一、准备

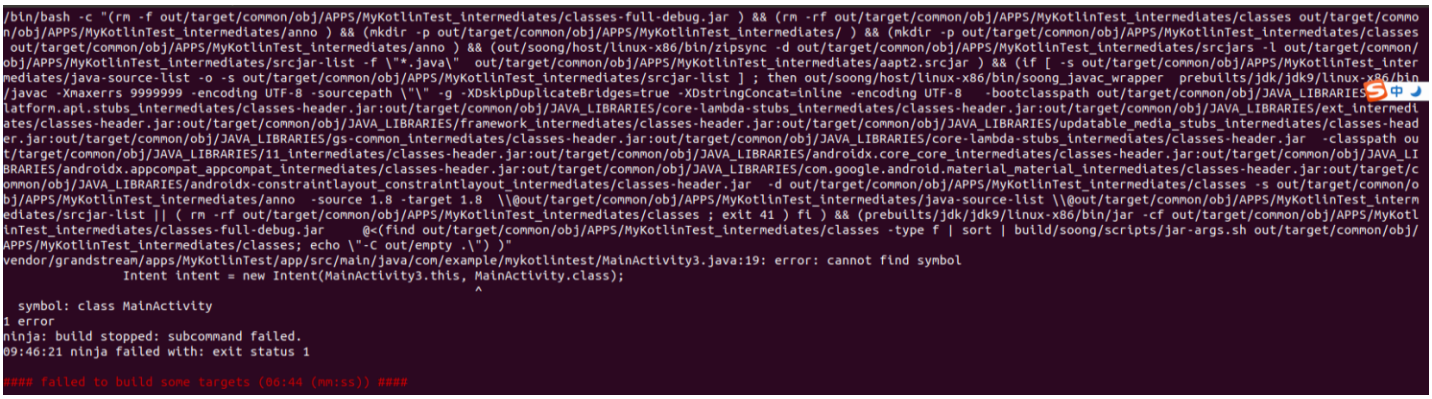
在Android Studio中New-->Activity-->XX Activity-->Source Language-->Kotlin，同理其余的Fragment，class等都有此选项。

使用kt的第一前提肯定是和java代码自然过渡使用，因此我尝试跳转到Kotlin的Activity

```
Intent intent = new Intent(MainActivity3.this, KotlinActivity.class);
startActivity(intent);
```

mmp编译一下，此时会报错KotlinActivity.class symbol can not find，项目中编译不了.kt文件。

```
vendor/grandstream/apps/MyKotlinTest/app/src/main/java/com/example/mykotlintest/MainActivity3.java:
                Intent intent = new Intent(MainActivity3.this, MainActivity.class);
                                                               ^
  symbol: class MainActivity
1 error
ninja: build stopped: subcommand failed.
09:46:21 ninja failed with: exit status 1
#### failed to build some targets (06:44 (mm:ss)) ####
```



方法一、Android.bp中 srcs: ["app/src/main/java/**/*.java"] + ["app/src/main/java/**/*.kt"]

方法二、Android.mk添加LOCAL_SRC_FILES:= $(call all-kt-files-under,app/src/main/java)，此外打开android/build/core/definitions.mk ，

```
//找到该代码块
define all-java-files-under
$(call all-named-files-under,*.java,$(1))
endef
//在下面添加
define all-kt-files-under
$(call all-named-files-under,*.kt,$(1))
endef
```

编译通过就可以开始自由发挥啦

二、语法、特性

1、不再需要逗号作为语句分割符，比如

```
package com.example.mykotlintest
```

```
import androidx.appcompat.app.AppCompatActivity
```

