3.5 保存筛选获取数据并调用接口的逻辑

- 现在接口调用逻辑在 ProblemThreeSelectDelegate 位置
- 两个Fragment之间通过接口实现调用吧
  - 接口：
    - ```
      // 定义接口：用于将保存的配置传递给 主Fragment，再由 主Fragment 转发到目标 Fragment
      interface FilterConfigListener {
          fun onConfigSaved(configName: String, selectedNames: List<String>)
      }
      ```
  - MainFragment中实现接口
    - ```
      class ProblemWorkBenchFragment : BaseChangeFragment(), FilterConfigListener {
          // 目标 Fragment 的引用
          private var targetFragment: TargetFragment? = null

          override fun onConfigSaved(configName: String, selectedNames: List<String>) {
              // 转发数据到目标 Fragment
              targetFragment = supportFragmentManager.findFragmentByTag("TargetFragment") as? TargetFragment
              targetFragment?.updateConfig(configName, selectedNames)
          }
      }
      ```
  - 源Fragment，也就是调用保存筛选按钮的Fragment
    - ```
      //和主Fragment通信的接口实例
      private var filterConfigListener: FilterConfigListener? = null


      // 保存当前筛选结果的引用（需根据你的实际代码获取）
      private lateinit var currentFilterResult: ProblemFilterResult
      // 保存所有配置的集合（Key=配置名称，Value=选中的筛选项name列表）
      private val savedConfigs = mutableMapOf<String, List<String>>()


      override fun onAttach(context: Context) {
              super.onAttach(context)
              // 绑定 MainFragment 的接口实现
              filterConfigListener = context as? FilterConfigListener
          }

      //保存筛选
      binding.llSave.setOnClickListener {
          dialog = SaveFilterDialog(requireContext()) { inputName ->
               handleSaveFilter(inputName) （这里不应该放在协程里面处理）
          }
          dialog?.show()
      ```
      ```

```
}

    // 处理保存逻辑
    private fun handleSaveFilter(configName: String) {
        // 获取当前选中的筛选项 name 列表
        val selectedNames = currentFilterResult.list
            .filter { it.isChecked && !it.name.isNullOrEmpty() }
            .map { it.name!! }

        // 保存到内存（可扩展为数据库）
        savedConfigs[configName] = selectedNames

        // 通知目标 Fragment 更新 UI
       filterConfigListener?.onConfigSaved(configName,
selectedNames)
    }


override fun onDetach() {
    super.onDetach()
    filterConfigListener = null // 避免内存泄漏
}
```

- 这里我们可以看到，handleSaveFilter(inputName) （这里不应该放在协程里面处理）
- 它的本质是：接口回调只适合于线程内部通信，不适合线程间通信，那么请问为什么线程内部的Fragment通信方式接口回调不适用于线程间通信?
  - 理解一下底层通信原理，接口回调本质是同一个线程内部的虚拟机栈中当前再执行的方法去调用另一个方法。这个方法的上下文都在当前线程里面。如果是不同线程，他们不共用虚拟机栈。
  - 说的有点抽象，我想想怎么可以更好的解释。
  - 这样，我们先来理解方法调用的本质：方法依赖于本身的字节码以及参数，他们分别是私有和公有的。
    - **方法区共享**：所有线程共享 JVM 的方法区，接口方法的字节码存储在方法区中。
    - **虚拟机栈私有**：每个线程有自己的虚拟机栈，存储方法调用的栈帧（局部变量、操作数栈、动态链接等）。
  - 那么线程内部接口调用的过程是什么：
    - **获取方法地址**：通过虚方法表（vtable）或接口方法表（itable）找到方法在方法区中的地址。
    - **压入当前栈帧**：将方法的参数和返回地址压入当前线程的虚拟机栈。
    - **执行上下文**：方法执行时，直接访问当前线程的栈帧中的局部变量和 `this` 引用（隐式参数）。
  - ok，现在来回答为什么线程内部的Fragment通信方式接口回调不适用于线程间通信?
    - **虚拟机栈私有性**：不同线程有自己的虚拟机栈，他们的栈帧完全隔离，后台线程无法直接访问主线程的栈帧（如 UI 组件的引用）。
    - **上下文丢失**：若接口方法依赖主线程的上下文（如更新 `TextView`），在后台线程调用时会因缺少主线程的 `Looper` 和消息队列而崩溃。

- 目标Fragment接收数据并更新UI
  - ```
    // 目标 Fragment（例如 SavedConfigFragment.kt）
    class SavedConfigFragment : Fragment() {
    ```
    ```

```
    // 保存配置的集合
    private val configs = mutableMapOf<String, List<String>>()

