# Kotlin 函数式编程

乔禹昂

# 自我介绍

Android 工程师

Kotlin 传教士

Kotlin 中文站译者

Kotlin 中文社区运营

什么是函数式编程?

**“一种编程范式，与“面向对象编程”，以及“面向过程编程”在层次上是类似的概念。”**

# 一个直观的例子

```
// Java 8 之前，匿名类
view.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // do something
    }
});

// Java 8 开始，lambda 表达式
view.setOnClickListener(v -> {
    // do something
});
```

# SAM 接口

```java
/**
 * 接口内只有一个抽象方法的接口，
 * 被称为 SAM 接口（Single Abstract Method）
 * 即——单抽象方法接口
 */
public interface OnClickListener {
    void onClick(View v);
}
```

函数式编程能做到什么?

**简单的说：“可以把函数（或一个代码块）作为对象传递（例如作为另一个函数或代码块的参数或返回值）”**

# 内容提要

- 一. 函数类型
- 二. lambda 表达式
- 三. 高阶函数
- 四. 实战

# 一. 函数类型

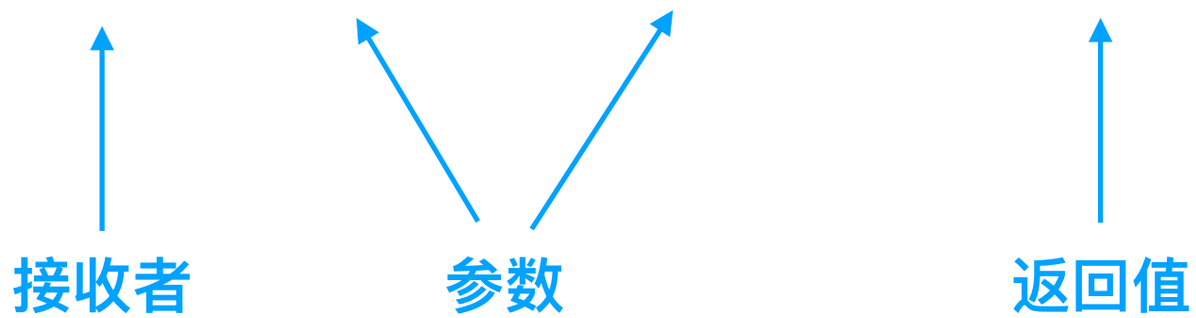
# 哪些因素决定了函数类型?

- 1.接收者类型（成员函数、扩展函数）
- 2.参数类型
- 3.返回值类型

# 一个例子

```
fun Int.sample(a: Float, b: Double): Long =
        this * (a + b).toLong()
```

```
val function: Int.(Float, Double) -> Long = Int::sample
```



# 一个简单的高阶函数

```
fun main() = println(sample(function)) // 输出: 9
```

```
fun sample(a: Int.(Float, Double) -> Long): Long = 3.a(1f, 2.0)
```

# 函数类型的继承关系

**(Fruit) -> Animal**

继承

继承

这三个函数类型之间
有继承关系吗?

