Kotlin系列之let、with、run、apply、also函数的使用

## 一、回调函数的Kotin的lambda的简化

在Kotlin中对Java中的一些的接口的回调做了一些优化，可以使用一个lambda函数来代替。可以简化写一些不必要的嵌套回调方法。但是需要注意:在lambda表达式，只支持单抽象方法模型，也就是说设计的接口里面只有一个抽象的方法，才符合lambda表达式的规则，多个回调方法不支持。

1、用Java代码实现一个接口的回调。

mView.setEventListener(new ExamPlanHomeEventListener(){

public void onSuccess(Data data){

//todo

}

});

2、在Kotlin中的实现一个接口的回调,不使用lambda表达式(这种方式非常适用于kotlin中对于一个接口中含有多个回调方法)。

mView.setEventListener(object: ExamPlanHomeEventListener{

public void onSuccess(Data data){

//todo

}

});

3、如果在Kotlin中的对于接口只有一个回调的方法，就符合使用lambda函数，我们可以把以上代码简化成这样。

mView.setEventListener({

data: Data ->

//todo

}

//或者可以直接省略Data,借助kotlin的智能类型推导

mView.setEventListener({

data ->

//todo

})

4、如果以上代码中的data参数没有使用到的话，可以直接把data去掉

mView.setEventListener({

//todo

})

5、以上代码还可以做个调整，由于setEventListener函数最后一个参数是一个函数的话，可以直接把括号的实现提到圆括号外面

mView.setEventListener(){

//todo

}

6、由于setEventListener这个函数只有一个参数，可以直接省略圆括号

mView.setEventListener{

//todo

}

## 二、内联扩展函数之let

let扩展函数的实际上是一个作用域函数，当你需要去定义一个变量在一个特定的作用域范围内，let函数的是一个不错的选择；let函数另一个作用就是可以避免写一些判断null的操作。

1、let函数的使用的一般结构

object.let{

it.todo()//在函数体内使用it替代object对象去访问其公有的属性和方法

...

}

//另一种用途 判断object为null的操作

object?.let{//表示object不为null的条件下，才会去执行let函数体

it.todo()

}

2、let函数底层的inline扩展函数+lambda结构

@kotlin.internal.InlineOnly

public inline fun <T, R> T.let(block: (T) -> R): R = block(this)

3、let函数inline结构的分析

从源码let函数的结构来看它是只有一个lambda函数块block作为参数的函数,调用T类型对象的let函数，则该对象为函数的参数。在函数块内可以通过 it 指代该对象。返回值为函数块的最后一行或指定return表达式。

4、let函数的kotlin和Java转化

//kotlin

fun main(args: Array<String>) {

val result = "testLet".let {

println(it.length)

1000

}

println(result)

}

//java

public final class LetFunctionKt {

public static final void main(@NotNull String[] args) {

Intrinsics.checkParameterIsNotNull(args, "args");

String var2 = "testLet";

int var4 = var2.length();

System.out.println(var4);

int result = 1000;

System.out.println(result);

}

}

5、let函数适用的场景

场景一: 最常用的场景就是使用let函数处理需要针对一个可null的对象统一做判空处理。

场景二: 然后就是需要去明确一个变量所处特定的作用域范围内可以使用

6、let函数使用前后的对比

没有使用let函数的代码是这样的，看起来不够优雅

mVideoPlayer?.setVideoView(activity.course\_video\_view)

mVideoPlayer?.setControllerView(activity.course\_video\_controller\_view)

mVideoPlayer?.setCurtainView(activity.course\_video\_curtain\_view)

使用let函数后的代码是这样的

mVideoPlayer?.let {

it.setVideoView(activity.course\_video\_view)

it.setControllerView(activity.course\_video\_controller\_view)

it.setCurtainView(activity.course\_video\_curtain\_view)

}

## 三、内联函数之with

1、with函数使用的一般结构

with(object){

//todo

}

2、with函数底层的inline扩展函数+lambda结构

@kotlin.internal.InlineOnly

public inline fun <T, R> with(receiver: T, block: T.() -> R): R = receiver.block()

3、with函数inline结构的分析

with函数和前面的几个函数使用方式略有不同，因为它不是以扩展的形式存在的。它是将某对象作为函数的参数，在函数块内可以通过 this 指代该对象。返回值为函数块的最后一行或指定return表达式。

可以看出with函数是接收了两个参数，分别为T类型的对象receiver和一个lambda函数块，所以with函数最原始样子如下:

val result = with(user, {

println("my name is $name, I am $age years old, my phone number is $phoneNum")

1000

})

但是由于with函数最后一个参数是一个函数，可以把函数提到圆括号的外部，所以最终with函数的调用形式如下:

val result = with(user) {

println("my name is $name, I am $age years old, my phone number is $phoneNum")

