

kotlin的特点和应用

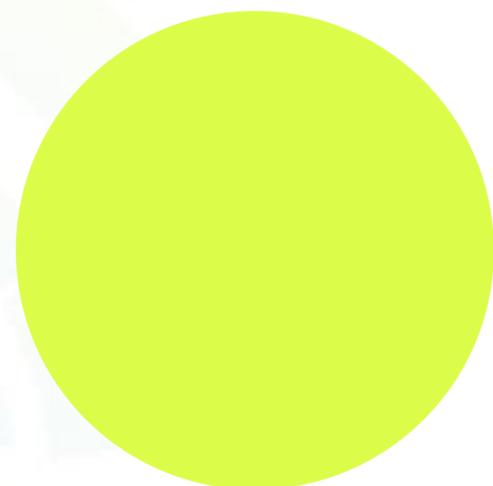
自我介绍



- 姓名：杨洪博
- 部门：平台事业部
- 简要介绍：2015年加入去哪儿，主要从事公司安卓客户端基础功能的研发

目录

- 1 ▶ kotlin简介
- 2 ▶ 比Java舒服的地方
- 3 ▶ Java做不了的地方
- 4 ▶ 做不了java的地方
- 5 ▶ 现有项目转化为kotlin项目的思路



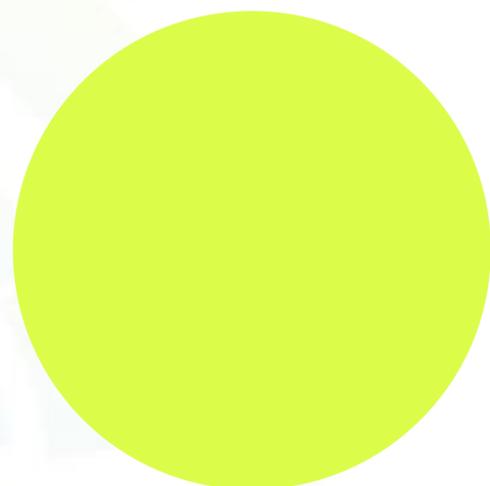
一、Kotlin简介

Kotlin历史

1. 2010年：JetBrains 着手开发 Kotlin。
2. 2011年7月：JetBrains 公开宣布 Kotlin。
3. 2012年2月：JetBrains 在 Apache 2 许可证下开源了 Kotlin 的源码。目前 Kotlin 的官方源代码在 Github 上 <https://github.com/JetBrains/kotlin>。
4. 2016年2月：JetBrains 发布了 Kotlin 1.0 版，算是比较稳定的正式版。许诺之后一直会保持向后兼容。
5. 2017年5月18日 Google I/O 2017 大会宣布 Kotlin 成为了 Android 的官方开发语言
6. 2017年目前最新版本：1.1.2（2017年6月）。

Kotlin属性

1. 基于jvm的静态编程语言
2. 可以编译成Java字节码，也可以编译成JavaScript，方便在没有JVM的设备上运行。
3. 开放源码
4. 允许项目中同时存在Java和Kotlin代码文件
5. 允许Java与Kotlin互调
6. 一切皆是对象



