Kotlin下的5种单例模式

## 1.前言

最近在学习Kotlin这门语言，在项目开发中，运用到了单例模式。因为其表达方式与Java是不同的。所以对不同单例模式的实现进行了分别探讨。主要单例模式实现如下：

饿汉式

懒汉式

线程安全的懒汉式

双重校验锁式

静态内部类式

PS:**该篇文章不讨论单例模式的运用场景与各种模式下的单例模式的优缺点。只讨论在Java下不同单例模式下的对应Kotlin实现。**

## 2.饿汉式实现

//Java实现

public class SingletonDemo {

private static SingletonDemo instance=new SingletonDemo();

private SingletonDemo(){

}

public static SingletonDemo getInstance(){

return instance;

}

}

//Kotlin实现

object SingletonDemo

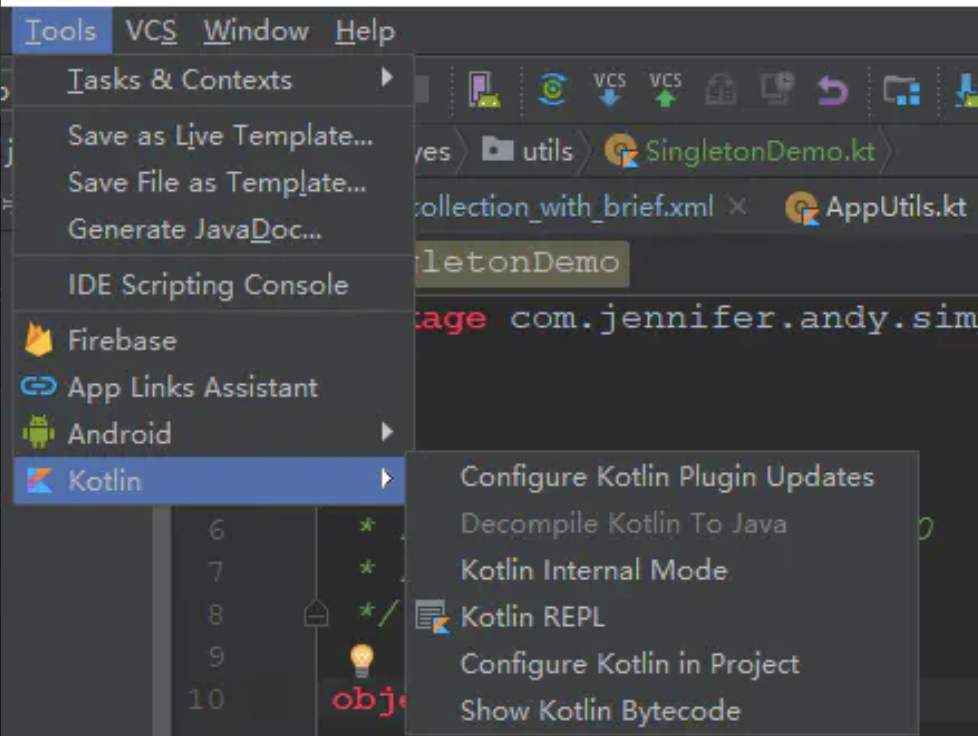
这里很多小伙伴，就吃了一惊。我靠一个**object** 关键字就完成相同的功能？一行代码？

**Kotlin的对象声明**

**在Kotlin中类没有静态方法**。如果你需要写一个可以无需用一个类的实例来调用，但需要访问类内部的函数（例如，工厂方法,单例等），你可以把该类声明为一个**对象**。该**对象**与其他语言的静态成员是类似的。如果你想了解Kotlin对象声明的更多内容。请点击- - - [传送门](https://www.kotlincn.net/docs/reference/object-declarations.html#%E4%BC%B4%E7%94%9F%E5%AF%B9%E8%B1%A1)