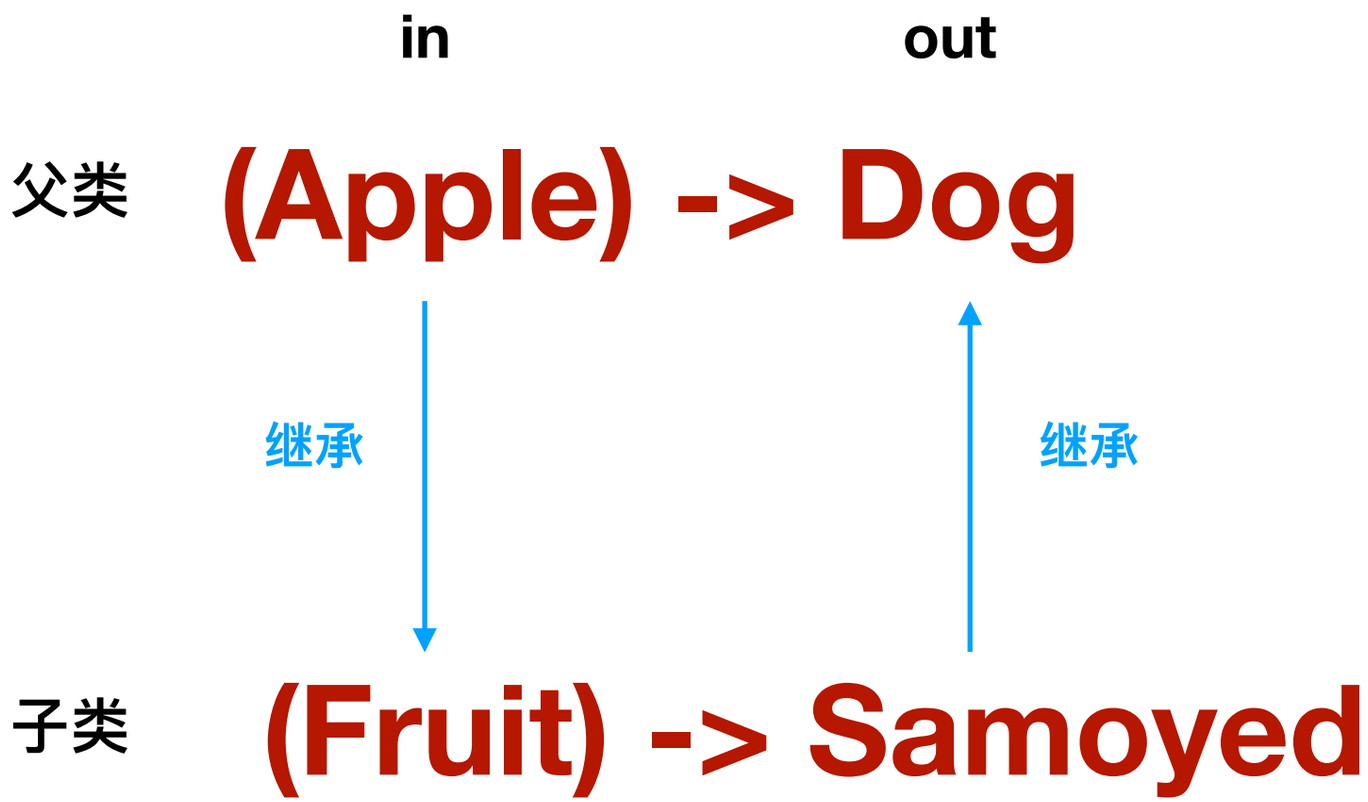
**(Apple) -> Dog**

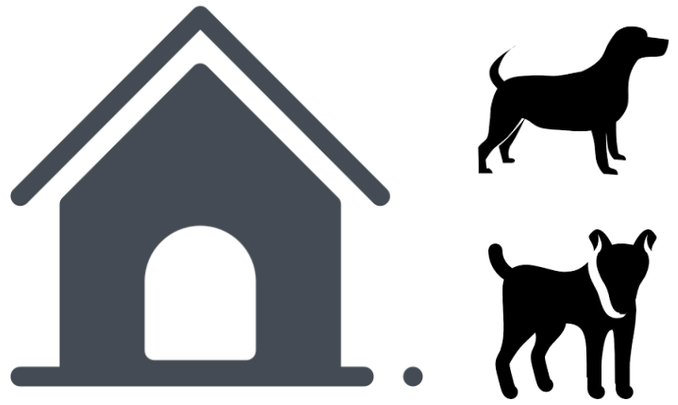
继承

继承

**(Fuji) -> Samoyed**

# 函数类型的继承关系



| | in（参数） | | out（返回值） |
|---|---|---|---|
| 父类 |  |  | (Apple) -> Dog |  |
| | 只接收苹果的购物车 | | 可以养各种狗的狗屋 |
| 子类 |  |  | (Fruit) -> Samoyed |  |
| | 能接收各种水果的购物车购物车 | | 只养萨摩耶的狗屋 |

```kotlin
open class Fruit
class Apple : Fruit()

open class Dog {
    open fun bark(): String = "汪汪"
}
class Samoyed : Dog() {
    override fun bark(): String = "萨摩耶：汪汪"
}

fun parent(apple: Apple): Dog {
    println(apple)
    return Dog()
}

fun child(fruit: Fruit): Samoyed {
    println(fruit)
    return Samoyed()
}
```

```kotlin
var functionValue: (Apple) -> Dog = ::parent

fun main() {
    val apple = Apple()
    var dog = functionValue(apple)
    println(dog.bark()) // 输出："汪汪"

    functionValue = ::child
    dog = functionValue(apple)
    println(dog.bark()) // 输出："萨摩耶：汪汪"

    /**
     * 报错，因为 functionValue 的函数类型的
     * 参数的静态类型仍然是 Apple
     */
    val fruit = Fruit()
    functionValue(fruit)
}
```

# 成员引用

```
// 最简单的成员引用

fun main() = function1(::functionParam)

fun function1(a: (Int) -> Unit) = a(5)

fun functionParam(a: Int) = println(a)
```

# 成员引用

```
fun main() {
    /**
     * 使用成员引用，即使用类名引用成员函数（扩展函数）
     * 此时 Test::functionParam 的类型为 Test.(Int) -> Unit
     */
    function1(Test::functionParam)

    /**
     * 使用绑定引用，即直接使用对象去引用成员函数（扩展函数）
