Success!

Menu

- 产品
  - Realm Mobile Platform
  - Realm Mobile Database
- Solutions
  - Realtime Collaboration
  - API Mobilization
  - Offline First
- 价格
- 文档
- Forums
- 新闻
-
  - 日本語
  - 中文
  - 한국어
  - English

main navigation

产品

- Realm Mobile Platform
- Realm Mobile Database

Solutions

- Realtime Collaboration
- API Mobilization
- Offline First

价格

文档

- Realm Mobile Platform
- Realm Object Server
- Realm Java
- Realm Objective-C
- Realm React Native
- Realm Swift
- Realm Xamarin

Forums

新闻

联系我们

Language

- 日本語
- 中文
- 한국어
- English

realm

Menu

Kotlin: Java 6 废土中的一线希望

Michael Pardo
on Oct 19 2015

去年，Java8 发布了，增加了很多新特性和提升，比如lambda，stream。Java 9 的标准也已经在制定了。但是超过半数的 Android 设备仍在运行着 Java 6，我们要怎么才能用上新的现代化语言呢？

在 DroidCon NYC 2015 的这个分享里，Michael Pardo 介绍了 Kotlin: 由 JetBrains 开发出的 JVM 静态语言。Kotlin 由很多新的特性，比如 lambdas，类扩展（class extensions），和 null 安全（null-safty）。它简洁明了，同时由很高的互操作性（interoperable）。

Kotlin (0:00)

大家好，我是 Michael Pardo，今天我要给大家展示一下 Kotlin 这门语言，同时看看他如何让你在 Android 开发的时候更开心，更有效率。

Kotlin 是一个基于 JVM 实现的静态语言。Kotlin 是 JetBrains 创造并在持续维护这门语言，对，就是那个创造了 Android Studio 和 IntelliJ 的公司。

Kotlin 有几个核心的目标：

1. ***简约***：帮你减少实现同一个功能的代码量。
2. ***易懂***：让你的代码更容易阅读，同时易于理解。
3. ***安全***：移除了你可能会犯错误的功能。
4. ***通用***：基于 JVM 和 Javascript，你可以在很多地方运行。

5. ***互操作性***：这就意味着 Kotlin 和 Java 可以相互调用，同时 Jetbrains 的目标是让他们 100% 兼容。

为什么不等 Java 8?

Java 和 Android: 一段历史 (1:11)

我们来看看 Java 和 Android 的历史以及他们的关系。 在 2006 年，Java 6 发布了。几年之后，Android 1 的 Alpha 版本发布了，四年后，Java 7 发布了。Android 在 2 年后紧随其后的开始支持 Java 7。去年，Java 8 又发布了。

你想想，你什么时候才能用上 Java 8？ 可能你学的很快，然后就能用上 Java 8。但是 Android 怎么说都得几年后才能开始支持 Java 8，大家适应 Java 8 又需要很长时间。Android 现在的碎片化很严重，Java 7 只支持 API 19 及以上。如果用了 Java 7，那你的 App 用户群一下子就少了一半。即便我们现在有了 Java 8，100% 的覆盖到了所有的用户设备上，但是 Java 本身还是有些问题的。

我们来看看 Java 的一些问题。

Java 有哪些问题？ (2:22)

- ***空引用（Null references）***：连空引用的发明者都成这是个 billion-dollar 错误（参见）。不论你费多大的功夫，你都无法避免它。因为 Java 的类型系统就是不安全的。
- ***原始类型（Raw types）***：我们在开发的时候总是会为了保持兼容性而卡在范型原始类型的问题上，我们都知道要努力避免 raw type 的警告，但是它们毕竟是在语言层面上的存在，这必定会造成误解和不安全因素。
- ***协变数组（Covariant arrays）***：你可以创建一个 string 类型的数组和一个 object 型的数组，然后把 string 数组分配给 object 数组。这样的代码可以通过编译，但是一旦你尝试在运行时分配一个数给那个数组的时候，他就会在运行时抛出异常。
- Java 8 存在高阶方法（ higher-order functions ），但是他们是通过 ***SAM 类型*** 实现的。SAM 是一个单个抽象方法，每个函数类型都需要一个对应的接口。如果你想要创建一个并不存在的 lambda 的时候或者不存着对应的函数类型的时候，你要自己去创建函数类型作为接口。
- **泛型中的通配符**：诡异的泛型总是难以操作，难以阅读，书写，以及理解。对编译器而言，异常检查也变得很困难。

我们来探索下 Kotlin 是如何解决上面的提到的这些问题的。

Kotlin To The Rescue! (4:24)

刚才我们提到过的这些缺陷，Kotlin 通常直接移除了那些特性。同时它也加了一些新的特性：

- Lambda 表达式
- 数据类 (Data classes)
- 函数字面量和内联函数（Function literals & inline functions）
- 函数扩展 (Extension functions)
- 空安全（Null safety）
- 智能转换（Smart casts）
- 字符串模板（String templates）
- 主构造函数（Primary constructors）
- 类委托（Class delegation）
- 类型推断（Type inference）
- 单例（Singletons）
- 声明点变量（Declaration-site variance）
- 区间表达式（Range expressions）

我们将在这篇文章里提及以上大多数特性。Kotlin 之所以能跟随者 JVM 的生态系统不断地进步，是因为他没有任何限制。它编译出来的正是 JVM 字节码。在 JVM 看来，它就跟其他语言一样样的。事实上，如果你在 IntelliJ 或者 Android Studio 上用 Kotlin 的插件，它自带里一个字节码查看器，可以显示每个方法生成的 JVM 字节码。

Hello, Kotlin (5:19)

我们来看看基本语法。下面是一个最简单的，用 Kotlin 书写的 Hello World：

```
fun main(args: Array<String>): Unit {
  println("Hello, World!")
}
```

