目录

[1、应用风格指南 3](#_Toc23285)

[2、源代码组织 3](#_Toc1393)

[2.1 目录结构 3](#_Toc24967)

[2.2 源文件名称 3](#_Toc19924)

[2.3 源文件组织 3](#_Toc3503)

[2.4 类布局 3](#_Toc27781)

[2.5 接口实现布局 4](#_Toc11969)

[2.6 重载布局 4](#_Toc24064)

[3、命名规则 4](#_Toc30681)

[3.1 函数名 4](#_Toc22334)

[3.2 测试方法的名称 5](#_Toc1838)

[3.3 属性名 5](#_Toc20020)

[3.4 幕后属性的名称 5](#_Toc28949)

[3.5 选择好名称 5](#_Toc27463)

[4、格式化 6](#_Toc13341)

[4.1 横向空白 6](#_Toc21388)

[4.2 冒号 7](#_Toc26333)

[4.3 类头格式化 7](#_Toc31171)

[4.4 修饰符 9](#_Toc13681)

[4.5 注解格式化 9](#_Toc21617)

[4.6 文件注解 9](#_Toc13251)

[4.7 函数格式化 10](#_Toc2439)

[4.8 表达式函数体格式化 10](#_Toc29442)

[4.9 属性格式化 10](#_Toc25711)

[4.10 格式化控制流语句 11](#_Toc31741)

[4.11 方法调用格式化 12](#_Toc20870)

[4.12 链式调用换行 12](#_Toc7488)

[4.13 Lambda表达式格式化 12](#_Toc12865)

[5、文档注释 13](#_Toc13088)

[6、避免重复结构 14](#_Toc14220)

[6.1 Unit 14](#_Toc282)

[6.2 分号 14](#_Toc7214)

[6.3 字符串模版 14](#_Toc12076)

[7、语言特性的惯用法 14](#_Toc10032)

[7.1 不可变性 14](#_Toc21278)

[7.2 默认参数值 15](#_Toc24217)

[7.3 类型别名 15](#_Toc27849)

[7.4 Lambda表达式参数 15](#_Toc21764)

[7.5 在Lambda表达式中返回 15](#_Toc13793)

[7.6 具名参数 15](#_Toc4307)

[7.7 使用条件语句 16](#_Toc26832)

[7.8 if还是when 16](#_Toc16910)

[7.9 在条件中使用可空的Boolean值 16](#_Toc32673)

[7.10 使用循环 16](#_Toc25651)

[7.11 区间上循环 17](#_Toc13069)

[7.12 使用字符串 17](#_Toc4988)

[7.13 函数还是属性 17](#_Toc809)

[7.14 使用扩展函数 17](#_Toc22889)

[7.15 使用中缀函数 18](#_Toc7748)

[7.16 工厂函数 18](#_Toc4755)

[7.17 平台类型 18](#_Toc24565)

[7.18 作用域函数apply/with/run/also/let 19](#_Toc24591)

[8、库的编码规范 19](#_Toc12657)

# 1、应用风格指南

如需根据本风格指南配置IntelliJ格式化程序，请安装Kotlin插件1.2.20或更高版本，转到Settings → Editor → Code Style → Kotlin，点击右上角的Set from...链接，并从菜单中选择Predefined style → Kotlin style guide。

如需验证代码已按风格指南格式化，请转到探查设置(Inspections)并启用Kotlin → Style issues → File is not formatted according to project settings探查项，验证风格指南中描述的其他问题(如命名约定)的附加探查项默认已启用。

# 2、源代码组织

## 2.1 目录结构

在纯Kotlin项目中，推荐的目录结构遵循省略了公共根包的包结构。例如，如果项目中的所有代码都位于org.example.kotlin包及其子包中，那么org.example.kotlin包的文件应该直接放在源代码根目录下，而 org.example.kotlin.network.socket中的文件应该放在源代码根目录下的network/socket子目录中。

|  |  |
| --- | --- |
| 1  2 | 对于JVM平台：Kotlin源文件应当与Java源文件位于同一源文件根目录下，并遵循  相同的目录结构(每个文件应存储在与其package语句对应的目录中)。 |