     * 此时 test::functionParam 的类型为 (Int) -> Unit
     */
    val test = Test()
    function2(test::functionParam)
}

class Test {
    fun functionParam(a: Int) = println(a)
}

fun function1(a: Test.(Int) -> Unit) = Test().a(6)
fun function2(a: (Int) -> Unit) = a(6)
```

# 二. lambda 表达式

- lambda 表达式就是一个拥有函数类型的代码块

- lambda 表达式是一种更灵活的函数

# 编写一个 lambda

// 将一个 lambda 表达式赋值给一个变量

```
val a = { x: Int, y: Int ->
    println("$x $y")
    (x + y).toDouble()
}
```

/* 花括号的范围即是 lambda 的范围 */

/* 根据第一行和最后一行可以推断出，该 lambda 的类型是: (Int, Int) -> Double */

// lambda 的最后一行就是该 lambda 的返回值

# 将lambda传递给高阶函数

```
fun main() {
    // 将 lambda 保存到一个值中，再传递给高阶函数
    sample18(lambdaValue)

    // 使用成员引用将函数传递给高阶函数
    sample18(::lambdaFunction)

    // 在调用高阶函数的地方直接声明 lambda，类似匿名类
    sample18({ x, y ->
        (x + y).toDouble()
    })

    // 更近一步，如果lambda 是高阶函数的唯一参数，可以省略括号
    sample18 { x, y ->
        (x + y).toDouble()
    }
}

val lambdaValue = { x: Int, y: Int ->
    (x + y).toDouble()
}

fun lambdaFunction(x: Int, y: Int): Double = (x + y).toDouble()

fun sample18(a: (Int, Int) -> Double): Double = a(3, 6)
```

# lambda 表达式应用

```
/**
 * Kotlin lambda 表达式可以用在所有的
 * Java SAM 接口上
 */
view.setOnClickListener { it: View!
    // view 响应点击事件
}

view.viewTreeObserver.addOnDrawListener {
    // view 绘制监听
}

dialog.setOnDismissListener { it: DialogInterface!
    // 监听 dialog 关闭
}
```

