

# Kotlin Grammer

deepwaterooo

2021 年 12 月 27 日

## 目录

<b>1</b>	<b>空安全</b>	<b>1</b>
1.1	可空类型与非空类型	1
1.2	在条件中检测 <code>null</code>	2
1.3	安全的调用	2
1.4	Elvis 操作符	3
1.5	!! 操作符：非空断言运算符 (!!)	3
1.6	安全的类型转换	3
1.7	可空类型的集合	4
<b>2</b>	<b>Kotlin 中常量和静态方法</b>	<b>4</b>
2.1	常量	4
2.1.1	Java 中：	4
2.1.2	Kotlin 中：	4
2.1.3	引用常量（这里的引用只针对于 java 引用 kotlin 代码）	4
2.2	静态方法	5
2.2.1	引用静态方法（这里的引用只针对于 java 引用 kotlin 代码）	5
<b>3</b>	<b>Kotlin 中的 <code>object</code> 与 <code>companion object</code> 的区别</b>	<b>5</b>
3.1	一、 <code>object</code> 关键字	6
3.1.1	1、对象表达式	6
3.1.2	2、对象声明	6
3.2	二、 <code>companion object</code>	7
3.3	三、在 <code>companion object</code> 中如何调用外部的成员变量	7
3.3.1	3.1 为什么 <code>companion object</code> 中调用不到外部成员变量	7
3.3.2	3.2 怎样解决才能调用到呢？	8
<b>4</b>	<b>伴生对象：A few facts about Companion objects（写得比较深和透彻一点儿）</b>	<b>8</b>
<b>5</b>	<b>Kotlin-查看 Kotlin 编译成 class 后代码的三种方式</b>	<b>11</b>
5.1	在 AndroidStudio 中查看	11
5.2	方法二	11
5.3	Jadx 反编译工具	13
<b>6</b>	<b>Synchronized、Volatile</b>	<b>13</b>
6.1	如何创建线程 Thread	13
6.2	如何使用 Synchronized 同步锁	14
6.2.1	例子 1	14
6.2.2	例子 2	14
6.3	Volatile 关键字	15

6.4	默认赋值	15
6.4.1	默认不为空	15
6.4.2	可以为空	16
6.4.3	默认值	16
6.4.4	两个默认值	17
6.5	构造函数	17
6.6	重载函数 @JvmOverloads	18
7	<b>Kotlin compiler errors and corrections</b>	<b>19</b>
7.1	Function declaration must have a name	19
7.2	This type (BaseActivity) has a constructor, and thus must be initialized here	20
7.3	@override	21
7.4	类中 var 与 val, set get 幕后字段与 field	21
7.5	幕后字段和 field	24
7.5.1	什么是幕后字段?	24
7.5.2	幕后属性大用处	25
7.6	接口	26
7.7	扩展	27
7.7.1	1、代替一些在 java 中需要 util 类辅助来完成的操作举个例子我们需要对 ArrayList 中的元素按位置做交换, 在 java 中通常会写一个静态方法放在某个 util 类中	27
7.7.2	2、给一些方法设置别名	28
7.8	数据类	31
7.9	嵌套类	32
7.9.1	静态内部类	32
7.9.2	内部类	32
7.9.3	匿名内部类	33

# 1 空安全

## 1.1 可空类型与非空类型

- Kotlin 的类型系统旨在消除来自代码空引用的危险，也称为《十亿美元的错误》。
- 许多编程语言（包括 Java）中最常见的陷阱之一，就是访问空引用的成员会导致空引用异常。在 Java 中，这等同于 NullPointerException 或简称 NPE。
- Kotlin 的类型系统旨在从我们的代码中消除 NullPointerException。NPE 的唯一可能的原因可能是：
  - 显式调用 throw NullPointerException();
  - 使用了下文描述的!! 操作符;
  - 有些数据在初始化时不一致，例如当：
    - \* 传递一个在构造函数中出现的未初始化的 this 并用于其他地方（“泄漏 this”）;
    - \* 超类的构造函数调用一个开放成员，该成员在派生中类的实现使用了未初始化的状态;
  - Java 互操作：
    - \* 企图访问平台类型的 null 引用的成员;

- \* 用于具有错误可空性的 Java 互操作的泛型类型，例如一段 Java 代码可能会向 Kotlin 的 `MutableList<String>` 中加入 `null`，这意味着应该使用 `MutableList<String?>` 来处理它；
  - \* 由外部 Java 代码引发的其他问题。
- 在 Kotlin 中，类型系统区分一个引用可以容纳 `null`（可空引用）还是不能容纳（非空引用）。例如，`String` 类型的常规变量不能容纳 `null`：

---

```
var a: String = "ab"
var a: String = "abc" // 默认情况下，常规初始化意味着非空
a = null // 编译错误
```

---

- 如果要允许为空，我们可以声明一个变量为可空字符串，写作 `String?`：

---

```
var b: String? = "abc" // 可以设置为空
b = null // ok
print(b)
```

---

- 现在，如果你调用 `a` 的方法或者访问它的属性，它保证不会导致 `NPE`，这样你就可以放心地使用：

---

```
val l = a.length
```

---

- 但是如果你想访问 `b` 的同一个属性，那么这是不安全的，并且编译器会报告一个错误：

---

```
val l = b.length // 错误：变量“b”可能为空
```

---

- 但是我们还是需要访问该属性，对吧？有几种方式可以做到。

## 1.2 在条件中检测 `null`

- 首先，你可以显式检测 `b` 是否为 `null`，并分别处理两种可能：

---

```
val l = if (b != null) b.length else -1
```

---

- 编译器会跟踪所执行检测的信息，并允许你在 `if` 内部调用 `length`。同时，也支持更复杂（更智能）的条件：

---

```
val b: String? = "Kotlin"
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
// String of length 6
```

---

- 请注意，这只适用于 `b` 是不可变的情况（即在检测和使用之间没有修改过的局部变量，或者不可覆盖并且有幕后字段的 `val` 成员），因为否则可能会发生在检测之后 `b` 又变为 `null` 的情。况

## 1.3 安全的调用

- 你的第二个选择是安全调用操作符，写作`?.`：

---

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length) // 无需安全调用
```

---

- 如果 `b` 非空，就返回 `b.length`，否则返回 `null`，这个表达式的类型是 `Int?`。
- 安全调用在链式调用中很有用。例如，如果一个员工 `Bob` 可能会（或者不会）分配给一个部门，并且可能有另外一个员工是该部门的负责人，那么获取 `Bob` 所在部门负责人（如果有的话）的名字，我们写作：

---

```
bob?.department?.head?.name
```

---

- 如果任意一个属性（环节）为空，这个链式调用就会返回 `null`。
- 如果要只对非空值执行某个操作，安全调用操作符可以与 `let` 一起使用：

---

```
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
    item?.let { println(it) } // 输出 Kotlin 并忽略 null
}
```

---

- 安全调用也可以出现在赋值的左侧。这样，如果调用链中的任何一个接收者为空都会跳过赋值，而右侧的表达式根本不会求值：

---

```
// 如果 `person` 或者 `person.department` 其中之一为空，都不会调用该函数：
person?.department?.head = managersPool.getManager()
```

---

## 1.4 Elvis 操作符

- 当我们有一个可空的引用 `b` 时，我们可以说“如果 `b` 非空，我使用它；否则使用某个非空的值”：

---

```
val l: Int = if (b != null) b.length else -1
val l: Int = if (b != null) b.length else -1
```

---

- 除了完整的 `if`-表达式，这还可以通过 Elvis 操作符表达，写作`?:`：

---

```
val l = b?.length ?: -1
val l = b?.length ?: -1
```

---

- 如果`?:` 左侧表达式非空，`elvis` 操作符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。
- 请注意，因为 `throw` 和 `return` 在 Kotlin 中都是表达式，所以它们也可以用在 `elvis` 操作符右侧。这可能会非常方便，例如，检测函数参数：

---

```
fun foo(node: Node): String? { // <<<==== 返回值可为空
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // .....
}
```

---

## 1.5 !! 操作符：非空断言运算符 (!!)

- 第三种选择是为 NPE 爱好者准备的：非空断言运算符 (!! ) 将任何值转换为非空类型，若该值为空则抛出异常。我们可以写 `b!!`，这会返回一个非空的 `b` 值（例如：在我们例子中的 `String`）或者如果 `b` 为空，就会抛出一个 `NPE` 异常：

---

