Kotlin 循环控制 Kotlin 继承

Kotlin 类和对象

类定义

Kotlin 类可以包含：构造函数和初始化代码块、函数、属性、内部类、对象声明。

Kotlin 中使用关键字 `class` 声明类，后面紧跟类名：

```
class Runoob {  // 类名为 Runoob
    // 大括号内是类体构成
}
```

我们也可以定义一个空类：

```
class Empty
```

可以在类中定义成员函数：

```
class Runoob() {
    fun foo() { print("Foo") } // 成员函数
}
```

类的属性

属性定义

类的属性可以用关键字 `var` 声明为可变的，否则使用只读关键字 `val` 声明为不可变。

```
class Runoob {
    var name: String = ……
    var url: String = ……
    var city: String = ……
}
```

我们可以像使用普通函数那样使用构造函数创建类实例：

```
val site = Runoob() // Kotlin 中没有 new 关键字
```

要使用一个属性，只要用名称引用它即可

```
site.name           // 使用 . 号来引用
site.url
```

Koltin 中的类可以有一个 主构造器，以及一个或多个次构造器，主构造器是类头部的一部分，位于类名称之后:

```
class Person constructor(firstName: String) {}
```

如果主构造器没有任何注解，也没有任何可见度修饰符，那么constructor关键字可以省略。

```
class Person(firstName: String) {
}
```

getter 和 setter

属性声明的完整语法：

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

getter 和 setter 都是可选

如果属性类型可以从初始化语句或者类的成员函数中推断出来，那就可以省去类型，val不允许设置setter函数，因为它是只读的。

```
var allByDefault: Int? // 错误: 需要一个初始化语句, 默认实现了 getter 和 setter 方法
var initialized = 1    // 类型为 Int, 默认实现了 getter 和 setter
val simple: Int?       // 类型为 Int ，默认实现 getter ，但必须在构造函数中初始化
val inferredType = 1   // 类型为 Int 类型,默认实现 getter
```

实例

以下实例定义了一个 Person 类，包含两个可变变量 lastName 和 no，lastName 修改了 getter 方法，no 修改了 setter 方法。

```
class Person {

    var lastName: String = "zhang"
        get() = field.toUpperCase()   // 将变量赋值后转换为大写
        set

    var no: Int = 100
        get() = field                // 后端变量
        set(value) {
            if (value < 10) {       // 如果传入的值小于 10 返回该值
```

```
                field = value
            } else {
                field = -1         // 如果传入的值大于等于 10 返回 -1
            }
        }

    var heiht: Float = 145.4f
        private set
}

// 测试
fun main(args: Array<String>) {
    var person: Person = Person()

    person.lastName = "wang"

    println("lastName:${person.lastName}")

    person.no = 9
    println("no:${person.no}")

    person.no = 20
    println("no:${person.no}")

}
```

输出结果为：

```
lastName:WANG
no:9
no:-1
```

Kotlin 中类不能有字段。提供了 Backing Fields(后端变量) 机制,备用字段使用field关键字声明,field 关键词只能用于属性的访问器，如以上实例：

```
var no: Int = 100
        get() = field                // 后端变量
        set(value) {
            if (value < 10) {       // 如果传入的值小于 10 返回该值
                field = value
            } else {
                field = -1         // 如果传入的值大于等于 10 返回 -1
            }
        }
```

非空属性必须在定义的时候初始化,kotlin提供了一种可以延迟初始化的方案,使用 **`lateinit`** 关键字描述属性：

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // dereference directly
    }
}
```

主构造器

主构造器中不能包含任何代码，初始化代码可以放在初始化代码段中，初始化代码段使用 init 关键字作为前缀。

```
class Person constructor(firstName: String) {
    init {
        println("FirstName is $firstName")
    }
}
```

注意：主构造器的参数可以在初始化代码段中使用，也可以在类主体n定义的属性初始化代码中使用。 一种简洁语法，可以通过主构造器来定义属性并初始化属性值（可以是var或val）：

```
class People(val firstName: String, val lastName: String) {
    //...
}
```

如果构造器有注解，或者有可见度修饰符，这时constructor关键字是必须的，注解和修饰符要放在它之前。

实例

创建一个 Runoob类，并通过构造函数传入网站名：

```
class Runoob  constructor(name: String) {  // 类名为 Runoob
    // 大括号内是类体构成
    var url: String = "http://www.runoob.com"
    var country: String = "CN"
    var siteName = name

    init {
        println("初始化网站名: ${name}")
    }
```

```
    fun printTest() {
        println("我是类的函数")
    }
}

fun main(args: Array<String>) {
    val runoob =  Runoob("菜鸟教程")
    println(runoob.siteName)
    println(runoob.url)
    println(runoob.country)
    runoob.printTest()
}
```

输出结果为：