# Kotlin lambda 表达式是如何在 JVM 上实现的?

# Java 中的相关原理

- Java 的匿名类原理：匿名类在编译期会生成新的 .class 文件。
- Java 8 的 lambda 表达式原理：编译期不产生额外结果，在运行时会生成一个静态方法，将 lambda 表达式的代码保存在静态方法中，然后生成一个 SAM 接口的实现类，在该实现类中会调用之前生成的静态方法。class 字节码调用指令使用的是 Java 7 新增的 invokeDynamic。

# 两种实现的优缺点

- 匿名类：频繁使用匿名类会在编译期创建大量 class 文件，导致程序包变大，但运行时没有其它开销。

- lambda：在编译期不会做额外工作，即不会让程序包变大，但在运行时会动态生成代码，拖慢运行时的速度。

# Kotlin 的选择

- Kotlin 目前的 lambda 表达式在编译后会生成名为 FunctionN 接口的匿名类（无论配置的Java target 版本是多少）。

- Kotlin 未来计划会支持 Java 8 lambda 表达式的字节码。

# FunctionN

```kotlin
/**
 * 我们在标准库中找不到这些接口的定义,
 * 但是可以在 Java 代码中访问它们,
 * 其原理是编译器使用了一种叫"合成的编译器生成类型"的技术
 */
interface Function0<R> {
    operator fun invoke(): R
}

interface Function1<P1, R> {
    operator fun invoke(p1: P1): R
}

interface Function2<P1, P2, R> {
    operator fun invoke(p1: P1, p2: P2): R
}

interface Function3<P1, P2, P3, R> {
    operator fun invoke(p1: P1, p2: P2, p3: P3): R
}
```

# 例子： Kotlin

```kotlin
/**
 * 假设我们有如下 Kotlin 代码
 */
fun main() {
    sample { p1, p2 -> p1 + p2 }
}

fun sample(call: (Int, Int) -> Int): Int = call(5, 6)
```

# 例子：编译结果

```
// 编译为 class 字节码之后对应的 Java 代码为:
public static void main(String[] args) {
    sample(new Function2<Integer, Integer, Integer>() {
        @Override
        public Integer invoke(Integer p1, Integer p2) { return p1 + p2; }
    });
}

public static int sample(Function2<Integer, Integer, Integer> call) {
    return call.invoke(5, 6);
}
```

# 三.高阶函数

“一种接受函数作为参数或返回值的函数”

# 声明高阶函数

```
// 接收一个函数类型作为参数
fun higherOrderFunction1(paramFunction: (Int) -> Unit) =
        paramFunction(6)

// 返回值类型为函数类型
fun higherOrderFunction2(): (Int) -> Unit = { it: Int
    println(it)
}

// 接收一个高阶函数作为参数，即一个高阶函数的参数也是高阶函数
fun higherOrderFunction3(paramFunction: ((Int) -> Unit) -> Unit) =
        paramFunction { println(it) }

// 返回值类型为一个高阶函数，即高阶函数又返回了一个高阶函数
fun higherOrderFunction4(): ((Int) -> Unit) -> Unit =
        this::higherOrderFunction1
```

# 抹消高阶函数的运行时开销
# ——内联函数

- 概念：内联函数可以把高阶函数的代码以及在调用该高阶函数处的 lambda，一起复制到函数的调用处。

- 优化一：避免创建匿名类的 .class 文件。

- 优化二：理论上，在虚拟机层面上，由于整个高阶函数的代码都被复制过去，所以会避免为该函数创建栈帧。

# 一个例子

```
// 内联函数——使用'inline'修饰符
inline fun higherOrderFunction(
        paramFunction: (Int) -> Unit) {
    var count = 0
    while (count < 10) {
        paramFunction(count)
        count++
    }
}

// 编译前
fun call() {
    println("开始")
    higherOrderFunction { it: Int
        println(it)
    }
    println("结束")
}
```