2、变量

val：不可变变量

var：可变变量

```
var a: Int = 1
//自动推断类型
var a = 1
```

尽可能使用val，java中定义不可变的常量使用final，kotlin中使用val，他们有一定的相似性。详细阅读 https://blog.csdn.net/willway_wang/article/details/100388344 我们得出kotlin中的val不等于java的final

```
//初始化数组
var b= arrayOfNulls<String>(1)
//emptyArray<String>等同于size为0的arrayOfNulls
var a=emptyArray<String>()
var c = Array(10) { k -> 2 * k } //后面代码块为逻辑函数表示给数组赋值第k个值为2*k
//数组里的内容为024681012141618
```

list集合、set集合、map集合，类似java的初始化赋值如下

```
var a = ArrayList<String>()
a[0] = ("A")
a.add("B")
```

list集合、set集合、map集合内置函数

可变集合：listOf，setOf，mapOf

不可变集合：mutableListOf，mutableSetOf，mutableMapOf

```
var a = listOf("A")
var a=mutableListOf("B")
var map = mapOf("apple" to 1)
var map = mutableMapOf("apple" to 1)
//集合的函数中提供了很多函数式API，和react有了相似之处，除了map函数遍历还有filter函数过滤等等
//甚至可以在初始化的时候做一些逻辑操作，{}代码块这种写法看完后续高阶函数之内联函数就能明白了
val a=listOf("a")
val b=a.map{ it.toUpperCase() } //此时b就是内容为"A"的集合
```

Pair、Triple，二元，三元元祖类型

```
val a = Pair("A", "B")
print(a.first+a.second)
//其实mapOf内部就是接受的Pair
var map = mapOf(Pair("A", "1"))

val b = Triple("A", "B", "C")
print(b.first+b.second+b.third)
```

3、区间

```
//双端闭区间，数学上代表[0,2]，表达了0，1，2，
var a=0..2
//左闭右开区间,数学上代表[0,2)，表达0，1
```

```
var a=0 until 2
//降序，以上两个区间仅仅只能表达升序，即0到2，不能时2到0，这个时候就使用downTo，
//数学上代表了[2,0]，表达了2，1，0
var a=2 downTo 0
```

4、for循环

```
for(i in a){}
//想要跳过一些元素使用step
for(i in 0 until 10 step 2)//kotlin中用step表达递增2运行
```

5、条件控制

```
一般方法if else
var x = 1;
if (x == 2) {
    x = 3
} else if (x == 3) {
    x = 4
} else {
    x = 5
}
 也可以这么写
 x = if (x == 2) {
     3
 } else if (x == 3) {
     4
 } else {
     5
 }
 最后也可简化
 x = if (x == 2) 3 else if (x == 3) 4 else 5
```

曾经的switch case被when取代了

```
var j = 1;
when (j) {
    1 -> {print(1)} //{} 只有一行代码时，{}可以省略
    2 -> print(2)
    else -> print(3)
}
//when也可以取代if-else
when{
    j==1->print(1)
     j==2->print(2)
     else->print(3)
 }
```

好了，至此会了if-else和for，我们就可以开开心心得在代码里面大杀四方了。

6、函数

函数可见修饰符

java中public，private，protected（当前类，子类，同一包路径下可见），default（默认）

kotlin中public（默认），private，protected（当前类，子类），internal（同一模块的类可见）

```
//java
private int initView(int a){
    return a;
}
//kotlin, Unit==void,fun a():Unit{} 相当于fun initView(){}
fun initView(a: Int): Int {
    return a
}
//自动推断返回值类型
 fun initView(a: Int) = a
 //参数指定默认值 变成可选参数,调用的时候 a()，则aa的值为1;调用a(2)，则aa的值为2
 fun a(aa: Int = 1) {}
```

字符串内敛表达式

```
val a="good"
println("Today is $a")
```

7、类和构造函数

```
//主构造函数跟在类名后面，constructor可以省略，参数没有也可以省略，()也可以省略
class MyClass constructor(name: String, age: Int){
    //主构造的方法体，前缀的初始化块
    init {}
    //次构造函数，无论如何都会先执行主构造函数init里面的代码
    //当一个类既有主构造函数又有次构造函数时，所有次构造函数必须【调用或间接调用】主构造函数
    constructor(name: String):this(name, 1){} //直接调用主构造函数

    constructor():this("1"){}//this调用了带一个参数的次构造函数(它已经调用了主构造函数)
     //所以该无参构造函数间接调用主构造函数
}
```