二、为什么要尝试Kotlin

迁移成本低

Kotlin和Java无缝互相调用

Android Studio完美支持

可以直接通过IDE将Java转化为Kotlin

学习曲线很平缓

技术风险小



Google官方支持

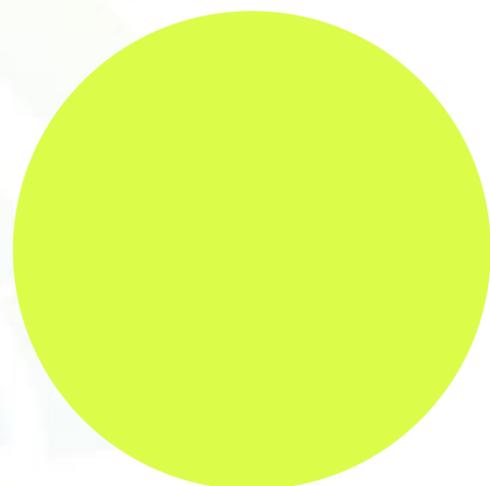
Kotlin和Java可以共存于项目



支持所有Java库，完全兼容Java

都是JVM语言





三、比Java舒服的地方

妈妈再也不用担心空指针异常

Kotlin和Java系统类型之间第一条也是最重要的一条区别就是：kotlin对可空类型的显示支持

```
class TestNull {  
    fun testNullSafeOperator(string: String?) {  
        System.out.println(string?.toCharArray()?.getOrNull(10)?.hashCode())  
    }  
}
```

```
public final class TestNull {  
    public final void testNullSafeOperator(@Nullable String string) {  
        if(string != null) {  
            if(string == null) {  
                throw new TypeCastException("null cannot be cast to non-null type java.lang.String");  
            }  
  
            Intrinsic.checkExpressionValueIsNotNull(string.toCharArray(), "(this as java.lang.String).toCharArray()");  
        } else {  
            Object var10000 = null;  
        }  
    }  
}
```

再也不用写findViewById

```
public class LoginActivity extends BaseFlipActivity {
    Button login;
    TextView textView1;
    TextView textView2;
    TextView textView3;
    TextView textView4;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.fedex_login_page);
        login = (Button) findViewById(R.id.fedex_button_login);
        textView1 = (TextView) findViewById(R.id.textview1);
        textView2 = (TextView) findViewById(R.id.textview1);
        textView3 = (TextView) findViewById(R.id.textview1);
        textView4 = (TextView) findViewById(R.id.textview1);
        login.setText("登录");
        textView1.setText("textview");
        textView2.setText("textview");
        textView3.setText("textview");
        textView4.setText("textview");
    }
}
```

java

```
import kotlinx.android.synthetic.main.fedex_login_page.*

/**
 * Created by seek on 15/12/9.
 */
class LoginActivity : BaseFlipActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.fedex_login_page)
        fedex_button_login.text = "登录"
        textView1.text = "textview"
        textView2.text = "textview"
        textView3.text = "textview"
        textView4.text = "textview"
    }
}
```

Kotlin加import和注释才这么长

再也不用写get、set方法

```
package com.mqunar.upgrader.atom;
public class Artist {
    private long id;
    private String name;
    private String url;
    private String mbid;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getMbid() {
        return mbid;
    }

    public void setMbid(String mbid) {
        this.mbid = mbid;
    }

    @Override public String toString() {
        return "Artist{" +
            "id=" + id +
            ", name=" + name + '\'' +
            ", url=" + url + '\'' +
            ", mbid=" + mbid + '\'' +
            '}';
    }
}
```

java实现一个bean类
都快放不下了

```
package com.mqunar.upgrader.atom

data class Artist(
    var id: Long,
    var name: String,
    var url: String,
    var mbid: String)
```

Kotlin实现一个bean类

简洁-单例

```
public class Obj {  
    private Obj() {  
    }  
  
    private static Obj INSTANCE;  
  
    public static Obj getObj() {  
        if (INSTANCE == null)  
            INSTANCE = new Obj();  
        return INSTANCE;  
    }  
}
```

java懒汉模式单例

```
object Obj{  
    init{  
        println("object init...")  
    }  
}
```

Kotlin一个关键字实现单例

swith everything

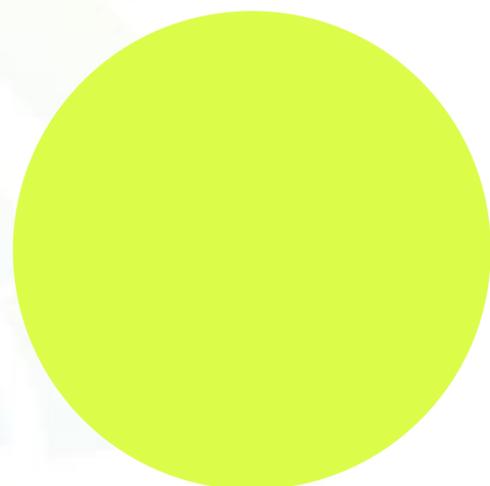
```
int [] a = {1,2,3,4}

if (x > 0 && x < 10) {
    QLog.d("x in 0..10");
}
if ( x > 10 && x < 20){
    QLog.d("x in 10 .. 20");
}
if ( x > 20 && x < 20){
    QLog.d("x in 20 .. 30 ");
}
for (int i : a) {
    if (x == i) {
        QLog.d("x in array");
    }
}
```

java

```
when (x) {
    in 1..10 -> print("x in 0..10")
    in 10..20 -> print("x in 10 .. 20")
    in 20..30 -> print("x in 20 .. 30")
    in intArrayOf(1, 2, 3)-> print("x in array")
}
```

