　Kotlin同一模块中的类可见
4. private　对当前类、同包下类可见

Lambda表达式

Lambda是一小段可以作为参数传递的代码，最后一行代码会自动作为Lambda表达式的返回值。

函数并不能传递，传递的是对象 | 匿名函数和 Lambda 表达式都是对象

使用方法

1. 如果Lambda是函数的最后一个参数，可以把Lambda写在括号的外面

```
view.setOnClickListener() { v: View ->
    switchToNextPage()
}
```

2. 如果Lambda是函数唯一的参数，可以直接去掉括号

```
view.setOnClickListener { v: View ->
    switchToNextPage()
}
```

3. 如果Lambda是单一参数，则参数可省略

```
view.setOnClickListener {
    switchToNextPage()
}
```

4. Lambda对省略的唯一参数有默认的名字`it`

```
view.setOnClickListener {
    switchToNextPage()
    it.setVisibility(GONE)
}
```

注意事项

1. Lambda通过上下文推断，得知参数类型和返回值类型

```
fun setOnClickListener(onClick: (View) -> Unit) {
  this.onClick = onClick
}   // 调用的函数声明处有明确的参数信息
```

2. 如果把一个匿名函数赋给变量时，不可省略掉Lambda的参数类型

```
val b = fun(param: Int): String {
    return param.toString()
}
val b= { param: Int ->
    return param.toString()
}
val b = {
    return it.toString()    // it报错，无法上下文推断出参数类型
}
val b: (Int) -> String = {
    it.toString()   // 变量显式声明类型后，it可推断为Int
}
```

3. Lambda表达式不用`return`返回，而直接取最后一行代码值，`return`将直接作为外层函数返回值而结束外层函数
4. 匿名函数、Lambda可以作为参数传递，也可赋值给变量，实质为一个函数类型的对象，与`::method`相同。

使用实例

```
// 1.java中设计回调
public interface OnClickListener {
    void onClick(View v);
}
public void setOnClickListener(OnClickListener listener) {
    this.listener = listener;
}
// 2.使用
view.setOnClickListener(new OnClickListener() {
   @Override
   void onClick(View v) {
       switchToNextPage();
   }
});
// kotlin中改写
fun setOnClickListener(onClick: (View) -> Unit) {
    this.onClick = onClick
}
view.setOnClickListener(fun(v: View): Unit) {
    switchToNextPage()
}
// 简化为lambda
view.setOnClickListener {
    switchToNextPage()
}
```

其它

Kotlin高阶函数

1. java中的常规调用方法

```
// 1.在java中，a方法需要调用b方法
int a() {
    return b(1);
```

```
}
a();
// 2.在a调用时动态设置b方法的参数，需要传参给a，由a带给b
int a(int param) {
    return b(param);
}
a(1); a(2);
// 3.在a方法内有一处对其他方法的调用，但不确定该方法，动态设置方法本身
int a(??? method) {
    return method(1);
}
a(method1); a(method2);
// 4.将方法作为参数传入另一个方法 -> 使用接口包装方法，将接口作为对象参数传递
public interface Wrapper {
    int method(int param);
}
int a(Wrapper wrapper) {    // 将接口作为外部方法参数传入
    return wrapper.method(1);
}
a(wrapper1); a(wrapper2);
```

2. kotlin中，函数参数可以是函数类型

```
fun a(funParam: Fun): String {
    return funParam(1)
}
fun b(param:Int): String {
    return param.toString()
}
a(b)    // a(b) -> b(1) -> 1.toString()
```

- 函数类型的参数，需要指明参数个数、参数类型、返回值类型

```
fun a(funParam: (Int) -> String): String {
    return funParam(1)
}
```

- 函数类型也可作为函数的返回值类型

```
fun c(param: Int): (Int) -> Unit {
    ...
}
```

3. Kotlin 里，这种**参数有函数类型或者返回值是函数类型的函数**，都叫做高阶函数。

函数参数默认值

1. 默认参数必须按照函数声明参数顺序给定，只有末尾参数可以省略。
2. 命名参数可省略任何有默认值的参数，也可按任何顺序传入参数

```
fun printParams(num: Int, str: String = "hello") { }
//键值对传参
printParams(str = "world", num = 123)
printParams(num = 444, "hello") //error，指定了一个参数的名称后，之后的所有参数都需要标明名称
```