```
val l = b!!.length
```

---

- 因此，如果你想要一个 `NPE`，你可以得到它，但是你必须显式要求它，否则它不会不期而至。

## 1.6 安全的类型转换

如果对象不是目标类型，那么常规类型转换可能会导致 `ClassCastException`。另一个选择是使用安全的类型转换，如果尝试转换不成功则返回 `null`：

---

```
val aInt: Int? = a as? Int
```

---

## 1.7 可空类型的集合

- 如果你有一个可空类型元素的集合，并且想要过滤非空元素，你可以使用 `filterNotNull` 来实现：

---

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

---

# 2 Kotlin 中常量和静态方法

## 2.1 常量

### 2.1.1 Java 中：

---

```
class StaticDemoActivity {
    public static final String LOAN_TYPE = "loanType";
    public static final String LOAN_TITLE = "loanTitle";
}
```

---

### 2.1.2 Kotlin 中：

---

```
class StaticDemoActivity {
    companion object {
        val LOAN_TYPE = "loanType"
        val LOAN_TITLE = "loanTitle"
    }
}

class StaticDemoActivity {
    companion object StaticParams {
        val LOAN_TYPE = "loanType"
        val LOAN_TITLE = "loanTitle"
    }
}

class StaticDemoActivity {
    companion object {
        const val LOAN_TYPE = "loanType"
        const val LOAN_TITLE = "loanTitle"
    }
}
```

---

- 注: `const` 关键字用来修饰常量, 且只能修饰 `val`, 不能修饰 `var`, `companion object` 的名字可以省略, 可以使用 `Companion` 来指代

### 2.1.3 引用常量 (这里的引用只针对于 `java` 引用 `kotlin` 代码)

- `TestEntity` 类引用 `StaticDemoActivity` 中的常量

---

```
class TestEntity {
    public TestEntity () {
        String title = StaticDemoActivity.Companion.getLOAN_TITLE();
    }
}
class TestEntity {
    public TestEntity () {
        String title = StaticDemoActivity.StaticParams.getLOAN_TITLE();
    }
}
class TestEntity {
    public TestEntity () {
        String title = StaticDemoActivity.LOAN_TITLE;
        String type= StaticDemoActivity.LOAN_TYPE;
    }
}
```

---

## 2.2 静态方法

- Java 代码:

---

```
class StaticDemoActivity {
    public static void test(){
    }
}
```

---

- Kotlin 中:

---

```
class StaticDemoActivity {
    companion object {
        fun test(){
        }
    }
}
class StaticDemoActivity {
    companion object StaticParams{
        fun test() {
        }
    }
}
```

---

### 2.2.1 引用静态方法 (这里的引用只针对于 `java` 引用 `kotlin` 代码)

- `TestEntity` 类引用 `StaticDemoActivity` 中的静态方法

---

```
class TestEntity {
    public TestEntity () {
        StaticDemoActivity.Companion.test();
    }
}
class TestEntity {
    public TestEntity () {
        StaticDemoActivity.StaticParams.test();
    }
}
```

---

- `companion object {}` 中用来修饰静态常量, 或者静态方法, 单例等等

## 3 Kotlin 中的 **object** 与 **companion object** 的区别

- 区别:
  - companion object 类中只能一个声明周期跟类同步, Only one companion object is allowed per class
  - object 没有限制声明更多用在对象声明, 对象表达式使用。
    - \* 可以声明在类里也可以声明在顶级包下 top-level declaration
- 使用原则:
  - 如果想写工具类的功能, 直接创建文件, 写 top-level 「顶层」函数。(声明在包下)
  - 如果需要继承别的类或者实现接口, 就用 object 或 companion object。

### 3.1 一、object 关键字

- object 关键字可以表达两种含义: 一种是对象表达式, 另一种是对象声明。

#### 3.1.1 1、对象表达式

- 继承一个匿名对象

---

```
val textView = findViewById<TextView>(R.id.tv)
textView.setOnClickListener(object : OnClickListener {
    override fun onClick(p0: View?) {
        Toast.makeText(this@TestActivity, " 点击事件生效", Toast.LENGTH_LONG)
    }
})
```

---

- 上面代码其实就是我们经常要给 view 设置的点击事件, OnClickListener 事件是一个匿名类的对象, 用 object 来修饰。

#### 3.1.2 2、对象声明

- 用 object 修饰的类为静态类, 里面的方法和变量都为静态的。

##### 1. 2.1 直接声明类

---

```
object DemoManager {
    private val TAG = "DemoManager"
    fun a() {
        Log.e(TAG, " 此时 object 表示 声明静态内部类")
    }
}
```

---

##### 2. 2.2 声明静态内部类

- 类内部的对象声明, 没有被 inner 修饰的内部类都是静态的

---

```
class DemoManager{
    object MyObject {
        fun a() {
            Log.e(TAG, " 此时 object 表示 直接声明类")
        }
    }
}
```

---

- 如果需要调用 a() 方法
- kotlin 中调用

---

```
fun init() {
    MyObject.a()
}
```

---

- java 中调用

---

```
MyObject.INSTANCE.a();
```

---

## 3.2 二、companion object

- companion object 修饰为伴生对象, 伴生对象在类中只能存在一个, 类似于 java 中的静态方法 Java 中使用类访问静态成员, 静态方法。

---

```
companion object {
    private val TAG = "DemoManager"
    fun b() {
        Log.e(TAG, "此时 companion object 表示 伴生对象")
    }
}
```

---

- kotlin 中调用

---

```
fun init(){
    b()
}
```

---

- java 中调用

---

```
DemoManager.Companion.b();
```

---

- companion object 相关的内容可以查阅 Kotlin 中常量和静态方法这篇文章, 在这里不多在具体描述。

## 3.3 三、在 companion object 中如何调用外部的成员变量

### 3.3.1 3.1 为什么 companion object 中调用不到外部成员变量

---

```
class DemoManager {
    private val MY_TAG = "DemoManager"
    fun init(){
        b()
    }
    companion object {
        fun b() {
            Log.e(MY_TAG, "此时 companion object 表示 伴生对象")
        }
    }
}
```

---

- 在上面代码中 MY\_TAG 是不会被调用到的。
- 原理很简单:



- 在 java 中我们写一个静态方法，如果需要调用成员变量，是无法调用到的

---

```
private String TAG = "MainActivity";
public static void init(){
    Log.e(TAG,"init() ");
}
```

---

- 只有将 TAG 修改为静态成员变量才能调用到

---

```
private static String TAG = "MainActivity";
public static void init(){
    Log.e(TAG,"init() ");
}
```

---

- 由此可以看出来，java 中静态方法调用成员变量，要求成员变量必须是静态的，在 kotlin 中也是一样，所以当 companion object 中调用非静态的成员变量也是调用不到的。

### 3.3.2 3.2 怎样解决才能调用到呢？

---

```
companion object {
    private val MY_TAG = "DemoManager"
    fun b() {
        Log.e(MY_TAG," 此时 companion object 表示 伴生对象")
    }
}
```

---

- 将所引用的成员变量也修饰静态的，这样就可以引用到了。
- 再透彻一点，偏源码与反编译一点儿的：<https://www.cnblogs.com/webor2006/p/11210181.html>
- <https://juejin.cn/post/6844903816446345224>

## 4 伴生对象：A few facts about Companion objects（写得比较深和透彻一点儿）

- Kotlin 给 Java 开发者带来最大改变之一就是废弃了 static 修饰符。与 Java 不同的是在 Kotlin 的类中不允许你声明静态成员或方法。相反，你必须向类中添加 Companion 对象来包装这些静态引用：差异看起来似乎很小，但是它有一些明显的不同。
- 首先，companion 伴生对象是个实际对象的单例实例。你实际上可以在你的类中声明一个单例，并且可以像 companion 伴生对象那样去使用它。这就意味着在实际开发中，你不仅仅只能使用一个静态对象来管理你所有的静态属性！companion 这个关键字实际上只是一个快捷方式，允许你通过类名访问该对象的内容（如果伴生对象存在一个特定的类中，并且只是用到其中的方法或属性名称，那么伴生对象的类名可以省略不写）。就编译而言，下面的 testCompanion 方法中的三行都是有效的语句。

---

```
class TopLevelClass {
    companion object {
        fun doSomeStuff() {
        }
    }
    object FakeCompanion {
        fun doOtherStuff() {
        }
    }
}
```

---