Kotlin when 可以用任意
表达式作为分支条件



三、Java做不了的地方

方法拓展

```
fun Context.longToast(message: String) {  
    Toast.makeText(this, message, Toast.LENGTH_LONG).show()  
}  
applicationContext.longToast("hello world" )  
// 使用扩展函数  
fun View.dp_f(dp: Float): Float {  
    // 引用View的context  
    return TypedValue.applyDimension(  
        TypedValue.COMPLEX_UNIT_DIP, dp, context.resources.displayMetrics)  
}  
// 转换Int  
fun View.dp_i(dp: Float): Int {  
    return dp_f(dp).toInt()  
}
```

扩展函数是静态解析的，也就是，它并不是接收者类型的虚拟成员，意味着调用扩展函数时，调用扩展函数时，具体被调用的的是哪一个函数，由调用函数的的对象表达式来决定的，而不是动态的类型决定的。

属性拓展

使用扩展属性(extension property)即把某些函数添加为数据, 使用"=", 直接设置或使用.

```
var View.padLeft: Int
    set(value) {
        setPadding(value, paddingTop, paddingRight, paddingBottom)
    }
    get() {
        return paddingLeft
    }
v<TextView> {
    layoutParams = LayoutParams(WRAP_CONTENT, WRAP_CONTENT)
    padLeft = dp_i(16) // 是不是强多了
    text = "Hello"
}
```

函数式编程

```
handler.post(new Runnable() {  
    @Override  
    public void run() {  
        //TODO  
    }  
});
```

```
textView.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        //TODO  
    }  
});
```

```
handler.post { //TODO }  
textView.setOnClickListener { //TODO }
```

JAVA匿名内部类

Kotlin函数式编程

lamda表达式

在Kotlin中支持函数式编程，有函数作为参数或返回值的函数在Kotlin中叫做“高阶函数”。

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

Lambda 表达式”(lambda expression)是一个匿名函数，Lambda 表达式基于数学中的 λ 演算得名，直接对应于其中的 lambda 抽象(lambda abstraction)，是一个匿名函数，即没有函数名的函数。Lambda 表达式可以表示闭包（注意和数学传统意义上的不同）。

局部函数作为返回值

```
fun main(args: Any) {  
    val addResult = lateAdd(2, 4)  
    addResult()  
}  
  
//局部函数，函数引用  
fun lateAdd(a: Int, b: Int): Function0<Int> {  
    fun add(): Int {  
        return a + b  
    }  
    return ::add  
}
```

在 lateAdd 内部定义了一个局部函数，最后返回了该局部函数的引用，对结果使用()操作符拿到最终的结果，达到延迟计算的目的。

响应式编程

```
fun printUpperLetter(list: ArrayList<String>) {  
    list  
        .filter(fun(item): Boolean {  
            return item.isNotEmpty()  
        })  
        .filter { item -> item.isNotBlank() }  
        .filter {  
            item ->  
            if (item.isNullOrEmpty()) {  
                return@filter false  
            }  
            return@filter item.matches(Regex("^([a-z|A-Z])$"))  
        }  
        .filter { it.isLetter() }  
        .map(String::toUpperCase)  
        .sortedBy { it }  
        .forEach { print("$it, ") }  
    println()  
}
```