《Effective Java》书中有提到过，如果一个类并非是专门为继承而设计的，那么我们就应该将它声明成final，使其不可被继承。而在Kotlin当中，一个类默认就是不可被继承的，除非我们主动将它声明成open。kotlin中的所有类都是继承自Any，默认所有类都是final的

```
//java中类继承
class father {
    void man() {}
}
class son extends father {
    @Override
    void man() {
        super.man();
```

```
    }
}
//kotlin 中需要把一般已经省略的final 换成open
open class father {
    open fun man() {}
}
//father()代表引用son引用父类的无参构造函数
class son : father() {
    override fun man() {
        super.man()
    }
}
//根据继承特性，子类的构造函数必须调用父类的构造函数
open class Person (val name: String,val age: Int){}
//这里name 和 age不加任何关键字，这样可以让他的作用域限定在主构造函数之中
class Student(val sno:String, val grade: Int, name: String, age: Int ): Person( name, age)
```

8、属性的get/set

记得JavaBean中的大量get/set操作吗，再见了

```
inner class Stu {
    var a = 1
}
@Test
fun z() {
    val s = Stu()
    s.a = 2
    print(s.a)
}
//kotlin中已经默认实现了get，set。但是如果想在里面做一些逻辑当然越是可以的
inner class Stu {
    var a = 1
        get() = field.plus(2)//field指代a
        set(value) {
            field = value.plus(1)
        }
}
```

9、数据类，关键字data

```
data class User(val name: String, val age: Int)
val jack = User("a", 1)
print("${jack.name},,,,${jack.age}")
//居然还可以解构，和react太相似了把
val (name, age) = jack
print("$name,$age")
```

10、委托

类委托：将一个类的实现委托给另一个类去实现。类委托的意义在于大部分实现方法调用辅助对象中的方法，少部分自己重写

```
//mySet类实现了Set的所有方法，这些方法我们都是委托给Set实现的
class MySet<T>(val helper: HashSet<T>) : Set<T> {
    fun adc()=print("1")
    override val size: Int
        get() = helper.size
    override fun contains(element: T) = helper.contains(element)
    override fun containsAll(elements: Collection<T>) = helper.containsAll(elements)
    override fun isEmpty() = helper.isEmpty()
    override fun iterator() = helper.iterator()
}
//但是上面的写法会导致被委托的类有多少方法我们就需要重写多少个方法
//我们在接口声明后面使用by关键字，只需要增加，或者单独重写我们需要的方法就行了
inner class MySet<T>(val helper: HashSet<T>) : Set<T> by helper {
    fun adc() = print("1")
    override fun isEmpty() = false
}
```

委托属性：将一个属性的具体实现委托给一个类去完成

```
inner class MyClass {
    var p by Delegate()
}
inner class Delegate {
    private var a: Any? = null
    operator fun getValue(myClass: ExampleUnitTest.MyClass, property: KProperty<*>): Any? {
        return a
    }
    operator fun setValue(myClass: MyClass, property: KProperty<*>, any: Any?) {
        this.a = any
    }
}
@Test
fun def() {
    val a = MyClass()
    //实际调用Delege的setValue
    a.p = 6
    //实际调用Delege的getValue
    print(a.p)
}
//还有一种方法可以不使用Delegate的setValue，那就是将属性声明为val
//那就意味这p是无法在被声明之后被赋值，所以不需要实现setValue
//懒加载val
@Test
fun byLazy() {
```

```
    println("开始")
    println(yu)
}
val yu by lazy {
    println("lazy打印")
}
//开始
//lazy打印
//kotlin.Unit
//上面输出证明，只有当yu变量首次被用到时bylazy里面的代码才会被执行到
```

11、标准函数，with，run，apply

```
val list = listOf("Apple","Banana")
val builder = StringBuilder()
for (fruit in list) {
    builder.append(fruit).append("/n")
}
val result = builder.toString()
print(result)
//上述代码多次调用了builder对象，怎么能使其更加简洁呢，我们可以用with
val result = with (StringBuilder()){
    for(fruit in list) {
        append(fruit).append("\n")
    }
    toString()//with最后一行会作为返回值返回
}
print(result)
//run函数的作用和with差不多，但是run不能直接调用，必须要调用某个对象的run函数才行
val result = StringBuilder().run() {
    for (fruit in list ){
        append(fruit).append("/n")
    }
    toString()//返回值
}
print(result)
//apply函数也需要在某个对象上调用，但是无法指定返回值
val result = StringBuilder().apply {
    for (fruit in list) {
        append(fruit).append("\n")
    }
}
print (result.toString())
```

