Rubik 1.8.0 使用手册

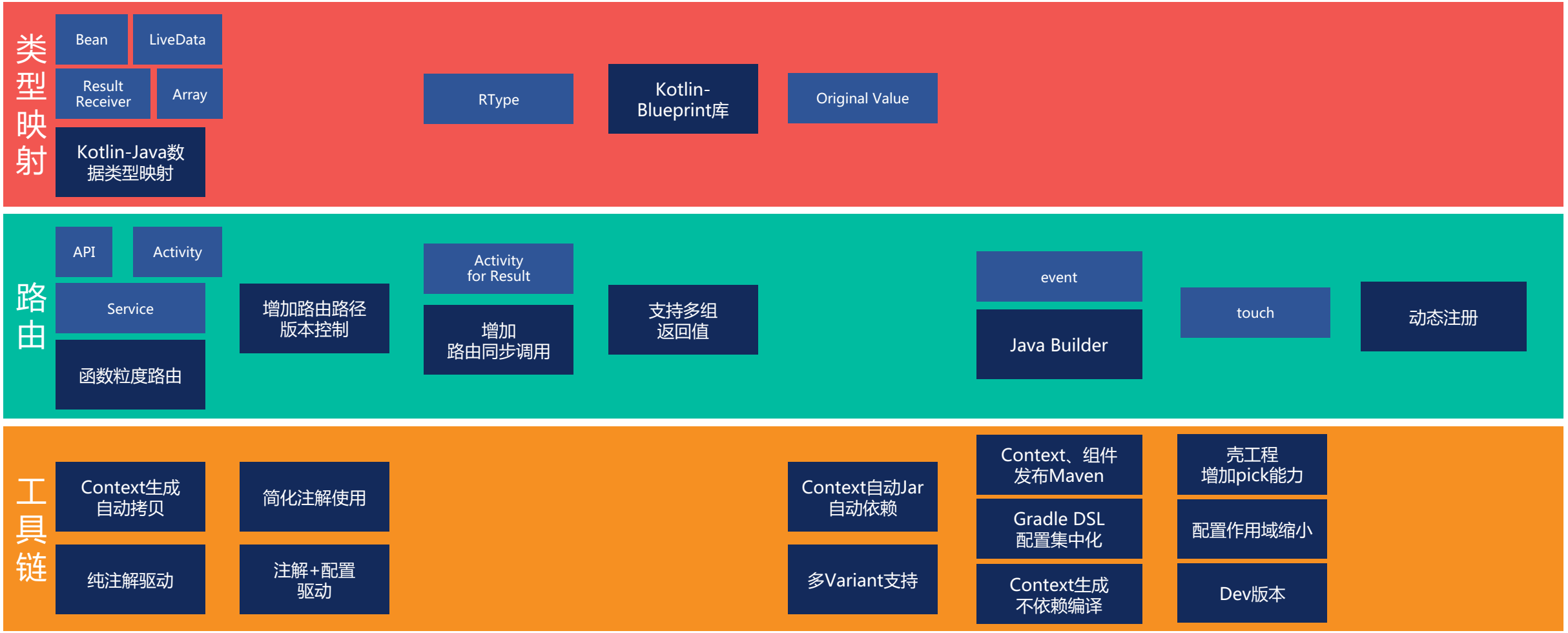
cuizhe01

目录

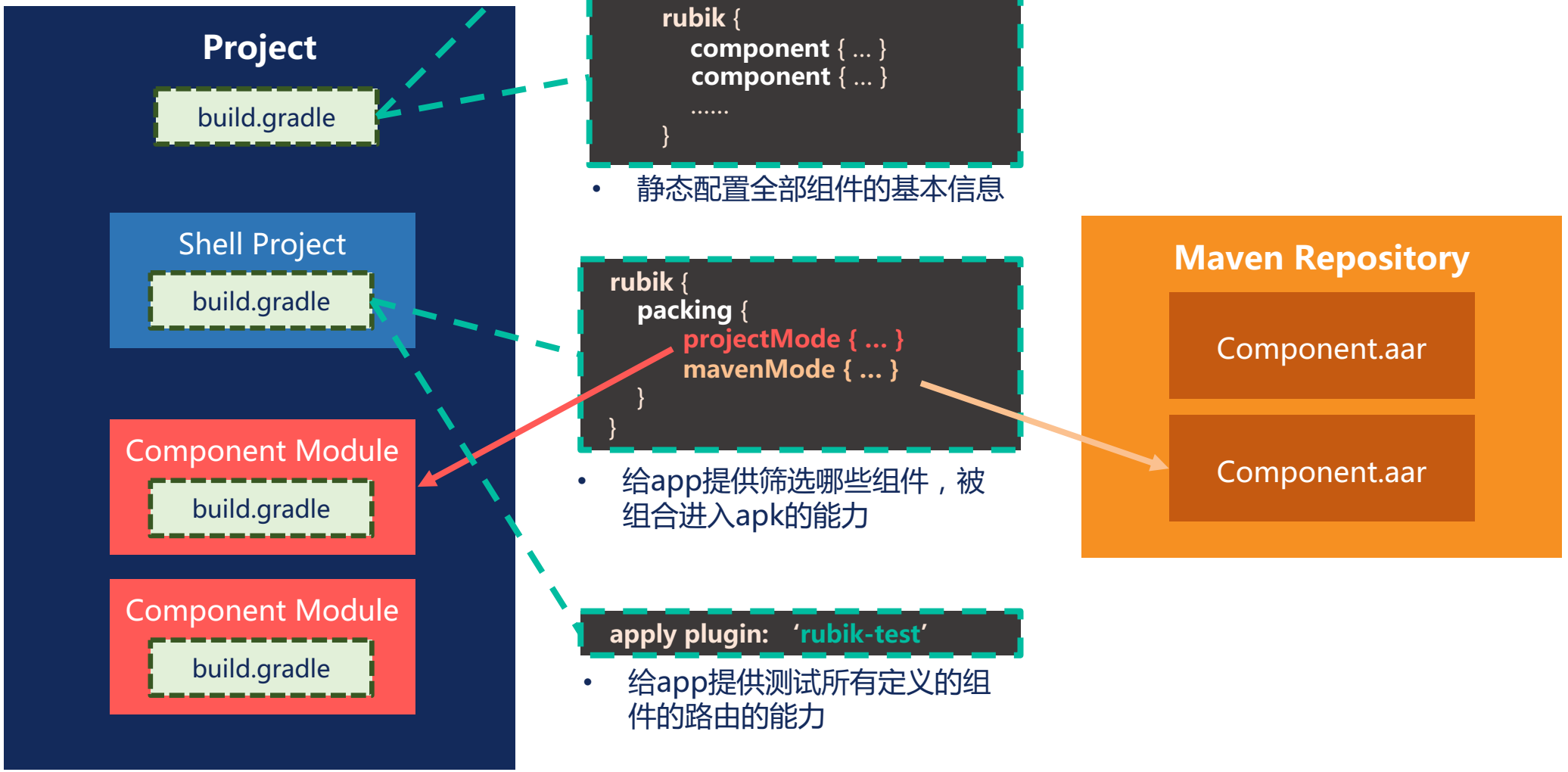
Rubik

Packing & Routeing.