# 编译前后

```
// 编译前
fun call() {
    println("开始")
    higherOrderFunction { it: Int
        println(it)
    }
    println("结束")
}
```

```
// 编译后
fun call() {
    println("开始")
    var count = 0
    while (count < 10) {
        println(count)
        count++
    }
    println("结束")
}
```

Q：“是不是所有高阶函数都能声明成内联函数？”

A：编译器对高阶函数有两条限制：

1.内联函数必须直接执行它的函数类型参数。
2.内联函数中调用的其它函数必须在内联函数的调用处是可以被访问的。

Q：“既然内联函数也能抹消函数调用的栈帧创建，为了性能，是否应该将所有普通的函数也声明成内联函数？”

A：“内联函数在 Kotlin 中只有两种场景应该被使用：

1.优化高阶函数的性能；

2.支持泛型实化；

其它场景不应该使用。”

Q：“只要编译器不反对，是否任意一个高阶函数都应该声明成内联函数？”

A：“在大多数情况下是这样的”；

特例：当高阶函数内的代码太长时，即使它的语法上不违反内联函数的限制；此时，要么它不应该被声明成内联函数；要么，开发者应该将它优化，例如将其内部的代码封装到几个独立的函数中。

# 不想内联所有的函数类型参数?

```
// 内联函数——使用'inline'修饰符
inline fun higherOrderFunction(
        param1: (Int) -> Unit,
        // 不想被内联的函数，使用'noinline'修饰符
        noinline param2: (Int) -> Unit):
        (Int) -> Unit = param2
```

# 四.实战

- 实战一：不变性与纯函数

- 实战二：函数组合

# 实战一：不变性与纯函数

- 1.不变性：一个变量在声明后即被赋值，之后不能更改。

- 2.纯函数：一个函数的输出只与输入有关，即该函数不引用任何作用域外部的变量。

```kotlin
// 传统写法
suspend fun demoFunction1(channel1: ReceiveChannel<Int>,
                          channel2: ReceiveChannel<Int>): Int = coroutineScope { this: CoroutineScope
    var count = 0
    val mutex = Mutex()
    val job1 = launch { this: CoroutineScope
        for (i in channel1) mutex.withLock {
            count += i
        }
    }
    val job2 = launch { this: CoroutineScope
        for (i in channel2) mutex.withLock {
            count += i
        }
    }
    job1.join()
    job2.join()
    count ^coroutineScope
}
```

```kotlin
// 传统写法
suspend fun demoFunction1(channel1: ReceiveChannel<Int>,
                          channel2: ReceiveChannel<Int>): Int = coroutineScope { this: CoroutineScope
    /**
     * 优化，为每个 Channel 使用独立的变量计数，避免竞争。
     * 但是，当 Channel 的数量过多的时候，编写容易出错。
     */
    var count1 = 0
    var count2 = 0
    val job1 = launch { this: CoroutineScope
        for (i in channel1)
            count1 += i
    }
    val job2 = launch { this: CoroutineScope
        for (i in channel2)
            count2 += i
    }
    job1.join()
    job2.join()
    count1 + count2 ^coroutineScope
}
```

```kotlin
// 函数式写法
suspend fun demoFunction2(channel1: ReceiveChannel<Int>,
                          channel2: ReceiveChannel<Int>): Int = coroutineScope { this: CoroutineScope
    val countDeferred1 = async { this: CoroutineScope
        channel1 getValue 0
    }
    val countDeferred2 = async { this: CoroutineScope
        channel2 getValue 0
    }
    countDeferred1.await() + countDeferred2.await() ^coroutineScope
}

tailrec suspend infix fun ReceiveChannel<Int>.getValue(old: Int): Int =
    if (isClosedForReceive) old else getValue( old: old + receive())
```

```kotlin
// 我认为的比较好的写法
suspend fun demoFunction3(channel1: ReceiveChannel<Int>,
                          channel2: ReceiveChannel<Int>): Int = coroutineScope { this: CoroutineScope
    val countDeferred1 = async { this: CoroutineScope
        var count = 0
        for (i in channel1) // for 循环比递归更直观
            count += i // i 变量不向外暴露，因此函数纯度不会改变
        count ^async
    }
    val countDeferred2 = async { this: CoroutineScope
        var count = 0
        for (i in channel2)
            count += i
        count ^async
    }
    countDeferred1.await() + countDeferred2.await() ^coroutineScope
}
```

