Kotlin之let,apply,run,with等函数区别2

流水不腐小夏 关注

2 2018.01.04 20:17:57 字数 780 阅读 19,221

Kotlin之let,apply,run,with等函数区别2

以前也总结过Kotlin的一些内置函数let,apply,run,with的区别——地址，后面又增加了also,takeIf,takeUnless等函数，所以这里重新总结下，然后介绍下使用场景。

前提介绍

Kotlin和Groovy等语言一样，支持闭包(block)，如果函数中最后一个参数为闭包，那么最后一个参可以不写在括号中，而写在括号后面，如果只有一个参数，括号也可以去掉。

如下所示

```
fun toast() {
    button.setOnClickListener({
       Toast.makeText(context, "test", Toast.LENGTH_SHORT).show()
    })
}

fun toast() {
    button.setOnClickListener {
        Toast.makeText(context, "test", Toast.LENGTH_SHORT).show()
    }
}
```

后面介绍的几个函数都是这样的，这样就很容理解。

repeat

repeat函数是一个单独的函数，定义如下。

```
/**
 * Executes the given function [action] specified number of [times].
 *
 * A zero-based index of current iteration is passed as a parameter to [action].
```

```
 */
@kotlin.internal.InlineOnly
public inline fun repeat(times: Int, action: (Int) -> Unit) {
    contract { callsInPlace(action) }

    for (index in 0..times - 1) {
        action(index)
    }
}
```

通过代码很容易理解，就是循环执行多少次block中内容。

如

```
fun main(args: Array<String>) {
    repeat(3) {
        println("Hello world")
    }
}

```

运行结果

```
Hello world
Hello world
Hello world
```

with

with函数也是一个单独的函数，并不是Kotlin中的extension，指定的T作为闭包的receiver，使用参数中闭包的返回结果

```
/**
 * Calls the specified function [block] with the given [receiver] as its receiver and
 */
@kotlin.internal.InlineOnly
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return receiver.block()
}
```

代码示例:

```
fun testWith() {
    // fun <T, R> with(receiver: T, f: T.() -> R): R = receiver.f()
    with(ArrayList<String>()) {
        add("testWith")
        add("testWith")
        add("testWith")
        println("this = " + this)
    }.let { println(it) }
}
// 运行结果
// this = [testWith, testWith, testWith]
// kotlin.Unit
```

class文件

```
 public static final void testWith()
  {
    Object localObject = new ArrayList();
    ArrayList localArrayList1 = (ArrayList)localObject;
    int $i$a$1$with;
    ArrayList $receiver;
    $receiver.add("testWith");
    $receiver.add("testWith");
    $receiver.add("testWith");
    String str = "this = " + $receiver;
```

```
    System.out.println(str);
    localObject = Unit.INSTANCE;
    Unit it = (Unit)localObject;
    int $i$a$2$let;
    System.out.println(it);
  }
```

let

首先let()的定义是这样的，默认当前这个对象作为闭包的it参数，返回值是函数里面最后一行，或者指定return

```
fun <T, R> T.let(f: (T) -> R): R = f(this)
```

简单示例:

```
fun testLet(): Int {
    // fun <T, R> T.let(f: (T) -> R): R { f(this)}
    "testLet".let {
        println(it)
        println(it)
        println(it)
        return 1
    }
}
//运行结果
//testLet
//testLet
//testLet
```

可以看看最后生成的class文件，代码已经经过格式化了，编译器只是在我们原先的变量后面添加了let里面的内容。

```
public static final int testLet() {
    String str1 = "testLet";
    String it = (String)str1;
    int $i$a$1$let;
    System.out.println(it);
    System.out.println(it);
    System.out.println(it);
    return 1;
}
```

来个复杂一定的例子

```
fun testLet(): Int {
    // fun <T, R> T.let(f: (T) -> R): R { f(this)}
    "testLet".let {
        if (Random().nextBoolean()) {
            println(it)
            return 1
        } else {
            println(it)
            return 2
        }
    }
}
```

编译过后的class文件

```
public static final int testLet() {
    String str1 = "testLet";
    String it = (String)str1;
    int $i$a$1$let;
    if (new Random().nextBoolean())
    {
        System.out.println(it);
        return 1;
    }
    System.out.println(it);
```

```
    return 2;
}
```

apply

apply函数是这样的，调用某对象的apply函数，在函数范围内，可以任意调用该对象的任意方法，并返回该对象

```
/**
 * Calls the specified function [block] with `this` value as its receiver and returns
 */
@kotlin.internal.InlineOnly
public inline fun <T> T.apply(block: T.() -> Unit): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    block()
    return this
}
```