只有一个函数和一个 print 语句。不需要包声明和类引用声明。这就是一个 Kotlin Hello World 程序的所有代码。声明函数的关键字是 `fun`，`fun` 后面跟的是函数的名称，然后括号包裹起来的是函数参数，这个跟 Java 类似。

然而，在 Kotlin 里，得把参数名放在前面，参数类型放在后面，用一个冒号隔开。函数的返回类型在最后，这个跟 Java 放在前面形式不太一样。如果一个函数没有返回任何类型，可以返回一个 `Unit` 类型，当然也可以省略。调用 Kotlin 标准库中的函数 `println` 就能打印 Hello World 出来，实际上它最终调用了 Java 的 `system.out.println`。

```
fun main(args: Array<String>) {
  println("Hello, World!")
}
```

接下来，我们来从 “Hello World” 中提取 “World” 这个词，并把这个词放到一个变量中。`var` 关键字后跟的是变量的名称。Kotlin 支持字符串内插入变量，只用在字符串内用 `$` 符号开头，随后跟上输出变量的变量名即可，就像如下这样：

```
fun main(args: Array<String>) {
  var name = "World"
  println("Hello, $name!")
}
```

随后，我们来检查我们是否给 main 函数传递了参数。先来判断这个字符串数组是不是空，如果不为空，我们把第一个字符串分配给 `name` 变量。Kotlin 里有个 `val` 类型的声明方法，类似 Java 里的 `final`，也就是常量。

```
fun main(args: Array<String>) {
  val name = "World"
  if (args.isNotEmpty()) {
    name = args[0]
  }

  println("Hello, $name!")
}
```

在我们编译这个程序的时候，我们遇到一个问题：无法重新分配新的值给一个常量。一种解决方法是用内联的 if-else 方法。Kotlin 里的多数的代码块都支持返回值。如果语句进入了 if 代码块儿，也就是说 args 非空，那么就返回 `arg[0]`，否则返回 “World”。 if-else 语句结束后，就直接赋值给我们之前声明的 `name` 常量，下面的例子就是条件赋值代码块：

```
fun main(args: Array<String>) {.
  val name = if (args.isNotEmpty()) {
    args[0]
  } else {
    "World"
  }

  println("Hello, $name!")
}
```

我们可以把上面的代码用一行来书写，看起来有点像 Java 里的三目运算符。移除掉那些大括号后，看起相当漂亮：

```
val name = if (args.isNotEmpty()) args[0] else "World"
```

类语法 (5:19)

我们来看看类。类的定义要通过 `class` 关键字，跟 Java 里的一样，关键字后是类名。Kotlin 有一个主构造函数，我们可以直接将构造函数参数列表写在类的声明处，还可以直接用 `var` 或者 `val` 关键字将参数声明为成员变量（又称：类属性），如下：

```
class Person(var name: String)
```

继续之前的例子，有了主构造函数以后，我们就不再需要成员变量赋值语句了。在 Kotlin 里创建实例的时候，不必使用 `new` 关键字。你只需要指明创建的类型名就可以创建实例了。

```
class Person(var name: String)

fun main(args: Array<String>) {
  val person = Person("Michael")
  println("Hello, $name!")
}
```

很容易发现，字符串插值实际上是错误的，因为 `name` 指向的是一个不存在的变量了。我们可以用刚才提到的 *字符串插值表达式* ，即用 `$` 符号和大括号包裹想要插入的变量，来修复这个问题：

```
class Person(var name: String)

fun main(args: Array<String>) {
  val person = Person("Michael")
  println("Hello, ${person.name}!")
}
```

下面是 `enum` 类。枚举跟 Java 里的枚举很像。定义一个枚举的方法如下：

```
enum class Language(val greeting: String) {
  EN("Hello"), ES("Hola"), FR("Bonjour")
}
```

我们来给 Person 类增加一个叫 lang 的属性，代表一个人的所说的语言。

```
class Person(var name: String, var lang: Language = Language.EN)
```

Kotlin 支持参数默认值，如上：language 的默认值就是 `Language.EN`，这样就可以在创建实例的时候忽略这个参数，除非你要改变 language 的属性值。我们来把这个例子变得更面向对象一些，给 person 增加一个打招呼的方法，简单地输出特定语言打招呼的方法还有人名：

```
enum class Language(val greeting: String) {
  EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
  fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
  val person = Person("Michael")
  person.greet()
}
```

现在在 main 函数里调用 `person.greet()` 方法，看看是不是很酷？！

集合和迭代 (11:32)

```
val people = listOf(
  Person("Michael"),
  Person("Miguel", Language.SP),
  Person("Michelle", Language.FR)
)
```

我们可以用标准库函数 `listOf` 方法创建一个 `person` 列表。遍历这些 `person` 可以用 `for-in` 关键字：

```
for (person in people) {
  person.greet()
```

```
}
```

随后，我们可以在每次遍历的时候执行 `person.greet()` 方法，甚至可以更简单，直接调用 people 集合的扩展方法 `forEach`，传入一个 lambda 表达式，在表达式里用 `it` 代表每次遍历到的 `person` 对象，然后调用它们的 `greet` 方法。

```
people.forEach { it.greet() }
```

我们来创建两个新的类，每个都传入一个默认的语系。我们可以不再像刚才那样重复声明，可以直接用继承的方法来实现。下面是一个扩展版本的 Hello World。展示了很多 Kotlin 的特性：

```
enum class Language(val greeting: String) {
  EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
  fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
  listOf(
    Person("Michael"),
    Hispanophone("Miguel"),
    Francophone("Michelle")
  ).forEach { it.greet() }
}
```

Kotlin 在 Java 上加了什么特性？ (13:11)

下面，我们来看看 Kotlin 在 Java 之上加了哪些更好用的新特性。

类型推导 (13:18)