```

}
fun testCompanion() {
    TopLevelClass.doSomeStuff()
    TopLevelClass.Companion.doSomeStuff()
    TopLevelClass.FakeCompanion.doOtherStuff()
}

```

---

- 为了兼容的公平性，`companion` 关键字还提供了更多选项，尤其是与 `Java` 互操作性相关选项。果您尝试在 `Java` 类中编写相同的测试代码，调用方式可能会略有不同：

```

public void testCompanion() {
    TopLevelClass.Companion.doSomeStuff();
    TopLevelClass.FakeCompanion.INSTANCE.doOtherStuff();
}

```

---

- 区别在于: `Companion` 作为 `Java` 代码中静态成员开放 (实际上它是一个对象实例，但是由于它的名称是以大写的 `C` 开头，所以有点存在误导性)，而 `FakeCompanion` 引用了我们的第二个单例对象的类名。在第二个方法调用中，我们需要使用它的 `INSTANCE` 属性来实际访问 `Java` 中的实例 (你可以打开 `IntelliJ IDEA` 或 `AndroidStudio` 中的 "Show Kotlin Bytecode" 菜单栏，并点击里面 "Decompile" 按钮来查看反编译后对应的 `Java` 代码)
- 在这两种情况下 (不管是 `Kotlin` 还是 `Java`)，使用伴生对象 `Companion` 类比 `FakeCompanion` 类那种调用语法更加简短。此外，由于 `Kotlin` 提供一些注解，可以让编译器生成一些简短的调用方式，以便于在 `Java` 代码中依然可以像在 `Kotlin` 中那样简短形式调用。
- `@JvmField` 注解，例如告诉编译器不要生成 `getter` 和 `setter`，而是生成 `Java` 中成员。在伴生对象的作用域内使用该注解标记某个成员，它产生的副作用是标记这个成员不在伴生对象内部作用域，而是作为一个 `Java` 最外层类的静态成员存在。从 `Kotlin` 的角度来看，这没有什么太大区别，但是如果你看一下反编译的字节代码，你就会注意到伴生对象以及他的成员都声明和最外层类的静态成员处于同一级别。
- 另一个有用的注解 `@JvmStatic`。这个注解允许你调用伴生对象中声明的方法就像是调用外层的类的静态方法一样。但是需要注意的是：在这种情况下，方法不会和上面的成员一样移出伴生对象的内部作用域。因为编译器只是向外层类中添加一个额外的静态方法，然后在该方法内部又委托给伴生对象。
- 一起来看一下这个简单的 `Kotlin` 类例子：

```

class MyClass {
    companion object {
        @JvmStatic
        fun aStaticFunction() {}
    }
}

```

---

- 这是相应编译后的 `Java` 简化版代码：

```

public class MyClass {
    public static final MyClass.Companion Companion = new MyClass.Companion();
    fun aStaticFunction() { // 外层类中添加一个额外的静态方法
        Companion.aStaticFunction(); // 方法内部又委托给伴生对象的 aStaticFunction 方法
    }
    public static final class Companion {
        public final void aStaticFunction() {}
    }
}

```

---

- 这里存在一个非常细微的差别，但在某些特殊的情况下可能会出问题。例如，考虑一下 **Dagger** 中的 **module**(模块)。当定义一个 **Dagger** 模块时，你可以使用静态方法去提升性能，但是如果你选择这样做，如果您的模块包含静态方法以外的任何内容，则编译将失败。由于 **Kotlin** 在类中既包含静态方法，也保留了静态伴生对象，因此无法以这种方式编写仅仅包含静态方法的 **Kotlin** 类。
- 但是不要那么快放弃！这并不意味着你不能这样做，只是它需要一个稍微不同的处理方式：在这种特殊的情况下，你可以使用 **Kotlin** 单例 (使用 **object** 对象表达式而不是 **class** 类) 替换含有静态方法的 **Java** 类并在每个方法上使用 **@JvmStatic** 注解。如下例所示：在这种情况下，生成的字节代码不再显示任何伴生对象，静态方法会附加到类中。

---

```
@Module
object MyModule {
    @Provides
    @Singleton
    @JvmStatic
    fun provideSomething(anObject: MyObject): MyInterface {
        return myObject
    }
}
```

---

- 这又让你再一次明白了伴生对象仅仅是单例对象的一个特例。但它至少表明与许多人的认知是相反的，你不一定需要一个伴生对象来维护静态方法或静态变量。你甚至根本不需要一个对象来维护，只要考虑顶层函数或常量：它们将作为静态成员被包含在一个自动生成的类中 (默认情况下，例如 **MyFileKt** 会作为 **MyFile.kt** 文件生成的类名，一般生成类名以 **Kt** 为后缀结尾)
- 我们有点偏离这篇文章的主题了，所以让我们继续回到伴生对象上来。现在你已经了解了伴生对象实质就是对象，也应该意识到它开放了更多的可能性，例如继承和多态。
- 这意味着你的伴生对象并不是没有类型或父类的匿名对象。它不仅可以拥有父类，而且它甚至可以实现接口以及含有对象名。它不需要被称为 **companion**。这就是为什么你可以这样写一个 **Parcelable** 类：

---

```
class ParcelableClass() : Parcelable {
    constructor(parcel: Parcel) : this()
    override fun writeToParcel(parcel: Parcel, flags: Int) {}
    override fun describeContents() = 0
    companion object CREATOR : Parcelable.Creator<ParcelableClass> {
        override fun createFromParcel(parcel: Parcel): ParcelableClass = ParcelableClass(parcel)
        override fun newArray(size: Int): Array<ParcelableClass?> = arrayOfNulls(size)
    }
}
```

---

- 这里，伴生对象名为 **CREATOR**，它实现了 **Android** 中的 **Parcelable.Creator** 接口，允许遵守 **Parcelable** 约定，同时保持比使用 **@JvmField** 注释在伴随对象内添加 **Creator** 对象更直观。**Kotlin** 中引入了 **@Parcelize** 注解，以便于可以获得所有样板代码，但是在这不是重点...
- 为了使它变得更简洁，如果你的伴生对象可以实现接口，它甚至可以使用 **Kotlin** 中的代理来执行此操作：

---

```
class MyObject {
    companion object : Runnable by MyRunnable()
}
```

---

- 这将允许您同时向多个对象中添加静态方法！请注意，伴生对象在这种情况下甚至不需要作用域体，因为它是由代理提供的。
- 最后但同样重要的是，你可以为伴生对象定义扩展函数！这意味着你可以在现有的类中添加静态方法或静态属性，如下例所示：

```
class MyObject {
    companion object
    fun useCompanionExtension() {
        someExtension()
    }
}
fun MyObject.Companion.someExtension() {}//定义扩展函数
```

- 这样做有什么意义？我真的不知道。虽然 Marcin Moskala 建议使用此操作将静态工厂方法以 Companion 的扩展函数的形式添加到类中。
- 总而言之，伴生对象不仅仅是为了给缺少 static 修饰符的使用场景提供解决方案：
- 它们是真正的 Kotlin 对象，包括名称和类型，以及一些额外的功能。
- 他们甚至可以不用仅仅为了提供静态成员或方法场景。可以有更多其他选择，比如他们可以用作单例对象或替代顶层函数的功能。
- 与大多数场景一样，Kotlin 意味着在你设计过程需要有一点点转变，但与 Java 相比，它并没有真正限制你的选择。如果有的话，也会通过提供一些新的、更简洁的方式让你去使用它。
- <https://cloud.tencent.com/developer/article/1381584> 这个某天上午的时候再读一遍就可以了
- <http://www.4k8k.xyz/article/u013064109/89199478> 这个没读，扫一眼

## 5 Kotlin-查看 Kotlin 编译成 class 后代码的三种方式

### 5.1 在 AndroidStudio 中查看

- 第一步：选择你要查看你的 kotlin 文件，然后点击 Tools->Kotlin->Show Kotlin Bytecode
- 第二步：点击左上角的 Decompile 按钮就会弹出你想要的 class 源码

### 5.2 方法二

- 利用命令行 javap [option] \*.class 命令

子命令	输出信息
-l	输出行和变量的表
-public	只输出公共的方法和成员
-protected	只输出 public 和 protected 类和成员
-package	只输出包，public 和 protected 类和成员，这是默认情况
-p -private	输出所有类和成员
-s	输出内部类型签名
-c	对代码进行反汇编。eg: 类中每一个方法内，包含 java 字节码的指令
-verbose	输出栈大小，方法参数的个数
-costants	输出静态 final 常量

- 样码