## 2.2 源文件名称

如果Kotlin文件包含单个类(以及可能相关的顶层声明)，那么文件名应该与该类的名称相同，并追加.kt扩展名。如果文件包含多个类或只包含顶层声明，那么选择一个描述该文件所包含内容的名称，并以此命名该文件，使用首字母大写的驼峰风格(例如ProcessDeclarations.kt)。

文件的名称应该描述文件中代码的作用，因此应避免在文件名中使用诸如"Util"之类的无意义词语。

## 2.3 源文件组织

鼓励多个声明(类、顶级函数或者属性)放在同一个Kotlin源文件中，只要这些声明在语义上彼此紧密关联并且文件保持合理大小(不超过几百行)。

特别是在为类定义与类的所有客户都相关的扩展函数时，请将它们放在与类自身定义相同的地方。而在定义仅对指定客户有意义的扩展函数时，请将它们放在紧挨该客户代码之后，不要只是为了保存"Foo的所有扩展函数"而创建文件。

## 2.4 类布局

通常，一个类的内容按以下顺序排列：

* 属性声明与初始化块；
* 次构造函数；
* 方法声明；
* 伴生对象。

不要按字母顺序或者可见性对方法声明排序，也不要将常规方法与扩展方法分开，而是要把相关的东西放在一起，这样从上到下阅读类的人就能够根据所发生事情的逻辑选择一个顺序(高级别优先，或者相反)，并坚持下去。

将嵌套类放在紧挨使用这些类的代码之后，如果打算在外部使用嵌套类，而且类中并没有引用这些类，那么把它们放到末尾，在伴生对象之后。

## 2.5 接口实现布局

在实现一个接口时，实现成员的顺序应该与该接口的成员顺序相同(如果需要， 还要插入用于实现的额外的私有方法)。

## 2.6 重载布局

在类中总是将重载放在一起。

# 3、命名规则

在Kotlin中，包名与类名的命名规则非常简单：

* 包的名称总是小写且不使用下划线(org.example.project)，通常不鼓励使用多个词的名称，但是如果确实需要使用多个词，可以将它们连接在一起或使用驼峰风格(org.example.myProject)。
* 类与对象的名称以大写字母开头并使用驼峰风格：

|  |  |
| --- | --- |
| 1  2 | open **class** **DeclarationProcessor** { /\*……\*/ }  object EmptyDeclarationProcessor : DeclarationProcessor() { /\*……\*/ } |

## 3.1 函数名

函数、属性与局部变量的名称以小写字母开头、使用驼峰风格而不使用下划线：

|  |  |
| --- | --- |
| 1  2 | **fun** processDeclarations() { /\*……\*/ }  **var** declarationCount = 1 |

例外——用于创建类实例的工厂函数可以与要创建的类具有相同的名称：

|  |  |
| --- | --- |
| 1  2  3 | **abstract** **class** **Foo** { /\*……\*/ }  **class** **FooImpl** : Foo { /\*……\*/ }  **fun** FooImpl(): Foo { **return** FooImpl() } |

## 3.2 测试方法的名称

当且仅当在测试中，可以使用反引号括起来的带空格的方法名，(请注意Android运行时目前不支持这样的方法名)测试代码中也允许方法名使用下划线：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **class** **MyTestCase** {  @Test **fun** 'ensure everything works'() { /\*...\*/ }  @Test **fun** ensureEverythingWorks\_onAndroid() { /\*...\*/ }  } |

## 3.3 属性名

常量名称(标有const的属性，或者保存不可变数据的没有自定义get函数的顶层/对象val属性)应该使用大写、下划线分隔的名称：

|  |  |
| --- | --- |
| 1  2 | const **val** MAX\_COUNT = 8  **val** USER\_NAME\_FIELD = "UserName" |

保存带有行为的对象或者可变数据的顶层/对象属性的名称应该使用驼峰风格名称：

|  |  |
| --- | --- |
| 1 | **val** mutableCollection: MutableSet<String> = HashSet() |