你可能在其他语言中看到过类型推导。在 Java 里，我们需要自己声明类型，变量名，以及数值。在 Kotlin 里，顺序有些不一样，你先声明变量名，然后是类型，然后是分配值。很多情况下，你不需要声明类型。一个字符串字面量足以指明这是个字符串类型。字符，整形，长整形，单浮点数，双浮点数，布尔值都是可以无需显性声明类型的。

```
var string: String = ""
var string = ""
var char = ' '

var int = 1
var long = 0L
var float = 0F
var double = 0.0

var boolean = true

var foo = MyFooType()
```

只要 Kotlin 可以推导，这个规则同样适用与其他一些类型。通常，如果是局部变量，当你在声明一个值或者变量的时候你不需要指明类型。在一些无法推导的场景里，你才需要用完整的声明变量语法指明变量类型。

空安全（null-safety） (14:22)

Kotlin 一个强大的特性是空安全。我们来看几个例子：

```
String a  = null;
System.out.println(a.length());
```

在 Java 里，声明一个 string 类型，赋一个 null 给这个变量。一旦我们要打印这个字符串的时候，会在运行时曝出空指针错误，因为我们在尝试去读一个空值。下面是这个问题的 kotlin 写法，我们定义一个空值，但是在我们尝试操作它之前，Kotlin 的编译器就告诉了我们问题所在：

```
val a:String = null
```

曝出的错误是：我们在尝试着给一个非空类型分配一个 null。在 Kotlin 的类型体系里，有空类型和非空类型。类型系统识别出了 string 是一个非空类型，并且阻止编译器让它以空的状态存在。想要让一个变量为空，我们需要在声明后面加一个 `?` 号，同时赋值为 null。

```
val a: String? = null
println(a.length())
```

现在，我们修复了这个问题，继续向下：就像在 Java 里一样，我们尝试打印 stirng 的长度，但是我们遇到了跟 Java 一样的问题，这个字符串有可能为空，不过幸好的是：Kotlin 编译器帮助我们发现了这个问题，而不像 Java 那样，在运行时爆出这个错误。

编译器在长度输出的代码前停止了。想要让编译器编译下去，我们得在调用 length 方法的时候考虑到可能为空的情况，要么赋值给这个 string，要么用一个问号在变量名后，这样，代码执行时在读取变量的时候检查它是否空。

```
val a: String? = null
println(a?.length())
```

如果值是空，则会返回空。如果不是空值，就返回真实的值。print 遇到 null 会输出空。

Ternary Null (16:19)

```
int length = a != null ? a.length() : -1
```

上面的代码你可能在 Java 里见到过。用三目运算符取值，检查是否为空，如果为空则返回真实的长度，否则返回 -1，Kotlin 里又相同的实现：

```
var length = if(a!= null) a.length() else -1
```

如果 a 不是 null， 那么就可以直接读值，否则返回默认值。这里用 *elvis操作符* 实现的简写：

```
var length = a?.length() ?: -1
```

我们用 ?号做了一个内联空检查。如果你还记得刚才我说的，如果 a 是 null，第一个 ? 表达式就会返回 null ,如果 elivs 操作符 左侧是空，那么他就会返回右侧，否则直接返回左侧的值。

智能转换 (17:30)

Kotlin 支持类型智能转换的特性。 如果一个局部对象传入一个类型检查，你可以直接通过这个类型来操作，而不需要再自己做转换，看下面的例子你就明白了：

```
if (x is String) {
  print(x.length())
}
```

我们检查了 x 是不是一个字符串，如果是，就打印它的长度。做类型检查，我们需要用到 is 关键字，其实跟 Java 里的 instanceOf 一样。道理很简单，我们既然通过了类型检查，我们就能把它当做这个类型来使用。

```
if (x !is String) {
  return
}

print(x.size())
```

逆操作也没有问题。上面这个例子检查了是否是一个非字符串变量。如果不是，则直接返回。在我们判断了以后就可以认为它是一个字符串了，我们也无需在做显式的类型转换。

```
if (x !is String || x.size() == 0) {
  return
}
```

上面的例子里，我们检查了一个字符串是否是一个非字符串变量，如果左侧的值是 false，就会调用右侧的 or 判断。 when 语句也适用上面的规则，when 其实是一个增强版的 switch。

```
when(x) {
  is Int -> print(x + 1)
  is String -> print(x.size() + 1)
  is Array<Int> -> print(x.sum())
}
```

我们只要做了类型检查，类型一切都会自动转换。从类型检查到类型自动转换，就是 Kotlin 的智能转换。

字符串模板 (19:07)

很多语言都有字符串模板和字符串插值。下面的例子大概就是你在 Java 里经常用到的：

```
val apples = 4
println("I have " + apples + " apples.")
```

你可以把 apples 变量和其他字符串串联起来。用更符合 Kotlin 风格的方式，你可以用插值的方法，在字符串中用 $ 符号前缀加变量名来代表这个字符串内容。

```
val apples = 4
println("I have $apples apples.")
```

你也可以用下面的表达式：

```
val apples = 4
val bananas = 3

println("I have $apples apples and " + (apples + bananas) + " fruits.") // Java-esque
println("I have $apples apples and ${apples+bananas} fruits.") // Kotlin
```

在 Java 中，可能最常见的方案是在需要显示个数的地方，用加号操作苹果和香蕉的个数，然后将字符串都串起来。但在 Kotlin 中，可以用前缀 $ 符号加上大括号将操作语句包裹起来代表操作语句的结果。

区间表达式 (20:00)

你可能在其他的语言里见到过这样的表达式。的确， Kotlin 的不少特性是借鉴自其他语言里。下面这个表达式：如果 i 大于等于 1，并且小于等于 10，就将其打印出来。我们检测的范围是 1 到 10。

```
if (1 <= i && i <= 10) {
  println(i)
}
```