    // 提供给 Activity 调用的更新方法
    fun updateConfig(configName: String, selectedNames:
List<String>) {
        configs[configName] = selectedNames
        refreshRecyclerView()
    }

    private fun refreshRecyclerView() {
        binding.recyclerView.adapter =
ConfigAdapter(configs.toList()).apply {
            // 点击项可删除配置（可选）
            setOnItemClickListener { position ->
                val configName = configs.keys.elementAt(position)
                configs.remove(configName)
                notifyItemRemoved(position)
            }
        }
    }
}
```

- 现在就是currentFilterResult数据的获取了。
- 该数据是用于保存点击筛选配置选项的文件，实机去Debug，点击按钮，发现频繁出现一个 `ProblemThreeSelectDelegate` 类的日志。
- 去里面查看代码逻辑，内部设置了onItemClick回调，会更新list中被选中项的isChecked的状态。
- list来源于其内部的 `getNewList(result.list)`，也就是将原始的 `ProblemFilterResult.list` 转换为适配 `GridSelectLayout` 显示的新列表。
- 那么这个result从哪来？构造方法中有接收这个，我想想，class ProblemFilterFragment : BaseChangeFragment()调用了他
  - ```
    selectDelegate =
        ProblemThreeSelectDelegate(mContext, titles, screenWidth,
    false, false, false,


    //刷新适配器
                    adapter?.datas?.clear()
                    adapter?.datas?.addAll(newAllList!!.toList())
                    adapter?.notifyDataSetChanged()
    ```
- 所以他的数据来自于newAllList，那么这个来自于哪？
  - ```
    newAllList?.addAll(allList)

     allList[getRealPos(6)].timeRange = 30

     newAllList?.get(index)!!.list.get(index1).name ==
    mContext.resources.getString(
                                            R.string.text_all
                                        )
    ```

- allList来源：
  - ```
    fun getInstance(
                jsonList: String?,
                allList: MutableList<ProblemFilterResult>?,
                titles: List<String>,
                callback: IProblemActionCallback?
            ): ProblemFilterFragment {
                val filterFragment = ProblemFilterFragment()
                filterFragment.jsonList = jsonList
                filterFragment.allList = allList!!
                filterFragment.titles = titles
                filterFragment.callback = callback
                return filterFragment
            }
    ```
- 谁构造了这个Fragment?
  - ProblemWorkBenchFragment中构造：
    - ```
      filterFragment = ProblemFilterFragment.getInstance(
                  jsonList,
                  allList,
                  titleList,
                  object : IProblemActionCallback {
                      override fun onCommit() {

      animateReveal(binding.problemCreateFilterLayout, false)
                          fragments[location].refreshProblemList(
                              createOrder,
                              commonOrder, priorityOrder,
      expirationOrder, filterFragment!!.allList
                          )
                      }

                      override fun onDismiss() {

      animateReveal(binding.problemCreateFilterLayout, false)
                      }
                  })
      ```
  - 那么这个allList和jsonList从哪来?
    - ```
      var jsonList = AssetsUtils.loadText(requireContext(),
      "problem_filter_pending.json")
      val subList: MutableList<ProblemFilterResult> =
      ArrayList<ProblemFilterResult>()//筛选显示项.

       fun loadText(context: Context, assetFielPath: String?):
      String? {
              var `is`: InputStream? = null
              try {
                  `is` =
      context.resources.assets.open(assetFielPath!!)
                  return convertStreamToString(`is`)
              } catch (e: IOException) {
                  e.printStackTrace()
              }
      ```
      ```

```
            return null
        }

    allList = subList
    jsonList = Gson().toJson(subList)
```

            - ok，jsonList为字符串类型，其实就是将输入流转化为String。
            - allList是通过GSON转换而成。
            - 所以，目前的一切数据保存在allList中
        - 这个Fragment也就是工作台Fragment。其会调用我们当前需要编码的源Fragment。
    - 我们要从里面获取数据，该怎么处理？接口暴露？
        - 采用接口？假如他要实现后续的逻辑，这里会形成的层级是：爷爷-父亲-孩子，爷爷和孩子之间通信，那么父亲也要实现接口啊。这样很繁琐啊，那通过EventBus？该怎么处理呢？
- ok，假如我现在拿到数据了，我该怎么操作呢？
    - 传递引用还是数据呢？或者我在这里重构筛选出合适数据的引用，然后发过去？
    - 传递引用吧，这样不用拷贝耗时，假设拿到的是allList
- 这样操作problemFilterResult要考虑多线程访问的数据安全性问题吗？
    - ```
      data class SelectedInfo(val id: Int, val name: String)

      val result = mutableListOf<SelectedInfo>()
      for (item in problemFilterResult.list) {
          if (item.isChecked) {
              result.add(SelectedInfo(item.id, item.nameId, item.name))
          }
      }
      ```
    - 我们上锁？不行，UI主线程上锁，你在想什么呢？那么我们该怎么操作这个problemFilterResult
    - ```
      val safeCopy = problemFilterResult.list.toList() // 创建独立副本
      val result = safeCopy
          .filter { it.isChecked }
          .map { SelectedInfo(it.id, it.name) }
      ```
    - 关键问题是：**外部数据在读取期间是否会被其他线程修改**？
    - 从操作逻辑来看，我们点击保存，他会执行这个方法，然后才是dialog的dismiss。这个方法会将数据结果保存并传递，理论上有dialog在前面挡着，不会出现数据安全性问题。那我们直接读取？