Rubik版本迭代


类型映射
Bean
LiveData
Result Receiver
Array
Kotlin-Java数据类型映射
RType
Kotlin-Blueprint库
Original Value
路由
API
Activity
Service
函数粒度路由
增加路由路径版本控制
Activity for Result
增加路由同步调用
支持多组返回值
event
Java Builder
touch
动态注册
工具链
Context生成自动拷贝
纯注解驱动
简化注解使用
注解+配置驱动
Context自动Jar自动依赖
多Variant支持
Context、组件发布Maven
Gradle DSL配置集中化
Context生成不依赖编译
壳工程增加pick能力
配置作用域缩小
Dev版本
V1.1.0 （19年10月）
V1.2.0 （19年12月）
V1.3.0 （20年4月）
V1.4.0 （20年5月）
V1.5.0 （20年9月）
V1.6.0 （20年12月）
V1.7.0 （21年1月）
V1.8.0 （21年11月）
函数级路由、注解驱动、以KAPT为核心
任务驱动、以Gradle Plugin为核心
开源

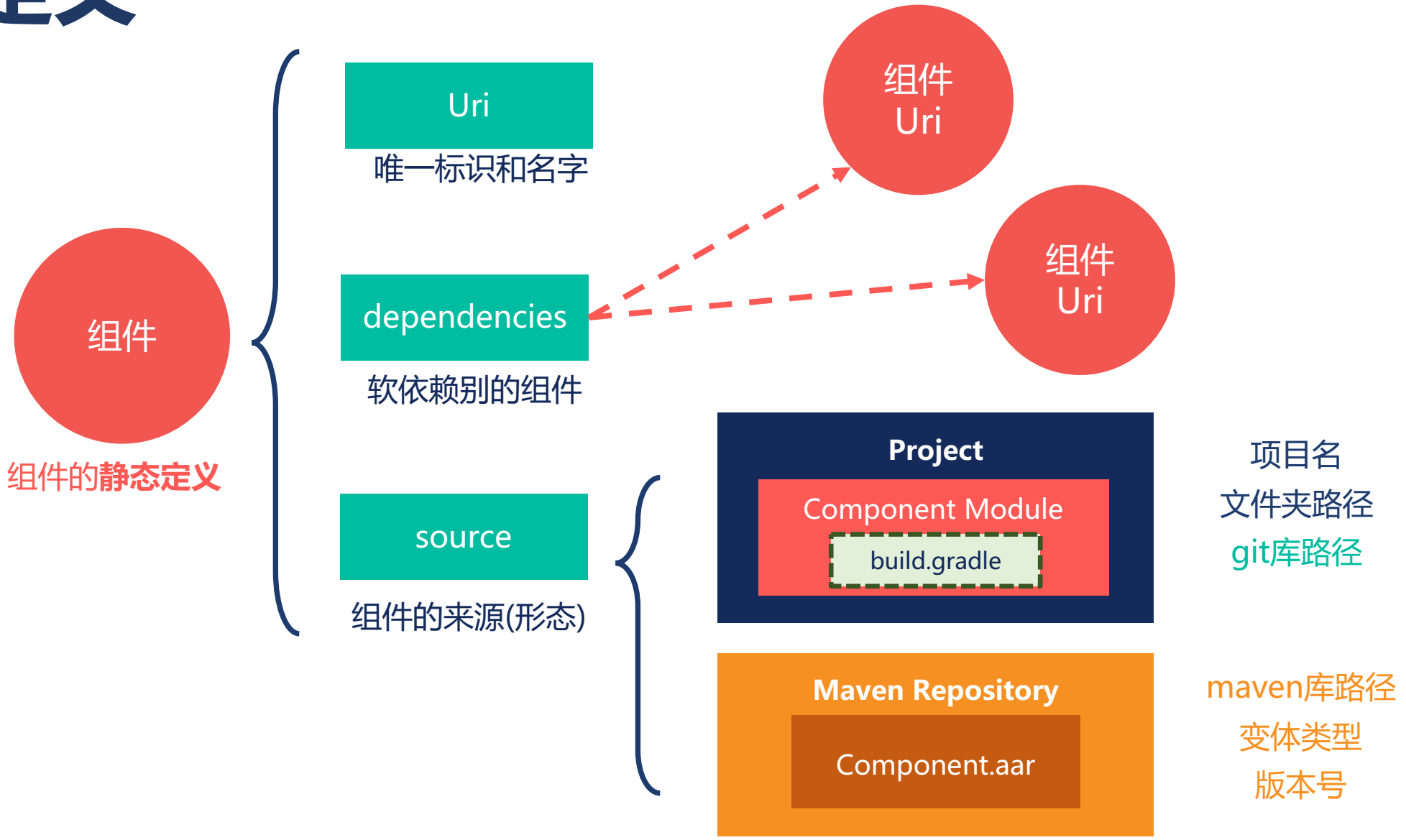
工程结构


Project
build.gradle
Shell Project
build.gradle
Component Module
build.gradle
Component Module
build.gradle
apply plugin: ‘rubik’
• 提供配置组件和参数的能力
rubik {
component { ... }
component { ... }
.....
}
• 静态配置全部组件的基本信息
rubik {
packing {
projectMode { ... }
mavenMode { ... }
}
}
• 给app提供筛选哪些组件，被组合进入apk的能力
apply plugin: ‘rubik-test’
• 给app提供测试所有定义的组件的路由的能力
Maven Repository
Component.aar
Component.aar

