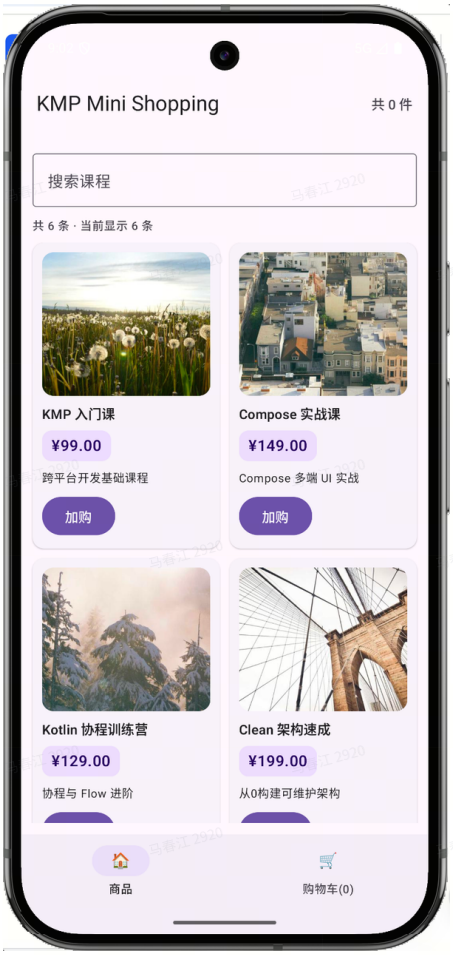
KMP实战-商城 副本

前言

这节课带着大家做一个简单的实战项目,一个商城列表+购物车







UI_参考资料

UI给大家准备好了,咱们不用太关心UI,只学习逻辑层代码实现

> 按步骤完成后可稳定实现：
> 1）商品列表加载；2）加入购物车；3）重启后购物车不丢。

一、课前准备

1. 环境

- Android Studio 最新稳定版
- 可运行 Android 模拟器或真机

2. 创建项目

确认项目是 KMP Wizard 标准结构：

```
kmp-mini-shopping/
├─ composeApp/                # Android UI 层
├─ shared/                    # 共享业务层
│   ├─ src/commonMain
│   ├─ src/androidMain
│   └─ src/iosMain
└─ docs/
```

3. AndroidManifest 网络权限

`composeApp/src/androidMain/AndroidManifest.xml`

```
<uses-permission android:name="android.permission.INTERNET" />
```

> 课程使用 HTTPS 地址：`https://shopneu.hmos.site`。
> 如你使用 HTTP 联调时，需要加 `usesCleartextTraffic=true`。

二、本节目标

今天只做三件事：

1. 在 `shared` 实现商品网络请求（Ktor）
2. 在 `shared` 实现购物车持久化（Settings）
3. Android 页面只做“展示 + 调用 shared”

> 强调：业务规则都在 shared，不写死在 Android 页面里。

三、依赖配置

> 目标：让 shared 能发请求、能序列化、能存本地。

`shared/build.gradle.kts`

确认（或添加）这些依赖：

代码块

```
kotlin {
    sourceSets {
        all {
            languageSettings.optIn("kotlinx.cinterop.ExperimentalForeignApi")
            languageSettings.optIn("kotlin.experimental.ExperimentalObjCName")
        }

        commonMain.dependencies {
            //网络
            implementation("io.ktor:ktor-client-core:3.1.3")
            implementation("io.ktor:ktor-client-content-negotiation:3.1.3")
            //数据序列化
            implementation("io.ktor:ktor-serialization-kotlinx-json:3.1.3")
            implementation("org.jetbrains.kotlinx:kotlinx-serialization-core:1.8.1")
            //跨平台存储
            implementation("com.russhwolf:multiplatform-settings-no-arg:1.1.1")
        }

        androidMain.dependencies {
            implementation("io.ktor:ktor-client-okhttp:3.1.3")
        }
    }
}
```

Android 侧引擎（`shared/androidMain`）需要：

- `ktor-client-okhttp`（或 Android 引擎）
- Ktor Multiplatform Client 官方教程：`https://ktor.io/docs/client-create-multiplatform-application.html`
- Ktor Serialization 官方说明（ContentNegotiation/JSON 的用途）：`https://ktor.io/docs/server-serialization.html`

课程统一后端地址

- 线上地址：`https://shopneu.hmos.site`
- 商品接口：`https://shopneu.hmos.site/api/products`
- 后端项目源码,有兴趣的可以自己跑

kmp-mini-shopping-backend.zip
240.27KB

四、商品列表走网络数据

1. 模型层

商品模型

文件：

`shared/src/commonMain/kotlin/com/example/minishop/model/Product.kt`

代码块

```kotlin
package com.example.minishop.model

import kotlinx.serialization.Serializable

@Serializable
data class Product(
    val id: String,
    val title: String,
    val price: Double,
    val description: String,
    val imageUrl: String = ""
)
```

