第八章 Kotlin 核心面试题汇总

8.1 Kotlin内置标准函数let的原理是什么？

本题在[享学Android高级架构师体系]Kotlin专题中有详细讲解

这道题想考察什么？

1. 是否了解Kotlin内置标准函数let的原理是什么与真实场景使用，是否熟悉Kotlin内置标准函数let的原理是什么本质？

考察的知识点

1. Kotlin内置标准函数let的原理是什么的概念在项目中使用与基本知识

考生应该如何回答

1.你工作这么些年，let内置标准函数一般用的很频繁吧，let的原理是什么？

答：

> 使用端的感受：

1.在使用的时候，任何的类型，都可以.let出来使用，这是为什么呢？ 因为标准let内置函数内部对泛型进行了let函数扩展，意味着所有的类型都等于泛型，所以任何地方都是可以使用let函数的。

2.所有类型.let {} 其实是一个匿名的Lambda表达式，Lambda表达式的特点是，最后一行会自动被认为是返回值类型，所以在表达式返回Boolean，那么当前的let函数就是Boolean类型，以此类推。

```
fun main() {
    val r1 = "Derry".let {
        true
        it.length
    }
    println(r1)

    val r2 = 123.let {
        999
        "【${it}】"
    }
    println(r2)
}
```

> 根据上面分析的两点使用感受，来分析他的原理：
>
> 1.inline : 是因为函数有lambda表达式，属于高阶函数，高阶函数规范来说要加inline
>
> 2.<T, R> T.let : T代表是要为T而扩展出一个函数名let(任何类型都可以 万能类型.let)， R代表是Lambda表达式最后一行返回的类型
>
> 3.block: (T) -> R : Lambda表达式名称block 输入参数是T本身 输出参数是R 也就是表达式最后一行返回推断的类型
>
> 4.: R { : R代表是Lambda表达式最后一行返回的类型，若表达式返回类型是Boolean, 那么这整个let函数的返回类型就是Boolean

```
// inline :           是因为函数有lambda表达式，属于高阶函数，高阶函数规范来说要加inline
// <T, R>  T.let :   T代表是要为T而扩展出一个函数名let(任何类型都可以 万能类型.let)， R代表是Lambda表达式最后一行返回的类型
// block: (T) -> R : Lambda表达式名称block 输入参数是T本身  输出参数是R 也就是表达式最后一行返回推断的类型
// : R {           : R代表是Lambda表达式最后一行返回的类型，若表达式返回类型是Boolean, 那么这整个let函数的返回类型就是Boolean
inline fun <T, R> T.let(block: (T) -> R): R {
    println("你${this}.let在${System.currentTimeMillis()}这个时间点调用了我")

    /*contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }*/
    // 调用Lambda表达式
    // 输入参数this == T == "Derry" / 123,
    // 输出参数：用户返回String类型，就全部是返回String类型
    return block(this)
}
```

> 总结：Kotlin内置标准let函数，运用了 高阶函数特性与Lambda，控制环节交给用户完成，用户在自己的Lambda表达式中，若返回Boolean，整个let函数 与 Lambda返回 都全部是Boolean
>
> 为了保证所有的类型都能正常使用let，给泛型增加了扩展函数let，所以所有的地方都可以使用let函数。

8.2 Kotlin语言的run高阶函数的原理是什么？

本题在[享学Android高级架构师体系]Kotlin专题中有详细讲解

这道题想考察什么？

1. 是否了解Kotlin语言的run高阶函数的原理是什么与真实场景使用，是否熟悉Kotlin语言的run高阶函数的原理是什么本质？

考察的知识点

1. Kotlin语言的run高阶函数的原理是什么的概念在项目中使用与基本知识

考生应该如何回答

1.你工作这么些年，Kotlin语言提供的高阶run函数一般用的很频繁吧，run的原理是什么？

答：

> run在Kotlin语法中使用端的感受：

1.在使用的时候，任何的类型，都可以.run出来使用，这是为什么呢？ 因为标准run内置函数内部对泛型进行run函数扩展，意味着所有的类型都等于泛型，所以任何地方都是可以使用run函数的。