```
初始化网站名: 菜鸟教程
菜鸟教程
http://www.runoob.com
CN
我是类的函数
```

次构造函数

类也可以有二级构造函数，需要加前缀 constructor:

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

如果类有主构造函数，每个次构造函数都要，或直接或间接通过另一个次构造函数代理主构造函数。在同一个类中代理另一个构造函数使用 this 关键字：

```
class Person(val name: String) {
    constructor (name: String, age:Int) : this(name) {
        // 初始化...
    }
}
```

如果一个非抽象类没有声明构造函数(主构造函数或次构造函数)，它会产生一个没有参数的构造函数。构造函数是 public 。如果你不想你的类有公共的构造函数，你就得声明一个空的主构造函数：

```
class DontCreateMe private constructor () {
}
```

注意：在 JVM 虚拟机中，如果主构造函数的所有参数都有默认值，编译器会生成一个附加的无参的构造函数，这个构造函数会直接使用默认值。这使得 Kotlin 可以更简单的使用像 Jackson 或者 JPA 这样使用无参构造函数来创建类实例的库。

```
class Customer(val customerName: String = "")
```

实例

```
class Runoob  constructor(name: String) {  // 类名为 Runoob
    // 大括号内是类体构成
    var url: String = "http://www.runoob.com"
    var country: String = "CN"
    var siteName = name

    init {
        println("初始化网站名: ${name}")
    }
    // 次构造函数
    constructor (name: String, alexa: Int) : this(name) {
        println("Alexa 排名 $alexa")
    }

    fun printTest() {
        println("我是类的函数")
    }
}

fun main(args: Array<String>) {
    val runoob =  Runoob("菜鸟教程", 10000)
    println(runoob.siteName)
    println(runoob.url)
    println(runoob.country)
    runoob.printTest()
}
```

输出结果为：

```
初始化网站名: 菜鸟教程
Alexa 排名 10000
菜鸟教程
http://www.runoob.com
```

```
CN
我是类的函数
```

抽象类

抽象是面向对象编程的特征之一，类本身，或类中的部分成员，都可以声明为abstract的。抽象成员在类中不存在具体的实现。

注意：无需对抽象类或抽象成员标注open注解。

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base() {
    override abstract fun f()
}
```

嵌套类

我们可以把类嵌套在其他类中，看以下实例：

```
class Outer {                  // 外部类
    private val bar: Int = 1
    class Nested {             // 嵌套类
        fun foo() = 2
    }
}

fun main(args: Array<String>) {
    val demo = Outer.Nested().foo() // 调用格式：外部类.嵌套类.嵌套类方法/属性
    println(demo)    // == 2
}
```

内部类

内部类使用 inner 关键字来表示。

内部类会带有一个对外部类的对象的引用，所以内部类可以访问外部类成员属性和成员函数。

```
class Outer {
    private val bar: Int = 1
    var v = "成员属性"
    /**嵌套内部类**/
    inner class Inner {
        fun foo() = bar  // 访问外部类成员
```

```
        fun innerTest() {
            var o = this@Outer //获取外部类的成员变量
            println("内部类可以引用外部类的成员，例如：" + o.v)
        }
    }
}

fun main(args: Array<String>) {
    val demo = Outer().Inner().foo()
    println(demo) //   1
    val demo2 = Outer().Inner().innerTest()
    println(demo2)   // 内部类可以引用外部类的成员，例如：成员属性
}
```

为了消除歧义，要访问来自外部作用域的 this，我们使用this@label，其中 @label 是一个 代指 this 来源的标签。

匿名内部类

使用对象表达式来创建匿名内部类：

```
class Test {
    var v = "成员属性"

    fun setInterFace(test: TestInterFace) {
        test.test()
    }
}

/**
 * 定义接口
 */
interface TestInterFace {
    fun test()
}

fun main(args: Array<String>) {
    var test = Test()

    /**
     * 采用对象表达式来创建接口对象，即匿名内部类的实例。
     */
    test.setInterFace(object : TestInterFace {
        override fun test() {
            println("对象表达式创建匿名内部类的实例")
        }
    })
}
```

类的修饰符

类的修饰符包括 classModifier 和_accessModifier_:

- classModifier: 类属性修饰符，标示类本身特性。

```
abstract    // 抽象类
final       // 类不可继承，默认属性
enum        // 枚举类
open        // 类可继承，类默认是final的
annotation  // 注解类
```

- accessModifier: 访问权限修饰符

```
private    // 仅在同一个文件中可见
protected  // 同一个文件中或子类可见
public     // 所有调用的地方都可见
internal   // 同一个模块中可见
```