Rubik

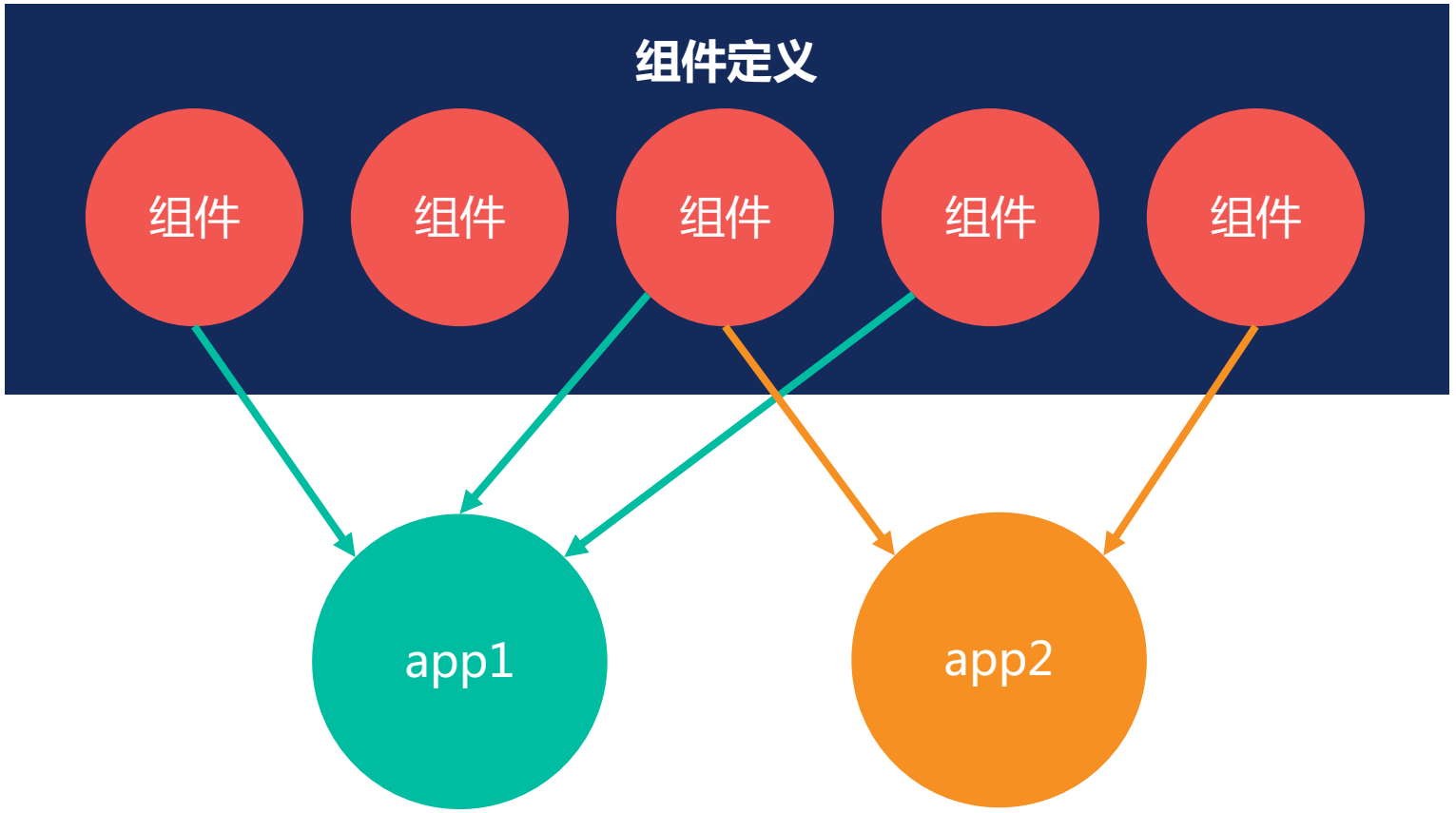
组件定义

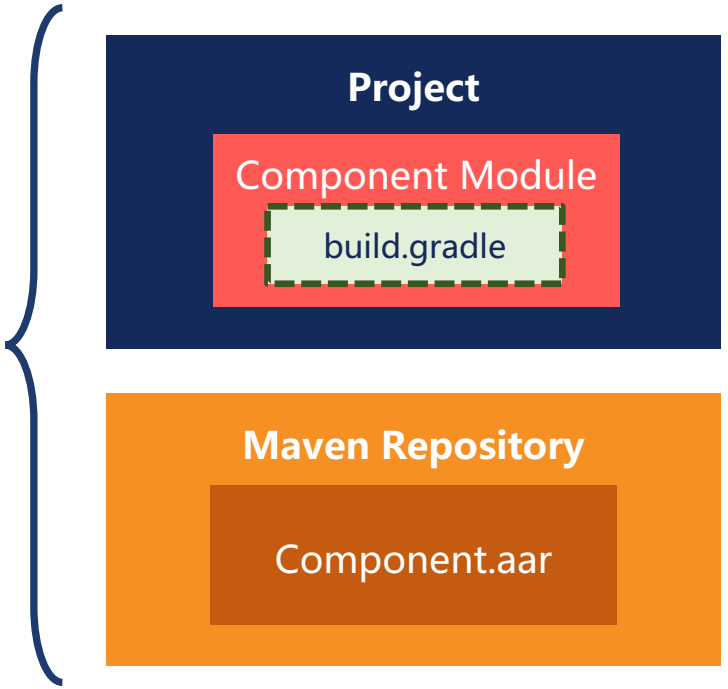
组件定义


组件
组件的静态定义
Uri
唯一标识和名字
dependencies
软依赖别的组件
source
组件的来源(形态)
组件
Uri
组件
Uri
Project
Component Module
build.gradle
项目名
文件夹路径
git库路径
Maven Repository
Component.aar
maven库路径
变体类型
版本号

筛选组件

对组件的**静态定义**





使用组件的哪个来源？

Project来源适合开发和debug

Maven来源适合加快编译速度

把组件组**组装(packing)**成不同的应用的**动态操作**

配置组件基本信息

- 在build.gradle中声明组件：
  - 最外层project
  - 组件所在module

uri：[scheme:][//authority]

```
rubik {
    component {
        uri "app://com.cloud-file"
        // uri 是组件的唯一 id，和路由根路径
        dependencies {    // 组件所依赖的其他组件uri
            uri ("app://com.local-file" )
            uri ("app://com.upload" )
        }
        source {    // 定义默认来源
            project (":lib-cloud-file")
            maven {   // 其他组件，通过maven依赖自己的默认版本
                version "0.2.0"
                variant "netdisk-english-debug"
            }
        }
    }
    component { ... }
}
```