2.所有类型.run{} 其实是一个匿名的Lambda表达式，Lambda表达式的特点是，最后一行会自动被认为是返回值类型，例如在表达式返回Boolean，那么当前的run函数就是Boolean类型，例如在表达式返回Int类型，那么当前的run函数就是Int类型，以此类推。

```
fun main() {
    val r1 : Int = "Derry".run {
        true
        length
    }
    println(r1)

    val r2 : String = 123.run {
        999
        "【${it}】"
    }
    println(r2)
}
```

> 根据上面分析的两点使用感受，来分析他的原理：
>
> 1.inline : 是因为函数有lambda表达式，属于高阶函数，高阶函数规范来说要加inline
>
> 2.<T, R> T.run : T代表是要为T而扩展出一个函数名run(任何类型都可以 万能类型.run)， R代表是Lambda表达式最后一行返回的类型
>
> 3.block: T.() -> R : Lambda表达式名称block 输入参数是T本身 输出参数是R 也就是表达式最后一行返回推断的类型

> 4.: R { : R代表是Lambda表达式最后一行返回的类型，若表达式返回类型是Boolean, 那么这整个run函数的返回类型就是Boolean
>
> 5.T.() 是让lambda表达式里面持有了this（run函数）， (T) 是让lambda表达式里面持有了it（let函数）

```
/*
1.inline  :  是因为函数有lambda表达式，属于高阶函数，高阶函数规范来说要加inline

2.<T, R>  T.run  :  T代表是要为T而扩展出一个函数名run(任何类型都可以 万能类型.run)，  R代表是Lambda表达式最后一行返回的类型

3.block: T.() -> R  :  Lambda表达式名称block 输入参数是T本身  输出参数是R 也就是表达式最后一行返回推断的类型

4.: R {  : R代表是Lambda表达式最后一行返回的类型，若表达式返回类型是Boolean，那么这整个run函数的返回类型就是Boolean

5.T.()  是让lambda表达式里面持有了this（run函数），  (T) 是让lambda表达式里面持有了it（let函数）
*/
public inline fun <T, R> T.run(block: T.() -> R): R {
    println("你${this}.run在${System.currentTimeMillis()}这个时间点调用了我")

    /*contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }*/

    // 调用Lambda表达式

    // 输入参数this == T == "Derry" / 123,

    // 输出参数：用户返回String类型，就全部是返回String类型

    return block()
}
```

> 总结：Kotlin内置标准run函数，运用了 高阶函数特性与Lambda，控制环节交给用户完成，用户在自己的Lambda表达式中，若返回Boolean，整个run函数 与 Lambda返回 都全部是Boolean
>
> 为了保证所有的类型都能正常使用run，给泛型增加了扩展函数run，所以所有的地方都可以使用run函数。

8.3 Kotlin语言泛型的形变是什么？

本题在[享学Android高级架构师体系]Kotlin专题中有详细讲解

这道题想考察什么？

1. 是否了解Kotlin语言泛型的形变是什么与真实场景使用，是否熟悉Kotlin语言泛型的形变是什么本质？

考察的知识点

1. Kotlin语言泛型的形变是什么的概念在项目中使用与基本知识

考生应该如何回答

1.你工作这么些年，对于Kotlin语言泛型的形变是什么，有了解么？

答：

形变一共分为三个区域：不变，协变，逆变

不变

不变指的是：这个泛型，可以是生产者，也可以是消费者，此泛型没有任何泛型继承相关的概念，可以理解是完全独立出来的泛型

例如：下面案例中，此泛型既可以是生产者，也可以是消费者

```
// 不变
class StudentSetGets<IO> {

    private var item : IO? = null

    // 消费者
    fun set(value : IO) {
        println("你传递进来的内容是:$value")
        item = value
    }

    // 生产者
    fun get() = item
}
```

协变

协变指的是，这个泛型，只能是生产者，此泛型有泛型继承相关的概念存在，可以理解此泛型，可以接收此泛型类型的子类型

例如：下面案例中，此泛型只能是生产者，说白了，只能给用户端，读取泛型，却不能修改泛型

```
class MyStudentGet<out T>(_item : T) {
    private val item = _item
    fun get() : T = item
}
```