基于以上函数式编程的特性，Kotlin 可以像 RxJava 一样很方便的进行响应式编程

JavaScript支持

Kotlin1.1版本正式加入了对JavaScript的支持，可以用Kotlin进行网页开发，并且Kotlin也支持了与JavaScript的相互操作。

将 Kotlin 代码编译为 Javascript 代码后会得到两个主要的文件：

Kotlin.js. 运行时和标准库。这部分代码只与 Kotlin 的版本有关而不会因为不同的应用而有所不同。

{module}.js。真正的应用代码。所有的应用代码最终都会编译成一个 JavaScript 文件并与模块的名字同名。

Kotlin与JavaScript互操作

在Kotlin代码中，如果想要调用JavaScript代码，基本上有两种方式：js()内联模式和头文件模式。

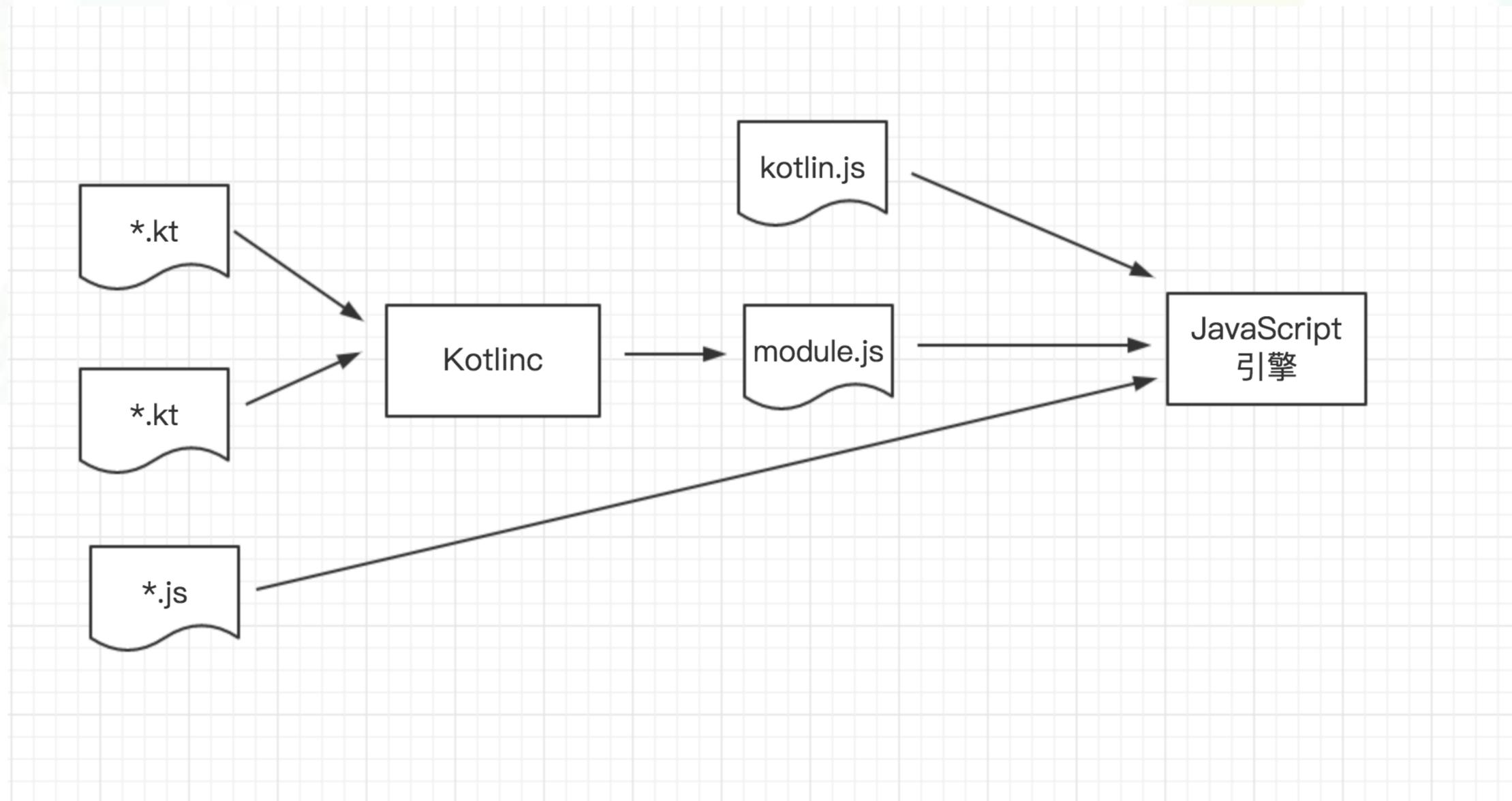
可以使用js("...")函数将一些JavaScript代码直接嵌入到Kotlin代码中。但是，有一点要求，js函数的参数必须是字符串常量。

```
fun main(args: Array<String>) {  
    val message = "Hello JavaScript"  
    js("console.log(message)")  
}
```

```
external修饰符定义头文件  
//hello.js  
function hello(message) {  
    console.log(message)  
}  
//Mian.kt  
external fun hello(message: String)  
fun main(args: Array<String>) {  
    val message = "Hello JavaScript"  
    hello(message)  
}
```