壳工程选择需要组装的组件

- 在壳工程的build.gradle中：

```
rubik {
    packing {
        projcetMode {    //  projectMode()，通过子工程选择打包组件源码
            uri ("app://com.cloud-file")
            uri ("app://com.preview-*") {    //  支持通过*匹配任意字符
            }

            mavenMode {    // mavenMode()，通过maven依赖aar
                uri ("app://com.preview-file")
                uri ("app://com.download-file") {
                    version "0.2.0"  //  这里指定的version、 variant优先级高于组件自己定义的source
                    variant "netdisk-english-debug"
                }
            }
    }
```

- packing能力，用来筛选哪些组件以哪种方式组装进最终的apk

壳工程筛选组件方式

- PackingMode：
  - **projectMode**： 通过子工程选择打包组件源码
  - **mavenMode**： 通过maven选择打包组件aar
  - **noSourceMode**： rubik不会自动打包此组件，可以选择是通过反射或者不初始化组件路由

组件全局配置

- 在global块内全局管理一些定义：

```
rubik {
    global {
        maven {
            variant  "release" // 全局默认maven variant
            version ("app://com.preview", "0.5.1")   // 全局默认maven version
            version ("app://com. cloud-file", "0.5.2")
        }
    }

    component {
        uri "app://com.cloud-file"
        dependencies {
            uri ("app:// com.preview" )  //  可以省略版本号和variant
        }
        source { ... } //  可以省略版本号和variant
    }
     component { ... }
}
```