保存单例对象引用的属性的名称可以使用与object声明相同的命名风格：

|  |  |
| --- | --- |
| 1 | **val** PersonComparator: Comparator<Person> = /\*...\*/ |

对于枚举常量，可以使用大写、下划线分隔的名称(enum class Color { RED, GREEN })也可使用首字母大写的常规驼峰名称，具体取决于用途。

## 3.4 幕后属性的名称

如果一个类有两个概念上相同的属性，一个是公共API的一部分，另一个是实现细节，那么使用下划线作为私有属性名称的前缀：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **class** **C** {  **private** **val** \_elementList = mutableListOf<Element>()  **val** elementList: List<Element>  **get**() = \_elementList  } |

## 3.5 选择好名称

类的名称通常是用来解释类是什么的名词或者名词短语——List、PersonReader。

方法的名称通常是动词或动词短语，说明该方法做什么——close、readPersons。 修改对象或者返回一个新对象的名称也应遵循建议，例如sort是对一个集合就地排序，而sorted是返回一个排序后的集合副本。

名称应该表明实体的目的是什么，所以最好避免在名称中使用无意义的单词 (Manager、Wrapper等)。

当使用首字母缩写作为名称的一部分时，如果缩写由两个字母组成，就将其大写(IOStream)；而如果缩写更长一些，就只大写其首字母(XmlFormatter、HttpInputStream)。

# 4、格式化

使用4个空格缩进，不要使用tab。

对于花括号，将左花括号放在结构起始处的行尾，而将右花括号放在与左括结构横向对齐的单独一行：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **if** (elements != **null**) {  **for** (element **in** elements) {  // ……  }  } |

(注意——在Kotlin中，分号是可选的，因此换行很重要。语言设计采用Java风格的花括号格式，如果尝试使用不同的格式化风格，那么可能会遇到意外的行为。)

## 4.1 横向空白

在二元操作符左右留空格(a + b)，例外——不要在"range to"操作符(0..i)左右留空格；不要在一元运算符左右留空格(a++)。

在控制流关键字( if、when、for以及while)与相应的左括号之间留空格。

不要在主构造函数声明、方法声明或者方法调用的左括号之前留空格：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **class** A(**val** x: Int)  **fun** foo(x: Int) { …… }  **fun** bar() {  foo(1)  } |

绝不在(、[之后或者]、)之前留空格。

绝不在.或者?.左右留空格：

|  |  |
| --- | --- |
| 1  2 | foo.bar().filter { it > 2 }.joinToString()  foo?.bar() |

在//之后留一个空格：

|  |  |
| --- | --- |
| 1 | // 这是一条注释 |

不要在用于指定类型参数的尖括号前后留空格：

|  |  |
| --- | --- |
| 1 | **class** **Map**<K, V> { …… } |

不要在::前后留空格：

|  |  |
| --- | --- |
| 1  2 | Foo::**class**  **String**::length |

不要在用于标记可空类型的?前留空格：

|  |  |
| --- | --- |
| 1 | String? |

作为一般规则，避免任何类型的水平对齐，将标识符重命名为不同长度的名称不应该影响声明或者任何用法的格式。

## 4.2 冒号

在以下场景中的:之前留一个空格：

* 当它用于分隔类型与超类型时；
* 当委托给一个超类的构造函数或者同一类的另一个构造函数时；
* 在object关键字之后。

而当分隔声明与其类型时，不要在:之前留空格。

在:之后总要留一个空格：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **abstract** **class** **Foo**<**out** T : Any> : IFoo {  **abstract** **fun** foo(a: Int): T  }  **class** **FooImpl** : Foo() {  constructor(x: String) : **this**(x) { /\*……\*/ }  **val** x = object : IFoo { /\*……\*/ }  } |

## 4.3 类头格式化

具有少数主构造函数参数的类可以写成一行：

|  |  |
| --- | --- |
| 1 | **class** Person(id: Int, name: String) |

具有较长类头的类应该格式化，以使每个主构造函数参数都在带有缩进的独立的行中；另外，右括号应该位于一个新行上；如果使用了继承，那么超类的构造函数调用或者所实现接口的列表应该与右括号位于同一行：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **class** Person(  id: Int,  name: String,  surname: String  ) : Human(id, name) { /\*……\*/ } |