1000

}

4、with函数的kotlin和Java转化

//kotlin

fun main(args: Array<String>) {

val user = User("Kotlin", 1, "1111111")

val result = with(user) {

println("my name is $name, I am $age years old, my phone number is $phoneNum")

1000

}

println("result: $result")

}

//java

public static final void main(@NotNull String[] args) {

Intrinsics.checkParameterIsNotNull(args, "args");

User user = new User("Kotlin", 1, "1111111");

String var4 = "my name is " + user.getName() + ", I am " + user.getAge() + " years old, my phone number is " + user.getPhoneNum();

System.out.println(var4);

int result = 1000;

String var3 = "result: " + result;

System.out.println(var3);

}

5、with函数的适用的场景

适用于调用同一个类的多个方法时，可以省去类名重复，直接调用类的方法即可，经常用于Android中RecyclerView中onBinderViewHolder中，数据model的属性映射到UI上

6、with函数使用前后的对比

没有使用kotlin中的实现

@Override

public void onBindViewHolder(ViewHolder holder, int position) {

ArticleSnippet item = getItem(position);

if (item == null) {

return;

}

holder.tvNewsTitle.setText(StringUtils.trimToEmpty(item.titleEn));

holder.tvNewsSummary.setText(StringUtils.trimToEmpty(item.summary));

String gradeInfo = "难度：" + item.gradeInfo;

String wordCount = "单词数：" + item.length;

String reviewNum = "读后感：" + item.numReviews;

String extraInfo = gradeInfo + " | " + wordCount + " | " + reviewNum;

holder.tvExtraInfo.setText(extraInfo);

...

}

kotlin的实现

override fun onBindViewHolder(holder: ViewHolder, position: Int){

val item = getItem(position)?: return

with(item){

holder.tvNewsTitle.text = StringUtils.trimToEmpty(titleEn)

holder.tvNewsSummary.text = StringUtils.trimToEmpty(summary)

holder.tvExtraInf.text = "难度：$gradeInfo | 单词数：$length | 读后感: $numReviews"

...

}

}

## 四、内联扩展函数之run

1、run函数使用的一般结构

object.run{

//todo

}

2、run函数的inline+lambda结构

@kotlin.internal.InlineOnly

public inline fun <T, R> T.run(block: T.() -> R): R = block()

3、run函数的inline结构分析

run函数实际上可以说是let和with两个函数的结合体，run函数只接收一个lambda函数为参数，以闭包形式返回，返回值为最后一行的值或者指定的return的表达式。

4、run函数的kotlin和Java转化

//kotlin

fun main(args: Array<String>) {

val user = User("Kotlin", 1, "1111111")

val result = user.run {

println("my name is $name, I am $age years old, my phone number is $phoneNum")

1000

}

println("result: $result")

}

//java

public static final void main(@NotNull String[] args) {

Intrinsics.checkParameterIsNotNull(args, "args");

User user = new User("Kotlin", 1, "1111111");

String var5 = "my name is " + user.getName() + ", I am " + user.getAge() + " years old, my phone number is " + user.getPhoneNum();

System.out.println(var5);

int result = 1000;

String var3 = "result: " + result;

System.out.println(var3);

}

5、run函数的适用场景

适用于let,with函数任何场景。因为run函数是let,with两个函数结合体，准确来说它弥补了let函数在函数体内必须使用it参数替代对象，在run函数中可以像with函数一样可以省略，直接访问实例的公有属性和方法，另一方面它弥补了with函数传入对象判空问题，在run函数中可以像let函数一样做判空处理

6、run函数使用前后的对比

还是借助上个例子kotlin代码

override fun onBindViewHolder(holder: ViewHolder, position: Int){

val item = getItem(position)?: return

with(item){

holder.tvNewsTitle.text = StringUtils.trimToEmpty(titleEn)

holder.tvNewsSummary.text = StringUtils.trimToEmpty(summary)

holder.tvExtraInf = "难度：$gradeInfo | 单词数：$length | 读后感: $numReviews"

...

}

}

使用run函数后的优化

override fun onBindViewHolder(holder: ViewHolder, position: Int){

getItem(position)?.run{

holder.tvNewsTitle.text = StringUtils.trimToEmpty(titleEn)

holder.tvNewsSummary.text = StringUtils.trimToEmpty(summary)

holder.tvExtraInf = "难度：$gradeInfo | 单词数：$length | 读后感: $numReviews"

...