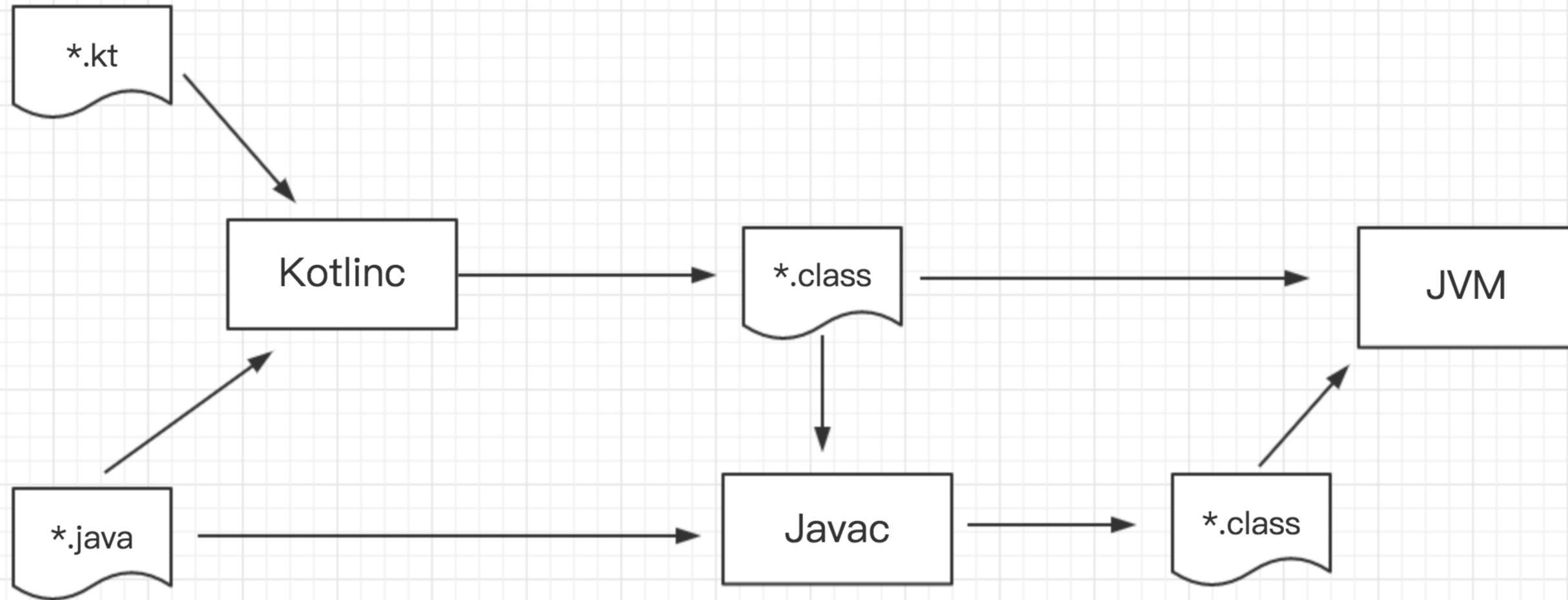
Kotlin与JavaScript互操作

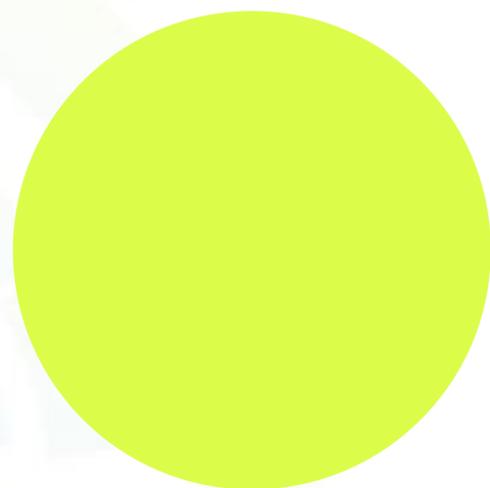
Kotlin-JavaScript模式中，Kotlinc(编译器)只是进行了转换JS的操作，然后与标准库kotlin.js、项目中JS文件一起再通过JavaScript引擎执行。