3. 可变参数可以把任意个数的参数打包数组中传给函数，使用`vararg`关键字声明。Kotlin需要将数组解包才能传给可变参数

```
fun compute(vararg name: String) {  //可变参数
    name.forEach { println(it)}
}
compute()
compute("a")
compute("a","B","c")
val names = arrayOf("leavesc","LeaveSC","asda")
compute(* names)    //显式的解包数组，* array固定写法
```

标准函数

1. with|run|apply

   标准函数指Standard.kt文件中定义的函数

   ```
   val list = listOf("Apple","Banana","Orange","Pear","Grape")
   val result = with(StringBuilder()) {    // param1:任意类型对象，param2:Lambda表达式，param1为表达式上下文，表达式最后一行代码为返回值。
       append("Start eating.\n")
       for(fruit in list) {
           append(fruit).append("\n")
       }
       append("Ate all fruits.")
       toString()
   }
   val result = StringBuilder().run {  //run,在某个对象的基础上调用，只接收一个Lambda表达式，且提供对象的上下文，最后一行代码为返回值。
       append("Start eating.\n")
       for(fruit in list) {
           append(fruit).append("\n")
       }
       append("Ate all fruits.")
       toString()
   }
   val result = StringBuilder().apply {    //apply,无法指定返回值，只能返回调用对象本身
       append("Start eating.\n")
       for(fruit in list) {
           append(fruit).append("\n")
       }
       append("Ate all fruits.")
   }
   println(result.toString())
   ```

扩展函数、属性和运算符重载

1. 扩展函数

   可以定义在任何一个现有类中，不一定需要创建新文件，不过最好定义为顶层方法(Top Level)内，不属于任何类，可以在任何类里使用。

```
package com.example
fun String.method1(i: Int) {
    ...
}

"example".method1(1)
```

1. 当给声明的函数名左边加上类名时，表示给该函数限定一个Receiver，虽然该函数是Top-Level Function，不属于任何类，但被限制了只有通过某个类的对象才可以调用该函数。
2. 扩展函数也可以被写在类中，可以在类中使用前缀类的对象调用该函数。此时该函数属于外部类的成员函数，又属于前缀类的扩展函数，只能在其所属的类中被调用。

```
class Example {
    fun String.method2(i: Int) {}

   "example".method2(1) // 可使用
}
"exmaple".method2(1)    // 不可使用
```

3. 扩展函数是**静态分发**的，由函数调用所在表达式的类型决定，而非由运行时求值结果决定

```
open class Shape
class Rectangle: Shape()
fun Shape.getName() = "Shape"
fun Rectangle.getName() = "Rectangle"
fun printClassName(s: Shape) {  //只取决于参数s的声明类型
    println(s.getName())
}
printClassName(Rectangle()) //输出Shape
```

4. 如果一个类有同名、同接收者类型、同参数的成员函数、扩展函数，成员函数优先

```
class Example {
    fun printFunctionType() { println("class method.")}
}
fun Example.printFunctionType() { println("extension method.")}
Example().printFunctionType()   //Class method.
```

5. 扩展函数内可使用this，但不能访问私有、保护成员

2. 扩展属性

扩展属性没有真的为类添加属性，只通过get、set显式定义，相当于定义了属性访问器方法，没有幕后字段field，因此没有初始化器

```
var Float.dp
    get() = TypedValue.applyDimension(
        TypedValue.COMPLEX_UNIT_DIP,
        this,
        Resources.getSystem().displayMetrics
    )
...
val RADIUS = 200f.dp
```

3. 运算符重载

operator关键字，在指定函数之前修饰即可实现运算符重载（plus\minus\times\div\rem）

```
operator fun plus(money: Money): Money {
    val sum = value + money.value
    return Money(sum)
}
val money1 = Money(5)
val money2 = Money(10)
val money3 = money1 + money2
//还可根据不同参数类型对同一函数实现多个重载
operator fun plus(newValue: Int): Money {
    val sum = value + newValue
    return Money(sum)
}
```