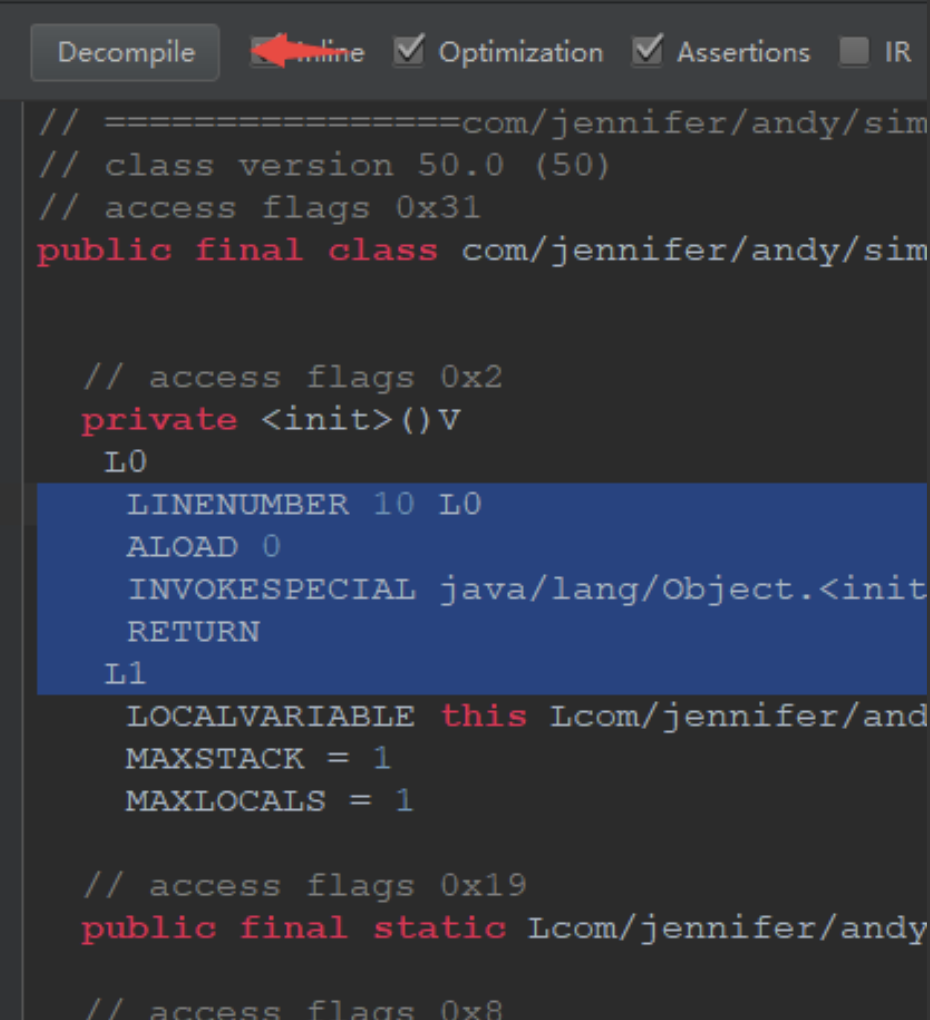
到这里，如果还是有很多小伙伴不是很相信一行代码就能解决这个功能，我们可以通过一下方式查看Kotlin的字节码。

查看Kotlin对应字节码



我们进入我们的Android Studio(我的Android Studio 3.0,如果你的编译器版本过低，请自动升级) 选择**Tools工具栏**，选择"**Kotlin**",选择“**Show Kotlin Bytecode**"

选择过后就会进入到下方界面：



点击"**Decompile**" 根据字节码得到以下代码

public final class SingletonDemo {

public static final SingletonDemo INSTANCE;

private SingletonDemo(){}

static {

SingletonDemo var0 = new SingletonDemo();

INSTANCE = var0;

}

}

通过以上代码，我们了解事实就是这个样子的，使用Kotlin"object"进行对象声明与我们的饿汉式单例的代码是相同的。

## 3、懒汉式

//Java实现

public class SingletonDemo {

private static SingletonDemo instance;

private SingletonDemo(){}

public static SingletonDemo getInstance(){

if(instance==null){

instance=new SingletonDemo();

}

return instance;

}

}

//Kotlin实现

class SingletonDemo private constructor() {

companion object {

private var instance: SingletonDemo? = null

get() {

if (field == null) {

field = SingletonDemo()

}

return field

}

fun get(): SingletonDemo{

return instance!!