购物车项模型

文件：

`shared/src/commonMain/kotlin/com/example/minishop/cart/CartItem.kt`

代码块

```kotlin
package com.example.minishop.cart

import kotlinx.serialization.Serializable

@Serializable
data class CartItem(
    val productId: String,
    //件数
    val quantity: Int
)
```

课堂检查点：

- 两个模型都加了 `@Serializable`

- `imageUrl` 给默认值，后续缺图不崩

---

2. 网络层

HttpClient 工厂（commonMain）

文件：`shared/.../network/HttpClientFactory.kt`

代码块

```
package com.example.minishop.network

import io.ktor.client.HttpClient
import io.ktor.client.engine.HttpClientEngine
import io.ktor.client.plugins.contentnegotiation.ContentNegotiation
import io.ktor.serialization.kotlinx.json.json
import kotlinx.serialization.json.Json

expect fun provideHttpEngine(): HttpClientEngine

fun createHttpClient(): HttpClient = HttpClient(provideHttpEngine()) {
    install(ContentNegotiation) {
        json(Json {
            ignoreUnknownKeys = true
            isLenient = true
        })
    }
}
```

`contentNegotiation` 是干什么的

`ktor-client-content-negotiation` + `ktor-serialization-kotlinx-json` 这两个依赖，
就是让 Ktor 能把接口返回的 JSON 自动反序列化成 Kotlin 数据类（例如 `List<Product>`）。

对应代码就是：

代码块

```
install(ContentNegotiation) {
    json(Json {
        ignoreUnknownKeys = true
        isLenient = true
    })
}
```

如果不加这两个依赖，`body<List<Product>>()` 这类写法会失败（无法自动解析 JSON）。

另外，`ignoreUnknownKeys = true` 的作用是：

- 当后端返回了“数据类里未定义的字段”时，不报错，直接忽略。

示例：

- `Product` 只定义了 `id/title`
- 后端额外返回 `tag/extra`
- 开启 `ignoreUnknownKeys = true` 后，`id/title` 仍可正常解析。

Android 实现 actual

文件：`shared/src/androidMain/.../network/HttpClientFactory.android.kt`

代码块

```
package com.example.minishop.network

import io.ktor.client.engine.HttpClientEngine
import io.ktor.client.engine.okhttp.OkHttp

actual fun provideHttpEngine(): HttpClientEngine = OkHttp.create()
```

商品 API

文件：`shared/.../network/ProductApi.kt`

代码块

```
package com.example.minishop.network

import com.example.minishop.model.Product
import io.ktor.client.HttpClient
import io.ktor.client.call.body
import io.ktor.client.request.get

class ProductApi(
    private val baseUrl: String,
    private val client: HttpClient = createHttpClient()
) {
    suspend fun fetchProducts(): List<Product> {
        return client.get("$baseUrl/products").body()
    }
}
```

`suspend`（译为「挂起」）是 Kotlin 中用于标记「可暂停执行」函数的关键字，其核心作用是：允许函数在执行过程中暂时挂起（释放线程资源），等待某个耗时操作（如网络请求、数据库读写、文件IO）完成后，再恢复执行后续代码，且整个过程不会阻塞当前线程。

简单来说，挂起函数就是「能暂停、能恢复、不阻塞」的特殊函数。

3. 仓储层

ProductRepository

文件：`shared/.../repository/ProductRepository.kt`

代码块

```
package com.example.minishop.repository

import com.example.minishop.model.Product
import com.example.minishop.network.ProductApi

class ProductRepository(
    private val api: ProductApi,
    private val fallback: List<Product> = emptyList()
) {
    suspend fun getProducts(): List<Product> = runCatching { api.fetchProducts() }.getOrDefault(fallback)
}
```

`runCatching` 执行给定代码块，成功就捕获结果，失败（抛异常）就捕获异常。可以把它看成 Kotlin 函数式风格的异常处理方式，返回 Result 类型（密封类，存成功或失败结果）。

`getOrDefault` 如果有值就返回否则返回默认值这里没有默认返回emptyList()

4. UI调用

App.kt中

代码块

```
val scope = rememberCoroutineScope()
val productRepository = remember {
    ProductRepository(
        api = ProductApi(baseUrl = "https://shopneu.hmos.site")
    )
}
fun loadProducts() {
    scope.launch {
        val list = runCatching { productRepository.getProducts() }
            .getOrDefault(emptyList())
```

```
            products = list
        }
    }

    LaunchedEffect(Unit) { loadProducts() }

    //products修改
    var products by remember { mutableStateOf<List<Product>>(emptyList()) }
```

`LaunchedEffect`：在某个可组合项的作用域内运行挂起函数.当 `LaunchedEffect` 进入组合时，它会启动一个协程，并将代码块作为参数传递。如果 `LaunchedEffect` 退出组合，协程将取消。

参考资料: https://developer.android.google.cn/develop/ui/compose/side-effects?hl=zh-cn

`rememberCoroutineScope` 它的核心作用是：在 Composable 函数内部获取一个与当前组合生命周期绑定的 CoroutineScope，以便在非挂起环境（如点击回调、普通逻辑块）中启动协程。

`协程` 是一种轻量级的线程管理机制，本质上是可暂停的计算单元。与传统线程相比，协程更高效，因为它允许在不阻塞线程的情况下暂停和恢复执行。

简单来说就是处理异步/多线程的.