指向函数的引用

对于一个声明好的函数，不管作为函数参数，或赋值给变量，都需要双冒号`::`关键字

```
a(::b)
val d = ::b
```

1. 函数引用Function Reference，**函数可以作为参数**的本质，是函数可以作为对象存在，使用双冒号表明将其作为一个和函数具有相同功能的对象，`::method`不是一个函数，而是一个函数类型的对象

```
b(1)    // 调用函数
(::b)(1)    // 等价b(1)，实际上调用(::b).invoke(1)
b.invoke()  // 报错，不能对一个函数调用invoke()，可以对一个函数类型的对象调用invoke()
```

2. 普通函数可以被指向，顶层扩展函数也可以被指向，但成员扩展函数不可被引用

```
fun String.method1(i: Int) {
}
String::method1
```

3. 扩展函数的引用也可被调用，直接调用或invoke()，但Receiver需要填为首个参数

```
(String::method1)("example", 1) // example.method1(1)
String::method1.invoke("test", 2)
```

4. 扩展函数的引用也可以赋值给变量

```
val a: String.(Int) -> Unit = String::method1
"example".a(1)
a("example", 1)
a.invoke("example", 1)
```

泛型和委托

泛型

1. 泛型允许在不指定具体类型的情况下进行编程，代码具有更好的扩展性。

```
class MyClass<T> {
    fun method(param: T): T {
        return param
    }
}
val myClass = MyClass<Int>()
val result = myClass.method(123)
//泛型方法
class MyClass {
    fun <T> method(param: T): T {
        return param
    }
}
val myClass = MyClass()
val result = myClass.method<Int>(123)
//对泛型进行类型限制，泛型的上界默认为Any?
fun <T: Number> method(param: T): T {
    return param
}
```

类委托和委托类型

委托模式：有两个对象参与处理请求，接受请求的对象将请求委托给另一对象来处理，通过关键字`by`指定实现者

1. 类委托：将一个类的具体实现委托给另一个类完成，如Set -> HashSet。大部分方法实现调用辅助对象方法，小部分自己重写，或加入独有方法

   ```
   class MySet<T>(val helperSet: HashSet<T>): Set<T> {
       override val size: Int
           get() = helperSet.size
       override fun contains(element: T) = helperSet.contains(element)
       ...
   }
   ```

   委托的关键字为`by`，可免去模板代码，只需要写特定功能。

   ```
   class MySet<T>(val helperSet: HashSet<T>): Set<T> by helperSet {
       fun helloworld() = println("Hello World.")
       override fun isEmpty() = false
   }
   ```

2. 委托属性：一个类的某属性值不是在类中直接定义，而委托给一个委托类，实现对这一类的属性的统一管理

3. lazy函数：把想要延迟执行的代码放到by lazy代码块中，当该属性首次被调用时，才执行代码块。

   ```
   val p by lazy{ ... }
   //实际是通过by将p委托给lazy{ ... }这一高阶函数
   ```

协程

协程是可以由程序自行控制挂起、恢复的程序；协程可以实现多任务的协作执行；协程可以用来解决异步任务控制流的灵活转移

- 让异步代码同步化
- 降低异步程序的设计复杂度
- 让异步代码更简单

> 为了保证界面流畅、及时响应用户输入事件，主线程保持16ms/次的刷新(调用 `onDraw()` 函数)，所以不能在主线程中做耗时的操作(比如 读写数据库，读写文件，做网络请求，解析较大的 Json 文件，处理较大的 list 数据）。

协程是**结构化并发**的

> **结构化并发**：如果在 `foo` 里协程启动了 `bar` 协程，那么 `bar` 协程必须在 `foo` 协程之前完成
>
> 结构化并发可以保证当一个作用域被取消，作用域里面的所有协程会被取消