对于多个接口，应该将超类构造函数调用放在首位，然后将每个接口应放在不同的行中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **class** Person(  id: Int,  name: String,  surname: String  ) : Human(id, name),  KotlinMaker { /\*……\*/ } |

对于具有很长超类型列表的类，在冒号后面换行，并横向对齐所有超类型名：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **class** **MyFavouriteVeryLongClassHolder** :  MyLongHolder<MyFavouriteVeryLongClass>(),  SomeOtherInterface,  AndAnotherOne {  **fun** foo() { /\*...\*/ }  } |

为了将类头与类体分隔清楚，当类头很长时，可以在类头后放一空行(如上例所示)或者将左花括号放在独立行上：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **class** **MyFavouriteVeryLongClassHolder** :  MyLongHolder<MyFavouriteVeryLongClass>(),  SomeOtherInterface,  AndAnotherOne  {  **fun** foo() { /\*...\*/ }  } |

构造函数参数使用常规缩进(4个空格)。

|  |  |
| --- | --- |
| 1  2 | 理由：这确保了在主构造函数中声明的属性与  在类体中声明的属性具有相同的缩进。 |

## 4.4 修饰符

如果一个声明有多个修饰符，请始终按照以下顺序安放：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **public** / **protected** / **private** / **internal**  expect / actual  final / open / **abstract** / **sealed** / const  External  **override**  lateinit  tailrec  vararg  suspend  Inner  **enum** / annotation  companion  inline  infix  operator  data |

将所有注解放在修饰符前：

|  |  |
| --- | --- |
| 1  2 | @Named("Foo")  **private** **val** foo: Foo |

除非你在编写库，否则请省略多余的修饰符(例如public)。

## 4.5 注解格式化

注解通常放在单独的行上，在它们所依附的声明之前，并使用相同的缩进：

|  |  |
| --- | --- |
| 1  2 | @Target(AnnotationTarget.PROPERTY)  annotation **class** **JsonExclude** |

无参数的注解可以放在同一行：

|  |  |
| --- | --- |
| 1  2 | @JsonExclude @JvmField  **var** x: String |

无参数的单个注解可以与相应的声明放在同一行：

|  |  |
| --- | --- |
| 1 | @Test **fun** foo() { /\*……\*/ } |

## 4.6 文件注解

文件注解位于文件注释(如果有的话)之后、package语句之前，并且用一个空白行与package分开(为了强调其针对文件而不是包)：

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\* 授权许可、版权以及任何其他内容 \*/  @file:JvmName("FooBar")  **package** **foo.bar** |

## 4.7 函数格式化

如果函数签名不适合单行，请使用以下语法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **fun** longMethodName(  argument: ArgumentType = defaultValue,  argument2: AnotherArgumentType  ): ReturnType {  // 函数体  } |

函数参数使用常规缩进(4个空格)。

|  |  |
| --- | --- |
| 1 | 理由：与构造函数参数一致。 |

对于由单个表达式构成的函数体，优先使用表达式形式：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **fun** foo(): Int { // 不良  **return** 1  }  **fun** foo() = 1 // 良好 |

## 4.8 表达式函数体格式化

如果函数的表达式函数体与函数声明不适合放在同一行，那么将=留在第一行， 将表达式函数体缩进4个空格：

|  |  |
| --- | --- |
| 1  2 | **fun** f(x: String) =  x.length |

## 4.9 属性格式化

对于非常简单的只读属性，请考虑单行格式：

|  |  |
| --- | --- |
| 1 | **val** isEmpty: Boolean **get**() = size == 0 |

对于更复杂的属性，总是将get与set关键字放在不同的行上：

|  |  |
| --- | --- |
| 1  2 | **val** foo: String  **get**() { /\*……\*/ } |

对于具有初始化器的属性，如果初始化器很长，那么在等号后增加一个换行并将初始化器缩进四个空格：

|  |  |
| --- | --- |
| 1  2 | **private** **val** defaultCharset: Charset? =  EncodingRegistry.getInstance().getCharset4PropertiesFiles(file) |