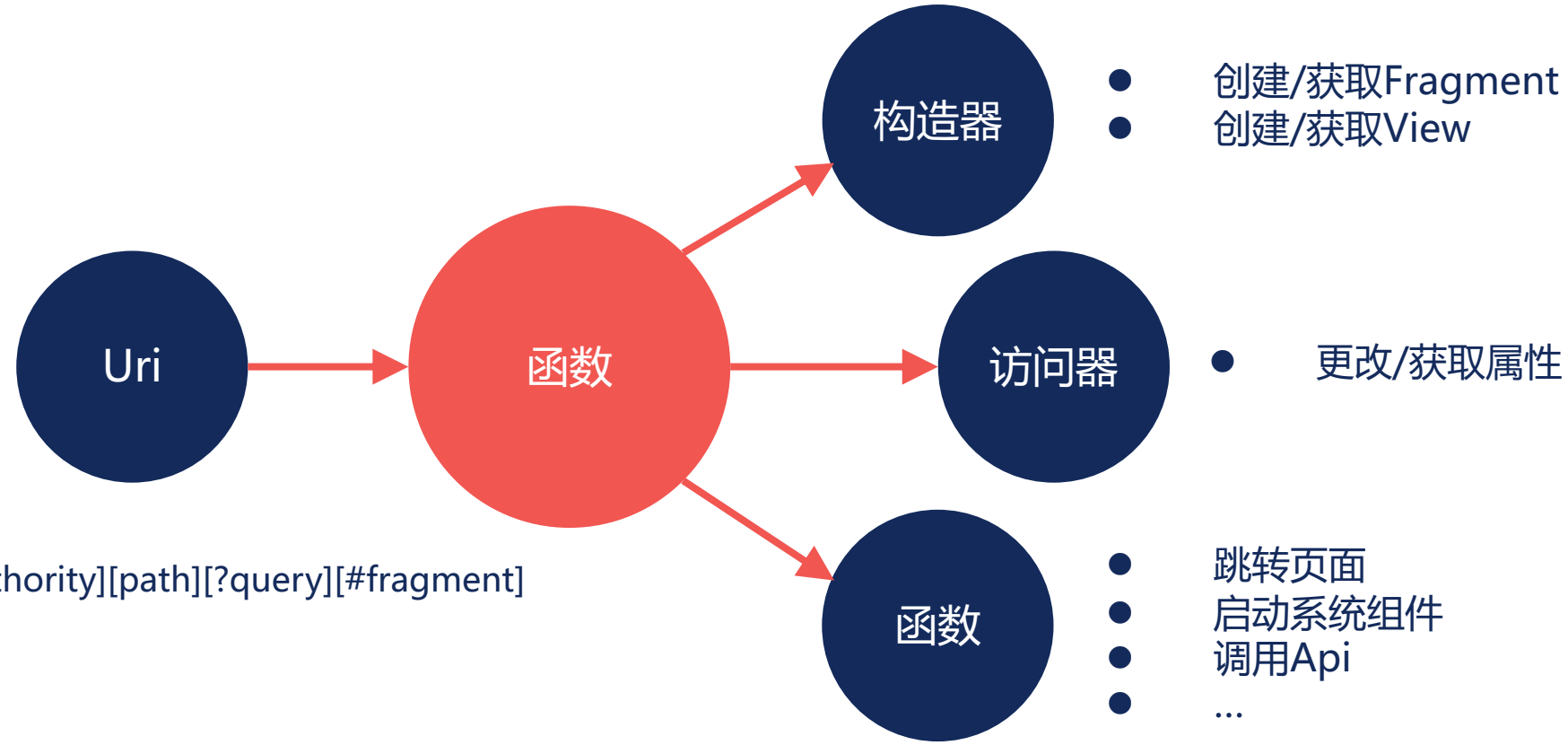
组件source的优先级

rubik.packing > compotent.tag > compotent > rubik.global

Rubik

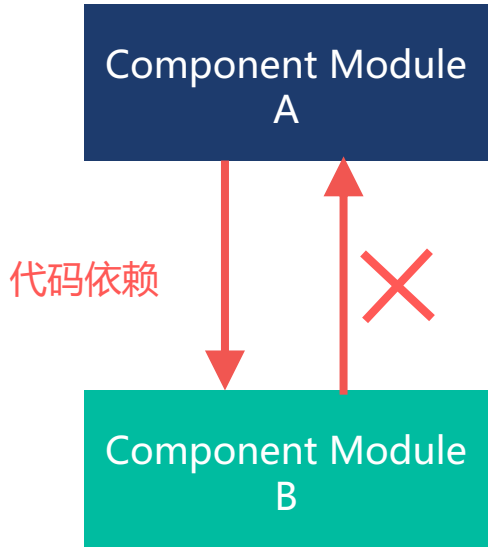
组件路由

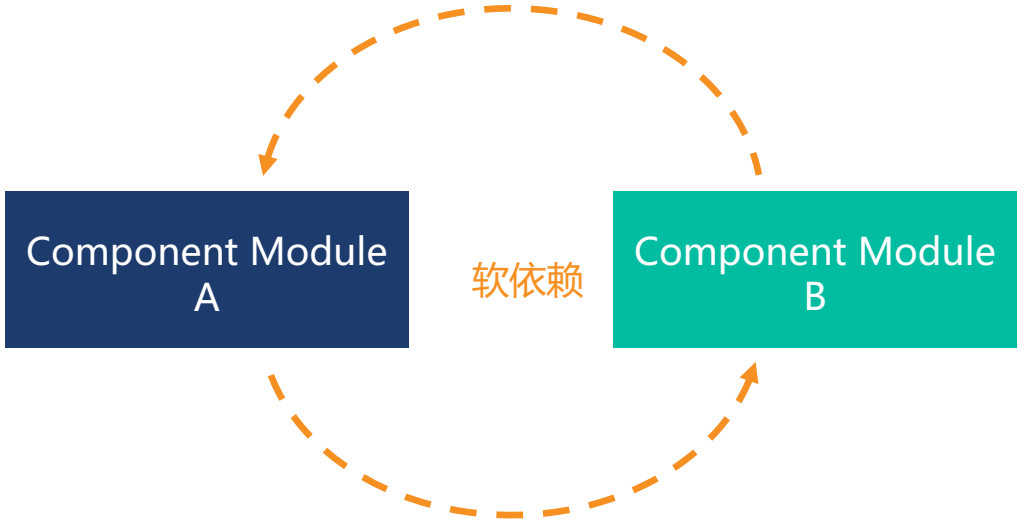
函数级路由



[scheme:][//authority][path][?query][#fragment]

软依赖





业务逻辑上有关连，但代码上没关联

函数路由

- 通过RRoute或者RFunction注解声明函数路由

- RRoute注解的target ：
  - 任意public方法(类、顶层、companion object、构造方法)
  - 任意public属性(类、顶层、companion object)
  - 任意public高阶函数(类、顶层、companion object)

```
@RRoute(path = "user")
fun getUser(id : Int, name : String) : User? {
        ...
}
```

路由调用

- DSL风格：
  - 基于uri完全解耦，不需要依赖任何代码

```
navigate {
        uri = "app://business.account/user"
        query {
                "id" with 400
                "name" with "zhangsan01"
        }
        result<User?> { user ->
        // 通过泛型指定接收数据类型，多次异步返回时，可以用多个result接收
                ...
        }
}
```

路由调用

- 函数风格：
  - 调用自动生成的XXContext类中的函数，可以约束参数和返回值类型，底层实现原理和DSL风格一致

```
AccountContext.user(400, "zhangsan01" ) { user ->
        ...
}
```

函数路由

哪些数据类型，可以在函数路由中作为参数或者返回值？

基于强制类型转换

- Kotlin、Java、Framework、第三方库等（运行环境及语言本身）提供类型
  - ✓ 基本数据类型、String
  - ✓ 语言自带的集合
  - ✓ Kotlin函数
  - ✓ LiveData
  - ✓ Cursor、ResultReceiver 、Context、Activity、Fragment、View 等
  - ✓ …
- 自己写的，提供者、调用者共同依赖（代码依赖）的类型
- 泛型是以上类型的以上类型

基于序列化

- Parcelable
- RValue
- 包含RValue的可序列化类型（集合、RValue多层嵌套）
- 泛型是RValue的LiveData
- 参数中有RValue的Kotlin函数（需要结合@RResult注解）

函数路由

- 通过resultType指定函数调用时的返回类型

```
class MyFragment
@RRoute(path = "your-fragment" , resultType = Fragment::class )
constructor() : Fragment {
        ...
}
```

```
RContext.yourFragment { value -> // 类型为Fragment，而不是MyFragment
        ...
}
```

自定义类型参数和返回值

- 通过RValue注解声明自定义Java Bean， 会自动生成一个结构一致的Java Bean，可以用作路由参数和返回值，可嵌套使用
- RValue注解的target ：
  - TYPE

```
@RValue
data class User (
        id : Int,
        name : String
)
```

函数路由

```
class Account(){
    @RRoute(path = "user")
    fun getUser(id : Int, name : String) : User {
        ...
    }
}
```

```
Account().user(id, name)
// 目标函数getUser为非静态函数，路由默认调用无参构造函数创建执行函数的对象
```

函数路由

- 目标函数为非静态函数时，可以通过RRouteInstance注解为函数路由提供函数所在类的实例，或调用其非默认构造函数获得实例
- RRouteInstance注解的target和RRoute注解保持一致，一个代码元素可以被多个RRouteInstance注解修饰

```
@RRouteInstance(forPath = "user")
@RRouteInstance (forPath = "admin")
fun provideAccount(dept : String) : Account {
        return Account(dept) // 通过非默认构造函数获得实例
        // return Utils.getAccount(dept) // 通过其他方式创建实例或获取单例
}
```

```
AccountContext.user("dept01" , 400, "zhangsan01" ) { user ->
        ...
} // assist的参数被整合进路由
```

异步或多次返回结果

- 通过RResult注解声明函数参数列表上的高阶函数或接口作为异步返回的回调

```
interface Callback{
        @RResult
        fun onCall(code : Int)
}
```

```
@RRoute(path = "do-sth-async")
fun getUser(
        @RResult firstStep : (Int) -> Unit ,
        @RResult secondStep : Callback
) {
        firstStep(101)
        secondStep(102)
}
```