基本用法

```
// 在模块级build.gradle中添加协程依赖库
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.1.1"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android0:1.1.1"

launch {
    val user = api.getUser()     // 网络请求(IO线程)
    nameTv.text = user.name // 更新UI(主线程)
}
```

```
// 方法一，使用runBlocking顶层函数；线程阻塞的，直到block执行完毕，主要用在main函数或测试中使用
runBlocking {
    getImage(imageId)
}
//方法二，使用GlobalScope单例对象，可直接调用launch开启协程；线程不阻塞的，但生命周期与app一致，不可取消
GlobalScope.launch {
    getImage(imageId)
}
//方法三，通过CoroutineContext创建一个CoroutineScope对象，在该对象中开启协程；推荐用法，生命周期与context一致，可管理控制
val coroutineScope = CoroutineScope(context)
coroutineScope.launch {
    getImage(imageId)
}
```

基础层API

协程的基础层API包括：`suspend`挂起函数、`Continuation`、`createCoroutine`创建协程、`startCoroutine`启动协程、`CoroutineContext`协程上下文

1. 挂起和恢复主要通过`Continuation`实现，`createCoroutine`传递了一个`Continuation`实例，其主要作用是执行挂起函数恢复后的代码并得到挂起函数的返回值作为参数返回
2. 协程上下文(CoroutineContext)中存在着拦截器`ContinuationInterceptor`，可以对协程上下文所在协程的`Continuation`进行拦截，可以实现**切换线程。**

3. 协程调度器(Dispatches)它确定了相关的协程在哪个线程上执行，或将它分派到一个线程池，亦或是让它不受限地运行
4. 所有的协程构建器诸如`launch/async`接收一个可选的`CoroutineContext`参数，显式的为一个新协程或其它上下文元素指定一个调度器。

```
suspend {   // 挂起函数
    ...
}.createCoroutine(object: Continuation<Int> {   // 创建协程
    override val context: CoroutineContext  // 协程上下文
        get() = EmptyCoroutineContext
    override fun resumeWith(result: Result<Int>) {
        ...
    }
}).resume(Unit) // resume启动协程

suspend {
    ...
}.startCoroutine(object: Continuation<Int> {    // 创建并启动协程
    override val context: CoroutineContext
        get() = EmptyCoroutineContext
    override fun resumeWith(result: Result<Int>) {
        ...
    }
}).resume(Unit)
```

协程作用域

协程函数的挂起需要有一个`CoroutineScope`空间，接受`CoroutineContext`作为参数，其由一系列配置参数组成，可指定名称、线程、异常处理等，可通过`+`组合这些参数，函数挂起并不影响CoroutineScope作用域之外的代码执行（非阻塞式挂起）。

1. `GlobalScope`

全局作用域，协程与APP生命周期一致，且不能通过`GlobalScope.cancel()`取消

```
public object GlobalScope: CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = EmptyCoroutineContext
}
```

- 是一个静态单例对象，某些场景(Activity)可能造成内存泄漏

- GlobalScope中启动的活动协程并不会使进程保证存活，类似守护线程

2. `CoroutineScope`

标准库函数，可以通过`CoroutineScope.cancel()`取消协程

```
public fun CoroutineScope(context: CoroutineContext): CoroutineScope =
    ContextScope(if (context[Job] != null) context else context + Job())
val coroutineScope = CoroutineScope(Dispatches.Main)
coroutineScope.launch {
    ...
}
coroutineScope.cancel() // 取消
```

- 继承父协程的作用域，取消操作、异常传递双向传播，会在任意一个协程发生异常后取消所有的子协程的运行，如果父作用域被取消，它会把取消的信息往下传递给所有新的协程

3. `supervisorScope`

继承父协程的作用域，取消操作、异常传递只能单向传播，父->子，任意一个协程发生异常后并不会取消其他的子协程。

使用`coroutineScope`或者`supervisorScope`可以安全地在`suspend`函数里面启动新的协程，不会造成泄漏，因为总是会`suspend`调用者直到所有的协程执行完毕。

4. 协程异常处理

   一般协程出现未捕获异常时，首先取消所有子协程，可能取消父协程

   1. 当异常属于`CancellationException`，只会取消当前协程和子协程，不影响父协程
   2. 使用`SupervisorJob`对象和`SupervisorScope()`挂起函数时，子协程出现未捕获异常也不影响父协程
   3. `launch/async`协程都将自动向上传播异常，取消父协程