## 4.10 格式化控制流语句

如果if或when语句的条件有多行，那么在语句体外边总是使用大括号；将该条件的每个后续行相对于条件语句起始处缩进4个空格；将该条件的右圆括号与左花括号放在单独一行：

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5 | | **if** (!component.isSyncing &&  !hasAnyKotlinRuntimeInScope(module)  ) {  **return** createKotlinNotConfiguredPanel(module)  } |
| 1 | 理由：对齐整齐并且将条件与语句体分隔清楚。 | |

将else、catch、finally关键字以及do/while循环的while关键字与之前的花括号放在相同的行上：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **if** (condition) {  // 主体  } **else** {  // else部分  }  **try** {  // 主体  } **finally** {  // 清理  } |

在when语句中，如果一个分支不止一行，可以考虑用空行将其与相邻的分支块分开：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **private** **fun** parsePropertyValue(propName: String, token: Token) {  **when** (token) {  **is** Token.ValueToken ->  callback.visitValue(propName, token.value)  Token.LBRACE -> { // ……  }  }  } |

将短分支放在与条件相同的行上，无需花括号：

|  |  |
| --- | --- |
| 1  2  3  4 | **when** (foo) {  **true** -> bar() // 良好  **false** -> { baz() } // 不良  } |

## 4.11 方法调用格式化

在较长参数列表的左括号后添加一个换行符，按4个空格缩进参数，将密切相关的多个参数分在同一行：

|  |  |
| --- | --- |
| 1  2  3  4  5 | drawSquare(  x = 10, y = 10,  width = 100, height = 100,  fill = **true**  ) |

在分隔参数名与值的=左右留空格。

## 4.12 链式调用换行

当对链式调用换行时，将.字符或者?.操作符放在下一行，并带有单倍缩进：

|  |  |
| --- | --- |
| 1  2  3  4 | **val** anchor = owner  ?.firstChild!!  .siblings(forward = **true**)  .dropWhile { it **is** PsiComment || it **is** PsiWhiteSpace } |

调用链的第一个调用通常在换行之前，当然如果能让代码更有意义也可以忽略这点。

## 4.13 Lambda表达式格式化

在Lambda表达式中，应该在花括号左右以及分隔参数与代码体的箭头左右留空格。如果一个调用接受单个Lambda表达式，应该尽可能将其放在圆括号外边传入：

|  |  |
| --- | --- |
| 1 | list.filter { it > 10 } |

如果为Lambda表达式分配一个标签，那么不要在该标签与左花括号之间留空格：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **fun** foo() {  ints.forEach lit@{  // ……  }  } |

在多行的Lambda表达式中声明参数名时，将参数名放在第一行，后跟箭头与换行符：

|  |  |
| --- | --- |
| 1  2  3 | appendCommaSeparated(properties) { prop ->  **val** propertyValue = prop.**get**(obj) // ……  } |

如果参数列表太长而无法放在一行上，请将箭头放在单独一行：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | foo {  context: Context,  environment: Env  ->  context.configureEnv(environment)  } |

# 5、文档注释

对于较长的文档注释，将开头/\*\*放在一个独立行中，并且后续每行都以星号开头：

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\*  \* 这是一条多行  \* 文档注释。  \*/ |

简短注释可以放在一行内：

|  |  |
| --- | --- |
| 1 | /\*\* 这是一条简短文档注释。\*/ |

通常，避免使用@param与@return标记，而是将参数与返回值的描述直接合并到文档注释中，并在提到参数的任何地方加上参数链接，只有当需要不适合放进主文本流程的冗长描述时才应使用@param与@return：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // 避免这样：  /\*\*  \* Returns the absolute value of the given number.  \* @param number The number to return the absolute value for.  \* @return The absolute value.  \*/  **fun** abs(number: Int) { /\*……\*/ }  // 而要这样：  /\*\*  \* Returns the absolute value of the given [number].  \*/  **fun** abs(number: Int) { /\*……\*/ } |