实例

```
// 文件名：example.kt
package foo

private fun foo() {} // 在 example.kt 内可见

public var bar: Int = 5 // 该属性随处可见

internal val baz = 6    // 相同模块内可见
```

← Kotlin 循环控制 Kotlin 继承 →

也就是说，在 Kotlin 中，任何时候当你写出“一个变量后边加等于号”这种形式的时候，比如我们定义 `var no: Int` 变量，当你写出 `no = ...` 这种形式的时候，这个等于号都会被编译器翻译成调用 **setter** 方法；而同样，在任何位置引用变量时，只要出现 **no** 变量的地方都会被编译器翻译成 **getter** 方法。那么问题就来了，当你在 **setter** 方法内部写出 `no = ...` 时，相当于在 **setter** 方法中调用 **setter** 方法，形成递归，进而形成死循环，例如文中的例子：

```
var no: Int = 100
    get() = field                // 后端变量
    set(value) {
        if (value < 10) {       // 如果传入的值小于 10 返回该值
            field = value
        } else {
            field = -1         // 如果传入的值大于等于 10 返回 -1
        }
    }
```

这段代码按以上这种写法是正确的，因为使用了 **field** 关键字，但是如果不用 **field** 关键字会怎么样呢？例如：

```
var no: Int = 100
    get() = no
    set(value) {
        if (value < 10) {       // 如果传入的值小于 10 返回该值
            no = value
        } else {
            no = -1         // 如果传入的值大于等于 10 返回 -1
        }
    }
```

注意这里我们使用的 Java 的思维写了 **getter** 和 **setter** 方法，那么这时，如果将这段代码翻译成 Java 代码会是怎么样呢？如下：

```
int no = 100;
public int getNo() {
    return getNo();// Kotlin中的get() = no语句中出来了变量no，直接被编译器理解成“调用getter方法”
}

public void setNo(int value) {
    if (value < 10) {
        setNo(value);// Kotlin中出现“no =”这样的字样，直接被编译器理解成“这里要调用setter方法”
    } else {
        setNo(-1);// 在setter方法中调用setter方法，这是不正确的
    }
}
```

翻译成 Java 代码之后就很直观了，在 **getter** 方法和 **setter** 方法中都形成了递归调用，显然是不正确的，最终程序会出现内存溢出而异常终止。

2、嵌套类和内部类在使用时的区别

（1）创建对象的区别

```
var demo = Outter.Nested()// 嵌套类，Outter后边没有括号
var demo = Outter().Inner();// 内部类，Outter后边有括号
```

也就是说，要想构造内部类的对象，必须先构造外部类的对象，而嵌套类则不需要；

（2）引用外部类的成员变量的方式不同

先来看嵌套类：

```
class Outer {                   // 外部类
    private val bar: Int = 1
    class Nested {              // 嵌套类
        var ot: Outer = Outer()
        println(ot.bar) // 嵌套类可以引用外部类私有变量，但要先创建外部类的实例，不能直接引用
        fun foo() = 2
    }
}
```

再来看一下内部类（引用文章中代码）：

```
class Outer {
    private val bar: Int = 1
    var v = "成员属性"
    /**嵌套内部类**/
    inner class Inner {
        fun foo() = bar  // 访问外部类成员
        fun innerTest() {
            var o = this@Outer //获取外部类的成员变量
            println("内部类可以引用外部类的成员，例如: " + o.v)
        }
    }
}
```

可以看来内部类可以直接通过 **this@** 外部类名 的形式引用外部类的成员变量，不需要创建外部类对象；

3、匿名内部类的实现

引用文章中的代码

```
fun main(args: Array<String>) {
    var test = Test()

    /**
     * 采用对象表达式来创建接口对象，即匿名内部类的实例。
     */
    test.setInterFace(object : TestInterFace {
        override fun test() {
            println("对象表达式创建匿名内部类的实例")
        }
    })
}
```

特别注意这里的 `object : TestInterFace`，这个 object 是 Kotlin 的关键字，要实现匿名内部类，就必须使用 object 关键字，不能随意替换其它单词，切记切记。

applixy 8个月前 (07-26)

关于 field 我也分享一下理解。

```
// 还是 JAVA 代码
int no = 100;
private int _no = 100;
public int getNo() {
    return _no;//
}

public void setNo(int value) {
    if (value < 10) {
        _no = value;// Kotlin中出现“no =”这样的字样，直接被编译器理解成“这里要调用setter方法”
    } else {
        _no = -1;// 在setter方法中调用setter方法，这是不正确的
    }
}
```

上面这样就没有问题, 而 field 就相当于编译器给你提供两一个隐式私有变量。

Cyandnow 4个月前 (11-22)