}

}

}

上述代码中，我们可以发现在Kotlin实现中，我们让其**主构造函数私有化**并自定义了其**属性访问器**，其余内容大同小异。

如果有小伙伴不清楚Kotlin构造函数的使用方式。请点击 - - - [构造函数](https://www.kotlincn.net/docs/reference/classes.html)

不清楚Kotlin的属性与访问器，请点击 - - -[属性和字段](https://www.kotlincn.net/docs/reference/properties.html)

## 3.线程安全的懒汉式

//Java实现

public class SingletonDemo {

private static SingletonDemo instance;

private SingletonDemo(){}

public static synchronized SingletonDemo getInstance(){//使用同步锁

if(instance==null){

instance=new SingletonDemo();

}

return instance;

}

}

//Kotlin实现

class SingletonDemo private constructor() {

companion object {

private var instance: SingletonDemo? = null

get() {

if (field == null) {

field = SingletonDemo()

}

return field

}

@Synchronized

fun get(): SingletonDemo{

return instance!!

}

}

}

大家都知道在使用懒汉式会出现线程安全的问题，需要使用使用同步锁，在Kotlin中，如果你需要将方法声明为同步，需要添加\*\*@Synchronized\*\*注解。

## 4.双重校验锁式（Double Check)

//Java实现

public class SingletonDemo {

private volatile static SingletonDemo instance;

private SingletonDemo(){}

public static SingletonDemo getInstance(){

if(instance==null){

synchronized (SingletonDemo.class){

if(instance==null){

instance=new SingletonDemo();

}

}

}

return instance;

}

}

//kotlin实现

class SingletonDemo private constructor() {

companion object {

val instance: SingletonDemo by lazy(mode = LazyThreadSafetyMode.SYNCHRONIZED) {

SingletonDemo() }

}

}

先来看LazyThreadSafetyMode，这是一个枚举类，支持三种模式：

SYNCHRONIZED 使用锁来确保只有一个线程来求值。

PUBLICATION 允许多个线程来初始化值，但是只有第一个返回的值有效。

NONE 允许多个线程来初始化值，但是行为就不确定了。

哇！小伙伴们惊喜不，感不感动啊。我们居然几行代码就实现了多行的Java代码。其中我们运用到了Kotlin的**延迟属性 Lazy**。

**Lazy**是接受一个 lambda 并返回一个 Lazy 实例的函数，返回的实例可以作为实现延迟属性的委托： 第一次调用 get() 会执行已传递给 lazy() 的 lambda 表达式并记录结果， 后续调用 get() 只是返回记录的结果。

这里还有有两个额外的知识点。

高阶函数，高阶函数是将函数用作参数或返回值的函数（我很纠结我到底讲不讲，哎）。大家还是看这个 ---[高阶函数](https://www.kotlincn.net/docs/reference/lambdas.html)

[委托属性](https://www.kotlincn.net/docs/reference/delegated-properties.html)

如果你了解以上知识点，我们直接来看Lazy的内部实现。

Lazy内部实现

public fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T> =

when (mode) {

LazyThreadSafetyMode.SYNCHRONIZED -> SynchronizedLazyImpl(initializer)

LazyThreadSafetyMode.PUBLICATION -> SafePublicationLazyImpl(initializer)

LazyThreadSafetyMode.NONE -> UnsafeLazyImpl(initializer)

}

观察上述代码，因为我们传入的**mode = LazyThreadSafetyMode.SYNCHRONIZED**， 那么会直接走 SynchronizedLazyImpl，我们继续观察SynchronizedLazyImpl。