其实我们可以用 intRange 函数来完成这个操作。我们传入 1 和 10，然后调用 contains 函数来判断是否在这个范围里。我们打印出 i 即可。

```
if (IntRange(1, 10).contains(i)) {
  println(i)
}
```

这个还可以用扩展函数来实现，1.rangeTo 创建了一个 1 到 10 的 intRange，我们可以用 contain 来判断它。

更完美的而简洁的写法，是用下面的操作符：

```
if(i in 1..10) { ... }
```

.. 是 rangeTo 的一个别名，它实际背后工作原理还是 rangeTo。

我们还可遍历一个区间，比如：可以用 step 关键字来决定每次遍历时候的跳跃幅度：

```
for(i in 1..4 step 2) { ... }
```

也可以逆向迭代，或者逆向遍历并且控制每次的 step：

```
for (i in 4 downTo 1 step 2) { ... }
```

在 Kotlin 里，也可以结合不同的函数来实现你想要的区间遍历。可以遍历很多不同的数据类型，比如创建 strings 或者你自己的类型。只要符合逻辑就行。

高阶函数 (22:55)

很多语言已经支持了高阶函数，比如 Java 8，但是你并不能用上 Java 8。如果你在用 Java 6 或者 Java 7，下面的例子实现了一个具有过滤功能的函数：

```
public interface Function<T, R> {
  R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
  final List<T> filtered = new ArrayList<T>();
  for (T item : items) if (f.call(item)) filtered.add(item);
  return filtered;
}

filter(numbers, new Function<Integer, Boolean>() {
  @Override
  public Boolean call(Integer value) {
    return value % 2 == 0;
  }
});
```

我们首先要声明一个函数接口，接受参数类型为 T，返回类型为 R。我们用接口中的方法遍历操作了目标集合，创建了一个新的列表，把符合条件的过滤了出来。

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
  val filtered = arrayListOf<T>()
  for (item in items) if (f(item)) filtered.add(item)
  return filtered
}
```

上面的代码是在 Kotlin 下的实现，是不是简单很多？我们调用的时候如下：

```
filter(numbers, { value ->
  value % 2 == 0
})
```

你可能也发现了，我们没有定义任何的函数接口，这是因为在 Kotlin 中，函数也是一种数据类型。看到 f:(T) -> Boolean 这个语句了吗？这就是函数类型作为参数的写法，f 是函数别名，T 是函数接受参数，Boolean 是这个函数的返回值。定义完成后，我们随后就能跟调用其他函数一样调用 f。调用 filter 的时候，我们是用 lambda 表达式来传入过滤函数的，即：{value -> value % 2 = 0}。

由于函数类型参数是可以通过函数声明的签名来推导的，所以其实还有下面的一种写法，大括号内就是第二个参数的函数体：

```
filter(numbers) {
  it % 2 == 0
}
```

内联函数 (25:27)

内联函数和高阶函数经常一起见到。在某些场景下，当你用到泛型的时候，你可以给函数加上 inline 关键字。在编译时，它会用 lambda 表达式替换掉整个函数，整个函数的代码会成为内联代码。

如果代码是这样的：

```
inline fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
  val filtered = arrayListOf<T>()
  for (item in items) if (f(item)) filtered.add(item)
  return filtered
}

filter(numbers) { it % 2 == 0 }
```

由 inline 关键字在编译后会变成如下这样：

```
val filtered = arrayListOf<T>()
for (item in items) if (it % 2 == 0) filtered.add(item)
```

这也意味着我们能实现一些常规函数实现不了的。比如：下面这个函数接受一个 lambda 表达式，但并不能直接返回：

```
fun call(f: () -> Unit) {
  f()
}

call {
  return // Not allowed
}
```

但是如果我们的函数变成内联函数，现在我们就能直接返回了，因为它是内联函数，会自动和其他代码混合在一起：

```
inline fun call(f: () -> Unit) {
  f()
}
call {
  return // Now allowed
}
```

内联函数也允许用 `reified` 类型。下面这个例子就是一个真实场景下的函数，通过一个 View 寻找类型为 `T` 的父元素：

```
inline fun <T : Any> View.findViewParent(): T? {
  var parent = getParent()
  while (parent != null && parent !is T) {
    parent = parent.getParent()
  }
  return parent as T // Cast warning
}
```

这个函数还有些问题。由于泛型类型被擦除了，所以我们无法检测类型，即便我们手工来做检查，依然会出现 warning。

解决方案是：我们给函数参数类型加上 `reified` 关键字。因为函数会被编译成内联代码，所以我们现在就能手工检查类型消除警告了：

```
inline fun <reified T : Any> View.findViewParent(): T? {
  var parent = getParent()
  while (parent != null && parent !is T) {
    parent = parent.getParent()
  }
  return parent as T // Type cast allowed
}
```

函数扩展 (27:20)

函数扩展是 Kotlin 最强大的特性之一。下面是一个工具函数，检测 App 是否运行在 Lollipop 或者更高的 Api 之上，它接受一个整数参数：

```
public fun isLollipopOrGreater(code: Int): Boolean {
  return code >= Build.VERSION_CODES.LOLLIPOP
}
```

通过 `被扩展类型.函数` 的写法，就能将函数变成被扩展类型的一部分，写法如下：

```
public fun Int.isLollipopOrGreater(): Boolean {
  return this >= Build.VERSION_CODES.LOLLIPOP
}
```

我们不在需要参数，想要在函数体内调用整数对象需要用 `this` 关键字。下面就是我们的调用方法，我们可以直接在整数类型上调用这个方法：