12、静态方法，kotlin中并没有直接的静态方法

java中在方法上声明一个static，调用静态方法不需要创建实例，适合编写工具类的功能。

kotlin中一般使用单例类来实现工具类，关键字object。

```
//单例类会使整个类都变成类似静态方法的调用方式
Object Util{
    fun doAction(){}
}
//只希望类中的某个函数变成静态调用方式
//该方法是在Util内部创建一个伴生类，只有kotlin会保证Util只有一个伴生对象的存在
class Util{
    //companion object生成伴生对象
    compaion object{
         fun doAction(){}
     }
 }
 //上述两种方法在java代码中用静态方法的形式是调用不到的
```

真正的静态方法，注解和顶层方法

1、我们给单例类或者compain object的方法加上@JvmStatic注解，那么kotlin编译器会把这些方法编译成真正的静态方法

2、顶层方法，顶层方法是指没有定义在任何类中的方法，kotlin会将所有顶层方法编译成静态方法

13、延迟初始化

```
//将d设置为全局变量
private var d: String? = null
d = "1"
//d必须加上判空，才能编译通过
d?.length
//lateinit关键字可以延迟初始化。它会告诉kotlin编译器我会晚些时候初始化，
//lateinit修饰符只能修饰不可空类型
//并且不允许修饰基础类型（四类八种基础类型int， double，boolean等）
private lateinit var d: String
 d = "1"
 d.length
 //延迟初始化之后，如果在没有初始化的时候调用会抛出UninitializedProppertyAccessException异常，
 //我们必须确保在被任何地方调用之前已经初始化了，并且也存在着重复初始化的风险
 //kotlin中提供了判断是否初始化的方法
 private lateinit var d: String
 d = "1"
 if (::d.isInitialized) {
     d.length
 }
```

14、密封类，关键字sealed，表示受限的类继承结构，它是枚举类的扩展

15、扩展函数

```
//创建一个String.kt文件，文件名可以随意，在里面写一个顶层函数
```

```
fun String.xx(): String {
    return "xx"
}
print("hh".xx()) //输出xx
//扩展函数能让api简洁，java中api可以调用的就只有固定的方法，但是kotlin中可以向String中任意扩展
```

16、重载运算符，关键字operator

```
@Test
fun sum() {
    print((Y(1) + Y(2)).num)
}
inner class Y(var num: Int) {
    operator fun plus(obj: Y): Y {
        return Y(num + obj.num)
    }
}
 //在以上例子中operator和plus是固定不变的，plus代表+，它实际调用的函数是a.plus(b)
 a-b，a.minus(b)
 a*b，a.times(b)
 a/b，a.div(b)
 a in b，b.contains(a)
 ....还有很多
```

17、高阶函数，函数当成参数使用

```
fun plus(num1: Int, num2: Int) = (num1 + num2)
fun times(num1: Int, num2: Int) = num1 * num2
//定义高阶函数
fun aAndb(num1: Int, num2: Int, operator: (Int, Int) -> Int): Int {
    return operator(num1, num2)
}
print(aAndb(1, 2, ::plus));//输出3
print(aAndb(1, 2, ::times));//输出2
```

这个时候你会说好麻烦，还得定义plus，times，想两个函数名多痛苦啊，kotlin贴心地想到了把高阶函数定义成内联函数

内联函数，关键字inline

```
print(aAndb(1, 2) { a, b -> a + b });
print(aAndb(1, 2) { a, b -> a * b });
//实际上kotlin编译器最后将转换成java支持的语法结构,Lambda表达式被转换成了匿名类的实现方式
//每调用一次Lambda函数就会造成额外的内存和性能消耗。
//内联函数就是为了解决这个问题而来，要内联函数带的lambda不宜过大,否则会造成生产class文件过大
//内联函数的原理https://blog.csdn.net/a644388262/article/details/110095658
inline fun aAndb(num1: Int, num2: Int, operator: (Int, Int) -> Int): Int {
    return operator(num1, num2)
}
```

