前言

> 学习要符合如下的标准化链条：了解概念->探究原理->深入思考->总结提炼->底层实现->延伸应用"

01.学习概述

- **学习主题**：Object关键字
- **知识类型**：
    - [ ] [x] Android/
        - [ ] [x] 01.基础组件
        - [ ] [x] 02.IPC机制
        - [ ] [x] 03.消息机制
        - [ ] [x] 04.View原理
        - [ ] [x] 05.事件分发机制
        - [ ] [x] 06.Window
        - [ ] [x] 07.复杂控件

- [ ] ☑08.性能优化
- [ ] ☑09.流行框架
- [ ] ☑10.数据处理
- [ ] ☑11.动画
- [ ] ☑12.Groovy

- [ ] ☑ Java/
  - [ ] ☑01.基础知识
  - [ ] ☑02.Java设计思想
  - [ ] ☑03.集合框架
  - [ ] ☑04.异常处理
  - [ ] ☑05.多线程与并发编程
  - [ ] ☑06.JVM
- [ ] ☑ Kotlin/
  - [x] ☑01.基础语法
  - [ ] ☑02.高阶扩展
  - [ ] ☑03.协程和流

- **学习来源**:
- **重要程度**: ★★★★★ (核心基础)
- **学习日期**:
- **记录人**: @panruiqi

1.1 学习目标

- 了解概念->探究原理->深入思考->总结提炼->底层实现->延伸应用"

1.2 前置知识

- [ ] Kotlin基础语法

02.核心概念

2.1 是什么?

object是关键字，用于声明一个单例对象，这种对象全局只有一个实例，并且在第一次使用时自动初始化。

- 举例:
  - ```
    object PopupFactory {
        fun createPopup() { /* ... */ }
    }
    ```
  - 对应如下
    ```

```
public final class PopupFactory {
    public static final PopupFactory INSTANCE = new PopupFactory();

    private PopupFactory() { } // 私有构造器，防止外部实例化

    public void createPopup() { /* ... */ }
}
```

- 在 Kotlin 里，你直接用 PopupFactory.createPopup()，编译器会自动转成 PopupFactory.INSTANCE.createPopup()。

2.2 解决什么问题?

- object 关键字就是为了让你用最简洁、安全的方式声明单例对象，避免手写模板代码，提高开发效率和代码可读性。

2.3 基本特性

03.原理机制

3.1 进一步思考

怎么理解这里的final?

- final 修饰的类不能被继承。
- final 修饰的字段只能赋值一次（即常量）。

他为什么是线程安全的?

- 线程安全：依赖 JVM 的类加载和初始化机制，保证单例对象只会被创建一次，且不会有并发问题。

3.2 进一步思考

object不是所有对象的父类吗？为什么这里是这样的解释

- 在 Kotlin 里，object 是用来声明单例对象的关键字，比如 object PopupFactory。
- Java 的 Object 类：所有 Java 类的根父类。
- Kotlin 的 Any 类：所有 Kotlin 类的根父类（类似于 Java 的 Object，但更精简）。

3.3 进一步思考

companion object（伴生对象）是什么?

- companion object 是 Kotlin 的一种特殊用法，为类声明一个“伴生单例对象”

伴生对象的原理是什么?

- 伴生对象本质上是一个静态单例对象，作为宿主类的静态字段存在。
- 伴生对象的方法和属性通过单例对象访问，Kotlin 编译器会生成静态代理方法，实现类似 Java 静态成员的访问体验。

- ```
  class MyClass {
      companion object {
          fun foo() = println("Hello from companion!")
      }
  }
  ```

- ```
  public final class MyClass {
      public static final MyClass.Companion Companion = new
  MyClass.Companion();

      public static final class Companion {
          public final void foo() {
              System.out.println("Hello from companion!");
          }
      }
  }
  ```

3.4 进一步思考

怎么理解这行代码 `class PopupManager private constructor(private val *activity*: Activity)`

- 它 表示这个类的构造函数是私有的，外部代码不能直接 new 这个类的实例。他通过companion为自己创建实例
- 举例：
  - ```
    class PopupManager private constructor(private val activity: Activity)
    {

        companion object {
            private val instances = mutableMapOf<String, PopupManager>()

            /**
             * 为每个Activity创建独立的弹窗管理器实例
             */
            fun getInstance(activity: Activity): PopupManager {
                val key = activity.javaClass.simpleName
                return instances.getOrPut(key) { PopupManager(activity) }
            }

            /**
             * 清理Activity对应的实例
             */
            fun clearInstance(activity: Activity) {
                val key = activity.javaClass.simpleName
                instances[key]?.cleanup()
                instances.remove(key)
            }
        }
    ```

3.5 进一步思考

还有其他的单例模式吗?
    ```

- ```
  fun getInstance(context: Context): VoiceManager {
       return INSTANCE ?: synchronized(this) {
       INSTANCE ?: VoiceManager(context.applicationContext).also { INSTANCE =
  it }
     }
  }
  ```

- 这是经典的双重检测锁定（Double-Checked Locking）单例模式。既保证了线程安全，又避免了每次获取实例都加锁，性能较好。
- 如果 INSTANCE 还没有被初始化（为 null），就进入同步代码块，保证多线程下只会有一个线程初始化 INSTANCE。
- 再次检查 INSTANCE 是否为 null（因为可能有多个线程同时到达 synchronized 之前）
- 如果还是 null，就创建一个新的 VoiceManager 实例，并赋值给 INSTANCE

04.底层原理

05.深度思考

5.1 关键问题探究

5.2 设计对比

06.实践验证

6.1 行为验证代码

6.2 性能测试

07.应用场景

7.1 最佳实践

7.2 使用禁忌
  ```

08.总结提炼

8.1 核心收获

8.2 知识图谱

8.3 延伸思考

09.参考资料

1.
2.
3.