```
class TestMain {
    public val msg1 = " 类中的变量"
    private val msg2 = " 类中的变量"
    companion object {
        val msg3 = " 伴生对象的变量"
```

```

    }
}
val msg4 = "Kotlin 文件中的变量"
fun sayHello(msg: String) {
    println("msg=$msg")
}
fun main() {
    val msg5 = " 我要看你的 class 文件"
    sayHello(msg5)
}

```

---

- Javap -public 输出结果
  - Javap -public 输出公共的方法和成员
- 输出的结果为

```

// javap -public TestMainKt.class
// Compiled from "TestMain.kt"
public final class com.yobo.yobo_kotlin.TestMainKt {
    public static final java.lang.String getMsg2();
    public static final void sayHello(java.lang.String);
    public static final void main();
    public static void main(java.lang.String[]);
}
// javap -public TestMain.class
// Compiled from "TestMain.kt"
public final class com.yobo.yobo_kotlin.TestMain {
    public static final com.yobo.yobo_kotlin.TestMain$Companion Companion;
    public final java.lang.String getMsg1();
    public com.yobo.yobo_kotlin.TestMain();
    public static final java.lang.String access$getMsg3$cp();
}

```

---

- javap -p -private
  - javap -p -private 显示所有的类和成员
- javap -p -private TestMainKt.class

```

public final class com.yobo.yobo_kotlin.TestMainKt {
    private static final java.lang.String msg4;
    public static final java.lang.String getMsg4();
    public static final void sayHello(java.lang.String);
    public static final void main();
    public static void main(java.lang.String[]);
    static {};
}

```

---

- Javap -c 输出结果
  - Javap -c 对代码进行反汇编。eg: 类中每一个方法内，包含 java 字节码的
- 可以看到虽然是我们整个 class 文件进行了反编译，但是可读性不是很好，这时候我们就推荐第二种方法。

```

public final class com.yobo.yobo_kotlin.TestMainKt {
    public static final java.lang.String getMsg2();
    Code:
        0: getstatic      #13                // Field msg2:Ljava/lang/String;
        3: areturn

    public static final void sayHello(java.lang.String);
    Code:
        0: aload_0

```

```

1: ldc          #17          // String msg
3: invokestatic #23          // Method kotlin/jvm/internal/Intrinsics.checkNotNull:(Ljava/lang/Ob
6: new          #25          // class java/lang/StringBuilder
9: dup
10: invokespecial #29         // Method java/lang/StringBuilder."<init>":()V
13: ldc          #31          // String msg=
15: invokevirtual #35         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringB
18: aload_0
19: invokevirtual #35         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringB
22: invokevirtual #38         // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
25: astore_1
26: iconst_0
27: istore_2
28: getstatic    #44          // Field java/lang/System.out:Ljava/io/PrintStream;
31: aload_1
32: invokevirtual #50         // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
35: return

public static final void main();
Code:
 0: ldc          #56          // String 我要看你的 class 文件
 2: astore_0
 3: aload_0
 4: invokestatic #58          // Method sayHello:(Ljava/lang/String;)V
 7: return

public static void main(java.lang.String[]);
Code:
 0: invokestatic #54          // Method main:()V
 3: return

static {};
Code:
 0: ldc          #8           // String Kotlin 文件中的变量
 2: putstatic    #13         // Field msg2:Ljava/lang/String;
 5: return
}

```

---

## 5.3 Jadx 反编译工具

- 先给出工具的 GitHub 地址，可以按照自己喜欢的方式安装并查看功能文档：
  - <https://github.com/skylot/jadx>

## 6 Synchronized、Volatile

### 6.1 如何创建线程 Thread

在 Kotlin 中，我们仍然可以使用 java 的语法创建一个线程

```

Thread(Runnable {
}).start()
// 或者使用 Lambda 表达式
Thread {
}.start()

```

---

事实上，kotlin 为我们提供了一个简单写法：Kotlin 中可以使用 `thread()` 方法创建新的线程，指定的语句块将在新线程中运行。语法简单，十分易用。

```

fun main() {
    thread {
        Log.d("yanjun", " 开启一个线程")
    }
}

```

---

用法够简单吧。你会好奇 `thread {}` 到底是什么黑科技，其实很简单，语法糖而已。其实是一个 `thread() {}` 函数

---

```
public fun thread(start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit
): Thread {
    val thread = object : Thread() {
        public override fun run() {
            block()
        }
    }
    if (isDaemon) thread.isDaemon = true
    if (priority > 0) thread.priority = priority
    if (name != null) thread.name = name
    if (contextClassLoader != null) thread.contextClassLoader = contextClassLoader
    if (start) thread.start()
    return thread
}
```

---

- 可以看到 `start` 参数默认为 `true`，自动执行线程。当然也可以通过 `name` 字段指定线程的名字：

---

```
fun main() {
    //指定线程的名字，是否自动执行
    thread(start = true, name = "my_thread") {
    }
}
```

---

## 6.2 如何使用 **Synchronized** 同步锁

- 在 Java 中，给一个方法加锁，需要给方法加 `synchronized` 关键字

---

```
public synchronized void run() {
}
```

---

- kotlin 中没有 `synchronized` 关键字，取而代之的是 `@Synchronized` 注解

---

```
class Util {
    @Synchronized
    fun main() {
    }
}
```

---

- 我们把 kotlin 代码反编译一下看看，

---

```
public final class Util {
    public final synchronized void main() {
    }
}
```

---

- 可以看到 `@Synchronized` 注解可以达到 Java `synchronized` 关键字的作用。
- 除此之外，kotlin 在方法内，可以使用 `block` 块

### 6.2.1 例子 1

```
class Util {  
    val lock = Any()  
    fun main() {  
        synchronized(this) {  
        }  
    }  
}
```

- 编译成 java 如下

```
public static final class Util {  
    @NotNull  
    private final Object lock = new Object();  
    @NotNull  
    public final Object getLock() {  
        return this.lock;  
    }  
    public final void main() {  
        synchronized(this) {  
            boolean var2 = false;  
            Unit var4 = Unit.INSTANCE;  
        }  
    }  
}
```

### 6.2.2 例子 2

```
class Util {  
    val lock = Any()  
    fun main() {  
        synchronized(lock) {  
        }  
    }  
}
```

- 编译成 java 如下

```
public static final class Util {  
    @NotNull  
    private final Object lock = new Object();  
    @NotNull  
    public final Object getLock() {  
        return this.lock;  
    }  
    public final void main() {  
        Object var1 = this.lock;  
        synchronized(var1) {  
            boolean var2 = false;  
            Unit var4 = Unit.INSTANCE;  
        }  
    }  
}
```

## 6.3 Volatile 关键字

- 在 kotlin 中没有 volatile 关键字，但是有 @Volatile 注解

```
class Util {  
    @Volatile  
    var lock = Any()  
}
```



- 编译成 java 如下

```
public static final class Util {
    @NotNull
    private volatile Object lock = new Object();
    @NotNull
    public final Object getLock() {
        return this.lock;
    }
    public final void setLock(@NotNull Object var1) {
        Intrinsics.checkNotNullParameter(var1, "<set-?>");
        this.lock = var1;
    }
}
```

## 6.4 默认赋值

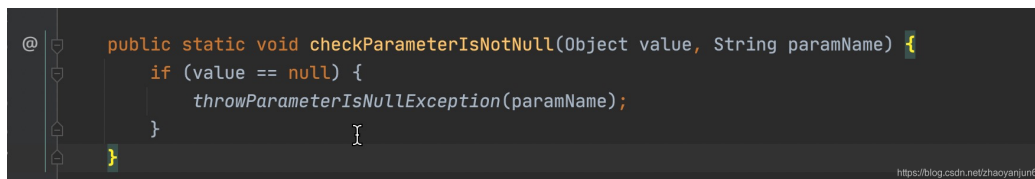
### 6.4.1 默认不为空

```
class A(val name: String, val age: Int)
```

- 代表 name、age 不能为 null，转换成 java，会看到在构造函数中会对 name 字段做空校验。

```
public static final class A {
    @NotNull
    private final String name;
    private final int age;
    @NotNull
    public final String getName() {
        return this.name;
    }
    public final int getAge() {
        return this.age;
    }
    public A(@NotNull String name, int age) {
        Intrinsics.checkNotNullParameter(name, "name");
        super();
        this.name = name;
        this.age = age;
    }
}
```

- 函数 checkNotNullParameter() 源码如下：