参数、方法

1. `Dispatches`协程调度器分类：
   1. `Dispatches.Main`　Android中的主线程，可以直接操作UI
   2. `Dispatches.IO`　针对磁盘和网络IO进行了优化，适合IO密集型的任务，比如：读写文件，操作数据库以及网络请求
   3. `Dispatches.Default`　适合CPU密集型的任务，比如解析JSON文件，排序一个较大的list
2. 关键参数
   - `CoroutineStart`协程启动模式
     - CoroutineStart.DEFAULT　不需要手动调用Job对象的join/start方法，调用launch时就自动执行block
     - CoroutineStart.LAZY　必须手动调用Job对象的join/start方法启动block
   - `Job`表示一个协程作业，是协程的唯一标识，并负责管理协程的生命周期。可以有层级关系，一个Job可包含多个child Job
     - 父协程手动调用`cancel()`或异常结束，所有子协程立即被取消
     - 父协程必须等待所有子协程完成(Completed/Cancelled)才能完成
     - 子协程抛出未捕获异常时，默认会取消父协程

```
public val isActive: Boolean
public val isCompleted: Boolean
public val isCancelled: Boolean
public fun start(): Boolean
public fun cancel(cause: CancellationException? = null)
public suspend fun join()
public fun invokeOnCompletion(handler: CompletionHandler):
DisposableHandle
// Job具有生命周期且可以取消，还可对其完成状态进行监听
```

> `cancel()`只是将协程状态更改为已取消，并不能取消block运算逻辑，大多系统API都会检测协程状态`isActive`

3. `launch`：启动一个新协程并返回`Job`对象，通常用于不关心结果的耗时任务，是CoroutineScope类型的扩展函数，可通过`cancel()`取消协程，遇异常抛出

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

4. `async`：启动一个带返回结果的协程，可通过`Deferred.await()`获取结果，异常不直接抛出，只在调用await时抛出，常用于关心结果的耗时任务

> `launch`与`async`：
>
> 相同点：都可启动一个协程，都返回Coroutine
>
> 不同点：`async`返回的Coroutine实现了Deferred接口，调用Deferred.await()即可得到结果

```
coroutineScope.launch(Dispatches.Main) {
    val avatar: Deferred = async{ api.getAvatar(user) }
    val logo: Deferred = async{ api.getCompanyLogo(user) }
    show(avatar.await(), logo.await())
}
```

4. `withContext`：传入`CoroutineContext`改变协程运行的线程，且在其闭包执行完毕后**自动切换回到原线程**；自动切换消除线程切换的嵌套，因此可放入单独函数内

```
coroutineScope.launch(Dispatches.Main) {
    val image = withContext(Dispatches.Io) {
        getImage(imageId)
    }
    avatarIv.setImageBitmap(image)
}
// withContext放入单独函数内
suspend fun getImage(imageId: Int) = withContext(Dispatches,IO) {
    ...
}   // 此时修改为val image = getImage(imageId)
```

5. `delay`：等待一段时间后再继续执行代码
6. `suspend`：声明一个挂起函数，挂起函数必须在协程或另一挂起函数中调用
    1. 线程执行到suspend函数时，1.后台线程 被系统回收或继续执行别的后台任务　2.主线程 继续工作
    2. 协程执行到suspend函数时，在withContext指定的线程继续向下执行，执行完成后，自动切换回原线程
    3. 自定义suspend函数的时机：1.进行耗时操作：IO操作和CPU计算工作时 2.需要等待的操作：5秒后再继续操作的情况

> 单独使用`suspend`关键字并不能声明一个完整的挂起函数，需要内部使用挂起函数API：`withContext、delay`指定要切换到的目标线程
>
> 挂起：**一个稍后会被自动切回来的线程调度操作**，该协程从当前线程挂起，协程与线程脱离

7. `Continuation`：表示挂起的协程在挂起点时的状态，在挂起点之后剩余应执行的代码，可看作一种回调机制，**剩余的计算。**

```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resumeWith(result: Result<T>)    // 恢复操作
}
// 扩展函数
fun <T> Continuation<T>.resume(value: T) =
    resumeWith(Result.success(value))
fun <T> Continuation<T>.resumeWithException(exception: Throwable) =
    resumeWith(Result.failure(exception))
```