```
16.isLollipopOrGreater()
```

函数扩展可以是任何整形，字面量或者包装类型，也可以在标记为 `final` 的类上做类似操作。因为扩展函数不是真的给类增加代码，任何人都没有办法去修改一个类，它实际上是创建了一个静态方法，用语法糖来让扩展函数看着像是类自带的方法一样。

Kotlin 在 Java 集合中充分利用了扩展函数，这有一个例子操作集合：

```
final Function<Customer, Order> customerMapper = // ...
final Function<Order, Boolean> orderFilter = // ...
final Function<Order, Float> orderSorter = // ...
final List<Order> vipOrders = sortBy(filter(map(customers,
    customerMapper),
    orderFilter),
    orderSorter);
```

我们对一个 customer 集合，执行了 map， filter， 以及 sort 操作。嵌套的写法混乱而且难以阅读。下面是标准库的扩展函数写法，是不是简洁了很多：

```
val vipOrders = customers
    .map { it.lastOrder }
    .filter { it.total >= 500F }
    .sortBy { it.total }
```

属性 (30:55)

Kotlin 把属性也变成了语言特性。

```
class Customer {
  private String firstName;
  private String lastName;
  private String email;

  public String getFirstName() { return firstName; }
  public String getLastName() { return lastName; }
  public String getEmail() { return email; }

  public void setFirstName(String firstName) { this.firstName = firstName }
  public void setLastName(String lastName) { this.lastName = lastName }
  public void setEmail(String email) { this.email = email }
}
```

上面是一个典型的 Java bean 类。可看到很多成员变量，和很多 `getter`， `setter` 方法，这可是只有三个属性的时候，就生成了这么多代码。来看看 Kotlin 的写法：

```
class Customer {
 var firstName: String = // ...
 var lastName: String = // ...
 var email: String = // ...
}
```

你只需要将成员变量定义成一个变量即可，默认是 `public` 的。编译器会自动生成 `getter` 和 `setter` 方法。

主构造函数 (31:49)

Kotlin 中，类可以拥有多个构造函数，这一点跟 Java 类似。但你也可以有一个主构造函数。下面的例子是我们从上面的例子里衍生出来的，在函数头里添加了一个主构造函数：

在主构造函数里，可以直接用这些参数变量赋值给类的属性，或者用构造代码块来实现初始化。

```
class Customer(firstName: String, lastName: String, email: String) {
  var firstName: String
  var lastName: String
  var email: String

  init {
    this.firstName = firstName
    this.lastName = lastName
    this.email = email
  }
}
```

当然，更好的方法是：直接在主构造函数里定义这些属性，定义的方法是在参数名前加上 `var` 或者 `val` 关键字，`val` 是代表属性是常量。

```
class Customer(
  var firstName: String,
  var lastName: String,
  var email: String)
```

单例 (35:53)

你可能经常会用到单例设计模式。比如一个 Logger 类，在 Java 里，有多种实现单例的写法。

在 Kotlin 里，你只要在 package 级别创建一个 object 即可！不论你在什么域里，你都可以像单例一样调用这个 object。

```
object Singleton
```

比如下面是一个 looger 的写法：

```
object Logger {
  val tag = "TAG"
  fun d(message: String) {
    Log.d(tag, message)
  }
}
```

你可以直接通过 `Logger.D` 的方法来调用 `D` 函数，它在任何地方都是可用的，而且始终只有一个实例。

Companion Objects (37:00)

Kotlin 移除了 `static` 的概念。通常用 `companion object` 来实现类似功能。你可能时常会看到一个 Activity 有一个 静态类型的 string，名叫 `tag`，和一个启动 Activity 的静态方法。Java 中的实现如下：

```
class LaunchActivity extends AppCompatActivity {
  public static final String TAG = LaunchActivity.class.getName();

  public static void start(Context context) {
    context.startActivity(new Intent(context, LaunchActivity.class));
  }
}
```

在 Kotlin 下的实现如下：

```
class LaunchActivity {
  companion object {
    val TAG: String = LaunchActivity::class.simpleName

    fun start(context: Context) {
      context.startActivity(Intent(context, LaunchActivity::class))
    }
  }
}

Timber.v("Starting activity ${LaunchActivity.TAG}")

LaunchActivity.start(context)
```

有了 `companion object` 后，就跟类多了一个单例的对象和方法一样。

类委托 (37:58)

委托是一个大家都知道的设计模式，Kotlin 把委托视为很重要的语言特性。下面是一个在 Java 中典型的委托写法：

```
public class MyList<E> implements List<E> {
  private List<E> delegate;

  public MyList(List<E> delegate) {
    this.delegate = delegate;
  }

 // ...

  public E get(int location) {
    return delegate.get(location)
```

```
  }

  // ...
}
```

我们有一个自己的 lists 实现，通过构造函数将一个 list 存储起来，存在内部的成员变量里，然后在调用相关方法的时候再委托给这个内部变量。下面是在 Kotlin 里的实现：

```
class MyList<E>(list: List<E>) : List<E> by list
```

用 `by` 关键字，我们实现了一个存储 `E` 类型的 list，在调用 `List` 相关的方法时，会自动委托到 list 上。 译者注：参考 Kotlin 官方文档了解更多。

声明点变型（Declaration-Site Variance）(39:03)

这个可能是一个比较容易让人迷惑的主题。首先，我们用一个协变数组来开始我们的例子，下面的代码能够很好的编译：

```
String[] strings = { "hello", "world" };
Object[] objects = strings;
```

string 数组可以正常的赋值给一个 object 数组。但是下面的不行：