```
@
public static void checkParameterIsNotNull(Object value, String paramName) {
    if (value == null) {
        throwParameterIsNullException(paramName);
    }
}
```

https://blog.csdn.net/zhaoyanjund

### 6.4.2 可以为空

```
class A(val name: String?, val age: Int)
```

- 代表 name 可为 null，转换成 java，会看到在构造函数中没有对 name 字段做空校验。

```
public static final class A {
    @Nullable
    private final String name;
    private final int age;
    @Nullable
```

```

        public final String getName() {
            return this.name;
        }
        public final int getAge() {
            return this.age;
        }
        public A(@Nullable String name, int age) {
            this.name = name;
            this.age = age;
        }
    }
}

```

---

### 6.4.3 默认值

```
class A(val name: String? = "zhaoyanjun", val age: Int)
```

---

- name 可为空，如果 name 为 null，使用默认值 “zhaoyanjun”

```

public static final class A {
    @Nullable
    private final String name;
    private final int age;
    @Nullable
    public final String getName() {
        return this.name;
    }
    public final int getAge() {
        return this.age;
    }
    public A(@Nullable String name, int age) {
        this.name = name;
        this.age = age;
    }
    // $FF: synthetic method
    public A(String var1, int var2, int var3, DefaultConstructorMarker var4) {
        if ((var3 & 1) != 0) {
            var1 = "zhaoyanjun";
        }
        this(var1, var2);
    }
}

```

---

### 6.4.4 两个默认值

```
class A(val name: String? = "zhaoyanjun", val age: Int ?= 10)
```

---

- 出来的 java 码:

```

public static final class A {
    @Nullable
    private final String name;
    @Nullable
    private final Integer age;
    @Nullable
    public final String getName() {
        return this.name;
    }
    @Nullable
    public final Integer getAge() {
        return this.age;
    }
    public A(@Nullable String name, @Nullable Integer age) {
        this.name = name;
        this.age = age;
    }
    // $FF: synthetic method

```

```

    public A(String var1, Integer var2, int var3, DefaultConstructorMarker var4) {
        if ((var3 & 1) != 0)
            var1 = "zhaoyanjun";
        if ((var3 & 2) != 0)
            var2 = 10;
        this(var1, var2);
    }
}

```

---

## 6.5 构造函数

```

class A(val name: String, val age: Int)
var a1 = A("zhaoyanjun", 10) //编译正常
var a = A() //编译失败, 因为没有无参构造函数

```

---

```

public static final class A {
    @NotNull
    private final String name;
    private final int age;
    @NotNull
    public final String getName() {
        return this.name;
    }
    public final int getAge() {
        return this.age;
    }
    public A(@NotNull String name, int age) {
        Intrinsic.checkNotNullParameter(name, "name");
        super();
        this.name = name;
        this.age = age;
    }
}

```

---

- 如何才能调用无参构造函数呢？其实很简单，给每个参数添加一个默认值就可以了

```

class A(val name: String? = "", val age: Int? = 0)

```

---

- 只要参数都有默认值，就会默认生成无参构造函数

```

public static final class A {
    @Nullable
    private final String name;
    @Nullable
    private final Integer age;
    @Nullable
    public final String getName() {
        return this.name;
    }
    @Nullable
    public final Integer getAge() {
        return this.age;
    }
    public A(@Nullable String name, @Nullable Integer age) {
        this.name = name;
        this.age = age;
    }
    // $FF: synthetic method
    public A(String var1, Integer var2, int var3, DefaultConstructorMarker var4) {
        if ((var3 & 1) != 0)
            var1 = "";
        if ((var3 & 2) != 0)
            var2 = 0;
        this(var1, var2);
    }
}

```

---

## 6.6 重载函数 @JvmOverloads

```
class A(val name: String, val age: Int)
var a1 = A("zhaoyanjun", 10) // 编译正常
var a2 = A("123") // 编译失败，没有只有一个参数的构造函数
```

- 如何才能自动生成重载函数呢？其实很简单
- 给每个参数添加默认值
- 标记 constructor 关键字
- 标记 @JvmOverloads 关键字

```
class A @JvmOverloads constructor(val name: String? = "", val age: Int? = 0)
```

- 生成的 java 代码如下：

```
public static final class A {
    @Nullable
    private final String name;
    @Nullable
    private final Integer age;
    @Nullable
    public final String getName() {
        return this.name;
    }
    @Nullable
    public final Integer getAge() {
        return this.age;
    }
    @JvmOverloads
    public A(@Nullable String name, @Nullable Integer age) {
        this.name = name;
        this.age = age;
    }
    // $FF: synthetic method
    public A(String var1, Integer var2, int var3, DefaultConstructorMarker var4) {
        if ((var3 & 1) != 0)
            var1 = "";
        if ((var3 & 2) != 0)
            var2 = 0;
        this(var1, var2);
    }
    @JvmOverloads
    public A(@Nullable String name) {
        this(name, (Integer)null, 2, (DefaultConstructorMarker)null);
    }
    @JvmOverloads
    public A() {
        this((String)null, (Integer)null, 3, (DefaultConstructorMarker)null);
    }
}
```

## 7 Kotlin compiler errors and corrections

### 7.1 Function declaration must have a name

- 下文 java 代码因为在 Android Studio 里反编译，出来的都是 scratch 里的 static 类，但它们原本应该不是 static 的才对。特此标注。
- 下现这段代码完全没有问题，安卓中我们常这样

---

```

public class CustomView extends View {
    public CustomView(Context context) {
        super(context);
        init();
    }
    public CustomView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
        init();
    }
    public CustomView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init();
    }
    public void init() {
        //do something
    }
}

```

---

- 然后照葫芦画瓢

---

```

class CustomView : View {
    init {
        //do something
    }
    constructor(ctx: Context) : super(ctx)
    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
    constructor(ctx: Context, attrs: AttributeSet, defStyleAttr: Int) : super(ctx, attrs, defStyleAttr)
}

```

---

- 但是接下来这段，问题就来了：

---

```

class Person {
    init () { // Function declaration must have a name
        print("Person initialized ${firstName}")
    }
    constructor(firstName: String) {
        print("constructor1: ${firstName}")
    }
    constructor(firstName: String, secondName: String) {
        print("constructor2 ${firstName} ${secondName}")
    }
}

```

---

- 怎么改正呢？

## 7.2 This type (BaseActivity) has a constructor, and thus must be initialized here

- kotlin 的继承是通过冒号: 操作符来完成的，相当于 java 中的 extends，所有被继承的类定义时需要加 open 操作符

---

```

open class BaseActivity {
}
class LoginActivity : BaseActivity { // This type (BaseActivity) has a constructor, and thus must be initialized here
}

```

---

- 改下的方法是：
  - error agains: BaseActivity: Supertype initialization is impossible without primary constructor

---

```
open class BaseActivity {
}
class LoginActivity : BaseActivity() { // BaseActivity: Supertype initialization is impossible without primary constructor
    constructor() : super()
}
}
```

---

- 改下的方法是:

---

```
open class BaseActivity {
}
class LoginActivity : BaseActivity { // BaseActivity
    constructor() : super()
}
}
```

---

- OR

---

```
open class BaseActivity {
}
class LoginActivity() : BaseActivity() {
    constructor() : this () { // this: Overload resolution ambiguity. All these functions match.
        // public constructor LoginActivity() defined in Scratch.LoginActivity
        // public constructor LoginActivity() defined in Scratch.LoginActivity
    }
}
}
```

---

- 不过上面的 this 仍然报错了
- 编译成的 java 文件版本为:

---

```
public static class BaseActivity {
}
public static final class LoginActivity extends Scratch.BaseActivity {
}
}
```

---

## 7.3 @override

- 在 kotlin 中只允许覆写父类中加了 open 描述符的方法，这一点与类继承比较相似 (类定义时只有加了 open 操作符才允许被继承);
- java 中覆写方法时一般会加 @override 注解不加也可以通过编译，
- 但是在 kotlin 中并且子类必须在覆写的方法上加上 override，类似于 java 的 @override

---

```
open class Base {
    open fun v() {}
    fun nv() {}
}
class Derived() : Base() {
    // 覆盖
    fun v() {} // v: 'v' hides member of supertype 'Base' and needs 'override' modifier
}
}
```

---

- 加上 override 就可以正常编译了

---

```
open class Base {
    open fun v() {}
    fun nv() {}
}
class Derived() : Base() {
    override fun v() {} // 覆盖
}
}
```

---

- kotlin =>

---