逆变

逆变指的是，这个泛型，只能是消费者，此泛型有泛型父类转子类的强转相关的概念存在，可理解此泛型，可以接收此泛型类型的父类型

例如：下面案例中，此泛型只能是消费者，说白了，只能给用户端，修改泛型，却不能读取泛型

```
class MyStudentSet<in T>() {
    fun set(value: T) = println("你传递进来的内容是:$value")
}
```

结论

为什么协变只能读取泛型，不能修改泛型？

答：因为 例如 = 泛型接收端是Object，而泛型具体端是String，由于具体端有很多很多Object的子类，

而泛型会被泛型擦除，所以无法明确你到底要修改那个子类啊

为什么逆变只能修改泛型，不能读取泛型？

答：因为 例如 = 泛型接收端是String，而泛型具体端是Object，由于接收端是String，而读取时，

会读取到String的父类，但是接收端是String，你却读取到String的父类，这个本来就是不合理的

8.4 Kotlin协程在工作中有用过吗？

本题在[享学Android高级架构师体系]Kotlin专题中有详细讲解

这道题想考察什么？

理解协程的目的是，简化复杂的异步代码逻辑，用同步的代码写出复杂的异步代码逻辑。

考察的知识点

kotlin、协程、线程、并发

考生应该如何回答

1.你工作这么些年，对于Kotlin语言协程是什么，有了解么？

答：

虽然对于一些人来说比如刚开始的我，协程(Coroutines) 是一个新的概念，但是协程这个术语早在1958年就被提出并用于构建汇编程序，协程是一种编程思想，并不局限于特定的语言，就像Rx也是一种思想，并不局限于使用Java实现的RxJava。不同语言实现的协程库可能名称或者使用上有所不同，但它们的设计思想是有相似之处的。

kotlinx.coroutines是由JetBrains开发的kotlin协程库，可以把它简单的理解为一个线程框架 。但是协程不是线程，也不是新版本的线程，它是基于线程封装的一套更上层工具库，我们可以使用kotlin协程库提供的api方便的灵活的指定协程中代码执行的线程、切换线程，但是不需要接触线程Thread类。说到这里，大家可能就会想到Android的AsyncTask或者RxJava的Schedulers，没错，从某种意义上来说它们和协程是相通的，都解决了异步线程切换的问题，然而协程最重要的是通过非阻塞挂起和恢复实现了异步代码的同步编写方式，把原本运行在不同线程的代码写在一个代码块{}里，看起来就像是同步代码。

```
// 注意：在真实开发过程中，MainScope作用域用的非常常用
MainScope().launch(){      // 注意：此协程块默认是在UI线程中启动协程
    // 下面的代码看起来会以同步的方式一行行执行（异步代码同步获取结果）
    val token = apiService.getToken()    // 网络请求：IO线程，获取用户token
    val user = apiService.getUser(token)// 网络请求：IO线程，获取用户信息
    nameTv.text = user.name              // 更新 UI：主线程，展示用户名
    val articleList = apiService.getArticleList(user.id)// 网络请求：IO线程，根据用户id获取用户的文章集合哦
    articleTv.text = "用户${user.name}的文章页数是:${articleList.size}页"    // 更新UI：主线程
}
```

协程并不是从操作系统层面创立的新的运行方式，代码是运行在线程中的，线程又是运行在进程中的，协程也是运行在线程中的，所以才说它是基于线程封装的库。然而有人会拿协程与线程比较，问协程是不是比线程效率更高？如果理解了协程是基于线程封装就应该知道，协程并没有改变代码运行在线程中的原则，单线程中的协程执行时间并不会比不用协程少，它们之间没有可比性，因为它们根本不属于同一类事物；协程也不是为了线程而生的，它是为了解决因为多线程带来的编码上的不便的问题而出现的。

2.那这样说的话，协程到底有什么用？

在Android开发中，通常会将耗时操作放到子线程中，然后通过回调的方式将结果返回后切换主线程更新UI，但是实际开发过程中可能遇到很多奇怪而合理的需求，它们可能是：