# 6、避免重复结构

一般来说，如果Kotlin中的某种语法结构是可选的并且被IDE高亮为冗余的，那么应该在代码中省略之。为了清楚起见，不要在代码中保留不必要的语法元素。

## 6.1 Unit

如果函数返回Unit，那么应该省略返回类型：

|  |  |
| --- | --- |
| 1  2 | **fun** foo() { // 这里省略了": Unit"  } |

## 6.2 分号

尽可能省略分号。

## 6.3 字符串模版

将简单变量传入到字符串模版中时不要使用花括号，只有用到更长表达式时才使用花括号：

|  |  |
| --- | --- |
| 1 | println("$name has ${children.size} children") |

# 7、语言特性的惯用法

## 7.1 不可变性

优先使用不可变(而不是可变)数据，初始化后未修改的局部变量与属性，总是将其声明为val而不是var。

总是使用不可变集合接口(Collection、List、Set、Map)来声明无需改变的集合，使用工厂函数创建集合实例时，尽可能选用返回不可变集合类型的函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // 不良：使用可变集合类型作为无需改变的值  **fun** validateValue(actualValue: String,  allowedValues: HashSet<String>) {  ……  }  // 良好：使用不可变集合类型  **fun** validateValue(actualValue: String,  allowedValues: Set<String>) {  ……  }  // 不良：arrayListOf()返回ArrayList<T>，  // 这是一个可变集合类型  **val** allowedValues = arrayListOf("a", "b", "c")  // 良好：listOf()返回List<T>  **val** allowedValues = listOf("a", "b", "c") |

## 7.2 默认参数值

优先声明带有默认参数的函数而不是声明重载函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 不良  **fun** foo() = foo("a")  **fun** foo(a: String) { /\*……\*/ }  // 良好  **fun** foo(a: String = "a") { /\*……\*/ } |

## 7.3 类型别名

如果有一个在代码库中多次用到的函数类型或者带有类型参数的类型，那么最好为它定义一个类型别名：

|  |  |
| --- | --- |
| 1  2 | typealias MouseClickHandler = (Any, MouseEvent) -> Unit  typealias PersonIndex = Map<String, Person> |

## 7.4 Lambda表达式参数

在简短、非嵌套的Lambda表达式中建议使用it用法而不是显式声明参数，而在有参数的嵌套Lambda表达式中，始终应该显式声明参数。

## 7.5 在Lambda表达式中返回

避免在Lambda表达式中使用多个返回到标签，请考虑重新组织这样的Lambda表达式使其只有单一退出点，如果这无法做到或者不够清晰，请考虑将Lambda表达式转换为匿名函数。

不要在Lambda表达式的最后一条语句中使用返回到标签。

## 7.6 具名参数

当一个方法接受多个相同的原生类型参数或者多个Boolean类型参数时，请使用具名参数语法，除非在上下文中的所有参数的含义都已绝对清楚：

|  |  |
| --- | --- |
| 1 | drawSquare(x = 10, y = 10, width = 100, height = 100, fill = **true**) |

## 7.7 使用条件语句

优先使用try、if与when的表达形式，例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **return** if (x) foo() **else** bar()  **return** when(x) {  0 -> "zero"  **else** -> "nonzero"  } |

优先选用上述代码而不是：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **if** (x)  **return** foo()  **else**  **return** bar()  **when**(x) {  0 -> **return** "zero"  **else** -> **return** "nonzero"  } |

## 7.8 if还是when

二元条件优先使用if而不是when，不要使用：

|  |  |
| --- | --- |
| 1  2  3  4 | **when** (x) {  **null** -> // ……  **else** -> // ……  } |

而应使用：

|  |  |
| --- | --- |
| 1 | **if** (x == **null**) …… **else** …… |

如果有三个或多个选项时优先使用when。

## 7.9 在条件中使用可空的Boolean值

如果需要在条件语句中用到可空的Boolean，使if (value == true)或if (value == false)检测。

## 7.10 使用循环

优先使用高阶函数(filter、map等)而不是循环，例外——forEach(优先使用常规的for循环，除非forEach的接收者是可空的或者forEach用做长调用链的一部分)。