```
public class Base {
    public void v() {
    }
    public final void nv() {
    }
}
public final class Derived extends Base {
    public void v() {
    }
}
```

---

## 7.4 类中 var 与 val, set get 幕后字段与 field

- Kotlin 的类中声明可变的属性使用 var 关键字，如果声明 java 中 final 的属性使用 val 关键字，kotlin 编译的时候会为带 var 关键字的属性自动生成 get 和 set 方法，带 val 关键字的属性只会生成 get 方法

---

```
class Address {
    val name: String = "typ0520"
    var phone: String? = null
    var city: String = "sh"

    fun copyAddress(address: Address): Address {
        val result = Address() // Kotlin 中没有 "new" 关键字
        result.phone = address.phone // 将调用访问器
        result.city = address.city

        print("origin: ${this.name} current: ${address.name}")
        return result
    }
}
```

---

- kotlin =>

---

```
public final class Address {
    @NotNull
    private final String name = "typ0520";
    @Nullable
    private String phone;
    @NotNull
    private String city = "sh";

    @NotNull
    public final String getName() {
        return this.name;
    }

    @Nullable
    public final String getPhone() {
        return this.phone;
    }
    public final void setPhone(@Nullable String var1) {
        this.phone = var1;
    }

    @NotNull
    public final String getCity() {
        return this.city;
    }
    public final void setCity(@NotNull String var1) {
        Intrinsics.checkNotNullParameter(var1, "<set-?>");
        this.city = var1;
    }
}

@NotNull
```

```

    public final Scratch.Address copyAddress(@NotNull Scratch.Address address) {
        Intrinsic.checkNotNullParameter(address, "address");
        Scratch.Address result = new Scratch.Address();
        result.phone = address.phone;
        result.city = address.city; // <<=====
        String var3 = "origin: " + this.name + " current: " + address.name;
        System.out.print(var3);
        return result;
    }
}

```

---

- 这里有两个细节
  - setCity 方法里第一行里会检查 city 是否为 null Intrinsic.checkNotNullParameterIsNotNull(<set-?>, "<set-?>"); 是因为 kotlin 的属性默认是不允许为 null 的, 如果某个属性允许为 null, 定义属性的时候需要在类型后面加上问号? swift 也有可选这个特性不知道它俩谁抄谁的

---

```
var phone: String? = null
```

---

- copyAddress 方法里对常量 name 的引用依然是 this.name、address.name, 如果在 java 中对常量的引用编译以后就会被优化成值 copy, 也就是说

---

```
String str = address.name + " " + this.name
```

---

这段会被编译成

---

```
String str = "typ0520" + " " + "typ0520";
```

---

- 如果对默认的 get 方法和 set 方法实现不满意, kotlin 也提供了机制可以干预

---

```

class Address {
    var city: String = ""
    get() {
        return "city: ${field}"
    }
    set (_city) {
        field = _city // 先别管 field 是啥玩意, 后面会介绍
        print("city: ${field}")
    }
    fun copyAddress(address: Address): Address {
        val result = Address() // Kotlin 中没有 "new" 关键字
        result.city = address.city
        //print("origin: ${this.name} current: ${address.name}") // 会报错: 两个 name
        return result
    }
}

```

---

- kotlin =>

---

```

public static final class Address {
    @NotNull
    private String city = "";
    @NotNull
    public final String getCity() {
        return "city: " + this.city;
    }
    public final void setCity(@NotNull String _city) {
        Intrinsic.checkNotNullParameter(_city, "_city");
        this.city = _city;
        String var2 = "city: " + this.city;
        System.out.print(var2);
    }
}

```



```

@NotNull
public final Scratch.Address copyAddress(@NotNull Scratch.Address address) {
    Intrinsic.checkNotNullParameter(address, "address");
    Scratch.Address result = new Scratch.Address();
    result.setCity(address.getCity()); // <<=====
    return result;
}
}

```

- 对比这两次编译出来的 copyAddress 方法中的内容，第一次对 city 的赋值使用的是

```
result.city = address.city;
```

- 第二次对 city 的赋值使用的是

```
result.setCity(address.getCity());
```

- 这点 kotlin 还是很严谨的，如果发现自定义了 get 和 set 的逻辑就会把直接引用属性的地方替换成调用 get set 方法，
- 但是如果 set{} 和 get{} 代码块中引用了对应的属性这个替换就会出问题，我们来看下如果在 set{} 和 get{} 直接使用 city 字段会被编译成什么

```

class Address {
    var city: String
    get() {
        return "city: ${city}"
    }
    set (_city) {
        city = _city
        print("city: ${city}")
    }
    fun copyAddress(address: Address): Address {
        val result = Address() // Kotlin 中没有 "new" 关键字
        result.city = address.city
        //print("origin: ${this.name} current: ${address.name}") // 会报错: 两个 name
        return result
    }
}

```

- kotlinc ==>

```

public static final class Address {
    @NotNull
    public final String getCity() {
        return "city: " + this.getCity(); // <==
    }
    public final void setCity(@NotNull String _city) {
        Intrinsic.checkNotNullParameter(_city, "_city");
        this.setCity(_city); // <==
        String var2 = "city: " + this.getCity();
        System.out.print(var2);
    }
    @NotNull
    public final Scratch.Address copyAddress(@NotNull Scratch.Address address) {
        Intrinsic.checkNotNullParameter(address, "address");
        Scratch.Address result = new Scratch.Address();
        result.setCity(address.getCity());
        return result;
    }
}

```

- 很明显 setCity 和 getCity 中都出现了死循环，kotlin 为了解决这个问题提供了一个叫做 field 的幕后字段，仅在 set{} get{} 代码块中代替当前的属性

## 7.5 幕后字段和 **field**

### 7.5.1 什么是幕后字段？

- 要理解这个概念，我们得再深一步探究：**getter** 和 **setter** 一定要与某个属性相关联吗？答案当然是否定的。**getter** 是从对象中获取特定的值，这个值完全可能是每次访问时临时计算的，也可能是从其他对象那里得到的；**setter** 也可能是在设置其他对象的属性。事实上，Kotlin 只会为满足特定条件的类属性添加 JVM 意义上的类属性。
- 我们为 **Person** 添加一个 **nameHash** 属性：

---

```
class Person(val name: String) {  
    val nameHash get() = name.hashCode()  
}
```

---

- 这个 **nameHash** 属性并没有初始化，只定义了一个 **getter** 方法。这个属性在编译之后就会消失，只留下一个 **getNameHash()** 方法：

---

```
public final int getNameHash() {  
    return name.hashCode();  
}
```

---

- 我们可以这样说：「**nameHash** 是一个没有幕后字段的属性」，它作为类属性只存在于 Kotlin 代码中，并没有一个真正的 JVM 类属性与之对应。怎么给 **nameHash** 添加一个幕后字段呢？Kotlin 并不允许我们手动添加，但会自动为满足条件的类属性添加幕后字段：
- 使用默认 **getter** / **setter** 的属性，一定有幕后字段。对于 **var** 属性来说，只要 **getter** / **setter** 中有一个使用默认实现，就会生成幕后字段；
- 在自定义 **getter** / **setter** 中使用了 **field** 的属性，一定有幕后字段。这个 **field** 就是我们访问幕后字段的「关键字」，它与 Lambda 表达式中的 **it** 类似，并不是一个真正的关键字，只在特定的语句内有特殊的意思，在其他语句内都不是关键字。
- 比如，我们可以写：

---

```
class Person(val name: String, val gender: Gender, age: Int) {  
    val age = age  
    get() = when (gender) {  
        Gender.FEMALE -> (field * 0.8).toInt()  
        Gender.MALE -> field  
    }  
}  
enum class Gender {  
    MALE, FEMALE  
}
```

---

- 我们这里通过幕后字段实现了 **age** 属性根据性别的不同行为。（**field** 和 **it**）
- 虽然我们为 **age** 属性自定义了 **getter**，但因为在 **getter** 中用了 **field** 关键字，Kotlin 就会为我们生成一个 **private final int age** 属性作为幕后字段。这个幕后字段同时需要初始化，我们就用主构造函数中传入的 **age** 来初始化它。

### 7.5.2 幕后属性大用处

- 很多时候，我们希望定义这样的属性：
- 对外表现为 **val** 属性，只能读不能写；

- 在类内部表现为 `var` 属性，也就是说只能在类内部改变它的值。
- 这在 Kotlin 集合框架中应用十分广泛。比如 `Collection` 接口定义的 `size` 属性就是一个 `val` 属性，对外只读；但对于一个 `MutableCollection` 来说，`size` 的值在内部一定是能改变的，也只允许内部修改它。我们可以这样设计：

---