一个页面需要同时并发请求多个接口，当所有接口都请求完成需要做一些合并处理然后更新UI
按照惯例，我们可能会为每个接口请求设置一个boolean标志，每当一个接口请求完将对应的boolean值改为true，当最后一个接口请求完成发现所有标志都为true再更新UI，这样就能达到并发请求的目的，然而管理这么多boolean值累不累？优雅不优雅？
初级程序员可能干脆来个单线程，一个接口请求完成后，再请求另一个接口，直到最后一个接口返回数据，玩暴力美学啊，本来能同时干的事情非得一件件干，你让用户浪费他宝贵的时间合适吗？浪费用户高配的性能过瘾吗？
高级一点的可能就上RxJava了，通过RxJava的zip操作符，达到发射一次，将结果合并处理的目的，但是说实话到现在还有很多人不会用

RxJava
先调用接口1获取数据，然后拿到接口1的结果作为参数调用接口2，然后将接口2的数据展示出来
按照惯例，我们可能会调用接口1，然后在接口1的回调中获取数据再嵌套调用接口2
高级一点的可能就上RxJava了

会引发回调地狱问题：

```
/**RetrofitClient单例*/
object RetrofitClient {
    /**log**/
    private val logger = HttpLoggingInterceptor.Logger {
        FLog.i(this::class.simpleName, it)
    }
```

```
    private val logInterceptor = HttpLoggingInterceptor(logger).apply {
        level = HttpLoggingInterceptor.Level.BODY
    }

    /**OkhttpClient*/
    private val okHttpClient = OkHttpClient.Builder()
            .callTimeout(10, TimeUnit.SECONDS)
            .addNetworkInterceptor(logInterceptor)
            .build()
    /**Retrofit*/
    private val retrofit = Retrofit.Builder()
        .client(okHttpClient)
        .baseUrl(ApiService.BASE_URL)
        .addCallAdapterFactory(RxJava3CallAdapterFactory.create())
        .addConverterFactory(GsonConverterFactory.create())
        .build()
    /**ApiService*/
    val apiService: ApiService = retrofit.create(ApiService::class.java)
}

/**接口定义*/
interface ApiService {
    companion object {
        const val BASE_URL = "https://www.wanandroid.com"
    }
    /*获取文章树结构*/
    @GET("tree/json")
    fun getTree(): Call<ApiResult<MutableList<Tree>>>

    /*根据数结构下某个分支id，获取分支下的文章*/
    @GET("article/list/{page}/json")
    fun getArticleList(
            @Path("page") page: Int,
            @Query("cid") cid: Int
    ): Call<ApiResult<Pagination<Article>>>
}

/**ViewModel*/
class SystemViewModel : BaseViewModel(){
    private val remoteRepository : SystemRemoteRepository by lazy {
SystemRemoteRepository() }

    val page = MutableLiveData<Pagination<Article>>()

    fun getArticleList() {
       remoteRepository.getArticleList(){
           page.value = it
       }
   }
}

/**数据仓库*/
class SystemRemoteRepository{
    /**
     * 1. 展示回调嵌套，回调地狱
     */
    fun getArticleList(responseBack: (result: Pagination<Article>?) -> Unit) {
            /**1. 获取文章树结构*/
```

```
            val call:Call<ApiResult<MutableList<Tree>>> =
RetrofitClient.apiService.getTree()
            //同步（需要自己手动切换线程）
            //val response : Response<ApiResult<MutableList<Tree>>> =
call.execute()
            //异步回调
            call.enqueue(object : Callback<ApiResult<MutableList<Tree>>> {
                override fun onFailure(call: Call<ApiResult<MutableList<Tree>>>,
t: Throwable) {
                }
                override fun onResponse(call:
Call<ApiResult<MutableList<Tree>>>, response:
Response<ApiResult<MutableList<Tree>>>) {
                    FLog.v("请求文章树结构成功："+response.body())

                    /**2. 获取分支id下的第一页文章*/
                    val treeid = response.body()?.data?.get(0)?.id
                    //当treeid不为null执行
                    treeid?.let {
                        RetrofitClient.apiService.getArticleList(0, treeid)
                                .enqueue(object :
Callback<ApiResult<Pagination<Article>>> {
                                    override fun onFailure(call:
Call<ApiResult<Pagination<Article>>>, t: Throwable) {
                                    }
                                    override fun onResponse(call:
Call<ApiResult<Pagination<Article>>>, response:
Response<ApiResult<Pagination<Article>>>) {
                                        //返回获取的文章列表
                                        responseBack(response.body()?.data)
                                    }
                                })
                    }

                }
            })
        }
}
```