```
List<String> strings = Arrays.asList("hello", "world");
List<Object> objects = strings;
```

你不能分配一个 string 类型的 list 给一个 object 类型的 list。因为 list 之间是没有继承关系的。如果你编译这个代码，会得到一个类型不兼容的错误。想要修复这个错误，我们得用到 Java 中的点变型（use-site variance）去声明，所谓的点变型就是在声明 list 可接受类型的时候，用 `extends` 关键字给出参数类型的可接受类型范围，比如类似如下的例子：

译者注：点变型只是一个名字，不要太在意为什么叫这个，简单理解就是类似通配符原理，具体可以查看这个维基页面。

```
public interface List<E> extends Collection<E> {
  public boolean addAll(Collection<? extends E> collection);
  public E get(int location);
}
```

`addAll` 方法可以接受一个参数，参数类型为所有继承自 `E` 的类型，这不是一个具体类型，而是一个类型范围。每次调用 `get` 方法时，依然返回类型 `E`。在 Kotlin 中，你可以用 `out` 关键字来实现类似的功能：

```
public interface List<out E> : Collection<E> {
  public fun get(index: Int): E
}

public interface MutableList<E> : List<E>, MutableCollection<E> {
  override fun addAll(c: Collection<E>): Boolean
}
```

上面的一系列被称为声明点变型，即在声明可接受参数的时候，就声明为它是可变的。比如上面例子：我们声明参数是可以允许所有继承自 `E` 类型的，返回类型也为 `E` 的。

现在，我们有了可变和不可变类型的列表。`可变性(variance)` 其实很简单，就是取决于我们在声明的时候是动作。

译者注：其实不论声明点变型（Declaration-Site Variance） 还是 点变型(Use-site variance) 都是为了实现泛型的类型声明，标注泛型类型可支持的范围，厘清泛型类型上下继承边界。参考 Generic Types。

操作符重载 (41:26)

```
enum class Coin(val cents: Int) {
  PENNY(1),
  NICKEL(5),
  DIME(10),
  QUARTER(25),
}

class Purse(var amount: Float) {
  fun plusAssign(coin: Coin): Unit {
    amount += (coin.cents / 100f)
  }
}