当在使用多个高阶函数的复杂表达式与循环之间进行选择时，请了解每种情况下所执行操作的开销并且记得考虑性能因素。

## 7.11 区间上循环

使用until函数在一个开区间上循环：

|  |  |
| --- | --- |
| 1  2 | **for** (i **in** 0..n - 1) { /\*……\*/ } // 不良  **for** (i **in** 0 until n) { /\*……\*/ } // 良好 |

## 7.12 使用字符串

优先使用字符串模板而不是字符串拼接；优先使用多行字符串而不是将\n转义序列嵌入到常规字符串字面值中；如需在多行字符串中维护缩进，当生成的字符串不需要任何内部缩进时使用trimIndent，而需要内部缩进时使用trimMargin：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | assertEquals(  """  Foo  Bar  """.trimIndent(),  value  )  **val** a = """if(a > 1) {  | **return** a  |}""".trimMargin() |

## 7.13 函数还是属性

在某些情况下，不带参数的函数可与只读属性互换。虽然语义相似，但是在某种程度上有一些风格上的约定。

底层算法优先使用属性而不是函数：

* 不会抛异常；
* 计算开销小(或者在首次运行时缓存)；
* 如果对象状态没有改变，那么多次调用都会返回相同结果。

## 7.14 使用扩展函数

放手去用扩展函数。每当你有一个主要用于某个对象的函数时，可以考虑使其成为一个以该对象为接收者的扩展函数。为了尽量减少API污染，尽可能地限制扩展函数的可见性。根据需要，使用局部扩展函数、成员扩展函数或者具有私有可视性的顶层扩展函数。

## 7.15 使用中缀函数

一个函数只有用于两个角色类似的对象时才将其声明为中缀函数，良好示例如——and、to、zip，不良示例如——add。

如果一个方法会改动其接收者，那么不要声明为中缀形式。

## 7.16 工厂函数

如果为一个类声明一个工厂函数，那么不要让它与类自身同名，优先使用独特的名称，该名称能表明为何该工厂函数的行为与众不同，只有当确实没有特殊的语义时，才可以使用与该类相同的名称。例如：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **class** Point(**val** x: Double, **val** y: Double) {  companion object {  **fun** fromPolar(angle: Double, radius: Double) = Point(...)  }  } |

如果一个对象有多个重载的构造函数，它们并非调用不同的超类构造函数，并且不能简化为具有默认参数值的单个构造函数，那么优先用工厂函数取代这些重载的构造函数。

## 7.17 平台类型

返回平台类型表达式的公有函数/方法必须显式声明其Kotlin类型：

|  |  |
| --- | --- |
| 1 | **fun** apiCall(): String = MyJavaApi.getProperty("name") |

任何使用平台类型表达式初始化的属性(包级别或类级别)必须显式声明其Kotlin类型：

|  |  |
| --- | --- |
| 1  2  3 | **class** **Person** {  **val** name: String = MyJavaApi.getProperty("name")  } |

使用平台类型表达式初始化的局部值可以有也可以没有类型声明：

|  |  |
| --- | --- |
| 1  2  3  4 | **fun** main() {  **val** name = MyJavaApi.getProperty("name")  println(name)  } |

## 7.18 作用域函数apply/with/run/also/let

Kotlin提供了一系列用来在给定对象上下文中执行代码块的函数——let、run、with、apply以及also。关于不同情况下选择正确作用域函数的准则，请参考作用域函数。

# 8、库的编码规范

在编写库时，建议遵循一组额外的规则以确保API的稳定性：

* 总是显式指定成员的可见性(以避免将声明意外暴露为公有API)；
* 总是显式指定函数返回类型以及属性类型(以避免当实现改变时意外更改返回类型)；
* 为所有公有成员提供KDoc注释，不需要任何新文档的覆盖成员除外(以支持为该库生成文档)