其实上面数据仓库中的方法是应该拆分为2个方法的，第二个接口请求拆分为方法后还可以复用（分页获取），这里仅仅为了展示嵌套回调的需求。可以看到仅仅是两层回调嵌套，可读性就已经很差了，然而这种需求并非不常见的甚至会出现更多层嵌套，这时候就会写出深>形的代码，非常不雅观而且不易于代码复用及后期维护。

有人会说将上面的嵌套回调拆分为2个方法，在第一个接口请求完成之后再调用另一个方法请求文章列表，这不就消除了嵌套回调了吗？从视觉上来说确实是消除了，但是从逻辑上来说嵌套依然存在，而且这种方式会让两个方法之间形成很强的业务关联，对代码维护带来的挑战不比嵌套回调小。代码就不展示了，就是简单的将两个请求拆分。

Rx解决回调地狱

使用Retrofit+RxJava组合通过Rx的链式调用就能消除嵌套回调：

```
interface ApiService {
    ...
    /**RxJava方式*/
```

```
    @GET("tree/json")
    fun getTreeByRx(): Observable<ApiResult<MutableList<Tree>>>

    @GET("article/list/{page}/json")
    fun getArticleListByRx(
        @Path("page") page: Int,
        @Query("cid") cid: Int
    ): Observable<ApiResult<Pagination<Article>>>
}

class SystemRemoteRepository{
    /**
     * 2. Retrofit+RxJava消除回调嵌套
     */
    fun getArticleListByRx(responseBack: (result: Pagination<Article>?) -> Unit)
{
        /**1. 获取文章树结构*/
        val observable1: Observable<ApiResult<MutableList<Tree>>> =
RetrofitClient.apiService.getTreeByRx()
        observable1.subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                //使用当前Observable发出的值调用给定的Consumer，然后将其转发给下游
                .doOnNext({
                    FLog.v("1请求文章树成功，切换到主线程处理数据：
${Thread.currentThread()}")
                })
                .observeOn(Schedulers.io())
                .flatMap({
                    FLog.v("2请求文章树成功，IO线程中获取接口1的数据，然后将被观察者变换
为接口2的Observable：${Thread.currentThread()}")
                    if(it?.errorCode == 0){
                        //当treeid不为null执行
                        it?.data?.get(0)?.id?.let { it1 ->
RetrofitClient.apiService.getArticleListByRx(0, it1) }
                    }else{
                        //请求错误的情况，发射一个Error
                        Observable.error({
                            Throwable("获取文章树失败：
${it.errorCode}:${it.errorMsg}")
                        })
                    }
                })
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(object: Observer<ApiResult<Pagination<Article>>> {
                    override fun onComplete() {}
                    override fun onSubscribe(d: Disposable?) {}
                    override fun onNext(t: ApiResult<Pagination<Article>>?) {
                        FLog.v("3请求文章列表成功：${t?.data}")
                        responseBack(t?.data)
                    }
                    override fun onError(e: Throwable?) {
                        FLog.e("3请求失败：${e?.message}")
                    }
                })
    }
}
```

Retrofit+RxJava确实消除了回调的嵌套，但是还是避免不了回调（Observer观察者可看作是回调），链式调用处理异步数据流确实比传统的嵌套回调好了太多，但是代码量不减反增，而且我们需要在正确的位置准确的插入不同的操作符用来处理异步数据，对于不熟悉Rx的同学来说也是很头痛的，所以还不是很好，下面将出现协程来解决