```
override val size get() = _size
private var _size: Int = 0
```

---

- 我们在内部增删元素时，改动的就是 `_size` 属性的值；外部只能访问到 `size` 属性，不能修改 `_size` 的值。这里的 `_size` 就叫做「幕后属性」。
- `kotlin.collections.AbstractMap.kt` 中也用到了幕后属性，我们看一下：

---

```
private @Volatile var _keys: Set<K>? = null
override val keys: Set<K> get() {
    if (_keys == null) {
        _keys = object : AbstractSet<K>() {
            override operator fun contains(element: K): Boolean = containsKey(element)
            override operator fun iterator(): Iterator<K> {
                val entryIterator = entries.iterator()
                return object : Iterator<K> {
                    override fun hasNext(): Boolean = entryIterator.hasNext()
                    override fun next(): K = entryIterator.next().key
                }
            }
        }
        override val size: Int get() = this@AbstractMap.size
    }
}
return _keys!!
}
```

---

- `keys` 是对外的 API，只读且非空，每次访问都被委托到访问 `_keys` 属性上，`keys` 属性在内部可以改动也可以为 `null`，给我们带来方便。
- 在设计类属性时，可以通过幕后属性灵活地实现功能。

## 7.6 接口

- kotlin 中接口的方法是允许有默认实现的，但是 java 从 1.8 才开始支持接口默认实现，那么 kotlin 编译出来的 class 只能在 1.8 上运行？显然不太可能，我们来看下 kotlin 是怎么处理这件事情的

---

```
interface MyInterface {
    fun bar()
    fun foo() {
        // 可选的方法体
        print("MyInterface foo")
        this.bar()
    }
}
class Child : MyInterface {
    override fun bar() {
        // 方法体
    }
}
```

---

- 执行 `kotlinc` 以后会生成三个 class 文件

---

```
Child.class  MyInterface$DefaultImpls.class  MyInterface.class
```

---

- 编成 java

---

```
public interface MyInterface {
    void bar();
    void foo();
    public static final class DefaultImpls { // static
        public static void foo(@NotNull Scratch.MyInterface $this) {
            String var1 = "MyInterface foo";
            System.out.print(var1);
            $this.bar();
        }
    }
}

public static final class Child implements Scratch.MyInterface {
    public void bar() {
    }
    public void foo() {
        Scratch.MyInterface.DefaultImpls.foo(this);
    }
}
```

---

- kotlin 编译的时候如果发现接口中的某个方法有默认实现,就会生成一个以接口名 +\$+DefaultImpls 方式命名的类, 然后会创建一个同名的静态方法以容纳接口中对应方法的 code, 并且这个静态方法的参数列表中会多加一个名字叫 \$this 类型是相应接口的变量, 这样子类中这个方法的实现就可以直接调用这个静态方法了
- 如果一个类实现了多个接口, 就有可能出现接口中方法冲突的问题, 可以通过 super<XXX> 语法来选择对应父类的方法

---

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
    override fun bar() {
        super<B>.bar()
    }
}
```

---

- kotlin =>

---

```
public interface A {
    void foo();
    void bar();
    public static final class DefaultImpls {
        public static void foo(@NotNull Scratch.A $this) {
            String var1 = "A";
            System.out.print(var1);
        }
    }
}

public interface B {
    void foo();
    void bar();
    public static final class DefaultImpls {
        public static void foo(@NotNull Scratch.B $this) {
            String var1 = "B";
            System.out.print(var1);
        }
    }
}
```

---

```

        public static void bar(@NotNull Scratch.B $this) {
            String var1 = "bar";
            System.out.print(var1);
        }
    }
}
public static final class D implements Scratch.A, Scratch.B {
    public void foo() {
        Scratch.A.DefaultImpls.foo(this);
        Scratch.B.DefaultImpls.foo(this);
    }
    public void bar() {
        Scratch.B.DefaultImpls.bar(this);
    }
}
}

```

---

## 7.7 扩展

- kotlin 提供了一种类似于 objective-c category 的扩展功能, 是继承的一个有力补充, 作用主要有下面这两点

**7.7.1 1、**代替一些在 **java** 中需要 **util** 类辅助来完成的操作举个例子我们需要对 **ArrayList** 中的元素按位置做交换, 在 **java** 中通常会写一个静态方法放在某个 **util** 类中

```

//Utils.java
public class Utils {
    public static void swap(List<Integer> list,int index1,int index2) {
        int idx1Val = list.get(index1);
        int idx2Val = list.get(index2);
        list.remove(index1);
        list.add(index1, idx2Val);
        list.remove(index2);
        list.add(index2, idx1Val);
    }
}

```

---

- 在 kotlin 中可以这样写

```

// extension.kt
fun MutableList<Int>.swap(idx1: Int, idx2: Int) {
    val tmp = this[idx1]
    this[idx1] = this[idx2]
    this[idx2] = tmp
}
fun hello() {
    val list = mutableListOf(1, 2, 3)
    list.swap(0, 2) // “swap()” 内部的 “this” 得到 “l” 的值
}

```

---

- 可以当做 **MutableList** 直接有这个 **swap** 这个方法, 这样比 **Util** 类用起来更加符合面向对象的思维
- 我们来编译下这段 kotlin 代码

```

public final void swap(@NotNull List $this$swap, int idx1, int idx2) {
    Intrinsics.checkNotNullParameter($this$swap, "$this$swap");
    int tmp = ((Number)$this$swap.get(idx1)).intValue();
    $this$swap.set(idx1, $this$swap.get(idx2));
    $this$swap.set(idx2, tmp);
}
public final void hello() {
    List list = CollectionsKt.mutableListOf(new Integer[]{1, 2, 3});
    ((Scratch)this).swap(list, 0, 2);
}

```

---

- 可以看出编译成 class 后还是使用静态方法实现的，并不是真正的改变了 List 的类结构 (如果改变了类结构应该是 list.swap(0, 2))

## 7.7.2 2、给一些方法设置别名

- 玩过 java 的肯定都知道 Object 中的 toString 方法，如果你以前是玩 objective-c 的应该还是习惯叫 description，那么你就可以通过扩展来给 toString 设置别名

---

```
fun Any.description() : String {
    return this.toString()
}
fun hello() {
    val str = "xxxx"
    str.description()
}
```

---

- 处于安全性的考虑扩展的方法是不允许和类中已有的方法冲突的，同时也不允许多个跟对同一个类的扩展中包含相同的方法，来看下两个例子

---

```
class C {
    fun foo() { println("member") }
}
fun C.foo() { // foo: Extension is shadowed by a member: public final fun foo(): Unit
    println("extension")
}
```

---

- kotlin =>

---

```
tmp.kt:4:7: warning: extension is shadowed by a member: public final fun foo(): Unit
fun C.foo() { println("extension") }
```

---



---

```
class C {
    fun foo() { println("member") }
}
fun C.foo2() { println("foo2") }
fun C.foo2() { println("foo2-2") }
```

---

- kotlin =>

---

```
kotlinc -version
info: kotlinc-jvm 1.6.0 (JRE 16.0.2+7-67)
tmp.kt:4:1: error: conflicting overloads: public fun C.foo2(): Unit defined in root package in file tmp.kt, public fun C.foo2() { println("foo2") }
^
tmp.kt:5:1: error: conflicting overloads: public fun C.foo2(): Unit defined in root package in file tmp.kt, public fun C.foo2() { println("foo2-2") }
^
```

---

- 上面应该是旧版上才会出现这种问题，比如我的 ubuntu 18 kotlin -version 1.6.0
- 用新版的 Android studio 1.6.10 就不会再报错
- kotlin 允许在一个类内部为另一个类声明扩展

---

```
class D {
    fun bar() {
        println("D.bar")
    }
    fun hello() {
        println("D hello")
    }
}
```

---

```

    }
}
class C {
    fun baz() {
        println("C.baz")
    }
    fun hello() {
        println("C hello")
    }
    fun D.foo() {
        bar()    // 调用 D.bar
        baz()    // 调用 C.baz
        // 重点看这个这里
        hello()
    }
    fun caller(d: D) {
        d.foo()  // 调用扩展函数
    }
}
fun main() {
    val d = D()
    val c = C()
    c.caller(d)
}