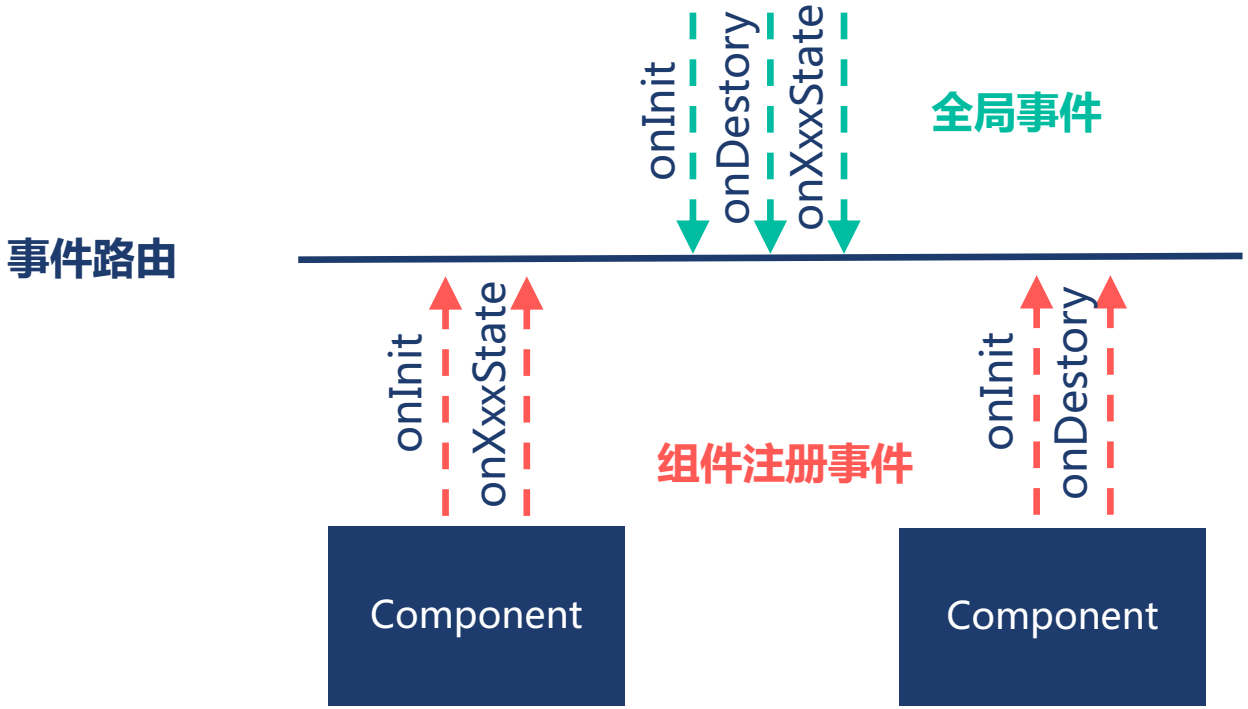
通过Uri携带参数

```
@RRoute(path = "user/{id}?name={name}&age={age}")
fun getUser(id : Int, name : String, age : String) : User? {
        ...
}
```

```
navigate {
        uri = "app://business.account/user/400?name=zhangsan01&age=23”
        // or
        uri = AccountContext.Uris.USER("400", "zhangsan01", "23")
}
```

事件路由

- 组件全局事件：比如应用声明周期变化、业务状态变化等全局的事件。



事件路由

- 事件函数

```
@REvent(msg = LifeEvent.INIT)
fun initAudioSdk(context : Context) {
        ...
}

@REvent(msg = LifeEvent.INIT)
fun initVideoSdk(context : Context) {
        ...
}
```

```
doEvent(LifeEvent.INIT)

// 执行所有组件msg为LifeEvent.INIT的函数
```

页面路由

- 通过@RRoute或者@RPage注解修饰Activity的子类
- 通过@RProperty注解指定启动Activity携带的参数

```
@RRoute(path = "home")
class HomeActivity : FragmentActivity() {

    @RProperty(name = "tabId")
    private val tabId : String
        get() = property("tabId") // 从Intent中获取指定参数

}
```

页面路由调用

- DSL风格：

```
navigate {
        uri = RContext.Uris.HOME
        query {
                flags = Intent.FLAG_ACTIVITY_SINGLE_TOP
                "tabId" with "home"
        }
}
```

页面路由调用

- 函数风格：

```
MainContext.home(10, Intent.FLAG_ACTIVITY_SINGLE_TOP )
```

Java语言支持

- Kotlin语言使用路由

```
PreviewContext.getView(context) { view ->
        ...
}
// Kotlin函数风格
```

```
navigate {
        uri = PreviewContext.GET_VIEW
        query {
                "context" with context
                "tag" with "tag"
        }
        result<View?> { view ->
                ...
        }
}
// DSL风格
```

- Java语言使用路由

```
PreviewContext.getView(context , view -> {
        ...
} );
// Java 8函数风格 @JvmStatic
```

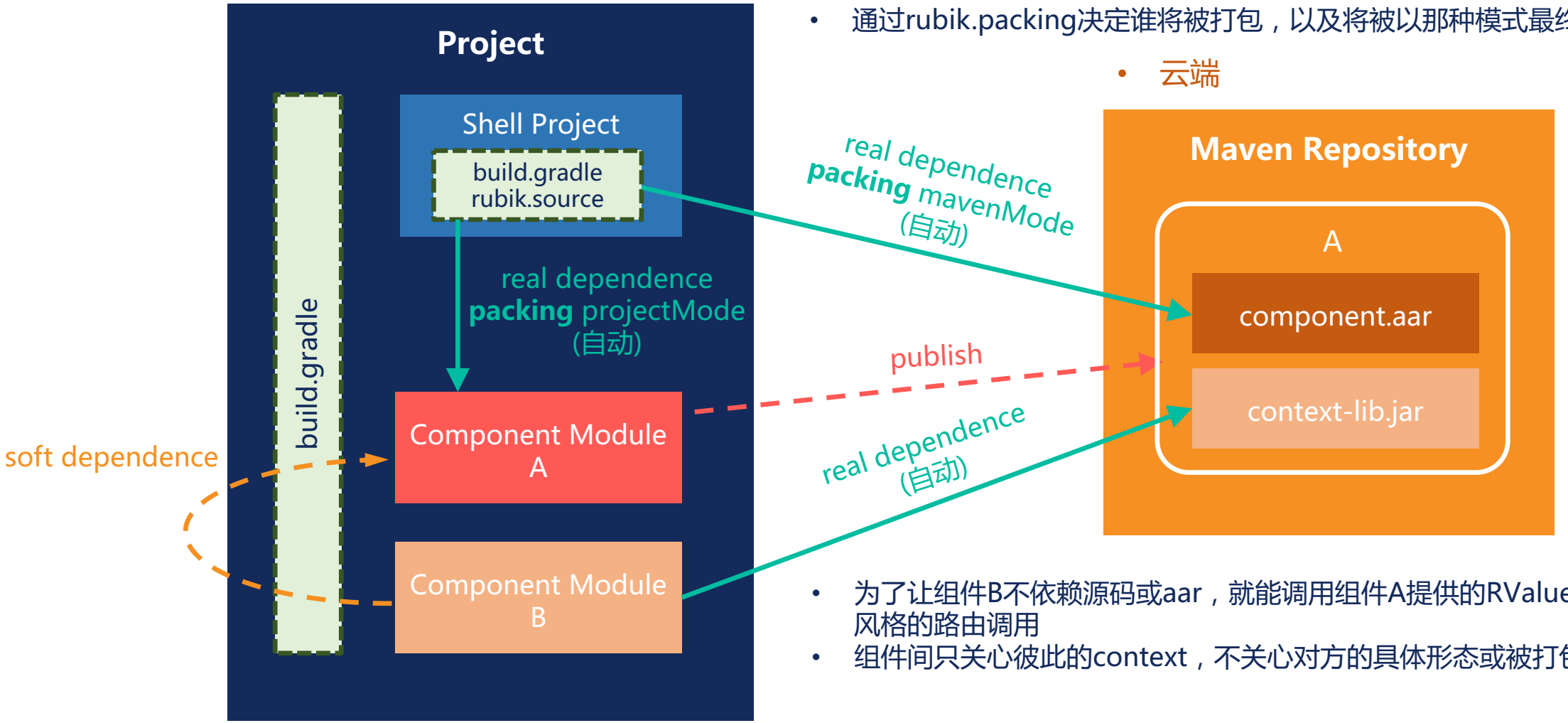
```
Router.builder()
        .uri(PreviewContext.GET_VIEW)
        .with("context", context)
        .with("tag", "tag")
        .result(View.class, view -> {
                ...
        }
        .build()
        .navigate();

// Builder风格
```