var purse = Purse(1.50f)
purse += Coin.QUARTER // 1.75
purse += Coin.DIME // 1.85
purse += Coin.PENNY // 1.86
```

上面的代码中，我们创建了一个硬币枚举，每个硬币枚举都代表一个特定数额的硬币。我们有一个 `Purse`（钱包） 类， 它拥有一个 `amount` 成员变量，代表钱包里现在有多少钱。我们创建了一个叫做 `plusAssign` 的函数，`plusAssign` 是一个保留关键字。这个函数会重载 `+=` 操作符，也就是说当你在调用 `+=` 符号的时候，就会调用这个函数。

随后，创建一个 `purse` 实例，可以直接用 `+=` 操作来实现给钱包里放钱进去。

让你的项目支持 Kotlin (42:25)

让你的项目支持 Kotlin 其实非常简单，你需要让你的项目里启用 gradle 插件，Kotlin Android 插件，拷贝代码文件，引入标准库即可。

开始 Kotlin 之路

- Kotlin 文档
- 快速入门
- Kotlin 和 Java 的不同

Q&A (43:40)

Q: Kotlin 有没有什么缺点？

Michael: Kotlin 的少数特性可能有性能问题，比如 annotation 处理速度很慢。除此之外强烈推荐。

Q: 支持 Unit Test 么？

Michael: 支持，我们用Robolectric 和 Mockito 来做测试。

Q: Debug 的时候会很顺利吗？

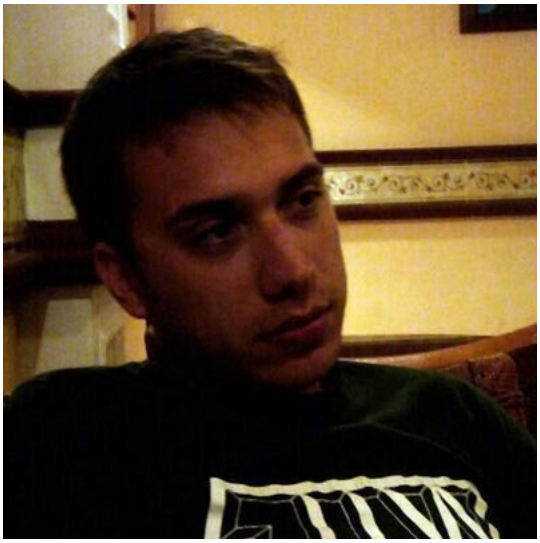
Michael: Kotlin 生成的只是 JVM 字节码而已，而且 Kotlin 和 IntelliJ 共存的非常好，毕竟都是一家做的。我们之前用的 retrolambda，但是他生成的字节码总是有些小问题。