18、泛型实化：https://www.cnblogs.com/jian0110/p/10690483.html
java中泛型是不可以被实例化的，比如说T a=new T();不允许的

19、泛型的协变和逆变

20、infix函数：增加代码可读性
前面我们声明map函数是var map= mapOf("a" to "b")，为什么可以使用to这种英语非常可读的方式呢，毕竟map函数参数是Pair，可以查看to的源码

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
//根据前面学习的扩展函数，我把这一句抄一下，并定义名称为h
infix fun <A, B> A.h(that: B): Pair<A, B> = Pair(this, that)
var map = mapOf("a" h "b")
//h()函数为自己定义的，功能和内置扩展函数to是一样的
```

21、协程：https://zhuanlan.zhihu.com/p/172471249。轻量级的线程。线程依靠操作系统的调度实现线程之间的切换，而协程可以在编程语言层面实现不同协程之间的切换。协程运行在线程之上，通过分时复用的方法运行多个协程

```
fun main() {
    //顶层协程，创建了一个协程作用域，所以在lunch{}中语句就是在协程执行的
    GlobalScope.launch {
        //delay能让当前协程延迟执行，它只能在协程的作用域或其他挂起函数调用。
        //Thread.sleep是阻塞当前线程
        delay(1000)
        print("1")
    }
}
//上述代码无任何输出，因为协程中的代码还没执行，main就执行完毕了，所以改造一下这段代码
fun main() {
    GlobalScope.launch {
        delay(1000)
        print("1")
    }
    Thread.sleep(4000)
}
//输出1
```

上述程序结束，协程代码块未执行完则会被强制中断，靠我们手动去等待GlobalScope里面的协程执行完毕是不靠谱的。
runBlocking函数可以让协程代码块执行完毕再结束应用

```
fun main() {
    //创建了一个协程作用域，它可以保证作用域内所有代码和子协程在没有执行完之前一直阻塞当前线程
    runBlocking {
        delay(1000)
        print("1")
    }
}
```

```
8 //完美输出1程序运行结束
```

创建多个协程launch里面创建的是子协程，子协程的特点就是外层作用域结束，该作用域下的子协程也会结束

```
1  fun main() {
2      runBlocking {
3          launch {
4              print("2")
5          }
6          print("1")
7          launch {
8              print("3")
9          }
10     }
11 }
```

当在lunch中写了非常多的逻辑，我们需要将一部分代码提取出来封装成一个函数（这个函数里面创建了子协程），但是很明显提取出来的函数已经不在协程作用域之中了，

关键字suspend，它能给将一个函数变成挂起函数，但是无法为它提供作用域

coroutineScope函数可以在其它挂起函数中调用，它可以继承外部的协程作用域，它会一直阻塞当前协程，直到coroutineScope执行完毕

```
1  fun main() {
2      runBlocking {
3          a()
4          launch {
5              print("2")
6          }
7          print("1")
8          launch {
9              print("3")
10         }
11     }
12 }
13 suspend fun a() = coroutineScope {
14     launch {
15         print("5")
16     }
17 }
18 //输出结构为5123
```

取消协程

```
1  fun main() {
2      val job= Job()
3      val scope= CoroutineScope(job)
4      scope.launch {}
5      //取消
6      scope.cancel()
7  }
```

创建协程并获取它的执行结果

```
//async会创建一个子协程并返回一个Deferred对象，await阻塞协程直到async函数有结果
fun main() {
    runBlocking {
        val result = async {
            print("1")
            delay(1000)
            2
        }.await()
        print(result)
         print("3")
    }
}
//输出123
//上述async、await的作用还有另一种写法
fun main() {
    runBlocking {
        //线程参数
        //Dispatchers.Default，低并发线程策略
        //Dispatchers.IO，高并发线程策略，网络请求
        //Dispatchers.Main，在主线程中执行
        val result = withContext(Dispatchers.Default) {
            delay(1000)
            print("1")
            1
        }
        print("2")
    }
}
```

三、Why Kotlin

1、简洁性

大大减少代码量

说再多不如实践上手操作，RecordingManagement中RecordingListActivity的代码数量为1016行。

通过code-->convert java file to kotlin file 得到后的文件为850行

编译不通过报错如下