}

}

## 五、内联扩展函数之apply

1、apply函数使用的一般结构

object.apply{

//todo

}

2、apply函数的inline+lambda结构

@kotlin.internal.InlineOnly

public inline fun <T> T.apply(block: T.() -> Unit): T { block(); return this }

3、apply函数的inline结构分析

从结构上来看apply函数和run函数很像，唯一不同点就是它们各自返回的值不一样，run函数是以闭包形式返回最后一行代码的值，而apply函数的返回的是传入对象的本身。

4、apply函数的kotlin和Java转化

//kotlin

fun main(args: Array<String>) {

val user = User("Kotlin", 1, "1111111")

val result = user.apply {

println("my name is $name, I am $age years old, my phone number is $phoneNum")

1000

}

println("result: $result")

}

//java

public final class ApplyFunctionKt {

public static final void main(@NotNull String[] args) {

Intrinsics.checkParameterIsNotNull(args, "args");

User user = new User("Kotlin", 1, "1111111");

String var5 = "my name is " + user.getName() + ", I am " + user.getAge() + " years old, my phone number is " + user.getPhoneNum();

System.out.println(var5);

String var3 = "result: " + user;

System.out.println(var3);

}

}

5、apply函数的适用场景

整体作用功能和run函数很像，唯一不同点就是它返回的值是对象本身，而run函数是一个闭包形式返回，返回的是最后一行的值。正是基于这一点差异它的适用场景稍微与run函数有点不一样。apply一般用于一个对象实例初始化的时候，需要对对象中的属性进行赋值。或者动态inflate出一个XML的View的时候需要给View绑定数据也会用到，这种情景非常常见。特别是在我们开发中会有一些数据model向View model转化实例化的过程中需要用到。

6、apply函数使用前后的对比

没有使用apply函数的代码是这样的，看起来不够优雅

mSheetDialogView = View.inflate(activity, R.layout.biz\_exam\_plan\_layout\_sheet\_inner, null)

mSheetDialogView.course\_comment\_tv\_label.paint.isFakeBoldText = true

mSheetDialogView.course\_comment\_tv\_score.paint.isFakeBoldText = true

mSheetDialogView.course\_comment\_tv\_cancel.paint.isFakeBoldText = true

mSheetDialogView.course\_comment\_tv\_confirm.paint.isFakeBoldText = true

mSheetDialogView.course\_comment\_seek\_bar.max = 10

mSheetDialogView.course\_comment\_seek\_bar.progress = 0

使用apply函数后的代码是这样的

```

mSheetDialogView = View.inflate(activity, R.layout.biz\_exam\_plan\_layout\_sheet\_inner, null).apply{

course\_comment\_tv\_label.paint.isFakeBoldText = true

course\_comment\_tv\_score.paint.isFakeBoldText = true

course\_comment\_tv\_cancel.paint.isFakeBoldText = true

course\_comment\_tv\_confirm.paint.isFakeBoldText = true

course\_comment\_seek\_bar.max = 10

course\_comment\_seek\_bar.progress = 0

}

多层级判空问题

if (mSectionMetaData == null || mSectionMetaData.questionnaire == null || mSectionMetaData.section == null) {

return;

}

if (mSectionMetaData.questionnaire.userProject != null) {

renderAnalysis();

return;

}

if (mSectionMetaData.section != null && !mSectionMetaData.section.sectionArticles.isEmpty()) {

fetchQuestionData();

return;

}

```

kotlin的apply函数优化

mSectionMetaData?.apply{

//mSectionMetaData不为空的时候操作mSectionMetaData

}?.questionnaire?.apply{

//questionnaire不为空的时候操作questionnaire

}?.section?.apply{

//section不为空的时候操作section

}?.sectionArticle?.apply{

//sectionArticle不为空的时候操作sectionArticle

}

## 六、内联扩展函数之also

1、also函数使用的一般结构

object.also{

//todo

}

2、also函数的inline+lambda结构

@kotlin.internal.InlineOnly

1

@SinceKotlin(“1.1”)

public inline fun T.also(block: (T) -> Unit): T { block(this); return this }

3、also函数的inline结构分析

also函数的结构实际上和let很像唯一的区别就是返回值的不一样，let是以闭包的形式返回，返回函数体内最后一行的值，如果最后一行为空就返回一个Unit类型的默认值。而also函数返回的则是传入对象的本身

4、also函数编译后的class文件

//kotlin

fun main(args: Array<String>) {

val result = "testLet".also {

println(it.length)

1000

}

println(result)

}

//java

public final class AlsoFunctionKt {

public static final void main(@NotNull String[] args) {

Intrinsics.checkParameterIsNotNull(args, "args");

String var2 = "testLet";

int var4 = var2.length();

System.out.println(var4);

System.out.println(var2);

}

}

5、also函数的适用场景

适用于let函数的任何场景，also函数和let很像，只是唯一的不同点就是let函数最后的返回值是最后一行的返回值而also函数的返回值是返回当前的这个对象。一般可用于多个扩展函数链式调用

6、also函数使用前后的对比

和let函数类似

七、let,with,run,apply,also函数区别



原文链接：https://blog.csdn.net/u013064109/article/details/78786646