1. CPS转换

```
// 编译前
suspend fun getCardsList(): Int
// 编译后
fun getCardsList(continuation: Continuation<T>): Any?
/* suspend函数都有一个Continuation类型的隐式参数
    返回结果类型由Int变为Any?
    挂起函数若被挂起会返回COROUTINE_SUSPENDED，执行完毕直接返回Int或异常。Any?为一个联合体
*/
```

2. `Continuation`在Kotlin中角色
    1. 执行到`suspend`函数时(CPS隐式传递Continuation参数)挂起，暂时不执行剩余协程代码
    2. 当`suspend`函数执行完毕，通过`Continuation`参数的`resume()`回调，继续执行剩下的协程代码

完整例子

架构组件中适合在`viewModel`中启动协程，在`onCleared`中取消协程。提供了`viewModelScope`扩展属性，该属性内初始化一个CoroutineScope，指定Dispatches.Main和Job

1. 异步操作改造

```
// 常见的异步回调：enqueue中创建回调实例接受回调
retrofit.create(MyService::class.java).listRepos("test")
    .enqueue(object: Callback<List<Repo>> {
        override fun onResponse(call: Call<List<Repo>>, response:
Response<List<Repo>>) {
            ...
        }
        override fun onFailure(call: Call<List<Repo>>, t: Throwable) {
            ...
        }
    })
// 协程改造的异步操作
@GET("user/{user}/repos")
suspend fun listRepoKx(@Path("user") user: String?): List<Repo> // 将retrofit网络请求接口声明为suspend挂起函数

try {
    GlobalScope.launch {    // 声明一个协程作用域，在内部调用挂起函数，函数执行完毕后恢复到此拿到结果
        val repos = retrofit.create(MyService::class.java).listRepoKx("test")
```

```
    }
} catch(e: Exception) {
    ...
}
```

2. 多个网络请求需要等待同步结束后再执行UI操作

```
// 回调式写法
api.getAvatar(user) { avatar ->
    api.getCompanyLogo(user) { logo ->
        show(merge(avatar, logo))
    }
}
// 协程,协作式任务
coroutineScope.launch(Dispatches.Main) {
    val avatar = async { api.getAvatar(user) }
    val logo = async { api.getCompanyLogo(user) }
    val merged = suspendingMerge(avatar, logo)
    show(merged)
}
```

3. 协程简化Dialog

```
suspend fun Context.alret(title: String, msg: String): Boolean =
    suspendCancellableCoroutine { continuation ->
        AlertDialog.Builder(this)
            .setNeutralButton("No") { dialog, _ ->
                dialog.dismiss()
                continueation.resume(false)
            }.setNegativeButton("Yes") { dialog, _ ->
                dialog.dismiss()
                continuation.resume(true)
            }.setTitle(title).setMessage(msg)
            .setOnCancelListener {
                continuation.resume(false)
            }.create()
            .also { dialog ->
                continuation.invokeOnCancellation { // 协程取消后将Dialog取消
                    dialog.dismiss()
                }
            }.show()
    }
// lifecycleScope和Activity/Fragment的生命周期绑定,可以直接拿到弹窗点击的结果进行处理
show_dialog.setOnClickListener {
    lifecyclesScope.launch {
        val myCheck = alret("警告", "Do you want this?")
    }
}
```

4. 协程简化Handler

```
suspend fun <T> Handler.run(block: () -> T) = suspendCoroutine<T> { continuation
->
    post {
        try {
            continuation.resume(block())
```

```
        } catch(e: Exception) {
            continuation.resumeWithException(e)
        }
    }
}
suspend fun <T> Handler.runDelay(delay: Long, block: () -> T) =
suspendCancelableCoroutine<T> { continuation ->
    val message = Message.obtain(this) {
       try {
           continuation.resume(block())
       } catch(e: Exceoption) {
           continuation.resumeWithException(e)
       }
    }.also {
        it.obj = continuation
    }
    continuation.invokeOnCancellation {
        removeCallbacksAndMessages(continuation)
    }
    sendMessageDelayed(message, delay)
}
lifecyclesScope.launch {
    val hanlder = Handler(Looper.getMainLooper())
    val h1 = handler.run("test")
    val h2 = handler.runDelay(1000)
}
```