Rubik

组件发布

组件发布

- 本地



- 通过rubik.packing决定谁将被打包，以及将被以那种模式最终打包到apk里
- 云端

- 为了让组件B不依赖源码或aar，就能调用组件A提供的RValue，和函数风格的路由调用
- 组件间只关心彼此的context，不关心对方的具体形态或被打包与否

组件发布任务

- 完成组件t定义后，工具链会在所在module下，生成两组gradle task：

```
assembleRubik{组件名}RContextLib
// 打包对应组件的Context lib，输出到本地临时目录，本地调试时使用
publishRubik{组件名}RContextLibs
// 打包对应组件的Context lib，并按publishVersion发布到maven

publishRubik{组件名}{variant}RComponent
// 打包对应组件的源码，并按publishVersion发布到maven
```

- 工具链还会在gradle root project下生成两个gradle task：

```
assembleRubikAllRContextLibs
// 打包本项目下的全部Context lib，输出到本地临时目录，本地调试时使用
publishRubikAllRContextLibs
// 打包本项目下的全部Context lib，并按publishVersion发布到maven，如该
publishVersion已经在maven存在，则报错，但不影响其他组件的发布
```

配置组件发布版本号

- 在context.source.project中配置：

```
rubik {
    component {
        uri "app://com.cloud-file"
        dependencies { ... }
        source {
            project (":lib-cloud-file") {
                publishVersion "0.2.1"  //  组件发布版本号
            }
        }
    }
    component { ... }
}
```

组件publishVersion的优先级

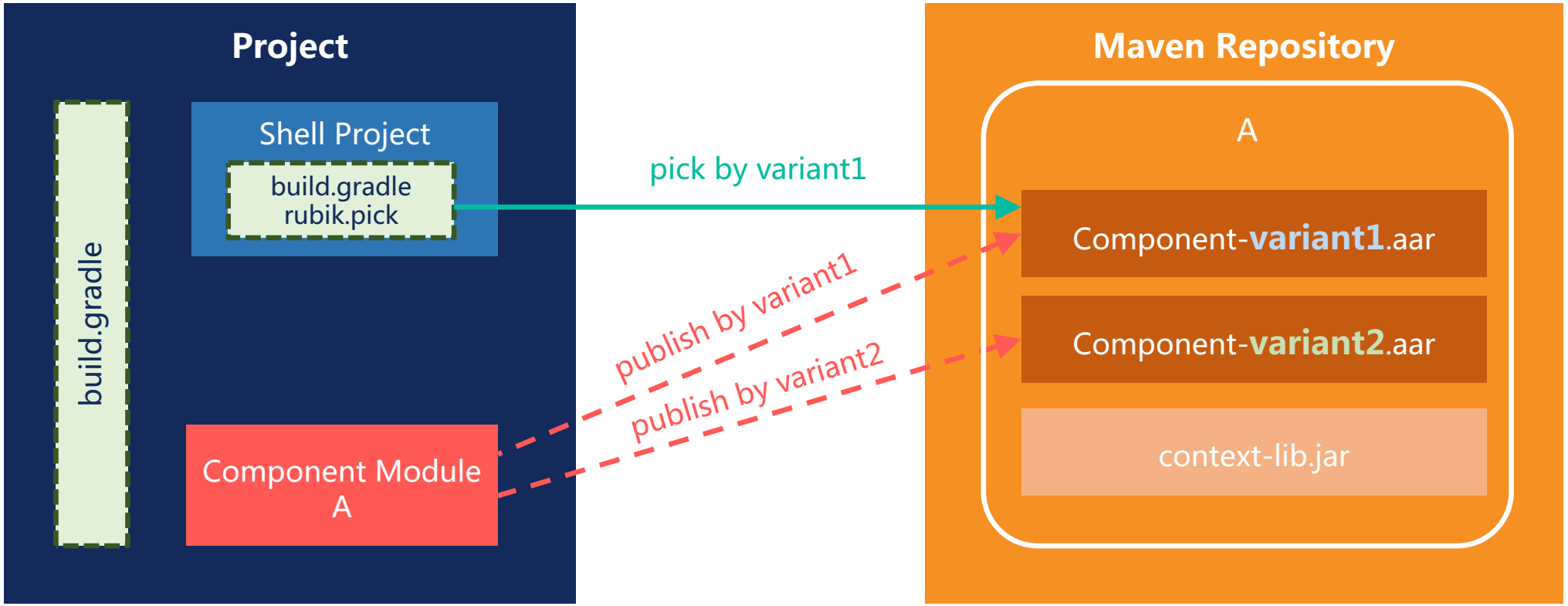
component.source.publishVersion > global.maven.version.publish