Q: Kotlin 对 Jack and Jill 的支持如何？

Michael: Jack and Jill 和 Kotlin 不能一起工作。

Q: Kotlin 上用反射怎么样？

Michael: Kotlin 上用反射的一大问题是回引入一个很大的库…… 不过，这些你可以用 Java 来做。

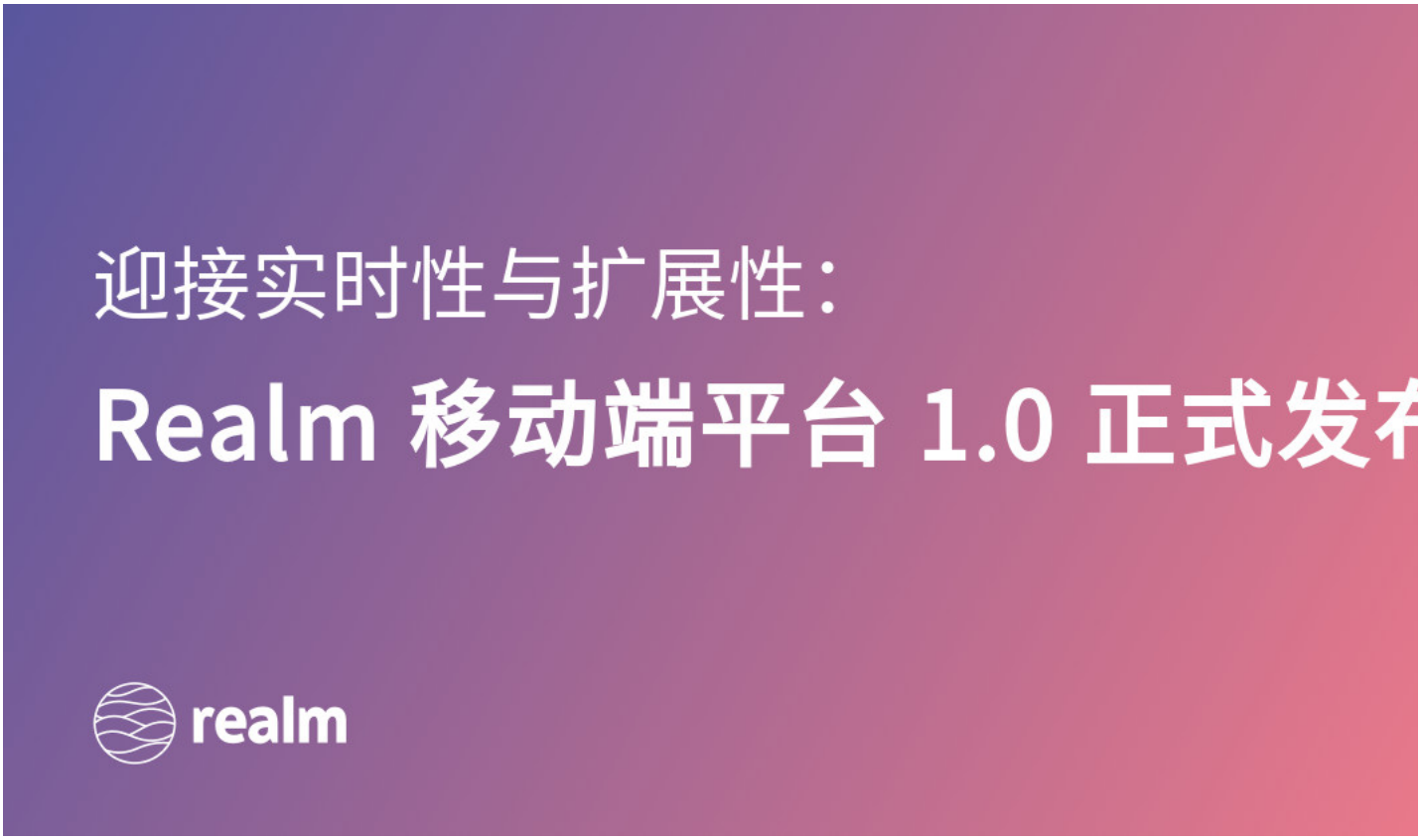


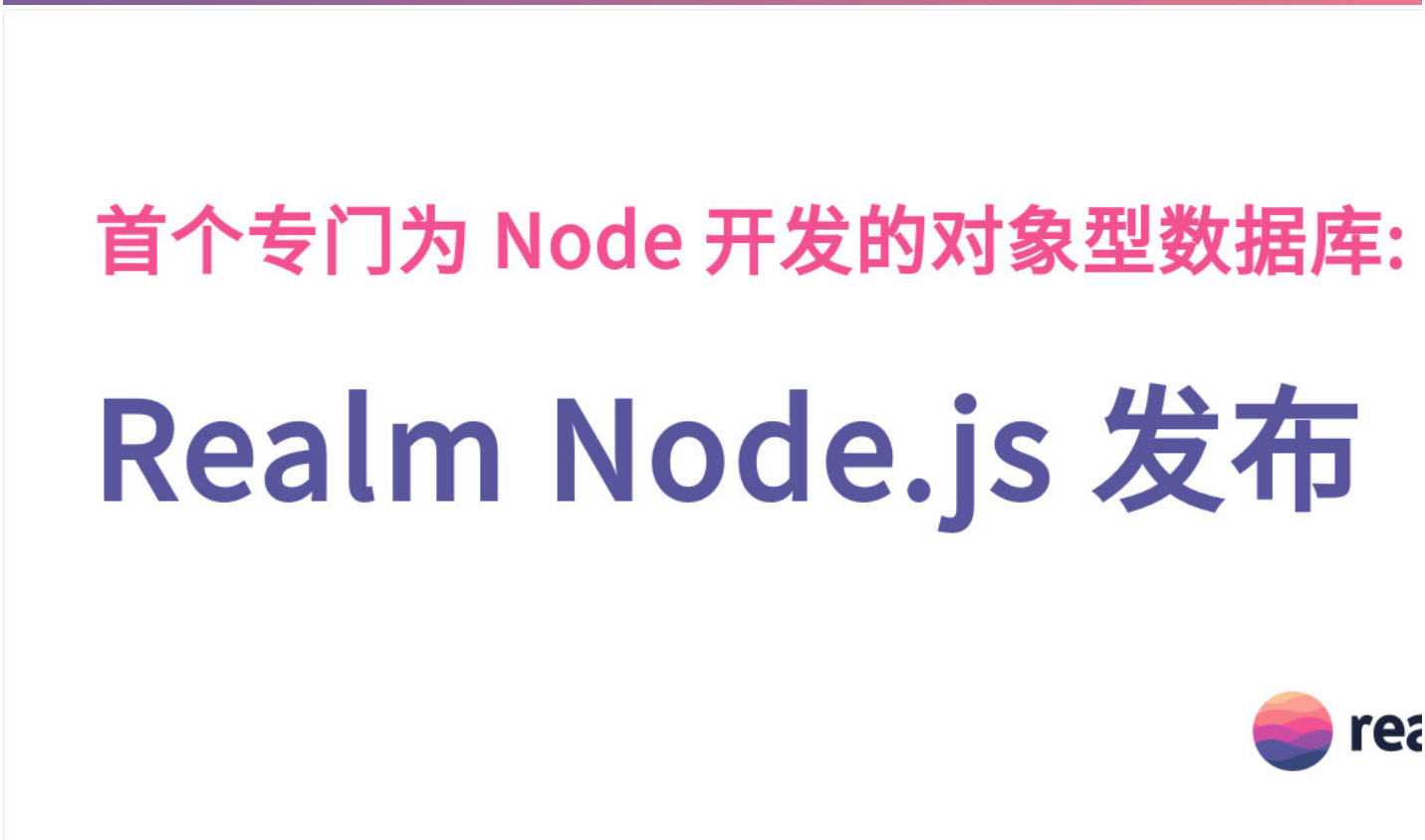
Michael Pardo

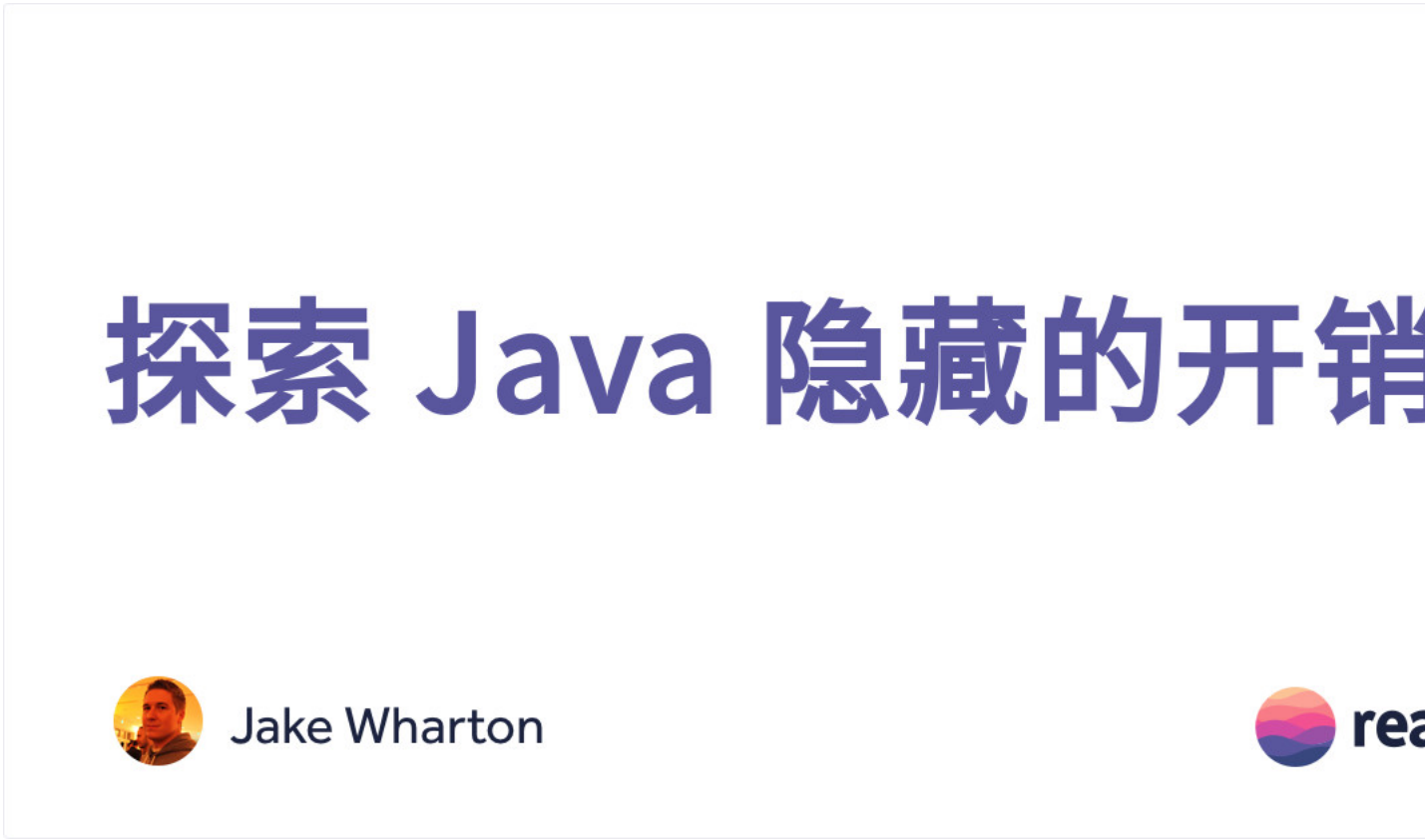
Twitter

新闻精选

新闻精选


迎接实时性与扩展性：
Realm 移动端平台 1.0 正式发布
realm


首个专门为 Node 开发的对象型数据库:
Realm Node.js 发布



更多新闻
Share

-
  -
  -
  -

Subscribe
Get more news like this 订阅

通过邮件获取新闻和 Realm 的更新消息。

分分钟了解如何在 app 中保存数据

文档

产品

- Realm 移动端平台
- Realm 移动端数据库

文档

- Realm 移动端平台
- Realm 对象服务
- Java
- Objective-C
- JavaScript
- Swift
- Xamarin

社区

- Java
- Objective-C
- JavaScript
- Swift
- Xamarin

- 新闻
- 扩展
- Forums

Realm 公司

- 关于
- 工作机会
- 报道我们
- 法律相关
- 价格
- 联系我们

获取 Realm 产品更新信息

输入你的邮箱地址 订阅

-
-
-
-

Realm: Build Better Apps Faster