代码示例

```
fun testApply() {
    // fun <T> T.apply(f: T.() -> Unit): T { f(); return this }
    ArrayList<String>().apply {
        add("testApply")
        add("testApply")
        add("testApply")
        println("this = " + this)
    }.let { println(it) }
}

// 运行结果
// this = [testApply, testApply, testApply]
// [testApply, testApply, testApply]
```

编译过后的class文件

```
  public static final void testApply()
  {
    ArrayList localArrayList1 = new ArrayList();
    ArrayList localArrayList2 = (ArrayList)localArrayList1;
    int $i$a$1$apply;
    ArrayList $receiver;
    $receiver.add("testApply");
    $receiver.add("testApply");
    $receiver.add("testApply");
    String str = "this = " + $receiver;
    System.out.println(str);
    localArrayList1 = localArrayList1;
    ArrayList it = (ArrayList)localArrayList1;
    int $i$a$2$let;
    System.out.println(it);
  }
```

run

run函数和apply函数很像，只不过run函数是使用最后一行的返回，apply返回当前自己的对象。

```
/**
 * Calls the specified function [block] with `this` value as its receiver and returns
 */
@kotlin.internal.InlineOnly
public inline fun <T, R> T.run(block: T.() -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block()
}
```

```
fun <T, R> T.run(f: T.() -> R): R = f()
```

代码示例

```
fun testRun() {
    // fun <T, R> T.run(f: T.() -> R): R = f()
    "testRun".run {
        println("this = " + this)
    }.let { println(it) }
}
// 运行结果
// this = testRun
// kotlin.Unit
```

class文件

```
  public static final void testRun()
  {
    Object localObject = "testRun";
    String str1 = (String)localObject;
    int $i$a$1$run;
    String $receiver;
    String str2 = "this = " + $receiver;
    System.out.println(str2);
    localObject = Unit.INSTANCE;
    Unit it = (Unit)localObject;
    int $i$a$2$let;
    System.out.println(it);
  }
```

另一个Run

还有个run函数，不是extension，它的定义如下，执行block，返回block的返回

```
/**
 * Calls the specified function [block] and returns its result.
 */
@kotlin.internal.InlineOnly
public inline fun <R> run(block: () -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block()
}
```

示例

```
fun main(args: Array<String>) {
    val date = run {
        Date()
    }

    println("date = $date")
}
// 运行结果
// date = Thu Jan 04 19:31:09 CST 2018
```

also

执行block，返回this，

```
/**
 * Calls the specified function [block] with `this` value as its argument and returns
 */
@kotlin.internal.InlineOnly
@SinceKotlin("1.1")
public inline fun <T> T.also(block: (T) -> Unit): T {
    contract {
```

```
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    block(this)
    return this
}
```

示例：

```
fun main(args: Array<String>) {
    val also = Date().also {
        println("in also time = " + it.time)
    }

    println("also = $also")
}
```

运行结果

```
in also time = 1515065830740
also = Thu Jan 04 19:37:10 CST 2018
```

takeIf

满足block中条件，则返回当前值，否则返回null，block的返回值Boolean类型

```
/**
 * Returns `this` value if it satisfies the given [predicate] or `null`, if it doesn't
 */
@kotlin.internal.InlineOnly
@SinceKotlin("1.1")
public inline fun <T> T.takeIf(predicate: (T) -> Boolean): T? {
    contract {
        callsInPlace(predicate, InvocationKind.EXACTLY_ONCE)
    }
    return if (predicate(this)) this else null
}
```

示例

```
fun main(args: Array<String>) {
    val date = Date().takeIf {
        // 是否在2018年元旦后
        it.after(Date(2018 - 1900, 0, 1))
    }

    println("date = $date")
}

// 运行结果
// date = Thu Jan 04 19:42:09 CST 2018
```

takeUnless

和takeIf相反，如不满足block中的条件，则返回当前对象，否则为null

```
/**
 * Returns `this` value if it _does not_ satisfy the given [predicate] or `null`, if i
 */
@kotlin.internal.InlineOnly
@SinceKotlin("1.1")
public inline fun <T> T.takeUnless(predicate: (T) -> Boolean): T? {
    contract {
        callsInPlace(predicate, InvocationKind.EXACTLY_ONCE)
    }
    return if (!predicate(this)) this else null
}
```