# 实战二：函数组合

```
data class Soy(private val weight: Float = 500f) {
    override fun toString(): String = "大豆"
}

data class SoyMilk(private val soy: Soy) {
    override fun toString(): String = "豆浆"
}

data class Tofu(private val soyMilk: SoyMilk) {
    override fun toString(): String = "豆腐"
}

data class StinkyTofu(private val tofu: Tofu) {
    override fun toString(): String = "臭豆腐"
}
```

```
// 研磨
fun grinding(soy: Soy): SoyMilk = SoyMilk(soy)

// 凝固
fun freeze(soyMilk: SoyMilk): Tofu = Tofu(soyMilk)

// 腌制
fun pickled(tofu: Tofu): StinkyTofu = StinkyTofu(tofu)
```

```kotlin
operator fun <T, R, S> ((T) -> R).plus(function: (R) -> S): (T) -> S =
    { x -> function(this(x)) }
```

```kotlin
infix fun <T, R> ((T) -> R).call(param: T): R = this(param)
```

```
fun buildTofu(soy: Soy): Tofu = ::grinding + ::freeze call soy

fun buildStinkTofu(soy: Soy): StinkyTofu = ::grinding + ::freeze + ::pickled call soy
```

```
fun main() = Soy().let { it: Soy
    println(buildTofu(it)) // 输出：豆腐
    println(buildStinkTofu(it)) // 输出：臭豆腐
}
```

```kotlin
fun buildTofu(soy: Soy): Tofu = freeze(grinding(soy))

// 如果流程函数过多，则会发生无数层括号嵌套
fun buildStinkTofu(soy: Soy): StinkyTofu = pickled(freeze(grinding(soy)))
```

```
/**
 * 要消除括号嵌套的话只能将每一个流程函数的结果保存下来，
 * 然后在下一行传递给下一个流程函数，但是同样，如果流程函数
 * 的数量过多（比如 10 个），我们就必须编写大量不优雅的样板式代码。
 */

fun buildTofu(soy: Soy): Tofu {
    val soyMilk = grinding(soy)
    return freeze(soyMilk)
}

fun buildStinkTofu(soy: Soy): StinkyTofu {
    val soyMilk = grinding(soy)
    val tofu = freeze(soyMilk)
    return pickled(tofu)
}
```

```
/**
 * 函数组合也方便我们在代码流程中按需组合出任意我们需要的函数，
 * 而不必每次都事先用 fun 关键字来进行定义
 */
fun main() = Soy().let { it: Soy
    val buildTofu = ::grinding + ::freeze
    val buildStinkTofu = ::grinding + ::freeze + ::pickled
    println(buildTofu call it) // 输出：豆腐
    println(buildStinkTofu call it) // 输出：臭豆腐
}
```

# 函数式编程的优势

- 代码更加简洁
- 不变性的思想会帮助我们规避不少资源竞争的场景
- 纯函数更加利于单元测试
- 函数组合让我们连续调用多个函数更加优雅

# 学习资源


中文官网：https://www.kotlincn.net

中文官方博客：https://www.kotliner.cn

微信公众号：Kotlin、Kotlin开发者联盟

简书、掘金、CSDN 等平台，搜索：Kotlin中文社区

# 个人联系方式

**微信：**

**Email:** qiaoyuang2012@gmail.com

**内推机会：** OKCOIN