```
e: /home/semirachenb/Documents/GAC2570/android/vendor/grandstream/apps/RecordingManagement/app/src/
Smart cast to 'ImageView!' is impossible, because 'sendIv' is a mutable property that could have be
```

遇到了问题，因为封装了findViewByIdT，所以控件实例都没有被很好地转换过来。既然如此就直接不用findviewById了。

注意以下关于kotlin-android-extensions的使用只需要留个印象就行了，不需要学会，因为它最近已经被废弃了，废弃的原由参见https://guolin.blog.csdn.net/article/details/113089706。不要慌，谷歌在AS4.1版本放弃之前，在AS3.6版本推出的ViewBinding可以解决我们的拮据。

```
//在project的build.gradle下
```

```
classpath"org.jetbrains.kotlin:kotlin-android-extensions:$kotlin_version"
```

```
//app的gradle下
apply plugin: 'kotlin-android-extensions'
//java 中给控件赋值
TextView tvName = findviewById(R.id.tv_name);
tvName.setText("快乐");
//通过kotlin的扩展，能直接使用id name，一键直达，再也没有重复重复的findViewbyId,不论代码的行数减少，
tv_name.text="快乐"
```

仅仅是去掉了findviewById这么一类的写法，得到的RecorderListActivity.kt为824行，小小的改动即获得二十几行的减少。

浏览改动后的kt文件，发现仍然有简化的空间

```
//java中点击事件的写法
export_tv.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {}
});
//转化得到的kotlin代码
export_tv.setOnClickListener(View.OnClickListener {return@OnClickListener})
//再简化
export_tv.setOnClickListener{}
```

根据语法学习中的知识修改原有的条件语句，没有改变逻辑，没有刻意删除注释的东西，代码变化为：

行数1016->800

字符数42812->32216

由此得出，kotlin拥有出色的简洁性。

2、安全

消除代码空引用的危险。在编译时期，检查空指针错误。kotlin中所有的参数和变量都默认不为空。

```
//java常规操作把a置为null，然后下次运行到这段代码的时候用a==null来判断是否是初次运行
//但这就给代码空指针错误提供了滋润的土壤
String a = null;
//kotlin不允许重新赋值为null，编译不通过
var a: String = null// 编译错误
//若确实需要赋值为null，也有方法
var b: String? = null
```

?操作符使得对象可以为空，但是为了在编译初期就处理掉所有的空指针异常，我们需要写许多对象的判空代码，为此kotlin提供了判空辅助工具以方便我们快速判空

?.操作符

```
if(a != null){
    a.xx=y
}
//kotlin中使用?.操作符，a为空则不执行a.xx=y
a?.xx=y
```

?:操作符

```
var c = if (a != null) {
    a
} else {
    b
}
var d = a ?: b
```

let 提供了函数式API编程接口，它能解决在某块区域中每次使用某对象都需要使用?进行非空判断

```
a?.let{aa ->
  a.xx=y
  a.zz=y
}
//再根据Lambbda语法特性简化
a?.let{
    it.xx=y
    it.zz=y
}
```

除了判空操作，还可以给变量写上默认值

val c = b.lenght ：1 //b为null，c为1

在此值得一提的是Kotlin取消了Checked Exception机制。大概解释就是不强制开发者使用try-catch去捕获一些毫无处的异常了，详情参阅https://guolin.blog.csdn.net/article/details/108817286。

3、互操作

Kotlin中调用Java代码，Java中调用Kotlin代码。通过上述简洁性的例子得知，Android studio中提供了把java一键翻译成Kotlin的工具，虽然翻译地很生硬但聊胜于无。很可惜，没有Kotlin转换成Java的工具，但是可以将Kotlin代码转换成Kotlin字节码，再通过反编译的方式还原成Java代码，这样获得的Java代码虽然无法直接编译，但是神形具备。具体方法：Tools-->Kotlin-->show Kotlin Bytecode，跳出一个字节码的窗口，点击左上角的DeCompile即得到反编译以后的java代码。