```

---

- 在 D.foo 扩展方法的作用域里存在两个 this，一个是扩展声明所在的类 C、一个是被扩展类 D 的。调用 bar 方法和 baz 方法没什么问题，但是 hello 方法既存在于 C 又存在于 D，那么 kotlin 会选择调用那个 hello 方法呢 (这段代码是可以通过编译的)？我们来编译运行下看下输出

```

kotlinc extension5.kt -include-runtime -d extension5.jar;java -jar extension5.jar=>
D.bar
C.baz
D hello // <<<=====

```

---

- 从输出的日志上可以看出调用的是被扩展类 D 的 hello 方法，我们再看下 extension5.jar 里的 class 内容

```

public static final class C {
    public final void baz() {
        String var1 = "C.baz";
        System.out.println(var1);
    }
    public final void hello() {
        String var1 = "C hello";
        System.out.println(var1);
    }
    public final void foo(@NotNull Scratch.D $this$foo) {
        Intrinsic.checkNotNullParameter($this$foo, "$this$foo");
        $this$foo.bar();
        this.baz();
        $this$foo.hello();
    }
    public final void caller(@NotNull Scratch.D d) {
        Intrinsic.checkNotNullParameter(d, "d");
        this.foo(d);
    }
}

```

---

- 如果想调用 C 类的 hello 方法，可以使用 this@C.hello()

```

class D {
    fun hello() {
        println("D hello")
    }
}

```

```

class C {
    fun hello() { println("C hello") }
    fun D.foo() {
        this@C.hello()
    }
    fun caller(d: D) { d.foo() }
}
fun main(args: Array<String>) {
    val d = D()
    val c = C()
    c.caller(d)
}

```

- =>

```

public static final class C {
    public final void hello() {
        String var1 = "C hello";
        System.out.println(var1);
    }
    public final void foo(@NotNull Scratch.D $this$foo) {
        Intrinsics.checkNotNullParameter($this$foo, "$this$foo");
        this.hello();
    }
    public final void caller(@NotNull Scratch.D d) {
        Intrinsics.checkNotNullParameter(d, "d");
        this.foo(d);
    }
}

```

- 除了可以扩展方法，kotlin 还允许扩展属性

```

val <T> List<T>.lastIndex: Int
    get() = size - 1
fun main(args: Array<String>) {
    val list = listOf(1,2,3,4)
    println(list.lastIndex)
}

```

- =>

```

kotlinc extension7.kt -include-runtime -d extension7.jar;java -jar extension7.jar
3
public final int getLastIndex(@NotNull List $this$lastIndex) {
    Intrinsics.checkNotNullParameter($this$lastIndex, "$this$lastIndex");
    return $this$lastIndex.size() - 1;
}
public final void main(@NotNull String[] args) {
    Intrinsics.checkNotNullParameter(args, "args");
    List list = CollectionsKt.listOf(new Integer[]{1, 2, 3, 4});
    int var3 = ((Scratch)this).getLastIndex(list);
    System.out.println(var3);
}

```

- 可以看出所谓的属性扩展和方法扩展一样都是用静态方法实现的

## 7.8 数据类

- 在 java 开发中充斥着大量的 model 类，除了属性定义其它的基本上都是一些模板代码，虽然 ide 能帮我们生成这些代码，看着还是很不爽的，kotlin 提供了一中叫数据类的玩意，编译器自动从主构造函数中声明的所有属性导出以下成员：

- equals()/hashCode()



- toString() 格式是"User(name=John, age=42)"
- componentN() 函数按声明顺序对应于所有属性
- copy() 函数

---

```
data class User(val name: String, val age: Int)
```

---

- kotlin =>

---

```
public static final class User {
    @NotNull
    private final String name;
    private final int age;
    @NotNull
    public final String getName() {
        return this.name;
    }
    public final int getAge() {
        return this.age;
    }
    public User(@NotNull String name, int age) {
        Intrinsic.checkNotNullParameter(name, "name");
        super();
        this.name = name;
        this.age = age;
    }
    @NotNull
    public final String component1() {
        return this.name;
    }
    public final int component2() {
        return this.age;
    }
    @NotNull
    public final Scratch.User copy(@NotNull String name, int age) {
        Intrinsic.checkNotNullParameter(name, "name");
        return new Scratch.User(name, age);
    }
    // $FF: synthetic method
    public static Scratch.User copy$default(Scratch.User var0, String var1, int var2, int var3, Object var4) {
        if ((var3 & 1) != 0)
            var1 = var0.name;
        if ((var3 & 2) != 0)
            var2 = var0.age;
        return var0.copy(var1, var2);
    }
    @NotNull
    public String toString() {
        return "User(name=" + this.name + ", age=" + this.age + ")";
    }
    public int hashCode() {
        String var10000 = this.name;
        return (var10000 != null ? var10000.hashCode() : 0) * 31 + Integer.hashCode(this.age);
    }
    public boolean equals(@Nullable Object var1) {
        if (this != var1) {
            if (var1 instanceof Scratch.User) {
                Scratch.User var2 = (Scratch.User)var1;
                if (Intrinsic.areEqual(this.name, var2.name) && this.age == var2.age) {
                    return true;
                }
            }
            return false;
        } else
            return true;
    }
}
```

---

## 7.9 嵌套类

- 与 java 类似 kotlin 的类定义也是允许嵌套的，我们分别看下 java 中的静态内部类、内部类、匿名内部类对应 kotlin 的语法

### 7.9.1 静态内部类

---

```
class Outer {  
    private val bar: Int = 1  
    class Nested {  
        fun foo() = 2  
    }  
}  
val demo = Outer.Nested().foo() // == 2
```

---

- java

---

```
private final int demo = (new Scratch.Outer.Nested()).foo();  
public final int getDemo() {  
    return this.demo;  
}  
public final class Outer {  
    private final int bar = 1;  
    public static final class Nested {  
        public final int foo() {  
            return 2;  
        }  
    }  
}
```

---

### 7.9.2 内部类

---

```
class Outer {  
    private val bar: Int = 1  
    fun hello() {  
        print("hello")  
    }  
    inner class Inner {  
        // fun foo() -> Int { // 这里 lambda 表达式不知道哪里出错了，报错  
        fun foo(): Int {  
            return bar  
        }  
        fun callHello() {  
            hello()  
        }  
    }  
}  
val demo = Outer().Inner().foo() // == 1
```

---

- kotlinc =>

---

```
private final int demo = (new Scratch.Outer().new Inner()).foo();  
public final int getDemo() {  
    return this.demo;  
}  
public static final class Outer {  
    private final int bar = 1;  
    public final void hello() {  
        String var1 = "hello";  
        System.out.print(var1);  
    }  
    public final class Inner {  
        public final int foo() {  
            return Outer.this.bar;  
        }  
    }  
}
```

---

```

        public final void callHello() {
            Outer.this.hello();
        }
    }
}

```

---

### 7.9.3 匿名内部类

---

```

class View {
    fun setOnClickListener(listener: OnClickListener) {
        print("${listener}")
    }
    class OnClickListener {
        fun onClick(view: View) {
        }
    }
}
fun hello() {
    val v = View()
    v.setOnClickListener(object: View.OnClickListener() {
        override fun onClick(view: View) {
        }
    })
}

```

---

- kotlin => 这个应该是需要放到一个安卓项目环境里去反编，暂时抄一下版主的代码，改天再验证一下

```

//View.class
public class View {
    public final void setOnClickListener(@NotNull View.OnClickListener listener) {
        Intrinsics.checkNotNull(listener, "listener");
        String str = String.valueOf(listener);
        System.out.print(str);
    }
    public static class OnClickListener {
        public void onClick(@NotNull View view) {
            Intrinsics.checkNotNull(view, "view");
        }
    }
}
public final class Nested_classes3Kt {
    public static final void hello() {
        View v = new View();
        v.setOnClickListener((View.OnClickListener)new View.OnClickListener() {
            public void onClick(@NotNull View view) {
                Intrinsics.checkNotNull(view, "view");
            }
        });
    }
}
public final class Nested_classes3Kt$hello$1 extends View.OnClickListener {
    public void onClick(@NotNull View view) {
        Intrinsics.checkNotNull(view, "view");
    }
}

```

---

- 匿名类编译的时候会生成一个子类容纳被覆盖的方法以及增加的属性，因此类定义的时候必须要加上 open。