Rubik

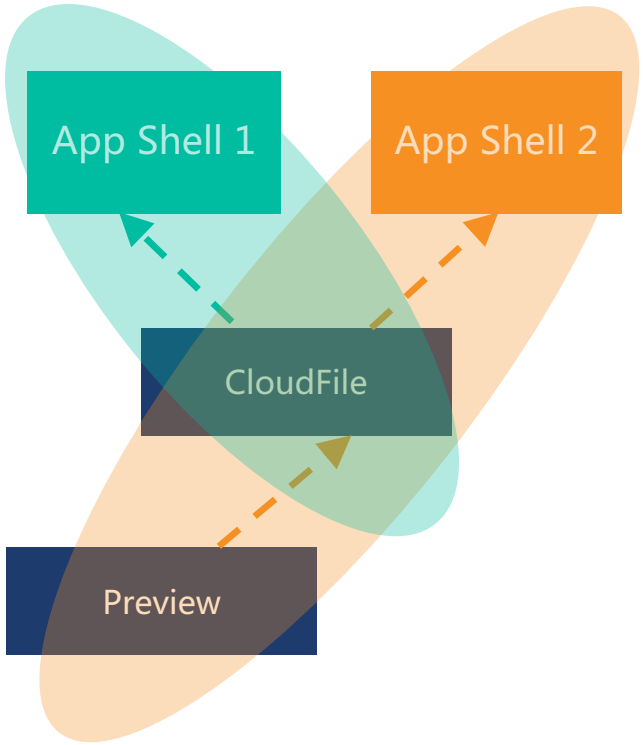
可变性管理

组件发布

- 本地
- 云端


Project
build.gradle
Shell Project
build.gradle
rubik.pick
Component Module
A
pick by variant1
publish by variant1
publish by variant2
Maven Repository
A
Component-variant1.aar
Component-variant2.aar
context-lib.jar

组件间touch能力



- 在CloudFile中，运行时检测Preview组件是否存在（被正确打包）。

```
PreviewContext.touch {
        // App Shell 2: 目标组件正常
        // 可以进行显示入口、初始化等操作
        PreviewContext().startPreviewActivity()
}.miss {
        //  App Shell 1: 目标组件没有被打包
        //  隐藏入口等异常处理操作
}

touch("app://com.preview") { ... }.miss { ... }
```

Rubik

支持单测

支持单测

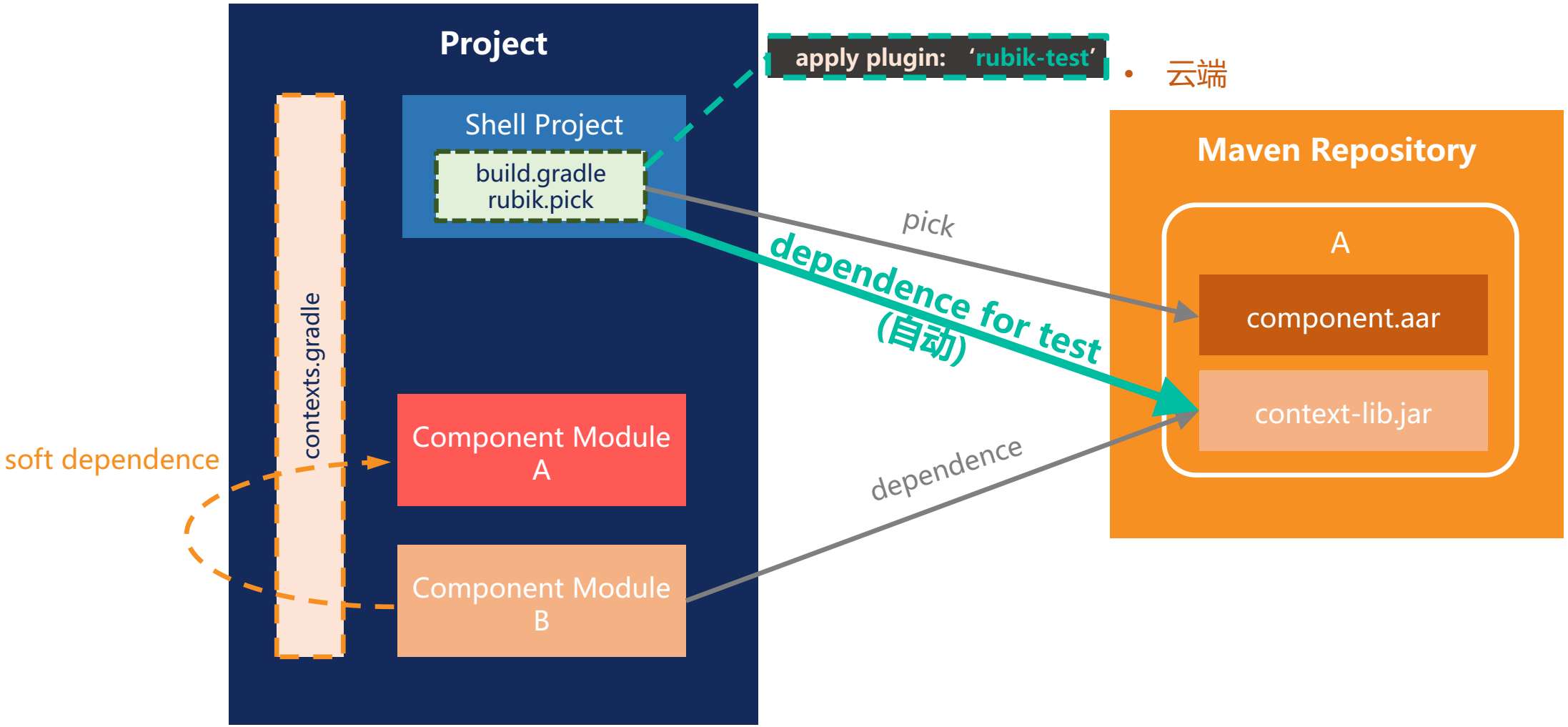
- rubik-test插件，给当前工程的androidTest variant添加全部可pick组件的context.jar依赖，便于写测试用例。

src/androidTest/java:

```
@RunWith(AndroidJUnit4::class)
class RouterTestCase {
    @Before
    fun init() {
        Rubik.init()
    } // 初始化Rubik
    @Test
    fun usePerview() {
        PerviewContext.preViewVideo(path : String) { success ->
            log("preViewVideo success:${success}")
        } // 测试用例
    }
    ... // 继续写测试用例
}
```

支持单测

- 本地


Project
apply plugin: 'rubik-test'
Shell Project
build.gradle
rubik.pick
contexts.gradle
Component Module A
Component Module B
soft dependence
pick
dependence for test (自动)
dependence
Maven Repository
A
component.aar
context-lib.jar

- 云端

Thanks