协程来解决此问题：

使用协程就可以让我们摆脱因为多线程带来的各种编码上的不便：

```
class SystemViewModel : BaseViewModel(){
    private val remoteRepository : SystemRemoteRepository by lazy {
SystemRemoteRepository() }
    val page = MutableLiveData<Pagination<Article>>()

    fun getArticleList() {
        viewModelScope.launch {  //主线程开启一个协程
            // 网络请求：IO线程
            val tree : ApiResult<MutableList<Tree>> =
RetrofitClient.apiService.getTreeByCoroutines()
            // 主线程
            val cid = tree?.data?.get(0)?.id
            if(cid!=null){
                // 网络请求：IO线程
                val pageResult : ApiResult<Pagination<Article>> =
RetrofitClient.apiService.getArticleListByCoroutines(0, cid)
                // 主线程
                page.value = pageResult.data!!
            }
        }
    }
}

/**接口定义，Retrofit从2.6.0版本开始支持协程*/
interface ApiService {
    /*获取文章树结构*/
    @GET("tree/json")
    suspend fun getTreeByCoroutines(): ApiResult<MutableList<Tree>>
    /*根据数结构下某个分支id，获取分支下的文章*/
    @GET("article/list/{page}/json")
    suspend fun getArticleListByCoroutines(
        @Path("page") page: Int,
        @Query("cid") cid: Int
    ): ApiResult<Pagination<Article>>
}
```

运行上面的代码，尽然成功了，刚刚发生了什么？这不就是同步调用吗？跟上面的同步有什么不一样吗？看起来差不多啊，确实差不多，就是在定义接口时，方法前加了个suspend关键字，调用接口的时候用viewModelScope.launch {}包裹。既然可以运行成功，就说明请求接口并不是在主线程中进行的，然而有的同学不信，他在getArticleList()方法中的任意位置通过Thread.currentThread()打印的结果都是Thread[main,5,main]，这不就是在主线程中调用的吗？上述协程中的代码是在主线程执行没错，但是接口请求的方法却是在子线程中执行的。

原因就在于我们定义接口的时候使用了suspend关键字，它的意思是挂起、暂停，函数被加了这个标记就称它为挂起函数，需要注意的是，suspend关键字并没有其他重要的作用，它仅仅标识某个函数是挂起函数，可以在函数中调用其他挂起函数，但是只能在协程中调用它。所以上面两个接口都被定义为挂起函数了，挂起函数只能在协程中调用，那谁是协程？

其实在kotlin协程库中是有一个类AbstractCoroutine来表示协程的，这个抽象类有很多子类代表不同的协程，但是这些子类都是private的，并没有暴露给我们，所以你在其他文章中看到别人说viewModelScope.launch{}包裹起来的闭包(代码块)就是协程也没问题，但这个代码块的真正意义是协程需要执行的代码。当在协程中调用到挂起函数时，协程就会在当前线程（主线程）中被挂起，这就是协程中著名的非阻塞式挂起，主线程暂时停止执行这个协程中剩余的代码，注意：暂停并不是阻塞等待（否则会ANR），而是主线程暂时从这个协程中被释放出来去处理其他Handler消息，比如响应用户操作、绘制View等等。

那挂起函数谁执行？这得看挂起函数内部是否有切换线程，如果没有切换线程当然就是主线程执行了，所以挂起函数不一定就是在子线程中执行的，但是通常在定义挂起函数时都会为它指定其他线程，这样挂起才有意义。比如上面定义的suspend的请求接口，Retorift在执行请求的时候就切换到了子线程并挂起主线程，当请求完成（挂起函数执行完毕）返回结果时，会通知主线程：我该干的都干完了，下面的事你接着干吧，主线程接到通知后就会拿到挂起函数返回的结果继续执行协程里面剩余的代码，这叫做协程恢复(resume)。如果又遇到挂起函数就会重复这个过程，直到协程中的代码被执行完。

通过协程可以彻底去除回调，使用同步的方式编写异步代码。什么是同步调用？调用一个方法能直接拿到方法的返回值，尽管这个方法是耗时的、在其他线程执行的，也能直接得到它的返回值，然后再执行下面的代码，协程不是通过等待的方式实现同步，而是通过非阻塞挂起实现看起来同步的效果。

总结：

协程的目的是，简化复杂的异步代码逻辑，用同步的代码写出复杂的异步代码逻辑。