示例

```
fun main(args: Array<String>) {
    val date = Date().takeUnless {
        // 是否在2018年元旦后
        it.after(Date(2018 - 1900, 0, 1))
    }

    println("date = $date")
}
// 运行结果
// date = null
```

总结

怎么样，是不是看晕了，没关系，我们来总结下。

函数名	定义	block参数	闭包返回返回值	函数返回值	extension	其他
repeat	fun repeat(times: Int, action: (Int) -> Unit)	无	Unit	Unit	否	普通函数
with	fun <T, R> with(receiver: T, f: T.() -> R): R = receiver.f()	无，可以使用this	Any	闭包返回	否	普通函数
run	<R> run(block: () -> R): R	无	Any	闭包返回	否	普通函数
let	fun <T, R> T.let(f: (T) -> R): R	it	Any	闭包返回	是	
apply	fun <T> T.apply(f: T.() -> Unit): T	无，可以使用this	Unit	this	是	
run	fun <T, R> T.run(f: T.() -> R): R	无，可以使用this	Any	闭包返回	是	
also	fun <T> T.also(block: (T) -> Unit): T	it	Unit	this	是	
takeIF	fun <T> T.takeIf(predicate: (T) -> Boolean): T?	it	Boolean	this或null	是	闭包返回类型必须是Boolean
takeUnless	fun <T> T.takeUnless(predicate: (T) -> Boolean): T?	it	Boolean	this或null	是	闭包返回类型必须是Boolean

示例

上面就是本人所理解的，最后再给个整体示例。

定义一个结构体

```
class User {
    var id: Int = 0
    var name: String? = null
    var hobbies: List<String>? = null

    override fun toString(): String {
        return "User(id=$id, name=$name, hobbies=$hobbies)"
    }
}
```

普通的赋值语句这样就可以了

```
var user = User()
user.id = 1
user.name = "test1"
user.hobbies = listOf("aa", "bb", "cc")
println("user = $user")
```

如果使用let,apply,run,with可以这样，let和also是需要it的，其他的默认使用this。

```
user.let {
    it.id = 2
    it.name = "test2"
    it.hobbies = listOf("aa", "bb", "cc")
}
println("user = $user")

user.also {
    it.id = 3
    it.name = "test3"
    it.hobbies = listOf("aa", "bb", "cc")
}
println("user = $user")

user.apply {
    id = 2
    name = "test2"
    hobbies = listOf("aa", "bb", "cc")
    Date()
}
println("user = $user")

user.run {
    id = 3
    name = "test3"
    hobbies = listOf("aa", "bb", "cc")
    Date()
}
println("user = $user")

with(user) {
    id = 4
    name = "test4"
    hobbies = listOf("aa", "bb", "cc")
    Date()
}
println("user = $user")
```

再举一个例子，一个http的response结构体。

```
class Resp<T> {
    var code: Int = 0
    var body: T? = null
    var errorMessage: String? = null

    fun isSuccess(): Boolean = code == 200

    override fun toString(): String {
        return "Resp(code=$code, body=$body, errorMessage=$errorMessage)"
    }
}
```

在处理网络数据的时候，需要各种判断，比如。

```
fun main(args: Array<String>) {
    var resp: Resp<String>? = Resp()

    if (resp != null) {
        if (resp.isSuccess()) {
            // do success
            println(resp.body)
        } else {
            // do fail
            println(resp.errorMessage)
        }
    }
}
```

当然也可以用apply,let,run等函数。

```
fun main(args: Array<String>) {
    var resp: Resp<String>? = Resp()

//    if (resp != null) {
//        if (resp.isSuccess()) {
//            // do success
//            println(resp.body)
//        } else {
//            println(resp.errorMessage)
//        }
//    }

    resp?.run {
        if (isSuccess()) {
            // do success
            println(resp.body)
        } else {
            println(resp.errorMessage)
        }
    }

    resp?.apply {
        if (isSuccess()) {
            // do success
            println(resp.body)
        } else {
            println(resp.errorMessage)
        }
    }

    resp?.let {
        if (it.isSuccess()) {
            // do success
            println(it.body)
        } else {
            println(it.errorMessage)
        }
    }

    resp?.also {
        if (it.isSuccess()) {
            // do success
            println(it.body)
        } else {
            println(it.errorMessage)
        }
    }
}
```