Lazy接口

SynchronizedLazyImpl实现了Lazy接口，Lazy具体接口如下：

public interface Lazy<out T> {

//当前实例化对象，一旦实例化后，该对象不会再改变

public val value: T

//返回true表示，已经延迟实例化过了，false 表示，没有被实例化，

//一旦方法返回true，该方法会一直返回true,且不会再继续实例化

public fun isInitialized(): Boolean

}

继续查看SynchronizedLazyImpl，具体实现如下：

SynchronizedLazyImpl内部实现

private class SynchronizedLazyImpl<out T>(initializer: () -> T, lock: Any? = null) : Lazy<T>, Serializable {

private var initializer: (() -> T)? = initializer

@Volatile private var \_value: Any? = UNINITIALIZED\_VALUE

// final field is required to enable safe publication of constructed instance

private val lock = lock ?: this

override val value: T

get() {

val \_v1 = \_value

//判断是否已经初始化过，如果初始化过直接返回，不在调用高级函数内部逻辑

if (\_v1 !== UNINITIALIZED\_VALUE) {

@Suppress("UNCHECKED\_CAST")

return \_v1 as T

}

return synchronized(lock) {

val \_v2 = \_value

if (\_v2 !== UNINITIALIZED\_VALUE) {

@Suppress("UNCHECKED\_CAST") (\_v2 as T)

}

else {

val typedValue = initializer!!()//调用高级函数获取其返回值

\_value = typedValue //将返回值赋值给\_value,用于下次判断时，直接返回高级函数的返回值

initializer = null

typedValue

}

}

}

//省略部分代码

}

通过上述代码，我们发现 SynchronizedLazyImpl 覆盖了Lazy接口的value属性，并且重新了其属性访问器。其具体逻辑与Java的双重检验是类似的。

到里这里其实大家还是肯定有疑问，我这里**只是实例化了SynchronizedLazyImpl对象，并没有进行值的获取，它是怎么拿到高阶函数的返回值呢？**。这里又涉及到了**委托属性**。

委托属性语法是： val/var <属性名>: <类型> by <表达式>。在 by 后面的表达式是该 委托， 因为属性对应的 get()（和 set()）会被委托给它的 getValue() 和 setValue() 方法。 属性的委托不必实现任何的接口，但是需要提供一个 getValue() 函数（和 setValue()——对于 var 属性）。

而Lazy.kt文件中，声明了Lazy接口的getValue扩展函数。故在最终赋值的时候会调用该方法。

@kotlin.internal.InlineOnly

//返回初始化的值。

public inline operator fun <T> Lazy<T>.getValue(thisRef: Any?, property: KProperty<\*>): T = value

复制代码

五、静态内部类式

//Java实现

public class SingletonDemo {

private static class SingletonHolder{

private static SingletonDemo instance=new SingletonDemo();

}

private SingletonDemo(){

System.out.println("Singleton has loaded");

}

public static SingletonDemo getInstance(){

return SingletonHolder.instance;

}

}

//kotlin实现

class SingletonDemo private constructor() {

companion object {

val instance = SingletonHolder.holder

}

private object SingletonHolder {

val holder= SingletonDemo()

}

}

静态内部类的实现方式，也没有什么好说的。Kotlin与Java实现基本雷同。

补充

在该篇文章结束后，有很多小伙伴咨询，如何在Kotlin版的Double Check，给单例添加一个属性，这里我给大家提供了一个实现的方式。（不好意思，最近才抽出时间来解决这个问题）

class SingletonDemo private constructor(private val property: Int) {//这里可以根据实际需求发生改变

companion object {

@Volatile private var instance: SingletonDemo? = null

fun getInstance(property: Int) =

instance ?: synchronized(this) {

instance ?: SingletonDemo(property).also { instance = it }

}

}

}

复制代码

其中关于?:操作符，如果 ?: 左侧表达式非空，就返回其左侧表达式，否则返回右侧表达式。 请注意，当且仅当左侧为空时，才会对右侧表达式求值。

观察代码我们可以发现大致上和我们的Java中的Double check是一样的。

链接：https://juejin.im/post/5acf49a06fb9a028d700ff4d