Kotlin与java互操作

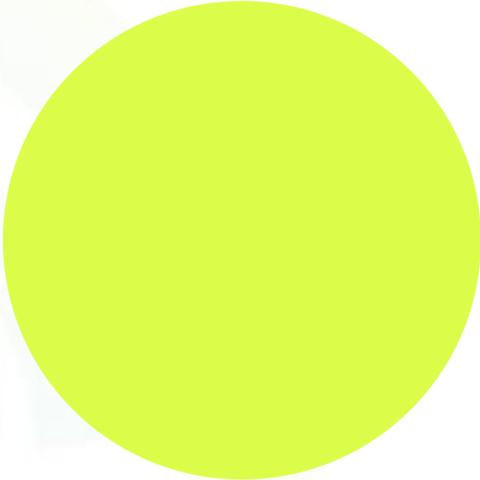
Kotlin-Java模式中，Kotlinc（编译器）将*.kt文件编译成了*.class字节码文件，同时*.java文件可通过Kotlinc或者Javac编译成*.class字节码文件，然后通过JVM虚拟机执行。





四、Kotlin的一些问题

- 在Kotlin1.0.0中，运行时+标准库的大小总共是210k+636k=846k
- 使用dex-method-counts统计出的方法数为6584，在包含Kotlin的应用中使用ProGuard可以一定程度控制其方法数。
- Kotlin目前缺乏代码静态检查工具
- Kotlin 肯定会减少项目中的代码行数，但是它也会提高代码在编译以后的方法数
- Kotlin 没有受检的异常(CE机制)
- Java编译比Kotlin快10%左右。
- 找出在应用开发过程中可能出现的问题的答案会比较困难



五、现有项目转化为kotlin项目的思路

环境搭建

- AS 3.0 直接支持kotlin的开发
- AS 2.3及其以下版本如何配置：

1. 安装Kotlin插件, Android Studio->Preferences->Plugins->Kotlin->Install->Restart AS

2. 添加Kotlin配置

a. SRC\gradle.properties添加配置kotlin_version=1.1.2-5

b. SRC\build.gradle

1. 新增仓库配置: `maven { url 'https://dl.google.com/dl/android/maven2/' }`

2. 新增依赖: `classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"`

c. SRC\app\build.gradle 导入Kotlin相关插件

`apply plugin: 'kotlin-android'`

`apply plugin: 'kotlin-android-extensions'`

d. SRC\config\dependencies.gradle 新增依赖 `compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"`

e. 源码路径java同目录新建kotlin源码文件夹: SRC\app\src\main\kotlin

f. SRC\config\qunar_build_config.gradle 设置代码路径集合sourceSets

android {

```
    sourceSets {
```

```
        main {
```

```
            java.srcDirs = ['src/main/java', 'src/main/kotlin']
```

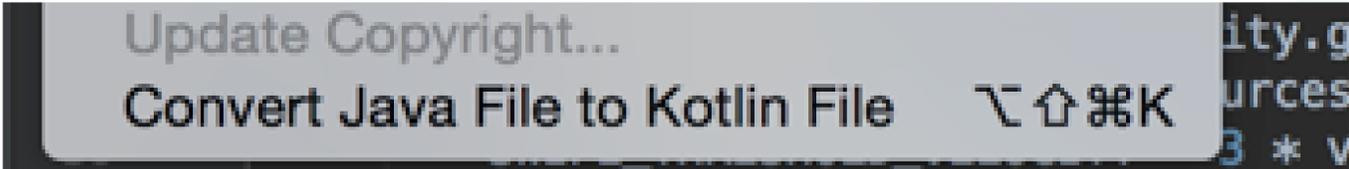
```
        }
```

```
    }
```

```
}
```

直接将java项目转化遇到的问题

安装Kotlin插件后，可以在code->



Update Copyright...
Convert Java File to Kotlin File ⌘⇧K

进行切换

- kotlin不支持在条件里面包含赋值语句，while语句很多需要重写
- 需要处理java中没有初始化的变量的问题
- 递归依赖的jar包需要去掉一些生成物
- 编译依赖kotlin库的项目也需要依赖jdk 1.8

总结回顾

- Kotlin可以提高开发效率，减少代码量
- 作为JAVA的超集，完全的兼容可以使转化无风险
- 建议原有的java代码不变，新代码用kotlin尝试

Q&A