参考资料: https://developer.android.google.cn/kotlin/coroutines.html?hl=zh-cn

5. mock代码替换

把之前mock代码UIProduct换成刚才新建的Product类然后运行,会发现报错

代码块

```
Serializer for class 'List' is not found.
Please ensure that class is marked as '@Serializable' and that the
serialization compiler plugin is applied.
```

序列化问题,要在 `build.gradle` 的 `plugins` 里加上

代码块

```
alias(libs.plugins.kotlinxSerialization)
```

libs.versions.toml中

代码块

```
kotlinxSerialization = { id = "org.jetbrains.kotlin.plugin.serialization", version.ref = "kotlin" }
```

这样数据就展示出来了

6. 网络图片

使用coil三方库实现,先导入到composeApp/androidMain的dependencies,因为只在安卓里用,这个是UI方面的了解一下就可以,文档:https://coil-kt.github.io/coil/compose/

代码块

```
implementation("io.coil-kt.coil3:coil-compose:3.4.0")
//网络图片必须的网络库
implementation("io.coil-kt.coil3:coil-network-ktor3:3.4.0")
```

改造Cover组件

代码块

```
@Composable
private fun Cover(url: String) {
    if (url.isNotBlank()) {
        AsyncImage(
            model = url,
            contentDescription = null,
        )
    } else {
        Image(
            painter = painterResource(id = R.drawable.course_cover_default),
            contentDescription = null,
            modifier = Modifier
                .fillMaxWidth()
                .height(150.dp)
                .clip(RoundedCornerShape(12.dp)),
            contentScale = ContentScale.Crop
        )
    }
}
```

五、购物车持久化

1. 编写CartStorage数据持久化

文件：`shared/.../data/CartStorage.kt`

代码块

```
package com.example.minishop.data

import com.russhwolf.settings.Settings
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

class CartStorage(
    private val settings: Settings = Settings(),
    private val json: Json = Json { ignoreUnknownKeys = true }
) {
    private val key = "cart_items"

    fun load(): List<CartItem> {
        val raw = settings.getStringOrNull(key) ?: return emptyList()
        return runCatching { json.decodeFromString<List<CartItem>>(raw) }
            .getOrDefault(emptyList())
    }

    fun save(items: List<CartItem>) {
        settings.putString(key, json.encodeToString(items))
    }
}
```

使用multiplatform setting库对数据持久化

2. 仓储层CartRepository

文件：`shared/.../repository/CartRepository.kt`

代码块

```
package com.example.minishop.cart

class CartRepository(
    private val storage: CartStorage = CartStorage()
) {
    //通过ProductId转成map
    private val map = storage.load().associateBy { it.productId
}.toMutableMap()

    fun items(): List<CartItem> = map.values.sortedBy { it.productId }
```

```
10
11      fun add(productId: String) {
12          val old = map[productId]
13          map[productId] = if (old == null) CartItem(productId, 1) else
    old.copy(quantity = old.quantity + 1)
14          persist()
15      }
16
17      fun increase(productId: String) {
18          val old = map[productId] ?: return
19          map[productId] = old.copy(quantity = old.quantity + 1)
20          persist()
21      }
22
23      fun decrease(productId: String) {
24          val old = map[productId] ?: return
25          if (old.quantity <= 1) map.remove(productId) else map[productId] =
    old.copy(quantity = old.quantity - 1)
26          persist()
27      }
28
29      fun remove(productId: String) {
30          map.remove(productId)
31          persist()
32      }
33
34      fun clear() {
35          map.clear()
36          persist()
37      }
38
39      private fun persist() = storage.save(items())
40  }
```

3. UI调用

在App.kt中初始化 `val cartRepository = remember { CartRepository() }`

新增状态变量

代码块

```
1  //用来刷新数据
2  var tick by remember { mutableStateOf(0) }
3  val cartItems = remember(tick) { cartRepository.items() }
```

点击加购调用 `cartRepository.add(product.id)`
tick++

六、知识点总结

1. Ktor/Multiplatform Setting/Coil/serialization 三方库的使用,分别对应网络,数据持久化,图片加载,数据序列化的能力
2. 简单的分层架构,数据model层-仓储层-网络层,以及UI调用到数据返回的完整闭环
3. 协程,suspend等异步/多线程编程相关知识(kotlin高阶)
4. 调试debug runCatching/onFailure抓报错的方法

作业

1. 完成详情页加购并显示已加购件数
2. 完成列表页点击加购旁边显示x几
3. 购物车item点击加减号/清空/结算弹出提示
4. 以上所有都要能联动
5. 提交运行录屏